第 3 章

ROS 2 编程基础

ROS 2 为了成为各种机器人的通用操作系统，其本身不提供控制机器人的方法，而是提供一个运行机器人算法的框架和环境，使用各种各样的功能包实现具体的机器人功能。许多优秀的第三方功能包就是依托 ROS 2 提供的框架进行开发而形成的。ROS 2 提供了开发自定义程序的功能，可编写自定义的机器人程序，完成特定的机器人功能。为了方便使用者开发 ROS 2 的程序，ROS 2 使用了一个专用的构建工具——colcon。本章主要介绍使用 colcon 开发 ROS 2 功能包的流程和方法。

3.1 ROS 2 项目

编写 ROS 2 的项目涉及工作空间、功能包和节点程序等内容。工作空间存放了整个项目，一个工作空间可以拥有多个功能包，节点是功能包具体功能的实现。

3.1.1 工作空间

在编写 ROS 2 程序时，需要使用工作空间进行项目管理。ROS 2 工作空间是具有特定结构的文件夹，用于放置源代码、构建临时文件、编译结果和日志等文件和文件夹。一个典型的 ROS 2 工作空间主要包含以下几个子文件夹。

(1) src 文件夹：编写的 ROS 2 程序就放置在此文件夹内，ROS 2 将程序以功能包的形式进行组织，功能包是 ROS 2 中最小的可执行程序。src 文件夹内可以放置一个或多个 ROS 2 的包。在创建 ROS 2 工作空间时，需要手动创建 src 文件夹。

(2) build 文件夹：用于放置功能包编译过程中产生的中间文件。工作空间中的各个功能包都会在 build 文件夹内创建一个自己的子文件夹。build 文件夹是功能包编译过程中由 colcon 自行创建和维护的，不需要用户修改。当需要完全重新编译工作空间时，在编译前可手动删除该文件夹。

(3) install 文件夹：用于放置编译好的功能包，每个包的可运行程序都在 install 文件夹中拥有一个文件夹。install 文件夹也是由 colcon 创建和维护的，不需要用户修改。当需要完全重新编译工作空间时，在编译前可手动删除该文件夹。

(4) log 文件夹：用于放置 colcon 编译过程中产生的日志信息。一般在功能包编译出错时用于查看错误原因，以及调试程序。

由于 ROS 2 的工作空间本质上是一个文件夹，因此只需创建一个文件夹。在终端创建工作空间，命令如下：

```
mkdir -p ~/ros2_ws/src
```

以上使用 mkdir 的级联文件夹创建命令在用户的主目录下创建一个名为 ros2_ws 的文件夹，并在 ros2_ws 文件夹内再创建一个用于放置功能包的 src 目录。

创建完成后，进入工作空间 ros2_ws，并使用 VS Code 打开，命令如下：

```
cd ~/ros2_ws
code .
```

```
hw@hw-VirtualBox:~$ mkdir -p ~/ros2_ws/src
hw@hw-VirtualBox:~$ cd ~/ros2_ws
hw@hw-VirtualBox:~/ros2_ws$ code .
```

图 3-1　创建并打开工作空间

上述代码的运行效果如图 3-1 所示。经过上述步骤就完成了 ROS 2 工作空间的创建，随后就可以在该工作空间中创建、编译、安装和运行自定义的功能包，以实现机器人的各种功能。

3.1.2　创建功能包

功能包是实现自定义 ROS 2 功能的程序。ROS 2 支持使用 C++和 Python 两种语言编写功能包。C++功能包使用基于 CMake 构建系统的一种构建类型 ament_cmake，Python 功能包使用构建类型 ament_python。

ROS 2 提供了创建自定义 C++和 Python 功能包的命令 ros2 pkg create。下面的示例分别展示了两种自定义功能包的创建。

在 ROS 2 工作空间中，自定义功能包需要放置在 src 文件夹下，在创建功能包前需要先进入该文件夹，命令如下：

```
cd ~/ros2_ws/src
```

(1) 创建一个 C++功能包，命令如下：

```
ros2 pkg create --build-type ament_cmake --license MIT --node-name
test_node test_package_c
```

以上利用 ROS 2 提供的 ros2 pkg 命令下的 create 功能创建了一个名为 test_package_c 的功能包，参数--build-type 用于将功能包的编写语言指定为 C++，参数--license 用于将功能包的协议指定为 MIT，参数--node-name 表示在功能包下创建一个名为 test_node 的节点。

(2) 创建一个 Python 包，命令如下：

```
ros2 pkg create --build-type ament_python --license MIT --node-name
test_node test_package_python
```

以上利用 ROS 2 提供的 ROS 2 pkg 命令下的 create 功能创建了一个名为 test_package_python 的功能包，参数--build-type 用于将功能包的编程语言指定为 Python，其他参数与 C++包的含义相同。

上述两条命令的执行效果如图 3-2 所示。执行完成后，ROS 2 就会利用其内置的模板在工作空间内创建好两个功能包。

```
hw@hw-VirtualBox:~/ros2_ws/src$ ros2 pkg create --build-type ament_cmake --license MIT --node-name test_node test_package_c
going to create a new package
package name: test_package_c
destination directory: /home/hw/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['hw <hw@todo.todo>']
```

(a) 创建C++功能包

```
hw@hw-VirtualBox:~/ros2_ws/src$ ros2 pkg create --build-type ament_python --license MIT --node-name test_node test_package_python
going to create a new package
package name: test_package_python
destination directory: /home/hw/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['hw <hw@todo.todo>']
```

(b) 创建Python功能包

图 3-2　创建功能包

在 VS Code 中可以显示两个功能包的结构，如图 3-3 所示。在工作空间的 src 目录下分别放置两个功能包的同名文件夹 test_package_c 和 test_package_python。

图 3-3　功能包的结构

3.1.3　编写程序

由于在创建功能包时设置了--node-name 参数，所以 ROS 2 pkg 命令会为两个功能包按照内置模板自动生成两个程序，如图 3-3 所示。

C++功能包中的程序位于功能包 test_package_c 的 src 文件夹下，名为 test_node.cpp，代码如下：

```
#ros2_ws3/src/test_package_c/src/test_node.cpp
#include <cstdio>

int main(int argc, char **argv)
{
```

```
    (void) argc;
    (void) argv;

    printf("hello world test_package_c package\n");
    return 0;
}
```

上述代码的功能是在终端里打印一行内容为“hello world test_package_c package”的字符串。修改该模板文件,即可为功能包编写新的程序。

Python 功能包中的程序位于功能包 test_package_python 内的同名文件夹下,名为 test_node.py,代码如下:

```
#ros2_ws3/src/test_package_python/test_package_python/test_node.py
def main():
    print('Hi from test_package_python.')

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

在上述代码中定义了一个名为 main 的函数,该函数的功能是在终端里打印一行内容为“Hi from test_package_python.”的字符串。如果要改变功能包的默认功能,则只需修改该模板文件中的 main 函数。

对于向功能包添加新的可执行程序不仅需要编写代码,还需要对功能包进行配置,具体方法和步骤将在后续进行详细介绍。

3.1.4 编译功能包

colcon 是一个用于编译和构建软件项目的命令行工具,能够自动化地处理软件包的构建顺序和设置软件包使用的环境,其目标是提供一个更加现代化、灵活且易于使用的构建系统,以满足复杂软件开发的需求。colcon 可用于 ROS 2 和 Gazebo 的编译,是 ROS 2 生态系统中默认的构建系统。

colcon 使用方便,只需一条指令就能完成 ROS 2 工作空间中功能包的编译。在工作空间的根目录(ros2_ws)中编译工作空间,命令如下:

```
cd ~/ros2_ws
colcon build --symlink-install
```

上述命令的功能是先进入工作空间根目录,然后使用 colcon 编译工作空间下所有的功能包,参数--symlink-install 对于 Python 功能包的构建十分重要,这使 colcon 在构建时使用链接的方式管理 Python 程序。这种编译方式在修改 Python 程序的代码后无须再次编译即可执行最新的程序,从而方便地在开发过程中调试 Python 功能包中的代码。上述命令的运行效果如图 3-4 所示,colcon 会对以上创建的两个功能包 test_package_c 和 test_

package_python 进行编译。

```
hw@hw-VirtualBox:~/ros2_ws$ colcon build --symlink-install
Starting >>> test_package_c
Starting >>> test_package_python
Finished <<< test_package_c [1.15s]
```

图 3-4　构建项目

在编译完成后，colcon 就会在工作空间的根目录下自动创建 build、install 和 log 共 3 个子文件夹，如图 3-5 所示。build 文件夹里分别为每个功能包创建一个同名文件夹，用于放置功能包在编译过程中的文件，install 文件夹里分别为每个功能包创建一个同名文件夹，用于放置编译好的程序，并且在 install 文件夹里提供了加载编译好功能包的配置脚本 setup. bash。

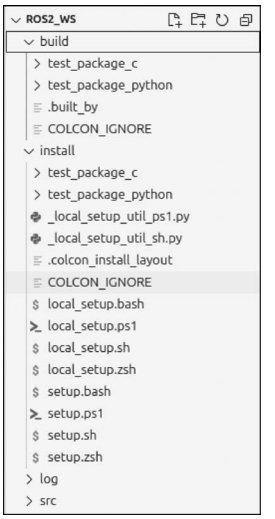

图 3-5　编译结果

除以上编译工作空间内全部功能包的命令外，编译指定功能包是更常见的需求。这在工作空间内包含大量功能包时可有效地减少编译时间。编译指定功包的方法是在 colcon build 命令里设置参数--pcakgeages-select 指定编译的功能包，例如只对 test_package_python 功能包进行编译，命令如下：

```
colcon build --packages-select test_package_python
--symlink-install
```

以上就是使用 colcon 编译功能包的方法。作为 ROS 2 的项目构建工具，colcon 被广泛地应用于 ROS 相关项目（例如 Gazebo）的构建中，并且包含更多更复杂的用法，具体可参阅 colcon 的文档。

3.1.5　运行功能包

功能包在编译完成后，就可以运行编译功能包后生成的可执行程序了。具体的步骤如下。

（1）在终端进入工作空间，命令如下：

```
cd ~/ros2_ws
```

（2）使用 source 命令，将工作空间内的功能包内的可执行程序添加到当前的终端环境中，命令如下：

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

（3）使用 ros2 run 命令执行功能包中的程序，命令如下：

```
ros2 run test_package_c test_node
ros2 run test_package_python test_node
```

以上两条命令分别执行了创建的 C++和 Python 功能包中的示例程序。ros2 run 命令后的两个参数分别是功能包名和可执行文件名。运行的效果如图 3-6 所示,两个命令运行后均会在终端输出一行字符串。

```
hw@hw-VirtualBox:~/ros2_ws$ source install/setup.bash
hw@hw-VirtualBox:~/ros2_ws$ ros2 run test_package_c test_node
hello world test_package_c package
hw@hw-VirtualBox:~/ros2_ws$ ros2 run test_package_python test_node
Hi from test_package_python.
```

图 3-6 运行功能包

注意:经过 source 命令加载配置后,在输入命令时,可以使用 Tab 键补全功能包名和节点名,例如输入 ros2 run test 后按下 Tab 键,系统会自动补全命令或给出命令提示。

3.1.6 功能包的结构

绝大多数的机器人功能能用 C++或 Python 语言实现,鉴于 Python 的易用性,在方法验证阶段更具有优势,本书主要介绍 Python 功能包的结构,以及使用 Python 编写功能包的方法和过程。

在创建 Python 的功能包 test_package_ptyhon 后得到了如图 3-7 所示的结构。

图 3-7 Python 功能包的结构

功能包中相关文件夹和文件的功能如下。

(1) package. xml 文件:这是一个 XML 格式的文件,包含该功能包的元数据,配置一些功能包在编译和运行时的依赖。

(2) resouce 文件夹:内部包含一个与功能包同名的文件夹,内部放置了一些标记文件。

(3) setup. cfg 文件:通过该配置文件为 ros2 run 命令提供该功能包内的可执行文件(程序),也就是使用 ros2 run 命令运行该功能包的程序时需要由 setup. cfg 提供。该文件由构建工具自动维护,无须人工干预。

(4) setup. py 文件:记录安装该功能包的配置项,当向功能包添加新的文件和新的可执行程序时,需要修改该文件,在该文件中注册。

(5) 与功能包同名的文件夹(在图 3-7 中为 test_package_python 文件夹):用于放置功能包代码,是编写 ROS 2 程序时最常使用的文件夹,编写的 Python 代码需放置在该目录中。

在编写 ROS 2 的程序时,除了需要编写实现功能的 Python 代码外,还需要对功能包内的 package. xml 和 setup. py 文件进行配置。下面以 test_package_python 功能包为例对这两个文件进行详细介绍。

package. xml 文件中的内容如下:

```
#ros2_ws3/src/test_package_python/package.xml
<?xml version="1.0"?>
<?xml - model  href =" http://download. ros. org/schema/package _ format3. xsd "
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>test_package_python</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="hw@todo.todo">hw</maintainer>
  <license>MIT</license>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

在以上 package. xml 文件的内容中，标签< name >中设置了当前功能包的名称，标签< version >中设置了当前功能包的版本，标签< description >中设置了该功能包的描述信息，标签< maintainer >中设置了功能包开发者的邮箱和姓名，标签< license >设置了功能包遵循的协议。修改和设置上述信息可以使功能包更清晰。< export >标签中将构建类型< build_type >设置为 ament_python，使在功能包构建时 colcon 将其以 Python 功能包的形式进行构建。< test_depend >标签给出了功能包的一些依赖。

setup. py 文件是一个 Python 代码文件，代码如下：

```
#ros2_ws3/src/test_package_python/setup.py
from setuptools import find_packages, setup

package_name = 'test_package_python'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(Exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='hw',
```

```
        maintainer_email='hw@todo.todo',
        description='TODO: Package description',
        license='MIT',
        tests_require=['pytest'],
        entry_points={
            'console_scripts': [
                'test_node = test_package_python.test_node:main'
            ],
        },
    )
```

在以上 setup.py 文件的内容中，变量 package_name 的值为功能包的名称，需要与 package.xml 文件中的名称一致，setup 函数中的参数 maintainer、maintainer_email、description、license 等与 package.xml 文件中的相应的标签一致，当修改这些信息时需保证两个文件中的内容一致。entry_points 参数是一个字典，其中 console_scripts 设置了当前功能包中可被 ros2 run 命令所能运行的程序名称和实际功能代码之间的映射关系，在修改可执行程序的名称及添加新的可执行程序时都需要修改该参数。下面介绍修改可执行程序名称和添加新可执行程序时对于 entry_points 参数的设置方法：

(1) 修改可执行程序的名称。上述 console_scripts 中设置了将 test_package_python/test_node.py 文件中的 main 函数映射为 test_node 名，以便在 ros2 run 命令中调用，例如 setup.py 文件中的 console_scripts 里的内容如下：

```
'test_node = test_package_python.test_node:main'
```

修改上述内容，修改后的内容如下：

```
'hello_node = test_package_python.test_node:main'
```

这样就会将该包中的输出字符串的程序从名称 test_node 修改为 hello_node。保存后重新编译、安装和运行该程序，命令如下：

```
colcon build --packages-select test_package_python --symlink-install
source install/setup.bash
ros2 run test_package_python hello_node
```

上述代码的运行效果如图 3-8 所示，test_package_python 功能包的可执行文件的名称就从 test_node 修改为 hello_node。

(2) 添加新的可执行程序。一个功能包可以包含多个可执行的程序，功能包 test_package_python 的同名子文件夹是可执行程序的存放位置。从 setup.py 文件中的 console_scripts 的设置可以知道 test_node 程序实际上运行了 test_node.py 文件中的 main 函数。通过向 console_scripts 添加新的参数就能为功能包添加新的可执行程序。为功能包添加新的可执行程序有两种方法：一种是直接在 test_node.py 文件中新建一个函数，并在 setup.py 文

```
hw@hw-VirtualBox:~/ros2_ws$ colcon build --packages-select test_package_python --symlink-install
Starting >>> test_package_python
Finished <<< test_package_python [1.16s]

Summary: 1 package finished [1.26s]
  1 package had stderr output: test_package_python
hw@hw-VirtualBox:~/ros2_ws$ source install/setup.bash
hw@hw-VirtualBox:~/ros2_ws$ ros2 run test_package_python hello_node
Hi from test_package_python.
```

图 3-8　修改可执行程序名称

件中配置；另一种方法是先新建一个 Python 文件并添加一个函数，然后在 setup.py 文件中配置。下面对这两种为功能包添加新的可执行程序的方法进行介绍。

① 在 test_node.py 文件中添加一个函数，修改后文件中的代码如下：

```
#ros2_ws3/src/test_package_python/test_package_python/test_node.py
def main():
    print('Hi from test_package_python.')

def another_main():
    print('another ros2 program from test_package_python')

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

修改 setup.py 文件中 csonle_scripts 列表内的内容，修改后的内容如下：

```
'hello_node = test_package_python.test_node:main',
'another_node = test_package_python.test_node:another_main',
```

上述代码的最后一行是将 test_node.py 文件中的 another_main 函数映射为功能包的 another_node 可执行程序。重新编译和添加功能包后即可执行 another_node 程序，如图 3-9 所示。

```
hw@hw-VirtualBox:~/ros2_ws$ colcon build --packages-select test_package_python --symlink-install
Starting >>> test_package_python
Finished <<< test_package_python [1.11s]

Summary: 1 package finished [1.21s]
  1 package had stderr output: test_package_python
hw@hw-VirtualBox:~/ros2_ws$ source install/setup.bash
hw@hw-VirtualBox:~/ros2_ws$ ros2 run test_package_python another_node
another ros2 program from test_package_python
```

图 3-9　添加可执行程序

② 在 test_node.py 文件所在的目录中新建一个 another_node.py 文件，随后添加以下内容：

```
def main():
    print('another node from another_node.py, not test_node.py')
```

修改 setup.py 文件中的 csonle_scripts 内的内容，修改后的内容如下：

```
'hello_node = test_package_python.test_node:main',
'another_node = test_package_python.test_node:another_main',
'my_node = test_package_python.another_node:main',
```

上述代码的最后一行用于将创建的 another_node.py 文件中的 main 函数映射为功能包的 my_node 可执行程序。重新编译和添加功能包后,即可执行 my_node 程序,如图 3-10 所示。

```
hw@hw-VirtualBox:~/ros2_ws$ colcon build --packages-select test_package_python --symlink-install
Starting >>> test_package_python
Finished <<< test_package_python [1.05s]

Summary: 1 package finished [1.15s]
  1 package had stderr output: test_package_python
hw@hw-VirtualBox:~/ros2_ws$ source install/setup.bash
hw@hw-VirtualBox:~/ros2_ws$ ros2 run test_package_python my_node
another node from another_node.py, not test_node.py
```

图 3-10 添加可执行程序

以上就是在 ROS 2 中进行自定义项目构建、编写、编译和运行的方法,整个流程和使用的关键命令如图 3-11 所示。

图 3-11 ROS 2 自定义功能包创建和运行流程

3.2 rclpy 库的使用

rclpy(ROS Client Library for the Python)库的中文意思是 Python 语言的 ROS 客户端库。rclpy 是 ROS 2 提供给开发者的接口,利用 rclpy 库就能实现对 ROS 2 中节点、话题、服务、动作、参数等核心概念进行编程,从而为实现特定的机器人功能提供便利。

rclpy 库会在安装 ROS 2 时自动安装,无须另行安装。rclpy 库在使用时只需在 Python 的代码中导入,代码如下:

```
import rclpy
```

使用 rclpy 创建 ROS 2 程序的流程如下:

(1) 初始化。在创建节点前必须调用 rclpy.init()函数进行初始化,rclpy.init()函数会为当前的环境创建一个特殊的 Context 对象。

(2) 创建节点,实现具体的功能,利用话题、服务、动作等通信机制实现具体的机器人功能。

(3) 使用 rclpy.spin()函数让节点进入一个循环,以便节点持续接收和处理回调函数。

(4) 结束程序。当程序需要结束时,需要注销初始化时创建的 Context 对象,函数 rclpy.shutdown()提供了该功能。

3.2.1　节点

Node类是rclpy库中表示节点的类，rclpy提供了两种创建Node对象的方法，一种是使用rclpy.create_node()函数，另一种是使用面向对象的方法自定义一个继承rclpy.node.Node类的子类。

rclpy.create_node()函数用于创建节点对象，调用格式如下：

```
create_node(
    node_name: str,
    *,
    context: Context | None = None,
    cli_args: List[str] | None = None,
    namespace: str | None = None,
    use_global_arguments: bool = True,
    enable_rosout: bool = True,
    start_parameter_services: bool = True,
    parameter_overrides: List[Parameter] | None = None,
    allow_undeclared_parameters: bool = False,
    automatically_declare_parameters_from_overrides: bool = False,
    enable_logger_service: bool = False
) -> Node
```

其中，各参数的含义如下。

(1) node_name：设置节点的名称，类型是字符串，必须设置该参数。

(2) context：与节点相关联的上下文对象(Context)，默认值为None，表示与全局的上下文对象相关联。

(3) cli_args：一个字符串列表，包含仅由该节点使用的命令行参数。这些参数用于提取节点使用的重映射及ROS特定的其他设置等。在命令行中使用--ros-args作用域标志，其位于这些参数之前，用于指示其后的参数应该按照ROS的规则进行解析，这些规则包括但不限于处理节点之间的消息重映射、环境变量设置等。

(4) namespace：设置命名空间。为节点及其内的话题等添加前缀，以进行相同名称节点的区分，例如当控制多个相同的机器人时，需要使用多个功能相同的节点、话题等，通过设置不同的命名空间可以防止名称上的冲突，例如节点名称为/my_node，添加命名空间为/robot1，则名称节点就被修改为/robot1/my_node。

(5) use_global_arguments：用于指示节点是否应该忽略进程范围内的命令行参数。如果将这个参数设置为False，则表示该节点不会使用那些为整个ROS进程设置的全局命令行参数，而是只会使用针对该节点本身指定的参数。

(6) enable_rosout：是否启用rosout日志输出功能，默认值为True，表示启用。

(7) start_parameter_services：是否启用节点的参数服务功能，默认值为True，表示启用。

(8) parameter_overrides：一个Parameter列表，用于设置或重置节点参数的值。

(9) allow_undeclared_parameters：如果值为 True，则在创建节点期间将使用 parameter_overrides 中的参数来隐式声明参数。默认值为 False，表示不允许。

(10) enable_logger_service：如果要创建 ROS 2 服务以允许外部节点获取和设置此节点的日志，则为 True，否则日志仅在本地管理而无法远程更改。

使用面向对象的方法自定义一个继承 rclpy.node.Node 类的子类可以创建 ROS 2 中节点的实例。Node 对象创建的格式如下：

```
class Node(
    node_name: str,
    *,
    context: Context | None = None,
    cli_args: List[str] | None = None,
    namespace: str | None = None,
    use_global_arguments: bool = True,
    enable_rosout: bool = True,
    start_parameter_services: bool = True,
    parameter_overrides: List[Parameter] | None = None,
    allow_undeclared_parameters: bool = False,
    automatically_declare_parameters_from_overrides: bool = False,
    enable_logger_service: bool = False
)
```

其中，各参数与 create_node()函数中的含义相同。

创建自定义节点类的方法如下：

```
from rclpy.node import Node
class MyNode(Node):
    def __init__(self, node_name='my_node'):
        super().__init__(node_name=node_name)
    #(... ...节点的具体实现)
```

其中，super().__init__()方法内的参数就是 Node 对象创建所需要的参数。

以上两种节点的创建方法在使用上没有区别，第 1 种方法与 ROS 1 中创建节点的方法相同，第 2 种方法将节点的功能封装在节点类的内部，符合面向对象的原则，在创建节点时推荐使用第 2 种方法。

节点是 ROS 2 中可执行程序的最小单位，节点对象提供了一系列用于实现节点功能的方法，例如创建话题的发布者和订阅者、创建服务的服务器和客户端、创建定时器和参数管理等。

3.2.2 话题

在话题中，有发布者(Publisher)和订阅者(Subscriber)两个参与对象。在创建发布者和订阅者的实例时不直接使用 rclpy 中的 Publisher 和 Subscriber 对象进行构造，而是使用节点实例的 create_publisher()和 create_subscription()方法进行创建。

(1) 使用 create_publisher()方法创建发布者的格式如下：

```
create_publisher(
    msg_type: Any,
    topic: str,
    qos_profile: QoSProfile | int,
    *,
    callback_group: CallbackGroup | None = None,
    event_callbacks: PublisherEventCallbacks | None = None,
    qos_overriding_options: QoSOverridingOptions | None = None,
    publisher_class: type[Publisher] = Publisher
) -> Publisher
```

其中，必须设置的 3 个参数的含义如下。

msg_type：发布的消息类型。ROS 2 对常见的消息类型进行了定义，常见类型有 String、Bool、Byte 等，在 std_msg 库的 msg 下定义了 ROS 2 标准的消息类型。通过 ros2 interface list 命令既可查看当前系统中的所有消息类型，也可以根据需要自行定义消息类型。

topic：发布者发布的话题名称，类型是字符串，一般以“/”字符作为话题名称的首字符。

qos_profile：发布者的服务质量配置(QoSProfile)或历史深度值。当传入整数时，系统会自动配置为 KEEP_LAST 历史策略，并将该数值作为历史深度(Depth)，其他 QoS 参数保持默认值。典型应用场景下，建议将该值设为 10 以内的正整数。

使用上述方法创建发布者的实例后，就可以使用发布者实例的 publish()方法向外发布数据了。

(2) 使用 create_subscription()方法创建订阅者的格式如下：

```
create_subscription(
    msg_type: Any,
    topic: str,
    callback: (MsgType@create_subscription) -> None,
    qos_profile: QoSProfile | int,
    *,
    callback_group: CallbackGroup | None = None,
    event_callbacks: SubscriptionEventCallbacks | None = None,
    qos_overriding_options: QoSOverridingOptions | None = None,
    raw: bool = False
) -> Subscription
```

其中，callback 参数是一个包含一个参数的回调函数，用于处理接收的消息，其他参数的含义与发布者中的参数的含义相同。

3.2.3　服务

在服务中，有服务器和客户端两个参与对象。在创建服务器和客户端的实例时，需要通过节点的 create_service()和 create_client()方法进行创建。

(1) 使用 create_service()方法创建服务器的格式如下：

```
create_service(
    srv_type: Any,
    srv_name: str,
    callback: (SrvTypeRequest@create_service, SrvTypeResponse@create_service)
-> SrvTypeResponse@create_service,
    *,
    qos_profile: QoSProfile = qos_profile_services_default,
    callback_group: CallbackGroup | None = None
) -> Service
```

其中,3 个必须设置的参数的含义如下。

srv_type：服务消息类型。常见类型有 Empty、Trigger、SetBool 等,在 std_srvs 库的 srv 下定义了 ROS 2 标准的服务消息类型。通过 ros2 interface list 命令既可查看当前系统中的所有服务消息类型,也可以根据需要自行定义服务消息类型。

srv_name：服务名称,通常是以“/”开头的字符串。

callback：由用户定义的服务器回调函数,回调函数接收 request 和 response 两个参数,分别用于获取用户的请求,以及返回请求后的响应。

(2) 使用 create_client()方法创建客户端的格式如下：

```
create_client(
    srv_type: Any,
    srv_name: str,
    *,
    qos_profile: QoSProfile = qos_profile_services_default,
    callback_group: CallbackGroup | None = None
) -> Client
```

其参数的含义与创建服务器的参数含义相同。在得到客户端的实例后,常用的客户端的方法如下。

service_is_ready()：检查客户端所请求的服务器是否正常(能够提供服务),当返回值为 True 时表示正常,否则为 False。

wait_for_service(timeout_sec：float | None=None)：等待服务器,可以设计一个等待的时长,当返回值为 True 时表示服务器正常,当返回值为 False 时表示超时。

call(request：SrvTypeRequest)：向服务器发起同步请求。

call_async(request：SrvTypeRequest)：向服务器发起异步请求,返回一个 Future 对象。通过 Future 对象的 add_done_callback()方法可以添加一个以该 Future 对象为参数的回调函数,该回调函数用于处理服务器响应。

3.2.4 动作

在动作中,有动作服务器和动作客户端两个参与对象。与话题和服务通过节点提供的

方法创建不同，动作服务器端和动作客户端由 rclpy.action 包下的 ActionServer 类和 ActionClient 类进行创建。

（1）动作服务器端 ActionServer 类创建的格式如下：

```
class ActionServer(
    node: Self@ActionserverNode,
    action_type: type[Fibonacci],
    action_name: str,
    execute_callback: Any,
    *,
    callback_group: Any | None = None,
    goal_callback: Any = default_goal_callback,
    handle_accepted_callback: Any = default_handle_accepted_callback,
    cancel_callback: Any = default_cancel_callback,
    goal_service_qos_profile: QoSProfile = qos_profile_services_default,
    result_service_qos_profile: QoSProfile = qos_profile_services_default,
    cancel_service_qos_profile: QoSProfile = qos_profile_services_default,
    feedback_pub_qos_profile: QoSProfile = QoSProfile(depth=10),
    status_pub_qos_profile: QoSProfile = qos_profile_action_status_default,
    result_timeout: int = 900
)
```

其中，4 个必须设置的参数含义如下。

node：运行该动作服务器的节点。

action_type：动作的消息类型。

action_name：动作的名称。客户端通过该名称与动作服务器进行通信。

execute_callback：用于处理动作客户端请求的回调函数。

（2）动作客户端 ActionServer 类创建的格式如下：

```
class ActionClient(
    node: Self@ActionclientNode,
    action_type: type[Fibonacci],
    action_name: str,
    *,
    callback_group: Any | None = None,
    goal_service_qos_profile: QoSProfile = qos_profile_services_default,
    result_service_qos_profile: QoSProfile = qos_profile_services_default,
    cancel_service_qos_profile: QoSProfile = qos_profile_services_default,
    feedback_sub_qos_profile: QoSProfile = QoSProfile(depth=10),
    status_sub_qos_profile: QoSProfile = qos_profile_action_status_default
)
```

动作客户端的参数与动作服务器的参数含义相同。在创建动作客户端后，通过动作客户端的 send_goal_async()方法向动作服务器发起异步请求。

3.2.5 参数

rclpy 中的 Parameter 类用于创建参数,通过节点的 set_parameters()方法将多个参数添加到节点中,此外节点的 declare_parameter()方法用于向节点创建和添加一个参数。当节点中的参数的值发生变化时,可以通过节点的 add_pre_set_parameters_callback()、add_on_set_parameters_callback()和 add_post_set_parameters_callback()方法对参数值在修改前、修改中和修改后设置相应的回调函数。

参数 Parameter 类的创建方法如下:

```
rclpy.parameter.Parameter(
    name: str,
    type_: Any | None = None,
    value: Any | None = None
)
```

其中,name 为参数名称,类型是字符串,value 为参数的值,type_为参数值的类型,需要是 rclpy. Parameter. Type 中定义的类型'BOOL'、'BOOL_ARRAY'、'BYTE_ARRAY'、'DOUBLE'、'DOUBLE_ARRAY'、'INTEGER'、'INTEGER_ARRAY'、'NOT_SET'、'STRING'和'STRING_ARRAY'之一。

例如,创建一个名为 number,类型为整数,值为 1 的参数,代码如下:

```
numparam=rclpy.parameter.Parameter('number', rclpy.Parameter.Type.INTEGER, 1)
```

创建后的参数可以通过节点的 set_parameters()设置为节点的参数,代码如下:

```
node.set_parameters([numparam])
```

相较于以上定义参数和向节点添加参数两个步骤,节点的 declare_parameter()方法可以直接向节点添加参数,例如为节点添加一个名为 msg、类型为字符串、值为"hello world!"的参数,代码如下:

```
node.declare_parameter('msg', 'hello world!')
```

由于参数用于对节点进行配置,一般在节点启动时添加命令行参数--ros-args -p name:=value -p name2:=value2…向节点传递参数,节点初始化时更新默认参数,从而确保节点在启动时使用最新的参数。在实现上,通过在节点的初始化方法内使用下面的方法使节点在启动时优先使用传入的参数,在没有传入参数时使用默认参数,代码如下:

```
node.declare_parameter('number', 1).get_parameter_value().integer_value
node.declare_parameter('msg', 'hello world!').get_parameter_value().
string_value
```

3.2.6　消息接口

消息接口(interface)用于规定话题、服务和动作等通信方式上参与的两方之间传递消息的格式,例如同一个话题的发布者和订阅者在创建时都要规定消息的格式是字符串还是数值等其他类型。ROS 2 使用接口定义语言作为消息定义的规范,只需定义消息的格式,编译后就能生成相应的消息类型作为通信双方间的消息格式进行数据传输。

(1) 话题消息接口的定义:在功能包下创建一个名为 msg 的文件夹,在该文件夹内创建以.msg 结尾的文件。例如定义一个表示机器人运动状态的消息 State.msg,内容如下:

```
string name
float32 speed
bool running
```

上述的每行表示消息内的一种数据,每行由空格分开的两列组成,第 1 列是数据类型,第 2 列是数据的名称,名称由小写字母和下画线构成,但必须以小写字母作为起始字符,可以带有多个下画线,但不能有连续的两个下画线,并且下画线不能作为名称的结尾。

数据类型既可以是基本类型,也可以是其他功能包中定义的消息类型。基本数据类型包括 bool、byte、char、float32、float64、int8、uint8、int16、uint16、int32、uint32、int64、uint64、string、wstring 等类型,以及与基本类型对应的数组类型。

(2) 服务消息接口的定义:在功能包下创建一个名为 srv 的文件夹,在该文件夹内创建以.srv 结尾的文件。相较于话题的消息接口,服务的消息接口需要包含两部分请求的数据格式和响应的数据格式,例如定义一个请求和响应均为字符串的消息 Msg.srv,内容如下:

```
string reqdatastr
---
string rspdatastr
```

上述消息接口由“---”分为两部分,第一部分为服务中客户端请求的格式,第二部分为服务中服务器响应的格式。上下两部分的定义方式与话题中消息的定义方式相同。

(3) 动作消息接口的定义:在功能包下创建一个名为 action 的文件夹,在该文件夹内创建以.action 结尾的文件。动作的消息接口由 3 部分构成,分别是客户端的请求,服务器端的最终响应结果和服务器端的反馈,例如定义一个名为 Fibonacci.action 的消息,内容如下:

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

注意：由于消息接口文件会在 Python 语言中以类名进行导入，因此消息接口文件的名称必须符合 Python 类名称的定义规范，并且消息接口文件的首字母要大写，例如，表示机器人状态的消息接口文件可以是 State.msg，而不能是 state.msg。

3.2.7 案例：创建话题发布者

在工作空间内创建一个名为 pubsub_test 的功能包，命令如下：

```
ROS 2 pkg create --build-type ament_python --node-name publisher --dependencies
rclpy std_msgs --license MIT pubsub_test
```

以上命令中的--dependencies 参数将 rclpy 和 std_msgs 库添加为当前创建包的依赖，在编写的 ROS 2 代码中需要使用这两个库。

打开 pubsub_test 功能包中的 publisher.py 文件，编辑内容如下：

```
#ros2_ws3/src/pubsub_test/pubsub_test/publisher.py
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class PubNode(Node):
    def __init__(self,node_name='pubnode'):
        super().__init__(node_name=node_name)
        self.pub=self.create_publisher(String,'/hello',5)
        self.msg=String()
        self.num=0
        self.logger=self.get_logger()
        self.create_timer(1.0,self.pubmsg)

    def pubmsg(self):
        self.msg.data=f'the {self.num}th hello message.'
        self.num+=1
        self.pub.publish(self.msg)
        self.logger.info(f'send message: {self.msg.data}')

def main(args=None):
    rclpy.init(args=args)
    node=PubNode()
    try:
        rclpy.spin(node)
    except Exception:
        rclpy.shutdown()
        exit(0)
if __name__ == '__main__':
    main()
```

在上述代码中，创建了一个继承自 rclpy. node. Node 且名为 PubNode 的自定义节点类，在类的初始化方法中，根据 node_name 参数设置了节点的名称，创建了一个消息类型为 String 且名称为/hello 的发布者，创建了一个 String 类型的消息变量，通过节点的 get_logger()获得了日志记录对象，创建了一个定时器对象，每隔 1s 会调用类的 pubmsg()方法。pubmsg()方法在调用时会构造一个字符串消息，并使用发布者发布出去，同时使用日志记录对象的 info()方法在终端输出消息。

main()函数使用了 rclpy 运行节点的标准结构，包含了初始化 rclpy 环境、创建节点、运行节点、退出节点等过程，提供了节点运行的框架，对所有节点的运行都适用。

保存上述代码后，进行编译、安装和运行该节点，命令如下：

```
colcon build --packages-select pubsub_test --symlink-install
source install/setup.bash
ros2 run pubsub_test publisher
```

上述命令的运行效果如图 3-12 所示，在节点启动后，在终端每隔 1s 会使用日志显示节点向外发送的消息。

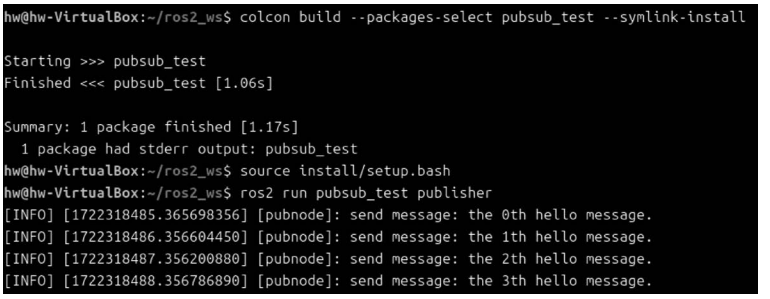

图 3-12　发布话题

检查话题的发布有 3 种方法：一是使用 ros2 topic echo 命令在终端显示；二是使用 rqt 在图形用户界面显示；三是使用代码编写一个接收者。一般在测试和调时，使用前两种方法进行检查。

(1) 使用终端订阅话题，查看话题上的传输数据，命令如下：

```
ros2 topic echo /hello
```

上述命令的运行效果如图 3-13 所示，在终端会显示在话题/hello 上接收的消息。

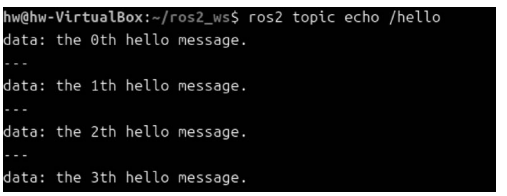

图 3-13　命令行显示话题

(2) 使用 rqt 订阅话题。启动 rqt,在终端输入命令 rqt,从 Plugins 菜单打开 Topic Monitor,勾选/hello 选项,启动对话题/hello 的监听,展开/hello 即可看到在话题/hello 上发布的消息等信息,如图 3-14 所示。

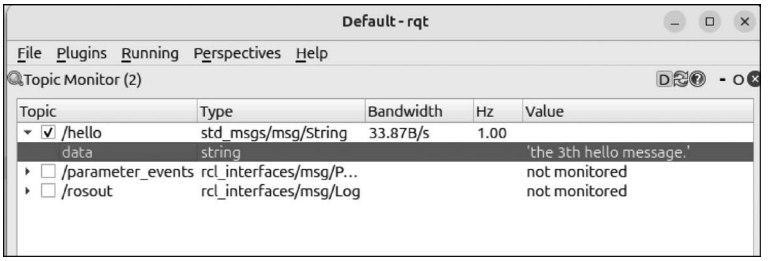

图 3-14 rqt 显示话题

3.2.8 案例:创建话题订阅者

在 3.2.7 节的案例中创建了一个话题发布者发布消息,以下将创建一个话题订阅者,以便接收以上发布者发布的消息。在上述的功能包 pubsub_test 内与 publisher.py 文件同目录内创建一个名为 subscriber.py 的文件,编写的代码如下:

```
#ros2_ws3/src/pubsub_test/pubsub_test/subscriber.py
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class SubNode(Node):
    def __init__(self,node_name='subnode'):
        super().__init__(node_name=node_name)
        self.logger=self.get_logger()
        self.sub=self.create_subscription(String,'/hello',self.process_msg,5)

    def process_msg(self,msg):
        self.logger.info(f'receive msg: {msg.data}')

def main(args=None):
    rclpy.init(args=args)
    node=SubNode()
    try:
        rclpy.spin(node)
    except Exception:
        rclpy.shutdown()
        exit(0)
if __name__=='__main__':
    main()
```

在上述代码中创建了一个 SubNode 类的节点,在该节点内创建了一个订阅者。该订阅者订阅了以 String 为消息类型的/hello 话题,并通过回调函数 process_msg()处理接收的

消息。在回调函数内，使用节点的日志输出功能向终端输出接收的消息。

此外，还需要修改 pubsub_test 功能包的 setup.py 文件，修改 entry_points 的值，代码如下：

```
entry_points={
        'console_scripts': [
            'publisher = pubsub_test.publisher:main',
            'subscriber =pubsub_test.subscriber:main',
        ],
    },
```

在上述代码中，将创建的订阅者通过名称 subscriber 提供给可执行程序。

经过编译、安装后，即可运行该节点：

```
ros2 run pubsub_test subscriber
```

当发布者仍处于消息的发布时，上述订阅者节点在运行后就会在终端里显示接收的消息，效果如图 3-15 所示。

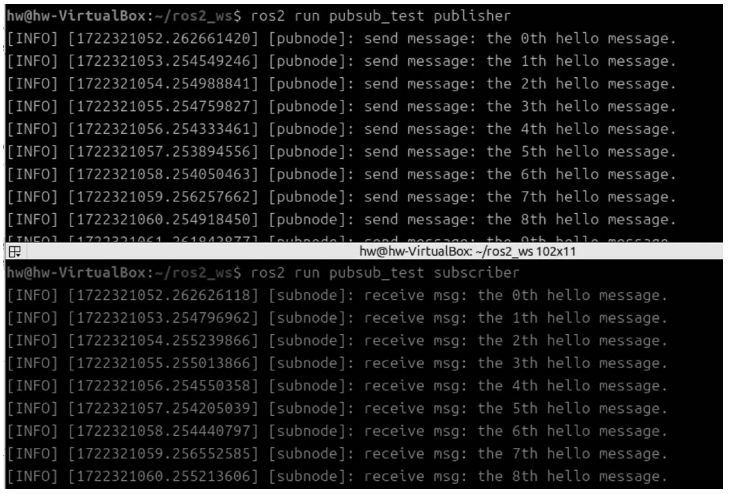

图 3-15　命令行测试订阅者

除了可以使用自定义的发布者对订阅者进行测试外，还可以使用终端和 rqt 来测试订阅者。

(1) 使用终端发送消息，命令如下：

```
ros2 topic pub /hello std_msgs/msg/String "{data: hello subscriber.}"
```

(2) 使用 rqt 的 Message Publisher 发送消息，如图 3-16 所示。

注意：当在新的终端里运行工作空间内自定义的程序时，先要使用命令 source install/

setup.bash 把工作空间的程序载入当前终端里，否则 ROS 2 无法识别工作空间内的自定义程序。

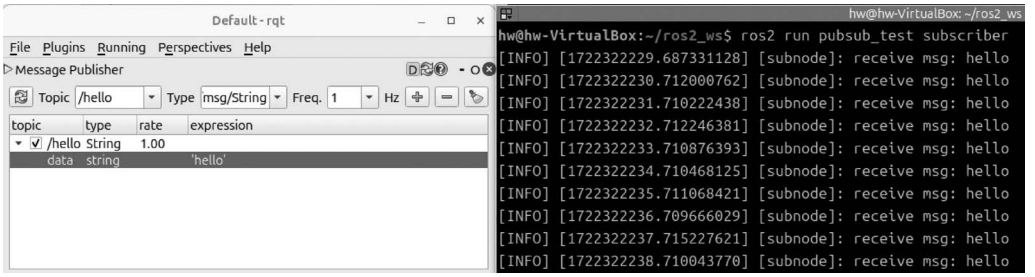

图 3-16　rqt 测试订阅者

此时，可以通过 rqt 的 node graph 功能查看当前 ROS 2 中的节点关系图，如图 3-17 所示，在节点关系图中可以看到发布消息的/pubnode 节点、接收消息的/subnode 节点以及连接两个节点的话题/hello。

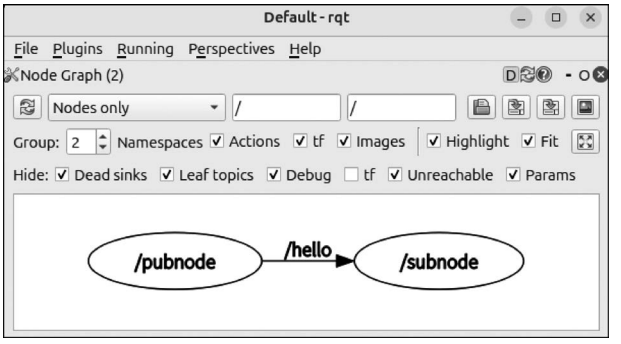

图 3-17　发布者与订阅者节点图

3.2.9　案例：创建服务器

在工作空间内创建一个名为 serviceclient_test 的功能包，命令如下：

```
ros2 pkg create --build-type ament_python --node-name server --dependencies
rclpy std_srvs --license MIT serviceclient_test
```

以上命令中--dependencies 参数用于将 rclpy 和 std_srvs 库添加为当前创建包的依赖，在编写的 ROS 2 代码中需要使用这两个库。

打开 serviceclient_test 功能包中的 server.py 文件，编写的代码如下：

```
#ros2_ws3/src/serviceclient_test/serviceclient_test/server.py
import rclpy
from rclpy.node import Node
```

```
from std_srvs.srv import SetBool
#ros2 interface package std_srvs

class ServerNode(Node):
    def __init__(self,node_name='servernode'):
        super().__init__(node_name=node_name)
        self.logger=self.get_logger()
        self.server=self.create_service(SetBool,'/testservice',self.response_
callback)

    def response_callback(self, request,response):
        reqdata=request.data
        if reqdata:
            response.success=True
            response.message='请求的数据为 True'
        else:
            response.success=True
            response.message='请求的数据为 False'
        self.logger.info(response.message)
        return response

def main(args=None):
    rclpy.init(args=args)
    node=ServerNode()
    try:
        rclpy.spin(node)
    except Exception:
        rclpy.shutdown()
        exit(0)

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

在上述代码中，在节点 ServerNode 中创建了一个名为/testservice 的服务，其消息类型为 ROS 2 自带的标准服务消息功能包 std_srvs 下的 SetBool 类型，并且设置其回调函数用于处理客户端的请求。回调函数 response_callback()会接收 request 和 response 两个参数，分别表示传入的请求和响应的结果，在回调函数内根据服务的消息类型 SetBool 的数据设置了对于请求的不同处理。

消息类型 SetBool 在请求和响应时的详细结构可以通过 ros2 interface show 查询，命令如下：

```
ros2 interface show std_srvs/srv/SetBool
```

以上命令的执行结果如下：

```
bool data #e.g. for hardware enabling / disabling
---
```

```
bool success #indicate successful run of triggered service
string message #informational, e.g. for error messages
```

结果显示：在 SetBool 消息类型中请求包含一个 bool 类型的 data 字段，响应包含一个 bool 类型的 success 字段和一个 string 类型的 message 字段。根据查询结果就可以在实际的代码中正确地使用该消息类型。

自定义服务的编译、安装和启动，如图 3-18 所示。在启动服务节点后，可在新的终端里发送服务请求命令进行测试，命令如下：

```
ros2 service call /testservice std_srvs/srv/SetBool "{data : True}"
```

上述命令的运行效果如图 3-18 所示，在发起请求后，服务器端接收处理通过日志信息显示了请求的消息，并向客户端发送了响应，客户端显示了服务器发送的响应。

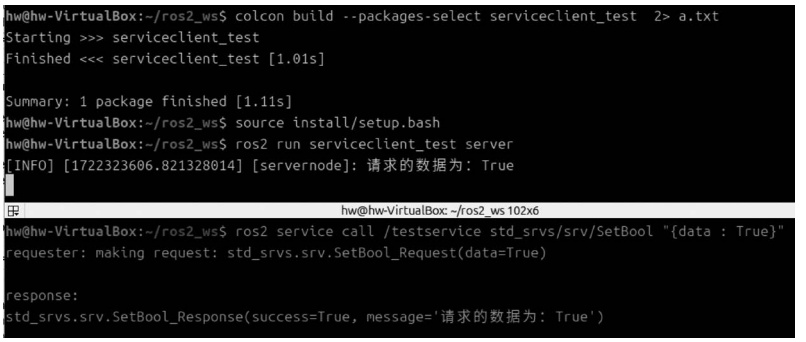

图 3-18　测试自定义服务器

同样，也可以使用 rqt 的 Service Caller 功能对自定义服务器进行测试，如图 3-19 所示。

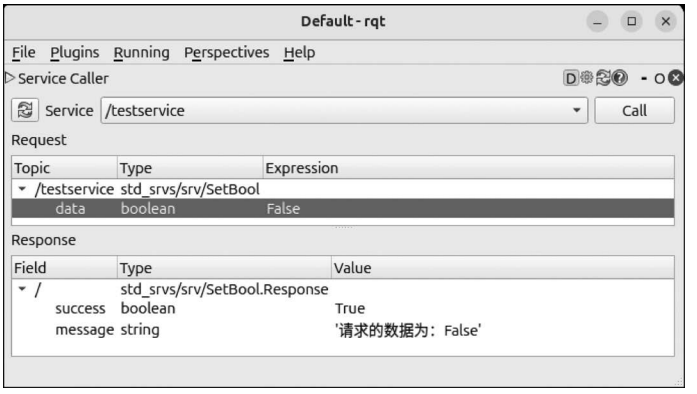

图 3-19　rqt 测试自定义服务器

3.2.10　案例：创建客户端

在 3.2.9 节的案例中创建了一个服务器，下面将创建一个客户端，以便向服务器发送请

求并显示服务器的响应。在上述的功能包 serviceclient_test 内与 server.py 文件同目录内创建一个名为 client.py 的文件，编写的代码如下：

```
#ros2_ws3/src/serviceclient_test/serviceclient_test/client.py
import rclpy
from rclpy.node import Node
from std_srvs.srv import SetBool

class ClientNode(Node):
    def __init__(self,node_name='clientnode'):
        super().__init__(node_name=node_name)
        self.req=SetBool.Request()
        self.req.data=False
        self.client=self.create_client(SetBool,'/testservice')
        while not self.client.wait_for_service(timeout_sec=1.0):
            print('服务器不可用,正在等待... ...')

        self.create_timer(1,self.request)

    def request(self):
        self.req.data= not self.req.data
        req= self.client.call_async(self.req)
        req.add_done_callback(self.response_callback)

    def response_callback(self,future):
        response = future.result()
        if response is not None:
            self.get_logger().info(f'服务运行结果{response.success},{response.
message}')
        else:
            self.get_logger().info('请求失败')
def main(args=None):
    rclpy.init(args=args)
    node=ClientNode()
    try:
        rclpy.spin(node)
    except Exception:
        node.destroy_node()
        rclpy.shutdown()
        exit(0)

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

在上述代码中，创建了一个客户端节点 ClientNode，在该节点内创建了一个消息类型为 SetBool 且名称为/testservice 的客户端。客户端的 wait_for_service()方法用于阻塞和等待服务可用，在服务可用后，创建了一个定时器，用于周期性地向服务器发送请求。在 request()方法内实现了请求消息的构造，向服务器发起异步请求，并且对异步请求设置了处理服务器

响应的回调函数 response_callback()。response_callback()回调函数用于处理接收的服务器响应,并在终端里显示。

注意:在上述客户端中使用了异步编程方法,异步编程是一种重要的编程技术,可以通过查看 Python 的标准库 asyncio 以了解异步编程的相关概念。

此外,还需要修改 serviceclient_test 功能包中的 setup.py 文件中 entry_points 变量的值,代码如下:

```
entry_points={
        'console_scripts': [
            'server = serviceclient_test.server:main',
            'client = serviceclient_test.client:main',
        ],
    },
```

在上述代码中,将创建的客户端可执行程序命名为 client。

经过编译、安装后,即可运行该节点,命令如下:

```
ros2 run serviceclient_test client
```

当服务器在运行时,上述客户端节点在运行后就会在终端显示服务器的响应,在服务器显示客户端的请求,如图 3-20 所示。

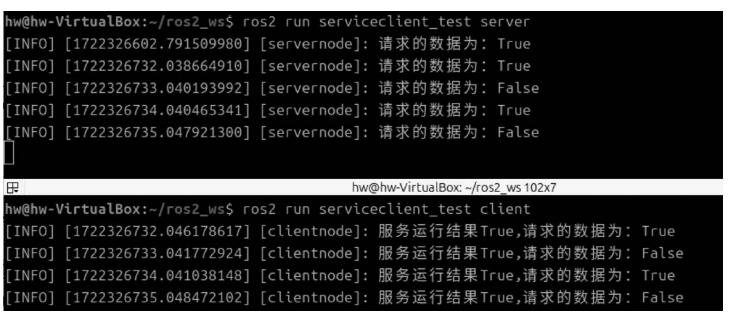

图 3-20 测试自定义客户端

3.2.11 案例:创建动作服务器

在该案例中,将创建一个动作服务器,用于计算 Fibonacci 数列中的前 n 个元素,n 的值由客户端给出。Fibonacci 数列使用递归的方式计算各项的值,计算公式如下:

$$f(n)=\begin{cases}0, & n=0\\ 1, & n=1\\ f(n-1)+f(n-2), & n>1\end{cases} \tag{3-1}$$

在式(3-1)中 n 为 Fibonacci 数列的项数,将上述 Fibonacci 数列计算的任务看作一个需

要长时间完成的工作，使用动作服务器在计算的过程中不断地向客户端反馈任务完成情况，在完成计算后将前 n 个元素发送给客户端。

首先，在工作空间内创建一个名为 action_test 的功能包，命令如下：

```
ros2 pkg create --build-type ament_python --node-name server --dependencies
rclpy test_msgs --license MIT action_test
```

以上命令中--dependencies 参数用于将 rclpy 和 test_msgs 库添加为当前创建包的依赖，在编写的 ROS 2 节点代码中需要使用这两个库。

然后打开 action_test 功能包中的 server.py 文件，编写的代码如下：

```
#ros2_ws3/src/action_test/action_test/server.py
import rclpy
from rclpy.node import Node
from test_msgs.action import Fibonacci
from rclpy.action import ActionServer
import time
#ros2 interface package test_msgs

class ActionserverNode(Node):
    def __init__(self,node_name='actionservernode'):
        super().__init__(node_name=node_name)
        self.logger=self.get_logger()
        self.actionserver = ActionServer ( self, Fibonacci, '/fibonacci ', self.
execute_callback)
        self.logger.info('action server started ...')

    def execute_callback(self, goal_handle):
        self.logger.info('Executing goal...')

        feedback_msg = Fibonacci.Feedback()
        partial_sequence = [0, 1]

        for i in range(1, goal_handle.request.order):
            partial_sequence.append(
                partial_sequence[i] + partial_sequence[i-1])
            progress=[i+1,goal_handle.request.order]
            self.logger.info(f'Feedback: {progress}')
            feedback_msg.sequence=progress
            goal_handle.publish_feedback(feedback_msg)
            time.sleep(1)

        goal_handle.succeed()
        result = Fibonacci.Result()
        result.sequence = partial_sequence
        self.logger.info('Reach goal...')
        return result
```

```
def main(args=None):
    rclpy.init(args=args)
    node=ActionserverNode()
    try:
        rclpy.spin(node)
    except Exception:
        rclpy.shutdown()
        exit(0)

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

在上述代码中定义了一个动作服务器的节点类,在类的初始化中为节点添加了一个消息类型为 Fibonacci 且名称为/fibonacc 的动作服务器,该动作服务器在接收到动作客户端的请求后会执行节点的回调函数 execute_callback()。回调函数 execute_callback()接收一个包含客户端请求的参数 goal_handle,从 goal_handle. request. order 中获得客户端请求计算的元素的个数,按照数列递推公式循环计算各个元素,并且将当前计算的元素序号与总元素个数保存到 progress 变量中,随后使用动作的反馈话题 goal_handle. publish_feedback 将反映计算进度的 progress 变量发送给动作客户端,为了体现任务执行的长期性使用了 time. sleep()函数进行延时(模拟一个耗时的计算过程)。当数列中的所有元素计算完成后,将目标请求置为成功,将结果保存到 result 变量中并返回,动作服务器发送 result 变量的值并结束整个动作。

在上述动作服务器的代码中,使用了 test_msgs. action 包中的 Fibonacci 消息。查询该消息的详细结构,命令如下:

```
ros2 interface show test_msgs/action/Fibonacci
```

上述命令执行后的输出结果如下:

```
#goal definition
int32 order
---
#result definition
int32[] sequence
---
#feedback
int32[] sequence
```

最后,对功能包 action_test 进行编译和安装后,启动动作服务器节点,命令如下:

```
ros2 run action_test server
```

使用 ROS 2 的命令行工具测试动作服务器,命令如下:

```
ros2 action send_goal /fibonacci test_msgs/action/Fibonacci "{order : 10}"
--feedback
```

在上述命令中添加了参数--feedback，用于显示动作服务器的反馈信息，命令的执行效果如图 3-21 所示，动作服务器在接收到客户端的请求后会在执行请求的同时向客户端反馈执行的进度，在执行完成后将结果发送回动作客户端。

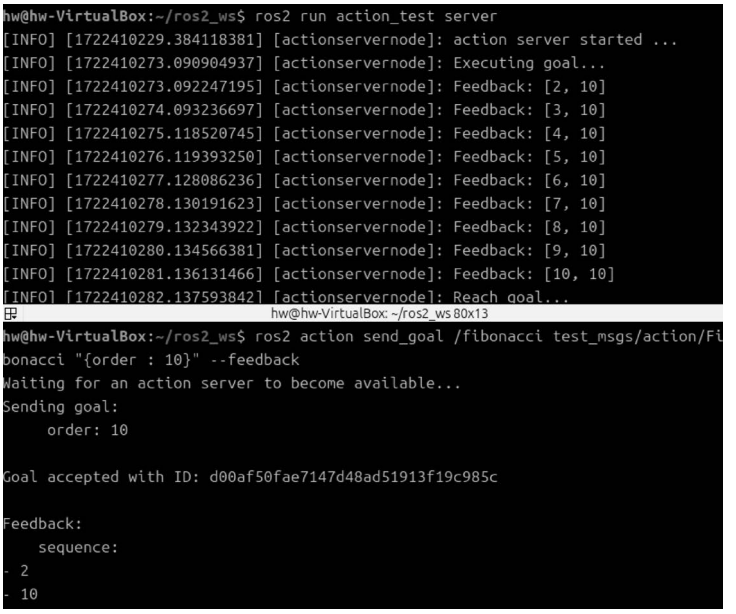

图 3-21　测试动作服务器

3.2.12　案例：创建动作客户端

在 3.2.11 节的案例中创建了一个计算 Fibonacci 数列的动作服务器，以下将创建一个动作客户端向服务器发送请求并显示动作服务器的反馈和最终结果。在上述功能包 action_test 内与 server.py 文件同目录内创建一个名为 client.py 的文件，编写的代码如下：

```
#ros2_ws3/src/action_test/action_test/client.py
import rclpy
from rclpy.node import Node
from test_msgs.action import Fibonacci
from rclpy.action import ActionClient
#ros2 interface package test_msgs

class ActionclientNode(Node):
    def __init__(self,node_name='actionclientnode'):
        super().__init__(node_name=node_name)
        self.logger=self.get_logger()
        self.actionclient=ActionClient(self,Fibonacci,'/fibonacci')
        while not self.actionclient.wait_for_server(timeout_sec=10.0):
            print('服务器不可用，正在等待... ...')
```

```
        self.send_goal(10)

    def send_goal(self, order=10):
        goal_msg = Fibonacci.Goal()
        goal_msg.order = order

        self.send_goal_future = self.actionclient.send_goal_async(goal_msg,
feedback_callback=self.feedback_callback)

        self.send_goal_future.add_done_callback(self.goal_response_callback)

    def feedback_callback(self, feedback_msg):
        feedback = feedback_msg.feedback
        self.logger.info(f'Received feedback: {feedback.sequence}')

    def goal_response_callback(self,future):
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.logger.info('Goal rejected :(')
            return

        self.logger.info('Goal accepted :)')
        get_result_future = goal_handle.get_result_async()
        get_result_future.add_done_callback(self.get_result_callback)

    def get_result_callback(self,future):
        result = future.result().result
        self.logger.info('Result: {0}'.format(result.sequence))
        rclpy.shutdown()

def main(args=None):
    rclpy.init(args=args)
    node=ActionclientNode()
    try:
        rclpy.spin(node)
    except Exception:
        rclpy.shutdown()
        exit(0)

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

在上述代码中创建了一个动作客户端的节点,在节点的初始化函数中创建了一个动作客户端对象,在动作服务器可用后执行方法 send_goal(10)向服务器发送计算前 10 个元素的请求。在 send_goal()方法中构造了一个请求目标变量 goal_msg,在使用动作客户端的 send_goal_async()方法向动作服务器发起目标请求的同时添加了处理服务器反馈的回调函数 feedback_callback(),最后添加了动作执行结束后的回调函数 add_done_callback()。

在 feedback_callback()函数中对接收的反馈消息进行了显示。在动作结束后的回调函数 add_done_callback()中将显示服务器发送来的结果。

此外,还需要修改 action_test 功能包中的 setup.py 文件中 entry_points 的值,代码如下:

```
entry_points={
        'console_scripts': [
            'server = action_test.server:main',
            'client = action_test.client:main'
        ],
    },
```

在上述代码中,将创建的动作客户端可执行程序命名为 client。

经过编译、安装后,即可运行该节点,命令如下:

```
ros2 run serviceclient_test client
```

当动作服务器保持运行时,上述动作客户端节点在运行后就会在终端显示服务器的反馈消息和最终的动作执行结果,动作服务器在终端显示客户端的请求和执行进度,如图 3-22 所示。

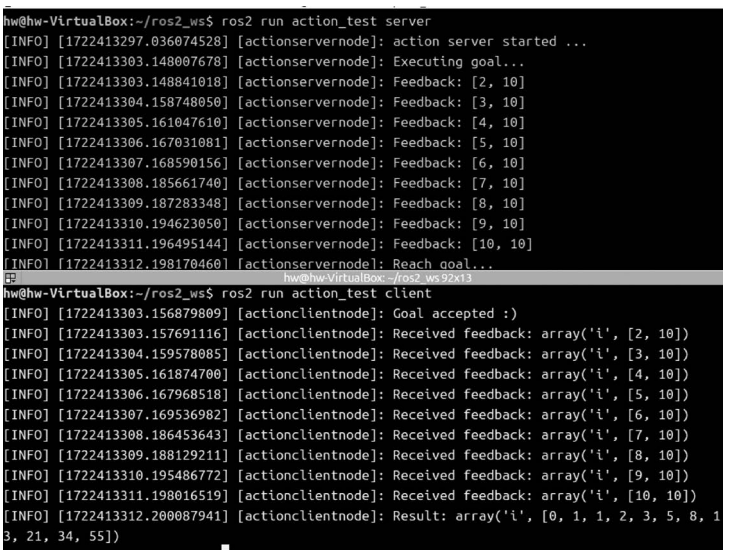

图 3-22　测试动作客户端

3.2.13　案例:创建参数服务

在 ROS 2 中参数属于节点,节点提供了参数的创建和修改等管理功能。以下通过例子说明参数在节点中的使用方法。

首先,创建一个功能包 parameter_test,命令如下:

```
ros2 pkg create --build-type ament_python --node-name parameter --dependencies
rclpy --license MIT parameter_test
```

其次,修改功能包中的 parameter.py 文件中的代码,修改后的代码如下:

```
#ros2_ws3/src/parameter_test/parameter_test/parameter.py
import rclpy
from rclpy.node import Node
import rclpy.parameter

class ParamNode(Node):
    def __init__(self):
        super().__init__('param_node')
        self.logger=self.get_logger()
        #1.创建参数
        self.declare_parameter('number', 10)
        self.declare_parameter('string','hello world!')
        #2.测试是否有参数
        if self.has_parameter('string'):
            self.logger.info(f'has param string')
        #3.创建并获取参数最新的值,可以被命令行中传入的参数修改
        self.length = self.declare_parameter(
          'length', 3).get_parameter_value().integer_value
        self.logger.info(f"parameter length is {self.length}")

        #4.参数修改回调函数
        self.add_post_set_parameters_callback(self.show)

        self.updateparam()

    def updateparam(self):
        #取得参数
        num_param = self.get_parameter('number').get_parameter_value().integer_value
        self.get_logger().info(f'param number = {num_param}')
        #创建参数
        my_new_param = rclpy.parameter.Parameter('number',rclpy.Parameter.
Type.INTEGER,1)
        all_new_parameters = [my_new_param]
        self.logger.info('change param number')
        #修改参数
        self.set_parameters(all_new_parameters)

    def show(self,params):
        for i in params:
            if i.type_==rclpy.Parameter.Type.INTEGER:
                value=i.get_parameter_value().integer_value
            else:
                value=i.get_parameter_value().string_value
```

```
            self.logger.info(f'param {i.name} = {value}')

    def main(args=None):
        rclpy.init(args=args)
        node = ParamNode()
        try:
            rclpy.spin(node)
        except Exception:
            rclpy.shutdown()
            exit(0)

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

在上述代码中创建了一个展示节点中参数的用法，declare_parameter()方法用于声明和创建参数，get_parameter_value()方法用于获取参数，has_parameter()方法用于判断节点是否包含指定名称的参数，set_parameters()方法用于一次性设置多个由 Parameter 类创建的参数。

编译、安装功能包后，运行功能包，命令如下：

```
ros2 run parameter_test parameter
```

上述命令的效果如图 3-23 所示，通过节点提供的参数功能对参数的存在进行了判断，显示了参数的值，显示了参数的值在修前后的变化，以及参数发生变化后执行的回调函数。

```
hw@hw-VirtualBox:~/ros2_ws$ ros2 run parameter_test parameter
[INFO] [1722734668.903605021] [param_node]: has param string
[INFO] [1722734668.904721524] [param_node]: parameter length is 3
[INFO] [1722734668.904914958] [param_node]: param number = 10
[INFO] [1722734668.905072054] [param_node]: change param number
[INFO] [1722734668.905346559] [param_node]: param number = 1
```

图 3-23　执行参数节点

在节点启动时，可通过 ROS 2 的命令行参数功能在参数初始化时通过命令行传入参数的值，可替代节点中声明参数的默认值，例如，在上述代码中声明的 length 参数的默认值为 3，将其值修改为 33 的命令如下：

```
ros2 run parameter_test parameter --ros-args -p length:=33
```

在上述命令中--ros-args 参数用于提供后续的命令行参数，-p 用于标识设置节点参数，length:=33 用于将参数 length 设置为 33。当一次性设置多个参数的值时，使用--ros-args -p p1:=value1 -p p2:=value2 -p p3:=value3 ... 的格式。

3.2.14　案例：创建自定义消息类型

在本案例中，将会创建一个用于生成自定义消息类型的功能包 json_interface。在 json_

interface 功能包中包含了一个话题消息类型、一个服务消息类型和一个动作消息类型，每种消息类型都是用字符串来传输的经过序列化后的 JSON 数据。json_interface 包的目的是简化自定消息类型的创建过程。

创建自定义消息类型目前不支持 Python 语言的功能包，需要使用 C++语言的功能包，使用 C++对 IDL 定义的消息进行编译。自定义的消息在编译完成后就能够像其他的消息类型一样被 Python 或 C++编写的节点使用。

首先，创建功能包 json_interface，将编译选项设置为 ament_cmake，命令如下：

```
ros2 pkg create --build-type ament_cmake --dependencies rosidl_default_
generators --license MIT json_interface
```

其次，在创建的功能包 json_interface 中创建 3 个子文件夹 msg、srv 和 action，并在各个目录中使用 IDL 定义消息的详细结构。

(1) 定义话题消息类型。在 msg 文件中创建一个 Jsonmsg.msg 文件，内容如下：

```
#a string represent JSON
string jsonstr
```

(2) 定义服务消息类型。在 srv 文件夹中创建一个 Jsonsrv.srv 文件，内容如下：

```
#client request is a string represent JSON
string reqjsonstr
---
#server response is a string represent JSON
string rspjsonstr
```

(3) 定义动作消息类型。在 acton 文件夹中创建一个 Jsonaction.action 文件，内容如下：

```
#action client request is a string represent JSON
string reqjsonstr
---
#server response is a string represent JSON
string rspjsonstr
---
#action server's feed back is a string represent JSON
string fbjsonstr
```

使用 ROS 2 的 IDL 语言创建好上述自义的话题、服务和动作消息类型后，整个 json_interface 功能包的结构如图 3-24 所示。

再次，修改功能包的 CMakeLists.txt 文件，将上述创建的 IDL 文件加入功能包编译过程中，向 CMakeLists.txt 文件添加以下内容：

图 3-24　json_interface 功能包的结构

```
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Jsonmsg.msg"
  "srv/Jsonsrv.srv"
  "action/Jsonaction.action"
)
```

最后，修改 package. xml 文件，在< package >元素内添加以下内容：

```
<member_of_group>rosidl_interface_packages</member_of_group>
```

完成上述操作后，就可以编译和安装功能包了，命令如下：

```
colcon build --packages-select json_interface
source install/setup.bash
```

上述命令运行后，就会将自定义功能包中的消息类型添加到 ROS 2 中，通过 ros2 interface package 命令即可查看功能包中的消息已经被 ROS 2 正确识别，如图 3-25 所示。

```
hw@hw-VirtualBox:~/ros2_ws$ ros2 interface package json_interface
json_interface/msg/Jsonmsg
json_interface/srv/Jsonsrv
json_interface/action/Jsonaction
```

图 3-25　编译消息功能包

在 Python 的功能包中使用自定义的消息时，与其他消息的使用方法相同。只需从功能包中导入所需的消息，代码如下：

```
from json_interface import msg, srv, action
strmsg=msg.Jsonmsg()                                  #构造自定义话题消息
reqsrv=srv.Jsonsrv_Request()                          #构造自定义服务请求消息
goalreq=action.Jsonaction_SendGoal_Request()          #构造自定义动作目标请求消息
```

以上对 rclpy 库中重要的类及其使用方法进行了介绍，并通过以上多个案例对 ROS 2 中的通信机制进行了示范，这些案例可以作为模板，用于机器人实际节点的编写。

52min

3.3 坐标系管理

运动是机器人最显著的特征,具体体现为位置的变化和自身姿态的变化。机器人的位置和姿态信息对于机器人具有重要的作用。在导航中,机器人知道自身的准确位置和方向是正确导航的前提;在机械臂抓取物品时各个关节处于准确的角度是抓取物品的前提;在四足机器人运动中,机器人各关节处于正确的角度是机器人保持稳定的前提。能够快速、精确地进行位置和姿态求解在机器人中就变得十分重要了。ROS 2 提供了一个功能包 tf2,用于管理整个系统中的坐标系,按照坐标系间的相对关系,将所有的坐标系构建为一棵 tf 树。通过 tf 树可以高效地查询到 ROS 2 中任意两个坐标系间的变换关系,实现高效的坐标变换。图 3-26 展示了一个机器人上的多个坐标系。

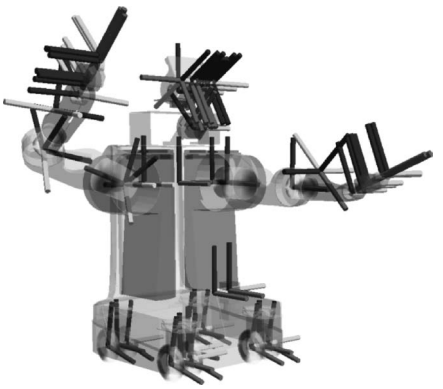

图 3-26 机器人上的多个坐标系

3.3.1 坐标变换原理

为了描述对象在空间中的位置和姿态,通常需要借助坐标系定位。在三维空间中,一般使用 3 个轴互相垂直的直角坐标系,3 个轴的关系要符合右手定则。直角坐标系中的 3 个轴的交点称为坐标系原点,简称原点。

右手定则的含义是将右手四指握拳,大拇指向上,大拇指指尖向上的方向为 z 轴正方向,四指从指根到指尖的旋转方向就是 x 轴到 y 轴的旋转方向,只需确定 x 轴和 y 轴中的一个,另一个就可以由旋转关系得到,如图 3-27 所示。

在符合右手定则的条件下,空间中直角坐标系的建立是任意的,也就是说坐标系原点的位置随意,3 个轴的朝向也随意。在空间中建立了两个坐标系 A 和 B,这两个坐标系没有优劣,如图 3-28 所示。通常在一个真实机器人世界中存在许多个坐标系。

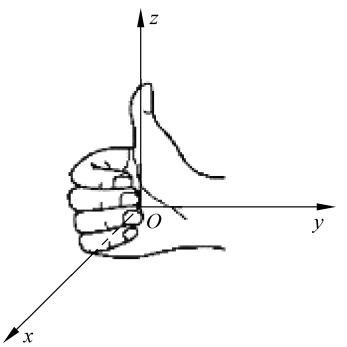

图 3-27 右手定则

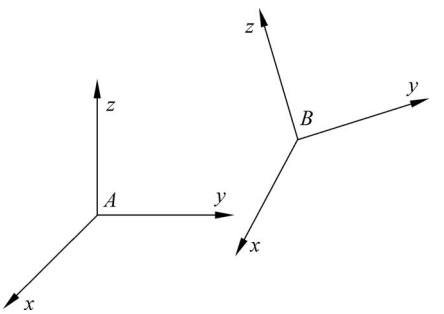

图 3-28 坐标系示例

当空间中有许多坐标系时就存在一个严肃的问题：如何描述同一个物体在不同坐标系下的坐标和姿态，也就是如何建立同一个点在不同坐标系下坐标间的变换关系，例如，两人相约到某处，A 说需要乘 3 站公交，B 说我需要乘 5 站公交，那么这个地点在 A 的坐标看来就是 3，而在 B 的坐标看来就是 5。与上述问题等价的两个问题：

(1) 不同坐标系间的关系。

(2) 一个物体在指定坐标系中经过旋转和平移运动后新的位置和姿态。

坐标变换原理给出了上述问题的解答。下面以不同坐标系间的关系为切入点，介绍坐标变换原理。

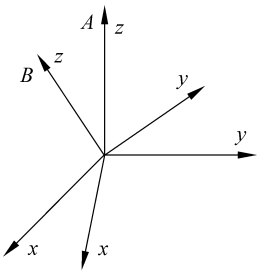

图 3-29　坐标系间的旋转

空间中坐标系间的关系可以分解为旋转和平移。首先介绍坐标系间的旋转变换关系，如图 3-29 所示，当坐标系 A 和坐标系 B 的原点重合时，两个坐标系只具有旋转关系，也就是坐标系 A 绕空间中某条直线旋转一定角度后会与坐标系 B 重合。坐标系 B 由其 3 个轴上的单位向量($\boldsymbol{x}_B$，$\boldsymbol{y}_B$，$\boldsymbol{z}_B$)所确定，将这 3 个单位向量分别投影到坐标系 A 的 3 个轴上(或者说这 3 个单位向量的坐标可以用 A 坐标系表示)，即

$$\begin{aligned} {}^A\boldsymbol{x}_B &= [\boldsymbol{x}_B \cdot \boldsymbol{x}_A, \boldsymbol{x}_B \cdot \boldsymbol{y}_A, \boldsymbol{x}_B \cdot \boldsymbol{z}_A]^{\mathrm{T}} \\ {}^A\boldsymbol{y}_B &= [\boldsymbol{y}_B \cdot \boldsymbol{x}_A, \boldsymbol{y}_B \cdot \boldsymbol{y}_A, \boldsymbol{y}_B \cdot \boldsymbol{z}_A]^{\mathrm{T}} \\ {}^A\boldsymbol{z}_B &= [\boldsymbol{z}_B \cdot \boldsymbol{x}_A, \boldsymbol{z}_B \cdot \boldsymbol{y}_A, \boldsymbol{z}_B \cdot \boldsymbol{z}_A]^{\mathrm{T}} \end{aligned} \tag{3-2}$$

其中，${}^A\boldsymbol{x}_B$ 为坐标系 B 的 x 轴上的单位向量在坐标系 A 下的坐标，${}^A\boldsymbol{y}_B$ 为坐标系 B 的 y 轴上的单位向量在坐标系 A 下的坐标，${}^A\boldsymbol{z}_B$ 为坐标系 B 的 z 轴上的单位向量在坐标系 A 的坐标，$\boldsymbol{x}_B \cdot \boldsymbol{x}_A$ 表示 B 坐标系 x 轴单位向量与 A 坐标系 x 轴单向量的内积，是单位向量投影后的坐标值，其余类似。

以上计算得到了坐标系 B 中的 3 个单位向量在坐标系 A 下的坐标。通过这些坐标就能够描述坐标系 B 与 A 的关系。一般将上述表示写为一个 3×3 的矩阵：

$${}^A_B\boldsymbol{R} = \begin{bmatrix} {}^A\boldsymbol{x}_B & {}^A\boldsymbol{y}_B & {}^A\boldsymbol{z}_B \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_B \cdot \boldsymbol{x}_A & \boldsymbol{y}_B \cdot \boldsymbol{x}_A & \boldsymbol{z}_B \cdot \boldsymbol{x}_A \\ \boldsymbol{x}_B \cdot \boldsymbol{y}_A & \boldsymbol{y}_B \cdot \boldsymbol{y}_A & \boldsymbol{z}_B \cdot \boldsymbol{y}_A \\ \boldsymbol{x}_B \cdot \boldsymbol{z}_A & \boldsymbol{y}_B \cdot \boldsymbol{z}_A & \boldsymbol{z}_B \cdot \boldsymbol{z}_A \end{bmatrix} \tag{3-3}$$

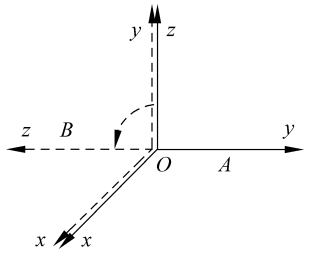

图 3-30　旋转矩阵练习

矩阵 ${}^A_B\boldsymbol{R}$ 通常称为坐标系 B 到坐标系 A 下的旋转矩阵，这个旋转矩阵的三列分别是坐标系 B 下 x、y 和 z 轴单位向量在坐标系 A 下的坐标值。使用该方法通常可用于求两个坐标系间的旋转矩阵。

例：写出图 3-30 中坐标系 B 到坐标系 A 的旋转矩阵 ${}^A_B\boldsymbol{R}$。

解：按照旋转矩阵的定义，求出坐标系 B 中各单位向量在 A 坐标系下的坐标：

由于坐标轴 x_B 与 x_A 重合，所以其在坐标系 A 下的坐标为

$$^{A}\boldsymbol{x}_B=[1,0,0]^{\mathrm{T}}$$

由于坐标轴 y_B 与 z_A 重合，所以其在坐标系 A 下的坐标为

$$^{A}\boldsymbol{y}_B=[0,0,1]^{\mathrm{T}}$$

由于坐标轴 z_B 在 y_A 的负方向，所以其在坐标系 A 下的坐标为

$$^{A}\boldsymbol{z}_B=[0,-1,0]^{\mathrm{T}}$$

将上述结果写入旋转矩阵中，得

$$^{A}_{B}\boldsymbol{R}=\begin{bmatrix}^{A}\boldsymbol{x}_B & ^{A}\boldsymbol{y}_B & ^{A}\boldsymbol{z}_B\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 0 & 0 & -1\\ 0 & 1 & 0\end{bmatrix}$$

实际对于坐标系 B 到坐标系的旋转矩阵 $^{A}_{B}\boldsymbol{R}$，还有以下两条性质：

(1) $^{A}_{B}\boldsymbol{R}$ 是正交矩阵。

(2) 坐标系 A 到 B 的旋转矩阵 $^{B}_{A}\boldsymbol{R}=^{A}_{B}\boldsymbol{R}^{-1}=^{A}_{B}\boldsymbol{R}^{\mathrm{T}}$。

对于空间中旋转的表示方式，除了使用旋转矩阵 $^{A}_{B}\boldsymbol{R}$ 外，常见还有四元数表示方法，以及欧拉转角 roll-pitch-yaw 表示方法。

其次，介绍坐标系间的平移关系。当两个坐标间只存在平移时，如图 3-31 所示。坐标系 B 与坐标系 A 的关系可以用 B 坐标系原点在 A 坐标系下的坐标进行描述。B 坐标系的原点 $^{A}\boldsymbol{P}_{\mathrm{Borg}}$ 在 A 坐标系下的坐标为

$$^{A}\boldsymbol{P}_{\mathrm{Borg}}=[x,y,z] \tag{3-4}$$

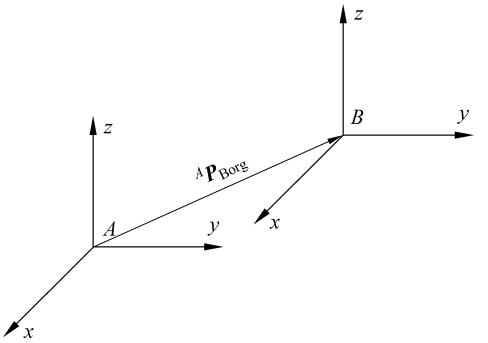

图 3-31　坐标系平移

最后，空间两个坐标系间的任意变换都可由旋转和平移组合而来，按照线性代数的写法，将平移和旋转可以写为一个 4×4 的变换矩阵 $^{A}_{B}\boldsymbol{T}$：

$$^{A}_{B}\boldsymbol{T}=\begin{bmatrix}^{A}_{B}\boldsymbol{R} & ^{A}\boldsymbol{P}_{\mathrm{Borg}}\\ 0\quad 0\quad 0 & 1\end{bmatrix} \tag{3-5}$$

变换矩阵 $^{A}_{B}\boldsymbol{T}$ 的前 3 行 3 列元素的值为坐标系间的旋转矩阵 $^{A}_{B}\boldsymbol{R}$ 的元素值，第 4 列的前 3 个元素的值为坐标系 B 原点在坐标系 A 下的坐标值。

下面给出变换矩阵的 3 个用途：

(1) 坐标系间的变换关系，例如，${}^{A}_{B}\boldsymbol{T}$ 描述了坐标系 B 和坐标系 A 之间的变换关系。

(2) 同一个点的坐标在不同坐标系下的变换，例如，将坐标系 B 下的点 $\boldsymbol{P}_B$ 的坐标转换为 A 坐标系下的坐标 $\boldsymbol{P}_A$，即 $\boldsymbol{P}_A = {}^{A}_{B}\boldsymbol{T}\,\boldsymbol{P}_B$。

(3) 在同一个坐标系中描述点在运动前后坐标值的关系。点根据变换矩阵 ${}^{A}_{B}\boldsymbol{T}$ 从坐标 $\boldsymbol{P}_A$ 运动到新坐标 $\boldsymbol{P}'_A$，则 $\boldsymbol{P}'_A = {}^{A}_{B}\boldsymbol{T}\,\boldsymbol{P}_A$。

一般来讲使用情况 1 来求出变换矩阵，而情况 2 和情况 3 则根据需求用来计算具体的变换。

在实际机器人情况中，坐标系的数量非常多。可以通过链式法则来求解任意两个坐标系间的变换，如图 3-32 所示。

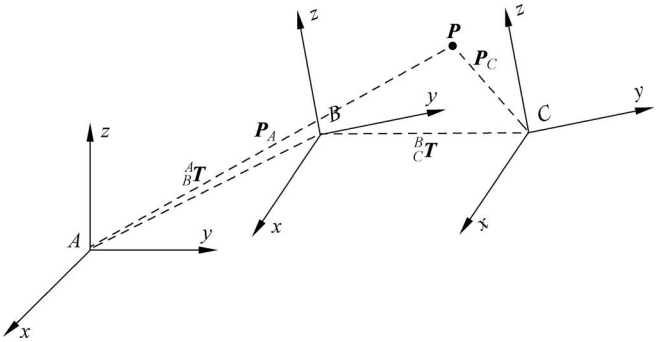

图 3-32　坐标系间的变换

在图 3-32 中有 3 个坐标系 A、B、C，其中坐标系 B 到坐标系 A 的变换为 ${}^{A}_{B}\boldsymbol{T}$，坐标系 C 到坐标系 B 的变换为 ${}^{B}_{C}\boldsymbol{T}$，则可求得坐标系 C 到坐标系 A 的变换为 ${}^{A}_{C}\boldsymbol{T} = {}^{A}_{B}\boldsymbol{T}\,{}^{B}_{C}\boldsymbol{T}$，从而可根据变换矩阵 ${}^{A}_{C}\boldsymbol{T}$ 直接计算在坐标系 C 下的点 $\boldsymbol{P}_C$ 在坐标系 A 下的坐标 $\boldsymbol{P}_A$ 为 $\boldsymbol{P}_A = {}^{A}_{C}\boldsymbol{T}\boldsymbol{P}_C$。

按照上述坐标系间变换的链式法则，就可以将整个空间中的所有坐标系关联起来，从而可以求得任意两个坐标系间的变换关系，完成坐标间的变换。鉴于坐标系在机器人中的重要性，ROS 2 的 TF2 功能包就专门用于管理和维护整个程序中的坐标系。

3.3.2　TF2 简介

ROS 2 中的 TF2 功能包基于坐标系变换的链式法则对多个坐标系进行管理和维护。为了保证一致性和实用性，TF2 使用树结构来管理坐标系而非图结构，具体来讲，通过选择一个根坐标系(世界坐标系)作为基准，按照坐标系间的变换关系逐渐扩展成一棵由坐标系构成的树，树中任意两个节点(坐标系间)都存在唯一路径，根据链式法则从而可方便地完成坐标系间的变换。

TF2 对原 TF 进行了改进和优化，为了与之前版本区分被称为 TF2。TF2 主要包含两个发布坐标系的功能和接收存储坐标系的功能。为应对随时间变换的坐标系，TF2 能够接收和处理异步的坐标系信息，并对延时和坐标系丢失具有很强的稳健性。

TF2 在广播坐标系时使用话题通信机制,以话题名/tf 进行广播 tf 树。

在 ROS 2 功能包 tf2_ros 和 tf2_tool 提供了有关坐标系的变换功能。两个功能包一般会在 ROS 2 安装时进行安装,也可以以功能包的方式安装,命令如下:

```
sudo apt install ros-jazzy-tf2-ros
sudo apt install ros-jazzy-tf2-tools
```

下面对这两个功能包的功能和使用方法进行介绍。

(1) tf2_ros 功能包提供了 4 个可执行的节点 buffer_server、tf2_echo、tf2_monitor 和 static_transform_publisher。buffer_server 用于启动一个缓存 TF 坐标系的服务器。tf2_echo 用于进行查询坐标系间的变换。static_tranform_publisher 向 TF2 注册一个静态坐标系,静态坐标系就是不随时间变化的坐标系,例如表示环境中的固定障碍物的位置。tf2_monitor 用于监听和显示 TF2 中活动的坐标系。

下面通过查询坐标系 A 和坐标系的关系来说明 tf2_ros 中各节点的使用方法。

首先,启动 buffer_server,命令如下:

```
ros2 run tf2_ros buffer_server
```

其次,发布两个以世界坐标系 World 为基的坐标系 A 和坐标系 B,命令如下:

```
ros2 run tf2_ros static_transform_publisher --frame-id World --child-frame-id A
ros2 run tf2_ros static_transform_publisher --frame-id World --child-frame-id
B --x 3 --y 4 --z 5
```

以上两个命令发布了坐标系 A 与坐标系 B,坐标系 A 与世界坐标系 World 重合,坐标系 B 位于世界坐标系 World 的(3,4,5)处,坐标轴与世界坐标系 World 相同。此外还可以通过--qx、--qy、--qz、--qw 参数以四元数的格式添加坐标系的旋转,或通过--roll、--pitch、--yaw 参数以欧拉转角的格式添加坐标系的旋转。

再次,查看发布的坐标系 A 和坐标系 B,命令如下:

```
ros2 run tf2_ros tf2_monitor
```

上述命令的运行效果如图 3-33 所示,经过 10s 的坐标系收集后,正确地显示了以上发布的坐标系 A 和坐标系 B。

最后,查询坐标系 B 到坐标系 A 的变换,命令如下:

```
ros2 run tf2_ros tf2_echo A B
```

在上述命令中,在 TF2 中将参数值为 A 所在的坐标系称为 source frame,而将参数值为 B 所在的坐标系称为 target frame。上述命令的运行效果如图 3-34 所示,虽然在发布坐标系时只发布了坐标系 A 和 B 与 World 的关系,在查询坐标系 B 到 A 的变换时,TF2 会自动根据 tf 树计算并得到变换关系,tf2_echo 显示了平移、四元数格式的旋转、欧拉转角

```
hw@hw-VirtualBox:~$ ros2 run tf2_ros tf2_monitor
Gathering data on all frames for 10 seconds...



RESULTS: for all Frames

Frames:
Frame: A, published by <no authority available>, Average Delay: 22.4384, Max Del
ay: 22.4384
Frame: B, published by <no authority available>, Average Delay: 406.446, Max Del
ay: 406.446

All Broadcasters:
Node: <no authority available> 346.136 Hz, Average Delay: 214.442 Max Delay: 406
.446
```

图 3-33　坐标系监测

RPY 格式的旋转，以及变换矩阵等格式的变换信息。

```
hw@hw-VirtualBox:~$ ros2 run tf2_ros tf2_echo A B
[INFO] [1723710082.755507406] [tf2_echo]: Waiting for transform A ->  B: Invalid
 frame ID "A" passed to canTransform argument target_frame - frame does not exis
t
At time 0.0
- Translation: [3.000, 4.000, 5.000]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
- Rotation: in RPY (radian) [0.000, -0.000, 0.000]
- Rotation: in RPY (degree) [0.000, -0.000, 0.000]
- Matrix:
  1.000  0.000  0.000  3.000
  0.000  1.000  0.000  4.000
  0.000  0.000  1.000  5.000
  0.000  0.000  0.000  1.000
```

图 3-34　查询坐标系间的变换关系

(2) tf2_tools 功能包提供了一个绘制当前系统 tf 树的程序 view_frames。该程序先监听系统的 tf 树 5s，然后根据监听结构将 tf 树绘制到一个 PDF 文件中并保存到当前目录。

绘制 tf 树，命令如下：

```
ros2 run tf2_tools view_frames
```

上述命令执行完成后会在当前目录下生成一个以 frames 开头的 PDF 文件。打开该 PDF 文件后会显示命令运行时系统中的 tf 树，如图 3-35 所示。

此外，也可以使用 RViz 来直观地对系统中的坐标系进行可视化，显示坐标系在三维空间中的关系。具体步骤如下。

首先，启动 RViz，命令如下：

```
rviz2
```

其次，将全局设置(Global Options)下的固定坐标系(Fixed Frame)设置为 World，添加 TF 可视化功能后，在三维可视化场景中即可可视化系统中的 3 个坐标系 World、A 和 B，如图 3-36 所示。

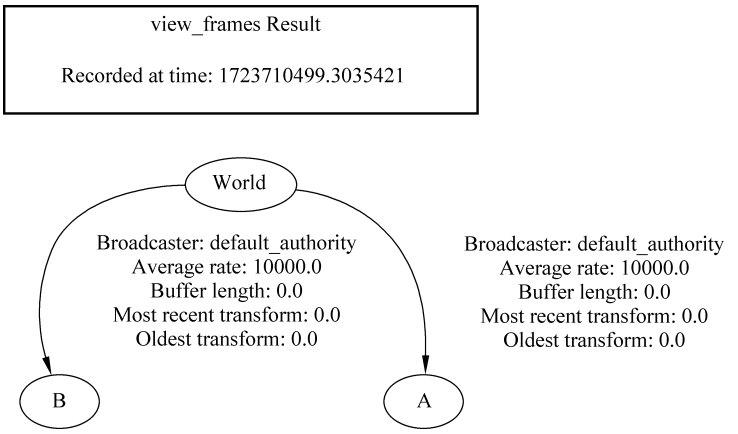

图 3-35　tf 树可视化

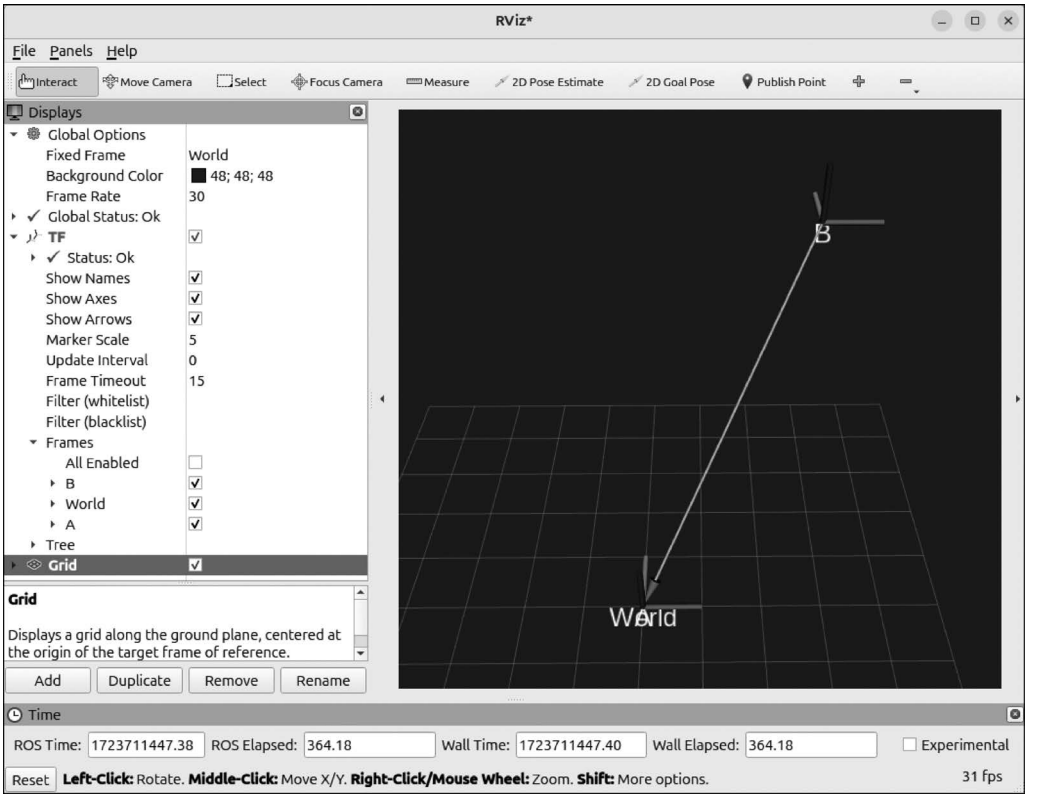

图 3-36　坐标系可视化

除了以上 TF2 命令行工具外，tf2_ros 功能包也提供了 Python 语言的编程接口，提供了发布静态坐标系、动态坐标系和查询坐标变换功能，从而以更灵活的方式在自定义的节点中处理坐标系。下面通过 3 个案例详细说明 tf2_ros 功能包的编程接口的使用方法。

3.3.3　案例：发布静态坐标系

机器人运行时的固定的障碍物、建筑物等坐标都是不随时间发生变化的，只需发布一次坐标信息向 TF2 注册坐标。tf2_ros 功能包中的 static_transform_broadcaster.StaticTransformBroadcaster 类提供了静态坐标发布功能。功能包 geometry_msgs 中的 msg.TransormStamped 提供了坐标系的消息类型。

首先，在工作空间中创建一个名为 tf_test 的功能包，用于存放案例的代码，命令如下：

```
ros2 pkg create --build-type ament_python --node-name staticframepub
--dependencies geometry_msgs tf2_ros --license MIT tf_test
```

其次，在功能包中的 staticframepub.py 文件中编写一个发布静态坐标系节点，代码如下：

```
#ros2_ws3/src/tf_test/tf_test/staticframepub.py
import rclpy
import math
from geometry_msgs.msg import TransformStamped
from rclpy.node import Node
from tf2_ros.static_transform_broadcaster import StaticTransformBroadcaster

def quaternion_from_euler(ai, aj, ak):
    #将欧拉角转换为四元数表示
    ai /= 2.0
    aj /= 2.0
    ak /= 2.0
    ci = math.cos(ai)
    si = math.sin(ai)
    cj = math.cos(aj)
    sj = math.sin(aj)
    ck = math.cos(ak)
    sk = math.sin(ak)
    cc = ci*ck
    cs = ci*sk
    sc = si*ck
    ss = si*sk

    q = [0]*4
    q[0] = cj*sc - sj*cs
    q[1] = cj*ss + sj*cc
    q[2] = cj*cs - sj*sc
    q[3] = cj*cc + sj*ss
    return q

class StaticFramePublisher(Node):
    """
    静态坐标系发布器
```

```
    """
    def __init__(self):
        super().__init__('static_frame_tf2_broadcaster')
        self.x=self.declare_parameter('x', 3.0).get_parameter_value().double_value
        self.y=self.declare_parameter('y', 4.0).get_parameter_value().double_value
        self.z=self.declare_parameter('z', 5.0).get_parameter_value().double_value
        self.roll=self.declare_parameter('roll', 0.0).get_parameter_value().
double_value
        self.pitch=self.declare_parameter('pitch', 0.0).get_parameter_value().
double_value
        self.yaw=self.declare_parameter('yaw', 0.0).get_parameter_value().
double_value
        self.child_frame=self.declare_parameter('child_frame',
                    'frame_A').get_parameter_value().string_value
        self.parent_frame=self.declare_parameter('parent_frame',
                      'World').get_parameter_value().string_value
        self.tf_static_broadcaster = StaticTransformBroadcaster(self)
        self.logger=self.get_logger()
        self.pub_transforms()

    def pub_transforms(self):
        t = TransformStamped()
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = self.parent_frame
        t.child_frame_id = self.child_frame
        t.transform.translation.x = self.x
        t.transform.translation.y = self.y
        t.transform.translation.z = self.z
        quat = quaternion_from_euler( self.roll/180*math.pi,
                                      self.pitch/180*math.pi,
                                      self.yaw/180*math.pi)
        t.transform.rotation.x = quat[0]
        t.transform.rotation.y = quat[1]
        t.transform.rotation.z = quat[2]
        t.transform.rotation.w = quat[3]
        self.logger.info(f'publish static tf: {t}')
        self.tf_static_broadcaster.sendTransform(t)

def main(args=None):
    rclpy.init(args=args)
    node = StaticFramePublisher()
    try:
        rclpy.spin(node)
    except Exception:
        rclpy.shutdown()
        exit(0)

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

在上述代码中，创建了一个发布静态坐标系的节点 StaticFramePublisher，在节点的初始化方法__init__()中定义了发布坐标系的一些参数，并且创建了一个静态坐标系发布者 StaticTransformBroadcaster 类的实例，最后调用了 pub_transforms()方法进行坐标的发布。

在 pub_transforms()方法中，创建了一个坐标系消息 TransformStamped 的实例，根据节点设置的参数设置坐标系，其中 quaternion_from_euler()函数实现了将欧拉转角变换为四元数的功能，最后通过静态坐标系发布者的 sendTransform()方法发布静态坐标系。

注意：坐标系消息 TransformStamped 的详细结构可以通过 ros2 interface 命令行工具查询，命令为 ros2 interface show geometry_msgs/msg/TransformStamped。

经过编译、安装后，即可运行上述编写的静态坐标系发布节点。下面打开两个终端，分别发布坐标系 frame_A 和坐标系 frame_B，命令如下：

```
ros2 run tf_test staticframepub --ros-args -p x:=1.0 -p y:=0.0 -p z:=0.0 -p yaw:=
90.0 -p child_frame:=frame_A
ros2 run tf_test staticframepub --ros-args -p x:=1.0 -p y:=0.0 -p z:=0.0 -p yaw:=
90.0 -p child_frame:=frame_B -p parent_frame:=frame_A
```

上述第 1 条命令发布了一个从坐标系 frame_A 到默认坐标系 World 的变换，第 2 条命令发布了一个从坐标系 frame_B 到坐标系 frame_A 的变换。上述命令的执行效果如图 3-37 所示。

图 3-37　发布静态坐标系

使用 RViz 可直观地显示上述静态坐标系的发布结果，可视化坐标系间的关系，如图 3-38 所示。

3.3.4　案例：发布动态坐标系

机器人的关节是活动的，位置和角度会随时间的变化而变化，设置在其上的坐标系会不

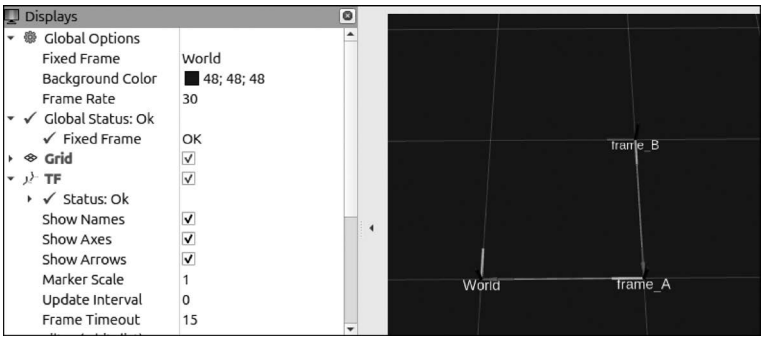

图 3-38　静态坐标系可视化

断地发生变化。tf2_ros 功能包中的 TransformBroadcaster 类提供了动态坐标的发布功能。消息类型同样使用功能包 geometry_msgs 中的 msg. TransormStamped。

首先,在功能包 tf_test 内创建一个名为 framepub. py 的文件,用于存放动态坐标系发布节点的代码。

其次,在 framepub. py 文件中编写一个发布动态坐标系节点,代码如下:

```
#ros2_ws3/src/tf_test/tf_test/framepub.py
import rclpy
import math
from geometry_msgs.msg import TransformStamped
from rclpy.node import Node
from tf2_ros import TransformBroadcaster

def quaternion_from_euler(ai, aj, ak):
    #将欧拉角转换为四元数表示
    ai /= 2.0
    aj /= 2.0
    ak /= 2.0
    ci = math.cos(ai)
    si = math.sin(ai)
    cj = math.cos(aj)
    sj = math.sin(aj)
    ck = math.cos(ak)
    sk = math.sin(ak)
    cc = ci *ck
    cs = ci *sk
    sc = si *ck
    ss = si *sk

    q = [0]*4
    q[0] = cj *sc - sj *cs
    q[1] = cj *ss + sj *cc
    q[2] = cj *cs - sj *sc
    q[3] = cj *cc + sj *ss
```

```
    return q

class FramePublisher(Node):
    """
    动态坐标系发布器
    """
    def __init__(self):
        super().__init__('frame_tf2_publisher')
        self.tf_broadcaster = TransformBroadcaster(self)
        self.num=0
        self.create_timer(0.03,self.pub_transforms)

    def pub_transforms(self):
        t = TransformStamped()
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'frame_A'
        t.child_frame_id = 'frame_B'

        theta=self.num/180*math.pi
        t.transform.translation.x = math.cos(theta)*2
        t.transform.translation.y = math.sin(theta)*2
        t.transform.translation.z = 0.0

        quat = quaternion_from_euler( 0, 0, theta+math.pi/2)
        t.transform.rotation.x = quat[0]
        t.transform.rotation.y = quat[1]
        t.transform.rotation.z = quat[2]
        t.transform.rotation.w = quat[3]
        self.num+=1
        if self.num>=360:
            self.num=0
        self.tf_broadcaster.sendTransform(t)

def main(args=None):
    rclpy.init(args=args)
    node = FramePublisher()
    try:
        rclpy.spin(node)
    except Exception:
        rclpy.shutdown()
        exit(0)

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

在上述代码中创建了一个发布动态坐标系的类 FramePublisher。在类初始化方法__init__()中创建了一个坐标系发布者 TransformBroadcaster 类的实例，并创建了一个定时器用于周期性地发布坐标系。定时器回调函数 pub_transforms()用于发布从坐标系 frame_B

到 frame_A 的坐标系变换，frame_B 的坐标系的 x 和 y 会随时间作余弦和正弦运动。

再次，在功能包的 setup.py 文件中将该节点添加为 framepub，代码如下：

```
'framepub = tf_test.framepub:main',
```

最后，经过编译、安装后，即可运行上述编写的动态坐标系发布节点。在终端运行节点，命令如下：

```
ros2 run tf_test framepub
```

在 RViz 中可视化坐标系，可以看到坐标系 frame_B 围绕坐标系 frame_A 作圆周运动，如图 3-39 所示。

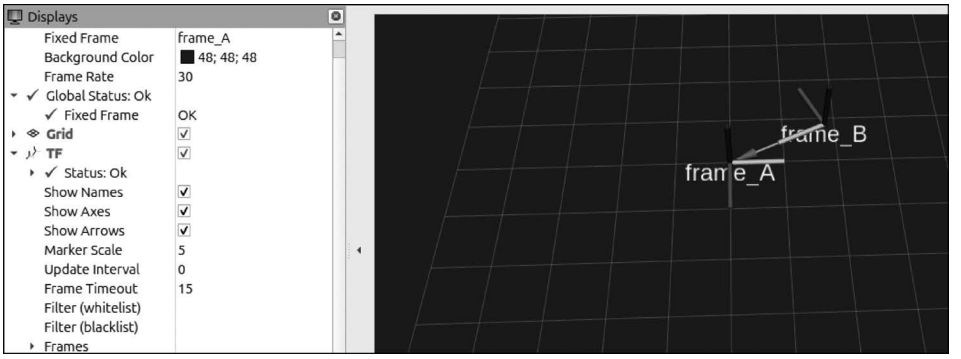

图 3-39 动态坐标系可视化

3.3.5 案例：查询坐标系变换

TF2 将系统中的所有坐标系构建为一棵树，只要查询这棵树就能获得树上任意两个坐标系的变换。tf2_ros 功能包的 transform_listener.TransformListener 类提供了监听坐标系功能。

首先，在功能包 tf_test 内创建一个名为 framesub.py 的文件，用于存放接收坐标系节点的代码。

其次，在 framesub.py 文件中编写一个接收坐标系并且显示的节点，代码如下：

```
#ros2_ws3/src/tf_test/tf_test/framesub.py
import rclpy
from rclpy.node import Node
from tf2_ros.buffer import Buffer
from tf2_ros.transform_listener import TransformListener

class FrameListener(Node):
    def __init__(self):
        super().__init__('tf2_frame_listener')
        self.logger=self.get_logger()
```

```
        self.target_frame = self.declare_parameter(
          'target_frame', 'frame_B').get_parameter_value().string_value
        self.source_frame = self.declare_parameter(
          'source_frame', 'frame_A').get_parameter_value().string_value

        self.tf_buffer = Buffer()
        self.tf_listener = TransformListener(self.tf_buffer, self)
        self.logger=self.get_logger()
        self.timer = self.create_timer(1.0, self.show_tf)

    def show_tf(self):
        try:
            t = self.tf_buffer.lookup_transform(
                self.source_frame,
                self.target_frame,
                rclpy.time.Time()
            )
             self.logger.info(f"parent frame:{t.header.frame_id}, child frame:
{t.child_frame_id}, transform is {t.transform}")
        except Exception as e:
            print(e)
            return


def main(args=None):
    rclpy.init(args=args)
    node = FrameListener()
    try:
        rclpy.spin(node)
    except Exception:
        rclpy.shutdown()
        exit(0)

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

在上述代码中创建了一个查询坐标系的类 FrameListener。在类初始化方法__init__()中创建了两个查询坐标系变换的参数 source_frame 和 target_frame，创建了一个存放坐标系缓存的 Buffer()容器，创建了一个坐标系接收者 TransformListener，以及一个用于定期查询坐标系的定时器。定时器回调函数 show_tf()使用节点内缓存的 tf 树的 lookup_transform()方法查询给定坐标系间的变换关系，查询成功后在终端显示查询结果。

再次，在功能包的 setup.py 文件中添加的该节点为 framepub，代码如下：

```
'framesub = tf_test.framesub:main',
```

最后，经过编译、安装后，即可运行上述编写的查询坐标系节点。在终端运行节点，命令如下：

```
ros2 run tf_test framesub
```

上述命令的运行效果如图 3-40 所示,每隔 1s,就会查询一次坐标系 frame_B 到坐标系 frame_A 的变换关系。需要注意的是运行此命令前,应当运行上个案例中的 famepub 节点以发布坐标系的变换。

```
hw@hw-VirtualBox:~/ros2_ws$ ros2 run tf_test framesub
[INFO] [1723779353.261741920] [tf2_frame_listener]: parent frame:frame_A, child
frame:frame_B, transform is geometry_msgs.msg.Transform(translation=geometry_msg
s.msg.Vector3(x=-1.9996953903127825, y=0.03490481287456688, z=0.0), rotation=geo
metry_msgs.msg.Quaternion(x=0.0, y=0.0, z=-0.7132504491541817, w=0.7009092642998
509))
[INFO] [1723779354.249696886] [tf2_frame_listener]: parent frame:frame_A, child
frame:frame_B, transform is geometry_msgs.msg.Transform(translation=geometry_msg
s.msg.Vector3(x=-1.6960961923128521, y=-1.0598385284664096, z=0.0), rotation=geo
metry_msgs.msg.Quaternion(x=0.0, y=0.0, z=-0.48480962024633717, w=0.874619707139
3957))
```

图 3-40　坐标系变换查询结果

坐标系变换是机器人中常用的功能,ROS 2 的 TF2 提供了坐标系的管理功能,TF2 的命令行工具提供了对 tf 树的便捷操作,此外编程接口为自定义节点提供坐标系的发布和为查询提供了相关方法,最后,以 3 个代码案例介绍了 TF2 编程接口的使用方法,可作为实际机器人坐标系使用和管理时的参考模板。

3.4　Launch 文件

一个完整机器人程序通常需要多个节点同时运行以便相互配合及完成复杂任务。在命令行中逐个节点启动效率低且烦琐,ROS 2 中的 Launch 文件的一个最主要功能就是提供了一次性启动多个节点的方法,解决了上述多节点的启动问题。

3.4.1　Launch 文件简介

Launch 文件用于描述系统启动时的配置,用于控制和管理整个 ROS 2 程序的启动。在 Launch 文件里可以设置运行哪些节点、在哪里运行、向节点传递哪些参数,以及一些 ROS 2 特有的配置。此外,Launch 文件在运行后还负责监控启动进程的状态,并对这些进程状态的变化做出反应。

一个 Launch 文件主要包含节点和参数两部分。节点定义了要启动的 ROS 2 节点,包括节点的名称、包名、执行文件路径等信息。参数用于设置节点的参数,可以是命令行参数、ROS 2 参数服务器的参数,或者私有命名空间内的参数。

具体来讲,Launch 文件主要有以下几个功能:

(1) 设置命令行参数及默认值。

(2) 包含另一个 Launch 文件。

(3) 在另一个命名空间中包含另一个启动文件。

(4) 启动一个节点,设置其命名空间,以及设置该节点参数。

(5) 创建一个节点,将消息从一个主题重映射到另一个主题。

在 ROS 2 中使用 Launch 文件主要有以下几个优点。

(1) 简化启动过程: Launch 文件可以将复杂的启动过程编排为一个文件。这样,用户只需运行一个 Launch 文件,就可以启动整个 ROS 2 系统中的多个节点,而无须逐个手动启动每个节点。

(2) 统一配置管理: Launch 文件可以集中管理所有节点的配置和参数,使系统的配置更加清晰和可维护。通过修改 Launch 文件中的参数,可以轻松地改变节点的行为,而无须修改每个节点的启动命令。

(3) 便于复用和分享: Launch 文件可以被复制、修改和分享,使其他开发人员可以轻松地部署同样的 ROS 2 系统,或者将自己的系统部署到不同的环境中。

(4) 支持条件和组合启动: Launch 文件支持条件判断、组合启动和命名空间配置,能够根据需要动态地加载节点,并且可以为每个节点设置不同的参数和运行环境。

(5) 提高效率和可靠性: 通过 Launch 文件,可以快速地启动整个 ROS 2 系统,减少手动操作的错误和复杂度,提高了系统的启动效率和可靠性。

(6) 集中管理和调试: Launch 文件使系统的配置和调试更加集中和直观,开发人员可以快速地定位问题并进行调整。

(7) 自动化部署: 在自动化测试和部署中,Launch 文件能够以编程方式定义系统的启动流程,实现持续集成和持续部署。

相较于 ROS 1 中的 Launch 文件,ROS 2 对 Launch 文件进行了增强和改进,提供了 Python、XML 和 YAML 共 3 种格式的 Launch 文件规范。XML 格式的 Launch 文件保持与 ROS 1 中的 Launch 文件相同,以保持兼容性。Python 和 YAML 格式为 ROS 2 中新增的 Launch 文件格式。

下面展示了使用 Python、XML 和 YAML 这 3 种不同格式编写的具有相同功能的 Launch 文件。

Python 格式的 Launch 文件,代码如下:

```
#ros2_ws3/test_launch.py
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
```

```
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

XML 格式的 Launch 文件,代码如下:

```
#ros2_ws3/test_launch.xml
<launch>
  < node  pkg =" turtlesim "  exec =" turtlesim _ node "  name =" sim "  namespace =
"turtlesim1"/>
  < node  pkg =" turtlesim "  exec =" turtlesim _ node "  name =" sim "  namespace =
"turtlesim2"/>
  <node pkg="turtlesim" exec="mimic" name="mimic">
    <remap from="/input/pose" to="/turtlesim1/turtle1/pose"/>
    <remap from="/output/cmd_vel" to="/turtlesim2/turtle1/cmd_vel"/>
  </node>
</launch>
```

YAML 格式的 Launch 文件,代码如下:

```
#ros2_ws3/test_launch.yaml
launch:

- node:
    pkg: "turtlesim"
    exec: "turtlesim_node"
    name: "sim"
    namespace: "turtlesim1"

- node:
    pkg: "turtlesim"
    exec: "turtlesim_node"
    name: "sim"
    namespace: "turtlesim2"

- node:
```

```
        pkg: "turtlesim"
        exec: "mimic"
        name: "mimic"
        remap:
        -
            from: "/input/pose"
            to: "/turtlesim1/turtle1/pose"
        -
            from: "/output/cmd_vel"
            to: "/turtlesim2/turtle1/cmd_vel"
```

将上述3个Launch文件分别保存为test_launch.py、test_launch.xml、test_launch.yaml。

Launch文件有两种启动方式，一种是将Launch文件作为功能包的一部分，命令格式如下：

```
ros2 launch <package_name> <launch_file_name>
```

其中，<package_name>为功能包名，<launch_file_name>为Launch文件名。

另一种是将Launch文件作为独立的启动文件，与任何功能包无关，命令格式如下：

```
ros2 launch <path_to_launch_file>
```

其中，<path_to_launch_file>为Launch文件的路径，例如，运行上述3种格式示例Launch文件的命令如下：

```
ros2 launch test_launch.py
ros2 launch test_launch.xml
ros2 launch test_launch.yaml
```

上述3个命令具有相同的功能，在运行后都会启动两个小海龟仿真窗口。

由于Python格式的Launch文件以Python语言编写，可以借助Python语言来完成许多其他两种Launch文件无法完成的“魔法”操作，具有很强的适应性和灵活性，因此使用Python格式的Launch文件成为编写Launch文件的首选项，以下介绍使用Python语言编写Launch文件的方法。

3.4.2　常用类介绍

Python的第三方库launch_ros提供了配置Launch文件的相关API。以下介绍launch_ros库中编写Launch文件常用的API。

1. LaunchDescription类

LaunchDescription是一个Python类，用于组织和描述Launch文件中的各个启动操作。它可以包含参数、其他Launch文件和ROS 2节点。LaunchDescription对象允许将多

个启动操作组合在一起,定义它们之间的依赖关系和顺序。以下是 LaunchDescription 的一些重要特性和用法。

(1) 组织和组合启动操作:可以使用 LaunchDescription 来组织和描述 Launch 文件中的各个启动操作,例如启动节点、加载参数文件、设置环境变量等。通过添加不同类型的启动操作,可以构建出完整的启动配置。

(2) 设置启动操作的依赖关系:使用 add_action() 方法将启动操作添加到 LaunchDescription 对象中,并设置它们之间的依赖关系。这样可以确保在启动时按照正确的顺序执行各个操作。

(3) 设置全局选项:LaunchDescription 对象还可以设置全局选项,例如设置 Launch 文件的名称、描述、命名空间等。这些选项可以影响所有的启动操作。

(4) 启动操作的执行方式:可以通过 LaunchDescription 设置启动操作的执行方式,例如设置启动操作是否在新的命名空间中执行,以及是否在新的 Shell 中执行等。

(5) 启动文件的编程接口:可以通过编写 Python 脚本来创建和配置 LaunchDescription 对象,以编程方式生成 Launch 文件。这样可以更灵活地处理各种复杂的启动配置需求。

2. Node 类

在 ROS 2 中使用 Python 编写 Launch 启动文件时,经常会用到 launch_ros.actions.Node 类。该类允许通过 Launch 文件启动 ROS 2 功能包中的一个节点,它提供了一种简单的方式来配置和启动节点,可以指定节点的名称、命名空间、输出方式、参数等,例如,创建一个节点实例,指定该节点的各种属性和配置选项,代码如下:

```
node = Node(
    package='my_package',
    executable='my_node_executable',
    name='my_node_name',
    namespace='my_namespace',
    output='screen',
    parameters=[{'my_param': 'value'}],
    remappings=[('/original_topic', '/new_topic')],
    arguments=['--my_argument', 'value']
)
```

上述构造方法的参数的前两项为必填项,后面各项均为可选项,各参数的含义如下。

(1) package:一个字符串,表示节点所在的 ROS 2 包的名称。

(2) executable:一个字符串,表示在给定包中的可执行文件的名称。

(3) name:可选字符串,用于指定节点的名称。如果未指定,则将使用可执行文件的名称。

(4) namespace:可选字符串,用于指定节点的命名空间。命名空间是一种组织节点的方式,可以帮助避免节点名称发生冲突。

（5）output：可选字符串，用于指定节点的输出应如何处理，例如，可以将其设置为"screen"，以便将节点的输出打印到屏幕上。

（6）parameters：可选列表，用于指定节点的参数。每个参数都是一个字典，包含参数的名称和值。

（7）remappings：可选列表，用于指定话题的重新映射。每个重新映射是一个元组，包含原始话题的名称和新话题的名称。

（8）arguments：可选列表，用于指定传递给可执行文件的命令行参数。

在上述变量中，parameters 指的是节点参数，即在节点上定义的参数，而 argument 指的是实际参数，即在调用函数时传递的具体数值。

3.4.3　案例：Launch 文件编写

以下将使用 Python 语言介绍向功能包添加 Launch 文件的方法，并说明参数设置、节点启动配置等功能的用法。

（1）创建 Launch 文件。Launch 文件按照约定应当放置在功能包的 launch 目录下。首先进入功能包 test_package_python 中，创建一个名为 launch 的文件夹，在文件夹内创建一个名为 test_launch.py 的文件。在 launch 文件夹内打开终端，创建 launch 文件夹，如图 3-41 所示，命令如下：

```
mkdir -p launch
cd launch
touch test_launch.py
cd ..
code .
```

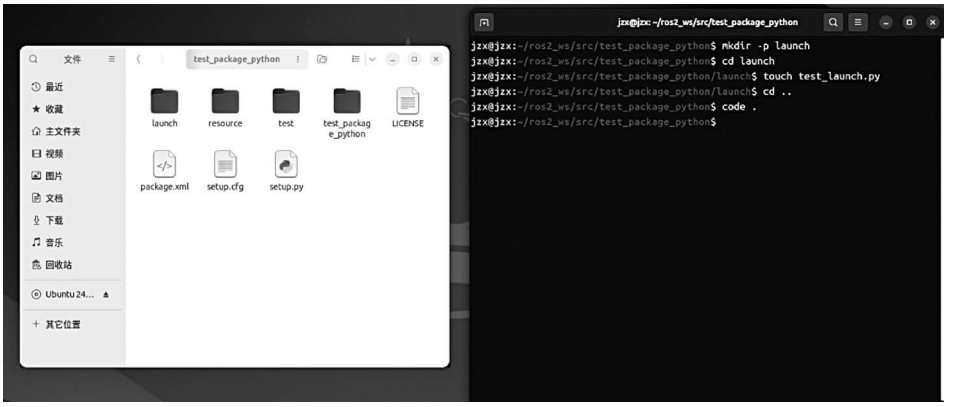

图 3-41　创建 Launch 文件

（2）编写 Launch 文件。此处先以启动一个 turtlesim 节点为例介绍 Launch 文件的基本结构，后续再逐渐添加节点和参数以完善 Launch 文件。在 launch 文件夹的 test_launch.py 文件中编写代码如下：

```
#ros2_ws3/src/test_package_python/launch/test_launch.py
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    ld = LaunchDescription()

    node = Node(package='turtlesim', executable='turtlesim_node' )

    ld.add_action(node)

    return ld
```

在上述代码中函数 generate_launch_description()为 Launch 文件所必需的函数,需要返回一个 LaunchDescription 类的实例。在函数 generate_launch_description()内首先创建了一个 LaunchDescription 类的实例 ld,然后创建了一个启动 turtlesim 包中 turtlesim_node 程序的节点 node,最后将创建的节点 node 添加到 LaunchDescription 类的实例 ld 中并返回。图 3-42 显示了在 VS Code 中编写 Launch 文件的结果。

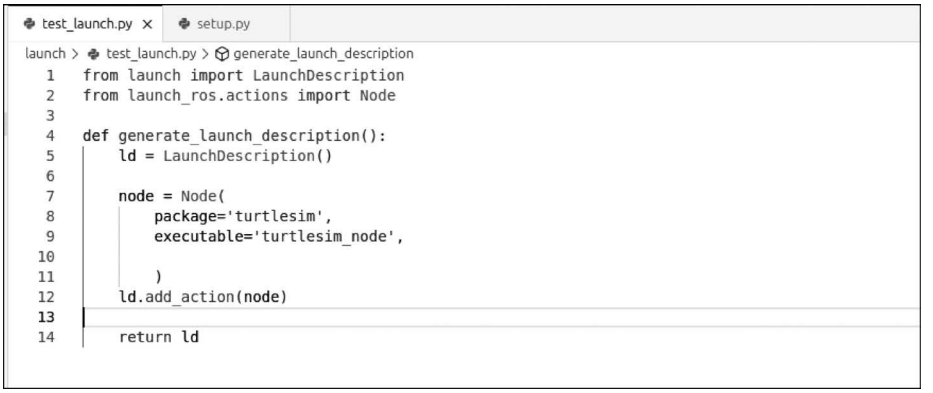

图 3-42　编写 Launch 文件

此外,也可以将创建 LaunchDescription 类的实例和添加节点两个步骤在 LaunchDescription 类初始化时一次性完成,代码如下:

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    node = Node(
        package='turtlesim',
        executable='turtlesim_node',
        )

    return LaunchDescription([node ])            #在列表中添加需要启动的节点
```

以上两种编写 Launch 文件的方法是等价的，按照习惯和场景选择任意方法即可。

(3) 配置 Launch 文件。修改功能包中的 setup.py 文件，在 setup.py 文件的头部添加导入 glob 库，代码如下：

```
from glob import glob
```

然后修改 setup()函数内的 data_files 变量，向 data_files 中添加一条语句，代码如下：

```
('share/'+package_name+'/launch', glob('launch/*launch.py')),
```

上述代码使用 glob()函数匹配 launch 文件夹中的所有以 launch.py 结尾的文件。使用通配符的好处是如果以后再编写新的 Launch 文件，则无须每次都修改 setup.py 文件中的 data_files 变量。setup.py 文件中的关键部分修改完成后的结果如图 3-43 所示。

src > test_package_python > setup.py > ...

```
1  from setuptools import find_packages, setup
2  from glob import glob
3  package_name = 'test_package_python'
4
5  setup(
6      name=package_name,
7      version='0.0.0',
8      packages=find_packages(exclude=['test']),
9      data_files=[
10         ('share/ament_index/resource_index/packages',
11             ['resource/' + package_name]),
12         ('share/' + package_name, ['package.xml']),
13         ('share/'+package_name+'/launch', glob('launch/*launch.py')),
14     ],
```

图 3-43　修改 setup.py 文件

(4) 运行 Launch 文件。打开终端，进入工作空间 ros2_ws，编译该功能包，使用 ros2launch 命令运行 Launch 文件，命令如下：

```
cd ~/ros2_ws
colcon build --packages-select test_package_python --symlink-install
ros2 launch test_package_python test_launch.py
```

运行后即可看到小海龟仿真环境的窗口，说明 Launch 文件编写成功，如图 3-44 所示。

3.4.4　案例：命名空间与节点名称设置

在 Launch 文件中，如果需要使一个节点程序运行多个副本，则只需重复创建多个节点程序的启动节点实例，例如启动两个小海龟仿真窗口，只需创建两个相同的节点。为了加以区分这些同名的节点，可以在创建节点添加一个 namespace 参数作为命令名空间，为节点加上一个前缀。修改 test_launch.py 文件中的代码，修改后的代码如下：

```
#ros2_ws3/src/test_package_python/launch/test1_launch.py
from launch import LaunchDescription
```

```
from launch_ros.actions import Node

def generate_launch_description():
    ld = LaunchDescription()

    node1 = Node(
        package='turtlesim',
        executable='turtlesim_node',
        namespace='turtle1',
        )
    ld.add_action(node1)

    node2 = Node(
        package='turtlesim',
        executable='turtlesim_node',
        namespace='turtle2',
        )
    ld.add_action(node2)

    return ld
```

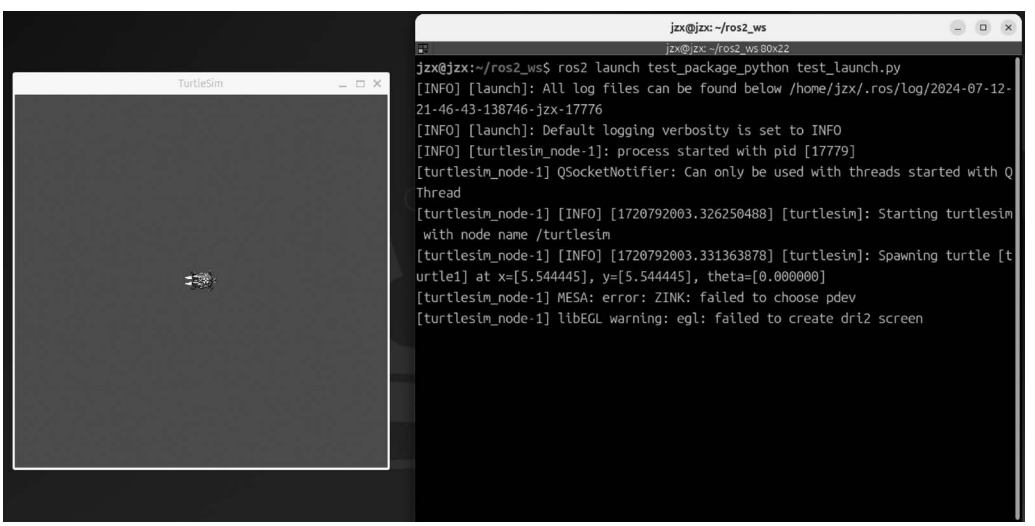

图 3-44　运行 Launch 文件

重新编译,运行 Launch 文件后就会打开两个小海龟窗口,使用 ros2 node list 命令查看节点信息会发现每个节点前面都会多一个前缀,也就是 Launch 文件中对节点设置的 namespace,如图 3-45 所示。

如果想要修改节点的启动名称,则可以在节点对象实始化时添加 name 参数,以 node1 为例,将其名称修改为 sim,代码如下:

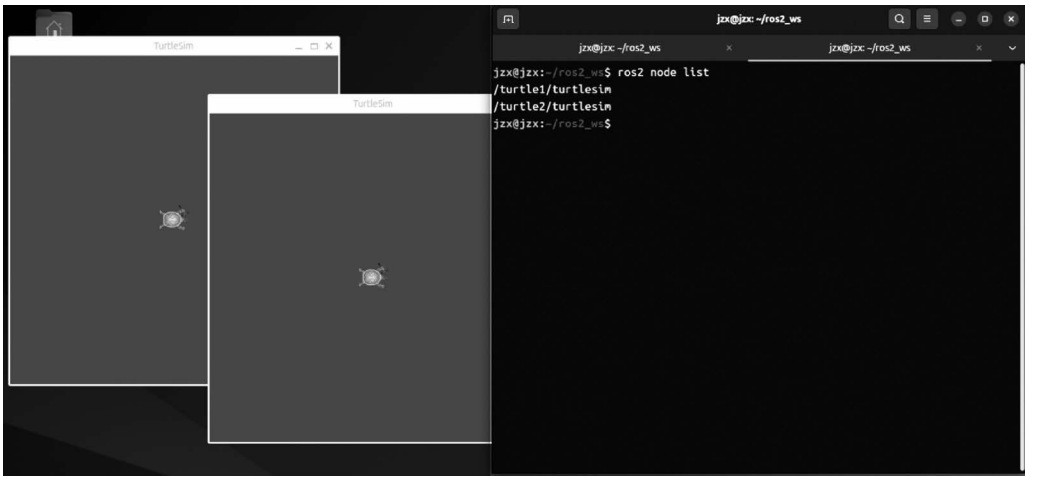

图 3-45　添加命名空间

```
node1 = Node( package='turtlesim', executable='turtlesim_node',
       namespace='turtle1', name='sim' )
ld.add_action(node1)
```

重新编译后运行，再次使用 ros2 node list 命令查询节点信息，便可发现节点名称发生了改变，如图 3-46 所示。

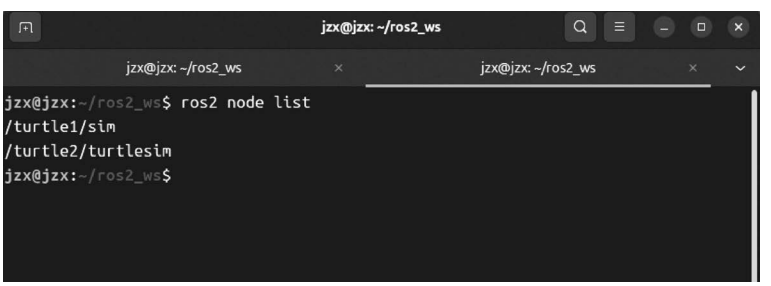

图 3-46　修改节点名称

3.4.5　案例：参数设置

使用 parameters 参数可以为节点配置参数。配置参数又分为不同的方式，以下介绍两种，一种是在 Launch 文件内设置参数，另一种是通过 YAML 文件导入参数进行配置。下面以小海龟仿真环境为例说明 Launch 文件中设置参数的方法。

(1) 在 Launch 文件内设置参数。DeclareLaunchArgument 类用于在 Launch 文件中创建参数。创建一个名为 test3_launch.py 的 Launch 文件，内容如下：

```
#ros2_ws3/src/test_package_python/launch/test3_launch.py
from launch import LaunchDescription
```

```
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration, TextSubstitution

def generate_launch_description():
    ld = LaunchDescription()
    r = DeclareLaunchArgument(
      'background_r', default_value=TextSubstitution(text='0')
   )
    g = DeclareLaunchArgument(
      'background_g', default_value=TextSubstitution(text='15')
   )
    b = DeclareLaunchArgument(
      'background_b', default_value=TextSubstitution(text='125')
   )
    ld.add_action(r)
    ld.add_action(g)
    ld.add_action(b)

    node1 = Node(
        package='turtlesim',
        executable='turtlesim_node',
        parameters=[{
            'background_r': LaunchConfiguration('background_r'),
            'background_g': LaunchConfiguration('background_g'),
            'background_b': LaunchConfiguration('background_b'),
        }]
        )
    ld.add_action(node1)

return ld
```

在上述代码中，通过 DeclareLaunchArgument 分别声明 3 个参数：background_r、background_g 和 background_b。每个参数都有一个默认值，使用了 TextSubstitution 来指定默认值为固定的文本字符串。TextSubstitution 是用来提供参数默认值的一种方法，允许将一个静态的文本字符串作为参数的默认值。最后在启动节点时要读取所设置的 3 个参数，LaunchConfiguration 是一个用于从 Launch 文件中读取参数值的机制。它允许将参数的值作为配置传递给节点或其他操作。

编译运行上述 Launch 文件后，打开的小海龟仿真窗口的背景色发生了变化，背景色被设置为 Launch 文件中给定参数的值，如图 3-47 所示。

此外，对于使用 DeclareLaunchArgument 创建的参数可通过 ros2 launch 命令的命令行参数在 Launch 文件运行时动态地进行设置。例如将小海龟仿真窗口的背景色修改为白色，命令如下：

```
ros2 launch test_launch.py background_r:=255 background_g:=255 background_b:=255
```

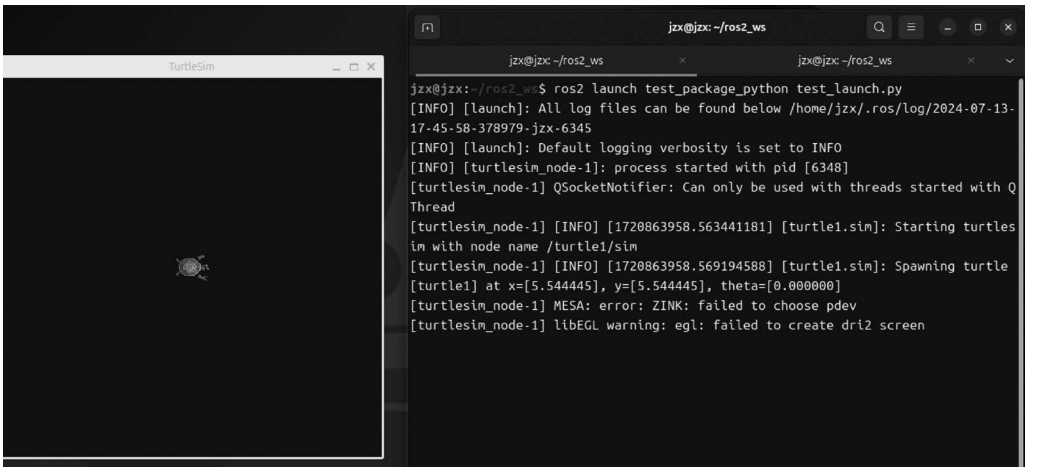

图 3-47　参数设置

(2) 使用 YAML 文件配置参数。可按照创建 launch 文件夹的方法,在功能包中创建一个 config 文件夹,在 config 文件夹内创建一个以 YAML 格式存放参数的 turtlesim.yaml 文件,命令如下:

```
cd ~/ros2_ws/src/test_package_python
mkdir config
cd config
touch turtlesim.yaml
```

其次,修改 turtlesim.yaml 文件,向 turtlesim.yaml 文件中添加参数信息,内容如下:

```
/**:
   ros__parameters:
      background_r: 0
      background_g: 255
      background_b: 255
```

以上用通配符的方式向所有节点定义了 background_r、background_g 和 background_B 这 3 个参数,并且将这些参数文件配置到多个节点。使用通配符避免了重复创建参数文件的复杂工作,简化了参数文件的使用。

注意:YAML 格式的参数配置文件可通过 ros2 param dump < node_name >命令生成一个参数模板配置文件后进行修改。

再次,修改功能包中的 setup.py 文件,向 setup.py 文件的变量 data_files 中添加以下元素,代码如下:

```
('share/' + package_name + '/config', glob('config/*')),
```

变量 data_files 被修改后的结果如图 3-48 所示。

```
data_files=[
    ('share/ament_index/resource_index/packages',
        ['resource/' + package_name]),
    ('share/' + package_name, ['package.xml']),
    ('share' + package_name + 'launch', glob('launch/*.py')),
    ('share/' + package_name + '/config', glob('config/*.yaml'))
],
```

图 3-48　setup.py 文件被修改后的结果

最后,修改 Launch 文件,代码如下:

```
#ros2_ws3/src/test_package_python/launch/test4_launch.py
import os
from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    ld = LaunchDescription()

    config = os.path.join(
    get_package_share_directory('test_package_python'),
    'config',
    'turtlesim.yaml'
    )

    node1 = Node(
        package='turtlesim',
        executable='turtlesim_node',
        parameters=[config]
        )
    ld.add_action(node1)

    return ld
```

在以上的 Launch 文件代码中,首先使用 get_package_share_directory()函数获取了功能包的安装路径,根据功能包的安装路径生成了配置文件的路径 config,然后将配置文件的路径设置为节点的参数。

对功能包进行编译和安装后,运行 Launch 文件,如图 3-49 所示,小海龟窗口的背景颜色发生了改变。

3.4.6　案例:话题重映射

重映射允许在启动节点时动态地更改其订阅和发布的话题名称,使节点之间即使最初设计时使用了不同的话题名称也能够相互匹配,从而协同工作。

下面以重映射小海龟的位置话题为例,介绍话题重映射的实现方法,代码如下:

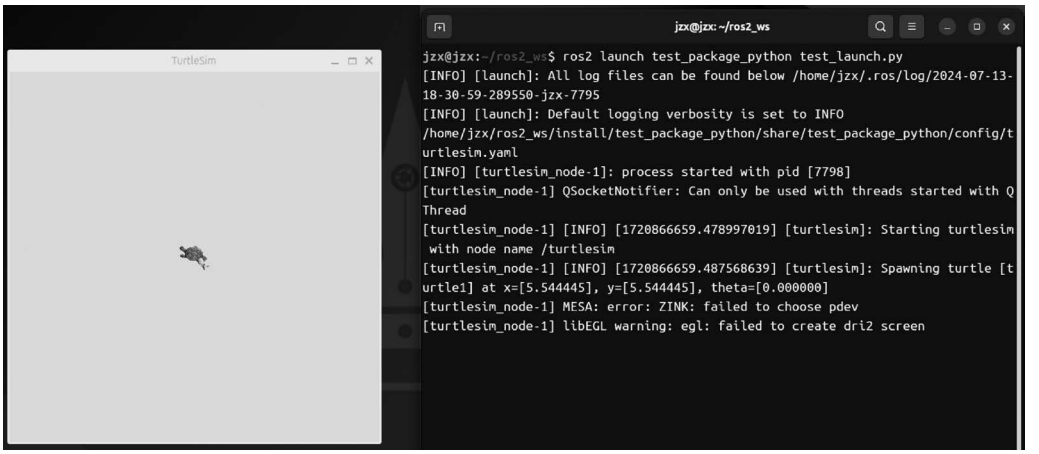

图 3-49　加载参数配置文件

```
#ros2_ws3/src/test_package_python/launch/test5_launch.py
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    ld = LaunchDescription()

    node1 = Node(
        package='turtlesim',
        executable='turtlesim_node',
        remappings=[('/turtle1/pose','/sim1/pose')]
        )
    ld.add_action(node1)

    return ld
```

在上述代码中通过设置节点的 remapping 参数来将小海龟在仿真环境中原来的/turtle1/pose 话题重映射为/sim1/pose 话题。

对功能包进行编译和安装后，首先运行 Launch 文件，然后打开一个新的终端，使用命令 ros2 topic list 查询话题列表，可以看到原来的/turtle1/pose 话题已经被重映射为/sim1/pose，如图 3-50 所示。

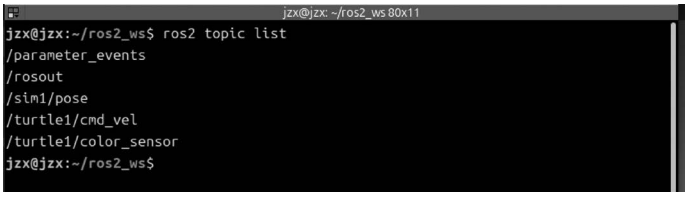

图 3-50　话题重映射

以上几个案例介绍了编写 Launch 文件的常用功能，此外，也可以在一个 Launch 文件中包含另一个 Launch 文件，组合多个 Launch 文件，在后续的机器人仿真中另行介绍。

3.5 URDF 简介

URDF(Unified Robot Description Format)是一种在 ROS 2 中描述机器人模型的 XML 文件格式。URDF 可描述机器人的连杆、关节、传感器、碰撞、转动惯量等信息。机器人的 URDF 文件能够被 RViz 可视化，直观地显示机器人的几何形态。机器人模型的 URDF 文件以. urdf 结尾。

XACRO(XML Macros)是一种带有宏定义的 URDF 文件格式，能够简化 URDF 文件。具体来讲，XACRO 格式允许用户在 URDF 中通过定义宏和参数来创建更简洁、可重用和易于管理的机器人模型描述。机器人模型 XACRO 文件以. xacro 结尾。ROS 2 中的 xacro 功能包中的 xacro 程序能够将 XACRO 格式的机器人模型文件转换为 URDF 格式。

3.5.1 机器人状态发布器

在 ROS 2 中使用 URDF 文件主要包括加载机器人模型和发布机器人关节状态两个过程。机器人模型的发布需要使用机器人状态发布器(Robot State Publisher)。机器人状态发布器的作用是将机器人的状态发布到 tf 树，使 ROS 2 能够获取机器人内部各连杆和关节的坐标系，从而确定机器人的状态。

机器人状态发布器是一个 ROS 2 的功能包，安装命令如下：

```
sudo apt install ros-jazzy-robot-state-publisher
```

机器人状态发布器提供了一个与功能包同名的节点程序 robot_state_pulisher。在启动该节点时，需要提供一个名为 robot_description 的字符串参数，参数的值为机器人 URDF 文件内的字符串。robot_state_publisher 节点启动后会订阅名为 joint_states 的话题(类型为 sensor_msgs/msg/JointState)，以获取机器人各关节的状态。首先根据机器人的关节状态更新机器人模型的姿态，然后将生成的机器人 3D 姿态发布到 tf 树。机器人状态发布器节点的完整接口如表 3-1 所示。

表 3-1 机器人状态发布器节点的完整接口

接口类型	名称	说明
参数	robot_description	字符串类型，这是 URDF 文件的内容，必须在节点启动时设置该参数。在节点运行时改变该参数会使节点发布的 robot_description 同步改变
	publish_frequency	浮点数类型，/tf 话题发布非静态坐标系的最大发布频率。默认值为 20.0Hz

续表

接口类型	名　　称	说　　明
参数	ignore_timestamp	布尔类型，当为 true 时会忽略话题 joint_states 中的时间戳，当为 false 时会检查话题 joint_states 中的时间戳只发布时间戳较新的关节状态
	frame_prefix	字符串类型，向发布的坐标系添加前缀，默认为空，表示不添加
订阅的话题	joint_states	消息类型为 sensor_msgs/msg/JointState，这是机器人的关节状态（例如关节的角度或关节的位移等），机器人状态发布器根据该信息计算机器人各关节的坐标系变换
发布的话题	robot_description	消息类型为 std_msgs/msg/String，对机器人状态发布器节点中的 robot_description 参数进行发布，方便 ROS 2 中的其他节点使用机器人模型
	tf	发布经计算得到的机器人的动态坐标系变换
	tf_static	发布经计算得到的机器人中静态坐标系变换

3.5.2　案例：URDF 可视化

URDF 可视化的主要流程如图 3-51 所示，先将机器人关节状态发布到/joint_states 话题，然后机器人状态发布器在接收到上述话题后利用机器人模型 URDF 计算机器人的坐标系，并将机器人的坐标系发布到/tf 和/tf_static 话题，以及将机器人的 URDF 发布到/robot_description 话题，最后 RViz 接收上述话题对机器人进行可视化。

图 3-51　URDF 可视化的主要流程

以下通过一个 URDF 在 RViz 中的可视化的案例介绍机器人模型在 ROS 2 中的使用方法。

（1）在工作空间中创建一个功能包 urdf_test，命令如下：

```
ros2 pkg create --build-type ament_python --license MIT urdf_test
```

（2）在功能包 urdf_test 中创建一个名为 urdf 的文件夹，并在该文件夹内创建一个名为 robot.urdf 的文件，内容如下：

```
#ros2_ws3/src/urdf_test/urdf/robot.urdf
<?xml version="1.0"?>
<robot name="simple_robot">

  <!-- 机械部分 -->
```

```
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.1 0.1 0.1"/>
      </geometry>
      <origin xyz="0 0 0.05" rpy="0 0 0"/>
      <material name="blue">
        <color rgba="0.0 0.0 1.0 1.0"/> <!-- 蓝色 -->
      </material>
    </visual>
  </link>
  <link name="link1">
    <visual>
      <geometry>
        <box size="0.1 0.1 0.1"/> <!-- 上面连杆 -->
      </geometry>
      <origin xyz="0 0 0.15" rpy="0 0 0"/>
      <material name="yellow">
        <color rgba="1.0 1.0 0.0 1.0"/> <!-- 黄色 -->
      </material>
    </visual>
  </link>

  <!-- 关节部分 -->
  <joint name="joint1" type="revolute">
    <parent link="base_link"/>
    <child link="link1"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
    <limit lower="-1.57" upper="1.57" effort="10" velocity="1.0"/> <!-- 限制关节
的旋转角度 -->
  </joint>
</robot>
```

以上内容定义了一个由两个连杆和一个关节构成的简单机器人模型的 URDF,其中,< robot >是描述机器人的根元素,将名称设置为 simple_robot;< link >用于定义机器人的连杆元素,总共有两个,名称分别为 base_link 和 link1;< visual >定义了连杆的视觉表示,主要包括< geometry >表示几何状态为边长为 0.1m 的立方体,< origin >用于设置连杆的位置,< material >用于设置连杆的颜色;< joint >元素中的 type 属性将关节类型设置为 revolute(转动关节)类型,并通过< parent >和< child >子元素设置了连杆间的关系,< axis >子元素设置了关节的旋转轴,< limit >设置了关节的转动范围,以及关节上的力和转动速度的限制。

(3) 在功能包 urdf_test 中创建一个 launch 文件夹,并在该文件夹内创建了一个名为 show_urdf_launch.py 的文件,代码如下:

```
#ros2_ws3/src/urdf_test/launch/show_urdf_launch.py
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

#获取当前包的安装地址
packagepath = get_package_share_directory('urdf_test')
print(packagepath)

#读取 urdf 文件
urdfpath=packagepath+'/urdf/robot.urdf'
robot_desc=open(urdfpath).read()

def generate_launch_description():
    robot_desc_node=Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='both',
        parameters=[
            {'use_sim_time': True},
            {'robot_description': robot_desc},
        ])

    joint_state_pub_node=Node(
        package='joint_state_publisher_gui',
        executable='joint_state_publisher_gui',
    )

    rviz_node=Node(
        package='rviz2',
        executable='rviz2',
        name='rviz',
        arguments=[ '-d', packagepath+'/urdf/rviz.rviz', ]
    )

    return LaunchDescription([
        robot_desc_node,
        joint_state_pub_node,
        rviz_node,
    ])
```

在上述 Launch 文件中，首先将 Python 读取的机器人 URDF 文件内容保存到变量 robot_desc 中，随后添加了一个机器人状态发布器节点 robot_state_publisher，并将参数 robot_descripttion 设置为读取的 URDF 字符串，再次添加了一个关节状态发布的 GUI 工具 joint_state_publisher_gui，用于发布人为设置伪造的 joint_states 话题，最后添加了一个可视化机器人模型的 RViz 节点。

注意：joint_state_publisher_gui 是一个 ROS 2 的功能包，安装命令是 sudo apt install ros-jazzy-joint-state-publisher-gui。

(4) 配置功能包 urdf_test 的 setup.py 文件，修改变量 data_files 的值，代码如下：

```
data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        ('share/'+package_name+'/urdf',glob('urdf/*.urdf')),
        ('share/'+package_name+'/urdf',glob('urdf/*.rviz')),
        ('share/'+package_name+'/launch',glob('launch/*launch.py')),
    ],
```

(5) 经过编译和安装后，使用 Launch 文件进行启动，命令如下：

```
colcon build --symlink-install --packages-select urdf_test
source install/setup.bash
ros2 launch urdf_test show_urdf_launch.py
```

启动后的效果如图 3-52 所示，在 RViz 左侧列表中将 Global Options 中的 Fixed Frame 选项设置为 base_link，添加和设置 RobotModel 数据，使其订阅/robot_description 话题，在 RViz 显示区域显示机器人模型，此外，在打开的 Joint State Pulisher 窗口内手动设置关节的角度时，RViz 中的机器人关节会发生相应的转动。

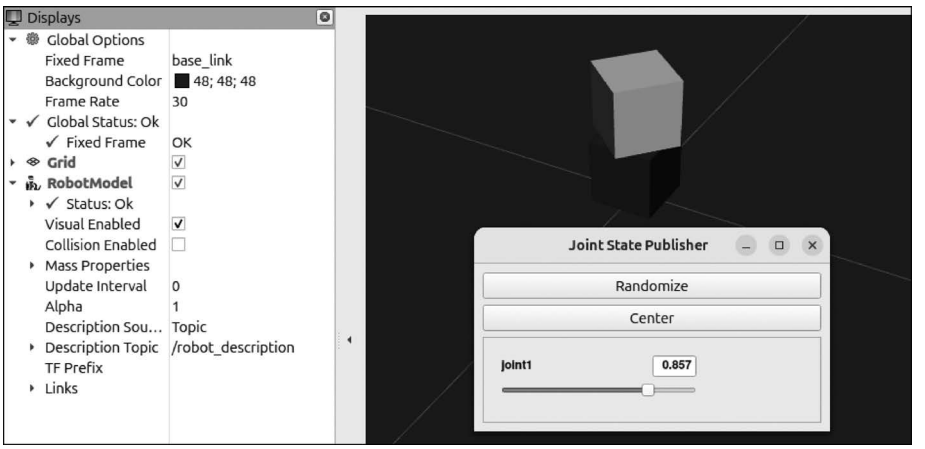

图 3-52　URDF 的加载与显示

上述案例介绍了在 ROS 2 中进行机器人可视化的方法。需要说明的是机器人的可视化与物理仿真有较大的差异。ROS 2 不具备对机器人进行物理仿真的能力，物理仿真需要借助物理仿真工具，例如 Gazebo。

在机器人仿真中 Gazebo 能够加载 URDF 文件，并通过在 URDF 文件中使用<gazebo>

元素对 URDF 文件进行了扩展，使用户能够为其机器人模型提供额外的仿真细节，例如物理特性、传感器模型和环境交互等，从而在 Gazebo 中对机器人进行仿真。

相较于 URDF，Gazebo 提供的 SDF 格式具备更强大的功能，能够方便地进行环境和机器人仿真，越来越成为机器人仿真领域的首选格式，有取代 URDF 的趋势。目前 ROS 2 也以插件的形式支持加载和使用 SDF 格式的机器人模型。在后续章节会详细介绍使用 SDF 格式编写仿真环境与机器人模型的方法。

3.6　本章小结

本章系统全面地介绍了 ROS 2 的编程方法。colcon 是 ROS 2 自定义功能包默认的构建工具，支持构建 C++和 Python 功能包，能够处理包的依赖关系，提高开发效率。rclpy 库是 Python 语言的 ROS 2 客户端库，提供了 ROS 2 中话题、服务、动作等通信机制，此外还支持管理节点参数、定义消息类型等功能。TF2 为 ROS 2 提供了坐标系管理功能，支持发布静态坐标系和动态坐标系，以及支持查询坐标系间的变换关系。Launch 文件提供了 ROS 2 系统中的多个节点的启动方法，方便了 ROS 2 程序的启动。URDF 是 ROS 2 默认的机器人模型格式，使用 RViz 可对 URDF 进行可视化。本章是 ROS 2 的核心内容，熟练掌握和灵活应用本章内容是后续开发 ROS 2 应用的基础。