cnROSTutorials

非初学者: 如果你已经很熟悉 ROS fuerte(要塞龟)或更早期版本的使用，而只是想使用在groovy(加岛象龟)和hydro(渔龟)中开始采用的最新编译构建系统catkin，那你可以深入学习catkin教程。 但是我们仍然建议所有人学完整个“初级”教程以方便理解新增的功能特性。

如果你是**Linux**初学者：也许先学习一些有关linux命令行工具的快速使用教程会对你很有帮助，这里有篇好教程（英文）。

核心ROS教程

初级

1. 安装并配置ROS环境

   本教程详细描述了ROS的安装与环境配置。
2. ROS文件系统介绍

   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。
3. 创建ROS程序包

   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。
4. 编译ROS程序包

   本教程介绍ROS程序包的编译方法
5. 理解 ROS节点

   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。
6. 理解ROS话题

   本教程介绍ROS话题（topics）以及如何使用rostopic 和 rxplot 命令行工具。
7. 理解ROS服务和参数

   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。
8. 使用 rqt_console 和 roslaunch

   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。
9. 使用rosed编辑ROS中的文件

   本教程将展示如何使用rosed来简化编辑过程。

10. 创建ROS消息和ROS服务
    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。
11. 编写简单的消息发布器和订阅器 (C++)
    本教程将介绍如何编写C++的发布器节点和订阅器节点。
12. 写一个简单的消息发布器和订阅器 (Python)
    本教程将通过Python编写一个发布器节点和订阅器节点。
13. 测试消息发布器和订阅器
    本教程将测试上一教程所写的消息发布器和订阅器。
14. 编写简单的Service和Client (C++)
    本教程介绍如何用C++编写Service和Client节点。
15. 编写简单的Service和Client (Python)
    本教程介绍如何用Python编写Service和Client节点。
16. 测试简单的Service和Client
    本教程将测试之前所写的Service和Client。
17. 录制与回放数据
    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
18. roswtf入门
    本教程介绍了roswtf工具的基本使用方法。
19. 探索ROS维基
    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
20. 接下来做什么？
    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。

现在你已经完成了初级教程的学习，请回答这个 问卷来检验一下自己的学习效果。

中级

大多数客户端API的使用教程可以在相关程序包(roscpp, rospy, roslisp)中找到。

1. 手动创建ROS package
   本教程将展示如何手动创建ROS package
2. 管理系统依赖项
   本教程将展示如何使用rosdep安装系统依赖项.
3. Roslaunch在大型项目中的使用技巧
   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用。我们将使用 2dnav_pr2 package作为学习案例。
4. ROS在多机器人上的使用
   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。
5. 自定义消息
   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.
6. 在python中使用C++类
   本教程阐述一种在python中使用C++类的方法。
7. 如何编写教程
   （概述：）本教程介绍在编辑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。

ROS标准

- ROS开发者指南 有关代码风格和软件包布局等相关指南。
- 标准测量单位和坐标约定。

外部ROS资源

外部教程

- ANU的ROS视频教程
- NooTriX的手把手ROS教程
- Jonathan Bohren的ROS教程
- Clearpath Robotics 的知识库

外部研讨会或课件

- 由位于东京的TORK提供的面向企业基础培训的研讨会

在机器人上运行ROS

- 创建你自己的URDF文件 创建一个定制的通用机器人格式化描述文件。
- ros_control 使用ROS的标准控制器框架来与硬件连接。
- 在Gazebo中使用URDF 在Gazebo机器人模拟器中添加必要的标记。
- 搭建 MoveIt! 建立配置程序包（configuration package）来使用MoveIt!运动规划框架。

其它ROS函数库教程

- Robot Model
- Visualization
- actionlib
- pluginlib
- nodelets
- navigation
- TF

提供ROS接口的函数库教程

- Stage
- PCL with ROS

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

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 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 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'sourc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

catkin | rosbuild

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.

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

catkin | rosbuild

目录

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、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/install/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，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，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。

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查看程序包的依赖系。

Tutorial Level: BEGINNER

Next Tutorial: 编译ROS程序包

catkin | rosbuild

目录

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。个简单的工作空间也许看起来像这样：

```
workspace_folder/          -- WORKSPACE
  src/                     -- SOURCE SPACE
    CMakeLists.txt         -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt       -- CMakeLists.txt file for package 1
      package.xml          -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt       -- CMakeLists.txt file for package_n
      package.xml          -- Package manifest for package n
```

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_ms 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命令工具来查一级依赖包。

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

```
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer>  -->
  <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e属性中填写邮箱地址：

切换行号显示

```
  <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

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

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 Licen 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

```
  <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_ 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_dep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</url>
  <author email="you@yourdomain.tld">Jane Doe</author>

  <buildtool_depend>catkin</buildtool_depend>

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

  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run depend>std msgs</run depend>

</package>
```

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

catkin | rosbuild

目录

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_tutor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
uild"
```

```
####
-- 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.

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

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：

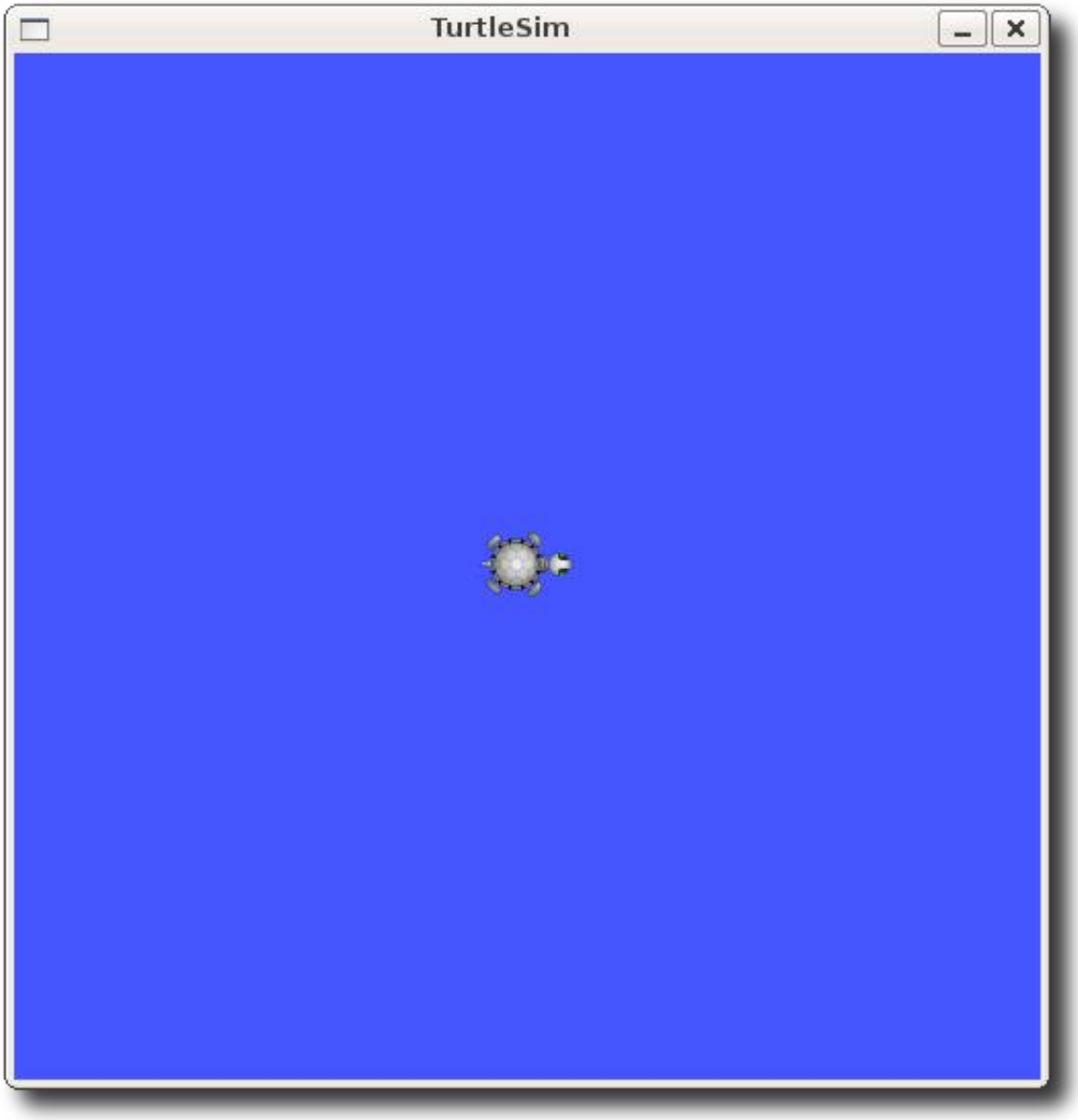
```
$ rosrun [package_name] [node_name]
```

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”

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

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
```

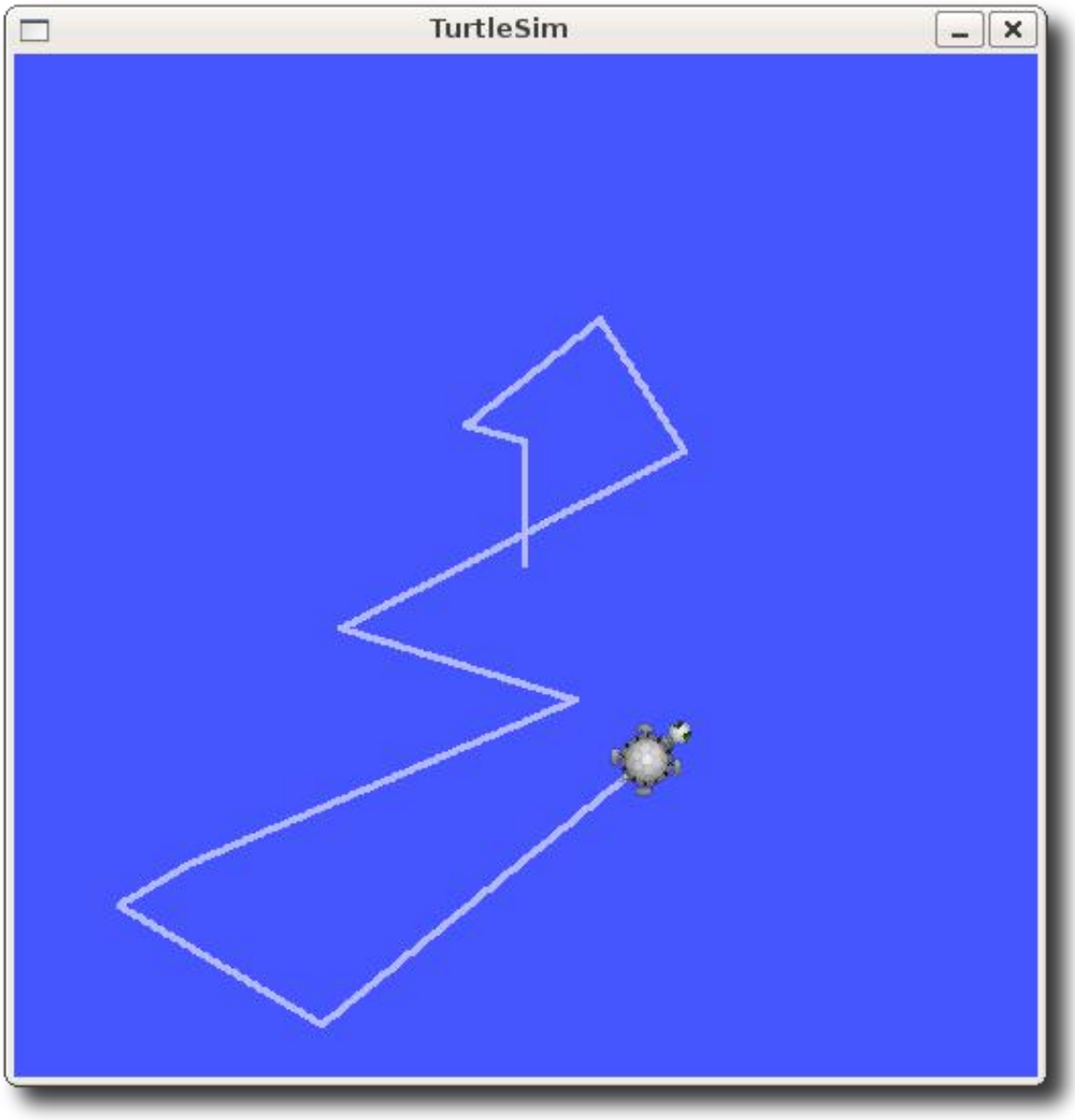
通过键盘远程控制turtle

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], pid [5528], boun
on [aqy], xmlrpc port [43918], tcpros port [55936], logging to [~/ros/ros/log
eleop_turtle_5528.log], using [real] 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

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_velocit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_k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。

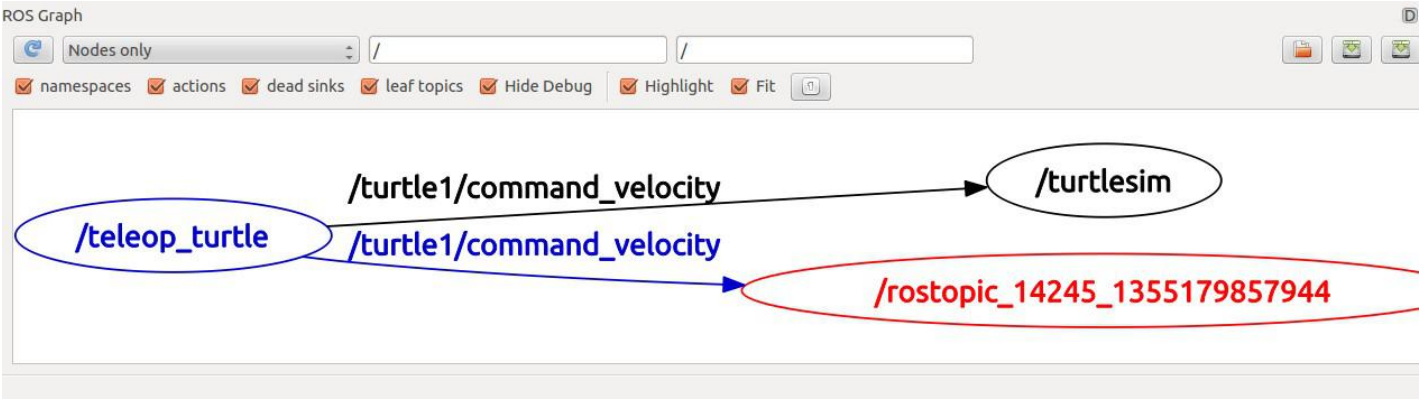
现在当你按下向上方向键时应该会看到下面的信息：

```
---
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]
```

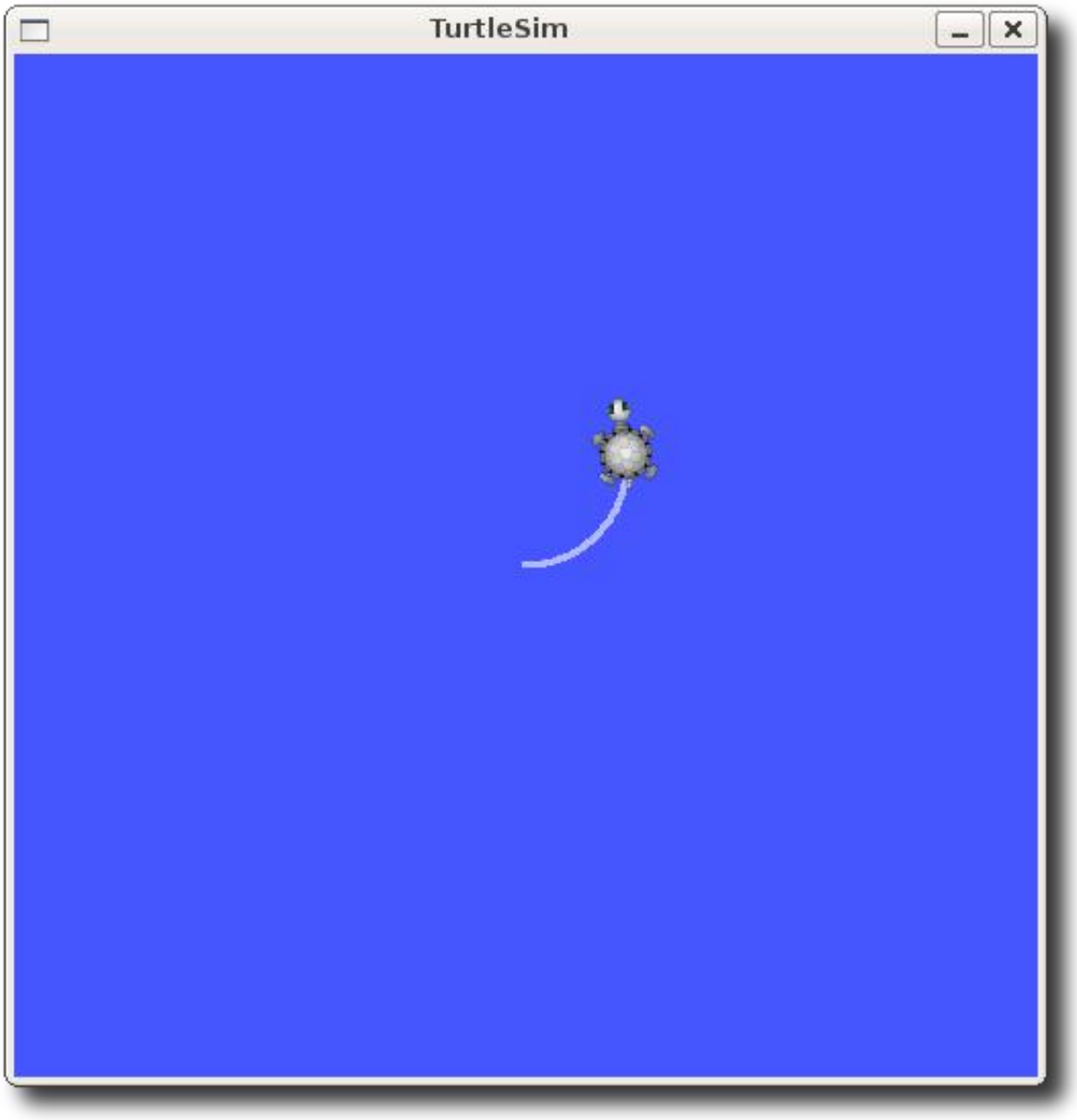
示例（非hydro版）：

```
$ 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。

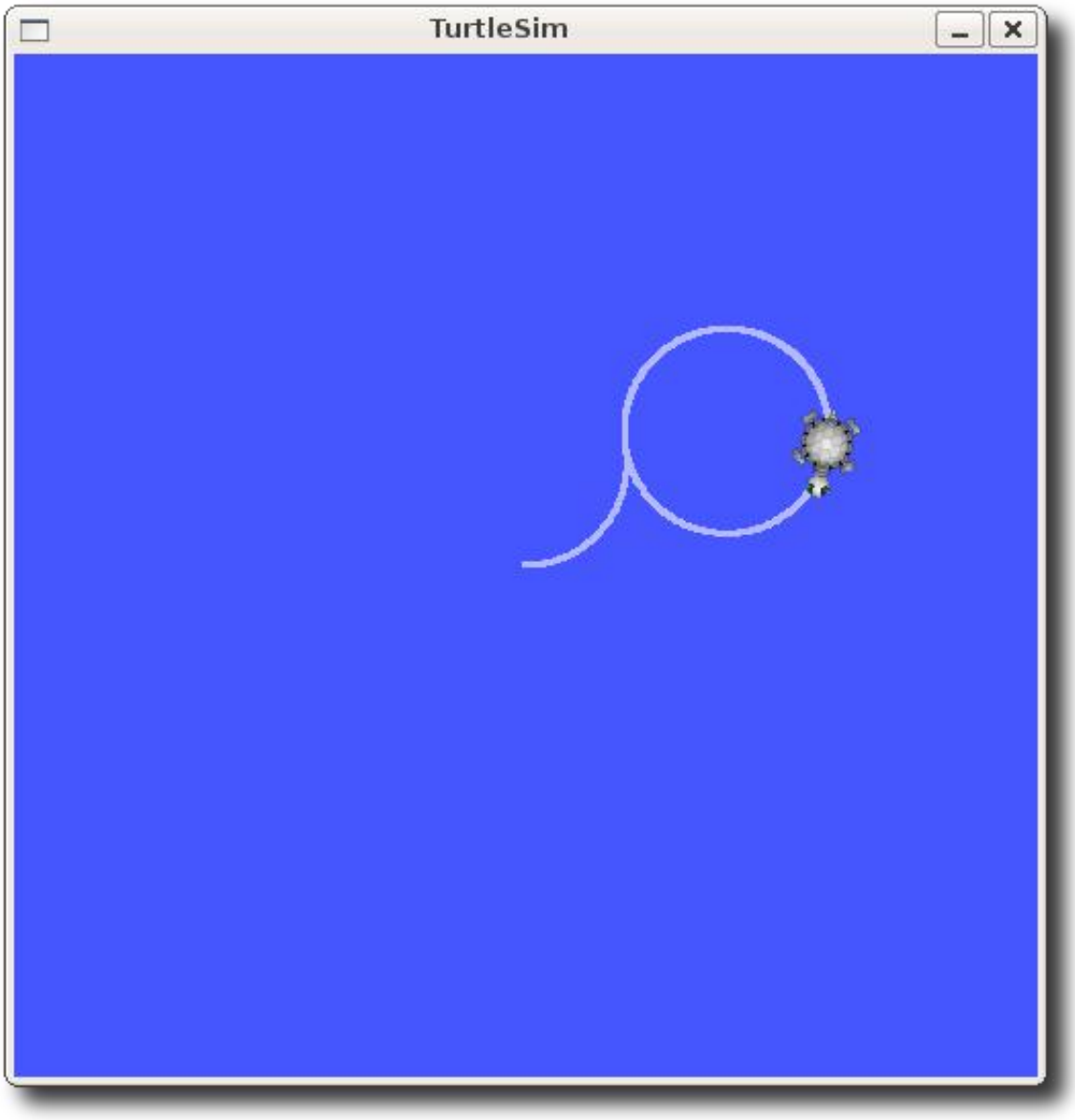
你可能已经注意到turtle已经停止移动了。这是因为turtle需要一个稳定的频率为1Hz的命令流来保持移动状态们可以使用rostopic pub -r命令来发布一个稳定的命令流（非hydro版）：

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

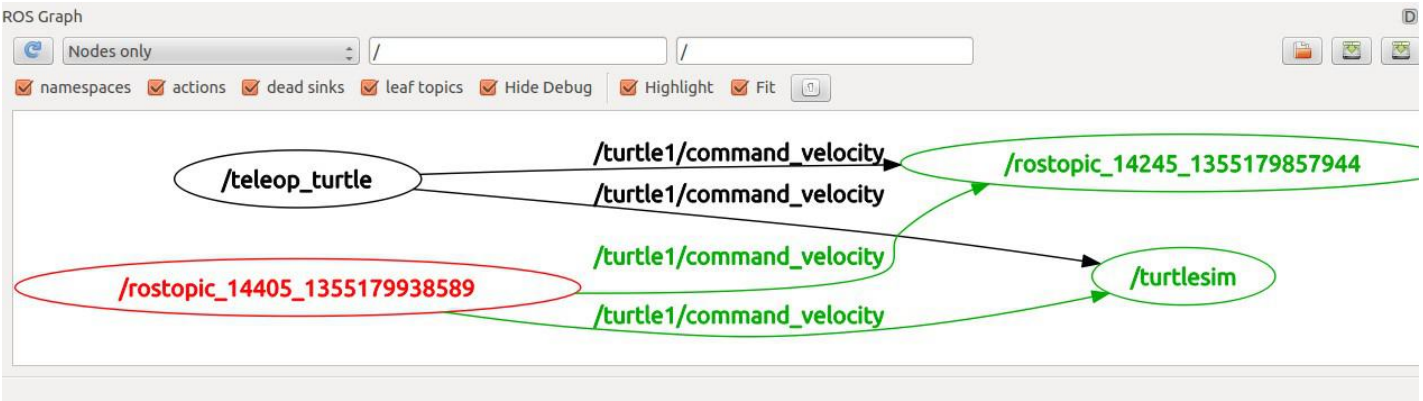
hydro版：

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

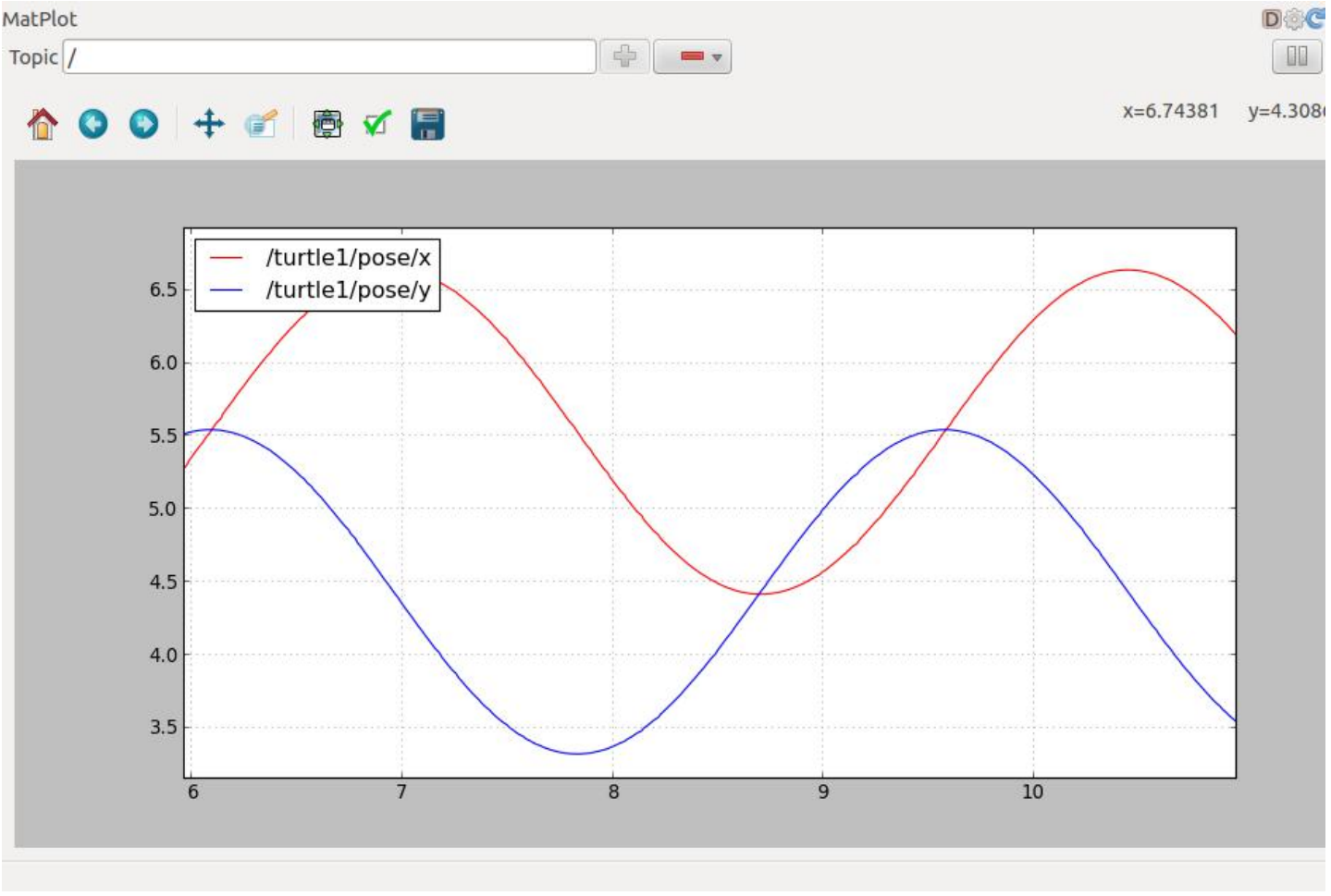
使用 rqt_plot

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。

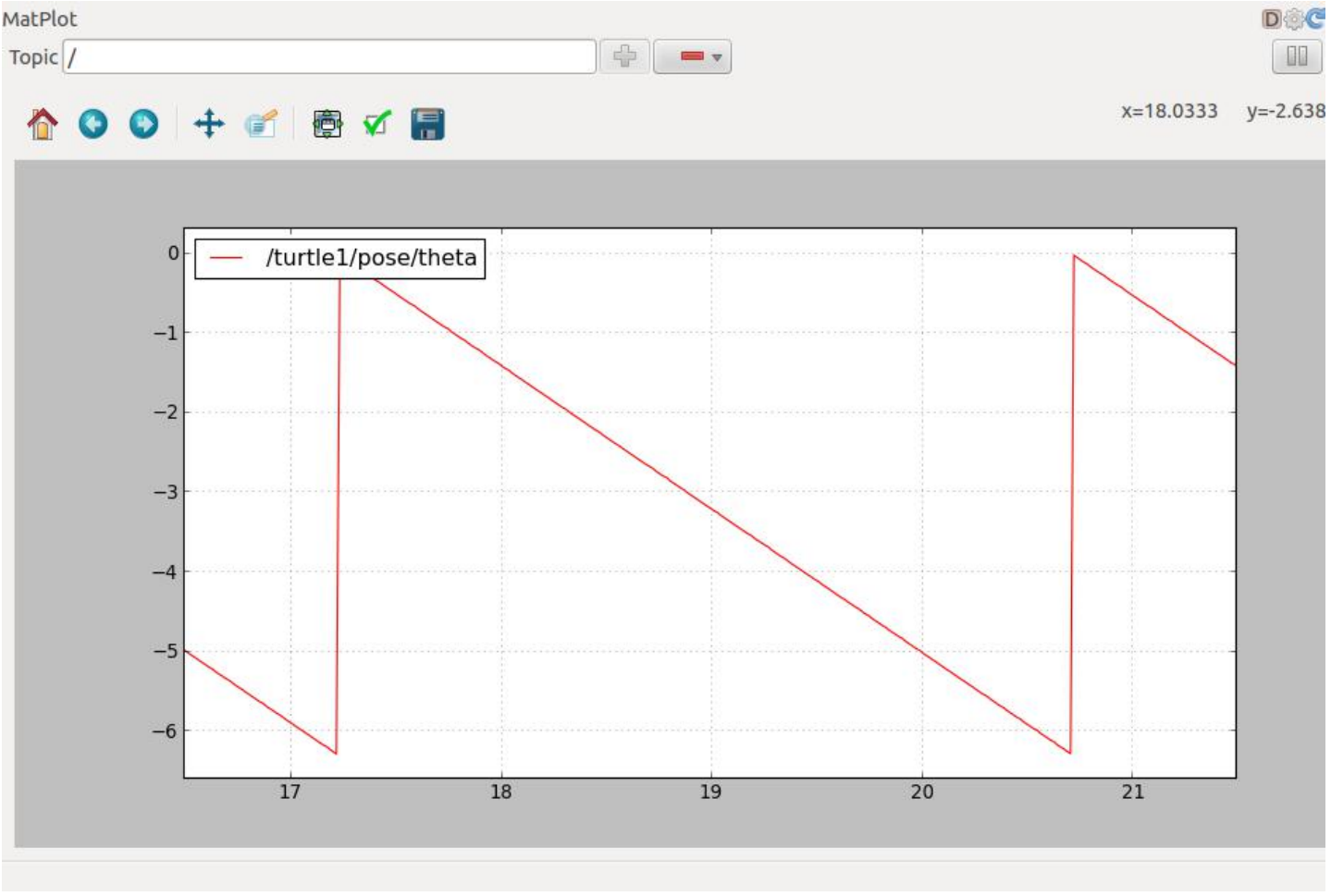
rqt_plot命令可以实时显示一个发布到某个话题上的数据变化图形。这里我们将使用rqt_plot命令来绘制正在布到/turtle1/pose话题上的数据变化图形。首先，在一个新终端中运行rqt_plot命令：

```
$ 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/po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。

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

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_loandturtlesim/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

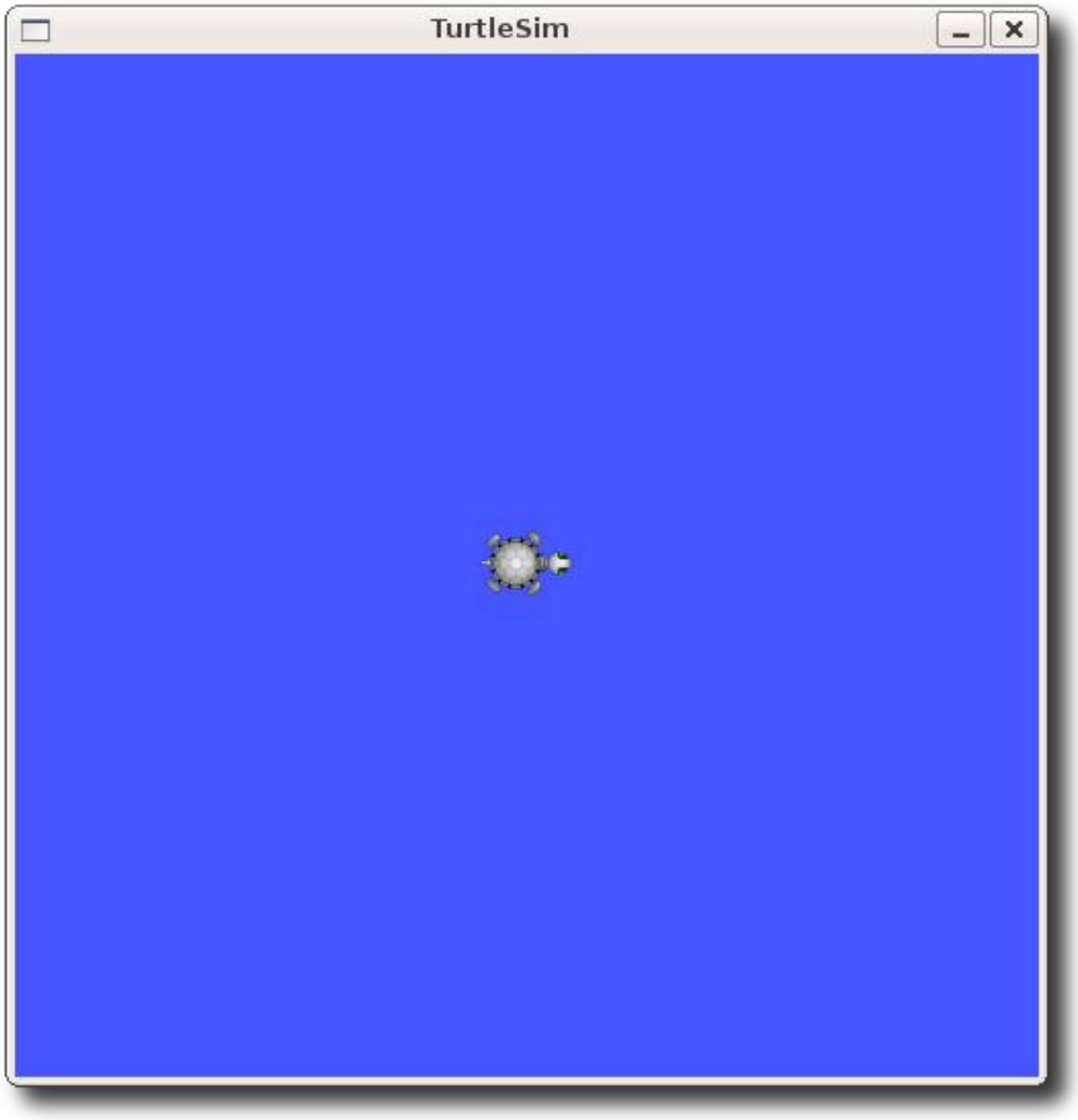
使用方法:

```
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
```

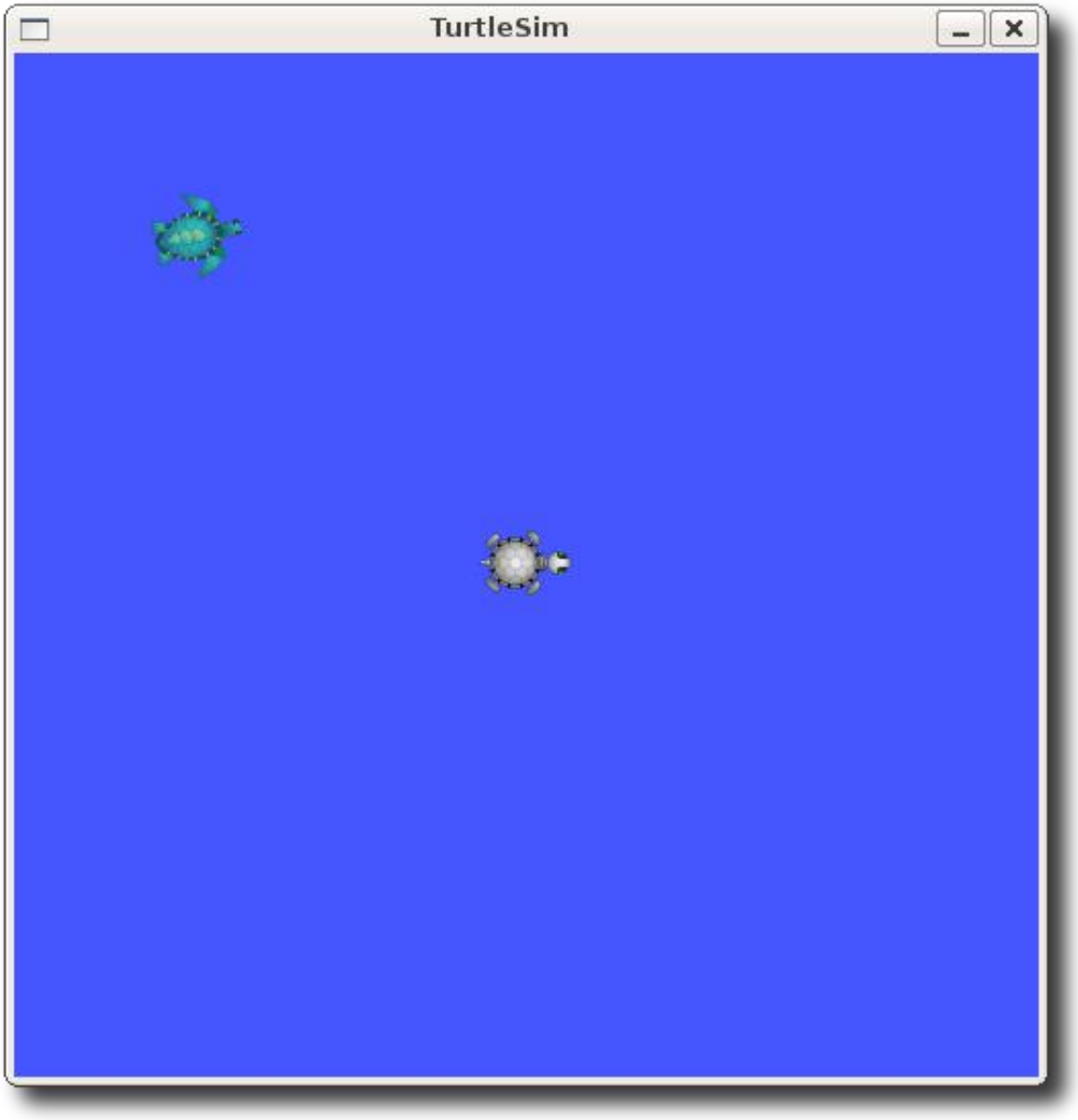
这个服务使得我们可以在给定的位置和角度生成一只新的乌龟。名字参数是可选的，这里我们不设具体的名字让turtlesim自动创建一个。

```
$ 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 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           从文件读取参数向
rosparam dump           文件中写入参数删
rosparam delete         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]
```

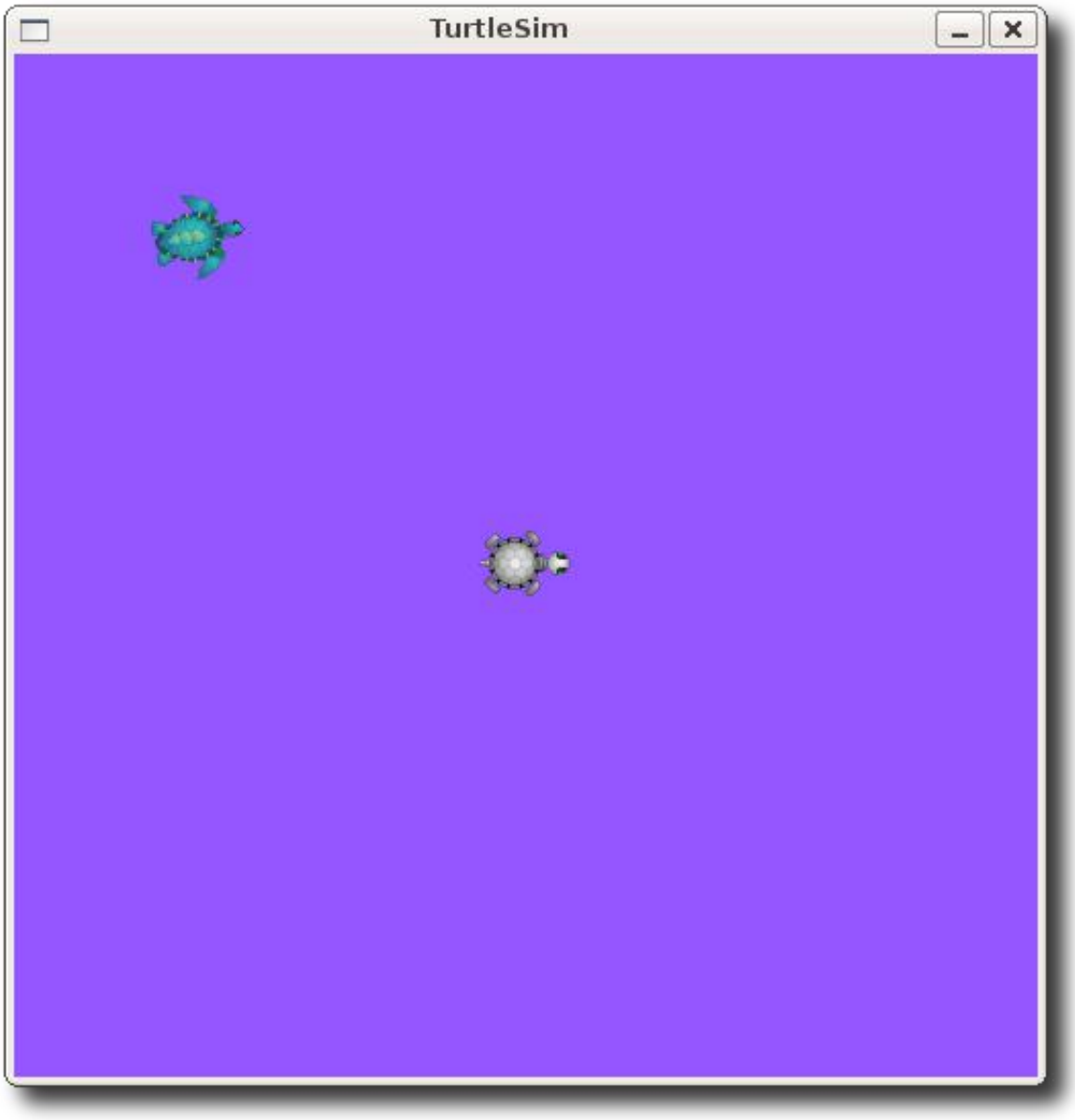
现在我们修改背景颜色的红色通道：

```
$ 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

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

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-plugins 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

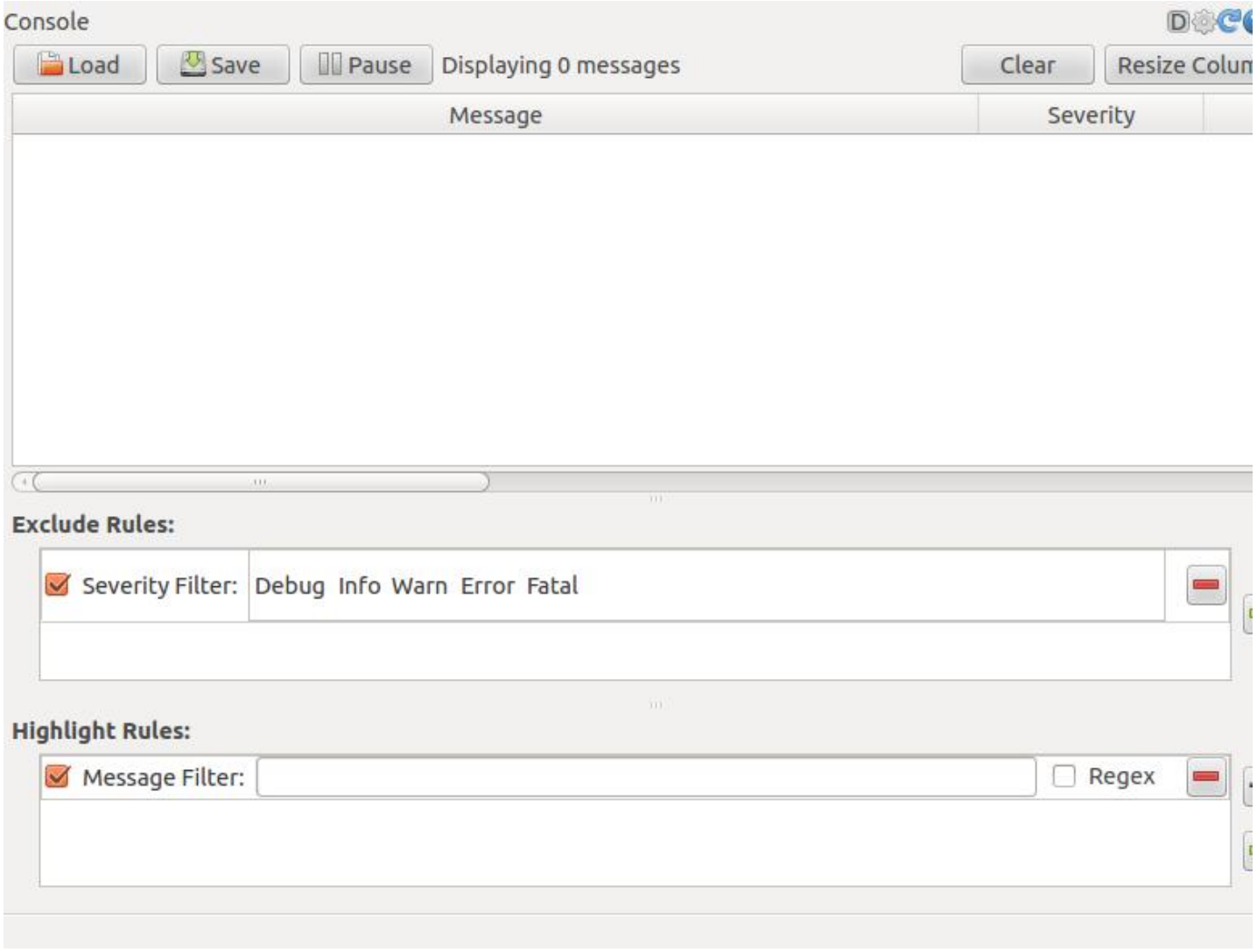
rqt_console属于ROS日志框架(logging framework)的一部分，用来显示节点的输出信息。rqt_logger_level我们修改节点运行时输出信息的日志等级（logger levels）（包括 DEBUG、WARN、INFO和ERROR）。

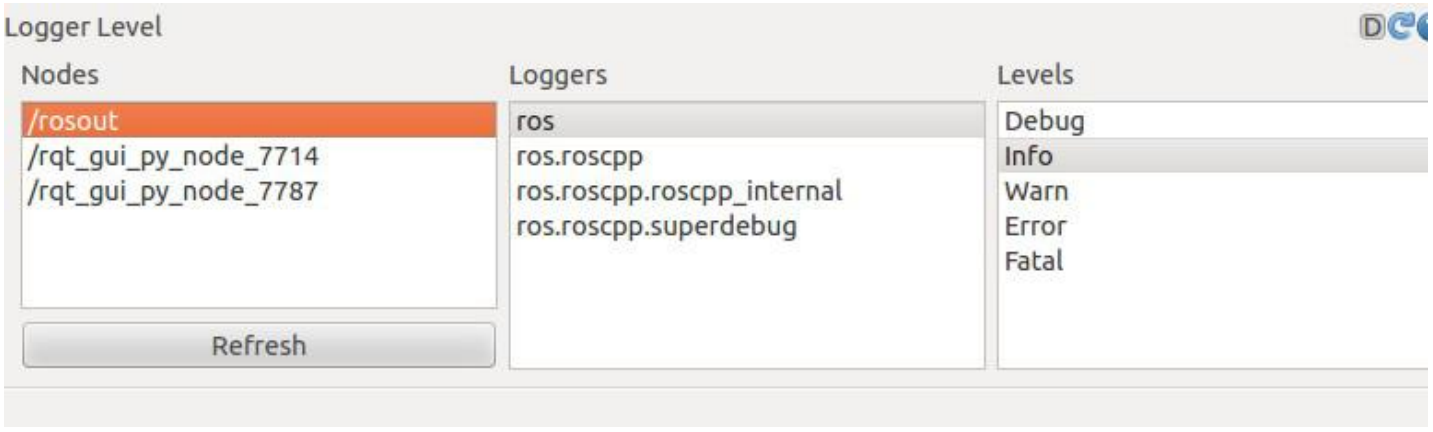
现在让我们来看一下turtlesim在rqt_console中的输出信息，同时在rqt_logger_level中修改日志等级。在启turtlesim之前先在另外两个新终端中运行rqt_console和rqt_logger_level：

```
$ rosrun rqt_console rqt_console
```

```
$ rosrun rqt_logger_level rqt_logger_level
```

你会看到弹出两个窗口：

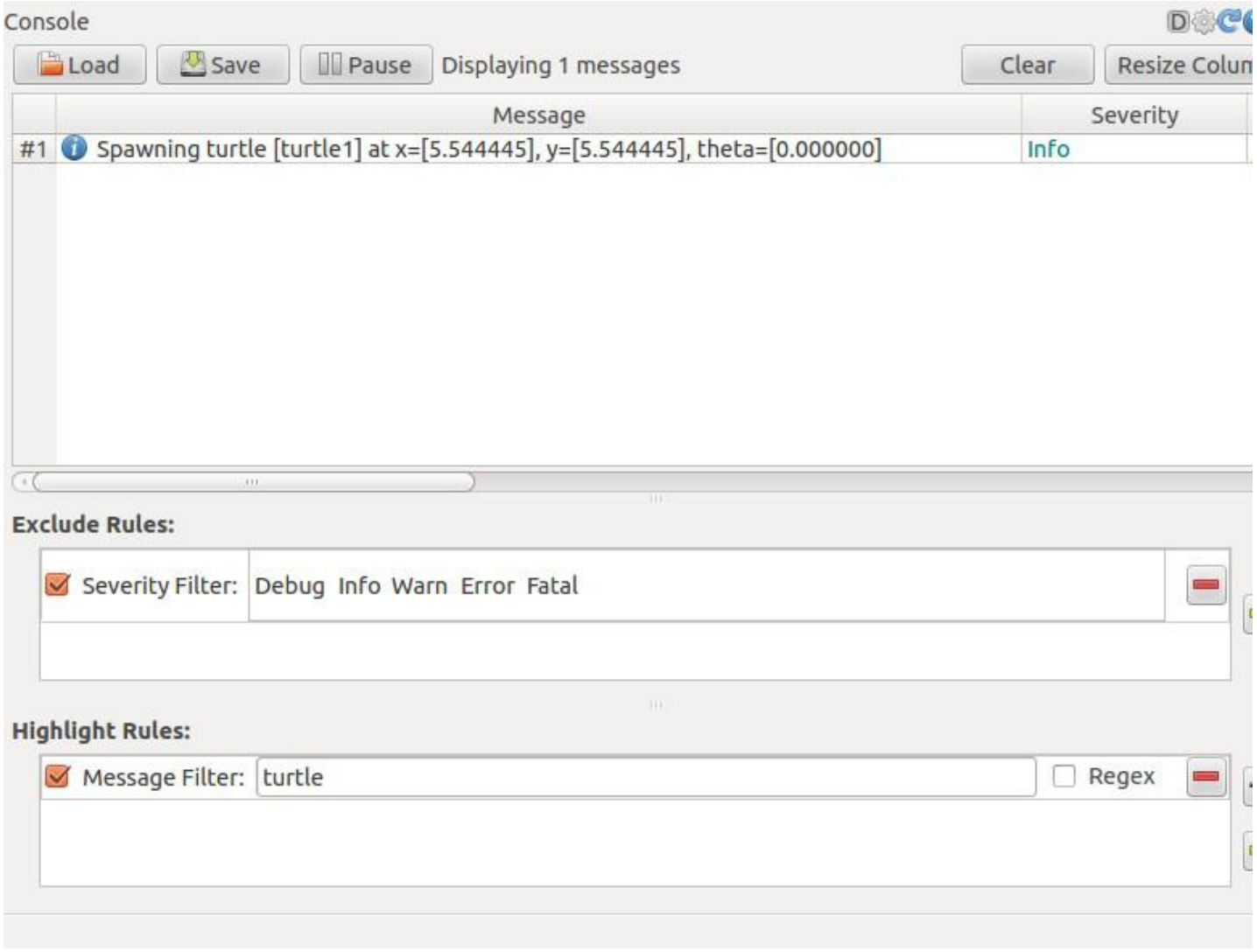




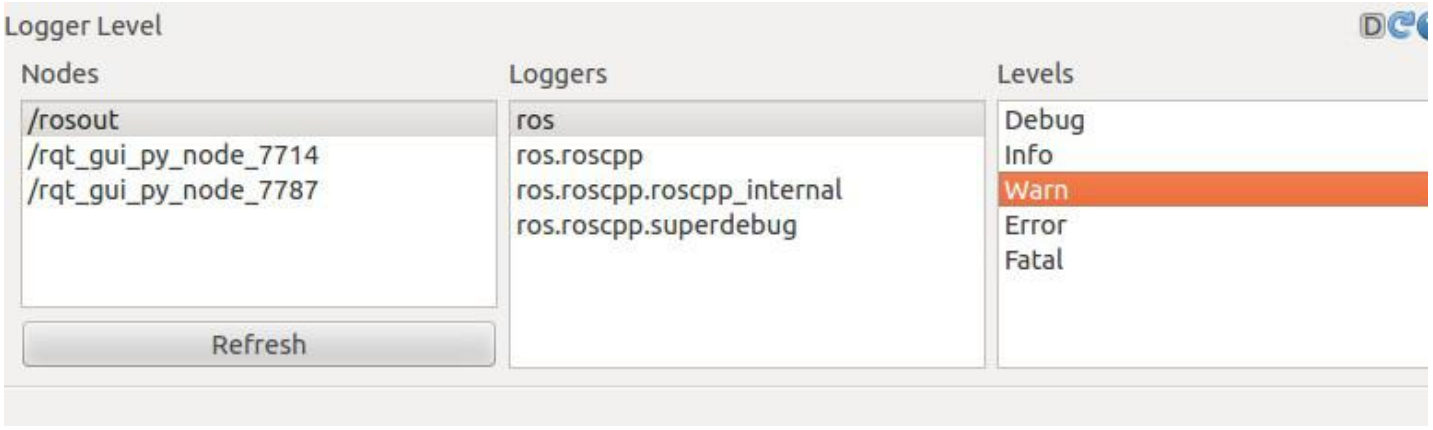
现在让我们在一个新终端中启动turtlesim：

```
$ 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：

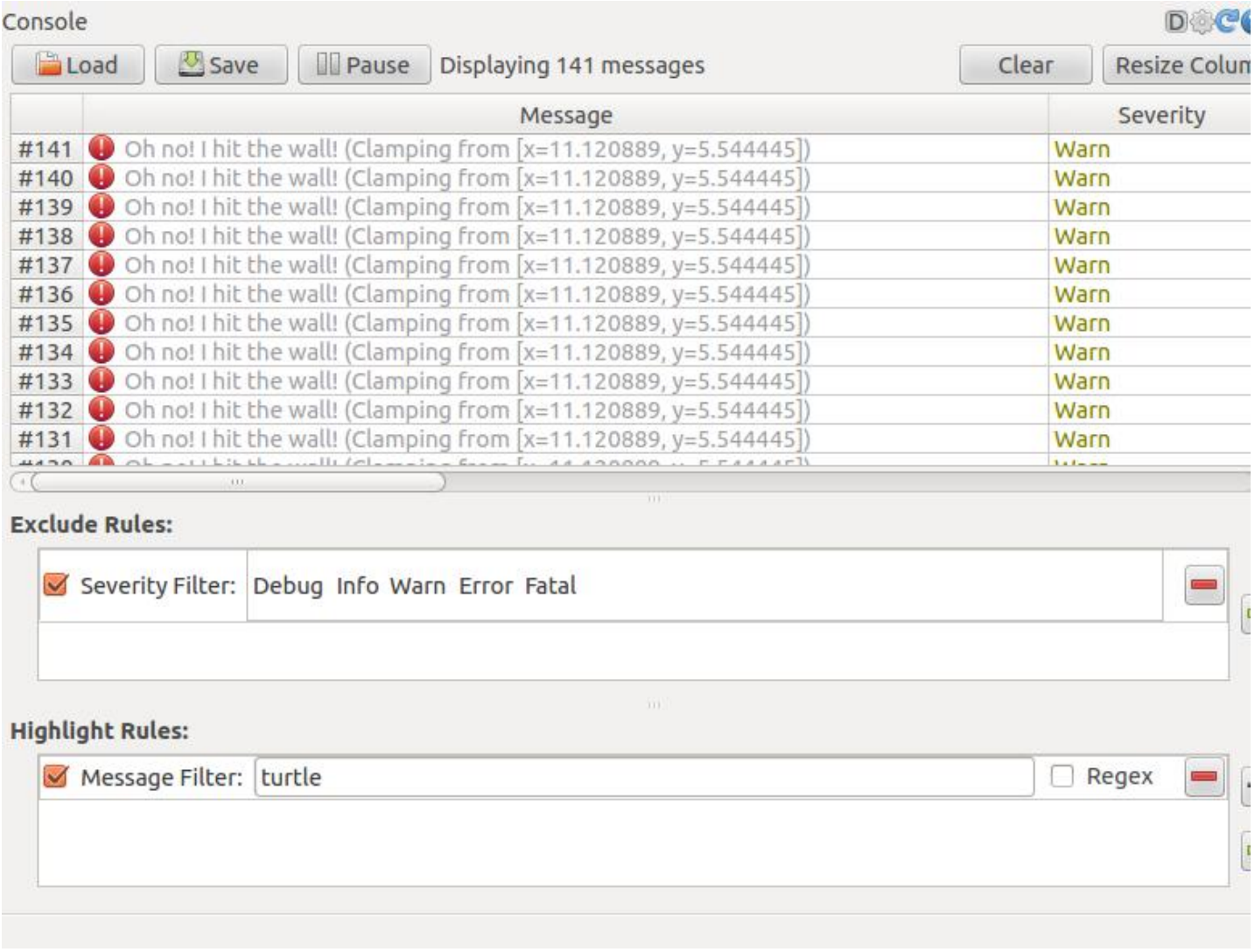


现在我们让turtle动起来并观察rqt_console中的输出（非hydro版）：

```
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

```
3   <group ns="turtlesim1">
4     <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
5   </group>
6
7   <group ns="turtlesim2">
8     <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
9   </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器而不会产生命名冲突。

切换行号显示

```
11   <node pkg="turtlesim" name="mimic" type="mimic">
12     <remap from="input" to="turtlesim1/turtle1"/>
13     <remap from="output" to="turtlesim2/turtle1"/>
14   </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

```
16 </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：

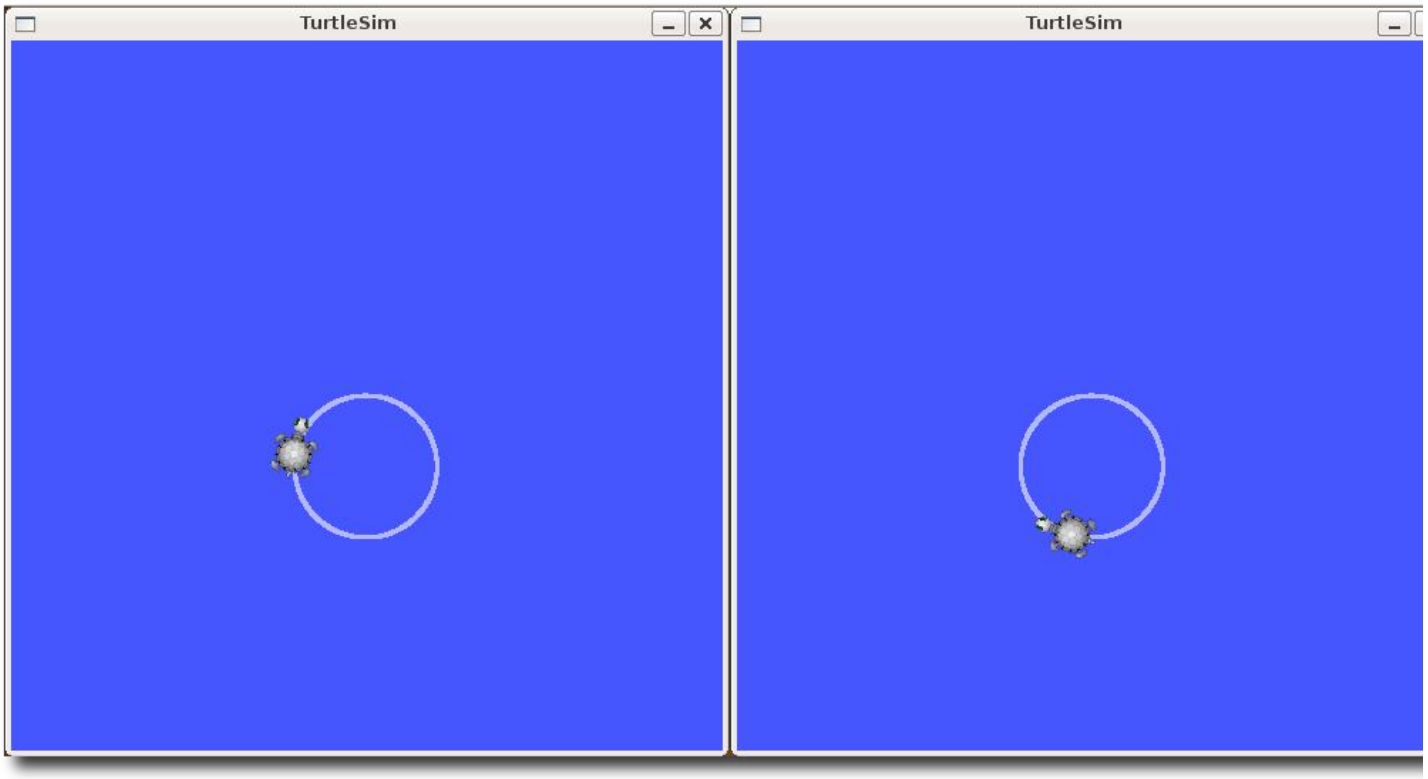
非hydro版：

```
$ rostopic pub /turtlesim1/turtle1/command_velocity turtlesim/Velocity -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。

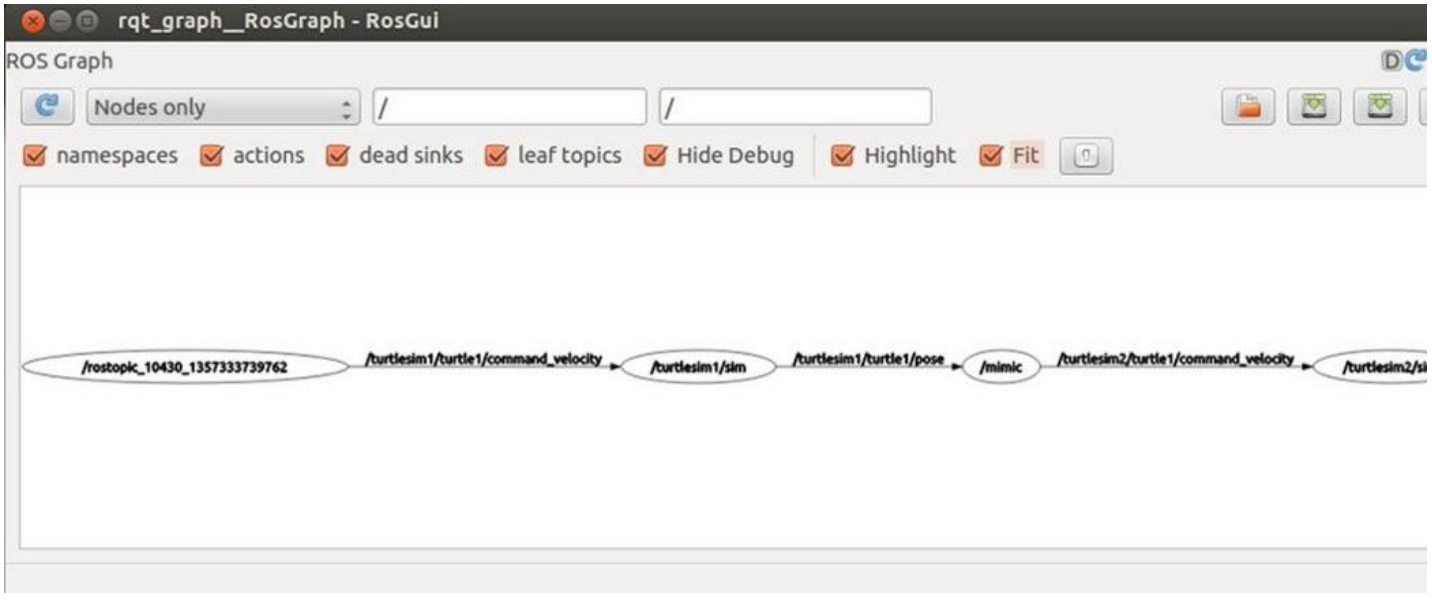


我们也可以通过rqt_graph来更好的理解在launch文件中所做的事情。运行rqt并在主窗口中选择rqt_graph：

```
$ 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。

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

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.

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

Tutorial Level: BEGINNER

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

catkin | rosbuild

目录

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 existing 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

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( FI
  LES
  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 ms 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 existing 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( FI
  LES
  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 ms 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roscpp,rospy)，结果如下：

```
generate_messages( DE
  PENDENCIES
  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 : **cp**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++).

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

catkin | rosbuild

目录

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

```
27 #include "ros/ros.h"
28 #include "std_msgs/String.h"
29
30 #include <sstream>
31
32 /**
33  * This tutorial demonstrates simple sending of messages  over  the ROS system.
34  */
35 int main(int argc, char **argv)
36 {
37   /**
```

```
38     * The ros::init() function needs to see argc and argv so that it can  perform
39     * any ROS arguments and name remapping that were provided at the command line. For
rogrammatic
40     * remappings you can use a different version of init() which takes  remappings
41     * directly, but for most command-line programs, passing argc and argv is the  easie
42     * way to do it.  The third argument to init() is the name of the node.
43     *
44     * You must call one of the versions of ros::init() before using any  other
45     * part of the ROS system.
46     */
47    ros::init(argc, argv, "talker");
48
49    /**
50     * NodeHandle is the main access point to communications with the ROS  system.
51     * The first NodeHandle constructed will fully initialize this node, and the  last
52     * NodeHandle destructed will close down the node.
53     */
54    ros::NodeHandle n;
55
56    /**
57     * The advertise() function is how you tell ROS that you want  to
58     * 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  master
61     * node will notify anyone who is trying to subscribe to this topic  name,
62     * and they will in turn negotiate a peer-to-peer connection with  this
63     * node.  advertise() returns a Publisher object which allows you  to
64     * publish messages on that topic through a call  to publish().  Once
65     * all copies of the returned Publisher object are destroyed, the  topic
66     * will be automatically unadvertised.
67     *
68     * The second parameter to advertise() is the size of the message  queue
69     * used for publishing messages.  If messages are published more  quickly
70     * than we can send them, the number here specifies how many messages  to
71     * buffer up before throwing some away.
72     */
73    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
74
75    ros::Rate loop_rate(10);
76
77    /**
78     * A count of how many messages we have sent. This is used to  create
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 publish it.
86       */
87      std_msgs::String msg;
88
89      std::stringstream ss;
90      ss << "hello world " << count;
91      msg.data = ss.str();
```

```
  92
  93     ROS_INFO("%s", msg.data.c_str());
  94
  95     /**
  96      * The publish() function is how you send messages. The parameter
  97      * is the message object. The type of this object must agree with the  type
  98      * given as a template parameter to the advertise<>() call, as was  done
  99      * in the constructor above.
 100      */
 101     chatter_pub.publish(msg);
 102
 103     ros::spinOnce();
 104
 105     loop_rate.sleep();
 106     ++count;
 107   }
 108
 109
 110   return 0;
 111 }
```

代码解释

现在，我们来分段解释代码.

切换行号显示

```
  27 #include "ros/ros.h"
  28
```

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

```
  28 #include "std_msgs/String.h"
  29
```

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。需要更详细的定义,参考msg页面.

切换行号显示

```
  47   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

```
  54   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源。

切换行号显示

```
73    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布消息； 2) 如果消息类型不对,它会拒绝发布。

```
切换行号显示

75    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

81    int count = 0;
82    while (ros::ok())
83    {
```

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

87      std_msgs::String msg;
88
89      std::stringstream ss;
90      ss << "hello world " << count;
91      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 it can  perform
  43    * any ROS arguments and name remapping that were provided at the command line. For
rogrammatic
  44    * remappings you can use a different version of init() which takes  remappings
  45    * directly, but for most command-line programs, passing argc and argv is the  easie
  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 the ROS  system.
55    * The first NodeHandle constructed will fully initialize this node, 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 receive  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 object that  you
66    * must hold on to until you want  to unsubscribe.  When all copies of the  Subscribe
67    * object go out of scope, this callback will automatically be unsubscribed  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 processed,  this
72    * is the number of messages that will be buffered up before beginning to  throw
73    * away the oldest ones.
74    */
75   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
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 one).  ros::spin()
80    * will exit when Ctrl-C is pressed, or the node is shutdown by the  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, chatterCallback);
```

告诉master我们要订阅chatter topic上的消息。当有消息到达topic时，ROS就会调用chatterCallback()函数。第二个参队列大小，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Ctl 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/CMakeL

```
切换行号显示

   1 cmake_minimum_required(VERSION 2.8.3)
   2 project(beginner_tutorials)
   3
   4 ## Find catkin and any catkin packages
   5 find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)
   6
   7 ## Declare ROS messages and services
   8 add_message_files(DIRECTORY msg FILES Num.msg)
   9 add_service_files(DIRECTORY srv FILES AddTwoInts.srv)
  10
  11 ## Generate added messages and services
  12 generate_messages(DEPENDENCIES std_msgs)
  13
  14 ## Declare a catkin package
  15 catkin_package()
```

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/CMakeLi

切换行号显示

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

这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*Groov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.

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

catkin | rosbuild

目录

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_t
ker_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

1 #!/usr/bin/env python
2 # license removed for brevity
3 import rospy
```

```
 4 from std_msgs.msg import String
 5
 6 def talker():
 7     pub = rospy.Publisher('chatter', String,  queue_size=10)
 8     rospy.init_node('talker', anonymous=True)
 9     rate = rospy.Rate(10) # 10hz
10     while not rospy.is_shutdown():
11         hello_str = "hello world %s" %  rospy.get_time()
12         rospy.loginfo(hello_str)
13         pub.publish(hello_str)
14         rate.sleep()
15
16 if   name   == '  main  ':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass
```

The Code Explained

Now, let's break the code down.

```
切换行号显示

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

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

```
切换行号显示

 3 import rospy
 4 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

 7     pub = rospy.Publisher('chatter', String, queue_size=10)
 8     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 publishi 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 subscri 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 rosp has this information, it cannot start communicating with the ROS Master. In this case, your node will take o

the name `talker`. NOTE: the name must be a base name, i.e. it cannot contain any slashes "/".

```
切换行号显示

    9     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 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

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

This loop is a fairly standard rospy construct: checking the `rospy.is_shutdown()` flag and then doing work. have to check `is_shutdown()` to check if your program should exit (e.g. if there is a `Ctrl-C` or otherwise). In 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 through the loop.

(You may also run across `rospy.sleep()` which is similar to `time.sleep()` except that it works with simulat 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 p 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

   17     try:
   18         talker()
   19     except rospy.ROSInterruptException:
   20         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_t
ker_listener/listener.py
```

The file contents look close to:

切换行号显示

```
1  #!/usr/bin/env python
2  import rospy
3  from std_msgs.msg import String
4
5  def callback(data):
6      rospy.loginfo(rospy.get caller id() + "I heard %s", data.data)
7
8  def listener():
9
10     # In ROS, nodes are uniquely named. If two nodes with the same
11     # node are launched, the previous one is kicked off. The
12     # anonymous=True flag means that rospy will choose a unique
13     # name for our 'listener' node so that multiple listeners can
14     # run simultaneously.
15     rospy.init_node('listener', anonymous=True)
16
17     rospy.Subscriber("chatter", String, callback)
18
19     # spin() simply keeps python from exiting until this node is stopped
20     rospy.spin()
21
22 if __name__ == '__main__':
23     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 mechanis for subscribing to messages.

```
切换行号显示

15      rospy.init_node('listener', anonymous=True)
16
17      rospy.Subscriber("chatter", String, callback)
18
19      # spin() simply keeps python from exiting until this node is stopped
20      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. U 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 sur 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.

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

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catk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_13149319687951 heard hel
world 1314931969.26
[INFO] [WallTime: 1314931970.262246] /listener_17657_13149319687951 heard hel
world 1314931970.26
[INFO] [WallTime: 1314931971.266348] /listener_17657_13149319687951 heard hel
world 1314931971.26
[INFO] [WallTime: 1314931972.270429] /listener_17657_13149319687951 heard hel
world 1314931972.27
[INFO] [WallTime: 1314931973.274382] /listener_17657_13149319687951 heard hel
world 1314931973.27
[INFO] [WallTime: 1314931974.277694] /listener_17657_13149319687951 heard hel
world 1314931974.28
[INFO] [WallTime: 1314931975.283708] /listener_17657_13149319687951 heard hel
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++).

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

catkin | rosbuild

目录

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;
}

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

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

```
  4
  5 int main(int argc, char **argv)
  6 {
  7   ros::init(argc, argv, "add_two_ints_client");
  8   if (argc != 3)
  9   {
 10     ROS_INFO("usage: add_two_ints_client X Y");
 11     return 1;
 12   }
 13
 14   ros::NodeHandle n;
 15   ros::ServiceClient client =  n.serviceClient<beginner_tutorials::AddTwoInts>("add_tw
o_ints");
 16   beginner_tutorials::AddTwoInts srv;
 17   srv.request.a = atoll(argv[1]);
 18   srv.request.b = atoll(argv[2]);
 19   if (client.call(srv))
 20   {
 21     ROS_INFO("Sum: %ld", (long  int)srv.response.sum);
 22   }
 23   else
 24   {
 25     ROS_ERROR("Failed to call service  add_two_ints");
 26     return 1;
 27   }
 28
 29   return 0;
 30 }
```

代码解释

现在，让我们来逐步分析代码。

切换行号显示

```
 15   ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_tw
o_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

```
 16   beginner_tutorials::AddTwoInts srv;
 17   srv.request.a = atoll(argv[1]);
 18   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req和response。同时也包括两个类定义Request和Response。

切换行号显示

```
 19   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.t
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/CMakeL

切换行号显示

```
27 add_executable(add_two_ints_server src/add_two_ints_server.cpp)
28 target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
29 add_dependencies(add_two_ints_server beginner_tutorials_gencpp)
30
31 add_executable(add_two_ints_client src/add_two_ints_client.cpp)
32 target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
33 add_dependencies(add_two_ints_client beginner_tutorials_gencpp)
```

这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dev 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。

编写简单的Service和Client (Python)

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

Tutorial Level: BEGINNER

Next Tutorial: 测试简单的Service和Client

catkin | rosbuild

目录

Writing a Service Node

Here we'll create the service ("add_two_ints_server") node which will receive two ints and return the sum.

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

```
$ roscd beginner_tutorials
```

Please make sure you have followed the directions in the previous tutorial for creating the service needed i this tutorial,creating the AddTwoInts.srv (be sure to choose the right version of build tool you're using at th top of wiki page in the link).

The Code

Create the **scripts/add_two_ints_server.py** file within the beginner_tutorials package and paste the follow inside it:

切换行号显示

```
1 #!/usr/bin/env python
2
3 from beginner_tutorials.srv import *
4 import rospy
5
6 def handle_add_two_ints(req):
7     print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
8     return AddTwoIntsResponse(req.a + req.b)
9
```

```
10 def add two ints server():
11     rospy.init node('add two ints server')
12     s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
13     print "Ready to add two ints."
14     rospy.spin()
15
16 if   name   == "  main  ":
17     add_two_ints_server()
```

Don't forget to make the node executable:

```
chmod +x scripts/add_two_ints_server.py
```

The Code Explained

Now, let's break the code down.

There's very little to writing a service using rospy. We declare our node using `init_node()` and then declar service:

```
切换行号显示

12     s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
```

This declares a new service named `add_two_ints` with the `AddTwoInts` service type. All requests are passed to`handle_add_two_ints` function. `handle_add_two_ints` is called with instances of `AddTwoIntsRequest` and returns instances of`AddTwoIntsResponse`.

Just like with the subscriber example, `rospy.spin()` keeps your code from exiting until the service is shutd

Writing the Client Node

The Code

Create the **scripts/add_two_ints_client.py** file within the beginner_tutorials package and paste the followi inside it:

```
切换行号显示

 1 #!/usr/bin/env python
 2
 3 import sys
 4 import rospy
 5 from beginner_tutorials.srv import *
 6
 7 def add_two_ints_client(x, y):
 8     rospy.wait_for_service('add_two_ints')
 9     try:
10         add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
11         resp1 = add_two_ints(x, y)
12         return resp1.sum
```

```
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

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

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x,  y))
```

Don't forget to make the node executable:

```
$ chmod +x scripts/add_two_ints_client.py
```

The Code Explained

Now, let's break the code down.

The client code for calling services is also simple. For clients you don't have to call `init_node()`. We first c

```
切换行号显示

    rospy.wait_for_service('add_two_ints')
```

This is a convenience method that blocks until the service named `add_two_ints` is available. Next we creat handle for calling the service:

```
切换行号显示

        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```

We can use this handle just like a normal function and call it:

```
切换行号显示

        resp1 = add_two_ints(x, y)
        return resp1.sum
```

Because we've declared the type of the service to be `AddTwoInts`, it does the work of generating the `AddTwoIntsRequest`object for you (you're free to pass in your own instead). The return value is an `AddTwoIntsResponse` object. If the call fails, a`rospy.ServiceException` may be thrown, so you should set the appropriate `try/except` block.

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 sur that the autogenerated Python code for messages and services is created.

Go to your catkin workspace and run `catkin_make`.

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

Try it out!

In a **new terminal**, run

```
$ cd ~/catkin_ws
$ . devel/setup.bash
$ rosrun beginner_tutorials add_two_ints_server.py
```

In a **new terminal**, run

```
$ cd ~/catkin_ws
$ . devel/setup.bash
$ rosrun beginner_tutorials add_two_ints_client.py
```

And you will see the usage information printed, similar to

```
/home/user/catkin_ws/src/beginner_tutorials/scripts/add_two_ints_client.py [x
y]
```

Then run

```
$ rosrun beginner_tutorials add_two_ints_client.py 4 5
```

And you will get

```
Requesting 4+5
4 + 5 = 9
```

And the server will print out

```
Returning [4 + 5 = 9]
```

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。

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

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Ser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LISP都可以.

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相似的运行过程。

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

Tutorial Level: BEGINNER

Next Tutorial: 使用roswt

目录

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turtlesi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**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/po
e...
[ INFO] 1260210511.072384000: Done sleeping.

[ INFO] 1260210511.277391000: Sleeping 0.200 seconds after advertising /turtle1/co
r_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**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（launfile）：

```
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**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**ros 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—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

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

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，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 directory:
 * 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

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维基找到你需要的信息。

Keywords: wiki

Tutorial Level: BEGINNER

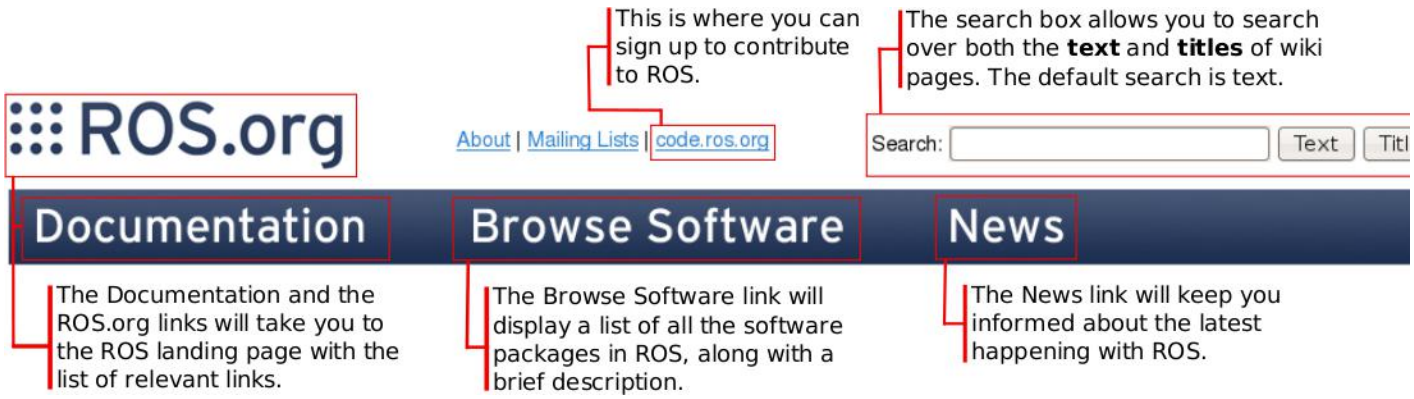
Next Tutorial: 接下来做什么？

目录

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。

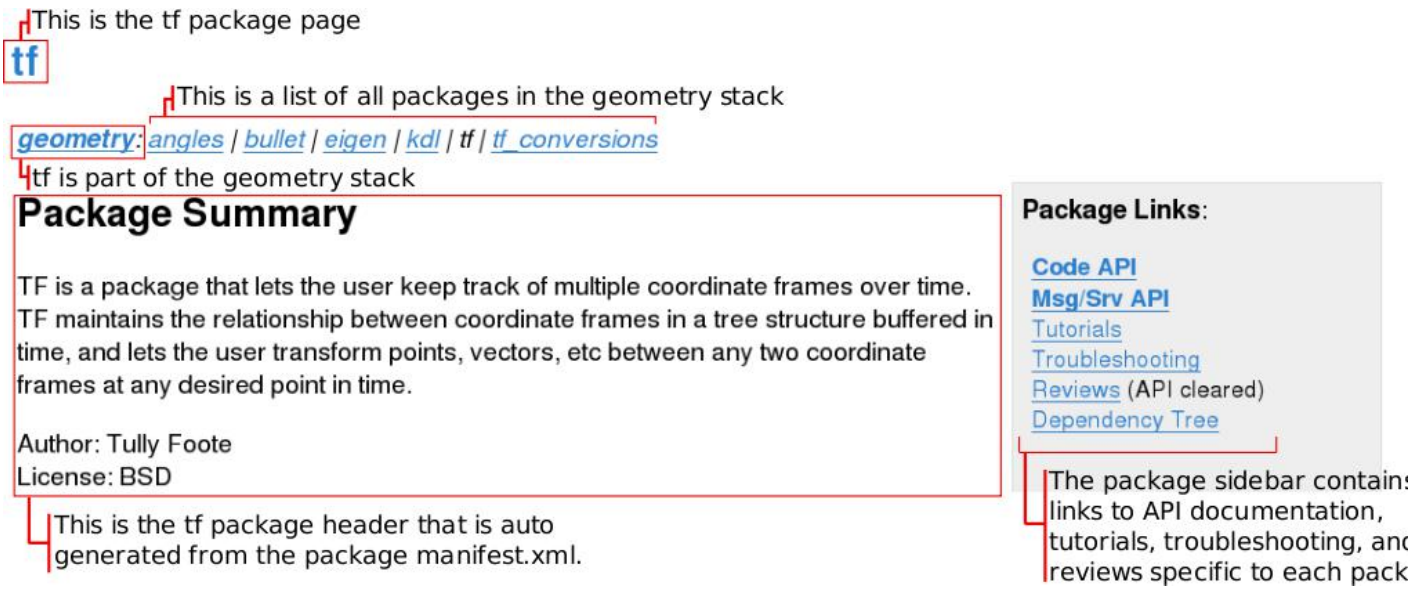
首页

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。



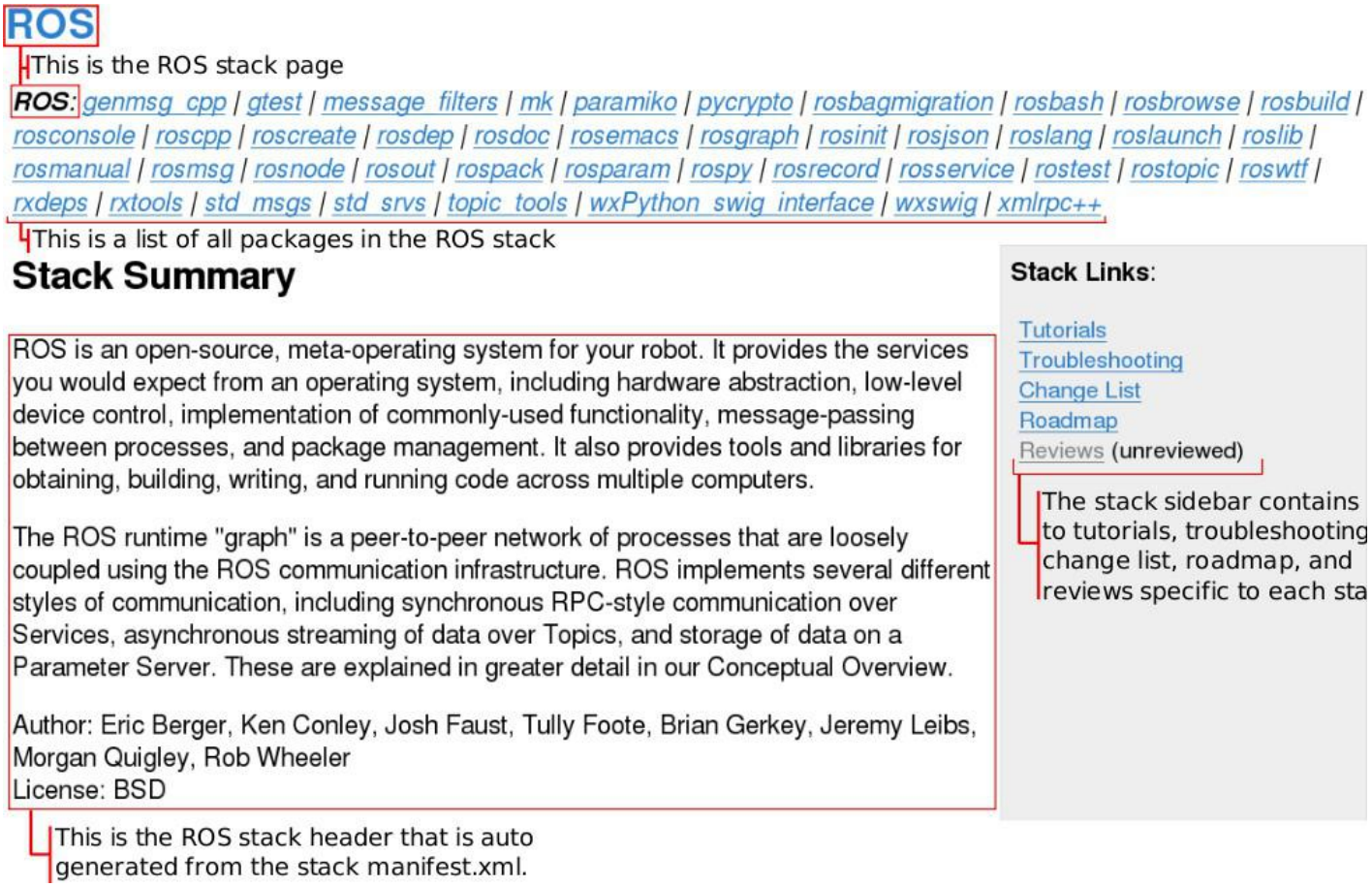
package页面

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.

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

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.

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setu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</maintainer>
  <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，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

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

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，你会看到如的报错消息：

```
ERROR: your rosdep installation has not been initialized yet.  Please 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 ] ; then
  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-log4cxx-0.10.0-
g_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。

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

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 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*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。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

```
<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="$(env ROS_ROOT)" ros-package-path="$
```

```
nv ROS_PACKAGE_PATH)" default="true" />
  <machine name="c2" address="pre2" ros-root="$(env ROS_ROOT)" ros-package-path="$
nv 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*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="$(find 2dnav_pr2)/config/costmap_common_params.yaml" command="lo
d" ns="global_costmap" />
  <rosparam file="$(find 2dnav_pr2)/config/costmap_common_params.yaml" command="lo
d" ns="local_costmap" />
  <rosparam file="$(find 2dnav_pr2)/move_base/local_costmap_params.yaml" command="
ad" />
  <rosparam file="$(find 2dnav_pr2)/move_base/global_costmap_params.yaml" command=
oad" />
  <rosparam file="$(find 2dnav_pr2)/move_base/navfn_params.yaml" command="load" />
  <rosparam file="$(find 2dnav_pr2)/move_base/base_local_planner_params.yaml" comm
d="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 "od 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_heigh
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="lo
d"/>
```

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ya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 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="$(find gazebo_w
lds)/Media/materials/textures/map3.png 0.1" respawn="true" machine="c1" />
  <include file="$(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_wor
s)/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*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**lau**文件的方法。

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来配置多台机器使一个master。

Tutorial Level: INTERMEDIATE

Next Tutorial: 自定义消息类型

目录

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*)

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: 在Pyhon中使用C++类

目录

catkin rosbuild

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。

在python中使用C++类

Description: 本教程阐述一种在python中使用C++类的方法。

Keywords: C++, Python, bindings

Tutorial Level: ADVANCED

Next Tutorial: 如何编写教程

catkin | rosbuild

目录

This tutorial illustrates a way to use a C++ class with ROS messages in Python. The Boost Python library is used. The difficulty is to translate Python objects of ROS messages written in pure Python into equivalent instances. This translation will be done through serialization/deserialization. The source files can be found at https://github.com/galou/python_bindings_tutorial.

Another solution is to use classical ROS services, the server written in C++ will be a wrapper around the C class and the client, C++ or Python, will call the service. The solution proposed here does not create a RO node, provided the class to be wrapped does not make use of `ros::NodeHandle`.

Another solution is to write a wrapper for all needed ROS messages and their dependencies. Some appare deprecated package proposed some solutions for the automation of this task: genpybindings and boost_python_ros.

Class without NodeHandle

Because roscpp is not initialized when calling `rospy.init_node`. `ros::NodeHandle` instances cannot be use the C++ class without generating an error. If the C++ does not make use of `ros::NodeHandle`, this is no iss though.

Creating the package and writing the C++ class

Create a package and create the C++ class for which we will want to make a Python binding. This class us

ROS messages as arguments and return type.

```
切换行号显示

1 catkin_create_pkg python_bindings_tutorial rospy roscpp std_msgs
2 cd python_bindings_tutorial/include/python_bindings_tutorial
3 touch add_two_ints.h
4 rosed python_bindings_tutorial add_two_ints.h
```

The content of `include/python_bindings_tutorial/add_two_ints.h` will be:

```
切换行号显示

1  #ifndef PYTHON_BINDINGS_TUTORIAL_ADD_TWO_INTS_H
2  #define PYTHON_BINDINGS_TUTORIAL_ADD_TWO_INTS_H
3
4  #include <std_msgs/Int64.h>
5
6  namespace python_bindings_tutorial {
7
8  class AddTwoInts
9  {
10   public:
11     std_msgs::Int64 add(const std_msgs::Int64& a, const std_msgs::Int64& b);
12 };
13
14 } // namespace python_bindings_tutorial
15
16 #endif // PYTHON_BINDINGS_TUTORIAL_ADD_TWO_INTS_H
17
```

Write the class implementation into .

```
roscd python_bindings_tutorial/src
touch add_two_ints.cpp
rosed python_bindings_tutorial add_two_ints.cpp
```

The content of `src/add_two_ints.cpp` will be:

```
切换行号显示

1  #include <python_bindings_tutorial/add_two_ints.h>
2
3  using namespace python_bindings_tutorial;
4
5  std_msgs::Int64 AddTwoInts::add(const std_msgs::Int64& a, const std_msgs::In
4& b)
6  {
7    std_msgs::Int64 sum;
8    sum.data = a.data + b.data;
9    return sum;
```

```
}
```

Binding, C++ part

The binding occurs through two wrapper classes, one in C++ and one in Python. The C++ wrapper translat input from serialized content to C++ message instances and output from C++ message instances into serialized content.

切换行号显示

```
roscd python_bindings_tutorial/src
touch add_two_ints_wrapper.cpp
rosed python_bindings_tutorial add_two_ints_wrapper.cpp
```

The content of `src/add_two_ints_wrapper.cpp` will be:

切换行号显示

```
#include <boost/python.hpp>

#include <string>

#include <ros/serialization.h>
#include <std_msgs/Int64.h>

#include <python_bindings_tutorial/add_two_ints.h>

/* Read a ROS message from a serialized string.
  */
template <typename M>
M from_python(const std::string str_msg)
{
  size_t serial_size = str_msg.size();
  boost::shared_array<uint8_t> buffer(new uint8_t[serial_size]);
  for (size_t i = 0; i < serial_size; ++i)
  {
    buffer[i] = str_msg[i];
  }
  ros::serialization::IStream stream(buffer.get(), serial_size);
  M msg;
  ros::serialization::Serializer<M>::read(stream, msg);
  return msg;
}

/* Write a ROS message into a serialized string.
*/
template <typename M>
std::string to_python(const M& msg)
{
  size_t serial_size = ros::serialization::serializationLength(msg);
```

```
  boost::shared_array<uint8_t>  buffer(new uint8_t[serial_size]);
  ros::serialization::OStream  stream(buffer.get(), serial_size);
  ros::serialization::serialize(stream, msg);
  std::string str_msg;
  str_msg.reserve(serial_size);
  for (size_t i = 0; i < serial_size; ++i)
  {
    str_msg.push_back(buffer[i]);
  }
  return str_msg;
}

class AddTwoIntsWrapper : public python_bindings_tutorial::AddTwoInts
{
  public:
    AddTwoIntsWrapper() : AddTwoInts() {}

    std::string add(const std::string& str_a, const std::string& str_b)
    {
      std_msgs::Int64 a = from_python<std_msgs::Int64>(str_a);
      std_msgs::Int64 b = from_python<std_msgs::Int64>(str_b);
      std_msgs::Int64 sum = AddTwoInts::add(a, b);

      return to_python(sum);
    }
};


BOOST_PYTHON_MODULE(_add_two_ints_wrapper_cpp)
{
  boost::python::class_<AddTwoIntsWrapper>("AddTwoIntsWrapper", boost::python::init<>())
    .def("add", &AddTwoIntsWrapper::add)
    ;
}
```

The line Error: No code_block found creates a Python module in the form of a dynamic library. The name o module will have to be the name of the exported library in `CMakeLists.txt`.

Binding, Python part

The Python wrapper translates input from Python message instances into serialized content and output fro serialized content to Python message instances. The translation from Python serialized content (`str`) into C serialized content (`std::string`) is built in the Boost Python library.

```
roscd python_bindings_tutorial/src
mkdir python_bindings_tutorial
roscd python_bindings_tutorial/src/python_bindings_tutorial
touch _add_two_ints_wrapper_py.py
rosed python_bindings_tutorial _add_two_ints_wrapper_py.py
```

The content of src/python_bindings_tutorial/_add_two_ints_wrapper_py.py will be

切换行号显示

```
from StringIO import StringIO

import rospy
from std_msgs.msg import Int64

from python_bindings_tutorial._add_two_ints_wrapper_cpp import AddTwoIntsWra
er


class AddTwoInts(object):
    def __init__(self):
        self._add_two_ints = AddTwoIntsWrapper()

    def _to_cpp(self, msg):
        """Return a serialized string from a ROS message

        Parameters
        ----------
        - msg: a ROS message instance.
        """
        buf = StringIO()
        msg.serialize(buf)
        return buf.getvalue()

    def _from_cpp(self, str_msg, cls):
        """Return a ROS message from a serialized string

        Parameters
        ----------
        - str_msg: str, serialized message
        - cls: ROS message class, e.g. sensor_msgs.msg.LaserScan.
        """
        msg = cls()
        return msg.deserialize(str_msg)

    def add(self, a, b):
        """Add two std_mgs/Int64 messages

        Return a std_msgs/Int64 instance.

        Parameters
        ----------
        - a: a std_msgs/Int64 instance.
        - b: a std_msgs/Int64 instance.
        """
        if not isinstance(a, Int64):
```

```
46              rospy.ROSException('Argument 1 is not a std_msgs/Int64')
47          if not isinstance(b, Int64):
48              rospy.ROSException('Argument 2 is not a std_msgs/Int64')
49          str_a = self._to_cpp(a)
50          str_b = self._to_cpp(b)
51          str_sum = self._add_two_ints.add(str_a, str_b)
52          return self._from_cpp(str_sum, Int64)
```

In order to be able to import the class as `python_bindings_tutorial.AddTwoInts`, we import the symbols in `_init_.py`. First, we create the file:

切换行号显示

```
1 roscd python_bindings_tutorial/src/python_bindings_tutorial
2 touch __init__.py
3 rosed python_bindings_tutorial __init__.py
```

The content of `src/python_bindings_tutorial/_init_.py` will be:

切换行号显示

```
1 from python_bindings_tutorial._add_two_ints_wrapper_py import AddTwoInts
```

Glueing everything together

Edit the `CMakeLists.txt` (`rosed python_bindings_tutorial CmakeLists.txt`) like this:

```
cmake_minimum_required(VERSION 2.8.3)
project(python_bindings_tutorial)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  roscpp_serialization
  std_msgs
)

## Both Boost.python and Python libs are required.
find_package(Boost REQUIRED COMPONENTS python)
find_package(PythonLibs 2.7 REQUIRED)


## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
catkin_python_setup()

###################################
## catkin specific configuration ##
###################################
```

```
catkin_package(
        INCLUDE_DIRS include
        LIBRARIES add_two_ints _add_two_ints_wrapper_cpp
        CATKIN_DEPENDS roscpp
        #  DEPENDS system_lib
)

###########
## Build ##
###########

# include Boost and Python.
include_directories(
        include
        ${catkin_INCLUDE_DIRS}
        ${Boost_INCLUDE_DIRS}
        ${PYTHON_INCLUDE_DIRS}
        )

## Declare a cpp library
add_library(add_two_ints src/add_two_ints.cpp)
add_library(_add_two_ints_wrapper_cpp src/add_two_ints_wrapper.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(add_two_ints ${catkin_LIBRARIES})
target_link_libraries(_add_two_ints_wrapper_cpp add_two_ints ${catkin_LIBRARIES} $
oost_LIBRARIES})

# Don't prepend wrapper library name with lib and add to Python libs.
set_target_properties(_add_two_ints_wrapper_cpp PROPERTIES
        PREFIX ""
        LIBRARY_OUTPUT_DIRECTORY ${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE_PYTHON_DE
INATION}
        )
```

The c++ wrapper library should be have the same name as the Python module. If the target name needs to different for a reason, the library name can be specified with`set_target_properties(_add_two_ints_wrapper_cpp PROPERTIES OUTPUT_NAME correct_library_n`

The line

```
catkin_python_setup()
```

is used to export the Python module and is associated with the file `setup.py`

```
切换行号显示

1 roscd python_bindings_tutorial
2 touch setup.py
```

The content of `setup.py` will be:

```
切换行号显示

1 # ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD
2
3 from distutils.core import setup
4 from catkin_pkg.python_setup import generate_distutils_setup
5
6 # fetch values from package.xml
7 setup_args = generate_distutils_setup(
8     packages=['python_bindings_tutorial'],
9     package_dir={'': 'src'})
10
11 setup(**setup_args)
```

We then build the package with `catkin_make`.

Testing the binding

You can now use the binding in Python scripts or in a Python shell

```
切换行号显示

1 from std_msgs.msg import Int64
2 from python_bindings_tutorial import AddTwoInts
3 a = Int64(4)
4 b = Int64(2)
5 addtwoints = AddTwoInts()
6 sum = addtwoints.add(a, b)
7 sum
```

Class with NodeHandle

As stated, a Python call to `rospy.init_node` does not initialize roscpp. If roscpp is not initialized, instanciating`ros::NodeHandle` will lead to a fatal error. A solution for this is provided by the moveit_ros_planning_interface. In any Python script that uses the wrapped class, two lines need to be added before instanciating `AddTwoInts`:

```
切换行号显示

1 from moveit_ros_planning_interface._moveit_roscpp_initializer import roscpp_
it
2 roscpp_init('node_name', [])
```

This will create a ROS node. The advantage of this method over a classical ROS service server/client implementation is thus not as clear as in the case without the need of `ros::NodeHandle`.

Class with container of ROS messages

If the class uses containers of ROS messages, an extra conversion step must be added. This step is not specific to ROS but is part of the Boost Python library.

`std::vector<M>` as return type

In the file where the C++ wrapper class is defined, add these lines:

切换行号显示

```
1  // Extracted from https://gist.github.com/avli/b0bf77449b090b768663.
2  template<class T>
3  struct vector_to_python
4  {
5    static PyObject* convert(const std::vector<T>& vec)
6    {
7      boost::python::list* l = new boost::python::list();
8      for(std::size_t i = 0; i < vec.size(); i++)
9        (*l).append(vec[i]);
10
11     return l->ptr();
12   }
13 };
14
15 class Wrapper : public WrappedClass
16 {
17 /*
18 ...
19 */
20     std::vector<std::string> wrapper_fun(const std::string str_msg)
21     {
22       /* ... */
23     }
24
25 };
26
27 BOOST_PYTHON_MODULE(module_wrapper_cpp)
28 {
29   boost::python::class_<Wrapper>("Wrapper", bp::init</* ... */>())
30     .def("fun", &Wrapper::wrapper_fun);
31
32   boost::python::to_python_converter<std::vector<std::string, std::allocator
td::string> >, vector_to_python<std::string> >();
33 }
```

`std::vector<M>` as argument type

Cf. Boost.Python documentation.

如何编写教程

Description: （概述：）本教程介绍在编辑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

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（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TutorialTemp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 completed 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 tuto*
（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 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 tutoria 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：

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

显示效果:

切换行号显示

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

latex

示例:

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

显示效果:

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

python

示例:

```
{{{
#!python
#!/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

```
1 #!/usr/bin/env python
2 import roslib; roslib.load_manifest('beginner_tutorials')
3 import rospy
4 from std_msgs.msg import String
5 def talker():
6     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. 如何编写教程

   （概述：）本教程介绍在编辑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の便利なテンプレートとマクロを押さえます。
4. 튜토리얼 작성하기

   이 튜토리얼은 튜토리얼 작성을 위한 튜토리얼입니다.

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

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

ROS developer's guide

目录

ROS is a large system, with lots of people writing lots of code. To keep things manageable, we have established the following development guidelines. Please follow these guidelines as you code.

See also:

- Quality assurance process
- ROS C++ Style Guide
- ROS Python Style Guide
- ROS Javascript Style Guide
- Naming Conventions
- CommonProcedures

Source control

We support using Git, Mercurial, Subversion and Bazaar for source control. As the ROS community is distributed, you are welcome to host your code anywhere that is publicly accessible (GitHub, Bitbucket, Google Code). The main ROS code base is hosted in several organization units on github.com. For the recommended repository usage please seeRecommendedRepositoryUsage

- Add to source control only the minimal set of manually written source and build files that are necess to build your package. Don't add machine-generated files, such as objects (.o), libraries (.a, .so, .dll), auto-generated configure scripts.
    - **svn add** on a directory will recursively add all of the directory's contents. Do a **make clean** be the **svn add**.
    - Don't add large binary files: upload them to a web server and download them in your build file
- Commit your code early and often. Anything not in source control should be considered temporary "scratch" storage.
- Try to keep commits focused on a particular change instead of lumping multiple changes together --

much easier to rollback that way.
- Give an informative message with each commit.
- Don't break the build. Verifies that your code compiles before checking it in.

Bug tracking

We use separate bug trackers for each package which include bug reports, enhancement requests, and ta assignments. Usually you will find the link to the package-specific bug tracker on the Wiki page of the package.

For packages hosted on GitHub this is commonly the issue tracker of the repository.

The maintainer will assign a milestone to each reported issue. This should give the reporter a reasonable feedback when this issue will be addressed. These milestone are usually ROS releases (like Groovy or Hyd or more fine grain milestones (like Hydro beta 1). Furthermore the milestone **untargeted** is used when a iss will likely not get fixed / implemented. That can be either due to lack of time of the developer or other viabil considerations.

To give users some idea as to whether a fix has made it to the release repository, the maintainer should eit mention the intended release version while closing the issue report, or setup milestones for every minor rel and tag the issue report with the next milestone. This will make the issue self-contained in determining whe a fixed release is available to the user.

When you find a bug, open a ticket. When you want a feature, open a ticket. Emails or posting on answers.ros.org or the mailing list are more likely to get lost. It's far less likely to be forgotten if there's a tic

- Be as descriptive as possible in tickets.
    - They should include instructions for reproducing the bug.
    - They should mention information your system and state (what version of related packages are used etc.).
- Don't be afraid to open tickets. Many developers open tickets assigned to themselves, using Trac as kind of external memory.

If you are unsure for which package the issue should be filled or if an issue you are facing is actually a bug please ask on answers.ros.org first.

Code layout

ROS code is organized into packages, and packages can be collected into a single repository. Packages a represent units of code that you build

Especially if using GitHub it is recommended to create a README.md at the root of your repository to expl to users what to find in the repository. It is recommended to link to the package documentation on the RO wiki for the contained packages. See this article for formatting help.

Packaging

The ROS package and build system relies on manifest.xml files.

- Every package must have a manifest.xml file, located in the package's top directory.
- At a minimum, the manifest file must contain:
    - description
    - author
    - license

Here is a template you can fill in for roscpp nodes:

```
<package>
  <description brief="BRIEF DESCRIPTION">
     LONGER DESCRIPTION
  </description>
  <author>You/you@willowgarage.com</author>
  <license>BSD</license>
  <url>http://www.ros.org/wiki/YOURPACKAGE</url>
  <depend package="roscpp"/>
</package>
```

And here is one for rospy nodes:

```
<package>
  <description brief="BRIEF DESCRIPTION">
     LONGER DESCRIPTION
  </description>
  <author>You/you@willowgarage.com</author>
  <license>BSD</license>
  <url>http://www.ros.org/wiki/YOURPACKAGE</url>
  <depend package="rospy"/>
</package>
```

GUI toolkits

We have migrated all new GUI development to rqt, a Qt-based GUI framework for ROS. Much of the existi code that were built before fuerte used wxWidgets, which has not maintained good cross-platform compatibility. Please consider using rqt for any new GUI development. Development instruction (including license consideration when writing in `python`) is available there.

Building

The basic build tool is CMake (more).

- Every package that has a build step must have a CMakeLists.txt file in the package's top directory.
- For now, every package that has a build step must also have a Makefile**, but it should be very shor**
- Packages that don't have build steps don't need any build files.

Licensing

ROS is an Open Source project. We aim to support a wide variety of users and developers, from grad stud to entrepreneurs.

- We prefer permissive Open Source licenses that facilitate commercial use of the code.
- The preferred license for the project is the BSD license. Whenever possible, new code should be licensed under BSD. All ROS core code is licensed BSD. Reasons for choosing BSD Any
- OSI-approved license is acceptable (for non-core code).
- We strongly recommend using a non-copyleft license (e.g. BSD) for ROS `.msg` and `.srv` files so that auto-generated source files and data structures are not encumbered.
- The full text statements of all licenses used in the project should be placed in the LICENSES director the top of the repository. If you add a package under a license that is not included in LICENSES, add license statement.
- Every source file should contain a commented license summary at the top. For convenience, the LICENSES directory also contains summaries, appropriately commented for different languages.
    - We preserve license and copyright statements on all third-party code that we redistribute.
- We rigorously obey license terms on third-party software that we use. For example:
    - If a library is licensed GPL or LGPL and you modify it, you must release the modified code. Ide you would send a patch to the library maintainer. Keeping the modified code in a publicly accessible repository (e.g., at SourceForge) also suffices.
    - If your package uses a GPL'd library, then your package code must also be licensed GPL.
    - If your package uses a GPL'd library, then it must not also use any code governed by a GPL-incompatible license. For example, the Creative Commons Attribution-Noncommercial-Share Alike license is GPL-incompatible, because it imposes extra constraints that the GPL does no (namely, requiring attribution, and prohibiting commercial use).
- Whenever possible, each ROS package is governed by a single license.
    - Common special case: BSD-licensed code (e.g., the ROS core) is used in a package that also uses GPL-licensed code. To comply with the GPL, the BSD-licensed code is multiply-licensed under BSD and GPL, with the user given the choice of which license to use. This is not a big d because the GPL simply adds restrictions that are not in BSD.
- The ROS packaging and communication system allows for fine-grained licensing. Because nodes communicate via ROS messages, code from multiple nodes is not linked together. Thus the packag provides a kind of "license boundary."
    - Exception: when a package is a library that is actually linked to by other packages. In this situation, license terms mix in the resulting binary.
- For reference: Maintaining Permissive-Licensed Files in a GPL-Licensed Project: Guidelines for Developers

Copyright

Under the Berne Convention, the author of a work automatically holds copyright, with or without a formal statement to that effect. However, making copyright explicit is helpful in long-term project management.

- Each source file should contain a commented copyright line at the top, e.g.: Copyright 2008 Jim Bob **This statement is usually directly above the license summary.**

- If you work for yourself, then you own the copyright.
- If you work for someone else (e.g. Willow Garage), then your employer owns the copyright.

Debugging

All standard debugging tools work within ROS, including but not limited to:

- GDB
- Oprofile
- Valgrind

General advice:

- If a program, say foo**, crashes, first try running it inside GDB:**
  - You can supply command-line arguments if necessary:
  - ```
    (gdb) run arg1 arg2 arg3
    ```
  - When the program crashes, use gdb's bt **command to get a backtrace and start digging.**
- For tough bugs, especially those related to memory corruption, trying running inside valgrind:
  - ```
    valgrind -v foo arg1 arg2 arg3
    ```
  - Valgrind will track all memory access and generally can identify the cause of the problem. It w also likely find other problems that you didn't know about yet. One caveat is that valgrind will make your program run slower, sometimes much *slower*.
- master and rospy logs are always created in `ROS_ROOT/log`. However, roscpp client node logs are no created by default. If you are debugging a nasty node, you can "or" in the WRITE_LOG_FILE consta the (optional) second parameter of the constructor (along with other fancy options like ANONYMOUS_NAME and such). Alternatively, if you don't want to recompile the node, you can give command-line parameter of log:=BLAHBLAH where BLAHBLAH is anything at all (or nothing).

Testing

We use two level of testing:

- Library: At the library level, we use standard unit-test frameworks. In C++, we use gtest. In Python, w use unittest.
- Message: At the message level, we use rostest to set up a system of ROS nodes, run a test node, th tear down the system.

We have established best practices and policies for writing and running tests.

If you are developing in the ros, ros-pkg, or wg-ros-pkg repositories, a build farm is set up to regularly test build and run automated tests on a variety of architectures. If the build or tests stop working after one of yo commits, you will get an email informing you of the error and will be expected to fix it. See the AutomatedTesting guidelines for more details.

Documentation

- All code should be documented according to QAProcess. Including:
  - All externally visible code-level APIs must be documented
  - All externally visible ROS-level APIs (topics, services, parameters) must be documented.

Releasing

The standard process for releasing code to the ROS community is described on the release page.

Standardization

- Code should use ROS services, follow guidelines for their use
  - use rosout for printing messages
  - uses the ROS Clock for time-based routines

Deprecation

As soon as there are users of your code, you have a responsibility not to pull the rug out from under them sudden breaking changes. Instead, use a process of **deprecation,** which means marking a feature or component as being no longer supported, with a schedule for its removal. Give users time to adapt, which usually one release cycle, then do the removal.

Deprecation can happen at multiple levels, including:

- **API features** : Say you want to remove a method call from a library. First mark it as deprecated in th API documentation; with DOxgyen, use @deprecated. If the language supports it, also mark the cod being deprecated; in C/C++, use __attribute__((deprecated)). In the next release, note the deprecati the ChangeList, with the future release at which you expect to remove it; if it's a widely used feature, make the deprecation notice prominent, and explain the reasoning behind it. In that future release, remove it.
- **Packages :** Say you want to remove a package. Mark it as deprecated in the Wiki documentation (e putDEPRECATED at the top), with a note as to when you expect to remove it. Include notice of the deprecation in the ChangeList with the next (stack) release. If it's a widely used package, you should also send mail to your users giving them as much advance warning as possible.

Large data files, Large test files

Large files (anything over 1MB, really) often don't belong in the *-ros-pkg repositories, especially if they are used for unit tests. These large files affect the time that it takes to checkout the repository, whether or not someone is building your package.

Large data files should be hosted in a public web hosting site. You can also just place the file you need on web server. Hosting for some files can be provided at download.ros.org. Please contact ros-release@lists.ros.org for more info. You're encouraged to search around if there are files that satisfy your n

before you'll open a upload request.

To download this file for building, use the catkin_download_test_data. OR if you're in older rosbuild era,`rosbuild_download_test_data(URL MD5SUM)` macro. For example:

```
catkin_download_test_data(
    ${PROJECT_NAME}_saloon.bag
    http://downloads.foo.com/bags/saloon.bag
    DESTINATION ${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE_SHARE_DESTINATION}/test
    MD5 01603ce158575da859b8afff5b676bf9)

rosbuild_download_test_data(http://code.ros.org/svn/data/robot_pose_ekf/zero_covar
nce.bag test/zero_covariance.bag 0a51b4f5001f446e8466bf7cc946fb86)
```

[REP Index] [REP Source]

REP:	103
Title:	Standard Units of Measure and Coordinate Conventions
Author:	Tully Foote, Mike Purvis
Status:	Active
Type:	Informational
Content-Type:	text/x-rst
Created:	07-Oct-2010
Post-History:	31-Dec-2014

Contents

Abstract

This REP provides a reference for the units and coordinate conventions used within ROS.

Rationale

Inconsistency in units and conventions is a common source of integration issues for developers and can also lead to software bugs. It can also create unnecessary computation due to data conversion. This REP documents the standard conventions for ROS in order to lessen these issues.

Exceptions

The scope of potential robotics software is too broad to require all ROS software to follow the guidelines of this REP. However, choosing different conventions should be well justified and well documented.

For example, there are domains where the default conventions are not appropriate. Interstellar lengths are not appropriately measured in meters, and space-oriented libraries may wish to choose a different convention. There are other exceptions that different domains may wish to address.

Units

We have chosen to standardize on SI units. These units are the most consistent international standard. SI units are maintained by Bureau International des Poids et Mesures. [1] There is good documentation on Wikipedia for International System Of Units [2]

Base Units

These are the base units which are commonly used

Quantity	Unit

length	meter

mass	kilogram
time	second
current	ampere

Derived Units

SI defines seven base units and many derived units. If you are not using SI base units, you should use SI-derived units.

Good documentation can be found on Wikipedia about SI derived units [3]

Commonly used SI-derived units in ROS are:

Quantity	Unit
angle	radian
frequency	hertz
force	newton
power	watt
voltage	volt
temperature	celsius
magnetism	tesla

Coordinate Frame Conventions

All coordinate frames should follow these conventions.

Chirality

All systems are right handed. This means they comply with the right hand rule [4].

Axis Orientation

In relation to a body the standard is:

- x forward
- y left
- z up

For short-range Cartesian representations of geographic locations, use the east north up [5] (ENU) convention:

- X east
- Y north
- Z up

To avoid precision problems with large float32 values, it is recommended to choose a nearby origin such as your system's starting position.

Suffix Frames

In the case of cameras, there is often a second frame defined with a "_optical" suffix. This uses a slightly different convention:

- z forward
- x right
- y down

For outdoor systems where it is desireable to work under the north east down [6] (NED) convention, define an appropriately transformed secondary frame with the "_ned" suffix:

- X north

- Y east
- Z down

Rotation Representation

There are many ways to represent rotations. The preferred order is listed below, along with rationale.

1. quaternion
    - Compact representation
    - No singularities
2. rotation matrix
    - No singularities
3. fixed axis roll, pitch, yaw about X, Y, Z axes respectively
    - No ambiguity on order
    - Used for angular velocities
4. euler angles yaw, pitch, and roll about Z, Y, X axes respectively
    - Euler angles are generally discouraged due to having 24 'valid' conventions with different domains using different conventions by default.

By the right hand rule, the yaw component of orientation increases as the child frame rotates counter-clockwise, and for geographic poses, yaw is zero when pointing east.

This requires special mention only because it differs from a traditional compass bearing, which is zero when pointing north and increments clockwise. Hardware drivers should make the appropriate transformations before publishing standard ROS messages.

Covariance Representation

Linear

```
float64[9] linear_acceleration_covariance # 3x3 row major matrix in x,
y, z order
```

Angular

```
float64[9] angular_velocity_covariance # 3x3 row major matrix about x,
y, z order with fixed axes
```

Six Dimensional

```
# Row-major representation of the 6x6 covariance  matrix
# The orientation parameters use a fixed-axis representation.
# In order, the parameters are:
# (x, y, z, rotation about X axis, rotation about Y axis, rotation about
Z axis)
float64[36] covariance
```

References

[1] Bureau International des Poids et Mesures (http://www.bipm.org/en/home/)

[2] http://en.wikipedia.org/wiki/International_System_of_Units

[3] http://en.wikipedia.org/wiki/SI_derived_units

[4] http://en.wikipedia.org/wiki/Right-hand_rule

[5] http://en.wikipedia.org/wiki/Geodetic_datum#Local_east.2C_north.2C_up_.28ENU.29_coordinates

[6] http://en.wikipedia.org/wiki/North_east_down

学习 URDF (Unified Robot Description Format，统一的机器人描述文件格式)

1. Create your own urdf file

   In this tutorial you start creating your own urdf robot description file.
2. Parse a urdf file

   This tutorial teaches you how to use the urdf parser
3. Using the robot state publisher on your own robot

   This tutorial explains how you can publish the state of your robot to tf, using the robot state publishe
4. Start using the KDL parser

   This tutorial teaches you how to create a KDL Tree from a URDF file
5. Using urdf with robot_state_publisher

   This tutorial gives a full example of a robot model with URDF that uses robot_state_publisher. First, create the URDF model with all the necessary parts. Then we write a node which publishes the JointState and transforms. Finally, we run all the parts together.

手把手学习URDF

1. Building a Visual Robot Model with URDF from Scratch

   Learn how to build a visual model of a robot that you can view in Rviz
2. Building a Movable Robot Model with URDF

   Learn how to define movable joints in URDF
3. Adding Physical and Collision Properties to a URDF Model

   Learn how to add collision and inertial properties to links, and how to add joint dynamics to joints.
4. Using Xacro to Clean Up a URDF File

   Learn some tricks to reduce the amount of code in a URDF file using Xacro

实例讲解一个完整的URDF文件

1. Understanding the PR2 Robot Description

   This tutorial explains the layout of the top level URDF Xacro file for a complex robot such as PR2.

ros_control

hydro | indigo | Documentation Status

Package Links
Tutorials
FAQ
Changelog
Change List
Reviews

Dependencies (10)
Used by (1)
Jenkins jobs (7)

Package Summary

Released | Continuous integration | Documented

A set of packages that include controller interfaces, controller managers, transmissions, hardware_interfaces and the control_toolbox.

- Maintainer status: developed
- Maintainer: Adolfo Rodriguez Tsouroukdissian <adolfo.rodriguez AT pal-robotics DOT com>, Dave Coleman <davetcoleman AT gmail DOT com>
- Author: Wim Meeussen
- License: BSD
- Bug / feature tracker: https://github.com/ros-controls/ros_control/issues
- Source: git https://github.com/ros-controls/ros_control.git (branch: indigo-devel)

目录

Overview

The ros_control packages are a rewrite of the pr2_mechanism packages to make controllers generic to all robots beyond just the PR2.

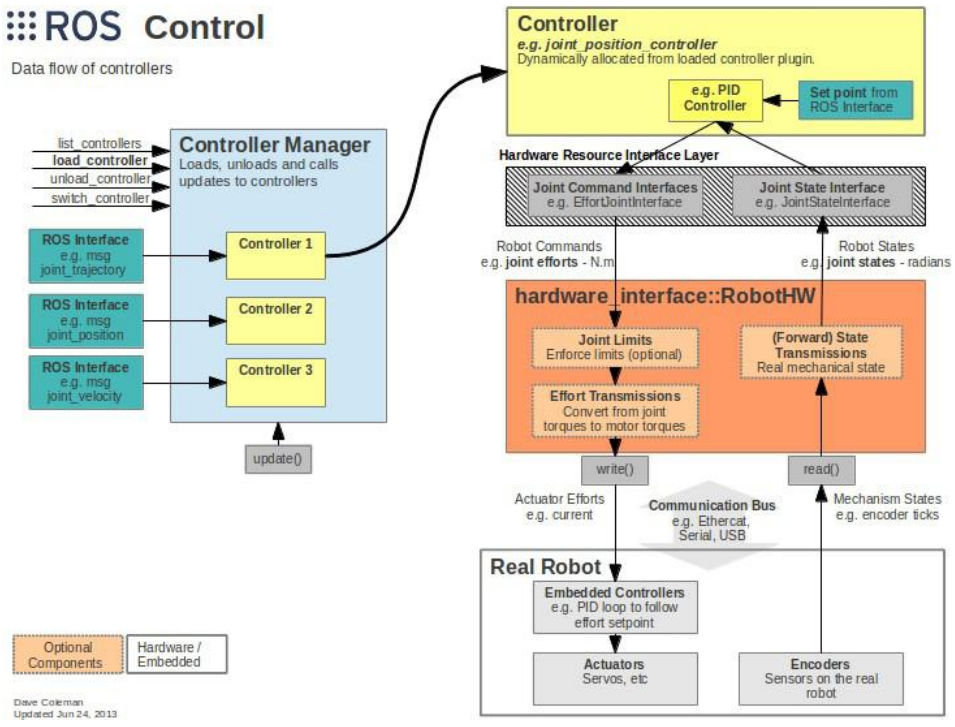


Diagram source in ros_control/documentation

The ros_control packages takes as input the joint state data from your robot's actuator's encoders and an input set point. It uses a generic control loop feedback mechanism, typically a PID controller, to control the output, typically effort, sent to your actuators. ros_control gets more complicated for physical mechanisms that do not have one-to-one mappings of joint positions, efforts, etc but these scenarios are accounted for using transmissions.

A high-level overview of the project can be found in the ROScon 2014 talk entitled *ros_control: An overview* (slides, video).

Additional documentation is available at the Github Wiki

Controllers

A list of available controller plugins, contained in ros_controllers, as of this writing. You can of course create your own and are not limited to this list:

- effort_controllers
  - joint_effort_controller
  - joint_position_controller
  - joint_velocity_controller
- joint_state_controller
  - joint_state_controller
- position_controllers
  - joint_position_controller
- velocity_controllers
  - joint_velocity_controllers

Hardware Interfaces

A list of available hardware interfaces (via the Hardware Resource Manager) as of this writing. You can of course create your own and are not limited to this list:

- Joint Command Interfaces
  - Effort Joint Interface
  - Velocity Joint Interface
  - Position Joint Interface
- Joint State Interfaces
- Actuator State Interfaces
- Actuator Command Interfaces
  - Effort Actuator Interface
  - Velocity Actuator Interface
  - Position Actuator Interface
- Force-torque sensor Interface
- IMU sensor Interface

Transmissions

A transmission is an element in your control pipeline that transforms efforts/flow variables such that their product - power - remains constant. A transmission interface implementation maps effort/flow variables to output effort/flow variables while preserving power.

Mechanical transmissions are power-preserving transformations, ie.

```
P_in         = P_out

F_in x V_in = F_out x V_out
```

where P, F and V stand for power, force and velocity. More generally, power is the product of an effort (eg. force, voltage) and a flow (eg. velocity, current) variable. For a simple mechanical reducer with ratio n, one has:

```
effort map: F_joint = F_actuator *  n

flow map:   V_joint = V_actuator /  n
```

From the above it can be seen that power remains constant between input and output. Complementary Wikipedia link(first part will do).

Transmission URDF Format

See URDF Transmissions.

Transmission Interfaces

Transmission-specific code (not robot-specific) implementing bidirectional (actuator <-> joint) effort and flow maps under a uniform interface shared across transmission types. This is hardware-interface-agnostic. A list of available transmission types as of this writing:

- Simple Reduction Transmission
- Differential Transmission
- Four Bar Linkage Transmission

Usages:

- transmission_interface::ActuatorToJointStateInterface to populate joint state from actuator variables.
- hardware_interface::JointStateInterface to expose the joint state to controllers.

Example

See here

Joint Limits

The **joint_limits_interface** contains data structures for representing joint limits, methods for populating them from common formats such as URDF and rosparam, and methods for enforcing limits on different kinds of joint commands.

The joint_limits_interface is not used by controllers themselves (it does not implement a *HardwareInterface*) but instead operates after the controllers have updated, in the *write()* method (or equivalent) of the robot abstraction. Enforcing limits will **overwrite** the commands set by the controllers, it does not operate on a separate raw data buffer.

Specification

- **Joint limits** Position, velocity, acceleration, jerk and effort.
- **Soft joint limits** Soft position limits, k_p, k_v (as described in pr2_controller_manager/safety_limits).
- Utility method for loading **joint limits** information from **URDF** (only position, velocity, effort)
- Utility method for loading **soft joint limits** information from **URDF**
- Utility method for loading **joint limits** from **ROS parameter server** (all values). Parameter specification is the same used in MoveIt, with the addition that we also parse jerk and effort limits.

Joint limits interface

- *ros_control* interface for enforcing joint limits.
- For *effort-controlled* joints, *position-controlled* joints, and *velocity-controlled* joints, two types of interfaces have been created. The first is a saturation interface, used for joints that have normal limits but not soft limits. The second is an interface that implements soft limits, similar to the one used on the PR2.

Example

See here

Examples

- Barrett WAM controllers at Johns Hopkins University: barrett_control on Github
- RRBot in Gazebo: ros_control with Gazebo tutorial
- Rethink Baxter hardware as used at the University of Colorado Boulder: Baxter on Github

Install

On Ubuntu, you can install ros_control from debian packages (recommended):

```
sudo apt-get install ros-indigo-ros-control ros-indigo-ros-controllers
```

Or on Ubuntu and other platforms from source. To ease installing from source a rosinstall file is provided:

```
cd CATKIN_WORKSPACE/src
wstool init
wstool merge https://raw.github.com/ros-controls/ros_control/indigo-devel/ros_control.rosinsta
ll
wstool update
cd ..
rosdep install --from-paths . --ignore-src --rosdistro indigo -y
catkin_make
```

Ideas and future perspectives

Not exactly a roadmap, but this page contains discussion and proposed solutions to allow ros_control to better accommodate more complex control setups and address shortcomings in the current implementation.

A ROS Control SIG exists with a mailing list for discussing ros_control issues and features. You are encouraged to join and help with ros_control's development!

飞拓科技

TEL:15255609545

邮箱：ftrobot@aliyun.com

技术交流群：707838113