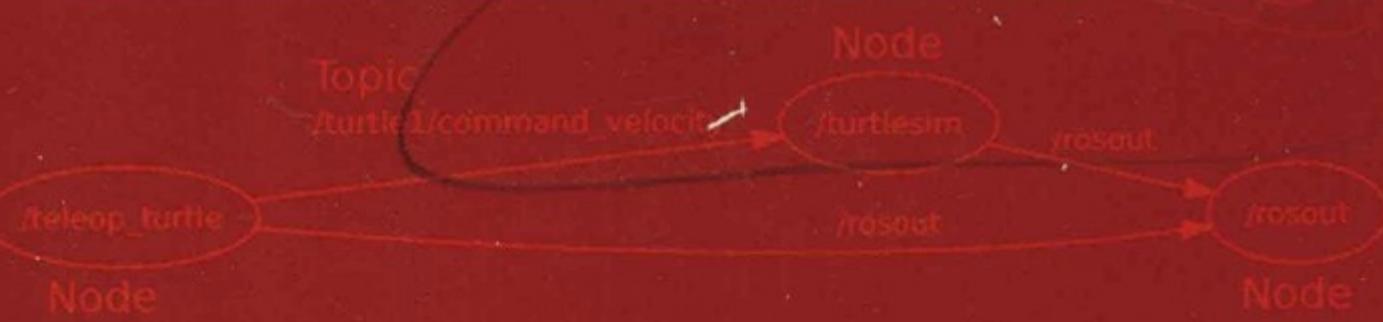


# 开源机器人操作系统

## — ROS

◎ 张建伟 张立伟 胡颖 张俊 编著



科学出版社

(TP-6033.0101)

科学出版社 工程技术分社  
电话: (010) 64019417  
Email: gcjs@mail.sciencep.com

销售分类建议: 计算机/机器人

科学出版社  
[www.sciencep.com](http://www.sciencep.com)

ISBN 978-7-03-035434-1



9 787030 354341 >

定价: 88.00元(含光盘)

# 开源机器人操作系统—ROS

张建伟 张立伟 胡 颖 张 俊 编著

科学出版社

北京



## 内 容 简 介

目前,ROS(robot operating system)逐步成为机器人研发领域的通用性软件平台。本书是国内第一本全面介绍 ROS 的中文版图书。

ROS 是开源的用于机器人的一种后操作系统,或者说次级操作系统。它提供类似操作系统所提供的功能,包含硬件抽象描述、底层驱动程序管理、共用功能的执行、程序间的消息传递、程序发行包管理,它也提供一些工具程序和库用于获取、建立、编写和运行多机整合的程序。

本书附光盘一张,内容包括书中的部分例子源代码和 Diamondback 及 Electric 版本安装后在本地硬盘上的全部程序,以便于读者对照自己的安装版本进行调试。

本书可作为机器人研究者以及机器人爱好者应用 ROS 构建机器人软件系统的参考手册。

### 图书在版编目(CIP)数据

开源机器人操作系统:ROS/张建伟等编著. —北京:科学出版社, 2012

ISBN 978-7-03-035434-1

I. ①开… II. ①张… III. ①机器人—操作系统 IV. ①TP242

中国版本图书馆 CIP 数据核字 (2012) 第 201958 号

责任编辑: 刘宝莉 / 责任校对: 宋玲玲

责任印制: 张倩 / 封面设计: 陈敬

科学出版社出版

北京东黄城根北街 16 号

邮政编码: 100717

<http://www.sciencep.com>

源海印刷有限责任公司印刷

科学出版社发行 各地新华书店经销

\*

2012 年 9 月第 一 版 开本: B5(720 × 1000)

2012 年 9 月第一次印刷 印张: 19 3/4 彩插: 4

字数: 417 000

定价: 88.00 元(含光盘)

(如有印装质量问题, 我社负责调换)



# 前　　言

近年来，机器研发领域对代码复用和模块化的需求越来越强烈，而已有的开源机器人系统又不能很好地适应该需求。2010 年 Willow Garage 公司发布了开源机器人操作系统 ROS (robot operating system)，由于其具有点对点设计、不依赖编程语言、开源等优点，很快在机器人研究领域展开了学习和使用 ROS 的热潮。

笔者所在的中国科学院深圳先进技术研究院认知技术研究中心致力于服务机器人的研发，在其面世伊始就开始使用 ROS，目前已经近两年。本书是根据研究组人员在学习和使用 ROS 过程中遇到的问题和经验编写而成，相信对广大同行和机器人爱好者会有所裨益。

ROS 是开源机器人操作系统，遵循 BSD 协议，对用户和商业应用免费。

本书是国内第一本全面介绍 ROS 的中文版图书。全书内容共分九章，覆盖了 ROS 基本编程的大部分内容。利用现有的因特网上最新资料为蓝本，深入浅出地介绍了 ROS 的总体架构和涉及的主要领域，全面地对 ROS 安装及使用过程中常见问题给出了解答。

书中第一、二章由张建伟执笔；第三、七、八章由张立伟执笔；第三、五章由胡颖执笔；第六章由张俊、胡颖执笔；第九章由张俊执笔。最后由张建伟统稿。其中，刘会芬和陈楠提供了部分实验演示代码。

全书结构清晰、合理，语言浅显易懂。对广大从事机器人、人工智能、计算机视觉等相关领域的科研人员和研究生及高年级本科生等，不失为一本重要的自学、教学参考书。

本书相关研究获得中国科学院知识创新工程重要方向项目“下一代自主机器人的关键技术——基于经验的记忆学习”(项目编号：KGCXZ-YW-128)以及国家自然科学基金项目(项目编号：61105130、61105132、61175124、61210013)的资助。

由于作者水平有限，书中难免存在不足之处，恳请广大读者和同行批评指正。

## 术语列表

英文术语	中文术语
ROS	机器人操作系统
Package	功能包
Stack	功能包集
Message	消息
Service	服务
Topic	主题
Node	节点
Master	节点管理器
Parameter Server	参数服务器
Bag	消息记录包
Publisher/Subscriber	主题发布者/订阅者
Launch	启动
Talker/Listener	消息发布者/接收者
Filesystem Level	文件系统级
Computation Graph Level	计算图级
Community Level	社区级
Package Resource Name	功能包源名称
Graph Resource Name	计算图源名称
Client Library	客户端库
Distribution	发行版
Repositories	软件版本仓库
Namespace	命名空间
Base Name	基本名称
Global Name	全局名称
Private Name	私有名称
Relative Name	相对名称
NodeHandle	节点句柄
Timer	计时器
Transform	变换

# 目 录

前言

术语列表

<b>第一章 ROS 简介</b>	1
1.1 ROS 简介	1
1.2 ROS 安装	4
1.3 ROS 支持的机器人	6
1.4 ROS 网上资源	6
<b>第二章 ROS 总体框架及基本命令</b>	7
2.1 ROS 总体框架	7
2.1.1 文件系统级	7
2.1.2 计算图级	9
2.1.3 社区级	11
2.1.4 更高层概念	12
2.1.5 名称	12
2.2 ROS 基本命令	15
2.2.1 ROS 文件系统命令	15
2.2.2 ROS 核心命令	26
2.3 工具	35
2.3.1 3D 可视化工具: rviz	35
2.3.2 传感器数据记录与可视化工具: rosbag 和 rxbag	45
2.3.3 画图工具: rxplot	47
2.3.4 系统可视化工具: rxgraph	49
2.3.5 rxconsole	49
2.3.6 tf 命令	51
2.4 例子	52
2.4.1 创建 ROS 消息和服务	52

2.4.2 记录和回放数据 .....	54
2.4.3 手工创建 ROS 功能包 .....	59
2.4.4 大项目上运行 roslaunch .....	60
2.4.5 在多台机器上运行 ROS 系统 .....	66
2.4.6 定义客户消息 .....	68
<b>第三章 ROS 客户端库 .....</b>	<b>70</b>
3.1 概述 .....	70
3.2 roscpp 客户端库 .....	71
3.2.1 简单的主题发布者和主题订阅者 .....	72
3.2.2 简单的服务器端和客户端 .....	78
3.2.3 roscpp 中参数的使用 .....	81
3.2.4 从节点句柄存取私有名称 .....	83
3.2.5 用类方法订阅和回调服务 .....	84
3.2.6 计时器 .....	85
3.2.7 带动态可重配置及参数服务器的主题发布者/订阅者节点 (C++) .....	87
3.2.8 带动态可重配置及参数服务器的主题发布者/订阅者节点 (Python) .....	95
3.2.9 组合 C++/Python 主题发布者/订阅者节点 .....	101
3.3 rospy 客户端库 .....	101
3.3.1 简单的主题发布者/订阅者 .....	101
3.3.2 简单的服务端和客户端 .....	105
3.3.3 rospy 中参数的使用 .....	108
3.3.4 rospy 中 numpy 的使用 .....	109
3.3.5 rospy 运行日志 .....	113
3.3.6 ROS Python Makefile 文件 .....	115
3.3.7 设置 PYTHONPATH .....	116
3.3.8 发布消息 .....	116
3.4 roslisp 客户端库 .....	117
3.5 实验阶段的客户端库 .....	117
3.5.1 rosjava .....	117
3.5.2 roslua .....	117

<b>第四章 OpenCV .....</b>	119
4.1 image_common 功能包集 .....	119
4.1.1 image_transport 功能包 .....	119
4.1.2 camera_calibration_parsers 功能包 .....	124
4.1.3 camera_info_manager 功能包 .....	126
4.1.4 polled_camera 功能包 .....	127
4.2 image_pipeline 功能包集 .....	127
4.3 vision_opencv 功能包集 .....	128
4.3.1 opencv2 .....	128
4.3.2 cv_bridge .....	128
4.3.3 image_geometry .....	139
4.4 投影 tf 坐标系到图像 (C++) .....	140
4.5 演示例子 .....	145
4.5.1 使用颜色追踪物体 .....	145
4.5.2 识别物体 .....	148
<b>第五章 SLAM 和导航 .....</b>	150
5.1 使用 tf 配置机器人 .....	150
5.2 通过 ROS 发布里程计信息 .....	154
5.3 通过 ROS 发布传感器数据流 .....	158
5.4 SLAM .....	163
5.4.1 SLAM 简介 .....	163
5.4.2 slam_gmapping 功能包 .....	163
5.4.3 使用记录的数据建立地图 .....	167
5.4.4 模拟器中建立地图 .....	168
5.4.5 模拟器中使用客户定制地图 .....	170
5.5 配置和使用导航功能包集 .....	172
5.5.1 导航功能包集基本操作 .....	172
5.5.2 在机器人上设置和配置导航功能包集 .....	173
5.5.3 rviz 与导航功能包集配合使用 .....	176
5.5.4 发送目标到导航功能包集 .....	178

<b>第六章 抓取操作</b>	182
6.1 机器人手臂的运动规划	182
6.1.1 安装和配置	182
6.1.2 编译手臂导航功能包集	182
6.1.3 启动模拟器和仿真环境	183
6.1.4 启动相关节点	183
6.1.5 控制手臂运动	185
6.2 运动规划的环境表示	203
6.2.1 基于自滤波数据构建碰撞地图	203
6.2.2 检测关节轨迹碰撞	208
6.2.3 给定机器人状态下的碰撞检测	212
6.2.4 添加已知点到运动规划环境	219
6.2.5 添加物体到机器人本体	224
6.3 用于 PR2 机器人手臂的运动学	230
6.3.1 从 PR2 运动学开始	230
6.3.2 从运动学节点获取运动学求解器信息	231
6.3.3 PR2 手臂运动学正解	233
6.3.4 PR2 手臂运动学逆解	237
6.3.5 PR2 手臂无碰撞运动学逆解	240
6.4 用于 PR2 机器人手臂的安全轨迹控制	245
6.5 使用轨迹滤波节点进行轨迹滤波	247
6.5.1 生成无碰撞三次样条轨迹	247
6.5.2 使用轨迹滤波服务器对关节轨迹进行滤波	248
6.5.3 学习如何创建自己的轨迹滤波	253
6.6 机器人状态和轨迹可视化	256
<b>第七章 Kinect</b>	260
7.1 Kinect 简介	260
7.2 安装驱动	261
7.2.1 Ubuntu 系统上安装 Kinect	261
7.2.2 基于源的安装	261

---

7.3 测试	263
7.3.1 测试 Kinect 彩色摄像机	263
7.3.2 测试 Kinect 深度摄像机	263
7.3.3 测试 Kinect 马达	264
7.4 openni camera	264
7.5 openni tracker	268
<b>第八章 点云库</b>	<b>270</b>
8.1 PCL 简介	270
8.1.1 PCL 架构	270
8.1.2 PCL 数据结构	272
8.1.3 PCL 与 ROS 的集成	273
8.2 PCL 可视化库	273
8.3 PCL 与 Kinect 连接	279
8.4 例子	284
<b>第九章 综合演示示例</b>	<b>288</b>
9.1 实验一：SLAM (即时定位与地图构建)	288
9.2 实验二：机器人导航	294
9.3 实验三：识别并抓取物体	300
<b>参考文献</b>	<b>303</b>
<b>彩图</b>	



# 第一章 ROS 简介

## 1.1 ROS 简介

随着机器人领域的快速发展和复杂化，代码复用和模块化的需求越来越强烈，而已有的开源机器人系统又不能很好地适应该需求。2010 年 Willow Garage<sup>[1]</sup> 公司发布了开源机器人操作系统 ROS(robot operating system)，很快在机器人研究领域展开了学习和使用 ROS 的热潮。

ROS<sup>[2]</sup> 是用于机器人的一种开源的后操作系统，或者说次级操作系统。它提供类似操作系统所提供的功能，包含硬件抽象描述、底层驱动程序管理、共用功能的执行、程序间的消息传递、程序发行包管理，它也提供一些工具程序和库用于获取、建立、编写和运行多机整合的程序。ROS 还提供了库和工具来帮助软件开发者创建机器人的应用程序。

ROS 的主要设计目标是便于机器人研发过程中的代码复用。因此 ROS 是一种分布式的进程框架，使得执行程序可以各自独立地设计，松散地、实时地组合起来。这些进程可以按照功能包和功能包集的方式分组，因而可以容易地分享和发布。ROS 也支持代码库的系统联合。

ROS 执行若干种类型的通信，包括基于服务的同步 RPC 的通信、基于主题的异步数据流及参数服务器上的数据存储。虽然 ROS 集成了实时的代码，但它本身并不具有实时性。

ROS 的主要特点可以归结为以下几条：

(1) 点对点设计。ROS 通过点对点设计以及服务和节点管理器等机制可以分散由于计算机视觉和语音识别等功能带来的实时计算压力，这种设计能够适应服务机器人遇到的挑战。

(2) 不依赖编程语言。ROS 支持多种现代编程语言。C++、Python 和 Lisp 语言已经在 ROS 中实现编译，并得到应用，Octave 和 Java 的测试库也已经实现。为了支持多语言编程，ROS 采用了一种语言中立的接口定义语言 (language-neutral interface definition language, IDL) 来实现各模块之间的消息传送。

(3) 精简与集成。ROS 不修改用户的 main() 函数。所以代码可以被其他的机器人软件使用。其优点是 ROS 很容易和其他的机器人软件平台集成。在计算

机视觉算法方面，ROS 已经与 OpenCV 实现集成。在驱动、导航和模拟器方面，ROS 已经与 Player 系统实现集成。在规划算法方面，ROS 已经与 OpenRAVE<sup>[3]</sup>系统实现集成。

(4) ROS-agnostic 库：首选的开发模型是带有清除功能的接口实现的 ROS-agnostic 库。

(5) 便于测试。ROS 拥有一个名为 rostest 的内建单元/集成测试平台，它很容易集成调试和分解调试。

(6) 规模。ROS 适用于大型运行系统和大型程序开发。

(7) 开源。ROS 遵从 BSD 协议，对个人及商业应用及修改完全免费。

ROS 在某些程度上和下列机器人架构有些相似之处：Player<sup>[4]</sup>、YARP<sup>[5]</sup>、Orocos<sup>[6]</sup>、CARMEN<sup>[7]</sup>、Orca<sup>[8]</sup>、MOOS<sup>[9]</sup> 和 Microsoft Robotics Studio<sup>[10]</sup>。对于简单的无机械手的移动平台机器人来说，Player 是非常不错的选择。ROS 则不同，它被设计为适用于有机械臂和运动传感器的移动平台（倾角激光、云台、机械臂传感器）。与 Player 相比，ROS 更有利于分布式计算环境。当然，Player 提供了较多的硬件驱动程序，ROS 则在高层架构上提供了更多的算法应用（如集成 OpenCV 的视觉算法）。

ROS 系统最早源于 2007 年斯坦福大学人工智能实验室的 STAIR 项目与机器人技术公司 Willow Garage 的个人机器人项目（Personal Robotics Program）之间的合作，2008 年之后由 Willow Garage 公司推动其发展。目前稳定版本情况如下：

- ROS Fuerte Turtle。2012 年 4 月 23 日发布（见图 1.1）。

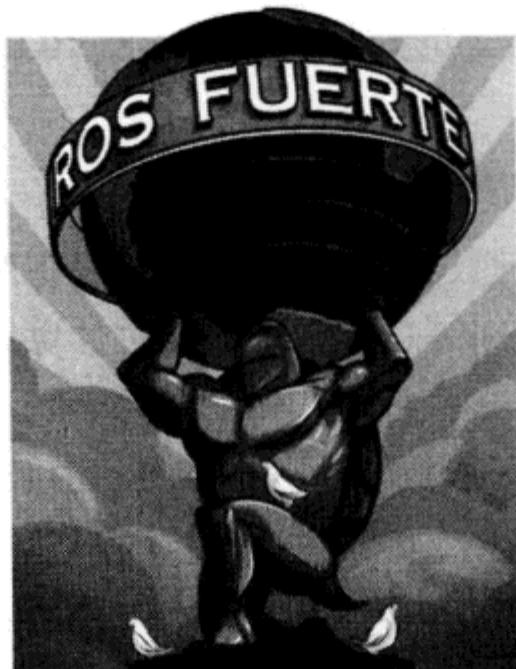


图 1.1 ROS 版本 Fuerte

- ROS Electric Emys。2011 年 8 月 30 日发布（见图 1.2）。



图 1.2 ROS 版本 Electric

- ROS Diamondback。2011 年 3 月 2 日发布 (见图 1.3)。



图 1.3 ROS 版本 Diamondback

- ROS C Turtle。2010 年 8 月 2 日发布 (见图 1.4)。

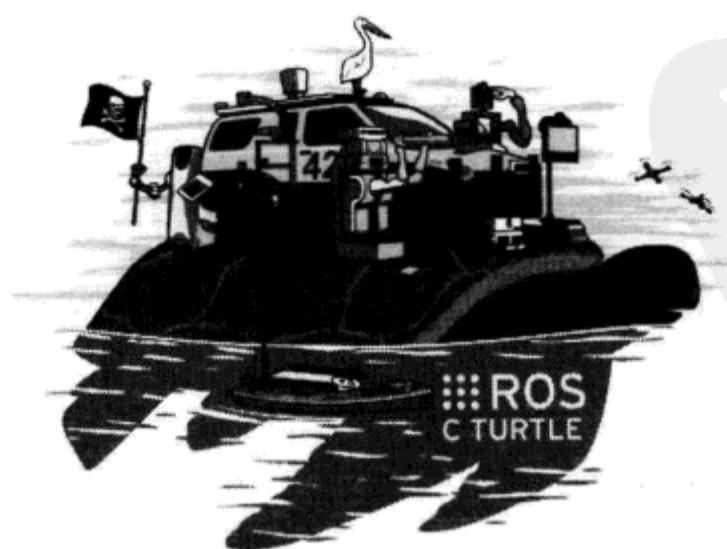


图 1.4 ROS 版本 C Turtle

- ROS Box Turtle。2010年3月2日发布(见图1.5)。

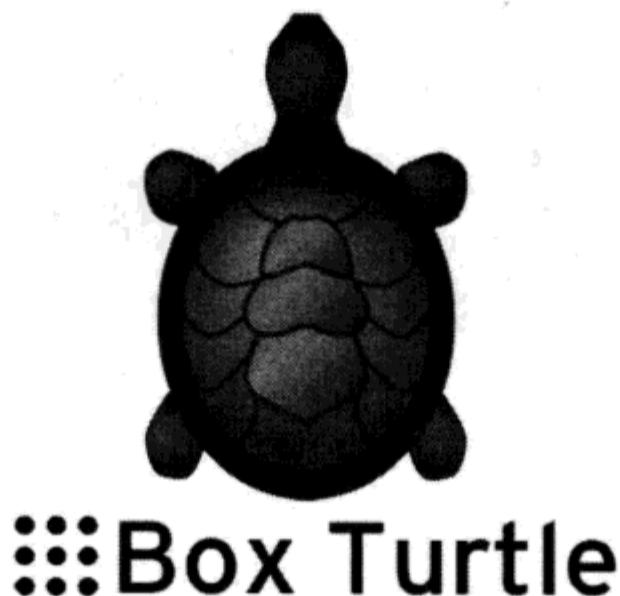


图1.5 ROS版本Box Turtle

## 1.2 ROS 安装

ROS 目前支持的操作系统有: Ubuntu、OS X、Arch、Federa、Gentoo、OpenSUSE、Slackware、Debian。另外, 还可以在 Windows 和 FreeBSD 上安装部分功能。由于 ROS 主要支持 Ubuntu 操作系统, 因此, 本书以 Ubuntu 操作系统下的安装及使用为例, 详细描述 ROS 的主要框架及使用方法。

本书在成书过程中, ROS 的稳定版本为 Diamondback, 第六章中抓取任务的部分程序涉及 ROS Electric Emys 版本, 因此书中如无说明, 所有程序测试以 Diamondback 版本为主。

本小节以 Diamondback 在 Ubuntu 10.04 LTS<sup>[11, 12]</sup> 上面安装为例, 介绍 ROS 安装过程。

### 1. 配置 Ubuntu 系统

配置 Ubuntu repositories 为“restricted”, “universe”和“multiverse”。

### 2. 配置 sources.list

设置计算机使得可以从 ROS.org 接收软件。

(1) Ubuntu 10.04 (Lucid)。

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

(2) Ubuntu 10.10 (Maverick)。

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

(3) Ubuntu 11.04 (Natty)。

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

### 3. 设置 keys

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

### 4. 安装

重新定向 ROS 服务器: `sudo apt-get update`

下面提供四种版本的安装命令:

(1) 桌面完全版安装 (推荐安装): ROS、rx、rviz、robot-generic 库、2D/3D simulators、navigation 和 2D/3D perception。

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

(2) 桌面板安装: ROS、rx、rviz 和 robot-generic 库。

```
sudo apt-get install ros-diamondback-desktop
```

(3) ROS-Base: ROS 主要功能包、build 和 communication 库。不安装 GUI 工具。

```
sudo apt-get install ros-diamondback-ros-base
```

(4) 独立的功能包集: 用户也可以安装特定的 ROS 功能包集。

```
sudo apt-get install ros-diamondback-STACK
```

例子:

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

### 5. 环境设置

环境变量设置是为了每次一个新的 shell 被调用的时候, ROS 的环境变量自动被加入到用户的 bash session 中。

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

如果安装了不止一个版本的 ROS，`~/.bashrc` 必须是当前使用版本的唯一源 `setup.bash`。

如果需要修改当前 shell 的环境，可以使用下面命令：

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

### 1.3 ROS 支持的机器人

ROS 官方网站上列出的支持的机器人系统如下：

- AscTec Pelican/Hummingbird
- Care-O-bot
- Erratic
- Lego NXT
- TurtleBot
- PR2
- Shadow Robot

### 1.4 ROS 网上资源

ROS 官方网站给出了详细的资料：[http://www.ros.org/。](http://www.ros.org/)



## 第二章 ROS 总体框架及基本命令

通过第一章中的简单介绍，已经知道 ROS 的一些基本特点：ROS 是一个开源的元级操作系统（后操作系统），一些包、软件工具的集合。它跨机器进行通信的体系架构提供了对系统进行实时数据分析、编程语言独立（C++、Python、Lisp、Java 等）等功能。它提供类似于操作系统的服务，包括硬件抽象描述、底层驱动程序管理、共用功能的执行、程序间消息传递、程序发行包管理，它也提供一些工具和库用于获取、建立、编写和执行多机融合的程序。本章将对 ROS 的总体框架和基本命令、基本工具进行介绍。

### 2.1 ROS 总体框架

根据 ROS 系统代码的维护者和分布来标识，主要有两大部分，一部分是核心部分，也是主要部分，一般称为 main。主要是 Willow Garage 公司和一些开发者来设计提供与维护。它们提供一些分布式计算的基本工具，以及整个 ROS 系统核心部分的程序编写。这部分内容即被存储在计算机的安装文件中。另外一部分是全球范围的代码，被称为 universe，由不同国家的 ROS 社区组织开发和维护。一种是各种库的代码，如 OpenCV、PCL 等；库的上一层是从功能的角度提供的代码，如人脸识别等，它们调用各种库来实现这些功能；最上层的代码是应用级代码，叫做 apps，可以让机器人完成某一种应用，如去拿啤酒，而这个过程则调用不同功能的代码进行组合，如啤酒的识别、抓取啤酒等。这个一般需要用户下载相应的功能包，然后学习和使用。

不过，对于使用者来说，无论是谁提供和维护的代码，用户都可以下载到自己的电脑上，然后进行下一步的工作。

我们还可以从另外的角度来理解 ROS。ROS 系统有三级概念：文件系统级、计算图级、社区级。除此之外，ROS 也有两种类型的命名：功能包源名称和计算图源名称。

#### 2.1.1 文件系统级

ROS 文件系统级指的是可以在硬盘上面查看的 ROS 源代码，包括如下几种形式：

(1) 功能包。功能包是 ROS 中组织软件的主要形式，一个功能包可能包含 ROS 运行过程(如节点)，一个 ROS 依赖库、数据集、配置文件或者组织在一起的任何其他文件。功能包是 ROS 软件的元级组织形式，它可以包含任何内容：库、工具、可执行文件等。

(2) Manifest。manifest 提供关于功能包的元数据 (meta data)，包括它的许可信息和依赖信息，指定的编程语言信息(像编译标记)。它是功能包的一种描述。事实上，它的最重要功能是定义功能包之间的依赖关系。

(3) 功能包集。功能包集是提供某种功能的一些功能包的集合，可以形成更高级的功能，如“导航功能包集”。它带有相关版本号，是 ROS 软件发布的主要形式。

(4) Stack Manifest。这里指的是 stack.xml 文件，它提供了关于功能包集的相关信息，包括版本信息及其依赖的其他功能包集。它与功能包中的 manifest 文件类似，但是作用于功能包集。

(5) Message(msg) type。消息的描述，定义了 ROS 中发送的消息的数据结构，存储在目录 my\_package/msg/MyMessageType.msg 下。

(6) Service(srv) type。服务的描述，定义了 ROS 中需求和响应的数据结构，存储在目录 my\_package/srv/MyServiceType.srv 下。



图 2.1 ROS 文件系统示例

从图 2.1 中可以看到一个功能包就是带有一个 `manifest.xml` 文件的文件夹，一个功能包集就是一个带有 `stack.xml` 文件的文件夹。

### 2.1.2 计算图级

计算图(见图 2.2)是 ROS 处理数据的一种点对点的网络形式。程序运行时，所有进程及它们所进行的数据处理，将会通过一种点对点的网络形式表现出来。将通过节点、节点管理器、主题、服务等来进行表现。ROS 中基本的计算图级概念包括：节点、节点管理器、参数服务器、消息、服务、主题和包。这些概念以各种形式来提供数据。

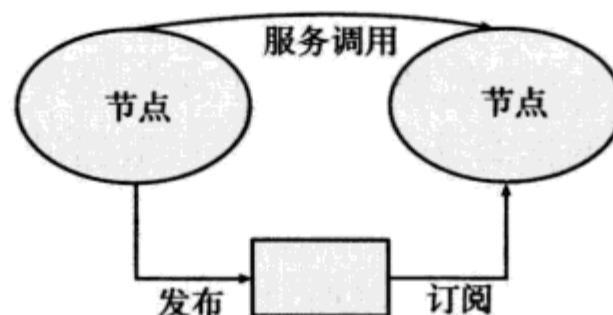


图 2.2 ROS 计算图级概念

在 Diamondback 版本中，下面概念在功能包集 `ros_comm` 中执行。

(1) 节点。节点是执行计算的过程。ROS 被设计为在很精细的尺度上模块化，一个机器人控制系统由很多节点组成。例如，一个节点控制一个激光距离传感器，一个节点控制轮子的马达，一个节点执行定位，一个节点执行路径规划，一个节点提供系统的整个视图……一个 ROS 节点是用 ROS 客户端库(如 `roscpp`、`rospy`)写成的。

(2) 节点管理器。节点管理器为其他计算图提供了名称注册和查找的功能。没有节点管理器，节点将不能互相找到，也不能进行消息交换或者调用服务。

(3) 参数服务器。参数服务器允许数据通过在一个中心位置的关键词来存储。目前它是节点管理器的一部分。

(4) 消息。节点之间通过消息来互相通信。一个消息是一个由类型域构成的简单数据结构。它以基本型的阵列形式支持标准的原始数据类型(像整型、浮点型、布尔型等)。消息可以包含任何嵌套的结构和阵列(这很像 C 语言中的结构体)。

(5) 主题。消息通过一个带有发布和订阅功能的传输系统来传送。一个节点通过把消息发送到一个给定的主题来发布一个消息。主题是用于识别消息内容的名称。一个节点对某一类型的数据感兴趣，它只需要订阅相关的主题即可。一个

主题可能同时有很多的并发主题发布者和主题订阅者，一个节点可以发布和订阅多个主题。一般来说，主题发布者和主题订阅者不知道对方的存在。发布者将信息发布在一个全局的工作区内，当订阅者发现该信息是它所订阅的，就可以接收到这个信息。

(6) 服务：发布/订阅模式是一种很弹性的通信范式。但是其多对多的传输方式是一种不适合于请求/回复交互的方式，请求/回复交互方式经常被用于分布式系统中。请求/回复通过服务来进行，其中服务被定义为一对消息结构：一个用于请求，一个用于回复。一个提供节点提供了某种名称的服务，一个客户通过发送请求信息并等待响应来使用服务。ROS 客户端库通常把这种交互表现为好像这是一个远程程序调用。

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

ROS 节点管理器的作用就像是 ROS 计算图中的名称服务。它为节点保存主题和服务的注册信息。节点通过与节点管理器通信来报告它们的注册信息。当这些节点和节点管理器通信时，它们可以接受别的注册节点的信息，并保持通信正常。当这些注册信息改变（这种改变允许当一个新的节点运行时，动态地创建连接）时，节点管理器也会回调这些节点。

节点可以和其他节点直接相连。节点管理器仅仅提供查找表信息，像 DNS 域名服务器。订阅一个主题的节点将会请求与发布主题的节点进行连接，并确定在一种连接协议上进行连接。ROS 中最通用的协议是 TCPROS，TCPROS 采用标准的 TCP/IP 套接字。

当构建了一个更大、更复杂的系统时，这种架构通过名称来处理和提取繁杂的信息。因此名称在 ROS 中具有非常重要的作用：节点、主题、服务和参数都有名称。每一个客户端库都支持名称的命令行再映射，这意味着一个编译过的程序可以在运行时被重新构建，以便于在不同的计算图级拓扑结构中操作。

例如，为了控制 Hokuyo 激光距离传感器，可以启动 `hokuyo_node` 节点驱动，该驱动将与激光传感器对话，并在主题 `scan` 发布消息 `sensor_msgs/LaserScan`。为了处理数据，用户可以用 `laser_filters` 写一个节点，该节点在主题 `scan` 上订阅消息。通过订阅，用户的滤波将自动从激光传感器接收消息。

节点 `hokuyo_node` 所需要做的就是发布扫描的消息，并不知道是谁来接收。所有的滤波需要做的是订阅扫描，这两个节点相互独立，即发布消息的节点并不知道是否有其他节点订阅了这个消息，而订阅的节点并不知道是否有其他节点会

发布这个消息。这两个节点可以以任何顺序开始、结束、再开始，不会导致任何错误。在实验中，建议先启动订阅的节点，然后启动发布的节点。

如果后面用户需要增加别的激光传感器，导致需要重新配置系统，用户需要做的只是重新映射名称。当开始第一个节点 `hokuyo_node`，可以告诉它把 `scan` 再映射到 `base_scan`，并且滤波也做同样的改变。现在，两个节点都会用 `base_scan` 通信，那么用户可以开始另一个用于新激光传感器的 `hokuyo_node` 节点。

### 2.1.3 社区级

ROS 社区级概念是 ROS 网络上进行代码发布的一种表现形式。因为 ROS 为了最大限度地提高社区参与，使它能够快速的发展，设计者们不再是由少部分人来存放、更新和维护 ROS 代码，而采用软件仓库的模式来处理。每个研究所和组织会以软件仓库为单位来发布他们的代码。这样做的目的是鼓励世界各地的开发者和用户，提供和维护自己的 ROS 软件仓库代码，而且他们对代码具有直接的所有权和控制权。

软件仓库中功能包的数量，从 2007 年 11 月的 1 个到 2008 年 11 月的 4 个，到 2009 年 11 月的 16 个，2010 年 11 月的 52 个，到 2011 年 7 月的 79 个，到最近 2012 年 2 月 institutions 的 repositories 达到 90 个，personal 的达到 14 个，single serving 的达到 15 个。

(1) 发行版本。ROS 发行版本是一系列的带有版本号的功能包集，可以用来安装 ROS。ROS 发行版本类似于 Linux 的发行版本。这使得安装一个软件集合更容易，并且通过一个软件集合来维持一致的版本。

(2) 软件版本仓库。ROS 依赖于一个软件版本仓库组织起来。该软件版本仓库被各个研究所和组织用来发展和发布他们自己的机器人软件组件。

(3) 社区百科。ROS 社区百科是用于记录 ROS 文档信息的主要论坛。任何人可以注册账号并发布他们自己的文档，提供修正或更新、编写教程等。

(4) Bug Ticket System。用于用户发现并提交 Bug 的系统。

(5) 邮件列表。邮件列表是主要的社区通信渠道，用于 ROS 更新和提问的论坛等。

(6) ROS 答案。提问和回答问题的地方。

(7) 博客。Willow Garage 公司的博客，用于提供正常更新，包括照片和视频等资源。

### 2.1.4 更高层概念

ROS 的核心平台，试图设计为尽可能与架构无关。它提供了几种不同的数据通信模式（主题、服务、参数服务器），但它并没有规定如何使用或如何命名。这种方法使 ROS 可以很容易地集成多种架构。但要基于 ROS 建立更大的系统，更高层的概念是必要的。

下面将描述这些功能包集，如 `common`、`common_msgs` 等是如何提供 ROS 更高层概念的。

(1) 坐标系/坐标变换。`tf` 功能包提供了一个基于 ROS 的分布式框架，可以随着时间的推移计算多个坐标系的位置。

(2) 行动/任务 (*Actions/Tasks*)。`actionlib` 功能包定义一个通用的基于主题的界面，来处理 ROS 中的抢占任务。

(3) 消息本体 (*Message Ontology*)。`common_msgs` 功能包集提供一个机器人系统的基本消息本体。这包括行动消息 (`actionlib_msgs`)、诊断消息 (`diagnostic_msgs`)、几何图元消息 (`geometry_msgs`)、机器人导航消息 (`nav_msgs`) 和普通传感器消息 (`sensor_msgs`)。

(4) 插件 (*Plugins*)。`Plugins` 提供一个库，可以在 C++ 代码中动态加载库。

(5) 滤波器 (*Filters*)。滤波器功能包集提供了一个一系列滤波方法的 C++ 库，用于数据处理。

(6) 机器人模型。`URDF` 功能包 (*Unified Robot Description Format*, 统一的机器人描述格式) 定义了一个 XML 格式来描述一个机器人模型，并提供了一个 C++ 解析器。

### 2.1.5 名称

#### 1. 计算图源名称

##### 1) 计算图源名称

计算图源名称提供了分层命名结构，用于在 ROS 计算图中所有源，如节点、参数、主题和服务。这些名称在 ROS 中非常有用，在构建更复杂的系统时更是重中之重。理解这些名称如何工作和如何操作它们非常关键。

在更进一步描述名称之前，给出一些名称的例子：

- /(全局命名空间)
- /foo

- /stanford/robot/name
- /wg/node1

计算图源名称提供了封装作用。每一个源被定义在一个命名空间内，该源可以与其他源共享资源。一般来说，源可以在它们的命名空间内创建源，它们在自己的命名空间内或者上一级命名空间内连接源。连接可以在不同的命名空间的源之间进行，不过这通常通过在两个命名空间集成代码来实现。这种封装分离了系统的不同部分，避免因为偶然的名称错误或者全局的名称劫持而导致错误。

名称是相对来决定的，因此源不一定需要知道它们在哪个命名空间内。这样简化编程为节点，这些节点可以一起起作用，就像是它们在顶级命名空间内。当这些节点集成到更大的系统中，它们可以被放到一个定义该代码集合的名称中。例如，用户可以把标准示例和用户自己的示例用 stanford 和 my 子图融合到一个新的示例中，如果两个示例都有一个节点名字为 camera，它们不会冲突。工具（像图形可视化）和参数（像 demo\_name）等需要对整个视图可见的材料都可以由拓扑节点创建。

## 2) 有效名称

ROS 中，一个有效的名称应该具有如下特征：

- (1) 第一个字符是字母 ([a-z|A-Z])、波浪号 (~) 或者斜线 (/)。
- (2) 后续字符可以是数字或者字母 ([0-9|a-z|A-Z])、下划线 (\_) 或者斜线 (/)。

例外的情况是基本名称中不能有波浪号 (~) 或者斜线 (/)。

## 3) 名称解析

ROS 有四种类型的计算图源名称：基本名称、相对名称、全局名称和私有名称，其语法如下：

- base
- relative/name
- /global/name
- ~ private/name

缺省情况下，名称解析是相对于命名空间进行的。如节点 /wg/node1 的命名空间为 /wg，因此名称 node2 解析为 /wg/node2。

没有名称验证符的名称都是基本名称。基本名称实际上是相对名称的子类，因此有同样的解析规则。基本名称是最常用来初始化节点名称的。

以斜线 (/) 开始的名称是全局名称，它们被完全解析。由于限制了代码可移

植性，全局名称应该尽可能地避免使用。

以符号“~”开始的名称是私有名称。它们可以转换节点名称到一个命名空间中。例如，节点 node1 在命名空间 /wg/ 中有私有命名空间 /wg/node1。私有名称在通过参数服务器传递参数到一个特殊节点时非常有用。

相对名称(缺省)解析例子：

① 如果在全局 / 命名空间中的节点 node1 获取源 bar，那么名称解析为 /bar。

② 如果在 /wg/ 命名空间中的节点 node2 获取源 foo，那么名称解析为 /wg/foo。

③ 如果在 /wg/ 命名空间中的节点 node3 获取源 foo/bar，那么名称解析为 /wg/foo/bar。

全局名称解析例子：

① 如果在全局 / 命名空间中的节点 node1 获取源 /bar，那么名称解析为 /bar。

② 如果在 /wg/ 命名空间中的节点 node2 获取源 /foo，那么名称解析为 /foo。

③ 如果在 /wg/ 命名空间中的节点 node3 获取源 /foo/bar，那么名称解析为 /foo/bar。

私有名称解析例子：

① 如果在全局 / 命名空间中的节点 node1 获取源 ~bar，那么名称解析为 /node1/bar。

② 如果在 /wg/ 命名空间中的节点 node2 获取源 ~foo，那么名称解析为 /wg/node2/foo。

③ 如果在 /wg/ 命名空间中的节点 node3 获取源 ~foo/bar，那么名称解析为 /wg/node3/foo/bar。

#### 4) 再映射

当节点通过命令行启动时，任何一个 ROS 节点内的名字可以被再映射。

## 2. 功能包源名称

### 1) 功能包源名称

为了简化处理磁盘上的首选文件和类型的过程，功能包源名称被用于 ROS 中文件系统级概念。功能包源名称很简单：它们只是在源名称前面加上所在源的

功能包的名字。例如，名称 std\_msgs/String 指的是在名为 std\_msgs 的功能包中 String 的消息类型。

一些 ROS 相关的功能包源名称文件名包括：

- (1) 消息 (msg) 类型。
- (2) 服务 (srv) 类型。
- (3) 节点类型。

功能包源名称类似于文件路径，但它们更短。这是由于 ROS 可以在硬盘上定位功能包的位置，并增加关于它们内容的假设。例如，消息描述总是存储在 msg 子目录下，并有 .msg 扩展名。因此 std\_msgs/String 是 path/to/std\_msgs/msg/String.msg 的缩写。类似地，节点类型 foo/bar 等价于在功能包 foo 内搜索一个名为 bar 的可执行文件。

## 2) 有效名称

功能包源名称由于经常用于自动生成代码，因此有严格的命名规则。由于这个原因，一个 ROS 的功能包名称除了下划线外不能有特殊字符，并且它们必须以字母开头。一个有效名称有如下特征：

- (1) 第一个字符是字母 ([a-z|A-Z])。
- (2) 随后的字符是数字或者字母 ([0-9|a-z|A-Z])、下划线 (\_) 或者斜线 (/)。
- (3) 最多有一个斜线 (/)。

## 2.2 ROS 基本命令

在 ROS 系统中，代码分布在许多功能包和功能包集中，在用户的硬盘上将分属在不同的文件夹。如果每次定位代码时，都要使用 ls 和 cd 这种命令，将非常繁琐。于是 ROS 提供了自己一系列以 ROS 为前缀的命令与工具（如 rosbash 套件中的 roscd、rosls、rosclp 等），许多命令其功能本质上与基本操作系统中常用的命令是一致的，但是它们在 ROS 环境下使用更加方便。下面将列出在使用 ROS 中最常用的命令。

### 2.2.1 ROS 文件系统命令

#### 1. rospack

rospack = ros + pack(age)。

`rospack` 是用于提取文件系统上的功能包信息的命令工具。该工具执行很多打印功能包信息的命令，所有这些命令输出结果到标准输出 `stdout`。任何错误或者警告信息输出到标准错误 `stderr`。

选项 `-q` 可以放在任何子命令之后。它抑制通常输出到标准错误 `stderr` 的大部分错误信息。如果有错误，那么返回代码非零。

(1) 用法：

```
rospack <command> [options] [package]
```

(2) 带参数命令。

① `rospack help`。打印帮助信息。

② `rospack find`。打印到功能包的绝对路径信息。如果没有发现功能包，则返回空字符串。

③ `rospack list`。打印 `<package-name> <package-dir>` 格式的所有功能包列表。

④ `rospack langs`。打印特定语言的客户端库列表。

⑤ `rospack depends`、`depends1`、`depends-manifests`、`depends-indent`、`depends-why`。

- `rospack depends [package]`。打印功能包的所有依赖项。

- `rospack depends1 [package]`。打印功能包的所有第一层依赖项。

- `rospack depends-manifests [package]`。打印所有依赖项的 `manifest.xml` 文件列表。

- `rospack depends-indent [package]`。打印带有缩进格式的依赖项列表，其中缩进表示了其临近关系。

- `rospack depends-why -target=TARGET [package]` (ROS 0.11 中新功能)。

⑥ `rospack vcs`、`vcs0`。

- `rospack vcs [package]`。打印功能包中及其所有依赖项的 `manifest.xml` 文件的版本控制标签。

- `rospack vcs0 [package]`。只打印功能包中的 `manifest.xml` 文件的版本控制标签。

⑦ `rospack depends-on`、`depends-on1`。

- `rospack depends-on [package]`。打印依赖参数值指定功能包的所有功能包。

- `rospack depends-on1 [package]`。打印直接依赖参数值指定功能包的所有功能包。

⑧ `rospack export`。

- `rospack export -lang=LANGUAGE -attrib=ATTRIBUTE [package]`。打印指定语言及属性的所有功能包列表。

- `-deps-only`。该参数的作用在于排除功能包本身。

⑨ `rospack cflag-only-I, cflags-only-other`。

- `rospack cflags-only-I [package]`。打印以参数 `-I` 开头的 `export/cpp/cflags` 列表。

- `rospack cflags-only-other [package]`。打印不是以参数 `-I` 开头的 `export/cpp/cflags` 列表。

- `-deps-only`。该参数的作用在于排除功能包本身。

⑩ `rospack libs-only-L, libs-only-l, libs-only-other`。`rospack export` 命令的变化版本。

- `rospack libs-only-L [package]`。打印以参数 `-L` 开头的 `export/cpp/libs` 列表。

- `rospack libs-only-l [package]`。打印以参数 `-l` 开头的 `export/cpp/libs` 列表。

- `rospack libs-only-other [package]`。打印不是以参数 `-L` 或 `-l` 开头的 `export/cpp/libs` 列表。

- `-deps-only`。该参数的作用在于排除功能包本身。

⑪ `rospack profile`。

- `rospack profile [-length=N]`。强制列出所有目录，并报告控制台 `N` 个花最长时间列表的目录。

- `-zombie-only`。只打印没有任何 `manifest` 的目录。

⑫ `rospack plugins`。

- `rospack plugins -attrib=<attrib> [package]`。检查直接依赖于给定功能包的功能包，提取其功能包名称，并给出指定属性。

- `-top=TOPPKG`。如果给定该参数，扫描除了直接依赖于给定功能包的依赖项，并且满足是 `TOPPKG` 的依赖项或者是 `TOPPKG` 本身。

- ⑬ `rospack_nosubdirs`。可以增加空的 `rospack_nosubdirs` 文件来防止 `rospack` 落入某一个目录。这在阻止功能包树过长以免影响其性能时很有用。

## 2. rosstack

**rosstack** = **ros** + **stack**。

**rosstack** 是用于提取文件系统上的功能包集信息的命令工具。它可以执行一系列与功能包集相关的命令，如给出功能包集列表、功能包集的依赖项列表等。

(1) 用法：

```
rosstack [options] <command> [stack]
```

(2) 带参数命令。

① **rosstack help**。打印帮助信息。

② **rosstack contains**, **contains-path**。

- **rosstack contains <package-name>**。打印包含指定功能包的功能包集。

- **rosstack contains-path <package-name>**。打印包含指定功能包的功能包集路径。

③ **rosstack find [stack]**。打印该功能包集的绝对路径。例子：运行 **rosstack find common**，结果：

```
/Users/homer/ros-pkg/common
```

④ **rosstack list**。

- **rosstack list**。打印 **<stack-name> <stack-dir>** 格式的所有功能包集列表。

- **rosstack list-names**。打印分行形式的功能包集列表。

⑤ **rosstack depends**、**depends1**、**depends-manifests**、**depends-indent**。

- **rosstack depends [stack]**。打印功能包集的所有依赖项。

- **rosstack depends1 [stack]**。打印功能包集的所有第一层依赖项。

- **rosstack depends-manifests [stack]**。打印所有依赖项的 **stack.xml** 文件列表。

- **rosstack depends-indent [stack]**。打印带有缩进格式的依赖项列表，其中缩进表示了其临近关系。

⑥ **rosstack depends-on**, **depends-on1**。

- **rosstack depends-on [stack]**。打印依赖参数值指定功能包集的所有功能包集。

● rosstack depends-on [stack]。打印直接依赖参数值指定功能包集的所有功能包集。

⑦ rosstack profile [-length=N]。强制列出所有目录，并报告控制台 N 个花最长时间列表的目录。

### 3. roscd

roscd = ros + cd。

改变路径到相应功能包或者功能包集。**roscd** 仅仅列出在 **ROS\_PACKAGE\_PATH** 目录下的功能包。

用法：

**roscd [package[/subdir]]**

例子：

(1) 运行 **roscd roscpp**, 结果：

**YOUR\_INSTALL\_PATH/ros/core/roscpp;**

(2) 运行 **roscd roscpp/include**, 结果：

**YOUR\_INSTALL\_PATH/ros/core/roscpp/include;**

(3) 后面不跟任何参数，将列出根目录。运行 **roscd**, 结果：

**YOUR\_INSTALL\_PATH/ros**

(4) 列出日志文件所在目录。运行 **roscd log**。

### 4. rosfs

rosfs = ros + ls。

罗列相应功能包、功能包集文件夹的命令。它是 rosbash 套件的一部分。它可以通过名称来列表一个文件夹下的文件，而不是根据目录列表。

用法：

**rosfs [package[/subdir]]**

例子：运行 **rosfs roscpp\_tutorials**, 结果：

**add\_two\_ints\_client**

**listener\_unreliable**

**add\_two\_ints\_server**

**listener\_with\_tracked\_object**

**add\_two\_ints\_server\_class**

**listener\_with\_userdata**

**anonymous\_listener**

**Makefile**

babbler	manifest.xml
CMakeLists.txt	node_handle_namespaces
custom_callback_processing	notify_connect
listener	srv
listener_async_spin	talker
listener_multiple	time_api
listener_single_message	timers
listener_threaded_spin	

## 5. roscreate-pkg

创建一个新的 ROS 功能包。用法:

```
roscreate-pkg [package_name]
roscreate-pkg [package_name] [depend1] [depend2] [depend3]
```

ROS 以功能包的方式组织起来。功能包包含节点、ROS 依赖库、数据套、配置文件、第三方软件或者任何其他逻辑构成。功能包的目标是提供一种易于使用的结构以便于软件的重复使用。总的来说，ROS 的功能包遵从短小精干的原则：所有的 ROS 功能包包括很多类似的文件：`manifests`、`CMakeLists.txt`、`mainpage.dox` 和 `make`。功能包很容易手工创建，也可使用其他工具如 `roscreate-pkg` 创建。ROS 的功能包可以是 `ROS_ROOT` 或者 `ROS_PACKAGE_PATH` 里面的一个目录/文件夹，包含一个 `manifest.xml` 文件。功能包可以是 ROS 功能包集的组成部分。

这里直接引用官网提供的一个例子：创建 `beginner_tutorials` 功能包，它依赖 `std_msgs`、`roscpp`、`rospy`。运行该例子分成三步。

(1) `roscreate-pkg beginner_tutorials std_msgs roscpp rospy`。运行结果：

```
Created package directory /opt/ros/diamondback/ros/beginner_
tutorials
Created include directory /opt/ros/diamondback/ros/beginner_
tutorials
/include/beginner_tutorials
Created cpp source directory /opt/ros/diamondback/ros/
beginner_tutorials/src
Created package file /opt/ros/diamondback/ros/beginner_
```

```
tutorials/Makefile  
Created package file /opt/ros/diamondback/ros/beginner_  
tutorials/manifest.xml  
Created package file /opt/ros/diamondback/ros/beginner_  
tutorials/CMakeLists.txt  
Created package file /opt/ros/diamondback/ros/beginner_  
tutorials/mainpage.dox  
Please edit beginner_tutorials/manifest.xml and mainpage.dox to  
finish creating your package
```

这里生成的 manifest.xml 如下：

```
1 <package>  
2   <description brief="beginner_tutorials"> beginner_tutorials  
3   </description>  
4   <author>root</author>  
5   <license>BSD</license>  
6   <review status="unreviewed" notes="" />  
7   <url>http://ros.org/wiki/beginner_tutorials</url>  
8   <depend package="std_msgs"/>  
9   <depend package="rospy"/>  
10  <depend package="roscpp"/>  
11 </package>
```

(2) 将该路径添加到 ROS 系统中 (即更新 ROS 查找路径)。

```
export ROS_PACKAGE_PATH=YOUR_BEGINNER_TUTORIALS_PATH:  
$ROS_PACKAGE_PATH
```

当前添加如下：

```
export ROS_PACKAGE_PATH=.:$ROS_PACKAGE_PATH
```

然后需要确认是否在 ROS 中可以找到：

```
rospack find beginner_tutorials
```

运行结果：

```
/opt/ros/diamondback/ros/beginner_tutorials
```

(3) 随后，可以查看功能包相关第一层依赖项，查询结果如下：

```
rospack depends1 beginner_tutorials
```

```
std_msgs
```

**rospy**

**roscpp**

功能包依赖项 **rospy** 的第一层依赖项查询:

**rospack depends1 rospy**

运行结果如下:

**roslib**

**roslang**

功能包所有依赖项查询:

**rospack depends beginner\_tutorials**

结果如下:

**genmsg\_cpp**

**rospack**

**roslib**

**rosconsole**

**std\_msgs**

**roslang**

**rospy**

**xmlrpcpp**

**roscpp**

## 6. roscreate-stack

创建一个新的 ROS 功能包集。其用法类似于 **roscreate-pkg**。

## 7. rosdep

安装 ROS 功能包系统依赖文件。

## 8. rosmake

编译安装一个 ROS 功能包。用法:

• **rosmake [package]**: 编译单个功能包。

• **rosmake [package1] [package2] [package3]**: 编译多个功能包。

这里用一个简单的例子 (**beginner\_tutorials** 功能包) 来看该命令的作用:

**rosmake beginner\_tutorials**, 运行结果:

```
[ rosmake ] Packages requested are: ['beginner_tutorials']
[ rosmake ] Logging to directory/home/mochen/.ros/rosmake/
[ rosmake ] rosmake_output-20120323-164433
[ rosmake ] Expanded args ['beginner_tutorials'] to:
['beginner_tutorials']
[ rosmake ] Checking rosdeps compliance for packages beginner_
```

```
tutorials. This may take a few seconds.  
[ rosmake ] rosdep check passed all system dependencies in  
    packages  
[rosmake-0] Starting >>> roslib [ make ]  
[rosmake-0] Finished <<< roslib ROS_NOBUILD in package roslib  
[rosmake-1] Starting >>> rosbuild [ make ]  
[rosmake-0] Starting >>> std_msgs [ make ]  
[rosmake-1] Finished <<< rosbuild ROS_NOBUILD in package  
    rosbuild No Makefile in package rosbuild  
[rosmake-0] Finished <<< std_msgs ROS_NOBUILD in package  
    std_msgs  
[rosmake-1] Starting >>> roslang [ make ]  
[rosmake-0] Starting >>> rosgraph_msgs [ make ]  
[rosmake-0] Finished <<< rosgraph_msgs ROS_NOBUILD in package  
    rosgraph_msgs  
[rosmake-1] Finished <<< roslang ROS_NOBUILD in package roslang  
    No Makefile in package roslang  
[rosmake-1] Starting >>> cpp_common [ make ]  
[rosmake-0] Starting >>> rospy [ make ]  
[rosmake-0] Finished <<< rospy ROS_NOBUILD in package rospy  
[rosmake-1] Finished <<< cpp_common ROS_NOBUILD in package  
    cpp_common  
[rosmake-1] Starting >>> roscpp_traits [ make ]  
[rosmake-1] Finished <<< roscpp_traits ROS_NOBUILD in package  
    roscpp_traits  
[rosmake-0] Starting >>> rostime [ make ]  
[rosmake-0] Finished <<< rostime ROS_NOBUILD in package rostime  
[rosmake-0] Starting >>> xmlrpcpp [ make ]  
[rosmake-0] Finished <<< xmlrpcpp ROS_NOBUILD in package  
    xmlrpcpp  
[rosmake-0] Starting >>> rosconsole [ make ]  
[rosmake-0] Finished <<< rosconsole ROS_NOBUILD in package  
    rosconsole  
[rosmake-1] Starting >>> roscpp_serialization [ make ]
```

```
[rosmake-1] Finished <<< roscpp_serialization ROS_NOBUILD in
package roscpp_serialization
[rosmake-1] Starting >>> roscpp [ make ]
[rosmake-1] Finished <<< roscpp ROS_NOBUILD in package roscpp
[rosmake-1] Starting >>> beginner_tutorials [ make ]
[rosmake-1] Finished <<< beginner_tutorials [PASS] [ 1.04
seconds ]
[ rosmake ] Results:
[ rosmake ] Built 15 packages with 0 failures.
[ rosmake ] Summary output to directory
[ rosmake ] /home/mochen/.ros/rosmake/rosmake_output-20120323-
164433
```

## 9. rosytic

显示 ROS 系统或者启动文件的错误或者警告信息。

用法: `rosytic` 或者 `rosytic [file]`。

## 10. rosdep

显示功能包结构和依赖文件信息。用法: `rosdep [options]`。

例子: `rosdep install turtlesim`。

如果采用二进制形式, 结果如下:

```
All required rosdeps installed successfully
```

否则, 运行结果如下:

```
executing this script:
set -o errexit
#No Packages to install

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 \sim/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 \sim/ros/ros-deps
cd \sim/ros/ros-deps
wget --tries=10 http://pr.willowgarage.com/downloads/
apache-log4cxx-0.10.0-wg_patched.tar.gz
tar xzf apache-log4cxx-0.10.0-wg_patched.tar.gz
cd apache-log4cxx-0.10.0
./configure --prefix=/opt/ros
make
sudo make install
fi

if [ ! -f /opt/ros/lib/libboost_date_time-gcc42-mt*-1_37.a ] ;
then
mkdir -p ~/ros/ros-deps
cd \sim/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 \sim/ros/ros-deps
    cd \sim/ros/ros-deps
    wget --tries=10 http://pr.willowgarage.com/downloads/apache-
        log4cxx-0.10.0-wg_patched.tar.gz
    tar xzf apache-log4cxx-0.10.0-wg_patched.tar.gz
    cd apache-log4cxx-0.10.0
    ./configure --prefix=/opt/ros
    make
    sudo make install
fi
```

## 2.2.2 ROS 核心命令

### 1. roscore

roscore = ros + core。

运行基于 ROS 系统必需的节点和程序的集合。为了保证节点能够通信，至少要有一个 roscore 在运行。roscore 当前定义为：

- master
- parameter server
- rosout

在当前计算机上运行：roscore，结果如下：

```
... logging to /home/zhang/.ros/log/9bbf9e48-170a-11e1-a71d-
    4487fca3f523/roslaunch-zhang-desktop-5208.log
```

```
Checking log directory for disk usage. This may take a while.
```

```
Press Ctrl+C to interrupt
```

```
Done checking log file disk usage. Usage is <1GB.
```

```
started roslaunch server http://zhang-desktop:52303/
ros_comm version 1.4.8
```

```
SUMMARY
```

```
=====
```

**PARAMETERS**

- \* /rosversion
- \* /rosdistro

**NODES**

```
auto-starting new master
process[master]: started with pid [5222]
ROS_MASTER_URI=http://zhang-desktop:11311/

setting /run_id to 9bbf9e48-170a-11e1-a71d-4487fca3f523
process[rosout-1]: started with pid [5235]
started core service [/rosout]
```

**2. rosmsg/rossrv**

rosmsg = ros + msg, rossrv = ros + srv。

显示消息或者服务的数据结构定义。

- (1) rosmsg show: 显示在消息中域的定义。
- (2) rosmsg users: 显示使用指定消息的代码。
- (3) rosmsg md5: 显示消息的 md5 值。
- (4) rosmsg package: 列出指定功能包中的所有消息。
- (5) rosnode packages: 列出带有该消息的所有功能包。

例子：

- (1) 显示 pose 的消息：

```
rosmsg show pose
```

- (2) 列出在 nav\_msgs 功能包中的消息：

```
rosmsg package nav_msgs
```

- (3) 列出在使用 sensor\_msgs/CameraInfo 的文件：

```
rosmsg users sensor_msgs/CameraInfo
```

**3. rosrun**

rosrun = ros + run。

**rosrun** 允许用户不必先改变到相应目录就可以执行在任意一个功能包下的可执行文件。

用法: **rosrun package executable**.

运行 **turtlesim** 例子: **rosrun turtlesim turtlesim\_node**. 结果如图 2.3 所示。

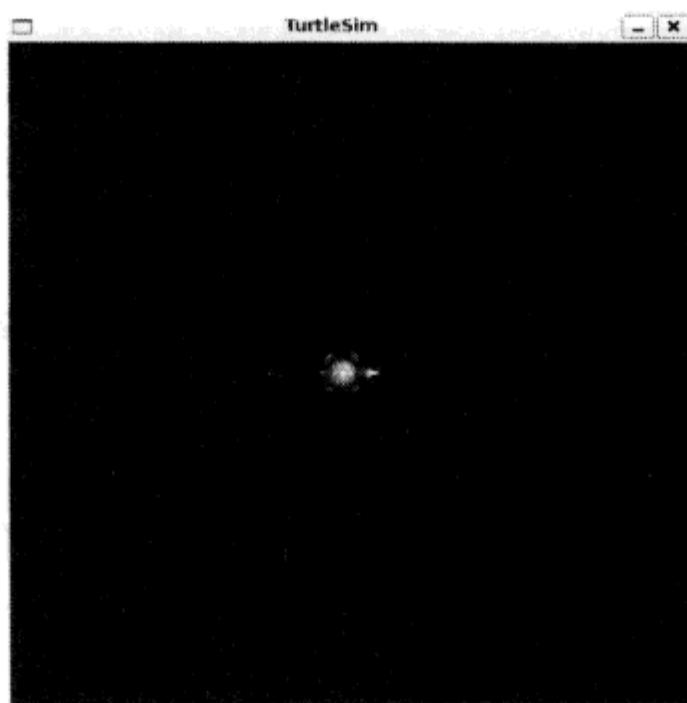


图 2.3 Turtlesim 小海龟例子

ROS 中还可以重命名节点:

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

可以用 **rosnode list** 查看其结果, 运行结果如下:

```
rosout  
my_turtle
```

用 **rosnode ping** 命令: **rosnode ping my\_turtle**, 可查看相关信息:

```
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
```

#### 4. rosnode

**rosnode** = **ros+node**.

显示关于 ROS 节点(包括发布、订阅和连接)的调试信息。命令:

(1) `rosnode ping`: 测试到一个节点的可连接性。

(2) `rosnode list`: 列出活动节点。

(3) `rosnode info`: 打印节点的信息。例子: `rosnode info/roslout`。运行结果如下:

```
-----  
Node [/rosout]
```

```
Publications:
```

```
* /rosout_agg [rosgraph_msgs/Log]
```

```
Subscriptions:
```

```
* /rosout [unknown type]
```

```
Services:
```

```
* /rosout/set_logger_level
```

```
* /rosout/get_loggers
```

```
contacting node http://foo.local:54614/ ...
```

```
Pid: 5092
```

(4) `rosnode machine`: 列出在特定机器上正在运行的节点。

(5) `rosnode kill`: 结束一个正在运行的节点。

例子:

① 结束所有节点: `rosnode kill -a`。

② 列出在机器 `aqy.local` 上的所有节点: `rosnode machine aqy.local`。

③ 测试所有节点的链接情况: `rosnode ping --all`。

## 5. roslaunch

`roslaunch` 通过 SSH 和在参数服务器上设置参数来局部和远程启动 ROS 节点。它通过调用一个或者多个 XML 配置文件来完成启动过程。在配置文件中会对每一个要启动的节点进行描述。用法:

(1) 在不同的接口启动: `roslaunch -p 1234 package filename.launch`。

(2) 在功能包内启动文件: `roslaunch package filename.launch`。

(3) 在局部节点启动文件: `roslaunch --local package filename.launch`.

例子: 在 `turtlesim` 例子中, 创建一个启动文件, 内容如下。可以看到, 启动文件是以一对 `<launch>` 来开始和结束的。

```

1 <launch>
2
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>
10
11 <node pkg="turtlesim" name="mimic" type="mimic">
12 <remap from="input" to="turtlesim1/turtle1"/>
13 <remap from="output" to="turtlesim2/turtle1"/>
14 </node>
15
16 </launch>

```

## 6. rostopic

一个用于显示 ROS 主题(包括发布、订阅、发布频率和消息)调试信息的工具。命令 `rostopic -h` 可以获取该命令的帮助信息。

(1) `rostopic bw`: 显示主题的带宽。

(2) `rostopic echo`: 输出主题信息到屏幕。用法: `rostopic echo [topic]`。

在前面 `turtlesim` 例子中, 可以查看其运行信息:

`rostopic echo /turtle1/command_velocity`。运行结果如下:

---

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
```

(3) rostopic hz: 显示主题发布频率。用法: **rostopic hz [topic]**。

在 turtlesim 的例子中, 运行 **rostopic hz /turtle1/pose**。运行结果如下:

```
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
```

(4) rostopic list: 打印活动主题的信息。用法: **rostopic list [/topic]**。

具体参数如下:

- **-h, --help** 显示帮助信息。

- **-b BAGFILE, --bag=BAGFILE** 列出在记录包文件中的主题。

- **-v, --verbose** 列出每个主题的详细信息。

- **-p,** 只列出发布者。

- **-s,** 只列出订阅者。

(5) rostopic pub: 发布数据到主题。

用法: **rostopic pub [topic] [msg\_type] [args]**。

例子:

```
rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity
```

```
-- 2.0 1.8
```

该命令告诉 `turtlesim` 以线速度 2.0 和角速度 1.8 运动，因此可以在屏幕上看到小海龟在做圆周运动。其中：

- ① 参数 `-1` 告诉程序仅发布一条消息然后退出。
- ② 参数 `/turtle1/command_velocity` 是将要发布的主题名称。
- ③ 参数 `turtlesim/Velocity` 是发布的主题的消息类型。
- ④ 参数 `--` 告诉程序随后的字符是必须的选项。当参数中出现短划线时，该参数是必需的。
- ⑤ 参数 `2.0` 和 `1.8` 是对应的线速度和角速度。

(6) `rostopic type`: 打印主题类型。用法：`rostopic type [topic]`。

ROS 节点之间通过传送消息实现通信。在 `turtlesim` 例子中，消息主题发布者 `turtle_teleop_key` 和消息订阅者 `turtlesim_node` 要实现通信，必须发送和接收同一类型的消息。这意味着主题类型是由发布在其上的消息类型决定的。消息的类型可以由命令 `rostopic type` 决定。

例子：`rostopic type /turtle1/command_velocity`。

结果如下：

`turtlesim/Velocity`

命令 `rosmsg` 可以给出该消息的详细信息。运行：

`rosmsg show turtlesim/Velocity`。运行结果如下：

`float32 linear`

`float32 angular`

(7) `rostopic find`: 通过类型查找主题。

上述各命令混合使用的例子：

① 以 10Hz 频率发布消息 `hello`:

`rostopic pub -r 10 /topic name std_msgs/String hello`

② 消息发布之后清屏:

`rostopic echo -c /topic name`

③ 显示匹配给定 Python 表达式的消息:

`rostopic echo --filter "m.data=='foo'" /topic_name`

④ 为了查看消息类型而显示 ROS 主题的输出:

`rostopic type /topic_name rosmsg show`

以前面运行的 `turtlesim` 为例(见图 2.4)，可以看到节点 `turtlesim` 和节点 `/teleop_turtle` 在以 `/turtle1/command_velocity` 为名的主题上互相通信。

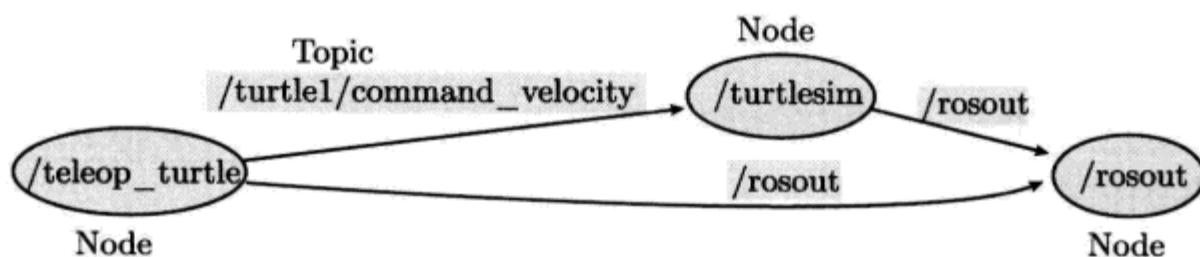


图 2.4 用 rxgraph 查看 Turtlesim 运行主题

## 7. rosparam

一个获取和设置参数服务器上用 YAML 编码的文件的工具。在简单的情况下，YAML 看起来很自然，1 是整型，1.0 是浮点型，one 是字符串，true 是布尔型，[1,2,3] 是整型列表，{a: b, c: d} 是字典。

命令：

(1) rosparam set：设置一个参数。用法：

```
rosparam set [param_name]
```

(2) rosparam get：获取一个参数。用法：

```
rosparam get [param_name]
```

(3) rosparam load：从一个文件中调取一个参数。用法：

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

(4) rosparam dump：写参数到一个文件。用法：

```
rosparam dump [file_name]
```

(5) rosparam delete：删除一个参数。

(6) rosparam list：列出参数名称。

例子：

① 列出在命名空间中的所有参数：

```
rosparam list /namespace
```

② 设置一个列表，其参数为字符串、整型和浮点型：

```
rosparam set /foo "['1', 1, 1.0]"
```

③ 把在特定命名空间中的参数写入一个文件：

```
rosparam dump dump.yaml /namespace
```

## 8. rosservice

一个用于列表和查询 ROS 服务器的工具。服务是节点之间互相通信的另一种方式，服务允许节点发送请求和接收响应。

命令：

(1) `rosservice list`: 打印活动服务的信息。在运行 `turtlesim` 例子后，可以查看其服务的信息，运行结果如下：

<code>/clear</code>	<code>/kill</code>
<code>/reset</code>	<code>/rosout/get_loggers</code>
<code>/rosout/set_logger_level</code>	<code>/spawn</code>
<code>/teleop_turtle/get_loggers</code>	<code>/teleop_turtle/set_logger_level</code>
<code>/turtle1/set_pen</code>	<code>/turtle1/teleport_absolute</code>
<code>/turtle1/teleport_relative</code>	<code>/turtlesim/get_loggers</code>
<code>/turtlesim/set_logger_level</code>	

(2) `rosservice node`: 打印提供一个服务的节点的名称。

(3) `rosservice call`: 启动给定变量的服务。用法：`rosservice call [service] [args]`。

在运行 `turtlesim` 例子后，运行命令 `rosservice call clear`，可以发现其界面背景被清除干净。

(4) `rosservice args`: 列出一个服务的变量。

(5) `rosservice type`: 打印服务类型。用法：`rosservice type [service]`。

在运行 `turtlesim` 例子后，运行命令

```
rosservice type spawn ~ rossrv show
```

运行结果如下：

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

(6) `rosservice uri`: 打印服务 ROSRPC uri。

(7) `rosservice find`: 通过服务类型查找服务。

例子：

① 从命令行启动一个服务：

```
rosservice call /add_two_ints 1 2
```

② 把 ROS 服务输出到 `rossrv` 以查看服务类型：

```
rosservice type add two ints ~ rossrv show
```

③ 显示特定类型的所有服务：

```
rosservice find rospy_tutorials/AddTwoInts
```

## 2.3 工具

### 2.3.1 3D 可视化工具：rviz

rviz 是 ROS 中的一个 3D 可视化工具。

命令参数及用法：

- -h, --help。打印描述命令行选项的帮助信息。

- -d, --display-config <arg>。开始时调用配置文件 <arg>。

- -t, --target-frame <arg>。设置目标坐标系为 <arg>。覆盖在配置文件中指定的目标坐标系。

- -f, --fixed-frame <arg>。设置固定坐标系为 <arg>。覆盖在配置文件中指定的目标坐标系。

#### 1. 安装及编译

首先需要安装及编译 rviz，然后运行。

```
rosdep install rviz  
rosmake rviz  
rosrun rviz rviz
```

rviz 启动时的效果如图 2.5 所示。中间黑色区域为 3D 可视化区域。左侧为显示面板，其中可以显示各种用户加载的选项。右侧为全局选项和时间。

#### 2. 显示

##### (1) 增加新的显示项。

为了增加新的显示选项，可以选择增加新的显示项按钮，如图 2.6 所示。

选择增加新的显示项按钮，会弹出新的对话框，如图 2.7 所示。在对话框上部包含了显示类型，其中详细描述了什么类型的数据可以可视化。中部的列表给出了选择的显示类型的描述。最后，用户必须给定这些显示项唯一的名称。例如，如果有两个激光扫描仪，则用户需创建两个 Laser Scan，并给出不同的名称。

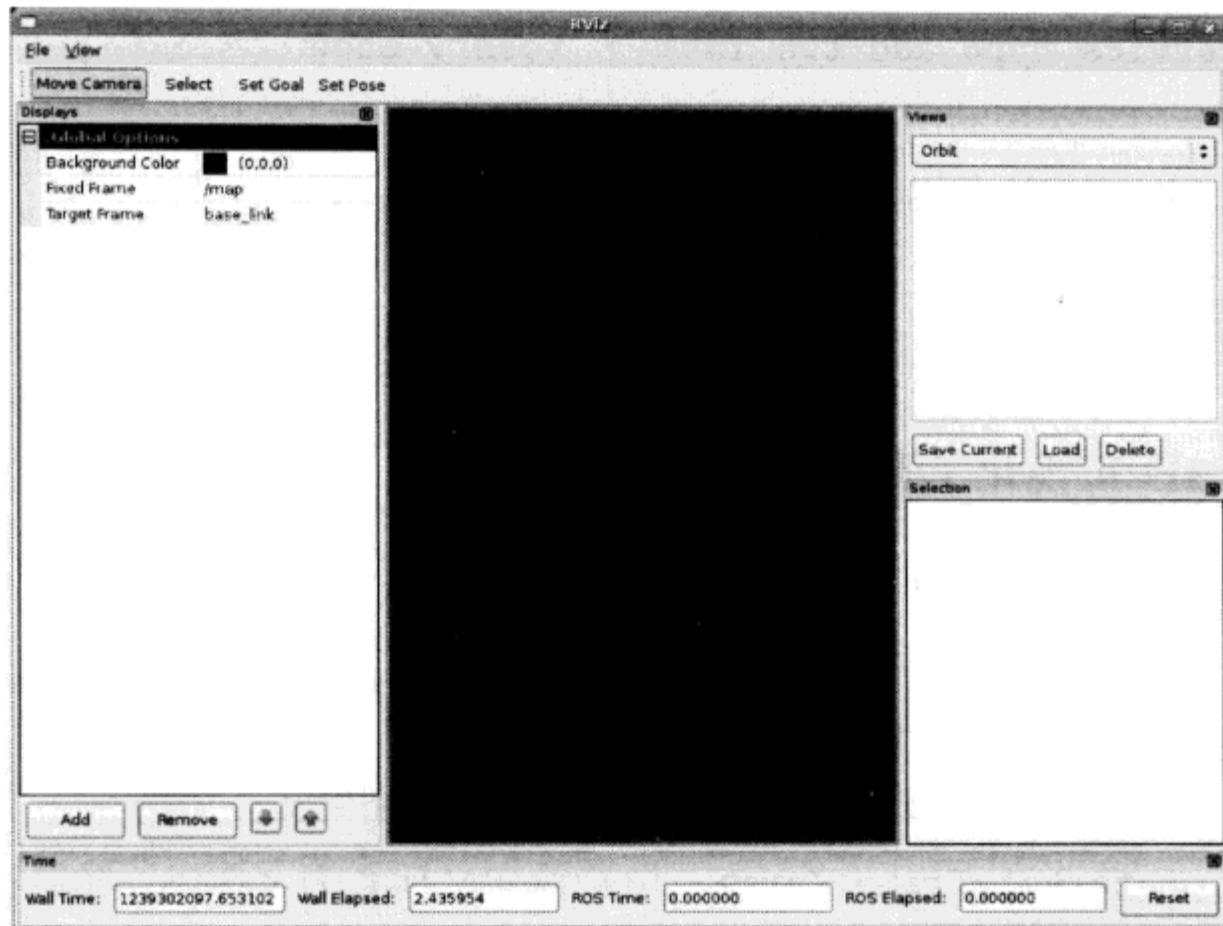


图 2.5 rviz 启动界面

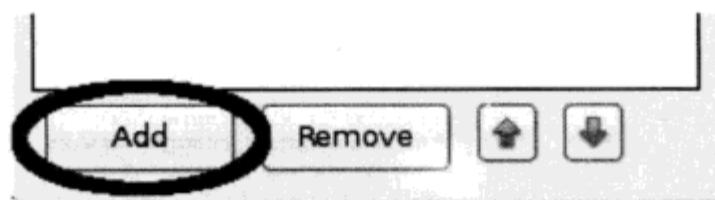


图 2.6 rviz 添加新显示按钮



图 2.7 rviz 添加新显示按钮对话框

(2) 显示属性。每一个显示都会有自己的属性列表，如图 2.8 所示。

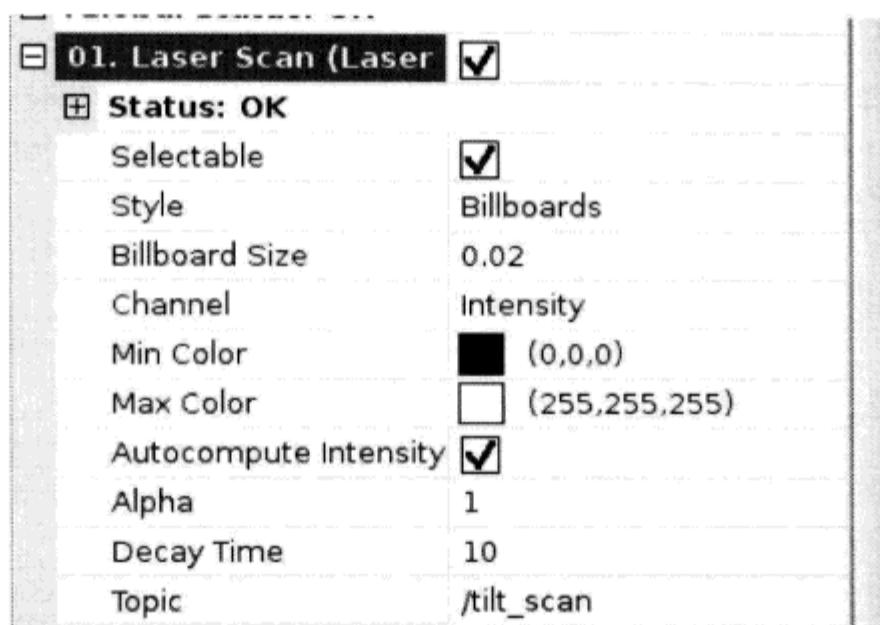


图 2.8 rviz 显示属性

(3) 显示状态。每一个显示项都有自己的状态，以帮助用户判断显示项是否有问题。显示状态有四种：OK、Warning、Error 和 Disabled。具体可见图 2.9。

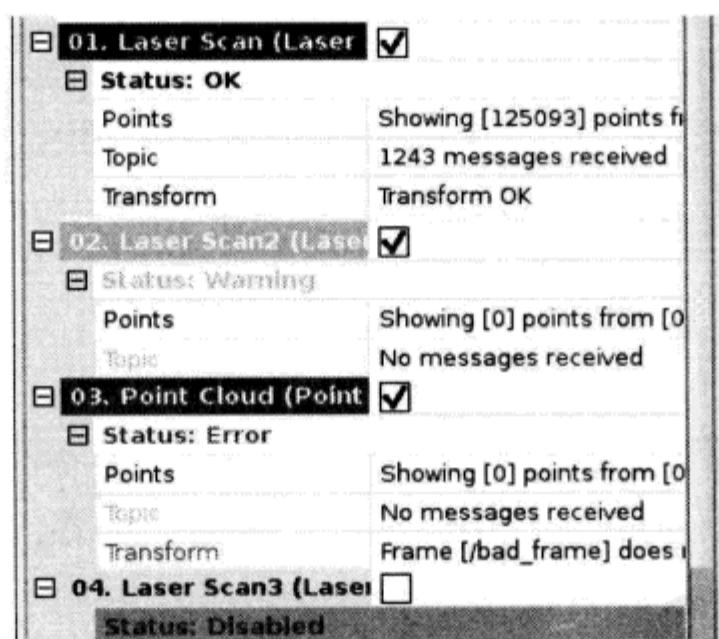


图 2.9 rviz 显示所有状态

(4) 移动显示项。用户可以使用向上或者向下箭头来移动显示项，如图 2.10 所示。

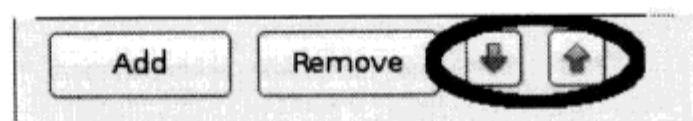


图 2.10 rviz 移动显示选项

(5) 内置显示类型(见表 2.1)。

表 2.1 rviz 内建显示类型

名称	描述	使用的消息
Axes	显示轴的集合	
Camera	从摄像机角度创建一个新的渲染窗口，并在上面覆盖图像	<code>sensor_msgs/Image</code> , <code>sensor_msgs/CameraInfo</code>
Grid	沿着一个平面显示一个 2D 或 3D 网格	
Grid Cells	从一个网格画一个新的单元，通常是代价地图中的障碍物	<code>nav_msgs/GridCells</code>
Image	用一幅图像创建一个新的渲染窗口，这个显示不使用摄像机信息	<code>sensor_msgs/Image</code>
InteractiveMarker	从一个或多个交互标记显示 3D 目标	<code>visualization_msgs/InteractiveMarker</code>
Laser Scan	从激光扫描仪获取数据并显示，不同选项采用不同模式显示	<code>sensor_msgs/LaserScan</code>
Map	在地面显示一幅地图	<code>nav_msgs/OccupancyGrid</code>
Markers	允许程序员通过主题显示任意元素	<code>visualization_msgs/Marker</code> , <code>visualization_msgs/MarkerArray</code>
Path	从导航功能包集显示一条路径	<code>nav_msgs/Path</code>
Pose	把位姿画成一个箭头或者轴	<code>geometry_msgs/PoseStamped</code>
Pose Array	画成箭头云，每一个位姿一个箭头	<code>geometry_msgs/PoseArray</code>
Point Cloud	由点云显示数据	<code>sensor_msgs/PointCloud</code> , <code>sensor_msgs/PointCloud2</code>
Polygon	以线段画出多边形轮廓	<code>geometry_msgs/Polygon</code>
Odometry	随时间累计里程计位姿	<code>nav_msgs/Odometry</code>
Range	由声纳或者红外传感器数据显示距离度量	<code>sensor_msgs/Range</code>
RobotModel	显示机器人位姿	—
tf	显示 tf 变换树	—

### 3. 配置

不同的显示配置应用于不同的可视化工具，适用于不同的机器人。可视化工具让用户可以装载和保存不同的配置。配置包含如下内容：

- (1) 显示 + 对应的属性。
- (2) 工具属性。
- (3) 摄像机类型 + 初始视点设置。

0.4 及以上版本中全局/局部配置已经被移除了，代替的是最近配置菜单，如图 2.11 所示。0.3 及以下版本中有一些内置配置可用。

### 4. 视图面板

在可视化工具中，有一些不同的摄像机类型可用，随着时间增加，列表长度也会增加，一个例子如图 2.12 所示。摄像机类型包含不同的控制方式和投影类型（正交或者透视）。

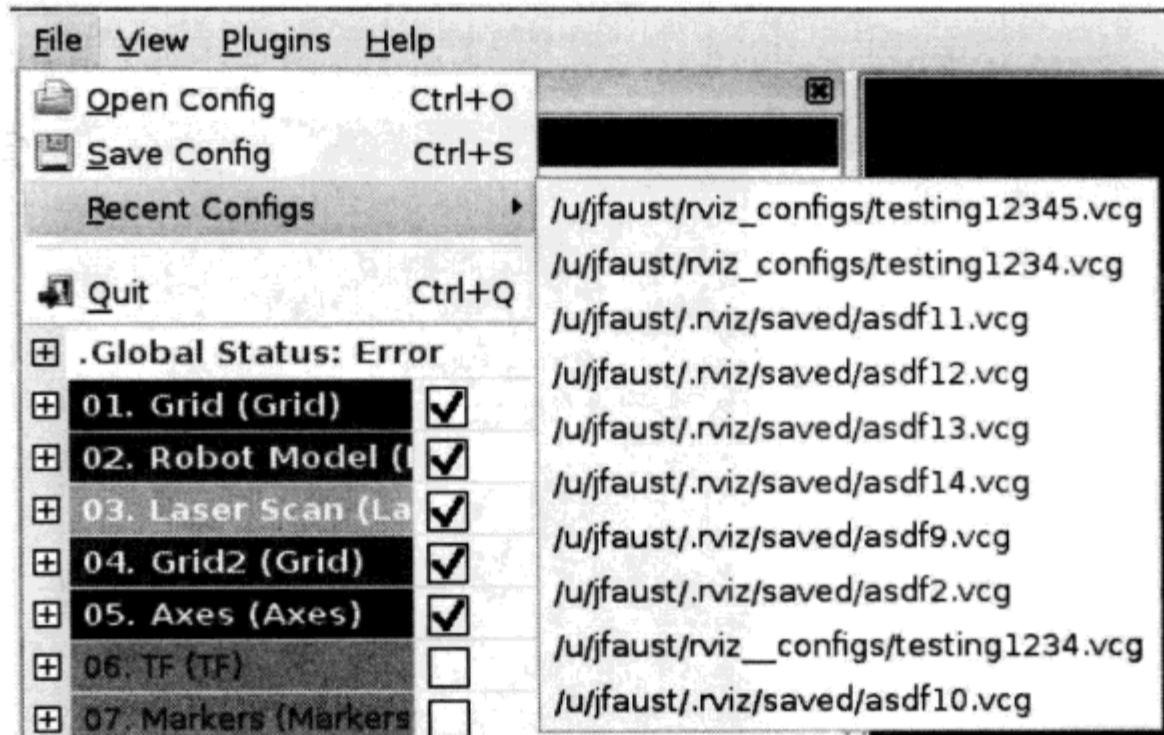


图 2.11 rviz 最近配置选项

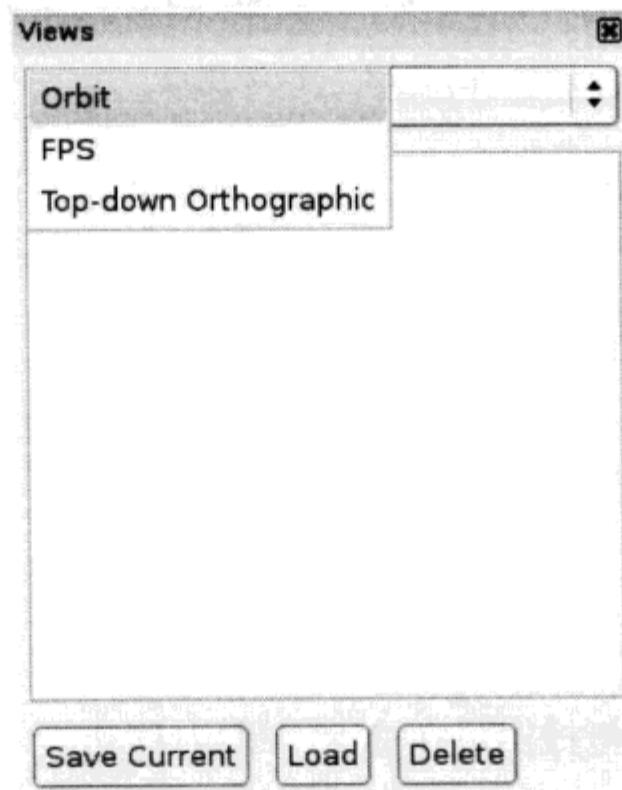


图 2.12 rviz 摄像机类型选项

### 1) 不同的摄像机类型

(1) Orbital Camera(缺省)。轨道摄像机只是沿着视点旋转。当移动摄像机时，视点显示为一个小的圆盘，如图 2.13 所示。控制方式如下：

- 鼠标左键：点击并拖拽，实现围绕视点旋转。
- 鼠标中键：点击并拖拽实现在平面内移动视点。
- 鼠标右键：点击并拖拽实现放大/缩小视点功能。
- 鼠标滚轮：实现放大/缩小视点功能。

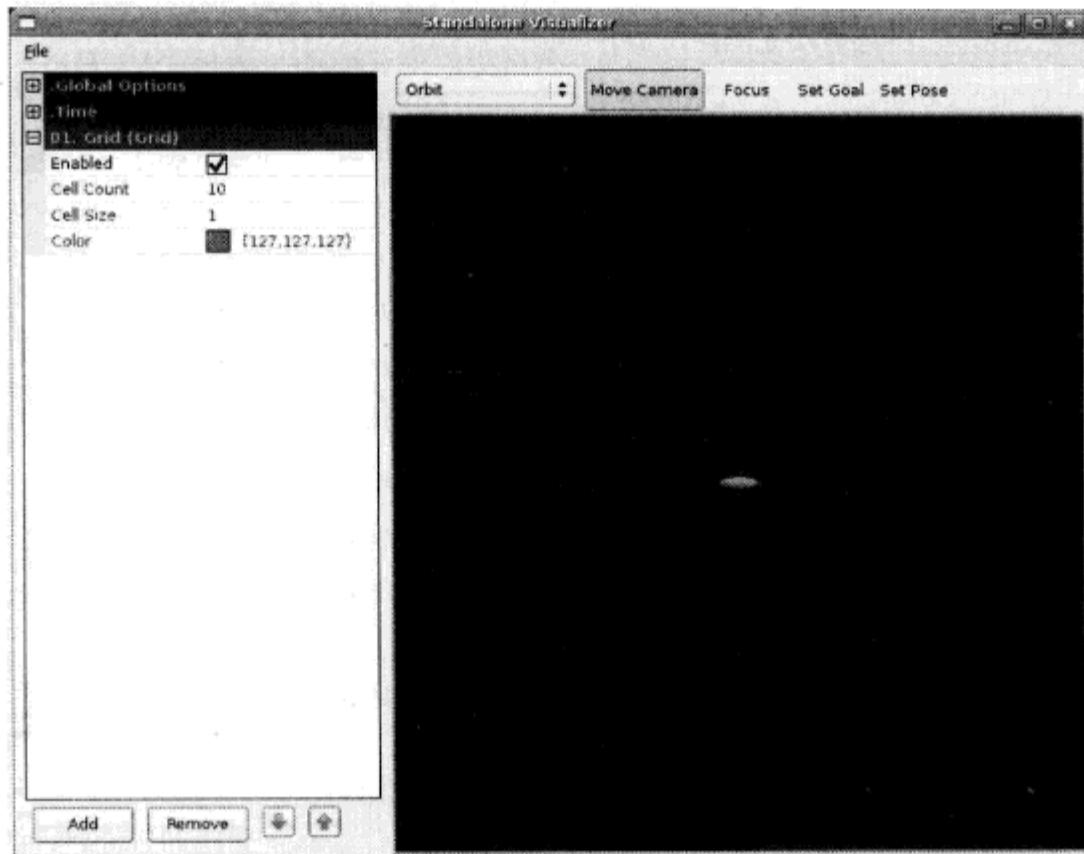


图 2.13 rviz 焦点

(2) FPS (first-person) Camera。控制方式如下：

- 鼠标左键：点击并拖拽，实现围绕视点旋转；点击选中目标。
- 鼠标中键：点击并拖拽实现在平面内移动视点。
- 鼠标右键：点击并拖拽实现沿着摄像机前向向量方向移动。
- 鼠标滚轮：前后移动。

(3) Top-down Orthographic。控制方式如下：

- 鼠标左键：点击并拖拽，实现围绕 Z 轴旋转。
- 鼠标中键：点击并拖拽实现在 XY 平面内移动。
- 鼠标右键：点击并拖拽实现放大图像。
- 鼠标滚轮：放大图像。

## 2) 视图

视图面板让用户可以创建不同名称的视图，这些视图可以保存并且可以在它们之间切换（见图 2.14）。

视图包括如下内容：

- (1) 视图控制类型。
- (2) 视图配置，如位置、方向等。
- (3) 目标坐标系。

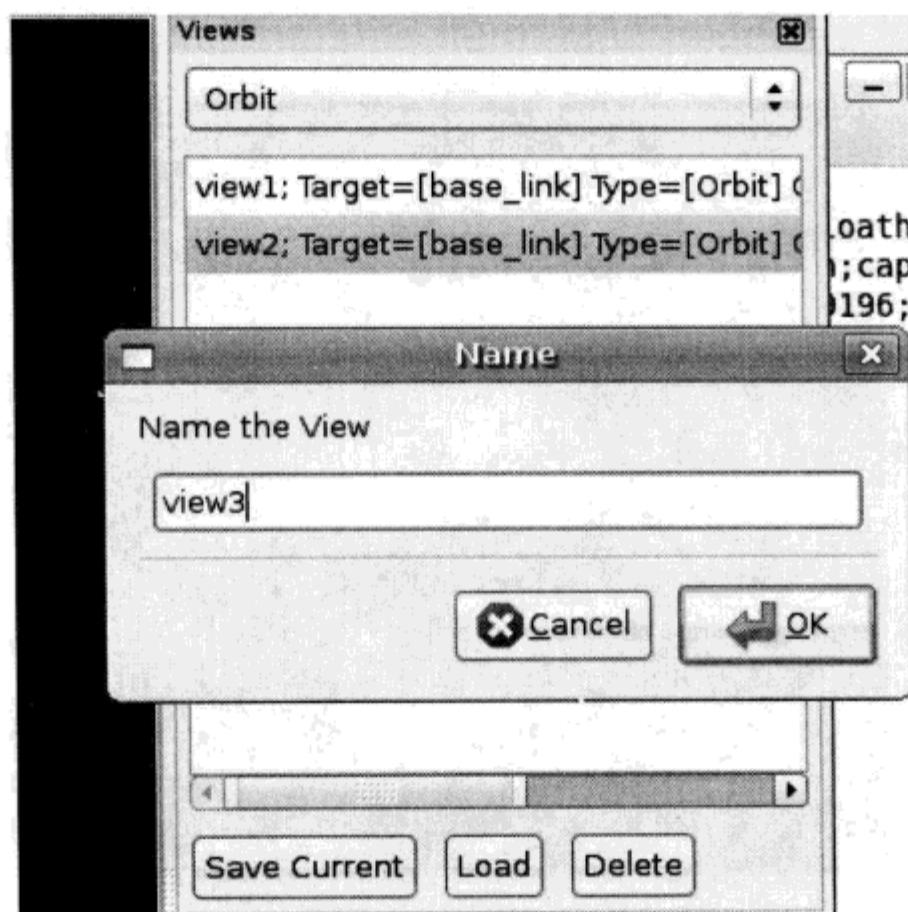


图 2.14 rviz 视图选项

## 5. 坐标系

rviz 使用 tf 变换系统来实现不同坐标系下数据变换。

### 1) 固定坐标系

两种坐标系中比较重要的一种是固定坐标系。固定坐标系是用于表示世界的参考坐标系。为了能够得到正确的结果，固定坐标系不能相对世界移动。

### 2) 目标坐标系

目标坐标系是相对于摄像机视角的参考坐标系。例如，若目标坐标系是地图，可以看到机器人沿着地图移动。如果目标坐标系是机器人基座，那么机器人将固定在某一位置，而其他物体相对于机器人移动。

## 6. 工具

(1) 移动摄像机 (快捷键：M)。这是缺省工具。

(2) 选择工具 (快捷键：S)。选择工具允许用户在 3D 视图中选择其中的项。它支持单点选择和点击拖拽等功能图 2.15~ 图 2.17 分别为 rviz 中选中区域、选中目标、设置目标效果截图。可以使用 Shift 键进行选择，使用 Ctrl 键移除选中的项。如果想选择的同时移动摄像机而不来回切换，可以同时按住 Alt 键。F 键将会把当前选择项定为摄像机。

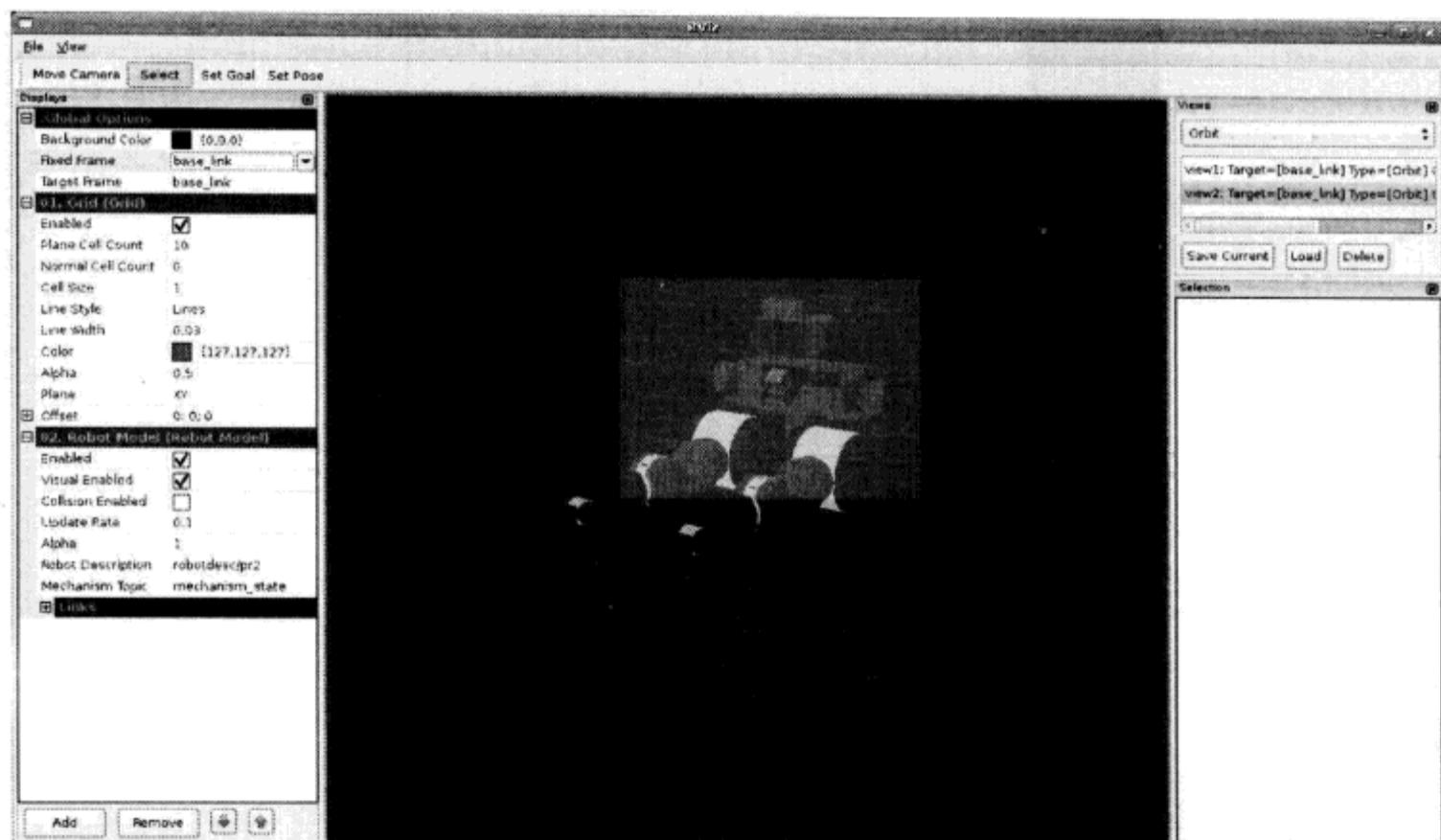


图 2.15 rviz 选中区域高亮显示



图 2.16 rviz 中选中目标

(3) 2D 导航目标 (快捷键: G)。在地面上点击定位位置并拖拽选中的方向即可。

(4) 2D 位姿估计 (快捷键: P)。这个工具让用户设置初始位姿来启动定位系统，在地面上点击定位位置并拖拽选中的方向即可。效果如图 2.18 所示。

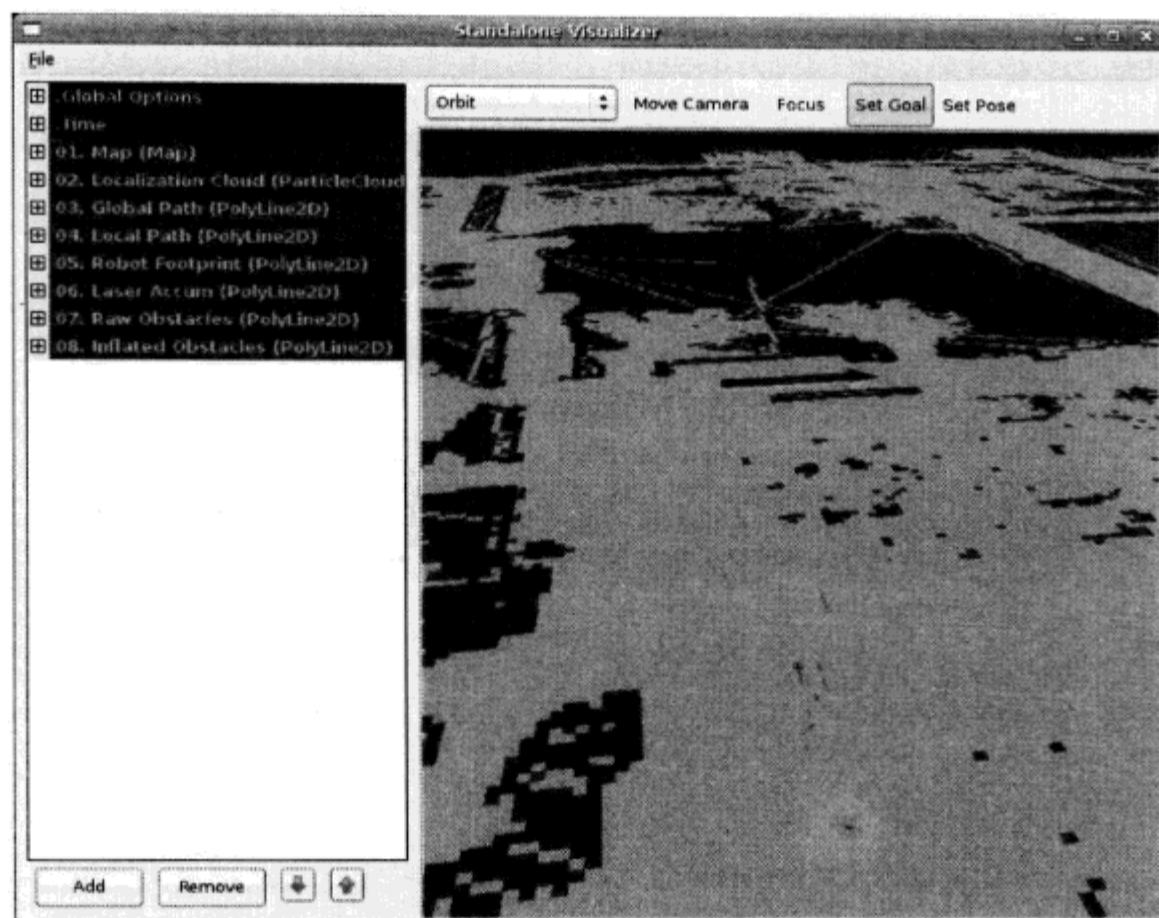


图 2.17 rviz 中设置目标

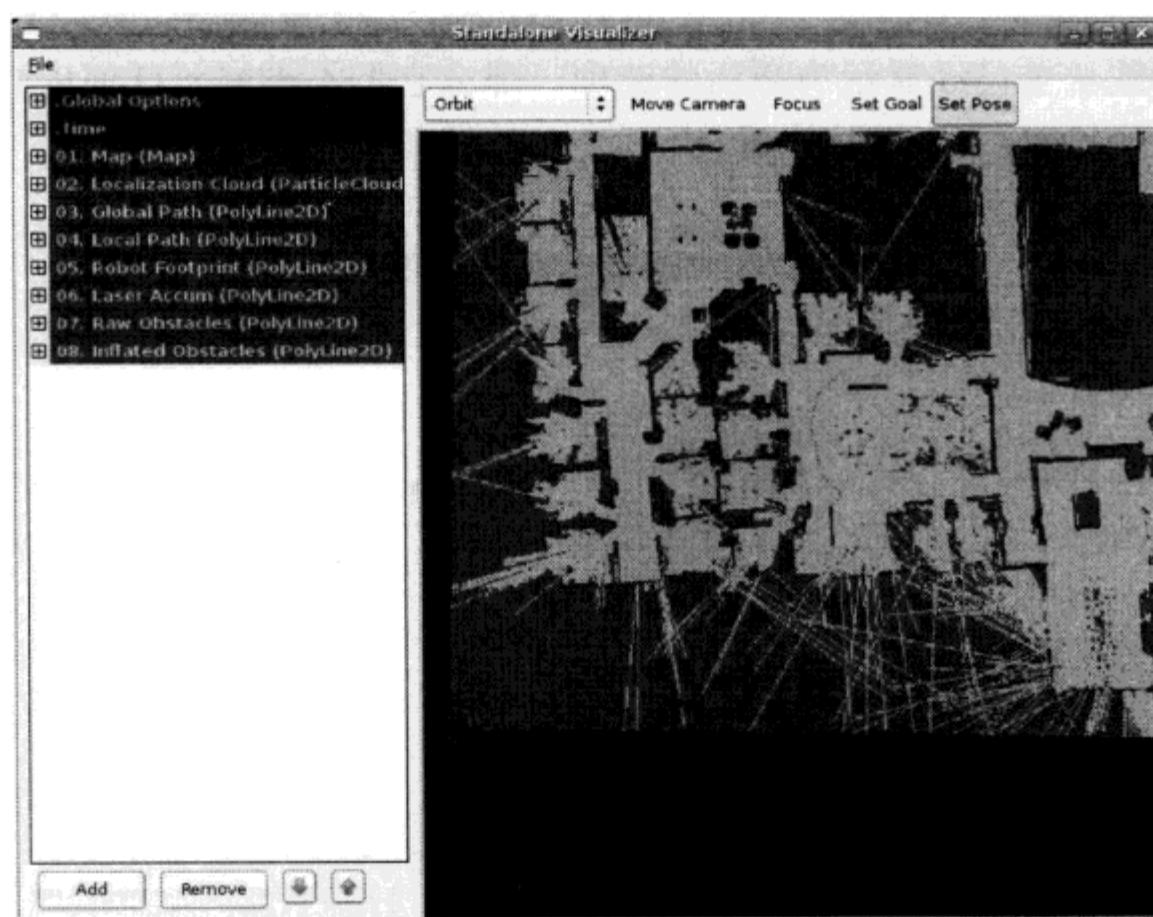


图 2.18 rviz 中设置位姿

## 7. 时间面板

时间面板经常在运行模拟器时用到，它允许用户查看系统已经运行多久或者实际运行时间是多少。时间面板还允许用户重设可视化工具内部时间状态。

## 8. 插件

rviz 设置为新的显示项以插件形式添加进来。实际上，内建的显示项已经通过缺省插件形式加载。用户可以通过用户界面来装载/卸载插件，如图 2.19 所示。

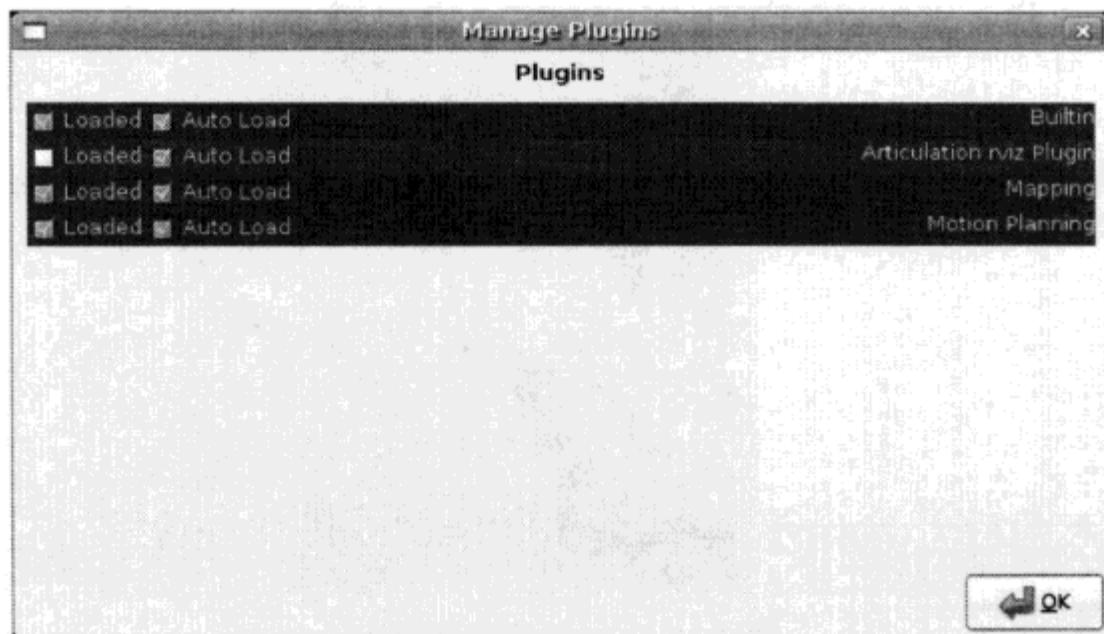


图 2.19 rviz 中管理插件

如果卸载一个已经显示为激活的选项，那么它们不再显示任何内容，但是还会保持它们的设置，如图 2.20 所示。

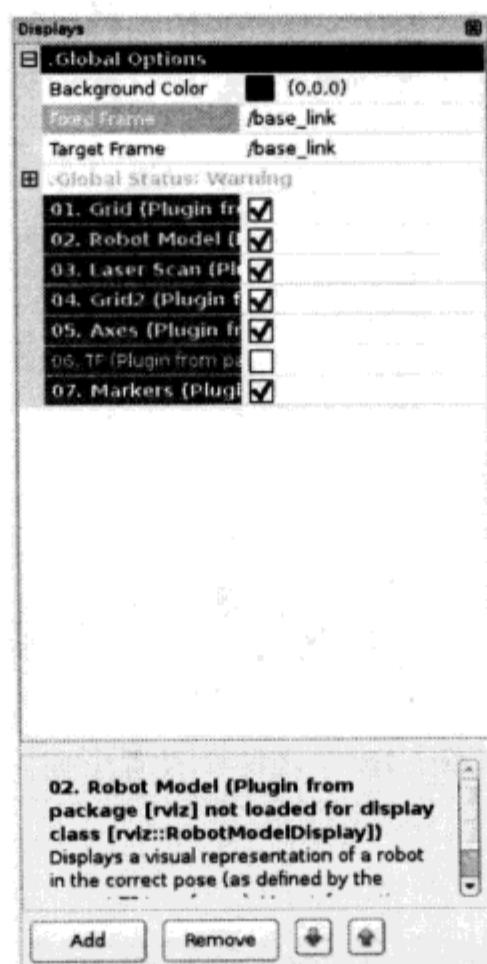


图 2.20 rviz 中卸载插件

### 2.3.2 传感器数据记录与可视化工具：rosbag 和 rxbag

#### 1. rosbag

rosbag 是一个用于记录和回放 ROS 主题的工具集合。它的目的是提供高效的执行，避免消息的反序列化和重序列化。

rosbag 可以实现记录消息，从一个或多个消息记录包重新发布消息，总结消息记录包的内容，检查消息定义，基于 Python 表达式过滤消息记录包消息，压缩及解压缩消息记录包，以及重建消息记录包索引等功能。

目前支持的命令列表：

- record：记录带有特定主题的消息记录包文件。
  - 记录所有主题：`rosbag record -a`
  - 记录选定主题：`rosbag record topic1 topic2`
- info：总结消息记录包的内容。
- play：回放一个或多个消息记录包内容。
  - 无需等待回放所有消息：`rosbag play -a demo_log.bag`
  - 一次回放几个包文件：`rosbag play demo1.bag demo2.bag`
- check：确定消息记录包是可回放的。
- fix：修复消息记录包中的消息。
- filter：使用 Python 表达式转换消息记录包文件。
- compress：压缩一个或多个消息记录包文件。
- decompress：解压缩一个或多个消息记录包文件。
- reindex：重建一个或多个消息记录包索引。

#### 2. rxbag

##### 1) 概述

rxbag 是一个记录和管理消息记录包文件的应用工具。rxbag 可以通过插件扩展，核心插件包含在 `visualization` 功能包集下的 `rxbag_plugins` 功能包中。

用法：`rxbag bag_file.bag`。

其主要特点为：

- (1) 显示消息记录包消息内容的树状视图。
- (2) 显示图像消息。

- (3) 画出可配置的消息值时间序列。
- (4) 在特定主题上发布/记录消息。
- (5) 导出一个时间段内的消息到一个新的消息记录包。
- (6) 导出消息内容到以逗号作为分隔符的 (CSV) 文件。

### 2) 运行窗口、时间线及菜单栏

主运行窗口显示消息流的时间线表示，如图 2.21 所示。消息显示在消息记录包文件的时间戳上。这里的时间戳不同于消息里面的 Header 时间戳，例如，`rosbag record` 记录了接收消息的时间。右击时间线，将会打开一个带有消息视图的菜单。

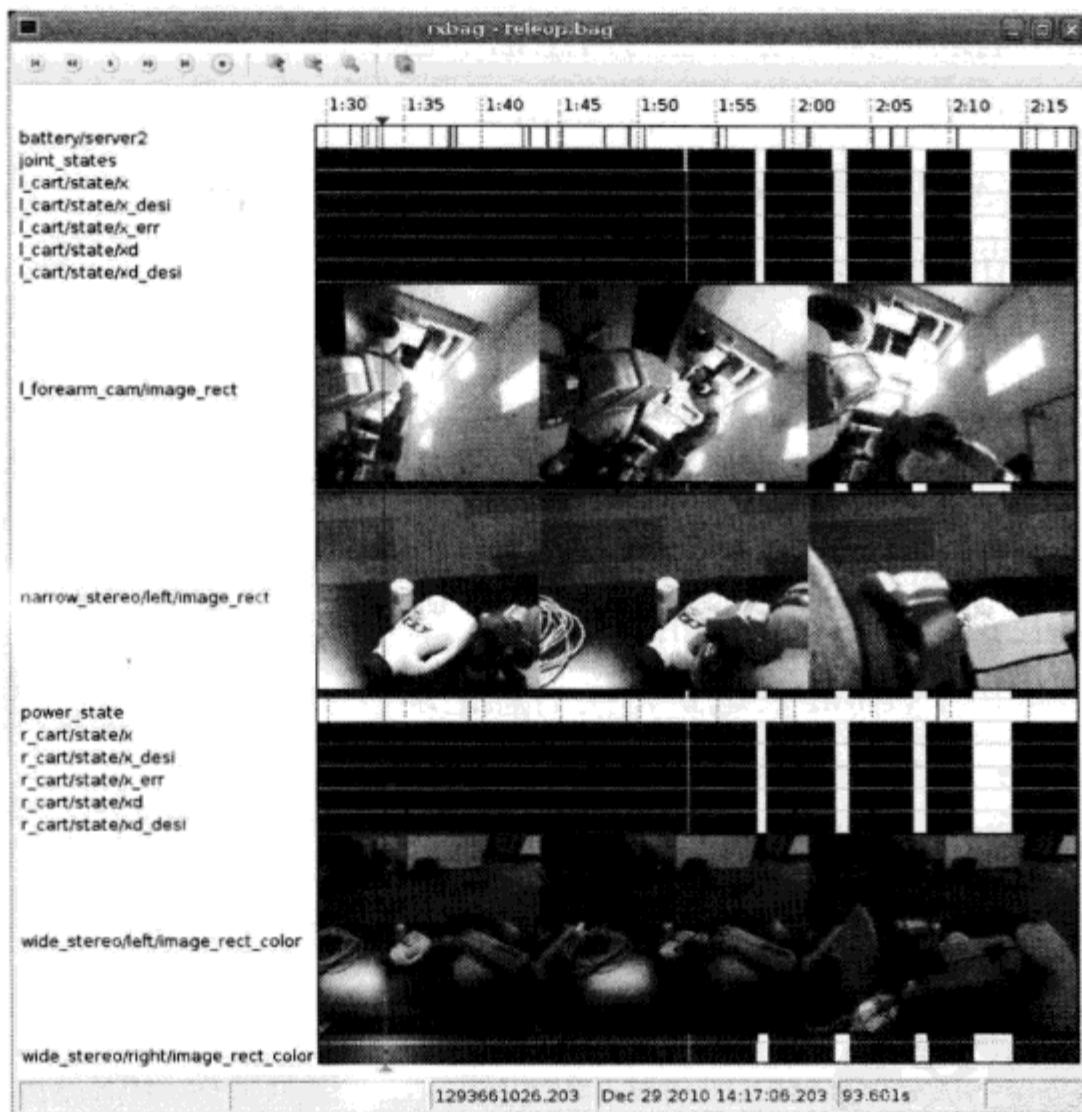


图 2.21 rx bag

在时间线窗口的上部是一个工具条，用于控制播放以及放大视图，如图 2.22 所示。在窗口的底部，是一个状态条，如图 2.23 所示。状态条从左至右，分别是状态文本、进程、时间戳、可读形式的时间、已经运行的时间、回放速度。

### 3) 消息视图

消息视图显示了关于时间线播放消息的信息。消息类型决定了哪一种视图是可见的。每一个主题有一个显示消息原始视图的选项。这里以树状显示消息的



图 2.22 rxbag 菜单栏

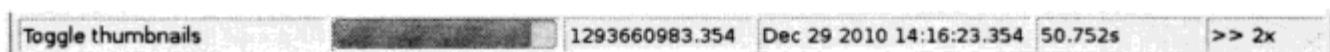


图 2.23 rxbag 状态栏

域, 如图 2.24 所示。另外, 还有两个快捷键可以使用: **Ctrl+A** 用于选择全部域, **Ctrl+C** 用于复制选中的域。

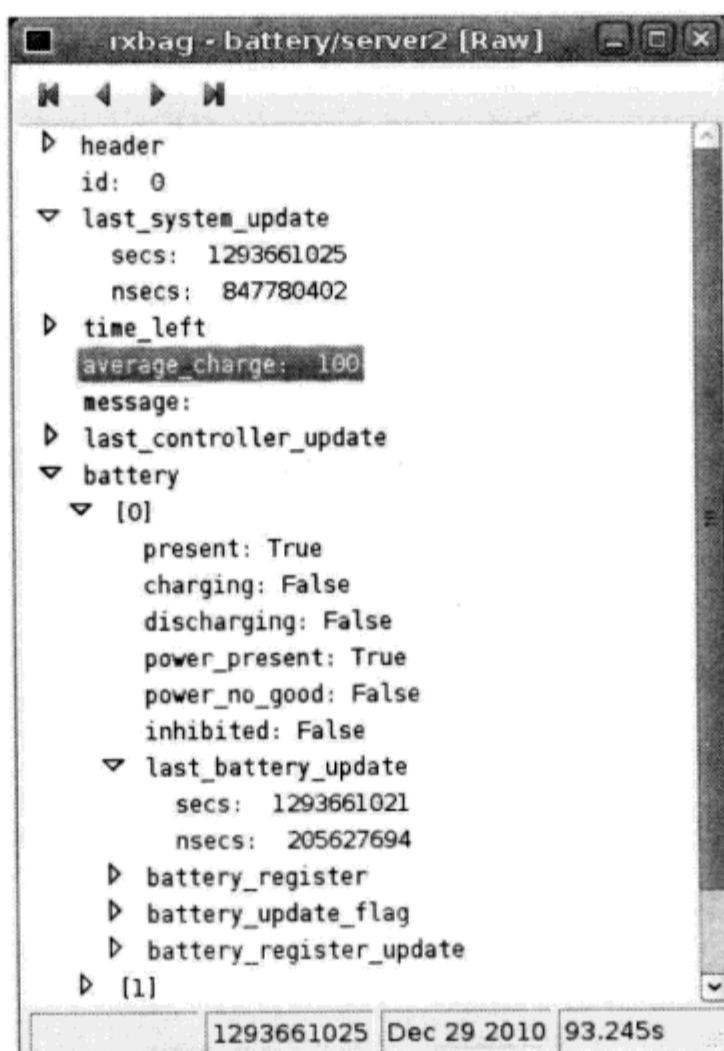


图 2.24 rxbag 树状消息域

### 2.3.3 画图工具: rxplot

rxplot 是 rxtools 功能包中用 `matplotlib` 画出正在使用一个或者多个 ROS 主题域的工具, 如图 2.25 所示。

用法: `rxplot /topic1/field1 /topic2/field2`。

例子:

(1) 在两个不同窗口画出图形:

```
rxplot /topic1/field1 /topic2/field2
```

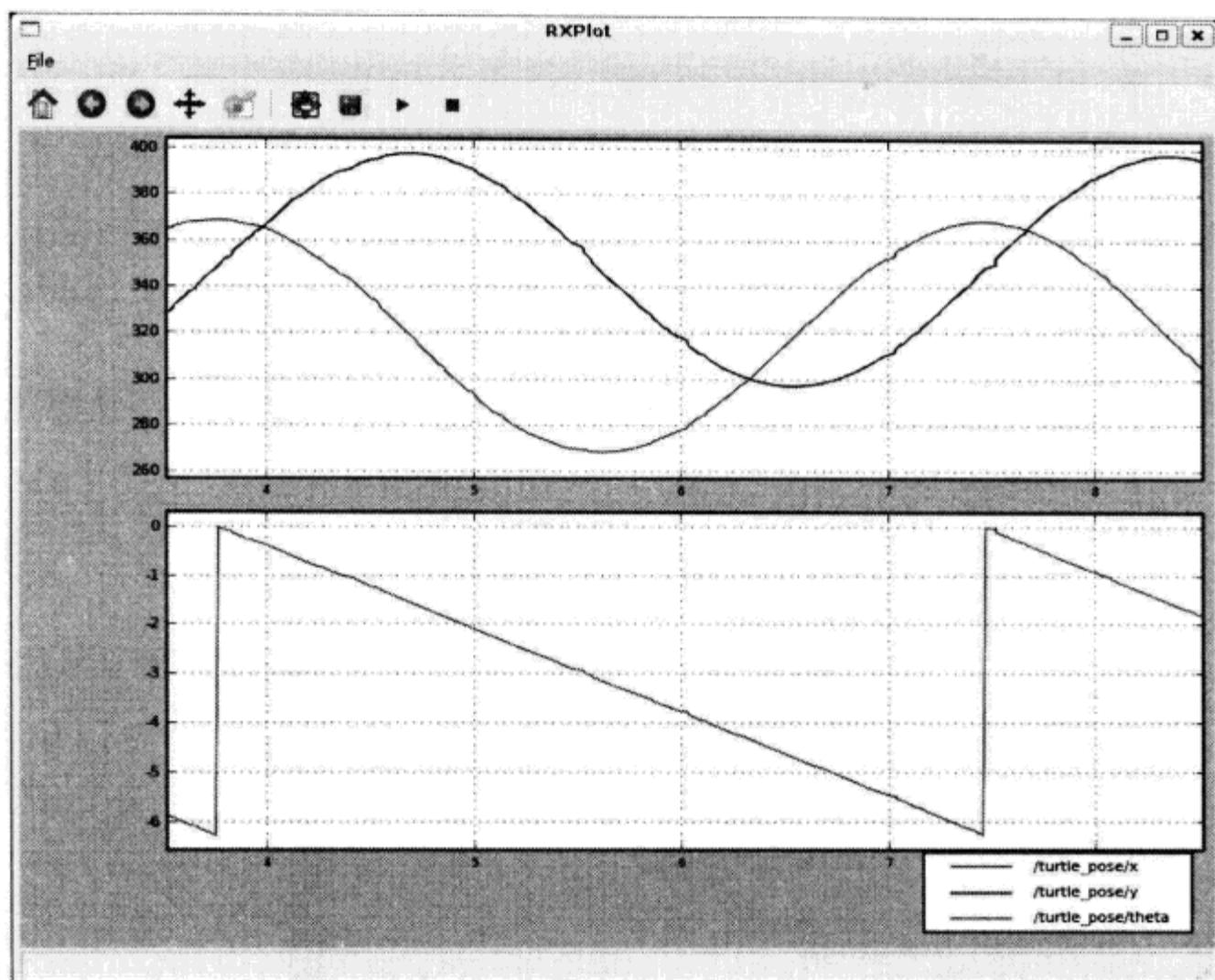


图 2.25 rxplot

(2) 在一个窗口中画出两幅图形:

```
rxplot /topic1/field1,/topic2/field2
```

(3) 画出一个消息的多个域值:

```
rxplot /topic1/field1:field2:field3
```

命令参数选项如下:

- **-l LEGEND, --legend=LEGEND**。设置 LEGEND，其中 LEGEND 是一个以逗号分隔的列表。

- **-p PERIOD, --period=PERIOD**。设置时间周期以秒显示。

- **-m MARKER, --marker=MARKER**。设置线型。

- **-t TITLE, --title=TITLE**。设置标题。

- **--mode=scatter <topic/field1> <topic/field2> [topic/field3]**。画出散乱数据。如果给出两个变量，时间作为第三个轴。

- **--mode=3d <topic/field1> <topic/field2> [topic/field3]**。画出 3D 图形。

- **-r REFRESH\_RATE, --refresh\_rate=REFRESH\_RATE**。设置刷新率 (Electric 版本中新功能)。
- **-P, --pause**。开始暂停状态 (Electric 版本中新功能)。

### 2.3.4 系统可视化工具: rxgraph

rxgraph 是以图形形式显示当前正在运行的节点和主题的工具。一个例子如图 2.26 所示。

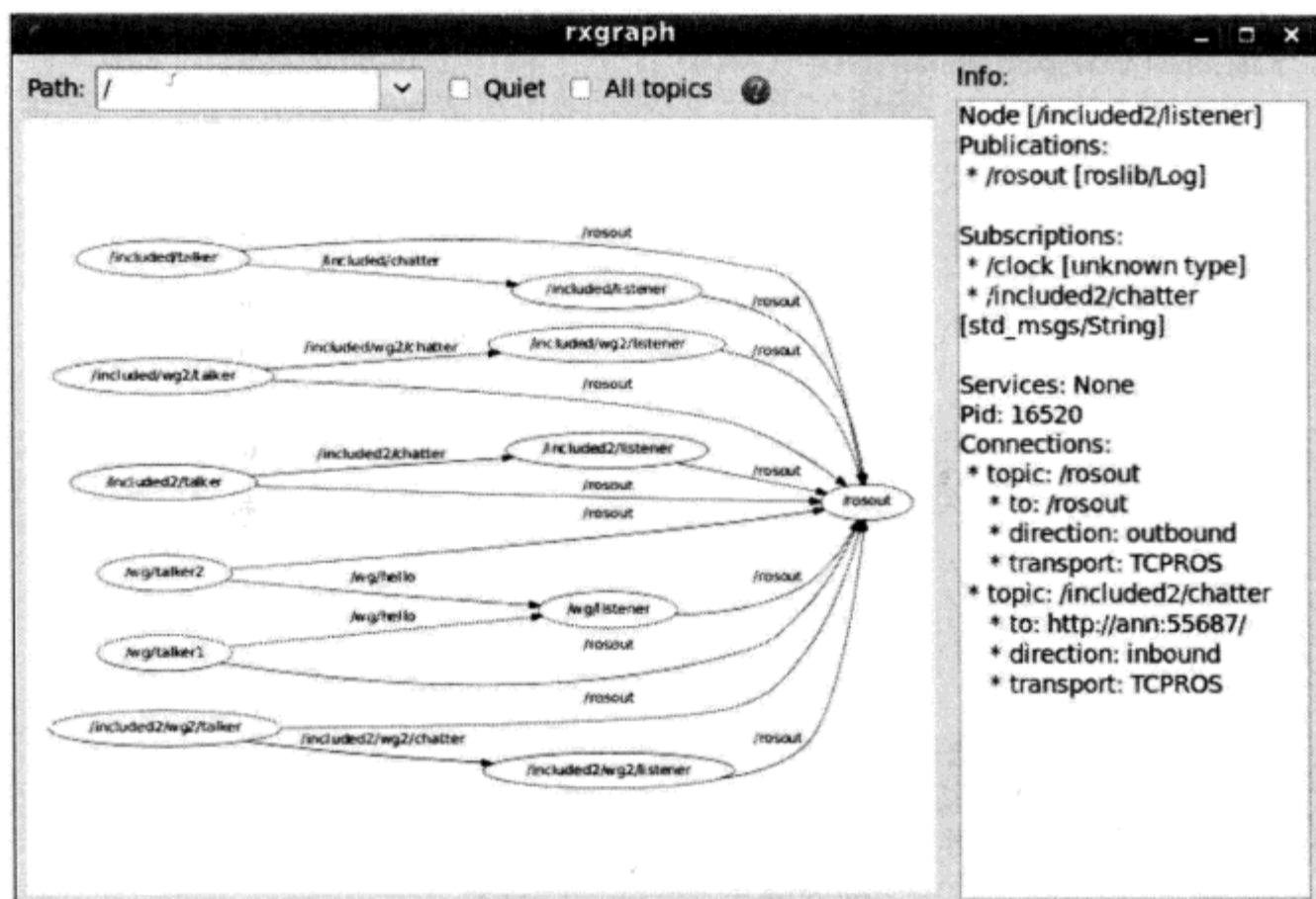


图 2.26 rxgraph 运行界面

用法: **rxgraph [options]**。

具体参数如下:

- **-h, --help** 显示帮助信息。
- **-o DOTFILE, --dot=DOTFILE** 输出图形文件并存储为 dot 文件。
- **--nodelns=NODE\_NS** 显示指定命名空间中节点。
- **--topicns=TOPIC\_NS** 显示指定命名空间中主题。

### 2.3.5 rxconsole

**rxconsole** 是 **rxtools** 包中一个消息查看工具。显示消息被发布到 **rosout**, **rxconsole** 根据时间记录所发布的消息，并提供按照重要性和文本索引两种方法

进行消息搜索。需要注意的是，C++ 和 Python 中将消息发布到 `rxconsole` 方法是完全不同的。

在运行 `turtlesim` 例子后，运行命令 `rosdep install rxtools turtlesim` 和 `rosmake rxtools turtlesim`，在两个新的终端中分别输入命令 `rxlogger-level` 和 `rxconsole`，运行结果如图 2.27 和图 2.28 所示。

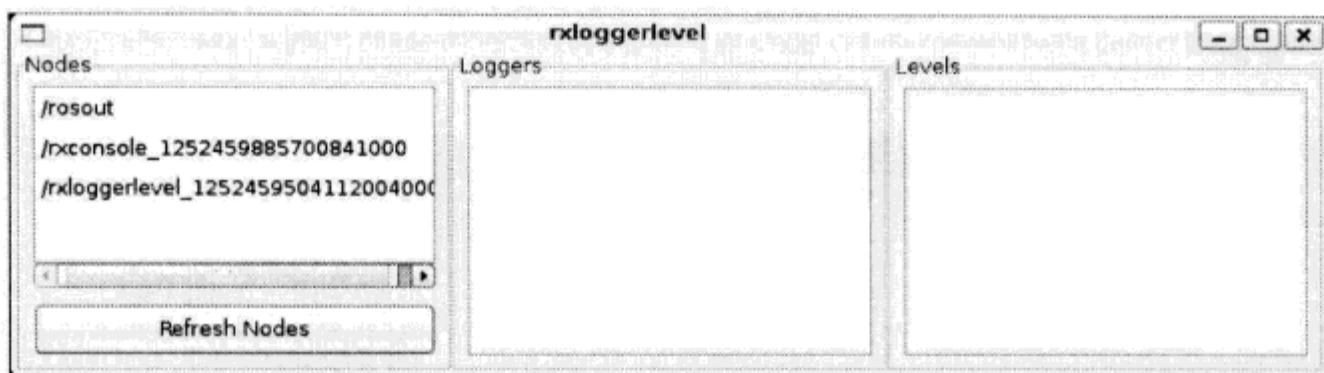


图 2.27 rxloggerlevel

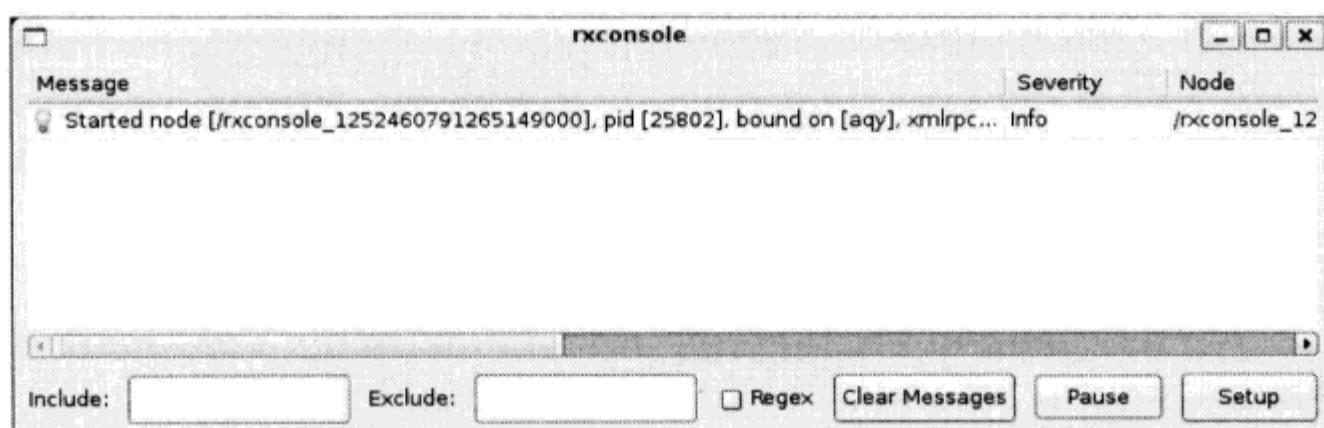


图 2.28 rxconsole 界面

缺省的日志级别为 `INFO`，在新的终端中输入命令 `rosrun turtlesim turtlesim_node`，运行结果如图 2.29 所示：其中可以看到 `turtlesim` 程序一开始运行时发布的信息。

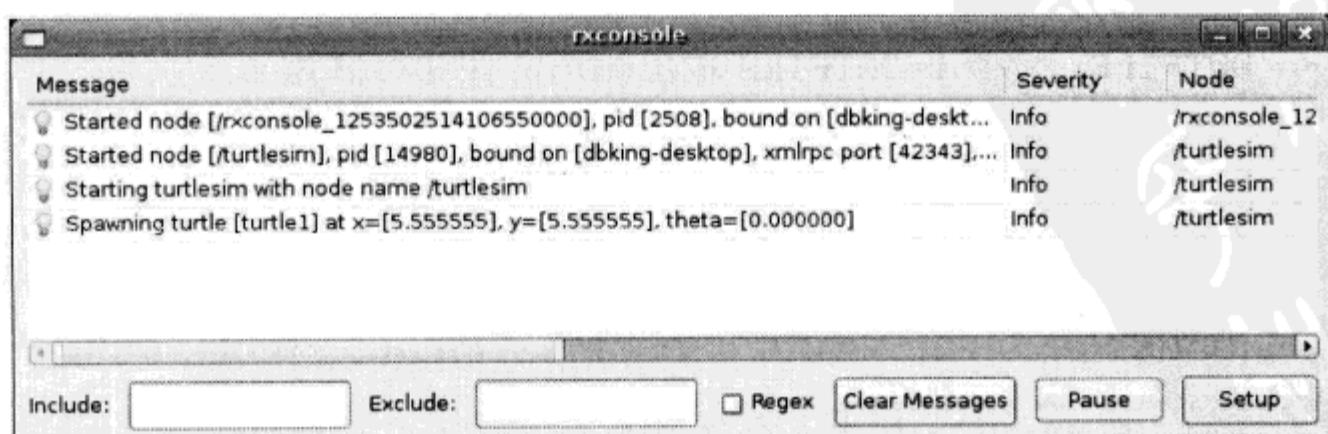


图 2.29 rxconsole Turtlesim 运行界面

当把日志级别设置为 Warn，然后刷新节点，可以在 rxloggerlevel 窗口选择 Warn，结果如图 2.30 所示。

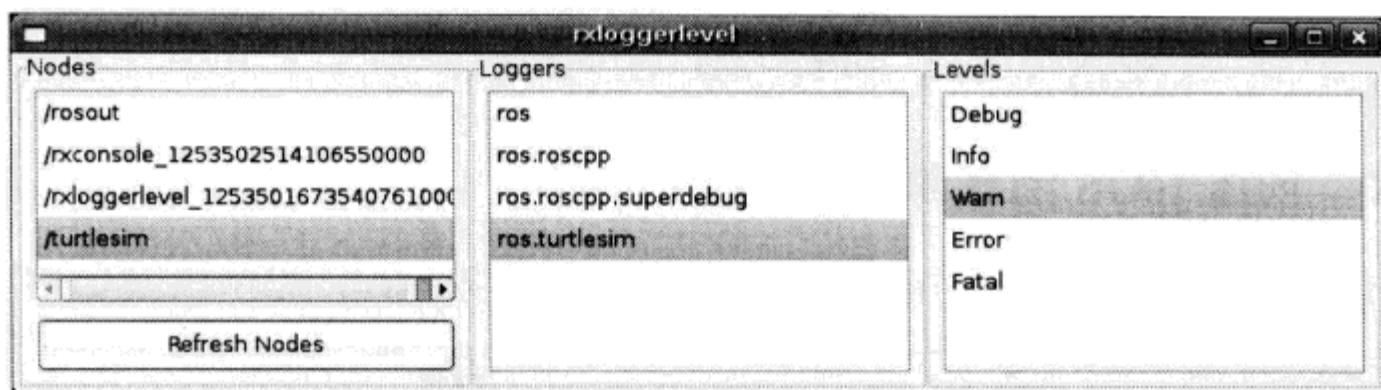


图 2.30 rxloggerlevel 日志 warn 示例

然后把小海龟运行到界面边缘，在 rxconsole 中显示信息如图 2.31 所示。

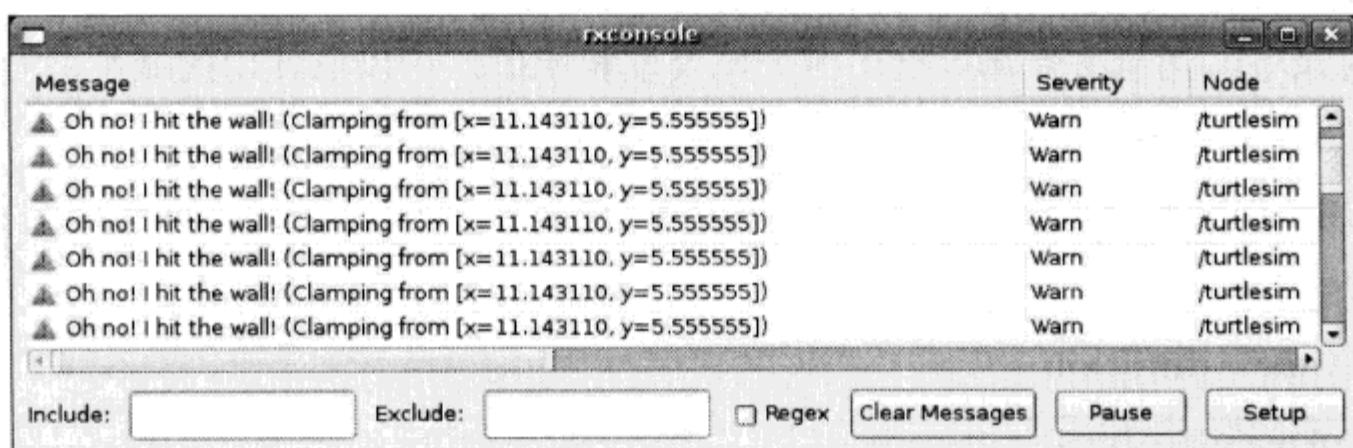


图 2.31 rxconsole Turtlesim 日志 error 示例

说明：日志级别按照下列优先级顺序：Fatal、Error、Warn、Info、Debug。其中 Fatal 是最高优先级，Debug 级别最低，通过设置日志级别，可以查看对应级别和更高一级的信息。

### 2.3.6 tf 命令

#### 1. tf echo

一个用于打印在源坐标系和目标坐标系传送信息的工具。

用法：`rosrun tf tf_echo <source_frame> <target_frame>`。

打印 /map 和 /odom 之间的传输的消息的例子：

```
rosrun tf tf_echo /map /odom
```

#### 2. view\_frames

一个可视化完整坐标树的工具。用法：

```
rosrun tf view_frames  
evince frame.pdf
```

## 2.4 例 子

### 2.4.1 创建 ROS 消息和服务

本节讲述怎样创建和编译消息 (msg) 和服务 (srv) 文件以及如何使用 `rosmsg`、`rossrv`、`roscp`、`rosmake` 命令。

#### 1. 消息和服务简介

(1) 消息文件 (扩展名为 msg) 是用于描述 ROS 消息域的简单文本文档，该文件用于生成以不同编程语言编写的源代码。msg 文件存储在一个功能包下的 msg 目录下。消息文件每行给出一个域类型和域值。可以供使用的域类型有：

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

在 ROS 中还有一个特殊的类型：`Header`，它包含了 ROS 中常用的时间戳和坐标系信息。通常来说，在消息文件第一行是这样的：`Header header`。下面是使用 `Header`、字符串和另外两个消息文件的例子：

```
1 Header header  
2 string child_frame_id  
3 geometry_msgs/PoseWithCovariance pose  
4 geometry_msgs/TwistWithCovariance twist
```

(2) 服务文件 (扩展名为 srv) 用于描述服务的文件。它由两部分组成：请求和响应。服务文件存储在一个功能包下的 srv 目录下。服务文件除了包含两部分 (这两部分以 --- 分隔) 之外，其他和消息文件是类似的：

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

上面例子中，A 和 B 是请求，Sum 是响应。

## 2. 创建消息文件

在本小节中，我们在前面创建的功能包中定义一个新的消息文件。

```
roscd beginner_tutorials  
mkdir msg  
echo "int64 num" > msg/Num.msg
```

还需要另外一步：需要确认消息文件转换为采用 C++ 或者 Python 等语言写的源代码。打开文件 CMakeLists.txt，移除下面语句中的 #：

```
# rosbuild_genmsg()
```

## 3. 使用 rosmsg

使用 rosmsg 命令创建一个消息文件，然后用 rosmsg show 命令确认 ROS 中已完成该消息文件建立：rosmsg show beginner\_tutorials/Num。运行结果：**int64 num**。

上面命令中，**beginner\_tutorials** 是消息所在的功能包，Num 是消息的名称。如果不记得功能包的名字，可以省略之：rosmsg show Num。运行结果：

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

## 4. 创建服务

下面用前面使用的功能包来创建一个服务文件：

```
roscd beginner_tutorials  
mkdir srv
```

用户可以从别的功能包文件夹拷贝一个文件过来，以代替手工创建该文件。

## 5. 使用 roscp

roscp 在从一个功能包拷贝文件到另外一个功能包时是一个非常有用的命令：

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

还需要另外一步：需要确认服务文件转换为采用 C++ 或者 Python 等语言写的源代码。打开文件 CMakeLists.txt，移除下面语句中的 #：

```
#rosbuild_gensrv()
```

## 6. 使用 rossrv

使用 rossrv 命令创建一个服务文件，然后用 rossrv show 命令确认 ROS 中已完成该服务文件建立：

```
rossrv show beginner_tutorials/AddTwoInts
```

结果如下：

```
int64 a
```

```
int64 b
```

```
---
```

```
int64 sum
```

由于增加了新的信息，因此需要重新编译功能包：

```
rosmake beginner_tutorials
```

## 7. 获得帮助信息

虽然已经有一些相关的 ROS 命令可以使用，但是如果追踪命令的相关参数比较困难，可以采用一些参数获取命令帮助：`rosmsg -h`。运行结果：

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
```

同样地，也可以获取相关子命令的帮助信息：

```
rosmsg show -h
```

运行结果：

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

Options:

```
-h, --help show this help message and exit
```

```
-r, --raw show raw message text, including comments
```

### 2.4.2 记录和回放数据

本节以小海龟为例，讲述怎样将运行中的 ROS 系统记录保存在记录文件 (.bag) 中，并使保存下来的文件回放时，得到类似的结果。

## 1. 运行前准备工作

安装 `turtle_teleop` 功能包和其相关的功能包，命令如下：

```
roscd turtle_teleop  
rosdep install turtle_teleop  
rosmake turtle_teleop
```

安装好 `turtle_teleop` 功能包的依赖项，然后生成 `turtle_teleop` 和所有没有生成的依赖项。

## 2. 记录数据 (生成消息记录包文件)

首先，执行下面两条命令，以启动键盘控制小海龟命令：

```
roscd turtle_teleop  
roslaunch launch/turtle_keyboard.launch
```

此处会执行两个节点，`turtlesim` 可视化节点和允许键盘执行节点。将鼠标放在终端，用键盘方向键控制 `turtlesim` 时，终端会生成：

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

使用 `rostopic` 命令还可以记录所有发布的主题。首先，检测所有正在发布的主题列表。打开新的终端，执行下面命令：`rostopic list -v`，得到输出结果：

```
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
```

这里所列举的主题只是消息类可能被记录的数据日志文件，因为只有发送的消息可以被记录。主题 `/turtle1/command_velocity` 是 `turtle_teleop` 发送的消息，输入到 `turtlesim` 进程。`/turtle1/color_sensor` 和 `/turtle1/pose` 消息是 `turtlesim` 发送的消息输出。

然后，开始记录发送的数据。打开新的终端，执行下面命令：

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

这里，创建一个临时目录来记录数据，然后采用 `-a` 选项，表示所有的发送主题都将记录在消息记录包文件中。将鼠标放到 `turtlesim` 终端上，运行小海龟的例子 10s 左右，用 `Ctrl+C` 强制关闭 `rosbag` 终端，查看 `~ /bagfiles`，可以看到一个以年、月、日和创建时间命名的 `bag` 文件。这个文件包含在 `rosbag record` 运行时所有节点发送的所有主题中。

### 3. 检查和运行消息记录包文件

用 `rosbag record` 记录发送数据，也可以检查它并采用命令 `rosbag info` 和 `rosbag play` 将它重现出来。

首先来查看消息记录包文件记载的数据，此时可以采用 `info` 命令检测消息记录包文件的内容：

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

得到在当前运行的计算机上生成的信息如下：

```
rosbag info 2011-02-21-17-06-35.bag
path:          2011-02-21-17-06-35.bag
version:       2.0
duration:     6:36s (396s)
start:         Feb 21 2011 17:06:36.76 (1298279196.76)
end:          Feb 21 2011 17:13:13.23 (1298279593.23)
size:          2.4 MB
messages:      36462
compression:   none [3/3 chunks]
types:          roslib/Log      [acffd30cd6b6de30f120938c17c593fb]
                turtlesim/Color [353891e354491c51aab32df673fb446]
                turtlesim/Pose  [863b248d5016ca62ea2e895ae5265cf9]
                turtlesim/Velocity
                           [9d5c2dcd348ac8f76ce2a4307bd63a13]
topics:         /rosout           4 msgs @ 323.2 Hz :
                           roslib/Log (2 connections)
                           /turtle1/color_sensor 18189 msgs @ 63.3 Hz :
```

```
turtlesim/Color  
/turtle1/command_velocity 80 msgs @ 24.8 Hz :  
turtlesim/Velocity  
/turtle1/pose 18189 msgs @ 63.7 Hz :  
turtlesim/Pose
```

这些信息告诉用户在消息记录包文件中存储的主题名字、类型、每个消息主题包含的数目。

下一步，运行这个消息记录包文件，重现运行系统的行为。首先，要关闭掉所有 turtlesim 可能运行的节点（在启动 `turtle_keyboard.launch` 的终端运行 `Ctrl+C`），然后在新的终端运行下命令：`rosrun turtlesim turtlesim_node`。

这个命令启动 turtlesim 节点，但没有触发键盘节点，可以用保存的消息记录包文件回放运行过程。在新的终端运行命令：`rosbag play 2011-02-21-17-06-35.bag`。运行结果如下：

```
[INFO] [1298281122.467928564]: Opening 2011-02-21-17-06-35.bag  
Waiting 0.2 seconds after advertising topics... done.  
Hit space to toggle paused, or 's' to step.  
[RUNNING] Bag Time: 1298279593.219138 Duration: 396.463736  
/396.479350  
Done.
```

会看到运行的结果与之前所运行的结果完全一样，在默认的模式下，发送消息时间间隔为 2s。时间间隔可以用 `-d` 选项来设置。一般地，`/turtle1/command_velocity topic` 将被发送，`turtle` 将在 turtlesim 运行，类似已经在 teleop 程序中运行的模式。运行 `rosbag play` 和 `turtle` 移动的间隔时间，应当大致等于原始 `rosbag record` 执行的时间。

也可以让 `rosbag play` 不从消息记录包文件开始，而是从一些已经开始的间隔中，采用 `-s` 来设置。最后的选项是 `-r`，该参数是用来指定发送率的。比如，如果执行下面命令：`rosbag play -r 2 <your bagfile>`，会发现 `turtle` 执行轨迹稍微不同，这是因为设置从键盘触发 2 倍速度运行。

#### 4. 记录数据子集

当运行完整的系统时，比如 PR2 软件系统，可能有上百个主题被发布。比如视频图像流，潜在的可发送的巨大数据。在这样的系统中，常常需要记录包括所有主题的日志文件到硬盘中。`rosbag record` 命令只支持一定的主题保存到消

息记录包文件，运行用户记录感兴趣的主題。如果 `turtlesim` 节点已经退出，重新启动键盘触发节点：

```
roscd turtle_teleop  
roslaunch launch/turtle_keyboard.launch
```

在消息记录包文件目录下，运行以下命令：

```
rosbag record -0 subset /turtle1/command_velocity /turtle1/pose
```

参数 `-0` 告诉 `rosbag record` 记录的日志文件名为 `subset.bag`，并且主题参数引起 `rosbag record` 仅仅订阅 2 个主题。触发键盘移动 `turtle` 几秒后，用 `Ctrl+C` 终止 `rosbag record`。然后查看消息记录包文件 (`rosbag info subset.bag`)。运行的结果如下：

```
path:          subset.bag  
version:       2.0  
duration:     1:44s (104s)  
start:        Feb 21 2011 19:59:57.04 (1298289597.04)  
end:          Feb 21 2011 20:01:41.27 (1298289701.27)  
size:         370.0 KB  
messages:     4796  
compression:  none [1/1 chunks]  
types:        turtlesim/Pose [863b248d5016ca62ea2e895ae5265cf9]  
                  turtlesim/Velocity  
                           [9d5c2dcd348ac8f76ce2a4307bd63a13]  
topics:       /turtle1/command_velocity  44 msgs @ 0.6  
                           Hz:turtlesim/Velocity  
/turtle1/pose           4752 msgs @ 64.0  
                           Hz:turtlesim/Pose
```

## 5. rosbag record/play 的限制

通过运行上述例子，用户可能发现，对照 `turtlesim` 运行的轨迹和原始轨迹，并不是完全一样的。这是因为 `turtlesim` 运行的轨迹，对系统运行时间非常敏感。`rosbag` 在消息被记录和处理的同时，在消息被 `rosplay` 产生和处理的同时，不能精确的复制系统运行的行为。对于 `turtlesim` 节点，在当命令消息被处理时有轻微变动，用户不可能期望完全精确地复制该行为。

### 2.4.3 手工创建 ROS 功能包

命令 `roscreate-pkg` 用于创建 ROS 功能包，这个命令可以防止一些错误并且可以有效保存。功能包只不过是一个文件夹和一个简单的 XML 文件。

下面来创建一个新的 `newpkg` 功能包，当前假设在 `ROS_PACKAGE_PATH` 下的 `pkgs` 目录下工作，运行命令：

```
mkdir newpkg  
cd newpkg
```

随后需要做的是添加 `manifest` 文件。`manifest.xml` 文件允许像 `rospack` 之类的命令确定该功能包的依赖信息。

在文件 `newpkg/manifest.xml` 内，添加如下代码：

```
1 <package>  
2   <description brief="example package tutorial">A new package  
3   </description>  
4   <author>Your Name Here</author>  
5   <license>BSD</license>  
6   <depend package="roscpp" />  
7   <depend package="std_msgs" />  
8 </package>
```

随后执行命令： `rospack find newpkg`，运行结果：

```
/home/user/ros/pkgs/newpkg
```

这说明 ROS 已经发现该功能包。

为了说明指定依赖项是有用的，执行下面命令：

```
rospack export --lang=cpp --attrib=cflags foobar  
rospack export --lang=cpp --attrib=lflags foobar
```

上述命令执行后，`rospack` 查找 `newpkg` 的依赖项，生成一个列表，其中包含或者链接相关声明以便用于编译和链接可执行文件。这些命令用于 ROS 系统正确地编译和链接功能包。实际上，由于系统自动维护相关文件，几乎不直接使用这些文件。

在 `newpkg/Makefile` 文件内，添加如下代码：

```
include $(shell rospack find mk)/cmake.mk
```

这个命令告诉 `make` 命令将要使用 CMake 而不是 `make` 来编译功能包。

由于编译多个交叉平台时，CMake 命令更具弹性，因此 ROS 使用 CMake 进行编译。在 `newpkg/CMakeLists.txt` 文件内，添加如下代码：

```

cmake_minimum_required(VERSION 2.4.6)
include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
rosbuild_init()

```

上述内容是在 ROS 内开始创建一个功能包所需的。

#### 2.4.4 大项目上运行 roslaunch

本节主要讲述用于建立大型项目时编写 `roslaunch` 文件所需的一些内容。其中重点在如何构建启动文件，以使得在不同环境下可以重复使用。这里以 `2dnav_pr2` 为例进行讲解。

##### 1. 简介

机器人上的大型应用往往涉及几个中间连接的节点，每个节点都会有很多参数。`2D` 导航是一个很好的例子。`2dnav_pr2` 应用包括 `move_base`、`localization`、`ground plane filtering`、`the base controller`、`the map server`。它们总共有几百个参数可以影响这些节点的行为。当然，也需要有些约束，像平面滤波必须在同一台机器上执行以便激光能够有效执行。

一个 `roslaunch` 文件即可实现上述功能。给定一台正在运行的机器人，启动 `2dnav_pr2` 功能包中的文件 `2dnav_pr2.launch`，即可实现机器人导航所需的内容。

通常希望 `roslaunch` 文件能够尽可能重用。在这种情况下，相同配置的机器人可以不需要修改文件即可使用。即使是从机器人移植到模拟器上面，也只需做很少修改即可。

##### 2. 上层组织

文件 `2dnav_pr2/move_base/2dnav_pr2.launch` 的源代码如下：

```

1 <launch>
2   <group name="wg">
3     <include file="$(find pr2_alpha)/$(env ROBOT).machine"
4       />
5     <include file="$(find 2dnav_pr2)/config/new_amcl_node.
6       xml" />
7     <include file="$(find 2dnav_pr2)/config/
8       base_odom_teleop.xml" />
9     <include file="$(find 2dnav_pr2)/config/
10      lasers_and_filters.xml" />

```

```
7      <include file="$(find 2dnav_pr2)/config/map_server.xml"
8          " />
9
10     <!-- The navigation stack and associated parameters -->
11     <include file="$(find 2dnav_pr2)/config/ground_plane.
12         xml" />
13   </group>
14 </launch>
```

这个文件包含了对其他文件的设置。每一项设置包含了节点和参数，用于完成某些功能，像定位、传感器处理和路径规划。

说明：上层启动文件应该简短，并且包含了其他文件中对应的部分应用或者是 ROS 中修改过的参数。

为了运行 PR2 机器人，首先需要构架一个核心，然后编写一个针对机器人的特定启动文件，像 `pr2_alpha` 功能包中的 `pre.launch` 文件，然后启动文件 `2dnav_pr2.launch`。这里包含了一个启动文件，而不是分别启动它们，这样实现了折中：

- (1) 优点：需要遵守尽量少开新终端的原则。
- (2) 缺点：启动 `launch` 文件，将会初始化一个持续大概一分钟的标定阶段。如果 `2dnav_pr2` 启动文件包含了机器人启动文件，每次运行需要结束 `roslaunch` 文件，然后再打开，标定需要重新进行。
- (3) 缺点：一些 2D 的导航节点在节点开始前标定已经完成。`roslaunch` 将特意不提供任何控制序列或者节点开始的时间序列。理想的解决方案是等到标定结束后再执行这些节点。因此需要把这些放到两个文件中，等到标定结束后，启动 `2dnav`。

### 3. 机器标签和环境变量

在有多台电脑同时运行 ROS 时，可能需要控制哪一个节点在哪一台机器上面运行，以便平衡资源和带宽。例如，可能需要 `amcl` 节点和 `base laser` 节点运行在同一台机器上。同时，为了重用，不要把机器名称写入启动文件中。`roslaunch` 采用机器标签来处理这种情况。

首先把下面代码包含进去：

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

首先需要注意的是上面命令为了使用机器人的环境变量的值而采用 `env` 参量。例如，执行 `roslaunch` 之前执行命令：`export ROBOT=pre`，将使得 `pre.machine` 被包含进来。

说明：采用 `env` 参量允许启动文件的一部分依赖于环境变量。

然后来看 `pr2_alpha` 功能包下的文件 `pre.machine` 的例子：

```
1 <launch>
2   <machine name="c1" address="pre1" ros-root="$(env
3     ROS_ROOT)" ros-package-path="$(env ROS_PACKAGE_PATH)"
4     default="true" />
5   <machine name="c2" address="pre2" ros-root="$(env
6     ROS_ROOT)" ros-package-path="$(env ROS_PACKAGE_PATH)"
7     />
8 </launch>
```

这个文件在逻辑机器名称 `c1` 和 `c2` 之间设置了映射，最终的主机名称可能是 `pre2`。它甚至允许你作为用户登录。

一旦映射定义，当启动节点时，即可使用。例如，在 `2dnav_pr2` 功能包中文件 `config/new_amcl_node.xml` 包含下列语句。该命令导致节点 `amcl` 运行在逻辑名称为 `c1` 的机器上。

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

当在一台新的机器上运行时，例如名称为 `prf`，只需要修改机器人环境变量即可。对应的机器人文件随后被装载。可以通过设置机器人为 `sim` (`export ROBOT=pre`) 让这些文件运行于模拟器上。查看 `pr2_alpha` 功能包下的文件 `sim.machine`，可以看到它只是把所有的逻辑机器名称映射到局域主机。

说明：机器标签用于平衡装载和控制节点使其运行于同一台机器。

#### 4. 参数、命名空间和 YAML 文件

查看文件 `move_base.xml`，下面列出其中部分代码：

```
1 <node pkg="move_base" type="move_base" name="move_base"
2   machine="c2">
3   <remap from="odom" to="pr2_base_odometry/odom" />
4   <param name="controller_frequency" value="10.0" />
```

```

4   <param name="footprint_padding" value="0.015" />
5   <param name="controller_patience" value="15.0" />
6   <param name="clearing_radius" value="0.59" />
7   <rosparam file="$(find 2dnav_pr2)/config/
        costmap_common_params.yaml" command="load" ns=
        "global_costmap" />
8   <rosparam file="$(find 2dnav_pr2)/config/
        costmap_common_params.yaml" command="load" ns=
        "local_costmap" />
9   <rosparam file="$(find 2dnav_pr2)/move_base/
        local_costmap_params.yaml" command="load" />
10  <rosparam file="$(find 2dnav_pr2)/move_base/
        global_costmap_params.yaml" command="load" />
11  <rosparam file="$(find 2dnav_pr2)/move_base/navfn_params.
        yaml" command="load" />
12  <rosparam file="$(find 2dnav_pr2)/move_base/
        base_local_planner_params.yaml" command="load" />
13 </node>

```

上面代码用于启动 `move_base` 节点。首先被包含的是 `remapping`。`move_base` 节点设计用来在主题 `odom` 上接收里程信息。在机器人系统为 PR2 的情形，里程信息在主题 `pr2_base_odometry` 上发布，因此我们重新映射它。

说明：当给定类型的信息被发布在不同情况下的不同主题时，用主题映射来解决。

随后有一系列参数标签。这些参数都在节点元素里面，因此它们是私有参数。在参数标签之后，有若干 `rosparam` 项，这些是在 `yaml` 中的可读参数。下面列出第一项 `rosparam` 装载文件 `costmap_common_params.yaml` 部分代码：

```

1 raytrace_range: 3.0
2 footprint: [[-0.325, -0.325], [-0.325, 0.325], [0.325,
    0.325], [0.46, 0.0], [0.325, -0.325]]
3 inflation_radius: 0.55
4
5 # BEGIN VOXEL STUFF
6 observation_sources: base_scan_marking base_scan tilt_scan
    ground_object_cloud
7
8 base_scan_marking: {sensor_frame: base_laser, topic:
    /base_scan_marking, data_type: PointCloud,

```

```
expected_update_rate: 0.2, observation_persistence:
0.0, marking: true, clearing: false,
min_obstacle_height: 0.08, max_obstacle_height: 2.0}
```

可见，yaml 可以写成向量形式（用于轨迹参数），它也允许把一些参数放到一个嵌套的命名空间内。例如：base\_scan\_marking/sensor\_frame 被设置为 base\_laser。注意到这些命名空间是相对于 yaml 文件自己的命名空间来定义的。在 yaml 文件中它以包含 rosparam 的具有  $n$  个属性的形式被声明为 global\_costmap。反之，既然 rosparam 被包含在节点元素中，参数的完整有效名称是 /move\_base/global\_costmap/base\_scan\_marking/sensor\_frame。

在 move\_base.xml 中，下一行代码是：

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

这实际上包含了相同的 yaml 文件，就像之前一行代码那样。它只不过在不同的命名空间中。这比要重新定义所有值的类型要好多了。

下一行代码：

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

不像前面的例子，这一项没有  $n$  个属性。因此这个 yaml 文件的命名空间是父命名空间 /move\_base。但是，查看一下 yaml 文件的前几行，可以看到参数在 /move\_base/local\_costmap 命名空间内。

```
1 local_costmap:
2   #Independent settings for the local costmap
3   publish voxel_map: true
4   global_frame: odom_combined
5   robot_base_frame: base_link
```

## 5. 启动文件重用

上面的很多说明是为了在不同情况下使得重用启动文件更加容易。我们已经看到一个采用 env 环境变量来修改行为而不必改变任何启动文件的例子。然而，还有很多情况使得上述方法不方便或者不可能实现。下面来看 pr2\_2dnav\_gazebo 功能包。这里包含了一个 2D 导航应用的版本，但是它用于

**Gazebo simulator** 模拟。如果用于导航，由于我们使用的 **Gazebo** 环境基于不同的静态地图，因此 **map\_server** 节点必须用一个不同的变量来装载。这里可以采用另外一个 **env** 变量。但是用户可能需要设置一系列环境变量以便于能够启动 **roslaunch**。另外一种方法，**2dnav\_gazebo** 包含了它自己的上层启动文件 **2dnav-stack-amcl.launch**:

```

1 <launch>
2   <include file="$(find pr2_alpha)/sim.machine" />
3   <include file="$(find 2dnav_pr2)/config/new_amcl_node.xml
4     " />
5   <include file="$(find 2dnav_pr2)/config/base_odom_teleop.
6     xml" />
7   <include file="$(find 2dnav_pr2)/config/
8     lasers_and_filters.xml" />
9   <node name="map_server" pkg="map_server" type="map_server
10    " args="$(find gazebo_worlds)/Media/materials/
11      textures/map3.png 0.1" respawn="true" machine="c1" />
12   <include file="$(find 2dnav_pr2)/config/ground_plane.xml
13     " />
14   <!-- The naviagtion stack and associated parameters -->
15   <include file="$(find 2dnav_pr2)/move_base/move_base.xml
16     " />
17 </launch>

```

第一个不同之处在于，既然知道机器人是在模拟器环境中，只需要用 **sim.machine** 文件即可，而不必用替代变量。第二个不同之处下面一行代码：

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

被替换成：

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

上面代码第一项里面包含了一个节点声明，该声明用于第二项，但采用了一个不同的地图文件。

说明：为了修改一个应用的上层的部分功能，只需要拷贝别的上层文件到该文件中并改变需要的部分即可。

## 6. 参数覆盖

上面描述的技术有时候不方便实现。假如使用 `2dnav_pr2`, 将局部代价地图的分辨率参数改为 0.5, 只需要对 `local_costmap_params.yaml` 进行局部改变。这只是临时改变的最简单形式, 但这意味着不能返回检查修改过的文件。但是可以做一个 `local_costmap_params.yaml` 备份, 并修改这个备份文件。因此, 需要修改 `2dnav_pr2.launch` 以包含修改的 `move_base.xml` 文件, 再改变 `move_base.xml` 以包含修改过的 `yaml` 文件。如果使用版本控制, 就不能看到对原始文件的修改了, 因此很耗时。

一种替代的方案是重构启动文件, 以使得参数 `move_base/local_costmap/resolution` 定义在上层文件 `2dnav_pr2.launch` 中, 这样只是修改该文件即可。如果事先知道哪一个参数需要修改, 那么这是一个好的选择。

另一个选择是用 `roslaunch` 的覆盖行为: 在处理完 `include` 文件之后, 按照顺序设置参数。这样, 可以使得新的上层文件覆盖原来的分辨率。

```
1 <launch>
2   <include file="$(find 2dnav_pr2)/move_base/2dnav_pr2.launch
3     "/>
4   <param name="move_base/local_costmap/resolution" value="0.5
5     "/>
6 </launch>
```

这种方法的缺点在于使得事情更难以理解: 要知道最终的值, 需要通过包含 `roslaunch` 文件来追踪。但是它确实避免了拷贝多个文件的麻烦。

说明: 为了修改一个文件树中的嵌套参数, 采用 `roslaunch` 参数覆盖方法。

## 7. roslaunch 变量

对于 C Turtle 版本来说, `roslaunch` 已经有带标签的替代变量的特性, 用于对启动文件的部分变量的值进行修改。这是比参数覆盖机制或者采用启动文件重用方式更加通用和清楚的一种方式。

### 2.4.5 在多台机器上运行 ROS 系统

本节说明怎样用两台机器运行一个 ROS 系统。其中揭示了如何用 `ROS_MASTER_URI` 命令实现一个节点管理器配置多台机器。

#### 1. 概览

ROS 是采用分布式计算思想设计以使得它在运行时不需要事先知道网络信

息。它允许在运行时被重载以匹配可获取的资源。在多台机器上运行一个 ROS 系统是很容易的。有几点需要注意：

- (1) 只需要一个节点管理器，选择其中一台机器运行节点管理器。
- (2) 通过 `ROS_MASTER_URI`，所有节点必须配置为使用同一个节点管理器。
- (3) 机器之间必须是在所有的端口上完整的双向连接。
- (4) 每个机器必须通过一个别的机器可以解析的名称进行广播。

## 2. 在两台机器上运行消息发布者/接收者系统

本小节给出一个例子：在两台机器上运行一个名为 `marvin` 和 `hal` 的消息发布者/接收者 (`talker/listener`) 系统。`marvin` 和 `hal` 是机器的主机名称，即登录机器时候需要的用户名。登陆 `marvin` 时，需要输入：`ssh marvin`。对 `hal`，情况是类似的。

### 1) 运行节点管理器

需要选择一台机器来运行节点管理器，这里选 `hal`，第一步就是登陆，然后启动 ROS：

```
ssh hal  
roscore
```

### 2) 运行消息接收者 (listener)

下面在主机 `hal` 上运行消息接收者并配置 `ROS_MASTER_URI`，使得可以使用刚才运行的节点管理器：

```
ssh hal  
export ROS_MASTER_URI=http://hal:11311  
rosrun rospy_tutorials listener
```

### 3) 运行消息发布者 (talker)

下一步在机器 `marvin` 上运行消息发布者，也需要配置 `ROS_MASTER_URI` 以便 `hal` 上正在运行的节点管理器可以使用：

```
ssh marvin  
export ROS_MASTER_URI=http://hal:11311  
rosrun rospy_tutorials talker
```

现在可以看到在 `hal` 上的消息接收者正在接收在 `marvin` 上的消息发布者发送的消息。

上面节点开始的顺序没有影响。唯一需要注意的是，在开始任何节点前先启动节点管理器。

#### 4) 改变：反向连接

下面试一下反向连接的效果。保留运行在 `hal` 上的节点管理器，结束消息发布者和消息接收者，然后互换程序所在机器。

首先在 `marvin` 上运行消息接收者：

```
ssh marvin
export ROS_MASTER_URI=http://hal:11311
rosrun rospy_tutorials listener
```

然后在 `hal` 上运行消息发布者：

```
ssh hal
export ROS_MASTER_URI=http://hal:11311
rosrun rospy_tutorials talker
```

### 2.4.6 定义客户消息

本小节展示如何采用 ROS 消息描述语言定义自己的客户消息数据类型。

#### 1. 生成消息

生成消息非常容易。在一个功能包的 `msg` 文件夹下放一个 `.msg` 文件即可。然后在文件 `CMakeLists.txt` 中，在 `rospack()` 之后添加 `genmsg()`：

```
1 cmake_minimum_required(VERSION 2.4.6)
2 include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
3 rospack(my_package)
4 genmsg()
```

消息文件可能需要下述内容：

```
1 string first_name
2 string last_name
3 uint8 age
4 uint32 score
```

任何在 `msg` 文件夹下的 `.msg` 文件将生成用于任何所支持的语言的代码。在消息描述语言中，将会有关于消息格式的完整说明。

## 2. 导出消息

如果其他功能包不能接收，消息是没有用的，因此消息必须以指定语言明确导出。下面是 C++ 语言的例子。

如果采用 C Turtle 或者更新版本，这一步已经不需要了。在 Box Turtle 版本中，必须通过 `manifest` 导出标签来导出消息。如果没有 C++ 导出标签，可以以下面格式导出消息：

```
1 <export>
2   <cpp cflags="-I${prefix}/msg/cpp"/>
3 </export>
```

## 3. 包含/导入服务

(1) C++ 语言。消息被放在能够与功能包名称匹配的命名空间下，例如：

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

(2) Python 语言。具体内容如下：

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



# 第三章 ROS 客户端库

## 3.1 概述

ROS 客户端库提供了一系列库文件以便于用户开发。它利用了很多 ROS 概念并使得它们通过代码可以获取。表 3.1 给出了 ROS 中应用程序接口 (API) 的列表。

表 3.1 ROS API 列表

API	ROS	C++	Python
ROS	ROS	roscpp	rospy
基本数据类型	common_msgs	common_msgs	common_msgs
操作消息流	topic_tools	message_filters	message_filters
驱动器	joystick_drivers, camera_drivers, laser_drivers, sound_drivers, imu_drivers	joystick_drivers, camera_drivers, laser_drivers, sound_drivers, imu_drivers	
驱动器执行	driver_common	driver_common	
滤波数据		filters	
三维处理	laser_pipeline, point_cloud_perception	laser_pipeline, point_cloud_perception	
图像处理		image_common, image_pipeline, vision_opencv	vision_opencv
坐标变换		tf, tf_conversions, robot_state_publisher	tf, tf_conversions
动作	actionlib	actionlib	actionlib
执行/任务管理器	executive_smach		executive_smach
导航	navigation	via actionlib	via actionlib
二维模拟	simulator_stage	simulator_stage	
三维模拟	simulator_gazebo	simulator_gazebo	
机器人模型		robot_model	
实时控制	pr2_controller_manager	pr2_controller_interface, realtime_tools	
手臂运动规划	ompl, chomp_motion_planner, sbpl	actionlib through move_arm	actionlib through move_arm

它针对 ROS 中的概念提供相应的代码实现，包括 ROS 节点编写，主题发布和订阅，服务编写和调用以及参数服务器的使用。ROS 客户端库允许以不同编

程语言编写和实现，相互通信的节点相对独立，可以用不同的语言实现。目前主要是提供比较稳定的 C++ 和 Python 两种语言支持。rosCPP 是用于 C++ 语言的客户端库。rospy 是用于 Python 语言的客户端库。还有一些实验中的用户库，如 rosJava 和 rosLua。

(1) **rosCPP**: ROS 中的 C++ 语言客户端库。执行效率高，在 ROS 中使用得最广泛。

(2) **rospy**: ROS 中的 Python 语言客户端库，使得 ROS 可以享有面向对象的脚本语言所带来的便利。rospy 注重开发效率而不是运行效率，使得算法可以很快在 ROS 内设计和测试。它对于非关键路径上的代码实现，例如配置和初始化代码都非常理想。很多采用 rospy 编写的工具具有类型自省能力。ROS 节点管理器、rosLaunch 和很多工具都是采用 rospy 开发的。

(3) **rosLisp**: 用于 LISP 语言的客户端库，它目前用于开发规划库。它支持单独的节点创建和互动使用。

一个客户端库至少具有以下功能：

(1) 执行主/从 API 的“从”方面。这里涉及到管理 XML/RPC 服务器，发行和响应针对 XML/RPC 的请求。

(2) 处理节点到节点之间的传输协调和连接设置。一个客户端库只支持可能的传输机制的一个子集，像 ROS/TCPROS。

(3) 处理特定传输消息的重序列化和反序列化。

同时，客户端库还应该具有以下特点：

(1) 解析命令行重映射参数。不能执行该命令的节点将不是可重配置的。

(2) 从一个模拟时钟订阅消息。

(3) 发布调试信息到标准输出 rosout。

(4) 依赖于 roslang 功能包，该功能包允许 rosbuild 和别的工具执行合适动作，例如基于消息 (msg) 和基于服务 (srv) 的代码生成。

(5) 消息类型的对象表示。

(6) 连接服务的事件循环。

(7) 在消息接收端的用户回调。

## 3.2 rosCPP 客户端库

rosCPP 是 ROS 中的 C++ 语言客户端库，它使程序员能够快速与 ROS 主题、服务和参数等交互。它是在 ROS 中使用得最广泛的可以高效执行的库。对

于 Diamondback 版本，roscpp 的内部库已经移植到下面独立的功能包内：

- `cpp_common`
- `roscpp_serialization`
- `roscpp_traits`
- `rostime`

### 3.2.1 简单的主题发布者和主题订阅者

#### 1. 主题发布者节点

节点是连接 ROS 网络的可执行文件。我们将创建一个主题发布者节点，该节点将连续广播一条消息。

改变当前目录到 `beginner_tutorials` 功能包：`roscore beginner_tutorials`。在 `beginner_tutorials` 功能包内创建 `src/talker.cpp` 文件，然后添加如下代码：

```
1  /* Copyright (C) 2008, Morgan Quigley and Willow Garage,
   Inc. */
2
3  #include "ros/ros.h"
4
5  #include "std_msgs/String.h"
6
7  #include <iostream>
8
9  int main(int argc, char **argv)
10 {
11     ros::init(argc, argv, "talker");
12
13     ros::NodeHandle n;
14
15     ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
16
17     ros::Rate loop_rate(10);
18
19     int count = 0;
20     while (ros::ok())
```

```
21  {
22      std_msgs::String msg;
23
24      std::stringstream ss;
25      ss << "hello world" << count;
26      msg.data = ss.str();
27
28      ROS_INFO("%s", msg.data.c_str());
29
30      chatter_pub.publish(msg);
31
32      ros::spinOnce();
33
34      loop_rate.sleep();
35
36      ++count;
37  }
38
39  return 0;
40 }
```

代码解释：

(1) 代码第 3 行。ros/ros.h 是一个能够包含 ROS 中所有常用必要的头文件。

(2) 代码第 5 行。这里包含了 std\_msgs/String 消息，其位置在 std\_msgs 功能包内。这是一个由 String.msg 文件自动生成的头文件。

(3) 代码第 11 行。该行用于初始化 ROS。它允许 ROS 通过命令行进行名称再映射。这也是指定节点名称的时刻。在 ROS 运行系统内，节点名称必须是唯一的。这里的名称是基本名称，其中不能有斜线 (/)。

(4) 代码第 13 行。创建用于处理该节点的句柄。开始节点句柄 (NodeHandle) 将完全初始化节点，最后节点句柄析构函数将清除其所用的资源。

(5) 代码第 15 行。该行命令告诉节点管理器在 chatter 主题上发布一个类型为 std\_msgs/String 的消息。这可以让节点管理器把我们将要在 chatter 主题上发布数据的事件告知任何正在监听这个主题的节点。第二个变量是发布队列的长度。如果消息发布过快，那么这个设置就是缓存的消息数。在当前设置下，它将缓冲最多 1000 条消息。

NodeHandle::advertise() 将返回一个 `ros::Publisher` 对象，它的目的有两个：① 它包含了 `publish()` 方法，该方法让用户可以在创建的主题上发布消息；② 当它超出范围之外时，将自动停止广播。当所有返回的 `publish()` 对象都被销毁，那么这个主题就自动地不再广播。`publish()` 函数是描述用户如何发送消息的。参数是消息的对象。这个对象必须和 `advertise()` 给出的模板参数相符合。

(6) 代码第 17 行。`ros::Rate` 允许指定循环频率(当前是 10Hz)，它追踪了自从上一次从 `Rate::sleep()` 被唤醒之后持续的时间，并在到达指定时间后休眠。

(7) 代码第 19~21 行。缺省情况下，`roscpp` 将安装一个 `SIGINT` 句柄，它提供了 `Ctrl+C` 命令，使 `ros::ok()` 函数返回的值为 `false`。

`ros::ok()` 在下列情况下返回值为 `false`：

- `SIGINT` 句柄接收到 `Ctrl+C` 命令。
- 被名称相同的节点剔除出网络。
- `ros::shutdown()` 被应用的另一部分唤醒。

一旦 `ros::ok()` 返回值为 `false`，所有的 ROS 请求将失效。

(8) 代码第 22~26 行。采用消息自适应类在 ROS 上广播消息，该消息一般从消息文件生成。目前采用标准的 String 消息，它有一个成员：`data`。

(9) 代码第 28 行。`ROS_INFO` 和友类用于替代 `printf/cout`。

(10) 代码第 30 行。这是把消息发送给任何连接到网络的节点。

(11) 代码第 32 行。就这个简单的程序来说，调用 `ros::spinOnce()` 并不是必需的，因为我们并不接收任何回调。但是，如果在这个应用程序中添加主题的订阅，而程序中却没有添加 `ros::spinOnce()` 语句，那么就会发现回调函数将不起作用。

(12) 代码第 34 行。现在使用对象 `ros::Rate` 使之休眠，以使得剩余时间内正好以 10Hz 频率发布消息。

综上所述，编写主题发布者节点需要做的是：

- (1) 初始化 ROS 系统。
- (2) 广播消息：在 `chatter` 主题上发布 `std_msgs/String` 消息到节点管理器。
- (3) 以指定频率循环发布消息到 `chatter` 主题。

## 2. 主题订阅者节点

```
1  /* Copyright (C) 2008, Morgan Quigley and Willow Garage,  
   Inc. */  
2  
3  #include "ros/ros.h"  
4  #include "std_msgs/String.h"  
5  
6  void chatterCallback(const std_msgs::String::ConstPtr& msg)  
7  {  
8      ROS_INFO("I heard: [%s]", msg->data.c_str());  
9  }  
10  
11 int main(int argc, char **argv)  
12 {  
13     ros::init(argc, argv, "listener");  
14  
15     ros::NodeHandle n;  
16  
17     ros::Subscriber sub = n.subscribe("chatter", 1000,  
18                                         chatterCallback);  
19  
20     ros::spin();  
21  
22 }
```

代码解释：

(1) 代码第 6~9 行。这是当一个新消息到达 *chatter* 主题时的回调函数。消息在 **boost shared\_ptr** 内传播，这意味着需要的话用户可以保存它，不用担心它会在用户备份处理的数据前被删除。

(2) 代码第 17 行。通过节点管理器订阅 *chatter* 主题。当一条新的消息到达时，ROS 将会调用 **chatterCallback()** 函数。第二个参数是队列的长度，以便能够足够快地处理消息。当前情况下，如果队列达到 1000 条消息长度，新的消息到达时扔掉旧的消息。

**NodeHandle::subscribe()** 返回 **ros::Subscriber** 对象。当订阅对象被销毁时，它将自动停止从 *chatter* 主题订阅消息。

有很多版本的 **NodeHandle::subscribe()** 允许用户指定一个类成员函数或

者可以由 Boost 函数可调用的任何对象。

(3) 代码第 19 行。`ros::spin()` 进入消息处理循环，尽可能快地调用消息回调函数。这不会占用很多 CPU 资源。在 `ros::ok()` 返回为 `false` 或者由 `Ctrl+C` 句柄调用之前，`ros::spin()` 将一直存在。也有别的方法来处理回调。回顾上一段程序，它在 `while` 循环中，使用了 `spinOnce()` 函数。表示处理一次消息回调函数。然后回到 `while` 循环中。而这一段程序中，只有消息处理这项工作，因此可以直接使用 `spinOnce()` 进入消息处理循环。用户可以根据自己的需要来选择使用 `spin()` 还是 `spinOnce()`。

综上所述，编写主题订阅者节点需要做的是：

- (1) 初始化 ROS 系统。
- (2) 从 `chatter` 主题订阅消息。
- (3) Spin，然后等待消息到达。
- (4) 当消息到达时，`chatterCallback()` 函数被调用。

### 3. 编译节点

`roscreate-pkg` 命令可以为功能包创建缺省的 `Makefile` 和 `CMakeLists.txt`：

```
rosed beginner_tutorials CMakeLists.txt
```

其内容类似于：

```

1 cmake_minimum_required(VERSION 2.4.6)
2 include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
3
4 #Set the build type. Options are:
5 #Coverage : w/ debug symbols, w/o optimization, w/ code-
   coverage
6 #Debug : w/ debug symbols, w/o optimization
7 #Release : w/o debug symbols, w/ optimization
8 #RelWithDebInfo : w/ debug symbols, w/ optimization
9 #MinSizeRel : w/o debug symbols, w/ optimization, stripped
   binaries
10 #set(ROS_BUILD_TYPE RelWithDebInfo) rosbuild_init()
11 #set the default path for built executables to the "bin"
   directory
12 set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
13 #set the default path for built libraries to the "lib"
   directory

```

```
14 set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
15
16 #uncomment if you have defined messages
17 #rosbuild_genmsg()
18 #uncomment if you have defined services
19 #rosbuild_gensrv()
20 #common commands for building c++ executables and libraries
21 #rosbuild_add_library(${PROJECT_NAME}\} src/example.cpp)
22 #target_link_libraries(${PROJECT_NAME}\} another_library)
23 #rosbuild_add_boost_directories()
24 #rosbuild_link_boost(${PROJECT_NAME}\} thread)
25 #rosbuild_add_executable(example examples/example.cpp)
26 #target_link_libraries(example ${PROJECT_NAME})
```

在文件尾增加下面两行代码：

```
1 rosbuild_add_executable(talker src/talker.cpp)
2 rosbuild_add_executable(listener src/listener.cpp)
```

上面两行命令将在缺省目录 bin 下创建两个可执行程序：talker 和 listener，并添加到 bin 路径中。然后，可以执行命令：make。至此，编写了一个很简单的主题发布者和主题订阅者。

#### 4. 运行节点

首先，运行 roscore 命令，然后打开一个新的 shell，进入到 talker/listener 功能包。键入内容：./bin/talker。在原始的 shell 中输入：./bin/listener。主题发布者将会开始输出类似于下面的文本内容：

```
[ INFO] I published [Hello there! This is message [0]]
[ INFO] I published [Hello there! This is message [1]]
[ INFO] I published [Hello there! This is message [2]]
...

```

在运行主题发布者前提下，运行主题订阅者：rosrun beginner\_tutorials listener \_name:=listener，得到结果类似于下面的文本内容：

```
[ INFO] Received [Hello there! This is message [20]]
[ INFO] Received [Hello there! This is message [21]]
[ INFO] Received [Hello there! This is message [22]]
...

```

### 3.2.2 简单的服务器端和客户端

#### 1. 服务器端

本小节创建服务 `add_two_ints_server`, 用于接收两个整型数据并返回其和。

在 `beginner_tutorials` 功能包内创建文件 `src/add_two_ints_server.cpp`, 并贴入下面代码:

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3
4 bool add(beginner_tutorials::AddTwoInts::Request &req,
5           beginner_tutorials::AddTwoInts::Response &res)
6 {
7     res.sum = req.a + req.b;
8     ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long
9             int)req.b);
10    ROS_INFO("sending back response: [%ld]", (long int)res.
11              sum);
12    return true;
13 }
14
15 int main(int argc, char **argv)
16 {
17     ros::init(argc, argv, "add_two_ints_server");
18     ros::NodeHandle n;
19
20     ros::ServiceServer service = n.advertiseService
21         ("add_two_ints", add);
22     ROS_INFO("Ready to add two ints.");
23     ros::spin();
24
25     return 0;
26 }
```

代码解释:

(1) 代码第 1、2 行。`beginner_tutorials/AddTwoInts.h` 是之前创建的服务文件中生成的头文件。

(2) 代码第 4 行。该函数提供了添加 2 个整型数据的服务，它提取定义在服务文件中的请求和响应，并返回一个布尔型数据。

(3) 代码第 5~10 行。这里对两个整型变量求和并存储在响应中。一些关于请求和响应的信息被记录入日志。当它完成时，最终服务返回 true。

(4) 代码第 17 行。这里创建服务，并在 ROS 中广播。

## 2. 客户端

在 beginner\_tutorials 功能包内创建文件 src/add\_two\_ints\_client.cpp，并贴入下面代码：

```
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<
16         beginner_tutorials::AddTwoInts>("add_two_ints");
17     beginner_tutorials::AddTwoInts srv;
18     srv.request.a = atol(argv[1]);
19     srv.request.b = atol(argv[2]);
20     if (client.call(srv))
21     {
22         ROS_INFO("Sum: %ld", (long int)srv.response.sum);
23     }
24     else
25     {
26         ROS_ERROR("Failed to call service add_two_ints");
27     }
28 }
```

```

28
29     return 0;
30 }

```

代码解释：

(1) 代码第 15 行。这行命令为服务 `add_two_ints` 创建了一个客户端。对象 `ros::ServiceClient` 用于后面调用服务。

(2) 代码第 16~18 行。这里实例化一个自动生成的服务类，并赋值到它的请求成员。一个服务类包含两个成员：`request` 和 `response`。它同时也包含两个类定义：`Request` 和 `Response`。

(3) 代码第 19 行。该命令调用服务，如果服务调用成功，`call()` 返回为 `true`，`srv.response` 中的值将会有有效。`call()` 返回为 `false`，`srv.response` 中的值将会有无效。

### 3. 编译代码

编辑 `beginner_tutorials CMakeLists.txt` 文件：

```
rosed beginner_tutorials CMakeLists.txt
```

添加下面两行命令：

```

1 rosbuild_add_executable(add_two_ints_server src/
    add_two_ints_server.cpp)
2 rosbuild_add_executable(add_two_ints_client src/
    add_two_ints_client.cpp)

```

这将创建 2 个可执行文件：`add_two_ints_server` 和 `add_two_ints_client`。缺省情况下，上面两个文件放在 `bin` 目录下。然后可以执行命令：`make`。

### 4. 运行结果

#### 1) 运行服务器端

运行命令：

```
rosrun beginner_tutorials add_two_ints_server
```

结果如下：

```
Ready to add two ints
```

#### 2) 运行客户端

增加必要的参数之后，运行客户端：

```
rosrun beginner_tutorials add_two_ints_client 1 3
```

得到结果如下：

```
[ INFO] [1294899213.947679150]: Ready to add two ints.  
[ INFO] [1294899238.547623511]: request: x=1, y=3  
[ INFO] [1294899238.547662895]: sending back response: [4]
```

### 3.2.3 rosccpp 中参数的使用

#### 1. 提取参数

##### 1) getParam()

有 2 种方法使用节点句柄来提取参数。在下面例子代码中，`n` 是节点句柄的实例。`getParam()` 有一系列重载，重载允许下列格式：

```
1 bool getParam (const std::string& key, parameter_type&  
    output_value) const
```

其中 `key` 是计算图源名称，`output_value` 是放置提取数据的地方，`parameter_type` 是一个布尔、整型、字符串或特定 `XmlRpcValue` 类型数据，其中 `XmlRpcValue` 类型可以表示任何类型，也可以是 `lists/maps`。

`getParam()` 的使用方法很简单：

```
1 std::string s;  
2 n.getParam("my_param", s);
```

注意到 `getParam()` 返回一个布尔型数据，这个布尔型数据能够检查是否成功提取参数。

```
1 std::string s;  
2 if (n.getParam("my_param", s))  
3 {  
4     ROS_INFO("Got param: %s", s.c_str());  
5 }  
6 else  
7 {  
8     ROS_ERROR("Failed to get param 'my_param'");  
9 }
```

##### 2) Param()

`param()` 类似于 `getParam()`，但是允许用户指定一个缺省值，以使参数能够正确提取：

```

1 int i;
2 n.param("my_num", i, 42);

```

有时候，编译器需要对字符串类型做一些提示：

```

1 std::string s;
2 n.param<std::string>("my_param", s, "default_value");

```

## 2. 设置参数

设置参数是通过 `setParam()` 方法实现的：

```
1 n.setParam("my_param", "hello there");
```

`setParam()` 可以取布尔、整型、双精度、字符串和特殊 `XmlRpcValue` 类型数据。

## 3. 删除参数

删除参数是通过 `deleteParam()` 方法实现的：

```
1 n.deleteParam("my_param");
```

## 4. 检查存在性

这通常不必要，但是有一个 `hasParam()` 方法可以检查参数存在性：

```

1 if (!n.hasParam("my_param"))
2 {
3     ROS_INFO("No param named 'my_param'");
4 }

```

## 5. 搜索参数

参数服务器允许用户搜索参数，在当前命名空间开始并且工作于父命名空间。

例如，如果参数 `/a/b` 在参数服务器存在，用户的节点句柄在 `/a/c` 命名空间内，`searchParam()` 搜索 `b` 将生成 `/a/b`。但是如果参数 `/a/c/b` 被加进来，`searchParam()` 搜索 `b` 将生成 `/a/c/b`。

```

1 std::string param_name;
2 if (n.searchParam("b", param_name))

```

```

3  {
4      int i = 0;
5      n.getParam(param_name, i);
6  }
7 else
8 {
9     ROS_INFO("No param 'b' found in an upward search");
10 }

```

### 3.2.4 从节点句柄存取私有名称

#### 1. 为什么不采用 ~ name 形式?

roscpp 中的节点句柄引入导致了处理私有名称时的困惑。如果它用自己的命名空间创建一个节点句柄:

```

1 ros::init(argc, argv, "my_node_name");
2 ros::NodeHandle nh("/my_node_handle_namespace");

```

那么私有名称将在哪里解析?一些可能的选择如下:

- /my\_node\_handle\_namespace/my\_node\_name/name
- my\_node\_name/my\_node\_handle\_namespace/name
- /my\_node\_handle\_namespace/name
- 其他位置

由于这个原因, 节点句柄不允许直接传递私有名称到它的方法或以节点句柄作为参数的构建算子。

#### 2. 存取私有名称

构建一个带有私有名称的节点句柄作为其命名空间的解决方案是:

```

1 ros::init(argc, argv, "my_node_name");
2 ros::NodeHandle nh1("~");
3 ros::NodeHandle nh2("~/~ foo");

```

其中 nh1 的命名空间是 /my\_node\_name, 而 nh2 的命名空间是 /my\_node\_name/~ foo。因此可以采用:

```

1 ros::NodeHandle nh("~");
2 nh.getParam("name", ... );

```

来代替：

```
1 ros::NodeHandle nh;
2 nh.getParam("~name", ...);
```

### 3.2.5 用类方法订阅和回调服务

大部分例子中，采用函数作为例子，而不是用类方法。这是因为采用函数更简单，而不是类方法不支持。本小节展示怎样用类方法来订阅和回调服务。

#### 1. 订阅部分

假设有一个简单的类 Listener：

```
1 class Listener
2 {
3 public:
4     void callback(const std_msgs::String::ConstPtr& msg);
5 };
```

其中 NodeHandle::subscribe() 采用函数形式调用将会是如下形式：

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

NodeHandle::subscribe() 采用类方法形式调用将会是如下形式：

```
1 Listener listener;
2 ros::Subscriber sub = n.subscribe("chatter", 1000, &
                                    Listener::callback, &listener);
```

#### 2. 服务器端部分

假设有一个简单的类 AddTwo：

```
1 class AddTwo
2 {
3 public:
4     bool add(roscpp_tutorials::TwoInts::Request& req,
5               roscpp_tutorials::TwoInts::Response& res);
6 };
```

其中原来的 NodeHandle::advertiseService() 是如下形式：

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

采用类方法形式调用将会是如下形式：

```
1 AddTwo a;  
2 ros::ServiceServer ss = n.advertiseService("add_two_ints",  
    &AddTwo::add, &a);
```

### 3.2.6 计时器

#### 1. 简介

计时器可以让用户规划使回调在特定时刻发生。需要注意的是计时器不是用来替代实时线程或者内核的，不能保证它们是完全准确的。

创建一个计时器类似于创建一个主题订阅者：

```
1 ros::Timer timer = n.createTimer(ros::Duration(0.1),  
    timerCallback);
```

计时器回调采用下面形式：

```
1 void timerCallback(const ros::TimerEvent& e);
```

#### 2. 例子

下面来看一个多计时器的例子：

```
1 #include "ros/ros.h"  
2  
3 void callback1(const ros::TimerEvent&)  
4 {  
5     ROS_INFO("Callback 1 triggered");  
6 }  
7  
8 void callback2(const ros::TimerEvent&)  
9 {  
10    ROS_INFO("Callback 2 triggered");  
11 }  
12  
13 int main(int argc, char **argv)  
14 {
```

```
15     ros::init(argc, argv, "talker");
16     ros::NodeHandle n;
17
18     ros::Timer timer1=n.createTimer(ros::Duration(0.1),
19                                     callback1);
20
21     ros::Timer timer2=n.createTimer(ros::Duration(1.0),
22                                     callback2);
23
24 }
```

代码解释：这里创建了 2 个计时器，一个每 100 个 milliseconds 激发一次，另一个每秒激发一次。其程序输出结果如下：

```
[ INFO] 1251854032.362376000: Callback 1 triggered
[ INFO] 1251854032.462840000: Callback 1 triggered
[ INFO] 1251854032.562464000: Callback 1 triggered
[ INFO] 1251854032.662169000: Callback 1 triggered
[ INFO] 1251854032.762649000: Callback 1 triggered
[ INFO] 1251854032.862853000: Callback 1 triggered
[ INFO] 1251854032.962642000: Callback 1 triggered
[ INFO] 1251854033.063118000: Callback 1 triggered
[ INFO] 1251854033.162221000: Callback 1 triggered
[ INFO] 1251854033.262749000: Callback 1 triggered
[ INFO] 1251854033.262864000: Callback 2 triggered
[ INFO] 1251854033.362643000: Callback 1 triggered
[ INFO] 1251854033.463158000: Callback 1 triggered
...
...
```

### 3. TimerEvent 结构

ros::TimerEvent 结构提供了当前计时器的时间信息，其定义如下：

```
1 struct TimerEvent
2 {
3     Time last_expected;
4     Time last_real;
```

```
5
6     Time current_expected;
7     Time current_real;
8
9     struct
10    {
11         WallDuration last_duration;
12     } profile;
13 };
```

### 3.2.7 带动态可重配置及参数服务器的主题发布者/订阅者节点(C++)

本节讲述定制消息及动态可重配置服务器的使用方法，主要包括如何创建主题发布者节点以实现如下功能：

- (1) 通过使用参数服务器来从启动文件或者命令行初始化几个变量
- (2) 使用动态可重配置服务器来修改几个变量
- (3) 使用客户消息在主题上发布数据

同时涉及的内容包括与主题发布者节点一起工作的主题订阅者节点：

- (4) 设置主题订阅者节点在特定主题上收听客户消息并输出消息数据

#### 1) 运行例子

这里需要创建两个程序，非常类似于前面的两个例子，甚至有相同的名字：`talker` 和 `listener`。可以通过下面命令查看例子代码：

```
svn co http://ibotics.ucsd.edu/svn/stingray/trunk/node_example ~
```

确保例子代码文件放在 `ROS_PACKAGE_PATH` 的 `~/.bashrc` 文件中。在 `~/.bashrc` 文件尾添加如下代码：

```
1 export ROS_PACKAGE_PATH = ~ /node_example:
$ROS_PACKAGE_PATH
```

添加上述代码之后，重新启动终端或者运行：`source ~/.bashrc`。这些命令使得系统知道新的环境变量并且允许 ROS 找到新代码。

随后可以执行命令：

```
cd ~ /node_example
cmake
rosmake
```

命令 `cmake` 将创建 Makefile 和任何自动生成的代码。自动生成的代码是当 `CMakeLists.txt` 文件中的 `rosbuild_genmsg()` 和 `gencfg()` 被激活时在头文件中创建的。命令 `rosmake` 将生成用户可运行的二进制可执行节点，其位置在 `~/node_example/bin/`。

通过运行下面命令打开四个终端：

```
roscore
rosrun node_example talker
rosrun node_example listener
rosrun dynamic_reconfigure reconfigure_gui
```

可以看到从 `listener` 节点输出的结果。替代地，用户可以运行下面命令来启动 `launch` 文件：

```
roslaunch node_example c++_node_example.launch
```

### 2) 创建客户消息

这里的客户消息包含：

```
string message
int32 a
int32 b
```

使用 `CMakeLists.txt` 文件中的 `rosbuild_genmsg()` 创建目录 `msg` 之后，不需要再做其他工作。

### 3) 为动态可重配置服务器创建配置

为动态可重配置服务器创建配置所需要的变量位于 `cfg` 目录中，它们位于下面代码中，并通过 `node_example_params.cfg` 来调用。

```
1  #! /usr/bin/env python
2
3  PACKAGE='node_example'
4  import roslib
5  roslib.load_manifest(PACKAGE)
6
7  from dynamic_reconfigure.parameter_generator import *
8
9  gen = ParameterGenerator()
10 #          Name          Type   Level Description  Default Min
11 #                                         Max
11  gen.add("message", str_t, 0,    "The message.", "hello")
```

```
12 gen.add("a",           int_t, 0,    "First number.", 1, -100,
13   100)
14
15 exit(gen.generate(PACKAGE, "node_example",
16   "node_example_params"))
```

上面代码意味着能够修改消息及两个整数，通过修改这些值可以看到其运行结果。通过修改文件的执行权限命令来确保文件 `node_example_params.cfg` 被执行：

```
chmod 755 ~/node_example/cfg/node_example_params.cfg
```

至此，拥有了所有使用动态配置服务器所需的条件。

## 1. ROS 节点模板

实现这个例子需要四个文件。下面给出描述类所需的一个源文件和一个头文件。

### 1) 发布者 (talker) 节点

文件 `node_example/src/talker.cpp` 的源代码如下：

```
1 #include "node_example_core.h"
2
3 int main(int argc, char **argv)
4 {
5     ros::init(argc, argv, "talker");
6     ros::NodeHandle n;
7
8     NodeExample *node_example = new NodeExample();
9
10    dynamic_reconfigure::Server<node_example::
11      node_example_paramsConfig> dr_srv;
12    dynamic_reconfigure::Server<node_example::
13      node_example_paramsConfig>::CallbackType cb;
14    cb = boost::bind(&NodeExample::configCallback,
15      node_example, _1, _2);
16    dr_srv.setCallback(cb);
17
18    int a;
```

```

16     int b;
17     string message;
18     int rate;
19     string topic;
20
21     ros::NodeHandle private_node_handle_("~");
22     private_node_handle_.param("a", a, int(1));
23     private_node_handle_.param("b", b, int(2));
24     private_node_handle_.param("message", message, string
25         ("hello"));
26     private_node_handle_.param("rate", rate, int(40));
27     private_node_handle_.param("topic", topic, string
28         ("example"));
29
30     ros::Publisher pub_message = n.advertise<node_example::
31         node_example_data>(topic.c_str(), 10);
32
33     ros::Rate r(rate);
34
35     while (n.ok())
36     {
37         node_example->publishMessage (&pub_message);
38         ros::spinOnce();
39         r.sleep();
40     }
41 }
```

## 2) 订阅者 (listener) 节点

文件 node\_example/src/listener.cpp 的源代码如下：

```

1 #include "node_example_core.h"
2
3 int main(int argc, char **argv)
4 {
5     ros::init(argc, argv, "listener");
6     ros::NodeHandle n;
7 }
```

```
8     int rate;
9     string topic;
10
11    ros::NodeHandle private_node_handle_("~");
12    private_node_handle_.param("rate", rate, int(40));
13    private_node_handle_.param("topic", topic, string
14        ("example"));
15
16    NodeExample *node_example = new NodeExample();
17
18    ros::Subscriber sub_message = n.subscribe(topic.c_str(),
19        1000, &NodeExample::messageCallback, node_example);
20
21    ros::Rate r(rate);
22
23    while (n.ok())
24    {
25        ros::spinOnce();
26        r.sleep();
27    }
28
29    return 0;
30 }
```

### 3) NodeExample 类

文件 node\_example/src/node\_example\_core.cpp 的源代码如下：

```
1 #include "node_example_core.h"
2
3 NodeExample::NodeExample()
4 {
5 }
6
7 NodeExample::~NodeExample()
8 {
9 }
10
11 void NodeExample::publishMessage(ros::Publisher *
12     pub_message)
13 {
14 }
```

```

13     node_example::node_example_data msg;
14     msg.message = message;
15     msg.a = a;
16     msg.b = b;
17
18     pub_message->publish(msg);
19 }
20
21 void NodeExample::messageCallback(const node_example::
22     node_example_data::ConstPtr &msg)
23 {
24     message = msg->message;
25     a = msg->a;
26     b = msg->b;
27
28     ROS_INFO("message is %s", message.c_str());
29     ROS_INFO("sum of a + b = %d", a + b);
30 }
31
32 void NodeExample::configCallback(node_example::
33     node_example_paramsConfig &config, uint32_t level)
34 {
35     message = config.message.c_str();
36     a = config.a;
37     b = config.b;
38 }
```

#### 4) NodeExample 头文件

文件 node\_example/include/node\_example\_core.h 的源代码如下：

```

1 #ifndef SR_NODE_EXAMPLE_CORE_H
2 #define SR_NODE_EXAMPLE_CORE_H
3
4 #include "ros/ros.h"
5 #include "ros/time.h"
6
7 #include "node_example/node_example_data.h"
8
9 #include <dynamic_reconfigure/server.h>
10 #include <node_example/node_example_paramsConfig.h>
```

```
11
12 using std::string;
13
14 class NodeExample
15 {
16 public:
17     NodeExample();
18
19     ~NodeExample();
20
21     void configCallback(node_example::
22                         node_example_paramsConfig &config, uint32_t level);
23
24     void publishMessage(ros::Publisher *pub_message);
25
26     void messageCallback(const node_example::
27                           node_example_data::ConstPtr &msg);
28
29     string message;
30
31     int a;
32
33     int b;
34 };
35
36 #endif
```

### 5) 编译节点

修改文件 `node_example/msg/node_example_data.msg` 的内容以实现添加、移除或者重命名变量功能，需要运行如下命令：

```
cd ~/node_example
cmake
rosmake node_example
```

## 2. 参数服务器及动态可重配置

### 1) 参数服务器

通过使用参数服务器，有几种方式可以实现设置变量来初始化值。一种是通过一个启动文件，另一种是用命令行方式。

在 `node_example/c++_node_example.launch` 中 `talker` 节点通过四个参数开始: `message`、`a`、`b` 和 `rate`。通过下面命令行可以实现同样功能:

```
rosrun node_example talker _message:="Hello world!" _a:=57
                                _b:=-15 _rate:=1
```

需要注意的是波浪线符号 " ~ " 已经被下划线替代。

然后运行如下命令可以看到不同之处: `rosrun node_example listener`。

## 2) 动态可重配置

动态可重配置工具有些奇怪, 因为它允许用户在运行时修改变量, 而不是在开始运行时。注意到文件 `node_example/manifest.xml` 必须有如下代码:

```
1 <depend package="dynamic_reconfigure"/>
```

`CMakeLists.txt` 文件中的 `gencfg()` 导致自动生成: `node_example/cfg/cpp/node_example/node_example_paramsConfig.h`。该文件位于 `node_example/include/node_example_core.h`。

## 3. 需要改变的地方

为了使用这个例子, 用户必须做如下改动:

(1) 编辑文件 `manifest.xml` 来确定功能包依赖项。

(2) 编辑文件 `CMakeLists.txt` 来改变需要编译的可执行文件名字以及用到的源文件名字。

(3) 重新命名 `cfg/node_example_params.cfg` 来匹配节点名称。

- 改变 `PACKAGE=` 所在行代码来匹配用户节点名称。

- 改变最后一行来使用 `node_example` 中表示的两处用户节点名称。

- 修改动态可重置服务器所需的变量。

(4) 重新命名 `msg/node_example_data.msg` 来匹配节点名称。

- 修改新消息中用户想要的变量。

(5) 重新命名 `include/node_example_core.h`。

- 修改 `#include` 所在行来使用从 `.cfg` 和 `.msg` 文件新生成的头文件。

- 修改类名称, 函数及变量。

(6) 重新命名 `src/node_example_core.cpp`。

- 使用新的包含文件。

- 修改类名称，函数及变量。

(7) 重新命名 `src/talker.cpp` 或者 `src/listener.cpp`。

- 仅设置例子中需要的部分。

• 如果需要，可以把配置参数、动态可重配置服务器、主题发布者和主题订阅者组合到一个节点中。

### 3.2.8 带动态可重配置及参数服务器的主题发布者/订阅者节点（Python）

本节讲述使用 Python 语言定制消息及动态可重配置服务器的使用方法，主要包括如何创建主题发布者节点以实现如下功能：

- (1) 通过使用参数服务器来从启动文件或者命令行初始化几个变量。
- (2) 使用动态可重配置服务器来修改几个变量。
- (3) 使用客户消息在主题上发布数据。

同时涉及的内容包括与主题发布者节点一起工作的主题订阅者节点：

- (4) 设置主题订阅者节点在特定主题上收听客户消息并输出消息数据。

#### 1) 运行例子

这里需要创建两个程序，非常类似于前面的两个例子，甚至有相同的名字：主题发布者和主题订阅者。可以通过下面命令查看例子代码：

```
svn co http://ibotics.ucsd.edu/svn/stingray/trunk/node_example ~
```

确保例子代码文件放在 `ROS_PACKAGE_PATH` 的 `~/.bashrc` 文件中。在 `~/.bashrc` 文件尾添加如下代码：

```
1 export ROS_PACKAGE_PATH = ~/node_example:$ROS_PACKAGE_PATH
```

添加上述代码之后，重新启动终端或者运行：`source ~/.bashrc`。这些命令使得系统知道新的环境变量并且允许 ROS 找到新代码。

随后可以执行命令：

```
cd ~/node_example
```

```
cmake
```

```
rosmake
```

命令 `cmake` 将创建 `Makefile` 和任何自动生成的代码。自动生成的代码是当 `CMakeLists.txt` 文件中的 `rosbuild_genmsg()` 和 `gencfg()` 被激活时在头文件中创建的。命令 `rosmake` 将生成用户可运行的二进制可执行节点，其位置在 `~/node_example/bin/`。

通过运行下面命令打开四个终端：

```
roscore
rosrun node_example pytalker.py
rosrun node_example pylistener.py
rosrun dynamic_reconfigure reconfigure_gui
```

如果在运行 Python 节点时有问题，通过修改文件的执行权限确保这些文件是可执行的：

```
chmod 755 ~ /node_example/src/pytalker.py ~ /node_example/src/
pylistener.py
```

可以看到 `pylistener.py` 节点输出的结果。替代地，用户可以运行下面命令来启动启动文件：

```
roslaunch node_example python_node_example.launch
```

启动文件将启动下列文件：`pytalker.py`、`pylistener.py`、`reconfigure_gui` 和 `rxconsole`。`pylistener.py` 的结果将会显示在 `rxconsole`，而不是使用启动文件的终端。

## 2) 创建客户消息

这里的客户消息包含：

```
string message
int32 a
int32 b
```

使用 `CMakeLists.txt` 文件中的 `rosbuild_genmsg()` 创建目录 `msg` 之后，不需要再做其他工作。

## 3) 为动态可重配置服务器创建配置

为动态可重配置服务器创建配置所需要的变量位于 `cfg` 目录中，它们位于下面代码中，并通过 `node_example_params.cfg` 来调用。

```
1 #! /usr/bin/env python
2
3 PACKAGE='node_example'
4 import roslib
5 roslib.load_manifest(PACKAGE)
6
7 from dynamic_reconfigure.parameter_generator import *
8
9 gen = ParameterGenerator()
```

```
10 # Name          Type   Level Description Default Min   Max
11 gen.add("message",str_t, 0,"The message.", "hello")
12 gen.add("a",      int_t, 0,"First number.", 1, -100, 100)
13 gen.add("b",      int_t, 0,"First number.", 2, -100, 100)
14
15 exit(gen.generate(PACKAGE, "node_example",
                     "node_example_params"))
```

上面代码意味着能够修改消息及两个整数，通过修改这些值可以看到其运行结果。通过指向下面命令来确保文件 `node_example_params.cfg` 被执行：

```
chmod 755 ~/node_example/cfg/node_example_params.cfg
```

至此，拥有了所有使用动态配置服务器所需的条件。

## 1. ROS 节点模板

实现这个例子需要四个文件。下面给出描述类所需的一个源文件和一个头文件。

### 1) 主题发布者 (pytalker) 节点

文件 `node_example/src/pytalker.py` 的源代码如下：

```
1 #!/usr/bin/env python
2
3 import roslib
4 roslib.load_manifest('node_example')
5 import rospy
6 import sys
7
8 from dynamic_reconfigure.server import Server as
      DynamicReconfigureServer
9
10 from node_example.msg import node_example_data
11 from node_example.cfg import node_example_paramsConfig as
      ConfigType
12
13 class NodeExample():
14     def __init__(self):
15         init_message = rospy.get_param('~message', 'hello')
16         rate = float(rospy.get_param('~rate', '1.0'))
17         topic = rospy.get_param('~topic', 'chatter')
```

```
18         rospy.loginfo('rate = %d', rate)
19         rospy.loginfo('topic = %s', topic)
20
21         self.server = DynamicReconfigureServer(ConfigType,
22                                         self.reconfigure)
23
24         pub = rospy.Publisher(topic, node_example_data)
25
26         msg = node_example_data()
27
28         msg.a = 1
29         msg.b = 2
30         msg.message = init_message
31
32         while not rospy.is_shutdown():
33
34             msg.message = self.message
35             msg.a = self.a
36             msg.b = self.b
37
38             pub.publish(msg)
39
40             if rate:
41                 rospy.sleep(1/rate)
42             else:
43                 rospy.sleep(1.0)
44
45             def reconfigure(self, config, level):
46
47                 self.message = config["message"]
48                 self.a = config["a"]
49                 self.b = config["b"]
50
51                 return config
52
53     if __name__ == '__main__':
54         rospy.init_node('pytalker')
55         try:
56             ne = NodeExample()
```

```
56     except rospy.ROSInterruptException: pass
```

## 2) 主题订阅者 (pylistener) 节点

文件 node\_example/src/pylistener.py 的源代码如下：

```
1 #!/usr/bin/env python
2
3 import roslib
4 roslib.load_manifest('node_example')
5 import rospy
6
7 from node_example.msg import node_example_data
8
9 def callback(data):
10     rospy.loginfo(rospy.get_name() + " I heard %s", data.
11                   message)
12     rospy.loginfo(rospy.get_name() + " a + b = %d", data.a
13                   + data.b)
14
15 def listener():
16
17     topic = rospy.get_param('~topic', 'chatter')
18
19     rospy.Subscriber(topic, node_example_data, callback)
20
21 if __name__ == '__main__':
22     rospy.init_node('pylistener', anonymous = True)
23     listener()
```

## 3) 编译节点

修改文件 node\_example/msg/node\_example\_data.msg 的内容以实现添加、移除或者重命名变量功能，需要运行如下命令：

```
cd ~/node_example
cmake
rosmake node_example
```

## 2. 使用参数服务器及动态可重配置

### 1) 参数服务器

通过使用参数服务器，有几种方式可以实现设置变量来初始化值。一种是通过一个启动文件，另一种是用命令行方式。

在 `node_example/c++_node_example.launch` 中 `pytalker` 节点通过四个参数开始：`message`、`a`、`b` 和 `rate`。通过下面命令行可以实现同样功能：

```
rosrun node_example pytalker.py _message:="Hello world!"  
_a:=57 _b:=-15 _rate:=1
```

需要注意的是波浪线符号“~”已经被下划线替代。

然后运行如下命令可以看到不同之处：`rosrun node_example pylistener.py`。

### 2) 动态可重配置

动态可重配置工具有些奇怪，因为它允许用户在运行时修改变量，而不是在开始运行时。注意到文件 `node_example/manifest.xml` 必须有如下代码：

```
1 <depend package="dynamic_reconfigure"/>
```

## 3. 需要改变的地方

为了使用这个例子，用户必须做如下改动：

- (1) 编辑文件 `manifest.xml` 来确定功能包依赖项。
- (2) 编辑文件 `CMakeLists.txt` 来改变需要编译的可执行文件名字以及用到的源文件名字。
- (3) 重新命名 `cfg/node_example_params.cfg` 来匹配节点名称。
  - ① 改变 `PACKAGE=` 所在行代码来匹配用户节点名称。
  - ② 改变最后一行来使用 `node_example` 中表示的两处用户节点名称。
  - ③ 修改动态可重置服务器所需的变量。
- (4) 重新命名 `msg/node_example_data.msg` 来匹配节点名称。
  - 修改新消息中用户想要的变量。
- (5) 重新命名 `src/pytalker.py` 和 `src/pylistener.py`。
  - ① 仅设置例子中需要的部分。
  - ② 导入新的客户消息和动态可重配置文件。

③ 修改 `#include` 所在行来使用从 .cfg 和 .msg 文件新生成的头文件。

④ 如果需要，可以把配置参数、动态可重配置服务器、主题发布者和主题订阅者组合到一个节点中。

### 3.2.9 组合 C++/Python 主题发布者/订阅者节点

本小节展示如何组合 C++/Python 主题发布者/订阅者节点。

执行下面命令启动所有节点：`talker`、`listener` 和 `pytalker.py`、`dynamic reconfigure GUI` 和 `rxconsole`：

```
roslaunch node_example node_example.launch
```

#### 1. 可能的组合

- C++ publisher → C++ listener
- C++ publisher → Python listener
- C++ publisher → C++ 和 Python listeners
- Python publisher → C++ listener
- Python publisher → Python listener
- Python publisher → C++ 和 Python listeners

#### 2. 演示

使用 `dynamic reconfigure GUI` 可以选择不同的主题发布者并改变消息或者数字，并在 `rxconsole` 中查看结果。

## 3.3 rospy 客户端库

rospy 是 ROS 中的 Python<sup>[13, 14]</sup> 语言客户端库。rospy 客户端 API 使得 Python 编程者能够快速与 ROS 主题、服务和参数进行交互。rospy 注重执行速度而不是运行效率，使得算法可以很快在 ROS 内设计和测试。它对于非关键路径上的代码，例如配置和初始化代码非常理想。

### 3.3.1 简单的主题发布者/订阅者

本小节将创建两个简单的 rospy 节点，其中主题发布者 (`talker`) 节点将在主题 `chatter` 上广播消息，而主题订阅者 (`listener`) 节点将接收和打印该消息。

## 1. 主题发布者节点

节点是连接到 ROS 网络的可执行文件。我们将创建一个主题发布者节点，该节点将连续广播一条消息。

改变当前目录到前面创建的 `beginner_tutorials` 功能包: `roscore beginner_tutorials`。

在 `beginner_tutorials` 功能包内创建 `nodes/talker.py` 文件:

```

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)
7     rospy.init_node('talker')
8     while not rospy.is_shutdown():
9         str = "hello world %s"%rospy.get_time()
10        rospy.loginfo(str)
11        pub.publish(String(str))
12        rospy.sleep(1.0)
13 if __name__ == '__main__':
14     try:
15         talker()
16     except rospy.ROSInterruptException: pass

```

最后，需要增加执行权限使之变成可执行文件: `chmod +x nodes/talker.py`。

**代码解释:**

(1) 代码第 1、2 行。每个 ROS 节点在开始时需要声明。第一行代码用来说明该脚本文件是可执行的，第二行代码告诉 `beginner_tutorials` 功能包 `roslib` 读 `manifest.xml` 文件及相应的依赖项到用户的 `PYTHONPATH`。这对于后面的代码很重要。

(2) 代码第 3、4 行。前面代码已经可找到标准的 Python 库文件 (像 `sys` 和 `time`)，但是 `roslib.load_manifest()` 使得它能够找到 `rospy` 和 `std_msgs`，这些已经在前面的文件 `manifest.xml` 的依赖项中声明。如果正在写一个节点，那么需要导入 `rospy`。通过 `std_msgs.msg` 的导入可以重用 `std_msgs/String` 消息类型来发布消息。

(3) 代码第 5~7 行。这几行代码用于定义到 ROS 其他部分的 `talker` 接口。代码第 6 行声明用户代码正在用消息类型 `String` 向 `chatter` 主题发布消息。这里的 `String` 实际上是类 `std_msgs.msg.String`。`rospy.init_node(NAME)` 用来告诉 `rospy` 该节点的名称，在此之前，它不能跟 ROS 节点管理器进行通信。这种情况下，节点将取名为 `talker`。

注意：名称必须是基本名称，不能包含任何斜线 (/)。

(4) 代码第 8~12 行。这个循环是标准的 `rospy` 结构：检查 `rospy.is_shutdown()` 标记，然后开始工作。循环内调用 `pub.publish(String(str))`，目的是使用新创建的 `String` 类型消息发布消息到主题 `chatter` 上。`rospy.sleep()` 的调用，作用类似于 `time.sleep()`。

循环内调用 `rospy.loginfo(str)` 有三个作用：① 打印消息到屏幕；② 写入节点日志文件；③ 输出到 `rosout`。由于 `rosout` 是用于手工调试的工具，因此可以用 `rxconsole` 来查看消息，而不必用节点输出的控制台窗口。

`std_msgs.msg.String` 是一个很简单的消息类型，但是看起来它好像发布了很复杂类型的消息。这里参数的一般规则是 `constructor args` 与消息文件中参数顺序一致。当然，用户也可以不传递任何参数，直接初始化：

```
1 msg = String()  
2 msg.data = str
```

或者也可以初始化部分域值，而剩余的采用缺省值初始化：

```
1 String('data'=str)
```

(5) 代码第 13~16 行。除了标准的 Python `__main__` 检查，这样捕获了一个 `rospy.ROSInterruptException` 异常，该异常由 `rospy.sleep()` 和 `rospy.Rate.sleep()` 方法在按 `Ctrl+C` 组合键或者节点意外结束时激发。异常激发的原因是用户不必在 `sleep()` 之后连续执行代码。

## 2. 主题订阅者节点

在 `beginner_tutorials` 功能包内创建 `nodes/listener.py` 文件，然后添加如下代码：

```
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 callback(data):
6      rospy.loginfo(rospy.get_name()+"I heard %s",data.data)
7
8  def listener():
9      rospy.init_node('listener', anonymous=True)
10     rospy.Subscriber("chatter", String, callback)
11     rospy.spin()
12
13 if __name__ == '__main__':
14     listener()

```

最后，需要增加执行权限使之变成可执行文件：`chmod +x nodes/listener.py`。

代码解释：

除了介绍了一种新的基于回调的订阅消息的机制，`listener.py` 的代码类似于 `talker.py`。

代码第 10 行用于声明代码向类型为 `std_msgs.msgs.String` 的主题 `chatter` 订阅消息。当新的消息接收到后，`callback` 以消息作为第一个参数被激发。

这里还增加了 `anonymous=True` 键值参数。ROS 需要每个节点有唯一的名称，如果有同一名称的节点出现，它与前面的抵触。这使得网络可以很轻易地将失效节点剔除。`anonymous=True` 标记告诉 `rospy` 为节点生成唯一的名称，使得用户可以有多个 `listener.py` 运行。

最后，`rospy.spin()` 使得节点在结束前一直运行。不像在 `roscpp` 下的情况，`rospy.spin()` 由于有自己的线程而不会影响主题订阅者回调函数。

### 3. 编译节点

使用 `CMake` 编译代码，`roscreate-pkg` 已经自动生成了 `Makefile` 文件，因此用户不需要编辑它。现在可以运行：`make`。

### 4. 运行节点

首先，运行 `roscore` 命令，然后打开一个新的 shell，输入：

```
rosrun beginner_tutorials talker.py
```

在主题发布者终端界面中，会输出如下内容：

```
Registered [/talker-9224-1233892469.83] with master node
```

```
http://localhost:11311
hello world 1233892469.86
hello world 1233892470.86
hello world 1233892471.86
hello world 1233892472.86
hello world 1233892473.86
...
...
```

在主题订阅者终端界面中，会输出如下内容：

```
/listener-7457-1233891102.92 I heard hello world 1233892470.86
/listener-7457-1233891102.92 I heard hello world 1233892471.86
/listener-7457-1233891102.92 I heard hello world 1233892472.86
/listener-7457-1233891102.92 I heard hello world 1233892473.86
/listener-7457-1233891102.92 I heard hello world 1233892474.86
...
...
```

### 3.3.2 简单的服务端和客户端

#### 1. 服务端

本小节解释怎样创建服务 `add_two_ints_server`，用于接收两个整型数据并返回其和。

在 `beginner_tutorials` 功能包内创建文件 `nodes/add_two_ints_server.py`:

```
1 #!/usr/bin/env python
2 import roslib; roslib.load_manifest('beginner_tutorials')
3
4 from beginner_tutorials.srv import *
5 import rospy
6
7 def handle_add_two_ints(req):
8     print"Returning [%s + %s = %s]"%(req.a, req.b, (req.a +
9         req.b))
10    return AddTwoIntsResponse(req.a + req.b)
11
12 def add_two_ints_server():
13     rospy.init_node('add_two_ints_server')
```

```

13     s = rospy.Service('add_two_ints', AddTwoInts,
14         handle_add_two_ints)
15     print "Ready to add two ints."
16
17 if __name__ == '__main__':
18     add_two_ints_server()

```

最后，需要增加执行权限使之变成可执行文件：`chmod +x nodes/add_two_ints_server.py`。

上面的代码中，服务中很少是用 `rospy` 编写的。用 `init_node()` 声明代码，然后声明服务。

在上面代码的第 13 行，声明了一个类型为 `AddTwoInts service`、名为 `add_two_ints` 是新服务。所有的请求被传送到 `handle_add_two_ints` 函数。以实例 `AddTwoIntsRequest` 为参数，调用 `handle_add_two_ints`，然后返回实例 `AddTwoIntsResponse`。

正像前面主题订阅者的例子，`rospy.spin()` 在服务被关闭前，一直保持代码存在。

## 2. 客户端

在 `beginner_tutorials` 功能包内创建文件 `nodes/add_two_ints_client.py`：

```

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

```

```
14         return resp1.sum
15     except rospy.ServiceException, e:
16         print "Service call failed: %s"%e
17
18 def usage():
19     return "%s [x y]"%sys.argv[0]
20
21 if __name__ == '__main__':
22     if len(sys.argv) == 3:
23         x = int(sys.argv[1])
24         y = int(sys.argv[2])
25     else:
26         print usage()
27         sys.exit(1)
28     print "Requesting %s+%s"%(x, y)
29     print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

最后，需要增加执行权限使之变成可执行文件：`chmod +x nodes/add_two_ints_client.py`。

代码解释：

(1) 对于客户端，不需要调用 `init_node()`。首先调用：

```
rospy.wait_for_service('add_two_ints')
```

这种方法非常简洁，当 `add_two_ints` 服务不可用时，程序将一直阻塞。

(2) 上面代码的第 12 行，创建了一个用于调用服务的句柄。

(3) 从代码的第 13~14 行可以看到，用户可以像使用正常函数一样使用该服务。

因为声明服务类型是 `AddTwoInts`，其作用是为用户生成 `AddTwoIntsRequest`，返回值是 `AddTwoIntsResponse` 对象。如果调用失败，`rospy.ServiceException` 异常被激发，因此用户应当配置合适的 `try/except` 语句。

### 3. 编译代码

使用 CMake 编译代码，`roscreate-pkg` 已经自动生成了 `Makefile` 文件，因此用户不需要编辑它。现在可以运行：`make`。

### 3.3.3 rospy 中参数的使用

#### 1. 参数类型

用户可以使用布尔、浮点型、字符串作为参数值。同样也可以使用列表和字典类型。字典类型等价于 ROS 命名空间。这是组织类似参数到一起时非常有用的方法，之后可以自动获取和设置参数。例如，如果有一个增益集合：

```
/gains/P = 1.0
/gains/I = 2.0
/gains/D = 3.0
```

rospy 中，可以设置和单独获取这些值，也可以作为一个字典整体使用它们。参数 /gains 有 Python 字典值：{'P': 1.0, 'I': 2.0, 'D': 3.0}。

#### 2. 获取、设置和删除参数

获取参数就像调用 `rospy.get_param(param_name)` 一样简单：

```
1 rospy.get_param('/global_param_name')
2
3 rospy.get_param('param_name')
4
5 rospy.get_param('~private_param_name')
```

如果参数不存在，也可以设置一个缺省值：

```
1 rospy.get_param('foo', 'default_value')
```

类似地，也可以通过调用 `rospy.set_param(param_name, param_value)` 设置参数：

```
1 rospy.set_param('some_numbers', [1., 2., 3., 4.])
2 rospy.set_param('truth', True)
3 rospy.set_param('~private_bar', 1+2)
```

删除参数可以通过调用 `rospy.delete_param(param_name)` 来实现：

```
1 rospy.delete_param('param_name')
```

如果不知道参数是否存在，可以通过 `rospy.has_param(param_name)` 查询：

```
1 if rospy.has_param('to_delete'):
2     rospy.delete_param('to_delete')
```

### 3. 解析参数名称

ROS 中名称可以再映射，节点名称可以放入命名空间中。rospy 将会做大部分工作，自动解析用户放入 `get_param`, `set_param` 中的名称。有时候，为了调试方便，可能需要打印出正在使用的参数名称。

调用 `rospy.resolve_name(name)` 可以实现确定参数确切名称的功能：

```
1 value = rospy.get_param('~foo')
2 rospy.loginfo('Parameter %s has value %s', rospy.
    resolve_name('~foo'), value)
```

`rospy.resolve_name(name)` 将会应用任何重映射功能，并理解用户的节点命名空间。

### 4. 搜索参数

ROS 中，如果不知道参数名称，可以搜索其名称。搜索路径从私有名称开始，一直到全局名称为止。可以使用 `rospy.search_param(param_name)` 来找到解析过的参数名称。同样，也可以正常获取和设置参数：

```
1 full_param_name = rospy.search_param('foo')
2 param_value = rospy.get_param(full_param_name)
```

#### 3.3.4 rospy 中 numpy 的使用

`numpy` 是一个用于 Python 的科学计算软件包。如果在使用过程中，涉及很多传感器数据，那么在 `rospy` 中使用 `numpy` 将会提高性能。

本小节中，将会涉及到 `rospy.numpy_msg` 模块。`rospy.numpy_msg` 模块与 `numpy_msg()` 一起，可以使节点反序列化消息到 `numpy` 阵列。`numpy_msg()` 也可以用于发布包含 `numpy` 阵列数据的消息。

##### 1. 第一步：功能包

首先创建一个功能包，它依赖于 `rospy_tutorials`，并使用它的消息类型中的一个：

```
1 roscreate-pkg numpy_tutorial rospy rospy_tutorials
```

为了确认功能包中已经声明了 `numpy`，在 `manifest.xml` 文件中增加下面 `rosdep` 行：

```
1 <rosdep name="python-numpy" />
```

## 2. 第二步：订阅者节点

下面使用 `rospy_tutorials/Floats` 消息类型创建标准的订阅者节点，它收听 `floats` 主题。`rospy_tutorials/Floats` 定义如下：

```

1  #!/usr/bin/env python
2  PKG = 'numpy_tutorial'
3  import roslib; roslib.load_manifest(PKG)
4
5  import rospy
6  from rospy_tutorials.msg import Floats
7
8  def callback(data):
9      print rospy.get_name(), "I heard %s"%str(data.data)
10
11 def listener():
12     rospy.init_node('listener')
13     rospy.Subscriber("floats", Floats, callback)
14     rospy.spin()
15
16 if __name__ == '__main__':
17     listener()
```

最后，需要增加执行权限使之变成可执行文件：`chmod +x numpy_listener.py`。

下面可以运行程序：首先执行 `roscore`，然后在新终端中运行：

`rosrun numpy_tutorial numpy_listener.py`

第三个终端窗口中，运行：

```
rostopic pub -r 1 floats rospy_tutorials/Floats
"[1.1, 2.2, 3.3, 4.4, 5.5]"
```

结果如下：

```
/listener-977-1248226102144 I heard (1.100000238418579,
2.2000000476837158, 3.2999999523162842,
4.4000000953674316, 5.5)
```

```
/listener-977-1248226102144 I heard (1.100000238418579,
2.2000000476837158, 3.2999999523162842,
```

4.400000953674316, 5.5)

... and so on

现在可以结束第二个终端窗口。

### 3. 第三步：Numpy 化订阅者

要实现 numpy 化订阅者，首先需要导入 `numpy_msg`，然后调用它：

```
from rospy.numpy_msg import numpy_msg  
...  
rospy.Subscriber("floats", numpy_msg(Floats), callback)
```

将上面两行代码放入整段代码中：

```
1 #!/usr/bin/env python  
2 PKG = 'numpy_tutorial'  
3 import roslib; roslib.load_manifest(PKG)  
4  
5 import rospy  
6 from rospy_tutorials.msg import Floats  
7 from rospy.numpy_msg import numpy_msg  
8  
9 def callback(data):  
10     print rospy.get_name(), "I heard %s"%str(data.data)  
11  
12 def listener():  
13     rospy.init_node('listener')  
14     rospy.Subscriber("floats", numpy_msg(Floats), callback)  
15     rospy.spin()  
16  
17 if __name__ == '__main__':  
18     listener()
```

加入代码之后，需要运行，在三个新终端中分别输入：

```
roscore  
rosrun numpy_tutorial numpy_listener.py  
rostopic pub -r 1 floats rospy_tutorials/Floats"  
[1.1, 2.2, 3.3, 4.4, 5.5]"
```

运行结果如下：

```
/listener-1243-1248226610835 I heard [ 1.10000002 2.20000005  
3.29999995 4.4000001 5.5]
```

```
/listener-1243-1248226610835 I heard [ 1.10000002 2.20000005  
3.29999995 4.4000001 5.5 ]
```

... and so on

#### 4. 第四步：Numpy 化发布者

Numpy 化订阅者之后，需要 numpy 化发布者。这类似于前面的情况，但是需要注意：所有的阵列数据必须初始化为 numpy 阵列。即使对那些用户希望以缺省值进行初始化的域也是需要小心处理的。由于 rospy 无法捕捉类型错误，用户必须指定正确的 numpy 数据类型。

创建一个新的文件 `numpy_talker.py`，添加如下内容：

```
1 #!/usr/bin/env python  
2 PKG = 'numpy_tutorial'  
3 import roslib; roslib.load_manifest(PKG)  
4  
5 import rospy  
6 from rospy.numpy_msg import numpy_msg  
7 from rospy_tutorials.msg import Floats  
8  
9 import numpy  
10 def talker():  
11     pub = rospy.Publisher('floats', numpy_msg(Floats))  
12     rospy.init_node('talker', anonymous=True)  
13     r = rospy.Rate(10) # 10hz  
14     while not rospy.is_shutdown():  
15         a = numpy.array([1.0, 2.1, 3.2, 4.3, 5.4, 6.5],  
16                         dtype=numpy.float32)  
17         pub.publish(a)  
18         r.sleep()  
19 if __name__ == '__main__':  
20     talker()
```

然后增加文件的执行权限: `chmod +x numpy_talker.py`。

正像在订阅者中例子一样, 为了获取 `numpy_msg()`, 这里也有 `from rospy.numpy_msg import numpy_msg`。

当创建发布者实例时, 也会有这样的应用:

```
pub = rospy.Publisher('floats', numpy_msg(Floats))
```

上面的这一步, 实际上创建了用户自己的 `numpy` 阵列。由于 `numpy` 数据类型直接映射到 ROS 数据类型, 选择正确的数据类型是很容易的。在下面例子中, 创建了一个简单的 `numpy.float32` 阵列:

```
a = numpy.array([1.0, 2.1, 3.2, 4.3, 5.4, 6.5],  
    dtype=numpy.float32)
```

现在该节点可以运行了。在三个新终端分别输入:

```
roscore  
rosrun numpy_tutorial numpy_listener.py  
rosrun numpy_tutorial numpy_talker.py
```

运行结果:

```
/listener-1423-1248226794834 I heard [1. 2.0999999 3.20000005  
4.30000019 5.4000001 6.5]
```

```
/listener-1423-1248226794834 I heard [1. 2.0999999 3.20000005  
4.30000019 5.4000001 6.5]
```

... and so on

### 3.3.5 rospy 运行日志

输出消息到 `rosout` 是很容易的。但是当运行很多节点时, 从命令行看到输出结果变得很困难。相对地, 将调试消息输出到 `rosout` 并在 `rxconsole` 查看是比较容易的。

#### 1. rospy 应用程序接口

rospy 有几种方法记录登录信息, 所有命令以 `log` 开头:

```
rospy.logdebug(msg, *args)  
rospy.logwarn(msg, *args)
```

```
rospy.loginfo(msg, *args)
rospy.logerr(msg, *args)
rospy.logfatal(msg, *args)
```

这些级别和 `rosout` `verbosity levels` 是一一对应的。根据 `verbosity level`, 有四个潜在的地方可以放置登录消息:

- (1) `stdout`: `loginfo`.
- (2) `stderr`: `logerr`, `logfatal`, `logwarn` (ROS 0.9).
- (3) 用户节点登录文件: `all`.
- (4) `/rosout` 主题: `loginfo`、`logwarn`、`logerr`、`logfatal`.

注意: 节点完全初始化之后, 消息才会出现在 `/rosout` 主题, 所以用户可能看不到初始化消息。

节点登录文件可能在 `ROS_ROOT/log` 或者 `~/.ros/log`, 除非用户用环境变量 `ROS_LOG_DIR` 覆盖它。

如果用户希望在 `/rosout` 上看到 `logdebug` 消息, 可以传递参数 `log_level` 到 `init_node()`。

```
rospy.init_node('my_node', log_level=rospy.DEBUG)
```

每一个 `rospy.log*` 方法可以放在一个字符串 `msg` 内。如果 `msg` 是一个格式化字符串, 用户可以为字符串单独传递参数:

```
rospy.logerr("%s returned the invalid value %s", other_name,
            other_value)
```

## 2. 例子

下面是一个发布者节点的简单例子:

```
1 topic = 'chatter'
2 pub = rospy.Publisher(topic, String)
3 rospy.init_node('talker', anonymous=True)
4 rospy.loginfo("I will publish to the topic %s", topic)
5 while not rospy.is_shutdown():
6     str ="hello world %s"%rospy.get_time()
7     rospy.loginfo(str)
8     pub.publish(str)
9     rospy.sleep(0.1)
```

在上面代码的第 4、7 行可以看到 `rospy.loginfo()` 的调用。这些内容同时被输出到屏幕并写入 `stdout` 中。因此可以用 `print` 插入这些信息。

一种快速查看 `/rosout` 上消息的方法是使用 `rostopic echo`, 运行结果如下:

```
---
header:
  seq: 1
  stamp: 1247874193220264911
  frame_id: 0
level: 2
name: /talker-13716-1247874192981
msg: hello world 1247874193.14
file:
function:
line: 0
topics: ['/time', '/chatter', '/rosout']
```

### 3.3.6 ROS Python Makefile 文件

`rospy` 中的编译文件虽然简单，但它们提供了很重要的功能：

- ① 自动生成消息和服务代码。
- ② 运行测试。

当测试新加进来的新功能包时，可以在其依赖的功能包上测试其功能，此时测试功能就显得很重要。使用 `roscreate-pkg` 可自动创建这些必需的编译文件：

```
roscreate-pkg my_pkg rospy
```

上面的命令将创建 `my_pkg` 并增加其相应的 `rospy` 依赖项。

如果用户想手工编写编译文件，可以执行如下两步。

#### 1. Makefile

只需要一行代码即可：

```
include $(shell rospack find mk)/cmake.mk.
```

#### 2. CMakeLists.txt

用户还需要提供 `CMakeLists.txt`:

```

1 cmake_minimum_required(VERSION 2.4.6)
2 include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
3 rosbuild_init()
4 rosbuild_genmsg()
5 rosbuild_gensrv()
6 rosbuild_add_rostest(test/talker-listener-test.launch)

```

### 3.3.7 设置 PYTHONPATH

设置自己的 PYTHONPATH 是一件很容易的事情。如果用户已经编写 manifest.xml 文件来声明其依赖项。ROS 将会使用这些 manifest.xml 文件来帮助用户设置 PYTHONPATH。用户需要执行下面两步：

- ① 确认所需的依赖项已经列在 manifest 文件中
- ② 在 Python 代码首行 (只需主文件首行) 添加：

```
1 import roslib; roslib.load_manifest('your_package_name')
```

roslib 将会装载 manifest 文件并设置 sys.path，以便指向依赖项的相应目录即可。

### 3.3.8 发布消息

rospy 中发布消息相对简单。首先来看发布消息时调用的代码：

```
1 pub.publish(String(str))
```

由于 rospy 已经知道用户将要发布 std\_msgs.msg.String 对象，因此上面代码可以缩写为：

```
1 pub.publish(str)
```

rospy 将会自动创建 std\_msgs.msg.String 实例。

如果消息使用了多个参量，必须按照在消息文件中指定的顺序来指定它们，例如：rosmsg show std\_msgs/ColorRGBA。结果：

```

float32 r
float32 g
float32 b
float32 a

```

上面命令将会以 r、g、b 初始化，其中 a 作为参量：

```
1 pub.publish(0.1, 0.2, 0.3, 0.4)
```

上面的命令的缺点在于手工添加，比较繁琐，易出错。还有一种更加鲁棒的方法，可以省略任何以缺省值代替的域：可以使用 Python 关键参数来指定用户希望指定的缺省值：

```
1 pub.publish(a=1.0)
```

该命令将以 `a=1.0` 和缺省 (`r`, `g`, `b`) 为 0 来发布一条 `ColorRGBA` 消息。

## 3.4 roslisp 客户端库

`roslisp` 是 ROS 的 Lisp 语言客户端库。该客户端库具有易使用、节点快速脚本化、可交互式调试等优点。

`roslisp` 已经在 Ubuntu 系统上得到广泛测试，在 OS X 10.5 系统上得到部分测试。它是 C Turtle 版本的一部分。目前已经支持较少。

`roslisp` 是基于 Common Lisp 的 SBCL 版本实现的。为避免 ROS 功能包集过度依赖于 SBCL 编译器，在 `roslisp_support` 功能包集中，有一个仿制的 `roslisp_runtime` 功能包。使用 `roslisp` 的功能包应当在其 manifest 文件中声明依赖项。这样可以使得适当版本的编译器导入功能包。

详细信息可参考网址：<http://www.ros.org/wiki/roslisp>

## 3.5 实验阶段的客户端库

### 3.5.1 rosjava

`rosjava` 是支持 Android 系统的纯 Java 实现。`rosjava` 的作用是提供 Java 程序员能够快速与主题、服务和参数交互的能力。它同时也实现了 `roscore` 的 Java 版本。下面两种情况较适合于 `rosjava`：

- ① 与优先的 Java 程序或者库进行交互。
- ② 编写与 ROS 一起工作的 Android 系统应用。

详细信息可参考网址：<http://www.ros.org/wiki/rosjava>

### 3.5.2 roslua

`roslua`<sup>[15]</sup> 是 ROS 的 Lua 客户端库。它是一个轻量级但是很有用的嵌入式

脚本语言库，目前处于实验阶段。它允许使用 Lua 编程实现 ROS 节点、发布和订阅主题、提供和使用服务、与节点管理器交互等功能。

详细信息可以参考网址：

- <http://www.ros.org/wiki/roslua>
- <http://github.com/timn/roslua>



## 第四章 OpenCV

OpenCV 作为一个开源视觉库于 1999 年 1 月由英特尔公司发布。它包含许多先进的计算机视觉算法及机器学习工具。2009 年 11 月 23 日，OpenCV 宣布将代码仓库从 Source Forge 迁移到 ros.org。ROS 中自带 OpenCV 的代码：Diamondback 版本包含了 OpenCV 2.2，C Turtle 包含了 OpenCV 2.0。从 Electric 开始，它被设计成为 `rosdep` 的系统依赖项，可以通过修改 `manifest` 等，然后使用 `rosdep install <your-package>` 命令来安装需要的版本。为了向前兼容，目前 `opencv2` 依然被保留。在以后的版本中，这个包将不复存在。

这一章主要介绍 ROS 中与图像处理和计算机视觉相关的功能包。其中 `image_common` 属于底层驱动，`image_pipeline` 主要用于 OpenCV<sup>[16]</sup> 与 ROS 之间的图像格式转换。`vision_opencv` 是 OpenCV 相关内容。

### 4.1 `image_common` 功能包集

`image_common` 功能包集有四个功能包：

(1) `image_transport`: 在低带宽压缩格式传输图像时提供支持。可以应用于任何发布或者订阅图像的节点。

(2) `camera_calibration_parsers`: 包含了用于读写摄像机标定参数的路线。主要用于相机驱动。

(3) `camera_info_manager`: 用于存储、再存储和设置摄像机标定信息的 C++ 界面。

(4) `polled_camera`: 定义一个 ROS 界面用于从一个修正过的摄像机驱动中请求图像。

#### 4.1.1 `image_transport` 功能包

`image_transport` 主要提供低带宽压缩格式图像（包括 JPEG/PNG 压缩及 theora 视频流）的传输支持。发布和订阅图像的用户均会用到它。

当传输图像时，需要指定特定的传输策略，如采用图像压缩还是视频流编码。`image_transport` 提供了类和节点用于任意单线表示 (over-the-wire) 的图像传输。

`image_transport` 仅提供了原始格式传输，特定的传输由插件来提供，这样可以避免增加功能包的不必要的依赖项。常用的系统会自带一些传输方式，在这些系统上则可以通过插件来完成这些特定方式的传输。Ubuntu 系统中，`ros-<distro>-base debians` 就包含了由 `image_transport_plugins` 功能包集提供的 compressed 和 theora 传输。

`image_transport` 必须应用于发布和订阅图像。在这些基本应用中，其作用非常类似于主题发布者和主题订阅者。使用 `image_transport` 来代替 ROS 中的组件，给节点之间互相传输图像带来极大便利。

## 1. 例子

下面给出一个 C++ 语言编写的 `image_transport` 例子：

```
1 #include <ros/ros.h>
2 #include <image_transport/image_transport.h>
3
4 void imageCallback(const sensor_msgs::ImageConstPtr& msg)
5 {
6     // ...
7 }
8
9 ros::NodeHandle nh;
10 image_transport::ImageTransport it(nh);
11 image_transport::Subscriber sub = it.subscribe
12     ("in_image_base_topic", 1, imageCallback);
13 image_transport::Publisher pub = it.advertise
14     ("out_image_base_topic", 1);
```

实际上，更高级的功能是，`image_transport` 主题发布者为每个有效的主题选择一种独立的传输方式，而 ROS 主题发布者则只能对同一个主题进行单一方式的广播。需要注意的是，所有同一种图像的所有接口共用一个基本主题名称 `base_topic`，这使得用户在获取图像时，不必刻意考虑如何引用特殊插件和对应的主题。如双目摄像机左侧的原始图像是 `stereo/left/image` (`sensor_msgs/Image`)，如果使用插件用 compressed 方式传输，则使用主题 `stereo/left/image/compressed` 就可以获取压缩后的图像。

`image_transport` 不支持 Python。如果 Python 节点需要传输压缩图像，需要通过 republish 节点实现。

## 2. 传输功能包

- `image_transport ("raw")`: 缺省传输方式, 通过 ROS 发送 `sensor_msgs/Image`。
- `compressed_image_transport ("compressed")`: JPEG 或 PNG 图像压缩。
- `theora_image_transport ("theora")`: 采用 theora 编码器传输视频流。

## 3. ROS 应用程序接口

ROS 主题发布者和主题订阅者可用于传输任意类型的消息。`image_transport` 提供给主题发布者和主题订阅者针对图像的特定类型。由于封装了复杂的通信行为, `image_transport` 主题发布者和主题订阅者有公用的 ROS 应用程序接口和 C++ 代码应用程序接口。

### 1) `image_transport` 主题发布者

`image_transport` 主题发布者的用法很像 ROS 主题发布者, 但是前者提供了很多特定的选项 (JPEG 压缩、视频流等)。不同的主题订阅者可能要求同一个主题发布者使用不同的传输方式。

#### (1) 发布的主题。

原始的 `sensor_msgs/Image` 发布在基本主题上, 就像是使用标准的 `roscpp ros::Publisher`。如果额外的插件可用, 它们广播基本主题的子主题, 传统上采取如下形式: `<base topic>/<transport name>`。例如, 使用插件, 采用 compressed 和 theora 方式传输, 其基本主题为 `/stereo/left/image`, 则对应的主题应该为:

① 基本主题用于原始传输, 此时类似于使用 `ros::Publisher`:

`stereo/left/image (sensor_msgs/Image)`

② 用 compressed 方式传输主题:

`stereo/left/image/compressed (<transport-specific type>)`

③ 用 theora 方式传输主题:

`stereo/left/image/theora (<transport-specific type>)`

④ 摄像机信息主题 (`image_transport::CameraPublisher-only`):

`stereo/left/camera_info (sensor_msgs/CameraInfo)`

#### (2) 参数。

`image_transport` 主题发布者没有独立的参数, 但是插件允许利用参数服务

器来配置选项，如比特率、压缩水平等。

为适应客户端的需要，主题发布者插件参数由主题订阅者插件接口来配置。查找是从定义基本主题名称 (<base\_topic>) 的公共命名空间开始的，而不是从发布节点的私有命名空间开始。这些参数是共享的资源，用于对这些图像的订阅者的观测行为进行操控和设置。

传统上，主题发布者插件参数采取如下形式：

```
<base topic>/<transport name>_image_transport_<parameter name>
(type, default: ?)
```

上述命令用于传输特定主题发布者参数。如果在命名空间 (<base\_topic>) 中没有发现参数，插件就搜索事先设定的参数树。

一个具体的例子如下：

```
stereo/compressed_image_transport_jpeg_quality
(int, default: 80)
```

## 2) image\_transport 主题订阅者

image\_transport 主题订阅者的用法很像 roscpp 中的 `ros::Subscriber`，但是它可以指定特定传输方式来接收图像。

### (1) 订阅的主题。

一个主题订阅者实例的名称包括：基本主题名称 + 传输名称。它订阅与基本主题相关的 ROS 特定传输主题。例如：基本主题为 /stereo/left/image，用于 raw 和 compressed 传输方式的订阅主题分别为：

① 基主题用于原始传输，此时类似于使用 `ros::Subscriber`：

```
stereo/left/image (sensor_msgs/Image)
```

② 用 compressed 方式传输主题：

```
stereo/left/image/compressed (<transport-specific type>)
```

③ 用 theora 方式传输主题：

```
stereo/left/image/theora (<transport-specific type>)
```

④ 摄像机信息主题 (`image_transport::CameraSubscriber-only`)：

```
stereo/left/camera_info (sensor_msgs/CameraInfo)
```

### (2) 参数。

用于传输用途的名称：`image_transport` (`string, default: "raw"`)。如果参数没有设置，来自 `image_transport::ImageTransport::subscribe()` 的参数 `image_transport::TransportHints` 的传输方式将会被使用。

`image_transport::TransportHints` 可能只用于为参数查找表指定一个不同的参数空间。这在再映射 `image_transport` 到独立的命名空间时很有用：这样做可以允许为不同的图像订阅不同的传输方式。用户可能会指定一个参数名称而不是使用 `image_transport`，此时用于图像主题的节点应该指定哪个参数来控制传输，尤其是在使用不同于 `image_transport` 的名称。

主题订阅者插件允许利用参数服务器来配置选项。主题订阅者插件参数配置了特定主题订阅者的行为，它们影响了数据解码的方式。这一点不同于主题发布者插件参数，主题发布者插件参数共享一个源来共享发送给主题订阅者的数据。用于参数查找表的命名空间是通过 `image_transport::TransportHints` 指定的，缺省情况下为主题订阅者节点的私有命名空间。

传统上，主题订阅者插件参数取如下形式：

```
~<transport name>_image_transport_<parameter name>
  (type, default: ?)
```

一个具体的例子如下：

```
~foo_image_transport_post_processing_level (int, default: 1)
```

#### 4. 节点

##### 1) 再发布

`republish` 收听一个基本图像主题（可以使用任何类型，缺省为 `raw`），然后重新发布图像到另一个基本主题。发布消息 `sensor_msgs/Image` 的缺省传输方式为 `raw`。其他传输方式是否可用依赖于所安装的插件。

##### 2) 用法

```
republish in_transport in:=<in_base_topic> [out_transport]
  out:=<out_base_topic>
```

##### 3) 例子

(1) 假设通过一个机器人使用 `theora` 传输方式发布图像。在另外一台计算机上有几个节点收听图像主题。这种设置方式由于每个节点都要独立处理压缩视频为静止图像，浪费了带宽和计算资源。取而代之，可以在另外一台计算机开始一个再发布节点，视频流只需要给处理节点发送消息 `sensor_msgs/Image`，再发送到别的节点：

```
republish theora in:=camera/image raw out:=camera/
  image_decompressed
```

(2) 上面命令对只把压缩图像发送到可以收听 `sensor_msgs/Image` 的节点的情况非常有用。

(3) 如果一个节点把图像发布为 `sensor_msgs/Image`, 用户可以使用完整传输方式重新发布它。由于基本主题必须与原始主题不同, 这种方法比在原始节点使用 `image_transport::Publisher` 开销稍微大一些。

```
republish raw in:=camera/image out:=camera/image_repub
```

#### 4) 订阅的主题

图像基本主题到哪里订阅信息:

```
in (<transport-specific type>)
```

#### 5) 发布的主题

图像基本主题将信息发布到哪里:

```
out (<transport-specific type>)
```

#### 6) 参数

再发布本身不使用参数服务器, 但插件可以读或设置针对特定插件的参数。

### 5. 命令行工具

`list_transports` 列出了所有 ROS 功能包声明的图像传输选项, 并尝试确定它们当前是否可用 (功能包编译, 插件能够正确加载等)。用法:

```
rosrun image_transport list_transports
```

#### 4.1.2 camera\_calibration\_parsers 功能包

`camera_calibration_parsers` 包含读写摄像机标定参数的内容, 主要用于摄像机驱动和摄像机标定工具, 其代码格式是人工可读的。

##### 1. 命令行工具

`convert` 命令: 用于标定文件之间的格式转换。

```
rosrun camera_calibration_parsers convert .in-file out-file
```

从 yml 到 ini 格式转换例子:

```
rosrun camera_calibration_parsers convert cal.yml cal.ini
```

从 ost.txt 到 yml 格式转换例子:

```
mv ost.txt ost.ini
```

```
rosrun camera_calibration_parsers convert ost.ini cal.yml
```

## 2. 文件格式

### 1) Videre INI

该格式用于 Videre 摄像机存储标定摄像机内参数。这种格式作为首选格式，在摄像机闪存上存储标定参数。例子：

```
# Prosilica camera intrinsics
```

```
[image]
```

```
width
```

```
2448
```

```
height
```

```
2050
```

```
[prosilica]
```

```
camera matrix
```

```
4827.93789 0.00000 1223.50000  
0.00000 4835.62362 1024.50000  
0.00000 0.00000 1.00000
```

```
distortion
```

```
-0.41527 0.31874 -0.00197 0.00071 0.00000
```

```
rectification
```

```
1.00000 0.00000 0.00000  
0.00000 1.00000 0.00000  
0.00000 0.00000 1.00000
```

```
projection
```

```
4827.93789 0.00000 1223.50000 0.00000  
0.00000 4835.62362 1024.50000 0.00000  
0.00000 0.00000 1.00000 0.00000
```

### 2) YAML

yaml 输出是基于使用 OpenCV 标定程序的。例子：

```
image_width: 2448
```

```
image_height: 2050
camera_name: prosilica
camera_matrix:
  rows: 3
  cols: 3
  data: [4827.94, 0, 1223.5, 0, 4835.62, 1024.5, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5
  data: [-0.41527, 0.31874, -0.00197, 0.00071, 0]
rectification_matrix:
  rows: 3
  cols: 3
  data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:
  rows: 3
  cols: 4
  data: [4827.94, 0, 1223.5, 0, 0, 4835.62, 1024.5, 0, 0, 0,
         1, 0]
```

### 4.1.3 camera\_info\_manager 功能包

#### 1. 概述

`camera_info_manager` 是用于保存、再存及设置摄像机标定信息的 C++ 接口的功能包。主要用于摄像机驱动，它提供了一个 C++ 类可供很多摄像机驱动用于管理由 `image_pipeline` 请求的摄像机标定数据。它提供了 `CameraInfo` 和句柄 `SetCameraInfo` 服务请求，保存及再存摄像机标定数据的功能。

在 Diamondback 版本中，该功能包移动到功能包集 `image_common`。在 C Turtle 版本中，它是 `camera_drivers` 的一部分。

在 Diamondback 版本中，`CameraInfoManager` 类移动至命名空间 `camera_info_manager`。为了兼容 C Turtle 版本，仍然支持全局类名称 `CameraInfoManager`。在 Electric 版本中，将会被移除。

## 2. 摄像机名称

当保存数据时, `CameraInfo` 需要应用程序接口包含摄像机名称, 并在数据加载时检查数据, 当读取数据名称不匹配时, 发出警报。

获取和保存标定数据时的位置由 Uniform Resource Locator (URL) 来表示。

## 3. ROS 应用程序接口

当实例化一个摄像机驱动节点, `CameraInfoManager C++` 类提供下面 ROS 接口。

服务: `set_camera_info(sensor_msgs/SetCameraInfo)` 用于标定信息;

参数: 该功能包不直接读取 ROS 参数。这里推荐用户使用 `camera_info_url` 参数, 该参数用于把 URL 字符串传递到摄像机信息管理器 (`CameraInfoManager`)。

## 4. C++ 应用程序接口

例子: `camera_info_manager` 用于 `camera1394` 驱动。用户可在源代码中搜索 `cinfo_` 来查看类 `CameraInfoManager` 被用在哪里, 尤其是要注意头文件时间戳和转换坐标系 ID 的处理。

### 4.1.4 polled\_camera 功能包

`polled_camera` 包含了一个服务和 C++ 帮助类用于矫正的摄像机驱动程序和请求图像。现阶段该功能包还在发展, 主要用于内部。

## 4.2 image\_pipeline 功能包集

### 1. 概述

`image_pipeline` 功能包集填补了从摄像机驱动器获取原始图像和高水平视觉处理之间的空白, 它可以处理校正的黑白/彩色图像, 立体视差图像, 立体点云。具体包括:

(1) 标定: 摄像机在使用前必须标定。`camera_calibration` 功能包提供了用于标定单目和双目立体视觉摄像机的工具。

(2) 单目摄像机处理: 原始视频流必须通过 `image_proc` 节点来移除摄像机畸变, 该节点也处理用于 Bayer 模式彩色摄像机的彩色图像插值。

(3) 双目摄像机处理: `stereo_image_proc` 节点执行类似于 `image_proc` 节点的功能, 实现双目图像的畸变移除。它也可以用于生成视差图像和点云。

(4) 可视化: `image_view` 功能包提供了用于替代 `rviz` 的轻量级的看图工具, 它也提供了 `image_view` 工具用于查看立体图像和视差图。

## 2. 硬件需求

摄像机驱动主要由 `camera_drivers` 来维护。这样的节点的最低硬件需求为:

(1) 一般摄像机节点。

(2) 发布的主题。

① 未处理的图像数据: `image_raw` (`sensor_msgs/Image`)

② 包含标定信息及其他摄像机配置的数据:

`camera_info` (`sensor_msgs/CameraInfo`)

(3) 服务。用于 `camera_calibration` 保存参数:

`set_camera_info` (`sensor_msgs/SetCameraInfo`)

正常情况下, 摄像机驱动带有用户在运行时可配置的参数。`stereo_image_proc` 带有自己的运行时可实时配置的参数。`dynamic_reconfigure` 中 `reconfigure_gui` 在调整参数以达到最好结果时非常有用。

## 4.3 vision\_opencv 功能包集

`vision_opencv` 功能包集提供了使用 C++ 和 Python 的 OpenCV 功能包。其中包含下列功能包:

(1) `opencv2`: OpenCV 2.2 库 (Diamondback) 或 2.0 (C Turtle)。

(2) `cv_bridge`: 介于 ROS 消息和 OpenCV 之间的桥接。

(3) `image_geometry`: 处理图像与像素几何的方法集合。

### 4.3.1 opencv2

`opencv2` 功能包包含了 OpenCV 2.2 库 (Diamondback) 或 2.0 (C Turtle)。从 Electric 版本开始, OpenCV 可以作为 `rosdep` 系统依赖项来使用。

### 4.3.2 cv\_bridge

ROS 与 OpenCV 之间的关系框架图如图 4.1 所示。

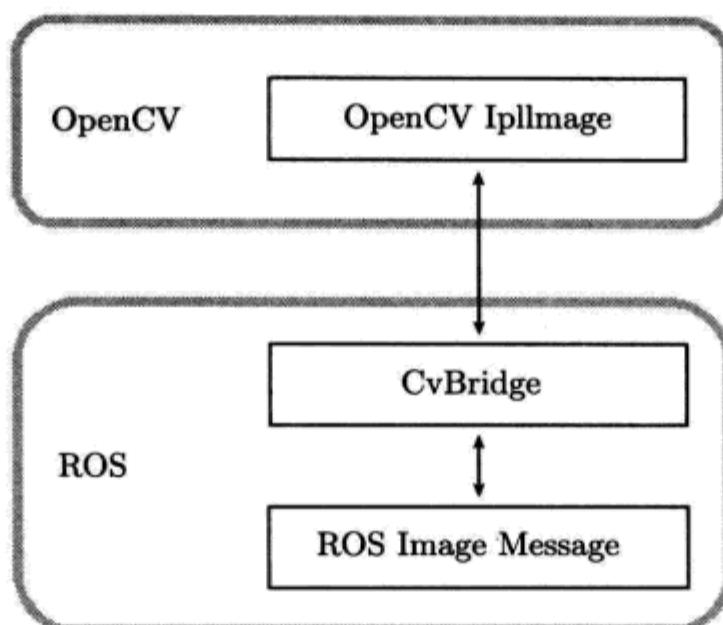


图 4.1 ROS 与 OpenCV 关系框架图

## 1. ROS 图像与 OpenCV 图像之间互相转换 (Python)

本小节讲述怎样使用 `cv_bridge` 在 ROS 和 OpenCV 之间建立接口，以便在 ROS 图像和 OpenCV 图像之间进行转换。这里给出一个例子，可以用作用户代码的模板。

### 1) 概念

ROS 用 `sensor_msgs/Image` 消息格式来传输图像，但是许多用户希望使用 OpenCV 来处理图像。`CvBridge` 提供了 ROS 和 OpenCV 之间的接口。`CvBridge` 位于 `vision_opencv` 功能包集下的 `cv_bridge` 功能包内。

### 2) 转换 ROS 图像消息为 OpenCV 图像

为了转换 ROS 图像消息为 `IplImage`，`cv_bridge.CvBridge` 模块提供了下列函数：

```
1 cv_image = bridge.imgmsg_to_cv(image_message, encoding="passthrough")
```

输入是图像消息和一个可选的编码参量。编码指的是最终的 `IplImage`。

如果给定缺省值 `passthrough`，目标图像编码与图像消息编码一致。图像编码可以是下列 OpenCV 图像编码的任意一种：

- 8UC[1-4]
- 8SC[1-4]
- 16UC[1-4]
- 16SC[1-4]

- 32SC[1-4]
- 32FC[1-4]
- 64FC[1-4]

对于常用的图像编码格式，`CvBridge` 将有选择地使用彩色或者像素深度转换格式。根据这一特性，制定如下编码格式：

- mono8: CV\_8UC1, 灰度尺度图像
- mono16: CV\_16UC1, 16 位灰度尺度图像
- bgr8: CV\_8UC3, BGR 彩色图像
- rgb8: CV\_8UC3, RGB 彩色图像
- bgra8: CV\_8UC4, 带有 alpha 通道的 BGR 彩色图像
- rgba8: CV\_8UC4, 带有 alpha 通道的 RGB 彩色图像

注意：mono8 和 bgr8 是 OpenCV 函数常用的两种图像编码格式。

### 3) 转换 OpenCV 图像为 ROS 图像消息

为了转换 `IplImage` 为 ROS 图像消息，`CvBridge` 提供了下列函数：

```
1 image_message = cv_to_imgmsg(cv_image, encoding=
    "passthrough")
```

这种情况下，参数 `encoding` 的使用稍微复杂一些。正像前面一样，它首选 `IplImage`。但是 `cv_to_imgmsg()` 并没有做任何转换（使用 `CvtColor` 和 `ConvertScale` 来代替）。ROS 必须总是与 `IplImage` 具有同样的通道数量和像素深度。此时上述通常使用的特定格式（bgr8、rgb8 等）将插入关于通道顺序的信息到图像消息中。这样，将来的用户将会知道接收到的图像是 RGB 或 BGR。

### 4) 一个 ROS 节点例子

本节展示一个节点的例子，将图像转换成 `IplImage`，并在上面画圆，然后使用 OpenCV 显示该图像，并将图像重新发布给 OpenCV。

在用户的 `manifest` 文件中，添加如下依赖项：

```
sensor_msgs
opencv2
cv_bridge
rospy
std_msgs
```

例子源代码：

```
1 #!/usr/bin/env python
2 import roslib
3 roslib.load_manifest('my_package')
4 import sys
5 import rospy
6 import cv
7 from std_msgs.msg import String
8 from sensor_msgs.msg import Image
9 from cv_bridge import CvBridge, CvBridgeError
10
11 class image_converter:
12
13     def __init__(self):
14         self.image_pub = rospy.Publisher("image_topic_2",Image)
15
16         cv.NamedWindow("Image window", 1)
17         self.bridge = CvBridge()
18         self.image_sub = rospy.Subscriber("image_topic",Image,
19                                         self.callback)
20
21     def callback(self,data):
22         try:
23             cv_image = self.bridge.imgmsg_to_cv(data,"bgr8")
24         except CvBridgeError, e:
25             print e
26
27         (cols,rows) = cv.GetSize(cv_image)
28         if cols > 60 and rows > 60 :
29             cv.Circle(cv_image, (50,50), 10, 255)
30
31         cv.ShowImage("Image window", cv_image)
32         cv.WaitKey(3)
33
34     try:
35         self.image_pub.publish(self.bridge.cv_to_imgmsg(
36             cv_image, "bgr8"))
37     except CvBridgeError, e:
38         print e
```

```

38 def main(args):
39     ic = image_converter()
40     rospy.init_node('image_converter', anonymous=True)
41     try:
42         rospy.spin()
43     except KeyboardInterrupt:
44         print("Shutting down")
45         cv.DestroyAllWindows()
46
47 if __name__ == '__main__':
48     main(sys.argv)

```

最后还需要增加执行权限以便执行该文件: `chmod +x`。

代码解释:

(1) 代码第 6 行。通过导入 `cv`, 所有的 OpenCV 头文件被包含进来。在 `manifest` 文件中, 添加依赖项到 `opencv2` 和 `cv_bridge`。

(2) 代码第 9 行。`CvBridge` 也存在于 `cv_bridge`。

(3) 代码第 21~24 行。转换图像消息指针到 OpenCV 消息仅需要调用函数 `imgmsg_to_cv()`。为了运行该节点, 需要有图像流输入进来。用户可以安装摄像头传感器或者打开视频文件, 以生成图像流。现在可以运行该节点, 重映射图像流主题到 `image_topic`。如果运行成功, 可以看到名为 `Image window` 的 `HighGui` 窗口, 其中显示了图像并在上面画出一个红色圆圈。

(4) 代码第 33~36 行。通过调用函数带有编码 `bgr8` 的 `cv_to_imgmsg()`, 被编辑的图像可以被转换回 ROS 图像消息格式, 以便将来的用户知道颜色顺序。

用户可以使用 `rostopic` 命令或者通过 `image_view` 查看图像来确认节点是否正确地通过 ROS 发布了图像。

## 2. ROS 图像与 OpenCV 图像之间互相转换 (C++)

本小节讲述怎样使用 `cv_bridge` 在 ROS 和 OpenCV 之间建立接口, 以便在 ROS 图像和 OpenCV 图像之间进行转换(见图 4.2)。这里给出一个例子, 可以用作用户代码的模板。

### 1) 概念

ROS 用 `sensor_msgs/Image` 消息格式来传输图像, 但是许多用户希望使用 OpenCV 来处理图像。`CvBridge` 提供了 ROS 和 OpenCV 之间的接口。`cv_bridge` 位于 `vision_opencv` 功能包集下的 `cv_bridge` 功能包内。

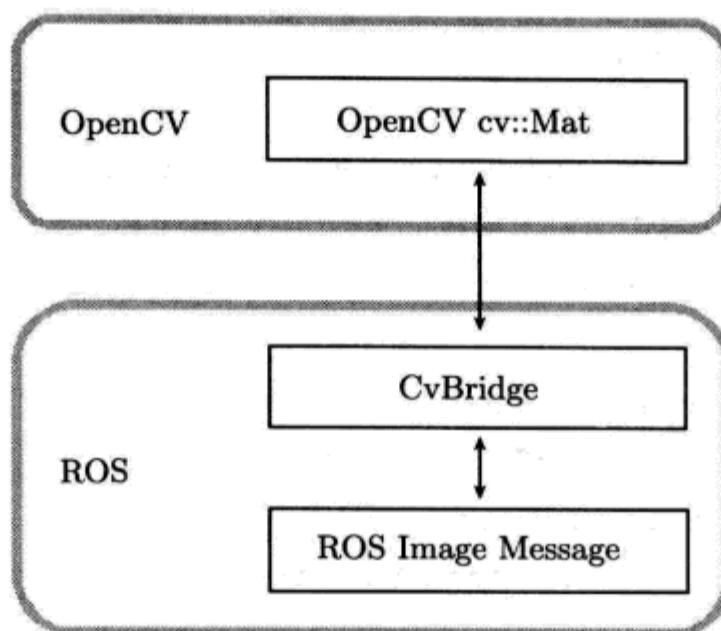


图 4.2 ROS 图像与 OpenCV 图像之间互相转换 (C++)

## 2) 转换 ROS 图像消息为 OpenCV 图像

`CvBridge` 定义了一个 `CvImage` 数据类，该数据类型包括 OpenCV 图像、编码和一个 ROS 头文件。`CvImage` 准确包含了 `sensor_msgs/Image` 具有的信息，因此可以在两种表达方式之间切换。

```
1 namespace cv_bridge {  
2  
3     class CvImage  
4 {  
5     public:  
6         std_msgs::Header header;  
7         std::string encoding;  
8         cv::Mat image;  
9     };  
10  
11     typedef boost::shared_ptr<CvImage> CvImagePtr;  
12     typedef boost::shared_ptr<CvImage const> CvImageConstPtr;  
13  
14 }
```

当把 ROS `sensor_msgs/Image` 转换为 `CvImage` 时，`CvBridge` 需要识别下列两种情况：

- (1) 用户需要修改数据：直接拷贝数据。
- (2) 用户不需要修改数据：共享数据，然后返回 `const CvImage`。

`CvBridge` 提供下列函数来转换成 `CvImage`：

```

1 // Case 1: Always copy, returning a mutable CvImage
2 CvImagePtr toCvCopy(const sensor_msgs::ImageConstPtr&
3   source, const std::string& encoding = std::string());
4
5 // Case 2: Share if possible, returning a const CvImage
6 CvImageConstPtr toCvShare(const sensor_msgs::ImageConstPtr&
7   source, const std::string& encoding = std::string());
8 CvImageConstPtr toCvShare(const sensor_msgs::Image& source,
9   const boost::shared_ptr<void const>& tracked_object,
10  const std::string& encoding = std::string());

```

输入是图像消息指针和一个可选的编码参量。编码指的是最终的 `CvImage`。

第一种情况：即使源码和目标编码一致，`toCvCopy` 仍然从 ROS 消息创建图像数据拷贝。用户仍可以修改返回的 `CvImage`。

第二种情况：源码和目标代码一致，为了避免拷贝，`toCvShare` 指向返回的 ROS 消息数据 `cv::Mat`。只要用户在使用返回的 `CvImage`，ROS 消息不能被释放。如果编码不匹配，重新分配缓冲并转换。由于可能与 ROS 图像共享数据，不允许用户修改返回的 `CvImage`。

说明：当有包含需要转换的 `sensor_msgs/Image` 指向其他消息类型的指针时，第二种重载方式 `toCvShare` 更方便。

若没有给定编码（甚至给定空字符串），目标图像编码与图像消息编码一致。这种情况下，`toCvShare` 保证了不拷贝图像数据。图像编码可以是下列 OpenCV 图像编码的任意一种：

- 8UC[1-4]
- 8SC[1-4]
- 16UC[1-4]
- 16SC[1-4]
- 32SC[1-4]
- 32FC[1-4]
- 64FC[1-4]

对于常用的图像编码格式，`CvBridge` 将有选择地使用彩色或者像素深度转换格式。根据这一特性，制定如下编码格式：

- mono8: CV\_8UC1, 灰度尺度图像
- mono16: CV\_16UC1, 16 位灰度尺度图像
- bgr8: CV\_8UC3, BGR 彩色图像
- rgb8: CV\_8UC3, RGB 彩色图像
- bgra8: CV\_8UC4, 带有 alpha 通道的 BGR 彩色图像
- rgba8: CV\_8UC4, 带有 alpha 通道的 RGB 彩色图像

注意: mono8 和 bgr8 是 OpenCV 函数常用的两种图像编码格式。最后, CvBridge 将 Bayer 格式识别为 OpenCV 类型 8UC1(单通道 8-bit 无符号整型)。这种情况下不需要从 Bayer 格式转换, 或者转换为 Bayer 格式。在 ROS 系统中, 这是由 `image_proc` 来实现的。CvBridge 可以识别下列 Bayer 格式:

- bayer\_rggb8
- bayer\_bggr8
- bayer\_gbrg8
- bayer\_grbg8

### 3) 转换 OpenCV 图像为 ROS 图像消息

使用 `toImageMsg()` 成员函数将 `CvImage` 转换为 ROS 图像消息:

```
1 class CvImage
2 {
3     sensor_msgs::ImagePtr toImageMsg() const;
4
5     void toImageMsg(sensor_msgs::Image& ros_image) const;
6 };
```

如果用户已经发布了 `CvImage`, 还需要添加头文件和编码域。

### 4) 一个 ROS 节点例子

本节展示一个节点的例子, 将图像转换成 `cv::Mat`, 并在上面画圆, 然后使用 OpenCV 显示该图像, 并将图像重新发布给 OpenCV。

在用户的 `manifest` 文件中, 添加如下依赖项:

```
sensor_msgs
opencv2
cv_bridge
roscpp
std_msgs
image_transport
```

例子源代码：

```
1 #include <ros/ros.h>
2 #include <image_transport/image_transport.h>
3 #include <cv_bridge/cv_bridge.h>
4 #include <sensor_msgs/image_encodings.h>
5 #include <opencv2/imgproc/imgproc.hpp>
6 #include <opencv2/highgui/highgui.hpp>
7
8 namespace enc = sensor_msgs::image_encodings;
9
10 static const char WINDOW[] = "Image window";
11
12 class ImageConverter
13 {
14     ros::NodeHandle nh_;
15     image_transport::ImageTransport it_;
16     image_transport::Subscriber image_sub_;
17     image_transport::Publisher image_pub_;
18
19 public:
20     ImageConverter()
21         : it_(nh_)
22     {
23         image_pub_ = it_.advertise("out", 1);
24         image_sub_ = it_.subscribe("in", 1, &ImageConverter::
25             imageCb, this);
26         cv::namedWindow(WINDOW);
27     }
28
29     ~ImageConverter()
30     {
31         cv::destroyWindow(WINDOW);
32     }
33
34     void imageCb(const sensor_msgs::ImageConstPtr& msg)
35     {
36         cv_bridge::CvImagePtr cv_ptr;
37         try
```

```
38     {
39         cv_ptr = cv_bridge::toCvCopy(msg, enc::BGR8);
40     }
41     catch (cv_bridge::Exception& e)
42     {
43         ROS_ERROR("cv_bridge exception: %s", e.what());
44         return;
45     }
46
47     if (cv_ptr->image.rows > 60 && cv_ptr->image.cols > 60)
48         cv::circle(cv_ptr->image, cv::Point(50, 50), 10,
49                     CV_RGB(255, 0, 0));
50
51     cv::imshow(WINDOW, cv_ptr->image);
52     cv::waitKey(3);
53
54     image_pub_.publish(cv_ptr->toImageMsg());
55 }
56
57 int main(int argc, char** argv)
58 {
59     ros::init(argc, argv, "image_converter");
60     ImageConverter ic;
61     ros::spin();
62     return 0;
63 }
```

代码解释：

(1) 代码第 2 行。在 ROS 中使用 `image_transport` 来发布和订阅图像，这样用户可以订阅压缩的图像流。这需要用户在 `manifest` 文件中包含 `image_transport`。

(2) 代码第 3、4 行。包含用于 `CvBridge` 和与图像编码相关的有用常量和函数。需要用户在 `manifest` 文件中包含 `cv_bridge`。

(3) 代码第 5、6 行。包含 OpenCV 图像处理和 GUI 模块的头文件。需要用户在 `manifest` 文件中包含 `opencv2`。

(4) 代码第 12~24 行。使用 `image_transport` 来订阅一幅图像，并发布该图像。

(5) 代码第 26~32 行。OpenCV HighGUI 在开始和结束时创建和销毁图像窗口。

(6) 代码第 34~45 行。在主题订阅者回调时，首先把 ROS 图像消息转换为 CvImage，以配合 OpenCV。下面需要显示图像，因此使用 toCvCopy()。`sensor_msgs::image_encodings::BGR8` 是用于 bgr8 的常量并且不易出现拼写错误。

(7) 代码第 47~51 行。在图像上画一个红色圆圈，并在图像窗口显示。

(8) 代码第 53 行。转换 CvImage 为 ROS 图像消息并发布它到输出主题。

为了运行该节点，需要有图像流输入进来。用户可以安装摄像机传感器或者打开视频文件，以生成图像流。然后运行该节点，重映射输入到实际的图像流主题。如果运行成功，可以看到名为 `Image window` 的 HighGUI 窗口，其中显示了图像并在上面画出一个红色圆圈。用户可以使用 `rostopic` 命令或者通过 `image_view` 来查看图像，来确认节点是否正确地通过 ROS 发布了图像。

### 5) 共享图像数据的例子

在上述例子中，通过精确拷贝图像来实现相关功能，实际上共享也是很容易实现的：

```

1 namespace enc = sensor_msgs::image_encodings;
2
3 void imageCb(const sensor_msgs::ImageConstPtr& msg)
4 {
5     cv_bridge::CvImageConstPtr cv_ptr;
6     try
7     {
8         cv_ptr = cv_bridge::toCvShare(msg, enc::BGR8);
9     }
10    catch (cv_bridge::Exception& e)
11    {
12        ROS_ERROR("cv_bridge exception: %s", e.what());
13        return;
14    }
15 }
```

如果输入消息有 bgr8 编码，`cv_ptr` 将会使用其别名而不是拷贝它。如果消息有别的可以转换的编码格式，例如 mono8，`CvBridge` 将分配新的缓冲并执行转换。除掉异常处理外，这只需要一行代码，但是如果输入消息带有不支持的

格式，将使得代码无法执行。例如，输入图像来自于主题为 `image_raw` 的 Bayer 模式摄像机，`CvBridge` 将会出现异常，因为它不支持从 Bayer 模式到彩色图像的自动转换。

一个更复杂的例子：

```
1 namespace enc = sensor_msgs::image_encodings;
2
3 void imageCb(const sensor_msgs::ImageConstPtr& msg)
4 {
5     cv_bridge::CvImageConstPtr cv_ptr;
6     try
7     {
8         if (enc::isColor(msg->encoding))
9             cv_ptr = cv_bridge::toCvShare(msg, enc::BGR8);
10    else
11        cv_ptr = cv_bridge::toCvShare(msg, enc::MONO8);
12    }
13    catch (cv_bridge::Exception& e)
14    {
15        ROS_ERROR("cv_bridge exception: %s", e.what());
16        return;
17    }
18 }
```

在这种情况下，用户希望如果有彩色图像，就使用彩色图像，否则使用 monochrome。如果输入图像是 bgr8 或 mono8，则可以避免拷贝数据。

### 4.3.3 image\_geometry

`image_geometry` 包含了用于解释图像几何的 C++ 和 Python 库。它在 `sensor_msgs/CameraInfo` 消息中的标定参数和 OpenCV (如图像矫正) 之间建立了接口。就像 `cv_bridge` 在 ROS `sensor_msgs/Image` 与 OpenCV 数据类型之间建立接口几乎一样。

虽然 `CameraInfo` 包含了用于矫正图像的所有信息，但是由于在所有摄像机上正确操作这些选项并不简单，因此仍然推荐用户使用该库。

`CameraInfo` 中的摄像机参数是用于全分辨率图像的。感兴趣区域使得创建矫正图像变的复杂，并需要调整投影矩阵。增加如 `subsampling (binning)` 之类的选项到 `CameraInfo` 中，将使得正确解释对应图像变得更加复杂。使用

`image_geometry` 简化了这一过程，并且它是一项不会过时的技术。

`image_geometry` 类被用于 `Image/CameraInfo` 消息进行回调，作用类似于 `cv_bridge`。为了维持不变性，`CameraModel` 类提供了为现有摄像机指定参数和矩阵的功能。设置 `CameraModel` 只需要使用函数 `fromCameraInfo()` 即可。这些类使用 OpenCV 摄像机模型。OpenCV 指定格式的摄像机矩阵很容易实现。

## 4.4 投影 tf 坐标系到图像 (C++)

本小节展示怎样使用 `image_geometry` 和 `tf` 投影 `tf` 坐标系到图像流中。这里将建立一个节点，该节点投影一个坐标系或多个 `tf` 坐标系到图像坐标，并且将它画在图像上。虽然在数学上这是很简单的操作，但由于它集成了多个不同的软件包（如 OpenCV、`tf`、`bag` 文件，`image_transport` 和 `image_geometry`），使得操作变得复杂。结果如图 4.3 所示。

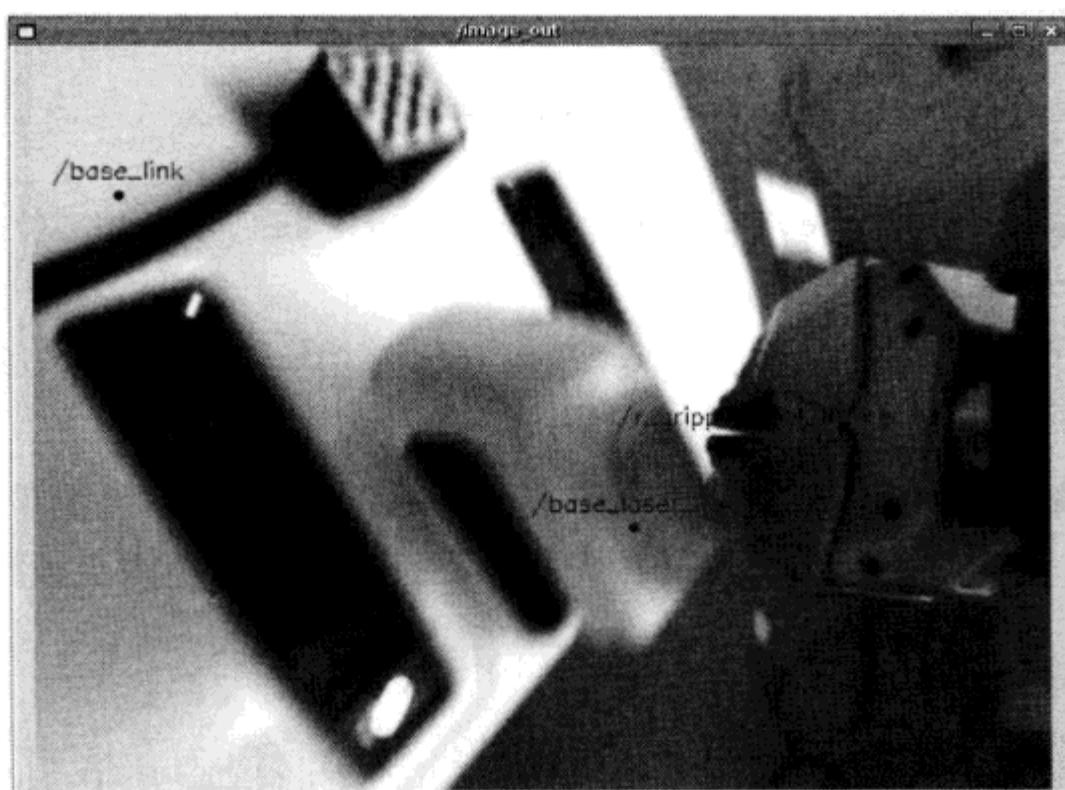


图 4.3 投影 `tf` 坐标系到图像

### 1. 准备工作

首先下载样本数据：`plug_on_base.bag` (750MB)，然后创建一个 `scratch` 功能包用于执行例子代码。这里创建一个名为 `learning_image_geometry` 的 `sandbox` 功能包，并增加下列依赖项：

```
roscreate-pkg learning_image_geometry image_geometry  
image_transport opencv2 cv_bridge tf image_view
```

另外需要确认 `learning_image_geometry` 文件夹包含在 `ROS_PACKAGE_PATH` 中。

## 2. 处理节点

这里创建一个节点用于订阅摄像机图像流和 `tf` 信息，并在输出的图像上画出需要的坐标系，并将注释过的图像发布到输出主题上。改变当前目录到刚创建的功能包：`roscore learning_image_geometry`。

创建新文件 `draw_frames.cpp` 并添加如下代码：

```
1 #include <ros/ros.h>
2 #include <image_transport/image_transport.h>
3 #include <opencv/cv.h>
4 #include <cv_bridge/CvBridge.h>
5 #include <image_geometry/pinhole_camera_model.h>
6 #include <tf/transform_listener.h>
7 #include <boost/foreach.hpp>
8
9 class FrameDrawer
10 {
11     ros::NodeHandle nh_;
12     image_transport::ImageTransport it_;
13     image_transport::CameraSubscriber sub_;
14     image_transport::Publisher pub_;
15     tf::TransformListener tf_listener_;
16     sensor_msgs::CvBridge bridge_;
17     image_geometry::PinholeCameraModel cam_model_;
18     std::vector<std::string> frame_ids_;
19     CvFont font_;
20
21 public:
22     FrameDrawer(const std::vector<std::string>& frame_ids) :
23         it_(nh_), frame_ids_(frame_ids)
24     {
25         std::string image_topic = nh_.resolveName("image");
26         sub_ = it_.subscribeCamera(image_topic, 1, &FrameDrawer
27             ::imageCb, this);
27         pub_ = it_.advertise("image_out", 1);
28         cvInitFont(&font_, CV_FONT_HERSHEY_SIMPLEX, 0.5, 0.5);
29     }
```

```
29
30     void imageCb(const sensor_msgs::ImageConstPtr & image_msg,
31                   const sensor_msgs::CameraInfoConstPtr & info_msg)
32 {
33     IplImage* image = NULL;
34     try {
35         image = bridge_.imgMsgToCv(image_msg, "bgr8");
36     }
37     catch (sensor_msgs::CvBridgeException& ex) {
38         ROS_ERROR("[draw_frames] Failed to convert image");
39         return;
40     }
41     cam_model_.fromCameraInfo(info_msg);
42
43     BOOST_FOREACH(const std::string& frame_id, frame_ids_) {
44         tf::StampedTransform transform;
45         try {
46             ros::Time acquisition_time=info_msg->header.stamp;
47             ros::Duration timeout(1.0/30);
48             tf_listener_.waitForTransform(cam_model_.tfFrame(),
49                                           frame_id, acquisition_time, timeout);
50             tf_listener_.lookupTransform(cam_model_.tfFrame(),
51                                           frame_id, acquisition_time, transform);
52         }
53         catch (tf::TransformException& ex) {
54             ROS_WARN("[draw_frames] TF exception:\n%s", ex.what());
55             return;
56         }
57         tf::Point pt = transform.getOrigin();
58         cv::Point3d pt_cv(pt.x(), pt.y(), pt.z());
59         cv::Point2d uv;
60         cam_model_.project3dToPixel(pt_cv, uv);
61
62         static const int RADIUS = 3;
63         cvCircle(image, uv, RADIUS, CV_RGB(255,0,0), -1);
64         CvSize text_size;
```

```
64     int baseline;
65     cvGetTextSize(frame_id.c_str(), &font_, &text_size, &
66                   baseline);
67     CvPoint origin = cvPoint(uv.x - text_size.width / 2,
68                               uv.y - RADIUS - baseline - 3);
69     cvPutText(image, frame_id.c_str(), origin, &font_,
70               CV_RGB(255,0,0));
71 }
72 }
73
74 int main(int argc, char** argv)
75 {
76     ros::init(argc, argv, "draw_frames");
77     std::vector<std::string> frame_ids(argv + 1, argv + argc);
78     FrameDrawer drawer(frame_ids);
79     ros::spin();
80 }
```

### 代码解释：

(1) 代码第 1~7 行。这几行代码主要是添加头文件。`image_geometry/pinhole_camera_model.h` 定义了类 `PinholeCameraModel`，其作用在于投影 3D 点到图像坐标。

(2) 代码第 74~80 行。这几行代码是主函数。`ros::init()` 移除了任何 ROS 命令行重映射参量。重映射参量是用户希望画出的坐标系的 ID，这里将它拷贝为一个向量。`FrameDrawer` 是这个例子中主要的类。

(3) 代码第 9~15 行。这些成员主要用于通信。

(4) 代码第 16、17 行。这两个对象是从 `sensor_msgs/Image` 和 `sensor_msgs/CameraInfo` 到 OpenCV 的桥梁。在图像回调中会用到它。

(5) 代码第 18、19 行。这里给出坐标系的列表用于绘图，一个 `CvFont` 对象用于渲染文本。

(6) 代码第 21、22 行。构造函数。节点句柄 `nh_` 首先被构建，并且初始化节点。然后传递它到 `ImageTransport` 构建函数。`tf_listener_` 自动连接到 `/tf` 主题。

(7) 代码第 23~25 行。用户订阅合成的 `Image` 与 `CameraInfo` 流。`image_transport::CameraSubscriber` 订阅图像主题和其成员 `camera_info` 主题。

(8) 代码第 26~28 行。广播添加了注释的图像主题。然后初始化字体对象。

(9) 代码第 30 行。现在可以获得图像回调。当从一个标定过的摄像机处理数据，同时需要图像和包含了标定参数的 `CameraInfo` 消息。

(10) 代码第 31~39 行。转换 `Image` 消息为 OpenCV `IplImage`。

(11) 代码第 41 行。利用从 `CameraInfo` 消息获得的信息作为 `PinholeCameraModel` 参数。`cam_model_` 是一个成员变量而不是局部变量。通常情况下，摄像机参数不会改变，更新前面摄像机模型参数实现起来代价很低。

(12) 代码第 43~54 行。对每一个需要的坐标系 ID，查询 `tf` 该坐标系在图像中的位置。在这个例子中，假设摄像机帧率为 30，因此在获取变换前最多等待  $1/30\text{s}$ 。

(13) 代码第 56~59 行。提取图像中需要的坐标系，将其存储在 `cv::Point3d`，然后使用 `image_geometry::PinholeCameraModel` 将点投影到图像坐标。这里假设收听一个没有畸变的图像主题。从摄像机驱动中输出的原始图像是有畸变的，使得从 3D 世界坐标系到 2D 图像坐标系的投影不精确。标定的主要目的之一就是修正畸变。`image_geometry` 中包含了用于移除畸变的方法，但是在 OpenCV 中，这个功能由 `image_proc` 或 `stereo_image_proc` 来实现。

(14) 代码第 61~68 行。最后，在图像上划出红色圆圈，并将坐标系 ID 印于其上。

(15) 代码第 70 行。将 OpenCV `IplImage` 转换为 ROS `Image` 并发布它。

### 3. 启动文件

使用 PR2 启动文件，以便使用离线数据，而不是使用用户自己的机器人。

### 4. 编译节点

打开 `CMakeLists.txt` 文件然后在文件最后追加：

```
rosbuild_add_executable(draw_frames draw_frames.cpp)
```

然后编译功能包：

```
rosmake learning_image_geometry
```

## 5. 运行

首先确认 `roscore` 在运行，然后启动相应的处理和可视化节点：

```
roslaunch draw.launch
```

然后记录和回放数据：

```
rosbag play plug_on_base.bag --clock
```

其中 `--clock` 选项告诉 `rosbag` 在 `/clock` 主题上发布模拟时间。现在用户可以在 `/image_out` 窗口看到添加注释后的图像。

## 4.5 演示例子

### 4.5.1 使用颜色追踪物体

本小节的例子中，实现了采用颜色追踪实现让物体指引机器人运行的功能。其中主要采用 OpenCV 中的 `cam_shift` 算法。这里使用鼠标选定追踪的物体，用矩形框标记，将该目标物体图进行颜色分割，反向投影，计算直方图并保存，然后根据 `cam_shift` 算法对视频流进行计算。回调函数如下：

```
1 void imageCallback(const sensor_msgs::ImageConstPtr &
2   msg_ptr)
3 {
4     try
5     {
6         frame = bridge_.imgMsgToCv(msg_ptr, "bgr8");
7     }
8     catch (sensor_msgs::CvBridgeException error)
9     {
10        ROS_ERROR("error");
11    }
12    if (!image)
13    {
14        image = cvCreateImage(cvGetSize(frame), 8, 3);
15        image->origin = frame->origin;
16        mask = cvCreateImage(cvGetSize(frame),
17                            IPL_DEPTH_8U, 1);
18        backproject = cvCreateImage(cvGetSize(frame),
19                                IPL_DEPTH_8U, 1);
```



```
        epsilon)/(cvQueryHistValue_3D(bg_hist,
        m,n,o)+epsilon));
49    }
50    track_window = selection;
51    track_object = 1;
52}
53 cvCalcBackProject(planes, backproject, lglikrat);
54
55 cvCamShift(backproject, track_window,
56             cvTermCriteria(CV_TERMCRIT_EPS +
57                             CV_TERMCRIT_ITER, 10, 1),
58             &track_comp, &track_box);
59 track_window = track_comp.rect;
60
61 if(backproject_mode)
62 cvCvtColor(backproject, image, CV_GRAY2BGR);
63 if(!image->origin)
64     track_box.angle = -track_box.angle;
65     cvEllipseBox(image, track_box, CV_RGB(255,0,0), 3,
66                 CV_AA, 0);
67
68 if(select_object && selection.width > 0 && selection.
69     height > 0)
70 {
71     cvSetImageROI(image, selection);
72     cvXorS(image, cvScalarAll(255), image, 0);
73     cvResetImageROI(image);
74
75     cvShowImage("Demo", image);
76     cvShowImage("Extra", backproject);
77     cvWaitKey(10);
78
79 }
```

追踪物体例子的运行结果如图 4.4 所示。

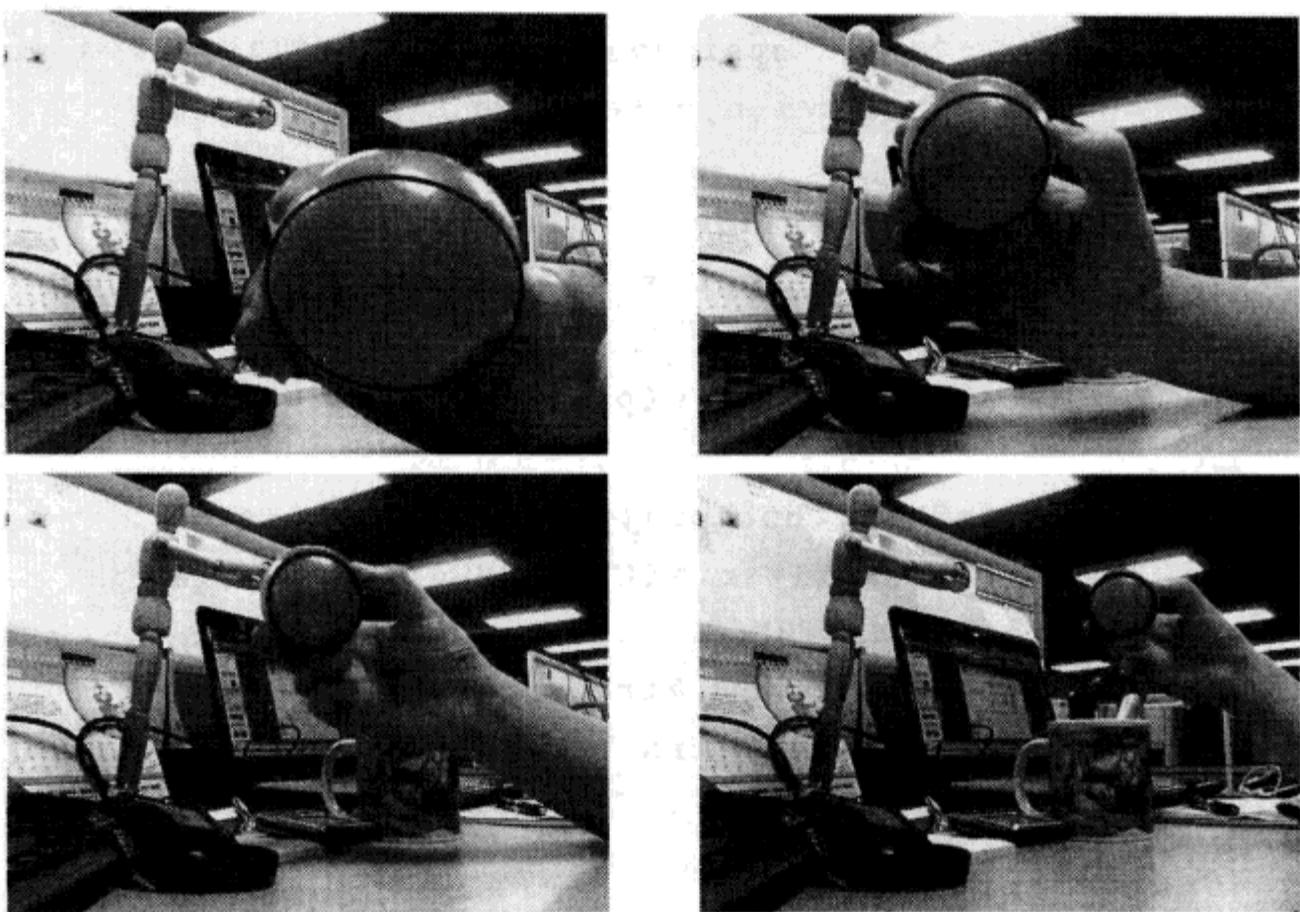


图 4.4 ROS OpenCV 追踪物体例子

#### 4.5.2 识别物体

在物体识别例子中，采用一个备选框，根据选定目标物体从实验场景中取出该物体。主函数如下：

```
1 int main(int argc, char **argv)
2 {
3     ros::init(argc, argv, "surf", ros::init_options::
4             AnonymousName);
5
6     CvCapture* capture=cvCaptureFromCAM(0);
7     IplImage* temps_image=NULL;
8     int *d=new int;
9     *d =0;
10    IplImage* mask_image=NULL;
11    int count=0;
12
13    while(1)
14    {
15        temps_image=cvQueryFrame(capture);
16        count++;
```

```
17         cvWaitKey(100);
18     }
19
20     ros::spin();
21
22     return 0;
23 }
```

识别物体例子的运行结果如图 4.5 所示。



图 4.5 ROS OpenCV 识别物体例子

## 第五章 SLAM 和导航

SLAM(simultaneous localization and mapping) 和导航是移动机器人最关键的部分。ROS 提供了完整的 SLAM 和导航功能包集，只需对机器人进行适当配置，即可在自己的机器人上实现完整的功能。要配置 ROS 下的 SLAM 和导航的功能包集，需要进行三个方面的配置：使用 `tf` 坐标变换软件包来配置机器人；通过 ROS 发布里程计的信息；通过 ROS 发布传感器数据流。

### 5.1 使用 `tf` 配置机器人

#### 1. 变换配置

很多 ROS 功能包需要使用 `tf` 坐标变换软件包来发布机器人的坐标变换树。抽象一点讲，变换树定义了不同坐标系之间的偏移。例如，一个简单的机器人，它有移动的基座和位于基座上方的激光传感器。在这台机器人上，可以定义两个坐标系：一个坐标系原点位于机器人基座中心，另一个坐标系原点位于激光传感器中心。将位于基座上的坐标系定义为 `base_link` (这对于导航来说很重要，因为需要把它放在机器人的旋转中心)，称位于激光传感器的坐标系为 `base_laser`。

现在开始，假设用户有若干数据，这些数据是基于激光传感器中心来表示的，即这些数据是 `base_laser` 坐标系下的。这些数据用来帮助机器人实现避障的功能。为了实现该功能，需要有从 `base_laser` 到 `base_link` 的坐标变换，本质上讲，这实际上定义了两个坐标系之间的关系。

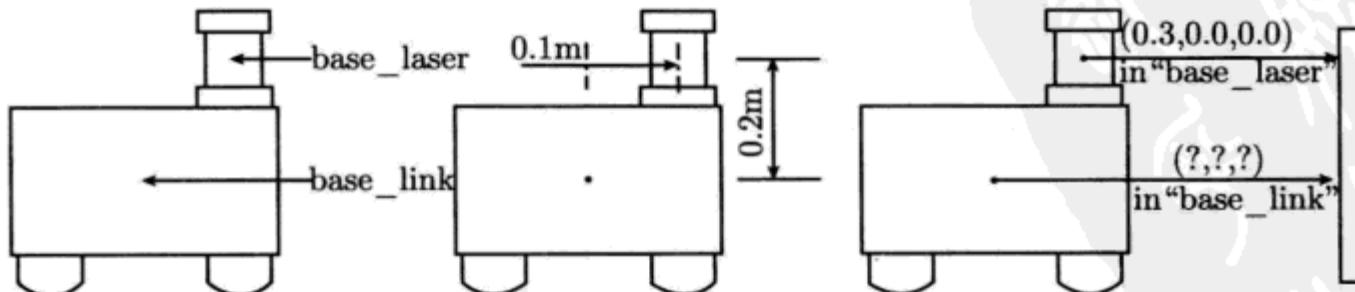


图 5.1 简单的机器人平台模型

如图 5.1 所示，已知激光传感器位于移动基座中心点前方 0.1m 且上方 0.2m 的位置。可以获取从 `base_link` 到 `base_laser` 的变换关系，`base_link`

坐标系必须平移 ( $x: 0.1m$ ,  $y: 0.0m$ ,  $z: 0.2m$ )。相应地, 从 `base_laser` 到 `base_link` 的反向平移为 ( $x: -0.1m$ ,  $y: 0.0m$ ,  $z: -0.2m$ )。

用户也可以自行建立这些变换关系, 但是当变换的数量增加时, 这变得困难起来。`tf` 软件包可以帮助用户完成这些变换的工作。为了使用 `tf` 定义和存储这些变换关系, 需要把它们添加到坐标变换树中。从概念上讲, 坐标变换树中的每一个节点对应于一个坐标系, 每条分支对应一个从当前节点到其子节点的变换。`tf` 软件包使用树形结构, 来保证连接两个坐标之间的变换是单向流, 并且树的分支是从父节点到子节点的有向边。

为了在当前例子中使用 `tf` 软件包进行坐标变换, 需要创建 2 个节点, 分别对应于 `base_link` 和 `base_laser` 坐标系。为了建立一条边 (坐标变换树的分支), 首先需要确定哪一个是父节点, 哪一个是子节点。需要记住的是, `tf` 假设所有的变换都是从父节点到子节点。在这个例子中, 假设 `base_link` 为父节点, 因为其他传感器都是相对于基座添加上来的。因此, `base_link` 和 `base_laser` 之间的变换矩阵是 ( $x: 0.1m$ ,  $y: 0.0m$ ,  $z: 0.2m$ ), 如图 5.2 所示。通过变换, 从 `base_laser` 接收激光传感器扫描数据, 然后转换到 `base_link` 坐标系下, 就变成很简单地调用 `tf` 库, 机器人可使用这些信息避障。

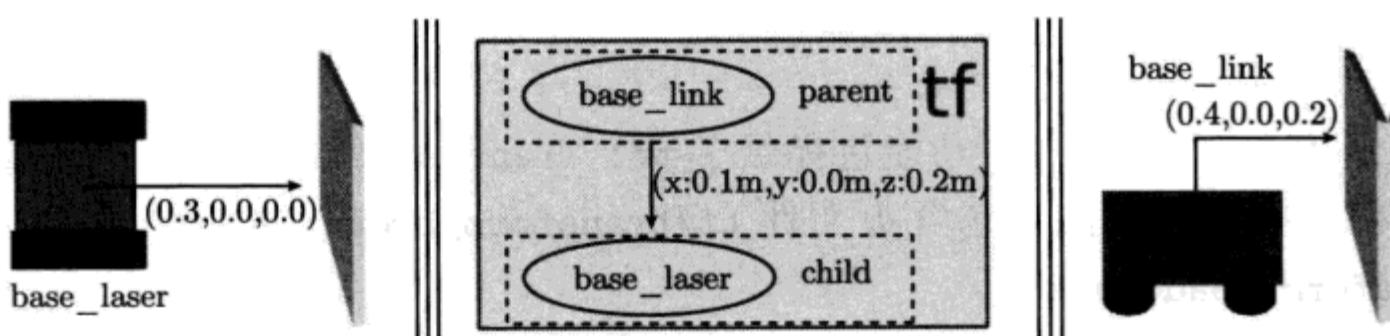


图 5.2 简单的机器人平台 `tf` 模型

## 2. 编写代码

针对上述任务, 从 `base_laser` 取点变换到 `base_link` 坐标系下, 首先需要做的是创建用于发布变换信息的节点, 然后创建收听变换信息的节点, 并应用该变换信息进行坐标变换:

```
roscreate-pkg robot_setup_tf roscpp tf geometry_msgs
```

## 3. 广播“变换”

下面创建能够通过 ROS 广播从 `base_laser` 到 `base_link` 坐标变换的节点。在刚刚创建的 `robot_setup_tf` 功能包中创建文件 `src/tf_broadcaster.cpp`, 并添加如下代码:

```

1 #include <ros/ros.h>
2 #include <tf/transform_broadcaster.h>
3
4 int main(int argc, char** argv){
5     ros::init(argc, argv, "robot_tf_publisher");
6     ros::NodeHandle n;
7
8     ros::Rate r(100);
9
10    tf::TransformBroadcaster broadcaster;
11
12    while(n.ok()){
13        broadcaster.sendTransform(tf::StampedTransform(tf::
14            Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3
15            (0.1, 0.0, 0.2)), ros::Time::now(), "base_link",
16            "base_laser")));
17        r.sleep();
18    }
19}

```

代码解释：

(1) 代码第 2 行。tf 功能包提供了 `tf::TransformBroadcaster`，使得发布变换变得容易。需要包含头文件 `tf/transform_broadcaster.h` 才能使用 `TransformBroadcaster`。

(2) 代码第 10 行。创建 `TransformBroadcaster` 对象，用于发送从 `base_link` 到 `base_laser` 的变换。

(3) 代码第 13 行。这是实现变换的部分。发送带有 `TransformBroadcaster` 的变换需要 4 个参量。首先传送旋转矩阵：它通过 `btQuaternion` 指定了两个坐标系之间的旋转关系。这个例子中，没有旋转，因此前面三个参量值为 0。随后是一个平移变量 `btVector3`。第三步，给将要发布的变换指定时间戳：`ros::Time::now()`。第四步，传送父节点名称：`base_link`。第五步，传送子节点名称：`base_laser`。

#### 4. 使用“变换”

下面的例子用于创建一个可以实现变换的节点。在 `robot_setup_tf` 功能包内创建文件 `src/tf_listener.cpp`，并添加如下代码：

```
1 #include <ros/ros.h>
2 #include <geometry_msgs/PointStamped.h>
3 #include <tf/transform_listener.h>
4
5 void transformPoint(const tf::TransformListener& listener){
6     geometry_msgs::PointStamped laser_point;
7     laser_point.header.frame_id = "base_laser";
8
9     laser_point.header.stamp = ros::Time();
10
11    laser_point.point.x = 1.0;
12    laser_point.point.y = 0.2;
13    laser_point.point.z = 0.0;
14
15    try{
16        geometry_msgs::PointStamped base_point;
17        listener.transformPoint("base_link",laser_point,
18                               base_point);
19
20        ROS_INFO("base_laser: (%.2f, %.2f, %.2f) --> base_link:
21                  (%.2f, %.2f, %.2f) at time %.2f",laser_point.point
22                  .x,laser_point.point.y,laser_point.point.z,
23                  base_point.point.x,base_point.point.y,base_point.
24                  point.z,base_point.header.stamp.toSec());
25    }
26
27    catch(tf::TransformException& ex){
28        ROS_ERROR("Received an exception trying to transform a
29                  point from \"base_laser\" to \"base_link\": %s", ex.
30                  what());
31    }
32 }
```

```

32     ros::Timer timer = n.createTimer(ros::Duration(1.0),
33                                     boost::bind(&transformPoint, boost::ref(listener)));
34
35 }

```

## 5. 编译代码

打开由 `roscreate-pkg` 自动生成的 `CMakeLists.txt` 文件，在文件尾添加：

```

rosbuild_add_executable(tf_broadcaster src/tf_broadcaster.cpp)
rosbuild_add_executable(tf_listener src/tf_listener.cpp)

```

保存文件，然后在 `robot_setup_tf` 功能包内，运行 `rosmake`。

## 6. 运行代码

(1) 在新终端运行：`roscore`。

(2) 在第二个新终端运行：`rosrun robot_setup_tf tf_broadcaster`。

(3) 在第三个新终端运行：`rosrun robot_setup_tf tf_listener`。

执行成功之后，显示信息如下：

```

[ INFO] 1248138528.200823000: base_laser: (1.00, 0.20. 0.00)
---> base_link: (1.10, 0.20, 0.20) at time 1248138528.19
[ INFO] 1248138529.200820000: base_laser: (1.00, 0.20. 0.00)
---> base_link: (1.10, 0.20, 0.20) at time 1248138529.19
[ INFO] 1248138530.200821000: base_laser: (1.00, 0.20. 0.00)
---> base_link: (1.10, 0.20, 0.20) at time 1248138530.19
[ INFO] 1248138531.200835000: base_laser: (1.00, 0.20. 0.00)
---> base_link: (1.10, 0.20, 0.20) at time 1248138531.19
[ INFO] 1248138532.200849000: base_laser: (1.00, 0.20. 0.00)
---> base_link: (1.10, 0.20, 0.20) at time 1248138532.19

```

## 5.2 通过 ROS 发布里程计信息

本小节介绍如何为导航功能包集发布里程计信息，包括通过 ROS 发布 `nav_msgs/Odometry` 信息，一个通过 `tf` 软件包发布从 `odom` 坐标系到 `base_link` 坐标系的坐标变换信息。

### 1. 通过 ROS 发布里程计信息

导航功能包集用 `tf` 软件包来确定机器人在世界坐标系中的位置和相对于静态地图的相关传感器信息，但是 `tf` 软件包不提供与机器人速度相关的任何信息，所以导航功能包集要求里程计源程序发布一个变换和一个包含速度信息的 `nav_msgs/Odometry` 消息。下面将介绍 `nav_msgs/Odometry` 消息和提供发布变换和 `tf` 消息的代码例子。

### 2. `nav_msgs/Odometry` 消息

```
# This represents an estimate of a position and velocity in
# free space.
# The pose in this message should be specified in the
# coordinate frame given by header.frame_id.
# The twist in this message should be specified in the
# coordinate frame given by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

`nav_msgs/Odometry` 消息存储机器人在自由空间位置和速度的估计。在这个消息中，`pose` 与机器人相对于里程计坐标系的位置估计与位置估计的协方差相关，`twist` 与在子坐标系里机器人的速度相关，通常是移动平台的坐标系连同速度估计的协方差。

### 3. 使用 `tf` 发布里程计变换

正如在 5.1 节中讨论过的，`tf` 软件包的作用在于管理机器人相关的坐标系之间的关系。里程计的程序必须发布它的坐标系信息。下面的代码是发布 `nav_msgs/Odometry` 消息的例子。

```
1 #include <ros/ros.h>
2 #include <tf/transform_broadcaster.h>
3 #include <nav_msgs/Odometry.h>
4
5 int main(int argc, char** argv){
6     ros::init(argc, argv, "odometry_publisher");
7 }
```

```
8     ros::NodeHandle n;
9     ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom", 50);
10    tf::TransformBroadcaster odom_broadcaster;
11
12    double x = 0.0;
13    double y = 0.0;
14    double th = 0.0;
15
16    double vx = 0.1;
17    double vy = -0.1;
18    double vth = 0.1;
19
20    ros::Time current_time, last_time;
21    current_time = ros::Time::now();
22    last_time = ros::Time::now();
23
24    ros::Rate r(1.0);
25    while(n.ok()){
26        current_time = ros::Time::now();
27
28        double dt = (current_time - last_time).toSec();
29        double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
30        double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
31        double delta_th = vth * dt;
32
33        x += delta_x;
34        y += delta_y;
35        th += delta_th;
36
37        geometry_msgs::Quaternion odom_quat = tf::
38            createQuaternionMsgFromYaw(th);
39
40        geometry_msgs::TransformStamped odom_trans;
41        odom_trans.header.stamp = current_time;
42        odom_trans.header.frame_id = "odom";
43        odom_trans.child_frame_id = "base_link";
44        odom_trans.transform.translation.x = x;
```

```
45     odom_trans.transform.translation.y = y;
46     odom_trans.transform.translation.z = 0.0;
47     odom_trans.transform.rotation = odom_quat;
48
49     odom_broadcaster.sendTransform(odom_trans);
50
51     nav_msgs::Odometry odom;
52     odom.header.stamp = current_time;
53     odom.header.frame_id = "odom";
54
55     odom.pose.pose.position.x = x;
56     odom.pose.pose.position.y = y;
57     odom.pose.pose.position.z = 0.0;
58     odom.pose.pose.orientation = odom_quat;
59
60     odom.child_frame_id = "base_link";
61     odom.twist.twist.linear.x = vx;
62     odom.twist.twist.linear.y = vy;
63     odom.twist.twist.angular.z = vth;
64
65     odom_pub.publish(odom);
66
67     last_time = current_time;
68     r.sleep();
69 }
70 }
```

代码详细解释：

- (1) 代码第 2~3 行。因为要发布一个从 `odom` 坐标系到 `base_link` 坐标系的变换和一个 `nav_msgs/Odometry` 消息，所以要添加相关头文件。
- (2) 代码第 9、10 行。创建一个 `ros::Publisher` 和 `tf::TransformBroadcaster` 对象来用 ROS 和 `tf` 分别发送消息。
- (3) 代码第 12~14 行。这里假定机器人初始位置在里程计坐标系的原点上。
- (4) 代码第 16~18 行。这里设置一个速度，该速度会使 `base_link` 坐标系和 `odom` 坐标系在  $x$  方向以 0.1m/s 的速度、在  $y$  方向以 -0.1m/s 的速度、在  $\theta$  方向以 0.1rad/s 的角速度移动。这会使模拟器中机器人运动起来。
- (5) 代码第 24 行。在下面的例子中，为了方便以 1Hz 的频率发布里程计信

息，实际中多数系统会用更高的频率来发布里程计信息。

(6) 代码第 29~35 行。这里基于设定好的固定速度更新里程计信息，实际的里程计系统会使用计算出来的速度而不是设定好的固定速度。

(7) 代码第 37 行。通常所有的消息都用 3D 形式创建以允许 2D 和 3D 形式同时存在并能同时工作，这是为了保持消息的种类数尽量少。所以就需要把里程计的导航角转换成一个四元数再发布出去。`tf` 软件包提供函数支持从导航角到四元数和从四元数到导航角的简单转换。

(8) 代码第 40~42 行。这里创建一个 `TransformedStamped` 消息通过 `tf` 发送出去，我们需要实时发布 `odom` 坐标系和 `base_link` 坐标系的转换信息。因此设置消息头和对应的坐标系，需要确定的是，`odom` 是父坐标系，`base_link` 是子坐标系。

(9) 代码第 44~49 行。填写里程计信息，然后用 `Transform Broadcaster` 将转换信息发布出去。

(10) 代码第 52、53 行。发布 `nav_msgs/Odometry` 消息来让导航功能包集能获得速度信息。把系统时间和 `odom` 坐标系设置为消息头。

(11) 代码第 55~65 行。完成里程计信息后发布出去。设置消息 `child_frame_id` 为 `base_link` 坐标系，因为这是输入速度所在的坐标系。

### 5.3 通过 ROS 发布传感器数据流

本节将说明通过 ROS 发布两种传感器信息：`sensor_msgs/LaserScan` 和 `sensor_msgs/PointCloud`。通过 ROS 发布传感器的信息对导航功能包集来说是非常重要的。如果导航功能包集不能从机器人传感器上接受信息，机器人可能会撞到东西。可以给导航功能包集提供信息的传感器有激光、摄像头、声纳、红外、碰撞传感器等。导航功能包集只接收 `sensor_msgs/LaserScan` 或 `sensor_msgs/PointCloud` 形式的传感器数据。下面具体介绍两种信息的典型设置和应用。

#### 1. ROS 消息头

与许多通过 ROS 发布的消息一样，`sensor_msgs/LaserScan` 和 `sensor_msgs/PointCloud` 消息类型包含 `tf` 的坐标系和时间相关信息，为了使这些发送的信息标准化，这些消息中的消息头都是必不可少的一部分。

下面列出三种类型的消息头。当消息从给定发布者发出时，`seq` 域自动增加

消息的识别符。`stamp` 域存储与消息中的数据有关的时间信息。例如，在激光扫描仪中，`stamp` 对应于扫描开始的时间。`frame_id` 域存储了与消息中的数据有关的 `tf` 坐标系信息。

```
#Standard metadata for higher-level flow data types
#sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.secs: seconds (stamp_secs) since epoch
# * stamp.nsecs: nanoseconds since stamp_secs
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
# 0: no frame
# 1: global frame
string frame_id
```

## 2. LaserScans 消息的发布

### 1) LaserScans 消息

对于带有激光扫描仪的机器人来说，ROS 在功能包 `sensor_msgs` 中提供了名为 `LaserScans` 的特殊消息类型。`LaserScans` 消息使得使用虚拟的激光传感器变得容易起来，只要传感器输出的数据符合 `LaserScans` 消息的格式。具体的消息参数如下：

```
Header header
float32 angle_min          # start angle of the scan [rad]
float32 angle_max          # end angle of the scan [rad]
float32 angle_increment    # angular distance between
                           measurements [rad]
float32 time_increment     # time between measurements [seconds]
float32 scan_time          # time between scans [seconds]
float32 range_min          # minimum range value [m]
float32 range_max          # maximum range value [m]
float32[] ranges           # range data [m] (Note: values
                           < range_min or > range_max
                           should be discarded)
```

```
float32[] intensities    # intensity data [device-specific  
                                units]
```

## 2) LaserScans 消息发布

下面是一段发布 LaserScan 消息的代码：

```
1 #include <ros/ros.h>  
2 #include <sensor_msgs/LaserScan.h>  
3  
4 int main(int argc, char** argv){  
5     ros::init(argc, argv, "laser_scan_publisher");  
6  
7     ros::NodeHandle n;  
8     ros::Publisher scan_pub = n.advertise<sensor_msgs::  
                                LaserScan>("scan", 50);  
9  
10    unsigned int num_readings = 100;  
11    double laser_frequency = 40;  
12    double ranges[num_readings];  
13    double intensities[num_readings];  
14  
15    int count = 0;  
16    ros::Rate r(1.0);  
17    while(n.ok()){  
18        for(unsigned int i = 0; i < num_readings; ++i){  
19            ranges[i] = count;  
20            intensities[i] = 100 + count;  
21        }  
22        ros::Time scan_time = ros::Time::now();  
23  
24        sensor_msgs::LaserScan scan;  
25        scan.header.stamp = scan_time;  
26        scan.header.frame_id = "laser_frame";  
27        scan.angle_min = -1.57;  
28        scan.angle_max = 1.57;  
29        scan.angle_increment = 3.14 / num_readings;  
30        scan.time_increment = (1 / laser_frequency) /  
                                (num_readings);  
31        scan.range_min = 0.0;  
32        scan.range_max = 100.0;
```

```
33
34     scan.set_ranges_size(num_readings);
35     scan.set_intensities_size(num_readings);
36     for(unsigned int i = 0; i < num_readings; ++i){
37         scan.ranges[i] = ranges[i];
38         scan.intensities[i] = intensities[i];
39     }
40
41     scan_pub.publish(scan);
42     ++count;
43     r.sleep();
44 }
45 }
```

代码解释：

- (1) 代码第 2 行。这里表示包含 `sensor_msgs/LaserScan` 消息类型。
- (2) 代码第 8 行。创建 `ros::Publisher` 用于后面发送消息。
- (3) 代码第 10~13 行。设置存储空间存储扫描数据。实际上，数据从传感器发出。
- (4) 代码第 15~22 行。编写虚拟的激光数据，每秒钟发布一次消息。
- (5) 代码第 24~39 行。创建 `scan_msgs::LaserScan` 消息并填入数据。

### 3. 通过 ROS 发布 PointClouds

#### 1) PointClouds 消息

ROS 提供了 `sensor_msgs/PointCloud` 消息类型来存储和共享一些以点云形式存在的数据。这个消息支持三维点阵列。

```
Header header
geometry_msgs/Point32[] points
#Array of 3d points
ChannelFloat32[] channels
#Each channel should have the same number of elements as
points array, and the data in each channel should
correspond 1:1 with each point
```

#### 2) PointClouds 消息发布

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

```
2 #include <sensor_msgs/PointCloud.h>
3
4 int main(int argc, char** argv){
5     ros::init(argc, argv, "point_cloud_publisher");
6
7     ros::NodeHandle n;
8     ros::Publisher cloud_pub = n.advertise<sensor_msgs::
9         PointCloud>("cloud", 50);
10
11    unsigned int num_points = 100;
12
13    int count = 0;
14    ros::Rate r(1.0);
15    while(n.ok()){
16        sensor_msgs::PointCloud cloud;
17        cloud.header.stamp = ros::Time::now();
18        cloud.header.frame_id = "sensor_frame";
19
20        cloud.set_points_size(num_points);
21
22        cloud.set_channels_size(1);
23        cloud.channels[0].name = "intensities";
24        cloud.channels[0].set_values_size (num_points);
25
26        for(unsigned int i = 0; i < num_points; ++i){
27            cloud.points[i].x = 1 + count;
28            cloud.points[i].y = 2 + count;
29            cloud.points[i].z = 3 + count;
30            cloud.channels[0].values[i] = 100 + count;
31        }
32
33        cloud_pub.publish(cloud);
34        ++count;
35        r.sleep();
36    }
```

代码解释：

(1) 代码第 2 行。这里表示包含 sensor\_msgs/PointCloud 消息类型。

- (2) 代码第 8 行。创建 `ros::Publisher` 用于后面发送消息。
- (3) 代码第 15~17 行。填充消息头，这里的时间戳和坐标系信息会随着消息一起发出。
- (4) 代码第 19 行。设置点云的点数，并使用虚拟数据进行填充。
- (5) 代码第 21~23 行。增加 `intensity` 频道到 `sensor_msgs/PointCloud` 消息。
- (6) 代码第 25~30 行。使用虚拟数据填充 `sensor_msgs/PointCloud` 消息。
- (7) 代码第 32 行。发布 `PointCloud` 消息。

## 5.4 SLAM

### 5.4.1 SLAM 简介

SLAM 也称为 CML(*concurrent mapping and localization*)，即时定位与地图构建，或并发建图与定位。SLAM 最早由 Smith、Self 和 Cheeseman 于 1988 年提出。由于其重要的理论与应用价值，很多学者认为它是实现真正全自主移动机器人的关键。SLAM 问题可以描述为：机器人在未知环境中从一个未知位置开始移动，在移动过程中根据位置估计和地图进行自身定位，同时在自身定位的基础上建造增量式地图，实现机器人的自主定位和导航。

### 5.4.2 slam\_gmapping 功能包

#### 1. 概述

本功能包包含了来自 OpenSlam<sup>[17]</sup> 的 ROS 软件包 GMapping。GMapping 功能包提供了基于激光的 SLAM，建立了名为 `slam_gmapping` 的节点。使用 `slam_gmapping`，用户可以根据从激光传感器输出的数据创建 2D 栅格地图。这个功能包用网站 openslam.org 上的 GMapping 中的 r39，并且加以改进以支持新版本的 GCC 和 OSX。

#### 2. 外部文档

这部分主要描述所使用的第三方软件功能包。GMapping 是一个从激光传感器数据建立地图的非常有效的 Rao-Blackwellized 粒子滤波软件包。GMapping 的文档可以查看：<https://svn.openslam.org/data/svn/gmapping>。

近年来，Rao-Blackwellized 粒子滤波已经被证实为是求解 SLAM 问题的有效

工具。该方法使用粒子滤波方法，其中每个粒子承载了一个独立的环境地图。其中，关键的问题在于怎样降低粒子数量。在该软件包中，采用自适应技术来降低粒子数量。其中采用的方法为不光考虑移动而且考虑最近的观测值来计算精确的粒子分布。这样，在滤波的预测步，极大降低了机器人位姿的不确定性。更进一步，算法中使用有选择地重采样技术，这样有效降低了粒子耗散对算法的影响。

### 3. 硬件需求

为了使用 `slam_gmapping`，用户首先需要一个机器人平台，该平台可以提供里程计数据并且装备了水平安装的固定的激光测距仪。`slam_gmapping` 节点将把收到的数据进行 `tf` 变换。

### 4. 例子

为了让机器人建立地图，需要在 `base_scan` 主题发布扫描数据：

```
rosrun gmapping slam_gmapping scan:=base_scan
```

### 5. 节点

#### 1) `slam_gmapping`。

`slam_gmapping` 节点在 `sensor_msgs/LaserScan` 消息内获取数据并建立地图 (`nav_msgs/OccupancyGrid`)。该地图可以通过 ROS 主题或服务来获取。

#### 2) 订阅的主题

(1) `tf` (`tf/tfMessage`)：与坐标系相关的变换。

(2) `scan` (`sensor_msgs/LaserScan`)：激光扫描数据。

#### 3) 发布的主题

(1) `map_metadata` (`nav_msgs/MapMetaData`)：从这个周期性更新的主题获取地图数据。

(2) `map` (`nav_msgs/OccupancyGrid`)：从这个周期性更新的主题获取地图。

(3) `entropy` (`std_msgs/Float64`)：估计机器人位姿分布的熵。

#### 4) 服务

`dynamic_map` (`nav_msgs/GetMap`)：调用该服务以获取地图数据。

#### 5) 参数

(1) `inverted_laser` (`string, default: "false"`)。查看激光传感器是否左右翻转或者上下翻转，该参数在 1.1.1 版本中移除。

(2) `throttle_scans` (int, default: 1)。默认值 1 每次处理 1 个扫描数据，可以设置为更大的数以便跳过一些扫描数据。

(3) `base_frame` (string, default: "base\_link")。附属在移动机器人基座上的坐标系。

(4) `map_frame` (string, default: "map")。地图上的坐标系。

(5) `odom_frame` (string, default: "odom")。里程计上的坐标系。

(6) `map_update_interval` (float, default: 5.0)。更新地图的时间频率(默认为 5s)。

(7) `maxUrange` (float, default: 80.0)。激光最大可测距离。

(8) `sigma` (float, default: 0.05)。扫描匹配中的偏差。

(9) `kernelSize` (int, default: 1)。查找对应所用的内核大小。

(10) `lstep` (float, default: 0.05)。平移中的优化步长。

(11) `astep` (float, default: 0.05)。旋转中的优化步长。

(12) `iterations` (int, default: 5)。扫描匹配的迭代步数。

(13) `lsigma` (float, default: 0.075)。扫描匹配中只考虑激光的标准偏差。

(14) `ogain` (float, default: 3.0)。估计似然值时用于平滑重采样数据。

(15) `lskip` (int, default: 0)。在每次扫描中跳过的扫描数据。

(16) `srr` (float, default: 0.1)。平移时，作为平移的里程误差函数(rho/rho)。

(17) `srt` (float, default: 0.2)。平移时，作为旋转的里程误差函数(rho(theta))。

(18) `str` (float, default: 0.1)。旋转时，作为平移的里程误差函数(theta/rho)。

(19) `stt` (float, default: 0.2)。旋转时，作为旋转的里程误差函数(theta(theta))。

(20) `linearUpdate` (float, default: 1.0)。距离更新频率，每当机器人平移这么远时处理一次扫描。

(21) `angularUpdate` (float, default: 0.5)。每当机器人旋转这么远时处理一次扫描。

(22) `temporalUpdate (float, default: -1.0)`。如果最新扫描处理得比更新慢时处理一次扫描。负值时，将关闭基于时间的更新。

(23) `resampleThreshold (float, default: 0.5)`。基于重采样域值的 `Neff`。

(24) `particles (int, default: 30)`。滤波中的粒子数。

(25) `xmin (float, default: -100.0)`。初始的地图尺寸参数。

(26) `ymin (float, default: -100.0)`。初始的地图尺寸参数。

(27) `xmax (float, default: 100.0)`。初始的地图尺寸参数。

(28) `ymax (float, default: 100.0)`。初始的地图尺寸参数。

(29) `delta (float, default: 0.05)`。地图分辨率参数。

(30) `llsamplerange (float, default: 0.01)`。用于似然计算的平移重采样距离。

(31) `llsamplestep (float, default: 0.01)`。用于似然计算的平移重采样步长。

(32) `lasamplerange (float, default: 0.005)`。用于似然计算的角度重采样距离。

(33) `lasamplestep (float, default: 0.005)`。用于似然计算的角度重采样步长。

(34) `transform_publish_period (float, default: 0.05)`。变换发布之间的时间间隔。

(35) `occ_thresh (float, default: 0.25)`。栅格地图的栅格值。

(36) `maxRange (float)`。传感器极大距离范围。如果在传感器距离范围内没有障碍物，应该在地图上显示为自由空间。

## 6) 需要的 tf 变换

(1) <the frame attached to incoming scans> → `base_link`: 通常是一个固定值，通过 `robot_state_publisher` 或 `tf` 的 `static_transform_publisher` 进行周期性广播。

(2) `base_link` → `odom`: 通常由里程计系统提供。

## 7) 提供的 tf 变换

`map` → `odom`: 当前地图坐标系中机器人位姿的估计。

### 5.4.3 使用记录的数据建立地图

本小节介绍如何从激光扫描数据及其变换中创建 2D 地图。

#### 1. 建立地图

- (1) 如果已经安装源文件，首先编译 GMapping: `rosmake gmapping`。
- (2) 获取一个 bag：自己创建或者从网站下载测试包。
- (3) 运行 ROS: `roscore`。
- (4) 设置参数: `rosparam set use_sim_time true`。
- (5) 运行 GMapping: `rosrun gmapping slam_gmapping scan:=base_scan`。

在新终端中，回放消息记录包文件，反馈数据给 `slam_gmapping`:

```
rosbag play <name of the bag that you downloaded / created in  
step 2>
```

运行之后，需要等待程序执行完毕，退出。

最终，可以保存建立的地图为 pgm 文件: `rosrun map_server map_saver`。用户可以使用自己喜欢的图像浏览器查看生成的地图，如图 5.3 所示。

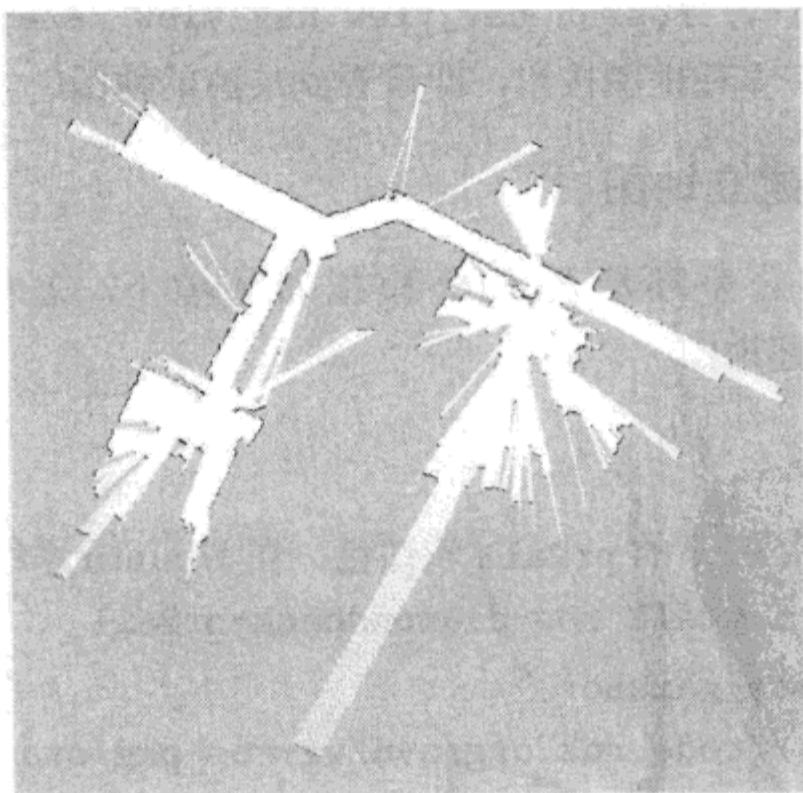


图 5.3 使用记录的数据建立地图

#### 2. 下载测试包

可以使用 `wget` 下载测试用的 bag:

```
wget http://pr.willowgarage.com/data/gmapping/basic_
localization_stage.bag
```

### 3. 建立个人程序包

(1) 搭建自己的机器人，装备激光传感器，发布变换数据，并可以遥控。

(2) 记录扫描与变换数据：

```
rosbag record -O mylaserdata /base_scan /tf
```

这将在当前目录写入一个名为 `mylaserdata_<DATE>-topic.bag` 文件。

(3) 驱动机器人移动。

(4) 结束 `rosbag` 实例，然后用户可以看到文件已经创建。

### 4. 实时查看建立地图的过程

如果用户不希望等到所有进程结束之后，才查看地图，而是需要实时查看地图，可以执行如下命令：

(1) 编译 `nav_view`: `rosmake nav_view`。

(2) 重复上述过程，直至有日志文件可以回放数据，然后运行 `slam_gmapping`。

(3) 运行 `nav_view`: `rosrun nav_view nav_view /static_map:=/dynamic_map`。随后用户单击重载地图按钮，即可看到最新的地图。

## 5.4.4 模拟器中建立地图

本节展示在办公室环境及 PR2 模拟器环境下，怎样使用 OpenSLAM GMapping 功能包创建地图。

### 1. 安装

首先需要安装 ROS 下的 `pr2all` 软件包。在 Ubuntu 系统下：

```
sudo apt-get install ros-diamondback-pr2all
```

还需要检查 `wg_robots_gazebo`:

```
svn co https://code.ros.org/svn/wg-ros-pkg/stacks/wg_robots_
gazebo/trunk/pr2_build_map_gazebo_demo/
```

### 2. 编译

添加上述 `pr2_build_map_gazebo_demo` 目录路径到用户 `ROS_PACKAGE_PATH` 环境变量，然后可以编译模拟插件：

```
rosmake pr2_build_map_gazebo_demo
```

### 3. 运行

(1) 建立地图演示例子。可以查看首选的启动文件是哪些：

```
roslaunch pr2_build_map_gazebo_demo pr2_build_map_gazebo_demo.launch
```

(2) `rviz` 界面如图 5.4 所示。

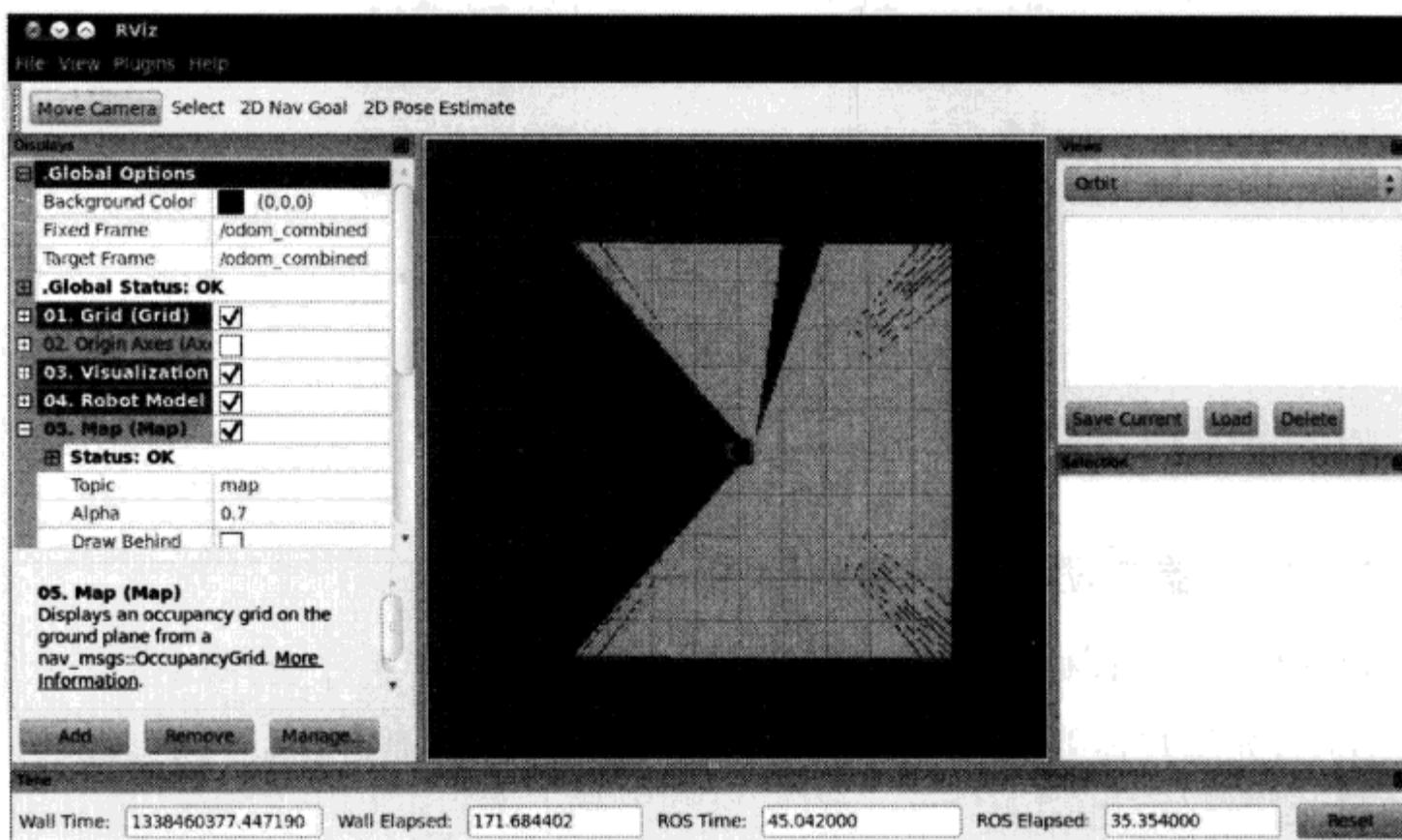


图 5.4 模拟器中建立地图

其中，`rviz` 窗口左侧面板中可以作如下设置：

① 左侧面板 **Global Options** 中固定坐标系 (**Fixed Frame**) 设置为 `/odom_combined`。

② 左侧面板 **Global Options** 中目标坐标系 (**Target Frame**) 设置为 `/odom_combined`。

在 `rviz` 中看到的只是地图的一部分，为了建立完整的地图，需要让机器人移动以收集传感器数据。

为了在模拟器中使用键盘控制让机器人移动，在新的终端中运行命令：

```
roslaunch pr2_teleop teleop_keyboard.launch
rosrun map_server map_saver -f map
```

上述第二条命令保存当前地图状态到文件 `map.pgm`，本例中的地图如图 5.5 所示。由于真实世界中各种复杂情况导致的数据不精确，用户在放大地图时会发现其中的数据并不精确。

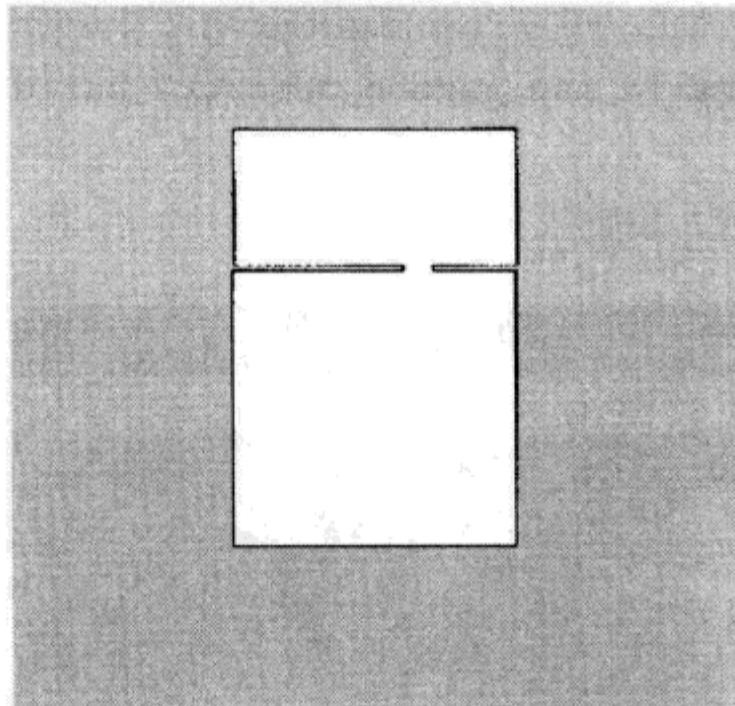


图 5.5 生成的地图

#### 5.4.5 模拟器中使用客户定制地图

本小节介绍如何使用创建的地图在 Gazebo 模拟器中进行 2D 导航。

- (1) 首先用户需要有一个地图文件 `map.pgm`，然后把它放在 `pr2_gazebo` 功能包下。
- (2) 编译 `2dnav-stack`: `rosmake pr2_2dnav_gazebo`。
- (3) 创建启动文件 `my_map_sim.launch`:

```
1 <launch>
2 <!-- start up world -->
3 <include file="$(find gazebo_worlds)/launch/office_world.
   launch"/>
4
5 <!-- start up robot -->
6 <include file="$(find pr2_gazebo)/pr2.launch"/>
7
8 <!-- load map -->
9 <node pkg="map_server" type="map_server" args="$(find
   pr2_gazebo)/map.pgm 0.05" respawn="true" name="map1" />
10 <!-- if you have a map.pgm and a map.yaml, use this instead
    -->
```

```
11 <!-- <node pkg="map_server" type="map_server" args="$(find  
      some_package)/map.yaml" respawn="true" name="map1" />  
    -->  
  
12  
13 <!-- Tuck Arms For Navigation -->  
14 <node pkg="pr2_tuckarm" type="tuck_arms.py" args="b" output=  
      "screen" name="tuck_arms"/>  
  
15  
16 <!-- nav-stack -->  
17 <include file="$(find pr2_2dnav_gazebo)/2dnav-stack-amcl.  
      launch"/>  
  
18  
19 <!-- for visualization -->  
20 <include file="$(find 2dnav_pr2)/rviz/rviz_move_base.launch  
      "/>  
  
21  
22 </launch>
```

#### (4) 启动程序:

```
export ROBOT=sim  
roslaunch my_map_sim.launch
```

#### (5) 执行结果如图 5.6 所示。

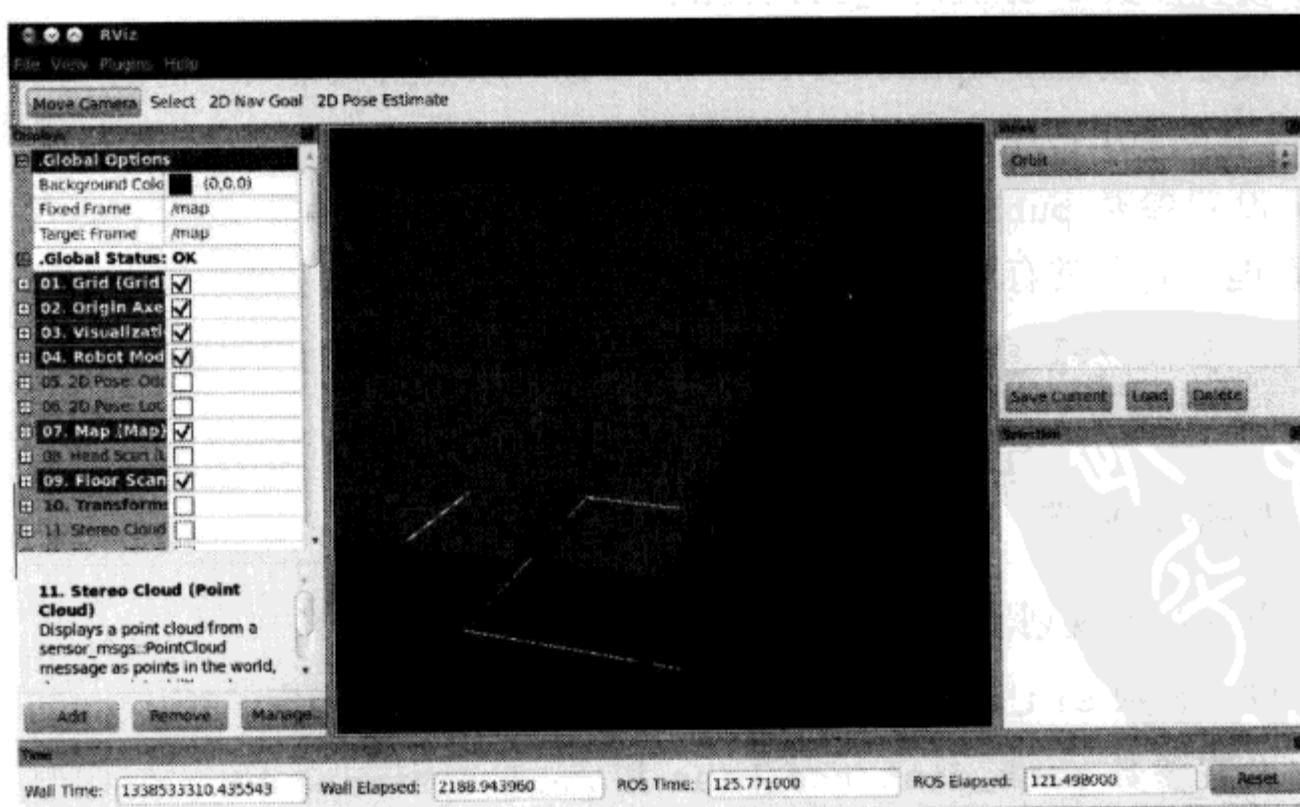


图 5.6 客户定制地图

## 5.5 配置和使用导航功能包集

### 5.5.1 导航功能包集基本操作

#### 1. 机器人准备工作

在一台新的机器人上调试导航功能包集的主要问题在于在局部规划器上调试参数。容易出错的地方在于里程计、定位、传感器等准备工作。要准备三部分的检查：距离传感器、里程计和定位。

(1) 距离传感器。如果机器人不能从距离传感器获取信息，则机器人不能执行导航。因此需要用户确认是否可以在 rviz 中获取传感器信息。

(2) 里程计。主要进行 2 个测试：一个是 `odometry` 参数设置是否适合于旋转，另一个是检查平移设置是否合适。

(3) 定位。首先运行 `GMapping` 或 `karto` 及操作杆使得机器人移动以生成地图。然后使用该地图以及 `AMCL` 实现机器人定位。

#### 2. 代价地图

下面需要确认代价地图配置正确。

(1) 设置观测值更新率：`expected_update_rate`。

(2) 设置参数 `transform_tolerance`。

(3) 设置参数 `map_update_rate`，该参数可以根据机器人当前的电量进行修改。如果电量不足，用户可以降低更新率。

(4) 设置参数 `publish_frequency`。该参数用来修改发布频率，它在使用 rviz 查看地图时非常有用。

(5) 参数 `voxel_grid` 或 `costmap` 的设置主要依赖于机器人所配置的传感器。

#### 3. 局部规划器

(1) 设置两个规划器的加速度限制参数。

(2) 设置参数 `sim_granularity` 使 CPU 降低推理分辨率。

(3) 设置参数 `path_distance_bias` 和 `goal_distance_bias` 以限制局部规划器与规划路径的偏差。

(4) 设置参数 `meter_scoring`。实现以智能方式推理代价函数。

### 5.5.2 在机器人上设置和配置导航功能包集

本小节介绍怎样在机器人上建立导航功能包集，内容包括：使用 `tf` 发送变换、发布里程计信息、发布传感器数据以及基本的导航配置。

#### 1. 机器人设置

当机器人已经配置完毕时，开始使用导航功能包集。图 5.7 显示了整个流程图框架。图中白色部分为 ROS 中必要的软件包模块，浅灰色部分为可选软件包模块，深灰色部分为必须为机器人平台创建的组件。

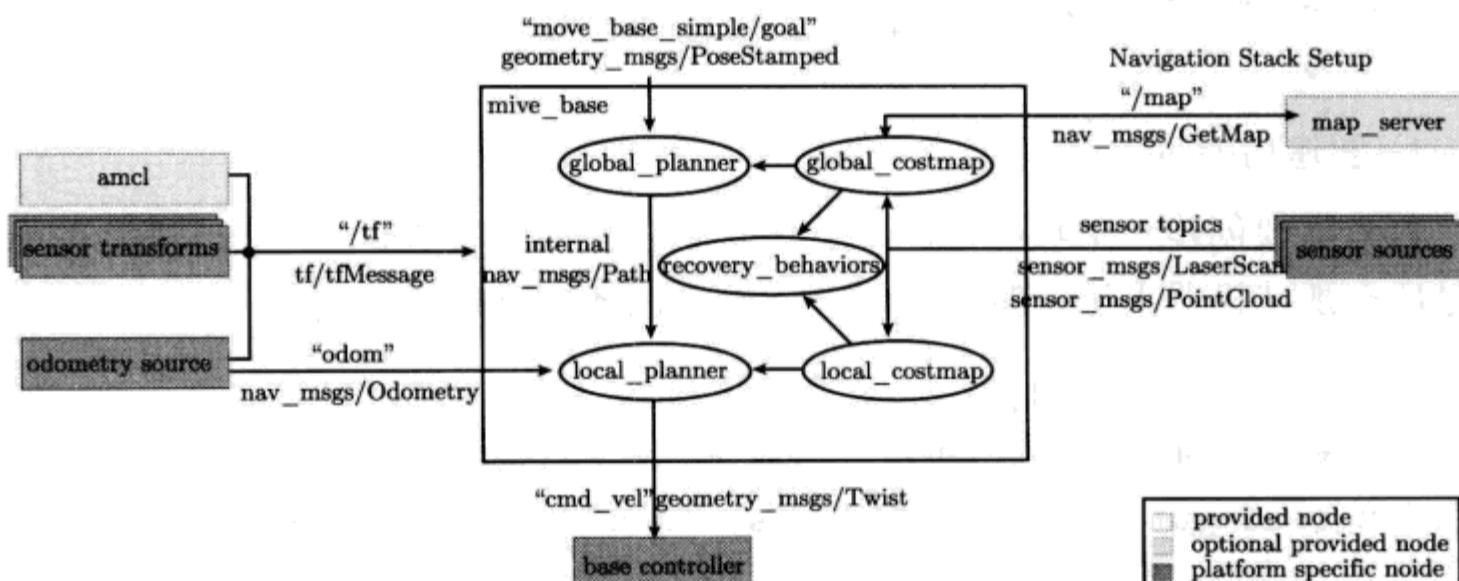


图 5.7 流程图框架

- (1) ROS。导航功能包集依赖于 ROS 系统。
- (2) 变换配置。导航功能包集需要机器人使用 `tf` 发布坐标系之间的信息。
- (3) 传感器信息。导航功能包集传感器信息避障，它假设传感器通过 ROS 发布 `sensor_msgs/LaserScan` 或 `sensor_msgs/PointCloud` 消息。支持的激光传感器有：SCIP2.0-compliant Hokuyo Laser Devices、Hokuyo Model 04LX-hokuyo\_node 和 SICK LMS2xx Lasers-sicktoolbox\_wrapper。
- (4) 里程计信息。导航功能包集需要使用 `tf` 和 `nav_msgs/Odometry` 消息来发布里程计信息。支持的平台有：Videre Erratic: erratic\_player 和 PR2: pr2\_mechanism\_controllers。
- (5) 基座控制。导航功能包集假设使用 `geometry_msgs/Twist` 消息发送速度命令，该消息位于机器人坐标系的 `cmd_vel` 主题。这也意味着必须有一个节点订阅主题 `cmd_vel`。该主题能够获取速度并转换为马达命令传送到机器人基座 ( $v_x, v_y, v_{theta} \leftrightarrow (cmd\_vel.linear.x, cmd\_vel.linear.y, cmd\_vel.angular.z)$ )。支持的平台有：Videre Erratic:erratic\_player 和 PR2: pr2\_mec-

hanism\_controllers。

(6) 地图 (map\_server)。导航功能包集本身并不需要操作地图，但是为了演示本小节的例子，假设用户有一幅地图可用。

## 2. 导航功能包集设置

本小节描述如何在机器人上配置导航功能包集。首先假设机器人需要的所有需求已经满足。特别地，这意味着机器人必须使用 tf 来发布坐标系信息，从所有传感器接收 sensor\_msgs/LaserScan 或 sensor\_msgs/PointCloud 消息，使用 tf 和 nav\_msgs/Odometry 消息发布里程计信息，同时也使用速度命令去发送消息到基座。

### 1) 创建功能包

首先需要创建一个功能包，用于存储所有配置和启动文件。该功能包的依赖项是所有那些机器人运行时需要的功能包。选择当前功能包创建的目录，运行如下命令：

```
roscreate-pkg my_robot_name_2dnav move_base my_tf_configuration_dep
my_odom_configuration_dep my_sensor_configuration_dep
```

### 2) 配置机器人启动文件

```

1 <launch>
2   <node pkg="sensor_node_pkg" type="sensor_node_type" name=
3     "sensor_node_name" output="screen">
4     <param name="sensor_param" value="param_value" />
5   </node>
6
6   <node pkg="odom_node_pkg" type="odom_node_type" name=
7     "odom_node" output="screen">
8     <param name="odom_param" value="param_value" />
9   </node>
10
10  <node pkg="transform_configuration_pkg" type=
11    "transform_configuration_type" name=
12    "transform_configuration_name" output="screen">
11    <param name="transform_configuration_param" value=
13      "param_value" />
12  </node>
13 </launch>
```

### 3) 代价地图配置: local\_costmap 和 global\_costmap

导航功能包集使用两种代价地图存储世界中的障碍物信息。一种代价地图用于全局规划, 意味着整个环境的长期规划, 另一种用于局部规划和避障。其中有些配置选项需要两个代价地图同时配合, 有些只需要一个代价地图配合。因此, 有三部分需要配置: 通用配置选项、全局配置选项、局部配置选项。

### 4) 基座局部规划器配置

TrajectoryPlannerROS:

```
max_vel_x: 0.45
min_vel_x: 0.1
max_rotational_vel: 1.0
min_in_place_rotational_vel: 0.4

acc_lim_th: 3.2
acc_lim_x: 2.5
acc_lim_y: 2.5

holonomic_robot: true
```

### 5) 创建导航功能包集启动文件

创建文件 move\_base.launch, 并添加如下代码:

```
1 <launch>
2   <master auto="start"/>
3
4   <!-- Run the map server -->
5   <node name="map_server" pkg="map_server" type="map_server"
     " args="$(find my_map_package)/my_map.pgm
       my_map_resolution"/>
6
7   <!-- Run AMCL -->
8   <include file="$(find amcl)/examples/amcl_omni.launch" />
9
10  <node pkg="move_base" type="move_base" respawn="false"
    name="move_base" output="screen">
11    <rosparam file="$(find my_robot_name_2dnav)/
      costmap_common_params.yaml" command="load" ns=
        "global_costmap" />
```

```
12 <rosparam file="$(find my_robot_name_2dnav)/  
13   costmap_common_params.yaml" command="load" ns=  
14     "local_costmap" />  
15 <rosparam file="$(find my_robot_name_2dnav)/  
16   local_costmap_params.yaml" command="load" />  
17 <rosparam file="$(find my_robot_name_2dnav)/  
18   global_costmap_params.yaml" command="load" />  
19 <rosparam file="$(find my_robot_name_2dnav)/  
20   base_local_planner_params.yaml" command="load" />  
21 </node>  
22 </launch>
```

## 6) 配置 AMCL

AMCL 有很多配置选项将会影响定位执行效果。相关信息可以参考: <http://www.ros.org/wiki/amcl>。

## 3. 运行导航功能包集

设置好导航功能包集后可以运行它。在两个新终端，分别运行：

```
roslaunch my_robot_configuration.launch  
roslaunch move_base.launch
```

### 5.5.3 rviz 与导航功能包集配合使用

rviz 是一个强大的可视化工具，它可以用在很多方面。导航功能包集需要设置的 rviz 参数包括以下几项。

#### 1. 2D 导航目标

- 主题: `move_base_simple/goal`
- 描述: 为机器人设置一个可以到达的目标。

#### 2. 2D 位姿估计

- 主题: `initialpose`
- 描述: 通过设置机器人在坐标系中的位置来初始化定位系统。

#### 3. 静态地图

- 主题: `nav_msg/GetMap`
- 描述: 显示地图服务器提供的静态地图。

#### 4. 粒子云

- 主题: `geometry_msgs/PoseArray`
- 描述: 显示机器人定位系统的粒子云, 粒子云表示机器人位姿定位系统的不确定性。分散程度高的点云代表高的不确定性, 反之密集的点云代表低不确定性。

#### 5. 机器人轨迹

- 主题: `local_costmap/robot_footprint`
- 类型: `geometry_msgs/Polygon`
- 描述: 显示机器人的轨迹。

#### 6. 障碍

- 主题: `local_costmap/obstacles`
- 类型: `nav_msgs/GridCells`
- 描述: 显示机器人在代价地图上的障碍, 为了避免碰撞, 机器人轨迹不能与障碍物单元格交叉。

#### 7. 膨胀障碍

- 主题: `local_costmap/inflated_obstacles`
- 类型: `nav_msgs/GridCells`
- 描述: 导航功能包集中的代价地图将根据机器人的半径将障碍物在代价地图上所占的范围扩大。为了避免碰撞, 机器人的中心点不能与包含扩大障碍的单元格相交叠。

#### 8. 全局规划

- 主题: `TrajectoryPlannerROS/global_plan`
- 类型: `nav_msgs/Path`
- 描述: 显示局部规划器正在执行的一段全局规划结果。

#### 9. 局部规划

- 主题: `TrajectoryPlannerROS/local_plan`
- 类型: `nav_msgs/Path`
- 描述: 显示局部规划器发送的跟速度指令相关的轨迹。

## 10. 规划器规划

- 主题: `NavfnROS/plan`
- 类型: `nav_msgs/Path`
- 描述: 显示全局规划器计算出的整个路径的规划。

## 11. 当前目标

- 主题: `current_goal`
- 类型: `geometry_msgs/PoseStamped`
- 描述: 显示导航功能包集尝试到达的目标。

### 5.5.4 发送目标到导航功能包集

导航功能包集的功能是使机器人从一个位置移动到另一个位置，并且途中能安全地避过障碍。通常，机器人被下达一个移动到目标位置的任务。例如，告诉机器人去一个特定的办公室，指令有两种形式：第一种，用户可以点击地图上办公室的位置，然后机器人就会尝试移动到该位置；第二种，用代码给机器人传达目标指令，就像使用 rviz 时在后台做的一样。例如，编程来让机器人插上插座的任务中，机器人首先检测到电源插座，然后机器人要与墙保持在一尺的地方，尝试用手臂把插头插入插座。本小节提供一个使用用户代码来给导航功能包集发送简单目标的例子。

#### 1. ROS 设置

为了创建一个给导航功能包集发送目标的 ROS 节点，首先需要创建一个功能包。我们使用 `roscreate-pkg` 命令来创建功能包和相关依赖项，如 `move_base_msgs`、`actionlib` 等。具体命令如下：

```
roscreate-pkg simple_navigation_goals move_base_msgs actionlib  
roscpp
```

在运行该指令之后，改变当前目录到创建功能包的目录下：

```
roscd simple_navigation_goals
```

#### 2. 创建节点

创建功能包之后，需要编写给机器人发送目标指令的程序。新建文件 `src/simple_navigation_goals.cpp`，并输入如下代码：

```
1 #include <ros/ros.h>
2 #include <move_base_msgs/MoveBaseAction.h>
3 #include <actionlib/client/simple_action_client.h>
4
5 typedef actionlib::SimpleActionClient<move_base_msgs::
6     MoveBaseAction> MoveBaseClient;
7
8 int main(int argc, char** argv){
9     ros::init(argc, argv, "simple_navigation_goals");
10
11     MoveBaseClient ac("move_base", true);
12
13     while(!ac.waitForServer(ros::Duration(5.0))){
14         ROS_INFO("Waiting for the move_base action server to
15             come up");
16     }
17
18     move_base_msgs::MoveBaseGoal goal;
19
20     goal.target_pose.header.frame_id = "base_link";
21     goal.target_pose.header.stamp = ros::Time::now();
22
23     goal.target_pose.pose.position.x = 1.0;
24     goal.target_pose.pose.orientation.w = 1.0;
25
26     ROS_INFO("Sending goal");
27     ac.sendGoal(goal);
28
29     ac.waitForResult();
30
31     if(ac.getState() == actionlib::SimpleClientGoalState::
32         SUCCEEDED)
33         ROS_INFO("Hooray, the base moved 1 meter forward");
34     else
35         ROS_INFO("The base failed to move forward 1 meter for
36             some reason");
37
38     return 0;
39 }
```

代码解释：

(1) 代码第 3 行。表示包含 `move_base` 的动作分类，重点是 `move_base` 动作接收客户端发送的目标，然后试着使机器人运动到一个世界坐标系中的特定位置。

(2) 代码第 5 行。为了方便 `SimpleActionClient` 与 `MoveBaseAction` 接口的动作交互，这里创建一个 `typedef`。

(3) 代码第 10 行。该行代码构建一个动作客户端与 `MoveBaseAction` 接口进行 `move_base` 交互。同时也通知动作客户端启动一个 `ros::spin()` 的线程来回查通过传递 `true` 作为 `MoveBaseClient` 构造函数的第二个参数。

(4) 代码第 12~13 行。这几行代码是等待行动服务器发布已经启动和开始接近目标。

(5) 代码第 16~25 行。这里创建一个目标，然后用于 `MoveBaseAction.h` 头文件中包含的 `move_base_msgs::MoveBaseGoal` 消息形式发送给 `move_base`。这将使得移动平台 `base_link` 坐标系向前移动 1m。`ac.sendGoal` 所在行代码将会把目标发送到 `move_base` 节点。

(6) 代码第 27~32 行。用 `ac.waitForGoalToFinish` 来查询在 `move_base` 完成发送给它的任务之前，哪一部分会阻塞，在函数调用的过程中，需要等待目标任务完成。

### 3. 编译并运行

将 `src/simple_navigation_goals.cpp` 文件添加到 `CMakeLists.txt` 文件文件的尾部：

```
rosbuild_add_executable(simple_navigation_goals src/simple_
navigation_goals.cpp)
```

完成后，将其编译成可执行文件：`make`。

下面开始运行，这里假定导航功能包集已经正确启动。需要确认的是动作的名字，这里设置为 `move_base`，需要跟用户代码中所用的名字一致。然后可以用 `rostopic` 的指令来测试它。

```
rostopic list | grep move_base/goal
```

如果有内容显示，即表示运行成功。除此之外，还需要找到导航功能包集用的动作的名字，在 PR2 平台上有一个可选择动作名称是 `move_base_local`，然

后更新 `src/simple_navigation_goals.cpp` 文件中相关部分。随后简单运行所创建的可执行文件即可。

```
./bin/simple_navigation_goals
```

在运行这个程序之后，机器人会向前移动 1m。



# 第六章 抓取操作

机器人不同于计算机的地方在于能够直接与环境交互并影响和改变环境。这种交互可以采用多种形式，例如获取（抓取）物体、放置物体、开门、使用电梯等。本章主要讲述 ROS 中与抓取和操作物体相关内容。本章中所提到内容如果没有特殊说明，均为 Electric 版本下执行。

## 6.1 机器人手臂的运动规划

本节通过一个基于 PR2 机器人手臂的运动规划和控制的演示实例，介绍手臂导航功能包集 `arm_navigation` 的功能，以及如何在执行运动规划和避障的同时对手臂进行控制。本节中提供了所有这些节点的缺省执行，其中一些是 PR2 特定的。

### 6.1.1 安装和配置

首先需要安装手臂导航功能包（Electric 版本下），在终端执行安装命令：

```
sudo apt-get install ros-electric-pr2-arm-navigation
```

安装完功能包集之后，需要配置环境变量。注意，在每个运行终端都需要设置：`export ROBOT=sim`。

### 6.1.2 编译手臂导航功能包集

手臂导航功能包集实现了运动规划、控制、轨迹滤波和求解之间的协调，这一系列的功能是通过多个相关节点的数据通信来实现的。这些节点设置在功能包集内部，每个节点只做很少计算，使得整个功能包更加模块化，以便于这些节点的重用和替换。

手臂运动 `move_arm` 是手臂导航功能包集的一个节点，主要功能是实现对手臂进行移动控制。在进行手臂控制之前，必须先编译 `pr2_arm_navigation_action` 功能包，执行如下命令：

```
roscd pr2_arm_navigation_actions  
rosmake
```

### 6.1.3 启动模拟器和仿真环境

本例中使用 `gazebo` 模拟器来实现 PR2 手臂控制的仿真。首先启动模拟器，并调用 `pr2_empty_world.launch` 仿真环境：

```
roslaunch pr2_gazebo pr2_empty_world.launch
```

结果如图 6.1 所示。

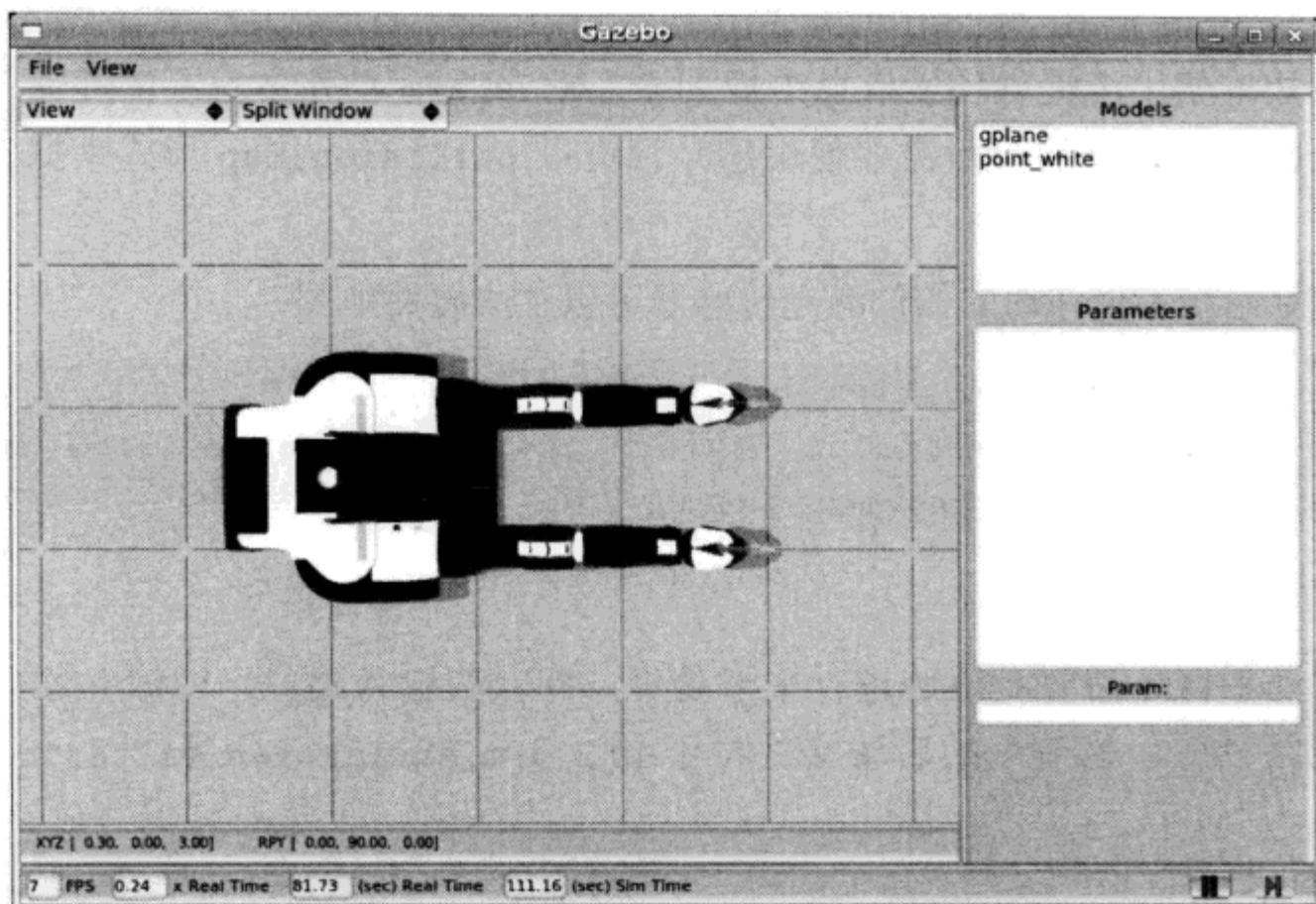


图 6.1 gazebo 中手臂界面截图

### 6.1.4 启动相关节点

移动手臂涉及规划、控制、逆运动学求解和轨迹监控等节点，因此需要生成一个启动文件，用于同时启动所有相关的节点。首先，在 `pr2_3dnav` 功能包中建立名为 `move_right_arm.launch` 的启动文件，并添加如下代码：

```
1 <launch>
2
3 </launch>
```

依次添加以下节点到启动文件中：

1) 规划节点

手臂需要使用规划节点来生成要执行的动作规划。目前有三种支持的规划器，这里使用 OMPL 规划器。每一个规划器有自己定制的启动文件来启动规划

器。这里 OMPL 规划器的启动文件位于 `pr2_arm_navigation_planning` 目录，这个文件需要包含在 `move_right_arm.launch` 中。

```
1 <!-- load planning -->
2 <include file="$(find pr2_arm_navigation_planning)/launch/
    ompl_planning.launch"/>
```

## 2) 感知节点

感知框架用于提供环境信息给规划器。`move_arm` 动作本身不需要使用感知管道，但与之相关的节点需要使用感知管道以 `collision_map` 形式来提供环境信息。

包括感知管道在内的俯仰激光传感器可以生成碰撞地图：

```
1 <!-- load perception -->
2 <include file="$(find pr2_arm_navigation_perception)/launch/
    /laser-perception.launch"/>
```

## 3) 轨迹滤波节点

移动手臂还需要对轨迹进行平滑处理，即获取输入路径，滤波并添加速度信息到路径中。该节点的启动文件位于 `pr2_arm_navigation_filtering` 功能包内。

```
1 <!-- load monitor -->
2 <include file="$(find pr2_arm_navigation_actions)/launch/
    environment_server_right_arm.launch"/>
```

## 4) 运动学逆解求解节点

移动手臂使用服务调用执行运动学逆解求解程序。这里使用特定用于 PR2 的逆解求解程序。为启动该节点，需要在启动文件中包含下面代码：

```
1 <!-- load ik -->
2 <include file="$(find pr2_arm_navigation_kinematics)/launch/
    /right_arm_collision_free_ik.launch"/>
```

## 5) 监控/碰撞检测节点

移动手臂需要监控/碰撞检测节点来提供关于碰撞的信息，用于对轨迹安全执行进行监控并提供整个机器人的信息。所有这些服务都在节点 `environment_server` 中执行。为启动该节点，需要在启动文件中包含下面代码：

```
1 <!-- load monitor -->
2 <include file="$(find pr2_arm_navigation_actions)/launch/
    environment_server_right_arm.launch"/>
```

### 6) 手臂简单动作节点

手臂简单动作节点 `move_arm_simple_action` 本身是可执行的，它完成了主要的工作，包括执行状态机、收听目标、询问计划和发送轨迹并监控执行。为启动该节点，需要在启动文件中包含下面代码：

```
1 <!-- load move_arm -->
2 <include file="$(find pr2_arm_navigation_actions)/launch/
    move_right_arm.launch"/>
```

完整的用于 PR2 左右手臂的手臂导航启动文件位于 `pr2_3dnav` 功能包 (`left_arm_navigation.launch` 和 `right_arm_navigation.launch`)。建立节点启动文件之后，就可以一次启动手臂导航功能的所有相关节点。在新的终端执行如下命令：

```
roslaunch right_arm_navigation.launch
```

所有的节点会依次启动。完成启动之后，可以看到类似下面的信息：

```
[ INFO] 706.573999990: Move arm action started
```

#### 6.1.5 控制手臂运动

完成上述准备工作后，便可以实现对手臂控制。本小节中的代码可以在 `pr2_arm_navigation_tutorials` 功能包的 `src/` 文件夹中找到。

##### 1. 给定关节变量移动手臂

运行可执行文件：

```
rosrun pr2_arm_navigation_tutorials move_arm_joint_goal
```

运行成功后，结果如下：

```
[ INFO] 115.660000000: Connected to server
[ INFO] 121.854000000: Action finished: SUCCEEDED
```

模拟器运行结果如图 6.2 所示。

`move_arm_joint_goal.cpp` 的源代码如下：

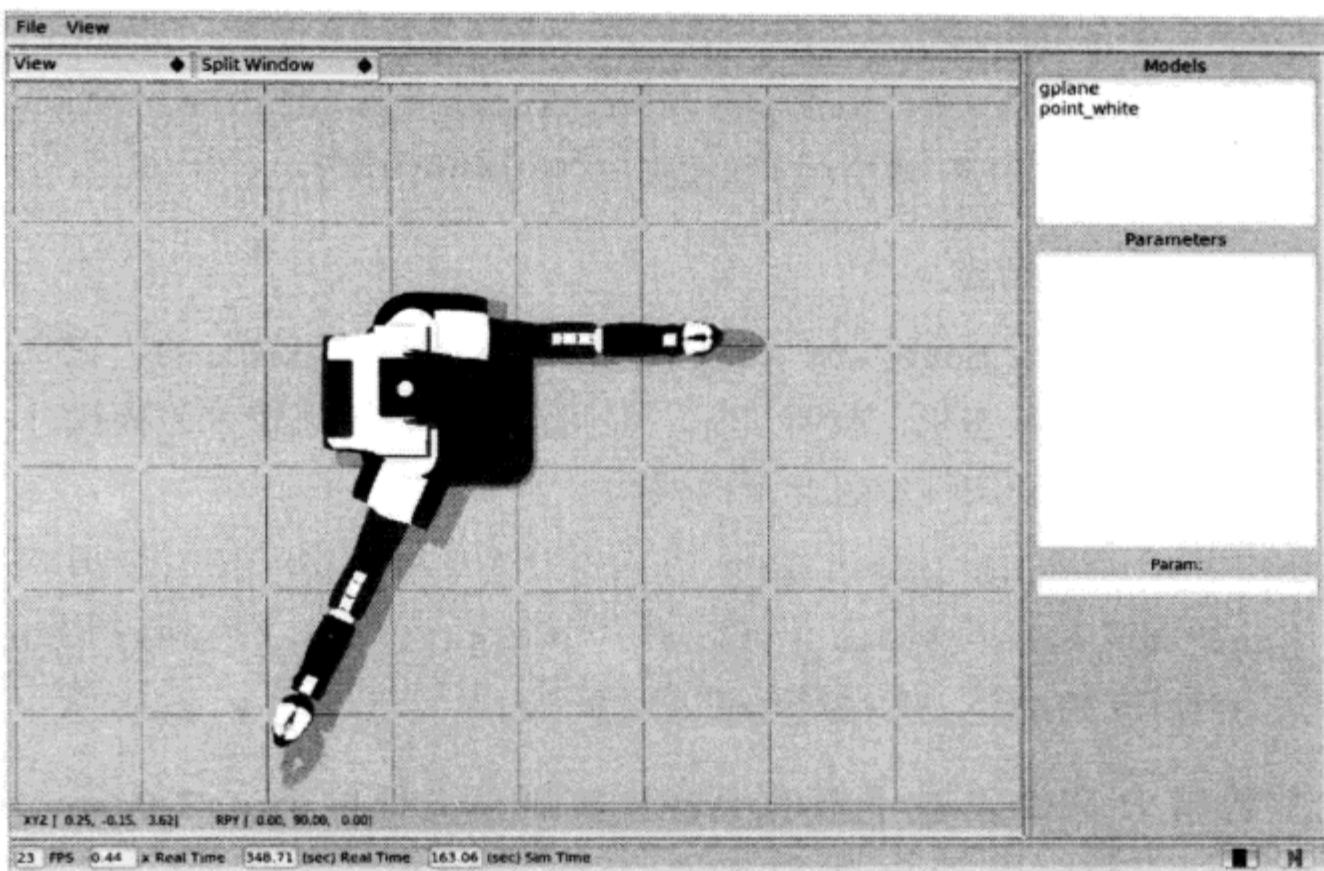


图 6.2 给定关节变量移动手臂

```

1 #include <ros/ros.h>
2 #include <actionlib/client/simple_action_client.h>
3 #include <move_arm_msgs/MoveArmAction.h>
4
5 int main(int argc, char **argv){
6     ros::init (argc, argv, "move_arm_joint_goal_test");
7     ros::NodeHandle nh;
8     actionlib::SimpleActionClient<move_arm_msgs::
9         MoveArmAction> move_arm("move_right_arm",true);
10
11     move_arm.waitForServer();
12     ROS_INFO("Connected to server");
13
14     move_arm_msgs::MoveArmGoal goalB;
15     std::vector<std::string> names(7);
16     names[0] = "r_shoulder_pan_joint";
17     names[1] = "r_shoulder_lift_joint";
18     names[2] = "r_upper_arm_roll_joint";
19     names[3] = "r_elbow_flex_joint";
20     names[4] = "r_forearm_roll_joint";
21     names[5] = "r_wrist_flex_joint";
22     names[6] = "r_wrist_roll_joint";

```

```
22
23     goalB.motion_plan_request.group_name = "right_arm";
24     goalB.motion_plan_request.num_planning_attempts = 1;
25     goalB.motion_plan_request.allowed_planning_time = ros::
26             Duration(5.0);
27
28     goalB.motion_plan_request.planner_id=std::string("");
29     goalB.planner_service_name = std::string("ompl_planning/
30             plan_kinematic_path");
31     goalB.motion_plan_request.goal_constraints .
32             joint_constraints.resize(names.size());
33
34     for (unsigned int i = 0 ; i < goalB.motion_plan_request .
35             goal_constraints.joint_constraints.size(); ++i)
36     {
37         goalB.motion_plan_request.goal_constraints .
38             joint_constraints[i].joint_name = names[i];
39         goalB.motion_plan_request.goal_constraints .
40             joint_constraints[i].position = 0.0;
41         goalB.motion_plan_request.goal_constraints .
42             joint_constraints[i].tolerance_below = 0.1;
43         goalB.motion_plan_request.goal_constraints .
44             joint_constraints[i].tolerance_above = 0.1;
45     }
46
47     goalB.motion_plan_request.goal_constraints .
48         joint_constraints[0].position = -2.0;
49     goalB.motion_plan_request.goal_constraints .
50         joint_constraints[3].position = -0.2;
51     goalB.motion_plan_request.goal_constraints .
52         joint_constraints[5].position = -0.15;
53
54     if (nh.ok())
55     {
56         bool finished_within_time = false;
57         move_arm.sendGoal(goalB);
58         finished_within_time = move_arm.waitForResult(ros::
59                 Duration(200.0));
60         if (!finished_within_time)
```

```

49     {
50         move_arm.cancelGoal();
51         ROS_INFO("Timed out achieving goal A");
52     }
53 else
54 {
55     actionlib::SimpleClientGoalState state = move_arm.
56         getState();
57     bool success = (state == actionlib::
58         SimpleClientGoalState::SUCCEEDED);
59     if(success)
60         ROS_INFO("Action finished: %s", state.toString().
61             c_str());
62     else
63         ROS_INFO("Action failed: %s", state.toString().c_str()
64     );
}

```

### 代码解释：

(1) 代码第 3 行。这行代码包含了用于 `move_arm` 功能包集的动作参数。本质上讲，`move_arm` 动作是从客户端接收目标点，并将机器人手臂移动到特定位置。

(2) 代码第 8 行。这行代码构建了一个动作客户端，该客户端用于与名为 `move_right_arm` 的动作进行通信。同时，它告诉动作客户端开始一个名为 `ros::spin()` 的线程，通过设置 `MoveArmClient` 构造函数的第二个参数为 `true`，以使得 ROS 的回调函数被处理。

(3) 代码第 10 行。该行程序等待动作服务器报告它已经准备好处理目标了。

(4) 代码第 13~41 行。这里使用 `move_arm_msgs::MoveArmGoal` 消息类型创建一个目标，并传送给 `move_arm`。目标是在移动手臂目标消息的 `motion_plan_request` 域中指定。

- `group_name`。群组的名称。模型名称是在参数服务器中使用 yaml 配置文件预定义的。每一个群组都必须包括一组关节和连杆。

- `planner_id`。规划器的名称。

• `planner_service_name`。这是 `move_arm` 用于调用规划器的服务调用名称。它允许同时寻址多个规划器。由于规划库可在同一个服务调用内提供不同类型的规划器，因此 `planner_id` 不同于 `planner_service_name`。

• `goal_constraints`。目标约束矢量及其公差。本例中有 7 个关节，也就有 7 个关节约束。

• `joint_constraints`。每一个关节约束包括 `joint_name`、目标位置及其允许公差。对于 `r_shoulder_pan_joint` 来说，希望到达的目标位置是 -2.0rad；可以接受的关节位置为 -2.1~ -1.9rad，因此，允许的公差范围为 `[position-tolerance_below, position+tolerance_above]`。

• `num_planning_attempts`。指定移动手臂调用规划器去尝试和获取成功规划的次数。

• `allowed_planning_time`。传递到规划器的参数，它指定了允许规划的最大时间数。

(5) 代码第 46 行。调用 `move_arm.sendGoal` 实际上用于 `move_arm` 节点来处理目标位置。

(6) 代码第 47 行。现在需要等待目标完成。这里指定了在 200s 内手臂要完成该动作：`move_arm.waitForGoalToFinish(ros::Duration(200.0))`。完成之后，用户可以根据返回的消息检查是否执行成功。

## 2. 给定位姿移动手臂

然后运行可执行文件：

```
rosrun pr2_arm_navigation_tutorials move_arm_simple_pose_goal
```

运行成功后，结果如下：

```
[ INFO] 115.660000000: Connected to server  
[ INFO] 121.854000000: Action finished: SUCCEEDED
```

模拟器运行结果如图 6.2 所示。

`move_arm_simple_pose_goal.cpp` 的源代码如下：

```
1 #include <ros/ros.h>  
2 #include <actionlib/client/simple_action_client.h>  
3  
4 #include <move_arm_msgs/MoveArmAction.h>  
5 #include <move_arm_msgs/utils.h>  
6
```

```
7 int main(int argc, char **argv){  
8     ros::init (argc, argv, "move_arm_pose_goal_test");  
9     ros::NodeHandle nh;  
10    actionlib::SimpleActionClient<move_arm_msgs::  
11        MoveArmAction> move_arm("move_right_arm",true);  
12    move_arm.waitForServer();  
13    ROS_INFO("Connected to server");  
14    move_arm_msgs::MoveArmGoal goalA;  
15  
16    goalA.motion_plan_request.group_name = "right_arm";  
17    goalA.motion_plan_request.num_planning_attempts = 1;  
18    goalA.motion_plan_request.planner_id=std::string("");  
19    goalA.planner_service_name = std::string("ompl_planning/  
20        plan_kinematic_path");  
21    goalA.motion_plan_request.allowed_planning_time = ros::  
22        Duration(5.0);  
23  
24    motion_planning_msgs::SimplePoseConstraint desired_pose;  
25    desired_pose.header.frame_id = "torso_lift_link";  
26    desired_pose.link_name = "r_wrist_roll_link";  
27    desired_pose.pose.position.x = 0.75;  
28    desired_pose.pose.position.y = -0.188;  
29    desired_pose.pose.position.z = 0;  
30  
31    desired_pose.pose.orientation.x = 0.0;  
32    desired_pose.pose.orientation.y = 0.0;  
33    desired_pose.pose.orientation.z = 0.0;  
34    desired_pose.pose.orientation.w = 1.0;  
35  
36    desired_pose.absolute_position_tolerance.x = 0.02;  
37    desired_pose.absolute_position_tolerance.y = 0.02;  
38    desired_pose.absolute_position_tolerance.z = 0.02;  
39  
40    desired_pose.absolute_roll_tolerance = 0.04;  
41    desired_pose.absolute_pitch_tolerance = 0.04;  
42    desired_pose.absolute_yaw_tolerance = 0.04;  
43  
44    move_arm_msgs::addGoalConstraintToMoveArmGoal(  
45        desired_pose,goalA);
```

```
42
43     if (nh.ok())
44     {
45         bool finished_within_time = false;
46         move_arm.sendGoal(goalA);
47         finished_within_time = move_arm.waitForResult(ros::
48             Duration(200.0));
49         if (!finished_within_time)
50         {
51             move_arm.cancelGoal();
52             ROS_INFO("Timed out achieving goal A");
53         }
54     }
55     actionlib::SimpleClientGoalState state = move_arm.
56         getState();
57     bool success = (state == actionlib::
58         SimpleClientGoalState::SUCCEEDED);
59     if(success)
60         ROS_INFO("Action finished: %s", state.toString().
61             c_str());
62     else
63         ROS_INFO("Action failed: %s", state.toString().c_str(
64             ));
```

代码解释：

(1) 代码第 4 行。这行代码包含了用于 `move_arm` 功能包集的动作参数。实际上，`move_arm` 动作是从客户端接收目标点，并将机器人手臂移动到目标位置。

(2) 代码第 10 行。这行代码构建了一个动作客户端，该客户端用于与名为 `move_right_arm` 的动作进行通信。同时，它告诉动作客户端开始一个名为 `ros::spin()` 的线程，通过设置 `MoveArmClient` 构造函数的第二个参数为 `true`，以使得 ROS 的回调被处理。

(3) 代码第 11 行。该行程序等待动作服务器报告它已经准备好处理目标了。

(4) 代码第 13~19 行。这里使用 `move_arm_msgs::MoveArmGoal` 消息类型

创建一个目标并传送给 `move_arm`。目标在移动手臂目标消息的 `motion_plan_request` 域中指定。

- `group_name`。群组的名称。模型名称是在参数服务器中使用 yaml 配置文件预定义的。

- `planner_id`。规划器的名称。

- `planner_service_name`。这是 `move_arm` 用于调用规划器的服务调用名称。它允许规划器同时寻址多个规划器。由于规划库可在同一个服务调用内提供不同类型的规划器，因此 `planner_id` 不同于 `planner_service_name`。

- `num_planning_attempts`。指定移动手臂规划器调用规划器去尝试和获取成功规划的次数。

- `allowed_planning_time`。传递到规划器的参数，它指定了允许规划的极大时间。

(5) 代码第 21~41 行。本段代码用于移动手臂到位姿目标。为了实现该目标，在手臂末端执行器上创建一个简单的位姿约束，它指定了手臂将要到达的位置和方向。然后将该约束增加到移动手臂的目标消息中。

每一个简单的位姿约束都包括一个头、连杆名称和目标位姿及其允许的公差。本例中，希望移动末端执行器连杆 (`r_wrist_roll_link`) 到 `torso_lift_link` 坐标系中的位置 (0.75, -0.188, 0)。允许的最终结果是以期望的目标位置为中心、边长为 0.02m 的立方体。允许的最终位姿为以期望位姿中心，其旋转角、俯仰角、偏转角的公差为 (0.04, 0.04, 0.04)rad。

(6) 代码第 41、42 行。该行增加位姿约束。位于 `move_arm_msgs/utils.h` 的函数 `addGoalConstraintToMoveArmGoal` 将 `SimplePoseConstraint` 消息转换为更复杂的可以为 `move_arm` 最终使用的形式。

(7) 代码第 46 行。调用 `move_arm.sendGoal` 就是将目标发送给 `move_arm` 节点来处理。

(8) 代码第 47 行。现在需要等待目标完成。这里指定了在 200s 内手臂要完成该动作：`move_arm.waitForGoalToFinish (ros::Duration(200.0))`。完成之后，用户可以根据返回的消息检查是否执行成功。

### 3. 复杂位姿目标约束

本节将使用动作应用端向 `move_arm` 节点发送一个复杂位姿目标，并将手臂移动到该指定位姿。

运行可执行文件：

```
rosrun pr2_arm_navigation_tutorials move_arm_pose_goal
```

运行成功后，结果如下：

```
[ INFO] 115.660000000: Connected to server  
[ INFO] 121.854000000: Action finished: SUCCEEDED
```

pr2\_arm\_navigation\_tutorials 功能包内 src/move\_arm\_pose\_goal.cpp

Diamondback 版本文件代码如下：

```
1 #include <ros/ros.h>  
2 #include <actionlib/client/simple_action_client.h>  
3 #include <move_arm_msgs/MoveArmAction.h>  
4  
5 int main(int argc, char **argv){  
6     ros::init (argc, argv, "move_arm_joint_goal_test");  
7     ros::NodeHandle nh;  
8     actionlib::SimpleActionClient<move_arm_msgs::  
9         MoveArmAction> move_arm("move_right_arm",true);  
10  
11     move_arm.waitForServer();  
12     ROS_INFO("Connected to server");  
13  
14     move_arm_msgs::MoveArmGoal goalA;  
15  
16     goalA.motion_plan_request.group_name = "right_arm";  
17     goalA.motion_plan_request.num_planning_attempts = 1;  
18     goalA.motion_plan_request.allowed_planning_time = ros::  
19         Duration(5.0);  
20  
21     nh.param<std::string>("planner_id",goalA.  
22         motion_plan_request.planner_id,std::string(""));  
23     nh.param<std::string>("planner_service_name",goalA.  
24         planner_service_name,std::string("ompl_planning/  
25             plan_kinematic_path"));  
26     goalA.motion_plan_request.goal_constraints.  
27         set_position_constraints_size(1);  
28     goalA.motion_plan_request.goal_constraints.  
29         position_constraints[0].header.stamp = ros::Time::now  
30         ();
```

```
23     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].header.frame_id =  
              "torso_lift_link";  
24  
25     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].link_name =  
              "r_wrist_roll_link";  
26     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].position.x = 0.75;  
27     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].position.y = -0.188;  
28     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].position.z = 0;  
29  
30     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].constraint_region_shape.type  
= geometric_shapes_msgs::Shape::BOX;  
31     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].constraint_region_shape.  
dimensions.push_back(0.02);  
32     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].constraint_region_shape.  
dimensions.push_back(0.02);  
33     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].constraint_region_shape.  
dimensions.push_back(0.02);  
34  
35     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].constraint_region_.  
orientation.w = 1.0;  
36     goalA.motion_plan_request.goal_constraints.  
          position_constraints[0].weight = 1.0;  
37  
38     goalA.motion_plan_request.goal_constraints.  
          set_orientation_constraints_size(1);  
39     goalA.motion_plan_request.goal_constraints.  
          orientation_constraints[0].header.stamp = ros::Time::  
now();  
40     goalA.motion_plan_request.goal_constraints.
```

```
        orientation_constraints[0].header.frame_id =
        "torso_lift_link";
41    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].link_name =
        "r_wrist_roll_link";
42    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].orientation.x = 0.0;
43    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].orientation.y = 0.0;
44    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].orientation.z = 0.0;
45    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].orientation.w = 1.0;
46
47    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].absolute_roll_tolerance =
        0.04;
48    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].absolute_pitch_tolerance =
        0.04;
49    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].absolute_yaw_tolerance =
        0.04;
50
51    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].weight = 1.0;
52
53    if (nh.ok())
54    {
55        bool finished_within_time = false;
56        move_arm.sendGoal(goalA);
57        finished_within_time = move_arm.waitForResult(ros::
58            Duration(200.0));
58        if (!finished_within_time)
59        {
60            move_arm.cancelGoal();
61            ROS_INFO("Timed out achieving goal A");
62        }
63    else
```

```

64      {
65          actionlib::SimpleClientGoalState state = move_arm.
66              getState();
67          bool success = (state == actionlib::
68              SimpleClientGoalState::SUCCEEDED);
69          if(success)
70              ROS_INFO("Action finished: %s", state.toString().
71                  c_str());
72          else
73              ROS_INFO("Action failed: %s", state.toString().c_str(
74          ));

```

代码解释：

(1) 代码第 3 行。这行代码包含了用于 `move_arm` 功能包集的动作参数。本质上讲，`move_arm` 动作是从客户端接收目标点，并将机器人手臂移动到目标位置。

(2) 代码第 8 行。这行代码构建了一个动作客户端，该客户端用于与名为 `move_right_arm` 的动作进行通信。同时，它告诉动作客户端开始一个名为 `ros::spin()` 的线程，通过设置 `MoveArmClient` 构造函数的第二个参数为 `true`，以使得 ROS 的回调被处理。

(3) 代码第 10 行。该行程序等待动作服务器报告它已经准备好处理目标了。

(4) 代码第 13~17 行。这里使用 `move_arm_msgs::MoveArmGoal` 消息类型创建一个目标，并传送给 `move_arm`。目标在移动手臂目标消息的 `motion_plan_request` 域中指定。本段代码中的参数含义与“给定关节变量移动手臂”小节中代码第 13~41 行含义相同。

(5) 代码第 19、20 行。每一个位置约束都包含一个头、连杆和目标位姿及其允许公差(通过 `geometric_shapes/Shape` 消息指定)。在这个例子中，我们希望移动末端执行器连杆(`r_wrist_roll_link`)到 `torso_lift_link` 坐标系中的位置(0.75, -0.188, 0)。

(6) 代码第 30~33 行。指定公差范围以期望位置为中心、边长为 0.02m 的立方体。

(7) 代码第 35 行。立方体沿着 `torso_lift_link` 坐标系的坐标轴方向。

(8) 代码第 36 行。每一个约束都带有权重因子。规划器可以任意它喜欢的形式来表示该约束。

(9) 代码第 38~49 行。每一个方向约束都包含一个头、连杆、允许位姿及其公差。目标位姿在 `geometry_msgs/Orientation` 消息中指定。在这个例子中，我们希望将末端执行器连杆 (`r_wrist_roll_link`) 移动到 `torso_lift_link` 坐标系中的 (`roll = 0.0, pitch = -0.0, yaw = 0.0`)。同时可以注意到，在文件头中指定了当前时间。

(10) 代码第 47~49 行。允许的位姿误差为 0.04rad，即转动角、俯仰角、偏转角的误差容忍限度为 0.04rad。允许公差范围是在 `link` 坐标系中缺省设置的。为了在文件头坐标系中指定这些允许公差，需要指定适合位姿约束类型。

(11) 代码第 51 行。每一个约束上都有一个选定权重因子。

(12) 代码第 56 行。调用 `move_arm.sendGoal` 就是将目标发送给 `move_arm` 节点处理。

(13) 代码第 57 行。现在需要等待目标完成。这里指定了在 200s 内手臂要完成该动作：`move_arm.waitForResult(ros::Duration(200.0))`。完成之后，用户可以根据返回的消息检查是否执行成功。

#### 4. 给定路径约束的运动规划

本节主要讲述如何为 `move_arm` 指定路径约束。这在很多情况下很有用，如需要机器人移动装满水的玻璃杯时，指定约束为玻璃杯需要竖直。路径约束为空间约束而非时间约束。

运行可执行文件：

```
rosrun pr2_arm_navigation_tutorials move_arm_path_constraints
```

运行成功后，结果如下：

```
[ INFO] 115.660000000: Connected to server
```

```
[ INFO] 121.854000000: Action finished: SUCCEEDED
```

机器人手臂将会首先移动到第一个位置，然后在保持夹持器竖直的同时移动到下一个目标位置。由于机器人初始位姿是机器人最终约束的一部分，因此需要记住的是：机器人初始状态必须满足这些路径约束。目前路径约束还不成熟，有些时候需要用户将允许公差设置得高一些。

`move_arm_path_constraints.cpp` 代码类似于复杂位姿目标约束的代码。主要的不同在于路径约束限制了手臂移动的路径。该代码要实现的主要功能首先

是不带路径约束并保持夹持器竖直；然后加上路径约束，保持夹持器竖直并将右臂移动到另一个位置。

`move_arm_path_constraints.cpp` 源代码如下：

```
1 #include <ros/ros.h>
2 #include <actionlib/client/simple_action_client.h>
3 #include <move_arm_msgs/MoveArmAction.h>
4
5 int main(int argc, char **argv){
6     ros::init(argc, argv, "move_arm_joint_goal_test");
7     ros::NodeHandle nh;
8     actionlib::SimpleActionClient<move_arm_msgs::
9         MoveArmAction> move_arm("move_right_arm",true);
10
11     move_arm.waitForServer();
12     ROS_INFO("Connected to server");
13
14     move_arm_msgs::MoveArmGoal goalA;
15
16     goalA.motion_plan_request.group_name = "right_arm";
17     goalA.motion_plan_request.num_planning_attempts = 1;
18     goalA.motion_plan_request.allowed_planning_time = ros::
19         Duration(5.0);
20
21     nh.param<std::string>("planner_id",goalA.
22         motion_plan_request.planner_id,std::string(""));
23     nh.param<std::string>("planner_service_name",goalA.
24         planner_service_name,std::string("ompl_planning/
25             plan_kinematic_path"));
26     goalA.motion_plan_request.goal_constraints.
27         set_position_constraints_size(1);
28     goalA.motion_plan_request.goal_constraints.
29         position_constraints[0].header.stamp = ros::Time::now
30         ();
31     goalA.motion_plan_request.goal_constraints.
32         position_constraints[0].header.frame_id =
33         "torso_lift_link";
34
35     goalA.motion_plan_request.goal_constraints.
36         position_constraints[0].link_name =
```

```
        "r_wrist_roll_link";
26   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].position.x = 0.2;
27   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].position.y = -0.90;
28   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].position.z = 0;
29
30   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].constraint_region_shape.type
    = geometric_shapes_msgs::Shape::BOX;
31   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].constraint_region_shape.
    dimensions.push_back(0.02);
32   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].constraint_region_shape.
    dimensions.push_back(0.02);
33   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].constraint_region_shape.
    dimensions.push_back(0.02);
34
35   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].constraint_region_
    orientation.w = 1.0;
36   goalA.motion_plan_request.goal_constraints.
    position_constraints[0].weight = 1.0;
37
38   goalA.motion_plan_request.goal_constraints.
    set_orientation_constraints_size(1);
39   goalA.motion_plan_request.goal_constraints.
    orientation_constraints[0].header.stamp = ros::Time::
    now();
40   goalA.motion_plan_request.goal_constraints.
    orientation_constraints[0].header.frame_id =
    "torso_lift_link";
41   goalA.motion_plan_request.goal_constraints.
    orientation_constraints[0].link_name =
    "r_wrist_roll_link";
42   goalA.motion_plan_request.goal_constraints.
```

```
        orientation_constraints[0].orientation.x = 0.0;
43    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].orientation.y = 0.0;
44    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].orientation.z = 0.0;
45    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].orientation.w = 1.0;
46
47    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].absolute_roll_tolerance =
        0.04;
48    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].absolute_pitch_tolerance =
        0.04;
49    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].absolute_yaw_tolerance =
        0.04;
50
51    goalA.motion_plan_request.goal_constraints.
        orientation_constraints[0].weight = 1.0;
52
53    if (nh.ok())
54    {
55        bool finished_within_time = false;
56        move_arm.sendGoal(goalA);
57        finished_within_time = move_arm.waitForResult(ros::
58            Duration(200.0));
59        if (!finished_within_time)
60        {
61            move_arm.cancelGoal();
62            ROS_INFO("Timed out achieving goal A");
63        }
64        else
65        {
66            actionlib::SimpleClientGoalState state = move_arm.
67                getState();
68            bool success = (state == actionlib::
69                SimpleClientGoalState::SUCCEEDED);
70            if (success)
```

```
68         ROS_INFO("Action finished: %s",state.toString().c_str());
69     else
70         ROS_INFO("Action failed: %s",state.toString().c_str());
71     }
72 }
73
74 ros::Duration(1.0).sleep();
75 goalA.motion_plan_request.goal_constraints.
    position_constraints[0].position.x = 0.60;
76 goalA.motion_plan_request.goal_constraints.
    position_constraints[0].position.y = -0.288;
77 goalA.motion_plan_request.goal_constraints.
    position_constraints[0].position.z = 0;
78
79 goalA.motion_plan_request.path_constraints.
    orientation_constraints.resize(1);
80 goalA.motion_plan_request.path_constraints.
    orientation_constraints[0].header.frame_id =
    "torso_lift_link";
81 goalA.motion_plan_request.path_constraints.
    orientation_constraints[0].header.stamp = ros::Time::
    now();
82 goalA.motion_plan_request.path_constraints.
    orientation_constraints[0].link_name =
    "r_wrist_roll_link";
83
84 goalA.motion_plan_request.path_constraints.
    orientation_constraints[0].orientation.x = 0.0;
85 goalA.motion_plan_request.path_constraints.
    orientation_constraints[0].orientation.y = 0.0;
86 goalA.motion_plan_request.path_constraints.
    orientation_constraints[0].orientation.z = 0.0;
87 goalA.motion_plan_request.path_constraints.
    orientation_constraints[0].orientation.w = 1.0;
88
89 goalA.motion_plan_request.path_constraints.
    orientation_constraints[0].type =
```

```
    motion_planning_msgs::OrientationConstraint::  
    HEADER_FRAME;  
90    goalA.motion_plan_request.path_constraints.  
        orientation_constraints[0].absolute_roll_tolerance =  
        0.2;  
91    goalA.motion_plan_request.path_constraints.  
        orientation_constraints[0].absolute_pitch_tolerance =  
        0.2;  
92    goalA.motion_plan_request.path_constraints.  
        orientation_constraints[0].absolute_yaw_tolerance =  
        M_PI;  
93  
94    if (nh.ok())  
95    {  
96        bool finished_within_time = false;  
97        move_arm.sendGoal(goalA);  
98        finished_within_time = move_arm.waitForResult(ros::  
            Duration(200.0));  
99        if (!finished_within_time)  
100        {  
101            move_arm.cancelGoal();  
102            ROS_INFO("Timed out achieving goal A");  
103        }  
104        else  
105        {  
106            actionlib::SimpleClientGoalState state = move_arm.  
                getState();  
107            bool success = (state == actionlib::  
                SimpleClientGoalState::SUCCEEDED);  
108            if (success)  
109                ROS_INFO("Action finished: %s", state.toString().  
                    c_str());  
110            else  
111                ROS_INFO("Action failed: %s", state.toString().c_str  
                    ());  
112        }  
113    }  
114    ros::shutdown();  
115 }
```

代码解释：首先需要指定保持夹持器竖直的约束。这对应于方向约束，`r_wrist_roll_link` 的偏转角没有限制，其转动角和俯仰角约束为 0.0。我们通过两条信息来指定这些约束。

(1) 代码第 79~87 行。指定名义位姿角。首先为 `torso_lift_link_frame` 中的 `r_wrist_roll_link` 指定名义位姿角。这些角用于保持夹持器竖直。

(2) 代码第 89 行。指定约束类型。方向约束类型可以是 `HEADER_FRAME` 或 `LINK_FRAME` (类型在消息本身内指定)。类型指定坐标系并计算该坐标系下考虑约束的各方向角。`type=HEADER_FRAME` 意味着这些角是在坐标系 `torso_lift_link` 内计算得到。`type=LINK_FRAME` 意味着这些角是在约束指定的目标连杆坐标系内计算得到的。

(3) 代码第 90~92 行。指定一个允许约束范围。首先使用消息中的公差域来确定允许约束范围。本质上是在指定一个名义位姿角和所选夹持器方向之间的允许位姿误差。本例中指定其转动角和俯仰角值为 0.0，因此指定了较小的转动角和俯仰角允许误差，而偏转角的允许公差较大。

## 6.2 运动规划的环境表示

本节主要介绍用于运动规划的环境的创建和使用。环境表示包含机器人、环境中可感知的物体、环境中障碍物。

### 6.2.1 基于自滤波数据构建碰撞地图

本小节介绍 PR2 的俯仰激光扫描仪、机器人的自滤波和碰撞地图构建。

#### 1. ROS 设置

首先需要确认设置环境：`export ROBOT=sim`。

然后运行 `rosmake` 编译 `pr2_arm_navigation_tutorials`。

#### 2. 模拟器中启动 PR2

```
roslaunch pr2_arm_navigation_tutorials pr2_floorobj_world.launch
```

#### 3. 启动带有预配置主题的 rviz

```
roslaunch pr2_arm_navigation_tutorials rviz_collision_tutorial_1.launch
```

在 rviz 中，可以看到与激光相关联的主题列表。

#### 4. 编写俯仰激光传感器处理文件

创建激光感知启动文件：pr2\_arm\_navigation\_tutorials/launch/laser-perception.launch，该文件启动了一系列节点，使用激光传感器数据构建碰撞地图。代码如下：

```
1 <launch>
2
3     <!-- set laser tilt rate -->
4     <node pkg="pr2_mechanism_controllers" type=
5         "send_periodic_cmd_srv.py" name=
6         "laser_tilt_controller_3dnav_params" args=
7         "laser_tilt_controller linear 3 .75 .25" />
8
9     <!-- convert tilt laser scan to pointcloud -->
10    <node pkg="laser_filters" type=
11        "scan_to_cloud_filter_chain" output="screen" name=
12        "scan_to_cloud_filter_chain_tilt_laser">
13            <remap from="scan" to="tilt_scan"/>
14            <remap from="cloud_filtered" to="tilt_scan_cloud"/>
15            <param name="target_frame" type="string" value=
16                "base_link"/>
17            <rosparam command="load" file="$(find
18                pr2_arm_navigation_tutorials)/config/shadow_filter.
19                yaml" />
20        </node>
21
22    <!-- need to clear known objects from scans -->
23    <node pkg="planning_environment" type=
24        "clear_known_objects" name="laser_clear_objects"
25        output="screen">
26        <remap from="cloud_in" to="tilt_scan_cloud" />
27        <remap from="cloud_out" to="tilt_scan_cloud_known" />
28        <param name="sensor_frame" type="string" value=
29            "laser_tilt_mount_link" />
30
31        <param name="fixed_frame" type="string" value=
32            "base_link" />
```

```
21      <param name="object_padd" type="double" value="0.02" />
22      <param name="object_scale" type="double" value="1.0" />
23
24  </node>
25
26  <node pkg="robot_self_filter" type="self_filter" respawn=
27      "true" name="tilt_laser_self_filter" output="screen">
28
29      <!-- The topic for the input cloud -->
30      <remap from="cloud_in" to="tilt_scan_cloud_known" />
31
32      <!-- The topic for the output cloud -->
33      <remap from="cloud_out" to="tilt_scan_cloud_filtered"
34          />
35
36      <!-- The frame of the sensor used to obtain the data
37          to be filtered; This parameter is optional. If it
38          is not specified, shadow points will be considered
39          outside -->
40
41      <param name="sensor_frame" type="string" value=
42          "laser_tilt_link" />
43
44      <param name="min_sensor_dist" type="double" value=".05
45          "/>
46      <param name="self_see_default_padding" type="double"
47          value=".01"/>
48      <param name="self_see_default_scale" type="double"
49          value="1.0"/>
50
51  </node>
52
53  <!-- assemble pointcloud into a full world view -->
54  <node pkg="laser_assembler" type="point_cloud_assembler"
55      output="screen" name="point_cloud_assembler">
56      <remap from="cloud" to="tilt_scan_cloud_filtered"/>
57      <param name="tf_cache_time_secs" type="double" value=
58          "10.0" />
59      <param name="tf_tolerance_secs" type="double" value=
60          "0.0" />
```

```
48     <param name="max_clouds" type="int" value="400" />
49     <param name="ignore_laser_skew" type="bool" value="true"
      " />
50     <param name="fixed_frame" type="string" value=
      "base_link" />
51   </node>
52
53   <!-- make snapshots whenever laser reaches the top or
      bottom of a nod -->
54   <node pkg="pr2_arm_navigation_perception" type=
      "pr2_laser_snapshotter" output="screen" name=
      "snapshotter">
55     <remap from="laser_scanner_signal" to=
      "laser_tilt_controller/laser_scanner_signal"/>
56     <remap from="build_cloud" to="point_cloud_assembler/
      build_cloud" />
57     <remap from="full_cloud" to="full_cloud_filtered" />
58   </node>
59
60   <!-- make point cloud snapshots into collision maps -->
61   <node pkg="collision_map" type=
      "collision_map_self_occ_node" name=
      "collision_map_self_occ_node" respawn="true" output=
      "screen">
62
63     <!-- The default padding to be added for the body parts
      the robot can see -->
64     <param name="self_see_default_padding" type="double"
      value="0.02" />
65
66     <!-- The default scaling to be added for the body parts
      the robot can see -->
67     <param name="self_see_default_scale" type="double"
      value="1.0" />
68
69     <!-- if someone asks for a stable map, publish the
      static map on the dynamic map topic and no longer
      publish dynamic maps -->
70     <param name="publish_static_over_dynamic_map" type=
```

```
    "bool" value="true" />
71   <param name="fixed_frame" type="string" value=
    "base_link" />
72
73   <!-- define a box of size 2x3x4 around (1.1, 0, 0) in
        the robot frame -->
74   <param name="robot_frame" type="string" value=
    "base_link" />
75
76   <param name="origin_x" type="double" value="1.1" />
77   <param name="origin_y" type="double" value="0.0" />
78   <param name="origin_z" type="double" value="0.0" />
79
80   <param name="dimension_x" type="double" value="1.0" />
81   <param name="dimension_y" type="double" value="1.5" />
82   <param name="dimension_z" type="double" value="2.0" />
83
84   <!-- set the resolution (1.0 cm) -->
85   <param name="resolution" type="double" value="0.01" />
86
87   <!-- cloud sources -->
88   <rosparam command="load" file="$(find
      pr2_arm_navigation_tutorials)/config/
      collision_map_sources.yaml" />
89   </node>
90
91 </launch>
```

代码解释：

(1) 启动文件。创建完 arm\_navigation\_tutorials 中文件之后，运行命令：

```
roslaunch pr2_arm_navigation_tutorials laser-perception.launch
```

(2) 代码第 3、4 行。设置激光传感器使其俯仰运动。这两行代码主要是配置参数。执行之后，可以看到激光传感器在运动且 rviz 中数据在更新。

(3) 代码第 6~12 行。转换激光数据为过滤阴影点云。这段代码有两个功能。首先，激光传感器的原始数据不是点云形式。scan\_to\_cloud\_filter\_chain 把原始数据转换为点云形式。其次，在滤波链上可以添加滤波处理原始数据中的一种噪声阴影点，这是由于激光撞击到场景的边缘之后返回一个边缘与其后的

物体造成的。用户可以通过调用阴影滤波来处理这些点，处理后的点云发布在 `tilt_scan_cloud` 主题。rviz 中的选项 `Tilt Scan Cloud` 允许用户可视化查看这些点云。

(4) 代码第 15~24 行。从激光数据中移除已知物体。该节点在本例中用不到，其主要对从场景中已经识别的物体点云进行滤波。

(5) 代码第 26~41 行。激光数据自动滤波。查看 rviz 中的主题 `tilt_scan_cloud`，可以看到机器人本体出现在扫描场景中。为了构建碰撞地图以在环境中避障，这里不希望手臂出现在障碍地图中。另外还有其他机制防止自碰撞。因此需要把机器人本体从场景中滤波出来：`robot_self_filt` 允许这样做。

如果设置 rviz 中第五项 (`Tilt scan self`) 为可用状态，用户可以看到机器人本体不再出现在场景中。

(6) 代码第 43~51 行。构建点云为复合点云。现在看到的是单个 2D 激光扫描，而不是 3D 点云。为了查看完整的 3D 世界视图，需要创建由很多点组合而成的点云。这项工作可以由装配器完成，其本质是创建点云数据的缓冲。当告诉节点进行装配工作时，节点将会生成一个包含特定时间段的复合点云。

(7) 代码第 53~58 行。获取复合点云快照。设置 rviz 中的第六项主题 (`Full Laser Cloud`) 为可用状态，即可看到激光快照。

(8) 代码第 60~90 行。合成激光碰撞地图。这段代码实际上转换处理过的激光数据为碰撞地图。节点 `collision_map_self_occ` 最终做了一系列滤波工作。它可以从数据中自滤波出机器人，对目前被机器人的部分本体遮挡的点，可以保存过去的扫描数据；或者根据用户需求获取环境的静态碰撞地图。该节点生成一个二值栅格地图，该地图可被传送至碰撞检测节点。

如果设置 rviz 中第七项 (`Collision Map`) 为可用状态，可以查看碰撞地图。

### 6.2.2 检测关节轨迹碰撞

本节介绍怎样检查一个输入的关节轨迹中是否有障碍、是否满足关节约束，如何使用环境服务器去检测机器人在给定轨迹下是否发生碰撞，其关节点是否满足几何和空间约束。

#### 1. 创建节点

之前已经创建了功能包，现在需要编写节点用于检查关节轨迹是否正确。添加下面代码到文件 `src/get_joint_trajectory_validity.cpp`。

```
1 #include <ros/ros.h>
2 #include <planning_environment_msgs/GetRobotState.h>
3 #include <planning_environment_msgs/
4             GetJointTrajectoryValidity.h>
5
6 int main(int argc, char **argv)
7 {
8     using namespace planning_environment_msgs;
9     ros::init (argc, argv, "get_trajectory_validity_test");
10    ros::NodeHandle rh;
11
12    std::string envServerName("environment_server_right_arm")
13        ;
14
15    ros::service::waitForService(envServerName + "/"
16                                get_trajectory_validity");
17
18    GetJointTrajectoryValidity::Request req;
19    GetJointTrajectoryValidity::Response res;
20    ros::ServiceClient check_trajectory_validity_client_=rh.
21                serviceClient<GetJointTrajectoryValidity>(
22                    envServerName + "/get_trajectory_validity");
23
24    ros::service::waitForService(envServerName + "/"
25                                get_robot_state");
26    ros::ServiceClient get_state_client_ = rh.serviceClient<
27        GetRobotState>(envServerName + "/get_robot_state");
28
29    GetRobotState::Request request;
30    GetRobotState::Response response;
31
32    if(get_state_client_.call(request, response))
33    {
34        req.robot_state = response.robot_state;
35    }
36    else
37    {
38        ROS_ERROR("Service call to get robot state failed on %s
39                  ",get_state_client_.getService().c_str());
```

```
32     }
33
34     req.trajectory.joint_names.push_back(
35         "r_shoulder_pan_joint");
36     req.trajectory.points.resize(100);
37     for(unsigned int i=0; i < 100; i++)
38     {
39         req.trajectory.points[i].positions.push_back(-(3*M_PI*i
40             /4.0)/100.0);
41     }
42
43     req.check_collisions = true;
44
45     if(check_trajectory_validity_client_.call(req,res))
46     {
47         if(res.error_code.val == res.error_code.SUCCESS)
48             ROS_INFO("Requested trajectory is not in collision");
49         else
50             ROS_INFO("Requested trajectory is in collision. Error
51             code: %d", res.error_code.val);
52     }
53     else
54     {
55         ROS_ERROR("Service call to check trajectory validity
56             failed %s",check_trajectory_validity_client_.
57             getService().c_str());
58     }
59
60     ros::shutdown();
61 }
```

## 2. 创建服务需求

针对 GetJointTrajectoryValidity 的服务请求如下：

- ① 希望检查关节轨迹上一系列位置点。
- ② 需要执行的检查的类型。
- ③ 包含机器人状态。

(1) 代码第 15~20 行。设置机器人状态。前面设置完成单个关节的目标轨迹。现在需要指定机器人其他关节的静态位置。服务需求检查机器人状态以使得它包含机器人上的每个关节值。下面利用 `GetRobotState` 服务来获取机器人当前状态并且在服务需求中填充 `robot_state` 域。

(2) 代码第 22~27 行。设置关节轨迹。首先实现用于单个关节的关节轨迹：`r_arm_shoulder_pan_joint`。

(3) 代码第 31~39 行。服务响应。在调用服务之后，需要检测服务响应中的错误，以确定轨迹检查是否成功。服务响应返回两个代码错误：一个是全局 `error_code`，它指出是否有点违反了要求检查；另一个是向量形式的代码错误，用于编码每一个单独的点的状态。只有在 `req.flag` 中包含了 `CHECK_FULL_TRAJECTORY` 标记时，向量 `error_code` 才会被完全填充。

(4) 代码第 41 行。检测类型，在服务需求中指定输入轨迹检测类型。有四种类型检测可以使用：

- ① 碰撞检测：`request.check_collisions = true`。
- ② 关节限制检测：`request.check_joint_limits = true`。
- ③ 目标约束检测：`request.check_goal_constraints = true`。
- ④ 路径约束检测：`request.check_path_constraints = true`。

这也是允许用户查看是否整个轨迹已经检查过的标记。缺省情况下，只要发现其中一个点不满足约束，服务调用可以返回结果。

全轨迹检测：`request.check_full_trajectory = true`。

如果所有检测通过，全局错误代码将会显示为“成功”。需要注意的是所有的检查只是针对单独的点，在点之间没有进行插值计算。

### 3. 编译节点

用户现在已经有了功能包和源文件，下面可以编译这些文件。首先需要添加文件 `src/get_joint_trajectory_validity.cpp` 到 `CMakeLists.txt` 中。打开 `CMakeLists.txt`，然后在文件尾添加：

```
1 rosbuild_add_executable(get_joint_trajectory_validity src/
    get_joint_trajectory_validity.cpp)
```

执行完毕后，可以运行：`make`。

#### 4. 在模拟器中运行

启动 rviz:

```
roslaunch pr2_arm_navigation_tutorials rviz_collision_tutorial_4.launch
```

运行 `get_joint_trajectory_validity`, 然后确保 rviz 的第十个分量 Collision Pose 被设置为可用状态:

```
./bin/get_joint_trajectory_validity
```

运行成功后, 结果如下:

```
[ INFO] 33.179000000: Requested trajectory is in collision.  
Error code: -23
```

rviz 中运行结果如图 6.3 所示。Collision Pose 显示表明最后一个点将会导致碰撞。由于 `GetJointTrajectoryValidity` 中没有选择 `check_full_trajectory` 选项, 环境服务器在发现有无效姿态之后停止了检查。因此, 显示的位姿为在发现导致碰撞之后的第一个位姿。

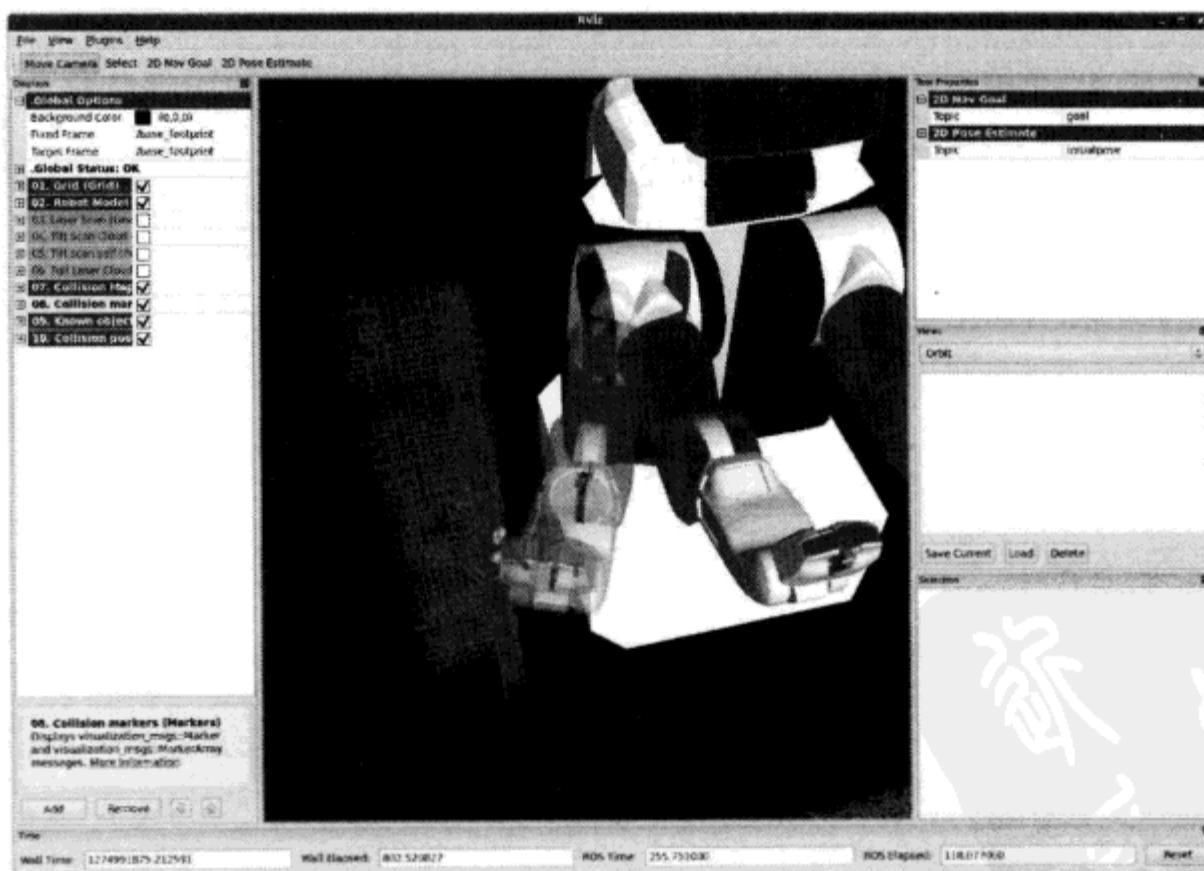


图 6.3 检测关节轨迹碰撞

#### 6.2.3 给定机器人状态下的碰撞检测

本小节讲述怎样利用环境服务器和激光碰撞地图数据, 检测机器人状态是否

会发生碰撞，是否满足关节的几何约束和空间约束。这里将使用一段简单的程序来调用环境服务器对机器人进行碰撞检测，并反馈环境服务器运行状态。

## 1. ROS 设置

首先需要做的是运行运动规划环境功能包，这里使用 `roscreate-pkg` 创建带有依赖项 `pr2_arm_navigation_tutorials` 的功能包：

```
roscreate-pkg arm_navigation_tutorials pr2_arm_navigation_
tutorials
```

改变当前目录到刚刚创建的功能包：

```
roscd arm_navigation_tutorials
```

设置环境服务器：`export ROBOT=sim`。

## 2. 创建节点

将下面代码放入文件 `src/get_state_validity.cpp` 中。

```
1 #include <ros/ros.h>
2 #include <planning_environment_msgs/GetStateValidity.h>
3
4 int main(int argc, char **argv){
5     ros::init (argc, argv,"get_state_validity_test");
6     ros::NodeHandle nh;
7
8     ros::service::waitForService("environment_server/
9         get_state_validity");
10    ros::ServiceClient check_state_validity_client_ = rh.
11        serviceClient<planning_environment_msgs::
12            GetStateValidity>("environment_server/
13            get_state_validity");
14
15    planning_environment_msgs::GetStateValidity::Request req;
16    planning_environment_msgs::GetStateValidity::Response
17        res;
18
19    req.robot_state.joint_state.name.push_back(
20        "r_shoulder_pan_joint");
21    req.robot_state.joint_state.name.push_back(
22        "r_shoulder_lift_joint");
```

```

16     req.robot_state.joint_state.name.push_back(
17         "r_upper_arm_roll_joint");
18     req.robot_state.joint_state.name.push_back(
19         "r_elbow_flex_joint");
20     req.robot_state.joint_state.name.push_back(
21         "r_forearm_roll_joint");
22     req.robot_state.joint_state.name.push_back(
23         "r_wrist_flex_joint");
24     req.robot_state.joint_state.name.push_back(
25         "r_wrist_roll_joint");
26     req.robot_state.joint_state.position.resize(7,0.0);
27
28     req.robot_state.joint_state.position[0] = 0.0;
29
30     req.robot_state.joint_state.header.stamp = ros::Time::now()
31         ();
32     req.check_collisions = true;
33     if(check_state_validity_client_.call(req,res))
34     {
35         if(res.error_code.val == res.error_code.SUCCESS)
36             ROS_INFO("Requested state is not in collision");
37         else
38             ROS_INFO("Requested state is in collision. Error code
39             : %d",res.error_code.val);
40     }
41     else
42     {
43         ROS_ERROR("Service call to check state validity failed
44             %s",check_state_validity_client_.getService().c_str()
45             ());
46         return false;
47     }
48     ros::shutdown();
49 }
```

### 3. 创建服务需求

针对 `GetStateValidity` 的服务需求如下：

- ① 需要检测的关节轨迹点集合。

② 需要执行的检测类型。

(1) 代码第 14~21 行：设置关节状态。首先对右臂进行设置，设置其关节状态及其目标位置。

(2) 代码第 26 行：检测类型。在服务需求中指定输入检测类型，可选类型如下：

- ① 碰撞检测：`request.check_collisions = true.`
- ② 关节限制检测：`request.check_joint_limits = true.`
- ③ 目标约束检测：`request.check_goal_constraints = true.`
- ④ 路径约束检测：`request.check_path_constraints = true.`

#### 4. 编译节点

用户现在已经有了功能包和源文件，下面可以编译这些文件。首先需要添加文件 `src/get_state_validity.cpp` 到 `CMakeLists.txt` 中。打开 `CMakeLists.txt`，然后在文件尾添加：

```
rosbuild_add_executable(get_state_validity src/get_state_
    validity.cpp)
```

执行完毕后，可以运行：`rosmake`。

#### 5. 模拟器中启动 PR2

执行命令：

```
roslaunch pr2_arm_navigation_tutorials pr2_floorobj_world.
    launch
```

#### 6. 启动带有预配置主题的 rviz

启动 `rviz`，可以查看正在使用的数据主题：

```
roslaunch pr2_arm_navigation_tutorials rviz_collision_tutorial_
    2.launch
```

#### 7. 启动激光感知管道

下一步需要启动俯仰激光传感器及基于滤波的碰撞地图管道：

```
roslaunch pr2_arm_navigation_tutorials laser-perception.launch
```

激光传感器将会启动节点，如果用户选中 `rviz` 窗口的第六项，将会看到一系列绿色方块用以表示碰撞地图 6.4。

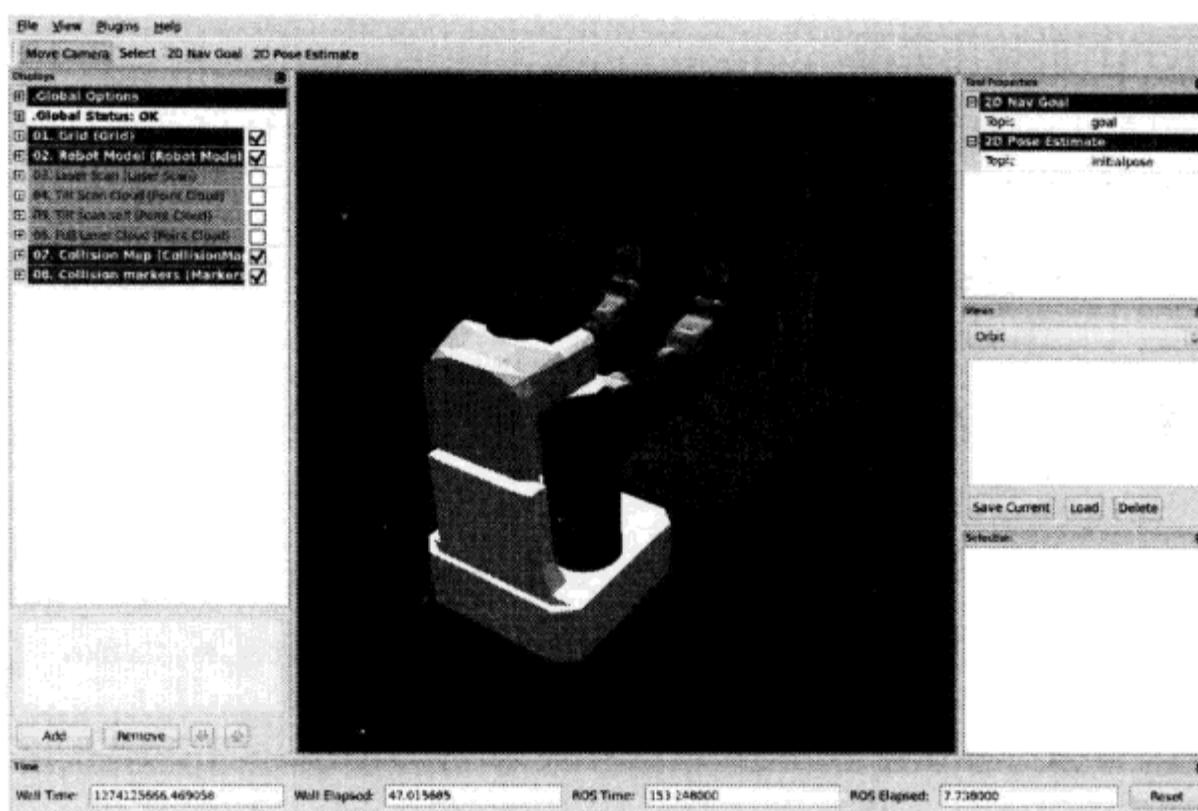


图 6.4 给定机器人状态下的碰撞检测地图

### 1) 设置环境服务器

```
roslaunch pr2_arm_navigation_actions environment_server.launch
```

结果如下：

```
process[environment_server-1]: started with pid [16573]
[ INFO] 1417.882999973: Configuring action for 'right_arm'
[ INFO] 1417.882999973: Waiting for robot state ...
[ INFO] 1417.882999973: Waiting for joint state ...
[ INFO] 1417.882999973: Pose ignored
[ WARN] 1417.882999973: Message from [/collision_map_self_occ_node] has a non-fully-qualified frame_id [base_link].
Resolved locally to [/base_link]. This is will likely not work
in multi-robot systems. This message will only print once.
[ INFO] 1417.883999973: Ignoring joint 'r_gripper_joint'
[ INFO] 1417.883999973: Ignoring joint 'l_gripper_joint'
[ INFO] 1417.933999973: Robot state received!
[ INFO] 1417.934999973: Waiting for map ...
[ INFO] 1419.052999973: Map received!
[ INFO] 1419.070999973: Found collisions will be displayed as
visualization markers
[ INFO] 1419.102999973: Environment monitor started
```

## 2) 在无障碍环境中运行程序

选中 rviz 中的第九个选项 **State Validity**，这将使得界面中出现一个与原来机器人类似的机器人。复制的机器人将显示需要检查的位置。

运行刚才创建的可执行文件：`./bin/get_state_validity`。运行成功后，将会看到如下结果：

```
[ INFO] 0.000000000: Requested state is not in collision
```

在 rviz 中，将会看到如图 6.5 所示结果。

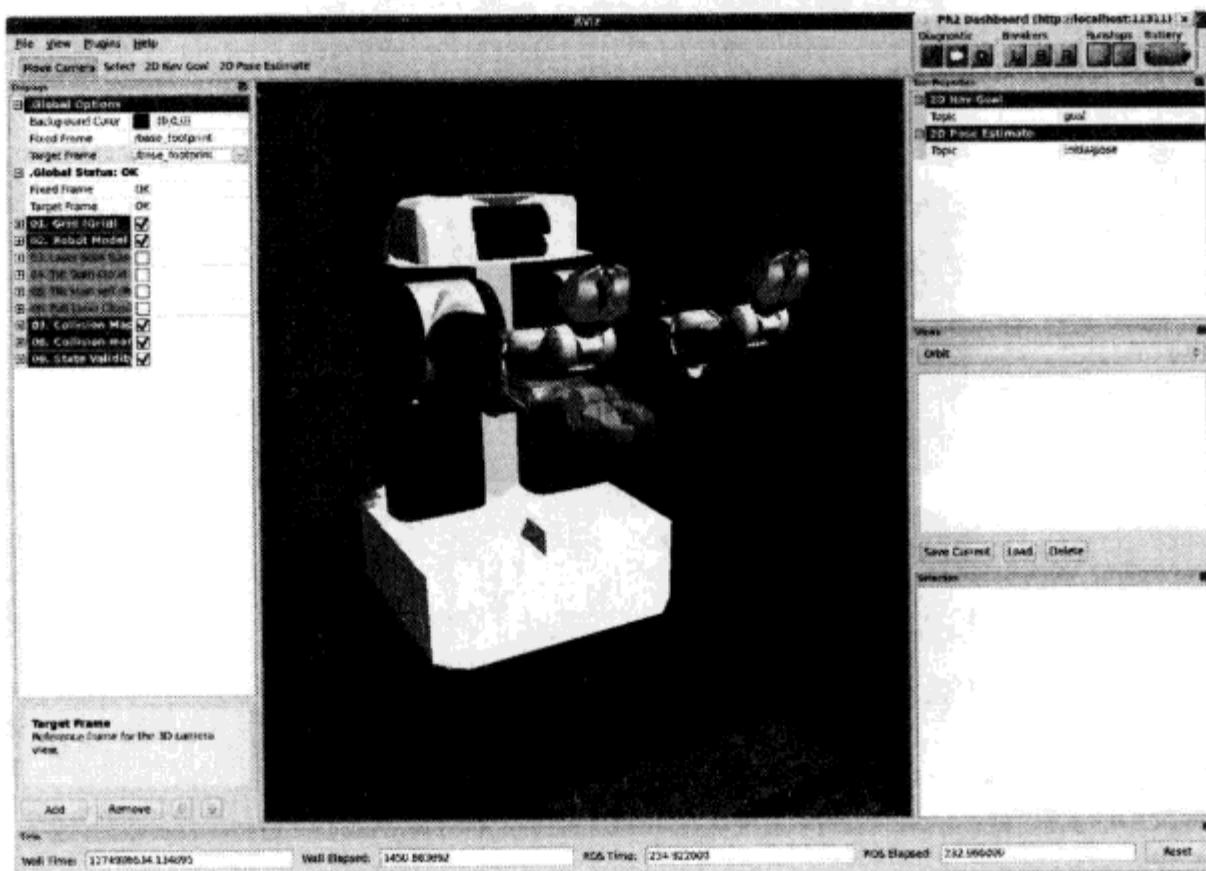


图 6.5 无障碍环境下的碰撞检测

复制的机器人显示机器人位置：所有关节赋值为 0，其他机器人部件位置不变。这个状态没有碰撞，并满足所有关节的约束条件，环境服务器给予确定。

为了查看一个机器人状态是否会发生碰撞，在文件 `get_state_validity.cpp` 中把下面代码：

```
23 req.robot_state.joint_state.position[0] = 0.0;
```

替换为

```
23 req.robot_state.joint_state.position[0] = -0.55;
```

这将告诉环境服务器检查机器人某个状态情况，第一个关节赋值为 -0.55，其他所有值为 0。

在重新运行 `get_state_validity` 之前，先确认第八个 rviz 分量值 `Collision markers` 设置为可以使用。重新编译并运行 `get_state_validity`。如果运行成功，结果如下：

```
[ INFO] 0.000000000: Requested state is in collision.  
Error code: -23
```

rviz 中结果如图 6.6 所示。



图 6.6 与杆的碰撞检测

环境服务器将会报告给定的状态是有碰撞的。但是它没有给出会发生什么样的碰撞。为了确定碰撞的状态，必须求助于 rviz。当环境服务器被要求检测一个状态是否有碰撞发生时，它会广播标记。如果用户已经设置了 `Collision Markers` 选项为可用状态，程序运行时，在碰撞地图中将会看到一系列黄色球体，在实际环境中，这是一个杆状物体，是那些碰撞物体形成的。这些点表示了碰撞检测后形成的碰撞点。要查看这些点，在 rviz 工具条上选择 `Select` 并选中一个或多个点。在 rviz 窗口中将会看到一些列标记，这些标记命名为 `r_forearm_link+points`，这表明该位置是右前臂与碰撞地图中的点碰撞形成的。

与环境中的物体碰撞检测，只是环境服务器能够做的检测的一种。下面来看另一种碰撞。在文件 `get_state_validity.cpp` 中，编辑：

```
23 req.robot_state.joint_state.position[0] = 0.4;
```

重新编译并运行 `get_state_validity`。如果运行成功，结果如下：

```
[ INFO] 0.000000000: Requested state is in collision.  
Error code: -23
```

此时可以看到在机器人左臂以黄色球表示的碰撞，效果如图 6.7 所示。

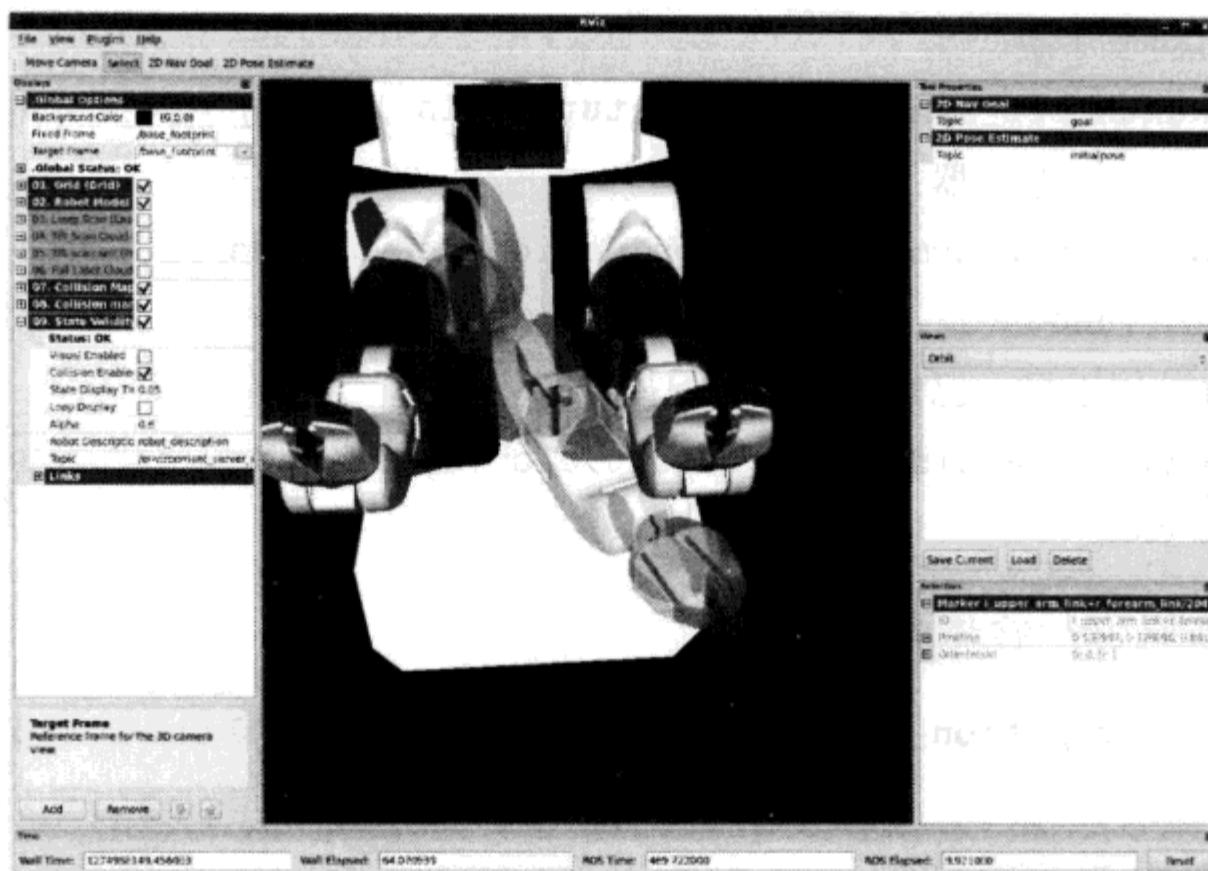


图 6.7 手臂自碰撞

选中一个碰撞，上面可以显示为左臂 + 右臂的链接。这是一种自碰撞，机器人应该可以避免这种情况。环境服务器有能力检测出这种碰撞。

#### 6.2.4 添加已知点到运动规划环境

本小节介绍如何将已知物体插入运动规划环境中并使用。这些物体可以直接以球的形式插入，也可以采用三角网格化的方法在环境中以特定位姿出现，这样它们可以用于碰撞检测。这里讲述如何增加物体、移动物体、删除物体、检测碰撞、根据激光数据滤波物体点。

首先来看前面例子中的屏幕截图 6.6。可以看到，机器人手臂的当前位置，遮挡了一个杆。这一节中，将会把杆以圆柱体形式插入到碰撞地图中，这意味着运动规划器可以发现一条用于避开杆的路径。这根杆将会变成已知物体。

##### 1. 设置

###### (1) 打开模拟器：

```
roslaunch pr2_arm_navigation_tutorials pr2_fLOORobj_world.launch
```

(2) 打开 rviz:

```
roslaunch pr2_arm_navigation_tutorials rviz_collision_tutorial_3.launch
```

(3) 启动感知器:

```
roslaunch pr2_arm_navigation_tutorials laser-perception.launch
```

(4) 启动环境服务器:

```
roslaunch pr2_arm_navigation_actions environment_server_right-arm.launch
```

(5) 启动用于显示已知模块的标记:

```
roslaunch planning_environment display_planner_collision_model.launch
```

## 2. 增加已知物体到环境中

在 arm\_navigation\_tutorials/src 下创建文件 addCylinder.cpp，并添加如下代码：

```
1 #include <ros/ros.h>
2
3 #include <mapping_msgs/CollisionObject.h>
4 #include <geometric_shapes_msgs/Shape.h>
5
6 int main(int argc, char** argv) {
7
8     ros::init(argc, argv, "addCylinder");
9
10    ros::NodeHandle nh;
11
12    ros::Publisher object_in_map_pub_;
13    object_in_map_pub_ = nh.advertise<mapping_msgs::CollisionObject>("collision_object", 10);
14
15    ros::Duration(2.0).sleep();
16
17    mapping_msgs::CollisionObject cylinder_object;
18    cylinder_object.id = "pole";
```

```
19     cylinder_object.operation.operation = mapping_msgs::  
20         CollisionObjectOperation::ADD;  
21     cylinder_object.header.frame_id = "base_link";  
22     cylinder_object.header.stamp = ros::Time::now();  
23     geometric_shapes_msgs::Shape object;  
24     object.type = geometric_shapes_msgs::Shape::CYLINDER;  
25     object.dimensions.resize(2);  
26     object.dimensions[0] = .1;  
27     object.dimensions[1] = .75;  
28     geometry_msgs::Pose pose;  
29     pose.position.x = .6;  
30     pose.position.y = -.6;  
31     pose.position.z = .375;  
32     pose.orientation.x = 0;  
33     pose.orientation.y = 0;  
34     pose.orientation.z = 0;  
35     pose.orientation.w = 1;  
36     cylinder_object.shapes.push_back(object);  
37     cylinder_object.poses.push_back(pose);  
38  
39     cylinder_object.id = "pole";  
40     object_in_map_pub_.publish(cylinder_object);  
41  
42     ROS_INFO("Should have published");  
43  
44     ros::Duration(2.0).sleep();  
45  
46 }
```

代码解释：

(1) 代码第 12、13 行。创建主题 `collision_object` 的标准发布者。系统有一系列不同进程用于处理已知物体。在任何进程中一个已知物体被放置到环境中时，所有进程都被通知有碰撞物体加入。

(2) 代码第 17~21 行。该部分代码创建物体，并指派其唯一的名称，告知用户将进行的操作是 ADD，这将增加物体到环境中或者更新物体的位置（当名称为 `pole` 的物体已经存在）。这里同时也增加头信息，这些头信息将允许运动规划环境去确定表示位姿信息的坐标系。注意 `id` 是很重要的：它允许物体可以被单独

移除、移动或者改变，在碰撞中显示为识别号码。

(3) 代码第 22~26 行。这部分代码指出了希望放置在环境中的形状的类型，这里是圆柱。其中第一维是宽度，第二维是高度。

(4) 代码第 27~34 行。创建物体位姿，该位姿应该在头文件指定的坐标系下。

(5) 代码第 35、36 行。增加形状和位姿信息到物体向量中。一个物体可以有任意多维的形状和位姿。

在 `arm_navigation_tutorials/CMakeLists.txt` 中添加如下代码：

```
rosbuild_add_executable(add_cylinder src/addCylinder.cpp)
```

编译并执行程序。运行成功后，结果类似下面：

```
[ INFO] [0.000000]: Added 1 object to namespace pole in
collision space
```

`rviz` 中结果如图 6.8 所示。可以注意到靠近已知物体的方块消失了。它们被位于文件 `laser-perception.launch` 中的节点 `clear_known_objects` 滤波过滤掉了。被过滤掉的原因是数据有噪声。如果物体的位置精确地知道，那么就可以避免噪声。这在抓取以及操作物体时尤其有用。

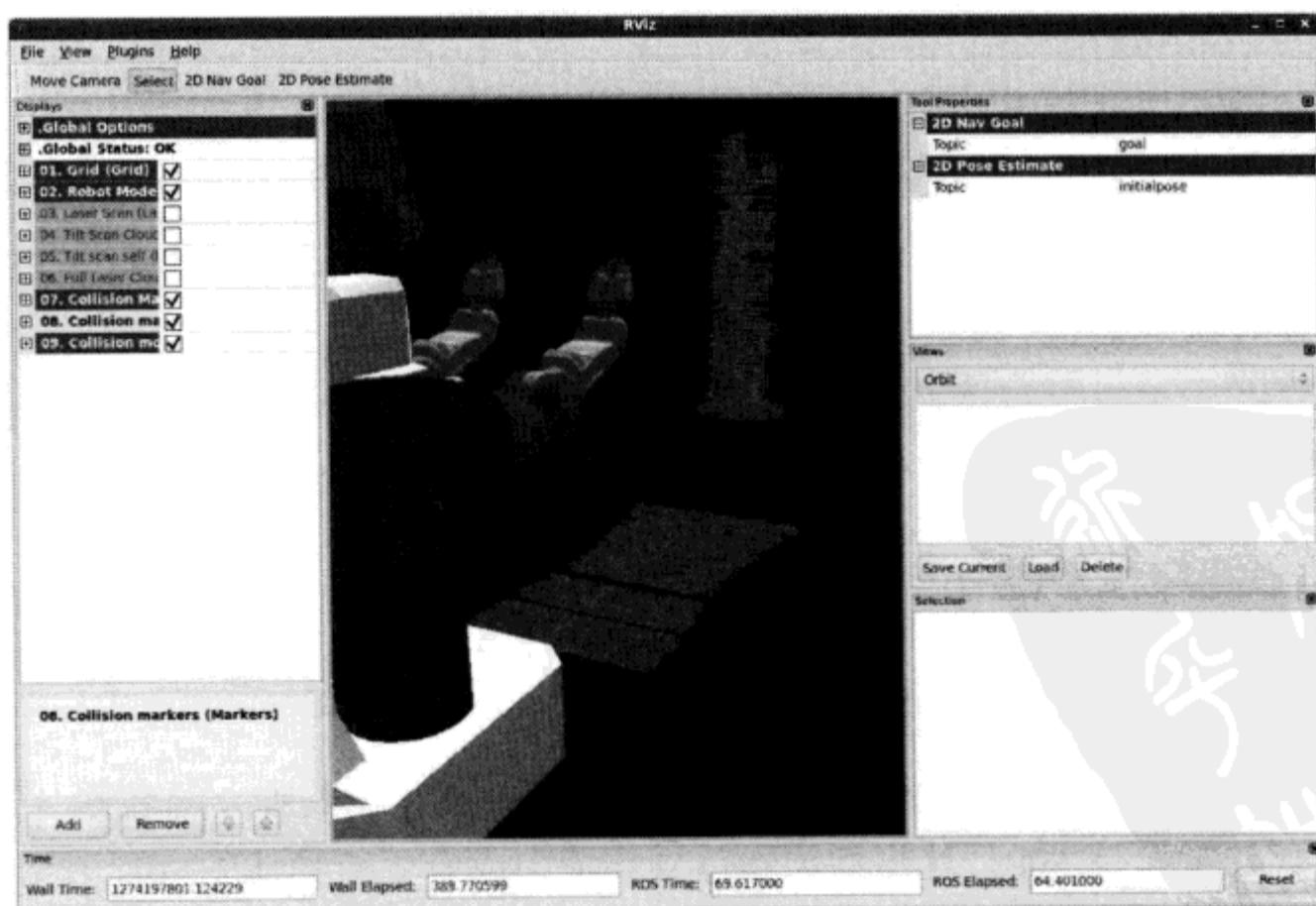


图 6.8 增加已知点到运动规划环境中

### 3. 更新已知物体

当使用 ADD 操作物体，环境中物体很容易更新。在文件 `addCylinder.cpp` 中，改变下面代码：

```
26 object.dimensions[1] = 1.5;

30 pose.position.z = .75;
```

重新编译并运行 `addCylinder`。其结果如图 6.9 所示。这里仍然只有一根杆，如果我们修改名称为 `pole2`，那么会有两根不同的杆出现。

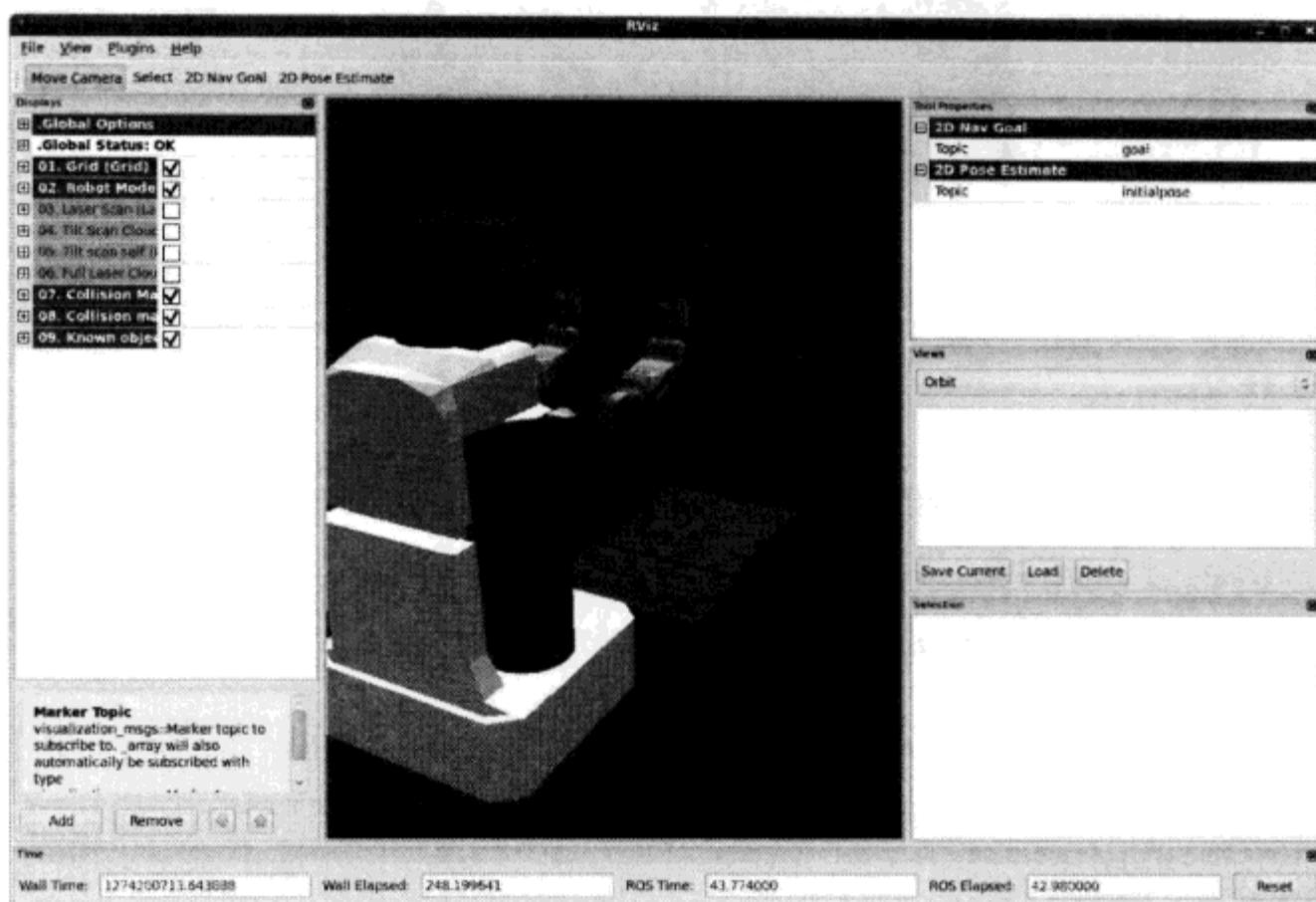


图 6.9 更新已知物体

### 4. 对已知物体进行碰撞检测

从用户的角度看，对已知物体进行碰撞检测与前面使用环境服务器检测碰撞一样。但碰撞标记报告可以给出一些额外的有用信息。编辑文件 `get_state_validity.cpp` 第 23 行：

```
23 req.robot_state.joint_state.position[0] = -0.55;
```

重新编译并运行 `get_state_validity`，其结果为出错，错误代码 23。在 `rviz` 中可以看到用于环境服务器碰撞标记的黄色小球，如果选中这些小球，可以看到其名称为一个机器人连杆及 `+pole`，就像在屏幕截图 6.10 中显示的那样。



图 6.10 对已知物体进行碰撞检测

## 5. 移除物体

已知物体可以很容易地删除。编辑文件 `addCylinder.cpp` 的第 19 行：

```
19 cylinder_object.operation.operation = mapping_msgs::  
    CollisionObjectOperation::REMOVE;
```

消息的其余部分可以不用理会。重新编译并运行，可以看到已知物体消失了，碰撞地图中的点逐渐返回。也可以替换上面代码中的 `REMOVE` 为 `all`，这样可以移除所有物体。

### 6.2.5 添加物体到机器人本体

添加物体到机器人本体的意思是当机器人本体移动时，该物体随着机器人本体一起移动。这项功能需要运动规划器和轨迹检测器一起来处理机器人及其所抓取物体与环境之间的避障情形。

机器人可以使用夹持器在环境中抓取物体。当机器人成功抓取物体后，该物体随机器人在环境中一起运动，以避免被抓取物体与环境中其他物体碰撞，导致被抓取物体掉落。更进一步，设计者可能希望将其他物体附加到机器人本体上，如外部传感器和装饰物等。这些物体随机器人一起进行碰撞检测。

本节中，物体被假设为刚体，附加到机器人本体的某些特定连杆上。物体与

机器人本体之间的变换被更新并固定。这里要介绍怎样将物体附加到机器人本体上以及附属物体与非附属物体之间的转换。

### 1. 添加附属物体到机器人夹持器

在 arm\_navigation\_tutorials 功能包内，编辑 attachCylinder.cpp 文件：

```
1 #include <ros/ros.h>
2 #include <mapping_msgs/AttachedCollisionObject.h>
3 #include <geometric_shapes_msgs/Shape.h>
4
5 int main(int argc, char** argv) {
6
7     ros::init(argc, argv, "attachCylinder");
8
9     ros::NodeHandle nh;
10
11    ros::Publisher att_object_in_map_pub_;
12    att_object_in_map_pub_ = nh.advertise<mapping_msgs::
13                                AttachedCollisionObject>("attached_collision_object",
14                                10);
15
16    mapping_msgs::AttachedCollisionObject att_object;
17    att_object.link_name = "r_gripper_r_finger_tip_link";
18    att_object.touch_links.push_back("r_gripper_palm_link");
19    att_object.touch_links.push_back("r_gripper_r_finger_link
20                                    ");
21
22    att_object.object.id = "attached_pole";
23    att_object.object.operation.operation = mapping_msgs::
24                                CollisionObjectOperation::ADD;
25    att_object.object.header.frame_id =
26        "r_gripper_r_finger_tip_link";
27    att_object.object.header.stamp = ros::Time::now();
```

```

26     geometric_shapes_msgs::Shape object;
27     object.type = geometric_shapes_msgs::Shape::CYLINDER;
28     object.dimensions.resize(2);
29     object.dimensions[0] = .02;
30     object.dimensions[1] = 1.2;
31     geometry_msgs::Pose pose;
32     pose.position.x = 0.0;
33     pose.position.y = 0.0;
34     pose.position.z = 0.0;
35     pose.orientation.x = 0;
36     pose.orientation.y = 0;
37     pose.orientation.z = 0;
38     pose.orientation.w = 1;
39     att_object.object.shapes.push_back(object);
40     att_object.object.poses.push_back(pose);
41
42     att_object_in_map_pub_.publish(att_object);
43
44     ROS_INFO("Should have published");
45
46     ros::Duration(2.0).sleep();
47
48     ros::shutdown();

```

代码解释：

(1) 代码第 11 行。为附属碰撞物体创建发布者。注意，其类型和主题不同于非附属物体。

(2) 代码第 15~20 行。这段代码声明 `AttachedObject` 并指定两个域：`link_name` 和 `touch_links`。`AttachedObject` 声明附属连接，`touch_links` 指明被附属物体的自碰撞行为，即附属物体允许与被附属物体发生碰撞。

(3) 代码第 22~38 行。被附属物体实际上包含了一个已知物体，因此该段代码与附属已知物体情形相似。不同之处在于我们指定了物体附属的位置。

运行代码。添加下面代码到文件 `arm_navigation_tutorials/CMakeLists.txt`。

```
rosbuild_add_executable(attach_cylinder src/attachCylinder.cpp)
```

在 rviz 中，运行结果如图 6.11 所示。图中杆被置于右手夹持器的中间。棕色区域表明该物体已经被附属于机器人，绿色区域为静态已知物体。

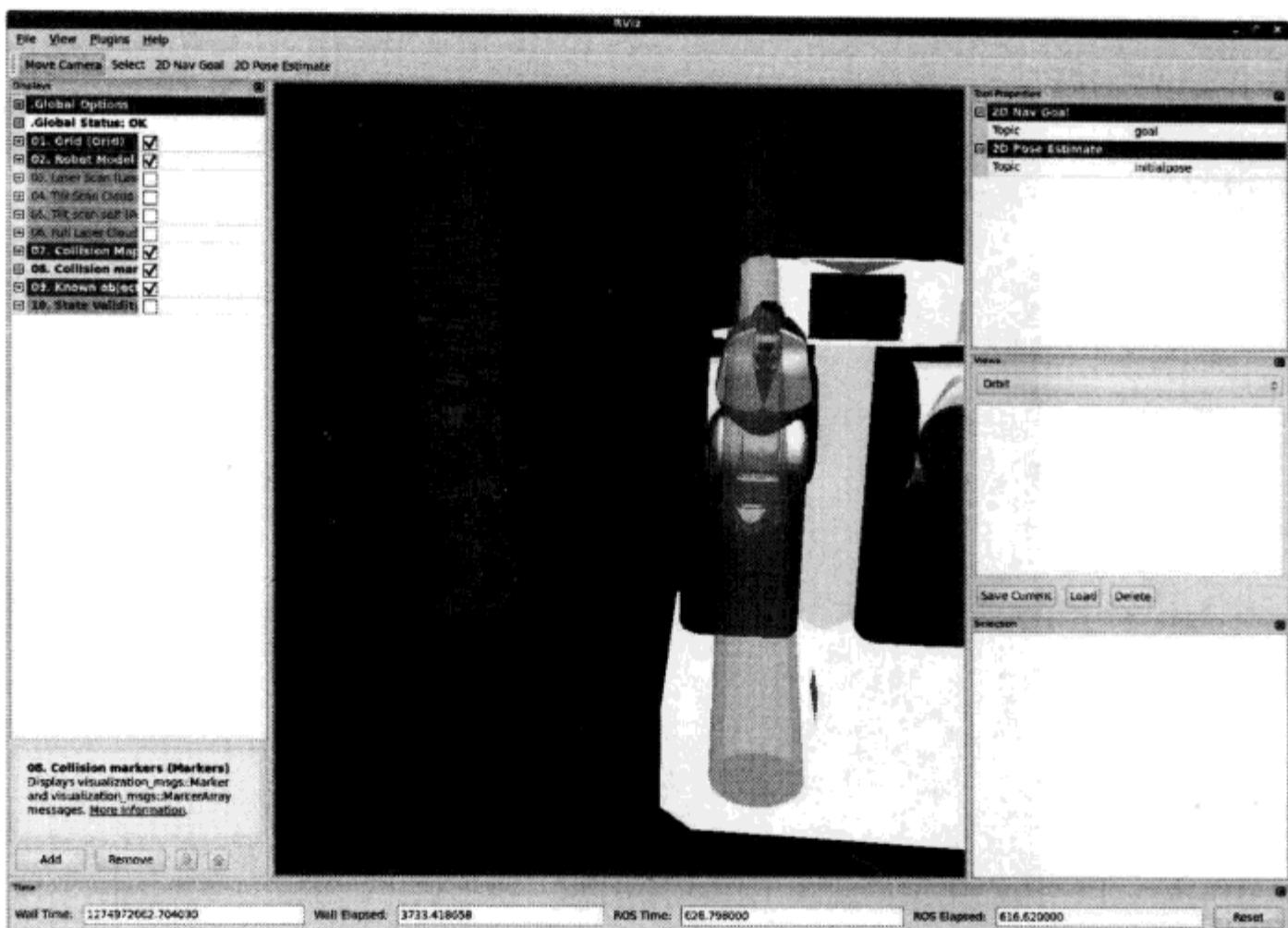


图 6.11 添加附属物体到机器人夹持器

## 2. 对附属物体进行状态检测

由于可视化标记每秒钟更新一次，手臂移动时，附属物体跟随手臂运动将会稍微有些延迟。下面将对附属物体进行碰撞检测。

编辑 pr2\_arm\_navigation\_tutorials/src 中的 get\_state\_validity，将下面一行注释：

```
//these set whatever non-zero joint angle values are desired
```

替换为：

```
1 req.robot_state.joint_state.position[3] = -1.2;
2 req.robot_state.joint_state.position[5] = -0.15;
```

并确认其他行代码没有将 joint\_states 设置为非零状态。重新编译 get\_state\_validity 并使得 rviz 中第十项置于可用状态：marked State Validity。在 rviz 中可以看到一个复制的机器人。然后可以运行 get\_state\_validity。运行结果如图 6.12 所示。

复制的机器人将显示在指定位置，但是附属物体并没有在该位置显示。现在

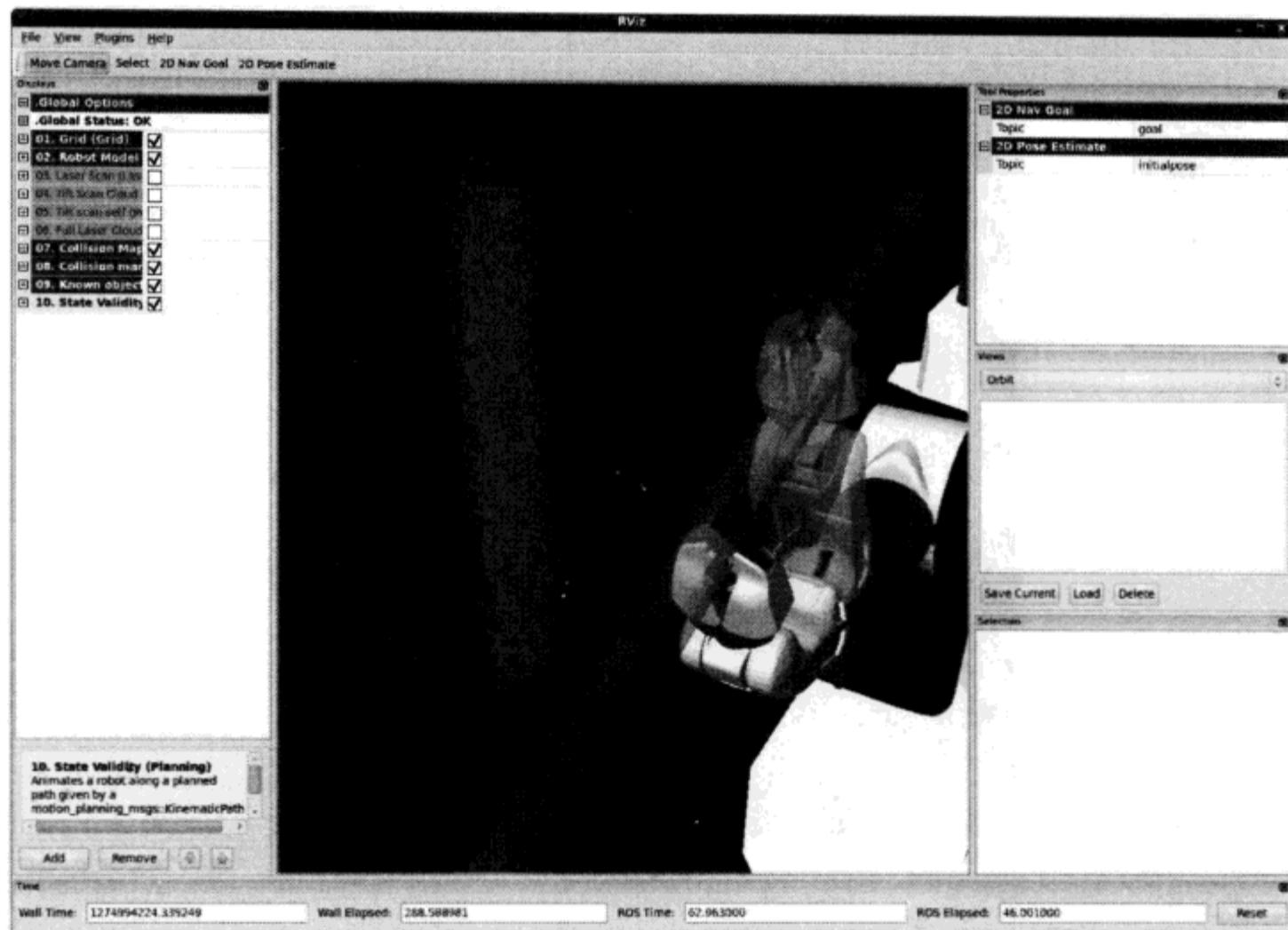


图 6.12 对附属物体进行状态检测

可以返回到 `attachCylinder` 并注释掉第 20 行：

```
20 //att_object.touch_links.push_back(
    "r_gripper_l_finger_tip_link");
```

重新编译并运行 `attached_cylinder`。正像已知物体的情况一样，发布相同名称的附属物体将会替换已有物体。重新运行 `get_state_validity`，将会报告有碰撞发生。

可以注意到是 `attached_pole` 和 `r_finger_r_finger_tip_link` 发生碰撞，碰撞位置恰好为指尖连接位置：由 `pr2_arm_navigation_actions/config/environment_server_padding.yaml` 文件指定的 1cm 大小的填充区域。

附属物体不仅与机器人产生碰撞，同时还可以与环境中点或者静态物体产生碰撞。将刚才注释掉的代码恢复，重新编译并运行。在 `get_state_validity.cpp` 设置关节位置的两行代码后面添加如下代码：

```
1 req.robot_state.joint_state.position[6] = 1.57;
```

重新编译并运行 `get_state_validity`，将会报告碰撞，并显示如图 6.13 所示结果。注意到在 `attached_pole` 和点之间，如果给定碰撞物为圆柱模型，随



图 6.13 附属物体与静态物体碰撞

着手腕旋转，附属的杆将直接穿过静态杆。

### 3. 附属物体与静态物体之间转换

一般情况下，机器人拿起物体之后，将会把物体放在某处。一种做法是记录附属物体的位置，然后从整个环境中删除该物体，然后增加该物体为静态碰撞物体。为简化流程，这里介绍一种只需要编辑单个物体的方法。在 `attach_cylinder.cpp` 文件中改变第 24 行为

```
24 att_object.object.operation.operation = mapping_msgs::  
    CollisionObjectOperation::DETACH_AND_ADD_AS_OBJECT;
```

用户不需要修改其他消息，将物体作为静态障碍物插入环境的当前位置。重新编译运行 `attach_cylinder`，结果如图 6.14 所示。

注意到物体变为绿色，并且收缩变小。因为附属物体是需要填充的，机器人需要缓冲来避免环境与附属物体相撞。附属物体被填充是在 `environment_server_padding.yaml` 中设置的。再次重新运行 `get_state_validity`。将会报告在杆和手臂当前位置之间有碰撞。

静态物体很容易被附加到机器人上。由于必须指定被附属的连接件，用户



图 6.14 附属物体与静态物体之间转换

此处必须使用 `AttachedCollisionObject`。将 `attach_cylinder.cpp` 第 24 行替换为：

```
24 att_object.object.operation.operation = mapping_msgs::  
    CollisionObjectOperation::ATTACH_AND_REMOVE_AS_OBJECT;
```

重新编译运行 `attached_cylinder`, 绿色物体将会再次被棕色填充的圆柱代替。

## 6.3 用于 PR2 机器人手臂的运动学

### 6.3.1 从 PR2 运动学开始

`kinematics` 功能包集主要是展示如何计算 PR2 运动学正解和逆解。同时，它也与 `planning_environment` 配合用于计算无碰撞运动学逆解。

#### 1. 运动学消息和服务

`kinematics` 功能包集使用来自 `kinematics_msgs` 功能包的消息。用户尤其要注意如下服务：

- (1) `kinematics_msgs/GetPositionFK`。用于获取运动学正解的基本服务。
- (2) `kinematics_msgs/GetPositionIK`。用于获取运动学逆解的基本服务。
- (3) `kinematics_msgs/GetKinematicSolverInfo`。用于从运动学求解器获取待处理关节集合及连杆集合信息。
- (4) `kinematics_msgs/GetConstraintAwarePositionIK`。用于从运动学求解器获取无碰撞运动学逆解的服务。

## 2. 编译运动学功能包集

随后有两个功能包需要编译：

```
rosmake pr2_arm_kinematics  
rosmake pr2_arm_kinematics_constraint_aware
```

### 6.3.2 从运动学节点获取运动学求解器信息

#### 1. ROS 设置

本小节介绍如何获取运动学求解器处理的连杆和关节信息。这些信息将会用于构造正解和逆解。该工作使用 `GetKinematicSolverInfo` 服务并通过查询运动学节点来实现。这里首先需要创建用于求解的功能包：

```
sudo apt-get install ros-electric-pr2-arm-navigation
```

然后改变当前目录到刚刚创建的功能包：

```
roscd kinematics_tutorials
```

另外还需要设置环境：`export ROBOT=sim`。

#### 2. 创建节点

```
1 #include <ros/ros.h>  
2 #include <kinematics_msgs/GetKinematicSolverInfo.h>  
3 int main(int argc, char **argv){  
4     ros::init (argc, argv, "get_ik_solver_info");  
5     ros::NodeHandle nh;  
6     ros::service::waitForService("pr2_right_arm_kinematics/  
7         get_ik_solver_info");  
8     ros::ServiceClient query_client = nh.serviceClient<  
9         kinematics_msgs::GetKinematicSolverInfo>(
```

```

8
9     kinematics_msgs::GetKinematicSolverInfo::Request request;
10    kinematics_msgs::GetKinematicSolverInfo::Response
11        response;
12
13    if(query_client.call(request, response))
14    {
15        for(unsigned int i=0; i< response.kinematic_solver_info
16            .joint_names.size(); i++)
17        {
18            ROS_INFO("Joint: %d %s", i, response.
19                  kinematic_solver_info.joint_names[i].c_str());
20        }
21    }
22
23    ros::shutdown();
24 }

```

代码解释：查询运动学的节点是 `kinematics_msgs/GetKinematicSolver-Info` 消息。对该服务的响应包含如下信息：

- (1) `joint_names`: 运动学节点使用的的关节的向量形式的名称。
- (2) `limits`: 向量形式的关节限制，尤其是用于运动学代码的内部关节限制。
- (3) `link_names`: 执行计算的运动学连杆列表。

### 3. 编译节点

编译代码时首先需要做的是添加 `src/get_solver_info.cpp` 文件到 CMakeLists.txt 以使之编译。打开文件 CMakeLists.txt，然后在文件尾添加如下命令：

```
rosbuild_add_executable(get_solver_info src/get_solver_info.
                        cpp)
```

然后执行命令：`make`。

### 4. 在模拟器中运行

为了使用节点 `pr2_arm_kinematics`，首先需要启动模拟器：

```
roslaunch pr2_gazebo pr2_empty_world.launch
```

然后启动用于右臂的运动学求解节点：

```
rosrun pr2_arm_navigation_tutorials get_solver_info
```

运行成功后，结果如下：

```
[ INFO] 0.000000000: Joint: 0 r_shoulder_pan_joint  
[ INFO] 0.000000000: Joint: 1 r_shoulder_lift_joint  
[ INFO] 0.000000000: Joint: 2 r_upper_arm_roll_joint  
[ INFO] 0.000000000: Joint: 3 r_elbow_flex_joint  
[ INFO] 0.000000000: Joint: 4 r_forearm_roll_joint  
[ INFO] 0.000000000: Joint: 5 r_wrist_flex_joint  
[ INFO] 0.000000000: Joint: 6 r_wrist_roll_joint
```

### 6.3.3 PR2 手臂运动学正解

本小节介绍如何使用运动学节点求解运动学正解并获取 PR2 手臂上连杆的空间坐标位置。

#### 1. 节点设置

```
1 #include <ros/ros.h>  
2 #include <kinematics_msgs/GetKinematicSolverInfo.h>  
3 #include <kinematics_msgs/GetPositionFK.h>  
4  
5 int main(int argc, char **argv){  
6     ros::init (argc, argv, "get_fk");  
7     ros::NodeHandle rh;  
8  
9     ros::service::waitForService("pr2_right_arm_kinematics/  
10                                get_fk_solver_info");  
11  
12     ros::service::waitForService("pr2_right_arm_kinematics/  
13                                get_fk");  
14  
15     ros::ServiceClient query_client = rh.serviceClient <  
16         kinematics_msgs::GetKinematicSolverInfo>(  
17             "pr2_right_arm_kinematics/get_fk_solver_info");  
18     ros::ServiceClient fk_client = rh.serviceClient <  
19         kinematics_msgs::GetPositionFK>(  
20             "pr2_right_arm_kinematics/get_fk");
```

```
14
15     kinematics_msgs::GetKinematicSolverInfo::Request request;
16     kinematics_msgs::GetKinematicSolverInfo::Response
17         response;
18     if(query_client.call(request, response))
19     {
20         for(unsigned int i=0;
21             i< response.kinematic_solver_info.joint_names.size
22                 (); i++)
23         {
24             ROS_DEBUG("Joint: %d %s",
25                     response.kinematic_solver_info.joint_names[i].c_str
26                     ());
27         }
28     }
29     else
30     {
31         ROS_ERROR("Could not call query service");
32         ros::shutdown();
33         exit(1);
34     }
35
36     kinematics_msgs::GetPositionFK::Request fk_request;
37     kinematics_msgs::GetPositionFK::Response fk_response;
38     fk_request.header.frame_id = "torso_lift_link";
39     fk_request.fk_link_names.resize(2);
40     fk_request.fk_link_names[0] = "r_wrist_roll_link";
41     fk_request.fk_link_names[1] = "r_elbow_flex_link";
42     fk_request.robot_state.joint_state.position.resize(
43         response.kinematic_solver_info.joint_names.size());
44     fk_request.robot_state.joint_state.name = response.
45         kinematic_solver_info.joint_names;
46     for(unsigned int i=0;
47         i< response.kinematic_solver_info.joint_names.size();
48             i++)
49     {
50         fk_request.robot_state.joint_state.position[i] = 0.5;
51     }
52     if(fk_client.call(fk_request, fk_response))
```

```
47  {
48      if(fk_response.error_code.val == fk_response.error_code
        .SUCCESS)
49      {
50          for(unsigned int i=0; i < fk_response.pose_stamped.
            size(); i++)
51          {
52              ROS_INFO_STREAM("Link      : " << fk_response.
                fk_link_names[i].c_str());
53              ROS_INFO_STREAM("Position: " << fk_response.
                pose_stamped[i].pose.position.x << "," <<
                fk_response.pose_stamped[i].pose.position.y <<
                "," << fk_response.pose_stamped[i].pose.
                position.z);
54              ROS_INFO("Orientation: %f %f %f %f", fk_response.
                pose_stamped[i].pose.orientation.x, fk_response
                .pose_stamped[i].pose.orientation.y,
                fk_response.pose_stamped[i].pose.orientation.z,
                fk_response.pose_stamped[i].pose.orientation.w
                );
55          }
56      }
57      else
58      {
59          ROS_ERROR("Forward kinematics failed");
60      }
61  }
62  else
63  {
64      ROS_ERROR("Forward kinematics service call failed");
65  }
66  ros::shutdown();
67 }
```

代码解释：

(1) 代码第 35~40 行。填入 GetPositionFK 服务来调用需求。这段代码使用从运动学求解器获得的信息，计算 PR2 运动学正解。查询响应的元素为关节名字。

这里假设用户需要计算右臂的两个连杆的空间坐标：r\_wrist\_roll\_link

和 `r_elbow_flex_link`。同时假设在 `torso_lift_link` 中需要该坐标。我们将填入 `GetPositionFK` 需求。注意到头文件中 `frame_id` 是我们需要的空间位置。因此把它设置为 `torso_lift_link`。

(2) 代码第 41~46 行。设置用于正解的目标关节状态。下一步需要做的是填入目标关节的关节名称和关节位置向量。如果关节值和关节名称向量维数不一致，运动学节点将会拒绝服务。

(3) 代码第 38~67 行。调用正解服务。当需要计算两个连杆的正解时，响应结果应包含 2 个元素的位姿向量。

## 2. 编译节点

编译代码时首先需要做的是添加 `src/get_fk.cpp` 文件到 `CMakeLists.txt` 以使之编译。打开文件 `CMakeLists.txt`，在文件尾添加：

```
rosbuild_add_executable(get_fk src/get_fk.cpp)
```

然后执行命令：`make`。

## 3. 在模拟器中运行

首先启动模拟器：

```
roscd pr2_gazebo  
roslaunch pr2_empty_world.launch
```

然后启动用于右臂的运动学求解节点：

```
roscd pr2_arm_kinematics  
roslaunch launch/pr2_ik_rarm_node.launch
```

输出结果如下：

```
[ INFO] 23.415000000: pr2_arm_kinematics active
```

最终可以运行节点：`./bin/get_solver_info`。运行成功后结果如下：

```
[ INFO] 1025.270999983: Link      : r_wrist_roll_link  
[ INFO] 1025.270999983: Position: 0.520578, 0.180466, -0.445349  
[ INFO] 1025.270999983: Orientation: 0.436766 0.736298  
                  -0.027756 0.516073  
[ INFO] 1025.270999983: Link      : r_elbow_flex_link  
[ INFO] 1025.270999983: Position: 0.395819, 0.0282368, -0.19177  
[ INFO] 1025.270999983: Orientation: 0.124788 0.511289  
                  0.210368 0.823867
```

### 6.3.4 PR2 手臂运动学逆解

本小节介绍如何使用运动学节点求解运动学逆解并获取 PR2 手臂上关节连杆的空间坐标位置。

#### 1. 节点设置

创建文件 `src/get_ik.cpp`, 并添加如下代码:

```
1 #include <ros/ros.h>
2 #include <kinematics_msgs/GetKinematicSolverInfo.h>
3 #include <kinematics_msgs/GetPositionIK.h>
4
5 int main(int argc, char **argv){
6     ros::init (argc, argv, "get_ik");
7     ros::NodeHandle nh;
8
9     ros::service::waitForService("pr2_right_arm_kinematics/
10                                get_ik_solver_info");
11
12    ros::ServiceClient query_client = nh.serviceClient<
13        kinematics_msgs::GetKinematicSolverInfo>(
14        "pr2_right_arm_kinematics/get_ik_solver_info");
15
16    ros::ServiceClient ik_client = nh.serviceClient<
17        kinematics_msgs::GetPositionIK>(
18        "pr2_right_arm_kinematics/get_ik");
19
20    kinematics_msgs::GetKinematicSolverInfo::Request request;
21    kinematics_msgs::GetKinematicSolverInfo::Response
22    response;
23
24    if(query_client.call(request, response))
25    {
26        for(unsigned int i=0; i< response.kinematic_solver_info
27            .joint_names.size(); i++)
28        {
29            ROS_DEBUG("Joint: %d %s", i, response.
30                  kinematic_solver_info.joint_names[i].c_str());
31        }
32    }
33}
```

```
24     }
25     else
26     {
27         ROS_ERROR("Could not call query service");
28         ros::shutdown();
29         exit(1);
30     }
31
32     kinematics_msgs::GetPositionIK::Request gpik_req;
33     kinematics_msgs::GetPositionIK::Response gpik_res;
34     gpik_req.timeout = ros::Duration(5.0);
35     gpik_req.ik_request.ik_link_name = "r_wrist_roll_link";
36
37     gpik_req.ik_request.pose_stamped.header.frame_id =
38         "torso_lift_link";
39     gpik_req.ik_request.pose_stamped.pose.position.x = 0.75;
40     gpik_req.ik_request.pose_stamped.pose.position.y =
41         -0.188;
42     gpik_req.ik_request.pose_stamped.pose.position.z = 0.0;
43
44     gpik_req.ik_request.pose_stamped.pose.orientation.x =
45         0.0;
46     gpik_req.ik_request.pose_stamped.pose.orientation.y =
47         0.0;
48     gpik_req.ik_request.pose_stamped.pose.orientation.z =
49         0.0;
50     gpik_req.ik_request.pose_stamped.pose.orientation.w =
51         1.0;
52     gpik_req.ik_request.ik_seed_state.joint_state.position.
53         resize(response.kinematic_solver_info.joint_names.
54             size());
55     gpik_req.ik_request.ik_seed_state.joint_state.name =
56         response.kinematic_solver_info.joint_names;
57     for(unsigned int i=0; i< response.kinematic_solver_info.
58         joint_names.size(); i++)
59     {
60         gpik_req.ik_request.ik_seed_state.joint_state.position[PDG
61             i] = (response.kinematic_solver_info.limits[i].
62             min_position + response.kinematic_solver_info.
```

```
    limits[i].max_position)/2.0;

51 }
52 if(ik_client.call(gpik_req, gpik_res))
53 {
54     if(gpik_res.error_code.val == gpik_res.error_code.
55         SUCCESS)
56         for(unsigned int i=0; i < gpik_res.solution.
57             joint_state.name.size(); i++)
58             ROS_INFO("Joint: %s %f",gpik_res.solution.
59                 joint_state.name[i].c_str(),gpik_res.solution.
60                 joint_state.position[i]);
61     else
62         ROS_ERROR("Inverse kinematics failed");
63 }
```

代码解释：

(1) 填入逆解服务需求。PR2 运动学逆解目前仅用于 2 个手臂的末端执行器，即 `r_wrist_roll_link` 和 `l_wrist_roll_link`。为了实现逆解的服务需求，需要填入以下三条信息：

① 设置超时。

② 用于计算逆解的连杆目标位姿。

③ 用于逆解计算的期望关节位姿集合。这个状态集合用于启动逆解求解器。

(2) 代码第 34 行。设置超时。设置超时非常重要，如果设置超时时间为 0，则运动学节点将会拒绝服务请求。虽然逆解求解器非常迅速，这里仍然设置超时为 5s。

(3) 代码第 35~45 行。设置用于计算逆解的连杆目标位姿。即 PR2 机器人的连杆 `r_wrist_roll_link`。我们将会在 `torso_lift_link` 坐标系中设置该连杆的目标位姿。

(4) 代码第 46~51 行。为逆解求解器设置初始值。通过域 `ik_seed_state` 启动逆解求解器并设置初始值。

(5) 代码第 52 行。调用逆解服务。由于请求了两个连杆的逆解，响应是包含

两个元素的向量。

(6) 代码第 54~58 行。解释响应。对逆解请求的响应包含在 `motion_planning_msgs/RobotState` 消息中。它包含了逆解的关节值和对应的关节名称。

## 2. 编译节点

编译代码时首先需要做的是添加 `src/get_ik.cpp` 文件到 `CMakeLists.txt` 以使之编译。打开文件 `CMakeLists.txt`，然后在文件尾添加如下命令：

```
rosbuild_add_executable(get_ik src/get_ik.cpp)
```

然后执行命令：`make`。

## 3. 在模拟器中运行

首先启动模拟器：

```
roscore  
roslaunch pr2_gazebo pr2_empty_world.launch
```

然后启动用于右臂的运动学求解节点：

```
roscore  
roslaunch launch/pr2_ik_rarm_node.launch
```

输出结果如下：

```
[ INFO] 1948.588999961: pr2_arm_kinematics active
```

最终，可以运行节点 `./bin/get_solver_info`。运行结果如下：

```
[ INFO] 0.000000000: Joint: r_shoulder_pan_joint -0.330230  
[ INFO] 0.000000000: Joint: r_shoulder_lift_joint 0.008300  
[ INFO] 0.000000000: Joint: r_upper_arm_roll_joint -1.550000  
[ INFO] 0.000000000: Joint: r_elbow_flex_joint -0.859908  
[ INFO] 0.000000000: Joint: r_forearm_roll_joint 3.139403  
[ INFO] 0.000000000: Joint: r_wrist_flex_joint -0.529580  
[ INFO] 0.000000000: Joint: r_wrist_roll_joint -1.591270
```

### 6.3.5 PR2 手臂无碰撞运动学逆解

本小节介绍如何使用运动学节点获取 PR2 手臂无碰撞运动学逆解。

## 1. 设置

无碰撞运动学逆解的设置和前面运动学逆解的设置几乎一样。关键的不同在于：

- (1) 服务调用类型：不使用 `kinematics_msgs` 功能包中的 `GetPositionIK` 服务，而是使用 `GetConstraintAwarePositionIK` 服务。
- (2) 服务调用名称：`get_collision_free_ik`。
- (3) 节点：必须启动另外一个节点集合来设置碰撞检测。
- (4) 运动学求解器：逆解求解器将会从不同的功能包 (`pr2_arm_kinematics_constraint_aware`) 启动。

## 2. 代码

```
1 #include <ros/ros.h>
2 #include <kinematics_msgs/GetKinematicSolverInfo.h>
3 #include <kinematics_msgs/GetConstraintAwarePositionIK.h>
4
5 int main(int argc, char **argv){
6     ros::init (argc, argv, "get_fk");
7     ros::NodeHandle nh;
8
9     ros::service::waitForService("pr2_right_arm_kinematics/
10                                get_ik_solver_info");
11
12    ros::ServiceClient query_client = nh.serviceClient<
13        kinematics_msgs::GetKinematicSolverInfo>(
14        "pr2_right_arm_kinematics/get_ik_solver_info");
15
16    ros::ServiceClient ik_client = nh.serviceClient<
17        kinematics_msgs::GetConstraintAwarePositionIK>(
18        "pr2_right_arm_kinematics/get_constraint_aware_ik");
19
20    kinematics_msgs::GetKinematicSolverInfo::Request request;
21    kinematics_msgs::GetKinematicSolverInfo::Response response;
22
23    if(query_client.call(request, response))
24        cout << "Success" << endl;
25    else
26        cout << "Failure" << endl;
27
28    return 0;
29}
```

```
19  {
20      for(unsigned int i=0; i< response.kinematic_solver_info
21          .joint_names.size(); i++)
22      {
23          ROS_DEBUG("Joint: %d %s",i,response.
24              kinematic_solver_info.joint_names[i].c_str());
25      }
26  }
27  ROS_ERROR("Could not call query service");
28  ros::shutdown();
29  exit(-1);
30 }
31
32 kinematics_msgs::GetConstraintAwarePositionIK::Request
33     gpik_req;
34 kinematics_msgs::GetConstraintAwarePositionIK::Response
35     gpik_res;
36
37 gpik_req.timeout = ros::Duration(5.0);
38 gpik_req.ik_request.ik_link_name = "r_wrist_roll_link";
39
40 gpik_req.ik_request.pose_stamped.header.frame_id =
41     "torso_lift_link";
42 gpik_req.ik_request.pose_stamped.pose.position.x = 0.75;
43 gpik_req.ik_request.pose_stamped.pose.position.y =
44     -0.188;
45 gpik_req.ik_request.pose_stamped.pose.position.z = 0.0;
46
47 gpik_req.ik_request.pose_stamped.pose.orientation.x =
48     0.0;
49 gpik_req.ik_request.pose_stamped.pose.orientation.y =
50     0.0;
51 gpik_req.ik_request.pose_stamped.pose.orientation.z =
52     0.0;
53 gpik_req.ik_request.pose_stamped.pose.orientation.w =
54     1.0;
```

```
48     gpik_req.ik_request.ik_seed_state.joint_state.position.
        .resize(response.kinematic_solver_info.joint_names.
        .size());
49     gpik_req.ik_request.ik_seed_state.joint_state.name =
        response.kinematic_solver_info.joint_names;
50
51     for(unsigned int i=0; i< response.kinematic_solver_info.
        joint_names.size(); i++)
52     {
53         gpik_req.ik_request.ik_seed_state.joint_state.position[
            i] = (response.kinematic_solver_info.limits[i].
            min_position + response.kinematic_solver_info.
            limits[i].max_position)/2.0;
54     }
55     if(ik_client.call(gpik_req, gpik_res))
56     {
57         if(gpik_res.error_code.val == gpik_res.error_code.
            SUCCESS)
58         {
59             for(unsigned int i=0; i < gpik_res.solution.
                joint_state.name.size(); i++)
60             {
61                 ROS_INFO("Joint: %s %f",gpik_res.solution.
                    joint_state.name[i].c_str(),gpik_res.solution.
                    joint_state.position[i]);
62             }
63         }
64     else
65     {
66         ROS_ERROR("Inverse kinematics failed");
67     }
68 }
69 else
70 {
71     ROS_ERROR("Inverse kinematics service call failed");
72 }
73 ros::shutdown();
74 }
```

### 3. 编译节点

编译代码时首先需要做的是添加 `src/get_collision_free_ik.cpp` 文件到 `CMakeLists.txt` 以使之编译。打开文件 `CMakeLists.txt`，然后在文件尾添加如下命令：

```
rosbuild_add_executable(get_collision_free_ik src/get_
    collision_free_ik.cpp)
```

然后执行命令：`make`。

### 4. 在模拟器中运行

首先启动模拟器：

```
roscd pr2_gazebo
roslaunch pr2_empty_world.launch
```

为了获取环境中的碰撞信息，首先需要获取传感器信息，并将其转换为碰撞空间表示的节点。这些工作可以通过使用一个启动文件来实现：

```
roscd pr2_arm_navigation_perception/launch
roslaunch laser-perception.launch
```

然后启动用于右臂的无碰撞运动学求解节点：

```
roscd pr2_arm_navigation_kinematics
roslaunch launch/right_arm_collision_free_ik.launch
```

输出结果如下：

```
[ INFO] 4119.125000380: Initialized pr2_arm_kinematics_
    constraint_aware
```

最终，可以运行节点：`./bin/get_collision_free_ik`。运行成功后，结果如下：

```
[ INFO] 0.000000000: Joint: r_shoulder_pan_joint -0.330230
[ INFO] 0.000000000: Joint: r_shoulder_lift_joint 0.008300
[ INFO] 0.000000000: Joint: r_upper_arm_roll_joint -1.550000
[ INFO] 0.000000000: Joint: r_elbow_flex_joint -0.859908
[ INFO] 0.000000000: Joint: r_forearm_roll_joint 3.139403
[ INFO] 0.000000000: Joint: r_wrist_flex_joint -0.529580
[ INFO] 0.000000000: Joint: r_wrist_roll_joint -1.591270
```

## 6.4 用于 PR2 机器人手臂的安全轨迹控制

本节建立一个功能包，它与安全手臂轨迹控制器进行简单交互。只有在确认机器人不会产生自碰撞或与环境中物体产生碰撞时，控制器才会执行目标轨迹。

### 1. ROS 设置

首先需要创建用于本小节例子的功能包：

```
roscreate-pkg arm_navigation_tutorials actionlib pr2_
    controllers_msgs
```

然后改变当前目录到该功能包下： `roscd arm_navigation_tutorials`。

同时需要设置环境变量：`export ROBOT=sim`。

### 2. 在模拟器中运行

为了使用环境服务器，首先需要启动模拟器：

```
roscd pr2_gazebo
roslaunch pr2_empty_world.launch
```

#### 1) 设置传感器节点

为了获取环境中的碰撞信息，首先需要获取传感器信息，并将其转换为碰撞空间表示的节点。这些工作可以通过使用一个启动文件来实现：

```
roscd pr2_arm_navigation_perception/launch
roslaunch laser-perception.launch
```

#### 2) 设置控制器：

```
roscd pr2_arm_navigation_actions/launch
roslaunch right_arm_collision_free_trajectory_control.launch
```

输出结果如下：

```
[INFO] 116.845000000: Collision free arm trajectory controller
started
```

#### 3) 给控制器发送命令

控制器有一个简单的接口，可以创建动作客户端来命令控制器执行动作。下面代码是传送命令来执行动作的例子。

```
1 #include <ros/ros.h>
2 #include <actionlib/client/simple_action_client.h>
3 #include <pr2_controllers_msgs/JointTrajectoryAction.h>
4
5 typedef actionlib::SimpleActionClient<pr2_controllers_msgs
6 ::JointTrajectoryAction> JointExecutorActionClient;
7
8 void spinThread()
9 {
10     ros::spin();
11 }
12
13 int main(int argc, char** argv)
14 {
15     ros::init(argc, argv, "test_controller");
16     boost::thread spin_thread(&spinThread);
17     JointExecutorActionClient *traj_action_client_ = new
18         JointExecutorActionClient(
19             "collision_free_arm_trajectory_action_right_arm");
20
21     while(!traj_action_client_->waitForServer(ros::Duration
22           (1.0))){
23         ROS_INFO("Waiting for the joint_trajectory_action
24             action server to come up");
25         if(!ros::ok()) {
26             exit(0);
27         }
28     }
29
30     pr2_controllers_msgs::JointTrajectoryGoal goal;
31     goal.trajectory.joint_names.push_back(
32         "r_shoulder_pan_joint");
33     goal.trajectory.joint_names.push_back(
34         "r_shoulder_lift_joint");
35     goal.trajectory.joint_names.push_back(
36         "r_upper_arm_roll_joint");
37     goal.trajectory.joint_names.push_back("r_elbow_flex_joint
38         ");
39     goal.trajectory.joint_names.push_back(
```

```
    "r_forearm_roll_joint");
31   goal.trajectory.joint_names.push_back("r_wrist_flex_joint
      ");
32   goal.trajectory.joint_names.push_back("r_wrist_roll_joint
      ");
33
34   goal.trajectory.points.resize(1);
35   for(unsigned int i=0; i < 7; i++)
36     goal.trajectory.points[0].positions.push_back(0.0);
37   goal.trajectory.points[0].positions[0] = -1.57/2.0;
38   goal.trajectory.points[0].time_from_start = ros::Duration
      (0.0);
39
40   traj_action_client_->sendGoal(goal);
41   ROS_INFO("Sent goal");
42
43   while(!traj_action_client_->getState().isDone() && ros::
44     ok())
45   {
46     ros::Duration(0.1).sleep();
47   }
48 }
```

如果 PR2 的俯仰激光传感器检测到前方路上有障碍物，或者机器人手臂开始移动时发现有障碍物，动作节点将会拒绝移动手臂。

## 6.5 使用轨迹滤波节点进行轨迹滤波

### 6.5.1 生成无碰撞三次样条轨迹

本节介绍如何配置关节轨迹滤波节点生成无碰撞的三次样条轨迹。这种类型的滤波适合于概率规划器，它返回不是很光滑的路径。这个滤波通过在随机路径点上做三次样条插值，然后检查样条轨迹来实现碰撞检查。

为滤波集合所用的 `filters.yaml` 如下：

```
1 service_type: FilterJointTrajectoryWithConstraints
2 filter_chain:
3   -
```

```

4      name: unnormalize_trajectory
5      type: UnNormalizeFilterJointTrajectoryWith - Constraints
6      -
7      name: cubic_spline_short_cutter_smoothen
8      type: CubicSplineShortCutterFilterJoint -
9      TrajectoryWithConstraints
9      params: {discretization: 0.01}

```

用于插值路径的离散参数指定了从滤波返回的路径的时间离散步长。为了查看滤波实际是怎样进行的，图 6.15 画出了单个关节的滤波输出结果。图 6.15(a) 和 (b) 显示了输入到滤波的采样直线路径 ( $y$  轴是关节位置， $x$  轴是时间戳)。滤波之后，路径将会更加光滑，并满足速度和加速度约束。图 6.15(c) 是运行于 PR2 上的一个更好的例子，图中显示的是从概率规划器中得到的原始路径，图 6.15(d) 为滤波之后的轨迹。

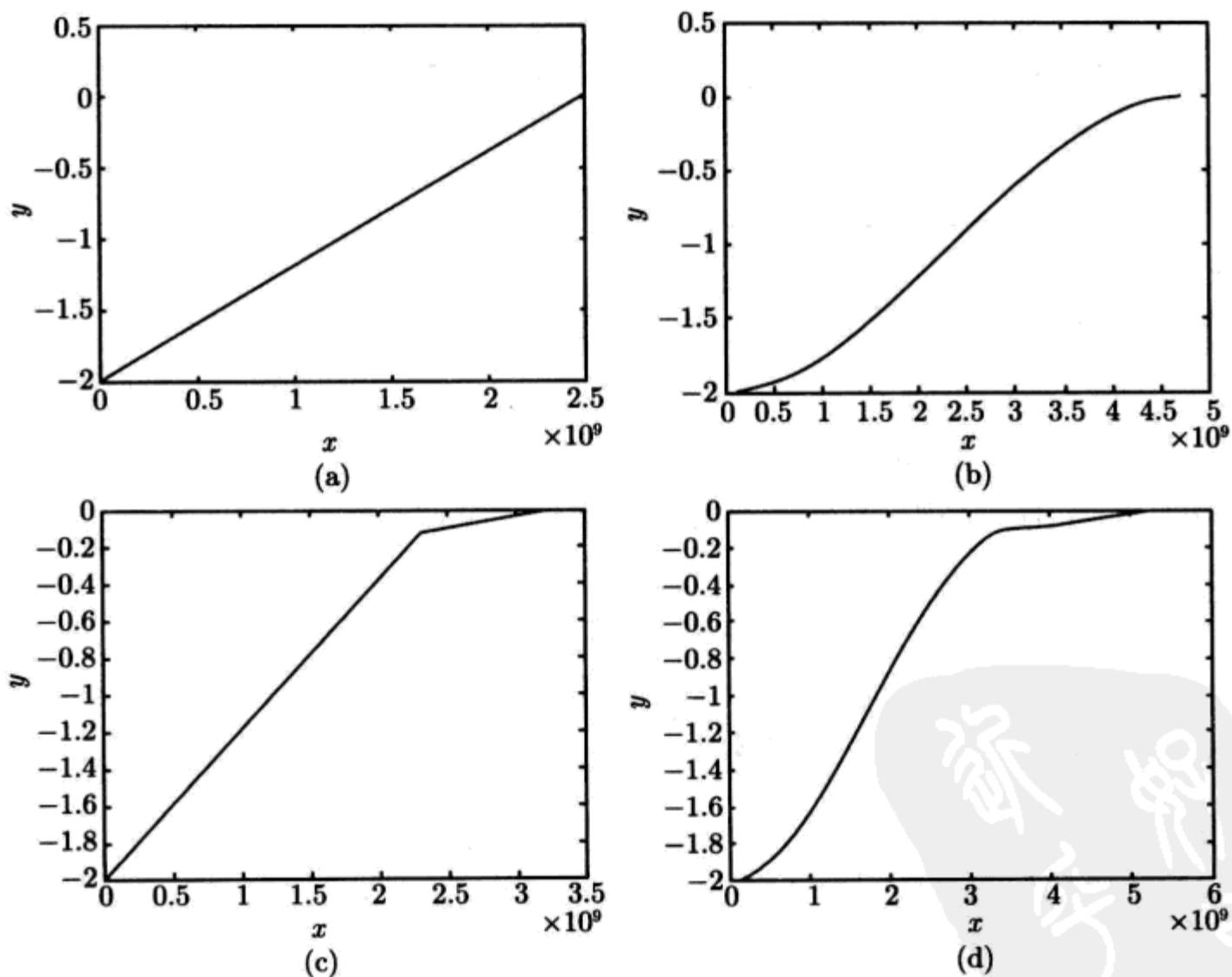


图 6.15 使用轨迹节点滤波

### 6.5.2 使用轨迹滤波服务器对关节轨迹进行滤波

本节介绍如何使用由轨迹滤波服务器提供的轨迹滤波服务。

## 1. ROS 设置

首先需要创建所需的功能包：

```
roscreate-pkg trajectory_tutorials motion_planning_msgs  
trajectory_filter_server pr2_description
```

这里需要注意的是，通常用户不需要依赖项来使用轨迹滤波服务节点，但是由于需要配置并从创建目录启动节点，因此需要添加依赖项。

随后，改变目录到刚才创建的功能包：

```
roscd trajectory_tutorials
```

同时，还需要设置环境变量：`export ROBOT=sim`。

## 2. 为轨迹滤波节点创建启动文件

为了启动轨迹滤波节点，需要配置三个文件。

### 1) 轨迹滤波启动文件

轨迹滤波启动文件 `trajectory_filter.launch` 代码如下：

```
1 <launch>  
2     <!-- send pr2 urdf to param server -->  
3     <param name="robot_description" command="$(find xacro)/  
4         xacro.py '$(find pr2_description)/robots/pr2.urdf.  
5         xacro'" />  
6  
6     <node pkg="trajectory_filter_server" name=  
7         "trajectory_filter" type="trajectory_filter_server"  
8         output="screen">  
9         <rosparam command="load" file="$(find  
10            trajectory_tutorials)/joint_limits.yaml"/>  
11         <rosparam command="load" file="$(find  
12            trajectory_tutorials)/filters.yaml"/>  
13     </node>  
14 </launch>
```

### 2) 滤波器 yaml 文件

轨迹滤波服务器设置了一个滤波链，该滤波链可以传送输入消息。滤波集合是在 `filters.yaml` 文件中指定的。下面是一个 `filters.yaml` 文件的例子：

```

1 service_type: FilterJointTrajectory
2 filter_chain:
3   -
4     name: numerical_differentiation
5     type: NumericalDifferentiationSplineSmoothen -
6       FilterJointTrajectory
7   -
8     name: linear_spline_velocity_scaler
9     type: LinearSplineVelocityScalerFilterJoint - Trajectory

```

第一个域 (`service_type`) 可以取 2 个值: `FilterJointTrajectory` 或者 `FilterJointTrajectoryWithConstraints`。当 `service_type` 为 `FilterJointTrajectory` 时, 轨迹滤波服务器内部设置为收听类型 `FilterJointTrajectory` 的服务。当 `service_type` 为 `FilterJointTrajectoryWithConstraints` 时, 轨迹滤波服务器内部设置为收听类型 `FilterJointTrajectoryServiceWithConstraints` 的服务。

`yaml` 文件指定了 2 个轨迹滤波。每一个滤波有一个滤波类型和一个名称, 该名称不同于相同类型的滤波。输入轨迹将通过第一个滤波 (类型为 `NumericalDifferentiationSplineSmoothen`) 传送, 然后再通过第二个滤波 (类型为 `LinearSplineVelocityScaler`) 传送。第一个滤波通过位置填入, 用数值微分来增加速度轨迹, 第二个滤波在轨迹上的进行连续的路径点线性地伸缩时间, 这样来确保速度在关节约束文件中指定的关节约束范围内。

注意: 如果需要指定相同类型的两个滤波, 需要保证给定它们不同的名称。

### 3) 关节约束参数

下面是一个 `joint_limits.yaml` 的例子, 用来给出两个关节的关节约束参数。

```

joint_limits:
  r_shoulder_pan_joint:
    has_position_limits: true
    min_position: -2.2853981634
    max_position: 0.714601836603
    has_velocity_limits: true
    max_velocity: 0.8
    has_acceleration_limits: true
    max_acceleration: 0.5

```

```
r_shoulder_lift_joint:  
    has_position_limits: true  
    min_position: -0.5236  
    max_position: 0.3963  
    has_velocity_limits: true  
    max_velocity: 0.82  
    has_acceleration_limits: true  
    max_acceleration: 0.5
```

另外还需要注意，除了指定关节限制，还需要设置标志点以检测是否有特殊的约束存在。

### 3. 编写客户端节点

创建 src/filter\_trajectory.cpp 文件：

```
1 #include <ros/ros.h>  
2 #include <motion_planning_msgs/FilterJointTrajectory.h>  
3  
4 int main(int argc, char **argv){  
5     ros::init (argc, argv, "filter_joint_trajectory");  
6     ros::NodeHandle nh;  
7     ros::service::waitForService("trajectory_filter/  
        filter_trajectory");  
8     motion_planning_msgs::FilterJointTrajectory::Request req;  
9     motion_planning_msgs::FilterJointTrajectory::Response  
        res;  
10    ros::ServiceClient filter_trajectory_client_ = rh.  
        serviceClient<motion_planning_msgs::  
        FilterJointTrajectory>("trajectory_filter/  
        filter_trajectory");  
11  
12    req.trajectory.joint_names.push_back(  
        "r_shoulder_pan_joint");  
13    req.trajectory.joint_names.push_back(  
        "r_shoulder_lift_joint");  
14    req.trajectory.points.resize(3);  
15  
16    for(unsigned int i=0; i < 3; i++)  
17    {
```

```
18     req.trajectory.points[i].positions.resize(2);
19 }
20
21 req.trajectory.points[1].positions[0] = 0.5;
22 req.trajectory.points[2].positions[0] = -1.5;
23
24 req.trajectory.points[1].positions[1] = 0.2;
25 req.trajectory.points[2].positions[1] = -0.5;
26
27 req.allowed_time = ros::Duration(1.0);
28
29 if(filter_trajectory_client_.call(req,res))
30 {
31     if(res.error_code.val == res.error_code.SUCCESS)
32     {
33         ROS_INFO("Requested trajectory was filtered");
34         for(unsigned int i=0; i < res.trajectory.points.size
35             (); i++)
36         {
37             ROS_INFO_STREAM(res.trajectory.points[i].positions
38                 [0] << "," << res.trajectory.points[i].
39                 velocities[0] << "," << res.trajectory.points[i]
40                 .positions[1] << "," << res.trajectory.points
41                 [i].velocities[1] << "," << res.trajectory.
42                 points[i].time_from_start.toSec());
43         }
44     }
45     else
46     {
47         ROS_ERROR("Service call to filter trajectory failed %s
48             ",filter_trajectory_client_.getService().c_str());
49     }
50     ros::shutdown();
51 }
```

代码解释：

(1) 代码第 12~25 行。为了调用服务请求，需要给出目标输入轨迹。首先在 joints.yaml 中给出 r\_shoulder\_pan\_joint 和 r\_shoulder\_lift\_joint 的关节轨迹。

(2) 代码第 27 行。指定允许滤波的时间限制。

(3) 代码第 31 行。调用服务之后，需要检查服务响应代码来确定是否有输入轨迹滤波成功。

#### 4. 编译节点

为了编译文件，需要添加文件 src/filter\_trajectory.cpp 到 CMakeLists.txt。打开文件 CMakeLists.txt，在文件尾添加：

```
rosbuild_add_executable(filter_trajectory src/filter_trajectory.cpp)
```

然后执行命令：make。

#### 5. 运行滤波

首先启动文件 trajectory\_filter.launch，并保证刚刚创建的 2 个 yaml 文件的位置与启动文件指定位置一致：

```
roslaunch trajectory_filter.launch
```

运行可执行文件：./bin/filter\_trajectory。

运行成功后，可以看到输出结果：

```
[ INFO] 1266027635.355954000: Requested trajectory was filtered  
[ INFO] 1266027635.356097000: 0,0.8,0,0.32,0  
[ INFO] 1266027635.356166000: 0.5,0.8,0.2,0.32,0.625  
[ INFO] 1266027635.356205000: -1.5,-0.8,-0.5,-0.28,3.125
```

### 6.5.3 学习如何创建自己的轨迹滤波

#### 1. 滤波代码

```
1 #include <motion_planning_msgs/JointTrajectoryWithLimits.h>  
2 #include <motion_planning_msgs/FilterJointTrajectory.h>  
3 #include <filters/filter_base.h>  
4 #include <pluginlib/class_list_macros.h>  
5  
6 template <typename T>
```

```
7
8 class MyFilter: public filters::FilterBase<T>
9 {
10 public:
11     MyFilter() {};
12     ~MyFilter() {};
13
14     bool configure() { return true; }
15     bool update(const T& trajectory_in, T& trajectory_out);
16 };
17
18 template <typename T>
19 inline bool MyFilter<T>::update(const T& trajectory_in, T&
20     trajectory_out)
21 {
22     bool success = true;
23     int size = trajectory_in.trajectory.points.size();
24     int num_traj = trajectory_in.trajectory.joint_names.size()
25         ();
26     trajectory_out = trajectory_in;
27
28     for (int i=0; i<size; ++i)
29     {
30         for (int j=0; j<num_traj; ++j)
31         {
32             double x1 = trajectory_in.trajectory.points[i].
33                 positions[j];
34             trajectory_out.trajectory.points[i].positions[j] =
35                 -x1;
36         }
37     }
38     return success;
39 }
40
41 PLUGINLIB_REGISTER_CLASS(MyFilter, MyFilter<
42     motion_planning_msgs::FilterJointTrajectory::Request>,
43     filters::FilterBase<motion_planning_msgs::
44     FilterJointTrajectory::Request>)
```

## 2. 插件声明

创建名称为 `default_plugins.xml` 的插件声明文件。

```
1 <class_libraries>
2   <library path="lib/libstyle_trajectory">
3     <class name="MyFilter" type="MyFilter<
4       motion_planning_msgs::FilterJointTrajectory::>
5         Request>"base_class_type="filters::FilterBase<
6           motion_planning_msgs::FilterJointTrajectory::>
7         Request>">
8       <description>
9         Generates velocities and accelerations from
          numerical differentiation
10      </description>
11    </class>
12  </library>
13 </class_libraries>
```

## 3. CMakeLists.txt

```
1 cmake_minimum_required(VERSION 2.4.6)
2 include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
3
4 rosbuild_init()
5
6 set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
7 set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
8
9 rosbuild_add_library(${PROJECT_NAME} src/filter.cpp)
```

## 4. Manifest.xml

```
1 <package>
2   <description brief="awesome_trajectory">
3     awesome_trajectory
4
5   </description>
6   <author>David!!</author>
7   <license>BSD</license>
```

```

8   <review status="unreviewed" notes="" />
9   <url>http://ros.org/wiki/awesome_trajectory</url>
10  <depend package="roscpp" />
11  <depend package="filters" />
12  <depend package="trajectory_msgs" />
13  <depend package="motion_planning_msgs" />
14
15  <export>
16    <cpp cflags="-I${prefix}/include -I${prefix}/msg/cpp"
17      lflags="-Wl,-rpath,${prefix}/lib -L${prefix}/lib -
18      lawesome_trajectory"/>
19    <filters plugin="${prefix}/default_plugins.xml" />
20  </export>
21
22 </package>

```

## 6.6 机器人状态和轨迹可视化

### 1. ROS 设置

首先需要创建所需的功能包:

```
roscreate-pkg rviz_display_trajectory_tutorial motion_planning
```

```
_msgs
```

```
planning_environment
```

随后，改变目录到刚才创建的功能包:

```
roscd rviz_display_trajectory_tutorial
```

同时，还需要设置环境变量: `export ROBOT=sim`。

### 2. 例子代码

添加如下代码到 `src/rviz_display_trajectory.cpp` 中:

```

1 #include <motion_planning_msgs/DisplayTrajectory.h>
2 #include <planning_environment/monitors/joint_state_monitor
3 .h>
4
5 void spinThread()
6 {

```

```
7     ros::spin();
8 }
9
10 int main(int argc, char** argv)
11 {
12     ros::init(argc, argv,"display_trajectory_publisher");
13     boost::thread spin_thread(&spinThread);
14     ros::NodeHandle root_handle;
15     planning_environment::JointStateMonitor
16         joint_state_monitor;
17     ros::Publisher display_trajectory_publisher = root_handle
18         .advertise<motion_planning_msgs::DisplayTrajectory>(
19             "joint_path_display", 1);
20     while(display_trajectory_publisher.getNumSubscribers() <
21           1 && root_handle.ok())
22     {
23         ROS_INFO("Waiting for subscriber");
24         ros::Duration(0.1).sleep();
25     }
26     motion_planning_msgs::DisplayTrajectory
27         display_trajectory;
28     unsigned int num_points = 100;
29
30     display_trajectory.model_id = "pr2";
31     display_trajectory.trajectory.joint_trajectory.header.
32         frame_id = "base_footprint";
33     display_trajectory.trajectory.joint_trajectory.header.
34         stamp = ros::Time::now();
35     display_trajectory.trajectory.joint_trajectory.
36         joint_names.push_back ("r_shoulder_pan_joint");
37     display_trajectory.trajectory.joint_trajectory.points.
38         resize(num_points);
39
40     for(unsigned int i=0; i < num_points; i++)
41     {
42         display_trajectory.trajectory.joint_trajectory.points[i].
43             positions.push_back(-(M_PI*i/4.0)/num_points);
44     }
45     display_trajectory.robot_state.joint_state =
```

```

        joint_state_monitor.getJointStateRealJoints();
36    ROS_INFO("Publishing path for display");
37    display_trajectory_publisher.publish(display_trajectory);
38    joint_state_monitor.stop();
39    ros::shutdown();
40    spin_thread.join();
41    return(0);
42 }

```

### 3. 代码解释

(1) 代码第 16~21 行。`visualizer` 使用类型为 `motion_planning_msgs/DisplayTrajectory` 的消息来收听主题，因此首先创建 `ros::Publisher`。用户需要等待直到订阅者连接。

(2) 代码第 25 行。指定了一个 `model_id` 以显示消息，它用于告诉 `rviz` 轨迹属于哪个机器人。

(3) 代码第 26~33 行。填写轨迹域值。指定给 `visualizer` 的轨迹可能只包含了机器人所有轨迹集合的一个子集。为了指定机器人上其他关节的位置，可以使用 `DisplayTrajectory` 消息内的 `robot_state` 域。这一步可以通过关节状态监控器很容易地实现。

关节状态监控器收听 `joint_states` 主题上的当前状态。它可以返回所有关节或者只有真实关节的状态。真实关节指的是物理上真实存在的关节，当然也可以有虚拟的关节。

(4) 代码第 35 行。`rviz` 显示需要真实关节信息。为填写这些真实关节信息，首先需要创建关节状态监控器，然后调用 `getJointStateRealJoints`。

(5) 代码第 37 行。创建并填写显示轨迹信息后，可以发布主题到 `rviz`。

### 4. 编译代码

为了编译文件，添加 `src/rviz_display_trajectory.cpp` 到 `CMakeLists.txt` 中。打开 `CMakeLists.txt` 文件，在文件尾添加如下代码：

```

1 rosbuild_add_executable(rviz_display_trajectory src/
                           rviz_display_trajectory.cpp)

```

然后运行命令：`make`。

## 5. 模拟器设置

```
roscd pr2_gazebo  
roslaunch pr2_empty_world.launch
```

## 6. rviz 设置

首先确保编译。

```
motion_planning_rviz_plugin : rosmake motion_planning_rviz_ plugin.
```

启动 rviz，从左边面板点击添加按钮。在弹出菜单中，Display Type 下，选择 Motion Planning 域下的 Planning，如图 6.16 所示。

点击 topic name 下的空域，填写要发布主题名称 joint\_path\_display。点击 robot\_description 下的空域，填写 ROS 参数的名称，如图 6.16 所示。

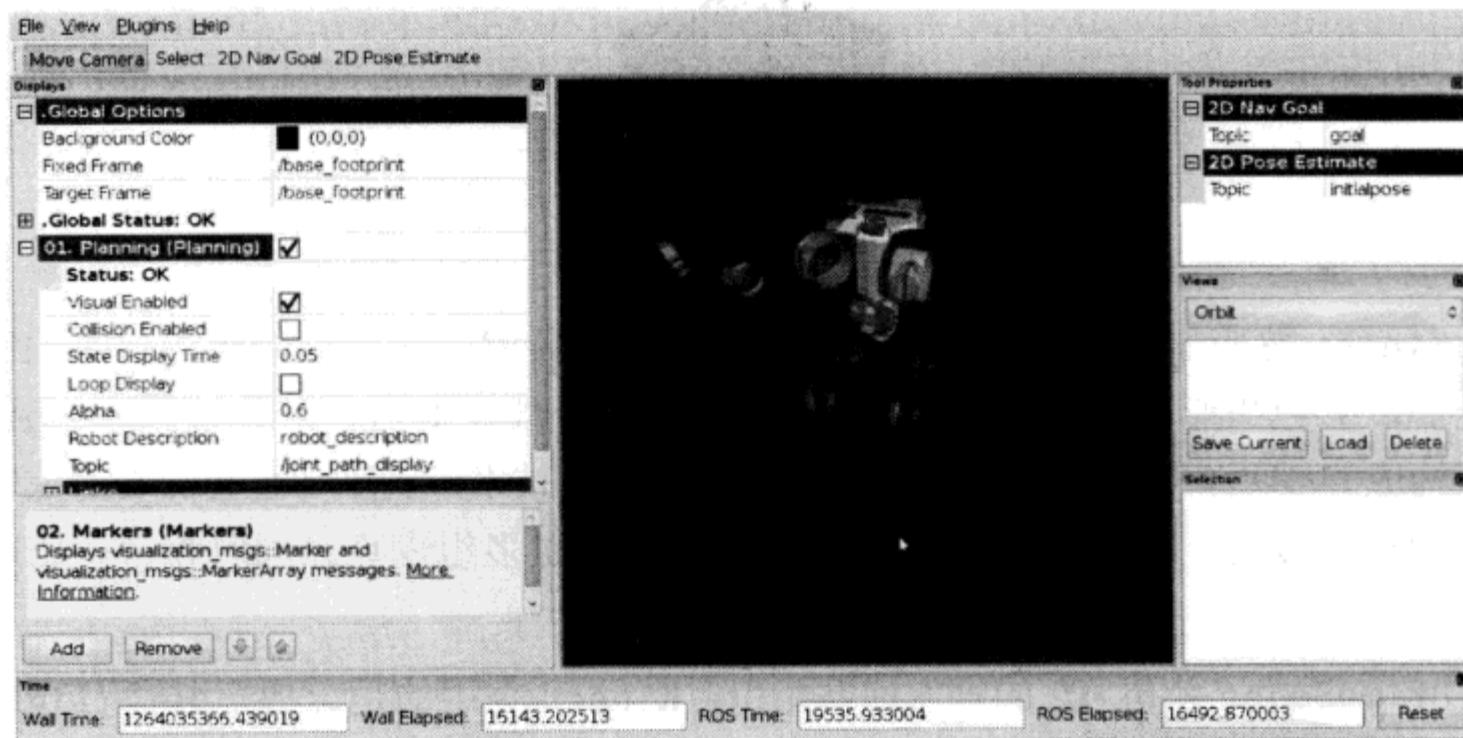


图 6.16 机器人状态和轨迹可视化时 rviz 的设置

随后运行可执行文件：`./bin/joint_trajectory_display`。rviz 中可以看到将手臂向最终位置移动的结果。可使用 State display time 加速或者减速关节轨迹位置的显示刷新频率。

# 第七章 Kinect

## 7.1 Kinect 简介

Kinect for Xbox 360，简称 Kinect<sup>[18]</sup>，是由微软开发应用于 Xbox 360 主机的周边设备（见图 7.1）。它使用语音指令或手势来操作 Xbox 360 的系统界面，让玩家不需要手持或踩踏控制器。它也能捕捉玩家全身上下动作，用身体来进行游戏，带给玩家“免控制器的游戏与娱乐体验”。Kinect 一词是动力学（kinetics）与连接（connection）两个英文单词各取一部分所创的新词汇。

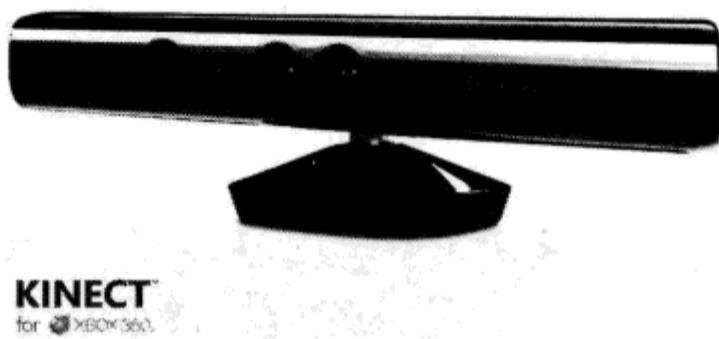


图 7.1 Kinect

Kinect 感应器是一个外型类似网络摄像头的装置，有三个镜头，中间的镜头是 RGB 彩色摄影机，左右两边镜头则分别为红外线发射器和红外线 CMOS 摄像机所构成的 3D 深度感应器。Kinect 还搭配了追焦技术，底座马达会随着对焦物体移动跟着转动。Kinect 也内建阵列式麦克风，由多组麦克风同时收音，比对后消除杂音。

PrimeSense 公司<sup>[19]</sup>的官方网站指出，Kinect 使用的是一种光编码（light coding）技术。不同于传统的 ToF（time of flight）或者结构光测量技术，光编码使用的是连续的照明而非脉冲。它不需要特制的感光芯片而只需要普通的 CMOS 感光芯片，这使得其成本大大降低。顾名思义，光编码就是用光源照明对需要测量的空间编码，其本质还是结构光技术。光编码与传统的结构光方法不同之处在于，其光源是激光散斑（laser speckle）。

激光散斑是当激光照射到粗糙物体或穿透毛玻璃后形成的随机衍射斑点，它是具有三维纵深的“体编码”。这些散斑具有高度的随机性，而且会随着距离的不同变换图案，即空间中任意两处的散斑图案都是不同的。空间中打上这种结构光，整个空间就都被做了标记，把一个物体放进这个空间，只要检测物体上面的

散斑图案，就可以知道这个物体在什么位置。当然，在这之前要把整个空间的散斑图案都记录下来，即先做一次光源的标定。

PrimeSense 公司提供的专利中描述的标定方法如下：每隔一段距离，取一个参考平面，把参考平面上的散斑图案记录下来。假设 Natal 用户活动空间是距离电视机 1~4m 的范围，每隔 10cm 取一个参考平面，那么标定下来我们就已经保存了 30 幅散斑图像。进行测量时，拍摄一幅待测场景的散斑图像，将这幅图像与已经保存下来的 30 幅参考图像依次做互相关运算，这样会得到 30 幅相关度图像，而空间中有物体所在的位置，在相关度图像上就会显示出峰值。把这些峰值一层层叠在一起，再经过一些插值，就会得到整个场景的三维形状了。

## 7.2 安装驱动

### 7.2.1 Ubuntu 系统上安装 Kinect

目前，Kinect 的开发驱动有很多种，如 MS SDK、OpenKinect、OpenNI 等，ROS 官网推荐使用 OpenNI 的驱动。

(1) 下载和安装 `openni_kinect`，打开终端，然后输入以下命令：

```
sudo apt-get install ros-diamondback-openni-kinect
```

(2) 编译 `openni_kinect` 这个功能包集，终端输入：

```
rosmake openni_kinect --rosdep-install
```

(3) 下载控制 Kinect 马达驱动 `kinect_aux`，终端输入：

```
git clone https://github.com/ros-pkg-git/kinect.git
```

(4) 把 `kinect_aux` 放到 `ROS_PACKAGE_PATH` 下，这样才能被 ROS 命令找到并编译：

```
rosmake kinect_aux
```

### 7.2.2 基于源的安装

#### 1. 安装 OpenNI 驱动

```
hg clone https://kforge.ros.org/openni/drivers  
cd drivers  
make  
make install
```

### 1) Debian 用户

此时不需要进行 make，用户可能需要编译和安装 Debian packages。这保证了功能包管理器可以追踪所有安装在系统上的功能包。

```
make debian  
sudo dpkg -i *.deb
```

### 2) Mac OS X 用户

假设用户有 NITE 1.3.1.5 Mac OS X 软件包，然后可按照 Bug Ticket System 中 36 号方案中第 5 项评论进行安装。

```
# Get revision #272  
hg clone https://kforge.ros.org/openni/drivers  
cd drivers  
hg checkout 272  
  
# Apply patches from Ticket #36...  
patch openni/Makefile < openni-Makefile.patch  
patch ps_engine/Makefile < ps_engine-Makefile.patch  
# ...and so forth  
  
# First build nite directory if you have a local tarball  
according  
to Ticket #36, Comment #5  
cd nite  
make TARBALL_URL=file:///Users/username/NITE-Bin-MacOSX-  
v1.3.1.5.tar.bz2  
TARBALL=NITE-Bin-MacOSX-v1.3.1.5.tar.bz2  
cd ..  
  
# Build and install drivers  
make  
make install
```

2. 运行 rosinstall

```
rosinstall ~ /openni_kinect PATH_TO_EXISTING_ROS_TREE
```

```
'http://www.ros.org/wiki/openni_kinect?action=AttachFile&do=get&target=openni_kinect.rosinstall'
```

### 3. 设置环境

每次打开新的终端时，添加下面一行程序到 `.bashrc` 或者源安装文件：

```
~ /openni_kinect/setup.bash
```

### 4. 编译

```
rosmake openni_kinect --rosdep-install
```

安装之后，用户需要重新插拔所有 OpenNI (Primesense SDK & Kinect) 兼容的设备。

## 7.3 测 试

### 7.3.1 测试 Kinect 彩色摄像机

(1) 在终端输入以下命令：

```
roslaunch openni_camera openni_node.launch
```

启动 `openni_camera` 节点，该节点启动后才能使用 Kinect 的彩色摄像机和深度摄像机。

(2) 使用 `image_view` 查看 Kinect 的 RGB 摄像机获取的图像，新开一个终端，输入：

```
rosrun image_view image_view image:=/camera/rgb/image_color
```

也可以查看灰度图像：

```
rosrun image_view image_view image:=/camera/rgb/image_mono
```

说明：这种方式比较快捷，也可以使用 `rviz` 来查看图像：`rosrun rviz rviz`。

### 7.3.2 测试 Kinect 深度摄像机

运行 `rviz`，终端输入：

```
rosrun rviz rviz
```

运行该命令之前请确保 `openni_camera` 节点正在运行，否则参考 7.3.1 小节的方法启动 `openni_camera`。

在 rviz 窗口左上角, Fixed Frame 设置为 /openni\_rgb\_optical\_frame。添加一个 Point Cloud2 显示类型 (rviz 窗口左下方点击 Add, 在新打开的窗口中选择 Point Cloud2, 点击 OK), 把它的 Topic 设置为 /camera/rgb/points。建议把背景设置为浅灰色, 方便查看结果。如果此时用户电脑运行非常慢, 把分辨率调小即可加快其运行速度。

### 7.3.3 测试 Kinect 马达

(1) 运行 `kinect_aux` 节点, 终端输入:

```
rosrun kinect_aux kinect_aux_node
```

(2) 查看当前 Kinect 的俯仰角, 新开一个终端, 输入:

```
rostopic echo /cur_tilt_angle
```

该终端里面一直有数字在显示, 显示的就是 Kinect 当前的俯仰角, + 说明是仰角, - 说明是俯角。

(3) 改变 Kinect 的俯仰角, 新开一个终端, 输入:

```
rostopic pub -1 /tilt_angle std_msgs/Float64 -- <N>
```

其中 `<N>` 是要转动的角度 (单位为度), 可以是正数也可以是负数, 正负号意义参见上面所述, 注意转动的角度范围为 [-31:31] 度。

## 7.4 openni\_camera

用于 OpenNI 深度 + 彩色摄像机的 ROS 驱动, 它包括:

- (1) Microsoft Kinect。
- (2) PrimeSense PSDK。
- (3) ASUS Xtion Pro (no RGB)。

`openni_launch` 是启动 Kinect 及其类似设备的最好的地方。从 Electric 版本开始, `openni_camera` 已经缩减到只是发布原始深度、红外以及 RGB 图像的驱动节点。

### 1. Kinect 精度

由于 Kinect 本质上是一个立体摄像机系统, 深度测量的误差与距离的平方成正比。图 7.2 中的数据结果证实了这种猜测。图形中的数据是通过下面几步得到的: 首先把 Kinect 指向一个平面, 通过测量点云来拟合 (使用 RANSAC, 随机

采样一致性算法<sup>[20]</sup>) 平面, 最后检查点云中的点到平面的距离。这意味着平面表示了平均距离度量, 图 7.2 表示了 Kinect 距离该平均距离有多远。

从 Kinect 获取的原始深度图像是没有矫正过的。为矫正图像, 用户需要首先标定 Kinect 的摄像机内参数。用户自己使用矫正图像时, 会发现其精度比图 7.2 中的精度稍差。

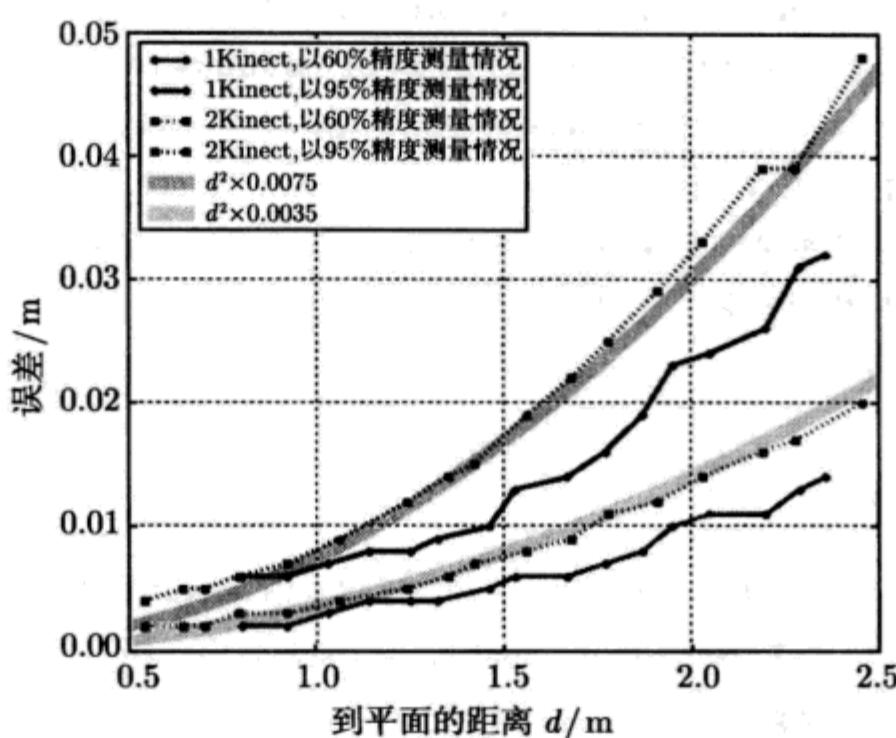


图 7.2 Kinect 精度

标定过的 Kinect 精度实际上是很高的。我们可以把 Kinect 图像与下面图像做比较: ①一个标定过的单目摄像机检测出的棋盘格的位姿; ②卷尺测量的距离。结果表明误差精度为正负 1mm, 这在其他两种方法的误差允许范围内。

最终 Kinect 的精度会极大地受到 Kinect 摄像机标定内外参数的影响。一幅未标定的深度图像甚至有 1cm 的误差。

## 2. Kinect 标定

### 1) 内参数 (Intrinsics) 标定

Kinect 带有板载的出厂标定, 基于高阶多项式拟合函数。对于大部分应用, 这已经足够了。用户可能需要使用 ROS 中 `image_pipeline` 所使用的畸变模型来获得更高的精度。

Kinect 不能存储用户的标定参数到固件上, 任何用户标定必须存于一个非板载的文件。标定文件位置由 `openni_node` 驱动 (或 `openni_camera/driver nodelet`) 中的 `rgb_camera_info_url` 和 `depth_camera_info_url` 参数指定。

简单的启动文件例子：

```

1 <launch>
2   <node pkg="nodelet" type="nodelet" name="manager" args=
    "manager"/>
3
4   <!-- Driver nodelet -->
5   <node pkg="nodelet" type="nodelet" name="openni_node" args
      ="load openni_camera/driver manager">
6     <param name="rgb_camera_info_url" value="file:///home/
      user/rgb.yaml" />
7     <param name="depth_camera_info_url" value="file:///home
      /user/depth.yaml" />
8   </node>
9 </launch>
```

## 2) 只用彩色摄像机

从 camera\_calibration 功能包的 cameracalibrator.py 文件开始：

```
rosrun camera_calibration cameracalibrator.py --size 8x6
--square 0.108
image:=/camera/rgb/image_mono camera:=/camera/rgb
--no-service-call
```

该例子假设用户有棋盘格，参数为  $8 \times 6$ ，每个格子点面积为  $0.108\text{m}^2$ 。完成标定后，可以看到位于 `rgb_camera_info_url` 的内参数文件。

## 3) 只使用深度摄像机

深度标定是基于 Kinect 的红外图像进行的，而不是 3D 点云。Kinect 有一个模式投影仪，可以投影散斑图到标定板上，并且在红外图像中是检测不到的。在标定时，可以把投影仪挡住，以提供红外光源来照亮标定板。

## 4) 同时使用深度和彩色摄像机

现在还处于不可用状态。由于用户不能同时从红外摄像机和彩色摄像机同时接收图像，沿着相对位姿尝试估计深度和 RGB 内参数会很麻烦。

## 3. 改变驱动器设置

开始 `dynamic_reconfigure`：

```
rosrun dynamic_reconfigure reconfigure_gui
```

选择 `/openni_node1`，可以看到图 7.3 所示的结果。

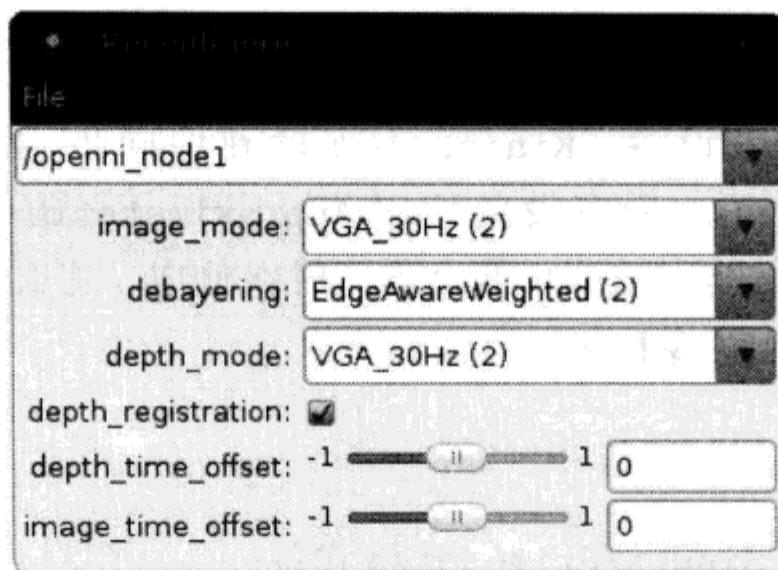


图 7.3 OpenNI 动态可重配置参数界面

具体参数解释如下：

(1) image mode:

① Kinect 默认支持显示分辨率及刷新频率 SXGA@15Hz，实际上视频流只有 10Hz 和 VGA@30Hz 两种模式。所有在 30Hz 模式下的采样下降为 25Hz，Kinect 不支持这种频率。

② Primesense 设备目前还不支持 SXGA。所有 VGA 和 QVGA 模式中，只有 QVGA 模式被下采样。

(2) debayering: 只适用于 Kinect。

① Bilinear: 双线性，快速 debayering 算法，边缘处效果不好。

② EdgeAware: 在最低梯度方向采用线性插值（推荐使用）。

③ EdgeAwareWeighted: 根据梯度使用双线性插值。

④ Note: 下采样图像时使用，通过  $2 \times 2$  块实现 debayering。

(3) depth mode: 与 image mode 相同，不支持 SXGA 模式。

(4) depth\_registration: 深度图像是否与彩色图像配准。

① Note: 如果要求配准，点云放在 /openni\_rgb\_optical\_frame，其他情况下放在 /openni\_depth\_optical\_frame。

② Note: 订阅 /camera/rgb/points 时自动打开配准，如果任何节点向该主题订阅，则不能打开配准。

(5) depth\_time\_offset: 由于设备不会返回与系统时间合成的时间戳，因此在捕捉图像与 ROS 创建图像之间有延时。这个参数用于设置延时。

(6) image\_time\_offset: 作用同上面参数类似。

#### 4. 同步

除了 Primesense 设备，Kinect 不支持硬件同步。彩色图像和深度图像捕捉时间相差 16ms。所有设备使用 `Approximatetimesynchronizer` 生成 `/camera/rgb/points` 点云。如果两个流同时在使用，那么 Primesense 设备硬件同步自动打开（订阅图像和深度信息）。

#### 5. 坐标系

下述坐标系用于 Kinect/Primesense 设备：

```
/openni_camera
|
|> /openni_rgb_frame
|
|> /openni_rgb_optical_frame (Z forward)
|
|> /openni_depth_frame
|
|> /openni_depth_optical_frame (Z forward)
```

### 7.5 openni\_tracker

`openni_tracker` 是一个独立节点，不需要其他节点即可运行：

```
rosrun openni_tracker openni_tracker
```

运行之后，需要静止站在 Kinect 前面，做出“投降”姿势。很快即可看到显示信息（见图 7.4）：

```
New User 1
Pose Psi detected for user 1
Calibration started for user 1
Calibration complete, start tracking user 1
Lost User 1
```

用户姿势将作为带有下列坐标系名的变换集合 (`/tf`) 发布出去：

- `/head`
- `/neck`

- /torso
- /left\_shoulder
- /left\_elbow
- /left\_hand
- /right\_shoulder
- /right\_elbow
- /right\_hand
- /left\_hip
- /left\_knee
- /left\_foot
- /right\_hip
- /right\_knee
- /right\_foot

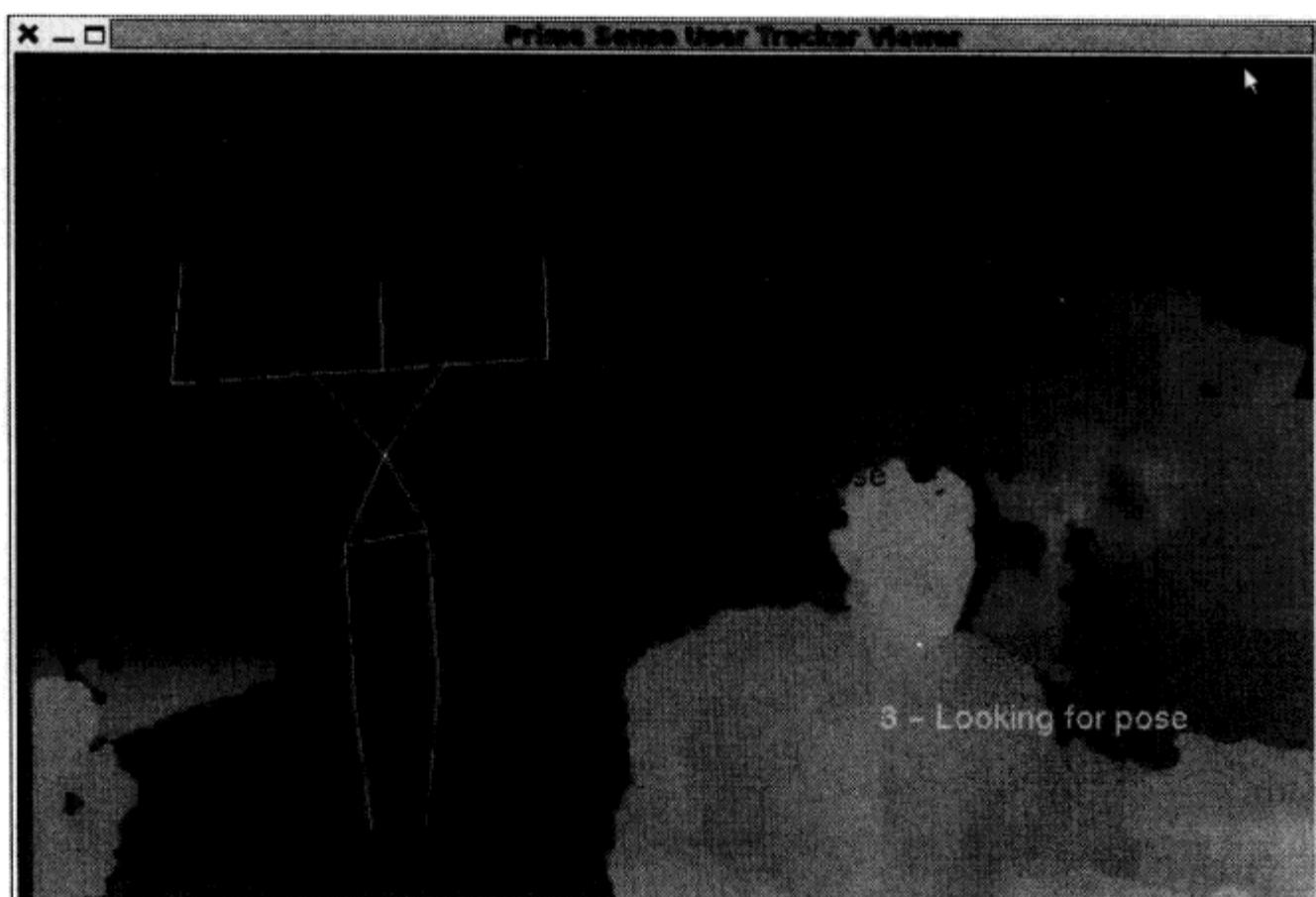


图 7.4 openni\_tracker

openni\_tracker 使用 tf 广播 OpenNI 坐标系。

## 第八章 点 云 库

经过 20 多年的发展，机器人传感器领域已经发生了巨大变化：从基于声纳和 IR 提供的简单测距功能到现在的视觉传感器和激光扫描仪。由视觉传感器和激光传感器提供的大量 3D 数据已经变得实用起来，像 DARPA 无人汽车挑战赛和 Willow Garage 公司提供的 PR2 机器人已经可以使用高质量的 3D 点云来感知世界。但是，激光传感器过于昂贵，无法普及。近年来，一些 3D 传感器价格已经变得可以为普通用户接受，像微软公司提供的 Xbox 360 游戏机附带的传感器 Kinect，价格只需要 1000 多元人民币。Kinect 可以提供 3D 点云，使得机器人可以看到 3D 世界。我们所需要的还有至关重要的一步：如何有效处理 3D 点云数据？PCL(point cloud library)<sup>[21~25]</sup> 给出了很好的解决方案。

### 8.1 PCL 简 介

PCL 是一个独立的，大规模开源点云处理软件包，现在已经完全集成到 ROS 中。PCL 被设计用于感知 3D 世界。由于其设计为处理在线实时传感器数据，它可以处理实时感知设备的数据，像激光扫描仪，双目立体视觉系统和 ToF (Time of Flight) 摄像机。图 8.1 为 PCL 点云库的图标。

#### 1. PCL 版本变化

- PCL 1.3.0 (2011.10.31)
- PCL 1.2.0 (2010.09.30)

#### 2. PCL 支持的操作系统

- Linux
- Microsoft Windows
- Apple Mac OS X

#### 8.1.1 PCL 架构

PCL 是一个模板化的，用于 3D 点云处理的 C++ 库。它的数据结构利用 SSE 优化技术和 GPUs(CUDA) 技术，可以高效执行并在现代 CPU 上实现。

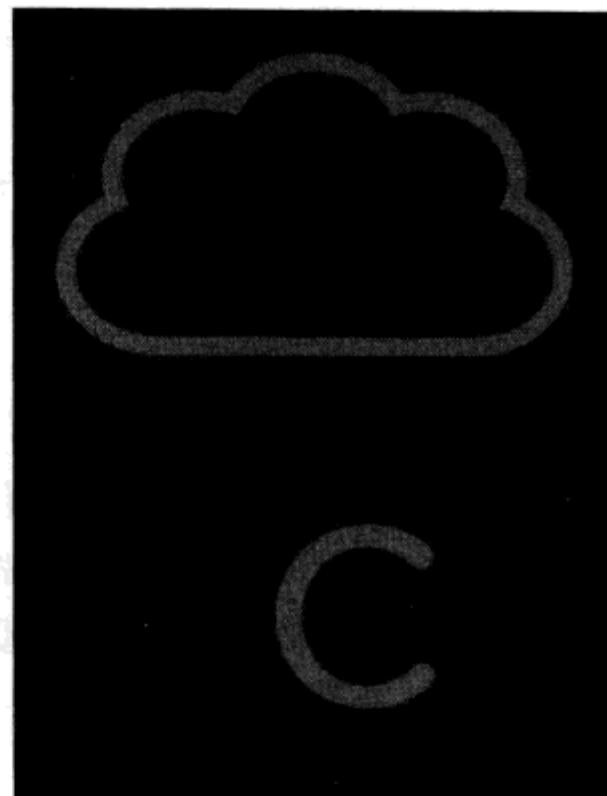


图 8.1 PCL 点云库图标

PCL 的数学操作基于 Eigen (一个开源的线性代数模板库<sup>[26]</sup>)。另外 PCL 还支持 OpenMP<sup>[27]</sup> 和 Intel 公司提供的用于多核并行计算的 Intel Threading Building Blocks (TBB) 库。PCL 的 k- 最近邻搜索算法框架是由 FLANN(fast library for approximate nearest neighbors)<sup>[28]</sup> 提供的。

PCL 提供针对点云数据的基本操作：滤波、特征估计、曲面重构、模型拟合、分割、配准等。其每一个算法集都是通过基类来定义的，这样做可以继承所有的常见功能，从而保持算法的紧凑和整洁。

PCL 基本接口功能有：

- (1) 创建被处理物体 (如滤波、特征估计、分割等)。
- (2) 使用 `setInputCloud` 传递输入数据集到数据处理模块。
- (3) 设置参数。
- (4) 调用计算机获得输出。

为了利于发展，PCL 分为一系列小的可独立编译的核心代码库：

- (1) libpcl filters: 执行数据滤波，如下采样、野值点移除 (outlier removal)、指标提取、投影等。
- (2) libpcl features: 计算 3D 特征，如曲面法向量和曲率、二值点估计、矩不变量、主曲率、PFH 和 FPFH 描述子、SIFT 特征等。
- (3) libpcl io: 执行输入/输出操作，如从 PCD (point cloud data) 文件写入/读取操作。

- (4) libpcl segmentation: 执行聚类提取、通过采样一致性方法进行模型拟合、提取多边形棱镜等。
- (5) libpcl surface: 曲面重构、网格化、凸包计算、移动最小二乘方法等。
- (6) libpcl registration: 点云配准算法, 如 ICP 等。
- (7) libpcl keypoints: 提取不同的关键点。
- (8) libpcl range image: 由点云数据集创建距离图像。

为保证 PCL 操作正确性, 上述核心库的方法和类包含了单元和回归测试。单元测试套件根据需要进行测试, 如果其中一个组件测试出现问题, 可以及时反馈给测试该组件的作者, 而不会影响其他组件。这样的设计可以使得增加新的功能和修改不会影响已经存在的代码。

### 8.1.2 PCL 数据结构

#### 1. PCL 数据结构

ROS 系统中, 目前点云的数据结构类型有以下几种:

(1) `sensor_msgs::PointCloud`: 它包含点的  $(x, y, z)$  三维坐标, 复合通道各自的类型及其数值大小, 在初始化 `point_cloud_mapping` 功能包, ROS1.0 及以前版本中显示三维点云等场合中经常使用。

(2) `sensor_msgs::PointCloud2`: 它是 ROS 依据新颁布的 PCL 标准修订后的点云数据类型, 目前主要描述  $n$  维数据。具体的数值类型可以是 `int`、`float`、`double` 等任何基本数据类型。同时, 它的消息机制可以被定义为有高度、宽度数值的“稠密”类型(对应于 2D 数据格式), 这可被转换成相同空间区域的一幅图片。

(3) `pcl::PointCloud<T>`: 这是 PCL 库中主要数据类型, 它可以是 `point_types.h` 定义的任何一种用户自定义类型。

#### 2. PCL 数据类型之间的转换

上述三种数据类型之间可以相互转换。调用 `pcl::fromROSMsg` 和 `pcl::toROSMsg` 方法可以转换 `sensor_msgs::PointCloud2` 和 `pcl::PointCloud<T>` 对象类型。`sensor_msgs::PointCloud` 和 `sensor_msgs::PointCloud2` 类型之间的转换既可以使用 `point_cloud_converter` 结点, 也可以在代码中自行调用 `sensor_msgs::convertPointCloud2ToPointCloud` 和 `sensor_msgs::convertPointCloudToPointCloud2` 方法。

### 8.1.3 PCL 与 ROS 的集成

PCL 设计的核心思想是通过 PPG(perception processing graphs) 来体现的。PPG 背后体现的思想是最常应用的模块应该通过建立参数化的块来达到不同的效果。例如，墙面的检测、门的检测、桌面检测都包含了相同的平面分割块算法。通过独立模块的设计和处理图 (processing graphs) 的创建，使得 PCL 可以在 ROS 生态系统中很容易地与其他节点连接起来。

更进一步，由于点云数据量很大，用户希望保证避免不必要的数据拷贝或序列化及反序列化，这样有利于算法的实时运行。为此，PCL 提供了 nodelet，它类似于 ROS 中的节点。

## 8.2 PCL 可视化库

为了快速提供原型和查看操作大量数据的算法结果，PCL 基于 VTK 提供了自己的可视化库。通过提供用于  $n$ -D 点云结构的 comprehensive visualization layer，PCL 可视化库与 VTK 集成在一起。PCL 可视化库也提供了用于 PCD 文件的通用性工具，这样可以实现 ROS 中传感器提供的实时数据可视化。

PCL 可视化库 (`pcl_visualization`) 提供的功能类似于 OpenCV 中的 `highgui` 所提供的功能：

(1) 以 `pcl::PointCloud<T>` 格式为  $n$ -D 点云数据提供渲染和设置可视化属性 (颜色，点的尺寸等如图 8.2 所示点云渲染后的兔子)。



图 8.2 PCL 点云渲染后的兔子

(2) 为从点集或者参数方程中的数据提供基本的 3D 形状 (如圆柱、球、线、多面体等) 画图功能 (见图 8.3)。

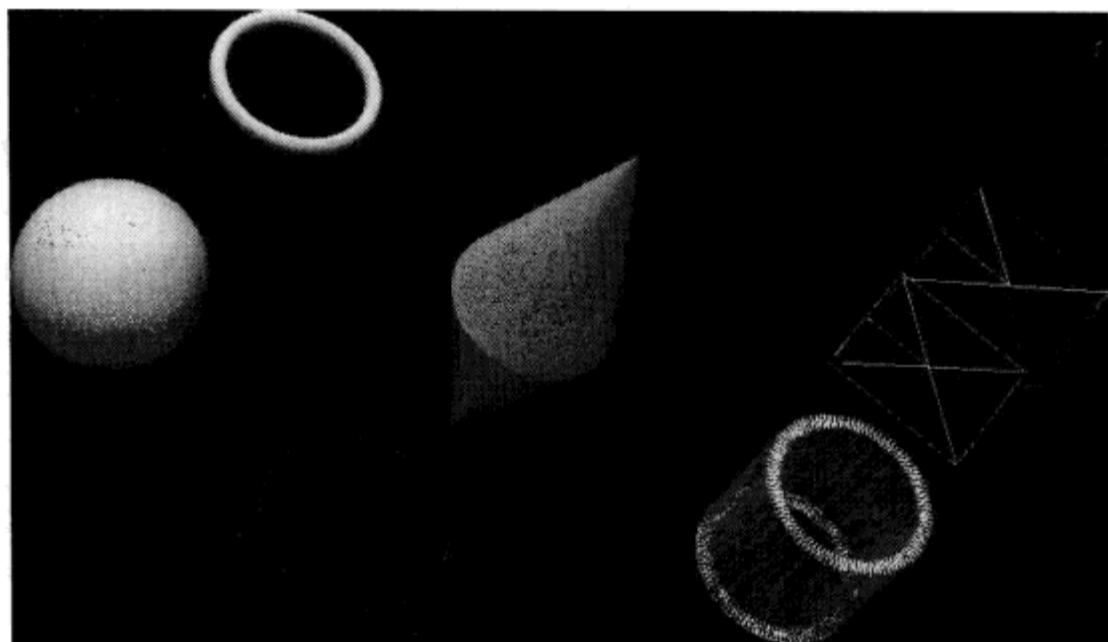


图 8.3 各种几何形状

(3) 用于 2D 画图的直方图可视化模块 (`PCLHistogramVisualizer`)，图 8.4 给出了一个例子。

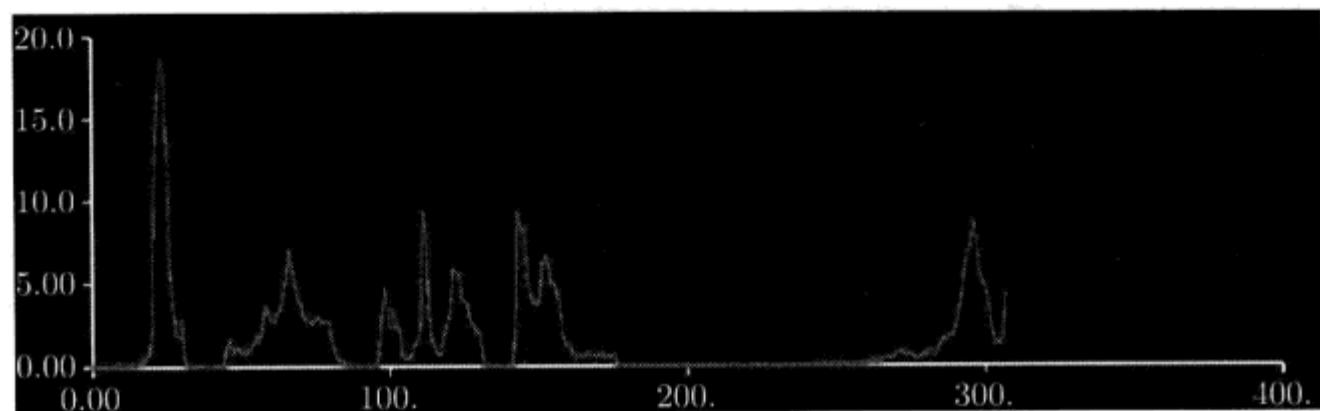


图 8.4 直方图可视化

(4) 为数据集 `pcl::PointCloud<T>` 提供大量几何和颜色句柄。图 8.5 和图 8.6 分别给出了一个杯子的法向量图和室内环境下的距离图像。

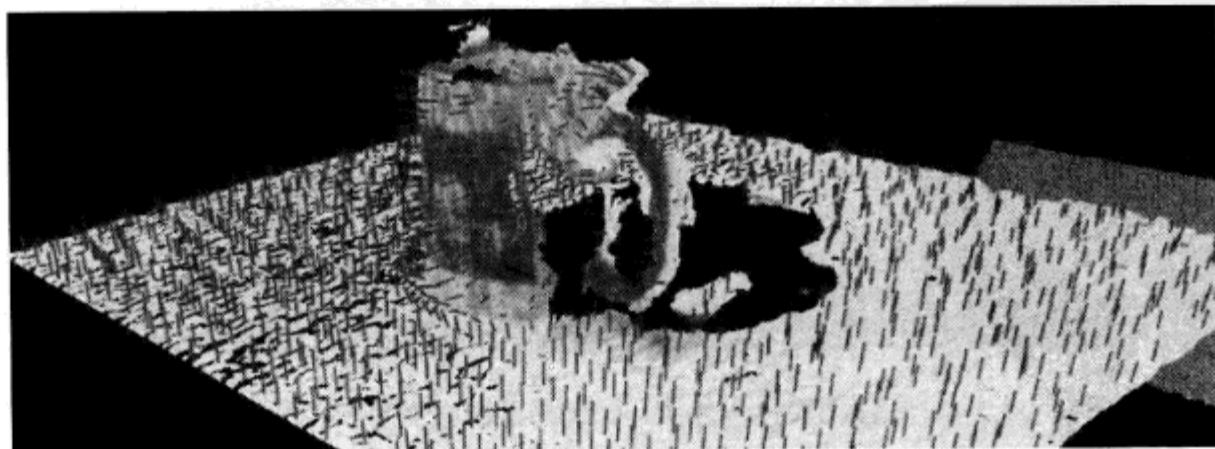


图 8.5 法向量图

(5) `pcl::RangeImage` 可视化模块。

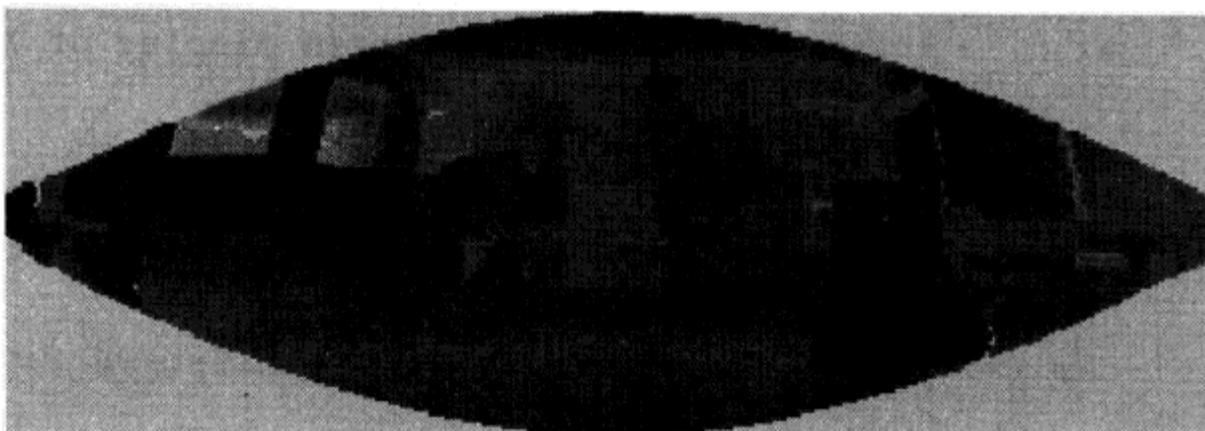


图 8.6 距离图像

### 1. 简单的点云可视化

如果用户希望在自己的应用中实现点云可视化功能，可以使用如下代码：

```
1 #include <pcl_visualization/cloud_viewer.h>
2 //...
3 void
4 foo ()
5 {
6     pcl::PointCloud<pcl::PointXYZRGB> cloud;
7     //... populate cloud
8     pcl_visualization::CloudViewer viewer("Simple Cloud
9         Viewer");
10    viewer.showCloud(cloud);
11    while (!viewer.wasStopped())
12    {
13    }
```

### 2. PCD 可视化器

使用 pcd\_viewer 建立快速可视化 PCD 文件。pcd\_viewer 0.2.7 版本的帮助文档如下：

Syntax is: pcd\_viewer <file\_name 1..N>.pcd <options>  
where options are:  

-bc r,g,b	= background color
-fc r,g,b	= foreground color
-ps X	= point size (1..64)
-opaque X	= rendered point cloud opacity (0..1)

-ax n = enable on-screen display of XYZ axes  
 and scale them to n  
 -ax\_pos X,Y,Z = if axes are enabled, set their X,Y,Z  
 position in space (default 0,0,0)  
 -cam (\*) = use given camera settings as initial  
 view  
 (\*) [Clipping Range / Focal Point / Position / ViewUp /  
 Distance  
 / Window Size / Window Pos] or use a <filename.cam> that  
 contains the same information.  
 -multiview 0/1 = enable/disable auto-multi viewport  
 rendering (default disabled)  
 -normals 0/X = disable/enable the display of every  
 Xth point's surface normal as lines  
 (default disabled)  
 -normals\_scale X = resize the normal unit vector size  
 to X (default 0.02)  
 -pc 0/X = disable/enable the display of every  
 Xth point's principal curvatures as  
 lines (default disabled)  
 -pc\_scale X = resize the principal curvatures  
 vectors size to X (default 0.02)

(Note: for multiple .pcd files, provide multiple -{fc,ps} parameters; they will be automatically assigned to the right file)

### 3. 使用方法示例

#### 1) 例子 1

```
pcd_viewer -multiview 1 data/partial_cup_model.pcd data/
```

```
partial_cup_model.pcd data/partial_cup_model.pcd
```

上述命令将载入 `partial_cup_model.pcd` 文件 3 次，并创建多视角的渲染图像，结果如图 8.7 所示。



图 8.7 渲染茶杯 3 次

在点云被渲染的时候，按键 h 将会在屏幕上输出如下结果：

```
| Help:  
-----  
p, P      : switch to a point-based representation  
w, W      : switch to a wireframe-based representation (where  
            available)  
s, S      : switch to a surface-based representation (where  
            available)  
  
j, J      : take a .PNG snapshot of the current window view  
c, C      : display current camera/window parameters  
  
+ / -    : increment/decrement overall point size  
  
g, G      : display scale grid (on/off)  
u, U      : display lookup table (on/off)  
  
r, R [+ Alt] : reset camera [to viewpoint = {0, 0, 0} ->  
            center_{x, y, z}]  
  
Alt + s, S   : turn stereo mode on/off  
Alt + f, F   : switch between maximized window mode and  
            original size
```

```

l, L      : list all available geometric and
            color handlers for the current
            actor map

Alt + 0..9 [+ Ctrl] : switch between different geometric
            handlers (where available)

0..9 [+ Ctrl] : switch between different color
            handlers (where available)

```

在点云被渲染的时候，按键 l 将会显示目前可以装载的几何/颜色句柄。在这个例子中，结果如下：

```

List of available geometry handlers for actor partial_cup_
model.pcd-0: xyz(1) normal_xyz(2)

List of available color handlers for actor partial_cup_
model.pcd-0: [random](1) x(2) y(3) z(4) normal_x(5)
normal_y(6) normal_z(7) curvature(8) boundary(9) k(10)
principal_curvature_x(11) principal_curvature_y(12)
principal_curvature_z(13) pc1(14) pc2(15)

```

使用 Alt+1 可以切换到 normal\_xyz 几何句柄，然后按 8，可以切换到曲率彩色句柄，结果如下：

```
pcd_viewer -normals 100 data/partial_cup_model.pcd
```

上述命令将加载文件 `partial_cup_model.pcd` 并渲染其每隔 100 个曲面法向量，结果如图 8.8 所示。



图 8.8 茶杯模型法向量图

## 2) 例子 2

```
pcd_viewer -pc 100 data/partial_cup_model.pcd
```

上述命令将加载文件 `partial_cup_model.pcd` 并渲染其每隔 100 个曲面法向量。结果如图 8.9 所示。

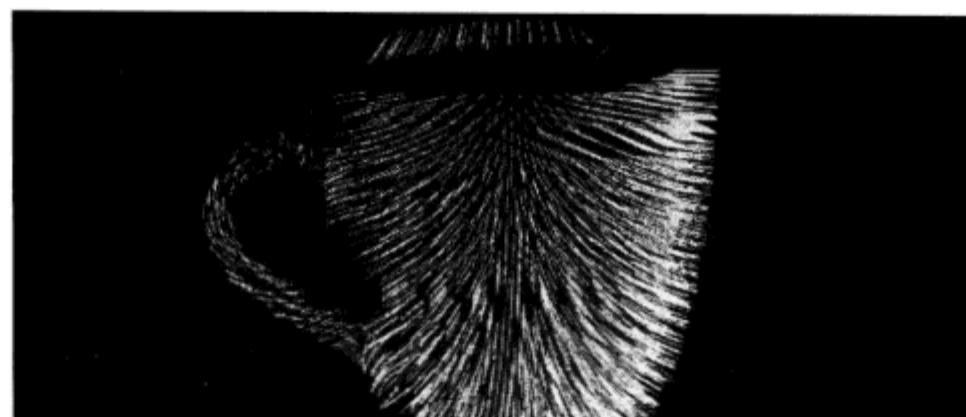


图 8.9 单个茶杯模型法向量

### 3) 例子 3

```
pcd_viewer data/bun000.pcd data/bun045.pcd -ax 0.5 -ps 3 -ps 1
```

上述命令假设 `bun000.pcd` 和 `bun045.pcd` 数据集已经下载并可用。在按键 U 和 G 之后将可以使用查找表和网格化显示功能，结果如图 8.10 所示。



图 8.10 网格化显示兔子模型

## 4. 距离图像可视化器

使用如下命令可以实现快速显示距离图像的功能：

```
tutorial_range_image_visualization data/office_scene.pcd
```

上述命令将加载文件 `office_scene.pcd`，并创建可视化距离图像，可视化图像可以同时显示点云和距离图像。

## 8.3 PCL 与 Kinect 连接

作为中间件，OpenNI 是 PCL 首选的硬件驱动，为 PCL 应用程序提供可靠的数据流。OpenNI 指开放自然交互联盟，该组织由 PrimeSense 于 2010 年底发起。OpenNI 目前致力于构建自然交互应用程序框架，该框架包括与底层视频和音频传感器进行通信的接口以及建立在其上的高级中间件构件，如视觉计算和语音识别等技术。

目前与 OpenNI 兼容性较好的底层硬件有 Primesense Reference Design、Microsoft Kinect 和 Asus Xtion Pro 等。

本小节主要介绍如何利用 OpenNI 连接 Kinect 为 PCL 提供可靠的原始数据源。OpenNI 使用工作节点来描述自己的工作流，一个工作节点可以调用底层节点提供的数据甚至更改节点配置，并能将数据传递至上层节点，若干个工作节点有效串联就构成了一个工作链。

当然，OpenNI 还支持强大的基于外置 XML 文件的节点配置方法，降低代码复杂度，提高应用程序可用性和灵活性。有一点需要注意的是，Kinect 采集图像的有效区间在 2~6m，因此在把有效数据输入到 PCD 文件保存之前需要排除野值点。

在 PCL 中，这种机制变得简单又高效，一个利用 OpenNI 通过 Kinect 生成深度图像的程序代码如下：

```
1 #include <stdlib.h>
2 #include <iostream>
3 #include <string>
4 #include <vector>
5 #include <pcl/io/pcd_io.h>
6 #include <pcl/point_types.h>
7 #include <XnCppWrapper.h>
8
9 using namespace std;
10 void CheckOpenNIError(XnStatus eResult, string sStatus)
11 {
12     if(eResult != XN_STATUS_OK)
13         cerr << sStatus << " Error: " << xnGetStatusString
14             (eResult) << endl;
15
16 //新建一个点云的数据结构
17 struct SColorPoint3D
18 {
19     float X;
20     float Y;
21     float Z;
22     float R;
23     float G;
24     float B;
```

```
25     SColorPoint3D(XnPoint3D pos, XnRGB24Pixel color)
26     {
27         X = pos.X;
28         Y = pos.Y;
29         Z = pos.Z;
30         R = (float)color.nRed / 255;
31         G = (float)color.nGreen / 255;
32         B = (float)color.nBlue / 255;
33     }
34 }
35
36 //产生点云vPointCloud
37 void GeneratePointCloud(xn::DepthGenerator& rDepthGen,xn::
38     ImageGenerator& rImageGen,const XnDepthPixel* pDepth,
39     const XnRGB24Pixel* pImage,vector<SColorPoint3D>&
40     vPointCloud)
41 {
42     pcl::PointCloud<pcl::PointXYZ> cloud;
43     //pcl::PointCloud<pcl::PointXYZRGB> cloud;
44
45     // 1. 提取图像像素
46     xn::DepthMetaData mDepthMD;
47     xn::ImageMetaData mImageMD;
48     rDepthGen.GetMetaData(mDepthMD);
49     //rImageGen.GetMetaData(mImageMD);
50
51     unsigned int uPointNum = mDepthMD.FullXRes()*mDepthMD.
52         FullYRes();
53     // 2. 建立转换之前数据结构
54     XnPoint3D*pDepthPointSet = new XnPoint3D[uPointNum];
55     XnRGB24Pixel*pImagePointSet = new XnRGB24Pixel
56         [uPointNum];
57
58     unsigned int i, j, idxShift, idx;
59     for(j = 0; j < mDepthMD.FullYRes(); ++j)
60     {
61         idxShift = j * mDepthMD.FullXRes();
62         for(i = 0; i < mDepthMD.FullXRes(); ++i)
63         {
64             idx = idxShift + i;
```

```
59
60         pDepthPointSet [idx].X = i;
61         pDepthPointSet [idx].Y = j;
62         pDepthPointSet [idx].Z = pDepth [idx];
63     }
64 }
65 // 3. 将图像坐标转换为真实世界三维坐标
66 XnPoint3D* p3DPointSet = new XnPoint3D[uPointNum];
67 rDepthGen.ConvertProjectiveToRealWorld(uPointNum,
68     pDepthPointSet, p3DPointSet);
69 delete [] pDepthPointSet;
70
71 //4. 建立点云
72 for(i = 0; i < uPointNum; ++ i)
73 {
74     if(p3DPointSet [i].Z == 0)
75         continue;
76     vPointCloud.push_back(SColorPoint3D(p3DPointSet [i],
77                                         pImage[i]));
78 }
79 //5. 生成文件
80 cloud.width      = vPointCloud.size();
81 cloud.height     = 1;
82 cloud.is_dense  = false;
83 cloud.points.resize (vPointCloud.size());
84
85 for (size_t i = 0; i < cloud.points.size (); ++i)
86 {
87     cloud.points[i].x = vPointCloud[i].X;
88     cloud.points[i].y = vPointCloud[i].Y;
89     cloud.points[i].z = vPointCloud[i].Z;
90
91 }
92
93 pcl::io::savePCDFileASCII ("F:\\test_pcd.pcd", cloud);
94
95 delete [] p3DPointSet;
```

```
96 }
97 int main(int argc, char** argv)
98 {
99     XnStatus eResult = XN_STATUS_OK;
100
101    xn::Context mContext;
102    eResult = mContext.Init();
103    CheckOpenNIError(eResult, "initialize context");
104
105    xn::DepthGenerator mDepthGenerator;
106    eResult = mDepthGenerator.Create(mContext);
107    CheckOpenNIError(eResult, "Create depth generator");
108
109    xn::ImageGenerator mImageGenerator;
110    eResult = mImageGenerator.Create(mContext);
111    CheckOpenNIError(eResult, "Create image generator");
112
113    XnMapOutputMode mapMode;
114    mapMode.nXRes = 640;
115    mapMode.nYRes = 480;
116    mapMode.nFPS = 30;
117    eResult = mDepthGenerator.SetMapOutputMode(mapMode);
118    eResult = mImageGenerator.SetMapOutputMode(mapMode);
119
120    mDepthGenerator.GetAlternativeViewPointCap().
121        SetViewPoint(mImageGenerator);
122
123    eResult = mContext.StartGeneratingAll();
124
125    eResult = mContext.WaitAndUpdateAll();
126
127    if(eResult == XN_STATUS_OK)
128    {
129        const XnDepthPixel* pDepthMap = mDepthGenerator.
130            GetDepthMap();
131
132        const XnRGB24Pixel* pImageMap = mImageGenerator.
133            GetRGB24ImageMap();
134
135    }
```

```

132         vector<SColorPoint3D> *vPointCloud = new vector<
133             SColorPoint3D>;
134
134     GeneratePointCloud(mDepthGenerator, mImageGenerator,
135                         pDepthMap, pImageMap, *vPointCloud);
135 }
136
137     mContext.StopGeneratingAll();
138     mContext.Shutdown();
139     return 0;
140 }
```

如果用 `pcl::PointCloud<pcl::PointXYZRGB> cloud` 语句还可以基于 Kinect 采集的彩色图像建立真彩色点云, 图 8.11 是 3D 点云与 2D 彩色图像对比显示效果。



图 8.11 点云图与原始图对比

## 8.4 例 子

本节给出一个在 ROS 中使用 Kinect 和 PCL 的实例。

首先定位到 `ROS_PACKAGE_PATH` 的路径下, 创建功能包:

```
roscreate-pkg my_pcl_tutorial pcl pcl_ros roscpp sensor_msgs;
```

然后在 `src` 文件夹里创建 `segmentation.cpp` 文件，在 `cloud_cb` 中添加如下代码：

```
1 //Convert the sensor_msgs/PointCloud2 data to pcl/  
2   PointCloud  
3  
4 //Create the filtering object  
5 sor.setFilterFieldName ("z");  
6 sor.setFilterLimits (0.0, 3.0);  
7 sor.setLeafSize (0.01, 0.01, 0.01);  
8 sor.setInputCloud (cloud.makeShared ());  
9 sor.filter (cloud_se);  
10  
11 // Init  
12 seg.setOptimizeCoefficients (true);  
13 seg.setModelType (pcl::SACMODEL_PLANE);  
14 seg.setMethodType (pcl::SAC_RANSAC);  
15 seg.setDistanceThreshold (0.01);  
16  
17 //segmentation  
18 seg.setInputCloud (cloud_se.makeShared ());  
19 seg.segment (*inliers, *coefficients);  
20 extract_.setNegative(true);  
21 extract_.setIndices (inliers);  
22 extract_.filter(*cloud_seg_prism);  
23  
24 // Publish the cloud_filtered  
25 pcl::toROSMsg (*cloud_seg_prism, cloud_filtered);  
26 pub.publish(cloud_filtered);  
27  
28 private:  
29  
30 sensor_msgs::PointCloud2 cloud_filtered;  
31 pcl::PointCloud<pcl::PointXYZRGB> cloud;  
32 pcl::PointCloud<pcl::PointXYZRGB> cloud_se;  
33 pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_seg_prism;  
34  
35 pcl::VoxelGrid<sensor_msgs::PointCloud2> sor;  
36 pcl::SACSegmentation<pcl::PointXYZRGB> seg;
```

```
37 pcl::ExtractIndices<PointType> extract_;  
38 pcl::ModelCoefficients::Ptr coefficients (new pcl::  
    ModelCoefficients());  
39 pcl::PointIndices::Ptr inliers (new pcl::PointIndices());
```

随后在 CMakeLists.txt 里面添加语句：

```
rosbuild_add_executable (segmentation src/ segmentation.cpp);
```

终端输入命令 rosmake my\_pcl\_tutorial， 在新建的 bin 文件夹下生成可执行程序 segmentation：

```
[ rosmake ] Results:  
[ rosmake ] Built 50 packages with 0 failures.  
[ rosmake ] Summary output to directory
```

在新的终端分别输入命令：

```
rosrun openni_camera openni_node
```

```
rosrun pcl_ros convert_pointcloud_to_image input:=/camera/rgb/  
points output:=/camera/rgb/cloud_image
```

```
1  
rosrun image_view image_view image:=/camera/rgb/cloud_image
```

```
rosrun my_pcl_tutorial segmentation input:=/camera/rgb/points
```

```
rosrun rviz rviz
```

原始图像如图 8.12 所示，运行结果如图 8.13 和图 8.14 所示。



图 8.12 原始图像



图 8.13 rviz 中图像

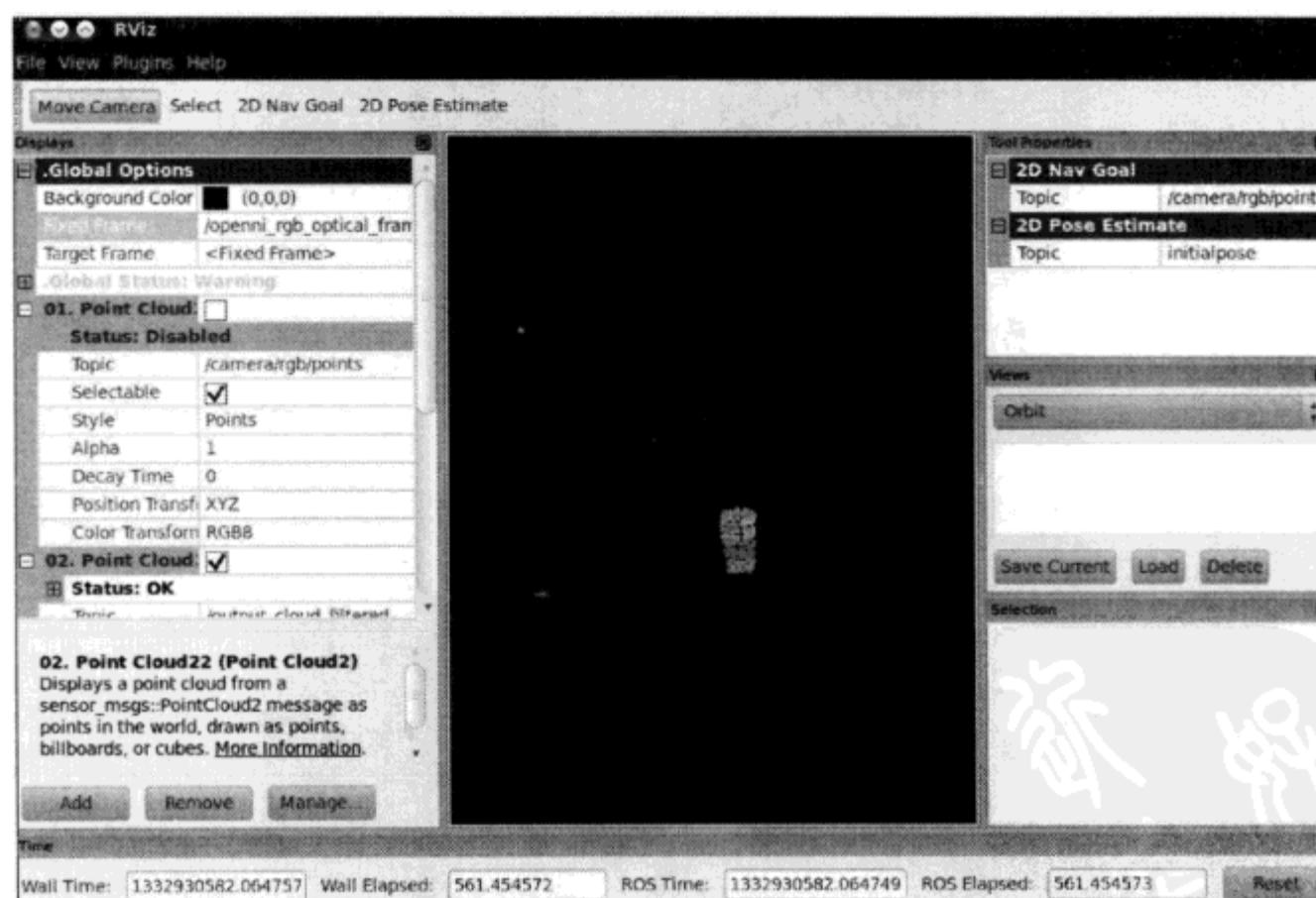


图 8.14 点云与原始彩色图合并

## 第九章 综合演示示例

本章内容主要针对 ROS 中涉及的主要内容，在中国科学院深圳先进技术研究院认知技术研究中心研制的机器人移动平台上进行实验。实验有三个，分别是机器人 SLAM (即时定位与地图构建)、自主导航和机器人识别并抓取物体。

### 9.1 实验一：SLAM (即时定位与地图构建)

SLAM 问题可以描述为：机器人在未知环境中从一个未知位置开始移动，在移动过程中根据位置估计和地图进行自身定位，同时在自身定位的基础上建造增量式地图，实现机器人的自主定位和导航。机器人采用 ROS 下的 Gmapping 软件包来实现 SLAM 算法。实验在本实验室研发的自主移动机器人上实现，机器人移动平台采用了轮式驱动方式，如图 9.1 所示。

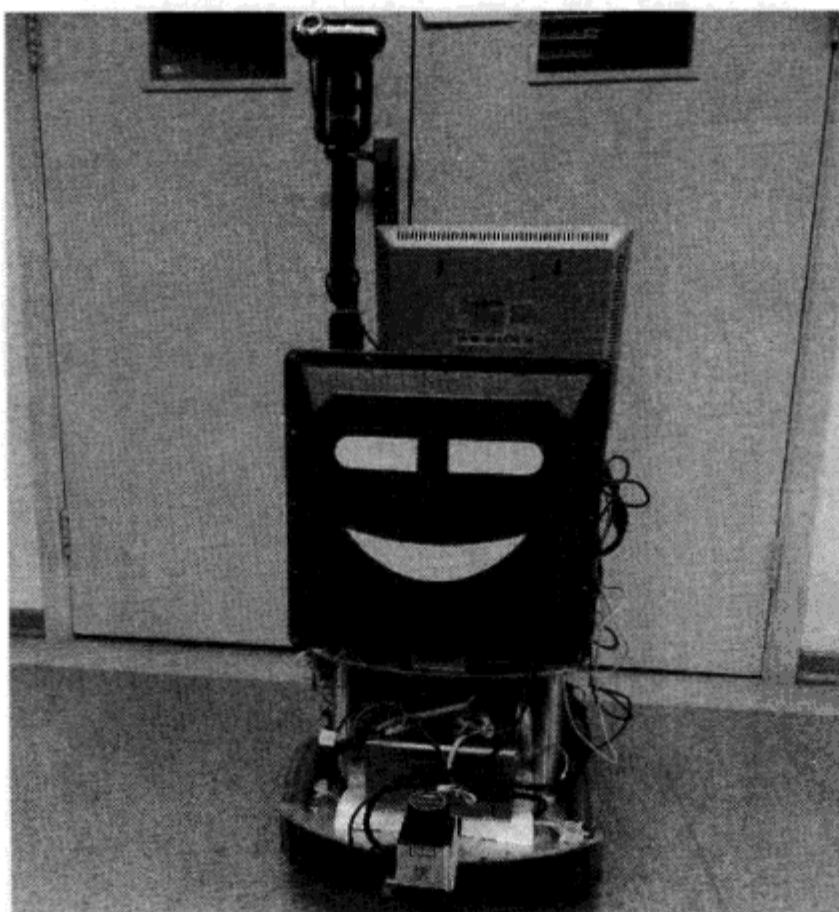


图 9.1 移动机器人平台

机器人采用激光、陀螺仪、里程计等传感器获取信息，为机器人提供自身的  
位置和外界环境的知识。

机器人使用的主要硬件列表如下：

1) 激光扫描仪

- 型号：Hokuyo URG-04LX 2D 激光扫描仪（见图 9.2）
- 电源：5VDC
- 测量距离：60-4095mm
- 扫描时间：100msec/scan
- 接口：USB/232

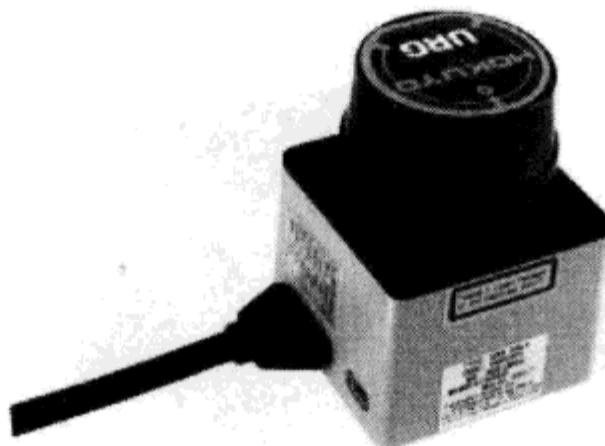


图 9.2 Hokuyo URG-04LX 2D 激光扫描仪

2) 里程计（见图 9.3）

- 电机：maxon motor
- 硬件控制器：CAN Interface
- 驱动器：Accelnet Micro Panel
- 数据接口：CANopen/DeviceNet/RS-232
- 软件库：Copley Motion Library(CML)、CAN Interface Driver

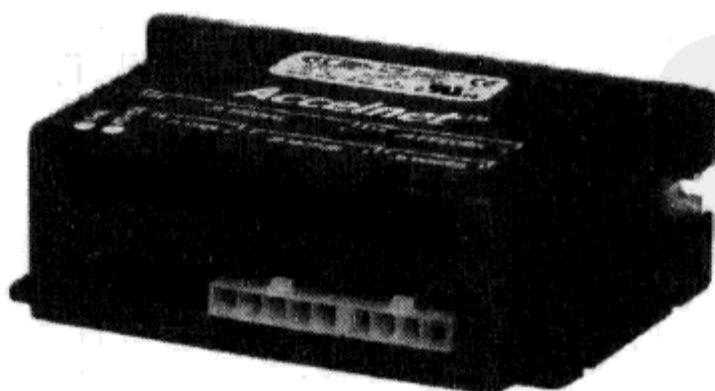


图 9.3 Copley 驱动器

3) 陀螺仪

- 型号：CruizCore XG1010 单轴 MEMS 数字陀螺仪（见图 9.4）

- 数据接口: RS232/USB
- 坐标系: 顺时针方向
- 测量范围:  $\pm 100^\circ/\text{s}$
- 带宽: 50Hz
- 输出速率: 100Hz
- 偏置漂移:  $10^\circ/\text{hrz}$

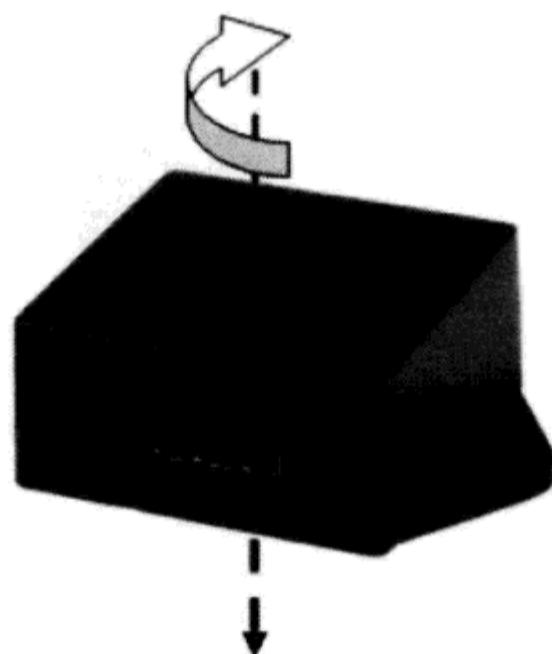


图 9.4 CruizCore XG1010

SLAM 实验采用前面所提到的移动机器人平台, 让机器人沿着实验室走廊根据避障功能包提供的路径搜索算法寻找一条路径建立二维栅格地图。机器人在如图 9.5 所示的环境中按带箭头折线所示的路径进行地图建立。

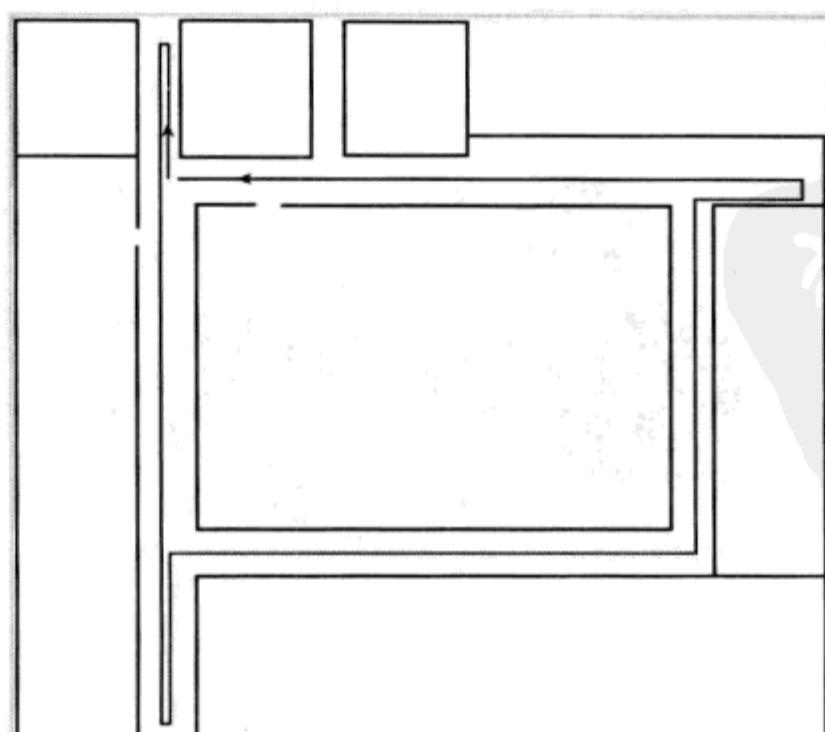


图 9.5 机器人建立地图的路径

实验过程中需要启动的节点 launch 文件依次如下：

1. 启动机器人硬件和路径探索功能包文件：roslaunch explore.launch

启动文件 explore.launch 源代码如下：

```
1 <launch>
2     <node pkg="hokuyo_node" type="hokuyo_node" name="scan">
3         <param name="port" value="/dev/robot/laser"/>
4         <param name="max_ang_degrees" value="90"/>
5         <param name="min_ang_degrees" value="-90"/>
6         <param name="frame_id" value="base_scan"/>
7     </node>
8     <node pkg="tf" type="static_transform_publisher" name=
9         "laser_static_tf_publisher" args="0.14 0 0.06 0 0 0
10        1 base_link base_scan 50"/>
11     <node pkg="tf" type="static_transform_publisher" name=
12         "webcam_static_tf_publisher" args="0.205 -0.035 0 0
13        0 0 1 base_link head_camera 50"/>
14     <node pkg = "avoid_obstacle" type=" avoid_obstacle"
15        name = "avoid_obstacle ">
16     </node>
17     <node pkg="gyro" type="gyro" name="gyro">
18     </node>
19     <node pkg="motion" type="motion" name="odom_publisher
20        ">
21     </node>
22     <node pkg="usb_cam" name="usb_cam" type="usb_cam_node">
23     </node>
24 </launch>
```

节点配置：

(1) /scan: 通过 USB 采集 Hokuyo 激光传感器的数据，并在 ROS 上发布激光扫描信息。

(2) /laser\_static\_tf\_publisher: 发布激光传感器相对于机器人中心的坐标变换关系。

(3) /webcam\_static\_tf\_publisher: 发布 webcamer 相对于机器人中心的坐标变换关系。

(4) `/avoid_obstacle`: 订阅 `/scan` 节点发布的激光传感器的数据，根据激光传感器反馈的信息发布消息控制电机进行环境探索和避障。

(5) `/gyro`: 利用 RS232 串口，采集陀螺仪的数据，并在 ROS 上发布陀螺仪的数据。

(6) `/odom`: 订阅 `/avoid_obstacle` 节点发布的电机控制消息控制电机，同时将里程计采集的编码器数据计算处理后发布出去。

(7) `/usb_cam_node`: 发布摄像头传感器信息。

## 2. 启动地图建立功能包: `roslaunch build_map.launch`

```

1 <launch>
2   <node pkg="gmapping" type="slam_gmapping" name=
3     "map_builder">
4     <param name="xmin" value="-10.0"/>
5     <param name="ymin" value="-10.0"/>
6     <param name="xmax" value="10.0"/>
7     <param name="ymax" value="10.0"/>
8     <param name="delta" value="0.025"/>
9     <param name="temporalUpdate" value="2.0"/>
10    </node>
11  </launch>

```

节点配置:

`/map_builder`: 接收 `/scan` 节点发布的激光传感器数据和 `/odom` 节点发出的里程计计算出的坐标信息，建立二维栅格地图，其中 Gmapping 功能包的参数配置如下：

- (1) `xmin=-10.0`: 地图上  $x$  轴最小坐标为  $-10\text{m}$ 。
- (2) `ymin=-10.0`: 地图上  $y$  轴最小坐标为  $-10\text{m}$ 。
- (3) `xmax=10.0`: 地图上  $x$  轴最大坐标为  $10\text{m}$ 。
- (4) `ymax=10.0`: 地图上  $y$  轴最大坐标为  $10\text{m}$ 。
- (5) `delta=0.0025`: 地图的分辨率为  $0.025\text{m}$ 。
- (6) `temporalUpdate=2.0`: 当间隔扫描时间低于  $2\text{s}$  时，进行新的一次扫描。

## 3. 启动 `rviz` 包，实时显示机器人建立的地图: `rosrun rviz rviz`

在 `shell` 中启动 `explore.launch` 文件启动上述节点后，节点运行图如图 9.6 所示。图 9.7 是机器人按图 9.5 所示路径扫描完后，所得到的地图，从图中可



图 9.6 SLAM 运行节点图

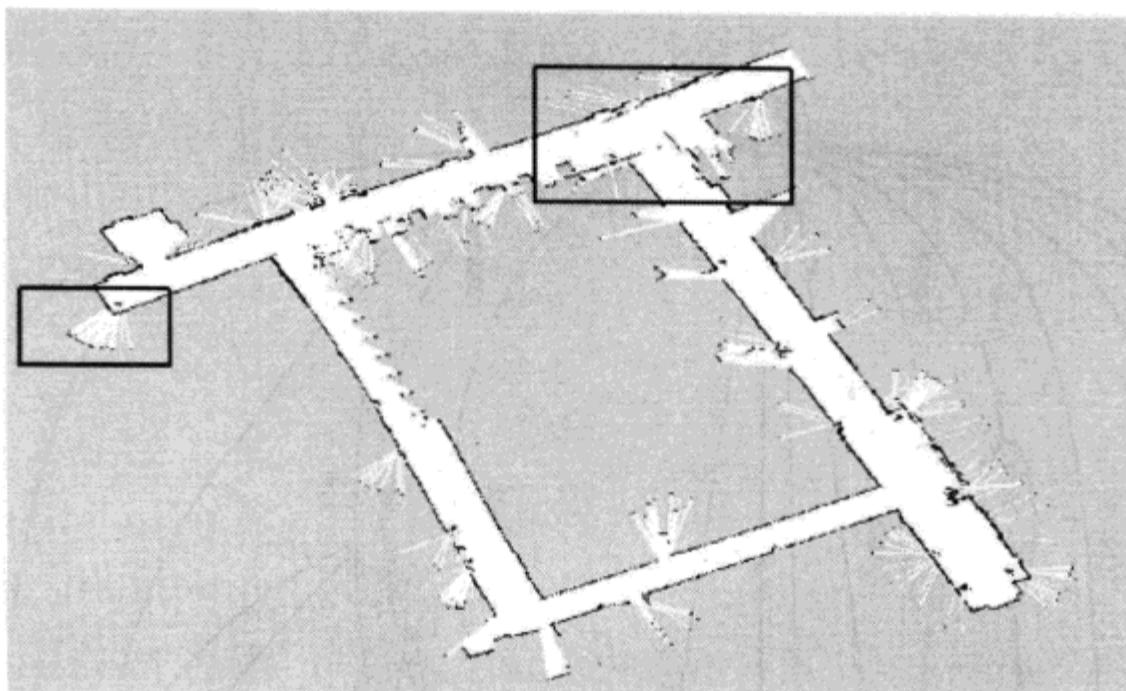


图 9.7 最终地图

以看出，机器建立的环状地图不能完全闭合，引起这个结果的主要原因是机器人定位误差，同时激光扫描的反射率高的物体，如玻璃、瓷器等也影响地图的真实度。图 9.8 是 SLAM 过程中屏幕截图。



图 9.8 建立地图过程中截图

## 9.2 实验二：机器人导航

移动机器人根据应用环境分为室内机器人和室外机器人，无论是哪种机器人，在它的运动过程中始终要求解决自身的导航与定位问题，也就是

Durrant Whyte 提出的三个问题：① 我现在何处？② 我要往何处去？③ 要如何到该处去？其中问题 ① 是移动机器人导航系统中的定位问题，② 和 ③ 是移动机器人导航系统中的路径规划问题。本导航实验基于上节 SLAM 实验的机器人硬件，对 ROS 中的 `navigation` 导航功能包集进行配置，并在机器人平台上实现。

## 1. 基于 ROS 的导航算法实现

下面分别介绍导航功能包集里的两个重点功能包：`AMCL` 包和代价地图包。

1) `AMCL` 包 (`adaptive monte carlo localization approach`, 自适应蒙特卡罗定位方法)

该包是移动机器人基于概率的 2D 定位系统。它采用自适应蒙特卡罗定位方法，用粒子滤波来估测机器人相对于已知地图的位姿。`AMCL` 将传入的激光传感器的数据转化到里程计坐标系 (`odom_frame_id`)。因此必须存在激光传感器和里程计坐标系的转换 `tf` 信息。激光在第一次扫描后，`AMCL` 包会寻找激光坐标系和本体坐标系之间的转换关系，然后将其锁存起来，`AMCL` 不能处理和本体有相对运动的激光平台。用里程计定位和用 `AMCL` 定位的区别可在地图中体现出来。在程序运行的过程中，`AMCL` 估计移动平台坐标系和全局坐标系之间的坐标转换关系。

### 2) 代价地图包

导航功能包集用代价地图来存储障碍物的信息，代价地图自动获取传感器的信息来进行自我更新。传感器被用来在地图中标记障碍信息、清除障碍信息。标记信息的过程其实是更改单元值的过程。如图 9.9 所示，在代价地图中红色栅格单元代表障碍，蓝色的单元代表由机器人半径而预留出的单元，绿色的圆圈代表机器人的轨迹。为了避免碰撞，机器人的轨迹不能与红色的单元交叉，机器人的中心也不能与蓝色的单元交叉。这个包提供的结构用传感器信息来存储或更新代价地图里的障碍信息。`costmap_2d` 类的 `Costmap2DROS` 对象提供两维的接口，传感器用来更新插入障碍物信息占用的单元或清除障碍物占用单元的信息。按照下面的步骤，我们在自己的机器人上实现了导航功能包集的配置。

首先使用启动文件 `my_robot_configuration.launch` 来启动所有的硬件节点，代码如下：

```
1 <launch>
2   <node pkg="hokuyo_node" type="hokuyo_node" name="scan"
      output="screen">
3   <param name="frame_id" value="laser_frame"/>
```

```

4   </node>
5   <node pkg="remote_control" name="remote_control" type=
     "remote_control">
6   </node>
7   <node pkg="motion_new" type="motion_navigation" name=
     "odom_publisher" output="screen">
8   </node>
9   <node pkg="tf" type="static_transform_publisher" name=
     "laser_static_tf_publisher" args="0.15 0 0.16 0 0 0 1
     base_link laser_frame 50"/>
10  <node pkg="move_gyro" type="talker" name="gyro">
11  </node>
12 </launch>

```

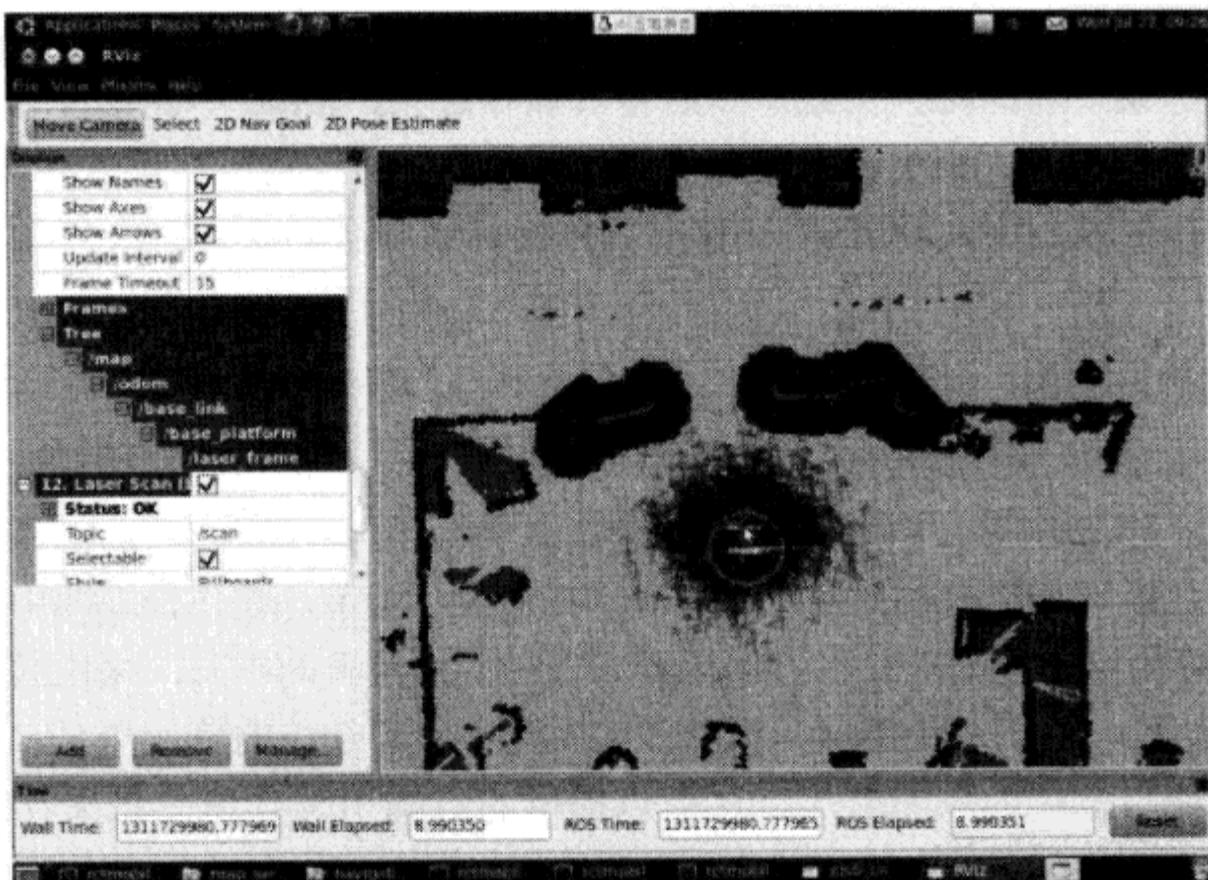


图 9.9 导航初始截图

然后用 `move_base.launch` 文件启动导航相关的功能包：

```

1 <launch>
2   <master auto="start"/>
3   <!-- Run the map server -->
4   <node name="map_server" pkg="map_server" type="map_server"
     " args="$(find map_server)/map.yaml"/>
5   <!-- Run AMCL -->

```

```
6 <include file="$(find amcl)/examples/amcl_omni.launch" />
7   <node pkg="move_base" type="move_base" respawn="false"
8     name="move_base" output="screen">
9     <rosparam file="$(find navigation_try)/
10       costmap_common_params.yaml" command="load" ns=
11         "global_costmap" />
12     <rosparam file="$(find navigation_try)/
13       costmap_common_params.yaml" command="load" ns=
14         "local_costmap" />
15     <rosparam file="$(find navigation_try)/
16       local_costmap_params.yaml" command="load" />
17     <rosparam file="$(find navigation_try)/
18       global_costmap_params.yaml" command="load" />
19     <rosparam file="$(find navigation_try)/
20       base_local_planner_params.yaml" command="load" />
21   </node>
22 </launch>
```

实验中节点配置如下：

- (1) **/scan**: 利用网口采集激光数据，并将其发送出去。
- (2) **/odometry\_publisher**: 发布里程计数据，其中里程计数据采用编码器的值，角度采用从 **/gyro** 订阅的数据，这个节点发布了 **/odom->/base\_link** 的 **tf** 信息以及用于地图建立的里程计数据；另外，这个节点还提供运动控制的服务。
- (3) **/tf**: 将激光传感器坐标系和机器人本体坐标系之间的相对关系发布出去。
- (4) **/gyro**: 利用串口 RS232，采集陀螺仪数据并在 ROS 中发布陀螺仪数据。
- (5) **/map\_server**: 加载二维栅格地图并将其信息发送出去。
- (6) **/move\_base**: 订阅机器人硬件发布的传感器信息，用机器人导航算法将转换后的电机执行信息发送到电机驱动器。

## 2. 导航功能包集各个参数设置

- (1) 设置 **base\_local\_planner\_params.yaml** 的内容及各参数。

```
1 obstacle_range: 2.5
2 raytrace_range: 3.0
```

```

3 robot_radius: 0.16
4 Robot_radius: 0.55
5 inflation_radius: 0.16
6 observation_sources: scan
7 scan: {sensor_frame: laser_frame, data_type: LaserScan,
          topic: scan, marking: true, clearing: true}

```

(2) 设置 `global_costmap_params.yaml` 源代码及各参数。

```

1 global_costmap:
2   global_frame: /map
3   robot_base_frame: base_link
4   update_frequency: 5.0
5   static_map: true

```

(3) 设置 `local_costmap_params.yaml` 源代码及各参数。

```

1 local_costmap:
2   global_frame: odom
3   robot_base_frame: base_link
4   update_frequency: 5.0
5   publish_frequency: 2.0
6   static_map: false
7   rolling_window: true
8   width: 6.0
9   height: 6.0
10  resolution: 0.05

```

### 3. 实验结果

机器人导航实验是在上面建立地图的实验室环境中，基于 SLAM 建立的二维栅格地图，实验中需要启动的节点的 `launch` 文件如下：

- (1) 启动机器人硬件节点：`roslaunch my_robot_configuration.launch`。
- (2) 启动导航算法程序节点：`roslaunch move_base.launch`。
- (3) 启动 rviz 界面：`roslaunch rviz rviz`。

如图 9.9 所示，先设置机器人在地图中的初始位置，图中圆圈表示机器人的初始位置。再设置机器人的目标位置，如图 9.10 所示。红色的箭头代表机器人的目标位姿，箭头方向代表机器人的目标方向，箭头的末端代表机器人的目标位置。在设置机器人的目标位姿后，根据导航算法，机器人规划出一条无碰撞的全局路径，绕过地图上显示的障碍物，如图 9.11 中绿线所示。然后机器人从起点

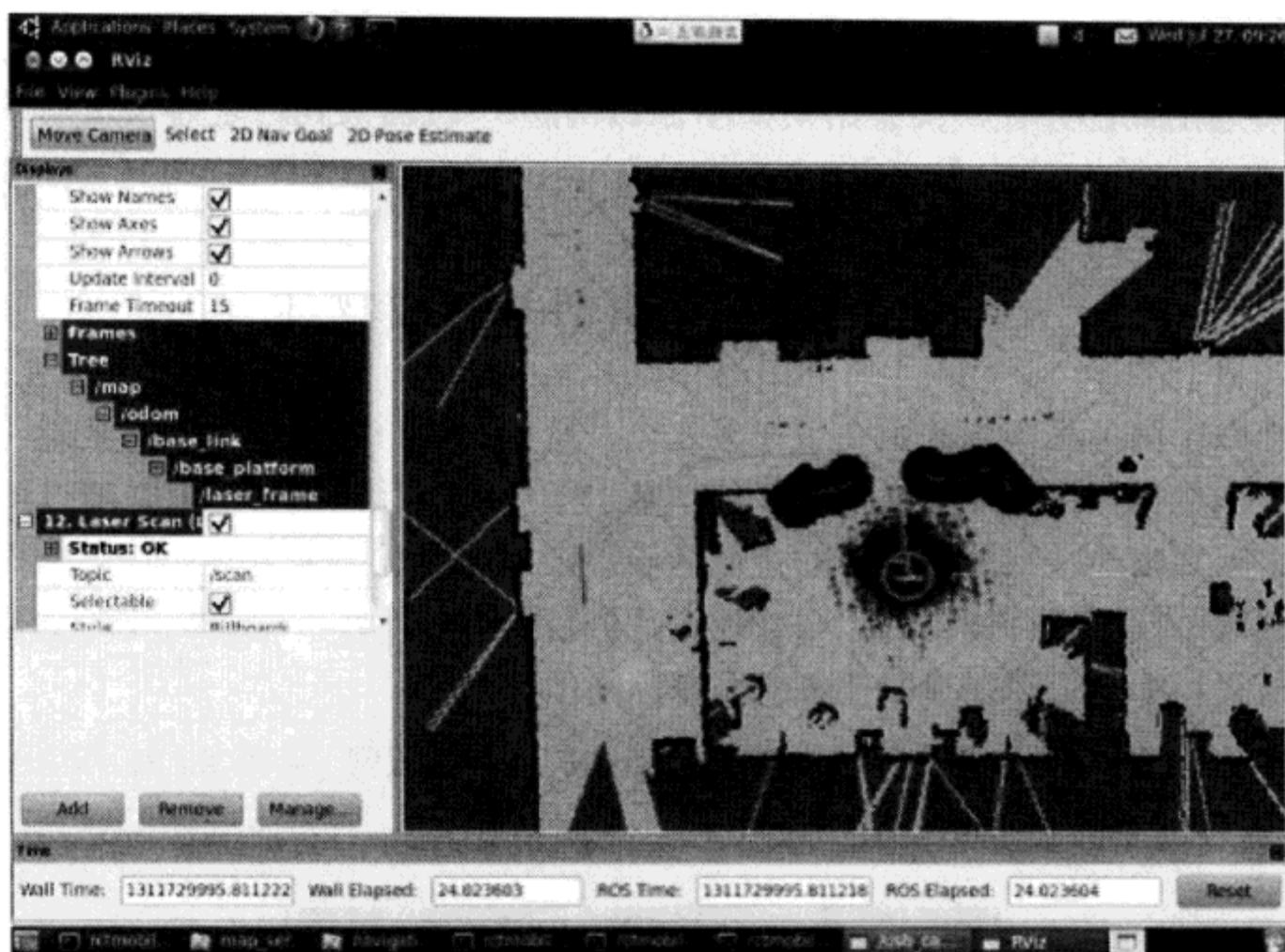


图 9.10 导航过程截图



图 9.11 导航结束截图

出发，沿着之前规划出的全局路径前进，在移动过程中实时检测周围的环境，并动态规划出局部路径。实验结果如图 9.11 所示，机器人到达之前设定的终点目标位置。

### 9.3 实验三：识别并抓取物体

抓取是服务机器人的重要子任务之一，是完成复杂的服务任务的基础。本实验基于图像目标识别，实现抓取任务。基于图像的目标识别与抓取是通过图像的特征来识别物体，将目标物体的特征从模板图片中提取出来，然后在摄像机的视场中截取图像进行特征提取，与模板图片的特征进行比较。根据比较的结果，判断当前视场中是否存在抓取任务所需目标物体。

实验中用到的主要硬件平台如下：Kuka 工业机械臂（见图 9.12）和 Bumblebee2 双目摄像机（见图 9.13）。

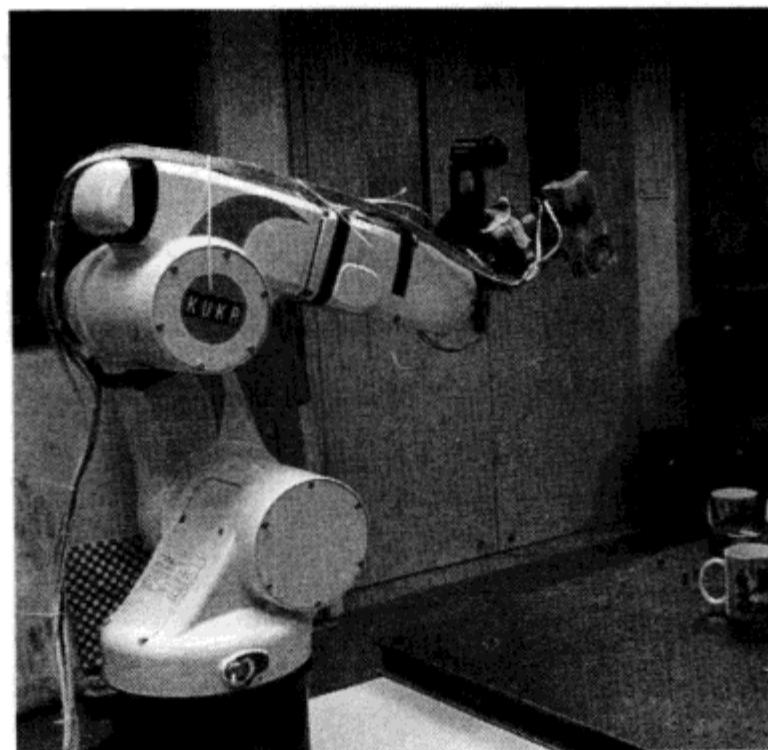


图 9.12 Kuka 工业机械臂

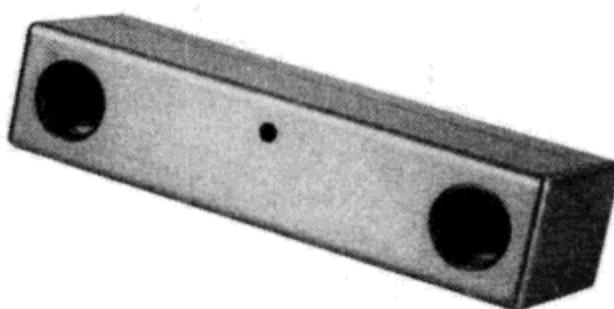


图 9.13 Bumblebee2 双目摄像机

本实验分为以下几个模块：目标检测模块、双目测距模块、机械臂运动控制模块、光学接近传感器和触觉传感器数据采集和手爪控制模块。首先是通过目标检测模块 (SURF 节点) 选定目标，在当前视场中进行目标检测，检测到目标后将目标的二维图像坐标传递给双目测距模块。双目摄像机 (Bumblebee 节点) 模块主要的任务是负责测距，接收到目标检测模块发送过来的数据之后，对深度信息图进行查询，获取目标的三维空间坐标信息。双目相机测量目标的位置与姿态，并将信息传递给机械臂运动控制节点 (Kuka 节点)，控制机械臂运动到预抓取的位姿，发送消息给二指手爪控制节点，实施抓取动作。在抓取过程中检测触觉传感器的信号，判断特征信息。实验中各个节点均通过 shell 命令终端输入启动指令运行，下面以抓取纸杯奶茶为例。图 9.14 为实验中运行程序界面。

- (1) 启动 ROS 的管理节点：`roscore`。
- (2) 启动双目摄像机节点：`rosrun bumblebee simplestero`。
- (3) 启动 SURF 目标检测模块：`rosrun surf surf`。
- (4) 启动机械臂控制模块：`rosrun kuka kuka`。

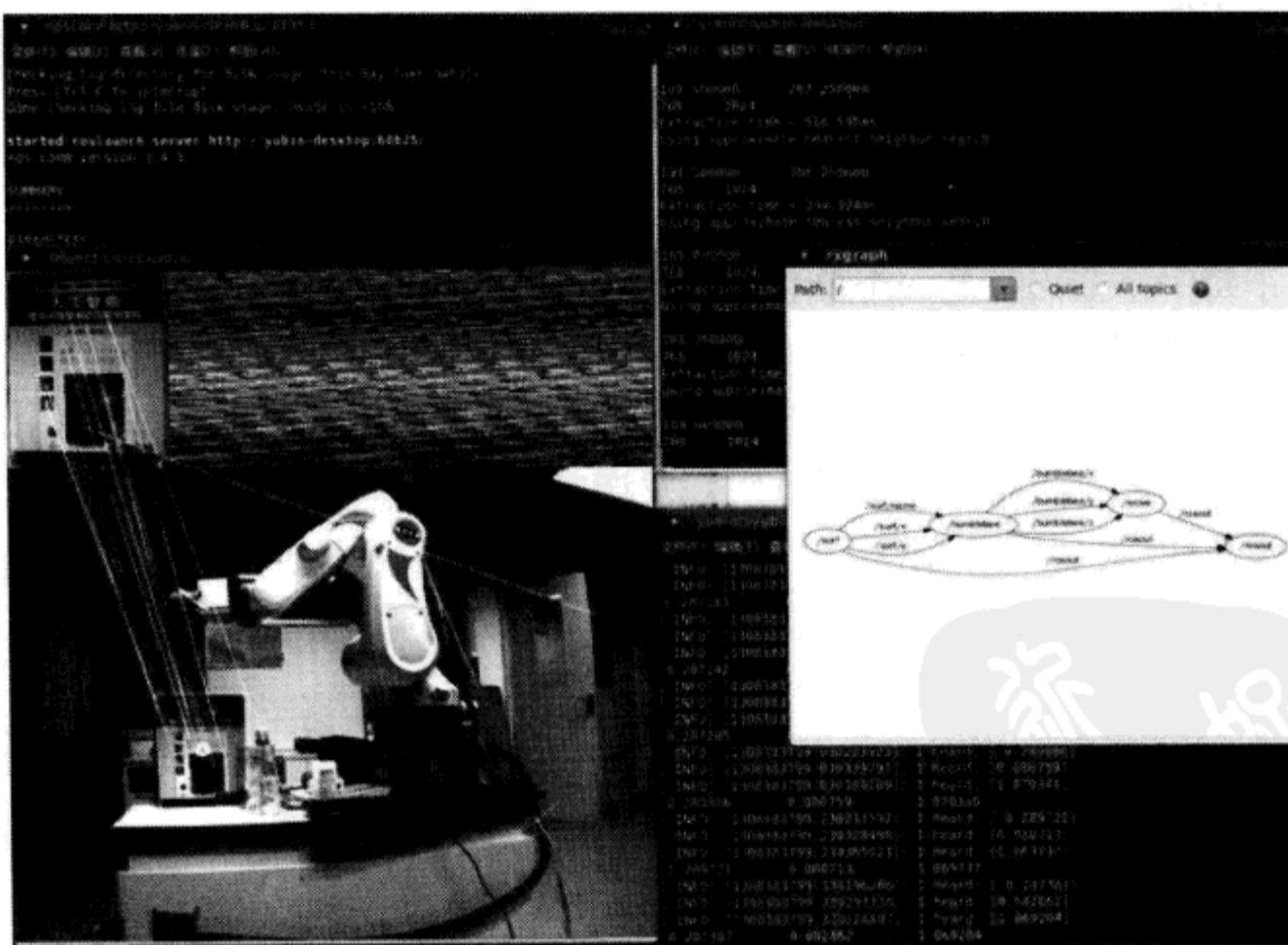


图 9.14 抓取任务运行界面

- (5) 启动机械手数据采集处理模块：`rosrun hand touch`。

通过运行 `shell` 命令，启动以上节点之后，使用图形化的程序管理工具，可

以得到的各节点运行时的动态消息传递图, 如图 9.15 所示, 可以看到图中一共包含五个节点, 管理节点负责管理其余的四个功能节点。

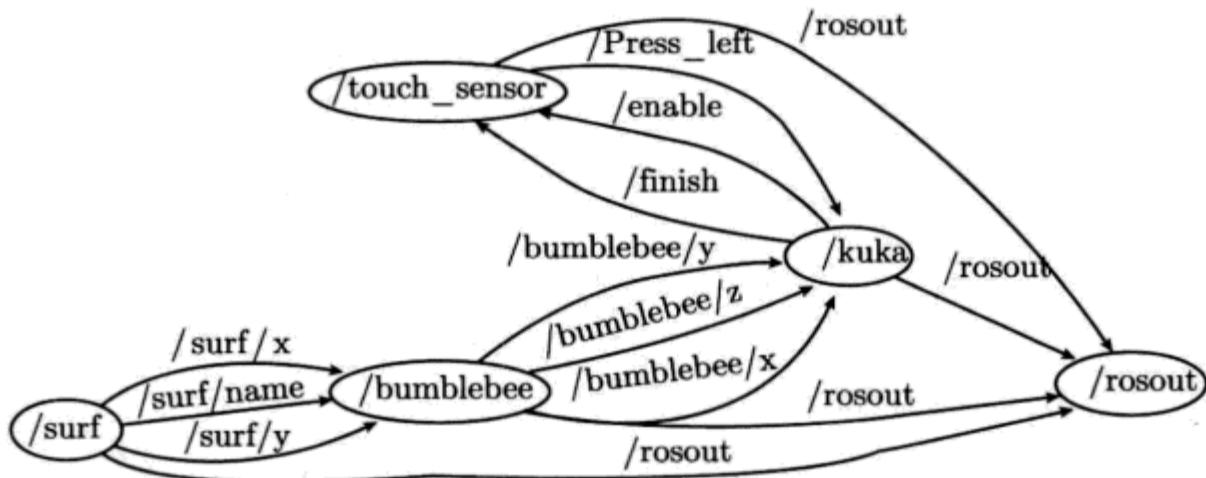


图 9.15 抓取任务节点运行图

其中节点配置如下:

- (1) **/Surf**: 图像匹配算法, 用于识别特定的目标物体。
- (2) **/Bumblebee**: 根据 SURF 算法识别物体后, 在指定区域, 计算出目标物体的位置信息。
- (3) **/Kuka**: 获取目标位置的位置之后, 进行坐标变换, 计算出到达目标位置的路径, 并控制 **/Kuka** 按照指定速度, 运动到目标位置。
- (4) **/Touch**: 夹持器的触觉信息采集, 用于判断抓取的状态。

由于每一个节点都是一个单独的程序, 相互之间通信的方式需要特定的数据格式。其中双目摄像机模块与目标检测模块之间存在三个数据传递, 目标的名称、目标的  $x$  坐标值与目标的  $y$  坐标值。其中名称使用字符串格式, 坐标值为浮点型数据格式, 每次循环只发送一组, 发送之后计数, 可以保证发出的和收到的数据能够匹配。

## 参 考 文 献

- [1] Willow Garage 公司网站. <http://www.willowgarage.com/> [2012-01-15].
- [2] ROS 网站. <http://www.ros.org/> [2012-01-15].
- [3] OpenRAVE 网 站. <http://openrave.programmingvision.com/en/main/index.html> [2012-01-15].
- [4] Player 网站. <http://playerstage.sourceforge.net/> [2012-01-15].
- [5] YARP 网站. <http://eris.liralab.it/yarp/> [2012-01-15].
- [6] Orococos 网站. <http://www.orocos.org/> [2012-01-15].
- [7] CARMEN 网站. <http://carmen.sourceforge.net/> [2012-01-15].
- [8] Orca 网站. <http://orca-robotics.sourceforge.net/> [2012-01-15].
- [9] MOOS 网站. <http://www.robots.ox.ac.uk/mobile/MOOS/wiki/pmwiki.php> [2012-01-15].
- [10] Microsoft Robotics Studio 网站. <http://www.microsoft.com/robotics/> [2012-01-15].
- [11] Python 网站. <http://www.ubuntu.com/> [2012-01-15].
- [12] Ubuntu 中文论坛. <http://forum.ubuntu.org.cn/> [2012-01-15].
- [13] Python 网站. <http://www.python.org/> [2012-01-15].
- [14] Python 网站. <http://python.cn/> [2012-01-15].
- [15] roslua 网站. <http://github.com/timn/roslua> [2012-01-15].
- [16] OpenCV 网站. <http://opencv.willowgarage.com/wiki/> [2012-01-15].
- [17] OpenSLAM 网站. <http://www.openslam.org/> [2012-01-15].
- [18] Kinect 体感游戏网. <http://www.cnkinect.com/> [2012-01-15].
- [19] PrimeSense 公司网站. <http://www.primesense.com/> [2012-01-15].
- [20] Fischler M A, Bolles R C. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM, 1981, 24(6): 381–395.
- [21] PCL 网站. <http://www.pointclouds.org/> [2012-01-15].
- [22] ROS 网站中 PCL 功能包. <http://www.ros.org/wiki/pcl> [2012-01-15].
- [23] Willow Garage 公司网站中 PCL 功能包. <http://www.willowgarage.com/pages/software/pcl> [2012-01-15].
- [24] 国际会议 2011 International Conference on Computer Vision 上关于 PCL 教程. <http://www.pointclouds.org/media/iccv2011.html> [2012-01-15].
- [25] Radu R B, Cousins S. 3D is here: Point Cloud Library (PCL) // 2011 IEEE International Conference on Robotics and Automation. Shanghai, China. 2011.

封面

书名

版权

前言

目录

## 第一章 ROS简介

- 1 . 1 ROS简介
- 1 . 2 ROS安装
- 1 . 3 ROS支持的机器人
- 1 . 4 ROS网上资源

## 第二章 ROS总体框架及基本命令

- 2 . 1 ROS总体框架
- 2 . 1 . 1 文件系统级
- 2 . 1 . 2 计算图级
- 2 . 1 . 3 社区级
- 2 . 1 . 4 更高层概念
- 2 . 1 . 5 名称
- 2 . 2 ROS基本命令
- 2 . 2 . 1 ROS文件系统命令
- 2 . 2 . 2 ROS核心命令
- 2 . 3 工具
  - 2 . 3 . 1 3D可视化工具：rviz
  - 2 . 3 . 2 传感器数据记录与可视化工具：rosbag和rxbag
  - 2 . 3 . 3 画图工具：rxtplot
  - 2 . 3 . 4 系统可视化工具：rxgraph
  - 2 . 3 . 5 rxconsole
  - 2 . 3 . 6 tf命令
- 2 . 4 例子
- 2 . 4 . 1 创建ROS消息和服务
- 2 . 4 . 2 记录和回放数据
- 2 . 4 . 3 手工创建ROS功能包
- 2 . 4 . 4 大项目上运行roslaunch
- 2 . 4 . 5 在多台机器上运行ROS系统
- 2 . 4 . 6 定义客户消息

## 第三章 ROS客户端库

- 3 . 1 概述
- 3 . 2 rosccp客户端库
  - 3 . 2 . 1 简单的主题发布者和主题订阅者
  - 3 . 2 . 2 简单的服务器端和客户端
  - 3 . 2 . 3 rosccp中参数的使用
  - 3 . 2 . 4 从节点句柄存取私有名称
  - 3 . 2 . 5 用类方法订阅和回调服务
  - 3 . 2 . 6 计时器
  - 3 . 2 . 7 带动态可重配置及参数服务器的主题发布者 / 订阅者节点 (C++)
  - 3 . 2 . 8 带动态可重配置及参数服务器的主题发布者 / 订阅者节点 (Python)
  - 3 . 2 . 9 组合C++ / Python主题发布者 / 订阅者节点
- 3 . 3 rospyc客户端库
  - 3 . 3 . 1 简单的主题发布者 / 订阅者
  - 3 . 3 . 2 简单的服务端和客户端
  - 3 . 3 . 3 rospyc中参数的使用
  - 3 . 3 . 4 rospyc中numpy的使用
  - 3 . 3 . 5 rospyc运行日志
  - 3 . 3 . 6 ROS Python Makefile文件
  - 3 . 3 . 7 设置PYTHONPATH
  - 3 . 3 . 8 发布消息

3.4 roslib客户端库

3.5 实验阶段的客户端库

3.5.1 rosjava

3.5.2 roslua

#### 第四章 OpenCV

4.1 image-common功能包集

4.1.1 image\_transport功能包

4.1.2 camera-calibration-parsers功能包

4.1.3 camerainfo\_manager功能包

4.1.4 polled-camera功能包

4.2 image-pipeline功能包集

4.3 vision\_opencv功能包集

4.3.1 opencv2

4.3.2 cv\_bridge

4.3.3 image-geometry

4.4 投影 tf 坐标系到图像 (C++)

4.5 演示例子

4.5.1 使用颜色追踪物体

4.5.2 识别物体

#### 第五章 SLAM和导航

5.1 使用 tf 配置机器人

5.2 通过ROS发布里程计信息

5.3 通过ROS发布传感器数据流

5.4 SLAM

5.4.1 SLAM简介

5.4.2 slam\_mapping功能包

5.4.3 使用记录的数据建立地图

5.4.4 模拟器中建立地图

5.4.5 模拟器中使用客户定制地图

5.5 配置和使用导航功能包集

5.5.1 导航功能包集基本操作

5.5.2 在机器人上设置和配置导航功能包集

5.5.3 rviz 与导航功能包集配合使用

5.5.4 发送目标到导航功能包集

#### 第六章 抓取操作

6.1 机器人手臂的运动规划

6.1.1 安装和配置

6.1.2 编译手臂导航功能包集

6.1.3 启动模拟器和仿真环境

6.1.4 启动相关节点

6.1.5 控制手臂运动

6.2 运动规划的环境表示

6.2.1 基于自滤波数据构建碰撞地图

6.2.2 检测关节轨迹碰撞

6.2.3 给定机器人状态下的碰撞检测

6.2.4 添加已知点到运动规划环境

6.2.5 添加物体到机器人本体

6.3 用于PR2机器人的运动学

6.3.1 从PR2运动学开始

6.3.2 从运动学节点获取运动学求解器信息

6.3.3 PR2手臂运动学正解

6.3.4 PR2手臂运动学逆解

6.3.5 PR2手臂无碰撞运动学逆解

6.4 用于PR2机器人的手臂的安全轨迹控制

6.5 使用轨迹滤波节点进行轨迹滤波

- 6 . 5 . 1 生成无碰撞三次样条轨迹
- 6 . 5 . 2 使用轨迹滤波服务器对关节轨迹进行滤波
- 6 . 5 . 3 学习如何创建自己的轨迹滤波
- 6 . 6 机器人状态和轨迹可视化

## 第七章 K i n e c t

- 7 . 1 K i n e c t 简介
- 7 . 2 安装驱动
  - 7 . 2 . 1 U b u n t u 系统上安装 K i n e c t
  - 7 . 2 . 2 基于源的安装
  - 7 . 3 测试
    - 7 . 3 . 1 测试 K i n e c t 彩色摄像机
    - 7 . 3 . 2 测试 K i n e c t 深度摄像机
    - 7 . 3 . 3 测试 K i n e c t 马达
  - 7 . 4 o p e n n i c a m e r a
  - 7 . 5 o p e n n i t r a c k e r

## 第八章 点云库

- 8 . 1 P C L 简介
  - 8 . 1 . 1 P C L 架构
  - 8 . 1 . 2 P C L 数据结构
  - 8 . 1 . 3 P C L 与 R O S 的集成
- 8 . 2 P C L 可视化库
- 8 . 3 P C L 与 K i n e c t 连接
- 8 . 4 例子

## 第九章 综合演示示例

- 9 . 1 实验一：S L A M (即时定位与地图构建)
- 9 . 2 实验二：机器人导航
- 9 . 3 实验三：识别并抓取物体

## 参考文献