安装 ROS：
ROS 官方安装网站：
http://wiki.ros.org/kinetic/Installation

> To install our previous release, **ROS Jade Turtle**, please see the Jade installation instructions.
>
> The previous long-term support release, **ROS Indigo Igloo**, is available for Ubuntu Trusty (14.04 LTS) and many other platforms. Please refer to the Indigo installation instructions if you need to use this version due to robot or platform compatibility reasons.

安装的 Linux 版本为 Ubuntu Trusty (ubuntu-14.04.1-desktop-amd64.iso), 对应的 ROS 的版本为 indigo,官网安装主页已更新对应更新新版本的 Ubuntu。考虑到 indigo 的丰富的支持资料和成熟度(LTS, long term support)，因此选择了此版本的 ROS。点击 indigo installation instructions 选择 ROS 平台为 Ubuntu 进入旧版 indigo 的安装指导界面。
安装完成，进入 Linux 界面，发现 wifi 连接不上，提示：wifi 已通过硬件开关禁用，应为系统 Bug，待机后可以成功连接或在 terminal 输入：
echo "options asus_nb_wmi wapf=4"| sudo tee /etc/modprobe.d/asus_nb_wmi.conf 重启即可。
(Ubuntu 细节问题不少，在输入上边的命令重启之前将电脑盖子盖住再打开发现网络可以连接，同样的问题出现在带耳机播放音乐时，没有声音，终端输入 *alsamixer* ，调大 speaker 声音，仍然没有声音，但当把电脑盖子盖住再打开就能听见声音了...
一种解释如下：http://www.2cto.com/os/201410/346653.html）
连网后按照教程依次在 terminal 输入相应命令依次进行添加软件源、添加 keys、安装 Debian 软件包、初始化 rosdep、环境设置、安装 rosinstall。
rosdep is a tool you can use to install system dependencies required by ROS packages. 安装后可安装其它 ROS 安装包，如下：
rosdep install turtlesim
安装过程也可参考文章 ROS 学习笔记(一):在 Ubuntu14.04 中安装 ROS Indigo
可以用下面的形式安装单个包：
sudo apt-get install ros-kinetic-PACKAGE
eg: sudo apt-get install ros-kinetic-slam-gmapping
可以通过下面的命令来执行 ros 包搜索：
apt-cache search ros-kinetic

ROS Documentation

ROS 文件系统介绍

1. 预备工作
下列命令会用到 ros-tutorials 程序包，请先安装：
sudo apt-get install ros-<distro>-ros-tutorials
将 <distro> 替换成所安装的版本（比如 Jade、Indigo、hydro、groovy、fuerte 等）：
sudo apt-get install ros-Indigo-ros-tutorials

文件系统概念
Packages: 软件包，是 ROS 应用程序代码的组织单元，每个软件包都可以包含程序库、可执行文件、脚本或者其它手动创建的东西。
Manifest (package.xml): 清单，是对于'软件包'相关信息的描述,用于定义软件包相关元信息之间的依赖关系，这些信息包括版本、维护者和许可协议等。

2. 文件系统工具
程序代码是分布在众多 ROS 软件包当中，当使用命令行工具（比如 ls 和 cd）来浏览时会非常繁琐，因此 ROS 提供了专门的命令工具来简化这些操作。
2.1 使用 rospack

rospack 允许获取软件包的有关信息。常结合 find 参数选项来返回软件包的路径信息。
用法： rospack find [包名称]
示例：
rospack find roscpp
2.2 使用 roscd
roscd 是 rosbash 命令集中的一部分，它允许你直接切换(cd)工作目录到某个软件包或者软件包集当中。

用法：roscd [本地包名称[/子目录]]
示例：
roscd roscpp　#切换到一个软件包或软件包集
roscd roscpp/cmake　#切换到一个软件包或软件包集的子目录中
注意，就像 ROS 中的其它工具一样，roscd 只能切换到那些路径已经包含在 ROS_PACKAGE_PATH 环境变量中的软件包，要查看 ROS_PACKAGE_PATH 中包含的路径可以输入：
echo $ROS_PACKAGE_PATH
特别地：
roscd log
可以切换到 ROS 保存日记文件的目录下。需要注意的是，如果没有执行过任何 ROS 程序，系统会报错说该目录不存在。
2.3 使用 rosls
rosls 是 rosbash 命令集中的一部分，它允许直接按软件包的名称而不是绝对路径执行 ls 命令（罗列目录）。
用法： rosls [本地包名称[/子目录]]
示例：
rosls roscpp_tutorials
2.4 rosed
rosed 也是 rosbash 命令集的一部分，它允许直接编辑软件包中的文件，而无须知道文件的具体位置，如：
rosed roscpp ros.h
rosed,rosbash,rosrun,roscd,roslaunch 等都是 ROS 提供的命令行工具，具体见：ROS Command-line tools

创建 ROS 工作空间/程序包，编写/编译/运行 ROS 程序
1. 创建 ROS 工作空间
工作空间（workspace）是 ROS 里面最小环境配置单位，可以把 workspace 看成是一个有结构的文件夹，它包含多个 package 以及一些结构性质文件。**一次将一个 workspace 配置进环境变量里面，才能使用 ROS 命令执行与这个 workspace 里面的 package 相关的操作。**
创建名为 test 的工作空间：
mkdir -p ~/test/src
ls

2. 创建一个功能包
功能包（package）是 ROS 里面最小的编译单位，也是 ROS 里面的搜索单位，你可以把 package 看成是一个有结构的文件夹，它包含多个节点（node）以及一些结构性质文件。ROS 里面的 node，是以 package 为单位进行编译的，一次编译 package 里面多个 node。
一个合格的 catkin 功能包必须符合以下要求：
① 必须包含 package.xml 文件。该文件提供有关功能包的元信息。
② 必须包含 CMakeLists.txt 文件。
③ 同一目录下不能有嵌套的或多个功能包存在。
一个简单的工作空间也许看起来像这样：

```
  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
```

使用 catkin_create_pkg 命令来创建一个新的 catkin 程序包。catkin_create_pkg 命令要求输入 package_name，如果有需要还可以在后面添加一些需要依赖的其它程序包。命令格式为：
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
首先切换到上边创建的 catkin 工作空间的 src 目录下：
cd ~/test/src
使用 catkin_create_pkg 命令来创建一个名为'hello_test'的新程序包，这个程序包依赖于 roscpp：
catkin_create_pkg hello_test roscpp

```
geds@ubuntu:~/test/src$ catkin_create_pkg hello_test roscpp
Created file hello_test/package.xml
Created file hello_test/CMakeLists.txt
Created folder hello_test/include/hello_test
Created folder hello_test/src
Successfully created files in /home/geds/test/src/hello_test. Please adjust the
values in package.xml.
```

由运行结果可以看出，在 hello_test 目录下创建了 2 个文件，分别是 package.xml 和 CMakeLists.txt。（其中 package.xml 称为清单文件，后面会详细介绍。CMakeLists.txt,是一个 Cmake 的脚本文件,Cmake 是一个符合工业标准的跨平台编译系统。这个文件包含了一系列的编译指令,包括应该生成哪种可执行文件,需要哪些源文件,以及在哪里可以找到所需的头文件和链接库。当然,这个文件表明 catkin 在内部使用了 Cmake。）

3. package.xml 文件
package.xml 文件主要由 4 个标签组成，分别为描述标签、维护者标签、许可证标签、依赖项标签。重点来看依赖项标签。
依赖项标签用来描述功能包的各种依赖项，这些依赖项分为：
build_depend、run_depend、buildtool_depend、test_depend。

```
<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>message_generation</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use run_depend for packages you need at runtime: -->
<!--   <run_depend>message_runtime</run_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!--   <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<run_depend>roscpp</run_depend>
```

由上图可看出，除了 catkin 中默认提供的 buildtool_depend，我们列出的依赖包都已经被添加到 build_depend 和 run_depend 标签中。显然，roscpp 在编译和运行时均需要用到。

4. 编写 ROS 程序
在 hello_test 目录下，创建一个简单的 cpp 程序文件，并命名为 hello.cpp。运行如下命令：
cd ~/test/src/hello_test
touch hello.cpp
编辑 hello.cpp 文件，运行如下命令：
gedit hello.cpp
编写 ROS 程序如下：

hello.cpp

```
#include <ros/ros.h>

int main ( int argc , char ** argv )
{
  ros::init ( argc , argv , "hello" ) ;
  ros::NodeHandle nh ;
  ROS_INFO_STREAM( " Hello , ROS! " ) ;
}
```

5.编译 ROS 程序

5.1 声明依赖库。

首先,我们需要声明程序所依赖的其他功能包。对于 c++程序而言,此步骤是必要的,以确保 catkin 能够向 c++编译器提供合适的标记来定位编译功能包所需的头文件和链接库。

为了给出依赖库,我们需要编辑 CMakeLists.txt 文件。即修改 find_package 行如下:

find_package(catkin REQUIRED COMPONENTS roscpp)

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_test)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
)
```

同样，我们也需要在 package.xml 文件中列出依赖库，由于 hello 程序在编译时和运行时都需要 roscpp 库，故 roscpp 添加至 bulid_depend（编译依赖）和 run_depend（运行依赖）。如下:

```
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>roscpp</run_depend>
```

（其实，由于我们之前在创建功能包 hello_test 时，在命令行中已经声明其依赖包为 roscpp，故此步骤可以省略。）

5.2 声明可执行文件。

接下来，我们需要在 CMakeLists.txt 中添加两行,来声明我们需要创建的可执行文件。

add_executable(hello hello.cpp)

target_link_libraries(hello ${catkin_LIBRARIES})

```
## Declare a C++ executable
# add_executable(hello_test_node src/hello_test_node.cpp)
add_executable(hello hello.cpp)

## Specify libraries to link a library or executable target against
# target_link_libraries(hello_test_node
#   ${catkin_LIBRARIES}
  target_link_libraries(hello
    ${catkin_LIBRARIES})
```

第一句声明了我们想要的可执行文件的文件名 hello,以及生成此可执行文件所需的源文件列表 hello.cpp。如果有多个源文件,把它们列在此处,并用空格将其区分开；第二句告诉 Cmake 当链接此可执行文件时需要链接哪些库(在上面的 find_package 中定义)。如果你的包中包括多个可执行文件,为每一个可执行文件复制和修改上述两行代码。

5.3 编译工作区。

这一步我们需要回到工作空间目录，使用 catkin_make 命令进行编译。运行如下命令:

cd ~/test

catkin_make

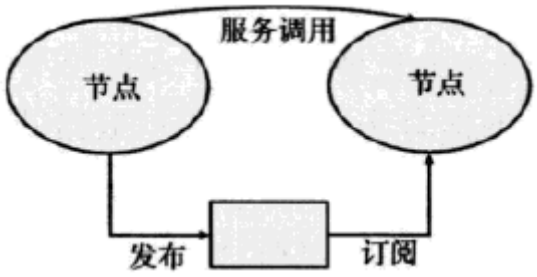
catkin_make 这个命令它将会完成一些配置步骤，并且在你的工作区中创建 devel 和 build 两个子目录。这两个新目录用于存放和编译相关的文件,例如自动生成的编译脚本、目标代码和可执行文件。此外,当完成功能包的相关工作后(译者注:即完成了编写、调试、测试等一系列工作后,此时代码基本定型),可以放心地删除 devel 和 build 两个子目录。

6. 运行 ROS 程序
hello 程序是一个节点（node），node 需要一个节点管理器才可以正常运行。运行下面命令启动节点管理器：
roscore
当我们创建一个 workspace 后，系统是不知道这个 workspace 存在的，一切与这个 workspace 相关的 ROS 命令都会失效——这个 workspace 里面的 package 不会被 ROS 命令发现。要想将这个 workspace 的信息配置到环境里面，就必须执行这个 workspace 里面的 setup.bash 脚本。故为了让 ROS 能够找到我们创建的功能包和新生成的可执行文件，在启动完节点管理器后，还需要执行 workspace 中 devel 子目录下名为 setup.bash 的脚本文件。在 devel 目录下运行下面程序:
cd ~/test/devel
source setup.bash
接着用 rosrun 命令运行程序：
rosrun hello_test hello
至此，完成了创建工作空间、功能包、编写、编译、运行简单的 ROS 程序。

7. rospack 命令
rospack 命令工具可以查看功能包的依赖关系。
根据依赖关系，将依赖包分为一级依赖包和间接依赖包，其中一级依赖包是指功能包直接依赖的依赖包，间接依赖包是指依赖包的依赖包...
现在我们用 rospack 查看 hello_test 的一级依赖包：
rospack depends1 hello_test
（如果提示:no such package hello_test，是因为没有 source 一下~/test/devel/目录下的 setup.bash 文件）
在很多情况中，一个依赖包还会有它自己的依赖包，如 roscpp:
rospack depends1 roscpp
使用 rospack depends 可以递归检测出所有的依赖包:
rospack depends hello_test

了解 ROS 计算图级，理解 ROS 节点、话题

1. ROS 基本概念：



计算图级——是 ROS 处理数据的一种点对点的网络形式。程序运行时，所有进程及它们所进行的数据处理，将会通过一种点对点的网络形式表现出来，即通过节点、节点管理器、话题、服务等来进行表现。ROS 中的基本计算图级概念包括：节点、节点管理器、参数服务器、消息、服务、话题和包。这些概念以各种形式来提供数据。

1.1 节点（Node）
一个节点就是 ROS 功能包中的一个可执行文件，节点之间可以通过 ROS 客户端库（如 roscpp、rospy）相互通信。一个机器人控制系统由许多节点组成，这些节点各司其职，如，一个节点控制激光距离传感器，一个节点控制轮子的马达，一个节点执行定位，一个节点执行路径规划，一个节点提供系统的整个视图……

1.2 节点管理器（Master）

节点管理器是 ROS 名称服务，能够帮助节点找到彼此。节点通过与节点管理器通信来报告他们的注册信息。值得注意的是，当这些节点和节点管理器通信时，它们可以接受其他注册节点的信息，并能保持通信正常。当这些注册信息改变时，节点管理器也会回调这些节点。所以，没有节点管理器，节点将不能相互找到，也不能进行消息交换或者调用服务。

1.3 参数管理器（Parameter Server）
是节点管理器的一部分。其允许数据通过在一个中心位置的关键词来存储。

1.4 消息（Message）
消息是一种 ROS 数据类型，节点之间通过消息来互相通信。

1.5 话题（Topic）
话题是用于识别消息的名称，节点可以发布消息到话题，也可以订阅话题以接收消息。一个话题可能对应有许多节点作为话题发布者和话题订阅者，当然，一个节点可以发布和订阅许多话题。一个节点对某一类型的数据感兴趣，它只需订阅相关话题即可。一般来说。话题发布者和话题订阅者不知道对方的存在。发布者将信息发布在一个全局的工作区内，当订阅者发现该信息是它所订阅的，就可以接收到这个信息。

1.6 服务（Service）
我们可以看出，发布/订阅模式是一种多对多的传输方式，显然这种传输方式不适用于请求/回复的交互方式。请求/回复交互方式经常被用于分布式系统中。请求服务通过 Service 来进行，Service 被定义为一对消息结构：一个用于请求，一个用于回复。一个客户通过发送请求信息并等待响应来使用服务。（ROS 中的请求/回复交互方式好像是一个远程程序调用。）

1.7 消息记录包（Bag）
消息记录包是一种用于保存和回放 ROS 消息数据的格式。消息记录包是检索传感器数据的重要机制，这些数据虽然很难收集，但是对于发展和测试算法很有必要。

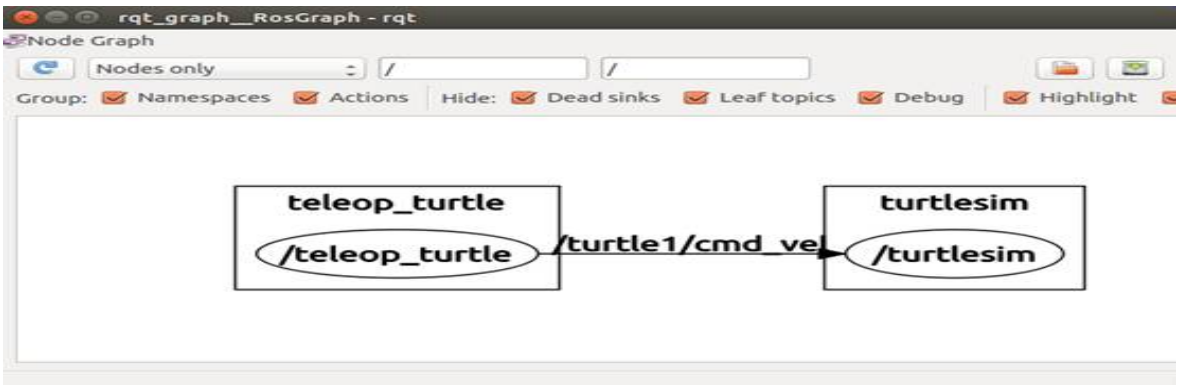
2. 基本命令：roscore, rosnode, rosrun
ros 命令的说明及参数可以通过 <命令> -h（或--help）来查看
2.1 roscore
在运行所有有关具体操作的 ROS 程序之前首先要运行该命令，然后打开新标签页运行其他 ROS 程序。
roscore

2.2 rosnode
显示当前运行的 ROS 节点信息。
列出活跃的节点：
rosnode list
rosnode info 命令返回的是关于一个特定节点的信息：
rosnode info /rosout

2.3 rosrun
rosrun 可以运行 package 中的可执行文件,不需要知道可执行 文件的位置
用法： rosrun [package_name] [node_name]
若文件为脚本文件，比如 python 文件 file_name.py。则格式为 rosrun [package_name] [file_name.py]
运行 turtlesim 包中的节点 turtlesim_node：
rosrun turtlesim turtlesim_node
通过命令来改变节点名称为 my_turtle：

rosrun turtlesim turtlesim_node __name:=my_turtle
在新标签页中运行命令：
rosrun turtlesim turtle_teleop_key
需要注意的是，应选中 turtle_teleop_key 所在的终端窗口，即最后打开的新标签页，以确保键盘输入能够被捕获。
在新的标签页用 rosnode list 命令查看此时运行的 ROS 节点:
rosnode list
可以看到，除了节点 turtlesim，现在又多出了节点 teleop_turtle。
可以用 rosnode ping 命令来测试节点：
rosnode ping my_turtle
总结：roscore= ros+core：master (provides name service for ROS) + rosout(stdout/stderr) + parameter server (parameter server will be introducedlater)
rosnode = ros+node : ROS tool to get information about a node
rosrun = ros+run : runs a node from a given package

3. ROS Topic
用键盘控制小海龟运动这一过程的通信机制是怎样的呢？其实，这两个节点是通过一个 ROS 话题（Topic）来相互通信的，turtle_teleop_key 在这个话题上发布键盘输入的的消息，而 turtlesim 则订阅该话题以接收该消息。
ROS 提供了命令行图标工具，主要有：
rqt_graph:显示节点和话题的互动信息
rqt_plot:显示主题的数学数据随时间的变化
rqt_bag:用来观察 bag 文件的图像工具
rqt_deps:生成 ROS 依赖项的 pdf 文件
下面通过 rqt 功能包和 rostopic 命令来查看相关信息：
3.1 rqt 功能包
安装 rqt 功能包：
sudo apt-get install ros-indigo-rqt
通过 rqt_graph 来查看当前系统的运行情况。在新的标签页运行如下命令：
rosrun rqt_graph rqt_graph



如图所示，节点 turtlesim_node 和节点 turtle_teleop_key 正通过一个名为/turtle1/cmd_vel 的话题来相互通信。
除了使用 rqt_graph，还可以用 rqt_plot 来实时显示一个发布到某个话题上的数据变化图形。这里我们将使用 rqt_plot 命令来绘制正在发布到/turtle1/pose 话题上的数据变化图形。首先，在一个新终端中运行 rqt_plot 命令。
rosrun rqt_plot rqt_plot
在弹出的新窗口左上角的一个文本框里面可以添加需要绘制的话题。在里面输入/turtle1/pose/x 后按 enter 键或点击“+”号按钮，并对/turtle1/pose/y 重复相同的过程。就会在图形中看到 turtle 的 x-y 位置坐标图。

3.2 rostopic

使用 rostopic 命令工具能获取有关 ROS 话题的信息。

运行下面命令查看 rostopic 子命令：

rostopic -h

① rostopic echo：显示某个话题上发布的数据。

用法： rostopic echo [topic]

rostopic echo /turtle1/cmd_vel

此时终端没有任何数据，切换到 turtle_teleop_key 节点运行时所在的终端窗口，通过键盘方向键控制小海龟移动，然后再切换回原来的终端窗口。此时出现发布到话题上的数据。

② rostopic list：能够列出所有当前订阅和发布的话题。

先看一下 rostopic list 子命令需要哪些参数。运行命令：

rostopic list -h

使用 verbose 选项，以列出相关话题的详细信息。运行命令：

rostopic list -v

显示了有关所发布和订阅的话题的详细信息，其中方括号中表示的是话题的类型。

③ rostopic type：用来查看所发布话题的消息类型。

用法：rostopic type [topic]

运行如下命令：

rostopic type /turtle1/cmd_vel

上面的 geometry_msgs/Twist 即为话题/turtle1/cmd_vel 的消息类型，这在执行命令 rostopic list -v 时也有所体现。

下面用 rosmsg 命令来查看消息的详细情况：

rosmsg show geometry_msgs/Twist

④ rostopic pub：把数据发布到当前某个正在广播的话题上。通过此命令可以通过直接在终端发送命令来控制小海龟

用法：rostopic pub [topic] [msg_type] [args]

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

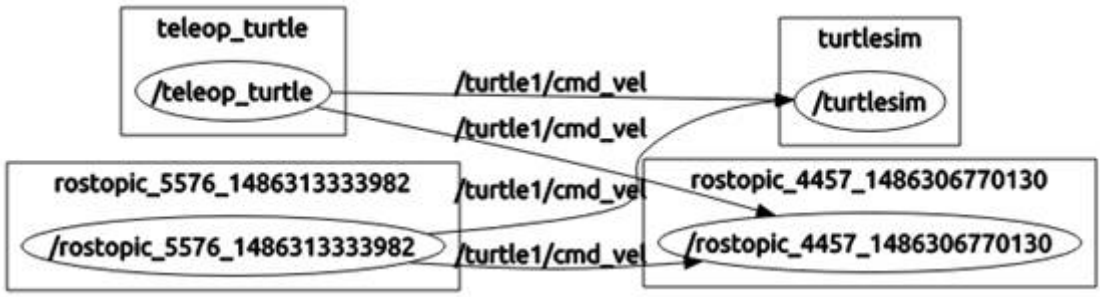
（注意格式， -- 两侧都有空格）以上命令会发送一条消息给 turtlesim（-1 表示命令只发布一次），这样小海龟就会以 2.0 大小的线速度和 1.8 大小的角速度开始移动。要想保持这个运动，turtle 需要一个 1Hz 稳定频率的命令流来保持移动状态。

使用 rostopic pub -r 命令发布一个稳定的命令流：

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

此时，我们再通过 rqt_graph 来查看当前系统的运行情况，即在新的标签页，运行：

rosrun rqt_graph rqt_graph



由上图可以看出，rostopic pub 发布消息到话题/turtle1/cmd_vel，rostopic echo 从该话题接收消息输出到终端，所以此时我们可以在新的终端用 rostopic echo 命令查看不断发布到话题的数据。

⑤ rostopic hz：命令可以用来查看数据发布的频率。

用法：rostopic hz [topic]

之前我们用 rostopic pub -r 命令发布一个 1HZ 的稳定命令流，现在我们用 rostopic hz 命令查看一下:

rostopic hz /turtle1/cmd_vel

理解 ROS 服务和参数

服务（services）是节点之间通讯的另一种方式。服务允许节点发送请求（request） 并获得一个响应（response）

1. 使用 rosservice

rosservice 可以很轻松的使用 ROS 客户端/服务器框架提供的服务。rosservice 提供了很多可以在 topic 上使用的命令，可以通过下列命令查看：

rosservice -h

1.1 rosservice list

输出可用服务的信息，如打开 turtlesim_node：

roscore

rosrun turtlesim turtlesim_node

rosservice list

可以查看到 turtlesim 节点提供的服务：重置（reset），清除（clear），再生（spawn），终止（kill）等。

1.2 rosservice type

输出服务类型

使用方法:

rosservice type [service]

我们来看看 clear 服务的类型:

rosservice type clear

结果显示 std_srvs/Empty，即服务的类型为空（empty),这表明在调用这个服务是不需要参数（比如，请求不需要发送数据，响应也没有数据）。

1.3 rosservice call

使用方法:

rosservice call [service] [args]

因为服务类型是空，所以进行无参数调用：

rosservice call clear

正如我们所期待的，服务清除了 turtlesim_node 的背景上的轨迹。

对于带参数的服务，比如再生（spawn）服务:

rosservice type spawn| rossrv show

根据结果这个服务可以在给定的位置和角度生成一只新的乌龟。名字参数是可选的，这里我们不设具体的名字，让 turtlesim 自动创建一个。

rosservice call spawn 2 2 0.2 ""

2. 使用 rosparam

rosparam 能够存储并操作 ROS 参数服务器（Parameter Server）上的数据。参数服务器能够存储整型、浮点、布尔、字符串、字典和列表等数据类型。rosparam 使用 YAML 标记语言的语法。

2.1 rosparam list

列出参数名，执行此命令

可以看到 turtlesim 节点在参数服务器上有 3 个参数用于设定背景颜色，下列通过 set 参数来改变背景色

2.2 rosparam set, rosparam get

用法：rosparam set [param_name]

rosparam get [param_name]

如使用 rosparam set 修改背景颜色的红色通道：

rosparam set background_r 150

上述指令修改了参数的值，现在我们调用清除服务使得修改后的参数生效：

rosservice call clear

再来使用 param get 查看参数服务器上的参数值——获取背景的绿色通道的值：

rosparam get background_g

特别地，如果想显示参数服务器上的所有内容，可以执行下面命令

rosparam get /

如果想存储这些信息以备今后重新读取。这通过 rosparam 的 dump, load 参数很容易就可以实现。

2.3 rosparam dump, rosparam load

使用方法:

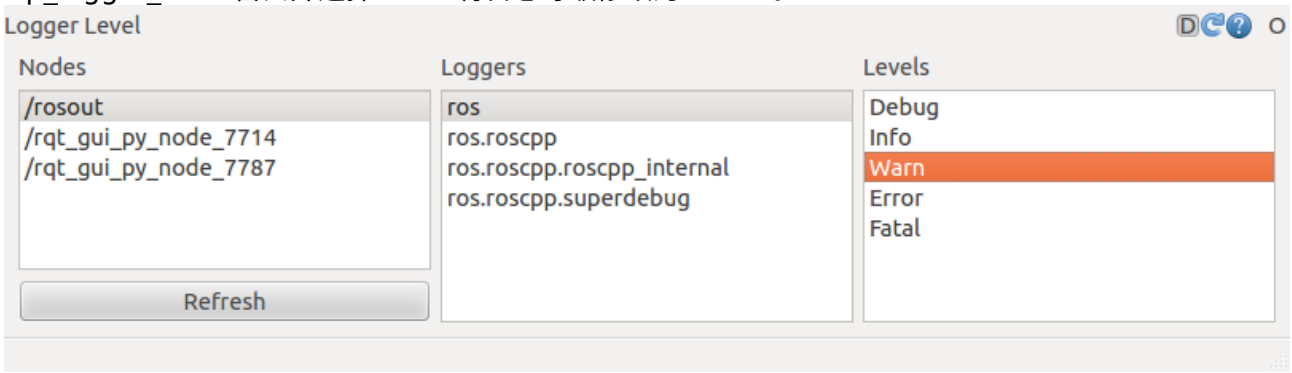
rosparam dump [file_name]

rosparam load [file_name] [namespace]

现在我们将所有的参数写入 params.yaml 文件：
rosparam dump params.yaml
甚至可以将 yaml 文件重载入新的命名空间，比如说 copy 空间:
rosparam load params.yaml copy
通过 rosparam get 命令来验证：
rosparam get copy/background_b

使用 rqt_console 和 roslaunch

1. 使用 rqt_console 和 rqt_logger_level
rqt_console 属于 ROS 日志框架(logging framework)的一部分，用来显示节点的输出信息。
rqt_logger_level 允许我们修改节点运行时输出信息的日志等级（logger levels）（包括 DEBUG、WARN、INFO 和 ERROR）。
现在让我们来看一下 turtlesim 在 rqt_console 中的输出信息，同时在 rqt_logger_level 中修改日志等级。在启动 turtlesim 之前先在另外两个新终端中运行 rqt_console 和 rqt_logger_level：
roscore
rosrun rqt_console rqt_console
rosrun rqt_logger_level rqt_logger_level
会看到弹出的两个窗口 rqt_console， rqt_logger_level。
现在在一个新标签页中启动 turtlesim：
rosrun turtlesim turtlesim_node
因为默认日志等级是 INFO，所以你会看到 turtlesim 启动后输出的所有信息，刷新一下 rqt_logger_level 窗口并选择 Warn 将日志等级修改为 WARN。



现在我们让 turtle 动起来并观察 rqt_console 中的输出：
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 0.0]'

日志等级说明
修改后的日志等级按以下优先顺序排列：
Fatal
Error
Warn
Info
Debug
Fatal 是最高优先级，Debug 是最低优先级。通过设置日志等级你可以获取该等级及其以上优先等级的所有日志消息。比如，这里将日志等级设为 Warn 时，会得到 Warn、Error 和 Fatal 这三个等级的所有日志消息。

现在让我们按 Ctrl-C 退出 turtlesim 节点，接下来我们将使用 roslaunch 来启动多个 turtlesim 节点和一个模仿节点以让一个 turtlesim 节点来模仿另一个 turtlesim 节点。
2. 使用 roslaunch
roslaunch 可以用来启动定义在 launch 文件中的多个节点。
用法：
roslaunch [package] [filename.launch]
先切换到 hello_test 程序包目录下：
cd ~/test/devel
source setup.bash # 执行 devel 里的 setup.bash 脚本将这个 workspace 的信息配置到环境里面
roscd hello_test
然后创建一个 launch 文件夹：

mkdir launch
cd launch

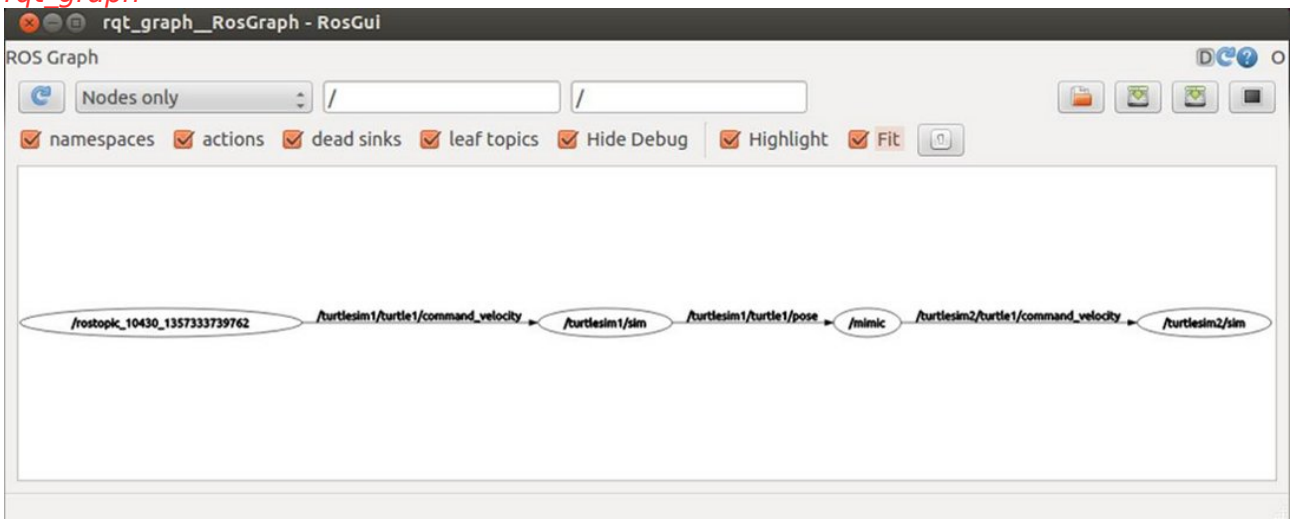
3. Launch 文件
现在我们来创建一个名为 turtlemimic.launch 的 launch 文件并复制粘贴以下内容到该文件里面：
#以 launch 标签开头以表明这是一个 launch 文件。
<launch>
#创建了两个节点分组并以'命名空间（namespace)'标签来区分，其中一个名为 turtulesim1，另一个名为 turtlesim2，两个组里面都使用相同的 turtlesim 节点并命名为'sim'。这样可以让我们同时启动两个 turtlesim 模拟器而不会产生命名冲突。
```
  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>
```

#启动模仿节点，并将所有话题的输入和输出分别重命名为 turtlesim1 和 turtlesim2，这样就会使 turtlesim2 模仿 turtlesim1。
```
  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>
```

#launch 文件的结束标签
</launch>

4. 使用 roslaunch
现在通过 roslaunch 命令来启动 launch 文件：
roslaunch hello_test turtlemimic.launch
现在将会有两个 turtlesims 被启动，然后我们在一个新标签页中使用 rostopic 命令发送速度设定消息：
rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
会看到两个 turtlesims 会同时开始移动，虽然发布命令只是给 turtlesim1 发送了速度设定消息。

也可以通过 rqt_graph 来更好的理解在 launch 文件中所做的事情：
rqt_graph



补充：使用 rosed 编辑 ROS 中的文件
rosed 是 rosbash 的一部分。利用它可以直接通过 package 名来获取到待编辑的文件而无需指定该文件的存储路径了。默认的编辑器是 vim。
使用方法:

rosed [package_name] [filename]
例子:
rosed roscpp Logger.msg
这个实例展示了如何编辑 roscpp package 里的 Logger.msg 文件。

创建 ROS 消息和 ROS 服务

1. 消息(msg)和服务(srv)介绍
消息(msg): msg 文件就是一个描述 ROS 中所使用消息类型的简单文本。它们会被用来生成不同语言的源代码。
服务(srv): 一个 srv 文件描述一项服务。它包含两个部分：请求和响应。
msg 文件存放在 package 的 msg 目录下，srv 文件则存放在 srv 目录下。
msg 文件实际上就是每行声明一个数据类型和变量名。可以使用的数据类型如下：

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

在 ROS 中有一个特殊的数据类型：Header，它含有时间戳和坐标系信息。在 msg 文件的第一行经常可以看到 Header header 的声明。下面是一个 msg 文件的样例，它使用了 Header，string，和其他另外两个消息类型。

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

srv 文件分为请求和响应两部分，由'---'分隔。下面是 srv 的一个样例：

```
int64 A
int64 B
---
int64 Sum
```

其中 A 和 B 是请求, 而 Sum 是响应。

2. 使用 msg
创建一个 msg
下面，我们将在之前创建的 package 里定义新的消息。
cd ~/test/devel
source setup.bash
roscd hello_test
mkdir msg
echo "int64 num" > msg/Num.msg
上面是最简单的例子——在.msg 文件中只有一行数据。当然，可以仿造上面的形式多增加几行以得到更为复杂的消息。

接下来，还有关键的一步：我们要确保 msg 文件被转换成为 C++，Python 和其他语言的源代码：
查看 package.xml, 确保它包含一下两条语句:
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
在构建的时候，需要"message_generation"。然而，在运行的时候，只需要"message_runtime"。
打开 CMakeLists.txt 文件，可以使用编辑命令直接打开，如上节补充的 rosed:
rosed hello_test CmakeLists.txt
进入 CMakeLists.txt 文件中，按 i 键进入插入模式。利用 find_packag 函数，增加对 message_generation 的依赖，这样就可以生成消息了。如下说示：

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

接着添加信息文件：

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

如果生成的消息类型依赖于其他消息（如 std_msgs），则需要添加：
Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
编辑完成后按 esc 键退出，按：键进入命令模式，输入 wq，按 enter 键后保存退出 vim 编辑器。
catkin_make 之后，在 msg 文件夹下的.msg 文件都会生成所有支持语言的代码。
引用和输出消息类型：
消息类型都被归属到与 package 相对应的域名空间下，在引用时需要加以限定：

```
#include <hello_test/Num.h>
hello_test::Num msg  (C++)
from hello_test.msg import Num
msg = Num()  (Python)
```

3. 使用 rosmsg
以上就是你创建消息的所有步骤。下面通过 rosmsg show 命令，检查 ROS 是否能够识消息。
使用方法:
rosmsg show [message type]
样例:

```
rosmsg show hello_test/Num
```

```
int64 num
```

在上边的样例中,消息类型是在指定的功能包 hello_test 中查找，也可以不指定功能包在全局查找：

```
rosmsg show Num
```

4. 使用 srv
创建一个 srv
在刚刚那个 package 中创建一个服务：

```
roscd hello_test
mkdir srv
```

这次我们不再手动创建服务，而是从其他的 package 中复制一个服务。 roscp 是一个很实用的命令行工具，它实现了将文件从一个 package 复制到另外一个 package 的功能。
使用方法:
roscp [package_name] [file_to_copy_path] [copy_path]
现在我们可以从 rospy_tutorials package 中复制一个服务文件了：

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

还有很关键的一步：我们要确保 srv 文件被转换成 C++，Python 和其他语言的源代码。
同样，跟 msg 文件类似，也需要在 package.xml 文件中做一些修改。查看上边的说明，增加同样的额外依赖项。然后在 CMakeLists.txt 文件中增加对 message_generation 的依赖，由于上边过程已经添加，这里直接可以进行下一步（对的, message_generation 对 msg 和 srv 都起作用）
接着添加服务文件：

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

5. 使用 rossrv
rossrv 与 rosservice 的区别：rosservice 显示实时信息，可以通过该命令查看正在发布到主题上的消息，而 rossrv 可以显示服务（Service）的数据结构定义，rosmsg 同理，可以显示消息（msg）的数据结构定义，但没有”rosmessage“。
下面通过 rossrv show 命令，检查 ROS 是否能够识该服务。
使用方法:
rossrv show <service type>
例子:
rossrv show hello_test/AddTwoInts

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

同样可以在全局搜索：
rossrv show AddTwoInts

6. msg 和 srv 都需要的步骤
接下来，在 CMakeLists.txt 中的 generate_messages 函数中去掉注释并附加上所有消息文件所依赖的那些含有.msg 文件的 package（这个例子无其他依赖,不要添加 roscpp），结果如下:

```
generate_messages(
  DEPENDENCIES
)
```

由于增加了新的消息，所以我们需要重新编译我们的 package：
In your catkin workspace
cd ../..
catkin_make
cd -
所有在 msg 路径下的.msg 文件都将转换为 ROS 所支持语言的源代码。生成的 C++头文件将会放置在 ~/test/devel/include/hello_test/。 Python 脚本语言会在 ~/test/devel/lib/python2.7/dist-packages/hello_test/msg 目录下创建。 lisp 文件会出现在 ~/test/devel/share/common-lisp/ros/hello_test/msg/ 路径下。

回顾
总结一下到目前为止我们接触过的一些命令：
rospack = ros+pack(age) : provides information related to ROS packages
rosstack = 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
rosmsg = 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

编写简单的消息发布器和订阅器 (C++)
1. 编写发布器节点
"节点(Node)" 是 ROS 中指代连接到 ROS 网络的可执行文件的术语。接下来，我们将会创建一个发布器节点("talker")，它将不断的在 ROS 网络中广播消息。
转移到之前在 catkin 工作空间所创建的 hello_test package 路径下:
cd ~/test/src/hello_test
（注意命令 roscd hello_test 无效，因为 test 工作空间并不在 ROS 环境变量中）

在 hello_test package 路径下创建 src 目录:
mkdir -p ~/hello/src/hello_test/src
这个目录将会存储 hello_test package 的所有源代码.
在 hello_test package 里创建 src/talker.cpp 文件，并编写如下代码：

```
#include "ros/ros.h"  //ros/ros.h是一个实用的头文件，它引用了ROS系统中大部分常用的头文件，使用它会使得编程很简便。
#include "std_msgs/String.h"  //引用std_msgs/String 消息，它存放在std_msgs package里，是由String.msg文件自动生成的头文件。
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker"); //初始化ROS,它允许ROS通过命令行进行名称重映射，第三个参数指定节点的名称—必须唯一
  ros::NodeHandle n; //为这个进程的节点创建一个句柄。第一个创建的NodeHandle会为节点进行初始化，最后一个销毁的会清理节点使用的所有资源。

  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
              //告诉master我们将要在chatter topic上发布一个std_msgs/String的消息。这样master就会告诉所有订阅了chatter topic的节点，将要有数据发布。
第二个参数是发布序列的大小。在这样的情况下，如果我们发布的消息太快，缓冲区中的消息在大于1000个的时候就会开始丢弃先前发布的消息。NodeHandle::advertise
() 返回一个 ros::Publisher对象,它有一个publish()成员函数可以让你在topic上发布消息
  ros::Rate loop_rate(10); // ros::Rate对象允许指定自循环的频率。
  int count = 0; //记录发送的消息数
  while (ros::ok()) //roscpp会默认安装一个SIGINT句柄，它负责处理Ctrl-C键盘操作—使得ros::ok()返回FALSE。
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();  //我们使用一个由msg file文件产生的‘消息自适应’类在ROS网络中广播消息。现在我们使用标准的String消息，它只有一个数据成
员"data"。当然也可以发布更复杂的消息类型。
    ROS_INFO("%s", msg.data.c_str()); //ROS_INFO和类似的函数用来替代printf/cout.
    chatter_pub.publish(msg);  //向所有连接到chatter topic的节点发送了消息

    ros::spinOnce(); //调用回调函数
    loop_rate.sleep();  //调用ros::Rate对象来休眠一段时间以使得发布频率为10hz
    ++count;
  }
  return 0;
}
```

整个代码的内容为：
初始化 ROS 系统
在 ROS 网络内广播我们将要在 chatter topic 上发布 std_msgs/String 消息
以每秒 10 次的频率在 chatter 上发布消息

2. 编写订阅器节点
在 hello_test package 目录下创建 src/listener.cpp 文件，并编辑如下代码：

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

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}  // 这是一个回调函数，当消息到达chatter topic的时候就会被调用。消息是以 boost shared_ptr指针的形式传输，这就意味着你可以存储它而又不需要复制数据

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

  ros::init(argc, argv, "listener");
  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback); //告诉master我们要订阅chatter topic上的消息。当有消息到达topic时，ROS就会调
用chatterCallback()函数。第二个参数是队列大小，以防我们处理消息的速度不够快，在缓存了1000个消息后，再有新的消息到来就将开始丢弃先前接收的消息。

  ros::spin();  // ros::spin()进入自循环，可以尽可能快的调用消息回调函数。如果没有消息到达，它不会占用很多CPU，所以不用担心。一旦ros::ok()返回FALSE，
ros::spin()就会立刻跳出自循环。这有可能是ros::shutdown()被调用，或者是用户按下了Ctrl-C，使得master告诉节点要shutdown。也有可能是节点被人为的关闭。

  return 0;
}
```

整个代码的内容为：
初始化 ROS 系统
订阅 chatter topic
进入自循环，等待消息的到达
当消息到达，调用 chatterCallback()函数

3. 编译节点
之前使用 catkin_create_pkg 创建了 package.xml 和 CMakeLists.txt 文件。 修改 CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_test)
find_package(catkin REQUIRED COMPONENTS
  roscpp message_generation std_msgs
)

add_message_files(
  FILES
  Num.msg
)

add_service_files(
  FILES
  AddTwoInts.srv
)

generate_messages(
  DEPENDENCIES
  std_msgs
)

catkin_package(
 CATKIN_DEPENDS roscpp message_runtime
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES}) ##生成可执行文件talker，默认存储到devel space目录,具体是在~/test/devel/lib/<package name>中
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})#生成可执行文件listener，默认存储到devel space目录,具体是在~/test/devel/lib/<package name>中

add_dependencies(listener hello_test_generate_messages_cpp) #为可执行文件添加对生成的消息文件的依赖
```

现在运行 catkin_make:
In your catkin workspace
cd ~/test
catkin_make

注意:如果是添加了新的 package，你需要通过—force-cmake 选项告诉 catkin 进行强制编译。

编写简单的消息发布器和订阅器 (Python)
cd ~/test/src/hello_test
mkdir scripts
cd scripts
gedit talker.py
talker.py 的内容如下：

```python
#!/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)
  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
```

gedit listener.py

listener.py 的内容如下：

```python
#!/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()
```

现在运行 catkin_make:
In your catkin workspace
catkin_make

测试消息发布器和订阅器
现在来测试上一节所写的消息发布器和订阅器。
1. 启动发布器
确保 roscore 可用，并运行：
roscore
在调用 catkin_make 后，运行自己的程序前需先确保 source 了 catkin 工作空间下的 setup.sh 文件：
cd ~/test
source ./devel/setup.bash
运行发布器节点 talker:
rosrun hello_test talker (C++)
rosrun hello_test talker.py (python)
可以看到不断的输出信息，表示发布器节点已经启动运行。现在需要一个订阅器节点来接受发布的消息。

2. 启动订阅器
运行订阅器节点 listener：
rosrun hello_test listener (C++)
rosrun hello_test talker.py (python)
（在新标签页中需要重新 source 以下 package 下 devel 文件夹中的 setup.bash）
可以看到源源的接收信息。

这样就完成了发布器和订阅器的测试。

编写简单的 Service 和 Client (C++)
1. 编写 Service 节点
创建一个简单的 service 节点("add_two_ints_server")，该节点将接收到两个整形数字，并返回它们的和，首先进入 package:
cd ~/test/src/hello_test
在 hello_test 包中创建 src/add_two_ints_server.cpp 文件，并复制粘贴下面的代码:

```
#include "ros/ros.h"
#include "hello_test/AddTwoInts.h" //hello_test/AddTwoInts.h是由编译系统自动根据我们先前创建的srv文件生成的对应该srv文件的头文件。

bool add(hello_test::AddTwoInts::Request  &req,
         hello_test::AddTwoInts::Response &res) //这个函数提供两个int值求和的服务，int值从request里面获取，而返回数据装入response内，这些数据类型都定义在srv文件内部，函数返回一个boolean值。
{
  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;
} //现在，两个int值已经相加，并存入了response。然后一些关于request和response的信息被记录下来。最后，service完成计算后返回true值。

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");  //service已经建立起来，并在ROS内发布出来。
  ros::spin();

  return 0;
}
```

2. 编写 Client 节点
在 hello_test 包中创建 src/add_two_ints_client.cpp 文件，并复制粘贴下面的代码:

```
#include "ros/ros.h"
#include "hello_test/AddTwoInts.h"
#include <cstdlib>

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

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<hello_test::AddTwoInts>("add_two_ints"); //这段代码为add_two_ints service创建一个client。ros::ServiceClient 对象待会用来调用service。
  hello_test::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]); //这里，实例化一个由ROS编译系统自动生成的service类，并给其request成员赋值。一个service类包含两个成员request和response。同时也包括两个类定义Request和Response。
  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;
  }  //这段代码是在调用service。由于service的调用是模态过程（调用的时候占用进程阻止其他代码的执行），一旦调用完成，将返回调用结果。如果service调用成功，call()函数将返回true，srv.response里面的值将是合法的值。如果调用失败，call()函数将返回false，srv.response里面的值将是非法的。

  return 0;
}
```

3. 编译节点
再来编辑一下 hello_test 里面的 CMakeLists.txt，文件位于~/test/src/hello_test/CMakeLists.txt，并将下面的代码添加在文件末尾：

```
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES}) #生成可执行文件add_two_ints_server
add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})  #生成可执行文件add_two_ints_client
add_dependencies(add_two_ints_client hello_test_gencpp)
```

现在运行 catkin_make 命令：
cd ~/test
catkin_make

测试简单的 Service 和 Client （C++）

现在来测试上一节所写的 Service 和 Client。
1. 运行 Service
确保 roscore 可用，运行：
roscore
在调用 catkin_make 后，运行自己的程序前需先确保 source 了 catkin 工作空间下的 setup.sh 文件：
cd ~/test

source ./devel/setup.bash
运行 Service
rosrun hello_test add_two_ints_server
在刚开始运行时，结果显示在 hello_test 目录下无法找到可执行文件 add_two_ints_server ，而可执行程序在调用 catkin_make 后，默认会被放在的 devel space 下的包目录下，默认路径 ~/test/devel/lib/share/<package name>。再次目录下查找果然没有，后发现是没有保存两个刚编写的 cpp 文件，保存后，再次 catkin_make，发现路径下出现对应的可执行文件，调用命令 rosrun 运行成功。如下所示：

```
[ INFO] [1487497022.945486868]: Ready to add two ints.
```

2. 运行 Client
现在，运行 Client 并附带一些参数：
rosrun hello_test add_two_ints_client 1 3 （在新标签页中需要重新 source 以下 package 下 devel 文件夹中的 setup.bash）
显示结果：

```
[ INFO] [1487497082.770102677]: Sum: 4
```

编写、测试简单的 Service 和 Client (Python)

1. 编写 Service 节点
在 hello_test 功能包 sricpts 文件夹下创建 add_two_ints_server.py 文件：

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

from hello_test.srv import *
import rospy

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

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

if __name__ == "__main__":
    add_two_ints_server()
```

修改权限，使文件可执行：
chmod +x ~/test/src/hello_test/scripts/add_two_ints_server.py

2. 编写 Client 节点
在 hello_test 包中创建 src/add_two_ints_client.py 文件，并复制粘贴下面的代码：

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

import sys
import rospy
from hello_test.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints') #This is a convenience method that blocks until the service named add_two_ints is available.
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts) #create a handle for calling the service
        resp1 = add_two_ints(x, y) #use this handle just like a normal function and call it
        return resp1.sum
    except rospy.ServiceException, 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))
```

chmod +x ~/test/src/hello_test/scripts/add_two_ints_client.py

3. 编译节点：
cd ~/test
cd ~/catkin_ws
catkin_make

4.测试节点：
rosrun &
. ~/test/devel/setup.bash
rosrun beginner_tutorials add_two_ints_server.py
在新终端：
. ~/test/devel/setup.bash
rosrun hello_test add_two_ints_client.py 4 5
可以看到两个终端输出的结果

```
maroon@freeman:~/test$ rosrun hello_test add_two_ints_client.py 4 5
Requesting 4+5
4 + 5 = 9
```

```
maroon@freeman:~/test$ rosrun hello_test add_two_ints_server.py
Ready to add two ints.
Returning [4 + 5 = 9]
```

录制与回放数据

1. 录制数据（通过创建一个 bag 文件）
如何记录 ROS 系统运行时的话题数据呢？可以将记录的话题数据会累积保存到 bag 文件中。
首先，执行以下命令：
roscore
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
录制所有发布的话题
先来检查看当前系统中发布的所有话题。要完成此操作打开一个新标签页并执行：
rostopic list -v

```
Published topics:
 * /turtle1/color_sensor [turtlesim/Color] 1 publisher
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
 * /rosout [rosgraph_msgs/Log] 2 publishers
 * /rosout_agg [rosgraph_msgs/Log] 1 publisher
 * /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
 * /rosout [rosgraph_msgs/Log] 1 subscriber
```

上面所发布话题部分列出的话题消息是唯一可以被录制保存到文件中的的话题消息，因为只有消息已经发布了才可以被录制。/turtle1/cmd_vel 话题是 teleop_turtle 节点所发布的命令消息并作为 turtlesim 节点的输入。而/turtle1/color_sensor 和/turtle1/pose 是 turtlesim 节点发布出来的话题消息。

现在开始录制。打开一个新的标签页，在终端中执行以下命令：
mkdir ~/bagfiles
cd ~/bagfiles
rosbag record -a
在这里先建立一个用于录制的临时目录，然后在该目录下运行 rosbag record 命令，并附加-a 选项，该选项表示将当前发布的所有话题数据都录制保存到一个 bag 文件中。
然后回到 turtle_teleop 节点所在的终端窗口并控制 turtle 随处移动 10 秒钟左右。
在运行 rosbag record 命令的窗口中按 Ctrl-C 退出该命令。现在检查看~/bagfiles 目录中的内容，应该会看到一个以年份、日期和时间命名并以.bag 作为后缀的文件。这个就是 bag 文件，它包含 rosbag record 运行期间所有节点发布的话题。

2. 检查并回放 bag 文件
现在我们已经使用 rosbag record 命令录制了一个 bag 文件，可以使用 rosbag info 检查看它的内容，在 bag 文件所在的目录下执行以下命令：
rosbag info 2017-02-19-19-54-35.bag
这些信息显示了 bag 文件中所包含话题的名称、类型和消息数量。

使用 rosbag play 命令可以回放 bag 文件以再现系统运行过程。首先在 turtle_teleop_key 节点运行时所在的终端窗口中按 Ctrl+C 退出该节点。让 turtlesim 节点继续运行。在终端中 bag 文件所在目录下运行以下命令：
rosbag play 2017-02-19-19-54-35.bag
默认模式下，rosbag play 命令在公告每条消息后会等待一小段时间（0.2 秒）后才真正开始发布 bag 文件中的内容。等待一段时间的过程可以通知消息订阅器消息已经公告了消息数据可能会马上到来。如果 rosbag play 在公告消息后立即发布，订阅器可能会接收不到几条最先发布的消息。等待时间可以通过-d 选项来指定。
另外，一个比较有趣的参数选项是-r 选项，可以通过设定一个参数来改变消息发布速率，如果执行：
rosbag play -r 2 2017-02-19-19-54-35.bag
应该会看到 turtle 的运动轨迹有点不同了，这时的轨迹应该是相当于以两倍的速度通过按键发布控制命令时产生的轨迹。
rosbag play -h 可以看到更多相关的回放选项，如下：
rosbag play 2017-08-07-10-48-00.bag -s 20 -u 60 -l
表示循环回放包 20~80s 的数据

3. 录制数据子集
当运行一个复杂的系统时，比如 PR2 软件系统，会有几百个话题被发布，有些话题会发布大量数据（比如包含摄像头图像流的话题）。在这种系统中，要想把所有话题都录制保存到硬盘上的单个 bag 文件中是不切实际的。rosbag record 命令支持只录制某些特别指定的话题到单个 bag 文件中，这样就允许用户只录制他们感兴趣的话题。
如果还有 turtlesim 节点在运行，先退出他们，然后重新启动：
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
在 bag 文件所在目录下执行以下命令：
rosbag record -O subset /turtle1/command_velocity /turtle1/pose
上述命令中的-O 参数告诉 rosbag record 将数据记录保存到名为 subset.bag 的文件中，同时后面的话题参数告诉 rosbag record 只能录制这两个指定的话题。然后通过键盘控制 turtle 随处移动几秒钟，最后按 Ctrl+C 退出 rosbag record 命令。
现在检查看 bag 文件中的内容：
rosbag info subset.bag
可以看到，只有指定话题的相关信息。

最后，rosbag record/play 命令的局限性
在前述部分中你可能已经注意到了 turtle 的路径可能并没有完全地映射到原先通过键盘控制时产生的路径上——整体形状应该是差不多的，但没有完全一样。造成该问题的原因是 turtlesim 的移动路径对系统定时精度的变化非常敏感。rosbag 受制于其本身的性能无法完全复制录制时的系统运行行为，rosplay 也一样。对于像 turtlesim 这样的节点，当处理消息的过程中系统定时发生极小变化时也会使其行为发生微妙变化，用户不应该期望能够完美的模仿系统行为。

roswtf 入门
1. 安装检查
roswtf 可以检查 ROS 系统并尝试发现问题，我们来试看：
roscd
roswtf

2.运行时检查（在有 ROS 节点运行时）
在这一步中，我们需要让 Master 运行起来：
roscore
现在按照相同的顺序再次运行以下命令：

roscd
roswtf
roswtf 发出警告说 rosout 节点订阅了一个没有节点向其发布的话题。在本例中，这正是所期望看到的，因为除了 roscore 没有任何其它节点在运行，所以我们可以忽略这些警告。

3. 错误报告
roswtf 会对一些系统中看起来异常但可能是正常的运行情况发出警告，也会对确实有问题的情况报告错误。如在 ROS_PACKAGE_PATH 环境变量中设置一个 bad 值，并退出 roscore 以简化检查输出信息:
roscd
ROS_PACKAGE_PATH=bad:$ROS_PACKAGE_PATH roswtf
由结果可以看到，roswtf 发现了一个有关 ROS_PACKAGE_PATH 设置的错误。
roswtf 还可以发现很多其它类型的问题。如果发现自己被一个编译或者通信之类的问题困扰的时候，可以尝试运行 roswtf 看能否帮你解决。

如何查找第三方的 ROS Packages
1. ROS Wiki 搜索栏
键入关键词查找,如查找 Arduino 会出现两个相关的功能包：rosserial_arduino 和 ros_arduino_bridge（需要翻墙）

2.使用 roslocate 命令
如查找 ros_arduino_bridge 功能包，在终端输入：
roslocate uri ros_arduino_bridge

```
maroon@freeman:~$ roslocate uri ros_arduino_bridge
Using ROS_DISTRO: indigo
Not found via rosdistro - falling back to information provided by rosdoc
https://github.com/hbrobotics/ros_arduino_bridge.git
```

3.浏览 ROS 软件目录
Browse Software

4.谷歌搜索
上边三种方法都没有查到相关库的话，最后可以尝试谷歌搜索。

参考：
核心 ROS 教程

补充：
ROS 命名
在 ROS 图级中，有四种类型的名称：
基本名：base　没有命名空间限定符，常用来初始化节点名称，实际是一种相对名的子类
相对名：relative/name
全局名：/global/name 以“/”开始的名称
私有名：~private/name 以“～”开始的名称
例子：

Node	Relative (default)	Global	Private
/node1	bar -> /bar	/bar -> /bar	~bar -> /node1/bar
/wg/node2	bar -> /wg/bar	/bar -> /bar	~bar -> /wg/node2/bar
/wg/node3	foo/bar -> /wg/foo/bar	/foo/bar -> /foo/bar	~foo/bar -> /wg/node3/foo/bar

ROS 节点中的任何名称都可以通过命令行重映射，语法格式为 name:=new_name，如:
rosrun rospy_tutorials talker chatter:=/wg/chatter

在参数进行匹配之前先确定其可变量项，如 foo:=bar，不仅可以匹配 foo，还可以匹配 /<node_namespace>/foo。
例子：

Node Namespace	Remapping Argument	Matching Names	Final Resolved Name
/	foo:=bar	foo, /foo	/bar
/baz	foo:=bar	foo, /baz/foo	/baz/bar
/	/foo:=bar	foo, /foo	/bar
/baz	/foo:=bar	/foo	/baz/bar
/baz	/foo:=/a/b/c/bar	/foo	/a/b/c/bar

参数管理器

通过命令行重映设时，私有参数将”~”变为”_”，如：
rosrun rospy_tutorials talker _param:=1.0

ROS 环境变量

有很多影响 ROS 工作的环境变量，总的来说环境变量对 ROS 的作用有以下三个层面：
1. 寻找功能包：这个最为常用，ROS_ROOT 和 ROS_PACKAGE_PATH 能够使得 ROS 定位 package 以及 stack；
作用于节点的运行：有很多环境变量影响节点的运行。ROS_MASTER_URI 告诉节点节点管理器的位置，ROS_IP 和 2. ROS_HOSTNAME 影响节点的网络地址，ROS_NAMESPACE 可以改变节点的命令空间，ROS_LOG_DIR 可以设置日志文件所在的文件夹，这些变量可以进行重映射；
3. 修正编译系统：ROS_BINDEPS_PATH,ROS_BOOST_ROOT,ROS_PARALLEL_JOBS 和 ROS_DISABLE 可以影响库文件的位置、编译过程以及编译文件。

许多系统只需要设置 ROS_ROOT, ROS_MASTER_URI, 和 PYTHONPATH。在安装 ROS 的时候它们会自动设置生成到 /opt/ros/ROSDISTRO/setup.bash，使用的时候 source 一下即可。
ROS_ROOT：设置 ROS 核心包的安装位置
ROS_MASTER_URI：定位 Master
PYTHONPATH:即时不使用 python 编程也要设置，因为 ROS 的很多基础工具依赖于 python
参考：http://wiki.ros.org/ROS/

功能包 catkin_make 出错提示缺少安装依赖项时可以运行：
rosdep install --from-path src

刚安装好 root 用户是没有密码的，没有密码就没法用 root 用户登录给 root 用户设置密码，因此直接输入命令：
sudo passwd
然后系统会提示输入密码，这时输入的密码就是 root 用户的密码了，设置完成之后就可以切换 root 用户登录了

以 root 用户运行 ROS
sudo rosrun [package_name] [node_name] 命令不识别，因此需要切换到 root 用户再运行 ROS。切换好后首次运行，root 并不识别 rosrun，因为 root 的环境变量中不包含 ros 库，因此运行：
gedit .bashrc
添加：
source /opt/ros/kinetic/setup.bash
同样的，添加 package 的环境变量：
source /home/maroon/imu/devel/setup.bash
注意，root 不识别主目录，因此需要全写为 */home/maroon*

ros Names
ros 中节点名称必须唯一，如果相同节点被运行了两次，则 roscore 会将旧的节点退出。
可以使用__**name** 来重映射节点名称，如下所示：
rosrun chat_pkg talker __name:=talker1
rosrun chat_pkg talker __name:=talker2

给 ros 节点传递参数
在 launch 文件中可以使用 **args** 参数来传递命令行参数给节点。
如在 launch 文件中，<node name=” ” pkg=” ” type=” “ args=”turtle1” output=”screen” />
节点程序接受来自命令行的参数：
string robot_name = string(argv[1]);

urdf 验证工具：check_urdf
check_urdf 07-physics.urdf
urdf 可视化工具：urdf_to_graphiz
urdf_to_graphiz 07-physics.urdf
evince physics （physics 为生成的文件名）

RVIZ 中的 urdf 模型 VS Gazebo 中的 sdf 模型

SDF vs URDF

- URDF can only specify the kinematic and dynamic properties of a single robot in isolation
  - URDF can not specify the pose of the robot itself within a world
  - It cannot specify objects that are not robots, such as lights, heightmaps, etc.
  - Lacks friction and other properties
- SDF is a complete description for everything from the world level down to the robot level

• To use a URDF file in Gazebo, some additional simulation-specific tags must be added to work properly with Gazebo.

Using URDF In Gazebo

- **Required**
  - An <inertia> element within each <link> element must be properly specified and configured
  - It's important to specify both the <visual> and <collision> tags for each link, otherwise Gazebo will treat the link as "invisible" to laser scanners and collision checking
- **Optional**
  - Add a <gazebo> element for every <link>
    - Convert visual colors to Gazebo format
    - Convert stl files to dae files for better textures
    - Add sensor plugins
  - Add a <gazebo> element for every <joint>
    - Set proper damping dynamics
    - Add actuator control plugins
  - Add a <gazebo> element for the <robot> element
  - Add a <link name="world"/> link if the robot should be rigidly attached to the world/base_link

• None of the elements within a <gazebo> element are required because default values will be automatically included
• There are 3 types of <gazebo> elements - for the <robot>, <link> tags, and <joint> tags

Spawn URDF Robots in Gazebo

- To launch your URDF-based robot into Gazebo you use ROS service all spawn method
- The **spawn_model** node in gazebo_ros package makes a service call request to the gazebo ROS node in order to add a custom URDF into Gazebo
- You can use this script in the following way:

```
$ rosrun gazebo_ros spawn_model -file `rospack find
r2d2_description`/urdf/r2d2.urdf -urdf -x 0 -y 0 -z 1 -model r2d2
```

Spawn XACRO Robots in Gazebo

- If your URDF is not in XML format but rather in XACRO format, you can make a similar modification to your launch file
- For example, to spawn a PR2 robot add the following to your launch file:

```
<!-- Convert an xacro and put on parameter server -->
  <param name="robot_description" command="$(find xacro)/xacro.py $(find
pr2_description)/robots/pr2.urdf.xacro" />

  <!-- Spawn a PR2 robot into Gazebo -->
  <node name="spawn_pr2_urdf" pkg="gazebo_ros" type="spawn_model"
args="-param robot_description -urdf -x 2 -model pr2" />
```