Learning ROS

for Ubuntu

资料来源：ROS Wiki

宣言：兴趣是一种恐怖的力量！

目录

在 Ubuntu 上安装 ROS Hydro(渔龟)....5
安装....6
配置 Ubuntu 软件仓库....6
添加 sources.list....6
添加 keys....7
安装....7
初始化 rosdep....8
环境配置....8
安装 rosinstall....8
学习 ROS....9
找答案....9
找代码....9
维基导航....9
安装并配置 ROS 环境....10
1.安装 ROS....10
管理环境....10
创建 ROS 工作空间....11
2.ROS 文件系统介绍....12
快速了解文件系统概念....12
文件系统工具....12
使用 rospack....13
使用 roscd....13
roscd log....14
使用 rosls....14
Tab 自动完成输入....15
回顾....16
3.创建 ROS 程序包....16
一个 catkin 程序包由什么组成?....17
在 catkin 工作空间中的程序包....17
创建一个 catkin 程序包....18
程序包依赖关系....18
一级依赖....18
间接依赖....19
自定义你的程序包....20
自定义 package.xml....20
自定义 CMakeLists.txt....23

编译 ROS 程序包 23
4.编译程序包 23
5.理解 ROS 节点 27
先决条件 28
图概念概述 28
节点 28
客户端库 28
roscore 28
使用 rosnode 30
使用 rosrun 31
回顾 33
6.理解 ROS 话题 34
开始 34
ROS Topics 36
ROS Messages 41
继续学习 rostopic 42
使用 rqt_plot 46
7.理解 ROS 服务和参数 48
ROS Services 49
使用 rosservice 49
Using rosparam 52
8.使用 rqt_console 和 roslaunch 55
预先安装 rqt 和 turtlesim 程序包 56
使用 rqt_console 和 rqt_logger_level 56
9.使用 rosed 编辑 ROS 中的文件 64
使用 rosed 64
使用 Tab 键补全文件名 65
编辑器 65
10.创建 ROS 消息和 ROS 服务 65
消息(msg)和服务(srv)介绍 66
使用 msg 67
创建一个 msg 67
使用 rosmsg 69
使用 srv 69
创建一个 srv 69
使用 rossrv 71
msg 和 srv 都需要的步骤 72
获得帮助 72
回顾 73
下一个教程 73
11.编写简单的消息发布器和订阅器 (C++) 74
编写发布器节点 74
编写订阅器节点 79

编译节点.....82
12.写一个简单的消息发布器和订阅器 (Python).....84
Writing the Publisher Node.....85
Writing the Subscriber Node.....88
Building your nodes.....89
13.测试消息发布器和订阅器.....90
启动发布器.....90
启动订阅器.....91
14.编写简单的 Service 和 Client (C++).....91
编写 Service 节点.....92
编写 Client 节点.....94
编译节点.....95
15.测试简单的 Service 和 Client.....96
运行 Service.....96
运行 Client.....96
关于 Service 和 Client 节点的更多例子.....97
16.录制与回放数据.....97
录制数据（通过创建一个 bag 文件）.....97
检查并回放 bag 文件.....99
录制数据子集.....101
rosbag record/play 命令的局限性.....102
17.roswtf 入门.....102
安装检查.....103
运行时检查（在有 ROS 节点运行时）.....103
错误报告.....104
18.探索 ROS 维基.....105
首页.....106
package 页面.....106
stack 页面.....107
19.接下来做什么？.....108
使用模拟器.....109
使用 RViz.....109
理解 TF.....109
更进一步.....109
中级教程.....110
20.手动创建 ROS package.....110
21.管理系统依赖项.....112
System Dependencies.....112
22.Roslaunch 在大型项目中的使用技巧.....114
Introduction.....114
高层级的结构.....115
Machine tags and Environment Variables.....116
Parameters, namespaces, and yaml files.....117

launch 文件的重用 119
参数重载 120
Roslaunch arguments 121
How to Roslaunch Nodes in Valgrind or GDB 121
Obtaining core dumps 122
23.ROS 在多机器人上的使用 123
概述 123
跨机器运行的 Talker / listener 124
启动[[master]] 124
启动 listener 124
启动 talker 124
反向测试 125
运行出错 125
译者注 125
24.自定义消息 125
自定义消息 126
引用和输出消息类型 126
C++ 126
Python 126
依赖项 126
25.如何写一篇教程 127
如何开始 127
创建一个教程 128
填写模板头信息 129
实用宏命令 131
示例教程 135

在 Ubuntu 上安装 ROS Hydro(渔龟)

我们已经创建好了一些适用于不同 Ubuntu 版本的安装包，使用这些安装包比通过编译源码来安装更加高效，所以推荐你在 Ubuntu 系统下使用这些安装包来安装。

如果你需要从源码安装 (**不推荐**)，请参见 源码安装(需要下载和编译)指南.

目录

1. 在 Ubuntu 上安装 ROS Hydro(渔龟)

1. 安装
1. 配置 Ubuntu 软件仓库
2. 添加 sources.list
3. 添加 keys
4. 安装
5. 初始化 rosdep
6. 环境配置
7. 安装 rosinstall
2. 教程

安装

配置 Ubuntu 软件仓库

配置你的 Ubuntu 软件仓库以允许系统安装"restricted"、"universe"和"multiverse"这三种组件类型的软件。你可以按照 Ubuntu 中的指南来完成这个操作。

添加 sources.list

配置你的电脑使其能够接收来自 packages.ros.org 的软件。目前 ROS Hydro 中的 debian 软件包只支持 Precise、Quantal 和 Raring 这三个版本的 ubuntu 操作系统。

- **Ubuntu 12.04 (Precise)**

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

  Ubuntu 12.10 (Quantal)

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

  Ubuntu 13.04 (Raring)

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

如果速度太慢的话可以使用中文镜像，可以明显提高下载速度加快安装过程：

- **Ubuntu 12.04 (Precise)**

  ```
  o  sudo sh -c 'echo "deb http://www.cosrobotics.org/mirror-cn-sh/ro
     s/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.lis
     t'
  ```

Ubuntu 12.10 (Quantal)

```
o   sudo sh -c 'echo "deb http://www.cosrobotics.org/mirror-cn-sh/ros/ubuntu quantal main" > /etc/apt/sources.list.d/ros-latest.list'
```

Ubuntu 13.04 (Raring)

```
o   sudo sh -c 'echo "deb http://www.cosrobotics.org/mirror-cn-sh/ros/ubuntu raring main" > /etc/apt/sources.list.d/ros-latest.list'
```

Mirrors（镜像）

提示： 有人反应添加 source.list 后在执行 sudo apt-get update 时总是提示找不到 ROS。 如果确定前面的软件仓库配置没问题反复尝试后还依然无法解决的话可以尝试一种技巧，就是直接执行 **sudo apt-get install ros-hydro-ros*** ，注意后面的＊号通配符，这个命令相当于安装核心库。为什么这样也可以安装呢，因为按下面提到的安装命令比如执行 sudo apt-get install ros-hydro-ros-base 命令的时候它本质上也是先找到所需的包然后再一个个下载安装，而刚好 ROS 核心库程序的软件包名称都是以 ros-hydro-ros 开头（比如ｒｏｓ－ｈｙｄｒｏ－ｒｏｓｃｐｐ），所以我们可以这样安装。

添加 keys

```
• wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

安装

首先，确保你的软件包列表是最新的：

```
• sudo apt-get update
```

在 ROS 中有很多各种不同的函数库和工具。为方便安装，我们提供了四种不同默认配置的安装版本。你也可以单独安装某个程序包。

- **桌面完整版安装: (推荐)** : ROS 核心组件、rqt、rviz、机器人通用库、2D/3D 模拟器、导航以及 2D/3D 感知

  ```
  o   sudo apt-get install ros-hydro-desktop-full
  ```

  或者 点击这里

- **桌面精简版安装:** ROS 核心组件、rqt、rviz、机器人通用库

  ```
  o   sudo apt-get install ros-hydro-desktop
  ```

或者 点击这里

- **ROS 核心安装:** ROS 核心组件、 编译工具以及通信相关的函数库，无 GUI 工具。

  ```
  o  sudo apt-get install ros-hydro-ros-base
  ```

  或者 点击这里

- **独立安装:** 你也可以单独安装某个 ROS 程序包(用需要安装的程序包名称替换破折号后的的“PACKAGE”):

  ```
  o  sudo apt-get install ros-hydro-PACKAGE
  ```

  e.g.

  ```
  sudo apt-get install ros-hydro-slam-gmapping
  ```

查看可供安装的 ROS 程序包，可以使用如下命令：

```
apt-cache search ros-hydro
```

初始化 rosdep

在使用 ROS 之前，你需要先初始化 rosdep。rosdep 能够让你在编译程序包源码时轻松地安装编译所依赖的库，并运行一些必要的 ROS 核心组件。

```
sudo rosdep init

rosdep update
```

环境配置

为方便起见，我们可以在每次打开终端时让系统自动配置好 ROS 环境变量，方法如下：

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

source ~/.bashrc
```

如果你安装了不止一个 ROS 版本，那么你应该在~/. bashrc 中只 source 你当前所使用的 ROS 版本对应的 setup. bash 文件。

如果你只是想切换当前终端环境下的 ROS 版本，那么你可以运行：

```
source /opt/ros/hydro/setup.bash
```

安装 rosinstall

rosinstall 是一个与 ROS 发行版无关的常用命令行工具。它只需一个命令就可以让你轻松地下载 ROS 程序包所需的资源树。

你可以使用如下命令安装 rosinstall：

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

入门指南: 介绍 | 基本概念 | 高级概念 | 客户端函数库 | 技术概述

目 录

1. 学习 ROS
2. 找答案
3. 找代码
4. 如何使用 wiki

学习 ROS

取决于你的学习方式和爱好，你可以通过以下两种方式来学习 ROS：

- 开始学习教程 - 立刻开始使用 ROS。
- 概览 - 阅读 ROS 及其功能概览。

有关 ROS 架构的更多详细信息请参考 ROS 核心文档。

找答案

有三个地方可以寻找到问题的答案。第一个就是本维基。请尝试右上角的“搜索”（Search）功能。如果你通过“搜索”无法找到问题的解决方法，就可以尝试搜索 http://answers.ros.org 或者邮件列表归档，在问答网上找答案。

查看支持页面获取更多信息。

找代码

想知道是否已经有 ROS 软件包满足你的需要吗？这个浏览软件的工具可以让你搜索到有用的软件包。你也可以通过浏览公开的 ROS 代码库列表来寻找到实现类似功能的按组归类的代码库。

维基导航

> 💡 It is appreciated that problems/questions regarding this tutorial are asked on answers.ros.org. Don't forget to include in your question

the link to this page, versions of your OS & ROS, and also add appropriate tags.

安装并配置 ROS 环境

Description: 本教程详细描述了 ROS 的安装与环境配置。

Tutorial Level: BEGINNER

Next Tutorial: ROS 文件系统介绍

目录

1. 安装 ROS
2. 管理环境
3. 创建 ROS 工作空间

1. 安装 ROS

在开始学习这些教程之前请先按照 ROS 安装说明完成安装。

注意: 如果你是使用类似 apt 这样的软件管理器来安装 ROS 的，那么安装后这些软件包将不具备写入权限，当前系统用户比如你自己也无法对这些软件包进行修改编辑。当你的开发涉及到 ROS 软件包源码层面的操作或者在创建一个新的 ROS 软件包时，你应该是在一个具备读写权限的目录下工作，就像在你当前系统用户的 home 目录下一样。

管理环境

在安装 ROS 期间，你会看到提示说需要 source 多个 setup.*sh 文件中的某一个，或者甚至提示添加这条'source'命令到你的启动脚本里面。这些操作是必须的，因为 ROS 是依赖于某种组合空间的概念，而这种概念就是通过配置脚本环境来实现的。这可以让针对不同版本或者不同软件包集的开发更加容易。

如果你在查找和使用 ROS 软件包方面遇到了问题，请确保你已经正确配置了脚本环境。一个检查的好方法是确保你已经设置了像 ROS_ROOT 和 ROS_PACKAGE_PATH 这样的环境变量：

```
$ export | grep ROS
```

如果发现没有配置，那这个时候你就需要'source'某些'setup.*sh'文件了。

ROS 会帮你自动生成这些'setup.*sh'文件，通过以下方式生成并保存在不同地方：

- 通过类似 apt 的软件包管理器安装 ROS 软件包时会生成 setup.*sh 文件。
- 在 rosbuild workspaces 中通过类似 rosws 的工具生成。
- 在编译 或 安装 catkin 软件包时自动生成。

注意： 在所有教程中你将会经常看到分别针对 rosbuild 和 catkin 的不同操作说明，这是因为目前有两种不同的方法可以用来组织和编译 ROS 应用程序。一般而言，rosbuild 比较简单也易于使用，而 catkin 使用了更加标准的 CMake 规则，所以比较复杂，但是也更加灵活，特别是对于那些想整合外部现有代码或者想发布自己代码的人。关于这些如果你想了解得更全面请参阅 catkin or rosbuild。

如果你是通过 ubuntu 上的 apt 工具来安装 ROS 的，那么你将会在'/opt/ros/<distro>/'目录中看到 setup.*sh 文件，然后你可以执行下面的 source 命令：

```
# source /opt/ros/<distro>/setup.bash
```

请使用具体的 ROS 发行版名称代替<distro>。

比如你安装的是 ROS Hydro，则上述命令改为：

```
$ source /opt/ros/hydro/setup.bash
```

在每次打开终端时你都需要先运行上面这条命令后才能运行 ros 相关的命令，为了避免这一繁琐过程，你可以事先在.bashrc 文件（初学者请注意：该文件是在当前系统用户的 home 目录下。）中添加这条命令，这样当你每次登录后系统已经帮你执行这些命令配置好环境。这样做也可以方便你在同一台计算机上安装并随时切换到不同版本的 ROS（比如 fuerte 和 groovy）。

此外，你也可以在其它系统平台上相应的 ROS 安装目录下找到这些 setup.*sh 文件。

创建 ROS 工作空间

catkinrosbuild

这些操作方法只适用于 ROS Groovy 及后期版本，对于 ROS Fuerte 及早期版本请选择 rosbuild。

下面我们开始创建一个 catkin 工作空间：

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

$ cd ~/catkin_ws/src
```

即使这个工作空间是空的（在'src'目录中没有任何软件包，只有一个 CMakeLists.txt 链接文件），你依然可以“build”它：

```
$ cd ~/catkin_ws/

$ catkin_make
```

catkin_make 命令在 catkin 工作空间中是一个非常方便的工具。如果你查看一下当前目录应该能看到'build'和'devel'这两个文件夹。在'devel'文件夹里面你可以看到几个 setup.*sh 文件。source 这些文件中的任何一个都可以将当前工作空间设置在 ROS 工作环境的最顶层。关于这些想了解更多请阅读 catkin 文档。接下来首先 source 一下新生成的 setup.*sh 文件：

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

到此你的工作环境已经搭建完成，接下来可以继续学习 ROS 文件系统教程.

2. ROS 文件系统介绍

Description: 本教程介绍 ROS 文件系统概念，包括命令行工具 roscd、rosls 和 rospack 的使用。

Tutorial Level: BEGINNER

Next Tutorial: 创建 ROS 软件包

catkinrosbuild

目录

1. 快速了解文件系统概念
2. 文件系统工具
1. 使用 rospack
2. 使用 roscd
1. 子目录
3. roscd log
4. 使用 rosls
5. Tab 自动完成输入
3. 回顾

快速了解文件系统概念

- **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。

Show note about stacks

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。

使用 rospack

rospack 允许你获取软件包的有关信息。在本教程中，我们只涉及到 rospack 中 find 参数选项，该选项可以返回软件包的路径信息。

用法：

```
# rospack find [包名称]
```

示例：

```
$ rospack find roscpp
```

应输出：

```
• YOUR_INSTALL_PATH/share/roscpp
```

如果你是在 Ubuntu Linux 操作系统上通过 apt 来安装 ROS，你应该会准确地看到：

```
• /opt/ros/groovy/share/roscpp
```

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
```

为了验证我们已经切换到了 roscpp 软件包目录下，现在我们可以使用 Unix 命令 pwd 来输出当前工作目录：

```
$ pwd
```

你应该会看到：

```
• YOUR_INSTALL_PATH/share/roscpp
```

你可以看到 YOUR_INSTALL_PATH/share/roscpp 和之前使用 rospack find 得到的路径名称是一样的。

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
```

你的 ROS_PACKAGE_PATH 环境变量应该包含那些保存有 ROS 软件包的路径，并且每个路径之间用冒号分隔开来。一个典型的 ROS_PACKAGE_PATH 环境变量如下：

```
• /opt/ros/groovy/base/install/share:/opt/ros/groovy/base/instal
  l/stacks
```

跟其他路径环境变量类似，你可以在 ROS_PACKAGE_PATH 中添加更多其它路径，每条路径使用冒号':'分隔。

子目录

使用 roscd 也可以切换到一个软件包或软件包集的子目录中。

执行：

```
$ roscd roscpp/cmake

$ pwd
```

应该会看到：

```
• YOUR_INSTALL_PATH/share/roscpp/cmake
```

roscd log

使用 roscd log 可以切换到 ROS 保存日记文件的目录下。需要注意的是，如果你没有执行过任何 ROS 程序，系统会报错说该目录不存在。

如果你已经运行过 ROS 程序，那么可以尝试：

```
$ roscd log
```

使用 rosls

rosls 是 rosbash 命令集中的一部分，它允许你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
```

应输出：

```
• cmake package.xml srv
```

Tab 自动完成输入

当要输入一个完整的软件包名称时会变得比较繁琐。在之前的例子中 roscpp tutorials 是个相当长的名称，幸运的是，一些 ROS 工具支持 TAB 自动完成输入的功能。

输入：

```
# roscd roscpp_tut<<< 现在请按 TAB 键 >>>
```

当按 **TAB** 键后，命令行中应该会自动补充剩余部分：

```
$ roscd roscpp_tutorials/
```

这应该有用，因为 roscpp tutorials 是当前唯一一个名称以 roscpp tut 作为开头的 ROS 软件包。

现在尝试输入：

```
# roscd tur<<< 现在请按 TAB 键 >>>
```

按 **TAB** 键后，命令应该会尽可能地自动补充完整：

```
$ roscd turtle
```

但是，在这种情况下有多个软件包是以 turtle 开头，当再次按 **TAB** 键后应该会列出所有以 turtle 开头的 ROS 软件包：

```
• turtle_actionlib/ turtlesim/ turtle_tf/
```

这时在命令行中你应该仍然只看到：

```
$ roscd turtle
```

现在在 turtle 后面输入 s 然后按 **TAB** 键：

```
# roscd turtles<<< 请按 TAB 键 >>>
```

因为只有一个软件包的名称以 turtles 开头，所以你应该会看到：

```
$ roscd turtlesim/
```

回顾

你也许已经注意到了 ROS 命令工具的的命名方式：

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

这种命名方式在许多 ROS 命令工具中都会用到。

到此你已经了解了 ROS 的文件系统结构，接下来我们开始创建一个工作空间。

3. 创建 ROS 程序包

Description: 本教程介绍如何使用 roscreate-pkg 或 catkin 创建一个新程序包,并使用 rospack 查看程序包的依赖关系。

Tutorial Level: BEGINNER

Next Tutorial: 编译 ROS 程序包

catkinrosbuild

目 录

1. 一个 catkin 程序包由什么组成?
2. 在 catkin 工作空间中的程序包
3. 创建一个 catkin 程序包
4. 程序包依赖关系
1. 一级依赖
2. 间接依赖
5. 自定义你的程序包
1. 自定义 package.xml
1. 描述标签
2. 维护者标签
3. 许可标签
4. 依赖项标签

5. 最后完成的 package.xml
2. 自定义 CMakeLists.txt

一个 catkin 程序包由什么组成?

一个程序包要想称为 catkin 程序包必须符合以下要求：

- 该程序包必须包含 catkin compliant package.xml 文件
  - 这个 package.xml 文件提供有关程序包的元信息。
- 程序包必须包含一个 catkin 版本的 CMakeLists.txt 文件，而 Catkin metapackages 中必须包含一个对 CMakeList.txt 文件的引用。
- 每个目录下只能有一个程序包。
  - 这意味着在同一个目录下不能有嵌套的或者多个程序包存在。

最简单的程序包也许看起来就像这样：

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

在 catkin 工作空间中的程序包

开发 catkin 程序包的一个推荐方法是使用 catkin 工作空间，但是你也可以单独开发(standalone)catkin 软件包。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 ca
  tkin
•     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
```

在继续本教程之前请先按照创建 catkin 工作空间教程创建一个空白的 catkin 工作空间。

创建一个 catkin 程序包

本部分教程将演示如何使用 catkin_create_pkg 命令来创建一个新的 catkin 程序包以及创建之后都能做些什么。

首先切换到之前通过创建 catkin 工作空间教程创建的 catkin 工作空间中的 src 目录下：

```
# You should have created this in the Creating a Workspace Tutorial

$ cd ~/catkin_ws/src
```

现在使用 catkin_create_pkg 命令来创建一个名为'beginner_tutorials'的新程序包，这个程序包依赖于 std_msgs、roscpp 和 rospy：

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

这将会创建一个名为 beginner_tutorials 的文件夹，这个文件夹里面包含一个 package.xml 文件和一个 CMakeLists.txt 文件，这两个文件都已经自动包含了部分你在执行 catkin_create_pkg 命令时提供的信息。

catkin_create_pkg 命令会要求你输入 package_name，如果有需要你还可以在后面添加一些需要依赖的其它程序包：

```
# This is an example, do not try to run this

# catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

catkin_create_pkg 命令也有更多的高级功能，这些功能在 catkin/commands/catkin_create_pkg 中有描述。

程序包依赖关系

一级依赖

之前在使用 catkin_create_pkg 命令时提供了几个程序包作为依赖包，现在我们可以使用 rospack 命令工具来查看**一级**依赖包。

(Jan 9, 2013) There is a bug reported and already fixed in rospack in groovy, which takes sometime until the change gets reflected on your computer. If you see a similar issue like this with the next command, you can skip to the next command.

```
$ rospack depends1 beginner_tutorials
```

```
- std_msgs
- rospy
```

```
- roscpp
```

就像你看到的，rospack 列出了在运行 catkin_create_pkg 命令时作为参数的依赖包，这些依赖包随后保存在 **package.xml** 文件中。

```
$ roscd beginner_tutorials

$ cat package.xml
```

```
- <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

在很多情况中，一个依赖包还会有它自己的依赖包，比如，rospy 还有其它依赖包。

(Jan 9, 2013) There is a bug reported and already fixed in rospack in groovy, which takes sometime until the change gets reflected on your computer. If you see a similar issue like this with the next command, you can skip to the next command.

```
$ rospack depends1 rospy
```

```
- genpy
- rosgraph
- rosgraph_msgs
- roslib
- std_msgs
```

一个程序包还可以有好几个间接的依赖包，幸运的是使用 rospack 可以递归检测出所有的依赖包。

```
$ rospack depends beginner_tutorials
```

```
cpp_common

rostime

roscpp_traits

roscpp_serialization

genmsg

genpy

message_runtime

rosconsole

std_msgs

rosgraph_msgs

xmlrpcpp

roscpp

rosgraph

catkin

rospack

roslib

rospy
```

自定义你的程序包

本部分教程将剖析 catkin_create_pkg 命令生成的每个文件并详细描述这些文件的组成部分以及如何自定义这些文件。

自定义 package.xml

自动生成的 package.xml 文件应该在你的新程序包中。现在让我们一起来看看新生成的 package.xml 文件以及每一个需要你注意的标签元素。

描述标签

首先更新**描述**标签：

切换行号显示

```
5   <description>The beginner_tutorials package</description>
```

将描述信息修改为任何你喜欢的内容，但是按照约定第一句话应该简短一些，因为它覆盖了程序包的范围。如果用一句话难以描述完全那就需要换行了。

维护者标签

接下来是**维护者**标签：

切换行号显示

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

这是 package.xml 中要求填写的一个重要标签，因为它能够让其他人联系到程序包的相关人员。至少需要填写一个维护者名称，但如果有需要的话你可以添加多个。除了在标签里面填写维护者的名称外，还应该在标签的 email 属性中填写邮箱地址：

切换行号显示

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

许可标签

再接下来是**许可**标签，同样的也需要：

切换行号显示

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

你应该选择一种许可协议并将它填写到这里。一些常见的开源许可协议有 BSD、MIT、Boost Software License、GPLv2、GPLv3、LGPLv2.1 和 LGPLv3。你可以在 Open Source Initiative 中阅读其中的若干个许可协议的相关信息。对于本教程我们将使用 BSD 协议，因为 ROS 核心组件的剩余部分已经使用了该协议：

切换行号显示

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

依赖项标签

接下来的标签用来描述程序包的各种依赖项，这些依赖项分为 build_depend、buildtool_depend、run_depend、test_depend。关于这些标签的更详细介绍请参考 Catkin Dependencies 相关的文档。在之前的操作中，因为我们将 std_msgs、 roscpp、和 rospy 作为 catkin_create_pkg 命令的参数，所以生成的依赖项看起来如下：

切换行号显示

```
  <!-- 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>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>
```

除了 catkin 中默认提供的 buildtool_depend，所有我们列出的依赖包都已经被添加到 build_depend 标签中。在本例中，因为在编译和运行时我们需要用到所有指定的依赖包，因此还需要将每一个依赖包分别添加到 run_depend 标签中：

切换行号显示

```
  <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>
```

最后完成的 package.xml

现在看下面最后去掉了注释和未使用标签后的 package.xml 文件就显得更加简洁了：

切换行号显示

```
<?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</ur
l>
  <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>
```

自定义 CMakeLists.txt

到此，这个包含程序包元信息的 package.xml 文件已经按照需要完成了裁剪整理，现在你可以继续下面的教程了。catkin_create_pkg 命令生成的 CMakeLists.txt 文件将在后续关于编译 ROS 程序代码的教程中讲述。

现在你已经创建了一个新的 ROS 程序包，接下来我们开始编译这个程序包

编译 ROS 程序包

Description: 本教程介绍 ROS 程序包的编译方法

Tutorial Level: BEGINNER

Next Tutorial: 理解 ROS 节点

catkinrosbuild

目录

1. 编译程序包
1. 使用 catkin_make
2. 开始编译你的程序包

4. 编译程序包

一旦安装了所需的系统依赖项，我们就可以开始编译刚才创建的程序包了。

注意: 如果你是通过 apt 或者其它软件包管理工具来安装 ROS 的，那么系统已经默认安装好所有依赖项。

记得事先 source 你的环境配置(setup)文件，在 Ubuntu 中的操作指令如下：

```
$ source /opt/ros/groovy/setup.bash
```

使用 catkin_make

catkin_make 是一个命令行工具，它简化了 catkin 的标准工作流程。你可以认为 catkin_make 是在 CMake 标准工作流程中依次调用了 cmake 和 make。

使用方法:

```
# 在 catkin 工作空间下

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

CMake 标准工作流程主要可以分为以下几个步骤：

注意: 如果你运行以下命令是无效的，因为它只是一个演示 CMake 工作流程的例子。

```
# 在一个 CMake 项目里

$ mkdir build

$ cd build

$ cmake ..

$ make

$ make install  # (可选)
```

每个 CMake 工程在编译时都会执行这个操作过程。相反，多个 catkin 项目可以放在工作空间中一起编译，工作流程如下：

```
# In a catkin workspace

$ catkin_make

$ catkin_make install  # (可选)
```

上述命令会编译 src 文件夹下的所有 catkin 工程。想更深入了解请参考 REP128。 如果你的源代码不在默认工作空间中（~/catkin_ws/src),比如说存放在了 my_src 中，那么你可以这样来使用 catkin_make:

注意: 运行以下命令时无效的，因为 my_src 不存在。

```
# In a catkin workspace
```

```
$ catkin_make --source my_src

$ catkin_make install --source my_src  # (optionally)
```

对于 catkin_make 更高级的使用方法，请参考 catkin/commands/catkin_make

开始编译你的程序包

对于正要马上编译自己代码的读者，请同时看一下后面的(C++)/(Python)教程，因为你可能需要修改 CMakeLists.txt 文件。

按照之前的创建一个 ROS 程序包教程，你应该已经创建好了一个 catkin 工作空间 和一个名为 beginner_tutorials 的 catkin 程序包。现在切换到 catkin workspace 并查看 src 文件夹：

```
$ cd ~/catkin_ws/

$ ls src
```

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

你可以看到一个名为 beginner_tutorials 的文件夹，这就是你在之前的 catkin_create_pkg 教程里创建的。现在我们可以使用 catkin_make 来编译它了：

```
$ catkin_make
```

你可以看到很多 cmake 和 make 输出的信息：

```
- Base path: /home/user/catkin_ws
- Source space: /home/user/catkin_ws/src
- Build space: /home/user/catkin_ws/build
- Devel space: /home/user/catkin_ws/devel
- Install space: /home/user/catkin_ws/install
- ####
- #### Running command: "cmake /home/user/catkin_ws/src
- -DCATKIN_DEVEL_PREFIX=/home/user/catkin_ws/devel
- -DCMAKE_INSTALL_PREFIX=/home/user/catkin_ws/install" in "/home/
  user/catkin_ws/build"
- ####
- -- The C compiler identification is GNU 4.2.1
```

```
-- The CXX compiler identification is Clang 4.0.0
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - yes
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Using CATKIN_DEVEL_PREFIX: /tmp/catkin_ws/devel
-- Using CMAKE_PREFIX_PATH: /opt/ros/groovy
-- This workspace overlays: /opt/ros/groovy
-- Found PythonInterp: /usr/bin/python (found version "2.7.1")
-- Found PY_em: /usr/lib/python2.7/dist-packages/em.pyc
-- Found gtest: gtests will be built
-- catkin 0.5.51
-- BUILD_SHARED_LIBS is on
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing packages in topological order:
-- ~~  - beginner_tutorials
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ add_subdirectory(beginner_tutorials)
-- Configuring done
-- Generating done
```

```
-- Build files have been written to: /home/user/catkin_ws/build
####
#### Running command: "make -j4" in "/home/user/catkin_ws/build"
####
```

catkin_make 首先输出它所使用到的每个**空间**所在的路径。更多关于**空间**的信息，请参考 REP128 和 catkin/workspaces。需要注意的是由于这些空间存在默认配置的原因，有几个文件夹已经在 catkin 工作空间自动生成了，使用 ls 查看:

```
$ ls
```

```
build
devel
src
```

build 目录是 build space 的默认所在位置，同时 cmake 和 make 也是在这里被调用来配置并编译你的程序包。devel 目录是 devel space 的默认所在位置，同时也是在你安装程序包之前存放可执行文件和库文件的地方。

现在我们已成功编译了一个 ROS 程序包，接下来我们将介绍 ROS 节点.

5. 理解 ROS 节点

Description: 本教程主要介绍 ROS 图(graph)概念 并讨论 roscore、rosnode 和 rosrun 命令行工具的使用。

Tutorial Level: BEGINNER

Next Tutorial: 理解 ROS 话题

目录

1. 先决条件
2. 图概念概述
3. 节点
4. 客户端库
5. roscore
6. 使用 rosnode
7. 使用 rosrun
8. 回顾

先决条件

在本教程中我们将使用到一个轻量级的模拟器，请使用以下命令来安装：

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

用你使用的ROS发行版本名称(例如electric、fuerte、groovy、h y d r o 等)替换掉'<distro>'。

图概念概述

- Nodes:节点,一个节点即为一个可执行文件,它可以通过 ROS 与其它节点进行通信。
- Messages:消息，消息是一种 ROS 数据类型，用于订阅或发布到一个话题。
- Topics:话题,节点可以发布消息到话题，也可以订阅话题以接收消息。
- Master:节点管理器，ROS 名称服务 (比如帮助节点找到彼此)。
- rosout: ROS 中相当于 stdout/stderr。
- roscore: 主机+ rosout + 参数服务器 (参数服务器会在后面介绍)。

节点

一个节点其实只不过是 ROS 程序包中的一个可执行文件。ROS 节点可以使用 ROS 客户库与其他节点

通信。节点可以发布或接收一个话题。节点也可以提供或使用某种服务。

客户端库

ROS 客户端库允许使用不同编程语言编写的节点之间互相通信:

- rospy = python 客户端库
- roscpp = c++ 客户端库

roscore

roscore 是你在运行所有 ROS 程序前首先要运行的命令。

请运行:

```
$ roscore
```

然后你会看到类似下面的输出信息:

```
... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/
roslaunch-
```

```
•
•  machine_name-13039.log
•  Checking log directory for disk usage. This may take awhile.
•  Press Ctrl-C to interrupt
•  Done checking log file disk usage. Usage is <1GB.
•
•  started roslaunch server http://machine_name:33919/
•  ros_comm version 1.4.7
•
•  SUMMARY
•  ========
•
•  PARAMETERS
•   * /rosversion
•   * /rosdistro
•
•  NODES
•
•  auto-starting new master
•  process[master]: started with pid [13054]
•  ROS_MASTER_URI=http://machine_name:11311/
•
•  setting /run_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
•  process[rosout-1]: started with pid [13067]
•  started core service [/rosout]
```

如果 roscore 运行后无法正常初始化，很有可能是存在网络配置问题。参见

网络设置——单机设置

如果 roscore 不能初始化并提示缺少权限，这可能是因为~/.ros 文件夹

归属于 root 用户（只有 root 用户才能访问），修改该文件夹的用户归属关系：

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

使用 rosnode

打开一个**新的终端**，可以使用 **rosnode** 像运行 roscore 一样看看在运行什么...

注意: 当打开一个新的终端时，你的运行环境会复位，同时你的~/.bashrc 文件会复原。如果你在运行类似于 rosnode 的指令时出现一些问题，也许你需要添加一些环境设置文件到你的~/.bashrc 或者手动重新配置他们。

rosnode 显示当前运行的 ROS 节点信息。rosnode list 指令列出活跃的节点:

```
$ rosnode list
```

- 你会看到:

```
/rosout
```

这表示当前只有一个节点在运行: rosout。因为这个节点用于收集和记录节点调试输出信息，所以它总是在运行的。

rosnode info 命令返回的是关于一个特定节点的信息。

```
$ rosnode info /rosout
```

这给了我们更多的一些有关于 rosout 的信息，例如，事实上由它发布 /rosout_agg.

```
--------------------------------------------------------------
----------
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
```

现在，让我们看看更多的节点。为此，我们将使用 rosrun 弹出另一个节点。

使用 rosrun

rosrun 允许你使用包名直接运行一个包内的节点(而不需要知道这个包的路径)。

用法:

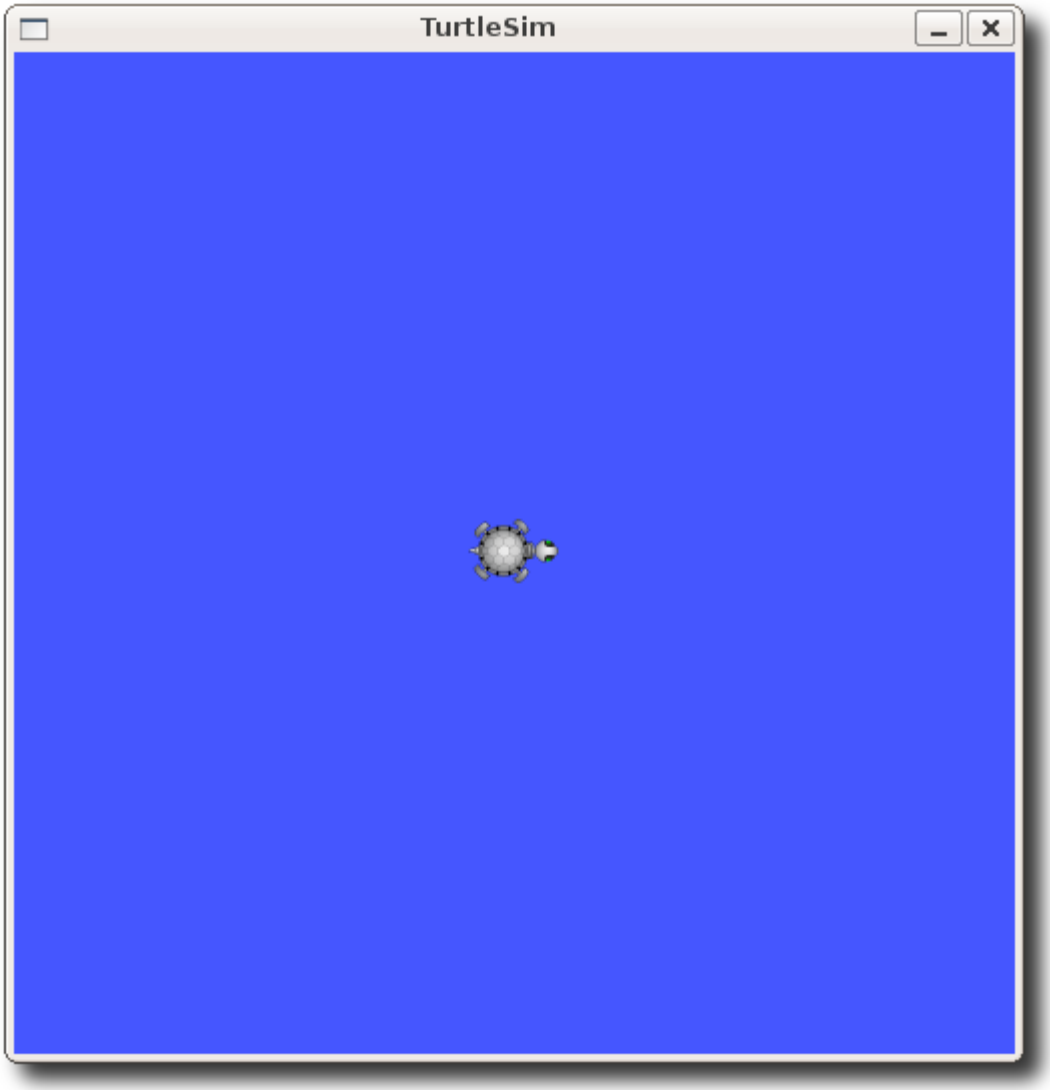
```
$ rosrun [package_name] [node_name]
```

现在我们可以运行 turtlesim 包中的 turtlesim_node。

然后，在一个 **新的终端**:

```
$ rosrun turtlesim turtlesim_node
```

你会看到 turtlesim 窗口:



-

注意: 这里的 turtle 可能和你的 turtlesim 窗口不同。别担心，这里有许多版本的 turtle ，而你的是一个惊喜!

在一个 **新的终端**:

```
$ rosnode list
```

你会看见类似于:

```
• /rosout
• /turtlesim
```

ROS 的一个强大特性就是你可以通过命令行重新配置名称。

关闭 turtlesim 窗口停止运行节点 (或者回到 `rosrun turtlesim` 终端并使用`ctrl -C`)。现在让我们重新运行它，但是这一次使用 Remapping Argument 改变节点名称:

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

现在，我们退回使用 rosnode list:

```
$ rosnode list
```

- 你会看见类似于:

```
- /rosout
- /my_turtle
```

注意: 如果你仍看到 /turtlesim 在列表中，这可能意味着你在终端中使用 ctrl-C 停止节点而不是关闭窗口，或者你没有$ROS_HOSTNAME 环境变量，这在 Network Setup - Single Machine Configuration 中有定义。你可以尝试清除 rosnode 列表，通过: $ rosnode cleanup

我们可以看到新的/my_turtle 节点。使用另外一个 rosnode 指令, ping, 来测试:

```
$ rosnode ping my_turtle
```

```
- 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
```

回顾

本节所涉及的内容:

- roscore = ros+core : master (provides name service for ROS) + rosout

(stdout/stderr) + parameter server (parameter server will be introduced later)

- rosnode = ros+node : ROS tool to get information about a node.
- rosrun = ros+run : runs a node from a given package.

到这里，您已经了解了 ROS 节点是如何工作的，下一步，我们来了解一下

ROS 话题。如果您想关闭

turtlesim_node，请按下“Ctrl-C”

6. 理解 ROS 话题

Description: 本教程介绍 ROS 话题（topics）以及如何使用 rostopic 和 rxplot 命令行工具。

Tutorial Level: BEGINNER

Next Tutorial: 理解 ROS 服务和参数

目 录

1. 开始
1. roscore
2. turtlesim
3. 通过键盘远程控制 turtle
2. ROS Topics
1. 使用 rqt_graph
2. rostopic 介绍
3. 使用 rostopic echo
4. 使用 rostopic list
3. ROS Messages
1. 使用 rostopic type
4. 继续学习 rostopic
1. 使用 rostopic pub
2. 使用 rostopic hz
5. 使用 rqt_plot

开始

roscore

首先确保 roscore 已经运行，打开一个新的终端：

```
$ roscore
```

如果你没有退出在上一篇教程中运行的 roscore，那么你可能会看到下面的错误信息：

```
• roscore cannot run as another roscore/master is already running.
```

```
• Please kill other roscore/master processes before relaunching
```

这是正常的，因为只需要有一个 roscore 在运行就够了。

turtlesim

在本教程中我们也会使用到 turtlesim，请**在一个新的终端中**运行：

```
$ rosrun turtlesim turtlesim_node
```

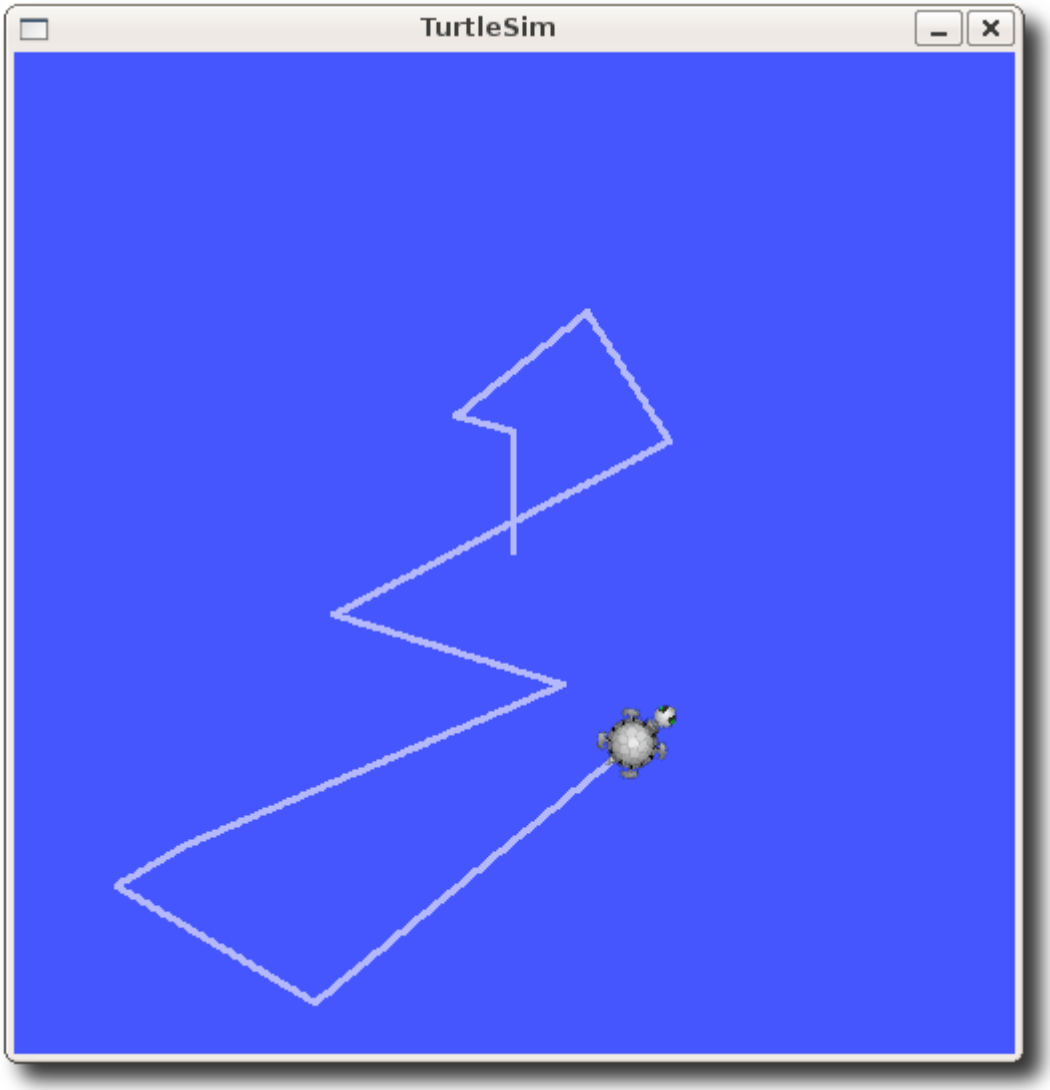
通过键盘远程控制 turtle

我们也需要通过键盘来控制 turtle 的运动，请**在一个新的终端中**运行：

```
$ rosrun turtlesim turtle_teleop_key
```

```
• [ INFO] 1254264546.878445000: Started node [/teleop_turtle], pi
  d [5528], bound on [aqy], xmlrpc port [43918], tcpros port [5593
  6], logging to [~/ros/ros/log/teleop_turtle_5528.log], using [r
  eal] time
• Reading from keyboard
• ---------------------------
• Use arrow keys to move the turtle.
```

现在你可以使用键盘上的方向键来控制 turtle 运动了。如果不能控制，请**选中 turtle_teleop_key 所在的终端窗口**以确保你的按键输入能够被捕获。



现在你可以控制 turtle 运动了，下面我们一起来看看这背后发生的事。

ROS Topics

turtlesim_node 节点和 turtle_teleop_key 节点之间是通过一个 ROS **话题**来互相通信的。turtle_teleop_key 在一个话题上**发布**按键输入消息，而 turtlesim 则**订阅**该话题以接收该消息。下面让我们使用 rqt_graph 来显示当前运行的节点和话题。

注意：如果你使用的是 electric 或更早期的版本，那么 rqt 是不可用的，请使用 rxgraph 代替。

使用 rqt_graph

rqt_graph 能够创建一个显示当前系统运行情况的动态图形。rqt_graph 是 rqt 程序包中的一部分。如果你没有安装，请通过以下命令来安装：

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

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

请使用你的 ROS 版本名称（比如 fuerte、groovy、hydro 等）来替换掉<distro>。

在一个新终端中运行:

```
$ rosrun rqt_graph rqt_graph
```

你会看到类似下图所示的图形：

如果你将鼠标放在/turtle1/command_velocity 上方，相应的 ROS 节点（蓝色和绿色）和话题（红色）就会高亮显示。正如你所看到的，turtlesim_node 和 turtle_teleop_key 节点正通过一个名为 /turtle1/command_velocity 的话题来互相通信。

rostopic 介绍

rostopic 命令工具能让你获取有关 ROS **话题**的信息。

你可以使用帮助选项查看 rostopic 的子命令：

```
$ rostopic -h
```

```
• 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
```

接下来我们将使用其中的一些子命令来查看 turtlesim。

使用 rostopic echo

rostopic echo 可以显示在某个话题上发布的数据。

用法：

```
rostopic echo [topic]
```

让我们在**一个新终端中**看一下 turtle_teleop_key 节点在 /turtle1/command_velocity 话题（非 hydro 版）上发布的数据。

```
$ rostopic echo /turtle1/command_velocity
```

如果你是用 ROS Hydro 及其之后的版本（下同），请运行：

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

你可能看不到任何东西因为现在还没有数据发布到该话题上。接下来我们通过按下方向键使 turtle_teleop_key 节点发布数据。**记住如果 turtle 没有动起来的话就需要你重新选中 turtle_teleop_key 节点运行时所在的终端窗口。**

现在当你按下向上方向键时应该会看到下面的信息：

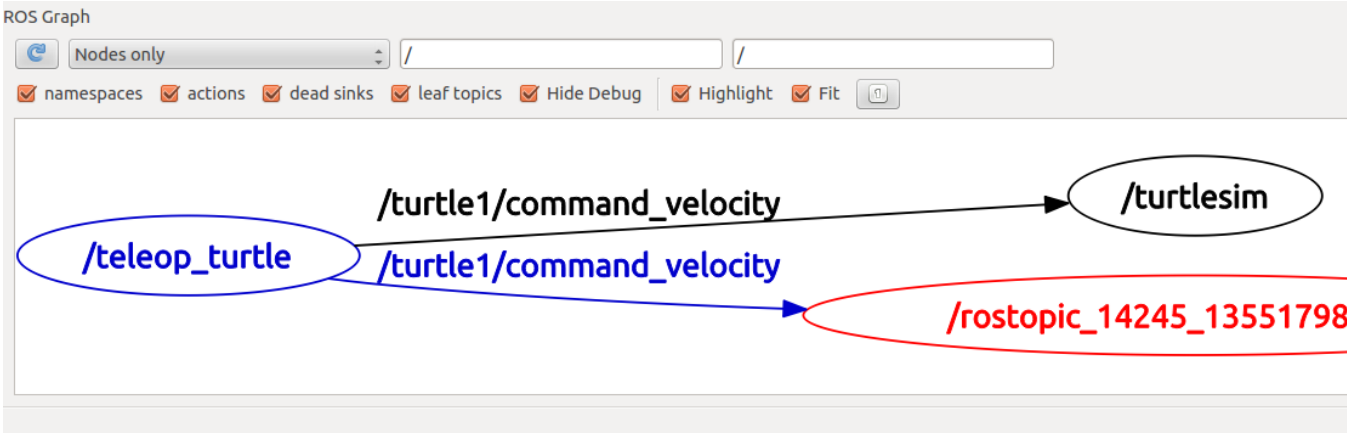
```
• ---
• linear: 2.0
• angular: 0.0
• ---
• linear: 2.0
• angular: 0.0
• ---
• linear: 2.0
```

```
•  angular: 0.0
•  ---
•  linear: 2.0
•  angular: 0.0
•  ---
•  linear: 2.0
•  angular: 0.0
```

或者在 hrydro 中如下：

```
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
---
```

现在让我们再看一下 rqt_graph（你可能需要刷新一下 ROS graph）。正如你所看到的，rostopic echo(红色显示部分）现在也**订阅**了 turtle1/command_velocity 话题。



使用 rostopic list

rostopic list 能够列出所有当前订阅和发布的话题。

让我们查看一下 list 子命令需要的参数，在**一个新终端中**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
```

在 rostopic list 中使用 **verbose** 选项：

```
$ rostopic list -v
```

这会显示出有关所发布和订阅的话题及其类型的详细信息。

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

```
•   * /rosout_agg [roslib/Log] 1 publisher
•   * /turtle1/pose [turtlesim/Pose] 1 publisher
•
•  Subscribed topics:
•   * /turtle1/command_velocity [turtlesim/Velocity] 1 subscriber
•   * /rosout [roslib/Log] 1 subscriber
```

ROS Messages

话题之间的通信是通过在节点之间发送 ROS **消息**实现的。对于发布器(turtle_teleop_key)和订阅器(turtulesim_node)之间的通信，发布器和订阅器之间必须发送和接收相同**类型**的消息。这意味着话题的**类型**是由发布在它上面的消息**类型**决定的。使用 rostopic type 命令可以查看发布在某个话题上的消息**类型**。

使用 rostopic type

rostopic type 命令用来查看所发布话题的消息类型。

用法：

```
rostopic type [topic]
```

运行(非 hydro 版)：

```
$ rostopic type /turtle1/command_velocity
```

- 你应该会看到:

  ```
  turtlesim/Velocity
  ```

hydro 版请运行：

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

- 你应该会看到:

  ```
  geometry_msgs/Twist
  ```

我们可以使用 rosmsg 命令来查看消息的详细情况（非 hydro 版)：

```
$ rosmsg show turtlesim/Velocity
```

```
- float32 linear
- float32 angular
```

hydro 版：

```
$ 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
```

现在我们已经知道了 turtlesim 节点所期望的消息类型，接下来我们就可以给 turtle 发布命令了。

继续学习 rostopic

现在我们已经了解了什么是 ROS 的**消息**，接下来我们开始结合消息来使用 rostopic。

使用 rostopic pub

rostopic pub 可以把数据发布到当前某个正在广播的话题上。

用法：

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

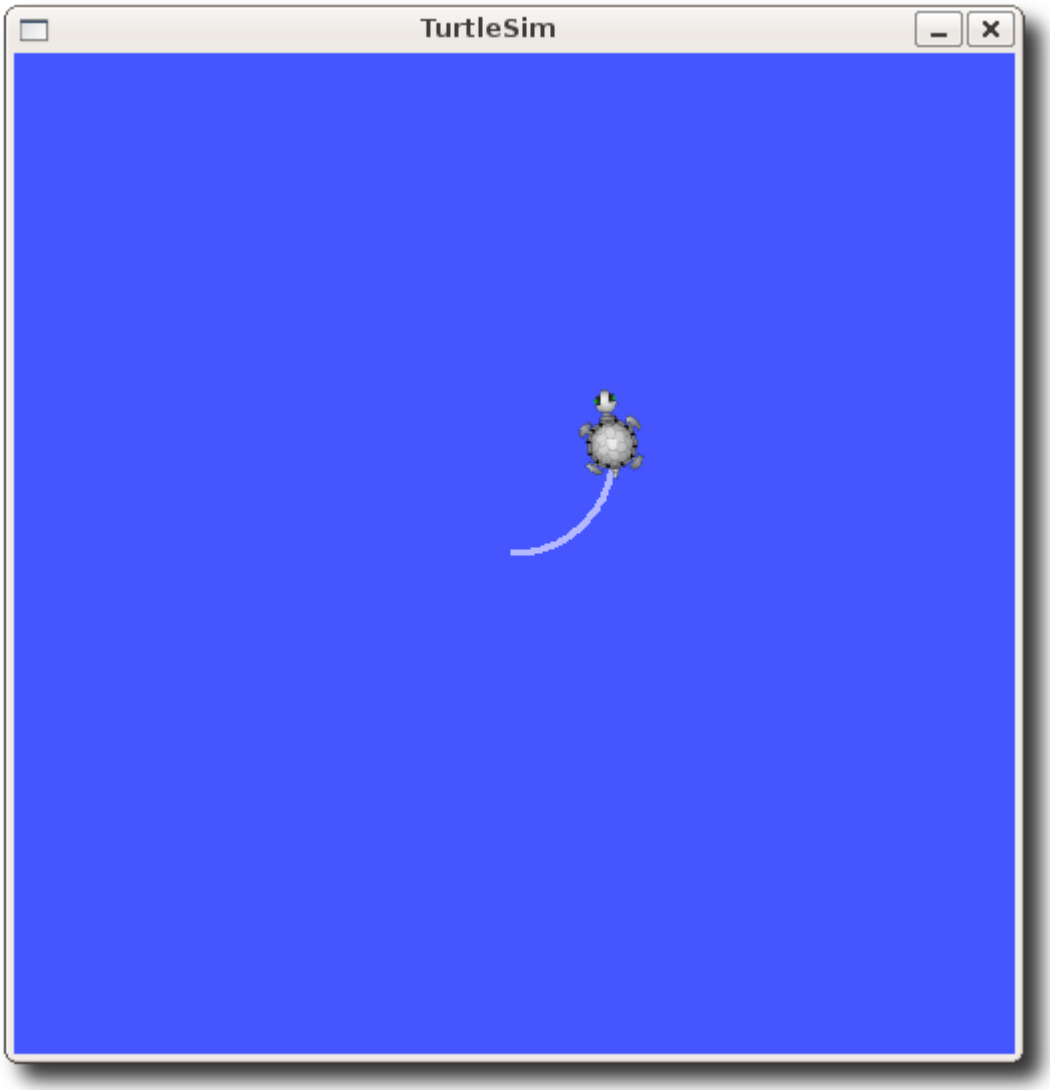
示例（非 hydro 版）：

```
$ rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity -- 2.0
  1.8
```

示例（hydro 版)：

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

以上命令会发送一条消息给 turtlesim，告诉它以 2.0 大小的线速度和 1.8 大小的角速度开始移动。



-

这是一个非常复杂的例子，因此让我们来详细分析一下其中的每一个参数。

- `rostopic pub`
  - 这条命令将会发布消息到某个给定的话题。
- `-1`
  - （单个破折号）这个参数选项使 rostopic 发布一条消息后马上退出。
- `/turtle1/command_velocity`
  - 这是消息所发布到的话题名称。

- `turtlesim/Velocity`
    - 这是所发布消息的类型。
- `--`
- （双破折号）这会告诉命令选项解析器接下来的参数部分**都不是**命令选项。这在参数里面包含有破折号-（比如负号）时是必须要添加的。
- `2.0 1.8`
- 正如之前提到的，在一个 turtlesim/Velocity 消息里面包含有两个浮点型元素：linear 和 angular。在本例中，2.0 是 linear 的值，1.8 是 angular 的值。这些参数其实是按照 YAML 语法格式编写的，这在 YAML 文档中有更多的描述。

你可能已经注意到 turtle 已经停止移动了。这是因为 turtle 需要一个稳定的频率为 1Hz 的命令流来保持移动状态。我们可以使用 rostopic pub -r 命令来发布一个稳定的命令流（非 hydro 版）：

```
$ rostopic pub /turtle1/command_velocity turtlesim/Velocity -r 1 -- 2.
0  -1.8
```

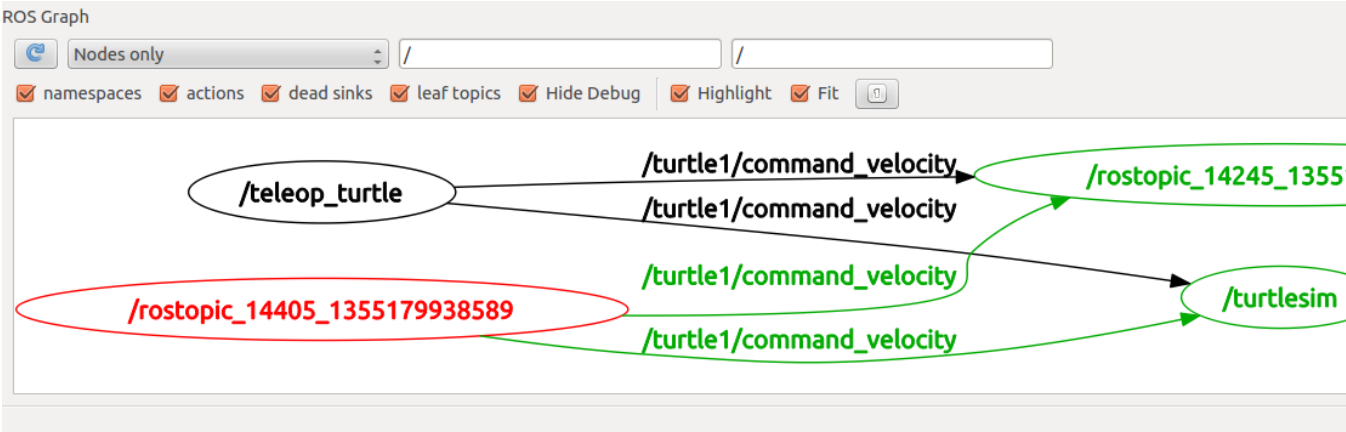
hydro 版：

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

这条命令以 1Hz 的频率发布速度命令到速度话题上。

•

我们也可以看一下 rqt_graph 中的情形，可以看到 rostopic 发布器节点（红色）正在与 rostopic echo 节点（绿色）进行通信：



正如你所看到的，turtle 正沿着一个圆形轨迹连续运动。我们可以在**一个新终端**中通过 rostopic echo 命令来查看 turtlesim 所发布的数据。

使用 rostopic hz

rostopic hz 命令可以用来查看数据发布的频率。

用法：

```
rostopic hz [topic]
```

我们看一下 turtlesim_node 发布/turtle/pose 时有多快：

```
$ 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
```

现在我们可以知道了 turtlesim 正以大约 60Hz 的频率发布数据给 turtle。我们也可以结合 rostopic type 和 rosmsg show 命令来获取关于某个话题的更深层次的信息（非 hydro 版）：

```
$ rostopic type /turtle1/command_velocity | rosmsg show
```

hydro 版：

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

到此我们已经完成了通过 rostopic 来查看话题相关情况的过程，接下来我将使用另一个工具来查看 turtlesim 发布的数据。

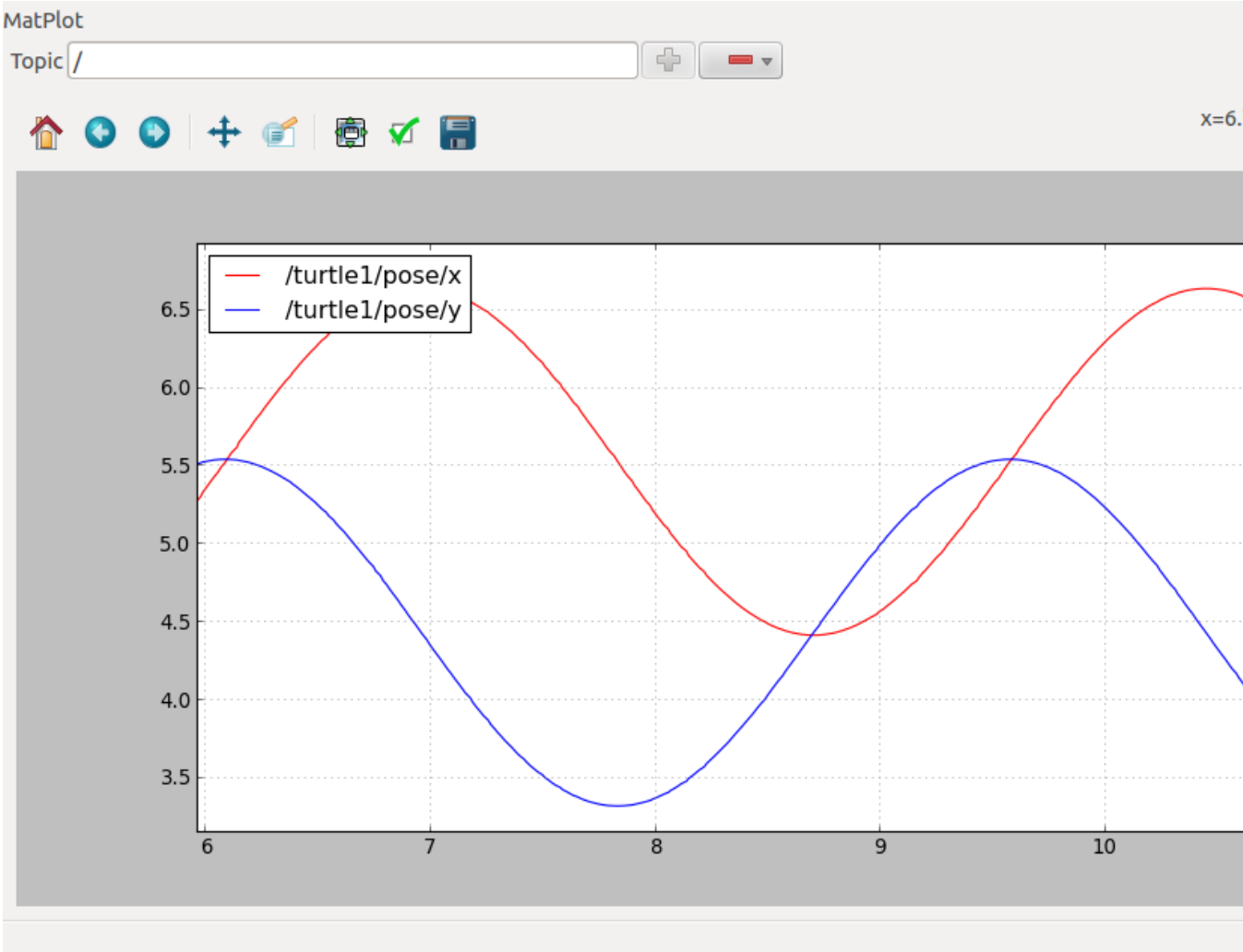
使用 rqt_plot

注意：如果你使用的是 electric 或更早期的 ROS 版本，那么 rqt 命令是不可用的，请使用 rxplot 命令来代替。

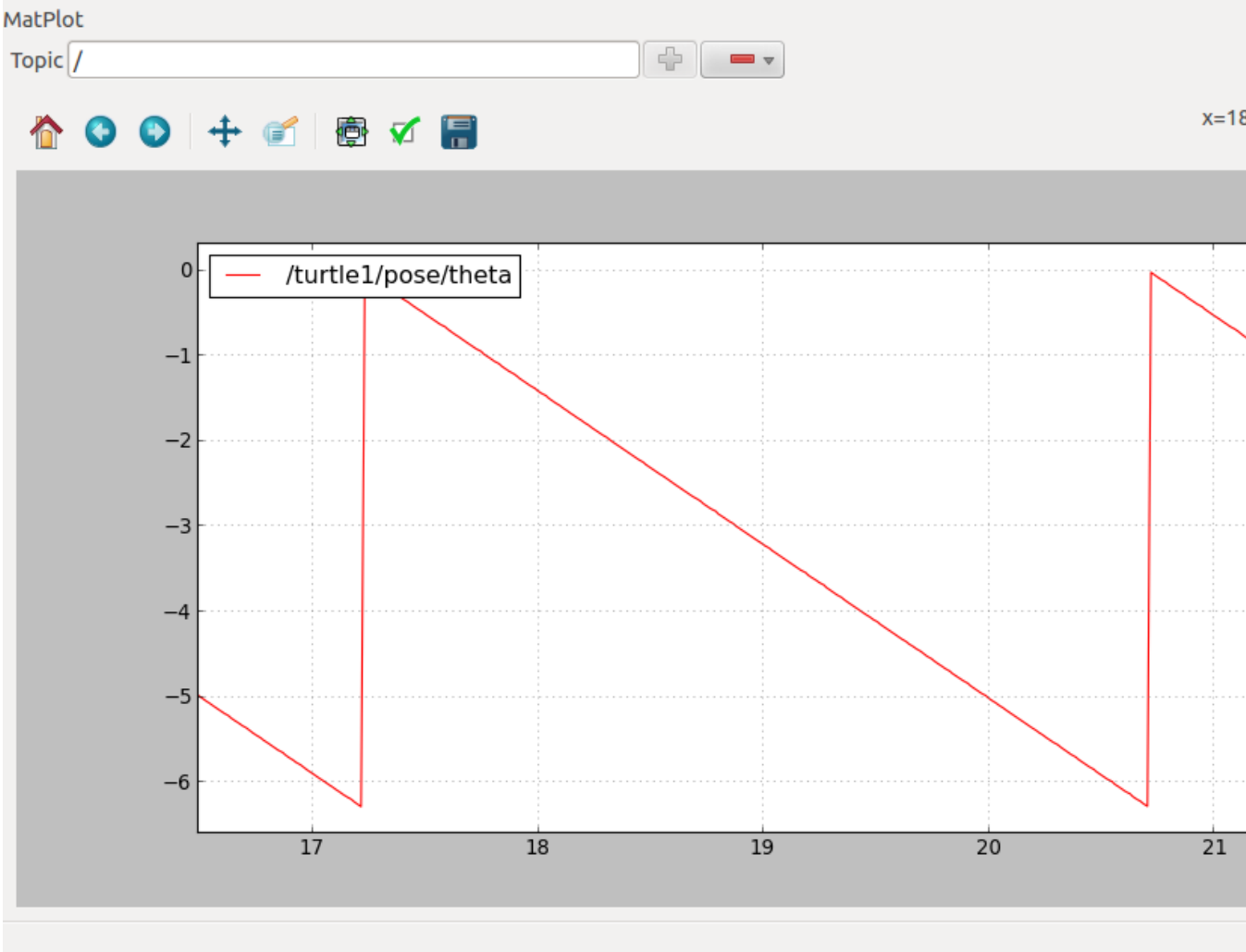
rqt_plot 命令可以实时显示一个发布到某个话题上的数据变化图形。这里我们将使用 rqt_plot 命令来绘制正在发布到/turtle1/pose 话题上的数据变化图形。首先，在一个新终端中运行 rqt_plot 命令：

```
$ rosrun rqt_plot rqt_plot
```

这会弹出一个新窗口，在窗口左上角的一个文本框里面你可以添加需要绘制的话题。在里面输入/turtle1/pose/x 后之前处于禁用状态的加号按钮将会被使能变亮。按一下该按钮，并对/turtle1/pose/y 重复相同的过程。现在你会在图形中看到 turtle 的 x-y 位置坐标图。



按下减号按钮会显示一组菜单让你隐藏图形中指定的话题。现在隐藏掉你刚才添加的话题并添加/turtle1/pose/theta，你会看到如下图所示的图形：



本部分教程到此为止，请使用 Ctrl-C 退出 rostopic 命令，但要保持 turtlesim 继续运行。

到此我们已经理解了 ROS 话题是如何工作的，接下来我们开始学习理解 ROS 服务和参数。

7. 理解 ROS 服务和参数

Description: 本教程介绍了 ROS 服务和参数的知识，以及命令行工具 rosservice 和 rosparam 的使用方法。

Tutorial Level: BEGINNER

Next Tutorial: 使用 rqt_console 和 roslaunch

目录

1. ROS Services
2. 使用 rosservice
1. rosservice list

2. rosservice type
3. rosservice call
3. Using rosparam
1. rosparam list
2. rosparam set and rosparam get
3. rosparam dump and rosparam load

本教程假设从前一教程启动的 turtlesim_node 仍在运行，现在我们来看看 turtlesim 提供了什么服务：

ROS Services

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）**

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，如下所示：

使用方法:

```
rosservice list         输出可用服务的信息
rosservice call         调用带参数的服务
rosservice type         输出服务类型
rosservice find         依据类型寻找服务 find services by service type
rosservice uri          输出服务的 ROSRPC uri
```

rosservice list

```
$ rosservice list
```

list 命令显示 turtlesim 节点提供了 9 个服务：重置（reset），清除（clear），再生（spawn），终止（kill）,turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/get_loggers, andturtlesim/set_logger_level. 同时还有另外两个 rosout 节点提供的服务: /rosout/get_loggers and/rosout/set_logger_level.

```
• /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
```

我们使用 rosservice type 命令更进一步查看 clear 服务:

rosservice type

使用方法:

```
rosservice type [service]
```

我们来看看 clear 服务的类型:

```
$ rosservice type clear

std_srvs/Empty
```

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）。下面我们使用 rosservice call 命令调用服务：

rosservice call

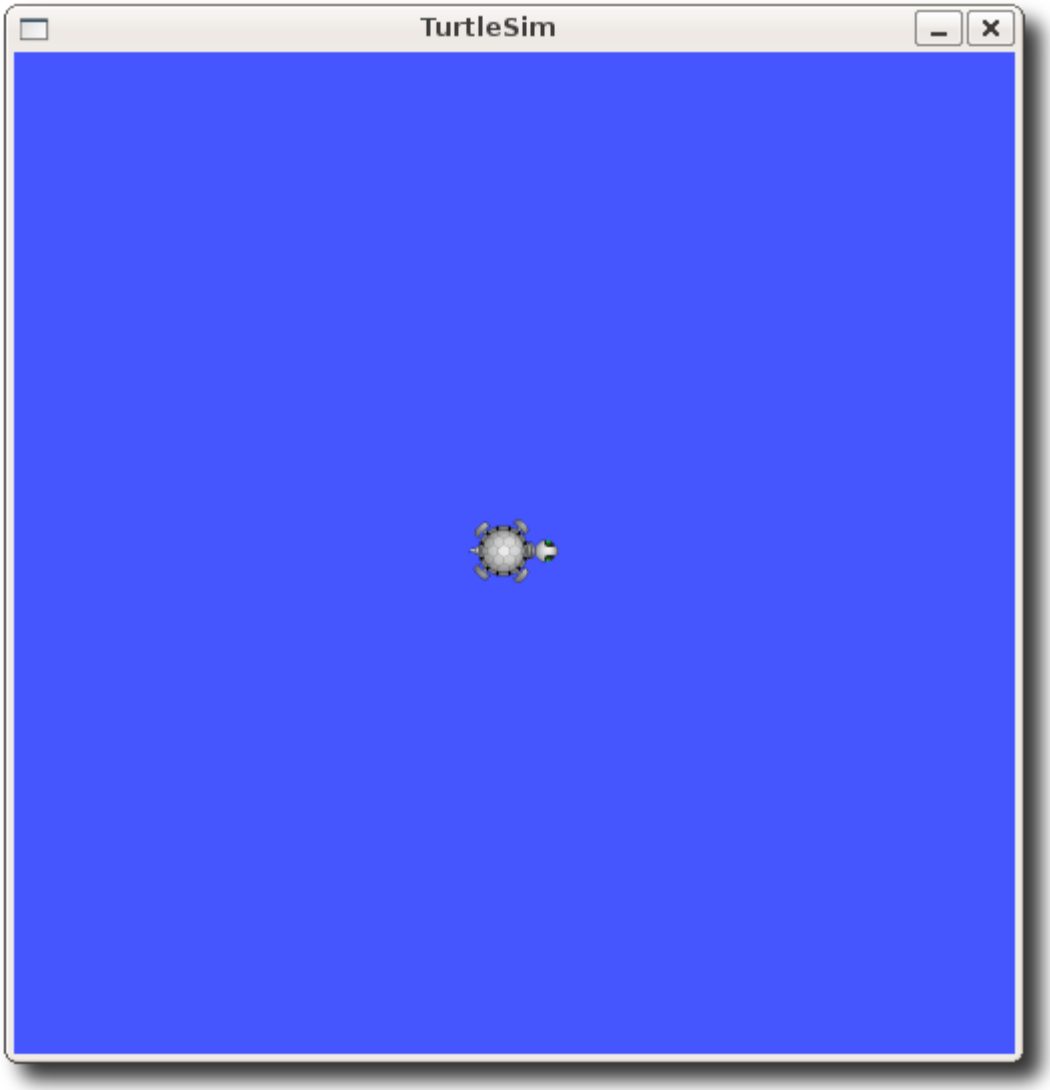
使用方法:

```
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。



通过查看再生（spawn）服务的信息，我们来了解带参数的服务:

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

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

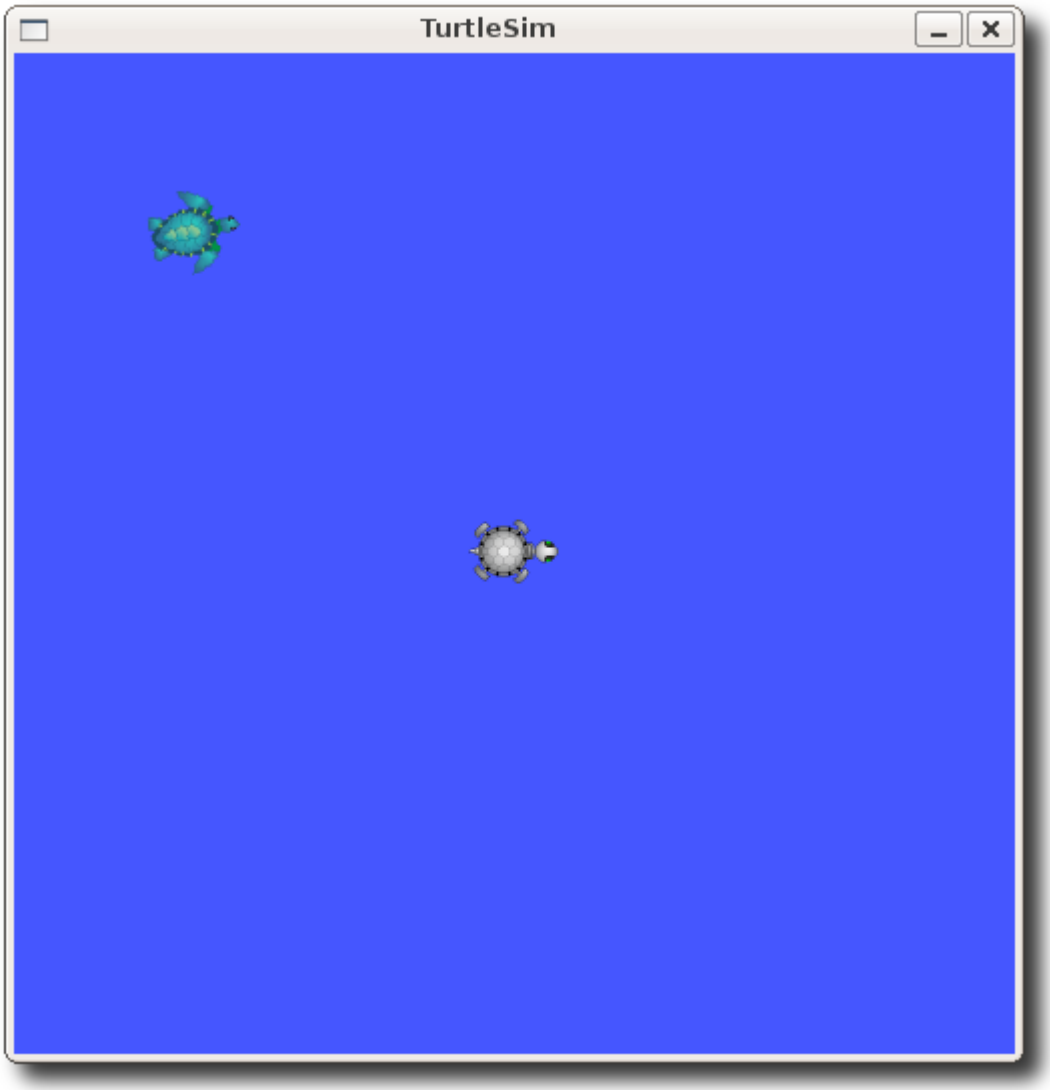
这个服务使得我们可以在给定的位置和角度生成一只新的乌龟。名字参数是可选的，这里我们不设具体的名字，让 turtlesim 自动创建一个。

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

服务返回了新产生的乌龟的名字：

```
• name: turtle2
```

现在我们的乌龟看起来应该是像这样的：



•

Using rosparam

rosparam 使得我们能够存储并操作 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。一般而言，YAML 的表述很自然：1 是整型, 1.0 是浮点型, one 是字符串, true 是布尔, [1, 2, 3]是整型列表, {a: b, c: d}是字典. rosparam 有很多指令可以用来操作参数，如下所示:

使用方法:

```
rosparam set            设置参数
rosparam get            获取参数
rosparam load           从文件读取参数
rosparam dump           向文件中写入参数
rosparam delete         删除参数
rosparam list           列出参数名
```

我们来看看现在参数服务器上都有哪些参数：

rosparam list

```
$ rosparam list
```

我们可以看到 turtlesim 节点在参数服务器上有 3 个参数用于设定背景颜色：

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

Let's change one of the parameter values using `rosparam set`:

rosparam set and rosparam get

Usage:

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

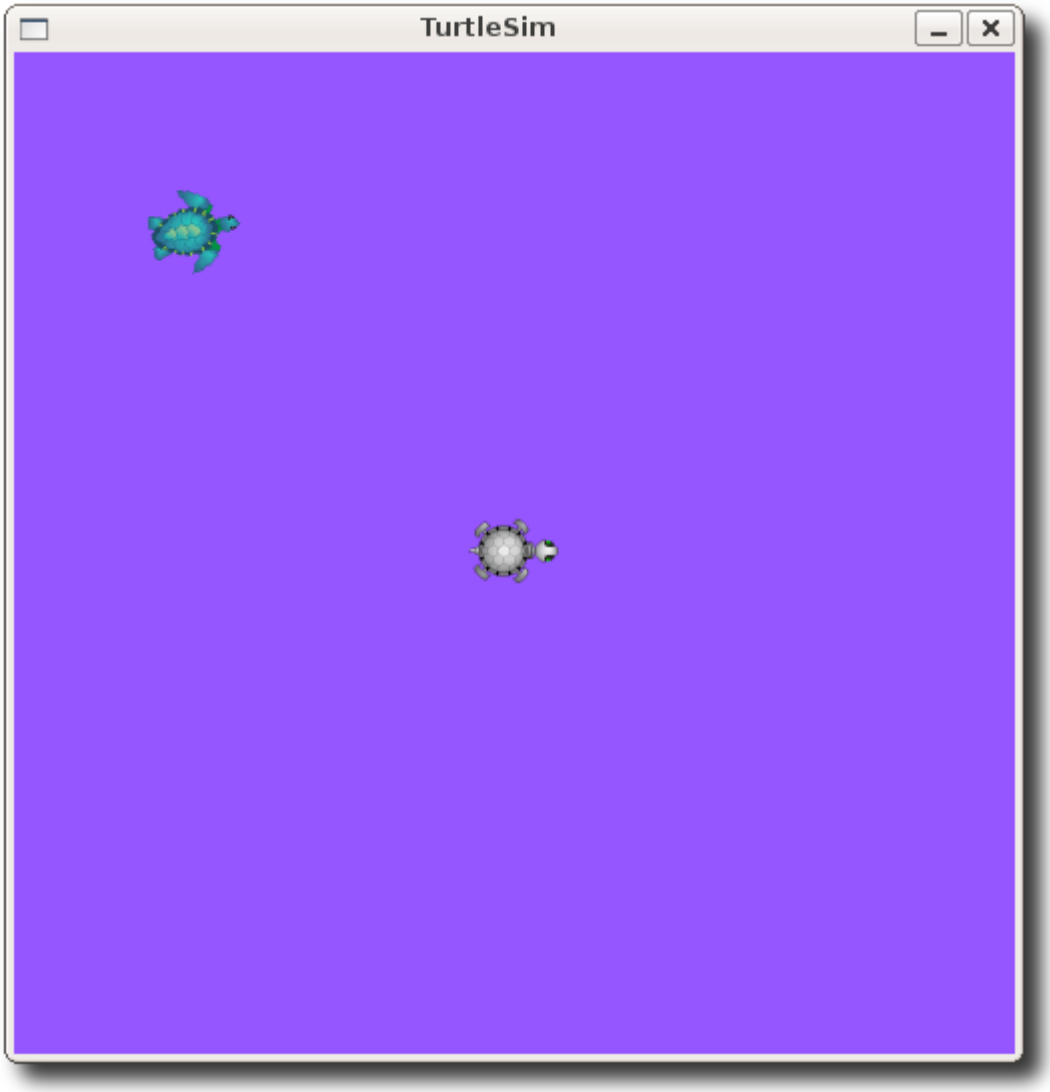
现在我们修改背景颜色的红色通道：

```
$ 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
```

现在 我们的小乌龟看起来应该是像这样：



-

现在我们来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
```

```
- 86
```

我们可以使用 rosparam get /来显示参数服务器上的所有内容：

```
$ 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
```

你可能希望存储这些信息以备今后重新读取。这通过 rosparam 很容易就可以实现:

rosparam dump and 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
```

你甚至可以将 yaml 文件重载入新的命名空间，比如说 copy 空间:

```
$ rosparam load params.yaml copy

$ rosparam get copy/background_b
```

```
• 255
```

至此，我们已经了解了 ROS 服务和参数服务器的使用，接下来，我们一同试试使用_rqt_console_和_roslaunch

8. 使用 rqt_console 和 roslaunch

Description: 本教程介绍如何使用 rqt_console 和 rqt_logger_level 进行调试，以及如何使用 roslaunch 同时运行多个节点。早期版本中的 rqt 工具并不完善，因此，如果你使用的是“ROS fuerte”或更早期的版本，请同时参考这个页面学习使用老版本的“rx”工具。

Tutorial Level: BEGINNER

Next Tutorial: 如何使用 rosed

目 录

1. 预先安装 rqt 和 turtlesim 程序包
2. 使用 rqt_console 和 rqt_logger_level
1. 日志等级说明
2. 使用 roslaunch
3. Launch_文件
4. Launch_文件解析

5. roslaunching

预先安装 rqt 和 turtlesim 程序包

本教程会用到 rqt 和 turtlesim 这两个程序包，如果你没有安装，请先安装：

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

请使用 ROS 发行版名称(比如 electric、fuerte、groovy、hydro 或最新的 indigo)替换掉 <distro>。

注意： 你可能已经在之前的某篇教程中编译过 rqt 和 turtlesim，如果你不确定的话重新编译一次也没事。

使用 rqt_console 和 rqt_logger_level

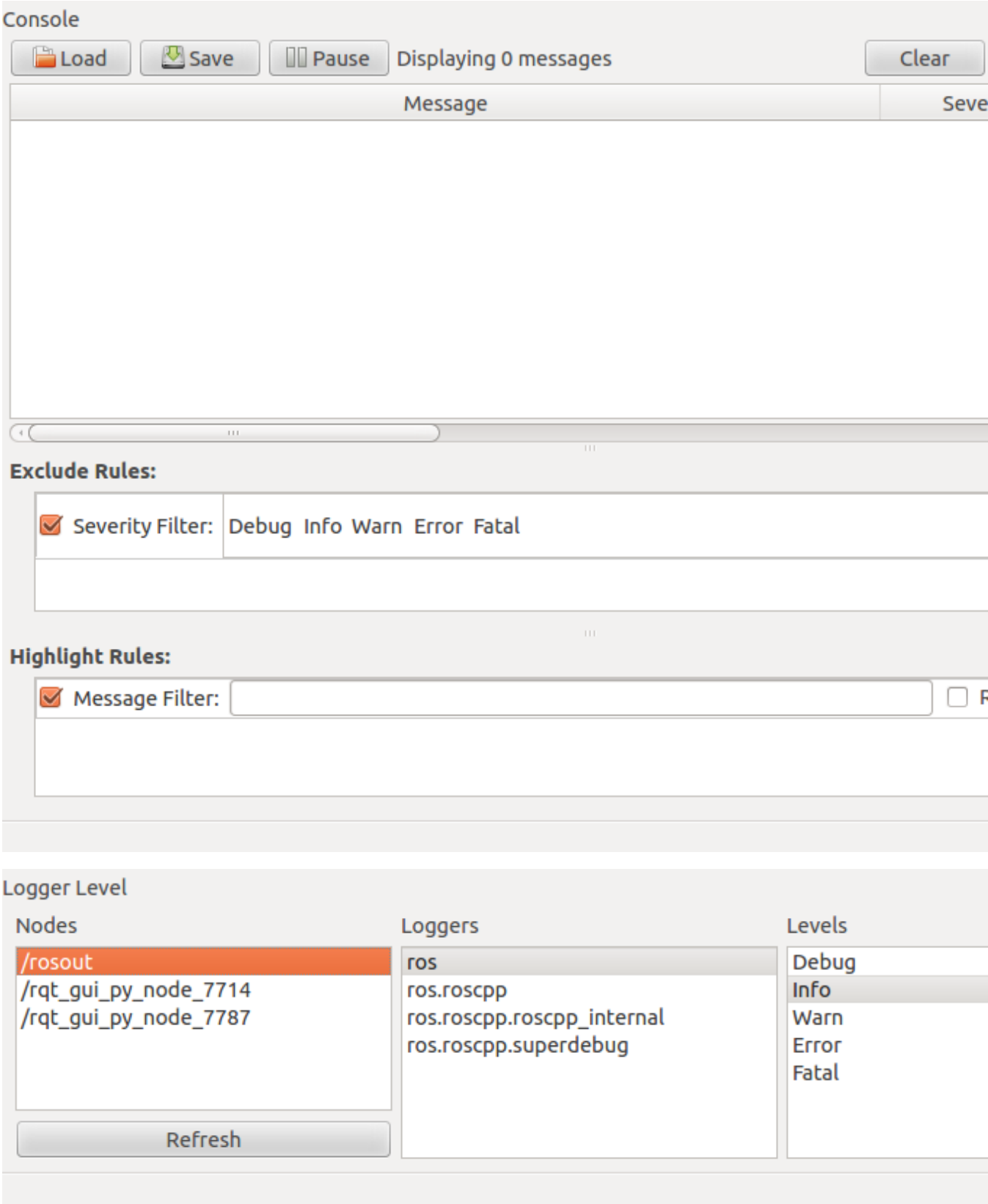
rqt_console 属于 ROS 日志框架(logging framework)的一部分，用来显示节点的输出信息。rqt_logger_level 允许我们修改节点运行时输出信息的日志等级（logger levels）（包括 DEBUG、WARN、INFO 和 ERROR）。

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：

```
$ rosrun rqt_console rqt_console

$ rosrun rqt_logger_level rqt_logger_level
```

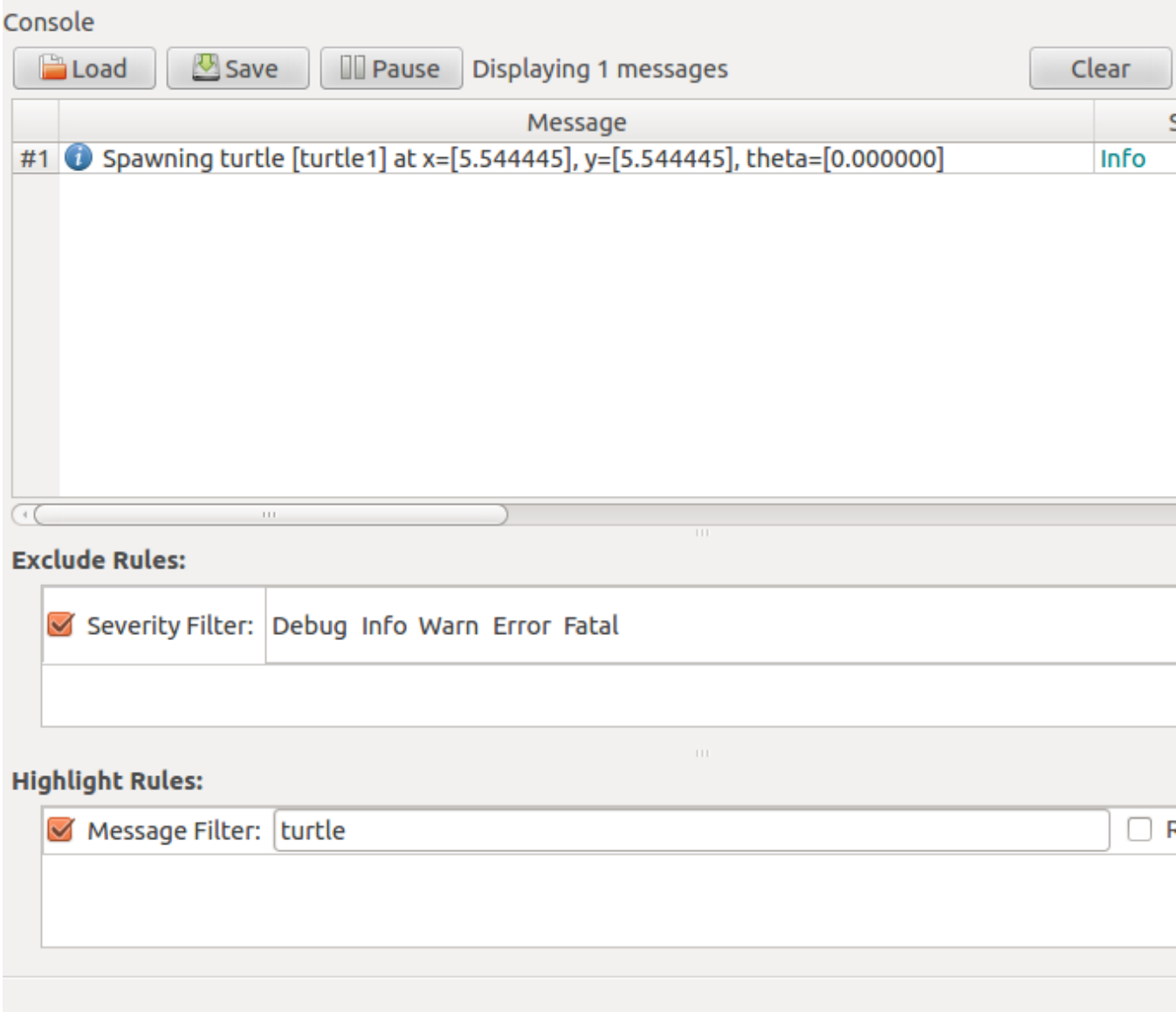
你会看到弹出两个窗口：



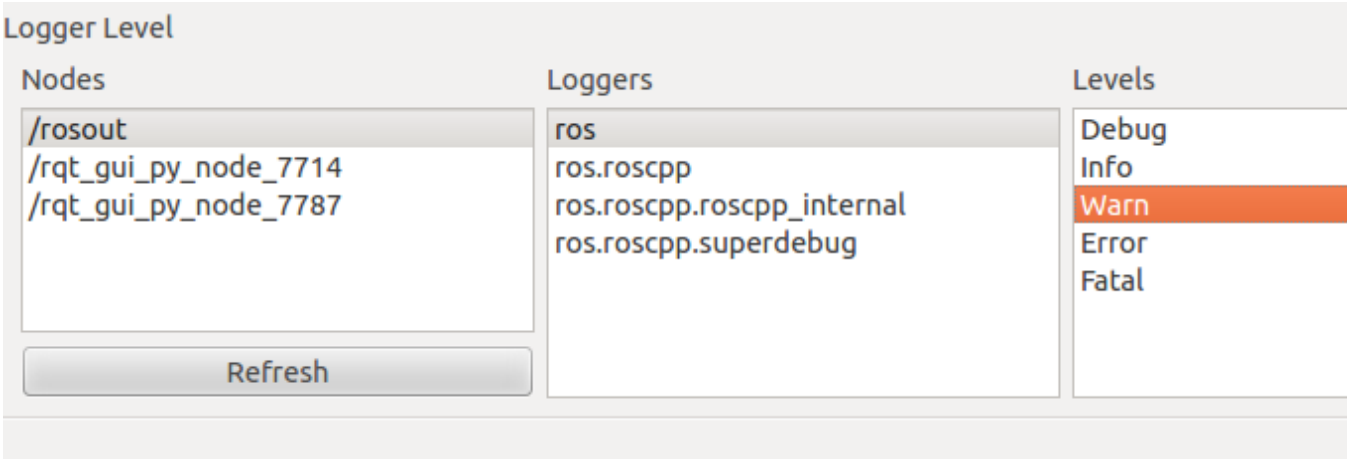
现在让我们在一个**新终端**中启动 turtlesim：

```
$ 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，如下图所示：



现在让我们刷新一下 rqt_logger_level 窗口并选择 Warn 将日志等级修改为 WARN，如下图所示：

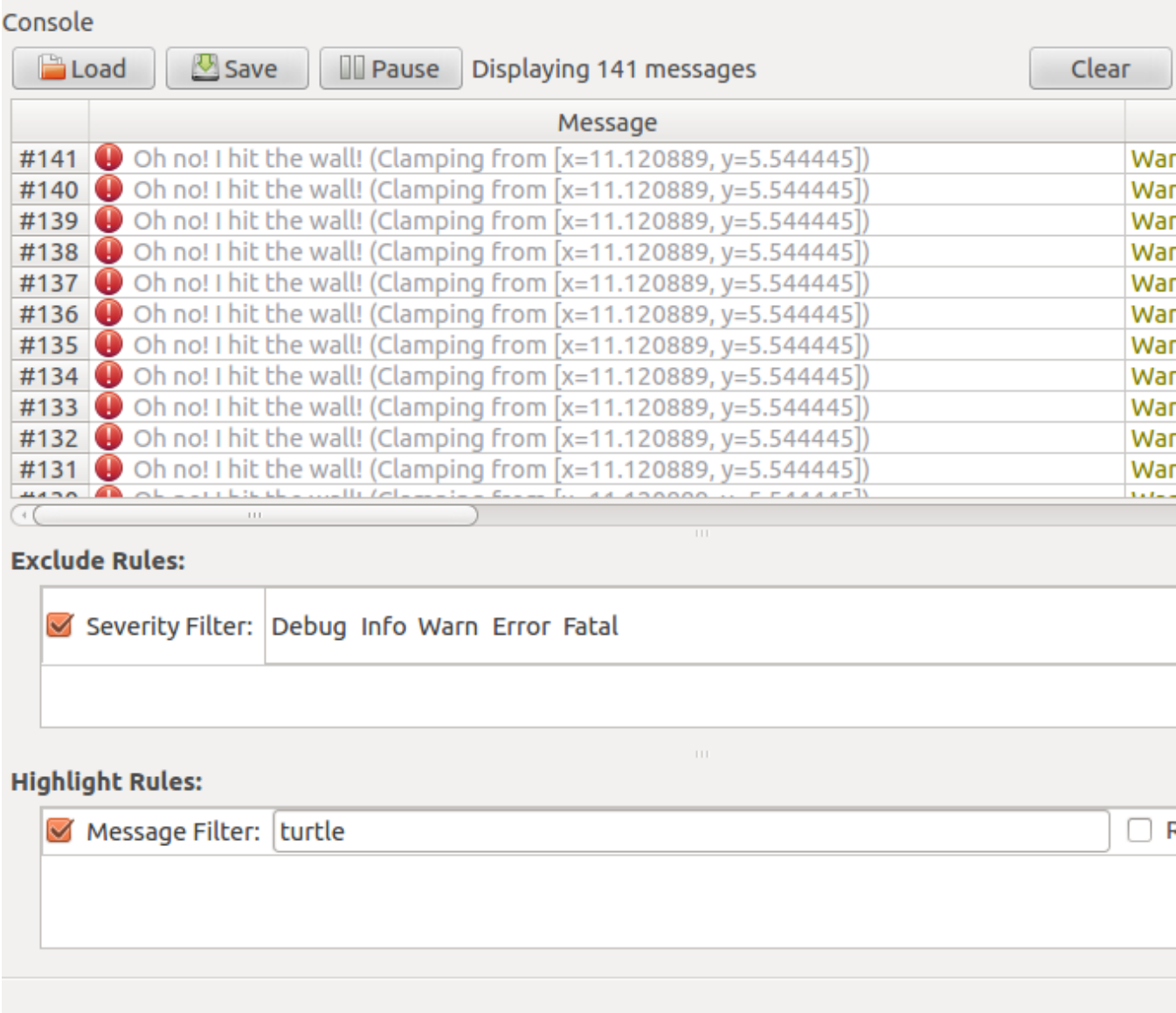


现在我们让 turtle 动起来并观察 rqt_console 中的输出（非 hydro 版）：

```
rostopic pub /turtle1/command_velocity turtlesim/Velocity -r 1 -- 2.0
 0.0
```

hydro 版：

```
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

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，将日志等级设为 Warn 时，你会得到 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。

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 beginner_tutorials 程序包目录下：

```
$ roscd beginner_tutorials
```

如果 roscd 执行失败了，记得设置你当前终端下的 ROS_PACKAGE_PATH 环境变量，设置方法如下：

```
$ export ROS_PACKAGE_PATH=~/<distro>_workspace/sandbox:$ROS_PACKAGE_PATH

$ roscd beginner_tutorials
```

如果你仍然无法找到 beginner_tutorials 程序包，说明该程序包还没有创建，那么请返回到 ROS/Tutorials/CreatingPackage 教程，并按照创建程序包的操作方法创建一个 beginner_tutorials 程序包。

然后创建一个 launch 文件夹：

```
$ mkdir launch

$ cd launch
```

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：

切换行号显示

```
<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 文件解析

现在我们开始逐句解析 launch xml 文件。

切换行号显示

```
<launch>
```

在这里我们以 launch 标签开头以表明这是一个 launch 文件。

切换行号显示

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

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

在这里我们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。

切换行号显示

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

```
  <remap from="output" to="turtlesim2/turtle1"/>
</node>
```

在这里我们启动模仿节点，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。

切换行号显示

```
</launch>
```

这个是 launch 文件的结束标签。

roslaunching

现在让我们通过 roslaunch 命令来启动 launch 文件：

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

现在将会有两个 turtlesims 被启动，然后我们在一个**新终端**中使用 rostopic 命令发送速度设定消息：

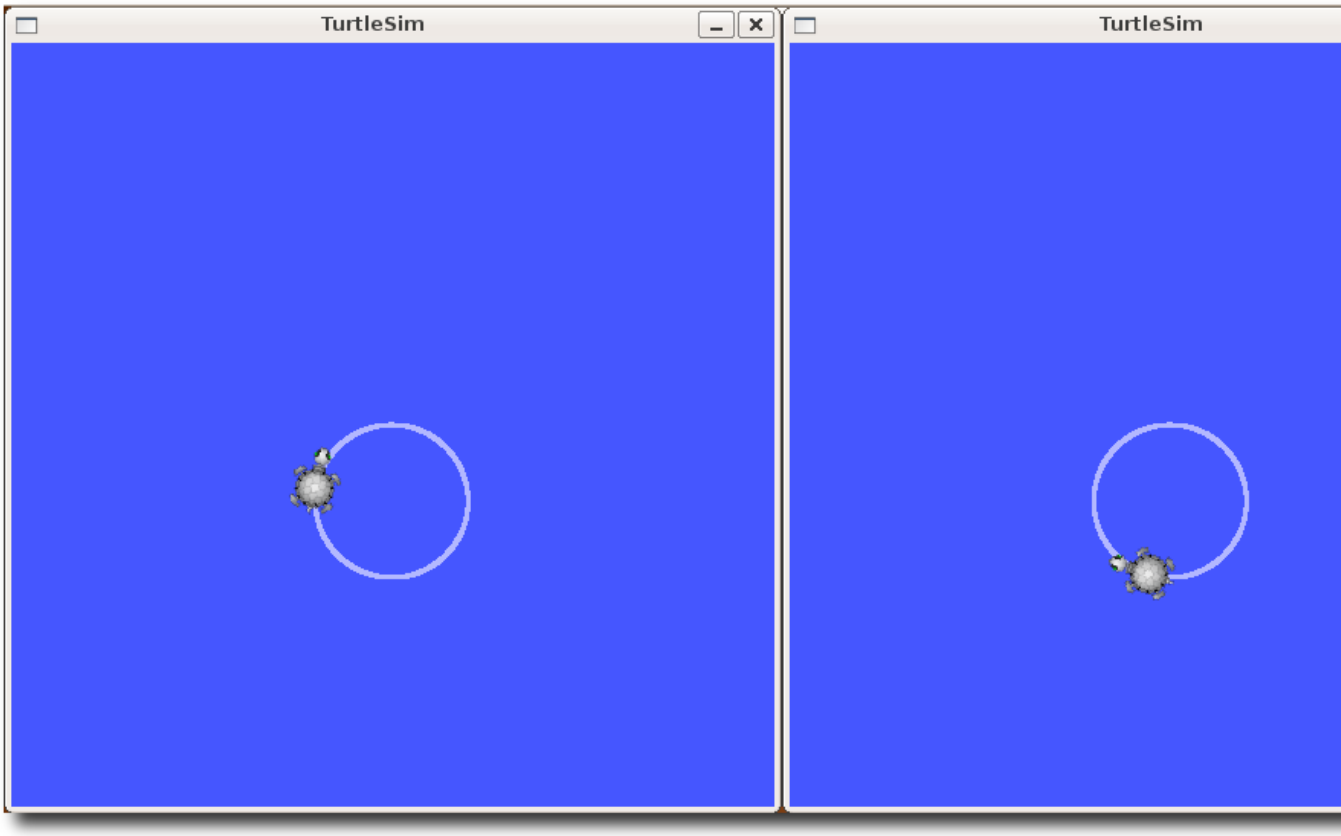
非 hydro 版：

```
$ rostopic pub /turtlesim1/turtle1/command_velocity turtlesim/Velocit
y -r 1 -- 2.0  -1.8
```

hydro 版：

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

你会看到两个 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。

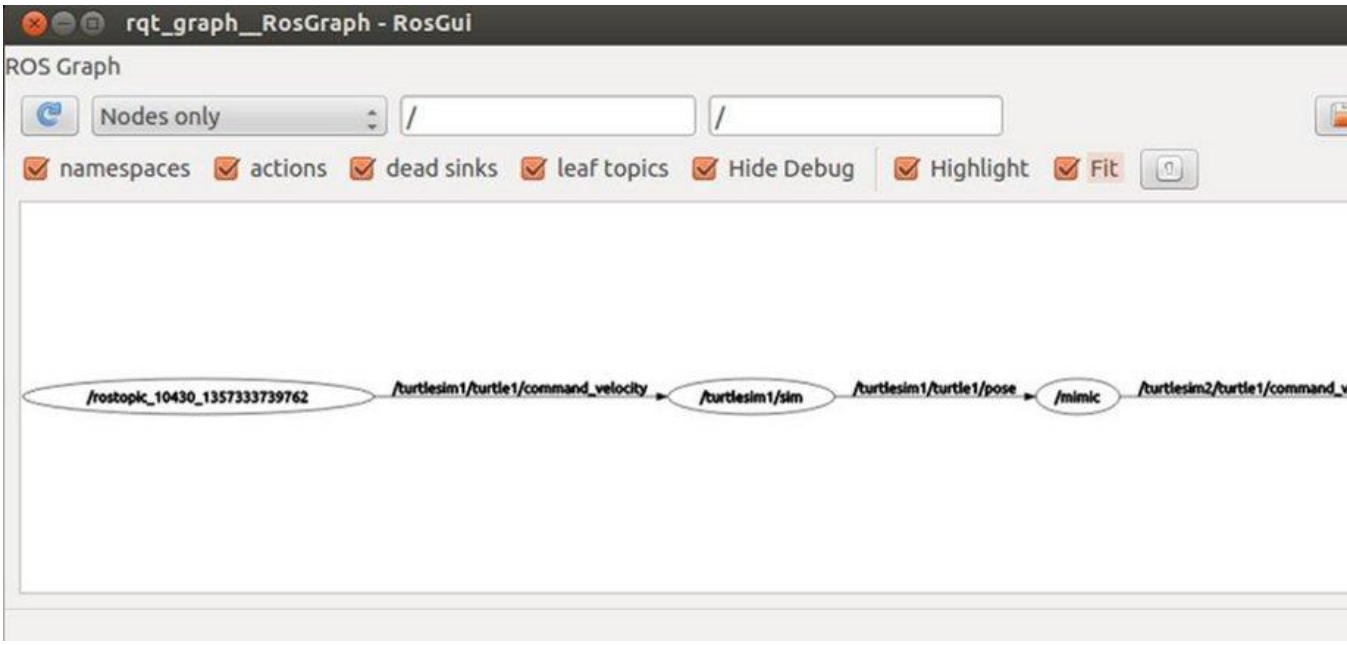


我们也可以通过 rqt_graph 来更好的理解在 launch 文件中所做的事情。运行 rqt 并在主窗口中选择 rqt_graph：

```
$ rqt
```

或者直接运行：

```
$ rqt_graph
```



到此，我们算是已经学会了 rqt_console 和 roslaunch 命令的使用，接下来我们开始学习使用 rosed——ROS 中的编辑器。现在你可以按 Ctrl-C 退出所有 turtlesims 节点了，因为在下一篇教程中你不会再用到它们。

9. 使用 rosed 编辑 ROS 中的文件

Description: 本教程将展示如何使用 rosed 来简化编辑过程。

Tutorial Level: BEGINNER

Next Tutorial: 创建 ROS 消息和 ROS 服务

目录

1. 使用 rosed
2. 使用 Tab 键补全文件名
3. 编辑器

使用 rosed

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。

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。

如果该实例没有运行成功，那么很有可能是你没有安装 vim 编辑器。请参考编辑器部分进行设置。

如果文件名在 package 里不是唯一的，那么会呈现出一个列表，让你选择编辑哪一个文件。

使用 Tab 键补全文件名

使用这个方法，在不知道准确文件名的情况下，你也可以看到并选择你所要编辑的文件。

使用方法:

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

编辑器

rosed 默认的编辑器是 vim。如果想要将其他的编辑器设置成默认的，你需要修改你的 ~/.bashrc 文件，增加如下语句:

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

这将 emacs 设置成为默认编辑器。

注意: *.bashrc 文件的改变，只会在新的终端才有效。已经打开的终端不受环境变量的影响。*

打开一个新的终端，看看那是否定义了 EDITOR:

```
$ echo $EDITOR
```

- `emacs -nw`

现在你已经成功设置并使用了 rosed，接下来我们将学习创建 ROS 消息和 ROS 服务.

10. 创建 ROS 消息和 ROS 服务

Description: 本教程详细介绍如何创建并编译 ROS 消息和服务，以及 rosmsg, rossrv 和 roscp 命令行工具的使用。

Tutorial Level: BEGINNER

Next Tutorial: 写一个简单的消息发布器和订阅器 (python) (c++)

catkinrosbuild

目 录

1.
1. 消息(msg)和服务(srv)介绍
2. 使用 msg
1. 创建一个 msg
2. 使用 rosmsg
3. 使用 srv
1. 创建一个 srv
2. 使用 rossrv
4. msg 和 srv 都需要的步骤
5. 获得帮助
6. 回顾
7. 下一个教程

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。

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 ~/catkin_ws/src/beginner_tutorials

$ 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，你可以仿造上面的形式多增加几行以得到更为复杂的消息：

```
string first_name

string last_name

uint8 age

uint32 score
```

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>
```

如果没有，添加进去。 注意，在构建的时候，我们只需要"message_generation"。然而，在运行的时候，我们只需要"message_runtime"。

在你最喜爱的编辑器中打开 CMakeLists.txt 文件(可以参考前边的教程 rosed).

在 CMakeLists.txt 文件中，利用 find_packag 函数，增加对 message_generation 的依赖，这样就可以生成消息了。 你可以直接在 COMPONENTS 的列表里增加 message_generation，就像这样：

```
# Do not just add this line to your CMakeLists.txt, modify the existin
g line

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 find_package,你也可以编译通过。这是因为 catkin 把你所有的 package 都整合在一起，因此，如果其他的 package 调用了 find_package，你的 package 的依赖就会是同样的配置。但是，在你单独编译时，忘记调用 find_package 会很容易出错。

同样，你需要确保你设置了运行依赖：

```
catkin_package(

  ...

  CATKIN_DEPENDS message_runtime ...

  ...)
```

找到如下代码块:

```
# add_message_files(

#   FILES

#   Message1.msg

#   Message2.msg

# )
```

去掉注释符号#，用你的.msg 文件替代 Message*.msg，就像下边这样：

```
add_message_files(

  FILES

  Num.msg

)
```

手动添加.msg 文件后，我们要确保 CMake 知道在什么时候重新配置我们的 project。 确保添加了如下代码:

```
generate_messages()
```

现在，你可以生成自己的消息源代码了。如果你想立即实现，那么就跳过以下部分，到 Common step for msg and srv.

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 beginner_tutorials/Num
```

你将会看到:

```
- int64 num
```

在上边的样例中,消息类型包含两部分：

- `beginner_tutorials` -- 消息所在的 package
- `Num` -- 消息名 `Num`.

如果你忘记了消息所在的 package，你也可以省略掉 package 名。输入：

```
$ rosmsg show Num
```

你将会看到:

```
- [beginner_tutorials/Num]:
- int64 num
```

使用 srv

创建一个 srv

在刚刚那个 package 中创建一个服务：

```
$ roscd beginner_tutorials
```

```
$ 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。

现在认为，你已经如前边所介绍的，在 CMakeLists.txt 文件中增加了对 message_generation 的依赖。:

```
# Do not just add this line to your CMakeLists.txt, modify the existin
g line

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

(对的, message_generation 对 msg 和 srv 都起作用)

同样，跟 msg 文件类似，你也需要在 package.xml 文件中做一些修改。查看上边的说明，增加额外的依赖项。

删掉#，去除对下边语句的注释:

```
# add_service_files(

#   FILES

#   Service1.srv

#   Service2.srv

# )
```

用你自己的 srv 文件名替换掉那些 Service*.srv 文件:

```
add_service_files(

  FILES

  AddTwoInts.srv

)
```

现在，你可以生成自己的服务源代码了。如果你想立即实现，那么就跳过以下部分，到 Common step for msg and srv.

使用 rossrv

以上就是创建一个服务所需的所有步骤。下面通过 rosmsg 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 beginner_tutorials/AddTwoInts
```

你将会看到:

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

跟 rosmsg 类似，你也可以不指定具体的 package 名来查找服务文件：

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

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

msg 和 srv 都需要的步骤

接下来，在 CMakeLists.txt 中找到如下部分:

```
# generate_messages(

#   DEPENDENCIES

# #  std_msgs  # Or other packages containing msgs

# )
```

去掉注释并附加上所有你消息文件所依赖的那些含有.msg 文件的 package（这个例子是依赖 std_msgs,不要添加 roscpp,rospy)，结果如下:

```
generate_messages(

  DEPENDENCIES

  std_msgs

)
```

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~/catkin_ws/devel/include/beginner_tutorials/。 Python 脚本语言会在 ~/catkin_ws/devel/lib/python2.7/dist-packages/beginner_tutorials/msg 目录下创建。 lisp 文件会出现在 ~/catkin_ws/devel/share/common-lisp/ros/beginner_tutorials/msg/ 路径下.

详尽的消息格式请参考 Message Description Language 页面.

获得帮助

我们已经接触到不少的 ROS 工具了。有时候很难记住他们所需要的参数。还好大多数 ROS 工具都提供了帮助。

输入:

```
$ rosmsg -h
```

- 你可以看到一系列的 rosmsg 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
```

同样你也可以获得子命令的帮助：

```
$ rosmsg show -h
```

- 这会现实 rosmsg show 所需的参数：

```
• Usage: rosmsg show [options] <message type>
•
• Options:
•   -h, --help  show this help message and exit
•   -r, --raw   show raw message text, including comments
```

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 : **c**hanges **d**irectory to a ROS package or stack
- rosls = ros+ls : **l**ist**s** files in a ROS package
- roscp = ros+cp : **cop**ies 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

下一个教程

既然已经学习了如何创建 ROS 消息和服务，接下来就就将学习如何编写简单的发布器和订阅器(python) (c++).

11. 编写简单的消息发布器和订阅器 (C++)

Description: 本教程将介绍如何编写 C++的发布器节点和订阅器节点。

Tutorial Level: BEGINNER

Next Tutorial: 测试消息发布器和订阅器

catkinrosbuild

目 录

1. 编写发布器节点
1. 源代码
2. 代码解释
2. 编写订阅器节点
1. 源代码
2. 代码解释
3. 编译节点
4. 编译节点

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。

转移到之前教程在 catkin 工作空间所创建的 beginner_tutorials package 路径下:

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

源代码

在 beginner_tutorials package 路径下创建 src 目录:

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

这个目录将会存储 beginner_tutorials package 的所有源代码.

在 beginner_tutorials package 里创建 src/talker.cpp 文件，并粘贴如下代码：

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

切换行号显示

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

#include <sstream>

/**
 * This tutorial demonstrates simple sending of messages over the 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 using 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 ROS
```

```
 59    * master node, which keeps a registry of who is publishing and
who
 60    * is subscribing. After this advertise() call is made, the mast
er
 61    * node will notify anyone who is trying to subscribe to this to
pic name,
 62    * and they will in turn negotiate a peer-to-peer connection wit
h this
 63    * node.  advertise() returns a Publisher object which allows yo
u to
 64    * publish messages on that topic through a call to publish().
Once
 65    * all copies of the returned Publisher object are destroyed, th
e topic
 66    * will be automatically unadvertised.
 67    *
 68    * The second parameter to advertise() is the size of the messag
e queue
 69    * used for publishing messages.  If messages are published more
 quickly
 70    * than we can send them, the number here specifies how many mes
sages to
 71    * buffer up before throwing some away.
 72    */
 73   ros::Publisher chatter_pub = n.advertise<std_msgs::String>("cha
tter", 1000);
 74
 75   ros::Rate loop_rate(10);
 76
 77   /**
 78    * A count of how many messages we have sent. This is used to cre
ate
 79    * a unique string for each message.
 80    */
 81   int count = 0;
 82   while (ros::ok())
 83   {
 84     /**
 85      * This is a message object. You stuff it with data, and then p
ublish it.
 86      */
 87     std_msgs::String msg;
 88
 89     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 parameter
     * 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，我们来分段解释代码.

切换行号显示

```
#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。

切换行号显示

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

```

这引用了 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。需要更详细的消息定义,参考 msg 页面.

切换行号显示

```
  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——目前，这不是重点。同样，我们也在这里指定我们节点的名称——必须唯一。

这里的名称必须是一个 base name，不能包含/。

切换行号显示

```
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。

切换行号显示

```
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,它有两个作用: 1) 它有一个 publish()成员函数可以让你在 topic 上发布消息； 2) 如果消息类型不对,它会拒绝发布。

切换行号显示

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

ros::Rate 对象可以允许你指定自循环的频率。它会追踪记录自上一次调用 Rate::sleep()后时间的流逝，并休眠直到一个频率周期的时间。

在这个例子中，我们让它以 10hz 的频率运行。

切换行号显示

```
int count = 0;
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。

ros::ok()返回 false，如果下列条件之一发生：

- SIGINT 接收到(Ctrl-C)
- 被另一同名节点踢出 ROS 网络
- ros::shutdown()被程序的另一部分调用
- 所有的 ros::NodeHandles 都已经被销毁

一旦 ros::ok()返回 false，所有的 ROS 调用都会失效。

切换行号显示

```
  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"。当然你也可以发布更复杂的消息类型。

切换行号显示
```
101     chatter_pub.publish(msg);
```

现在我们已经向所有连接到 chatter topic 的节点发送了消息。

切换行号显示
```
 93     ROS_INFO("%s", msg.data.c_str());
```

ROS_INFO 和类似的函数用来替代 printf/cout. 参考 rosconsole documentation 以获得更详细的信息。

切换行号显示
```
103     ros::spinOnce();
```

在这个例子中并不是一定要调用 ros::spinOnce()，因为我们不接受回调。然而，如果你想拓展这个程序，却又没有在这调用 ros::spinOnce()，你的回调函数就永远也不会被调用。所以，在这里最好还是加上这一语句。

切换行号显示
```
105     loop_rate.sleep();
```

这条语句是调用 ros::Rate 对象来休眠一段时间以使得发布频率为 10hz。

对上边的内容进行一下总结：

- 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

接下来我们要编写一个节点来接收消息。

编写订阅器节点

源代码

在 beginner_tutorials package 目录下创建 src/listener.cpp 文件，并粘贴如下代码：

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

切换行号显示
```
 28 #include "ros/ros.h"
 29 #include "std_msgs/String.h"
 30
 31 /**
 32  * This tutorial demonstrates simple receipt of messages over the
ROS system.
 33  */
```

```
34 void chatterCallback(const std_msgs::String::ConstPtr& msg)
35 {
36   ROS_INFO("I heard: [%s]", msg->data.c_str());
37 }
38
39 int main(int argc, char **argv)
40 {
41   /**
42    * The ros::init() function needs to see argc and argv so that i
t can perform
43    * any ROS arguments and name remapping that were provided at th
e command line. For programmatic
44    * remappings you can use a different version of init() which ta
kes remappings
45    * directly, but for most command-line programs, passing argc an
d argv is the easiest
46    * way to do it.  The third argument to init() is the name of the
 node.
47    *
48    * You must call one of the versions of ros::init() before using
 any other
49    * part of the ROS system.
50    */
51   ros::init(argc, argv, "listener");
52
53   /**
54    * NodeHandle is the main access point to communications with th
e ROS system.
55    * The first NodeHandle constructed will fully initialize this n
ode, and the last
56    * NodeHandle destructed will close down the node.
57    */
58   ros::NodeHandle n;
59
60   /**
61    * The subscribe() call is how you tell ROS that you want to rece
ive messages
62    * on a given topic.  This invokes a call to the ROS
63    * master node, which keeps a registry of who is publishing and
who
64    * is subscribing.  Messages are passed to a callback function,
here
65    * called chatterCallback.  subscribe() returns a Subscriber obj
ect that you
```

```
 66    * must hold on to until you want to unsubscribe.  When all copie
s of the Subscriber
 67    * object go out of scope, this callback will automatically be u
nsubscribed from
 68    * this topic.
 69    *
 70    * The second parameter to the subscribe() function is the size
of the message
 71    * queue.  If messages are arriving faster than they are being p
rocessed, this
 72    * is the number of messages that will be buffered up before beg
inning to throw
 73    * away the oldest ones.
 74    */
 75   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallb
ack);
 76
 77   /**
 78    * ros::spin() will enter a loop, pumping callbacks.  With this
version, all
 79    * callbacks will be called from within this thread (the main on
e).  ros::spin()
 80    * will exit when Ctrl-C is pressed, or the node is shutdown by t
he master.
 81    */
 82   ros::spin();
 83
 84   return 0;
 85 }
```

代码解释

下面我们将逐条解释代码，当然，之前解释过的代码就不再赘述了。

切换行号显示

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

这是一个回调函数，当消息到达 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

切换行号显示

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

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。

NodeHandle::subscribe()返回 ros::Subscriber 对象,你必须让它处于活动状态直到你不再想订阅该消息。当这个对象销毁时，它将自动退订 上的消息。

有各种不同的 NodeHandle::subscribe()函数，允许你指定类的成员函数，甚至是 Boost.Function 对象可以调用的任何数据类型。roscpp overview 提供了更为详尽的信息。

切换行号显示

```
82   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。

还有其他的方法进行回调，但在这里我们不涉及。想要了解，可以参考 roscpp_tutorials package 里的一些 demo 应用。需要更为详尽的信息，参考 roscpp overview。

下边，我们来总结一下:

- 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

编译节点

之前教程中使用 catkin_create_pkg 创建了 package.xml 和 CMakeLists.txt 文件。

生成的 CMakeLists.txt 看起来应该是这样(在 Creating Msgs and Srvs 教程中的修改和未被使用的注释和例子都被移除了):

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

切换行号显示

```
1 cmake_minimum_required(VERSION 2.8.3)
2 project(beginner_tutorials)
3
4 ## Find catkin and any catkin packages
5 find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs gen
msg)
6
7 ## Declare ROS messages and services
8 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()
```

在 CMakeLists.txt 文件末尾加入几条语句:

```
include_directories(include ${catkin_INCLUDE_DIRS})


add_executable(talker src/talker.cpp)

target_link_libraries(talker ${catkin_LIBRARIES})


add_executable(listener src/listener.cpp)

target_link_libraries(listener ${catkin_LIBRARIES})
```

结果，CMakeLists.txt 文件看起来像这样:

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

切换行号显示

```
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 gen
msg)

## 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_cpp)
```

这会生成两个可执行文件, talker 和 listener, 默认存储到 devel space 目录,具体是在 ~/catkin_ws/devel/lib/share/<package name>中.

现在要为可执行文件添加对生成的消息文件的依赖:

```
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
```

这样就可以确保自定义消息的头文件在被使用之前已经被生成。因为 catkin 把所有的 package 并行的编译，所以如果你要使用其他 catkin 工作空间中其他 package 的消息，你同样也需要添加对他们各自生成的消息文件的依赖。当然，如果在*Groovy*版本下，你可以使用下边的这个变量来添加对所有必须的文件依赖:

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

你可以直接调用可执行文件，也可以使用 rosrun 来调用他们。他们不会被安装到 '<prefix>/bin'路径下，因为那样会改变系统的 PATH 环境变量。如果你确定要将可执行文件安装到该路径下，你需要设置安装目标，请参考 catkin/CMakeLists.txt

需要关于 CMakeLists.txt 更详细的信息，请参考 catkin/CMakeLists.txt

现在运行 catkin_make:

```
# In your catkin workspace

$ catkin_make
```

注意:如果你是添加了新的 package，你需要通过--force-cmake 选项告诉 catkin 进行强制编译。参考 catkin/Tutorials/using_a_workspace#With_catkin_make.

既然已经编写好了发布器和订阅器，下面让我们来测试消息发布器和订阅器.

12. 写一个简单的消息发布器和订阅器(Python)

Description: 本教程将通过 Python 编写一个发布器节点和订阅器节点。

Tutorial Level: BEGINNER

Next Tutorial: 执行你的消息发布器和订阅器

catkinrosbuild

目录

1. Writing the Publisher Node
1. The Code
2. The Code Explained
2. Writing the Subscriber Node
1. The Code
2. The Code Explained
3. Building your nodes

Writing the Publisher Node

"Node" is the ROS term for an executable that is connected to the ROS network. Here we'll create the publisher ("talker") node which will continually broadcast a message.

Change directory into the beginner_tutorials package, you created in the earlier tutorial, creating a package:

```
$ roscd beginner_tutorials
```

The Code

First lets create a 'scripts' folder to store our Python scripts in:

```
$ mkdir scripts

$ cd scripts
```

Then download the example script talker.py to your new `scripts` directory and make it executable:

```
$ wget https://raw.github.com/ros/ros_tutorials/indigo-devel/rospy_tutorials/001_talker_listener/talker.py

$ chmod +x talker.py
```

You can view and edit the file with `$ rosed beginner_tutorials talker.py` or just look below.

切换行号显示

```python
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

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

The Code Explained

Now, let's break the code down.

切换行号显示

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

Every Python ROS Node will have this declaration at the top. The first line makes sure your script is executed as a Python script.

切换行号显示

```python
import rospy
from std_msgs.msg import String
```

You need to import `rospy` if you are writing a ROS Node. The `std_msgs.msg` import is so that we can reuse the`std_msgs/String` message type (a simple string container) for publishing.

切换行号显示

```python
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
```

This section of code defines the talker's interface to the rest of ROS.`pub = rospy.Publisher("chatter", String, queue_size=10)` declares that your node is publishing to the `chatter` topic using the message type `String`. `String` here is actually the class `std_msgs.msg.String`.
The `queue_size` argument is **New in ROS hydro** and limits the amount of queued

messages if any subscriber is not receiving the them fast enough. In older ROS distributions just omit the argument.

The next line, `rospy.init_node(NAME)`, is very important as it tells rospy the name of your node -- until rospy has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name `talker`. NOTE: the name must be a base name, i.e. it cannot contain any slashes "/".

切换行号显示

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

This line creates a `Rate` object `r`. With the help of its method `sleep()`, it offers a convenient way for looping at the desired rate. With its argument of `10`, we should expect to go through the loop 10 times per second (as long as our processing time does not exceed 1/10th of a second!)

切换行号显示

```
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()
```

This loop is a fairly standard rospy construct: checking the `rospy.is_shutdown()` flag and then doing work. You have to check `is_shutdown()` to check if your program should exit (e.g. if there is a `Ctrl-C` or otherwise). In this case, the "work" is a call to `pub.publish(String(str))` that publishes to our `chatter` topic using a newly created `String` message. The loop calls `r.sleep()`, which sleeps just long enough to maintain the desired rate through the loop.

(You may also run across `rospy.sleep()` which is similar to `time.sleep()` except that it works with simulated time as well (see Clock).)

This loop also calls `rospy.loginfo(str)`, which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout. rosout is a handy for debugging: you can pull up messages usingrqt_console instead of having to find the console window with your Node's output.

`std_msgs.msg.String` is a very simple message type, so you may be wondering what it looks like to publish more complicated types. The general rule of thumb is that *constructor args are in the same order as in the `.msg` file*. You can also pass in no arguments and initialize the fields directly, e.g.

```
msg = String()

msg.data = str
```

or you can initialize some of the fields and leave the rest with default values:

```
String(data=str)
```

You may be wondering about the last little bit:

```
切换行号显示
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

In addition to the standard Python `__main__` check, this catches a `rospy.ROSInterruptException` exception, which can be thrown by `rospy.sleep()` and `rospy.Rate.sleep()` methods when `Ctrl-C` is pressed or your Node is otherwise shutdown. The reason this exception is raised is so that you don't accidentally continue executing code after the `sleep()`.

Now we need to write a node to receive the messages.

Writing the Subscriber Node

The Code

Download the listener.py file into your scripts directory:

```
$ roscd beginner_tutorials/scripts/

$ wget https://raw.github.com/ros/ros_tutorials/indigo-devel/rospy_tutorials/001_talker_listener/listener.py
```

The file contents look close to:

```
切换行号显示
#!/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():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

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

```
    # spin() simply keeps python from exiting until this node is s
topped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Don't forget to make the node executable:

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

The Code Explained

The code for `listener.py` is similar to `talker.py`, except we've introduced a new callback-based mechanism for subscribing to messages.

```
切换行号显示
    rospy.init_node('listener', anonymous=True)

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

    # spin() simply keeps python from exiting until this node is s
topped
    rospy.spin()
```

This declares that your node subscribes to the `chatter` topic which is of type `std_msgs.msgs.String`. When new messages are received, `callback` is invoked with the message as the first argument.

We also changed up the call to `rospy.init_node()` somewhat. We've added the `anonymous=True` keyword argument. ROS requires that each node have a unique name. If a node with the same name comes up, it bumps the previous one. This is so that malfunctioning nodes can easily be kicked off the network. The `anonymous=True` flag tells `rospy` to generate a unique name for the node so that you can have multiple `listener.py` nodes run easily.

The final addition, `rospy.spin()` simply keeps your node from exiting until the node has been shutdown. Unlike roscpp, rospy.spin() does not affect the subscriber callback functions, as those have their own threads.

Building your nodes

We use CMake as our build system and, yes, you have to use it even for Python nodes. This is to make sure that the autogenerated Python code for messages and services is created.

Go to your catkin workspace and run `catkin_make`:

```
$ cd ~/catkin_ws

$ catkin_make
```

Now that you have written a simple publisher and subscriber, let's examine the simple publisher and subscriber.

13. 测试消息发布器和订阅器

Description: 本教程将测试上一教程所写的消息发布器和订阅器。

Tutorial Level: BEGINNER

Next Tutorial: 写一个简单的服务端和客户端 (python) (c++)

目录

1. 启动发布器
2. 启动订阅器

启动发布器

确保 roscore 可用，并运行：

```
$ roscore
```

catkin specific 如果使用 catkin，确保你在调用 `catkin_make` 后，在运行你自己的程序前，已经 source 了 catkin 工作空间下的 setup.sh 文件：

```
# In your catkin workspace

$ cd ~/catkin_ws

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

In the last tutorial we made a publisher called "talker". Let's run it:

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

$ rosrun beginner_tutorials talker.py   (Python)
```

你将看到如下的输出信息:

```
• [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
```

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。

启动订阅器

上一教程，我们编写了一个名为"listener"的订阅器节点。现在运行它：

```
$ rosrun beginner_tutorials listener     (C++)
$ rosrun beginner_tutorials listener.py  (Python)
```

你将会看到如下的输出信息:

```
• [INFO] [WallTime: 1314931969.258941] /listener_17657_1314931968
  795I heard hello world 1314931969.26
• [INFO] [WallTime: 1314931970.262246] /listener_17657_1314931968
  795I heard hello world 1314931970.26
• [INFO] [WallTime: 1314931971.266348] /listener_17657_1314931968
  795I heard hello world 1314931971.26
• [INFO] [WallTime: 1314931972.270429] /listener_17657_1314931968
  795I heard hello world 1314931972.27
• [INFO] [WallTime: 1314931973.274382] /listener_17657_1314931968
  795I heard hello world 1314931973.27
• [INFO] [WallTime: 1314931974.277694] /listener_17657_1314931968
  795I heard hello world 1314931974.28
• [INFO] [WallTime: 1314931975.283708] /listener_17657_1314931968
  795I heard hello world 1314931975.28
```

你已经测试完了发布器和订阅器，下面我们来编写一个服务和客户端(python) (c++).

14. 编写简单的Service和Client (C++)

Description: 本教程介绍如何用 C++编写 Service 和 Client 节点。

Tutorial Level: BEGINNER

Next Tutorial: 测试简单的 Service 和 Client

catkinrosbuild

目录

1. 编写 Service 节点
1. 代码
2. 代码解释
2. 编写 Client 节点
1. 代码
2. 代码解释
3. 编译节点
4. 编译节点

编写 Service 节点

这里，我们将创建一个简单的 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。

进入先前你在 catkin workspace 教程中所创建的 beginner_tutorials 包所在的目录：

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

请确保已经按照 creating the AddTwoInts.srv 教程的步骤创建了本教程所需要的 srv（确保选择了对应的编译系统“catkin”和“rosbuild”）。

代码

在 beginner_tutorials 包中创建 src/add_two_ints_server.cpp 文件，并复制粘贴下面的代码：

切换行号显示

```
#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)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}
```

```
12
13 int main(int argc, char **argv)
14 {
15   ros::init(argc, argv, "add_two_ints_server");
16   ros::NodeHandle n;
17
18   ros::ServiceServer service = n.advertiseService("add_two_ints",
add);
19   ROS_INFO("Ready to add two ints.");
20   ros::spin();
21
22   return 0;
23 }
```

代码解释

现在，让我们来逐步分析代码。

切换行号显示

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3
```

beginner_tutorials/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。

切换行号显示

```
4 bool add(beginner_tutorials::AddTwoInts::Request  &req,
5         beginner_tutorials::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。

切换行号显示

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

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。

切换行号显示

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

这里，service 已经建立起来，并在 ROS 内发布出来。

编写 Client 节点

代码

在 beginner_tutorials 包中创建 src/add_two_ints_client.cpp 文件，并复制粘贴下面的代码：

切换行号显示

```
#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::ServiceClient 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，让我们来逐步分析代码。

切换行号显示

```
ros::ServiceClient client = n.serviceClient<beginner_tutorials::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。

切换行号显示

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

这里，我们实例化一个由 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。

切换行号显示

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

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。

编译节点

再来编辑一下 beginner_tutorials 里面的 CMakeLists.txt，文件位于 ~/catkin_ws/src/beginner_tutorials/CMakeLists.txt，并将下面的代码添加在文件末尾：

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

切换行号显示

```
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)
```

这段代码将生成两个可执行程序"add_two_ints_server"和"add_two_ints_client"，这两个可执行程序默认被放在你的 devel space 下的包目录下，默认为 ~/catkin_ws/devel/lib/share/<package name>。你可以直接调用可执行程序，或者使用 rosrun 命令去调用它们。它们不会被装在<prefix>/bin 目录下，因为当你在你的系统里安装这个包的时候，这样做会污染 PATH 变量。如果你希望在安装的时候你的可执行程序在 PATH 变量里面，你需要设置一下 install target，请参考：catkin/CMakeLists.txt

关于 CMakeLists.txt 文件更详细的描述请参考：catkin/CMakeLists.txt

现在运行 catkin_make 命令：

```
# In your catkin workspace

cd ~/catkin_ws

catkin_make
```

如果你的编译过程因为某些原因而失败：

- 确保你已经依照先前的 creating the AddTwoInts.srv 教程里的步骤完成操作。

现在你已经学会如何编写简单的 Service 和 Client，开始测试简单的 Service 和 Client 吧。

15. 测试简单的 Service 和 Client

Description: 本教程将测试之前所写的 Service 和 Client。

Tutorial Level: BEGINNER

Next Tutorial: 记录与回放数据

目 录

1. 运行 Service
2. 运行 Client
3. 关于 Service 和 Client 节点的更多例子

运行 Service

让我们从运行 Service 开始：

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

$ rosrun beginner_tutorials add_two_ints_server.py  (Python)
```

你将会看到如下类似的信息：

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

运行 Client

现在，运行 Client 并附带一些参数：

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

$ rosrun beginner_tutorials add_two_ints_client.py 1 3  (Python)
```

你将会看到如下类似的信息：

```
request: x=1, y=3

sending back response: [4]
```

现在，你已经成功地运行了你的第一个 Service 和 Client 程序，可以开始学习如何记录与回放数据了.

关于 Service 和 Client 节点的更多例子

如果你想做更深入的研究，或者是得到更多的操作示例，你可以从这个链接找到 here. 一个简单的 Client 与 Service 的组合程序演示了自定义消息类型的使用方法. 如果 Service 节点是用 C++写的，写 Client 用 C++，Python 或者是 LISP 都可以.

16. 录制与回放数据

Description: 本教程将教你如何将 ROS 系统运行过程中的数据录制到一个.bag 文件中，然后通过回放数据来重现相似的运行过程。

Keywords: data, rosbag, record, play, info, bag

Tutorial Level: BEGINNER

Next Tutorial: 使用 roswtf

目 录

1. 录制数据（通过创建一个 bag 文件）
1. 录制所有发布的话题
2. 检查并回放 bag 文件
3. 录制数据子集
4. rosbag record/play 命令的局限性

录制数据（通过创建一个 bag 文件）

本小节将教你如何记录 ROS 系统运行时的话题数据，记录的话题数据将会累积保存到 bag 文件中。

首先，执行以下命令：

```
roscore

rosrun turtlesim turtlesim_node
```

```
rosrun turtlesim turtle_teleop_key
```

以上操作将会启动两个节点——一个 turtlesim 可视化节点和一个 turtlesim 键盘控制节点。在运行 turtlesim 键盘控制节点的终端窗口中你应该会看到如下类似信息：

```
Reading from keyboard

---------------------------

Use arrow keys to move the turtle.
```

这时按下键盘上的方向键应该会让 turtle 运动起来。需要注意的是要想控制 turtle 运动你必须先选中启动 turtlesim 键盘控制节点时所在的终端窗口而不是显示虚拟 turtle 所在的窗口。

录制所有发布的话题

首先让我们来检查看当前系统中发布的所有话题。要完成此操作请打开一个新终端并执行：

```
rostopic list -v
```

这应该会生成以下输出：

```
Published topics:

 * /turtle1/color_sensor [turtlesim/Color] 1 publisher

 * /turtle1/command_velocity [turtlesim/Velocity] 1 publisher

 * /rosout [roslib/Log] 2 publishers

 * /rosout_agg [roslib/Log] 1 publisher

 * /turtle1/pose [turtlesim/Pose] 1 publisher


Subscribed topics:

 * /turtle1/command_velocity [turtlesim/Velocity] 1 subscriber

 * /rosout [roslib/Log] 1 subscriber
```

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/command_velocity 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。

现在我们开始录制。打开一个新的终端窗口，在终端中执行以下命令：

```
mkdir ~/bagfiles
```

```
cd ~/bagfiles

rosbag record -a
```

在这里我们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，你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。

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，接下来我们可以使用 **rosbag info** 检查看它的内容，使用 **rosbag play** 命令回放出来。接下来我们首先会看到在 bag 文件中都录制了哪些东西。我们可以使用 **info** 命令，该命令可以检查看 bag 文件中的内容而无需回放出来。在 bag 文件所在的目录下执行以下命令：

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

你应该会看到如下类似信息：

```
bag: 2009-12-04-15-02-56.bag

version: 1.2

start_time: 1259967777871383000

end_time: 1259967797238692999

length: 19367309999

topics:

  - name: /rosout

    count: 2

    datatype: roslib/Log

    md5sum: acffd30cd6b6de30f120938c17c593fb

  - name: /turtle1/color_sensor

    count: 1122

    datatype: turtlesim/Color

    md5sum: 353891e354491c51aabe32df673fb446

  - name: /turtle1/command_velocity
```

```
    count: 23

    datatype: turtlesim/Velocity

    md5sum: 9d5c2dcd348ac8f76ce2a4307bd63a13

  - name: /turtle1/pose

    count: 1121

    datatype: turtlesim/Pose

    md5sum: 863b248d5016ca62ea2e895ae5265cf9
```

这些信息告诉你 bag 文件中所包含话题的名称、类型和消息数量。我们可以看到，在之前使用 **rostopic** 命令查看到的五个已公告的话题中，其实只有其中的四个在我们录制过程中发布了消息。因为我们带-a 参数选项运行 **rosbag record** 命令时系统会录制下所有节点发布的所有消息。

下一步是回放 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 <your bagfile>
```

在这个窗口中你应该会立即看到如下类似信息：

```
Hit space to pause.

[ INFO] 1260210510.566003000: Sleeping 0.200 seconds after advertising
 /rosout...

[ INFO] 1260210510.766582000: Done sleeping.


[ INFO] 1260210510.872197000: Sleeping 0.200 seconds after advertising
 /turtle1/pose...

[ INFO] 1260210511.072384000: Done sleeping.


[ INFO] 1260210511.277391000: Sleeping 0.200 seconds after advertising
 /turtle1/color_sensor...

[ INFO] 1260210511.477525000: Done sleeping.
```

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。

最终/turtle1/command_velocity 话题将会被发布，同时在 turtuelsim 虚拟画面中 turtle 应该会像之前你通过 turtle_teleop_key 节点控制它一样开始移动。从运行 **rosbag play** 到 turtle 开始移动时所经历时间应该近似等于之前在本教程开始部分运行 **rosbag record** 后到开始按下键盘发出控制命令时所经历时间。你可以通过-s 参数选项让 **rosbag play** 命令等待一段时间跳过 bag 文件初始部分后再真正开始回放。最后一个可能比较有趣的参数选项是-r 选项，它允许你通过设定一个参数来改变消息发布速率。如果你执行：

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

你应该会看到 turtle 的运动轨迹有点不同了，这时的轨迹应该是相当于当你以两倍的速度通过按键发布控制命令时产生的轨迹。

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（relaunch）键盘控制节点相关的启动文件（launch file)：

```
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**）。你应该会看到如下类似信息，里面只包含录制时指定的话题：

```
bag: subset.bag

version: 1.2

start_time: 3196900000000

end_time: 3215400000000

length: 18500000000

topics:
```

```
  - name: /turtle1/command_velocity
    count: 8
    datatype: turtlesim/Velocity
    md5sum: 9d5c2dcd348ac8f76ce2a4307bd63a13
  - name: /turtle1/pose
    count: 1068
    datatype: turtlesim/Pose
    md5sum: 863b248d5016ca62ea2e895ae5265cf9
```

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。

现在你已经学会了如何录制和回放数据，接下来我们开始学习如何使用 roswtf 来检查系统故障

17. roswtf 入门

Description: 本教程介绍了 roswtf 工具的基本使用方法。

Keywords: roswtf

Tutorial Level: BEGINNER

Next Tutorial: wiki 导航

目 录

1. 安装检查
2. 运行时检查（在有 ROS 节点运行时）
3. 错误报告

在你开始本教程之前请确保 roscore 没在运行。

安装检查

roswtf 可以检查你的 ROS 系统并尝试发现问题，我们来试看：

```
$ roscd

$ roswtf
```

你应该会看到（各种详细的输出信息）：

```
Stack: ros

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

Static checks summary:


No errors or warnings

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


Cannot communicate with master, ignoring graph checks
```

如果你的 ROS 安装没问题，你应该会看到类似上面的输出信息，它的含义是： * "Stack: ros": roswtf 根据你当前目录来确定需要做的检查，这里表示你是在 rosstack 中启动 roswtf。 * "Static checks summary": 这是有关文件系统问题的检查报告，现在的检查结果表示文件系统没问题。 * "Cannot communicate with master, ignoring graph checks（无法与 master 连接，忽略图（graph）检查）": roscore 没有运行，所以 roswtf 没有做运行时检查。

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，所以得先启动 roscore。

现在按照相同的顺序再次运行以下命令：

```
$ roscd

$ roswtf
```

你应该会看到：

```
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
```

既然 roscore 已经运行了所以 roswtf 做了一些运行时检查。检查过程的长短取决于正在运行的 ROS 节点数量，可能会花费很长时间才能完成。正如你看到的，这一次出现了警告：

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

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。

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。

接下来我们在 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
```

这次我们会看到：

```
Stack: ros

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

Static checks summary:


Found 1 error(s).


ERROR Not all paths in ROS_PACKAGE_PATH [bad] point to an existing dir
ectory:

 * bad


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


Cannot communicate with master, ignoring graph checks
```

正如你看到的，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。如果你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。

现在你已经知道如何使用 roswtf 了，接下来可以花点时间通过 wiki 导航了解一下 wiki.ros.org 网站是如何组织的。

18. 探索 ROS 维基

Description: 本教程介绍了 ROS 维基(wiki.ros.org)的组织结构以及使用方法。同时讲解了如何才能从 ROS 维基中找到你需要的信息。

Keywords: wiki

Tutorial Level: BEGINNER

Next Tutorial: 接下来做什么？

目 录

1. 首页
2. package 页面
3. stack 页面

注：显然本教程已经过时（包括英文版的）。

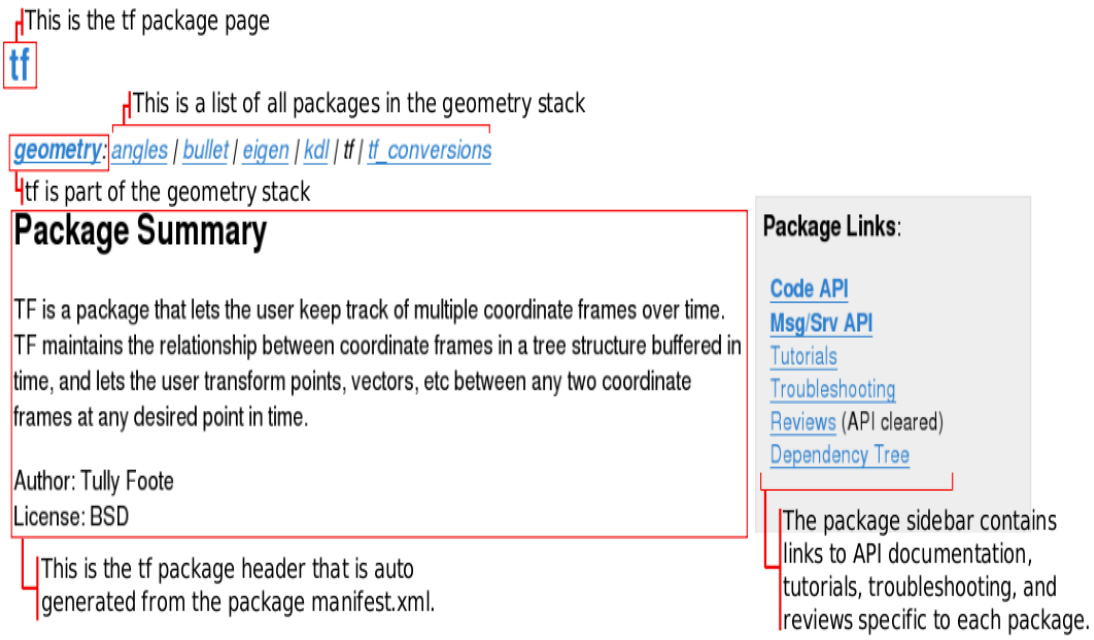
本教程我们会介绍 wiki 上各种不同的标题、链接和侧边栏，从而帮助你理解 wiki.ros.org 网页是如何布局的。

首页

首页就是你在浏览器中打开 wiki.ros.org 后看到的页面。我们来看一下首页顶部的标题栏。

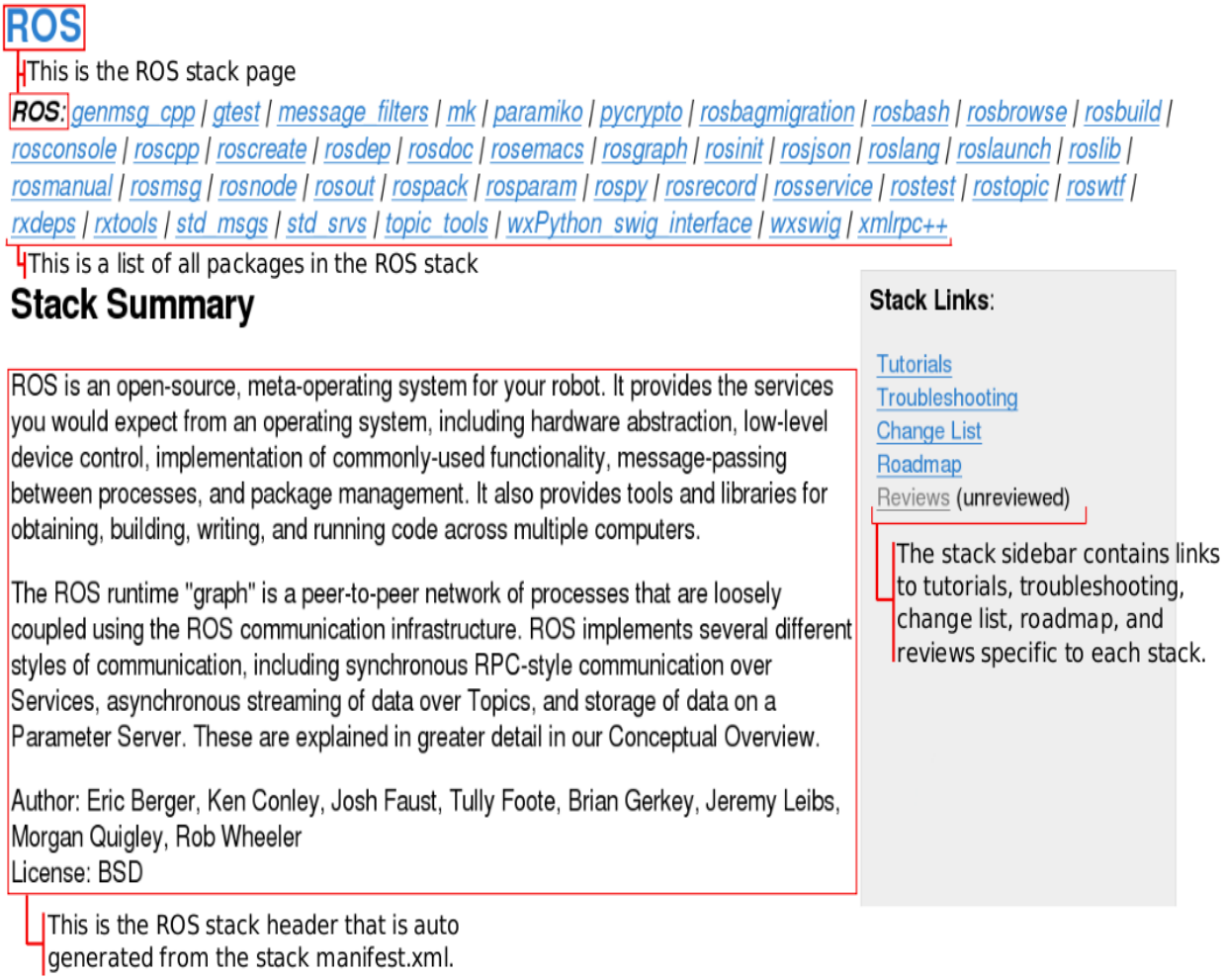
package 页面

我们来看一下 *ros-pkg* 维基页面中的 tf (www.ros.org/wiki/tf)页面。每个 package 的标题栏是根据 package 中的 manifest 文件自动生成的。正如你看到的，每个 package 页面上都包含针对该 package 的教程及故障排除链接。



stack 页面

我们来看一下 *ros* stack 维基页面上的 ROS (www.ros.org/wiki/ROS)页面。stack 标题栏也是根据 stack 中的 manifest 文件自动生成的。



As you can see each stack contains tutorials and troubleshooting specific to the stack.

19. 接下来做什么？

Description: 本教程将讨论获取更多知识的途径，以帮助你更好地使用 ROS 搭建真实或虚拟机器人。

Tutorial Level: BEGINNER

目 录

1. 使用模拟器
2. 使用 RViz
3. 理解 TF
4. 更进一步

此时你应该已经对 ROS 中的一些核心概念有了一定的理解。

给你一台运行 ROS 的机器人，你应该能够运用所学知识来列出机器人上发布和订阅的各种话题（topic），查看话题中发布的消息，然后编写你自己的节点（node）来处理传感器数据，最后让机器人在真实环境中动起来。

ROS 真正的吸引力不在于它自己发布或订阅的中间件，而在于 ROS 为世界各地的开发者提供了一种标准机制来分享他们的代码。ROS 最大的特色在于庞大的社区。

大量现成可用的程序包（packages）已经势不可挡。所以本教程试图让你知道接下来能做什么？

使用模拟器

即使你有一台真实的机器人，先从模拟器开始学习也是一个不错的选择，这样出问题的时候你不用担心机器人会伤害到你自己或者损坏一台昂贵的机器人。

你可以从 PR2 模拟器 或者 Turtlebot 模拟器入手。还可以搜索一台机器人并查看它是否提供了自己的模拟器。

现在你可能会尝试使用'teleop'程序包（package）来控制模拟的机器人（比如：turtlebot_teleop）或者运用你掌握的知识来查看话题(topic)，编写程序发布恰当的消息来驱动你的机器人。

使用 RViz

RViz 是一款强大的可视化工具，它允许你查看机器人中的传感器和内部状态。rviz 用户指南会教你如何使用。

理解 TF

TF 程序包（package）提供在机器人所使用到的各种坐标系之间的变换功能，并保持跟踪这些变换的变化。当你使用任何真实的机器人时理解好 TF 是必不可少的，所以学习 TF 相关的教程是值得的。

如果你正在打造一台你自己的机器人，此刻你可能会考虑给你的机器人设计一个 URDF 模型。如果你正在用的是一台“标配”的机器人，可能已经有人帮你设计好了。不过，你可能依然值得去简单了解一下 URDF 程序包（package）。

更进一步

现在，你可能已经准备好开始让你的机器人去执行更复杂的任务，那么下面的链接可能会对你有帮助：

1. actionlib - actionlib 程序包（package）为可抢占式的任务提供一个标准接口，并广泛应用在 ROS 中的“高层次”程序包（packages）中。
2. navigation - 2D 导航：绘图及路径规划。

3. MoveIt - 用来控制机器人的手臂.

中级教程

20. 手动创建 ROS package

Description: 本教程将展示如何手动创建 ROS package

Tutorial Level: INTERMEDIATE

Next Tutorial: 管理系统依赖项

我们可以使用(catkin_create_pkg)工具来自动创建 ROS package，不过，接下来你就会发现，这不是什么难事。 roscreate-pkg 节省精力避免错误，但 package 只不过就是一个文件夹外加一个 XML 文件。

现在，我们要手动创建一个名为 foobar 的 package。请转至你的 catkin workspace，并确认你已经刷新了 setup 文件。

```
catkin_ws_top $ mkdir -p src/foobar

catkin_ws_top $ cd src/foobar
```

我们要做的第一件事就是添加配置文件。package.xml 文件使得像 rospack 这类工具能够检测出我们所创建的 package 的依赖项.

在 foobar/package.xml 文件里添加如下语句:

```
<package>

  <name>foobar</name>

  <version>1.2.4</version>

  <description>

  This package provides foo capability.

  </description>

  <maintainer email="foobar@foo.bar.willowgarage.com">PR-foobar</main
tainer>

  <license>BSD</license>
```

```
  <buildtool_depend>catkin</buildtool_depend>


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


  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
</package>
```

同样，你可以参考 this page from catkin tutorial 以获得更多关于 catkin/package.xml 的信息。

现在，你的 package 里已经包含配置文件(package.xml)，ROS 能够找到它。试一试，执行如下指令：

```
rospack find foobar
```

如果 ROS 配置正确，你将会看到类似如下的信息： /home/user/ros/catkin_ws_top/src/foobar. 这就是 ROS 如何在后台寻找 package 的机理。

应该注意到我们刚才所创建的 package 依赖于 roscpp 和 std_msgs.而 catkin 恰恰是利用这些依赖项来配置所创建的 package。

至此，我们还需要一个 CMakeLists.txt 文件，这样 catkin_make 才能够利用 CMake 强大的跨平台特性来编译所创建的 package。

在 foobar/CMakeLists.txt 文件里输入：

```
cmake_minimum_required(VERSION 2.8.3)

project(foobar)

find_package(catkin REQUIRED roscpp std_msgs)

catkin_package()
```

这就是你想在 ROS 下用 catkin 编译 package 所要做的全部工作。当然了，如果想要让它正真的编译一些东西，你还需要学习关于 CMake 宏的一些知识。请参考 CMakeLists.txt 以获得更多信息。回顾初级教程如 (CreatingPackage 等) 来修改你的 package.xml 和 CMakeLists.txt 文件

21. 管理系统依赖项

Description: 本教程将展示如何使用 rosdep 安装系统依赖项.

Tutorial Level: INTERMEDIATE

Next Tutorial: Roslaunch 在大型项目中的使用技巧

目 录

1. System Dependencies
1. rosdep

electricfuertegroovyhydroindigojade

System Dependencies

ROS packages 有时会需要操作系统提供一些外部函数库，这些函数库就是所谓的“系统依赖项”。在一些情况下，这些依赖项并没有被系统默认安装，因此，ROS 提供了一个工具 rosdep 来下载并安装所需系统依赖项。

ROS packages 必须在配置文件中声明他们需要哪些系统依赖项。下面我们来看看 turtlesim package 的配置文件：

```
$ roscd turtlesim
```

然后,

rosdep

rosdep 是一个能够下载并安装 ROS packages 所需要的系统依赖项的小工具 使用方法:

```
rosdep install [package]
```

为 turtlesim 下载并安装系统依赖项:

```
$ rosdep install turtlesim
```

如果是按照教程的顺序学习，那么这很可能是你第一次使用 rosdep 工具，因此，当执行这条指令，你会看到如下的报错消息：

```
• ERROR: your rosdep installation has not been initialized yet.  P
  lease run:
•
```

```
•     sudo rosdep init
•     rosdep update
```

按照提示执行上边两条指令，并再次安装 turtlesim 的系统依赖项。

如果你安装的是二进制文件，那么会看到如下信息：

```
•  All required rosdeps installed successfully
```

不然，你将看到如下的 turtlesim 依赖项安装信息：

```
•  set -o errexit
•  set -o verbose
•
•
•  if [ ! -f /opt/ros/lib/libboost_date_time-gcc42-mt*-1_37.a ] ; t
   hen
•    mkdir -p ~/ros/ros-deps
•    cd ~/ros/ros-deps
•    wget --tries=10 http://pr.willowgarage.com/downloads/boost_1_
   37_0.tar.gz
•    tar xzf boost_1_37_0.tar.gz
•    cd boost_1_37_0
•    ./configure --prefix=/opt/ros
•    make
•    sudo make install
•  fi
•
•
•  if [ ! -f /opt/ros/lib/liblog4cxx.so.10 ] ; then
•    mkdir -p ~/ros/ros-deps
•    cd ~/ros/ros-deps
•    wget --tries=10 http://pr.willowgarage.com/downloads/apache-l
   og4cxx-0.10.0-wg_patched.tar.gz
```

```
•    tar xzf apache-log4cxx-0.10.0-wg_patched.tar.gz
•    cd apache-log4cxx-0.10.0
•    ./configure --prefix=/opt/ros
•    make
•    sudo make install
•  fi
```

rosdep 运行上述 bash 脚本，并在完成后退出。

22. Roslaunch 在大型项目中的使用技巧

Description: 本教程主要介绍 roslaunch 在大型项目中的使用技巧。重点关注如何构建 launch 文件使得它能够在不同的情况下重复利用。我们将使用 2dnav_pr2 package 作为学习案例。

Tutorial Level: INTERMEDIATE

Next Tutorial: Roslaunch Nodes in Valgrind or GDB

目 录

1. Introduction
2. 高层级的结构
3. Machine tags and Environment Variables
4. Parameters, namespaces, and yaml files
5. launch 文件的重用
6. 参数重载
7. Roslaunch arguments

Introduction

机器人上的应用经常涉及到 nodes 间的交互，每个 node 都有很多的 parameters。在二维平面上的导航就是一个很好的例子。2dnav_pr2 包含了基本的运动节点、定位、地面识别、底层控制器和地图服务器。 同时，还有几百个 ROS parameters 影响着这些 node 的行为模式。此外，也还存在着一些额外的约束。例如为了提高效率，地面识别节点应该跟倾斜的激光节点运行在同一台机器上。

一个 roslaunch 文件能够让你一次性将这些都配置好。在一个机器人上，roslaunch 一下 *2dnav_pr2* package 里的 *2dnav_pr2.launch* 文件就可以启动机器人导航时所需的所有东西。在本教程中，我们将仔细研究 launch 文件和它所具有的功能。

我们希望 launch 文件能够尽可能的重用，这样在不同的机器人平台上就不需要修改这些 launch 文件就能使用。即使是从真实的环境转到模拟环境中也只要稍微修改即可。接下来，我们将要研究如何构建 launch 文件才能实现其最大化的重用。

高层级的结构

这是一个高层级的 launch 文件 (利用指令 "rospack find 2dnav_pr2/move_base/2dnav_pr2.launch"可以找到).

```
<launch>

  <group name="wg">

    <include file="$(find pr2_alpha)/$(env ROBOT).machine" />

    <include file="$(find 2dnav_pr2)/config/new_amcl_node.xml" />

    <include file="$(find 2dnav_pr2)/config/base_odom_teleop.xml" />

    <include file="$(find 2dnav_pr2)/config/lasers_and_filters.xml" />

    <include file="$(find 2dnav_pr2)/config/map_server.xml" />

    <include file="$(find 2dnav_pr2)/config/ground_plane.xml" />


    <!-- The navigation stack and associated parameters -->

    <include file="$(find 2dnav_pr2)/move_base/move_base.xml" />

  </group>

</launch>
```

这个文件引用了其他的文件。在这些被引用的文件中都包含有与系统有关的 node 和 parameter（甚至是嵌套引用），比如定位、传感器处理和路径规划。

编写技巧: 高层级的 launch 文件应该简短，利用 include 指令将系统的组成部分和 ROS parameter 引用过来即可。

接下来我们将会看到，这种技巧使得我们可以很容易的替换掉系统的某个部分。

想要在 PR2 运行这个应用，我们需要启动 core，接着 roslaunch 一个具体机器人的 launch 文件，例如在 *pr2_alpha* package 里的 *pre.launch* 文件，最后 roslaunch 一下 *2dnav_pr2.launch*。与其分开 roslaunch 这么多的文件，我们可以一次性将它们 roslaunch 起来。这会有一些利弊权衡：

- 优点: 我们可以少做几个 "打开新终端, roslaunch" 的步骤。
- 缺陷 1: roslaunch 一个 launch 文件会有一个持续一段时间的校准过程。如果 2dnav_pr2 launch 文件引用了机器人的 launch 文件，当我们用 control-c 终止这个 roslaunch 进程，然后又再次开启这个进程，校准过程还得再来一遍。

- 缺陷 2: 一些导航 node 要求校准过程必须在它启动之前完成。roslaunch 目前还没有对节点的启动时间和顺序进行控制。最完美的方案当然是让导航节点等到校准过程完成后再启动，但就目前的情况来看，把他们分别放在两个 launch 文件里，直到校准过程结束再启动导航节点是个可行的方案。

因此，对于是否应该把多个启动项放到一个 launch 文件里并没有一个统一的标准。就本教程的案例而言，我们把他们放到两个 launch 文件里。

编写技巧:在决定应用需要多少个高层级的 launch 文件时，你要考虑利弊的权衡。

Machine tags and Environment Variables

为了平衡负载和管理带宽，我们需要控制哪些节点在哪个机器上运行。比如，我们希望 amcl 跟 base laser 在同一台机器上运行。同时，考虑到重用性，我们不希望把具体的机器名写入 launch 文件。roslaunch 使用 **machine tags** 来解决这个问题。

第一个引用如下：

```
<include file="$(find pr2_alpha)/$(env ROBOT).machine" />
```

首先应该注意到这个文件使用 *env* 置换符来使用 ROBOT 变量的值。例如，在 roslaunch 指令前执行：

```
export ROBOT=pre
```

将会使得 *pre.machine* 被引用。

编写技巧: 使用 env 置换符可以使得 launch 文件的一部分依赖于环境变量的值。

接下来，我们来看看 *pr2_alpha* package 里的 *pre.machine* 文件。

```
<launch>

  <machine name="c1" address="pre1" ros-root="$(env ROS_ROOT)" ros-pac
kage-path="$(env ROS_PACKAGE_PATH)" default="true" />

  <machine name="c2" address="pre2" ros-root="$(env ROS_ROOT)" ros-pac
kage-path="$(env ROS_PACKAGE_PATH)" />

</launch>
```

这个文件对本地机器名进行了一个映射，例如"c1" 和 "c2"分别对应于机器名"pre1"和"pre2"。甚至可以控制你登录的用户名。（前提是你有 ssh 证书）

一旦这个映射建立好了之后，就可以用于控制节点的启动。比如，*2dnav_pr2* package 里所引用的 *config/new_amcl_node.xml* 文件包含这样的语句：

```
<node pkg="amcl" type="amcl" name="amcl" machine="c1">
```

这可以控制 *amcl* 节点在机器名为 *c1* 的机器上运行。（查看其他的 launch 文件，你可以看到大多数激光传感器处理节点都在这个机器上运行）

当我们要在另一个机器人上运行程序，比如说机器人 *prf*，我们只要修改 ROBOT 环境变量的值就可以了。相应的机器配置文件（*pr2_alpha* package 里的 *prf.machine* 文件）就会被加载。我们甚至可以通过设置 ROBOT 为 *sim* 从而是得程序可以在一个模拟机器人上运行。查看 *pr2_alpha* package 里的 *sim.machine* 文件，它只是将所有的机器名映射到了本地主机名

编写技巧：使用 *machine tags* 来平衡负载并控制节点在机器上的启动，同时也因考虑将机器配置文件（.machine)跟系统变量关联起来以便于重复利用。

Parameters, namespaces, and yaml files

我们来看一下被引用的 *move_base.xml* 文件。文件的一部分如下：

```
<node pkg="move_base" type="move_base" name="move_base" machine="c2">

  <remap from="odom" to="pr2_base_odometry/odom" />

  <param name="controller_frequency" value="10.0" />

  <param name="footprint_padding" value="0.015" />

  <param name="controller_patience" value="15.0" />

  <param name="clearing_radius" value="0.59" />

  <rosparam file="$(find 2dnav_pr2)/config/costmap_common_params.yaml
" command="load" ns="global_costmap" />

  <rosparam file="$(find 2dnav_pr2)/config/costmap_common_params.yaml
" command="load" ns="local_costmap" />

  <rosparam file="$(find 2dnav_pr2)/move_base/local_costmap_params.ya
ml" command="load" />

  <rosparam file="$(find 2dnav_pr2)/move_base/global_costmap_params.y
aml" command="load" />

  <rosparam file="$(find 2dnav_pr2)/move_base/navfn_params.yaml" comm
and="load" />

  <rosparam file="$(find 2dnav_pr2)/move_base/base_local_planner_para
ms.yaml" command="load" />

</node>
```

这一小段代码负责启动 *move_base* 节点。 第一个引用元素是 **remapping**. 设计 Move_base 时是希望它从 "odom" topic 接收里程计信息的。在这个 pr2 案例里，里程计信息是发布在 *pr2_base_odometry* topic 上,所以我们要重新映射一下。

编写技巧: 当一个给定类型的信息在不同的情况下发布在不同的 topic 上，我们可以使用 topic remapping

这个文件有好几个<param>标签。这些参数是节点的内部元素（因为它们都写在</node>之前），因此它们是节点的私有参数私有参数。比如，第一个参数将 *move_base/controller_frequency* 设置为 10.0。

在<param>元素之后,还有一些<rosparam>元素，它们将从 yaml 文件中读取参数。yaml 是一种易于人类读取的文件格式，支持复杂数据的结构。这是第一个<rosparam>所加载的 *costmap_common_params.yaml* 文件一部分:

```
raytrace_range: 3.0

footprint: [[-0.325, -0.325], [-0.325, 0.325], [0.325, 0.325], [0.46, 0.0], [0.325, -0.325]]

inflation_radius: 0.55


# BEGIN VOXEL STUFF

observation_sources: base_scan_marking base_scan tilt_scan ground_object_cloud


base_scan_marking: {sensor_frame: base_laser, topic: /base_scan_marking, data_type: PointCloud, expected_update_rate: 0.2,

  observation_persistence: 0.0, marking: true, clearing: false, min_obstacle_height: 0.08, max_obstacle_height: 2.0}
```

我们看到 yaml 支持向量等数据结构（如上边的 *footprint* 就是向量）。它同样支持将嵌套的域名空间，比如 *base_laser* 被归属到了 *base_scan_marking/sensor_frame* 这样的嵌套域名下。注意这些域名都是归属于 yaml 文件自身域名 *global_costmap* 之下，而 yaml 文件的域名是由 *ns* 变量来控制的。 同样的，由于 *rosparam* 都被包含在节点里，所以参数的完整名称就是*/move_base/global_costmap/base_scan_marking/sensor_frame*.

接着一行是:

```
<rosparam file="$(find 2dnav_pr2)/config/costmap_common_params.yaml" command="load" ns="local_costmap" />
```

这跟上一行引用的是完全一样的 yaml 文件，只不过他们的域名空间不一样（local_costmap 域名只影响运动路径控制器，而 global_costmap 影响到全局的导航规划）。这样可以避免重新给同样的变量再次赋值。

再下一行是:

```
<rosparam file="$(find 2dnav_pr2)/move_base/local_costmap_params.yaml
" command="load"/>
```

跟上一行不同，这一行没有 *ns* 属性。因此这个 yaml 文件的域名就是*/move_base*。但是再仔细查看一下这个 yaml 文件的前几行：

```
local_costmap:

  #Independent settings for the local costmap

  publish_voxel_map: true

  global_frame: odom_combined

  robot_base_frame: base_link
```

最终我们可以确定参数都是归属于*/move_base/local_costmap* 域名之下。

编写技巧: Yaml 文件允许复杂的嵌套域名的参数，相同的参数值可以在多个地方重复使用。

launch 文件的重用

上述的编写技巧都是为了使得 launch 文件能再不同的环境下更易于重用。从上边的一个例子我们已经知道，使用 *env* 子变量可以在不改动 launch 文件的情况下就改变其行为模式。但是仍然在某些情况下，重用 launch 文件还是很麻烦甚至不肯能。我们来看一下 pr2_2dnav_gazebo package。它有 2d 导航功能，但是只是为 Gazebo 模拟器而设计的。对于导航来说，唯一改变了的就是我们所使用的 Gazebo 环境是一张静态地图，因此 map_server 节点必须重载其参数。当然这里我们可以使用另外一个 env 变量，但这会使得用户还得设置一大堆变量才能够 roslaunch。因此，2dnav gazebo 有它自己的高层级 launch 文件，叫'2dnav-stack-amcl.launch'，如下所示：

```
<launch>

  <include file="$(find pr2_alpha)/sim.machine" />

  <include file="$(find 2dnav_pr2)/config/new_amcl_node.xml" />

  <include file="$(find 2dnav_pr2)/config/base_odom_teleop.xml" />

  <include file="$(find 2dnav_pr2)/config/lasers_and_filters.xml" />

  <node name="map_server" pkg="map_server" type="map_server" args="
$(find gazebo_worlds)/Media/materials/textures/map3.png 0.1" respawn=
"true" machine="c1" />

  <include file="$(find 2dnav_pr2)/config/ground_plane.xml" />

  <!-- The naviagtion stack and associated parameters -->

  <include file="$(find 2dnav_pr2)/move_base/move_base.xml" />
```

```
</launch>
```

首先，因为我们知道这是一个模拟器，所以直接使用 sim.machine 文件，而不必再使用$(env ROBOT)变量来选择。其次，原来的

```
<include file="$(find 2dnav_pr2)/config/map_server.xml" />
```

已经被

```
<node name="map_server" pkg="map_server" type="map_server" args="$(find gazebo_worlds)/Media/materials/textures/map3.png 0.1" respawn="true" machine="c1" />
```

所替换。两者都包含了节点的声明，但他们来自不同的文件。

编写技巧：想要改变应用的高层级功能，只要修改 launch 文件的相应部分即可。

参数重载

在某些情况下，上述技巧会很不方便。比如，使用 *2dnav_pr2* 但希望修改 local costmap 的分辨率为 0.5。我们只需要修改 *local_costmap_params.yaml* 文件即可。我们本来只是想临时的修改，但这种方法却意味着它被永久修改了。也许我们可以将 *local_costmap_params.yaml* 文件拷贝一份并做修改，但这样我们还需要修改 *move_base.xml* 文件去引用修改后的 yaml 文件。 接着我们还得修改 *2dnav_pr2.launch* 文件去引用被修改后的 xml 文件。这是很花时间的工作，而且假如我们使用了版本控制，我却看不到版本间有任何的改变。另外一种方法是新建立一个 launch 文件，这样就可以在 *2dnav_pr2.launch* 文件中定义 *move_base/local_costmap/resolution* 参数，修改这个参数就可以满足我们都要求。如果我们能够提前知道哪些参数有可能被修改，这会是一个很好的办法。

更好的方法是利用 roslaunch 的重载特性：参数按顺序赋值(在引用被执行之后)。这样，我们可以构建另外一个可以重载参数的高层级文件：

```
<launch>

<include file="$(find 2dnav_pr2)/move_base/2dnav_pr2.launch" />

<param name="move_base/local_costmap/resolution" value="0.5"/>

</launch>
```

这个方法的主要缺点在于它使得文件难以理解：想要知道一个参数的值需要追溯 launch 的引用。但它确实避免了多次拷贝文件然后修改它们。

编写技巧：利用 roslaunch 重载功能来修改一个深深嵌套在 launch 文件分支中的参数。

Roslaunch arguments

在 CTurtle 版本中，roslaunch 还有一个和标签(tags)类似的参数替换特性，它允许依据变量的值来有条件启动某个 launch 文件。这比上述的参数重载和 launch 文件重用来得更简洁，适用性更强。但它需要一些额外操作来完成这个功能：修改原始 launch 文件来指定哪些变量是可变的。请参考 roslaunch XML documentation.

编写技巧：在能够修改原始 launch 文件的情况下，优先选择使用 roslaunch 变量，而不是参数重载和拷贝 launch 文件的方法。

How to Roslaunch Nodes in Valgrind or GDB

Description: When debugging roscpp nodes that you are launching with roslaunch, you may wish to launch the node in a debugging program like gdb or valgrind instead. Doing this is very simple.

Keywords: roslaunch, valgrind, gdb, pdb

Tutorial Level: INTERMEDIATE

The `launch-prefix` attribute of the `<node>` tag that, among other things, makes it easy to debug a ROS node process. Here are some example `launch-prefix`es you might find useful:

- `launch-prefix="xterm -e gdb --args"` : run your node in a gdb in a separate xterm window, manually type `run` to start it
- `launch-prefix="gdb -ex run --args"` : run your node in gdb in the same xterm as your launch without having to type `run` to start it
- `launch-prefix="valgrind"` : run your node in valgrind
- `launch-prefix="xterm -e"` : run your node in a separate xterm window
- `launch-prefix="nice"` : nice your process to lower its CPU usage
- `launch-prefix="screen -d -m gdb --args"` : useful if the node is being run on another machine; you can then ssh to that machine and do `screen -D -R` to see the gdb session
- `launch-prefix="xterm -e python -m pdb"` : run your python node a separate xterm window in pdb for debugging; manually type `run` to start it
- `launch-prefix="/path/to/run_tmux"`: run your node in a new tmux window; you'll need to create `/path/to/run_tmux`with the contents:

```
#!/bin/sh
```

```
tmux new-window "gdb --args $*"
```

Obtaining core dumps

To obtain core dumps when processes crash, first set the core file size limit. To check the limits, run:

```
$ ulimit -a

core file size          (blocks, -c) 0          #  <-- Prevents core dump
s

data seg size           (kbytes, -d) unlimited

scheduling priority             (-e) 20

file size               (blocks, -f) unlimited

pending signals                 (-i) 16382

max locked memory       (kbytes, -l) 64

max memory size         (kbytes, -m) unlimited

open files                      (-n) 1024

pipe size            (512 bytes, -p) 8

POSIX message queues     (bytes, -q) 819200

real-time priority              (-r) 0

stack size              (kbytes, -s) 8192

cpu time               (seconds, -t) unlimited

max user processes              (-u) unlimited

virtual memory          (kbytes, -v) unlimited

file locks                      (-x) unlimited
```

Set the core size to unlimited:

```
ulimit -c unlimited
```

Now crashing processes will attempt to create core files. Currently (2010-07-02) they will fail to create the file because the default roslaunch working directory, $ROS_HOME, contains a directory named "core". This directory prevents the core dump from being created.

To allow core dumps to be created, set the core filename to use the process pid by default. Run the following as root:

```
echo 1 > /proc/sys/kernel/core_uses_pid
```

Now core dumps will show up as $ROS_HOME/core.PID

23. ROS 在多机器人上的使用

Description: 本教程将展示如何在两台机器上使用 ROS 系统，详述了使用 ROS_MASTER_URI 来配置多台机器使用同一个 master。

Tutorial Level: INTERMEDIATE

Next Tutorial: 自定义消息类型

目录

1. 概述
2. 跨机器运行的 Talker / listener
1. 启动[[master]]
2. 启动 listener
3. 启动 talker
4. 反向测试
3. 运行出错
4. 译者注

概述

ROS 设计的灵魂就在于其分布式计算。一个 well-written 的节点不需要考虑在哪台机器上运行，它允许实时分配计算量以最大化的利用系统资源。(有一个特例——驱动节点必须运行在跟硬件设备有物理连接的机器上）。在多个机器人上使用 ROS 是很简单的一件事，你只需要记住一下几点：

- 你只需要一个 master，只要在一个机器上运行它就可以了。
- 所有节点都必须通过配置 ROS_MASTER_URI 连接到同一个 master。
- 任意两台机器间任意两端口都必须要有完整的、双向连接的网络。(参考 ROS/NetworkSetup).
- 每台机器都必须向其他机器广播其能够解析的名字。(参考 ROS/NetworkSetup).

跨机器运行的 Talker / listener

假如说我们希望在两台机器上分别运行 talker / listener， 主机名分别为 **marvin** 和 **hal**.登陆主机名为 **marvin** 的机器,你只要:

```
ssh marvin
```

同样的方法可以登陆 **hal**.

启动[[master]]

我们需要选择一台机器运行 master，这里我们选 **hal**. 启动 master 的第一步是:

```
ssh hal

roscore
```

启动 listener

接下来我们在机器 **hal** 上启动 listener, 并配置 ROS_MASTER_URI，这样就可以使用刚刚启动的 master 了:

```
ssh hal

export ROS_MASTER_URI=http://hal:11311

rosrun rospy_tutorials listener.py
```

启动 talker

现在我们要在 **marvin** 机器上启动 talker，同样通过配置 ROS_MASTER_URI 来使用 **hal** 机器上的 master:

```
ssh marvin

export ROS_MASTER_URI=http://hal:11311

rosrun rospy_tutorials talker.py
```

小惊喜: 现在你可以看到机器 **hal** 上的 listener 正在接收来自 **marvin** 机器上 talker 发布的消息。

请注意，talker / listener 启动的顺序是没有要求的， 唯一的要求就是 master 必须先于节点启动。

反向测试

现在我们来尝试一下反向测试。终止 talker 和 listener 的运行，但仍然保留 master 在机器 **hal** 上，然后让 talker 和 listerner 交换机器运行。

首先，在机器 **marvin** 启动 listerner:

```
ssh marvin

export ROS_MASTER_URI=http://hal:11311

rosrun rospy_tutorials listener.py
```

然后在机器 **hal** 上启动 talker:

```
ssh hal

export ROS_MASTER_URI=http://hal:11311

rosrun rospy_tutorials talker.py
```

运行出错

如果没有取得如上预期的效果，那么很有可能是你的网络配置出错了。参考 ROS/NetworkSetup 重新配置你的网络。

译者注

根据译者的尝试，如果你想取得如上预期效果，你还需配置 ROS_IP 为当前的局域网 ip 地址。(利用 ifconfig 指令可以查看你当前的 ip 地址）。其次，很有可能你的主机名不能够被其他机器解析，所以保险的方法是利用 ssh hostname@local_ip 的方式进行登陆(如 *ssh turtlebot@192.168.1.100*)。再者，ROS_MASTER_URI 最好也用运行 master 的那台机器的 ip 地址来替换主机名（如：*export ROS_MASTER_URI=http://192.168.1.100:11311*)

24. 自定义消息

Description: 本教程将展示如何使用 ROS Message Description Language 来定义你自己的消息类型.

Tutorial Level: INTERMEDIATE

Next Tutorial: 如何写一篇教程

目录

1. 自定义消息
2. 引用和输出消息类型
1. C++
2. Python
3. 依赖项

catkinrosbuild

自定义消息

自定义一个消息类型很简单，只要将.msg 文件放到一个 package 的 msg 文件夹下即可。请参考创建.msg 文件 (不要忘记选择相应的编译构建系统)。

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，例如：

C++

切换行号显示

```
1 #include <std_msgs/String.h>
2
3 std_msgs::String msg;
```

Python

切换行号显示

```
1 from std_msgs.msg import String
2
3 msg = String()
```

依赖项

如果你要使用在其他 package 里定义的消息类型，不要忘记添加以下语句：

```
<build_depend>name_of_package_containing_custom_msg</build_depend>

<run_depend>name_of_package_containing_custom_msg</run_depend>
```

到 package.xml。

教程 ROSNodeTutorialPython 展示了使用自定义消息类型来创建 talker 和 listener 的 C++ 和 Python 实现。

25. 如何写一篇教程

Description: （概述：）本教程介绍了在编辑 ros.org 维基时可以用到的模板和宏定义，并附有示例以供参考。

Keywords: teaching, tutorials, writing，教程，编辑，维基，学习

Tutorial Level: BEGINNER

目 录

1. 如何开始
2. 创建一个教程
3. 填写模板头信息
1. 提醒
2. 标题
3. 概述
4. 后续
5. 等级
6. 关键字
4. 实用宏命令
1. 代码显示框
2. CodeRef
3. FullSearchWithDescriptionsCS
4. IncludeCSTemplate
5. 嵌入视频
5. 示例教程

如何开始

在写教程之前，先浏览一下您的教程所在页面。如果这个页面没有被占用，那么此页面就会显示为“不存在的维基”页面（non-existent /Tutorials wiki page）。

一旦您打开这个“不存在的维基”页面，您将会被重定向到如下界面：

This page does not exist yet. What type of page are you trying to create?

Please see EditingTheWiki for guidelines on how our wiki is organized and tips on creating new pages.

ROS package or Stack

If you're creating the initial page for a stack or package, please give it the page the same name as the stack or package itself, and use one of the following templates:

- StackTemplate - Template this new page to hold usage documentation for a stack
- PackageTemplate - Template this new page to hold usage documentation for a package

If you're creating additional documentation for a package or stack, feel free to structure them however you like, but please keep them within the namespace of your package (e.g. "ros.org/wiki/my_package/more_details") Create new empty page

Tutorials

To keep tutorials organized, each stack and package has a link to tutorials at the bottom of the page. If you got here via that link please start a list of tutorials with this template:

- TutorialIndexTemplate

Once you save that page, it will have a "Create Tutorial" button that you can use to create new tutorial pages which will be automatically indexed and easier for users to find.

单击“TutorialIndexTemplate”链接并保存默认的预览模板，您就可以创建一个如下的新页面：

Create a new tutorial:

Enter tutorial name

Existing Tutorials

No results found.

这个页面对创建您的教程非常有用，同时也会显示一个当前教程的列表。 在栈或包的介绍页面的边栏中会创建一个如下的教程链接：

Package Links:

Code API
Tutorials
Troubleshooting
Reviews (unreviewed)
Dependency Tree

Stack Links:

Tutorials
Troubleshooting
Change List
Roadmap
Reviews (unreviewed)

创建一个教程

在新建教程框（Create a new tutorial）中输入您教程的名字并单击“Enter tutorial name”按钮创建一个基于 TutorialTemplate 的页面（一些关键字前会显示！符号，使用时请去掉它们）：

切换行号显示

```
#####################################
##FILL ME IN
#####################################
## for a custom note with links:
## !note =
## for the canned note of "This tutorial assumes that you have complet
ed the
## previous tutorials:" just add the links
## !note.0.link=
## descriptive title for the tutorial
## !title =
## multi-line description to be displayed in search
## !description =
## the next tutorial description (optional)
## !next =
## links to next tutorial (optional)
## !next.0.link=
## !next.1.link=
## what level user is this tutorial for
## !level= (BeginnerCategory, IntermediateCategory, AdvancedCategory)
## !keywords =
#####################################

!<<IncludeCSTemplate(TutorialCSHeaderTemplate)>>
```

请注意保存您的页面，因为预览按钮（preview）有事不能正常地工作。

填写模板头信息

上面展示的模板是系统自动生成的文档头信息。效果与本教程的开头类似。现在我们来一起看看这些模板是如何工作的。

提醒

“提示”（note）标签可以生成类似于页面顶部的蓝色提醒框，以用来提示用户如何遵循您的教程。

向其中填写的内容将会生成如下效果：

```
note= 狂拽炫酷吊炸天。
```

Note: 狂拽炫酷吊炸天。

又如，您可以使用下列关键字：

```
note.0=[[ROS/Tutorials|ROS 教程]]

note.1=[[Documentation| ros.org]]
```

这样就可以为您的教程添加一个标准的前导教程提示：*This tutorial assumes that you have completed the previous tutorials:*（本教程假设您已经完成且掌握了下列教程的内容：）

Note: This tutorial assumes that you have completed the previous tutorials: ROS 教程, ros.org.

请注意，如果您想在您的自定义提示中加入链接，您可以直接在 note 标签中包含这个链接而不是单独使用 note.0, note.1 等标签单独标记链接。如果您错误的使用了 note.0, note.1 等标签以下提示内容可能会错误的出现在您的链接前："This tutorial assumes that you have completed the previous tutorials:"

标题

“标题”(title)标签可以创建页面开头的标题，同时它也标记了您教程的标题。教程标题将出现在搜索结果中。

```
!title= 风骚但不失矜持的名字很重要
```

概述

“概述”（description）标签会在您的页面顶端显示您的教程概述，此概述也会出现在搜索结果中。

```
description= 我的葵花宝典狂拽炫酷吊炸天，还不快来试试！
```

效果如下：

Description: 我的葵花宝典狂拽炫酷吊炸天，还不快来试试！

后续

“后续”（next）标签用于显示完成您当前教程之后可以继续学习的其他教程。这个标签是可选的，如果空下将不会显示任何内容。

```
next=练完葵花宝典的亲们还可以看看

next.0.link=[[ROS/Tutorials|ROS 教程]]

next.1.link=[[actionlib_tutorials/Tutorials|actionlib 教程]]
```

效果如下：

Next Tutorial: 练完葵花宝典的亲们还可以看看 ROS 教程 actionlib 教程.

您也可以这样使用：

```
next.0.link=[[actionlib_tutorials/Tutorials|Actionlib 教程]]
```

效果如下：

Next Tutorial: Actionlib 教程.

等级

“等级”（level）标签有双重意义,它不但告诉用户您的教程需要他们达到什么水平，而且也是教程搜索引擎的分类标准之一。BeginnerCategory：初级 IntermediateCategory：中级 AdvancedCategory：高级

```
level=AdvancedCategory
```

效果如下：

Tutorial Level: ADVANCED

关键字

“关键字”（keywords）标签可以帮助用户通过关键字找到您的教程。

```
keywords= 狂拽，炫酷，吊炸天
```

效果如下：

Keywords: 狂拽，炫酷，吊炸天

实用宏命令

ROS 维基提供了许多有用的宏命令以帮助您更好的完成您的教程。

代码显示框

目前 ROS 维基支持以下语言的代码提示框：

c++

- 示例:
  - {{{

```
o  #!cplusplus
o  #include <ros/ros.h>
o  int main(int argc, char** argv)
o  {
o    ros::init(argc, argv, "talker");
o    ros::NodeHandle n;
o  ...
o  }}}
```

显示效果:

```
o  切换行号显示
o    1 #include <ros/ros.h>
o    2 int main(int argc, char** argv)
o    3 {
o    4   ros::init(argc, argv, "talker");
o    5   ros::NodeHandle n;
o    6 ...
```

latex

- 示例:

```
o  {{{
o  #!latex
o  $$\overline{\overline{J}} \dot{\overline{v}} = -\overline{k}$$
o  }}}
```

显示效果:

o $\overline{\overline{J}}\dot{\overline{v}} = -\overline{k}$

python

- 示例:

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

```
import roslib; roslib.load_manifest('beginner_tutorials')
import rospy
from std_msgs.msg import String
def talker():
    pub = rospy.Publisher('chatter', String)
    rospy.init_node('talker', anonymous=True)
...
}}}
```

显示效果:

切换行号显示

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')
import rospy
from std_msgs.msg import String
def talker():
    pub = rospy.Publisher('chatter', String)
    rospy.init_node('talker', anonymous=True)
...
```

CodeRef

在您编写教程的过程中，您有时可能会引用到您之前所写的代码块。这时，您可以使用“CodeRef”引用它们。

```
{{{
#!cplusplus block=blockname
葵花宝典....
不但 高端大气上档次
而且 狂拽炫酷吊炸天
}}}
```

你可以使用“CodeRef”宏来引用上面代码块的特定行：

```
<<CodeRef(blockname,1,2)>>
```

效果如下：

切换行号显示

```
1 葵花宝典....
2 不但 高端大气上档次
```

FullSearchWithDescriptionsCS

“FullSearchWithDescriptionsCS”宏可以根据关键字自动生成教程索引和搜索结果。

```
<<FullSearchWithDescriptionsCS(title:WritingTutorials BeginnerCategory)>>
```

这条代码用来显示标题为“WritingTutorials”且分类为“BeginnerCategory”的教程的标题及概述。如果将分类改为“AdvancedCategory”，将没有任何显示结果。

1. How to Write a Tutorial

   This tutorial covers useful template and macros for writing tutorials, along with example tutorials that are available for guidance onros.org

2. 如何写一篇教程

   （概述：）本教程介绍了在编辑 ros.org 维基时可以用到的模板和宏定义，并附有示例以供参考。

3. チュートリアルの書き方

   このチュートリアルは ros.org に関するガイダンスのために利用できるサンプルチュートリアルにそってチュートリアルを記述するための便利なテンプレートとマクロを押さえます。

IncludeCSTemplate

这个宏将页面顶端的关键字转换为 clearsilver 变量，这些变量可以被 clearsilver 模板命令使用。本教程中的关键字被用在 TutorialCSHeaderTemplate 中

```
!<<IncludeCSTemplate(TutorialCSHeaderTemplate)>>
```

嵌入视频

这个宏可以在本维基中添加视频。将您的视频文献上传至 Youtube 夹并使用这个宏命令指向您的视频路径就可在您的教程中添加视频。

注意: MediaServer 宏已停止使用，所以你必须重新上传你的视频。

这个例子演示了一个在 wge100_camera 文件夹中的视频。

```
<<Youtube(Q5KC-trrw_o)>>
```

示例教程

以下列出了一些优秀的教程共您参考（英文）：

- ROS Tutorials
- Driver Tutorials
- actionlib tutorials

  共同建设 ROS 中文社区，请联系我。ZhiweiChu