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：

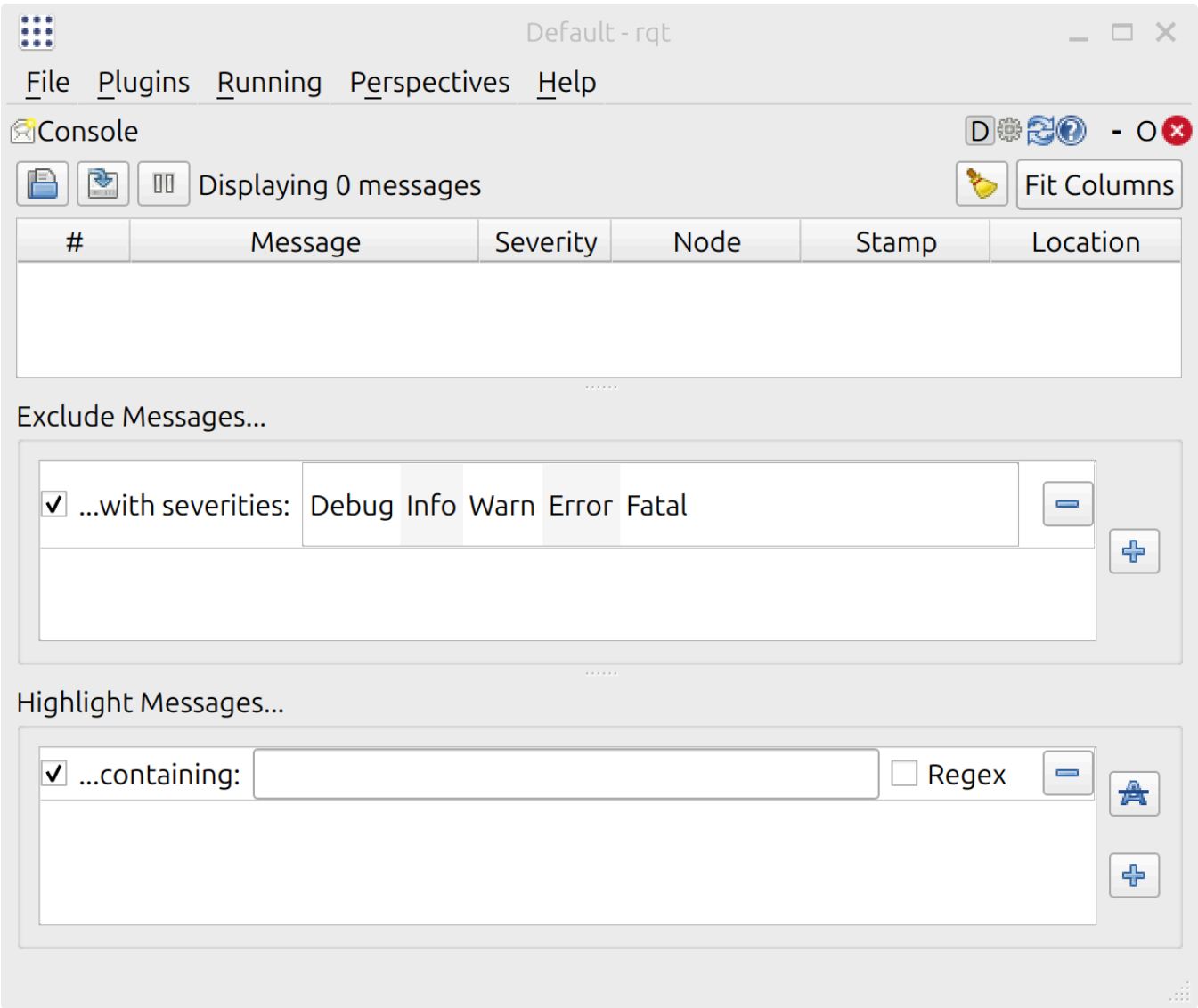
```
ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=turtle2/cmd_vel
```

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
```

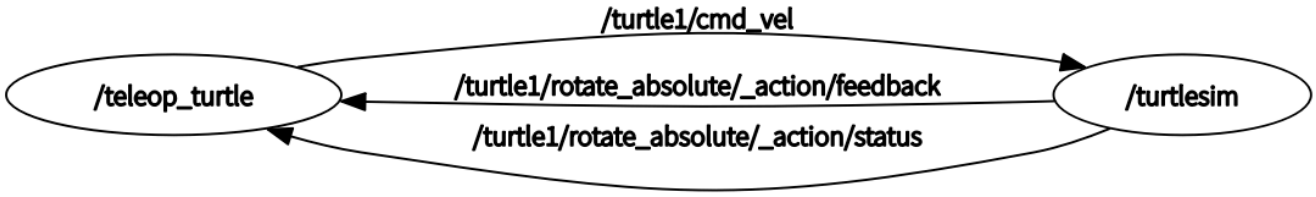
访问节点`my_turtle`的信息（这条命令原型为`ros2 node info <node_name>`）

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

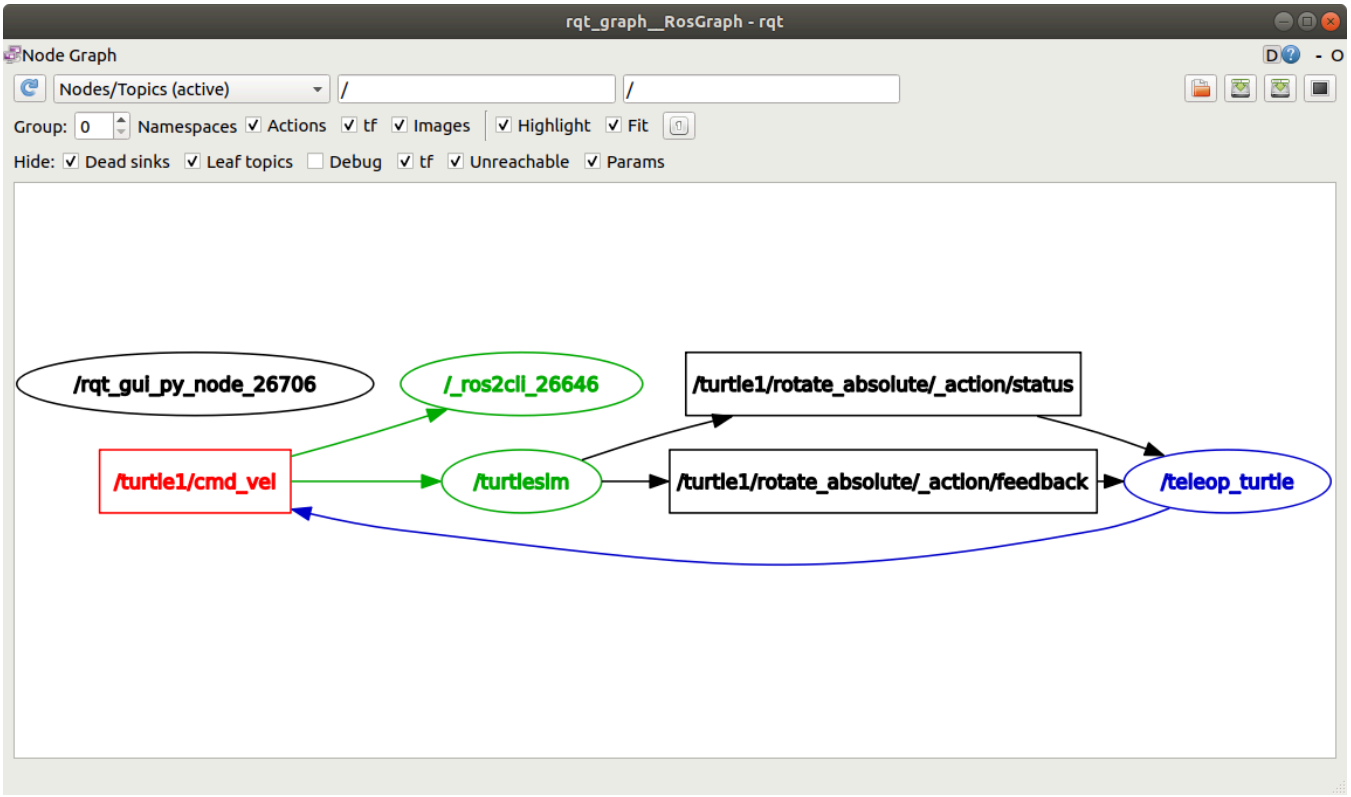
```
ros2 topic echo <topic_name>
```

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

某些地方称Pub为酒吧，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`

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
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Subscriber和Publisher

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"
..
```