目錄

Introduction	1.1
Summary	1.2
CH01 Ubuntu14.04 安装 ROS Indigo	1.3
CH02 安装ROS和设置ROS环境	1.4
CH03 ROS 文件系统导航	1.5
CH04 创建一个 ROS package	1.6
CH05 编译一个 ROS package	1.7
CH06 理解 ROS Nodes	1.8
CH07 理解 ROS topic	1.9
CH08 service和parameter	1.10
CH09 rqt_console 和 roslaunch	1.11
CH10 使用rosed编辑ROS文件	1.12
CH11 创建一个ROS msg和srv	1.13
CH12 用C++语言写一个简单的发布者和订阅者	1.14
CH13 验证简单的发布者和订阅者	1.15
CH14 用C++语言写一个简单的service和client	1.16
CH15 验证简单的service和client	1.17
CH16 记录和重放数据	1.18
CH17 roswtf入门指南	1.19
CH18 ROS wiki 导航	1.20
CH19 接下来做什么	1.21

机器人操作系统 **ROS Indigo** 入门学习

本文分19章介绍机器人操作系统 ROS Indigo 的入门知识。

CH01 Ubuntu14.04 安装 ROS Indigo

1. 配置Ubuntu的软件中心：

配置Ubuntu要求允许接受 `restricted` , `universe` and `multiverse` 的软件源, 可以根据下面的链接配置:

```
https://help.ubuntu.com/community/Repositories/Ubuntu
```

2. 设置你的sources.list（软件源）：

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty
main" > /etc/apt/sources.list.d/ros-latest.list'
```

3. 设置你的密钥:

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

4. 安装:

首先确认你的Debian的软件包索引是最新的: (Debian 计划是一个致力于创建一个自由操作系统的合作组织。我们所创建的这个操作系统名为 Debian。Debian 系统目前采用 Linux 内核或者 FreeBSD 内核。)

```
sudo apt-get update
```

在ROS中有许多不同的函数库和工具,建议是完全安装,也可以根据自己的要求分别安装.完全安装时的工具包括ROS,rqt,rviz,robot-generic libraries,2D/3D simulators,navigation and 2D/3D，perception。

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

5. 初始化 rosdep:

```
sudo rosdep init
rosdep update
```

6. 设置环境:

添加ROS的环境变量,这样,当你打开你新的shell时,你的**bash**回话中会自动添加环境变量.

```
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
#使环境变量设置立即生效
source ~/.bashrc
```

7. 安装rosinstall:

```
sudo apt-get install python-rosinstall
```

rosinstall命令是一个使用的非常频繁的命令,使用这个命令可以轻松的下载许多ROS软件包。

CH02 安装ROS和设置ROS环境

1. 安装ROS

请参考前面的教程。

> 注意：当你用像 apt 这样的软件包安装管理器安装 ROS ，那么这些软件包用户是没有权利的去编辑的，当创建一个 ROS package 和处理一个 ROS package 时，你应该始终选择一个你有权限工作的目录作为工作目录。

2. 管理你的环境

在安装ROS的时候，你会看到提示source（命令）几个setup.*sh文件，或者甚至添加“sourcing”到你的shell启动脚本中。这是必须的，因为ROS依赖于结合使用shell环境的概念上。这使得开发依赖不同版本的ROS或者不同系列的package更加容易。

如果你在寻找或者使用你的ROS package上有问题，请确定的你的ROS环境变量设置好了，检查是否有ROS_ROOT and ROS_PACKAGE_PATH这些环境变量。

```
export | grep ROS
```

如果没有你需要使用‘source’一些setup.*sh文件。

环境设置文件时为你产生的，但是可以来自不同的地方：

- 使用package 管理器安装的ROS package提供setup.*sh文件；
- rosbuild workspace使用像rosws这样的工具提供setup.*sh文件； -setup.*sh文件在编译和安装catkin package时作为副产品创建。

> 注意：rosbuild和catkin是两种组织和编译ROS代码的方式，前者简单易用，后者更加复杂但是提供更多灵活性尤其是对那些需要去集成外部代码或者想发布自己软件的人。

如果你在ubuntu上使用apt工具安装ROS，那么你会在 `'/opt/ros/<distro>/'`

目录中有 `setup.*sh` 文件，你可以这样 `‘source’` 它们： `# source /opt/ros/<distro>/setup.bash`

这样的你每次打开你的新的shell都需要运行这个命令，如果你把 `source /opt/ros/indigo/setup.bash` 添加进 `.bashrc` 文件就不必要每次打开一个新的shell都运行这条命令才能使用ROS的命令了。

3. 创建ROS工作环境

对于ROS Groovy和之后的版本可以参考以下方式建立catkin工作环境：

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

可以看到在src文件夹中可以看到一个CMakeLists.txt的链接文件，即使这个工作空间是空的（在src中没有package），任然可以建立一个工作空间。

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

catkin_make命令可以非常方便的建立一个catkin工作空间，在你的当前目录中可以看到有build和devel两个文件夹，在devel文件夹中可以看到许多个setup.*sh 文件。启用这些文件都会覆盖你现在的环境变量，想了解更多可以查看文档catkin。在继续下一步之前先启动你的新的setup.*sh 文件。

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

为了确认你的环境变量是否被setup脚本覆盖了，可以运行一下命令确认你的当前目录是否在环境变量中：

```
echo $ROS_PACKAGE_PATH
```

输出：

```
/home/youruser/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indi
go/stacks
```

至此，你的环境已经建立好了，可以继续学习ROS文件系统了！

CH03 ROS 文件系统导航

1. 文件系统概述:

Packages: Packages是ROS代码的软件组织单元。每个Packages都包含函数库，可执行文件，脚本或者其他文件。

Manifest：一个Manifest是一个package的描述，用来指定packages之间的依赖，并且捕获packages的包括版本,维护者和协议等元信息。

2. 文件系统工具:

代码分布在许多ROS packages中，用命令行工具比如ls和cd去寻找起来非常的枯燥，这就是为什么提供ROS工具去帮助你的原因。

rospack

rospack可以看到许多packages的信息，这里我们讨论find选项，用来返回package的路径。用法如下：

```
# rospack find [package_name]
rospack find roscpp
```

返回：

```
/opt/ros/indigo/share/roscpp
```

roscd

roscd是rodbash套件的一部分，可以用来改变目录。用法如下：

```
# roscd [locationname[/subdir]]
roscd roscpp
```

可以用Unix命令打印绝对路径pwd

可以看到：

```
/opt/ros/indigo/share/roscpp
```

> 注意：roscd和其它ros工具只会在ROS_PACKAGE_PATH中指定了的目录中才能找到ROS packages，查看ROS_PACKAGE_PATH,可以用命令：

```
echo $ROS_PACKAGE_PATH
```

ROS_PACKAGE_PATH中有许多用冒号分开的路径，看起来像：

```
/home/ros/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indigo/stacks
```

可以加冒号增加路径。

roscd命令也可以进入packages或者stack的子目录

```
roscd roscpp/cmake
```

输出：

```
/opt/ros/indigo/share/roscpp/cmake$
```

roscd log

roscd log命令可以进入ROS储存log文件的文件夹。注意如果你至今还没有运行任何ros程序，那么会出现错误说命令不存在。

如果你之前已经运行过一些ros程序，可以运行：

```
roscd log
```

使用rosls

rosls是rosbash套件的一部分，可以列出一个packages中的目录。用法如下：

```
# rosls [locationname[/subdir]]
rosls roscpp_tutorials
```

返回：

```
cmake package.xml srv
```

Tab键补齐

Tab键可以补齐命令或者路径，不必给出完整的路径，双击tab键可以列出在当前目录下所有和你给出路径的前面部分相同的文件。

3. 回顾：

不难发现ros工具的名字都是在对应的UNIX命令的前面加上一个ros。

```
rospack = ros + pack(age)
roscd = ros + cd
rosls = ros + ls
```

4. 结语：

现在你可以在ROS中到天马行空了，让我们一起创建一个package吧。

CH04 创建一个 ROS package

1. 创建一个ROS Package：

这个教程包括使用roscreate-pkg或者catkin去创建一个新的package，以及使用rospack去列出package的依赖。

2. catkin Package由什么组成

一个package被认为是catkin packages必须满足这些要求：

a. 必须包含一个catkin compliant(编译) package.xml文件（提供关于package的元信息）；

b.必须包含一个使用catkin的CMakeLists.txt文件。Catkin metapackages(元package)必须有一个CMakeLists.txt样本文件；

c.在一个文件夹中不允许有超过两个的package（这就着意味着没有其他packages共享这个目录）。

这个简单的package可能像这个样子：

```
my_package/
    CMakeLists.txt
    package.xml
```

3. catkin工作空间中的Packages

推荐使用catkin packages 的方式是使用catkin工作空间，但是你也可以单独使用catkin建立packages。一般的工作空间看起来像这样：

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by c
atkin
    package1/
    CMakeLists.txt       -- CMakeLists.txt file for package1
    package.xml          -- Package manifest for package_1

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

4. 创建一个catkin package

这里将会用catkin_create_pkg脚本去创建一个新的catkin package。

首先去到你之前创建的工作空间目录:

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

用catkin_create_pkg 脚本创建一个做'beginner_tutorials'新package,它依赖于std_msgs，roscpp，和roscpy。命令：

```
catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

现在创建了一个 beginner_tutorials文件夹，包含有package.xml ， CMakeLists.txt 文件，其中部分填写了由你提供给catkin_create_pkg的信息。

catkin_create_pkg要求你提供一个package的名字和它所需要的依赖。

catkin_create_pkg在catkin/commands/catkin_create_pkg.中有对更加高级功能的描叙。

5. 建立一个catkin工作空间并且启用setup 文件

现在需要在catkin工作空间编译package。

```
cd ~/catkin_ws
catkin_make
```

执行该命令后会在devel文件夹下生成一个和
在/opt/ros/$ROSDISTRO_NAME（ROS版本名这里是/opt/ros/indigo）下相似的结构。

为把工作空间添加到ROS环境变量你需要执行：

```
# 使setup文件生效
source ~/catkin_ws/devel/setup.bash
```

6. package 的依赖

第一层依赖

用catkin_create_pkg时，一些package提供了依赖。这些第一层依赖可以通过rospack工具重现。

```
rospack depends1 beginner_tutorials
```

输出：

```
std_msgs
rospy
roscpp
```

列出了运行catkin_create_pkg命令时的一些依赖，这些依赖储存在package.xml文件中。

```
roscd beginner_tutorials
cat package.xml
```

输出：

```
<package>
    ...
    <buildtool_depend>catkin</buildtool_depend>
    <build_depend>roscpp</build_depend>
    <build_depend>rospy</build_depend>
    <build_depend>std_msgs</build_depend>
    ...
</package>
```

间接依赖

许多情况下，依赖也有它的依赖。比如：

```
rospack depends1 rospy
```

输出:

```
genpy
rosgraph
rosgraph_msgs
roslib
std_msgs
```

rospack可以显示所有递归的嵌套依赖。

```
rospack depends beginner_tutorials
```

输出:

```
cpp_common
rostime
roscpp_traits
roscpp_serialization
genmsg
genpy
message_runtime
rosconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
catkin
rospack
roslib
rospy
```

7. 定制你的package

这部分将教你怎样定制自己的package。

定制package.xml

package.xml包含在package中。

叙标签

首先更新描叙标签，可以根据你的爱好改变description，但是最好短一些并且能够概括package

```
  <!-- One maintainer tag required, multiple allowed, one person
 per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com"Jane Doe</maintai
ner> -->
  <maintainer email="user@todo.todo"user</maintainer>
```

维护者标签

这个非常重要因为他可以让其他人知道和谁去交流这个package，这个维护者的名字会作为标签，邮箱也应该填写：

```
  <maintainer email="you@yourdomain.tld"Your Name</maintainer>
```

协议标签

```
  <!-  - One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!-- BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>
```

选择一个协议填写。

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

通常用的一些协议是BSD，MIT，Boost Software License, GPLv2, GPLv3, LGPLv2.1, and LGPLv3。可以在这里读到一些其他的开源协议。（http://opensource.org/licenses/alphabetical）。 这里作为新手教程，我们选择BSD协议，因为ROS的其他核心部分已经遵从BSD协议了。

依赖标签

一系列标签描叙依赖。这些依赖被分为build_depend, buildtool_depend, run_depend, test_depend。

```
  <!-- The *_depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencie
s -->
  <!-- Examples: -->
  <!-- Use build_depend for packages you need at compile time: -
-> 
  <!--   <build_depend>genmsg</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>python-yaml</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>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
```

所有列出来的都是build_depend，除了之外，我们需要指定的所有依赖在build和run time时都是可用的，我们需要增加run_depend标签在每个依赖后面。

```
  <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>
```

最终的**package.xml:**

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

    <maintainer email="you@yourdomain.tld"Your Name</maintainer>
    <license>BSD</license>
    <url type="website">http://wiki.ros.org/beginner_tutorials</
url>
    <author email="you@yourdomain.tld"Jane Doe</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>
```

定制CMakeLists.txt

含有元信息的package.xml文件已经为你的package裁剪好了，接下来的教程将会讨论CMakeLists.txt文件。由catkin_create_pkg创建的CMakeLists.txt文件将会在下面的关于编译代码的教程中涉及。

CH05 编译一个 ROS package

1. 编译package

只要所有的package系统依赖都安装好了，就可以编译了。

如果你是用apt或者其他package管理器安装的ROS，那就应该已经有所有的依赖了。

记得使你的环境设置文件生效：

```
$ source /opt/ros/%YOUR_ROS_DISTRO%/setup.bash
# For Groovy for instance
$ source /opt/ros/groovy/setup.bash
```

2. 使用catkin_make

catkin_make命令行工具对与标准的catkin工作流程来说是一个非常方便的，你可以理解为它把调用cmake和编译结合起来了。用法如下：

```
# In a catkin workspace
catkin_make [make_targets][-DCMAKE_VARIABLES=...]
```

对于一个不熟悉标准Cmake流程的人来说，可以分解为以下几个步骤（但是实际上执行这些命令是没用的，它只是说明*CMake*是怎样工作的）：

```
# In a CMake project
mkdir build
cd build
cmake ..
make
# (optionally)
make install
```

这是每个CMake工程的过程，但是它可以在一个工作空间中编译多个catkin工程。在一个工作空间中编译多个catkin packages是这样操作的：

```
# In a catkin workspace
catkin_make
# (optionally)
catkin_make install
```

以上代码可以编译在src文件夹中的任何catkin 工程，这里参考了http://www.ros.org/reps/rep-0128.html，如果你的源代码不在src中，可以用my_src代替编译（如果出错，说明my_src不存在）：

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

CMake的更多用法参考http://wiki.ros.org/catkin/commands/catkin_make。

3. 编译你的package

想要编译你自己的代码的读者请看看之后的(C++))/(Python))教程，因为你也许需要修改CMakeList.txt。

经过上一个教程Creating a Package.，现在你已经有了一个catkin工作空间和一个叫做beginner_tutorials的新的package。进入catkin的工作空间，查看src文件：

```
cd ~/catkin_ws/
ls src
```

输出：

```
beginner_tutorials/ CMakeLists.txt@
```

在src文件夹中可以看到你在之前用catkin_create_pkg创建的叫做beginner_tutorials的文件，现在我们可以用catkin_make来编译这个package：

```
catkin_make
```

可以看到从cmake和make输出很多信息，大概是这个样子：

```
Base path: /home/ros/catkin_ws
Source space: /home/ros/catkin_ws/src
Build space: /home/ros/catkin_ws/build
Devel space: /home/ros/catkin_ws/devel
Install space: /home/ros/catkin_ws/install
\####
\#### Running command: "make cmake_check_build_system" in "/home
/ros/catkin_ws/build"
\####
-- Using CATKIN_DEVEL_PREFIX: /home/ros/catkin_ws/devel
-- Using CMAKE_PREFIX_PATH: /home/ros/catkin_ws/devel;/opt/ros/i
ndigo
-- This workspace overlays: /home/ros/catkin_ws/devel;/opt/ros/i
ndigo
-- Using PYTHON_EXECUTABLE: /usr/bin/python
-- Using Debian Python package layout
-- Using empy: /usr/bin/empy
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/ros/catkin_ws/build/test
_results
-- Found gtest sources under '/usr/src/gtest': gtests will be bu
ilt
-- Using Python nosetests: /usr/bin/nosetests-2.7
-- catkin 0.6.11
-- BUILD_SHARED_LIBS is on
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing 1 packages in topological order:
-- ~~  - beginner_tutorial
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'beginner_tutorial'
-- ==> add_subdirectory(beginner_tutorial)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ros/catkin_ws/build
\####
\#### Running command: "make -j4 -l4" in "/home/ros/catkin_ws/bu
ild"
\####
```

注意到catkin_make首先显示每个“space”的路径，这些路径在REP128和catkin/workspaces中有描述。值得注意的是，因为这个操作在你的工作空间生成了几个默认的文件夹。

```
ls
```

输出:

```
build
devel
src
```

build文件夹是在编译空间的默认位置，并且是调用cmake和make去配置和编译你的package的地方。devel文件夹是默认的devel空间，是你安装package之前可执行文件和库的所在地。

4. 结语

既然已经编译好ROS package了，让我们讨论一下ROS Node吧！

CH06 理解 ROS Nodes

这个教程将会介绍ROS图的概念并且会讨论roscoe,rosnode,和rosrun命令行工具。

1. 前提

在这个教程中我们会用到小型仿真器，请安装:

```
# <distro>是你的版本名字
sudo apt-get install ros-<distro>-ros-tutorials
```

2. 图概念的概论:

Nodes: node是使用ROS去和其它node通信的可执行文件。 **Messages**: ROS中订阅或者发布给topic的一种数据形式。 **Topics**: Nodes可以发布messages给一个topic，也可以订阅一个topic去接受它的messages。 **Master**: 为ROS提供名称服务（比如帮助*nodes*找到彼此）。 **rosout**: 相当于ROS中的stdout/stderr。 **roscore**: Master+rosout+parameter server*(参数服务之后会介绍)*。

3. Nodes

Node 不过是 ROS package中一个可执行文件。ROS Node 利用ROS用户库去和其他node进行通信。nodes也可以向topic发起发布或者订阅，nodes也可以提供或者使用一个service.

4. 用户库

用户库允许用不同语言编写的nodes之间进行通信：

```
rospy = python client library
roscpp = c++ client library
```

5. roscore

roscore是使用ROS时第一个要使用的工具。

```
roscore
```

输出：

```
... logging to /home/ros/.ros/log/add59068-aab1-11e4-99b0-6c71d9
2ff4a1/roslaunch-ros-K45VD-17626.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://ros-K45VD:42183/
ros_comm version 1.11.10

SUMMARY
========
PARAMETERS
* /rosdistro: indigo
* /rosversion: 1.11.10

NODES

auto-starting new master
process[master]: started with pid [17638]
ROS_MASTER_URI=http://ros-K45VD:11311/

setting /run_id to add59068-aab1-11e4-99b0-6c71d92ff4a1
process[rosout-1]: started with pid [17651]
started core service [/rosout]
```

如果roscore没有初始化，你也许会遇到网络配置问题。查看Network Setup - Single Machine Configuration解决.

如果roscore没有初始化并且说缺少权限，也许是~/.ros文件夹的用户是root，可以用一下命令递归改变文件夹的所有者：

```
sudo chown -R <your_username> ~/.ros
```

6. 使用rosnode

打开一个新的终端（前面那个roscore不要关闭），你的环境变量会重置，~/.bashrc被启用了，如果运行rosnode等命令时有问题时，需要添加一些环境变量setup文件去使这些命令到~/.bashrc中，或者手动使它们生效。

rosnode显示了关于正在运行的ros node的信息。rosnode list列出活动的node.

```
rosnode list
```

可以看到：

```
    这告诉我们这里只有一个node在运行。这个node总是在运行因为它会收集和记录node的调试信息。

    rosnode info命令可以返回特定node的信息

```shell
rosnode info /rosout
```

输出：
```

```
\-------------------------------------------------------------------
---------
Node [/rosout]
Publications:
* /rosout_agg [rosgraph_msgs/Log]

Subscriptions:
* /rosout [unknown type]

Services:
* /rosout/set_logger_level
* /rosout/get_loggers

contacting node http://machine_name:54614/ ...
Pid: 5092
```

接下来让我们看看其他的nodes。我们打算用rosrun去运行另一个node。

7. 使用rosrun

rosrun允许你在一个package中去用package的名字直接运行一个node（不需要知道package的路径）。用法如下：

```
rosrun [package_name][node_name]
```

因此我们可以利用rosrun去运行在turtlesim的package中的turtlesim_node

在一个新的终端中运行：

```
rosrun turtlesim turtlesim_node
```

你会在窗口看到：

在一个新终端中运行：

```
rosnode list
```

将会看到：

```
/rosout
/turtlesim
```

ROS一个强大的特点是可以在命令行重命名。

关闭窗口，再用Remapping Argument重新命名node

```
rosrun turtlesim turtlesim_node __name:=my_turtle
```

再次运行：

```
rosnode list
```

看到：

```
/rosout
/my_turtle
```

如果用Ctrl+c结束进程而不是关闭turtle的窗口那么在rosnode list时还会看到之前关闭的node，可以用rosnode cleanup清理。

再看看新的/my_turtle node,用rosnode ping命令去测试它是否正在运行：

```
rosnode ping my_turtle
```

输出：

```
rosnode: node is [/my_turtle]
pinging /my_turtle with a timeout of 3.0s
xmlrpc reply from http://aqy:42235/     time=1.152992ms
xmlrpc reply from http://aqy:42235/     time=1.120090ms
xmlrpc reply from http://aqy:42235/     time=1.700878ms
xmlrpc reply from http://aqy:42235/     time=1.127958ms
```

8. 回顾

roscore = ros +core:master（提供ros的命名服务）＋rosout（stdout/stderr）+ parameter server （参数服务之后会介绍）；

rosnode = ros +node ;ros用来获取关于node信息的工具；

rosrun = ros + run;从一个给定的package运行一个node.

9. 结语

既然你已经理解ros node 是怎样工作的了，再来看看ros的topic怎样工作的吧。

CH07 理解 ROS topic

这个教程介绍ROS topic和rostopic和rqt_plot命令行工具·

1. 建立

roscore

先在一个新的终端中运行roscore:

```
# 注意：只能运行一个roscore
roscore
```

tuetlesim

在新的终端中运行turtlesim:

```
rosrun turtlesim turtlesim_node
```

turtle 键盘遥控:

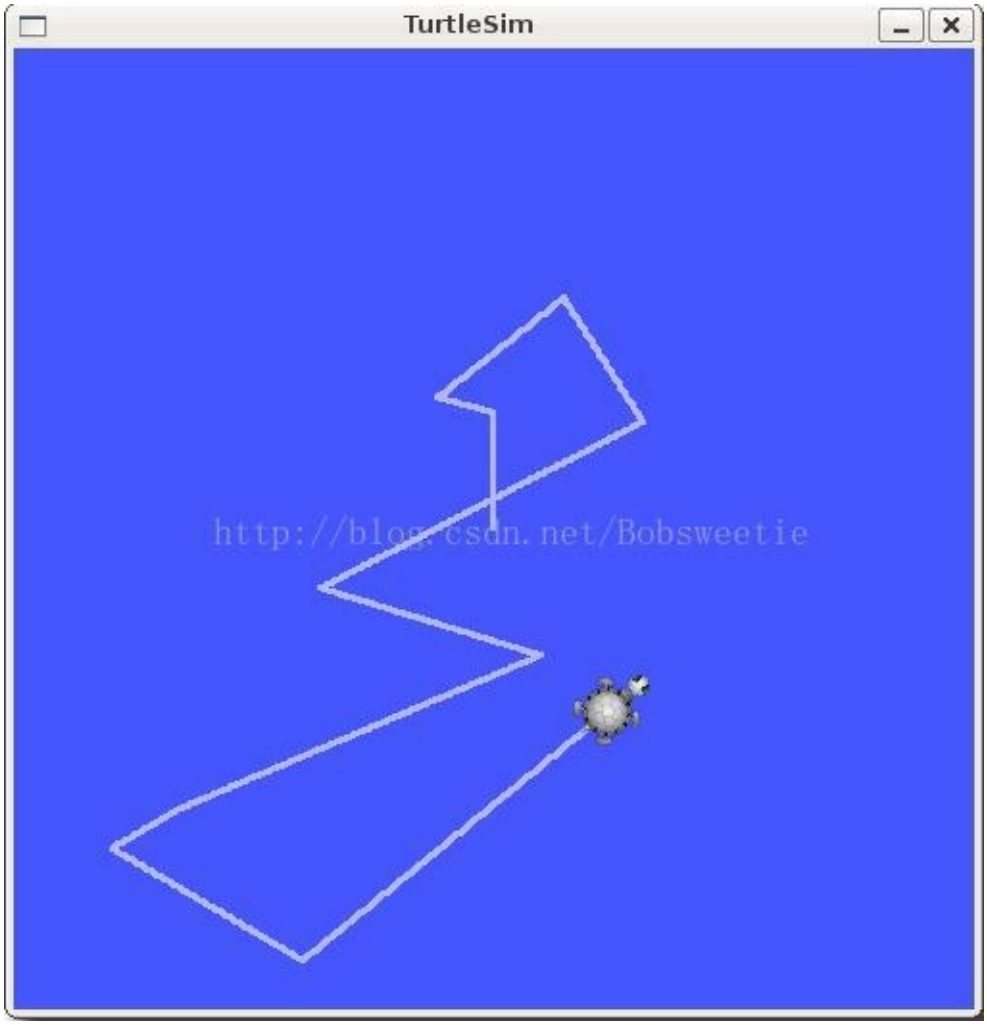
我们需要用一些东西去控制turtle，在新的终端运行：

```
rosrun turtlesim turtle_teleop_key
```

输出：

```
[ INFO] 1254264546.878445000: Started node [/teleop_turtle], pid
 [5528], bound on [aqy], xmlrpc port [43918], tcpros port [55936
], logging to [~/ros/ros/log/teleop_turtle_5528.log], using [rea
l] time

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



现在你可以用键盘上的方向键控制小乌龟了（保持键盘输入窗口在焦点）．再看看窗口后面发生了什么。

2. ROS Topic

turtlesim_node和turtle_teleop_key node之间用topic交流通信．turtle_teleop_key在这个topic上发布按键敲击，而turtlesim订阅同样的topic接受按键敲击．

让我们用rqt_graph显示现在正在运行的topic和nodes.

如果你是用electric或更早的版本，建议用rxygrah替代。

rqt_grah

rqt_grah创建了一个动态的图形显示系统上正在进行什么，rqt_grah是qpt package的一部分．运行：

```
# <distro>用你的ros版本名替代
sudo apt-get install ros-<distro>-rqt
sudo apt-get install ros-<distro>-rqt-common-plugins
```

再在新终端输入：

```
rosrun rqt_graph rqt_graph
```

你将会看到：

如果把鼠标放上去将会高亮，node是蓝色或者绿色，topic是红色．正如你所见到的一样，turtlesim_node and the turtle_teleop_key nodes 是通过叫做 `/turtle1/command_velocity` .的在通信．

rostopic

rostopic工具允许你从rostopic中获得信息，通过：

```
rostopic -h
```

可以获得等多命令选项:

```
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
```

让我们用这些命令去检验一下turtlesim.

rostopic echo

用法：

```
rostopic echo [topic]
```

允许我们看看turtle_teleop_key node. 发布的命令速度数据；

对于ROS Hydro和之后的版本，这个数据在发布在/turtle1/cmd_vel topic，在新终端，运行：

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

选择turtle_teleop_key终端，用方向键控制小乌龟，你将会看到:

```
linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0
---
linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0
```

让我们再来看看rqt_graph．点击左上角的刷新按钮显示新的node．你将会看到rostopic echo也订阅了turtle1/command_velocity　topic.

rostopic list

rostopic list会列出现在所有被订阅和发布的topic.

看看这个命令需要什么参数，运行:

```
rostopic list -h
```

```
Usage: rostopic list [/topic]

Options:
-h, --help            show this help message and exit
-b BAGFILE, --bag=BAGFILE
list topics in .bag file
-v, --verbose         list full details about each topic
-p                    list only publishers
-s                    list only subscribers
```

对于verbose选项：

```
rostopic list -v
```

显示详细的一列topic信息包括发布的，订阅的和它们的类型:

```
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 subscribe
```

3. ROS Messages

Topic上的通信通过在nodes之间发送messages来实现．对于发布者(turtle_teleop_key)和订阅者(turtlesim_node)之间的通信必须使用相同的message类型．这就意味着，topic的类型由发布的message类型决定．发布在topic上的message的类型可以由rostopic type来决定。

rostopic type

rostopic type返回任何topic发布的message类型。用法如下：

```
rostopic type [topic]
```

对于ROS Hydro 和之后的版本：

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

你会得到：

```
geometry_msgs/Twist
```

想得到详细信息运行：

```
rosmsg show geometry_msgs/Twist
```

```
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
```

既然我们已经知道turtlesim期待什么类型的message了，我们可以发布命令给小乌龟．

4. 继续rostopic

既然我们已经知道ros message 了，让我们一起用ros messages吧。

rostopic pub

rostopic pub把数据发布到正被广播的topic上。用法如下：

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

对于ROS Hydro和之后的版本：

```
rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0
.0, 0.0]' '[0.0, 0.0, 1.8]'
```

这会给turtle发布一个message告诉它线速度2.0,角速度1.8。

这个命令十分复杂，所以我们仔细看看它的每个参数。

对于ROS Hydro和之后的版本：

这个命令会发送messages给给定的topic: `rostopic pub`

这个选项使得rostopic只发布一条message然后退出： `1`

这是要发布给它信息的那个topic: `/turtle1/cmd_vel`

这是发布topic时的message类型： `geometry_msgs/Twist`

双虚线告诉选项剖析器接下来的参数都不是选择，以免把负号后面的参数当成参数选项: `--`

一个geometry_msgs/Twist 有两组由三个浮点元素组成得的向量：线性的和角度的．这样的话，'[2.0, 0.0, 0.0]'就是线性值x=2.0,y=0.0,z=0.0,而'[0.0, 0.0, 1.8]'就是角度的值x=0.0,y=0.0,z=1.8．这些参数遵从YAML 语法规则。更多信息查看YAML command line documentation. `'[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'`

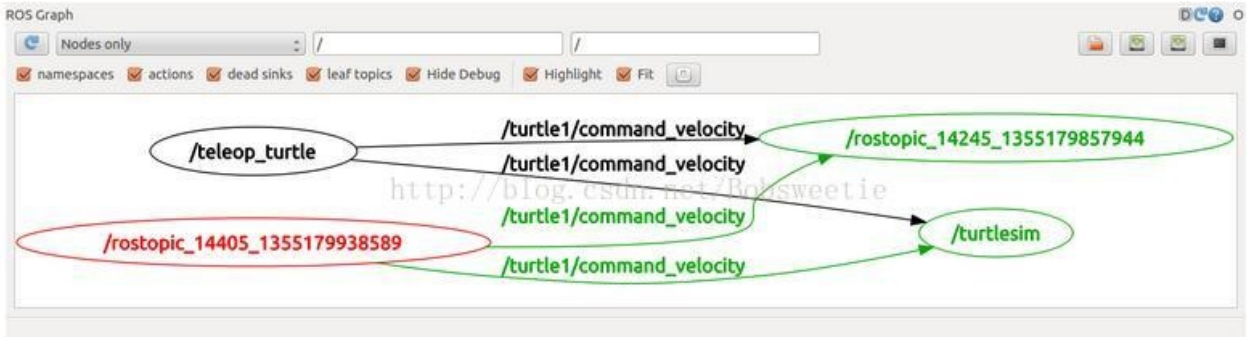
你也许注意到小乌龟已经停止了；这是因为小乌龟需要稳定的 1 HZ的命令流去保持运动。我们可以用rostopic pub -r命令发布一个稳定的命令流。

对于ROS Hydro和之后的版本：

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

这将以1HZ的速度去发布速度命令给速度topic。

再刷新一下rqt_graph，看到rostopic pub node 在和rostopic echo node 通信。



当你看到小乌龟在绕圈圈的时候可以在一个新的终端输入rostopic echo 命令看看turtlesim发布的数据。

rostopic hz

rostopic hz 报告数据发布的速度。用法如下：

```
rostopic hz [topic]
```

来看看turtkesim node 以多快的速度发布/turtle1/pose。

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

你会看到：

```
  - subscribed to [/turtle1/pose]
  average rate: 59.354
  min: 0.005s max: 0.027s std dev: 0.00284s window: 58
  average rate: 59.459
  min: 0.005s max: 0.027s std dev: 0.00271s window: 118
  average rate: 59.539
  min: 0.004s max: 0.030s std dev: 0.00339s window: 177
  average rate: 59.492
  min: 0.004s max: 0.030s std dev: 0.00380s window: 237
  average rate: 59.463
  min: 0.004s max: 0.030s std dev: 0.00380s window: 290
```

现在我们知道turtlesim大概以60HZ的速度发布数据。也可以用rostopic type和rosmsg获取关于topic进一步的信息。

对于ROS Hydro和之后的版本：

```
rostopic type /turtle1/cmd_vel | rosmsg show
```

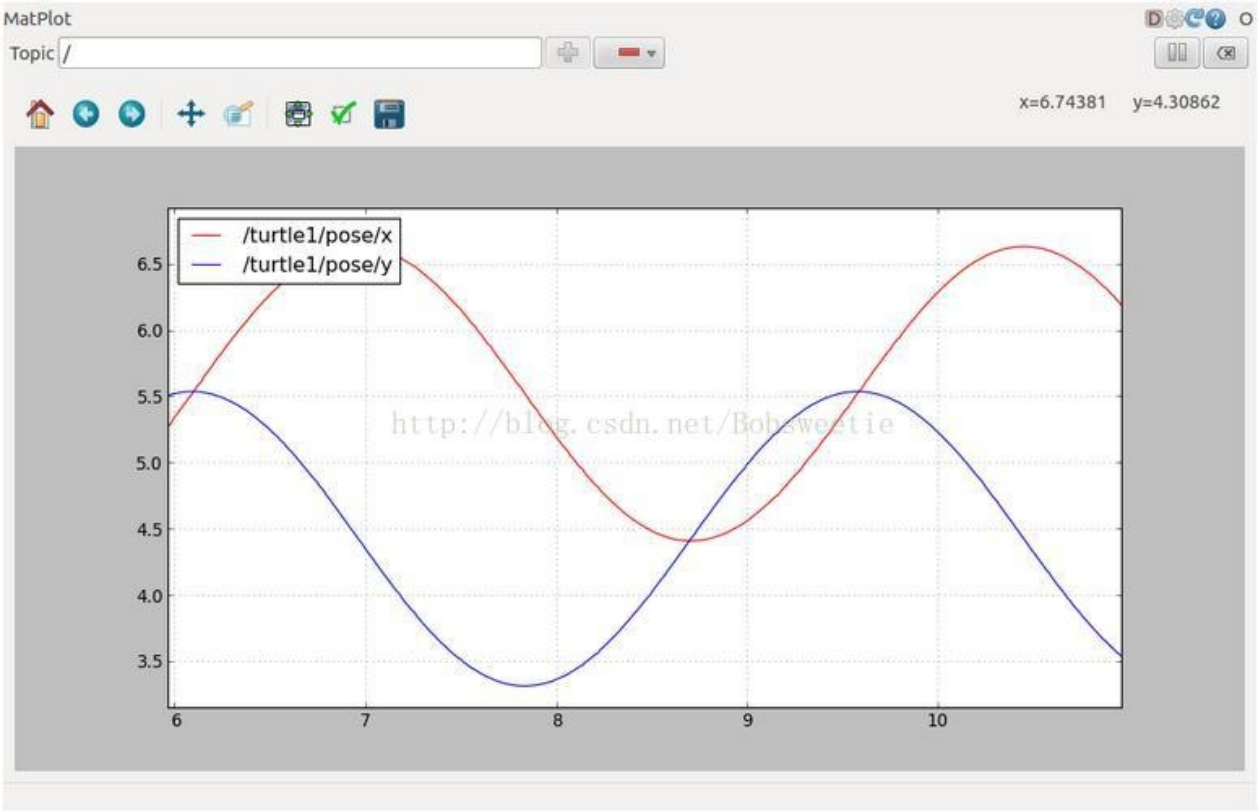
现在我们已经用rostopic检验了topic，让我们用其他工具看看turtlesim发布的数据把。

5. 使用rqt_plot （*electric*之前的版本用*rxplot*代替）

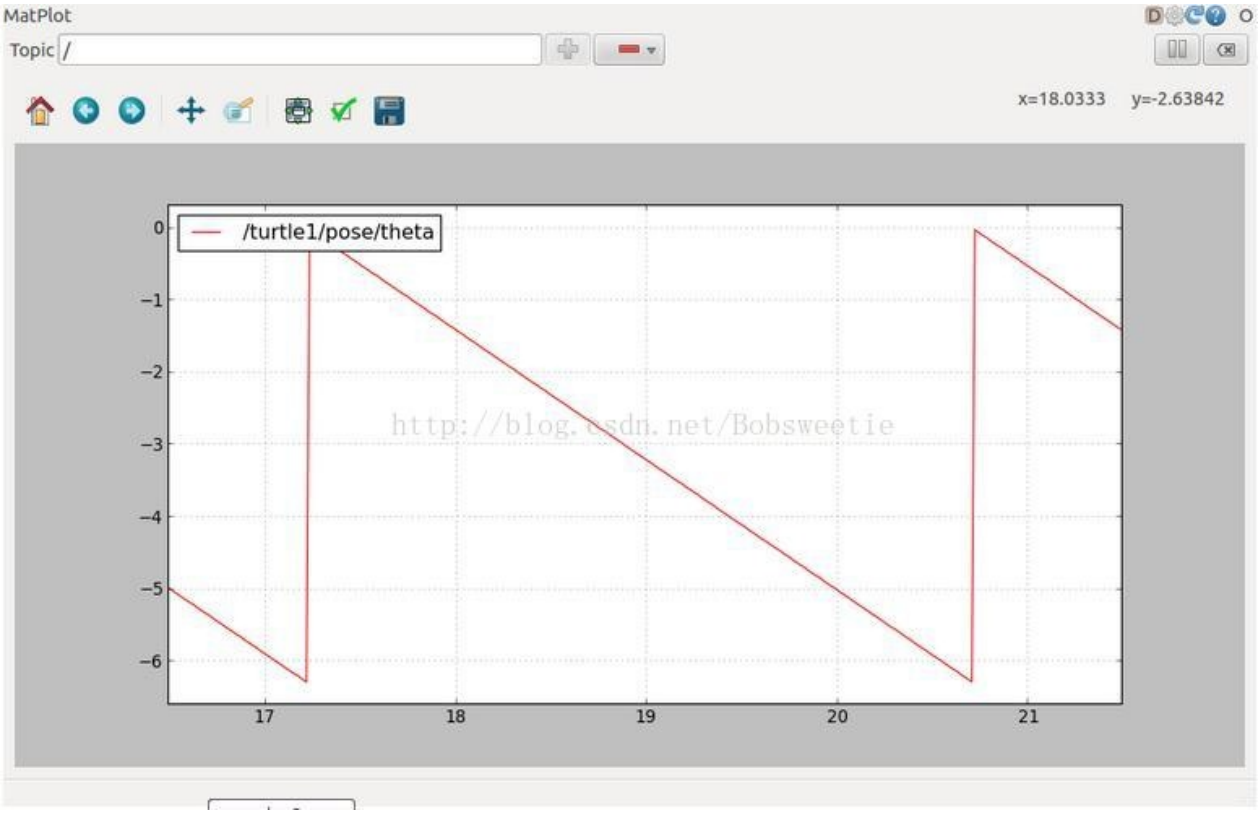
rqt_plot以时间轴的形式显示发布在topic上的数据。这里我们用它去显示发布在/turtle1/pose topic上的数据。首先：

```
rosrun rqt_plot rqt_plot
```

这时会弹出一个新的窗口，一个文本框出现再左上角，文本框是输入topicd的，这里我们输入/turtle1/pose/x会高亮原来不亮的按钮。按下它并且用/turtle1/pose/y重复同样的操作，就会看到x-y位置的图像出现。



点击负号按钮可以隐藏指定的topic。全部影藏，增加/turtle1/pose/theta，你会看到下面的图像:



这章节就是这些了，Ctrl+c 杀死 rostopic 终端但是让 turtlesim 继续运行。

6. 结语

既然你明白ros topic是怎样工作的了，让我们来看看service和 parameter是怎样工作的·

CH08 service和parameter

这个教程将介绍ROS service和parameter和命令行工具rosservice 和 rosparameter。

1. ROS Services

ROS Services是nodes之间进行通信的另一种方式，**service**允许**nodes**之间发送请求和接受应答。

2. 使用rosservice

rosservice可以轻易的附着在ROS的客户或者服务框上，rosservice 可以有许多命令可以在topic上使用，如下所示：

```
rosservice list         print information about active services
rosservice call         call the service with the provided args
rosservice type         print service type
rosservice find         find services by service type
rosservice uri          print service ROSRPC uri
```

rosservice list

```
rosservice list
```

打印：

```
/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
```

这个命令会显示出node可以提供9种服务，其中 `/rosout/get_loggers` 和 `/rosout/set_logger_level` 是和 `rosoutnode` 有关的·

rossercie type

让我们用rosservice type仔细看看这些服务,用法：

```
rosservice type [service]
```

clear服务的服务类型是：

```
rosservice type clear
```

打印：

```
std_srvs/Empty
```

这个服务是空的，这就意味着当调用这个服务时不带任何参数（比如，当发送请求时和接受回应时没有任何的数据）。

rosservice call

让我们用rosservice call调用这个服务吧，用法：

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

这里我们不用任何参数调用这个服务，因为这个服务是空的：

```
rosservice call /clear
```

这个命令确实清除了小乌龟的行走痕迹·

再看看有参数的service，看看service spawn的信息：

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

输出：

```
float32 x
float32 y
float32 theta
string name
---
string name
```

这个服务将产生另一个小乌龟，它的名字是可选的，我们自己不给它起名字，让turtlesim这个package给它取名字：

```
rosservice call spawn 2 2 0.2 ""
```

参数的分别是x.y的坐标和角度还有名字

这个服务调用返回新的小乌龟的名字：

```
name: turtle2
```

现在应该看起来像这样

3. 使用rosparam

rosparam允许你储存和操作在ROS parameter server上的数据，parameter server可以储存整形，浮点型，布尔型，字典型和链表型的数据．rosparam 使用**YAML**审定语言以符合语法．简单的例子：YAML看起来十分自然，１是整形，1.0是浮点型，one是字符串，true是布尔型，[1,2,3]是一列的整形，{a:b,c:d}是字典型，rosparam有许多命令可以运行用在parameters上，如下所示：

```
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
```

rosparam list

让我们看看现在参数服务器上是什么参数:

```
rosparam list
```

我们可以看到turtlesim node 的背景颜色有三个参数

```
/background_b
/background_g
/background_r
/roslaunch/uris/aqy:51932
/run_id
```

rosparam set和rosparam get

我们用rosparam set改变其中一个参数的值：

用法：

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

改变背景颜色中的红色比例：

```
rosparam set background_r 150
```

然后调用clear service使这个参数改变生效：

```
rosservice call clear
```

现在turtlesim看起来像这个样子：

让我们看看参数服务器上的其他参数的值：

```
rosparam get background_g
```

```
也可以用rosparam get/ 显示整个参数服务器的内容：

```shell
rosparam get /
```
```

```
background_b: 255
background_g: 86
background_r: 150
roslaunch:
  uris: {'aqy:51932': 'http://aqy:51932/'}
run_id: e07ea71e-98df-11de-8875-001b21201aa8
```

rosparam dump和rosparam load

如果想保存这些数据到文件，可以在其他时间重载，对于rosparam来说这很容易:

用法：

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

我们把所有参数都写入params.yaml文件：

```
rosparam dump params.yaml
```

你可以载入这些yaml文件到新的命名空间，比如copy:

```
rosparam load params.yaml copy
rosparam get copy/background_b
```

输出

```
255
```

CH09 rqt_console 和 roslaunch

这篇教程将介绍使用rqt_console和rqt_logger_level来调试以及使用roslaunch一次启动许多nodes.如果你使用ROS fuerte或者更早的版本，rqt不是十分完善，请查看这篇文章使用基于old rxthis page·

1. 前提 rqt和tuetlesim package

需要用到rqt和tuetlesim package·如果没有安装，请执行：

```
sudo apt-get install ros-<distro>-rqt ros-<distro>-rqt-common-pl
ugins ros-<distro>-turtlesim
```

> 注意：前面的教程中已经编译过rqt和tuetlesim这两个package了，如果不确定，再安装一次也无妨·

2. 使用rqt_console和rqt_logger_level

rqt_console附着在ROS logging框架上去显示nodes的输出结果。

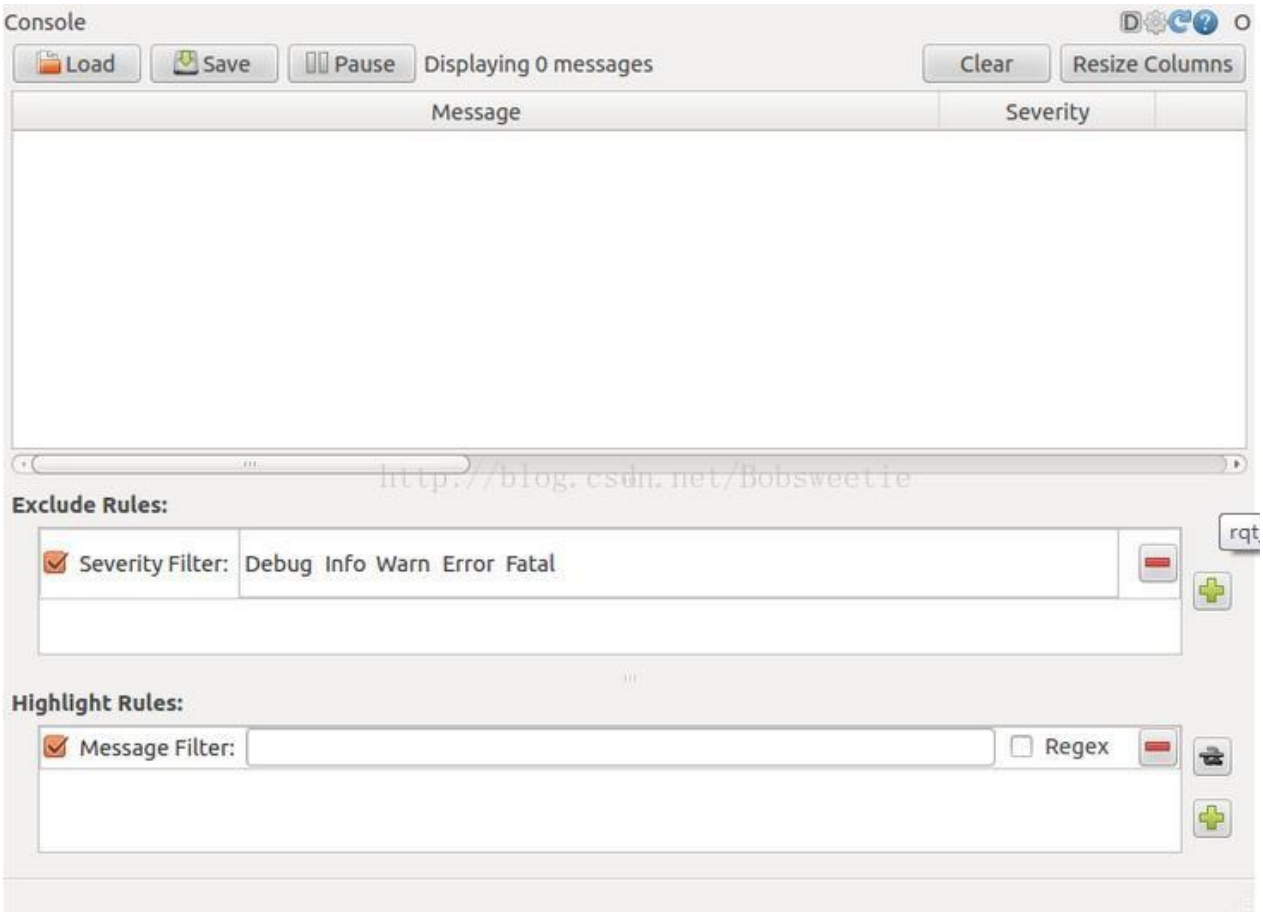
rqt_logger_level允许我们去改变nodes运行时候的信息显示级别（调试，警告，信息和错误）。

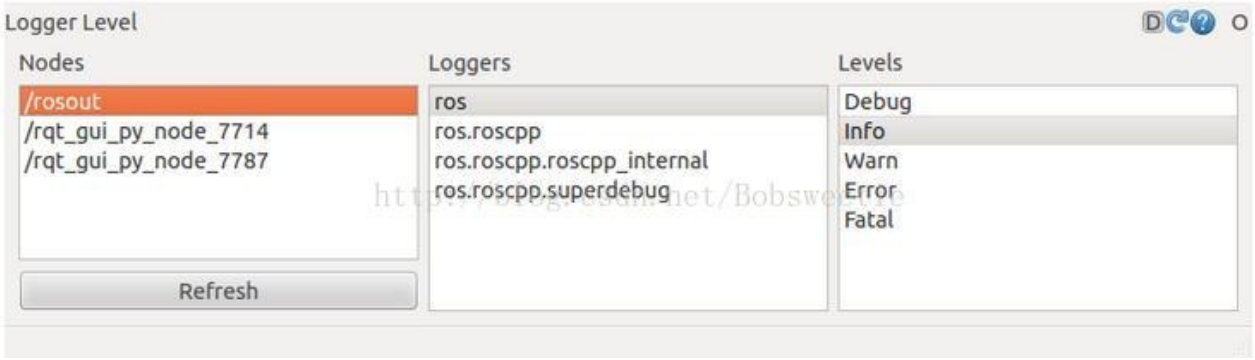
现在让我们看看turtlesim在rqt_console上的输出并且当我们使用turtlesim的时候变化logger级别·

在运行turtlesim之前，在两个新的终端中分别运rqt_consolerqt和rqt_logger_level:

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

你会看到两个弹出的窗口：

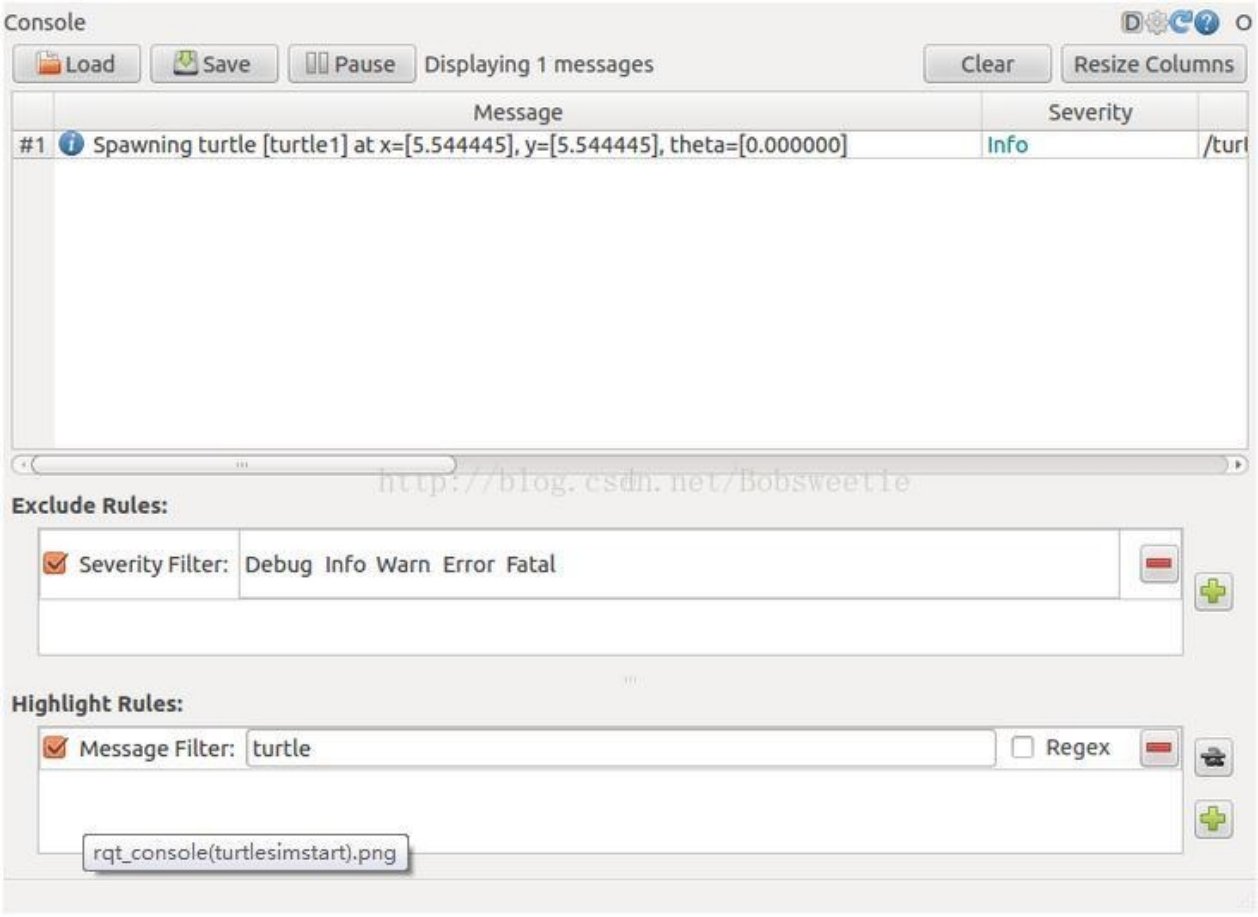




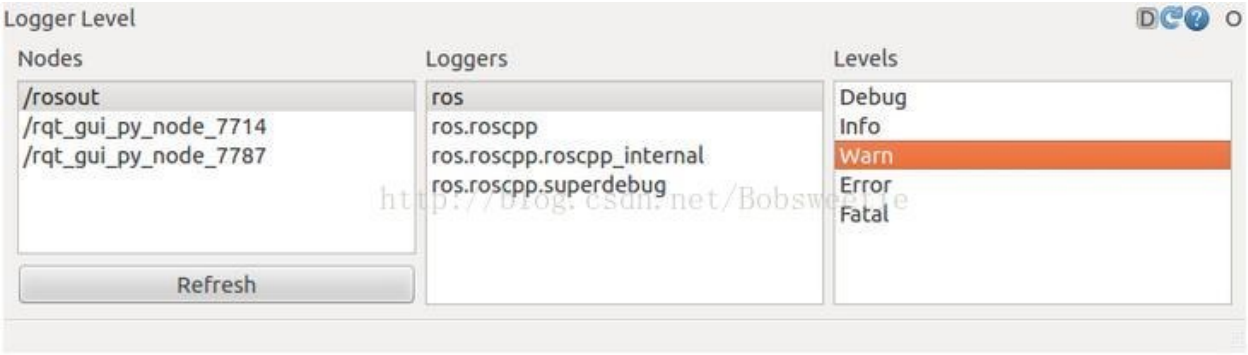
现在再在新的窗口中运行：

```
rosrun turtlesim turtlesim_node
```

因为默认的记录器级别是INFO所以你会看到turtlesim启动时发布的信息,大概是这个样子：



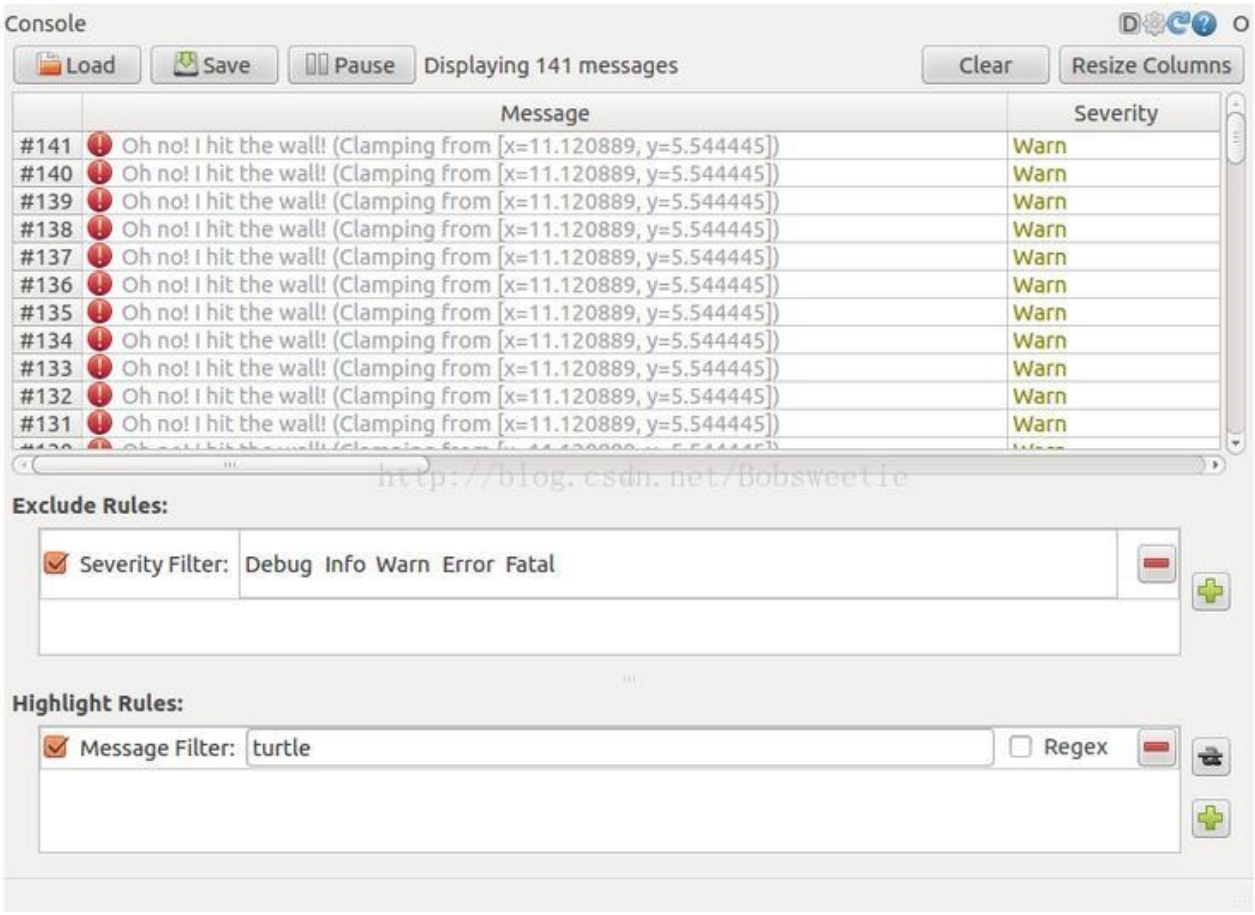
现在我们把记录器级别改为Warn，在rqt_logger_level窗口中刷新nodes并且选择Warn作为显示选项:



现在把小乌龟遥控到墙边看看在rqt_console上有什么显示：

对于ROS Hydro和之后的版本：

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



logger级别的概述

记录级别是按下列的的优先级别区分的：

```
Fatal
Error
Warn
Info
Debug
```

Fata的级别最高，Debug的级别最低．通过设置logger级别，你会得到　这个优先级别或更高级别的message.比如，通过设置级别为Warn,我们会得到所有的Warn,Error,和Fatal的记录消息.

先ctrl+c turtlesim，并且用roslaunch去生成更多的turtlesim nodes和一个mimicking node，让一个turtlesim去模仿另一个。

使用roslaunch

roslaunch按照launch文件中的定义去启动nodes·

用法：

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

首先进入我们之前创建和编译的beginner_tutorials package：

```
roscd beginner_tutorials
```

如果roscd说类似于：No such package/stack 'beginner_tutorials' 你需要启动环境变量设置的文件，像你之前在create_a_workspace教程末尾中做的一样。

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

创建一个launch目录：

```
mkdir launch
cd launch
```

launch文件

现在创建一个叫做turtlemimic.launch的launch文件并且把下面的东西粘贴在上面：

```
<launch>

  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

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

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>

</launch>
```

Launch文件的解释

现在我们把xml分解：

```
<launch>
```

我们用lauch标签开始launch文件，所以launch文件是这样鉴定的.

```
<group ns="turtlesim1">
  <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
</group>

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

我们用一个叫做sim的turtlesim node定义两个命名空间turtlesim1 和turtlesim2 ，这样我们就可以启动两个仿真器而不会有名字冲突了·

```
<node pkg="turtlesim" name="mimic" type="mimic">
  <remap from="input" to="turtlesim1/turtle1"/>
  <remap from="output" to="turtlesim2/turtle1"/>
</node>
```

我们通过把topic的输入和输出去重命名为turtlesim1和turtlesim２来定义mimic node(即messages在topic中从turtlesim1输入，从turtlesim２输出），这样重命名会导致turtlesim２模仿turtlesim1。

```
 </launch>
```

末尾的xml标签也是代表launch文件·

roslaunching

现在我们用roslaunch 启动launch文件：

```
roslaunch beginner_tutorials turtlemimic.launch
```

两个turtlesim会启动，在新的终端启动中并且发送rostopic命令,对于ROS Hyhydro:

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

你将会看到即使命令只是发布给turtlesim1但是两个小乌龟都开始运动·

可以用rqt_graph去更好的理解launch文件做了什么，运行rqt的主窗口选择rqt_graph：

```
rqt
```

或者直接：

```
rqt_graph
```

CH10 使用rosed编辑ROS文件

教程将展示怎样用rosed是编辑更容易·

1. 使用rosed.

rosed是rosbash套件的一部分·它可以使你通过package的名字直接编辑一个package中的文件而不用输入package的整个路径·

用法：

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

例子：

```
rosed roscpp Logger.msg
```

如果这个例子没有效果说明你没有安装vim工具·请参考Editor部分，如果你不知道怎样使用vim,点击这里click here。

如果package中的文件名不是唯一的，会列出一个菜单让你选择那个文件去编辑。

2. 使用rosed的时候用Tab补全

这样的话你就可以轻松的看到package中的可以编辑的文件而不需要知道它的具体名字。

用法：

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

Example:

```
rosed roscpp <tab><tab>
```

```
Empty.srv                   package.xml
GetLoggers.srv              roscpp-msg-extras.cmake
Logger.msg                  roscpp-msg-paths.cmake
SetLoggerLevel.srv          roscpp.cmake
genmsg_cpp.py               roscppConfig-version.cmake
gensrv_cpp.py               roscppConfig.cmake
msg_gen.py
```

3. 编辑

rosed的默认编辑工具是vim。ubuntu中默认安装了更加易上手的编辑器nano，你可以通过编辑你的~/.bashrc文件添加如下环境变量来启用它：

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

要设置默认编辑器可以在~/.bashrc文件中添加

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

> 注意：改变.bashrc文件只会在新的终端中生效，已经打开的终端不会看到变化（需要source一下）。

打开一个新的终端查看EDITOR是否已经定义;

```
echo $EDITOR
nano -w
```

或者

```
emacs -nw
```

CH11 创建一个ROS msg和srv

这篇教程将涉及怎样创建和编译msg和srv文件，以及怎样使用命令行工具rosmsg,rossrv 和roscp。

1. 介绍msg和srv

msg:msg文件是描叙ROS message字段的简单文本文件．它们用来为messsage产生不同程序语言的源代码。

Srv:一个srv文件描叙了一种服务．它由两部分组成：一个请求和一个响应．

msg文件储存在一个package的msg目录，而srv文件储存在srv目录．

msg只是每行有字段类型和字段名字的简单文本文件．可以使用的字段类型有：

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

ROS中有一种特殊的类型：Header,header包含一个时间戳和一个ROS中运用很普遍的坐标系信息。在一个msg文件中你会经常看到有 `Header header`：

这是一个使用一个Header，一个原始字符和两个其它msgs的msg的例子，

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

srv文件类似于msg文件，不同之处是它有两个部分：一个请求和一个应答。这两个部分由'—'线分隔．下面是一个例子：

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

上面的例子中，A和B是请求，而Sun是响应·

2. 使用msg

创建一个msg

让我们在之前的教程创建的package中创建一个新的msg.

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

上面例子的.msg文件只有一行·当然你也可以通过添加其它元素创建一个更加复杂的文件,每行一个元素，像这样：

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

还有一步要做·我们需要确定在msg文件能够转化为C++,Python源代码或者其它语言：

打开package.xml文件，确认有下面两行并且没有被注释掉·

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

> 注意：在编译的时候我们需要”message_generation”,而在运行的时候，我们只需要”message_runtime”。

在你最喜欢的编辑器中打开CMkeLists.txt（rosed是一个不错的选择）·

增加message_generation依赖到CMakeLists.txt中已经存在的find_package调用中，这样你就可以产生message.你只需要简单的增加message_generation到COMPONENTS的列表中，看起来大概是这个样子：

```
# Do not just add this to your CMakeLists.txt, modify the existi
ng text to add message_generation before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation
)
```

有时候你会发现即使你没有调用有所有依赖的find_package,工程编译也没错。这是因为catkin结合了你的所有工程，所以如果你之前的工程调用过find_package,那么你的配置会是一样的·但是忘记调用意味着你的工程在独自编译时会轻易的中断·

同样也要确认你输出message运行时的依赖·

```
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
  ...)
```

找到下面的代码段：

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

通过移除#号解除注释，用你的.msg文件替代Message*.msg文件，大概看起来是这个样子：

```
add_message_files(
  FILES
  Num.msg
)
```

通过手动添加.msg文件，我们可以保证CMake在你添加其他.msg文件后知道什么时候去配置你的工程·

现在我们必须保证generate_messages()函数能被调用·

对于ROS Hydro和之后的版本，需要解除下面三个注释：

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

看起来是这样：

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

现在你已经准备好在你的msg定义中产生源代码·如果你现在就想做，跳过下面的部分直接去msg和srv的一般步骤

使用rosmsg

这就是你创建一个msg文件需要做的·现在用rosmsg命令确认一下ROS可以看到这些。用法：

```
rosmsg show [message type]
```

例子：

```
rosmsg show beginner_tutorials/Num
```

你会看到：

```
int64 num
```

在之前的例子中，message类型由两部分组成：

```
beginner_tutorials    —定义message的package；
Num                    —msg Num的名字；
```

如果你不记得msg在哪个package里面，你可以列出package的名字：

```
rosmsg show Num
```

你会看到：

```
[beginner_tutorials/Num]:
Int64 num
```

3. 使用srv

创建一个srv

让我们用刚刚创建的package创建一个srv:

```
roscd beginner_tutorials
mkdir srv
```

我们会从其它package中复制已经存在的srv,不而是手动创建一个srv定义·

这样的话，从一个package复制文件到另一个package是roscp一个非常有用的工具。用法：

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

现在我们可以从rospy_tutorialspackage复制一个服务：

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

还有一步要做·我们需要保证srv文件转变为了C++,Python和其他语言代码，除非你已经做过了，否则打开package.xml,确认下面这两行语句没有被注释掉·

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

正如之前说的一样，编译的时候，我们需要”message_generation”,而在运行的时候，我们需要”message_tuntime”.

除非在之前的步骤中已经做过了，否则在CMakeLists.txt中添加messager_generatoin依赖：

```
# Do not just add this line to your CMakeLists.txt, modify the e
xisting line
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
 message_generation
)
```

（除了名字，message_generation可以供msg和srv使用）

正如message中的一样，在services中你同样也需要改变package.xml，所以看看上面添加要求的依赖：

移除#号解除下面的几行的注释：

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

用你的sercvice文件代替service*.srv文件：

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

现在你已经准备好从你的service定义中产生源文件了。如果你的想现在就做，跳过下面的步骤去msg和srv的一般步骤

使用rossrv

这些就是创建一个srv所有需要做的，让我们用rossrv show命令确认ROS可以看见：

用法：

```
rossrv show <message type>
```

例子：

```
rossrv  show beginner_tutorials/AddTwoInts
```

你会看到:

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

同rosmsg相似，你会看到service 文件没有指定package的名字：

```
rossrv show AddTwoInts
```

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

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

4. msg和srv的一般步骤

除非已经在前面的步骤做过，否则请在CMakeLists.txt中改变：

```
# generate_messages(
#   DEPENDENCIES
# #  std_msgs  # Or other packages containing msgs
# )
```

解除它的注释并且添加任何包含你的message（这里是std_msgs）使用的.msg文件的package,这看起来是这个样子：

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

既然你已经生成了一些新的messages我们需要重新生成package·

```
# In your catkin workspace
cd ../..
catkin_make
cd -
```

任何msg目录中.msg文件会产生所有支持的语言代码，C++message header文件会产生在 `~/catkin_ws/devel/include/beginner_tutorials/` 中，Python脚本会创建
在 `~/catkin_ws/devel/lib/python2.7/distpackages/beginner_tutorials/msg` 中，lisp文件 `~/catkin_ws/devel/share/common-lisp/ros/beginner_tutorials/msg/` 中。

messages形式的完整描叙在Message Description Language中。

5. 获取帮助

我们已经明白了许多ROS工具，要了解每一个命令要求的参数很困难，幸好，大部分的ROS工具提供了它们的帮助。使用：

```
rosmsg -h
```

你会看到一列的不同的rosmsg的子命令, Commands:

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

你也可以得到子命令的帮助：

```
rosmsg show -h
```

这里显示了rosmsg命令需要的参数：

```
Usage: rosmsg show [options] <message type>

Options:
  -h, --help  show this help message and exit
  -r, --raw   show raw message text, including comments
```

6. 回顾

让我们列举一下至今我们已经使用的命令：

```
rospack = ros + pack(age):提供ROS packages的信息
roscd = ros +cd :改变目录到ROS packages或者stack.
rosls = ros +ls:列出ROSpackage中的文件·
roscp = ros + cp :从一个package中拷贝文件，或者拷贝到一个 package中·
rosmsg = ros +msg:提供关于ROSmessage定义的信息·
rossrv = ros +srv :提供关于ROS messages定义的信息
catkin_make:编译一个ROS packages
rosmake = ros + make:编译一个ROS package(如果你不是在 catkin工作空间
中)
```

CH12 用C++语言写一个简单的发布者和订阅者

这个教程将会包含怎样用C++去写一个发布者和订阅者·

1. 写一个发布者Node

“Node”是连接在ROS网络中一个可执行单元的术语·这里我们创建一个会不断广播messages的发布者（“talker”）node.

改变目录到你之前创建的工作空间的beginner_tutorials package中：

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

代码

在beginner_tutorials packag目录中创建一个srv目录：

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

这个目录会包含所有beginner_tutorials package中的源文件·

在beginner_tutorials package中创建一个src/talker.cpp文件。并且把下面的代码粘贴上去”：

https://raw.github.com/ros/ros_tutorials/groovy-devel/roscpp_tutorials/talker/talker.cpp

```
/*
 * Copyright (C) 2008, Morgan Quigley and Willow Garage, Inc.
 *
 * Redistribution and use in source and binary forms, with or wi
thout
 * modification, are permitted provided that the following condi
tions are met:
 *   * Redistributions of source code must retain the above copy
right notice,
 *     this list of conditions and the following disclaimer.
```

```
 *   * Redistributions in binary form must reproduce the above c
opyright
 *     notice, this list of conditions and the following disclai
mer in the
 *     documentation and/or other materials provided with the di
stribution.
 *   * Neither the names of Stanford University or Willow Garage
, Inc. nor the names of its
 *     contributors may be used to endorse or promote products d
erived from
 *     this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRI
BUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIM
ITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTI
CULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONT
RIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLA
RY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCURE
MENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF A
DVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

/**
```

```
 * This tutorial demonstrates simple sending of messages over th
e ROS system.
 */
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that
 it can perform
   * any ROS arguments and name remapping that were provided at
the command line. For programmatic
   * remappings you can use a different version of init() which
takes remappings
   * directly, but for most command-line programs, passing argc
and argv is the easiest
   * way to do it.  The third argument to init() is the name of
the node.
   *
   * You must call one of the versions of ros::init() before usi
ng any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "talker");

  /**
   * NodeHandle is the main access point to communications with
the ROS system.
   * The first NodeHandle constructed will fully initialize this
 node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The advertise() function is how you tell ROS that you want
to
   * publish on a given topic name. This invokes a call to the R
OS
   * master node, which keeps a registry of who is publishing an
d who
   * is subscribing. After this advertise() call is made, the ma
```

```
ster
   * node will notify anyone who is trying to subscribe to this
topic name,
   * and they will in turn negotiate a peer-to-peer connection w
ith this
   * node.  advertise() returns a Publisher object which allows
you to
   * publish messages on that topic through a call to publish().
  Once
   * all copies of the returned Publisher object are destroyed,
the topic
   * will be automatically unadvertised.
   *
   * The second parameter to advertise() is the size of the mess
age queue
   * used for publishing messages.  If messages are published mo
re quickly
   * than we can send them, the number here specifies how many m
essages to
   * buffer up before throwing some away.
   */
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("ch
atter", 1000);

  ros::Rate loop_rate(10);
  /**
   * A count of how many messages we have sent. This is used to
create
   * a unique string for each message.
   */
  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and the
n publish it.
     */
    std_msgs::String msg;

    std::stringstream ss;
```

```
    ss << "hello world " << count;
    msg.data = ss.str();

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

    /**
     * The publish() function is how you send messages. The para
meter
     * is the message object. The type of this object must agree
 with the type
     * given as a template parameter to the advertise<>() call,
as was done
     * in the constructor above.
     */
    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

代码解释

现在我们分解代码：

```
#include “ros/ros.h”
```

ros/ros.h是一个非常方便的头文件它包含了最常用的ROS系统部分所必须的一些头文件：

```
#include “std_msgs/String.h”
```

这里包含了std_msgs package中的std_msgs/String message。这个头文件自动的从String.msg.file中产生：更多关于message的信息，请查看msg page。

```
Ros::init(argc,argv,”talker”);
```

初始化ROS.这个允许ROS通过命令行重新映射名字 –现在不重要.同样可以用来指定node的名字．在系统中Nodes的名字必须是唯一的．

名字必须是一个基本的名字(base name)，比如，不能有/在里面．

```
Ros::NodesHandle n;
```

为这个node创建一个handle．创建的第一个NodeHandle用来初始化node,最后一个销毁的NodeHandle会清除所有node占有的资源．

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

告诉master我们将要在chatter topic中要发布一个std_msgs::String类型的message，这就会让master告诉所有的nodes听取chatter这个topic，在这个tpoic上在我们将要发布数据．第二个参数是发布队列的大小．这样的话如果我们发布的太快，在开始丢弃之前的message前，它会最大缓冲是1000个messages．

```
NodeHandle::advertise()
```

返回一个ros::Publisher的对象，它有两个作用：

(1) 它允许你发布message到它创建的topic上的publish()

(2) 当它超出范围时，会自动解除广播．

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

ros::Rate对象会指定一个你想循环的频率．它会跟踪距离上次调用Rate::sleep(),有多长时间了，并且休眠正确长度的时间．

这里我们设置为10hz:

```
int count  =0;
while(ros::ok()){}
```

默认roscpp会安装一个SIGINT信号处理函数提供对ctrl+c的处理，ctrl+c会导致ros::ok()返回错误·

ros::ok()会返回错误如果:

(1) 接受到SIGINT(ctrl+c)

(2) 我们通过用另一个有同样名字的node网络。

(3) ros::shutdown()被应用的另一部分调用。

(4) 所有的 ros::NodeHandles都被摧毁了。

一旦ros::ok()返回错误，所有的ROS调用都会失败·

```
std_msgs::String msg;

std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

我们使用适应message的类在ROS上广播了一个message，通常从一个msg文件产生出来。其他复杂的数据类型也是是可以的，但是现在我们准备用标准的String message，它有一个成员：”data”。

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

现在实际上我们在向任何一个连接上的人广播这个message.

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

ROS_INFO和它的友元类都是用来替代printf/cout的。更多信息请看rosconsole documentation

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

对于这个程序调用ros::splinOnce()不是必要的，因为我们不会接受到任何回叫信号。然而，如果你打算为这个应用添加一个订阅，并且没有调用ros::splinOnce(),你绝不会得到回叫信号，所以还是添加的好。

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

现在使用ros::Rate对象去空耗掉剩下的时间以满足10hz的发布速度。

这里是步骤的简要描叙：

1. 初始化ROS系统
2. 广告给master我们将要发布std_msgs/String message到chatter topic上
3. 在发布message的时候循环以满足10次每秒钟

2. 写一个订阅者 Node

代码

在beginner_tutorials package中src目录下创建listener.cpp文件，并且把下面的代码粘贴进去：

https://raw.github.com/ros/ros_tutorials/groovy-devel/roscpp_tutorials/listener/listener.cpp

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

/**
 * This tutorial demonstrates simple receipt of messages over th
e ROS system.
 */
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

```
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that
 it can perform
   * any ROS arguments and name remapping that were provided at
the command line. For programmatic
   * remappings you can use a different version of init() which
takes remappings
   * directly, but for most command-line programs, passing argc
and argv is the easiest
   * way to do it.  The third argument to init() is the name of
the node.
   *
   * You must call one of the versions of ros::init() before usi
ng any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "listener");

  /**
   * NodeHandle is the main access point to communications with
the ROS system.
   * The first NodeHandle constructed will fully initialize this
 node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The subscribe() call is how you tell ROS that you want to r
eceive messages
   * on a given topic.  This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing an
d who
   * is subscribing.  Messages are passed to a callback function
, here
   * called chatterCallback.  subscribe() returns a Subscriber o
bject that you
   * must hold on to until you want to unsubscribe.  When all co
```

```
pies of the Subscriber
   * object go out of scope, this callback will automatically be
 unsubscribed from
   * this topic.
   *
   * The second parameter to the subscribe() function is the siz
e of the message
   * queue.  If messages are arriving faster than they are being
 processed, this
   * is the number of messages that will be buffered up before b
eginning to throw
   * away the oldest ones.
   */
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCall
back);

  /**
   * ros::spin() will enter a loop, pumping callbacks.  With thi
s version, all
   * callbacks will be called from within this thread (the main
one).  ros::spin()
   * will exit when Ctrl-C is pressed, or the node is shutdown b
y the master.
   */
  ros::spin();

  return 0;
}
```

代码解释

现在，我们把代码打断成一段段的，忽略上面已经分析过的部分

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

当一个新的message抵达chatter topic时这个回调函数会被调用．这个message以boost shared_ptr,的形式传递，这意味着你可以储存它，而不用担心它会在被删除，并且无需拷贝底层的数据．

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

在master启动的前提下订阅chatter topic，ROS会调用chatter Callback()函数只要一个新的message到达时．第二个参数是队列的大小，假设我们没有足够的能力去使发送message足够的快．这样的话，如果队列达到1000个messages,随着新的message的到来，我们会开始丢掉旧的message.

`NodeHandle::subscribe()` 会返回一个ros::Subscriber 对象，你必须坚持这个订阅对象直到你想取消订阅．当订阅对象被摧毁时，它会自动取消订阅chatter topic.

这里有不同版本的NodeHandle::subscribe()函数允许你指定一个类的成员函数，或者甚至任何被BoostFunction对象调用的东西，roscpp overview包含更多的信息．

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

ros::spin()进入了一个循环，调用message回调尽可能的快．即使这样，但不用担心，如果没有什么要做就不会占用许多CPU资源，ros::spin()会退出一旦ros::ok()返回错误，这意味着ros::shutdown()被调用了，不是被默认的Ctrl+c处理函数，然后master告诉我们去关机，就是被手动调用．

还有其他调用回调函数的方法，但是这里我们不关心它．roscpp_tutorials package中有一些关于这个的应用演示。roscpp overview　也包含更多的信息．

这里再一次简要的概括以上内容：

1. 初始化ROS系统
2. 订阅chatter topic Spin,等待message到来
3. 当一个message到来时，chatterCallback()函数被调用

3. 编译代码

在之前的教程中你用catkin_create_pkg去创建一个package.xml和一个CMakeLists.txt文件．

产生的这个CMakeLists.txt文件看起来应该像这样（保留了在Creating Msgs and Srvs中的修改和除去没有用的注释和例子）：

https://raw.github.com/ros/catkin_tutorials/master/create_package_modified/catkin_ws/src/beginner_tutorials/CMakeLists.txt

```
# %Tag(FULLTEXT)%
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs ge
nmsg)

## Declare ROS messages and services
add_message_files(DIRECTORY msg FILES Num.msg)
add_service_files(DIRECTORY srv FILES AddTwoInts.srv)

## Generate added messages and services
generate_messages(DEPENDENCIES std_msgs)

## Declare a catkin package
catkin_package()

# %EndTag(FULLTEXT)%
```

不要担心修改被注释掉的例子，只需要添加下面几行到你的CMakeLists.txt文件就好了：

```
include_directories(include ${catkin_INCLUDE_DIRS})

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_c
pp)
```

最后的CMakeLists.txt文件看起来应该像这样：

https://raw.github.com/ros/catkin_tutorials/master/create_package_pubsub/catkin_ws/src/beginner_tutorials/CMakeLists.txt

```
# %Tag(FULLTEXT)%
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs ge
nmsg)

## Declare ROS messages and services
add_message_files(FILES Num.msg)
add_service_files(FILES AddTwoInts.srv)

## Generate added messages and services
generate_messages(DEPENDENCIES std_msgs)

## Declare a catkin package
catkin_package()

## Build talker and listener
include_directories(include ${catkin_INCLUDE_DIRS})

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_c
pp)

# %EndTag(FULLTEXT)%
```

这会创建两个可执行的文件，talker和listener,默认是在的devel的package目录中，默认是在 `~/catkin_ws/devel/lib/share/<package name>` 。

> 注意：你应该为可执行目标添加依赖到message generation 目标：

```
add_dependencies(talker beginner_tutorials_generate_messages_cpp
)
```

这就可以保证package的message header在使用之前可以生成，如果你使用你工作空间的其他package产生的messages,你也需要为它们单独生成的目标添加依赖。

因为catkin平行编译所有的工程。如果是“Groovy”你可以用下面的变量去依靠所有必须的目标：

```
add_dependencies(talker ${catkin_EXPORTED_TARGETS})
```

如果你可以使用rosrun去调用他们，可以直接调用它们。他们不是放在'/bin' 中因为在安装package到你的系统时会破坏PATH.如果你希望你的可执行文件安装的时候在PATH上，你可以建立一个安装目标，查看：catkin/CMakeLists.txt

现在运行：

```
catkin_make
```

> 注意：如果你在添加一个新的pkg,也许需要告诉catkin去强制编译通过—force-cmake选项。参阅catkin/Tutorials/using_a_workspace#With_catkin_make。

CH13 验证简单的发布者和订阅者

1. 运行发布者

首先运行roscore:

```
rosrun
```

在尝试使用你的应用前，请确认你已经在调用catkin_make后启用了你工作空间的setup.sh文件．（最好将它写入~/.bashrc文件）

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

现在运行上一篇教程中创建的叫做"talker"的发布者：

```
rosrun beginner_tutorials talker      (C++)
```

你将会看到类似于：

```
[INFO] [WallTime: 1314931831.774057] hello world 1314931831.77
[INFO] [WallTime: 1314931832.775497] hello world 1314931832.77
[INFO] [WallTime: 1314931833.778937] hello world 1314931833.78
[INFO] [WallTime: 1314931834.782059] hello world 1314931834.78
[INFO] [WallTime: 1314931835.784853] hello world 1314931835.78
[INFO] [WallTime: 1314931836.788106] hello world 1314931836.79
```

2. 运行订阅者

发布者node在运行，现在需要运行从发布者接收message的订阅者：

运行上一个教程中创建的叫做"listneer"的订阅者：

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

你将会看到类似于这样的输出：

```
[INFO] [WallTime: 1314931969.258941] /listener_17657_13149319687
95I heard hello world 1314931969.26
[INFO] [WallTime: 1314931970.262246] /listener_17657_13149319687
95I heard hello world 1314931970.26
[INFO] [WallTime: 1314931971.266348] /listener_17657_13149319687
95I heard hello world 1314931971.26
[INFO] [WallTime: 1314931972.270429] /listener_17657_13149319687
95I heard hello world 1314931972.27
[INFO] [WallTime: 1314931973.274382] /listener_17657_13149319687
95I heard hello world 1314931973.27
[INFO] [WallTime: 1314931974.277694] /listener_17657_13149319687
95I heard hello world 1314931974.28
[INFO] [WallTime: 1314931975.283708] /listener_17657_13149319687
95I heard hello world 1314931975.28
```

至此，你已经验证了一个简单的发布者这和订阅者

CH14 用C++语言写一个简单的service和client

这篇教程包括怎样去用C++写一个service和一个client。

1. 写一个Service Node

这里我们会创建一个 `service(“add_two_ints_server”)node` ，它会接受两个整形数据并且返回它们的和。

进入你在之前教程中创建的catkin工作空间的beginner_tutorials package目录。

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

请确认你已经遵循之前教程的指示创建这个教程所需的service, creating the AddTwoInts.srv(请确认你已经选择了正确版本的编译工具在这个链接的网页的上面)。

代码

在beginner_tutorials package中src目录下创建一个名add_two_ints_server.cpp文件,并且把下面的代码粘贴上去

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

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)r
eq.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}

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.");
  ros::spin();

  return 0;
}
```

代码解释

现在我们分解代码:

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

`beginner_tutorials/AddTwoInts.h` 是由我们之前创建的srv文件中产生的头文件。

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

这个函数用来使两个整数的相加,它吸收在srv文件中定义的request和response类型,并且返回一个布尔量。

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

这里表示将两个整数的相加并且将结果储存在response中.然后一些关于request和response的信息被记录了.最后完成时返回真。

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

这里service在ROS上被创建和广播。

2. 写一个Client Node

代码

在beginner_tutorials package中src目录下创建一个叫add_two_int_client.cpp文件,并且把下面的代码粘贴上去:

```
#include "ros/ros.h"
#include "beginner_tutorials/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::ServiceCient client = n.serviceClient<beginner_tutorials:
:AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

代码解释

现在,我们分解代码

```
 ros::ServiceClient client = n.serviceClient<beginner_tutorials:
:AddTwoInts>("add_two_ints");
```

这里为add_two_ints service创建了一个client.ros::ServiceClient对象被用来之后调用service.

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

这里我们示例了一个自动产生的servcie类,并且给它的request成员分配值.一个service类包含两个成员,request和response.它也包含两个类的定义,Request和Response.

```
if (client.call(srv))
```

这里实际上调用了services.因为service的调用一直处于被阻塞状态,一旦调用结束它就会返回.如果service调用成功,call()函数会返回真并且srv.response的值会有效；如果调用没有成功,call()函数会返回错误并且srv.response的值会无效.

3. 编译源码

再一次编辑在 `~/catkin_ws/src/beginner_tutorials/` 目录下的CmakeLists.txt,把下面的东西添加到末尾:

https://raw.github.com/ros/catkin_tutorials/master/create_package_srvclient/catkin_ws/src/beginner_tutorials/CMakeLists.txt

```
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(add_two_ints_server beginner_tutorials_gencpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
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` , 默认会在你的devel空间目录中,默认是 `~/catkin_ws/devel/lib/share/<package name>` 。你可以直接调用它们或者使用rosrun去调用它们.它们不在'/bin' 中因为这

样当你安装你的package到你的系统时会破坏PATH.如果你希望你的可执行文件在PATH的安装时间,你可以建立一个目标,查看:catkin/CMakeLists.txt

现在运行catkin_make:

```
# In your catkin workspace
cd ~/catkin_ws
catkin_make
```

如果编译错误可能是下面的原因:

确认你已经遵循了之前的教程creating the AddTwoInts.srv的指示。

CH15 验证简单的service和client

这个教程将验证简单的service和client.

1. 运行Service

首先要运行:

```
roscore
```

然后运行service:

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

你会看到:

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

2. 运行Client:

运行client,加上适当的参数:

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

你会看到:

```
Requesting 1+3
1 + 3 = 4
```

至此你已经成功的运行了第一个service和client.

3. 关于Service和Client nodes的深入的例子

如果你想继续研究service和client,获取一些容易上手的;例子,可以看这里here.

一个简单的Servcie和Client的结合显示了message类型的客户化使用.Service nodes是用C++语言编写,而Client 可以是C++,Python和LISP.

CH16 记录和重放数据

这个教程教会告诉怎样将运行的ROS系统上的数据记录到一个.bag文件,然后再重放数据再产生相同的效果。

1. 记录数据(创建一个bag文件)

这部分将会指导你怎样从一个运行的ROS系统中记录topic的数据.这个topic数据会在一个bag文件中积累.

首先,执行下面的命令:

```
roscore
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

这会创建两个节点-可观则的tuetlesim和一个用方向键键盘控制tuetlesim中的小乌龟的node.如果你选择你启动turtle_keyboard的终端窗口,你会看到:

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

按下键盘上的方向键就可以控制屏幕上的小乌龟了.注意:必须让你的光标处在运行turtle_teleop_key node的终端窗口内。

1.1. 记录所有发布的topics

首先检查一下现在在系统上运行 的所有topics的列表.在新的终端中运行:

```
rostopic lsit -v
```

应该输出:

```
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
```

这些发布的topic只是可能被记录在数据记录文件上的message文件类型,因为只有发布的messages才能被记录。 `topic /turtle1/cmd_vel` 是由teleop_turtle发布的命令message,它被作为turtlesim进程的输入。 `/turtle1/color_sensor` 和 `/turtle1/pose` 是turtlesim发布的messages。

我们现在会记录发布的数据.打开新的终端,输入:

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

这里我们创建了一个临时目录记录数据,运行rosbag record命令时带着选项-a暗示所有发布的topics会在一个bag文件中聚集。

回到有turtle_teleop的那个窗口使小乌龟运动10秒钟左右。

在运行rosbag record的窗口摁Ctrl+c退出。 现在检验一下目录 `~/bagfiles` 中的内容.你会看到一个以年,数据,和时间为名,后缀是.bag的文件.这个bag文件记录了在rosbag record运行时任何node发布的topics。

2. 检验和回放bag文件

既然我们已经用rosbag record在一个bag文件中记录了许多数据，我们可以通过命令rosbag info和rosbag play来回放以检验数据。首先，我们看看bag文件中记录了些什么。我们可以用info命令检查bag的内容而不用回放它。在bag文件的目录下执行：

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

你应该看到:

```
path:        2014-12-10-20-08-34.bag
version:     2.0
duration:    1:38s (98s)
start:       Dec 10 2014 20:08:35.83 (1418270915.83)
end:         Dec 10 2014 20:10:14.38 (1418271014.38)
size:        865.0 KB
messages:    12471
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca8
4a]
             rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593
fb]
             turtlesim/Color     [353891e354491c51aabe32df673fb4
46]
             turtlesim/Pose      [863b248d5016ca62ea2e895ae5265c
f9]
topics:      /rosout                   4 msgs    : rosgraph_msg
s/Log   (2 connections)
             /turtle1/cmd_vel         169 msgs    : geometry_msg
s/Twist
             /turtle1/color_sensor   6149 msgs    : turtlesim/Co
lor
             /turtle1/pose           6149 msgs    : turtlesim/Po
se
```

这里显示了topic的名字和类型还有每个topic在bag文件中包含的messages数量。我们可以看到我们在topic输出中看到的被广播的topic，五个中的四个实际上是经过我们的记录间隔发布的。当我们运行带选项-a的rosbag record命令时它会记录所有nodes发布的所有messages。

下一步是重放bag文件，以在运行系统上产生相同的效果。首先在你运行turtle_teleop_key的终端用Ctrl+c杀死上个部分还在运行的teleop程序。让turtle继续运行。在产生原始bag文件的目录中运行以下命令：

```
rosbag play <your bagfile>
```

你会看到：

```
INFO: Opening 2014-12-10-20-08-34.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
```

在广播每个message之后，在它实际开始发布内容到bag文件按之前，默认rosbag会等待一个特定的时间段（0.2秒）。等待一段时间可以允许messages的订阅者注意到已经被广播了的message和将要被广播的messages。如果rosbag play广播完messages马上就发布messages，订阅者可能会接受不到前面几个发布的messages。等待可以用-d来特别指定。

最后 `topic /turtle1/cmd_vel` 会被发布，并且小乌龟会像你之前用键盘遥控的一样去移动。小乌龟移动的时间应该和你用rosbag record命令记录的时间相同。你也可以不在bag文件的开头开始返回，而是使用-s参数在不是bag文件开始的其他部分开始回放。最后一个有意思的选项是-r，它允许你通过一个特定的因子改变发布数据的速度。如果你执行：

```
rosbag play -r 2 <your bagfile>
```

你会看到小乌龟在有点不同的轨道上运动-这个轨道是你以两倍于之前的速度来发出键盘命令时会出现的轨迹。

3. 记录数据子集

当运行一个复杂的系统时，比如pr2的软件套装，可能会有上百的topics被发布。有些topics，比如摄像头图像流，可能会发布巨大量的数据，在这样的系统中写一个包含所有topics在单一的一个bag文件的日志文件是不实际的。rosbag record命令值支持在bag文件中记录一些特定的topics，允许用户值记录它们感兴趣的topics。

如果turtle nodes退出了，重启keyboard teleop启动文件：

```
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

在你的bag文件目录中运行：

```
rosbag record -O subset /turtle1/cmd_vel /turtle1/pose
```

-O参数告诉rosbag record在一个叫做subset。bag的文件中记录，并且topic参数导致rosbag record值记录这里两个topics。用键盘方向键控制小乌龟运动几秒钟，然后在ctrl+c结束rosbag record。

现在检查bag文件的内容（rosbag info subset.bag）。你会看到只有两个只有两个指定的topics：

```
path:        subset.bag
version:     2.0
duration:    12.6s
start:       Dec 10 2014 20:20:49.45 (1418271649.45)
end:         Dec 10 2014 20:21:02.07 (1418271662.07)
size:        68.3 KB
messages:    813
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca8
4a]
             turtlesim/Pose      [863b248d5016ca62ea2e895ae5265c
f9]
topics:      /turtle1/cmd_vel    23 msgs    : geometry_msgs/Twis
t
             /turtle1/pose      790 msgs    : turtlesim/Pose
```

rosbag record/play的局限

在之前的部分你也许注意到小乌龟的路径跟你用键盘控制的路径不是非常准确的吻合，即使大概的形状是一样的，但是小乌龟没有非常完全跟踪那个路径。原因是小乌龟跟踪的路径对系统中时间变化非常敏感。在系统中，在messages被roscore记

录和处理时以及使用rosplay 产生和处理messages时，rosbag复制运行系统的行为的能力是有限的。对于像turtlesim这样的nodes，在命令messages被处理时，微小的时间改变，都导致敏锐的行为改变。用户不要期望完美的模仿行为。

CH17 roswtf入门指南

roswtf的基本介绍

1. 检查你的安装

roswtf会尝试发现你的系统问题：

```
roscd
roswtf
```

你应该看到：

```
Stack: ros
================================================================================
Static checks summary:

No errors or warnings
================================================================================

Cannot communicate with master, ignoring graph checks
```

如果你你的安装正确英爱看到和上面类似的输出。这些输出告诉你：

"Stack: ros"：你的当前目录是什么决定着roswtf会做什么。这是告诉我们roswtf从ros stack启动。

"Static checks summary"：这是所有文件系统问题的报告。它告诉我们这里没有错误。

"Cannot communicate with master, ignoring graph checks"：roscore内核没有运行，所以roswtf不做任何在线检查。

2. 尝试在线运行

这里我们先运行roscore，激活Master：

再运行相同的序列：

```
roscd
roswtf
```

你应该看到：

```
Stack: ros
================================================================================
================
Static checks summary:

No errors or warnings
================================================================================
================
Beginning tests of your ROS graph. These may take awhile...
analyzing graph...
... done analyzing graph
running graph rules...
... done running graph rules

Online checks summary:

Found 1 warning(s).
Warnings are things that may be just fine, but are sometimes at
fault

WARNING The following node subscriptions are unconnected:
 * /rosout:
   * /rosout
```

在你roscore运行时，roswtf对你的graph进行了一些在线检查。取决于有多少ROS nodes正在运行，这可能会花费很长时间才能完成。正如你所见到的一样：

产生了一个警告：

```
WARNING The following node subscriptions are unconnected:
 * /rosout:
   * /rosout
```

roswtf警告说rosout node没有订阅了一个没有任何人发布的topic，这个警告是正常的，可以忽略，因为iexianzai没有东西在运行。

3. 错误

roswtf会警告你系统中正常的东西看起来可疑。也可能报告出它知道有错误的问题。

```
roscd
ROS_PACKAGE_PATH=bad:$ROS_PACKAGE_PATH roswtf
```

这时会看到：

```
Stack: ros
================================================================================
Static checks summary:

Found 1 error(s).

ERROR Not all paths in ROS_PACKAGE_PATH [bad] point to an existing directory:
 * bad

================================================================================

Cannot communicate with master, ignoring graph checks
```

正如你所见，roswtf会给我们关于ROS_PACKAGE_PATH设置的错误

roswtf可以发现许多类型的错误。如果你被一个编译或者通信难住了，可以试试这个命令看看它能否给你制定正确的方向。

CH18 ROS wiki 导航

这个教程将会讨论ROS wiki的版面设计和怎样去找到你想要知道的。

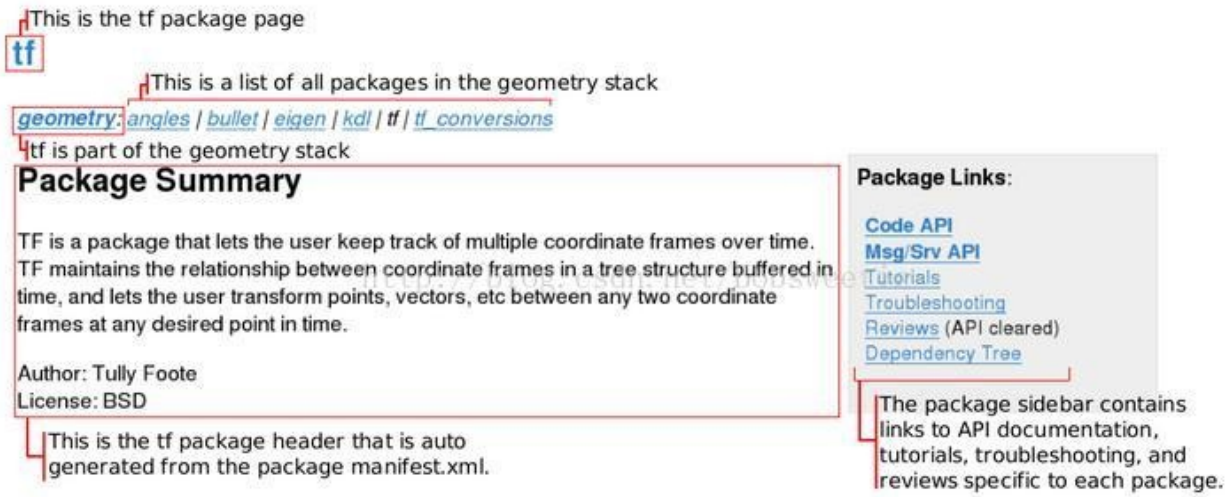
1. ROS.org登录网页

登录界面是你当你输入www.ros.org到你的浏览器时直接进入的网页。让我们看看每个wiki网页上面展示的标题。

正如你所看见的一样每个网页都包含了教程和网页简介。

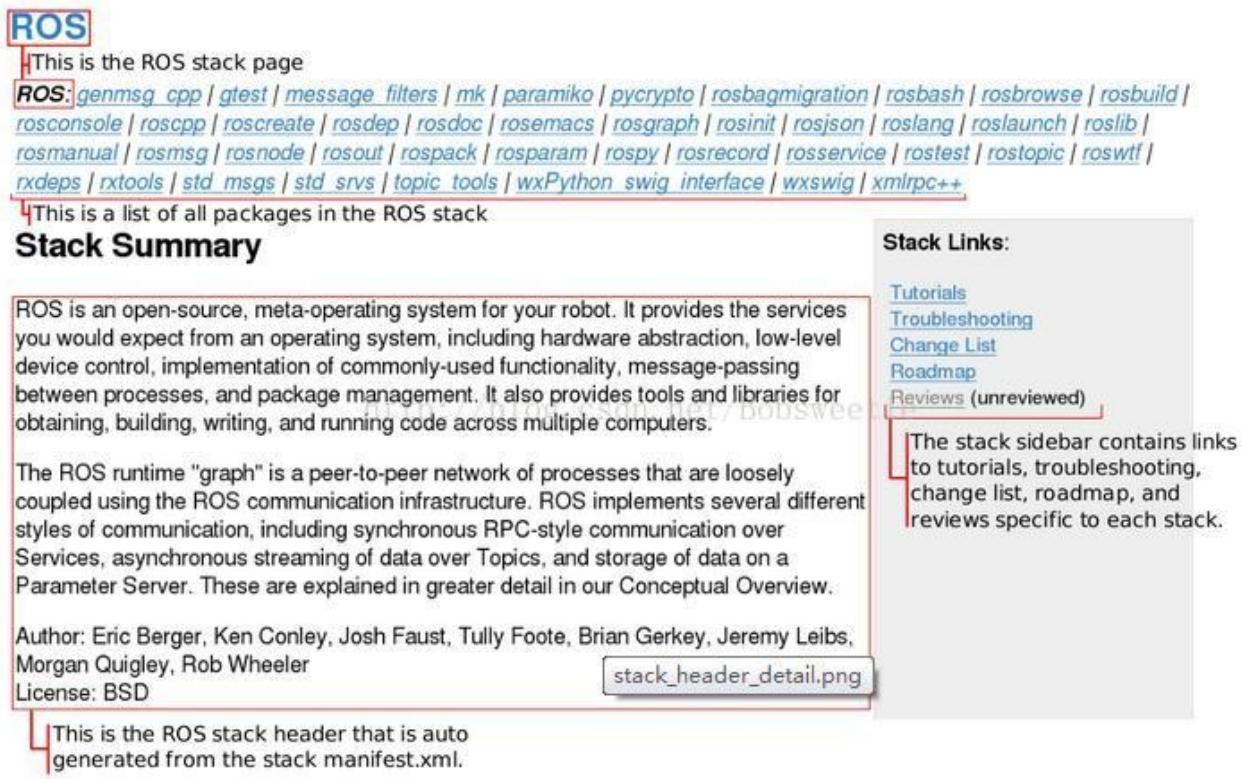
2. ROS package网页

让我们看看tf 的ros-pkg package wiki。每个网页的标题都是从stack和package的清单中自动产生。



3. ROS Stack网页

让我们看看ROS的ros stack wiki网页。每个stack的stack标题都是从stack的列表中自动产生。



正如你见到的一样，每个stack都包含了stack的教程和简介。

CH19 接下来做什么

这个教程将会讨论了解关于在实体或者仿真机器人上使用ROS的更信息。

这个时候你应该理解了ROS的核心概念。

给你一个运行ROS的机器人，你可以运用这些了解去列出机器人发布和订阅的topics，确定这些topics消耗的messags并且写下你自己的处理传感器数据的nodes，并在实际世界中起作用。

ROS最有魅力的地方不是在于它自身中发布者或者订阅者，而是ROS提供了一个标准的架构可以让全世界的开发者共享它们的代码。ROS最大的“feature”是无数的的社区。

可用的packages数量多得惊人。这个教程将告诉你接下来该探索什么。

1. 启动一个仿真

即使你有一个实体机器人，用一个仿真机器人开始还是非常好的，这样，如果有什么错误，不会伤害到你自己，或者损坏一个昂贵的机器人。

这个时候你也许用'teleop' package在控制仿真机器人，或者根据你对ROS的理解去寻找和些一个代码可能发送一个合适的message去驱动你的机器人。

2. 探索RViz

RViz是一个强大的可视化工具，允许你看到机器人的传感器和内部状态。用户指南user guide会帮助你开始。

3. 理解TF

TF package 在你机器人使用的不同的坐标框架之间进行转换，随时跟踪这些转换。对TF的好的理解对于处理一个真的机器人是必要的。学习整个教程是值得的。

如果你正在编译你的机器人，你也许会考虑为你的机器人构建URDF模型。如果你正在使用标准的机器人，那么这个可能已经编译好了，不管怎样，简单的熟悉URDF package都是有意义的。

4. 更深入

这个点你也许准备开始让你的机器人去执行更加复杂的任务。下面的网页也许会帮到你：

1.actionlib ：actionlib package为与可抢占任务的接口提供一个标准的接口，这在“高水平pakage中广泛使用。

2.navigation-2D导航：地图建立和路径规划。

3.MoveIt：控制机器人的手臂。