

实验指导书

实验 1 熟悉 ROS 操作系统、学习 ROS 常用指令

1、实验目的

掌握 ROS 的安装以及常用命令，具体内容如下：

- (1) 学会在 ubuntu18.04 系统下安装 ROS。
- (2) 了解 ROS 基础命令。
- (3) 运行海龟示例。

2、实验设备

硬件环境：PC 机

系统环境：Ubuntu18.04、ROS Melodic

3、实验内容

- (1) 安装 ROS（机房电脑已安装，无需重复安装，此为提供安装教程可以在自己电脑安装）

- 添加国内源（清华源）

```
sudo sh -c 'cat /etc/lsb-release && echo "deb http://mirrors.tuna.tsinghua.edu.cn/ros/ubuntu/`lsb_release -cs` main" > /etc/apt/sources.list.d/ros-latest.list'
```

- 添加私钥

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add -
```

如果提示不能添加，则使用下面的命令：

```
sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net --recv-key 0xB01FA116
```

- 更新软件列表

```
sudo apt-get update
```

若出现如下错误

W: GPG 错误：http://mirrors.ustc.edu.cn/ros/ubuntu xenial InRelease: 由于没有公钥，无法验证下列签名： NO_PUBKEY F42ED6FBAB17C654

则执行如下命令添加公钥

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys F42ED6FBAB17C654
```

- 安装 ROS-Melodic（时间很长）

```
sudo apt-get install ros-melodic-desktop-full
```

- 配置环境

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

- 安装 **ros package**

```
sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool
build-essential
```

- 安装 **rosdep**

```
sudo apt install python-rosdep
sudo rosdep init
```

- 更新 **rosdep**

```
rosdep update
```

测试 ros 是否安装成功，执行

```
roscore
```

若最后显示 started core service [/rosout]，则安装成功。

终止 roscore 的运行，使用 `ctrl + c`。

(2) ROS 常用指令以及小海龟示例

- **roscore 命令**

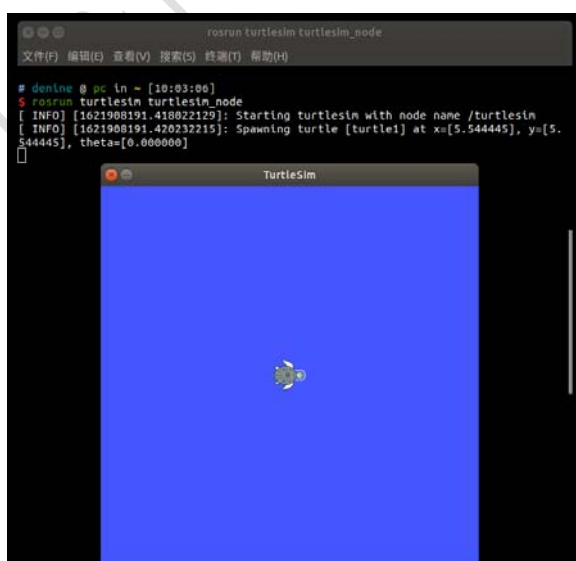
ROS 的使用需要一个主节点 master，通过执行 roscore 命令即可启动一个主节点。

- **roslaunch 命令**

roslaunch 命令可以启动某个包中的某个节点，以安装 ROS 时附带的小海龟功能包为例，在启动 roscore 后，在新窗口执行

```
roslaunch turtlesim turtlesim_node
```

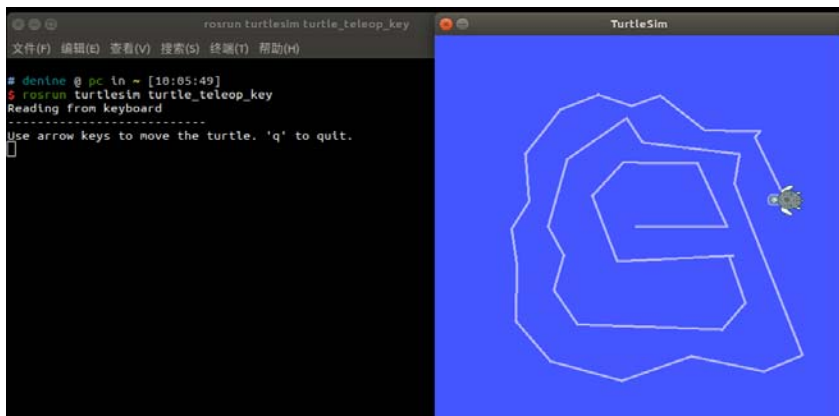
即可启动 turtlesim 包下的 turtlesim_node 节点，出现小海龟窗口，如下图所示



此外，继续打开新窗口输入 `roslaunch turtlesim turtlesim` 后敲击两下 Tab 键，即可显示出 `turtlesim` 包下的所有节点，然后补全命令，执行 `turtle_teleop_key`

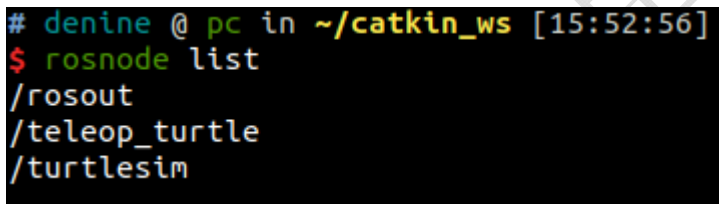
```
roslaunch turtlesim turtlesim
```

即可启动小海龟的控制节点，此时光标聚焦于该窗口，即可使用键盘的上下左右键控制，之前打开的小海龟，如下图所示。



• roslaunch 命令

执行 `roslaunch list` 命令，即可查看当前运行中的节点。如下图所示。



• roscd 命令

上面所运行的小海龟示例为 ROS 自带的包，我们可以通过使用 `roscd` 命令定位到该包所在的路径。使用方法为 `roscd 包名`，比如定位 `turtlesim` 包

```
roscd turtlesim
```

即可发现当前终端的路径已经定位到 `/opt/ros/melodic/share/turtlesim`，如下图所示。当我们自己编写功能包时，使用 `roscd` 即可快速定位到该位置，从而进行查看、编辑等操作。



• rostopic 命令

如果还没有关闭之前运行的 roscore、turtlesim_node 以及 turtle_teleop_key 节点，此时可以通过 rostopic 命令当前节点间存在的通信话题，执行如下面命令。结果如下图所示，除了 /rosout、/rosout_agg 话题外，其他三个话题为小海龟示例的通信话题。

rostopic list



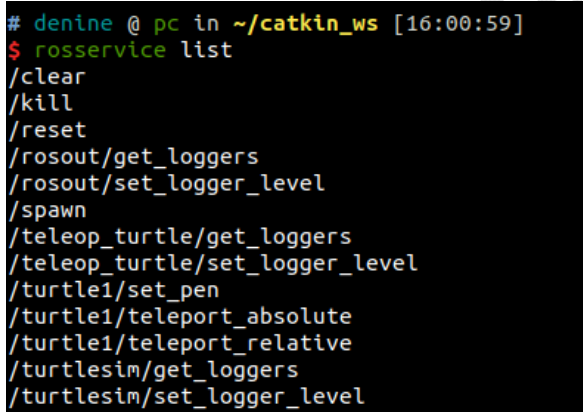
```
denine@
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

# denine @ pc in ~ [15:21:22]
$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

动手实现 1：通过手动给/turtle1/cmd_vel 话题发送消息，控制小海龟移动。

• rosservice 命令

使用 rosservice list 命令即可查看当前运行的服务，如下图所示。

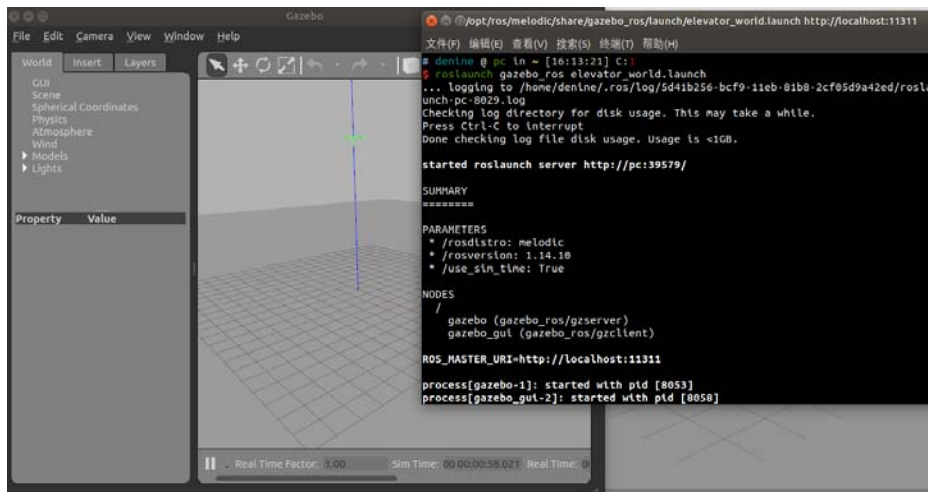


```
# denine @ pc in ~/catkin_ws [16:00:59]
$ rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

• roslaunch 命令

与前面 rosrn 命令启动一个节点相对比，roslaunch 可以一次启动多个节点，值得注意的是，在使用 rosrn 启动节点前需要执行 roscore 启动 master 节点，而执行 roslaunch 命令时会自动启动 master 节点，无需执行 roscore。以运行 gazebo_ros 包中的一个 launch 为例，执行命令（此时可以关闭之前启动的命令）

roslaunch gazebo_ros empty_world.launch



动手实现 2：当使用 `roslaunch` 启动节点后，再运行 `roscore` 会发生什么，为什么？

动手实现 3：使用 `roscd` 命令或者 `roscd` 命令快速定位到 `empty_world.launch` 所在位置并打开查看内容

实验2 熟悉ROS话题通信架构，实现话题通信过程

1、实验目的

- (1) 了解 ROS 节点间的话题通信过程
- (2) 编写节点实现话题通信

2、实验设备

硬件环境：PC

软件环境：Ubuntu18.04、ROS Melodic

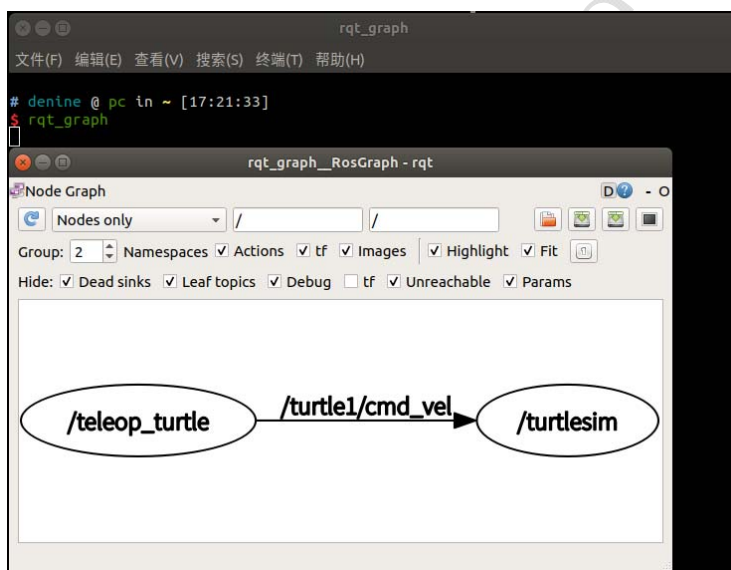
3、实验内容

(1) 了解 ROS 节点间的通信过程：

• 使用 `rqt_graph` 查看 ROS 节点关系

还是以小海龟为例，依次启动 `master` 节点、海龟节点及其控制节点，然后执行命令 `rqt_graph`

得到的结果如下图所示，可以发现我们所启动的两个节点以椭圆的形式展现，而海龟控制节点正是通过 `/turtle1/cmd_vel` 来控制海龟的移动，这就是节点间的话题通信。



• 使用 `rostopic` 查看话题信息

进一步研究 `/turtle1/cmd_vel` 话题，可以使用 `rostopic info` 命令查看该话题的信息，执行如下命令

```
rostopic info turtle1/cmd_vel
```

结果如下图所示，显示了该话题的消息类型为 `geometry_msgs/Twist`，话题的发布者 `/teleop_turtle` 节点，话题的订阅者有 `/turtlesim` 节点。

```
# denine @ pc in ~ [17:45:20]
$ rostopic info turtle1/cmd_vel
Type: geometry_msgs/Twist

Publishers:
* /teleop_turtle (http://pc:33037/)

Subscribers:
* /turtlesim (http://pc:42705/)
```

动手实现 1：利用实验一所学知识，查看 `turtle1/cmd_vel` 话题的消息类型 `Twist` 的内容是什么？

(2) 编写节点实现话题通信

• 创建功能包

首先创建一个工作空间，并使用 `catkin_make` 编译。

```
mkdir -p catkin_ws/src
```

```
cd catkin_ws
```

```
catkin_make
```

然后创建名为 `test` 的功能包，为其加上 `std_msgs`、`rospy`、`roscpp` 依赖。

```
cd src
```

```
catkin_create_pkg test std_msgs rospy roscpp
```

• 编写发布者节点 (c++版本)

进入该包内，创建 `c++` 文件并编写

```
mkdir -p test/src
```

```
cd test/src
```

```
gedit talker.cpp
```

代码如下

```
#include "ros/ros.h"
```

```
#include "std_msgs/String.h"
```

```
#include <sstream>
```

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

```
{
```

```
    // 初始化 ROS 节点
```

```
    ros::init(argc, argv, "talker");
```

```
    // 为这个进程的节点创建句柄。
```

```
    ros::NodeHandle n;
```

```
    // 告诉主节点我们将要在 chatter 话题上发布一个类型为 std_msgs/String 的消息。
```

```
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

// ros::Rate 对象能指定循环的频率为 10Hz。

```
ros::Rate loop_rate(10);

int count = 0;
while (ros::ok())
{
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    // ROS_INFO 可用来取代 printf/cout。
    ROS_INFO("%s", msg.data.c_str());
    // 向话题发送消息
    chatter_pub.publish(msg);
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
}

return 0;
}
```

• 编写接收者节点 (c++版本)

gedit listener.cpp

代码如下

```
#include "ros/ros.h"
#include "std_msgs/String.h"
```

// 这是一个回调函数，当有新消息到达 chatter 话题时它就会被调用。

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

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

```
{
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;
    // 通过主节点订阅 chatter 话题。每当有新消息到达时，ROS 将调用 chatterCallback() 函数。
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
    ros::spin();
}
```



```
    return 0;
}
```

• 构建节点

打开 test 包内的 CMakeLists.txt,

找到

```
find_package(catkin REQUIRED COMPONENTS
    roscpp
    rospy
    std_msgs
)
```

修改为

```
find_package(catkin REQUIRED COMPONENTS
    roscpp
    rospy
    std_msgs
    message_generation
)
```

找到

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

将其注释取消。

// 建立 talker 节点

```
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker test_generate_messages_cpp)
```

// 建立 listener 节点

```
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener test_generate_messages_cpp)
```

保存文件，并编译

```
cd ~/catkin_ws
catkin_make
```

将该工作空间添加到.bashrc 文件中，让系统能够知道我们所写的功能包在哪。

```
gedit ~/.bashrc
```

添加如下内容

```
source /home/denine/catkin_ws/devel/setup.bash
```

保存后重新打开终端，即可运行我们写的两个节点。

首先启动 roscore

启动 talker 节点

roslaunch test talker

启动 listener 节点

roslaunch test listener

此时，两个节点就建立了通信，如下图所示

```

roscore http://pc:11311/
[ INFO] [1621950795.378179020]: I hear
[ INFO] [1621950795.478402581]: I hear
[ INFO] [1621950795.578486014]: I hear
[ INFO] [1621950795.678499887]: I hear
[ INFO] [1621950795.778488370]: I hear
[ INFO] [1621950795.878456776]: I hear
[ INFO] [1621950795.978451180]: I hear
[ INFO] [1621950796.078478243]: I hear
[ INFO] [1621950796.178138590]: I hear
[ INFO] [1621950796.278449315]: I hear
[ INFO] [1621950796.378502435]: I hear
[ INFO] [1621950796.478507783]: I hear
[ INFO] [1621950796.578396933]: I hear
[ INFO] [1621950796.678184022]: I hear
[ INFO] [1621950796.778395423]: I hear
[ INFO] [1621950796.878167125]: I hear
[ INFO] [1621950796.978179793]: I hear
[ INFO] [1621950797.078164965]: I hear
[ INFO] [1621950797.178237967]: I hear
[ INFO] [1621950797.278488318]: I hear
[ INFO] [1621950797.378487895]: I hear
[ INFO] [1621950797.478069473]: I hear
[ INFO] [1621950795.377886865]: hello world 11
[ INFO] [1621950795.477908806]: hello world 12
[ INFO] [1621950795.577909582]: hello world 13
[ INFO] [1621950795.677922917]: hello world 14
[ INFO] [1621950795.777911360]: hello world 15
[ INFO] [1621950795.877885250]: hello world 16
[ INFO] [1621950795.977919032]: hello world 17
[ INFO] [1621950796.077913036]: hello world 18
[ INFO] [1621950796.177889211]: hello world 19
[ INFO] [1621950796.277908058]: hello world 20
[ INFO] [1621950796.377905534]: hello world 21
[ INFO] [1621950796.477912077]: hello world 22
[ INFO] [1621950796.577871244]: hello world 23
[ INFO] [1621950796.677843010]: hello world 24
[ INFO] [1621950796.777901636]: hello world 25
[ INFO] [1621950796.877839312]: hello world 26
[ INFO] [1621950796.977838523]: hello world 27
[ INFO] [1621950797.077836373]: hello world 28
[ INFO] [1621950797.177874905]: hello world 29
[ INFO] [1621950797.277931228]: hello world 30
[ INFO] [1621950797.377912519]: hello world 31
[ INFO] [1621950797.477920462]: hello world 32

```

动手实现 2：利用所学知识，编写 launch 文件，通过 roslaunch 同时启动 talker 与 listener 节点。

• 编写 Python 版本话题通信

与 C++文件不同，python 文件一般存储在 scripts 文件夹内，接下来编写 talker.py

cd test

mkdir scripts

cd scripts

gedit talker.py

填入如下代码

```
#!/usr/bin/env python
```

```
import rospy
```

```
from std_msgs.msg import String
```

```
def talker():
```

```
    pub = rospy.Publisher('chatter', String, queue_size=10)
```

```
    rospy.init_node('talker', anonymous=True)
```

```
    rate = rospy.Rate(10) # 10hz
```

```
    while not rospy.is_shutdown():
```

```
        hello_str = "hello world %s" % rospy.get_time()
```

```
        rospy.loginfo(hello_str)
```

```

        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass

```

编写 listener.py

gedit listener.py

代码如下

```
#!/usr/bin/env python
```

```
import rospy
```

```
from std_msgs.msg import String
```

```
def callback(data):
```

```
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
```

```
def listener():
```

```
    rospy.init_node('listener', anonymous=True)
```

```
    rospy.Subscriber("chatter", String, callback)
```

```
    # spin() simply keeps python from exiting until this node is stopped
```

```
    rospy.spin()
```

```
if __name__ == '__main__':
```

```
    listener()
```

修改 CmakeLists.txt

找到

```
# catkin_install_python(PROGRAMS
```

```
#   scripts/my_python_script
```

```
#   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
```

```
# )
```

修改为

```
catkin_install_python(PROGRAMS
```

```
    scripts/talker.py
```

```
    scripts/listener.py
```

```
    DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
```

```
)
```

保存退出，此时还需为 python 文件添加执行权限

```
cd scripts
```

```
chmod +x listener.py
```

```
chmod +x talker.py
```

此时，即可运行，首先运行 `roscore`，然后分别执行

```
roslaunch test talker.py
```

```
roslaunch test listener.py
```

此时发送与接收的消息为当前的系统时间。

动手实现 3：当话题通信时，使用实验一学习的命令，查看话题的信息，比如 `rostopic`、`rostopic`、`rosservice`。

实验3 熟悉ROS服务通信架构，实现服务通信过程

1、实验目的

- (1) 了解 ROS 节点间的服务通信过程
- (2) 编写节点实现服务通信

2、实验设备

硬件环境：PC

软件环境：Ubuntu18.04、ROS Melodic

3、实验内容

(1) 了解服务通信过程

还是以小海龟为例，依次启动 master 节点、海龟节点及其控制节点，然后执行命令
rosservice list

获得当前存在服务列表，/turtle1/set_pen 服务是设置小海龟路线的画笔，查看其详细信息，
执行

```
rosservice info /turtle1/set_pen
```

得到如下信息

Node: /turtlesim

URI: rosrpc://pc:53541

Type: turtlesim/SetPen

Args: r g b width off

动手实现 1：手动调用/turtle1/set_pen 服务设置画笔为红色，宽度为 10，然后控制海龟移动查看效果。

(2) 编写节点实现服务通信

• 编写 srv 消息类型 add

与之前使用标准类型不同，这里我们首先创建一个自定义的服务消息类型 add.srv

```
cd test
```

```
mkdir srv
```

```
gedit add.srv
```

添加如下代码

```
int64 a
```

```
int64 b
```

```
---
```

```
int64 sum
```

其中---上方是请求的数据，下方是响应的数据。

编辑 test 包内的 CMakeLists.txt, 找到

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)
```

修改为

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

找到

```
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )
```

修改为

```
add_service_files(
  FILES
  add.srv
)
```

保存退出, 执行 `catkin_make` 编译。

此时我们所编写的 `srv` 就可以用 `rossrv show` 查看到, 命令如下

```
rossrv show test/add
```

• 编写服务端节点 `server.cpp`

在 `test/src` 下创建 `server.cpp`, 代码如下

```
#include "ros/ros.h"
```

// 所写的 `add.srv` 文件在编译时会自动创建这个头文件

```
#include "test/add.h"
```

```
bool add(test::add::Request &req,
         test::add::Response &res)
```

```
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}
```

```

int main(int argc, char **argv)
{
    ros::init(argc, argv, "server");
    ros::NodeHandle n;
    // 创建服务 add_two_ints
    ros::ServiceServer service = n.advertiseService("add_two_ints", add);
    ROS_INFO("Ready to add two ints.");
    ros::spin();

    return 0;
}

```

• 编写客户端节点 client.cpp

在 test/src 下创建 client.cpp，代码如下

```

#include "ros/ros.h"
#include "test/add.h"
#include <cstdlib>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "client");
    if (argc != 3)
    {
        ROS_INFO("usage: client X Y");
        return 1;
    }

    ros::NodeHandle n;
    // 创建服务客户端
    ros::ServiceClient client = n.serviceClient<test::add>("add_two_ints");
    // 创建服务消息
    test::add srv;
    srv.request.a = atoll(argv[1]);
    srv.request.b = atoll(argv[2]);
    if (client.call(srv))
    {
        ROS_INFO("Sum: %ld", (long int)srv.response.sum);
    }
    else
    {
        ROS_ERROR("Failed to call service add_two_ints");
        return 1;
    }
}

```

```

    return 0;
}

```

• 修改 CMakeLists.txt

在文件末尾添加

```

add_executable(server src/server.cpp)
target_link_libraries(server ${catkin_LIBRARIES})
add_dependencies(server test_generate_messages_cpp)

```

```

add_executable(client src/client.cpp)
target_link_libraries(client ${catkin_LIBRARIES})
add_dependencies(client test_generate_messages_cpp)

```

• 修改 package.xml

除了修改 CMakeLists.txt, 此时还需修改 package.xml, 同样是在 test 包内, 打开并添加如下语句

```

<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>

```

• 编译运行

cd 至 catkin_ws 下, catkin_make

编译完成后, 如同启动话题通信一样, 启动 roscore, 然后启动 server 节点

roslaunch test server

启动 client 节点, 但 client 节点需要参数

roslaunch test client 1 2

结果如下图所示, server 收到请求并响应, client 收到响应结果。

```

# denine @ pc in ~/catkin_ws [9:58:23]
$ roslaunch test server
[INFO] [1621994311.526329130]: Ready to add two ints.
[INFO] [1621994349.945217499]: request: x=1, y=2
[INFO] [1621994349.945269043]: sending back response: [3]

# denine @ pc in ~ [9:58:43]
$ roslaunch test client 1 2
[INFO] [1621994349.945352872]: Sum: 3

# denine @ pc in ~ [9:59:10]
$

```

• 编写 Python 版本服务通信

cd 至 test 目录下, 编写服务端 server.py

gedit server.py

代码如下

```
#!/usr/bin/env python
```



```

from __future__ import print_function

from test.srv import add,addResponse
import rospy

def handle_add_two_ints(req):
    print("Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b)))
    return addResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', add, handle_add_two_ints)
    print("Ready to add two ints.")
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()

```

编写客户端 client.py
gedit client.py

代码如下

```

#!/usr/bin/env python
from __future__ import print_function
import sys
import rospy
from test.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', add)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:

```

```
print(usage())
sys.exit(1)
print("Requesting %s+%s"%(x, y))
print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

同样，

添加执行权限

```
chmod +x server.py
```

```
chmod +x client.py
```

编辑 CMakeLists.txt，

找到

```
catkin_install_python(PROGRAMS
  scripts/talker.py
  scripts/listener.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

修改为

```
catkin_install_python(PROGRAMS
  scripts/talker.py
  scripts/listener.py
  scripts/server.py
  scripts/client.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

编译

```
catkin_make
```

运行 roscore，启动节点

```
roslaunch test server.py
```

```
roslaunch test client.py 1 2
```

即可获取服务端响应结果。

动手实现 2：尝试改写服务通信的服务功能，比如拼接客户端传来的参数并返回。

动手实现 3：思考是否能够将 server 与 listener 使用 launch 启动？若可以请实现。

实验 4 了解 ROS 中常用组件：launch 启动文件、Qt 工具箱和 rviz 三维可视化环境

1、实验目的

了解 ROS 中常用组件，具体内容如下：

- (1) 学会使用 launch 启动文件同时启动多个节点。
- (2) 学会使用 Qt 工具箱中的常用工具。
- (3) 学会使用 rviz 相关插件。

2、实验设备

硬件环境：PC 机

系统环境：Ubuntu18.04、ROS Melodic

3、实验内容

(1) launch 启动文件

• launch 启动文件引入目的

当运行一个 ROS 节点时，需要打开一个新的终端运行相应命令。当系统中的节点数量不断增加时，此时会打开较多的终端，显得较为繁琐。因此引入 launch 启动文件的目的是一次性启动多个节点。

• launch 启动文件作用

同时启动多个节点，并且可以自动启动 ROS Master 节点管理器，还可以实现每个节点的各种配置，为多个节点操作提供较多遍历。

• 简单示例

launch 文件使用 XML（可扩展标记语言）形式进行描述，包含一个根元素<launch>和其他节点元素，如<node>、<arg>、<param>等等。如下 demo.launch，展示了一个完整的 launch 启动文件，其作用是启动三个乌龟节点，效果如图 3.1 所示。

```
<launch>
  <node pkg = "turtlesim" name = "turtle1" type = "turtlesim_node"/>
  <node pkg = "turtlesim" name = "turtle2" type = "turtlesim_node"/>
  <node pkg = "turtlesim" name = "turtle3" type = "turtlesim_node"/>
</launch>
```

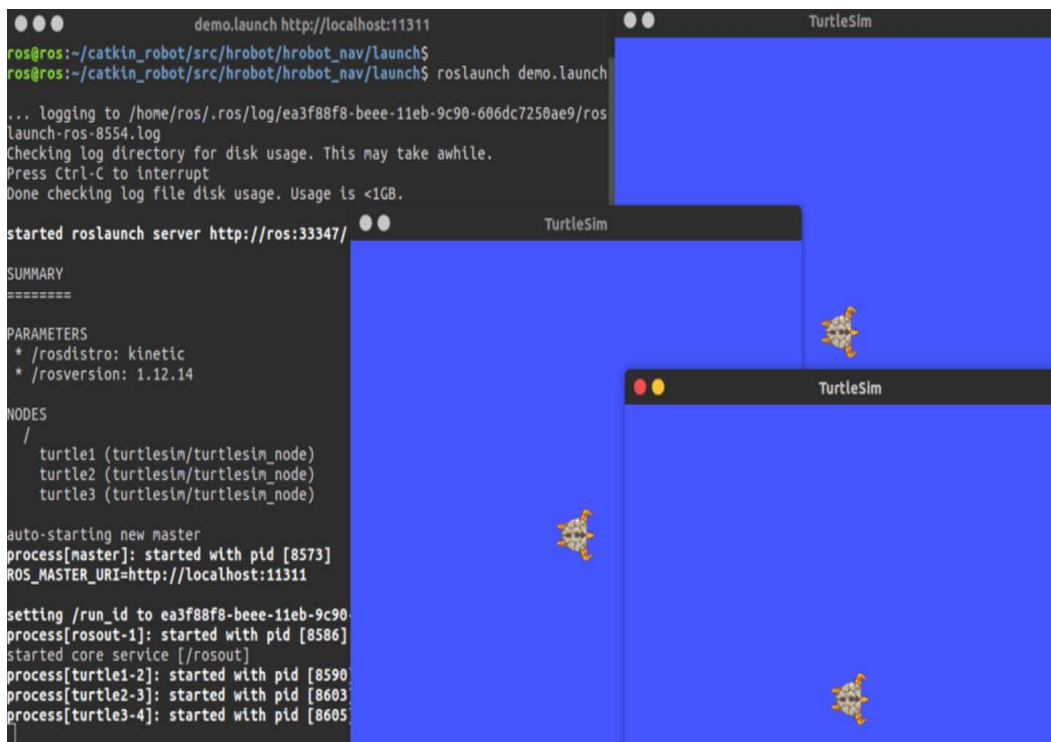


图 3.1 demo.launch 启动效果图

- **<launch>**

XML 文件必须有一个根元素，launch 启动文件中的根元素为<launch> </launch>，中间添加其他标签元素。

- **<node>**

该标签作用是启动节点，语法如下：

```
<node pkg = "package-name" type = "executable-name" name = "node-name"/>
```

pkg 定义节点所在的功能包名称；type 定义节点的可执行文件名称；name 定义节点运行的名称。此外，node 还有其他属性：

output= "screen"：将节点的标准输出打印到屏幕终端，默认输出到日志文档。

required = "true"：必要节点，当该节点终止时，launch 文件中启动的其他节点也被终止。

args = "arguments"：节点需要的输入参数。

- **<include>**

该标签主要作用是嵌套启动其他 launch 文件，语法如下：

```
<include file= "file-path"/>
```

file 定义其他 launch 文件的的路径。

(2) Qt 工具箱

- **Qt 工具箱引入目的**

为方便可视化调试和现实，ROS 提供一个 Qt 架构的后台图形工具套件 qrt_common_plugins，其中包含很多常用的工具。

- **安装命令**

```
$ sudo apt-get install ros-melodic-rqt
$ sudo apt-get install ros-melodic-rqt-common-plugins
```

- 日志输出工具: **rqt_console**

rqt_console 工具用来图像化显示和过滤 ROS 系统运行状态中的所有日志信息, 包括 info, warn, error 等级别的日志。通过如下命令, 启动该工具, 效果如图 3.2 所示。

```
$ rqt_console
或者
$ rosrun rqt_console rqt_console
```

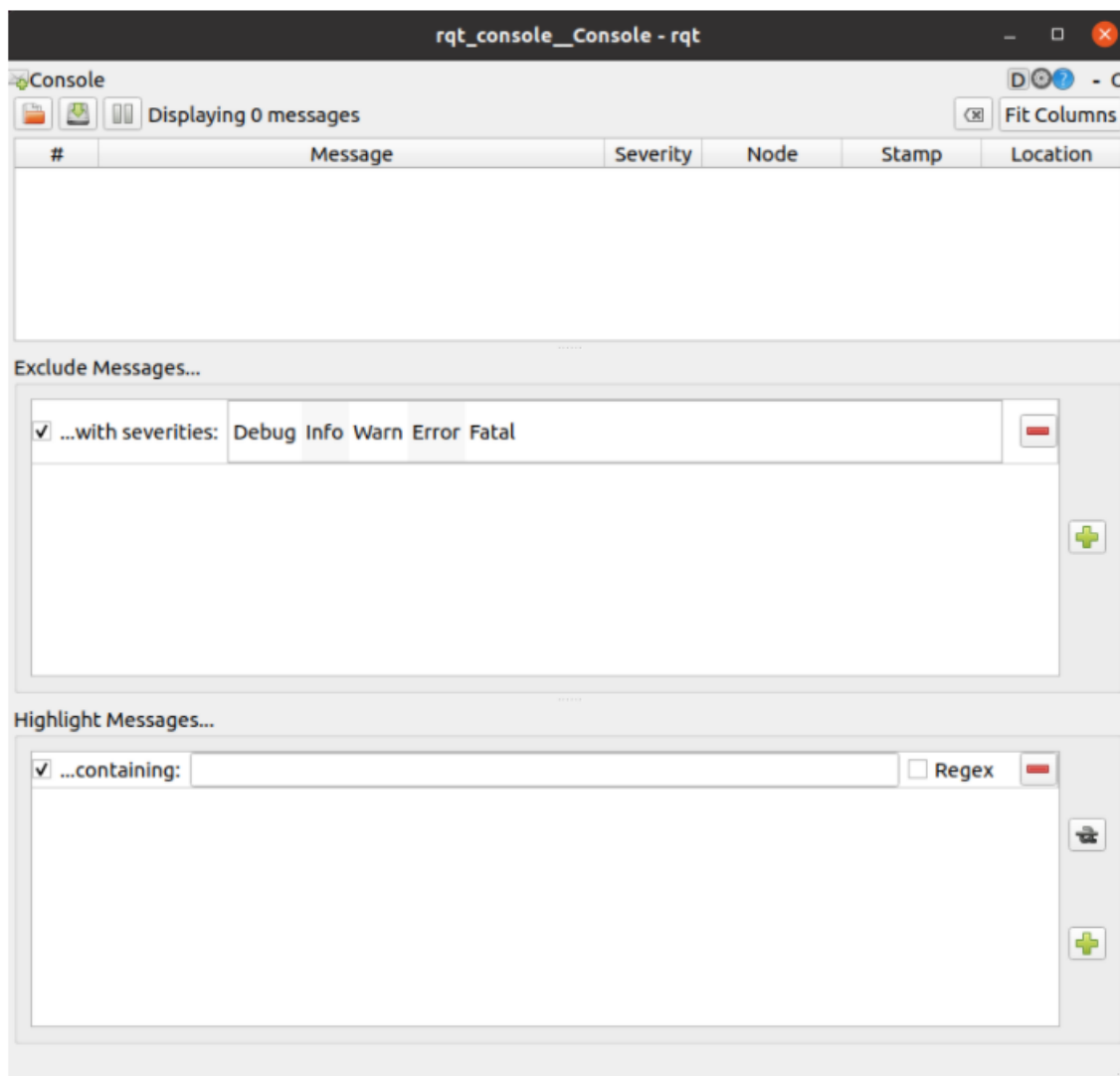


图 3.2 rqt_console 启动效果

在打开的窗口中, 可以看到三个子窗口。最上边一个窗口会显示所有 ROS 系统中的日志, 是内容最多的, 如果我们想做筛选, 可以在中间的窗口中选择日志级别, 此时中间窗口则会剔除无关级别的日志。

启动小海龟仿真器

```
$ rosrun turtlesim turtlesim_node
```

为了产生日志信息, 我们可以控制海龟撞到仿真器的边缘:

```
$ rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0},
angular: {x: 0.0,y: 0.0,z: 0.0}}"
```

当海龟撞到边缘时，`rqt_console` 中就会弹出很多警告信息，如图 3.3 所示：

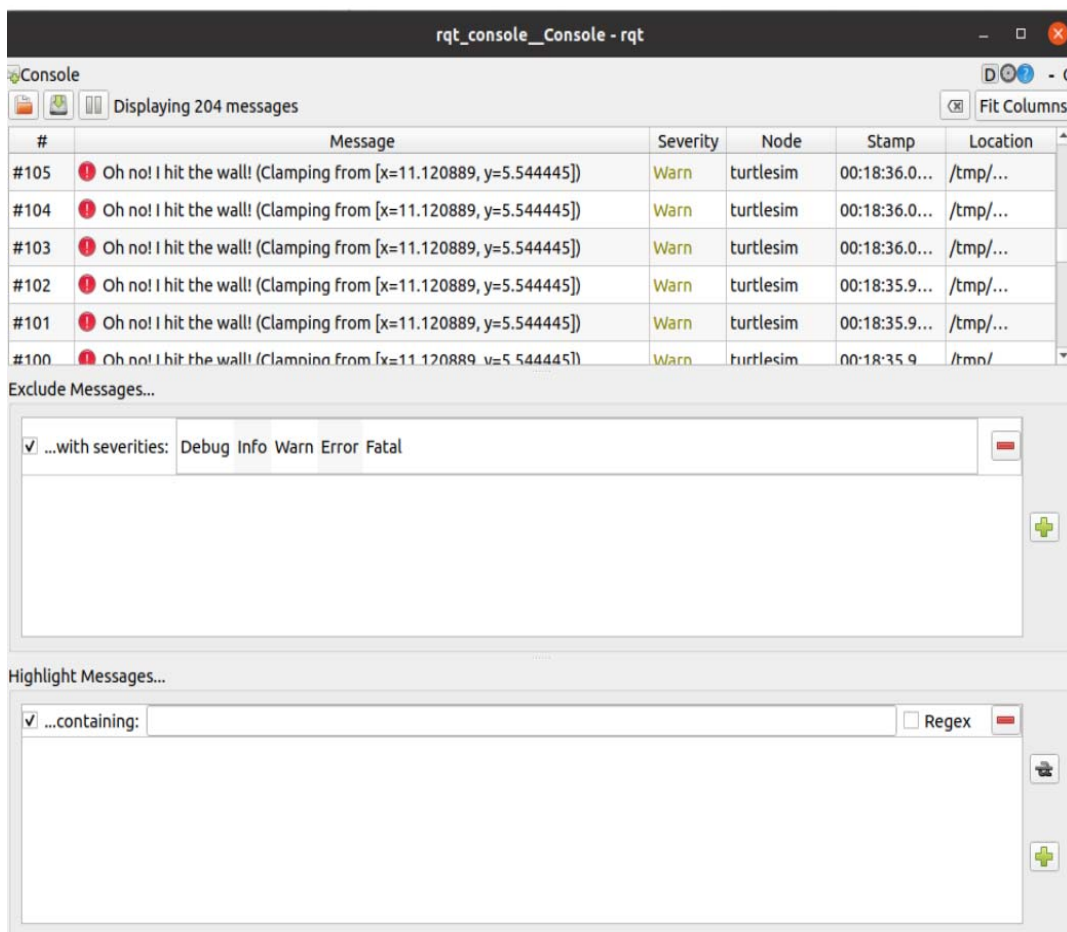


图 3.3 `rqt_console` 中弹出警告信息示例

• 节点可视化工具：`rqt_graph`

`rqt_console` 工具用来图形化显示当前 ROS 系统中的计算图，即各个节点之间的通信关系。通过如下命令，启动该工具，效果如图 3.4 所示。

```
$ rqt_graph
```

或者

```
$ rosrun rqt_graph rqt_graph
```

（下图截图是在已启动 `demo.launch` 和 `turtle_teleop_key` 节点情况下的通信状态）



图 3.4 rqt_graph 启动效果图

- 数据绘图工具: **rqt_plot**

rqt_plot 是一个二维数值曲线绘制工具, 可以将需要显示的数据在 XY 坐标系中使用曲线绘制。通过如下命令, 启动该工具, 效果如图 3.5 所示。

```
$ rqt_plot
```

或者

```
$ rosrun rqt_plot rqt_plot
```

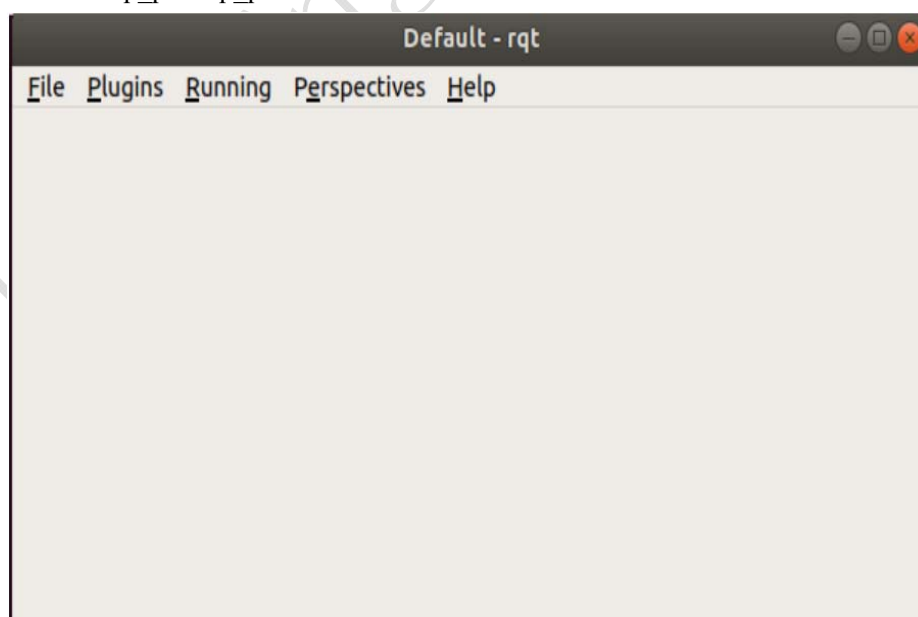


图 3.5 rqt_plot 启动效果图

以小海龟例程为例，打开小海龟节点后，在 `rqt_plot` 工具中输入话题名称，按回车键即可，如图 3.6 所示。其中红色的减号是用于选择性删除显示的话题曲线。绿色的加号用于查看所有打开的话题。

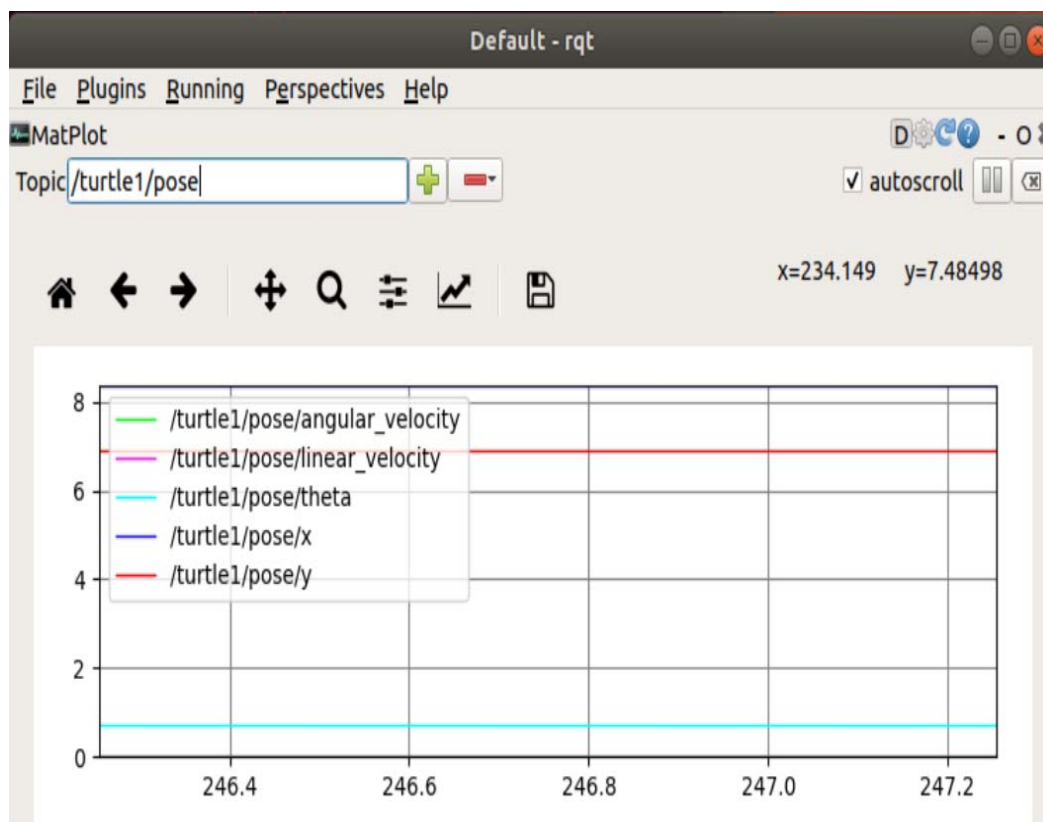


图 3.6 小海龟例程

(3) rviz 三维可视化环境

• rviz 引入目的

由于机器人模型的可视化、图像数据的可视化以及地图数据的可视化等等需求，ROS 为广大用户提供了可以显示多种数据的三维可视化平台，`rviz` 工具。

• 安装并运行 `rviz`

`Rviz` 已经继承在完整版的 ROS 系统中，实验 1 已经成功安装了桌面完整版的 ROS 系统，因此可以直接使用 `rviz` 工具。否则，使用如下命令安装：

```
$ sudo apt-get install ros-melodic-rviz
```

安装完成后，启动 `rviz`：

```
$ roscore
```

```
$ rosruncvz rviz
```

启动的 `rviz` 对应的主界面如下图 3.7 所示：

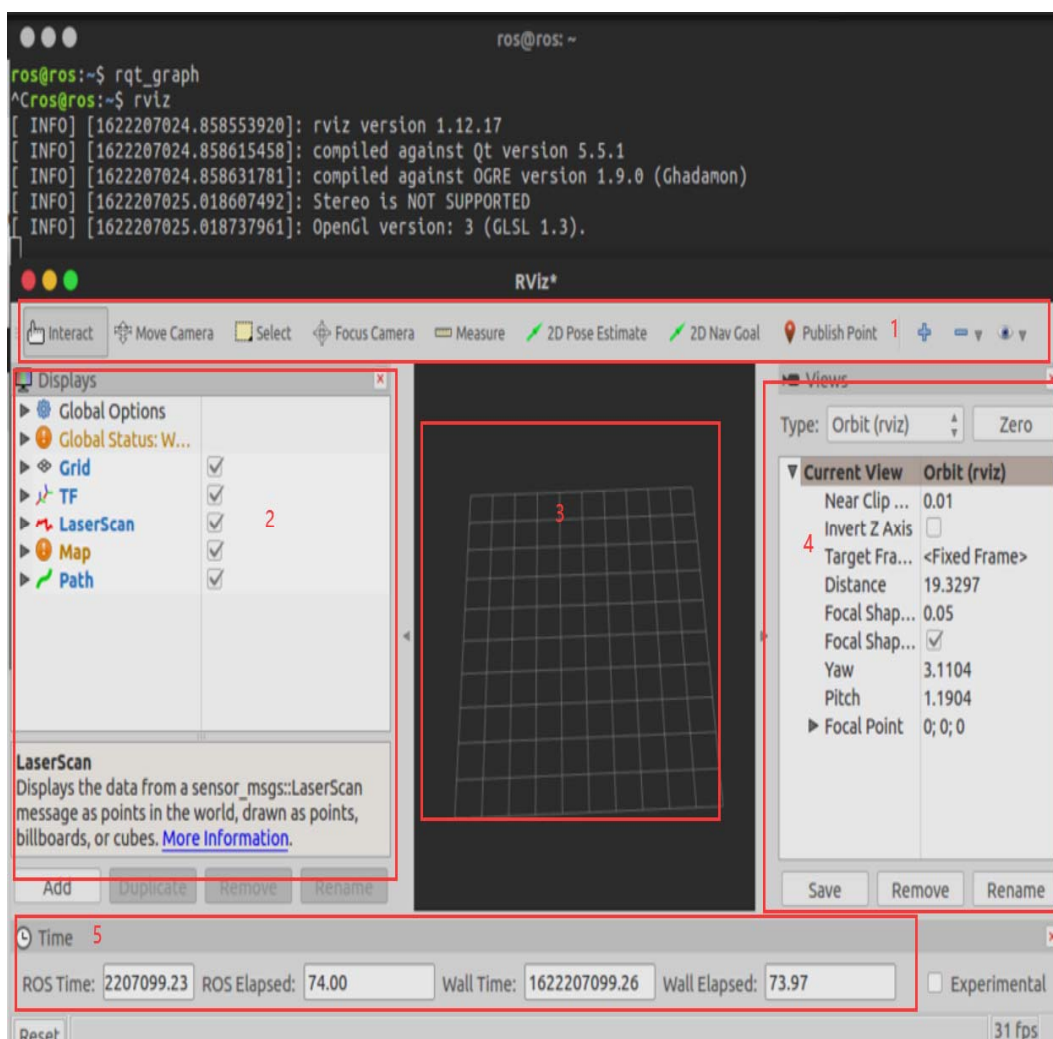


图 3.7 rviz 对应的主界面

该界面主要包含如下部分：

- (1) 工具栏：提供视角控制、目标设置、发布地点等工具；
- (2) 显示项列表：用于显示当前选择的显示插件，可以通过该列表配置每个插件的属性；
- (3) 3D 视图区：可视化显示数据，没有数据时显示黑色背景；
- (4) 视角设置区：选择多种观测视角；
- (5) 时间显示区：显示当前系统时间和 ROS 时间。

动手实现 1

使用 OpenCV 读取图像序列并通过话题在 rviz 中显示

• 给定条件

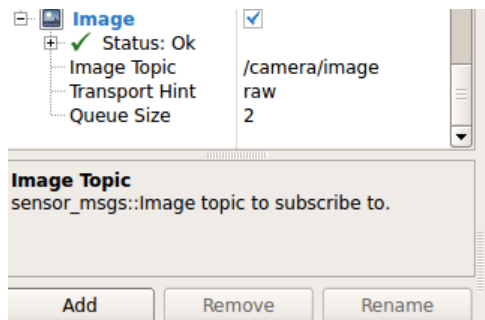
- 1) 文件夹实验4中给定图片序列文件夹 `rgb`
- 2) 文件夹实验4中的 `rgb.txt` 给出图片序列文件夹中的所有图片路径

• 核心步骤

- 1) 创建工程包，创建 `node`，确定所使用 `topic`；
- 2) 使用 OpenCV 读取循环本地 `rgb` 图像序列；
- 3) 将读取的图像转换为 `topic` 对应消息，循环发送消息；
- 4) 在 `rviz` 中调用相应插件显示 `topic` 内容。

• 核心步骤提示

- 1) 选择使用 ROS 预先定义的 topic: `camera/image`;
- 2) 调用 `cv::imread` 函数读取本地 rgb 图像;
- 3) 调用 `cv_bridge::CvImage().toImageMsg()` 将本地图像转换为 `sensor_msgs::ImagePtr` 类型的消息;
- 4) 在 rviz 中调用 image 插件显示 topic 内容:



• 代码提示

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    ... ..
    //首先将所有图片路径读到 vector<string> nameRGBs 中; 方法不唯一, 此处给出示例方法
    ifstream read;
    read.open("路径/rgb.txt".c_str());
    while(!read.eof()){
        string s;
        getline(read,s);
        if(!s.empty()){
            nameRGBs.push_back(s);
        }
    }
    //读取图片
    cv::Mat image = cv::imread("图片路径",CV_LOAD_IMAGE_COLOR);

    //将图片转换为消息
    sensor_msgs::ImagePtr msg =
        cv_bridge::CvImage(std_msgs::Header(), "bgr8",image).toImageMsg();
    ... ..
}
```

实验5 ROS简单应用：激光SLAM

1、实验目的

- (1) 了解机器人和 SLAM 之间关系
- (2) 了解激光 SLAM
- (3) 了解 gmapping

2、实验设备

硬件环境：PC

软件环境：Ubuntu18.04、ROS Melodic

3、实验内容

(1) SLAM 技术由来

近些年来，随着世界各国经济和科技的迅猛发展，目前已经形成了全球信息化和自动化的生产和生活格局。伴随其发展的机器人、无人机和自动驾驶等人工智能技术受到了科技界和企业界的越发青睐。智能机器人技术作为现代人工智能新兴产业之一，不仅改变了企业的生产方式，而且还提高了普通民众的生活质量水平。

机器人实现智能化的重要基础是能够自主移动，其不仅需要机器人获取位于环境中的方位信息，而且还需要对环境进行感知和建模。如家庭服务扫地机器人、智能无人机、无人驾驶汽车等需要对周围环境进行建图，并定位自身在地图中的方位，同时执行路径规划和碰撞检测等操作。因此，机器人要实现自主移动，需解决定位、建图和路径规划等关键问题，其中定位和建图问题主要由同时定位与地图构建（Simultaneous Localization and Mapping, SLAM）技术解决，如图 4.1 所示。SLAM 技术最早是在机器人研究领域被专家们提出，近年来随着人工智能技术的快速发展，该项技术受到了研究者的广泛关注。

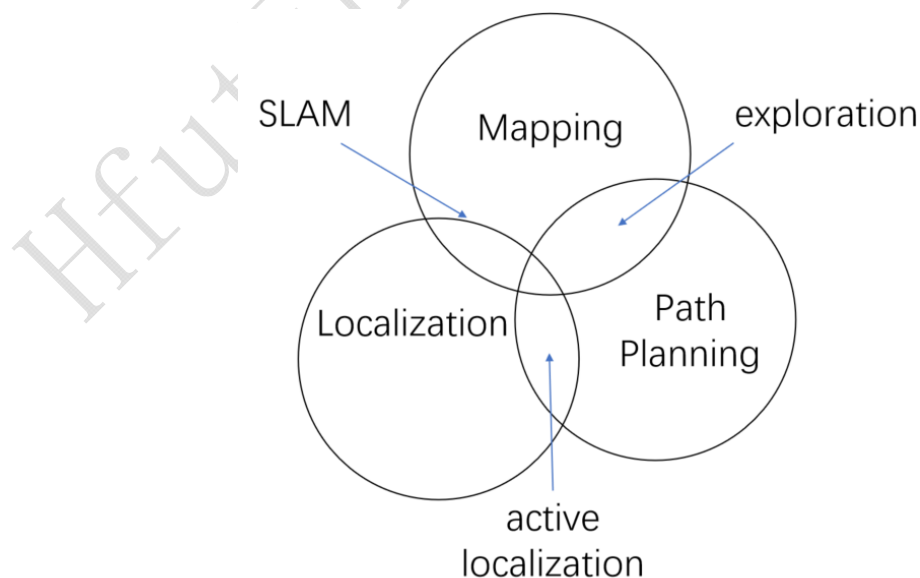


图 4.1 机器人自主移动相关技术

(2) SLAM 分类

一般地, SLAM 可根据传感器的种类分为激光 SLAM 和视觉 SLAM 两种典型方法。基于激光的 SLAM 理论研究时间较早, 技术成熟, 因此它的应用产品丰富, 如利用多维激光进行 3D 点云建图等。视觉 SLAM 搭载的传感器(单目、双目、RGB-D 等等)可以从环境中采集并提取海量丰富的纹理信息, 这些信息可以提供给计算机视觉相关任务处理(如目标检测和语义分割), 进而使系统拥有较强的语义场景辨识能力; 同时可以利用视觉信息跟踪环境中的动态物体, 如行人、车辆等, 对于在复杂动态环境中的应用至关重要。

(3) gmapping

• 介绍

gmapping 通过粒子滤波将激光距离数据转化为栅格地图。优点: 在长廊及低特征场景中建图效果好; 构建小场景地图所需的计算量较小且精度较高。缺点: 依赖里程计, 无法适用无人机及地面小车不平坦区域; 无回环; 大的场景, 粒子较多的情况下, 特别消耗资源

• 安装 gmapping (此处 ros 为 kinetic 版), 编译 gmapping 功能包

1、先安装依赖库:

```
$ sudo apt-get install libstdl1.2-dev
```

```
$ sudo apt install libstdl-image1.2-dev
```

2、安装 gmapping

```
$ sudo apt-get install ros-melodic-slam-gmapping
```

3、安装 gmapping 的 github 仓库(可选)

```
$ gitclone https://github.com/ros-perception/slam_gmapping.git
```

```
$ gitclone https://github.com/ros-perception/openslam_gmapping.git
```

• gmapping 功能包中的话题和服务

	名称	类型	描述
Topic 订阅	tf	tf/tfMessage	用于激光雷达坐标系, 基坐标系, 里程计坐标系之间的变换
	scan	sensor_msgs/LaserScan	激光雷达扫描数据
Topic 发布	map_metadata	nav_msgs/MapMetaData	发布地图Meta数据
	map	nav_msgs/OccupancyGrid	发布地图栅格数据
	~entropy	std_msgs/Float64	发布机器人姿态分布熵的估计
Service	dynamic_map	nav_msgs/GetMap	获取地图数据

• 测试 gmapping

下载用于测试的数据包:

```
https://pan.baidu.com/s/1yFdkmxZSRVZHYKA11YgFFw#list/path=%2F
```

启动 roscore: \$ roscore

运行 slam_gmapping 节点:\$ rosrn gmapping slam_gmapping scan:=base_scan

加载 bag 文件:\$ rosbag play --clock basic_localization_stage.bag

打开 rviz: \$ rviz

• 实验结果

```
wh000@wh000-Lenovo-ideapad-500S-15ISK:~$ roscore
... logging to /home/wh000/.ros/log/bf5ff3c0-cb1e-11eb-a7d3-00dbdfd81788/roslaunch-wh000-Lenovo-ideapad-500S-15ISK-5028.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://wh000-Lenovo-ideapad-500S-15ISK:39449/
ros_comm version 1.12.14

SUMMARY
=====

PARAMETERS
* /rostdistro: kinetic
* /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [5038]
ROS_MASTER_URI=http://wh000-Lenovo-ideapad-500S-15ISK:11311/
```

图一 启动 roscore

```
wh000@wh000-Lenovo-ideapad-500S-15ISK: ~
wh000@wh000-Lenovo-ideapad-500S-15ISK:~$ rosrn gmapping slam_gmapping scan:=base_scan
[ INFO] [1623461966.122290532]: Laser is mounted upwards.
-maxUrange 29.99 -maxUrange 29.99 -sigma 0.05 -kernelSize 1 -lstep 0.05 -lo
bsGain 3 -astep 0.05
-srr 0.1 -srt 0.2 -str 0.1 -stt 0.2
-linearUpdate 1 -angularUpdate 0.5 -resampleThreshold 0.5
-xmin -100 -xmax 100 -ymin -100 -ymax 100 -delta 0.05 -particles 30
[ INFO] [1623461966.163239802]: Initialization complete
update frame 0
update ld=0 ad=0
Laser Pose= 0.275 0 0
m_count 0
Registering First Scan
update frame 48
update ld=0.943924 ad=0.523599
Laser Pose= 1.03816 0.1375 0.523599
m_count 1
Average Scan Matching Score=854.499
neff= 29.8214
Registering Scans:Done
update frame 70
update ld=1.01435 ad=0.418879
Laser Pose= 1.49065 0.950595 0.942478
```

图2 启动 gmapping


```

wh000@wh000-Lenovo-ideapad-500S-15ISK: ~
wh000@wh000-Lenovo-ideapad-500S-15ISK:~$ rosbag play --clock basic_localization_
stage.bag
[ INFO] [1623461965.814565485]: Opening basic_localization_stage.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
[DELAYED] Bag Time:      34.600000    Duration: 0.000000 / 1239670987.745222  D
[RUNNING] Bag Time:      34.600000    Duration: 0.000000 / 1239670987.745222
[RUNNING] Bag Time:      34.600000    Duration: 0.000000 / 1239670987.745222
[RUNNING] Bag Time:      34.693070    Duration: 0.093070 / 1239670987.745222
[RUNNING] Bag Time:      34.793905    Duration: 0.193905 / 1239670987.745222
[RUNNING] Bag Time:      34.894632    Duration: 0.294632 / 1239670987.745222
[RUNNING] Bag Time:      34.995391    Duration: 0.395391 / 1239670987.745222
[RUNNING] Bag Time:      35.096123    Duration: 0.496123 / 1239670987.745222
[RUNNING] Bag Time:      35.196956    Duration: 0.596956 / 1239670987.745222
[RUNNING] Bag Time:      35.298569    Duration: 0.698569 / 1239670987.745222
[RUNNING] Bag Time:      35.399650    Duration: 0.799650 / 1239670987.745222
[RUNNING] Bag Time:      35.490983    Duration: 0.890983 / 1239670987.745222
[RUNNING] Bag Time:      35.592995    Duration: 0.992995 / 1239670987.745222
[RUNNING] Bag Time:      35.694625    Duration: 1.094625 / 1239670987.745222
[RUNNING] Bag Time:      35.796376    Duration: 1.196376 / 1239670987.745222
[RUNNING] Bag Time:      35.898309    Duration: 1.298309 / 1239670987.745222
[RUNNING] Bag Time:      35.999803    Duration: 1.399803 / 1239670987.745222

```

图 3 加载 bag 文件

```

wh000@wh000-Lenovo-ideapad-500S-15ISK: ~
wh000@wh000-Lenovo-ideapad-500S-15ISK:~$ rviz
[ INFO] [1623461975.647009807]: rviz version 1.12.17
[ INFO] [1623461975.647093016]: compiled against Qt version 5.5.1
[ INFO] [1623461975.647109864]: compiled against OGRE version 1.9.0 (Ghadamon)
[ INFO] [1623461976.857828076]: Stereo is NOT SUPPORTED
[ INFO] [1623461976.857927274]: OpenGL version: 4.5 (GLSL 4.5).
[ INFO] [1623461977.768815745]: Creating 1 swatches

```

图 4 启动 rviz

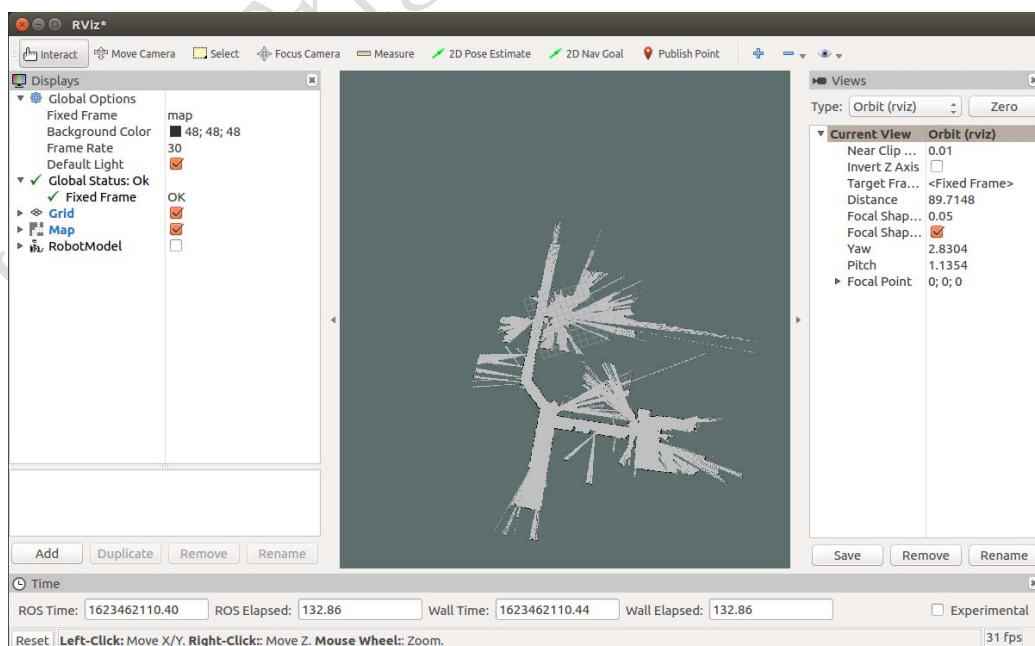


图 5 建图结果

动手实现 1:

下载教学包:

`git clone https://github.com/DroidAITech/ROS-Academy-for-Beginners.git` , 然后利用 `gmapping` 建立环境地图并且录屏保存。

HfutEngine电子文档

实验6 ROS简单应用：视觉SLAM

实验目的

- (1) 了解视觉 SLAM
- (2) 了解 ORB-SLAM

2、实验设备

硬件环境：PC

软件环境：Ubuntu18.04、ROS Melodic

3、实验内容

(1) 视觉 SLAM

视觉 SLAM 主要是基于相机来完成环境的感知工作，相对而言，相机成本较低，容易放到硬件上，且图像信息丰富，因此视觉 SLAM 也备受关注。目前，视觉 SLAM 可分为单目、双目（多目）、RGBD 这三类，另还有鱼眼、全景等特殊相机，但目前在研究和产品中还属于少数，此外，结合惯性测量器件（Inertial Measurement Unit, IMU）的视觉 SLAM 也是现在研究热点之一。

(2) SLAM 分类

单目相机 SLAM 简称 MonoSLAM，仅用一支摄像头就能完成 SLAM。其最大的优点是传感器简单且成本低廉，但不能确切的得到图像深度。双目相机与单目不同的是，立体视觉既可以在运动时估计深度，亦可在静止时估计，消除了单目视觉的许多麻烦。不过，双目或多目相机配置与标定均较为复杂，其深度量程也随双目的基线与分辨率限制。RGBD 相机是 2010 年左右开始兴起的一种相机，它最大的特点是可以通过红外结构光或 TOF 原理，直接测出图像中各像素离相机的距离。因此，它比传统相机能够提供更丰富的信息，也不必像单目或双目那样费时费力地计算深度。

(3) ORB-SLAM

• 介绍

ORB-SLAM 是一种基于 ORB 特征的三维定位与地图构建算法（SLAM）[1]。该算法由 Raul Mur-Artal, J. M. M. Montiel 和 Juan D. Tardos 于 2015 年发表在 IEEE Transactions on Robotics。ORB-SLAM 基于 PTAM 架构，增加了地图初始化和闭环检测的功能，优化了关键帧选取和地图构建的方法，在处理速度、追踪效果和地图精度上都取得了不错的效果。

ORB-SLAM 算法的一大特点是在所有步骤统一使用图像的 ORB 特征。ORB 特征是一种非常快速的特征提取方法，具有旋转不变性，并可以利用金字塔构建出尺度不变性。使用统一的 ORB 特征有助于 SLAM 算法在特征提取与追踪、关键帧选取、三维重建、闭环检测等步骤具有内生的一致性。

• 优缺点

①优点：一个代码构造优秀的视觉 SLAM 系统，非常适合移植到实际项目；采用 g2o 作为后端优化工具，能有效地减少对特征点位置和自身位姿的估计误差；采用 DBOW 减少了

寻找特征的计算量，同时回环匹配和重定位效果较好。可以使用开源代码，并且还支持使用 ROS。

②缺点：构建出的地图是稀疏点云图，只保留了图像中特征点的一部分作为关键点，固定在空间中进行定位，很难描绘地图中的障碍物的存在；初始化时最好保持低速运动，对准特征和几何纹理丰富的物体；旋转时比较容易丢帧，特别是对于纯旋转，对噪声敏感，不具备尺度不变性。

• 框架

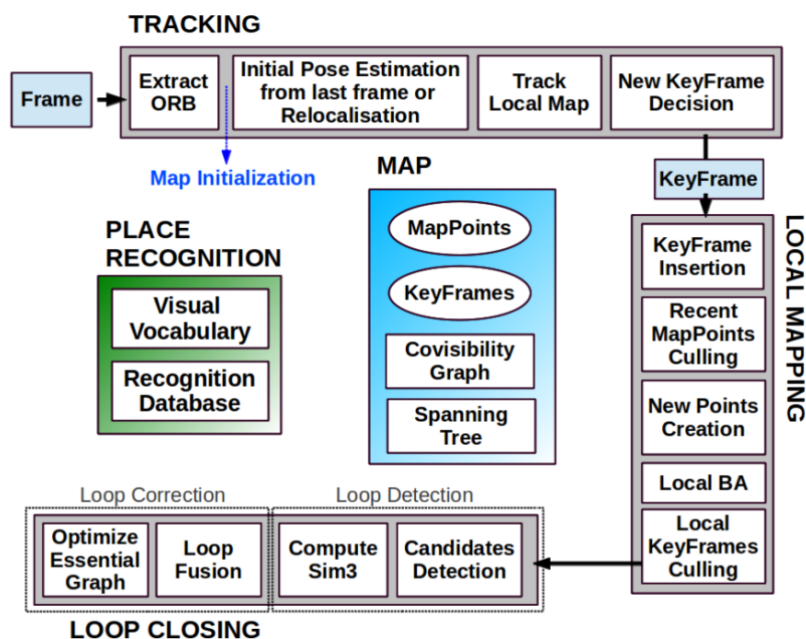


图 5 ORB-SLAM 框架示意图

①追踪

ORB 特征提取

初始姿态估计（速度估计）

姿态优化（Track local map，利用邻近的地图点寻找更多的特征匹配，优化姿态）

选取关键帧

②地图构建

加入关键帧（更新各种图）

验证最近加入的地图点（去除 Outlier）

生成新的地图点（三角法）

局部 Bundle adjustment（该关键帧和邻近关键帧，去除 Outlier）

验证关键帧（去除重复帧）

③闭环检测

选取相似帧（bag of words）

检测闭环（计算相似变换，RANSAC 计算内点数）

融合三维点，更新各种图

图优化（传导变换矩阵），更新地图所有点

• 安装

ORB-SLAM 已在 GitHub 上开源，请参照链接 https://github.com/raulmur/ORB_SLAM 配

置相关环境。

- 运行

①新开窗口启动 ROS

```
$ roscore
```

②新开窗口，在 `~/orb-slam_ws/ORB_SLAM` 下执行：

```
$ roslaunch ExampleGroovyOrNewer.launch
```

注意不同 ROS 版本，需要运行不同的指令，详见原作者 GitHub 网页（安装中给出链接地址）。这时，就会看到 Rviz 自动运行，同时出现一个黑色没有显示的窗口，因为没有数据输入。

③然后新开窗口运行数据包（先进入数据包所在文件夹）：

```
$ cd ~/orb-slam_ws
```

```
$ rosbag play --pause Example.bag
```

然后在这个窗口中通过 空格键 来控制数据包的运行与暂停，运行效果如下：

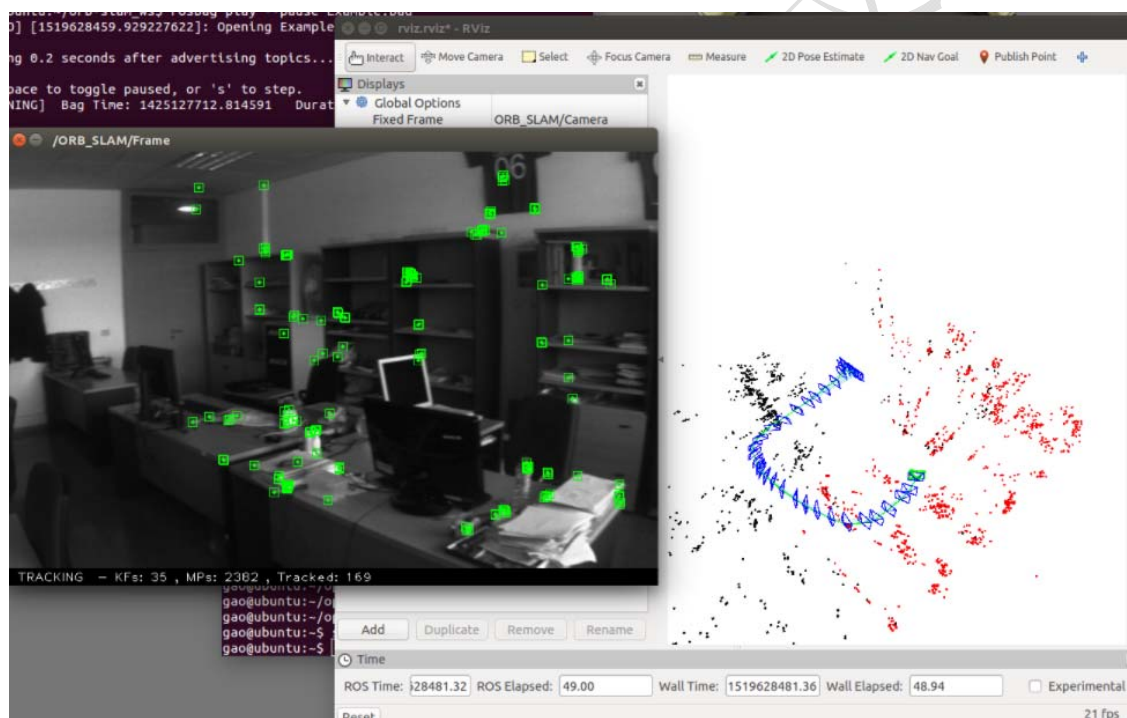


图 6 ORB-SLAM 运行示意图

动手实现 1：尝试了解 ORB-SLAM2 相关工作原理，并尝试编译并运行 ORB-SLAM2 源码。