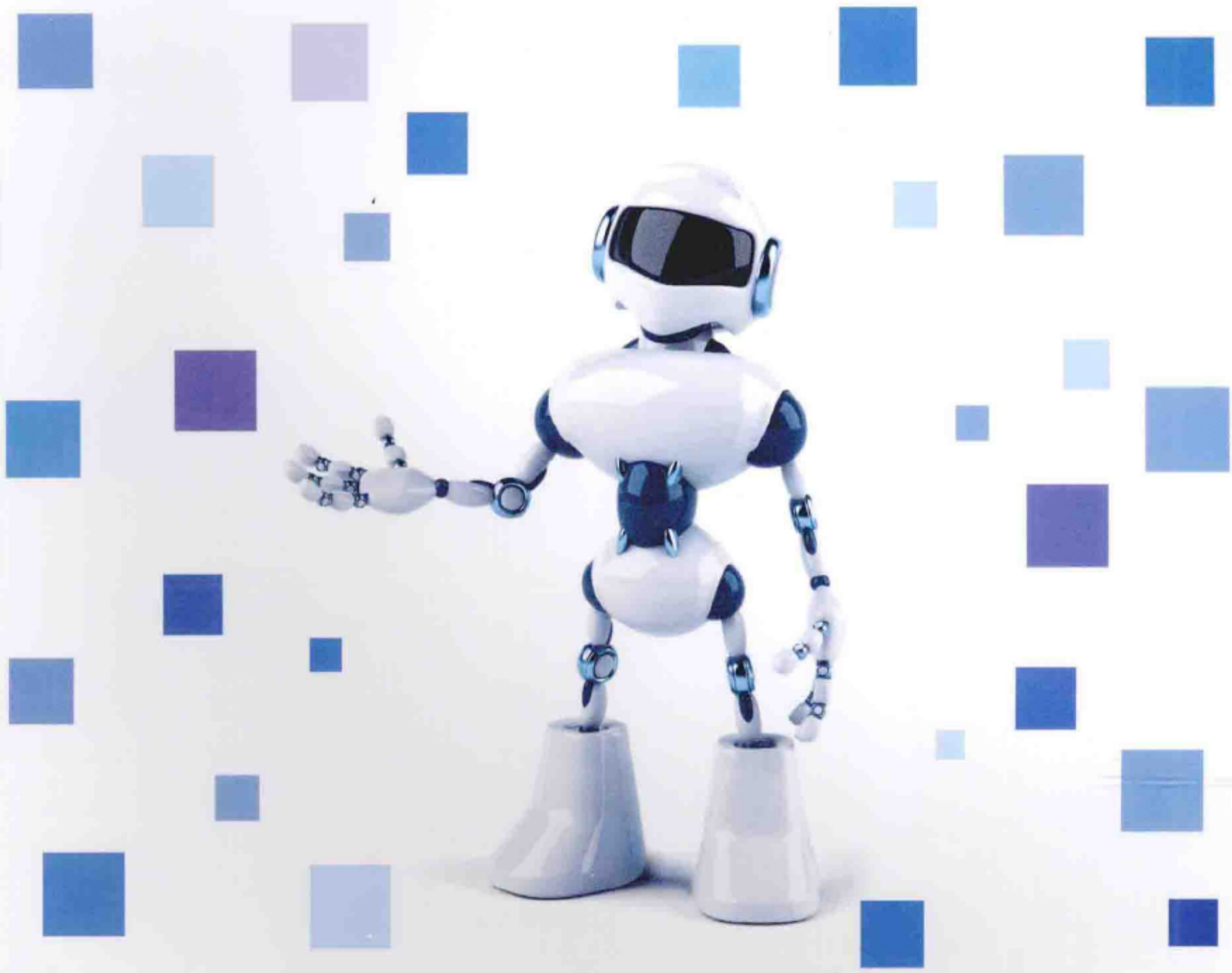
国内首本引进的ROS机器人程序设计译著，让你全面了解 ROS系统的各种工具。

提供了各种实际的示例代码供读者学习和理解ROS的软件框架。

本书可以帮助读者从对ROS一无所知到能够通过ROS系统完成小型机器人系统的开发和编程工作。



ROS机器人程序设计

Learning ROS for Robotics Programming

[西班牙] Aaron Martinez Enrique Fernández 著

刘品杰 译

机械工业出版社
China Machine Press

读者对象

本书主要针对希望学习机器人制作和设计，尤其是想要设计自己的机器人、真正热爱机器人的爱好者。本书将会为你提供让你的机器人可维护可升级，并且能够相互分享的途径。为了能够更好地学习本书的内容，你最好具备一定的C++程序设计基础以及GNU/Linux系统和计算机科学知识。而你并不需要对ROS有任何了解，因为本书正是要介绍这些基本知识和技能。还有，你最好能够了解一定的版本控制系统的知识，如SVN或GIT。在本书中会经常使用它们。

作者简介

Aaron Martinez 是一位计算机工程师、企业家和数字化制造专家。他硕士毕业于拉斯帕尔马斯大学的IUCTC（科学与网络技术研究所）。之后，参与过拉斯帕尔马斯大学AVORA项目。在这个项目中，他负责设计AUV（自主式水下机器人），并在意大利参加了欧洲学生自主式水下挑战（SAUC-E）。

Enrique Fernández 是一名计算机工程师和机器人专家。他硕士毕业于拉斯帕尔马斯大学智能系统与计算工程学院。在2012年参加了欧洲学生自主式水下挑战（SAUC-E），并作为合作者参加了2013年的比赛。 2012年，他因开发水下云台视觉系统而获奖。现在，他是Pal-Robotics实验室的SLAM工程师。Enrique在博士学习期间发表了数篇学术论文和专著。其中，有两篇论文在2011年被国际机器人与自动化会议（ICRA 2011）所收录。

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/程序设计/Java
ISBN 978-7-111-47396-1
9 787111 473961 >
定价：59.00元

ROS机器人程序设计

Learning ROS for Robotics Programming

[西班牙] Aaron Martinez Enrique Fernández 著
刘品杰 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

ROS 机器人程序设计 /（西）马丁内斯（Martinez，A.）等著；刘品杰译．—北京：机械工业出版社，2014.8
（电子与嵌入式系统设计丛书）
书名原文：Learning ROS for Robotics Programming

ISBN 978-7-111-47396-1

I. R…　II. ① 马…　② 刘…　III. 机器人－程序设计　IV. TP242

中国版本图书馆 CIP 数据核字（2014）第 161738 号

本书版权登记号：图字：01-2014-1838

Aaron Martinez, Enrique Fernández: Learning ROS for Robotics Programming（ISBN: 978-1-78216-144-8）

Copyright © 2013 Packt Publishing. First published in the English language under the title"Learning ROS for Robotics Programming".

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2014 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

ROS 是一个机器人操作系统框架，现今已有数百个研究团体和公司将其应用在机器人技术产业中。本书是一本 ROS 的入门手册，从 ROS 系统的安装到主要功能包里各项工具的使用方法如先进的计算机视觉和导航工具等都进行细致的介绍和说明，并且提供全面的相关示例代码及详细的解释，读者可以按照本书指导进行相关练习。本书适用于机械、自动化、计算机等相关专业高年级本科生及研究生学习，也适合于准备学习 ROS 的科研人员及企业研发人员学习和使用。

ROS 机器人程序设计

［西班牙］ Aaron Martinez　等著

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：朱　瑛　　责任校对：董纪丽
印　　刷：藁城市京瑞印刷有限公司　　版　　次：2014 年 9 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：15
书　　号：ISBN 978-7-111-47396-1　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有・侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

The Translator's Words | 译者序

在过去的几十年里，机器人主要是自动化或机械专业的研究领域，计算机往往作为辅助仿真的工具，机器人的程序设计也往往仅限于使用诸如 matlab 机器人工具箱之类的仿真工具。今天很多大型 IT 企业都投入大量资源开发了机器人相关的软件开发工具，例如 Visual Studio 的 Robotics Developer Studio，这些软件工具的目的是颠覆传统的机器人开发和设计模式。机器人操作系统就是基于这个目的产生的。它们能够帮助开发人员在原有机器人的基础上不断地进行二次开发，深入拓展机器人技术。而其中最具代表性的正是开源机器人操作系统 ROS（Robot Operating System）。

与其说 ROS 是一个操作系统，不如说它是一种分布式模块化的开源软件框架。它借用标准的 TCP(UDP)/IP 协议实现了系统内部各个节点之间的通信，网络化的接口允许将第三方组件泛化成为其操作系统的一部分。如果你真正使用过 ROS，就会为其无所不包的开放性而感到惊讶。正因如此，ROS 社区集中了全世界顶尖的机器人研究人员。今天，ROS 能够无缝集成现在已知的大部分机器人操作系统；能满足最高的实时性与可靠性要求；通过使用 EtherCAT 等高速总线能以每秒 Gb 级的速度传输视频数据；能够通过集成其他开源组件及增强现实技术提供多种仿真环境；能够通过多种接口与各种硬件设备或传感器进行通信；能够替你计算繁琐的正逆运动学，通过标定和配置轻松完成视觉伺服等高难度任务；你可以方便地移植其他机器人上已经测试过的各类算法和代码，或引入其他开源库。

虽然 ROS 大幅度降低了开发和应用机器人的难度，但是考虑到机器人领域无所不包，想要学习和使用 ROS 还是有几点需要注意的。首先，你应该学习过 C/C++ 程序设计语言，能够使用 Linux 编译和运行程序，这是阅读本书的基本要求。当然，如果你是一个编程新手，也可以跟着本书示例在实践中学习 Linux 和编程知识。其次，在学习和使用 ROS 的时候，往往还需要懂得程序设计之外的很多机器人知识，尤其是数学知识。但由于机器人领域过于宽泛，因此在学习过程中最好能够专注于某一方面的相关知识，不要贪多求全。再次，ROS 是一个开源软件，因此我们也要保持一种开放的心态。虽然本书能帮助你入门相对轻松一些，但这并不会彻底优化你的学习曲线。如果想要学好 ROS，一定要去 www.ros.org 上学习最

新的资料，尤其是ROS wiki。最顶尖的ROS国际会议是ROSCON，你可以到Google或者YouTube上搜索相关内容，里面有ROS最新的发展与介绍。最后，本书英文版虽然是2013年9月出版，但ROS和Linux一样半年出一个新的版本，因此书中并没有介绍最新的ROS Hydro和ROS Groovy。其实这些并不妨碍你对ROS的学习，没有谁为了学习Linux编程而非要用最新出版的图书，因为软件的基本框架和组织方式不会发生改变，而且支持ROS的机器人硬件并不一定支持最新的ROS。ROS Hydro和ROS Groovy主要对ROS原有的编译系统和部分功能包有所升级和替换，如果有兴趣的话，建议在学习完本书之后尚有余力的情况下，可以通过wiki自学ROS Hydro或ROS Groovy。

本书的两位作者并不是以英语为母语的，因此英文原著中存在着较多的语法错误和校对错误。虽然在翻译过程中参考了原著的勘误和所附代码，考虑到译者的水平亦有限，因此仍然不免存在错误，还请大家海涵。在本书之前，ROS的中文译著甚少，因此有些专业名词的翻译亦有待商榷。为了方便读者学习，译者共享了一个VMware 9.0下的虚拟机，里面包含完整安装的Vbuntu 12.04和ROS Fuerte（下载地址是http://pan.baidu.com/s/1nt6uH9R）。这样，使用VMware虚拟机的读者可以下载并在VMware中运行，然后直接学习第2章到第8章的内容。关于本书中的各种问题，欢迎大家与我联系，我的邮箱是liupinjie@gmail.com。

虽然ROS是机器人技术，但是其分布式的节点布置同样满足物联网及很多工业应用的需求，同时也有很多外国企业利用它进行机床、船舶、汽车或自动化系统的制造。记住，ROS代表了未来的智能化时代，赋予了我们新的创造与改变世界的能力。

刘品杰

2014年4月于北京

Preface | 前 言

本书概括性地介绍了ROS系统的各种工具。ROS是一个先进的机器人操作系统框架，现今已有数百个研究团体和公司将其应用在机器人技术产业中。对于机器人技术的非专业人士来说，它也相对容易上手。在本书中，你将了解如何安装ROS，如何开始使用ROS的基本工具，以及如何最终应用先进的计算机视觉和导航工具。

在阅读本书的过程中无需使用任何特殊的设备。书中每一章都附带了一系列的源代码示例和教程，你可以在自己的计算机上运行。这是你唯一需要做的事情。当然，我们还会告诉你如何使用硬件，这样，你可以将你的算法应用到现实环境中。我们在选择设备时特意去选择一些业余用户负担得起的设备，并同时涵盖了在机器人研究中最典型的传感器或执行机构。

最后，由于ROS系统的存在使得整个机器人具备在虚拟环境中工作的能力。你将学习如何创建自己的机器人并结合功能强大的导航功能包集。此外如果使用Gazebo仿真环境，你将能够在虚拟环境中运行一切。我们会在本书的结尾提供一个能够在ROS虚拟环境中进行模拟试验的机器人列表。你将发现你已经可以与机器人一起工作，并理解其背后的原理。

主要内容

第1章简单介绍安装ROS系统的方法，同时还介绍ROS不同版本的安装包的安装，本书使用的是ROS Fuerte。这一章还会说明如何从Debian软件包安装或从源代码进行编译安装，以及在虚拟机中安装。

第2章涉及ROS框架及相关的概念和工具。该章介绍节点、主题和服务，以及如何使用它们，还将通过一系列示例来说明如何调试一个节点或通过可视化方法直观地查看通过主题所发布的消息。

第3章进一步展示ROS强大的调试工具，以及通过对节点主题的图形化可以将节点间的通信数据可视化。ROS提供一个日志记录API，允许轻松地诊断节点的问题。事实上在使用过程中，我们会看到一些功能强大的图形化工具如rxconsole和rxgraph，以及可视化接口

如 rxplot 和 rviz。最后介绍如何使用 rosbag 和 rxbag 记录和回放消息。

第 4 章介绍 ROS 系统与真实世界如何连接。这一章介绍在 ROS 下使用的一些常见传感器和执行机构，如激光雷达、伺服电动机、摄像头、RGB-D 传感器等设备。此外，还会解释如何使用嵌入式系统与微控制器，例如非常流行的 Arduino。

第 5 章介绍我们在 ROS 系统中实现机器人的第一步是在 ROS 中创建一个机器人模型，包括在 Gazebo 仿真环境中如何从头开始对一个机器人进行建模和仿真，并使其在仿真环境中运行。这是后续学习如何使用 ROS 的导航功能包集和其他工具的前提条件。

第 6 章介绍 ROS 对摄像头和计算机视觉任务的支持。首先使用 FireWire 和 USB 摄像头驱动程序将摄像头连接到计算机并采集图像。然后，你就可以使用 ROS 的标定工具标定你的摄像头。我们会详细介绍和说明什么是图像管道，学习如何使用集成了 OpenCV 的多个机器视觉 API。最后，安装并使用一个视觉测距软件。

第 7 章是本书关于 ROS 导航功能包集的两章中的第 1 章。介绍如何对你的机器人进行使用导航功能包集所需的初始化配置。然后用几个例子对导航功能包集进行说明。

第 8 章延续第 7 章的内容，介绍如何使用导航功能包集使我们的机器人有效地自主导航。本章介绍使用 ROS 的 Gazebo 仿真环境和 rviz 创建一个虚拟的环境，并在其中构建地图、定位我们的机器人并做路径规划与避障。

第 9 章结合前面几章所学的内容，介绍能够支持 ROS 并使用 Gazebo 仿真环境的一些机器人。在该章中，你将看到如何在仿真环境中运行这些机器人，并执行几项本书中介绍过的任务，尤其是与导航功能包集相关的。

预备知识

我们写作本书的目的是希望尽可能让每位读者都可以完成本书的学习并运行示例代码。基本上，你只需要在计算机上安装一个 Linux 发行版。虽然每个 Linux 发行版本应该都能使用，但还是建议你使用 Ubuntu 的最新版。这样你可以根据第 1 章的内容安装 ROS Fuerte。对于 ROS 的这一版本，你将需要 Ubuntu 12.10 之前的版本，因为之后的版本已经不再支持 Fuerte 了。

对于硬件要求，一般来说任何台式计算机或笔记本电脑都满足。但是，最好使用独立显卡来运行 Gazebo 仿真环境。此外，如果能够有足够的外围接口将会更好，因为这样你可以连接几个传感器和执行机构，包括摄像头和 Arduino。

你还需要 Git（git-core Debian 软件包），以便从本书提供的软件源下载源代码。同样，你需要具备 Bash 命令行、GNU/Linux 工具的基本知识和一些 C/C++ 编程技巧。

目标读者

本书的目标读者包括所有机器人开发人员，可以是初学者也可以是专业人员。它涵盖了整个机器人系统的各个方面，展示了 ROS 系统如何帮助完成机器人真正自主化的任务。对于听说过 ROS 却从未使用过的机器人领域的学生或科研人员来说，本书将是非常有益的。ROS 初学者能从本书中学习很多 ROS 软件框架的先进理念和工具。不仅如此，经常使用 ROS 的用户也可能从某些章节中学习到一些新东西。当然，只有前 3 章是纯粹为初学者准备的，所以那些已经使用过 ROS 的人可以跳过这部分直接阅读后面的章节。

排版说明

在本书中，你会发现一些文本的样式与其他正文内容不同。下面是一些样式的示例及含义。

当我们希望能够着重显示示例代码中的某些内容的时候，相关的代码行会如下表示：

```
<launch>
<node pkg="chapter3_tutorials" type="example1"
            name="example1" output="screen"
            launch-prefix="xterm -e gdb --args"/>
</launch>
```

Bash 命令行下输入的命令或显示的结果如下表示：

```
$ rosrun book_tutorials tutorialX _param:=9.0
```

警告或注意事项会这样显示。

提示与小窍门会这样显示。

读者反馈

我们非常欢迎读者能够提出意见与建议。让我们知道你关于本书的一些想法，例如：你喜欢什么，不喜欢什么。读者的反馈对我们是非常重要的，这让我们知道我们最需要做的是什么。

向我们反馈信息只需发送电子邮件至 feedback@packtpub.com，并在邮件的主题中说明书籍名称。

如果你在某项上有所专长，或者你有兴趣写一本书或者为书籍的出版做出贡献，你可以浏览 www.packtpub.com/authors 阅读我们的作者指南。

读者支持

你现在自豪地拥有了 Packt 出版的书籍，我们会帮助你使你购买的产品物超所值。

源代码下载

可以访问 http://www.packtpub.com/support，完成网站注册并通过 E-mail 接收所有代码文件。

彩色图片下载

我们同时提供包含了本书所有彩色的屏幕截图、对话框的 PDF 文件，这些彩色图片能够更好地帮助你理解输出的变化。可以从这里下载这个文件：http://www.packtpub.com/sites/default/files/ downloads/1448OS_Graphics.pdf。

勘误

虽然我们已尽力确保本书内容的准确性，但错误在所难免。如果你在我们的书中发现错误，即使是一个错字或一段错误的代码，我们将非常感谢你把错误告知我们。这样可以避免其他读者发生困扰并帮助我们提高本书后续版本的质量。如果你发现任何错误，请访问 http://www.packtpub.com/ submit-errata 报告这些问题。访问网页，选择你的书，点击勘误提交表格的链接，并输入你的勘误细节。一旦确认了你的勘误，我们会告知你的提交被接受并在我们的网站上更新勘误表。你可以在 http://www. packtpub.com/support 根据你选择的标题查看现有的勘误表。

问题

如果关于本书你有任何疑问，请联系我们 questions@packtpub.com，我们会竭尽所能为你解答。

Contents | 目 录

译者序

前 言

第 1 章 ROS 系统入门 …… 1

1.1 使用软件源安装 ROS Electric …… 3

1.1.1 添加软件源到 sources.list 文件中 …… 4

1.1.2 设置密码 …… 4

1.1.3 安装 …… 4

1.1.4 环境配置 …… 5

1.2 使用软件源安装 ROS Fuerte …… 6

1.2.1 配置 Ubuntu 软件源 …… 6

1.2.2 配置 source.list 文件 …… 6

1.2.3 设置密码 …… 7

1.2.4 安装 …… 7

1.2.5 环境配置 …… 8

1.2.6 独立工具 …… 9

1.3 如何安装 VirtualBox 和 Ubuntu …… 9

1.3.1 下载 VirtualBox …… 9

1.3.2 创建虚拟机 …… 10

1.4 本章小结 …… 12

第 2 章 ROS 系统架构及示例 …… 13

2.1 理解 ROS 文件系统级 …… 13

2.1.1 功能包 …… 14

2.1.2 功能包集 …… 16

2.1.3 消息类型 …… 16

2.1.4 服务类型 …… 17

2.2 理解 ROS 计算图级 …… 18

2.2.1 节点 …… 19

2.2.2 主题 …… 20

2.2.3 服务 …… 21

2.2.4 消息 …… 22

2.2.5 消息记录包 …… 22

2.2.6 节点管理器 …… 22

2.2.7 参数服务器 …… 22

2.3 理解 ROS 开源社区级 …… 23

2.4 ROS 系统试用练习 …… 23

2.4.1 ROS 文件系统导览 …… 24

2.4.2 创建工作空间 …… 24

2.4.3 创建 ROS 功能包 …… 25

2.4.4 编译 ROS 功能包 …… 26

2.4.5 使用 ROS 节点 …… 26

2.4.6 使用主题与节点交互 …… 28

2.4.7 学习如何使用服务 …… 31

2.4.8 使用参数服务器 …… 33

2.4.9 创建节点 …… 34

2.4.10 编译节点 …… 36

2.4.11 创建 msg 和 srv 文件 …… 37
2.4.12 使用新建的 srv 和 msg 文件 …… 38
2.5 本章小结 …… 42

第 3 章 调试和可视化 …… 43

3.1 调试 ROS 节点 …… 44
3.1.1 使用 GDB 调试器调试 ROS 节点 …… 45
3.1.2 ROS 节点启动时调用 GDB 调试器 …… 46
3.1.3 设置 ROS 节点 core 文件转存 …… 47
3.2 调试信息 …… 47
3.2.1 输出调试信息 …… 47
3.2.2 设置调试信息级别 …… 48
3.2.3 为特定节点配置调试信息级别 …… 48
3.2.4 信息命名 …… 50
3.2.5 条件显示信息与过滤信息 …… 50
3.2.6 信息的更多功能——单次显示、可调、组合 …… 51
3.2.7 使用 rosconsole 和 rxconsole 在运行时修改调试级别 …… 52
3.3 监视系统状态 …… 56
3.3.1 节点、主题与服务列表 …… 56
3.3.2 使用 rxgraph 在线监视节点状态图 …… 56
3.4 当奇怪的事情发生——使用 roswtf …… 58
3.5 画标量数据图 …… 58
3.5.1 用 rxplot 画出时间趋势曲线 …… 59
3.5.2 另一个画图工具 rxtools …… 60
3.6 图像可视化 …… 61
3.6.1 显示单一图片 …… 61
3.6.2 FireWire 接口摄像头 …… 62
3.6.3 使用双目立体视觉 …… 63
3.7 3D 可视化 …… 64
3.7.1 使用 rviz 在 3D 世界中实现数据可视化 …… 64
3.7.2 主题与坐标系的关系 …… 66
3.7.3 可视化坐标变换 …… 67
3.8 保存与回放数据 …… 68
3.8.1 什么是消息记录包文件 …… 69
3.8.2 使用 rosbag 在包文件中记录数据 …… 69
3.8.3 回放消息记录文件 …… 70
3.8.4 使用 rxbag 检查消息记录包的主题和消息 …… 71
3.9 rqt 插件与 rx 应用 …… 72
3.10 本章小结 …… 73

第 4 章 在 ROS 下使用传感器和执行机构 …… 74

4.1 使用游戏杆或游戏手柄 …… 74
4.1.1 joy_node 如何发送游戏杆动作消息 …… 75
4.1.2 使用游戏杆数据在 turtlesim 中移动海龟 …… 76
4.2 使用激光雷达——Hokuyo URG-04lx …… 79
4.2.1 了解激光雷达如何在 ROS 中发送数据 …… 80
4.2.2 访问和修改激光雷达数据 …… 82

4.3 使用 Kinect 传感器查看 3D 环境 …… 84
4.3.1 如何发送和查看 Kinect 数据 …… 85
4.3.2 创建和使用 Kinect 示例 …… 86
4.4 使用伺服电动机——Dynamixel …… 88
4.4.1 Dynamixel 如何发送和接收运动命令 …… 89
4.4.2 创建和使用伺服电动机示例 …… 90
4.5 使用 Arduino 添加更多的传感器和执行机构 …… 91
4.6 使用惯性测量模组——Xsens MTi …… 94
4.6.1 Xsens 如何在 ROS 中发送数据 …… 95
4.6.2 创建和使用 Xsens 示例 …… 96
4.7 使用低成本惯性测量模组 IMU-10 自由度 …… 98
4.7.1 下载加速度传感器库 …… 99
4.7.2 Arduino Nano 和 10 自由度传感器编程 …… 99
4.7.3 创建 ROS 节点并使用 10 自由度传感器数据 …… 101
4.8 本章小结 …… 103

第 5 章 3D 建模与仿真 …… 104

5.1 自定义机器人在 ROS 中的 3D 模型 …… 104
5.2 创建第一个 URDF 文件 …… 104
5.2.1 解释文件格式 …… 106
5.2.2 在 rviz 里查看 3D 模型 …… 107
5.2.3 加载图形到机器人模型 …… 109
5.2.4 使机器人模型运动 …… 109
5.2.5 物理和碰撞属性 …… 110
5.3 xacro——一个写机器人模型的更好方法 …… 111
5.3.1 使用常量 …… 111
5.3.2 使用数学方法 …… 112
5.3.3 使用宏 …… 112
5.3.4 使用代码移动机器人 …… 112
5.3.5 使用 SketchUp 进行 3D 建模 …… 116
5.4 在 ROS 中仿真 …… 117
5.4.1 在 Gazebo 中使用 URDF3D 模型 …… 117
5.4.2 在 Gazebo 中添加传感器 …… 120
5.4.3 在 Gazebo 中加载和使用地图 …… 121
5.4.4 在 Gazebo 中移动机器人 …… 123
5.5 本章小结 …… 125

第 6 章 机器视觉 …… 126

6.1 连接和运行摄像头 …… 128
6.1.1 FireWire IEEE1394 摄像头 …… 128
6.1.2 USB 摄像头 …… 132
6.2 使用 OpenCV 制作 USB 摄像头驱动程序 …… 133
6.2.1 创建 USB 摄像头驱动功能包 …… 134
6.2.2 使用 ImageTransport API 发布摄像头帧 …… 135
6.2.3 使用 cv_bridge 进行 OpenCV 和 ROS 图像处理 …… 138

6.2.4 使用 ImageTransport 发布图像 …… 139
6.2.5 在 ROS 中使用 OpenCV …… 139
6.2.6 显示摄像头输入的图像 …… 140
6.3 如何标定摄像头 …… 140
6.4 ROS 图像管道 …… 147
6.5 对于计算机视觉任务有用的 ROS 功能包 …… 152
6.6 使用 viso2 执行视觉测距 …… 153
6.6.1 摄像头位姿标定 …… 154
6.6.2 运行 viso2 在线演示 …… 156
6.6.3 使用低成本双目摄像头运行 viso2 …… 158
6.7 本章小结 …… 159

第 7 章 导航功能包集入门 …… 160
7.1 ROS 导航功能包集 …… 160
7.2 创建转换 …… 161
7.2.1 创建广播机构 …… 162
7.2.2 创建侦听器 …… 162
7.2.3 查看坐标变换树 …… 164
7.3 发布传感器信息 …… 165
7.4 发布里程数据 …… 168
7.4.1 Gazebo 如何获取里程数据 …… 169
7.4.2 创建自定义里程数据 …… 171
7.5 创建基础控制器 …… 175
7.5.1 使用 Gazebo 创建里程数据 …… 176
7.5.2 创建基础控制器 …… 178
7.6 使用 ROS 创建地图 …… 180
7.6.1 使用 map_server 保存地图 …… 181
7.6.2 使用 map_server 加载地图 …… 182
7.7 本章小结 …… 183

第 8 章 导航功能包集进阶 …… 184
8.1 创建功能包 …… 184
8.2 创建机器人配置 …… 184
8.3 配置全局和局部代价地图 …… 187
8.3.1 基本参数的配置 …… 187
8.3.2 全局代价地图的配置 …… 188
8.3.3 局部代价地图的配置 …… 189
8.4 基本局部规划器配置 …… 189
8.5 为导航功能包集创建启动文件 … 190
8.6 为导航功能包集设置 rviz …… 191
8.6.1 2D 位姿估计 …… 191
8.6.2 2D 导航目标 …… 192
8.6.3 静态地图 …… 193
8.6.4 点云 …… 193
8.6.5 机器人立足点 …… 193
8.6.6 障碍 …… 194
8.6.7 膨胀障碍 …… 194
8.6.8 全局规划 …… 195
8.6.9 局部规划 …… 195
8.6.10 规划器规划 …… 196
8.6.11 当前目标 …… 196
8.7 自适应蒙特卡罗定位 …… 197
8.8 避免障碍 …… 199
8.9 发送目标 …… 200
8.10 本章小结 …… 202

第 9 章　在实践中学习 …… 203

9.1　REEM——类人形 PAL 机器人 …… 204

9.1.1　从官方软件源安装 REEM …… 205

9.1.2　使用 Gazebo 仿真环境运行 REEM …… 208

9.2　PR2——柳树车库机器人 …… 210

9.2.1　安装 PR2 仿真环境 …… 210

9.2.2　在仿真环境中运行 PR2 …… 211

9.2.3　生成地图与定位 …… 214

9.2.4　在仿真环境中运行 PR2 演示程序 …… 216

9.3　Robonaut 2——NASA 的敏捷型人形机器人 …… 217

9.3.1　从软件源安装 Robonaut 2 …… 217

9.3.2　在国际空间站的固定支座上运行 Robonaut2 …… 218

9.4　Husky——Clearpath 的轮式机器人 …… 222

9.4.1　安装 Husky 仿真环境 …… 222

9.4.2　运行 Husky 仿真环境 …… 222

9.5　TurtleBot——低成本移动机器人 …… 224

9.5.1　安装 TurtleBot 仿真环境 …… 224

9.5.2　运行 TurtleBot 仿真环境 …… 224

9.6　本章小结 …… 225

第 1 章
ROS 系统入门

欢迎来到本书的第 1 章。本章会介绍如何安装 ROS 系统。与其说 ROS 是一个操作系统，不如说它是一种新的标准化机器人软件框架。通过 ROS，你可以使用大量的示例代码和开源程序轻松地完成机器人编程和控制。同时，你还能够理解如何使用各种传感器与执行机构，并为你的机器人增加如自动导航和视觉定位等新的功能。在这里，我们要感谢相关的基金会及社区对开源软件的贡献，并不断地开发出最新的算法和功能使 ROS 系统不断进步。

本书包含以下知识：

- 在特定版本的 Ubuntu 系统下安装 ROS 框架
- 学习 ROS 的基本操作
- 调试数据和数据可视化
- 在 ROS 框架下对机器人编程
- 创造 3D 模型并进行仿真
- 使用导航功能包集使机器人进行自动导航

在本章中，首先要在 Ubuntu 系统中安装完整版本的 ROS。Ubuntu 不仅能够全面支持 ROS，而且它也是 ROS 官方推荐的操作系统。当然，你也可以在其他的操作系统中安装 ROS。但是在其他操作系统中，ROS 大多是实验性质的移植版本，因此容易出现各种安装和程序执行错误。因此在学习和使用本书的过程中，推荐使用 Ubuntu。

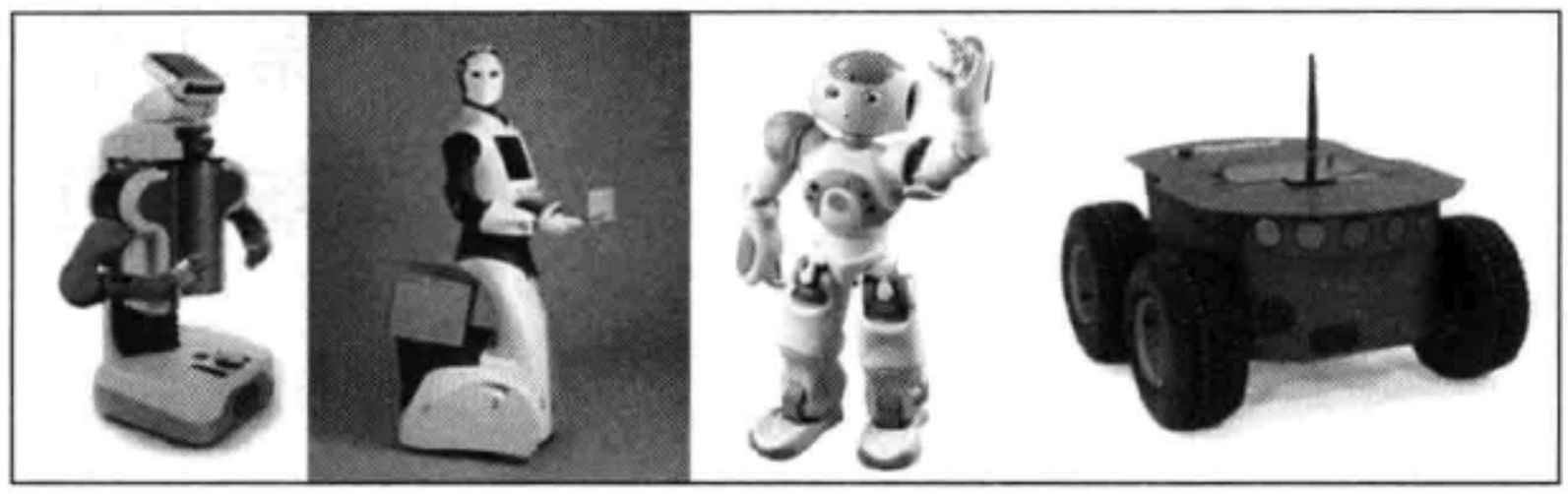
在开始安装之前，首先了解一下 ROS 的历史。

Robot Operating System（ROS）是一种得到广泛使用的机器人操作与控制系统软件框架。该框架使用了当前最流行的面向服务（SOA）的软件技术，通过网络协议将节点间数据通信解耦。这样就能够轻松地集成不同语言不同功能的代码。ROS 的基本原理是无需改动就能够在不同的机器人上复用代码。基于这些，我们就可以在不同的机器人上分享和复用已经实现的功能，而不需要做太多的工作，避免了重复劳动。

2007 年，斯坦福大学人工智能实验室 (SAIL) 在斯坦福 AI 机器人项目（Stanford AI Robot project）的支持下开发了 ROS。2008 年之后，则主要在 Willow Garage 公司支持下与其他 20 多家研究机构进行联合研发 ROS。

现在已经有很多家研究机构通过增加 ROS 支持的硬件或开放软件源代码的方式加入 ROS 系统的开发中。同样，也有很多家公司将其产品逐步进行软件迁移并在 ROS 系统中应

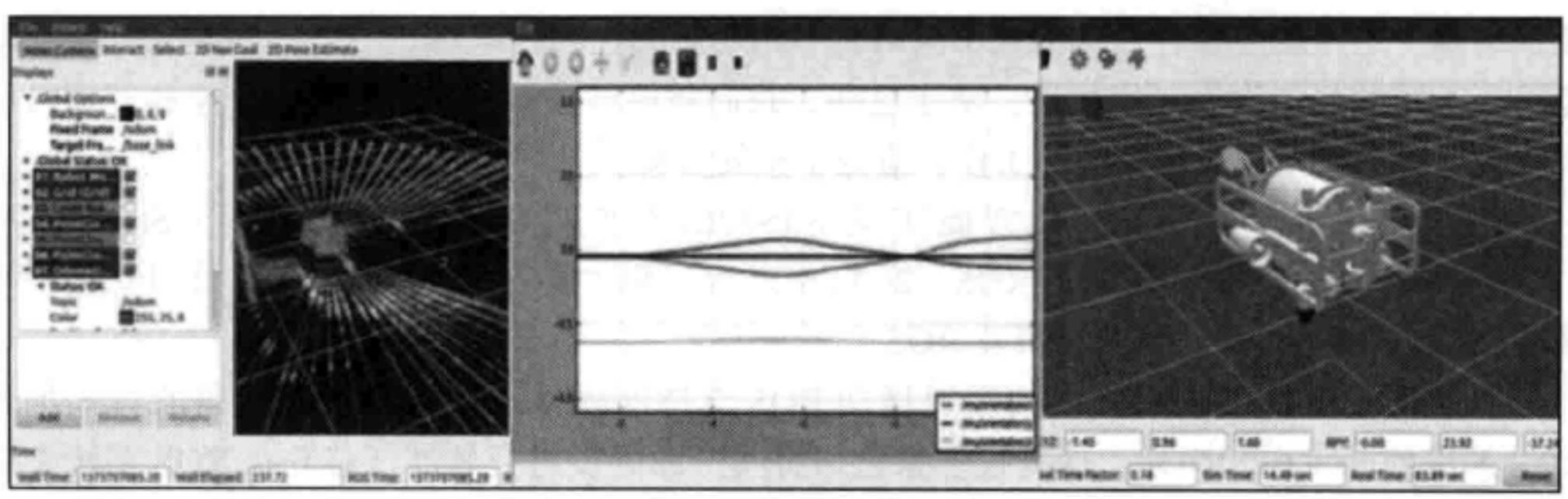
用。部分 ROS 系统支持的平台如下图所示。这些平台往往会开放大量的代码、示例和仿真环境，以便开发人员轻松地开展工作。



ROS 系统已经支持这些机器人中的传感器和执行机构，同时每天 ROS 软件框架所支持的设备也在大量增加。

ROS 提供了一个标准的操作系统环境，包括硬件抽象、底层设备控制、通用功能的实现、进程间消息转发和功能包管理等。它将多路复用传感器、控制器、运动状态、规划目标、执行机构及其他设备全部抽象成节点（node），并能对各个节点进程间消息的发生和接受的处理过程通过节点状态图展示。它的各种库都是面向类 Unix 系统的（ROS 主要支持 Ubuntu Linux，而其他系统如 Fedora 和 Mac OS X 则是实验性质的）。

`*-ros-pkg` 包主要作为开发高级库的基础软件源。其很多功能是和 ROS 系统绑定的，如导航库和 rviz 可视化界面都基于这个源。其中的一些库包含很多强大的工具，可以帮助我们方便使用 ROS 并了解机器人的实时状态。其中，可视化工具、仿真环境和调试工具是最重要的几个。



ROS 是一个使用 BSD（Berkeley Software Distribution）开源协议的开源软件。无论是商业应用，还是科学研究，它都是免费的。*-ros-pkg 包受到了多个开源协议的限制。

ROS 致力于机器人的代码复用，这样每次研发新的机器人时，开发人员和科学家们再也不必全部推倒重来。通过 ROS，你可以集中精力做更多的事情，应用不同代码库中的代码，再完善，并再次分享。

ROS 已经发布了多个版本，最新的版本是 Groovy。在本书中，我们使用的版本是 Fuerte，因为这个版本更加稳定。因此需要说明的是，本书中的示例和教程不一定都能够在 Groovy 下使用。

下面我们会介绍如何安装 Electric 版本和 Fuerte 版本的 ROS。不同的 ROS 版本对应不同的 Ubuntu 版本，所以某些机器人可能仍然使用老版本的 Ubuntu。即使在本书中我们使用 Fuerte，但是在实际工作中，你仍然可能需要安装 Electric 以便运行一些老版本的代码。

如前文所述，本书中所使用的操作系统是 Ubuntu，如果你习惯使用其他操作系统又想完成本书的学习，最好的选择就是安装一个带有 Ubuntu 的虚拟机。因此，我会在最后介绍虚拟机的安装方法。

当然，如果你想在其他系统中安装 ROS，你可以根据链接 http://wiki.ros.org/fuerte/Installation 中的指导来完成。

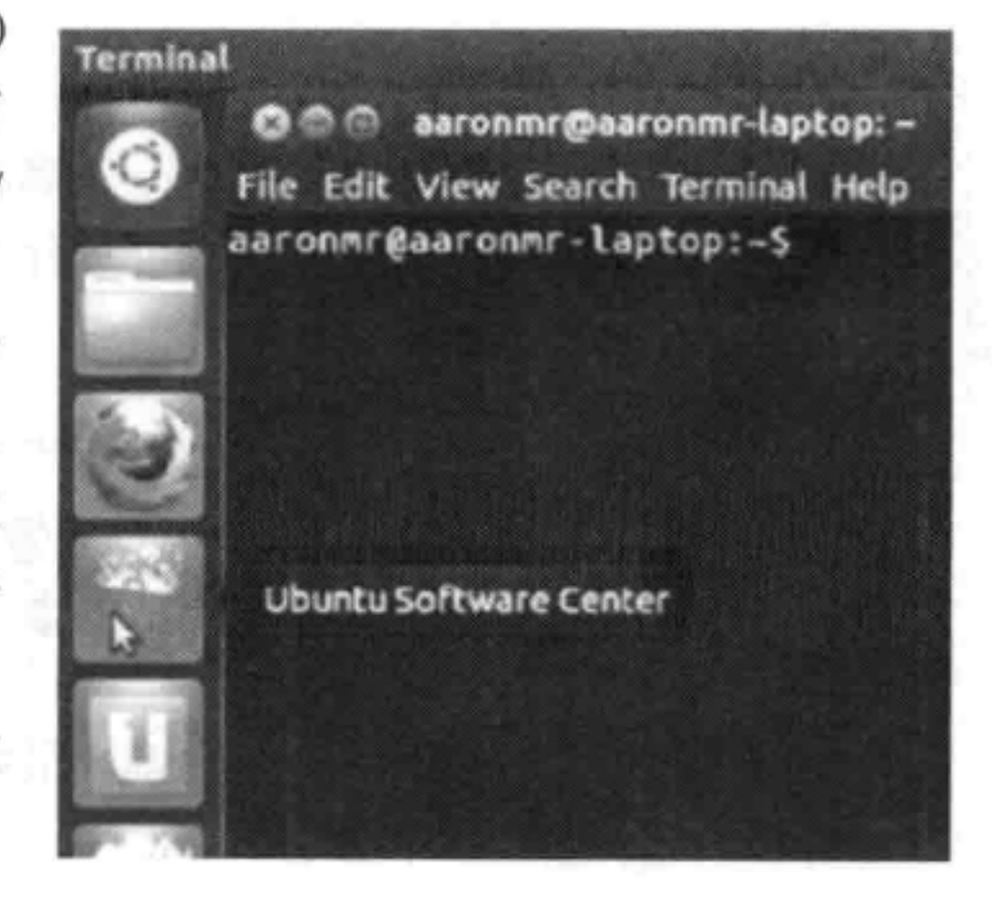
1.1 使用软件源安装 ROS Electric

安装 ROS 有多种方法，你可以按照本书的方法直接使用软件源，也可以使用源文件并编译后安装。使用软件源的方法相对安全，不容易出现错误。

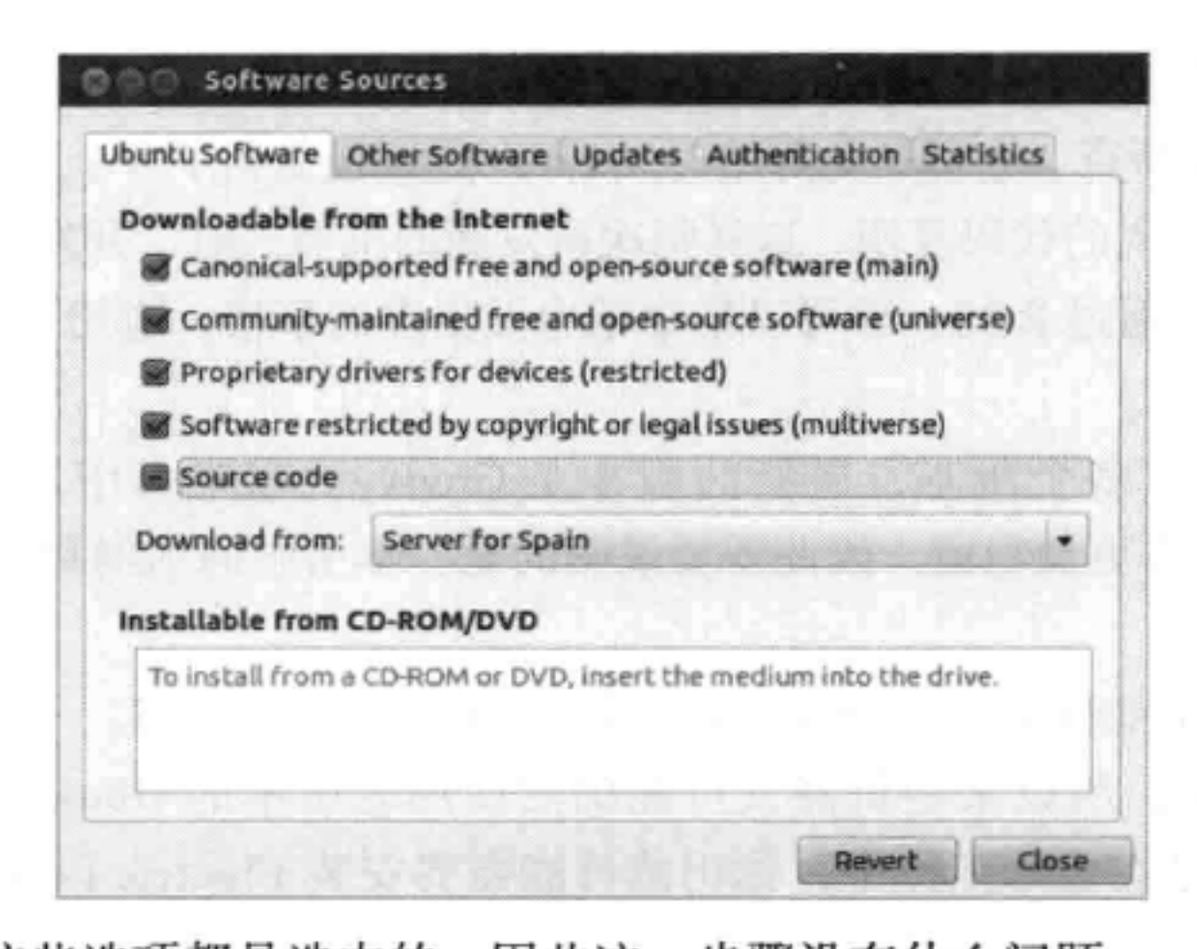
按照本节介绍的步骤，你可以在自己的电脑上逐步安装 ROS Electric，安装过程详见官网页面 http://wiki.ros.org/electric/Installation。

我们假设你知道 Ubuntu 软件源（repository）的含义，并且知道如何管理它。如果你有任何疑问请查询 https://help.ubuntu.com/community/Repositories/Ubuntu 了解相关知识。

在开始安装之前，需要首先配置软件源，为此需要先把软件源属性设为 restricted、universal、multiverse，为了检查你的 Ubuntu 版本是否支持这些软件源，请双击打开你桌面左面的 Ubuntu 软件中心（Ubuntu Software Center），如右图所示。



打开 Edit | Software Sources 标签页，你能看到以下界面，你要保证各个选项与下图中一致。



通常情况下，这些选项都是选中的，因此这一步骤没有什么问题。

1.1.1 添加软件源到 sources.list 文件中

在这一步中，你应该先选择 Ubuntu 的版本，在多种版本的操作系统中都可以安装 ROS Electric。你可以使用任何一个版本，但是推荐最新版本，以避免发生问题。

- 基于 Ubuntu 的发行版，如 Ubuntu Lucid Lynx（10.04），安装软件源的具体方法如下：

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

- 安装任意 Ubuntu 发行版的一种通用方法是使用所有基于 Debian 的 Linux 发行版都支持的 `lsb_release` 命令：

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

一旦添加了正确的软件源，操作系统就知道去哪里下载程序，并根据命令自动安装软件。

1.1.2 设置密码

这一步是为了确认原始的代码是正确的，并且没有人在未经所有者授权的情况下修改任何程序代码。通常情况下，当添加完软件源，你就已经添加了软件源的密码，并将其添加到操作系统的可信任列表中。

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

现在我们能够确定代码来自授权网站。

1.1.3 安装

现在准备开始安装。在开始之前，最好先升级一下软件，避免错误的库版本或软件版本产生各种问题，输入以下命令：

```
$ sudo apt-get update
```

ROS 非常大，有时候你会安装一些永远也用不到的库和程序。通常情况下，根据用途有四种不同的安装方式。例如，你是一个高级用户，你只需要为你的机器人进行基本安装，而不需要在硬盘上留过多的空间。在本书中，我们推荐完全安装，因为这样能够保证包含本书中所有示例和教程需要的库。不用担心，如果你不知道正在安装什么，它可能是 rviz、仿真环境或导航功能。你将会在后续章节中学习这些内容。

- 最简单的安装方式（并且是推荐的安装方式，但你需要有足够大的硬盘空间）就是桌面完整安装（desktop-full）。这将安装 ROS、Rx 工具箱、rviz 可视化环境（3D）、通用机器人库、2D（如 stage）和 3D（如 Gazebo）仿真环境、导航工功能包集（移动、定位、地图绘制、机械臂控制）和其他感知库如视觉、激光雷达和 RGB-D 摄像头（深度摄像头）：

```
$ sudo apt-get install ros-electric-desktop-full
```

- 如果你没有足够的硬盘空间，或更喜欢安装特定部分功能包集，那么第一次安装可以仅安装桌面安装文件，包括 ROS、Rx 工具箱、rviz 和其他通用机器人库。之后在需要的时候，再安装其他功能包集（使用 apt 命令并查找 `ros-electric-*` 功能包集）：

```
$ sudo apt-get install ros-electric-desktop
```

- 如果你仅想尝试一下，请安装 ROS-base。ROS-base 通常是直接安装在机器人上，尤其是机器人并没有屏幕和人机界面，只能 TTY 远程登录的情况下。它只安装 ROS 的编译和通信库，而没有任何的 GUI 工具：

```
$ sudo apt-get install ros-electric-ros-base
```

- 最后，无论你选择哪一个选项进行安装，你都可以独立安装特定的 ROS 功能包集（将 STACK 替换成给定功能包集的名称）：

```
$ sudo apt-get install ros-electric-STACK
```

1.1.4 环境配置

恭喜你！能到这一步，说明你已经安装了某个版本的 ROS。为了使用 ROS，必须让操作系统知道可执行文件或二进制文件与系统命令的关联，这就需要执行下面的脚本。如果你已经安装了某一版本的 ROS，并想安装另一个 ROS 发行版，你可以每一次都调用你所需要的对应脚本。因为这个脚本能够非常方便地设置你的环境变量。在这里，我们将会使用 ROS Electric，但如果你想尝试其他的发行版，只需要把下面的命令中的 `electric` 替换成 `fuerte` 或 `groovy` 即可：

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

如果你在 shell 中输入 `roscore`，那么将会看到有程序启动。这是测试是否完成 ROS 安

装和 ROS 安装是否正确的最佳方法。

请注意，如果你再打开一个 shell 窗口，并输入 roscore 或其他 ROS 命令，就行不通了。这是因为需要你再一次执行脚本来配置全局变量和 ROS 的安装路径。

如何能够让系统记住你的这些配置呢？你只需要在命令行脚本末尾处增加 .bashrc 文件，这样当你再次启动 shell 的时候，会自动执行这些命令并对环境进行配置。命令脚本如下：

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

如果安装了多个 ROS 的发行版，~/.bashrc 文件必须来自当前正在使用版本的 setup.bash。这是因为最后一次调用将覆盖环境中的其他设置。正如我们前面所提到的，在同一个系统中安装多个发行版时需要在它们之间进行切换。

1.2 使用软件源安装 ROS Fuerte

在本节中，我们将在计算机上安装 ROS Fuerte。你可以在同一台计算机上安装不同版本，这都没有问题。只需要在 .bashrc 文件中选择你想要使用的版本。你将在本节学习如何做到这一点。

如果你想在官方页面查看对这个过程的解释，可以访问 http://wiki.ros.org/fuerte/Installation。

你可以使用两种方法安装 ROS：使用软件源和使用源代码。普通用户可以使用软件源来完成 ROS 的安装。你也可以使用源代码安装 ROS，但这个方法仅建议高级用户使用，不建议普通用户这样做。

1.2.1 配置 Ubuntu 软件源

在开始安装之前，首先配置软件源，为此先把软件源属性设为 restricted、universal、multiverse。具体安装方法请参考 1.1 节的相关内容。

通常情况下，Ubuntu 默认支持这些选项，并不需要进行特殊设置。

1.2.2 配置 source.list 文件

现在我们需要添加下载软件的 URL。由于 ROS Fuerte 并不支持 Ubuntu 10.10 Maverick 和 11.04 Natty，因此我们需要在电脑上安装 Ubuntu 10.04、11.10 或 12.04。

在本书中，我们选择 Ubuntu 12.04。在这个版本下，所有的示例程序都被检查、编译和执行过。

打开 shell 命令行，输入以下命令：

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

这些命令在任何版本的 Ubuntu 下都应该能够正确运行。

1.2.3 设置密码

为软件源增加密码是非常重要的，因为这样能够保证源代码是正确的，并且没有人在未经所有者知晓的情况下修改代码或程序。

如果你已经按照步骤安装完成 ROS Electric，那么你并不需要重新设置密码；如果你没有安装过 ROS Electric，那么你需要输入如下命令：

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

1.2.4 安装

我们现在准备安装 ROS Fuerte。在安装之前，需要先升级一下 ROS 可能使用到的程序。这样可以避免出现兼容性问题。

输入以下命令并等待：

```
$ sudo apt-get update
```

这个命令会花一定时间才能完成，具体取决于你是否升级过系统。

ROS 同样有很多个组件，在一个机器人上完整安装全部系统会略显笨重而没有一点个性。基于这个原因，你可以根据需要选择安装。

在本书中，我们还是会安装完整版本。这个版本会安装所有的示例、功能包集和程序。考虑到我们在本书的后续章节会使用大量的工具，完整安装是一个不错的选择。当然，如果你并不想现在就完整安装，可以之后再进行。

- 最简单的安装方式（并且是推荐的安装方式，但你需要有足够大的硬盘空间）就是桌面完整安装（desktop-full）。这将安装 ROS、Rx 工具箱、rviz 可视化环境（3D）、通用机器人库、2D（如 stage）和 3D（如 Gazebo）仿真环境、导航功能包集（移动、定位、地图绘制、路径规划、机械臂控制）和其他感知库如视觉、激光雷达和 RGB-D 摄像头（深度摄像头）：

  ```
  $ sudo apt-get install ros-fuerte-desktop-full
  ```

- 如果你没有足够的硬盘空间，或更喜欢安装部分功能包集，那么第一次仅安装桌面安装文件，包括 ROS、Rx 工具箱、rviz 和其他通用机器人库。你可以在需要的时候，再安装其他的功能包集（使用 apt 命令并查找 `ros-fuerte-*` 功能包集）：

  ```
  $ sudo apt-get install ros-fuerte-desktop
  ```

- 如果你仅想尝试一下，请安装 ROS-base。它通常是直接安装在机器人上，尤其是机器人并没有屏幕和人机界面，只能 TTY 远程登录的情况下。只安装 ROS 的编译和通信库，而没有任何的 GUI 工具：

  ```
  $ sudo apt-get install ros-fuerte-ros-comm
  ```

- 最后，无论选择哪一个选项进行安装，你都可以独立安装特定的 ROS 功能包集（将

STACK 替换成给定功能包集的名称）：

```
$ sudo apt-get install ros-fuerte-STACK
```

在安装过程中，请不要担心你对所安装的软件并不了解。在后面的章节，你会学到包括如何使用在内的所有相关知识。

当你逐渐积累 ROS 系统的经验之后，就能够在机器人上仅安装 ROS 内核，使用更少的资源，并且逐项添加你所需要的工具。

1.2.5 环境配置

现在 ROS 安装完毕，并准备开始使用。这时你需要向 Ubuntu 提供 ROS 的安装路径。打开一个新的 shell 并输入以下命令：

```
$ roscore
roscore: command not found
```

之所以会看到这样的消息，主要是因为 Ubuntu 并不知道去哪里找这个命令对应的执行程序。为了解决这个问题，需要在 shell 环境中输入以下命令：

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

然后，再次输入 `roscore` 命令，你会看到以下输出：

```
...
started roslaunch server http://localhost:45631/
ros_comm version 1.8.11

SUMMARY
========

PARAMETERS
* /rosdistro
* /rosversion

NODES

auto-starting new master
....
```

这意味着 Ubuntu 已经知道了到哪里去寻找这条命令所对应的执行程序，并启动了 ROS。而如果你重新打开一个 shell 命令行，并再次输入 `roscore`，就又不管用了。这是因为你必须将这个命令行脚本加入 `.bashrc` 文件中。这样在 shell 打开时 `.bashrc` 会自动运行，所以每次你打开一个新的 shell，这个配置环境变量的脚本都会运行。

在命令行中使用以下脚本命令：

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

和前面提到的一样，setup.bash 只能为一个 ROS 发行版进行初始化配置。试想一下，当你同时安装了 ROS Electric 和 Fuerte，而平时需要使用 Fuerte 作为常用的版本。如果你想在 shell 环境中更改版本，只需要输入以下命令：

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

如果你希望使用其他版本作为常用的版本，只需要修改 .bashrc 文件并将所需版本的相关脚本加入文件中。

1.2.6 独立工具

ROS 中的很多工具是需要在 ROS 安装之后进行独立安装的。这些工具能够帮助我们对编程和编译的安装依赖项进行管理、下载和安装 ROS 程序等。这些工具包括 rosinstall 和 rosdep。我们推荐将这两个工具都安装，因为在后续的章节中需要使用它们。我们需要在 shell 命令行中输入以下命令：

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

现在我们已经在系统中安装了一个完整版本的 ROS。如你所见，只有部分步骤是必须做的。

你可以在机器上安装两个甚至更多的 ROS 发行版。另外，如果你的电脑并不使用 Ubuntu 系统，你可以在虚拟机中安装 ROS。下一节我们将介绍如何安装虚拟机和使用镜像而且这可能是对于初学者来说最好的方法。

1.3 如何安装 VirtualBox 和 Ubuntu

VirtualBox 是一个通用的完整的，适用于 x86 硬件，定位于服务器、台式机和嵌入式应用的虚拟机。 VirtualBox 是免费的，支持所有主流的操作系统。几乎每一个 Linux 爱好者都使用它。

由于我们推荐使用 Ubuntu，你可能不希望更改计算机现有的操作系统。而如 VirtualBox 之类的工具就可以满足此类需求。它能帮助我们在电脑上建立虚拟的新操作系统，而无需对电脑硬件做任何改动。

在后面的章节中，我们将展示如何安装 VirtualBox 和 Ubuntu。此外，通过安装虚拟机，你可以在一个干净的操作系统中完成开发。如果你遇到任何问题，能够快速重启计算机解决；也可以备份虚拟机及所有必要的机器人安装文件。

1.3.1 下载 VirtualBox

第一步是下载 VirtualBox 的安装文件。在编写本书时，以下链接能够下载最新的可用版本：

- https：//www.virtualbox.org/wiki/Downloads
- http://download.virtualbox.org/virtualbox/4.2.0/VirtualBox- 4.2.1-80871-OSX.dmg

一旦安装完成，你就需要下载 Ubuntu 的镜像文件。在本教程中，我们使用了一个已经安装了 ROS Ruerte 的 Ubuntu 镜像文件。你可以通过以下链接下载：

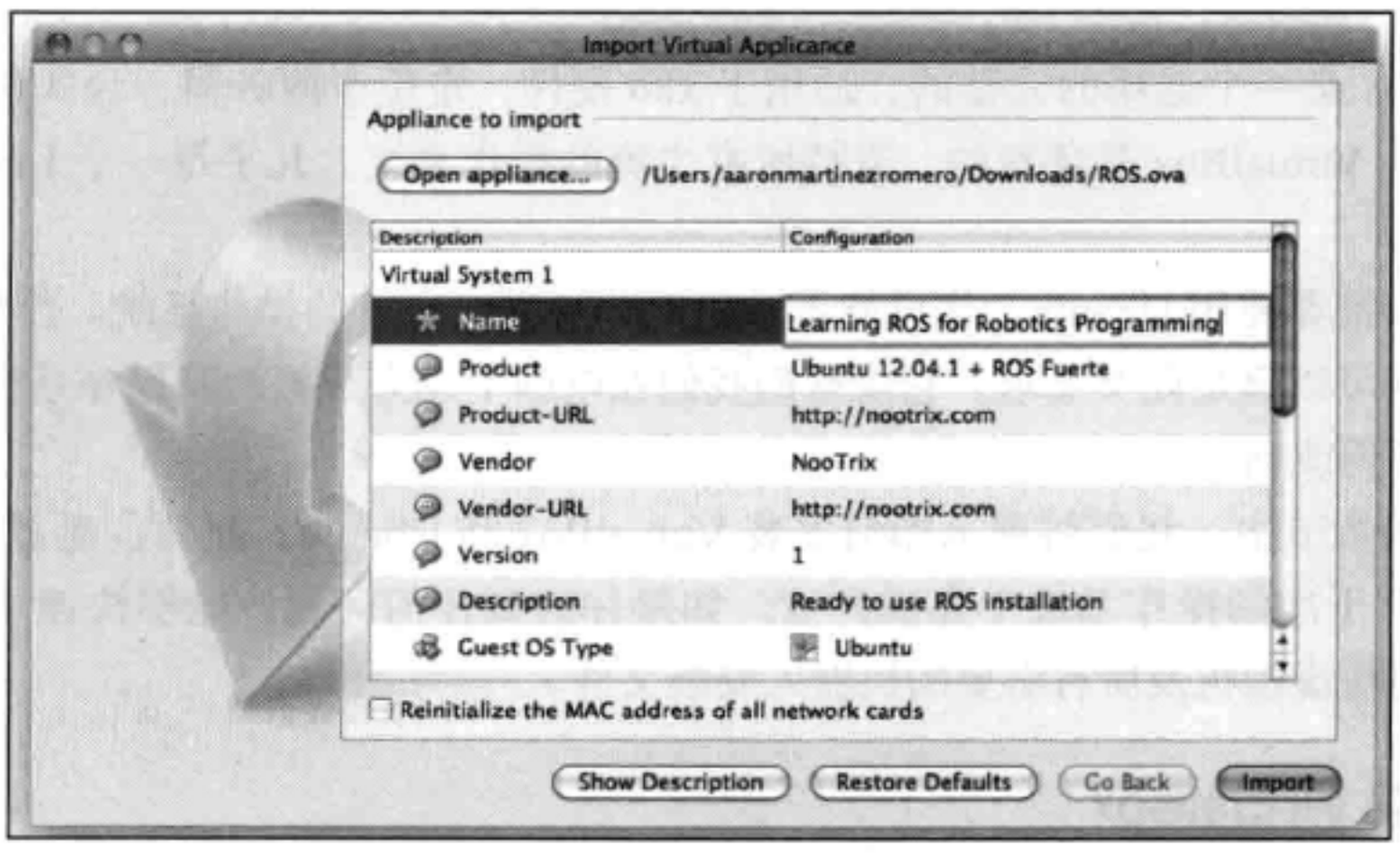
http://nootrix.com/wp-content/uploads/2012/08/ROS.ova。

你也可以找到带有 Ubuntu 和 ROS 的其他虚拟机，但是我们还是将使用这个 ROS 官网推荐的版本。

1.3.2 创建虚拟机

通过下载好的文件创建一个虚拟机非常简单，只需要按照本节的内容一步一步进行即可。打开 VirtualBox 虚拟机软件并打开 File | Import Appliance...，然后点击 Open appliance 并选择之前下载好的 `ROS.ova` 文件。

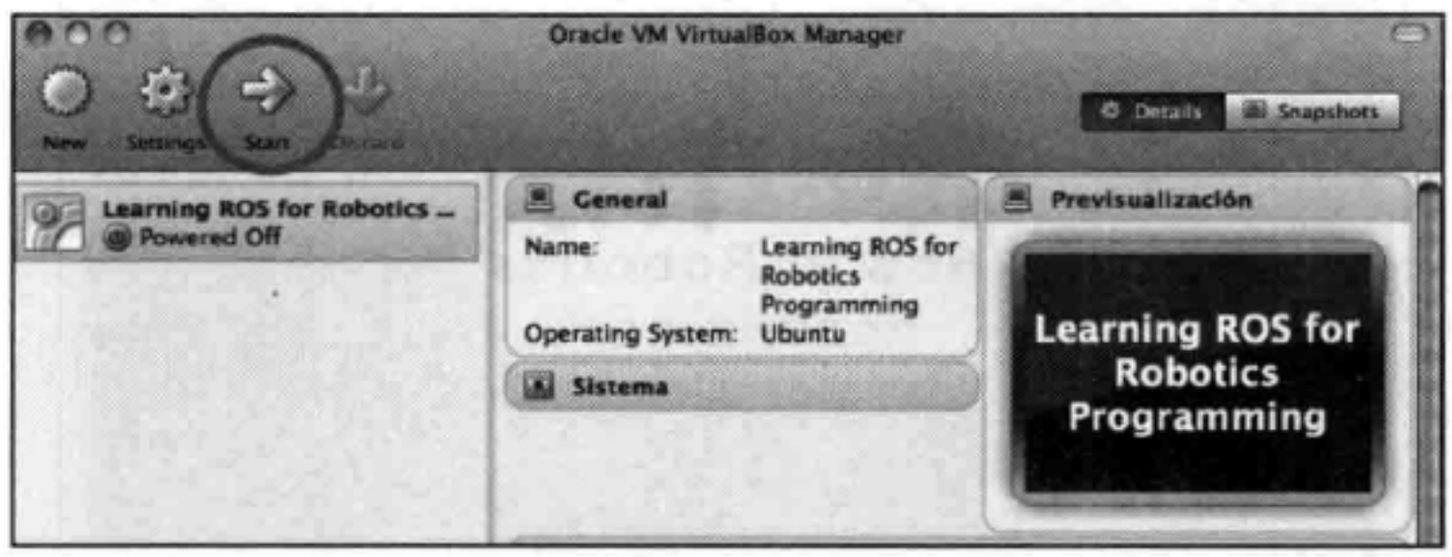
在下一个窗口中，可以配置新虚拟机的参数。我们保持默认配置并且仅仅改变虚拟机的名称。这个名称帮助我们区分不同的虚拟机。我们推荐给它起一个容易理解的名称，在这里我们使用本书的名称。



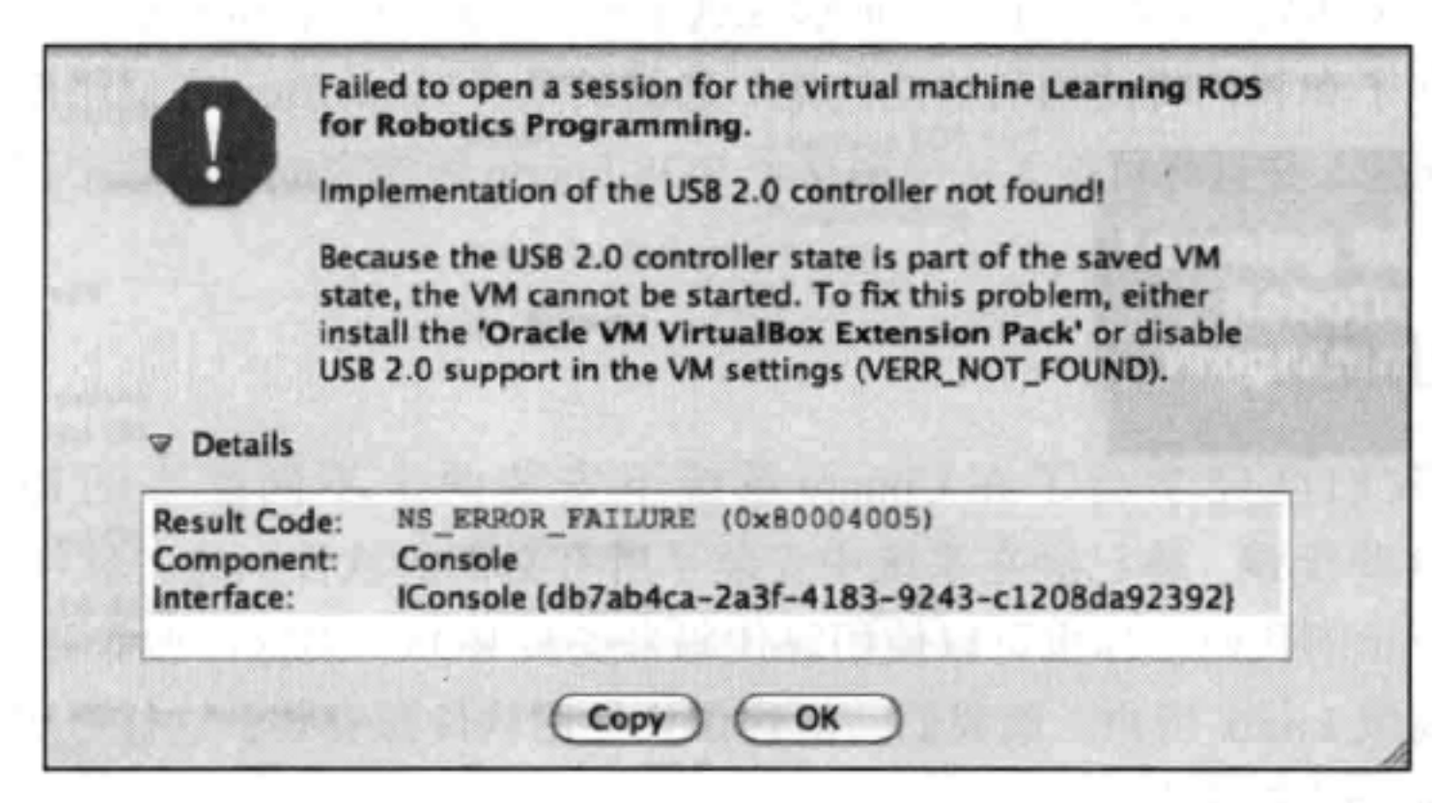
点击 Import 按钮，并在下一个窗口中接受软件授权许可。然后，你将看到一个进度条。这表明 VirtualBox 正在复制虚拟机镜像文件。

需要说明的是，这个过程并不会影响原有的 ROS.ova 文件，并且你可以通过对原文件进行多次复制创建多个虚拟机。

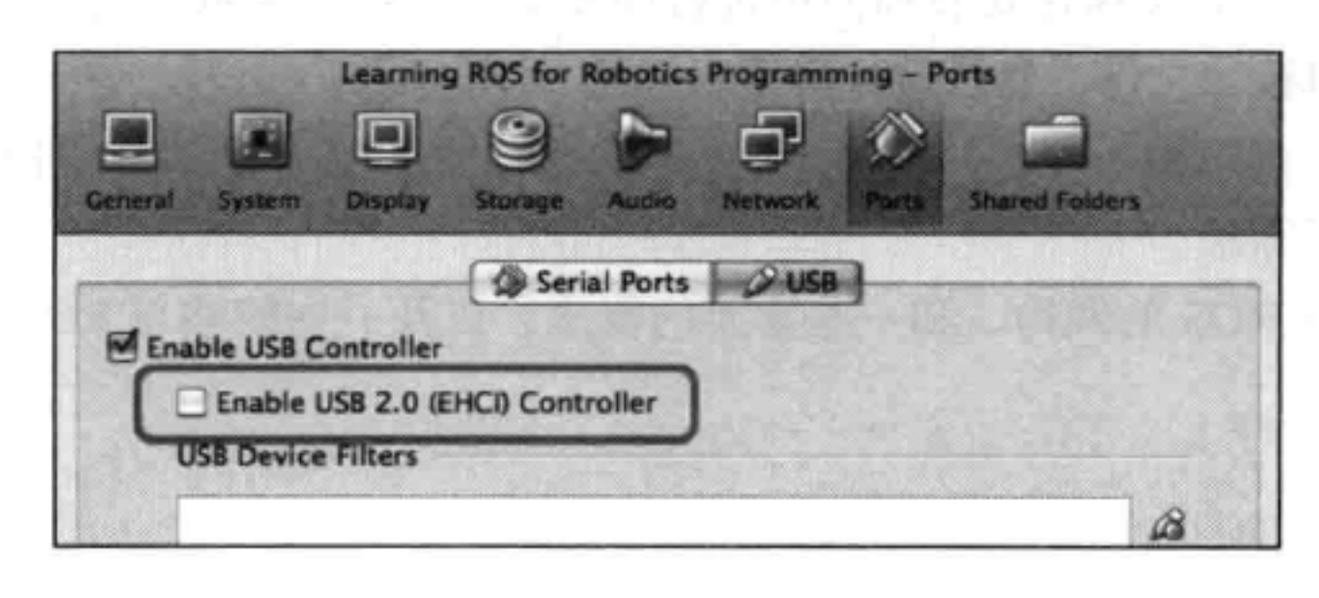
复制过程所需要的时间取决于你计算机的执行速度。当它完成时，你可以点击 Start 按钮启动你的虚拟机。需要注意的是，如果你的机器上有多个虚拟机，在启动前应该选择正确的那一个。当然，在我们这个例子里只有一个虚拟机。



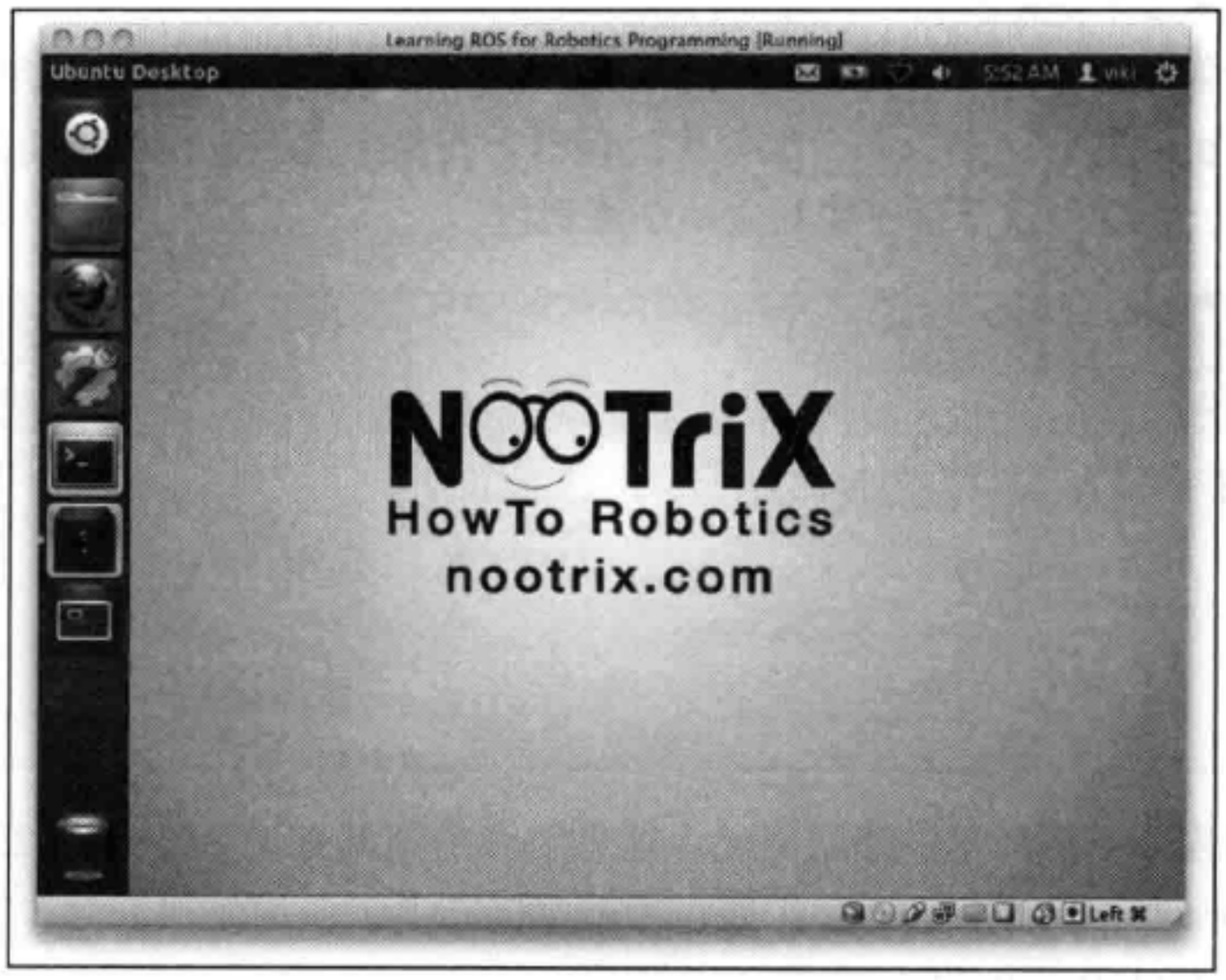
有时候会出现如下图所示的错误提示。这是因为电脑没有正确的 USB 2.0 驱动程序。你可以通过安装 Oracle VM VirtualBox 扩展包（Extension Pack）来解决问题，当然你也可以通过在虚拟机中禁用 USB 来解决。



为了禁用 USB，在虚拟机上右键单击并选择 Settings。在工具栏中，选择 Ports | USB，并取消勾选 Enable USB 2.0 (EHCI) Controller。重启虚拟机，就不会再出现任何问题。



虚拟机启动之后，你能看到完成 ROS 安装之后的 Ubuntu 12.04 界面，如下图显示：



当完成这些步骤后，你就有了一个能够在这本书中使用的完整版本的 ROS Fuerte。你可以运行所有的例子和我们将使用的功能包集。遗憾的是，VirtualBox 在使用部分实际外接设备的时候会有问题，并且你可能无法使用这个 ROS Fuerte 镜像完成第 4 章中列出的步骤。

1.4 本章小结

本章中，我们已经学会了在 Ubuntu 系统下安装两个不同版本的 ROS（Electric 和 Fuerte）。通过这些步骤，你已经在系统中安装了所有必需的软件，这些软件能够支持 ROS 和这本书的所有示例代码。你也可以使用源代码来安装 ROS。但这样做需要编译所有代码，因此只适用于高级 Linux 用户。而我们一般建议你使用软件源安装，这样做更加通用，且通常不会发生任何错误或问题。

如果你对 Ubuntu 系统不是很熟悉，那么建立一个虚拟机并在虚拟机上学习使用 ROS，会更加方便。这样，如果你在安装和使用过程中发生任何问题，都无需重新安装操作系统，只需要恢复虚拟机镜像文件，然后就可以重新开始。

通常情况下，虚拟机不能访问扩展出的实际硬件，如传感器和执行机构。尽管如此，你仍可以用它来测试算法。

下一章将学习 ROS 的架构、和一些重要的概念，以及一些能够与 ROS 进行直接交互的工具。

第 2 章
ROS 系统架构及示例

一旦你完成了 ROS 系统的安装，你肯定会想“好了，我已经安装完成，那么下一步是什么呢?”本章我们会学习 ROS 系统架构及它都包含哪些部分。然后，我们会开始创建节点和功能包，并使用 ROS 系统自带的 TurtleSim 示例。

ROS 系统的架构主要被设计和划分成了三部分，每一部分都代表一个层级的概念：

- 文件系统级（Filesystem level）
- 计算图级（Computation Graph level）
- 开源社区级（Community level）

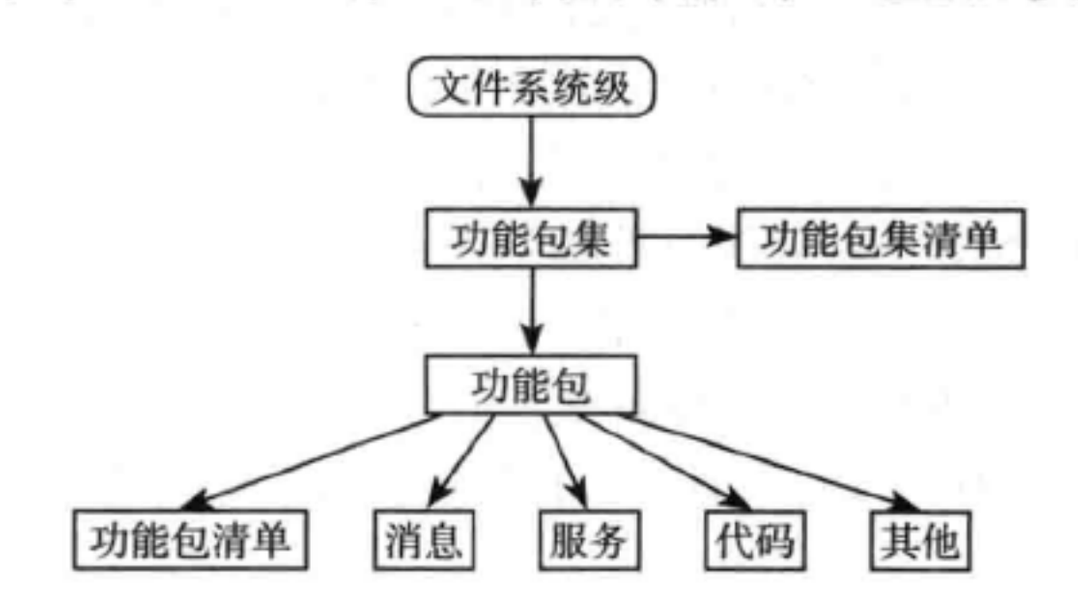
第一级是文件系统级。在这一级，我们将会使用一组概念来解释 ROS 的内部构成、文件夹结构，以及工作所需的核心文件。

第二级是计算图级，体现的是进程和系统之间的通信。在相关章节中，我们将看到 ROS 的各个概念和功能，包括建立系统、处理各类进程、与多台计算机通信等。

第三级是开源社区级，我们将解释一系列的工具和概念，其中包括在开发人员之间如何共享知识、算法和代码。这个层级是非常重要的，因为在开源社区的大力支持下 ROS 系统才得以快速成长。

2.1 理解 ROS 文件系统级

如果你刚接触 ROS，无论是准备使用 ROS 系统还是准备开发 ROS 项目，你都会觉得 ROS 中的各种概念非常奇怪。而一旦你已经驾轻就熟，那么这些概念就会变得亲切了。



与其他操作系统相类似，一个 ROS 程序的不同组件要被放在不同的文件夹下。这些文件夹是根据功能的不同来对文件进行组织的：

- **功能包（Package）** 功能包是 ROS 中软件组织的基本形式。一个功能包具有最小的结构和最少的内容，用于创建 ROS 程序。它可以包含 ROS 运行的进程（节点）、配置文件等。
- **功能包清单（Manifest）** 功能包清单提供关于功能包、许可信息、依赖关系、编译标志等的信息。功能包清单是一个 `manifests.xml` 文件，通过这个文件能够实现对功能包的管理。
- **功能包集（Stack）** 如果你将几个具有某些功能的功能包组织在一起，那么你将会获得一个功能包集。在 ROS 系统中，存在大量的不同用途的功能包集，例如导航功能包集。
- **功能包集清单（Stack manifest）** 功能包集清单（`stack.xml`）提供一个关于功能包集的清单，包括开源代码的许可证信息、与其他功能包集的依赖关系等。
- **消息类型（Message /msg type）** 消息是一个进程发送到其他进程的信息。ROS 系统有很多的标准类型消息。消息类型的说明存储在 `my_package/msg/MyMessageType.msg` 中，也就是对应功能包的 `msg` 文件夹下。
- **服务类型（Service/srv type）** 对服务的类型进行描述说明的文件在 ROS 系统中定义了服务的请求和响应的数据结构。这些描述说明存储在 `my_package/srv/ MyServiceType.srv` 中，也就是对应功能包的 `srv` 文件夹下。

```
chapter3_tutorials/
├── CMakeLists.txt
├── config
│   └── chapter3_tutorials.config
├── launch
│   ├── example1_gdb.launch
│   ├── example1.launch
│   ├── example1_valgrind.launch
│   ├── example2.launch
│   └── example3.launch
├── mainpage.dox
├── Makefile
├── manifest.xml
├── output
│   └── gdb_run_node_example1.txt
└── src
    ├── example1.cpp
    ├── example2.cpp
    └── example3.cpp

4 directories, 14 files
```

在右面的截屏中，你能看到本书第 3 章文件夹下的内容。该文件夹就是一个功能包，其中包含了本章示例代码、功能包清单等多项内容。第 3 章的文件中没有消息和服务。但如果有，你还可以看到 `srv` 和 `msg` 文件夹。

2.1.1 功能包

当每次提到功能包时，指的是一种特定的文件结构和文件夹组合。这种结构如下所示：

- `bin/` 这是我们编译和链接程序后，用于存储可执行文件的文件夹。
- `include/package_name/` 此目录包含了你所需要库的头文件。不要忘记导出功能包清单，因为它们还会被其他功能包所使用。
- `msg/` 如果你要开发非标准消息，请把文件放在这里。
- `scripts/` 其中包括 Bash、Python 或任何其他脚本的可执行脚本文件。
- `src/` 这是存储程序源文件的地方。你可能会为节点创建一个文件夹或按照你希望的方式去组织它。

- srv/ 这表示的是服务（srv）类型。
- CMakeLists.txt 这是 CMake 的生成文件。
- manifest.xml 这是功能包清单文件。

为了创建、修改或使用功能包，ROS 给我们提供了一些工具：

- rospack 使用此命令来获取信息或在系统中查找功能包。
- roscreate-pkg 当你想要创建一个新的功能包时，使用此命令。
- rosmake 使用此命令来编译功能包。
- rosdep 此命令安装功能包的系统依赖项。
- rxdeps 如果你想要查看功能包的依赖关系图，使用此命令。

若要在文件夹和功能包之间移动文件，ROS 提供了非常有用的 rosbash 功能包，其中包含了一些非常类似于 Linux 命令的命令。例如：

- roscd 此命令用于更改目录，类似于 Linux 中的 cd 命令。
- rosed 此命令用来编辑文件。
- roscp 此命令用于从一些功能包复制文件。
- rosd 此命令列出功能包的目录。
- rosls 此命令列出功能包下的文件，类似于 Linux 中的 ls 命令。

文件 manifest.xml 必须在功能包中，用来说明此功能包相关的各类信息。如果你发现在某个文件夹内包含有此文件，那么这个文件夹很可能是一个功能包。

打开一个 manifest.xml 文件，可以看到包的名称、依赖关系等信息。功能包清单的作用就是为了更容易地安装和分发这些功能包。

在功能包清单文件中使用的两个典型标记是 <depend> 和 <export>。<depend> 标记会显示当前功能包安装之前必须先安装哪些功能包。这是因为新的功能包会使用其他包的一些功能。<export> 标记告诉系统编译该功能包需要使用什么编译标志，引用哪些头文件等。

下面的代码段是此文件的示例：

```
<package>
  <description brief="short description">
    long description,
  </description>
  <author>Aaron Martinez, Enrique Fernandez</author>
  <license>BSD</license>
  <url>http://example.com/</url>
   <depend package="roscpp"/>
  <depend package="common"/>
  <depend package="otherPackage"/>
  <versioncontrol type="svn" url="https://urlofpackage/trunk"/>
  <export>
    <cpp cflags="-I${prefix}/include" lflags="-L${prefix}/lib -lros"/>
  </export>
</package>
```

2.1.2 功能包集

ROS 系统中的很多功能包被组织成 ROS 功能包集。如果说功能包的目标是创建易于代码复用的最小代码集合，那么功能包集的目标则是简化代码共享的过程。

功能包集同样是一个由文件和文件夹构成的基本结构。你可以手动创建它，但 ROS 为我们提供了 roscreate-stacck 命令行工具来简化这一过程。

功能包集必需的三个文件是：CMakeList.txt、Makefile 和 stack.xml。如果你在文件夹中见到 stack.xml，那么可以确定这是一个功能包集。

下面的代码段是此文件的示例：

```
<stack>
  <description brief="Sample_Stack">Sample_Stack1</description>
  <author>Maintained by AaronMR</author>
  <license>BSD,LGPL,proprietary</license>
  <review status="unreviewed" notes=""/>
  <url>http://someurl.blablabla</url>
  <depend stack="common_msgs" /> <!-- nav_msgs, sensor_msgs, geometry_
msgs -->
  <depend stack="ros_tutorials" /> <!-- turtlesim -->
</stack>
```

2.1.3 消息类型

ROS 使用了一种简化的消息类型描述语言来描述 ROS 节点发布的数据值。通过这样的描述语言，ROS 能够使用多种编程语言生成不同类型消息的源代码。

ROS 提供了很多预定义消息类型。如果你创建了一种新的消息类型，那么就要把消息的类型定义放到功能包的 msg/ 文件夹下。在该文件夹中，有用于定义各种消息的文件。这些文件都以 .msg 为扩展名。

消息类型必须具有两个主要部分：字段和常量。字段定义了要在消息中传输的数据的类型，例如 int32、float32、string 或之前创建的自定义类型，如叫做 type1 和 type2 的新类型。常量用于定义字段的名称。

一个消息文件的示例如下：

```
int32 id
float32 vel
string name
```

我们能够在下表中找到很多 ROS 消息所使用的标准数据类型：

基本类型	串行化	C++	Python
bool	Unsigned 8-bit int	uint8_t	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int
int16	Signed 16-bit int	int16_t	int

（续）

基本类型	串行化	C++	Python
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ASCII string (4-bit)	std::string	string
time	Secs/nsecs signed 32- bit ints	ros::Time	rospy.Time
duration	Secs/nsecs signed 32- bit ints	ros::Duration	rospy.Duration

ROS 消息中的一种特殊数据类型是报文头，主要用于添加时间戳、坐标位置等。报文头还允许对消息进行编号。通过在报文头内部附加信息，我们可以知道是哪个节点发出的消息，或者可以添加一些能够被 ROS 处理的其他功能。

报文头类型包含以下字段：

```
uint32 seq
time stamp
string frame_id
```

我们将在后续的章节中看到，正是通过报文头才能够记录当前机器人运行的时间戳和坐标位置。

在 ROS 中有一些处理消息的工具。例如：rosmsg 命令行工具能够输出的消息具体信息，并可以找到使用该消息类型的源文件。

在后面的章节中，我们将会学习如何使用正确的工具创建消息。

2.1.4 服务类型

ROS 使用了一种简化的服务描述语言来描述 ROS 的服务类型。这直接借鉴了 ROS 消息的数据格式，以实现节点之间的请求 / 响应通信。服务的描述存储在功能包的 srv/ 子目录下 .srv 文件中。

若要调用服务，你需要使用该功能包的名称及服务名称。例如，对于 sample_ package1/srv/sample1.srv 文件，可以将它称为 sample_package1/sample1 服务。

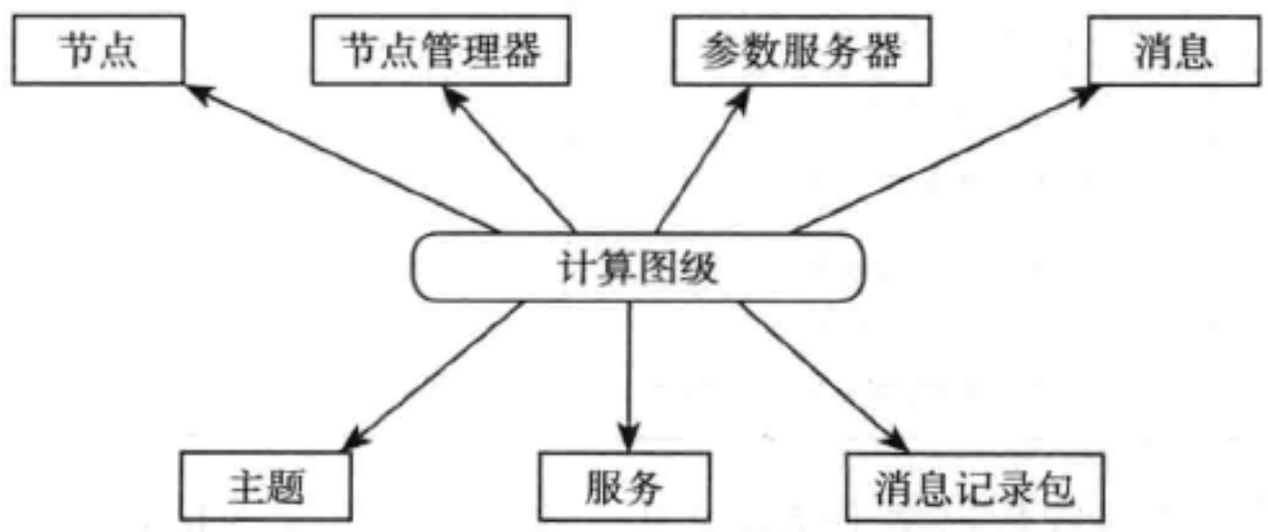
ROS 中有一些执行某些功能与服务的工具。rossrv 工具能输出服务说明、.srv 文件所在的功能包名称，并可以找到使用某一服务类型的源代码文件。

如果你想要在 ROS 中创建一个服务，可以使用服务生成器。这些工具能够从基本的服务说明中生成代码。你只需要在 CMakeLists.txt 文件中加一行 gensrv() 命令。

在后面的章节中，我们将会学习如何创建服务。

2.2 理解 ROS 计算图级

ROS 会创建一个连接到所有进程的网络。在系统中的任何节点都可以访问此网络，并通过该网络与其他节点交互，获取其他节点发布的信息，并将自身数据发布到网络上。

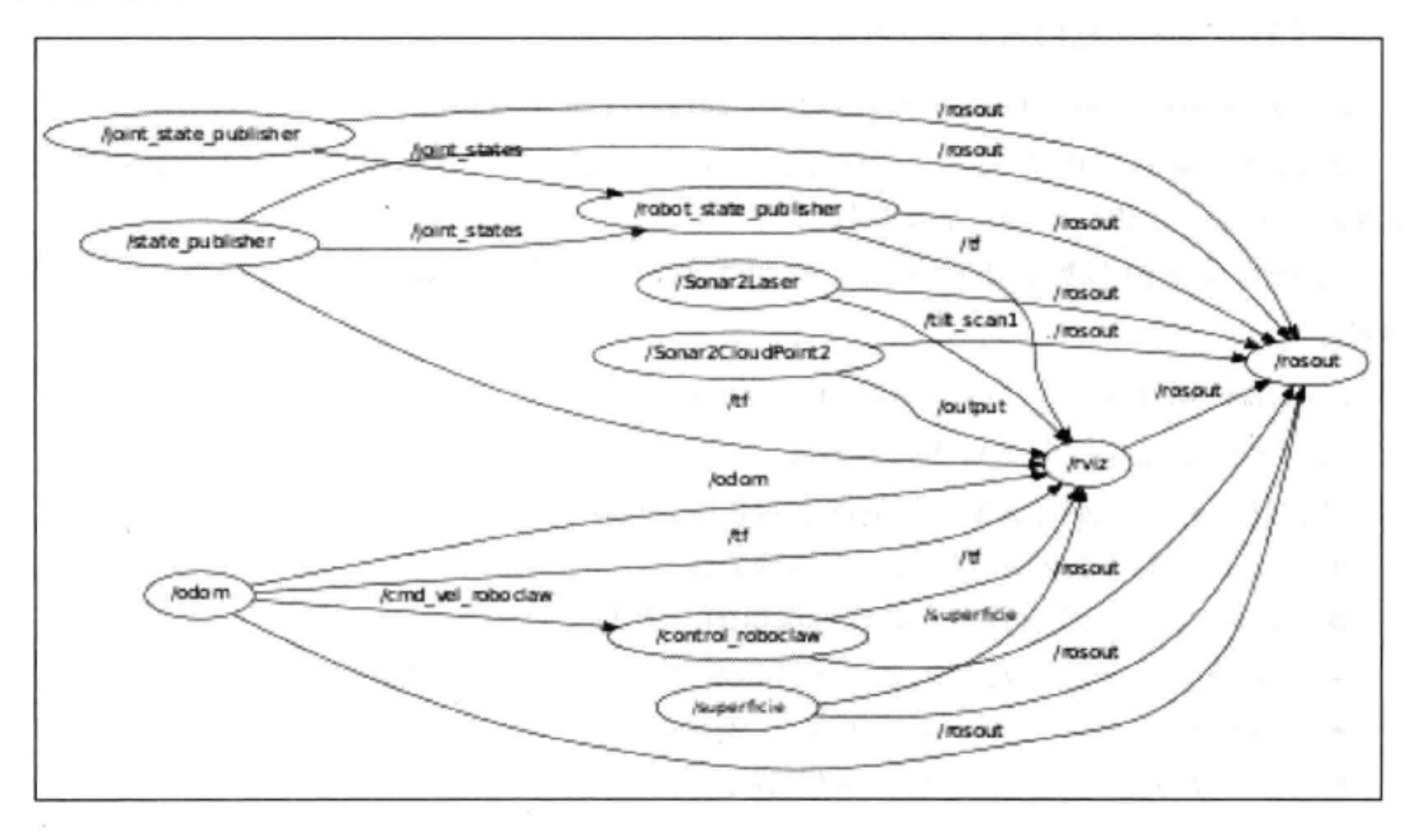


在这一层级中最基本的概念包括节点、节点管理器、参数服务器、消息、服务、主题和消息记录包，这些概念都以不同的方式向计算图级提供数据：

- **节点（Node）** 节点是主要的计算执行进程。如果你想要有一个可以与其他节点进行交互的进程，那么你需要创建一个节点，并将此节点连接到 ROS 网络。通常情况下，系统包含能够实现不同功能的多个节点。你最好让每一个节点都具有特定的单一的功能，而不是在系统中创建一个包罗万象的大节点。节点需要使用如 `roscpp` 或 `rospy` 的 ROS 客户端库进行编写。
- **节点管理器（Master）** 节点管理器用于节点的名称注册和查找等。如果在你的整个 ROS 系统中没有节点管理器，就不会有节点、服务、消息之间的通信。需要注意的是，由于 ROS 本身就是一个分布式网络系统，你可以在某一台计算机上运行节点管理器，在其他计算机上运行由该管理器管理的节点。
- **参数服务器（Parameter Server）** 参数服务器能够使数据通过关键词存储在一个系统的核心位置。通过使用参数，就能够在运行时配置节点或改变节点的工作任务。
- **消息（Message）** 节点通过消息完成彼此的沟通。消息包含一个节点发送到其他节点的数据信息。ROS 中包含很多种标准类型的消息，同时你也可以基于标准消息开发自定义类型的消息。
- **主题（Topic）** 主题是由 ROS 网络对消息进行路由和消息管理的数据总线。每一条消息都要发布到相应的主题。当一个节点发送数据时，我们就说该节点正在向主题发布消息。节点可以通过订阅某个主题，接收来自其他节点的消息。一个节点可以订阅一个主题，而并不需要该节点同时发布该主题。这就保证了消息的发布者和订阅者之间相互解耦，完全无需知晓对方的存在。主题的名称必须是独一无二的，否则在同名主题之间的消息路由就会发生错误。

- **服务（Service）** 在发布主题时，正在发送的数据能够以多对多的方式交互。但当你需要从某个节点获得一个请求或应答时，就不能通过主题来实现了。在这种情况下，服务能够允许我们直接与某个节点进行交互。此外，服务必须有一个唯一的名称。当一个节点提供某个服务时，所有的节点都可以通过使用 ROS 客户端库所编写的代码与它通信。
- **消息记录包（Bag）** 消息记录包是一种用于保存和回放 ROS 消息数据的文件格式。消息记录包是一种用于存储数据的重要机制。它能够获取并记录各种难以收集的传感器数据。我们可以通过消息记录包反复获取实验数据，进行必要的开发和算法测试。在使用复杂机器人进行实验工作时，需要经常使用消息记录包。

在下图中，你可以看到计算图级的图形化表示（节点状态图）。它表示了一个真实机器人在真实条件下系统的工作状态。在图中，你可以看到节点和主题，以及哪些节点订阅哪些主题等。此节点状态图中并没有消息、消息记录包、参数服务器和服务。这些内容需要使用其他工具进行图形化展示。用于创建该图表的工具是 rxgraph，在后面的章节中将学习到更多相关知识。



2.2.1 节点

节点都是各自独立的可执行文件，能够通过主题、服务或参数服务器与其他进程（节点）通信。ROS 通过使用节点将代码和功能解耦，提高了系统容错能力和可维护性，使系统简化。同时，节点允许了 ROS 系统能够布置在任意多个机器上并同时运行。

节点在系统中必须有唯一的名称。节点使用特定名称与其他节点进行通信而不产生

歧义。节点可以使用不同的库进行编写，如 roscpp 和 rospy。roscpp 基于 C++，而 rospy 基于 Python。在这本书里，我们将使用 roscpp。

ROS 提供了处理节点的工具，如 rosnode。rosnode 是一个用于显示节点信息的命令行工具，例如，列出当前正在运行的节点等。支持的命令如下所示：

- rosnode info node 输出当前节点信息。
- rosnode kill node 结束当前运行节点进程或发送给定信号。
- rosnode list 列出当前活动节点。
- rosnode machine hostname 列出某一特定计算机上运行的节点或列出主机名称。
- rosnode ping node 测试节点间的连通性。
- rosnode cleanup 将无法访问节点的注册信息清除。

在后面的章节中，我们将通过一些示例学习如何使用这些命令。

ROS 节点的一个强大功能是在启动该节点时更改参数。此功能使我们能够改变节点名称、主题名称和参数名称。我们无需重新编译代码就能重新配置节点，这样就可以在不同的场景中使用该节点。

一个改变主题名称的例子如下所示：

```
$ rosrun book_tutorials tutorialX topic1:=/level1/topic1
```

此命令将主题名称从 topic1 改为 /level1/topic1。我相信你现在还不是很明白，但在后面的章节中你会发现它的实用性。

更改节点中的参数和更改主题名称很类似。只需要在参数名称前添加一个下划线，例如：

```
$ rosrun book_tutorials tutorialX _param:=9.0
```

这样参数（param）就设置为浮点数 9.0。

请记住，你不能使用系统保留关键字的名称，包括：

- _name 为节点名称保留的一个特殊关键字。
- _log 为记录节点日志存储地址保留的一个关键字。
- _ip_hostname 表示 ROS_IP 和 ROS_HOSTNAME 的关键字。
- _master 表示 ROS_MASTER_URI 的关键字。
- _ns 表示 ROS_NAMESPACE 的关键字。

2.2.2 主题

主题是节点间用来传输数据的总线。通过主题进行消息路由不需要节点之间直接连接。这就意味着发布者和订阅者之间不需要知道彼此是否存在。同一个主题也可以有很多个订阅者。

每个主题都是强类型的，发布到主题上的消息必须与主题的 ROS 消息类型相匹配，并

且节点只能接收类型匹配的消息。一个节点要想订阅一个主题，它们就必须具有相同的消息类型。

ROS 的主题可以使用 TCP/IP 和 UDP 传输。基于 TCP 传输称为 TCPROS，它使用 TCP/IP 长连接。这是 ROS 默认的传输方式。

基于 UDP 传输称为 UDPROS，它是一种低延迟高效率的传输方式，但可能产生数据丢失。所以它最适合于远程操控任务。

ROS 有一个 rostopic 工具用于主题操作。它是一个命令行工具，允许我们获取主题的相关信息或直接在网络上发布数据。此工具的参数如下：

- `rostopic bw /topic` 显示主题所使用的带宽。
- `rostopic echo /topic` 将消息输出到屏幕。
- `rostopic find message_type` 按照类型查找主题。
- `rostopic hz /topic` 显示主题的发布率。
- `rostopic info /topic` 输出活动主题、发布的主题、主题订阅者和服务的信息。
- `rostopic list` 输出活动主题的列表。
- `rostopic pub /topic type args` 将数据发布到主题。它允许我们直接从命令行中对任意主题创建和发布数据。
- `rostopic type /topic` 输出主题的类型，或者说主题中发布的消息类型。

我们会在后面的章节中学习如何使用这些命令。

2.2.3 服务

当你需要直接与节点通信并获得应答时，将无法通过主题实现，而需要使用服务。

服务需要由用户开发，节点并不提供标准服务。包含消息源代码的文件存储在 srv 文件夹中。

像主题一样，服务关联一个以功能包中 .srv 文件名称来命名的服务类型。与其他基于 ROS 文件系统的类型一样，服务类型是功能包名称和 .srv 文件名称的组合。例如，chapter2_tutorials/srv/chapter2_srv1.srv 文件的服务类型是 chapter2_tutorials/chapter2_srv1。

ROS 关于服务的命令行工具有两个：rossrv 和 rosservice。我们可以通过 rossrv 看到有关服务数据结构的信息，并且与 rosmsg 具有完全一致的用法。通过 rosservice 可以列出服务列表和查询某个服务。支持的命令如下所示：

- `rosservice call /service args` 根据命令行参数调用服务。
- `rosservice find msg-type` 根据服务类型查询服务。
- `rosservice info /service` 输出服务信息。
- `rosservice list` 输出活动服务清单。
- `rosservice type /service` 输出服务类型。

- rosservice uri /service 输出服务的 ROSRPC URI。

2.2.4 消息

一个节点通过向特定主题发布消息，从而将数据发送到另一个节点。消息具有一定的类型和数据结构，包括 ROS 提供的标准类型和用户自定义类型。

消息的类型在 ROS 中按照以下标准命名方式进行约定：功能包名称 /.msg 文件名称。例如，std_msgs/msg/String.msg 的消息类型是 std_msgs/String。

ROS 使用命令行工具 rosmsg 来获取有关消息的信息。惯用参数如下所示：

- rosmsg show 显示一条消息的字段。
- rosmsg list 列出所有消息。
- rosmsg package 列出功能包的所有消息。
- rosmsg packages 列出所有具有该消息的功能包。
- rosmsg users 搜索使用该消息类型的代码文件。
- rosmsg md5 显示一条消息的 MD5 求和结果。

2.2.5 消息记录包

消息记录包是由 ROS 创建的一组文件。它使用 .bag 格式保存消息、主题、服务和其他 ROS 数据信息。你可以在事件发生后，通过使用可视化工具调用和回放数据，检查在系统中到底发生了什么；你可以播放、停止、后退及执行其他操作。

记录包文件可以像实时会话一样在 ROS 中再现情景，在相同时间向主题发送相同的数据。通常情况下，我们可以使用此功能来调试算法。若要使用记录包文件，我们可以使用以下 ROS 工具：

- rosbag 用来录制、播放和执行其他操作。
- rxbag 用于可视化图形环境中的数据。
- rostopic 帮助我们看到节点发送的主题。

2.2.6 节点管理器

ROS 节点管理器向 ROS 系统中其他节点提供命名和注册服务。它像服务一样跟踪和记录主题的发布者和订阅者。节点管理器的作用是使 ROS 节点之间能够相互查找。一旦这些节点找到了彼此，就能建立一种点对点的通信方式。

节点管理器还提供了参数服务器。节点管理器通常使用 roscore 命令运行，它会加载 ROS 节点管理器及其他 ROS 核心组件。

2.2.7 参数服务器

参数服务器是可通过网络访问的共享的多变量字典。节点使用此服务器来存储和检索运

行时的参数。

参数服务器使用 XML-RPC 实现并在 ROS 节点管理器下运行。这意味着它的 API 可通过通用的 XMLRPC 库进行访问。

参数服务器使用 XMLRPC 数据类型为参数赋值，包括以下类型：

- 32 位整数
- 布尔值
- 字符串
- 双精度浮点
- ISO 8601 日期
- 列表（List）
- 基于 64 位编码的二进制数据

ROS 中关于参数服务器的工具是 `rosparam`。其支持的参数如下所示：

- `rosparam list` 列出了服务器中的所有参数。
- `rosparam get parameter` 获取参数值。
- `rosparam set parameter value` 设置参数值。
- `rosparam delete parameter` 删除参数。
- `rosparam dump file` 将参数服务器保存到一个文件。
- `rosparam load file` 加载参数文件到参数服务器。

2.3 理解 ROS 开源社区级

ROS 开源社区级的概念主要关于 ROS 资源，能够通过独立的网络社区分享软件和知识。这些资源包括：

- **发行版（Distribution）** ROS 发行版是可以独立安装的、带有版本号的一系列功能包集。ROS 发行版像 Linux 发行版一样发挥类似的作用。这使得 ROS 软件安装更加容易，而且能够通过一个软件集合来维持一致的版本。
- **软件源（Repositorie）** ROS 依赖于共享开源代码与软件源的网站或主机服务，在这里不同的机构能够发布和分享各自的机器人软件与程序。
- **ROS Wiki** ROS Wiki 是用于记录有关 ROS 系统信息的主要论坛。任何人都可以注册账户和贡献自己的文件、提供更正或更新、编写教程以及其他信息。
- **邮件列表（Mailing list）** ROS 用户邮件列表是关于 ROS 的主要交流渠道，能够交流从 ROS 软件更新到 ROS 软件使用中的各种疑问或信息。

2.4 ROS 系统试用练习

现在，是时候对之前学习的内容进行一些练习了。在下面的几小节中，你会看到一组试

用 TurtleSim 的练习，其中包括如何建立功能包、使用节点、使用参数服务和移动一个虚拟机器人。通过这些练习，你会明确上面这些概念的具体含义。

2.4.1 ROS 文件系统导览

我们通过命令行工具来浏览一下 ROS 的文件系统。我们将要解释最常用的部分。

为了获得功能包和功能包集的信息，或移动文件，需要使用 rospack、rosstack、roscd 和 rosls 命令。

我们使用 rospack 和 rosstack 来获取有关功能包、功能包集、路径和依赖性等信息。例如，如果你想要找 turtlesim 包的路径，可以使用以下命令：

```
$ rospack find turtlesim
```

你将获得以下信息：

```
/opt/ros/fuerte/share/turtlesim
```

同样的，如果你想要找到你已经在系统中安装过的某个功能包集，示例如下：

```
$ rosstack find 'nameofstack'
```

想要获得功能包或功能包集下面的文件列表，那么需要使用：

```
$ rosls turtlesim
```

你将获得以下信息：

```
cmake     images     mimic   srv          turtle_teleop_key
draw_square  manifest.xml  msg turtlesim_node
```

如果你想进入到某个文件夹，可以使用 roscd 命令：

```
$ roscd turtlesim
$ pwd
```

你将获得以下新路径：

```
/opt/ros/fuerte/share/turtlesim
```

2.4.2 创建工作空间

在开始具体工作之前，首先创建工作空间。在这个工作空间中，我们将会完成本书中所使用的所有代码。

若要查看 ROS 正在使用的工作空间，请使用下面的命令：

```
$ echo $ROS_PACKAGE_PATH
```

你将会看到如下信息：

```
/opt/ros/fuerte/share:/opt/ros/fuerte/stacks
```

我们将要创建的文件夹是在 ~/dev/rosbook/ 中。若要新建此文件夹，使用以下命令：

```
$ cd ~
$ mkdir -p dev/rosbook
```

一旦我们在目录下建好了文件夹，就有必要将此新路径添加到 ROS_PACKAGE_ PATH。要做到这一点，只需要在 ~/.bashrc 文件的末尾添加一个新行：

```
$ echo "export ROS_PACKAGE_PATH=~/dev/rosbook:${ROS_PACKAGE_PATH}" >>
~/.bashrc
$ . ~/.bashrc
```

现在我们已经在 ROS 系统中创建和配置完成自己的新文件夹。

2.4.3 创建 ROS 功能包

就像之前所说，你也可以手动创建功能包。但是为了避免那些繁琐的工作，最好使用 roscreate-pkg 命令行工具。

使用以下命令在之前建立的文件夹下创建新的功能包：

```
$ cd ~/dev/rosbook
$ roscreate-pkg chapter2_tutorials std_msgs rospy roscpp
```

此命令的格式包括功能包的名称和依赖项；在这个示例中，依赖项包括 std_msgs、rospy 和 roscpp。如以下命令行所示：

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

这些依赖项包括：

- std_msgs　包含了常见消息类型，表示基本数据类型和其他基本的消息构造，如多维数组。
- rospy　一个 ROS 的纯 Python 客户端库。
- roscpp　使用 C++ 实现 ROS 的各种功能。它提供了一个客户端库。程序员能够调用这些接口快速完成与 ROS 的主题、服务和参数相关的开发工作。

如果所有步骤都正确执行，结果如下图所示：

```
Created package directory /home/aaronmr/dev/rosbook/chapter2_tutorials
Created include directory /home/aaronmr/dev/rosbook/chapter2_tutorials/include/chapter2_tutorials
Created cpp source directory /home/aaronmr/dev/rosbook/chapter2_tutorials/src
Created package file /home/aaronmr/dev/rosbook/chapter2_tutorials/Makefile
Created package file /home/aaronmr/dev/rosbook/chapter2_tutorials/manifest.xml
Created package file /home/aaronmr/dev/rosbook/chapter2_tutorials/CMakeLists.txt
Created package file /home/aaronmr/dev/rosbook/chapter2_tutorials/mainpage.dox

Please edit chapter2_tutorials/manifest.xml and mainpage.dox to finish creating your package
```

正如我们之前看到的，你可以使用 rospack、roscd 和 rosls 命令来获取新的功能包信息：

- rospack find chapter2_tutorials　此命令用于查找路径。
- rospack depends chapter2_tutorials　此命令用于查看依赖关系。

- rosls chapter2_tutorials 此命令用于查看内容。
- roscd chapter2_tutorials 此命令会更改实际路径。

2.4.4 编译 ROS 功能包

一旦你创建了一个功能包，并且编写了一些代码，就需要编译功能包了。当你编译功能包的时候，主要是代码的编译过程。

为了编译功能包，可以使用 rosmake 命令：

```
$ rosmake chapter2_tutorials
```

在几秒之后，你会看到：

```
[ rosmake ] Results:
[ rosmake ] Cleaned 5 packages.
[ rosmake ] Built 5 packages with 0 failures.
[ rosmake ] Summary output to directory
[ rosmake ] /home/aaronmr/.ros/rosmake/rosmake_output-20130713-183756
```

如果没有看到错误提示信息，说明功能包编译成功。

2.4.5 使用 ROS 节点

正如我们在 2.2.1 章节中解释的，节点都是可执行程序，这些可执行文件位于 packagename/bin 目录中。要学习和了解有关节点的知识，我们要使用一个名为 turtlesim 的功能包进行练习。

如果你进行了 ROS 系统的完整安装，那么你已经有了 turtlesim 功能包。如果还没有，请使用以下命令安装：

```
$ sudo apt-get install ros-fuerte-ros-tutorials
```

在开始之前，必须使用如下命令启动 roscore：

```
$ roscore
```

为了获得节点信息，可以使用 rosnode 命令。为了查看命令接受哪些参数，可以输入以下命令：

```
$ rosnode
```

你会获得一个可接受参数的清单，如下图所示：

```
rosnode is a command-line tool for printing information about ROS Nodes.

Commands:
        rosnode ping    test connectivity to node
        rosnode list    list active nodes
        rosnode info    print information about node
        rosnode machine list nodes running on a particular machine or list mach
ines
        rosnode kill    kill a running node

Type rosnode <command> -h for more detailed usage, e.g. 'rosnode ping -h'
```

如果你想获得关于这些参数更详细的解释，请使用以下命令：

```
$ rosnode <param> -h
```

现在 roscore 已经运行，我们想要获取正在运行的节点的相关信息：

```
$ rosnode list
```

你会看到运行的节点仅有 /rosout。这是正常的，因为这个节点总是随着 roscore 的运行而运行。

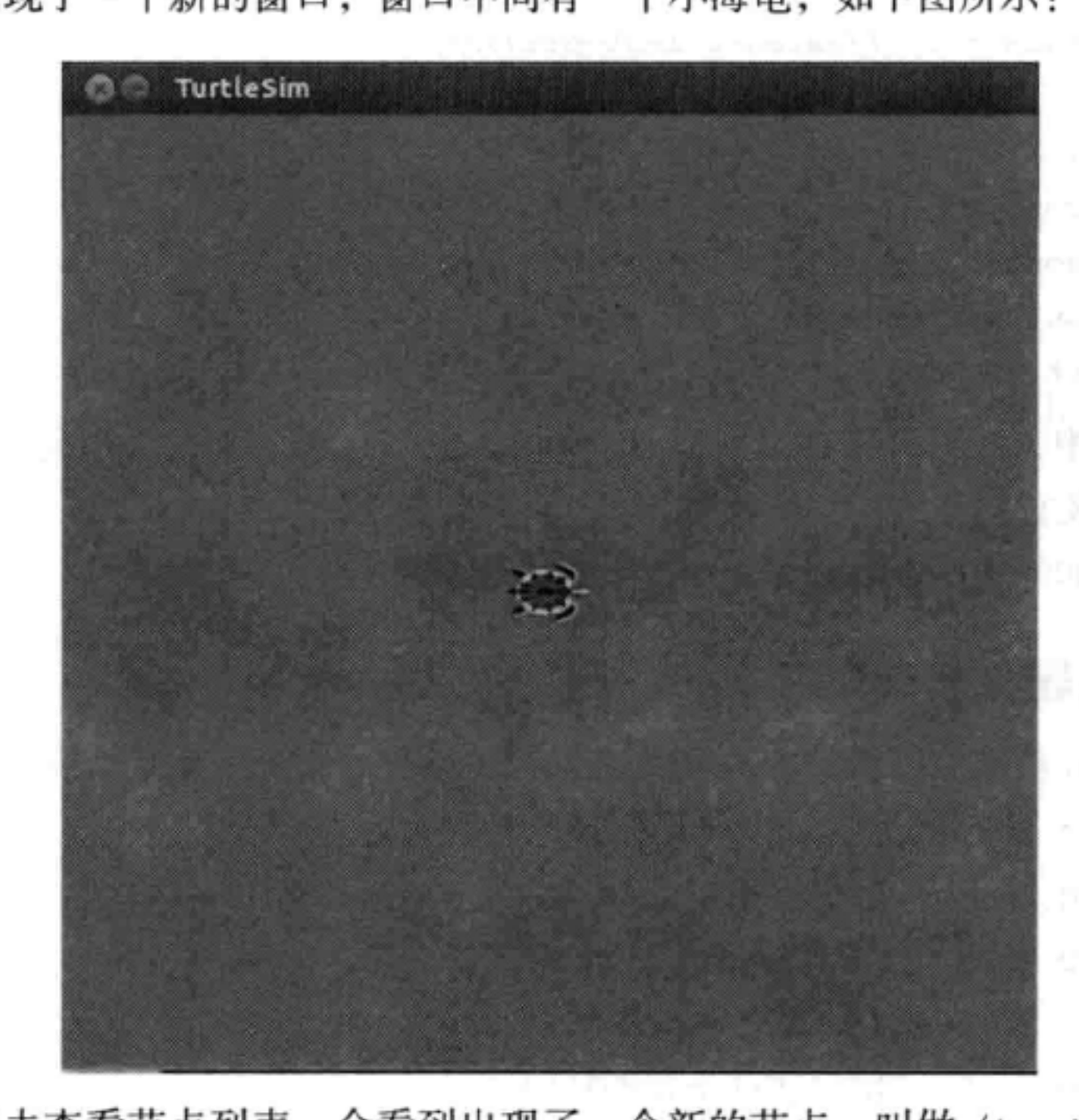
通过使用参数我们可以获得此节点的所有信息。也可以使用下列命令获得更详细的信息：

```
$ rosnode info
$ rosnode ping
$ rosnode machine
$ rosnode kill
```

现在我们要用 rosrun 命令启动一个新的节点，如下所示：

```
$ rosrun turtlesim turtlesim_node
```

我们看到出现了一个新的窗口，窗口中间有一个小海龟，如下图所示：



如果我们再去查看节点列表，会看到出现了一个新的节点，叫做 /turtlesim。

你可以通过使用 rosnode info Nodename 命令查看节点信息。你可以看到很多可以用于程序调试的信息：

```
$ rosnode info /turtlesim
Node [/turtlesim]
Publications:
 * /turtle1/color_sensor [turtlesim/Color]
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/pose [turtlesim/Pose]
Subscriptions:
 * /turtle1/command_velocity [unknown type]
Services:
 * /turtle1/teleport_absolute
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
 * /reset
 * /spawn
 * /clear
 * /turtle1/set_pen
 * /turtle1/teleport_relative
 * /kill
contacting node http://aaronmr-laptop:42791/ ...
Pid: 28337
Connections:
 * topic: /rosout
  * to: /rosout
  * direction: outbound
  * transport: TCPROS
```

在以上信息中，我们可以看到发布者（及相应主题）、订阅者（及相应主题）和该节点具有的服务（srv）及它们各自唯一的名称。

接下来介绍如何使用主题和服务与该节点进行交互。

2.4.6 使用主题与节点交互

要进行交互并获取主题的信息，可以使用 rostopic 工具。此工具接受以下参数：

- rostopic bw　显示主题所使用的带宽。
- rostopic echo　将消息输出到屏幕。
- rostopic find　按照类型查找主题。
- rostopic hz　显示主题的发布频率。
- rostopic info　输出活动主题的信息。
- rostopic list　输出活动主题的列表。
- rostopic pub　将数据发布到主题。

- rostopic type 输出主题的类型。

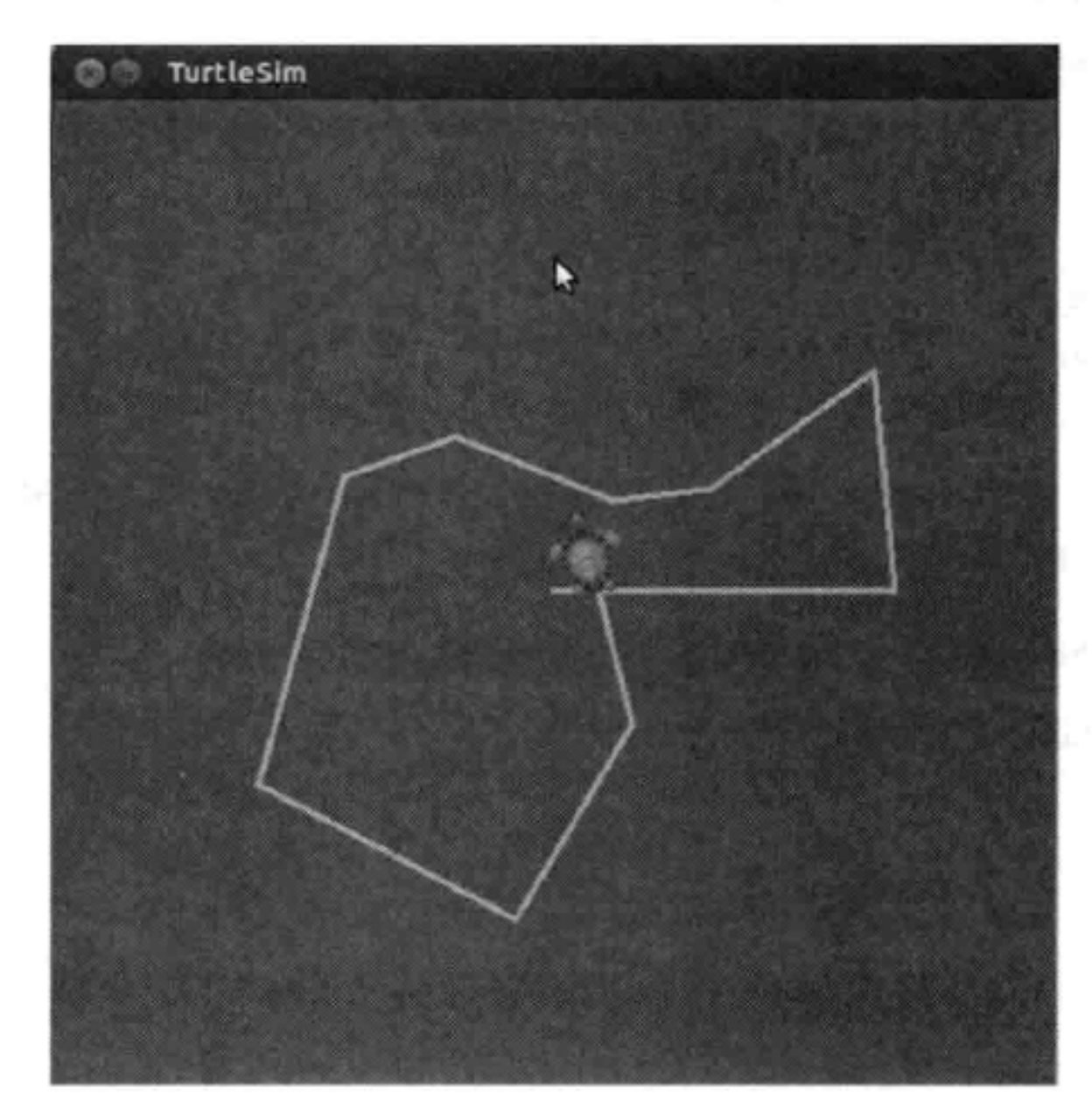
如果想要查看有关这些参数的详细信息，请使用 -h，如下所示：

```
$ rostopic bw -h
```

通过使用 pub 参数，可以发布任何节点都可以订阅的主题。我们只需要用正确的名称将主题发布出去。我们将会在以后做这个测试，现在要使用一个节点，并让节点做如下工作：

```
$ rosrun turtlesim turtle_teleop_key
```

通过节点订阅的主题，我们可以使用箭头键移动乌龟，如下图所示：



为什么 turtle_teleop_key 执行时，小海龟会移动呢？

如果你想要看到 /teleop_turtle 和 /turtlesim 节点的信息，可以看到在下面的代码中，在发布者（Publications）部分的第一个节点有一个主题叫 */turtle1/ command_velocity [turtlesim/Velocity]；在订阅者（Subscriptions）部分的第二个节点有 */turtle1/command_ velocity [turtlesim/Velocity]:

```
$ rosnode info /teleop_turtle

Node [/teleop_turtle]
...
Publications:
 * /turtle1/command_velocity [turtlesim/Velocity]
 ...
```

```
$ rosnode info /turtlesim

Node [/turtlesim]
...
Subscriptions:
 * /turtle1/command_velocity [turtlesim/Velocity]
...
```

这意味着前面的节点发布了一个主题，而后面的节点订阅了这个主题。

你可以使用以下命令查看主题清单：

```
$ rostopic list

/rosout
/rosout_agg
/turtle1/color_sensor
/turtle1/command_velocity
/turtle1/pose
```

通过使用 echo 参数，可以查看节点发出的信息。运行以下命令行并使用箭头键触发消息，查看消息产生时发送了哪些数据：

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

你会看到类似如下显示：

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

你可以使用以下命令行查看由主题发送的消息类型：

```
$  rostopic type /turtle1/command_velocity

turtlesim/Velocity
```

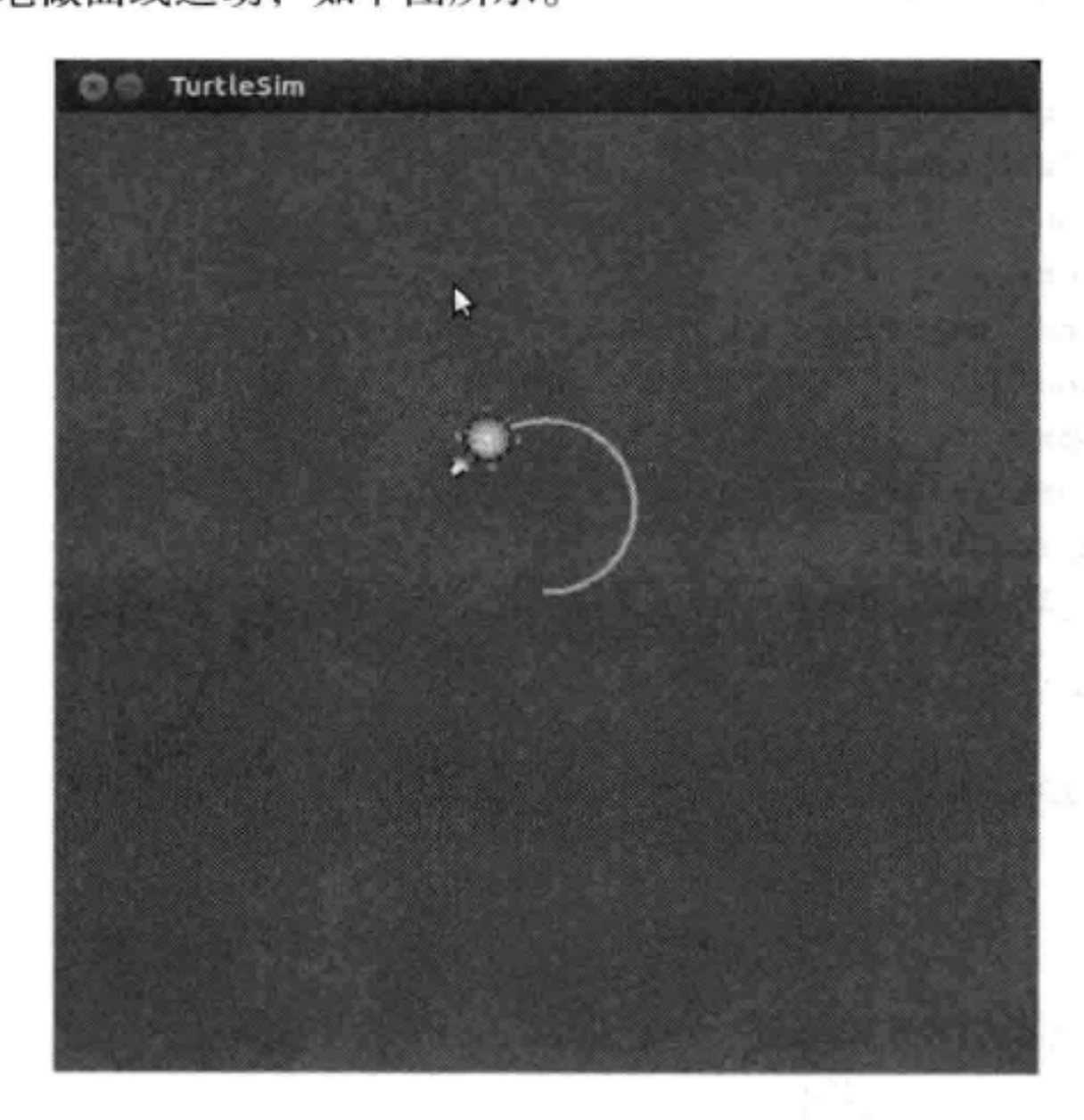
如果你想要看到消息字段，可以使用以下命令：

```
$ rosmsg show turtlesim/Velocity

float32 linear
float32 angular
```

这些工具非常有用，因为我们可以通过这些工具使用 rostopic pub [topic] [msg_type] [args] 命令直接发布主题：

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

你会看到小海龟做曲线运动，如下图所示。



2.4.7 学习如何使用服务

服务是能够使节点之间相互通信的另一种方法。服务允许节点发送请求和接收响应。我们要使用 rosservice 工具与服务进行交互。此命令接受的参数如下所示：

- rosservice args /service 输出服务参数。
- rosservice call /service args 根据命令行参数调用服务。
- rosservice find msg-type 根据服务类型查询服务。
- rosservice info /service 输出服务信息。
- rosservice list 输出活动服务清单。
- rosservice type /service 输出服务类型。
- rosservice uri /service 输出服务的 ROSRPC URI。

我们要使用以下命令列出在 turtlesim 节点运行时系统提供的服务。如果你运行了这个命令，却没有任何反应，那么请记住以后要先运行 roscore 并启动 turtlesim 节点：

```
$ rosservice list
```

你会获得以下输出：

```
/clear
/kill
```

```
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtle2/set_pen
/turtle2/teleport_absolute
/turtle2/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

如果你想查看某个服务的类型，例如 /clear 服务，请使用：

```
$ rosservice type /clear
```

你会获得：

```
std_srvs/Empty
```

若要调用服务，你需要使用 rosservice call [service] [args] 命令。所以如果想要调用 /clear 服务，请使用：

```
$ rosservice call /clear
```

在 turtlesim 的窗口中，你会看到由于小海龟移动所产生的线条消失了。

现在我们尝试其他的服务，例如 /spawn 服务。这项服务将在不同的方向和不同的位置创建另一只小海龟。开始之前，我们要去查看以下类型的消息：

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

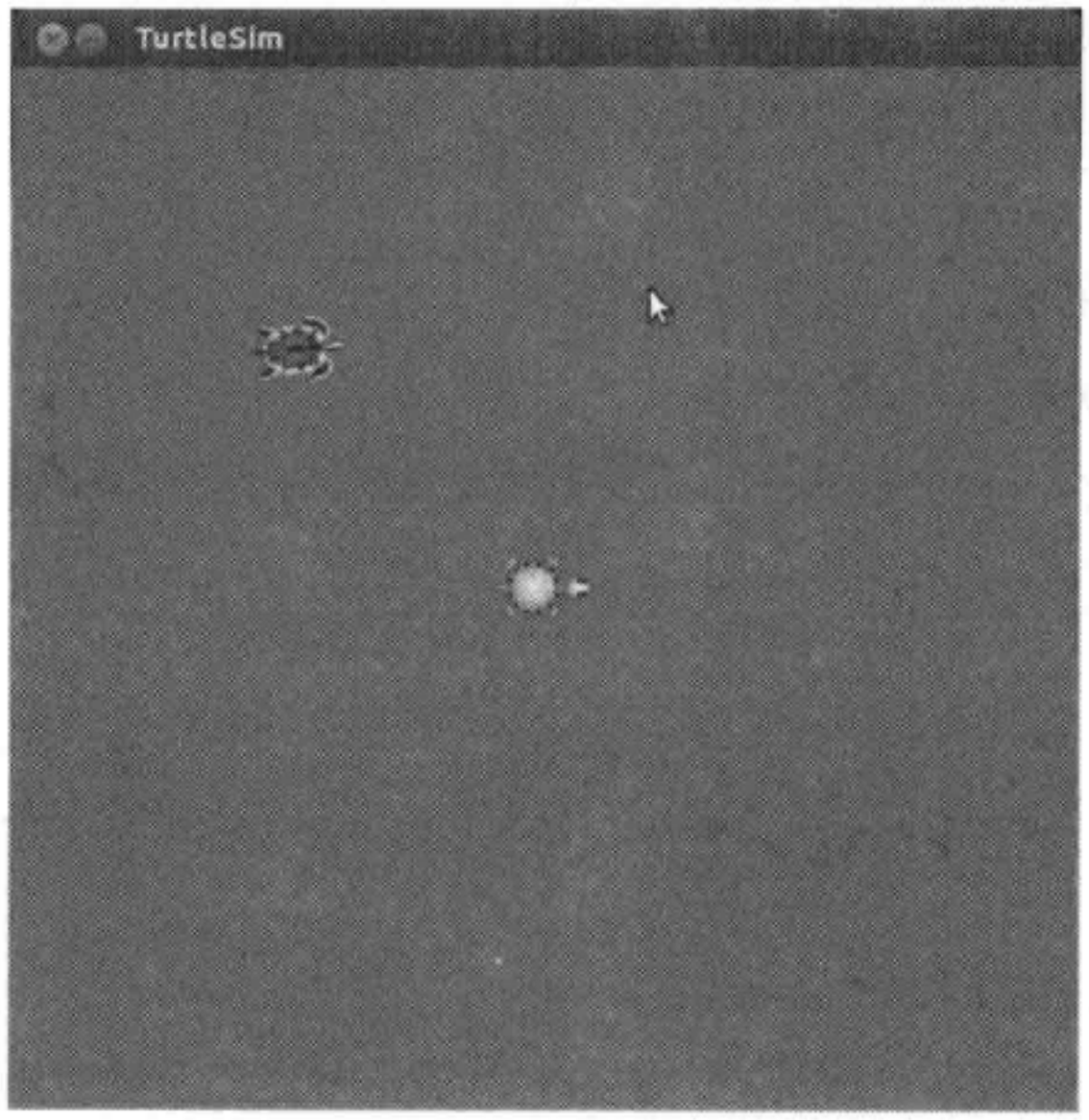
我们会获得以下参数：

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

通过这些字段，可以知道如何来调用服务。我们需要新海龟位置的 *x* 和 *y*、方向 *theta* 和新海龟的名称：

```
$ rosservice call /spawn 3 3 0.2 "new_turtle"
```

我们会获得以下结果：



2.4.8 使用参数服务器

参数服务器用于存储所有节点均可访问的共享数据。ROS 中用来管理参数服务器的工具称为 rosparam。接受的参数如下所示：

- rosparam set parameter value　设置参数值。
- rosparam get parameter　获取参数值。
- rosparam load file　加载参数文件到参数服务器。
- rosparam delete parameter　删除参数。
- rosparam dump file　将参数服务器保存到一个文件。
- rosparam list　列出了服务器中的所有参数。

例如，查看被所有节点所使用的服务器参数：

```
$ rosparam list
```

我们会获得以下输出：

```
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_aaronmr_laptop__60878
/rosversion
/run_id
```

上面的背景（background）参数是 turtlesim 节点的参数。窗口初始化为蓝色，这些参数可以改变窗口的颜色。如果你想要读取某个值，可以使用 get 参数：

```
$ rosparam get /background_b
```

为了设定一个新的值，可以使用 set 参数：

```
$ rosparam set /background_b 100
```

命令行工具 rosparam 的另外一个重要特性是参数转存。通过该参数，你可以保存或加载参数服务器的内容。

我们使用 rosparam dump [file_name] 来保存参数服务器：

```
$ rosparam dump save.yaml
```

使用 rosparam load [file_name] [namespace] 向参数服务器加载一个新的数据文件：

```
$ rosparam load load.yaml namespace
```

2.4.9 创建节点

在本节中，我们要学习如何创建两个节点：一个发布一些数据，另一个接收这些数据。也就是说，两个节点之间使用最基本的方式进行通信，发送和接收数据并使用这些数据来做些工作：

```
$ roscd chapter2_tutorials/src/
```

创建两个文件并分别命名为 example1_a.cpp 和 example1_b.cpp。

example1_a.cpp 文件将会发送带有节点名称的数据，example1_b.cpp 文件会把这些数据显示在 shell 窗口中。

将下面的代码复制到 example1_a.cpp 文件或者从下载的本书示例代码中找到它：

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example1_a");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_
msgs::String>("message", 1000);
  ros::Rate loop_rate(10);
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << " I am the example1_a node ";
```

```
    msg.data = ss.str();
    //ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

这里是一些关于上面代码的进一步解释。要包含的头文件是 ros/ros.h、std_msgs/String.h 和 sstream。其中，ros/ros.h 包含了使用 ROS 节点所有必要的文件，而 std_msgs/String.h 则包含了我们要使用的消息类型：

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
```

启动该节点并设置其名称，请记住该名称必须是唯一的：

```
ros::init(argc, argv, "example1_a");
```

下面是设置节点进程的句柄：

```
ros::NodeHandle n;
```

将节点设置成发布者，并将所发布主题的类型和名称告知节点管理器。第一个参数是消息的名称，设置为 message；第二个参数是缓冲区的大小。如果主题发布数据速度较快，那么将缓冲区设置为 1000 个消息，如下所示：

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

在本例中，设置发送数据的频率为 10Hz：

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

当收到 Ctrl + C 的按键消息或 ROS 停止当前节点运行时，ros : : ok() 库会执行停止节点运行的命令：

```
while (ros::ok())
{
```

在这里，我们创建一个消息变量，变量的类型必须符合发送数据的要求：

```
std_msgs::String msg;
std::stringstream ss;
ss << " I am the example1_a node ";
msg.data = ss.str();
```

这样，消息被发布：

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

如果有一个订阅者出现，ROS 就会更新和读取所有主题：

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

按照 10Hz 的频率将程序挂起：

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

将下面的代码复制到 example1_b.cpp 文件或者从下载的本书示例代码中找到它：

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

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

int main(int argc, char **argv)

{
  ros::init(argc, argv, "example1_b");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("message", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

这里是一些对上面代码的进一步解释。首先要包含头文件和主题所使用的消息类型：

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

每次该节点收到一条消息时都将调用此函数，我们就可以使用或处理数据。在本示例中，我们将收到的数据在命令行窗口中输出出来：

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

ROS_INFO() 用于在命令行窗口中输出数据。

创建一个订阅者，并从主题获取以 message 为名称的消息数据。设置缓冲区为 1000 个消息，处理消息句柄的回调函数为 messageCallback：

```
ros::Subscriber sub = n.subscribe("message", 1000, messageCallback);
```

ros : : spin() 库是节点读取数据的消息响应循环，当消息到达的时候，回调函数 messageCallback 就会被调用。当用户按下 Ctrl + C 时，节点会退出消息循环，于是循环结束：

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

2.4.10 编译节点

当使用 chapter2_tutorials 功能包时，需要自行编辑 CMakeLists.txt 文件。你

可以使用你喜欢的编辑器或直接使用 rosed 工具。这里我们将会在 VIM 编辑器下打开这个文件：

```
$ rosed chapter2_tutorials CMakeLists.txt
```

将以下命令行复制到文件的末尾处：

```
rosbuild_add_executable(example1_a src/example1_a.cpp)
rosbuild_add_executable(example1_b src/example1_b.cpp)
```

这些命令行会在 /bin 文件夹下创建两个可执行文件。

现在我们使用 rosmake 工具编译功能包就会编译全部的节点：

```
$ rosmake chapter2_tutorials
```

如果在你的电脑上还没有启动 ROS，需要首先调用：

```
$ roscore
```

你可以使用 rosnode list 命令检查 ROS 是否运行：

```
$ rosnode list
```

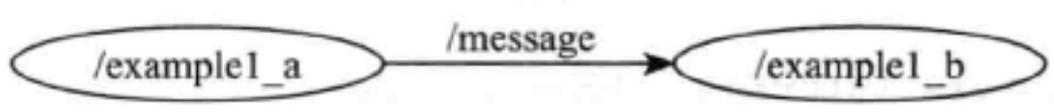
然后，在不同的 shell 窗口下分别运行以下命令：

```
$ rosrun chapter2_tutorials example1_a
$ rosrun chapter2_tutorials example1_b
```

如果你检查一下 example1_b 正在运行的 shell 窗口，你就会看到以下信息：

```
...
[ INFO] [1355228542.283236850]: I heard: [ I am the example1_a node ]
[ INFO] [1355228542.383221843]: I heard: [ I am the example1_a node ]
[ INFO] [1355228542.483249861]: I heard: [ I am the example1_a node ]
...
```

我们可以在下图中看到正在发生的消息传递。example1_a 节点发布消息到 message 主题，同时节点 example2_b 订阅了这个主题。



你可以使用 rosnode 和 rostopic 命令来调试和查看当前节点的运行状况。尝试使用以下命令：

```
$ rosnode list
$ rosnode info /example1_a
$ rosnode info /example1_b
$ rostopic list
$ rostopic info /message
$ rostopic type /message
$ rostopic bw /message
```

2.4.11 创建 msg 和 srv 文件

在这一节中，我们将会学习如何在节点中创建 msg 和 srv 文件。它们是用于定义传输

数据的类型和数据值的文件。ROS 会根据这些文件内容自动地为我们创建所需的代码，以便 msg 和 srv 文件能够被节点所使用。第一步，我们先学习 msg 文件。

在上一节所使用的示例中，我们已经创建了两个具有标准类型 massage 的节点。现在，我们要学习如何使用 ROS 工具创建自定义消息。

首先，在 chapter2_tutorials 功能包下创建 msg 文件夹，并在其中创建一个新的文件 chapter2_msg1.msg。在文件中增加以下行：

```
int32 A
int32 B
int32 C
```

现在编辑 CMakeList.txt，从 # rosbuild_genmsg() 这一行中删除 #，然后使用 rosmake 命令编译功能包：

```
$ rosmake chapter2_tutorials
```

为了检查编译是否正确，可以使用 rosmsg 命令：

```
$ rosmsg show chapter2_tutorials/chapter2_msg1
```

如果你在 chapter2_msg1.msg 文件中看到一样的内容，说明编译正确。

现在创建一个 srv 文件。在 chapter2_tutorials 文件夹下创建一个名为 srv 的文件夹，并新建文件 chapter2_srv1.srv，在文件中增加以下行：

```
int32 A
int32 B
int32 C
---
int32 sum
```

编辑 CMakeList.txt 文件，从 #rosbuild_gensrv() 这一行中删除 #，并且使用 rosmake chapter2_tutorials 命令编译功能包。

你可以通过 rossrv 工具来检查编译是否正确：

```
$ rossrv show chapter2_tutorials/chapter2_srv1
```

如果你在 chapter2_srv1.srv 文件中看到相同的内容，说明编译正确。

2.4.12 使用新建的 srv 和 msg 文件

首先，我们将会学习如何创建一个服务并且在 ROS 中使用它。该服务将会对三个整数求和。我们需要两个节点，一个服务器一个客户端。

在 chapter2_tutorials 功能包中，新建两个节点并使用 example2_a.cpp 和 example2_b.cpp 为名称。别忘了要在 src 文件夹下创建这两个文件。

在第一个文件 example2_a.cpp 中，添加以下代码：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"
```

```
bool add(chapter2_tutorials::chapter2_srv1::Request  &req,
         chapter2_tutorials::chapter2_srv1::Response &res)
{
  res.sum = req.A + req.B + req.C;
  ROS_INFO("request: A=%ld, B=%ld C=%ld", (int)req.A, (int)req.B,
(int)req.C);
  ROS_INFO("sending back response: [%ld]", (int)res.sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_3_ints_server");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("add_3_ints", add);
  ROS_INFO("Ready to add 3 ints.");
  ros::spin();

  return 0;
}
```

还需要包含必要的头文件和我们创建的 srv 文件：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"
```

这个函数会对 3 个变量求和，并将计算结果发送给其他节点：

```
bool add(chapter2_tutorials::chapter2_srv1::Request  &req,
         chapter2_tutorials::chapter2_srv1::Response &res)
```

在这里，创建服务并在 ROS 中发布广播，如下：

```
ros::ServiceServer service = n.advertiseService("add_3_ints", add);
```

在第 2 个文件 example2_b.cpp 中，增加以下代码：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_3_ints_client");
  if (argc != 4)
  {
    ROS_INFO("usage: add_3_ints_client A B C ");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<chapter2_
tutorials::chapter2_srv1>("add_3_ints");
  chapter2_tutorials::chapter2_srv1 srv;
```

```
  srv.request.A = atoll(argv[1]);
  srv.request.B = atoll(argv[2]);
  srv.request.C = atoll(argv[3]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_3_ints");
    return 1;
  }

  return 0;
}
```

使用 add_3_ints 为名称创建一个服务的客户端：

```
ros::ServiceClient client = n.serviceClient<chapter2_
tutorials::chapter2_srv1>("add_3_ints");
```

下面创建 srv 文件的一个实例，并且加入需要发送的数据值。如果你还记得，这个消息需要 3 个字段：

```
chapter2_tutorials::chapter2_srv1 srv;
srv.request.A = atoll(argv[1]);
srv.request.B = atoll(argv[2]);
srv.request.C = atoll(argv[3]);
```

这些代码会调用服务并发送数据。如果调用成功，call() 函数会返回 true；如果没成功，call() 函数会返回 false：

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

为了编译节点，在 CMakeList.txt 文件中增加以下行：

```
rosbuild_add_executable(example2_a src/example2_a.cpp)
rosbuild_add_executable(example2_b src/example2_b.cpp)
```

现在执行以下命令：

```
$ rosmake chapter2_tutorials
```

为了启动节点，需要执行以下命令行：

```
$rosrun chapter2_tutorials example2_a
$rosrun chapter2_tutorials example2_b 1 2 3
```

并且你会看到如下显示：

```
Node example2_a
[ INFO] [1355256113.014539262]: Ready to add 3 ints.
[ INFO] [1355256115.792442091]: request: A=1, B=2 C=3
[ INFO] [1355256115.792607196]: sending back response: [6]
```

```
Node example2_b
[ INFO] [1355256115.794134975]: Sum: 6
```

现在将要用自定义的 msg 文件来创建节点。这个例子也一样，创建 example3_a.cpp 和 example3_b.cpp 文件，同时调用 chapter2_msg1.msg。

将下面的代码放在 example3_a.cpp 文件中：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_msg1.h"
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example1_a");
  ros::NodeHandle n;
  ros::Publisher pub = n.advertise<chapter2_tutorials::chapter2_
msg1>("message", 1000);
  ros::Rate loop_rate(10);
  while (ros::ok())
  {
    chapter2_tutorials::chapter2_msg1 msg;
    msg.A = 1;
    msg.B = 2;
    msg.C = 3;
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

将下面的代码放在 example3_b.cpp 文件中：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_msg1.h"

void messageCallback(const chapter2_tutorials::chapter2_
msg1::ConstPtr& msg)
{
  ROS_INFO("I heard: [%d] [%d] [%d]", msg->A, msg->B, msg->C);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example1_b");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("message", 1000, messageCallback);
  ros::spin();
  return 0;
}
```

如果现在运行这两个节点，将会看到如下信息：

```
...
[ INFO] [1355270835.920368620]: I heard: [1] [2] [3]
[ INFO] [1355270836.020326372]: I heard: [1] [2] [3]
[ INFO] [1355270836.120367449]: I heard: [1] [2] [3]
[ INFO] [1355270836.220266466]: I heard: [1] [2] [3]
...
```

2.5 本章小结

本章介绍了 ROS 系统的基本架构及其典型的工作方式。学习了一些基本概念、工具及如何同节点、主题和服务进行交互的示例。一开始，所有这些概念可能看起来有些复杂且不太实用，但在后面的章节中，你会逐渐理解如何使用这些概念所代表的对象。

各位读者最好在继续后续章节的学习之前，对这些概念及示例进行练习，因为在后面的章节里，我们将假定你已经熟悉所有的概念及其用途。

请注意如果想查询某个名词或功能的解释，且无法在这本书中找到相关内容或答案，那么可以通过以下链接访问 ROS 官方资源 http://www.ros.org。而且，你还可以通过访问 ROS 社区 http://answers.ros.org 提出自己的问题。

在下一章中，将学习如何使用 ROS 工具查询调试信息和实现数据的可视化。这些将帮助你理解 ROS 的运行状态，发现软件运行的问题，并且指导你对它的运行进行调整。

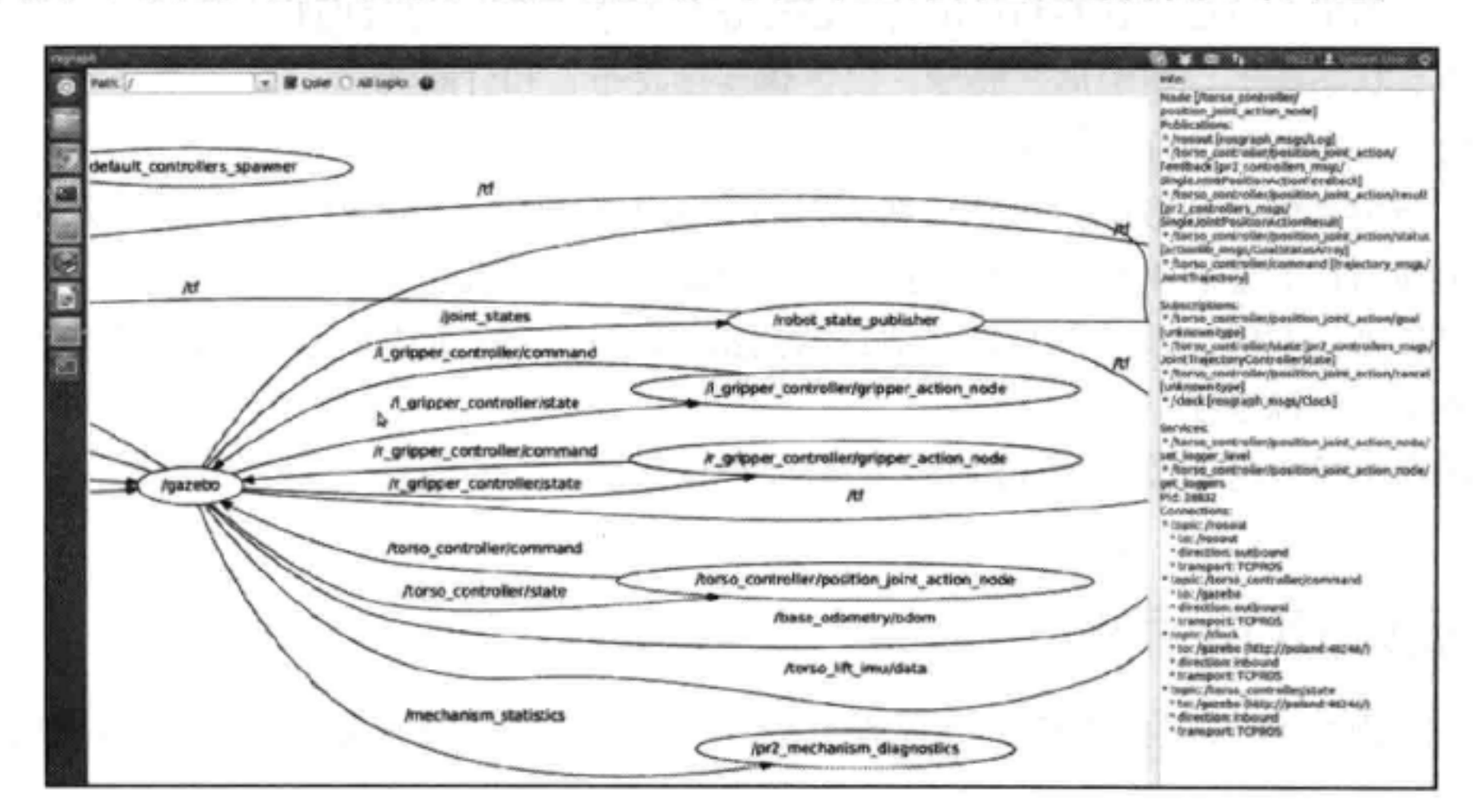
第 3 章
调试和可视化

ROS 软件框架附带了大量功能强大的工具，帮助用户和开发人员检测软硬件问题并完成代码调试。这包括了调试工具（如日志消息记录与展示工具）、数据可视化工具和系统监测组件。这些工具使得用户轻松地以自己喜欢的方式查看系统运行状态。

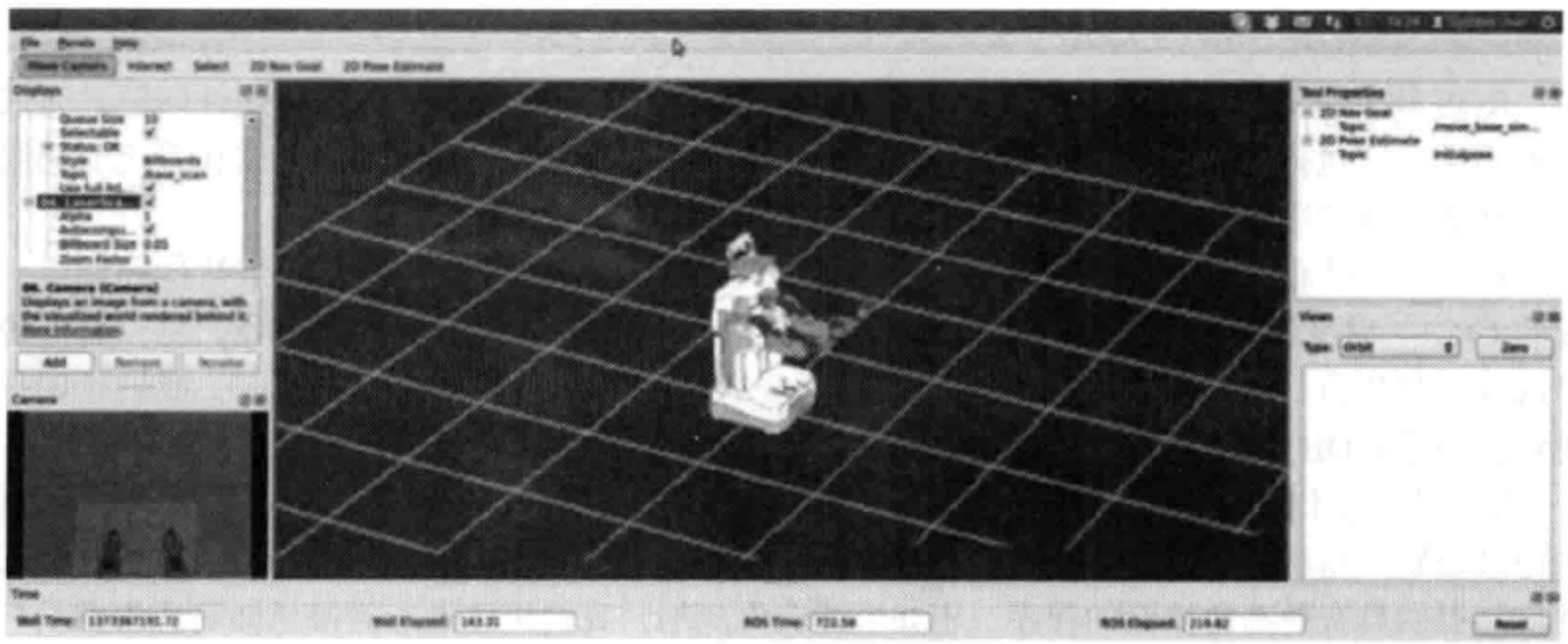
在这里，我们还会以 GDB 调试器为例解释调试 ROS 节点的工作流程。虽然这和调试一般的 C/C++ 程序没有什么不同，但还是需要说明几个必须要考虑到的地方。我们将只着重说明这几个特殊点，因为解释如何使用调试器已经超出了本章的范围。相对而言，我们希望你能够去阅读 GDB 的参考文档和用户手册。

ROS 提供用于日志记录的 API。在输出信息时，它允许根据信息的重要性来选择不同的日志记录级别。这样做的目的在于在发生故障时，能够提供更丰富内容帮助开发人员进行调试，并最终提供具有更高鲁棒性的程序。正如后面会看到的，我们可以通过高级通知消息告知用户算法运行信息，或警告用户出现了遗漏的值和参数，或记录无法恢复的故障和致命的错误。

然而，在程序编译和运行时仍然可能会失败。这时，可能发生两个关键的问题：首先，一个问题与节点、主题、服务或任何其他 ROS 组件有关；第二，我们的算法本身也可能导致问题。ROS 提供了一组功能强大的工具来监视系统状态。例如下面截屏中所示的节点状态图，在图中能够显示主题间（发布者和订阅者）的所有连接。无论是本地节点还是远程节点都被无缝部署，以便用户能够轻松、迅速地检测如节点未运行或主题连接缺失等问题。



在某种程度上，ROS 提供了一系列通用绘图工具以分析自定义算法的输出或结果，以便更容易地检测 bug。首先，有针对标量值的时序绘图工具，这常用于节点间传递的消息字段的图形化展示；其次，有支持双目立体视觉的图像展示工具；再次，有如 rviz 等 3D 可视化工具（如下面截屏所示，图中为 PR2 机器人），这些工具能够呈现点云、激光扫描等 3D 数据。这些是很特殊的数据格式，数据相关于特定的传感器坐标空间，并通过坐标来呈现数据。一个机器人一般由多个坐标空间组成，坐标空间之间能够进行变换。为了帮助用户查看它们之间的关系，也有查看坐标空间层次结构的工具。



在本章中，将介绍以下内容：

- 创建 ROS 节点时，调试和优化代码。
- 为代码添加消息日志并设置不同的级别，包括调试消息、错误、致命错误。
- 日志命名、应用特定条件、消息日志条件显示，这些在大型项目中非常有用。
- 介绍一个管理所有消息的图形化工具。
- 通过列出节点运行状态和可用的主题与服务来监视 ROS 系统的状态。
- 节点状态图形以图形形式展示，以主题连接发布者和订阅者。
- 绘制特定消息的标量数据。
- 复杂类型的标量数据可视化。特别是将介绍图像显示、FireWire 接口摄像头的实例和多种消息类型的 3D 可视化。这些都在 ROS 中实现了无缝支持。
- 解释坐标空间以及它们与特定主题的消息之间的关系。同样将会看到在坐标变换树中的空间坐标变换。
- 保存主题发送的消息和如何为了仿真或测试重播已发的消息。

3.1 调试 ROS 节点

为了检测 ROS 节点内部算法执行的问题，可以从不同层面面对问题，使软件调试更加

容易。首先，必须针对算法、驱动或其他软件代码提供一些可读的关于计算进度和状态的信息。为达到这个目的，ROS 提供了一套覆盖整个系统的日志记录宏。其次，可以根据需要配置不同的日志级别，这需要使用日志工具来确定将哪个节点的相关日志配置成哪个具体级别。在 ROS 的示例中，还将看到如何针对特定消息的条件和名称设置调试 / 日志记录级别。再次，必须能够使用调试器单步执行源代码。我们将看到广为人知的 GDB 调试器与 ROS 节点无缝集成。最后，通过查看 ROS 节点和主题的抽象语义，有效地监视整个系统的当前状态。这种 ROS 系统的内部检查通过绘制主题间的节点连接图工具实现。用户或开发人员可以一目了然地发现某个连接断开，这些会在后续章节中介绍。

3.1.1 使用 GDB 调试器调试 ROS 节点

从调试任意 C/C++ 可执行文件的标准方法开始。ROS 的灵活性允许使用众所周知的 GDB 调试器对常规的 C/C++ 程序进行调试，唯一要知道的是可执行文件的路径，这在 ROS 系统里通常是用 C++ 程序编写的某个节点的位置。因此，我们只需要将当前路径移动到包含二进制文件的节点目录下并在 GDB 中运行这些代码。实际上，还可以直接运行节点而无需使用标准的 `rosrun <package><node>` 的语法。

简单地说，我们将向你展示如何在 GDB 中运行 `chapter3_tutorials` 功能包中的 `example1` 节点。首先，使用 `roscd` 命令移动到功能包路径下：

```
roscd chapter3_tutorials
```

然后，要记得在功能包文件夹下的 bin 文件夹中创建 C++ 程序文件。因此，只需在 GDB 里面运行以下命令：

```
gdb bin/example1
```

请记住在启动节点之前请务必调用 `roscore` 命令，因为 ROS 需要启动节点管理器及其他核心服务进程。

一旦 `roscore` 已经运行，你就可以通过点击 R 键和 Enter 键从 GDB 中启动节点；也可以用 L 键列出相关源代码，以及设置断点或使用任何 GDB 附带的功能。如果一切工作正常，在调试器内运行节点后就能在 GDB 终端看到下面的输出：

```
(gdb) r
Starting program: /home/enrique/dev/rosbook/chapter3_tutorials/bin/
example1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_
db.so.1".
[New Thread 0x7ffff2664700 (LWP 3204)]
...
```

```
...
[Thread 0x7ffff1e63700 (LWP 3205) exited]
[Inferior 1 (process 3200) exited normally]
```

3.1.2 ROS 节点启动时调用 GDB 调试器

如果在启动节点时使用启动文件来完成的话，就可以通过使用 XML 语法修改启动文件中的节点属性，在节点启动时调用 GDB 调试器。以前文示例中的节点为例，即 chapter3_tutorials 功能包中的 example1，添加以下节点信息到 launch 文件：

```
<launch>
<node pkg="chapter3_tutorials" type="example1"
            name="example1"/>
</launch>
```

请注意在上面的配置中，功能包名传递给 pkg 属性，节点名传递给 type 属性。由于同一节点可以运行多个实例，因此需要给这个节点的实例起一个名称。在这种情况下，我们给出了与节点类型相同的名称，也就是把 name 属性赋值为 example1。同时，最好将这些属性配置输出到屏幕，这样就会在与启动节点相同的 shell 窗口中出现相应调试信息，正如下面的代码所示：

```
<node pkg="chapter3_tutorials" type="example1"
            name="example1" output="screen"/>
```

想要在节点启动时调用 GDB 调试器，需要添加 launch-prefix="xterm-egdb--args"：

```
<launch>
<node pkg="chapter3_tutorials" type="example1"
            name="example1" output="screen"
            launch-prefix="xterm -e gdb --args"/>
</launch>
```

这个语句的功能很好理解。它会在加载节点的同时启动 GDB 调试器，一直等待用户按 C 键或 R 键，也就是说加载节点并等待运行。这种方式，用户可以在节点运行之前设置断点或与一般 GDB 会话一样执行各种功能。此外，请注意这时会打开一个新窗口。因为是在 xterm 上创建 GDB 会话，所以区别于程序输出窗口，还会有一个调试窗口。

此外，我们可以使用相同的属性在节点启动时调用其他诊断工具。例如，可以在程序上启动 valgrind 来检测内存泄漏，并执行性能分析。你可以访问 http://valgrind.org 获取有关 valgrind 的详细信息。我们可以使用和 GDB 类似的方式把节点附加上去。这时不需要额外的窗口，所以不启动 xterm，只需设置 valgrind 为启动前缀：

```
<launch>
<node pkg="chapter3_tutorials" type="example1"
            name="example1" output="screen"
            launch-prefix="valgrind"/>
</launch>
```

3.1.3 设置 ROS 节点 core 文件转存

虽然 ROS 节点实际上就是一般的可执行文件，但在设置 GDB 的 core 内存文件转存（core dump）时仍有一些棘手的问题需要注意。首先，要取消 core 文件大小限制。请注意这适用于任何可执行文件，不只是 ROS 节点：

```
ulimit -c unlimited
```

然后，为了能够创建 core 文件转存，必须将 core 文件名设置为默认使用的进程 pid，否则，它们会无法创建，因为在 $ ROS_HOME 已有一个 core 目录会防止 core 文件转存。因此，为了创建 core 文件转存的名称和路径 $ ROS_ HOME/core.PID，必须按照如下命令：

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

3.2 调试信息

在调试时，通过信息记录显示程序的运行状态是非常重要的，但需要确定这样做不会影响软件的运行效率，不会和正常输出混淆。在 ROS 中有满足以上要求并且内置于 log4cxx（众所周知的 log4j 记录库的一个端口）之上的 API。简单地说，我们有不同层级的调试信息输出，每条信息都有自己的名称，并根据相应条件输出消息，它们对性能没有任何影响并与 ROS 框架中的其他工具完全集成。此外，它们无缝集成在并发执行的节点上，也就是说信息没有被打散，而是完全可以根据它们的时间戳进行交叉展示。在下面的小节中，我们将进行详细的解释以及如何充分使用它们。

3.2.1 输出调试信息

ROS 自带了大量的能够输出调试信息的函数和宏。这些信息包括错误、警告或简单的提示信息。它提供了如信息（或日志）级别、条件触发消息和 STL 的流接口等诸多方式。更加直截了当地说，如果想要输出任何提示信息，你只需要在代码中的任意位置插入以下代码：

```
ROS_INFO( "My INFO message." );
```

注意，只要主要的 ROS 头文件被包括在内，那么在源代码中就不需要包括任何特定的库。但我们可以添加 ros/console.h 头文件，如下面的代码所示：

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

作为前面的消息处理程序运行的结果，将得到如下输出：

```
[ INFO] [1356440230.837067170]: My INFO message.
```

所有输出的调试信息都附带其级别和当前时间戳（因为这个原因你的输出可能有所不

同)，这两个值放在实际计算值之前的方括号中。时间戳以公历时间计时，代表着自1970年以来的秒和纳秒计数，时间戳之后是具体的信息。

此函数允许以C语言中的printf函数的方式增加参数，这意味着可以用printf格式为特殊字符传递数值。例如，可以按照下面代码输出变量val对应的浮点数值：

```
const double val = 3.14;
ROS_INFO( "My INFO message with argument: %f", val );
```

此外，C++STL流支持*_STREAM函数。因此，前面的指令相当于下面使用流：

```
ROS_INFO_STREAM(
  "My INFO stream message with argument: " << val
);
```

请注意，我们没有指定任何流，因为它一般默认为cout或cerr。调试信息级别会决定具体是哪一个。我们将在下一节中做详细介绍。

3.2.2 设置调试信息级别

ROS有五个标准级别，按照顺序排列分别是DEBUG、INFO、WARN、ERROR和FATAL。这些名称是输出信息的函数或宏的一部分，它们遵循以下语法：

```
ROS_<LEVEL>[_<OTHER>]
```

DEBUG和INFO信息是输出到标准输出cout（或stdout）；而WARN、ERROR和FATAL会输出到错误信息输出cerr（或stderr）。同时，只要显示终端能够支持，那么每一种信息都会以特定的颜色输出在屏幕上。这些颜色分别是：DEBUG为绿色，INFO为白色，WARN为黄色，ERROR为红色，以及FATAL为紫色。

这些信息的名称清楚地代表了信息的类型。因此，用户只需要相应地使用它们。正如我们将在后续章节看到的，我们只在用户需要的最小范围内输出消息。在代码稳定之后，就能够按条件显示调试信息。此外，我们会在后续章节中解释OTHER变量。

3.2.3 为特定节点配置调试信息级别

默认情况下，系统会显示INFO及更高级别的调试信息，并使用ROS默认级别来过滤特定节点所需输出的信息。要实现这一功能有很多方法。其中有些是在编译时设定，而其他的则可以在执行前使用配置文件进行更改。另外，也可以动态地改变级别。下面将介绍使用rosconsole和rxconsole来实现这一功能。

首先，将看到在编译时如何设置源代码的调试级别。只需要找到main函数，在ros::init代码之后，插入以下代码：

```
// Set the logging level manually to DEBUG
ROSCONSOLE_AUTOINIT;
log4cxx::LoggerPtr my_logger =
  log4cxx::Logger::getLogger( ROSCONSOLE_DEFAULT_NAME );
```

```
my_logger->setLevel(
  ros::console::g_level_lookup[ros::console::levels::Debug]
);
```

你不需要包含任何特殊的头文件，但在 CMakeLists.txt 文件中必须将头文件链接到 log4cxx 库。要做到这一点，必须输入：

```
find_library(LOG4CXX_LIBRARY log4cxx)
```

同时也要把我们的节点链接过去：

```
rosbuild_add_executable(example1 src/example1.cpp)
target_link_libraries(example1 ${LOG4CXX_LIBRARY})
```

现在，当节点运行时，DEBUG 及更高级别的信息才会显示，因为在前面的示例中设置了 ros: : console: : levels: : Debug。你可以运行 example1 节点来检查它是否改变了级别。

另一种替代前面的方法包括使用编译时宏记录去除器。请注意，这种方法会在编译时移除低于特定级别的所有调试信息，而在此之后再也无法恢复这些信息。这是我们在发布程序时经常使用的。我们必须设置 ROSCONSOLE_MIN_SEVERITY 宏来实现，可以将其设为最低严重性级别才显示或者设置不显示，这样任何调试信息（包括 FATAL 级别）都会被抑制输出。这个宏可以在源代码中或者在 CMakeLists.txt 文件中进行设置。这种方法可能用到的宏包括 ROSCONSOLE_SEVERITY_DEBUG、ROSCONSOLE_SEVERITY_INFO、ROSCONSOLE_SEVERITY_WARN、ROSCONSOLE_SEVERITY_ERROR、ROSCONSOLE_SEVERITY_FATAL 和 ROSCONSOLE_SEVERITY_NONE。

ROSCONSOLE_MIN_SEVERITY 宏在 ros/console.h 中默认定义为 DEBUG 调试。于是可将它作为一个编译参数（使用 -D）传递或把它放在头文件前。例如，若要仅显示 ERROR 或更高级别的调试信息，在 example2 这个节点中将执行以下操作：

```
#define ROSCONSOLE_MIN_SEVERITY ROSCONSOLE_SEVERITY_DEBUG
```

除此之外，还有一个更灵活的方法就是在配置文件中设置到最低调试级别。为了方便，创建一个名为的 config 文件夹和名为 chapter3_tutorials.config 的文件，文件内容如下：

```
log4j.logger.ros.chapter3_tutorials=DEBUG
```

我们可以设置成 ROS 所支持的任何级别。然后，必须设置 ROSCONSOLE_CONFIG_FILE 环境变量指向我们的文件。由于环境变量在指向某个特定文件后，如果我们需要使其他文件也支持这个功能，就要不断修改环境变量。因此，还有更好的选择。我们可以使用一个 launch 文件来代替配置环境变量，这样做还会直接运行我们的节点。因此，我们可以通过 env 字段扩展 launch 文件，如下面的代码所示：

```
<launch>
  <!-- Logger config -->
  <env name="ROSCONSOLE_CONFIG_FILE"
       value="$(find chapter3_tutorials)/config/chapter3_tutorials.
config"/>

  <!-- Example 1 -->
  <node pkg="chapter3_tutorials" type="example1" name="example1"
        output="screen"/>
</launch>
```

如前面所述，环境变量会找到 config 文件，其中包含每个已命名日志的级别说明。然后我们的节点就可以通过 launch 文件运行。

3.2.4 信息命名

因为可以在同一节点的很多位置放置信息，ROS 允许为程序中的每一条信息取名。这样的话我们就会更加容易地知道每一条信息是来自于哪一段代码。为了实现这个功能，我们使用 <LEVEL>[_STREAM]_NAMED 函数，如以下代码所示（以 example2 节点为例）：

```
ROS_INFO_STREAM_NAMED(
  "named_msg",
  "My named INFO stream message; val = " << val
);
```

这样，我们就可以在 config 文件中为每个带有名称的信息设置不同调试级别。这允许在节点定义文件中使用信息的名称对信息进行配置。例如可以设定 named_msg 的信息只在 ERROR 及更高级别中显示，（请注意尽管 ROS 使用 log4cxx，但其配置文件使用 log4j 根目录名称）如下列代码所示：

```
log4j.logger.ros.chapter3_tutorials.named_msg=ERROR
```

3.2.5 条件显示信息与过滤信息

条件显示（conditional）信息是指仅当满足给定的条件时才输出的信息。在某些情况下，这就像是调试信息的条件断点。

我们需要使用 ROS_<LEVEL>[_STREAM]_COND[_NAMED] 函数来调用它们，请注意它们也可以是命名的信息。以下是以 example2 节点为例的代码：

```
// Conditional messages:
ROS_INFO_STREAM_COND(
  val < 0.,
  "My conditional INFO stream message; val (" << val << ") < 0"
);
ROS_INFO_STREAM_COND(
  val >= 0.,
  "My conditional INFO stream message; val (" << val << ") >= 0"
);
```

```
ROS_INFO_STREAM_COND(
  "cond_named_msg", val >= 0.,
  "My conditional INFO stream message; val (" << val << ") >= 0"
);
```

过滤（Filtered）信息在本质上与条件显示信息类似，但它允许我们指定一个用户自定义的过滤器。这个自定义过滤器继承自 ros: : console: : FilterBase 结构体。

我们必须将过滤器作为指针传递给以 ROS_<LEVEL>[_STREAM]_FILTER [_NAMED] 为格式的宏的第一个参数。下面的示例来自于 example2 节点：

```
struct MyLowerFilter : public ros::console::FilterBase {
  MyLowerFilter( const double& val ) : value( val ) {}

  inline virtual bool isEnabled()
  {
    return value < 0.;
  }

  double value;
};

struct MyGreaterEqualFilter : public ros::console::FilterBase {
  MyGreaterEqualFilter( const double& val ) : value( val ) {}

  inline virtual bool isEnabled()
  {
    return value >= 0.;
  }

  double value;
};

MyLowerFilter filter_lower( val );
MyGreaterEqualFilter filter_greater_equal( val );

ROS_INFO_STREAM_FILTER(
  &filter_lower,
  "My filter INFO stream message; val (" << val << ") < 0"

);
ROS_INFO_STREAM_FILTER(
  &filter_greater_equal,
  "My filter INFO stream message; val (" << val << ") >= 0"
);
```

3.2.6 信息的更多功能——单次显示、可调、组合

你可以控制信息的显示次数。通过使用 ROS_<LEVEL>[_STREAM]_ONCE[_NAMED]，可以让信息只输出一次。这个功能在循环中非常有用，因为考虑到效率的关系我们不希望有过多输出，但同时又能知道程序进入了该循环运算。

```
for( int i = 0; i < 10; ++i ) {
  ROS_INFO_STREAM_ONCE(
    "My once INFO stream message; i = " << i
  );
}
```

这段代码也是来自 example2 节点，信息只会在 i==0 时显示 1 次。

然而，有时候在迭代中每次都显示信息也是有必要的。这就需要可调信息。它们和前面单次显示的消息格式是一样的，但是你需要将函数名中的 ONCE 替换成 THROTTLE，函数会将可调时间（Duration）作为第一个参数，也就是说，它将会只在特殊的时间间隔后输出：

```
for( int i = 0; i < 10; ++i ) {
  ROS_INFO_STREAM_ONCE(
    2,
    "My once INFO stream message; i = " << i
  );
  ros::Duration( 1 ).sleep();
}
```

最后要说明的是，无论对于哪个级别的信息，命名信息、条件显示信息、单次 / 可调信息等都能够组合起来使用。

动态加载节点（Nodelet）对于日志和调试信息也提供了一定的支持。因为它们有自己的命名空间，而且有各自唯一的名称，这样才能够让一个动态加载节点的提示信息与其他节点的区别开。简单地说，前面提到的所有宏对于动态加载节点都是可用的，只是宏的名称上，需要将 ROS_* 开头替换成 NODELET_*。这些宏将只能够在动态加载节点内部编译。同时，它们会使用动态加载节点运行时的名称设置一个命名的日志。这样你就能够区分同一个动态加载节点管理器下运行的两个相同类型动态加载节点的输出。动态加载节点的另外一个优势是它们能够帮助你将某个动态加载节点转换到 DEBUG 级别，而不是把整个特定类型的动态加载节点都转换过去。

3.2.7 使用 rosconsole 和 rxconsole 在运行时修改调试级别

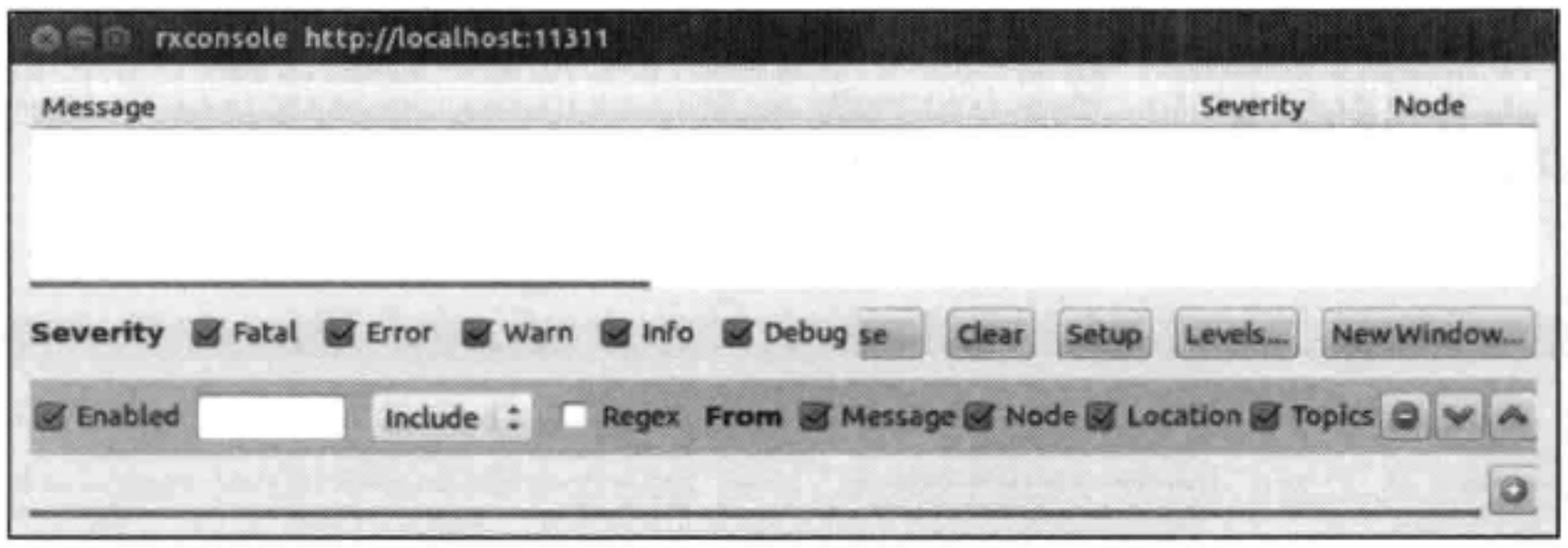
日志信息集成了一系列的窗口化显示及配置工具。ROS 框架附带了名为 rosconsole 的 API。其实，在某种程度上我们已经在前面的章节中使用了该 API。如果你想使用该 API 的高级特性，那么你可能需要通过其他途径查询相关资料，但我们认为本书已经涵盖了机器人应用及开发人员必须要了解的基本知识。

除了能在你的节点中配置日志记录信息的 API 之外，ROS 还在 rxtools 功能包下提供一种图形化工具。这个工具是 rxconsole，你只需要将其敲入命令行，然后就会看到一个图形界面，允许你查看、监视和配置所有正在运行的节点日志记录子系统。

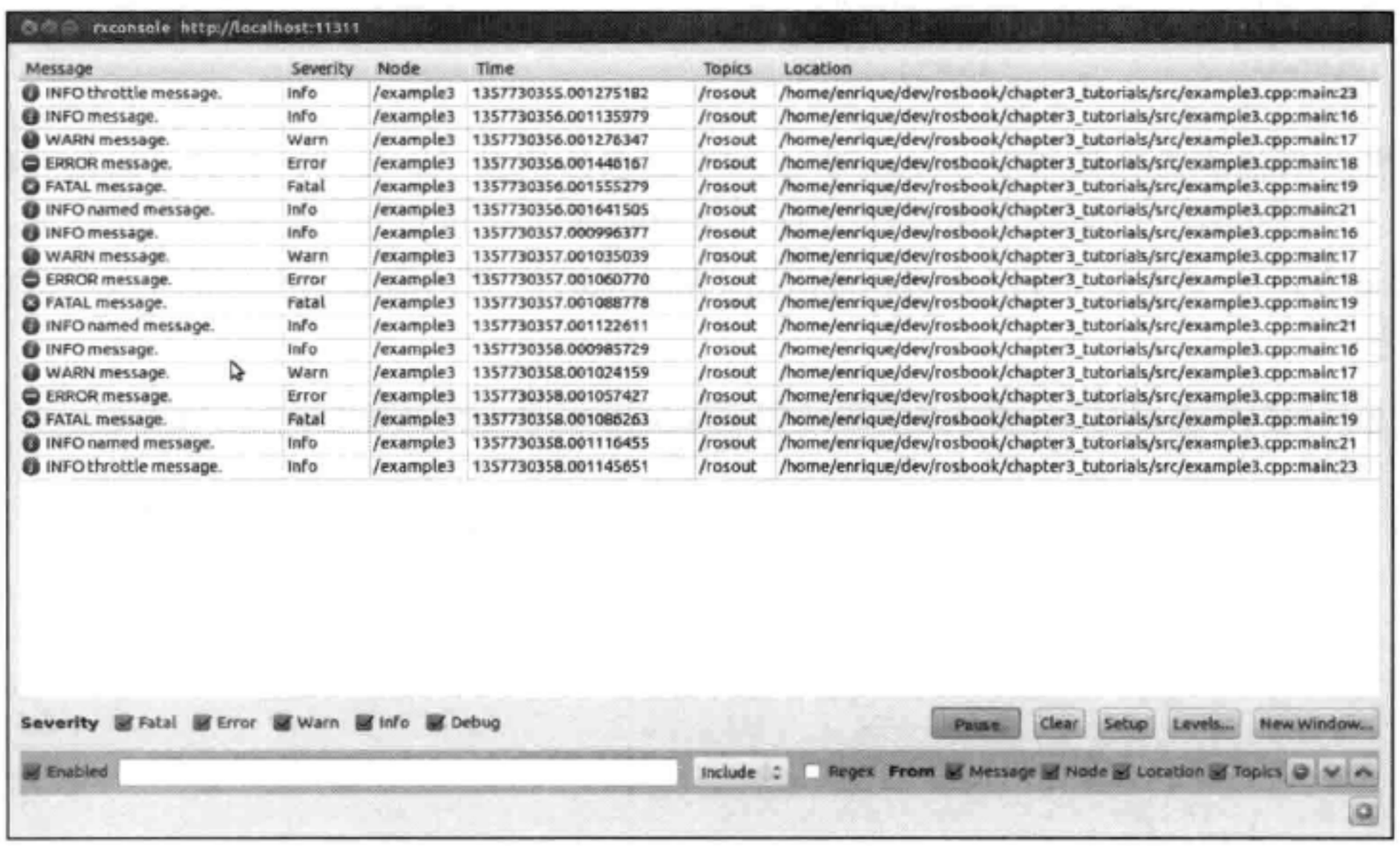
为了测试这个功能，将使用代码示例 example3，在一个命令行窗口中运行 roscore，在另一个终端窗口中运行节点：

```
rosrun chapter3_tutorials example3
```

在运行这个节点的同时，在另外一个终端窗口中运行 rxconsole。这时会打开下面的窗口。请注意，你也可以先运行 rxconsole，然后再运行节点。这样就会有日志信息出现如以下截屏所示：

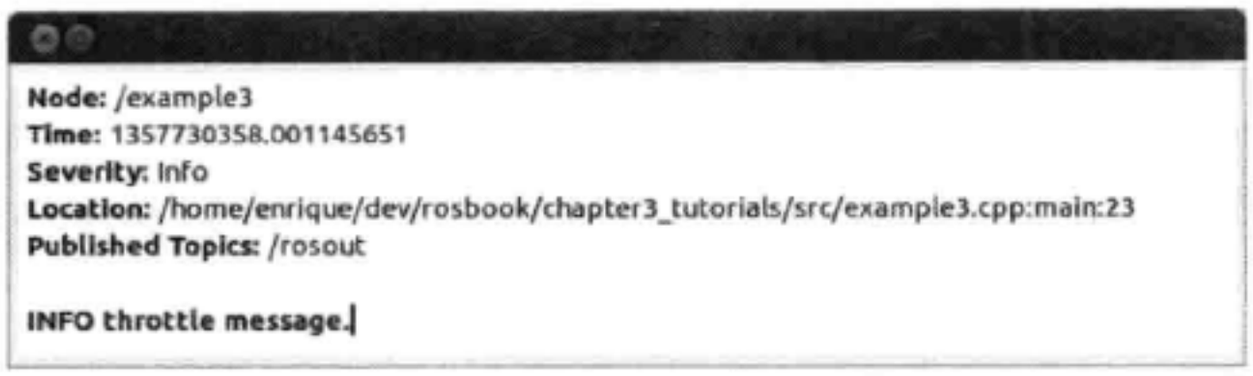


一旦 example3 节点开始运行，就会看到如下图所示的信息：

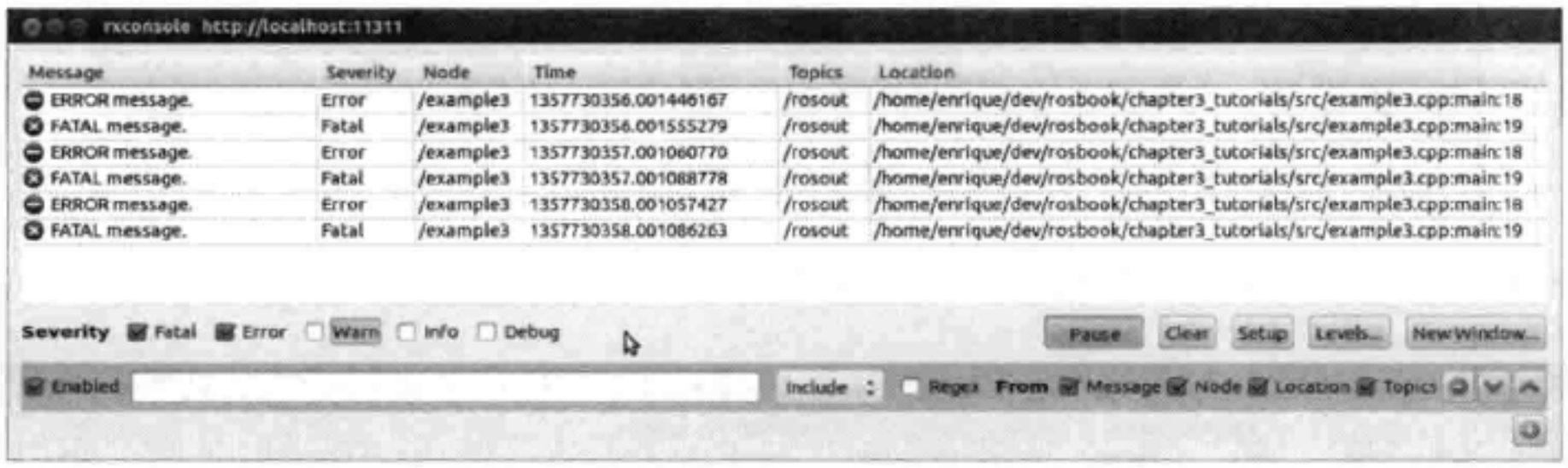


在上表中，是关于给定信息的几列内容（如图所示顺序排列）。这些列包括信息本身、严重级别、产生该信息的节点、时间戳、主题（通常是 /rosout 聚合器聚合了来自 ROS 服务器的各类信息）以及产生日志的宏所在的节点代码位置。

我们可以通过点击 Pause 按钮来停止接收新信息，再次点击就可恢复监听。对于这个表中的每一条信息，我们都可以通过点击 Pause 来查看它的细节，如下图所示，正是上一个截屏的最后一条信息。



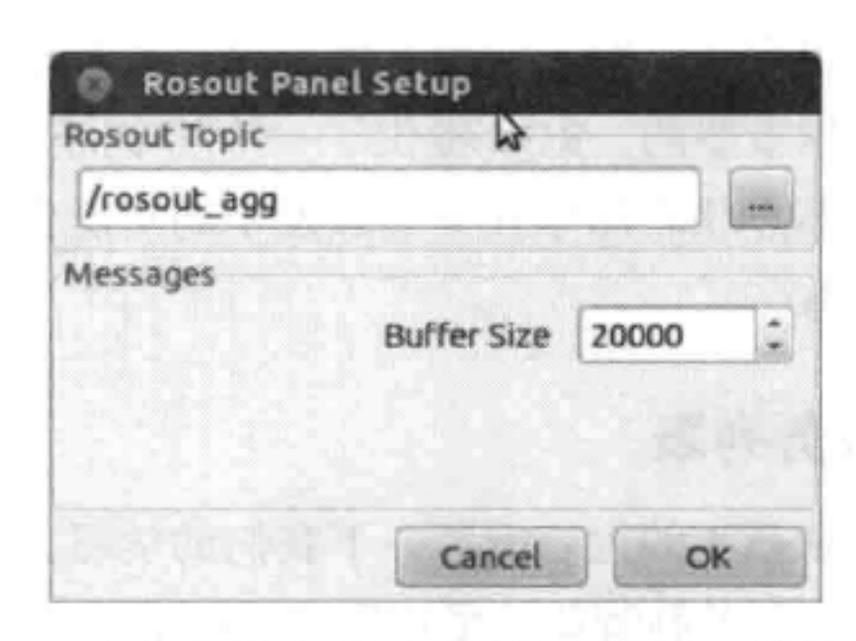
还有一个非常好的特性，就是 rxconsole 能够过滤信息。最直接的过滤方法是设定严重级别。如果你只想看 FATAL 和 ERROR 级别的信息，必须取消勾选其他级别，如下截屏所示：



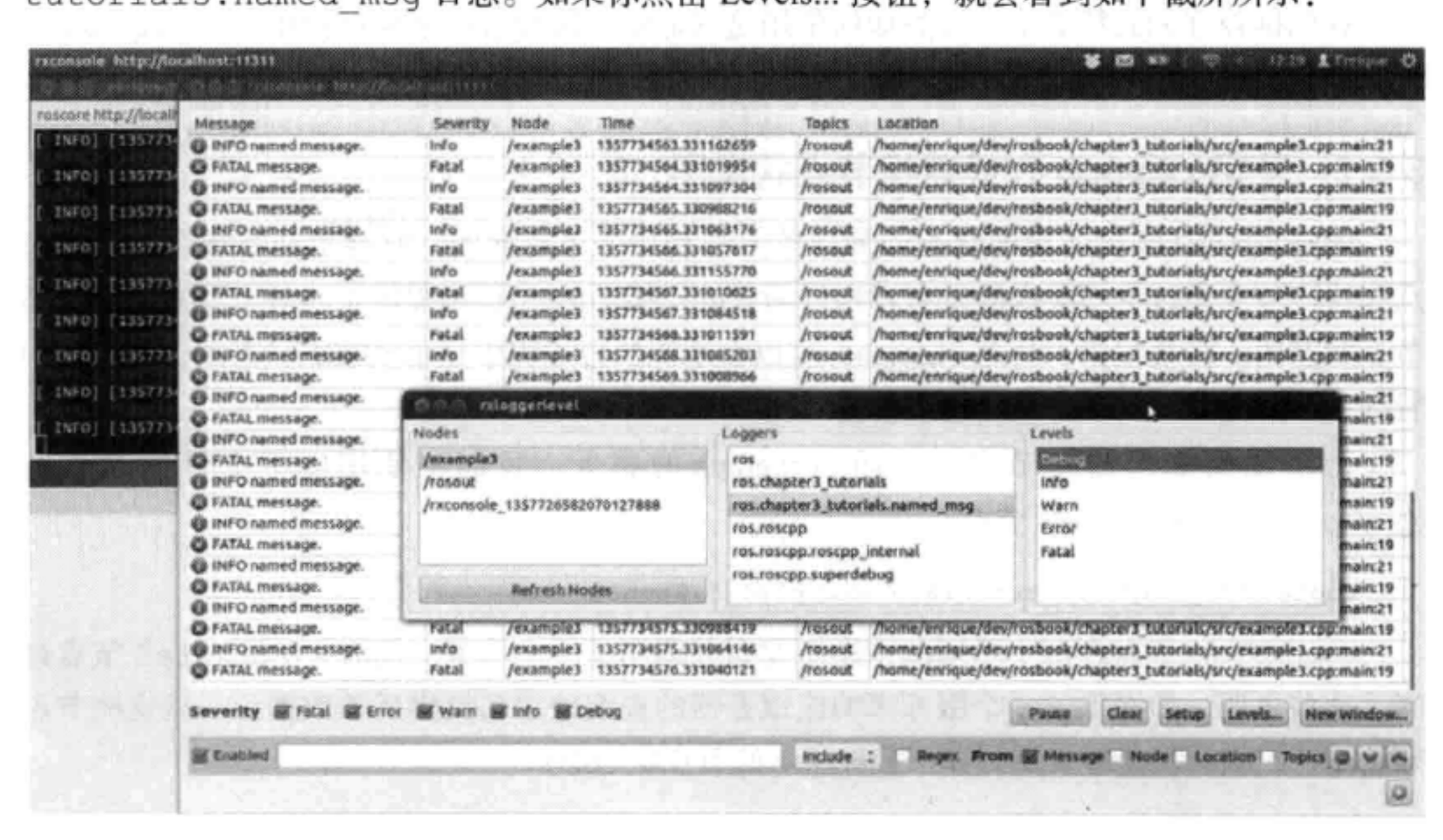
你立刻就能看到只剩下 FATAL 和 ERROR 级别的信息了。另外，我们可以按照信息内容、节点名称、位置、主题名称及查询关键词等方式进行过滤（选择 include 方式或 exclude 方式）。下图显示了我们选择在信息（Message）一栏内筛选包含关键词 named 的所有信息：

请注意，我们可以添加足够多的过滤器，也可以直接移除或取消消息。

我们同样可以通过 rxconsole 移除所有捕捉的信息，只需要点击 Clear 按钮。Setup 按钮允许我们配置诊断信息聚合器（diagnostic_aggregator）主题。聚合器主题主要是 `rosout_agg`，还有一定数量由 GUI 保存的滚屏前屏幕内的历史数据（如下图所示）。请注意，诊断信息聚合器仅仅是 ROS 服务器中接收所有日志信息的一个接收器。只有这样，我们才能够监视哪个设备失效了或者设备如何工作。高级的开发者会发现这些工具对于了解诊断 API 是很有用的。这不仅能够支持在更高层级上建立继承关系，而且对于复杂系统或机器人来说非常有帮助。



最后，我们可以根据日志的名称设定日志的严重级别。默认情况下，每个节点都有一个使用功能包名称的日志，但我们还可以使用前面章节介绍过的 NAMED 宏，来定义日志信息的名称。对于 `example3` 节点来说，有 `ros.chapter3_tutorials` 和 `ros.chapter3_tutorials.named_msg` 日志。如果你点击 Levels... 按钮，就会看到如下截屏所示：



这里我们选择使用 example3 节点，然后选择相应的日志信息，并设定其级别。请注意，这里还有一些内部日志，你需要把这些日志和前面提到的日志区分开。在前面的截屏中，我们为 ros.chapter3_tutorials.named_msg 日志选择了 DEBUG 级别，这样只有该级别或更高级别的信息才能被显示。也就是说，所有的信息都符合这一设定。

3.3 监视系统状态

当系统运行时，可能有数个节点和数十个主题在发布消息，通过订阅与其他节点连接。同时，有些节点可能也会提供服务。对于大型系统来说，通过提供一些工具让我们看到系统在给定时间的运行状态是非常重要的。ROS 提供了一些基本而又非常强大的工具，它们不仅能实现状态监视，还能探测节点状态图中任何节点发生的功能失效。简而言之，使用主题来连接节点展现系统架构的状态。

3.3.1 节点、主题与服务列表

坦白地讲，我们应该先从基本的内容回顾一下我们的学习。我们将学习如何获得正在运行节点、主题、当前所有可用服务的清单。尽管这非常简单，却是最有效和最常用的。

- 获得运行中节点的清单：**rosnode list**
- 所有节点的主题请使用：**rostopic list**
- 类似地，显示所有服务使用：**rosservice list**

我们推荐你回到第 2 章，熟悉如何使用这些命令，同时看一下如何使用 rosmsg show 获取特定主题所发出的消息类型。

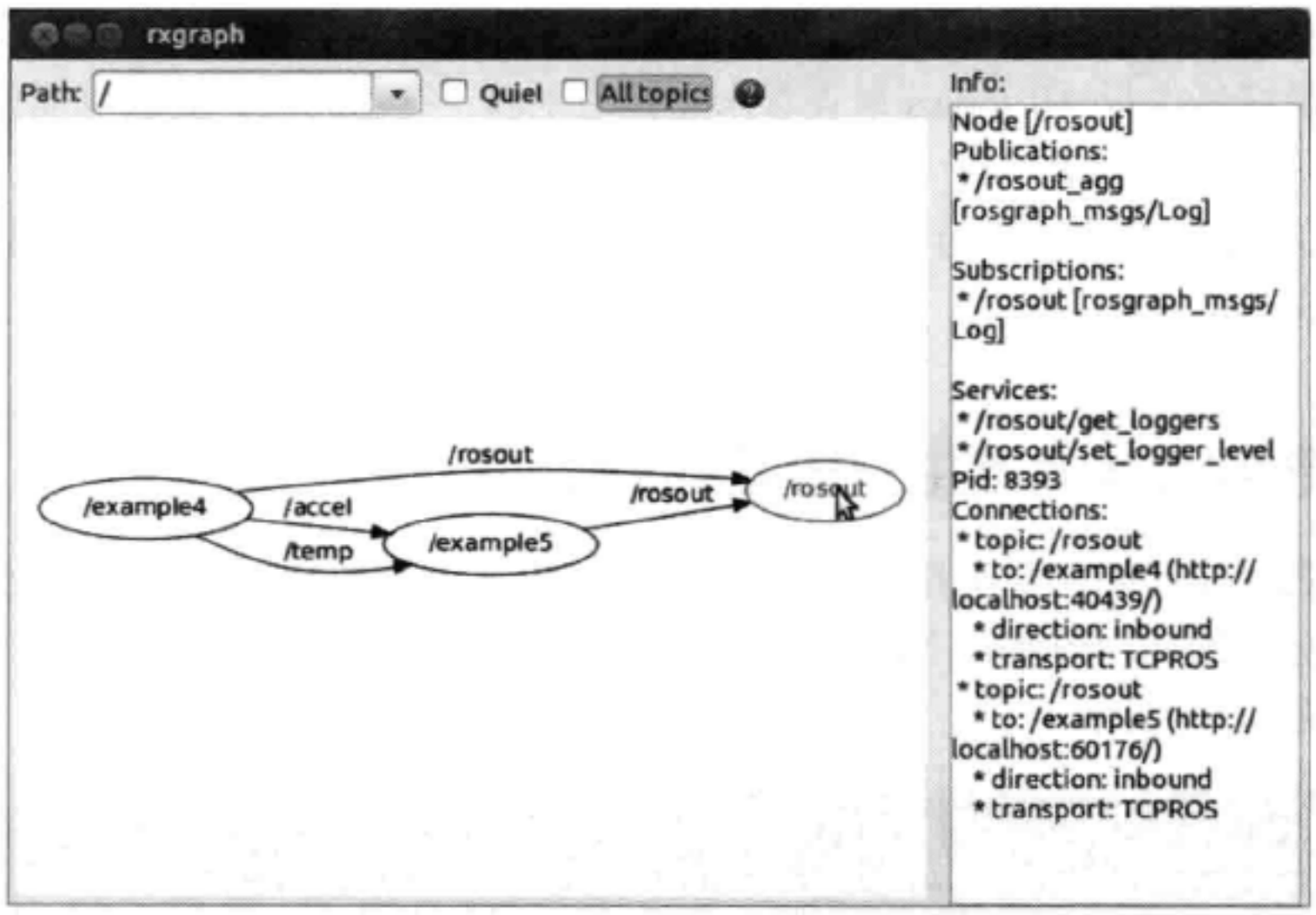
3.3.2 使用 rxgraph 在线监视节点状态图

能够说明 ROS 会话当前状态的最简单方法是用一个有向图，它显示在系统中运行的节点和这些节点通过主题实现的发布者到订阅者的连接。ROS 框架提供了一些工具来生成此类节点状态图。其最主要工具是 rxgraph，可以在系统执行过程中显示该节点状态图，并使我们看到一个节点是如何动态地出现或消失。

为了说明如何监视节点、主题和 rxgraph 的服务，我们将使用以下文件同时运行 example4 和 example5 节点：

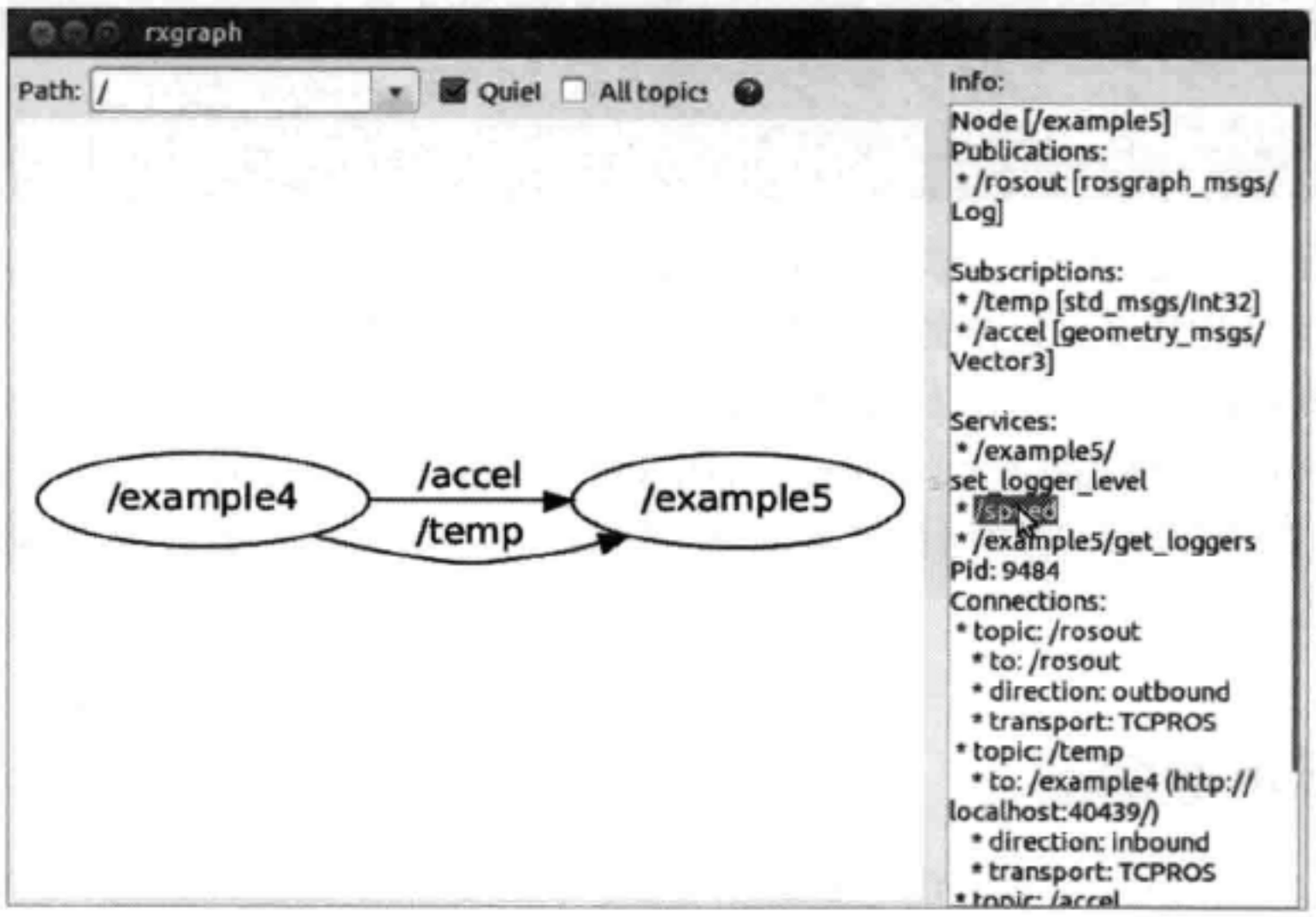
```
roslaunch chapter3_tutorials example4_5.launch
```

example4 节点在两个不同的主题中发布并调用了一个服务。同时，example5 节点订阅了这些主题，并提供了一个服务来响应服务器的查询请求和提供反馈数据。一旦这些节点开始运行，我们就能够查询到如下截屏所示的节点状态图：



在上面的截屏中，我们看到节点通过主题相连，rosout 节点可以理解为 ROS 服务器。还有如我们之前所学的，rosout 主题会向服务器中的诊断聚合器发布日志信息。在右侧的面板上有选定节点相关的信息。我们还能看到服务器相关的信息，包括其 IP 地址、远程节点连接端口、主题和连接。

我们可以启用简洁（quiet）视图，这就会省略 ROS 服务器。这对于大型的系统是非常有用的，因为 ROS 服务器总是存在，却在我们的系统拓扑图上不会提供任何有用的信息。要查看节点的服务，我们先打开节点再查看右侧面板。在下面的截屏中，我们选中了 `example5` 节点的 speed 服务：



当系统发生问题时，节点会一直显示为红色（而不仅仅在我们鼠标悬停在其上时）。在这种情况下，最好是选择 All topics 来查看未连接的主题。有些时候，问题可能仅仅是一个主题名称的拼写错误造成的。

当节点在不同的机器上运行时，rxgraph 会展示其高级别调试能力，因为它能够显示不同机器的两个节点间是否连接。

3.4 当奇怪的事情发生——使用 roswtf

ROS 还提供了另外一个工具来检测给定功能包中所有组件的潜在问题。使用 roscd 移动到你想要分析的功能包路径下，然后运行 roswtf。在 chapter3_tutorials 示例中，我们可以获得以下输出：

```
user@pc$ roswtf
Loaded plugin tf.tfwtf
Package: chapter3_tutorials
================================================================================
Static checks summary:

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

WARNING ROS_IP may be incorrect: ROS_IP [localhost] does not appear to be a local IP address ['127.0.0.1', '192.168.1.35'].

Found 1 error(s).

ERROR The following packages have rpath issues in manifest.xml:
 * chapter3_tutorials: found flag "-L/opt/ros/fuerte/lib", but no matching "-Wl,-rpath,/opt/ros/fuerte/lib"
 * geometry_msgs: found flag "-L/opt/ros/fuerte/lib", but no matching "-Wl,-rpath,/opt/ros/fuerte/lib"
 * std_msgs: found flag "-L/opt/ros/fuerte/lib", but no matching "-Wl,-rpath,/opt/ros/fuerte/lib"
 * roscpp: found flag "-L/opt/ros/fuerte/lib", but no matching "-Wl,-rpath,/opt/ros/fuerte/lib"

================================================================================
Beginning tests of your ROS graph. These may take awhile...
analyzing graph...
... done analyzing graph
running graph rules...
... done running graph rules

Online checks summary:

Found 3 error(s).

ERROR Communication with [/example4] raised an error:
ERROR Communication with [/rosout] raised an error:
ERROR The following nodes should be connected but aren't:
 * /example4->/rosout (/rosout)
```

通常情况下，我们所希望获得的结果是没有任何的错误和警告，但是很多错误和警告在系统正常运行时是没有影响的。在上面的截屏中，我们看到 roswtf 已经检测到无法和 example4 节点相连接。之所以会这样是因为这个节点已经收到了挂起（sleep）指令，而当你在它挂起时进行分析的时候，总是会出现这样的提示。而后面的错误都是这个错误造成的。roswtf 的作用是检测信号的潜在问题，然后就像上面示例一样，我们再来检查这些提示是否有意义。

3.5 画标量数据图

我们可以使用 ROS 中现有的一些通用工具，轻松地绘制标量数据图。它要求对每一个

标量字段数据分别绘制成二维曲线。这就是我们在这里讨论标量数据的原因，大部分非标量数据结构更适合于使用专用的可视化工具，我们会在后面进行部分介绍，例如，图形、位姿、方向和角度。

3.5.1 用 rxplot 画出时间趋势曲线

在 ROS 系统中，标量数据可以根据消息中提供的时间戳作为时间序列绘制图形。然后，我们就能够在 y 轴上使用 rxplot 工具绘制标量数据。rxplot 工具有一套功能强大的参数语法，允许我们在结构化消息中指定多个字段（当然使用了相当简明的方式）。

为了能够实际展示 rxplot 工具，我们使用 example4 节点，它在两个不同的主题中分别发布一个标量和一个矢量（非标量），这两个主题分别是温度（temp）和加速度（accel）。在这些消息中的值是随机生成的，所以它们没有实际意义，仅用于示范曲线绘制。那么首先运行节点：

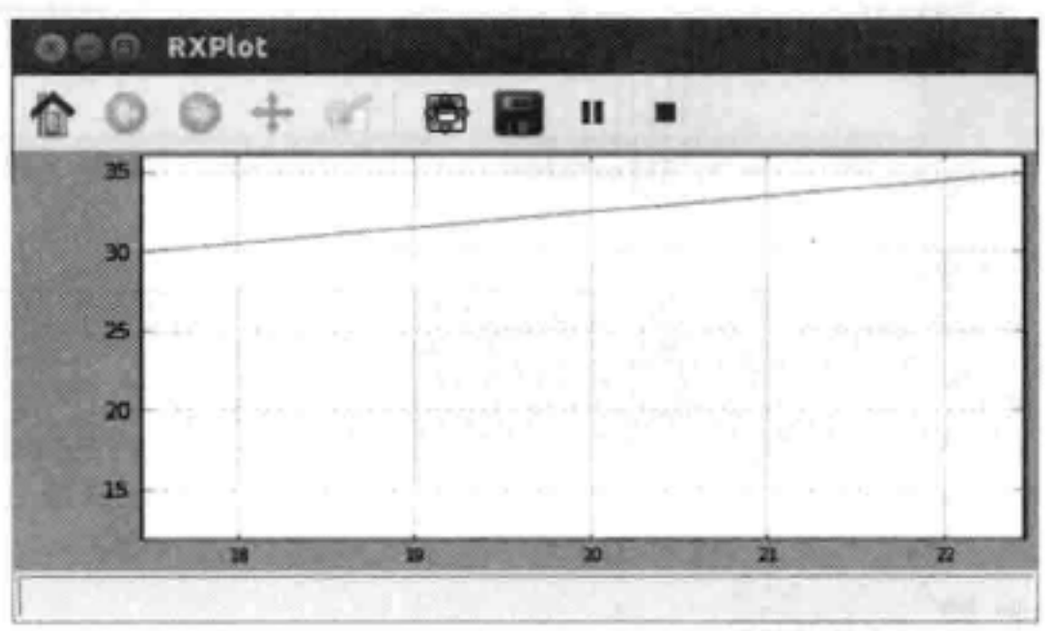
```
rosrun chapter3_tutorials example4
```

通过 rostopic list，你能看到两个活动主题 temp 和 accel。现在，我们不再去使用 rostopic echo <topic>，而是通过 rxplot 命令就能够看到随时间变化的值的曲线。

为了能够画出消息，我们必须知道具体的格式；如果你不知道具体格式则使用 rosmg show <msg type> 获取。对于标量数据，我们通常使用 data 作为字段名来表示实际的值。因此，对于 temp 主题，以 Int32 为数据格式，我们就用：

```
rxplot /temp/data
```

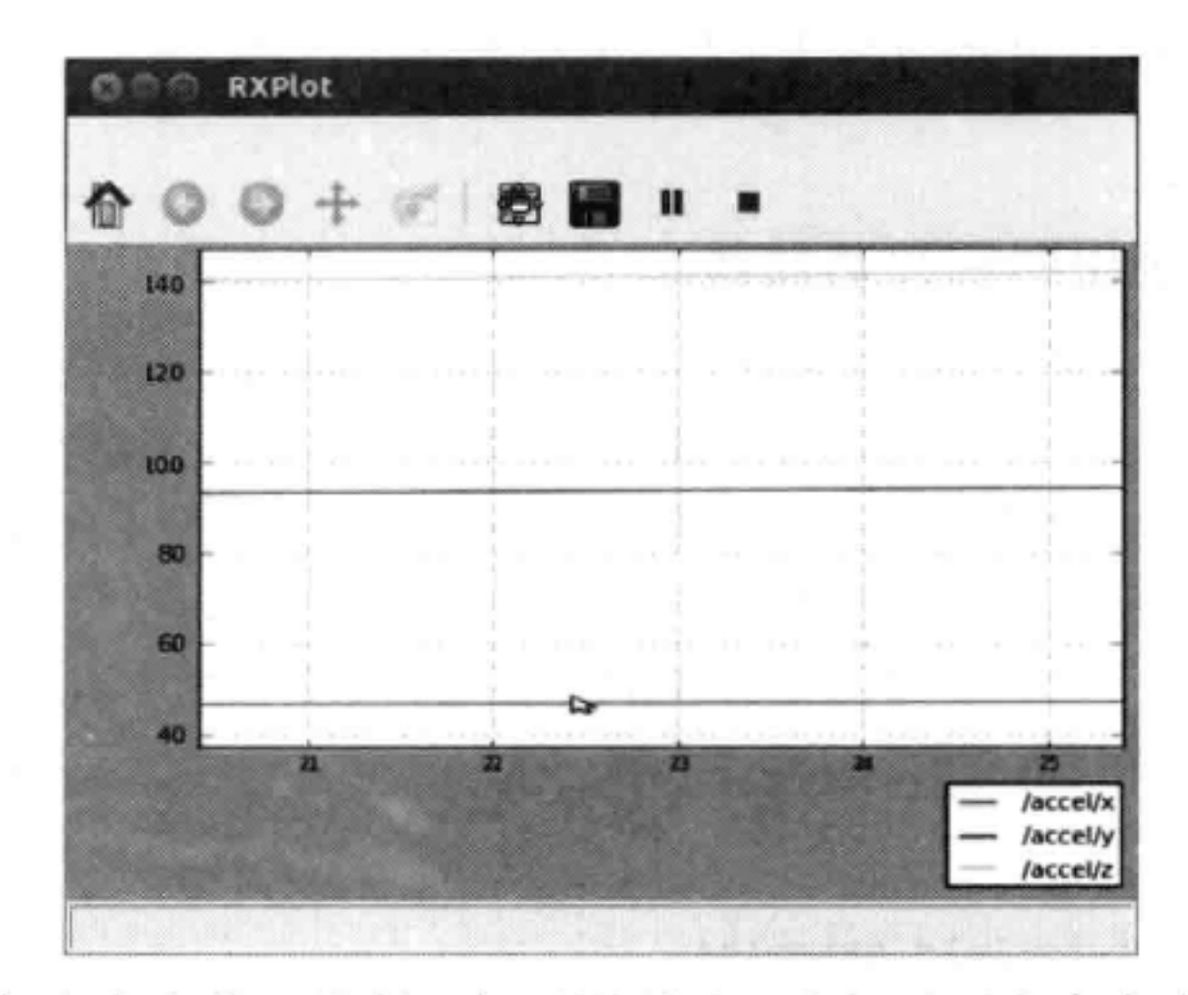
只要节点正常运行，我们就能看到消息中的数值随时间变化的图形。如下截屏中显示：



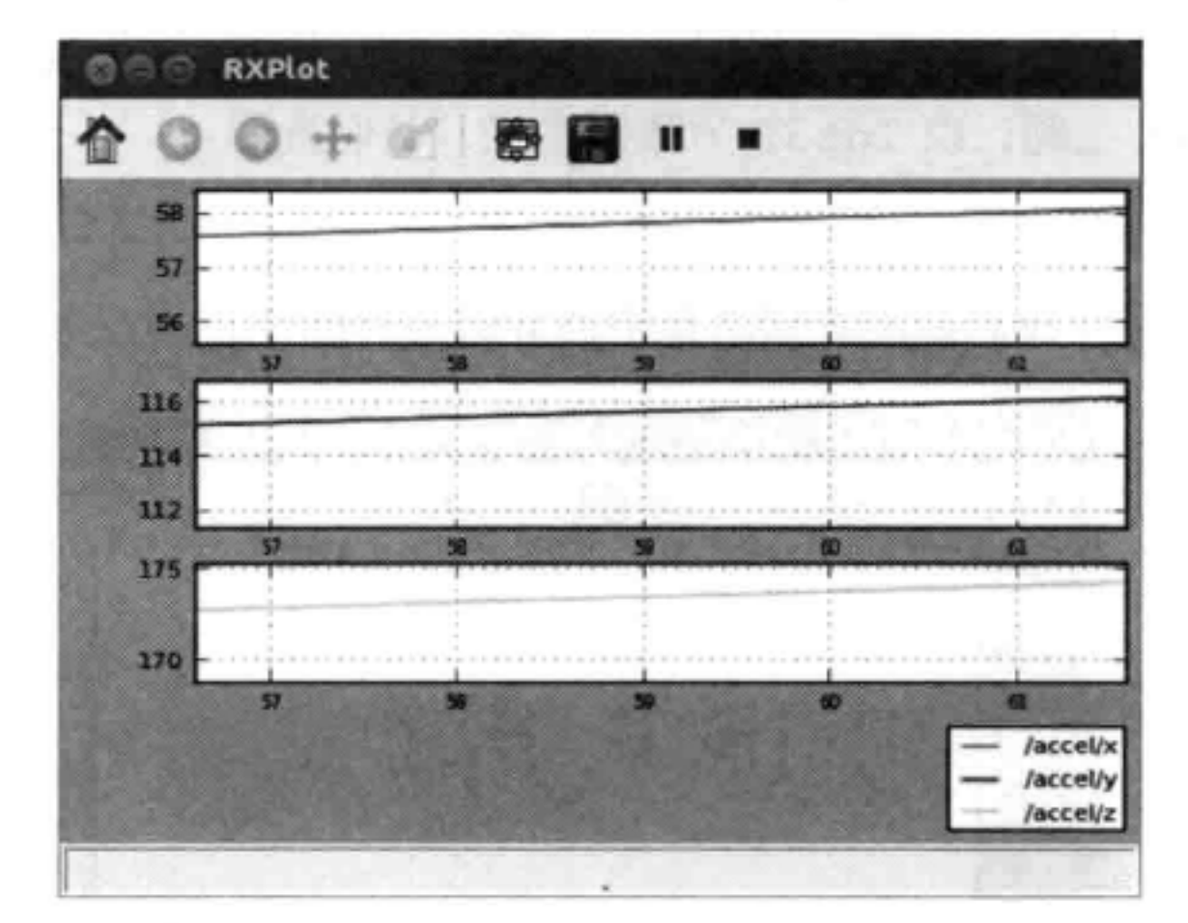
在示例节点提供的 accel 主题里，我们看到一个 Vector3 的消息（你可以通过 rostopic type /accel 来查看），我们可以在一个 plot 图中分别绘制三个字段的曲线，这是 rxplot 工具的一大优势。Vector3 消息包含三个字段 x、y 和 z。我们可以使用逗号来区分字段或者像下面一样使用更加简洁的方式：

```
rxplot /accel/x:y:z
```

绘制的曲线图如下：



我们还可以将每个字段分开绘制，如下图所示。我们需要在命令中增加空格间隔字段名才能够实现这个功能。记住如果你使用逗号，则不允许加入任何空格。因此，如果你运行 `rxplot /accel/x /accel/y /accel/z`，就会出现：



3.5.2 另一个画图工具 rxtools

在 rxtools 功能包内除了 rxplot 工具之外还有其他很多工具。你可以到这个功能包的路径下去查看更多的 GUI 或批处理工具，这些工具都可以帮助更加快捷地开发机器人应用程序、进行调试、查看数据和检查内部信息。这些工具还能够作为功能包内的一个节点运行，这是非常重要的功能，它允许通过 launch 文件启动这些工具。

以 rxplot 为例，为了能够通过 launch 文件运行它，需要插入以下代码：

```
<node pkg="rxtools" type="rxplot" name="accel_plot"
          args="/accel/x:y:z"/>
```

需要注意的是在 launch 文件中我们通过 node 的 args 字段向 rxplot 传递相关参数。

3.6 图像可视化

在 ROS 系统中，我们可以创建一个节点，在节点中展示来自即插即用摄像头的图像。我们需要系统连接一个摄像头来完成这项工作。在 ROS 中也能够在某个节点上复现一段视频，但是在示例里我们会直接使用笔记本上的 webcam 摄像头。

在 example6 节点中，通过调用 OpenCV 库插入一段基本的摄像头捕捉程序，然后在 ROS 中将采集到的 cv：：Mat 图像转换到 ROS 图像，这样就可以在主题中发布了。这个节点会在 /camera 主题里发布图像帧。

我们将会使用 launch 文件运行节点并进行图像捕捉和发布工作。节点中的代码对于读者来说可能很陌生，但是在下面的章节中，我们将会介绍如何在 ROS 中使用摄像头和图像，到时候再回来看这些代码，你就会完全理解节点的工作原理和每行代码的含义：

```
roslaunch chapter3_tutorials example6.launch
```

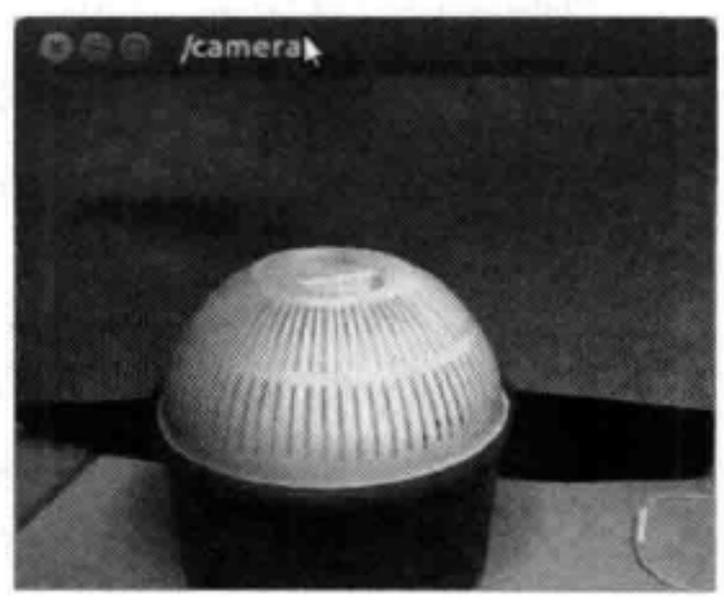
一旦节点运行起来，我们就能够列出主题列表(rostopic list)，查看是否包含 /camera 主题。想要查看我们是否正确捕捉到图像有一个简单直接的方法，使用 rostopic hz /camera 语句查看我们在主题中收到的图像是在哪个频率下。频率通常是 30Hz，但是我们至少看到了图像的一些参数：

```
subscribed to [/camera]
average rate: 30.131
min: 0.025s max: 0.045s std dev: 0.00529s window: 30
```

3.6.1 显示单一图片

想要查看一张图像，我们不能直接使用 rostopic echo /camera 命令，因为这会使用文本格式输出图片的信息，这些数据量会非常巨大，不可能进行任何有效的分析。因此，我们会调用下面的命令：

```
rosrun image_view image_view image:=/camera
```



这里使用了 image_view 节点，在窗口中展示了给定主题（使用 image 参数）的图像，如右图所示。

这样我们就能简单直接地显示在主题内发布的图像的每一帧，即使数据来自网络也可以实现。你可以将当前帧保存在硬盘里，通常会存在 home 目录下或者 ~/.ros 目录下。

3.6.2 FireWire 接口摄像头

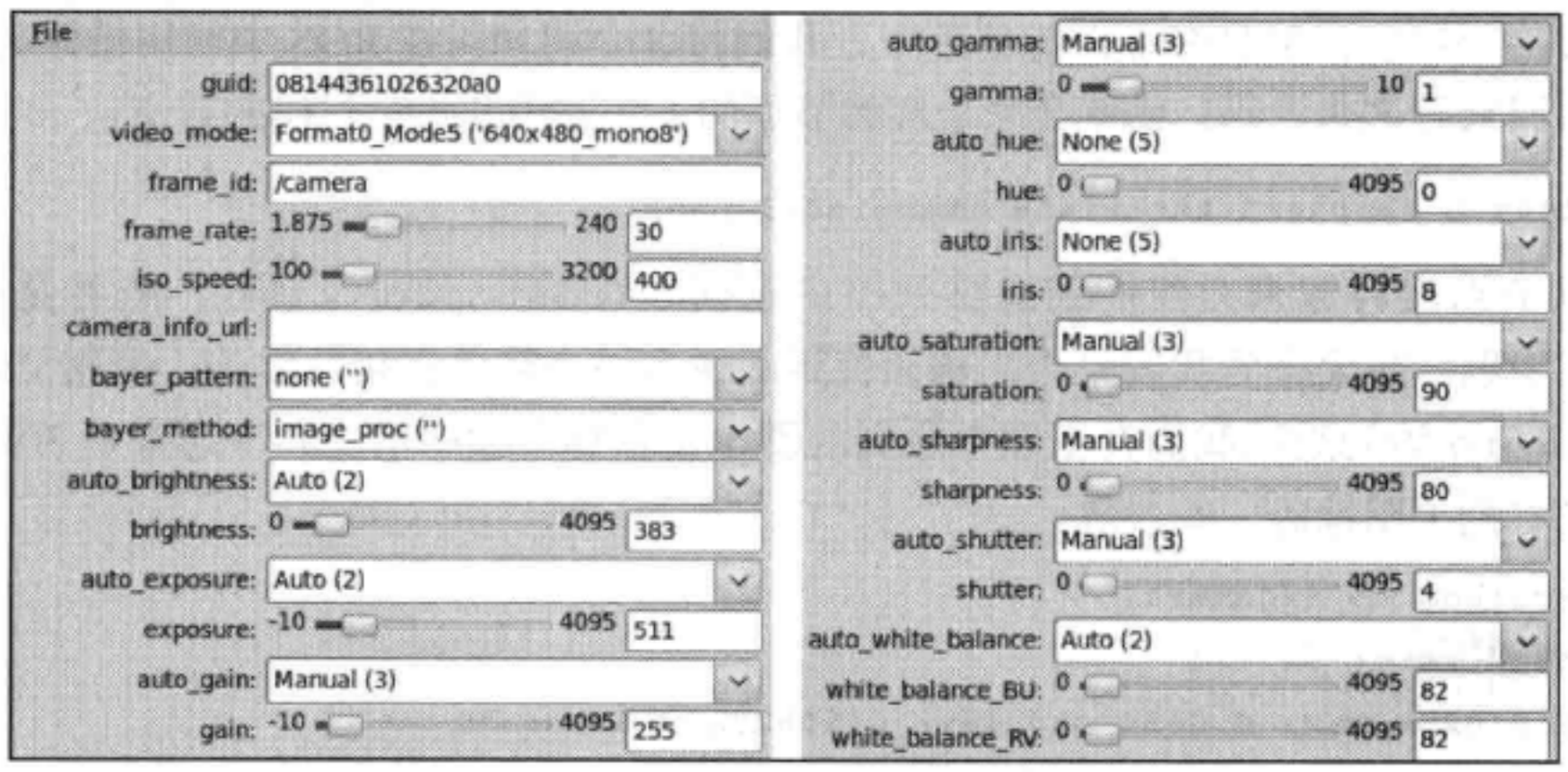
在使用 FireWire 接口的摄像头时，ROS 会提供大量的工具来支持摄像头标定，无论是单目视觉还是双目视觉都支持，还可以通过 reconfigure_gui 节点动态调整摄像头的参数。通常情况下，FireWire 摄像头允许改变传感器的配置参数，例如帧分辨率、快门速度和亮度。ROS 直接带有 FireWire(IEEE1394，a 和 b) 摄像头的驱动，我们可以使用以下命令来调用：

```
rosrun camera1394 camera1394_node
```

一旦摄像头启动，我们能通过 reconfigure_gui 节点配置这些参数，我们只需要选择要配置的节点，然后运行以下命令：

```
rosrun dynamic_reconfigure reconfigure_gui
```

这时将会看到带有所有配置参数和一系列下拉框的界面，界面具体的内容取决于摄像头数据的类型和各个数值的限值。下图就是某种 FireWire 摄像头的参数配置截图：



请注意我们将会以开发者的角度在后面的章节专门介绍如何使用摄像头，以及如何配置这些参数。这些内容都会在第 6 章中介绍。

另一个要说明的是 ROS 系统能够帮助开发者标定摄像头，这些标定接口都是通过 OpenCV 和其他 API 实现的。

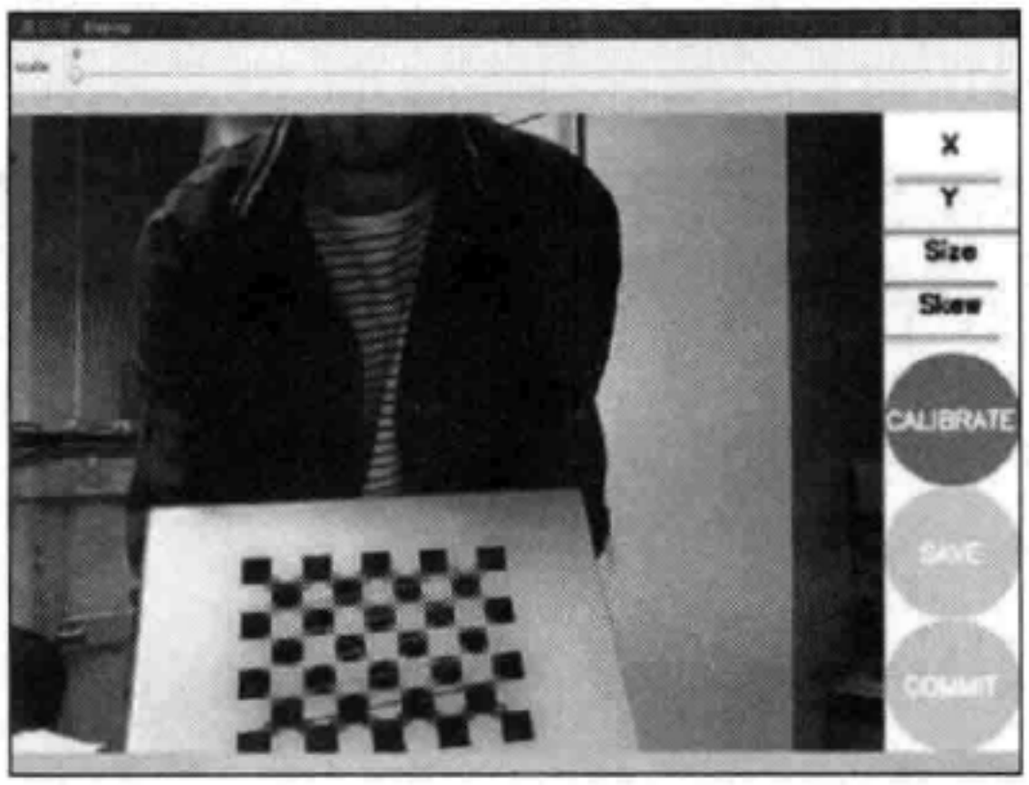
我们同样会在第 6 章介绍如何使用这些接口。这些工具使用起来非常简单，摄像头一运行起来，我们只需要使用下面代码调用一些常用标定图案（通常使用棋盘）：

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108
image:=/camera/image_raw camera:=/camera
```

我们可以在下图中看到棋盘图案。

标定之后，标定结果会保存为摄像头信息和失真矩阵参数，以及计算这些参数的视图。我们在后面会经常看到这样的图像。在使用 FireWire 摄像头的情况下，会将所有的标定信息存储在一个能够说明摄像头配置的文件中。ROS 系统无缝集成了 image_ proc 工具实现

图像的修正，也就是修正失真或者说对拜耳（Bayer）模式的源图像进行恢复和去拜耳化（de-Bayer）。

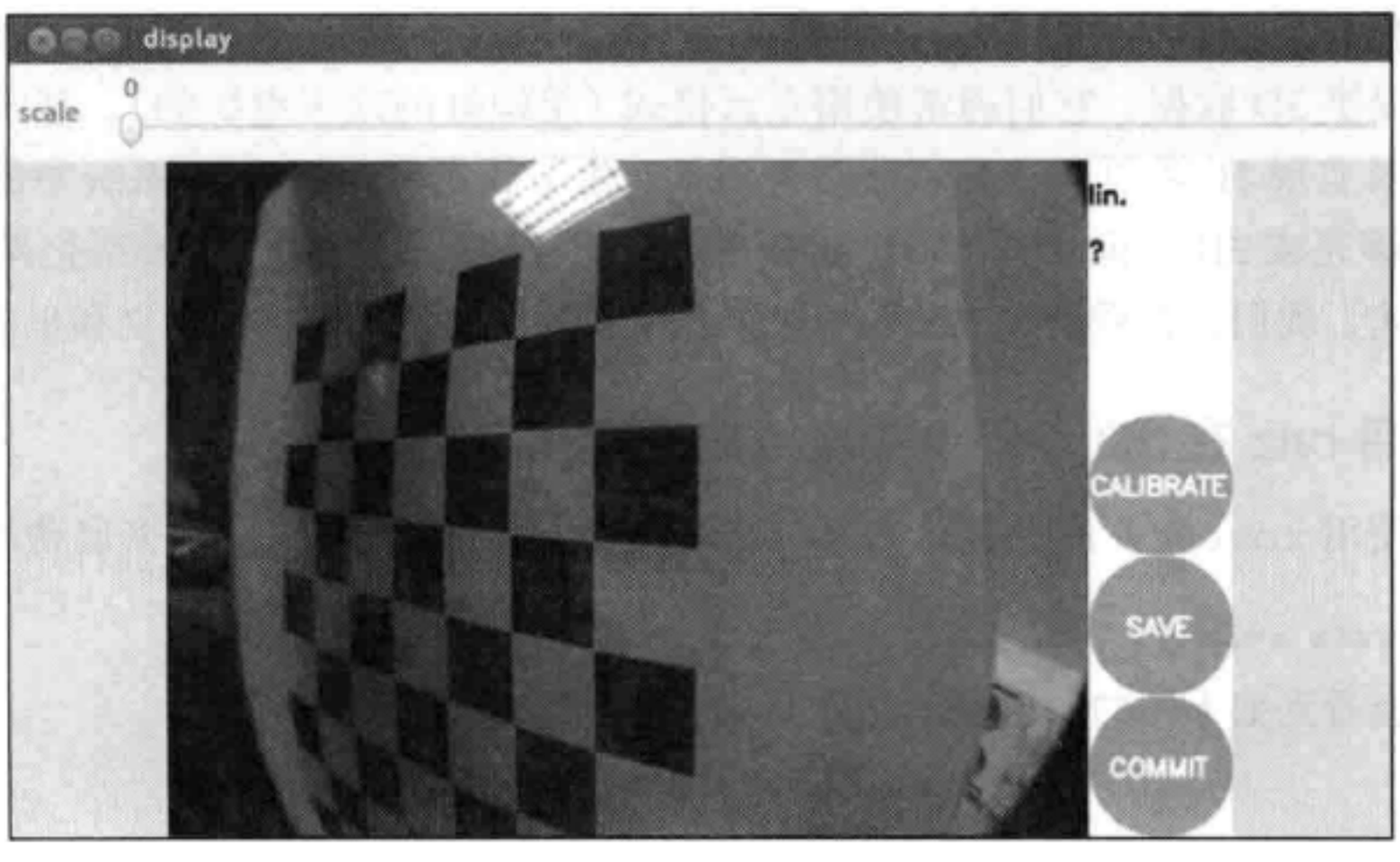


3.6.3 使用双目立体视觉

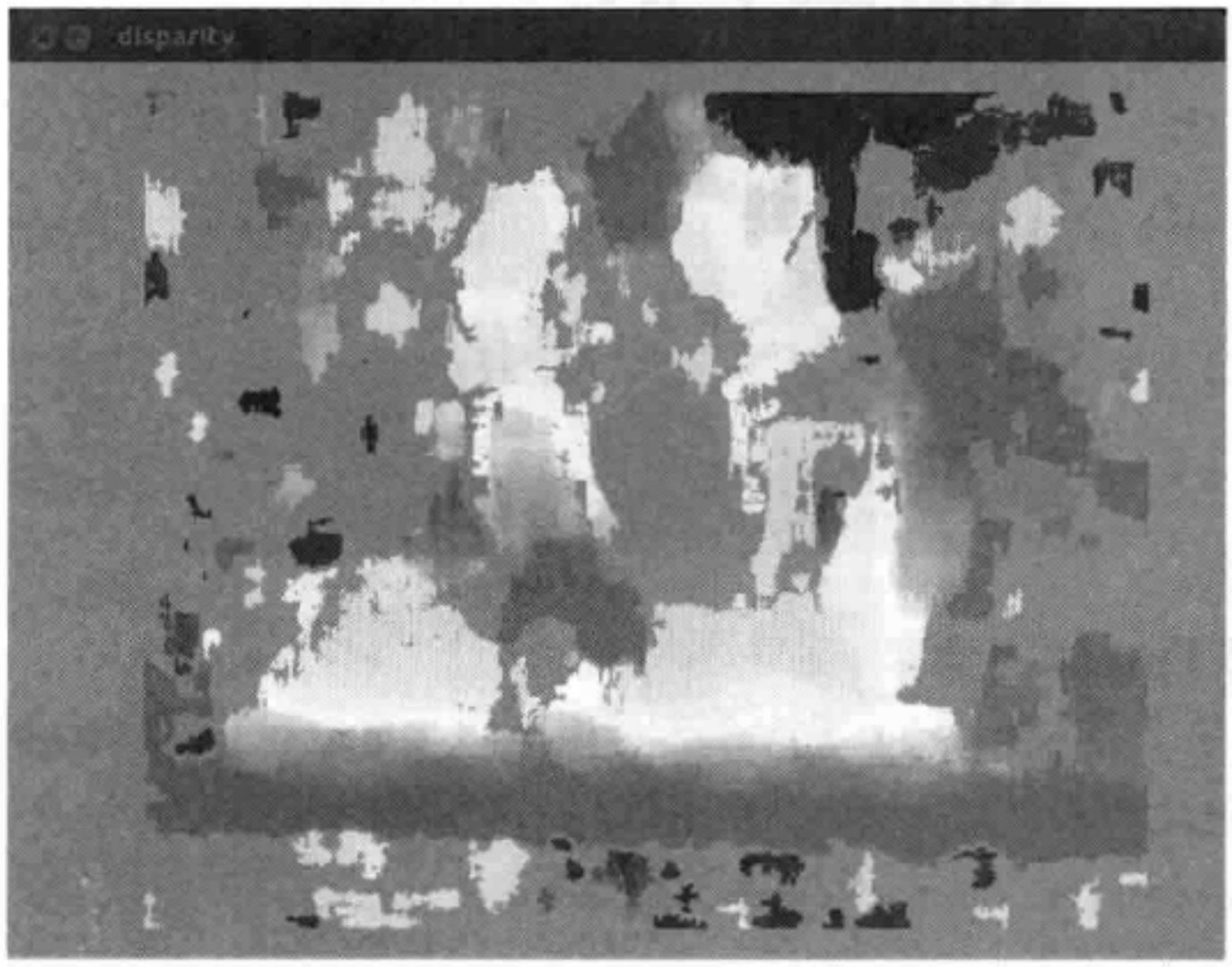
在某种程度上，ROS 系统同时支持双目立体视觉。如果你有双目摄像头，就能够同时标定两个摄像头和调整它们的基线。我们可以调用以下代码来实现：

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108
right:=/my_stereo/right/image_raw left:=/my_stereo/left/image_raw right_
camera:=/my_stereo/right left_camera:=/my_stereo/left
```

这些命令能在 Python 摄像头标定节点中运行，并会收到两种表明标定模式的初始化参数。特别地，指定的大小是内角的数目（示例中是 8×6）和单元格的尺寸。然后，会给出发布左右两个源图像的主题和各种摄像头信息的主题。在示例中，接口会显示左右摄像头的图像，结果也是分别展示的。其中也会包含摄像头的基线，这对很多双目视觉工具非常重要的。



双目视觉专用工具会允许你计算视差图像，如下图所示。这实际上是一种获取 3D 点云的方法（3D 点云能够显示每个像素点在实际物理世界中的深度）。因此，摄像头标定和基线会提供给 3D 点云，处理图像错误和噪声分布，从而能够表征每个像素在真实世界中的真实 3D 位置及其颜色和纹理。



对于单目摄像头来说也是一样的，我们能够通过拍摄类似于双目视觉的左右图像，生成视差图像。在这样的情况下，需要使用 `stereo_image_proc`。

3.7 3D 可视化

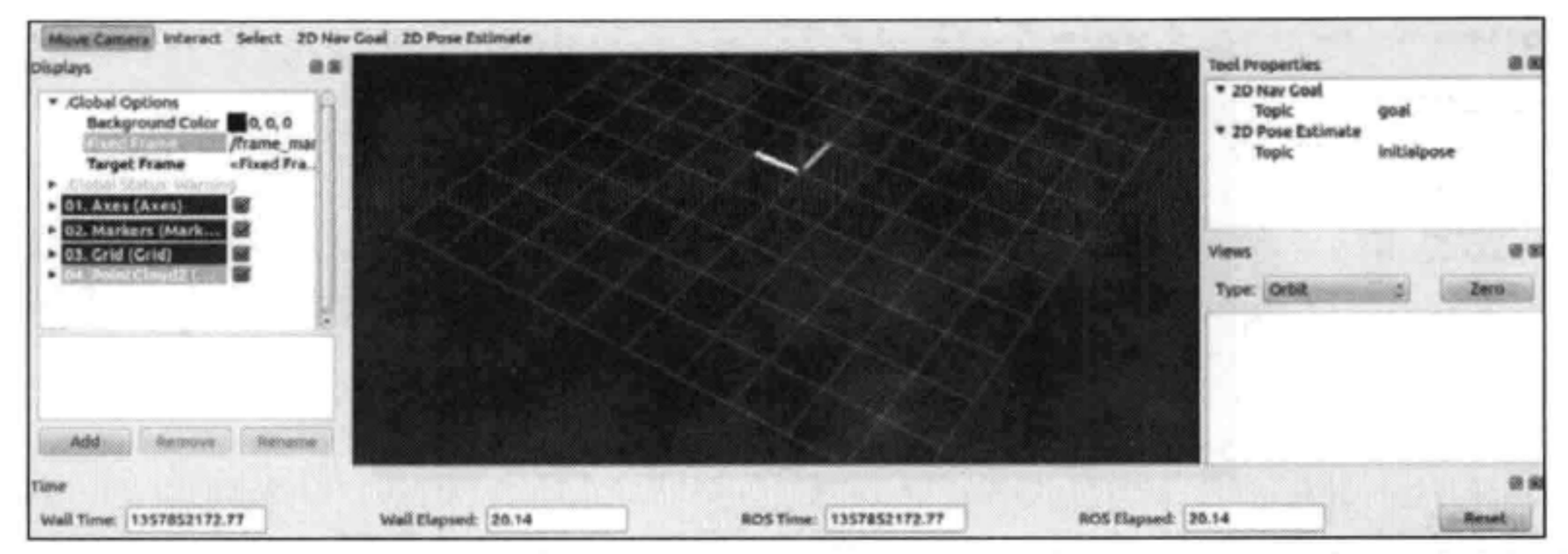
正如我们在前面章节中所见，有很多设备（例如双目摄像头、3D 激光雷达和 Kinect 传感器）能够提供 3D 数据。它们通常使用点云格式（组织好的或未组织的）。基于上述原因，能够使用工具实现 3D 数据可视化就非常有用了。本节我们将介绍 ROS 系统中的 rviz 工具。它集成了能够完成 3D 数据处理的 OpenGL 接口，能够将传感器数据在模型化世界（world）中展示。因此，我们将会看到在复杂系统中至关重要的传感器的坐标系建立和坐标变换。

3.7.1 使用 rviz 在 3D 世界中实现数据可视化

首先，使用 `roscore` 命令启动 ROS，然后我们仅需要使用以下命令来启动 rviz：

```
rosrun rviz rviz
```

我们将会看到如下图的图形化工作界面：



在左边，有一个 Displays 面板，在面板的中间有一个包含了模拟环境下不同参数项的树形列表。在示例中，我们已经加载了部分参数项。实例中的配置和布局都存储在了 `config/example7.cvg` 文件中，可以通过点击 File|OpenConfig 加载配置。

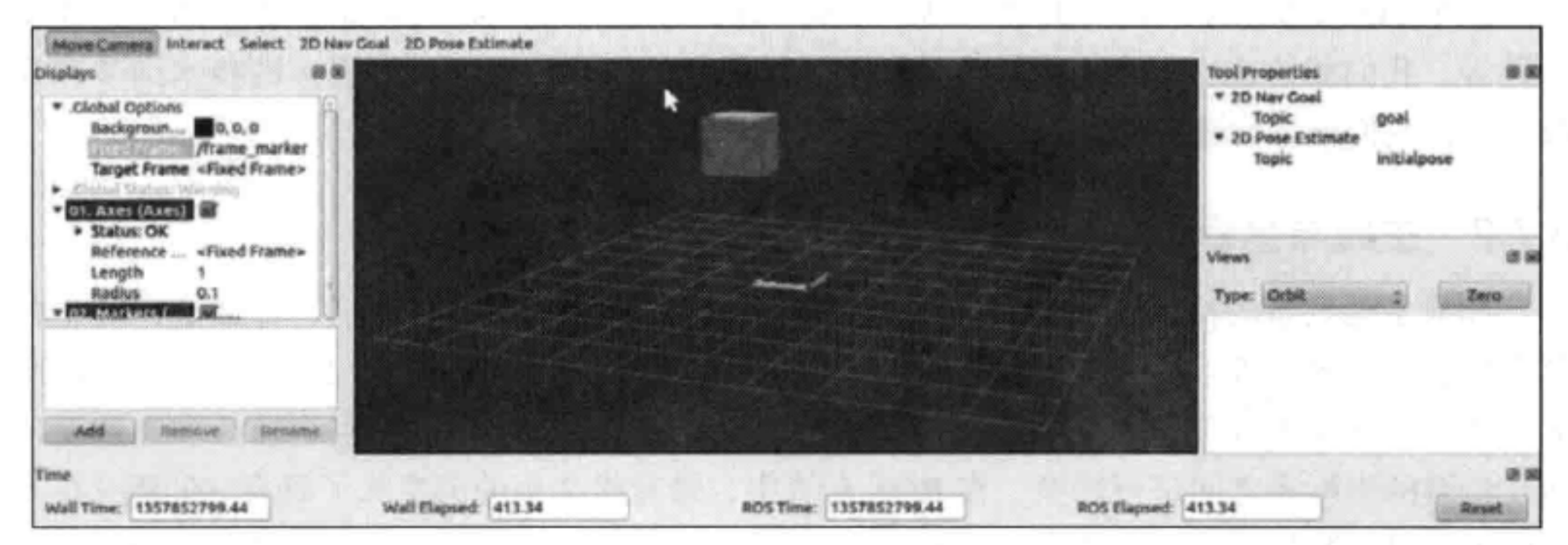
在 Displays 区域之下，我们有一个 Add 按钮，允许添加更多的参数项。同时，注意到这里还有一些全局选项，基本上是用于设定固定坐标系的工具，因为坐标系是可以移动的。其次，还有轴（Axes）和网格（Grid）作为各个参数项的参照物。在示例中，对于 `example7` 节点，我们将会看到标记（Markers）和点云（PointCloud2）。

最后，在状态栏上，有时间相关的信息提示，以及在右侧，有用于在虚拟环境中导航、选择和操纵参数项的菜单。

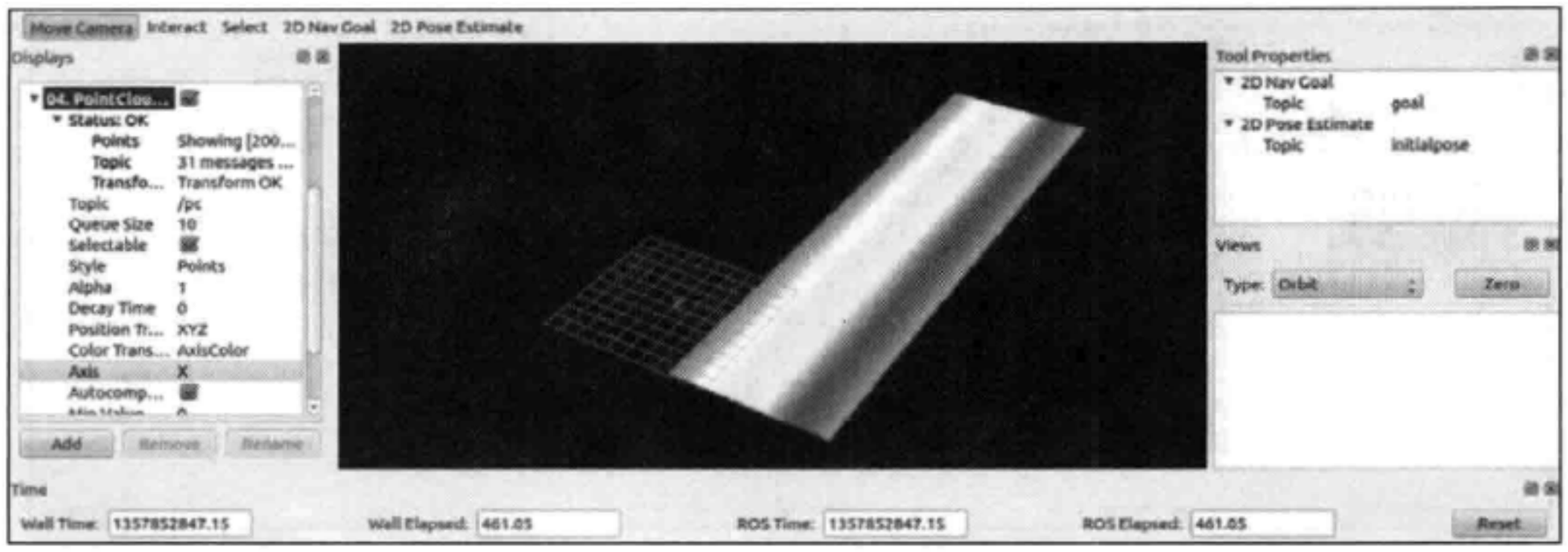
现在运行 `example7` 节点：

```
roslaunch chapter3_tutorials example7.launch
```

在 rviz 中，我们将会把 `frame_id` 设置为标记，这个标记是固定坐标系的坐标标记（frame_marker）。我们将会看到如下图所示的红色方块标记在移动：



类似地，如果设置固定坐标系为 `frame_pc`，我们将看到一个 200 × 100 像素点平面所组成的点云，如下图所示：



它支持的 rviz 内置类型的参数列表也包括影像和图像。它们会在一个类似于 image_view 的窗口中进行展示。在使用摄像头的时候，需要对它先进行标定；在使用双目视觉图像的时候，它允许我们覆盖点云。我们同样也可以看到激光雷达的激光扫描数据，以及 IR/SONAR（红外 / 声呐）传感器的锥状距离数据（range cone）。

它能够表现机器人相关的各种基本元素，例如多边形、各种不同的标记、地图（通常是 2D 网格地图），甚至是交互式标记对象。交互式标记对象允许用户设定标记对象在 3D 环境中的位姿（位置和方向）。

对于导航功能包集，将会在下一章进行详细的介绍。我们会使用 rviz 支持的多种数据类型，例如里程（画出机器人的里程位姿）、路径（画出机器人所走过的路径），这些包括了机器人的多个连续位姿对象。在这些类型中，需要说明的是机器人的模型，它能够展示机器人组件的 CAD 模型，并将每个组件的坐标系之间的转换考虑进去，甚至还能画出坐标变换（Transform Frame，TF）树，并且在仿真环境中为坐标系调试提供非常大的帮助。我们会在下一小节看到示例。

这个 3D 图形化接口还能够嵌入新的 `rqt_gui` 的 GUI 中，我们也能够为新的类型开发插件等。我们相信在本书中提供的信息已经足够了。如果你有兴趣学习更多的技术细节和高级技巧，我们推荐你查看 rviz 官方文档。

3.7.2 主题与坐标系的关系

如果主题发布的特定传感器数据在真实世界里具有一个物理位置的话，那么这样的主题就需要有一个坐标系。例如，相对于机器人质心有一段距离的某个位置上有一个加速度计。如果我们要集成加速度数据来估计机器人的速度和位姿，那么必须对机器人质心和加速度计所在位置的坐标系之间进行转换。在 ROS 系统中，带有报文头的消息除了具有时间戳（在不同的消息进行数据同步时也非常重要）之外，还要附上 `frame_id`（坐标系标签）。坐标系标签用于区分消息所属的坐标系。

但当机器人具有多个设备，且每个设备都有不同的坐标系或位姿时，坐标系本身并没有

更多的用处。我们仅利用其进行坐标变换而已。实际上，机器人的坐标变换树都会有一个根坐标系（root）作为参照。这样，我们就能够在 rviz 中通过对比根坐标系和其他坐标系查看机器人相对于真实世界坐标系的运动。

3.7.3 可视化坐标变换

为了解释如何进行坐标变换的可视化，我们将会再次使用 turtlesim 的示例。运行以下 launch 文件：

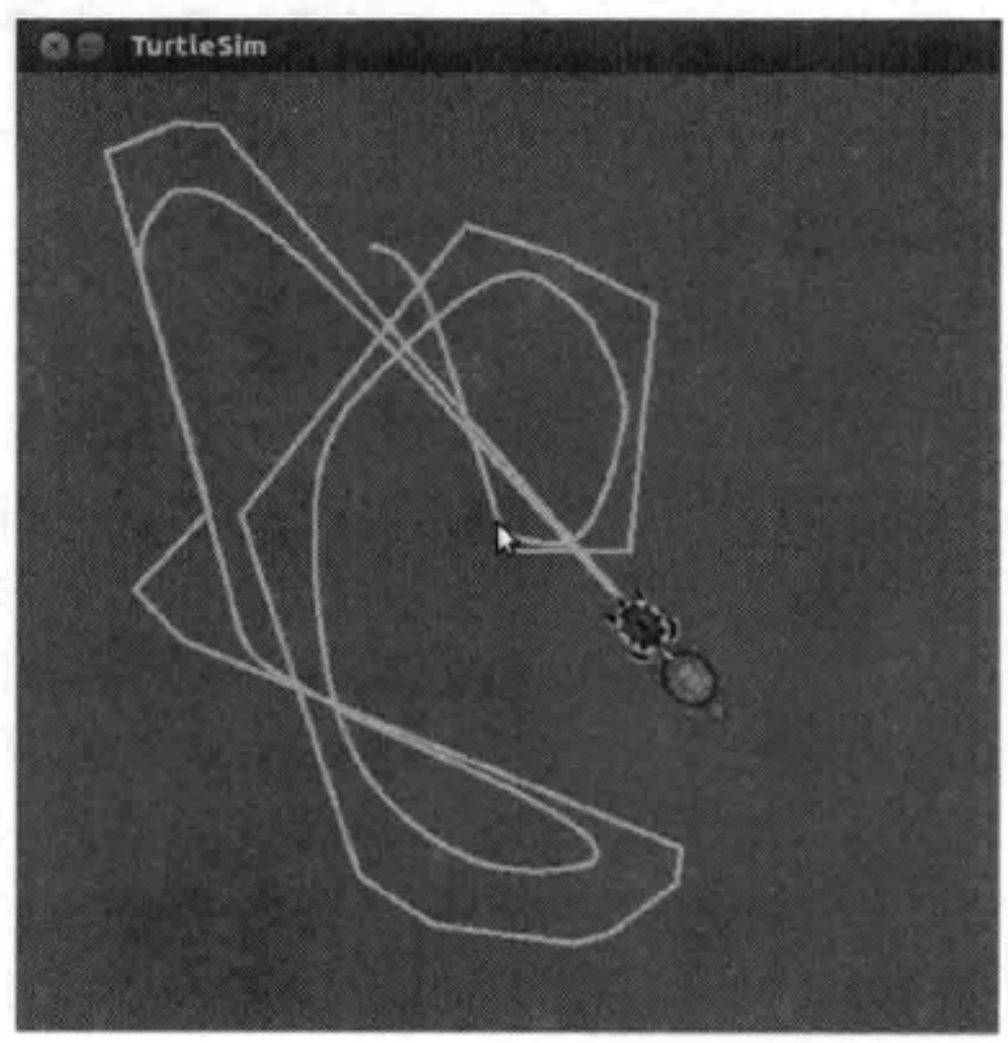
```
roslaunch turtle_tf turtle_tf_demo.launch
```

这是一个在 rviz 里能够用于解释坐标变换可视化的最基本的示例。需要注意的是 TF API 提供了很多不同的可能性，你可以参考本书的后续章节，特别是第 7 章和第 8 章。而对于现在的示例，我们只需要知道它们能够在某个坐标系内进行计算并进行坐标变换到另外一个坐标系，而且包含时间延迟即可。

我们还需要了解在系统中坐标变换会以某个特定的频率进行发布，这样它就会像子系统一样允许我们遍历坐标变换树以获取其中任意两个坐标系之间的转换，并且可以在系统的任意节点中通过调用 TF API 进行变换。

如果你收到一个错误，这很可能是因为在 launch 文件启动时侦听器（listener）失效了。侦听器是一个必备的节点，必须准备好才能进行坐标变换。因此，请在另外一个命令行窗口下运行以下命令：

```
rosrun turtle_tf turtle_tf_listener
```



现在你应该会看到一个带有两只小海龟的窗口（小海龟的图标不同），一只小海龟跟着另一只。你可以使用键盘上的上下左右四个箭头按键控制小海龟。要注意要让操作系统的焦点停留在运行 launch 文件的那个命令行窗口上，而不是在 TurtleSim 窗口上，这样才能让小海龟移动。右图显示了在我们控制一只小海龟移动了一段距离之后，另一个小海龟是如何跟随着这只小海龟的。

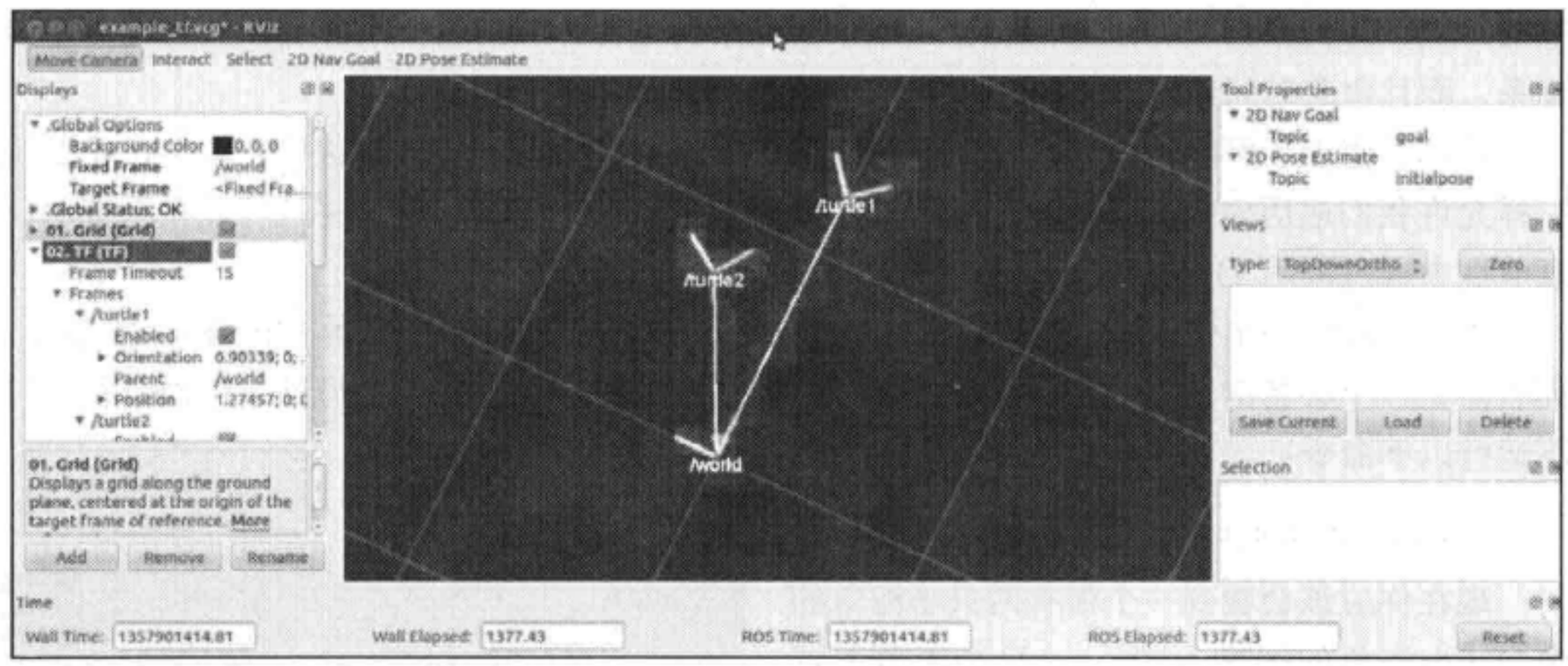
每一只小海龟都有自己的坐标系，可以在 rviz 中查看它们：

```
rosrun rviz rviz
```

现在，先放下 TurtleSim，我们先来看当使用箭头键移动小海龟时 rviz 里小海龟的坐标系是如何移动的。先设定固定坐标为

/world，然后将坐标转换树添加到左边的区域。我们会看到 /turtle1 和 /turtle2 两个坐标系，它们同为 /world 坐标系的子坐标系。在 world 展现的环境中，坐标系由轴组成。/world 坐标系是固定的，因为我们在 rviz 中将其配置成这样。它同时是根坐标系和小海龟所在坐标系的父坐标系，由一个带有粉红色尾巴的黄色箭头所表示。同时，设定 world 的视图（view）为 TopDownOrtho，因为这会使示例中检查坐标系的移动更容易，它们仅在 2D 平面中移动。你还会发现通过鼠标操作转换 world 视图显示的中心非常有用，只需要你按下 shift 按键同时转动鼠标。

在下图中，你能看到不同海龟所在的两个坐标系如何转换到 /world 坐标系中并进行显示。我们建议读者针对示例加以练习，并观察实时的现象。同时，你也可以更改固定坐标系。请注意 config/example_tf.vcg 是用来作本示例的基本 rviz 配置的。



3.8 保存与回放数据

通常情况下，当我们使用机器人系统时，资源都是共享的，并不一定总能提供给你一个人使用，或者因为种种原因实验不能顺利完成，如准备和实施实验就消耗了大量的时间。基于以上原因，将实验会话数据记录下来用于未来的分析、处理、开发和算法验证，就是一个学习者必修的课程。然而要保证存储数据正确并能够用于离线回放并不是容易的事情，因为很多实验并不一定能反复进行。幸运的是，我们拥有 ROS 中的强大工具能够完成这个功能。

ROS 能够存储所有节点通过主题发布的消息。它能够创建一个消息记录包文件来保存消息，并包含消息的所有字段参数和时间戳。这允许离线回放实验并模拟真实的状态，包括消息的时间延迟。甚至 ROS 工具能够非常有效地在处理高速数据流时进行高效的数据组织的方式来实现这一切。

在下面的小节中，我们会解释 ROS 中提供的这些用于存储和回放数据的工具，以及由

ROS 开发者设计的使用二进制格式存储数据的消息记录包文件。我们还将看到如何管理这些文件，也就是说，检查文件内容（消息的数量、主题等）、文件压缩、分割和合并多个记录包文件。

3.8.1 什么是消息记录包文件

消息记录包是一个包含各个主题所发消息的容器，用于记录机器人各个节点间的会话过程。简而言之，它们是系统运行期间消息传送的记录文件，并能允许我们回放所有一切，甚至包括时间延迟。因此所有消息在记录时都会打上时间戳，不仅是报文头里的时间戳，还包括使用消息的功能包。用于记录的时间戳和报文头内的时间戳之间的区别在于前者是消息被记录时的时间，而后者是消息产生和发布时的时间。

消息记录包中存储的数据使用二进制格式。这个容器使用的数据结构非常特殊，能够支持对超高速数据流的处理和记录。这是记录数据最关键的能力。当然，这和记录包文件的大小也是相关的。随着文件的增大会牺牲数据的记录速度。我们能选择 bz2 算法进行数据压缩，只需要在你使用 `rosbag record` 命令记录时使用 `-j` 参数，你会在下一节看到具体示例。

每个消息与发布它的主题都被记录下来。因此，我们可以区分某个主题进行记录，或者直接选择全部（使用 `-a`）。然后等你回放记录包的时候，同样可以通过指明特定主题的名称来选择记录包中全部主题的某个特定子集进行回放。

3.8.2 使用 rosbag 在包文件中记录数据

我们首先要做的事情是简单地记录数据。使用一个非常简单的系统，以 `example4` 节点为例。因此，先运行节点：

```
rosrun chapter3_tutorials example4
```

现在我们有两个选择。第一个是记录所有的主题：

```
rosbag record -a
```

或者，选择第二个，仅记录一些特定（用户自定义）的主题。在这个示例中，仅记录 `example4` 的主题是合理的选择，我们可以使用以下指令：

```
rosbag record /temp /accel
```

默认情况下，当运行前面的命令，`rosbag` 程序会订阅相关节点并开始记录消息，将数据存储在当前目录下以数据内容作为文件名的消息记录包文件中。一旦你完成实验或者想结束记录，只需要点击 Ctrl + C。下面就是一个记录数据和包文件结果的示例：

```
[ INFO] [1357815371.768263730]: Subscribing to /temp
[ INFO] [1357815371.771339658]: Subscribing to /accel
[ INFO] [1357815371.774950563]: Recording to 2013-01-10-10-56-11.bag.
```

使用 rosbag help record 可以看到更多的选项，其中包括记录包文件的大小、记录的持续时间、分割文件到某个给定大小的选项。或者像前文所提到的那样，使用文件压缩（使用 -j 参数）。坦率地讲，这只在数据存储速率较低时有用，因为这同时还会消耗大量的 CPU 时间并可能丢失部分消息。或者，我们可以以 MB 为单位增加记录缓存（-b）大小。缓存大小默认值是 256MB，但在某些数据存储速率特别高的情况下也可能需要 GB 级的缓存（特别是处理图像时）。

将对 rosbag record 的调用直接包含到 launch 文件中去也是可以的。我们需要在 launch 文件中增加一个节点如下所示：

```
<node pkg="rosbag" type="record" name="bag_record"
            args="/temp /accel"/>
```

需要注意的是主题和其他命令传递的参数会使用 args 参数。同时，在使用 launch 文件直接启动记录的时候，记录包文件会默认创建在 ~/.ros 路径下，除非使用 -o（前缀）或 -O（全名）给文件命名。

3.8.3 回放消息记录文件

现在我们有了一个消息记录包文件，可以使用它回放所记录主题发布的所有消息数据。只需要运行 roscore，而不需要再运行其他任何节点。然后，移动到想要回放的消息记录包文件所在的文件夹下（在本章的示例中 bag 文件夹下提供了两个记录包示例），并且输入以下命令：

```
rosbag play 2013-01-10-10-56-11.bag
```

我们会看到如下输出：

```
[ INFO] [1357820252.241049890]: Opening bag/2013-01-10-10-56-11.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
 [RUNNING]  Bag Time: 1357815375.147705   Duration: 2.300787 / 39.999868
```

在回放消息记录包的命令行窗口内，可以暂停（点击空格键）或一步一步前进（点击 S 键），或者使用 Ctrl + C 来马上停止回复。一旦回放完毕，窗口就会关闭，但是有一个选项（-l）允许循环播放，这在有些时候也很有用。

我们可以通过 rostopic list 命令查看主题列表：

```
/accel
/clock
/rosout
```

```
/rosout_agg
/temp
```

/clock 主题是用于指定调整系统时钟以便加快仿真的回放速度。它能够通过 -r 选项进行配置。在 /clock 主题中能够发布仿真时间，并以 --hz 为配置频率（默认是 100Hz）。

同时，我们可以指定文件中将要发布主题的一个子集。这可以通过 --topics 选项来完成配置。为了查看文件中包含哪些内容，可以使用 rosbag info <bag_file> 命令。我们会在下一小节具体解释这个命令。

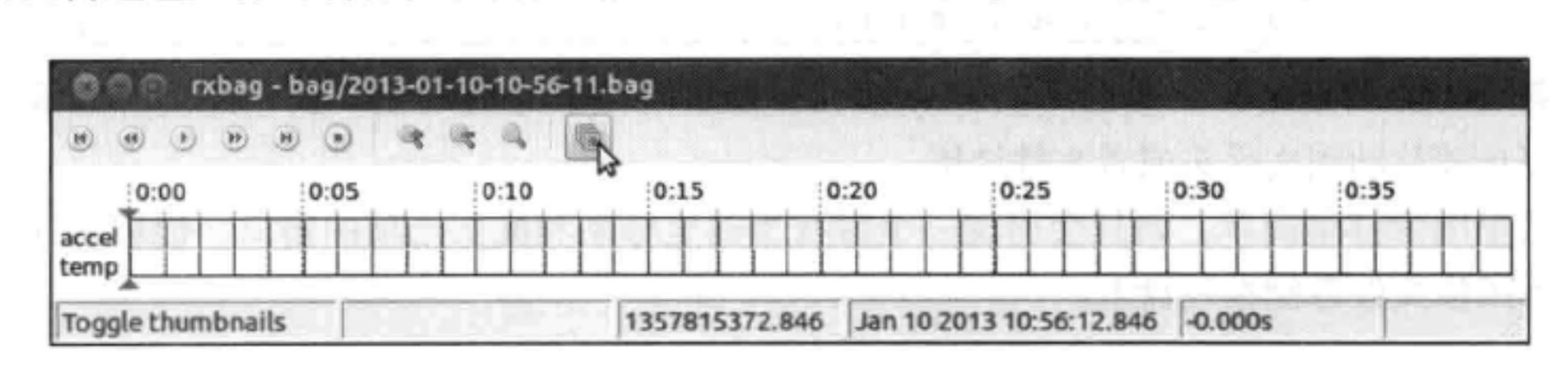
3.8.4 使用 rxbag 检查消息记录包的主题和消息

有两个主要的方法来查看消息记录包内部的数据。第一个非常简单。我们只需要输入 rosbag info <bag_file>，然后结果会如下显示：

```
user@pc$ rosbag info bag/2013-01-10-10-56-11.bag
path:        bag/2013-01-10-10-56-11.bag
version:     2.0
duration:    40.0s
start:       Jan 10 2013 10:56:12.85 (1357815372.85)
end:         Jan 10 2013 10:56:52.85 (1357815412.85)
size:        10.9 KB
messages:    82
compression: none [1/1 chunks]
types:       geometry_msgs/Vector3 [4a842b65f413084dc2b10fb484ea7f17]
             std_msgs/Int32        [da5909fbe378aeaf85e547e830cc1bb7]
topics:      /accel   41 msgs    : geometry_msgs/Vector3
             /temp    41 msgs    : std_msgs/Int32user@pc$ rosbag info bag/2013-01-10-10-56-11.bag
```

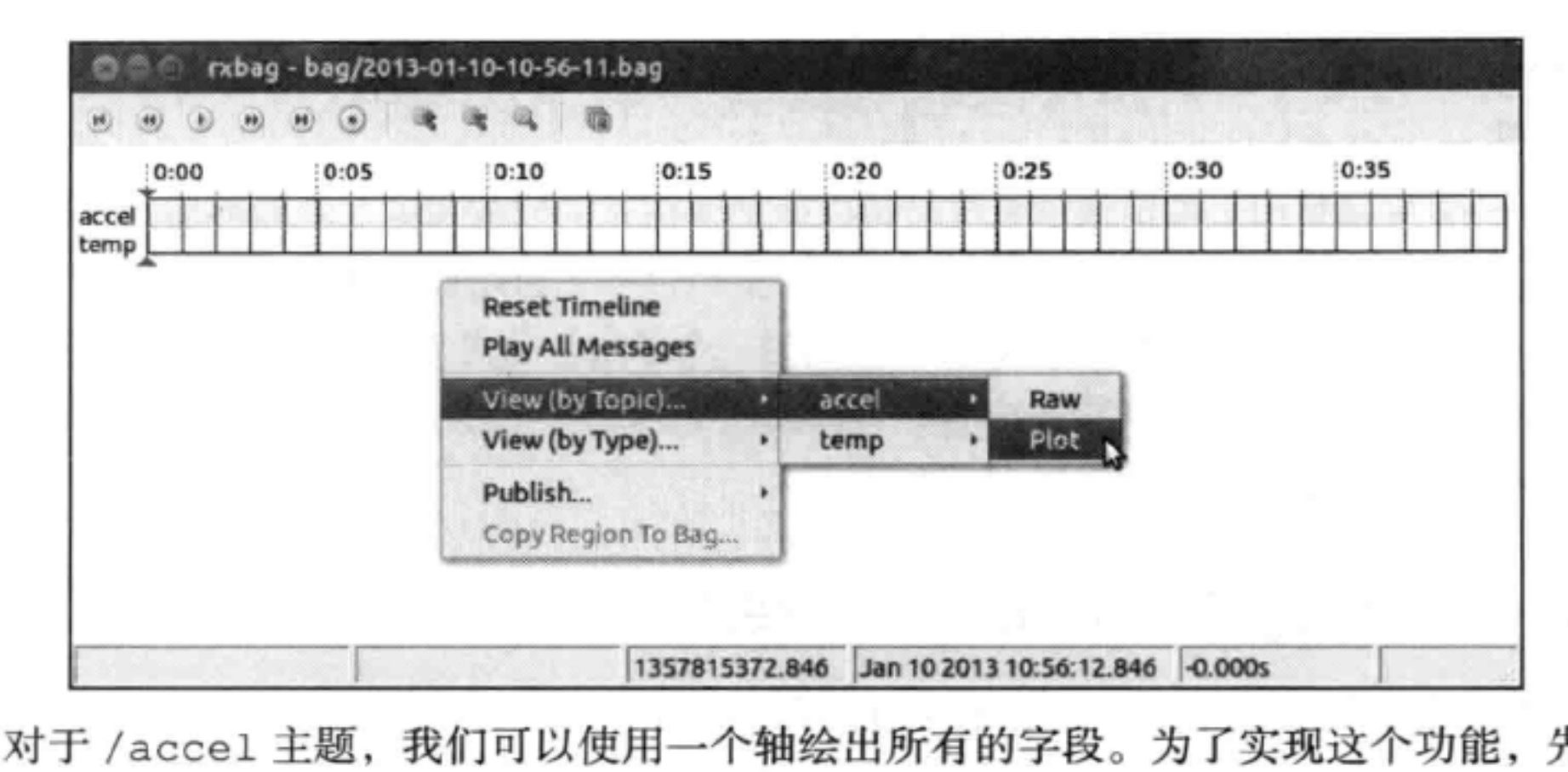
我们有消息记录包文件本身的信息，例如创建的日期、持续时间、文件大小，以及内部消息的数量和文件的压缩格式（如果有压缩）。然后，我们还会有文件内部数据类型的列表。最后，有主题的列表，并对应它们的名称、消息数量和消息类型。

第二种检查包文件的方法则非常强大。这个工具叫做 rxbag。它不仅具有图形界面，还允许我们回放消息记录包、查看图像（如果有）、绘制标量数据图和消息的数据原结构。我们只需要传递包文件的名称，就会看到如下图所示的界面（使用之前的包文件）：

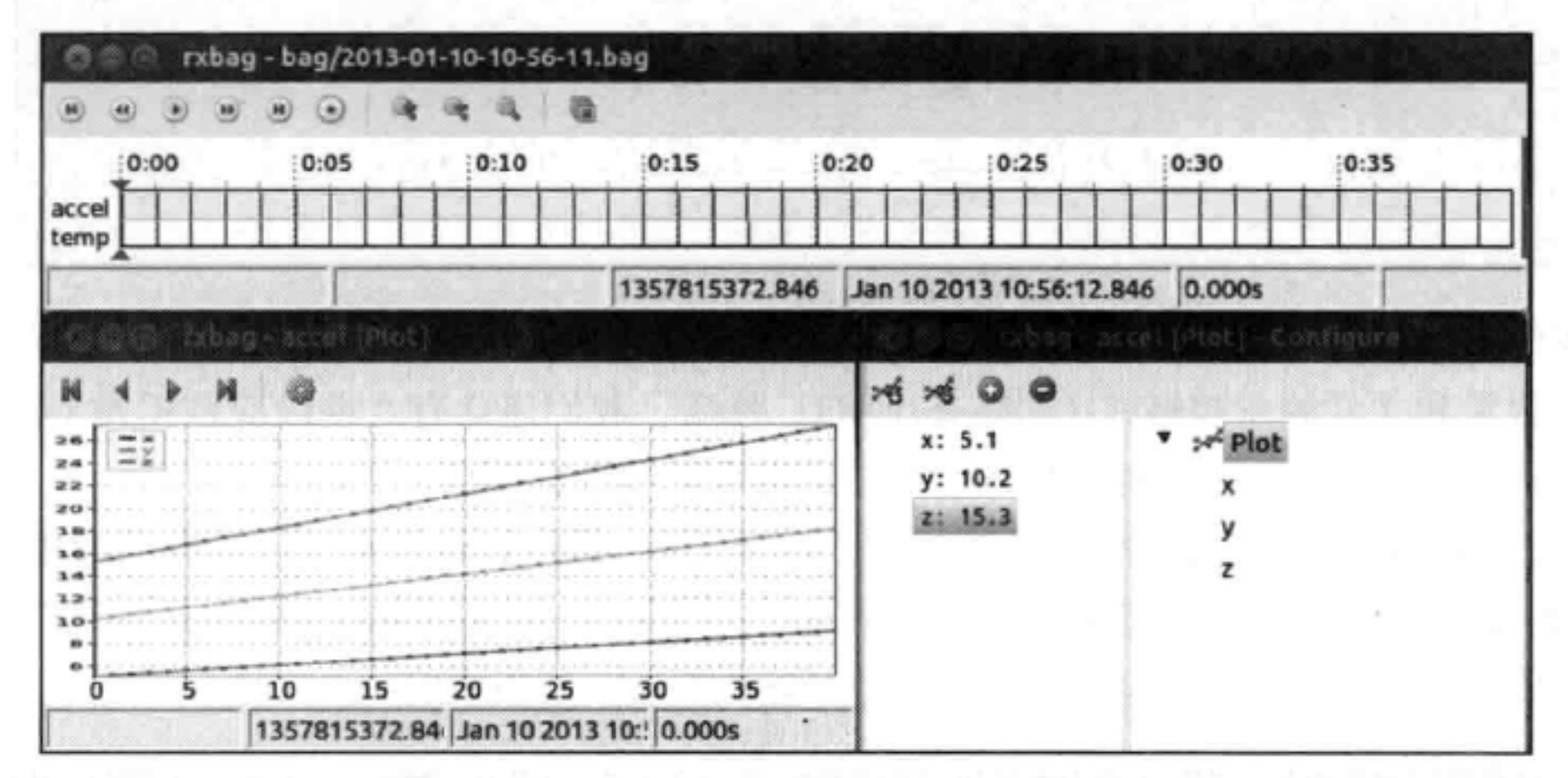


我们有一个能用于所有主题的时间条。对于每一个消息都会带有这个标记。在本示例图中，能够在时间条中使用缩略图查看消息（通过鼠标指针来选择）。

在下图中，我们能够看到如何调用 Raw、Plot 和 Image 指令（如果是图像类型的主题），来查看包文件中的主题。在时间条上单击右键就会弹出这个菜单。



对于 /accel 主题，我们可以使用一个轴绘出所有的字段。为了实现这个功能，先要进入绘画（Plot）的视图中，点击齿轮状的按钮 / 图标，然后选择所有字段。请注意，我们可以随后再删除它们或者新建一个轴（在右下角的窗口中）。这个绘图可以用于文件中所有数值，其纵轴表示在回放中的当前位置。



需要注意的是至少要点击一次 Play 按钮才能够对数据进行绘制。我们还能够播放、暂停、停止和移动到文件头或者文件结尾。

这里图像都很简单，而且会出现一个带有当前坐标系和配置选项的窗口，能够将这些内容作为图像文件存储在硬盘上。

3.9 rqt 插件与 rx 应用

从 ROS Fuerte 发布以来，rx 应用或工具已经被弃用了，而我们需要使用 rqt 节点。它们基本上是一样的，只有其中一小部分进行了小的升级、bug 修正和加入了新功能。同时，

它们能在 rqt_gui 的窗口和其他应用中当做插件加载。我们使用以下列表对本章中介绍的工具所对应的节点进行说明：

- rxconsole 被替代为 rosrun rqt_console rqt_console
- rxgraph 被替代为 rosrun rqt_graph rqt_graph
- rxplot 被替代为 rosrun rqt_plot rqt_plot
- rxbag 被替代为 rosrun rqt_bag rqt_bag

更多要说明的是，这些可被单独运行的插件能够更广泛地应用，能够作为命令行（shell）、主题的发布者、消息类型的检查者。甚至 rviz 都有一个名为 rqt_rviz 的插件，还能够被集成在最新的 rqt_gui 界面中。上面的这一切都被集成在 ROS Groovy 和 Hydro 中。在这两版 ROS 中，虽然 rx 工具已经被弃用了，但是仍然保留在软件包中。在 ROS Fuerte 中也是一样的，这个版本是第一个弃用 rqt 工具的版本。

3.10 本章小结

在阅读了本章并运行了示例代码之后，你就能够掌握加快机器人系统建造速度、错误调试、结果可视化的大量实用工具。你可以通过这些工具评估或验证设计质量。作为机器人开发人员，你会在开发过程中逐渐深入体验这些特殊的概念和工具：

- 现在你知道了如何在代码中包含不同级别不同详细程度的日志信息。这能够帮助你调试节点错误。为了实现这个目的，你还需要使用其他一些 ROS 中包含的工具，例如 rxconsole 界面。另外，你也可以检查和列出运行中的节点、发布的主题和系统运行时所提供的服务。这包含了使用 rxgraph 工具检查节点状态图。
- 对于可视化工具，已经能够使用 rxplot 绘制标量数据曲线。它能够以更加直观的方式分析你的节点所发布的特定变量。类似地，你能够查看更加复杂的数据类型（非标量数据）。这包括使用 rviz 查看图像和 3D 数据。
- 最后，你已经通过学习 rosbag 工具学会了记录和回放主题中的消息，还知道了如何使用 rxbag 查看消息回放包文件的内容，这就允许你基于以往的经验去记录数据和使用 AI 算法或机器人算法处理数据。

第 4 章
在 ROS 下使用传感器和执行机构

当你想到机器人，你可能会马上想到人形机器人，有手臂、大量传感器和极为复杂的运动系统。

现在我们已经学习了如何在 ROS 里编写小程序并管理它们。接下来将会学习如何使用传感器和执行机构，这样就能够与真实的世界交互了。

你能够在 http://www.ros.org/wiki/Sensors 找到一个很长的列表清单，上面都是 ROS 支持的设备。

在本章中，我们将会学到：

- 项目中可能会使用的廉价且通用的传感器
- 如 Kinect 之类的 3D 传感器和激光雷达
- 使用 Arduino 连接更多的传感器和执行机构

我们知道在本章中将所有类型的传感器都介绍一遍是不太现实的。基于这个原因，我们会选择那些最通用的和大多数使用者能够负担得起的传感器。

传感器和执行机构被分成不同的类别：激光雷达、摄像头、位姿估计设备等。这样会帮助你更加快捷地找到这些传感器和执行机构。

4.1 使用游戏杆或游戏手柄

我敢大胆地确定你曾经或多或少使用过游戏杆或游戏手柄（joystick/ gamepad）连接到某个视频终端上。

游戏杆只不过是一系列的按钮和电位器。通过这些装置，你能够实现或控制很多种运动模式。

在 ROS 系统中，游戏杆能够通过改变速度和方向来远程控制机器人。

在开始之前，我们将会安装一些功能包。想要安装这些功能包，首先在 Ubuntu 系统中执行以下命令：

```
$ sudo apt-get install ros-fuerte-joystick-drivers
$ sudo apt-get install ros-fuerte-joystick-drivers-tutorials
```

在这些功能包中，你能找到学习如何使用游戏杆的代码和如何建立我们自己的功能包的指导手册。

首先，将游戏手柄连接到你的电脑，检查你的游戏手柄是否能够被识别。

```
$ ls /dev/input/
```

我们会看到以下输出：

```
by-id    event0  event2  event4  event6  event8  js0   mouse0
by-path  event1  event3  event5  event7  event9  mice
```

所创建的端口是 js0；我们能使用 jstest 命令检查它是否工作。

```
$ sudo jstest /dev/input/js0

Axes:  0:  0  1:  0  2:  0 Buttons:  0:off  1:off  2:off  3:off  4:off
5:off  6:off  7:off  8:off  9:off  10:off
```

我们的游戏杆是罗技（Logitech）Attack 3，有 3 个轴向输入和 11 个按钮。如果移动游戏杆，就会产生数值变化。

如果你能确定游戏杆的功能正常，那么我们就直接在 ROS 系统中测试它的功能。为了达到测试的目的，我们需要使用 joy 和 joy_node 功能包。

```
$ rosrun joy joy_node
```

如果所有配置都正确，你会看到：

```
[ INFO] [1357571588.441808789]: Opened joystick: /dev/input/js0.
deadzone_: 0.050000.
```

4.1.1 joy_node 如何发送游戏杆动作消息

如果你的 joy_node 功能包已经可用，我们就来看一下如何使用这个节点发送消息。这会帮助我们理解它是如何发送轴向输入和按钮的信息的。

为了查看节点所发布的消息，我们可以使用以下命令：

```
$ rostopic echo /joy
```

然后我们看到每次消息发出。

```
---
header:
seq: 157
stamp:
  secs: 1357571648
  nsecs: 430257462
  frame_id: ''
```

```
axes: [-0.0, -0.0, 0.0]
buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---
```

你会看到两个主要的向量，一个代表轴向输入，另一个代表了按钮。很明显，这些向量是用于发布实际硬件按钮和轴向输入的状态。

如果你想知道消息的类型，在命令行中输入以下指令：

```
$ rosnode type /joy
```

你会获取消息所使用的类型；在本例中，是 sensor_msgs/ Joy.

现在来查看消息中使用的字段，使用以下命令行：

```
$ rosmsg show sensor_msgs/Joy
```

然后你会看到：

```
  uint32 seq
  time stamp
  string frame_id
float32[] axes
int32[] buttons
```

这就是你在使用游戏杆时将会使用的消息结构。在下一小节中，你将会学到如何写一个订阅游戏杆主题的节点，并生成用于移动小海龟的移动指令。

4.1.2 使用游戏杆数据在 turtlesim 中移动海龟

现在，我们将会创建一个节点通过 joy_node 获取数据并通过发布主题控制 turtlesim。首先，必须要知道我们将要发布的消息所在主题的名称。这样，才能运行 turtlesim 并进行导航。

```
$ rosrun turtlesim turtlesim_node
```

为了看到主题列表，请使用以下命令：

```
$ rostopic list
```

在我们将要使用的 turtle1/command_velocity 主题下，你会看到以下输出：

```
/rosout
/rosout_agg
/turtle1/color_sensor
/turtle1/command_velocity
/turtle1/pose
```

现在我们需要知道主题的类型，请使用以下命令查看：

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

你将会看到这样的输出：

```
turtlesim/Velocity
```

为了知道消息的内容，执行以下命令：

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

你将会看到两个用于发送速度的字段。

```
float32 linear
float32 angular
```

好的，现在我们已经定位了主题和要使用的数据结构，剩下的只是需要花些时间创建程序使用从游戏杆中获取的数据生成速度指令。

在 chapter4_tutorials/src 目录下创建一个新的文件 example.1.cpp，并输入以下代码：

```
#include<ros/ros.h>
#include<turtlesim/Velocity.h>
#include<sensor_msgs/Joy.h>
#include<iostream>

using namespace std;

class TeleopJoy{
public:
  TeleopJoy();
private:
  void callBack(const sensor_msgs::Joy::ConstPtr& joy);
  ros::NodeHandle n;
  ros::Publisher pub;
  ros::Subscriber sub;
  int i_velLinear, i_velAngular;
};

TeleopJoy::TeleopJoy()
{
n.param("axis_linear",i_velLinear,i_velLinear);
n.param("axis_angular",i_velAngular,i_velAngular);
pub = n.advertise<turtlesim::Velocity>("turtle1/command_velocity",1);
sub = n.subscribe<sensor_msgs::Joy>("joy", 10, &TeleopJoy::callBack,
this);
}

void TeleopJoy::callBack(const sensor_msgs::Joy::ConstPtr& joy)
{
turtlesim::Velocity vel;
vel.angular = joy->axes[i_velAngular];
vel.linear = joy->axes[i_velLinear];
pub.publish(vel);
}

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

```
int main(int argc, char** argv)
{
ros::init(argc, argv, "teleopJoy");
    TeleopJoy teleop_turtle;

ros::spin();
}
```

现在我们解释以上代码是如何工作的。

在 main 函数中，我们创建一个 TeleopJoy 类的实例。

```
int main(int argc, char** argv)
{
   ...
   TeleopJoy teleop_turtle;
   ...
```

在构造函数中，初始化了四个变量。前两个变量是使用参数服务器中的数据填充的。这些变量是游戏杆轴向输入。后面两个变量是发布者和订阅者。发布者会使用 turtlesim::Velocity 类型发布主题，这个类型是在 turtlesim 功能包中声明的。订阅者会通过名为 joy 的主题得到数据。处理游戏杆的节点会发布数据到这个主题。

```
TeleopJoy::TeleopJoy()
{
n.param("axis_linear",i_velLinear,i_velLinear);
n.param("axis_angular",i_velAngular,i_velAngular);

pub = n.advertise<turtlesim::Velocity>("command_velocity",1);
sub = n.subscribe<sensor_msgs::Joy>("joy", 10, &TeleopJoy::callBack,
this);

}
```

每一次节点都会收到一个消息，调用 callBack 函数。我们使用 vel 为名称创建一个新的用于发布数据的变量。游戏杆轴向输入的变量被赋给 vel 变量。在这部分里，你能够在发布数据之前对数据进行一定的预处理。

```
void TeleopJoy::callBack(const sensor_msgs::Joy::ConstPtr& joy)
{
turtlesim::Velocity vel;
vel.angular = joy->axes[i_velAngular];
vel.linear = joy->axes[i_velLinear];
pub.publish(vel);
}
```

最后，通过调用 pub.publish(vel) 发布主题。

我们将会为这个示例创建一个 launch 文件。在 launch 文件中，为参数服务器声明一些变量并启动 joy 和 example1 节点。

将以下的代码复制到 chapter4_ tutorials/src 文件夹下的一个新文件 example1.launch 中：

```
<launch>

<node pkg="turtlesim" type="turtlesim_node" name="sim"/>
<node pkg="chapter4_tutorials" type="example1" name="example1" />
<param name="axis_linear" value="1" type="int" />
<param name="axis_angular" value="0" type="int" />

<node respawn="true" pkg="joy"type="joy" name="teleopJoy">
    <param name="dev" type="string" value="/dev/input/js0" />
    <param name="deadzone" value="0.12" />
</node>

</launch>
```

你可能会注意到在 launch 文件中，有三个不同的节点：example1、sim 和 joy。

在 launch 文件中有四个参数，这些参数会向参数服务器添加数据，并被我们的节点使用。axis_linear 和 axis_angular 两个参数将会用于配置游戏杆的轴向输入。如果你想修改轴向输入的配置，你只需要修改这两个变量并指明你想修改哪一个轴向输入。dev 和 deadzone 参数分别用于配置游戏杆连接的端口和不能被设备识别的运动区域。

要运行 launch 文件，请使用下面的命令行：

```
$ roslaunch chapter4_tutorials example1.launch
```

如果一切顺利，你就能通过 rosnode list 和 rostopic list 命令检查正在运行的节点和主题列表。如果你想以图形化的方式查看，则需要使用 rxgraph。

4.2 使用激光雷达——Hokuyo URG-04lx

在移动机器人中，能够获取障碍物的具体位置、房间的内部轮廓等都是非常重要的。机器人创建并使用地图进行导航和移动通过未知的区域。使用激光雷达（Lidar）正是为了实现这个目的。这个传感器专门用于测量机器人和物体之间的距离。

在本节中，你将会看到如何使用在机器人领域内广泛应用的低成本激光雷达。本章的传感器选用 Hokuyo URG-04lx 激光雷达。你可以通过以下链接获取更加详细的信息：http://www.hokuyo-aut.jp/。Hokuyo 激光雷达是能够实现机器人导航和实时地图创建的设备。

在本节中，你会首先学习如何在 ROS 中调用标准驱动来使用它和如何修改数据。

标准的 Hokuyo URG-04lx 是机器人领域最常用的低成本激光雷达。它具有极佳的分辨率而且非常容易上手。要想使用它首先需要安装激光雷达的驱动。

```
$ sudo apt-get install ros-fuerte-laser-drivers
```

一旦驱动安装完毕，就应检查是否安装正确。连上你的激光雷达然后检查是否系统能够正确地检测到它，和它是否被正确地配置。

```
$ ls -l /dev/ttyACM0
```

当我们连上激光雷达，系统能够通过以下命令查看它：

```
crw-rw---- 1 root dialout 166, 0 Jan 13 11:09 /dev/ttyACM0
```

在我们的示例中，需要重新配置激光雷达以便 ROS 能够访问和使用它，也就是说我们需要获取适当的访问权限。

```
$ sudo chmod a+rw /dev/ttyACM0
```

你将会看到以下输出：

```
crw-rw-rw- 1 root dialout 166, 0 Jan 13 11:09 /dev/ttyACM0
```

如果一切都没问题，我们将会转换到激光雷达上。在一个命令行窗口中启动 roscore，在另一个命令行窗口中执行以下命令：

```
$ rosrun hokuyo_node hokuyo_node
```

如果一切正常，你会看到以下输出，：

```
[ INFO] [1358076340.184643618]: Connected to device with ID: H1000484
```

4.2.1 了解激光雷达如何在 ROS 中发送数据

想要检查节点是否在发送数据，需要使用 rostopic。

```
$ rostopic list
```

然后你会看到以下主题：

```
/diagnostics
/hokuyo_node/parameter_descriptions
/hokuyo_node/parameter_updates
/rosout
/rosout_agg
/scan
```

/scan 是节点正在发布消息的主题。节点所使用的数据类型如下所示：

```
$ rostopic type /scan
```

你会看到用于发送激光雷达信息的消息类型：

```
sensor_msgs/LaserScan
```

你能使用以下命令看到消息的数据结构：

```
$ rosmsg show sensor_msgs/ LaserScan.
```

如果想了解更多关于激光雷达工作和数据发送的信息，可以使用 rostopic 命令来查看实时消息。

```
$ rostopic echo /scan
```

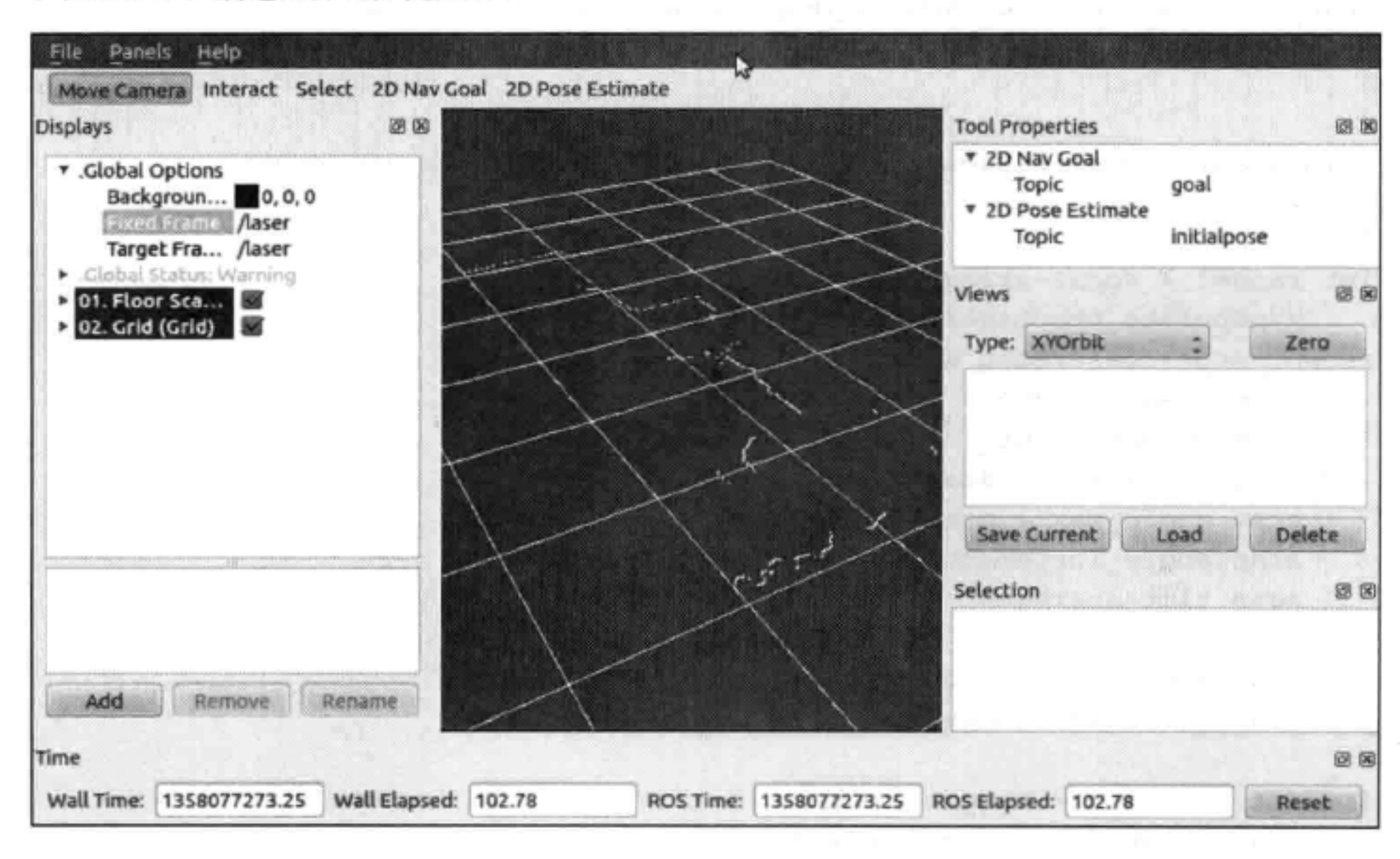
然后你会看到激光雷达所发出的消息如下：

```
---
header:
seq: 3895
stamp:
  secs: 1358076731
  nsecs: 284896750
  frame_id: laser
...
ranges: [1.1119999885559082, 1.1119999885559082, 1.1109999418258667, ...]
intensities: []
---
```

这些数据很难被人理解。如果你想更好地去理解数据的含义，最好的办法是使用 rviz 在图形化界面下展示数据。在一个命令行窗口下输入以下命令启动 rviz，并加载正确的配置文件：

```
$ rosrun rviz rviz -d 'rospack find hokuyo_node'/hokuyo_test.vcg
```

下图显示了消息的图形化展示：



你能够从屏幕上看出轮廓。如果移动激光雷达，你会看到轮廓持续变化。

4.2.2 访问和修改激光雷达数据

现在，我们将会创建一个节点获取雷达数据，并使用数据做些工作，然后发布信息的数据。这有时候会很有用，而等你学完这个示例，你就会知道如何使用它。

在 /chapter4_ tutorials/src 文件夹下将以下代码复制到 example2.cpp 文件中：

```
#include <ros/ros.h>
#include "std_msgs/String.h"
#include <sensor_msgs/LaserScan.h>
#include<stdio.h>

using namespace std;
class Scan2{
public:
  Scan2();
private:
  ros::NodeHandle n;
    ros::Publisher scan_pub;
    ros::Subscriber scan_sub;
  void scanCallBack(const sensor_msgs::LaserScan::ConstPtr& scan2);
};

Scan2::Scan2()
{
scan_pub = n.advertise<sensor_msgs::LaserScan>("/scan2",1);
scan_sub = n.subscribe<sensor_msgs::LaserScan>("/scan",1,
&Scan2::scanCallBack, this);
}

void Scan2::scanCallBack(const sensor_msgs::LaserScan::ConstPtr&
scan2)
{
int ranges = scan2->ranges.size();
    //populate the LaserScan message
sensor_msgs::LaserScan scan;
    scan.header.stamp = scan2->header.stamp;
    scan.header.frame_id = scan2->header.frame_id;
    scan.angle_min = scan2->angle_min;
    scan.angle_max = scan2->angle_max;
    scan.angle_increment = scan2->angle_increment;
    scan.time_increment = scan2->time_increment;
scan.range_min = 0.0;
    scan.range_max = 100.0;
scan.ranges.resize(ranges);

for(int i = 0; i < ranges; ++i){
    scan.ranges[i] = scan2->ranges[i] + 1;
```

```
    }
scan_pub.publish(scan);
}

int main(int argc, char** argv){
ros::init(argc, argv, "example2_laser_scan_publisher");
    Scan2 scan2;
ros::spin();
}
```

让我们深入代码看一看到底发生了什么。

在 main 函数中，初始化了一个名为 example2_laser_scan_publisher 的节点，然后创建了之前在文件中已经定义好的类的一个实例。

在构造函数中，创建了两个主题：一个主题将会订阅另一个。第二个主题具有激光雷达的原始数据，并将会对原始数据进行修改然后再发布。

这个例子非常简单，我们只对来自雷达主题的数据加 1 然后重新发布它。我们在 scanCallBack() 函数中实现这个功能。获取输入消息并将所有字段都复制到另一个变量中，再获取存储数据的字段并全部加 1。一旦新的值存储完毕，就将其发布到新的主题中。

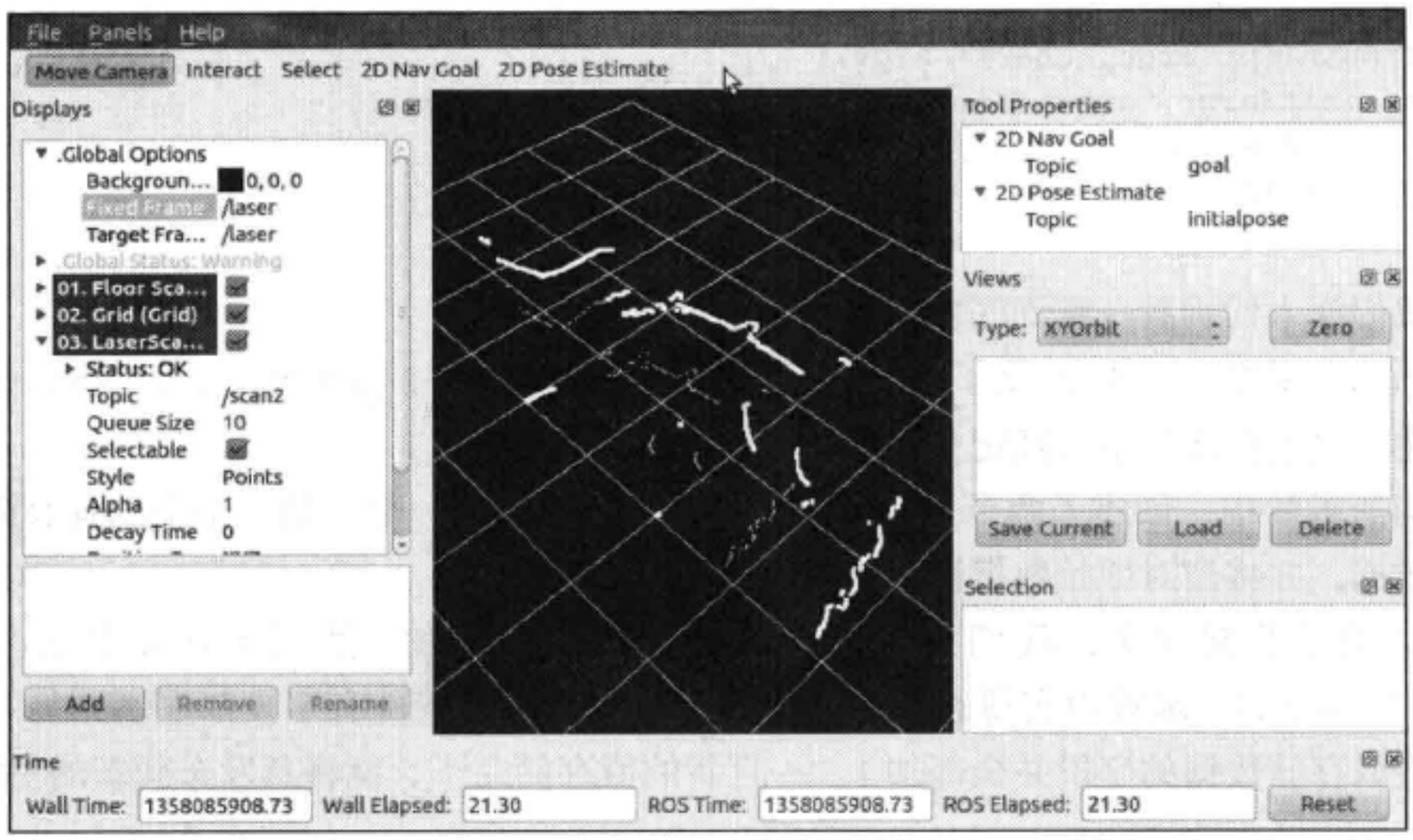
```
void Scan2::scanCallBack(const sensor_msgs::LaserScan::ConstPtr&
scan2)
{
    ...
sensor_msgs::LaserScan scan;
    scan.header.stamp = scan2->header.stamp;
 ...
 ...
    scan.range_max = 100.0;
scan.ranges.resize(ranges);

for(int i = 0; i < ranges; ++i){
    scan.ranges[i] = scan2->ranges[i] + 1;
    }

scan_pub.publish(scan);
}
```

创建启动文件

为了启动这一切，需要创建一个 launche 文件 example2.launch。

```
<launch>
  <node pkg="hokuyo_node" type="hokuyo_node" name="hokuyo_node"/>
  <node pkg="rviz" type="rviz" name="rviz"
  args="-d $(find chapter4_tutorials)/example2.vcg"/>

  <node pkg="chapter4_tutorials" type="example2" name="example2" />
</launch>
```

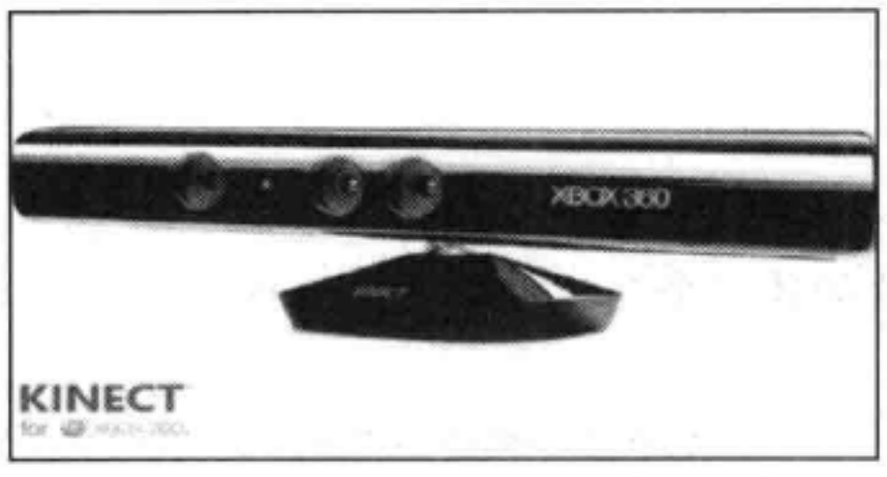
现在如果你启动 example2.launch 文件，有三个节点会启动：hokuyo_node、rviz 和 example2。你将会在 rviz 的屏幕内看到两条激光轮廓。绿颜色的轮廓是新的数据。



4.3 使用 Kinect 传感器查看 3D 环境

Kinect 传感器是一个扁平的小黑盒子，其下方是一个可以活动的小平台，能够固定在桌子上或者是电视机的架子上（它是 Xbox360 的体感游戏套件）。这个设备上面有三种设备能够帮助我们完成机器视觉和机器人任务：

- 一个彩色 VGA 视频摄像头来查看彩色的世界。
- 一个深度传感器，包括一个红外色斑投影和一个单色 CMOS 传感器配合工作，获取物体的深度信息并转换为 3D 数据。
- 用于分离使用者的声音和室内噪声的麦克风阵列。



在 ROS 系统中，我们将使用这两类传感器：RGB 摄像头和深度传感器。而在最新版本的 ROS 中，你甚至能够用三种传感器。

在开始使用它之前，我们需要安装功能包和驱动。使用下面的命令行来安装它：

```
$ sudo apt-get install ros-fuerte-openni-camera ros-fuerte-openni-launch
$ rosstack profile
$ rospackage profile
```

一旦安装完成，插入 Kinect 传感器，我们就能运行节点并开始使用它。在命令行中，启动 roscore。在另外一个命令行窗口中运行以下指令：

```
$ rosrun openni_camera openni_node
$ roslaunch openni_launch openni.launch
```

如果一切正常，你就不会看到任何错误信息产生。

4.3.1 如何发送和查看 Kinect 数据

现在试一下我们能够使用这些节点做些什么样的事情。使用以下命令列出你正运行的主题：

```
$ rostopic list
```

然后，你会看到很多个主题，但是对于我们来说最重要的就是下面这几个：

```
...
/camera/rgb/image_color
/camera/rgb/image_mono
/camera/rgb/image_raw
/camera/rgb/image_rect
/camera/rgb/image_rect_color
...
```

你将会看到节点创建了很多个主题。如果你想要查看某一个传感器，例如 RGB 摄像头，可以使用主题 /camera/rgb/image_color。想要查看传感器获取的图像，我们将会使用 image_view 功能包。在一个命令行窗口中运行以下指令：

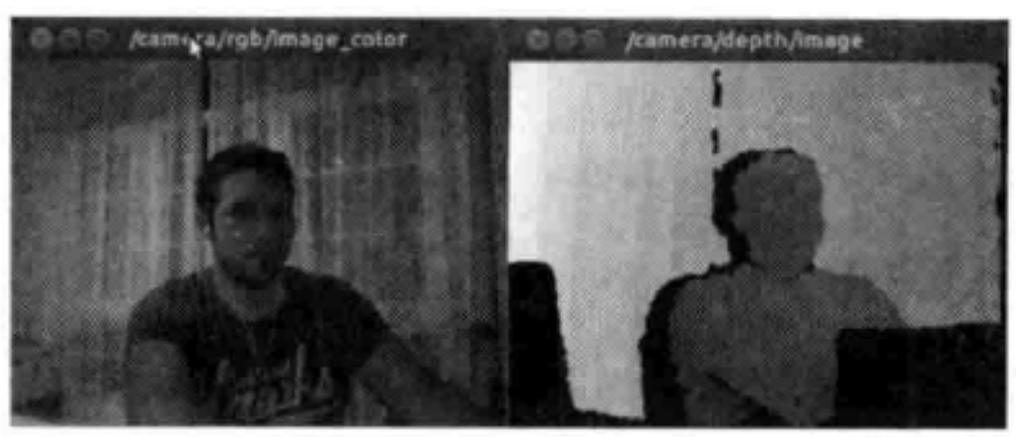
```
$ rosrun image_view image_view image:=/camera/rgb/image_color
```

注意到我们需要将图像主题重命名（重映射）为 /camera/rgb/image_ color。如果一切正常，会有一个新的窗口弹出并显示来自于 Kinect 的图像。

如果你想查看深度传感器，同样只需要在命令行中将主题名称改变一下：

```
$ rosrun image_view image_view image:=/camera/depth/image
```

你会看到一个和彩色图像非常相似的窗口，如下显示：



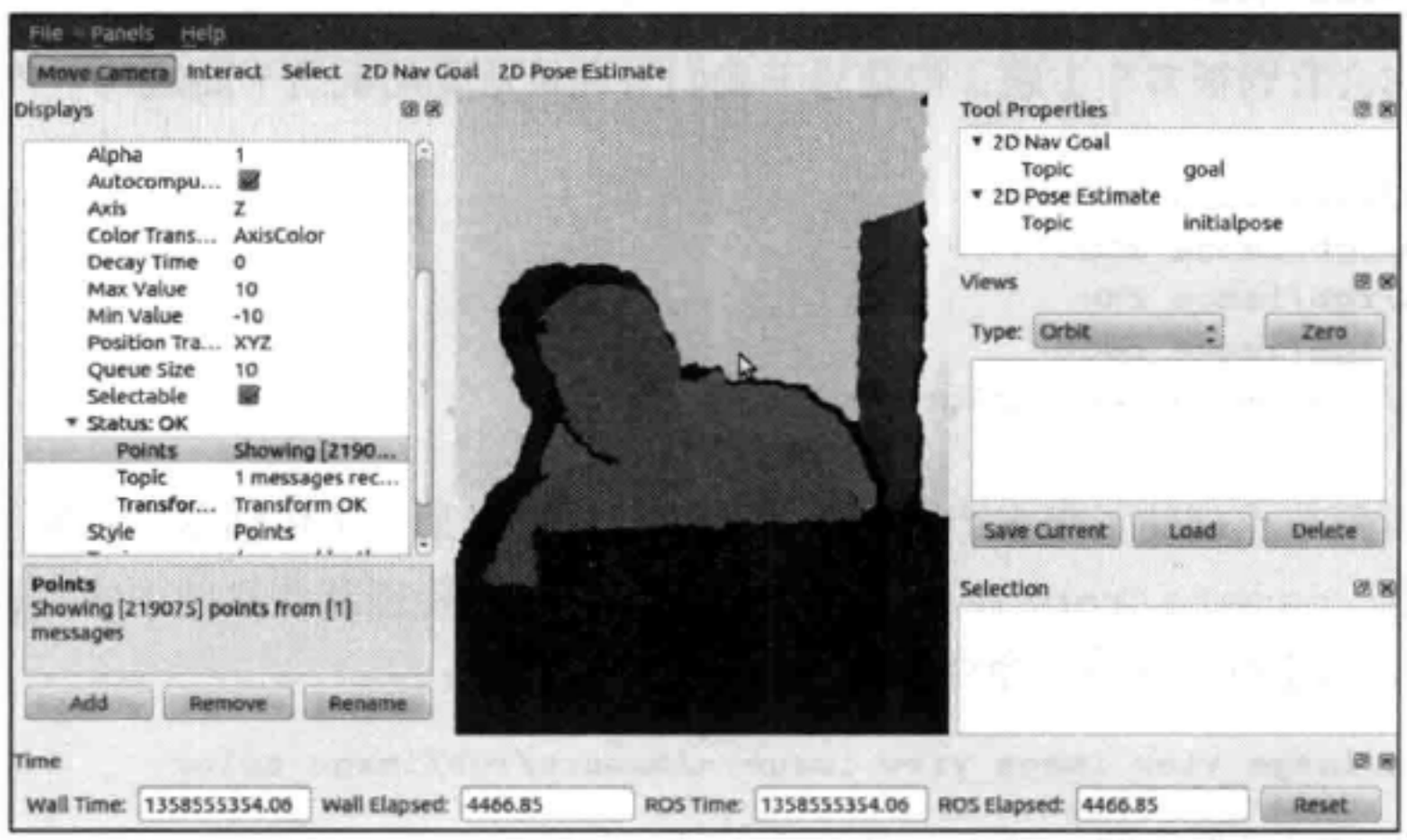
另一个重要的主题是传递点云数据的主题。这种数据是一种使用了 3D 展示形式的深度图像。你可以在以下主题中找到此类数据：/camera/depth/points、/camera/depth_registered/points 等。

我们可以查看这种消息的具体类型，使用 rostopic type 命令就可以实现。如果想查看消息的具体字段，我们能使用 rostopic type /topic_name | rosmsg show。在本示例中，我们将会使用 /camera/depth/points 主题。

```
$ rostopic type /camera/depth/points | rosmsg show
```

如果想查看消息的官方说明，请访问 http://ros.org/doc/api/sensor_msgs/ html/msg/PointCloud2.html。

如果想实现此类数据的可视化，那么在一个新的命令行窗口中运行 rviz，并添加一个新的 PointCloud2 数据可视化工具。你能否实时地看到 3D 图像还取决于你电脑的速度。如果你在传感器前移动，你将会看到自己在 3D 环境中移动。如下图所示：



4.3.2 创建和使用 Kinect 示例

现在我们将会使用一段程序来生成一个节点，过滤来自 Kinect 传感器的点云数据。这个节点将会应用一个过滤器来减小原始数据中点云中点的数量。这会减少采样的数据。

在 chapter4_tutorials/src 文件夹下创建一个新文件 example3.cpp，并输入以下代码段：

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/ros/conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
```

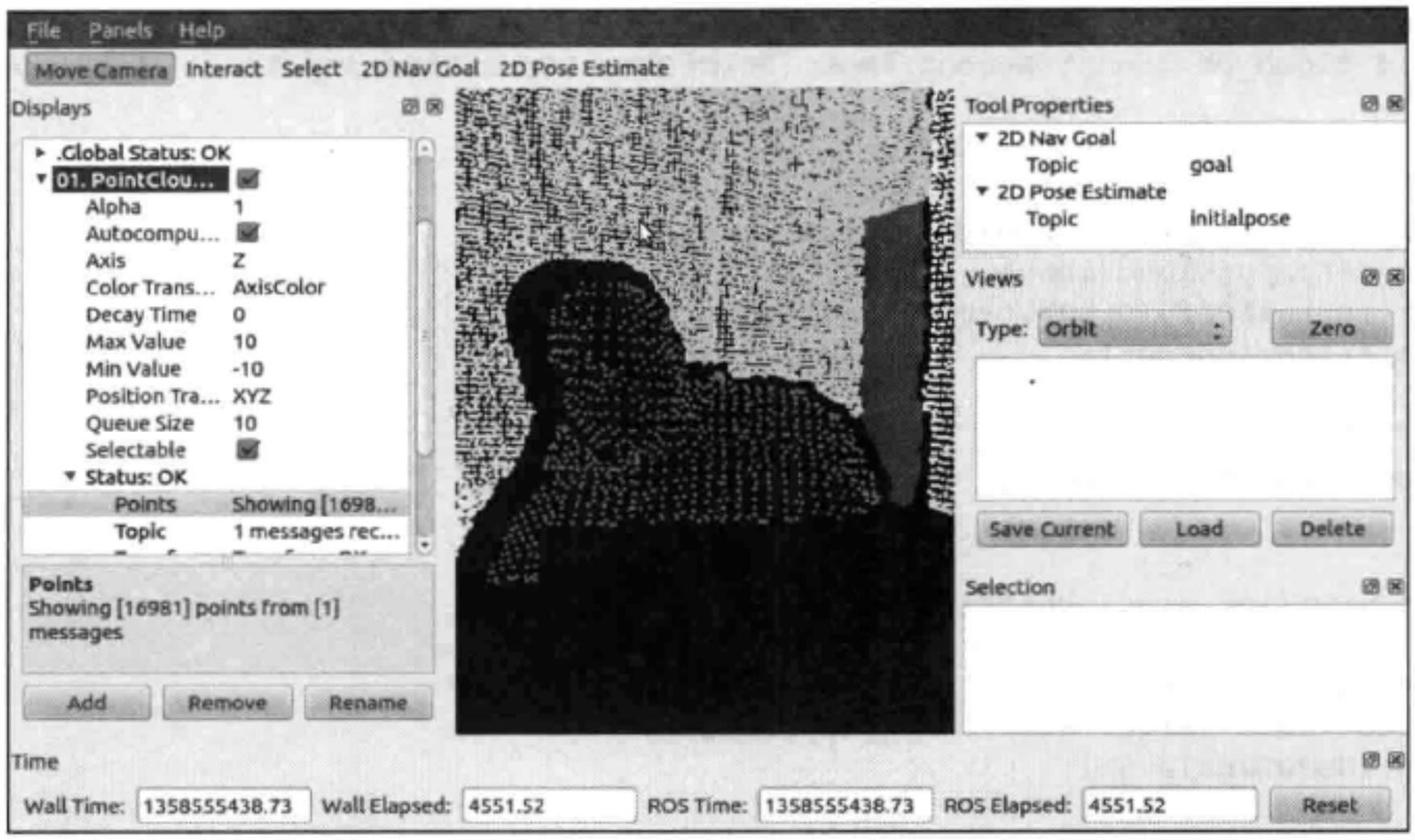
```
ros::Publisher pub;
void cloud_cb (const sensor_msgs::PointCloud2ConstPtr& input)
{
  // ... do data processing
sensor_msgs::PointCloud2 output;
pcl::VoxelGrid<sensor_msgs::PointCloud2> sor;
sor.setInputCloud(input);
sor.setLeafSize(0.02f,0.02f,0.02f);
sor.filter(output);

  // Publish the data
  pub.publish (output);
}

int main(int argc, char** argv)
{
  // Initialize ROS
  ros::init (argc, argv, "my_pcl_tutorial");
ros::NodeHandle nh;

  // Create a ROS subscriber for the input point cloud
ros::Subscriber sub = nh.subscribe ("/camera/depth/points", 1, cloud_
cb);

  // Create a ROS publisher for the output point cloud
pub = nh.advertise<sensor_msgs::PointCloud2> ("output", 1);

  // Spin
  ros::spin ();
}
```

本示例是基于点云库(PCL)的教程。你可以通过此链接查看 http://point clouds.org/documentation/tutorials/voxel_grid. php#voxelgrid。

所有的工作都是在 cb() 函数中完成，每当收到消息都会调用这个函数。我们创建一个 VoxelGrid 类型的变量 sor，在 sor.setLeafSize() 中改变网格的大小。网格是用于过滤器的参数。当你增加这些值的大小，你会获得更低的分辨率，点云上的点也会越少。

```
cloud_cb (const sensor_msgs::PointCloud2ConstPtr& input)
{
...
  pcl::VoxelGrid<sensor_msgs::PointCloud2> sor;
   ...
   sor.setLeafSize(0.02f,0.02f,0.02f);
 ...
 }
```

如果运行一个新的节点来打开 rviz，会看到新的点云呈现在窗口中，而你会很容易发现分辨率比原有的数据低了不少。



在 rviz 中，你能够看到一个消息中包含的点的数量。对于原始数据，我们能看到点的数量是 219 075；而新的点云中，数量只有 16 981。可见数据有极大的减少。

在 http://pointclouds.org/ 中，你能找到更多过滤器和教程，并能够学习如何使用此类数据。

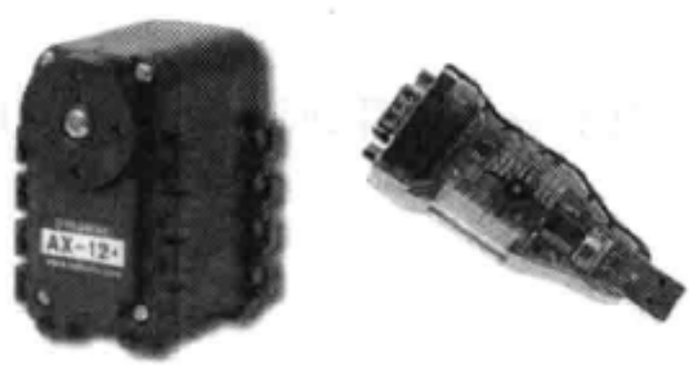
4.4 使用伺服电动机——Dynamixel

在移动机器人领域里，伺服电动机可谓是广泛使用。此类执行机构是用于移动传感器、轮子和机械臂。其中有一种低成本解决方案是使用 RC 伺服电机（舵机）。它能够在 180° 的范围内运动，并且提供较大的扭矩。

本节中将会介绍的伺服电机是一种专为机器人设计和使用的新型伺服电机，这就是 Dynamixel 伺服电机。

Dynamixel 是由韩国生产商 ROBOTIS 公司开发的一种机器人专用的线性、高性能网络执行机构。ROBOTIS 公司还是 OLLO、Bioloid 和 DARwIn-OP DXL 等机器人的设备研发和供应商。这些设备被众多的企业、大学和爱好者所使用，由于它们极佳的可扩展性、强大的反馈能力、位置、速度、内部温度、输入电压和其他特性，还有其总线型拓扑结构大大简化了接线方式。

在下面的图片中，你会看到 Dynamixel AX-12 和 USB 接口。它们都用于这个示例。



首先，我们将会安装必需的功能包和驱动。在命令行窗口输入以下命令：

```
$ sudo apt-get install ros-fuerte-dynamixel-motor
```

一旦安装完毕，将转换器连接到计算机并检查能否被检测到。通常情况下，它会在 /dev/ 文件夹下创建一个新的以 ttyUSBX 为名的接口。如果你看到这个接口，那么一切就绪，就让我们启动节点并试一试这个伺服电机吧。

在命令行窗口中启动 roscore，然后在另外一个窗口中输入以下命令：

```
$ roslaunch dynamixel_tutorials controller_manager.launch
```

如果你连接了一个或多个电机，那么你会看到电机被驱动检测到。

在我们的示例中，检测到了一个名为 ID 6 的电机，并进行初始化配置。

```
process[dynamixel_manager-1]: started with pid [3966]
[INFO] [WallTime: 1359377042.681841] pan_tilt_port: Pinging motor IDs 1
through 25...
[INFO] [WallTime: 1359377044.846779] pan_tilt_port: Found 1 motors - 1
AX-12 [6], initialization complete.
```

4.4.1 Dynamixel 如何发送和接收运动命令

一旦启动了 controller_manager.launch 文件，就可以查看主题的列表。记住请使用以下命令行查看主题：

```
$ rostopic list
```

这些主题会显示配置好的电机状态。

```
/diagnostics
/motor_states/pan_tilt_port
/rosout
/rosout_agg
```

如果使用 rostopic echo 命令查看 /motor_states/pan_tilt_port，你会看到所有电机的状态。在我们的示例中，仅仅有一个 ID 6 的电机。然而，我们不能使用这些主题来驱动电机，所以我们需要加载另一个 launch 文件来做这些工作。

这个 launch 文件会创建驱动电机所必要的主题。

```
$ roslaunch dynamixel_tutorials controller_spawner.launch
```

这时，在主题的清单中会出现两个新的主题，如下所示，其中一个新主题将会用来驱动电机：

```
/diagnostics
/motor_states/pan_tilt_port
/rosout
```

```
/rosout_agg
/tilt_controller/command
/tilt_controller/state
```

我们将会使用 /tilt_controller/command 通过 rostopic pub 命令发布主题来驱动电机。首先查看主题的字段和类型，需要输入以下命令行：

```
$ rostopic type /tilt_controller/command

std_msgs/Float64
```

正如你所见，这是一个 Float64 类型的变量。这些变量通过一个以弧度表示的位置指令来驱动电机。所以，通过如下指令发布主题：

```
$ rostopic pub /tilt_controller/command std_msgs/Float64 -- 0.5
```

一旦命令下发出去，你会看到电机转动，但是它会停止在 0.5 rad 或 28.647 889 8° 上。

4.4.2 创建和使用伺服电动机示例

现在，我们将会向你展示如何使用节点来驱动电机。在你的 /chapter4_tutorials/src 目录下创建一个新文件 example4.cpp，并输入以下代码：

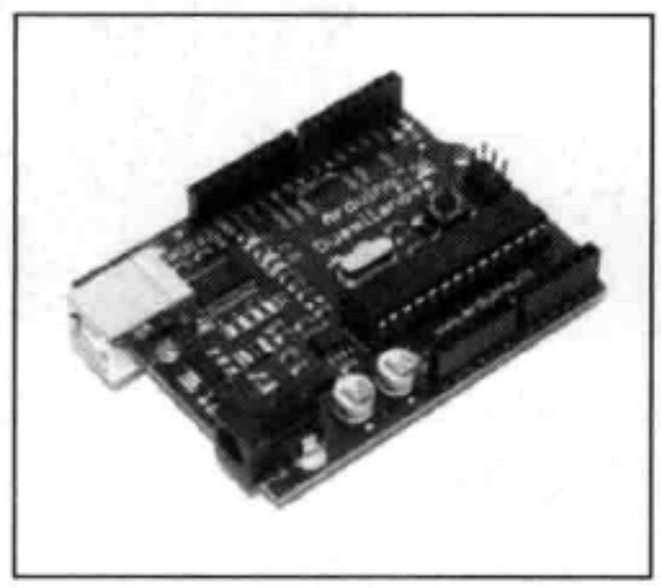
```
#include<ros/ros.h>
#include<std_msgs/Float64.h>
#include<stdio.h>

using namespace std;

class Dynamixel{
  private:
    ros::NodeHandle n;
    ros::Publisher pub_n;
  public:
    Dynamixel();
    int moveMotor(double position);
};

Dynamixel::Dynamixel(){
  pub_n = n.advertise<std_msgs::Float64>("/tilt_controller/
command",1);
}
int Dynamixel::moveMotor(double position)
{
  std_msgs::Float64 aux;
  aux.data = position;
  pub_n.publish(aux);
  return 1;
}

int main(int argc,char** argv)
{
  ros::init(argc, argv, "example4_move_motor");
```

```
Dynamixel motors;

float counter = -180;
ros::Rate loop_rate(100);

while(ros::ok())
{
  if(counter < 180)
  {
    motors.moveMotor(counter*3.14/180);
    counter++;
  }else{
    counter = -180;
  }
  loop_rate.sleep();
}
}
```

这个节点会驱动电机在 −180° ～ 180° 之间不断运动。这是一个非常简单的例子，但你能够使用它进行复杂的运动或者控制更多的电机。我们假设你能够理解这些代码，所以没有必要进行过多的解释。请注意你是在向 /tilt_controller/command 主题发布数据，这就是电机的名称。

4.5 使用 Arduino 添加更多的传感器和执行机构

Arduino 是一种基于柔性的易于开发的软件和硬件组成的开源电子原型设计平台。它服务于艺术家、设计人员、爱好者和所有志愿于创造交互式物体和环境的任何人。



ROS 能通过 rosserial 功能包使用此类设备。基本上，Arduino 使用串口和电脑连接，数据传递也是通过串口。你可以通过 rosserial 使用很多串口连接的设备，例如 GPS、伺服控制器等各种设备。

首先，我们需要安装功能包。输入以下命令：

```
$ sudo apt-get install ros-fuerte-rosserial
$ rosstack profile
$ rospackage profile
```

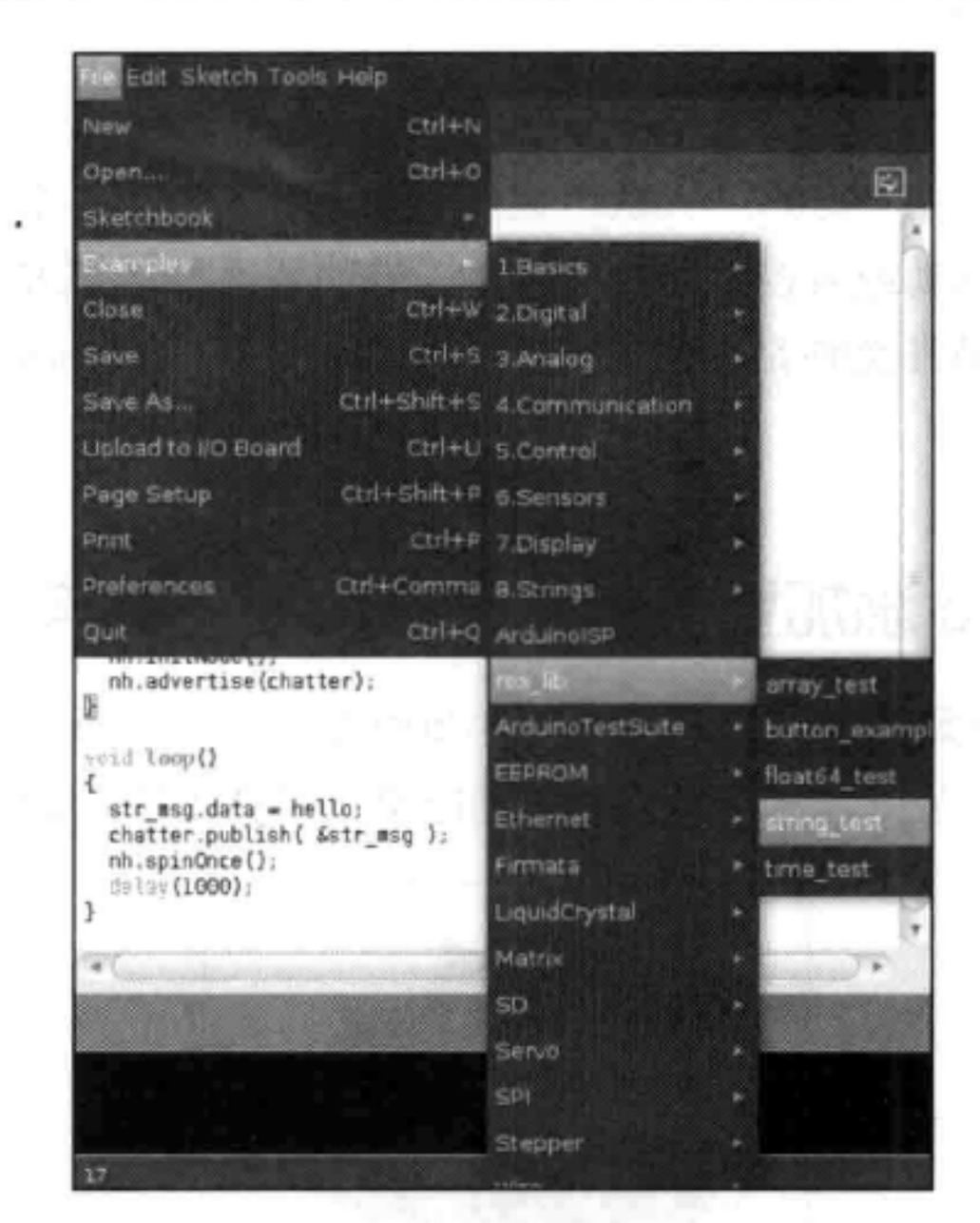
好的，假设你已经安装了 Arduino IDE 。如果你还没有，请根据 http://arduino.cc/en/Main/Software 描述的步骤进行安装。

一旦你将所有功能包和 IDE 都安装好了，就能够从 rosserial 功能包复制 ros_lib 到 sketchbook 库文件夹下。这个库文件夹会在你的电脑运行 Arduino IDE 之后创建。

```
$ roscd rosserial_arduino/libraries
$ cp -r ros_lib <sketchbook>/libraries/ros_lib
```

创建和使用 Arduino 示例

现在我们将通过 IED 上传示例代码到 Arduino 。选择 Hello World 示例并上传 Sketch。



这些代码和下面的代码都很类似。在这些代码中，你能看到在 include 行中包含了 ros.h 头文件。它就是我们之前安装的 rosserial 库。还有，你能够查看一个库通过主题所发送的消息，在本例中，是 std_msgs/String 类型。

以下代码都保存在 example5_1.ino 文件中：

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

ros::NodeHandle  nh;

std_msgs::String str_msg;
ros::Publisher chatter("chatter", &str_msg);
```

```
char hello[19] = "chapter4_tutorials";

void setup()
{
  nh.initNode();
  nh.advertise(chatter);
}

void loop()
{
  str_msg.data = hello;
  chatter.publish( &str_msg );
  nh.spinOnce();
delay(1000);
}
```

在 setup() 函数中，设定了主题的名称。本示例中名称是 chatter。现在我们需要启动一个节点监听端口并通过 Arduino 向 ROS 网络中发布主题。在命令行窗口中输入以下命令，记住之前还要运行 roscore。

```
$ rosrun rosserial_python serial_node.py /dev/ttyUSB0
```

现在你可以通过 rostopic echo 命令看到 Arduino 发出的消息。

```
$ rostopic echo chatter
```

你会在命令行窗口中看到以下数据：

```
data: chapter4_tutorials
```

最后的例子是从 Arduino 向计算机发送数据。现在我们将会使用一个例子，其中 Arduino 会订阅一个主题并改变连接到 PIN 13 上的 LED 灯的状态。示例的名称是 blink，你可以从 Arduino IDE 的标题栏上直接查找 File | Examples | ros_lib | Blink。

下面的代码片段已经保存在 example5_2.ino 文件中：

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

ros::NodeHandle nh;
void messageCb( const std_msgs::Empty& toggle_msg){
digitalWrite(13, HIGH-digitalRead(13));   // blink the led
}

ros::Subscriber<std_msgs::Empty> sub("toggle_led", &messageCb );

void setup()
{
  pinMode(13, OUTPUT);
  nh.initNode();
  nh.subscribe(sub);
```

```
}

void loop()
{
  nh.spinOnce();
  delay(1);
}
```

记住要启动节点与 Arduino 板进行通信。

```
$ rosrun rosserial_python serial_node.py /dev/ttyUSB0
```

现在，如果你想改变 LED 灯的状态，可以使用 `rostopic pub` 命令发布一个新的状态。

```
$ rostopic pub toggle_led std_msgs/Empty -once

publishing and latching message for 3.0 seconds
```

你会发现 LED 改变了状态。如果 LED 已经点亮，那么它会灭掉。为了再次改变状态，你需要向主题再次发布消息。

```
$ rostopic pub toggle_led std_msgs/Empty --once

publishing and latching message for 3.0 seconds
```

现在你能够使用 ROS 中所有支持 Arduino 的设备了。这是一项非常有用的功能，因为你可以使用各种廉价的传感器和执行机构装备你的机器人了。

当我们开始编写本章的时候，我们发现有些 Arduino 设备不支持 `rosserial`，例如 Arduino Leonardo 等。因此，在使用本功能包之前，最好小心你所选的设备。我们在使用 Arduino Mega、Arduino Duemilanove、Arduino Nano 的时候，没有出过任何问题。

4.6 使用惯性测量模组——Xsens MTi

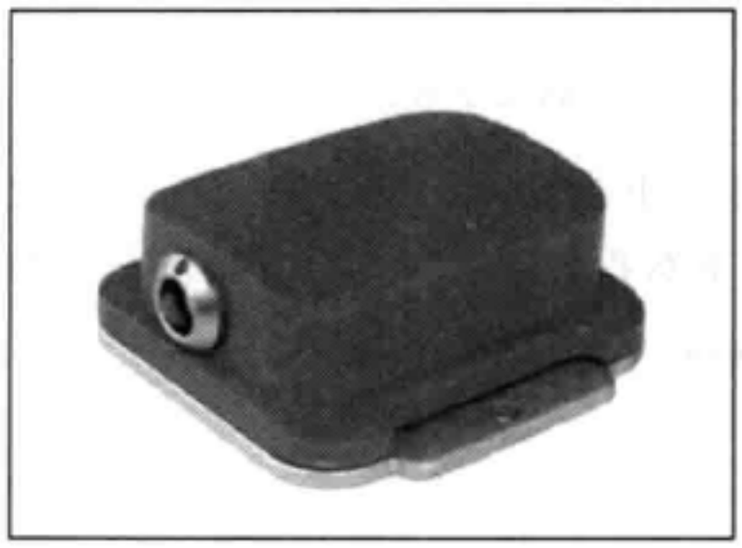
惯性测量模组（IMU）是一种用于测量和报告被测设备速度、方向和重力的电子设备，它能将加速度计和陀螺仪，甚至是磁场强度计等传感器的数据进行综合。

IMU 最典型的应用是从人工飞行器包括各种无人飞机（UAV），一直到空间飞船包括各种卫星和登陆飞船。

——来源：维基百科英文版

在下面的图片中，你会看到 Xsens MTi，它是本节的主角。

在本节中，我们将会学习在 ROS 中 Xsens MTi 的工作方式，通过传感器发布的主题内容和一小段代码示例来说明如何从传感器取得数据并发布一个新主题。



你能够在 ROS 系统中使用多种 IMU。本节中，我们将使用 Xsens IMU。它必须正确地安装驱动才能正常工作。但如果你想使用 MicroStrain 3DM-GX2 或者带有 Wii Motion Plus 的 Wiimote，你就需要下载以下驱动：

- MicroStrain 3DM-GX2 IMU 可使用 http://www.ros.org/wiki/ microstrain_3dmgx2_imu。
- Wiimote with Wii Motion Plus 可使用 http://www.ros.org/wiki/ wiimote。

为了使用我们的设备，我们将会使用 `lse_xsens_mti` 驱动。你能够通过以下命令安装：

```
$ svn co http://isr-uc-ros-pkg.googlecode.com/svn/stacks/lse_imu_drivers/
trunk/
```

我们需要同时安装两个功能包，因为所下载的驱动需要依靠它们运行。

```
$ svn co http://isr-uc-ros-pkg.googlecode.com/svn/stacks/serial_
communication/trunk/cereal_port/
$ sudo apt-get install ros-fuerte-gps-umd
```

要记得把它们安装到 `ROS_PACKAGE_PATH` 路径下。一旦安装完毕，刷新功能包和功能包集的安装记录。

```
$ rosstack profile
$ rospack profile
```

现在，我们将会开始使用 IMU 并学习它是如何工作的。在一个命令行窗口中启动 `roscore`，然后在另一个窗口中输入以下命令：

```
$ rosrun lse_imu_drivers mti_node
```

确定设备已经连接到了 `/dev/ttyUSB0` 端口上，否则，你必须通过代码更改端口或者像我们在其他章节中介绍的一样使用参数服务器。

4.6.1 Xsens 如何在 ROS 中发送数据

如果一切正常，你可以通过 `rostopic` 命令查看主题的列表：

```
$ rostopic list
```

这个节点将会发布三个主题。在本节中，我们将会使用 `/imu/data` 主题。首先，我们看一下所发布主题的类型和数据。想要查看类型和字段，需要使用以下命令行：

```
$ rostopic type /imu/data
$ rostopic type /imu/data | rosmsg show
```

这些字段是用来指明方向、加速度和速度。在我们的示例中，将会使用 orientation 字段。先查看一个消息学习一下数据发送的实例。你可以通过以下命令来查看：

```
$ rostopic echo /imu/data

---
header:
seq: 288
stamp:
  secs: 1330631562
  nsecs: 789304161
frame_id: xsens_mti_imu
orientation:
x: 0.00401890464127
y: -0.00402884092182
z: 0.679586052895
w: 0.73357373476
...
---
```

如果你仔细观察方向字段，就会看到四个变量，而不是像大家通常认为的三个。这是因为在 ROS 系统中，特殊的方向需要有四元数来表征。你能够从网上找到很多关于这种简明又无歧义的方向表达方式的文献。

4.6.2 创建和使用 Xsens 示例

现在我们知道了发送的数据类型和将使用哪些数据，那么让我们开始下面的示例。

在本示例中，我们将会使用 IMU 来移动 turtlesim。要做到这一点，需要从四元数中取出数据并将其转换为欧拉角（滚动、俯仰和偏航）并以在 x 轴和 y 轴的旋转值（滚动和俯仰）作为让小海龟进行线性和角度运动的值。

下面的代码和游戏杆的代码非常接近。在 chapter4_tutorials/src/ 文件夹下创建一个新的文件 example6.cpp，并输入以下代码：

```
#include<ros/ros.h>
#include<turtlesim/Velocity.h>
#include<sensor_msgs/Imu.h>
#include<iostream>
#include<tf/LinearMath/Matrix3x3.h>
#include<tf/LinearMath/Quaternion.h>

using namespace std;

class TeleopImu{
public:
  TeleopImu();
private:
  void callBack(const sensor_msgs::Imu::ConstPtr& imu);
```

```
  ros::NodeHandle n;
  ros::Publisher pub;
  ros::Subscriber sub;
};

TeleopImu::TeleopImu()
{
  pub = n.advertise<turtlesim::Velocity>("turtle1/command_velocity",1);
  sub = n.subscribe<sensor_msgs::Imu>("imu/data", 10,
  &TeleopImu::callBack, this);
}

void TeleopImu::callBack(const sensor_msgs::Imu::ConstPtr& imu)
{
  turtlesim::Velocity vel;
  tf::Quaternion bq(imu->orientation.x,imu->orientation.y,imu-
  >orientation.z,imu->orientation.w);
    double roll,pitch,yaw;
    tf::Matrix3x3(bq).getRPY(roll,pitch,yaw);
      vel.angular = roll;
      vel.linear = pitch;
  pub.publish(vel);

}

int main(int argc, char** argv)
{
ros::init(argc, argv, "teleopImu");
    TeleopImu teleop_turtle;

ros::spin();
}
```

这个节点将会订阅 imu/data 主题并发布一个 turtlesim 运动指令的新主题。

```
TeleopImu::TeleopImu()
{
pub = n.advertise<turtlesim::Velocity>("turtle1/command_velocity",1);
sub = n.subscribe<sensor_msgs::Imu>("imu/data", 10,
&TeleopImu::callBack, this);
}
```

这部分代码中最重要的是 callBack 函数。在这个函数内部，接收 IMU 主题并处理成一个新的 turtlesim/velocity 主题。也许你还记得，这个消息的类型能够控制小海龟移动的速度。

```
void TeleopImu::callBack(const sensor_msgs::Imu::ConstPtr& imu)
{
...
tf::Matrix3x3(bq).getRPY(roll,pitch,yaw);
  vel.angular = roll;
    vel.linear = pitch;
pub.publish(vel);
}
```

通过 Matrix3x3 类实现四元数到欧拉角的转换。然后，我们将会使用这个类中提供的

getRPY 访问器方法。通过这个方法，我们能够从矩阵中获得相对于 x、y 和 z 轴的滚动、俯仰和偏航值。

在这之后我们需要将滚动和俯仰的值分别给线速度和角速度变量。

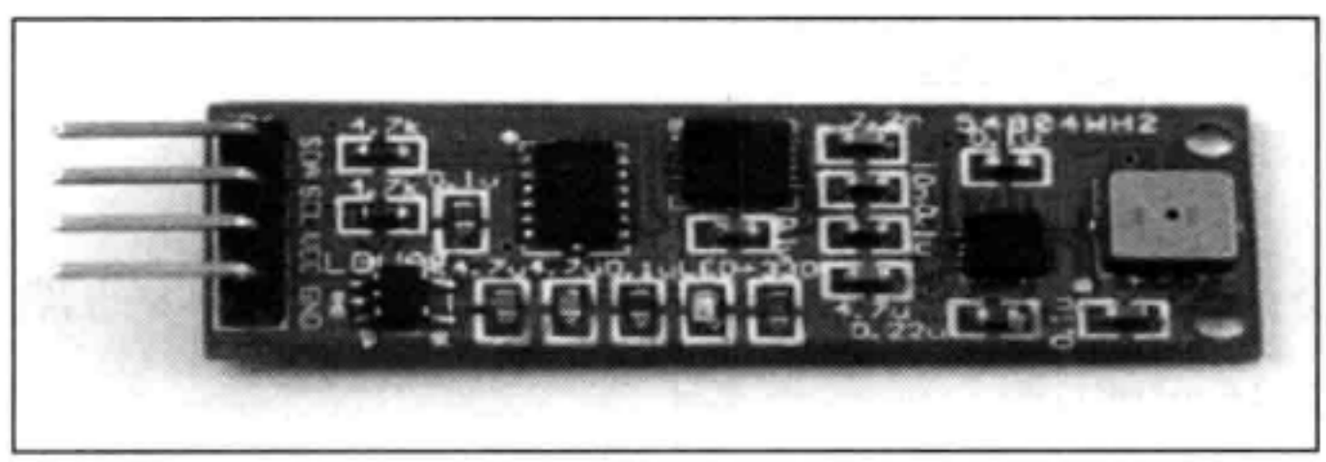
现在，如果你同时运行所有节点，那么你会看到小海龟会根据 IMU 的运动进行移动，就像用游戏杆一样的效果。

4.7 使用低成本惯性测量模组 IMU-10 自由度

在本节中，我们将会学习使用一种低成本传感器 IMU-10 自由度（DoF）。这个传感器和 Xsens Mti 非常相似，都有加速度计（×3）、磁场强度计（×3）、气压计（×1）和陀螺仪（×3）。通过 I2C 接口能够轻松地控制它，并且在本示例中，它会与 Arduino Nano 进行连接（http://arduino.cc/en/Main/ArduinoBoardNano）。

这个传感器的用途也是非常相似的。购买 Xsens 大概要花费 $2000，对于一般的用户来说是比较昂贵的，而我们本节中所要介绍的传感器只需要大约 $20，这个传感器亲民的价格让它能够用在很多项目上。

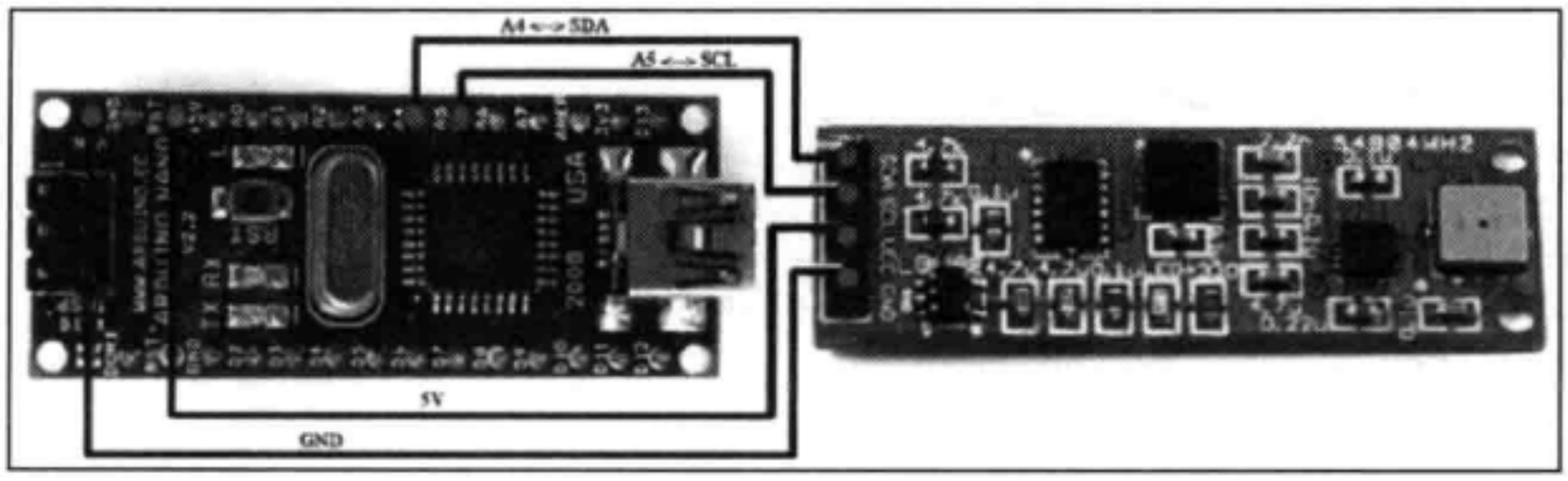
通过下面的图片你能看到这个传感器。它很薄并且只有十几个元件。



这块板子具有以下传感器：

- **ADXL345**：这是一个高精度（13-bit）三轴加速度计，最高测量范围为 ±16g。这个传感器广泛地被用于移动设备终端。在倾角测量时它能够测量静态的重力加速度，同时也能够很好地完成移动和振动的动态加速度计算。
- **HMC5883L**：这个传感器是用来测量低磁场强度传感，并配有针对低成本指南针和磁强计等设备的数字化接口。
- **BMP085**：这是高精度大气压力传感器用于先进的移动应用。它性能强大，具有最高 0.03 hPa 的绝对精度和 3 μA 的超低功耗。
- **L3G4200D**：这是高精度（16-bit）三轴陀螺仪，测量范围达到每秒 2000 度（d/s）。这个陀螺仪能够同时对三个轴的转动量进行测量。

正如前面所说，这块板子通过 I2C 接口进行控制。我们能够使用 Arduino 来控制它。在下图中，你能看到如何连接 Arduino 和 IMU 这两块板子：



唯一需要做的事情是连接四根线让板子工作。从 Arduino 的 GND 和 5V 分别连线到 IMU 的 GND 和 Vcc。

串口数据链路 (SDL) 需要连接到模拟量 pin4，串口时钟 (SCK) 需要连接到模拟量 pin5。如果你把这些针脚连接错误，Arduino 就无法和传感器通信。

4.7.1 下载加速度传感器库

在使用这些传感器之前，需要先下载正确的 Arduino 库。

在互联网上，你能够找到很多很多各种功能的库，但在本书中使用的库可以从 https://github.com/jenschr/ Arduino-libraries 下载。

一旦你已经下载了库文件，在你的 `sketchbook` 文件夹下解压缩并加载它。

你可以在网上寻找其他传感器的库。但是为了方便工作，你可以在 `chapter4_tutorials/libraries` 文件夹下找到使用 10DoF 传感器所需要的所有库。在每一个库里，你都能找到使用传感器的示例。

4.7.2 Arduino Nano 和 10 自由度传感器编程

在本小节中，我们将会编写 Arduino 程序来获取加速度计的数据并在 ROS 中发布。

打开 Arduino IDE 并且以 `example7.ino` 为名创建一个新文件，输入下列代码：

```
#include <ros.h>
#include <std_msgs/Float32.h>

#include <Wire.h>
#include <ADXL345.h>

ADXL345 Accel;

ros::NodeHandle  nh;
std_msgs::Float32 velLinear_x;
std_msgs::Float32 velAngular_z;

ros::Publisher velLinear_x_pub("velLinear_x", &velLinear_x);
ros::Publisher velAngular_z_pub("velAngular_z", &velAngular_z);

void setup(){
```

```
  nh.initNode();
  nh.advertise(velLinear_x_pub);
  nh.advertise(velAngular_z_pub);

  Wire.begin();
  delay(100);
    Accel.set_bw(ADXL345_BW_12);

}

void loop(){

  double acc_data[3];
    Accel.get_Gxyz(acc_data);

  velLinear_x.data = acc_data[0];
  velAngular_z.data = acc_data[1];

  velLinear_x_pub.publish(&velLinear_x);
  velAngular_z_pub.publish(&velAngular_z);

  nh.spinOnce();
  delay(10);
}
```

让我们一句一句地分析代码看看如何提取数据并发布数据。

我们将会使用两个 Float 32 变量来发布数据，并且需要在程序中包含以下头文件：

```
#include <std_msgs/Float32.h>
```

为了能够使用 I2C 总线与传感器进行通信，我们需要调用 Wire.h 库。这个库是标准的 Arduino 库。

```
#include <Wire.h>
```

为了使用之前下载的库，我们先要添加对这个头文件的引用：

```
#include <ADXL345.h>
```

使用 Accel 对象与传感器进行互动。

```
ADXL345 Accel;
```

从传感器读取的数据会被存储到以下变量中：velLinear_x 用于线性速度，其数值读取于加速度计的 x 轴方向；velAngular_z 用于角速度，其数值读取于加速度计的 z 轴方向。

```
std_msgs::Float32 velLinear_x;
std_msgs::Float32 velAngular_z;
```

这段程序会发布两个不同的主题：一个用于线性速度，另一个用于角速度。

```
ros::Publisher velLinear_x_pub("velLinear_x", &velLinear_x);
ros::Publisher velAngular_z_pub("velAngular_z", &velAngular_z);
```

这就是主题被创建的地方。一旦你执行了这些行，就会在 ROS 中看到两个分别名为 velAngular_z 和 velLinear_x 的主题。

```
nh.advertise(velLinear_x_pub);
nh.advertise(velAngular_z_pub);
```

传感器和它的带宽通过以下代码完成初始化：

```
Accel.set_bw(ADXL345_BW_12);
```

通过 get_Gxyz 返回的数据被存储在一个三维向量中。

```
double acc_data[3];
  Accel.get_Gxyz(acc_data);
```

最后，数据被发布：

```
velLinear_x_pub.publish(&velLinear_x);
velAngular_z_pub.publish(&velAngular_z);
```

4.7.3 创建 ROS 节点并使用 10 自由度传感器数据

在本小节，我们将会创建一段新程序用于使用传感器数据生成在 turtlesim 中移动小海龟的指令，目标是使用 10DoF 传感器板子作为 turtlesim 的输入并使用加速度计移动小海龟。

所以，我们需要在 chapter4_tutorials/src 文件夹下创建一个新文件 example8.cpp，并输入以下代码：

```
#include<ros/ros.h>
#include<turtlesim/Velocity.h>
#include<std_msgs/Float32.h>

class TeleopImu{
  public:
  TeleopImu();
  private:
    void velLinearCallBack(const std_msgs::Float32::ConstPtr& vx);
    void velAngularCallBack(const std_msgs::Float32::ConstPtr& wz);
    ros::NodeHandle n;
    ros::Publisher pub;
    ros::Subscriber velAngular_z_sub;
    ros::Subscriber velLinear_x_sub;
  turtlesim::Velocity vel;
};

TeleopImu::TeleopImu()
{
  pub = n.advertise<turtlesim::Velocity>("turtle1/command_
velocity",1);
```

```
  velLinear_x_sub = n.subscribe<std_msgs::Float32>("/velLinear_x", 1,
&TeleopImu::velLinearCallBack, this);
  velAngular_z_sub = n.subscribe<std_msgs::Float32>("/velAngular_z",
1, &TeleopImu::velAngularCallBack, this);
}
void TeleopImu::velAngularCallBack(const std_msgs::Float32::ConstPtr&
wz){
    vel.linear = -1 * wz->data;
    pub.publish(vel);
}

void TeleopImu::velLinearCallBack(const std_msgs::Float32::ConstPtr&
vx){
  vel.angular = vx->data;
    pub.publish(vel);
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "example8");
  TeleopImu teleop_turtle;
  ros::spin();
}
```

这段代码和使用了 Xsens IMU 的 example6.cpp 非常相近，唯一的区别是订阅的主题和数据的类型。在本例中，我们订阅了两个由 Arduino Nano 创建的主题，而总共使用三个主题一起控制 turtlesim。

我们必须要订阅 /velLinear_x 和 velAngular_z 主题，通过以下代码可以实现：

```
velLinear_x_sub = n.subscribe<std_msgs::Float32>("/velLinear_x", 1,
&TeleopImu::velLinearCallBack, this);
velAngular_z_sub = n.subscribe<std_msgs::Float32>("/velAngular_z", 1,
&TeleopImu::velAngularCallBack, this);
```

每一次节点收到消息，它会从消息中提取数据并创建一个新的 turtlesim/Velocity 消息并发布出去。

```
void TeleopImu::velAngularCallBack(const std_msgs::Float32::ConstPtr&
wz){
  vel.linear = -1 * wz->data;
  pub.publish(vel);
}

void TeleopImu::velLinearCallBack(const std_msgs::Float32::ConstPtr&
vx){
vel.angular =  vx->data;
  pub.publish(vel);
}
```

为了运行这个示例，请记住编译代码并按照以下步骤执行：

1. 在新的命令行窗口启动 roscore。连接 Arduino 和电脑并启动以下命令：

```
$ rosrun rosserial_python serial_node.py
```

2. 现在，通过以下命令启动 turtlesim：

```
$ rosrun turtlesim turtlesim_node
```

最后，输入以下命令启动节点：

```
$ rosrun chapter4_tutorials example8
```

如果一切运行正常，你会看到 TurtleSim 运行界面中小海龟在移动。如果你移动传感器，小海龟就会直线运动或者改变方向。

4.8 本章小结

在机器人中使用传感器和执行机构是非常重要的，因为这是和现实世界进行互动的唯一办法。在本章中，我们进一步学习了如何使用、配置和检查一些通用的传感器和执行机构，这些设备在全世界的机器人领域里广泛使用。可以确定，如果你想使用其他类型的传感器，你都能够从网络上找到相关信息并轻松地使用它。

在我看来，Arduino 是最有趣的设备。因为有了它，你能够向计算机添加更多的设备和各种廉价的传感器，能够很简单轻松地在 ROS 框架下使用它们。Arduino 还有非常大的社区，这样你能够找到各种设备的各种信息，绝对能过覆盖你所能想象到的各种应用。

最后，需要说明的是，激光雷达在后面的章节中是最常用的传感器。因为这是应用导航功能包集所必须使用的设备。导航功能包依赖于激光雷达提供的高精确度和高频率的数据。在下一章中，你会看到如何进行机器人建模，在 rviz 中对机器人进行可视化，并在 ROS 已经集成了的 Gazebo 虚拟环境中加载和使用它。

第 5 章
3D 建模与仿真

直接在一个真正的机器人上进行编程能够给我们最佳的反馈，并且也更加令人激动，但是并不是每个人都有机会接触到真正的机器人。基于这个原因，我们也有能够仿真整个物理世界的程序。

在本章中我们将会学习：

- 创建机器人的 3D 模型
- 为机器人提供运动、物理限制、惯性和其他物理响应
- 为机器人 3D 模型添加仿真传感器
- 在仿真环境中使用该模型

5.1 自定义机器人在 ROS 中的 3D 模型

在 ROS 中使用机器人 3D 模型或部分结构模型主要是用于仿真机器人或者是为了帮助开发者简化他们的日常工作，当然具体方法都是通过使用 URDF 文件。

标准化机器人描述格式（URDF）是一种用于描述机器人、其部分结构、关节、自由度等的 XML 格式文件。你每次在 ROS 系统中看到 3D 的机器人都会有对应的 URDF 文件与之对应，例如 PR2（Willow Garage）或者是 Robonaut（NASA）。在下面的章节中我们将会学习如何创建这种文件和格式以定义不同的值。

5.2 创建第一个 URDF 文件

我们将在下面章节中建立的机器人是一种最常见的移动机器人，它有四个轮子、一个带有抓取器的手臂。

为了打好基础，我们先做一个带有四个轮子的机器人底座。在 `chapter5_tutorials/urdf` 文件夹下创建一个新文件并命名为 `robot1.urdf`。将下面的代码复制到文件中，这些 URDF 代码是基于 XML 的，这种格式并不强制缩进但还是建议大家这么做。所以，建议大家使用一个支持缩进的编辑器或者找到适当插件并进行配置（例如，比较好的是 VIM 中的 `.vimrc` 文件）：

```
<?xml version="1.0"?>
  <robot name="Robot1">
    <link name="base_link">
      <visual>
        <geometry>
          <box size="0.2 .3 .1"/>
        </geometry>
        <origin rpy="0 0 0" xyz="0 0 0.05"/>
        <material name="white">
          <color rgba="1 1 1 1"/>
        </material>
      </visual>
    </link>

    <link name="wheel_1">
      <visual>
        <geometry>
          <cylinder length="0.05" radius="0.05"/>
        </geometry>
        <origin rpy="0 1.5 0" xyz="0.1 0.1 0"/>
          <material name="black">
            <color rgba="0 0 0 1"/>
          </material>
      </visual>
    </link>

    <link name="wheel_2">
      <visual>
        <geometry>
          <cylinder length="0.05" radius="0.05"/>
        </geometry>
        <origin rpy="0 1.5 0" xyz="-0.1 0.1 0"/>
        <material name="black"/>
      </visual>
    </link>

    <link name="wheel_3">
      <visual>
        <geometry>
          <cylinder length="0.05" radius="0.05"/>
        </geometry>
        <origin rpy="0 1.5 0" xyz="0.1 -0.1 0"/>
        <material name="black"/>
      </visual>
    </link>

    <link name="wheel_4">
      <visual>
        <geometry>
          <cylinder length="0.05" radius="0.05"/>
        </geometry>
        <origin rpy="0 1.5 0" xyz="-0.1 -0.1 0"/>
        <material name="black"/>
      </visual>
    </link>

    <joint name="base_to_wheel1" type="fixed">
```

```
    <parent link="base_link"/>
    <child link="wheel_1"/>
    <origin xyz="0 0 0"/>
  </joint>

  <joint name="base_to_wheel2" type="fixed">
    <parent link="base_link"/>
    <child link="wheel_2"/>
    <origin xyz="0 0 0"/>
  </joint>

  <joint name="base_to_wheel3" type="fixed">
    <parent link="base_link"/>
    <child link="wheel_3"/>
    <origin xyz="0 0 0"/>
  </joint>

  <joint name="base_to_wheel4" type="fixed">
    <parent link="base_link"/>
    <child link="wheel_4"/>
    <origin xyz="0 0 0"/>
  </joint>
</robot>
```

5.2.1 解释文件格式

如你在代码中所见，有两种用于描述机器人几何结构的基本字段：连接（link）和关节（joint）。

第一个连接的名字是 base_link（基本连接），这个名称在文件中必须是唯一的：

```
<link name="base_link">
  <visual>
    <geometry>
      <box size="0.2 .3 .1"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0.05"/>
    <material name="white">
      <color rgba="1 1 1 1"/>
    </material>
  </visual>
</link>
```

为了定义我们在仿真环境中的所见，在前面的代码中使用了 visual 字段。在代码中你能够定义几何形状（圆柱体、立方体、球体和网格）、材质（颜色和纹理）和原点。然后使用以下代码定义关节，如下所示：

```
<joint name="base_to_wheel1" type="fixed">
      <parent link="base_link"/>
      <child link="wheel_1"/>
      <origin xyz="0 0 0"/>
</joint>
```

在 joint 字段中，我们先定义名称，名称需要唯一。然后定义关节类型（固定关节 fixed、转动关节 revolute、旋转关节 continuous、6 自由度浮动关节 floating 或平面运动关节 planar）、父连接坐标系和子连接坐标系（关节相连的前后坐标系）。在本例中，wheel_1 是 base_link 的子连接坐标系。它是固定关节（fixed），但如果这个关节是一个有转动角度限制的轴，那么我们可以设置其为转动关节（revolute）。

为了检查我们书写的语法是否正确和配置是否有误，可以使用 check_urdf 命令工具：

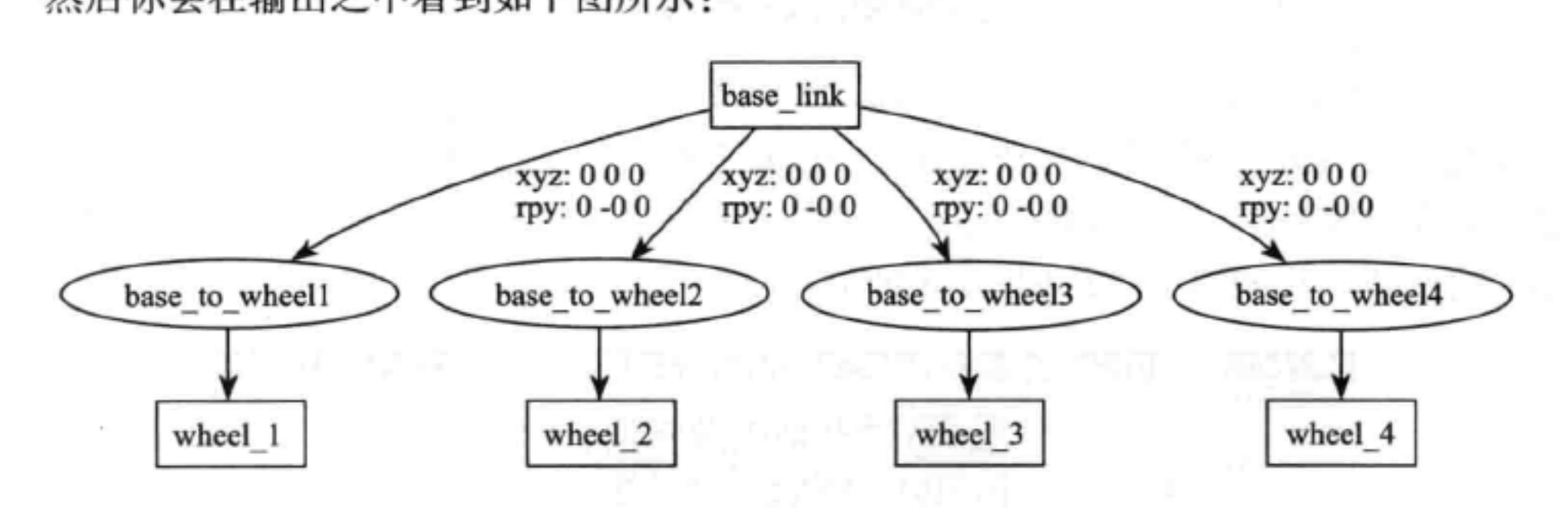
```
$ rosrun urdf_parser check_urdf robot1.urdf
robot name is: Robot1
---------- Successfully Parsed XML ---------------
root Link: base_link has 4 child(ren)
  child(1):  wheel_1
  child(2):  wheel_2
  child(3):  wheel_3
  child(4):  wheel_4
```

这个工具将会在 ROS Fuerte 之后被弃用，如果你使用的版本是比较新的 ROS 发行版本，例如 Groovy，那么你应该使用 urdfdom 功能包，其中包含了 check_urdf 节点 / 工具。

如果你希望以图形的方式来查看它，那么可以使用 urdf_to_graphiz 命令工具：

```
$ rosrun urdf_parser urdf_to_graphiz "`rospack find chapter5_tutorials`/
urdf/robot1.urdf"
```

然后你会在输出之中看到如下图所示：



5.2.2 在 rviz 里查看 3D 模型

现在我们有了自己机器人的模型，能够用 rviz 使用它并查看它的 3D 展示，以及它关节的运动。

在 chapter5_tutorials/launch 文件夹下创建 display.launch 文件，并在文件中输入以下代码：

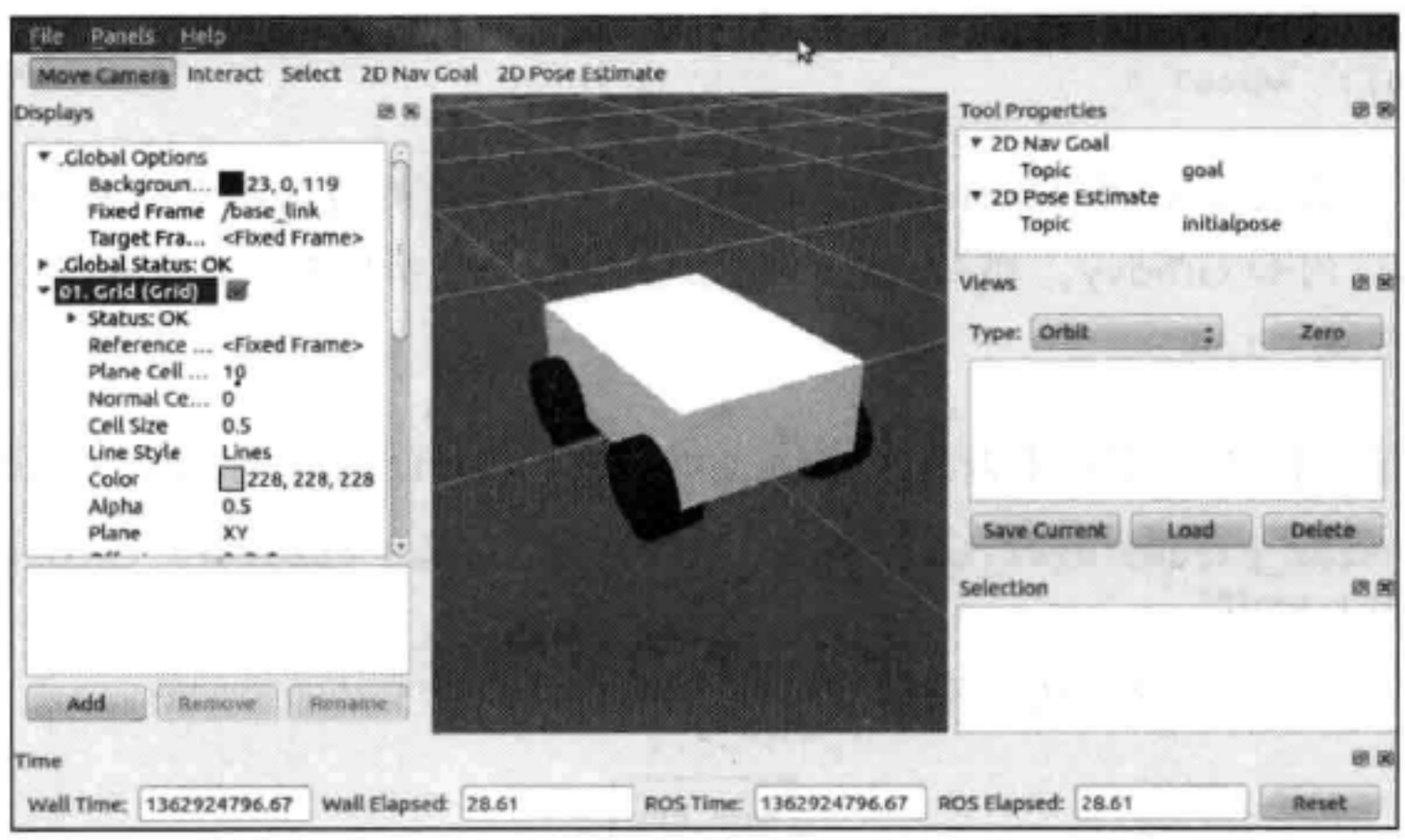
```
<launch>
      <arg name="model" />
      <arg name="gui" default="False" />
```

```
      <param name="robot_description" textfile="$(arg model)" />
      <param name="use_gui" value="$(arg gui)"/>
      <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
      <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
      <node name="rviz" pkg="rviz" type="rviz" args="-d $(find urdf_
tutorial)/urdf.vcg" />
</launch>
```

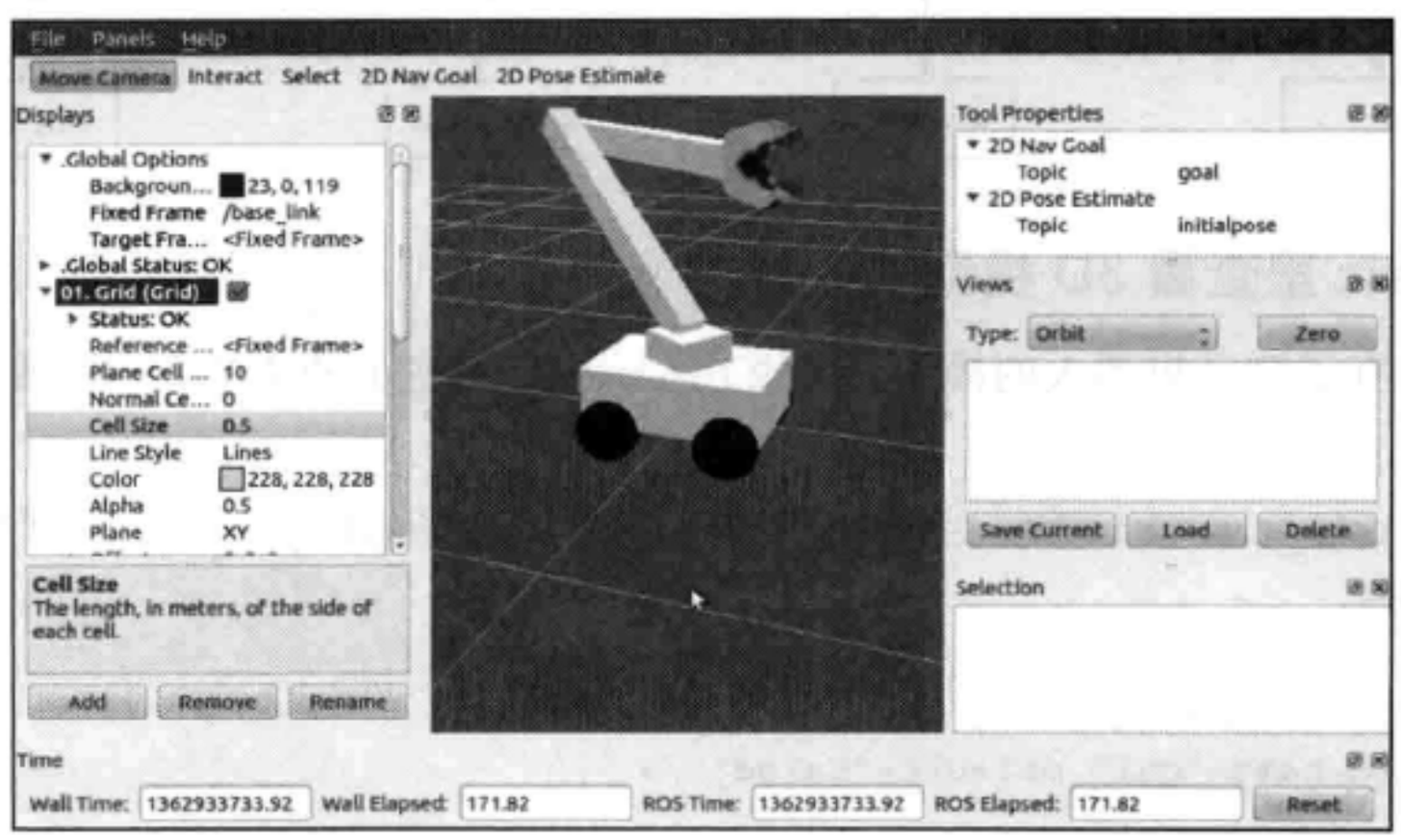
使用以下命令启动它：

```
$ roslaunch chapter5_tutorials display.launch model:="`rospack find
chapter5_tutorials`/urdf/robot1.urdf"
```

如果一切正常，你会看到如下带有 3D 模型图像的窗口：



让我们再添加一些组件来完成设计：一段基座臂、一段连接臂和一个夹持器。我们尝试自己来完成它的设计，你可以在 chapter5_tutorials/urdf/ robot1.urdf 文件中找到最终的模型。你能够看到如下图所示的最终设计：

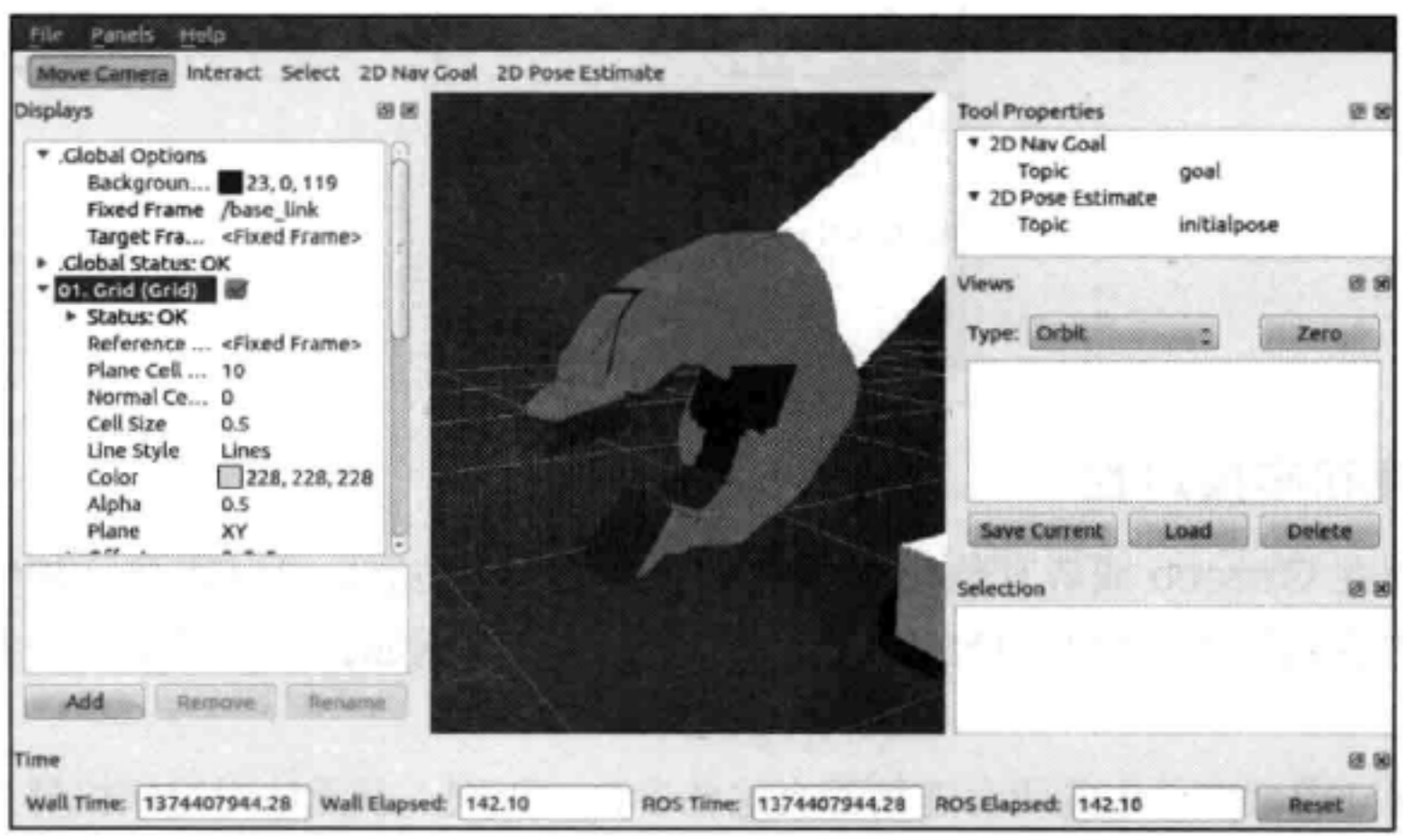


5.2.3 加载图形到机器人模型

有时候你希望自己构建的模型能够更加真实，并不是说简单地增加更多的几何形状和模块，而是通过添加更多的元素使模型变得更加丰富和细致。这就需要加载我们自行创建的图形（mesh）或者是使用其他机器人模型的图形。在本模型中，我们使用 PR2 机器人的夹持器。在下面的代码中你会看到我们是如何使用它的：

```
<link name="left_gripper">
<visual>
       <origin rpy="0 0 0" xyz="0 0 0"/>
           <geometry>
             <mesh filename="package://pr2_description/meshes/gripper_
v0/l_finger.dae"/>
           </geometry>
     </visual>
</link>
```

下面这个连接似乎和之前我们看到的一样，但是我们在几何部分添加了上面使用的图形。你会在下图中看到结果：



5.2.4 使机器人模型运动

为了将模型转换成能实际运动的机器人，我们唯一需要做的是检查好关节所选用的类型。如果你检查 URDF 模型文件，将会看到我们在模型中使用了不同类型的关节。

最常用的关节是转动关节。例如，我们在 arm_1_to_arm_base 上使用的就是，其代码如下所示：

```
<joint name="arm_1_to_arm_base" type="revolute">
              <parent link="arm_base"/>
              <child link="arm_1"/>
              <axis xyz="1 0 0"/>
```

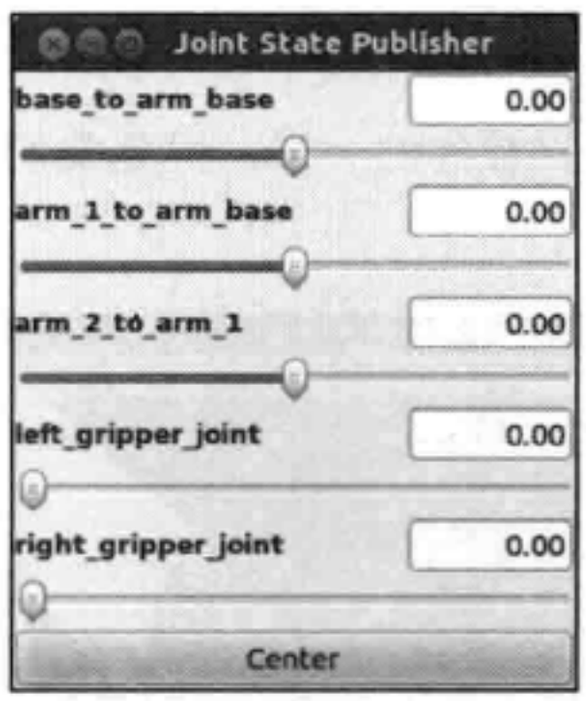
```
            <origin xyz="0 0 0.15"/>
<limit effort ="1000.0" lower="-1.0" upper="1.0" velocity="0.5"/>
</joint>
```

这意味着它们能够像旋转关节一样转动，但是它们的转动角具有一定的限值。限值通过 `<limit effort ="1000.0" lower="-1.0" upper="1.0" velocity="0.5"/>` 这一行设置为固定值，可以通过配置 `axis xyz="1 0 0"` 选择其转动轴。

想要知道关节是否有一个轴或转动限值，有一种好的办法就是使用 `Join_State_Publisher` GUI 运行 `rviz`:

```
$ roslaunch chapter5_tutorials display.launch model:="`rospack find
chapter5_tutorials`/urdf/robot1.urdf" gui:=true
```

我们会看到 `rviz` 界面的另一种带有很多滑块的窗口，其中每个滑块都能控制一个关节：



5.2.5 物理和碰撞属性

如果你想在 Gazebo 或者其他仿真软件中进行机器人仿真，这就需要添加物理和碰撞属性。这意味着我们需要设定几何尺寸来计算可能的碰撞，例如，需要设定重量我们才能够计算惯性等。

你需要保证模型文件中的所有连接都有这些参数，否则，就无法对这些机器人进行仿真。

对于图形模型文件来说，ROS 中建模的简单几何形状比引入的图形模型更加容易进行碰撞计算。两个图形模型之间进行碰撞计算相比简单几何形状来说，要使用更加复杂的计算方法，也会耗费更多的计算资源。

在下面的代码中，你会看到我们向名为 `wheel_1` 的连接中添加这些参数：

```
<link name="wheel_1">
  ...
  <collision>
    <geometry>
      <cylinder length="0.05" radius="0.05"/>
    </geometry>
```

```
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0"
izz="1.0"/>
  </inertial>
</link>
```

对于其他的连接也是这样做。记住要为所有连接添加碰撞 collision 和惯性 inertial 的属性，因为如果你不这样做的话，Gazebo 将会无法使用这些模型。

你能够在 chapter5_tutorials/urdf/ robot1_physics.urdf 中查看带有所有参数的完整文件。

5.3 xacro——一个写机器人模型的更好方法

我们看一下 robot1_physics.urdf 文件的大小。它使用了 314 行代码来定义机器人。想象一下如果我们开始添加摄像头、腿或者其他几何部件，那么这个文件将会急剧增大，并且维护这些代码将会变得非常困难。

xacro 可帮助我们减小 URDF 文件的尺寸，并且增加文件的可读性和可维护性。它还允许我们创建模型并复用这些模型去创建相同的结构，如更多的手臂和腿。

为了开始使用 xacro，我们需要指定一个命名空间，以便文件能够被正确地解析。例如，这是一个 xacro 文件的前两行：

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot1_
xacro">
```

其中，我们定义了模型的名称，在本例中是 robot1_xacro。记住文件的扩展名必须是 .xacro，而不是 .urdf。

5.3.1 使用常量

我们使用 xacro 声明常量，因此能够避免在很多行重复使用相同的值。如果不使用 xacro，若需要改变一个值，那就要去修改若干个地方，从而很难进行文件的维护。

例如，四个轮子使用相同的长度和半径。如果我们希望修改这个值，那么需要在每一行进行修改；而如果使用如下的定义，那么修改就会变得轻松：

```
<xacro:property name="length_wheel" value="0.05" />
<xacro:property name="radius_wheel" value="0.05" />
```

现在，只需要使用 ${name_of_variable} 来引用刚才想要修改的这些值：

```
<cylinder length="${length_wheel}" radius="${radius_wheel}"/>
```

5.3.2 使用数学方法

你可以在 ${} 结构中使用基本的四则运算 (+、-、*、/、一元减号和括号) 表达式来构建任意复杂程度的模型。但是不包括求幂和求模运算：

```
<cylinder radius="${wheeldiam/2}" length=".1"/>
<origin xyz="${reflect*(width+.02)} 0 .25" />
```

通过使用数学方法，我们能够通过修改某个值来更改模型的大小。而在此之前，我们需要先做好参数设计。

5.3.3 使用宏

宏是 xacro 功能包中最有用的组件。为了能够更多地减小文件，我们将会使用以下宏来做 inertial 初始化：

```
<xacro:macro name="default_inertial" params="mass">
                <inertial>
                        <mass value="${mass}" />
                        <inertia ixx="1.0" ixy="0.0" ixz="0.0"
                                 iyy="1.0" iyz="0.0"
                                 izz="1.0" />
                </inertial>
        </xacro:macro>
<xacro:default_inertial mass="100"/>
```

如果将 robot1.urdf 文件和 robot1.xacro 文件进行比较，我们能够删除掉 30 行没有任何功能的重复行。通过使用宏和变量，我们还能够进一步缩小文件。

为了在 rviz 和 Gazebo 中使用 xacro 文件，你需要将它转换成 .urdf 文件。你可以在 chapter5_tutorials/urdf 文件夹下执行以下命令来完成转换：

```
$ rosrun xacro xacro.py robot1.xacro > robot1_processed.urdf
```

你也可以在任意地方执行以下命令，也会起到和前面命令相同的作用：

```
$ rosrun xacro xacro.py "`rospack find chapter5_tutorials`/urdf/robot1.
xacro" > "`rospack find chapter5_tutorials`/urdf/robot1_processed.urdf"
```

尽管如此，为了保证你书写的命令更加简洁，我还是建议你在同一个文件夹下工作。

5.3.4 使用代码移动机器人

好了，我们已经建好了机器人的 3D 模型，并且能够在 rviz 中看到它，但如何通过节点让机器人运动呢？

在 chapter5_tutorials/src 文件夹下以 state_publisher.cpp 为名称创建一个新文件，并复制以下代码：

```
#include <string>
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <tf/transform_broadcaster.h>

int main(int argc, char** argv) {
  ros::init(argc, argv, "state_publisher");
  ros::NodeHandle n;
  ros::Publisher joint_pub = n.advertise<sensor_
msgs::JointState>("joint_states", 1);
  tf::TransformBroadcaster broadcaster;
  ros::Rate loop_rate(30);

  const double degree = M_PI/180;

  // robot state
  double inc= 0.005, base_arm_inc= 0.005, arm1_armbase_inc= 0.005,
arm2_arm1_inc= 0.005, gripper_inc= 0.005, tip_inc= 0.005;

  double angle= 0 ,base_arm = 0, arm1_armbase = 0, arm2_arm1 = 0,
gripper = 0, tip = 0;
  // message declarations
  geometry_msgs::TransformStamped odom_trans;
  sensor_msgs::JointState joint_state;
  odom_trans.header.frame_id = "odom";
  odom_trans.child_frame_id = "base_link";

  while (ros::ok()) {
      //update joint_state
      joint_state.header.stamp = ros::Time::now();
      joint_state.name.resize(7);
      joint_state.position.resize(7);
      joint_state.name[0] ="base_to_arm_base";
      joint_state.position[0] = base_arm;
      joint_state.name[1] ="arm_1_to_arm_base";
      joint_state.position[1] = arm1_armbase;
      joint_state.name[2] ="arm_2_to_arm_1";
      joint_state.position[2] = arm2_arm1;
      joint_state.name[3] ="left_gripper_joint";
      joint_state.position[3] = gripper;
      joint_state.name[4] ="left_tip_joint";
      joint_state.position[4] = tip;
      joint_state.name[5] ="right_gripper_joint";
      joint_state.position[5] = gripper;
      joint_state.name[6] ="right_tip_joint";
      joint_state.position[6] = tip;

      // update transform
      // (moving in a circle with radius 1)
      odom_trans.header.stamp = ros::Time::now();
      odom_trans.transform.translation.x = cos(angle);
      odom_trans.transform.translation.y = sin(angle);
```

```
        odom_trans.transform.translation.z = 0.0;
        odom_trans.transform.rotation = tf::createQuaternionMsgFromYaw(
angle);

         //send the joint state and transform
        joint_pub.publish(joint_state);
        broadcaster.sendTransform(odom_trans);

          // Create new robot state
        arm2_arm1 += arm2_arm1_inc;
        if (arm2_arm1<-1.5 || arm2_arm1>1.5) arm2_arm1_inc *= -1;
          arm1_armbase += arm1_armbase_inc;
if (arm1_armbase>1.2 || arm1_armbase<-1.0) arm1_armbase_inc *= -1;
        base_arm += base_arm_inc;
        if (base_arm>1. || base_arm<-1.0) base_arm_inc *= -1;
        gripper += gripper_inc;
        if (gripper<0 || gripper>1) gripper_inc *= -1;
         angle += degree/4;

        // This will adjust as needed per iteration
        loop_rate.sleep();
    }
    return 0;
}
```

下面来对这些能够让机器人通过节点运动的代码进行分析。

首先，我们创建了一个名为 odom 的坐标系，所有的变换都以这个新坐标系作为参考。如果你还记得，所有连接都是 base_link 的子连接坐标系，所有坐标系都会连接到 odom 坐标系：

```
...
geometry_msgs::TransformStamped odom_trans;
odom_trans.header.frame_id = "odom";
odom_trans.child_frame_id = "base_link";
...
```

现在，我们将会创建一个用于控制模型所有关节的新主题。Joint_state 是用来存储一系列转矩控制关节状态的消息。因为模型总共有 7 个关节，因此我们创建一个带有 7 个字段的消息：

```
sensor_msgs::JointState joint_state;

joint_state.header.stamp = ros::Time::now();
joint_state.name.resize(7);
joint_state.position.resize(7);
joint_state.name[0] ="base_to_arm_base";
joint_state.position[0] = base_arm;
...
```

在本示例中，机器人会在一个圆形的轨迹上运动。我们在下一段代码中对机器人的坐标

和运动进行计算：

```
odom_trans.header.stamp = ros::Time::now();
odom_trans.transform.translation.x = cos(angle)*1;
odom_trans.transform.translation.y = sin(angle)*1;
odom_trans.transform.translation.z = 0.0;
odom_trans.transform.rotation = tf::createQuaternionMsgFromYaw(angle);
```

最后，我们发布机器人的最新状态：

```
joint_pub.publish(joint_state);
broadcaster.sendTransform(odom_trans);
```

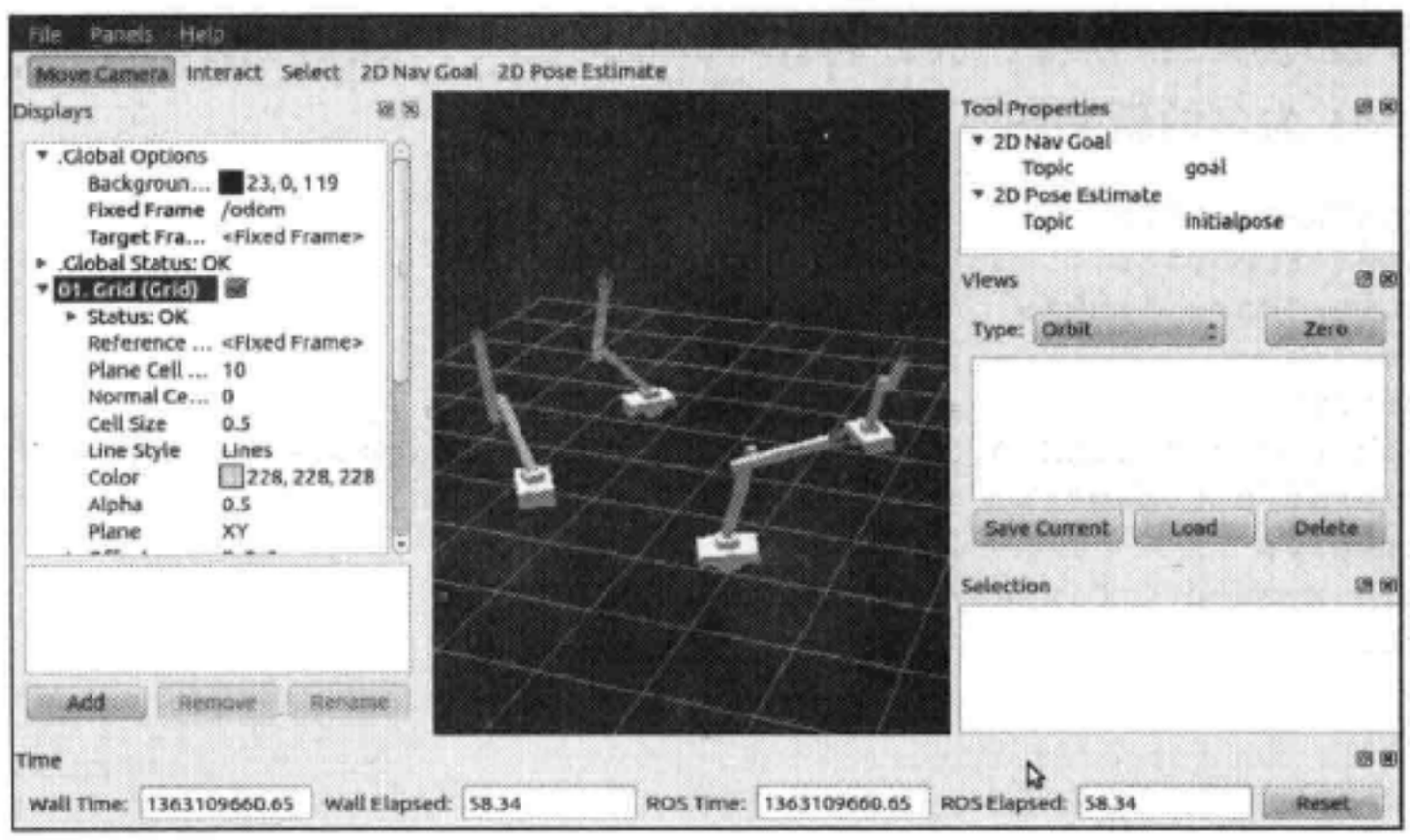
我们还会创建 launch 文件来启动节点、模型和所有必要的组件。在 chapter5_tutorials/launch 文件夹下以 display_xacro.launch 为名创建一个新文件（内容如下）：

```
<launch>
      <arg name="model" />
      <arg name="gui" default="False" />
      <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />
      <param name="use_gui" value="$(arg gui)"/>
      <node name="state_publisher" pkg="chapter5_tutorials"
type="state_publisher" />
      <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
      <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
chapter5_tutorial)/urdf.vcg" />
</launch>
```

使用以下命令能启动带有完整模型的新节点。我们将会在 rviz 中看到 3D 模型每一个结合处都在运动：

```
$ roslaunch chapter5_tutorials state_xacro.launch model:="`rospack find
chapter5_tutorials`/urdf/robot1.xacro"
```

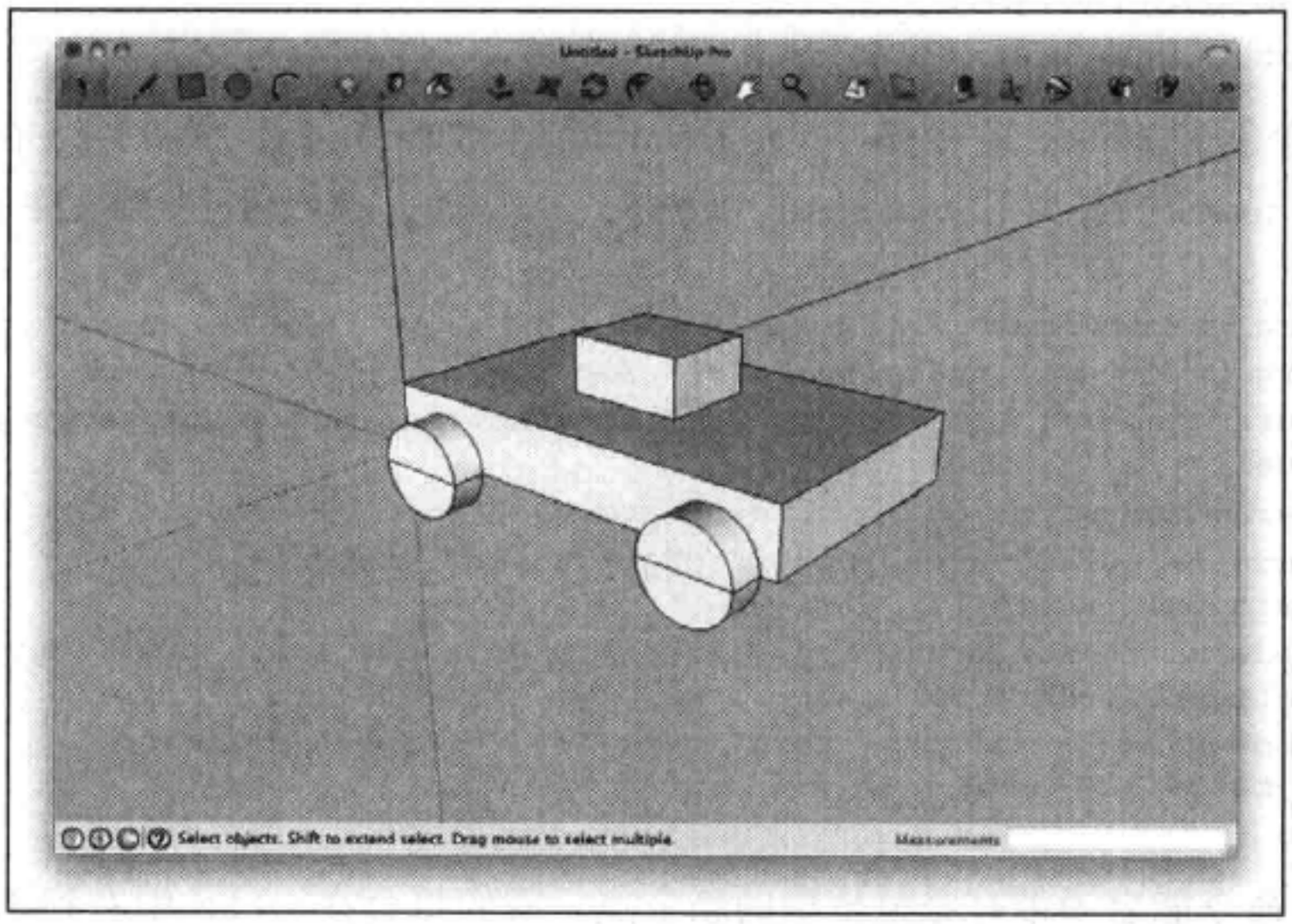
在下图中，你能看到一个带有 4 个屏幕截屏的组合，它们分别显示了我们通过节点所捕捉的机器人的运动。如果你能很好地看到它，就能看到手臂移动及其圆形的轨迹：



5.3.5 使用 SketchUp 进行 3D 建模

我们还可以使用像 SketchUp 之类的 3D 建模程序来创建模型。在本小节中，我们将会展示如何创建简单模型、导出它、生成 urdf 文件，并在 rviz 中查看模型。请注意，SketchUp 只能在 Windows 系统或 Mac 系统下运行，而本书中的模型是在 Mac 中开发的，而不是在 Linux 中。

首先，你需要在电脑中安装 SketchUp。当安装好之后，创建一个和下图类似的模型：



模型在导出之后仅有一个文件，因此轮子和底盘都在同一个文件中。如果想要创建一个部件能够移动的机器人，必须分别使用不同的文件导出模型的每一个部分。

为了导出模型，从菜单栏上找到 Export | 3D Model | Save As COLLADA File(*.dae)。

我们将文件命名为 bot.dae，并将这个文件保存在 chapter5_tutorials/urdf 文件夹下。

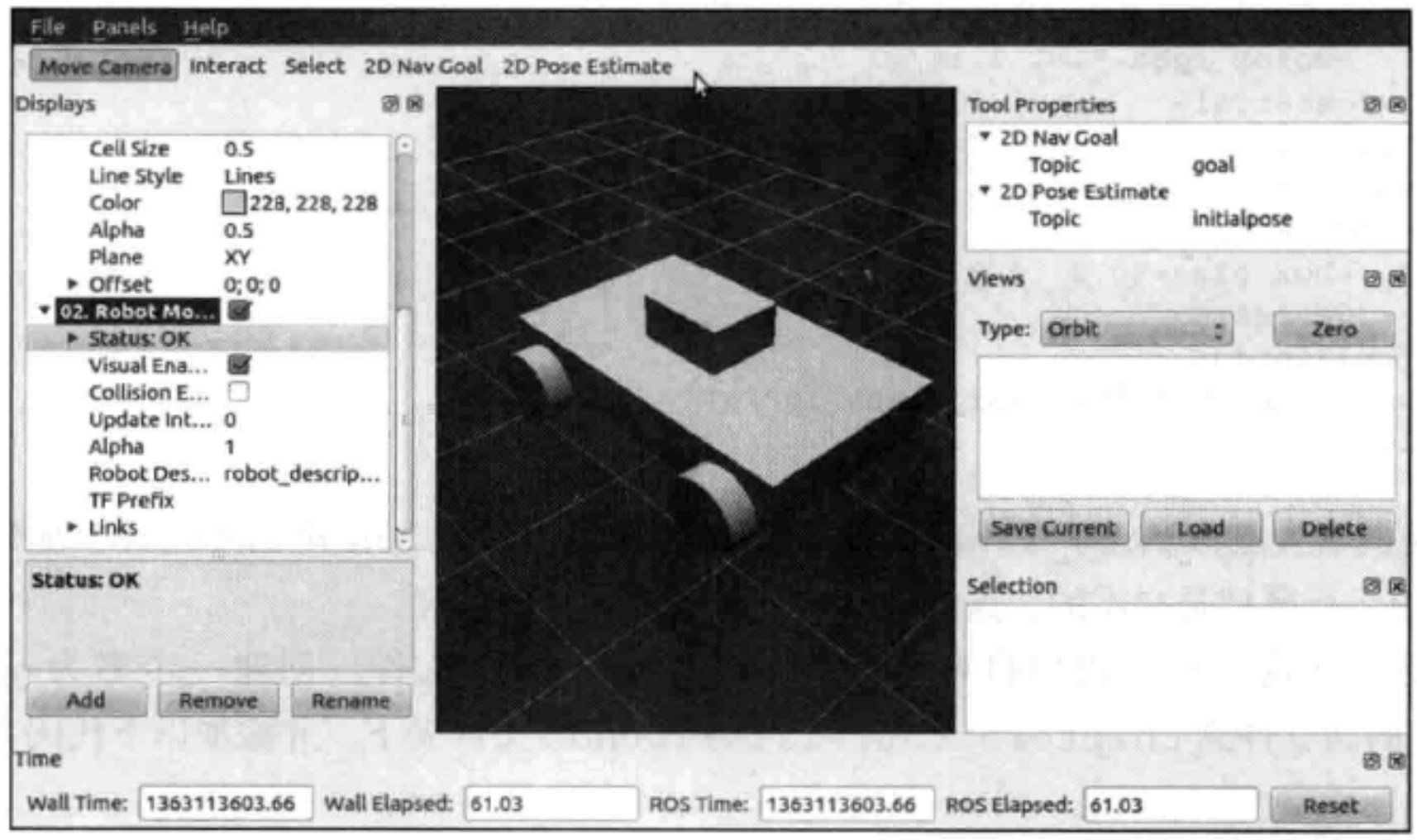
现在，为了能使用 3D 模型，我们将在 chapter5_ tutorials/urdf 文件夹下创建一个以 dae.urdf 为名的新文件。在文件中添加以下代码：

```
<?xml version="1.0"?>
<robot name="robot1">
  <link name="base_link">
  <visual>
    <geometry>
      <mesh scale="0.025 0.025 0.025" filename="package://chapter5_
tutorials/urdf/bot.dae"/>
    </geometry>
    <origin xyz="0 0 0.226"/>
  </visual>
  </link>
</robot>
```

使用以下命令测试模型：

```
$ roslaunch chapter5_tutorials display.launch model:="`rospack find
chapter5_tutorials`/urdf/dae.urdf"
```

你会看到以下输出：



5.4 在 ROS 中仿真

想要在 ROS 系统中对我们的模型进行仿真，需要使用 Gazebo。

Gazebo（http://gazebosim.org/）是一种适用于多机器人和室外环境的仿真环境。它能够在三维环境中对多个机器人、传感器及物体进行仿真，产生实际传感器反馈和物体之间的物理响应。

在本节中，你会学习如何使用之前创建的机器人模型，如何加载一个激光雷达，并使机器人模型像真的机器人一样移动。

5.4.1 在 Gazebo 中使用 URDF3D 模型

我们将会使用在上一节中创建的模型，但是并不包含手臂，这样能够让示例更简单一些。

你需要先完成 URDF 模型，因为要想在 Gazebo 中使用模型还需要声明很多字段。我们也将使用 `.xacro` 文件，虽然这可能更复杂，但是对于代码开发来说其功能非常强大。你能够在 `chapter5_tutorials/urdf/robot1_base_01.xacro` 找到修改后的文件：

```
<link name="base_link">
  <visual>
    <geometry>
      <box size="0.2 .3 .1"/>
    </geometry>
    <origin rpy="0 0 1.54" xyz="0 0 0.05"/>
    <material name="white">
      <color rgba="1 1 1 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <box size="0.2 .3 0.1"/>
    </geometry>
  </collision>
  <xacro:default_inertial mass="10"/>
</link>
```

这是机器人底盘 base_link 的新代码。请注意 collision 和 inertial 部分对于在 Gazebo 中运行模型是必需的，这样才能计算机器人的物理响应。

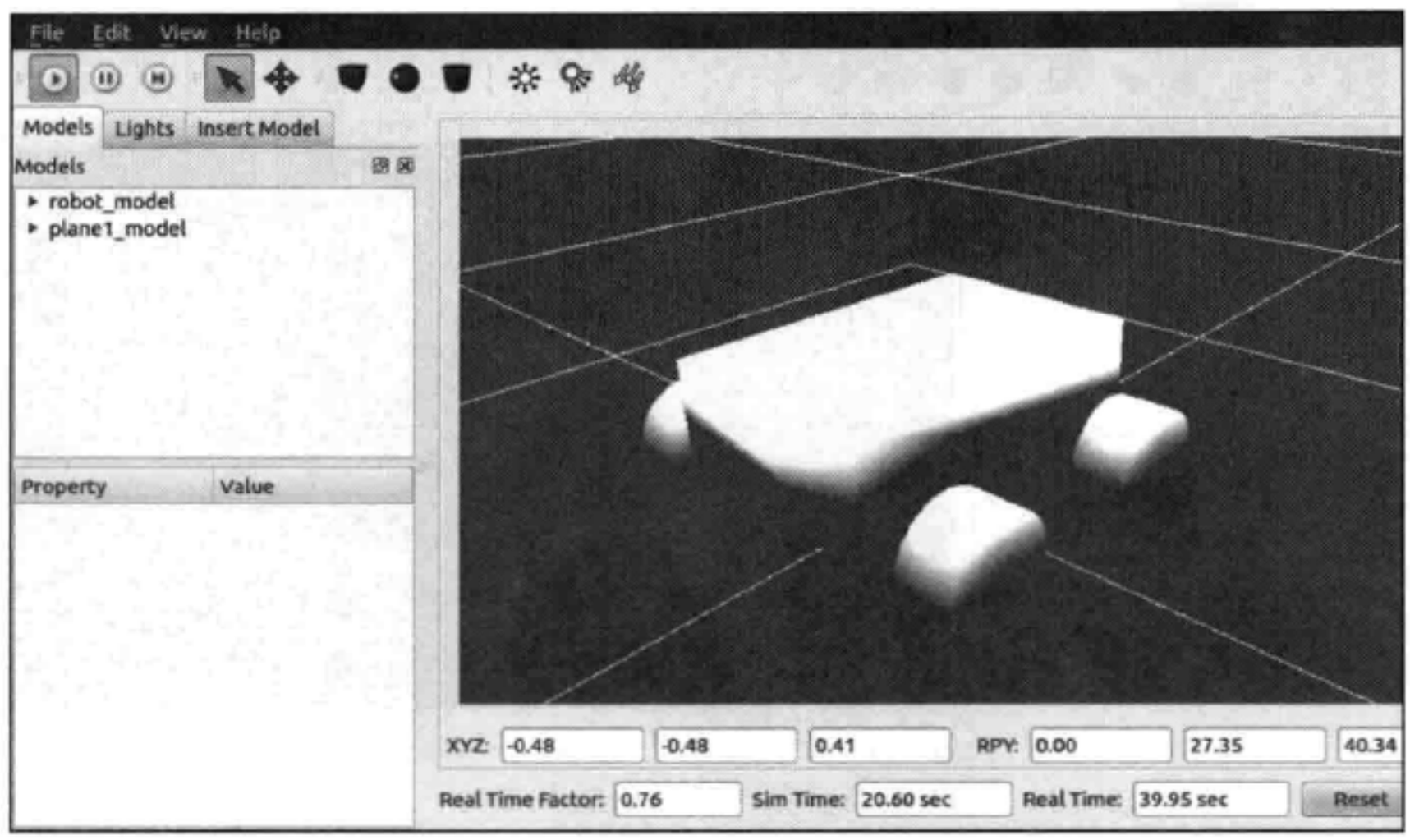
想要启动这一切，我们将要创建一个新的 .launch 文件。创建一个名为 gazebo.launch 的新文件到 chapter5_tutorials/launch 文件夹下，并添加以下代码：

```
<?xml version="1.0"?>
<launch>
<param name="/use_sim_time" value="true" />
    <include file="$(find gazebo_worlds)/launch/empty_world.launch"/>
    <arg name="model" />
    <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />
    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>

    <node pkg="robot_state_publisher" type="state_publisher"
name="robot_state_publisher" output="screen" >
  <param name="publish_frequency" type="double" value="50.0" />
   </node>
    <node name="spawn_robot" pkg="gazebo" type="spawn_model"
args="-urdf -param robot_description -z 0.1 -model robot_model"
respawn="false" output="screen" />
</launch>
```

想要启动文件，需要使用以下命令：

```
$ roslaunch chapter5_tutorials gazebo.launch  model:="`rospack find
chapter5_tutorials`/urdf/robot1_base_01.xacro"
```

你现在会在 Gazebo 中看到机器人。祝贺你！这是你在虚拟世界中迈出的第一步：



正如你所见，模型并没有任何的纹理渲染。在 rviz 中你能看到在 URDF 文件中声明的纹理。但是在 Gazebo 中，你却看不到它们。

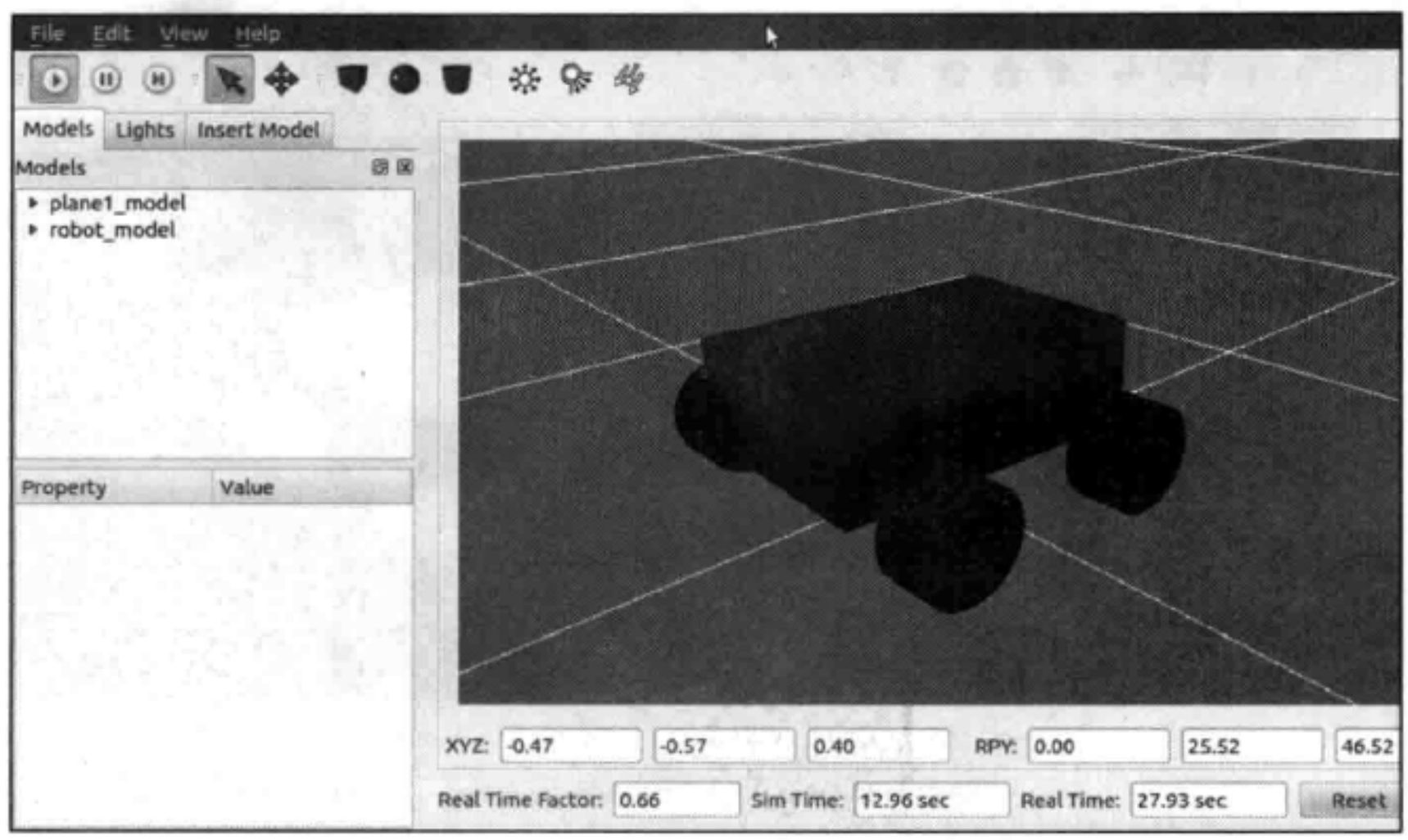
为了在 Gazebo 中添加可见的纹理，需要在模型文件中使用以下代码。将 chapter5_tutorials/urdf/robot1_base_01.xacro 文件另存为 robot1_base_02. xacro，并添加以下代码：

```
<gazebo reference="base_link">
      <material>Erratic/BlueBrushedAluminum</material>
 </gazebo>

 <gazebo reference="wheel_1">
      <material>Erratic/Black</material>
 </gazebo>

 <gazebo reference="wheel_2">
      <material>Erratic/Black</material>
 </gazebo>

 <gazebo reference="wheel_3">
      <material>Erratic/Black</material>
 </gazebo>

 <gazebo reference="wheel_4">
      <material>Erratic/Black</material>
 </gazebo>
```

启动这个新文件。你看到的机器人虽然还是同一个，但是已经带有纹理了：

```
$ roslaunch chapter5_tutorials gazebo.launch  model:="`rospack find
chapter5_tutorials`/urdf/robot1_base_02.xacro"
```

你将会看到以下输出：



5.4.2 在 Gazebo 中添加传感器

在 Gazebo 中，你能够对机器人的物理运动进行仿真。同样地，你还能仿真它的传感器。

通常情况下，当你想要添加一个新的传感器时，你需要实现其行为。幸运的是，有些传感器已经在 Gazebo 和 ROS 中完成了开发和集成。

在本节中我们将会向模型中添加一个激光雷达传感器。这个传感器将会成为机器人的一个新部件。你需要选好它的安装位置。在 Gazebo 中，你会看到一个新的 3D 模型，它很像 Hokuyo 激光雷达。我们在上一章中曾经介绍过它。

我们会从 `erratic_description` 功能包中调用激光雷达。这是 ROS 的神奇之处，你能够从其他功能包中复用代码从而简化开发。

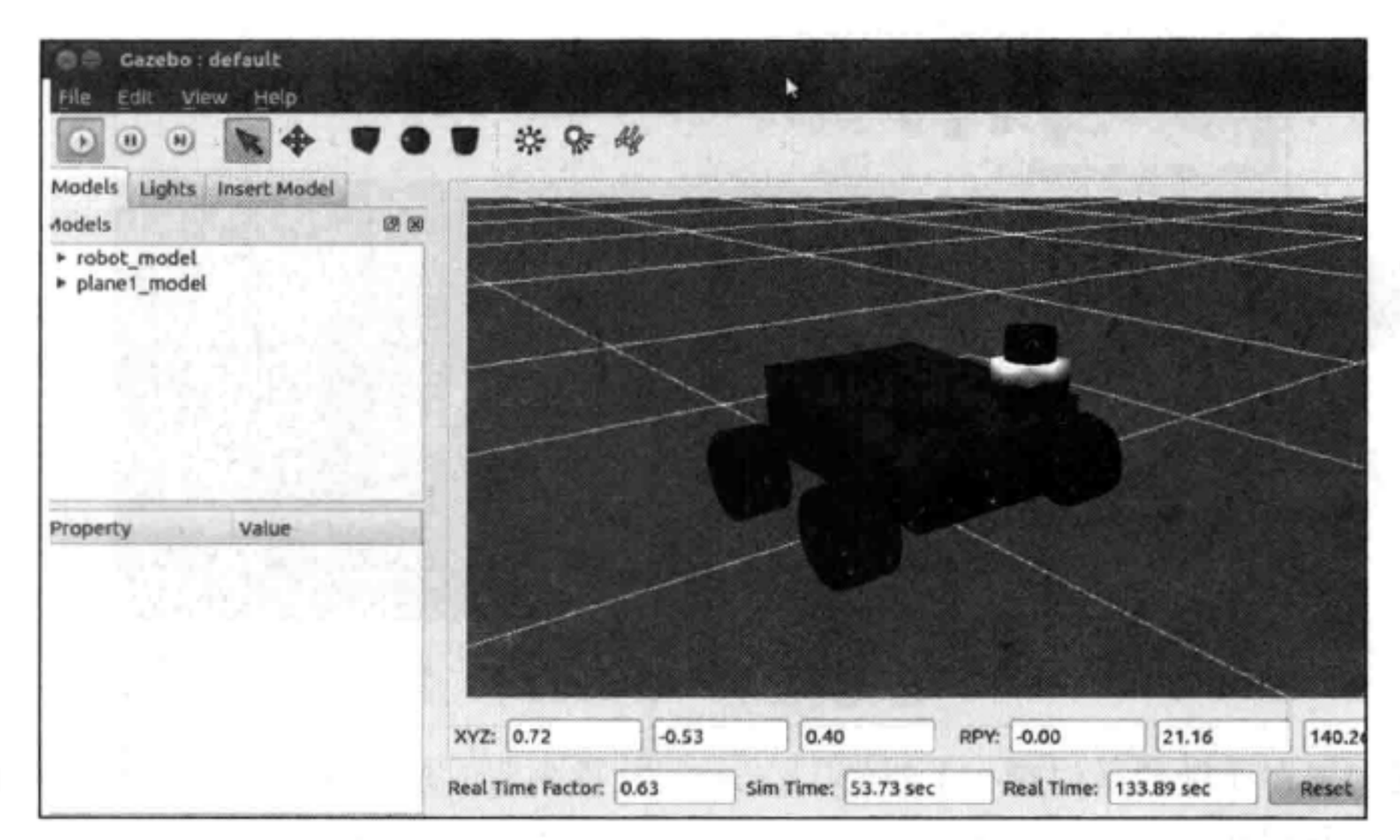
我们唯一需要做的是向 `.xacro` 文件中增加这些行：

```
<include filename="$(find erratic_description)/urdf/erratic_hokuyo_
laser.xacro" />
<!-- BASE LASER ATTACHMENT -->
<erratic_hokuyo_laser parent="base_link">
  <origin xyz="0.18 0 0.11" rpy="0 0 0" />
</erratic_hokuyo_laser>
```

使用以下命令启动新的模型：

```
$ roslaunch chapter5_tutorials gazebo.launch model:="`rospack find
chapter5_tutorials`/urdf/robot1_base_03.xacro"
```

你将会看到附带有雷达模块的机器人，如下图所示：



注意，这个激光雷达会像真实的雷达一样产生“真实”的传感器数据。你能够通过 rostopic echo 命令看到这些数据：

```
$ rostopic echo /base_scan/scan
```

5.4.3 在 Gazebo 中加载和使用地图

在 Gazebo 中，你能够看到如办公室、大山等虚拟环境。

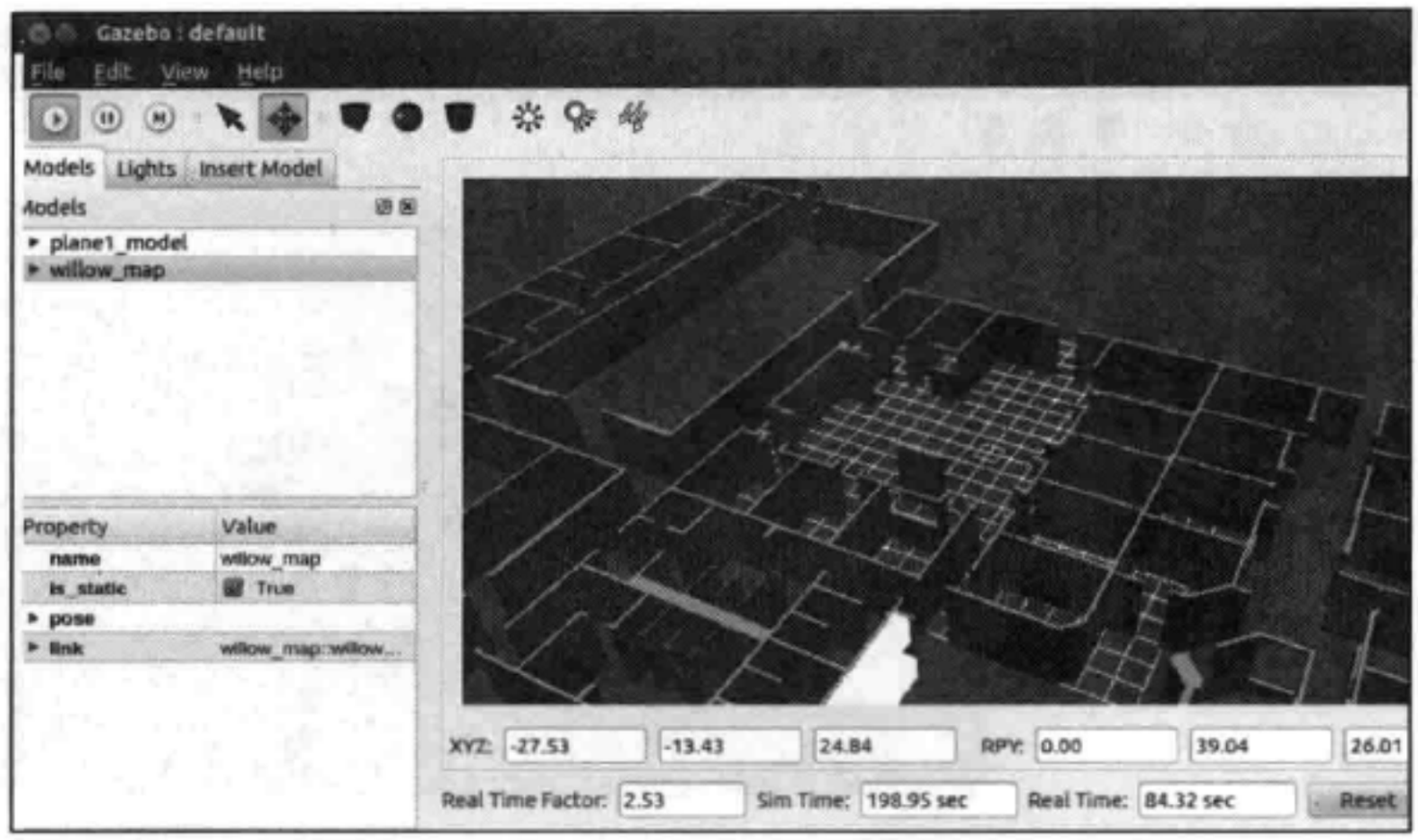
在本小节中我们将会使用一张柳树车库公司办公室的地图，这张地图在我们的 ROS 软件中应该已经默认安装了。

这个 3D 模型保存在 gazebo_worlds 功能包中。如果你没有安装这个功能包，请在进行以下步骤之前安装它。

为了检查模型，你将只需要使用以下命令启动 .launch 文件：

```
$ roslaunch gazebo_worlds wg_collada_world.launch
```

你会在 Gazebo 中看到 3D 的办公室。这个办公室只有墙壁。你可以添加桌子、椅子和其他你想添加的物体。请注意 Gazebo 需要使用性能较好的电脑和处理能力较强的独立显卡。你可以在 Gazebo 官方网站上查找你的显卡是否支持。同时，请注意有些时候会出现软件崩溃，当然 ROS 社区已经为使软件更加稳定不断地付出了大量的努力。通常情况下，你可以在它崩溃之后再次启动它（可能会重复多次）。如果这个问题始终出现，那么我们的建议是升级为最新的版本。最新版本会在较新的 ROS 分发包中默认安装。



我们现在要做的是创建一个新的 .launch 文件来同时加载地图和机器人。为实现这个功能，我们在 chapter5_tutorials/launch 文件夹下创建一个名为 gazebo_map_robot.launch 的文件，并添加以下代码：

```
<?xml version="1.0"?>
<launch>
  <!-- this launch file corresponds to robot model in ros-pkg/robot_
descriptions/pr2/erratic_defs/robots for full erratic -->

  <param name="/use_sim_time" value="true" />

  <!-- start up wg world -->
  <include file="$(find gazebo_worlds)/launch/wg_collada_world.
launch"/>
  <arg name="model" />
  <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />

  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
  <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="state_publisher"
name="robot_state_publisher" output="screen" >
   <param name="publish_frequency" type="double" value="50.0" />
  </node>

  <node name="spawn_robot" pkg="gazebo" type="spawn_model" args="-urdf
-param robot_description -z 0.1 -model robot_model" respawn="false"
output="screen" />
</launch>
```

现在运行带有激光雷达的模型文件：

```
$ roslaunch chapter5_tutorials gazebo_map_robot.launch  model:="`rospack
find chapter5_tutorials`/urdf/robot1_base_03.xacro"
```

你会在 Gazebo GUI 中看到机器人和地图。下一步是在仿真虚拟环境中命令机器人移动和接收其传感器的仿真读数。

5.4.4 在 Gazebo 中移动机器人

差分轮式机器人是一种对机身两侧轮子分别进行驱动的移动机器人。它通过将两侧轮子控制在不同的转速（所产生的转速差）进行转向，而不需要轮子有任何转向运动。

正如前面所说，在 Gazebo 中我们需要对机器人、关节、传感器等设备的行为进行编程。和前面的激光雷达一样，Gazebo 也已经有了差分驱动的实现，我们能够直接使用它移动机器人。

在继续之前，本节需要使用 erratic_robot 功能包集。本示例将在 Gazebo 中使用 erratic_robot 功能包集中的差分不定式机器人驱动（控制器）来移动机器人模型。为了安装功能包集，需要使用以下命令：

```
$ sudo apt-get install ros-fuerte-erratic-robot
```

差分驱动通常意味着机器人有 2 个驱动轮，而我们的机器人有 4 个轮子。因此在运动中会出现问题，所以要进行一些修改。

无论如何，我们都会让机器人能够使用这个驱动。因此选择比较简单的修改方法。事实上，我们可以在差分驱动的代码基础上将其修改成 4 个轮子。

想要使用这个控制器，你只需要将以下代码添加到模型文件中去：

```
<gazebo>
<controller:diffdrive_plugin name="differential_drive_controller"
plugin="libdiffdrive_plugin.so">
      <alwaysOn>true</alwaysOn>
      <update>100</update>
      <updateRate>100.0</updateRate>
      <leftJoint>base_to_wheel4</leftJoint>
      <rightJoint>base_to_wheel1</rightJoint>
      <wheelSeparation>0.33</wheelSeparation>
      <wheelDiameter>0.1</wheelDiameter>
      <torque>5</torque>
      <interface:position name="position_iface_0"/>
      <topicName>cmd_vel</topicName>
</controller:diffdrive_plugin>
</gazebo>
```

你在代码中能够看到的参数都是一些简单的配置，以便这个控制器能够支持 4 个轮子的机器人工作。

例如，我们选择 base_to_wheel4 和 base_to_wheel1 关节作为机器人的驱动轮。记住我们有 4 个轮子，而控制器只需要 2 个就够了。

另外一个有趣的参数是 topicName。我们需要以这个参数为名发布命令来控制机器人。

还需要在模型的起始处添加另一个连接。之前的第一个连接是 base_link，现在第一个连接是 base_footprint。

这个控制器会使这些连接之间进行转换，并且它会创建另一个以 odom 为名的连接。这

个 odom 连接将会在后面导航堆栈的章节中使用：

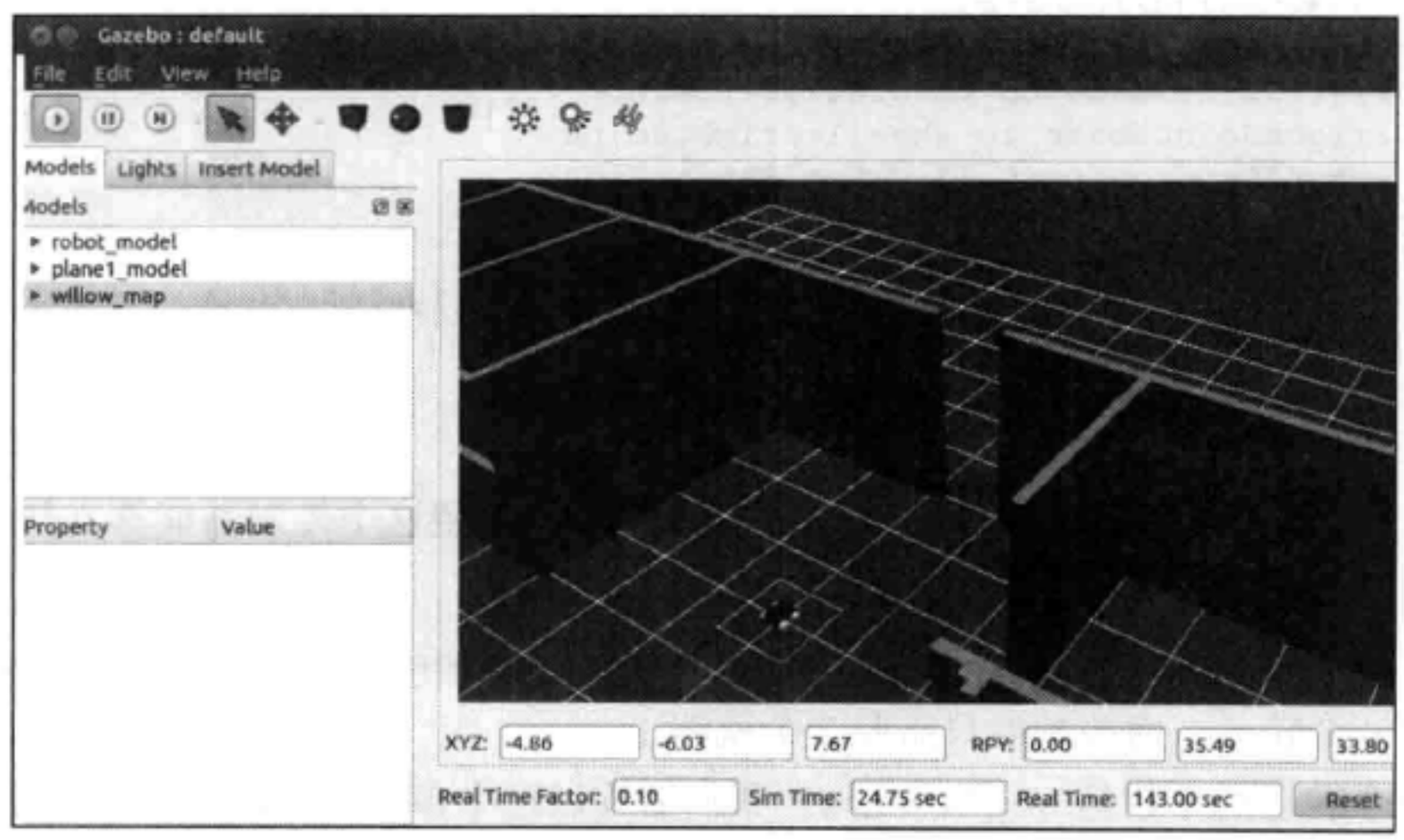
```
<link name="base_footprint">
  <visual>
       <geometry>
             <box size="0.001 0.001 0.001"/>
       </geometry>

          <origin rpy="0 0 0" xyz="0 0 0"/>
  </visual>
  <xacro:default_inertial mass="0.0001"/>
</link>

<gazebo reference="base_footprint">
  <material>Gazebo/Green</material>
    <turnGravityOff>false</turnGravityOff>
</gazebo>

<joint name="base_footprint_joint" type="fixed">
   <origin xyz="0 0 0" />
   <parent link="base_footprint" />
   <child link="base_link" />
</joint>
```

所有这些修改都保存在 chapter5_tutorials/urfd/robot1_base_04.xacro 文件中。

现在，我们使用以下命令启动带有控制器和地图的模型：

```
$ roslaunch chapter5_tutorials gazebo_map_robot.launch model:="`rospack
find chapter5_tutorials`/urdf/robot1_base_04.xacro"
```

你将会在 Gazebo 屏幕中看到机器人在地图中：

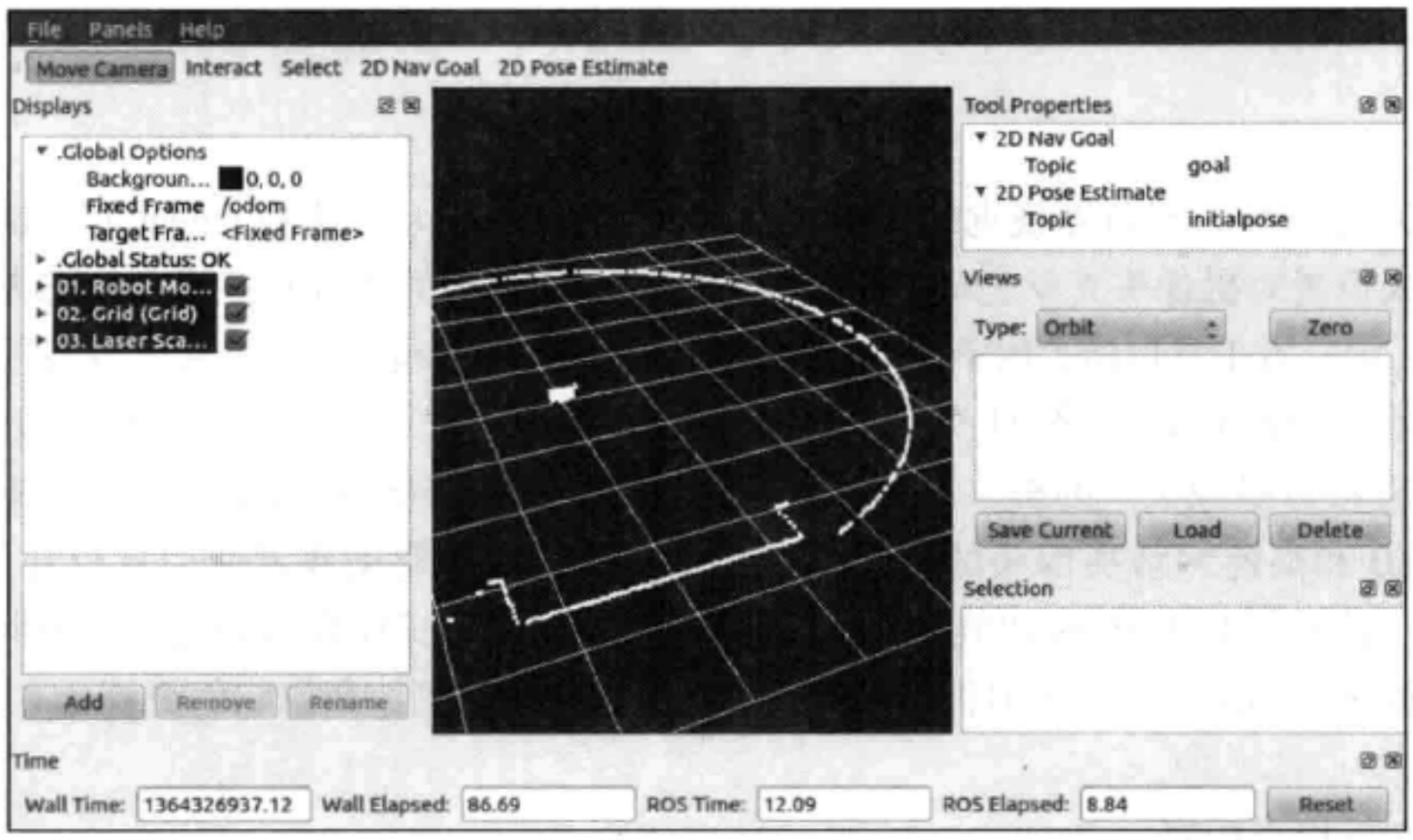


现在我们将使用节点来移动那个机器人。这个节点在 erratic_teleop 功能包中。运行以下命令：

```
$ rosrun erratic_teleop erratic_keyboard_teleop
```

现在你会看到一个运行了很多命令的新命令行窗口，你可以使用 W/A/S/D 按键来移动机器人。

如果现在启动 rviz，你将会在屏幕中看到 3D 模型和激光雷达数据。如果移动机器人，那么你也会在 rviz 中看到机器人在移动。



5.5 本章小结

对于学习机器人的朋友来说，能够接触到实际的机器人是非常有趣也是非常有帮助的，但并不是每个人都有机会。所以会有仿真工具存在。

在本章中，你学会了如何创建一个属于自己的 3D 机器人模型。这包括对创建节点、添加纹理和说明如何使用节点来移动机器人等工作的指导和详细解释。

然后我们介绍了 Gazebo，这是一个能够加载机器人的 3D 模型并对其运动和环境感知进行仿真的仿真环境。这个仿真环境被 ROS 社区广泛使用并且它能够在仿真环境中支持很多种真正的机器人。

第 6 章
机 器 视 觉

在 ROS 系统中，对机器视觉的支持是通过添加摄像头驱动，集成 OpenCV 算法库，集成基于机器人位置对摄像头光学坐标系进行坐标系变换（tf）的工具和一系列第三方工具实现的。其中，第三方工具包括了用于视觉测量、增强现实、物体检测和智能感知等各种算法。

ROS 在处理视觉任务时提供给我们首要的能力是管理 FireWire (IEEE1394a or IEEE1394b) 接口摄像头。的确，在 ROS 的主框架内的功能包就包含了相应摄像头的驱动。而对于 USB 和高速网络摄像头的情况，则必须安装第三方驱动或者使用自行开发的驱动。在本章中，我们将列出 ROS 功能包中集成的 USB 驱动，并提供带有 OpenCV 视频捕捉 API 的驱动源程序。USB 摄像头不但价格低廉，而且也很容易买到，所以它们更符合本书的目标：学以致用。

本章包括以下内容：

- 首先，你会学习如何在 ROS 系统中连接和运行摄像头，先从 ROS 系统提供官方支持的 FireWire 摄像头开始。然后我们将会学习使用不同的配置策略来运行 USB 摄像头。USB 摄像头更容易买到也更便宜。还有，我们会介绍如何使用 USB 摄像头相关的 OpenCV 库安装摄像头驱动。

OpenCV 可能是世界上使用最广泛的开源机器视觉库。ROS 采用了其最新稳定版本的代码（或者取决于你系统中安装的 Debian 软件包内的 OpenCV 库版本）。因此我们可以在节点中轻松地使用它。为了能够订阅和发布图像，OpenCV 图像格式需要通过 cv_bridge 和 image_transport 功能包转换为 ROS 图像。这些功能包的 API 函数会在本章中进行介绍。

- USB 摄像头驱动通常是最常用的 OpenCV 的 API 了。我们将会学习如何使用 cv_bridge 功能包对 ROS 图像进行图像捕捉，并对具体方法进行解释。类似地，我们将会介绍图像转换 API。这个 API 能够以多种格式发布图像，并可选进行或不进行压缩。还有，我们将会学习如何配置摄像头的参数，其中有些特性是专用于支持 FireWire 摄像头的。
- 通过 ROS 中的 OpenCV，我们能够应用大量的机器视觉和机器学习算法，或者运行

一些已经在 OpenCV 库中包含的算法和示例。在这里，我们将不会深入探讨 OpenCV API，因为这超出了本书的范围。但是，我们会建议读者阅读一下在线文档 (http://docs.opencv.org) 或者一些关于机器视觉和 OpenCV 的书籍。需要记住的是你能够在 ROS 节点中使用它们。而我们在本书中要解释的就是如何在你的功能包和节点中调用 OpenCV。

- 我们将会使用在第 3 章中介绍过的可视化工具。我们需要参考 `image_view` 功能包和单目 / 双目视觉。需要特别注意的是双目视觉，因为它能够帮助正确配置图像视差算法。然后我们在 rviz 中查看 3D 点云数据。这些数据是通过标定后的双目视觉和视差图像获取的。
- 还有要注意的是摄像头标定。标定的结果会包含在特定的带有摄像头信息的消息中。有一些工具能够帮助我们使用不同类型的标定模式和内核中的标定优化引擎进行标定。然后，我们会介绍双目摄像头，以及如何管理那些比双目摄像头复杂得多的带有两个甚至更多摄像头的装置。

  请注意，在第 3 章中我们曾经简要地介绍过如何标定摄像头，但是我们将会进一步介绍这一切是如何集成在系统中的。例如，双目视觉将会让你从某种程度上或某个范围内获取环境的深度信息。因此，我们将会看到如何以点云的形式来查看这些信息，如何让摄像头能够获得最佳的效果和质量。
- ROS 带有的图像管道能简化摄像头获取的源图像（RAW image）转换到单色（灰度）图像和彩色图像的转换过程。这在某些时候意味着如果图像被编码为拜耳模式，那么就需要对源图像进行去拜耳化（de-Bayer）。这对于高质量的 FireWire 摄像头来说非常常见。如果摄像头已经被标定过，那么这些标定信息会被用于修正图像。修正图像指的是使用在标定过程中计算出的失真系数对图像扭曲进行更正。最后，对于双目图像，因为我们有左右摄像头之间的基线，所以能够计算出允许我们获取深度信息的视差图像，并在调整之后获得 3D 点云图像。这里我们将会给出一些调整图像的建议，当然这可能对于低精度摄像头来说比较困难，而且有时候要以获得较高的标定结果为前提。
- 这里还会提供 ROS 中一些带有强大算法的功能包集或单独的功能包，它们都能实现一些有趣的机器人任务。我们会列举其中一些任务，并展示一个视觉测距的示例。特别地，在本章中通过使用一对固定在标定板上的网络摄像头组成的双目视觉摄像头组，完成对 libviso2 视觉测距（Visual odometry）库中 viso2_ros 的安装和运行的示例。我们也会提到其他视觉测距库，例如 fovis，还有关于使用它们的一些建议；如何提高 RGB-D 传感器的采集结果，如 Kinect；还有传感器融合或在单目视觉方面的一些信息。

注意本章中的大部分工具都比较新，你可能需要使用 ROS Groovy 和 catkin 风格的编译系统。我们将会在第 6.6 节中进行介绍。由于视觉测距需要使用非常好的摄像头，我们还使用了一些网上能找到的图片做成了一个 demo，方便大家的学习。

6.1 连接和运行摄像头

我们将会从头开始学习。要完成的第一步是将摄像头与电脑连接，运行驱动并检查是否能在 ROS 中获取图像。其实在使用 ROS 之前，最好是使用其他工具来检查摄像头是否被操作系统正确识别，在这里可以使用 Ubuntu distro。我们将从 FireWire 摄像头讲起，因为 ROS 系统对它的支持最好，然后再学习 USB 摄像头。

6.1.1 FireWire IEEE1394 摄像头

将你的摄像头连接到电脑上。你的电脑需要有 FireWire IEEE1394a 或者 IEEE1394b 插槽（你可能需要在你的笔记本上安装一个 IEEE1394 卡）。然后在 Ubuntu 中，你只需要 Coriander 来检查摄像头是否被识别和工作。如果你还没安装它，先安装 Coriander。然后再运行（在老的 Ubuntu 发行版中，你可能需要使用 sudo 来运行）：

```
coriander
```

它会自动检测你的 FireWire 摄像头，如下图所示：

使用 Coriander 最大的好处在于它允许我们查看图像并配置摄像头。事实上，我们建

议使用 Coriander 的摄像头配置界面并在 ROS 中使用这些配置参数。我们会在后面看到如何做。这种方法的好处是 Coriander 能够为我们提供不同维度的参数值，而且有些参数在 ROS 下基本无法进行调试，例如 gamma，它们必须在 Coriander 中预先设置以便作为默认参数启动。

现在我们知道了摄像头是在工作的。关闭 Coriander 并使用以下命令运行 ROS 驱动：

```
rosrun camera1394 camera1394_node
```

只需要先运行 roscore 并运行上面的指令即可。它会在总线上启动第一个摄像头，但请注意你能够通过设定它的 GUID 来选择使用哪个摄像头。你能够在 Coriander 的 GUI 上看到摄像头的 GUID。 为了能够通过 GUID 来选择摄像头，设定其工作模式和各种工作参数，我们需要使用参数文件。因为在这些参数中很多并不需要每次都在命令行中一行行设定。而且这也是一种方便维护的方法，能够把各种参数集成到 .launch 文件中。

FireWire 摄像头的支持参数在 params/firewire_camera/format7_mode0.yaml 文件中赋值，如下所示：

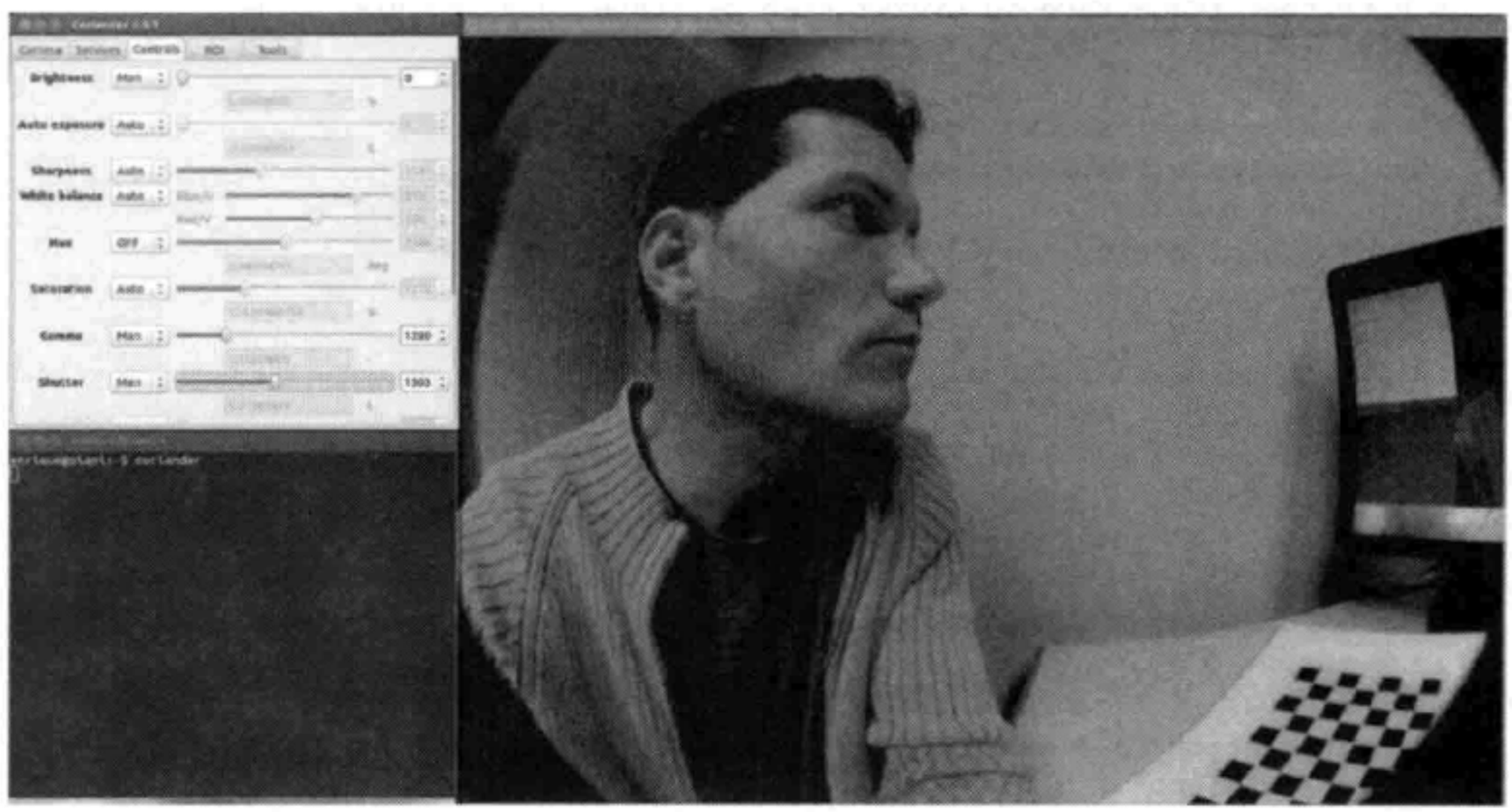
```
guid: 00b09d0100ab1324        # (defaults to first camera on bus)
iso_speed: 800  # IEEE1394b
video_mode: format7_mode0  # 1384x1036 @ 30fps bayer pattern
# Note that frame_rate is overwritten by frame_rate_feature; some
useful values:
# 21fps (480)
frame_rate: 21 # max fps (Hz)
auto_frame_rate_feature: 3 # Manual (3)
frame_rate_feature: 480
format7_color_coding: raw8  # for bayer
bayer_pattern: rggb
bayer_method: HQ
auto_brightness: 3 # Manual (3)
brightness: 0
auto_exposure: 3 # Manual (3)
exposure: 350
auto_gain: 3 # Manual (3)
gain: 700
# We cannot set gamma manually in ROS, so we switch it off
auto_gamma: 0 # Off (0)
#gamma: 1024    # gamma 1
auto_saturation: 3 # Manual (3)
saturation: 1000
auto_sharpness: 3 # Manual (3)
sharpness: 1000
auto_shutter: 3 # Manual (3)
#shutter: 1000 # = 10ms
shutter: 1512 # = 20ms (1/50Hz), max. in 30fps
auto_white_balance: 3 # Manual (3)
```

```
white_balance_BU: 820
white_balance_RV: 520
frame_id: firewire_camera
camera_info_url: package://chapter6_tutorials/calibration/firewire_
camera/calibration_firewire_camera.yaml
```

这些值是通过对所采集的图像进行观察而来。例如，在 Coriander 中通过设定这些值获得更好的图像效果。每个摄像头都有唯一的 GUID 参数，它可以用于选择摄像头。你应该将快门速度设定为与你房间内的供电频率（例如在欧洲和美国分别是 50Hz 和 60Hz）相等或整数倍。这应该是和传统的灯泡或者荧光灯的频率相同。如果是在有阳光的户外，你可能会担心的是较高的快门速度会造成曝光不足。你可以提高增益，但这样会引入噪声。然而，总体来说即使图像上出现那些椒盐噪声也比降低快门速度（以接收更多的光线）要好得多。因为在快门速度较低时，你采集的运动图像会变得很模糊，而大部分的算法对于模糊图像的处理效果非常不好。正如你将看到的，你的配置取决于你所处环境的光线条件，而且你必须去适应它们。如果使用 ROS Fuerte `reconfigure_gui` 界面还是比较容易做到的：

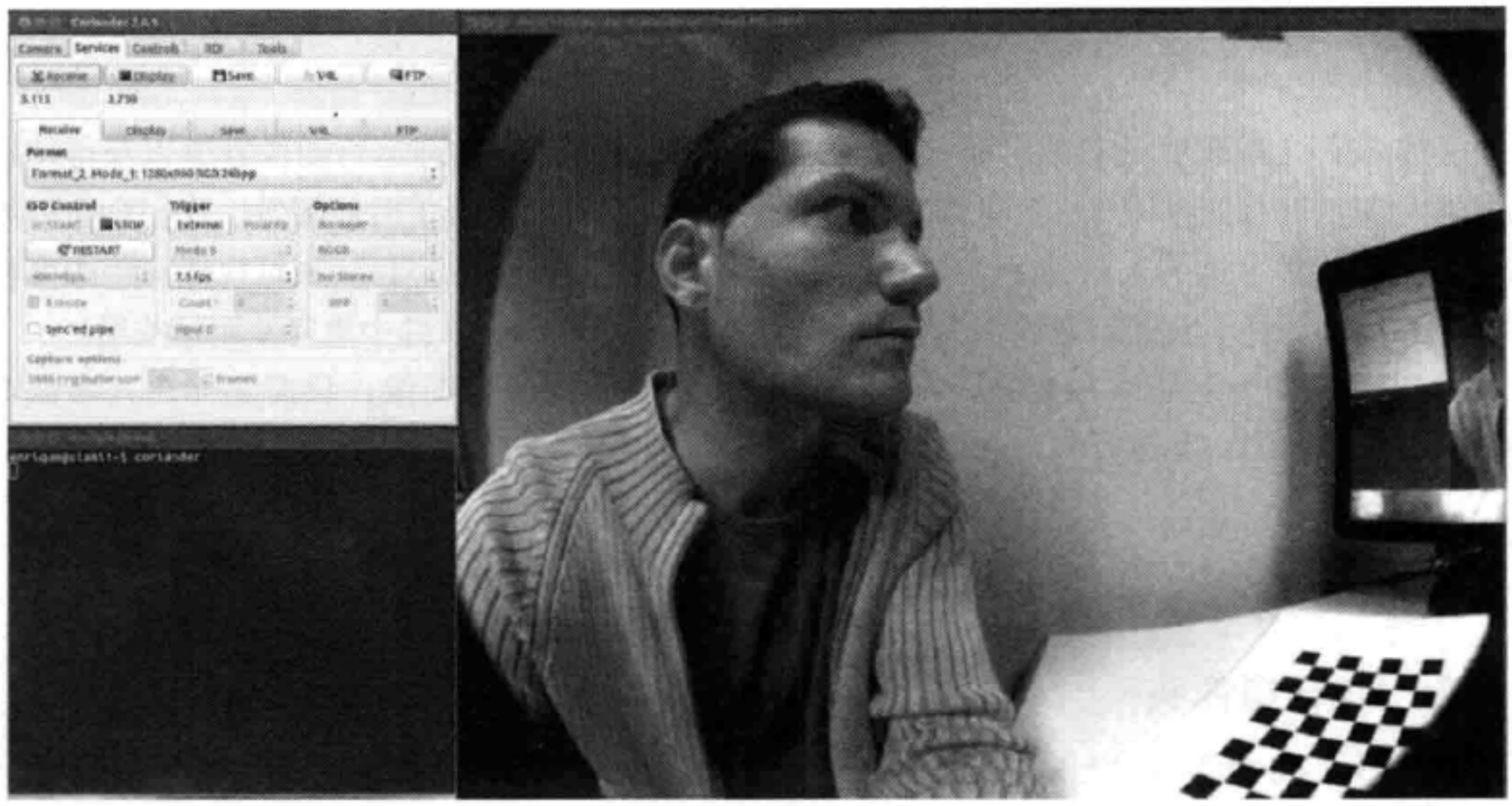
```
rosrun dynamic_reconfigure dynamic_reconfigure_gui /camera
```

在下图中，我们能看到在 Coriander 中的特定配置和输出图像：



在下图中，我们会看到不同的配置能够带来更好的曝光结果：

我们假定摄像头的命名空间是 `/camera`。然后更改其所有的参数，如在第 3 章中介绍的这些参数在 `camera1394 CFG` 文件中被指定。在这里为了使用方便，以 `firewire_camera.launch` 为名创建一个 `.launch` 文件并添加以下代码，将其放置在 `launch` 文件夹下：




```
<launch>
  <!-- Arguments -->
  <!-- Show video output (both RAW and rectified) -->
  <arg name="view" default="false"/>
  <!-- Camera params (config) -->
  <arg name="params" default="$(find chapter6_tutorials)/params/
firewire_camera/format7_mode0.yaml"/>

  <!-- Camera driver -->
  <node pkg="camera1394" type="camera1394_node" name="camera1394_
node">
    <rosparam file="$(arg params)"/>
  </node>

  <!-- Camera image processing (color + rectification) -->
  <node ns="camera" pkg="image_proc" type="image_proc" name="image_
proc"/>

  <!-- Show video output -->
  <group if="$(arg view)">
    <!-- Image viewer (non-rectified image) -->
    <node pkg="image_view" type="image_view" name="non_rectified_
image">
      <remap from="image" to="camera/image_color"/>
    </node>

    <!-- Image viewer (rectified image) -->
    <node pkg="image_view" type="image_view" name="rectified_image">
      <remap from="image" to="camera/image_rect_color"/>
    </node>
```

```
  </group>
</launch>
```

这会按照之前配置的参数启动 camera1394 驱动。然后，它会运行图像管道以获得使用去拜耳化算法和标定参数（一旦摄像头已经被标定过）进行颜色修正后的图像。我们会在后面代码中进一步学习这一点。最后，建立条件图像组并使用 image_view 来可视化彩色图像和颜色修正后的图像。我们就能够在本书所附代码中其他 .launch 文件中重复使用这些组。

总结一下，为了能够在 ROS 中运行 FireWire 摄像头和查看图像，只需要你在其参数文件中设定好 GUID，并运行以下命令：

```
roslaunch chapter6_tutorials firewire_camera.launch view:=true
```

然后，你还能够使用 reconfigure_gui 动态地配置它。

6.1.2 USB 摄像头

现在我们将会学习对 USB 摄像头进行相同的配置。令人惊讶的是，唯一的问题是 ROS 的原生功能包并不支持它们。首先，如果你已经将摄像头和电脑连接，可以先在在线聊天或视频会议软件中测试一下，例如 Skype 或者 Cheese。摄像头资源应在 /dev/videoX 下，其中的 X 是从 0 开始的序号 (如果你用的是笔记本，那么它可能是内置的摄像头)。

值得一提的是有两个主要的选项能用于 USB 摄像头驱动。首先，我们有 usb_cam。如果想安装它，需要运行以下命令：

```
svn co http://svn.code.sf.net/p/bosch-ros-pkg/code/trunk/stacks/bosch_
drivers/usb_cam
rosstack profile && rospack profile
roscd usb_cam
rosmake
```

在这之后，你能运行以下命令：

```
roslaunch chapter6_tutorials usb_cam.launch view:=true
```

运行 rosrun usb_cam usb_cam_node，并使用 image_view 显示摄像头图像，你会看到类似于下图所示。它本身就是 USB 摄像头的原始图像，而且本身就是彩色的。

类似地，其他好的选择还有 gscam，请使用以下命令安装：

```
$ svn co http://brown-ros-pkg.googlecode.com/svn/trunk/experimental/gscam
$ echo 'include $(shell rospack find mk)/cmake.mk' > Makefile
rosmake
```

请注意，我们需要修改 Makefile 文件，因为它是错误的（这是 ROS 功能包中所有 Makefile 文件的通病，好像是软件源中某种通用的忽视规则将其忽视了）。然后只需要运行以下命令：

```
roslaunch chapter6_tutorials usb_cam.launch view:=true
```

对于 usb_cam，这个 .launch 文件运行了 rosrun gscam gscam，并且对摄像头的一些参数进行配置。它还使用 image_view 显示了摄像头图像，如下图所示：

由 gscam 获取的参数（查看 params/gscam/logitech.yaml）如下：

```
gscam_config: v4l2src device=/dev/video0 ! video/x-raw-rgb,framerate=30/1
! ffmpegcolorspace
frame_id: gscam
camera_info_url: package://chapter6_tutorials/calibration/gscam/
calibration_gscam.yaml
```

配置命令 gscam_config 会使用合适的参数调用 v4l2src 命令来驱动摄像头。一旦摄像头完成标定并在 ROS 的图像管道中使用，其余的参数也会是非常重要的。

6.2 使用 OpenCV 制作 USB 摄像头驱动程序

虽然我们之前已经提到了两种方法，但本书还是会提供我们自己的 USB 摄像头去驱动。驱动是通过调用 OpenCV 的 cv::VideoCapture 类来实现的。它能够驱动摄像头并且允许我们去修改它的一些本书所支持的参数。这非常的简单而且它允许我们像配置 FireWire 摄像头一样去设置标定信息。事实上，使用 usb_cam 不可能实现这些功能，因为 CameraInfo 消息是不可用的。因此，camera_info_url 并不能在参数文件 params/usb_cam/logitech.yaml 中设置。而在 gscam 中，我们可以进行更多的控制。

我们可以在 ROS 中彻底地改变摄像头配置，并查看如何发布摄像头图像和信息。根据摄像头图像的读取方式，我们有两种使用 OpenCV 实现摄像头驱动的方法。第一个是根据每秒钟给定的图像数量（FPS）进行轮询；第二个是我们能够根据 FPS 周期设定一个计时器，并在计时器的回调函数中完成实际的读取工作。由于我们的方法依赖于 FPS，因此在 CPU 的消耗上要比其他方法好。无论如何，当 OpenCV 读取功能被阻断时，是无法激活轮询的，而且在图像准备好之前，其他进程会占据 CPU。通常情况下，在 FPS 较快时，最好使用轮询的方法，这样我们无需承担使用计时器和回调函数所造成的时间损失。在 FPS 较慢时，计时器方法和轮询方法就差不多了，而前者的代码更清晰明了。我们建议读者能够比较这两种实现方式，它们分别存储在 src/ camera_polling.cpp 和 src/camera_timer.cpp 文件中。为了节省大家的时间，我们在这里介绍基于计时器的实现方法。在 src/camera.cpp 文件中最终实现的驱动也使用了计时器方法。请注意在最终的驱动程序中还包括了我们将在后面章节中介绍的摄像头信息管理。

6.2.1 创建 USB 摄像头驱动功能包

在 manifest.xml 文件中，我们必须设置 OpenCV、ROS Image 消息库和相关功能包的依赖项。如下所示：

```
<depend package="sensor_msgs"/>
<depend package="opencv2"/>
<depend package="cv_bridge"/>
<depend package="image_transport"/>
```

因此，在 src/camera_timer.cpp 文件中我们要包含以下头文件：

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/highgui/highgui.hpp>
```

image_transport API 允许我们使用多种传输格式发布图像，其中包括各种压缩图像格式和在 ROS 系统中以插件形式无缝集成的各种编解码器，例如，Theora 压缩。上面的 cv_bridge 字段用于从 OpenCV 图像到 ROS Image 消息的转换，其中在有灰度 / 颜色转换时还会用到 sensor_msgs 进行图像编码。最后为了使用 cv : : VideoCapture 还需要使用 OpenCV(opencv2) 中的 highgui API。

这里我们会对 src/camera_timer.cpp 文件中的大部分代码进行解释，其中有一个实现摄像头驱动的类。它的属性如下：

```
ros::NodeHandle nh;
image_transport::ImageTransport it;
image_transport::Publisher pub_image_raw;
```

```
cv::VideoCapture camera;
cv::Mat image;
cv_bridge::CvImagePtr frame;

ros::Timer timer;

int camera_index;
int fps;
```

6.2.2 使用 ImageTransport API 发布摄像头帧

像往常一样，我们需要节点句柄。然后我们需要一个用于以任何可能格式发送图像的 ImageTransport 对象。在代码中，我们只需要使用发布者，但注意它必须是 image_transport 库的实现，而不是一般用于图像消息的 ros::Publisher。

然后我们使用 OpenCV 来捕捉图像 / 帧。在捕捉帧的时候，我们直接使用 cv_brigde 帧。它是 CvImagePtr 类型的，通过它我们可以直接访问图像字段。

最后，我们有计时器和用于驱动程序工作的基本摄像头参数。这些参数包括摄像头的索引名，也就是 /dev/videoX 设备的编号，例如，0 代表 /dev/video0。这个索引名被传递给 cv::VideoCapture。FPS 用于设定摄像头（如果可能的话，因为有些摄像头不支持这个参数）和计时器。这里我们使用了一个 int 值，但在最终版本的 src/camera.cpp 中它也可能是一个 double 值。

这个驱动会使用用于安装和初始化节点、摄像头、计时器的类构造函数：

```
nh.param<int>( "camera_index", camera_index, DEFAULT_CAMERA_INDEX );

if ( not camera.isOpened() )
{
    ROS_ERROR_STREAM( "Failed to open camera device!" );
    ros::shutdown();
}

nh.param<int>( "fps", fps, DEFAULT_FPS );
ros::Duration period = ros::Duration( 1. / fps );

pub_image_raw = it.advertise( "image_raw", 1 );

frame = boost::make_shared< cv_bridge::CvImage >();
frame->encoding = sensor_msgs::image_encodings::BGR8;

timer = nh.createTimer( period, &CameraDriver::capture, this );
```

首先，我们要打开摄像头，如果无法打开则需要终止程序。注意，我们必须在属性构造函数内完成这些工作：

```
camera( camera_index )
```

这里 camera_index 是作为参数进行传递。

然后，我们读取 fps 参数并计算计时器周期。这些参数用于创建计时器并设定用于图像捕捉的回调函数。我们使用 ImageTransport API 作为 image_raw 图像（源图像）的发布者，并初始化 frame 变量。

下面代码是用于图像捕捉的回调函数读取和发布图像：

```
camera >> frame->image;
if( not frame->image.empty() )
{
    frame->header.stamp = ros::Time::now();
    pub_image_raw.publish( frame->toImageMsg() );
}
```

它捕捉图像，并检查帧是否被实际捕捉。在这种情况下，设定时间戳并发布已经被转换成 ROS Image 的图像。

你可以使用以下代码启动这个节点：

```
rosrun chapter6_tutorials camera_timer camera_index:=0 fps:=15
```

我们将会以 15FPS 的速度启动 /dev/video0 摄像头。

然后你能使用 image_view 查看图像。和轮询的实现方式类似的是，你有一个 .launch 文件，并能够使用以下命令运行：

```
roslaunch chapter6_tutorials camera_polling.launch camera_index:=0
fps:=15 view:=true
```

现在，你能看到 /camera/image_raw 主题的图像。

在计时器的实现方式中，最终的实现中包含 camera.launch 文件，并提供更多的选项。这些选项需要贯穿本章的全部内容。在最终的实现中，最大的改进是能够支持动态参数再配置，也就是说，它能够提供包括摄像头标定在内的摄像头信息在线配置功能。我们会简要说明这是如何实现的，并建议读者自行查看源代码来进一步学习完整的代码设计。

和 FireWire 摄像头类似，我们能够对摄像头参数的动态再配置提供支持。然而，大多数的 USB 摄像头并不支持对部分参数的修改。我们所做的主要针对所有 OpenCV 支持的参数，并在发生错误（或部分参数失效）时向用户提示警告信息。配置文件是 cfg/Camera.cfg。请查看文件以了解详细内容。它支持这些参数：

- camera_index　用于选择 /dev/videoX 设备。
- frame_width 和 frame_height　用于提供图像的分辨率。
- fps　用于摄像头的 FPS 值。
- fourcc　表示 FOURCC 的定义格式，用于指定摄像头像素（查看 http://www.fourcc.org，虽然这个标识符所指的格式通常是 YUYV 或者 MJPG，但对于大多数 USB 摄像

头它们无法使用 OpenCV 进行改变)。

- brightness、contrast、saturation 和 hue　这些变量设定摄像头的属性。在数字摄像头中，它们通过软件修改，并在传感器的图像获取过程中进行调整或直接对最终的图像进行调整。
- gain　设定了传感器模数转换 (ADC) 的增压。它会向图像中引入椒盐噪声，也会增加在黑暗环境中图像的亮度。
- exposure　决定了图像的曝光，也就设定了图像的亮度。通常通过调整增压和快门速度（在廉价摄像头中，这就是积分时间内进入传感器的光）进行调整。
- frame_id　用于指定摄像头坐标系，而且在导航功能中这非常有用，同样我们在第 6.6 节中也会用到它。
- camera_info_url　这是一个到摄像头信息的路径，所保存内容主要关于摄像头标定。

然后，在驱动中，我们通过以下语句使用动态再配置服务：

```
#include <dynamic_reconfigure/server.h>
```

在构造函数中设置回调函数：

```
server.setCallback( boost::bind( &CameraDriver::reconfig, this, _1, _2
) );
```

回调函数会再配置摄像头。我们甚至允许改变摄像头或停止当前使用的摄像头，然后使用 OpenCV 的 cv::VideoCapture 类来配置摄像头属性。其中包括了部分前面提到的参数。以 frame_width 参数为例说明：

```
newconfig.frame_width = setProperty( camera, CV_CAP_PROP_FRAME_WIDTH ,
newconfig.frame_width );
```

它依赖于前面提到的 setProperty 方法，它会调用 cv::VideoCapture 中的 set 方法，并控制实例，在失败时发送 ROS 警告信息。

FPS 在计时器中改变，而且在摄像头中通常不能像其他参数那样被修改。

最后，最需要注意的是，所有的再配置过程都受到一个锁定的互斥标记的制约，以避免在获取图像的同时进行驱动的再配置。

为了能设定摄像头信息，ROS 有一个 camera_info_manager 库，能够帮助我们完成这些工作。简而言之，是使用：

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

我们用它来获取 CameraInfo 消息。现在，在计时器中捕获图像的回调函数里，我们能使用 image_transport::CameraPublisher（不只是用于图像）。代码如下所示：

```
camera >> frame->image;
if( not frame->image.empty() )
{
```

```
    frame->header.stamp = ros::Time::now();

    *camera_info = camera_info_manager.getCameraInfo();
    camera_info->header = frame->header;

    camera_pub.publish( frame->toImageMsg(), camera_info );
  }
```

这会在前面提到的再配置方法中所使用的互斥标记的制约下工作。现在，我们为第一版驱动进行了各种参数配置，但在 manager 库中仍然要使用再配置方法（在摄像头加载过程中已经调用过一次）获取摄像头信息，用于设定节点句柄、摄像头名称和 camera_info_url 参数。然后我们将图像 / 帧（ROS Image）和 CameraImage 消息都发布出去。

为了能使用驱动，运行以下命令：

```
roslaunch chapter6_tutorials camera.launch view:=true
```

它将会使用 params/camera/webcam.yaml 作为默认参数，其中设定了到目前为止的所有动态再配置参数。

你能够使用 rostopic list、rostopic hz /camera/image_raw 和 image_view 来检查摄像头是否在工作。

我们使用了 ROS 中所有可用资源来完成驱动程序，以便进行摄像头、图像与机器视觉等工作。为了能讲解得更清晰，我们将会在下面的章节中分别解释每一项工作。

6.2.3 使用 cv_bridge 进行 OpenCV 和 ROS 图像处理

假设我们有一个 OpenCV 图像，那么它会是一个 cv::Mat image。我们需要使用 cv_bridge 库将其转换为 ROS Image 消息并将其发布。我们可以选择分别使用 CvShare 或 CvCopy 去共享图像或者复制图像。然而，如果可能的话，通过 cv_bridge 在 CvImage 类中使用 OpenCV image 字段更容易。这正是我们做摄像头驱动时的做法，将其作为一个指针：

```
cv_bridge::CvImagePtr frame;
```

对于一个指针，我们应该按照这样的方法进行初始化：

```
frame = boost::make_shared< cv_bridge::CvImage >();
```

如果我们事先知道图像的编码方式：

```
frame->encoding = sensor_msgs::image_encodings::BGR8;
```

最后让我们将 OpenCV image 设为指针，例如，从摄像头对其进行捕捉的时候：

```
camera >> frame->image;
```

在这个指针中直接设置消息的时间戳也很常见：

```
frame->header.stamp = ros::Time::now();
```

现在只需要将其发布。我们需要一个发布者来实现这个功能。这个实现者需要使用 ROS 中的 image_transport API。我们在下一小节中来介绍这些内容。

6.2.4 使用 ImageTransport 发布图像

我们可以使用 ros::Publisher 发布单一图像，但更好的办法是使用 image_transport 发布者。它能够发布单一图像或多个图像，并附带相应的摄像头信息。这和我们之前在摄像头驱动中的设计是一样的。ImageTransport API 很强大的一点在于能够无缝地提供不同的传输格式。你所发布的图像会出现在几个主题中，图像被分为基本图像、未压缩图像、压缩图形等多种类型。所支持的传输类型的数量是由在 ROS 中安装的插件决定的，最常用的是 compressed 和 theora 传输。你能够通过简单的 rostopic call 命令来查看它们。

在你的代码中，你需要使用节点句柄创建图像传输和发布者。在本示例中我们将会使用一个简单的图像发布者。请检查最终的 USB 摄像头驱动中的 CameraPublisher，看一下它的使用方法：

```
ros::NodeHandle nh;
image_transport::ImageTransport it;
image_transport::Publisher pub_image_raw;
```

节点句柄和图像传输通过以下代码构建（在类的属性构造函数中）：

```
nh( "~" ),
it( nh )
```

然后，发布者是通过使用 node 的命名空间和一个 image_raw 主题来实现的：

```
pub_image_raw = it.advertise( "image_raw", 1 );
```

因此，前面章节中展示的 frame 属性现在就能够使用下面的代码声明进行发布了：

```
pub_image_raw.publish( frame->toImageMsg() );
```

6.2.5 在 ROS 中使用 OpenCV

ROS Fuerte 附带着最新稳定版的 OpenCV。在更新的 ROS 分发包中，这取决于你在操作系统中安装的 OpenCV 库中的 Debian 功能包。为了能在我们的节点中使用它，在你的 manifest.xml 文件中输入以下代码片段：

```
<depend package="opencv2"/>
```

在 CMakeLists.xml 文件中，只需要增加一行就能够编译我们的节点。也就是说，我们不需要对 OpenCV 库再做任何修改。

在节点的 .cpp 文件中，我们要包含所需的所有 OpenCV 库。例如，我们使用以下声明来包含 highgui.hpp 文件：

```
#include <opencv2/highgui/highgui.hpp>
```

现在，你能够在你的代码中使用任何 OpenCV API 的类、函数及其他代码段。如果你是一个 OpenCV 的新手，那么你也可以简单地引用它的命名空间和 cv，并学习一下 OpenCV 的自带教程。需要说明的是本书并不是用于介绍 OpenCV 的，仅对 ROS 中的机器视觉功能作介绍。那么在学习过程中也请使用 ROS 功能包中的 rosmake 对程序进行编译。

6.2.6 显示摄像头输入的图像

在第 3 章中，我们介绍了如何在 ROS 框架下显示发布的图像。这是通过 image_view 功能包下的 image_view 主题实现的：

```
rosrun image_view image_view image:=/camera/image_raw
```

在这里最重要的是通过使用图像传输，我们能够选择不同的主题来查看图像，并在必要时使用压缩格式。还有，在双目视觉中，我们使用 rviz 还能查看由差分图像生成的点云。我们会在后续章节中进行介绍。

6.3 如何标定摄像头

大部分摄像头，尤其是广角镜头，会引入巨大的失真。我们可以从径向和切向对这种失真进行建模，并使用标定算法计算模型的参数。这种摄像头标定算法同时还允许我们获得一个带有镜头焦距和主点的标定矩阵，这就为我们提供了一个方法能根据所获取的图像以米为单位测量实际环境。在双目视觉的情况下，它也能帮助我们得到深度信息，也就是，摄像头获取的每个像素之间的距离，我们会在后面再去介绍这部分内容。而最终的结果就是我们获取了真实环境的 3D 信息。

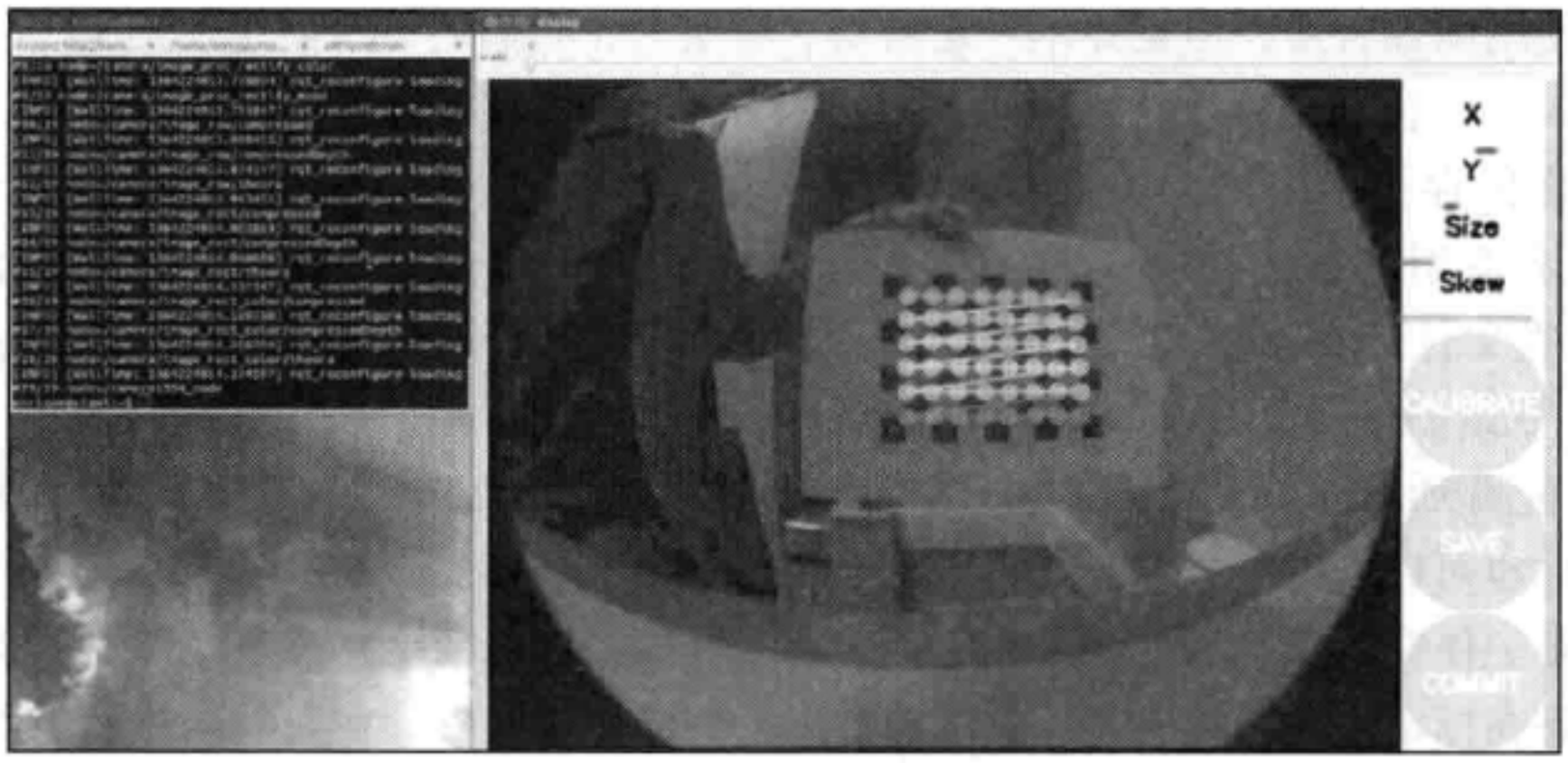
标定是通过使用一种已知的标定图案，并对图案的多种角度的视图进行辨识来实现的。最典型的图案是棋盘，也有使用一列圆圈或者非对称的圆圈作为图案的，但要注意当视图倾斜时，圆圈看起来就变成了椭圆。检测算法会获取棋盘上每个单元格的内角点，并用来估计摄像头的内参数和外参数。简而言之，外参数是摄像头的姿态，或者换句话说，就是在摄像头保持在某个固定位置时，图案相对于摄像头的位姿。我们最需要的是内参数，因为它是不会发生改变的，这就能在摄像头在任何位姿时使用，能够测量图像的距离，修正图像的失真，也就是纠正图像。

运行我们的摄像头驱动，就能够使用 ROS 的标定工具完成摄像头标定。在此过程中，摄像头驱动提供了 CameraInfo 消息和 camera_info_set 服务。这个服务允许设定标定结果保存的文件路径。之后，我们每次使用摄像头时，标定信息就能够被图像管道加载。

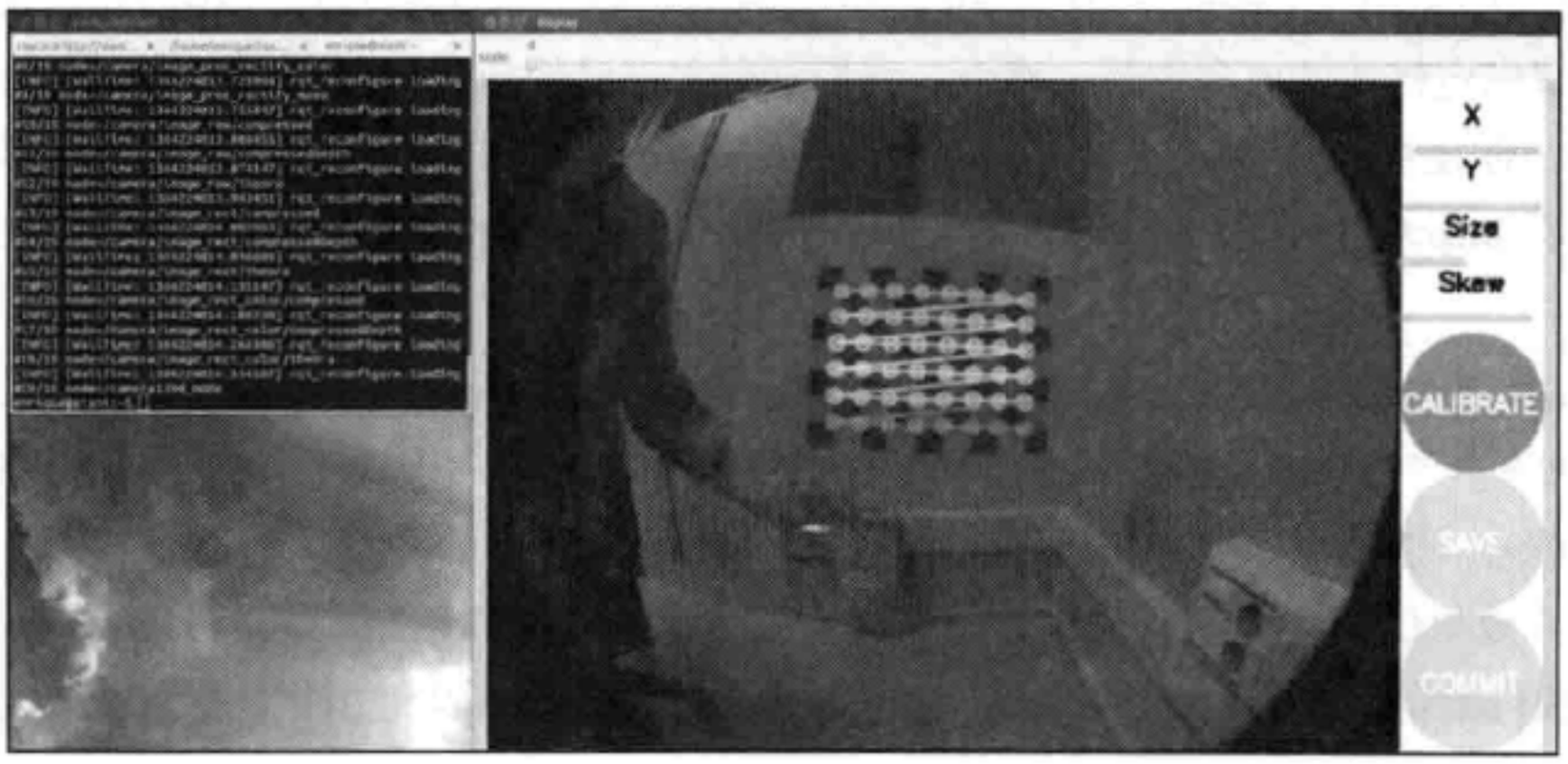
FireWire 摄像头的 camera1394 驱动恰好满足这些先决条件。想要标定你的 FireWire 摄像头，需要运行以下命令：

```
roslaunch chapter6_tutorials calibration_firewire_camera_chessboard.launch
```

这会打开一个 GUI 界面，并自动选择我们标定图案的视图。它在右上角会显示一些指示条来表现在视图中的每一个“轴”，其中包括了 x 轴和 y 轴，或者分别代表横轴和纵轴。显示的这些轴与图像平面上的这些轴表示了相同的图案模式。然后，size 表示我们的标定范围从近到远（直到我们打算进行探测工作的最大距离）。最后是 skew 显示视图中所识别的 x 轴和 y 轴的斜率。在这些指示条下面的三个按钮默认是禁用的，如下图所示：



你能看到每次图案重叠的时候标定算法还是能找到这些点。这些视图会被自动选择并覆盖一定数量的不同视图。这样你在指示的引导下能够让指示条在某个时刻从左到右完全变绿。理论上，有两个视图就足够了，但是实际使用中一般都需要 10 个左右。而这个界面会捕捉更多（30 ～ 40 个）。你应该避免过快的移动速度，因为模糊的图像对于检测来说是非常不利的。一旦这个工具收集了足够的视图，它就会允许你进行标定，也就是说根据标定图案的视图中找到的点对摄像头模型进行求解和系统优化。



然后，你可以保存标定数据并向摄像头提交标定结果。这就需要使用 camera_info_set 服务来提交。在后续使用中 ROS 图像管道就会自动检测结果。

对于使用了标定的 .launch 文件，只需要使用 ROS camera_calibration 功能包中的 cameracalibrator.py 文件，使用以下命令：

```
<node pkg="camera_calibration" type="cameracalibrator.py"
      name="cameracalibrator" args="--size 8x6 --square 0.030"
      output="screen">
  <remap from="image" to="camera/image_color" />
  <remap from="camera" to="camera" />
</node>
```

它也使用图像管道，但是并不一定需要。事实上，我们还可以使用 image_raw 来替代 image_color 主题。

一旦你保存了标定结果（使用 Save 按钮），就会在 temp 文件夹下创建一个文件。它保存了标定使用的标定图案的视图文件。你能够在 /tmp/calibrationdata.tar.gz 中找到这些文件。在本书中用于标定的视图文件存储在 calibration 文件夹下，其中 firewire_camera 是 FireWire 摄像头的第一个实例。类似地，在命令行终端上（stdout 输出），你能看到获取对应视图的信息和标定结果。本书中获取的标定结果保存在同一个文件夹下。你也可以查看 calibrationdata.tar.gz 中的 ost.txt 文件来查询标定结果。无论如何要记住在提交之后，会用标定矩阵和失真模型的系数更新标定文件。实验这个功能的好办法是在标定之前在相应文件夹下放置一个假的标定文件。在我们的功能包中，这个文件是 calibration/firewire_camera/calibration_firewire_camera.Yaml，它由参数文件访问：

```
camera_info_url: package://chapter6_tutorials/calibration/firewire_
camera/calibration_firewire_camera.yaml
```

现在我们使用图像管道再次运行我们的摄像头。如果摄像头被正确标定，那么很明显修正后的图像会修正失真。我们将会在后面介绍图像管道时看到这部分内容。

在 ROS 中采用了 OpenCV 中实现的张正友标定方法。如果有兴趣，我们的建议是查看相关文档了解标定格式的更多细节。然而，我们觉得本书已经包含了用户所应该了解的知识。

最后，你还可以使用下面的 .launch 文件试着应用不同的标定图案，如圆形和不对称圆形（查看 http://docs.opencv.org/trunk/_downloads/acircles_ pattern.png），这都针对 FireWire 摄像头：

```
roslaunch chapter6_tutorials calibration_firewire_camera_circles.launch
roslaunch chapter6_tutorials calibration_firewire_camera_acircles.launch
```

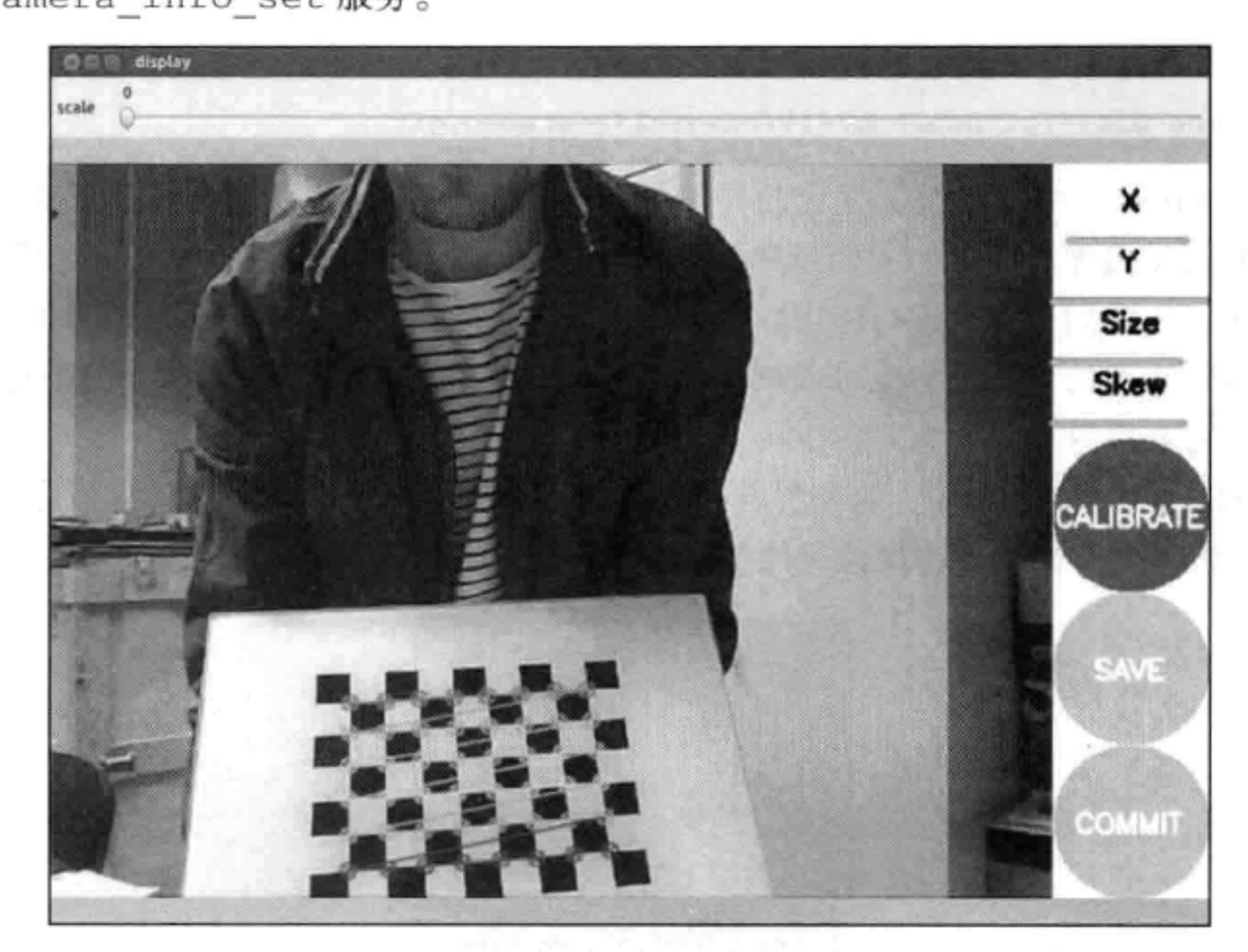
你还能在一次标定中使用多种棋盘图案和使用不同大小的图案。然而，我们认为仅使用一种有较好效果的棋盘图案就已经足够了。事实上，对于 USB 摄像头驱动我们也只这样做。

在 USB 摄像头驱动的情况下，我们有更强大的 .launch 文件来集成摄像头标定节点。这

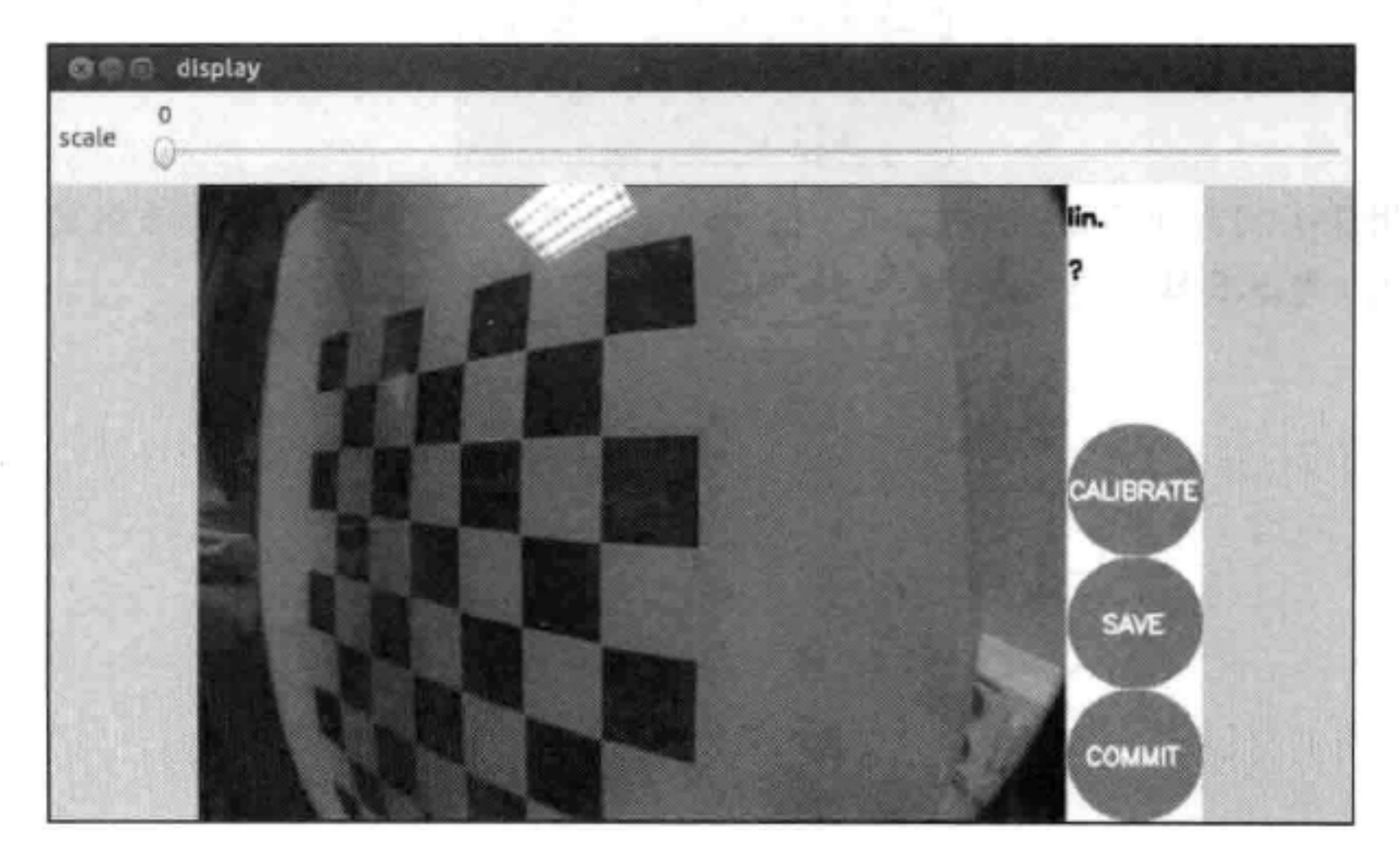
和 FireWire 摄像头一样，也是独立的启动文件。因此，要标定你的摄像头，请使用以下命令：

```
roslaunch chapter6_tutorials camera.launch calibrate:=true
```

在下图中，你会看到一个和 FireWire 摄像头一样的 GUI 中的标定过程，也就是说我们也要运行 camera_info_set 服务。



上图展示的示例中已经获取了足够多的图案视图，这样标定（calibrate）按钮被允许使用。因为这时已经可以求解标定模型了。下图展示了标定过程的结束，允许我们保存标定数据并提交到摄像头配置文件中去，而且我们不需要做更多的设置。



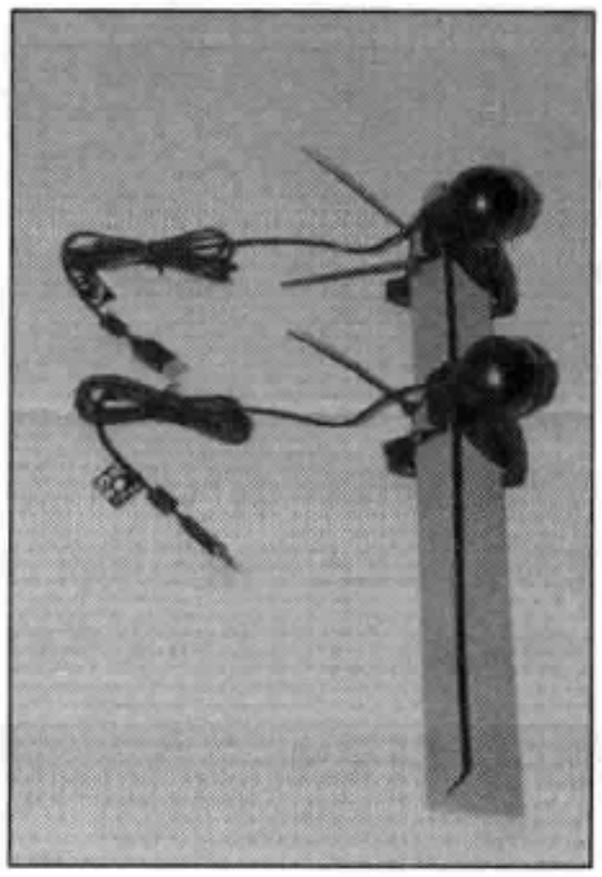
双目标定

在下面的章节中我们将会介绍如何使用双目摄像头。一个方法是运行两个单目摄像头节点。但通常情况下将双目摄像头组作为一个传感器来处理会更好，因为图像必须是同步的。在 ROS 中，没有 FireWire 双目摄像头专用的驱动，但是我们能够从下面找到一个双目应用的扩展：

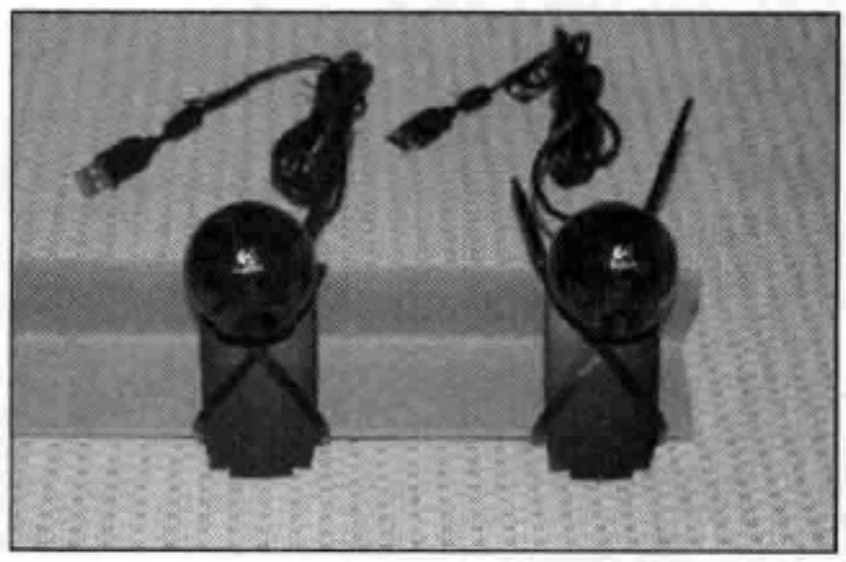
```
git clone git://github.com/srv/camera1394stereo.git
```

然而，FireWire 的双目视觉是比较昂贵的。基于这个原因，我们提供一个基于 USB 摄像头的双目视觉摄像头驱动。我们使用罗技的 C120 USB 摄像头，它的价格相对低廉。它的噪声也很大，但我们可以在进行标定后看看是不是能改善一下。保证双目视觉中的两个摄像头一致是非常重要的，但是你也可以试着使用不同的摄像头。我们安装的两个摄像头如下图所示。你只需要将两个摄像头安装在同一个标定板上。

我们的基线大约是 12cm，这个参数在双面摄像头标定过程中也要被计算进去。正如你所见，你只需要一个固定两个摄像头的标定板，并用扎带扎好，例如：



一个摄像头的背面照和正面照，它们能够手动调整焦距。请注意你应该将其设定到一个合适的距离（考虑到基线的长度，1 ～ 2m 应该能满足大多数情况的需要）。



现在把摄像头和你的 USB 插槽连起来。最好养成习惯每次都先插左边的摄像头后插右边的摄像头。这样他们就总能被分配成 /dev/video0 和 /dev/video1。当然，如果 0 被占用，就是 1 和 2。

然后，你能够分别测试每个摄像头，就像使用单一摄像头时一样。你可能需要一些工具如 v4l-utils 和 qv4l2，你可以使用以下命令来安装：

```
sudo apt-get install v4l-utils qv4l2
```

你可能会遇到这样的错误：

```
libv4l2: error turning on stream: No space left on device
```

之所以会发生这样的情况是因为你应该把两个摄像头分别连接到不同的 USB 控制器上。请注意，有些 USB 插口是由同一个控制器管理的。这样在连接多于一个摄像头时就没有足够的带宽处理数据。有几个方法你可以试一下。首先是在两个摄像头上试着使用压缩格式，例如 MJPG。你可以检查你的摄像头是否支持压缩：

```
v4l2-ctl -d /dev/video2 --list-formats
ioctl: VIDIOC_ENUM_FMT
Index       : 0
Type        : Video Capture
Pixel Format: 'YUYV'
Name        : YUV 4:2:2 (YUYV)

Index       : 1
Type        : Video Capture
Pixel Format: 'MJPG' (compressed)
Name        : MJPEG
```

如果支持 MJPG，我们就可以在同一个 USB 控制器下使用多个摄像头。除此之外，使用非压缩格式时，我们就必须使用不同的 USB 控制器，或者将分辨率调低到 320 × 240 及以下。类似地，在 qv4l2 的 GUI 界面下你能够检查这个功能并测试你的摄像头。你还可以试着将其设到理想的像素格式。有时候这样都行不通。还有使用 OpenCV set 方法对 USB 摄像头来说并不起作用。所以我们还是在不同的 USB 控制器下使用不同的 USB 插口。

本书中附带的 USB 双目视觉驱动是基于前面章节介绍过的 USB 摄像头驱动。基本上，它扩展了对摄像头发布者的支持。发布者会发送左边图像和右边图像，并附带摄像头信息。你能够运行它并使用以下命令查看图像：

```
roslaunch chapter6_tutorials camera_stereo.launch view:=true
```

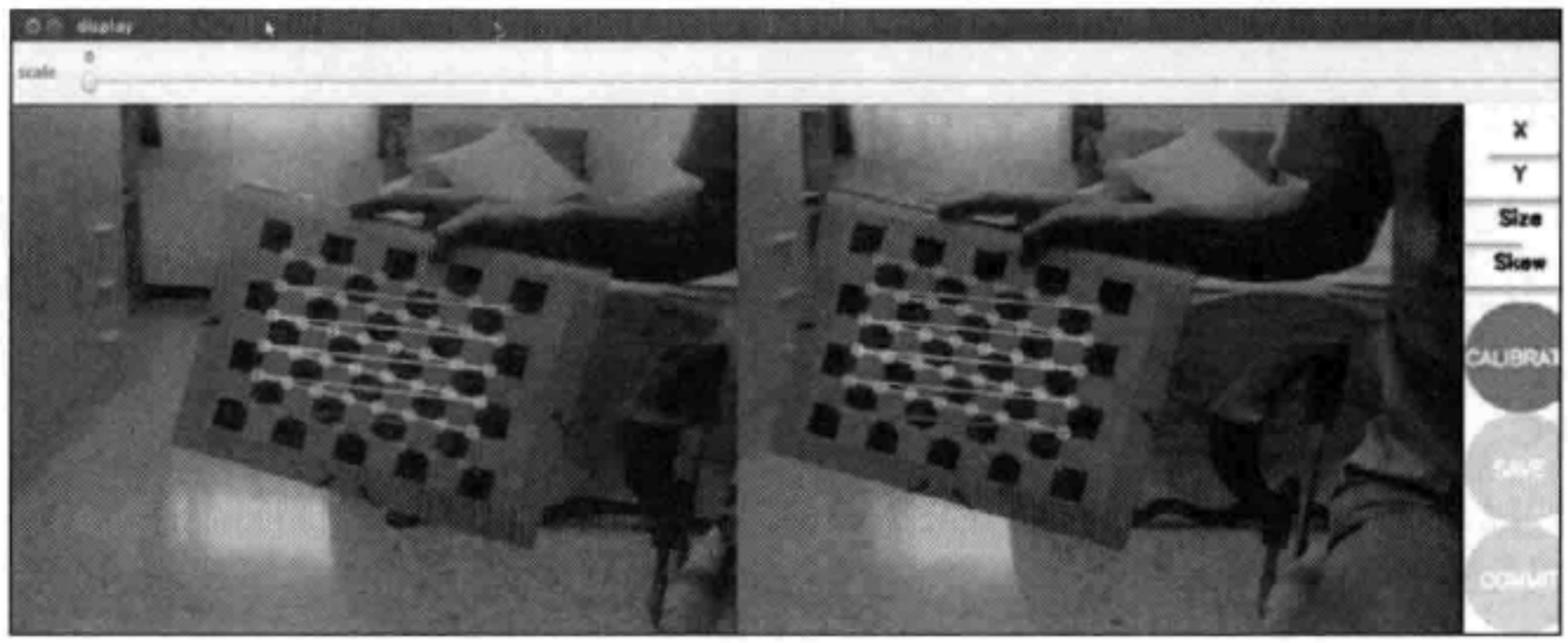
如果你在电脑上只连接了两个摄像头，这个命令就应该正确工作，并且摄像头的 ID 是 1 和 2(本例中之前已经有一个摄像头了)。如果你需要修改摄像头 ID，就需要编辑 params/camera_stereo/ logitech_c120.yaml 文件（这也是前面 .launch 文件默认的）并在以下字段中设定适当的 ID：

```
camera_index_left: 1
camera_index_right: 2
```

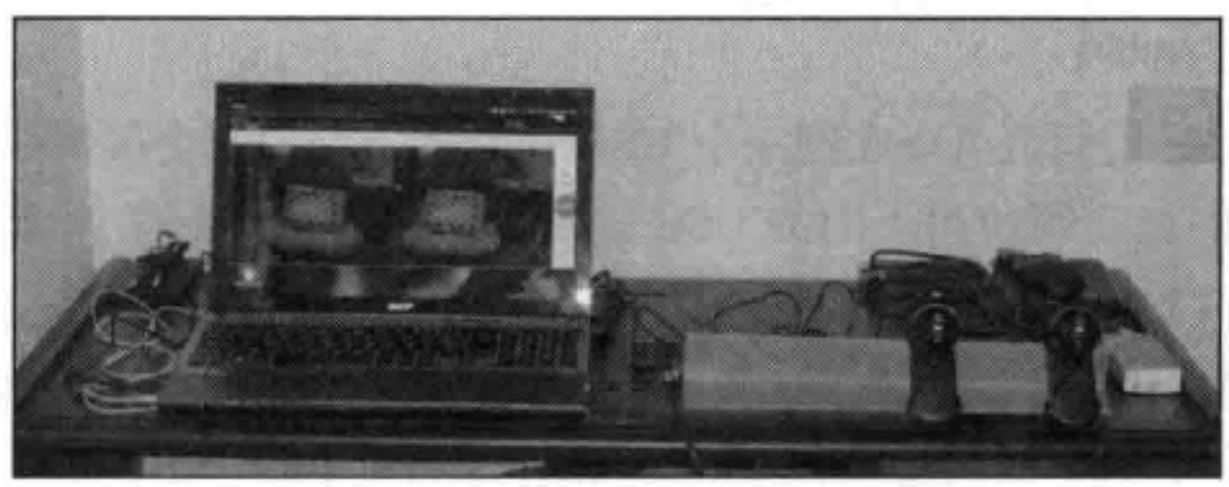
这还会显示左右摄像头的视差图像。视差图像是非常重要的，我们会在后面专门讨论。一旦摄像头被标定好，就能够使用 ROS 图像管道。为了能够标定，我们需要运行以下命令：

```
roslaunch chapter6_tutorials camera_stereo.launch calibrate:=true
```

你能看到和前面单目摄像头类似的 GUI。



在上图截屏的时刻，我们已经有了足够的视图来启动标定。注意标定图案必须被两个摄像头同时检测到，才能完成后续的标定优化步骤。这取决于摄像头的安装，所以可能会有些棘手。你应该将图案置于摄像头适当的距离范围内。在下图中你能看到本书中用于标定的安装方法：



标定是由和单目摄像头相同的 cameracalibrator.py 节点完成。我们直接传递左右摄像头和图像，这样工具就会知道我们将进行双目标定。下面是 .launch 文件中的 node 字段：

```
<node ns="$(arg camera)" name="cameracalibrator"
      pkg="camera_calibration" type="cameracalibrator.py"
      args="--size 8x6 --square 0.030" output="screen">
  <remap from="left" to="left/image_raw"/>
  <remap from="right" to="right/image_raw"/>
  <remap from="left_camera" to="left"/>
  <remap from="right_camera" to="right"/>
</node>
```

标定的结果和单目摄像头是相同的，但是在这种情况下我们会分别保存两个摄像头的标定文件。根据 params/camera_stereo/logitech_c120.yaml 中的参数文件，我们有：

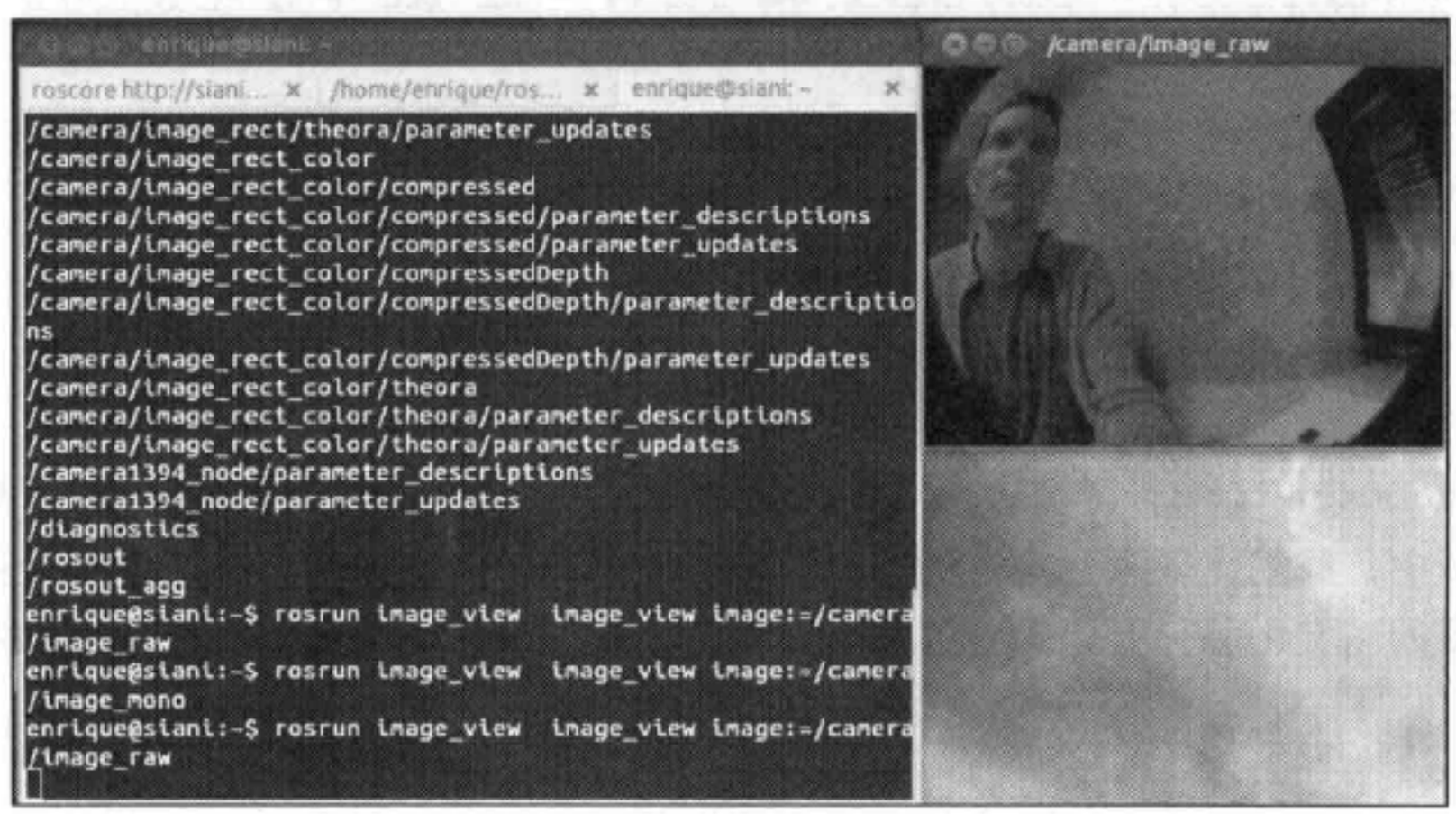
```
camera_info_url_left: package://chapter6_tutorials/calibration/camera_
stereo/${NAME}.yaml
camera_info_url_right: package://chapter6_tutorials/calibration/
camera_stereo/${NAME}.yaml
```

这里，${NAME} 是摄像头的名称，分别代表 logitech_c120_left 和 logitech_c120_right，也就是左右摄像头。在提交标定之后，会将每个摄像头的标定更新到这些文件。这包含了标定矩阵、失真模型参数和透视投影矩阵，其中还包含基线。基线表示的是在图像平面 x 轴方向上两个摄像头的距离。在参数文件中，你还能看到一些摄像头的属性值用来设定人工灯光的室内环境。这个摄像头是自动校正的，所以有时候图像质量比较差，但大多数情况还是可以接受的。

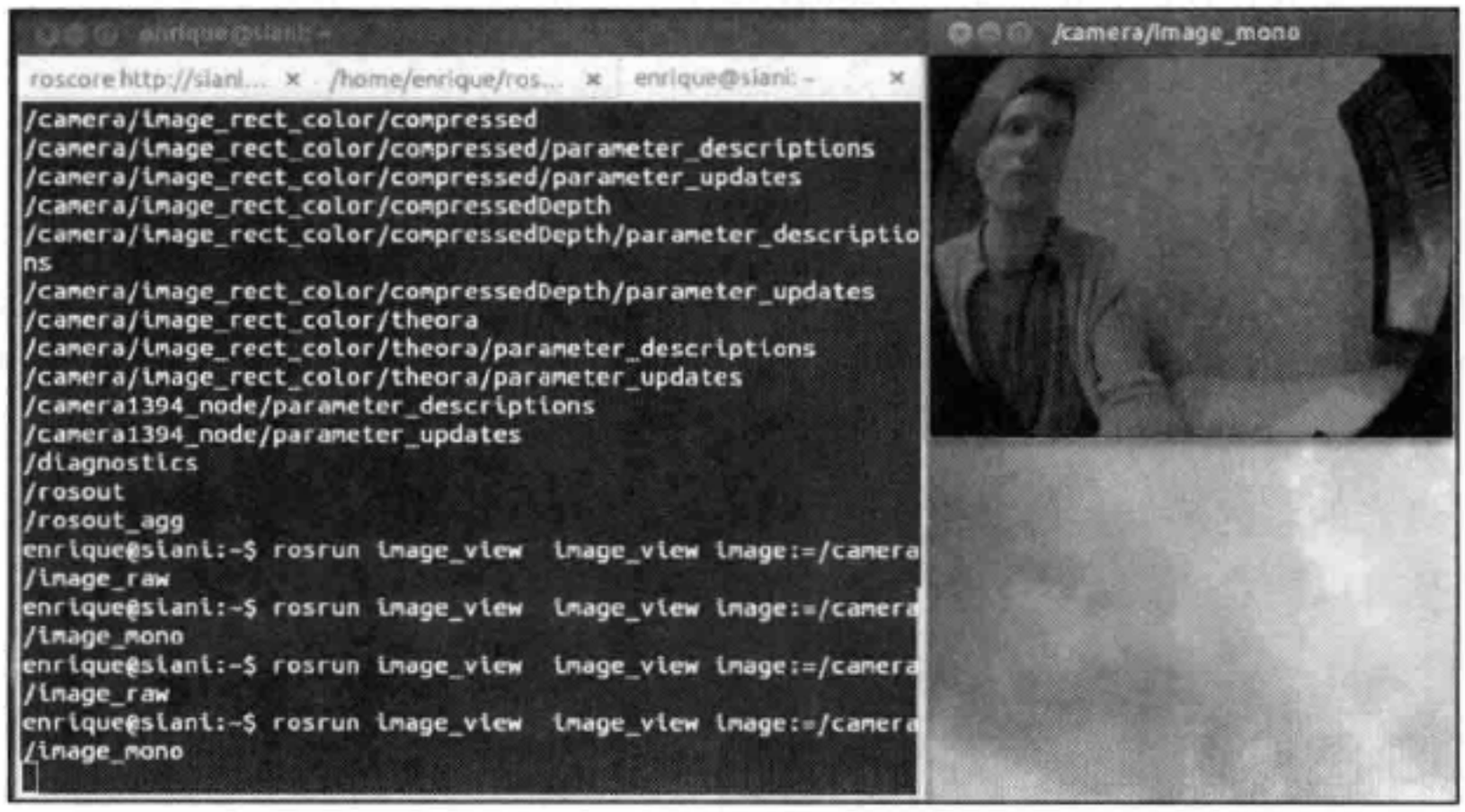
6.4 ROS 图像管道

ROS 图像管道通过 image_proc 功能包运行。它提供了各种用于从摄像机采集的源图像中获取单色和彩色图像的转换功能。在使用 FireWire 摄像头的情况下，摄像头 (在其图像传感器中) 很可能使用了拜耳（Bayer）模式进行图像编码。需要进行去拜耳化来获得彩色图像。一旦你标定完摄像头，图像管道就会提取 CameraInfo 消息（其中包含了去拜耳化模式信息）并修正你的图像。这里，修正意味着修复图像，这样就能用失真模型的参数来修正径向和切向的失真。

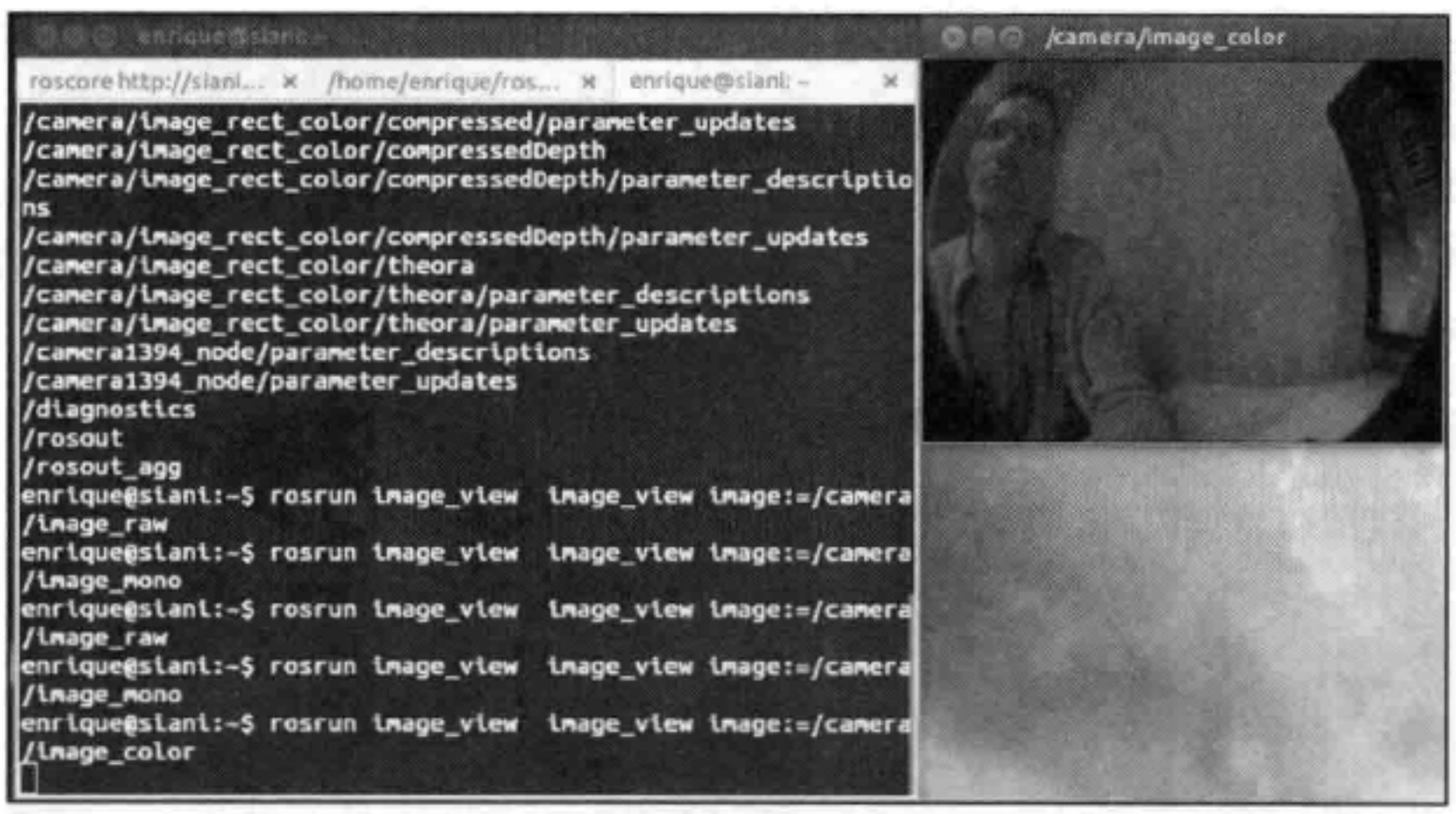
ROS 图像管道的结果是，我们能够在命名空间内看到更多的关于摄像头的主题。在下图中，你能看到主题 image_raw、image_mono 和 image_color 分别显示了源图像、单色图像、彩色图像：



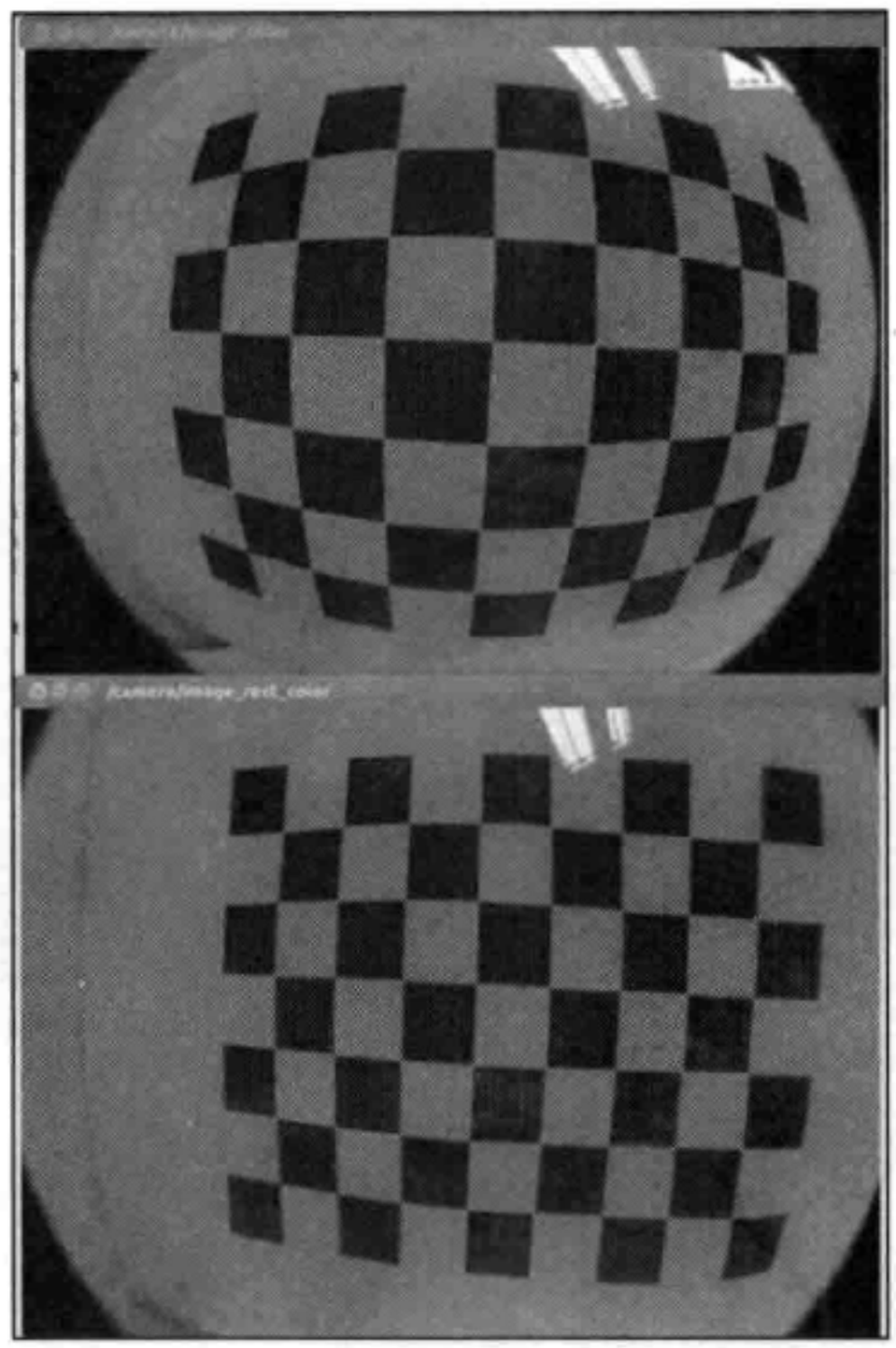
单色图像在下图中显示。在本例中，和源图像是一样的，但是在源图像中传感器是带有拜耳模式的：



最后，在 /image_color 主题中我们看到了彩色图像。需要注意的是转换的过程并不一定是源图像、单色图像到彩色图像，因为很多摄像头能够直接输出彩色图像。如果是这样的话，源图像和彩色图像就是一样的，而单色图像是通过彩色图像去饱和得到的：



在主题 image_rect 和 image_rect_color 中，提供的修正后的图像是单色和彩色的。在下图中，我们比较未标定过的失真源图像和修正后的图像。你能分别哪个是修正的，因为在下图的图案中，只有修正后的才有直线。尤其是在远离图像中心（传感器的原点）的区域里：



你能通过 rostopic list 或 rxgraph 命令查看所有活动的主题，这也包括图像传输。你能直接查看单目摄像头的 image_raw 主题，使用以下命令：

```
roslaunch chapter6_tutorials camera.launch view:=true
```

它也能用来查看其他的主题。但是对于一些摄像头，源图像已经是彩色的了。如果要查看修正后的图像，也可以用 image_view 查看 image_rect_color 主题，或者修改 .launch 文件。通过 image_proc 节点可以激活这些主题，只需要在 .launch 文件中加入以下代码：

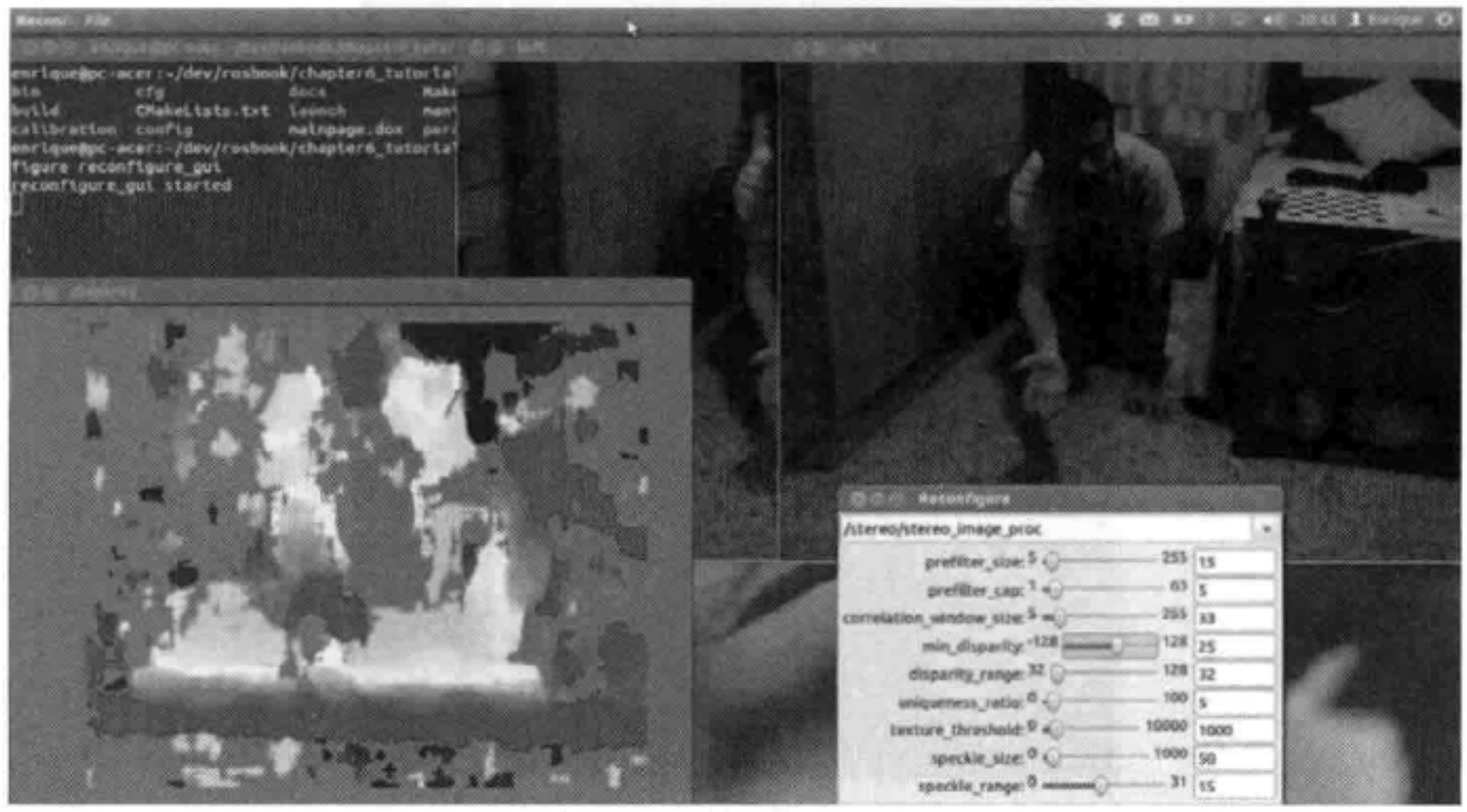
```
<node ns="$(arg camera)" pkg="image_proc" type="image_proc"
name="image_proc"/>
```

图像管道的双目摄像头

在双目摄像头的情况下，对左右摄像头来说都是相同的。然而，也有一些专门针对它们开发的可视化工具，这样我们就能够使用左右图像来计算和查看视差图像。其主要算法是使用双目视觉标定和左右图像纹理估计每个像素的深度，也就是视差图像。为了获得优秀的计算结果，必须调整计算这种图像的算法。在下图中我们看到左、右和视差图像，也就是使用 stereo_image_proc 的 reconfiguire_gui，在其中节点为双目图像建立了图像管道。在 .launch 文件中我们只需要：

```
<node ns="$(arg camera)" pkg="stereo_image_proc"
    type="stereo_image_proc"
    name="stereo_image_proc" output="screen">
  <rosparam file="$(arg params_disparity)"/>
</node>
```

如下图所示，它需要能够设定 reconfigure_gui 的视差参数，并与 rosparam dump /stereo/stereo_image_proc 一起保存：



我们有一套调整好的适用于在本书编写时环境的参数，保存在参数文件 params/camera_stereo/disparity.yaml 中：

```
{correlation_window_size: 33, disparity_range: 32, min_disparity: 25,
prefilter_cap: 5,
  prefilter_size: 15, speckle_range: 15, speckle_size: 50, texture_
threshold: 1000,
  uniqueness_ratio: 5.0}
```

然而，这些参数很大程度上依赖于标定质量和环境。你应该调整一套适合自己使用的环境参数。这是很费时间的，有时候也很棘手。但是你可以参考以下 ROS 官网的指导进行：http://www.ros.org/wiki/stereo_image_proc/Tutorials/ ChoosingGoodStereoParameters

基本上，你会从设定 disparity_range 的值开始，这个值允许一定范围的视差色块。你还需要设定 min_disparity，你能看到覆盖一整片区域的深度显示（从红色到蓝色 / 紫色）。然后，你要调整好 speckle_size 来消除小的噪声斑点，并通过修改 uniqueness_ratio 和 texture_threshold 来获得较大的视差色块。参数 correlation_window_size 会影响到初始视差色块的检测。

如果总是很难获得好的检测结果，那么你也许会不得不进行多次标定，或者根据你的环境和灯光条件换一套更好的摄像头，你也可以试着在其他的环境中进行实验或者多点几

个灯。你最好是找一个纹理比较多的环境，因为站在一个大白墙前面你是很难找到任何视差的。还有由于基线的原因，你不能获取与摄像头特别近的深度信息。类似地，与摄像头距离过远时，对深度信息的估计也非常不准确。总结一下，基线距离的选取要根据应用目标而定，它应该与我们想要测量的距离成比例。我们使用 12cm 作为基线距离，相对比较适合 1 ～ 2m 的距离。因为在后面我们会尝试进行视觉测距，这通常需要测量相对远一点的距离（>1m）。我们现在的程序里只能获得距摄像头大约 1m 的深度信息。如果选择更小的基线，那么就能够获取更近物体的深度信息。这不适于导航，因为无法找到较远一点路径的解决方案，但却对精确观察和抓取物体很有利。

对于标定问题，你能够通过 cameracheck.py 节点检查标定结果，它已经集成在了单目和双目摄像头的 .launch 文件中：

```
roslaunch chapter6_tutorials camera.launch view:=true check:=true
roslaunch chapter6_tutorials camera_stereo.launch view:=true check:=true
```

对于单目摄像头，我们的标定会产生均方根（RMS）误差（通过 calibration/camera/cameracheck-stdout.log 查看更多信息）：

```
Linearity RMS Error: 1.319 Pixels      Reprojection RMS Error: 1.278
Pixels
Linearity RMS Error: 1.542 Pixels      Reprojection RMS Error: 1.368
Pixels
Linearity RMS Error: 1.437 Pixels      Reprojection RMS Error: 1.112
Pixels
Linearity RMS Error: 1.455 Pixels      Reprojection RMS Error: 1.035
Pixels
Linearity RMS Error: 2.210 Pixels      Reprojection RMS Error: 1.584
Pixels
Linearity RMS Error: 2.604 Pixels      Reprojection RMS Error: 2.286
Pixels
Linearity RMS Error: 0.611 Pixels      Reprojection RMS Error: 0.349
Pixels
```

对于双目摄像头，我们产生的极线误差（epipolar error）和标定图案中单元格大小的估计（查看 calibration/camera_stereo/cameracheck- stdout.log 文件）：

```
epipolar error: 0.738753 pixels   dimension: 0.033301 m
epipolar error: 1.145886 pixels   dimension: 0.033356 m
epipolar error: 1.810118 pixels   dimension: 0.033636 m
epipolar error: 2.071419 pixels   dimension: 0.033772 m
epipolar error: 2.193602 pixels   dimension: 0.033635 m
epipolar error: 2.822543 pixels   dimension: 0.033535 m
```

为了获取这些结果，你只需要向摄像头提交标定图案。这也是我们会传递 view：

=true 到 .launch 文件的原因。一般来说均方根误差如果大于两个像素，那么就是比较大的误差了。即使如此我们还是能够在这个条件下进行一些实验的，因为你也要记住这些都是廉价的摄像头。当然如果你的误差小于 1 个像素，那就是非常理想的了。对于双目摄像头，极线误差应该低于 1 个像素。但在我们的示例中，它还是非常大的（有时候甚至高于 3 个像素），即使如此它也是可用的。事实上，视差图像就是通过 stereo_view 节点显示和表征每个像素的深度信息。我们还有在 rviz 中能够查看纹理的 3D 点云。我们会使用这个工具进行视觉测距。

6.5 对于计算机视觉任务有用的 ROS 功能包

在 ROS 中使用机器视觉最为有利的条件是我们不需要重复发明。我们可以使用大量的第三方软件，也可以将视觉设备与真实的机器人相连接，或者仅仅是做仿真。这里，我们将会列举一些有趣的机器视觉工具。它们能够应用于大量通用的视觉工作。但是我们只会在后面详细介绍其中的一个（包括安装和使用的所有步骤）。这就是视觉测距（Visual odometry）库，当然其他的库也非常容易安装和使用。只需要跟着超链接中的示例或手册进行学习即可。具体的列表如下：

- Visual Servoing（也被称为 Vision-based Robot Control）：这是一种使用从视觉传感器获取的反馈信息来控制机器人的动作的技术，特别是手臂抓取。在 ROS 中我们可以使用 Visual Servoing Platform (ViSP) 软件控制夹持器。(http://www.irisa.fr/lagadic/visp/visp.html 和 http://www.ros. org/wiki/vision_visp)。ViSP 是一种完全的跨平台库，能够在视觉追踪和视觉伺服领域进行原型设计和应用开发。ROS 软件包提供了一种跟踪器，它能够通过 visp_tracker（移动边缘跟踪器）节点或 visp_auto_tracker（基于模型的跟踪器）节点运行。它还能帮助摄像头标定和手眼标定。手眼标定对于抓取任务中的视觉伺服是极为重要的。
- Augmented Reality (AR)：一种通过在真实环境中覆盖虚拟图像实现的增强现实应用。实现这一功能的最著名的库是 ARToolkit (http://www.hitl.washington.edu/artoolkit/)。这个应用的主要问题在于跟踪用户的视点，所以虚拟图像要画到用户在真实环境中所关注的视点上。ARToolkit 视频追踪库会计算实际摄像头的位置和方向并进行实时的物理标记。在 ROS 中我们有一个名为 ar_pose (http://www.ros.org/wiki/ar_pose) 的软件包。它允许我们追踪一个或多个标记，并在相应位置呈现我们的虚拟图像（例如，3D 模型）。
- Perception and object recognition：使用 OpenCV 库就能完成最基本的感知与物体识别。然而还有一个更有价值去探索的工具就是 RoboEarth (http://www.roboearth.org)。它允许我们对物体进行探测和 3D 建模，并将其存储在一个全球性的数据库中。这个数据库允许全世界所有机器人（或人）访问。存储模型可以是 2D 或 3D 的，它可

以用于识别相似的物体或它们的视点，也就是说，识别摄像头 / 机器人看到的物体。RoboEarth（机器人地球）计划已经被集成在 ROS 中，而且已经有很多个教程提供用来系统运行和试用 (http://www.ros.org/wiki /roboearth).

- Visual odometry：视觉测距算法会假定在一个静止的环境中使用环境图像来跟踪特征并估计机器人的运动。它使用单目或双目视觉系统能够解决机器人 6 自由度的定位问题。当然在单目视觉时还需要一些附加信息。视觉测距可以使用两个算法库：`libviso2` (http://www.cvlibs.net/software/libviso2.html) 和 `libfovis` (http://www.ros.org/wiki/fovis_ros)，这两者都是 ROS 软件包的一部分。你可以直接在 ROS 中找到，它们分别是 `viso2` 和 `fovis`。

在下面的章节中，我们会使用自制的双目摄像头调用 `viso2` 中的 `viso2_ros` 节点进行视觉测距。库 `libviso2` 允许我们完成单目或双目视觉测距，但是对于单目测距来说，还需要进行地板平面的位置估计。你可以用 IMU（见第 4 章）来试一下单一摄像头的单目测距，但是如果用双目摄像头将总能得到更好的结果，更正确的标定，正如本章中前几节所介绍的。最后，`libfovis` 库并不支持单目视觉，但是它支持 RGB-D 相机，例如 Kinect 传感器（见第 4 章）。对于使用双目视觉来说，你最好试一下这两个库，并看一下哪个库在你的环境中性能更好。这里，我们使用 `viso2` 库为大家提供一个在 ROS 中从安装到运行的详细指导。

6.6 使用 viso2 执行视觉测距

在 ROS Fuerte 和 Groovy 中都包含了当前版本的 `viso2`。然而，它在这两个版本中都需要使用 `catkin` 编译系统。在本书中，我们已经介绍了 `rosmake` 经典编译系统，因此在此我们会为基于 `catkin` 安装的 `viso2` 提供详细的指导。依次运行以下命令：

```
cd ~
mkdir ros
cd ros
mkdir -p caktin_ws/src
cd caktin_ws/src/
catkin_init_workspace
```

现在我们已经创建了 `catkin` 工作空间，继续安装 `viso2`。我们将使用 `wstool` 来完成这个工作，`wstool` 集成了系统的下载命令。这意味着我们只需要运行 `git clone git://github.com/srv/viso2.git` 就能完成安装，这会复制 `viso2` 的软件包。我们先安装 `wstool`，然后下载 `viso2`：

```
sudo apt-get install python-wstool
wstool init
wstool set viso2 --git git://github.com/srv/viso2.git
wstool update
```

在使用 catkin 的时候，需要先选择最常用的环境变量。我们希望能够更方便地安装 viso2，那么运行：

```
source /opt/ros/groovy/setup.bash
cd ..
catkin_make
```

现在，安装完 viso2，设定好开发环境变量，就能够运行 viso2_ros 节点，例如将要使用的 stereo_odometer。在此之前，还需要发布摄像头和机器人之间的坐标变换和它的基本连接（base link）。双目摄像头驱动已经安装好，但我们还会在后面的章节介绍它是如何工作的。

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

6.6.1 摄像头位姿标定

为了能够将不同的坐标系变换到机器人系统坐标系，我们必须发布这些变换的 tf 消息。最合适和通用的方法是使用 camera_pose 功能包集。我们使用的最新版本来自以下软件源：https://github.com/jbohren-forks/camera_pose。因为 ROS 中提供的库有些问题（例如，你在 ROS Fuerte 中运行它的时候总是崩溃，因为里面的 .launch 文件有点过时了）。这个功能包集提供了一系列的 .launch 文件，能够对摄像头进行相互位姿的标定。它提供了为 2、3、4 甚至更多个摄像头的 .launch 文件。在示例中，我们只使用两个摄像头（双目视觉），所以要这样做。首先，将 calibrate_pose 参数添加到 camera_stereo.launch 文件中，这样就会调用 camera_pose 功能包集中的 caliration_tf_publisher.launch 文件：

```
<include file="$(find camera_pose_calibration)/blocks/calibration_tf_
publisher.launch">
    <arg name="cache_file" value="/tmp/camera_pose_calibration_cache.
bag"/>
</include>
```

现在，运行以下命令：

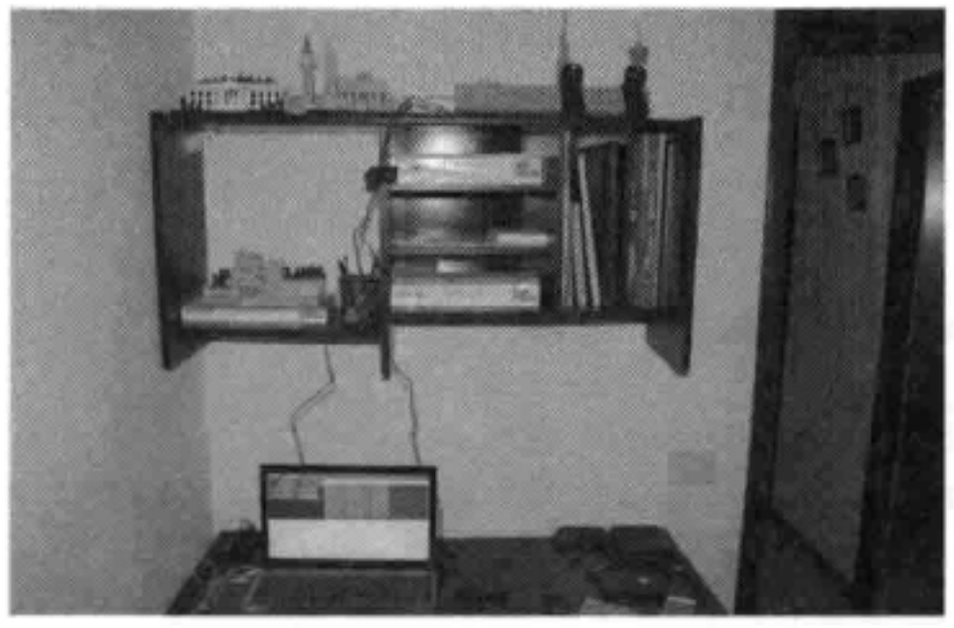
```
roslaunch chapter6_tutorials camera_stereo.launch calibrate_pose:=true
```

然后一旦标定正确完成，calibration_tf_publisher 文件会发布坐标系变换 (tf)。这个标定和我们之前介绍的非常相似。但是它使用了 camera_pose 中的特定工具，需要使用以下命令运行：

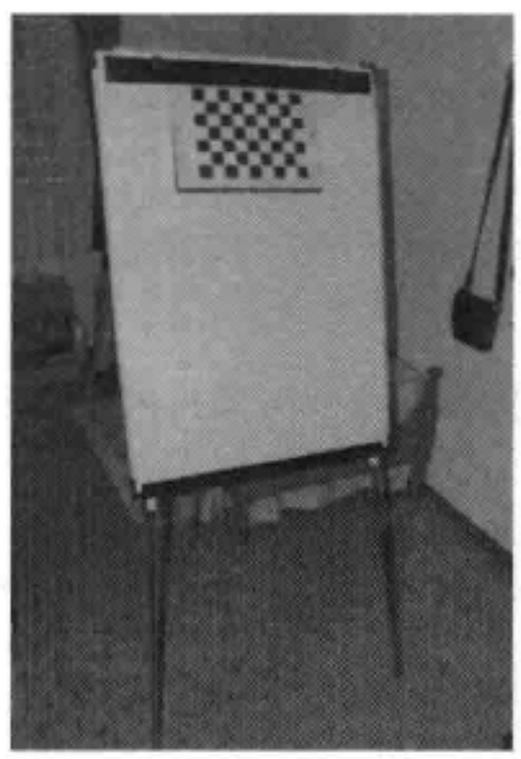
```
roslaunch camera_pose_calibration calibrate_2_camera.launch camera1_ns:=/
stereo/left camera2_ns:=/stereo/right checker_rows:=6 checker_cols:=8
checker_size:=0.03
```

通过这个命令，我们可以使用与之前相同的标定图案。然而，它需要图像完全静态。只有当图像相对于所以摄像头都保持静止一段时间之后，一些指示条才会从左到右完全变绿。

由于我们的摄像头本身就有很多噪声，因此我们还需要标定图案的支持，并且将摄像头和标定图案固定在三脚架或者平板上。如下图所示：



下面的图片显示了一个简易三脚架固定的标定图案。这是非常重要的，因为摄像头和标定图案都必须固定在某个位置上：



然后我们就能进行标定，如下图所示：

同时，这也会完成从左摄像头到右摄像头的 tf 坐标变换。虽然这是最恰当的方法来完成摄像头位姿标定，但是我们还是会使用一个更简单更实用的方法。它同样使用 viso2，因为它能够将整个双目摄像头作为一个单元 / 传感器进行处理。而在其内部，使用 cameracalibrator.py 的双目标定结果来检查基线。

我们有一个 .launch 文件中使用 static_transform_publisher 完成摄像头连接到基本连接 (例如 robot base) 的变换及摄像头连接到光学镜头连接的变换，因为光学镜头连接也需要一个旋转变换。试想一下，摄像头坐标系的 z 轴是沿着光学镜头方向向前的，而其他的坐标系（例如世界坐标系、导航坐标系、测距坐标系）的 z 轴都是指向上的。这个 .launch 文件在 launch/frames/stereo_frames.launch 中：

```
<launch>
  <arg name="camera" default="stereo" />

  <arg name="baseline/2" value="0.06"/>
  <arg name="optical_translation" value="0 -$(arg baseline/2) 0"/>

  <arg name="pi/2" value="1.5707963267948966"/>
  <arg name="optical_rotation" value="-$(arg pi/2) 0 -$(arg pi/2)"/>

  <node pkg="tf" type="static_transform_publisher" name="$(arg
camera)_link"
        args="0 0 0.1 0 0 0 /base_link /$(arg camera) 100"/>
  <node pkg="tf" type="static_transform_publisher" name="$(arg
camera)_optical_link"
        args="$(arg optical_translation) $(arg optical_rotation)
/$(arg camera) /$(arg camera)_optical 100"/>
</launch>
```

这个文件包含在我们的双目摄像头 launch 文件中，并发布这些静态的坐标变换。因此只需要运行以下命令来发布它们：

```
roslaunch chapter6_tutorials camera_stereo.launch tf:=true
```

然后你就能够使用 rviz 查看 tf 数据是否发布成功。我们将会在下一小节中介绍。

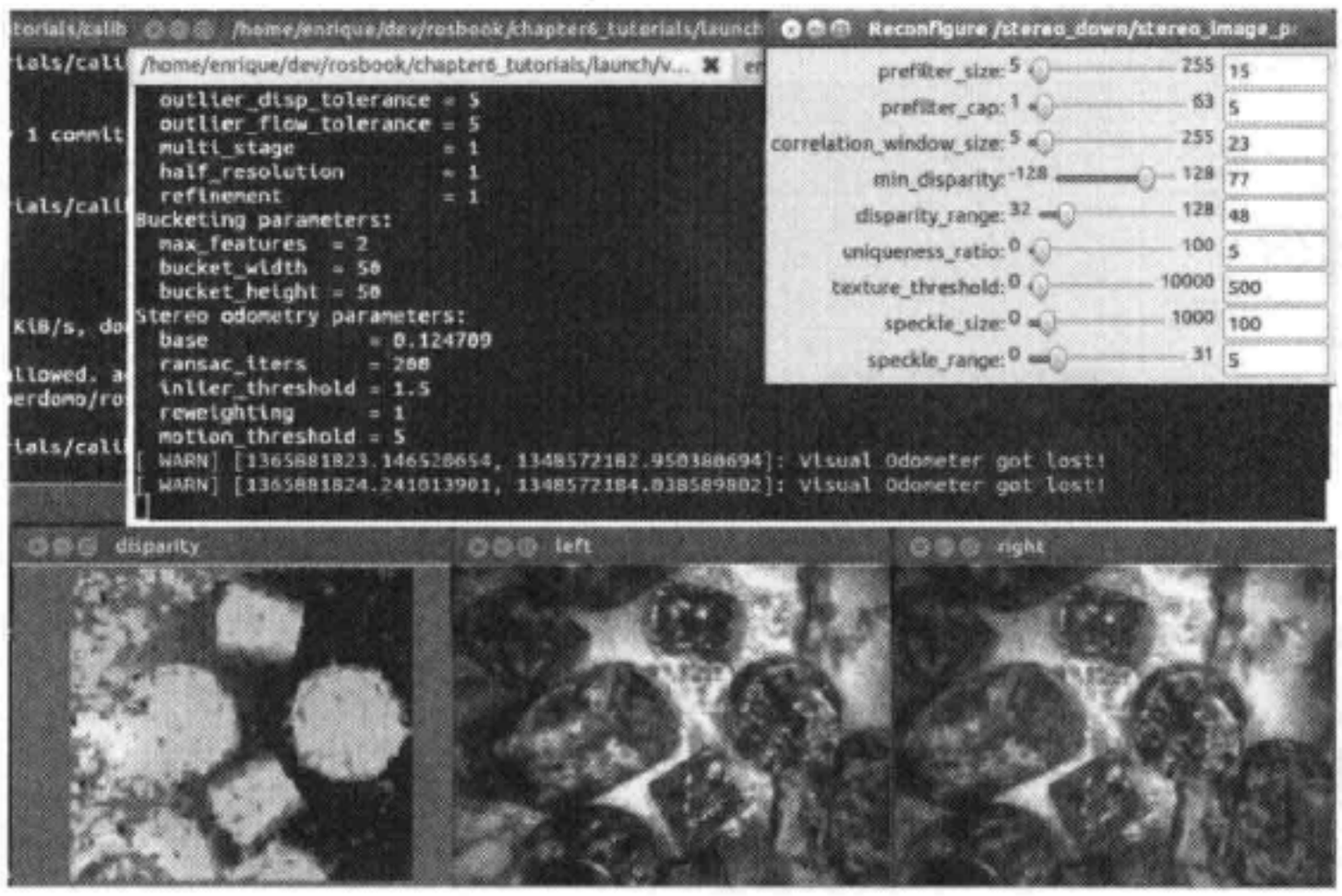
6.6.2 运行 viso2 在线演示

讲到这里，我们就要运行视觉测距算法了。双目视觉摄像头已经被标定过，它们的坐标系有适用于 viso2 的名称 (以 _optical 为结尾)，摄像头坐标系和光学坐标系的 tf 参数也已经发布。但在使用双目视觉摄像头之前，还要通过 ftp://opt.uib.es/bagfiles/viso2_ros 提供的消息记录包对 viso2 进行测试。直接运行 bag/viso2_demo/download_amphoras_pool_bag_files.sh 来获取所有的消息记录包（大约 4GB）。然后，我们在 launch/visual_odometry 目录下有一个 .launch 文件支持单目和双目测距。对于要运行的双目视觉 demo，上面的 .launch 文件还能够播放消息记录包文件并允许我们检查和可视化查看其内容。举例来说，要校正视差图像算法，运行以下命令：

```
roslaunch chapter6_tutorials viso2_demo.launch config_disparity:=true
view:=true
```

我们将看到左、右和视差图像，和一个 reconfigure_gui 界面用来配置视差算法。

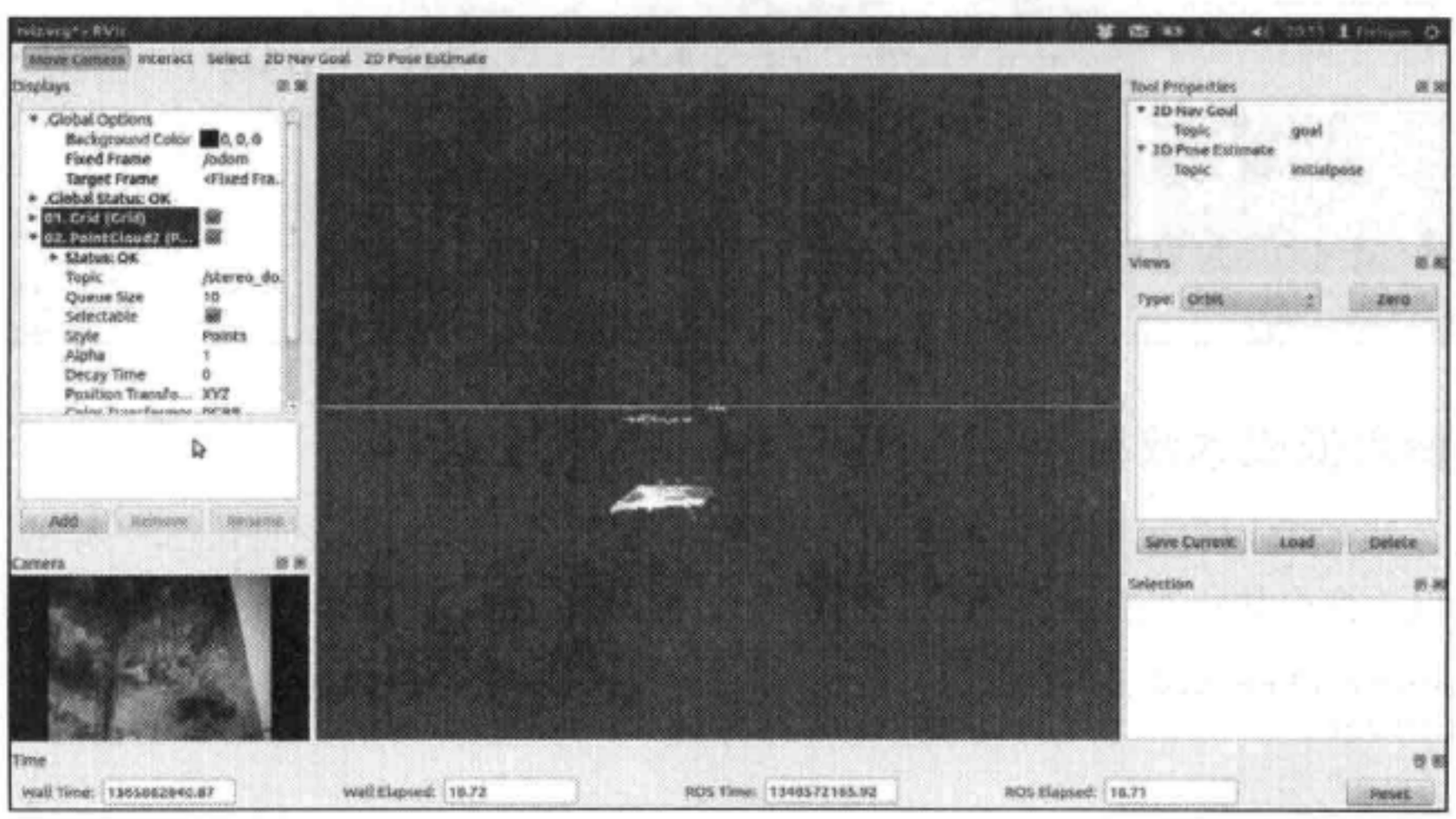
你之所以需要做这些调节是因为在消息回放包文件中只有源图像。我们已经在 params/viso2_demo/disparity.yaml 文件中保存了一些调好的参数。在下图中，你能够看到由此获得的结果，其中能清晰地看到双目图像中石头的深度：



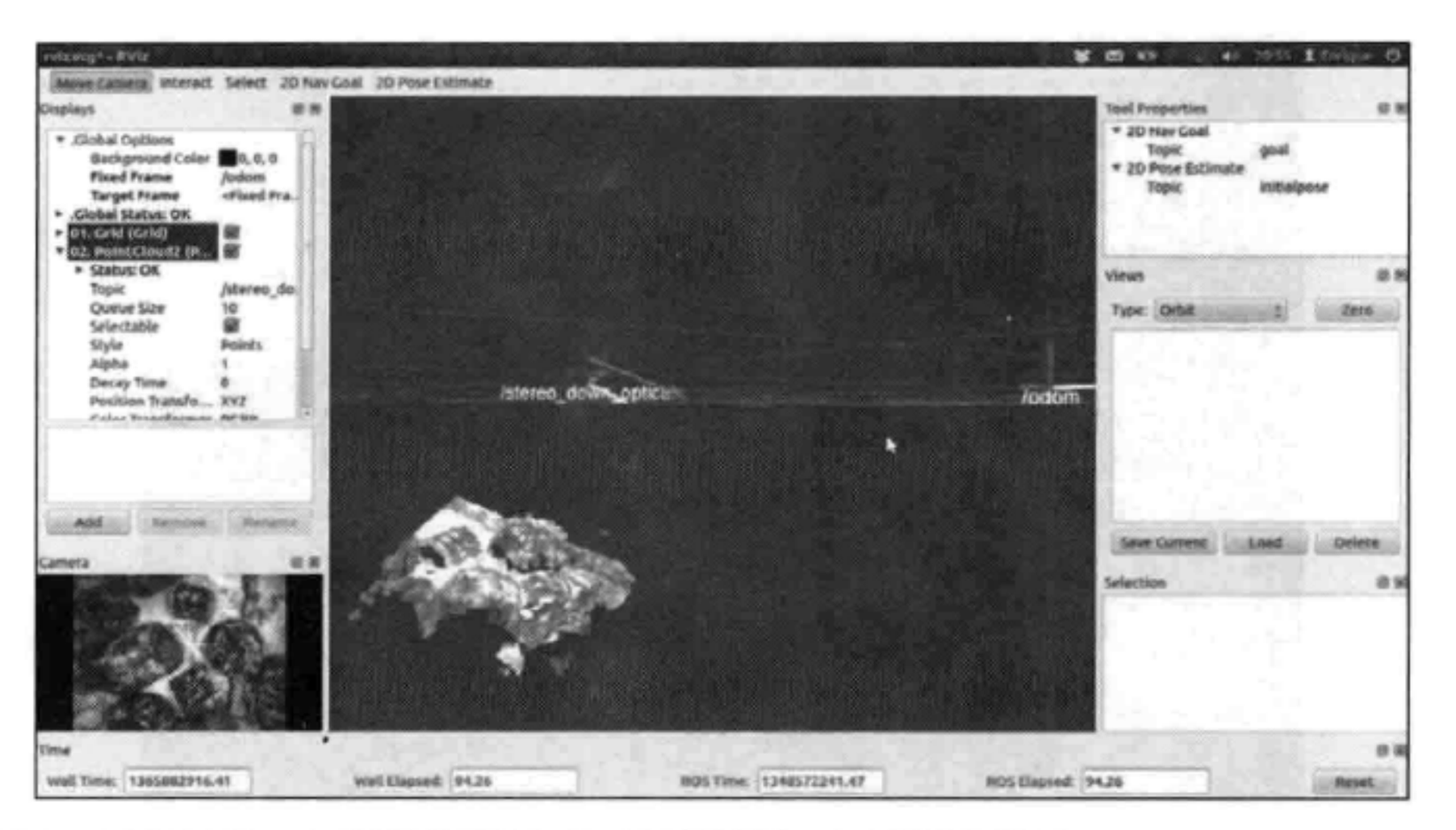
运行以下命令来启动双目测距并在 rviz 中查看结果：

```
roslaunch chapter6_tutorials viso2_demo.launch odometry:=true rviz:=true
```

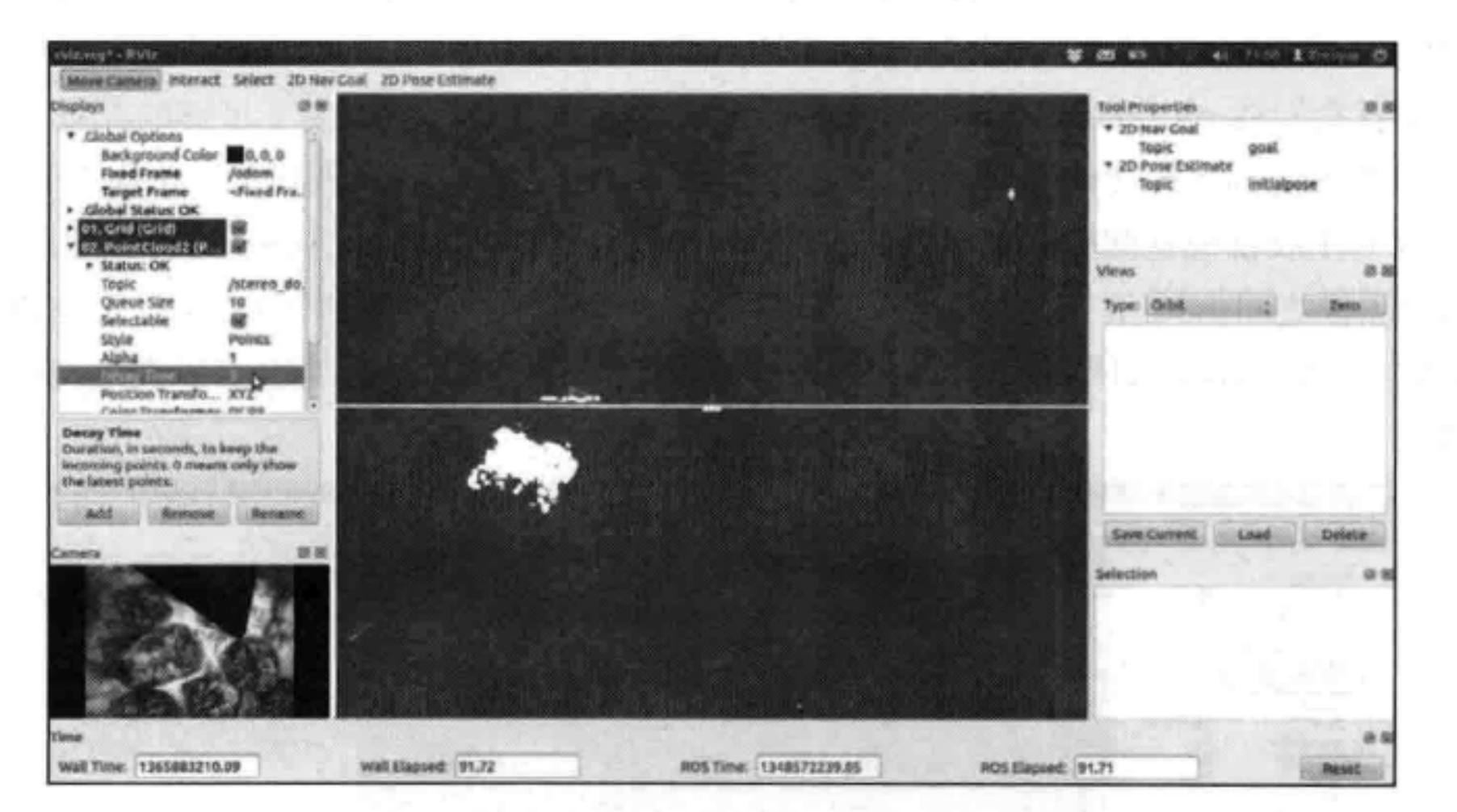
请注意我们在 params/viso2_demo/rviz.cfg 中为 rviz 提供了足够的配置。该配置文件会自动加载 launch 文件。下面三个图片显示了摄像头采集的纹理化 3D 点云的不同瞬间，以及显示了在双目视觉测距中用来摄像头位置估计的 /odom 和 /stereo_optical 坐标系。三张点云图像之间大约有三秒钟的延迟时间，这样我们能够看出随着时间变换点云是如何覆盖和演变的。也就是说，如果图像质量和测距算法都足够好，那么我们甚至可以在 rviz 中画一张地图：



下图是看一张稍晚一点时间且距离更近的截图，我们可以好好欣赏一下 3D 重建的效果，虽然其中有一些方向上的漂移（在 3D 重建过程中有部分倾斜，而实际应该基本保持水平）：



最后，还有另外一个视觉测距的 3D 重建视图，如下图像所示：

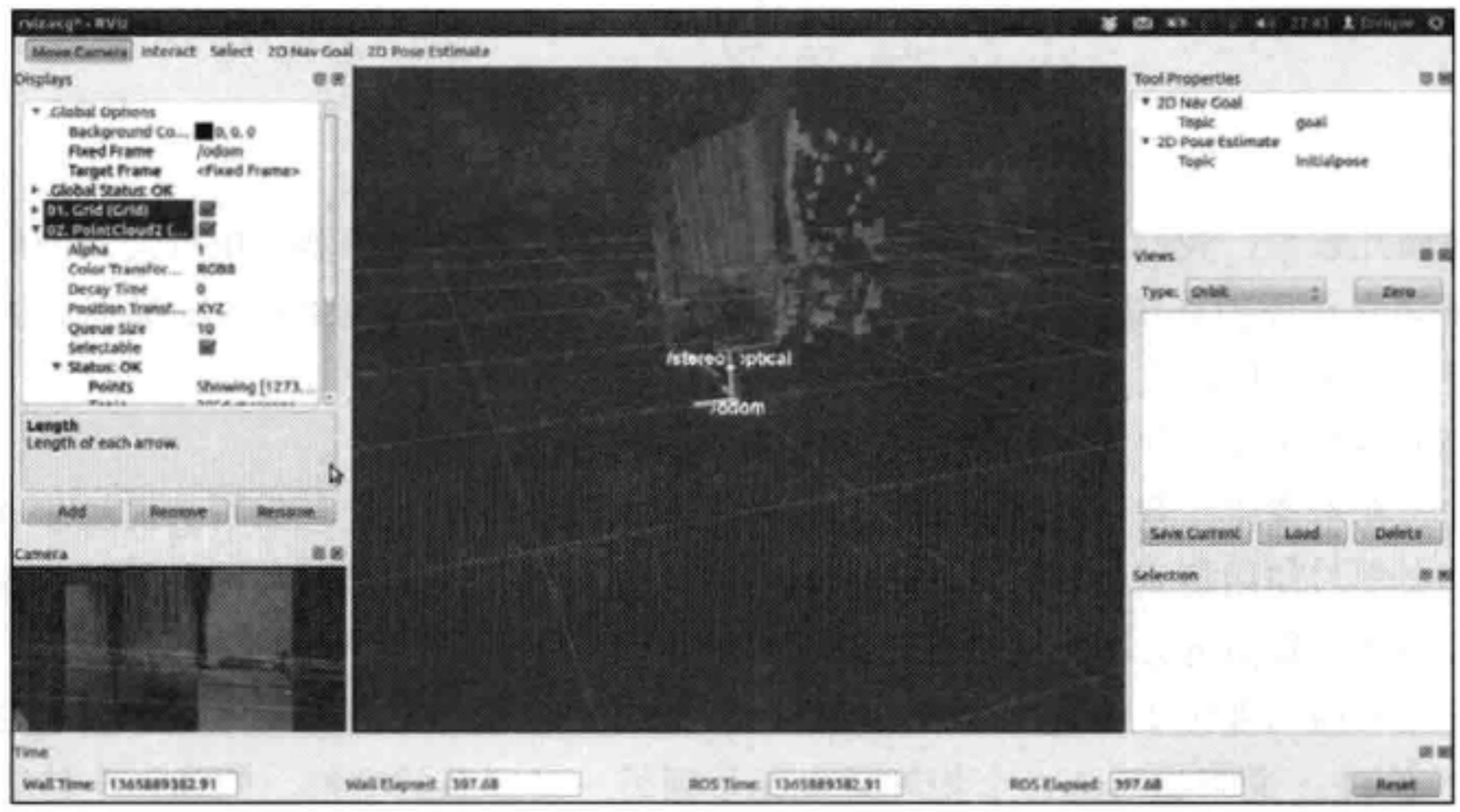


6.6.3 使用低成本双目摄像头运行 viso2

最后，我们改用自己做的低成本双目摄像头完成与前面 viso2_demo 一样的工作。只需要运行以下命令执行视觉测距并在 rviz 中查看结果（默认不发布坐标变换 (tf)）：

```
roslaunch chapter6_tutorials camera_stereo.launch odometry:=true
rviz:=true
```

下图显示了低成本双目摄像头视觉测距的结果。如果你移动摄像头，会看到 /odom 坐标系跟着移动。如果标定得不好或者摄像头噪声太大，测距信息可能会丢失，并会向你运行的命令行窗口中传递一条警告信息。在这样的情况下，你应该使用更好的摄像头或者重新标定摄像头，再试一下是不是会有更好的结果。你也可以试着调一下视差算法的参数。



6.7 本章小结

本章对 ROS 提供的机器视觉工具进行了一个概述。我们从介绍如何连接和运行多种摄像头开始讲起，尤其是 FireWire 和 USB 摄像头。然后介绍了改变摄像头参数的基本功能，以便于你能够通过调整参数获得更高质量的图像。除此之外，我们还提供了一个完整的 USB 摄像头驱动示例。

然后，我们开始介绍摄像头标定。在这之中你学习到标定一个摄像头是多么的简单。进行标定的重要性在于它能够纠正广角镜头的失真现象，尤其对于廉价的摄像头。还有，标定矩阵允许你进行多种机器视觉任务，例如视觉测距和感知。

我们展示了如何在 ROS 中使用双目视觉，如何安装和使用两个廉价摄像头，还解释了图像管道和 ROS 中机器视觉相关的几个 API，例如 `cv_bridge`、ImageTransport 和 ROS 功能包中集成的 OpenCV 库。

最后，我们列举了 ROS 支持的一些机器视觉工具及这些工具的应用方向。特别地，我们介绍了使用 `viso2` 库完成视觉测距的示例，展示了一些使用高质量摄像头所记录数据的示例和使用廉价摄像头的示例。因此，在学习完本章并运行示例代码之后，你就可以开始机器视觉的使用和研究了。用不了多长时间，你就会有突破性的进步。

第 7 章
导航功能包集入门

在前面几章中，我们介绍了如何创建自己的机器人、添加传感器和执行器和使用游戏杆或键盘驱动它在虚拟环境中移动。现在，在本章中，你会学习到 ROS 系统最强大的特性之一，它能够让你的机器人自主导航和运动。

这要感谢开源社区和共享代码，ROS 拥有大量用于导航的算法。

首先，在本章中，我们会学习使用自己的机器人配置导航功能包集所必需的步骤。在下一章中，会学习使用仿真机器人配置和启动导航功能包集，给定目标和配置参数以获得最优结果。特别地，我们将会在本章中介绍以下内容：

- 介绍导航功能包集和它们的强大功能——很明显这是 ROS 中最重要的软件包之一。
- 介绍 TF，说明如何从一个物理量坐标系到另一坐标系的变换。例如一个传感器采集的数据或一个执行机构收到期望位置的命令。
- 了解如何创建一个激光雷达的驱动或对其仿真。
- 学习测距结果是如何被计算和发布的，Gazebo 如何提供测距结果。
- 提出一个基本控制器，包括如何为你的机器人创建它的细节描述。
- 我们会看到如何在 ROS 中执行 SLAM。也就是说，我们会向你展示你如何使用自己的机器人从环境中构建一个地图，并在其中移动。
- 最后，你能够使用导航功能包集的定位算法在地图中定位你的机器人。

7.1 ROS 导航功能包集

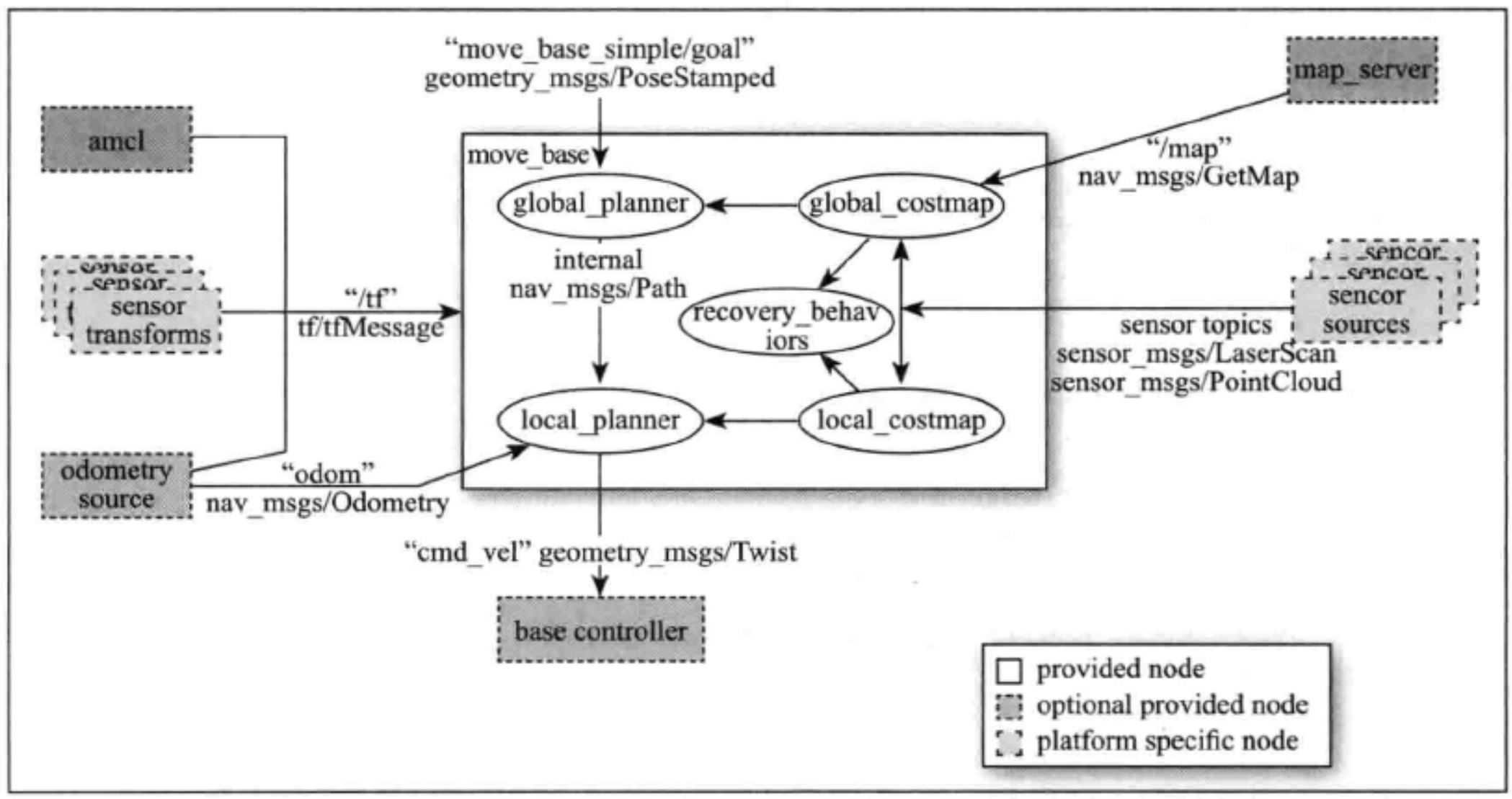
为了能够理解导航功能包集，你应该把它看做一套使用机器人传感器和测距功能的算法，你还能够使用标准的消息来控制机器人，毫无难度地移动机器人到你想要的位置（例如，不会产生碰撞或者卡在某个位置，或者丢失控制信号）。

你可以认为这个功能包集能够为任何机器人所用。几乎可是说是这样的，但是也需要调整一些配置文件并且调用功能包集写一些节点。

在使用导航功能包集之前，我们的机器人还是要满足一些条件的：

- 导航功能包集只能够处理双轮差分驱动和完全轮[⊖]驱动的机器人。机器人的形状必须是方形或者矩形。当然，双足机器人也可以使用它做些事情，如机器人定位等，只要你不去横向移动机器人就行。
- 需要机器人能发布关于所有关节和传感器位置关系的信息。
- 机器人必须能发送线速度和角速度消息。
- 机器人必须有一个平面激光雷达来完成地图构建和定位。或者，你能生成一些能够伪装成激光雷达或者声呐的数据，并给定采集这些数据的传感器在你机器人上的安装位置。

下图展示了导航功能包集的组织方式。在图中，你能够看到有白色、灰色和虚线的三种框。白框表示了其中的这些功能包集已经在 ROS 中集成了，并且它们提供的多种节点能够为你的机器人实现自主导航：



在下面几节中，我们还会看到如何创建上图中灰色框表示的内容。这些部分取决于你所使用的机器人平台。换句话说就是，这些部分的代码并不是通用的，需要根据你所使用的机器人，编写一定的代码实现 ROS 和导航功能包集的数据接口。

7.2 创建转换

导航功能包集需要知道传感器、轮子和关节的位置。

我们使用 TF（Transform Frame）软件库来完成这部分工作。它会管理坐标变换树。你可以使用数学工具来完成这部分工作，但如果你需要计算很多的坐标系，那么这就会显得有些

⊖ 非麦克纳姆轮，因为这种轮子有着很强的非线性。——译者注

复杂和混乱了。

这要感谢 TF 软件库，使得我们可以向机器人添加更多的传感器和组件，TF 会为我们处理这些设备之间的关系。

如果将激光雷达向后移动 10cm 或者向前移动 20cm（相对于在 base_link 坐标系中的原始坐标位置），我们都需要添加一个带有这些偏移的新坐标系到坐标变换树。

一旦插入和创建完成，我们就能轻松地知道激光雷达的相对于 base_link 坐标系或者是相对于轮子的位置。我们唯一需要做的就是调用 TF 库并进行坐标变换。

7.2.1 创建广播机构

让我们用一些简单的代码测试一下。在 chapter7_tutorials/src 文件夹下以 tf_broadcaster.cpp 为名称创建一个新的文件，并将以下代码放入文件中：

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

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

  ros::Rate r(100);

  tf::TransformBroadcaster broadcaster;

  while(n.ok()){
    broadcaster.sendTransform(
      tf::StampedTransform(
        tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.1,
0.0, 0.2)),
        ros::Time::now(),"base_link", "base_laser"));
    r.sleep();
  }
}
```

记住将下面这行加入到你的 CMakelist.txt 文件中并创建新的可执行文件：

```
rosbuild_add_executable(tf_broadcaster src/tf_broadcaster.cpp)
```

我们还创建了另外一个用于变换的节点，它会为我们提供传感器相对于 base_link 坐标系（我们机器人）原点的位置。

7.2.2 创建侦听器

在 chapter7_tutorials/src 文件夹下以 tf_listener.cpp 为名称创建一个新的文件，并将以下代码放入文件中：

```
#include <ros/ros.h>
#include <geometry_msgs/PointStamped.h>
#include <tf/transform_listener.h>
```

```
void transformPoint(const tf::TransformListener& listener){
  //we'll create a point in the base_laser frame that we'd like to
transform to the base_link frame
  geometry_msgs::PointStamped laser_point;
  laser_point.header.frame_id = "base_laser";

  //we'll just use the most recent transform available for our simple
example
  laser_point.header.stamp = ros::Time();

  //just an arbitrary point in space
  laser_point.point.x = 1.0;
  laser_point.point.y = 2.0;
  laser_point.point.z = 0.0;

    geometry_msgs::PointStamped base_point;
    listener.transformPoint("base_link", laser_point, base_point);

    ROS_INFO("base_laser: (%.2f, %.2f. %.2f) -----> base_link: (%.2f,
%.2f, %.2f) at time %.2f",
        laser_point.point.x, laser_point.point.y, laser_point.point.z,
        base_point.point.x, base_point.point.y, base_point.point.z,
base_point.header.stamp.toSec());

    ROS_ERROR("Received an exception trying to transform a point from
\"base_laser\" to \"base_link\": %s", ex.what());

}

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

  tf::TransformListener listener(ros::Duration(10));

  //we'll transform a point once every second
  ros::Timer timer = n.createTimer(ros::Duration(1.0),
boost::bind(&transformPoint, boost::ref(listener)));

  ros::spin();

}
```

记住还要增加一行代码到你的 CMakelist.txt 文件中并编译生成新的可执行文件。编译这个功能包并通过以下命令运行这两个节点：

```
$ rosmake chapter7_tutorials
$ rosrun chapter7_tutorials tf_broadcaster
$ rosrun chapter7_tutorials tf_listener
```

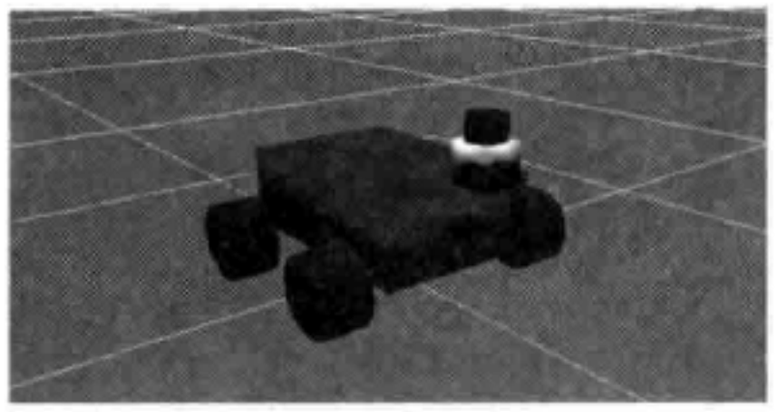
然后你会看到如下消息：

```
[ INFO] [1368521854.336910465]: base_laser: (1.00, 2.00. 0.00) ----->
base_link: (1.10, 2.00, 0.20) at time 1368521854.33

[ INFO] [1368521855.336347545]: base_laser: (1.00, 2.00. 0.00) ----->
base_link: (1.10, 2.00, 0.20) at time 1368521855.33
```

这表示了你在节点上发布的点若以 base_laser 坐标系为参照位于（1.00，2.00，0.00）的位置上，若以 base_link 坐标系为参照则位于（1.10，2.00，0.20）的位置上。

正如你所见，tf 库替你完成了所有的数学运算，得到了点的坐标或关节之间的相对位置。

一个变换树会以平移和旋转两种方式定义不同坐标系之间的偏移量。



让我们看一个示例来帮助你理解。我们使用第 5 章中的模型，可以再增加一个激光雷达，比如放在机器人的后面（base_link），如右图所示。

这样，系统必须知道这个新的激光雷达的具体位置，包括和轮子、和墙面之间的相对位置，只有这样才能够避免碰撞。通过坐标变换树，这些位置关系不但非常容易计算，而且也保持了可维护性和可扩展性。我们想要添加或者删除一个传感器或者机器人组件都非常的方便，因为 tf 坐标变换库会为我们完成所有的计算和相互关系的维护。所有的传感器和关节必须在坐标变换树中配置正确，这样才能保证导航功能包集清楚地知道机器人每个组件的具体位置，并准确无误地驱动机器人。

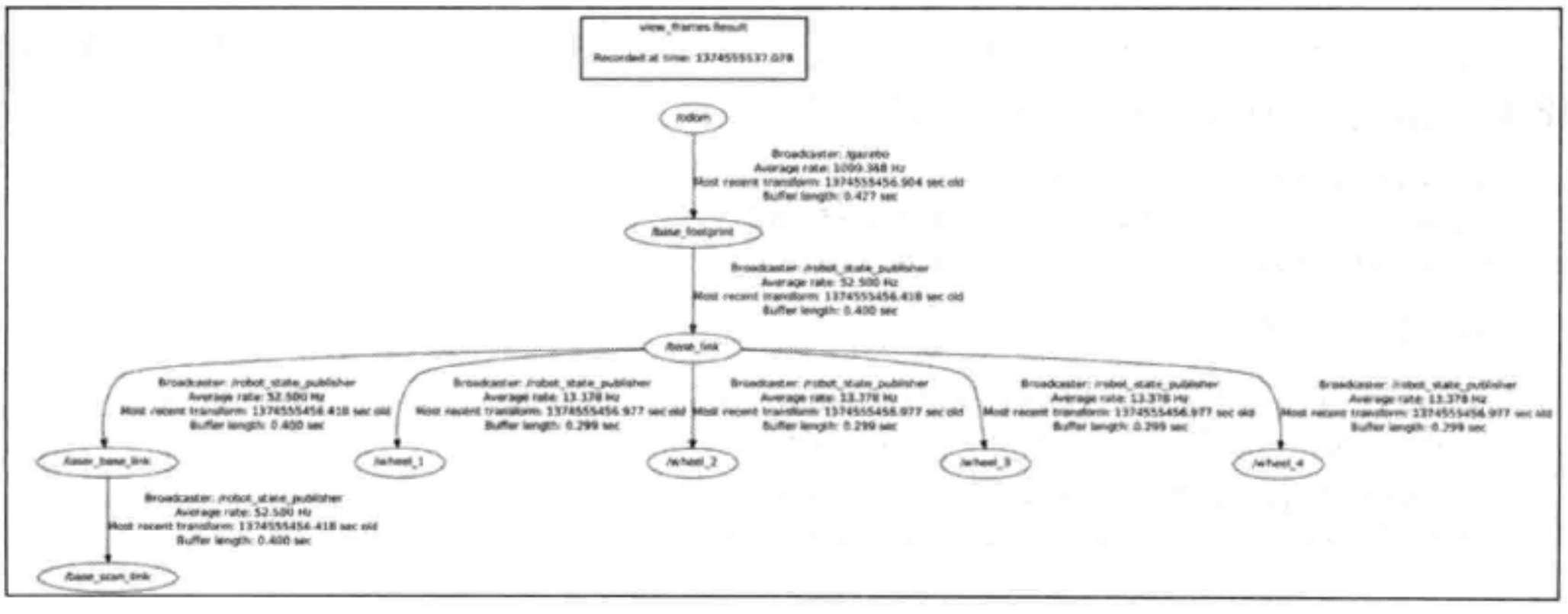
在开始编写代码进行组件的配置之前，你要记住首先要有完整的机器人几何外形的定义文件（URDF 文件）。在第 5 章中，我们就已经定义好了我们自己机器人的 URDF 文件。而且你可能已经忘记了，我们甚至还使用过 robot_state_publisher 功能包发布过我们机器人的坐标变换树。这样，其实你已经有了一个能够应用导航功能包集的机器人配置。基于以上原因，我们没有必要在这里再对机器人配置进行介绍和说明了。

7.2.3 查看坐标变换树

如果你想查看自己机器人的变换树，那么使用以下命令：

```
$ roslaunch chapter7_tutorials gazebo_map_robot.launch model:="`rospack
find chapter7_tutorials`/urdf/robot1_base_04.xacro"
$ rosrun tf view_frames
```

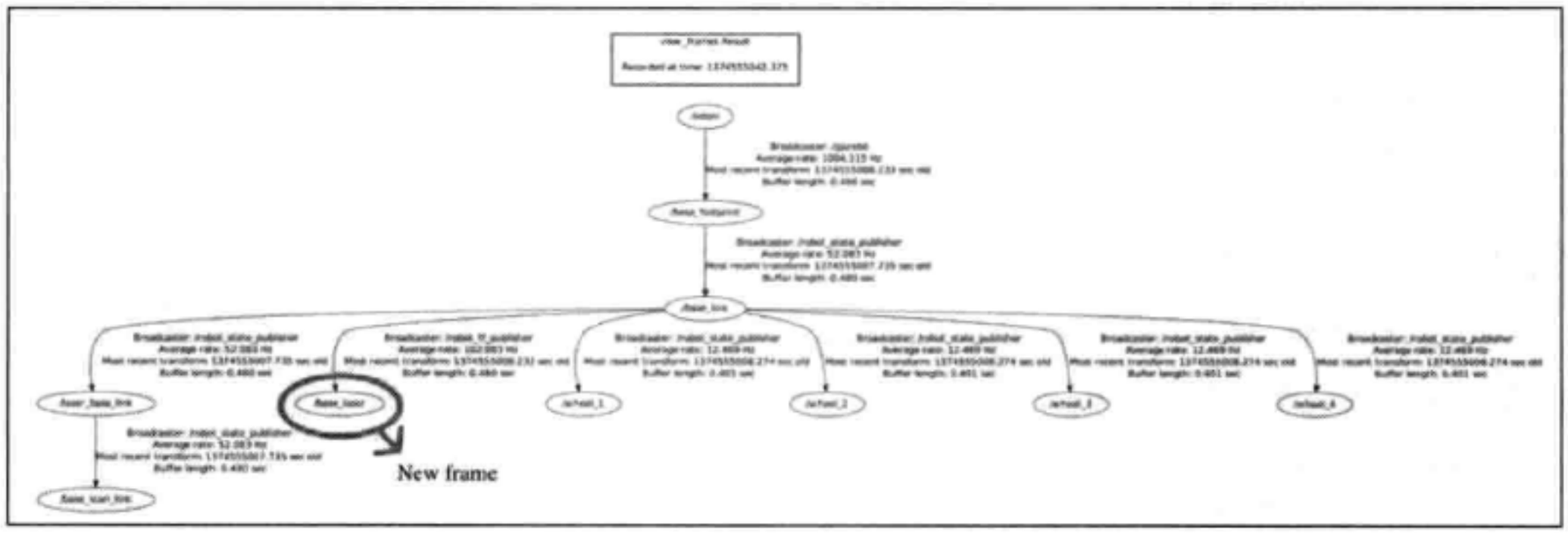
坐标变换树使用如下图形描述：



现在，你可以再一次运行 tf_broadcaster 和 rosrun tf view_frames 命令，你会看到你通过代码创建的坐标系：

```
$ rosrun chapter7_tutorials tf_broadcaster
$ rosrun tf view_frames
```

坐标变换树使用如下图形描述：



7.3 发布传感器信息

你的机器人可以有很多个传感器来感知世界，你可以编写很多个节点来处理这些数据并实现一些功能，但是导航功能包集仅支持使用平面激光雷达传感器。所以，你的传感器必须使用以下格式发布数据：sensor_msgs/LaserScan 或 sensor_msgs/ PointCloud。

我们将要在 Gazebo 中使用位于机器人前部的激光雷达进行导航。记住这个激光雷达是在 Gazebo 中仿真出来的，它在 base_scan/scan 坐标系上发布数据。

如果使用我们原有的机器人模型，那么不需要再进行任何配置就能够在导航功能包集中使用我们的激光雷达了。这是因为我们在 .urdf 文件中配置过坐标变换树，而且激光雷达也能够使用正确的格式发布数据。

如果你使用了一个真实的激光雷达，ROS 很可能已经包含了它的驱动。请参考第 4 章中如何在 ROS 中使用 Hokuyo 激光雷达的相关内容。但是你还是有可能会用到一个 ROS 没有提供官方驱动的激光雷达，这时你需要自己编写一个节点来发布传感器数据，其数据格式需要是 sensor_msgs/LaserScan。在下一节中我们会专门介绍相关功能的实现，并为你提供一个示例模板。

首先还是请你记住消息的类型是 sensor_msgs/LaserScan。使用以下命令：

```
$ rosmsg show sensor_msgs/LaserScan

std_msgs/Header header
  uint32 seq
  time stamp
```

```
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

创建激光雷达节点

现在我们在 chapter7_tutorials/src 文件夹下以 laser.cpp 为名创建一个新的文件，并加入以下代码：

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

int main(int argc, char** argv){
  ros::init(argc, argv, "laser_scan_publisher");

  ros::NodeHandle n;
  ros::Publisher scan_pub = n.advertise<sensor_
msgs::LaserScan>("scan", 50);

  unsigned int num_readings = 100;
  double laser_frequency = 40;
  double ranges[num_readings];
  double intensities[num_readings];

  int count = 0;
  ros::Rate r(1.0);
  while(n.ok()){
    //generate some fake data for our laser scan
    for(unsigned int i = 0; i < num_readings; ++i){
      ranges[i] = count;
      intensities[i] = 100 + count;
    }
    ros::Time scan_time = ros::Time::now();
    //populate the LaserScan message
    sensor_msgs::LaserScan scan;
    scan.header.stamp = scan_time;
    scan.header.frame_id = "base_link";
    scan.angle_min = -1.57;
    scan.angle_max = 1.57;
    scan.angle_increment = 3.14 / num_readings;
```

```
    scan.time_increment = (1 / laser_frequency) / (num_readings);
    scan.range_min = 0.0;
    scan.range_max = 100.0;

    scan.ranges.resize(num_readings);
    scan.intensities.resize(num_readings);
    for(unsigned int i = 0; i < num_readings; ++i){
      scan.ranges[i] = ranges[i];
      scan.intensities[i] = intensities[i];
    }

    scan_pub.publish(scan);
    ++count;
    r.sleep();
  }
}
```

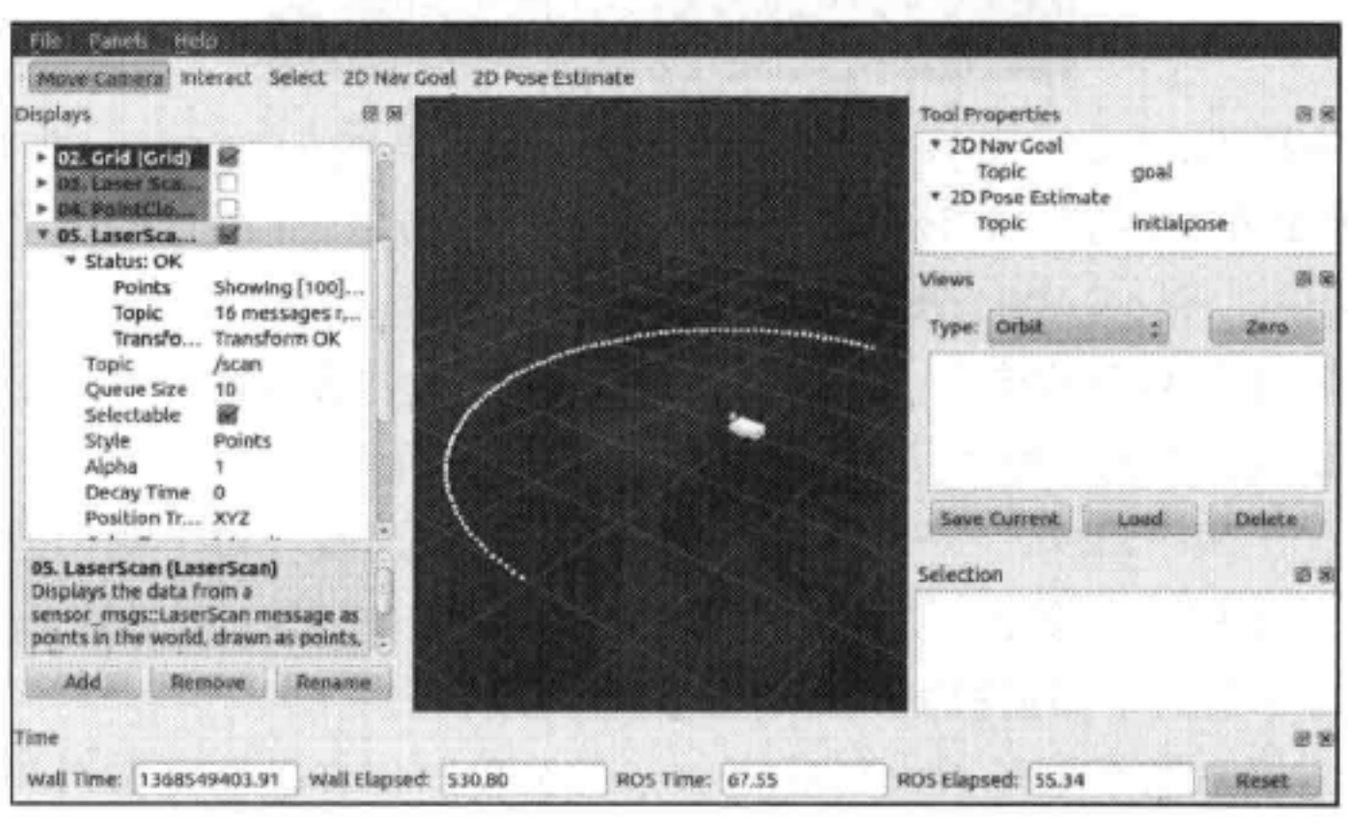
正如你所见，我们将要以 scan 为名创建一个新的主题，其消息格式为 sensor_msgs/LaserScan。你应该对这种消息格式比较熟悉，它在第 4 章出现过。还有，主题的名称必须是唯一的。当你配置导航功能包集时，需要选择这个主题来进行导航。以下代码显示了如何使用正确的名称创建主题：

```
ros::Publisher scan_pub = n.advertise<sensor_msgs::LaserScan>("scan",
50);
```

你必须要发布 header、stamp 和 frame_id 等字段的数据。因为要想让导航功能包集运行起来，必须有这些数据的支持：

```
scan.header.stamp = scan_time;
scan.header.frame_id = "base_link";
```

在 header 中的 frame_id 字段也是非常重要的。它必须是在 .urdf 文件中已经创建好的坐标系，且其坐标系相关数据也已经发布到 tf 坐标变换中。导航功能包集会通过这些信息获取传感器的真实位置并进行变换，例如在数据传感器和障碍物的坐标系之间的变换。



通过使用这个示例模板，你可以使用任何激光雷达，而无需考虑 ROS 中是否已经有了它的驱动。你只需要用你的激光雷达上的真实数据来替代模板中的默认数据。

这个模板还能用于将其他传感器通过数据格式转换伪装成一个激光雷达。例如，你能够将双目视觉测距系统或声呐传感器模拟成一个激光雷达。

7.4 发布里程数据

导航功能包集还需要获取机器人的里程信息。里程信息指的是机器人相对于某一点的距离。在我们的示例中，它应该是从 base_link 坐标系原点到 odom 坐标系原点的距离。

导航功能包集使用的消息类型是 nav_msgs/Odometry。我们可以使用以下指令查看消息的数据结构：

```
$ rosmsg show nav_msgs/Odometry
```

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

正如在消息中所见，nav_msgs/Odometry 提供了从机器人 frame_id 坐标系到 child_frame_id 坐标系的相对位置。它还通过 geometry_msgs/Pose 消息提供了机器人的位姿信息，通过 geometry_msgs/Twist 消息提供了速度信息。

位姿信息中包含着两个结构，一个显示了欧拉坐标系中的位置，另一个则使用了一个四元向量显示了机器人的方向。机器人的方向也是机器人的角位移。

速度信息包含了两个结构，一个是线速度，一个是角速度。对于我们的机器人，我们经常使用的是线速度 x 和角速度 z。使用线速度 x 是为了知道机器人是在向前移动还是在向后移动。使用角速度 z 是为了知道机器人是否在向左转或者向右转。

因为里程其实就是两个坐标系之间的位移，那么我们就有必要发布两个坐标系之间的坐

标变换信息。我们在这里已经对最后一个点进行了里程变换，然后还会在本节的后续部分继续介绍如何发布里程数据以及机器人的坐标变换树信息。

现在，让我们看看 Gazebo 是如何处理里程信息的。

7.4.1 Gazebo 如何获取里程数据

正如你在其他关于 Gazebo 的示例中所见的，我们的机器人在仿真环境中的移动和真实世界中的机器人是一样的。机器人使用的驱动插件是 `diffdrive_plugin`。我们曾经在第 5 章中配置过这个插件。当时，我们在 Gazebo 中创建了机器人并通过这个插件驱动机器人。

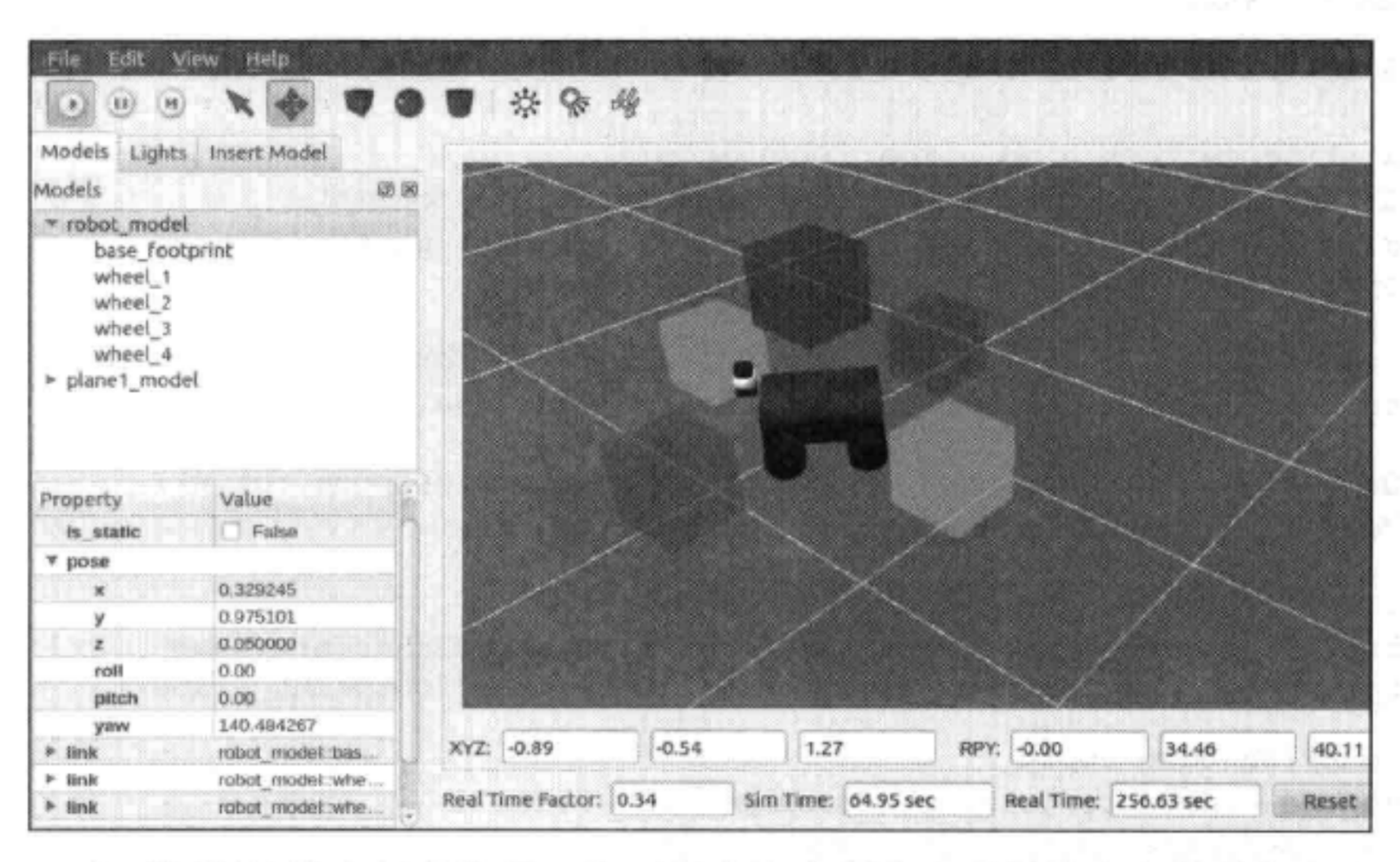
这个驱动就会发布机器人在仿真环境中的里程信息，所以我们并不需要为 Gazebo 再去编写任何代码。

在命令行窗口中输入以下命令在 Gazebo 中执行我们的示例机器人，并查看里程数据：

```
$ roslaunch chapter7_tutorials gazebo_xacro.launch model:="`rospack find
chapter7_tutorials`/urdf/robot1_base_04.xacro"
$ rosrun erratic_teleop erratic_keyboard_teleop
```

然后，通过 teleop 远程操作节点，先移动机器人运行一段时间，以便在 odometry 里程主题中生成足够多的数据。

在 Gazebo 仿真环境的界面上，如果你点击 robot_model1，就会看到各种组件和字段的属性值，其中一个属性是机器人的位姿（pose）。点击位姿，你就会看到相应字段的值。你现在能够看到的是机器人在仿真环境中的位置。如果你移动机器人，这个值就会发生改变。



Gazebo 会不间断地发布里程数据。你可以点击主题来查看具体发布的数据，也可以在命令行窗口中输入以下命令：

```
$ rostopic echo /odom/pose/pose
```

你会看到以下输出：

```
---
position:
  x: 0.32924763712
  y: 0.97509878254
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.941128847661
  w: 0.33804806182
---
```

注意这些数据和你在 Gazebo 屏幕上看到的数据是一致的。

正如你所看到的，Gazebo 在机器人移动时会发布里程数据。我们来仔细看一下这个插件的源代码以便弄明白 Gazebo 是如何获取这些数据的。

插件的源文件保存在 erratic_gazebo_plugins 功能包中，具体的文件名是 diffdrive_plugin.cpp。打开文件之后，你会看到如下源代码：

```
$ rosed erratic_gazebo_plugins diffdrive_plugin.cpp
```

这个文件有很多行代码，但是对于我们来说最重要的部分就是下面这个 publish_odometry() 函数：

```
void DiffDrivePlugin::publish_odometry()
{
 ros::Time current_time = ros::Time::now();
 std::string odom_frame = tf::resolve(tf_prefix_, "odom");
 std::string base_footprint_frame = tf::resolve(tf_prefix_, "base_
footprint");

 // getting data for base_footprint to odom transform
 math::Pose pose = this->parent->GetState().GetPose();

 btQuaternion qt(pose.rot.x, pose.rot.y, pose.rot.z, pose.rot.w);
 btVector3 vt(pose.pos.x, pose.pos.y, pose.pos.z);

 tf::Transform base_footprint_to_odom(qt, vt);
 transform_broadcaster_->sendTransform(tf::StampedTransform(base_
footprint_to_odom, current_time, odom_frame, base_footprint_frame));

 // publish odom topic
 odom_.pose.pose.position.x = pose.pos.x;
 odom_.pose.pose.position.y = pose.pos.y;

 odom_.pose.pose.orientation.x = pose.rot.x;
 odom_.pose.pose.orientation.y = pose.rot.y;
```

```
  odom_.pose.pose.orientation.z = pose.rot.z;
  odom_.pose.pose.orientation.w = pose.rot.w;

  math::Vector3 linear = this->parent->GetWorldLinearVel();
  odom_.twist.twist.linear.x = linear.x;
  odom_.twist.twist.linear.y = linear.y;
  odom_.twist.twist.angular.z = this->parent->GetWorldAngularVel().z;
  odom_.header.stamp = current_time;
  odom_.header.frame_id = odom_frame;
  odom_.child_frame_id = base_footprint_frame;

  pub_.publish(odom_);
}
```

publish_odometry() 函数的功能就是发布里程数据。你能看到结构体的各个字段是如何被赋值的，如何设定里程主题的名称（在本例中就是 odom）。机器人位姿数据创建是在代码的其他部分中完成的。我们会在后面的小节中进行具体介绍。

一旦你懂得了 Gazebo 是如何获取里程数据，下一步就是去学习它是如何发布里程数据和相对一个真实机器人的 tf 数据的。在下面的代码中，会展示一个机器人一直围绕圆形运动。最终的运行结果并不重要，重要的是通过学习代码理解如何正确地发布机器人的数据。

7.4.2 创建自定义里程数据

在 chapter7_tutorials/src 文件夹下以 odometry.cpp 为名创建一个新的文件，并加入以下代码：

```
#include <string>
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>

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

ros::init(argc, argv, "state_publisher");
    ros::NodeHandle n;
    ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom",
10);

    // initial position
    double x = 0.0;
    double y = 0.0;
    double th = 0;

    // velocity
    double vx = 0.4;
    double vy = 0.0;
    double vth = 0.4;
```

```
    ros::Time current_time;
    ros::Time last_time;
    current_time = ros::Time::now();
    last_time = ros::Time::now();

    tf::TransformBroadcaster broadcaster;
    ros::Rate loop_rate(20);

    const double degree = M_PI/180;

    // message declarations
    geometry_msgs::TransformStamped odom_trans;
    odom_trans.header.frame_id = "odom";
    odom_trans.child_frame_id = "base_footprint";

    while (ros::ok()) {
        current_time = ros::Time::now();

        double dt = (current_time - last_time).toSec();
        double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
        double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
        double delta_th = vth * dt;

        x += delta_x;
        y += delta_y;
        th += delta_th;

        geometry_msgs::Quaternion odom_quat;
        odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);

        // update transform
        odom_trans.header.stamp = current_time;
        odom_trans.transform.translation.x = x;
        odom_trans.transform.translation.y = y;
        odom_trans.transform.translation.z = 0.0;
        odom_trans.transform.rotation = tf::createQuaternionMsgFromYa
w(th);

        //filling the odometry
        nav_msgs::Odometry odom;
        odom.header.stamp = current_time;
        odom.header.frame_id = "odom";
        odom.child_frame_id = "base_footprint";

        // position
        odom.pose.pose.position.x = x;
        odom.pose.pose.position.y = y;
        odom.pose.pose.position.z = 0.0;
        odom.pose.pose.orientation = odom_quat;

        // velocity
```

```
        odom.twist.twist.linear.x = vx;
        odom.twist.twist.linear.y = vy;
        odom.twist.twist.linear.z = 0.0;
        odom.twist.twist.angular.x = 0.0;
        odom.twist.twist.angular.y = 0.0;
        odom.twist.twist.angular.z = vth;

    last_time = current_time;

        // publishing the odometry and the new tf
        broadcaster.sendTransform(odom_trans);
        odom_pub.publish(odom);

    loop_rate.sleep();
  }
  return 0;
}
```

首先，创建一个坐标变换的结构体变量，并分别给 frame_id 字段和 child_ frame_id 字段赋值，以便能够知道何时发生了移动。在我们的示例中，基本坐标系 base_footprint 将会相对于 odom 坐标系移动：

```
geometry_msgs::TransformStamped odom_trans;
    odom_trans.header.frame_id = "odom";
    odom_trans.child_frame_id = "base_footprint";
```

在这段代码中，还会生成机器人的位姿信息。然后根据线速度和角速度还能够计算一段时间后机器人的理论位置：

```
double dt = (current_time - last_time).toSec();
double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
double delta_th = vth * dt;

x += delta_x;
y += delta_y;
th += delta_th;

geometry_msgs::Quaternion odom_quat;
odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);
```

在本书中，你可能找不到关于机器人运动学的任何解释说明，因为你从网络上就能找到相关的大量文献资料。例如，你可以查找“轮式移动机器人运动学”。

在这个坐标变换结构体中，我们将只给 x 和 rotation 字段赋值，因为我们的机器人只能前后运动和转向：

```
        odom_trans.header.stamp = current_time;
        odom_trans.transform.translation.x = x;
        odom_trans.transform.translation.y = 0.0;
        odom_trans.transform.translation.z = 0.0;
        odom_trans.transform.rotation = tf::createQuaternionMsgFromYa
w(th);
```

对于里程的结构体变量，也要做一样的事情。将 odom 坐标系和 base_footprint 坐标系分别赋值给 frame_id 和 child_frame_id。

里程的结构体变量里面还包含着两个结构体。我们为 pose 结构体中的 x、y 和 orientation 字段分别赋值。在 twist 结构体中，我们分别为线速度 x 和角速度 z 赋值：

```
// position
odom.pose.pose.position.x = x;
odom.pose.pose.position.y = y;
odom.pose.pose.orientation = odom_quat;

// velocity
odom.twist.twist.linear.x = vx;
odom.twist.twist.angular.z = vth;
```

一旦完成对所有必须的字段的赋值，则发布数据：

```
// publishing the odometry and the new tf
broadcaster.sendTransform(odom_trans);
odom_pub.publish(odom);
```

请记住你需要在 CMakeLists.txt 文件中添加以下行：

```
rosbuild_add_executable(odometry src/odometry.cpp)
```

之后使用以下命令进行编译：

```
rosmake chapter7_tutorials:
```

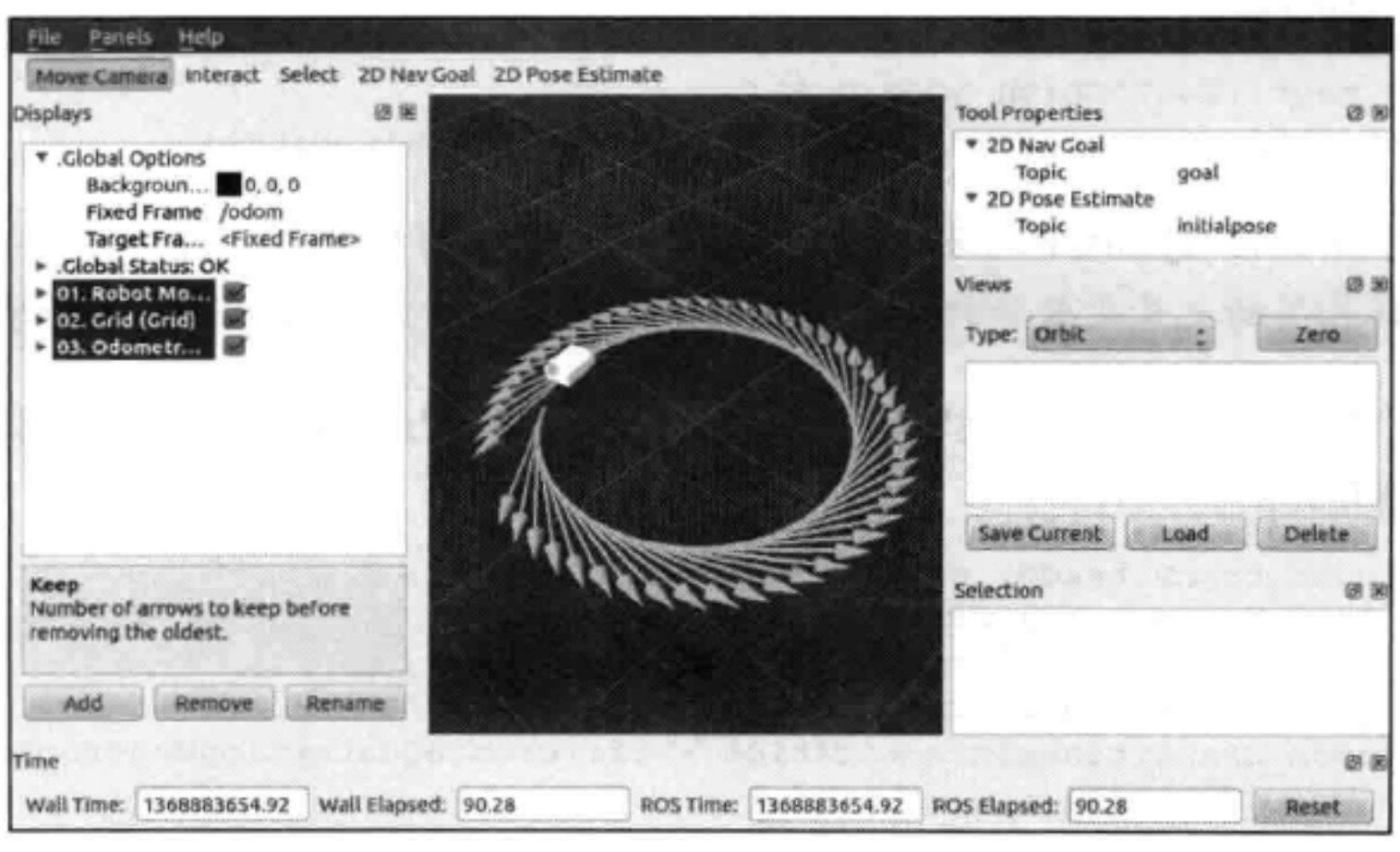
编译完功能包之后，调用以下命令启动机器人并使用 rviz 来查看模型和机器人的运动：

```
$ roslaunch chapter7_tutorials display_xacro.launch model:="`rospack find
chapter7_tutorials`/urdf/robot1_base_04.xacro"
```

使用以下命令运行里程数据处理节点：

```
$ rosrun chapter7_tutorials odometry
```

你会看到如下图所示：



在 rviz 的屏幕上，你能看到机器人沿着一些红色的箭头移动，并跨越不同的背景网格。机器人跨越网格，说明了系统正在为机器人发布新的坐标变换。红色的箭头则图形化表示了里程信息。如果一直运动下去，你会看到正如我们在代码中所定义的那样，机器人是围绕着一个圆形不断地运动。

7.5 创建基础控制器

对于导航功能包集来说，一个基础控制器是非常重要的，因为这是唯一能够有效地控制机器人的方法。它能够直接和你的机器人的电子设备通信。

ROS 并不提供任何标准的基础控制器，因此你必须自己编写针对你机器人平台的基础控制器。

你的机器人是通过 geometry_msgs/Twist 类型的消息所控制的。这个类型正是我们之前看到的 Odometry 消息所使用的。

你的基础控制器必须订阅以 cmd_vel 为名称的主题，必须生成正确的线速度和角速度命令来驱动机器人。

现在我们先复习一下消息的结构。在命令行窗口内输入以下命令查看消息的具体结构：

```
$ rosmsg show geometry_msgs/Twist
```

这个命令的输出结果如下所示：

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

其中，线速度向量 linear 包含了 x、y 和 z 轴的线速度。角速度向量 angular 包含了各个轴向的角速度。

对于我们的机器人，只需要使用线速度 x 和角速度 z。这是因为我们的机器人是基于差分轮驱动平台的，驱动它的两个电机只能够让机器人前进、后退或者转向。

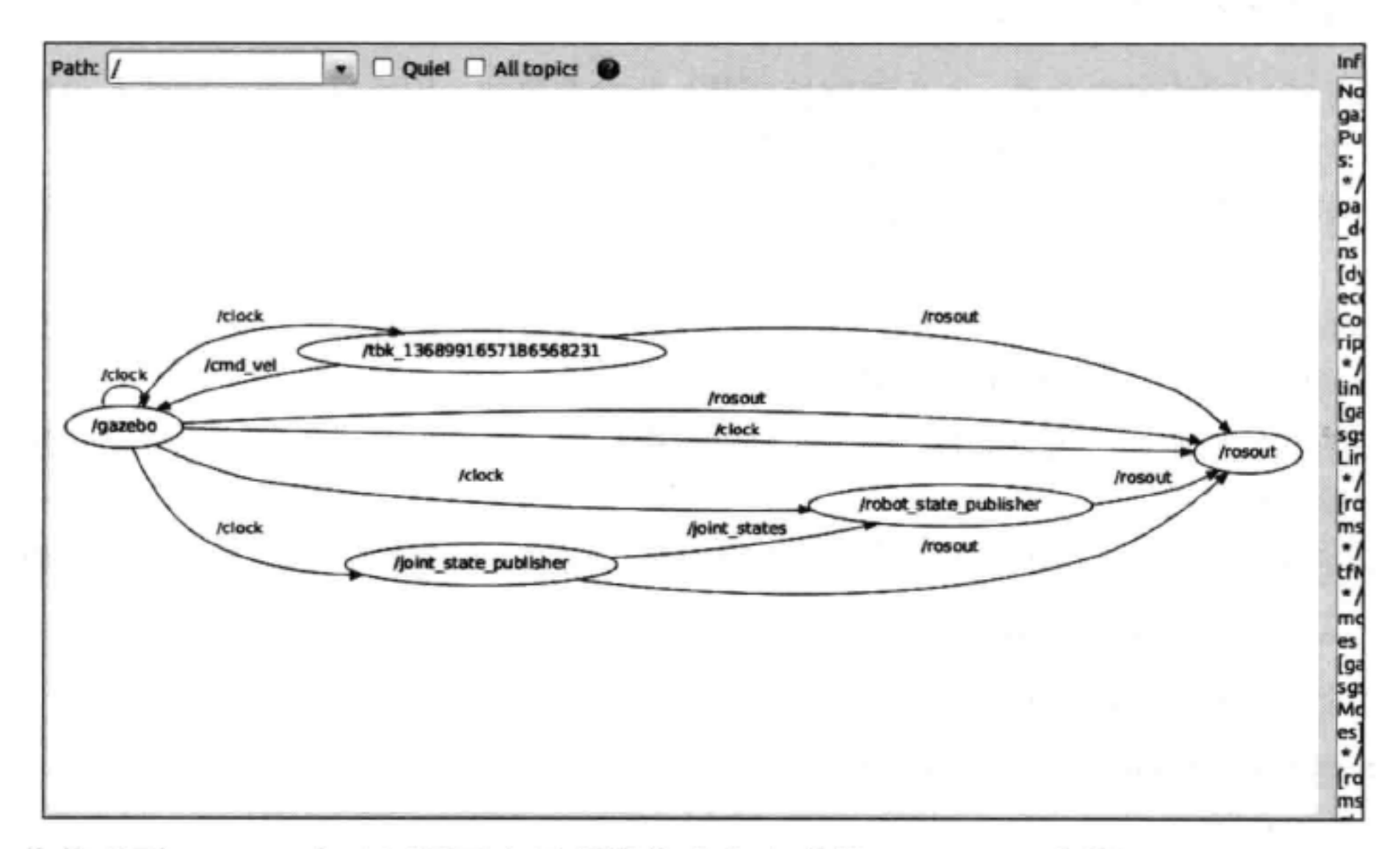
我们先在 Gazebo 中对这个机器人进行仿真，模拟使用基础控制器来驱动机器人的运动和仿真。这也就是说在 Gazebo 中我们并不需要为机器人实际创建一个基础控制器。

在本章中，你还会学到如何实现一个实际物理机器人的基础控制器。在此之前，先在 Gazebo 中运行我们的机器人，来理解基础控制器的作用。在一个新的命令行窗口中，运行以下命令：

```
$ roslaunch chapter7_tutorials gazebo_xacro.launch model:="`rospack find
chapter7_tutorials`/urdf/robot1_base_04.xacro"
$ rosrun erratic_teleop erratic_keyboard_teleop
```

在所有节点都启动和正常运行之后，我们使用节点状态图命令 rxgraph 来查看各个节点之间的关系：

```
$ rxgraph
```



你能看到 Gazebo 自动订阅了由远程操作节点生成的 cmd_vel 主题。

在 Gazebo 仿真环境中，正运行着我们的差分轮式机器人仿真程序插件。仿真程序插件通过 cmd_vel 主题获取控制命令数据。同时，仿真程序插件在 Gazebo 环境中完成机器人的驱动并生成里程信息。

7.5.1 使用 Gazebo 创建里程数据

为了了解 Gazebo 内部的工作原理，我们来仔细查看一下仿真程序插件 diffdrive_plugin.cpp 文件中的代码：

```
$ rosed erratic_gazebo_plugins diffdrive_plugin.cpp
```

Load(...) 函数会完成对主题的订阅，然后当 cmd_vel 主题中的消息到达时，cmdVelCallback() 函数会执行对消息的处理：

```
void DiffDrivePlugin::Load(physics::ModelPtr _parent, sdf::ElementPtr
_sdf)
{
…
```

```
…
ros::SubscribeOptions so =
    ros::SubscribeOptions::create<geometry_msgs::Twist>(topicName,
1, boost::bind(&DiffDrivePlugin::cmdVelCallback, this, _1),
ros::VoidPtr(), &queue_);

}
```

当消息到达的时候，线速度和角速度被存储为内部变量，然后执行以下操作：

```
void DiffDrivePlugin::cmdVelCallback(const geometry_
msgs::Twist::ConstPtr& cmd_msg)
{
  ...
  ...
    x_ = cmd_msg->linear.x;
    rot_ = cmd_msg->angular.z;
  ...
  ...
}
```

仿真程序插件会调用机器人运动学模块中的计算公式对每个电机的速度进行估计，如下面函数所示：

```
void DiffDrivePlugin::GetPositionCmd()
{
  ...
    vr = x_;
    va = rot_;

    wheelSpeed[LEFT] = vr + va * wheelSeparation / 2.0;
    wheelSpeed[RIGHT] = vr - va * wheelSeparation / 2.0;
  ...
}
```

最后，它会调用更多动力学模块中与机器人运动模型相匹配的函数，对机器人走过的路径进行估计。正如你在代码中看到的，你必须知道机器人轮子的直径（wheelDiameter）和左右轮子的间距（wheelSeparation）：

```
// Update the controller
void DiffDrivePlugin::UpdateChild()
{
    ...
  ...
    wd = wheelDiameter;
    ws = wheelSeparation;

    // Distance travelled by front wheels
    d1 = stepTime * wd / 2 * joints[LEFT]->GetVelocity(0);
    d2 = stepTime * wd / 2 * joints[RIGHT]->GetVelocity(0);

    dr = (d1 + d2) / 2;
    da = (d1 - d2) / ws;
```

```
    // Compute odometric pose
    odomPose[0] += dr * cos(odomPose[2]);
    odomPose[1] += dr * sin(odomPose[2]);
    odomPose[2] += da;

    // Compute odometric instantaneous velocity
    odomVel[0] = dr / stepTime;
    odomVel[1] = 0.0;
    odomVel[2] = da / stepTime;
...
  ...
}
```

这就是 diffdrive_plugin 仿真程序插件在 Gazebo 中控制我们仿真机器人的实现方法。

7.5.2 创建基础控制器

下面要做的工作和刚才有些类似。也就是说上面的很多代码也可以直接用在带有两个驱动轮、电机和编码器的实际机器人身上。在 chapter7_tutorials/src 文件夹中以 base_controller. cpp 为名创建一个新文件，并在其中添加以下代码：

```
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <iostream>

using namespace std;

double width_robot = 0.1;
double vl = 0.0;
double vr = 0.0;
ros::Time last_time;
double right_enc = 0.0;
double left_enc = 0.0;
double right_enc_old = 0.0;
double left_enc_old = 0.0;
double distance_left = 0.0;
double distance_right = 0.0;
double ticks_per_meter = 100;
double x = 0.0;
double y = 0.0;
double th = 0.0;
geometry_msgs::Quaternion odom_quat;

void cmd_velCallback(const geometry_msgs::Twist &twist_aux)
{
    geometry_msgs::Twist twist = twist_aux;
    double vel_x = twist_aux.linear.x;
    double vel_th = twist_aux.angular.z;
    double right_vel = 0.0;
```

```
    double left_vel = 0.0;

    if(vel_x == 0){
// turning
        right_vel = vel_th * width_robot / 2.0;
        left_vel = (-1) * right_vel;
    }else if(vel_th == 0){
// forward / backward
        left_vel = right_vel = vel_x;
    }else{
// moving doing arcs
        left_vel = vel_x - vel_th * width_robot / 2.0;
        right_vel = vel_x + vel_th * width_robot / 2.0;
    }
    vl = left_vel;
    vr = right_vel;
}
int main(int argc, char** argv){
    ros::init(argc, argv, "base_controller");
    ros::NodeHandle n;
    ros::Subscriber cmd_vel_sub = n.subscribe("cmd_vel", 10, cmd_
velCallback);
    ros::Rate loop_rate(10);

    while(ros::ok())
    {

        double dxy = 0.0;
        double dth = 0.0;
        ros::Time current_time = ros::Time::now();
        double dt;
        double velxy = dxy / dt;
        double velth = dth / dt;

      ros::spinOnce();
        dt =  (current_time - last_time).toSec();;
        last_time = current_time;

        // calculate odomety
        if(right_enc == 0.0){
          distance_left = 0.0;
          distance_right = 0.0;
        }else{
          distance_left = (left_enc - left_enc_old) / ticks_per_meter;
          distance_right = (right_enc - right_enc_old) / ticks_per_
meter;
        }

        left_enc_old = left_enc;
        right_enc_old = right_enc;

         dxy = (distance_left + distance_right) / 2.0;
         dth = (distance_right - distance_left) / width_robot;

        if(dxy != 0){
          x += dxy * cosf(dth);
          y += dxy * sinf(dth);
```

```
        }

        if(dth != 0){
          th += dth;
        }
        odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);
        loop_rate.sleep();
      }
}
```

请注意这上面的代码和 diffdrive_plugin 仿真程序插件非常相似，这是因为这两个机器人都是差分轮式机器人。

不要忘记在 CMakeLists.txt 文件中插入以下代码并编译生成可执行文件：

```
rosbuild_add_executable(base_controller src/base_controller.cpp)
```

这段代码只是一段最简单最通用的示例，如果你要将这段代码移植到特定机器人上，那么你可能还要添加相当多的代码才能让它正常工作。这还取决于你所使用的控制器、编码器及机器人上的其他设备。如果你有一定的背景知识，就能够添加所需的代码并让你的机器人运转起来。

7.6 使用 ROS 创建地图

创建地图在有些时候是非常复杂的工作。因为你必须选择正确的工具来简化你的工作。ROS 就有这样的工具能够帮助你使用激光雷达和机器人的里程信息创建地图。这个工具是 map_server 地图服务器（http://www.ros.org/wiki/slam_gmapping）。在本示例中，你将会学到如何使用我们在 Gazebo 中创建的机器人来创建地图、保存地图和加载地图。

我们将会使用一个 .launch 文件来简化创建的过程。在 chapter7_tutorials/launch 文件夹下以 gazebo_mapping_robot.launch 为名称创建一个新文件，然后添加以下代码：

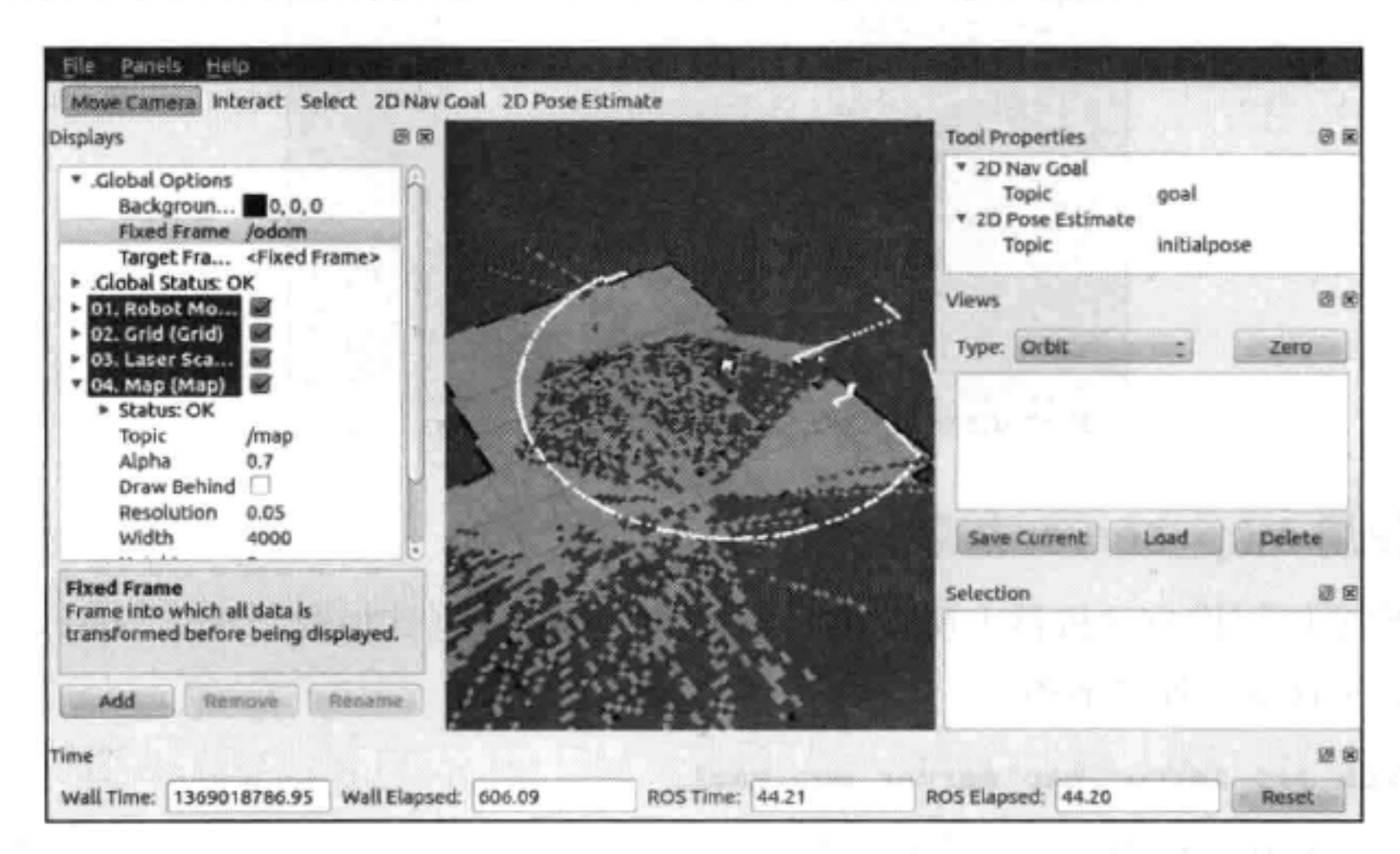
```
<?xml version="1.0"?>
<launch>
  <param name="/use_sim_time" value="true" />

  <!-- start up wg world -->
  <include file="$(find gazebo_worlds)/launch/wg_collada_world.
launch"/>

  <arg name="model" />
  <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />

  <node name="spawn_robot" pkg="gazebo" type="spawn_model" args="-urdf
-param robot_description -z 0.1 -model robot_model" respawn="false"
output="screen" />

  <node name="rviz" pkg="rviz" type="rviz" args="$(find chapter7_
tutorials/)launch/mapping.vcg"/>
```

```
  <node name="slam_gmapping" pkg="gmapping" type="slam_gmapping">
  <remap from="scan" to="base_scan/scan"/>
  </node>
</launch>
```

有了这个 .launch 文件，你就能够启动我们所需的 Gazebo 仿真环境。在这个仿真环境中，你不仅能够使用 3D 模型，还能够通过 rviz 配置文件和 slam_mapping 来实时构建地图。在命令行窗口中运行这个启动文件，在另一个命令行窗口中运行远程操作节点来移动机器人：

```
$ roslaunch chapter7_tutorials gazebo_mapping_robot.launch
model:="`rospack find chapter7_tutorials`/urdf/robot1_base_04.xacro"
$ rosrun erratic_teleop erratic_keyboard_teleop
```

当你使用键盘移动机器人的时候，你会在 rviz 屏幕上看到很多空白和未知的空间，也有一部分空间已经被地图所覆盖。这部分已知的地图就被称为**覆盖网格地图**（Occupancy Grid Map，OGM）。每当机器人移动时，slam_mapping 节点都会更新地图状态。或者更详细一点说，是在机器人移动之后，它会更新激光雷达和里程数据，并在对机器人的位置估计和地图的计算上获得更优的计算结果时，将 OGM 发布到屏幕上显示。



7.6.1 使用 map_server 保存地图

一旦完成了整个地图或者完成了你想要完成的部分，那么你就能够通过导航功能包集将地图保存起来并在下次使用时调用它。我们可以使用以下命令来保存地图：

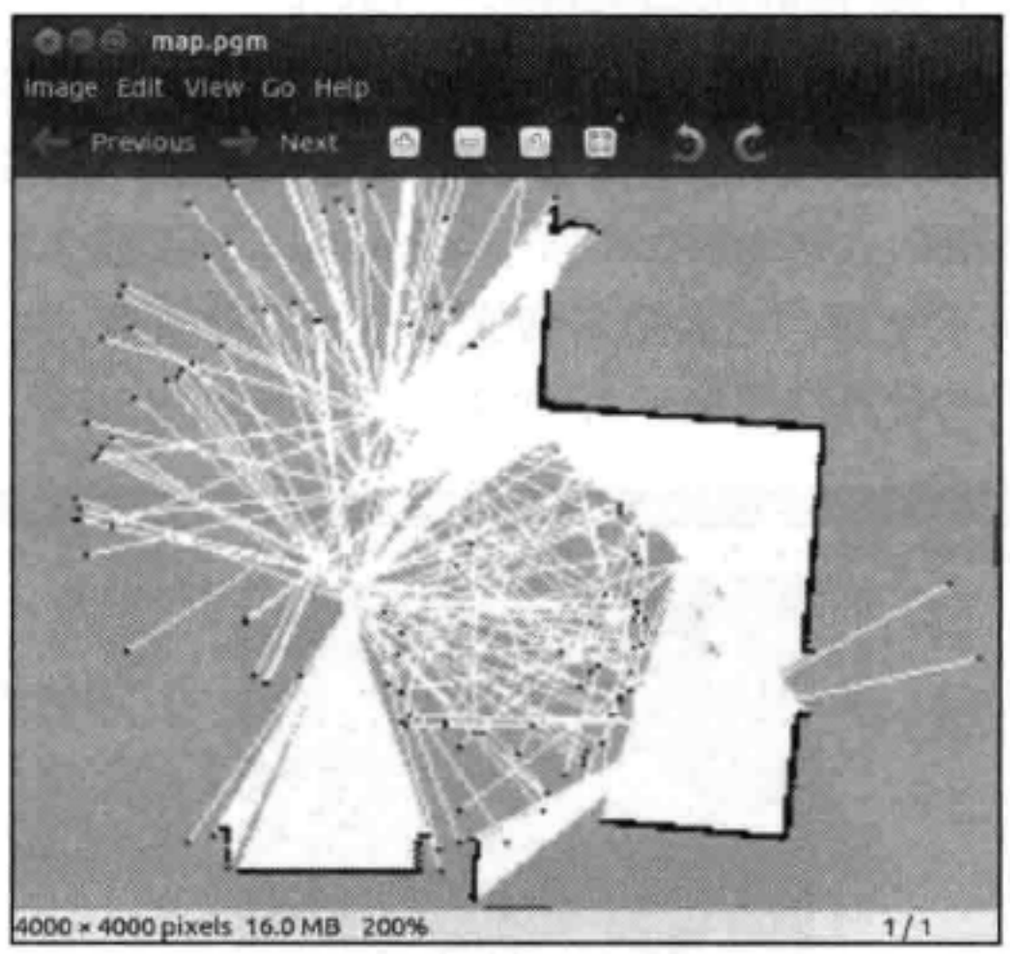
```
$ rosrun map_server map_saver -f map
```

```
[ INFO] [1374457427.140784742]: Waiting for the map
[ INFO] [1374457427.875320754, 19.644000000]: Received a 4000 X 4000 map @ 0.050 m/pix
[ INFO] [1374457427.876429277, 19.644000000]: Writing map occupancy data to map.pgm
[ INFO] [1374457432.247321199, 20.040000000]: Writing map occupancy data to map.yaml
[ INFO] [1374457432.257071257, 20.040000000]: Done
```

这个命令会创建两个文件，map.pgm 和 map.yaml。第一个文件是以 .pgm 为格式的地图（可输出灰色地图格式）；另一个则是该地图的配置文件。如果你打开它，你就会看到以下输出：

```
image: map.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

现在，你可以使用自己常用的图像查看工具打开 .pgm 图像文件，你将会看到你之前所创建的地图：



7.6.2 使用 map_server 加载地图

当你想要使用你自己机器人所创建的地图时，你必须将其加载到 map_server 功能包中。使用以下命令来加载地图：

```
$ rosrun map_server map_server map.yaml
```

但是为了简化我们的工作，我们也可以在 chapter7_tutorials/launch 文件夹下以 gazebo_map_robot.launch 为名称创建一个新的 .launch 启动文件，并添加以下代码：

```
<?xml version="1.0"?>
<launch>
   <param name="/use_sim_time" value="true" />
    <!-- start up wg world -->
    <include file="$(find gazebo_worlds)/launch/wg_collada_world.
launch"/>
    <arg name="model" />
    <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />
```

```
    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
    <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="state_publisher"
name="robot_state_publisher" output="screen" >
  <param name="publish_frequency" type="double" value="50.0" />
  </node>
    <node name="spawn_robot" pkg="gazebo" type="spawn_model"
args="-urdf -param robot_description -z 0.1 -model robot_model"
respawn="false" output="screen" />
    <node name="map_server" pkg="map_server" type="map_server" args="
$(find chapter7_tutorials)/maps/map.yaml" />
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find chapter7_
tutorials)/launch/mapping.vcg" />
</launch>
```

现在，使用以下命令来启动文件，记住你还要指定你所要使用的机器人模型：

```
$ roslaunch chapter7_tutorials gazebo_map_robot.launch model:="`rospack
find chapter7_tutorials`/urdf/robot1_base_04.xacro"
```

然后你就会在 rviz 中看到机器人和地图。为了能够获取机器人的位置，导航功能包集会使用地图服务器发布的地图数据和激光雷达的读数。这主要是通过使用扫描匹配算法来实现的，即使用自**适应蒙特卡罗定位**（adaptive Monte Carlo localization，AMCL）节点来估计机器人的位置。

在下一章中，我们会学习更多关于地图的知识和更多有用的工具。

7.7 本章小结

在本章中，你一步一步学习了使用导航功能包集之前，所需要为你的机器人进行的配置工作。介绍了应用导航功能包集对机器人平台的要求：需要使用平面激光雷达，需要使用差分轮式机器人，需要使用基础控制器，而且还要满足特定的几何形状。

请记住，我们之所以使用 Gazebo 来展示示例和解释导航功能包集的工作原理，是因为它的配置更加简单，也便于修改。如果直接使用一个真的机器人平台来解释这一切就会麻烦许多，而且并不是每位读者都能拥有或者能接触到这样的机器人。在任何情况下，使用不同的机器人平台，所下发的指令都有可能是不同的，其硬件执行机构也可能会出现故障，因此在仿真环境中执行这些代码更加安全也更加可靠。在此之后，在满足我们学习的基础之上，我们也可以将这些代码移植到实际的机器人上应用。

在下一章中，我们将会学习如何配置导航功能包集，创建 .launch 文件，并在 Gazebo 中完成我们之前所创建的机器人的自主导航。

简单来说，你在本章之后所学的内容都将会是非常有用的，因为它不仅会指导你在本书的示例中进行正确的配置，还会指导你在其他的机器人（无论是仿真机器人还是真正的机器人）中使用导航功能包集。

第 8 章
导航功能包集进阶

我们逐渐接近本书的结尾，而这正是使用在本书中获得所有知识的好机会。我们已经创建了功能包、节点、机器人的 3D 模型等。在第 7 章中，你对你的机器人进行了配置，以便于使用导航功能包集。那么在本章中，我们将会完成对导航功能包集的配置，这样你就能够学会如何在自己的机器人上使用它们。

在前面章节做的所有工作都成了现在工作的铺垫，而最大的乐趣也即将开始，这是赋予我们的机器人生命的时刻。

在本章中我们将会学习以下内容：

- 应用在第 7 章所学的知识，并继续程序开发。
- 理解导航功能包集及其工作方式。
- 配置所有所需文件。
- 创建启动文件并开始导航。

让我们开始吧。

8.1 创建功能包

想要正确创建功能包，并不是仅仅把文件放到一起，而是要为我们机器人创建的功能包添加依赖项。例如，你应该使用以下命令来创建功能包：

```
$ roscreate-pkg my_robot_name_2dnav move_base my_tf_configuration_dep my_
odom_configuration_dep my_sensor_configuration_dep
```

但是在我们的示例中，已经把所有代码都放在一个功能包中了，所以只需要执行以下命令：

```
$ roscreate-pkg chapter8_tutorials roscpp
```

记住，在 ROS 的软件源中，你也可能找到本章所需要的所有文件。

8.2 创建机器人配置

为了一次启动整个机器人，我们将会创建一个能够启动所有所需文件的 launch 启

动文件。

请记住这是在 Gazebo 中进行机器人仿真的示例。如果你学会了前面章节的内容，就应该知道 Gazebo 会为我们发布里程信息和坐标变换信息。

无论如何，如果你想为一台实际的机器人编写一个启动文件，你可以用下面的文件作为模板。

下面的代码都保存在 configuration_template.launch 文件中：

```
<launch>
  <node pkg="sensor_node_pkg" type="sensor_node_type" name="sensor_
node_name" output="screen">
    <param name="sensor_param" value="param_value" />
  </node>

  <node pkg="odom_node_pkg" type="odom_node_type" name="odom_node"
output="screen">
    <param name="odom_param" value="param_value" />
  </node>

  <node pkg="transform_configuration_pkg" type="transform_
configuration_type" name="transform_configuration_name"
output="screen">
    <param name="transform_configuration_param" value="param_value" />
  </node>
</launch>
```

这个文件会通过启动三个节点完成机器人的启动。

第一个节点用于负责完成传感器如**激光雷达系统**（Laser Imaging Detection and Ranging，LIDAR）的初始化配置。参数 sensor_param 用于配置传感器的端口。例如，有的传感器会使用 USB 接口。如果你配置的传感器要更多的参数，那么你需要增加相应的配置行和必要的参数。有些机器人具有能够用于导航功能包集的多个传感器 。在这样的情况下，你可以为这些传感器添加更多的节点，并在创建 launch 启动文件时，将必要的节点和工具都包含在启动文件中。从个人经验上讲，在同一个文件中管理这些节点会比较清晰，能够避免遗漏和错误。

第二个节点用于实现里程数据、基础控制器、机器人移动和定位等必需的功能。在第 7 章中，我们学习过这些内容。正如在前一章节一样，你能够使用参数在结构体的对应字段中赋值完成配置。如果必要的话，你可以添加节点并相应地添加配置参数。

第三个需要启动的节点主要负责发布和计算机器人的里程数据，各个传感器、手臂和关节之间的坐标变换等。

前面的文件主要是在你使用真实机器人时使用，而对于我们现在的示例，下面这个启动文件才是我们真正所需要的。

在 chapter8_tutorials/launch 文件夹下以 chapter8_configuration_gazebo.launch 为名创建一个文件，并添加以下代码：

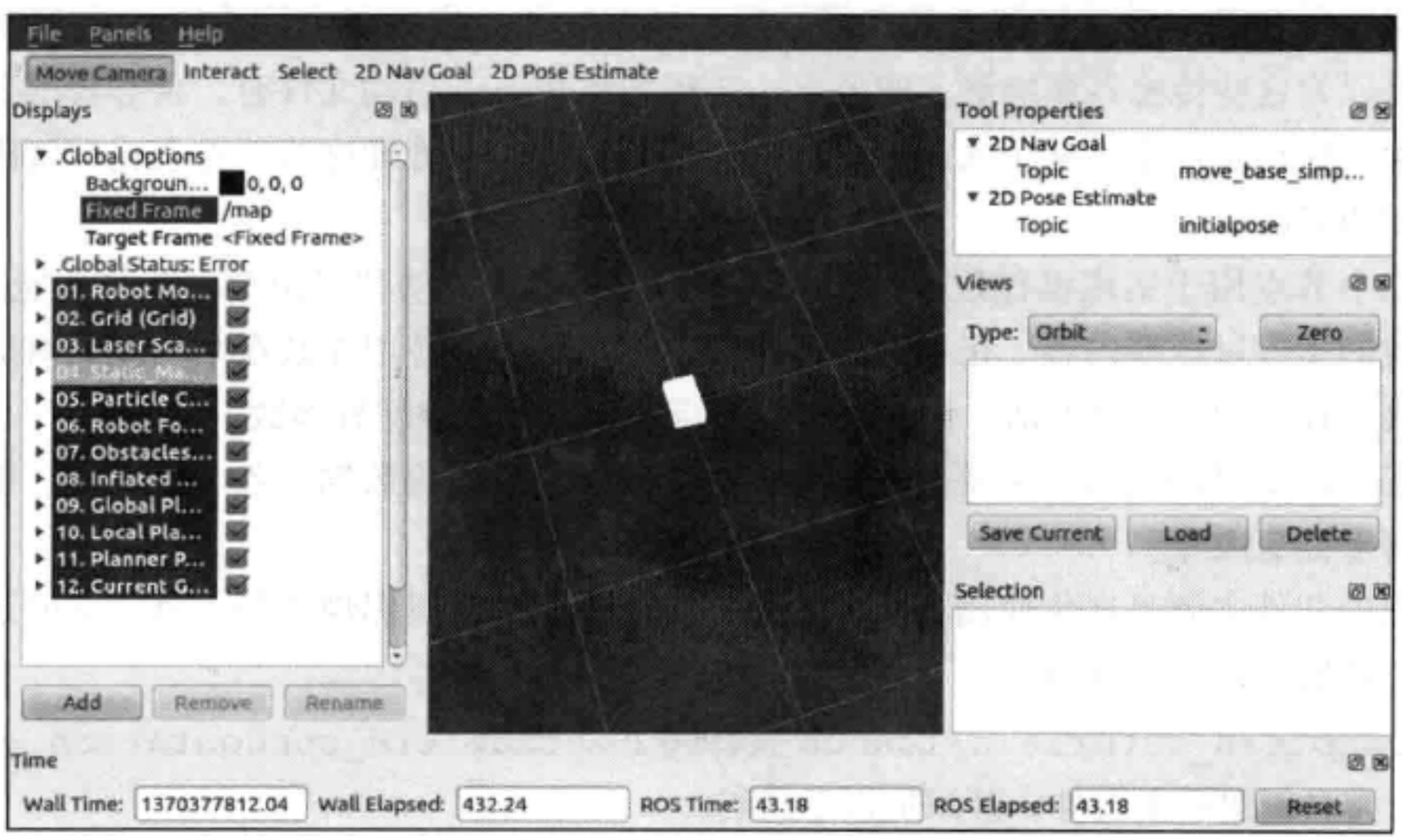
```
<launch>

  <param name="/use_sim_time" value="true" />

  <!-- start up wg world -->
  <include file="$(find gazebo_worlds)/launch/wg_collada_world.
launch"/>

  <arg name="model" default="$(find chapter8_tutorials)/urdf/robot1_
base_04.xacro"/>
  <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />

  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
  <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="state_publisher"
name="robot_state_publisher" output="screen" />

  <node name="spawn_robot" pkg="gazebo" type="spawn_model" args="-urdf
-param robot_description -z 0.1 -model robot_model" respawn="false"
output="screen" />

  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find chapter8_
tutorials)/launch/navigation.vcg" />

</launch>
```

这个文件其实和我们在前面几章中所使用的文件是完全相同的，所以就不做过多的解释了。

现在使用以下命令启动文件：

```
$ roslaunch chapter8_tutorials chapter8_configuration_gazebo.launch
```

你会看到如下窗口：



请注意在前面的截屏中，有一些字段在你没有进行任何配置时，显示出红色、蓝色和黄色的底纹。这是因为在启动文件中，包含了一个 rviz 布局的配置文件。这个文件会随着 rviz 启动而启动。它的具体配置在本书前面的几章中曾经介绍过。

在下面的章节中，我们会学习如何配置 rviz 并使其应用导航功能包集和查询所有的主题。

8.3 配置全局和局部代价地图

好的，现在我们将要开始配置导航功能包集和所有启动时必需的文件。为了开始进行配置，首先我们将会学习什么是代价地图和代价地图的作用。我们的机器人将使用两种导航算法在地图中移动——全局导航（global）和局部导航（local）。

- 全局导航用于建立到地图上最终目标或一个远距离目标的路径。
- 局部导航用于建立到近距离目标或者为了临时躲避障碍物的路径。例如，机器人四周一个 4m × 4m 的方形窗户。

这些导航算法通过使用代价地图来处理地图中的各种信息。全局代价地图用于全局导航，局部代价地图用于局部导航。

代价地图的参数用于配置算法计算行为。它们也有最基本的通用参数，这些参数会保存在共享的文件中。

我们可以在三个最基本的配置文件中设定不同的参数。这三个文件如下：

- costmap_common_params.yaml
- global_costmap_params.yaml
- local_costmap_params.yaml

你只需要看一下这三个文件的名称，就能大致猜出来它们的用途。现在你可能刚刚对代价地图的作用有一个初步的认识，让我们来创建这些配置文件并解释这些配置参数的作用。

8.3.1 基本参数的配置

让我们从最基本的参数讲起。在 chapter8_tutorials/launch 文件夹下以 costmap_common_params.yaml 为名称创建一个新文件，并添加以下代码：

```
obstacle_range: 2.5
raytrace_range: 3.0
footprint: [[-0.2,-0.2],[-0.2,0.2], [0.2, 0.2], [0.2,-0.2]]
#robot_radius: ir_of_robot
inflation_radius: 0.55

observation_sources: laser_scan_sensor

laser_scan_sensor: {sensor_frame: laser_base_link, data_type:
LaserScan, topic: /base_scan/scan, marking: true, clearing: true}
```

这个文件主要用于配置基本参数。这些参数会被用于 local_costmap 和 global_costmap。让我们对代码进行详细的解释。

obstacle_range 和 raytrace_range 参数用于表示传感器的最大探测距离并在代价地图中引入探测的障碍信息。前者主要用于障碍的探测。在我们的示例中，如果机器人检测到一个距离小于 2.5m 的障碍物，就会将这个障碍物引入到代价地图中。后者则用于在机器人运动过程中，实时清除代价地图中的障碍物，并更新可移动的自由空间数据。请注意，我们只能检测到障碍物对雷达激光信号投影或者声呐的回波，因此我们无法感知完整的物体形状和大小。但在实际应用中，这样的测量结果就足够我们完成地图的绘制和机器人自身的定位了。

参数 footprint 用于将机器人的几何参数告知导航功能包集。这样，就能在机器人和障碍物之间保持一个合理的距离，或者说提前获知机器人能否穿越某个门等。参数 inflation_radius 则给定了机器人与障碍物之间必须要保持的最小距离。

我们还需要配置 observation_sources 参数来设定导航功能包集所使用的传感器。只有配置了正确的传感器才能计算机器人在实际环境中的导航路径。

在我们的示例中，我们会在 Gazebo 中使用一个模拟的 LIDAR。当然也可以使用点云来完成类似的功能。

通过下面的代码，我们会对传感器的坐标系和数据进行配置：

```
laser_scan_sensor: {sensor_frame: laser_base_link, data_type:
LaserScan, topic: /base_scan/scan, marking: true, clearing: true}
```

上面配置的这些参数也会被用于在代价地图中添加和清除障碍。例如，你可以添加一个探测范围较大的传感器用于在代价地图中添加障碍物，再添加一个传感器用于导航和清除障碍物。在上面还会配置主题的名称，这是不能被遗漏的，如果你不进行配置，那么导航功能包集会使用默认的主题以保证程序能够正常运行，那么一旦你的机器人移动起来，很可能就会撞到墙上或者障碍物上。

8.3.2 全局代价地图的配置

下一个配置文件主要用于全局代价地图 global costmap 的配置。在 chapter8_tutorials/launch 文件夹下创建一个以 global_costmap_params.yaml 为名的新文件，并添加以下代码：

```
global_costmap:
  global_frame: /map
  robot_base_frame: /base_footprint
  update_frequency: 1.0
  static_map: true
```

其中，global_frame 和 robot_base_frame 参数用于定义机器人和地图之间的坐标变换。建立全局代价地图必须要使用这个变换。

你还能配置代价地图更新的频率。在我们的示例中，这个频率为 1Hz。参数 static_map 用于配置是否使用一个地图或者地图服务器来初始化全局代价地图。如果你不准备使用静态的地图，那么这个参数应该为 false。

8.3.3 局部代价地图的配置

下面的这个文件用于配置局部代价地图。在 chapter8_ tutorials/launch 文件夹下创建一个以 local_costmap_params.yaml 为名的新文件，并添加以下代码：

```
local_costmap:
  global_frame: /map
  robot_base_frame: /base_footprint
  update_frequency: 1.0
  publish_frequency: 2.0
  static_map: true
  rolling_window: false
  width: 10.0
  height: 10.0
  resolution: 0.1
```

其中，global_frame、robot_base_frame、update_frequency 和 static_map 参数和 8.2 节中所描述的内容一致。参数 publish_frequency 定义了发布信息的频率。参数 rolling_window 用于配置在机器人运动过程中，代价地图始终以机器人为中心。

你能够通过 width、height 和 resolution 参数来配置代价地图的尺寸和分辨率。这些参数都是以米为单位的。

8.4 基本局部规划器配置

一旦我们完成代价地图的配置，那么就能够开始进行一个基本规划器的配置了。这个基本规划器会产生一个速度命令来移动我们的机器人。在 chapter8_ tutorials/launch 文件夹下创建一个以 base_local_planner_params.yaml 为名的新文件，并添加以下代码：

```
TrajectoryPlannerROS:
  max_vel_x: 1
  min_vel_x: 0.5
  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: false
```

这个配置文件设定了我们机器人的最大和最小速度限值，同时也设定了加速度的限值。当你使用的是一台**全向移动机器人**（holonomic vehicle）时，那么参数 holonomic_

robot 就应设为 true。在我们的示例中，使用的不是全向移动机器人，所以这个参数是 false。全向移动机器人指的是它能够以任何方向向已配置的空间中任何位置移动。换句话说，如果机器人能够沿所配置的空间的 x 轴和 y 轴移动，这就意味着机器人能够全向移动。更简单一点讲，如果机器人能够前后左右地移动，那么就是全向移动机器人。一种典型的非全向移动的例子就是汽车，因为它就不能直接沿左右方向移动到一个给定的位置。如果从某个位置（和位姿）开始运动的话，它并不能一次到达地图上的每一个地点，也不能保证到达时的位姿。类似地，我们的差分移动机器人平台就是一个非全向移动机器人平台。

8.5 为导航功能包集创建启动文件

现在，我们已经创建和配置了所有导航功能包集所需的文件。为了运行导航功能包集，我们还要创建一个启动文件。在 chapter8_tutorials/launch 文件夹下创建一个以 move_base.launch 为名的新文件，并添加以下代码：

```
<launch>

    <!-- Run the map server -->
   <node name="map_server" pkg="map_server" type="map_server"
args="$(find chapter8_tutorials)/maps/map.yaml" output="screen"/>

  <include file="$(find amcl)/examples/amcl_diff.launch" >
  </include>

  <node pkg="move_base" type="move_base" respawn="false" name="move_
base" output="screen">
  <rosparam file="$(find chapter8_tutorials)/launch/costmap_common_
params.yaml" command="load" ns="global_costmap" />
  <rosparam file="$(find chapter8_tutorials)/launch/costmap_common_
params.yaml" command="load" ns="local_costmap" />
  <rosparam file="$(find chapter8_tutorials)/launch/local_costmap_
params.yaml" command="load" />
  <rosparam file="$(find chapter8_tutorials)/launch/global_costmap_
params.yaml" command="load" />
  <rosparam file="$(find chapter8_tutorials)/launch/base_local_
planner_params.yaml" command="load" />
  </node>
</launch>
```

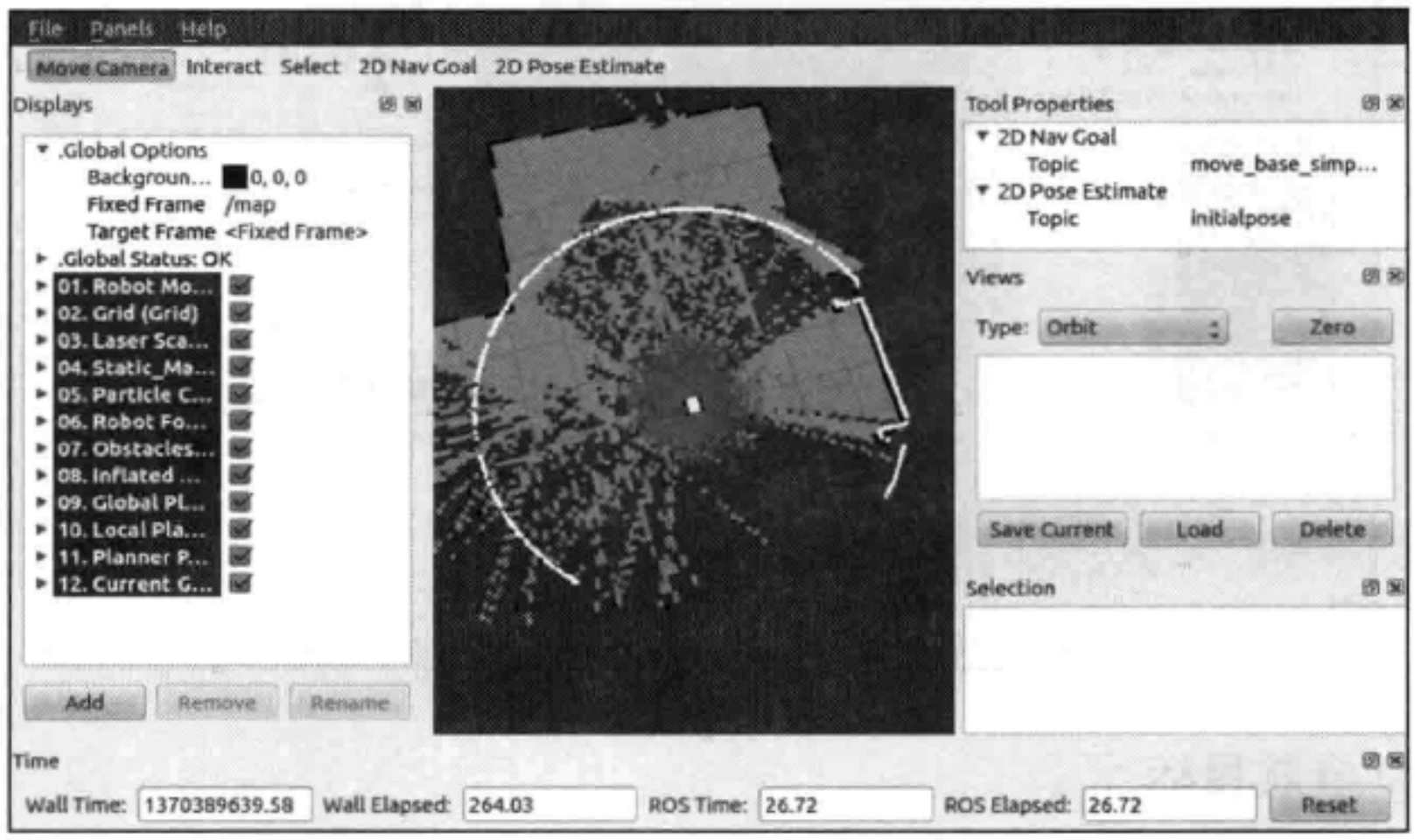
请注意，在这个文件中我们会启动之前创建的所有文件。我们还会启动一个带有第 7 章中创建的地图的地图服务器和 amcl 节点。

我们将会使用的这个 amcl 节点是专用于支持差分驱动机器人平台的。如果你希望用它驱动一个全向移动机器人，那么你就需要使用 amcl_omni.launch 文件。如果你想使用其他地图，那么请参看第 7 章并创建一个地图。

现在启动文件，并在一个新的命令行窗口中输入以下命令。请记住在启动这个文件之前，你需要先启动 chapter8_configuration_gazebo.launch 文件。

```
$ roslaunch chapter8_tutorials move_base.launch
```

然后你会看到以下窗口：



如果你比较这时候的屏幕与之前你启动 chapter8_configuration_ gazebo.launch 文件时所看到的屏幕，就会发现所有的选项都变成蓝色的了，这说明你的导航功能包集已经正常启动了。

正如前面所讲，我们还会在后面的章节中介绍使用导航功能包集时通过窗口查看可视化数据所需要的配置选项。

8.6 为导航功能包集设置 rviz

通过窗口可视化查看导航功能包集运行时的各类可视化数据对于学习导航功能包集来说是非常重要的。在本节中，我们会展示如何在 rviz 中查看导航功能包集发布的可视化数据，并针对导航功能包集发布的每一个可视化数据进行介绍。

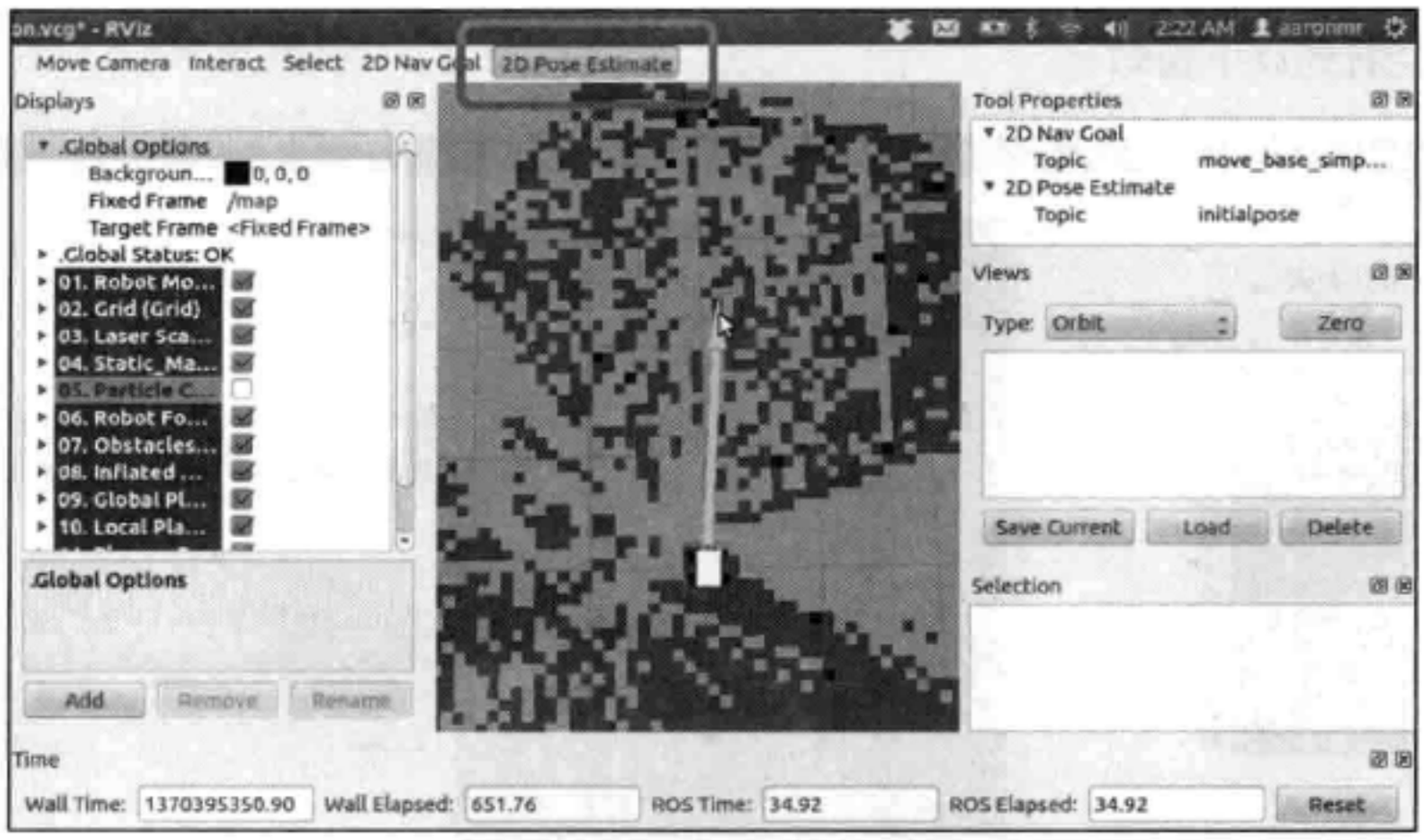
8.6.1 2D 位姿估计

2D 位姿估计（快捷键 P）允许用户初始化导航功能包集的定位系统，并设定机器人在实际环境中的位姿。导航功能包集会等待名为 initialpose 的新主题的初始化位姿消息。这个主题是通过 rviz 窗口发布的。我们要在窗口中事先修改主题的名称。

你能在下图中看到 initialpose 主题。点击 2D Pose Estimate 按钮，并点击地图来指定机器人的初始位姿。如果你不在一开始就进行这项工作，机器人会启动一个自动定位进程来初始化初始位姿。

- Topic: initialpose

- Type: geometry_msgs/PoseWithCovarianceStamped

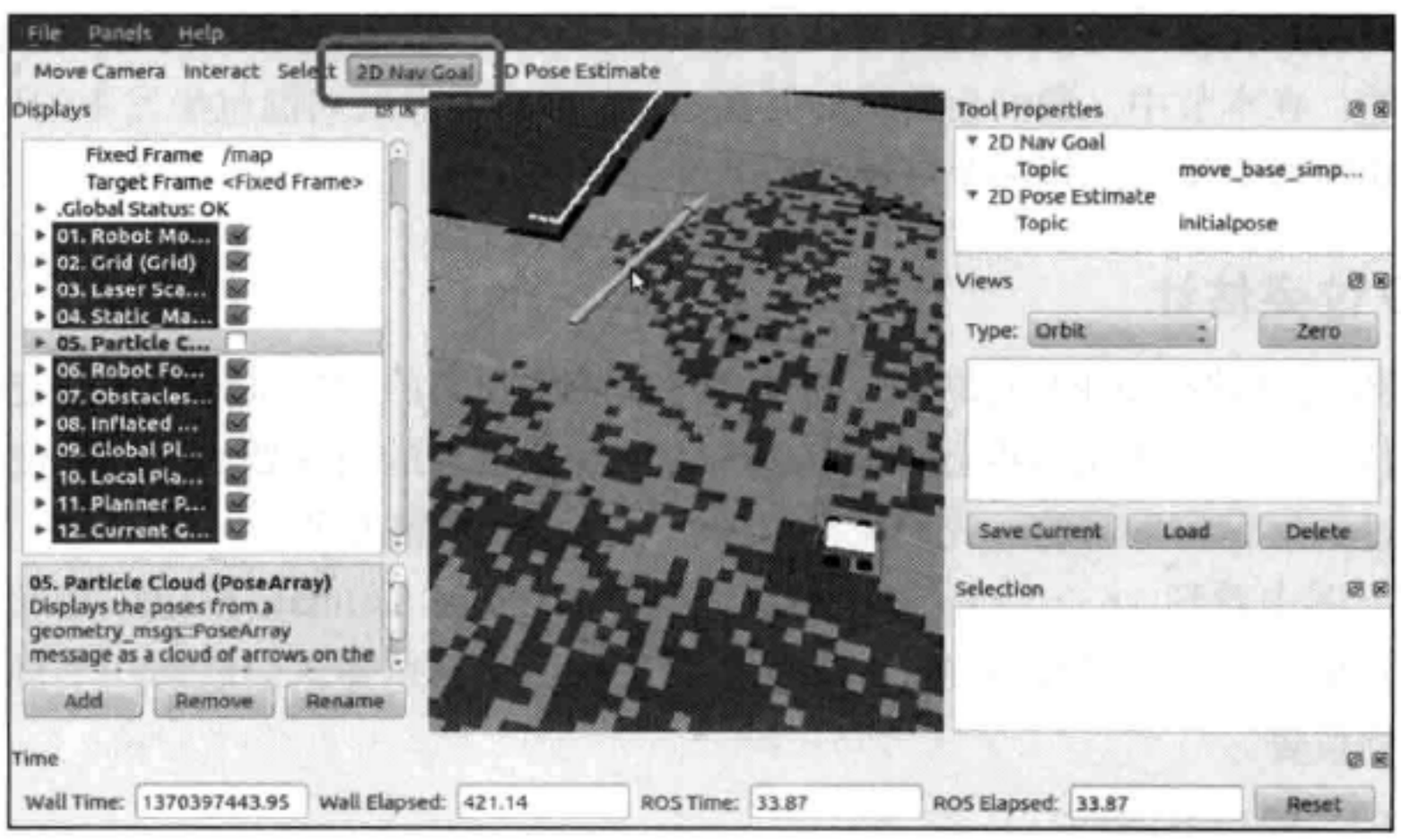


8.6.2 2D 导航目标

2D 导航目标（快捷键 G）允许用户向导航功能包集设定一个期望位姿作为机器人的导航目标。导航功能包集会等待名为 /move_base_simple/goal 的新主题的初始化目标消息。因此，你必须在 Tool Properties 子窗口的 2D Nav Goal 选项中修改主题的名称，在主题文本框中输入新的名称 /move_base_simple/goal。

在下图中，你会看到具体配置方法。点击 2D Nav Goal 按钮并在地图上选择机器人的运动目标。你能够选择 x 轴和 y 轴坐标，以及机器人最后的方向。

- Topic: move_base_simple/goal
- Type: geometry_msgs/PoseStamped

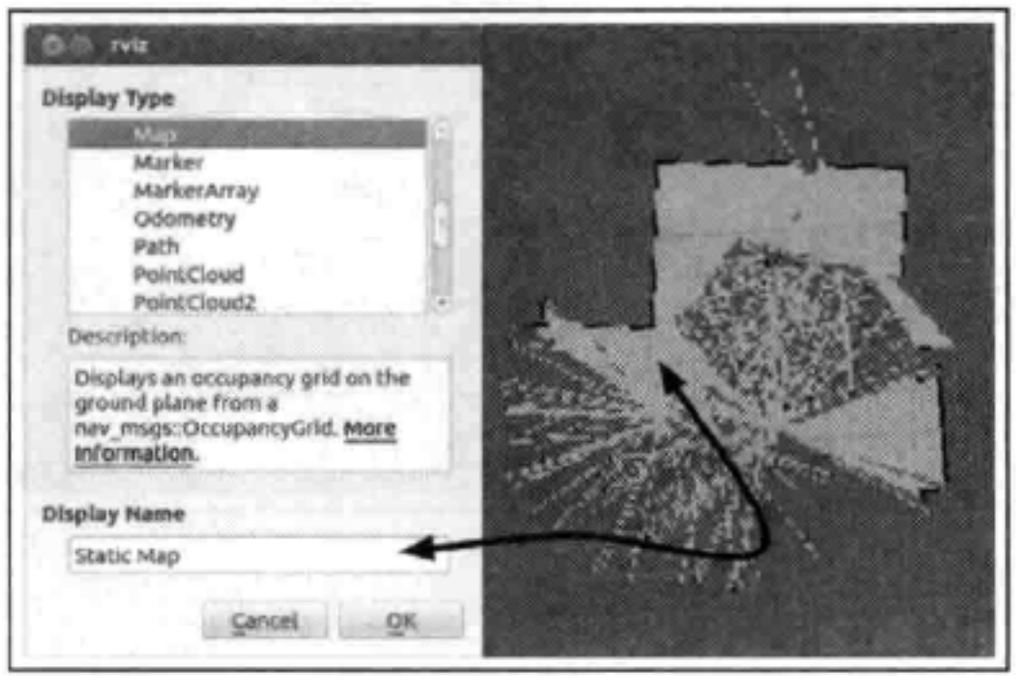


8.6.3 静态地图

下图显示了地图服务器所维护的静态地图（你需要在前面的配置文件中选择使用地图服务器和静态地图）。当你在 rviz 中查看地图的时候，你会看到在 7.6 节所创建的地图。

在下面的窗口中，你能够看到要选择的显示类型，并填写显示名称。

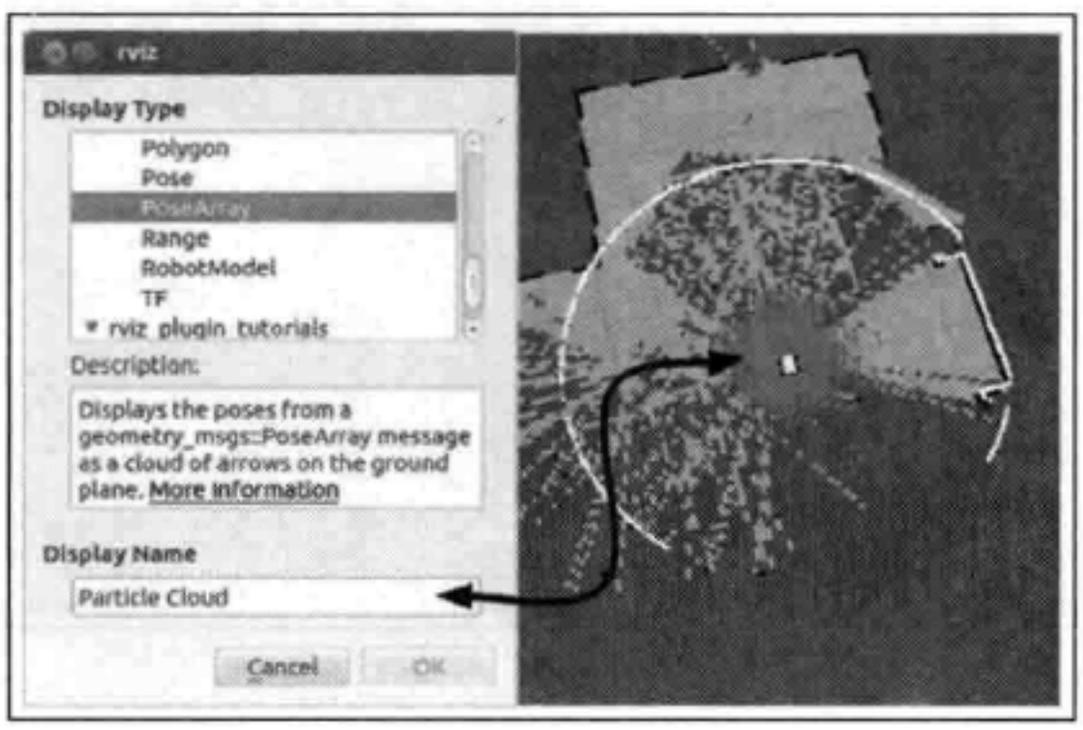
- Topic: map
- Type: nav_msgs/GetMap



8.6.4 点云

下图显示了用于机器人定位系统的点云。点云的分布表示了在定位系统中机器人位姿的不确定性。分散的点云代表了较高的不确定性；聚集的点云代表了较低的不确定性。在我们的示例中，机器人周围低不确定性的点太少，因此我们应该等待机器人更新点云。

- Topic: particlecloud
- Type: geometry_msgs/PoseArray

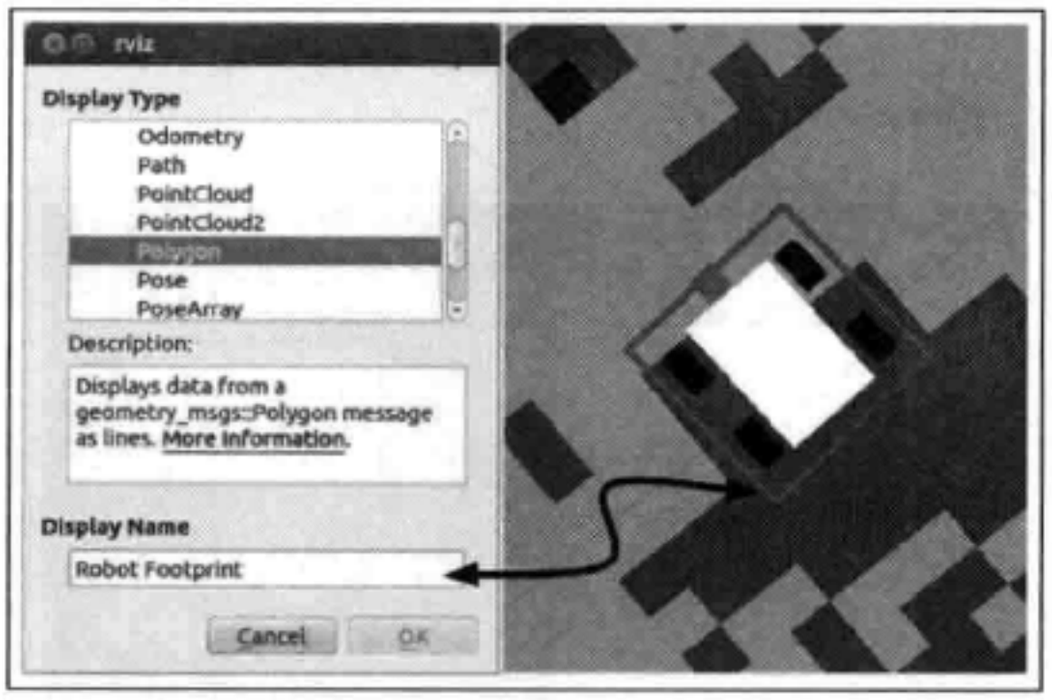


8.6.5 机器人立足点

下图显示了机器人的立足点。在我们的示例中，机器人是有立足点的。它的立足点有

0.4m 宽和 0.4m 高。这个参数可以在 costmap_common_params 文件中进行配置。这个尺寸对于导航功能包集来说很重要，因为只有配置正确才能保证机器人的运动安全。

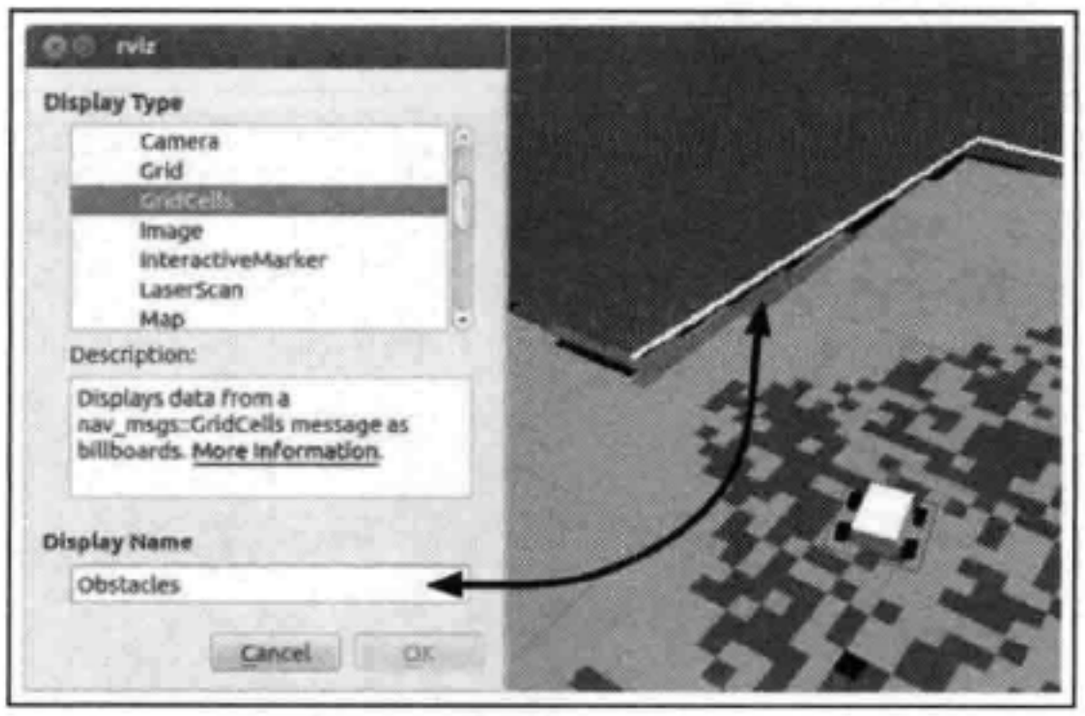
- Topic: local_costmap/robot_footprint
- Type: geometry_msgs/Polygon



8.6.6 障碍

下图显示了在导航功能包集的代价地图中看到的障碍物。在地图中，黑线的边上有几条红色的线，这些红线就是检测到的障碍物。如果某个障碍物暂时性地出现在你机器人前面（例如移动中的人或机器人），你也能在地图的中间看到红线。想要让机器人避免碰撞，那么就需要避免让机器人的立足点和带有障碍物的方格重合。

- Topic: local_costmap/obstacles
- Type: nav_msgs/GridCells

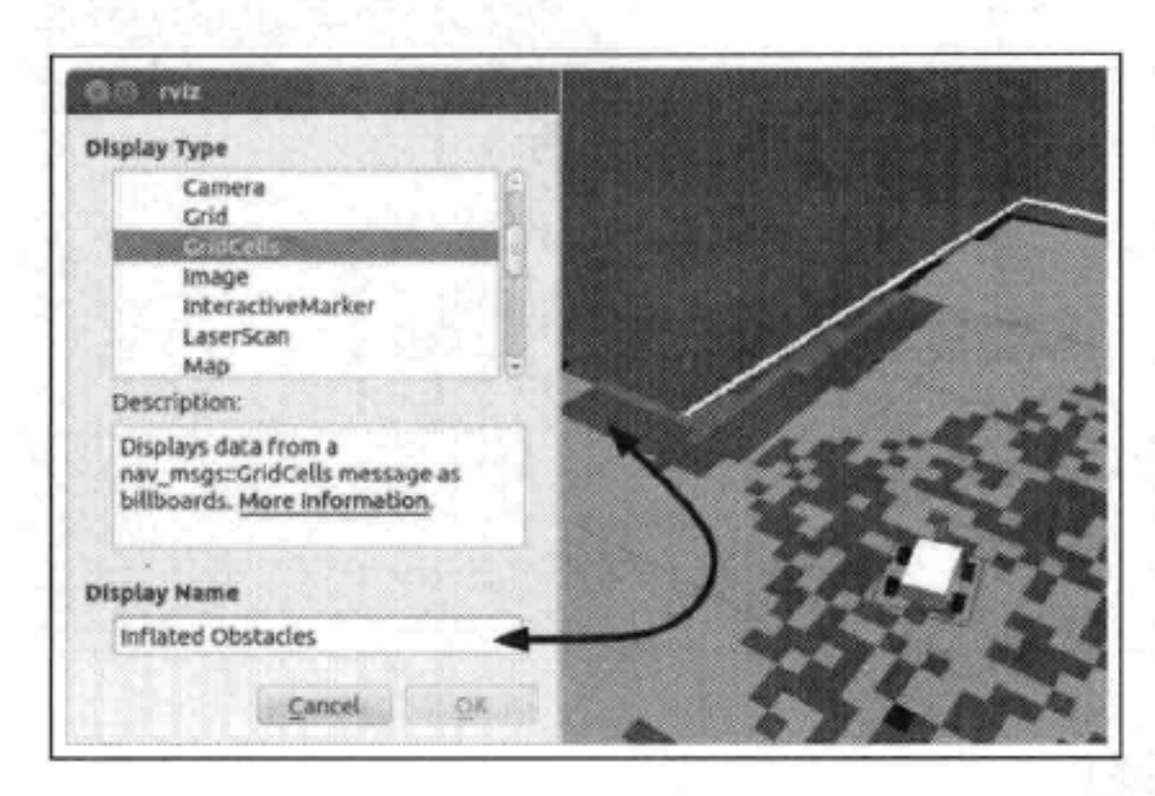


8.6.7 膨胀障碍

下图显示了导航功能包集的代价地图中按照机器人的内切圆半径对障碍物进行了膨胀处理。和红线表示的障碍类似，你能够看到地图中用蓝色的线条或形状表示的在机器人前面暂

时出现的障碍物。想要让机器人避免碰撞，那么就需要避免让机器人的立足点和带有膨胀处理后的障碍物的方格相重合。

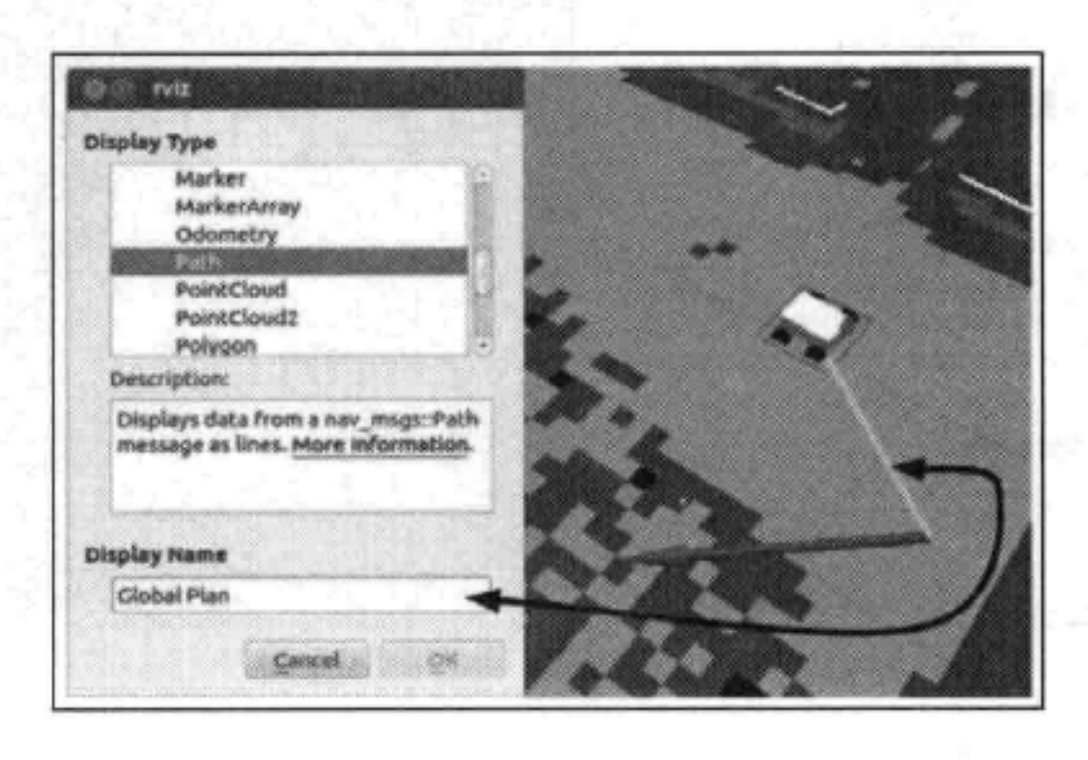
- Topic: local_costmap/inflated_obstacles
- Type: nav_msgs/GridCells



8.6.8 全局规划

下图显示了当前局部规划器处理的全局路径规划中的部分路径。你能看到其显示为绿色的线条。也许机器人在运动过程中还会发现障碍物，导航功能包集为了避免碰撞就会在尽量保证全局规划的基础上重新计算路径。

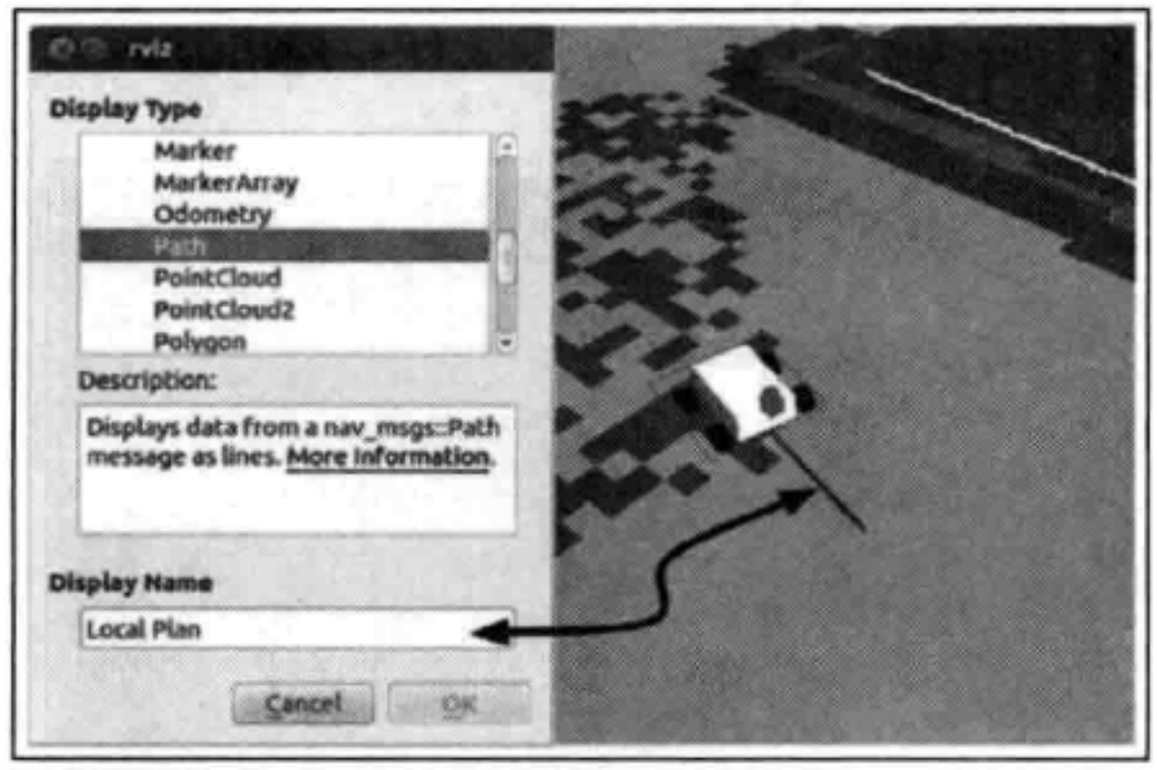
- Topic: TrajectoryPlannerROS/global_plan
- Type: nav_msgs/Path



8.6.9 局部规划

下图显示了机器人执行着由局部规划器生成的速度命令以及将会形成的运动轨迹。你能看到其显示为蓝色的线条。你通过这个状态显示能够了解到机器人是否在运动，并根据蓝色线条的长度对机器人的运动速度进行估计。

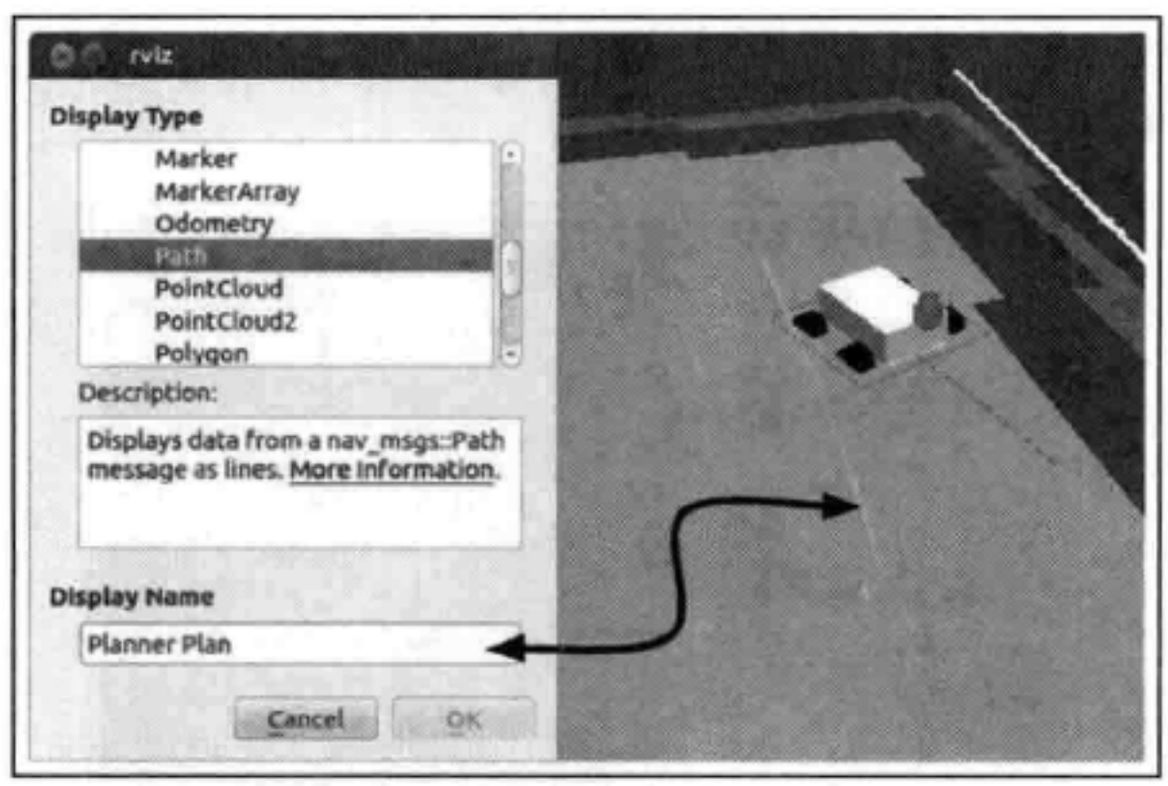
- Topic: TrajectoryPlannerROS/local_plan
- Type: nav_msgs/Path



8.6.10 规划器规划

下图显示了由全局规划器计算的完整规划。你能看到这些线条和全局规划中的线条很类似。

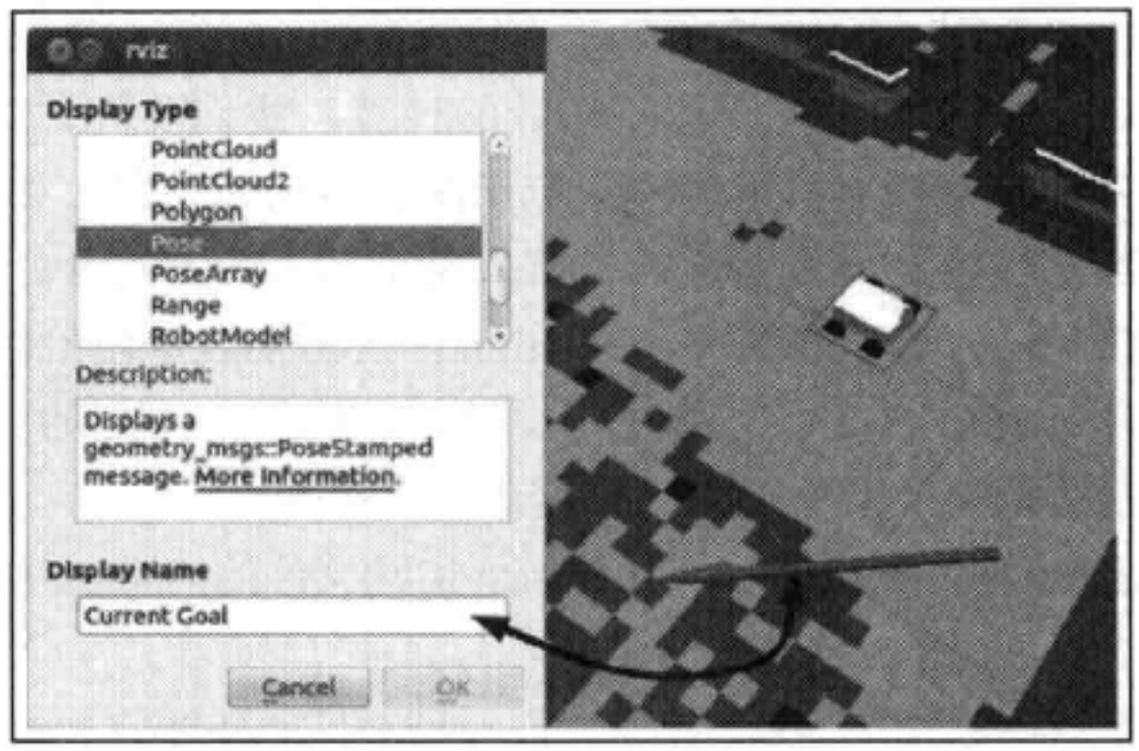
- Topic: NavfnROS/plan
- Type: nav_msgs/Path



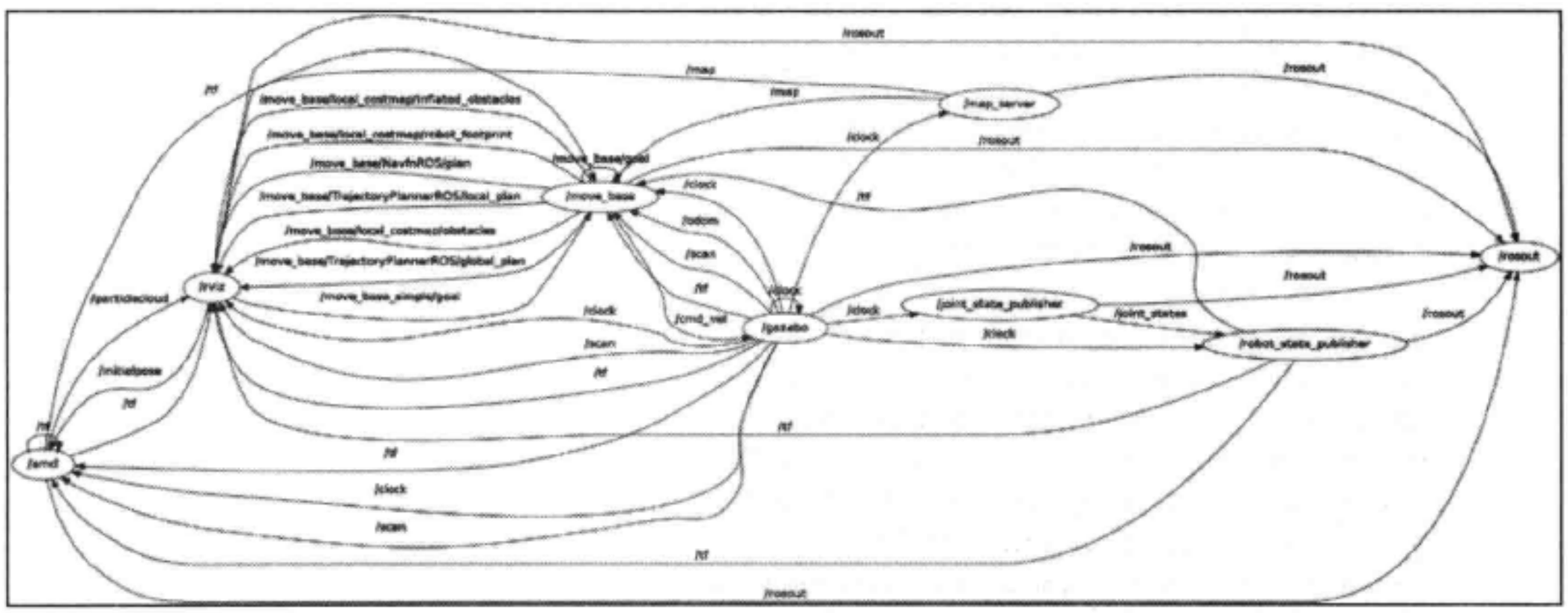
8.6.11 当前目标

下图显示了导航功能包集正在执行中的导航位姿目标。它是一个红色的箭头，你必须重新配置一个新的 2D 导航目标它才会出现。它用于用户了解机器人最终的导航目的地。

- Topic: current_goal
- Type: geometry_msgs/PoseStamped



这些可视化数据是你使用导航功能包集时经常需要在 rviz 中查看的。通过查看这些数据，你就会知道机器人是否正确执行了你的指令。现在我们来查看整个系统的总览图。我们通过运行 `rxgraph` 命令来查看是否所有节点都在正确运行以及节点之间的相互关系。



8.7 自适应蒙特卡罗定位

在本章中，我们将会使用自适应蒙特卡罗定位（AMCL）算法完成机器人定位。自适应蒙特卡罗定位是一种用于在 2D 环境下移动机器人的概率统计定位方法。这种算法在 ROS 系统中的具体实现是通过在已知地图的基础上使用粒子滤波算法跟踪机器人的位姿。

ROS 中的自适应蒙特卡罗定位算法在系统中有很多个配置选项。这些选项对定位算法的性能影响较大。如果你想了解该算法的具体信息，需要查看 AMCL 的官方文件。在以下链接中，你会找到关于算法的更详细介绍：

- http://www.ros.org/wiki/amcl
- http://www.probabilistic-robotics.org/

自适应蒙特卡罗定位节点主要使用激光扫描和激光雷达地图。当然它可以通过修改代码

以适应其他类型的传感器数据，例如声呐或双目视觉系统。对于本章，我们仅使用激光扫描和雷达生成的地图，通过消息传递完成位姿估计的计算。我们要先针对 ROS 系统提供的各个初始化参数，完成自适应蒙特卡罗定位算法粒子滤波器的初始化。如果你没有设定初始化位姿，自适应蒙特卡罗定位算法会假定你的机器人从坐标系的原点开始运行，这样计算会相对复杂。所以建议你在 rviz 中通过 2D Pose Estimate 按钮来设定初始化位姿。

然后我们需要引用 amcl_diff.launch 文件，并调用一系列的配置参数来启动节点。这些配置一般都使用默认配置或最小设置，来驱动算法运行。

下面，我们将要逐项学习 amcl_diff.launch 启动文件的内容，并解释其中的各个参数：

```
<launch>
<node pkg="amcl" type="amcl" name="amcl" output="screen">
  <!-- Publish scans from best pose at a max of 10 Hz -->
  <param name="odom_model_type" value="diff"/>
  <param name="odom_alpha5" value="0.1"/>
  <param name="transform_tolerance" value="0.2" />
  <param name="gui_publish_rate" value="10.0"/>
  <param name="laser_max_beams" value="30"/>
  <param name="min_particles" value="500"/>
  <param name="max_particles" value="5000"/>
  <param name="kld_err" value="0.05"/>
  <param name="kld_z" value="0.99"/>
  <param name="odom_alpha1" value="0.2"/>
  <param name="odom_alpha2" value="0.2"/>
  <!-- translation std dev, m -->
  <param name="odom_alpha3" value="0.8"/>
  <param name="odom_alpha4" value="0.2"/>
  <param name="laser_z_hit" value="0.5"/>
  <param name="laser_z_short" value="0.05"/>
  <param name="laser_z_max" value="0.05"/>
  <param name="laser_z_rand" value="0.5"/>
  <param name="laser_sigma_hit" value="0.2"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_model_type" value="likelihood_field"/>
  <!-- <param name="laser_model_type" value="beam"/> -->
  <param name="laser_likelihood_max_dist" value="2.0"/>
  <param name="update_min_d" value="0.2"/>
  <param name="update_min_a" value="0.5"/>
  <param name="odom_frame_id" value="odom"/>
  <param name="resample_interval" value="1"/>
  <param name="transform_tolerance" value="0.1"/>
  <param name="recovery_alpha_slow" value="0.0"/>
  <param name="recovery_alpha_fast" value="0.0"/>
</node>
</launch>
```

其中，min_particles 和 max_particles 参数设定了算法运行所允许的粒子的最小和最大数量。如果你使用更多的粒子，那么计算结果会更加精确，当然这也会消耗更多的

CPU 资源。

参数 laser_model_type 用于配置激光雷达的类型。在我们的示例中，将参数设置为 likelihood_field。在算法中，也可以设置为 beam 光束雷达。

参数 `laser_likelihood_max_dist` 用于设置地图中障碍物膨胀的最大距离。使用这个参数的前提条件是前面选择了 `likelihood_field` 模式。

参数 `initial_pose_x`、`initial_pose_y` 和 `initial_pose_a` 并不包含在启动文件中。如果使用它们，就可以设定在算法启动时机器人的初始位置。例如，你的机器人每次都从某个固定的充电桩启动，那么你可能就会在启动文件中去设定这些参数了（而无需在 rviz 中设定了）。

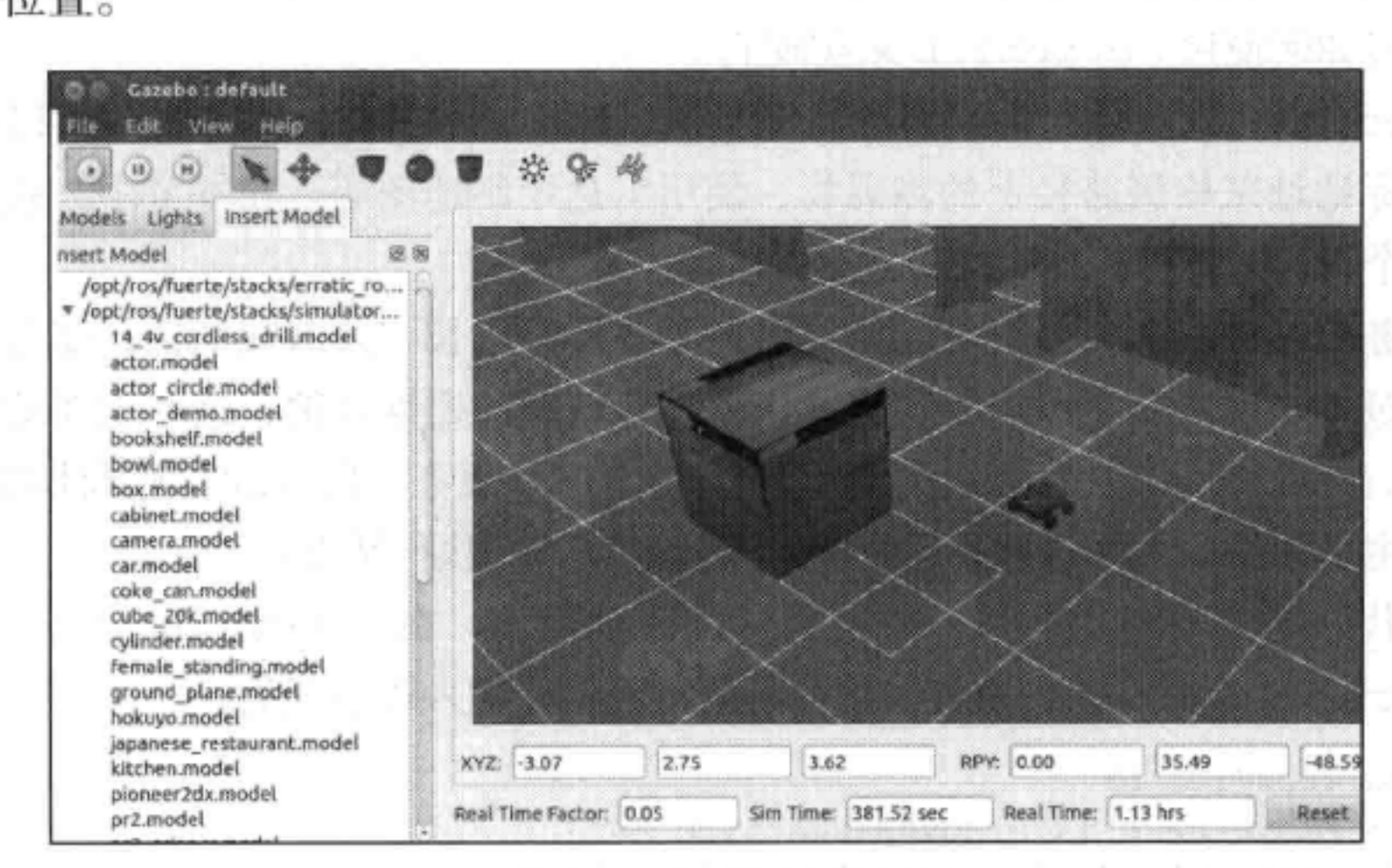
也许你应该试着改变启动文件中的一些参数，使你的机器人定位算法获得最佳的计算结果。在 ROS 官方网站上，你能够找到很多关于这些参数具体配置的信息。

8.8 避免障碍

导航功能包集最重要的一个功能就是在机器人运行时遇到障碍物，能够对原定路径进行重新规划和计算。在 Gazebo 中，只要你在机器人前面放一个物体，你就会发现机器人的运动发生改变。例如，在我们的仿真环境中添加一个大的箱子放到机器人路径的中央。导航功能包集会检测到新的障碍物并自动创建新的路径。

在下图中，你能够看到我们添加了一个物体。Gazebo 能够提供一些预定义的 3D 物体。你能够在仿真和测试移动机器人时在仿真环境中直接调用。

想要查看这些物体的清单，你可以点击 Insert model。选择一个物体并点击鼠标选择你要放置物体的位置。



如果你在 `rviz` 窗口中，就会看到一个新的能够避开障碍物的全局规划。如果你有一个真的机器人，并且在有很多人走来走去的环境中应用导航功能包集，那么你会发现这个功能

非常有趣。如果机器人探测到可能的碰撞，那么它会改变运动方向，并尝试向目标移动。你能够在下图中清晰地看到这个功能的图形化表示。记住对这些障碍物的检测会减小局部规划器代价地图的覆盖范围（例如，机器人周围 4m × 4m 的一个区域）。

8.9 发送目标

我相信你肯定会试着在地图上移动机器人，看看它到底能够做什么。但是每次都是差不多的结果，可能慢慢地你就会觉得有点无聊了。

也许你可能会想，如果它能够通过编程设定一系列的运动指令，一次就能让机器人去很多个地方。这样即使我们不在电脑前操作 rviz，它也会根据预先的指令去运动。好吧，现在你就会学习如何使用 actionlib 来实现它。

这个 actionlib 功能包为这样的功能提供了标准化的接口。例如，你可以使用它向机器人发送目标地址来检测路径上的障碍物、使用雷达绘制地图等。在本节中，我们将会为机器人设定一个目标并等待机器人完成这项工作。

这和服务看起来很相似，但是如果你的任务要持续很长一段时间的话，那么你可能想要在命令执行过程中取消命令或者获得当前指令执行情况的定期反馈，这些功能就无法通过服务来实现了。更进一步说，actionlib 通过消息（不是服务）和主题实现具体功能。这样我们可以通过主题来发送目标，并在需要的时候获取反馈或执行结果。

下面的代码就是向移动机器人发送目标的一个示例。在 chapter8_tutorials/src 文件夹下以 sendGoals.cpp 为名创建一个新文件，并加入以下代码：

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_broadcaster.h>
#include <sstream>
```

```
typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;

int main(int argc, char** argv){
  ros::init(argc, argv, "navigation_goals");

  MoveBaseClient ac("move_base", true);

  while(!ac.waitForServer(ros::Duration(5.0))){
    ROS_INFO("Waiting for the move_base action server");
  }

  move_base_msgs::MoveBaseGoal goal;

  goal.target_pose.header.frame_id = "map";
  goal.target_pose.header.stamp = ros::Time::now();

  goal.target_pose.pose.position.x = 1.0;
  goal.target_pose.pose.position.y = 1.0;
  goal.target_pose.pose.orientation.w = 1.0;

  ROS_INFO("Sending goal");
  ac.sendGoal(goal);
  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    ROS_INFO("You have arrived to the goal position");
  else{
    ROS_INFO("The base failed for some reason");
 }
  return 0;
}
```

在编译之前，在 manifest 文件中添加以下依赖项：

```
<depend package="move_base_msgs" />
<depend package="actionlib" />
<depend package="geometry_msgs" />
<depend package="tf" />
```

在为程序编译生产可执行文件之前，在你的 CMakeList.txt 文件中添加以下文件：

```
rosbuild_add_executable(sendGoals src/sendGoals.cpp)
```

现在，使用以下命令编译功能包：

```
$ rosmake chapter8_tutorials
```

下一步就是使用启动文件启动仿真环境并测试新程序。使用下面的指令来启动所有节点和配置：

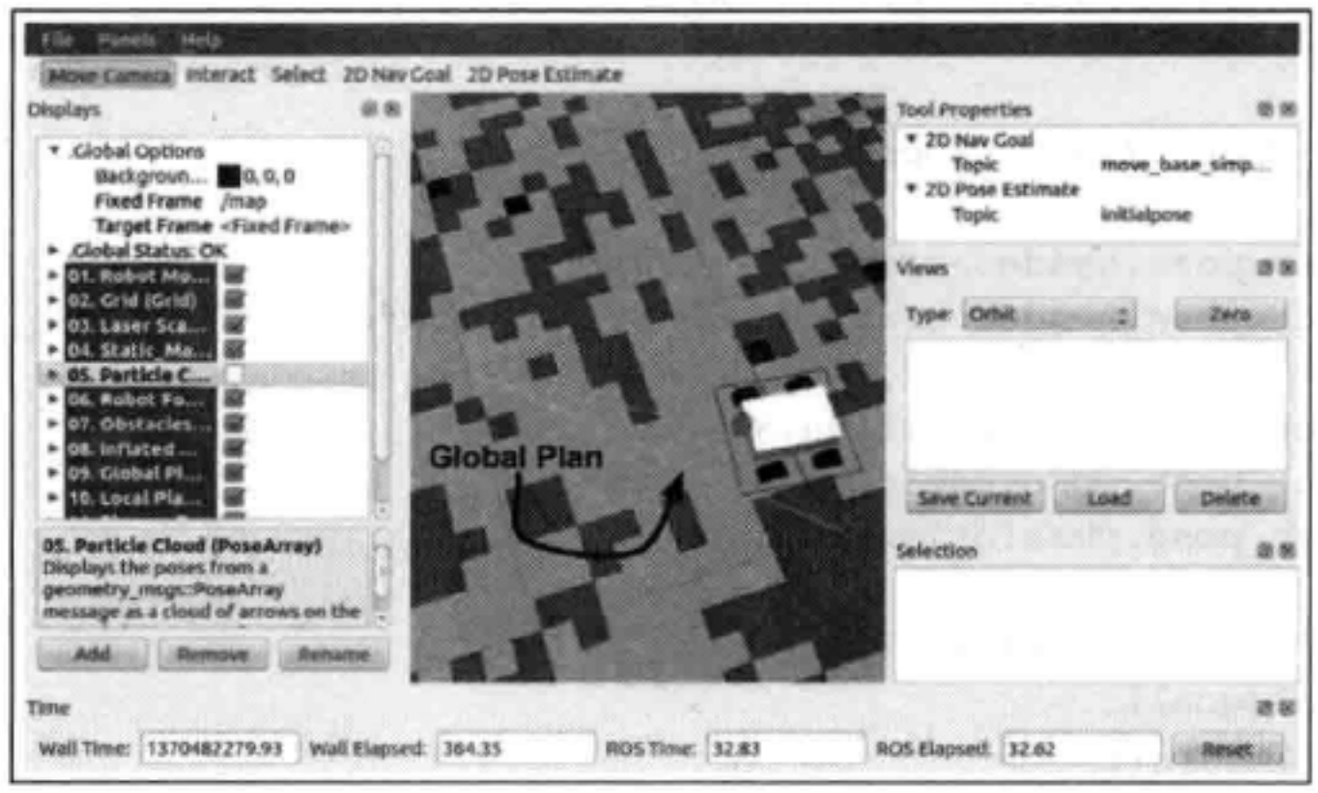
```
$ roslaunch chapter8_tutorials chapter8_configuration_gazebo.launch
```

```
$ roslaunch chapter8_tutorials move_base.launch
```

一旦你配置好 2D 位姿估计，使用以下命令在一个新的命令行窗口中启动 sendGoal 节点：

```
$ rosrun chapter8_tutorials sendGoal
```

如果你查看 rviz 屏幕，你会在地图中看到一个新的全局规划（绿线）。这意味着导航功能包集已经接受了新的目标并开始执行这个目标。



当机器人到达目标时，你会在运行节点的命令行窗口中看到下面的信息：

```
[ INFO ] [...,...]: You have arrived to the goal position
```

你可以设置一系列的路径点，或者帮助你的机器人在复杂地图或复杂任务中规划一条大致的线路。这样你就能够编写任务程序、引导机器人运送物品或者查看其他房间的情况。

8.10 本章小结

完成本章的学习之后，你应该已经可以通过导航功能包集驱动你的机器人了。无论你使用的是真正的机器人还是仿真机器人，无论是真实的环境还是仿真环境，只要按照本章中的步骤，不需要太复杂的配置，你的机器人就可以在地图中自主移动了。你可以根据 ROS 代码复用的理论编写控制方案和机器人定位算法，将其他机器人的算法应用到你的机器人上。本章中的难点在于理解各个配置参数并学会如何使用和配置这些参数，以使你的机器人和算法的运算效果达到最佳。判断这些参数的基本要求是看你的机器人能否正常工作。如果连正常工作都无法达到，你就需要检查和调整一下参数，看看机器人的反应如何。

在下一章中，你会看到一些你能够在 ROS 中下载和仿真的机器人。这些机器人有 3D 模型并且能够在 Gazebo 中进行仿真。它们中有一些能够应用导航功能包集，你可以学着去使用它们。对于你来说应该没有什么问题，因为你已经有对类似功能编程、启动系统、定位错误并修改代码的知识了。

第 9 章
在实践中学习

我们将通过介绍如何使用真正的机器人来结束本书。这些机器人被全世界的研究机构和机器人公司所广泛使用。其中一些组织和公司已经将自己的机器人软件发布到 ROS 框架中，并且制作和发布了很多与其机器人相匹配的 ROS 功能包。正如我们将在本章所见的，这些功能包通常包含了许多功能包集，包括导航、机器人专用驱动程序、远程操作和用于 Gazebo 仿真的机器人 URDF 模型。

这里，我们会向你展示如何安装这些最著名机器人所对应的功能包，并用它们进行仿真。请注意这里的大部分机器人非常非常昂贵，但是你也能开发它们的代码，并把你的算法提交给它们的生产商。当然，也有一些机器人是业余爱好者也能够负担得起的，例如基于 roomba 的 turtlebot。无论如何，你都可以使用一台电脑和带有 Gazebo 和导航功能包集的 ROS 系统运行并使用这里列出的所有机器人。

对于本章所讲的每一种机器人，我们都会对机器人特性进行一个简要的描述，介绍所属开发公司、资源链接及相应的 ROS 功能包。然后我们会展示安装步骤，还可能对特定的 ROS 发行版进行一些建议或说明。在系统安装完毕之后，我们会运行一些示例来让机器人动起来，并做些激动人心的事情。对于每一个机器人，“hello world”示例都会是在 Gazebo 中加载 URDF 模型。然后我们会展示如何移动它，在仿真环境中进行导航，并对传感器和执行机构进行仿真，特别是手臂和末端执行器。

本章的主要内容包括以下机器人和示例：

- PAL 机器人公司的 REEM 机器人。主要介绍 REEM-H 型，这是一个差分轮式驱动的人形机器人平台。我们还会大致介绍这家公司的其他机器人，如双足型的 REEM-C。然后展示如何安装这个机器人的公共代码，并在仿真环境中对 REEM-H 进行导航控制。
- 柳树车库公司的 PR2 机器人。正是这家公司创立了 ROS 系统，并为其提供维护和开发服务。我们将会学习如何从官方软件源下载和安装这个机器人，并使用 Gazebo 完成机器人导航和选择不同部件。我们还会利用前两章所学的知识创建一个地图。最后，会给大家列举一些可用的 demo 程序。
- Robonaut 2，简称 R2，是 NASA 开发的敏捷型人形机器人。我们将会演示如何在仿

真环境下安装和加载机器人，通过配置不同的选项进行机器人仿真，如选择固定机器人、为机器人安装腿、在国际空间站环境中仿真等。

- Husky 漫游者是由 Clearpath 机器人公司设计的四轮户外型巡逻机器人。我们会介绍如何在仿真环境中安装和运行。
- 最后会介绍 TurtleBot，这是一个由 Yujin 机器人公司和 iRobot 公司联合研发的低成本移动机器人。它也是我们所介绍的机器人中唯一能够让爱好者负担得起的轮式机器人。我们会从官方软件源中下载并安装这个机器人，然后在 Gazebo 中运行它。

虽然从本书一开始到现在，我们都在使用 ROS Fuerte，但是在将要介绍的这些机器人中，有些我们会建议你使用 ROS Groovy。其实，我们已经在某些特定功能上使用过 ROS Groovy 了，当然这主要是因为这些功能最近才加入到最新的版本中，或者是这些功能能在新版中得到更好得支持。在这里也是一样的。对于某些机器人，对 Groovy 支持得更好。当然如果你非要在 Fuerte 中运行，那也不是不可以。在第 1 章中，我们就已经介绍过如何安装 ROS 了，Groovy 和 Fuerte 的安装方法也没有什么区别，你只需要在 Ubuntu 中的 Debian 软件包中选择 `ros-groovy-*` 功能包即可。还有，你还需要知道如何通过 `.bashrc` 文件调用你所使用的 Ubuntu 系统发行版的 setup.bash 文件，这样你就能在你的机器中同时安装这两种版本了。

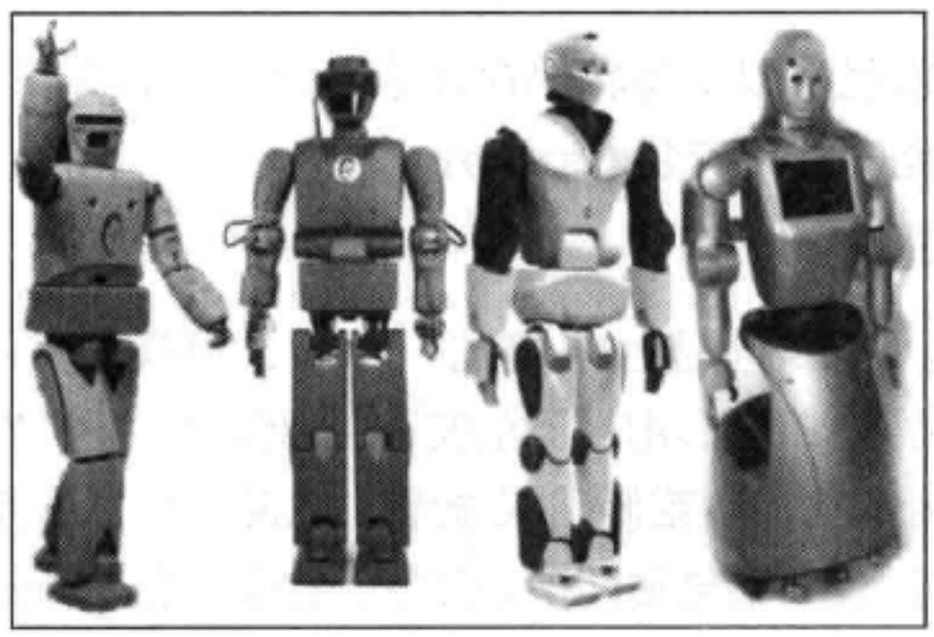
9.1 REEM——类人形 PAL 机器人

REEM 是一种 1.65m 高的人形服务机器人。它由 PAL 机器人公司设计制造，而且它只是该公司研发的众多人形机器人中的一款。简而言之它还被发展成了两种类人形机器人：双足型机器人和差分轮式驱动机器人。REEM 是最新款的使用差分轮式驱动的机器人，而且具有非常时髦的外观。它带有一套自动导航系统，胸前是一个非常易于操作的显示界面来完成**人机交互**（Human Robot Interaction，HRI），还集成了语音和人脸识别系统。

REEM 主要用于在酒店、博物馆、购物广场、机场和酒店等公共场所为人们提供娱乐和救助等服务。它还能运送小的行李；作为动态信息节点，为人们提供多种多样的多媒体应用，例如显示周围建筑的地图，查询各种信息（天气、周边餐馆、航班信息等）；还能通过网络视频系统为人们提供远程帮助。可以通过以下链接访问他们的网站：http://pal-robotics.com/robots。

PAL 机器人公司（http://pal-robotics.com）是阿布扎比皇家集团旗下的一家公司。这家公司以为每一个人设计理想的机器人为最终研发目标。他们的第一台机器人是 2005 年设计完成的双足机器人 REEM-A。它参加了 2006 年的 RoboCup 世界机器人足球锦标赛并赢得了行走组冠军。REEM-B 是这家公司研发的第二种人形机器人，近期他们又推出了 REEM-C。下图从左到右分别展示了以上这三种机器人。他们还设计了两种差分驱动人形机器人（使用轮式移动底盘），其中 REEM-H1 是它们中的第一种。

如下图最右侧所示，REEM-H1 是前面图片中展示的最新型 REEM 机器人的上一代。



可以通过 PAL 机器人公司的主页来查看他们的机器人和产品、公司的新闻和公开资料。如在他们的博客（http://pal-robotics.com/software）上就会发布一些 demo 程序或是会议资料。当然，他们提供的软件是一些 ROS 系统使用的开源功能包。从本节的软件介绍部分开始，我们都会使用 GitHub 软件源来安装机器人程序。GitHub 是一个存储了大量 git 软件源的资源网站，这样用户和开发者都能够方便地更新和下载开源程序。下面我们就会介绍如何安装 REEM 的软件，以及 PAL 官方在软件源中存放的各个功能包集。

9.1.1 从官方软件源安装 REEM

如果访问链接 https://github.com/pal-robotics，你会看到在 ROS 中集成为 REEM 机器人的整个软件包是由一系列功能包集所组成的。想要轻松快捷地安装这些功能包集，最好还是使用 `pal-ros-pkg` 软件源中的 `rosinstall` 文件。但在正式安装这些功能包集之前，我们还需要先安装一些安装和使用功能包集时所需的依赖项。第一步，我们先要安装一个独立版本的 Gazebo 仿真环境。ROS 自带的版本与 REEM 所使用的版本相比比较老。想要安装独立版本的 Gazebo，首先需要在 Ubuntu 系统的软件源中添加源路径。这些代码是由 OSRF（开源机器人基金会）进行维护的，我们需要按如下所示设置 `packages.osrfoundation.org` 路径（请注意需要选择你所使用的 Ubuntu 系统的版本，在本示例中是 `quantal`）：

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu
quantal main" > /etc/apt/sources.list.d/gazebo-latest.list'
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add
-
```

使用以下命令更新系统安装包清单：

```
sudo apt-get update
```

使用以下命令安装 Gazebo：

```
sudo apt-get install gazebo
```

最后，在 .bahrc 文件中使用 source 命令添加 Gazebo 的 setup.sh 文件路径：

```
echo "source /usr/share/gazebo/setup.sh" >> ~/.bashrc
source ~/.bashrc
```

为了方便读者，我们已经将以上步骤保存在本书附加代码文件中的 scripts 文件夹下。你只需要执行以下命令就能够直接安装 Gazebo：

```
./gazebo_install.sh
```

第二个依赖项实际是 drcsim 仿真环境的一部分，而我们并不需要完全使用这个仿真环境。我们只需要其中的 OSRF 通用库。在安装之前，你需要先移动到 ROS_PACKAGE_PATH 功能包路径下，使用版本控制系统从系统源下载源文件。常用的版本控制系统有 cvs、subversion 和 git（请注意如果要使用 hg 命令，你需要使用 mercurial）。

```
hg clone https://bitbucket.org/osrf/osrf-common
```

使用以下命令对其进行编译和安装：

```
cd osrf-common
mkdir build
cd build
cmake ..
make
sudo make install
```

为了方便读者，上面的这些命令我们同样集成在了以下命令行脚本中：

```
./osrf_common_install.sh
```

其余的依赖项可以通过 apt-get 命令轻松安装：

```
sudo apt-get install ros-groovy-pr2-simulator
sudo apt-get install libglew-dev libdevil-dev
```

请注意，我们在以上命令中使用了 ROS Groovy。使用 ROS Fuerte 应该也能正常运行。但是 REEM 的开发人员已经将开发代码转移到了 Groovy 软件源中，因此使用 Groovy 会更加安全。否则的话，在编译和使用过程中，可能会出现更多的错误或软件崩溃。

实际上，这个仿真环境库是来自于 PR2 的，但我们并不需要 PR2 的源代码，而只使用这个库。

这样，我们还需要在 .bashrc 文件中添加一些环境变量，我们可以在文件的末尾加入以下行：

```
# Gazebo - atlas_msgs; not required if drcsim is installed
export GAZEBO_PLUGIN_PATH="`rospack find atlas_msgs`/lib:$GAZEBO_PLUGIN_
PATH"
export LD_LIBRARY_PATH="`rospack find atlas_msgs`/lib:$LD_LIBRARY_PATH"

# Gazebo - PAL
export GAZEBO_PLUGIN_PATH="`rospack find pal_gazebo_plugins`/lib:$GAZEBO_
PLUGIN_PATH"
export LD_LIBRARY_PATH="`rospack find pal_gazebo_plugins`/lib:$LD_
LIBRARY_PATH"

# Gazebo - REEM-H3
export GAZEBO_MODEL_PATH="`rospack find reem_gazebo`/models:$GAZEBO_
MODEL_PATH"
export GAZEBO_RESOURCE_PATH="`rospack find reem_gazebo`/worlds:$GAZEBO_
RESOURCE_PATH"
```

为了方便读者，我们将所有环境变量都保存在environment/pal_ros_pkg_environment.sh文件中，你只需要执行以下命令：

```
cat scripts/environment/pal_ros_pkg_environment.sh >> ~/.bashrc
```

现在我们已经完成了REEM机器人软件所需的所有依赖项的安装和环境配置。移动到ROS_PACKAGE _PATH文件夹下，创建一个新的文件夹。在这个新文件夹中我们会存储REEM机器人所有的功能包集，所以我们命名为ros-pal-pkg文件夹。进入到这个目录中，并运行以下命令：

```
rosinstall . https://raw.github.com/pal-robotics/pal-ros-pkg/master/pal-
ros-pkg-gazebo-standalone.rosinstall
```

这会下载包含REEM功能包集的所有软件包。由于软件包数量较多，可能需要等待比较长的时间。在下载前，我们也可以不创建这个ros-pal-pkg文件夹，但是这样的话，就不好统计下载的软件包，也不知道到底下载了哪些功能包和功能包集。请注意，我们还应该使用rosintall文件安装一个独立版本的Gazebo，然而考虑到在本书之前的章节中已经安装过独立版本的Gazebo，就不再详细说明。你也可以下载其他的Gazebo版本，不过有些版本比较不稳定，使用时需要多加注意。

等所有软件下载完毕之后，我们需要在安装软件之前先更新环境。我们只需要使用以下命令：

```
rosstack profile && rospack profile
source ~/.bashrc
```

下面我们就来完成各个功能包集的安装。我们只需要运行以下批处理脚本：

```
stacks=(atlas_msgs pal_gazebo_plugins pal_image_processing perception_
blort reem_arm_navigation reem_common reem_controllers reem_kinematics
reem_msgs reem_robot reem_simulation)
```

```
for stack in ${stacks[@]}
do
  rosmake -i $stack
done
```

现在，我们就完成了 REEM 软件的安装。上面这些命令也可以在脚本文件中执行。你可以直接使用以下命令，虽然它不是用于整个系统的安装：

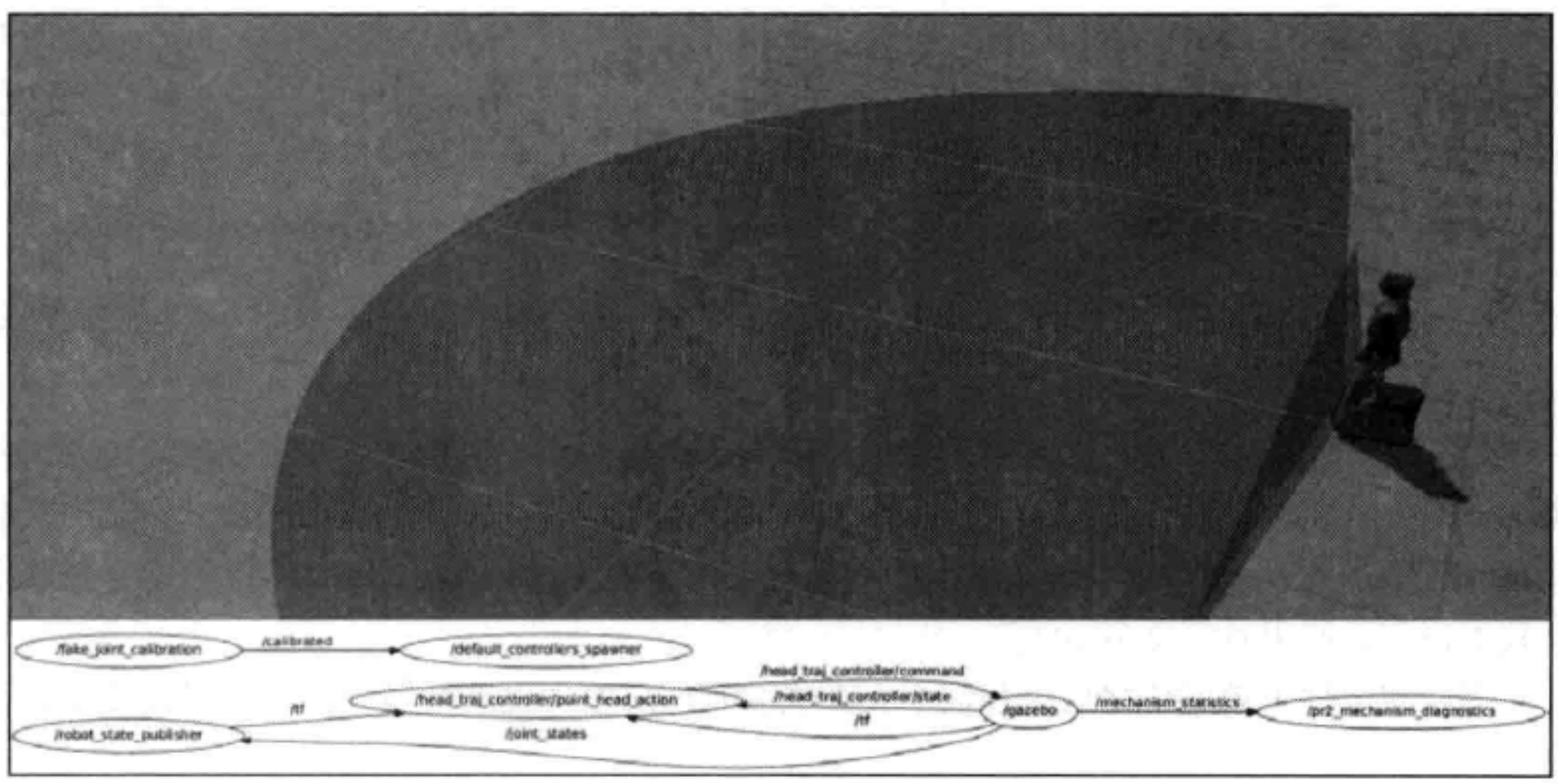
```
./pal_ros_pkg_install.sh
```

在安装的所有功能包集中，最重要的几个是 reem_simulation、reem_teleop 和 reem_arm_navigation，这些功能包集能够提供在 Gazebo 中进行机器人仿真和远程控制的功能。

9.1.2 使用 Gazebo 仿真环境运行 REEM

REEM 源代码中包含了在 Gazebo 中运行机器人仿真的启动文件。只需要执行以下代码就能在 Gazebo 环境中看到 REEM 机器人了。具体如下图所示，也可以运行 rxgraph 命令查看节点状态图：

```
roslaunch reem_gazebo reem_empty_world.launch
```



在上图中，你能够观察到 REEM 机器人有两个前向激光雷达。一个检测前方大范围目标，另一个检测近距离地面目标。我们可以在 Gazebo 中加载任何环境，并在仿真环境中使用导航功能包集完成机器人移动避障等工作。具体的方法请参看前面的章节。

如果你在 Gazebo 中运行 REEM 时发现了任何问题，一般都是和你所使用的 Gazebo 版本相关的。在这种情况下，你可以试着访问 http://gazebosim.org/assets/distributions/ 找一些预

编译完成的 Gazebo 版本，如 1.6.2 版本，虽然更新的版本也支持。

我们可以使用 TurtleBot 的启动文件远程控制 REEM 机器人，两者是完全兼容的。如果你还没有安装 Turtlebot，那么你只需要使用以下命令安装：

```
sudo apt-get install ros-groovy-turtlebot ros-groovy-turtlebot-apps
```

然后，运行以下命令：

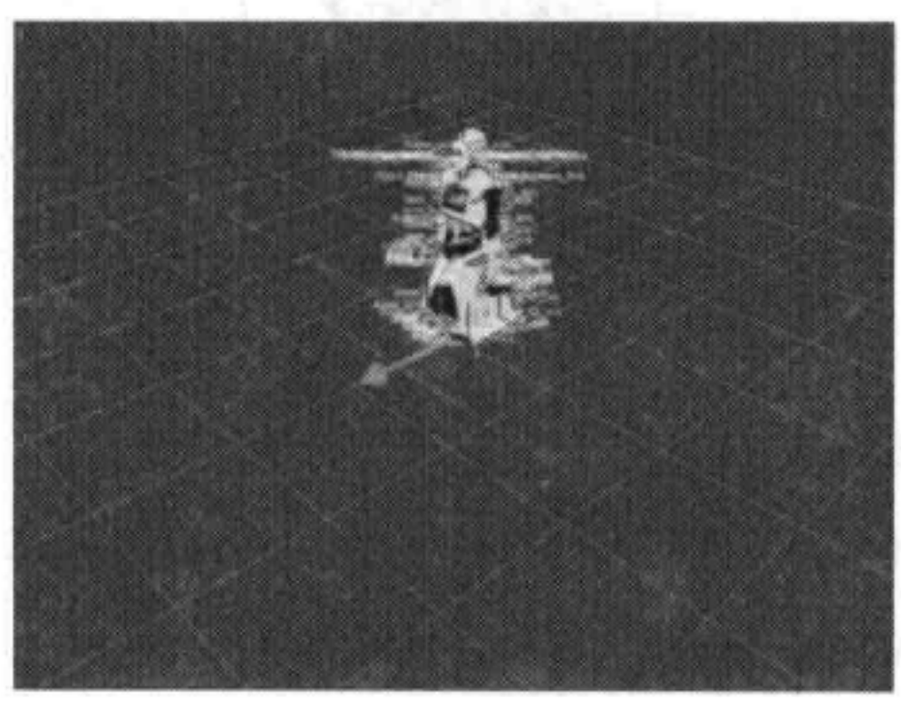
```
rosrun turtlebot_teleop turtlebot_teleop_key
```

使用以下命令向 REEM 机器人传递速度指令：

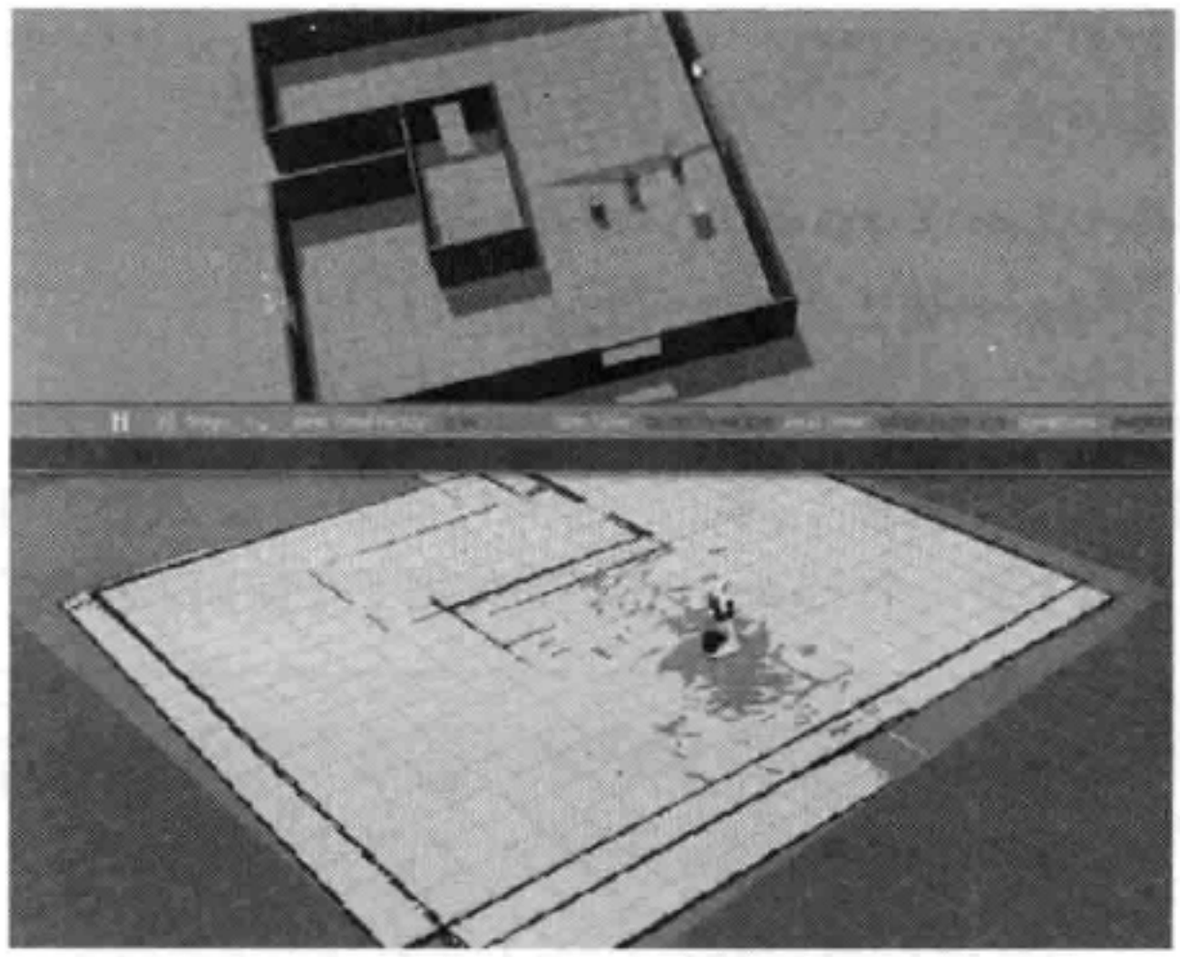
```
rosrun topic_toolrelay turtlebot_teleop/cmd_vel cmd_vel
```

然后直接使用键盘来移动机器人。同时，你可以启动 `rviz` 来查看机器人的传感器数据，如下图所示（`cfg` 文件夹下有一个 `reem.rviz` 文件用于安装 REEM 所需的 `rviz` 接口）：

类似地，通过使用 URDF 模型，能够查看 REEM 的所有的 TF 变换和坐标系。如下图所示：



最后，下图展示了在机器人完成移动之后，在 Gazebo 中所创建出的一张周围环境的地图。同时，在该图中，我们还能看到机器人能够通过所建立的地图使用 ROS 中的路径规划器以及进行机器人定位时概率估计算法所使用的粒子，进行导航。



9.2 PR2——柳树车库机器人

PR2 是由创建了 ROS 系统的柳树车库公司开发的移动式操作机器人。这个机器人装备了两个手臂和一个头部。它的头部带有可调远近视距的双目视觉系统以及激光雷达系统、上半身可以通过液压抬起或下降，通过 8 个可左右旋转的轮子实现全向移动。

9.2.1 安装 PR2 仿真环境

我们可以直接通过以下命令使用 Debian 安装包安装 R2 机器人：

```
sudo apt-get install ros-fuerte-nasa-r2-simulator
```

或者，也可按照以下指导通过其他软件源完成安装。

想要运行 PR2 机器人的模拟器，你需要使用 Gazebo 和 PR2 机器人的 URDF 模型。我们需要调用以下命令：

```
sudo apt-get install ros-groovy-pr2-simulator
```

这会安装若干个功能包及功能包集。这些包的功能都很强大，不仅可以用于移动机器人、控制手臂，甚至还能够组合起来做很多复杂的仿真。例如，PR2 机器人能够接受指令自动打开一扇门。但要实现这种程度的复杂功能，还需要很多 demo 文件，而这些文件和具体功能并不在本书的介绍范围内。

请注意，对于 PR2 机器人来说，我们还是需要使用 ROS Groovy。当然你可以尝试使用其他发行版。如果能使用更新版本的 Gazebo 肯定是最好的。你可以在最新版本中检查其运行是否正常。我们通过以下指令启动 Gazebo:

```
gazebo
```

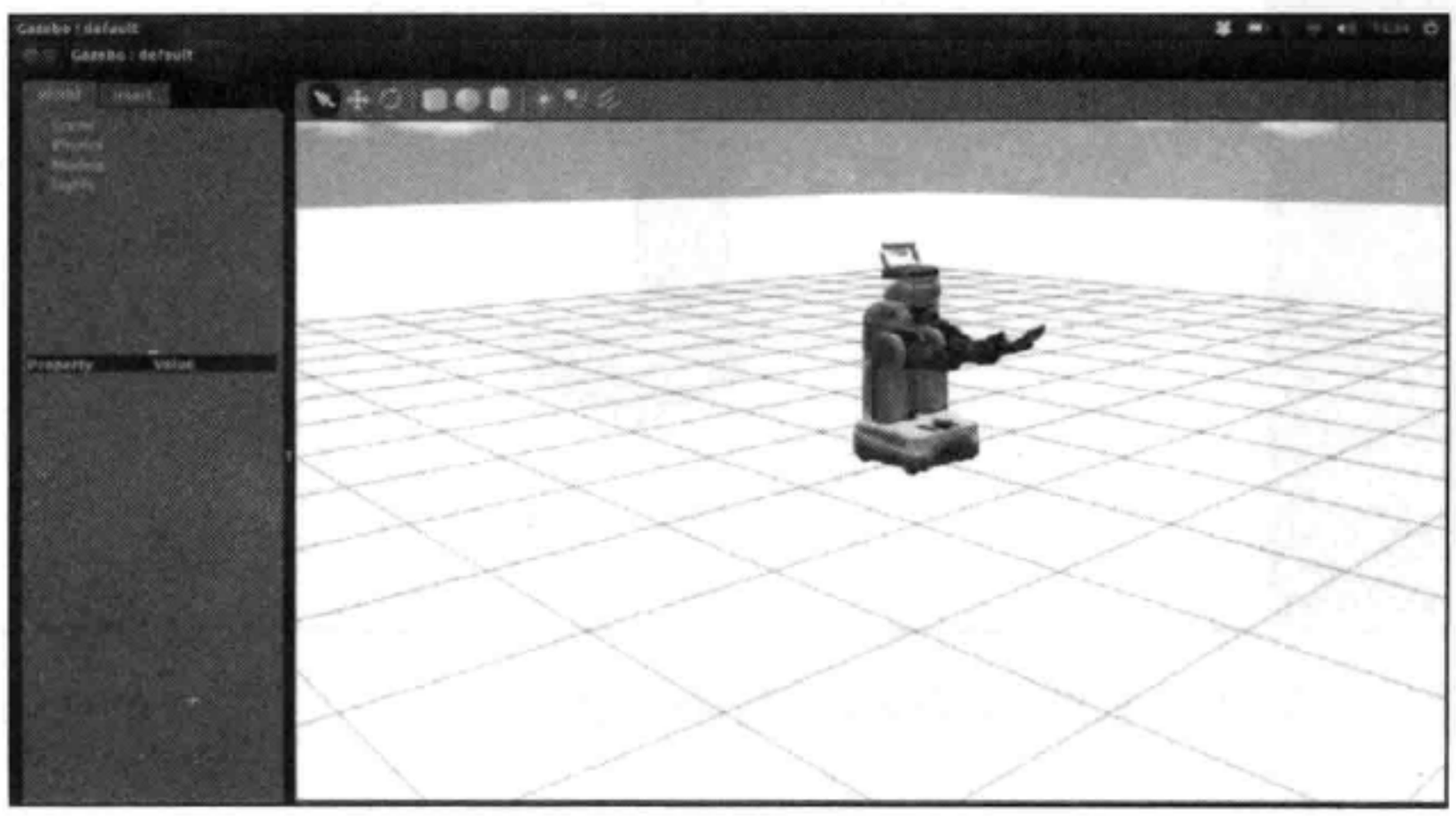
最新版本的 Gazebo 也会集成在最新的 ROS 系统发行版中，如果你感兴趣的话可以直接安装最新的 ROS 系统。如果你使用了新版本的 ROS，你也要试一下，看看 PR2 机器人是否在 ROS 环境中被很好地集成。

9.2.2 在仿真环境中运行 PR2

下面我们来学习如何启动仿真环境查看 PR2 的模型，并移动机器人底盘和手臂。我们可以在 Gazebo 中将 PR2 机器人加载到一个空的背景环境中：

```
roslaunch pr2_gazebo pr2_empty_world.launch
```

类似地，也可以通过使用 `pr2_gazebo` 功能包中 `launch` 文件夹下不同的启动文件将 PR2 机器人加载到其他的环境中。如果你使用的是上面提到的空的背景环境，就会看到如下图所示：



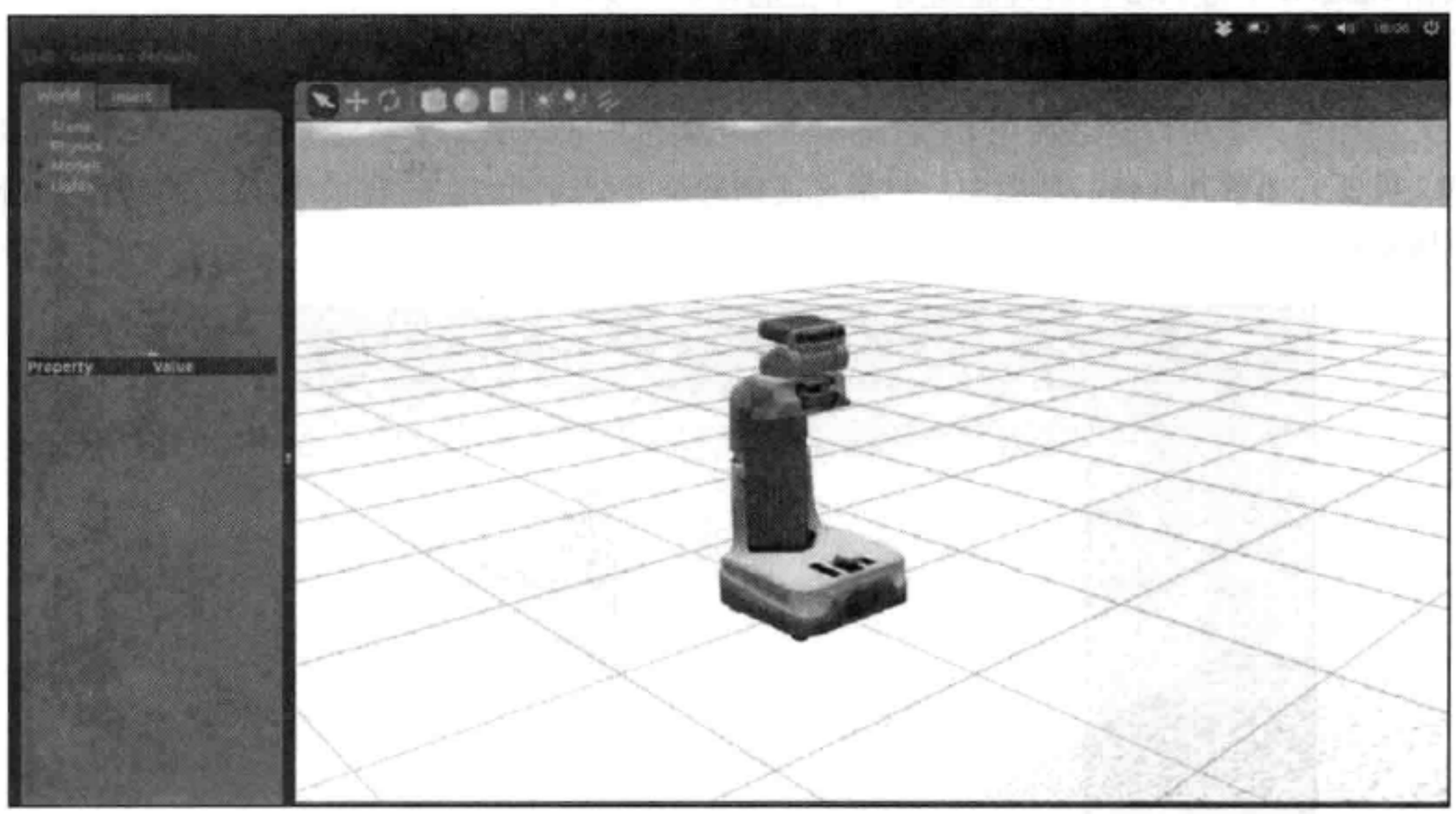
请注意，Gazebo 已经运行了加载模型所需的若干个启动文件。我们还能加载一个没有手臂的机器人，这样会减轻一些不必要的系统仿真负荷。如果你在仿真中不需要使用手臂，那么可以使用以下命令运行 PR2 机器人：

```
roslaunch pr2 pr2_no_arms_empty_world.launch
```

这使用了 chapter9_tutorials 文件夹下的启动文件 pr2_empty_world.launch，其中包含了以下代码：

```
<launch>
  <!-- start up empty world -->
  <arg name="gui" default="true"/>
  <arg name="throttled" default="false"/>
  <arg name="paused" default="true"/>

  <include file="$(find gazebo_worlds)/launch/empty_world_paused.
launch">
    <arg name="gui" value="$(arg gui)" />
    <arg name="throttled" value="$(arg throttled)" />

    <arg name="paused" value="$(arg paused)" />
  </include>

  <!-- start pr2 robot w/o arms -->
  <include file="$(find pr2_gazebo)/launch/pr2_no_arms.launch"/>
</launch>
```

我们会在 Gazebo 中看到如下图显示：



如果你想要移动 PR2 机器人，那么需要启动仿真程序：

```
roslaunch pr2_gazebo pr2_empty_world.launch
```

然后可以通过调用 Debian 安装包中的一个节点，使用键盘或者游戏杆来远程控制机器人：

```
sudo apt-get install ros-groovy-pr2-teleop-app
```

如果通过键盘控制机器人，那么使用：

```
rosrun pr2_teleop teleop_pr2_keyboard
```

如果你需要使用游戏杆，那么你还要另外一个节点。使用下面这个启动文件会将两个节点同时启动。

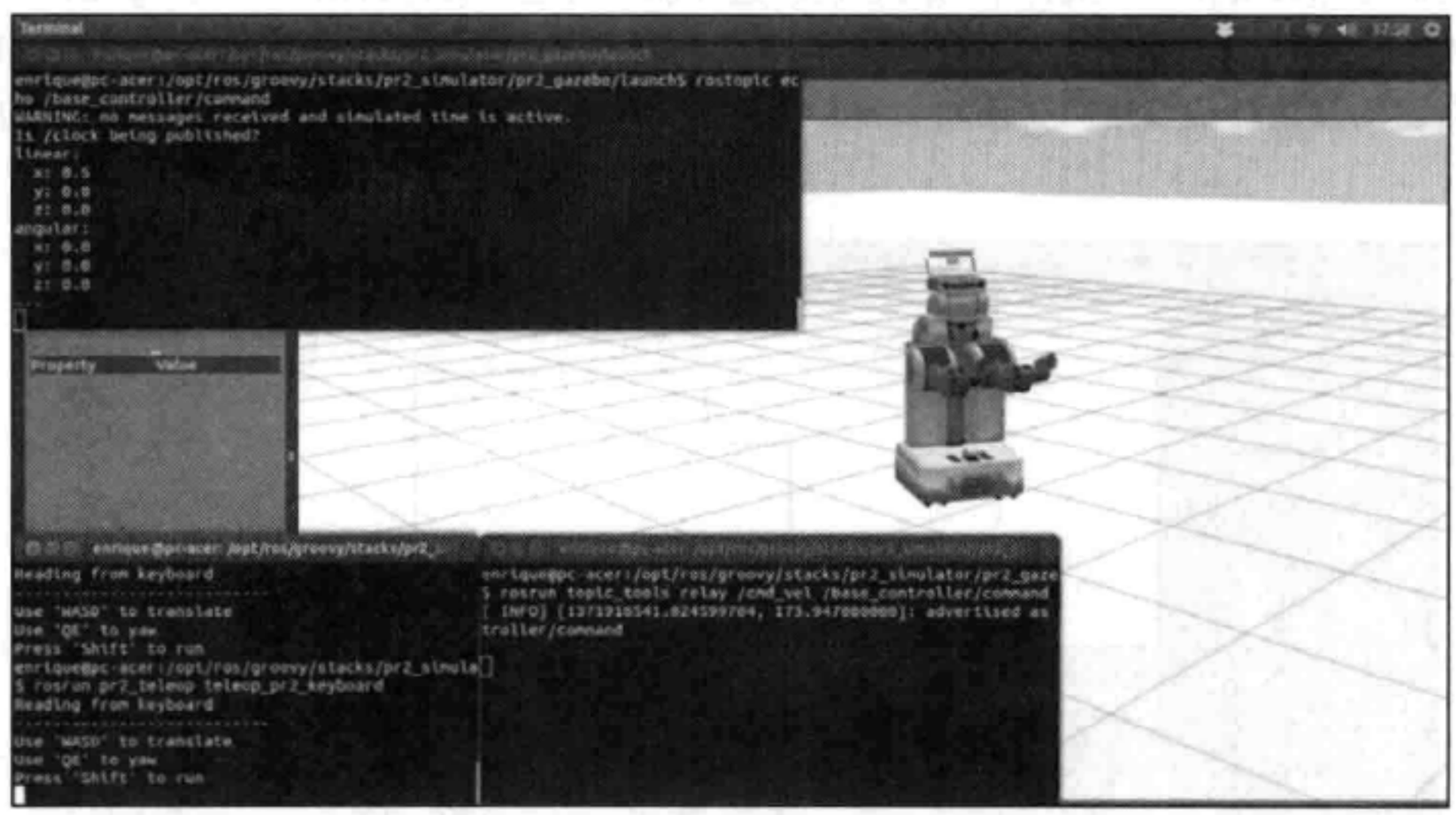
其中，我们需要将 PR2 机器人的 /base_controller/command 与 /cmd_vel 主题相连。使用如下命令：

```
rosrun topic_tools relay /cmd_vel /base_controller/command
```

我们通过输出 Twist 消息来检查机器人收到的速度命令：

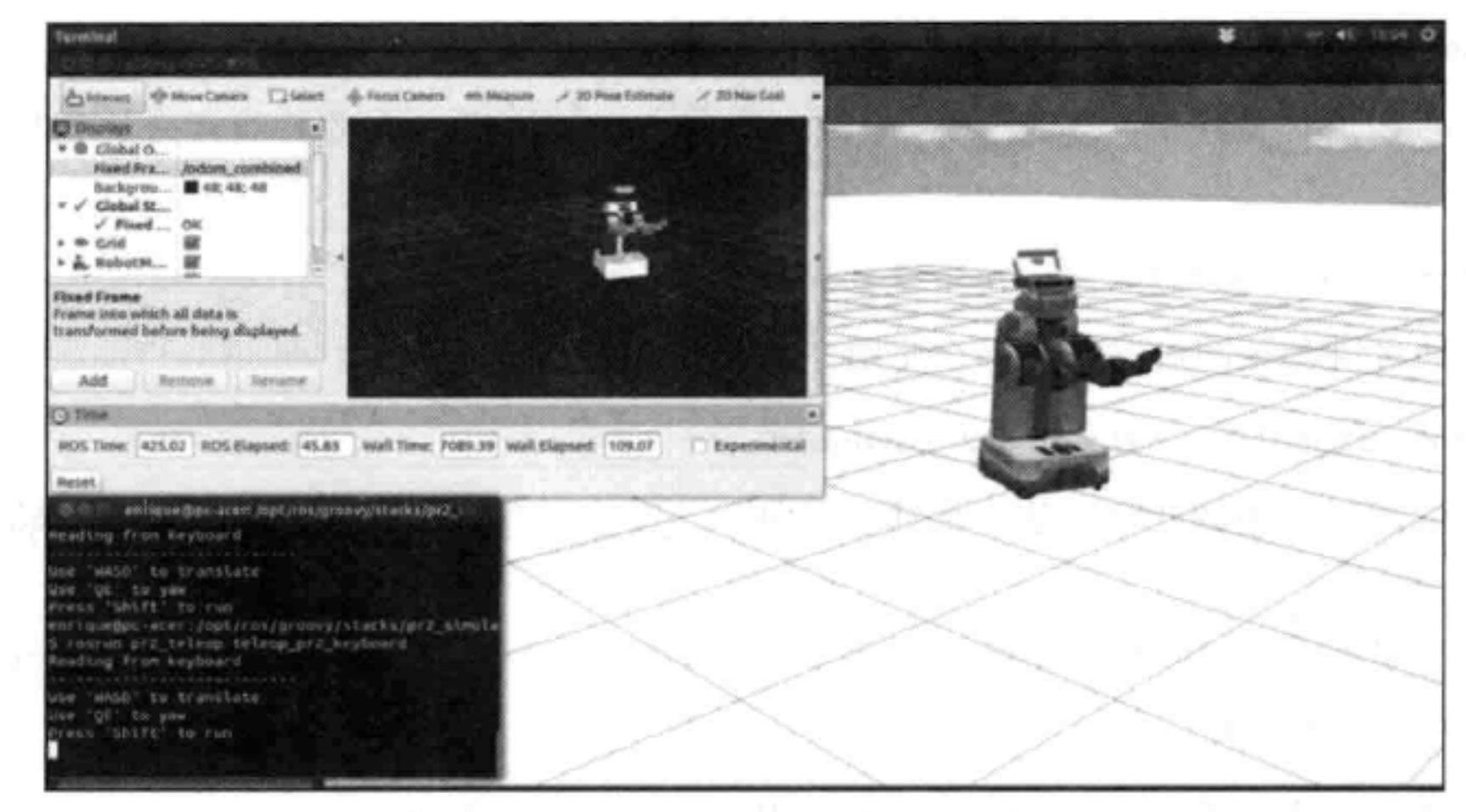
```
rostopic ho /base_controller/command
```

下图显示了正在运动中的机器人：



最后，在 rivz 中使用 /odom_combined 坐标系来查看机器人，如下图所示：

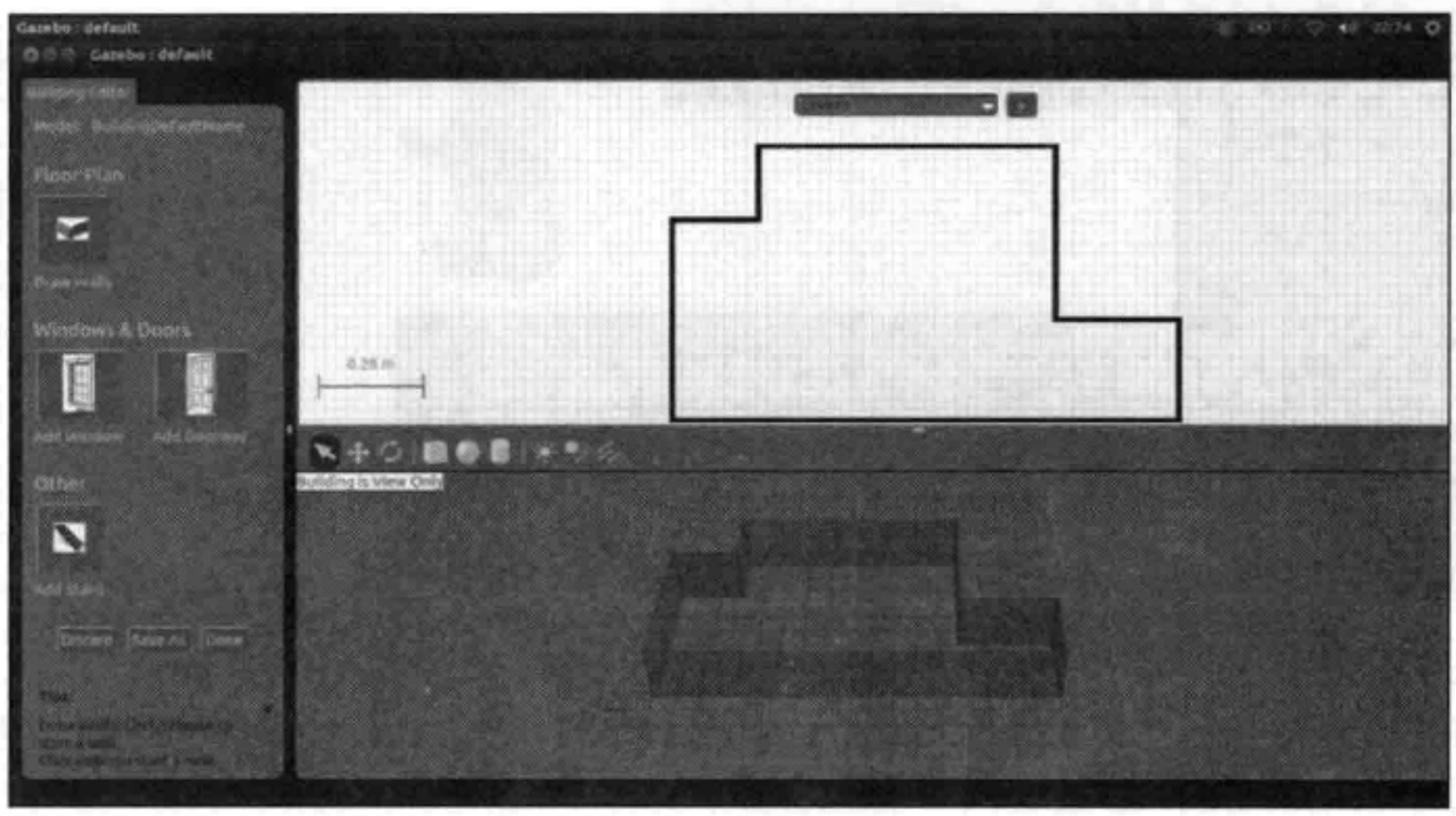
PR2 与 ROS 导航功能包集的集成非常完美，因此我们可以在 Gazebo 中加载适当的环境，创建地图并调用自适应蒙特卡罗定位算法来实现机器人定位，或者在机器人移动过程中通过路径规划器向机器人发送一个指定目的地。



9.2.3 生成地图与定位

想要在仿真环境中给定某一种背景环境，我们需要创建一个用于机器人定位的背景布置图。有网站内提供了大量的模型和背景环境：http://gazebosim.org/models/。

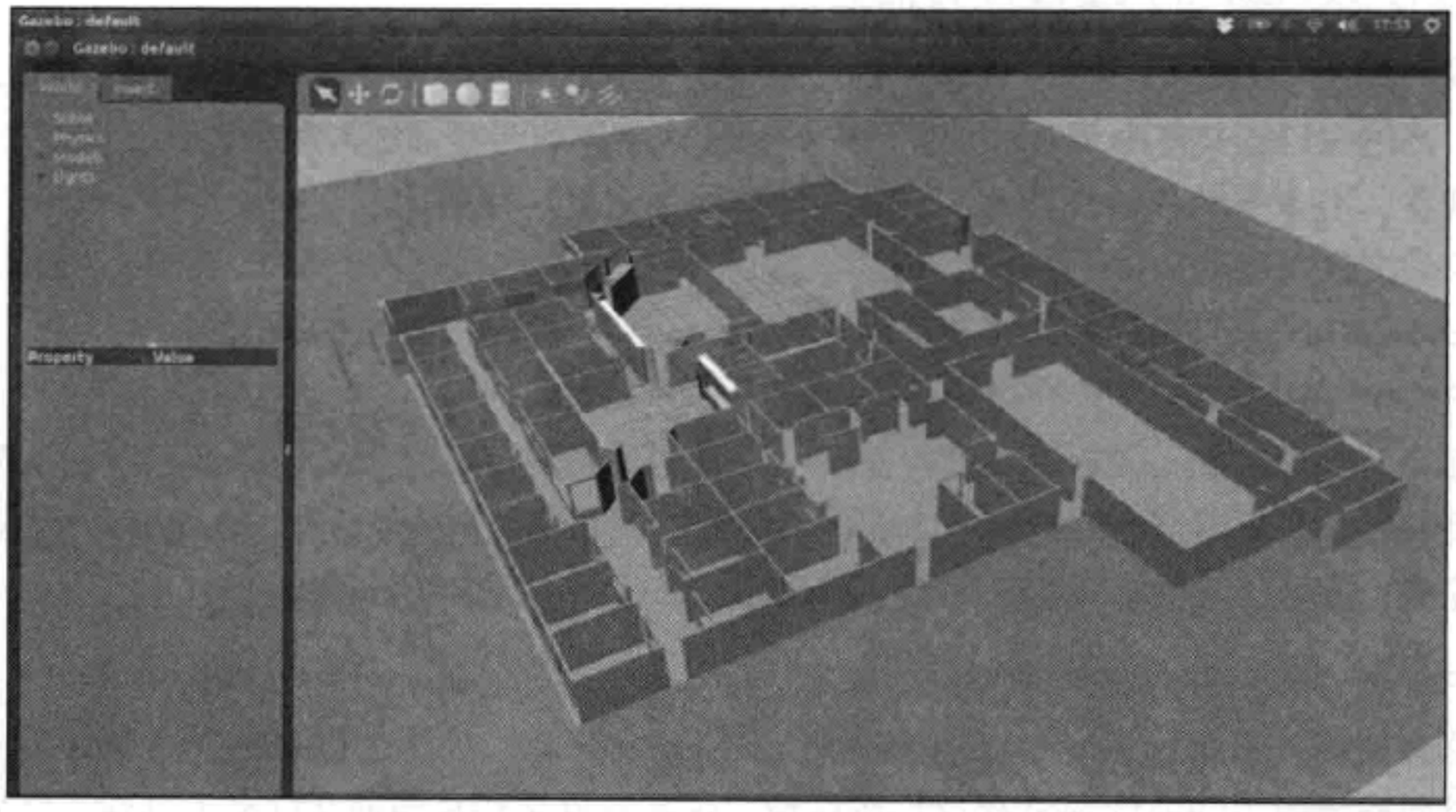
我们可以直接在环境文件中引用它们，这样能够将这些文件自动下载到 .gazebo 主目录下（在 $HOME 变量所示路径下），也可以手动的插入这些文件。还有就是能够在 Gazebo 中定义自己的背景环境。这需要使用 Building Editor。在示例中，我们会使用一个这样的简单房间：



我们也能找到并使用更复杂的背景环境，但是使用这些背景环境会增加完成地图的仿真建模所需要的时间。下面是柳树车库办公室环境的一个示例，我们可以通过以下指令将其加

载到 Gazebo 中：

```
roslaunch pr2_gazebo pr2_wg_world.launch
```

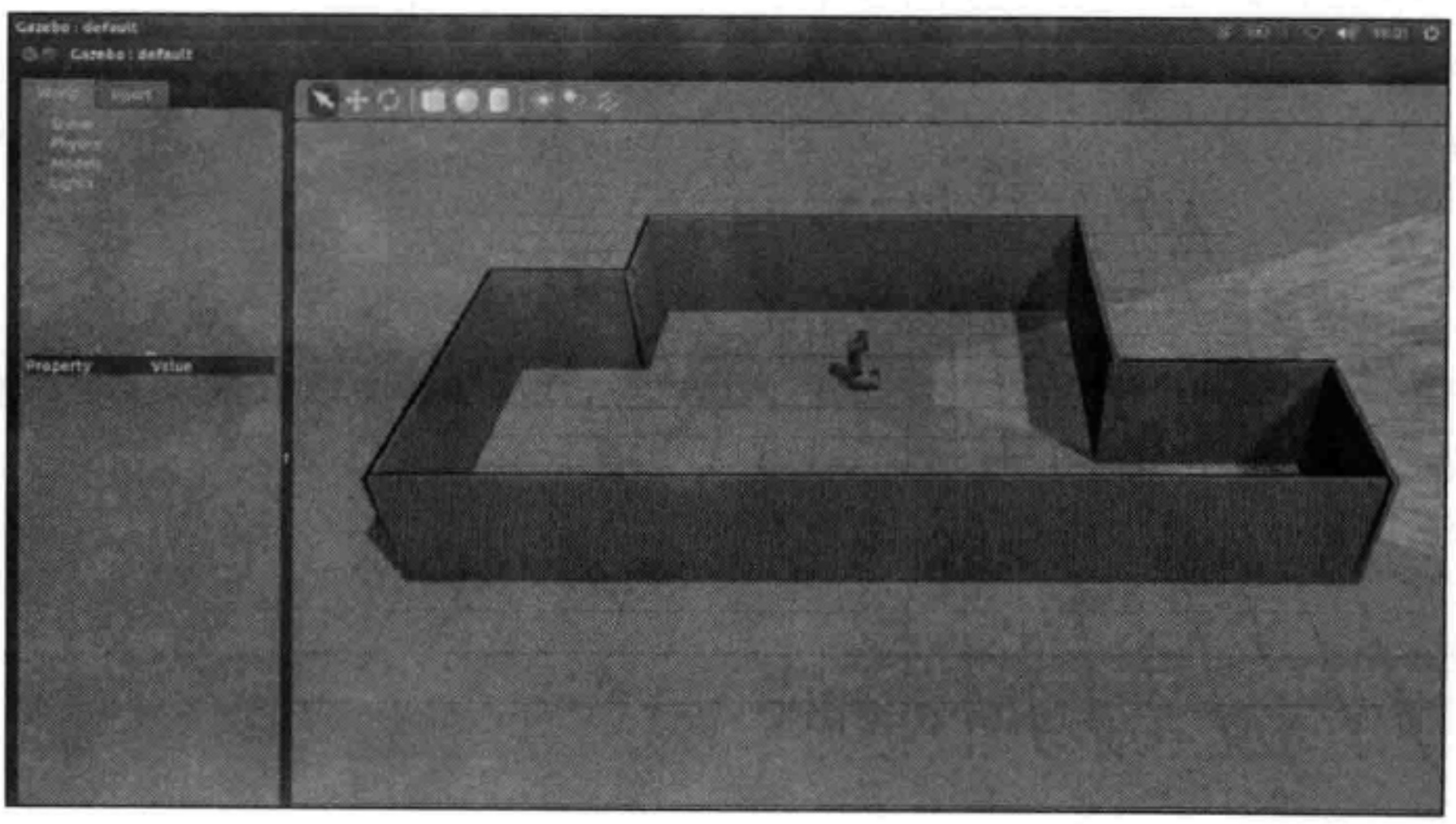


简单的房间背景环境被保存为 .sdf 文件。这样能方便我们加载一个背景环境，或者在 Gazebo 中不断修改成你所需要的背景环境。我们提供了一个启动文件来加载背景：

```
roslaunch pr2 simple_world.launch
```

然后可以将 PR2 机器人的模型加载到环境中（请注意在一些版本的 Gazebo 中你还能看到人工渲染的效果，并且还能够保证工作正常）：

```
roslaunch pr2 pr2_no_arms_simple_world.launch
```



现在，我们需要根据环境建立一个地图来完成导航。我们会使用 gmapping 算法来实现这个功能。运行本书附加代码中提供的以下启动文件：

```
roslaunch pr2 pr2_mapping.launch
```

我们要使用 PR2 机器人软件包中自带的启动文件加载先前设计的背景环境和 gmapping 节点。当然在启动文件中要对部分参数进行调整，具体的调整方法大家可以自己试一下。

现在，准备手动移动机器人了。无论你是使用键盘还是使用游戏杆，你可以选择一个对应的启动文件来启动：

```
roslaunch pr2_teleop teleop_joystick.launch
roslaunch pr2_teleop teleop_keyboard.launch
```

你会看到如何在 rviz 中逐渐构建出一个地图。使用启动文件之前同样也有一些配置参数需要设置，只有适当的配置参数才能保证你能看到完整的地图逐渐被构建出来。一旦地图建立完成，并且建立的结果比较理想，那么你能够使用以下命令保存地图：

```
rosrun map_server_map_saver
```

这会创建一个 map.yaml 文件和一个 map.pgm 文件。一个会保存地图特征，另一个会用**覆盖网格地图（OGM）**格式保存地图图像。这些文件直接在当前文件夹下创建，然后你可以将它们复制到本章中 pr2 文件夹的 maps 文件夹下。其中，你还会看到本书中附带的两个文件。这些是我们已经创建好的地图文件，可以方便读者的使用。这些文件可以用于下一步工作：定位。

对于机器人定位和路径规划，我们会使用自适应蒙特卡罗算法（AMCL）和 costmap_2d 中的 move_base 功能包。这个功能包提供了周围环境的代价地图，从而保证机器人不会与周围环境中的障碍物相撞。

你可以从 maps 文件夹下运行以下启动文件并自动加载之前创建的地图。

```
roslaunch pr2 pr2_navigation.launch
```

这会启动使用了 maps/map.yaml 文件的 map_server 节点、amcl 算法和 move_base 节点。

9.2.4 在仿真环境中运行 PR2 演示程序

只要我们有环境地图，并能够在地图中进行定位和移动机器人，就可以运行一些 PR2 机器人的 demo 程序了。这样程序包含了一些抓取物体或者是与环境互动的示例。

你可以让 PR2 机器人去开门，或者看它是如何将自己的电源插到插座中充电，或者是从桌子上拿一个物体。你只需要将本书中所学的知识与以下链接中的指导相结合，一步步来完成这些实验：

- http://ros.org/wiki/pr2_simulator/Tutorials/PR2OpenDoor/diamondback

- http://ros.org/wiki/pr2_simulator/Tutorials/PR2PluggingIn
- http://ros.org/wiki/pr2_simulator/Tutorials/SimpleGraspingDemo

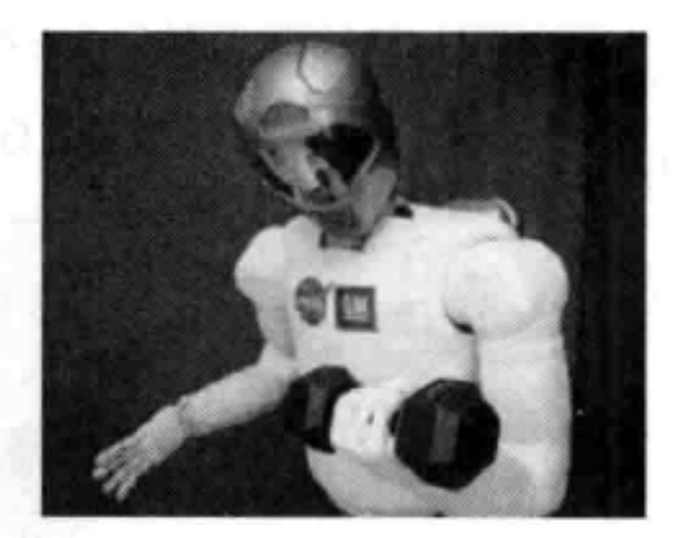
在某些示例中，你可能还需要下载并编译源文件，但是也可以直接安装柳树车库的 PR2 应用程序：

```
sudo apt-get install ros-groovy-wg-pr2-apps
```

这些应用程序包含了之前列出的示例，也有一些我们没有提到的。你可以自己试着探索一下。

9.3 Robonaut 2——NASA 的敏捷型人形机器人

Robonaut 2 简称为 R2，是由美国航空航天局开发的第二代敏捷型人形机器人。这个机器人能够帮助宇航员完成空间任务。它主要的功能是具有非常敏捷的执行机构（多自由度手臂和机械手）。在本书编写时，美航局已经完成了四台机器人的建造工作。在本书中，我们也会介绍 R2 在 ROS 系统中的仿真环境以及网络上可用的软件和代码资源。



右图中就是已经被送到国际空间站（ISS）上的 R2 机器人。这是 STS-133 任务的一部分，也是第一台登上太空的空间机器人。但是要说明的是，它在被送上国际空间站之后，被安装在了一个固定的底座上。其具体的样子我们也会在后面介绍的仿真环境中看到。然而，后续的 Robonaut 则会被安装上四个轮子或者是特殊定制的腿。

9.3.1 从软件源安装 Robonaut 2

首先，我们需要下载源代码，在 bitbucket 网站上有几个源。我们有两种选择，一个是分别下载这几个软件源，如下所示：

- `git clone https://bitbucket.org/nasa_ros_pkg/nasa_rosdep.git`
- `git clone https://bitbucket.org/nasa_ros_pkg/nasa_r2_common.git`
- `git clone https://bitbucket.org/nasa_ros_pkg/nasa_r2_simulator.git`

还有一个就是使用 `rosinstall` 命令，它会自动执行上面提到的三个指令。在 ROS Groovy 环境下，我们可以直接使用以下命令：

```
Rosinstall . https://bitbucket.org/nasa_ros_pkg/misc/raw/master/nasa-ros-
pkg.rosinstall
```

无论如何，不要忘了更新功能包和功能包集的数据库：

```
rosstack profile && rospack profile
```

一旦下载了 R2 的源代码，我们就能够编译它们：

```
roscd nasa_r2_simulator
```

你可以选择 master 分支。但是在示例中，我们使用的是 ROS Groovy，所以我们需要选择 groovy 分支。使用如下指令：

```
git checkout groovy
```

使用以下命令来编译代码：

```
rosmake
```

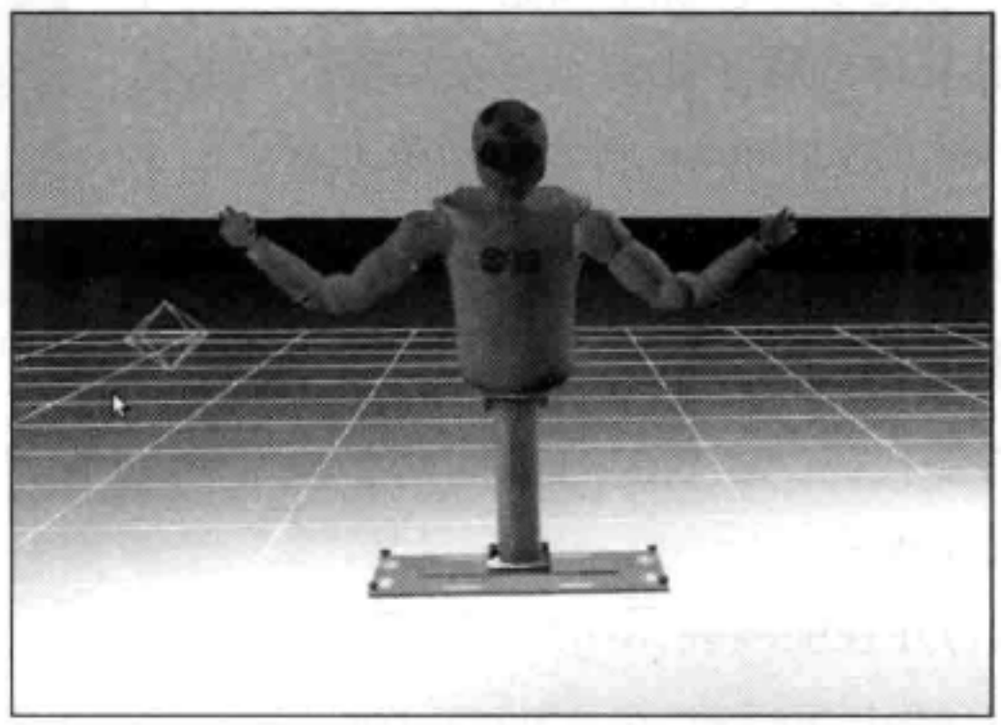
这会编译 R2 的仿真环境。它也会依赖于很多功能包和功能包集（都在我们下载的软件源中包含了）。

9.3.2 在国际空间站的固定支座上运行 Robonaut2

首先，第一个最基本的示例是在一个空的背景下启动带有 R2 机器人的 Gazebo 仿真环境。

```
roslaunch r2_gazebo r2_empty.launch
```

你会看到 R2 机器人和 Gazebo 中空的背景，如下图所示：



1. 控制 Robonaut 2 的手臂

首先要做的事情是将机器人放置到正确的位置上：

```
rosrun r2_controllers_gazebo r2_readypose.py
```

然后需要选择手臂的控制模型。这两个模型对于所有机器人手臂都是通用的。

- **关节模式**（Joint mode）：会对每一个关节单独设置角度。
- **笛卡尔模式**（Cartesian mode）：给定了每个连接的期望位姿，例如，指定末端执行器的位置和方向。算法驱动会自动求解逆运动学公式以获得每个关节的角度状态，以便使指定连接能移动到期望位姿。当然计算的前提是计算结果能够被求解。

默认情况下，机器人会使用关节模式。当然，在不同控制模式之间的切换也很简单。如果想要进入笛卡尔模式，只需要使用以下命令：

```
rosservice call /r2_controller/set_tip_name <arm_name> <link_name>
```

其中，arm_name 参数用于指定哪只手，这里可以是 left 或者 right；link_name 参数则代表了手臂上的某一个连接名。例如，左手中指就是以 left_middle_base 为名的连接。

类似地，想要进入关节控制模式，只需要运行：

```
rosservice call /r2_controller/set_joint_mode <arm_name>
```

无论我们在哪种模式下，都能够通过主题来控制手臂的位姿。在关节模式下，我们可以发布 JointState 消息到这些主题中的任意一个。这些主题分别对应着机器人的左手和右手：

```
/r2_controller/left_arm/joint_command
/r2_controller/right_arm/joint_command
```

头部和腰部也都是通过 JointState 来进行控制。机器人的头部是一种云台设备，通过 /r2_controller/neck/joint_command 主题进行控制；机器人的腰部通过 /r2_controller/waist/joint_command 主题进行控制。

2. 使用互动标记轻松地控制机器人

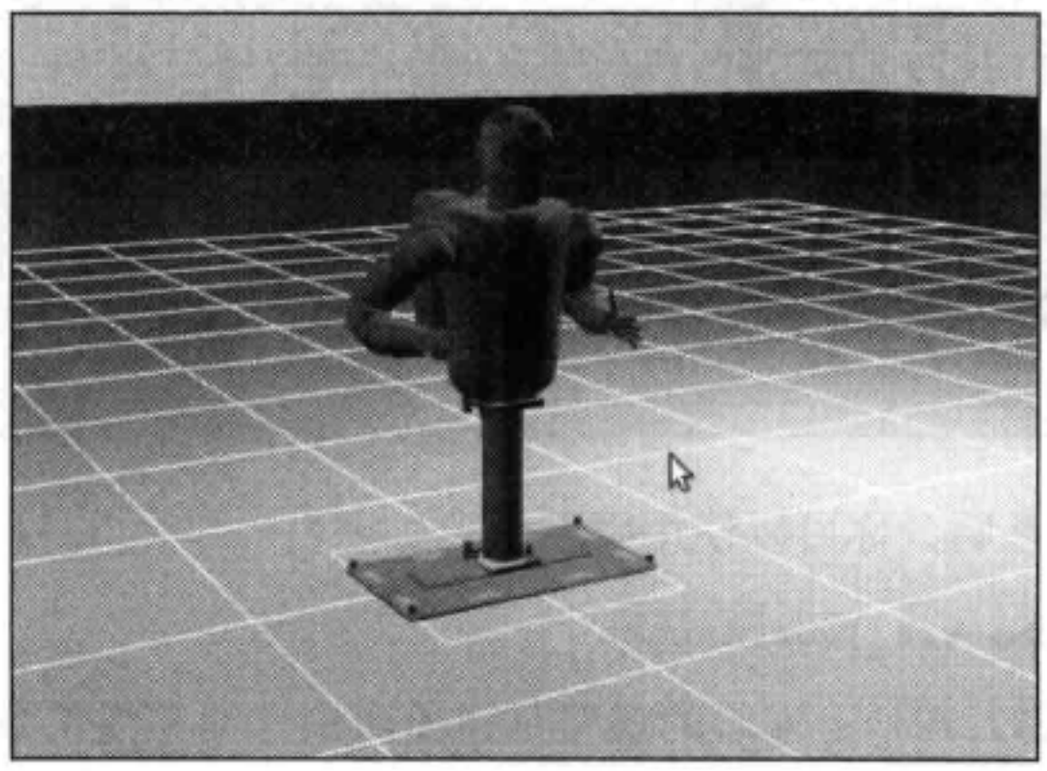
在 rviz 中控制机器人最灵活的方法是通过互动标记。当 R2 机器人的仿真环境运行时，打开 rviz：

```
rosrun rviz rviz
```

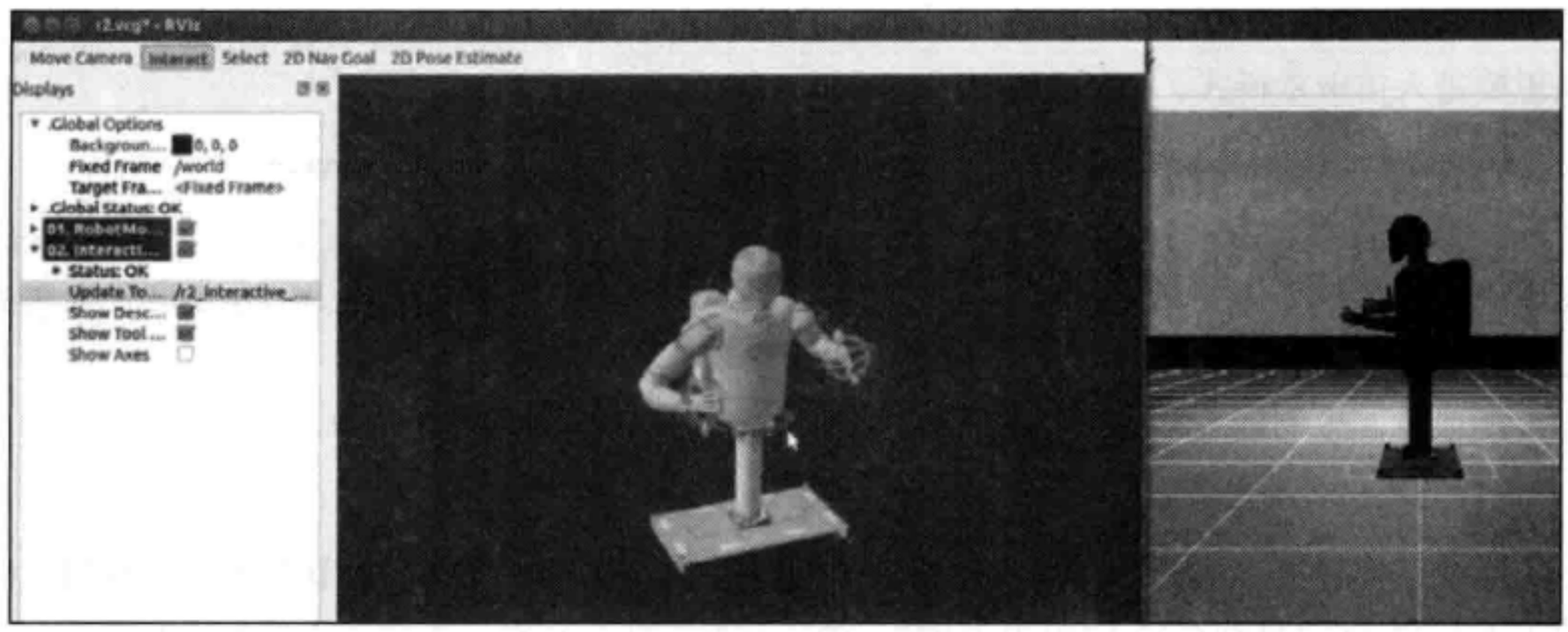
然后，打开远程互动控制器：

```
rosrun r2_teleop r2_interactive_control.py
```

你能够在启动 rviz 时在 r2/config 中使用本书提供的 vcg 文件，这个文件用于 R2 机器人和互动标记。在下图中，你能看到正常情况下两个手臂的位置：



如果使用了互动标记，你则会看到下面的图像：

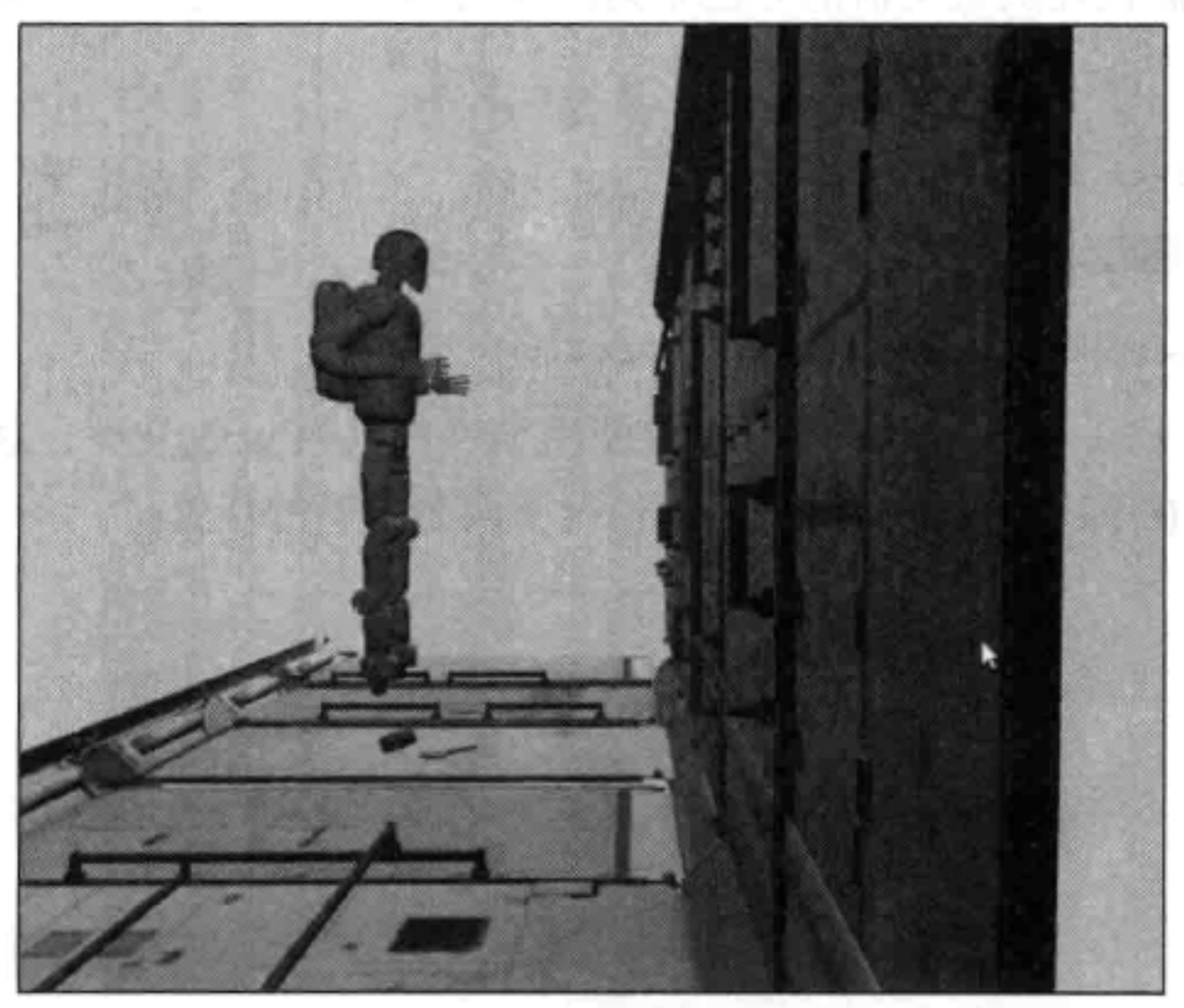


我们会发现在 rviz 中可以直接控制这些互动标记，直接给定手的期望位置，并且通过逆运动学计算手臂每个关节的位置，让整条手臂最终到达一个合适的位姿。

3. 为 Robonaut 2 安上腿

在仿真环境中，我们能够查看带有腿的 R2 机器人或者只是两条腿。如果想要运行整个身体，可以使用以下命令：

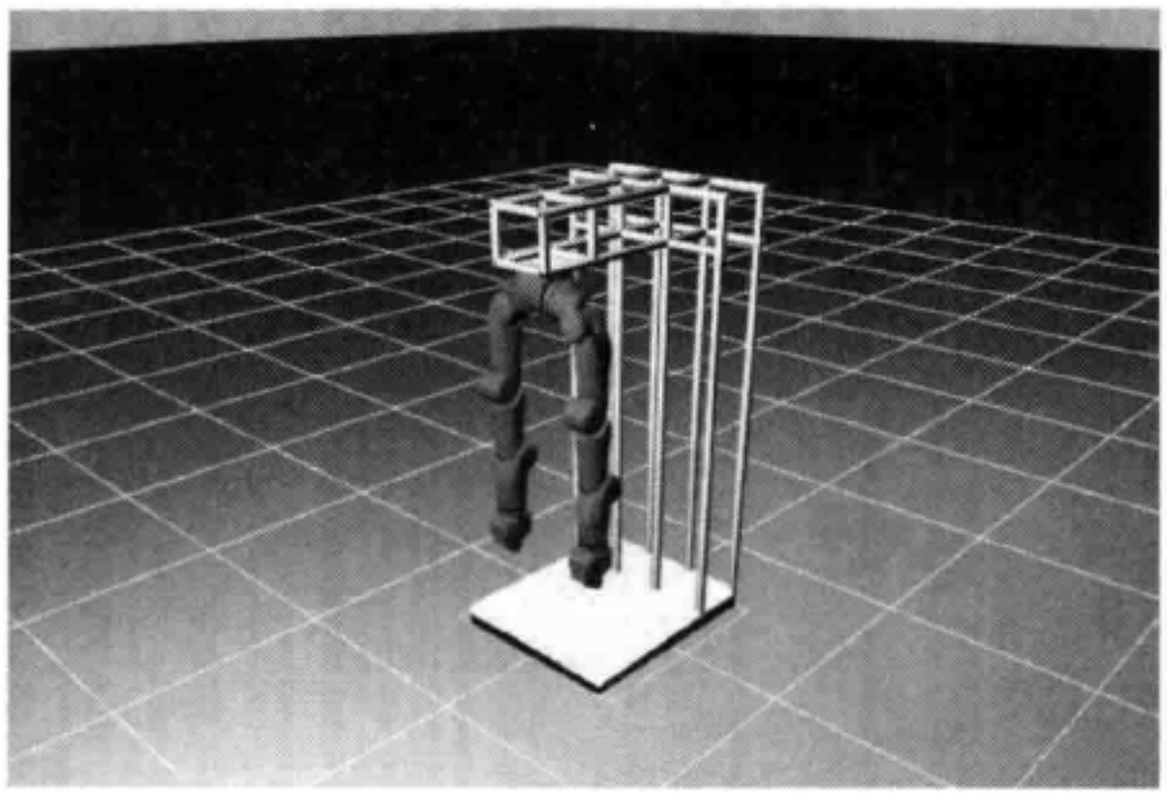
```
roslaunch r2_gazebo r2c_full_body.launch
```



如果你只想运行两条腿，你可以使用以下命令：

```
roslaunch r2_gazebo r2c_legs.launch
```

这些腿是对 R2 机器人四号上安装的攀爬足建模而来，而不仅仅是仿真模型而已。



R2 机器人还提供了若干个 Gazebo 插件来完成机器人仿真。这些插件都会被 Gazebo 自动加载和运行。

4. 加载国际空间站环境

我们还可以为 R2 机器人加载国际空间站环境：

```
roslaunch r2_gazebo r2_ISS.launch
```

如下图所示，你会在 Gazebo 中看到 R2 机器人被固定在国际空间站中的底座上。

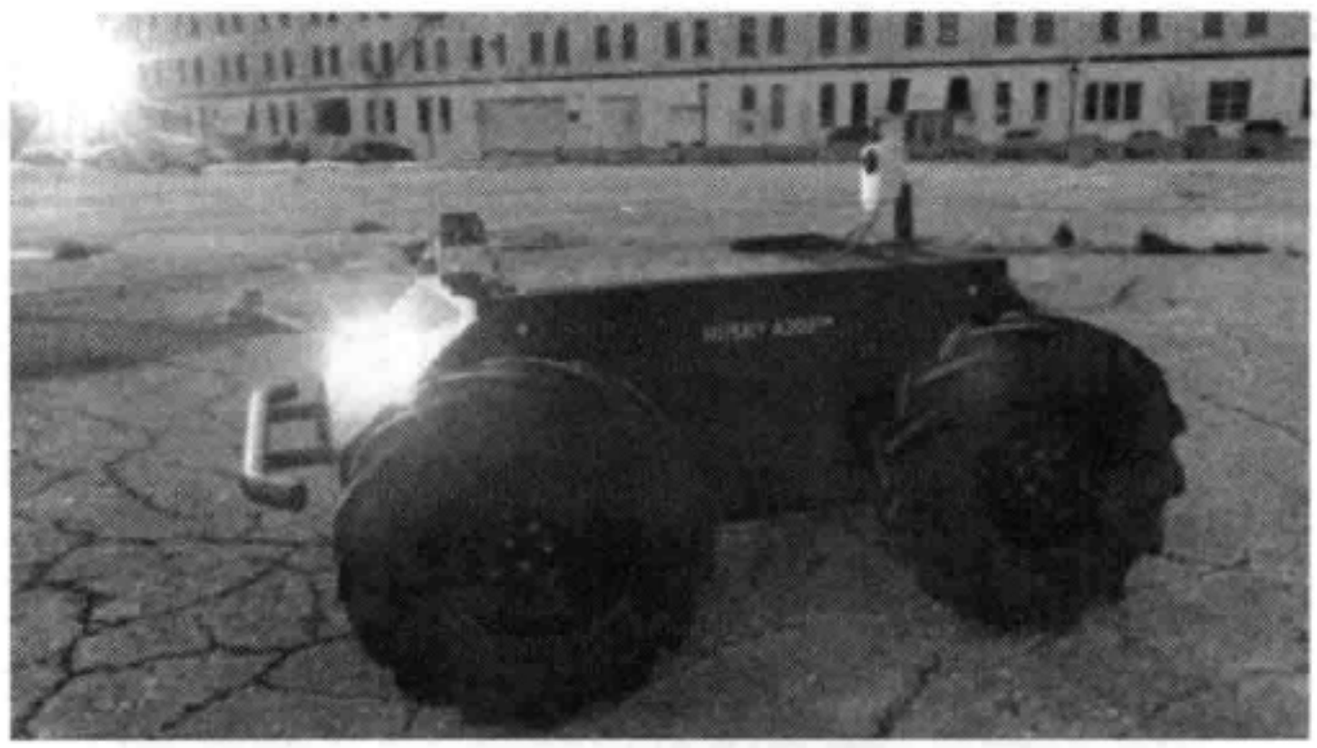
这可能会消耗系统中较多的资源，所以你可以选择仅加载国际空间站的工作面板。如下所示：

```
roslaunch r2_gazebo r2_taskboard.launch
```

在下图中，你就只会看到 R2 机器人和国际空间站的工作面板。这和我们在国际空间站完整环境中看到的工作面板是同样的，只是不再包含国际空间站环境模型的其他物体。

9.4 Husky——Clearpath 的轮式机器人

Husky A200 型机器人是由 Clearpath 机器人公司研发的一种**无人驾驶地面巡逻机器人**（UGV）。像 OSRF 一样，Clearpath 是柳树车库的一家子公司。在下图中我们能看到，它是一种结实的户外型机器人。它装备有可旋转摄像头、GPS，并集成了 ROS 软件。



9.4.1 安装 Husky 仿真环境

我们想要在仿真环境中运行 Husky，首先需要使用以下命令安装它：

```
sudo apt-get install ros-groovy-husky-simulator
```

或者，你可以从软件源下载源代码的方式以保证你能够获得最新版的软件：

```
git clone git://github.com/clearpathrobotics/husky_simulator.git
```

9.4.2 运行 Husky 仿真环境

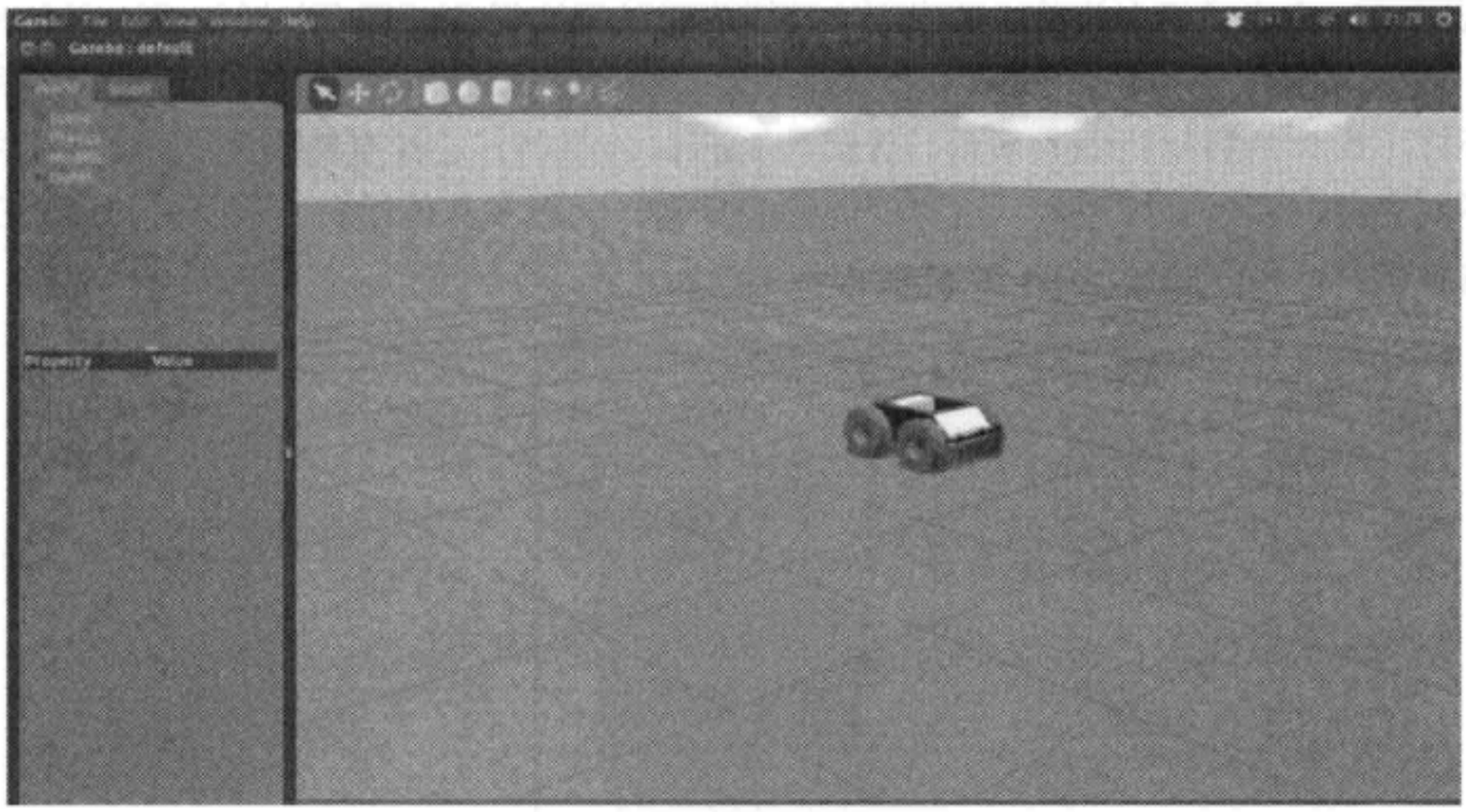
我们在 Gazebo 中加载一个空地图：

```
roslaunch husky_gazebo map_empty.launch
```

然后使用以下命令加载 Husky：

```
roslaunch husky_gazebo husky_base.launch
```

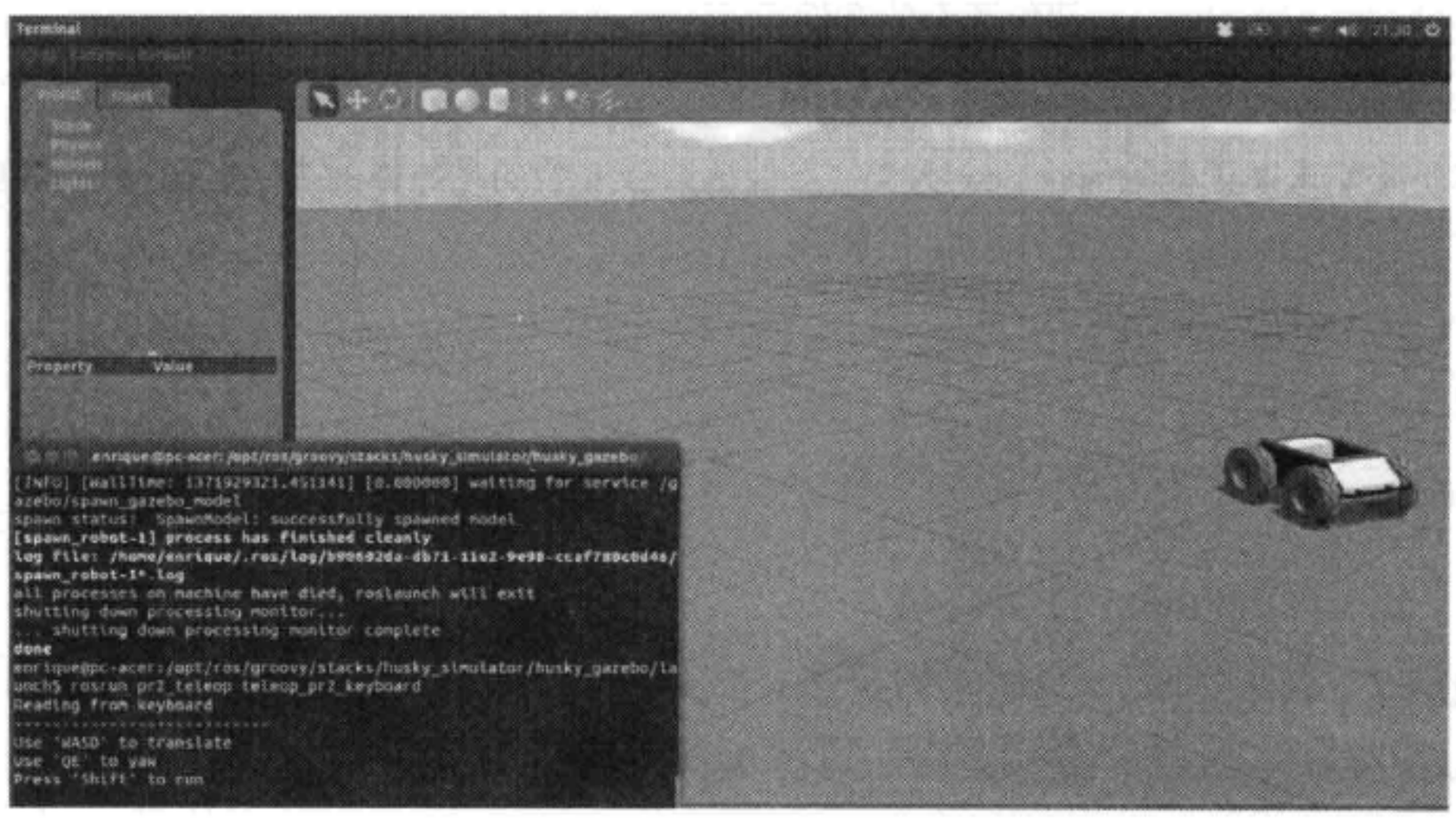
我们也可在 Gazebo 中手动完成这些工作。如下图所示，你会看到加载完成后的机器人：



然后，我们可以借用 PR2 机器人的远程操作节点来移动机器人。只需要调用以下指令：

```
rosrun pr2_teleop teleop_pr2_keyboard
```

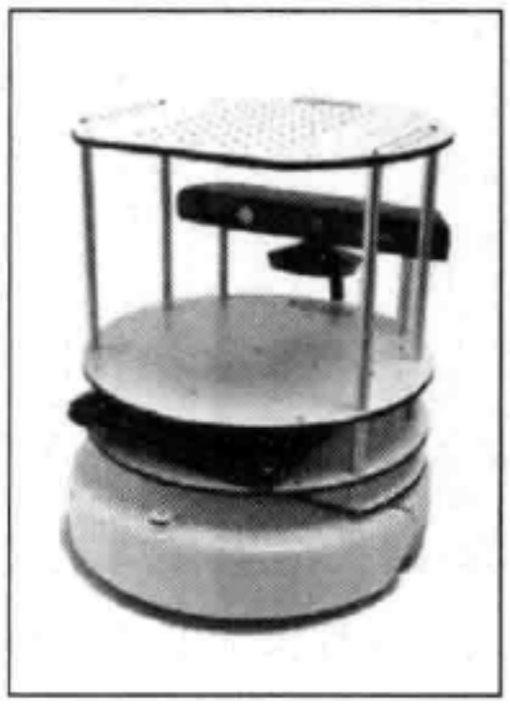
请注意，Husky 机器人并不是一个全向移动机器人，因此它不能够直接左右移动。因此，当你按下 A 或 D 键时，它是不会移动的。下图展示了移动后的 Husky 机器人：



现在，Husky 机器人已经集成了导航功能包集。你只需要应用本书中所学的各种知识，进行一些机器人定位、创建仿真环境地图、自主导航等工作。

9.5 TurtleBot——低成本移动机器人

TurtleBot 集成了很多简单、低成本、技术先进的硬件设备。其中包括了微软的 Kinect 传感器、Yujin 机器人公司的 Kobuki 机器人地盘和 iRobot 公司的一些附加组件。它是一种专为室内环境下设计的移动机器人平台。凭借着 Kinect 传感器，它具有精确的 3D 感知能力。它具有不同高度的多个支撑平台。这样就能够非常轻松地集成其他设备，甚至是使用自己的笔记本电脑来测试我们的程序。它的外观如下图所示：



9.5.1 安装 TurtleBot 仿真环境

TurtleBot 的仿真环境并不包含在 ROS 的官方软件源中，因此我们需要先下载 `turtlebot_simulator` 软件源中的文件：

```
git clone https://github.com/turtlebot/turtlebot_simulator.git
```

你还需要选择正确的 `master` 分支，以保证最后安装的是与你的 ROS 发行版相匹配的版本。对于 ROS Groovy，需要使用以下命令：

```
git checkout groovy
```

现在，编译代码：

```
rosmake
```

9.5.2 运行 TurtleBot 仿真环境

使用以下命令在空的背景环境中运行 TurtleBot：

```
roslaunch turtlebot_gazebo turtlebot_empty_world.launch
```

正如你所见，遵循典型的标准，使用 `<robot_name>_gazebo` 功能包的 launch 文件调用机器人。和之前介绍的其他机器人一样，现在你能够通过机器人的传感器来感知仿真环境，也可以进行地图创建、定位和导航任务。

9.6 本章小结

在本章中我们学习了几个机器人领域内最著名公司研发的机器人，并在 Gazebo 中对它们进行仿真。在本章中，你使用到了之前所有章节内所学的背景知识。现在你就能够使用本章所学来研究和使用世界上最重要的机器人研究机构和公司所开发的类似的工具和机器人。

我们展示了几个人形机器人，如 REEM、PR2 和 Robonaut。这些机器人最大的特定是将灵活的手臂集成到了躯干之上，允许你控制这些手臂抓取物体。类似地，它们也有一个可以活动的头部，并能将视线聚焦到周围环境中特定的某个物体或区域。这三种机器人包含了轮式驱动机器人、全向和非全向移动机器人。其中 REEM 和 Robonaut 的开发人员还开发出了双足型机器人。REEM-C 能够行走，R2 的双足型不仅能行走，还能在复杂的环境中进行攀爬。

在此之后，我们展示的其他机器人平台都是基本的轮式平台，提供了基本的支撑结构来放置传感器和笔记本。它们就是 Husky 和 TurtleBot。实际上，TurtleBot 是同类型中最便宜的机器人。你们已经学习了如何在仿真环境中运行这些机器人，如果有条件的话，也可以购买一个机器人。它能在实际环境中做一些 SLAM、定位、路径规划等。所有此类机器人所必须掌握的基础知识，在本书中都已经覆盖到了。

现在你就可以用你自己的机器人或者是世界上最广泛使用的机器人做些你想做的事情了。你可能会发现，你想实现的某些功能已经在 ROS 中集成了，或者是你能够很轻松地将你从网络上找到的其他组件和 ROS 系统集成起来。无论如何剩下的这部分都需要由你自己去探索。祝你有一个愉快的机器人程序设计之旅！