

## 目 录

目录.....	1
一、基本命令.....	3
二、创建一个 WorkSpace.....	3
三、创建一个 Package.....	3
3.1 一个包的组成.....	3
3.2 包与 catkin WorkSpace 空间关系.....	3
3.3 创建一个 catkin Package.....	4
3.4 包的依赖.....	5
3.4.1 First-order dependencies .....	5
3.4.2.Indirect dependencies .....	5
3.5 定制自己的包.....	5
3.5.1 认识 package.xml 文件各部分含义.....	5
四、编译一个 ROS 包 .....	6
4.1 Building Packages .....	6
五、理解 ROS 节点 .....	6
5.1 熟悉如下几个概念 .....	7
5.2 Client Libraries.....	7
5.3 roscore .....	7
5.4 开始使用 rosnodet .....	8
5.5 用 rosrund.....	9
六、理解 ROS 中的 Topics .....	10
6.1 键盘遥控乌龟.....	11
6.2 ROS Topics.....	11
6.3 介绍 rostopic .....	12
6.3.1 rostopic echo .....	12
6.3.2 rostopic list .....	13
6.4 ROS Messages.....	13
6.4.1 用 rostopic type .....	14
6.4.2 用 rostopic pub 命令 .....	15
6.4.3 用 rostopic hz 命令 .....	16
七、理解 ROS 中的 Services 和 Parameters.....	18
7.1 ROS Services.....	18
7.2 用 rosservice 命令.....	18
7.2.1 rosservice list.....	19
7.2.2 rosservice type.....	19
7.2.3 rosservice call .....	20
7.3 用 rosparam.....	21
7.3.1 rosparam list .....	21
7.3.2 rosparam set 和 get 命令.....	21
7.3.3 rosparam dump 和 rosparam load 命令 .....	23
八、使用 rqt_console 和 roslaunch.....	24
8.1 安装 rqt 和 turtlesim 包。 .....	24
8.2 rqt_console 和 rqt_logger_level 的用法.....	24

8.2.1 关于 logger 的输出等级。 .....	26
8.2.2 roslaunch 的用法 .....	27
8.2.3 roslaunching .....	27
九、用 rosed 编辑文件 .....	29
9.1 rosed 的使用 .....	29
9.2 用 tab 键编辑一个包下的所有文件。 .....	30
9.3 Editor .....	30
十、创建一个 ROS msg 和 srv .....	31
10.1 msg 和 srv 介绍 .....	31
10.2 msg .....	32
10.2.1 创建一个 msg .....	32
10.2.2 使用 rosmmsg .....	33
10.3 使用 srv .....	34
10.3.1 创建一个 srv .....	34
10.3.2 使用 rossrv .....	35
10.4 msg 和 srv 的一般步骤 .....	35
10.5 一些帮助工具 .....	37
十一、写一个简单的发布者 Publisher 和订阅者 (Subscriber) (C++语言) .....	37
11.1 写一个 Publisher 节点 .....	37
11.2 写一个 Subscriber 节点。 .....	39
11.3 编译你的节点 .....	40
十二、测试发布节点和订阅节点。 .....	41
12.1 运行 Publisher .....	41
12.2 运行 Subscriber .....	42
十三、写一个简单的 Service 和 Client .....	42
13.1 写一个 Service 节点 .....	42
13.2 写一个 Client 节点 .....	43
13.3 编译你的节点 .....	44
十四、测试 Service 和 Client .....	45
14.1 运行 Service (记得先运行 roscore) .....	45
14.2 运行 Client .....	45
十五、记录数据以及回复数据 .....	46
15.1 记录数据 (创建 bag 文件) .....	46
15.1.1 记录所有发布的 topics .....	46
15.2 查看 bag 文件及 playing bag 文件 .....	47
15.3 记录数据的一部分 .....	48
十六、roswtf 的使用 .....	48

## 一、基本命令

`cd [file_name]`: 进入某一文件路径

`cd ..`: 返回上一级目录

`ls`: 列出当前路径下的所有文件

`rospack find [package_name]`: 返回所要找的包的路径

`roscd`: 将当前路径设置为某一个 ROS 包或栈的路径或其子路径

`pwd`: 显示出当前

`roscd log`: 将路径设置为 ROS 存储日志文件的路径。

`rosls [locationname[/subdir]]`: 直接将路径改为某个包下的某个子目录

TAB 键, 相当于 Eclipse 里面的 ALT+/. 但没有 ALT+/ 功能那么强大, 只能在路径唯一时才会自动出来。不会给出可选列表

## 二、创建一个 Workspace

```
$ mkdir -p ~/catkin_ws/src
```

```
$ cd ~/catkin_ws/src
```

```
$ catkin_init_workspace
```

初始化了工作空间, 创建了一个 `catkin_ws` 空间, 在该空间下有一个 `src` 文件夹。

```
$ cd ~/catkin_ws/
```

```
$ catkin_make
```

执行完该命令后, 查看当前路径, 会发现该空间除了 `src`, 又多了 `build` 和 `devel` 文件夹。这两个文件夹中是一些配置信息、编译信息等。

## 三、创建一个 Package

### 3.1 一个包的组成

必须满足三个要求:

1. 一个包里必须包含一个 `package.xml` 文件, 用于说明关于包的基本信息 (meta information) —— 相当于 Android 里德 `Manifest.xml` 文件。
2. 包必须包含一个 `CMakeLists.txt` 文件。
3. 一个文件夹里只能有一个包。这也意味着, 不会有多个包共用相同的路径。

### 3.2 包与 catkin Workspace 空间关系

一个包通常要在一个 `catkin Workspace` 中。也可以单独创建包。

包在 `Workspace` 空间中的结构是这样的:

```
workspace_folder/      -- WORKSPACE
  src/                  -- SOURCE SPACE
    CMakeLists.txt      -- 'Toplevel' CMake file, provided by catkin
  package_1/
```

```
CMakeLists.txt    -- CMakeLists.txt file for package_1
package.xml       -- Package manifest for package_1
...
package_n/
CMakeLists.txt    -- CMakeLists.txt file for package_n
package.xml       -- Package manifest for package_n
```

### 3.3 创建一个 catkin Package

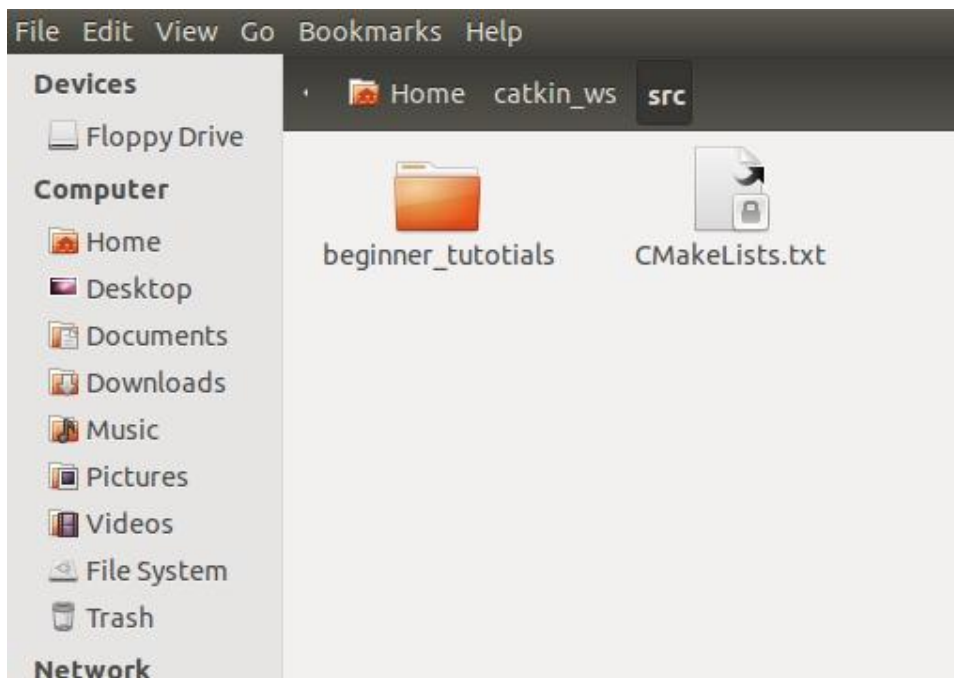
首先要先进入之前创建的一个 Workspace 的 src 目录中

```
$ cd ~/catkin_ws/src
```

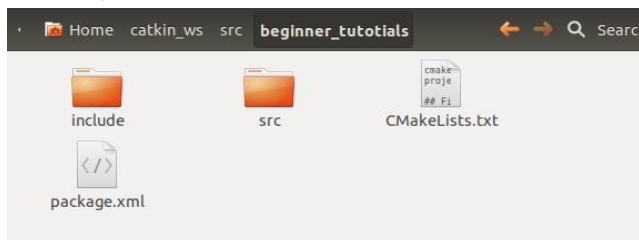
然后用 `catkin_create_pkg <package_name> [depend1] [depend2] [depend3]` 创建一个包。

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

依照上述命令，创建了一个名为 `beginner_tutorials` 的包。这个包里包含了一个 `package.xml` 和 `CMakeList.txt` 文件。进入 `src` 文件可以看到多了一个名为 `beginner_tutorials` 的包。



进入 `beginner_tutorials` 查看一下。里面有如下文件：



那么，什么是包的依赖呢？上述的命令里，有三个依赖，std\_msgs rospy roscpp，他们分别是什么含义呢？

### 3.4 包的依赖

#### 3.4.1 First-order dependencies

一个包直接依赖的文件？类？包？库？

#### 3.4.2. Indirect dependencies

包所依赖的文件可能还依赖别的文件。

用 rospack depends1 <name> 查询想要查询的类的依赖  
rospack depends <package\_name> 列出包所有依赖的对象

### 3.5 定制自己的包

#### 3.5.1 认识 package.xml 文件各部分含义

##### 1. description 标签

```
<description>this can be your description</description>
```

##### 2. maintainer 标签

```
<maintainer email=user@todo.todo>user</maintainer>
```

这个标签描述了谁是这个包的维护者。至少需要一个维护者。标签中给出了维护者的邮箱

##### 3. license 标签

```
<license>BSD</license>
```

需要选择一个开放源码许可证。开放源码许可证有：BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, and LGPLv3 等，在 ROS 里采用 BSD 开源许可证

##### 4. dependencies 标签

给出了该包依赖于哪些包或文件

默认为 buildtool\_depend、编译 build\_depend、运行 run\_depend

切换行号显示

```
12 <buildtool_depend>catkin</buildtool_depend>
13
14 <build_depend>roscpp</build_depend>
15 <build_depend>rospy</build_depend>
16 <build_depend>std_msgs</build_depend>
17
18 <run_depend>roscpp</run_depend>
19 <run_depend>rospy</run_depend>
20 <run_depend>std_msgs</run_depend>
```

#### 5. 去掉各种注释和无用的标签之后的 package.xml 文件如下：

```
<?xml version="1.0"?>
  <package>
    <name>beginner_tutorials</name>
    <version>0.1.0</version>
    <description>The beginner_tutorials package</description>

    <maintainer email="ligf@robot.nankai.edu.cn">Li Gaofeng</maintainer>
```

```

<license>BSD</license>
<url type="website">http://wiki.ros.org/beginner_tutorials</url>
<author email="ligf@robot.nankai.edu.cn">Li Gaofeng</author>

<buildtool_depend>catkin</buildtool_depend>

<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>

<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>

</package>

```

## 四、编译一个 ROS 包

### 4.1 Building Packages

#### 1. 利用 catkin\_make 命令

该命令是一个命令行工具，为标准 catkin 工作流程提供了方便。可以看做 catkin\_make 命令把 Cmake 工作流程中的 cmake 和 make 的响应结合到了一起。

```
$ catkin_make [make_targets] [-DCMAKE_VARIABLES=...]
```

#### 2. Building Your Package

```
$ cd ~/catkin_ws/
$ catkin_make
```

首先必须进入一个 catkin 空间，不然在进行 catkin\_make 编译时，catkin\_make 默认的编译路径是空间下的 src 目录，如果不在空间目录，则找不到 src 文件，编译不成功。

如果更改编译路径：

```
# In a catkin workspace
$ catkin_make --source my_src
$ catkin_make install --source my_src # (optionally)
```

编译之后，在工作空间中除了 src 文件夹，还有 build 和 devel 文件夹。build 是 cmake 和 make 在编译包时生成的一些设置文件。查看该文件夹会发现其中多了一个 beginner\_tutorials 的包。devel 文件夹是默认的 devel 空间路径，是安装包之前的一些必要的可执行文件和库。

## 五、理解 ROS 节点

我们会用到一个轻量级的模拟器，用下面命令安装：

```
$ sudo apt-get install ros-<distro>-ros-tutorials
```

将其中的 `distro` 替换成自己安装的 `ros` 版本的名字，比如 `ros-hydro`

## 5.1 熟悉如下几个概念

Nodes: 节点

Messages: 消息。一种数据类型，用来向一个 Topic 订阅或发布数据。

Topics: 话题

Master: 主节点

rosout: 标准输出/标准错误的输出

roscore: Master+rosout+parameter server

## 5.2 Client Libraries

ROS client library 允许节点以不同的变成语言与其他节点通信。其中有两个包：

rospy = python client library

roscpp = c++ client library

## 5.3 roscore

在运行 ROS 时应该首先运行的。运行：

```
$ roscore
```

会看到如下样式的输出：

```
• ... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/roslauch-machine_name-13039.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://machine_name:33919/
• ros_comm version 1.4.7
•
• SUMMARY
• =====
•
• PARAMETERS
•   * /rosversion
•   * /rostdistro
•
• NODES
```

- 
- auto-starting new master
- process[master]: started with pid [13054]
- ROS\_MASTER\_URI=http://machine\_name:11311/
- 
- setting /run\_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
- process[rosout-1]: started with pid [13067]
- started core service [/rosout]

## 5.4 开始使用 rosnode

好了，现在打开一个新的 **terminal**，让我们用 **roscout** 来看看 **roscout** 都做了些什么吧。

**Note:** 打开一个新的 **terminal** 会重设环境，`~/.bashrc` 文件会 **is sourced**，如果在运行像 **roscout** 这样的命令时出现问题，你需要添加以下环境设置文件到 `~/.bashrc` 文件中，或手动重新 **source** 他们。

下面的命令会列出当前处于活动状态的所有节点：

```
$ roscout list
```

如果什么都没有运行，会看到只有 **roscout** 节点

下面命令会返回关于某指定节点的信息：

```
$ roscout info /roscout
```

返回信息如下所示：

- -----  
---
- Node [/roscout]
- Publications:
- \* /roscout\_agg [roscout\_msgs/Log]
- 
- Subscriptions:
- \* /roscout [unknown type]
- 
- Services:
- \* /roscout/set\_logger\_level
- \* /roscout/get\_loggers
-



- contacting node http://machine\_name:54614/ ...
- Pid: 5092

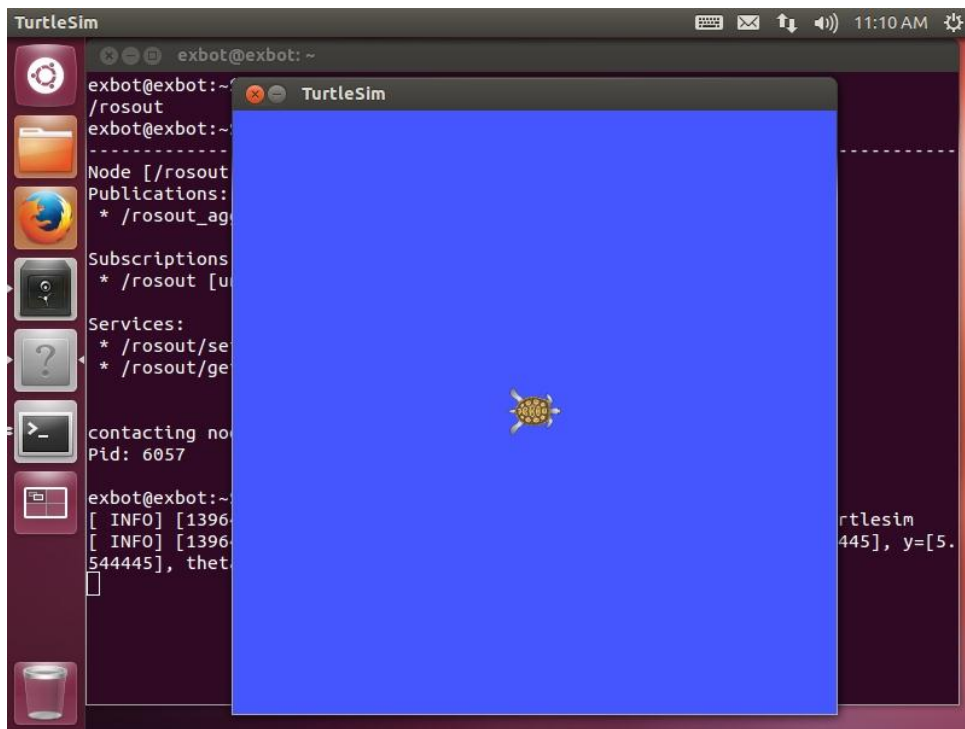
(Note: 在打开新的 terminal 时不要关掉最初的那个 terminal, 因为关掉 terminal 的话原先运行的 Master 节点也被关闭了。如果碰到这种情况, 在运行 `rostopic list` 时会发现报错: `cann't communicate with Master`。解决方法是: 运行 `rostopic` 命令开启 Master, 然后再重新打开一个 terminal)

## 5.5 用 `rostopic`.

`rostopic` 命令能够直接利用包名运行一个节点, 而不需要知道包的路径: (问题: 包与节点的关系是什么?)

```
$ rostopic [package_name] [node_name]
```

现在我们可以运行 `turtlesim` 包中的 `turtlesim_node` 试试小乌龟的仿真程序:  
我的运行效果如下:



这时候再新开一个 terminal 运行 `rostopic list` 命令会发现多了一个 `turtlesim` 节点。  
你还可以用下面命令更改节点的名字

```
$ rostopic turtlesim turtlesim_node __name:=my_turtle
```

新的效果图:



roscnode 的 ping 命令:

```
$ roscnode ping my_turtle
```

```
exbot@exbot: ~  
exbot@exbot:~$ roscnode ping my_turtle  
roscnode: node is [/my_turtle]  
pinging /my_turtle with a timeout of 3.0s  
xmlrpc reply from http://exbot:52558/    time=0.443935ms  
xmlrpc reply from http://exbot:52558/    time=0.730038ms  
xmlrpc reply from http://exbot:52558/    time=0.720024ms  
xmlrpc reply from http://exbot:52558/    time=0.730991ms  
xmlrpc reply from http://exbot:52558/    time=0.783920ms  
xmlrpc reply from http://exbot:52558/    time=0.725985ms  
xmlrpc reply from http://exbot:52558/    time=0.823021ms  
xmlrpc reply from http://exbot:52558/    time=0.715017ms  
xmlrpc reply from http://exbot:52558/    time=0.725031ms  
xmlrpc reply from http://exbot:52558/    time=0.777960ms  
xmlrpc reply from http://exbot:52558/    time=1.271009ms  
xmlrpc reply from http://exbot:52558/    time=0.840902ms  
xmlrpc reply from http://exbot:52558/    time=0.705004ms  
xmlrpc reply from http://exbot:52558/    time=0.965118ms  
xmlrpc reply from http://exbot:52558/    time=0.705004ms  
xmlrpc reply from http://exbot:52558/    time=0.787973ms  
xmlrpc reply from http://exbot:52558/    time=0.746012ms  
xmlrpc reply from http://exbot:52558/    time=0.714779ms  
xmlrpc reply from http://exbot:52558/    time=0.699043ms
```

## 六、理解 ROS 中的 Topics

在本节中，需要用 roscore 运行一个 Master 节点，然后用 roslaunch turtlesim turtlesim\_node 运行乌龟仿真程序。

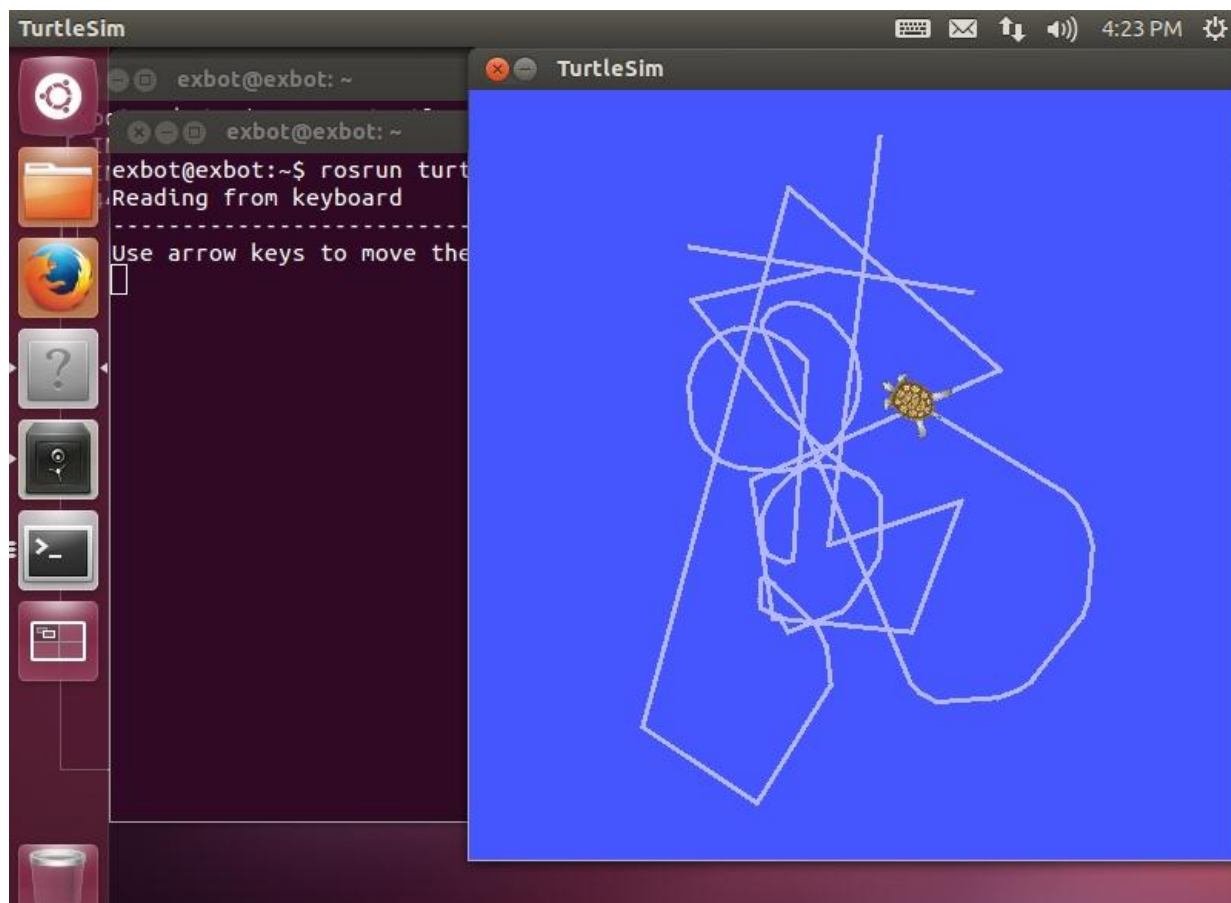
## 6.1 键盘遥控乌龟

我们已经运行起了乌龟的仿真节点，也需要驱动乌龟的程序。在新的 terminal 中运行下列命令：

```
$ rosrun turtlesim turtle_teleop_key
```

运行之后可以通过键盘的方向键控制小乌龟的移动：

需要注意的是，控制时，屏幕的焦点应该是打开 turtle\_teleop\_key 的那个 terminal。



那么在这个场景的背景究竟是怎么运行的呢？

## 6.2 ROS Topics

turtlesim\_node 和 turtle\_teleop\_key 节点通过 ROS Topics 与其他节点通信。turtle\_teleop\_key 在一个 topic 上发布键盘输入信息，而 turtlesim 订阅同一个 topic 的信息来接收键盘输入。我们可以用 rqt\_graph 查看当前正在运行的节点和 Topics。

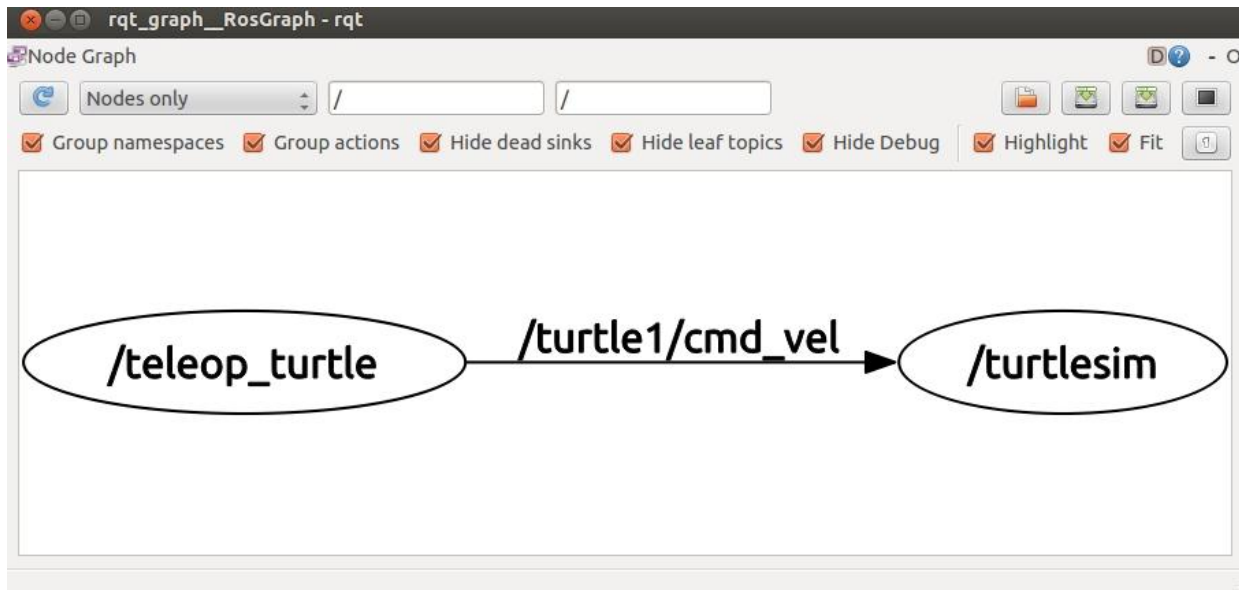
rqt\_graph 创建当前正在运行的系统的一个动态框图。rqt\_graph 是 rqt 包的一部分，如果没安装的话用下列命令安装。

- \$ sudo apt-get install ros-<distro>-rqt
- \$ sudo apt-get install ros-<distro>-rqt-common-plugins

同样， 其中的 distro 要替换成自己安装的 ROS 系统的版本名，如 hydro。

在新的 terminal 中运行下列命令，能看到如下结果：

```
$ rosrun rqt_graph rqt_graph
```



如果把鼠标放在上图中的/turtle1/cmd\_vel 上，与/turtle1/cmd\_vel 相关的节点会高亮。turtlesim\_node 和 turtle\_teleop\_key 节点是通过一个叫/turtle1/cmd\_vel 的 Topic 通信。

### 6.3 介绍 rostopic

可以用 rostopic 工具来得到关于 ROS Topics 的信息。可以用帮助命令查看命令参数

```
$ rostopic -h
```

命令列表如下：

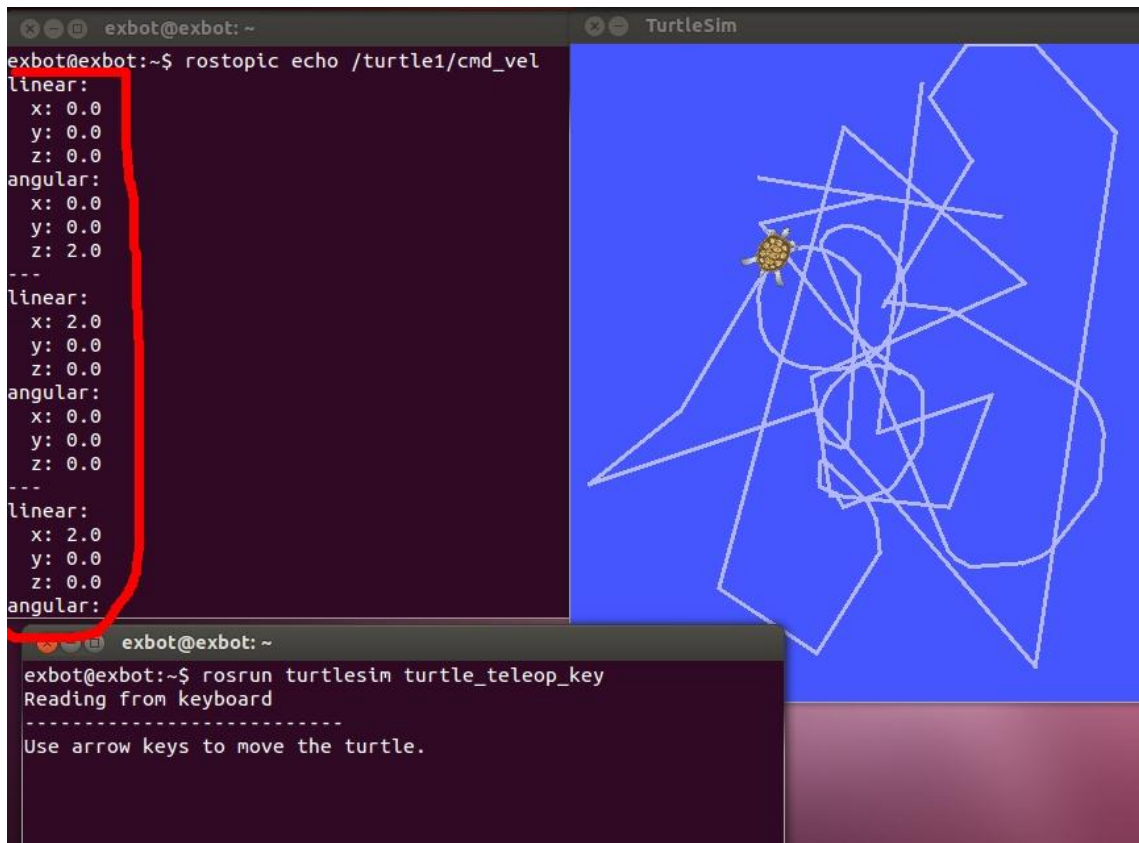
- rostopic bw display bandwidth used by topic
- rostopic echo print messages to screen
- rostopic hz display publishing rate of topic
- rostopic list print information about active topics
- rostopic pub publish data to topic
- rostopic type print topic type

#### 6.3.1 rostopic echo

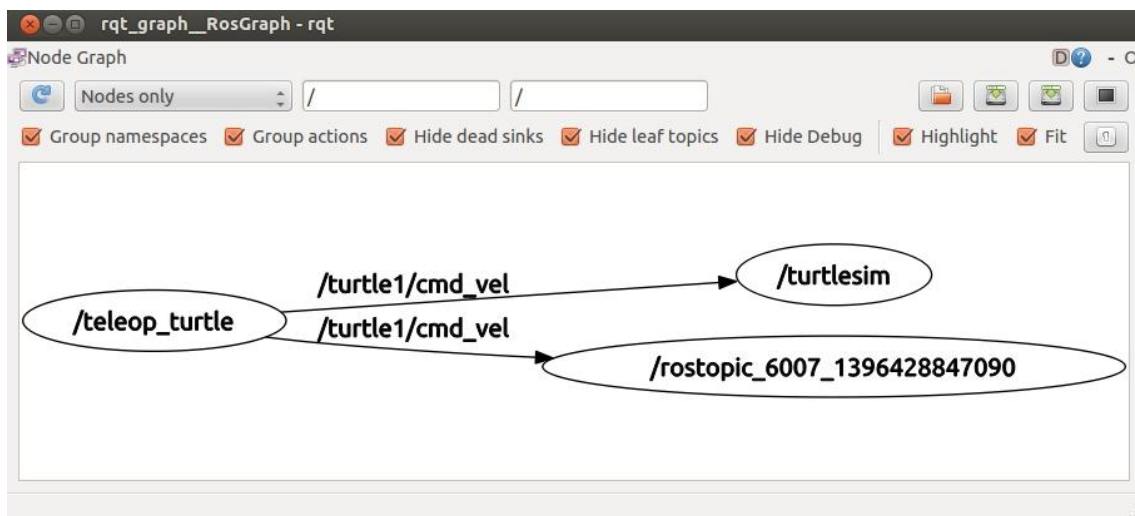
用 rostopic echo [topicName]可查看一些中间信息。

```
$ rostopic echo /turtle1/cmd_vel
```

输出结果如下图所示：



这时再看看 rqt\_graph 图（可以按左上角的刷新按钮）：



### 6.3.2 rostopic list

该命令列出当前活动的 topics。

用 -h 命令可以查看 list 命令下有哪些可选参数。

```
$ rostopic list -h
```

我们可以用 `rostopic list -v` 来列出发布和订阅的 Topics。

### 6.4 ROS Messages

话题(Topics)上的通信在节点间传递消息(Messages)时发生。为完成发布者(publisher:，如



turtle\_teleop\_key 节点)和订阅者(subscriber,如 turtlesim\_node 节点)之间的通信,发布者和订阅者必须发送和接收相同类型的消息。这意味着一个 topic 的类型是由发布到这个 topic 上的 message 的类型定义的。发布到 topic 上的 message 的类型可用 rostopic type 命令来决定。

#### 6.4.1 用 rostopic type

该命令返回任意被发布的 topic 的 message 类型。

```
rostopic type [topic]
```

对于 Hydro 和之后的版本,运行命令:

```
$ rostopic type /turtle1/cmd_vel
```

会返回如下信息:

```
geometry_msgs/Twist
```

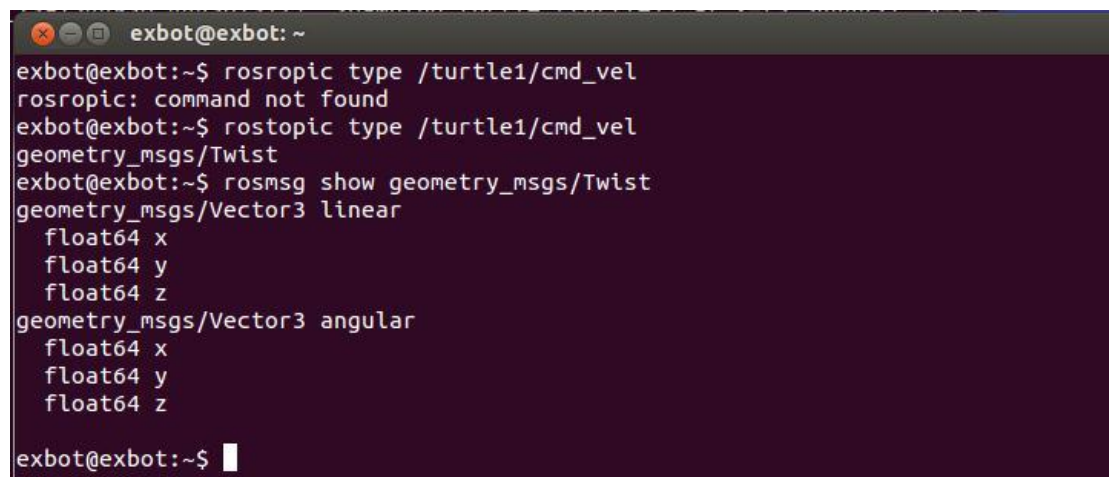
我们可以用 rosmmsg 命令查看一下这个 message 的详细信息:

```
$ rosmmsg show geometry_msgs/Twist
```

返回如下信息:

- geometry\_msgs/Vector3 linear
  - float64 x
  - float64 y
  - float64 z
- geometry\_msgs/Vector3 angular
  - float64 x
  - float64 y
  - float64 z

见下图:



```
exbot@exbot:~$ rostopic type /turtle1/cmd_vel
rostopic: command not found
exbot@exbot:~$ rostopic type /turtle1/cmd_vel
geometry_msgs/Twist
exbot@exbot:~$ rosmmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
exbot@exbot:~$
```

分析上述数据，应该分别是绕 x/y/z 轴的线速度和角速度

#### 6.4.2 用 rostopic pub 命令

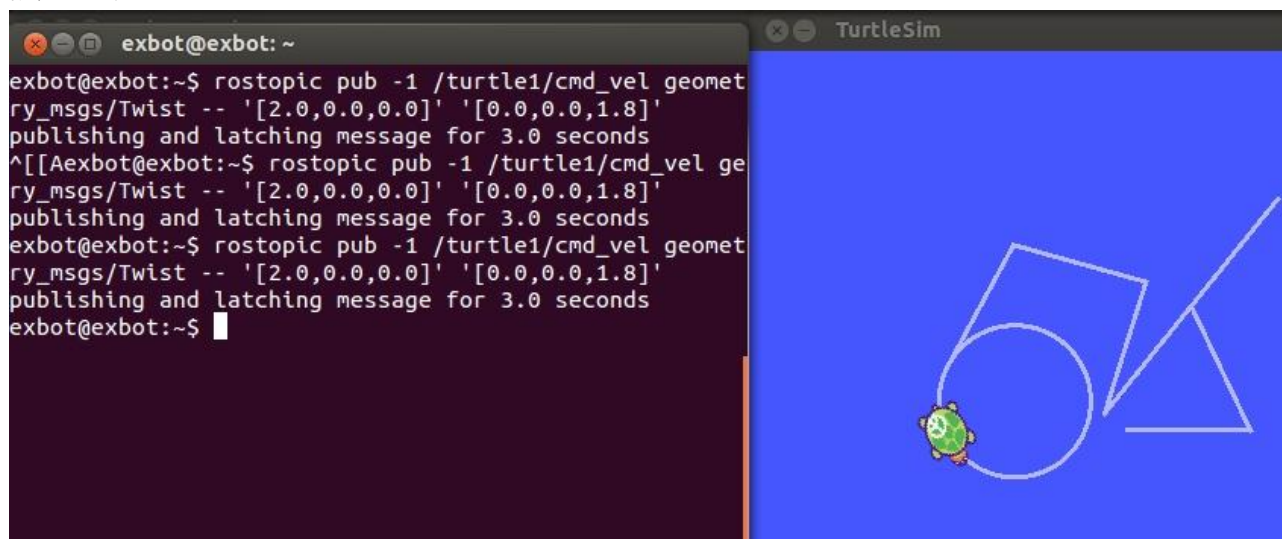
rostopic pub 命令可以向一个 topic 上发布数据。命令格式如下：

```
rostopic pub [topic] [msg_type] [args]
```

对于 Hydro 和其之后的版本，命令实例如下：

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

命令解析：该命令向 /turtle1/cmd\_vel 这个 topic 上发布了一个消息，这个消息的类型是 geometry\_msgs/Twist 类型的（分别来表征小龟的线速度和角速度），这个消息的值为“[2.0, 0.0, 0.0] [0.0, 0.0, 1.8]”。也就是说小龟在 x 轴（这个坐标系是固结在小龟身上的，以小龟向前的方向为 x 轴，垂直于小龟移动平面，从小龟腹部指向龟壳的方向为 z 轴，y 轴方向以右手坐标系标准确定）方向速度为 2.0，y 方向线速度为 0，z 轴方向速度为 0。绕 x 轴角速度为 0，绕 y 轴角速度为 0，绕 z 轴角速度为 1.8。因此，小龟就开始绕圈跑，绕圈半径为 2/1.8。参数 -1 的含义是 rostopic 只发布一个 message 然后退出。

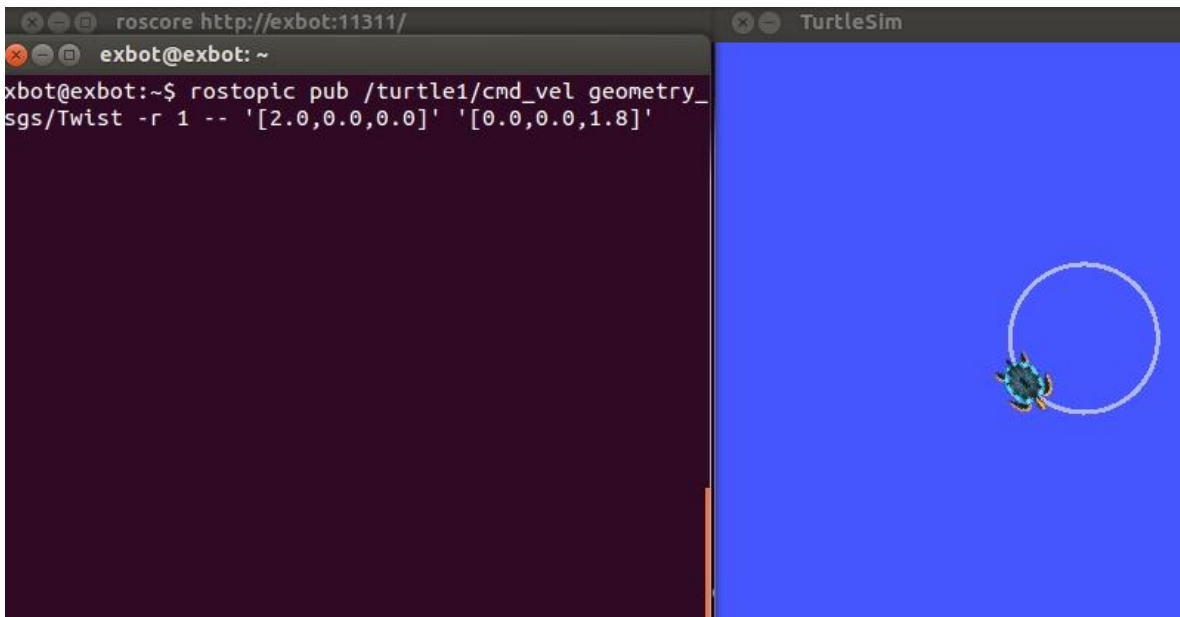


在命令行里，-称为 dash，意思是短横，单个短横称为 dash-one。含义是该参数是可缺省的。--称为 double-dash。是双短横，表示这之后的参数是不可缺省的。这是 linux 命令的规则。

执行完上面的命令后，小龟在运行一小段后就停了下来，这是因为 turtle 要求一个 1HZ 的持续不断的命令流，才能使小龟持续不断的运动，不然小龟在运动 3s 后就停了下来。我们可以用 rostopic pub -r 命令来发布一个持续的命令流：

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

可以看到，小龟在不停滴转圈。



这样就发布了一个 1HZ 的速度命令到 topic 上。

把线速度改成 3.0 之后：



把 x 方向线速度改为 0，y 方向改为 3.0 后，小龟原地转圈

可用 `rqt_graph` 和 `rostopic echo /turtle1/cmd_vel` 等命令来查看当前的一些信息。这里就不再贴图了。

### 6.4.3 用 `rostopic hz` 命令

`rostopic hz` 命令报告每一个数据发布的频率。

格式：

```
rostopic hz [topic]
```

如：

```
$ rostopic hz /turtle1/pose
```



```

exbot@exbot: ~
exbot@exbot:~$ rosrn rqt_graph
Usage: rosrn PACKAGE EXECUTABLE [ARGS]
  rosrn will locate PACKAGE and try to find
  an executable named EXECUTABLE in the PACKAGE tree.
  If it finds it, it will run it with ARGS.
exbot@exbot:~$ rqt_graph
exbot@exbot:~$ rostopic hz /turtle1/pose
subscribed to [/turtle1/pose]
average rate: 62.794
  min: 0.013s max: 0.018s std dev: 0.00070s window: 63
average rate: 62.614
  min: 0.010s max: 0.023s std dev: 0.00101s window: 125
average rate: 62.593
  min: 0.010s max: 0.023s std dev: 0.00085s window: 188
average rate: 62.573
  min: 0.010s max: 0.023s std dev: 0.00075s window: 250
average rate: 62.558
  min: 0.010s max: 0.023s std dev: 0.00070s window: 313
average rate: 62.549
  min: 0.010s max: 0.023s std dev: 0.00065s window: 376
average rate: 62.541
  min: 0.010s max: 0.023s std dev: 0.00061s window: 438
average rate: 62.535
  min: 0.010s max: 0.023s std dev: 0.00058s window: 501

```

可以看出，turtlesim 向小龟发布数据的频率是 62Hz，我们也可以用 `rostopic type` 配合 `rosmmsg show` 命令得到关于一个 topic 的更多信息：

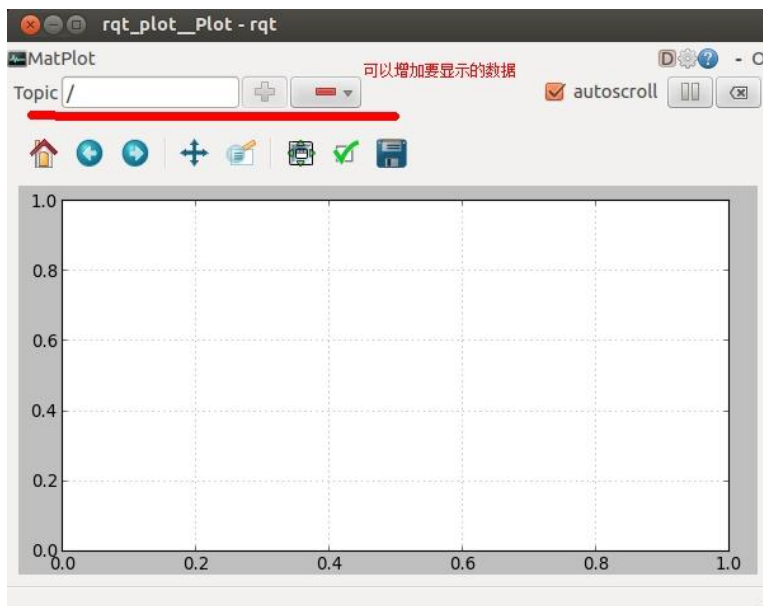
- `$ rostopic type /turtle1/cmd_vel | rosmmsg show`

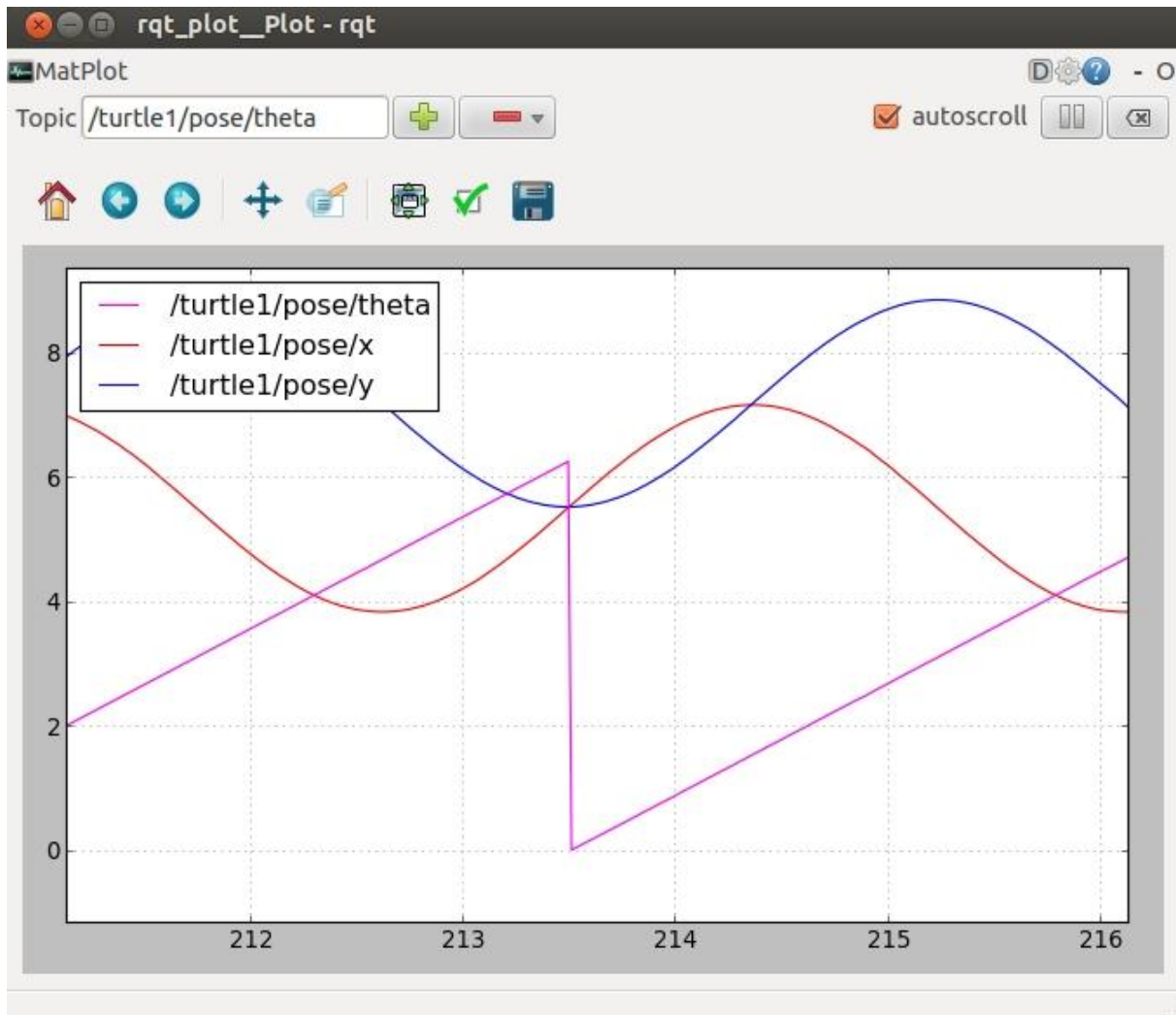
#### 6.4.4 用 `rqt_plot` 命令

`rqt_plot` 可以动态显示发布到 topic 中的数据。我们可以用 `rqt_plot` 命令画出发布到 `/turtle1/pose` 这个 Topic 上的数据。首先，用下面命令启动 `rqt_plot`。

```
$ rosrn rqt_plot rqt_plot
```

运行后会弹出一个新的窗口：





## 七、理解 ROS 中的 Services 和 Parameters

本节的学习中，需要运行 `turtlesim_node`。看看 `turtlesim` 提供了什么服务。

### 7.1 ROS Services

Services 是另一种节点间通信的方式，Services 允许节点发布一个请求并接收一个响应。

### 7.2 用 `rosservice` 命令

`rosservice` 命令可以很容易地连接到 ROS 的 client/service 框架。`rosservice` 有很多命令：

<code>rosservice list</code>	print information about active services
<code>rosservice call</code>	call the service with the provided args
<code>rosservice type</code>	print service type
<code>rosservice find</code>	find services by service type
<code>rosservice uri</code>	print service ROSRPC uri

接下来挨个讲解各个命令

### 7.2.1 rosservice list

```
$ rosservice list
```

这个命令列出了当前活动的 `turtlesim` 提供了 9 种服务：

- `/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`

上面表中，

- `/teleop_turtle/get_loggers`
- `/teleop_turtle/set_logger_level`

这两个是跟 `teleop_turtle` 相关的，如果没有开启键盘控制小龟的节点，则不会有这两项。  
而

- `/rosout/get_loggers`
- `/rosout/set_logger_level`

是跟 `rosout` 节点相关的。

### 7.2.2 rosservice type

```
rosservice type [service]
```

我们可以用这个命令看看 `clear` 服务是什么样的：

```
$ rosservice type clear
```

这个 `service` 是空的，这意味着当使用 `service call` 命令时，不带参数。（当他发布请求或接收响应时

没有传递数据)。我们可以用 `rosservice call` 命令进行 service call.

### 7.2.3 rosservice call

```
rosservice call [service] [args]
```

因为 `clear` 服务没有参数，所以我们直接输入如下命令：

```
$ rosservice call clear
```

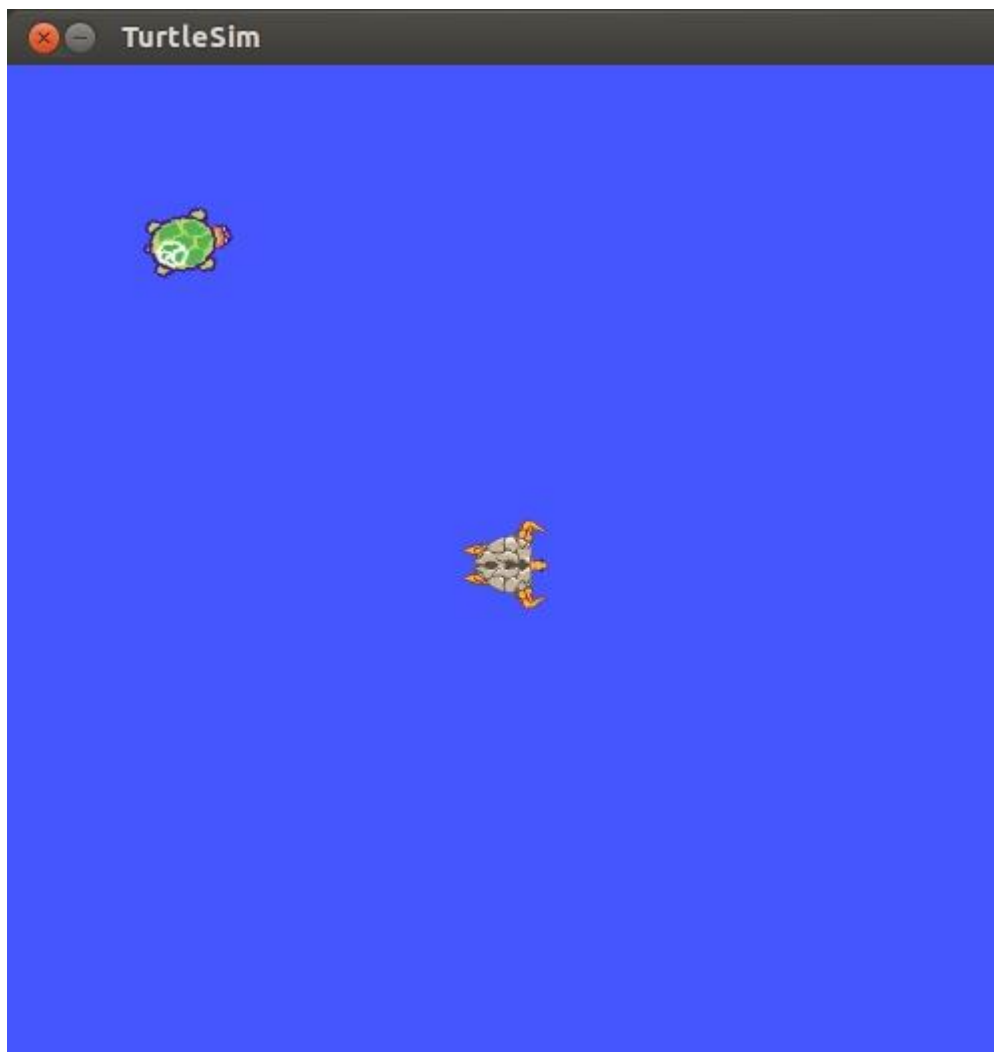
这个命令会清除 `turtlesim_node` 的背景。

如果服务是有参数的，如 `spawn`(这个单词的意思是“卵”)服务。

```
$ rosservice type spawn | rossrv show
```

这个服务让我们产生一个新的 `turtle`。然后用下面命令产生一个新的 `turtle`，给出小龟的 `x,y,theta` 和名字。

```
$ rosservice call spawn 2 2 0.2 turtle2
```



### 7.3 用 rosparam

rosparam 允许你存储和操作 ROS Parameter Server 上的数据。Parameter Server 可以存储 integers、floats、boolean、dictionaries、lists。rosparam 用 YAML 标记语言 (YAML markup language) 做句法(syntax)。在简单的情形, YAML 很接近于自然形式, 如, 1 是 integers, 1.0 是 float, one 是 String, true 是 boolean。[1, 2, 3]是整型数组, {a:b,c:d}是字典。

rosparam 命令列表如下:

rosparam set	set parameter
rosparam get	get parameter
rosparam load	load parameters from file
rosparam dump	dump parameters to file
rosparam delete	delete parameter
rosparam list	list parameter names

如上表所示, 我们可以用 rosparam list 查看一下当前都有哪些参数在 param server 上。

#### 7.3.1 rosparam list

```
$ rosparam list
```

可以看到 turtlesim 节点有三个与背景颜色相关的参数存储在 param server 上。

- /background\_b
- /background\_g
- /background\_r
- /roslaunch/uris/aqy:51932
- /run\_id

#### 7.3.2 rosparam set 和 get 命令

```
rosparam set [param_name]
rosparam get [param_name]
```

比如我们可以把背景色的红色分量设为 150.如

```
$ rosparam set background_r 150
```

这个命令改变了参数的值, 但是你会看到还没什么变化啊。接下来我们必须调用 clear service 使参数变化生效:

```
$ rosservice call clear
```



笔者按：从这里我们可以体会到 topic 和 service 之间的一点不同。topic 用于异步通信，建立 topic 和订阅及发布节点间的联系，命令发布后响应，命令结束后动作停止。比如直接我们让小龟移动，小龟 3s 之后就会停下来，如果我们想让小龟不停地运动，我们必须持续不断地向 topic 发布命令。而 service 用于同步通讯，先下达多个命令设置，然后调用 service call 服务统一响应。

这时我们可以查看一下颜色的其他分量值，比如：

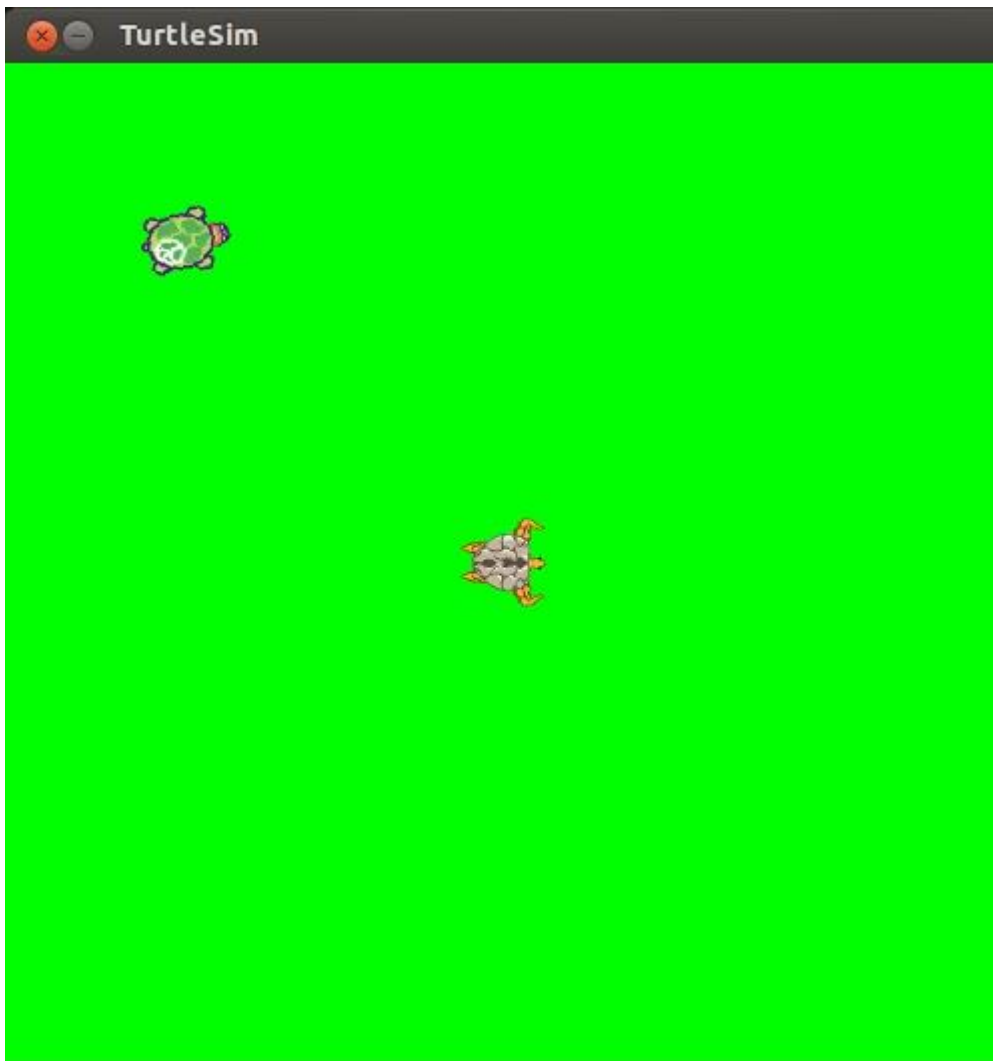
```
$ rosparam get background_g
```

```
exbot@exbot:~$ rosparam get background_g
86
exbot@exbot:~$ rosparam get background_b
255
exbot@exbot:~$ rosparam get background_r
150
exbot@exbot:~$
```

我们可以把背景设成绿色。即其他两个分量置 0，绿色分量置为 255。.

```
exbot@exbot:~$ rosparam set background_b 0
exbot@exbot:~$ rosparam set background_r 0
exbot@exbot:~$ rosparam set background_g 255
exbot@exbot:~$ rosservice call clear
```

命令执行效果：



我们也可以下面这个命令，得到 parameter server 上所有参数的值。

```
$ rosparam get /
```

这个颜色不好看，我们还是恢复成之前的：g:86,b:255,r:0。

有可能你会想要把这些参数信息存到一个文件中以备以后重新装载。可以用下面的命令方便的实现。

### 7.3.3 rosparam dump 和 rosparam load 命令

命令格式：

```
rosparam dump [file_name]
rosparam load [file_name] [namespace]
```

比如，我们可以把所有参数写入一个名为“params.yaml”的文件

```
$ rosparam dump params.yaml
```

也可以把这些 yaml 文件装载如一个 namespaces，比如一名名为 copy 的 namespaces。

```
$ rosparam load params.yaml copy
```

我们可以查看下这个数据

```
$ rosparam get copy/background_b
```

可以看到返回值是 255。

## 八、使用 `rqt_console` 和 `roslaunch`

`rqt_console` 和 `ros_logger_level` 用于 debug 调试，`roslaunch` 用于一次打开多个节点。ROS fuerte 及之前的版本不支持 `rqt`。

### 8.1 安装 `rqt` 和 `turtlesim` 包。

本节中需要用到 `rqt` 和 `turtlesim` 包，如果还没有安装，需要提前安装。安装方式：

```
$ sudo apt-get install ros-<distro>-rqt ros-<distro>-rqt-common-plugins ros-<distro>-turtlesim
```

其中<distro>替换为你的 ROS 的版本的名字，比如 Hydro。

### 8.2 `rqt_console` 和 `rqt_logger_level` 的用法

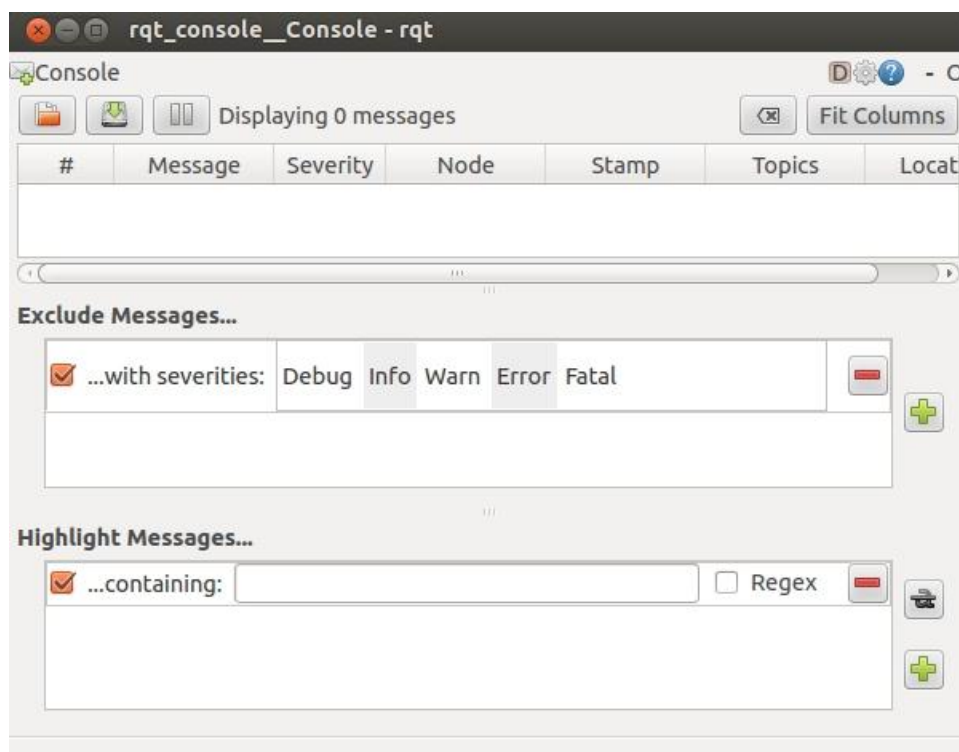
`rqt_console` 与 ROS 的日志框架相连，可以输出节点信息。`rqt_logger_level` 可以改变节点要输出的信息的级别（DEBUG，WARN，INFO 或 ERROR）。

下面我们试着用用这两个工具。

在我们打开 `turtlesim` 节点前，让我们在两个新的 terminal 中打开 `rqt_console` 和 `rqt_logger_level`。（我在用的时候已经打开 `turtlesim` 节点了，所以直接打开两个工具即可。）

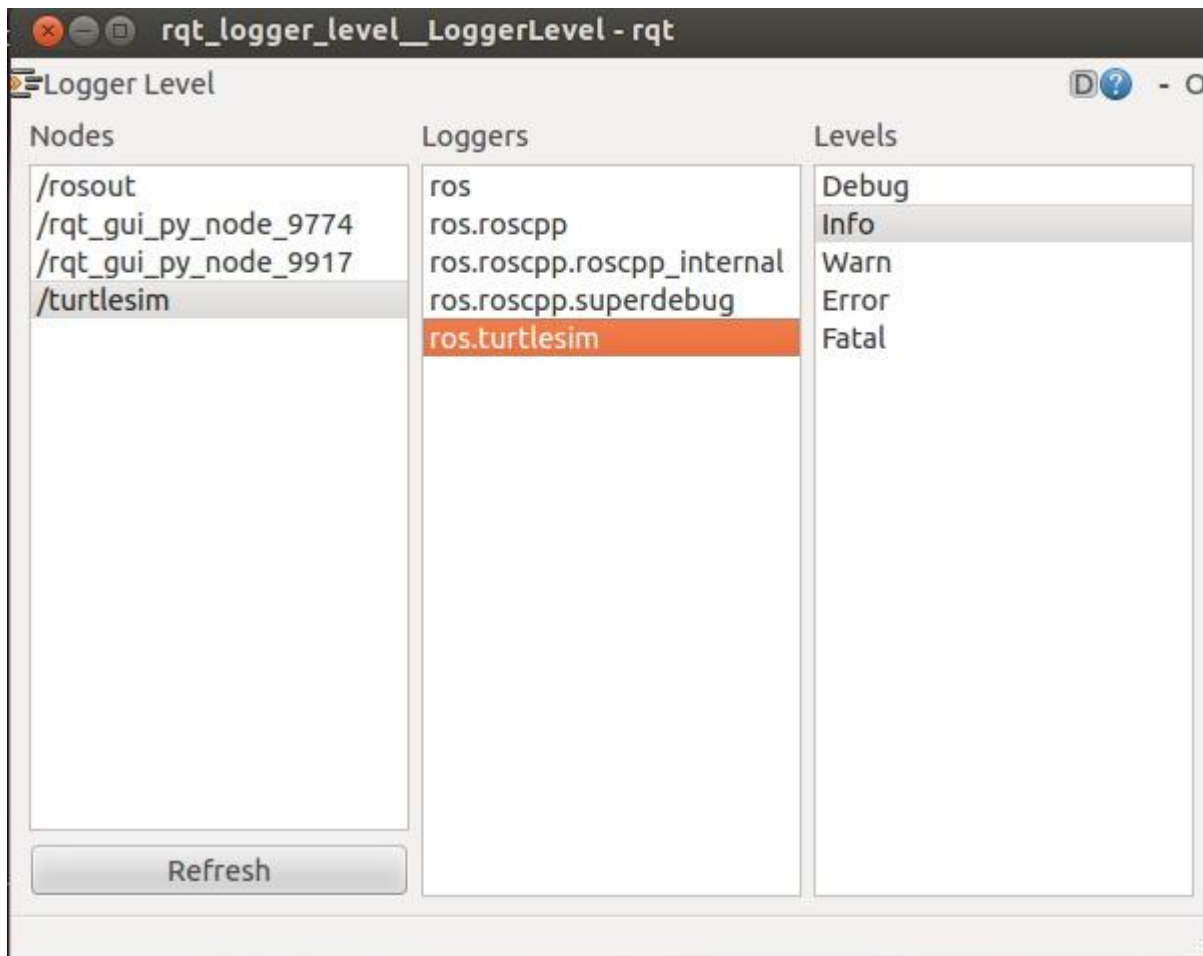
```
$ rosrun rqt_console rqt_console  
  
$ rosrun rqt_logger_level rqt_logger_level
```

然后会看到两个弹出窗口：





以及:

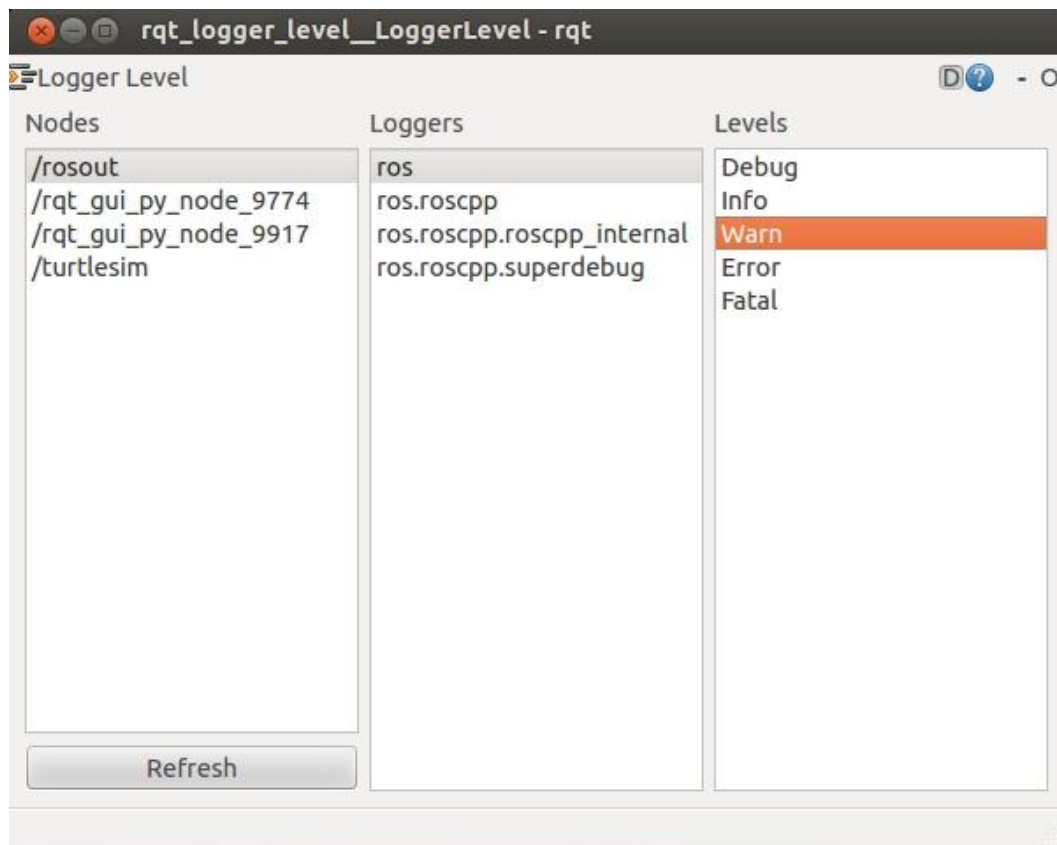


默认的输出等级是 INFO，所以你可以看到当 turtlesim 运行时，你可以看到他发布的所有信息。

笔者按：由于我是先打开的 turtlesim 后打开的 log 窗口，所以 turtlesim 启动时输出的一些信息，我们暂时看不到。但是，我们之前在运行 turtlesim 的那个 terminal 里也能看到这些输出信息。

```
exbot@exbot: ~
[ INFO] [1397544830.439797282]: Starting turtlesim with node name /turtlesim
[ INFO] [1397544830.444367312]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
^C
exbot@exbot:~$ rosrn turtlesim turtlesim_node
[ INFO] [1397549512.588480259]: Starting turtlesim with node name /turtlesim
[ INFO] [1397549512.594350993]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
exbot@exbot:~$ rosrn turtlesim turtlesim_node
[ INFO] [1397550967.412396263]: Starting turtlesim with node name /turtlesim
[ INFO] [1397550967.416777185]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
^C
exbot@exbot:~$ rosrn turtlesim turtlesim_node
[ INFO] [1397553297.310369361]: Starting turtlesim with node name /turtlesim
[ INFO] [1397553297.333769739]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
[ INFO] [1397554127.197692675]: Clearing turtlesim.
[ INFO] [1397557018.380270637]: Spawning turtle [turtle2] at x=[2.000000], y=[2.000000], theta=[0.200000]
[ INFO] [1397612082.044336203]: Clearing turtlesim.
[ INFO] [1397612676.536174893]: Clearing turtlesim.
[ INFO] [1397612875.729626005]: Clearing turtlesim.
```

我们可以把输出信息的等级，改为 warn。

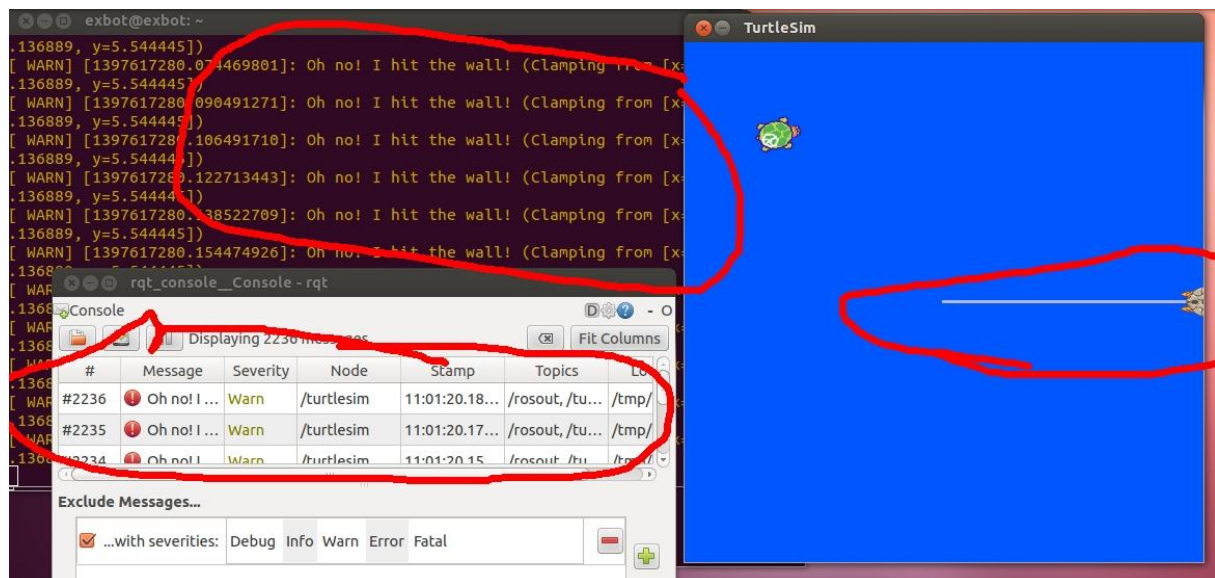


然后点 Refresh。

接下来，我们可以让小车不停向前走，直到撞到墙：

- `rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 0.0]'`

运行效果如图：



### 8.2.1 关于 logger 的输出等级。

五个输出等级从高到低，从最不详细到最详细的顺序如下：

Fatal→Error→Warn→Info→Debug

可以通过设置来改变要输出的等级。这与其他程序的调试方式都是相同的。

现在我们可以关掉我们的 `turtlesim` 节点了。让我们用 `roslaunch` 来产生多个 `turtlesim` 节点（2 个）：一个模仿节点，一个主动节点。实现让其中一个节点模仿另一个节点的动作。

### 8.2.2 roslaunch 的用法

`roslaunch` 的使用需要一个 `launch` 文件，`roslaunch` 通过 `launch` 文件中的描述打开节点。使用格式：

```
$ roslaunch [package] [filename.launch]
```

首先，我们进入之前建的 `beginner_tutorials` 包。

```
$ cd ~/catkin_ws
$ source devel/setup.bash
$ roscd beginner_tutorials
```

在 `beginner_tutorials` 目录下建一个名为“`launch`”的目录

```
$ mkdir launch
$ cd launch
```

创建一个 `launch` 文件。命名为 `turtlemimic`。内容及各行含义如下：

```
<launch>
//用 launch 标签开始 launch 文件，这样把文件类型定义为 launch 文件
<group ns="turtlesim1">
  <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
</group>

<group ns="turtlesim2">
  <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
</group>
//在一个名为 sim 的 turtlesim 节点里打开了两个 namespace 标签：turtlesim1 和 turtlesim2。这能够允许我们在开启两个仿真器时避免命名冲突
<node pkg="turtlesim" name="mimic" type="mimic">
  <remap from="input" to="turtlesim1/turtle1"/>
  <remap from="output" to="turtlesim2/turtle1"/>
</node>
//这里我们用 input 和 output 话题打开了一个跟随节点，input 和 output 重命名了 turtlesim1 和 turtlesim2，这会导致 turtlesim2 跟随 turtlesim1 的动作。
</launch>
```

### 8.2.3 roslaunching

现在用 `roslaunch` 命令运行 `launch` 文件：

```
$ roslaunch beginner_tutorials turtlemimic.launch
```

系统执行 `launch` 文件后会打开两个 `turtlesim` 节点。执行过程中产生的信息如下：

两个打开的 turtlesim 节点如下图:



打开一个新的 terminal，在其中用 rostopic 向 turtlesim1 发送命令：

```
$ rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

会看到，虽然只想 turtle1 发布了 topic 命令，两个节点同步运动。



我们可以用 rqt\_graph 来查看这时的关系，可以帮助我们更好地理解 launch 文件做了什么。

```
$ rqt_graph
```



## 九、用 rosed 编辑文件

这一讲介绍如何用 rosed 工具使文件编辑更简便。

### 9.1 rosed 的使用

rosed 是 rosbash 的一部分，这个工具让你能通过包名直接编辑包中的一个文件而不用打印包的整个路径。使用格式：

```
$ rosed [package_name] [filename]
```

比如:

```
$ rosed roscpp Logger.msg
```

这个例子展示了如何编辑 roscpp 包中的 Logger.msg 文件。  
如果上面的例子运行不成功说明你的系统没装 vim 编辑器。

```
exbot@ubuntu: ~
exbot@ubuntu:~$ rosed roscpp Logger.msg
The program 'vim' can be found in the following packages:
* vim
* vim-gnome
* vim-tiny
* vim-athena
* vim-gtk
* vim-nox
Try: sudo apt-get install <selected package>
exbot@ubuntu:~$ sudo apt-get install vim
```

## 9.2 用 tab 键编辑一个包下的所有文件。

```
$ rosed [package_name] <tab><tab>
```

上式中<tab>表示按一下 tab 键。连按两下会列出该包下的所有文件。

```
exbot@ubuntu: ~
exbot@ubuntu:~$ rosed roscpp
Empty.srv          roscpp.cmake
genmsg_cpp.py      roscppConfig.cmake
gensrv_cpp.py      roscppConfig-version.cmake
GetLoggers.srv     roscpp-msg-extras.cmake
Logger.msg         roscpp-msg-paths.cmake
msg_gen.py         SetLoggerLevel.srv
package.xml
exbot@ubuntu:~$ rosed roscpp
```

## 9.3 Editor

rosed 的 default 编辑器是 vim。而对于新手来说更友好的编辑器 nano 在安装 Ubuntu 系统时已经安装。  
你可以通过编辑 ~/.bashrc 文件更改使用哪个编辑器的设置。

```
export EDITOR='nano -w'
```

如果要把 emacs 设置成默认编辑器，你可以这样设置：

```
export EDITOR='emacs -nw'
```

NOTE: 在 .bashrc 文件中的设置只在打开新的 terminals 时生效，在已经打开的 terminal 里不会看到环境变量的改变。

可以打开一个新的 terminal 通过下面的命令查看当前默认的编辑器是哪个。

```
$ echo $EDITOR
```

```
exbot@ubuntu:~$ export EDITOR='nano -w'
exbot@ubuntu:~$ export EDITOR='emacs -nw'
exbot@ubuntu:~$ echo $EDITOR
emacs -nw
exbot@ubuntu:~$
```

## 十、创建一个 ROS msg 和 srv

这节介绍了如何创建和编译 msg、srv 文件，并介绍了 rosmmsg, rossrv 和 roscp 命令行工具。

### 10.1 msg 和 srv 介绍

**msg:** msg 文件是简单的文本文件，描述了 ROS message 的变量(fields、数据)。msg 文件的作用是让不同语言写的源码生成 messages。

**srv:** srv 文件描述了一个 service 的数据。由两个部分组成，请求部分和响应部分。

msg 文件存储在包中的 msg 目录下，srv 文件存储在包中的 srv 目录下。

msgs 是简单的文本文件，每一行都是一个变量类型和变量名。这些变量名可以是：

- int8, int16, int32, int64 (plus uint\*)
- float32, float64
- string
- time, duration
- other msg files
- variable-length array[] and fixed-length array[C]

在 ROS 中还有一种特殊的数据类型：Header。Header 包含了一个时间戳和一个笛卡尔坐标信息，通常在一个 msg 文件的头一行都有一个 Header 数据。

下面是一个 msg 文件的例子，这个 msg 文件中有一个 Header 变量，一个 String 变量，还有两个 msgs 变量：

```
Header header

string child_frame_id

geometry_msgs/PoseWithCovariance pose

geometry_msgs/TwistWithCovariance twist
```

srv 文件与 msg 文件类似，只是 srv 文件有两个部分：request 和 response。两个部分用 ‘---’ 分割。下面是一个 srv 文件例子：

```
int64 A

int64 B

---
```

```
int64 Sum
```

上面的例子中，A 和 B 是 request，Sum 是 response。

## 10.2 msg

### 10.2.1 创建一个 msg

我们可以在先前创的包里定义一个新的 msg。

```
$ cd ~/catkin_ws/src/beginner_tutorials
$ mkdir msg
$ echo "int64 num" > msg/Num.msg
```

上面的是一个最简单的例子，在 Num.msg 这个 msg 文件中只有一行，定义了一个 int64 的变量。当然，你也可以创建更多更复杂的元素。比如：

```
string first_name
string last_name
uint8 age
uint32 score
```

还有一步，我们需要确认 msg 文件已经被翻译成 C++、Python 或其他语音的源码。

还需要做如下工作：

① 打开 package.xml，添加如下两行，如果已经存在则不用重复添加：

```
<build_depend>message_generation</build_depend>

<run_depend>message_runtime</run_depend>
```

② 打开包的 CMakeLists.txt 文件。将 message\_generation 依赖加入 find\_package 中，这样允许你生成 messages。find\_package 已经在 CMakeLists.txt 中，你所要做的只是在 find\_package 中加上 message\_generation 即可。像下图这样：



```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  message_generation 添加上这一句
)
```

也许你会发现，有时不调用 find\_package 中的所有依赖也能编译成功你的工程 project。这是因为 catkin 把所有你的 projects 编译成一个 find\_package，有可能之前的 project 调用了 find\_package，那你的设置就跟之前的相同，因此不用。但如果忘了调用的话，你的这个 project 单独编译时很可能会 break。

③ 确保已经导入了 message runtime 依赖。方法是找到 catkin\_package。添加上 message\_runtime。



```
catkin_package(
# INCLUDE_DIRS include
# LIBRARIES beginner_tutorials
# CATKIN_DEPENDS roscpp rospy std_msgs
# DEPENDS system_lib
  CATKIN_DEPENDS message_runtime
)
```

④ 找到下面的代码块：

```
# add_message_files(
#
#   FILES
#
#   Message1.msg
#   Message2.msg
# )
```

去掉注释符号#，将 Message 信息改成我们自己写的 Message。改完之后效果如下：

```
## Generate messages in the 'msg' folder
add_message_files(
  FILES
  Num.msg
)
```

靠手动添加.msg 文件，我们已经确信 CMake 已经知道在你添加其他的.msg 文件之后，何时必须重新配置 project。

现在，我们应该已经确信 generate\_messages()函数已经被调用。在文档中，找到下图代码块，去掉注释。

```
## Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

现在，你已经准备好了由你的 msg 定义生成源文件的准备。如果你想现在就生成，你可以跳过下面的部分，直接看“Common step for msg and srv”

[http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv#Common\\_step\\_for\\_msg\\_and\\_srv](http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv#Common_step_for_msg_and_srv)

这里我们接着跟着教程做。

### 10.2.2 使用 rosmmsg

上述讲的是所有要创建一个 msg 的步骤。现在，让我们用 rosmmsg show 命令确保 ROS 可以看到我们的 message。

```
$ rosmmsg show beginner_tutorials/Num
```

运行结果如下：

```
exbot@ubuntu:~$ rosmmsg show beginner_tutorials/Num
int64 num
string first_name
string last_name
uint8 age
uint32 score

exbot@ubuntu:~$
```

在先前的例子中，message 类型包括了两个部分：一个是 `beginner_tutorials`，这是 message 所在的包的名字，一个是 `Num`，这是 msg 文件的名字。

如果你没记住定义 message 时的包名，你也可以不管报名，直接用下面命令：

```
$ rosmmsg show Num
```

会看到如下结果。

```
exbot@ubuntu:~$ rosmmsg show Num
[beginner_tutorials/Num]:
int64 num
string first_name
string last_name
uint8 age
uint32 score

exbot@ubuntu:~$
```

## 10.3 使用 srv

### 10.3.1 创建一个 srv

我们还用之前的 `beginner_tutorials` 这个包来创建一个 srv。

```
$ roscd beginner_tutorials
```

```
$ mkdir srv
```

在这个包下创建一个 `srv` 目录。

这次我们不手动创建一个 `srv` 定义，而是从另外一个包里拷贝一个已存在的 `srv` 文件。

执行下面命令。命令格式为：

```
$ roscp [package_name] [file_to_copy_path] [copy_path]
```

具体的命令为：

```
$ roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts.srv
```

可看到 `srv` 目录下多了一个 `AddTwoInts.srv` 文件，其内容为：

```
AddTwoInts.srv ✕
int64 a
int64 b
---
int64 sum
```

与创建 msg 类似，这里也还差一步，需要确认 srv 文件已经生成 C++/Python 等其他语言源码。

① 除非之前做过，否则也要在 CMakeLists.txt 的 find\_package 中添加 message\_generation。（我们在创建 msg 时已经添加过了，因此此处不用再添加）

虽然 message\_generation 这个包名里是 ‘message’，但生成 msg 和 srv 都是用这同一个包。

② 同样，也需要在 package.xml 文件中添加相应内容，具体参见 msg 创建时的讲解。

③ 去掉 add\_service\_files 中的注释。并把自定义的 srv 添加上

```
## Generate services in the 'srv' folder
add_service_files(
  FILES
  AddTwoInts.srv
)
```

现在已经生成了 service 定义的源文件。

### 10.3.2 使用 rossrv

同 msg 相同，现在可以通过下面的命令格式查看自定义的 srv。

```
$ rossrv show <service type>
```

比如，查看我们刚刚自定义的 AddTwoInts.srv

```
$ rossrv show beginner_tutorials/AddTwoInts
```

可看到如下结果：

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

与 rosmmsg 类似，你也可以直接通过 service 的文件名来查询而忽略包名。

```
exbot@ubuntu:~$ rossrv show AddTwoInts
[beginner_tutorials/AddTwoInts]:
int64 a
int64 b
---
int64 sum

[rospy_tutorials/AddTwoInts]:
int64 a
int64 b
---
int64 sum
```

可以看到，除了我们自定义的 srv 外，在之前我们 srv 的拷贝源 rospy 那里还有一个名为 AssTwoInts 的 srv。

## 10.4 msg 和 srv 的一般步骤

在 CMakeLists.txt 中找到如下代码块：

```
# generate_messages(
```

```
# DEPENDENCIES

# # std_msgs # Or other packages containing msgs

# )
```

去掉注释，添加任何你依赖的包含你用到的 messages 定义的.msg 文件的包，比如 std\_msgs。就像这样：

```
generate_messages(

  DEPENDENCIES

  std_msgs

)
```

现在我们已经新建了一些新的 messages，我们需要重新 make 我们的包：

```
# In your catkin workspace

$ cd ../../..

$ catkin_make

$ cd -
```

```
####
Scanning dependencies of target std_msgs_generate_messages_lisp
[ 0%] Built target std_msgs_generate_messages_lisp
Scanning dependencies of target beginner_tutorials_generate_messages_lisp
[ 12%] Generating Lisp code from beginner_tutorials/Num.msg
[ 25%] Generating Lisp code from beginner_tutorials/AddTwoInts.srv
[ 25%] Built target beginner_tutorials_generate_messages_lisp
Scanning dependencies of target std_msgs_generate_messages_cpp
[ 25%] Built target std_msgs_generate_messages_cpp
Scanning dependencies of target beginner_tutorials_generate_messages_cpp
[ 50%] Generating C++ code from beginner_tutorials/Num.msg
[ 50%] Generating C++ code from beginner_tutorials/AddTwoInts.srv
[ 50%] Built target beginner_tutorials_generate_messages_cpp
Scanning dependencies of target std_msgs_generate_messages_py
[ 50%] Built target std_msgs_generate_messages_py
Scanning dependencies of target beginner_tutorials_generate_messages_py
[ 62%] Generating Python from MSG beginner_tutorials/Num
[ 75%] Generating Python code from SRV beginner_tutorials/AddTwoInts
[ 87%] Generating Python msg __init__.py for beginner_tutorials
[100%] Generating Python srv __init__.py for beginner_tutorials
[100%] Built target beginner_tutorials_generate_messages_py
Scanning dependencies of target beginner_tutorials_generate_messages
[100%] Built target beginner_tutorials_generate_messages
exbot@ubuntu:~/catkin_ws$
```

任何 msg 目录下的.msg 文件都是生成支持各种编程语言的代码。C++的 message header 文件生成到

下面目录中： `~/catkin_ws/devel/include/beginner_tutorials/`。而 **Python** 脚本会创建到下面目录中：

`~/catkin_ws/devel/lib/python2.7/dist-packages/beginner_tutorials/msg.lisp` 文件会出现在下面目录中：

`~/catkin_ws/devel/share/common-lisp/ros/beginner_tutorials/msg/`

## 10.5 一些帮助工具

用 `rosmmsg -h` 会得到命令列表：

```
$ rosmmsg -h
```

如得到下面列表：

Commands:

```
rosmmsg show Show message description
rosmmsg users Find files that use message
rosmmsg md5 Display message md5sum
rosmmsg package List messages in a package
rosmmsg packages List packages that contain messages
```

而且还可以使用 `-h` 命令查询子命令，比如：

```
$ rosmmsg show -h
```

10.6 复习一下现在已经用到的一些命令：

- `rospack = ros+pack(age)` : provides information related to ROS packages
- `rostack = ros+stack` : provides information related to ROS stacks
- `roscd = ros+cd` : **changes** **directory** to a ROS package or stack
- `rosls = ros+ls` : **lists** files in a ROS package
- `roscp = ros+cp` : **copies** files from/to a ROS package
- `rosmmsg = ros+msg` : provides information related to ROS message definitions
- `rossrv = ros+srv` : provides information related to ROS service definitions
- `rosmake = ros+make` : makes (compiles) a ROS package
  - You should use `catkin_make` if you are using a catkin workspace.

## 十一、写一个简单的发布者 **Publisher** 和订阅者 (**Subscriber**) (C++语言)

### 11.1 写一个 **Publisher** 节点

**Node** 是一个连接到 ROS 网络的可执行节点。我们在本节中要创建一个 **Publisher** 节点，这个节点会不停地广播一个 `message`。

现在我们进入先前创建的包的路径：

```
cd ~/catkin_ws/src/beginner_tutorials
```



进入 `beginner_tutorials` 下的 `src` 目录，创建一个 `talker.cpp` 文件。

`talker.cpp` 的代码参见以下网站：

[https://raw.githubusercontent.com/ros/ros\\_tutorials/groovy-devel/roscpp\\_tutorials/talker/talker.cpp](https://raw.githubusercontent.com/ros/ros_tutorials/groovy-devel/roscpp_tutorials/talker/talker.cpp)

代码解释：

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

`ros/ros.h` 包含了一些基本的公共的 ROS 操作系统的头文件。

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

这个导入了 `std_msgs` 包中的 `String.h` 文件。这是由 `std_msgs` 包中的 `String.msg` 文件自动生成的头文件。对于这个 `message` 定义的更多信息，可以看看 `msg` 页。 <http://wiki.ros.org/msg>

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

初始化 ROS。这允许 ROS 系统通过命令行重新命名——现在还不重要。这也允许我们自定义命名我们节点的名字。节点名字在正在运行的系统中必须是独一无二的。

这里的名字必须是一个 `base name`，不能包含 `'/'`。

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

创建一个指向该线程节点的 `handle`。第一个创建的 `NodeHandle` 会做一些节点的初始化工作，最后一个会销毁节点并回收节点占用的资源。

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

告知 `master` 节点，我们将要向 `chatter` 这个 `Topic` 上发布一个 `std_msgs/String` 型的 `message`。这会让主节点告知所有订阅 `chatter` 这个 `Topic` 的节点，我们将向这个 `topic` 发布数据了。第二个参数是我们 `publishing` 的队列的大小。在这种情况下，如果我们 `publishing` 的太快，系统最大会缓存 1000 个 `messages`，然后再有 `messages` 传来，那 1000 之前的 `message` 就会被扔掉。

`NodeHandle::advertise()` 返回一个 `ros::Publisher` 对象，这个对象有两个作用：（1）它包含一个 `publish()` 方法，能够让你向它所创建的 `topic` 上发布信息。（2）当它超出范围时，会自动 `unadvertise`。

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

一个 `ros::Rate` 对象允许你指定一个循环的频率，从上一次调用 `Rate::sleep()` 方法后，他会不停地循环指定时间，然后会休眠一定的时间。在这里我们设定为 10Hz。

```
81 int count = 0;
```

```
82 while (ros::ok())
```

```
83 {
```

缺省情况下，`roscpp` 会安装一个 `SIGINT` 的 `handler`，这个 `handler` 会处理 `Ctrl-C` 事件，使 `ros::ok()` 返回 `false`。

`ros::ok()` 在下列情形时会返回 `false`：

- `SIGINT` 发生 (`Ctrl-C`)
- 当我们被网络中的另一个同名节点顶替掉时
- `ros::shutdown()` 被程序的其他部分调用时
- 所有的 `ros::NodeHandles` 被 `destroyed` 时

一旦 `ros::ok()` 返回 `false`，所有的 ROS calls 会失败。

```
87 std_msgs::String msg;
```

```
88
```

```
89 std::stringstream ss;
```

```
90 ss << "hello world " << count;
```

```
91 msg.data = ss.str();
```

我们在 ROS 中广播了一个 `message`，`message` 类通常是由 `msg` 文件生成的。更复杂的数据类型也可

以，但现在我们只用标准的 String 型 message，这个 message 只有一个 string 数据。

```
101 chatter_pub.publish(msg);
```

现在，我们已经把这个 message 广播给任意连接到该 topic 的节点了。

```
93 ROS_INFO("%s", msg.data.c_str());
```

在 ROS 里，用 ROS\_INFO 替代了 printf/cout 的输出功能。更详细用法参见 <http://wiki.ros.org/rosconsole>

```
103 ros::spinOnce();
```

对于这个简单例子而言，调用 ros::spinOnce() 不是必须的。因为我们不接收任何 callbacks。然而，如果我们要在这个应用中加入一个订阅者 subscriber，如果这没有 ros::spinOnce() 的话，你的 callbacks 永远不会被调用，所以在这里加上是比较好的。

```
105 loop_rate.sleep();
```

现在，我们用 ros::Rate 对象休眠一定时间，保证 10Hz 的发布速率。

这里我们简单总结一下这段代码，主要完成了以下事情：

- 初始化 ROS 系统。
- 通知主节点，我们将要向 chatter 这个 topic 发布 std\_msgs/String 型的 messages。
- 用 Loop 循环，1s 向 chatter 发送 10 次 messages。

现在，我们可以写一个节点来接收 messages 信息。

## 11.2 写一个 Subscriber 节点。

在 beginner\_tutorials 这个包的 src 目录下创建一个 listener.cpp 文件。把下面的代码粘过去。代码地址：

[https://raw.githubusercontent.com/ros/ros\\_tutorials/groovy-devel/roscpp\\_tutorials/listener/listener.cpp](https://raw.githubusercontent.com/ros/ros_tutorials/groovy-devel/roscpp_tutorials/listener/listener.cpp)

代码解释：

```
34 void chatterCallback(const std_msgs::String::ConstPtr& msg)
35 {
36   ROS_INFO("I heard: [%s]", msg->data.c_str());
37 }
```

这是一个回调函数。当一个新的 message 发布到 chatter topic 时会调用该函数。如果需要，你可以存下这个 message 而不用担心没有复制该数据就被删掉。

```
75 ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

当一个新消息到达时，chatter 这个 topic 的订阅者和 ROS 主节点会调用 chatterCallback() 方法。上面这个函数就设置了 sub 这个订阅者订阅的是 chatter 这个 topic，回调函数是 chatterCallback()，而第二个参数是队列大小。万一我们处理 message 的速度不够快，我们可以缓存 1000 个 messages。

NodeHandle::subscribe() 返回一个 ros::Subscriber 对象。当这个 Subscriber 对象被销毁时，会自动从 chatter Topic 上解除订阅。

有不同版本的 NodeHandle::subscribe() 函数，允许你定制你的类成员函数，甚至 anything callable by a Boost.Function object。

```
82 ros::spin();
```

ros::spin 会进入一个循环，尽快地请求 message 回调函数。如果没有任务，这个循环不会占用很多 CPU 资源。ros::spin() 在 ros::ok() 函数一旦返回 false 时推出，这意味着 ros::shutdown() 被调用或 Ctrl-C 事件发生等其他导致 ros::ok() 返回 false 的时间。

也有一些其他调用回调函数的方式，我们以后再说。

简单总结，在这个程序中主要做了一下工作：

- 初始化 ROS 系统
- 订阅 chatter Topic
- spin,等待 messages 到达
- 当 message 到达时，调用回调函数 chatterCallback()

### 11.3 编译你的节点

在原先生成的包的 CMakeLists.txt 文件（已经有了再 msg 和 srv 中的更改）中，加下下面几行到文件的末尾。

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

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

去掉注释等无关内容，应该向这样：

切换行号显示

```
1 cmake_minimum_required(VERSION 2.8.3)
2 project(beginner_tutorials)
3
4 ## Find catkin and any catkin packages
5 find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)
6
7 ## Declare ROS messages and services
8 add_message_files(FILES Num.msg)
9 add_service_files(FILES AddTwoInts.srv)
10
11 ## Generate added messages and services
12 generate_messages(DEPENDENCIES std_msgs)
13
14 ## Declare a catkin package
15 catkin_package()
16
17 ## Build talker and listener
18 include_directories(include ${catkin_INCLUDE_DIRS})
19
20 add_executable(talker src/talker.cpp)
21 target_link_libraries(talker ${catkin_LIBRARIES})
22 add_dependencies(talker beginner_tutorials_generate_messages_cpp)
23
24 add_executable(listener src/listener.cpp)
25 target_link_libraries(listener ${catkin_LIBRARIES})
26 add_dependencies(listener beginner_tutorials_generate_messages_cpp)
```

这样，在编译时就会创建两个可执行的 talker 和 listener，默认情况下，这两个可执行文件会存到包的目录下的 devel 空间下：~/catkin\_ws/devel/lib/share/<package name>



同时也看到，也必须把依赖加进去。

```
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
```

这确保这个包的 `message` 头文件在使用之前被生成。如果你用到了你的工作空间之外的其他包的 `messages`，你需要添加依赖到他们各自的生成目标中，因为 `catkin` 是平行编译所有的工程的。

你可以调用可执行文件的路径或用 `roslaunch` 调用他们。他们没放在 '`<prefix>/bin`' 里因为这样在安装你的包到系统时会破坏 `PATH`。如果你希望你的可执行文件再安装时放到 `PATH` 中，你可以设置安装目标。

进一步的信息可以查看 <http://wiki.ros.org/catkin/CMakeLists.txt>

现在，我们可以运行 `catkin_make` 进行编译了。

```
# In your catkin workspace
$ catkin_make
```

如果你是在添加一个新的包，你需要告诉 `catkin` 强制 `making`，用 '`--force-cmake`' 选项。

现在已经写了一个简单的发布者和订阅者了，下一节我们可以测试一下这两个节点。

## 十二、测试发布节点和订阅节点。

### 12.1 运行 Publisher

首先要运行起 `roscore`。

```
$ roscore
```

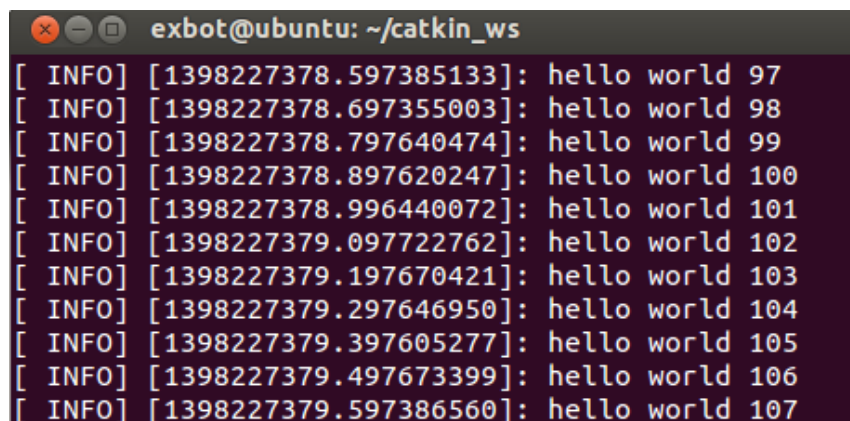
如果正在使用 `catkin` 空间，在 `catkin_make` 之后，如果要运行你的应用，请确保你已经 `sourced` 你的工作空间的 `setup.sh` 文件

```
# In your catkin workspace
$ cd ~/catkin_ws
$ source ./devel/setup.bash
```

让我们运行上一节中写的 `publisher`。

```
$ roslaunch beginner_tutorials talker          (C++)
```

运行结果

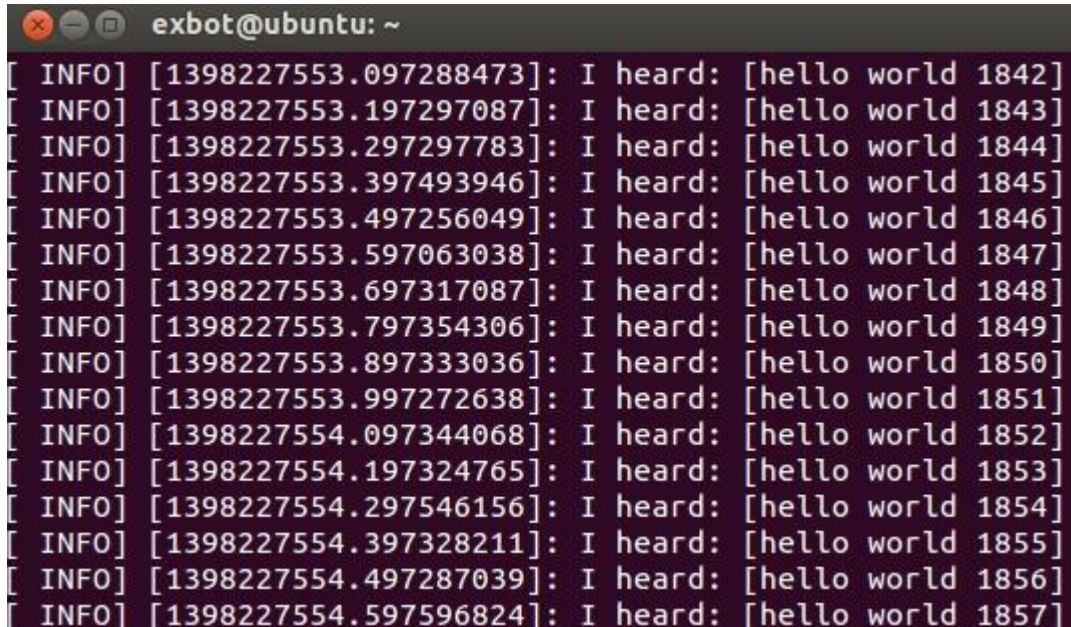


```
exbot@ubuntu: ~/catkin_ws
[ INFO] [1398227378.597385133]: hello world 97
[ INFO] [1398227378.697355003]: hello world 98
[ INFO] [1398227378.797640474]: hello world 99
[ INFO] [1398227378.897620247]: hello world 100
[ INFO] [1398227378.996440072]: hello world 101
[ INFO] [1398227379.097722762]: hello world 102
[ INFO] [1398227379.197670421]: hello world 103
[ INFO] [1398227379.297646950]: hello world 104
[ INFO] [1398227379.397605277]: hello world 105
[ INFO] [1398227379.497673399]: hello world 106
[ INFO] [1398227379.597386560]: hello world 107
```

## 12.2 运行 Subscriber

在新的 terminal 中运行 listener:

```
$ rosrun beginner_tutorials listener (C++)
```



```
exbot@ubuntu: ~
[ INFO] [1398227553.097288473]: I heard: [hello world 1842]
[ INFO] [1398227553.197297087]: I heard: [hello world 1843]
[ INFO] [1398227553.297297783]: I heard: [hello world 1844]
[ INFO] [1398227553.397493946]: I heard: [hello world 1845]
[ INFO] [1398227553.497256049]: I heard: [hello world 1846]
[ INFO] [1398227553.597063038]: I heard: [hello world 1847]
[ INFO] [1398227553.697317087]: I heard: [hello world 1848]
[ INFO] [1398227553.797354306]: I heard: [hello world 1849]
[ INFO] [1398227553.897333036]: I heard: [hello world 1850]
[ INFO] [1398227553.997272638]: I heard: [hello world 1851]
[ INFO] [1398227554.097344068]: I heard: [hello world 1852]
[ INFO] [1398227554.197324765]: I heard: [hello world 1853]
[ INFO] [1398227554.297546156]: I heard: [hello world 1854]
[ INFO] [1398227554.397328211]: I heard: [hello world 1855]
[ INFO] [1398227554.497287039]: I heard: [hello world 1856]
[ INFO] [1398227554.597596824]: I heard: [hello world 1857]
```

## 十三、写一个简单的 Service 和 Client

### 13.1 写一个 Service 节点

这一节我们将创建两个 service 节点，用来接收两个 ints 数并返回两个数之和。

进入之前建的包的目录:

```
cd ~/catkin_ws/src/beginner_tutorials
```

确认你之前已经在 beginner\_tutorials 里建立一个 srv 文件。

在 beginner\_tutorials 包的 src 目录里创建一个文件: add\_two\_ints\_server.cpp

代码如下:

切换行号显示

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3
4 bool add(beginner_tutorials::AddTwoInts::Request &req,
5          beginner_tutorials::AddTwoInts::Response &res)
6 {
7     res.sum = req.a + req.b;
8     ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
9     ROS_INFO("sending back response: [%ld]", (long int)res.sum);
10    return true;
11 }
12
```

```

13 int main(int argc, char **argv)
14 {
15     ros::init(argc, argv, "add_two_ints_server");
16     ros::NodeHandle n;
17
18     ros::ServiceServer service = n.advertiseService("add_two_ints", add);
19     ROS_INFO("Ready to add two ints.");
20     ros::spin();
21
22     return 0;
23 }

```

代码解释：

```

1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"

```

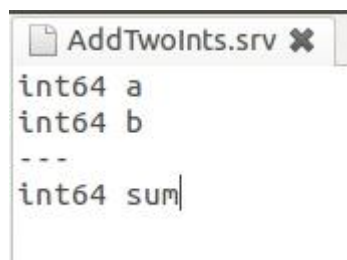
导入头文件。其中 `beginner_tutorials/AddTwoInts.h` 是由 `srv` 文件生成的头文件。

```

4 bool add(beginner_tutorials::AddTwoInts::Request &req,
5           beginner_tutorials::AddTwoInts::Response &res)

```

这个函数提供了两个 `ints` 相加的 `service`。它接收 `request` 中的数据，把计算结果放到 `response` 中，`request` 和 `response` 中的数据及变量名字是由 `srv` 文件定义的。函数的返回值是一个 `boolean` 型。我们可以再看看 `srv` 文件。



```

AddTwoInts.srv
int64 a
int64 b
---
int64 sum

```

```

6 {
7     res.sum = req.a + req.b;
8     ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
9     ROS_INFO("sending back response: [%ld]", (long int)res.sum);
10    return true;
11 }

```

这里将两个整数相加，并最后放到 `response` 中。然后输出一些日志信息。最后 `service` 完成所有任务后返回一个 `true` 值。

```

18 ros::ServiceServer service = n.advertiseService("add_two_ints", add);

```

这一句是创建 `service` 并将其广播给 ROS，告知有 “`add_two_ints`” 这么一个 `service`。

## 13.2 写一个 Client 节点

在 `beginner_tutorials` 的 `src` 目录下建一个 `add_two_ints_client.cpp` 文件。把下面代码粘贴进去：

```

1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 #include <cstdlib>
4
5 int main(int argc, char **argv)

```

```

6 {
7   ros::init(argc, argv, "add_two_ints_client");
8   if (argc != 3)
9   {
10    ROS_INFO("usage: add_two_ints_client X Y");
11    return 1;
12  }
13
14  ros::NodeHandle n;
15  ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwo
oInts>("add_two_ints");
16  beginner_tutorials::AddTwoInts srv;
17  srv.request.a = atoll(argv[1]);
18  srv.request.b = atoll(argv[2]);
19  if (client.call(srv))
20  {
21    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
22  }
23  else
24  {
25    ROS_ERROR("Failed to call service add_two_ints");
26    return 1;
27  }
28
29  return 0;
30 }

```

代码解释:

```

15  ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTw
oInts>("add_two_ints");

```

这句创建一个应用 `add_two_ints` 服务的 `client`。`ros::ServiceClient` 对象用于稍后调用服务。

```

16  beginner_tutorials::AddTwoInts srv;
17  srv.request.a = atoll(argv[1]);
18  srv.request.b = atoll(argv[2]);

```

这是一个自动生成 (autogenerated) `service` 类的实例 (instantiate) 并把参数值放入 `request` 成员。一个 `service` 类包括两个成员, `request` 和 `response`。它也包含两个类定义, `Request` 和 `Response`。

```

19  if (client.call(srv))

```

这会调用服务。`service calls` 正在阻塞, 它在请求之后会立刻返回。如果 `service call` 成功, `call()` 会返回 `true` 并且 `srv.response` 中的值有效。如果 `call` 调用不成功, `call()` 会返回 `false`。`srv.response` 中的值无效。

### 13.3 编译你的节点

再修改 `beginner_tutorials` 中的 `CmakeLists.txt` 文件, 添加如下代码:

```

27 add_executable(add_two_ints_server src/add_two_ints_server.cpp)
28 target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
29 add_dependencies(add_two_ints_server beginner_tutorials_gencpp)

```

30

```

31 add_executable(add_two_ints_client src/add_two_ints_client.cpp)
32 target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
33 add_dependencies(add_two_ints_client beginner_tutorials_gencpp)

```

这会创建两个可执行文件：add\_two\_ints\_server 和 add\_two\_ints\_client。默认情况会放入包的 devel 空间。位于：~/catkin\_ws/devel/lib/share/<package name>。你可以直接调用可执行文件或用 rosrun 执行它们。

现在可以运行 catkin\_make 编译工程。

```

# In your catkin workspace

cd ~/catkin_ws

catkin_make

```

## 十四、测试 Service 和 Client

### 14.1 运行 Service（记得先运行 roscore）

```
$ rosrun beginner_tutorials add_two_ints_server (C++)
```

你会看到如下输出信息：

```
Ready to add two ints.
```

### 14.2 运行 Client

```
$ rosrun beginner_tutorials add_two_ints_client 1 3 (C++)
```

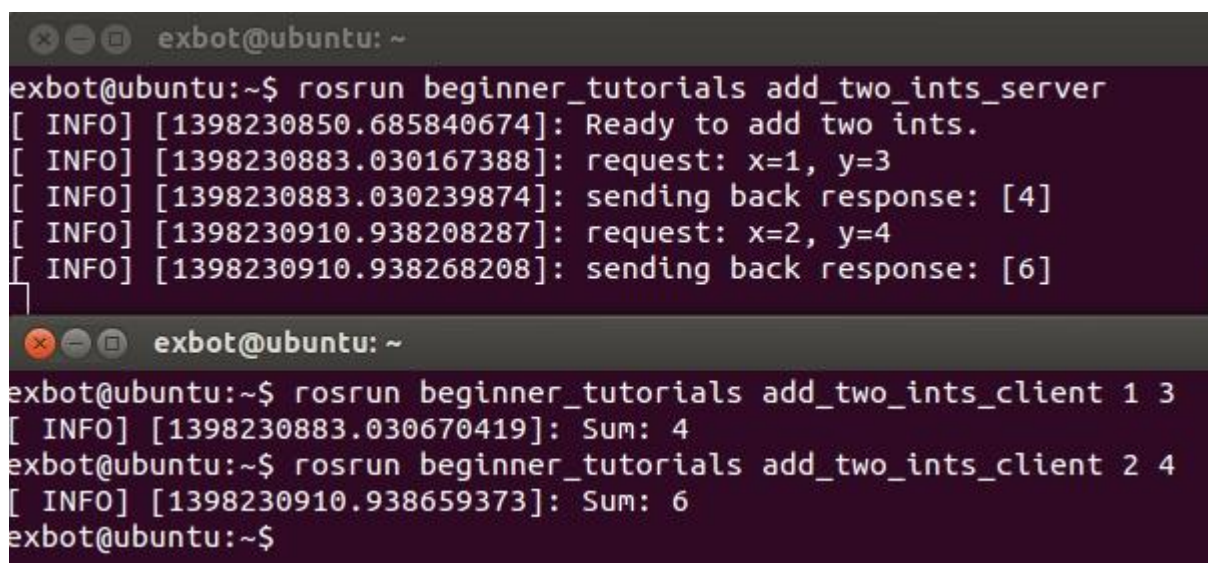
会看到如下信息：

```

Requesting 1+3

1 + 3 = 4

```



```

exbot@ubuntu: ~
exbot@ubuntu:~$ rosrun beginner_tutorials add_two_ints_server
[ INFO] [1398230850.685840674]: Ready to add two ints.
[ INFO] [1398230883.030167388]: request: x=1, y=3
[ INFO] [1398230883.030239874]: sending back response: [4]
[ INFO] [1398230910.938208287]: request: x=2, y=4
[ INFO] [1398230910.938268208]: sending back response: [6]

exbot@ubuntu: ~
exbot@ubuntu:~$ rosrun beginner_tutorials add_two_ints_client 1 3
[ INFO] [1398230883.030670419]: Sum: 4
exbot@ubuntu:~$ rosrun beginner_tutorials add_two_ints_client 2 4
[ INFO] [1398230910.938659373]: Sum: 6
exbot@ubuntu:~$

```



## 十五、记录数据以及回复数据

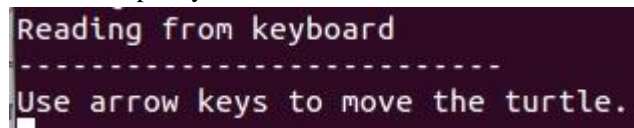
本节主要示范运行着的 ROS 系统如何记录数据到一个.bag 文件中，又如何恢复数据，这样能产生一个与之前的设置类似的动作。

### 15.1 记录数据（创建 bag 文件）

首先，执行下面两条指令

```
roscore  
  
roslaunch turtlesim turtlesim_node  
  
roslaunch turtlesim turtle_teleop_key
```

这会开启两个节点：`turtlesim` 仿真工具节点和一个接收键盘输入控制 `turtlesim` 的节点。在 `turtle_teleop_key` 窗口会看到如下信息：



```
Reading from keyboard  
-----  
Use arrow keys to move the turtle.
```

按方向键就会控制小龟移动。

#### 15.1.1 记录所有发布的 topics

首先，让我们用下面指令查看一下有哪些 topic 正在运行。

```
rostopic list -v
```

可以列出当前的发布者和订阅者信息：

```
Published topics:  
  
* /turtle1/color_sensor [turtlesim/Color] 1 publisher  
* /turtle1/command_velocity [turtlesim/Velocity] 1 publisher  
* /rosout [roslib/Log] 2 publishers  
* /rosout_agg [roslib/Log] 1 publisher  
* /turtle1/pose [turtlesim/Pose] 1 publisher  
  
Subscribed topics:  
  
* /turtle1/command_velocity [turtlesim/Velocity] 1 subscriber  
* /rosout [roslib/Log] 1 subscriber
```

只有 published 的 topics 相关的 message 是可以记录在 data log 文件里的 message 类型。`/turtle1/command_velocity` 这个 topic 是一个由 `teleop_turtle` 发布的命令信息为输入。`/turtle1/color_sensor` 这个 Message 和 `/turtle1/pose` 是由 `turtlesim` 发布的输出信息。

现在，我们要记录发布的数据。打开一个新的 terminal，在这个窗口中运行以下命令：



```
mkdir ~/bagfiles
cd ~/bagfiles
rosbag record -a
```

这里我们只是创建了一个临时的路径来记录数据，然后运行带有-a 参数的 rosbag record 命令，表明所有发布的 topics 都应该记录在 bag 文件中。

回到之前的 turtle\_teleop\_key 窗口，移动小龟 10s 左右。

在窗口中运行 rosbag record，并用 Ctrl-C 退出。现在检查 home/bagfiles 目录。你可看到一个以年、日期、时间和suffix.bag 开头命名的bag 文件。这是包含在 rosbag record 运行期间所有节点发布的 topics 的信息。

## 15.2 查看 bag 文件及 playing bag 文件

现在我们已经用 rosbag record 记录了一个 bag 文件，我们可以用 rosbag info 查看 bag 文件，用 rosbag play 恢复 bag 文件。

首先，我们可以用 rosbag info 查看 bag 文件（需要在~/bagfiles 路径下）。

```
rosbag info <your bagfile>
```

```
exbot@ubuntu:~$ cd ~/bagfiles
exbot@ubuntu:~/bagfiles$ rosbag info 2014-04-23-18-00-09.bag
path:          2014-04-23-18-00-09.bag
version:       2.0
duration:      32.8s
start:         Apr 23 2014 18:00:10.44 (1398247210.44)
end:           Apr 23 2014 18:00:43.23 (1398247243.23)
size:          355.5 KB
messages:      4306
compression:   none [1/1 chunks]
types:         geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
               rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593fb]
               turtlesim/Color      [353891e354491c51aabe32df673fb446]
               turtlesim/Pose       [863b248d5016ca62ea2e895ae5265cf9]
topics:        /rosout              104 msgs      : rosgraph_msgs/Log (2
actions)
               /rosout_agg          100 msgs      : rosgraph_msgs/Log
               /turtle1/cmd_vel     17 msgs       : geometry_msgs/Twist
               /turtle1/color_sensor 2043 msgs     : turtlesim/Color
               /turtle1/pose        2042 msgs     : turtlesim/Pose
exbot@ubuntu:~/bagfiles$
```

在默认模式下，rosbag play 会在每个 message advertising 之后等待固定周期（2s），然后真正开始发布 bag file 的内容。等待一定的时间是允许订阅这个 message 的订阅者接到通知，知道 message 已经 advertised。如果 rosbag play 在 advertising 之后立即发布，订阅者有可能收不到最开始发布的几个 messages。等待时间长短可用-d 参数设置。-s 参数可设置在一定间隔后开始 rosbag play。-r 参数可以设置发布 message 的频率。

### 15.3 记录数据的一部分

有可能我们不需要记录全部数据，只需要记录一部分 topic。格式如下：

```
rosviz record -O subset /turtle1/command_velocity /turtle1/pose
```

## 十六、roswtf 的使用

```
$ roscd  
  
$ roswtf
```