机器人操作系统与编程

机 电 工 程 学 院

机器人智能制造研究团队

2025 · 秋

课程安排

第 二 章

ROS通信机制

目录

CONTENTS

01

第一节
ROS通信机制简介

一、ROS通信架构

ROS是进程（也称为Nodes）的分布式框架

机器人是一种高度复杂的系统性实现，在机器人上可能集成各种传感器(雷达、摄像头、GPS...)以及运动控制实现，为了解耦合，在ROS中每一个功能点都是一个单独的进程，每一个进程都是独立运行的。更确切的讲，ROS是进程（也称为Nodes）的分布式框架。因为这些进程甚至还可分布于不同主机，不同主机协同工作，从而分散计算压力。

不过随之也有一个问题: 不同的进程是如何通信的？也即不同进程间如何实现数据交换的？

ROS的通信架构是ROS的灵魂，也是整个ROS正常运行的关键所在。

ROS通信架构包括各种数据的处理，进程的运行，消息的传递等等。

二、master&node

Node

在ROS的世界里，最小的进程单元就是节点（node）。一个软件包里可以有多个可执行文件，可执行文件在运行之后就成了一个进程(process)，这个进程在ROS中就叫做节点。

- 从程序角度来说，node就是一个可执行文件（通常为C++编译生成的可执行文件、Python脚本）被执行，加载到了内存之中；从功能角度来说，通常一个node负责机器人的某一个单独的功能。由于机器人的功能模块非常复杂，我们往往不会把所有功能都集中到一个node上，而会采用分布式的方式，把鸡蛋放到不同的篮子里。
- 例如有一个node来控制底盘轮子的运动，有一个node驱动摄像头获取图像，有一个node驱动激光雷达，有一个node根据传感器信息进行路径规划……这样做可以降低程序发生崩溃的可能性，试想一下如果把所有功能都写到一个程序中，模块间的通信、异常处理将会很麻烦。

二、master&node

Master

由于机器人的元器件很多，功能庞大，因此实际运行时往往会运行众多的node，负责感知世界、控制运动、决策和计算等功能。

那么如何合理的进行调配、管理这些node?

这就要利用ROS提供给我们的节点管理器master, master在整个网络通信架构里相当于管理中心，管理着各个node。node首先在master处进行注册，之后master会将该node纳入整个ROS程序中。node之间的通信也是先由master进行“牵线”，才能两两的进行点对点通信。当ROS程序启动时，第一步首先启动master，由节点管理器处理依次启动node。

二、master&node

启动master和node

当我们要启动ROS时，首先输入命令:

```
$ roscore
```

此时ROS master启动，同时启动的还有rosout和parameter server,其中rosout是负责日志输出的一个节点，其作用是告知用户当前系统的状态，包括输出系统的error、warning等等，并且将log记录于日志文件中，parameter server即是参数服务器，它并不是一个node，而是存储参数配置的一个服务器，后文我们会单独介绍。每一次我们运行ROS的节点前，都需要把master启动起来，这样才能够让节点启动和注册。

启动master之后，节点管理器就开始按照系统的安排协调进行启动具体的节点。节点就是一个进程，只不过在ROS中它被赋予了专用的名字——node。
在下一章我们会介绍ROS的文件系统，有一个package中存放着可执行文件，可执行文件是静态的，当系统执行这些可执行文件，将这些文件加载到内存中，它就成为了动态的node。
具体启动node的语句是:

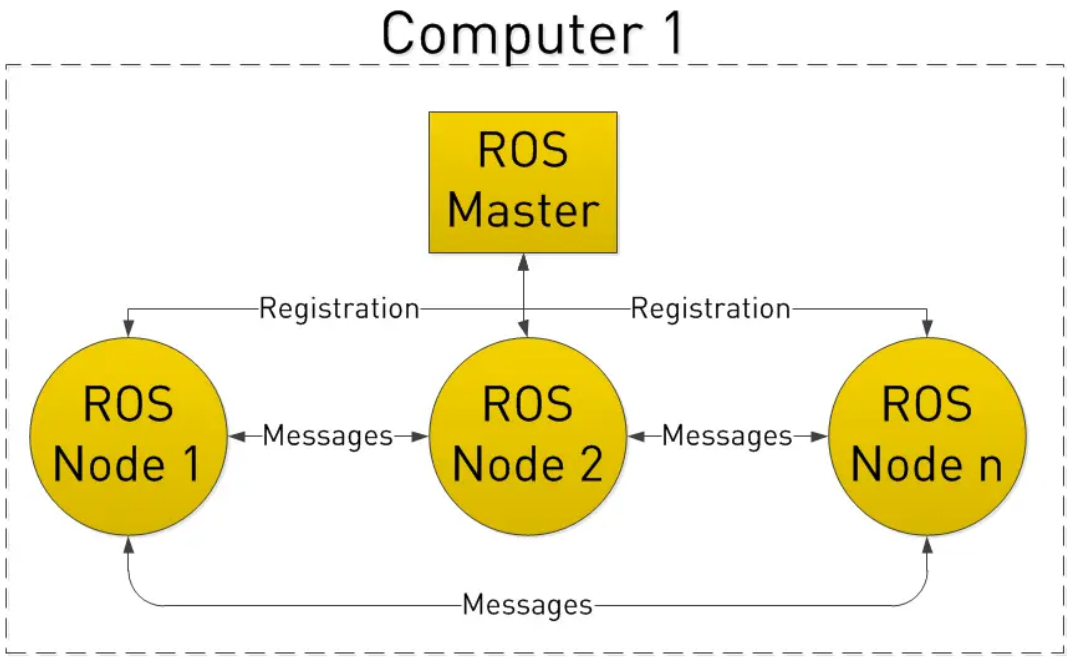
```
$ rosrun pkg_name node_name
```

二、master&node

启动master和node

通常我们运行ROS，就是按照这样的顺序启动，有时候节点太多，我们会选择用launch文件来启动，下一章会有介绍。

Master、Node之间以及Node之间的关系如下图所示：



三、常用命令

rosrun

rosrun命令的详细用法如下：

```
$ rosrun [--prefix cmd] [--debug] pkg_name node_name [ARGS]
```

rosrun将会寻找PACKAGE下的名为EXECUTABLE的可执行程序，rosrun允许在任意包中运行可执行文件，不必先进入（cd）文件夹。

用法：

```
$ rosrun package executable
```

例：

```
$ rosrun roscpp_tutorials talker
```

三、常用命令

rosnode

命令	功能
rosnode ping	测试到节点的连接状态
rosnode list	列出活动节点
rosnode info	打印节点信息
rosnode machine	列出指定设备上节点
rosnode kill	杀死某个节点
rosnode cleanup	清除不可连接的节点

清除无用节点，启动某节点，ctrl + c 关闭后，该节点并没被彻底清除，可以使用 cleanup 清除节点。

三、常用命令

rostopic

rostopic包含rostopic命令行工具，用于显示有关ROS 主题的调试信息，包括发布者，订阅者，发布频率和ROS消息。它还包含一个实验性Python库，用于动态获取有关主题的信息并与之交互。

$ rostopic bw	显示话题使用的带宽
$ rostopic delay	显示带有 header 的话题延迟
$ rostopic echo	打印消息到屏幕，获取指定话题当前发布的消息
$ rostopic find	根据消息类型查找话题
$ rostopic hz	显示话题的发布频率
$ rostopic info	显示话题相关信息：消息类型、发布者信息、订阅者信息
$ rostopic list	显示所有活动状态下的话题
$ rostopic pub	将数据发布到话题
$ rostopic type	列出话题的消息类型

三、常用命令

rostopic

- rostopic list(-v)

直接调用即可，控制台将打印当前运行状态下的主题名称

rostopic list -v：获取话题详情(比如列出： 发布者和订阅者个数...)

- rostopic pub

可以直接调用命令向订阅者发布消息

为roboware 自动生成的 发布/订阅 模型案例中的 订阅者发布一条字符串。

$ rostopic pub /主题名称 消息类型 消息内容

$ rostopic pub /chatter std_msgs gagaxixi

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist
 "linear:
  x: 1.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0"
//只发布一次运动信息

rostopic pub -r 10 /turtle1/cmd_vel geometry_msgs/Twist
 "linear:
  x: 1.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0"
// 以 10HZ 的频率循环发送运动信息
```

三、常用命令

rosmsg

rosmsg是用于显示有关 ROS消息类型的 信息的命令行工具。

$ rosmsg show	显示消息描述
$ rosmsg info	显示消息相关信息
$ rosmsg list	列出当前ROS中的所有消息
$ rosmsg md5	显示 md5 加密后的消息
$ rosmsg package	显示某个功能包下的所有消息
$ rosmsg packages	列出包含消息的功能包

```
//rosmsg package 包名
rosmsg package turtlesim
```

```
//rosmsg show 消息名称
rosmsg show turtlesim/Pose
结果:
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity
```

三、常用命令

rosservice

- 为小乌龟案例生成一只新的乌龟

```
rosservice call /spawn "x: 1.0
y: 2.0
theta: 0.0
name: 'xxx'"
name: "xxx"
```

```
rosservice args /spawn
x y theta name
```

rosservice包含用于列出和查询ROS Services的命令行工具。

➢ 调用部分服务时，如果对相关工作空间没有配置 path，需要进入工作空间调用 source ./devel/setup.bash

$ rosservice args	打印服务参数
$ rosservice call	使用提供的参数调用服务
$ rosservice find	按照服务类型查找服务
$ rosservice info	打印有关服务的信息
$ rosservice list	列出所有活动的服务
$ rosservice type	打印服务类型

```
zilu@zilu-virtual-machine:~$ rosservice list
/eff_joint_traj_controller/gains/elbow_joint/set_parameters
/eff_joint_traj_controller/gains/shoulder_lift_joint/set_parameters
/eff_joint_traj_controller/gains/shoulder_pan_joint/set_parameters
/eff_joint_traj_controller/gains/wrist_1_joint/set_parameters
/eff_joint_traj_controller/gains/wrist_2_joint/set_parameters
/eff_joint_traj_controller/gains/wrist_3_joint/set_parameters
/eff_joint_traj_controller/query_state
/gazebo/apply_body_wrench
/gazebo/apply_joint_effort
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
/gazebo/delete_light
/gazebo/delete_model
/gazebo/get_joint_properties
/gazebo/get_light_properties
/gazebo/get_link_properties
/gazebo/get_link_state
/gazebo/get_loggers
/gazebo/get_model_properties
/gazebo/get_model_state
/gazebo/get_physics_properties
/gazebo/get_world_properties
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_light_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_logger_level
/gazebo/set_model_configuration
/gazebo/set_model_state
```

三、常用命令

rossrv

rossrv是用于显示有关ROS服务类型的信息的命令行工具，与 rosmsg 使用语法高度雷同。

$ rossrv show	显示服务消息描述
$ rossrv info	显示服务消息相关信息
$ rossrv list	列出当前ROS中的所有服务消息
$ rossrv md5	显示 md5 加密后的服务消息
$ rossrv package	显示某个功能包下的所有服务消息
$ rossrv packages	列出包含服务消息的功能包

```
//rossrv package 包名
rossrv package turtlesim
```

```
//rossrv show 消息名称
rossrv show turtlesim/Spawn
结果:
float32 x
float32 y
float32 theta
string name
---
string name
```

三、常用命令

rosparam

rosparam包含rosparam命令行工具，用于使用YAML编码文件在参数服务器上获取和设置ROS参数。

$ rosparam set	设置参数
$ rosparam get	获取参数
$ rosparam load	从外部文件加载参数
$ rosparam dump	将参数写出到外部文件
$ rosparam delete	删除参数
$ rosparam list	列出所有参数

```
rosparam load xxx.yaml
```

```
rosparam dump yyy.yaml
```

```
rosparam set name huluwa

//再次调用 rosparam list 结果
/name
/rosdistro
/roslaunch/uris/host_helloros
/rosversion
/run_id
```

```
rosparam get name

//结果
huluwa
```

```
rosparam delete name

//结果
//去除了name
```

五、三种基本通信方式

ROS通信方式

ROS 中的基本通信机制主要有如下三种实现策略:

- Topic（话题）通信：发布-订阅模式
- Service（服务）通信：请求-响应模式
- Parameter（参数）服务器：参数共享模式

02

第二节
ROS话题通信

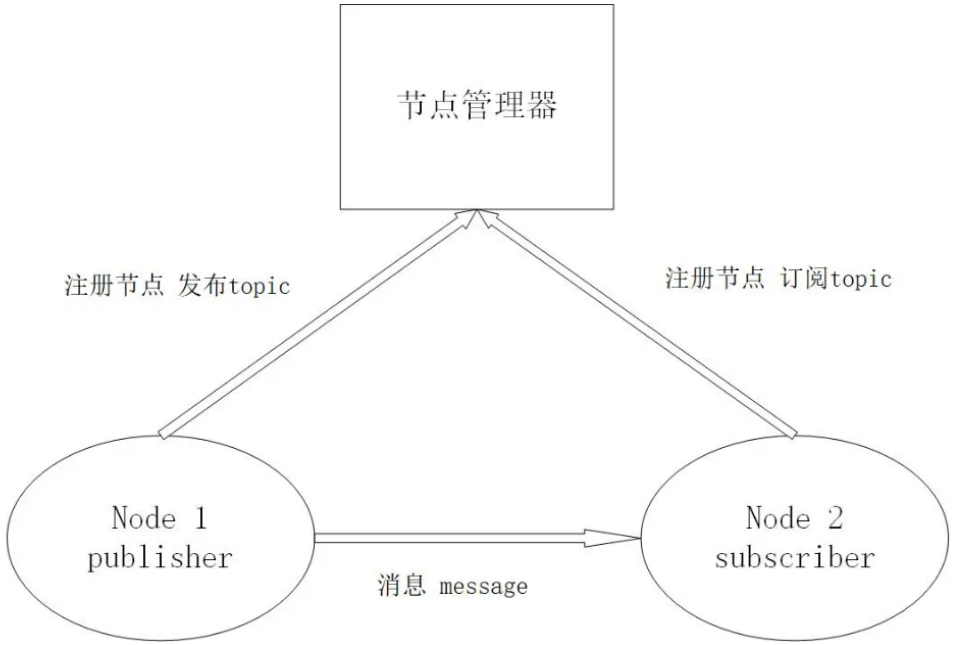
一、ROS话题通信简介

topic话题通信概念

话题通信是ROS中使用频率最高的一种通信模式，话题通信基于发布/订阅的异步通信模式，也即:一个节点发布消息，另一个节点订阅该消息。

对于实时性、周期性的消息，使用topic来传输是最佳的选择。topic是一种点对点的单向通信方式，这里的“点”指的是node，也就是说node之间可以通过topic方式来传递信息。

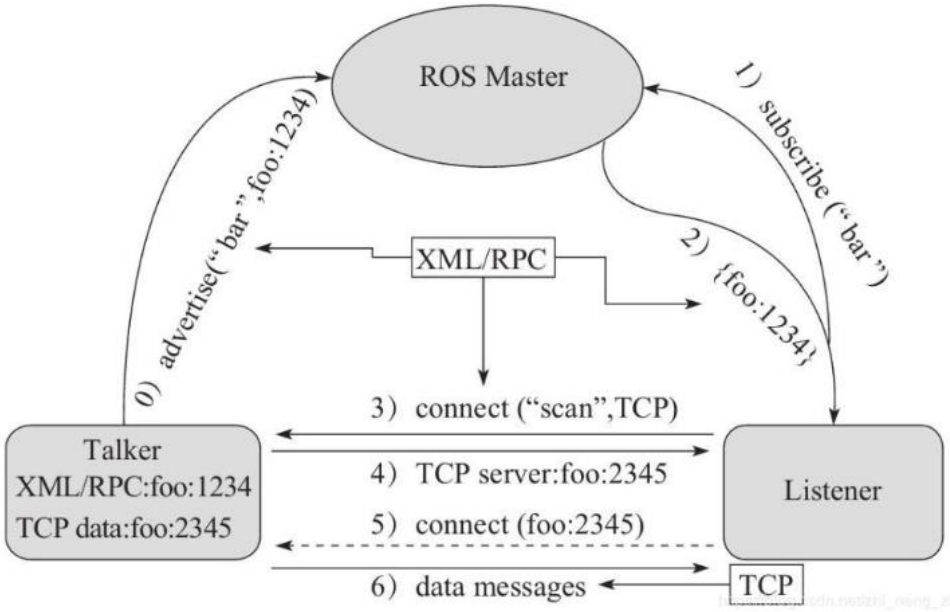
其结构示意图如下:



第二节 ROS话题通信

一、ROS话题通信简介

topic话题通信理论模型



- ROS Master (管理者)
- Talker (发布者)
- Listener (订阅者)

RPC 指的是 Remote Procedure Call（远程过程调用）。它是一种允许计算机程序通过网络向远程服务器或另一个进程请求执行函数或方法的机制，类似于本地调用函数的方式，但实际在远程执行。RPC 提供了一种方便的机制，使得分布式系统中的不同进程能够相互调用和通信。

0.Talker注册

Talker启动后，会通过RPC在 ROS Master 中注册自身信息，其中包含所发布消息的话题名称。ROS Master 会将节点的注册信息加入到注册表中。

1.Listener注册

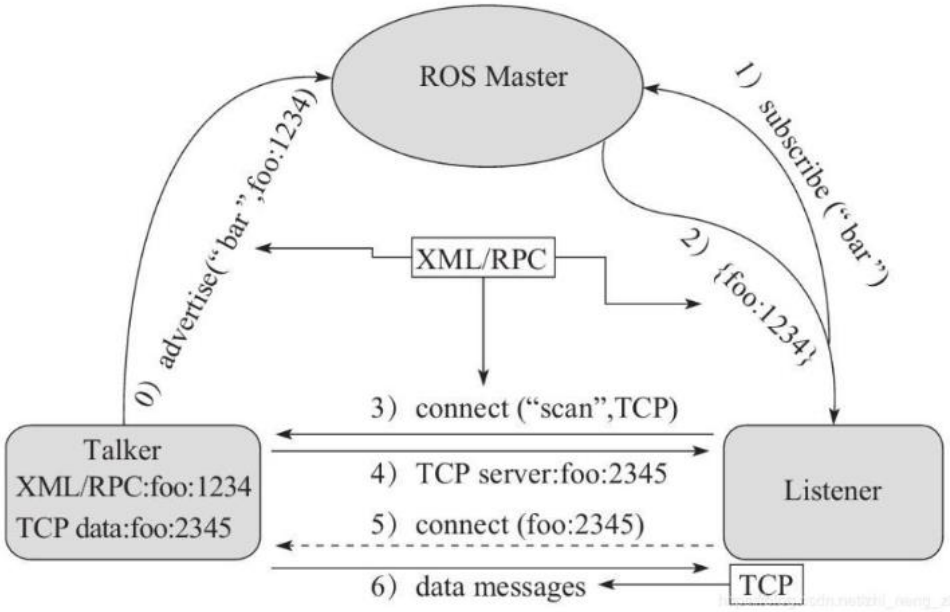
Listener启动后，也会通过RPC在 ROS Master 中注册自身信息，包含需要订阅消息的话题名。ROS Master 会将节点的注册信息加入到注册表中。

2.ROS Master实现信息匹配

ROS Master 会根据注册表中的信息匹配Talker 和 Listener，并通过 RPC 向 Listener 发送 Talker 的 RPC 地址信息。

一、ROS话题通信简介

topic话题通信理论模型



3.Listener向Talker发送请求

Listener 根据接收到的 RPC 地址，通过 RPC 向 Talker 发送连接请求，传输订阅的话题名称、消息类型以及通信协议(TCP/UDP)。

4.Talker确认请求

Talker 接收到 Listener 的请求后，也是通过 RPC 向 Listener 确认连接信息，并发送自身的 TCP 地址信息。

5.Listener与Talker建立连接

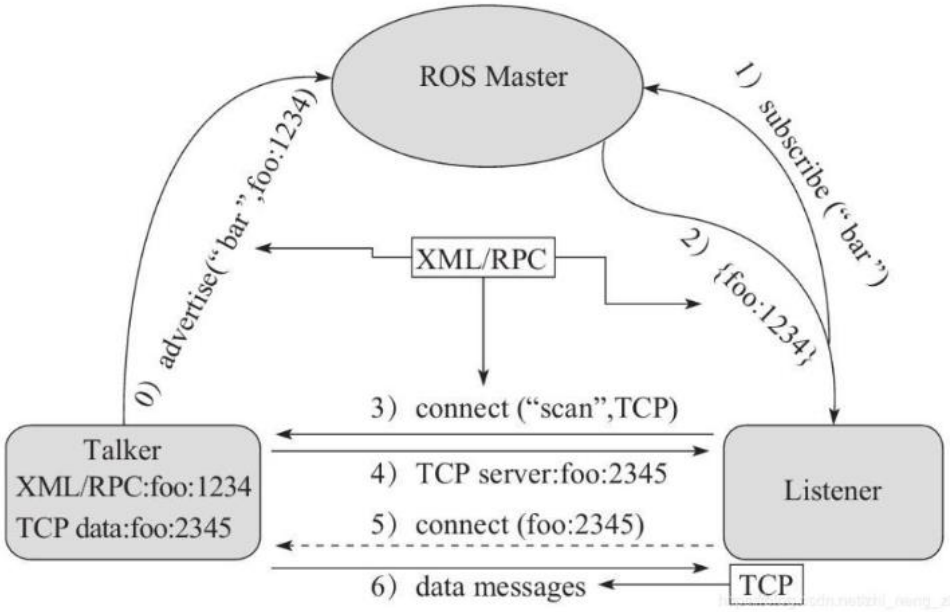
Listener 根据步骤4 返回的消息使用 TCP 与 Talker 建立网络连接。

6.Talker向Listener发送消息

连接建立后，Talker 开始向 Listener 发布消息。

一、ROS话题通信简介

topic话题通信理论模型



注意：

- 上述实现流程中，前五步使用的 RPC 协议，最后两步使用的是TCP 协议
- Talker 与 Listener 的启动无先后顺序要求
- Talker 与 Listener 都可以有多个
- Talker 与 Listener 连接建立后，不再需要 ROS Master。也即，即便关闭ROS Master，Talker 与 Listern 照常通信。

一、ROS话题通信简介

topic话题通信场景示例

机器人在执行导航功能，使用的传感器是激光雷达，机器人会采集激光雷达感知到的信息并计算，然后生成运动控制信息驱动机器人底盘运动。
在上述场景中，就不止一次使用到了话题通信。

- 以激光雷达信息的采集处理为例，在 ROS 中有一个节点需要实时发布当前雷达采集到的数据，导航模块中也有节点会订阅并解析雷达数据。
- 再以运动消息的发布为例，导航模块会根据传感器采集的数据实时的计算出运动控制信息并发布给底盘，底盘也可以有一个节点订阅运动信息并最终转换成控制电机的脉冲信号。

以此类推，像雷达、摄像头、GPS.... 等等一些传感器数据的采集，也都是使用了话题通信，换言之，话题通信适用于不断更新的数据传输相关的应用场景。

一、ROS话题通信简介

topic话题通信特点

- topic通信方式是单向异步的，发送时调用publish()方法，发送完成立即返回，不用等待反馈。

以上述场景为例，所谓异步，即在激光雷达节点每发布一次消息之后，就会继续执行下一个动作，至于消息是什么状态、被怎样处理，它不需要了解；而对于导航模块节点，它只管接收和处理激光雷达发布的topic上的消息，至于是谁发来的，它不会关心。所以激光雷达节点和导航模块节点两者都是各司其责，不存在协同工作，我们称这样的通信方式是异步的。

- subscriber通过回调函数的方式来处理消息。

Subscriber接收消息会进行处理，一般这个过程叫做回调(Callback)。所谓回调就是提前定义好了一个处理函数（写在代码中），当有消息来就会触发这个处理函数，函数会对消息进行处理。

- topic可以同时有多个subscribers，也可以同时有多个publishers。ROS中这样的例子有：/rosout、/tf等等

一、ROS话题通信简介

topic操作指令

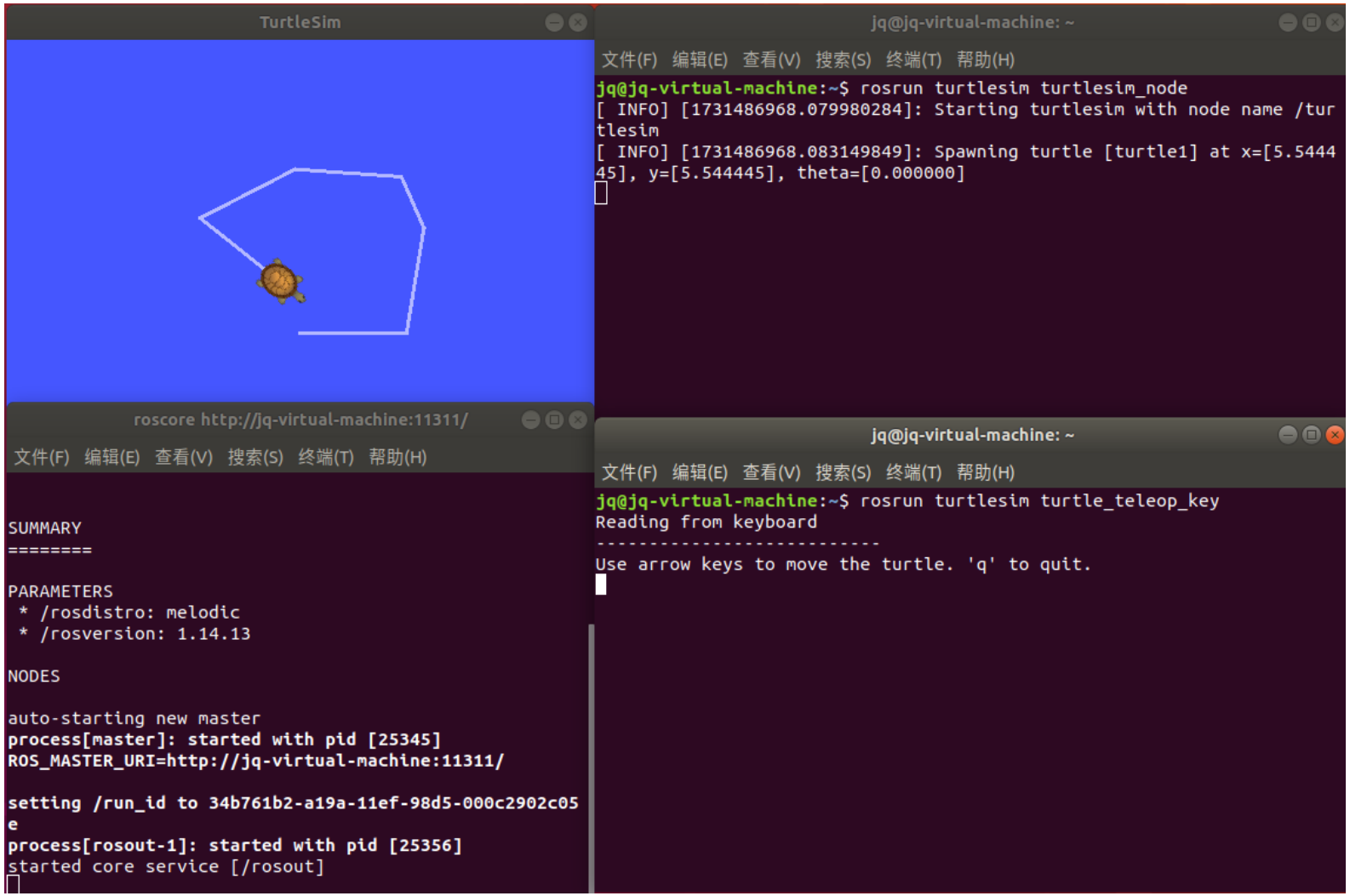
命令	作用
`rostopic list`	列出当前所有的topic
`rostopic info topic_name`	显示某个topic的属性信息
`rostopic echo topic_name`	显示某个topic的内容
`rostopic pub topic_name ...`	向某个topic发布内容
`rostopic bw topic_name`	查看某个topic的带宽
`rostopic hz topic_name`	查看某个topic的频率
`rostopic find topic_type`	查找某个类型的topic
`rostopic type topic_name`	查看某个topic的类型(msg)

如果你一时忘记了命令的写法，可以通过 `rostopic help` 或 `rostopic command -h` 查看具体用法。

1) roscore **启动 ROS 核心服务，提供节点间的通信功能。**

2) rosrun turtlesim turtlesim_node **使用rosrun工具启动turtlesim包中的turtlesim_node节点**

3) rosrun turtlesim turtle_teleop_key **启动键盘控制节点，通过按键发布速度指令控制乌龟的移动。**



- 在 ROS（Robot Operating System）中，**节点是一个独立运行的可执行程序。**ROS 采用分布式架构，**机器人系统中的每个独立功能或模块都被称为一个节点。**节点可以发布消息、订阅消息、提供服务或调用服务。ROS 通过将复杂的机器人系统分解成多个小的、协作的节点来实现系统的模块化和灵活性。

节点是 ROS 系统的基本组成单元

- 一个节点通常是一个独立的进程，可以由不同的编程语言（如 C++、Python 等）编写和运行。
- 例如，一个控制机器人运动的节点和一个处理传感器数据的节点可以彼此独立运行。

节点之间的通信

- 节点通过 **话题（topics）** 进行发布/订阅消息的通信。一个节点可以发布数据到一个话题，其他节点可以订阅该话题并接收数据。
- 节点也可以通过 **服务（services）** 进行请求/响应通信。这种通信方式更类似于函数调用，节点可以请求另一个节点提供的服务并等待其响应。

`turtlesim_node` 的实现

- `turtlesim_node` 是由 **C++ 语言** 编写的，它属于 `turtlesim` 包。你可以在 `turtlesim` 包的源码中找到它的实现文件，这通常是一个 C++ 文件，经过编译后生成一个可执行程序（节点）。
- 这符合我们对节点的描述，即节点通常是一个独立的可执行程序，可能用 C++ 或 Python 等编程语言编写。`turtlesim_node` 就是一个这样的独立可执行程序，它被 `rosrun turtlesim turtlesim_node` 启动，作为一个 ROS 节点运行在系统中。

1. **节点可以用不同的编程语言实现**：ROS 支持多种编程语言，最常见的是 **C++** 和 **Python**。开发者可以选择使用自己熟悉的语言来编写节点，`roscpp` 是 C++ 节点的客户端库，而 `rospy` 是 Python 节点的客户端库。
2. **每个节点的实现方式不同**：像 `turtlesim_node` 这样的节点用 C++ 编写后，编译成二进制文件后可以被执行。另一方面，Python 节点通常是一个 `.py` 文件，带有合适的 shebang（如 `#!/usr/bin/env python`），并设置了可执行权限。

节点是一个独立的进程

- 在 ROS 中，**节点被定义为一个独立的进程**，它执行特定的功能。每个节点运行时都会在操作系统上创建一个进程，与其他节点并行运行。
- 节点的核心作用是通过 ROS 通信框架与其他节点进行交互，例如通过话题（topics）发布/订阅消息、提供/调用服务等。
- 运行中的节点与其他节点交互时，操作系统将它们视为独立的进程，可以独立崩溃、重启或在不同的机器上运行。正是这种特性，使得 ROS 系统具备了高扩展性和分布式特性。

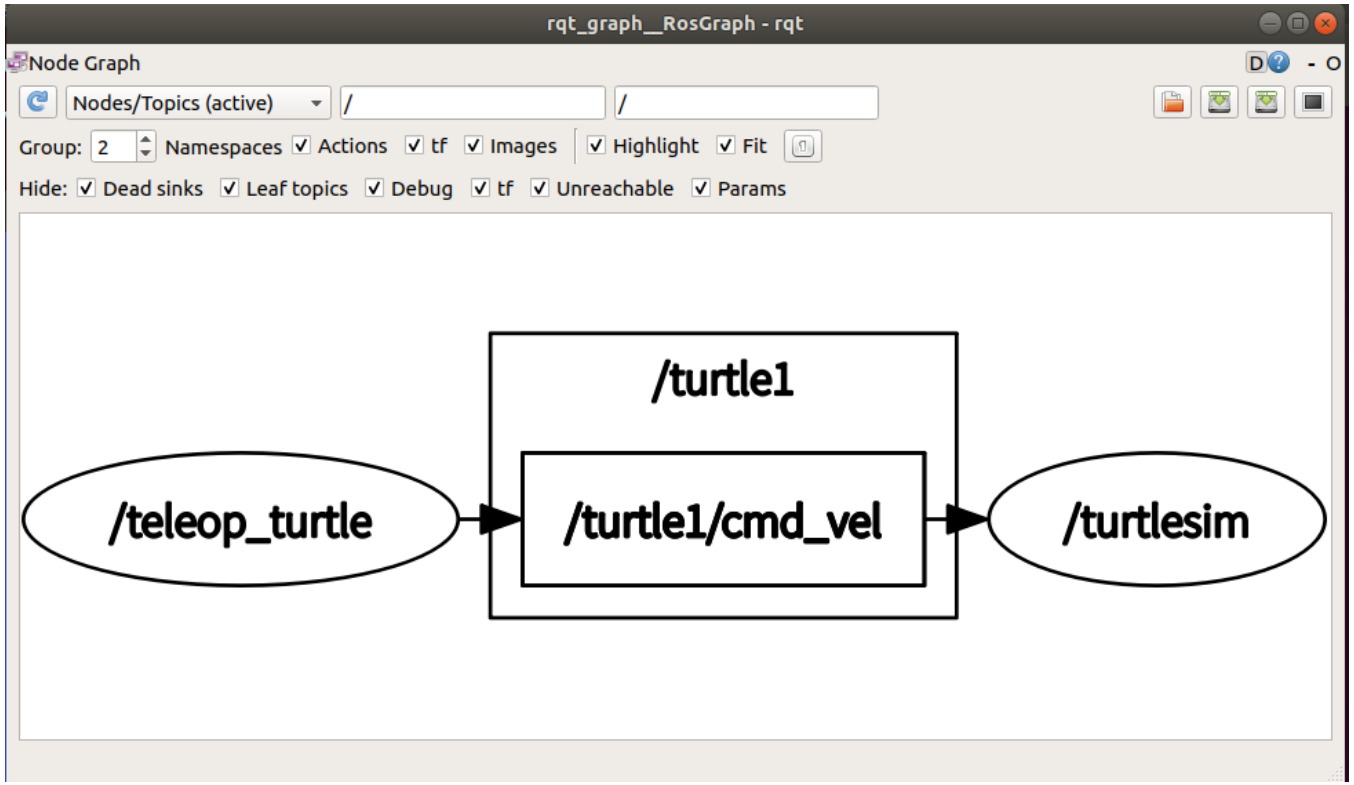
可执行文件和节点的关系

- **可执行文件是节点的实现载体**。一个节点通常由一个源文件（如 C++ 文件或 Python 脚本）编写，编译或解释后生成一个可执行文件。这个可执行文件被 `rosrun` 或 `roslaunch` 启动时，生成一个独立的进程，成为一个运行中的 ROS 节点。
- 换句话说，一个节点的**程序逻辑**被封装在一个可执行文件中，而节点本身是这个可执行文件运行时的实例，是一个独立的进程。

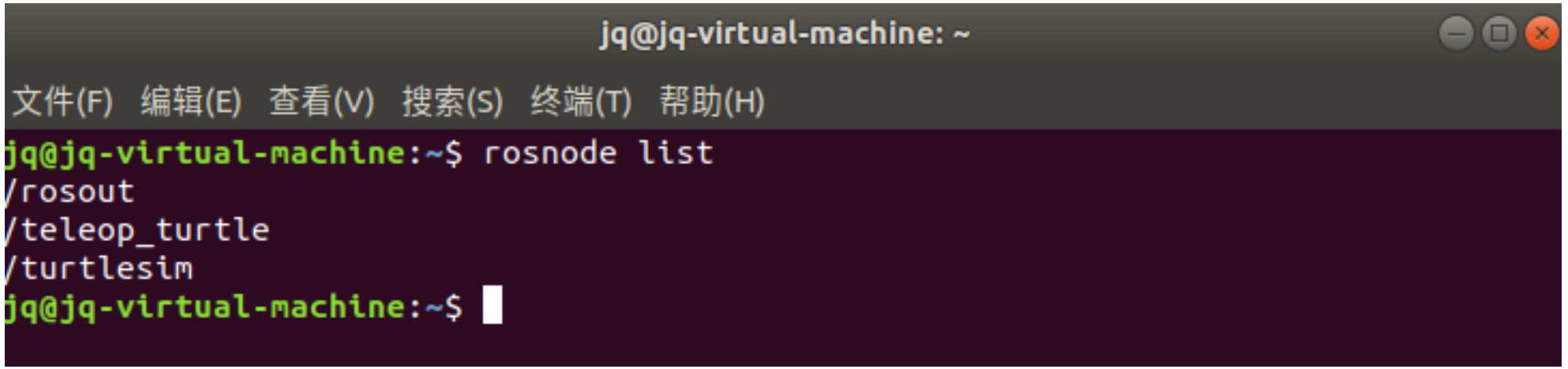
- rosrun rqt_graph rqt_graph

该命令会启动 **RQT Graph**，这是 ROS 中的一个图形化工具，用于可视化当前 ROS 系统中的节点和话题之间的通信关系。

`rqt_graph` 可以帮助你以图形化的方式查看 ROS 系统中节点和话题的连接关系，方便理解系统的结构和数据流。它可以显示节点之间的发布、订阅关系，以及哪些节点在发布或订阅哪些话题。

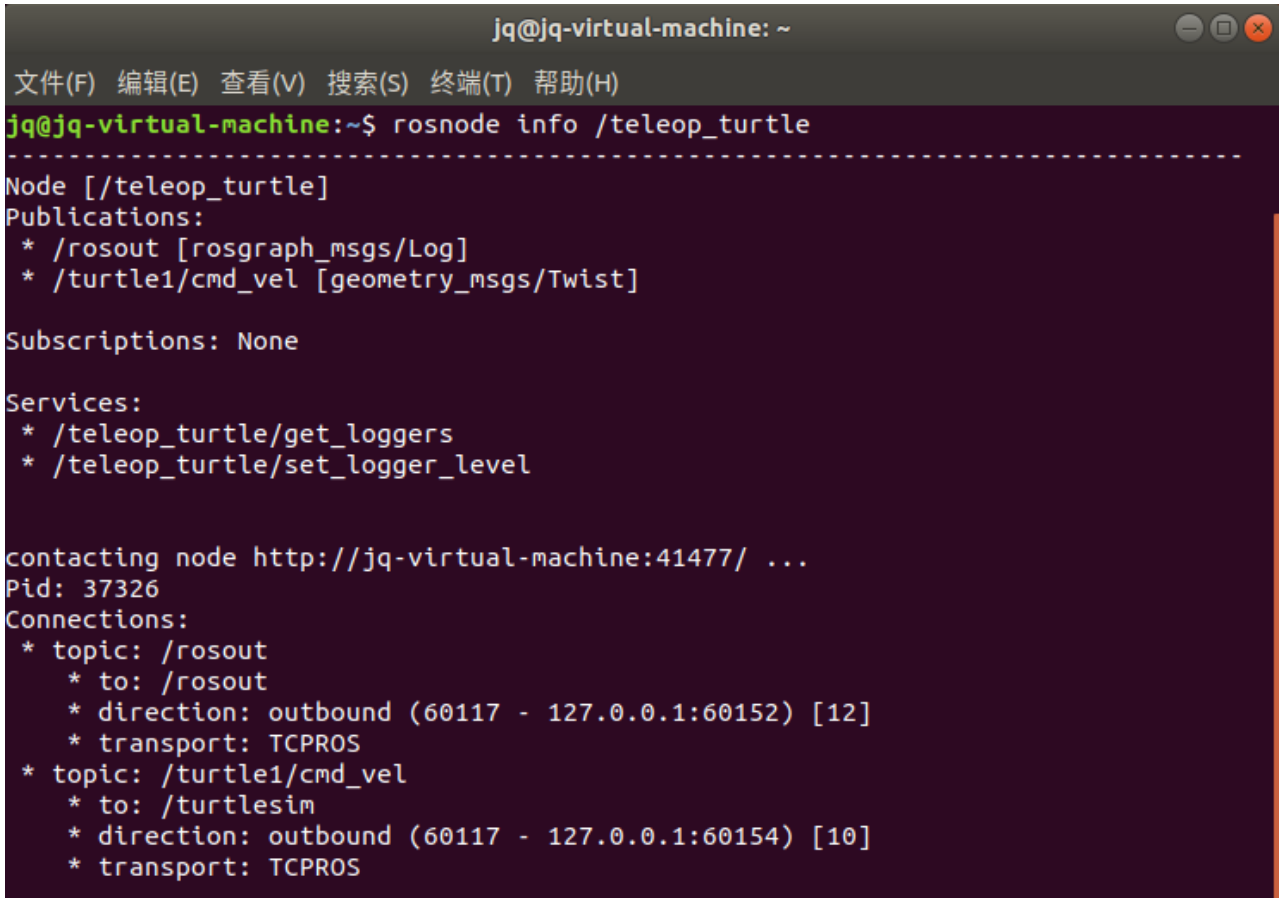


■ rosnode list **列出当前系统中所有正在运行的节点。**



- **/turtlesim**：这是一个节点的名称，可能是由 `turtlesim_node` 启动的仿真节点。
- **/teleop_turtle**：这是一个用于键盘控制乌龟运动的节点。
- **/rosout**：这是一个 ROS 系统的日志节点，所有节点的日志消息会自动发布到 `rosout`，以便进行调试和监控。

■rosnode info /teleop_turtle



- `/teleop_turtle` 是一个用于键盘控制 `turtlesim` 仿真乌龟的节点。
- 它发布两个主题：`/rosout` 用于日志信息，`/turtle1/cmd_vel` 用于速度控制。
- 它没有订阅任何主题。
- 它提供两个服务：日志信息获取和日志级别设置。

- **Node [/teleop_turtle]**：显示该节点的名称，这里是 `/teleop_turtle`。这是一个用于键盘控制 `turtlesim` 中乌龟的节点，通常由 `turtle_teleop_key` 程序启动。

- **Publications**：显示该节点发布的主题（topic），用于与其他节点共享数据。
  - `/rosout`：该节点发布日志信息到 `/rosout` 主题，供 ROS 系统的日志记录器使用。
    - **消息类型**：`rosgraph_msgs/Log`，表示日志消息类型。
  - `/turtle1/cmd_vel`：该节点发布速度指令到 `/turtle1/cmd_vel` 主题，这些指令控制 `turtlesim` 仿真乌龟的运动。
    - **消息类型**：`geometry_msgs/Twist`，表示速度消息的类型，包括线速度和角速度。

- **Subscriptions**：显示该节点订阅的主题。这里显示为 `None`，说明 `/teleop_turtle` 节点没有订阅任何主题，即它不从其他节点接收消息，而是仅仅负责发布控制指令。

- **Services**：显示该节点提供的服务。
  - `/teleop_turtle/get_loggers`：此服务允许获取日志记录器的信息。
  - `/teleop_turtle/set_logger_level`：此服务允许设置日志记录器的级别。

这些服务通常用于调试和日志管理，允许用户查看和控制节点的日志输出。

- **contacting node http://jq-virtual-machine:41477/**：显示节点的 URI 地址，用于与节点通信。
  - `http://jq-virtual-machine:41477/` 是节点的网络地址，表示该节点在 `jq-virtual-machine` 主机上运行，监听端口 `41477`。
- **Pid: 37326**：该节点在操作系统中的进程 ID（PID）。`37326` 是操作系统分配给 `/teleop_turtle` 节点的进程标识符，可以用于在系统中查找或管理该进程。

- **Connections**：显示节点与其他节点的连接信息，包括话题、连接方向、端口等。

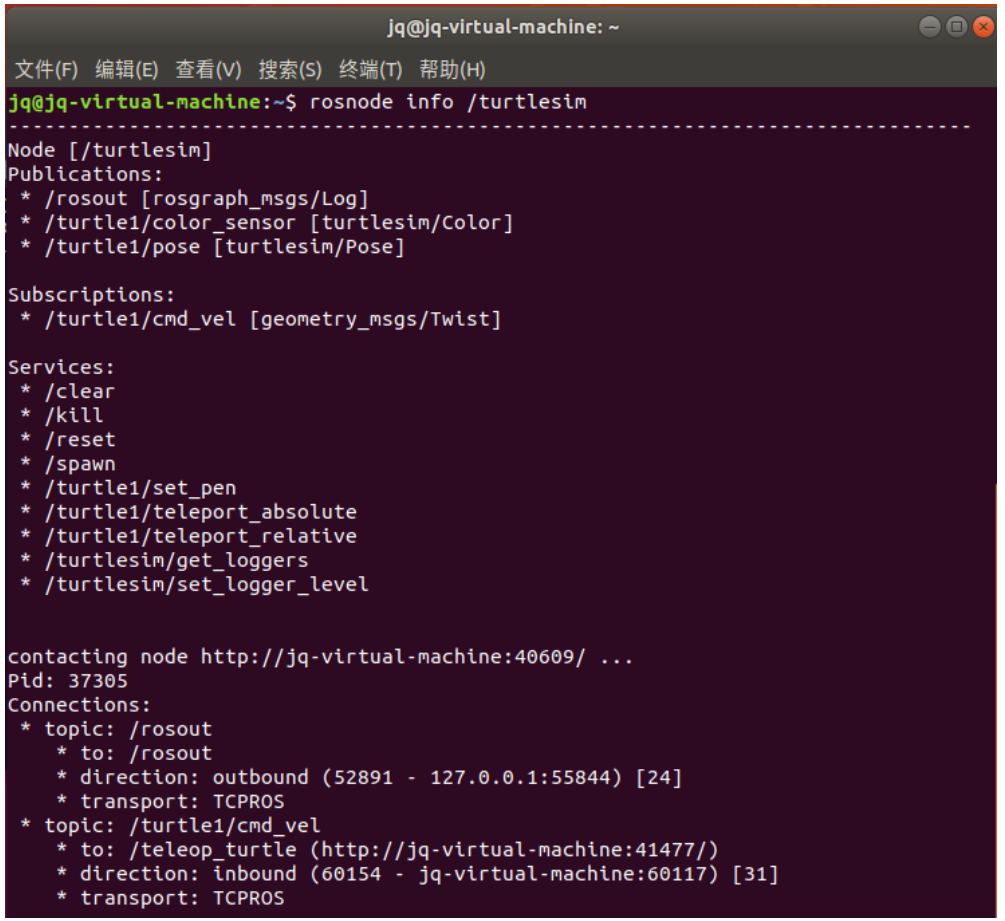
第一个连接： `/rosout`

- **topic: /rosout**：该连接用于向 `/rosout` 主题发布日志消息。
- **to: /rosout**：目标是 `rosout` 节点，这是 ROS 系统的日志节点。
- **direction: outbound**：表示这是一个出站连接，即消息从 `/teleop_turtle` 节点发送到 `/rosout`。
- **(60117 - 127.0.0.1:60152)**：端口信息。
  - **60117**：`/teleop_turtle` 节点的本地端口，用于发送日志数据。
  - **127.0.0.1:60152**：目标是本地（127.0.0.1）上的端口 60152。
- **[12]**：表示此连接的队列大小，可以缓存最多 12 条消息。
- **transport: TCPROS**：使用的传输协议是 TCPROS，这是 ROS 基于 TCP/IP 的传输协议，用于确保消息传递的可靠性。

第二个连接： `/turtle1/cmd_vel`

- **topic: /turtle1/cmd_vel**：该连接用于向 `/turtle1/cmd_vel` 主题发布速度指令。
- **to: /turtlesim**：目标节点是 `/turtlesim`，即用于接收速度指令的 `turtlesim` 仿真节点。
- **direction: outbound**：这是一个出站连接，消息从 `/teleop_turtle` 节点发送到 `/turtlesim` 节点。
- **(60117 - 127.0.0.1:60154)**：端口信息。
  - **60117**：`/teleop_turtle` 节点的本地端口，用于发送速度指令。
  - **127.0.0.1:60154**：目标节点在本地主机上（127.0.0.1）的端口 60154。
- **[10]**：表示此连接的队列大小，最多可以缓存 10 条消息。
- **transport: TCPROS**：传输协议为 TCPROS，确保消息传递的可靠性。

■ rosnode info /turtlesim



- `/turtlesim` 节点发布的主题包括 `/rosout`、`/turtle1/color_sensor` 和 `/turtle1/pose`，用于日志记录、颜色传感器数据和位姿数据的发布。
- `/turtlesim` 节点订阅了 `/turtle1/cmd_vel` 主题，用于接收来自 `/teleop_turtle` 的速度指令来控制乌龟的移动。
- 该节点提供了多个服务用于控制和管理乌龟的状态、位置等。

- **Node [/turtlesim]**：这是节点的名称，表示这是一个名为 `/turtlesim` 的节点，用于控制 `turtlesim` 仿真环境中的乌龟。

- **Publications**：显示该节点发布的主题，用于与其他节点共享数据。
  - `/rosout`：节点发布日志信息到 `/rosout` 主题，供 ROS 系统的日志记录器使用。
    - **消息类型**：`rosgraph_msgs/Log`
  - `/turtle1/color_sensor`：该节点发布乌龟的颜色传感器数据，这个主题会显示乌龟在画布上当前位置的颜色。
    - **消息类型**：`turtlesim/Color`
  - `/turtle1/pose`：该节点发布乌龟的位姿（位置和朝向）。
    - **消息类型**：`turtlesim/Pose`

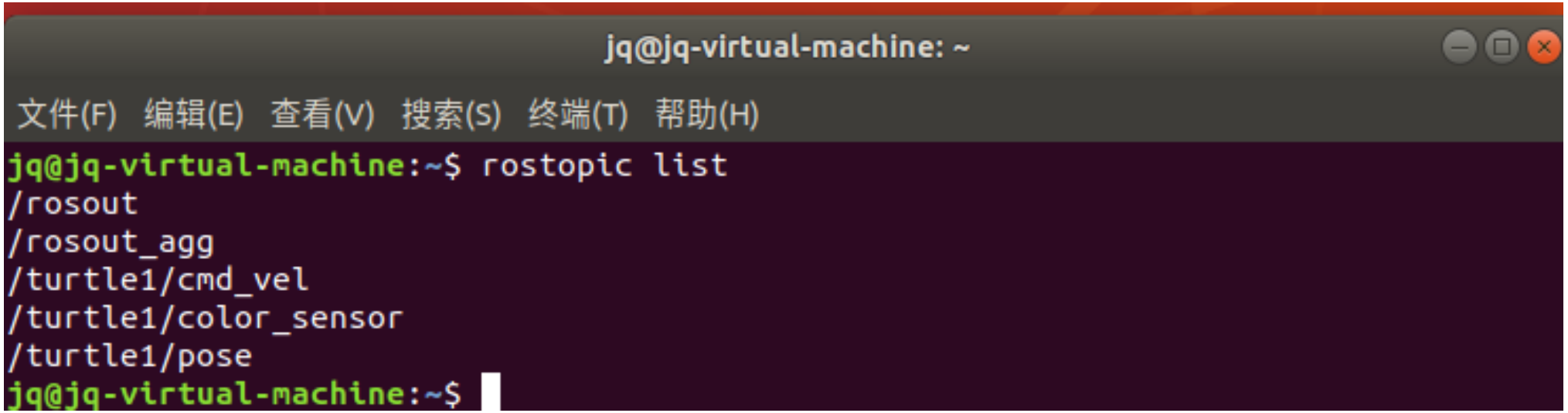
- **Subscriptions**：显示该节点订阅的主题，用于从其他节点接收数据。
  - `/turtle1/cmd_vel`：该节点订阅 `/turtle1/cmd_vel` 主题，用于接收速度控制指令（由 `teleop_turtle` 节点发送的键盘控制指令），从而控制乌龟的移动。
    - **消息类型**：`geometry_msgs/Twist`

- **Services**：显示该节点提供的所有服务。
  - **/clear**：清除画布上的轨迹。
  - **/kill**：删除指定的乌龟。
  - **/reset**：重置仿真环境。
  - **/spawn**：生成一只新的乌龟。
  - **/turtle1/set_pen**：设置乌龟的画笔属性（颜色、粗细等）。
  - **/turtle1/teleport_absolute**：让乌龟瞬间移动到指定的绝对位置。
  - **/turtle1/teleport_relative**：让乌龟瞬间移动到相对位置。
  - **/turtlesim/get_loggers** 和 **/turtlesim/set_logger_level**：管理节点的日志记录功能。

- **topic: /turtle1/cmd_vel**：该连接用于接收 `/teleop_turtle` 节点发送的速度指令。
- **to: /teleop_turtle**：数据来自 `/teleop_turtle` 节点，即用于键盘控制乌龟运动的节点。
- **direction: inbound**：这是一个入站连接，消息从 `/teleop_turtle` 节点发送到 `/turtlesim` 节点。
- **(60154 - jq-virtual-machine:60117)**：端口信息。
  - **60154**：`/turtlesim` 节点的本地端口，用于接收速度指令。
  - **jq-virtual-machine:60117**：`/teleop_turtle` 节点在本地主机上的端口 60117。
- **[31]**：表示此连接的队列大小，最多可以缓存 31 条消息。
- **transport: TCPROS**：传输协议为 TCPROS，确保消息传递的可靠性。

- **topic: /turtle1/cmd_vel**：该连接用于向 `/turtle1/cmd_vel` 主题发布速度指令。
- **to: /turtlesim**：目标节点是 `/turtlesim`，即用于接收速度指令的 `turtlesim` 仿真节点。
- **direction: outbound**：这是一个出站连接，消息从 `/teleop_turtle` 节点发送到 `/turtlesim` 节点。
- **(60117 - 127.0.0.1:60154)**：端口信息。
  - **60117**：`/teleop_turtle` 节点的本地端口，用于发送速度指令。
  - **127.0.0.1:60154**：目标节点在本地主机上（127.0.0.1）的端口 60154。
- **[10]**：表示此连接的队列大小，最多可以缓存 10 条消息。
- **transport: TCPROS**：传输协议为 TCPROS，确保消息传递的可靠性。

■ rostopic list



`/turtle1/cmd_vel`

- **作用**：用于控制 `turtlesim` 仿真环境中乌龟的速度。
- **消息类型**：`geometry_msgs/Twist`
- **发布者**：`teleop_turtle` 节点（用于键盘控制乌龟）或其他控制节点可以发布此话题来控制乌龟的移动。
- **订阅者**：`turtlesim_node` 节点订阅此话题，以获取速度指令并更新乌龟的位置和方向。

■rostopic list

`/turtle1/color_sensor`

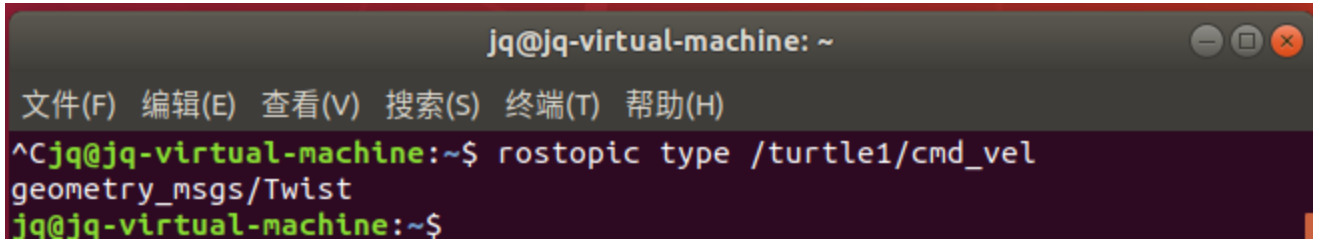
- **作用**：用于发布乌龟当前位置的颜色信息。
- **消息类型**：`turtlesim/Color`
- **发布者**：`turtlesim_node` 节点发布此话题，以报告乌龟在画布上的当前位置的颜色。
- **订阅者**：可以有显示节点或记录数据的节点订阅此话题，用于实时监测或记录乌龟的颜色信息。

`/turtle1/pose`

- **作用**：用于发布乌龟的位姿信息，包括位置和朝向。
- **消息类型**：`turtlesim/Pose`
- **发布者**：`turtlesim_node` 节点发布此话题，以报告乌龟在仿真环境中的当前位姿。
- **订阅者**：其他节点可以订阅此话题来实时获取乌龟的位置和朝向，例如用于监控或进一步控制。

■ rostopic echo /turtle1/cmd_vel

■rostopic echo /turtle1/cmd_vel



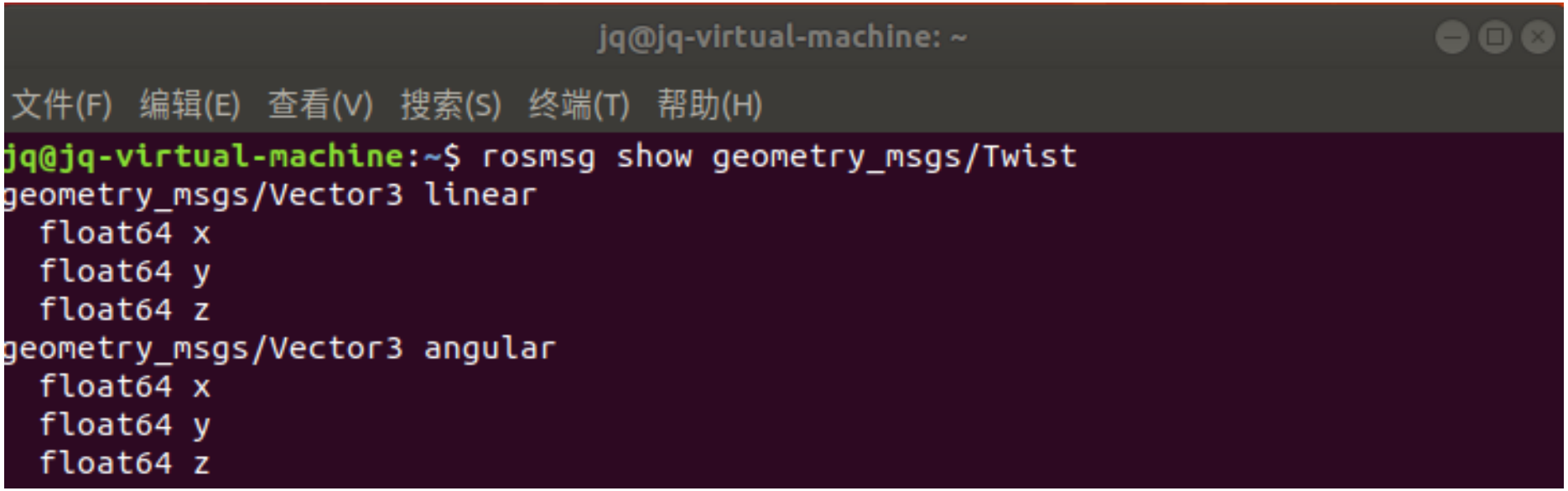
- `geometry_msgs/Twist`：这是 ROS 中定义的消息类型，包含两个部分：`linear` 和 `angular`，用于表示一个物体的线速度和角速度。

1. **linear（线速度）**
   - **x**：沿 x 轴的线速度（浮点数）。
   - **y**：沿 y 轴的线速度（浮点数）。
   - **z**：沿 z 轴的线速度（浮点数）。
2. **angular（角速度）**
   - **x**：绕 x 轴的角速度（浮点数）。
   - **y**：绕 y 轴的角速度（浮点数）。
   - **z**：绕 z 轴的角速度（浮点数）。

- 在 `turtlesim` 中，`/turtle1/cmd_vel` 话题使用 `geometry_msgs/Twist` 类型的消息来控制乌龟的运动。
- 通过设置 `linear.x` 来控制乌龟的前进或后退速度，设置 `angular.z` 来控制乌龟的左右旋转。

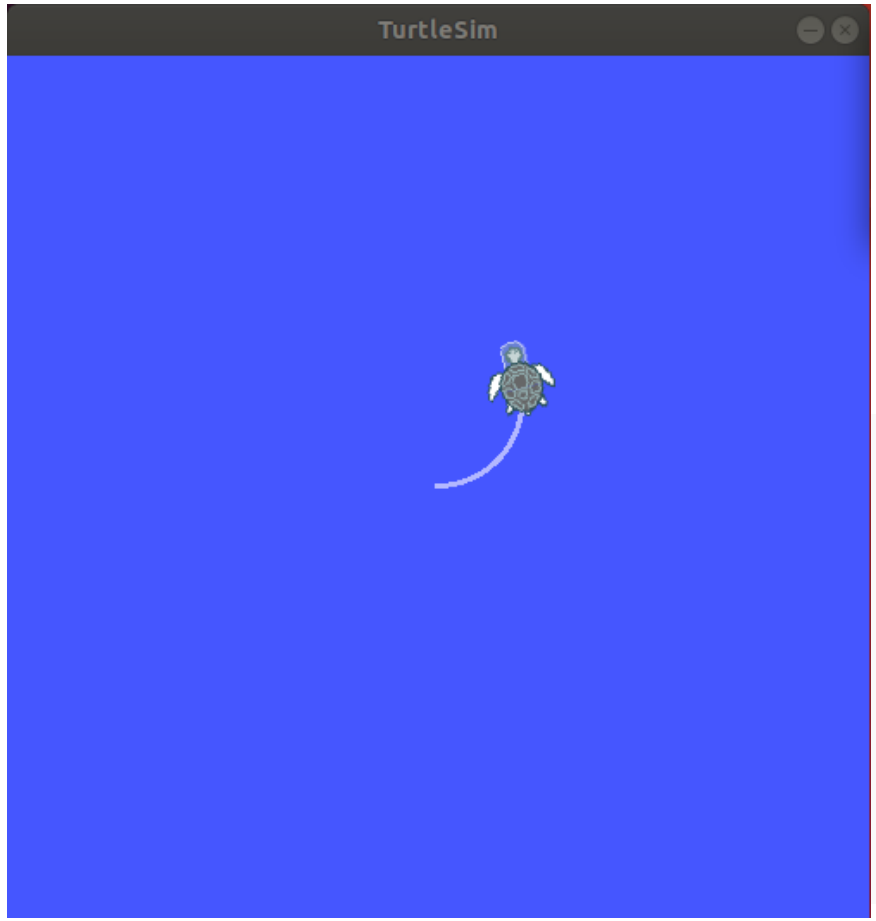
使用 `rostopic type` 命令可以快速查阅话题的消息类型，帮助了解该话题的结构和用途，特别是在调试和开发中非常有用。

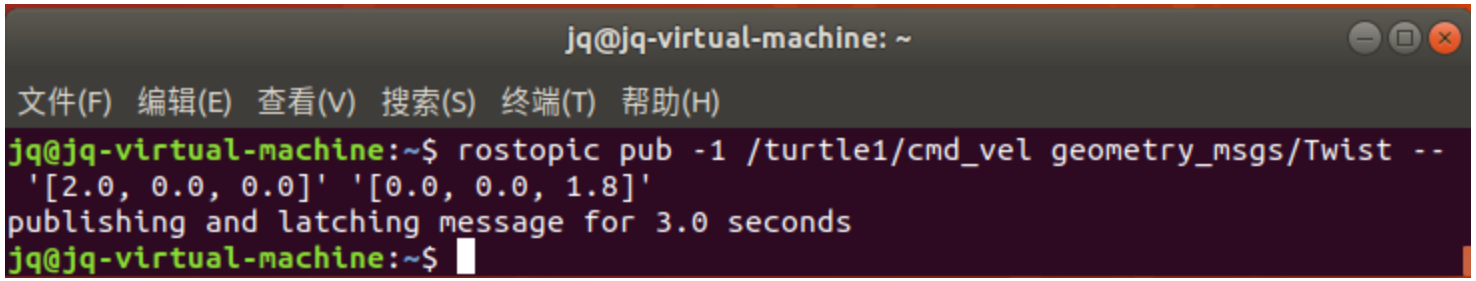
■rosmsg show geometry_msgs/Twist



`geometry_msgs/Twist` 常用于控制移动机器人、无人机等具有线速度和平移速度的设备。通过设置 `linear` 和 `angular` 的不同值，可以实现复杂的运动控制。

■rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'





运行该命令后，`/turtle1/cmd_vel` 话题上会收到一条 `geometry_msgs/Twist` 类型的消息，其内容为：

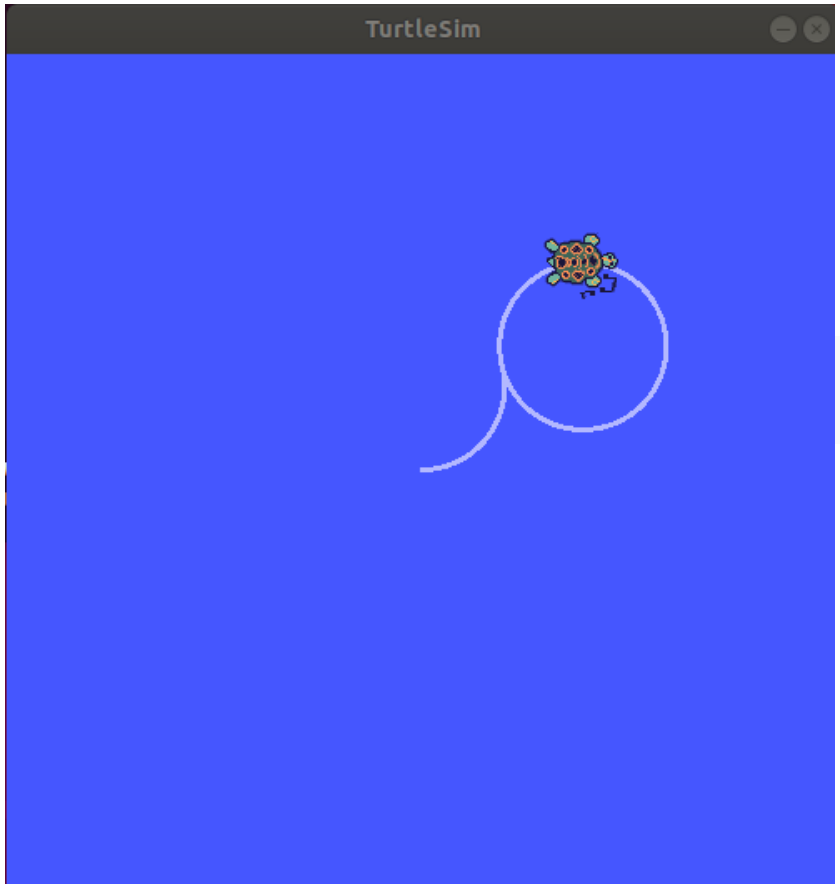
- **线速度**：x 方向为 2.0，y 和 z 方向为 0.0。
  - 这将使乌龟在平面上以 2.0 的速度向前移动。
- **角速度**：z 方向为 1.8，x 和 y 方向为 0.0。
  - 这将使乌龟在平面上以 1.8 的角速度旋转（顺时针旋转）。

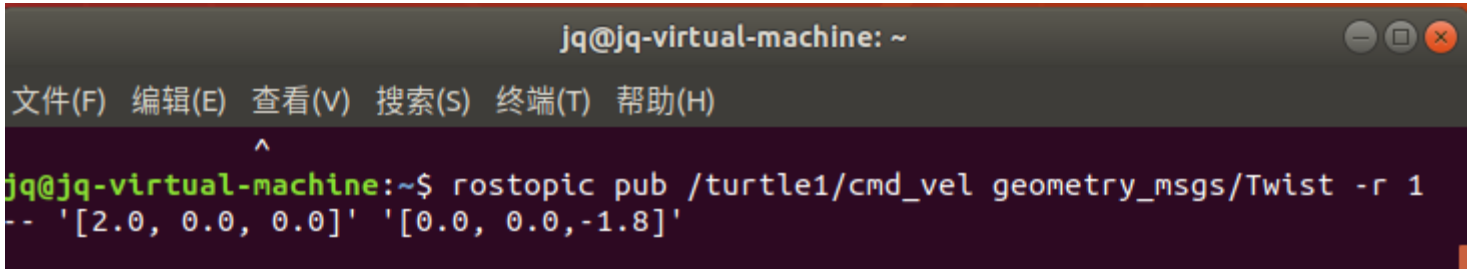
综合效果：乌龟将同时向前移动并顺时针旋转，形成一个螺旋前进的轨迹。

- `rostopic pub`： `rostopic` 的子命令，用于发布消息到指定的话题。
- `-1`：指定只发布一次消息。没有这个参数时， `rostopic pub` 会持续发布消息。
- `/turtle1/cmd_vel`：这是要发布消息的目标话题。在 `turtlesim` 仿真中， `/turtle1/cmd_vel` 是控制乌龟移动的速度指令话题。
- `geometry_msgs/Twist`：指定消息的类型为 `geometry_msgs/Twist`，该消息类型包含线速度（ `linear` ）和角速度（ `angular` ）信息。
- `--`：将消息数据与命令参数分隔开。

- `'[2.0, 0.0, 0.0]'`：设置 `linear` 的值，表示线速度。
  - `2.0`：沿 x 轴的线速度，为 2.0，表示向前移动。
  - `0.0`：沿 y 轴的线速度，为 0.0，乌龟不会在 y 方向移动。
  - `0.0`：沿 z 轴的线速度，为 0.0，乌龟不会在 z 方向移动。
- `'[0.0, 0.0, 1.8]'`：设置 `angular` 的值，表示角速度。
  - `0.0`：绕 x 轴的角速度，为 0.0。
  - `0.0`：绕 y 轴的角速度，为 0.0。
  - `1.8`：绕 z 轴的角速度，为 1.8，表示顺时针旋转。

■rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- ‘[2.0, 0.0, 0.0]’ ‘[0.0, 0.0, -1.8]'

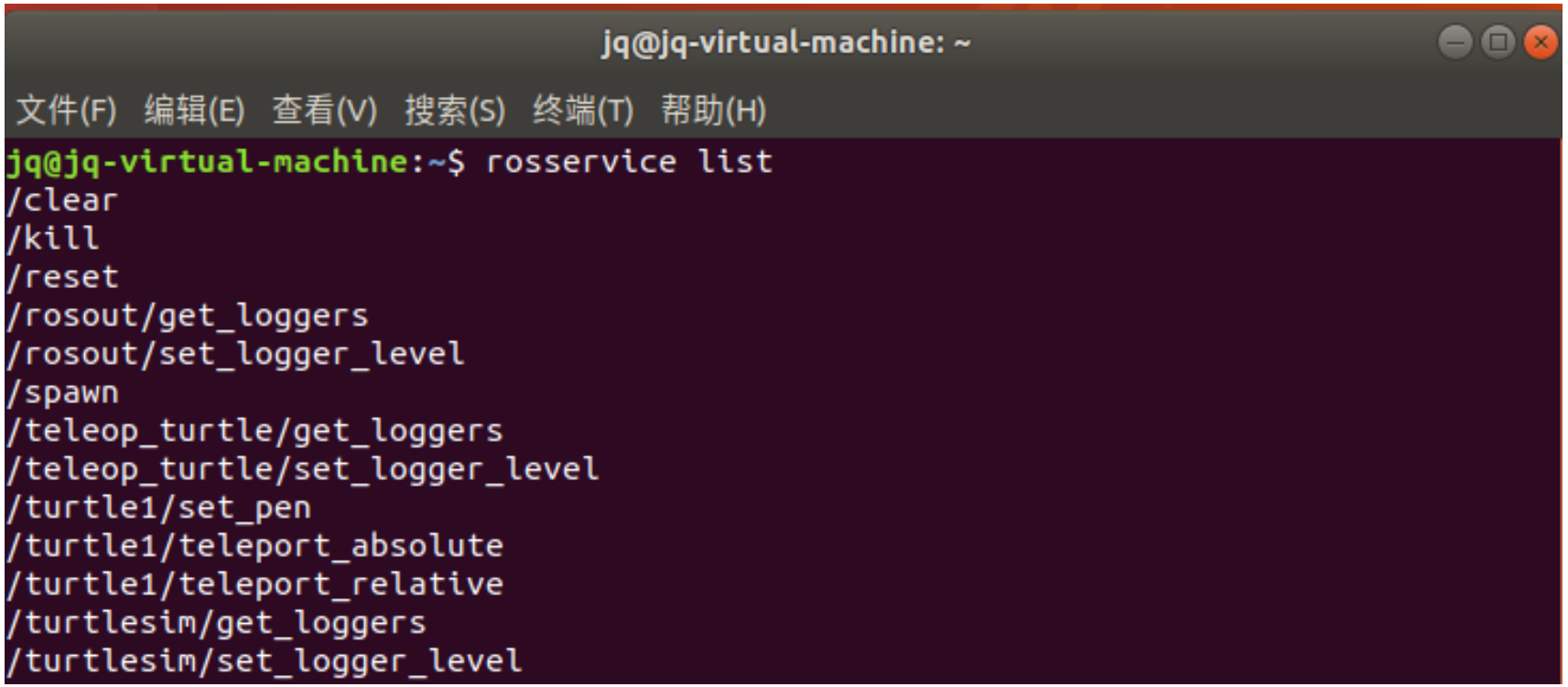




- `-r 1`：指定发布频率为每秒 1 次。这样会以 1Hz 的频率持续发布消息。

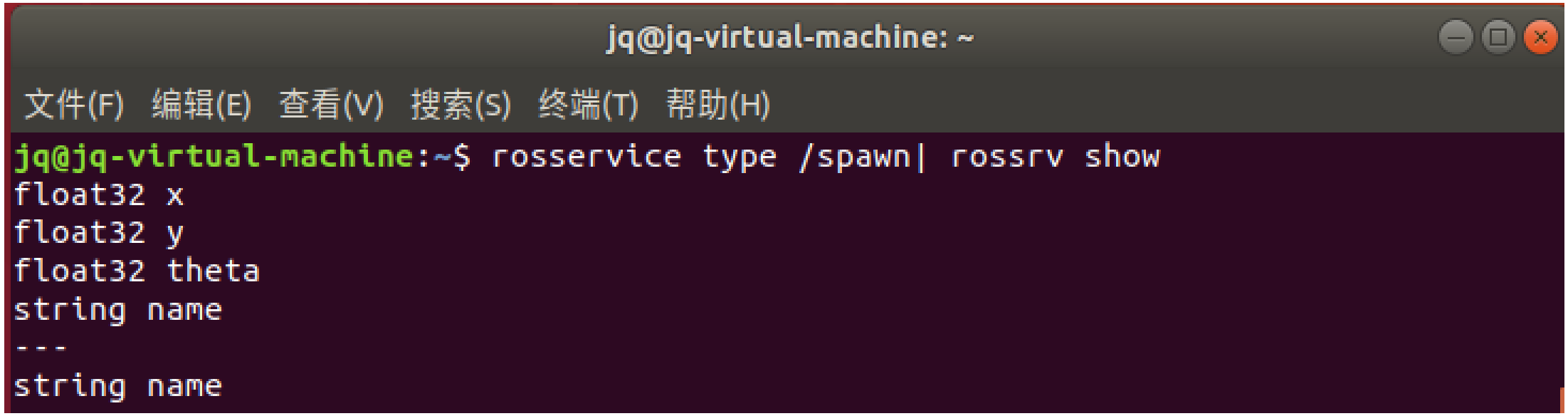
- **角速度**：z 方向为 -1.8，x 和 y 方向为 0.0。
  - 这将使乌龟在平面上以 1.8 的角速度逆时针旋转。

■rosservice list



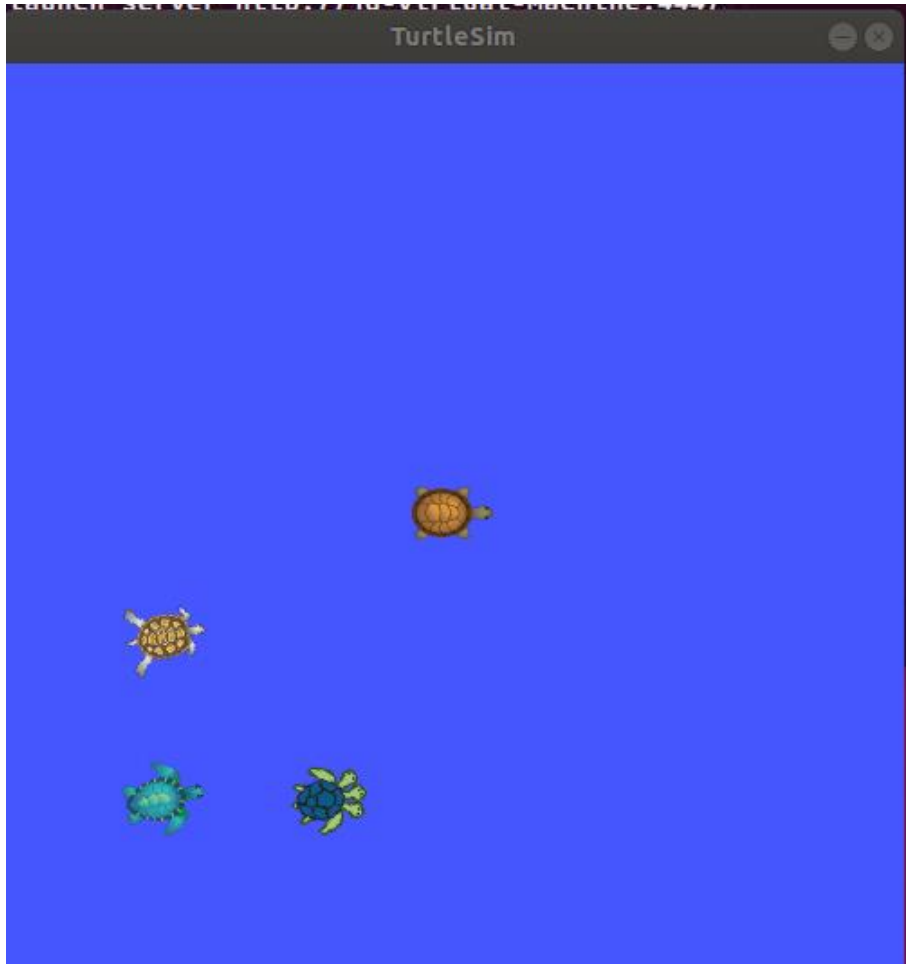
- **/clear**：清除 `turtlesim` 窗口中的所有绘制内容，相当于清空画布。
- **/kill**：用于移除指定名字的乌龟。
- **/reset**：重置 `turtlesim` 模拟器，将所有乌龟返回到默认状态。
- **/rosout/get_loggers** 和 **/rosout/set_logger_level**：用于获取和设置 `rosout` 日志级别的服务。
- **/spawn**：在指定位置和角度生成一个新的乌龟实例。

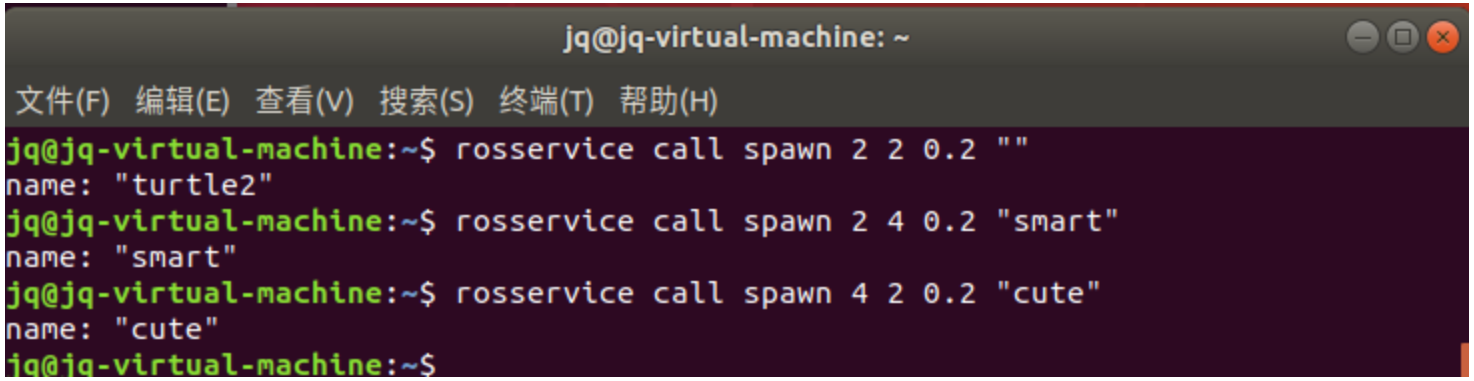
■rosservice type /spawn| rossrv show



- `float32 x`：表示请求中用于指定 `x` 坐标的浮点数。
- `float32 y`：表示请求中用于指定 `y` 坐标的浮点数。
- `float32 theta`：表示请求中用于指定角度（方向）的浮点数。
- `string name`：表示请求中指定的名字，通常用于标识新生成的乌龟名称。
- `---` 表示分隔符，用于区分请求和响应。
- `string name`：表示服务的响应部分，返回生成的乌龟的名字。

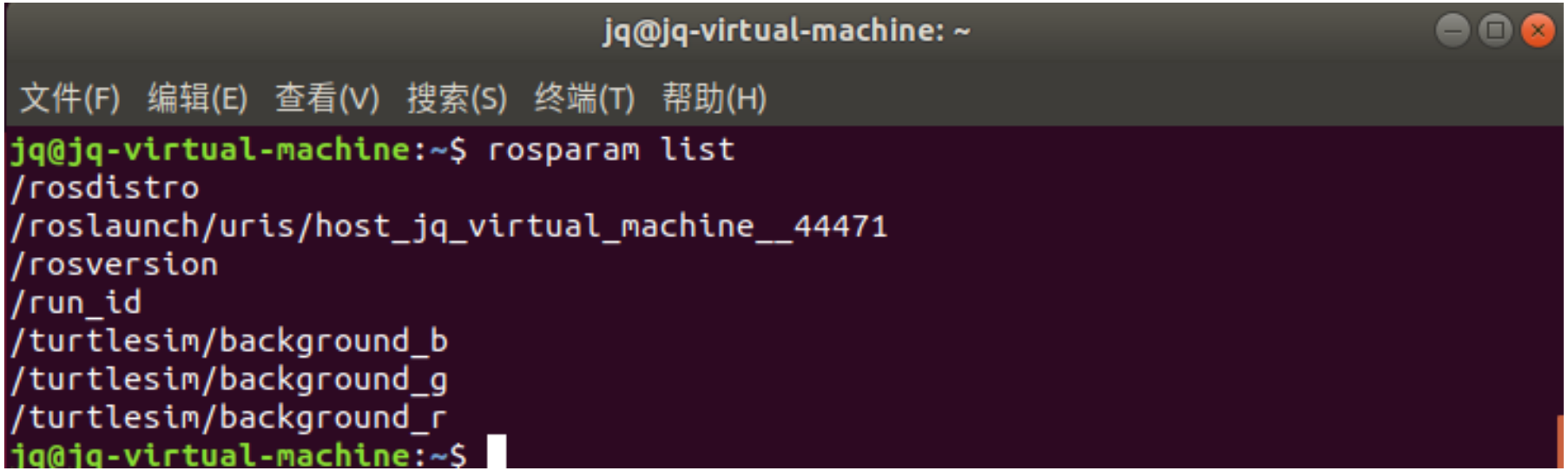
■rosservice call spawn 2 2 0.2 ""





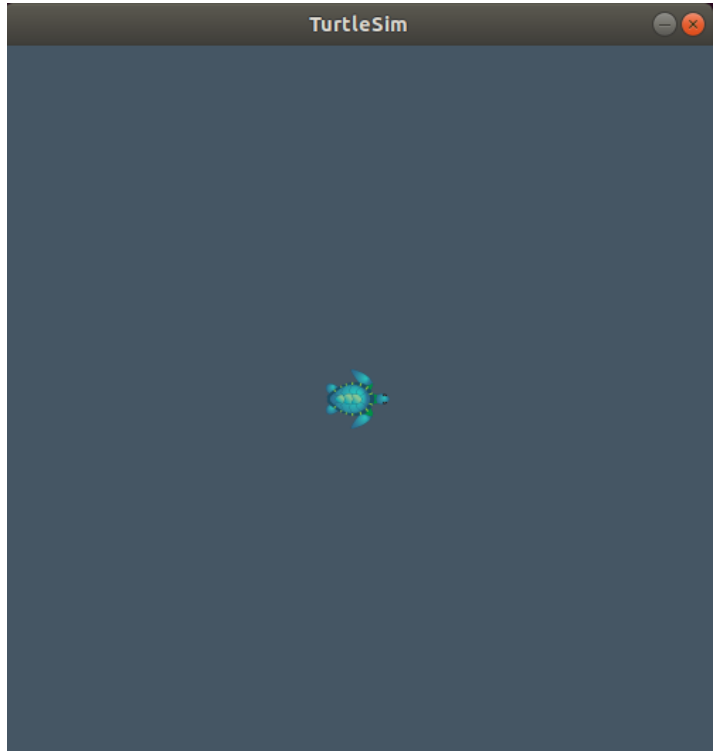
- `rosservice call spawn 2 2 0.2 ""`：
  - 在 `(x=2, y=2)` 的位置生成了一只新乌龟，初始角度为 `0.2` 弧度。
  - 名字留空，系统自动分配了名字 `turtle2`。
- `rosservice call spawn 2 4 0.2 "smart"`：
  - 在 `(x=2, y=4)` 的位置生成了一只新乌龟，初始角度为 `0.2` 弧度。
  - 这只乌龟的名字被指定为 `"smart"`。
- `rosservice call spawn 4 2 0.2 "cute"`：
  - 在 `(x=4, y=2)` 的位置生成了一只新乌龟，初始角度为 `0.2` 弧度。
  - 这只乌龟的名字被指定为 `"cute"`。

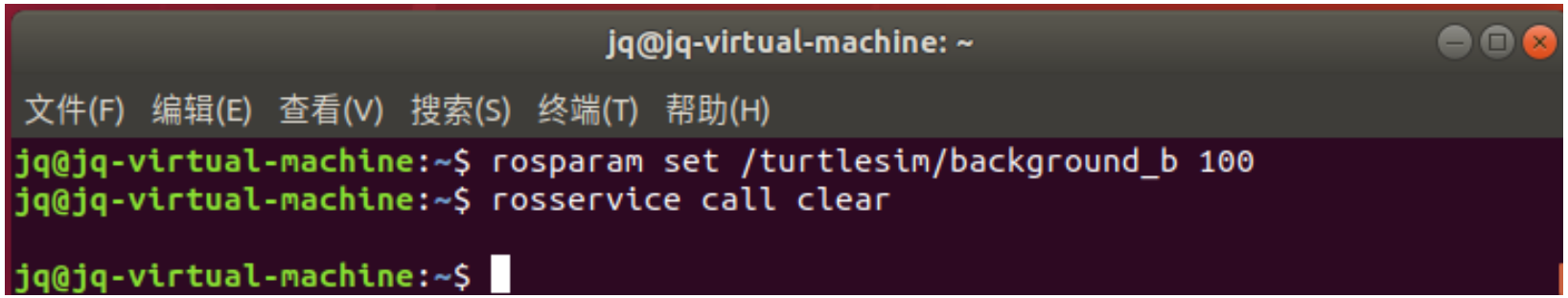
■rosparam list



- **/rosdistro**：表示当前使用的 ROS 发行版名称（例如 `melodic`、`noetic` 等）。
- **/roslaunch/uris/host_jq_virtual_machine__44471**：这是 `roslaunch` URI，用于标识在 `roslaunch` 启动过程中管理的节点集合及其地址信息。
- **/rosversion**：当前使用的 ROS 版本号。
- **/run_id**：一个 UUID，标识每次 `roscore` 启动时生成的唯一运行 ID。
- **/turtlesim/background_b**：`turtlesim` 窗口背景的蓝色通道值。
- **/turtlesim/background_g**：`turtlesim` 窗口背景的绿色通道值。
- **/turtlesim/background_r**：`turtlesim` 窗口背景的红色通道值。

- rosparam set /turtlesim/background_b 100
- rosservice call clear





1. `rosparam set /turtlesim/background_b 100`：此命令将 `turtlesim` 背景的蓝色通道值设置为 `100`。通过修改这个参数，可以调整 `turtlesim` 模拟器窗口的背景颜色。
2. `rosservice call clear`：这个命令会清除 `turtlesim` 窗口中的所有绘制内容（即清空乌龟的轨迹），并且在调用 `clear` 服务后，背景颜色会更新为你设置的新值。这是因为 `turtlesim` 会在 `clear` 操作后重新渲染背景颜色，显示你刚刚设置的参数值。

实践项目：

1. 编写python脚本文件控制乌龟做正方形运动（发布者）

创建 Python 脚本：

- 编写一个 Python 脚本来控制乌龟的运动。
- 使用 ROS 的 `rospy` 库与 ROS 通信，并发布 `geometry_msgs/Twist` 消息来控制乌龟的线速度和角速度。

实现脚本逻辑：

- 初始化 ROS 节点。
- 创建一个 `Publisher`，将消息发布到 `/turtle1/cmd_vel` 话题，用于控制乌龟的速度。
- 设置 `Twist` 消息中的线速度和角速度，使得乌龟沿着圆周路径运动。
- 使用 `rospy.Rate` 控制发布频率，使得运动平滑持续。

实践项目：

1. 编写python脚本文件控制乌龟做正方形运动

- 将脚本保存并赋予执行权限。 `chmod +x move_circle.py`
- 启动 ROS 主节点（`roscore`）和 `turtlesim` 节点。 `rosrun turtlesim turtlesim_node`

然后运行脚本以观察乌龟的圆周运动。 `rosrun <your_package_name> move_circle.py`

实践项目：

2. 编写python脚本文件手动控制乌龟运动，并在终端实时打印当前乌龟的位姿。（订阅者）

整体流程

1. 启动 `roscore`。
2. 启动 `turtlesim_node`。
3. 使用 `turtle_teleop_key` 控制乌龟的运动。
4. 运行订阅者脚本 `turtle_pose_listener.py`，观察终端输出的乌龟位姿信息。

三、ROS话题通信消息

msg简介

topic有很严格的格式要求，比如摄像头节点发布的rgb图像topic，它就必然要遵循ROS中定义好的rgb图像格式。这种数据格式就是Message。Message按照定义解释就是topic内容的数据类型，也称之为topic的格式标准。

在 ROS 通信协议中，数据载体是一个较为重要组成部分，ROS 中通过 std_msgs 封装了一些原生的数据类型,比如:String、Int32、Int64、Char、Bool、Empty.... 但是，这些数据一般只包含一个 data 字段，结构的单一意味着功能上的局限性，当传输一些复杂的数据，比如: 激光雷达的信息... std_msgs
由于描述性较差而显得力不从心，这种场景下可以使用自定义的消息类型

msgs只是简单的文本文件，每行具有字段类型和字段名称，可以使用的字段类型有:

- int8, int16, int32, int64 (或者无符号类型: uint*)
- string
- other msg files
- float32, float64
- time, duration
- variable-length array[] and fixed-length array[C]

ROS中还有一种特殊类型： Header ，标头包含时间戳和ROS中常用的坐标帧信息。会经常看到msg文件的第一行具有Header标头 。

三、ROS话题通信消息

常见的message

常见的message类型，包括std_msgs, sensor_msgs, nav_msgs, geometry_msgs等

Vector3.msg
#文件位置:geometry_msgs/Vector3.msg
float64 x
float64 y
float64 z

Accel.msg
#定义加速度项，包括线性加速度和角加速度
#文件位置:geometry_msgs/Accel.msg
Vector3 linear
Vector3 angular

Header.msg
#定义数据的参考时间和参考坐标
#文件位置:std_msgs/Header.msg
uint32 seq #数据ID
time stamp #数据时间戳
string frame_id #数据的参考坐标系

Echos.msg
#定义超声传感器
#文件位置:自定义msg文件
Header header
uint16 front_left
uint16 front_center
uint16 front_right
uint16 rear_left
uint16 rear_center
uint16 rear_right

Quaternion.msg
#定义空间中旋转的四元数
#文件位置:geometry_msgs/Quaternion.msg
float64 x
float64 y
float64 z
float64 w

三、ROS话题通信消息

常见的message

Imu.msg

#消息包含了从惯性原件中得到的数据，加速度为m/^2，角速度为rad/s
#如果所有的测量协方差已知，则需要全部填充进来如果只知道方差，则只填充协方差矩阵的对角数据即可

#文件位置：sensor_msgs/Imu.msg
Header header
Quaternion orientation
float64[9] orientation_covariance
Vector3 angular_velocity
float64[9] angular_velocity_covariance
Vector3 linear_acceleration
float64[] linear_acceleration_covariance

Point.msg

#空间中的点的位置
#文件位置:geometry_msgs/Point.msg
float64 x
float64 y
float64 z

Pose.msg

#消息定义自由空间中的位姿信息，包括位置和指向信息
#文件位置:geometry_msgs/Pose.msg
Point position
Quaternion orientation

PoseStamped.msg

#定义有时空基准的位姿
#文件位置：geometry_msgs/PoseStamped.msg
Header header
Pose pose

三、ROS话题通信消息

自定义msg

示例:创建自定义消息，该消息包含人的信息:姓名、身高、年龄等。

流程:

1. 按照固定格式创建 msg 文件
2. 编辑配置文件
3. 编译生成可以被 Python 或 C++ 调用的中间文件

1.定义msg文件

功能包下新建 msg 目录，添加文件 Person.msg

```
string name
uint16 age
float64 height
```

2.编辑配置文件

package.xml中添加编译依赖与执行依赖

```
<build_depend>message_generationbuild_depend>
<exec_depend>message_runtimeexec_depend>
```

CMakeLists.txt编辑 msg 相关配置

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
# 需要加入 message_generation,必须有 std_msgs

## 配置 msg 源文件
add_message_files(
  FILES
  Person.msg
)
# 生成消息时依赖于 std_msgs
generate_messages(
  DEPENDENCIES
  std_msgs
)
#执行时依赖
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES demo02_talker_listener
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
#  DEPENDS system_lib
)
```

三、ROS话题通信消息

- rosmsg操作命令

rosmsg命令	作用
`rosmsg list`	列出系统上所有的msg
`rosmsg show msg_name`	显示某个msg的内容

实践项目：

3.创建自定义msg（包含4种字段类型），编写python脚本文件实现话题通信中自定义消息的发布和订阅。

1. **创建消息文件**：
   - 在 ROS 包的 `msg` 目录中创建一个新的消息文件，例如 `CustomMessage.msg`。
   - 在 `msg/CustomMessage.msg` 文件中定义消息的字段和数据类型（例如 `int32`、`float32`、`string`、`bool`）：

2. **修改 `CMakeLists.txt` 文件**：
   - 在 `add_message_files` 部分中添加自定义消息文件：

   - 在 `generate_messages` 部分中确保包含 `std_msgs` 依赖：

实践项目：

3.创建自定义msg（包含4种字段类型），编写python脚本文件实现话题通信中自定义消息的发布和订阅。

3. **修改** `package.xml` **文件**：
    - 添加 `message_generation` 和 `message_runtime` 依赖：

4. **编译工作空间**：
    - 在工作空间目录下运行以下命令以生成消息文件：

```
catkin_make
source devel/setup.bash
```

    - 确保自定义消息被成功编译。

实践项目：

3.创建自定义msg（包含4种字段类型），编写python脚本文件实现话题通信中自定义消息的发布和订阅。

2. 编写发布节点

创建一个 Python 脚本 `custom_publisher.py`，用于发布自定义消息。

3. 编写订阅节点

创建另一个 Python 脚本 `custom_subscriber.py`，用于订阅并打印接收到的自定义消息。

4. 运行节点

1. **启动** `roscore`：
2. **运行发布节点**：`rosrun my_custom_package custom_publisher.py`
3. **运行订阅节点**：`rosrun my_custom_package custom_subscriber.py`

03

第三节
ROS服务通信

一、ROS服务通信简介

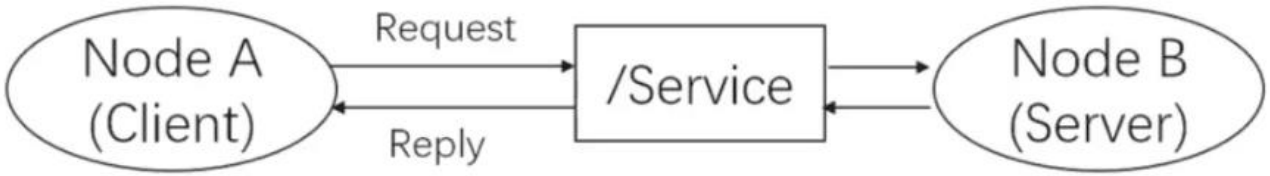
service服务通信概念

上一节我们介绍了ROS的通信方式中的topic(话题)通信，我们知道topic是ROS中的一种单向的异步通信方式。然而有些时候单向的通信满足不了通信要求，比如当一些节点只是临时而非周期性的需要某些数据，如果用topic通信方式时就会消耗大量不必要的系统资源，造成系统的低效率高功耗。

这种情况下，就需要有另外一种基于请求-响应式的通信方式——service(服务)通信。

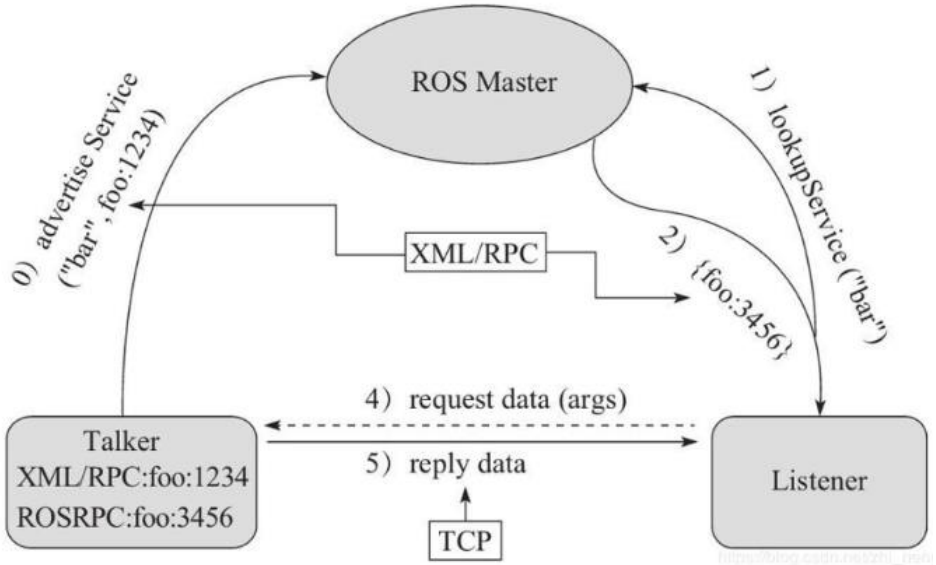
为了解决以上问题，service方式在通信模型上与topic做了区别。Service通信是双向的，它不仅可以发送消息，同时还会有反馈。即: 一个节点A向另一个节点B发送请求，B接收处理请求并产生响应结果返回给A。

其结构示意图如下:



一、ROS服务通信简介

service服务通信理论模型



- ROS Master (管理者)
- Server (服务端)
- Client (客户端)

0.Server注册

Server 启动后，会通过RPC在 ROS Master 中注册自身信息，其中包含提供的服务的名称。ROS Master 会将节点的注册信息加入到注册表中。

1.Client注册

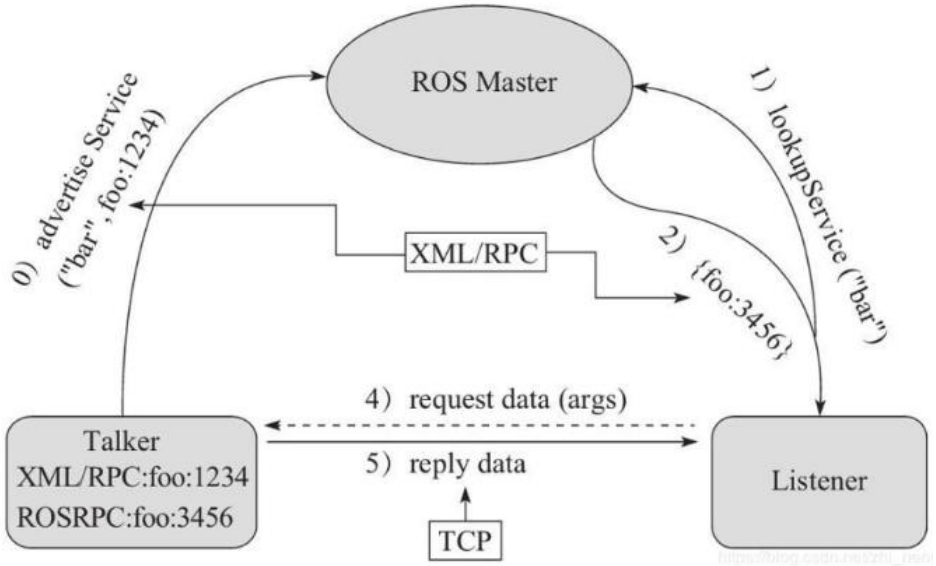
Client 启动后，也会通过RPC在 ROS Master 中注册自身信息，包含需要请求的服务的名称。ROS Master 会将节点的注册信息加入到注册表中。

2.ROS Master实现信息匹配

ROS Master 会根据注册表中的信息匹配 Server和 Client，并通过 RPC 向 Client 发送 Server 的 TCP 地址信息。

一、ROS服务通信简介

service服务通信理论模型



3.Client发送请求

Client 根据步骤2 响应的信息，使用 TCP 与 Server 建立网络连接，并发送请求数据。

4.Server发送响应

Server 接收、解析请求的数据，并产生响应结果返回给 Client。

注意：

- 客户端请求被处理时，需要保证服务器已经启动；
- 服务端和客户端都可以存在多个。

一、ROS服务通信简介

service服务通信场景示例

机器人巡逻过程中，控制系统分析传感器数据发现可疑物体或人... 此时需要拍摄照片并留存。

在上述场景中，就使用到了服务通信。

- 一个节点需要向相机节点发送拍照请求，相机节点处理请求，并返回处理结果

与上述类似，服务通信更适用于偶然的、对实时性有要求、有一定逻辑处理需求的数据传输场景。

service服务通信特点

- service通信方式是双向同步的

所谓同步就是说，此时Node A发布请求后会在原地等待reply，直到Node B处理完了请求并且完成了reply，Node A才会继续执行。Node A等待过程中，是处于阻塞状态的通信。这样的通信模型没有频繁的消息传递，没有冲突与高系统资源的占用，只有接受请求才执行服务，简单而且高效。

一、ROS服务通信简介

- service && topic 对比

名称	Topic	Service
通信方式	异步通信	同步通信
实现原理	TCP/IP	TCP/IP
通信模型	Publish-Subscribe	Request-Reply
映射关系	Publish-Subscribe(多对多)	Request-Reply（多对一）
特点	接受者收到数据会回调（Callback）	远程过程调用（RPC）服务器端的服务
应用场景	连续、高频的数据发布	偶尔使用的功能/具体的任务
举例	激光雷达、里程计发布数据	开关传感器、拍照、逆解计算

一、ROS服务通信简介

service操作指令

rosservice 命令	作用
`rosservice list`	显示服务列表
`rosservice info`	打印服务信息
`rosservice type`	打印服务类型
`rosservice uri`	打印服务ROSRPC uri
`rosservice find`	按服务类型查找服务
`rosservice call`	使用所提供的args调用服务
`rosservice args`	打印服务参数

二、ROS服务通信srv

简介

类似msg文件，srv文件是用来描述服务service数据类型的，service通信的数据格式定义在*.srv中。它声明了一个服务，包括请求(request)和响应（reply）两部分。格式声明示例：

msgs_demo/srv/DetectHuman.srv

```
bool start_detect
---
my_pkg/HumanPose[] pose_data
```

msgs_demo/msg/HumanPose.msg

```
std_msgs/Header header
string uuid
int32 number_of_joints
my_pkg/JointPose[]joint_data
```

msgs_demo/msg/JointPose.msg

```
string joint_name
geometry_msgs/Pose pose
floar32 confidence
```

以DetectHUman.srv文件为例，该服务例子取自OpenNI的人体检测ROS软件包。它是用来查询当前深度摄像头中的人体姿态和关节数的。

srv文件格式很固定，第一行是请求的格式，中间用“---”隔开，第三行是应答的格式。

在本例中，请求为是否开始检测，应答为一个数组，数组的每个元素为某个人的姿态（HumanPose）。

而对于人的姿态，其实是一个msg，所以srv中可以嵌套msg，但msg不能嵌套srv。

二、ROS服务通信srv

自定义srv

示例:创建自定义srv，客户端请求发送的两个数字，服务器响应返回的两数之和

流程:

1. 按照固定格式创建srv文件
2. 编辑配置文件
3. 编译生成中间文件

1.定义srv文件

功能包下新建 srv 目录，添加 xxx.srv 文件，内容:

```
# 客户端请求时发送的两个数字
int32 num1
int32 num2
---
# 服务器响应发送的数据
int32 sum
```

3.编译

编译后的中间文件查看:

C++ 需要调用的中间文件(.../工作空间/devel/include/包名/xxx.h)

Python 需要调用的中间文件(.../工作空间/devel/lib/python3/dist-packages/包名/srv)

后续调用相关 srv 时，是从这些中间文件调用的

2.编辑配置文件

package.xml中添加编译依赖与执行依赖

```
<build_depend>message_generationbuild_depend>
<exec_depend>message_runtimeexec_depend>
```

CMakeLists.txt编辑 srv 相关配置

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
# 需要加入 message_generation,必须有 std_msgs
```

```
add_service_files(
  FILES
  AddInts.srv
)
```

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

三、ROS服务通信案例与实现

service服务通信案例

需求:

服务通信中，客户端提交两个整数至服务端，服务端求和并响应结果到客户端，请创建服务器与客户端通信的数据载体。

分析:

在模型实现中，ROS master 不需要实现，而连接的建立也已经被封装了，需要关注的关键点有三个:

1. 服务端
2. 客户端
3. 数据

c++流程:

1. 编写服务端实现;
2. 编写客户端实现;
3. 编辑配置文件;
4. 编译并执行。

python流程:

1. 编写服务端实现;
2. 编写客户端实现;
3. 为python文件添加可执行权限;
4. 编辑配置文件;

三、ROS服务通信案例与实现

service服务通信基本操作（python）

1. 编写服务端实现

1.导包
2.初始化 ROS 节点
3.创建服务对象
4.回调函数处理请求并产生响应
5.spin 函数

```python
# 1.导包
import rospy
from demo03_server_client.srv import AddInts,AddIntsRequest,AddIntsResponse
# 回调函数的参数是请求对象，返回值是响应对象
def doReq(req):
    # 解析提交的数据
    sum = req.num1 + req.num2
    rospy.loginfo("提交的数据:num1 = %d, num2 = %d, sum = %d",req.num1, req.num2, sum)

    # 创建响应对象，赋值并返回
    # resp = AddIntsResponse()
    # resp.sum = sum
    resp = AddIntsResponse(sum)
    return resp


if __name__ == "__main__":
    # 2.初始化 ROS 节点
    rospy.init_node("addints_server_p")
    # 3.创建服务对象
    server = rospy.Service("AddInts",AddInts,doReq)
    # 4.回调函数处理请求并产生响应
    # 5.spin 函数
    rospy.spin()
```

三、ROS服务通信案例与实现

service服务通信基本操作（python）

2. 编写客户端实现

1.导包
2.初始化 ROS 节点
3.创建请求对象
4.发送请求
5.接收并处理响应

```python
#1.导包
import rospy
from demo03_server_client.srv import *
import sys

if __name__ == "__main__":

    #优化实现
    if len(sys.argv) != 3:
        rospy.logerr("请正确提交参数")
        sys.exit(1)
```

```python
# 2.初始化 ROS 节点
rospy.init_node("AddInts_Client_p")
# 3.创建请求对象
client = rospy.ServiceProxy("AddInts",AddInts)
# 请求前，等待服务已经就绪
# 方式1:
# rospy.wait_for_service("AddInts")
```

```python
# 方式2
client.wait_for_service()
# 4.发送请求,接收并处理响应
# 方式1
# resp = client(3,4)
# 方式2
# resp = client(AddIntsRequest(1,5))
# 方式3
req = AddIntsRequest()
# req.num1 = 100
# req.num2 = 200

#优化
req.num1 = int(sys.argv[1])
req.num2 = int(sys.argv[2])

resp = client.call(req)
rospy.loginfo("响应结果:%d",resp.sum)
```

三、ROS服务通信案例与实现

service服务通信基本操作（python）

3.设置权限

终端下进入 scripts 执行: chmod +x *.py

4.配置 CMakeLists.txt

CMakeLists.txt

```
catkin_install_python(PROGRAMS
  scripts/AddInts_Server_p.py
  scripts/AddInts_Client_p.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

5.执行

流程:

- 需要先启动服务: rosrun 包名 服务
- 然后再调用客户端 : rosrun 包名 客户端 参数1 参数2

结果:

会根据提交的数据响应相加后的结果。

实践项目：

4. 在乌龟显示节点的指定位置生成4只自定义名字的乌龟，并修改窗口背景色（编写python脚本文件实现）。

1. **创建自定义服务类型**：
   - 在 ROS 包的 `srv` 目录中创建自定义服务文件 `SpawnTurtlesWithBackground.srv`。
   - 定义服务请求和响应格式，包括乌龟的位置信息、名字和背景颜色。
2. **修改** `CMakeLists.txt` **和** `package.xml`：
   - 在 `CMakeLists.txt` 中添加 `add_service_files` 和 `generate_messages`：
   - 在 `package.xml` 中确保包含 `message_generation` 和 `message_runtime` 依赖：
3. **编译消息和服务**：

实践项目：

4. 在乌龟显示节点的指定位置生成4只自定义名字的乌龟，并修改窗口背景色（编写python脚本文件实现）。

4. **编写服务器端脚本**：
   - 创建服务器端 Python 脚本 `spawn_turtles_with_background_server.py`，用于生成乌龟并设置背景颜色。
5. **编写客户端脚本**：
   - 创建客户端 Python 脚本 `spawn_turtles_with_background_client.py`，用于发送请求以生成乌龟并修改背景颜色。
6. **运行服务器和客户端**：
   1. **启动** `roscore`： 2. **启动** `turtlesim_node`：3. **运行服务器节点**：4. **运行客户端节点**：

实践项目：

5. 创建自定义srv，客户端请求发送四个数字a、b、c、d，服务器响应返回（a - b）×（c+d）（编写python脚本文件实现）。

1. **创建自定义服务类型**
   - 在 ROS 包的 `srv` 目录中创建一个新的服务文件，命名为 `CalculateOperation.srv`。
   - 定义服务的请求和响应格式：请求包含四个数字 `a`、`b`、`c`、`d`，响应包含计算结果。
2. **修改 `CMakeLists.txt` 和 `package.xml`**
   - 在 `CMakeLists.txt` 文件中添加服务文件，并生成消息和服务代码：
   - 在 `package.xml` 文件中，确保包含 `message_generation` 和 `message_runtime` 依赖：
3. **编译服务文件**

实践项目：

5. 创建自定义srv，客户端请求发送四个数字a、b、c、d，服务器响应返回（a - b）×（c+d）（编写python脚本文件实现）。

4. **编写服务器端脚本**
   - 创建一个 Python 脚本 `calculate_operation_server.py`，用于实现服务器的逻辑来处理请求并计算 `(a - b) * (c + d)`。
5. **编写客户端脚本**
   - 创建另一个 Python 脚本 `calculate_operation_client.py`，用于发送请求并接收服务器的响应。
6. **运行服务器和客户端**

   1. **启动** `roscore`： 2. **运行服务器节点**： 3. **运行客户端节点**：

04

第四节
ROS参数服务器

一、ROS参数服务器简介

ROS参数服务器概念

参数服务器在ROS中主要用于实现不同节点之间的数据共享。参数服务器相当于是独立于所有节点的一个公共容器，可以将数据存储在该容器中，被不同的节点调用，当然不同的节点也可以往其中存储数据，用于配置参数，全局共享参数。参数服务器使用互联网传输，在节点管理器中运行，实现整个通信过程。

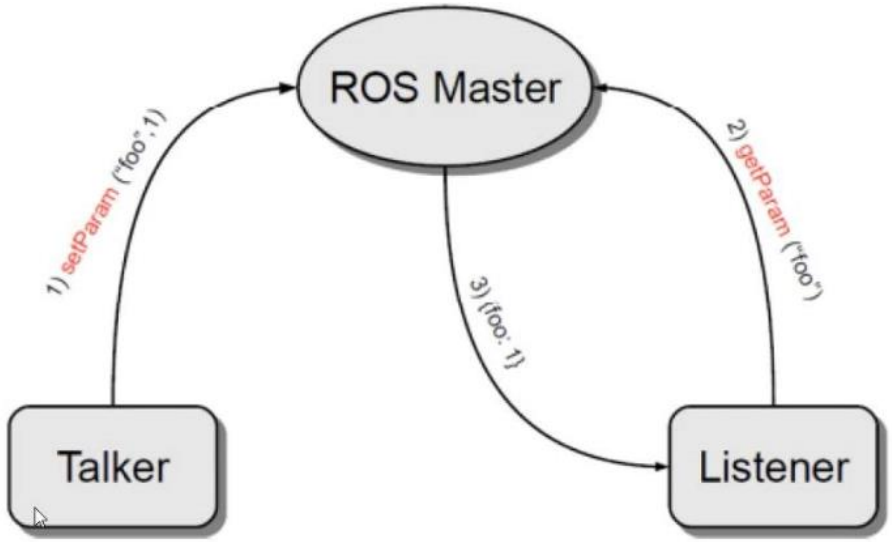
参数服务器，作为ROS中另外一种数据传输方式，有别于topic和service，它更加的静态。参数服务器维护着一个数据字典，字典里存储着各种参数和配置。

字典其实就是一个个的键值对（key : value），每一个key不重复，且每一个key对应着一个value。字典就是一种映射关系，在实际的项目应用中，因为字典的这种静态的映射特点，我们往往将一些不常用到的参数和配置放入参数服务器里的字典里，这样对这些数据进行读写都将方便高效。

第四节 ROS参数服务器

一、ROS参数服务器简介

ROS参数服务器理论模型



- ROS Master (管理者)
- Talker (参数设置者)
- Listener (参数调用者)

1.Talker 设置参数

Talker 通过 RPC 向参数服务器发送参数(包括参数名与参数值)，ROS Master 将参数保存到参数列表中。

2.Listener 获取参数

Listener 通过 RPC 向参数服务器发送参数查找请求，请求中包含要查找的参数名。

3.ROS Master 向 Listener 发送参数值

ROS Master 根据步骤2请求提供的参数名查找参数值，并将查询结果通过 RPC 发送给 Listener。

参数可使用数据类型: 32-bit integers、booleans、strings、doubles、iso8601 dates、lists、base64-encoded binary data、字典。

注意: 参数服务器不是为高性能而设计的，因此最好用于存储静态的非二进制的简单数据。

一、ROS参数服务器简介

ROS参数服务器场景示例

导航实现时，会进行路径规划，比如：全局路径规划，设计一个从出发点到目标点的大致路径。本地路径规划，会根据当前路况生成实时的行进路径。

上述场景中，全局路径规划和本地路径规划时，就会使用到参数服务器：

- 路径规划时，需要参考小车的尺寸，我们可以将这些尺寸信息存储到参数服务器，全局路径规划节点与本地路径规划节点都可以从参数服务器中调用这些参数。

参数服务器，一般适用于存在数据共享的一些应用场景。

ROS参数服务器维护方式

参数服务器的维护方式非常的简单灵活，总的来讲有三种方式：

- 命令行维护
- launch文件内读写
- node源码

二、ROS参数服务器操作

命令行维护

使用命令行来维护参数服务器，主要使用rosparam语句来进行操作的各种命令

rosparam 命令	作用
`rosparam set param_key param_value`	设置参数
`rosparam get param_key`	显示参数
`rosparam load file_name`	从文件加载参数
`rosparam dump file_name`	保存参数到文件
`rosparam delete`	删除参数
`rosparam list`	列出参数名称

load和dump文件需要遵守YAML格式（键值对的形式），示例如下：

```
name:'Zhangsan'
age:20
gender:'M'
score{Chinese:80,Math:90}
score_history:[85,82,88,90]
```

二、ROS参数服务器操作

launch文件内读写

launch文件中有很多标签，而与参数服务器相关的标签只有两个，一个是<param>，另一个<rosparam>。这两个标签功能比较相近，但<param>一般只设置一个参数。
示例：

```
（1）<param name="robot_description" command="$(find xacro)/xacro.py $(find robot_sim_demo)/urdf/robot.xacro" />
<!--在Gazebo中启动机器人模型-->
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
    args="-urdf -x $(arg x) -y $(arg y) -z $(arg z) -Y $(arg yaw) -model xbot2 -param robot_description"/>
```

param只给出了key，没有直接给出value，这里的value是由脚本运行结果定义的。

```
<!--把关节控制的配置信息读到参数服务器-->
（2）<rosparam file="$(find robot_sim_demo)/config/xbot2_control.yaml" command="load"/>
```

rosparam的典型用法，先指定一个YAML文件，然后施加command,其效果等于rosparam load file_name。

```
（3）<param name="publish_frequency" value="100.0"/>
```

param定义了一个key和一个value，交给了参数服务器维护。

node源码

除了上述最常用的两种读写参数服务器的方法，还有一种就是修改ROS的源码，也就是利用API来对参数服务器进行操作。