ROS2 学习笔记

Turtlesim及rqt

turtlesim是学习ros2的轻量级模拟器。它说明了ros2在最基本的层面上做了什么，让你知道你以后将如何处理一个真正的机器人或机器人仿真。
rqt是ros2的GUI工具。在ros2中完成的所有操作都可以在命令行上完成，但是它提供了一种更简单、更用户友好的方式来操作ros2基本模块。
使用以下命令可以启动一个turtlesim控制台

```
ros2 run turtlesim turtlesim_node
```

此时会弹出一个窗口，出现一只随机乌龟在窗口中心，并且能从控制台知道部分信息例：

```
[INFO] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000
```

现在打开一个新终端，输入以下命令可以开始通过箭头键控制这只乌龟

```
ros2 run turtlesim turtle_teleop_key
```

输入`q`退出窗口
如果有安装rqt，可以通过输入以下命令运行

```
rqt
```

然后按照如下步骤选择：Plugins > Services > ServiceCaller
就可以调到turtlesim对应窗口了，选择Service下拉栏选择/spawn，设置好下方信息后（注意name参数不能重名，否则报错）选择右上角的Call生成一只名为name参数栏填的数值的新乌龟
你还可以选择/name/set_pen来更改某只乌龟的画笔参数
打开新终端，输入以下命令可以开始控制第二只名为`turtle2`的乌龟：

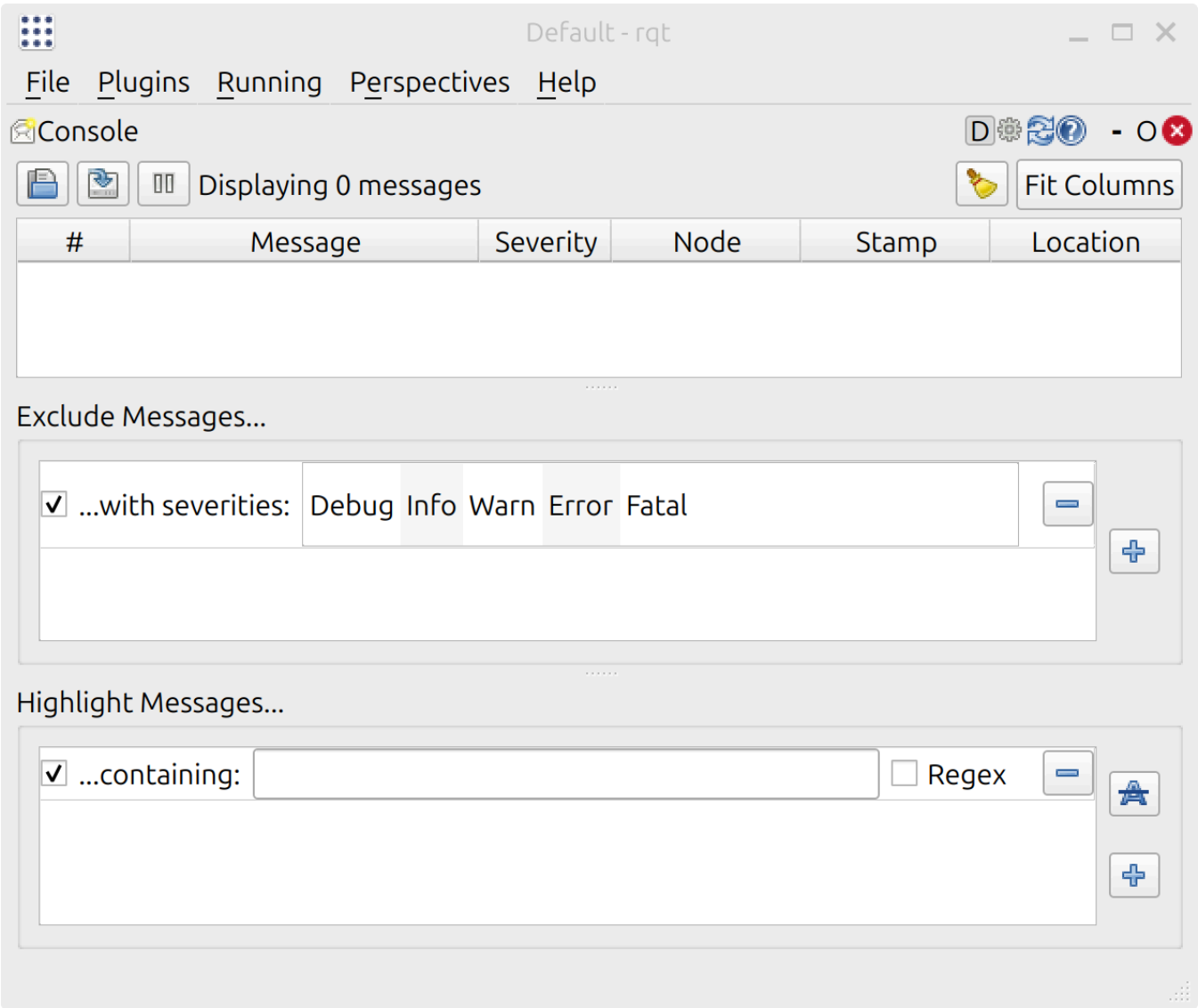
```
ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=turtle2/cmd_vel
```

rqt_console

在控制台输入

```
ros2 run rqt_console rqt_console
```

或者在rqt上选择Plugins > Logging > Console进入，界面如图所示



第一部分显示来自系统的日志消息，第二部分是一个过滤器，可以选择严重性级别来过滤信息，也可以用右侧的加号来添加过滤条件，第三部分突出显示输入的`string`的信息，也可以添加过滤条件

当你运行

```
ros2 run turtlesim turtlesim_node
```

第一部分会显示两个信息，接下来输入

```
ros2 topic pub -r 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z
```

这个命令会一直发送topic，让乌龟不断向一个地方前进，然后`rqt_console`上第一部分会不断显示`Warn`级别的信息表示乌龟碰到`Wall`即边界，以下是严重性级别：

```
Fatal
Error
Warn
Info
Debug
```

分别表示：

级别	意义
Fatal	系统即将终止，以免受到损害
Error	重大问题不一定损害系统，但会阻碍正常系统的运行
Warn	表示意想不到的结果，一般不会直接损害功能
Info	表示事件的状态更新
Debug	详细说明了系统执行的整个步骤

一般情况下，严重性级别为`Info`，也就是只能看到其及其往上级别的信息，如果设置

```
ros2 run turtlesim turtlesim_node --ros-args --log-level WARN
```

则将级别设置为`Warn`，在节点创建时无法看到两个新的`Info`级别的信息

ROS 2图

Node

事实上ros2的运行指令

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

意义为在包名`<package_name>`中寻找可执行文件`<executable_name>`
我们在运行一个turtlesim控制台后，可以在新控制台中通过以下命令获取目前的所有节点

```
ros2 node list
```

然后会返回节点的名称，我们使用

```
ros2 run turtlesim turtle_teleop_key
```

实际上就是启动一个teleop节点
此时再次运行节点查看命令就会返回2个节点
我们通过

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

重新分配节点的名称（即**重新映射**），此时查看节点发现有3个节点
于是我们可以通过

```
ros2 node info /my_turtle
```

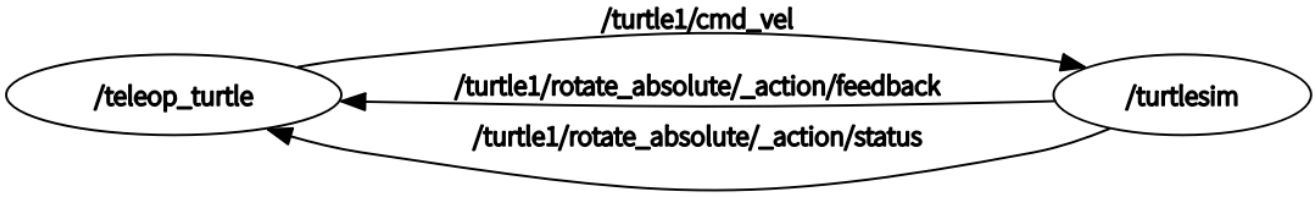
访问节点`my_turtle`的信息（这条命令原型为`ros2 node info <node_name>`）

rqt_graph

我们可以通过

```
rqt_graph
```

来可视化节点和变化的话题(Topic)
也可以通过rqt下的Plugins > Introspection > Node Graph来打开节点图，大概如图所示：



这表明`/turtlesim`节点和`/teleop_turtle`节点通过一个话题相互通信,`/teleop_turtle`节点出版（出版者Publisher）数据 (按键输入移动乌龟) 到`/turtle1/cmd_vel`话题，`/turtlesim`节点订阅（订阅者Subscriber） 话题接收数据。
运行

```
ros2 topic list
```

可以得知当前活动下所有话题的列表，使用

```
ros2 topic list -t
```

返回相同的话题列表，不过这次在`[]`括号中附加了话题类型，在Node Graph界面可以将Hide栏的所有选项钩掉，会显示话题所有信息

回声(Echo)

可以使用

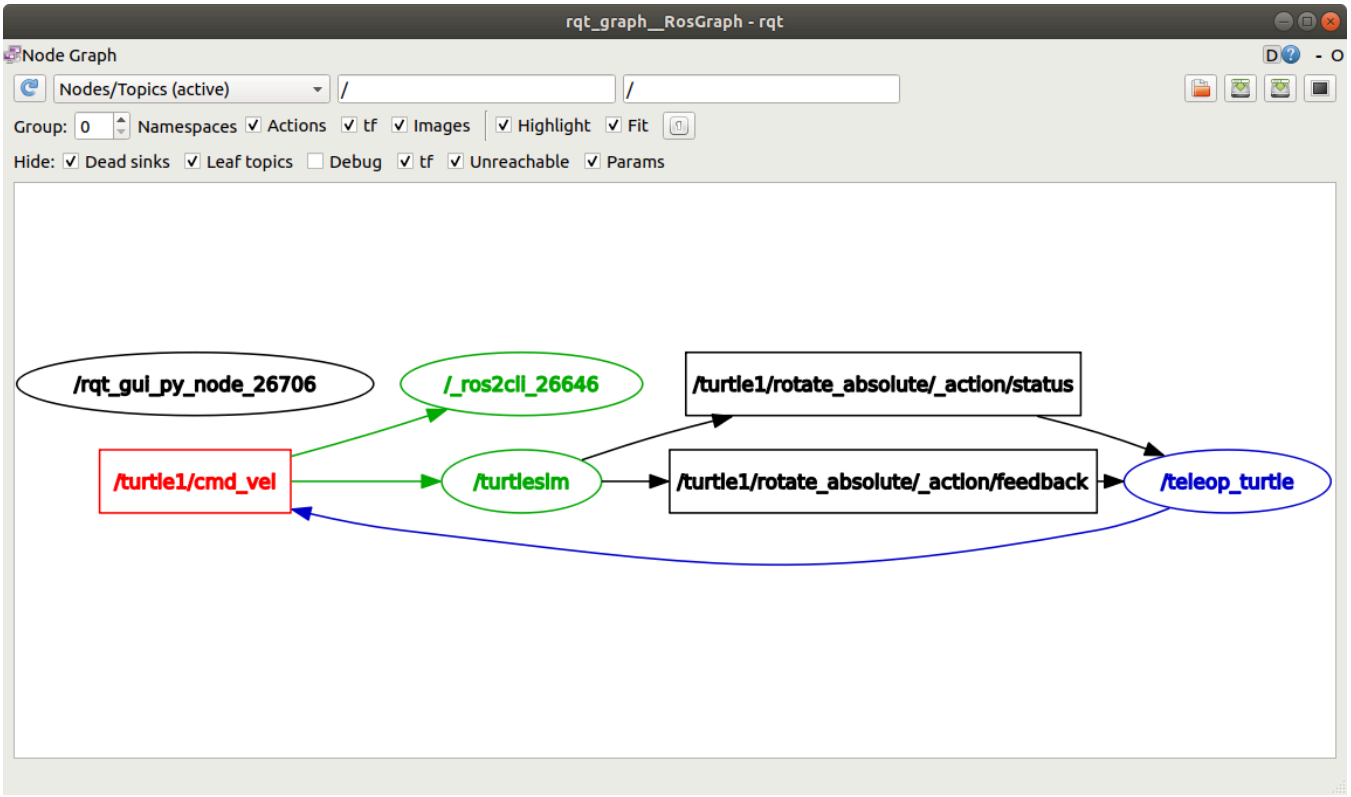
```
ros2 topic echo <topic_name>
```

来查看正在发布的关于某个话题的数据例：

```
ros2 topic echo /turtle1/cmd_vel
```

这个命令一开始并不会返回任何数据，因为它输出的数据都是通过`/teleop_turtle`发布的，即

当你在`turtle_teleop_key`终端上按键移动乌龟时，你的`echo`终端才会返回实时信息
现在回到Node Graph中关掉调试的hide，会发现`/_ros2cli_26646`是由`echo`终端创建的（数字可能有所不同），所以你可以发现`teleop_turtle`节点正在作为发布者不断发布`cmd_vel`的相关数据，同时`/_ros2cli_26646`作为`/turtle1/cmd_vel`的订阅者订阅这个数据，所以这个节点拥有两个订阅者



我们可以通过

```
ros2 topic info /turtle1/cmd_vel
```

来看到topic相关的信息：

```
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
```

发布者1个，订阅者2个

ROS 2界面展示

有关上一节讲到的`geometry_msgs/msg/Twist`意味着`geometry_msgs`包装中有一个`msg`调

用`Twist`，我们可以通过

```
ros2 interface show geometry_msgs/msg/Twist
```

查看相关信息，我们在`ros2 topic list -t`后得到的列表也表明了每个topic使用了什么信息类型，可以知道`/turtle1/cmd_vel`使用的就是`geometry_msgs/msg/Twist`类型，上文命令执行后会出现

```
# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
        float64 x
        float64 y
        float64 z
Vector3  angular
        float64 x
        float64 y
        float64 z
```

这样的信息，这告诉我们`/turtle1`期望有两个向量的信息`linear`和`angular`，每个向量有三个元素，如果重新调用数据，我们可以看到`/teleop_turtle`通过`echo`命令传给`/turtlesim`，结构是一样的：

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
  ---
```

Pub

Pub就是Publisher，它允许我们将数据直接发送到topic上，使用如下命令：

```
ros2 topic pub <topic_name> <msg_type> '<args>'
```

其中`<topic_name>`指topic名称，可为`/turtle1/cmd_vel`，`<msg_type>`指消息类型，也就是上文提到的`geometry_msgs/msg/Twist`，最后`'<args>'`是将传递给topic的实际数据，位于我们在上一节中刚刚发现的结构中，例如我们可以：

```
ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0,
```

来发送一个数据，然后会返回

```
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=2.0, y=0.0, z=0
```

此时你会看到你的乌龟画了一段圆弧，我们采取

```
ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0
```

可以让乌龟一直画圆弧，与上述命令不同在于将`--once`改成了`--rate 1`，意味着将从只发送一次命令改为了按1Hz的稳定流发送命令
此时在Node Graph中能看到pub的节点`/_ros2cli_8382`正在发布关于`/turtle1/cmd_vel`这个topic的内容，并且被`echo`的节点`/_ros2cli_7466`与`/turtlesim`接收，最后我们在`pose`这个topic上运行`echo`

```
ros2 topic echo /turtle1/pose
```

此时Node Graph里显示`/turtlesim`节点也在发布`pose`话题，新的`echo`节点subscribe了这个话题
我们可以通过

```
ros2 topic hz /turtle1/pose
```

来查看数据发布的频率，可能会返回

```
average rate: 59.354
  min: 0.005s max: 0.027s std dev: 0.00284s window: 58
```

这样类似的结果，现在将上述命令中的`/turtle1/pose`换成`/turtle1/cmd_vel`，会发现返回变成了稳定的平均值：

```
..
average rate: 1.000
        min: 1.000s max: 1.000s std dev: 0.00022s window: 18
average rate: 1.000
        min: 1.000s max: 1.000s std dev: 0.00021s window: 19
..
```

Service

Service是不同于Publisher – Subscriber模型的另一种通讯方式，通过

```
ros2 service list
```

可以返回当前系统中活动的所有service列表，可以发现两个节点都有相同的六个服务，名字均含有`parameters`，我们可以通过

```
ros2 service type <service_name>
```

来得到某个服务的类型，例如`/clear`服务，它告诉我们它是`std_srvs/srv/Empty`类型的，`Empty`类型表示调用时不发送任何数据，收到响应时也不接收任何数据。也可以通过

```
ros2 service list -t
```

来获取所有服务的类型，如果我们想知道特定类型的服务有哪些，可以用

```
ros2 service find <type_name>
```

这个指令会返回所有某个类型的服务，例如查找`std_srvs/srv/Empty`即`Empty`类型的服务会返回

```
/clear
/reset
```

通过

```
ros2 interface show <type_name>.srv
```

我们可以获取某个类型的输入参数结构，例如`std_srvs/srv/Empty.srv`即`Empty`类型，则会返回

```
---
```

其中`---`分割上下两个部分，上面部分是**请求结构**，下面部分是**响应结构**，由于`Empty`类型不会发送和接收任何数据，所以上下均为空
我们能知道`/spawn`的类型是`Spawn`，使用

```
ros2 interface show turtlesim/srv/Spawn
```

返回

```
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be created and returned if this is empty
---
string name
```

这个`# Optional`表明`name`这个参数是可选的
我们可以使用

```
ros2 service call <service_name> <service_type> <arguments>
```

来调用服务，其中`<arguments>`是可选的，例如：

```
ros2 service call /clear std_srvs/srv/Empty
```

它的效果是清除海龟画出来的轨迹，用

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"
```

可以生成一只新海龟，但是没有给出名字，此时默认赋予`turtle2`为名，接着使用

```
ros2 service call /kill turtlesim/srv/Kill "{name: 'turtle2'}"
```

就可以杀死这个新海龟

Parameter

使用

```
ros2 param list
```

可以获取参数列表，可以看到命名空间、`/teleop_turtle`和`/turtlesim`，后面是每个节点的参数：

```
/teleop_turtle:
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  use_sim_time
```

用

```
ros2 param get <node_name> <parameter_name>
```

可以获取某个节点的某个参数的类型，例如`ros2 param get /turtlesim background_g`可以获取`/turtlesim`节点下`background_g`参数的类型，默认情况下返回参数值：

```
Integer value is: 86
```

也就是说它是一个整数值，同理`background_r`和`background_b`
我们可以通过

```
ros2 param set <node_name> <parameter_name> <value>
```

来更改参数的值，例如`ros2 param set /turtlesim background_r 150`可以改变背景的红色色值为150，终端会返回`Set parameter successful`如果成功了的话
为了能不只在这个会话内生效这些配置，可以保存设置，利用转储到`.yaml`文件上可以保存配置，使用如下命令可以将当前节点的所有参数值转储到文件中

```
ros2 param dump <node_name>
```

如果保存的是`/turtlesim`，终端也许会返回`Saving to:  ./turtlesim.yaml`（如果出现问题，即没有找到保存的文件，可以该为输
入`ros2 param dump /turtlesim >>turtlesim.yaml`来明确指定也许就有了，文件一般保存在`/home/user`），于是你能在工作区中找到这样一个文件，里面的内容是这样的：

```
turtlesim:
  ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
    use_sim_time: false
```

而使用

```
ros2 param load <node_name> <parameter_file>
```

就可以加载文件中的配置了，例如加载刚刚生成的`./turtlesim.yaml`，会返回

```
Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
Set parameter use_sim_time successful
```

如果想在创建节点的同时加载配置，可以使用

```
ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
```

例如：

```
ros2 run turtlesim turtlesim_node --ros-args --params-file ./turtlesim.yaml
```

将在`turtlesim`节点启动的时候加载这个节点的所有配置

Action

Action也是一种**通讯类型**，用于长期运行的任务，它由三部分组成：目标、反馈、结果，Action使用Client – Sever模型，类似于Publisher – Subscriber模型，Action Client节点向Action Sever节点发送**目标**(Goal)，该节点确认目标并返回**反馈流**(Feedback)和**结果**(Result)
当你运行`/teleop_turtle`节点时，控制台会返回

```
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
Use g|b|v|c|d|e|r|t keys to rotate to absolute orientations. 'f' to cancel a rotation.
'q' to quit.
```

其中第四行的所有键都围绕在`F`键周围，表示海龟转向的方向，每按下一个键，如果成功响应则会返回

```
[INFO] [turtlesim]: Rotation goal completed successfully
```

如果企图在一个操作执行完前立即执行另一个操作例如按下`D`后乌龟还没转向完毕就按下`G`控

制台就会返回

```
[WARN] [turtlesim]: Rotation goal received before a previous goal finished. Aborting previ
```

我们使用

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

来返回节点的action，例如`/turtlesim`，则会返回

```
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/get_type_description: type_description_interfaces/srv/GetTypeDescription
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
```

是一个包含Subscriber、Publisher、Service、Action Service、Action Client的列表，其中`/turtlesim`的`/turtle1/rotate_absolute`这个action在Action Servers之下。这意味着`/turtlesim`对`/turtle1/rotate_absolute`做出反应并提供反馈。
同样我们可以用`ros2 node info /teleop_turtle`来获取`/teleop_turtle`的相关节点信息
如果想知道一个节点的所有action，可以用

```
ros2 action list
```

获取action列表，目前应该会返回

```
/turtle1/rotate_absolute
```

这是目前唯一的action
同样如果要获取这个action的类型，我们可以

```
ros2 action list -t
```

这样就会返回

```
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

我们可以通过

```
ros2 action info /turtle1/rotate_absolute
```

进一步获取这个action的相关信息，可以得到

```
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

如果我们使用

```
ros2 interface show turtlesim/action/RotateAbsolute
```

会得到有关这个action接口的相关信息

```
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

从上到下分别是**目标请求结构、结果结构、反馈结构**
我们可以用

```
ros2 action send_goal <action_name> <action_type> <values>
```

来向命令行发送一个action目标，其中`<values>`需用`.yaml`格式传递，例如

```
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1
```

我们向`/turtlesim`节点的action`/turtle1/rotate_absolute`发送，终端应该会输出

```
Waiting for an action server to become available...
Sending goal:
   theta: 1.57

Goal accepted with ID: f8db8f44410849eaa93d3feb747dd444

Result:
  delta: -1.568000316619873

Goal finished with status: SUCCEEDED
```

此时乌龟在原地旋转直到朝向弧度为1.57的方向，`delta`那一栏表示的就是转向的差值，每个目标都有一个唯一的账号即`ID`，这里的就是`f8db8f44410849eaa93d3feb747dd444`，这个ID并非不会变，每次调用都会发生变化
我们可以使用

```
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: -1
```

来实时查看命令执行过程中的feedback即实时状态（仅仅加了个`--feedback`），也许会返回

```
Sending goal:
   theta: -1.57

Goal accepted with ID: e6092c831f994afda92f0086f220da27

Feedback:
  remaining: -3.1268222332000732

Feedback:
  remaining: -3.1108222007751465

…

Result:
  delta: 3.1200008392333984

Goal finished with status: SUCCEEDED
```

这样类似的东西，直到action结束才输出`Result:`往下的那几行东西

Launch

使用`ros2 launch`这个命令可以同时在一个系统中启动多个配置好的节点，避免打开过多的终端
例如运行

```
ros2 launch turtlesim multisim.launch.py
```

这个命令运行了一个`launch.py`的`Python`文件，当然也可以用`.xml`和`.yaml`来创建`launch`文件。
这条命令等价于运行以下launch文件：

```
# turtlesim/launch/multisim.launch.py

from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            namespace= "turtlesim1", package='turtlesim', executable='turtlesim_node', ou
        launch_ros.actions.Node(
            namespace= "turtlesim2", package='turtlesim', executable='turtlesim_node', ou
    ])
```

运行了两个`turtlesim`节点
我们可以在其他终端使用pub来将数据发送到topic上，例如

```
ros2 topic pub  /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y:
```

将数据发送给话题`/turtlesim1/turtle1/cmd_vel`，此时会发现`turtle1`正在画圈

Bag

Bag用于记录和回放发布在某个topic上的数据，首先通过

```
mkdir bag_files
cd bag_files
```

创建一个存放回放的文件夹并进入
接下来我们选择一个话题，我们知道`/turtle_teleop`在`/turtle1/cmd_vel`这个topic上发布命令，让海龟移动，它发布的数据可用`echo`获取
在进入回放目录的情况下，使用

```
ros2 bag record <topic_name>
```

可以记录发布到topic的数据，以`/turlte1/cmd_vel`为例
如果使用的是Jazzy版本的ros2，运行后会出现

```
[WARN] [ros2bag]: Positional "topics" argument deprecated. Please use optional "--topics"
[INFO] [1764768960.806863941] [rosbag2_recorder]: Press SPACE for pausing/resuming
[INFO] [1764768960.818911904] [rosbag2_recorder]: Listening for topics...
[INFO] [1764768960.818971456] [rosbag2_recorder]: Recording...
[INFO] [1764768960.818989699] [rosbag2_recorder]: Event publisher thread: Starting
[INFO] [1764768961.328299202] [rosbag2_recorder]: Subscribed to topic '/turtle1/cmd_vel'
[INFO] [1764768961.428535211] [rosbag2_recorder]: All requested topics are subscribed. St
```

反之如果是使用Foxy版本的，则会返回：

```
[INFO] [rosbag2_storage]: Opened database 'rosbag2_2019_10_11-05_18_45'.
[INFO] [rosbag2_transport]: Listening for topics...
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/cmd_vel'
[INFO] [rosbag2_transport]: All requested topics are subscribed. Stopping discovery...
```

这主要是因为版本的不同，要取出jazzy版本下的第一行弃用警告，只需要将命令改为

```
ros2 bag record --topics <topic_name>
```

并且jazzy版本下还支持通过空格来**暂停、继续**记录，更加灵活
按下`ctrl+c`来停止记录，此时会留下一个文件记录所有数据，一般名为`rosbag2_year_month_day-hour_minute_second`
这个文件所在的文件夹一般也是通过这个规律命名的
可以使用

```
ros2 bag record -o subset <topic_name1> <topic_name2> ..
```

来同时记录多个话题，其中`-o`允许我们为存放文件的文件夹选择唯一的一个名字，这里是`subset`，以及如果加上`-a`可以记录我们这个系统下的所有topic，同理也是通过`ctrl+c`结束记录
我们使用

```
ros2 bag info <bag_file_name>
```

其中`<bag_file_name>`是包文件，即存文件的文件夹，以`subset`为例，它会返回

```
Files:             subset_0.mcap
Bag size:          104.0 KiB
Storage id:        mcap
ROS Distro:        jazzy
Duration:          22.031464610s
Start:             Dec  3 2025 21:45:20.641051266 (1764769520.641051266)
End:               Dec  3 2025 21:45:42.672515876 (1764769542.672515876)
Messages:          1383
Topic information: Topic: /turtle1/cmd_vel | Type: geometry_msgs/msg/Twist | Count: 5 | Se
                   Topic: /turtle1/pose | Type: turtlesim/msg/Pose | Count: 1378 | Serial
Service:           0
Service information:
```

之后在存放bag文件的文件夹中输入

```
ros2 bag play subset
```

可以开始回放`subset`中记录的数据，jazzy版本下会看到以下信息返回

```
[INFO] [1764769769.989943352] [rosbag2_player]: Set rate to 1
[INFO] [1764769769.993538001] [rosbag2_player]: Adding keyboard callbacks.
[INFO] [1764769769.993556009] [rosbag2_player]: Press SPACE for Pause/Resume
[INFO] [1764769769.993559955] [rosbag2_player]: Press CURSOR_RIGHT for Play Next Message
[INFO] [1764769769.993563834] [rosbag2_player]: Press CURSOR_UP for Increase Rate 10%
[INFO] [1764769769.993567162] [rosbag2_player]: Press CURSOR_DOWN for Decrease Rate 10%
[INFO] [1764769769.994738952] [rosbag2_player]: Playback until timestamp: -1
```

然后海龟会完全（不一定，海龟对系统时间的微小变化极为敏感，但大差不差）按照记录的数据运行，包括等待的时间，途中可以通过空格暂停和继续，完成后会自动结束命令的运行，我们可以通过某个话题的`Count`值查看这个topic的数据被发表了多少次，如果要了解发布位置数据的频率，可以用`ros2 topic hz /turtle1/pose`

TF2

tf2用于维护ros2中所有节点所在的不同坐标系之间的变换关系，形成一棵“坐标树”
先通过

```
sudo apt-get install ros-jazzy-turtle-tf2-py ros-jazzy-tf2-tools ros-jazzy-tf-transformat
```

安装tf2的例程包，然后安装一下py的`transforms3d`包

```
sudo apt update
sudo apt install python3-transforms3d
```

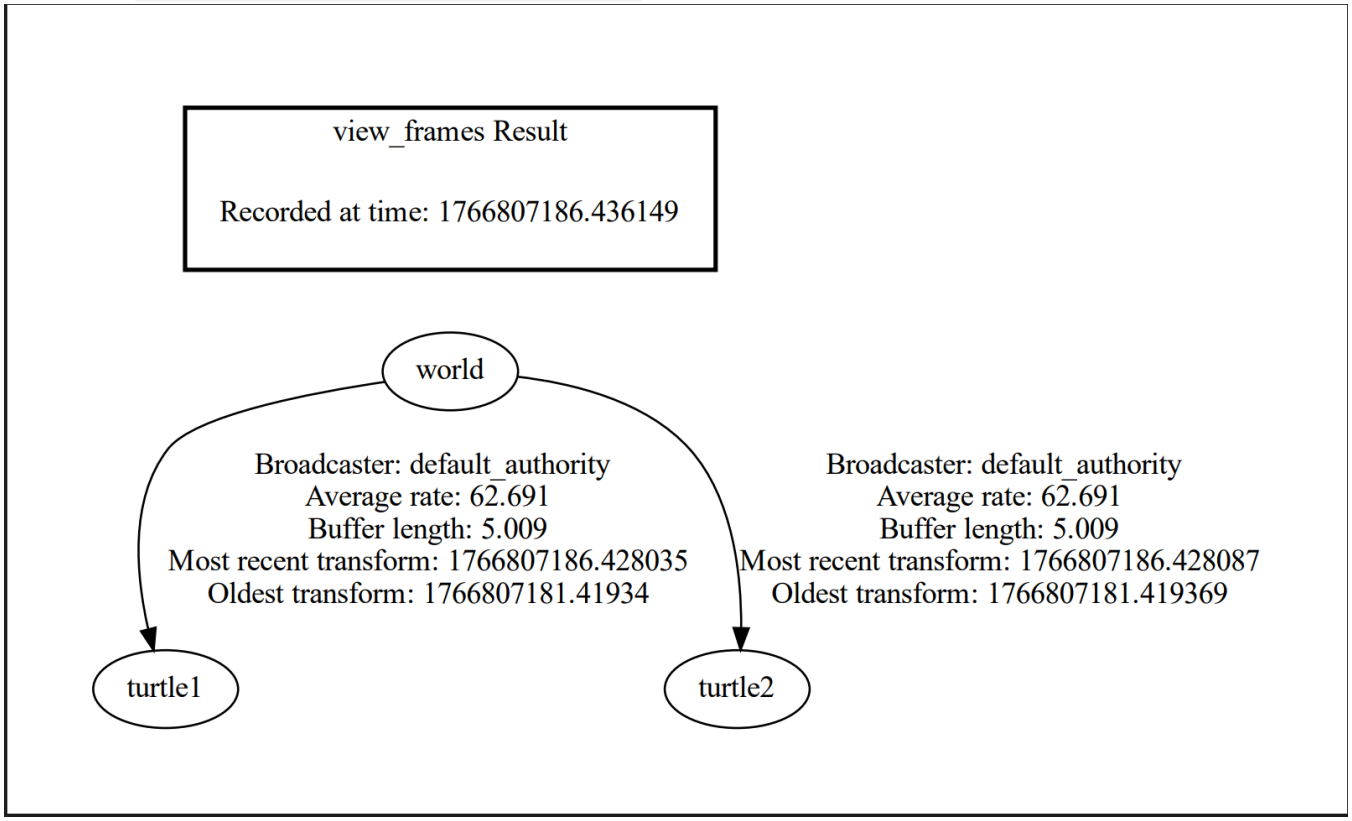
上述安装命令均基于Ubuntu24.04的Jazzy版本，其他版本命令有所不同
通过

```
ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
```

运行demo文件，如果再运行一个teleop节点，可以看到一只乌龟永远追随着另一只被控制的乌龟，我们可以用

```
ros2 run tf2_tools view_frames
```

生成一个`frames_2025-12-27_11.46.26.pdf`内有一张图



在这里，tf2持续发布两只乌龟相对于世界坐标系的位置和朝向，从而使turtle2持续跟随turtle1成为可能

我们使用`ros2 run tf2_ros tf2_echo [reference_frame] [target_frame]`可以持续返回两个帧的相对变换，如果写

```
ros2 run tf2_ros tf2_echo turtle2 turtle1
```

可以得到turtle2帧相对于turtle1帧的变换，持续返回

```
At time 1766810488.299839012
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion (xyzw) [0.000, 0.000, 0.235, 0.972]
- Rotation: in RPY (radian) [0.000, -0.000, 0.474]
- Rotation: in RPY (degree) [0.000, -0.000, 27.181]
- Matrix:
  0.890 -0.457  0.000  0.000
  0.457  0.890  0.000  0.000
  0.000  0.000  1.000  0.000
  0.000  0.000  0.000  1.000
```

我们一行一行来看，`- Translation: [0.000, 0.000, 0.000]`表示从 turtle1 坐标系原点到 turtle2 坐标系原点的平移向量

`- Rotation: in Quaternion (xyzw) [0.000, 0.000, 0.235, 0.972]`表示从 turtle1 坐标系到 turtle2 坐标系的旋转，使用**四元数**表示

`- Rotation: in RPY (radian) [0.000, -0.000, 0.474]`则表示四元数在欧拉角下的弧度表示，下面一行是角度表示，参数分别对应$\mathrm{Roll}, \mathrm{Pitch}, \mathrm{Yaw}$即**滚转（自身）、俯仰（上下）、偏航（左右）**

接下来的矩阵是**齐次变换矩阵**：

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 旋转部分（左上 3×3 纯旋转矩阵）：
  表示坐标系旋转
  目前纯旋转矩阵显示绕 z 轴旋转约 27°，纯旋转矩阵为：

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- 平移部分（右列前三行）：
  $\begin{bmatrix} 0.000 & 0.000 & 0.000 \end{bmatrix}^T$ - 位置重合

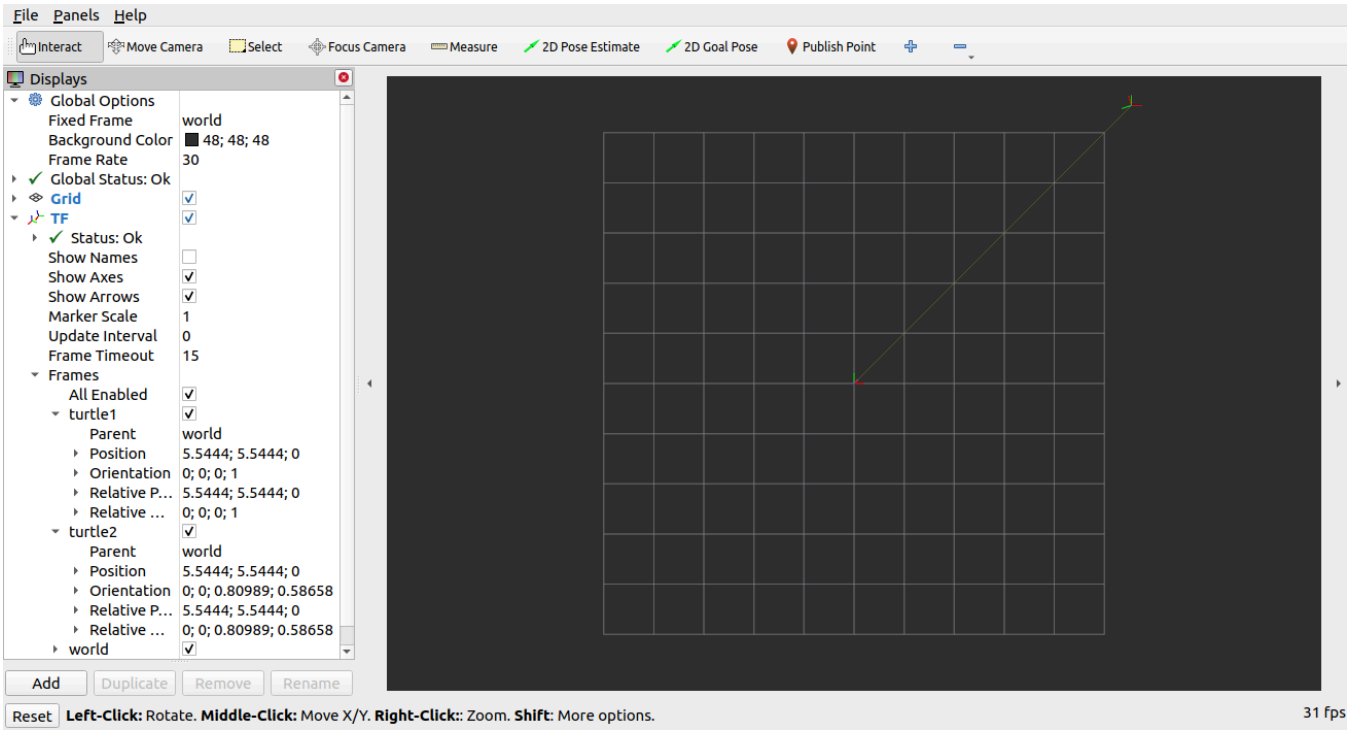
在这里对四元数

$$q = \begin{bmatrix} x\sin\frac{\theta}{2} \\ y\sin\frac{\theta}{2} \\ z\sin\frac{\theta}{2} \\ \cos\frac{\theta}{2} \end{bmatrix}$$

表示绕旋转向量轴$\begin{bmatrix} x & y & z \end{bmatrix}^T$旋转$\theta$，在这里旋转轴只能为$\vec{z} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$
我们可以使用rviz来查看有关tf2的信息，输入

```
ros2 run rviz2 rviz2 -d $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz
```

可以启动目前 turtle2 和 turtle1 相关的一个rviz工具：



右上角两个小箭头分别表示两只海龟，它们现在是重合的，中心原点的右上半正方形部分表示默认 turtlesim 窗口内的样子

Fast DDS Discovery Server

C++环境

创建与运行功能包

先输入

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

每次打开终端要运行`.cpp`的时候都需要的配置环境变量，如果不想每次进入都输入这个，可以输入

```
echo "source /home/goose/dev_ws/install/setup.bash" >> ~/.bashrc
```

这样每次打开终端默认配置好环境变量了，然后输入这个

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

就在主目录下创建了一个`dev_ws`文件夹及其子文件夹`src`，先进行一次编译

```
colcon build
```

会生成几个子文件夹，然后我们输入

```
ros2 pkg create pkg01_test_cpp --build-type ament_cmake --dependencies rclcpp --node-name
```

会在目前文件夹下新建一个`pkg01_test_cpp`文件夹，这就是我们自己创建的一个功能包（而`rclcpp`是依赖的另一个功能包），内有`include`和`src`子文件夹以及`CmakeLists.txt`和`package.xml`文件，进入`src`文件夹会发现自带一个`helloworld.cpp`的程序（基础命令是`ros2 pkg create <name>`）
我们`cd ..`回到上一级目录（因为要在工作空间进行，也就是`~/dev_ws`），输入`colcon build`再进行一次编译，然后

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

来刷新环境变量，然后我们就可以使用

```
ros2 run pkg01_helloworld__cpp helloworld
```

来运行这个文件了，输出

```
hello world pkg01_helloworld__cpp package
```

然后我们修改这个`.cpp`文件为

```
/*
            需求: 终端输出 Hello World
            流程:
                    1.包含头文件
                    2.初始化Ros2客户端
                    3.创建节点指针
                    4.输出日志
                    5.释放资源
*/

#include "rclcpp/rclcpp.hpp"

int main(int argc, char **argv)
{
        // 初始化Ros2客户端
        rclcpp::init(argc, argv);
        // 创建节点指针
        auto node = rclcpp::Node::make_shared("helloworld_node_cpp");
        // 输出日志
        RCLCPP_INFO(node->get_logger(), "hello world!");
        // 释放资源
        rclcpp::shutdown();
        return 0;
}
```

编辑器会提示有错误但是不用管它，因为它是识别不到`rclcpp`的，这个存在于其他文件中，我们也没有设置编译路径包含这个，到时候由cmake自助链接
然后步骤同上，先`colcon build`编译，
再`ros2 run pkg01_helloworld__cpp helloworld`（如果该文件夹下有多个`.cpp`文件，需要先到cmake配置好后才能通过这个命令运行）就能看到终端输出了

```
[INFO] [1764835077.406100667] [helloworld_node_cpp]: hello world!
```

当你的工作空间的`src`文件夹下有多个项目（即多个子文件夹），在编译时可以用

```
colcon build --packages-select <file_name1> <file_name2> ..
```

来指定编译若干个文件
接下来我们可以通过`git clone`命令将github上的一些ros2相关的资源克隆到本机上

```
git clone https://github.com/ros/ros_tutorials.git -b jazzy
```

可将`ros_tutorials`文件夹拷贝下来，内有很多官方的ros2相关教程
接下来使用（在工作空间的根目录下）

```
rosdep install -i --from-path src --rosdistro jazzy -y
```

来解决好依赖环境，如果没有安装rosdep需提前安装，控制台返回

```
#All required rosdeps installed successfully
```

表明已拥有所有依赖项
我们可以在`turtlesim`中的`src`中的`turtlesim`中的`turtle_frame.cpp`修改一些东西，然后编译的时候使用

```
colcon build --packages-select turtlesim --allow-overriding turtlesim
```

刷新运行环境后直接运行`ros2 run turtlesim turtlesim_node`你会发现你的改动作用于这个命令创建出的窗口，并且如果你新打开窗口，继续运
行`ros2 run turtlesim turtlesim_node`发现同样也被改变了，这是因为加
了`--allow-overriding turtlesim`导致编译的时候叠加层叠加到了底层上，如果取消了的话那么改动只会作用于你现在的运行环境下，新窗口不受影响
这里要注意的是：
底层指的是系统ROS2安装路径（例如 /opt/ros/jazzy），这里存放着像原始 turtlesim这样的功能包
叠加层指的是工作空间（比如 ~/ros2_ws），当使用`. install/setup.bash`（也就是`source install/setup.bash`）后，将当前终端的环境叠加到了工作空间之上
如果选择关闭现有的终端，打开新终端，同样可以恢复原样，但是注意不要source到工作空间的`setup.bash`（如果有在`~/.bashrc`自动source到工作空间，可以先临时注释掉，然后执行`source ~/.bashrc`或重启终端），这样就会用原始的`turtlesim`
有时候仍会出现无论怎么source改动依旧存在的情况，这很可能是`~/.bashrc`写了自动source到工作空间的代码，最好直接去掉，反正每次编译完成都需要刷新运行环境，所以其实用处不

大，修改后重新开启终端再运行就会发现是原始的了
如果想一劳永逸，最好的办法就是重命名工作空间下的`turtlesim`包文件名

编写简单的项目

Publisher和Subscriber

先新建一个功能包`cpp_pubsub`，然后使用

```
wget -O publisher_member_function.cpp https://raw.githubusercontent.com/ros2/examples/jaz
```

拷贝一个`publisher_member_function.cpp`，这个文件在标准库的导入后还导入了`rclcpp/rclcpp.hpp`和`std_msgs/msg/string.hpp`这两个头文件，前者允许使用ros2中最基本的函数，后者包含发布数据的内置消息类型，里面的代码大概是这样的：

```cpp
// Copyright 2016 Open Source Robotics Foundation, Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include <chrono>
#include <functional>
#include <memory>
#include <string>

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

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to register a
 * member function as a callback from the timer. */

class MinimalPublisher : public rclcpp::Node
{
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
    timer_ = this->create_wall_timer(
      500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }

private:
```

```
  void timer_callback()
  {
    auto message = std_msgs::msg::String();
    message.data = "Hello, world! " + std::to_string(count_++);
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

这段代码写了一个简单的Publisher节点，每$500ms$发布一条`Hello World`消息到topic，并在控制台打印发送的信息

开头的`class MinimalPubisher`继承自所有ros2节点的基类`rclcpp::Node`，并在构造函数中使用了两个基类的函数`create_publisher<T>()`和`create_wall_timer()`

分别表示创建一个Publisher发布到名为topic的topic，队列大小为10

创建一个$500ms$的定时器，绑定（新名字并自动填写参数）到`timer_callback`函数

而后定义的`timer_callback()`主要是

用`std_msgs::msg::String()`创建新的消息对象，通过其`data`成员设置消息内容

利用带参宏`RCLCPP_INFO`在控制台打印日志，最后调用publisher的publish函数发布消息

而在主函数中，`rclcpp::init(argc, argv);`表示初始化ros2系

统，`rclcpp::spin(std::make_shared<MinimalPublisher>());`传入一个刚创建

的`MinimalPublisher`类型的匿名共享指针来启动**节点循环**，最后的`rclcpp::shutdown();`关闭ros2

现在去到这个功能包下的`packages.xml`文件处，将

```
<description>TODO: Package description</description>
<maintainer email="goose@todo.todo">goose</maintainer>
<license>TODO: License declaration</license>
```

改为

```
<description>Examples of minimal publisher/subscriber using rclcpp</description>
<maintainer email="goose@todo.todo">goose</maintainer>
<license>Apache License 2.0</license>
```

补全包的信息，然后在`<buildtool_depend>`和`<test_depend>`之间添加

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

来添加依赖项，同理我们去到`CmakeLists.txt`下，
将`find_package(ament_cmake REQUIRED)`下方添加

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

来让编译器找到对应的库，然后在其下方加上

```
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

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

表示将可执行程序变为`talker`并添加Publisher节点的配置
现在我们来添加Subscriber节点的配置，依旧在`./src`下在终端输入

```
wget -O subscriber_member_function.cpp https://raw.githubusercontent.com/ros2/examples/fo
```

就能将一个`subscriber_member_function.cpp`的ros2官方Subscriber教程拷贝下来了，代码是这样的：

```
// Copyright 2016 Open Source Robotics Foundation, Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include <functional>
#include <memory>

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

using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
public:
  MinimalSubscriber()
  : Node("minimal_subscriber")
  {
    subscription_ = this->create_subscription<std_msgs::msg::String>(
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
  }

private:
  void topic_callback(const std_msgs::msg::String & msg) const
  {
    RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg.data.c_str());
  }
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};
```

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

和Publisher很相似，区别在于没有创建定时器，因为收到信息即响应，不过声明了一个`topic_callback()`的函数，用于接收发布的消息，在主函数中两者是完全一样的
这个节点与发布节点有相同的依赖关系所以不需要改`package.xml`，但是需要更改`CmakeLists.txt`
我们将原本的`Install`栏目的代码替换为

```
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)

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

现在用`colcon build --packages-select cpp_pubsub`编译，刷新环境变量后我们打开另一个终端，`source`到需求的工作空间
目前有2个终端，一个用于发布一个用于接收，分别运行`ros2 run cpp_pubsub talker`和`ros2 run cpp_pubsub listener`
会发现发布窗口每$0.5s$发布一个消息，而订阅窗口对应的会收到消息：

```
Publisher:
[INFO] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [minimal_publisher]: Publishing: "Hello World: 2"
[INFO] [minimal_publisher]: Publishing: "Hello World: 3"
[INFO] [minimal_publisher]: Publishing: "Hello World: 4"
..
---
Subscriber:
[INFO] [minimal_subscriber]: I heard: "Hello World: 10"
[INFO] [minimal_subscriber]: I heard: "Hello World: 11"
[INFO] [minimal_subscriber]: I heard: "Hello World: 12"
[INFO] [minimal_subscriber]: I heard: "Hello World: 13"
[INFO] [minimal_subscriber]: I heard: "Hello World: 14"
..
```

Server和Client

通过`ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp example_interfa`新建一个`cpp_srvcli`功能包，在`src`里创建`add_two_ints_server.cpp`文件，键入

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

#include <memory>

void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
          std::shared_ptr<example_interfaces::srv::AddTwoInts::Response>      response)
{
  response->sum = request->a + request->b;
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
                request->a, request->b);
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)resp
}

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");

  rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
    node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);

  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");

  rclcpp::spin(node);
  rclcpp::shutdown();
}
```

在`main`函数中依次下来，分别是初始化client客户端库、创建节点、为其创建名为`add_two_ints`的服务并自动调用`add`方法在网络上发布服务、打印日志消息、启动服务、结束服务

接着在cmake文件里配置好可执行文件

```
add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
rclcpp example_interfaces)

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

这样完成了一个服务节点，接下来创建一个客户端节点，同样在`src`里创建`add_two_ints_client.cpp`，键入

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

#include <chrono>
#include <cstdlib>
#include <memory>

using namespace std::chrono_literals;

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  if (argc != 3) {
      RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client X Y");
      return 1;
  }

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
  rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
    node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");

  auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
  request->a = atoll(argv[1]);
  request->b = atoll(argv[2]);

  while (!client->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the servic
      return 0;
    }
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
  }

  auto result = client->async_send_request(request);
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
    rclcpp::FutureReturnCode::SUCCESS)
  {
```

```
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
  } else {
    RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_two_ints");
  }

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

`main`中前半部分和server节点一致，从

```
auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
request->a = atoll(argv[1]);
request->b = atoll(argv[2]);
```

开始，创建了一个请求，它的结构由`.srv`文件定义：

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

接下来的`while`结构代表每秒搜索网络中的服务节点，直到找到为止，如果客户端中途被取消（例如通过`ctrl+c`结束运行）会返回结束消息，如果接收到了就返回对应消息
同样修改好cmake文件中的添加可执行程序方面和`Install`方面即可
在更新运行环境后在两个终端分别启动client节点和server节点，在启动client节点时使用

```
ros2 run cpp_srvcli client 2 3
```

在启动同时发表$2+3$的值，此时会发现服务节点终端返回

```
[INFO] [rclcpp]: Incoming request
a: 2 b: 3
[INFO] [rclcpp]: sending back response: [5]
```

`.srv`和`.msg`文件

接下来我们定义并使用这两个定义接口的文件，先介绍一下这两个文件

`.msg`

- 定义话题中传输的数据结构
- 用于Publisher-Subscriber模式的**单向通信**

例如这样定义一个`.msg`文件：

```
string name
int32 age
float64 height
bool is_student
```

`.srv`

- 定义服务的请求和响应结构
- 用于Client-Server模式的**请求-响应同步通信**

例如，这样定义一个`.srv`文件：

```
# 请求部分（客户端发送）
string request_data
int32 parameter
---
# 响应部分（服务器返回）
bool success
string response_message
float64 result
```

`.srv`和`.msg`的应用

使用`ros2 pkg create --build-type ament_cmake tutorial_interfaces`创建一个功能包，在其目录下使用

```
mkdir msg

mkdir srv
```

来创建两个容纳这两个文件的文件夹，我们在`/msg`文件夹内新建一个`Num.msg`文件，里面写上

```
int64 num
```

表示声明数据结构，即自定义消息传输num的单个64位整数
同理我们在`/srv`文件夹新建`AddThreeInts.srv`，里面写上

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

表示请求三个整数a,b,c并用num作为响应，然后我们在目录对应的cmake文档下添加

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "srv/AddThreeInts.srv"
)
```

可以将这两个接口转换为特定C++的代码，同样的，我们需要在`package.xml`下声明依赖（添加到`<export>`之前）（为了能在其他功能包调用这个包时检测到`.msg`和`.srv`文件能自动生成对应接口名的`.hpp`文件）

```
<build_depend>rosidl_default_generators</build_depend>

<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

编译后刷新环境并运行

```
ros2 interface show tutorial_interfaces/msg/Num
```

应该返回`int64 num`，运行

```
ros2 interface show tutorial_interfaces/srv/AddThreeInts
```

应该返回

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

现在我们将之前做的发布者和订阅者项目中的代码改为

```cpp
#include <chrono>
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/msg/num.hpp"     // CHANGE

using namespace std::chrono_literals;

class MinimalPublisher : public rclcpp::Node
{
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<tutorial_interfaces::msg::Num>("topic", 10);    //
    timer_ = this->create_wall_timer(
      500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }

private:
  void timer_callback()
  {
    auto message = tutorial_interfaces::msg::Num();                                  // CHANG
    message.num = this->count_++;                                                    // CHANGE
    RCLCPP_INFO(this->get_logger(), "Publishing: '%d'", message.num);    // CHANGE
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<tutorial_interfaces::msg::Num>::SharedPtr publisher_;         // CHANG
  size_t count_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

几个`CHANGE`的地方表示将原本使用官方提供的`String.hpp`下创建自定义消息的格式改为了我们自己的`num.hpp`下的格式，同理我们也需要更改订阅者：

```
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/msg/num.hpp"     // CHANGE
using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
public:
  MinimalSubscriber()
  : Node("minimal_subscriber")
  {
    subscription_ = this->create_subscription<tutorial_interfaces::msg::Num>(          //
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
  }

private:
  void topic_callback(const tutorial_interfaces::msg::Num::SharedPtr msg) const       // C
  {
    RCLCPP_INFO(this->get_logger(), "I heard: '%d'", msg->num);              // CHANGE
  }
  rclcpp::Subscription<tutorial_interfaces::msg::Num>::SharedPtr subscription_;      // C
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

cmake方面也需要更改：

```
cmake_minimum_required(VERSION 3.8)
project(cpp_pubsub)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tutorial_interfaces REQUIRED)                          # CHANGE

add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp tutorial_interfaces)          # CHANGE

add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp tutorial_interfaces)      # CHANGE

install(TARGETS
  talker
  listener
  DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

同样在`package.xml`中添加`<depend>tutorial_interfaces</depend>`
随后先编译`tutorial_interface`功能包再编译`cpp_pubsub`，随后新建两个终端，刷新到运行

环境后分别启动talker和listener节点，会正常返回发送和接收成功的消息：
Publisher:

```
..
[INFO] [1765183692.356420008] [minimal_publisher]: Publishing: '15'
[INFO] [1765183692.856420842] [minimal_publisher]: Publishing: '16'
[INFO] [1765183693.356449354] [minimal_publisher]: Publishing: '17'
..
```

Subscriber:

```
..
[INFO] [1765183692.356761601] [minimal_subscriber]: I heard: '15'
[INFO] [1765183692.856737511] [minimal_subscriber]: I heard: '16'
[INFO] [1765183693.356911984] [minimal_subscriber]: I heard: '17'
..
```

同理我们也可以修改服务和客户即`cpp_srvcli`功能包方面，将`add_two_ints_server.cpp`内的改为

```
#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/srv/add_three_ints.hpp"     // CHANGE

#include <memory>

void add(const std::shared_ptr<tutorial_interfaces::srv::AddThreeInts::Request> request,
          std::shared_ptr<tutorial_interfaces::srv::AddThreeInts::Response>       response
{
  response->sum = request->a + request->b + request->c;
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld" " c: %ld"
                request->a, request->b, request->c);
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)resp
}

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_three_ints_server")

  rclcpp::Service<tutorial_interfaces::srv::AddThreeInts>::SharedPtr service =
    node->create_service<tutorial_interfaces::srv::AddThreeInts>("add_three_ints",  &add)

  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add three ints.");      // CHANGE

  rclcpp::spin(node);
  rclcpp::shutdown();
}
```

这里由于官方提供的接口是接收两个参数，这里我们是三个参数所以要把`two`改为`three`

`add_two_ints_client.cpp`

```cpp
#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/srv/add_three_ints.hpp"        // CHANGE

#include <chrono>
#include <cstdlib>
#include <memory>

using namespace std::chrono_literals;

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  if (argc != 4) { // CHANGE
      RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_three_ints_client X Y Z");
      return 1;
  }

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_three_ints_client")
  rclcpp::Client<tutorial_interfaces::srv::AddThreeInts>::SharedPtr client =
    node->create_client<tutorial_interfaces::srv::AddThreeInts>("add_three_ints");

  auto request = std::make_shared<tutorial_interfaces::srv::AddThreeInts::Request>();
  request->a = atoll(argv[1]);
  request->b = atoll(argv[2]);
  request->c = atoll(argv[3]);               // CHANGE

  while (!client->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the servic
      return 0;
    }
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
  }

  auto result = client->async_send_request(request);
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
    rclcpp::FutureReturnCode::SUCCESS)
```

```
  {
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
  } else {
    RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_three_ints");
  }

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

注意，这里如果按照Foxy版本的wiki上的对client文件的改法回报错，这是Ros2版本差异导致的API变化问题。 在Jazzy版本中，`rclcpp::executor`命名空间已经发生了变化，即涉及到：

- Foxy版本：使用`rclcpp::executor::FutureReturnCode`
- Jazzy版本：简化了命名空间，直接使用`rclcpp::FutureReturnCode`或推荐使用标准库的`std::future_status`

所以我们可以把`rclcpp::executor::FutureReturnCode::SUCCESS`改为`rclcpp::FutureReturnCode::SUCCESS`
同理将cmake文件改为

```
cmake_minimum_required(VERSION 3.8)
project(cpp_srvcli)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tutorial_interfaces REQUIRED)        # CHANGE

add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
  rclcpp tutorial_interfaces)                      #CHANGE

add_executable(client src/add_two_ints_client.cpp)
ament_target_dependencies(client
  rclcpp tutorial_interfaces)                      #CHANGE

install(TARGETS
  server
  client
  DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

在`package.xml`加上`<depend>tutorial_interfaces</depend>`
同理编译运行两个节点后会出现
Server:

```
[INFO] [1765184834.078728119] [rclcpp]: Ready to add three ints.
[INFO] [1765184840.032429165] [rclcpp]: Incoming request
a: 2 b: 3 c: 1
[INFO] [1765184840.032480902] [rclcpp]: sending back response: [6]
```

Client:

```
[INFO] [1765184840.033016433] [rclcpp]: Sum: 6
```

同一包上使用接口

虽然可以专门定义功能包来定义接口，并在其它功能包调用这个功能包的接口来使用，但是有时候在同一功能包中定义并使用接口更方便，接下来来展示相关做法
我们继续创建功能包`more_interfaces`，在其`/msg`下创建`AddressBook.msg`，并输入

```
bool FEMALE=true
bool MALE=false

string first_name
string last_name
bool gender
uint8 age
string address
```

随后在`package.xml`中添加接口包声明以及构建工具依赖

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http:/
<package format="3">
  <name>more_interfaces</name>
  <version>0.0.0</version>
  <description>Package containing custom interfaces</description>
  <maintainer email="goose@todo.todo">goose</maintainer>
  <license>Apache License 2.0</license>

  <!--构建工具依赖-->
  <buildtool_depend>ament_cmake</buildtool_depend>
  <buildtool_depend>rosidl_default_generators</buildtool_depend>

  <!--运行时依赖-->
  <exec_depend>rosidl_default_runtime</exec_depend>

  <!--测试依赖-->
  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <!--接口包声明-->
  <member_of_group>rosidl_interface_packages</member_of_group>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

正如注释所说：

标签	意味
`<buildtool_depend>`	表示构建工具依赖
`<exec_depend>`	表示运行时依赖
`<test_depend>`	表示测试依赖
`<member_of_group>`	表示接口包声明

标签	意味
`<export>`	表示导出配置
`<build_type>`	指定包的构建系统类型

（值得注意的是，`package.xml`中的`<name>`后跟的名字**必须**是小写字母，可以和下划线，数字组合，这个名字应与`CmakeLists.txt`的`project(name)`中的name相同）
同样在`CmakeLists.txt`中添加`find_package(rosidl_default_generators REQUIRED)`来查找msg/srv文件生成消息代码的包，然后声明要生成的消息列表：

```
set(msg_files
  "msg/AddressBook.msg"
)
```

生成消息：

```
rosidl_generate_interfaces(${PROJECT_NAME}
  ${msg_files}
)
```

并确保导出消息运行时的依赖项`ament_export_dependencies(rosidl_default_runtime)`
最后我们在`/src`内新建`publish_address_book.cpp`并写入

```cpp
#include <chrono>
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "more_interfaces/msg/address_book.hpp"

using namespace std::chrono_literals;

class AddressBookPublisher : public rclcpp::Node
{
public:
  AddressBookPublisher()
  : Node("address_book_publisher")
  {
    address_book_publisher_ =
      this->create_publisher<more_interfaces::msg::AddressBook>("address_book", 10);

    auto publish_msg = [this]() -> void {
        auto message = more_interfaces::msg::AddressBook();

        message.first_name = "John";
        message.last_name = "Doe";
        message.age = 30;
        message.gender = message.MALE;
        message.address = "unknown";

        std::cout << "Publishing Contact\nFirst:" << message.first_name <<
          "  Last:" << message.last_name << std::endl;

        this->address_book_publisher_->publish(message);
      };
    timer_ = this->create_wall_timer(1s, publish_msg);
  }

private:
  rclcpp::Publisher<more_interfaces::msg::AddressBook>::SharedPtr address_book_publisher_
  rclcpp::TimerBase::SharedPtr timer_;
};
```

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<AddressBookPublisher>());
  rclcpp::shutdown();

  return 0;
}
```

在类`AddressBookPublisher`中，到`auto publish_msg = [this]() -> void {`为止是在创建一个节点和一个`AddressBook`发布服务器，而这行是创建一个可调用对象作为回调函数（用于定时器），`[this]`表示捕获并使用当前类的this指针
然后`auto message = more_interfaces::msg::AddressBook();`创建一个`AddressBook`消息实例，稍后将发布该实例，之后填充`AddressBook`字段
最后

```
std::cout << "Publishing Contact\nFirst:" << message.first_name <<
  "  Last:" << message.last_name << std::endl;

this->address_book_publisher_->publish(message);
```

周期调用函数发送消息，最后一行
的`timer_ = this->create_wall_timer(1s, publish_msg);`表示创建一个1s计时器每秒调用`publish_msg`函数
然后，我们在`CmakeLists.txt`文件中继续完成没添加的内容

```
add_executable(publish_address_book
  src/publish_address_book.cpp
)

ament_target_dependencies(publish_address_book
  "rclcpp"
)

rosidl_target_interfaces(publish_address_book
  ${PROJECT_NAME} "rosidl_typesupport_cpp")

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

表示找到包、创建可执行文件、链接依赖、连接同一包接口、安装可执行文件，最后的样子应该是这样的：

```
cmake_minimum_required(VERSION 3.8)
project(more_interfaces)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

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

set(msg_files
  "msg/AddressBook.msg"
)
rosidl_generate_interfaces(${PROJECT_NAME}
  ${msg_files}
)
ament_export_dependencies(rosidl_default_runtime)

add_executable(publish_address_book
  src/publish_address_book.cpp
)

ament_target_dependencies(publish_address_book
  "rclcpp"
)

rosidl_target_interfaces(publish_address_book
  ${PROJECT_NAME} "rosidl_typesupport_cpp")

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

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

```
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

编译+刷新环境后分别启动`ros2 run more_interfaces publish_address_book`和`ros2 topic echo /address_book`会在控制台得到：

Publisher:

```
..
Publishing Contact
First:John  Last:Doe
Publishing Contact
First:John  Last:Doe
..
```

Echo:

```
..
first_name: John
last_name: Doe
gender: false
age: 30
address: unknown
---
first_name: John
last_name: Doe
gender: false
age: 30
address: unknown
---
..
```

Parameter

用`ros2 pkg create --build-type ament_cmake cpp_parameters --dependencies rclcpp`新建一个`cpp_parameters`功能包，这行代码添加了`--dependencies`所以无需自行在`package.xml`和`CmakeLists.txt`额外修改
我们在这个功能包内新建一个`cpp_parameters_node.cpp`，并写入：

```cpp
#include <rclcpp/rclcpp.hpp>
#include <chrono>
#include <string>
#include <functional>

using namespace std::chrono_literals;

class ParametersClass: public rclcpp::Node
{
  public:
    ParametersClass()
      : Node("parameter_node")
    {
      this->declare_parameter<std::string>("my_parameter", "world");
      timer_ = this->create_wall_timer(
      1000ms, std::bind(&ParametersClass::respond, this));
    }
    void respond()
    {
      this->get_parameter("my_parameter", parameter_string_);
      RCLCPP_INFO(this->get_logger(), "Hello %s", parameter_string_.c_str());
    }
  private:
    std::string parameter_string_;
    rclcpp::TimerBase::SharedPtr timer_;
};

int main(int argc, char** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<ParametersClass>());
  rclcpp::shutdown();
  return 0;
}
```

这段代码在类中，构造函数中首先创建了一个名为`my_parameter`的参数，默认值为`world`，随后将`timer_`设为绑定`respond()`的每秒调用一次`respond()`函数的周期函数
而`respond()`函数每次调用都会获取`my_parameter`中的值并储存到`parameter_string_`中，然后输出到控制台，随后main函数追踪我们的`ParameterClass`类

打开cmake文件在`find_package(rclcpp REQUIRED)`后添加

```
add_executable(parameter_node src/cpp_parameters_node.cpp)
ament_target_dependencies(parameter_node rclcpp)

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

然后按之前的步骤编译 刷新环境 运行节点
应该能看到每秒发送

```
[INFO] [1765261649.791392532] [parameter_node]: Hello world
```

此时打开另一个终端，键入`ros2 param list`查看当前的参数列表，会发现有一个`my_parameter`，我们可以通过

```
ros2 param set /parameter_node my_parameter earth
```

更改参数内容从`world`变为`earth`，与此同时另一边控制台也会开始输出

```
[INFO] [1765261760.178164263] [parameter_node]: Hello earth
```

我们可以通过Launch来更改参数，进到`cpp_parameters`下，创建一个`launch`文件夹，然后在里面创建一个`cpp_parameters_launch.py`，键入

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package="cpp_parameters",
            executable="parameter_node",
            name="custom_parameter_node",
            output="screen",  # 这行确保输出到控制台
            emulate_tty=True, # 这行确保节点输出环境为控制台
            parameters=[
                {"my_parameter": "earth"}
            ]
        )
    ])
```

表示将`parameter_node`启动时，将`my_parameter`的内容设置为`earth`
打开`CmakeLists.txt`添加

```
install(
  DIRECTORY launch
  DESTINATION share/${PROJECT_NAME}
)
```

表示安装launch文件启动多节点，然后编译，正常会返回

```
[parameter_node-1] [INFO] [1765262927.281273496] [custom_parameter_node]: Hello earth
```

以下是一个合法的用于加载本地参数的`.yaml`文件：

```yaml
sentry_controller:
    ros__parameters:
    # 巡逻点坐标 + 补血点坐标
        x1: 2.0
        y1: 6.0
        x2: 2.0
        y2: 2.0
        x3: 4.0
        y3: 4.0
        r: 4.0 # 半径
        rz: 7.0 # 旋转速度
        max_speed: 2.5 # 最大移动速度
        speed_lerp: 0.75 # 最大速度插值比率
        max_aspeed: 0.7 # 最大加速度
        aspeed_lerp: 1.0 # 最大加速度插值比率
        aaspeed: 0.15 # 加加速度
        max_health: 100.0 # 最大血量
        low_rate: 0.25 # 低血模式标志比率
```

表示这些参数均提供给节点`sentry_controller`作为Ros2的参数使用，其他节点**无法使用**，强行加载会加载失败导致使用默认值，如果要给其他节点使用，那就要另写一个节点并加上上述所有参数，也可以将`sentry_controller`改成`/**`，这表示给文件下所有的节点使用

Action

创建动作接口

创建功能包`my_action_tutorials_interfaces`（注意：`action_tutorials_interfaces`是官方自带的功能包，不能重名），在其下创建`action`文件夹，创建一个`Fibonacci.action`，`.action`文件主要由

```
# Request
---
# Result
---
# Feedback
```

三个部分组成，首先动作客户端会向动作服务器发送Request信息从而启动一个新的目标任

务。
当目标任务完成时，动作服务器会向动作客户端返回一条Result消息。
同时，动作服务器还会向动作客户端定时发送Feedback消息，包含任务目标的进展和更新信息
我们键入

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

接下来在`CmakeLists.txt`中的`find_package(ament_cmake REQUIRED)`后添加

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Fibonacci.action"
)
```

帮忙转换为`.hpp`，并在`package.xml`中加入

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>

<depend>action_msgs</depend>

...

<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

随后编译，刷新环境后运
行`ros2 interface show my_action_tutorials_interfaces/action/Fibonacci`应该会显示

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

动作服务器和动作客户端

为了方便后续编写，我们创建一个新的工作空间`action_ws`并通过

```
ros2 pkg create --dependencies action_tutorials_interfaces rclcpp rclcpp_action rclcpp_co
```

创建新的功能包，这个功能包初始配置好了一些依赖，我们先把前文所完成`Fibonacci.action`移植到这里，我们配置好`./include/action_tutorials_cpp`下的`visibility_control.h`文件（这里不展开具体写入什么），创建一个`fibonacii_action_server.cpp`并键入

```cpp
#include <functional>
#include <memory>
#include <thread>

#include "action_tutorials_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "rclcpp_components/register_node_macro.hpp"

#include "action_tutorials_cpp/visibility_control.h"

namespace action_tutorials_cpp
{
class FibonacciActionServer : public rclcpp::Node
{
public:
  using Fibonacci = action_tutorials_interfaces::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

  ACTION_TUTORIALS_CPP_PUBLIC
  explicit FibonacciActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions
  : Node("fibonacci_action_server", options)
  {
    using namespace std::placeholders;

    this->action_server_ = rclcpp_action::create_server<Fibonacci>(
      this,
      "fibonacci",
      std::bind(&FibonacciActionServer::handle_goal, this, _1, _2),
      std::bind(&FibonacciActionServer::handle_cancel, this, _1),
      std::bind(&FibonacciActionServer::handle_accepted, this, _1));
  }

private:
  rclcpp_action::Server<Fibonacci>::SharedPtr action_server_;

  rclcpp_action::GoalResponse handle_goal(
    const rclcpp_action::GoalUUID & uuid,
    std::shared_ptr<const Fibonacci::Goal> goal)
```

```
  {
    RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
    (void)uuid;
    return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
  }

  rclcpp_action::CancelResponse handle_cancel(
    const std::shared_ptr<GoalHandleFibonacci> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
    (void)goal_handle;
    return rclcpp_action::CancelResponse::ACCEPT;
  }

  void handle_accepted(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
  {
    using namespace std::placeholders;
    // this needs to return quickly to avoid blocking the executor, so spin up a new threa
    std::thread{std::bind(&FibonacciActionServer::execute, this, _1), goal_handle}.detach
  }

  void execute(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Executing goal");
    rclcpp::Rate loop_rate(1);
    const auto goal = goal_handle->get_goal();
    auto feedback = std::make_shared<Fibonacci::Feedback>();
    auto & sequence = feedback->partial_sequence;
    sequence.push_back(0);
    sequence.push_back(1);
    auto result = std::make_shared<Fibonacci::Result>();

    for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
      // Check if there is a cancel request
      if (goal_handle->is_canceling()) {
        result->sequence = sequence;
        goal_handle->canceled(result);
        RCLCPP_INFO(this->get_logger(), "Goal canceled");
        return;
      }
```

```
      // Update sequence
      sequence.push_back(sequence[i] + sequence[i - 1]);
      // Publish feedback
      goal_handle->publish_feedback(feedback);
      RCLCPP_INFO(this->get_logger(), "Publish feedback");

      loop_rate.sleep();
    }

    // Check if goal is done
    if (rclcpp::ok()) {
      result->sequence = sequence;
      goal_handle->succeed(result);
      RCLCPP_INFO(this->get_logger(), "Goal succeeded");
    }
  }
};  // class FibonacciActionServer

}  // namespace action_tutorials_cpp

RCLCPP_COMPONENTS_REGISTER_NODE(action_tutorials_cpp::FibonacciActionServer)
```

首先`FibonacciActionServer`类的构造函数将节点名称初始化为`fibonacci_action_server`并实例化新的动作服务器
一个动作服务器需要六件事：

- 模板化动作类型名称`Fibonacci`
- 一个用于将动作添加到`this`的节点
- 动作名称为`fibonacci`
- 处理目标的返回函数`handle_goal`
- 处理取消的返回函数`handle_cancel`
- 处理目标接受的返回函数`handle_accepted`

注意，所有调用都需要立即返回，否则会使执行进程有空白段
后面两部分就是关于`handle_goal`和`handle_cancel`的实现，由于执行是一个长期的操作，我们生成一个线程来完成实际工作，并快速从`handle_accepted`返回：

```
void handle_accepted(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
  using namespace std::placeholders;
  // this needs to return quickly to avoid blocking the executor, so spin up a new thread
  std::thread{std::bind(&FibonacciActionServer::execute, this, _1), goal_handle}.detach()
}
```

所有进一步的处理和更新都在新线程的`execute`方法中完成，它每秒处理一个Fibonacci数列的序列号，并发布每个步骤的反馈，完成后将`goal_handle`标记为成功，随后进入cmake的设置，我们在`find_package`后添加

```
add_library(action_server SHARED
  src/fibonacci_action_server.cpp)
target_include_directories(action_server PRIVATE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>)
target_compile_definitions(action_server
  PRIVATE "ACTION_TUTORIALS_CPP_BUILDING_DLL")
ament_target_dependencies(action_server
  "action_tutorials_interfaces"
  "rclcpp"
  "rclcpp_action"
  "rclcpp_components")
rclcpp_components_register_node(action_server PLUGIN "action_tutorials_cpp::FibonacciActio
install(TARGETS
  action_server
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin)
```

随后编译（这次编译需要直接`colcon build`编译所有文件）
接下来编写动作客户端，我们新建`fibonacci_action_client.cpp`并键入：

```cpp
#include <functional>
#include <future>
#include <memory>
#include <string>
#include <sstream>

#include "action_tutorials_interfaces/action/fibonacci.hpp"

#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "rclcpp_components/register_node_macro.hpp"

namespace action_tutorials_cpp
{
class FibonacciActionClient : public rclcpp::Node
{
public:
  using Fibonacci = action_tutorials_interfaces::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;

  explicit FibonacciActionClient(const rclcpp::NodeOptions & options)
  : Node("fibonacci_action_client", options)
  {
    this->client_ptr_ = rclcpp_action::create_client<Fibonacci>(
      this,
      "fibonacci");

    this->timer_ = this->create_wall_timer(
      std::chrono::milliseconds(500),
      std::bind(&FibonacciActionClient::send_goal, this));
  }

  void send_goal()
  {
    using namespace std::placeholders;

    this->timer_->cancel();

    if (!this->client_ptr_->wait_for_action_server()) {
```

```
      RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
      rclcpp::shutdown();
    }

    auto goal_msg = Fibonacci::Goal();
    goal_msg.order = 10;

    RCLCPP_INFO(this->get_logger(), "Sending goal");

    auto send_goal_options = rclcpp_action::Client<Fibonacci>::SendGoalOptions();
    send_goal_options.goal_response_callback =
      std::bind(&FibonacciActionClient::goal_response_callback, this, _1);
    send_goal_options.feedback_callback =
      std::bind(&FibonacciActionClient::feedback_callback, this, _1, _2);
    send_goal_options.result_callback =
      std::bind(&FibonacciActionClient::result_callback, this, _1);
    this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
  }

private:
  rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;
  rclcpp::TimerBase::SharedPtr timer_;

  void goal_response_callback(GoalHandleFibonacci::SharedPtr goal_handle)
  {
    if (!goal_handle) {
      RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
    } else {
      RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
    }
  }

  void feedback_callback(
    GoalHandleFibonacci::SharedPtr,
    const std::shared_ptr<const Fibonacci::Feedback> feedback)
  {
    std::stringstream ss;
    ss << "Next number in sequence received: ";
    for (auto number : feedback->partial_sequence) {
      ss << number << " ";
```

```
    }
    RCLCPP_INFO(this->get_logger(), ss.str().c_str());
  }

  void result_callback(const GoalHandleFibonacci::WrappedResult & result)
  {
    switch (result.code) {
      case rclcpp_action::ResultCode::SUCCEEDED:
        break;
      case rclcpp_action::ResultCode::ABORTED:
        RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
        return;
      case rclcpp_action::ResultCode::CANCELED:
        RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
        return;
      default:
        RCLCPP_ERROR(this->get_logger(), "Unknown result code");
        return;
    }
    std::stringstream ss;
    ss << "Result received: ";
    for (auto number : result.result->sequence) {
      ss << number << " ";
    }
    RCLCPP_INFO(this->get_logger(), ss.str().c_str());
    rclcpp::shutdown();
  }
};  // class FibonacciActionClient

}  // namespace action_tutorials_cpp

RCLCPP_COMPONENTS_REGISTER_NODE(action_tutorials_cpp::FibonacciActionClient)
```

注意，在Foxy版本下，`goal_response_callback()`的实现应该是：

```
void goal_response_callback(std::shared_future<GoalHandleFibonacci::SharedPtr> future)
{
  auto goal_handle = future.get();
  if (!goal_handle) {
    RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
  } else {
    RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
  }
}
```

这是因为Jazzy版本下无需通过`.get()`获取目标句柄，这个函数的参数类型不再是`std::shared_future<GoalHandle::SharedPtr>`而是`GoalHandle::SharedPtr`
一个动作客户端需要三件事：

- 模板化动作类型名称：`Fibonacci`
- 一个用于将动作客户端添加到`this`的节点
- 动作名称：`fibonacci`

同时设置了一个计时器，其将唯一调用`send_goal()`函数

```
this->timer_ = this->create_wall_timer(
std::chrono::milliseconds(500),
std::bind(&FibonacciActionClient::send_goal, this));
```

而在`send_goal()`函数中，它会执行以下操作：

- 取消计时器（因此只调用一次）
- 等待动作服务器启动
- 实例化一个新的Fibonacci数列目标
- 设置响应、反馈和结果调用
- 将目标发送到服务器

当服务器接收并接受目标时，它向客户端发送响应，其由`goal_response_callback()`函数处理
假设目标已被服务器接受，它将开始处理。给客户的任何反馈将由`feedback_callback()`处理
当服务器完成处理后，它将向客户端返回结果。结果由`result_callback()`处理
接下来在cmake文件中的`find_package`后添加

```
add_library(action_client SHARED
  src/fibonacci_action_client.cpp)
target_include_directories(action_client PRIVATE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>)
target_compile_definitions(action_client
  PRIVATE "ACTION_TUTORIALS_CPP_BUILDING_DLL")
ament_target_dependencies(action_client
  "action_tutorials_interfaces"
  "rclcpp"
  "rclcpp_action"
  "rclcpp_components")
rclcpp_components_register_node(action_client PLUGIN "action_tutorials_cpp::FibonacciActi
install(TARGETS
  action_client
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin)
```

随后编译整个包，开启两个终端一个运行服务器一个运行客户端，应该会得到：

Action Server:

```
[INFO] [1765343856.942174981] [fibonacci_action_server]: Publish feedback
[INFO] [1765343857.942152223] [fibonacci_action_server]: Publish feedback
[INFO] [1765343858.942124113] [fibonacci_action_server]: Publish feedback
```

Action Client:

```
[INFO] [1765343856.942508960] [fibonacci_action_client]: Next number in sequence received
[INFO] [1765343857.942594248] [fibonacci_action_client]: Next number in sequence received
[INFO] [1765343858.942644077] [fibonacci_action_client]: Next number in sequence received
```

TF2

静态广播器

我们在`dev_ws`工作空间的`src`目录下通过

```
ros2 pkg create --build-type ament_cmake learning_tf2_cpp
```

创建一个功能包，在该功能包的`src`目录下用

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/sta
```

保存一个`.cpp`为tf2持续发布相对变换的信息的演示文件，内容为

```cpp
// Copyright 2021 Open Source Robotics Foundation, Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include <memory>

#include "geometry_msgs/msg/transform_stamped.hpp"
#include "rclcpp/rclcpp.hpp"
#include "tf2/LinearMath/Quaternion.h"
#include "tf2_ros/static_transform_broadcaster.h"

class StaticFramePublisher : public rclcpp::Node
{
public:
  explicit StaticFramePublisher(char * transformation[])
  : Node("static_turtle_tf2_broadcaster")
  {
    tf_static_broadcaster_ = std::make_shared<tf2_ros::StaticTransformBroadcaster>(this);

    // Publish static transforms once at startup
    this->make_transforms(transformation);
  }

private:
  void make_transforms(char * transformation[])
  {
    geometry_msgs::msg::TransformStamped t;

    t.header.stamp = this->get_clock()->now();
```

```
    t.header.frame_id = "world";
    t.child_frame_id = transformation[1];

    t.transform.translation.x = atof(transformation[2]);
    t.transform.translation.y = atof(transformation[3]);
    t.transform.translation.z = atof(transformation[4]);
    tf2::Quaternion q;
    q.setRPY(
      atof(transformation[5]),
      atof(transformation[6]),
      atof(transformation[7]));
    t.transform.rotation.x = q.x();
    t.transform.rotation.y = q.y();
    t.transform.rotation.z = q.z();
    t.transform.rotation.w = q.w();

    tf_static_broadcaster_->sendTransform(t);
  }

  std::shared_ptr<tf2_ros::StaticTransformBroadcaster> tf_static_broadcaster_;
};

int main(int argc, char * argv[])
{
  auto logger = rclcpp::get_logger("logger");

  // Obtain parameters from command line arguments
  if (argc != 8) {
    RCLCPP_INFO(
      logger, "Invalid number of parameters\nusage: "
      "$ ros2 run learning_tf2_cpp static_turtle_tf2_broadcaster "
      "child_frame_name x y z roll pitch yaw");
    return 1;
  }

  // As the parent frame of the transform is `world`, it is
  // necessary to check that the frame name passed is different
  if (strcmp(argv[1], "world") == 0) {
    RCLCPP_INFO(logger, "Your static turtle name cannot be 'world'");
    return 2;
```

```
  }

  // Pass parameters and initialize node
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<StaticFramePublisher>(argv));
  rclcpp::shutdown();
  return 0;
}
```

该功能包主要用于大致实现`ros2 run tf2_ros tf2_echo turtle2 turtle1`持续返回信息的功能
先从头文件看起，`#include "geometry_msgs/msg/transform_stamped.hpp"`是为了使用`TransformStamped`消息类型，用于发布消息到转换树中

```
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_ros/static_transform_broadcaster.h>
```

这里第一个头文件用于将四元数与欧拉角互相转换，第二个头文件是使用`StaticTransformBroadcaster`消息类型，使静态转换的发布变得容易
随后声明了一个类`StaticFramePublisher`初始化了一个名为`static_turtle_tf2_broadcaster`的节点，然后构造函数创建了一个`TransformStamped`对象，启动时发送一次**静态转换**信息
接着在发送函数中，首先获取时间戳并赋予`t.header.stamp`，其中`t`是`TransformStamped`消息类型的一个对象，设置其链接父帧的名称为`world`即世界坐标系，再创建子帧的名称`transformation[1]`实际上就是你接下来要创建的海龟的名称，所以本功能包其实是**子帧**相对于**父帧**的静态变换，输出的变换（平移、旋转等）向量表示子帧到父帧中的位置和朝向
接下来就是一些海龟的变换向量数据填充，最后使用`tf_static_broadcaster_->sendTransform(t);`发送消息
最后老样子，我们添加

```
<depend>geometry_msgs</depend>
<depend>rclcpp</depend>
<depend>tf2</depend>
<depend>tf2_ros</depend>
<depend>turtlesim</depend>
```

到`package.xml`，cmake文件应为：

```
cmake_minimum_required(VERSION 3.8)
project(learning_tf2_cpp)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tf2 REQUIRED)
find_package(tf2_ros REQUIRED)
find_package(turtlesim REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

add_executable(static_turtle_tf2_broadcaster src/static_turtle_tf2_broadcaster.cpp)
ament_target_dependencies(
   static_turtle_tf2_broadcaster
   geometry_msgs
   rclcpp
   tf2
   tf2_ros
   turtlesim
)

install(TARGETS
   static_turtle_tf2_broadcaster
   DESTINATION lib/${PROJECT_NAME})

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

```
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

编译`colcon build --packages-select learning_tf2_cpp`，运行`ros2 run learning_tf2_cpp static_turtle_tf2_broadcaster mystaticturtle 0 0 1 0 0 0`，我们用`ros2 topic echo --qos-reliability reliable --qos-durability transient_local /tf_static`来检查话题是否成功发表，如果确实，则返回：

```
transforms:
- header:
    stamp:
      sec: 1766824388
      nanosec: 290890619
    frame_id: world
  child_frame_id: mystaticturtle
  transform:
    translation:
      x: 0.0
      y: 0.0
      z: 1.0
    rotation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
---
```

广播器

依旧先通过

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/tur
```

保存一个示例文件到上一节的功能包的`src`目录下，代码应为

```
// Copyright 2021 Open Source Robotics Foundation, Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include <functional>
#include <memory>
#include <sstream>
#include <string>

#include "geometry_msgs/msg/transform_stamped.hpp"
#include "rclcpp/rclcpp.hpp"
#include "tf2/LinearMath/Quaternion.h"
#include "tf2_ros/transform_broadcaster.h"
#include "turtlesim/msg/pose.hpp"

class FramePublisher : public rclcpp::Node
{
public:
  FramePublisher()
  : Node("turtle_tf2_frame_publisher")
  {
    // Declare and acquire `turtlename` parameter
    turtlename_ = this->declare_parameter<std::string>("turtlename", "turtle");

    // Initialize the transform broadcaster
    tf_broadcaster_ =
      std::make_unique<tf2_ros::TransformBroadcaster>(*this);

    // Subscribe to a turtle{1}{2}/pose topic and call handle_turtle_pose
```

```
    // callback function on each message
    std::ostringstream stream;
    stream << "/" << turtlename_.c_str() << "/pose";
    std::string topic_name = stream.str();

    auto handle_turtle_pose = [this](const std::shared_ptr<turtlesim::msg::Pose> msg) {
        geometry_msgs::msg::TransformStamped t;

        // Read message content and assign it to
        // corresponding tf variables
        t.header.stamp = this->get_clock()->now();
        t.header.frame_id = "world";
        t.child_frame_id = turtlename_.c_str();

        // Turtle only exists in 2D, thus we get x and y translation
        // coordinates from the message and set the z coordinate to 0
        t.transform.translation.x = msg->x;
        t.transform.translation.y = msg->y;
        t.transform.translation.z = 0.0;

        // For the same reason, turtle can only rotate around one axis
        // and this why we set rotation in x and y to 0 and obtain
        // rotation in z axis from the message
        tf2::Quaternion q;
        q.setRPY(0, 0, msg->theta);
        t.transform.rotation.x = q.x();
        t.transform.rotation.y = q.y();
        t.transform.rotation.z = q.z();
        t.transform.rotation.w = q.w();

        // Send the transformation
        tf_broadcaster_->sendTransform(t);
      };
    subscription_ = this->create_subscription<turtlesim::msg::Pose>(
      topic_name, 10,
      handle_turtle_pose);
  }

private:
  rclcpp::Subscription<turtlesim::msg::Pose>::SharedPtr subscription_;
```

```
  std::unique_ptr<tf2_ros::TransformBroadcaster> tf_broadcaster_;
  std::string turtlename_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<FramePublisher>());
  rclcpp::shutdown();
  return 0;
}
```

整个构造函数内实际上就是在定义一个lambda表达式，并作为回调函数给subscription_，表示海龟的`/pose`接收到消息时就发送一次变换消息的广播

这个**广播器**和上一节的**静态广播器**的区别是，前者可以周期发送到话题`/tf`，开销较大，适用于实时变化的变换关系，后者只能开头发送或者手动发送到话题`/tf_static`，开销较小，适用于固定不变的变换关系

随后我们需要增加一个`turtle_tf2_demo.launch.py`作为launch文件：

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


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
    ])
```

配置好cmake文件：

```
cmake_minimum_required(VERSION 3.8)
project(learning_tf2_cpp)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tf2 REQUIRED)
find_package(tf2_ros REQUIRED)
find_package(turtlesim REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

add_executable(turtle_tf2_broadcaster src/turtle_tf2_broadcaster.cpp)
ament_target_dependencies(
    turtle_tf2_broadcaster
    geometry_msgs
    rclcpp
    tf2
    tf2_ros
    turtlesim
)

install(TARGETS
    turtle_tf2_broadcaster
    DESTINATION lib/${PROJECT_NAME})

install(DIRECTORY launch
    DESTINATION share/${PROJECT_NAME}
)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
```

```
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

和`package.xml`文件：

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http:/
<package format="3">
  <name>learning_tf2_cpp</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="goose@todo.todo">goose</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

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

  <depend>geometry_msgs</depend>
  <depend>rclcpp</depend>
  <depend>tf2</depend>
  <depend>tf2_ros</depend>
  <depend>turtlesim</depend>

  <exec_depend>launch</exec_depend>
  <exec_depend>launch_ros</exec_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

编译后运

行`ros2 launch learning_tf2_cpp turtle_tf2_demo.launch.py`

与`ros2 run turtlesim turtle_teleop_key`

这下我们可以自由操控海龟，再运行`ros2 run tf2_ros tf2_echo world turtle1`我们能看到控制台在不断返回

```
At time 1766826317.156279554
- Translation: [6.460, 3.758, 0.000]
- Rotation: in Quaternion (xyzw) [0.000, 0.000, -0.467, 0.884]
- Rotation: in RPY (radian) [0.000, 0.000, -0.971]
- Rotation: in RPY (degree) [0.000, 0.000, -55.645]
- Matrix:
  0.564  0.826  0.000  6.460
 -0.826  0.564 -0.000  3.758
 -0.000  0.000  1.000  0.000
  0.000  0.000  0.000  1.000
```

类似的消息，这表示的是 turtle1 坐标系相对于世界坐标系的位置和朝向，也就是 turtle1 在世界坐标系的坐标和角度等数值

监听

继续使用

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/tur
```

保存示例程序到`src`下，其中构造函数中设置的两个布尔参数为`false`表示海龟服务是否就绪，以及海龟是否已生成，由于需要异步生成海龟，所以一开始肯定是没生成也没就绪的
随后声明了一个`tf_buffer`缓冲区用来储存变换数据，用`tf_listener_`监听到的tf2话题并填充缓冲区，用缓冲区是因为在时间同步、发布频率等种种问题上不用缓冲区会导致数据接收短暂丢失，引发错误
对于`on_timer`这个每秒调用一次的计时器回调函数，在开头的`fromFrameRel`和`toFreamRel`中定义了源帧和目标帧的海龟名字
随后再检测完 turtle2 是否正常生成后尝试获取 turtle2 到 turtle1 的变换参数，如果查询到了就进行计算整合后再发布给 turtle2 控制其移动
如果 turtle2 没有生成，那就异步发送生成指令
我们在launch文件中添加一些节点同时启动，最终修改为：

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
        DeclareLaunchArgument(
            'target_frame', default_value='turtle1',
            description='Target frame name.'
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_broadcaster',
            name='broadcaster2',
            parameters=[
                {'turtlename': 'turtle2'}
            ]
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_listener',
            name='listener',
            parameters=[
                {'target_frame': LaunchConfiguration('target_frame')}
            ]
```

```
        ),
    ])
```

对cmake来说，我们需要添加

```
add_executable(turtle_tf2_listener src/turtle_tf2_listener.cpp)
ament_target_dependencies(turtle_tf2_listener
  rclcpp
  tf2_ros
  geometry_msgs
  turtlesim
)
```

创建监听器的可执行文件，以及在安装中添加`turtle_tf2_listener`
至于`package.xml`是不需要改的，因为依赖足够
最后编译后运行，会先生成一个 turtle1 随后生成一个 turtle2，这个 turtle2 会跟随 turtle1 移动

PluginLib

PluginLib允许从ros包内加载和卸载插件的C++库，使用它可以避免再显式将应同程序和包含这些类的库进行链接
我们在路径`ros2_ws/src`下使用

```
ros2 pkg create --build-type ament_cmake --dependencies pluginlib --node-name area_node -
ros2 pkg create --build-type ament_cmake --dependencies polygon_base pluginlib --library-n
```

创建两个功能包，第一个作为软件包第二个作为插件包
我们在第一个功能包的`include`下创建`regular_polygon.hpp`并写入

```
#ifndef POLYGON_BASE_REGULAR_POLYGON_HPP
#define POLYGON_BASE_REGULAR_POLYGON_HPP

namespace polygon_base
{
  class RegularPolygon
  {
    public:
      virtual void initialize(double side_length) = 0;
      virtual double area() = 0;
      virtual ~RegularPolygon(){}

    protected:
      RegularPolygon(){}
  };
}  // namespace polygon_base

#endif  // POLYGON_BASE_REGULAR_POLYGON_HPP
```

在cmake文件中改为

```
cmake_minimum_required(VERSION 3.8)
project(polygon_base)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(pluginlib REQUIRED)

add_executable(area_node src/area_node.cpp)
target_include_directories(area_node PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include/${PROJECT_NAME}>)
target_compile_features(area_node PUBLIC c_std_99 cxx_std_17)  # Require C99 and C++17
ament_target_dependencies(
  area_node
  "pluginlib"
)

install(
  DIRECTORY include/
  DESTINATION include
)

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

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
```

```
endif()

ament_export_include_directories(
  include
)

ament_package()
```

在第二个包的`src`下找到`polygon_plugins.cpp`写入

```cpp
#include <polygon_base/regular_polygon.hpp>
#include <cmath>

namespace polygon_plugins
{
  class Square : public polygon_base::RegularPolygon
  {
    public:
      void initialize(double side_length) override
      {
        side_length_ = side_length;
      }

      double area() override
      {
        return side_length_ * side_length_;
      }

    protected:
      double side_length_;
  };

  class Triangle : public polygon_base::RegularPolygon
  {
    public:
      void initialize(double side_length) override
      {
        side_length_ = side_length;
      }

      double area() override
      {
        return 0.5 * side_length_ * getHeight();
      }

      double getHeight()
      {
        return sqrt((side_length_ * side_length_) - ((side_length_ / 2) * (side_length_ /
      }
```

```
    protected:
      double side_length_;
  };
}

#include <pluginlib/class_list_macros.hpp>

PLUGINLIB_EXPORT_CLASS(polygon_plugins::Square, polygon_base::RegularPolygon)
PLUGINLIB_EXPORT_CLASS(polygon_plugins::Triangle, polygon_base::RegularPolygon)
```

这里
Square 和 Triangle 类的实现为：保存边长，并使用它来计算面积
唯一与 pluginlib 相关的部分是最后三行代码，它们调用了一些**注册类**为**实际插件**的宏
该宏的参数为例：

```
PLUGINLIB_EXPORT_CLASS(polygon_plugins::Square, polygon_base::RegularPolygon)
```

第一个是插件类的完全限定类型，第二个是基类的完全限定类型
随后我们创建一个`plugins.xml`文件，写入：

```
<library path="polygon_plugins">
  <class type="polygon_plugins::Square" base_class_type="polygon_base::RegularPolygon">
    <description>This is a square plugin.</description>
  </class>
  <class type="polygon_plugins::Triangle" base_class_type="polygon_base::RegularPolygon">
    <description>This is a triangle plugin.</description>
  </class>
</library>
```

- `library`标签提供了一个相对路径，指向包含我们要导出的插件的库
  在ros2中，这只是库的名称。在ros中，它包含前缀`lib`或者有时
  是`lib/lib`（即`lib/libpolygon_plugins`），但在这里更简单。
- `class`标签声明了我们想要从库中导出的插件，让我们来看一下它的参数：
  - `type`：插件的完全限定类型，在这里是`polygon_plugins::Square`
  - `base_class`：插件的完全限定基类类型，在这里是`polygon_base::RegularPolygon`
  - `description`：插件的描述以及它的功能。

同时在`package.xml`下的`<export>`标签下输入

```
<polygon_base plugin="${prefix}/plugins.xml" />
```

最后通过cmake导出插件，在该包的`CMakeLists.txt`中的`find_package(pluginlib REQUIRED)`添加：

```
pluginlib_export_plugin_description_file(polygon_base plugins.xml)
```

目前的cmake文件最好是如下形式的：

```
cmake_minimum_required(VERSION 3.8)
project(polygon_plugins)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# 1. 查找依赖
find_package(ament_cmake REQUIRED)
find_package(ament_cmake_ros REQUIRED)
find_package(polygon_base REQUIRED)
find_package(pluginlib REQUIRED)
find_package(rclcpp REQUIRED) # 插件通常也需要 rclcpp

# 2. 创建库目标 (关键：建议显式指定 SHARED)
# 确保包含所有实现插件类的 .cpp 文件，比如 src/triangle.cpp 等
# 如果你所有的代码都在 polygon_plugins.cpp 里，就保持原样
add_library(polygon_plugins SHARED
  src/polygon_plugins.cpp
)

# 3. 设置包含目录
target_include_directories(polygon_plugins PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include/${PROJECT_NAME}>)

# 4. 绑定依赖
ament_target_dependencies(
  polygon_plugins
  "polygon_base"
  "pluginlib"
  "rclcpp"
)

# 5. 导出插件描述文件 (关键：建议放在 add_library 之后)
# 这样 CMake 能够确保在处理插件逻辑时，库目标已经存在
pluginlib_export_plugin_description_file(polygon_base plugins.xml)

# 6. 安装头文件
```

```
install(
  DIRECTORY include/
  DESTINATION include/${PROJECT_NAME}
)

# 7. 安装库目标
install(
  TARGETS polygon_plugins
  EXPORT export_${PROJECT_NAME}
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin
)

# 8. 导出 ament 信息
ament_export_include_directories("include/${PROJECT_NAME}")
ament_export_libraries(polygon_plugins)
ament_export_targets(export_${PROJECT_NAME})

ament_package()
```

在ros中和ros2不同的地方在于ros是通过`package.xml`导出插件，上述命令的两个参数分别为

- 基类的包，即`polygon_base`
- 插件声明xml的相对路径，即`plugins.xml`

最后我们回到第一个包中，对其`src`下的`area_node.cpp`写入

```cpp
#include <pluginlib/class_loader.hpp>
#include <polygon_base/regular_polygon.hpp>

int main(int argc, char** argv)
{
  // To avoid unused parameter warnings
  (void) argc;
  (void) argv;

  pluginlib::ClassLoader<polygon_base::RegularPolygon> poly_loader("polygon_base", "polygo

  try
  {
    std::shared_ptr<polygon_base::RegularPolygon> triangle = poly_loader.createSharedInsta
    triangle->initialize(10.0);

    std::shared_ptr<polygon_base::RegularPolygon> square = poly_loader.createSharedInstanc
    square->initialize(10.0);

    printf("Triangle area: %.2f\n", triangle->area());
    printf("Square area: %.2f\n", square->area());
  }
  catch(pluginlib::PluginlibException& ex)
  {
    printf("The plugin failed to load for some reason. Error: %s\n", ex.what());
  }

  return 0;
}
```

`ClassLoader`是理解的关键类，定义在`class_loader.hpp`中
它以基类`polygon_base::RegularPolygon`进行模板化
第一个参数是**基类**的包名字符串，例如`polygon_base`
第二个参数是**插件**的**完全限定基类类型**的字符串，例如`polygon_base::RegularPolygon`

有许多实例化类的方法，在这个例子中，我们使用了共享指针
我们只需要使用完全限定的插件类类型调用`createSharedInstance`，在这里
是`polygon_plugins::Square`

注意：定义这个节点的`polygon_base`包不依赖于`polygon_plugins`类
插件将会在不需要声明任何依赖关系的情况下进行动态加载，此外，我们通过硬编码插件名称来实例化类，但也可以使用参数等动态方式来实现

最后编译

```
colcon build --packages-select polygon_base polygon_plugins
```

并运行（刷新环境后）

```
ros2 run polygon_base area_node
```

会出现

```
Triangle area: 43.30
Square area: 100.00
```

从头到尾梳理一遍pluginlib

定义标准

定义一个所有插件都必须遵守的规则，在这里是位于`polygon_base`的包中
核心要点是：

- 必须有一个虚析构函数
- 使用例`virtual void initialize(...) = 0`或类似的**纯虚函数**作为统一接口

实现插件

开始实现具体逻辑，在这里位于`polygon_plugins`包中
核心要点：

- 继承基类并实现方法
- 导出类，在`.cpp`文件的末尾使用宏，告诉编译器它是插件，在这里是`PLUGINLIB_EXPORT_CLASS(polygon_plugins::Triangle, polygon_base::RegularPolygon)`

编写插件描述文件

与ROS 1不同的是，ROS 2需要用到`plugins.xml`作为插件的描述文件，它位

于`polygon_plugins`包中
核心要点：

- `path`：对应cmake里的库名
- `type`：插件的完整类名（带命名空间）
- `base_class_type`：基类的完整类名
  例如：

```
<library path="polygon_plugins">
  <class type="polygon_plugins::Triangle" base_class_type="polygon_base::RegularPolygon">
    <description>这是一个三角形插件</description>
  </class>
</library>
```

编写cmake文件

最后需要我们**创建库、链接依赖、导出描述文件**，都在`CMakeLists.txt`中完成
例如：

```
# 1. 创建动态库 (必须是 SHARED)
add_library(polygon_plugins SHARED src/triangle.cpp)

# 2. 告诉 CMake 你的描述文件在哪里 (Jazzy/Humble 专用宏)
pluginlib_export_plugin_description_file(polygon_base plugins.xml)

# 3. 安装库到指定位置
install(TARGETS polygon_plugins DESTINATION lib/${PROJECT_NAME})
```

加载插件

之后我们需要加载插件并使用它，在主程序例`area_node`（位于`polygon_base`）中把插件加载
使用`pluginlib::ClassLoader`来加载插件，它会去`ament_index`里搜索所有声明了`<polygon_base plugin=...>`的包
例：

```cpp
// polygon_base/src/area_node.cpp
pluginlib::ClassLoader<polygon_base::RegularPolygon> loader("polygon_base", "polygon_base

try {
    // 动态加载三角形插件，不需要在编译时链接 triangle.cpp
    std::shared_ptr<polygon_base::RegularPolygon> triangle = loader.createSharedInstance("
    triangle->initialize(10.0);
    printf("Area: %.2f\n", triangle->area());
} catch (pluginlib::PluginlibException& ex) {
    // 报错：通常是路径不对、类名写错或没写导出宏
}
```

这里的`.createSharedInstance`表示用共享指针实例化对象

主题统计

部分ROS 2语法

对一个节点的`.cpp`或`.hpp`文件如果要让他派上用场首先需要导入库`rclcpp.hpp`，这是一个ros2的**基本函数库**，我们声明对应的类，且这个类应该继承`rclcpp::Node`，在类的构造函数中我们应顺便调用`Node`的构造函数，这个构造函数需要一个字符串参数，表示这个节点的名字

启动一个节点

当我们有了一个`.cpp`文件作为节点，我们可以用

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

    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<EController>());
    rclcpp::shutdown();
    return 0;
}
```

来启动一个名为`EController`的节点（实际上就是类名），并用`spin`表示节点循环（这样会造成线程堵塞），最后等节点死亡再`shutdown()`来进行释放

节点通信

节点之间的通信模型有Action和Publisher-Subscriber两种，以后者为例，通信首先需要一个Publisher以及一个Subscriber，我们需要在类内声明这两个类型的变量，例如：

```
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr speedPublisher;
rclcpp::Subscription<turtlesim::msg::Pose>::SharedPtr sentryPose;
```

分别声明了一个消息类型为`geometry_msgs::msg::Twist`的Publisher以及一个消息类型为`turtlesim::msg::Pose`的Subscriber
这自然引出声明这两种类型的变量都需要输入对应的消息类型，以下有几个常用的消息类型

	用途	消息类型	需要的
1	海龟的坐标，角度等信息	`turtlesim::msg::Pose`	`turtlesim/msg/pose`
2	海龟的移动向量，旋转向量	`geometry_msgs::msg::Twist`	`geometry_msgs/msg/`
3	以字符串方式存储消息	`std_msgs::msg::String`	`std_msgs/msg/string`
4	自定义消息类型	工作空间名称`::msg`文件夹名称`::.msg`文件名	工作空间名称`/msg`文件夹

注：自定义消息类型通常定义在`.msg`文件内，其需要先经过cmake编译生成一个`.hpp`文件，这个文件通常位于工作空间目录下的install文件夹下的功能包文件夹下的msg文件夹下，并且这个名字均为小写+下划线+数字的组合，通常与`.msg`的文件名别无二致，即将大驼峰命名方式转化为下划线命名方式（以大写字母为区分，变为小写字母后前面加个下划线），以下是一个自定义消息类型的文件：

```
float64 max_speed    # 最大移动速度
float64 speed_lerp   # 最大速度插值比率
float64 max_aspeed   # 最大加速度
float64 aspeed_lerp # 最大加速度插值比率
```

在声明好节点后我们可以在构造函数中进行两者的初始化，以上面的例子为例，用

```
speedPublisher = this->create_publisher<turtle_sentry::msg::SentrySpeed>(
    "/speed_adj/params", 10
);
```

可以创建一个发布者，并指定发布信息到`/speed_adj/params`上，消息队列最多容纳10条信息，而对于接收者，需要

```
sentryPose = this->create_subscription<turtlesim::msg::Pose>(
    "/shaobing_zjl/pose", 10,
    std::bind(&Sentry::sentry_pose_callback, this, std::placeholders::_1)
);
```

前几个参数和发布者别无二致，后面的`std::bind`是为了绑定一个叫`sentry_pose_callback`的回调函数，在我们的类内进行了定义，`bind`的最后两个参数表示先传入一个`this`指针（对象实例）来确保能调用类成员函数（调用回调函数），而第二个参数是占位符（需要导入头文件`functional`），表示**回调调用时传入的第一个参数**（即消息指针）会自动填充，为了替代`std::bind`，可以使用Lambda写法，即：

```
sentryPose = this->create_subscription<turtlesim::msg::Pose>(
    "/shaobing_zjl/pose", 10,
    [this](const turtlesim::msg::Pose::SharedPtr msg) {
        sentry_pose_callback(msg);  // 回调函数或直接写逻辑
    }
);
```

总之创建的Publisher能通过`peedPublisher->publish(msg)`发布`msg`信息，而Subscriber将会在收到信息的时候调用回调函数

计时器

通过`rclcpp::TimerBase::Sharedtr sendParams`可以创建一个名为`sendParams`的计时器指针（`SharedPtr`不是必要的），我们在构造函数中可以通过

```
sendParams = this->create_wall_timer(
    10ms,
    std::bind(&Sentry::send_params, this)
);
```

来设置它每`10ms`调用一次`send_params`这个回调函数（`10ms`的写法需要使用`using namespace std::chrono_literals`先声明命名空间，表示10ms）

多线程

通过`std::thread keyThread`可以声明一个线程变量，需要导入头文件`thread`
它允许我们分离一个线程单独运行，防止线程堵塞（通常用于键盘输入等）
我们可以在构造函数中用

```
key_loop();
keyThread = std::thread(&EController::key_loop, this);
keyThread.detach();
```

先调用一个应该在多线程通过`while (rclcpp::ok)`（`rclcpp::ok`表示节点正常运行）循环执行的函数再绑定好`keyThread`线程对应的函数`key_loop`，最后通过`.detach`分离线程，这里有两个原因：

- 如果不使用`detach()`分离进程，该进程会随着`std::thread`析构函数的调用随之终止，而不是通过自行输入`q`来退出
- 我们在`main`函数中用了`spin`来持续运行该节点，如果不分离，这个`key_loop()`的函数会一直阻塞主节点，只有分离了才会作为单独的节点

如上文提到的，`spin`用于将一个节点循环作用，但是这样会堵塞主进程，我们如果想要实现一个键盘输入功能，就必须分离一个新进程，至于这个新进程如何操作？我们可以
用`launch.py`文件启动这个节点的同时启动一个新终端

Doctor

控制台输入`ros2 doctor`可以检查doctor有没有安装正确，如果设置处于完美状态，会返回

```
All <n> checks passed
```

可能会收到一些额外的警告，但只要出现上面的消息，就不影响正常使用
当运行一个`turtlesim_node`节点和一个`turtle_teleop_key`节点时，再运行`ros2 doctor`会出现除了上次运行出现的警告外的

```
/opt/ros/jazzy/lib/python3.12/site-packages/ros2doctor/api/topic.py: 42:
UserWarning: Publisher without subscriber detected on /turtle1/color_sensor.
/opt/ros/jazzy/lib/python3.12/site-packages/ros2doctor/api/topic.py: 42:
UserWarning: Publisher without subscriber detected on /turtle1/pose.
```

这表示`/turtlesim`节点将数据发表到两个未订阅任何东西的话题，doctor认为会出现问题，如果现在运行`/color_sensor`和`/pose`这些警告就会消失，因为Publisher将有Subscriber运行

```
ros2 topic echo /turtle1/color_sensor
```

和

```
ros2 topic echo /turtle1/pose
```

可以通过`echo`创建临时订阅者来订阅`/pose`和`/color_sensor`，这样就不再是“有发无收”状态，警告会消失
同时可以使用`ros2 doctor --report`来获取一个完整的报告，例如：

```
NETWORK CONFIGURATION
...

PLATFORM INFORMATION
...

RMW MIDDLEWARE
...

ROS 2 INFORMATION
...

TOPIC LIST
...
```

rosdep

URDF

示例包

事前准备

首先安装rosdep，进入工作空间根目录使用

```
sudo rosdep init
```

初始化rosdep，随后

```
rosdep update
```

更新软件包，然后运行

```
rosdep install --from-paths src --ignore-src -y
```

检查所有包的`package.xml`下的`<depend>`，同时只安装系统库`apt`，如果都已完备，会提示`All required rosdeps installed successfully`
否则会自动运行

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

等命令
使用**URDF**需要`joint_state_publisher`包，所以使用rosdep检查是必要的
同时我们需要`urdf_tutorial`这个示例包，可以用

```
ros2 pkg prefix urdf_tutorial
```

来检测有没有这个包，之后可以

```
cd ~/ros2_ws
rosdep update
rosdep install --from-paths src --ignore-src -y
```

用rosdep检测缺少的东西并安装
也可以通过

```
sudo apt install ros-jazzy-urdf-tutorail
```

来安装

一个形状

`urdf_tutorial`内有一个`01-myfirst.urdf`的urdf示例文件，里面是

```
<?xml version="1.0"?>
<robot name="myfirst">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```

这是一个名为`myfirst`的机器人，它只包含一个链接（即部件），其可视组件只是一个长为 0.6 米、半径为 0.2 米的圆柱体
我们可以使用这个包内的万能启动文件`display.launch.py`来查看这个机器人：

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/01-myfirst.urdf
```

这个指令貌似会自动检索相对目录和`urdf_tutorial`包所在的目录（一个没找到就去另一个），所以不在`urdf_tutorial`目录也可以运行
（注意：貌似有些情况下会出现报错，可能是版本不同，可能在22.04下的humble会出现识别不到的情况，这个时候就需要cd到`urdf_tutorial`目录或使用

```
ros2 launch urdf_tutorial display.launch.py model:=`ros2 pkg prefix --share urdf_tutorial
```

来在任意位置运行）

多个形状

查看示例代码`02-multipleshapes.urdf`：

```
<?xml version="1.0"?>
<robot name="multipleshapes">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>

  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
    </visual>
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
  </joint>

</robot>
```

这个代码多定义了一个$0.6 \times 0.1 \times 0.2$的红色长方体
值得注意的是，关节是由父节点和子节点定义的，因此urdf最终是一棵**树状结构**，只有一个根链接，这意味着腿的位置取决于base_link的位置。
运行

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/02-multipleshapes.urdf
```

可以看到rviz中有两个物体
如果不想让它们两个重叠，我们可以定义更多的节点：

```
<?xml version="1.0"?>
<robot name="origins">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>

  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    </visual>
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0 -0.22 0.25"/>
  </joint>

</robot>
```

新增的代码，在`joint`方面，表示**关节**（在这里作为**腿**）（绑定了刚刚增加的红色长方体作为腿）相对于身体的几何中心的转换（位移和旋转）
也就是说腿是一个子坐标系，这行代码表示子坐标系相对于父坐标系的转换（即直接作用于子坐标系），在这里是向y轴负方向移动0.22米，z轴正方向移动0.25米
在`visual`方面，顾名思义就是看起来如何，表示刚刚新增的长方体的几何形状转换（即作用于

子坐标系上的物体），在这里表示自身绕y轴逆时针旋转$\frac{\pi}{2}$后在z轴的负方向移动0.3米

使用

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/03-origins.urdf
```

查看

RGBA通道

RGBA是可更改的，如：

```xml
<?xml version="1.0"?>
<robot name="materials">

  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>

  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>

  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
      <material name="blue"/>
    </visual>
  </link>

  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
      <material name="white"/>
    </visual>
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0 -0.22 0.25"/>
  </joint>

  <link name="left_leg">
    <visual>
      <geometry>
```

```
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
      <material name="white"/>
    </visual>
  </link>

  <joint name="base_to_left_leg" type="fixed">
    <parent link="base_link"/>
    <child link="left_leg"/>
    <origin xyz="0 0.22 0.25"/>
  </joint>

</robot>
```

使用

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/04-materials.urdf
```

运行查看

完成模型

以下是一个较为完整的机器人（？）：

```
<?xml version="1.0"?>
<robot name="visual">

  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>
  <material name="black">
    <color rgba="0 0 0 1"/>
  </material>
  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>

  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
      <material name="blue"/>
    </visual>
  </link>

  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
      <material name="white"/>
    </visual>
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0 -0.22 0.25"/>
  </joint>

  <link name="right_base">
```

```
    <visual>
      <geometry>
        <box size="0.4 0.1 0.1"/>
      </geometry>
      <material name="white"/>
    </visual>
  </link>

  <joint name="right_base_joint" type="fixed">
    <parent link="right_leg"/>
    <child link="right_base"/>
    <origin xyz="0 0 -0.6"/>
  </joint>

  <link name="right_front_wheel">
    <visual>
      <origin rpy="1.57075 0 0" xyz="0 0 0"/>
      <geometry>
        <cylinder length="0.1" radius="0.035"/>
      </geometry>
      <material name="black"/>
    </visual>
  </link>
  <joint name="right_front_wheel_joint" type="fixed">
    <parent link="right_base"/>
    <child link="right_front_wheel"/>
    <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
  </joint>

  <link name="right_back_wheel">
    <visual>
      <origin rpy="1.57075 0 0" xyz="0 0 0"/>
      <geometry>
        <cylinder length="0.1" radius="0.035"/>
      </geometry>
      <material name="black"/>
    </visual>
  </link>
  <joint name="right_back_wheel_joint" type="fixed">
    <parent link="right_base"/>
```

```
  <child link="right_back_wheel"/>
  <origin rpy="0 0 0" xyz="-0.133333333333 0 -0.085"/>
</joint>

<link name="left_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>

<joint name="base_to_left_leg" type="fixed">
  <parent link="base_link"/>
  <child link="left_leg"/>
  <origin xyz="0 0.22 0.25"/>
</joint>

<link name="left_base">
  <visual>
    <geometry>
      <box size="0.4 0.1 0.1"/>
    </geometry>
    <material name="white"/>
  </visual>
</link>

<joint name="left_base_joint" type="fixed">
  <parent link="left_leg"/>
  <child link="left_base"/>
  <origin xyz="0 0 -0.6"/>
</joint>

<link name="left_front_wheel">
  <visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
```

```
    </geometry>
    <material name="black"/>
  </visual>
</link>
<joint name="left_front_wheel_joint" type="fixed">
  <parent link="left_base"/>
  <child link="left_front_wheel"/>
  <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
</joint>

<link name="left_back_wheel">
  <visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>
<joint name="left_back_wheel_joint" type="fixed">
  <parent link="left_base"/>
  <child link="left_back_wheel"/>
  <origin rpy="0 0 0" xyz="-0.133333333333 0 -0.085"/>
</joint>

<joint name="gripper_extension" type="fixed">
  <parent link="base_link"/>
  <child link="gripper_pole"/>
  <origin rpy="0 0 0" xyz="0.19 0 0.2"/>
</joint>

<link name="gripper_pole">
  <visual>
    <geometry>
      <cylinder length="0.2" radius="0.01"/>
    </geometry>
    <origin rpy="0 1.57075 0 " xyz="0.1 0 0"/>
  </visual>
</link>
```

```
  <joint name="left_gripper_joint" type="fixed">
    <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
    <parent link="gripper_pole"/>
    <child link="left_gripper"/>
  </joint>

  <link name="left_gripper">
    <visual>
      <origin rpy="0.0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://urdf_tutorial/meshes/l_finger.dae"/>
      </geometry>
    </visual>
  </link>

  <joint name="left_tip_joint" type="fixed">
    <parent link="left_gripper"/>
    <child link="left_tip"/>
  </joint>

  <link name="left_tip">
    <visual>
      <origin rpy="0.0 0 0" xyz="0.09137 0.00495 0"/>
      <geometry>
        <mesh filename="package://urdf_tutorial/meshes/l_finger_tip.dae"/>
      </geometry>
    </visual>
  </link>
  <joint name="right_gripper_joint" type="fixed">
    <origin rpy="0 0 0" xyz="0.2 -0.01 0"/>
    <parent link="gripper_pole"/>
    <child link="right_gripper"/>
  </joint>

  <link name="right_gripper">
    <visual>
      <origin rpy="-3.1415 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://urdf_tutorial/meshes/l_finger.dae"/>
      </geometry>
```

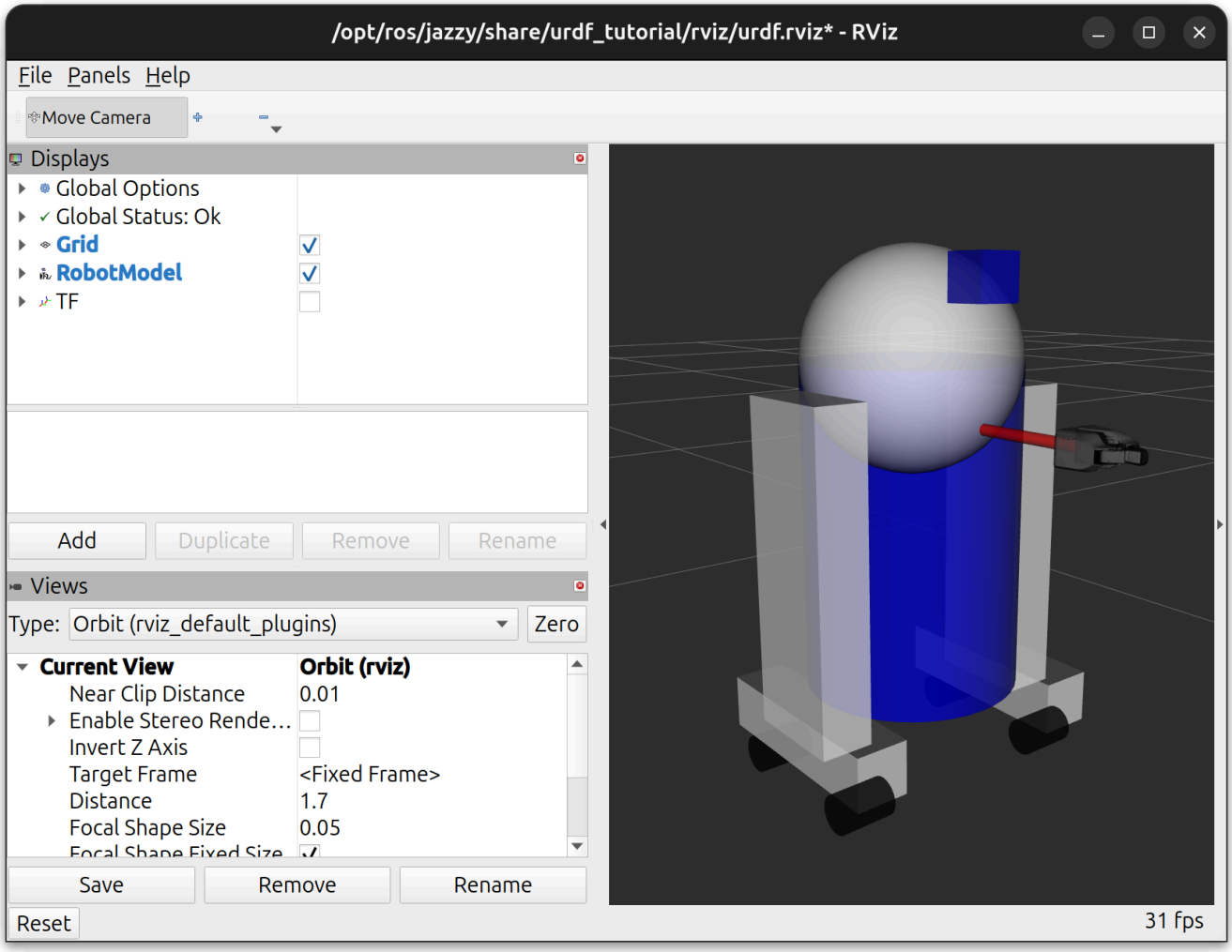
```
  </visual>
</link>

<joint name="right_tip_joint" type="fixed">
  <parent link="right_gripper"/>
  <child link="right_tip"/>
</joint>

<link name="right_tip">
  <visual>
    <origin rpy="-3.1415 0 0" xyz="0.09137 0.00495 0"/>
    <geometry>
      <mesh filename="package://urdf_tutorial/meshes/l_finger_tip.dae"/>
    </geometry>
  </visual>
</link>

<link name="head">
  <visual>
    <geometry>
      <sphere radius="0.2"/>
    </geometry>
    <material name="white"/>
  </visual>
</link>
<joint name="head_swivel" type="fixed">
  <parent link="base_link"/>
  <child link="head"/>
  <origin xyz="0 0 0.3"/>
</joint>

<link name="box">
  <visual>
    <geometry>
      <box size="0.08 0.08 0.08"/>
    </geometry>
    <material name="blue"/>
  </visual>
</link>
```

```
  <joint name="tobox" type="fixed">
    <parent link="head"/>
    <child link="box"/>
    <origin xyz="0.1814 0 0.1414"/>
  </joint>
</robot>
```

使用

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/05-visual.urdf
```

运行查看



建立可移动机器人模型

我们可以先运行

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/06-flexible.urdf
```

这将会额外启动一个gui界面，它可以通过滑条控制各个关节
（这里的原理是，滑动后将值发布到对应话题，随后由`robot_state_publisher`计算不同部分之间的所有变换，然后用生成的变换树显示在rviz中）
一个可移动机器人会包含以下关节：

- **头**
  身体和头部之间的连接是一个连续的关节，这意味着它可以从负无穷大到正无穷大的任何角度，车轮也是这样的，因此它们可以永远向两个方向滚动

  ```
  <joint name="head_swivel" type="continuous">
    <parent link="base_link"/>
    <child link="head"/>
    <axis xyz="0 0 1"/>
    <origin xyz="0 0 0.3"/>
  </joint>
  ```

- **抓爪**
  左右夹持器接头都被称为旋转接头，这意味着它们以与连续关节相同的方式旋转，但是它们有严格的限制
  因此，我们必须包括**限制标签**，指定关节的上限和下限 (以弧度为单位)
  除此之外，我们还必须为这个关节指定一个最大速度和力矩

  ```
  <joint name="left_gripper_joint" type="revolute">
    <axis xyz="0 0 1"/>
    <limit effort="1000.0" lower="0.0" upper="0.548" velocity="0.5"/>
    <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
    <parent link="gripper_pole"/>
    <child link="left_gripper"/>
  </joint>
  ```

- **抓臂**
  夹持器臂是一种不同的接头，即棱柱形接头，这意味着它沿着轴移动，而不是围绕它
  这种平移运动使得机器人模型能够延伸和缩回它的抓臂
  棱柱形臂的极限以与旋转接头相同的方式规定，单位是米

```
<joint name="gripper_extension" type="prismatic">
  <parent link="base_link"/>
  <child link="gripper_pole"/>
  <limit effort="1000.0" lower="-0.38" upper="0" velocity="0.5"/>
  <origin rpy="0 0 0" xyz="0.19 0 0.2"/>
</joint>
```

还有其他两种关节，以后再讨论
我们前几节都在讨论`visual`元素，即视觉上，接下来我们还需要定义`collision`元素，用于管理机器人的碰撞，他和`visual`几乎一模一样，都有`<origin>`来定义位姿，`<geometry>`来定义形状，它们的等级也是一样的，是链接对象的**直接子元素**
所以一个基本链接代码应该是这样的

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
      <color rgba="0 0 .8 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
</link>
```

在定义这个元素时，最好遵循这两个建议

- 简单的几何体
  检测两个简单的几何体碰撞比精细几何体检测更快，更简单，所以最好将碰撞模型改为比较符合的简单几何体
- 安全区域
  为了避免与敏感区域碰撞（例如头部），可以将几何体定义为一个包围它头部的圆柱体

接下来需要定义物理属性

- 惯性

  每个元素都需要惯性标签`<inertia>`例：

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
      <color rgba="0 0 .8 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="1e-3" ixy="0.0" ixz="0.0" iyy="1e-3" iyz="0.0" izz="1e-3"/>
  </inertial>
</link>
```

  单位是kg，它是一个3×3旋转惯性矩阵，由惯性元素指定且为对角矩阵，所以用六个元素表示即可

$$\begin{bmatrix} i_{xx} & i_{xy} & i_{xz} \\ i_{xy} & i_{yy} & i_{yz} \\ i_{xz} & i_{yz} & i_{zz} \end{bmatrix}$$

  除此之外还可以指定一个原点标签，以指定重心和惯性参考系（相对于链接的参考系）

- 接触系数

  可以通过在标签`<contact_coefficients>`定义：

  - mu 摩擦系数
  - kp 刚度系数
  - kd 阻尼系数

- 关节属性

  关节的运动由关节动力学标签来指定

- 摩擦 - 物理静摩擦力
  对于平移关节，单位为N，对于旋转关节，单位为$\mathrm{N}\cdot\mathrm{m}$
- 阻尼 - 物理阻尼值
  对于平移关节，单位为$\mathrm{N}\cdot\mathrm{s}/\mathrm{m}$，对于旋转关节，单位为$\mathrm{N}\cdot\mathrm{s}\cdot\mathrm{m}/\mathrm{rad}$

未指定情况下一般为0

其他标签以后再讨论

Xacro

使用步骤

xacro是一种用于xml的宏语言，使用它有以下几个步骤
首先将`.urdf`改为`.xacro`，再在根标签`<robot>`中声明xacro的命名空间，让系统认识xacro开头的标签：

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="my_robot">
...
</robot>
```

功能

- 常量替换
  使用

  ```
  <xacro:property name = "wheel_radius" value="0.05" />
  ```

  这样我们就可以用`${wheel_radius}`替换数值0.05了，例如：

  ```
  <xacro:property name="PI" value="3.1415926535897931" />
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder radius="${wheel_radius}" length="${wheel_radius * 2}" />
      </geometry>
    </visual>
  </link>
  ```

这些常量替换在`${}`的`{}`内可以使用数学符号进行数学运算

- 宏
  也就是可以写一个类似函数的东西，随时调用
  们可以写一个这样的宏：

```
<xacro:macro name="wheel_func" params="wheel_name reflect">
  <link name="${wheel_name}_link">
    <visual>
      <origin xyz="0 0 0" rpy="${PI/2} 0 0"/>
      <geometry>
        <cylinder radius="${wheel_radius}" length="0.04"/>
      </geometry>
    </visual>
  </link>

  <joint name="${wheel_name}_joint" type="continuous">
    <parent link="base_link"/>
    <child link="${wheel_name}_link"/>
    <origin xyz="0.1 ${reflect * 0.15} 0" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
  </joint>
</xacro:macro>
```

  调用的时候直接

```
<xacro:wheel_func wheel_name="left"  reflect="1" />
<xacro:wheel_func wheel_name="right" reflect="-1" />
```

  就行了，这样编译器就会先把上面的宏填入对应的参数（1和-1）后再整体代码复制替换到对应行上

- 合并文件
  我们可以将不同的元件写到不同的`.xacro`上再通过总文件合并，例如：

```
<xacro:include filename="lidar.xacro" />
<xacro:include filename="arm.xacro" />
```

搞定一切后，我们最后需要的还是`.urdf`文件，可以通过

```
xacro my_robot.xacro > my_robot.urdf
```

来查看生成的urdf文件对不对，也是一个简单的手动转换方法
而在实际可以使用效率更高的自动转换，即在launch文件中写

```
import xacro
from launch_ros.descriptions import ParameterValue

# ... 在生成描述时
robot_description_config = xacro.process_file(xacro_file_path)
robot_description = {'robot_description': robot_description_config.toxml()}
```

类似这样的代码即可

`robot_state_publisher`

使用`robot_state_publisher`能将机器人的状态发布到`/tf2`，我们先创建一个工作空间`second_ros2_ws`，然后进入其下的`src`文件夹，使用

```
ros2 pkg create urdf_tutorial_r2d2 --build-type ament_python --dependencies rclpy --licens
```

创建一个功能包`urdf_tutorial_r2d2`并进入这个文件夹内
创建一个文件夹`urdf`并进入
随后我们在这个文件夹下创建一个`r2d2.urdf.xml`文件，写一个精简的urdf代码：

```
<?xml version="1.0"?>
<robot name="r2d2">

  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>
  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>

  <link name="axis">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
      <material name="blue"/>
    </visual>
  </link>

  <link name="head">
    <visual>
      <geometry>
        <sphere radius="0.2"/>
      </geometry>
      <material name="white"/>
    </visual>
  </link>

  <joint name="swivel" type="continuous">
    <parent link="axis"/>
    <child link="head"/>
    <axis xyz="0 0 1"/>
    <origin xyz="0 0 0.3"/>
  </joint>

  <link name="tilt_link">
    <visual>
      <geometry>
        <box size="0.05 0.1 0.05"/>
```

```
      </geometry>
      <material name="blue"/>
    </visual>
  </link>

  <joint name="tilt" type="revolute">
    <parent link="head"/>
    <child link="tilt_link"/>
    <origin xyz="0.15 0 0.1" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
    <limit lower="-0.5" upper="0.5" effort="10" velocity="1.0"/>
  </joint>

  <link name="periscope">
    <visual>
      <geometry>
        <cylinder length="0.2" radius="0.02"/>
      </geometry>
      <material name="white"/>
    </visual>
  </link>

  <joint name="periscope" type="prismatic">
    <parent link="head"/>
    <child link="periscope"/>
    <axis xyz="0 0 1"/>
    <origin xyz="0 0 0.2"/>
    <limit lower="0" upper="0.2" effort="10" velocity="1.0"/>
  </joint>

</robot>
```

这个代码不要随便写，必须要与后续的`state_publisher.py`对应
然后实际上还需要一个与`urdf`文件夹同级的`rviz`文件夹存rviz2的配置文件，但实际上对于初学者不需要，先跳过
接下来使用Python编写ros2节点（因为教程没有cpp代码），进入`~/second_ros2_ws/src/urdf_tutorial_r2d2/urdf_tutorial_r2d2`目录并创建一个`state_publisher.py`文件，填入

```
from math import sin, cos, pi
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile
from geometry_msgs.msg import Quaternion
from sensor_msgs.msg import JointState
from tf2_ros import TransformBroadcaster, TransformStamped

class StatePublisher(Node):

    def __init__(self):
        rclpy.init()
        super().__init__('state_publisher')

        qos_profile = QoSProfile(depth=10)
        self.joint_pub = self.create_publisher(JointState, 'joint_states', qos_profile)
        self.broadcaster = TransformBroadcaster(self, qos=qos_profile)
        self.nodeName = self.get_name()
        self.get_logger().info("{0} started".format(self.nodeName))

        degree = pi / 180.0
        loop_rate = self.create_rate(30)

        # robot state
        tilt = 0.
        tinc = degree
        swivel = 0.
        angle = 0.
        height = 0.
        hinc = 0.005

        # message declarations
        odom_trans = TransformStamped()
        odom_trans.header.frame_id = 'odom'
        odom_trans.child_frame_id = 'axis'
        joint_state = JointState()

        try:
            while rclpy.ok():
```

```
                rclpy.spin_once(self)

                # update joint_state
                now = self.get_clock().now()
                joint_state.header.stamp = now.to_msg()
                joint_state.name = ['swivel', 'tilt', 'periscope']
                joint_state.position = [swivel, tilt, height]

                # update transform
                # (moving in a circle with radius=2)
                odom_trans.header.stamp = now.to_msg()
                odom_trans.transform.translation.x = cos(angle)*2
                odom_trans.transform.translation.y = sin(angle)*2
                odom_trans.transform.translation.z = 0.7
                odom_trans.transform.rotation = \
                    euler_to_quaternion(0, 0, angle + pi/2) # roll,pitch,yaw

                # send the joint state and transform
                self.joint_pub.publish(joint_state)
                self.broadcaster.sendTransform(odom_trans)

                # Create new robot state
                tilt += tinc
                if tilt < -0.5 or tilt > 0.0:
                    tinc *= -1
                height += hinc
                if height > 0.2 or height < 0.0:
                    hinc *= -1
                swivel += degree
                angle += degree/4

                # This will adjust as needed per iteration
                loop_rate.sleep()

        except KeyboardInterrupt:
            pass

def euler_to_quaternion(roll, pitch, yaw):
    qx = sin(roll/2) * cos(pitch/2) * cos(yaw/2) - cos(roll/2) * sin(pitch/2) * sin(yaw/2
    qy = cos(roll/2) * sin(pitch/2) * cos(yaw/2) + sin(roll/2) * cos(pitch/2) * sin(yaw/2
```

```
    qz = cos(roll/2) * cos(pitch/2) * sin(yaw/2) - sin(roll/2) * sin(pitch/2) * cos(yaw/2
    qw = cos(roll/2) * cos(pitch/2) * cos(yaw/2) + sin(roll/2) * sin(pitch/2) * sin(yaw/2
    return Quaternion(x=qx, y=qy, z=qz, w=qw)

def main():
    node = StatePublisher()

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

这是一个发布状态的代码，随后编写一个launch文件，在`src/urdf_tutorial_r2d2`目录下创建一个`launch`文件夹创建一个文件`demo.launch.py`并填入

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():

    use_sim_time = LaunchConfiguration('use_sim_time', default='false')

    urdf_file_name = 'r2d2.urdf.xml'
    urdf = os.path.join(
        get_package_share_directory('urdf_tutorial_r2d2'),
        urdf_file_name)
    with open(urdf, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use simulation (Gazebo) clock if true'),
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[{'use_sim_time': use_sim_time, 'robot_description': robot_desc}],
            arguments=[urdf]),
        Node(
            package='urdf_tutorial_r2d2',
            executable='state_publisher',
            name='state_publisher',
            output='screen'),
    ])
```

随后退出该文件夹，编写`setup.py`文件，它最后应该是这样的

```
import os
from glob import glob
from setuptools import setup
from setuptools import find_packages

package_name = 'urdf_tutorial_r2d2'

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']),
        (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch', '*laun
        (os.path.join('share', package_name), glob('urdf/*')),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='goose',
    maintainer_email='meis38@126.com',
    description='TODO: Package description',
    license='Apache-2.0',
    extras_require={
        'test': [
            'pytest',
        ],
    },
    entry_points={
        'console_scripts': [
            'state_publisher = urdf_tutorial_r2d2.state_publisher:main'
        ],
    },
)
```

随后退到工作空间使用

```
colcon build --symlink-install --packages-select urdf_tutorial_r2d2
```

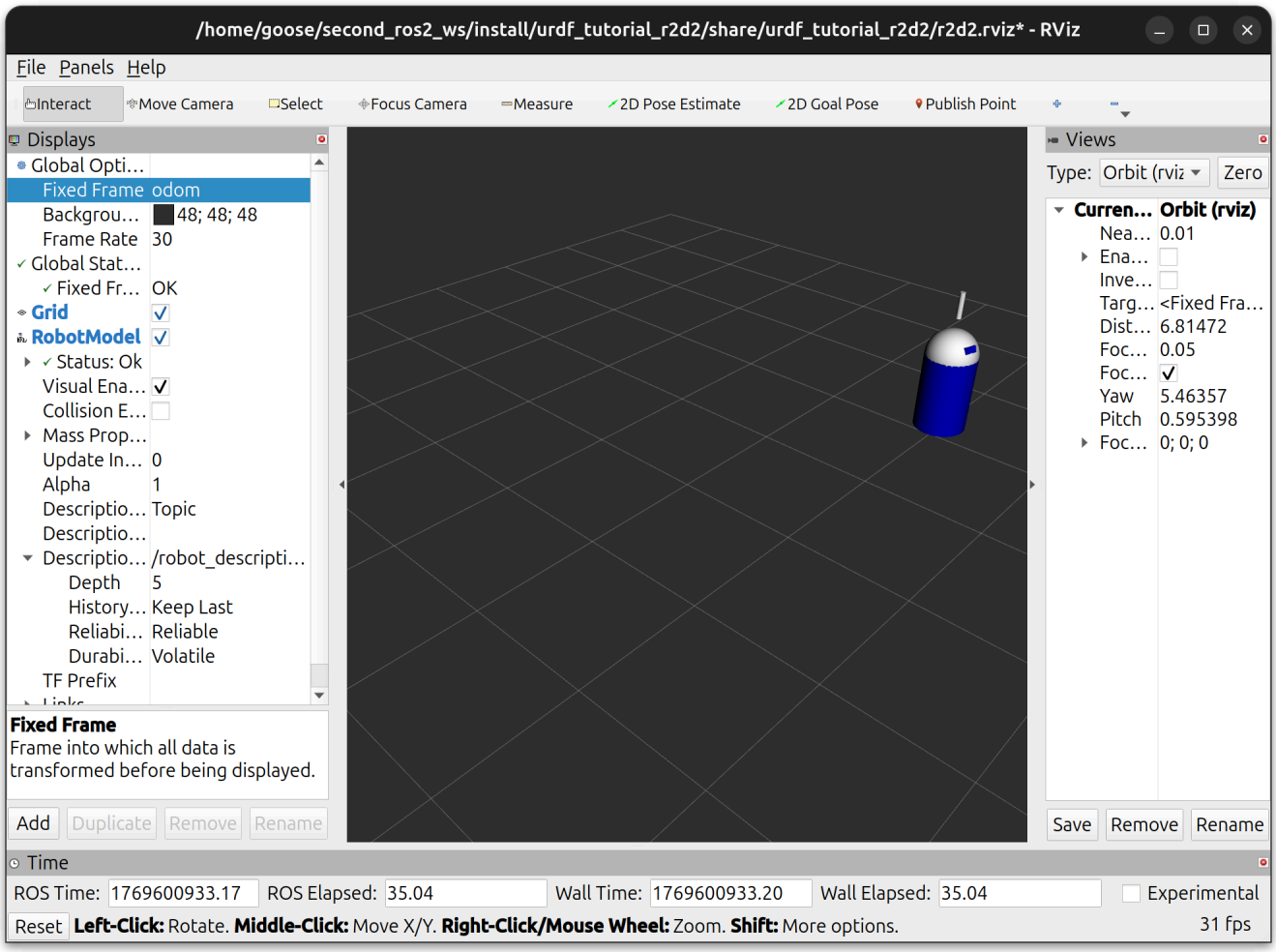
软链接编译包，刷新环境后运行

```
ros2 launch urdf_tutorial_r2d2 demo.launch.py
```

启动包，然后打开一个rviz

```
rviz2 -d ~/second_ros2_ws/install/urdf_tutorial_r2d2/share/urdf_tutorial_r2d2/r2d2.rviz
```

由于我们没有设置`.rviz`配置文件，所以我们需要手动用左下角的Add添加`RobotModel`，并在`Description Topic`选择`/robot_description`，我们在`Fixed Frame`选择`odom`，就能看到一个机器人（？）绕圆周运动了，并且头还会伸缩
大概会看到这样的画面：



终端

缓冲区

以下是一个控制海龟移动的按键控制节点核心代码：

```
void EController::key_loop() {

    RCLCPP_INFO(this->get_logger(), "\n[Info] 已启动敌人控制器");
    while (rclcpp::ok()) {

        char key;
        std::cin >> key;
        geometry_msgs::msg::Twist vec;
        vec.linear.x = 0;
        vec.angular.z = 0;
        if (key == 'W' || key == 'w') vec.linear.x = 2;
        if (key == 'S' || key == 's') vec.linear.x = -2;
        if (key == 'A' || key == 'a') vec.angular.z = M_PI / 6;
        if (key == 'D' || key == 'd') vec.angular.z = -M_PI / 6;
        if (key == 'q' || key == 'Q') break;
        vecPub->publish(vec);
    }
    rclcpp::shutdown();
}
```

这个方法实现的很简单易懂，但是有个缺点就是每次输入后**需要回车**才能响应输入内容，这本质是因为终端使用**行缓冲模式**，以下是几个不同**缓冲区**的介绍：

名称	表示	
全缓冲	在这种情况下，当填满标准I/O缓存后才进行实际I/O操作	全缓冲的典型代表是对磁盘
行缓冲	在这种情况下，当在输入和输出中遇到换行符时，执行真正的I/O操作	我们输入的字符先存放在缓 O操作。典型代表是标准输
无缓冲	也就是不进行缓冲，可以直接进行I/O操作	标准出错情况`std::cerr`是

我们可以将终端设置为Raw模式，即终端每次只收集一个字符就返回，不进行缓冲（禁用**行缓冲**和**回显**，回显是避免重复显示信息），同时为了实现持续响应按住按键不止判断一次的效果，

我们还需要设置**非阻塞模式**，即我们读取字符需要一个`read()`函数，这个函数如果在阻塞模式下读不到输入会一直卡住，如果在非阻塞模式下没读到数据则会持续返回0，这个可以用于实现松手即停以及持续移动的效果

我们需要导入`termios.h`、`unistd.h`、`fcntl.h`这三个头文件，分别用于终端控制、系统终端相关操作函数、设置文件描述符标志

我们在`key_loop()`函数开头输入

```
struct termios old_t, new_t;
// 保存终端旧设置
tcgetattr(STDIN_FILENO, &old_t);
new_t = old_t;
// 禁用 canonical（行缓冲）和 echo（回显）
new_t.c_lflag &= ~(ICANON | ECHO);
new_t.c_cc[VMIN] = 0;// 非阻塞模式相关
new_t.c_cc[VTIME] = 0;
// 设置终端新模式
tcsetattr(STDIN_FILENO, TCSANOW, &new_t);

int flags = fcntl(STDIN_FILENO, F_GETFL, 0); // 获取当前标志
fcntl(STDIN_FILENO, F_SETFL, flags | O_NONBLOCK);// 设置 std::cin 非阻塞
```

效果如注释所示，最后我们的核心代码部分为：

```
while (rclcpp::ok()) {

    char key = 0;
    std::cin >> key;
    geometry_msgs::msg::Twist vec;
    vec.linear.x = 0;
    vec.angular.z = 0;
    if (read(STDIN_FILENO, &key, 1) == 1) { // 有键按下，并赋予key

        if (key == 'W' || key == 'w') vec.linear.x = 2;
        if (key == 'S' || key == 's') vec.linear.x = -2;
        if (key == 'A' || key == 'a') vec.angular.z = M_PI / 6;
        if (key == 'D' || key == 'd') vec.angular.z = -M_PI / 6;
        if (key == 'q' || key == 'Q') break;
    }
    vecPub->publish(vec);

    // 线程暂停20ms，控制发布频率
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
}
// 恢复终端设置
tcsetattr(STDIN_FILENO, TCSANOW, &old_t);
rclcpp::shutdown();
```

效果如注释所示，其中线程暂停的主要目的还有防止cpu高速空转占用大量cpu资源，另外注意，我们需要持续使用`vecPub`发布信息，所以绑定它的回调函数的时候队列不宜超过5，否则会导致大量信息堆积，从而产生响应超出和滞后的现象，一般设置为3左右最为合适