



ROS机器人程序设计

(原书第2版)

Learning ROS for Robotics Programming Second Edition

恩里克·费尔南德斯 (Enrique Fernández)
路易斯·桑切斯·克雷斯波 (Luis Sánchez Crespo) 著
[西班牙] 阿尼尔·马哈塔尼 (Anil Mahtani)
亚伦·马丁内斯 (Aaron Martinez)

刘锦涛 张瑞雷 等译



机械工业出版社
China Machine Press

ROS机器人程序设计

(原书第2版)

Learning ROS for Robotics Programming Second Edition

[西班牙] 恩里克·费尔南德斯 (Enrique Fernández)
路易斯·桑切斯·克雷斯波 (Luis Sánchez Crespo) 著
阿尼尔·马哈塔尼 (Anil Mahtani)
亚伦·马丁内斯 (Aaron Martinez)

刘锦涛 张瑞雷 等译



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 (原书第 2 版) / (西) 恩里克·费尔南德斯 (Enrique Fernández) 等著; 刘锦涛等译. —北京: 机械工业出版社, 2016.9

(机器人设计与制作系列)

书名原文: Learning ROS for Robotics Programming, Second Edition

ISBN 978-7-111-55105-8

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

中国版本图书馆 CIP 数据核字 (2016) 第 248166 号

本书版权登记号: 图字: 01-2016-1894

Enrique Fernández, Luis Sánchez Crespo, Anil Mahtani, Aaron Martinez: Learning ROS for Robotics Programming, Second Edition (ISBN: 978-1-78398-758-0).

Copyright © 2015 Packt Publishing. First published in the English language under the title “Learning ROS for Robotics Programming, Second Edition”.

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2016 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 机器人程序设计 (原书第 2 版)

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: 刘诗灏

责任校对: 董纪丽

印 刷: 中国电影出版社印刷厂

版 次: 2016 年 11 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 20

书 号: ISBN 978-7-111-55105-8

定 价: 69.00 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) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@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

Foreword | 推荐序一

21 世纪是什么样的世纪？是物联网的世纪？是 VR 的世纪？也许吧，但我更相信 21 世纪是机器人的世纪。

目前，我国已经步入经济转型的拐点区间，人口红利越来越难以支撑中国经济的发展和进步。在很多行业都已经开始了机器换人、生产工艺升级换代的步伐。工信部、发改委、财政部日前联合印发了《机器人产业发展规划（2016—2020 年）》。这份规划指出：“到 2020 年，我国工业机器人年产量达到 10 万台，其中六轴及以上机器人达到 5 万台以上；服务机器人年销售收入超过 300 亿元，在助老、助残、医疗康复等领域实现小批量生产及应用；要培育 3 家以上的龙头产业，打造 5 个以上机器人配套产业集群；工业机器人平均无故障时间要达到 8 万小时；智能机器人实现创新应用。”从这个规划中可以看出，机器人未来的政策空间和市场发展空间都是非常巨大的。一方面，发展工业机器人在满足我国制造业的转型升级、提质增效，实现“中国制造 2025”等方面具有极为重大的意义，是全面推进实施制造强国战略的重要一步。另一方面，从服务机器人来说，也要满足未来市场需求的增长。首先，这包括了基本生活需求，比如说养老、助老、助残等。其次是国家安全需求，比如救灾、抢险、海底勘探、航天、国防。最后还有家庭服务和娱乐机器人，比如娱乐、儿童教育、智能家居应用等。同时，科技部目前也在进行“十三五”科技创新规划战略研究，根据已经披露的内容，其中对机器人（尤其是服务机器人）非常重视，并会在近期遴选并启动一批相应的重大科技项目。

也许看完上面这些，你会觉得这更像是新闻描述，机器人产业真的这么火爆吗？当然。即使刨除工业机器人，只谈其他的智能机器人或服务机器人，这也是一个相当庞大的产业。例如，目前世界上最大的服务机器人公司应该还是美国的直觉外科公司。他们生产的达芬奇机器人系统在全世界已经应用了 3000 余套，其完成的手术超过千万例。由机器人完成的各类微创手术让无数患者获得新生。这家公司的市值超过 300 亿美元。中国目前估值最高的机器人公司是大疆创新，其年营收额已经超过 10 亿美元，估值超过百亿美元。要知道，10 年前这家公司刚刚在深圳创立时，还比较弱小。因此，相信机器人产业在未来的前景一定非常

广阔。

但是我们也注意到在军用国防、救灾救援、养老、家庭服务、儿童教育等领域，至今没有一个世界级的机器人公司存在。有很多爱好者都希望自己能够制造一个智能轮椅、一个智能儿童教育机器人或是一个家庭服务机器人，也许这样的机器人会像乔布斯和沃兹尼亚克在车库里做出的苹果电脑一样改变世界。但我们看到的很多人，自己只是机械工程师、电气工程师或者自动控制工程师，开发一个控制机器人的软件系统是遥不可及的事情；又或者自己虽然是软件工程师，但是并不知道如何控制和驱动底层设备。这要怎么办呢？没关系，ROS 机器人操作系统可以帮忙。

ROS 最初是作为科研辅助工具由斯坦福大学开发的。类似的机器人操作系统在世界各国还有很多。有些操作系统面向实时机器人控制，有些操作系统面向机器人仿真，有些操作系统面向用户交互。这些操作系统大部分都相对封闭，各成体系，没有在学术界和产业界造成影响。而由于 ROS 极大的开放性和包容性，它能够兼容其他机器人开发工具、仿真工具和操作系统，使之融为一体。这使得 ROS 不断发展壮大，并成为应用和影响力最广泛的机器人软件平台。

随着 ROS 2.0 的开发，ROS 能够兼容除 Linux 之外更多的操作系统，如 Windows、Android；能够支持从工业计算机到 Arduino 开发板等各类型的硬件；能够采集 RGB-D 摄像头、普通摄像头和各种类型的传感器数据；能够驱动类人形机器人、四轴飞行器等各类型的机器人。而且新版本的 ROS 在采用 SOA 架构的基础上，集成了 MVC 框架，更加有利于机器人人机交互界面的开发与机器人控制。

学习 ROS，本身就是掌握一把通往未来的钥匙。自从本人 2012 年翻译本书第 1 版之后，关于 ROS 的书籍也渐渐多了起来。但和其他书籍相比，本书经过了时间的检验，第 2 版从内容到示例对学习 ROS 都更有帮助。希望大家都能够成为机器人软件设计方面的专家。希望在中国的机器人设计领域能够出现新的领军人物，将中国的机器人带入世界一流行列。

刘品杰，本书第 1 版译者

Foreword | 推荐序二

记得第一次接触 ROS 的时候我还在学校做研究，当时跟一些海外学者交流时得知有这个专为机器人设计的操作系统。我很是兴奋，一心想把 ROS 用在我们最新研发的机器人上，于是就马上动手玩起来。由于当时 ROS 尚在起步阶段，说明文档不太全面，同时社区支持又很少，不知道经过多少折腾才好不容易把它运行起来。体验后发现它的设计框架确实很适合作为机器人敏捷开发工具——算法及控制等代码很容易被复用，避免了很多重复劳动。但奈何当时的功能包不多，而且系统对运算资源要求高，最终没有用在当时的机器人项目上。

由于其开源性以及商用友好的版权协议，ROS 很快得到越来越多的关注及支持。现在 ROS 已有飞快的发展，越来越多机器人相关的软件工具亦加入 ROS 的行列。国外一些商用的机器人也开始支持 ROS 系统，甚至基于 ROS 进行开发。相信这个趋势会一直持续下去并会蔓延到世界各地。而我亦深深体会到国内对 ROS 的关注在近年有显著的上升。

从前在国内学习 ROS 可谓孤军作战，身边没几个人听说过 ROS，而且只能从国外网站学习 ROS 的相关知识，完全没有中文的资料可以查看。幸好国内有不少有心人积极推动 ROS 的发展，不遗余力地对国外 ROS 相关的文章进行翻译，并且发表一些原创的教学文章，丰富 ROS 的中文资源，这使学习 ROS 变得更方便。

我与本书译者通过共同举办 ROS 国内推广课程而结缘。他对推动 ROS 在国内的发展起到了举足轻重的作用，并运营着国内著名的 ROS 交流社区。本书亦是他贡献 ROS 中文社群的作品之一。而本书的作者同样是 ROS 界的权威，有丰富的 ROS 实战经验，使用 ROS 进行过多种机器人的开发。书中从 ROS 的架构概念到常用的调试工具、功能包及传感器的信息处理都有所涉及，是 ROS 入门必看书之一。希望本书能让你快速进入 ROS 的世界，探索 ROS 的精彩。

林天麟，博士，NXROBO 创始人 & CEO

译者序 | The Translators' Words

机器人对于现代人类而言并不陌生和神秘，它在百年前的科幻小说中首次出现，而现在已经逐步进入人类生活的方方面面，机器时代即将到来！

智能机器人的程序究竟是如何设计出来的呢？

智能机器人需要具备强健的“肢”、明亮的“眼”、灵巧的“嘴”以及聪慧的“脑”，这一切的实现实际上涉及诸多技术领域，需要艰辛的设计、开发与调试过程，必然会遇到棘手的问题和挑战。而一个小型的开发团队难以完成机器人各个方面的开发工作，因而需要一套合作开发的框架与模式，以期能够快速集成已有的功能，省却重复劳动的时间。早在 2008 年，我们在与澳大利亚的布劳恩教授交流时，就得知他们开发了一套商业化的“RoBIOS”机器人操作系统，这套系统将一些常用的机器人底层功能进行了封装，可极大简化高级功能的开发。据他们介绍，这是最早的“机器人操作系统”，但由于产品不开源且价格昂贵，我们最终未能一试为快。后来在网络中不断地寻觅，最终发现了 ROS，由于其开源、开放的特性，一下子就引起了我们极大的兴趣。

我们于 2010 年建立了易科机器人 QQ 群进行讨论，从而结识了国内最早期的一些机器人研究者和 ROS 探索者。由于早期相关资料非常匮乏，我们于 2012 年创建了博客（blog.exbot.net）以进行技术分享与交流。易科机器人开发组成员在此期间贡献了大量的教程和开发笔记，在此向他们的无私奉献表示感谢与敬意！近年来，随着机器人的迅猛发展，ROS 得到了更为广泛的使用，国内也出现了一些优秀的项目，包括“星火计划”ROS 公开课（blog.exbot.net/spark）、“HandsFree”ROS 机器人开发平台（wiki.exbot.net）等。

出版界近年来也是硕果累累，本书第 1 版便是国内第一本 ROS 译著，由于实用性强，已经多次重印。第 2 版针对近年来 ROS 的最新发展，对书中部分内容进行了修订，并增加了第 6 章和第 10 章。本书涵盖了使用 ROS 进行机器人编程的最新知识与方法，通过 ROS 编程实践能够帮助你理解机器人系统设计与应用的现实问题。在机器人开发实践中，我们认为除了成功的喜悦外，还看到机器人学目前所处的发展阶段：核心技术尚未成熟、诸多功能尚不完备、bug 多……但我们相信，有了 ROS 的开源精神和完备的合作开发框架，很多问题会

迎刃而解。唯一迫切需要的就是期待你加入机器的开发和研究中来，一起推动开源机器人技术的发展与普及！

本书第2版与第1版的重叠部分主要沿用了刘品杰在第1版中的翻译，个别词汇根据习惯进行了修改。具体来说，第1至5章和第10章由张瑞雷翻译，第6章由张波翻译，第7至9章由刘锦涛翻译，吴中红和李静老师对全书进行了审阅，最后由刘锦涛对全书进行了修改润色和统稿整理。感谢杨维保、马文科等人对本书提出的修改建议！

我们将会 **在 books.exbot.net 发布本书的其他相关资源。**

前言 | Preface

本书第 2 版概括性地介绍了 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行业中。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，以及最终如何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选择一些业余用户负担得起的设备，同时涵盖了在机器人研究中最典型的传感器或执行器。

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。第 2 版在最后增加了一章，讲如何使用“Move it!”包控制机械臂执行抓取任务。读完本书后，你会发现已经可以使用 ROS 机器人进行工作了，并理解其背后的原理。

主要内容

第 1 章介绍安装 ROS 系统最简单的方法，以及如何在不同平台上安装 ROS，本书使用的版本是 ROS Hydro。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，以及在虚拟机和 ARM CPU 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说明如何调试一个节点或利用可视化方法直观地查看通过主题发布的消息。

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将节点间的通

信数据可视化。ROS 提供了一个日志记录 API 来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 `rqt_console` 和 `rqt_graph`），以及可视化接口（如 `rqt_plot` 和 `rviz`）。最后介绍如何使用 `rosviz` 和 `rqt_bag` 记录并回放消息。

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器，如激光雷达、伺服电动机、摄像头、RGB-D 传感器、GPS 等。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 开发板）。

第 5 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里程计软件。

第 6 章将展示如何在 ROS 节点中使用点云库。该章从基本功能入手，如读或写 PCL 数据片段以及发布或订阅这些消息所必需的转换。然后，将在不同节点间创建一个管道来处理 3D 数据，以及使用 PCL 进行缩减采样、过滤和搜索特征点。

第 7 章介绍在 ROS 系统中实现机器人的第一步是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。你也可以仿真摄像头和激光测距传感器，为后续学习如何使用 ROS 的导航功能包集和其他工具奠定基础。

第 8 章是两章关于 ROS 导航功能包集中的第 1 章。该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 9 章延续第 8 章的内容，介绍如何使用导航功能包集使机器人有效地自主导航。该章介绍使用 ROS 的 Gazebo 仿真环境和 `rviz` 创建一个虚拟环境，在其中构建地图、定位机器人并用障碍回避做路径规划。

第 10 章讨论 ROS 中移动机器人机械臂的一个工具包。该章包含安装这个包所需要的文档，以及使用 MoveIt! 操作机械臂进行抓取、放置，简单的运动规划等任务的演示示例。

预备知识

我们写作本书的目的是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 发行版应该都能使用，但还是建议你使用 Ubuntu 12.04 LTS。这样你可以根据第 1 章的内容安装 ROS Hydro。

对于 ROS 的这一版本，你将需要 Ubuntu 14.04 之前的版本，因为之后的版本已经不再

支持 Hydro 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器，包括摄像头和 Arduino 开发板。

你还需要 Git (git-core Debian 软件包)，以便从本书提供的源代码中复制库。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 系统如何帮助完成使机器人真正自主化的任务。对于听说过却从未使用过 ROS 的机器人专业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 ROS 软件框架的很多先进理念和工具。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。

源代码和彩色图片下载

可以从你在 <http://www.packtpub.com> 中的账户下载所有已购买的 Packt Publishing 出版的书籍的示例代码文件。如果你在其他地方购买了这本书，可以访问 <http://www.packtpub.com/support> 并注册，文件会通过电子邮件直接发送给你。还可以从 https://github.com/AaronMR/ROS_Book_Hydro 下载源代码文件。

我们同时提供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 http://www.packtpub.com/sites/default/files/downloads/7580OS_ColorImages.pdf 下载这个文件。

About the Authors | 作者简介

Enrique Fernández 在拉斯帕尔马斯大学获得计算机工程博士学位，目前是 Clearpath Robotics 公司高级机器人工程师。2009 年他完成了关于 SLAM 的硕士学位论文。2013 年他的博士论文解决了自主水下滑翔器 (AUG) 的路径规划问题。那段时间，他还研究了计算机视觉、人工智能以及其他机器人学课题，例如赫罗纳大学的 CIRS/ViCOROB 研究实验室 AUV 的惯性导航系统和视觉 SLAM。他在 2012 年参加了欧洲学生自主水下航行器设计挑战赛 (Student Autonomous Underwater Challenge-Europe, SAUC-E) 并获奖，在 2013 年作为合作者参与了 SAUC-E。

获得博士学位后，Enrique 作为高级机器人工程师在 2013 年 6 月加入 PAL Robotics 公司的自主导航部门。在那里，他开发了用于 REEM、REEM-C 和移动机器人以及相关项目的软件，如使用 ROS 框架的 Stockbot。他的研究方向包括运动规划 (路径规划和移动机器人控制)、机器人定位和 SLAM。在 2015 年，他作为高级自主系统开发人员加盟 Clearpath Robotics 公司的自主系统部门从事 SLAM 相关工作。

在学术方面，Enrique 发表了多篇会议论文，其中两篇于 2011 年发表在《International Conference of Robotics and Automation》(ICRA) 上。他是 Packt Publishing 出版的第 1 版《ROS 机器人程序设计》和其他一些书部分章节的作者。他的硕士学位论文是关于室内机器人的 FastSLAM 算法，此机器人装备了 SICK 激光扫描仪以及 Pioneer 差动平台的轮式里程计。他的博士学位论文是关于 AUG 的路径规划算法和工具。他还拥有电子和嵌入式系统 (如 PC104 和 Arduino) 的开发经验。他的研究背景包括 SLAM、计算机视觉、路径规划、优化、机器人学和人工智能。

我要感谢这本书的合著者，感谢他们为完成这本书所付出的努力以及提供了无数示例的代码。我还要感谢博士学位论文期间大学智能系统和计算工程研究所 (University Institute of Intelligent Systems and Computational Engineering, SIANI) 和 underwater 机器人研究中心 (Center of Underwater Robotics Research, CIRS/ViCOROB) 的研究小组成员。我也要感谢在 PAL 机器

人公司的同事，在那里我学到很多关于 ROS、机器人运动以及仿人双足机器人的知识，不仅有软件，还有电子和硬件设计。最后，我要感谢我的家人和朋友的帮助与支持。

Luis Sánchez Crespo 在拉斯帕尔马斯大学获得了电子与电信工程的双硕士学位。他曾在技术开发和创新研究所 (IDETIC)、加那利群岛海洋平台 (PLOCAN) 和应用微电子研究所 (IUMA) 与不同的研究小组合作，进行超分辨率算法成像研究。

他的专业兴趣包括应用于机器人系统的计算机视觉、信号处理和电子设计。因此，他加入了 AVORA 团队，这批年轻的工程师和学生从零开始从事自主水下航行器 (AUV) 的开发工作。在这个项目中，Luis 开始开发声学 and 计算机视觉系统，用于提取不同传感器的信息，例如水听器、声呐和摄像头。

依托海洋技术的强大背景，Luis 与人合作创办了一家新的初创公司 Subsea Mechatronics，致力于为水下环境开发遥控操作和自主航行器。

下面是海洋技术工程师和企业家 (LPA Fabrika: Gran Canaria Maker Space 的联合创始人和制造商) Dario Sosa Cabrera 对 Luis 的评价：

“他很热情，是一个多学科的工程师。他对工作负责。他可以自我管理，并承担一个团队领导者的责任，如在 SAUC-E 竞赛中领导了 AVORA 团队。他在电子和电信领域的背景让其具备从信号处理和软件到电子设计和制造的广泛专业知识。”

Luis 作为技术审校者参与了 Packt Publishing 出版的第 1 版《ROS 机器人程序设计》的相关工作。

首先，我要感谢 Aaron、Anil 以及 Enrique 邀请我参与编写这本书。和他们一起工作非常快乐。同时，我也要感谢水下机电团队关于重型水下机器人的丰富经验，这些年我们一起成长。我必须提到 LPA Fabrika: Gran Canaria Maker Space，他们备课和讲授教育机器人及技术项目极富热情，与他们共享工作环境也非常令人激动。

最后，我要感谢我的家人和女朋友对我参与的每个项目的大力支持和鼓励。

Anil Mahtani 是一名从事水下机器人工作 5 年的计算机科学家。他第一次在该领域工作是在完成硕士论文期间为低成本 ROV 开发软件架构。在此期间，他也成为 AVORA 的团队领导者和主要开发人员，这个团队的高校学生设计和开发了一个自主水下航行器并参加了 2012 年的 SAUC-E。同年，他完成了论文并获得了拉斯帕尔马斯大学的计算机科学硕士学位。此后不久，他成为 SeeByte 公司的软件工程师，这家公司是水下系统智能软件解决方案的全球领导者。

在 SeeByte 公司工作期间，Anil 参与了军方、石油和天然气公司的一些半自主和自主水下系统的核心开发。在这些项目中，他积极参与自主系统开发、分布式软件体系结构设计和底层软件开发，同时也为前视声呐图像提供计算机视觉解决方案。他还获得了项目经理职位，管理一个开发和维护内核 C++ 库的工程师团队。

他的专业兴趣主要包括软件工程、算法、分布式系统、网络和操作系统。Anil 在机器人方向主要负责提供高效和健壮的软件解决方案，不仅解决当前存在的问题，还预见未来的问题或可能的改进。鉴于他的经验，他在计算机视觉、机器学习和控制问题上也有独特的见解。Anil 对 DIY 和电子学有兴趣，并且开发了一些 Arduino 库回馈社区。

首先，我要感谢我的家人和朋友的支持，他们总是在我最需要的时候帮助我。我也要感谢我的同事和朋友 David Rubio Vidal、Emilio Migueláñez Martín 和 John Brydon 给我最大的支持，他们以专业的方式教我很多知识。我还要感谢我在 SeeByte 和 AVORA 团队的同事，这些年从他们那里学习并经历很多。最后，我要特别感谢 Jorge Cabrera Gámez，他的指导和建议成就了我自己从未想象到的职业生涯。

Aaron Martinez 是数字化制造领域的电脑工程师、企业家和专家。他于 2010 年在拉斯帕尔马斯大学的 Instituto Universitario de Ciencias y Tecnologías Cibernéticas (IUCTC) 完成硕士学位论文。他在远程监控领域使用沉浸式设备和机器人平台准备硕士论文。得到学位后，他参加了在奥地利林茨开普勒大学研究所的机器人学实习计划。在实习期间，他作为团队的一员使用 ROS 和导航包集进行移动平台开发。之后，他参与了有关机器人的项目，其中一个是在拉斯帕尔马斯大学的 AVORA 项目。在这个项目中，他参与自主水下航行器制作，并参与意大利的 SAUC-E。2012 年，他负责维护这个项目；2013 年，他帮助从 ROS 向机器人平台移植导航包集和其他算法。

最近，Aaron 与人共同创立了一家名为 SubSeaMechatronics SL 的公司。这家公司从事与水下机器人和遥控系统相关的项目，还设计和制造水下传感器。公司的主要目标是开发用于研发原型和重型机械手的定制解决方案。

Aaron 有许多领域的经验，比如编程、机器人、机电一体化、数字化制造以及 Arduino、BeagleBone、服务器和激光雷达等设备。如今，他在 SubSeaMechatronics SL 公司从事水下和空中环境的机器人平台设计。

我要感谢我的女朋友，她在我写这本书时给我支持以及给我继续成长的动力。我还要感

谢 Donato Monopoli (加那利群岛技术研究所 (ITC) 生物医学工程部门的主管), 以及 ITC 所有的工作人员, 感谢他们使我懂得数字制造、机械以及组织工程, 我在此度过了生命中最美好的时光。

感谢我大学的同事, 特别是 Alexis Quesada, 他给了我在准备硕士论文时创建我第一个机器人的机会。和他们一起工作, 使我学习到很多关于机器人的知识。

最后, 我要感谢家人和朋友的帮助与支持。

About the Reviewers | 审校者简介

Piotr Gródek 是一位对计算机视觉和图像处理感兴趣的 C++ 程序员。他曾经是嵌入式程序员，现在工作于银行。他是开源游戏和无人驾驶汽车的开发者。空闲时间他喜欢跑步、打壁球和阅读。

Akihiko HONDA 是空间机器人工程师。他于 2012 年在东京工业大学完成了硕士论文。目前他正在东京工业大学进行博士课程学习。

他的研究兴趣包括与空间机器人的远程操作和自动化相互对应的灵活或可变形的材料。他的一个目标是通过开发更好的自动控制系统提高飞船在太空的性能和稳定性。在早期的研究中，他参与了包含大型太阳能电池阵列和用于捕捉国际空间站的空间机械臂的地球观测卫星的工作。目前，他计划将其研究成果应用于空间太阳能电力系统、行星探测车等。他作为最佳参赛者获奖，并在日本天文学会举办的 JSF 卫星设计竞赛中因提出使用其研究的一个新型探索宇宙飞船而获奖。

他在大学的研究过程中，还参与了由日本太空发展署指挥的一些项目。在 JEM (REX-J) 项目的机器人实验中，他参与了在轨道上的实验设备支持操作并在研究中获得灵感。他还参加了一个为宇航员开发可穿戴机械臂的项目并开发了手动控制系统。目前他正工作于两个探索机器人项目。其中一个开发名为“KENAGE”的可变形的探测车，用于克服月球和火星特殊的崎岖地形。这个探测车正在使用 GAZEBO 仿真器进行可行性实验测试。在另一个项目中，他为 JumpingScouter 开发了一个环境识别系统。

2013 年，他参与了萨利大学的 SMART 探测器项目，开发环境保护和识别系统。同时，他也参与了探测车在真实环境中检测实际功能的现场实验。

我要感谢 JAXA 的 Hiroki KATO 给我打开了 ROS 的大门并对我的研究给予宝贵建议。我还要感谢 Mitsushige ODA 教授、Hiroki NAKANISHI 教授以及东京工业大学空间机器人实验室的同事。他们分享了空间机器人的美好前景，提供了建议，并支持我在研究中使用 ROS

实现它们。我也要感谢萨利大学 STAR 实验室的教授和同事给我提供在真实环境中使用 ROS 的重要建议。我特别感谢来自大加那利岛的朋友给我介绍这令人振奋的工作。

最后，我要感谢我的家人 Yoshihiko、Nobuko 和 Ayaka 对我的生活和梦想的支持，同时感谢女朋友对我的理解。

Matthieu Keller 是一位热爱技术和计算机科学的法国工程师。他接受的教育涉及计算和机器人学，这使他成为一名爱好者。他审校了本书的第 1 版。

Aridane J. Sarrionandia de León 研究计算机科学，并对机器人学和自主航行器有非常大的兴趣。他的学位课题是关于使用声呐的水下地图构建，为此他在自主水下航行器中使用 ROS 工作。他有自主系统和 ROS 的经验。他熟悉 OpenCV 和 PCL，目前正在开发自主水面航行器的控制系统。

我要感谢 Luis 和 Aaron 给我这个机会审校这本书。同时，我要感谢拉斯帕尔马斯大学的 AVORA 团队，特别是 Aaron、Luis 和 Enrique，他们给我介绍了 ROS 的神奇之处，并帮助我探索自主航行器世界。感谢我的导师 Jorge Cabrera Gámez，他让我有机会成为 AVORA 团队的一员。

最后，我要感谢我的家人和朋友在我生活中出现问题时支持我，特别要感谢 Eva 纠正我不清晰的语句。

Contents | 目 录

推荐序一

推荐序二

译者序

前言

作者简介

审校者简介

第 1 章 ROS Hydro 系统入门 1

- 1.1 PC 安装教程 3
- 1.2 使用软件库安装 ROS Hydro 3
 - 1.2.1 配置 Ubuntu 软件库 4
 - 1.2.2 添加软件库到 sources.list 文件中 4
 - 1.2.3 设置密钥 5
 - 1.2.4 安装 ROS 5
 - 1.2.5 初始化 rosdep 6
 - 1.2.6 配置环境 6
 - 1.2.7 安装 rosininstall 7
- 1.3 如何安装 VirtualBox 和 Ubuntu 8
 - 1.3.1 下载 VirtualBox 8
 - 1.3.2 创建虚拟机 8

1.4 在 BeagleBone Black 上安装 ROS

- Hydro 11
 - 1.4.1 准备工作 12
 - 1.4.2 配置主机和 source.list 文件 13
 - 1.4.3 设置密钥 14
 - 1.4.4 安装 ROS 功能包 14
 - 1.4.5 初始化 rosdep 15
 - 1.4.6 在 BeagleBone Black 中配置环境 15
 - 1.4.7 在 BeagleBone Black 中安装 rosininstall 15
- 1.5 本章小结 15

第 2 章 ROS 系统架构及概念 16

- 2.1 理解 ROS 文件系统级 16
 - 2.1.1 工作空间 17
 - 2.1.2 功能包 18
 - 2.1.3 综合功能包 19
 - 2.1.4 消息 20
 - 2.1.5 服务 21
- 2.2 理解 ROS 计算图级 22
 - 2.2.1 节点与 nodelet 23
 - 2.2.2 主题 24
 - 2.2.3 服务 25

2.2.4	消息	26	3.1.4	设置 ROS 节点 core 文件 转储	59
2.2.5	消息记录包	26	3.2	日志信息	59
2.2.6	节点管理器	26	3.2.1	输出日志信息	59
2.2.7	参数服务器	27	3.2.2	设置调试信息级别	60
2.3	理解 ROS 开源社区级	27	3.2.3	为特定节点配置调试信息 级别	61
2.4	ROS 系统试用练习	28	3.2.4	信息命名	62
2.4.1	ROS 文件系统导览	28	3.2.5	按条件显示信息与过滤 信息	62
2.4.2	创建工作空间	29	3.2.6	显示信息的方式——单次、 可调、组合	63
2.4.3	创建 ROS 功能包和综合 功能包	30	3.2.7	使用 rqt_console 和 rqt_ logger_level 在运行时修改 调试级别	63
2.4.4	编译 ROS 功能包	30	3.3	检测系统状态	66
2.4.5	使用 ROS 节点	31	3.3.1	检测节点、主题、服务和 参数	67
2.4.6	如何使用主题与节点 交互	33	3.3.2	使用 rqt_graph 在线检测 节点状态图	70
2.4.7	如何使用服务	36	3.4	设置动态参数	71
2.4.8	使用参数服务器	38	3.5	当出现异常状况时使用 roswtf	72
2.4.9	创建节点	38	3.6	可视化节点诊断	74
2.4.10	编译节点	41	3.7	绘制标量数据图	75
2.4.11	创建 msg 和 srv 文件	42	3.8	图像可视化	77
2.4.12	使用新建的 srv 和 msg 文件	44	3.9	3D 可视化	79
2.4.13	启动文件	48	3.9.1	使用 rqt_rviz 在 3D 世界中 实现数据可视化	79
2.4.14	动态参数	50	3.9.2	主题与坐标系的关系	82
2.5	本章小结	54	3.9.3	可视化坐标变换	82
第 3 章	可视化和调试工具	55	3.10	保存与回放数据	83
3.1	调试 ROS 节点	57	3.10.1	什么是消息记录包文件	84
3.1.1	使用 gdb 调试器调试 ROS 节点	57			
3.1.2	ROS 节点启动时调用 gdb 调试器	58			
3.1.3	ROS 节点启动时调用 valgrind 分析节点	59			

3.10.2 使用 rosbag 在消息记录包中记录数据	84	4.7.3 创建使用 Xsens 的示例	116
3.10.3 回放消息记录包文件	85	4.8 使用 10 自由度低成本惯性测量模组 IMU	118
3.10.4 检查消息记录包文件的主题和消息	86	4.8.1 下载加速度传感器库	119
3.11 应用 rqt 与 rqt_gui 插件	88	4.8.2 Arduino Nano 和 10 自由度传感器编程	120
3.12 本章小结	88	4.8.3 创建 ROS 节点以使用 10 自由度传感器数据	121
第 4 章 在 ROS 下使用传感器和执行器	90	4.9 GPS 的使用	123
4.1 使用游戏杆或游戏手柄	90	4.9.1 GPS 如何发送信息	125
4.1.1 joy_node 如何发送游戏杆动作消息	91	4.9.2 创建一个使用 GPS 的工程实例	126
4.1.2 使用游戏杆数据在 turtlesim 中移动海龟	92	4.10 本章小结	127
4.2 使用激光雷达——Hokuyo URG-04lx	95	第 5 章 计算机视觉	128
4.2.1 了解激光雷达如何在 ROS 中发送数据	96	5.1 连接和运行摄像头	129
4.2.2 访问和修改激光雷达数据	98	5.1.1 FireWire IEEE1394 摄像头	129
4.3 使用 Kinect 传感器查看 3D 环境中的对象	100	5.1.2 USB 摄像头	133
4.3.1 如何发送和查看 Kinect 数据	101	5.2 使用 OpenCV 制作 USB 摄像头驱动程序	134
4.3.2 创建使用 Kinect 的示例	102	5.2.1 通过 cv_bridge 使用 OpenCV 处理 ROS 图像	139
4.4 使用伺服电动机——Dynamixel	104	5.2.2 使用 image transport 发布图像	139
4.5 使用 Arduino 添加更多的传感器和执行器	107	5.2.3 在 ROS 中使用 OpenCV	140
4.6 在 Arduino 上使用超声波传感器	111	5.2.4 显示摄像头输入的图像	140
4.7 距离传感器如何发送消息	113	5.3 标定摄像头	141
4.7.1 创建使用超声波的示例	113	5.4 ROS 图像管道	148
4.7.2 Xsens 如何在 ROS 中发送数据	116	5.5 计算机视觉任务中有用的 ROS 功能包	152
		5.6 使用 viso2 实现视觉里程计	153
		5.6.1 摄像头位姿标定	154

5.6.2 运行 viso2 在线演示	157	7.2.5 物理属性和碰撞属性	198
5.6.3 使用低成本双目摄像头 运行 viso2	159	7.3 xacro——一个更好的机器人建模 方法	199
5.7 使用 RGBD 深度摄像头实现 视觉里程计	160	7.3.1 使用常量	199
5.7.1 安装 foveis	160	7.3.2 使用数学方法	200
5.7.2 用 Kinect RGBD 深度 摄像头运行 foveis	160	7.3.3 使用宏	200
5.8 计算两幅图像的单应性	161	7.3.4 使用代码移动机器人	201
5.9 本章小结	162	7.3.5 使用 SketchUp 进行 3D 建模	204
第 6 章 点云	163	7.4 在 ROS 中仿真	205
6.1 理解点云库	163	7.4.1 在 Gazebo 中使用 URDF 3D 模型	206
6.1.1 不同的点云类型	164	7.4.2 在 Gazebo 中添加 传感器	208
6.1.2 PCL 中的算法	164	7.4.3 在 Gazebo 中加载和使用 地图	211
6.1.3 ROS 的 PCL 接口	165	7.4.4 在 Gazebo 中移动机器人	213
6.2 我的第一个 PCL 程序	166	7.5 本章小结	215
6.2.1 创建点云	167	第 8 章 导航功能包集入门	216
6.2.2 加载和保存点云到硬盘	170	8.1 ROS 导航功能包集	216
6.2.3 可视化点云	173	8.2 创建变换	217
6.2.4 滤波和缩减采样	176	8.2.1 创建广播机构	218
6.2.5 配准与匹配	181	8.2.2 创建侦听器	218
6.2.6 点云分区	184	8.2.3 查看坐标变换树	221
6.3 分割	187	8.3 发布传感器信息	221
6.4 本章小结	191	8.4 发布里程数据信息	224
第 7 章 3D 建模与仿真	192	8.4.1 Gazebo 如何获取里程 数据	225
7.1 在 ROS 中自定义机器人的 3D 模型	192	8.4.2 创建自定义里程数据	228
7.2 创建第一个 URDF 文件	192	8.5 创建基础控制器	232
7.2.1 解释文件格式	194	8.5.1 使用 Gazebo 创建里程 数据	233
7.2.2 在 rviz 里查看 3D 模型	195	8.5.2 创建自己的基础控制器	235
7.2.3 加载网格到机器人模型	197		
7.2.4 使机器人模型运动	198		

8.6 使用 ROS 创建地图	237
8.6.1 使用 map_server 保存 地图	238
8.6.2 使用 map_server 加载地图	239
8.7 本章小结	240
第 9 章 导航功能包集进阶	241
9.1 创建功能包	241
9.2 创建机器人配置	241
9.3 配置全局和局部代价地图	243
9.3.1 基本参数的配置	244
9.3.2 全局代价地图的配置	245
9.3.3 局部代价地图的配置	245
9.3.4 基本局部规划器配置	246
9.4 为导航功能包集创建启动 文件	247
9.5 为导航功能包集设置 rviz	248
9.5.1 2D 位姿估计	248
9.5.2 2D 导航目标	249
9.5.3 静态地图	249
9.5.4 粒子云	251
9.5.5 机器人占地空间	251
9.5.6 局部代价地图	252
9.5.7 全局代价地图	252
9.5.8 全局规划	254
9.5.9 局部规划	254
9.5.10 规划器规划	254
9.5.11 当前目标	255
9.6 自适应蒙特卡罗定位	256
9.7 使用 rqt_reconfigure 修改参数	258
9.8 机器人避障	259
9.9 发送目标	260
9.10 本章小结	262

第 10 章 使用 MoveIt!	264
10.1 MoveIt! 体系结构	264
10.1.1 运动规划	265
10.1.2 规划场景	267
10.1.3 运动学	268
10.1.4 碰撞检测	268
10.2 在 MoveIt! 中集成一个机械臂	268
10.2.1 工具箱里有什么	268
10.2.2 使用设置助手生成一个 MoveIt! 包	269
10.2.3 集成到 rviz	273
10.2.4 集成到 Gazebo 或实际 机器人的手臂	276
10.3 简单的运动规划	277
10.3.1 规划单个目标	278
10.3.2 规划一个随机目标	278
10.3.3 规划预定义的群组状态	280
10.3.4 显示目标的运动	280
10.4 考虑碰撞的运动规划	280
10.4.1 将对象添加到规划 场景中	281
10.4.2 从规划的场景中删除 对象	282
10.4.3 应用点云进行运动规划	283
10.5 抓取和放置任务	284
10.5.1 规划的场景	285
10.5.2 感知	288
10.5.3 抓取	288
10.5.4 抓取操作	290
10.5.5 放置操作	292
10.5.6 演示模式	295
10.5.7 在 Gazebo 中仿真	295
10.6 本章小结	296

第 1 章

ROS Hydro 系统入门

欢迎开始阅读本书第 1 章。本章将介绍如何安装 ROS 系统，它是一种新的标准化机器人系统软件框架。本书是基于 ROS Fuerte 的《ROS 机器人程序设计》一书的升级版。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器，并为你的机器人增加新的功能，如自动导航和视觉感知等。得益于开源理念，以及持续开发最先进算法并不断提供新功能的开源社区，ROS 不断进步完善。

通过本书，你将学习到如下内容：

- 在特定版本的 Ubuntu 系统下安装 ROS Hydro 框架
- ROS 的基本操作
- 调试以及数据可视化
- 在 ROS 框架下进行机器人编程
- 连接传感器、执行器和硬件设备以创建机器人
- 创造三维（3D）模型并进行仿真
- 使用导航功能包集使机器人实现自主行驶

本章主要介绍怎样在 Ubuntu 系统中安装完整版本的 ROS Hydro。Ubuntu 不但能够全面支持 ROS，而且是 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。这本书使用的 Ubuntu 版本是 12.04（Precise Pangolin），你可以在 <http://releases.ubuntu.com/12.04/> 免费下载安装。

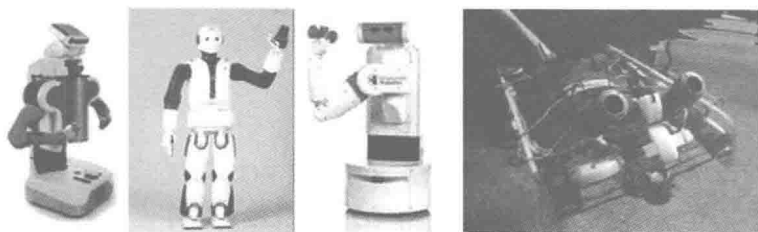
在开始安装之前，我们首先了解一下 ROS 的历史。

Robot Operating System（ROS）是一个得到广泛使用的机器人系统的软件框架。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（Stanford Artificial Intelligence Laboratory, 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，其主要在 Willow Garage 公司支持下与超过 20 多家研究机构联合研发 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。一些完全支持 ROS 系统的平台如下图所示。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。前三个发布代码的机器人例子是人形机器人。最后一个是拉斯帕尔马斯大学开发的水下机器人，代码尚未公布。你可以在 <http://wiki.ros.org/Robots> 找到很多这样的例子。

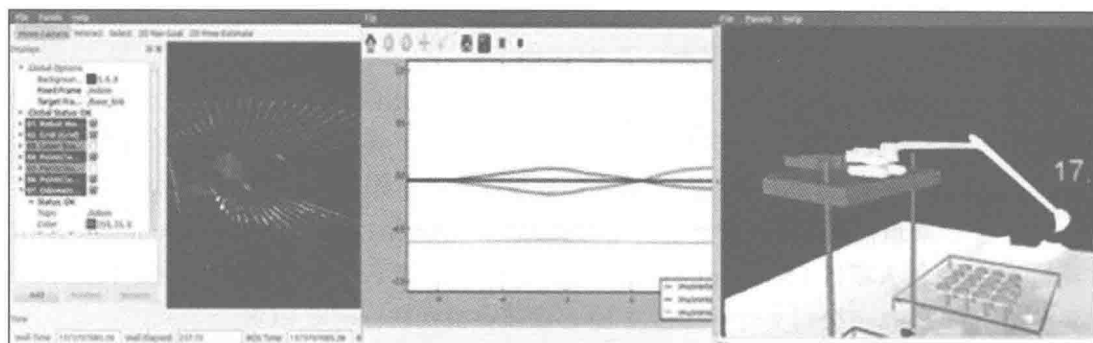


ROS 系统已经支持这些机器人中的传感器和执行器，同时每天 ROS 软件框架支持的设备也在增加。此外，得益于 ROS 和开放硬件，大量公司正在创建更便宜和更强大的传感器。Arduino 开发板是一个很好的例子，使用廉价的电路板可以添加很多类型的传感器如编码器、光和温度传感器等。

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。

它基于一个集中式拓扑的图结构，在节点中接收许多传感器、控制、状态、规划、执行器数据进行计算处理，并发送。它的各种库与功能包都是面向类 UNIX 系统的。

*-ros-pkg 作为一种社区化的软件库使开发高级库更为容易。其中，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库。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 并了解机器人当前的实时状态。其中，可视化工具、仿真环境和调试工具是最重要的几个。在下图中你可以看到两个工具，rviz 和 rqt_plot。中间是 rqt_plot 的截图，你可以看到由传感器数据绘制的曲线。另外两个截图是 rviz；截图中可以看到真实机器人的三维显示。



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。

用 ROS 你可以做更多工作。你可以使用库中的代码,改进后再次共享。这种观念就是开源软件的本质。

ROS 已经发布了多个版本,最新的版本是 Indigo。在本书中,我们使用的版本是 Hydro,因为这个版本更加稳定,而 Indigo 是实验版本,可能存在错误。

下面会介绍如何安装 Hydro 版本的 ROS。即使在本书中我们使用 Hydro,但是在实际工作中,你仍然可能需要安装老版本以便运行一些代码。

如前所述,本书中所使用的操作系统是 Ubuntu,全书的内容及教程将以该系统为基础。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,本章末尾会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/hydro/Installation> 中的指导来完成。

1.1 PC 安装教程

我们假设你已经安装了 Ubuntu 12.04 系统。我们使用这个版本的 Ubuntu,因为它是一个长期支持版本,配备了 Long-Term Support (LTS)。这意味着社区在 5 年内将对这个版本提供维护。

此外,你需要具备一定的 Linux 和命令工具基本知识,例如终端、vim、创建文件夹等。如果需要学习这些工具,你可以在网上找到很多相关的资源,或者你也可以参考这些主题相关的图书。

1.2 使用软件库安装 ROS Hydro

2014 年,ROS 网页更新了设计风格和内容的组织。你可以看到如下的网页截图:



在菜单中，可以找到关于 ROS 的信息以及 ROS 是否适用于你的系统等内容。

ROS 的安装说明可以在开始 (Getting Started) 部分的安装 (Install) 选项卡中找到。

建议你在系统使用软件库而不是源代码安装 ROS，除非你是一个专业用户，想进行自定义安装；在这种情况下，你可能更喜欢使用源代码安装 ROS。

所以我们使用软件库安装 ROS，下面将开始在我们的系统中配置 Ubuntu 软件库。

1.2.1 配置 Ubuntu 软件库

在本节中，你将学习安装 ROS Hydro 的步骤。这个过程基于官方安装页面的内容讲述，链接地址是 <http://wiki.ros.org/hydro/Installation/Ubuntu>。

我们假设你理解 Ubuntu 软件库 (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。

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

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



通常情况下，这些选项都是默认选中的，因此这一步骤你不会遇到什么问题。

1.2.2 添加软件库到 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 Hydro。虽然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。请牢记，Hydro 在 Precise Pangolin (12.04)、Quantal Quetzal (12.10) 和 Raring Ringtail (13.04) 下可以正常工作。

- 在 Ubuntu 12.04 (Precise Pangolin) 下, 安装软件库的具体方法如下:

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

- 在 Ubuntu 12.10 (Quantal Quetzal) 下安装软件库的具体方法如下:

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

- 在 Ubuntu 13.04 (Raring Ringtail) 下安装软件库的具体方法如下:

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

一旦添加了正确的软件库, 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。

1.2.3 设置密钥

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, 当添加完软件库时, 你就已经添加了软件库的密钥, 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 -
```

现在我们可以确定代码来自授权网站并且没有被修改。

1.2.4 安装 ROS

现在准备开始安装 ROS。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。输入以下命令升级该软件:

```
$ 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内容。如果你不知道正在安装的 `rviz`、仿真环境或导航程序是什么, 不用担心, 你将会在后续章节中学习如下内容:

- 最简单的安装方式 (并且是推荐的安装方式, 但你需要足够大的硬盘空间) 就是桌面完整安装 (`desktop-full`)。这将安装 ROS、`rqt` 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、机械臂控制), 以及其他感知库, 如视觉、激光雷达和 RGBD 摄像头 (深度摄像头):

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

- 如果你没有足够的硬盘空间, 或更喜欢安装特定部分的功能包集, 那么第一次安装可

以仅安装桌面安装文件，包括 ROS、rqt 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-hydro-* 功能包集）：

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

- 如果你只是想尝试一下，请安装 ROS-base。ROS-base 通常直接安装在机器人上，尤其是机器人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 的编译和通信包，而没有任何的 GUI 工具。在 BeagleBone Black (BBB) 中，你将使用下面命令：

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

1.2.5 初始化 rosdep

在使用 ROS 之前，必须先安装和初始化 rosdep 命令行工具。这可以使你轻松地安装库和编译源代码时的系统依赖，来运行在 ROS 中的一些核心组件。在 ROS Fuerte 中，你必须安装完 ROS 后再安装 rosdep，因为它是一个独立的工具。现在 rosdep 默认安装在 ROS 中。你可以使用下面的命令安装和初始化 rosdep：

```
$ sudo rosdep init
```

```
$ rosdep update
```

1.2.6 配置环境

恭喜你！能到这一步，说明你已经成功安装了某个版本的 ROS！为了能够运行它，系统需要知道可执行或二进制文件以及其他命令的位置。为了实现以上目的，你需要执行以下脚本：

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

如果你还安装了另一个 ROS 发行版，每次你需要通过调用脚本来使用它，这个脚本会直接配置你的环境。在此我们使用的是 ROS Hydro 的脚本，如果你想尝试其他发行版，只需要用 Fuerte 或 Groovy 代替 Hydro 即可。

如果你在命令行中输入 roscore，那么将看到有程序启动。这是用来测试是否完成 ROS 安装以及是否正确安装最好的方法。

请注意，如果你再次打开一个命令行窗口，并输入 roscore 或其他 ROS 命令，却无法工作了。这是因为你需要再一次执行脚本来配置全局变量和 ROS 的安装路径。

这个问题很容易解决，你只需要在 .bashrc 文件最后添加脚本，当你开始新命令行时，

该脚本将执行并配置环境。

.bashrc 文件在用户的 home 文件夹下 (/home/ 用户名 /.bashrc)。每次用户打开终端，这个文件加载命令行或终端的配置。所以你可以添加命令或配置方便用户使用。出于这个原因，我们将在 .bashrc 文件结束时添加脚本，以避免我们每次打开一个新终端时都要重复输入命令。我们用下面命令：

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

如果要使配置生效，你必须使用下面的命令去执行这个文件，或关闭当前终端，打开另一个新终端。

```
$ source ~/.bashrc
```

一些用户需要在他们的系统中安装不止一个 ROS 的发行版。由于每次调用脚本都会覆盖系统当前配置，所以 ~/.bashrc 只能设置你正在使用的那一个版本的 setup.bash。为了实现在几个发行版之间切换，你需要调用不同的 setup.bash 脚本。

例如，在 .bashrc 文件下面可能有这么几行代码：

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

在这种情况下，ROS Electric 版本将被执行。所以你必须确保将要运行的版本是文件中的最后一个。

如果你想通过终端检查使用的版本，可以非常简单地使用 echo \$ROS_DISTRO 命令。

1.2.7 安装 rosinstall

现在，下一步工作是安装一个命令工具，以帮助我们使用一条命令安装其他包。这个工具是基于 Python 的，但是别担心，使用它不需要你掌握 Python。你在接下来的章节中将学习如何使用这个工具：

运行以下命令在 Ubuntu 中安装这个工具：

```
$ sudo apt-get install python-roscpp
```

这就完成了！你已经在你的系统完成了一个完整的 ROS 系统安装。当我完成一个新安装的 ROS 后，我个人喜欢测试两个东西：roscpp 和 turtlesim。

如果你想做相同的事，在不同命令行分别输入以下命令：

```
$ roscpp
$ roscpprun turtlesim turtlesim_node
```

如果一切正常，你将看到右边的界面：



1.3 如何安装 VirtualBox 和 Ubuntu

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

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。它能帮助我们在计算机上虚拟化新的操作系统，而无需对计算机硬件做任何改动。

后面的章节将展示如何安装 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，能够通过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，以下链接能够在 Windows 系统中下载最新的可用版本：<http://download.virtualbox.org/virtualbox/4.3.12/VirtualBox4.3.12-93733-Win.exe>。

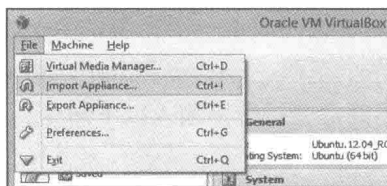
一旦安装完成，你就需要下载 Ubuntu 的镜像文件。在本教程中，我们使用一个已经安装了 ROS Hydro 的 Ubuntu 镜像文件。可以通过以下链接下载它：<http://nootrix.com/2014/04/virtualized-ros-hydro/>。

对于这个版本，Nootrix 团队使用 torrent 下载虚拟机。我尝试了以这种方式下载文件并且效果很好。

你也可以找到预安装了 Ubuntu 和 ROS 的其他虚拟机，但我们还是要使用这个 ROS 官网推荐的版本。

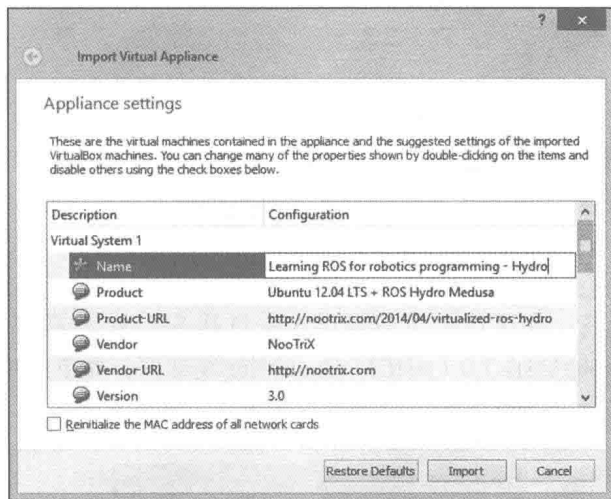
1.3.2 创建虚拟机

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 软件并单击 File | Import Appliance，然后单击 Open appliance 并选择之前下载好的 ROSHydro.ova 文件，如下图所示。



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

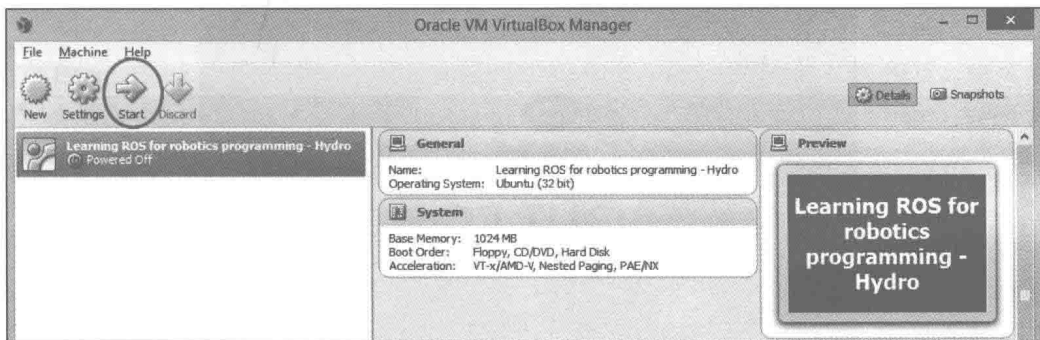
名称。这个名称帮助我们区分不同的虚拟机。我们推荐给它起一个容易理解的名称，在这里我们使用本书的名称，如下图所示。



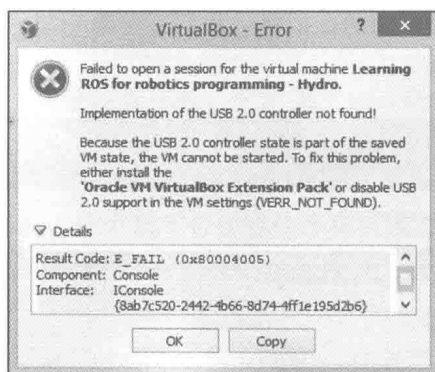
点击 Import 按钮，并在下一个窗口中接受软件授权许可。你将看到一个进度条。这表明 VirtualBox 正在复制虚拟机镜像文件，它在以新的名称创建新的副本。

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。

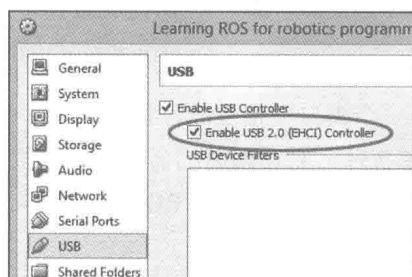
复制过程所需要的时间取决于计算机的执行速度。当它完成时，可以点击 Start 按钮启动虚拟机。需要注意的是，如果你的机器上有多个虚拟机，在启动前应该选择正确的那一个。当然，在这个例子里只有一个虚拟机。



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



为了禁用 USB 支持，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 复选框，如下图所示。重启虚拟机后，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：

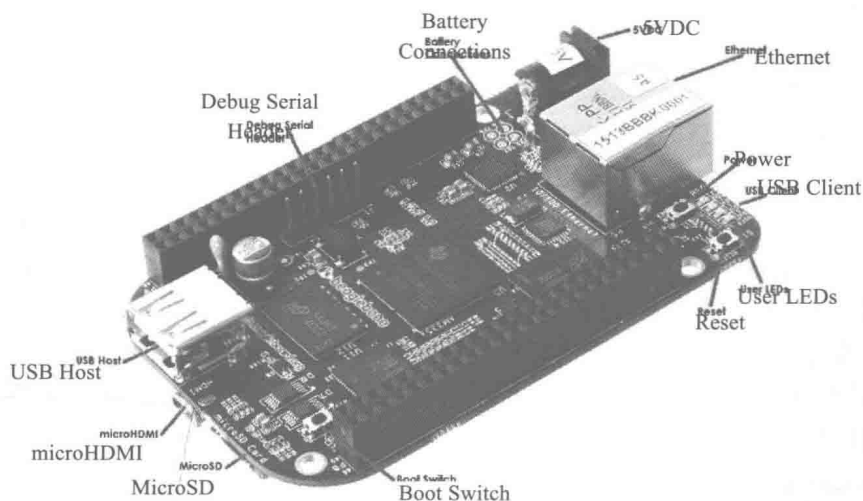


当完成这些步骤后，你就有了一个能够在这本书中使用的完整版本的 ROS Hydro。你可以运行所有的例子和我们将使用的功能包集。遗憾的是，VirtualBox 在使用部分实际外接设备的时候会有问题，并且你可能无法使用这个 ROS Hydro 镜像完成第 4 章中给出的例子。

1.4 在 BeagleBone Black 上安装 ROS Hydro

BeagleBone Black (BBB) 是一种基于 ARM Cortex A8 处理器的低成本开发平台。此开发板是基于 Ångström Linux 发行版制作的。Ångström 由一支希望统一嵌入式系统 Linux 发行版的小型团队开发，他们希望操作系统是稳定且用户友好的。

考虑到社区的开发人员需要一个具有一些通用输入 / 输出 (GPIO) 引脚的机载计算机设备，德州仪器设计了 BeagleBone Black。BeagleBone Black 平台是 BeagleBone 的改进版。开发板的主要特性包括 ARM Cortex A8 处理器 (时钟频率为 1GHz，内存为 512MB)，具有以太网、USB 接口、HDMI、46 引脚 GPIO 接口。这些 GPIO 可以设置为数字 I/O、ADC、脉宽调制，以及 I2C、SPI 或者 UART 等通信协议接口。GPIO 是一种直接将传感器和执行器与 BeagleBone 连接的简单方法。BeagleBone 如下图所示：



在 BeagleBone 开发板刚推出时，无法直接在 Ångström 发行版上安装 ROS。由于这个原因，通常在 BeagleBone 上安装基于 Ubuntu 的操作系统。有不同版本的 Ubuntu ARM 兼容 BeagleBone Black 和 ROS，推荐在运行 ROS 的平台上使用 Ubuntu 13.04 ARM raring armhf 的镜像。

目前已有 Ångström 发行版的 ROS 版本安装文件。安装步骤可以参考网址 <http://wiki.ros.org/hydro/Installation/Angstrom>。除此之外，我们选择在 Ubuntu ARM 上安装 ROS 还因为这个发行版更常用，此外它还可以用于其他基于 ARM 的开发板，如 UDOO Odroid U3、Odroid X2 或 Gumstick。

ARM 技术在智能手机和平板电脑等移动设备领域蓬勃发展。除了增加的 ARM cortex 运算性能，高集成度和低功耗也使这项技术更适合于自主机器人系统开发。在过去的几年里，开发人员已经在市场上推出多款 ARM 平台。其中一些特性类似于 BeagleBone Black、

Raspberry PI 或 Gumstick Overo。此外，更强大的开发板（如具备双核 ARM Cortex A9 的 Gumstick DuoVero 或四核版 Odroid U3、Odroid X2 或 UDOO）也已经上市。

1.4.1 准备工作

在安装 ROS 到 Beaglebone Black 之前，我们需要做一些准备工作。本书的重点是介绍 ROS，我们将列出这些准备工作但不详细介绍。很多关于 Beaglebone Black 和 Ubuntu ARM 的信息可以在网站、论坛和书中找到。

首先，我们必须安装一个与 ROS 兼容的 Ubuntu ARM 发行版。所以需要 Ubuntu ARM 的安装镜像。可以通过下面的命令使用 wget 获得 Ubuntu 13.04 Raring armhf:

```
$ wget https://rcn-ee.net/deb/flasher/raring/BBB-eMMC-flasher-ubuntu-13.04-2013-10-08.img
```

下载 Ubuntu 13.04 armhf 镜像，将其安装到 SD 卡上。可以在 eLinux 网址上得到关于如何安装 Ubuntu 到 Beaglebone Black 的更多细节：http://elinux.org/Beagleboard:Ubuntu_On_BeagleBone_Black#Ubuntu_Raring_On_Micro_SD。

在之前的网页上描述的过程工作良好，但是我们必须注意所使用的 Ubuntu 版本。由于网站会定期更新，因此它们现在使用的是 Ubuntu 14.04，但这可能与我们所使用的 ROS 不兼容。我们将使用之前提及的 Ubuntu 13.04 Raring armhf。

一旦在开发平台中安装好 Ubuntu ARM，就需要配置 Beaglebone Black 的网络接口以实现网络访问。所以，必须配置如 IP、DNS 和网关等网络配置。

记住，在另一个计算机上挂载 SD 卡并编辑 /etc/network/interfaces 可能是最简单的方式。

设置好网络后，为了安装 ROS 中如 CMake、Python 或 Vim 等可能需要的功能包、程序和库，使用以下命令：

```
$ sudo apt-get install cmake python-catkin-pkg python-empy python-nose python-setuptools libgtest-dev build-essential
```

```
$ sudo apt-get install g++ curl pkg-config libv4l-dev libjpeg-dev build-essential libssl-dev vim
```

用于 Beaglebone Black 的操作系统在微型 SD 卡上配置的空间为 1 至 4GB。这个存储空间是非常有限的，如果我们想要使用一个大的 ROS Hydro 功能包，它可能就不够用了。为了解决这个问题，我们可以使用空间更大的 SD 卡，重新分区，扩大文件系统占可用空间的比例。

如果需要使用更大的存储空间，建议扩大 Beaglebone Black 内存文件系统。在网址 http://elinux.org/Beagleboard:Expanding_File_System_Partition_On_A_microSD 可以得到相关内容的进一步介绍。

通过下列命令可以实现上述目的：

1. 需要切换到超级用户模式，输入下面的命令并输入密码：


```
$ sudo su
```

2. 查看 SD 卡的分区信息:

```
$ fdisk /dev/mmcblk0
```

3. 输入 p, 可见 SD 卡的两个分区:

```
$ p
```

4. 之后, 输入 'd' 删除分区, 然后输入 2 指定要删除的分区 /dev/mmcblk0p2:

```
$ d
```

```
$ 2
```

5. 输入 n, 创建一个新分区; 如果输入 p 将创建一个主分区。我们输入 2 指定第二个分区的编号。

```
$ n
```

```
$ p
```

```
$ 2
```

6. 如果没有问题, 输入 w 保存这些操作, 或按 Ctrl + Z 组合键取消更改:

```
$ w
```

7. 完成后重启开发板:

```
$ reboot
```

8. 完成重启后, 再次切换到超级用户模式:

```
$ sudo su
```

9. 最后, 运行下面的命令执行操作系统内存文件系统的扩容。

```
$ resize2fs /dev/mmcblk0p2
```

现在我们准备好安装 ROS 了。安装的过程非常类似于在本章之前介绍过的安装, 主要区别是我们不能安装 ROS full-desktop, 必须单独安装每一个功能包。

1.4.2 配置主机和 source.list 文件

现在开始配置主机:

```
$ sudo update-locale LANG=C LANGUAGE=C LC_ALL=C LC_MESSAGES=POSIX
```

在这之后, 我们将配置源列表, 这基于我们安装在 BeagleBone Black 中的 Ubuntu 版本。兼容 BeagleBone Black 的 Ubuntu 版本数量有限, 目前活跃的发行版是 Ubuntu 13.04 raring armhf, 它也是 Ubuntu ARM 最受欢迎的版本。

- Ubuntu 13.04 Raring armhf:

```
$ sudo sh -c 'echo "deb http://packages.namniart.com/repos/ros  
raring main" > /etc/apt/sources.list.d/ros-latest.list'
```

- Ubuntu 12.10 Quantal armhf:

```
$ sudo sh -c 'echo "deb http://packages.namniart.com/repos/ros  
quantal main" > /etc/apt/sources.list.d/ros-latest.list'
```

- Ubuntu 12.04 Precise armhf:

```
$ sudo sh -c 'echo "deb http://packages.namniart.com/repos/ros  
precise main" > /etc/apt/sources.list.d/ros-latest.list'
```

1.4.3 设置密钥

正如前面所解释的，这一步需要确认源代码是正确的，并且没有人在未经所有者授权的情况下修改过代码或程序：

```
$ wget http://packages.namniart.com/repos/namniart.key -O - | sudo apt-  
key add -
```

1.4.4 安装 ROS 功能包

在安装 ROS 功能包之前，我们必须更新系统以避免出现库依赖的问题。

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

这部分安装与 Beaglebone Black 略有不同。ROS 中有很多库和功能包，并不是全部都能在 ARM 上完整编译。所以不可能实现一个完整的桌面版安装。建议独立安装各功能包，以确保它们能在 ARM 平台上运行。

你可以尝试安装 ROS-base，称为 ROS Bare Bones。ROS-base 会安装包括编译和通信库以及 ROS 功能包，但不包括 GUI 工具：

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

我们可以使用下面的命令来安装指定的 ROS 功能包：

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

如果我们需要查找在 BeagleBone Black 中可用的 ROS 功能包，运行下面的命令：

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

例如，下面的包为 ROS 正常工作的基础，可以使用 apt-get 单独安装：

```
$ sudo apt-get install ros-hydro-ros
```

```
$ sudo apt-get install ros-hydro-roslaunch
```

```
$ sudo apt-get install ros-hydro-rosparam
```

```
$ sudo apt-get install ros-hydro-rosservice
```

虽然从理论上讲，BeagleBone Black 并不支持所有的 ROS 功能包，但实际上我们已经能够在 PC 上开发的整个项目移植到 BeagleBone Black。我们成功尝试了很多功能包，只有安装 rviz 没有实现。

1.4.5 初始化 rosdep

在使用 ROS 之前，必须首先安装并初始化 rosdep 命令行工具，可使你轻松地安装库并解决你准备编译源代码的系统依赖问题，以及提供 ROS 运行需要的一些核心组件。可以使用下面的命令安装并初始化 rosdep：

```
$ sudo apt-get install python-rosdep
$ sudo rosdep init
$ rosdep update
```

1.4.6 在 BeagleBone Black 中配置环境

如果你已经到达这一步，恭喜你，因为你已经在 BeagleBone Black 中成功地安装了 ROS。添加下面的 ROS 环境变量到 bash 中，这样它们就会在每次命令行启动时自动加载：

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

如果在系统中有多个版本的 ROS 我们必须注意。bashrc 的变量必须设置为我们正在使用的版本。

如果我们想要在当前命令行中配置环境，运行命令如下：

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

1.4.7 在 BeagleBone Black 中安装 rosinstall

rosinstall 是 ROS 中一个常见的命令行工具，使安装功能包更方便。如果你要安装它，可以在 Ubuntu 中使用下面的命令行：

```
$ sudo apt-get install python-roinstall
```

1.5 本章小结

在这一章，我们学习了如何在不同 Ubuntu 设备上（计算机、VirtualBox、BeagleBone Black）安装 ROS Hydro。通过这些步骤，你已经在系统上安装了一切必要的软件，可以使用 ROS 开始工作，也可以练习本书中的示例。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库安装，这样做更通用，且一般不会出现任何错误或问题。

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，如传感器和执行器。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、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 系统架构及概念

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，那么下一步要做什么呢？”在本章我们将学习 ROS 系统架构及它的组成。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 (The Filesystem level)
- 计算图级 (The Computation Graph level)
- 开源社区级 (The Community level)

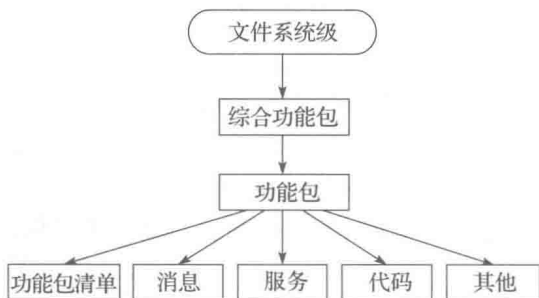
第一级是文件系统级。在这一级，我们会使用一组概念来解释 ROS 的内部构成、文件夹结构，以及工作所需的核心文件。

第二级是计算图级，体现的是进程和系统之间的通信。在相关小节中，我们将学习 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。这个层级非常重要，正是由于开源社区的大力支持，ROS 系统才得以快速成长。

2.1 理解 ROS 文件系统级

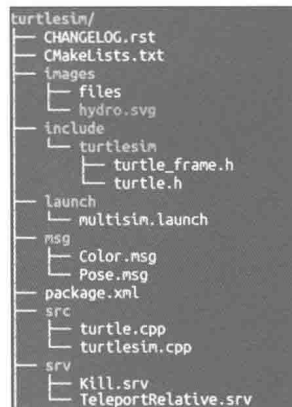
如果你刚接触 ROS，无论是准备使用 ROS 系统还是准备开发 ROS 项目，你都会觉得 ROS 中的各种概念非常奇怪。而一旦你驾轻就熟，那么这些概念就会变得熟悉了。



与其他操作系统类似，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。一个功能包具有用于创建 ROS 程序的最小结构和最少内容。它可以包含 ROS 运行的进程（节点）、配置文件等。
- **功能包清单 (Package 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。一个包的清单由一个名为 `package.xml` 的文件管理。
- **综合功能包 (Metapackage)**：如果你将几个具有某些功能的功能包组织在一起，那么你将获得一个综合功能包。在 ROS Fuerte 中，这种包的组织形式被称为功能包集 (Stack)。为了保持 ROS 简洁，功能包集被移除，现在使用综合功能包实现这个功能。在 ROS 系统中，存在大量不同用途的综合功能包，例如导航功能包集。
- **综合功能包清单 (Metapackage manifest)**：综合功能包清单 (`package.xml`) 类似普通功能包但有一个 XML 格式的导出标记。它在结构上也有一定的限制。
- **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)**：服务描述说明存储在 `my_package/srv/MyServiceType.srv` 中，定义了 ROS 中由每个进程提供的关于服务请求和响应的数据结构。

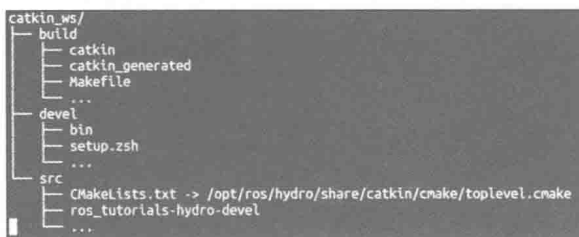
在右侧的截图中，可以看到 `turtlesim` 功能包的内容。你看到的是一系列文件和文件夹，包含代码、图片、启动文件、服务和消息。记住，截图显示文件的一个简短列表，真正的功能包会更多。



2.1.1 工作空间

概言之，工作空间就是一个包含功能包、可编辑源文件或编译包的文件夹。当你想同时编译不同的功能包时非常有用，并且可以用来保存本地开发包。

下图所示的是一个典型的工作空间。每个文件夹都是一个具有不同功能的空间：



- **源文件空间 (The Source space)**: 在源空间 (src 文件夹) 放置了功能包、项目、克隆包等。在这个空间最重要的一个文件是 CMakeLists.txt。当你在工作空间中配置功能包时, src 文件夹 CMakeLists.txt 调用 CMake。这个文件是通过 catkin_init_workspace 命令创建的。
- **编译空间 (The Build space)**: 在 build 文件夹里, CMake 和 catkin 为功能包和项目保存缓存信息、配置和其他中间文件。
- **开发空间 (The Development (devel) space)**: devel 文件夹用来保存编译后的程序, 这些是无需安装就能用来测试的程序。一旦项目通过测试, 你可以安装或导出功能包与其他开发人员分享。

catkin 编译包有两个选项。第一个是使用标准 CMake 工作流程, 通过此方式可以一次编译一个包, 见以下命令:

```
$ cmake packageToBuild/  
$ make
```

如果你想编译所有的包, 可以使用 catkin_make 命令行, 见以下命令:

```
$ cd workspace  
$ catkin_make
```

在 ROS 配置的编译空间目录中, 这两个命令编译可执行文件。

ROS 的另一个有趣的特性是它的覆盖。当你正在使用 ROS 功能包, 例如 Turtlesim, 你可以使用安装版本, 也可以下载源文件并编译它来使用你的修改版本。

ROS 允许使用你自己版本的功能包以替代安装版本。如果你正在安装升级包, 这是非常有用的。或许此时你并不理解它的作用, 但无需担心, 在下一章我们将使用这个功能来创建自己的插件。

2.1.2 功能包

功能包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:

- include/package_name/: 此目录包含了你需要的库的头文件。
- 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/: 这表示服务 (srv) 类型。
- CMakeLists.txt: 这是 CMake 的生成文件。
- package.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工作空间。

- `catkin_create_pkg` 当你想要创建一个新的功能包时，使用此命令。
- `catkin_make` 使用此命令来编译工作空间。
- `rosdep` 此命令安装功能包的系统依赖项。
- `rqt_dep`：这个命令用来查看包的依赖关系图。如果你想看包的依赖关系图，你会在 `rqt` 发现一个称为包图 (package graph) 插件。选择一个包并查看依赖关系。

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 `roscd` 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。
- `roscd` 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。
- `roscd` 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 `package.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此文件，那么这个文件夹很可能是一个功能包或综合功能包。

打开一个 `package.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 `package.xml` 文件中使用的两个典型标记是 `<build_depend>` 和 `<run_depend>`。

`<build_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。

`<run_depend>` 标记显示运行功能包代码所需要的包。右面截图是 `package.xml` 文件的示例。

```
<?xml version="1.0"?>
<package>
  <name>example</name>
  <version>0.0.1</version>
  <description>
    this is a example.
  </description>
  <maintainer email="test@test.com">test</maintainer>
  <license>BSD</license>

  <url type="website">http://www.test.com</url>
  <author>test</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>geometry_msgs</build_depend>
  <run_depend>geometry_msgs</run_depend>
</package>
```

2.1.3 综合功能包

如前所述，综合功能包是一些只有一个文件的特定包，它是 `package.xml`。它不包含其他文件，如代码等。

综合功能包用于引用其他功能特性类似的功能包，例如导航包、`ros_tutorials` 等。

你可以将 ROS Fuerte 的功能包和功能包集转为 Hydro 和 catkin。具体参考 http://wiki.ros.org/catkin/migrating_from_rosbuild。

在右面截图中，你可以看到在 `ros_tutorials` 综合功能包中 `package.xml` 的内容。你可以看到 `<export>` 标记和 `<run_depend>` 标记，这些是功能包清单中必不可少的。

```
<?xml version="1.0"?>
<package>
  ...
  <buildtool_depend>catkin</buildtool_depend>
  ...
  <run_depend>roscpp_tutorials</run_depend>
  <run_depend>rospy_tutorials</run_depend>
  <run_depend>turtlesim</run_depend>
  ...
  <export>
    <metapackage/>
  </export>
  ...
</package>
```

如果你想定位 `ros_tutorials` 综合功能包，可以使用下面的命令：

```
$ rosstack find ros_tutorials
```

显示路径为：`/opt/ros/hydro/share/ros_tutorials`。

查看里面的代码，通过下面命令：

```
$ vim /opt/ros/hydro/share/ros_tutorials/package.xml
```

记住 Hydro 使用综合功能包，不是功能包集，但还是用 `rosstack` 命令用于寻找综合功能包。

2.1.4 消息

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类型，例如 `int32`、`float32`、`string` 或之前创建的新类型，如叫做 `type1` 和 `type2` 的新类型。常量用于定义字段的名称。

一个 `msg` 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	序列化	C++	Python
<code>bool(1)</code>	unsigned 8-bit int	<code>uint8_t(2)</code>	<code>bool</code>
<code>int8</code>	signed 8-bit int	<code>int8_t</code>	<code>int</code>
<code>uint8</code>	unsigned 8-bit int	<code>uint8_t</code>	<code>int(3)</code>
<code>int16</code>	signed 16-bit int	<code>int16_t</code>	<code>int</code>
<code>uint16</code>	unsigned 16-bit int	<code>uint16_t</code>	<code>int</code>
<code>int32</code>	signed 32-bit int	<code>int32_t</code>	<code>int</code>
<code>uint32</code>	unsigned 32-bit int	<code>uint32_t</code>	<code>int</code>
<code>int64</code>	signed 64-bit int	<code>int64_t</code>	<code>long</code>
<code>uint64</code>	unsigned 64-bit int	<code>uint64_t</code>	<code>long</code>
<code>float32</code>	32-bit IEEE float	<code>float</code>	<code>float</code>

(续)

基本类型	序列化	C++	Python
float64	64-bit IEEE float	double	float
string	ascii string (4)	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
```

可以通过下面命令查看消息的结构：

```
$ rosmmsg show std_msgs/Header
```

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 rosmmsg 命令行工具能够输出消息定义信息，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。

2.1.5 服务

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 msg 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。

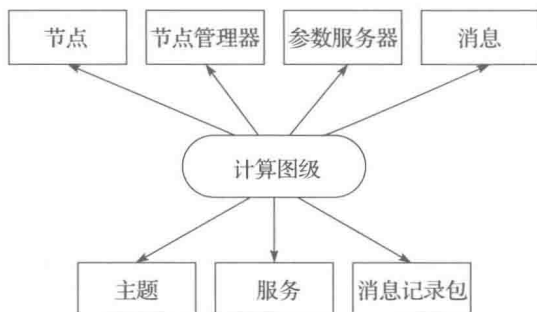
ROS 中有一些执行某些功能与服务的工具。rossrv 工具能输出服务说明、.srv 文件所在的功能包名称，并可以找到使用某一服务类型的源代码文件。

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。

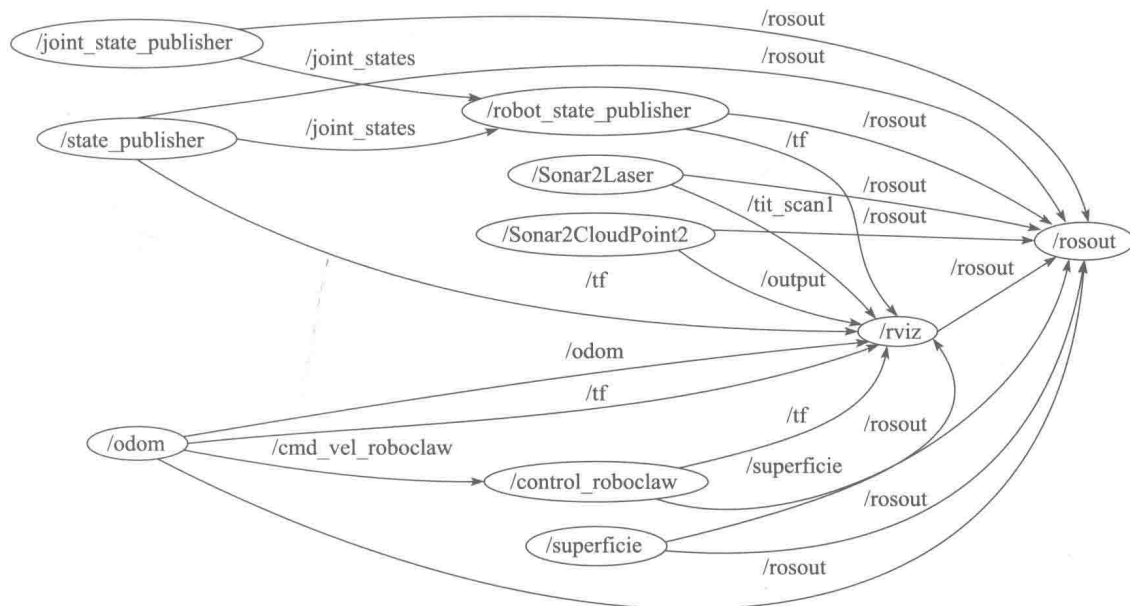


在这一层级中最基本的概念包括节点、节点管理器、参数服务器、消息、服务、主题和消息记录包，这些概念都以不同的方式向计算图级提供数据：

- **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。它也设置节点间的通信。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，在该管理器或其他计算机上运行节点。
- **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信息数据。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）** 每个消息都必须有一个名称来被 ROS 网络路由。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 `rqt_graph`，在第 3 章中将学习到更多相关知识。这些概念在 `ros-comm` 库中实现。



2.2.1 节点与 nodelet

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。

ROS 有另一种类型的节点，称为 `nodelet`。这类特殊节点可以在单个进程中运行多个节

点, 其中每个 `nodelet` 为一个线程 (轻量级进程)。这样, 可以在不使用 ROS 网络的情况下与其他节点通信, 节点通信效率更高, 并避免网络拥塞。`nodelet` 对于摄像头和 3D 传感器这类数据传输量非常大的设备特别有用。

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 `roscpp`。`roscpp` 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。支持的命令如下所示:

- `roscpp info NODE` 输出当前节点信息。
- `roscpp kill NODE` 结束当前运行节点进程或发送给定信号。
- `roscpp list` 列出当前活动节点。
- `roscpp machine hostname` 列出某一特定计算机上运行的节点或列出主机名称。
- `roscpp ping NODE` 测试节点间的连通性。
- `roscpp 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可以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:

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

此命令将主题名称从 `topic1` 改为 `/level1/topic1`。相信你现在还不甚明了, 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:

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

这样参数 (`param`) 就设置为浮点数 9.0。

请记住, 不能使用系统保留的关键字名称, 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` 和 `_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。一个主题可以有多个订阅者也可以有多个发布者，但是你需要用不同的节点发布同样的主题，否则会产生冲突。

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消息类型相匹配，并且节点只能接收类型匹配的消息。节点要想订阅主题，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。所以它最适合于像远程操控的任务。

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` 显示主题的发布频率。
- `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 `rosmg` 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 `rosmmsg` 来获取有关消息的信息。常用参数如下所示：

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

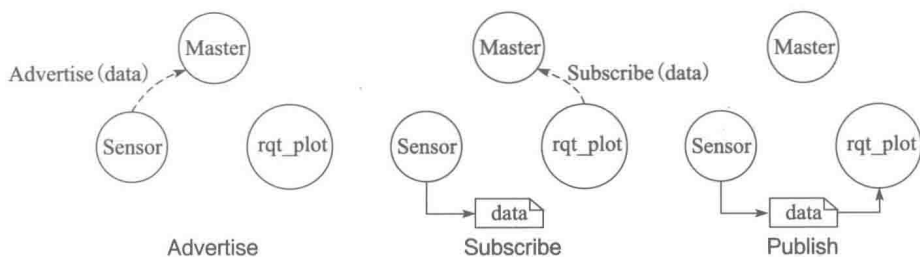
记录包文件可以像实时会话一样在 ROS 中再现情景，在相同时间向主题发送相同的数据。通常情况下，我们可以使用此功能来调试算法。

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。
- `rqt_bag` 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，就能建立点对点的通信。你可以看到以图例显示的 ROS 执行步骤，包括广播一个主题，订阅一个主题，发布一个消息，如下图所示：



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 XMLRPC 实现并在 ROS 节点管理器下运行，这意味着它的 API 可以通过通用的 XMLRPC 库进行访问。XMLRPC 是一个使用 XML 编码并以 HTTP 作为传输机制的远程调用 (Remote Procedure Call, RPC) 协议。

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 (32-bit integer)
- 布尔值 (Boolean)
- 字符串 (String)
- 双精度浮点 (Double)
- ISO 8601 日期 (ISO8601 date)
- 列表 (List)
- 基于 64 位编码的二进制数据 (Base64-encoded binary data)

ROS 中关于参数服务器的工具是 `roscpp`。其支持的参数如下所示：

- `roscpp list` 列出了服务器中的所有参数。
- `roscpp get parameter` 获取参数值。
- `roscpp set parameter value` 设置参数值。
- `roscpp delete parameter` 删除参数。
- `roscpp dump file` 将参数服务器保存到一个文件。
- `roscpp 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是 ROS 资源，其能够通过独立的网络社区分享软件 and 知识。

这些资源包括：

- **发行版 (Distribution)** ROS 发行版是可以独立安装、带有版本号的一系列综合功能包。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维持一致的版本。
- **软件库 (Repository)** ROS 依赖于共享开源代码与软件库的网站或主机服务，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 维基 (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、贡献自己的文件、提供更正或更新、编写教程以及其他行为。
- **Bug 提交系统 (Bug Ticket System)** 如果你发现问题或者想提出一个新功能，ROS 提供这个资源去做这些。
- **邮件列表 (Mailing list)** ROS 用户邮件列表是关于 ROS 的主要交流渠道，能够像论坛一样交流从 ROS 软件更新到 ROS 软件使用中的各种疑问或信息。
- **ROS 问答 (ROS Answer)** 用户可以使用这个资源去提问题。
- **博客 (Blog)** 你可以看到定期更新、照片和新闻，网址是 <http://www.ros.org/news>。

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，你将看到练习包的创建，使用节点、参数服务器以及通过 Turtlesim 移动仿真机器人。

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，我们将使用 `rospack`、`rostack`、`roscd` 和 `rosls` 命令。

我们使用 `rospack` 和 `rostack` 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/hydro/share/turtlesim
```

同样，如果你想要找到你已经在系统中安装过的某个综合功能包，示例如下：

```
$ rostack find ros_comm
```

你将获得到 `ros-comm` 综合功能包的路径，如下：

```
/opt/ros/hydro/share/ros_comm
```

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
```

之前命令的输出如下所示：

```
cmake      images      srv      package.xml  msg
```

如果你想进入某个文件夹，可以使用 `roscd` 命令：

```
$ roscd turtlesim
```

```
$ pwd
```

你将获得以下新路径：

```
/opt/ros/hydro/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
```

你将会看到如下类似信息：

```
/opt/ros/hydro/share:/opt/ros/hydro/stacks
```

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

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

```
$ cd ~/dev/catkin_ws/src
```

```
$ catkin_init_workspace
```

当我们创建工作空间文件夹后，里面并没有功能包，只有 `CMakeList.txt`。下一步是编译工作空间，使用下面命令：

```
$ cd ~/dev/catkin_ws
```

```
$ catkin_make
```

现在，如果你输入 `ls` 命令，可以看到上面命令创建的新文件夹，分别是 `build` 和 `devel` 文件夹。

完成配置，使用下面命令：

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

这一步只是重新加载了 `setup.bash` 文件。如果你关闭并打开一个新的终端，也将得到同样的效果。你应该已经在你的 `~/.bashrc` 文件中加入了该命令行，因为我们在第 1 章用过。如果没有，你可以使用下面命令添加它：

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

2.4.3 创建 ROS 功能包和综合功能包

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 `catkin_create_pkg` 命令行工具。

使用以下命令在之前建立的工作空间创建新的功能包：

```
$ cd ~/dev/catkin_ws/src
$ catkin_create_pkg chapter2_tutorials std_msgs 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` 和 `roscpp`。如以下命令行所示：

```
Catkin_create_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。
- `roscpp` 使用 C++ 实现 ROS 的各种功能。它提供了一个客户端库，让 C++ 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 file chapter2_tutorials/package.xml
Created file chapter2_tutorials/CMakeLists.txt
Created folder chapter2_tutorials/include/chapter2_tutorials
Created folder chapter2_tutorials/src
Successfully created files in /home/aaronmr/dev/catkin_ws/src/chapter2_tutorials.
Please adjust the values in package.xml.
```

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。下面所示是独立使用的：

- `rospack profile` 此命令用于通知新添加的内容
- `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 `catkin_make` 工具：

```
$ cd ~/dev/catkin_ws/

$ catkin_make
```

在几秒之后，你会看到：

```
...
Base path: /home/aaronmr/dev/catkin_ws
Source space: /home/aaronmr/dev/catkin_ws/src
Build space: /home/aaronmr/dev/catkin_ws/build
Devel space: /home/aaronmr/dev/catkin_ws/devel
Install space: /home/aaronmr/dev/catkin_ws/install
...
-- BUILD_SHARED_LIBS is on
-- ~~~~ traversing 29 packages in topological order:
-- ~~~~ - chapter2_tutorials
...
-- +++ processing catkin package: 'chapter2_tutorials'
-- ==> add_subdirectory(chapter2_tutorials)
...
[100%] Built target .....
```

* 如果没有看到错误提示信息，说明功能包编译成功。

记住你必须在 workspace 文件夹运行 `catkin_make` 命令。如果你在其他文件夹这样做，命令无法执行，下面是一个例子：

```
$ roscd chapter2_tutorials/
$ catkin_make
```

当你在 `chapter2_tutorials` 文件夹试图用 `catkin_make` 编译功能包，你会看到如下错误：

```
The specified base path "/home/your_user/dev/catkin_ws/src/chapter2_tutorials" contains a CMakeLists.txt but "catkin_make" must be invoked in the root of workspace
```

当你在 `catkin_ws` 文件夹执行 `catkin_make` 命令，则会编译成功。

2.4.5 使用 ROS 节点

正如我们在 2.2.1 节中解释的，节点都是可执行程序，这些可执行文件位于开发空间中。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-hydro-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 `rostopic` 工具。为了查看命令接受哪些参数，可以输入以下命令：

```
$ rostopic
```

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 machines
  rosnode kill      kill a running node
  rosnode cleanup   purge registration information of unreachable nodes

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 正在运行，我们想要获取正在运行节点的相关信息：

```
$ 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
$ rosnode cleanup

```

现在我们要用 rosrund 命令启动一个新的节点，如下所示：

```
$ rosrund 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，如右图所示：



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。可以看到很多能用于程序调试的信息：

```

$ rosnode info /turtlesim

上一个命令输出如下信息：

Node [/turtlesim]
Publications:
  * /turtle1/color_sensor [turtlesim/Color]
  * /rosout [rosgaph_msgs/Log]
  * /turtle1/pose [turtlesim/Pose]

```

```
Subscriptions:
```

```

* /turtle1/cmd_vel [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://127.0.0.1:43753/ ...
Pid: 32298
Connections:
* topic: /rosout
* to: /rosout
* direction: outbound
* transport: TCPROS

```

在以上信息中，我们可以看到 Publications（及相应主题）、Subscriptions（及相应主题）、该节点具有的 Services（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 如何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 TOPIC` 显示主题所使用的带宽。
- `rostopic echo TOPIC` 将消息输出到屏幕。
- `rostopic find TOPIC` 按照类型查找主题。
- `rostopic hz TOPIC` 显示主题的发布频率。
- `rostopic info TOPIC` 输出活动主题的信息。
- `rostopic list TOPIC` 列出活动主题。
- `rostopic pubs TOPIC` 将数据发布到主题。
- `rostopic type TOPIC` 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海龟，如右图所示：

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，在第一个节点的发布者 (Publications) 部分有一个主题叫 `/turtle1/cmd_vel` [geometry_msgs/Twist]；在第二个节点的订阅者 (Subscriptions) 部分有 `/turtle1/cmd_vel` [geometry_msgs/Twist]：

```
$ rosnode info /teleop_turtle
```

```
Node [/teleop_turtle]
...
Publications:
* /turtle1/cmd_vel [geometry_msgs/Twist]
...
```

```
$ rosnode info /turtlesim
```

```
Node [/turtlesim]
...
Subscriptions:
* /turtle1/cmd_vel [geometry_msgs/Twist]
...
```

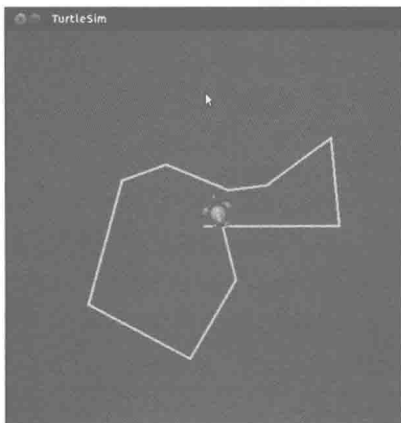
这意味着前面的节点发布了一个主题，而后面的节点可以订阅。你可以使用以下命令查看主题清单：

```
$ rostopic list
```

输出如下：

```
/rosout
/rosout_agg
/turtle1/color_sensor
/turtle1/cmd_vel
/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查看消息产生时发送了哪些数据：



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

你会看到类似下面的显示:

```
---
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 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/cmd_vel
```

你会看到类似如下的显示:

```
Geometry_msgs/Twist
```

如果你想要看到消息字段, 可以使用以下命令:

```
$ rosmmsg show geometry_msgs/Twist
```

你会看到类似如下的显示:

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 zfloat32 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 /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- "linear:
x: 1.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 1.0"
```

你会看到小海龟做曲线运动，如右图所示。

2.4.7 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。

可以使用 `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` 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` 列出活动服务清单。
- `rosservice type/service` 输出服务类型。
- `rosservice uri/service` 输出 ROSRPC URI 服务。

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：

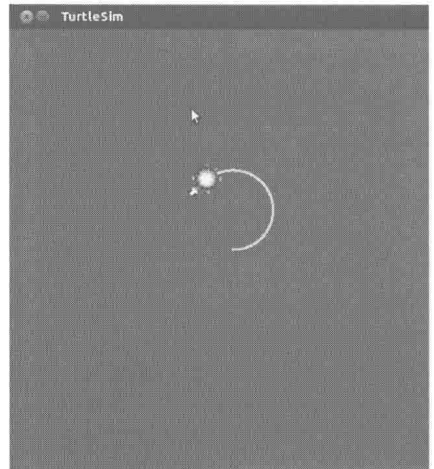
```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

如果你想查看某个服务的类型，例如 `/clear` 服务，请使用：

```
$ rosservice type /clear
```

你会获得类似下面的输出：

```
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，你会看到由小海龟移动产生的线条消失了。

现在我们尝试其他的 service，例如 `/spawn` 服务。这项服务将以不同的方向在另一个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
```

前面的命令和下面的命令是相同的。如果你想知道为什么这些命令相同，可以在搜索引擎里搜索 `piping Linux`：

```
$ rosservice type /spawn
```

你会看到如下类似的结果：

```
turtlesim/Spawn
```

输入下面命令：

```
$ rossrv show turtlesim/Spawn
```

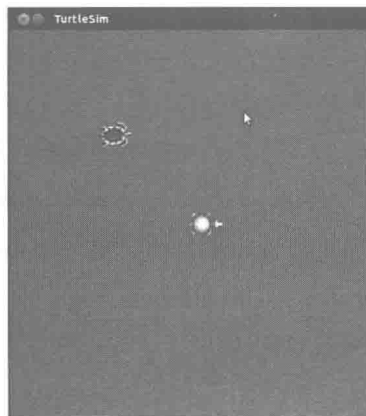
你会看到如下类似的结果：

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

通过这些字段，可以知道如何调用服务。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"
```

我们会获得右侧的结果：



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` 从文件加载参数。
- `rosparam dump file` 将参数保存到一个文件。
- `rosparam delete parameter` 删除参数。
- `rosparam list` 列出了所有参数名。

例如，查看被所有节点使用的服务器参数：

```
$ rosparam list
```

我们会获得以下输出：

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

上面的背景（background）是 `turtlesim` 节点的参数。这些参数可以改变窗口的颜色，窗口初始化为蓝色。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 `dump` 参数。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：一个发布数据，另一个接收这些数据。这是两个节点之间最基本的通信方式，也就是操作数据并使用这些数据来做些工作。

使用以下命令返回 chapter2_tutorials/src/ 文件夹：

```
$ 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命令行窗口中。将下面的代码复制到 example1_a.cpp 文件或者从库中下载它：

```
#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;
}
```

就以上代码做进一步解释。

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

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 包含了我们要使用的消息类型：

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

启动该节点并设置其名称，请记住该名称必须是唯一的：

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

下面是设置节点进程的句柄：

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

将节点设置成发布者，并将所发布主题和类型的名称告知节点管理器。第一个参数是缓

缓冲区的大小，名为 `message`。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::Rate loop_rate(10);
```

在本例中，设置发送数据的频率为 10Hz：

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

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

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

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：

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

这样，消息被发布：

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

如果有一个订阅者出现，ROS 就会更新并读取所有主题：

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

按照 10Hz 的频率将程序挂起。

现在我们创建另一个节点。将下面的代码复制到 `example1_b.cpp` 文件或者从库中下载它：

```
#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解释。

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

首先要包含头文件和主题所使用的消息类型：

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

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：

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

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，处理消息的回调函数为 `messageCallback`：

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

`ros::spin()` 行是节点开始读取主题和在消息到达时，回调函数 `messageCallback` 被调用的循环。当用户按下 `Ctrl + C`，节点会退出消息循环，于是循环结束。

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 `roscd` 工具。这里我们将会用 `Vim` 编辑器下打开这个文件：

```
$ roscd chapter2_tutorials CMakeLists.txt
```

将以下命令行复制到文件的末尾处：

```
include_directories(
    include
    ${catkin_INCLUDE_DIRS}
)

add_executable(chap2_example1_a src/example1_a.cpp)
add_executable(chap2_example1_b src/example1_b.cpp)

add_dependencies(chap2_example1_a chapter2_tutorials_generate_
messages_cpp)
add_dependencies(chap2_example1_b chapter2_tutorials_generate_
messages_cpp)

target_link_libraries(chap2_example1_a ${catkin_LIBRARIES})
target_link_libraries(chap2_example1_b ${catkin_LIBRARIES})
```

现在我们使用 `catkin_make` 工具来编译功能包和全部的节点：

```
$ cd ~/dev/catkin_ws/
$ catkin_make 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 `rostopic list` 命令检查 ROS 是否运行：

```
$ rostopic list
```

然后，在不同的命令行窗口下分别运行两个节点：

```
$ rostopic run chapter2_tutorials example1_a
```

```
$ rostopic run chapter2_tutorials example1_b
```

如果你检查一下 `example1_b` 节点正在运行的命令行窗口，你就会看到以下信息：

```
...
[ INFO] [1403252419.452448698]: I heard: [ I am the example1_a node ]
[ INFO] [1403252419.552163326]: I heard: [ I am the example1_a node ]
[ INFO] [1403252419.653701929]: I heard: [ I am the example1_a node ]
[ INFO] [1403252419.752261663]: I heard: [ I am the example1_a node ]
[ INFO] [1403252419.854459847]: 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 `message` 主题，同时节点 `example2_b` 节点订阅了这个主题。



你可以使用 `rostopic` 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：

```
$ rostopic list
$ rostopic info /example1_a
$ rostopic 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。它们用于说明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` 文件能够被节点使用。第一步，我们先学习 `msg` 文件。

在上一节使用的示例中，我们已经创建了两个具有标准类型 `message` 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 package.xml, 从 `<build_depend>message_generation</build_depend>` 和 `<run_depend>message_runtime</run_depend>` 行删除 `<!-- -->`, 然后编辑 CMakeList.txt, 按下面所示加入 message_generation:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
  message_generation
)
```

找到如下行, 取消注释, 并加入新消息名称:

```
## Generate messages in the 'msg' folder
add_message_files(
  FILES
  chapter2_msg1.msg
)

## Generate added messages and services with any dependencies listed
here
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

现在, 用下面命令进行编译:

```
$ cd ~/dev/catkin_ws/
$ catkin_make
```

检查编译是否成功, 使用下面 rosmmsg 命令:

```
$ rosmmsg 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
```

为了编译新的 msg 和 srv 文件, 必须取消在 package.xml 和 CMakeLists.txt 中的如下行的注释。这些包含消息和服务的配置信息, 并告诉 ROS 如何编译。

首先, 按下面方式从 chapter2_tutorials 功能包中打开 package.xml 文件:

```
$ rosed chapter2_tutorials package.xml
```

找到下面行并取消注释:

```
<build_depend>message_generation</build_depend>
```

```
<run_depend>message_runtime</run_depend>
```

使用下面命令打开 CMakeLists.txt:

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

找到下面行, 取消注释, 并改为正确数据:

```
catkin_package(  
  CATKIN_DEPENDS message_runtime  
)
```

为了生成消息, 你需要在 find_package 部分添加 message_generation 行:

```
find_package(catkin REQUIRED COMPONENTS  
  roscpp  
  std_msgs  
  message_generation  
)
```

在 add_message_files 如下位置添加消息和服务文件的名字:

```
## Generate messages in the 'msg' folder  
add_message_files(  
  FILES  
    chapter2_msg1.msg  
)
```

```
## Generate services in the 'srv' folder  
add_service_files(  
  FILES  
    chapter2_srv1.srv  
)
```

取消 generate_messages 部分的注释, 使得消息和服务可以顺利生成:

```
## Generate added messages and services with any dependencies listed  
here  
generate_messages(  
  DEPENDENCIES  
    std_msgs  
)
```

测试编译是否成功, 使用如下 rossrv 命令:

```
$ 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。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, 新建两个节点并以 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;
}
```

解释一下这些代码:

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

这些行包含必要的头文件和我们创建的 `srv` 文件:

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

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:

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

在这里, 创建服务并在 ROS 中发布广播。

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;
}

```

代码解释:

```

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

```

以 add_3_ints 为名称创建一个服务的客户端。

```

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

```

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。

```

if (client.call(srv))

```

这行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。

为了编译节点，在 CMakeList.txt 文件中增加以下行：

```

add_executable(chap2_example2_a src/example2_a.cpp)
add_executable(chap2_example2_b src/example2_b.cpp)

add_dependencies(chap2_example2_a chapter2_tutorials_generate_
messages_cpp)
add_dependencies(chap2_example2_b chapter2_tutorials_generate_
messages_cpp)

```

```
target_link_libraries(chap2_example2_a ${catkin_LIBRARIES})
target_link_libraries(chap2_example2_b ${catkin_LIBRARIES})
```

现在执行以下命令：

```
$ cd ~/dev/catkin_ws
$ catkin_make
```

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 example1_a.cpp 和 example1_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, "example3_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, "example3_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]
...
```

2.4.13 启动文件

启动 (launch) 文件是 ROS 中一个非常有用的功能, 可以启动多个节点。在之前的小节中, 我们已经学习了创建了节点, 并且在不同的命令行窗口执行, 想象一下, 如果在每一个命令行窗口启动 20 个节点会是多么恐怖的一件事情!

通过启动文件我们可以在命令行窗口方便地实现以上任务, 只需要运行后缀名为 `.launch` 的配置文件来启动多个节点。

为了练习这个例子, 我们在功能包中创建一个新文件夹:

```
$ roscd chapter2_tutorials/
$ mkdir launch
$ cd launch
$ vim chapter2.launch
```

现在, 在 `chapter2.launch` 文件中输入下面的代码:

```
<?xml version="1.0"?>
<launch>
  <node name="example1_a" pkg="chapter2_tutorials"
type="example1_a"/>
  <node name="example1_b" pkg="chapter2_tutorials"
```

```
type="example1_b"/>
</launch>
```

这是个简单的例子，如果需要的话，也可以编写非常复杂的文件。例如，控制一个完整机器人，如 PR2 或 Robonaut，包括真实的机器人和在 ROS 中仿真的机器人。

这个文件包括 launch 启动标签，在标签内部可以看到 node 节点标签。这个节点标签用于从功能包中启动节点，例如从 chapter2_tutorials 包中启动 example1_a 节点。

这个启动文件将执行两个节点，即这章最前的两个例子。如果你还记得，就是节点 example1_a 发送消息到节点 example1_b。你可以通过如下命令启动这个文件：

```
$ roslaunch chapter2_tutorials chapter2.launch
```

你可以看到类似下图的显示：

```
started roslaunch server http://127.0.0.1:40930/

SUMMARY
=====

PARAMETERS
* /roscdistro
* /rosversion

NODES
/
  example1_a (chapter2_tutorials/example1_a)
  example1_b (chapter2_tutorials/example1_b)

auto-starting new master
process[master]: started with pid [19889]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to b334800a-f940-11e3-989f-080027b05884
process[rosout-1]: started with pid [19902]
started core service [/rosout]
process[example1_a-2]: started with pid [19914]
process[example1_b-3]: started with pid [19925]
```

运行的节点在上面的截图中列出，你可以通过下面命令查看运行的节点信息：

```
$ rosnode list
```

可以看到如右图所示的三个节点：

当执行启动文件时，并不需要在 roscore 命令前启动，roslaunch 会启动它。

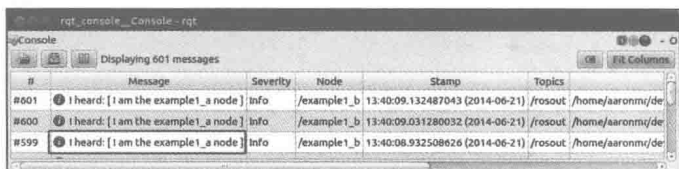
还记得节点 example1_b 会在屏幕输出从其他节点收到的消息，现在却看不到了，这是因为 example1_b 使用了 ROS_INFO 输出消息。当你在命令行只运行一个节点时，可以看到。但是当你运行启动文件时，则看不到。

现在，如果想看到信息，你可以运行 rqt_console 应用。在接下来的章节中，会做详细介绍。运行以下命令：

```
$ rqt_console
```

你从下面的截图中可以看到 example1_b 发送的消息：

```
/example1_a
/example1_b
/rosout
```



在框中，你可以看到节点发送的消息以及来源文件。

2.4.14 动态参数

ROS 的另一个功能是动态重配置应用。通常情况下，当你正在编写一个新节点，你只能以数据初始化节点内的变量。如果你想从外部节点动态地改变这些值，你可以使用参数服务器、服务或主题。如果你使用一个 PID 节点来控制一个电动机，则应该使用动态重配置应用。

在本节中，你将学习如何配置一个包含此功能的基本节点。添加必要的内容到 CMakeLists.txt 和 package.xml 文件。

为了使用动态重配置，你需要写一个配置文件并保存在你功能包的 cfg 文件夹中，创建一个文件夹和新文件如下：

```
$ roscd chapter2_tutorials
```

```
$ mkdir cfg
```

```
$ vim chapter2.cfg
```

在 chapter2.cfg 文件中添加如下代码：

```
#!/usr/bin/env python
PACKAGE = "chapter2_tutorials"

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()

gen.add("double_param", double_t, 0, "A double parameter", .1, 0, 1)
gen.add("str_param", str_t, 0, "A string parameter", "Chapter2_
dynamic_reconfigure")
gen.add("int_param", int_t, 0, "An Integer parameter", 1, 0, 100)
gen.add("bool_param", bool_t, 0, "A Boolean parameter", True)

size_enum = gen.enum([ gen.const("Low", int_t, 0, "Low is 0"),
                        gen.const("Medium", int_t, 1, "Medium is 1"),
                        gen.const("High", int_t, 2, "Hight is 2")],
                      "Select from the list")

gen.add("size", int_t, 0, "Select from the list", 1, 0, 3, edit_
method=size_enum)

exit(gen.generate(PACKAGE, "chapter2_tutorials", "chapter2_"))
```

代码解释:

```
#!/usr/bin/env python
PACKAGE = "chapter2_tutorials"
```

```
from dynamic_reconfigure.parameter_generator_catkin import *
```

以上代码初始化 ROS 并导入参数生成器:

```
gen = ParameterGenerator()
```

以上代码初始化参数生成器, 通过它我们可以开始用下面的代码行添加参数:

```
gen.add("double_param", double_t, 0, "A double parameter", .1, 0, 1)
gen.add("str_param", str_t, 0, "A string parameter", "Chapter2_
dynamic_reconfigure")
gen.add("int_param", int_t, 0, "An Integer parameter", 1, 0, 100)
gen.add("bool_param", bool_t, 0, "A Boolean parameter", True)
```

以上代码加入不同的参数类型并设置默认值、描述、范围等, 参数有如下内容:

```
gen.add(name, type, level, description, default, min, max)
```

- name 参数的名称
- type 参数值的类型
- level 一个传递给回调的位掩码
- description 一个描述参数
- default 节点启动的初始值
- min 参数最小值
- max 参数最大值

参数的名称必须唯一, 参数值必须在最小和最大的范围内:

```
exit(gen.generate(PACKAGE, "chapter2_tutorials", "chapter2_"))
```

最后一行生成必要的文件并退出程序。注意 .cfg 文件是用 Python 写的。本书的实例代码主要使用 C++ 编写, 但有时会使用 Python 代码片段。

因为文件将由 ROS 执行, 所以需要改变文件的权限, 我们将使用 `chmod a+x` 使文件可由任何用户执行和运行, 如下所示:

```
$ chmod a+x cfg/chapter2.cfg
```

打开 CMakeList.txt, 加入下面代码:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
  message_generation
  dynamic_reconfigure
)
```

```
generate_dynamic_reconfigure_options(
```

```

    cfg/chapter2.cfg
)

add_dependencies(example4 chapter2_tutorials_gencfg)

```

现在，我们要写具有动态重配置支持的新节点。在 `src` 文件夹中创建一个新文件如下：

```

$ roscd chapter2_tutorials
$ vim src/example4.cpp

```

在文件中写入如下代码段：

```

#include <ros/ros.h>
#include <dynamic_reconfigure/server.h>
#include <chapter2_tutorials/chapter2Config.h>

void callback(chapter2_tutorials::chapter2Config &config, uint32_t
level) {
    ROS_INFO("Reconfigure Request: %d %f %s %s %d",
        config.int_param,
        config.double_param,
        config.str_param.c_str(),
        config.bool_param?"True":"False",
        config.size);
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "example4_dynamic_reconfigure");

    dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>
server;
    dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>::Ca
llbackType f;

    f = boost::bind(&callback, _1, _2);
    server.setCallback(f);

    ros::spin();
    return 0;
}

```

这里进行代码解释，注意一些重要的行：

```

#include <ros/ros.h>
#include <dynamic_reconfigure/server.h>
#include <chapter2_tutorials/chapter2Config.h>

```

这些行包括 ROS 头文件、参数服务器以及我们先前创建的 `config` 文件。

```

void callback(chapter2_tutorials::chapter2Config &config, uint32_t
level) {
    ROS_INFO("Reconfigure Request: %d %f %s %s %d",
        config.int_param,
        config.double_param,

```



```

    config.str_param.c_str(),
    config.bool_param?"True":"False",
    config.size);
}

```

回调函数将输出参数的新值。这是参数访问的方式，例如 `config.int_param`。参数的名称必须与 `example2.cfg` 配置文件相同：

```

dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>
server;

```

初始化服务器，忽略 `chapter2_Config` 配置文件：

```

dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>::Call
backType f;

```

```

f = boost::bind(&callback, _1, _2);
server.setCallback(f);

```

现在，我们向服务器发送回调函数。当服务器得到重新配置请求，它会调用回调函数。一旦我们完成以上讲解的步骤，我们需要在 `CMakeLists.txt` 文件中添加如下代码：

```

add_executable(chap2_example4 src/example4.cpp)

add_dependencies(chap2_example4 chapter2_tutorials_gencfg)

target_link_libraries(chap2_example4 ${catkin_LIBRARIES})

```

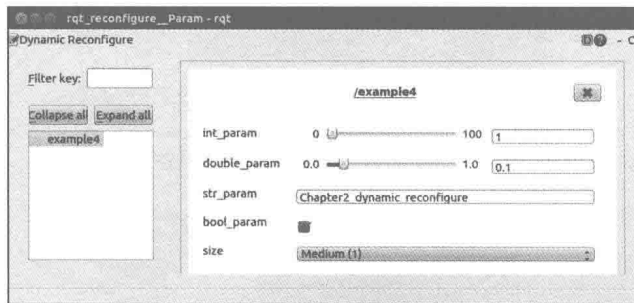
现在，你必须编译并运行节点和动态重配置界面（Dynamic Reconfigure GUI），如下：

```

$ roscore
$ rosrunc chapter2_tutorials example4
$ rosrunc rqt_reconfigure rqt_reconfigure

```

在你执行最后一行命令后，你会看到一个新窗口，通过它你可以动态重配置节点的参数，如下图所示：



每当你通过滑块、文本框等调整参数时，你可以在命令行看到正在运行的节点的这些改变，如下图所示：

```
[ INFO] [1403367196.752115948]: Reconfigure Request: 20 0.800000 qwert True 1
[ INFO] [1403367196.942722848]: Reconfigure Request: 20 0.800000 qwerty True 1
[ INFO] [1403367196.973132691]: Reconfigure Request: 20 0.800000 qwerty True 1
[ INFO] [1403367197.183714401]: Reconfigure Request: 20 0.800000 qwertyu True 1
[ INFO] [1403367197.217819018]: Reconfigure Request: 20 0.800000 qwertyu True 1
[ INFO] [1403367203.160337570]: Reconfigure Request: 1 0.800000 qwertyu True 1
[ INFO] [1403367203.188864110]: Reconfigure Request: 1 0.800000 qwertyu True 1
```

通过动态重配置，你可以更高效地开发和测试节点。与硬件一起配合使用这个程序是不错的选择。你将在接下来的章节中学习到更多内容。

2.5 本章小结

本章介绍了 ROS 系统的架构及其工作方式的基本信息。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，你会逐渐理解这样的应用。

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。

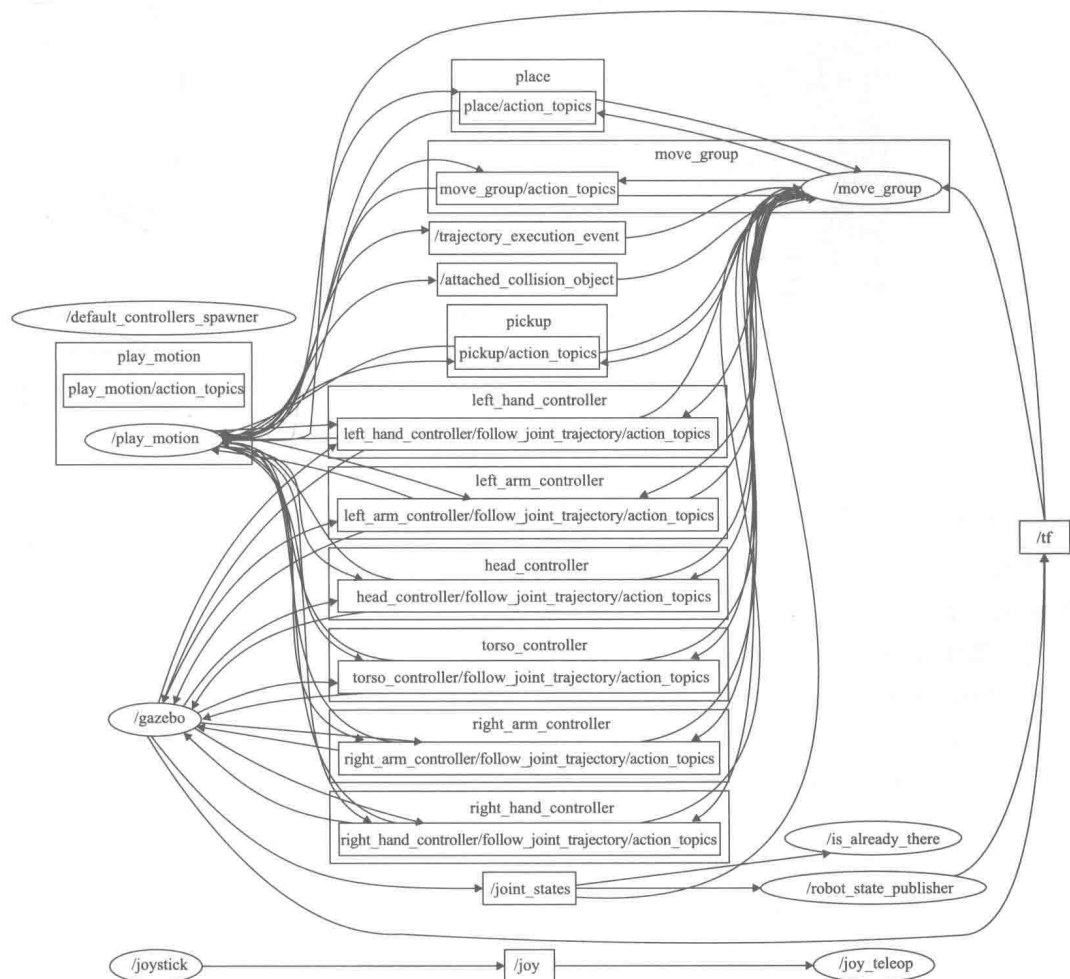
请注意如果想查询某个名词或功能的解释，且无法在这本书中找到相关内容或答案，那么可以通过以下链接访问 ROS 官方资源 <http://www.ros.org>。而且你还可以通过访问 ROS 社区 <http://answers.ros.org> 提出自己的问题。

在下一章中，将学习如何使用 ROS 工具调试和可视化数据。这些将帮助你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

可视化和调试工具

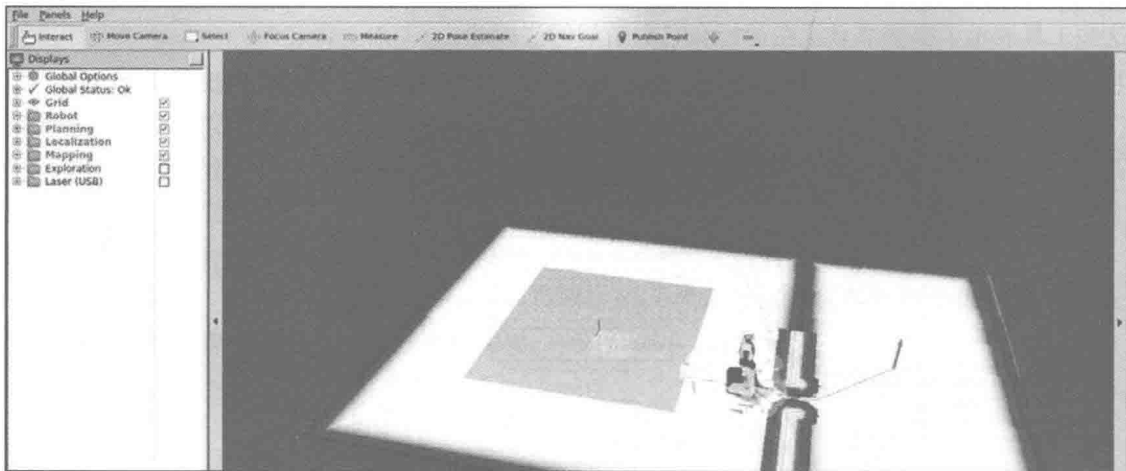
ROS 附带了大量功能强大的工具，帮助用户和开发人员可视化和调试代码，以便检测并解决软硬件问题。其中包括消息日志系统（类似 log4cxx）、诊断消息、可视化以及检测工具。这些工具展示了哪些节点正在运行和它们是如何连接的。



本章我们还会展示如何用 GDB 调试器调试 ROS 节点，介绍用于日志记录的 API，以及如何设置日志记录级别。接着，我们将解释如何用 ROS 工具集检测哪些进程正在运行以及它们之间通信的内容。例如，在下图所示的系统可视化图中可以看到正在运行的节点以及用连线表示的数据流。这个工具是 `rqt_graph`，这里显示的是 REEM 机器人在 Gazebo 仿真中运行的节点和主题。

从图中我们可以看到多个关于机械臂、肢体和头部的控制器、MoveIt! 的 `move_group` 节点、抓取和放置操作服务以及 `play_motion` 节点的预存储动作。其他节点发布 `joint_states`、大量机器人控制器以及移动底盘的手柄控制等信息。

同样，本章会介绍标量数据的时序绘图工具，视频流数据的可视化图，以及用于不同类型数据的 3D 可视化工具 `rviz`（或 `rqt_rviz`）等，如下图所示：



可以使用下面命令运行上图的 REEM 机器人仿真：

```
$ roslaunch reem_2dnav_gazebo reem_navigation.launch
```

注意，在开始安装此仿真前，请认真阅读下面网页的使用说明：<http://wiki.ros.org/Robots/REEM>。

在本节中，将介绍以下内容：

- 如何在 ROS 中调试和优化代码。
- 介绍在代码中添加消息日志并设置不同的级别、命名、特定条件和流选项。这里我们将解释 `rqt_logger_level` 和 `rqt_console` 接口，它们可以分别设置节点错误级别和消息可视化。
- 介绍如何通过列表来检测 ROS 系统状态，包括运行的节点、主题、服务和它们之间传递信息的行为以及 ROS 节点管理器中声明的参数等。我们将介绍以有向图形式显示主题和节点的 `rqt_graph`，和可以用来修改动态参数的 `rqt_reconfigure`。

- 讲解如何使用 `runtime_monitor` 和 `robot_monitor` 接口可视化诊断信息。
- 讲解如何使用 `rqt_plot` 绘制特定消息的标量数据。对于非标量数据，我们将讲解 ROS 中的其他 `rqt` 工具，例如用 `rqt_image_view` 显示图像以及用 `rqt_rviz` 以 3D 形式显示多维数据。还包括如何可视化标记和交互式标记。
- 介绍坐标系以及如何将它们集成到 ROS 消息和可视化工具之中。以及如何使用 `rqt_tf_tree` 来可视化转换坐标系树 Transform Frame (tf)。
- 讲解如何保存主题发送的消息，以及如何重播它们用于仿真或测试目的，并介绍 `rqt_bag` 接口。
- 最后，将介绍 `rqt_gui` 接口，以及如何在 GUI 窗口中排列显示它们。

大部分 `rqt` 工具可以在终端输入名称运行，例如 `rqt_console`，但有时不行，必须使用 `roslaunch rqt_reconfigure rqt_reconfigure`，注意名字虽然是重复的，但其实前一个是功能包的名称，后一个是节点名称。

3.1 调试 ROS 节点

ROS 节点可以像正常程序一样调试。调试程序在系统中运行时有一个进程号 (PID)。你可以用任何标准工具（如 `gdb`）进行调试。同样可以用 `valgrind` 检查内存泄漏，或者用 `callgrind` 分析算法性能。请记住使用下面的命令运行一个节点：

```
$ roslaunch chapter3_tutorials example1
```

很遗憾，你不能通过下面的方式启动命令：

```
$ gdb roslaunch chapter3_tutorials example1
```

在接下来的部分中，我们将解释如何调用这些工具调试 ROS 节点，以及如何在代码中添加日志消息，让问题诊断更简单。这样即使没有二进制调试文件，也可以诊断基本问题。然后，我们将讨论 ROS 自检工具，测试节点间损坏的连接。因此，这章的概述是自下而上，实际的问题诊断方式是自上向下。

3.1.1 使用 `gdb` 调试器调试 ROS 节点

为了使用 `gdb` 调试器调试一个 C/C++ 节点，唯一要知道的是可执行节点的路径。在 ROS hydro 和 catkin 功能包中，节点的可执行文件在工作空间的 `devel/lib/<package>` 文件下。例如，为了在 `gdb` 中运行 `chapter3_tutorials` 功能包中的 `example1` 节点，我们需要按如下步骤进行，首先到工作空间文件夹下（`~/dev/catkin_ws`）：

```
$ cd devel/lib/chapter3_tutorials
```

如果已经运行过 `catkin_make install`，你也使用下面命令导航到 `install/lib/chapter3_tutorials` 文件夹下：

```
$ cd install/lib/chapter3_tutorials
```

现在可以使用 gdb 命令运行节点：

```
$ gdb example1
```



记住，必须在启动节点之前保证 roscore 运行，因为节点需要管理器 / 服务器运行。

一旦 roscore 在运行，你可以通过点击 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，在运行节点后就能在 gdb 终端看到下面的输出：

```
(gdb) r
Starting program: ~/dev/catkin_ws/devel/lib/chapter3_tutorials /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)]
[New Thread 0x7ffff1e63700 (LWP 3205)]
[New Thread 0x7ffff1662700 (LWP 3206)]
[New Thread 0x7ffff0e61700 (LWP 3211)]
[DEBUG] [1356342615.325647326]: This is a simple DEBUG message!
[DEBUG] [1356342615.326124607]: This is a DEBUG message with an argument:
3.140000
[DEBUG] [1356342615.326254667]: This is DEBUG stream message with an
argument: 3.14
[Thread 0x7ffff0e61700 (LWP 3211) exited]
[Thread 0x7ffff1662700 (LWP 3206) exited]
[Thread 0x7ffff2664700 (LWP 3204) exited]
[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

我们需要一个启动文件 (launch) 去启动节点，如下：

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

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

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

类似地，也可以添加 `output="screen"`，使节点在终端显示。这个启动前缀会创建一个调用 `gdb` 节点的新 `xterm` 终端。依据需要设置断点，按 `C` 键或 `R` 键启动节点并调试。这在节点崩溃时，可以得到回溯（`backtrace`，`bt`）。

3.1.3 ROS 节点启动时调用 `valgrind` 分析节点

此外，我们可以使用相同的属性把节点附加到诊断工具上。例如，可以启动 `valgrind` 来检测我们程序的内存泄漏情况，并执行性能分析。你可以访问 <http://valgrind.org> 获取详细信息。与调用 `gdb` 的方式相反，现在我们无需重新启动 `xterm`，只需如下设置：

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

3.1.4 设置 ROS 节点 `core` 文件转储

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` 文件转储（`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，当前值可以通过 `ulimit -a` 查看。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` 文件转储，必须将 `core` 文件名设置为默认使用的进程 `pid`，否则无法创建，因为在 `$ROS_HOME` 已有一个 `core` 目录会防止 `core` 文件转储。因此，为了创建 `core` 文件转储的名称和路径为 `$ROS_HOME/core.PID`，必须运行如下命令：

```
$ echo 1 | sudo tee /proc/sys/kernel/core_uses_pid
```

3.2 日志信息

通过信息记录显示程序的运行状态是好的习惯，但需要确定这样做不会影响软件的运行效率和输出的清晰度。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。如果它们被当前冗长级别掩盖（甚至在编译时），它们对性能没有影响。它们与 ROS 其他工具完全集成来可视化或过滤来自所有运行节点的消息。

3.2.1 输出日志信息

ROS 自带了大量的能够输出日志信息的函数和宏。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。从简单的开始，用 C++ 代码输出一个消息信息：

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

为了获取日志记录的函数和宏，这个头文件足够了：

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

这包括了以下头文件：

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

前面的消息处理程序运行的结果如下所示：

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

所有输出的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），这两个值放在实际信息之前的方括号中。时间戳以公历时间计时，代表着自 1970 年 1 月 1 日以来的秒和纳秒计数。于是我们在新一行输出了信息。

此函数允许以和 C 语言中的 `printf` 函数相同的方式增加参数。例如，可以按照下面代码输出变量 `val` 对应的浮点数值：

```
float val = 1.23;  
ROS_INFO("My INFO message with argument: %f", val);
```

此外，C++ STL 流被 `*_STREAM` 函数支持。因此，前面的指令相当于使用流：

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

请注意，我们没有指定任何流，因为 API 负责这些，通过重定向到 `cout/cerr`、一个文件或两者。

3.2.2 设置调试信息级别

ROS 有五个日志记录标准级别，按照顺序排列分别是：

- DEBUG 调试
- INFO 信息
- WARN 警告
- ERROR 错误
- FATAL 致命

这些名称是输出信息的函数的一部分，它们遵循以下语法：

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

每一种信息都会以特定的颜色在屏幕上输出。这些颜色分别是：

```
DEBUG in green  绿色  
INFO in white   白色  
WARN in yellow  黄色  
ERROR in red    红色  
FATAL in purple 紫色
```


每个消息级别用于不同的目的。在这里，我们建议：

- DEBUG：只有调试时有用
- INFO：说明重要步骤或节点在执行操作
- WARN：提醒你一些错误、缺失或者不正常
- ERROR：提示错误，尽管节点仍然可以运行
- FATAL：通常防止节点继续运行

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 `rqt_console` 和 `rqt_logger_level` 来实现这一功能。

在编译源代码时可以设置日志级别，但不推荐这么做，这需要我们修改源代码定制日志级别，如果你想这样做，请参考《ROS 机器人程序设计》(第 1 版)。

然而，在一些时候，我们需要删除低于设定级别的日志。这时，我们希望看到那些消息后，将它们删除而不是禁用。为此需要将 `ROSCONSOLE_MIN_SEVERITY` 设置为期望的最低严重级别或者避免任何消息（甚至是 FATAL）。宏如下：

```
ROSCONSOLE_SEVERITY_DEBUG
ROSCONSOLE_SEVERITY_INFO
ROSCONSOLE_SEVERITY_WARN
ROSCONSOLE_SEVERITY_ERROR
ROSCONSOLE_SEVERITY_FATAL
ROSCONSOLE_SEVERITY_NONE
```

`ROSCONSOLE_MIN_SEVERITY` 宏在 `<ros/console.h>` 中默认定义为 DEBUG 级别。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，若想仅显示 ERROR 或更高级别的调试信息，在源代码中加入下面代码：

```
#define ROSCONSOLE_MIN_SEVERITY ROSCONSOLE_SEVERITY_ERROR
```

或者，在 `CMakeLists.txt` 中使用下面代码设置包中所有节点的宏：

```
add_definitions(-DROSCONSOLE_MIN_SEVERITY=ROSCONSOLE_SEVERITY_ERROR)
```

除此之外，还有一个更灵活的方法就是在配置文件中设置最低日志级别。用文件创建一个名为 `config` 文件夹和名为 `chapter3_tutorials.config` 的文件，文件内容如下（从它设置为 DEBUG 级别开始编辑给定文件）：

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

然后，我们设置 `ROSCONSOLE_CONFIG_FILE` 环境变量指向我们的文件。我们可以使用一个启动（launch）文件来替代配置环境变量，但这样做会直接运行节点。因此，我们可以

通过 env (环境变量) 字段扩展 launch 文件, 如下所示:

```
<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>
```

如上所述, 环境变量会找到之前显示的配置文件, 其中包含每个已命名日志的日志级别说明。在这个例子中是功能包名称。

3.2.4 信息命名

默认情况, ROS 分配一些名字给节点记录器。目前讨论过的消息在节点名字后命名。对于复杂的节点, 我们可以为一个给定的模块或功能的消息提供一个名字。ROS_<LEVEL>[_STREAM]_NAMED 函数, 如下面代码所示 (以 example2 节点为例):

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

通过命名的消息, 我们可以使用配置文件为每个命名的消息设置不同的初始日志级别, 之后可以单独修改它们。在命名规范上我们必须使用消息的名称作为功能包的子包。例如可以设定 named_msg 的信息, 如下列代码所示:

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

3.2.5 按条件显示信息与过滤信息

按条件显示 (conditional) 信息是指仅当满足给定的条件时才输出的信息。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。下面是以 example2 节点为例的代码:

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

过滤 (Filtered) 信息在本质上与按条件显示信息类似, 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;
};

MyLowerFilter filter_lower( val );

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

3.2.6 显示信息的方式——单次、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。

```
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，只会显示信息 1 次。

然而，有时候在迭代中以一定频率显示信息更好。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，但是需要将函数名中的 `ONCE` 替换成 `THROTTLE`，函数会将 `period` 作为第一个参数，也就是说，它将会间隔每个 `period` 秒后输出：

```
for( int i = 0; i < 10; ++i ) {
    ROS_INFO_STREAM_THROTTLE(2,
        "My throttle 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提示信息区别开。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记录器。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调试级别，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 `rqt_console` 和 `rqt_logger_level` 在运行时修改调试级别

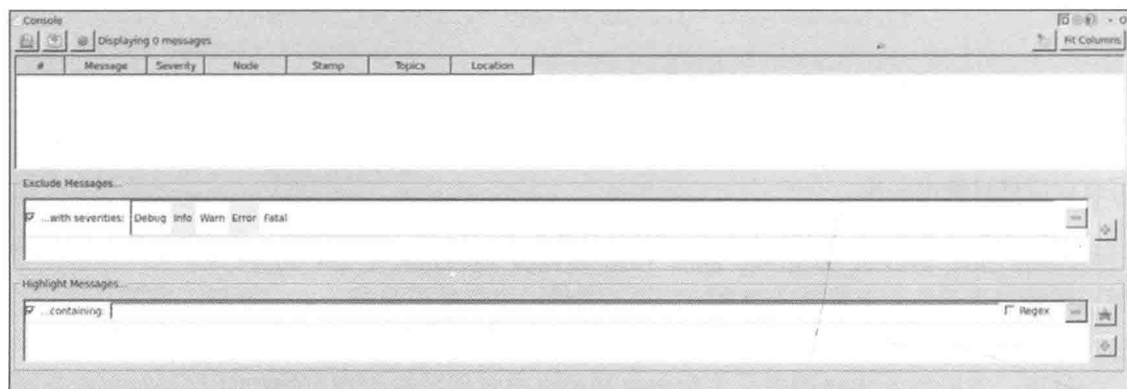
ROS 中提供了一系列工具去管理日志信息。在 ROS hydro 中，我们有两个独立的 GUI：

rqt_logger_level 设置节点或者指定日志记录器的日志记录级别；rqt_console 对日志信息进行可视化、过滤和分析。

使用代码示例 example3 测试这个功能。运行 roscore 和 rqt_console 来看日志信息：

```
$ rosrund rqt_console rqt_console
```

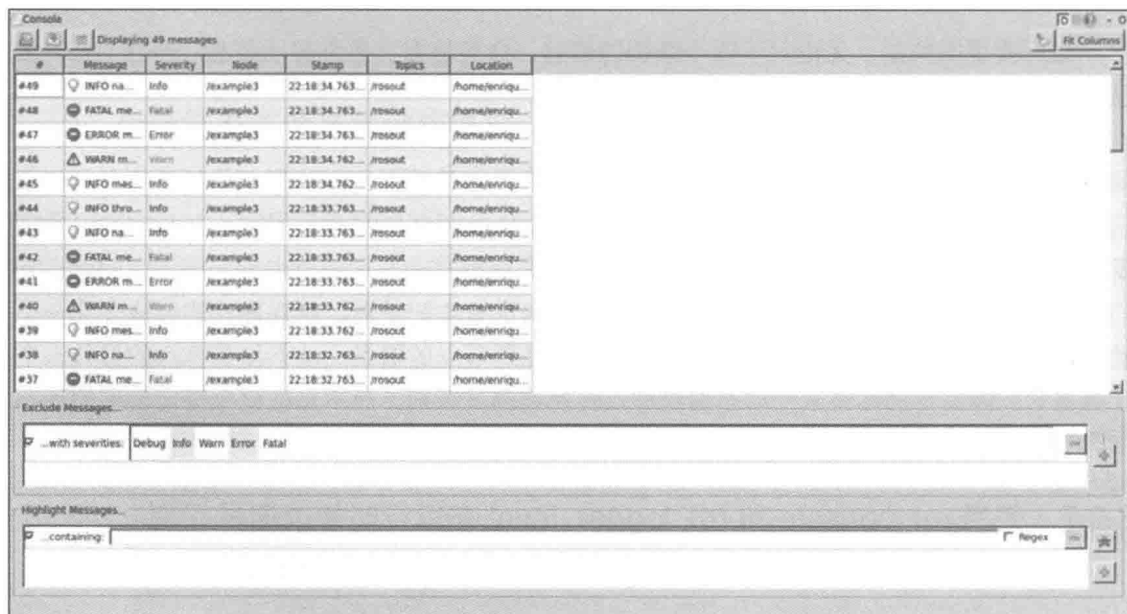
将看到如下窗口：



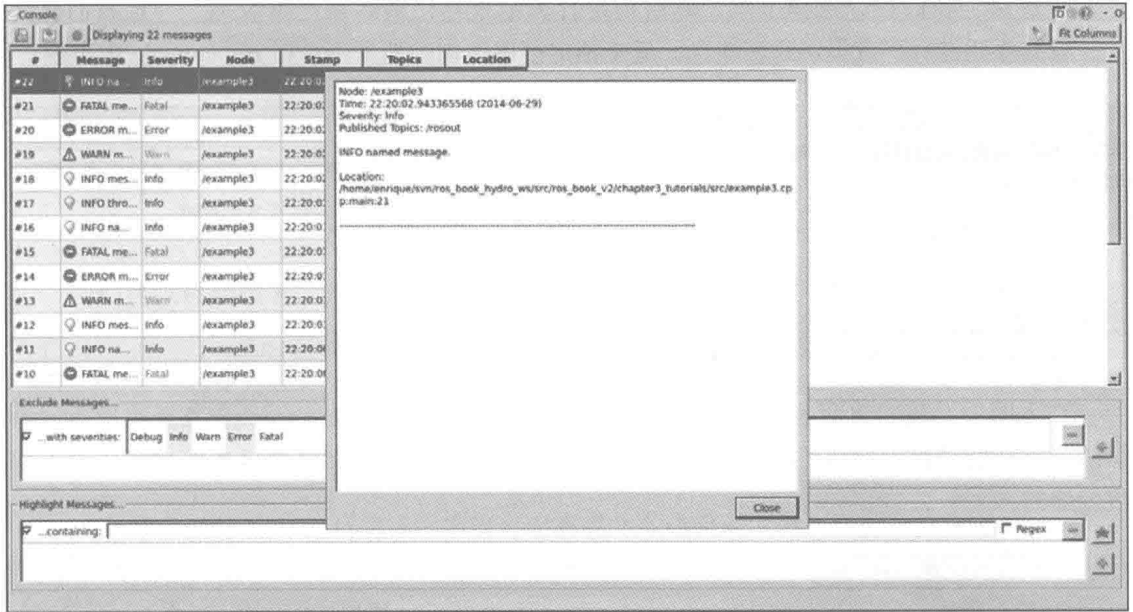
运行下面节点：

```
$ rosrund chapter3_tutorials example3
```

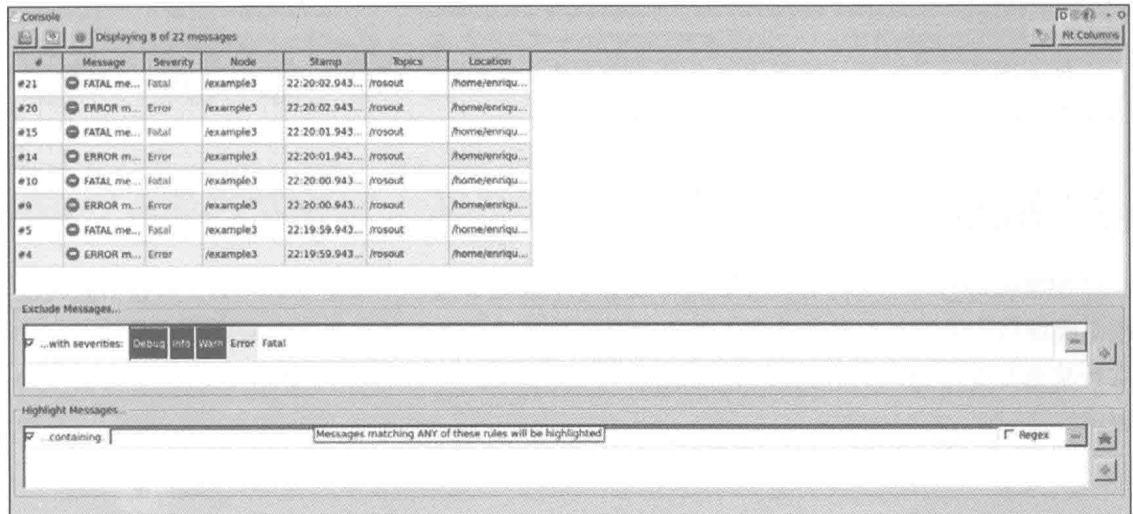
一旦 example3 节点开始运行，就会看到如下图所示的信息。注意 roscore 必须已经启动，也必须点击在 rqt_console 窗口上的记录（recording）按钮。



在 `rqt_console` 下, 消息按类别进行收集和显示, 如通过时间戳、信息类型、严重级别以及产生这些信息的节点等。你可以通过点击 `Resize columns` 自动调整格式。双击一个消息, 你可以看到所有信息, 包括生成它的那行代码, 如下图所示:



交互窗口可以实现暂停、保存、读取和载入保存的日志信息。我们可以清理信息列表并过滤它们。在 ROS hydro 下, 除了过滤掉的信息外, 信息依赖于过滤条件的特定接口, 例如, 节点可以通过一条规则进行过滤, 将我们选择的节点排除。另外, 通过相同的方式, 我们可以设置高亮显示过滤器。如下图所示:

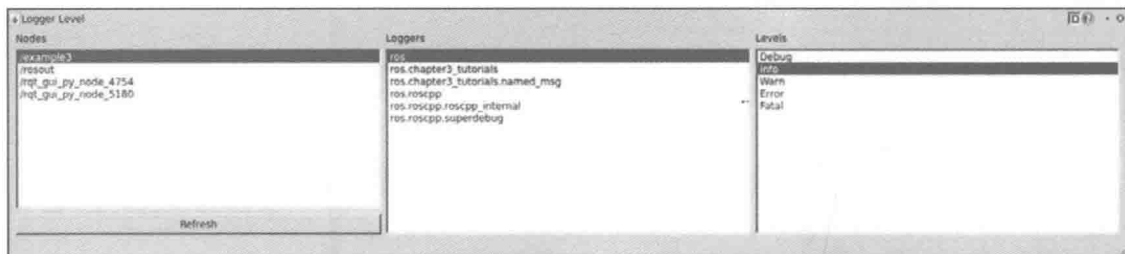


举一个例子，上图所示信息想过滤除 FATAL 和 ERROR 外其他级别的信息。

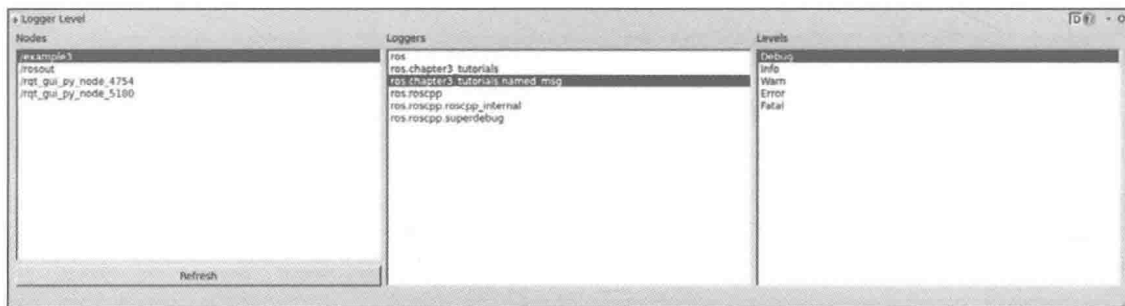
为了设置日志记录器的严重级别，必须运行以下命令：

```
$ rosrund rqt_logger_level rqt_logger_level
```

在这里，我们可以选择节点，然后指定日志记录器，最后是严重级别。一旦我们修改它，就会收到带有严重级别的新消息，低于预定严重级别的消息不会在 rqt_console 中出现：



在下图中，我们将命名为 `ros.chapter3_tutorials.named_msg` 的 `example3` 节点的日志记录器的严重级别设置为最低（DEBUG）。记住，所命名的日志记录器是通过 `*_NAMED` 日志函数创建的：



如上图所示，在默认情况下每个节点都有多个内部日志记录器，与 ROS 通信 API 相关，一般不要降低它们的严重级别。

3.3 检测系统状态

当系统运行时，可能有数个节点和数十个主题在节点中发布消息。同时，有些节点可能也会提供行为或服务。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 对此提供了一些基本但非常强大的工具，包括从 CLI 到 GUI 应用。

3.3.1 检测节点、主题、服务和参数

坦白地讲，我们应该先回顾一下学习过的基本内容。如何获得正在运行的节点、主题、给定时间可用服务的清单，如下表所示。

获取所有列表	命令
运行的节点	<code>rostopic list</code>
所有节点运行的主题	<code>rostopic list</code>
所有节点运行的服务	<code>rosservice list</code>
服务器中的参数	<code>rosparam list</code>

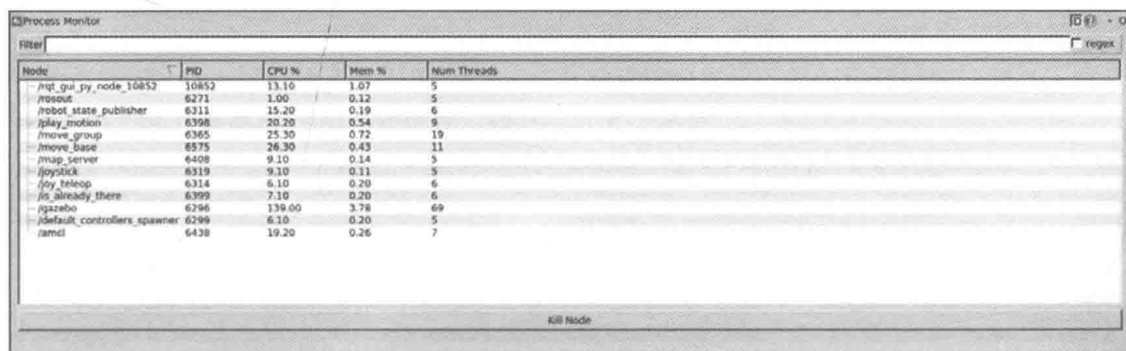
我们建议你回顾第 2 章，熟悉如何使用这些命令，同时看一下如何使用 `rosmmsg show` 获取特定主题和字段发出的消息类型。

所有这些命令可以结合 `bash` 命令，如 `grep`，寻找所需的节点、主题、服务或参数。例如，启动目标主题可以使用以下命令：

```
$ rostopic list | grep goal
```

`bash` 命令 `grep` 在文件列表或标准输出中查找文本或模式，如本例所示。

此外，ROS 提供几个 GUI 检测主题和服务。首先，在一个类似于进程表 (ToP) 窗口 `rqt_top` 中显示运行的节点，可以快速查看正在使用的所有节点和资源。以 REEM 导航栈运行仿真为例，如下图所示：



Node	PID	CPU %	Mem %	Num Threads
rqt_gui_py_node_10852	10852	13.10	1.07	5
roscout	6271	1.00	0.12	5
robot_state_publisher	6311	15.20	0.19	6
ipkey_motion	6398	20.20	0.54	9
move_group	6365	25.30	0.72	19
move_base	6575	26.30	0.43	11
map_server	6408	9.10	0.14	5
joystick	6319	9.10	0.11	5
joy_teleop	6314	6.10	0.20	6
is_already_there	6399	7.10	0.20	6
gazebo	6296	136.00	3.78	60
default_controllers_spawner	6299	6.10	0.20	5
ramd	6438	19.20	0.26	7

另一方面，`rqt_topic` 显示主题调试信息，包括发布者、接收者、发布速率和发布的消息。可以查看消息字段的主题并选择你想要订阅的主题，分析带宽和速率 (Hz) 以及查看最新消息发布。注意锁定的主题通常不会持续发布，所以不会看到任何关于它们的信息。如下图所示。

Topic	Type	Bandwidth	Freq	Value
/jam3d_pose	geometry_msgs/PoseWithCovarianceStamped	unknown	unknown	
header	std_msgs/Header			
frame_id	string			"map"
seq	uint32			0
stamp	time			genspy.Time[591700000]
pose	geometry_msgs/PoseWithCovariance			(0.20457495899314834, -0.001707561741432752, ...)
covariance	float64[36]			
pose	geometry_msgs/Pose			
orientation	geometry_msgs/Quaternion			
quaternion	float64			0.9999948663062841
x	float64			0.0
y	float64			0.0
z	float64			0.003204247349652341
position	geometry_msgs/Point			
x	float64			0.005243756470619018
y	float64			0.023378910660500424
z	float64			0.0
/jam3d/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/jam3d/parameter_updates	dynamic_reconfigure/Config			not monitored
/attached_collision_object	moveit_msgs/AttachedCollisionObject			not monitored
/back_camera/camera_info	sensor_msgs/CameraInfo			not monitored
/back_camera/image	sensor_msgs/Image	10.22MB/s	9.34	not monitored
/back_camera/image/compressed	sensor_msgs/CompressedImage			not monitored
/back_camera/image/compressed/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/back_camera/image/compressed/parameter_updates	dynamic_reconfigure/Config			not monitored
/back_camera/image/compressed/Depth	sensor_msgs/CompressedImage			not monitored
/back_camera/image/compressed/Depth/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/back_camera/image/compressed/Depth/parameter_updates	dynamic_reconfigure/Config			not monitored
/back_camera/parameter_descriptions	dynamic_reconfigure/ConfigDescription			not monitored
/back_camera/parameter_updates	dynamic_reconfigure/Config			not monitored
/base_inclinometer	geometry_msgs/Vector3	16.80KB/s	50.00	(-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
angular_velocity	geometry_msgs/Vector3			
angular_velocity_covariance	float64[9]			
header	std_msgs/Header			
linear_acceleration	geometry_msgs/Vector3			(-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
x	float64			0.0
y	float64			0.0
z	float64			0.0
linear_acceleration_covariance	float64[9]			
orientation	geometry_msgs/Quaternion			(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
orientation_covariance	float64[9]			
/bumper_states	gazebo_msgs/ContactsState			can not get message class for type "gazebo_msgs/C..."
/clock	roscpp_msgs/Clock			not monitored
/diagnostics	diagnostic_msgs/DiagnosticArray			not monitored

同样，`rqt_publisher` 允许我们在一个接口管理 `rostopic pub` 命令的多个实例。它还支持 Python 发布消息和固定值的表达式。在下图中，我们看到发布了两个示例主题（我们将在两个不同的终端看到使用 `rostopic echo <topic>` 发布消息）。

Topic	Type	Rate	Expression
/distance	std_msgs/Int32	2.00	2%
/bumper	std_msgs/Bool	1.00	false
/data	bool		
/data	std_msgs/Int32	2.00	2%
/data	int32		

```

enrique@pc-acer[hydro]:$ rostopic echo /bumper
data: false
data: false
...

```

```

enrique@pc-acer[hydro]:$ rostopic echo /distance
data: 5200
data: 3200
...

```

另一种更灵活的 GUI 是 `rqt_ez_publisher`。对 ROS hydro 而言，必须使用下面的代码从一个空的工作空间手动安装：

```

$ cd src
$ git clone https://github.com/OTL/rqt_ez_publisher.git
$ cd ..
$ catkin_make
$ source devel/setup.bash

```

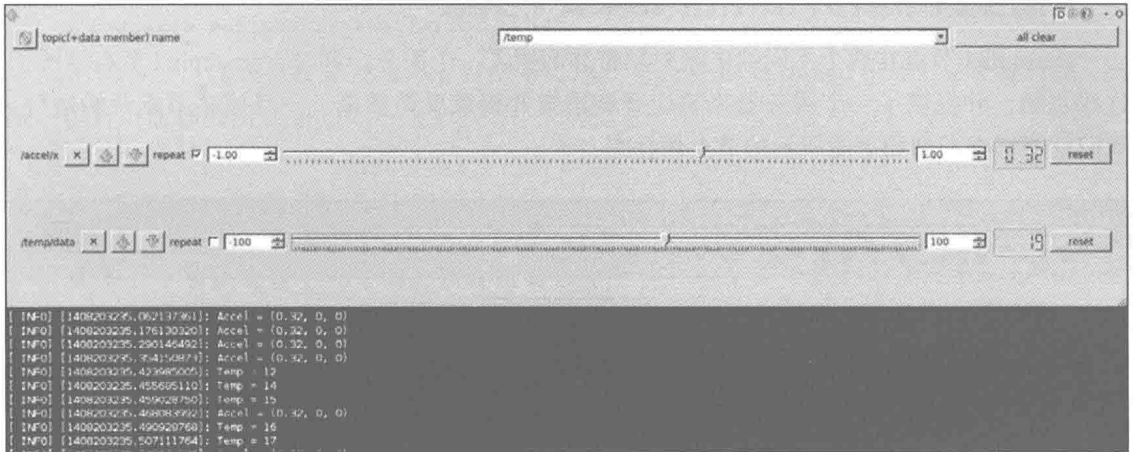

对 ROS indigo 而言, 它将会作为一个 Debian 功能包, 所以只需要运行下面命令:

```
$ sudo apt-get install ros-indigo-rqt-ez-publisher
```

然后, 运行:

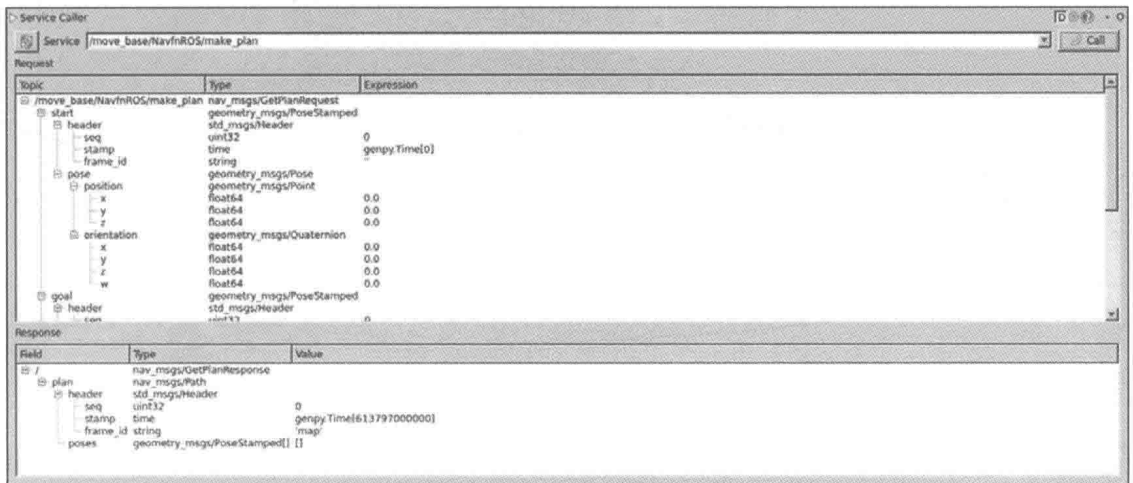
```
$ rosrn rqt_ez_publisher rqt_ez_publisher
```

在 example5 节点运行时, 你可以发布将被该节点读取的消息。在下图中, 我们将选择 accel 和 temp 主题 (删除 accel/y 和 accel/z 字段):



注意, 当启用重复 (repeat) 后, 消息会不断发布。否则, GUI 只在你改变值时发布消息。

rqt_service_caller 和 rosservice call 命令的多个实例一样。在下图中, 我们将调用 /move_base/NavfnROS/make_plan 服务, 我们必须为空服务设置请求。这对于来自 /amcl 节点的 /global_localization 服务而言是不需要的。点击 Call 按钮之后, 我们将获得响应消息。对于本例, 我们运行导航栈的 REEM 仿真, 如下图所示:



3.3.2 使用 rqt_graph 在线检测节点状态图

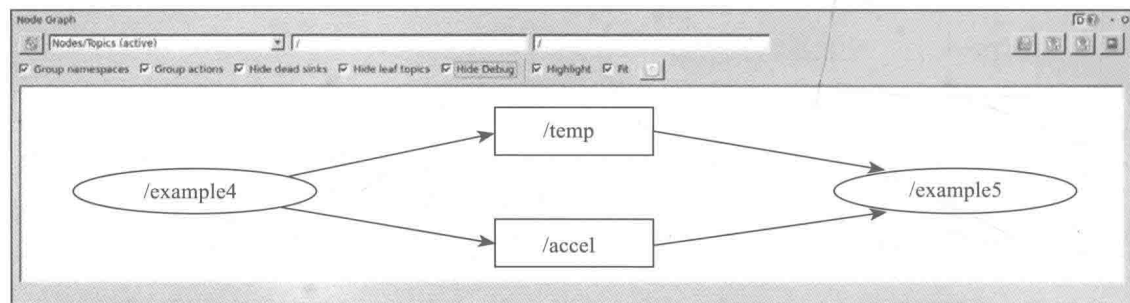
可以用有向图来显示 ROS 会话的当前状态，其中运行的节点是图中的节点，边为发布者 - 订阅者在这些节点与主题间的连接。此图形由 rqt_graph 动态绘制。

```
$ rosrun rqt_graph rqt_graph
```

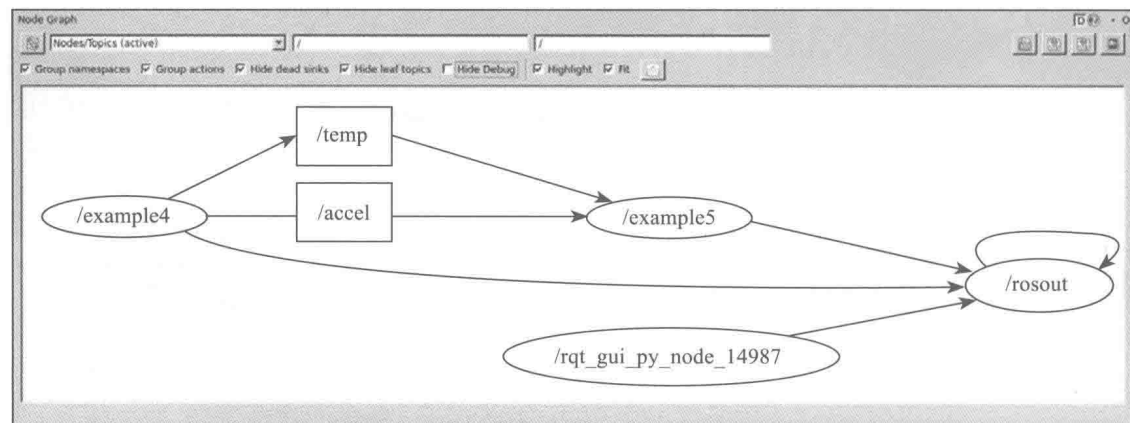
为了说明如何使用 rqt_graph 检测节点、主题和服务，使用以下 launch 文件同时运行 example4 和 example5 节点：

```
$ 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，并提供了一个服务器来响应查询请求并提供反馈数据。一旦这些节点开始运行，我们就能够查询到如下图所示的节点拓扑图：



在上图中，我们看到节点通过主题相连。由于选择 Hide Debug，我们不会看到 ROS 服务器节点 `rosout` 以及 `rosout` 主题发布到诊断聚合器 (diagnostic aggregator) 上的日志信息，如我们之前做的那样。取消这个选项就可以显示该节点和主题的调试信息，以便显示 ROS 服务器和 rqt_graph 节点本身 (如下图所示)。这对于调试大型系统非常有用，因为图可以使其简化。同时，ROS hydro 中相同命名空间的节点被分组，例如图像管道的节点：



当系统出现问题时，会一直（而不只是当我们移动鼠标掠过它们时）以红色显示。在这种情况下，选择 all topics 查看没有连接的主题非常有用，这通常显示由于主题名称拼写错误导致节点之间连接的中断。

当节点在不同的机器上运行时，rqt_graph 会显示其强大的高级调试功能，包括节点能否从各自的机器看到对方、列举连接等。

最后，我们可以启用统计查看在主题边上显示的信息速率、带宽，以及写入速率和行速率，如下一个图所示。我们必须在运行 rqt_graph 之前设置参数来使信息有效：

```
$ rosparam set enable_statistics true
```

不幸的是，这些参数 ROS hydro 目前不可用。它会在 ROS indigo 中（下一个发行版）出现，也有可能移植回 hydro 中。

3.4 设置动态参数

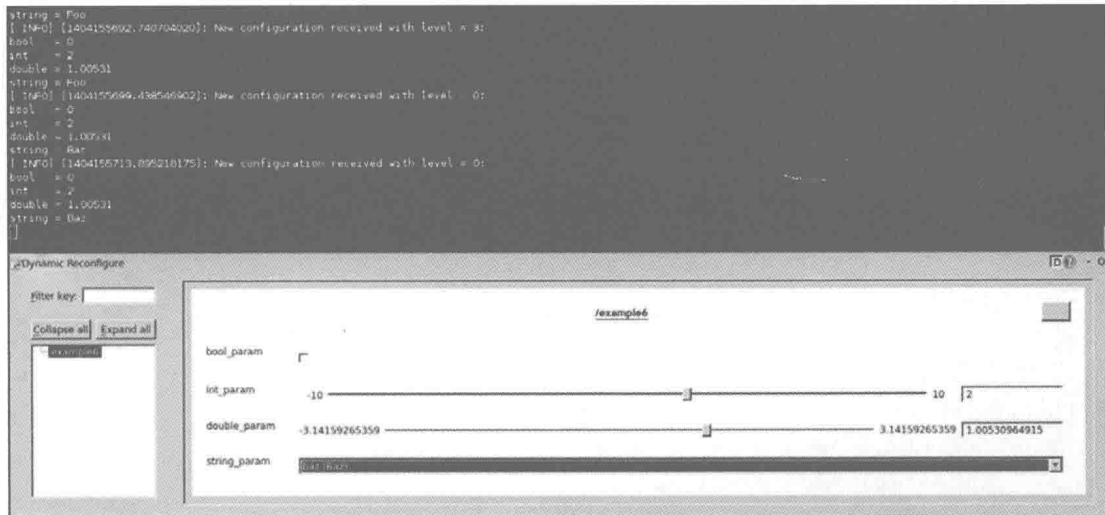
如果一个节点配置了一个动态重配置参数服务器，在工作中就可以使用 rqt_reconfigure 进行修改。运行下面的代码启动一个带有几个参数动态重配置的服务器（见功能包 cfg 文件夹中的 cfg 文件）：

```
$ roslaunch chapter3_tutorials example6.launch
```

使用下面命令启动动态重配置服务器，打开 GUI：

```
$ rosrn rqt_reconfigure rqt_reconfigure
```

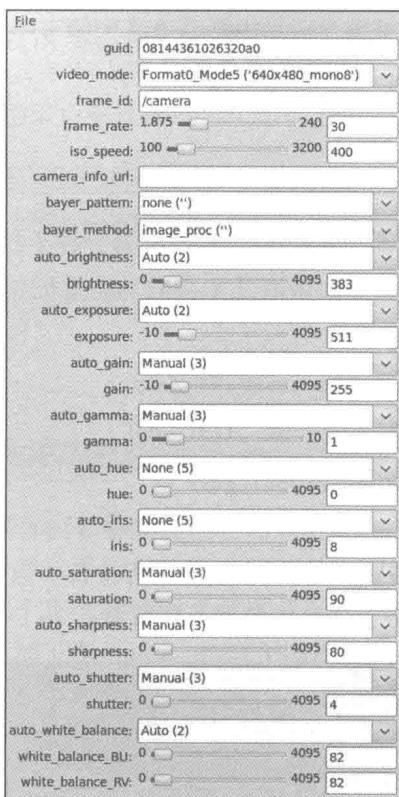
在左边的列表中选择 example6 服务器，然后就能看到它的参数，并可以直接修改。运行源代码中一个回调方法的代码会对值的有效性进行检查。也就是当回调方法执行时，在示例中这些参数的更改将立即生效。所述内容如下图所示：



动态参数原本是为驱动程序设计的，这使参数修改变得简单。因此动态参数已经在一些驱动上得以应用。尽管如此，它们也可以用于任何其他节点。驱动程序实现的示例，如 Hokuyo 激光测距仪的 `hokuyo_node` 驱动或 FireWire camera1394 驱动。FireWire 摄像头采用通用的驱动程序以支持传感器一些配置参数的改变，如帧速率、快门速度和亮度等。可以运行下面的命令启动 FireWire ROS 摄像头驱动 (IEEE 1394, a 和 b):

```
$ rosrund camera1394 camera1394_node
```

当摄像头运行时，我们可以用 `rqt_reconfigure` 配置参数，将会看到类似下图的界面：



请注意，我们将在第 5 章介绍如何使用摄像头，我们还将从开发人员的角度解释这些参数。

3.5 当出现异常状况时使用 `roswtf`

ROS 还提供了另外一些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`。对于 `chapter3_tutorials`，我们可以获得以下输出。请注意，如果你运行了一些代码，就可以用 ROS 图进行分析。运行 `roslaunch chapter3_tutorials example6.launch` 命令，输出如图所示。

```

enrique@pc-acer:hydro:/svn:
~/svn/res_book/hydro_ws/src/res_book_v2/chapter3_tutorials$ roswtf
No package or stack in context
=====
Static checks summary:

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

WARNING: You have pip installed packages on Ubuntu, remove and install using Debian packages: rospkg

=====
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:

No errors or warnings
enrique@pc-acer:hydro:/svn:
~/svn/res_book/hydro_ws/src/res_book_v2/chapter3_tutorials$

WARNING: Package name "3dof_bringup" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_robot" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_description" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_controller_configuration" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
process[roscout-1]: started with pid [12411]
started core service [/roscout]
process[example-2]: started with pid [12423]
[ INFO] [1404156269.270210812]: New configuration received with level = 4294967295:
bool    = 1
int     = 0
double  = 0
string  = Foo

```

通常情况下，我们希望获得的结果没有任何错误和警告，但是很多错误和警告在系统正常运行时是没有影响的。在上图中，roswtf 没有检测到任何错误，只有一个关于 pip 的警告，这有时可能是由系统安装的 Python 代码产生的问题。请注意，roswtf 的作用是检测信号的问题，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。

另一个有用的工具是 catkin_lint，可以帮助 catkin 诊断错误，通常在 CMakeLists.txt 和 package.xml 文件中。在 chapter3_tutorials 中，得到如下输出：

```
$ catkin_lint -W2 --pkg chapter3_tutorials
```

使用参数 -W2，就看到通常会被忽略的警告，如下图所示：

```

enrique@pc-acer:hydro:/svn:
~/svn/res_book/hydro_ws/src/res_book_v2/chapter3_tutorials$ catkin_lint -W2 --pkg chapter3_tutorials
chapter3_tutorials: notice: target name 'example6' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example7' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example4' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example5' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example2' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example3' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example1' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example1' might not be sufficiently unique
chapter3_tutorials: CMakeLists.txt(00): notice: extra arguments in endforeach()
catkin_lint: checked 1 packages and found 9 problems
enrique@pc-acer:hydro:/svn:
~/svn/res_book/hydro_ws/src/res_book_v2/chapter3_tutorials$

WARNING: Package name "3dof_bringup" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_robot" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_description" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_controller_configuration" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
process[roscout-1]: started with pid [12411]
started core service [/roscout]
process[example-2]: started with pid [12423]

```

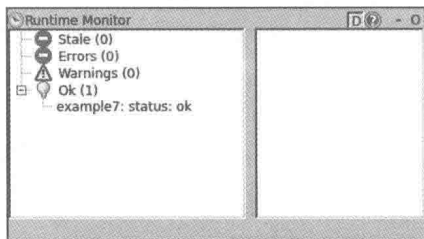
3.6 可视化节点诊断

ROS 节点可以使用 diagnostics (诊断) 主题提供诊断信息, 并为此提供了一个 API 用于以标准方式发布诊断信息。信息遵循 diagnostic_msgs/DiagnosticStatus 的消息类型, 它允许我们指定一个级别 (警告、错误)、名称、消息和硬件 ID 以及 diagnostic_msgs /KeyValue 列表, 对应键 (key) 和值 (value) 字符串。

有趣的部分是收集诊断信息并将其可视化的工具。在最基本的层面上, rqt_runtime_monitor 通过 diagnostics 主题直接发布可视化信息。运行 example7 节点, 它通过诊断主题发布信息并可以通过可视化工具查看诊断信息:

```
$ roslaunch chapter3_tutorials example7.launch
$ rosrn rqt_runtime_monitor rqt_runtime_monitor
```

前面的命令输出如下所示:



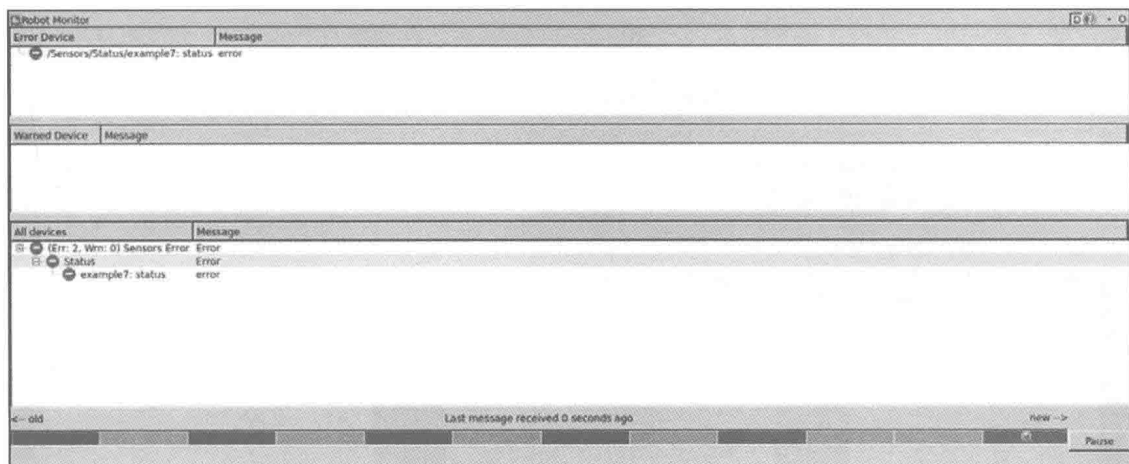
当系统很大时, 我们可以使用 diagnostic_aggregator 汇总诊断信息。在 diagnostics_agg 里处理和归类 diagnostics 主题的消息并重新发布。这些汇总诊断消息通过 rqt_robot_monitor 实现可视化。诊断汇总器通过一个配置文件进行配置, 例如下面的这一个 (参看 chapter3_tutorials 中的 config/diagnostic_aggregator.yaml 文件), 并使用 AnalyzerGroup 定义不同的 analyzers (分析器):

```
type: AnalyzerGroup
path: Sensors
analyzers:
  status:
    type: GenericAnalyzer
    path: Status
    startswith: example7
    num_items: 1
```

launch 文件已经在之前的代码中使用, 并以之前的配置运行诊断节点 aggregator_node, 如下:

```
$ rosrn rqt_robot_monitor rqt_robot_monitor
```

现在, 我们可以比较 rqt_runtime_monitor 与 rqt_robot_monitor 的可视化, 如下图所示。



3.7 绘制标量数据图

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。当然非标量数据也可以绘制，但是要分别在不同的标量域里进行。我们之所以在此仅讨论标量数据，是因为对于大多数非标量数据，有专门的工具能够更好地对其进行表示，我们会在后面进行部分介绍，例如图形、位姿、方向和角度等。

用 rqt_plot 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。然后，我们就能够在 y 轴上使用 rqt_plot 工具绘制标量数据。rqt_plot 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），也可以在 GUI 中手动添加或删除主题和字段。

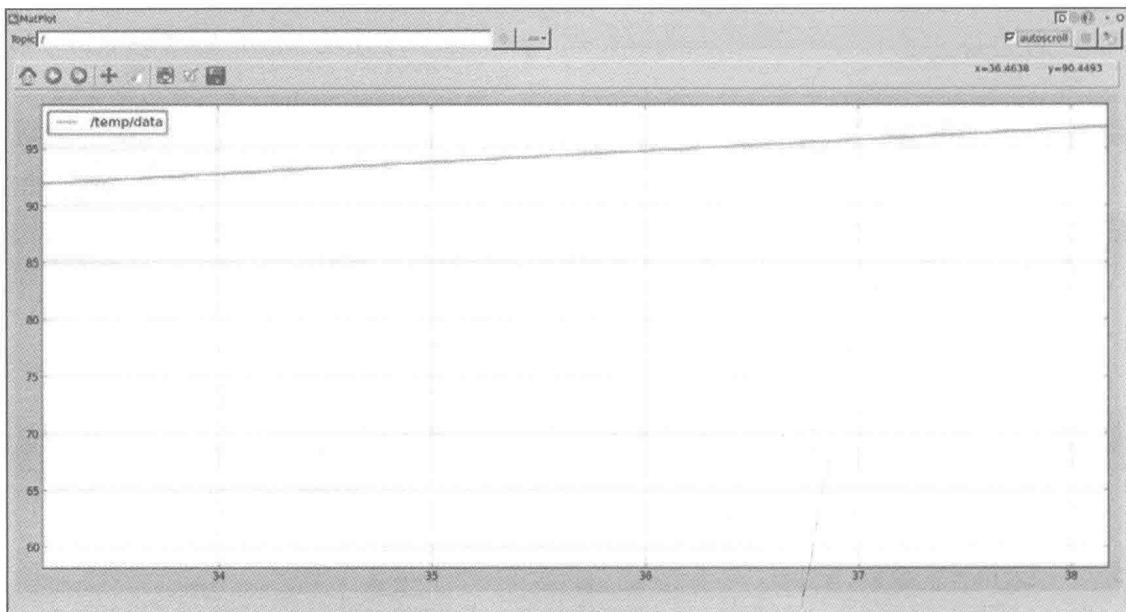
为了能够实际展示 rqt_plot 工具，我们使用 example4 节点，它在两个不同的主题中分别发布一个标量和一个矢量（非标量），这两个主题分别是温度（temp）和加速度（accel）。在这些消息中的值是随机生成的，所以它们没有实际意义，仅用于示范曲线绘制。那么首先以下的命令运行节点：

```
$ rosrun chapter3_tutorials example4
```

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，数据格式为 Int32，运行下面命令：

```
$ rosrun rqt_plot rqt_plot /temp/data
```

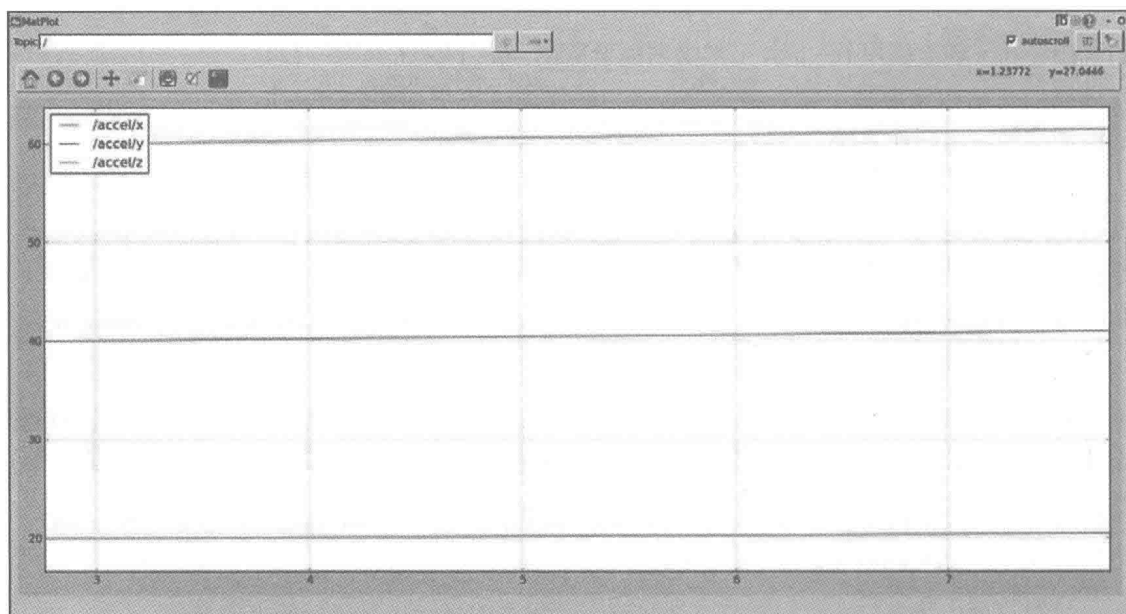
只要节点正常运行，我们就能看到曲线图随输入消息一直变化。如下图显示。



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。`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：

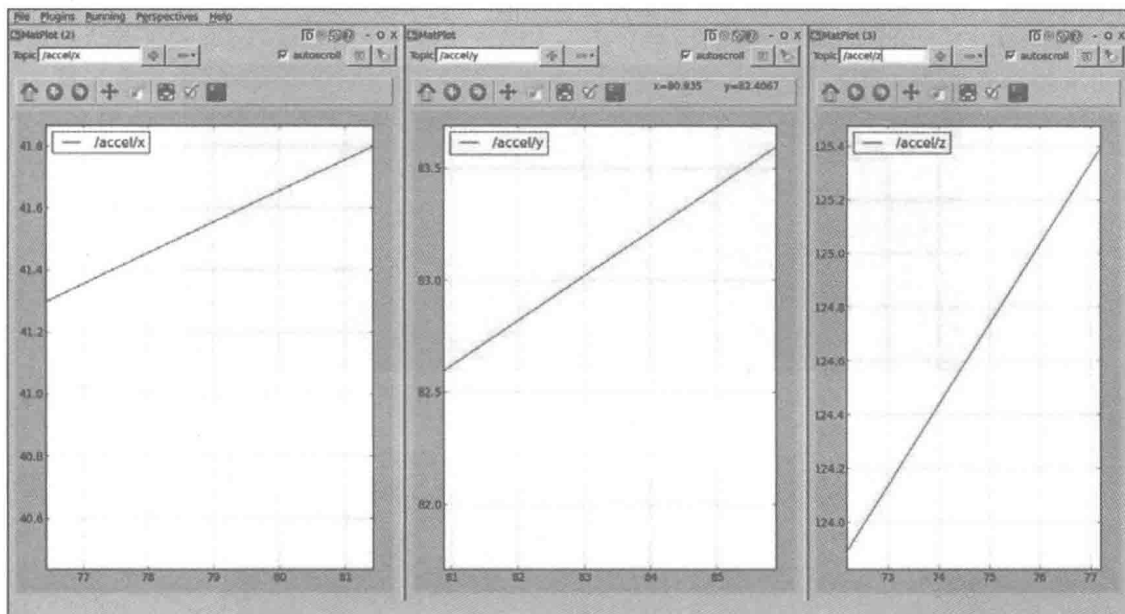
```
$ rosrund rqt_plot rqt_plot /accel/x:y:z
```

绘制的曲线图如下：



我们还可以将每个字段分开绘制。由于 `rqt_plot` 不直接支持这个功能，因此我们需要使用 `rqt_gui` 将三个图表手动分开，如下图所示：

```
$ rosrun rqt_gui rqt_gui
```

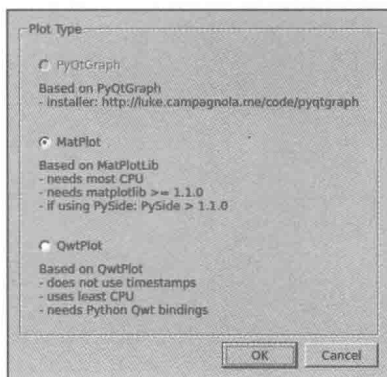


`rqt_plot` 的 GUI 支持三种绘图前端。在 ROS hydro 之前，只有 `matplotlib` 支持。现在，我们可以使用更快的 QT 前端，同时它支持更多的时间序列。可以点击配置 (configuration) 按钮选择 (如右图所示)：

3.8 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。这是一个复杂数据主题的例子，这些数据可以使用特殊工具更好地可视化或分析。你只需要一个摄像头来完成这些，例如你笔记本上的 webcam。在 `example8` 节点中，通过调用 `OpenCV` 库实现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。

我们仅会使用 `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 example8.launch
```

一旦节点运行起来，我们就能够列出主题列表（rostopic list），并查看是否包含 /camera 主题。查看是否正确捕捉到图像有一个简单直接的方法，使用 rostopic hz /camera 语句查看在主题中收到的图像更新频率是多少。频率通常是 30Hz，如下图所示：

```
enrique@pc-acer[hydro]:svn:
~/svn/ros_book_hydro_ws/src/ros_book_v2/chapter3_tutorials$ rostopic hz /camera
subscribed to [/camera]
average rate: 10.728
  min: 0.084s max: 0.099s std dev: 0.00474s window: 10
average rate: 10.746
  min: 0.084s max: 0.099s std dev: 0.00441s window: 21
average rate: 10.725
  min: 0.084s max: 0.099s std dev: 0.00426s window: 31
average rate: 10.710
  min: 0.084s max: 0.100s std dev: 0.00409s window: 42
average rate: 10.702
  min: 0.084s max: 0.100s std dev: 0.00398s window: 53
□
```

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，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：

```
$ rosrund image_view image_view image:=/camera
```

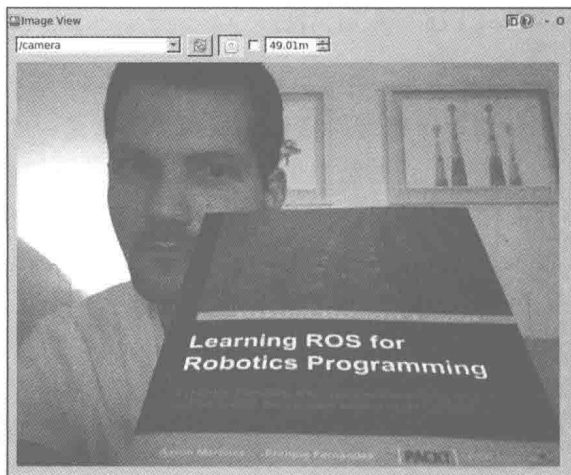
这里使用了 image_view 节点，在窗口中展示了给定主题（使用 image 参数）的图像，如下图所示。这样我们就能简单直接地显示在主题内发布的每一幅图像或帧，即使数据来自网络也可以实现。你可以通过在窗口中点击右边的按钮将当前帧保存在硬盘里，通常会存在 home 目录下或者 ~/.ros 目录下。

ROS hydro 还有 rqt_image_view，支持在一个窗口查看多个图像，但不允许点击右边按钮保存图像。我们可以在 GUI 上手动选择图像主题或使用 image_view：

```
$ rosrund rqt_image_view rqt_image_view
```

前面命令结果如右图所示：

ROS 提供了一个基于 OpenCV 校准 API 的摄像头校准界面。我们将在第 5 章介绍如何使用摄像头，包括单目和双目摄像头以及 ROS 图像管道（image_proc 和 stereo_image_proc）相关内容，这些能修正摄像头图像失真、计算双目摄像头深度差异，并得到一个点云。



3.9 3D 可视化

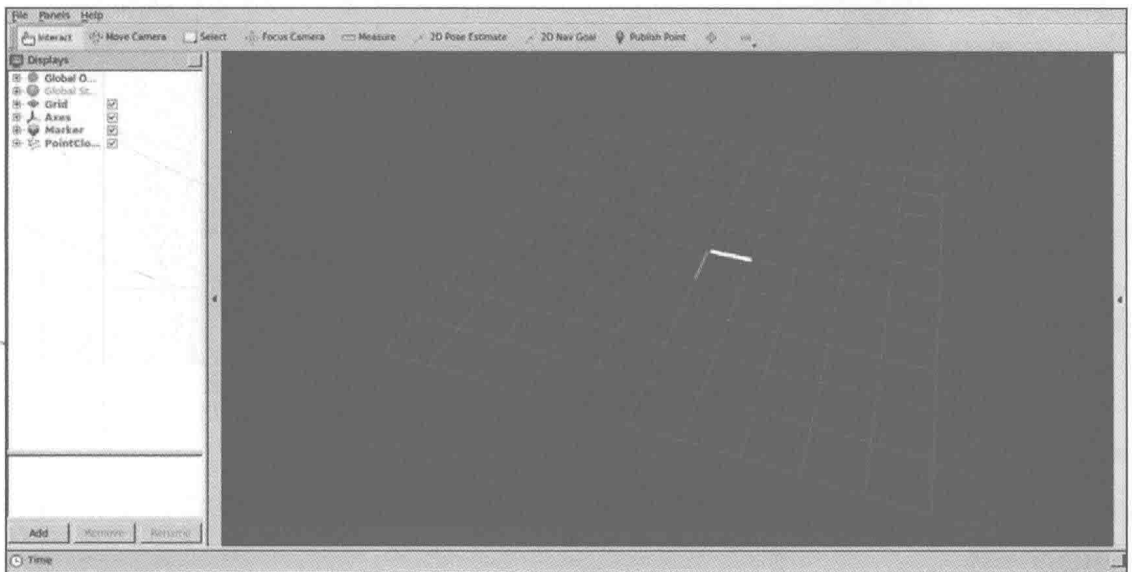
正如我们在前面章节中所见，有很多设备（例如双目摄像头、3D 激光和 Kinect 传感器等）能够提供 3D 数据。它们通常使用点云格式（组织好或未组织的）。基于上述原因，能够使用工具实现 3D 数据可视化就非常有用。本节我们将介绍 ROS 系统中的 `rviz` 或 `rqt_rviz` 工具。它集成了能够完成 3D 数据处理的 OpenGL 接口，能够将传感器数据在模型化世界（world）中展示，过程是先使用传感器坐标系读取测量值，再将这些读数按照之间的相对位置在正确的位置绘制。

3.9.1 使用 `rqt_rviz` 在 3D 世界中实现数据可视化

在 `roscore` 运行时，启动 `rqt_rviz`（请注意 `rviz` 在 ROS hydro 中依然有效）：

```
$ rosrun rqt_rviz rqt_rviz
```

我们将会看到如下图所示的图形化工作界面：



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，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/example9.rviz` 文件中，可以通过点击 `File | Open Config` 加载。

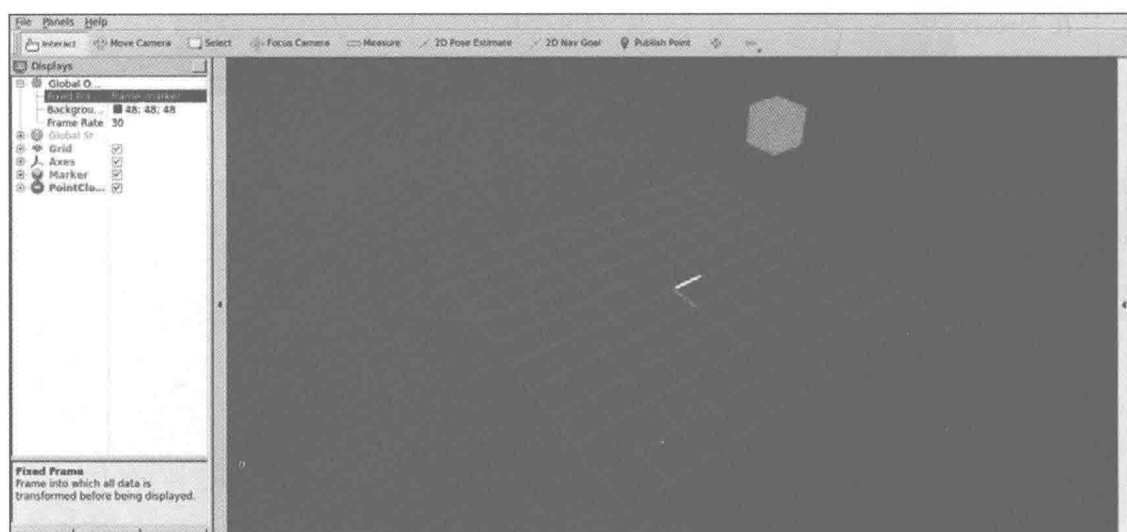
在 `Displays` 区域之下有一个 `Add` 按钮，允许通过主题或类型添加更多的参数项。同时，注意到这里还有一些全局选项，基本上是由于设定固定坐标系的工具，因为坐标系是可以移动的。其次，还有轴（`Axes`）和网格（`Grid`）作为各个参数项的参照物。在示例中，对于 `example9` 节点，我们将会看到标记（`Marker`）和点云（`PointCloud2`）。

最后，在状态栏上有时间相关的信息提示，以及在右侧有一些菜单。Tools 用于配置一些插件参数，如 2D Nav Goal 和 2D Pose Estimate 主题名等。在 Views 目录提供了不同查看类型，一般而言有 Orbit 和 TopDownOrtho 就足够了，一个用于 3D 查看，另一个用于 2D 俯视图查看，其他目录显示了环境中一些可以选择的元素。在顶部，有一个当前操作模式的菜单栏，包括交互（Interact）、移动（Move）、测量（Measure）等，以及一些插件。

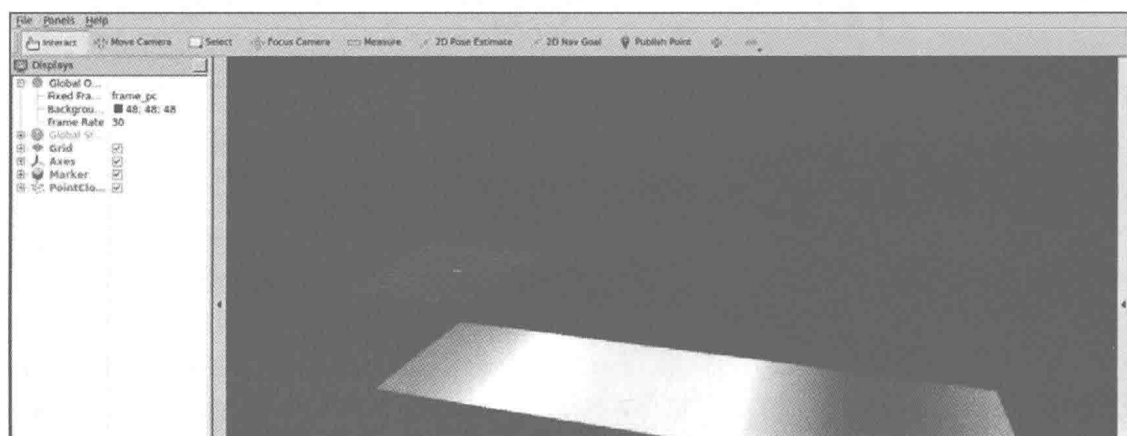
现在运行 example9 节点：

```
$ roslaunch chapter3_tutorials example9.launch
```

在 rqt_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）。我们将会看到红色方块标记在移动，如下图所示：



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：



支持的 `rqt_rviz` 内置类型的参数列表包括 `camera` 和 `image`。它们会在一个类似于 `image_view` 的窗口中显示。选择 `camera` 的时候，需要对它先进行校准，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 `LaserScan` 数据，红外 / 声呐 (IR/sonar) 传感器的锥状距离数据 `range`，以及 3D 传感器例如 Kinect 的 `PointCloud2`。

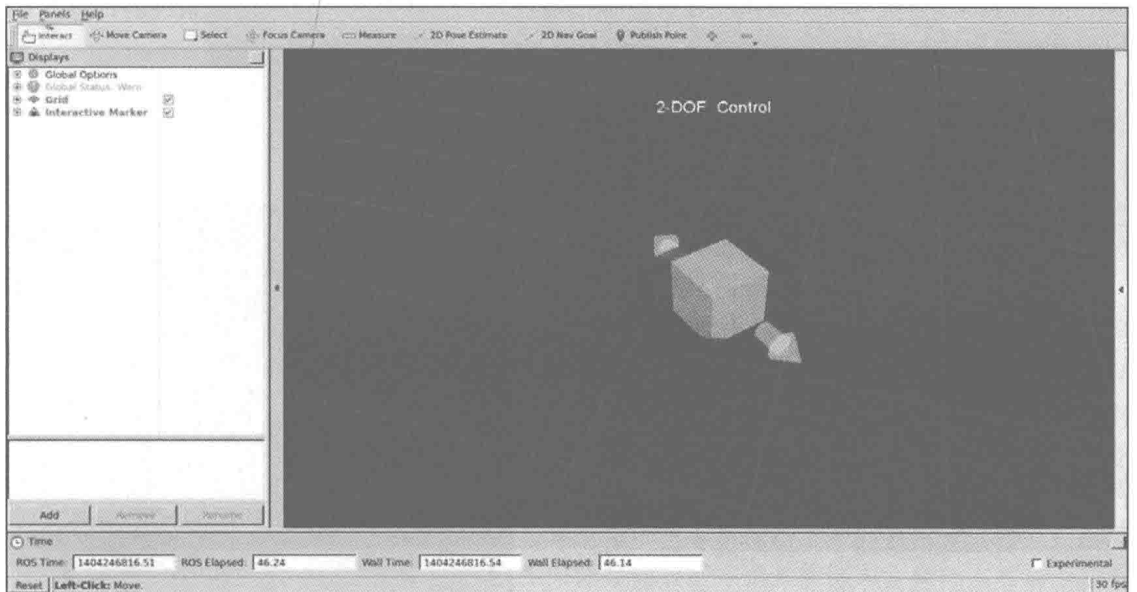
对于导航功能包集，将会在下一章进行详细介绍。我们会使用多种数据类型，例如里程 `Odometry`（画出机器人的里程位姿），路径 `Path`（画出机器人所走过的路径），物体位姿 `Pose`，带有机器人位姿估计的粒子云 `PoseArray`，使用 `Occupancy Grid Map`（OGM）的地图 `Map`，以及 `costmaps`（这是 ROS hydro 的 `Map` 类型，之前是 `GridCell`）。

在这些类型中，需要说明的是机器人的模型 `RobotModel`，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 (tf) 树，并且在仿真环境中为坐标系调试提供非常大的帮助。我们会在下一节展示示例。在 `RobotModel` 中，我们用机器人 URDF 描述中的关节绘制一条轨迹，看它们如何随时间变化移动。

基本元素也可以被表示，例如机器人足迹的 `Polygon`、各种不同的标记 `Markers` 通常支持基本几何元素，如立方体、球体、线条等，甚至是交互式标记对象 `InteractiveMarker`。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）。使用下面命令，运行 `example8` 节点查看简单交互式标记：

```
$ roslaunch chapter3_tutorials example10.launch
```

你将看到一个标记，可以在 `rqt_rviz` 交互模式中移动它。它的位姿可以用于修改系统中另一个参数的位姿，例如机器人关节：



3.9.2 主题与坐标系的关系

如果数据从真实世界中一个物理位置的特定传感器数据发布，主题必须有一个坐标系。例如，相对于机器人底盘的位置上有一个激光传感器（通常在轮式机器人两个轮子的轮轴中间）。如果我们需要用激光扫描数据去检测环境中障碍物或者构建地图，就必须对激光传感器和底盘所在的位置进行坐标转换。在 ROS 系统中，带有报文头的消息除了具有时间戳（在不同的消息间进行数据同步非常重要）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。

坐标系自身并没有意义，我们需要的是它们之间的坐标变换。实际上，机器人的 `tf` 坐标变换树都会有一个 `base_link` 作为根坐标系（或是地图，如果运行导航包的话）。这样，我们就能够在 `rqt_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。

3.9.3 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：

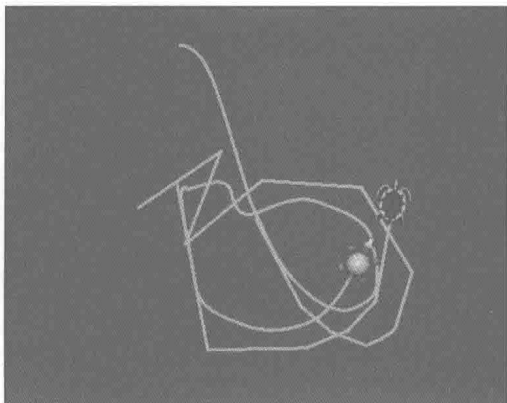
```
$ roslaunch turtle_tf turtle_tf_demo.launch
```

这是一个在 `rqt_rviz` 里能够用于展示 `tf` 可视化的最基本示例。需要注意的是 `tf` API 提供了很多不同的可能性，可以参考本书的后续章节。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和从一个坐标系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 `tf` 在系统中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` 进行变换。

如果你收到一个错误，这很可能是因为 `launch` 文件启动时监听器（`listener`）失效了。监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：

```
$ rosrunc 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。需要注意的是要停留在运行 `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它的。

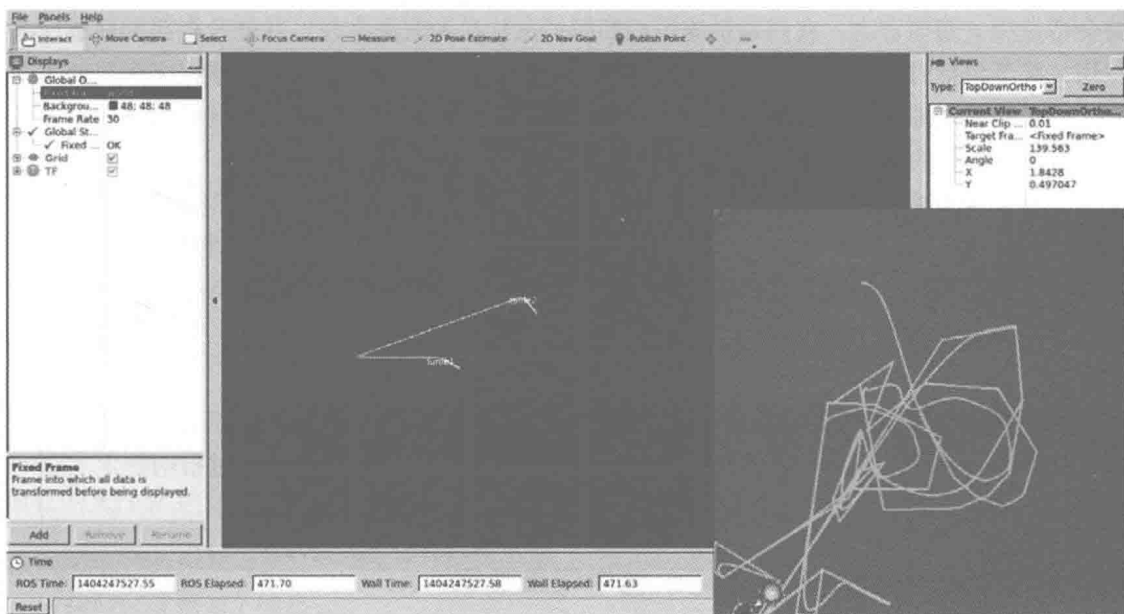


每一只小海龟都有自己的坐标系，可以在 `rqt_rviz` 中查看它们：

```
$ rosrn rqt_rviz rqt_rviz
```

现在，先放下 `turtlesim`，我们先来看当使用箭头键移动小海龟时 `rqt_rviz` 里小海龟的坐标系是如何移动的。首先设定固定坐标为 `/world`，然后将坐标转换 `tf` 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` 的显示中，坐标系由坐标轴组成，父 - 子之间的连接是一个带有粉红色尾巴的黄色箭头。同时，设定 `world` 的视图（view）为 `TopDownOrtho`，这会使示例中查看坐标系的移动更容易，因为它们仅在 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。同时，你也可以用这个示例和 `tf` 更改固定坐标系。请注意 `config/example_tf.rviz` 用作本示例的基本 `rviz` 配置：



3.10 保存与回放数据

通常情况下，当我们使用机器人系统时，资源通常是共享的，并不一定总能给你一个人使用。或者由于准备和实施实验消耗了大量的时间，或是实验难以重现等诸如此类的原因导致实验不能顺利完成。基于以上原因，将实验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很好的实践方法。然而要保证存储数据正确并能够用于离线回放却不是容

易的事情，因为很多实验并不一定能反复进行。幸运的是，ROS 中的强大工具使我们能够完成这项任务。

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 (bag) 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以充足的数据组织方式来实现这一切。

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

3.10.1 什么是消息记录包文件

消息记录包 (bag) 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，还包括使用消息的数据包。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，能够支持超高速数据流的处理和记录。这是记录数据最关键的功能。当然，这和消息记录包文件的大小也是相关的。但随着文件的增大会牺牲数据的记录速度。我们能选择 bz2 算法进行实时数据压缩，只需要在你使用 `rosvag 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` 参数）。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消息记录包中全部主题的某个特定子集进行回放。

3.10.2 使用 rosvag 在消息记录包中记录数据

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：

```
$ rosvun chapter3_tutorials example4
```

现在我们有二个选择。第一个是记录所有的主题：

```
$ rosvag 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，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消息记录包结果的示例：

```
[ INFO] [1404248014.668263731]: Subscribing to /temp
[ INFO] [1404248014.671339658]: Subscribing to /accel
[ INFO] [1404248014.674950564]: Recording to 2014-07-01-22-54-34.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等。或者像前文提到的那样，文件可以实时压缩（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，我们能以 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）。

在 launch 文件中可以直接调用 rosbag record，这在我们需要为一些主题设置记录器时非常有用。为此，我们需要在 launch 文件中增加一个如下所示的节点：

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

需要注意的是主题和传递给其他命令的参数会使用 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。

3.10.3 回放消息记录包文件

现在我们有了一个消息记录包文件（bag），可以使用它回放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 2014-07-01-22-54-34.bag
```

我们会看到如下输出：

```
[ INFO] [1404248314.594700096]: Opening 2014-07-01-22-54-34.bag
```

```
Waiting 0.2 seconds after advertising topics... done.
```

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

```
[RUNNING] Bag Time: 1404248078.757944 Duration: 2.801764 / 39.999515
```

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 来马上停止回放。一旦回放完毕，窗口就会关闭，但是有一个选项

(-1) 允许循环播放，这在有些时候也很有用。

通过 `rostopic list` 命令查看主题列表，如下所示：

```
/accel
/clock
/rosout
/rosout_agg
/temp
```

`/clock` 主题用于指定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` 为参数配置频率（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。

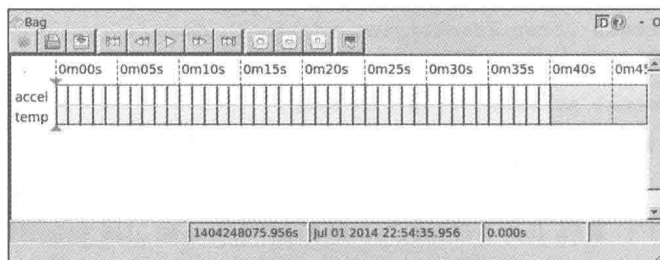
3.10.4 检查消息记录包文件的主题和消息

这里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>`，然后结果如下图所示：

```
~/svn/ros_book_hydro_ws/src/ros_book_v2/chapter3_tutorials/bag$ rosbag info 2014-07-01-22-54-34.bag
path:      2014-07-01-22-54-34.bag
version:   2.0
duration:  40.0s
start:     Jul 01 2014 22:54:35.96 (1404248075.96)
end:       Jul 01 2014 22:55:15.96 (1404248115.96)
size:      10.9 KB
messages:  82
compression: none [1/1 chunks]
types:     geometry_msgs/Vector3 [4a842b65f413084dc2b10fb484ea7f17]
           std_msgs/Int32 [da5909f8e378aef85e547e830cc1bb7]
topics:    /accel  41 msgs : geometry_msgs/Vector3
           /temp   41 msgs : std_msgs/Int32
```

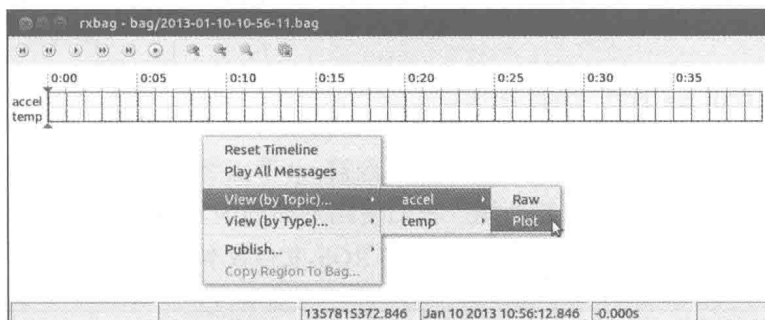
我们有消息记录包文件本身的信息，例如创建的日期、持续时间、文件大小，以及内部消息的数量和文件的压缩格式（如果有压缩）。然后，我们还会有文件内部数据类型的列表。最后有主题的列表，并对应它们的名称、消息数量和类型。

第二种检查包文件的方法则非常强大。这个 GUI 工具叫做 `rqt_bag`。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（RAW）。我们只需要传递记录包文件的名称，就会看到如下图所示的界面（使用之前的记录包文件）：

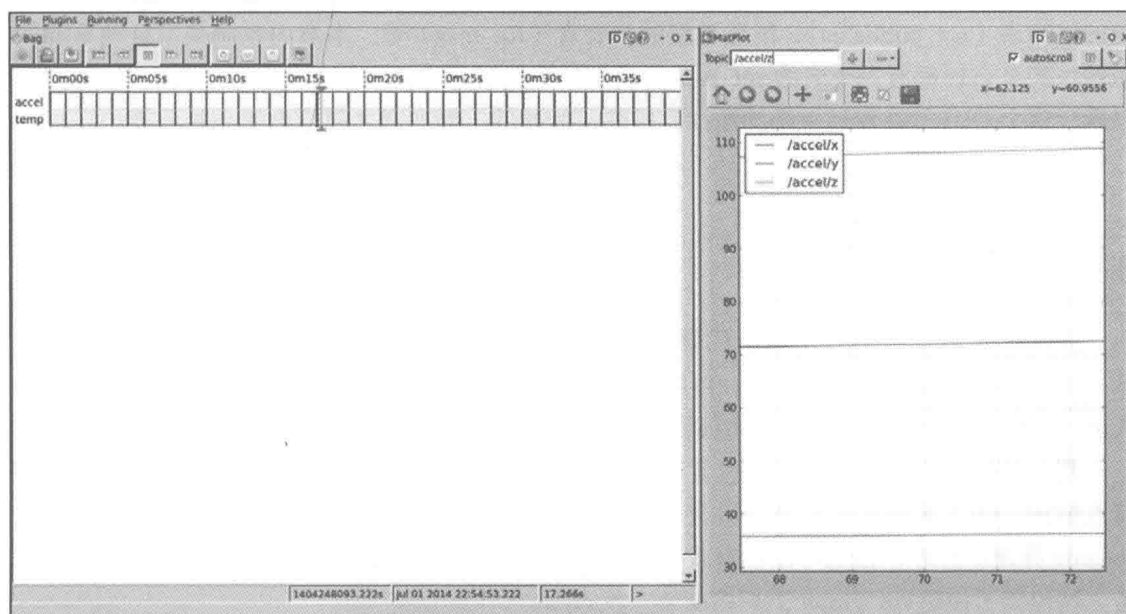


我们有一个能用于所有主题的时间条。对于每一个消息都会带有这个标记。在本例中，能够在时间条中使用缩略图查看消息。

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），来查看记录包文件中的主题。在时间条上单击右键就会弹出这个菜单。下图对应老 rxbag，这个功能已经被添加到 rqt_bag，但目前在 Debian 库中可能还不可用：



作为补充，我们使用 rqt_gui 将 rqt_bag 和 rqt_plot 插件放到同一窗口中。通过 config/bag_plot.perspective 文件夹导入下图所示的布局。不同于 rxbag，我们使用 Publish All 和播放查看绘图。对于 /accel 主题，我们可以使用一个坐标轴绘出所有的字段。为了实现这个功能，先要进入绘图（plot）的视图中，点击 + 按钮 / 图标选择所有字段。请注意，我们可以随后再删除它们或者新建一个轴。如之前提及的，绘图工具不生成文件中所有数值，它只是简单显示数据的回放和发布：



需要注意的是由于 rxbag 的特点，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。

由于 rqt_bag 的第一版还没有前面提到的预览和可视化功能，你将需要为 ROS hydro 编译 rxbag 或升级最新的 rqt_bag（或者从 GitHub 库中克隆 hydro-devel 分支并编译）。

3.11 应用 rqt 与 rqt_gui 插件

从 ROS Fuerte 发布以来，rx 应用或工具已经被 rqt 节点替代。它们基本上是一样的，只有一小部分的升级、bug 修正和新功能。我们使用下面列表对本章中介绍的工具进行说明（ROS hydro 的 rqt 工具以及对应被替代的之前的 ROS 发布版本）：

ROS hydro rqt 工具	替代 (ROS fuerte 或之前版本)
rqt_console 和 rqt_logger_level	rxconsole
rqt_graph	rxgraph
rqt_reconfigure rqt_reconfigure	dynamic_reconfigure reconfigure_gui
rqt_plot	rxplot
rqt_image_view	image_view
rqt_bag	rxbag

在 ROS hydro 中，这些可以单独运行的插件甚至更多，能够作为命令行（rqt_shell）、主题的发布者（rqt_publisher）、消息类型的检查者（rqt_msg）等，本章内容涵盖了这些重要的插件。甚至 rviz 都有一个名为 rqt_rviz 的插件，还能够集成在最新的 rqt_gui 界面中。我们可以运行这个 GUI 并在窗口上手动添加和排列一些插件，如本章之前给出的一些示例：

```
$ rosrun rqt_gui rqt_gui
```

3.12 本章小结

在阅读了本章并运行了示例代码之后，你就掌握了加快机器人系统构建速度、错误调试、结果可视化等大量实用工具。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，你将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 中包含的强大工具，例如 rqt_console 界面。另外，你也可以检查或列出运行中的节点、发布的主题和系统运行时提供的服务。这包含了使用 rqt_graph 工具检测节点状态图。

对于可视化工具，你已学习了使用 `rqt_plot` 绘制标量数据曲线。它能够以更加直观的特定变量分析方式发布你的节点。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 `rqt_image_view` 和 `rqt_rviz` 查看图像和 3D 数据，也可以使用这些工具校正和调整摄像头图像。

最后，你已经通过学习 `rosvbag` 工具学会了记录和回放主题中的消息，还知道了如何使用 `rqt_bag` 查看消息记录包文件的内容，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。

在接下来一章中，你将会使用本章讲过的这些工具可视化不同类型的数据，以及一些传感器在 ROS 中的使用说明和它们输出数据的可视化等。

第 4 章

在 ROS 下使用传感器和执行器

当你想到机器人，可能首先是人形机器人，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器，这样就能够与现实世界交互了。

在 <http://www.ros.org/wiki/Sensors> 有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器

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买得起的传感器。

传感器和执行器分成不同的类别：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器。

4.1 使用游戏杆或游戏手柄

我敢肯定你曾经或多或少使用过某个视频终端的游戏杆或游戏手柄 (joystick/gamepad)。

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。要安装这些功能包在 Ubuntu 系统中，首先执行下面的命令：

```
$ sudo apt-get install ros-hydro-joystick-drivers  
$ rosstack profile & rospack profile
```

在这些功能包中，你能找到学习如何使用游戏杆的代码和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，使用如下代码，检查你的游戏手柄是否能够被识别：

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

我们会看到下面输出：

```
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如何发送关于轴向输入和按钮的信息。

要查看节点所发布的消息，使用下面命令：

```
$ rostopic echo /joy
```

然后我们看到每次发出的消息：

```
---
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, 0]
---
```

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

如果你想知道消息的类型，在命令行中输入下面指令：

```
$ rostopic type /joy
```

你会获取消息所使用的类型。在本例中是 `sensor_msgs/Joy`。

现在，要查看消息中使用的字段，使用下面的命令行：

```
$ rosmmsg show sensor_msgs/Joy
```

然后你会看到：

```
std_msgs/Header header
  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节点，并生成用于移动小海龟（`turtlesim`）的指令。

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，它通过 `joy_node` 获取数据并通过发布（`Published`）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。

```
$ rosrunc turtlesim turtlesim_node
```

为了查看主题列表，请使用下面命令：

```
$ 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/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

现在我们需要知道主题的类型，请使用下面命令查看：

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

你将会看到这样的输出：

```
geometry_msgs/Twist
```


为了知道消息的内容，执行以下命令：

```
$ rosmmsg show geometry_msgs/Twist
```

你将会看到两个用于发送速度的字段：

```
geometry_msgs/Vector3 linear
```

```
float64 x
```

```
float64 y
```

```
float64 z
```

```
geometry_msgs/Vector3 angular
```

```
float64 x
```

```
float64 y
```

```
float64 z
```

现在我们已经定位了主题和要使用的数据结构，接下来创建程序，使用从游戏杆中获取的数据生成速度指令。

在 chapter4_tutorials/src 文件夹下创建一个新的文件 c4_example.1.cpp，并输入以下代码：

```
#include<ros/ros.h>
#include<geometry_msgs/Twist.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<geometry_msgs::Twist>("/turtle1/cmd_vel", 1);
    sub = n.subscribe<sensor_msgs::Joy>("joy", 10, &TeleopJoy::callBack,
```

```

this);
}

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

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

```

现在我们解释上面的代码。在 main 函数中，我们创建 TeleopJoy 类的一个实例。

```

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

```

在构造函数中，初始化 4 个变量。前两个变量使用参数服务器中的数据填充。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 geometry_msgs::Twist 类型发布主题。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<geometry_msgs::Twist>("/turtle1/cmd_vel", 1);
    sub = n.subscribe<sensor_msgs::Joy>("joy", 10, &TeleopJoy::callBack,
    this);
}

```

每一次节点都会收到一条消息，调用 callBack 函数。我们使用 vel 名称创建一个新的变量，它用于发布数据。游戏杆轴向输入的值被赋给 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)
{
    geometry_msgs::Twist vel;
    vel.angular.z = joy->axes[i_velAngular];
    vel.linear.x = 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="c4_example1" name="c4_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：c4_example1、sim 和 joy。

在 launch 文件中有 4 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值并指明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。

使用下面的命令运行 launch 文件：

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

如果一切顺利，你就能通过 `roslaunch 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 `rqt_graph`。

4.2 使用激光雷达——Hokuyo URG-04lx

在移动机器人中，获取障碍物的具体位置、房间的内部轮廓等信息非常重要。机器人使用地图进行导航和通过未知的区域。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。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。

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。要使用它，首先需要安装激光雷



达的驱动。

```
$ sudo apt-get install ros-hydro-hokuyo-node
$ rosstack profile && rospack profile
```

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系统是否能够正确地检测到它和它是否正确地配置：

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

当我们连上激光雷达时，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
```

在该示例中，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
```

使用下面的命令检查配置是否成功：

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

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

如果一切顺利，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：

```
$ rosrn 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

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
```

使用以下命令查看消息的数据结构：

```
$ rosmmsg show sensor_msgs/LaserScan
```

如果了解关于激光雷达工作和数据发送的更多信息，可以使用 `rostopic` 命令来查看实时消息：

```
$ rostopic echo /scan
```

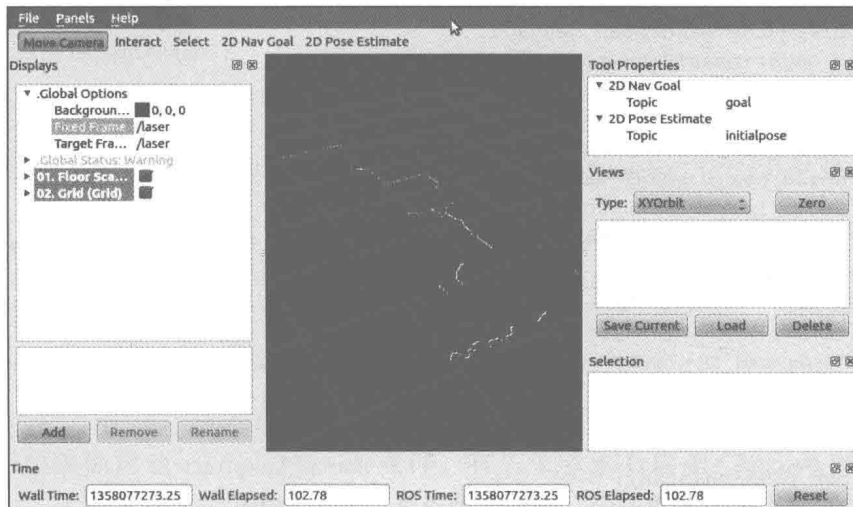
然后，会看到激光雷达发出的消息如下：

```
---
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。在一个命令行窗口输入以下命令启动 `rviz`，并加载正确的配置文件：

```
$ rosrn rviz rviz -d `rospack find chapter4_tutorials`/rviz/laser.rviz
```

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，然后发布新的数据。这有时候会很有用，而等你学完这个示例，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 c4_example2.cpp 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 launch 文件

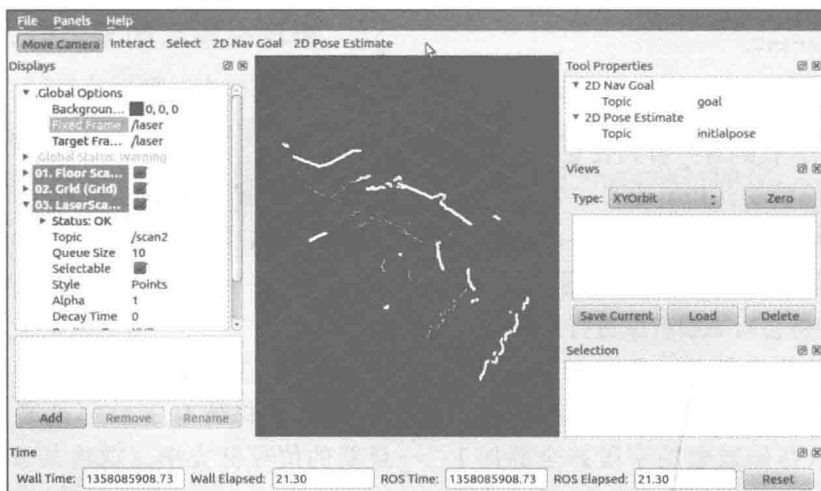
为了启动这一切，需要创建一个 launch 文件 example2.launch:

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

  <node pkg="chapter4_tutorials" type="c4_example2" name="c4_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 c4_example2。你将会在 rviz 的界面内看到两条激光轮廓。绿颜色的轮廓线是新数据产生的，

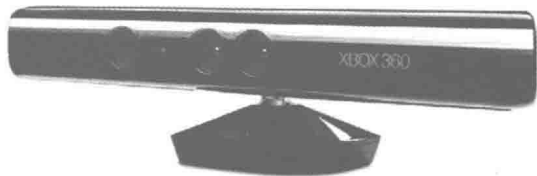
如下图所示：



4.3 使用 Kinect 传感器查看 3D 环境中的对象

Kinect 传感器是一个扁平的黑盒子，其下方是一个可以活动的小平台，能够固定在桌子上或者你使用 Xbox 360 的电视机架子上。这个设备上面有三种传感器能够帮助我们完成视觉和机器人任务：

- 一个彩色 VGA 视频摄像头用来查看彩色的世界
- 一个深度传感器，一个红外外斑投影仪和一个单色 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多阵列麦克风



KINECT
for XBOX 360

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们：

```
$ sudo apt-get install ros-hydro-openni-camera ros-hydro-openni-launch
$ rosstack profile && rospack 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下面指令：

```
$ rosrund oppenni_camera oppenni_node
$ roslaunch oppenni_launch oppenni.launch
```


如果一切正常，你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已创建的主题：

```
$ 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`。要查看从传感器获取的图像，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：

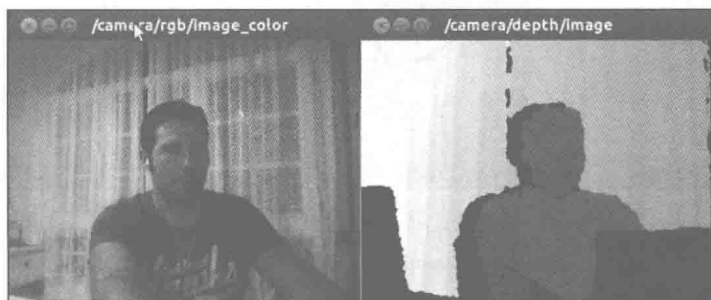
```
$ rosrun image_view image_view image:=/camera/rgb/image_color
```

注意，我们需要使用参数的图像将图像主题重命名（重映射）为 `/camera/rgb/image_color`。如果一切正常，会弹出一个新的窗口并显示来自 Kinect 的图像。

如果你想查看深度传感器，同样只需要在最后一个命令行中将主题名称改变一下：

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

你会在右侧界面中看到一幅和左侧图像类似的图像：



另一个重要的主题是发布点云数据的主题。这种数据是深度图像的 3D 展示形式。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 | rosmmsg show`。在本示例中，我们将会使用 `/camera/depth/points` 主题：

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

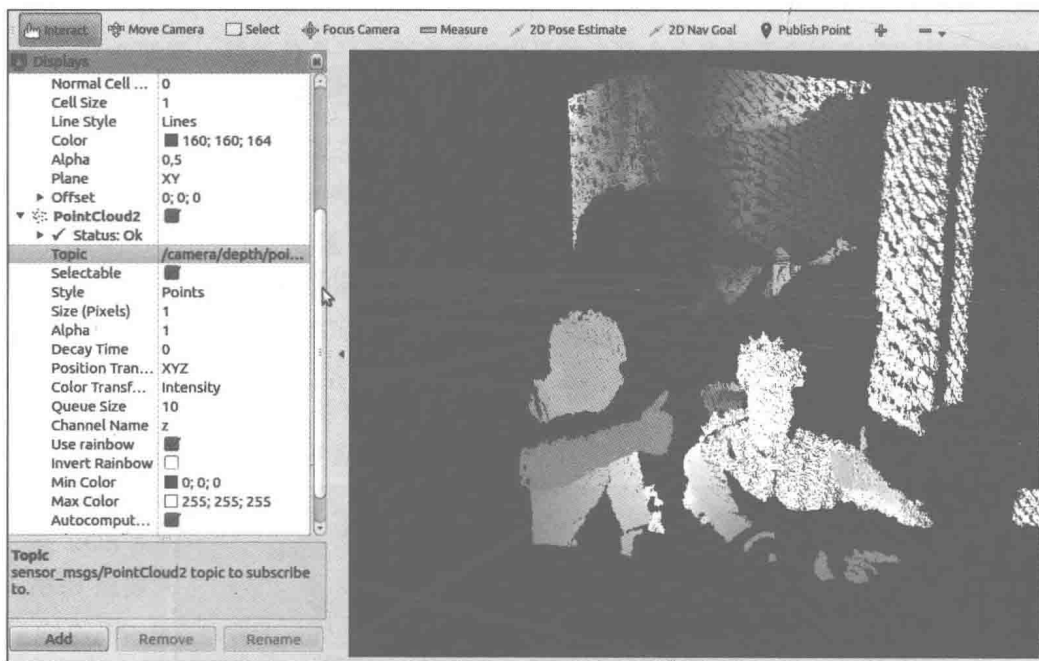
如果想查看消息的官方说明，请访问 http://ros.org/doc/api/sensor_msgs/html/msg/PointCloud2.html。

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：

```
$ rosrn rviz rviz
```

点击 **Add** 按钮，按显示类型订阅主题，并选择 `PointCloud2`。在添加 `PointCloud2` 显示类型时，必须选择 `camera/depth/points` 主题。

在你的电脑中，你能实时地看到 3D 图像。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 创建使用 Kinect 的示例

现在，我们将会使用一段程序来实现一个节点，它过滤来自 Kinect 传感器的点云数据。这个节点将会应用过滤器来减小原始数据中点云点的数量，从而减少采样的数据。

在 `chapter4_tutorials/src` 文件夹下创建一个新文件 `c4_example3.cpp`，并输入下面的代码段：

```

#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
// PCL specific includes
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>

#include <pcl/io/pcd_io.h>

ros::Publisher pub;

void cloud_cb (const pcl::PCLPointCloud2ConstPtr& input)
{
    pcl::PCLPointCloud2 cloud_filtered;
    pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
    sor.setInputCloud (input);
    sor.setLeafSize (0.01, 0.01, 0.01);
    sor.filter (cloud_filtered);
    // Publish the data
    pub.publish (cloud_filtered);
}

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 ();
}

```

这个例子基于点云库 (Point Cloud Library, PCL) 的教程。可以参考 http://pointclouds.org/documentation/tutorials/voxel_grid.php#voxelgrid。

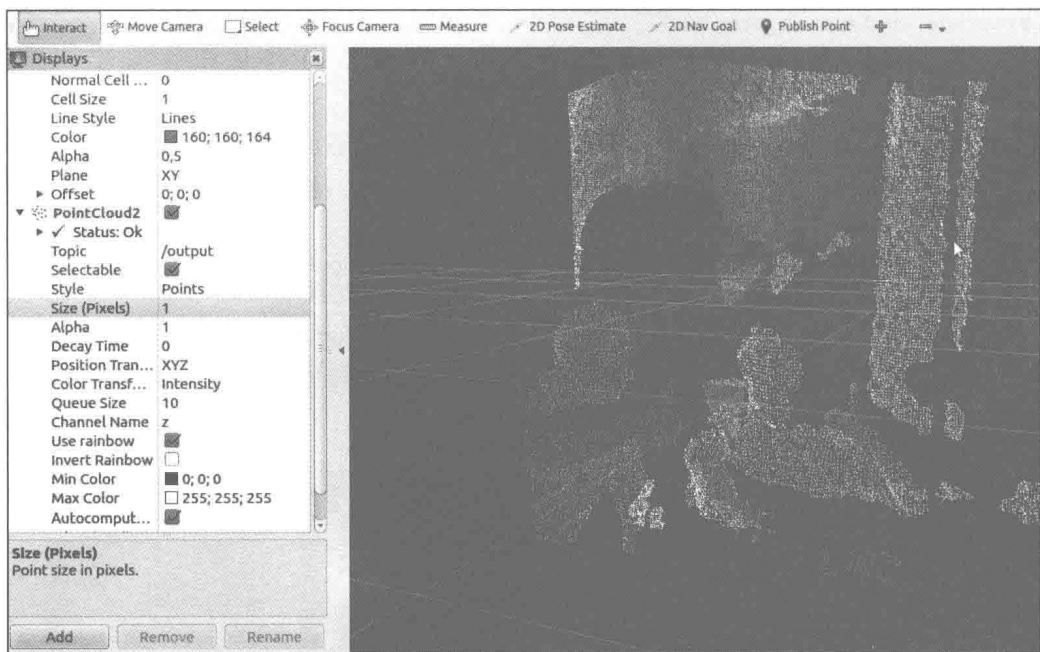
所有的工作都在 `cb()` 函数中完成, 当收到消息时会调用这个函数。我们创建一个 `VoxelGrid` 类型的变量 `sor`, 在 `sor.setLeafSize()` 中改变网格的大小。这些值会改变用于过滤器的网格参数。当增加这些值时, 在点云上会获得更低的分辨率和更少的点。

```

cloud_cb (const sensor_msgs::PointCloud2ConstPtr& input)
{
    ...
    pcl::VoxelGrid<sensor_msgs::PointCloud2> sor;
    ...
    sor.setLeafSize(0.01f, 0.01f, 0.01f);
    ...
}

```

当运行一个新的节点来打开 rviz 时, 将会在窗口中看到新的点云展示, 会很容易发现分辨率比原有的数据低了不少, 如下图所示:



在 rviz 中, 你能够看到一个消息中包含的点数量。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。

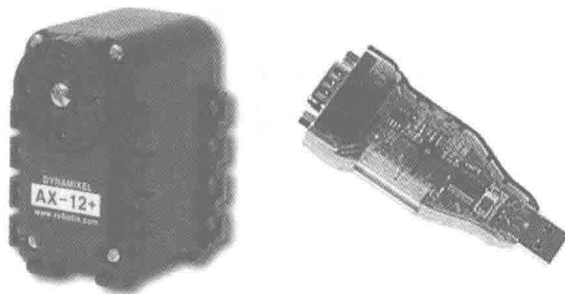
4.4 使用伺服电动机——Dynamixel

在移动机器人领域里, 伺服电动机可谓应用广泛。此类执行器主要用于移动传感器、轮子和机械臂。一种低成本解决方案是使用 RC 伺服电动机 (舵机)。它能够在 180° 的范围内运动, 并且提供较大的扭矩。

本节将会介绍一种专为机器人设计和使用的新型伺服电动机, 这就是 Dynamixel 伺服电动机。

Dynamixel 是由韩国生产商 ROBOTIS 公司开发的一种机器人专用的线性、高性能网络执行器。ROBOTIS 公司还是 OLLO、Bioloid 和 DARwIn-OP DXL 等机器人的设备研发和供应商。由于它们极佳的可扩展性、强大的位置、速度、内部温度、输入电压的反馈能力, 还有简单的菊花链拓扑简化了布线连接, 这些设备被众多的企业、大学和爱好者使用。

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



首先，我们将会安装必需的功能包和驱动。在命令行窗口输入下面命令：

```
$ sudo apt-get install ros-hydro-dynamixel-motor
$ rosstack profile && rospack profile
```

一旦安装完毕，将转换器连接到计算机并检查它能否被检测到。通常情况下，它会在 `/dev/` 文件夹下创建一个以 `ttyUSBX` 为名的新接口。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动机吧。

在命令行窗口中启动 `roscore`，然后在另外一个窗口中输入下面命令：

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

如果你连接了一个或多个电动机，那么你会看到电动机被驱动检测到。在该示例中，检测到了一个名为 ID 6 的电动机，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.
```

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。记住请使用下面命令行查看主题：

```
$ rostopic list
```

这些主题会显示配置好的电动机状态。

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

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

这个 launch 文件会创建驱动电动机所必需的主题:

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

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

```
/diagnostics
/motor_states/pan_tilt_port
/rosout
/rosout_agg
/tilt_controller/command
/tilt_controller/state
```

我们使用 /tilt_controller/command 通过 rostopic pub 命令发布主题来驱动电动机。首先查看主题的字段和类型, 需要输入下面命令行:

```
$ 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动机。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, 你会看到电动机转动, 但是它会停止在 0.5rad 或 28.647 889 8° 上。

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动机。在你的 /chapter4_tutorials/src 文件夹下创建一个新 c4_example4.cpp, 并输入下面代码:

```
#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();
    }
}
```

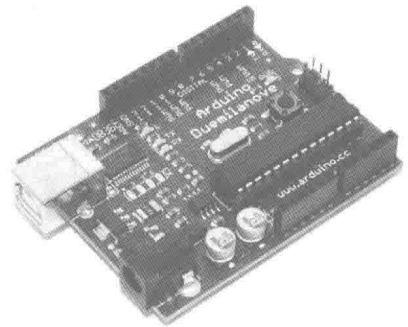
这个节点会驱动电动机在 $-180^\circ \sim 180^\circ$ 之间不断运动。这是一个非常简单的例子，但你能够使用它进行复杂的运动或者控制更多的电动机。我们假设你能够理解这些代码，所以没有必要进行过多的解释。请注意你是在向 `/tilt_controller/command` 主题发布数据，这就是电动机的名称。

4.5 使用 Arduino 添加更多的传感器和执行器

Arduino 是一种由灵活、易于开发的软硬件组成的开源电子设计平台。它用于艺术家、设计人员、爱好者和所有喜爱创造交互式物体和环境的人。右图是一个 Arduino 开发板：

ROS 能通过 `roserial` 功能包使用此类设备。通常 Arduino 使用串口和电脑连接，数据传输也是通过串口。你可以通过 `roserial` 使用大量支持串口连接的设备，例如 GPS、伺服控制器等。

首先输入下面命令，安装功能包：



```
$ sudo apt-get install ros-hydro-rosserial-arduino
$ sudo apt-get install ros-hydro-rosserial
```

然后到 catkin 工作空间，先复制 rosserial 库。用下面代码创建 rosserial 消息并编译 ros_lib:

```
$ cd ~/dev/catkin_ws/src/
$ git clone https://github.com/ros-drivers/rosserial.git
$ cd ~/dev/catkin_ws/

$ catkin_make
$ catkin_make install
$ source install/setup.bash
```

好的，我们假设你已经安装了 Arduino IDE。如果还没有，请根据 <http://arduino.cc/en/Main/Software> 描述的步骤进行安装。对于 Ubuntu 12.04，可以通过下面命令进行安装：

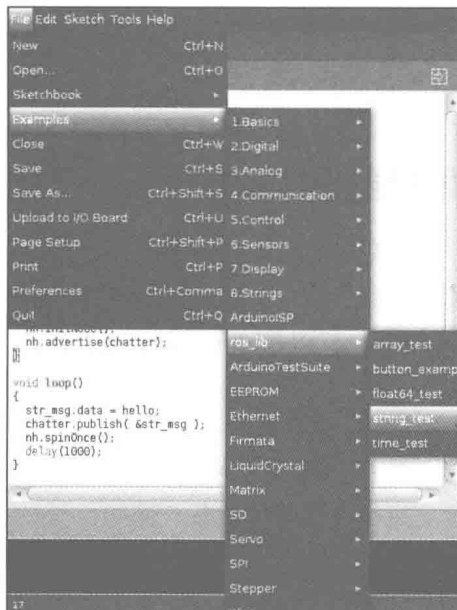
```
$ sudo apt-get update && sudo apt-get install arduino arduino-core
```

当你将所有功能包和 IDE 都安装好，就能够从 rosserial 功能包复制 ros_lib 到 sketchbook 库文件夹下。这个库的文件夹会在你的计算机运行 Arduino IDE 之后创建。然后，需要运行 make_libraries.py:

```
$ cd ~/sketchbook/libraries
$ rosrn rosserial_arduino make_libraries.py .
```

创建使用 Arduino 的示例

现在我们将从 IDE 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。其中，在 `include` 行中包含了 `ros.h` 头文件。它就是之前安装的 `rosterial` 库。还能够通过主题查看一个库发送的消息，在本例中是 `std_msgs/String` 类型。

下面代码都保存在 `c4_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);
}
```

Arduino 代码分为两个函数：`setup()` 和 `loop()`。`setup()` 函数执行一次，通常用于开发板配置初始化，之后 `loop()` 函数连续运行。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下面的命令，记得运行 `roscore`。

```
$ rosrund rosterial_python serial_node.py /dev/ttyACM0
```

现在可以通过 `rostopic echo` 命令看到 Arduino 发出的消息：

```
$ rostopic echo chatter
```

在命令行窗口中看到下面数据：

```
data: chapter4_tutorials
```

最后的示例是从 Arduino 向计算机发送数据。现在我们将使用一个示例，其中 Arduino 会订阅一个主题并改变连接到 13 号引脚上的 LED 灯状态。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 `c4_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);
}

```

注意启动节点与 Arduino 板进行通信：

```
$ rosrund rosserial_python serial_node.py /dev/ttyACM0
```

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，你需要再次发布主题：

```
$ rostopic pub /toggle_led std_msgs/Empty "{}" -once
```

publishing and latching message for 3.0 seconds

现在你能够在 ROS 中使用所有 Arduino 的设备了。这非常有用，因为你可以使用各种廉价的传感器和执行器装备你的机器人了。



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。因此在使用本功能包之前，最好小心你所选的设备。

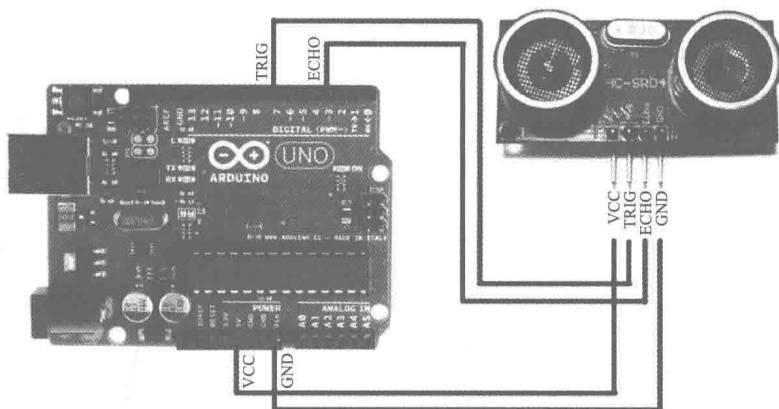
我们在使用 Arduino UNO R3、Arduino Mega、Arduino Duemilanove 或 Arduino Nano 的时候，没有出过任何问题。

4.6 在 Arduino 上使用超声波传感器

使用一个 I/O 开发板，如 Arduino，可以连接多个传感器和执行器。在本节中，我们将对 Arduino 编程来控制 HC-SR04，将超声波距离传感器用于 ROS。HC-SR04 是一款低成本传感器，通常用于测量机器人和障碍物之间距离。

这个设备包括一个印刷电路板（PCB）与两个类似麦克风和压电耳机的压电转换器，但工作在超声波范围。其中一个传感器是发射器，另一个是接收器。这种传感器测量距离的原理基于动物的回声定位系统，如蝙蝠。传感器会发出超声波，如果遇到障碍，超声波会反射，它将返回传感器。根据空气中的声速，我们将测量声音发射和接收之间的时间差来计算障碍物的距离。

HC-SR04 传感器有四个引脚，VCC 将连接到 5V，将 GND 连接到 GND，触发器（TRIG）将连接到 7 号引脚，ECHO 连接到 3 号引脚。触发器用来激活发射超声波。当我们在 7 号引脚中输出逻辑高电平，传感器会发出超声波。如果传感器接收超声波，回声引脚电压将在一个较高的水平，如下图所示：



使用 Arduino IDE 打开 c4_examples5_2.ino 并上传到开发板。代码如下：

```
#include <ros.h>
#include <std_msgs/Int16.h>

ros::NodeHandle nh;
std_msgs::Int16 range;
ros::Publisher range_pub("range", &range);
const int trigpin = 7;
const int echopin = 3;
long duration = 0;

void setup()
{
    nh.initNode();
```

```

    nh.advertise(range_pub);
}

void loop()
{
    range.data = ping();
    range_pub.publish(&range);
    nh.spinOnce();
    delay(100);
}

long ping()
{
    // Send out PING)) signal pulse
    pinMode(trigpin, OUTPUT);
    pinMode(echopin, INPUT);
    digitalWrite(trigpin, LOW);
    delayMicroseconds(2);
    digitalWrite(trigpin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigpin, LOW);
    //Get duration it takes to receive echo
    duration = pulseIn(echopin, HIGH);
    //Convert duration into distance
    return duration /29/2;
}

```

这里包括 `ros_lib` 和 `std_msgs/Int16` 库, `ros` 句柄声明和 `std_msgs::Int16` 命名为 "range"。在这个主题中, `Arduino` 将发送最近障碍的距离信息:

```

std_msgs::Int16 range;
ros::Publisher range_pub("range", &range);

```

在 `setup()` 函数中, 初始化 `ROS` 节点并发布消息, 如下所示:

```

nh.initNode();

nh.advertise(range_pub);

```

在 `Arduino` 的 `loop()` 函数, `ping()` 函数被调用来管理超声波传感器、距离计算, 然后以 `cm` 为单位发布距离信息。100 毫秒的延迟周期, 即以接近 10Hz 速度生成数据:

```

range.data = ping();

range_pub.publish(&range);

nh.spinOnce();

delay(100);

```

`ping()` 函数声明引脚 7 作为数字输出以及引脚 3 为数字输入。我们在引脚 7 输出低电平以确保没有声音发出并保持通道的清洁。然后, 写一个数字高电平持续 10 微秒。

之后, Arduino 的 `pulseIn()` 函数测量时间, 直到引脚处于高电平, 表明接收到超声波。这个函数返回以 `ms` 为单位的时间。脉冲的持续时间乘以声音的速度来计算距离。需要注意的是距离除以 2, 因为声音脉冲经过的路径是距离的 2 倍。首先, 它遇到障碍, 然后回到传感器。

启动这个节点, 我们需要在终端运行 `roscore`。在一个新的终端, 我们将运行 `rosserial` 节点与 Arduino 通信:

```
$ rosrund rosserial_python serial_node.py /dev/ttyACM0
```

现在, 我们为在 ROS 中使用超声波传感器做好了准备。

有趣的是在水下机器人领域, 声呐是非常基础的技术之一, 它扮演的角色与地面机器人中的激光测距传感器相同。由于声音在水中传播特性比光更好, 因此在水下往往采用声呐传感器来检测障碍、构建地图以及确定位置。

4.7 距离传感器如何发送消息

如果你上传的 Arduino 代码正确, 并且正常启动 `serial_node.py`, 我们可以在主题列表中看到主题名 `/range`:

```
$ rostopic list
```

可以像往常一样使用 `rostopic` 查看主题的类型:

```
$ rostopic type /range
std_msgs/Int16
```

使用下面命令查看消息中包含的数据:

```
$ rostopic echo /range
```

你可以使用这个传感器, 将它指向不同距离的对象, 查看距离信息。

4.7.1 创建使用超声波的示例

在本例中, 超声波传感器将指挥 `turtlesim` 节点的海龟运动。例子中程序的行为取决于对象的距离, 在 `turtlesim1/cmd_vel` 发布 `geometry_msgs/Twist`, 用这种方式实现避障。当超声波发现一个物体距离太近, 即不到 20cm, 海龟会后退。如果障碍物在一个适中的距离, 即 20cm 到 40cm, 海龟会转圈。如果障碍未找到或发现距离超过 40cm, 海龟会继续前进。你可以打开 `c4_example5_2.cpp` 查看代码:

```
#include<ros/ros.h>
#include<geometry_msgs/Twist.h>
#include<std_msgs/Int16.h>
#include<iostream>
using namespace std;
```

```

ros::Publisher pub;
ros::Subscriber sub;

void rangeCallBack(const std_msgs::Int16 &range)
{
    geometry_msgs::Twist vel;
    if (range.data > 40)
    {
        vel.angular.z = 0;
        vel.linear.x = 1;
    }
    else if (range.data > 20)
    {
        vel.angular.z = 1;
        vel.linear.x = 0;
    }
    else
    {
        vel.angular.z = 0;
        vel.linear.x = -1;
    }
    pub.publish(vel);
}

int main(int argc, char** argv)
{
    ros::init(argc, argv, "turtle_ultra_sound");
    ros::NodeHandle n;
    pub = n.advertise<geometry_msgs::Twist>("/turtle1/cmd_vel",1);
    sub = n.subscribe("range", 10, &rangeCallBack);
    ros::spin();
}

```

在 main() 函数，ROS 初始化并声明发布者和订阅者。发布者 geometry_msgs::Twist/turtle1/cmd_vel 控制 turtlesim 节点。订阅者数据来自于超声波传感器，每当 /range 消息发布，/rangeCallBack 函数将运行：

```

ros::init(argc, argv, "turtle_ultra_sound");
ros::NodeHandle n;
pub = n.advertise<geometry_msgs::Twist>("/turtle1/cmd_vel",1);
sub = n.subscribe("range", 10, &rangeCallBack);
ros::spin();

```

在 /rangeCallBack 需要输入 std_msgs::Int16 类型的数据，但程序中声明的类型是 geometry_msgs::Twist。根据数据范围，将不同的值分配给 /vel 模拟三个行为来避障：

```

void rangeCallBack(const std_msgs::Int16 &range)
{
    geometry_msgs::Twist vel;

```

```
if (range.data > 40)
{
    vel.angular.z = 0;
    vel.linear.x = 1
}
...
pub.publish(vel);
}
```

运行 `roscpp`、`turtlesim` 节点，然后是 `c4_example5_2`。在不同的终端分别运行下面命令来使用这个示例：

```
$ roscore
$ roslaunch roscpp_serial_node serial_node.py /dev/ttyACM0
$ roslaunch chapter4_tutorials c4_example5_2
$ roslaunch turtlesim turtlesim_node
```

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载体的速度、方向和重力的电子设备，它组合使用加速度计和陀螺仪，甚至是磁场强度计等传感器。IMU 最典型的应用是操纵飞行器，包括无人机 (UAV)；以及空间飞船，包括各种卫星和着陆器。

——维基百科

在右侧的图片中，你会看到 Xsens MTi，它是本节的主角。

在本节中，我们将会学习 Xsens MTi 在 ROS 中的工作方式以及如何通过传感器发布主题。这里有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。它需要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，你需要下载下面的驱动：

- MicroStrain 3DM-GX2 IMU 可使用 http://www.ros.org/wiki/microstrain_3dmgx2_imu
- 带有 Wii Motion Plus 的 Wiimote 可使用 <http://www.ros.org/wiki/wiimote>

如果使用这个设备，我们需要 `xsens_driver`。你可以通过下面命令进行安装：

```
$ sudo apt-get install ros-hydro-xsens-driver
```

我们需要同时安装两个功能包，因为下载的驱动需要依靠它们运行。

```
$ roscpp profile
$ rospack profile
```

现在，我们将会开始使用 IMU 并学习它如何工作。在一个命令行窗口中输入下面命令：

```
$ roslaunch xsens_driver xsens_driver.launch
```



这个驱动将直接检测 USB 端口和带宽速率。

4.7.2 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，需要使用下面命令行：

```
$ rostopic type /imu/data
$ rostopic type /imu/data | rosmmsg show
```

`/imu/data` 主题是 `sensor_msgs/Imu`。这些字段用以指明方向、加速度和速度。在我们的示例中，将会使用方向字段。先查看一个消息学习数据发送的实例。你可以通过下面命令来查看：

```
$ rostopic echo /imu/data
```

你将看到如下类似的显示：

```
---
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，特殊的方向需要用四元数来表示。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。

我们可以在 `rviz` 中运行并加入 `imu` 的显示类型去观察 `imu` 方位：

```
$ rosruncv rviz
```

4.7.3 创建使用 Xsens 的示例

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，需要从四元数中提取

数据并将其转换为欧拉角（roll、pitch 和 yaw）并以在 x 轴和 y 轴的旋转值（滚转和俯仰）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类似。在 chapter4_tutorials/src/ 文件夹下创建一个新的文件 c4_example6.cpp，并输入下面代码：

```
#include<ros/ros.h>
#include<geometry_msgs/Twist.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<geometry_msgs::Twist>("/turtle1/cmd_vel",1);
    sub = n.subscribe<sensor_msgs::Imu>("imu/data", 10,
    &TeleopImu::callBack, this);
}

void TeleopImu::callBack(const sensor_msgs::Imu::ConstPtr& imu)
{
    geometry_msgs::Twist 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.z = roll;
    vel.linear.x = 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<geometry_msgs::Twist>("turtle1/cmd_vel",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 `geometry_msgs/Twist` 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。下面的代码是对该段的描述:

```
void TeleopImu::callBack(const sensor_msgs::Imu::ConstPtr& imu)
{
  ...
  tf::Matrix3x3(bq).getRPY(roll,pitch,yaw);
  vel.angular.z = roll;
  vel.linear.x = pitch;
  pub.publish(vel);
}
```

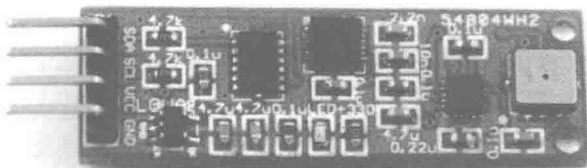
通过 `Matrix3x3` 类实现四元数到欧拉角的转换。然后, 我们将会使用这个类中提供的 `getRPY` 访问方法。通过这个方法, 我们能够从矩阵中获得相对于 x 、 y 和 z 轴的滚转、俯仰和偏航值。

在这之后我们只需要将滚转和俯仰的值分别给赋给线速度和角速度变量, 此时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, 就像用游戏杆的效果。

4.8 使用 10 自由度低成本惯性测量模组 IMU

在本节中, 我们将会学习使用一种具有 10 自由度 (DoF) 的低成本 IMU 传感器。这个传感器和 Xsens MTi 非常相似, 都有加速度计 ($\times 3$)、磁场强度计 ($\times 3$)、气压计 ($\times 1$) 和陀螺仪 ($\times 3$)。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>)。

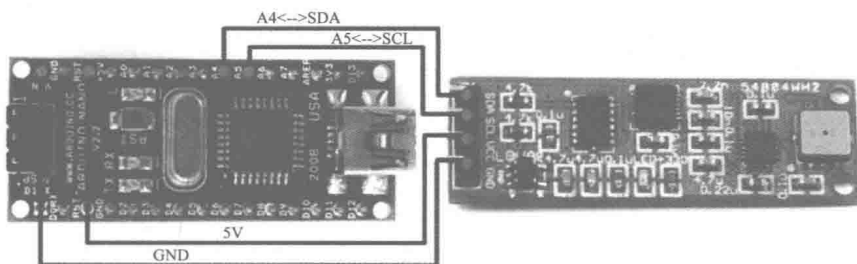
这个传感器的用途也非常相似。购买 Xsens 大概要花费 \$2000, 对于一般的用户来说是比较昂贵的, 而我们本节中要介绍的传感器只需要大约 \$20, 这个传感器亲民的价格让它能够用在很多项目上。通过下面的图片你能看到这个传感器。它很薄并且只有十几个元件。



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 $\pm 16g$ 。这个传感器广泛地用于移动设备终端。它能够测量静态的重力加速度用于倾角测量用途，同时也能够测量由于运动和冲击造成的动态加速度。
- HMC5883L：这个传感器用来测量低磁场强度，并配有数字接口，如低成本罗盘和磁强计等。
- BMP085：这是用于移动应用的先进高精度大气压力传感器。它性能强大，具有最高 0.03hPa 的绝对精度和 $3\mu A$ 的超低功耗。
- L3G4200D：这是高精度（16-bit）三轴陀螺仪，测量范围高达 2000 度每秒（dps）。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 协议进行控制。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 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 4 号引脚，串口时钟（SCK）需要连接到模拟量 5 号引脚。如果这些引脚连接错误，Arduino 就无法和传感器通信。

4.8.1 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 10 DOF 传感器需要的所有库。在每一个库里，你都能找到如何使用传感器的示例。

4.8.2 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 `c4_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);
}
```

让我们逐行分析代码看看如何提取并发布数据。

我们将会使用两个 Float32 变量来发布数据，因此首先在程序中包含下面头文件：

```
#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下面代码完成初始化：

```
Accel.set_bw(ADXL345_BW_12);
```

通过 get_Gxyz 返回的数据存储在一个三维向量中。

```
double acc_data[3];  
Accel.get_Gxyz(acc_data);
```

最后，发布数据：

```
velLinear_x_pub.publish(&velLinear_x);  
velAngular_z_pub.publish(&velAngular_z);
```

4.8.3 创建 ROS 节点以使用 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和生成在 turtlesim 中移动小海龟的指令，目标是使用 10 DOF 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 c4_example8.cpp，并输入下面代码：

```
#include<ros/ros.h>
#include<geometry_msgs/Twist.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;
    geometry_msgs::Twist vel;
};

TeleopImu::TeleopImu()
{
    pub = n.advertise<geometry_msgs::Twist>("turtle1/cmd_vel",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.x = -1 * wz->data;
    pub.publish(vel);
}

void TeleopImu::velLinearCallBack(const std_msgs::Float32::ConstPtr&
vx){
    vel.angular.z = 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 `c4_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 2 个由 Arduino Nano 创建的主题，而总共使用 3 个主题一起控制 `turtlesim`。

我们必须订阅 `/velLinear_x` 和 `velAngular_z` 主题，通过下面代码可以实现：

```
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 `geometry_msgs/Twist` 消息并发布出去。

```
void TeleopImu::velAngularCallBack(const std_msgs::Float32::ConstPtr&
wz){
    vel.linear.x = -1 * wz->data;
    pub.publish(vel);
}

void TeleopImu::velLinearCallBack(const std_msgs::Float32::ConstPtr&
vx){
    vel.angular.z = vx->data;
    pub.publish(vel);
}
```

为了运行这个示例，请记住编译代码并按照下面步骤执行：

1. 在新的命令行窗口启动 `roscore`。连接 Arduino 至电脑并启动下面命令：

```
$ rosrund rosserial_python serial_node.py
```

2. 现在，通过输入下面命令启动 `turtlesim`：

```
$ rosrund turtlesim turtlesim_node
```

3. 最后，输入下面命令启动节点：

```
$ rosrund chapter4_tutorials c4_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。

4.9 GPS 的使用

全球定位系统 (GPS) 是一种基于空间的系统，它能在任何天气、地球表面及其附近的任何地区提供关于位置和时间信息。你必须与四个全球定位系统卫星之间有畅通的通道才能获得有效数据。

通过 GPS 获得的数据符合由 NMEA 建立的通信标准和不同类型的协议。在这里，我们可以得到接收器的位置信息。查阅 NMEA 所有信息的类型你可以访问 <http://www.>

gpsinformation.org/daled/nmea.htm。

GPS 中最有趣的信息是在 GGA 中，它们提供通用的 Fix 数据以及 GPS 的 3D 位置。下面是这个 sentence 和每一块的解释：

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
```

Where: (解释)

GGA Global Positioning System Fix Data (全球定位修正数据)

123519 Fix taken at 12:35:19 UTC (修正于)

4807.038,N Latitude 48 deg 07.038' N (纬度)

01131.000,E Longitude 11 deg 31.000' E (经度)

1 Fix quality: 0 = invalid (修正质量)

1 = GPS fix (SPS) (修正)

2 = DGPS fix (修正)

3 = PPS fix (修正)

4 = Real Time Kinematic (实时运动学)

5 = Float RTK

6 = estimated (dead reckoning) (2.3

feature) (估计航路推算)

7 = Manual input mode (手动输入模式)

8 = Simulation mode (仿真模式)

08 Number of satellites being tracked (已跟踪卫星数)

0.9 Horizontal dilution of position (水平位置精度)

545.4,M Altitude, Meters, above mean sea level (高度, 米, 平均海平面)

46.9,M Height of geoid (mean sea level) above WGS84
ellipsoid (大地水准面高 (平均海平面) 高于 WGS84 椭球)

(empty field) time in seconds since last DGPS update

(自上一次 DGPS 更新时间间隔, 以秒为单位)

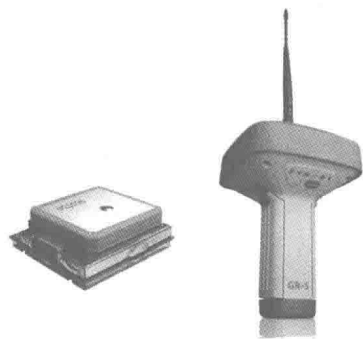
(empty field) DGPS station ID number (DGPS 基站 ID 号)

*47 the checksum data, always begins with * (校验和数据, 总是始于)

GPS 接收器不同，它的性能和精度也不同。通用廉价的 GPS 可以应用在不同的地方，比如说无人飞行器 (UAV)。它们的误差可能在几米。也可以用比较昂贵的 GPS 设备把它配制成差分 GPS，或者工作在实时动态 (RTK) 模式，第二个 GPS 在一个可知的位置给第一台 GPS 发送校正数据。这种 GPS 可以得到位置误差在 10cm 以下的信息。

GPS 通过串口协议向计算机或者微型控制器 (如 Arduino) 发送接收的数据。我们可以用 TTL、RS232 通过 USB 适配器很容易地与计算机相连。这一节，我们将用到廉价的 GPS(EM-406a) 和一个精确的系统，如在实时动态模式下的 GR-3 Topcon。将会看到在相同的驱动下，我们可以从这两个设备中得到经度、纬度和高度。

为了用 ROS 控制 GPS 传感器，我们将用下面的命令安装 NMEA GPS 的驱动包 (不要忘了在那之后运行 rosstack 和 rospack 配置文件)：




```
$ sudo apt-get install ros-hydro-nmea-gps-driver
$ rosstack profile & rospack profile
```

运行 `nmea_gpst_drivers.py` 文件，启动 GPS 驱动。同时我们要声明两个参数：与 GPS 相接的端口以及波特率。

```
$ rosrunc nmea_gps_driver nmea_gps_driver.py _port:=/dev/ttyUSB0 _
baud:=4800
```

EM-406a GPS 和 Topcon GR-3 的波特率分别是 4800Hz 和 115200Hz，如果我們想在 ROS 下用它们，必须像下面命令行中那样声明 `_baud` 参数。

```
$ rosrunc nmea_gps_driver nmea_gps_driver.py _port:=/dev/ttyUSB0 _
baud:=115200
```

4.9.1 GPS 如何发送信息

按照上面的步骤完成后，我们将在主题列表中看到一个叫 `/fix` 的主题：

```
$ rostopic list
```

我们用 `rostopic type` 命令查看使用的数据类型。NMEA GPS 将用 `sensor_msgs/NavSatFix` 消息发送 GPS 的状态信息。

```
$ rostopic type /fix
sensor_msgs/NavSatFix
```

`/fix` 主题是 `sensor_msgs/Navsat Fix`。`sensor_NavSatFix` 消息用来表明设备的经度、纬度、高度、状态、质量以及协方差矩阵。在此实例中，我们将利用经度和纬度来产生 2D 笛卡儿直角坐标系，这个方法被称为墨卡托方位法 (UTM)。

你可以用下面的命令查看发送数据的消息内容。

```
$ rostopic echo /fix
```

你将会看到这样的输出：

```
---
header:
  seq: 3
  stamp:
    secs: 1404993925
    nsecs: 255094051
  frame_id: /gps
status:
  status: 0
  service: 1
latitude: 28.0800916667
longitude: -15.451595
altitude: 315.3
```

```
position_covariance: [3.24, 0.0, 0.0, 0.0, 3.24, 0.0, 0.0, 0.0, 12.96]
position_covariance_type: 1
---
```

4.9.2 创建一个使用 GPS 的工程实例

在本例中，我们将 GPS 的经度和纬度映射到 2D 笛卡儿空间，我们将使用由 Chuck Gantz 写的函数将经度和纬度转换成 UTM 坐标。节点将订阅 GPS 发送数据的 /fix 主题。你可以在 chapter4_tutorials 的 c4_example8.cpp 文件中查看代码。

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <stdio.h>
#include <iostream>
#include <sensor_msgs/NavSatFix.h>
geometry_msgs::Point global_position;
ros::Publisher position_pub;
void gpsCallBack(const sensor_msgs::NavSatFixConstPtr& gps)
{
    double northing, easting;
    char zone;
    LLtoUTM(gps->latitude, gps->longitude, northing, easting, &zone);
    global_position.x = easting;
    global_position.y = northing;
    global_position.z = gps->altitude;
}
int main(int argc, char** argv){
    ros::init(argc,argv, "Geoposition");
    ros::NodeHandle n;
    ros::Subscriber gps_sub = n.subscribe("fix",10, gpsCallBack);
    position_pub = n.advertise<geometry_msgs::Point> ("global_position",
1);
    ros::Rate loop_rate(10);
    while(n.ok())
    {
        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

首先，使用 `#include <sensor_msgs/NavSatFix.h>` 定义 NavSatFix 消息。通过这种方式我们可以在 /fix 主题发布 `ros::Subscriber gps_sub = n.subscribe("fix", 10, gpsCallBack)` main 函数。所有的动作在 `gpsCallBack()` 函数中发生，我们将用 `LLtoUTM()` 函数将经度和纬度转换到 UTM 空间，并将发送一个名为 /global_position 的 geometry_msg/point 主题，/global_position 中包含 UTM 空间下的北向和东向坐标以及 GPS 的高度。

你可以在启动 GPS 驱动之后使用下面的命令来验证一下代码：

```
$ rosrun chapter4_tutorials c4_example8
```

4.10 本章小结

在机器人中使用传感器和执行器是非常重要的，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器，这些设备在全世界的机器人领域里广泛使用。可以确定，如果想使用其他类型的传感器，你都能够从网络和 ROS 文档中找到相关信息并轻松地使用。

在我看来，Arduino 是非常有趣的设备。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传感器的信息，绝对能够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有用的传感器。因为这是应用导航功能包集必须使用的设备。导航功能包集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 已经集成的 Gazebo 仿真环境中加载和使用它。

第 5 章

计算机视觉

ROS 对计算机视觉提供了基本的支持。首先，ROS 在处理视觉任务时提供给我们包含了用于各类摄像头和协议的驱动程序，尤其是 FireWire (IEEE1394a 或 IEEE1394b) 接口的摄像头。图像管道提供了摄像头标定过程、扭曲矫正、颜色解码和其他底层操作。对于更为复杂的任务，你可以使用 OpenCV、cv_bridge 和 image_transport 库与其连接，订阅和发布图像主题来进行处理。最后，还有一些能实现如物体识别、增强现实、视觉里程计等算法的功能包。

尽管 ROS 的功能包集成了 FireWire 摄像头的驱动，而对于 USB 和高速网络摄像头，支持这些协议也不困难。USB 摄像头不但价格低廉，而且也很容易买到，在本章中，我们将讨论这其中的一些，此外我们还提供了一个使用了 OpenCV 视频捕捉 API 的驱动，该驱动能够无缝集成于图像管道中。

我们会详细解释摄像头标定以及包含在图像管道中标定的结果。ROS 能够提供 GUI 来帮助我们使用标定模板进行摄像头标定。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。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从摄像头获取的源图像 (RAW image) 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 (debayer)。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，当然这对于低精度摄像头来说可能比较困难，而且有时候要以获得较高的标定结果为前提。

最后，通过 ROS 中的 OpenCV (只有 2.x 版本，3.x 版本还不支持)，我们能够应用大量的计算机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>）或者一些关于计算机视觉和 OpenCV 的书籍。这里，我们将通过一个示例，向你简单介绍如何在节点中使用 OpenCV，具体是用特征提取和匹配方法计算两幅图像之间的单应性。此外，本章将通过完成一个教程在 ROS 中设置并运行一个视觉里程计。viso2_ros 是 libviso2 视觉里程库的封装，其使用的双目视觉通过在一个支撑杆上固定两个便宜的网络摄像头实现。其他视觉里程计库也会讲到，例如 fovis。包括一些使用它们的建议以及如何提高 RGBD 传感器的效果，如 Kinect。甚至还有传感器融合或在单目视觉方面的一些信息。

5.1 连接和运行摄像头

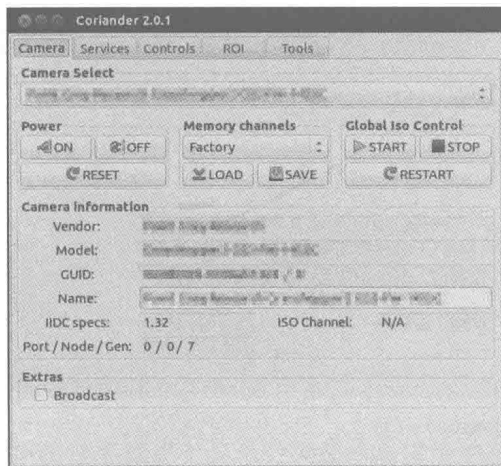
我们需要按一定的步骤进行操作，第一步是将摄像头与计算机连接，运行驱动并检查是否能在 ROS 中获取图像。其实在使用 ROS 之前，最好使用其他工具来检查摄像头是否被操作系统正确识别，在这里可以使用 Ubuntu 发行版。我们将从 FireWire 摄像头讲起，因为 ROS 系统对它的支持更好，然后再学习 USB 摄像头。

5.1.1 FireWire IEEE1394 摄像头

将摄像头连接到计算机上。计算机需要有 FireWire IEEE1394a 或者 IEEE1394b 插槽。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。然后再运行以下命令（在旧的 Ubuntu 发行版中，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 功能包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 下有时无法进行调试，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在工作。关闭 coriander 并使用以下命令运行 ROS FireWire 摄像头驱动（需要 roscore 运行）：

```
$ rosrun camera1394 camera1394_node
```

只需要先运行 roscore 和上面的指令即可。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 功能包的 GUI 上看到摄像头的 GUID。

FireWire 摄像头支持的参数在 camera1394/config/firewire_camera/format7_mode0.yaml 文件中被列出并赋值，如下所示：

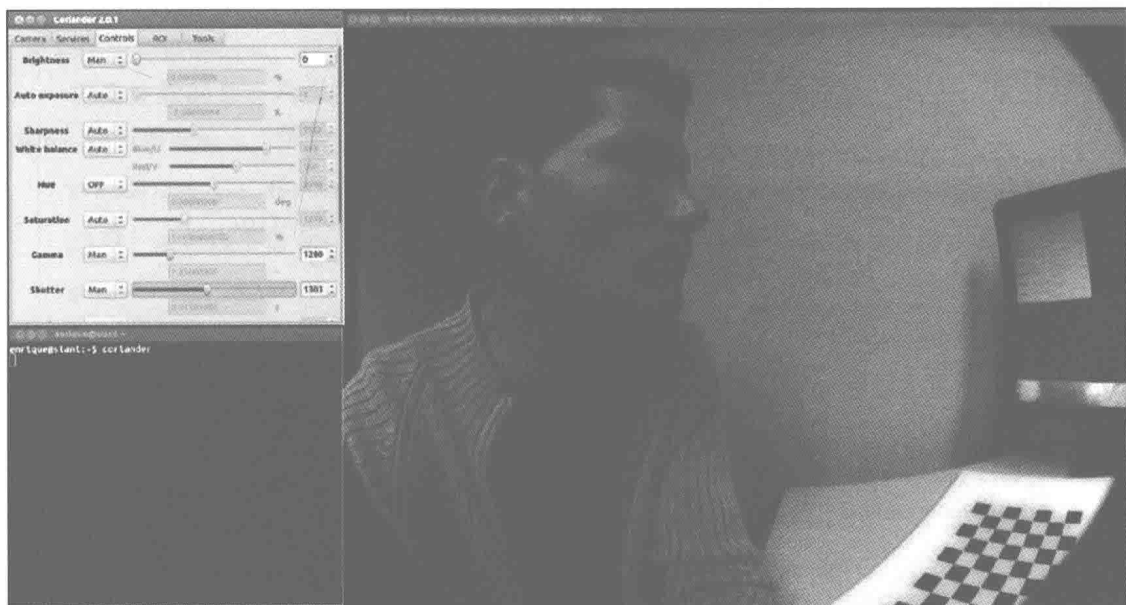
```
guid: 00b09d0100ab1324      # (defaults to first camera on bus)
iso_speed: 800 # IEEE1394b
video_mode: format7_mode0 # 1384x1036 @ 30fps bayer pattern
# Note that frame_rate is overwritten by frame_rate_feature; some useful
values:
# 21fps (480)
frame_rate: 21 # max fps (Hz)
auto_frame_rate_feature: 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://chapter5_tutorials/calibration/firewire_
camera/calibration_firewire_camera.yaml
```

这些值通过对所采集的图像进行观察获取。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相等或为其整数倍以避免闪烁。如果是在有阳光的户外，你可能担心的是设置合适的亮度值，这可以提高增益，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，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。正如你看到的，你的配置取决于所处环境的光线条件，而且你必须去适应它们。如果使用 `coriander` 或 `rqt_reconfigure` 界面还是比较容易做到的：

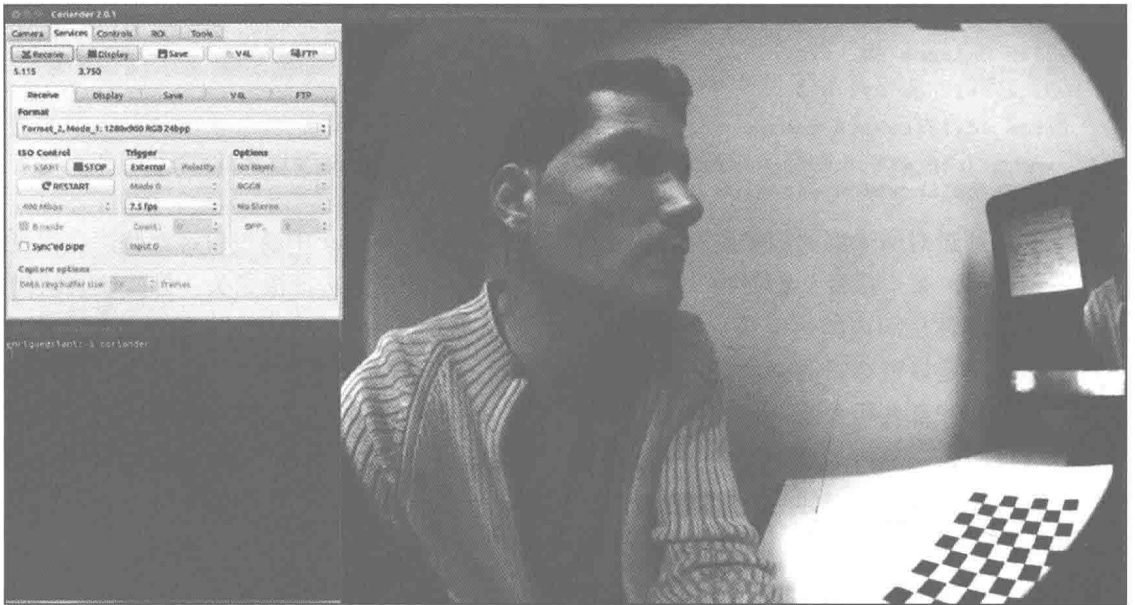
```
$ rosrn rqt_reconfigure rqt_reconfigure /camera
$ coriander
```

在下图中，我们能看到在 `coriander` 中的特定配置和输出图像：



为了更好地理解如何正确设置摄像头的参数来获得高质量的图像（也是算法友好的），可以多查阅关于摄影方面的基本概念、如曝光角度，它结合了快门速度、ISO 和光圈。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：



这里摄像头的命名空间是 /camera。然后可以按照第3章中介绍的方法对 camera1394 位于动态重配置 (cfg) 文件中的所有参数进行修改。为了使用方便，创建一个 launch 文件，它也在 launch/firewire_camera.launch 中

```
<launch>
  <!-- Arguments -->
  <!-- Show video output (both RAW and rectified) -->
  <arg name="view" default="false"/>
  <!-- Camera params (config) -->
  <arg name="params" default="$(find chapter5_tutorials)/config/
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）进行颜色修正后的图像。最后，建立条件图像组并使用 image_view（或 rqt_image_view）来可视化彩色图像和颜色修正后的图像。

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 chapter5_tutorials firewire_camera.launch view:=true
```

然后，你还能够使用 rqt_reconfigure 进行动态地配置。

5.1.2 USB 摄像头

现在我们将学习对 USB 摄像头进行相同的配置。问题在于 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计算机连接，可以先在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/video? 下，其中的 ? 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）

需要注意的是有两个主要的选项能用于 ROS 的 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：

```
$ sudo apt-get install ros-hydro-usb-cam
```

然后，运行以下命令：

```
$ roslaunch chapter5_tutorials usb_cam.launch view:=true
```

直接运行 rosrn usb_cam usb_cam_node，并使用 image_view（或者 rqt_image_view）显示摄像头图像，你会看到类似于下图所示的样子。这是 USB 摄像头的原始图像（RAW），彩色的。



类似地，其他好的选择还有 gscam，请使用以下命令安装：

```
$ sudo apt-get install ros-hydro-gscam
```

接着运行下面命令：

```
$ roslaunch chapter5_tutorials gscam.launch view:=true
```

对于 usb_cam，这个 launch 文件运行了 rosrn gscam 的 gscam，并且对摄像头的一些参数进行配置。它还使用 image_view（或者 rqt_image_view）显示了摄像头图像，如下图所示：



由 gscam 获取的参数（查看 config/gscam/logitech.yaml）如下：

```
gscam_config: v4l2src device=/dev/video0 ! video/x-raw-  
rgb,framerate=30/1 ! ffmpegcolospace  
frame_id: gscam  
camera_info_url: package://chapter5_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，其余的参数将会非常重要。

5.2 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一旦它们受摄像头固件支持，允许我们去修改它的一些参数。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。而在 gscam 中，我们可以进行更多的控制。我们可以在 ROS 中改变摄像头配置，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帧（Frames Per Second，FPS）进行轮询；第二个是我们能够根据像 FPS 的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 较大时，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 较低时，计时

器方法和轮询方法就差不多了，而前者的代码更清晰。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。

在功能包中，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 系统中以插件形式安装的各种编解码器，例如 `compressed` 和 `theora`。上面的 `cv_bridge` 字段用于从 OpenCV 图像到 ROS Image 消息的转换，其中有灰度 / 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 OpenCV(`opencv2`) 中的 `highgui` API。

这里我们会对 `src/camera_timer.cpp` 文件中主要部分的代码进行解释，其中有一个实现摄像头驱动类。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;
```

像往常一样，我们需要节点句柄。然后我们需要用于以任何可能的格式发送图像的 `ImageTransport` 对象。在代码中，我们只需要使用发布者 `Publisher`（唯一的），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（`images/frames`）。在捕捉帧的时候，我们直接使

用 `cv_bridge` 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/video?` 设备的编号，例如，0 代表 `/dev/video0`。这个索引名被传递给 `cv::VideoCapture`。fps 参数用于设定摄像头 FPS（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，如下所示：

```
camera( camera_index )
```

这里 `camera_index` 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 `image transport API` 作为 `image_raw` 图像（RAW 图像）的发布者，并初始化 `frame` 变量。图像捕捉的回调函数读取和发布图像如下所示：

```
camera >> frame->image;
if( not frame->image.empty() )
{
    frame->header.stamp = ros::Time::now();
    pub_image_raw.publish( frame->toImageMsg() );
}
```

上面的方法捕捉图像，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转换成 ROS Image 的图像。

使用以下代码启动这个节点：

```
$ rosrunk chapter5_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` 或 `rqt_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 chapter5_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。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理解。

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/video? 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 MJPEG，但对于大多数 USB 摄像头它们无法使用 OpenCV 进行修改）。
- `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）的增益。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，也就是设定了图像的亮度。通常通过调整增压和快门速度（在廉价摄像头中，这就是进入传感器光的积分时间）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 5.6 节也会用到它。
- `camera_info_url`：这个参数提供到摄像头信息的路径，所保存的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。当 `camera_index` 改变时，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 警告消息。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。

运行以下命令使用驱动：

```
$ roslaunch chapter5_tutorials camera.launch view:=true
```

它将会使用 `config/camera/webcam.yaml` 作为默认参数，其中设定了到目前为止所有动态再配置参数。

你能够使用 `rostopic list`、`rostopic hz/camera/image_raw` 和 `image_view`（或 `rqt_image_view`）来检查摄像头是否在工作。

我们使用了 ROS 中所有可用的资源来完成驱动程序，以便进行摄像头、图像与计算机

视觉等工作。为了能讲解得更清晰，我们将会下面的小节中分别解释每一项工作。

5.2.1 通过 cv_bridge 使用 OpenCV 处理 ROS 图像

假设有一个 OpenCV 图像，那么它会是一个 `cv::Mat` image。需要使用 `cv_bridge` 库将其转换为 ROS Image 消息并将其发布。可以选择分别使用 `CvShare` 或 `CvCopy` 去共享或者复制。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，应该按照以下方法进行初始化：

```
frame = boost::make_shared< cv_bridge::CvImage >();
```

如果事先知道图像的编码方式，使用以下代码：

```
frame->encoding = sensor_msgs::image_encodings::BGR8;
```

最后将 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。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。我们在下一小节介绍这些内容。

5.2.2 使用 image transport 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。`Image_transport` 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、压缩图像等多种类型。所支持的传输类型的数量由在 ROS 中安装的插件决定，最常用的是 `compressed` 和 `theora` 传输。你能够通过 `rostopic info` 命令来查看它们。使用下面命令安装所有插件：

```
$ sudo apt-get install ros-hydro-image-transport-plugins
```

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，前面小节中展示的 `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() );
```

5.2.3 在 ROS 中使用 OpenCV

ROS 使用安装在系统中的独立 OpenCV 库。为了能在节点中使用它，必须在 `package.xml` 文件中指定一个编译和运行需要的 `opencv2` 依赖包：

```
<build_depend>opencv2</build_depend>
<run_depend>opencv2</run_depend>
```

我们需要在 `CMakeLists.xml` 加入下面一行：

```
find_package(OpenCV)
include_directories(${catkin_INCLUDE_DIRS} ${OpenCV_INCLUDE_DIRS})
```

然后，对于每个使用 OpenCV 的库或可执行文件，我们必须在 `target_link_libraries` 中增加 `${OpenCV_LIBS}`（在 `chapter5_tutorials` 的 `CMakeLists.txt` 中）。

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 `cv` 命名空间，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 中的计算机视觉功能作介绍。

5.2.4 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` 节点或者 `rqt_image_view` 主题实现的：

```
$ rosrn 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 `rqtrviz` 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。

5.3 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，也有以一系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，圆圈看起来成了椭圆。检测算法会获取棋盘上每个单元格的角点，并用来估计摄像头的内参数和外参数。简单讲，外参数是摄像头的姿态，或者换句话说，就是当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，因为它不会发生改变，这就能在摄像头处于任何位姿时使用，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 chapter5_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，`scale` 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。这些指示条下面的三个按钮默认是禁用的，如下页图上所示：

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的点对摄像头模型进行求解和系统优化。如下页图下所示：



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>
```

标定工具只需要模板的特征（这里方格数量和大小分别是 8×6 个和 0.030 平方米）、图像主题、camera 的命名空间。

它也使用图像管道，但是并不必要。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 /tmp 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 子文件夹是 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 ZIP 文件中的 ost.txt 文件来查询标定结果。无论如何要记住在提交之后，标定矩阵和失真模型的系数会更新标定文件。试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://chapter5_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应该了解的知识（http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html）。

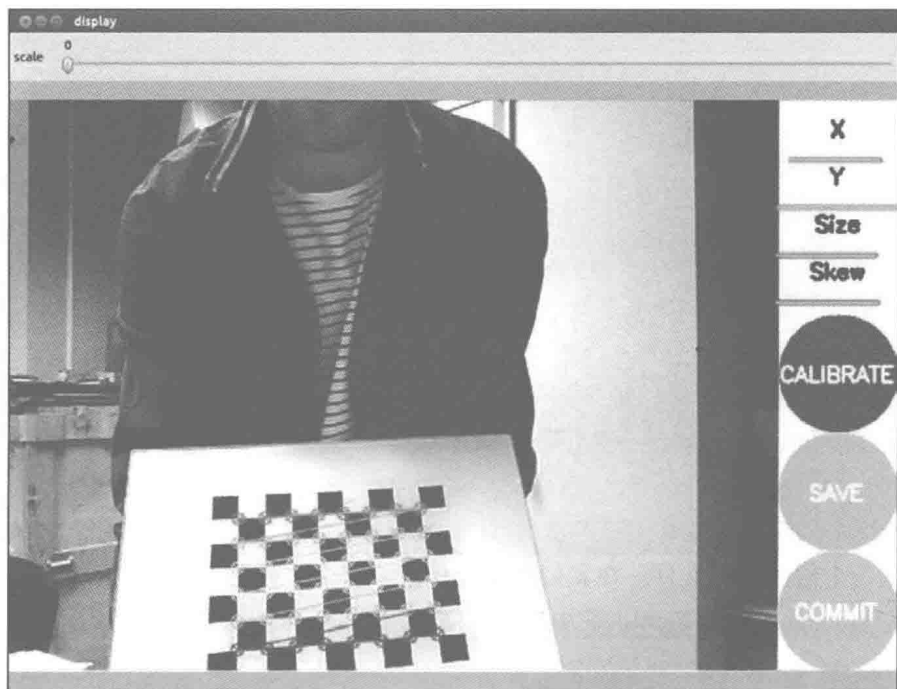
最后，你还可以使用下面的 launch 文件试着应用不同的标定图案，如圆形和不对称圆形（查看 http://docs.opencv.org/_downloads/acircles_pattern.png），这都针对 FireWire 摄像头：

```
roslaunch chapter5_tutorials calibration_firewire_camera_circles.
launch
roslaunch chapter5_tutorials calibration_firewire_camera_acircles.
launch
```

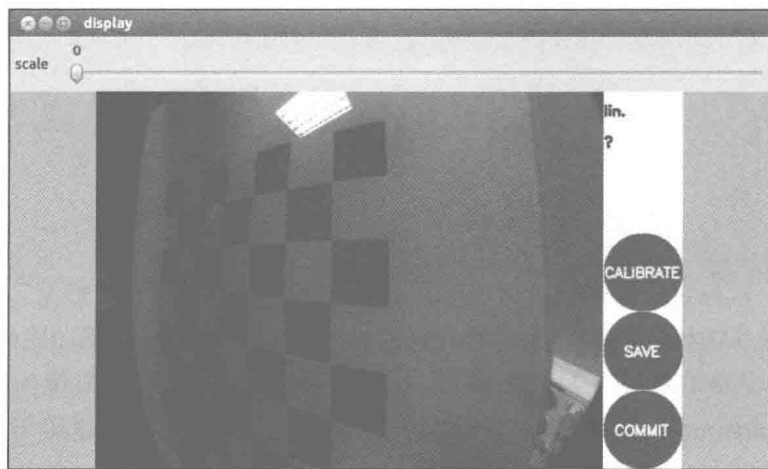
你还能在一次标定中使用多种棋盘图案和使用不同大小的图案。然而，我们认为仅使用一种有较好效果的棋盘图案就足够了。事实上，对于 USB 摄像头驱动我们也只这样做。在 USB 摄像头驱动的情况下，我们有更强大的 launch 文件来集成摄像头标定节点。这和 FireWire 摄像头一样，也是独立的启动文件。因此，要标定你的摄像头，请使用以下命令：

```
$ roslaunch chapter5_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。



点击标定 (CALIBRATE) 按钮之后, 标定优化算法会得到最好的摄像头内外参数。下图展示了标定过程结束后, 允许我们保存标定数据并提交到摄像头配置文件中, 而且我们不需要做更多的设置。



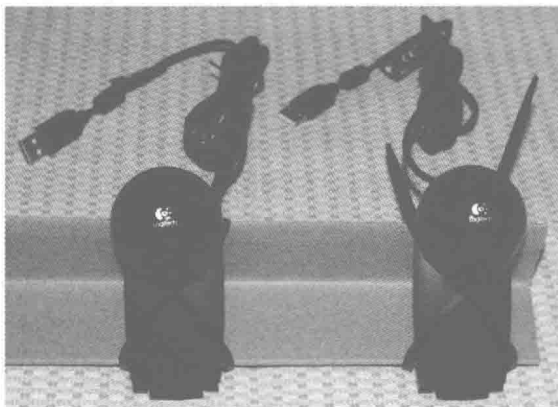
双目标定

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 网络摄像头，它的价格相对低廉，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并朝向平行的方向。我们的基线大约是 12cm，这个参数在双目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，并用扎带捆好：



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。这样它们就总能被分配成 `/dev/video0` 和 `/dev/video1`。当然，如果 0 被占用，就是 1 和 2。另外，你可以新建一个 udev 规则。

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。你可能需要有用的 video4linux 摄像头控制面板，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：

```
In case of problems with stereo:
```

```
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。如果只有一个 USB 控制器，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 MJPEG。你可以检查你的摄像头是否支持压缩：

```
$ v4l2-ctl -d /dev/video2 --list-formats
```

你会看到类似如下的输出：

```
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 MJPEG，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置到理想的像素格式。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。

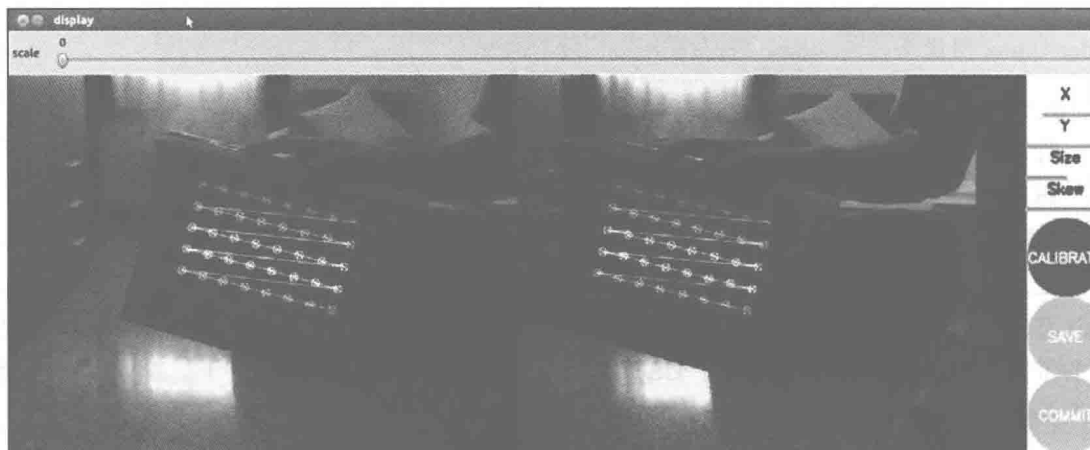
本书中附带的 USB 双目视觉驱动是基于前面章节介绍过的 USB 摄像头驱动。基本上，它扩展了对摄像头发布者的支持。发布者会发送左边和右边的图像，并附带摄像头信息。你能够运行它并使用以下命令查看图像：

```
$ roslaunch chapter5_tutorials camera_stereo.launch view:=true
```

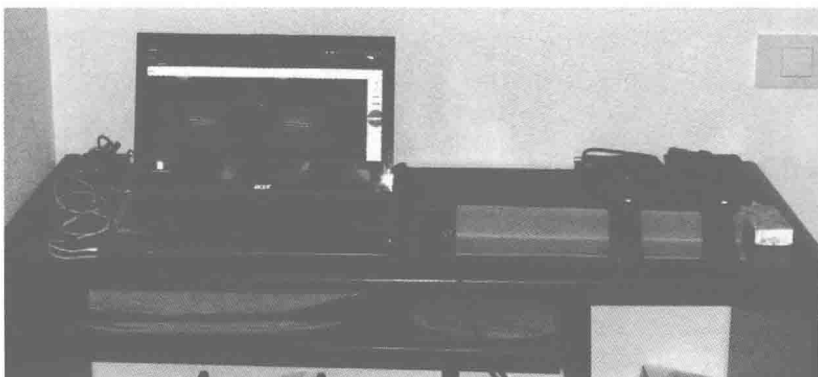
这也显示左右摄像头的视差图像。视差图像是非常重要的，我们会在后面专门讨论。一旦摄像头被标定好，就能够使用 ROS 图像管道。为了能够标定，我们需要运行以下命令：

```
$ roslaunch chapter5_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。在下图你能看到本书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 `config/camera_stereo/logitech_c120.yaml` 中的参数文件，我们有：

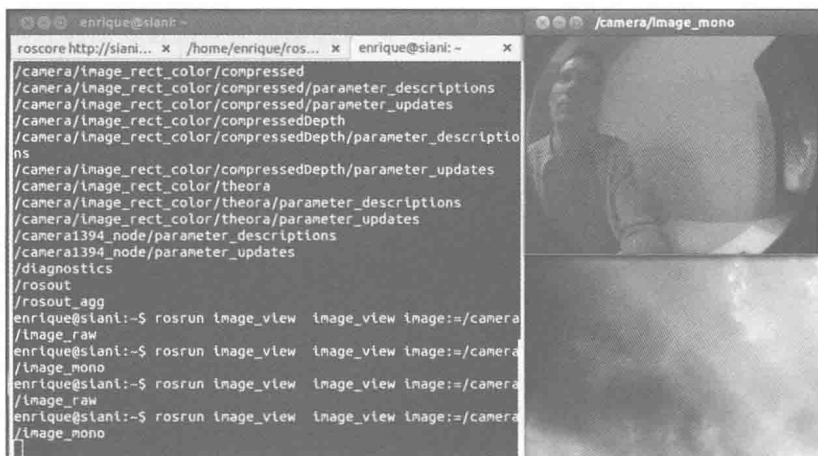
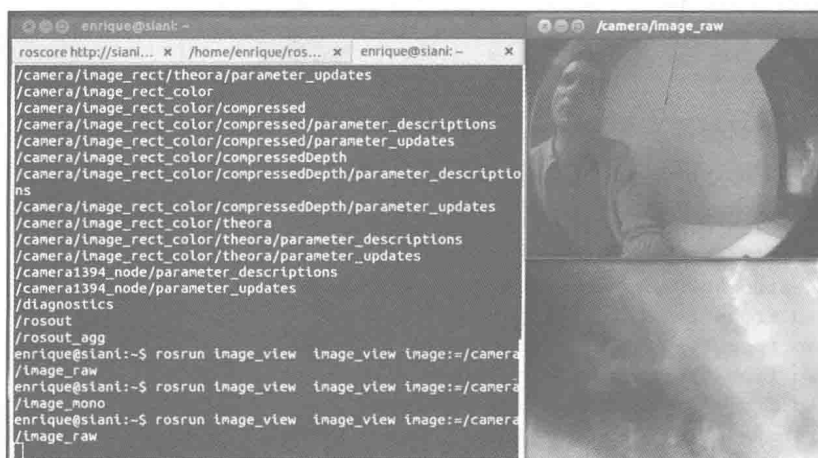
```
camera_info_url_left: package://chapter5_tutorials/calibration/camera_
stereo/${NAME}.yaml
camera_info_url_right: package://chapter5_tutorials/calibration/
camera_stereo/${NAME}.yaml
```

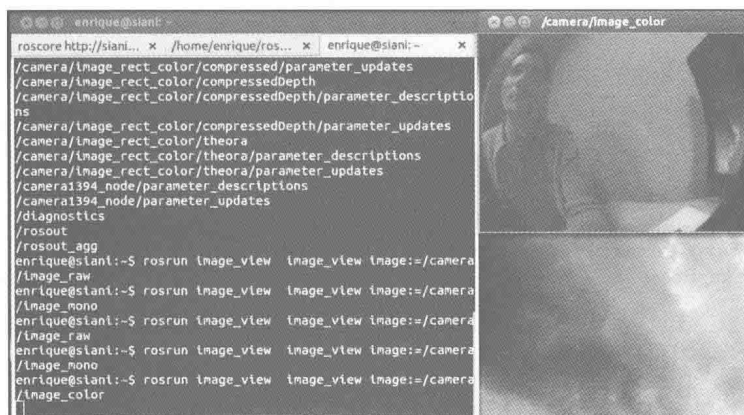
这里，`${NAME}` 是摄像头的名称，分别代表 `logitech_c120_left` 和 `logitech_c120_right`，也就是左右摄像头。在提交标定之后，会将每个摄像头的标定更新到这些文件。这包含了标定矩阵、失真模型参数和透视投影矩阵，其中还包含基线。基线表示的是在图像平面 x 轴方向上两个摄像头的距离。在参数文件中，你还能看到一些用来设定人工灯光的室内环境的摄像头属性值。这个摄像头是自动校正的，所以有时候图像质量比较差，但大多数情况下还是可以接受的。

5.4 ROS 图像管道

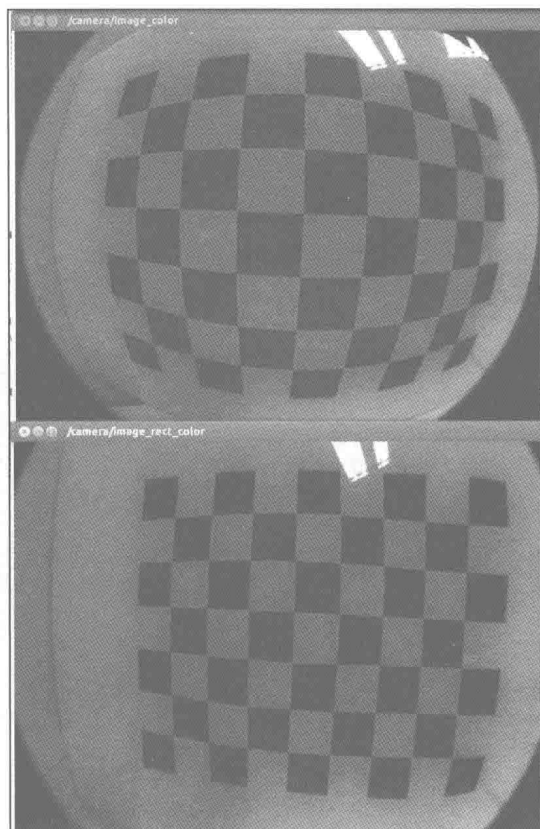
ROS 图像管道通过 `image_proc` 功能包运行。它提供了各种用于从摄像头采集的 RAW 图像中获取单色和彩色图像的转换功能。在使用 FireWire 摄像头的情况下，摄像头（在其图像传感器中）很可能使用了拜耳（Bayer）模式进行图像编码。需要进行去拜耳化来获得彩色图像。一旦你标定完摄像头，图像管道就会提取 `CameraInfo` 消息（其中包含了去拜耳化模式信息）并修正你的图像。这里，修正意味着修复图像，这样就能用失真模型的参数来修正径向和切向的失真。

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 RAW、单色图像和彩色图像：





在主题 `image_rect` 和 `image_rect_color` 中，分别提供修正后的单色和彩色图像。在下图中，我们比较未标定过的、扭曲的源图像和修正后的图像。你能分辨出哪个是修正的是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 `rqt_graph` 命令查看所有活动的主题，这也包括 `image_transport` 主题。你能直接查看单目摄像头的 `image_raw` 主题，使用以下命令：

我们有一套调整好并适用于本书编写时环境的参数，保存在参数文件 `config/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。你应该调整它来适合自己使用的环境。这是很费时间的，有时候也很棘手。但是你可以参考以下的指导进行。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`，以便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` 也会影响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。你最好找一个纹理比较多的环境，因为站在一个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。对于双目视觉导航，最好有一个长基线，一般而言是 12cm 或更远。这里用到它是因为之后我们将尝试视觉里程计。然而，使用这种配置，我们只有距离摄像头 1 米的深度信息。如果选择更小的基线，那么就on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 chapter5_tutorials camera.launch view:=true check:=true
$ roslaunch chapter5_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
Linearity RMS Error: 1.542 Pixels	Reprojection RMS Error: 1.368
Linearity RMS Error: 1.437 Pixels	Reprojection RMS Error: 1.112
Linearity RMS Error: 1.455 Pixels	Reprojection RMS Error: 1.035
Linearity RMS Error: 2.210 Pixels	Reprojection RMS Error: 1.584
Linearity RMS Error: 2.604 Pixels	Reprojection RMS Error: 2.286
Linearity RMS Error: 0.611 Pixels	Reprojection RMS Error: 0.349

对于双目摄像头，我们产生极线误差（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 2 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，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，那就非常理想了。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` 节点显示和表示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做视觉里程计。

5.5 计算机视觉任务中有用的 ROS 功能包

在 ROS 中使用计算机视觉最为有利的条件是我们不需要重复开发。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，或者仅仅做仿真。这里，我们将会列举一些有趣的计算机视觉工具。它们能够应用于大量通用的视觉任务。但是我们只会在后面详细介绍其中的一个（包括安装和使用的步骤）。这就是视觉里程计（visual odometry），当然其他的功能包也非常容易安装和使用。只需要跟着超链接中的教程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 是一种完全跨平台的库，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。当然，也有一些功能包提供物体识别管道，例如 `object_recognition` 提供 `tabletop_object_detector` 识别桌子上的物体，更通用的解决方案是 Object Recognition Kitchen (ORK) 可以参考网址 http://wg-perception.github.io/object_recognition_core。然而还有一个更有价值去探索的工具就是 RoboEarth (<http://www.robearth.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供系统运行和试用（<http://www.ros.org/wiki/robearth>）。
- **Visual odometry:** 视觉里程计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。当然在单目视觉系统下还需要一些附加信息。视觉里程计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`。在下面的小节中，我们会使用自制的双目摄像头调用 `viso2` 中的 `viso2_ros` 节点实现视觉里程计。库 `libviso2` 允许我们完成单目或双目视觉里程计，但是对于单目里程计来说，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里程计，但是如果用双目摄像头总能得到更好的结果和更正确的标定，正如本章前几节所介绍的。最后，`libfovis` 库并不支持单目视觉，但是它支持 RGBD 摄像头，例如 Kinect 传感器（见第 6 章）。对于使用双目视觉来说，你最好试一下这两个库，并看哪个库在你的环境中性能更好。这里，我们为大家提供一个在 ROS 中从安装到运行 `viso2` 和 `fovis` 库，并支持 Kinect 的详细教程。

5.6 使用 viso2 实现视觉里程计

为了使用 `viso2`，你需要切换到 `catkin` 工作空间目录（`~/dev/catkin_ws`）并使用如下命令：

```
$ cd src
$ wstool init
```

```
$ wstool set viso2 --git git://github.com/srv/viso2.git
$ wstool update
```

现在通过下面命令进行编译：

```
$ cd ..
$ catkin_make
```

一旦编译完成，我们需要使用下面命令设置环境变量：

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

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（或机器人基本连接）之间的坐标变换。双目摄像头驱动已经安装好，我们会在后面的小节介绍它是如何工作的。

5.6.1 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库：https://github.com/jbohren-forks/camera_pose。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 2、3、4 甚至更多个摄像头所需的 launch 文件。在示例中，我们只使用两个摄像头（双目视觉），所以使用 camera_pose 功能包集。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 calibration_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 chapter5_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
```

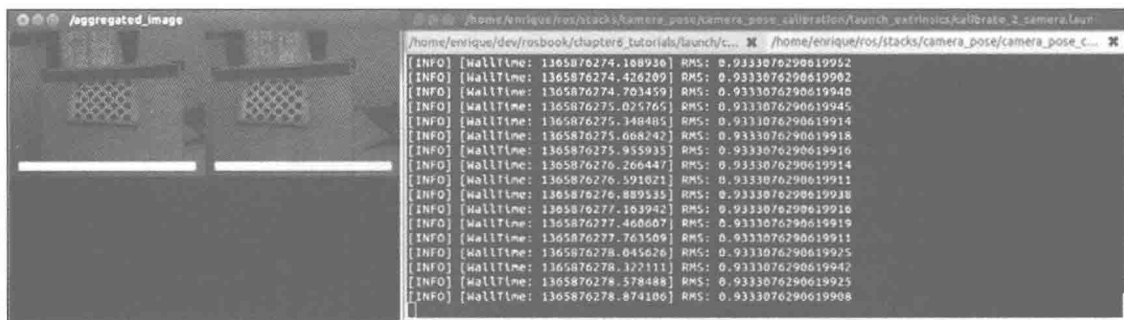
通过这个命令，我们可以使用与之前相同的标定图案。然而，它需要图像完全静态。只有当图像相对于所有摄像头都保持静止一段时间之后，一些指示条才会从左到右完全变绿，如下图所示。



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：



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。虽然这是完成摄像头位姿标定最恰当的方法，但我们还是会使用一个更简单实用的方法。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，使用 static_transform_publisher 完成摄像头连接到基本连接（例如 robot base）的变换，同时由于光学镜头连接也需要一个旋转变换，因此该文件也完成了摄像头连接到光学镜头连接的变换。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、里程计坐标系）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 chapter5_tutorials camera_stereo.launch tf:=true
```

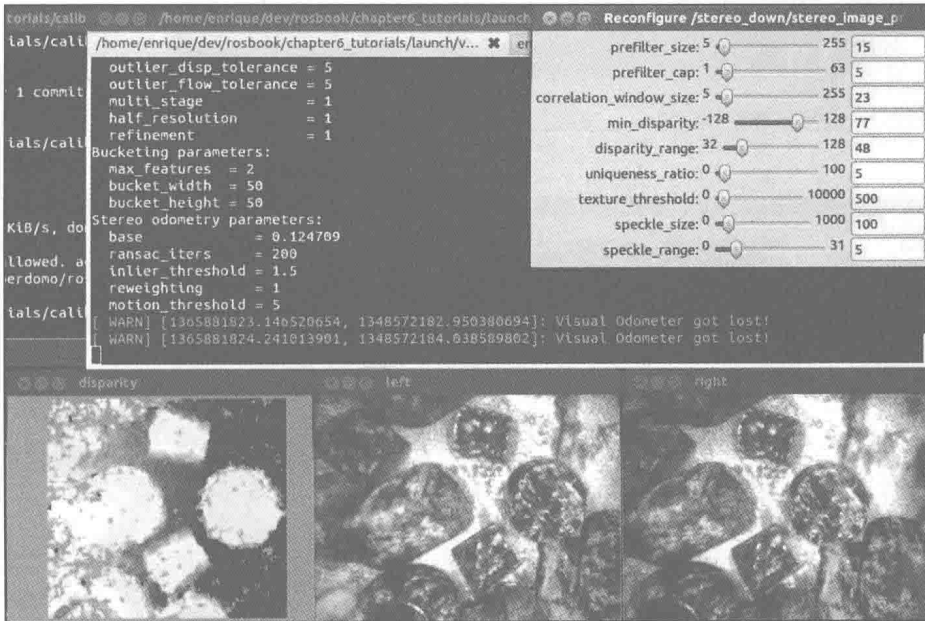
然后你就能够使用 rqt_rviz 查看 TF 数据是否发布成功。我们将会在下接下来的 viso2 在线演示中介绍。你也可以使用 rqt_tf 列出它们的树状结构（见第 3 章）

5.6.2 运行 viso2 在线演示

讲到这里，我们就要运行视觉里程计算法了。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 http://srv.uib.es/public/viso2_ros/sample_bagfiles/ 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里程计。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其内容。举例来说，要校正视差图像算法，运行以下命令：

```
$ roslaunch chapter5_tutorials viso2_demo.launch config_disparity:=true
view:=true
```

我们将看到左、右和视差图像，和一个 `rqt_reconfigure` 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记录包文件中只有 RAW 图像。我们已经在 `config/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：

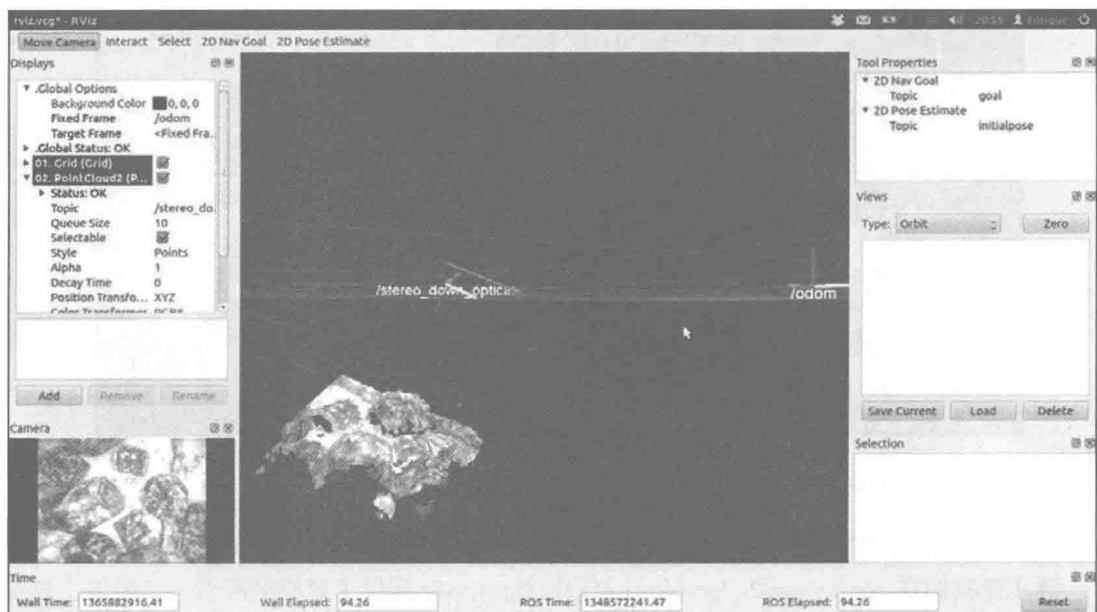


运行以下命令来启动双目里程计并在 `rqt_rviz` 中查看结果：

```
$ roslaunch chapter5_tutorials viso2_demo.launch odometry:=true rviz:=true
```

请注意我们在 `config/viso2_demo/rviz.cfg` 中为 `rqt_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

显示了在双目视觉里程计中用来进行摄像头位置估计的 `/odom` 和 `/stereo_optical` 坐标系。第三张点云图像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, 如果图像质量和里程计算法都足够好, 那么我们甚至可以在 `rqt_rviz` 中画一张地图, 但这比较困难并且需要 SLAM 算法 (请参考第 8 章)。以上所有内容如下图所示。





5.6.3 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里程计并在 rqt_rviz 中查看结果（默认不发布 tf 树）：

```
$ roslaunch chapter5_tutorials camera_stereo.launch odometry:=true  
rviz:=true
```

下图显示了低成本双目摄像头视觉里程计的结果。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，里程计信息可能会丢失，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来看是否会有更好的结果。你也可以试着调一下视差算法的参数。



5.7 使用 RGBD 深度摄像头实现视觉里程计

现在我们学习如何使用 RGBD 深度摄像头和 fovis 软件实现视觉里程计。

5.7.1 安装 fovis

由于 fovis 没有提供 Debian 功能包，需要在 catkin 工作空间输入下面命令（使用与 chapter5-tutorials 相同的工作空间）：

```
$ cd src
$ git clone https://github.com/srv/libfovis.git
$ git clone https://github.com/srv/fovis.git
$ cd ..
$ catkin_make
```

这会克隆两个源，使我们能够在 ROS 集成 fovis 软件。注意，原始代码托管在 Google 代码项目 <https://code.google.com/p/fovis/>。

一旦编译成功，在使用软件前先使用下面命令设置环境：

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

5.7.2 用 Kinect RGBD 深度摄像头运行 fovis

现在我们可以用 Kinect 深度摄像头运行 fovis。利用三维信息进行视觉里程计算，比使用 viso2 的双目和单目视觉效果更好。我们只需要启动 Kinect RGBD 摄像头驱动程序和 fovis。为了方便起见，我们提供一个启动（launch）文件同时运行这两者。

```
$ roslaunch chapter5_tutorials fovis_demo.launch
```

移动摄像头，可以得到摄像头轨迹的良好里程估计。下图所示是在 rviz 中没有移动摄像头的初始状态，你可以看到 RGBD 点云、两个箭头指示里程和摄像头当前位置：



随着摄像头移动，你可以看到箭头指示摄像头位置（如上图所示）。由于软件需要花费时间去计算里程（这取决于你所用计算机的性能），所以你需要缓慢移动摄像头：



默认情况下，fovis_demo.launch 文件使用 no_registered 深度信息，这意味着深度图像没有配准或转化为 RGB 摄像头帧。虽然最好进行配准，但这会使原来 Kinect 传感器提供的 30Hz 原始图像输出帧率显著下降到 2.5Hz 左右，依你的计算资源而定。

尽管如此，仍然可以在 RGB 摄像头上使用 throttle 来使用版本。可由所提供的 launch 文件自动运行。你可以选择以下方式：no_registered（默认），hw_registered，sw_registered。注意，原则上 Kinect 传感器不支持硬件配准模式（hw_registered），这被认为是最快的一个。因此，你可以尝试软件配准模式（sw_registered），我们调节 RGB 摄像头消息速率在 2.5Hz，你可以在 foveis_sw_registered.launch 进行修改，如下所示：

```
$ roslaunch chapter5_tutorials foveis_demo.launch mode:=sw_registered
```

5.8 计算两幅图像的单应性

单应性矩阵（homography matrix）是一个 3×3 矩阵，它提供从一个给定的图像到一个新图像的共面转换。在示例 src/homography.cpp 中，摄像头采集第一帧图片，然后通过它对比新采集的每一帧图片并计算单应性。这个示例首先需要一些平的物体，比如一本书的封面，然后运行以下命令：

```
$ roslaunch chapter5_tutorials homography.launch
```

摄像头驱动程序从摄像头启动并采集帧，检测特征（默认 SURF），提取描述符，使用

Flann-based 交叉确认过滤器 (cross-check filter) 对比第一帧图像提取匹配信息。一旦程序匹配, 计算单应性矩阵 H 。使用 H , 我们可以将新一帧的图像变形至原始帧, 如下面截图所示 (顶部是匹配点和通过 H 变形后的图像, 终端中是相应的文本显示):



5.9 本章小结

本章对 ROS 提供的计算机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计算机视觉任务, 例如视觉里程计和感知。

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 中计算机视觉相关的几个 API, 例如 `cv_bridge`、`image_transport` 和 ROS 功能包中集成的 OpenCV 库。

最后, 我们列举了 ROS 支持的一些计算机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` 和 `fovis` 库完成视觉里程计的示例, 展示了一些使用高质量摄像头记录数据的示例和使用廉价摄像头的示例。最后, 展示了如何使用特征检测、描述符提取和匹配获取两张图像之间的单应性。因此, 在学习完本章并运行示例代码之后, 你就可以在 ROS 中开始计算机视觉的使用和研究了。

在下一章, 你将学习使用 PCL 处理点云 (point cloud), 这样就可以运用 RGBD 摄像头了。

第 6 章

点 云

机器人学的工具中，点云是一种能够直观地表示和操作 3D 传感器所提供数据的方式，这类传感器包括飞行时间（Time of Flight）摄像头和激光扫描仪。该类传感器在 3D 坐标参考系下对空间进行有限点集采样构成点云。点云库（PCL）提供了大量数据类型和数据结构，不仅能够方便地表示采样空间中的点，而且可以表示采样空间的不同属性，比如颜色、法向量等。PCL 同样提供了许多最先进的算法对数据样本进行处理，比如滤波、模型估计、表面重建等。

ROS 提供了一种基于消息的接口（PCL 点云可以通过该接口进行有效的通信），还有一组将本地的 PCL 类型转换到 ROS 消息的转换函数，这和处理 OpenCV 图像一样。除了 ROS API 的标准函数之外，还有一些标准的功能包可以用来与常见的 3D 传感器进行交互，比如广泛运用的微软的 Kinect 或者 Hokuyo 的激光功能，并且可以在 rviz 的不同参考坐标系下实现数据可视化。

本章首先介绍 PCL 库的背景、相关的数据类型，以及 ROS 接口消息，然后展示一些关于如何使用 PCL 库处理数据以及如何通过 ROS 发送和接收数据的技术。

6.1 理解点云库

在研究代码之前，理解点云库和 ROS 的 PCL 接口的基本概念很重要。就像前面提到的，前者为处理 3D 数据提供了一组数据结构和算法，后者提供了一组消息以及消息与 PCL 数据结构之间的转换函数。所有这些软件功能包和库，再结合 ROS 提供的分布式通信层的能力，扩展了机器人领域的众多新应用。

概言之，PCL 包含了一个非常重要的数据结构，那就是点云。这个数据结构被设计成一个模板类，它把点的类型当作模板类的参数。因此，点云类实际上是一个点容器，这个容器里包含了所有点云需要的公共信息，而不管点是什么类型。下面是点云中最重要的公共字段：

- header：这个字段是 `pcl::PCLHeader` 类型，指定了点云的获取时间。
- points：这个字段是 `std::vector<PointT, ...>` 类型，它是储存所有点的容器。`vector` 定义中的 `PointT` 对应于类的模板参数，即点的类型。

- `width`: 这个字段指定了点云组织成一种图像时的宽度, 否则它包含的是云中点的数量。
- `height`: 这个字段指定了点云组织成一种图像时的高度, 否则它总是 1。
- `is_dense`: 这个字段指定了点云中是否有无效值 (无穷大或 NaN 值)。
- `sensor_origin_`: 这个字段是 `Eigen::Vector4f` 类型, 并且定义了传感器根据相对于原点的平移所得到的位姿。
- `sensor_orientation_`: 这个字段是 `Eigen::Quaternionf` 类型, 并且定义了传感器旋转所得到的位姿。

PCL 算法利用这些字段来处理数据, 并且用户可以利用它们来创造自己的算法。一旦明白了点云的结构, 下一步就是理解一个点云可以包含不同的点类型、PCL 如何工作以及 ROS 中的 PCL 接口。

6.1.1 不同的点云类型

正如前面描述的一样, `pcl::PointCloud` 包含了一个字段, 它作为一个容器为点提供服务。这个字段就是 `PointT` 类型, 它是 `pcl::PointCloud` 类的模板参数, 并且定义了云所要储存的点类型。PCL 定义了许多不同类型的点, 下面是一些最常用到的类型:

- `pcl::PointXYZ`: 这是最简单也可能是最常用到的点类型; 它只储存了 3D xyz 的信息。
- `pcl::PointXYZI`: 这种类型非常类似于上面的那种, 但它还包含了一个描述点亮度 (intensity) 的字段。当想要获取传感器返回的亮度高于一定级别的点时非常有用。还有与此相似的其他两种标准的点数据类型: 一是 `pcl::InterestPoint`, 它有一个字段储存强度 (strength); 二是 `pcl::PointWithRange`; 它有一个字段储存距离 (视点 to 采样点)。
- `pcl::PointXYZRGBA`: 这种点类型储存 3D 信息, 也储存颜色 (RGB=Red, Green, Blue) 和透明度 (A=Alpha)。
- `pcl::PointXYZRGB`: 这种点类型与前面的点类型相似, 但是它没有透明度字段。
- `pcl::Normal`: 这是最常用的点类型, 表示曲面上给定点处的法线以及测量的曲率。
- `pcl::PointNormal`: 这种点类型跟前一个点类型一样; 它包含了给定点所在曲面法线以及曲率信息, 但是它也包含了点的 3D xyz 坐标。这种点类型的变异类型是 `PointXYZRGBNormal` 和 `PointXYZINormal`, 正如名字所提示, 它们包含了颜色 (前者) 和亮度 (后者)。

除了这些常用的点类型外, 还有许多标准的 PCL 类型, 比如 `PointWithViewpoint`、`MomentInvariants`、`Boundary`、`PrincipalCurvatures`、`Histogram` 等。更重要的是, PCL 算法都是模板化的, 所以这样不仅可以使已经可用的类型, 理论上还可以使用用户定义的语法正确的类型。

6.1.2 PCL 中的算法

整个 PCL 函数库都使用了非常具体的设计模式, 该设计模式定义了点云处理算法。通

常来说, 这些类型算法的问题是它们高度可配置, 为了完全发挥它们的潜能, 这个库必须为用户提供一个可以指定所有要求的参数的机制以及常用的默认值。

为了解决这个问题, PCL 的开发者的决定把每个算法做成一个类, 这个类属于一个有着特定共性的继承类。这个方法允许 PCL 开发者通过获取已经存在的算法并加上新算法所需要的参数, 以重用这些算法, 并且它允许用户通过存取器轻松地给算法提供它所需要的参数值, 而其余的参数都取默认值。下面的代码展示了通常是如何使用 PCL 算法的:

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);  
pcl::PointCloud<pcl::PointXYZ>::Ptr result(new pcl::PointCloud<pcl::PointXYZ>);  
  
pcl::Algorithm<pcl::PointXYZ> algorithm;  
algorithm.setInputCloud(cloud);  
algorithm.setParameter(1.0);  
algorithm.setAnotherParameter(0.33);  
algorithm.process(*result);
```

这个方法只有在库中有要求时才遵守, 所以可能有例外, 比如 I/O 函数就不受同样要求的约束。

6.1.3 ROS 的 PCL 接口

通过 ROS 自带的基于消息的通信系统, ROS 的 PCL 接口提供了与 PCL 数据结构进行通信所需要的方法。为此, 这里定义了不同的消息类型去处理点云和其他 PCL 算法中产生的数据。结合这些消息类型, 也提供了一组将本地 PCL 数据类型转换为消息的函数。其中一些最重要的消息类型如下所示:

- `std_msgs::Header`: 这不是真的消息类型, 但它通常是每一个 ROS 消息的一部分。它包含消息发送时间、序列号和坐标系名称等信息。这个 PCL 类型等价于 `pcl::Header` type。
- `sensor_msgs::PointCloud2`: 这也许是最重要的消息类型。这个消息用来转换 `pcl::PointCloud` 类型。然而必须考虑的是, 在未来支持 `pcl::PCLPointCloud2` 的 PCL 版本中这个消息类型将会弃用。
- `pcl_msgs::PointIndices`: 这个消息类型储存了一个点云中点的索引, 等价的 PCL 类型是 `pcl::PointIndices`。
- `pcl_msgs::PolygonMesh`: 这个消息类型保存了描绘网格、顶点和多边形的信息, 等价的 PCL 类型是 `pcl::PolygonMesh`。
- `pcl_msgs::Vertices`: 这个消息类型将一组顶点的索引保存在一个数组中, 例如描述一个多边形。等价的 PCL 类型是 `pcl::Vertices`。
- `pcl_msgs::ModelCoefficients`: 这个消息类型储存了一个模型的不同系数, 例如描述一个平面需要的四个参数。等价的 PCL 类型是 `pcl::ModelCoefficients`。

通过 ROS 的 PCL 功能包提供的转换函数可以将前面的消息转换为 PCL 类型或者从 PCL 类型转换为消息。所有这些函数都有一个相似签名 (signature)，这意味着一旦我们知道如何转换一个类型，就知道如何转换所有的类型了。下面的函数是由 `pcl_conversions` 命名空间提供的：

```
void fromPCL(const <PCL Type> &, <ROS Message type> &);
void moveFromPCL(<PCL Type> &, <ROS Message type> &);
void toPCL(const <ROS Message type> &, <PCL Type> &);
void moveToPCL(<ROS Message type> &, <PCL Type> &);
```

这里，PCL Type 必须用一个预先指定的 PCL 类型替代，ROS Message type 必须用消息替代。`sensor_msgs::PointCloud2` 指定了一组函数执行这些转换：

```
void toROSMsg(const pcl::PointCloud<T> &, sensor_msgs::PointCloud2 &);
void fromROSMsg(const sensor_msgs::PointCloud2 &, pcl::PointCloud<T>
&);
void moveFromROSMsg(sensor_msgs::PointCloud2 &, pcl::PointCloud<T> &);
```

你也许会好奇每个函数和它的 `move` 版本之间的区别。答案很简单，标准版本执行对数据的深复制，而 `move` 版本执行浅复制并注销源数据容器。这称为“移动语义”(move semantic)。

6.2 我的第一个 PCL 程序

本节将学习如何集成 PCL 和 ROS。非常有必要知道并理解 ROS 软件包如何布局以及如何编译，尽管这些步骤是简单的重复。在第一个 PCL 程序中用到的示例除了能够成功编译成一个有效的 ROS 节点之外没有任何其他作用。

第一步是在你的工作空间为本章创建一个 ROS 软件包。这个软件包依赖于 `pcl_conversions`、`pcl_ros`、`pcl_msgs` 和 `sensor_msgs` 包：

```
$ catkin_create_pkg chapter6_tutorials pcl_conversions pcl_ros pcl_msgs
sensor_msgs
```

下一步是用下面的命令在软件包中创建一个源文件目录：

```
$ rospack profile
$ roscd chapter6_tutorials
$ mkdir src
```

在新的源文件目录中，应该创建一个名为 `pcl_sample.cpp` 的文件，并输入以下代码。它可以创建一个 ROS 节点并且发布一个带有 100 个元素的点云。再次说明，不用关心这些代码会做什么，因为它的目的只是实现一个能够有效使用 PCL 并且编译没有问题的节点：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_ros/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
```

```

main (int argc, char** argv)
{
    ros::init (argc, argv, "pcl_sample");
    ros::NodeHandle nh;
    ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
("pcl_output", 1);

    sensor_msgs::PointCloud2 output;
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pc
l::PointXYZ>);

    // Fill in the cloud data
    cloud->width = 100;
    cloud->height = 1;
    cloud->points.resize (cloud->width * cloud->height);

    //Convert the cloud to ROS message
    pcl::toROSMsg (*cloud, output);

    pcl_pub.publish(output);
    ros::spinOnce();

    return 0;
}

```

下一步是添加 PCL 库到 CMakeLists.txt，这样可执行的 ROS 节点就可以正确地链接到系统的 PCL 库。

```

find_package(PCL REQUIRED)

include_directories(include ${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})

```

最后，添加产生可执行程序 and 链接到合适的库的代码：

```

add_executable(pcl_sample src/pcl_sample.cpp)
target_link_libraries(pcl_sample ${catkin_LIBRARIES} ${PCL_LIBRARIES})

```

一旦到了最后一个步骤，功能包可以通过在工作空间的根目录调用 `catkin_make` 来编译。

6.2.1 创建点云

在下面第一个示例中，读者将会学习如何创建仅由随机的点组成的 PCL 点云。这个 PCL 点云最终会通过一个称为 `/pcl_output` 的主题发布出去。下面的示例展示了为将点云广播到订阅者，如何产生带有定制数据的点云以及如何将它们转换为相应的 ROS 消息类型。第一个示例的源代码可以在 `chapter6_tutorials/src` 文件夹中找到，文件名为 `pcl_create.cpp`：

```

#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>

```

```

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

    ros::NodeHandle nh;
    ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
("pcl_output", 1);
    pcl::PointCloud<pcl::PointXYZ> cloud;
    sensor_msgs::PointCloud2 output;

    // Fill in the cloud data
    cloud.width = 100;
    cloud.height = 1;
    cloud.points.resize(cloud.width * cloud.height);

    for (size_t i = 0; i < cloud.points.size (); ++i)
    {
        cloud.points[i].x = 1024 * rand () / (RAND_MAX + 1.0f);
        cloud.points[i].y = 1024 * rand () / (RAND_MAX + 1.0f);
        cloud.points[i].z = 1024 * rand () / (RAND_MAX + 1.0f);
    }

    //Convert the cloud to ROS message
    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "odom";

    ros::Rate loop_rate(1);
    while (ros::ok())
    {
        pcl_pub.publish(output);
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}

```

在这个示例的第一步，并且也是其他代码段的第一步，包含适当的头文件。在这个示例中我们会包含一些 PCL 指定的头文件，也包含一些标准的 ROS 头文件，还包含声明 PointCloud2 消息的头文件。

```

#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>

```

在节点初始化后，创建了一个 ROS 发布者 PointCloud2 并且进行了广播。这个发布者随后会用来发布 PCL 创建的点云。一旦创建了发布者，就会创建两个变量：第一个是 PointCloud2 类型，这个消息类型用来储存通过发布者发送的信息；第二个是 PointCloud<PointXYZ> 类型，这个本地 PCL 类型用来产生最初的点云。

```
ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2> ("pcl_
output", 1);
pcl::PointCloud<pcl::PointXYZ> cloud;
sensor_msgs::PointCloud2 output;
```

下一步是产生一个带有相关数据的点云。为了达到这个目的，我们需要在点云结构中分配必要的空间并且设置合适的字段。在这个示例中，创建的点云大小为 100。因为这个点云不是用来展现图像的，所以高度为 1：

```
// Fill in the cloud data
cloud.width = 100;
cloud.height = 1;
cloud.points.resize(cloud.width * cloud.height);
```

分配好空间，设置好合适的字段后，点云将填满 0 和 1024 之间的随机数：

```
for (size_t i = 0; i < cloud.points.size (); ++i)
{
    cloud.points[i].x = 1024 * rand () / (RAND_MAX + 1.0f);
    cloud.points[i].y = 1024 * rand () / (RAND_MAX + 1.0f);
    cloud.points[i].z = 1024 * rand () / (RAND_MAX + 1.0f);
}
```

这时候点云就已经创建好了，并且带有数据。由于这个节点主要为了创建一个数据源，下一步也是最后一步是将 PCL 点云类型转化为 ROS 消息类型并发布出去。为了执行转换，会用 toROSMsg 函数去执行一个将数据从 PCL 点云类型转换到 PointCloud2 消息的深复制。

最后，为了能够有一个恒定的信息源，会以 1Hz 的频率周期性发布 PointCloud2 消息，尽管消息内容是不变的：

```
//Convert the cloud to ROS message
pcl::toROSMsg(cloud, output);
output.header.frame_id = "odom";

ros::Rate loop_rate(1);

while (ros::ok())
{
    pcl_pub.publish(output);
    ros::spinOnce();
    loop_rate.sleep();
}
```

也许读者已经注意到消息头文件中的 frame_id 字段已经设置为 odom 的值。这样做的原因是为了能够在 rviz 中可视化 PointCloud2 消息。

为了运行这个示例，首先要打开一个终端并且运行 roscore 命令：

```
$ roscore
```

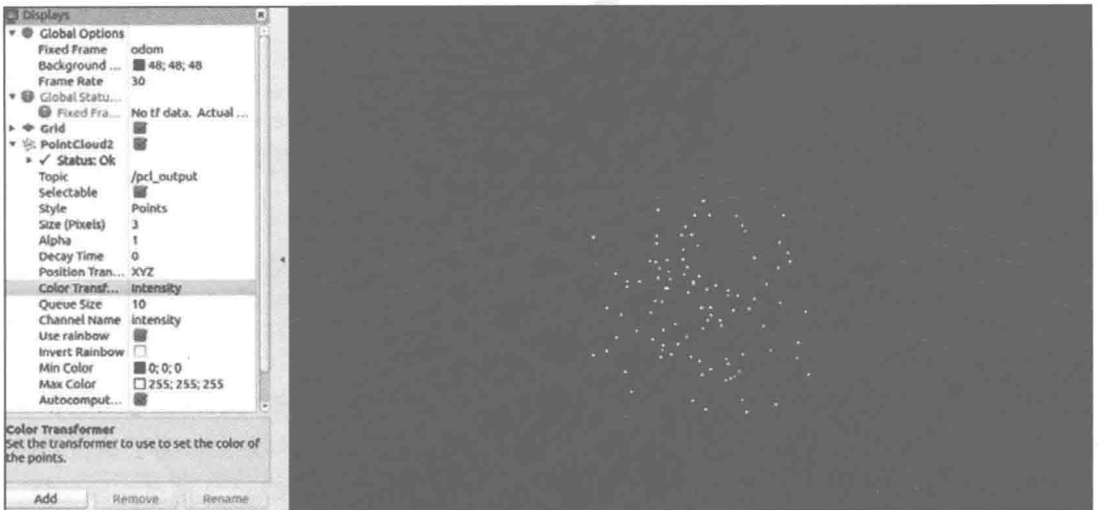
在另一个终端中，用下面的命令来运行这个示例：

```
roslaunch chapter6_tutorials pcl_create
```

为了可视化点云，必须用下面的命令来运行 rviz：

```
$ rosrun rviz rviz
```

一旦 rviz 加载好，就可以通过点击 Add 并添加 pcl_output 主题来增加一个 PointCloud2 的对象。读者必须确保在 Global Options 中将 odom 设置为固定的坐标系。如果一切工作正常，一个随机分布的点云将会在 3D 视野中呈现出来，如下面截图所示：



6.2.2 加载和保存点云到硬盘

PCL 提供了一些标准的文件格式去加载和储存点云到硬盘，研究者常用这种方法将有趣的数据集分享给其他人去试验。这种格式称为 PCD，并且经过设计它已经可以支持 PCL 指定的扩展。

这种格式非常简单：开始是一个包含关于点云中点类型和元素数目信息的数据头，然后是符合指定类型点的列表。下面是一个 PCD 文件头的示例：

```
# .PCD v.5 - Point Cloud Data file format
FIELDS x y z intensity distance sid
SIZE 4 4 4 4 4 4
TYPE F F F F F F
COUNT 1 1 1 1 1 1
WIDTH 460400
HEIGHT 1
POINTS 460400
DATA ascii
```

利用 PCL 的 API 读取 PCD 文件是非常直接的过程。下面的示例可以在 `chapter6_tutorials/src` 中找到，文件名为 `pcl_read.cpp`。这个示例展示了如何加载 PCD 文件并且将点云结果发布为 ROS 消息：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/io/pcd_io.h>

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

    ros::NodeHandle nh;
    ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
("pcl_output", 1);

    sensor_msgs::PointCloud2 output;
    pcl::PointCloud<pcl::PointXYZ> cloud;

    pcl::io::loadPCDFile ("test_pcd.pcd", cloud);

    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "odom";

    ros::Rate loop_rate(1);
    while (ros::ok())
    {
        pcl_pub.publish(output);
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

跟之前一样，第一步是包含必要的头文件。在这个特殊的示例中，唯一要添加的新头文件是 `pcl/io/pcd_io.h`，它包含了加载和储存点云到 PCD 和其他文件格式必要的定义。

跟前一个示例的主要区别在于获取点云的机制。在第一个示例中我们用随机点人工填充了点云，在这里则是从硬盘中加载它们：

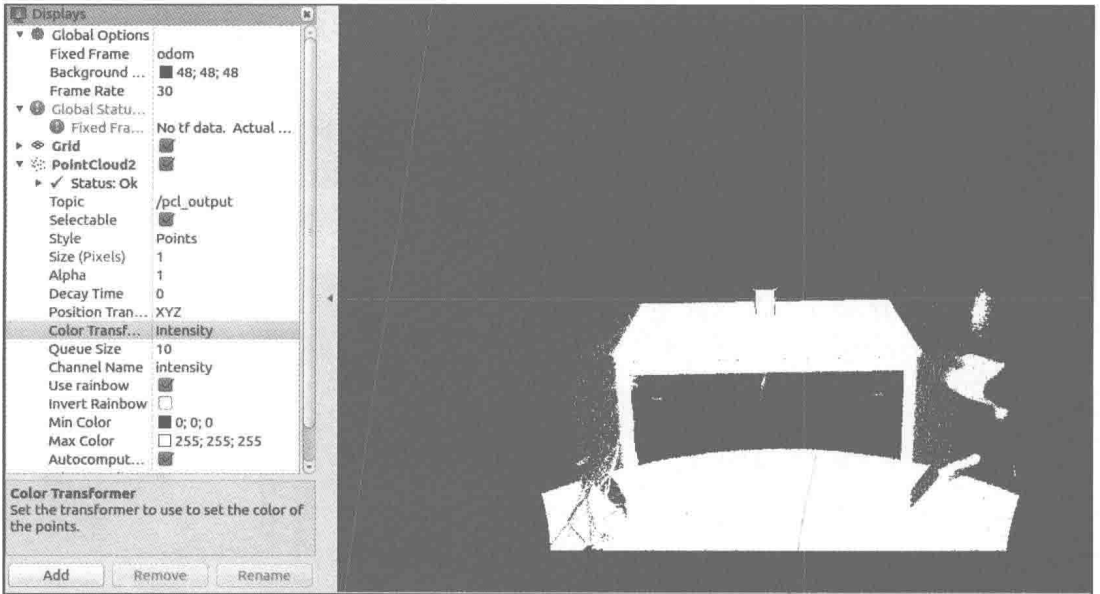
```
pcl::io::loadPCDFile ("test_pcd.pcd", cloud);
```

正如我们所看到的，加载 PCD 文件并不复杂，更高版本的 PCD 文件还可以读写点云的起始位置与方向。

为了运行前面的示例，我们要进入功能包中的数据目录，它包含了一个 PCD 文件示例，这个文件中包含的点云会在本章后面用到：

```
$ roscd chapter6_tutorials/data
$ rosrunc chapter6_tutorials pcl_read
```

如前一示例，通过 `rviz` 可以轻松实现点云可视化。



处理 PCD 文件时第二个有趣的操作是创建它们。在下面的示例中，我们的目标是订阅一个 `sensor_msgs/PointCloud2` 主题，并且将接收的点云储存到一个文件中。这个代码可以在 `chapter6_tutorials` 中找到，文件名为 `pcl_write.cpp`：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/io/pcd_io.h>

void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::fromROSMsg(input, cloud);
    pcl::io::savePCDFileASCII ("write_pcd_test.pcd", cloud);
}

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

    ros::NodeHandle nh;
    ros::Subscriber bat_sub = nh.subscribe("pcl_output", 10, cloudCB);

    ros::spin();

    return 0;
}
```


这里订阅的主题与前面的两个示例是一样的，即 `pcl_output`，所以它们可以链接在一起测试：

```
ros::Subscriber bat_sub = nh.subscribe("pcl_output", 10, cloudCB);
```

当接收一个消息后，一个回调函数被调用。在这个回调函数中的第一步是定义一个 PCL 云，并且使用 PCL-conversions 函数 `fromROSMsg` 转换接收到的 `PointCloud2`。最后，点云会以 ASCII 格式储存到硬盘，但是它也可以储存为二进制格式，二进制格式会产生更小的 PCD 文件：

```
void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::fromROSMsg(input, cloud);
    pcl::io::savePCDFileASCII ("write_pcd_test.pcd", cloud);
}
```

为了运行这个示例，必须要有一个发布者通过 `pcl_output` 主题来提供点云。在这个示例中，我们会用前面展示的 `pcl_read` 示例来满足这个要求。在三个不同的终端中，我们分别运行 `roscore`、`pcl_read` 节点和 `pcl_write` 节点：

```
$ roscore
$ roscd chapter6_tutorials/data && rosrunc chapter6_tutorials pcl_read
$ roscd chapter6_tutorials/data && rosrunc chapter6_tutorials pcl_write
```

如果一切正常，在第一个（或者第二个）消息产生后，`pcl_write` 节点应该已在 `chapter6_tutorials` 功能包的数据文件夹中创建了一个称为 `write_pcd_test.pcd` 的文件。

6.2.3 可视化点云

PCL 提供了几种方法来可视化点云。第一种也是最简单的一种是通过基本的点云查看器，它可以在 3D 查看器中展示任何类型的 PCL 点云，同时还提供一组回调函数供用户交互使用。在下面的示例中，我们会创建一个小节点用来订阅 `sensor_msgs/PointCloud2`，这个节点会使用库中的 `cloud_viewer (basic)` 来显示 `sensor_msgs/PointCloud2`。这个示例的代码可以在 `chapter6_tutorials/src` 源文件夹中找到，它的文件名为 `pcl_visualize.cpp`：

```
#include <iostream>
#include <ros/ros.h>
#include <pcl/visualization/cloud_viewer.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>

class cloudHandler
{
public:
    cloudHandler()
        : viewer("Cloud Viewer")
    {
```

```

        pcl_sub = nh.subscribe("pcl_output", 10,
&cloudHandler::cloudCB, this);
        viewer_timer = nh.createTimer(ros::Duration(0.1),
&cloudHandler::timerCB, this);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::fromROSMsg(input, cloud);

        viewer.showCloud(cloud.makeShared());
    }

    void timerCB(const ros::TimerEvent&)
    {
        if (viewer.wasStopped())
        {
            ros::shutdown();
        }
    }
}

protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    pcl::visualization::CloudViewer viewer;
    ros::Timer viewer_timer;
};

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

    cloudHandler handler;

    ros::spin();

    return 0;
}

```

这个示例代码引入了一种不同的模式。在这个示例中，不同于使用全局变量，所有的函数都封装在一个类中，这样能够提供一个使用回调函数分享变量的简洁方式。

这个构造通过默认的构造函数隐式地初始化节点句柄，它可以自动调用初始化列表中遗漏的对象。在所有的东西都正确地初始化后，用一个非常简单的与窗口名字相对应的字符串来初始化云句柄的显示。订阅者将设置接收 `pcl_output` 主题。以及设置一个定时器，它每 100ms 会触发一次回调。这个定时器用来周期性地检查窗口是否已经关闭，如果已经关闭则终止代码的执行：

```
cloudHandler()
```

```

: viewer("Cloud Viewer")
{
    pcl_sub = nh.subscribe("pcl_output", 10, &cloudHandler::cloudCB,
this);
    viewer_timer = nh.createTimer(ros::Duration(0.1),
&cloudHandler::timerCB, this);
}

```

点云回调函数和前面的示例并无太多不同，在这种特殊的情况下，PCL 点云通过 `showCloud` 函数直接传递给查看器，查看器会自动更新显示：

```

void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::fromROSMsg(input, cloud);

    viewer.showCloud(cloud.makeShared());
}

```

由于查看器窗口通常带有一个关闭按钮，也可以通过一个键盘的快捷键关闭窗口，因此将这个事件和动作考虑进去很重要，例如在终止代码时。在这种特殊的情况下，我们用回调函数来处理窗口当前状态，回调函数通过一个 ROS 定时器每隔 100ms 调用一次。如果查看器关闭了，我们的动作则是终止节点：

```

void timerCB(const ros::TimerEvent&)
{
    if (viewer.wasStopped())
    {
        ros::shutdown();
    }
}

```

为了执行这个示例，和执行其他示例一样，第一步是在一个终端运行 `roscore` 命令：

```
$ roscore
```

在第二个终端中，我们运行 `pcl_read` 示例和数据源，比如一个提示，使用下面的命令：

```
$ roscd chapter6_tutorials/data
```

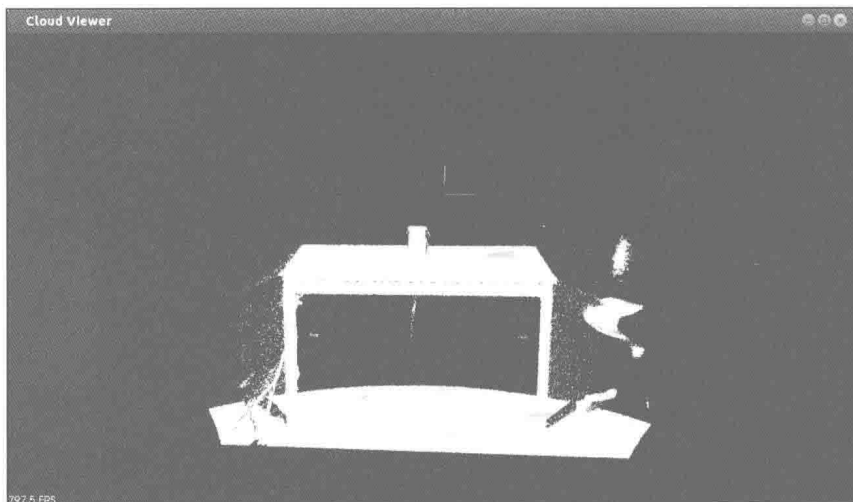
```
$ rosrund chapter6_tutorials pcl_read
```

最后，在第三个终端中，我们运行下面的命令：

```
$ rosrund chapter6_tutorials pcl_visualize
```

运行这些代码会启动一个窗口。这个窗口会显示示例提供的 PCD 测试文件中包含的点云。正如下面的屏幕截图所示。

当前的示例用到的可能是最简单的查看器，即 PCL `cloud_viewer`，这个库还提供了一个更加复杂以及完整的可视化组件，称为 `PCLVisualizer`。这个观察器能够展示点云、网格和曲面，也包括很多视角和颜色空间。在 `chapter6_tutorials` 的源代码文件夹中提供了一个称为 `pcl_visualize2.cpp` 的文件，展示了如何使用这个观察器。



通常，PCL 提供的所有观察器使用相同的底层函数并且工作的方式也大致相同。鼠标可以用来在 3D 视图中移动；组合 shift 键，可以缩放图像，组合 control 键，可以旋转图像。最后，只要按下 h 就可以在当前终端打印帮助文档，如下面的截图所示：

```
| Help:
-----
p, P : switch to a point-based representation
w, W : switch to a wireframe-based representation (where available)
s, S : switch to a surface-based representation (where available)

j, J : take a .PNG snapshot of the current window view
c, C : display current camera/window parameters
f, F : fly to point mode

e, E : exit the interactor
q, Q : stop and call VTK's TerminateApp

+/- : increment/decrement overall point size
+/- [+ ALT] : zoom in/out

g, G : display scale grid (on/off)
u, U : display lookup table (on/off)

r, R [+ ALT] : reset camera [to viewpoint = {0, 0, 0} -> center_{x, y, z}]

ALT + s, S : turn stereo mode on/off
ALT + f, F : switch between maximized window mode and original size

l, L : list all available geometric and color handlers for the current actor map
ALT + 0..9 [+ CTRL] : switch between different geometric handlers (where available)
0..9 [+ CTRL] : switch between different color handlers (where available)

SHIFT + left click : select a point

x, X : toggle rubber band selection mode for left mouse button
```

6.2.4 滤波和缩减采样

当我们尝试处理点云时，可能会遇到两个主要问题：过多的噪声和太大的密度。前者导

致算法错误地解释数据，并且导致不正确或者不准确的结果，而后者则使算法需要花费很多时间去完成运算。在本节中，我们将深入探讨如何减少点云中的噪声或者离群值（outlier），以及如何减少点的密度而不损失有价值的信息。

第一部分是创建一个节点负责过滤 `pcl_output` 主题中产生的点云离群值，并且通过 `pcl_filtered` 主题将它们发送回来。这个示例可以在 `chapter6_tutorials` 包的源文件夹中找到，文件名为 `pcl_filter.cpp`：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/filters/statistical_outlier_removal.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_output", 10,
&cloudHandler::cloudCB, this);
        pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_
filtered", 1);
    }

    void cloudCB(const sensor_msgs::PointCloud2& input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::PointCloud<pcl::PointXYZ> cloud_filtered;
        sensor_msgs::PointCloud2 output;

        pcl::fromROSMsg(input, cloud);

        pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statFilter;
        statFilter.setInputCloud(cloud.makeShared());
        statFilter.setMeanK(10);
        statFilter.setStddevMulThresh(0.2);
        statFilter.filter(cloud_filtered);

        pcl::toROSMsg(cloud_filtered, output);
        pcl_pub.publish(output);
    }

protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    ros::Publisher pcl_pub;
};

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

```
{
    ros::init(argc, argv, "pcl_filter");

    cloudHandler handler;

    ros::spin();

    return 0;
}
```

和上一个示例一样，这个例子也用到一个类，类中包含了一个发布者作为一个成员变量，这个变量用在回调函数中。这个回调函数定义了两个 PCL 点云：一个是输入消息，另一个是滤波后的点云。和往常一样，使用标准的转换函数对输入的点云进行转换：

```
pcl::PointCloud<pcl::PointXYZ> cloud;
pcl::PointCloud<pcl::PointXYZ> cloud_filtered;
sensor_msgs::PointCloud2 output;

pcl::fromROSMsg(input, cloud);
```

现在，事情开始变得有趣了。为了滤波，我们会用到 PCL 提供的统计离群值剔除算法。这个算法执行点云的分析并且能够剔除不满足指定统计特征的点。本例中的统计特征就是处于平均值附近的一个范围内，并剔除那些偏离平均值太多的点。可以通过 `setMeanK` 函数设置用来计算平均值的相邻点的数目，可以通过 `setStddevMulThresh` 设置标准偏差阈值的乘值。下面一段代码用来处理滤波以及用新的几乎无噪声的云来设置 `cloud_filtered` 点云：

```
pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statFilter;
statFilter.setInputCloud(cloud.makeShared());
statFilter.setMeanK(10);
statFilter.setStddevMulThresh(0.2);
statFilter.filter(cloud_filtered);
```

最后，跟之前一样，滤波后的点云会转换为 `PointCloud2` 并发布，这样其他的算法就可以利用这种新点云以提供更加准确的结果：

```
pcl::toROSMsg (cloud_filtered, output);
pcl_pub.publish(output);
```

在下面的截图中，可以看到前面的代码使用 PCD 测试文件所提供的点云的结果。左边是原始的点云，右边是滤波后的点云。结果虽然并不完美，但我们可以观察到已经移除了许多噪声，这意味着可以继续减少滤波后点云的密度了。



减少点云或者其他数据的密度称为缩减采样 (downsampling)。已有很多可以用来缩减点云采样的技术, 其中一些方法更加严苛或结果更好。通常来说, 缩减点云采样的目的是提高算法的执行效率。正因为如此, 需要缩减采样算法能够保持点云的基本特性和结构, 这样算法的结果不会变化太大。

在接下来的示例中, 我们将会演示如何用体素栅格滤波器 (Voxel Grid Filter) 来缩减点云的采样。在这个例子中, 输入的点云是上一个示例中滤波后的, 这样的话就可以将两个示例连接到一起, 以便在后续的计算中产生更好的结果。这个示例可以在 chapter6_tutorials 功能包的源文件夹中找到, 文件名为 pcl_downsampling.cpp:

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

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_filtered", 10,
        &cloudHandler::cloudCB, this);
        pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_
        downsamped", 1);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::PointCloud<pcl::PointXYZ> cloud_downsampled;
        sensor_msgs::PointCloud2 output;

        pcl::fromROSMsg(input, cloud);

        pcl::VoxelGrid<pcl::PointXYZ> voxelSampler;
        voxelSampler.setInputCloud(cloud.makeShared());
        voxelSampler.setLeafSize(0.01f, 0.01f, 0.01f);
        voxelSampler.filter(cloud_downsampled);

        pcl::toROSMsg(cloud_downsampled, output);
        pcl_pub.publish(output);
    }
protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    ros::Publisher pcl_pub;
};
```

```

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

    cloudHandler handler;

    ros::spin();

    return 0;
}

```

这个示例与前面的一样，不同的是主题的订阅者和发布者，在这个例子中是 `pcl_filtered` 和 `pcl_downsampled`，对点云执行的滤波算法也不相同。

正如前文所言，所用到的算法是体素栅格滤波器，这个算法将点云分解成体素（Voxel），或者是更加精确的 3D 网格，并且用子云的中心点代替每个体素中包含的所有点。每个体素的大小可以通过 `setLeafSize` 设定，并且这将确定点云密度。

```

pcl::VoxelGrid<pcl::PointXYZ> voxelSampler;
voxelSampler.setInputCloud(cloud.makeShared());
voxelSampler.setLeafSize(0.01f, 0.01f, 0.01f);
voxelSampler.filter(cloud_downsampled);

```

下面的图像显示了通过滤波和缩减采样后的图像和原始图像比较的结果。可以看到结构保留了下来，但密度减少了，并且大量噪声被完全消除了。



为了运行两个示例，像往常一样，启动 `roscore`：

```
$ roscore
```

在第二个终端，运行 `pcl_read` 示例和一个数据源：

```
$ roscd chapter6_tutorials/data
```

```
$ rosrund chapter6_tutorials pcl_read
```

在第三个终端，运行滤波的示例，它会产生 `pcl_filtered` 图像供缩减采样使用：

```
$ rosrund chapter6_tutorials pcl_filter
```

最后在第四个终端，运行缩减采样的示例：

```
$ rosrund chapter6_tutorials pcl_downsampling
```

跟之前一样，可以在 `rviz` 中看到结果，但是在这个例子中，功能包中提供的 `pcl_`

visualizer2 示例也可以使用，但是你可能需要调整订阅的主题。

6.2.5 配准与匹配

配准与匹配是一种在很多应用场景中经常使用的技术，这些应用场景包括在两个数据集中寻找共同的结构或特征，然后利用它们将数据集拼接到一起。在点云处理中，这可能和找到一个点云的结束和另一个点云的开始一样简单。当从一个高速移动的源中获取一个点云时这个技术非常有用，并且我们会得到对源的运动的一个估计。有了这个算法，我们可以将这些点云集拼接在一起并且降低估计传感器姿态时的不确定性。

PCL 提供了一个称为迭代最近点（Iterative Closest Point）算法执行配准和匹配。我们会在下面的示例中使用这个算法，这个示例可以在 chapter6_tutorials 的源文件夹中找到，它的文件名为 pcl_matching.cpp：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl/registration/icp.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_downsampled", 10,
                                &cloudHandler::cloudCB, this);
        pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_
matched", 1);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud_in;
        pcl::PointCloud<pcl::PointXYZ> cloud_out;
        pcl::PointCloud<pcl::PointXYZ> cloud_aligned;
        sensor_msgs::PointCloud2 output;

        pcl::fromROSMsg(input, cloud_in);

        cloud_out = cloud_in;

        for (size_t i = 0; i < cloud_in.points.size (); ++i)
        {
            cloud_out.points[i].x = cloud_in.points[i].x + 0.7f;
        }

        pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
        icp.setInputSource(cloud_in.makeShared());
```

```

        icp.setInputTarget(cloud_out.makeShared());

        icp.setMaxCorrespondenceDistance(5);
        icp.setMaximumIterations(100);
        icp.setTransformationEpsilon(1e-12);
        icp.setEuclideanFitnessEpsilon(0.1);

        icp.align(cloud_aligned);

        pcl::toROSMsg(cloud_aligned, output);
        pcl_pub.publish(output);
    }

protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    ros::Publisher pcl_pub;
};

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

    cloudHandler handler;

    ros::spin();

    return 0;
}

```

为了提高算法的执行效率，这个示例使用 `pcl_downsampled` 主题作为点云的输入源，最后的结果通过 `pcl_matched` 主题发布。此算法使用三个点云进行配准和匹配：第一个是要转换的点云，第二个是需要第一个点云与之对齐的固定点云，第三个是最终结果的点云。

```

pcl::PointCloud<pcl::PointXYZ> cloud_in;
pcl::PointCloud<pcl::PointXYZ> cloud_out;
pcl::PointCloud<pcl::PointXYZ> cloud_aligned;

```

由于没有连续的点云源，为了简化问题，我们将使用相同的原始点云作为固定点云，但是沿 x 轴进行了平移。算法会将两个点云对齐：

```

cloud_out = cloud_in;

for (size_t i = 0; i < cloud_in.points.size(); ++i)
{
    cloud_out.points[i].x = cloud_in.points[i].x + 0.7f;
}

```

下一步是调用迭代最近点算法进行配准和匹配。这个迭代算法使用奇异值分解（SVD），朝着输入点云到固定点云之间间隔减少的方向求解转换方程。这个算法有三个基本的停止条件：

- 上一个转换和当前转换之间的差别小于特定的阈值。这个阈值可以通过 `setTransformationEpsilon` 函数设置。
- 迭代次数已经达到用户设置的最大值。最大值可以通过 `setMaximumIterations` 函数设置。
- 最后，在循环中两次连续步骤之间的欧几里得平方误差之和低于特定的阈值。这个阈值可以通过 `setEuclideanFitnessEpsilon` 函数进行设置。

另一个用来提高结果精度的有趣参数是对应距离 (Correspondance Distance)，可以通过 `setMaxCorrespondanceDistance` 函数进行设置。这个参数定义了对准过程中两个对应点之间应具有的最小距离。

有了所有这些参数、固定的点云和输入的点云，算法就能够执行配准和匹配并返回迭代转换后的结果点云：

```
pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
icp.setInputSource(cloud_in.makeShared());
icp.setInputTarget(cloud_out.makeShared());
icp.setMaxCorrespondenceDistance(5);
icp.setMaximumIterations(100);
icp.setTransformationEpsilon(1e-12);
icp.setEuclideanFitnessEpsilon(0.1);
icp.align(cloud_aligned);
```

最后，结果点云会转换为 `PointCloud2` 并且通过相应的主题发布出去：

```
pcl::toROSMsg(cloud_aligned, output);
pcl_pub.publish(output);
```

为了运行这个示例，我们需要按照和滤波及缩减采样示例中相同的指令，从在一个终端中运行 `roscore` 开始：

```
$ roscore
```

在第二个终端中，运行 `pcl_read` 示例和数据源：

```
$ roscd chapter6_tutorials/data
```

```
$ rosrund chapter6_tutorials pcl_read
```

在第三个终端中，运行滤波示例：

```
$ rosrund chapter6_tutorials pcl_filter
```

在第四个终端中，运行缩减采样的示例：

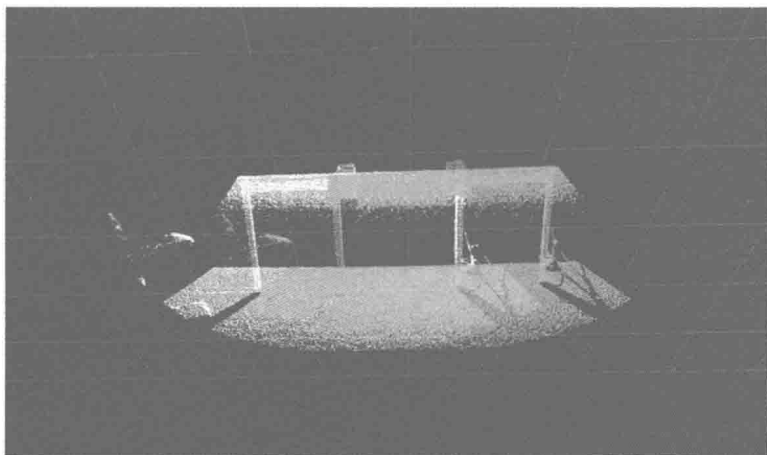
```
$ rosrund chapter6_tutorials pcl_downsampling
```

最后运行配准和匹配的节点，此节点需要订阅由前面的节点所产生的 `pcl_Downsamped` 主题：

```
$ rosrund chapter6_tutorials pcl_matching
```

可以从下面 `rviz` 中获取的图像来查看最后的结果。蓝色的点云是从 PCD 文件中获取的原始点云，而白色的点云是对应的从迭代最近点算法中获取的点云。需要说明的是原始的点

云在 x 轴上进行了平移，所以结果点云和这些点云是一致的，完全覆盖了平移后的图像，如下面的截图所示：



6.2.6 点云分区

通常，当我们处理点云数据时，也许需要访问一个局部区域的点云或者操作特定点的相邻区域。因为点云在一维数据结构中储存数据，造成这些操作固有的复杂性。为了解决这个问题，PCL 提供了两种空间数据结构，称为 kd 树 (kd-tree) 和八叉树 (octree)，它们可以作为另一种替代选择并且能够更加结构化地展示任何点云。

顾名思义，八叉树大致上说是在一个树结构中每个节点有八个孩子，并且可以用来分割 3D 空间。与之相反，kd 树是一棵二叉树，其中的节点表示 k 维的点。两种数据结构都很有趣，但在这个示例中，我们计划学习如何使用八叉树搜索和检索一个指定点周围的所有点。这个示例可以在 chapter6_tutorials 功能包的源文件夹中找到，它的文件名为 pcl_partitioning.cpp：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/octree/octree.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_downsampled", 10,
        &cloudHandler::cloudCB, this);
        pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_
        partitioned", 1);
    }
};
```

```

void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::PointCloud<pcl::PointXYZ> cloud_partitioned;
    sensor_msgs::PointCloud2 output;

    pcl::fromROSMsg(input, cloud);

    float resolution = 128.0f;
    pcl::octree::OctreePointCloudSearch<pcl::PointXYZ> octree
(resolution);

    octree.setInputCloud (cloud.makeShared());
    octree.addPointsFromInputCloud ();

    pcl::PointXYZ center_point;
    center_point.x = 0 ;
    center_point.y = 0.4;
    center_point.z = -1.4;

    float radius = 0.5;
    std::vector<int> radiusIdx;
    std::vector<float> radiusSQDist;
    if (octree.radiusSearch (center_point, radius, radiusIdx,
radiusSQDist) > 0)
    {
        for (size_t i = 0; i < radiusIdx.size (); ++i)
        {
            cloud_partitioned.points.push_back(cloud.
points[radiusIdx[i]]);
        }
    }

    pcl::toROSMsg(cloud_partitioned, output);
    output.header.frame_id = "odom";
    pcl_pub.publish(output);
}

protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    ros::Publisher pcl_pub;
};

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

    cloudHandler handler;

    ros::spin();
}

```

```
    return 0;
}
```

跟之前一样，这个示例使用 `pcl_downsampled` 主题作为点云的输入数据源并且将分割好的点云发布到 `pcl_partitioned` 主题。这个处理函数一开始将输入的点云数据转换为 PCL 点云，接下来创建一个八叉树搜索算法，它需要传递一个分辨率值，这个值决定了树最低层体素的大小，因此也决定其他属性，比如树的深度。算法也要求给定一个点云来明确地加载点：

```
float resolution = 128.0f;
pcl::octree::OctreePointCloudSearch<pcl::PointXYZ> octree
(resolution);

octree.setInputCloud (cloud.makeShared());
octree.addPointsFromInputCloud ();
```

下一步是定义分区的一个中心点。在这里，已经精心挑选出接近点云顶部的点：

```
pcl::PointXYZ center_point;
center_point.x = 0 ;
center_point.y = 0.4;
center_point.z = -1.4;
```

现在可以在一个指定点的半径范围内使用八叉树算法中的 `radiusSearch` 函数进行搜索。这个函数用来输出一些参数，它们返回落在半径内的点的索引，还有这些点到中心点距离的平方。有了这些索引，可以创建一个只包含属于这个分区点的新点云：

```
float radius = 0.5;
std::vector<int> radiusIdx;
std::vector<float> radiusSQDist;
if (octree.radiusSearch (center_point, radius, radiusIdx,
radiusSQDist) > 0)
{
    for (size_t i = 0; i < radiusIdx.size (); ++i)
    {
        cloud_partitioned.points.push_back(cloud.points[radiusIdx[i]]);
    }
}
```

最后，点云转换为 `PointCloud2` 消息类型并且发布到输出主题上：

```
pcl::toROSMsg(cloud_partitioned, output);
output.header.frame_id = "odom";
pcl_pub.publish(output);
```

为了运行这个示例，我们需要运行常用的一连串节点，从启动 `roscore` 开始：

```
$ roscore
```

在第二个终端中，运行 `pcl_read` 示例和一个数据源：

```
$ roscd chapter6_tutorials/data
$ rosrn chapter6_tutorials pcl_read
```

在第三个终端中，运行滤波示例：

```
$ rosrun chapter6_tutorials pcl_filter
```

在第四个终端中，运行缩减采样的示例：

```
$ rosrun chapter6_tutorials pcl_downsampling
```

最后，运行这个示例：

```
$ rosrun chapter6_tutorials pcl_partitioning
```

在下面的图像中，可以看到分割处理后的最后结果。因为我们精心挑选出接近于点云顶点的点，我们努力提取杯子和桌子的一部分。这个示例仅仅显示了八叉树数据结构的一小部分潜能，但是对于你加深理解是一个好的开始。



6.3 分割

分割是将数据组分割为满足特定标准的不同数据块的过程。分割可以通过许多不同的方法按照不同的标准实现。有时候，也许会涉及从一个点云中基于统计特性提取结构化信息，而在其他情况下，也可能仅仅需要提取指定颜色范围内的点。

在许多情况下，我们的数据可能满足一个特定的数学模型，例如一个平面、一条直线，或是一个球体，以及其他模型。在这种情况下，可以使用模型估计算法计算与数据相匹配的模型参数。有了这些参数，就有可能提取出属于这个模型的点并评估它们的匹配度。

在本节示例中，我们将展现如何执行一个基于模型的点云分割。我们准备将自己限定到一个平面模型中，它是最常用的数学模型，通常可以匹配一个点云，对于这个示例，我们也会执行模型估计中广泛流行的随机抽样一致（RANdom SAMple Consensus, RANSAC）算法，它是一个迭代算法，即使存在离群值也可以进行准确的估计。

这个示例可以在 chapter6_tutorials 功能包中找到，文件名为 pcl_planar_segmentation.cpp：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/ModelCoefficients.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
```

```

#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/filters/extract_indices.h>
#include <sensor_msgs/PointCloud2.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_downsampled", 10,
        &cloudHandler::cloudCB, this);
        pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_
segmented", 1);
        ind_pub = nh.advertise<pcl_msgs::PointIndices>("point_
indices", 1);
        coef_pub = nh.advertise<pcl_msgs::ModelCoefficients>("plana
r_coef", 1);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::PointCloud<pcl::PointXYZ> cloud_segmented;

        pcl::fromROSMsg(input, cloud);

        pcl::ModelCoefficients coefficients;
        pcl::PointIndices::Ptr inliers(new pcl::PointIndices());

        pcl::SACSegmentation<pcl::PointXYZ> segmentation;
        segmentation.setModelType(pcl::SACMODEL_PLANE);
        segmentation.setMethodType(pcl::SAC_RANSAC);
        segmentation.setMaxIterations(1000);
        segmentation.setDistanceThreshold(0.01);
        segmentation.setInputCloud(cloud.makeShared());
        segmentation.segment(*inliers, coefficients);

        pcl_msgs::ModelCoefficients ros_coefficients;
        pcl_conversions::fromPCL(coefficients, ros_coefficients);
        ros_coefficients.header.stamp = input.header.stamp;
        coef_pub.publish(ros_coefficients);

        pcl_msgs::PointIndices ros_inliers;
        pcl_conversions::fromPCL(*inliers, ros_inliers);
        ros_inliers.header.stamp = input.header.stamp;
        ind_pub.publish(ros_inliers);

        pcl::ExtractIndices<pcl::PointXYZ> extract;
        extract.setInputCloud(cloud.makeShared());
        extract.setIndices(inliers);
    }
};

```



```

        extract.setNegative(false);
        extract.filter(cloud_segmented);

        sensor_msgs::PointCloud2 output;
        pcl::toROSMsg(cloud_segmented, output);
        pcl_pub.publish(output);
    }

protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    ros::Publisher pcl_pub, ind_pub, coef_pub;
};

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

    cloudHandler handler;

    ros::spin();

    return 0;
}

```

读者可能注意到，这个广播的主题中用到了两种新的数据类型。正如它们的名字所提示的，ModelCoefficients 消息储存了一个数学模型的系数，PointIndices 储存了一个点云中点的索引。我们发布它们作为一个表示提取信息的替代方式，它们可以用来与原来的点云（pcl_downsampled）结合以便提取正确的点。提示一下，这可以通过将发布对象的时间戳与原始点云消息的时间戳设置成一样，并使用 ROS 消息滤波来实现：

```

pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_segmented", 1);
ind_pub = nh.advertise<pcl_msgs::PointIndices>("point_indices", 1);
coef_pub = nh.advertise<pcl_msgs::ModelCoefficients>("planar_coef",
1);

```

跟之前一样，在回调函数中，执行从 PointCloud2 消息到点云类型的转换。在这种情况下，也需要定义两个新的对象对应本地的 ModelCoefficients 和 PointIndices 类型，这些会在分割算法中用到：

```

pcl::PointCloud<pcl::PointXYZ> cloud;
pcl::PointCloud<pcl::PointXYZ> cloud_segmented;

pcl::fromROSMsg(input, cloud);

pcl::ModelCoefficients coefficients;
pcl::PointIndices::Ptr inliers(new pcl::PointIndices());

```

分割算法允许我们定义 ModelType 和 MethodType，前者是期望匹配的数学模型，后者

是要用到的算法。正如之前解释的，我们使用随机抽样一致算法是因为它们对离群点的鲁棒性好。这个算法也允许我们定义两个停止准则：迭代的最大次数（setMaxIterations）和到模型的最大距离（setDistanceThreshold）。有了这些参数设置，加上输入的点云，算法就可以工作了，返回内点（落入模型内的点）和模型的系数：

```
pcl::SACSegmentation<pcl::PointXYZ> segmentation;
segmentation.setModelType(pcl::SACMODEL_PLANE);
segmentation.setMethodType(pcl::SAC_RANSAC);
segmentation.setMaxIterations(1000);
segmentation.setDistanceThreshold(0.01);
segmentation.setInputCloud(cloud.makeShared());
segmentation.segment(*inliers, coefficients);
```

下一步是转换并发布内点和模型系数。跟之前一样，转换时通过标准函数执行，但是你也也许注意到转换函数的命名空间和签名与用来做点云转换的非常不同。为进一步加深这个示例，这些消息也包含原始点云的时间戳，以便将它们链接到一起。这也允许在其他节点上使用 ROS 消息的滤波器来创建包含链接在一起的对象回调函数：

```
pcl_msgs::ModelCoefficients ros_coefficients;
pcl_conversions::fromPCL(coefficients, ros_coefficients);
ros_coefficients.header.stamp = input.header.stamp;
coef_pub.publish(ros_coefficients);

pcl_msgs::PointIndices ros_inliers;
pcl_conversions::fromPCL(*inliers, ros_inliers);
ros_inliers.header.stamp = input.header.stamp;
ind_pub.publish(ros_inliers);
```

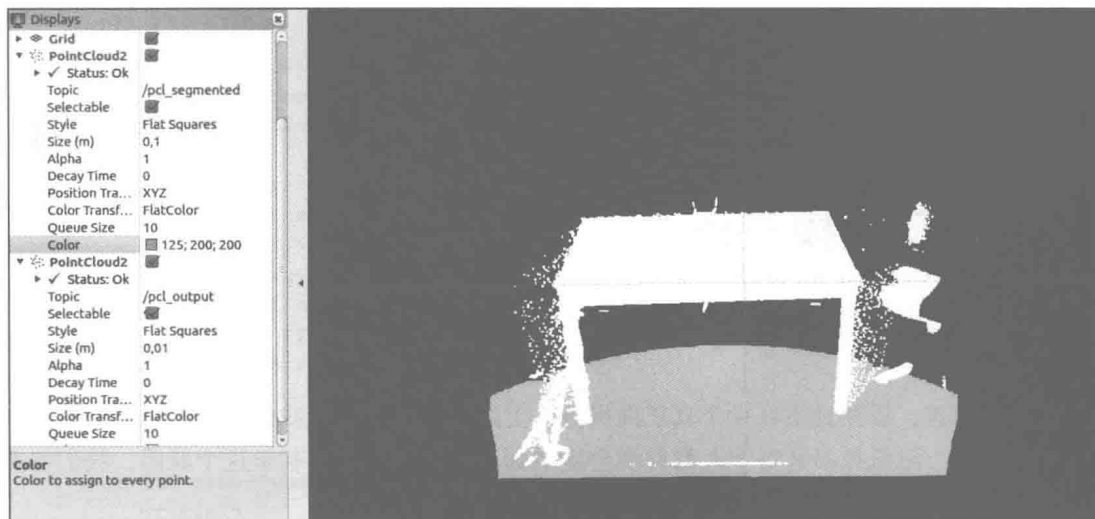
为了创建分割点云，从点云中提取出内点。最简单的方法是使用 ExtractIndices 对象，但是也可以仅仅通过在索引中进行循环，并将相应的点放到一个新的点云中轻松实现：

```
pcl::ExtractIndices<pcl::PointXYZ> extract;
extract.setInputCloud(cloud.makeShared());
extract.setIndices(inliers);
extract.setNegative(false);
extract.filter(cloud_segmented);
```

最后，将分割的点云转换为 PointCloud2 消息类型并发布它：

```
sensor_msgs::PointCloud2 output;
pcl::toROSMsg (cloud_segmented, output);
pcl_pub.publish(output)
```

转换的结果可以通过下面的图像来查看：原始的点云显示为白色，而分割的内点显示为青绿色。在这个示例中，最大的平面被提取为地板。它通常可能是我们想要从点云中提取的主要元素之一，因而这是非常方便的。



6.4 本章小结

本章我们探索了 PCL 中各种不同的工具、算法和用来与 ROS 中的点云进行交互的接口。读者可能会注意到我们努力将所有的示例链接到一起，以深入理解如何以一种可重用的方式使用这种节点。不管怎样，考虑到点云处理的计算成本，任何架构设计都不可避免地会使用系统的计算能力相关。

我们给出的示例的数据流源自于数据产生者，它们就是 `pcl_create` 和 `pcl_read`。然后进入数据滤波器 `pcl_filter` 和 `pcl_downsampling`。进行滤波后，通过 `pcl_planar_segmentation`、`pcl_partitioning` 和 `pcl_matching` 将更为复杂的信息提取出来。最后，数据通过 `pcl_write` 写入硬盘或者通过 `pcl_visualize` 可视化显示。

本章的主要目的是提供清晰且简洁的示例来展现如何将 PCL 库的基本功能与 ROS 集成，其中的一些功能可以限制为消息和转换函数。为了完成这个目标，我们解释了用于对点云进行数据处理的基础技术和通用算法，因为我们意识到这些知识越来越重要了。

第 7 章

3D 建模与仿真

相比于仿真，虽然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，我们需要对整个物理世界进行建模与仿真。

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

7.1 在 ROS 中自定义机器人的 3D 模型

机器人 3D 模型或部分结构模型主要用于仿真机器人或者为了帮助开发者简化他们的日常工作，在 ROS 中这通过 URDDF 文件实现。

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 机器人都会有 URDF 文件与之对应，例如 PR2（Willow Garage）或者 Robonaut（NASA）。在下面的小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。

7.2 创建第一个 URDF 文件

我们将在下面小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 `chapter7_tutorials/robot1_description/src/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>

```

7.2.1 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唯一：

```

<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物体，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：

```
<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（名称要求唯一）。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），以及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：

```
$ check_urdf robot1.urdf
```

此命令的输出如下所示：

```
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
```

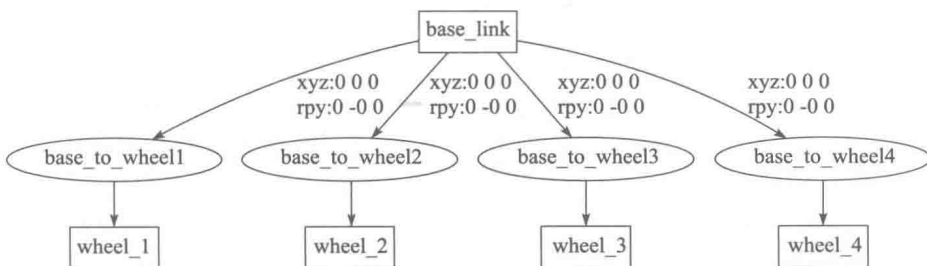
如果你希望以图形的方式来查看它，那么可以使用 urdf_to_graphviz 命令工具：

```
$ urdf_to_graphviz robot1.urdf
```

此命令将生成两个文件：origins.pdf 和 origins.gv。可以使用 evince 打开：

```
$ evince origins.pdf
```

然后你会在输出之中看到如下的图：



7.2.2 在 rviz 里查看 3D 模型

既然我们有了自己机器人的模型，就能够在 rviz 使用它并查看它的 3D 展示，以及它关

节的运动。

在 robot1_description/launch 文件夹下创建 display.launch 文件，并在文件中输入以下代码：

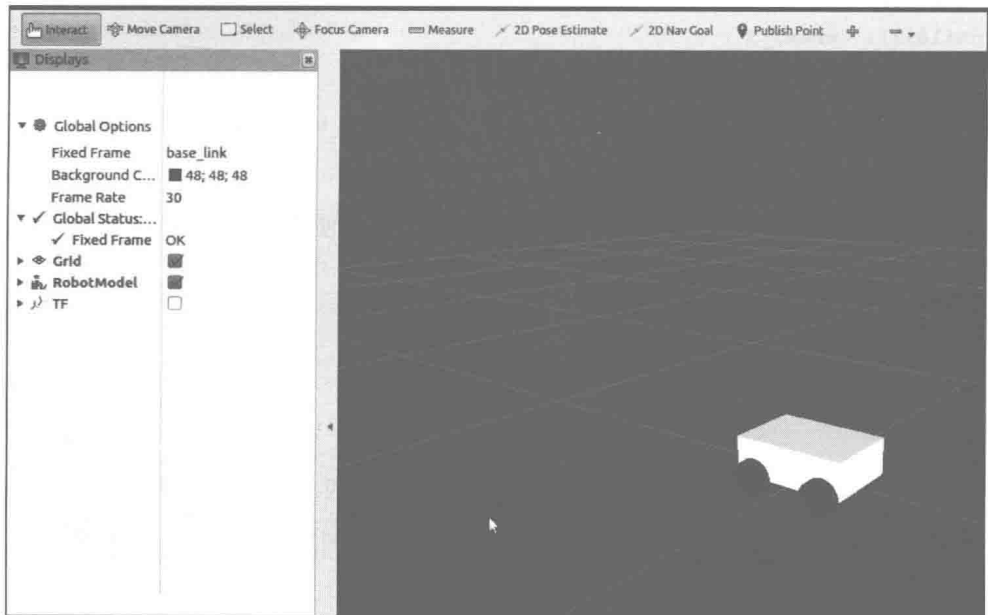
```
<?xml version="1.0"?>

<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 name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
</launch>
```

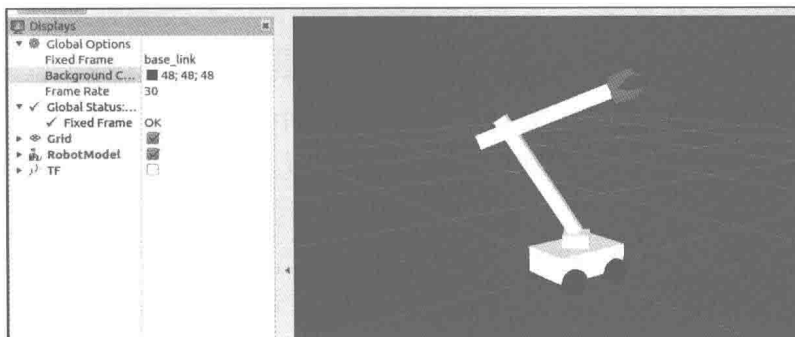
使用以下命令启动它：

```
$ roslaunch robot1_description display.launch model:="$ rosdep find
robot1_description~/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 chapter7_tutorials/robot1_description/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：

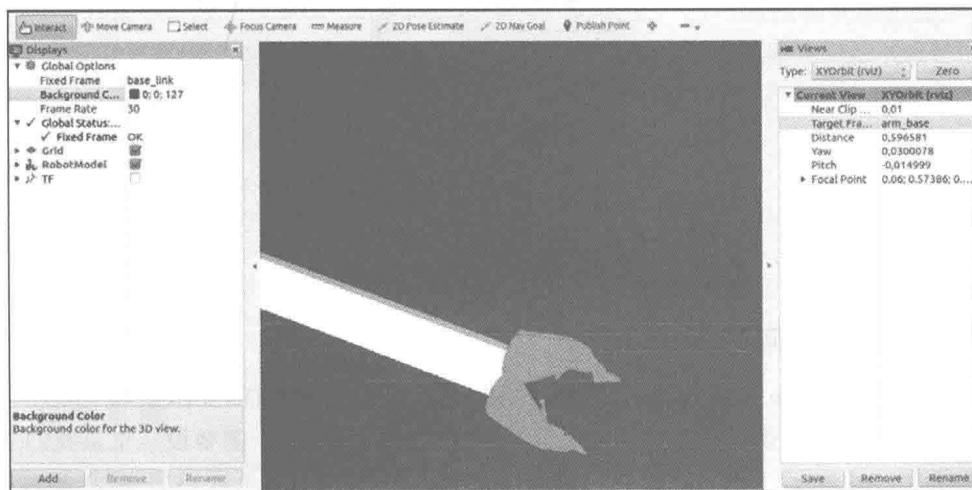


7.2.3 加载网格到机器人模型

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基本几何形状和模块，而是通过添加更多的现实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网格（mesh）^①或者使用其他机器人模型的网格。URDF 模型支持 .stl 和 .dae 格式的网格。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我们之前看到的一样，但是我们在几何部分添加了上面使用的网格。你会在下图中看到结果：



① 原文为 meshes，意为网格，此节及后续涉及多个 mesh，存疑，有待商榷。——译者注

7.2.4 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所选用的关节类型。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：

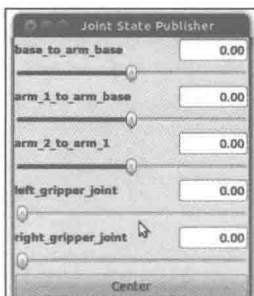
```
<joint name="arm_1_to_arm_base" type="revolute">
  <parent link="arm_base"/>
  <child link="arm_1"/>
  <axis xyz="1 0 0"/>
  <origin xyz="0 0 0.15"/>
  <limit effort="1000.0" lower="-1.0" upper="1.0" velocity="0.5"/>
</joint>
```

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制。限制通过 `<limit effort="1000.0" lower="-1.0" upper="1.0" velocity="0.5"/>` 这一行设置，可以用 `axisxyz="1 0 0"` 选择转动轴来运动。`<limit>` 标签用于选择以下属性：`effort`（关节所承受的最大力），`lower` 赋值给关节的下限（旋转接头（revolute joint）的单位是弧度，带滑梯链（prismatic joint）的单位是米），`upper` 赋值给关节的上限，`velocity` 强制关节的最大速度。

要判断关节的轴或转动限值是否合适，有一种好的办法就是使用 `Join_State_Publisher` GUI 运行 `rviz`：

```
$ roslaunch robot1_description display.launch model:="$rospack find
robot1_description"/urdf/robot1.urdf" gui:=true
```

我们会看到 `rviz` 界面，它位于另一种带有很多滑块的窗口中，其中每个滑块都能控制一个关节：



7.2.5 物理属性和碰撞属性

如果你想在 Gazebo 或者其他仿真软件中进行机器人仿真，就需要添加物理属性和碰撞属性。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。

对于网格模型文件来说, ROS 中建模的简单几何形状比引入的网格模型更加容易进行碰撞计算。相比计算简单几何形状来说, 两个网格模型之间进行碰撞计算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` 的连接中添加这些新参数:

```
<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 `collision` 和 `inertial` 元素, 因为如果你不这样做的话, Gazebo 将无法使用这些模型。

你能够在 `robot1_description/urdf/robot1_physics.urdf` 中查看带有所有参数的完整文件。

7.3 xacro——一个更好的机器人建模方法

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` 可帮助我们压缩 URDF 文件的大小, 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, 以便文件能够正确地解析。例如, 这是一个有效的 `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`。

7.3.1 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同一个数值。不使用 `xacro` 的话, 若需要改变一个值, 那就要修改若干个地方, 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：

```
${name_of_variable}:
<cylinder length="${length_wheel}" radius="${radius_wheel}"/>
```

7.3.2 使用数学方法

可以在 `${}` 结构中使用基本的四则运算（+、-、*、/）、一元负号和圆括号来构建任意复杂的表达式。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。

7.3.3 使用宏

宏是 xacro 功能包中最有用的组件。为了能够更进一步减小文件，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，还能够进一步缩小文件。

为了在 rviz 和 Gazebo 中使用 xacro 文件，需要将它转换成 .urdf 文件。可以在 robot1_description/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
```

也可以在任意地方执行以下命令，它会起到和前面命令相同的作用：

```
$ rosrun xacro xacro.py "`rospack find robot1_description`/urdf/robot1.
xacro" > "`rospack find robot1_description`/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。

7.3.4 使用代码移动机器人

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？

如果你想学习更多，我们可以创建一个简单的节点来移动机器人。ROS 提供了用于控制机器人的优秀工具，如 `ros_control` 功能包。在 `robot1_description/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。

为了移动机器人，我们必须了解 ROS 中一些通常使用的 tf 坐标系，如 map、odom 和 base_link。map tf 坐标系是世界固连坐标系，它可用于长时间的全局参考。odom 坐标系可用于精确的、短时间的局部参考。base_link 与移动机器人的底座严格相连。通常这些坐标系是相互关联的，它们之间的关系可通过图形表示为 map | odom | base_link。

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，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 是一条消息，它

保存的数据用来描述一系列转矩控制关节的状态。因为模型总共有 7 个关节，所以我们创建一条带有 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 robot1_description/launch 文件夹下以 display_xacro.launch 为名创建一个新文件（内容如下）：

```
<?xml version="1.0"?>

<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_tutorials" pkg="robot1_description"
type="state_publisher_tutorials" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find robot1_
description)/urdf.rviz" />
</launch>
```

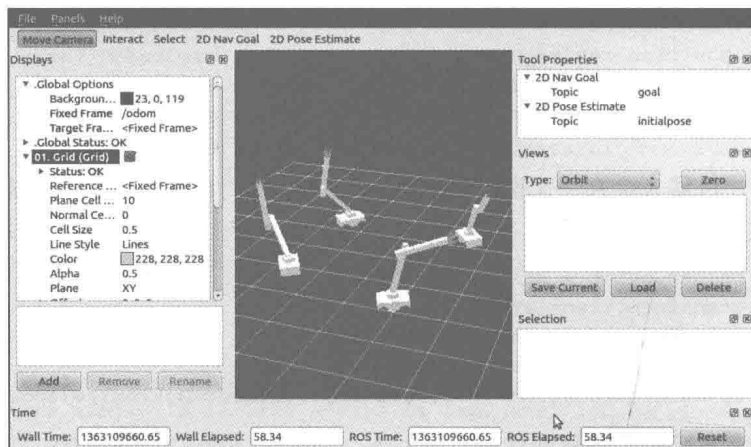
启动该节点前，必须安装以下功能包：

```
$ sudo apt-get instal ros-hydro-map-server
$ sudo apt-get install ros-hydro-fake-localization
$ cd ~/dev/catkin_ws && catkin_make
```

使用以下命令，启动带有完整模型的新节点。我们将会看到 3D 模型的每一个关节都在运动：

```
$ roslaunch robot1_description state_xacro.launch model:="$(rospack find robot1_description)/urdf/robot1.xacro"
```

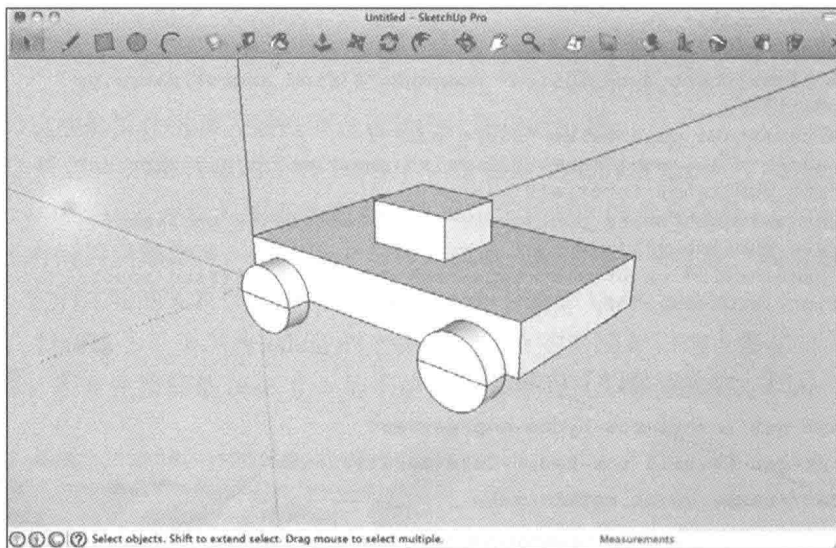
在下图中，你能看到 4 个屏幕截图的一个组合，它们分别显示了我们通过节点捕捉的机器人运动。如果你能认真观察它，就能看到手臂移动及其圆形的轨迹：



7.3.5 使用 SketchUp 进行 3D 建模

我们还可以使用诸如 SketchUp 之类的 3D 建模程序来创建模型。本节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，而本节的模型是在 Mac 中开发的，而不是在 Linux 中。

首先，你需要在电脑中安装 SketchUp。安装好之后，创建一个和下图类似的模型：



模型仅导出一个文件，因此轮子和底盘都在同一个物体中。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 robot1_description/meshes 文件夹下。

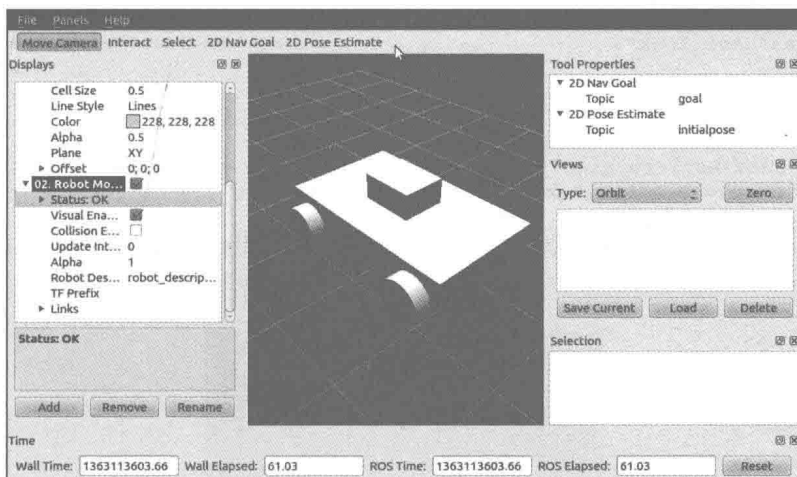
现在，为了使用 3D 模型，我们将在 robot1_description/urdf 文件夹下创建一个以 dae.urdf 为名的新文件。在文件中添加以下代码：

```
<?xml version="1.0"?>
<robot name="robot1">
  <link name="base_link">
    <visual>
      <geometry>
        <mesh scale="0.025 0.025 0.025" filename="package://robot1_
description/meshes/bot.dae"/>
      </geometry>
      <origin xyz="0 0 0.226"/>
    </visual>
  </link>
</robot>
```

你可能注意到，当加载网格时，我们可以通过命令 `<mesh scale="0.025 0.025 0.025" filename="package://robot1_description/meshes/bot.dae"/>` 选择模型的比例。使用以下命令测试模型：

```
$ roslaunch robot1_description display.launch model:="$rospack find
robot1_description"/urdf/dae.urdf"
```

你会看到以下输出：



7.4 在 ROS 中仿真

想要在 ROS 系统中对我们的机器人进行仿真，需要使用 Gazebo。

Gazebo (<http://gazebo.org/>) 是一种适用于复杂室内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。

Gazebo 现在独立于 ROS，并在 Ubuntu 中以独立功能包安装。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传感器和摄像头，并使机器人模型像真的机器人一样移动。

7.4.1 在 Gazebo 中使用 URDF 3D 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。通过以下命令确定已安装 Gazebo：

```
$ gazebo
```

在 Gazebo 工作前，我们需要安装 ROS 功能包与 Gazebo 交互：

```
$ sudo apt-get install ros-hydro-gazebo-ros-pkgs ros-hydro-gazebo-ros-control
```

接下来我们将打开 Gazebo GUI。如果一切正常的话，我们将准备好要在 Gazebo 中运行的机器人模型，可以使用以下命令测试 Gazebo 与 ROS 的集成，并检查 GUI 是否打开。

```
$ roscore & rosrunc gazebo_ros gazebo
```

为了在 Gazebo 中导入机器人模型，你需要先完成 URDF 模型。要想在 Gazebo 中使用模型，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 chapter7_tutorials/robot1_description/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 chapter7_tutorials/robot1_gazebo/launch 文件夹下，并添加以下代码：

```

<?xml version="1.0"?>

<launch>
  <!-- these are the arguments you can pass this launch file, for
example paused:=true -->
  <arg name="paused" default="true" />
  <arg name="use_sim_time" default="false" />
  <arg name="gui" default="true" />
  <arg name="headless" default="false" />
  <arg name="debug" default="true" />
  <!-- We resume the logic in empty_world.launch, changing only the
name of the world to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find robot1_gazebo)/worlds/robot.
world" />
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)" />
    <arg name="use_sim_time" value="$(arg use_sim_time)" />
    <arg name="headless" value="$(arg headless)" />
  </include>
  <!-- Load the URDF into the ROS Parameter Server -->
  <arg name="model" />
  <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />
  <!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen" args="-urdf -model robot1 -param
robot_description -z 0.05" />
</launch>

```

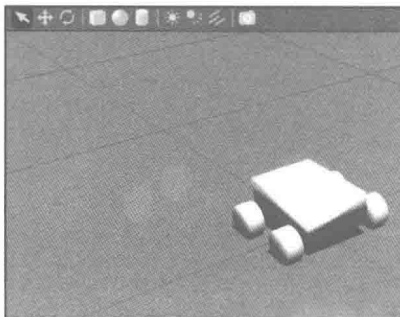
想要启动文件，需要使用以下命令：

```

$ roslaunch robot1_gazebo gazebo.launch model:="$(rospack find robot1_
description)/urdf/robot1_base_01.xacro"

```

你现在会在 Gazebo 中看到机器人。仿真初始状态是暂停的，你可点击 play 来运行它。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，你看不到它们。

为了在 Gazebo 中添加可见的纹理，需要在你的 .gazebo 模型文件中使用以下代码在 robot1_description/urdf 中创建 robot.gazebo：

```
<gazebo reference="base_link">
  <material>Gazebo/Orange</material>
</gazebo>

<gazebo reference="wheel_1">
  <material>Gazebo/Black</material>
</gazebo>

<gazebo reference="wheel_2">
  <material>Gazebo/Black</material>
</gazebo>

<gazebo reference="wheel_3">
  <material>Gazebo/Black</material>
</gazebo>

<gazebo reference="wheel_4">
  <material>Gazebo/Black</material>
</gazebo>
```

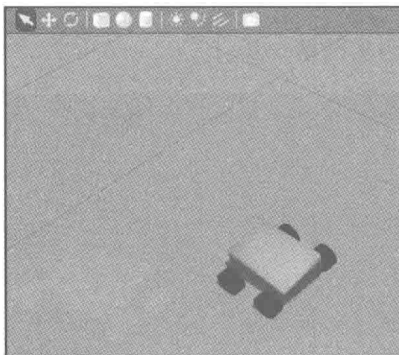
将 robot1_description/urdf/robot1_base_01.xacro 文件另存为 robot1_base_02.xacro，并添加以下代码：

```
<xacro:include filename="$(find robot1_description)/urdf/robot.gazebo" />
```

启动这个新文件。你看到的机器人虽然还是同一个，但已经带有纹理了：

```
$ roslaunch robot1_gazebo gazebo.launch model:="$(rospack find robot1_
description)/urdf/robot1_base_02.xacro"
```

你将会看到以下输出：



7.4.2 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。你同样能仿真它的传感器。

通常情况下,当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摄像头和激光雷达传感器。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,并有一个红色立方体,代表的是摄像头。我们在上一章中曾经介绍过它。

我们会从 gazebo_ros_demos 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 文件中增加这些行来为机器人添加 Hokuyo 激光雷达 3D 模型:

```
<?xml version="1.0" encoding="UTF-8"?>
<link name="hokuyo_link">
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="0.1 0.1 0.1" />
    </geometry>
  </collision>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh filename="package://robot1_description/meshes/hokuyo.dae"
    />
    </geometry>
  </visual>
  <inertial>
    <mass value="1e-5" />
    <origin xyz="0 0 0" rpy="0 0 0" />
    <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6"
  />
  </inertial>
</link>
```

在 .gazebo 文件里,我们将添加 libgazebo_ros_laser 插件,这样就可以模拟 Hokuyo 激光测距雷达的行为。

```
<gazebo reference="hokuyo_link">
  <sensor type="ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>false</visualize>
    <update_rate>40</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>720</samples>
          <resolution>1</resolution>
          <min_angle>-1.570796</min_angle>
          <max_angle>1.570796</max_angle>
        </horizontal>
```

```

</scan>
<range>
  <min>0.10</min>
  <max>30.0</max>
  <resolution>0.01</resolution>
</range>
<noise>
  <type>gaussian</type>
  <!-- Noise parameters based on published spec for Hokuyo laser
        achieving "+-30mm" accuracy at range < 10m. A mean of
0.0m and
        stddev of 0.01m will put 99.7% of samples within 0.03m
of the true
        reading. -->
  <mean>0.0</mean>
  <stddev>0.01</stddev>
</noise>
</ray>
<plugin name="gazebo_ros_head_hokuyo_controller"
filename="libgazebo_ros_laser.so">
  <topicName>/robot/laser/scan</topicName>
  <frameName>hokuyo_link</frameName>
</plugin>
</sensor>
</gazebo>

```

使用以下命令启动新的模型：

```
$ roslaunch robot1_gazebo gazebo.launch model:="$(rospack find robot1_
description)/urdf/robot1_base_03.xacro"
```

你将会看到附带激光雷达模块的机器人。

采用类似方法，我们向 robot.gazebo 和 robot1_base_03.xacro 添加几行代码以增加另一个传感器（一个摄像头）。请查阅以上文件！

在下面截图中，你可以看到机器人模型带有 Hokuyo 激光雷达和模拟摄像头的红色立方体。



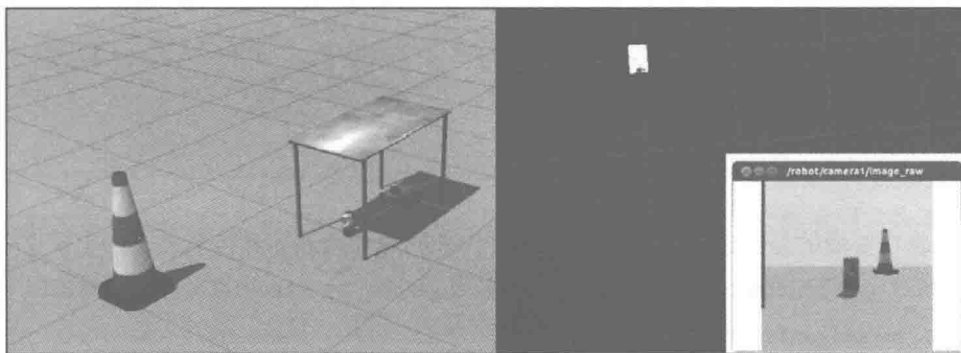
注意，这个激光雷达会像真实的雷达一样产生“真实”的传感器数据。你能够通过 `rostopic echo` 命令看到这些数据：

```
$ rostopic echo /robot/laser/scan
```

我们可以向摄像头发出相同命令，但如果你想观察摄像头看到的 gazebo 仿真图像，你可以在终端中写入以下指令：

```
$ rosrn image_view image_view image:=/robot/camera1/image_raw
```

Gazebo 允许我们使用右边的菜单添加对象，我们已经添加了一些物体，如一个交通锥标、一张桌子、一个罐子，来监测传感器对它们如何反应。你可以看到展示以上情况的三个截图：第一个截图是 Gazebo 和我们仿真的世界，第二个是从上向下视角的 rviz 及激光雷达数据，最后是摄像头图像的可视化。



7.4.3 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（Willow Garage）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。

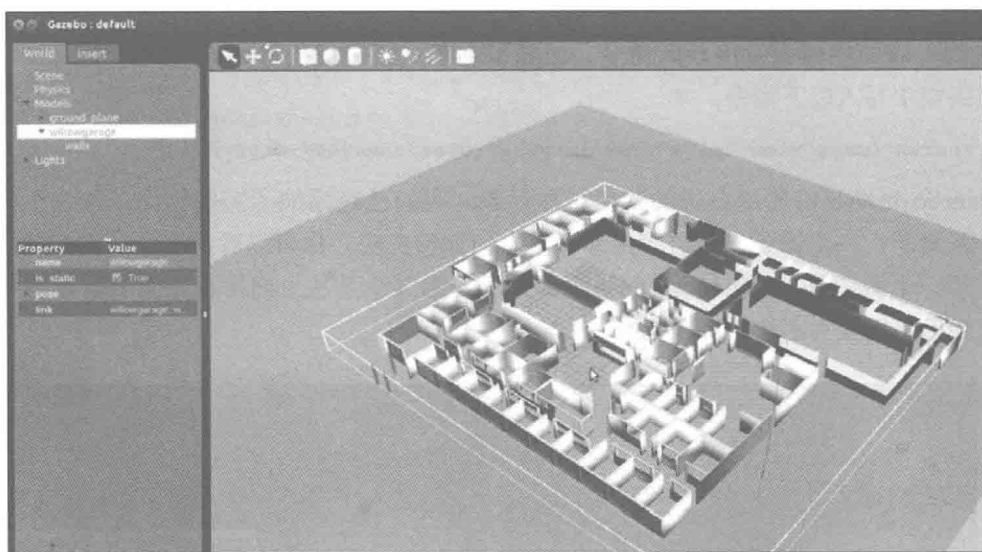
为了检查模型，你将只需要使用以下命令启动 `.launch` 文件：

```
$ roslaunch gazebo_ros willowgarage_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。通过插入和放置物体，你可以在 Gazebo 中建立自己的世界来仿真你的机器人。通过选择 `Menu | Save as` 选项来保存你的世界。

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 官方网站上查看你的显卡是否支持。同时，请注意有些时候会出现软件崩溃，当然 ROS 社区已经为软件更加稳定付出了大量的努力。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,那么我们建议升级它为最新的版本。最新版本会在较新的 ROS 发布版中默认安装。



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 robot1_gazebo/launch 文件夹下创建一个名为 gazebo_wg.launch 的文件,并添加以下代码:

```
<?xml version="1.0"?>
<launch>
  <include file="$(find gazebo_ros)/launch/willowgarage_world.launch">
  </include>

  <!-- Load the URDF into the ROS Parameter Server -->
  <param name="robot_description"
    command="$(find xacro)/xacro.py '$(find robot1_description)/
urdf/robot1_base_03.xacro'" />

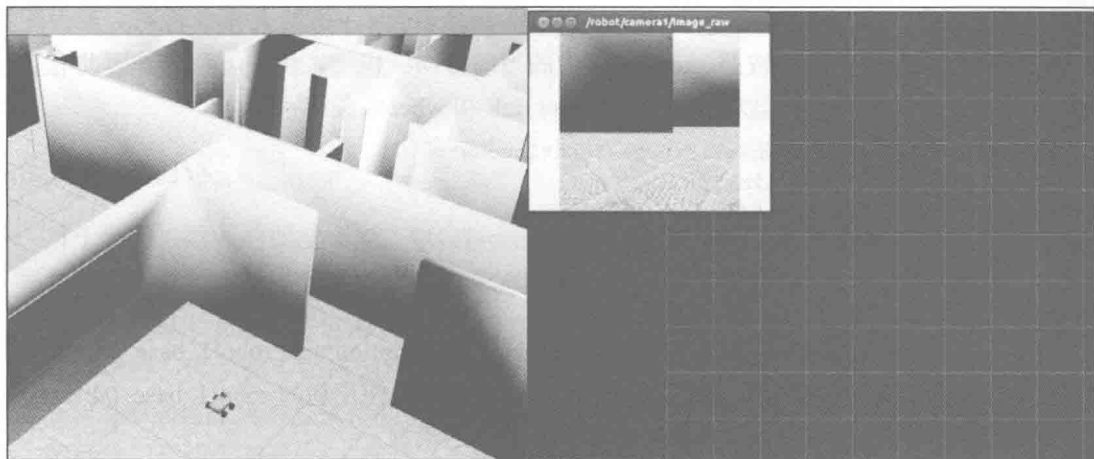
  <!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen"
    args="-urdf -model robot1 -param robot_description -z 0.05"/>

</launch>
```

现在运行带有激光雷达的模型文件:

```
$ roslaunch robot1_gazebo gazebo_wg.launch
```

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。



7.4.4 在 Gazebo 中移动机器人

滑移转向 (skid-steer) 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 skid 驱动的实现, 我们能够直接使用它移动机器人。

使用此控制器, 只需要在模型文件中增加以下代码:

```
<gazebo>
  <plugin name="skid_steel_drive_controller" filename="libgazebo_ros_
skid_steel_drive.so">
    <updateRate>100.0</updateRate>
    <robotNamespace>/</robotNamespace>
    <leftFrontJoint>base_to_wheel1</leftFrontJoint>
    <rightFrontJoint>base_to_wheel3</rightFrontJoint>
    <leftRearJoint>base_to_wheel2</leftRearJoint>
    <rightRearJoint>base_to_wheel4</rightRearJoint>
    <wheelSeparation>4</wheelSeparation>
    <wheelDiameter>0.1</wheelDiameter>
    <robotBaseFrame>base_link</robotBaseFrame>
    <torque>1</torque>
    <topicName>cmd_vel</topicName>
    <broadcastTF>0</broadcastTF>
  </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_wheel1、base_to_wheel2、base_to_wheel3 和 base_to_wheel4 关节作为机器人的驱动轮。

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。

此时, 当你发布一个 `sensor_msgs/Twist` 主题调用 `/cmd_vel` 时, 机器人将会移动。正确配置轮子关节的方向非常重要, 按照 `xacro` 文件中的当前方向, 机器人将上下移动, 因此有必要修改四个轮子的初始 `rpy`, 如以下代码中的 `base link` 和 `wheel1` 关节:

```
<joint name="base_to_wheel1" type="continuous">
  <parent link="base_link"/>
  <child link="wheel_1"/>
  <origin rpy="-1.5707 0 0" xyz="0.1 0.15 0"/>
  <axis xyz="0 0 1" />
</joint>
```

所有这些修改都保存在 `chapter7_tutorials/robot1_description/urfd/robot1_base_04.xacro` 文件中。在 `gazebo_wg.launch` 中, 我们必须更新机器人模型以使用新的 `robot1_base_04.xacro`。现在, 我们使用以下命令启动带有控制器和地图的模型:

```
$ roslaunch robot1_gazebo gazebo_wg.launch
```

你将会在 Gazebo 屏幕中看到机器人在地图上。现在我们将使用键盘来移动地图中的机器人。这个节点在 `teleop_twist_keyboard` 功能包中, 发布 `/cmd_vel` 主题。

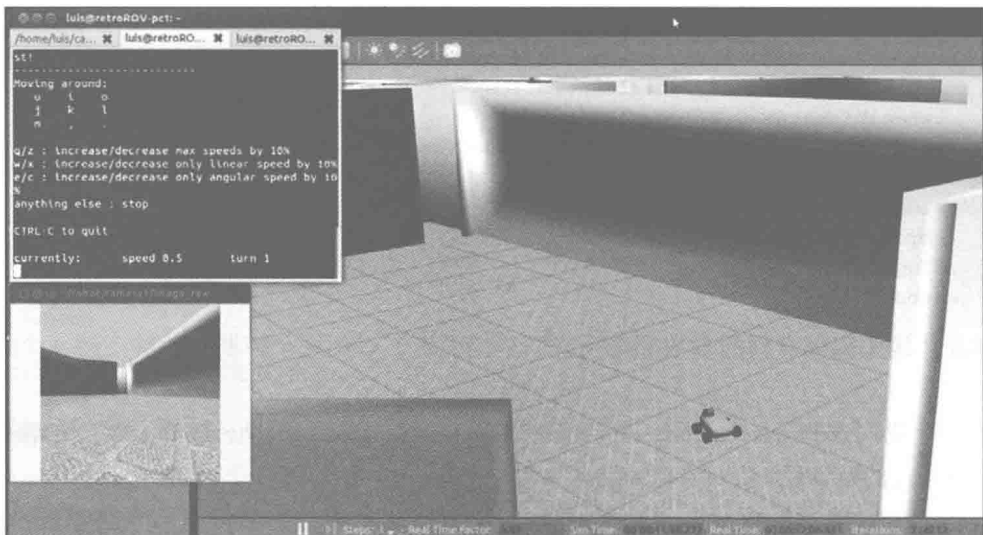
运行以下命令以安装此功能包:

```
$ sudo apt-get install ros-hydro-teleop-twist-keyboard
$ rosstack profile
$ rospack profile
```

然后, 运行节点如下:

```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

现在你会看到一个有很多说明的新命令行窗口, 你可以使用 (`u, i, o, j, k, l, m, ", ", "."`) 按键来移动机器人并设置最大速度。



如果一切顺利的话，你可以驾驶机器人穿越 Willow Garage 的办公室。你可以观察到激光雷达数据和摄像头显示的图像。

7.5 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非常有趣也非常有帮助，但并不是每个人都有机会。当你难以接触到实际的机器人时，仿真则是一个非常棒的工具。一个算法在实际的机器人上运行前也需要先测试下算法的行为，所以便有了仿真工具。

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。这包括对添加纹理、创建关节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。

概括来讲，我们已经看到了如何复用其他机器人的部件来设计我们自己的机器人，特别是我们已经包含了抓取器并添加了传感器，例如一个激光雷达和一个摄像头。

因此，在仿真中不需要完全从无到有来建造一个机器人，社区里已经开发了大量的机器人，你可以下载代码并在 ROS 和 Gazebo 中运行它们，如有必要也可以修改它们。

你可以在下面链接找到 ROS 所支持的机器人列表 <http://www.ros.org/wiki/Robots>。关于 Gazebo 的教程见下面链接 <http://gazebosim.org/tutorials>。

在下一章，我们将学习 abond 功能包，例如 SLAM 和使用激光实现导航。

第 8 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以及使用游戏杆或键盘驱动它在虚拟环境中移动。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。

得益于开源社区和共享的代码，ROS 拥有了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，会学习对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，说明一个物理量如何从一个坐标系变换到另一坐标系。例如一个传感器采集的数据或一个执行器收到期望位置的命令。tf 是一个保持坐标系跟踪的库。
- 创建一个激光雷达的驱动或对其进行仿真。
- 计算并发布里程计 (odometry) 数据，以及 Gazebo 是如何提供结果的。
- 基础控制器，包括如何为你的机器人创建它。
- 用 ROS 执行 SLAM (Simultaneous Localization And Mapping)。使用自己的机器人在环境中移动时，为此环境构建一个地图。使用导航包集里的 AMCL 算法为机器人在此地图中进行定位。AMCL 是用于 2D 机器人运动的概率定位系统。它采用了自适应蒙特卡罗定位 (adaptive Monte Carlo localization, AMCL) 方法，此方法使用了一个粒子滤波器在一个已知地图中跟踪机器人的位姿。
- 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。

8.1 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，以便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（不会产生碰撞或者卡在某个位置，或者丢失控制信号）。

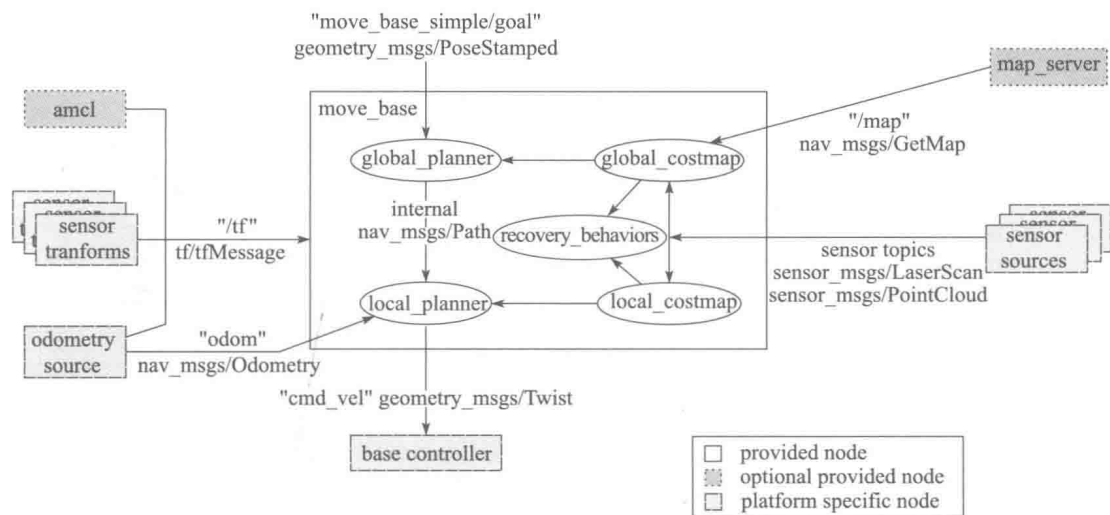
你可以认为这个功能包集能够为任何机器人所用。几乎可以说是这样的，但是有时也需

要调整一些配置文件并且调用功能包集编写一些节点。

在使用导航功能包集之前，我们的机器人还要满足一些条件：

- 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，只要你不进行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，你能生成一些相当于激光雷达或者声呐的数据，如果它们安装在机器人的其他地方，也可以将它们的值映射到地面上。

下图展示了导航功能包集的组织方式。在图中，你能够看到白色、灰色和虚线三种框。白框表示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。

8.2 创建变换

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

复杂和混乱了。

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相对于 base_link 值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。

8.2.1 创建广播机构

让我们用一些简单的代码测试一下。在 chapter8_tutorials/src 文件夹下以 tf_broadcaster.cpp 为名创建一个新的文件，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 文件中以创建新的可执行文件：

```
add_executable(tf_broadcaster src/tf_broadcaster.cpp)
target_link_libraries(tf_broadcaster ${catkin_LIBRARIES})
```

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（我们的机器人）原点的位置。

8.2.2 创建侦听器

在 chapter8_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 文件中以生成新的可执行文件。编译这个功能包，并在每个终端中分别通过以下命令运行这两个节点：

```

$ catkin_make
$ rosrn chapter8_tutorials tf_broadcaster
$ rosrn chapter8_tutorials tf_listener

```

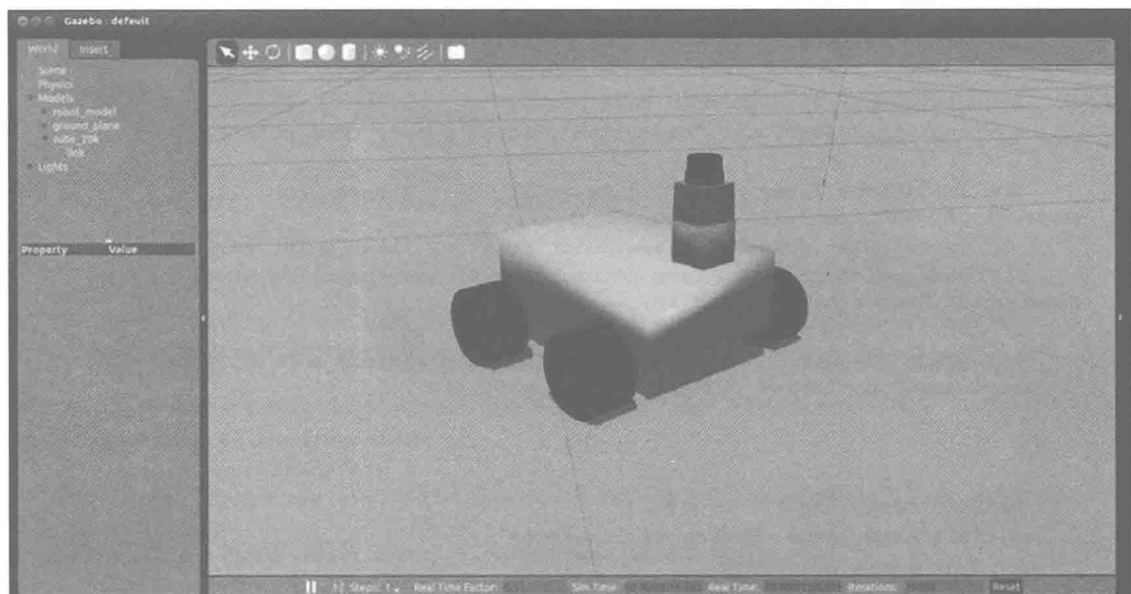
不要忘记每次要首先运行 roscore，然后你会看到如下消息：

```
[ 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7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`)。



这样，机器人中的系统必须知道这个新激光雷达的具体位置，包括和轮子、墙面之间的相对位置，只有这样才能避免碰撞。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。我们想要添加一个传感器或者机器人组件都非常方便，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。

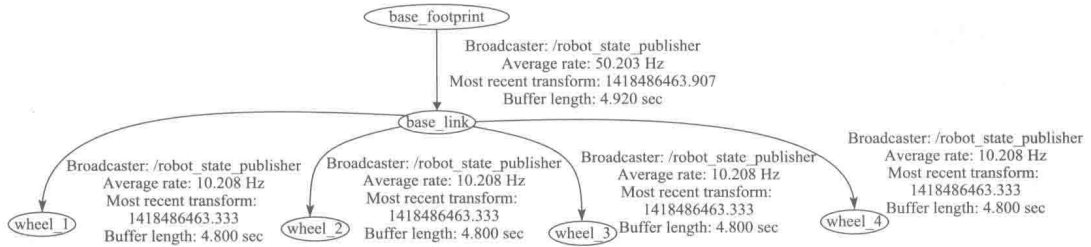
在开始编写代码进行组件配置之前，你要记住首先要有完整的机器人几何外形的定义文件（URDF 文件）。而且你可能已经忘记了，我们甚至使用 `robot_state_publisher` 功能包发布过我们机器人的坐标变换树。在第7章中我们首次使用它。这样，其实你已经有了一个能够应用导航功能包集的机器人配置。基于以上原因，我们没有必要在这里再对机器人配置进行介绍和说明了。

8.2.3 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 chapter8_tutorials gazebo_map_robot.launch model:="$ROS_PACKAGE_NAME"
find chapter8_tutorials/urdf/robot1_base_01.xacro"
$ rosrn tf view_frames
```

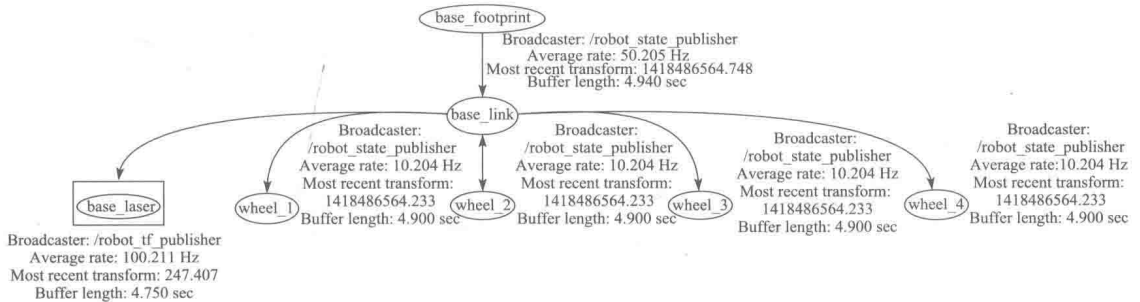
所生成的坐标系结构如下图所示：



现在，你可以再一次运行 `tf_broadcaster` 和 `roslaunch tf view_frames` 命令，你会看到你通过代码创建的坐标系：

```
$ roslaunch chapter8_tutorials tf_broadcaster
$ roslaunch tf view_frames
```

所生成的坐标系结构如下图所示：



8.3 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/PointCloud2`。

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 `hokuyo_link` 坐标系中以 `/robot/laser/scan` 为主题名发布数据。

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。但你还是有可能用到一个 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，并为你提供示例模板。

首先请你记住消息的类型是 sensor_msgs/LaserScan。使用以下命令：

```
$ rosmmsg show sensor_msgs/LaserScan
```

随后将产生以下输出：

```
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 chapter8_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`。你应该对这种消息类型比较熟悉，它在第 4 章出现过。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：

```

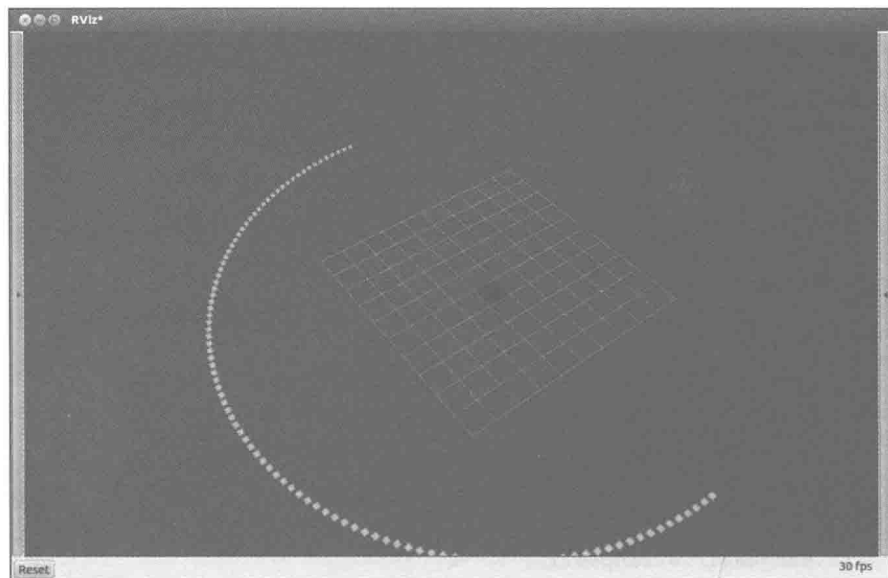
scan.header.stamp = scan_time;
scan.header.frame_id = "base_link";

```

在 `header` 中的 `frame_id` 字段也非常重要。它必须是在 `.urdf` 文件中已经创建好的坐标系之一，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 中是否有它的驱动。你只需要用你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。



8.4 发布里程数据信息

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：

```
$ rosmmsg show nav_msgs/Odometry
```

将产生以下输出：

```
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

```

正如在消息结构中所见，`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数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，另一个是角速度。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 是为了知道机器人在向左转还是向右转。

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。

8.4.1 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 7 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它。

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 Gazebo 中执行我们的示例机器人，并查看里程数据。在命令行窗口中输入以下命令：

```

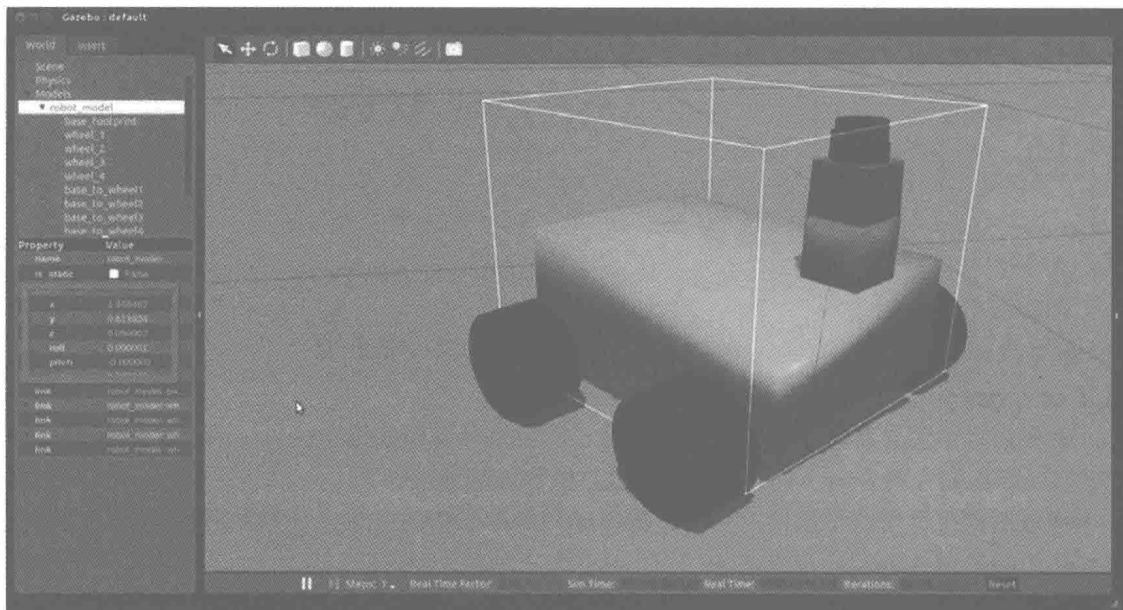
$ roslaunch chapter8_tutorials gazebo_xacro.launch model:="$(rospack find
robot1_description)/urdf/robot1_base_04.xacro"
$ rosrund teleop_twist_keyboard teleop_twist_keyboard.py

```

然后，通过 `teleop` 远程操作节点，先移动机器人一段时间，以便在 `odometry` 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数据。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，这个数据就会发生改变。



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: 1.36988769868
  y: 0.620282427846
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.28708429626
  w: 0.957905322477
---
```

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 gazebo_plugins 功能包中, 文件名为 gazebo_ros_skid_steer_drive.cpp, 下载地址为: https://github.com/ros-simulation/gazebo_ros_pkgs/blob/hydro-devel/gazebo_plugins/src/gazebo_ros_skid_steer_drive.cpp

这个文件有很多行代码, 但对于我们来说最重要的部分就是下面这个 publishOdometry() 函数:

```
void GazeboRosSkidSteerDrive::publishOdometry(double step_time)
{
    ros::Time current_time = ros::Time::now();
    std::string odom_frame =
    tf::resolve(tf_prefix_, odometry_frame_);
    std::string base_footprint_frame =
    tf::resolve(tf_prefix_, robot_base_frame_);
    // TODO create some non-perfect odometry!
    // getting data for base_footprint to odom transform
    math::Pose pose = this->parent->GetWorldPose();
    tf::Quaternion qt(pose.rot.x, pose.rot.y, pose.rot.z, pose.rot.w);
    tf::Vector3 vt(pose.pos.x, pose.pos.y, pose.pos.z);
    tf::Transform base_footprint_to_odom(qt, vt);
    if (this->broadcast_tf_)
    {
        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;
    odom_.pose.covariance[0] = 0.00001;
    odom_.pose.covariance[7] = 0.00001;
    odom_.pose.covariance[14] = 1000000000000.0;
    odom_.pose.covariance[21] = 1000000000000.0;
    odom_.pose.covariance[28] = 1000000000000.0;
    odom_.pose.covariance[35] = 0.01;
    // get velocity in /odom frame
    math::Vector3 linear;
    linear = this->parent->GetWorldLinearVel();
    odom_.twist.twist.angular.z = this->parent->GetWorldAngularVel().z;
    // convert velocity to child_frame_id (aka base_footprint)
    float yaw = pose.rot.GetYaw();
    odom_.twist.twist.linear.x = cosf(yaw) * linear.x + sinf(yaw) *
linear.y;
    odom_.twist.twist.linear.y = cosf(yaw) * linear.y - sinf(yaw) *
linear.x;
    odom_.header.stamp = current_time;
    odom_.header.frame_id = odom_frame;
```

```
odom_.child_frame_id = base_footprint_frame;
odometry_publisher_.publish(odom_);
}
```

publishOdometry() 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如何被赋值以及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 如何获取里程数据，下一步就是学习它如何发布里程数据和相对一个真实机器人的坐标变换数据。下面的代码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。

8.4.2 创建自定义里程数据

在 chapter8_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, 并分别以 `frame_id` 字段和 `child_frame_id` 字段赋值, 以便能够知道何时坐标系发生了移动。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::createQuaternionMsgFromYaw(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` 字段。

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;

```

一旦完成对所有必需字段的赋值，则发布数据：

```
// publishing the odometry and the new tf
broadcaster.sendTransform(odom_trans);
odom_pub.publish(odom);
```

请记住在编译前你需要在 CMakeLists.txt 文件中添加以下行：

```
add_executable(odometry src/odometry.cpp)
target_link_libraries(odometry ${catkin_LIBRARIES})
```

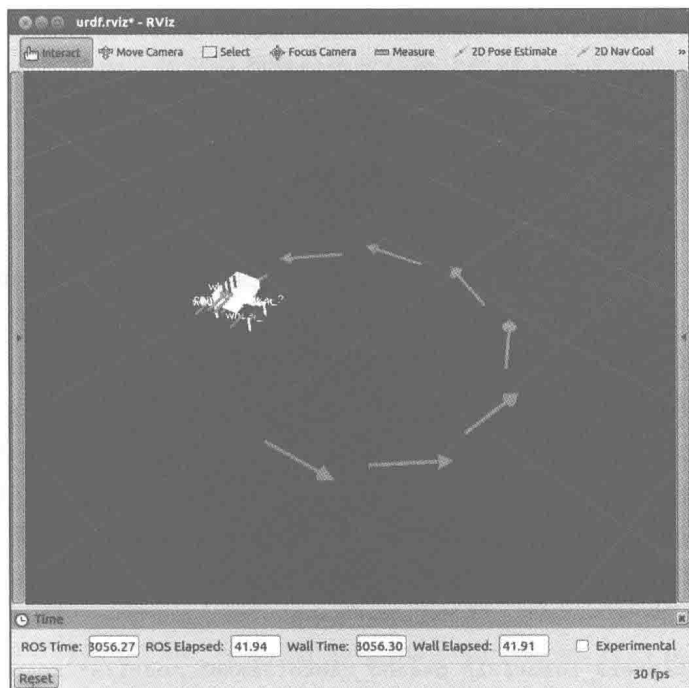
编译完功能包之后，不使用 Gazebo 而只使用 rviz 来可视化机器人模型及其运动，运行以下命令：

```
$ roslaunch chapter8_tutorials display_xacro.launch model:="$rospack find
chapter8_tutorials"/urdf/robot1_base_04.xacro"
```

使用以下命令运行里程数据处理节点：

```
$ rosrun chapter8_tutorials odometry
```

输出如下图所示：



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。红色的箭头是图形化表示的里程信息。如果一直运动下去，你会看到正如我们在代码中定义的那样，机器人围绕着一个圆形不断地运动。

8.5 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移动平台的基础控制器。

你的机器人是通过 `geometry_msgs/Twist` 类型的消息控制的。这个类型正是我们之前看到的 `Odometry` 消息所使用的。

所以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平台。

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：

```
$ rosmmsg 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基于差分轮驱动平台，驱动它的两个电动机只能够让机器人前进、后退或者转向。

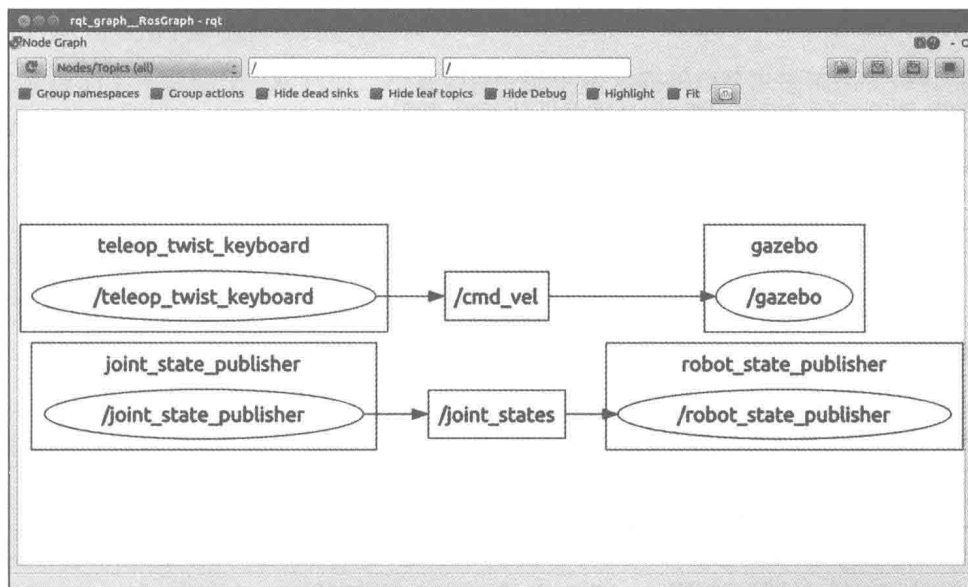
我们先在 `Gazebo` 中对这个机器人进行仿真，基础控制器在驱动中实现，用于机器人的运动和仿真。这也就是说在 `Gazebo` 中我们并不必为机器人实际创建一个基础控制器。

在本节中，你还会学到实现一个实际物理机器人的基础控制器的示例。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 chapter8_tutorials gazebo_xacro.launch model:="$(rospack find
robot1_description)/urdf/robot1_base_04.xacro"
$ rosrunc teleop_twist_keyboard teleop_twist_keyboard.py
```

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：

```
$ rqt_graph
```



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 环境中移动机器人并生成里程信息。

8.5.1 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，文件地址为：https://github.com/ros-simulation/gazebo_ros_pkgs/blob/hydro-devel/gazebo_plugins/src/gazebo_ros_skid_steer_drive.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 GazeboRosSkidSteerDrive::Load(physics::ModelPtr _parent,
sdf::ElementPtr _sdf)
{
    ...
    // ROS: Subscribe to the velocity command topic (usually "cmd_vel")
    ros::SubscribeOptions so =
    ros::SubscribeOptions::create<geometry_msgs::Twist>(command_topic_,
1,
    boost::bind(&GazeboRosSkidSteerDrive::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 GazeboRosSkidSteerDrive::cmdVelCallback(
const geometry_msgs::Twist::ConstPtr& cmd_msg)
{
    boost::mutex::scoped_lock scoped_lock(lock);
    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动机的速度进行估计，如下面函数所示：

```
void GazeboRosSkidSteerDrive::getWheelVelocities() {
    boost::mutex::scoped_lock scoped_lock(lock);
    double vr = x_;
    double va = rot_;
    wheel_speed_[RIGHT_FRONT] = vr + va * wheel_separation_ / 2.0;
    wheel_speed_[RIGHT_REAR] = vr + va * wheel_separation_ / 2.0;
    wheel_speed_[LEFT_FRONT] = vr - va * wheel_separation_ / 2.0;
    wheel_speed_[LEFT_REAR] = vr - va * wheel_separation_ / 2.0;
}
```

最后，它会调用更多机器人动力学模块中的公式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（wheel diameter）和左右轮子的间距（wheel separation）：

```
// Update the controller
void GazeboRosSkidSteerDrive::UpdateChild()
{
    common::Time current_time = this->world->GetSimTime();
    double seconds_since_last_update =
        (current_time - last_update_time_).Double();
    if (seconds_since_last_update > update_period_)
    {
        publishOdometry(seconds_since_last_update);
        // Update robot in case new velocities have been requested
        getWheelVelocities();
        joints[LEFT_FRONT]->SetVelocity(0, wheel_speed_[LEFT_FRONT] /
            wheel_diameter_);
        joints[RIGHT_FRONT]->SetVelocity(0, wheel_speed_[RIGHT_FRONT] /
            wheel_diameter_);
        joints[LEFT_REAR]->SetVelocity(0, wheel_speed_[LEFT_REAR] / wheel_
            diameter_);
        joints[RIGHT_REAR]->SetVelocity(0, wheel_speed_[RIGHT_REAR] /
            wheel_diameter_);
        last_update_time_ += common::Time(update_period_);
    }
}
```

这就是 gazebo_ros_skid_steer_drive 在 Gazebo 中控制我们仿真机器人的实现方法。

8.5.2 创建自己的基础控制器

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和编码器的实际机器人上。

在 chapter8_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.linear.x;
    double vel_th = twist.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 odometry
        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();
    }
}

```



```
}
}
```

不要忘记在 CMakeLists.txt 文件中插入以下代码来编译生成可执行文件：

```
add_executable(base_controller src/base_controller.cpp)
target_link_libraries(base_controller ${catkin_LIBRARIES})
```

这段代码只是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更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机器人运转起来。

8.6 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://wiki.ros.org/map_server)。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、保存和加载地图。

我们将会使用一个 .launch 文件来简化创建的过程。在 chapter8_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：

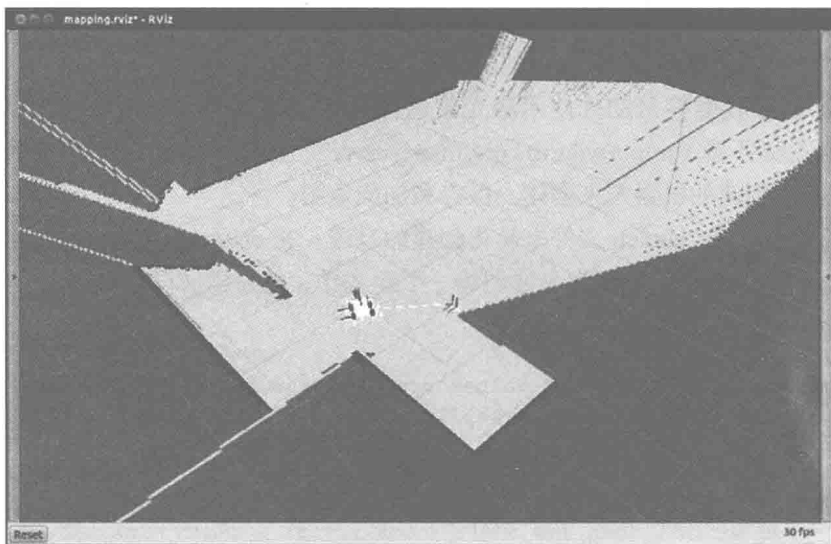
```
<?xml version="1.0"?>
<launch>
  <param name="/use_sim_time" value="true" />
  <include file="$(find gazebo_ros)/launch/willowgarage_world.
launch"/>
  <arg name="model" />
  <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
  <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen" >
    <param name="publish_frequency" type="double" value="50.0" />
  </node>
  <node name="spawn_robot" pkg="gazebo_ros" 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/mapping.rviz"/>
  <node name="slam_gmapping" pkg="gmapping" type="slam_gmapping">
    <remap from="scan" to="/robot/laser/scan"/>
    <param name="base_link" value="base_footprint"/>
  </node>
</launch>
```

有了这个 .launch 文件，你就能够以 3D 模型启动我们所需的 Gazebo 仿真环境。在这个仿真环境中，你能够通过 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，并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 chapter8 tutorials gazebo_mapping_robot.launch
model:="$rospack find robot1_description`/urdf/robot1_base_04.xacro"
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

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，在机器人移动之后，将会得到机器人位置更好的估计和地图的样子。将会使用激光扫描和里程计来构建 OGM。



8.6.1 使用 map_server 保存地图

一旦完成了整个地图或者你想要完成的部分，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：

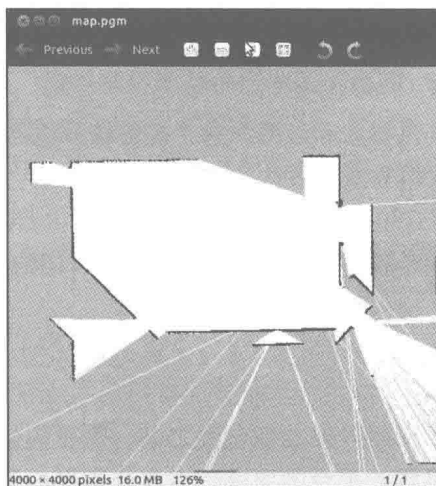
```
$ rosrun map_server map_saver -f map
```

```
[ INFO] [1418594807.613374681]: Waiting for the map
[ INFO] [1418594807.958979924, 126.530000000]: Received a 4000 X 4000 map @ 0.050 m/pix
[ INFO] [1418594807.959452501, 126.530000000]: Writing map occupancy data to map.pgm
[ INFO] [1418594808.997886519, 127.085000000]: Writing map occupancy data to map.yaml
[ INFO] [1418594808.998301431, 127.085000000]: 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，你可以使用自己喜欢的图像查看工具打开 .pgm 图像文件，你将会看到之前创建的地图：



8.6.2 使用 map_server 加载地图

当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 chapter8_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="$(find gazebo_ros)/launch/willowgarage_world.
launch"/>
  <arg name="model" />
  <param name="robot_description" command="$(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="robot_state_publisher"
```

```
name="robot_state_publisher" output="screen" >
  <param name="publish_frequency" type="double" value="50.0" />
</node>
<node name="spawn_robot" pkg="gazebo_ros" 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 chapter8_tutorials)/maps/map.yaml" />
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find chapter8_
tutorials)/launch/mapping.rviz" />
</launch>
```

现在，使用以下命令来启动文件，记住还要指定你将要使用的机器人模型：

```
$ roslaunch chapter8_tutorials gazebo_map_robot.launch model:="$rospack
find chapter8_tutorials"~/urdf/robot1_base_04.xacro"
```

然后你将会在 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。这将帮助它执行扫描匹配算法，此算法在使用 amcl（自适应蒙特卡罗定位）节点中的粒子滤波估计机器人的位置时有帮助。

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。

8.7 本章小结

在本章中，你逐步学习了使用导航功能包集之前，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、差分轮式机器人、基础控制器，而且还要满足特定的几何形状。

请记住，我们使用 Gazebo 来展示示例和解释不同配置下导航功能包集的工作原理。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，所下发的指令都可能是不一样的，其硬件执行器也可能会出现故障，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，在满足我们需要的基础之上，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 9 章

导航功能包集进阶

我们逐渐接近本书的结尾，而本章将对本书所学的所有知识进行综合运用。我们已经创建了功能包、节点、机器人的 3D 模型等。在第 8 章中，你对机器人进行了配置，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 8 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。
- 配置所有必要文件。
- 创建启动文件并开始导航。

让我们开始吧。

9.1 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：

```
$ catkin_create_pkg chapter9_tutorials roscpp tf
```

记住，在 ROS 的软件库中，你也可能找到本章所需要的所有文件。

9.2 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。

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激活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8章中,我们学习过这些内容。正如在其他小节一样,如在其他部分中,你可以使用参数来配置里程计,或复制行以添加更多的节点。

第三个需要启动的节点主要负责发布和计算机器人的几何结构,各个手臂、传感器之间的坐标变换等。

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,下面这个启动文件包括了所有我们需要的。

在 `chapter9_tutorials/launch` 文件夹下以 `chapter9_configuration_gazebo.launch` 为名创建一个新文件,并添加以下代码:

```
<?xml version="1.0"?>
<launch>
  <param name="/use_sim_time" value="true" />
  <remap from="robot/laser/scan" to="/scan" />
  <!-- start up wg world -->
  <include file="$(find gazebo_ros)/launch/willowgarage_world.
launch"/>
  <arg name="model" default="$(find robot1_description)/urdf/
```

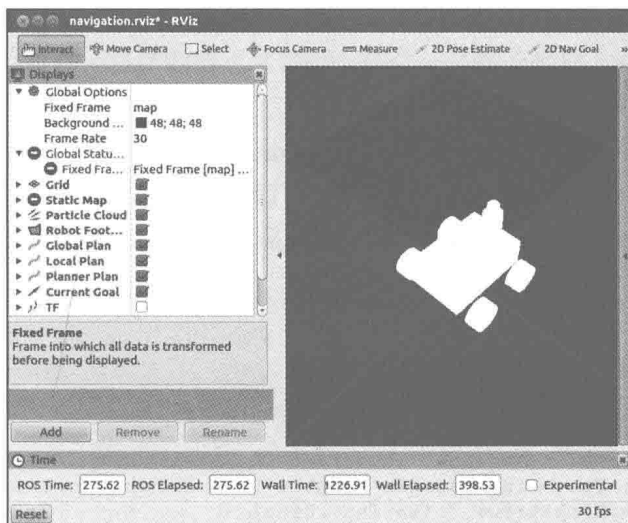
```
robot1_base_04.xacro"/>
  <param name="robot_description" command="$(find xacro)/xacro.
py $(arg model)" />

  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
  <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen" />
  <node name="spawn_robot" pkg="gazebo_ros" 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
chapter9_tutorials)/launch/navigation.rviz" />
</launch>
```

这个文件其实和我们在前面几章中所使用的文件完全相同，所以就不做过多的解释了。现在使用以下命令启动文件：

```
$ roslaunch chapter9_tutorials chapter9_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。

9.3 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和为了临时躲避障碍物的路径。例如, 机器人四周一个 $4 \times 4\text{m}$ 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。

9.3.1 基本参数的配置

让我们从最基本的参数讲起。在 `chapter9_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。

下面的脚本在 `costmap_common_params.yaml` 呈现:

```
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]]
inflation_radius: 0.5
cost_scaling_factor: 10.0
observation_sources: scan
scan: {sensor_frame: base_link, observation_persistence: 0.0, max_obstacle_height: 0.4, min_obstacle_height: 0.0, data_type: LaserScan, topic: /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, 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。

`cost_scaling_factor` 参数修改机器人绕过障碍物的行为，你可以通过修改参数设计一个激进或保守的行为。

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，以获取实际环境的数据并计算路径。

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：

```
scan: {sensor_frame: base_link, data_type: LaserScan, topic: /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寻找障碍物，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。

9.3.2 全局代价地图的配置

下一个用于配置的文件主要是全局代价地图 `global costmap`。在 `chapter9_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` 参数用于定义地图和机器人之间的坐标变换。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`。

9.3.3 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 `chapter9_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: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 5.0
  height: 5.0
  resolution: 0.02
  transform_tolerance: 0.5
  planner_frequency: 1.0
  planner_patiente: 5.0
```

其中, `global_frame`、`robot_base_frame`、`update_frequency` 和 `static_map` 参数和上一节描述的内容一致, 用来配置全局代价地图。参数 `publish_frequency` 定义了发布信息的频率。参数 `rolling_window` 用于保持在机器人运动过程中, 代价地图始终以机器人为中心。

通过 `transform_tolerance` 参数配置转换的最大延迟, 在本例中为 0.5s。通过 `planner_frequency` 参数配置规划循环的频率 (以 Hz 为单位), `planner_patiente` 参数配置在执行空间清理操作前, 规划器寻找一条有效路径的等待时间 (以 s 为单位)

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 m 为单位的。

9.3.4 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 `chapter9_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: 0.2
  min_vel_x: 0.05
  max_rotational_vel: 0.15
  min_in_place_rotational_vel: 0.01
  min_in_place_vel_theta: 0.01
  max_vel_theta: 0.15
  min_vel_theta: -0.15
  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。

当你使用的是一台完整约束的平台 (holonomic platform) 时, 那么参数 `holonomic_robot` 就应设为 `true`。在我们的示例中使用的不是完整约束的运动载体, 所以这个参数是 `false`。全向移动机器人指的是它能够以从任意位置向已配置空间移动。换句话说, 如果机器人可以到

达的位置已经以 x 、 y 轴坐标值的形式定义，那么全向移动机器人能够从任意位置出发向该位置移动。举例来说，如果机器人能够前后左右移动，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。

9.4 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 `chapter9_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 chapter9_tutorials)/maps/map.yaml" output="screen"/>

  <include file="$(find amcl)/examples/amcl_diff.launch" />

  <node pkg="move_base" type="move_base" respawn="false" name="move_
base" output="screen">

    <rosparam file="$(find chapter9_tutorials)/launch/costmap_common_
params.yaml" command="load" ns="global_costmap" />

    <rosparam file="$(find chapter9_tutorials)/launch/costmap_common_
params.yaml" command="load" ns="local_costmap" />

    <rosparam file="$(find chapter9_tutorials)/launch/local_costmap_
params.yaml" command="load" />

    <rosparam file="$(find chapter9_tutorials)/launch/global_costmap_
params.yaml" command="load" />

    <rosparam file="$(find chapter9_tutorials)/launch/base_local_
planner_params.yaml" command="load" />

  </node>
</launch>
```

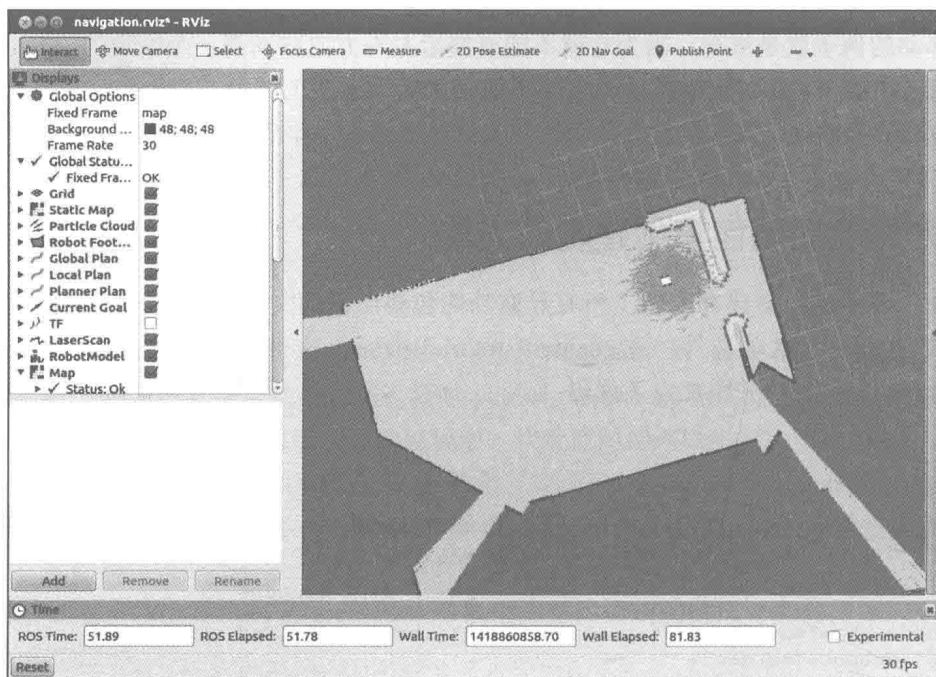
请注意，在这个文件中我们会启动之前创建的所有文件。我们还会启动一个带有第 8 章中创建地图的地图服务器和 `amcl` 节点。

我们将会使用的这个 `amcl` 节点专用于支持差分驱动机器人平台。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 8 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 `chapter9_configuration_gazebo.launch` 文件。

```
$ roslaunch chapter9_tutorials move_base.launch
```

然后你会看到以下窗口：



如果你比较这时候的界面与之前你启动 `chapter9_configuration_gazebo.launch` 文件时所看到的界面，就会发现所有的选项都变成蓝色了，这说明你的导航功能包集已经正常启动。

正如前面所讲，我们还会在后面的小节中介绍用于导航功能包集可视化所有主题必需的配置选项。

9.5 为导航功能包集设置 rviz

通过窗口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非常重要。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。

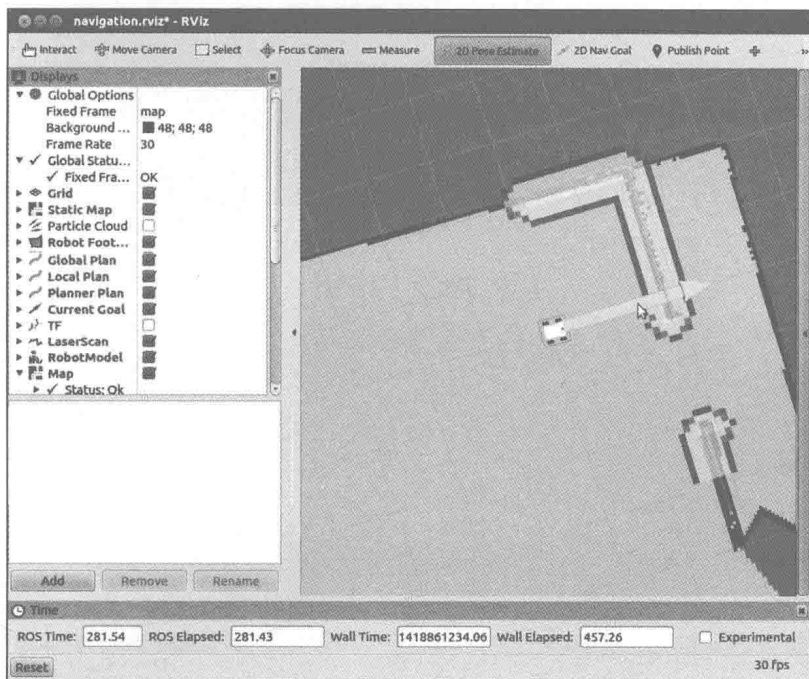
9.5.1 2D 位姿估计

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。我们之前在窗口中修改过主题的名称。

你能在下图中看到如何使用 `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。如果你一开始不进行这项工作，机器人会启动一个自动定位进程

并设定初始化位姿。

- **Topic:** initialpose
- **Type:** geometry_msgs/PoseWithCovarianceStamped



9.5.2 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 rviz 窗口 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`。在下页上图中你会看到具体配置方法。点击 2D Nav Goal 按钮并选择地图和机器人的运动目标。你能够选择 x 轴和 y 轴坐标，以及机器人最后的方向。

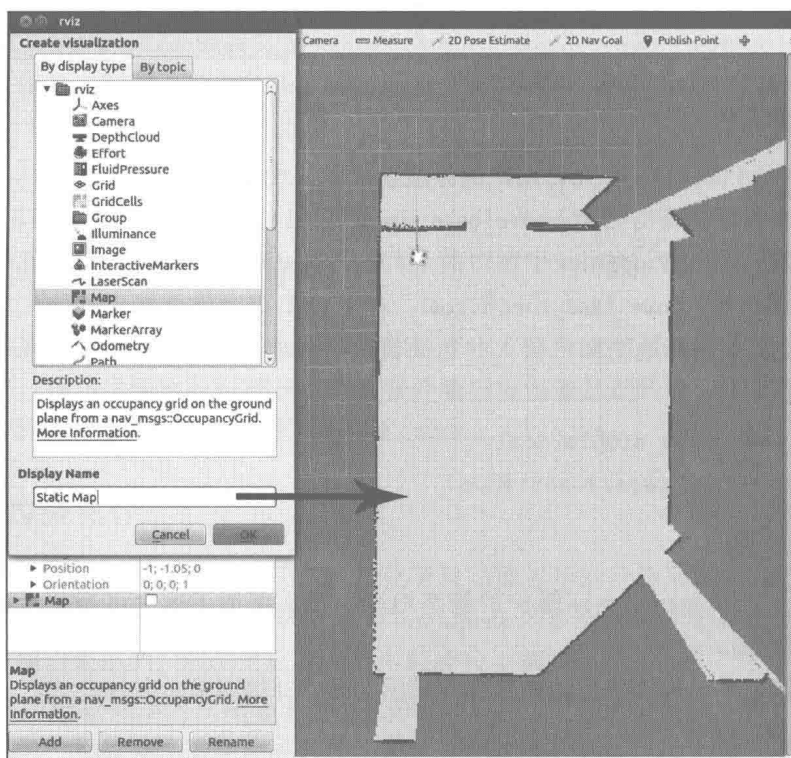
- **Topic:** move_base_simple/goal
- **Type:** geometry_msgs/PoseStamped

9.5.3 静态地图

下页下图显示了 `map_server` 维护的静态地图。当你增加了这个可视化后，就会看到在 8.6 节中获取的地图。

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。

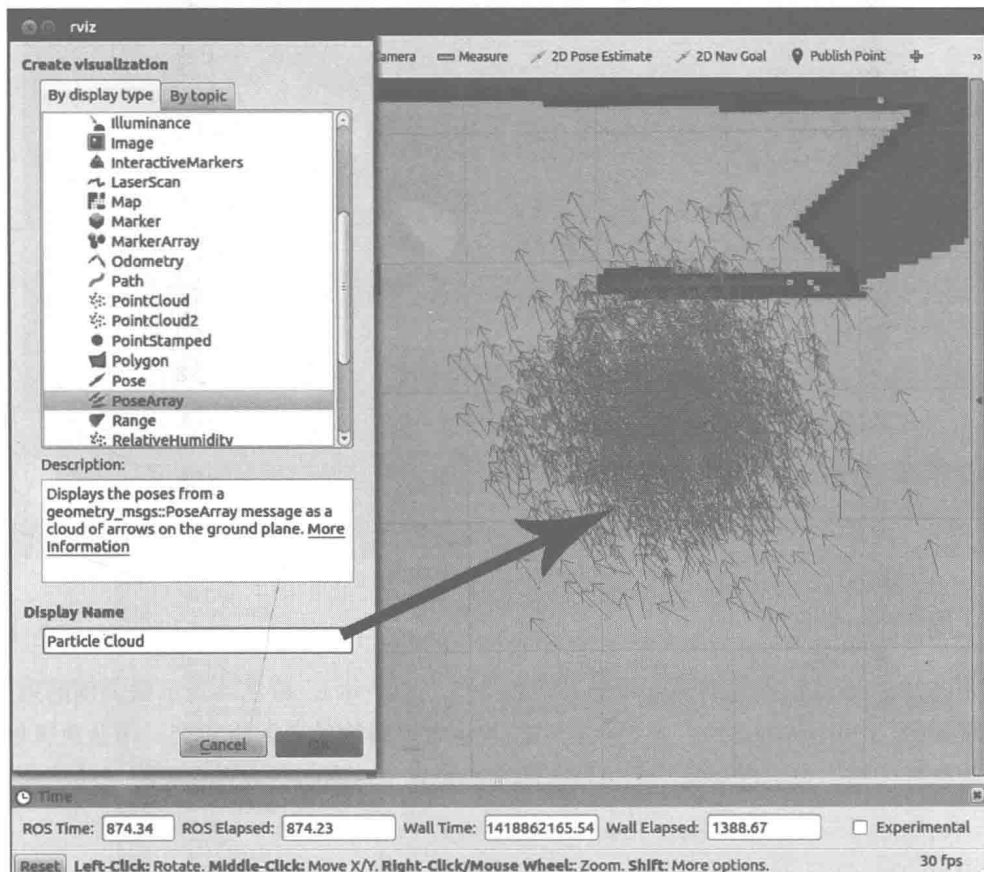
- **Topic:** map
- **Type:** nav_msgs/GetMap



9.5.4 粒子云

下图显示了用于机器人定位系统的粒子云。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，你将为机器人添加如下的点云。

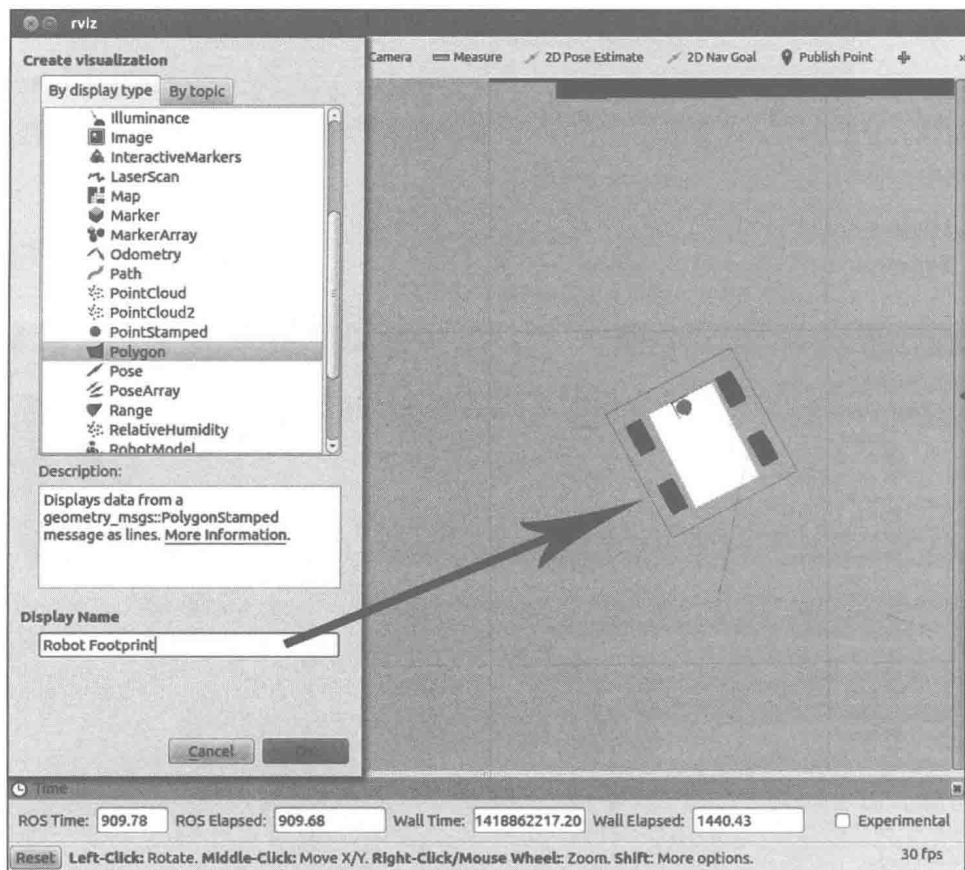
- **Topic:** particlecloud
- **Type:** geometry_msgs/PoseArray



9.5.5 机器人占地空间

下图显示了机器人的占地空间。在我们的示例中，机器人是有占地空间的。它的占地空间有 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



9.5.6 局部代价地图

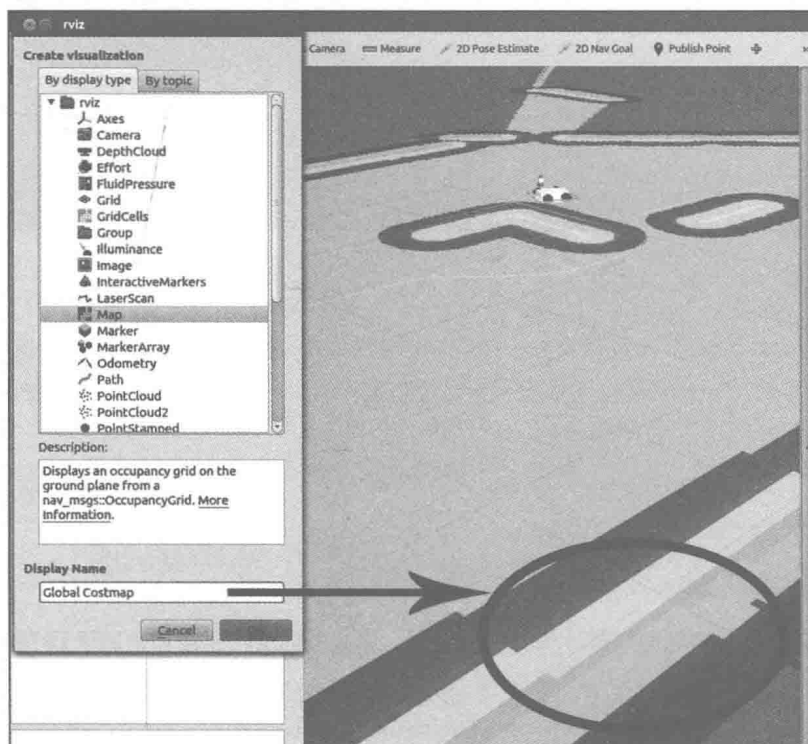
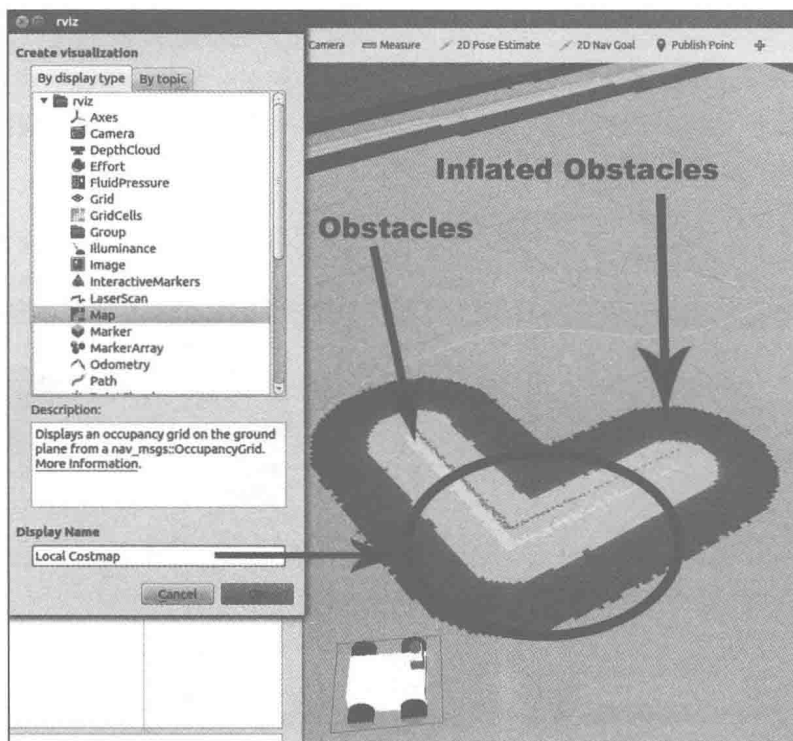
接下来显示导航包集的局部代价地图（如下页上图所示），黄色线表示检测到的障碍物，为了实现避障，机器人的立足点应当永远不能与包含障碍物的单元格相交。用蓝色区域表示障碍物的膨胀区域，为实现避障，则机器人的中心位置不应在包含障碍物膨胀区的单元格内。

- **Topic:** /move_base/local_costmap/costmap
- **Type:** nav_msgs/OccupancyGrid

9.5.7 全局代价地图

接下来显示导航包集的全局代价地图（如下页下图所示），黄色线表示检测到的障碍物，为了实现避障，机器人的立足点应当永远不能与包含障碍物的单元格相交。用蓝色区域表示障碍物的膨胀区域，为实现避障，则机器人的中心位置不应在包含障碍物膨胀区的单元格内。

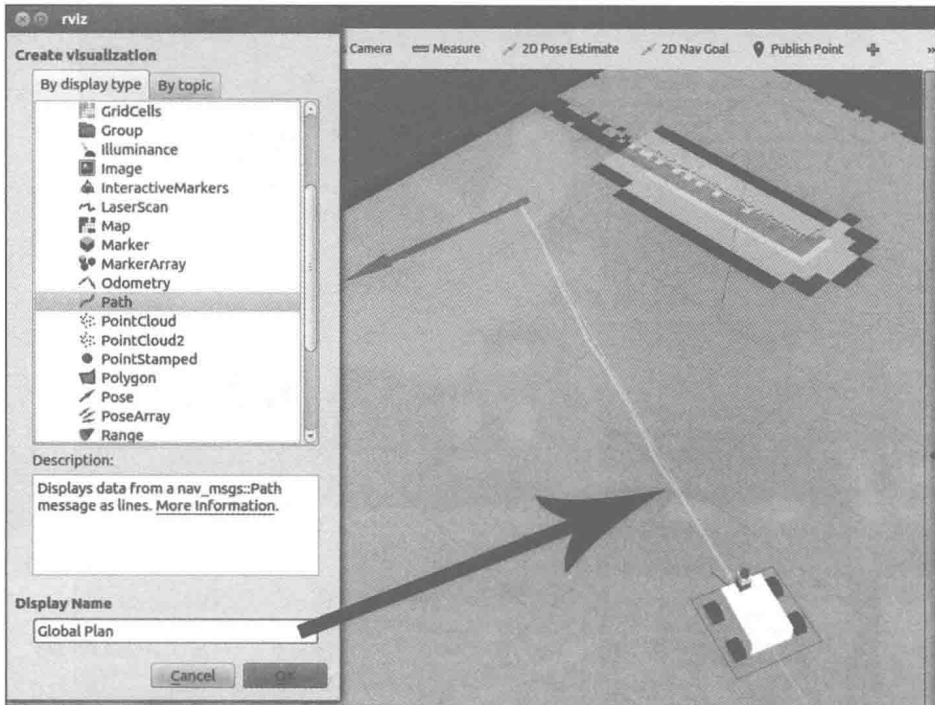
- **Topic:** /move_base/global_costmap/costmap
- **Type:** nav_msgs/OccupancyGrid



9.5.8 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



9.5.9 局部规划

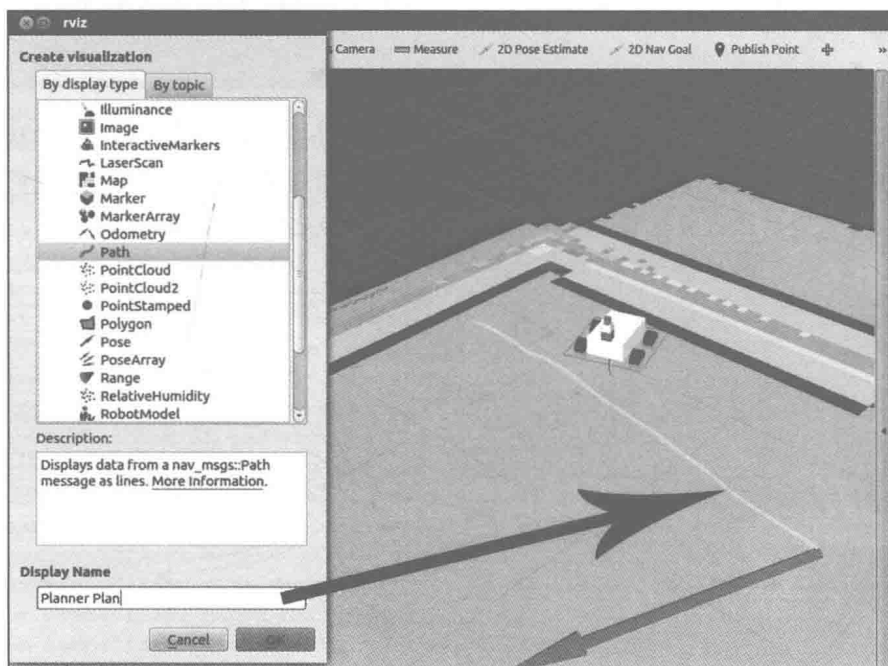
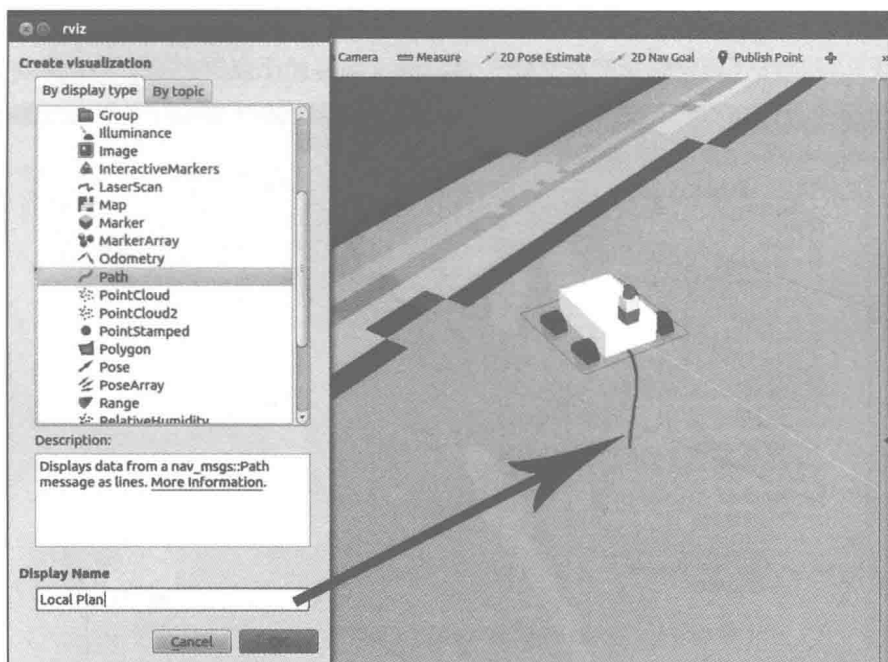
下页上图显示了机器人执行着由局部规划器生成的速度命令以及将会形成的运动轨迹。你能看到其显示为蓝色的线条。你通过这个状态显示能够了解到机器人是否在运动，并根据蓝色线条的长度对机器人的运动速度进行估计。

- **Topic:** TrajectoryPlannerROS/local_plan
- **Type:** nav_msgs/Path

9.5.10 规划器规划

下页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。

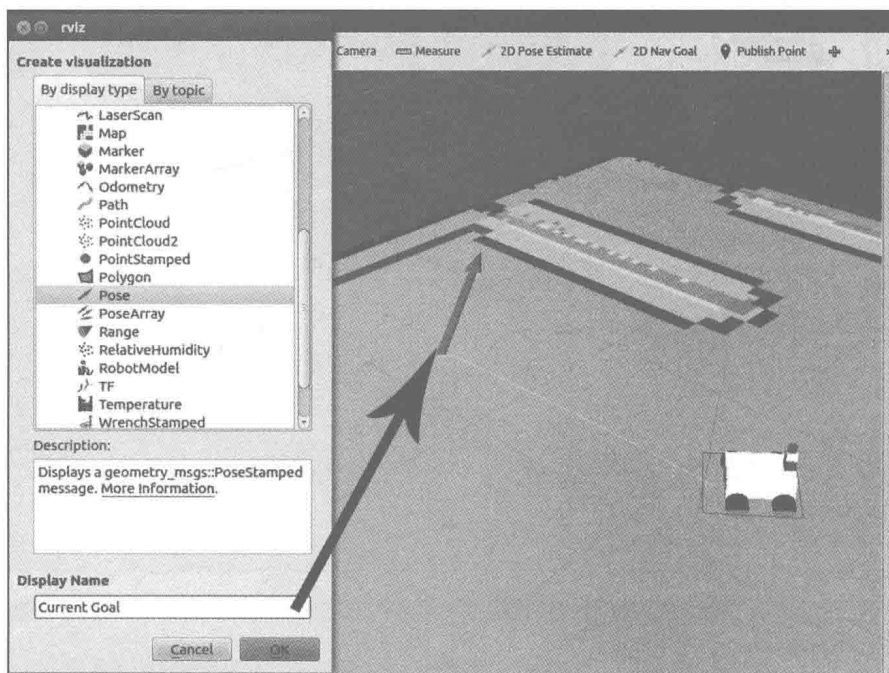
- **Topic:** NavfnROS/plan
- **Type:** nav_msgs/Path



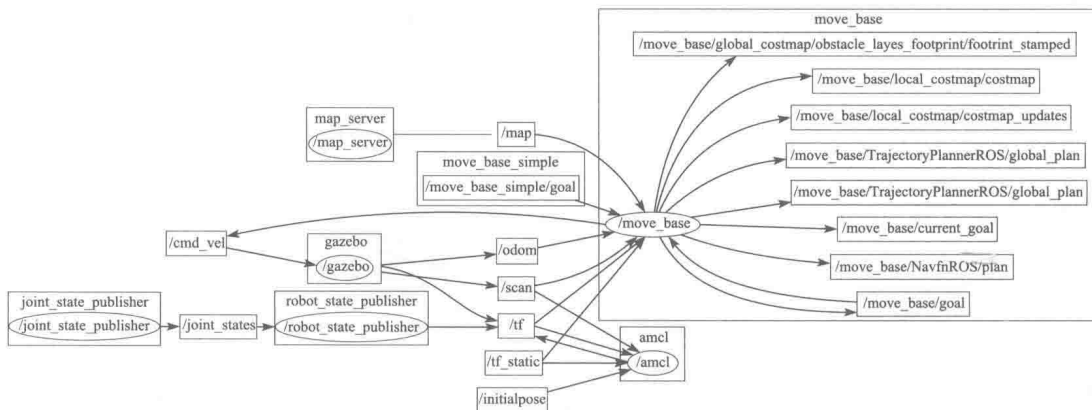
9.5.11 当前目标

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，你重新配置一个新的 2D 导航目标后它才会出现。它用于了解机器人最终的导航目的地。

- Topic: current_goal
- Type: geometry_msgs/PoseStamped



这些可视化数据是所有你需要在 rviz 中查看的导航功能包集。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 `rqt_graph` 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。



9.6 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> 以及 <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激光雷达生成的地图，传递消息并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驱动算法运行的最小设置。

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_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"/>
</node>
```

```

<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`。但在算法中, 也可以设置为 `beam` 激光雷达。

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。但可以通过使用它们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位置 (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来调试机器人, 使定位算法获得最佳的计算结果。在 <http://wiki.ros.org/amcl> 页面上, 你能够找到很多关于这些参数具体配置和能改变的信息。

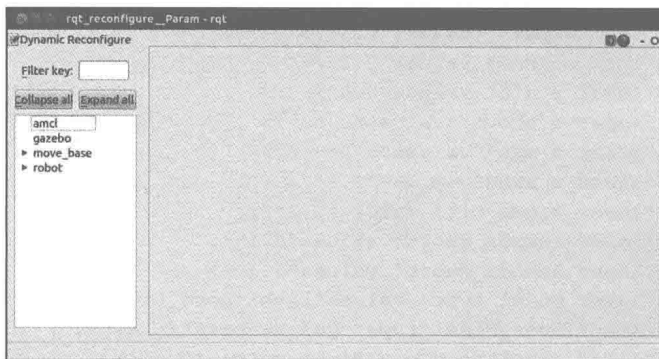
9.7 使用 `rqt_reconfigure` 修改参数

通过使用 `rqt_reconfigure` 在不重新启动仿真的情况下对参数值进行修改, 能够更好地对本章中所用的所有参数进行理解。

使用以下指令启动 `rqt_reconfigure`:

```
$ rosrund rqt_reconfigure rqt_reconfigure
```

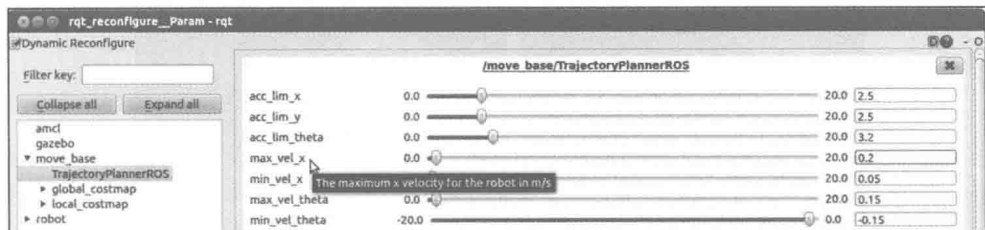
你将看到以下窗口:



作为示例, 我们将对 `base_local_planner_params.yaml` 文件中的 `max_vel_x` 参数进行修改

来演示使用 `rqt_reconfigure` 进行参数修改的过程。单击上图中的“`move_base`”菜单，展开该菜单项。然后在菜单树下选择“`TrajectoryPlannerROS`”，你将看到一个参数列表。如你所见，`max_vel_x` 参数的数值和我们在配置文件中设置的相同。

将鼠标悬停在参数名字上几秒钟，你可以看到该参数的一个简短描述，通过这一功能你能够了解每一个参数的含义。

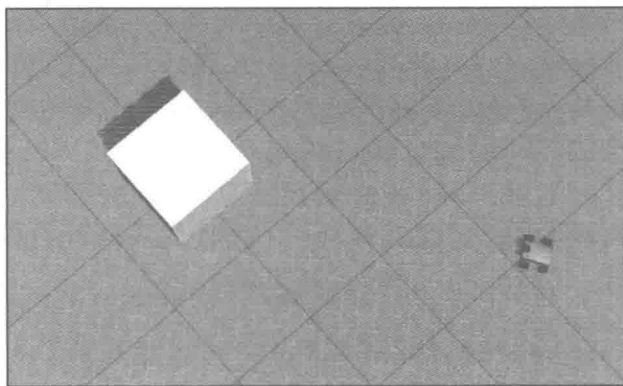


9.8 机器人避障

导航功能包集最重要的一个功能就是在机器人遇到障碍物时，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。

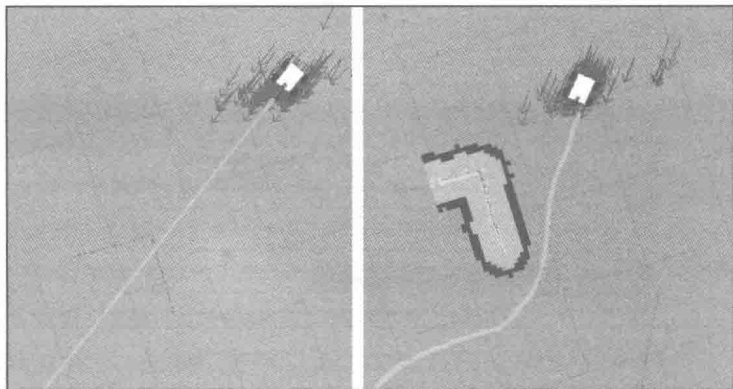
在下图中，你能够看到我们添加了一个物体。Gazebo 有一些预定义的 3D 物体。它们可以在仿真中与移动机器人、手臂、类人机器人等一起使用。

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的位置，如下图所示。



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 4×4 m 的一个区域）。



9.9 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，一个按钮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坐标来检测路径上的障碍物，使用激光雷达扫描等。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的能力，这些功能就无法通过服务来实现了。更进一步说，`actionlib` 创建消息（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 `chapter9_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;
}

add_executable(sendGoals src/sendGoals.cpp)
target_link_libraries(sendGoals ${catkin_LIBRARIES})

```

在你的 CMakeList.txt 文件中添加以下文件来为程序编译生成可执行文件：

```
$ catkin_make
```

现在，使用以下命令编译功能包：

```
$ roslaunch chapter9_tutorials chapter9_configuration_gazebo.launch
```

```
$ roslaunch chapter9_tutorials move_base.launch
```

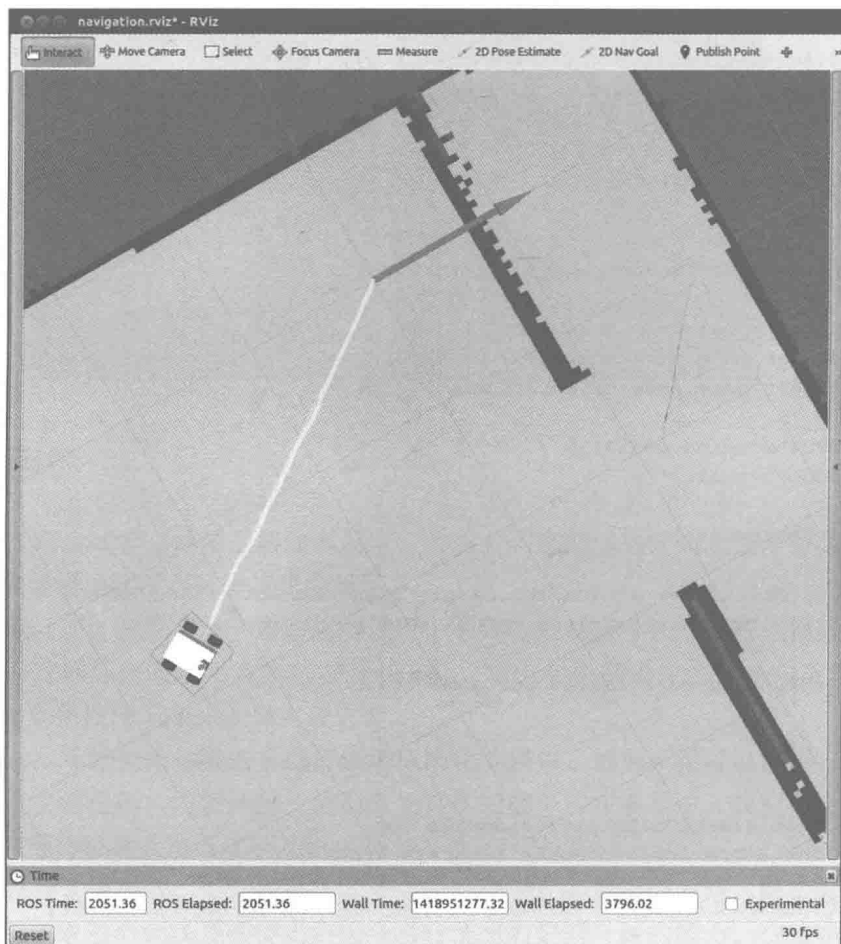
下一步就是使用启动文件启动仿真环境并测试新程序。使用下面的指令来启动所有节点和配置：

```
$ rosrunc chapter9_tutorials sendGoals
```

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：

```
$ rosrn chapter9_tutorials sendGoals
```

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，并帮助你的机器人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和查看其他房间的情况。

9.10 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，使用导航功能包集，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和机器人定位算法，以便你可以较容易地完成机器人的全部配置。本章中的难点在于理解各个配置参数并学会如何使用这些参数。参数的正确使用决定了机器人能否正常工作。所以你必须检查和调整参数，看看机器人的反应如何。

在下一章中，你将通过一些教程和例子学习如何使用 MoveIt!。可能你不熟悉 MoveIt!，它是一种开发移动机械臂的软件。使用它，可以很简单地移动关节连接的机器人。

第 10 章

使用 MoveIt!

ROS 中有针对机器人进行移动操作的一套工具——MoveIt!。在主页 <http://moveit.ros.org> 上包含使用 MoveIt! 的文档、教程、安装说明以及多种机械臂（或机器人）的示例演示，如一些移动操作任务，包括抓握、拾取和放置，或简单的逆向运动学的运动规划。

这个库包含一个快速的逆运动学解算器（作为运动规划单元的一部分）、先进的操作算法、三维感知抓握（通常以点云的形式）、运动学、控制和导航等功能。除了后台功能，它还提供了一个易于使用的图形用户界面（GUI）通过 MoveIt! 和 rviz 插件配置新的机械臂，使用户能以直观的方式进行运动规划任务的开发。

在本章中，我们将学习如何使用 URDF 格式创建一个简单的机械臂，以及使用 MoveIt! 配置工具定义运动规划群组。对于每个机械臂，我们将对应一个群组，之后就可以使用逆运动学解算器去执行由 rviz 接口设定的操作任务。最后通过一个抓取和放置任务来展示 MoveIt! 的功能及工具的使用。

10.1 节介绍 MoveIt! 的体系结构，解释这套框架中所用到的基本概念（如关节群组和规划场景），以及一般性的概念（如轨迹规划、（逆）运动和碰撞检测等）。然后，将介绍如何将机械臂集成到 MoveIt! 中，创建规划群组和场景。接下来，将展示如何在有障碍物的环境中进行运动规划，以及如何结合点云数据，避免碰撞到动态障碍物。

最后，介绍感知和目标识别工具，然后将其应用于抓取和放置示例的演示。在这个示例中，我们将在 rviz 使用 MoveIt! 插件。

10.1 MoveIt! 体系结构

MoveIt! 的体系结构如图 1 所示，内容摘自官方文档的概念部分（<http://moveit.ros.org/documentation/concepts/>）。在这里，我们简要介绍主要概念。若想安装 MoveIt!，只需要运行这个命令：

```
$ sudo apt-get install ros-hydro-moveit-full
```

另外，也可以通过在它相应的工作空间运行下面的命令来安装本章附带代码的所有依赖包：

```
$ rosdep install --from-paths src -iy
```

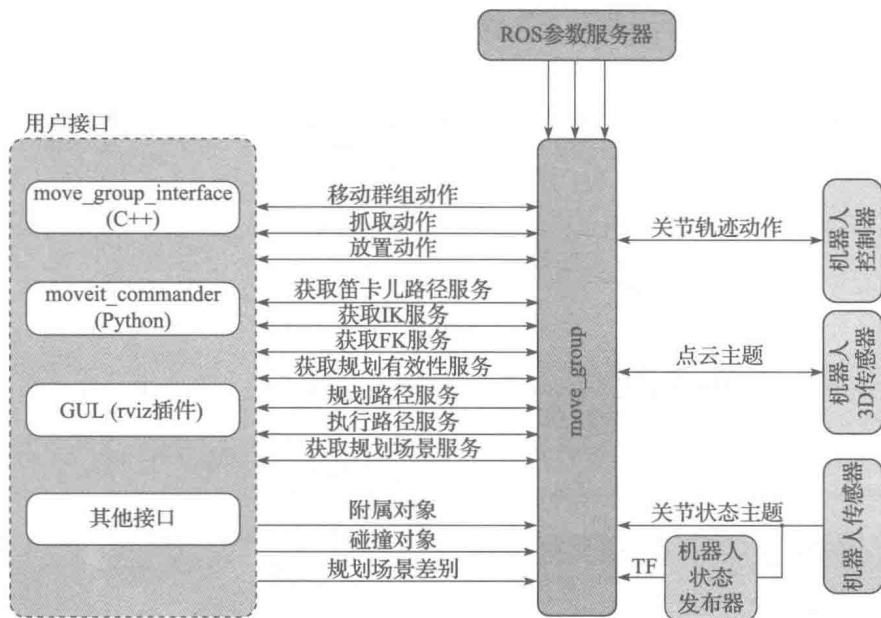


图1 MoveIt! 的体系结构框图

体系结构的中心是 `move_group` 单元。其主要思想是，首先需要定义由关节和其他元件构成的群组（group），然后使用运动规划算法执行移动操作。这类算法考虑与物体交互的场景以及该群组关节的特性。

群组使用标准的 ROS 工具和定义语言进行定义，如 YAML、URDF 和 SDF 等。简而言之，我们必须对关节进行定义，它们是一个群组的部分并包含关节约束。同样，可定义末端执行器的工具，如一个夹持器和感知传感器。机器人必须开放 `JointTrajectoryAction` 控制器，从而使运动规划的输出可以在机器人的硬件（或仿真器）规划执行。为了监控执行情况，需要通过机器人状态发布者发布 `/joint_states`。所有内容都由 ROS 控制并由特定传感器驱动发布。需要注意的是 MoveIt! 提供了一个图形用户界面向导去定义给定的机器人关节群组，可以通过下面的命令直接调用：

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

当 `move_group` 配置正确后，就可以和它连接通信。MoveIt! 提供了 C++ 和 Python 的应用程序接口（API）以及一个无缝集成的 rviz 插件来实现此功能，使我们可以向机器人发送运动目标，规划任务并发送给机器人执行等，如图 2 所示。

10.1.1 运动规划

运动规划解决如何将机械臂移动到期望位置和状态的问题，在使末端执行器到达设定位姿的同时，避免在移动群组时与任何障碍发生碰撞——包括自身连杆或由传感器感知（一般

为点云) 的其他物体, 或违反关节约束。MoveIt! 用户界面允许通过 ROS 行为或服务使用不同的库进行运动规划, 如 OMPL (<http://ompl.kavrakilab.org>)。

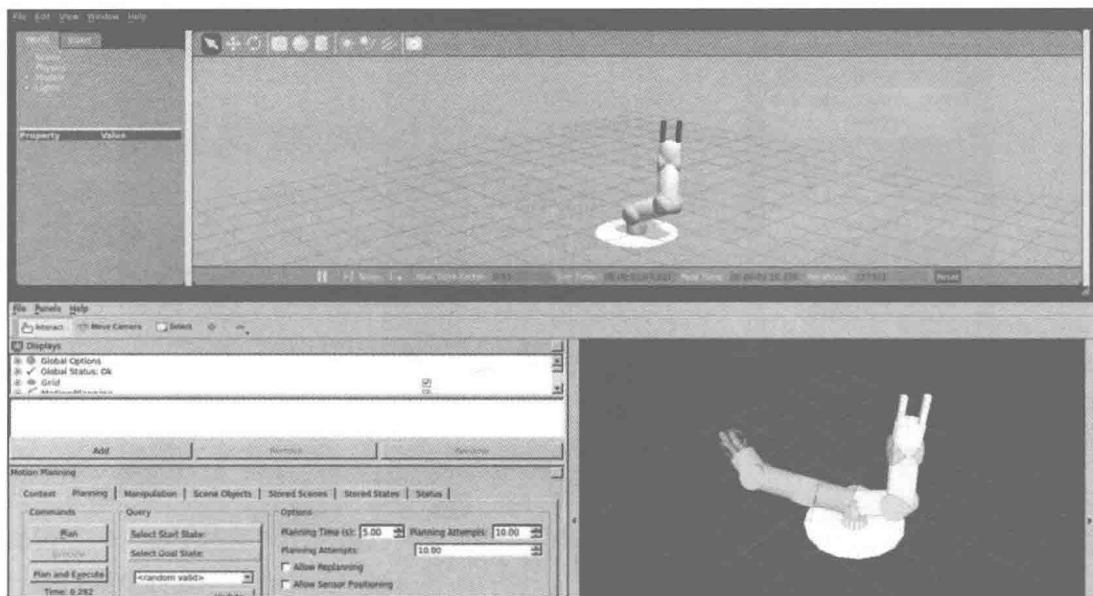


图 2 MoveIt! 集成在 Gazebo 中的机械臂仿真

当一个运动规划请求发给运动规划器后, 就要求机械臂在避免碰撞(包括自身碰撞)的同时, 找到使运动机械臂群组中所有关节到达期望位置的一条轨迹。这样的目标包括关节空间位置或末端执行器位姿, 其中也可能包括目标对象(如用夹持器抓取物体)以及运动约束:

- **位置约束:** 限制连杆的位置。
- **方向约束:** 限制连杆的方向。
- **可见性约束:** 限制某一连杆上的某一点在某一特定区域可见(在传感器能观测到的锥区内)。
- **用户规定约束:** 由用户自定义限制设置。

运动规划的结果是得到一条能使机械臂移动到预定目标位置的轨迹, 这条轨迹需要避免碰撞, 并且满足关节层速度和加速度的约束条件。

最后, MoveIt! 有一个由运动规划器和规划要求适配器组成的运动规划管道。规划要求适配器对运动规划请求进行预处理和后处理。例如, 当机械臂的初始状态在关节约束条件外时, 预处理将起作用。将路径转换为有时间参数的轨迹时, 后处理则非常有用。一些由 MoveIt! 提供的默认运动规划适配器如下:

- **FixStartStateBounds:** 此适配器将初始/开始状态固定在由 URDF 所定义的关节限制范围内。如果没有适配器, 当关节在其限制范围外时, 由于机械臂已经违反了关节限制, 运动规划将无法找到一个可用的规划。此时适配器会将关节移动到关节限制范围

内,但是关节状态处于限制范围内的因素有多种,故而在此情况下适配器未必会采用最佳方案将关节移动到限制范围内。

- **FixWorkspaceBounds** : 定义了一个默认的工作空间来规划一个 $10 \times 10 \times 10\text{m}^3$ 的立体空间。
- **FixStartStateCollision** : 试图在初始位置附近障碍物的环境中,通过少量干预关节状态尝试得到无碰撞的配置。
- **FixStartStatePathConstraints** : 用于初始状态不符合设定的路径约束的情况。它将试图找到一条从初始配置到新配置并满足约束条件的路径,然后以此配置作为初始状态应用运动规划。
- **AddTimeParameterization** : 通常将运动规划(不服从任何速度或加速度约束的路径)通过机器人的 `joint_limits.yaml` 文件中设定的速度和加速度约束生成含有时间参数约束的运动路径。

10.1.2 规划场景

规划场景用于重现机器人状态以及机器人周围的世界环境,由图 3 所示的规划场景监听模块实现。内容可参考官方文档的概念部分 <http://moveit.ros.org/documentation/concepts/>。它是 `move_group` 的一个子部分,监听 `joint_state` 状态、传感器信息(通常指点云)以及由用户在 `planning_scene` 主题中发布的世界几何结构等。

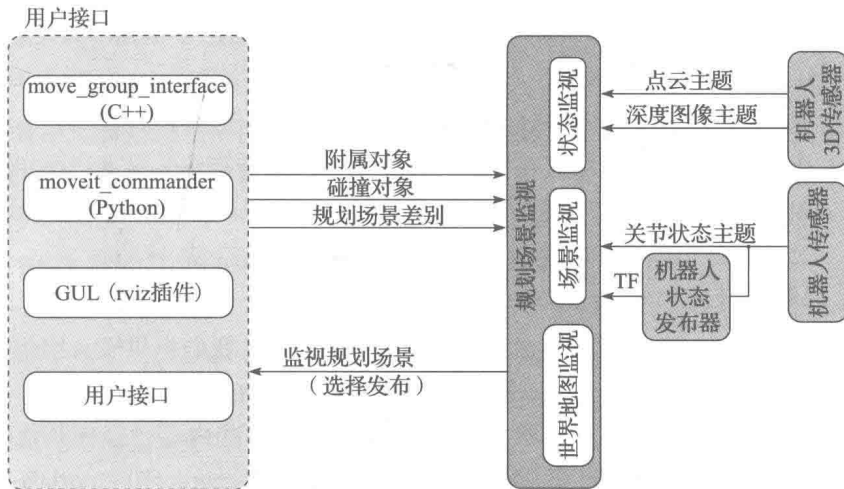


图 3 MoveIt! 规划场景框图

世界几何结构监听模块采用了占有地图监听器来构建机器人周围的三维环境,同时扩充 `planning_scene` 主题的信息,如对象(例如抓取对象);并应用一个三维点云地图 `octomap` 来表示所有这些信息。为了生成三维重现,MoveIt! 支持不同的传感器插件去感知环境,主要

通过下面两种方式：

- 点云 (Point Cloud)：通过点云占有地图更新插件处理。
- 深度图像 (Depth image)：由深度图像占有地图更新插件处理，包括自滤波器，用于消除机器人在深度图中自身可见部分的状态信息。

10.1.3 运动学

正向运动学及其雅可比矩阵都集成在 `RobotState` 类中。另一方面，为了求解逆运动学，`MoveIt!` 提供了一个基于数字化雅可比矩阵解算器的默认插件，可以由设置助手自动配置。与 `MoveIt!` 的其他部件一样，用户可以编写自己的逆运动学插件，如 `IKFast`。

10.1.4 碰撞检测

规划场景的 `CollisionWorld` 对象使用弹性碰撞库 (Flexible Collision Library, FCL) 功能包进行碰撞检测配置。支持碰撞检测的对象包括网格、原始形状 (例如，方形、圆柱体、圆锥体、球体和圆盘等) 以及三维点云地图 `octomap`。

碰撞检测是一个非常费时的操作，通常占整个运动规划过程的 90%。由于这个原因，这里在允许碰撞矩阵 (Allowed Collision Matrix, ACM) 中使用布尔值编码，用于设置是否需要两个物体 (在机器人上或在环境中) 进行碰撞检测。值设置为 1 时，表示不需要对这两个物体进行碰撞检测。例如当这两个物体相距很远时，永远也不可能碰撞。

10.2 在 MoveIt! 中集成一个机械臂

在本节中，我们将通过不同的步骤实现在 `MoveIt!` 中使用一个机械臂。尽管一些内容将在本章后面讲到，但仍然需要事先补充几个基础知识，如手臂描述文件 (URDF)，以及一些在 `Gazebo` 运行的组件。

10.2.1 工具箱里有什么

为了更容易理解将一个机械臂集成到 `MoveIt!` 中的过程，我们提供了一组包含所有必要配置的功能包括机器人描述文件、`launch` 脚本和模块等来将 `MoveIt!` 集成到 `ROS`、`Gazebo` 和 `rviz` 中。我们不会具体介绍如何将机器人导入 `Gazebo` 中，这些内容已经在其他章节中介绍过，但会详细说明如何将 `MoveIt!` 集成到 `Gazebo` 中。在本章 `chapter10_tutorials` 文件夹下的资源库中，提供了下列软件包：

- `chapter10_tutorials`：这个库相当于一个容器，包含了将在本章中使用的软件包。这种结构通常需要综合功能包告诉 `catkin` 这些包是松散关联的。因此，这个包是资源库的综合功能包。
- `rosbook_arm_bringup`：集中了启动控制器和 `MoveIt!` 以及 `play_motion` 的实用程序，

可以用于请求预定义机械臂的配置，以及初始化实际或者仿真中的机器人。

- `rosbook_arm_controller_configuration`：包括移动机械臂需加载的控制器启动文件。这些轨迹控制器（JointTrajectoryController）用于支持 MoveIt! 运动规划。
- `rosbook_arm_controller_configuration_gazebo`：包含关节轨迹控制器的配置文件。此配置也包括在 Gazebo 中控制中机械臂需要的 PID 值。
- `rosbook_arm_description`：包含描述机器人手臂所需的全部元素，包括 URDF 文件（实际上为 xacro）、网格和配置文件。
- `rosbook_arm_gazebo`：最重要的功能包之一，包含 Gazebo 启动文件，也可以启动 MoveIt! 仿真环境和控制器，以及负责运行的启动文件（主要调用在 `rosbook_arm_bringup` 中的启动文件和所有以前的包）。它还包括用于包含可交互对象的世界描述。
- `rosbook_arm_hardware_gazebo`：此功能包使用 ROS 控制插件来模拟在 Gazebo 的关节。它使用机器人描述注册不同关节和执行器，以便能够控制它们的位置。这个功能包完全独立于 MoveIt!，但需要与 Gazebo 集成。
- `rosbook_arm_moveit_config`：通过 MoveIt! 生成设置助手。包含了 MoveIt! 和 rviz 插件所需的大部分启动文件，以及一些 MoveIt! 的配置文件。
- `rosbook_arm_snippets`：除了抓取和放置示例，此功能包包含本章中使用的所有代码段。
- `rosbook_arm_pick_and_place`：此功能包是本书中最庞大和最复杂的示例，包含一个如何使用 MoveIt! 完成对象的抓取和放置的文档。

10.2.2 使用设置助手生成一个 MoveIt! 包

MoveIt! 提供了一个友好的用户图形界面用于导入一个新的机械臂。设置助手基于用户提供的信息负责生成全部的配置文件和 launch 脚本。一般而言，这是开始使用 MoveIt! 最简单的方法。同时还会生成几个范例 launch 脚本，系统可以在无真实机械臂或仿真情况下运行。

若要启动设置助手，在终端中运行下面的命令：

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

当命令执行后，会出现一个类似图 4 所示的窗口。此时，我们的目标就是生成一个新的配置，所以那个按钮（Create New MoveIt Configuration Package）就是我们的目标。一旦该按钮被按下，配置助手便会请求机器人手臂的 URDF 或 COLLADA 模型，以我们的机械臂为例，库功能包在下面位置中：`rosbook_arm_description/robots/rosbook_arm_base.urdf.xacro`。

请注意提供的机器人描述文件是 XML Macros 宏（Xacro）格式，这样更容易生成复杂的 URDF 文件。当机器人描述文件被加载时，用户需要通过每个选项卡添加所需的信息。第一个选项卡如图 5 所示，用来生成自碰撞矩阵。幸运的是对于用户而言，这一过程可以通过简单地设置采样密度（或使用默认值）自动进行，单击重新生成默认碰撞矩阵（Regenerate Default Collision Matrix）按钮。为了提高运动规划性能，碰撞矩阵包含了有关连杆碰撞的信息。图 5 给出了详细的说明。



图 4 MoveIt! 设置助手的初始屏幕

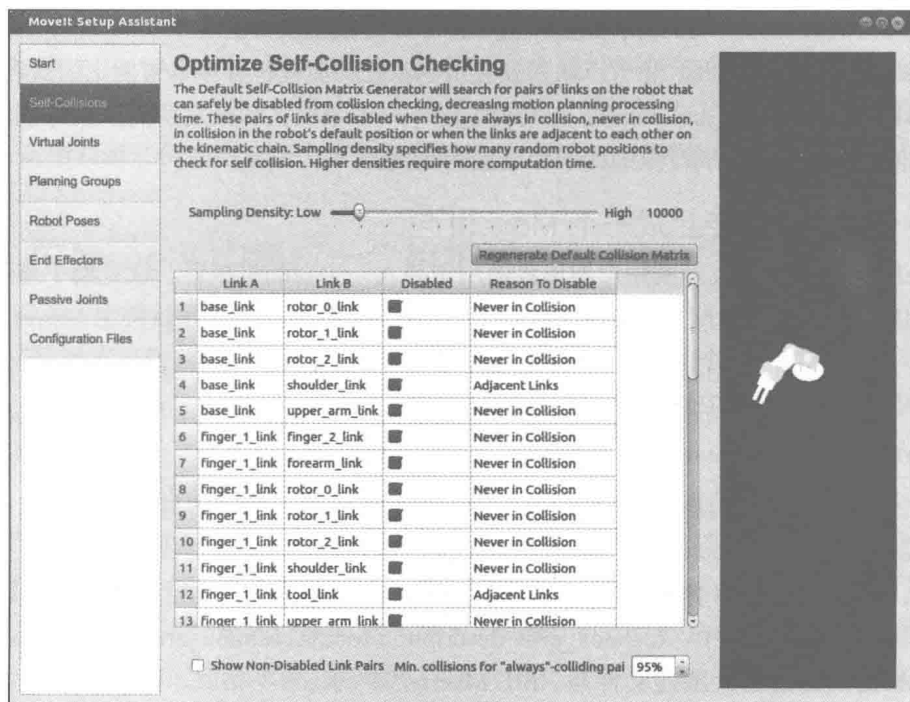


图 5 MoveIt! 设置助手的自碰撞监测选项卡

第二个选项卡如图 6 所示，用于分配机器人的虚拟关节。虚拟关节用于连接世界环境和机械臂，它随着机器人位姿的变化而改变，但在机械臂底座不会移动的特定情况下，不需要虚拟关节。当机械臂不固定在某一个地方时，就需要虚拟关节，例如，当机械臂在移动平台上的顶端，

就需要一个对应里程计的虚拟关节，这是因为 base_link（底座）相对于里程计（odom）框架移动。

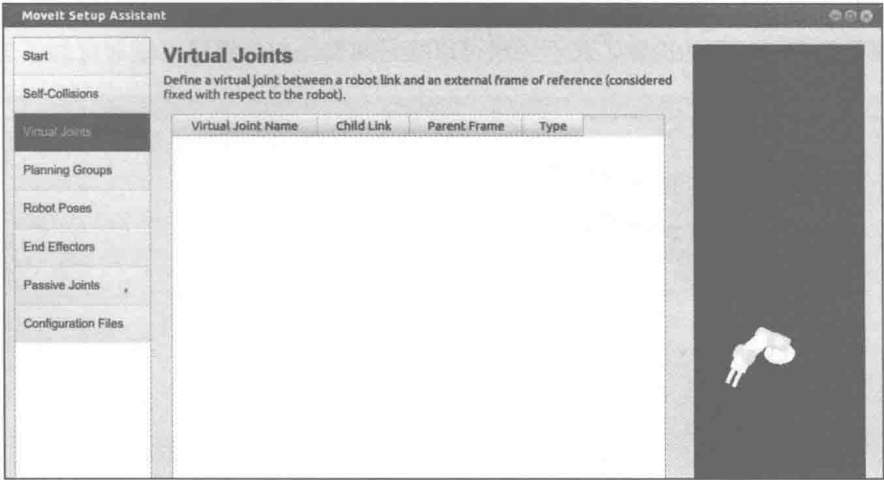


图 6 MoveIt! 设置助手上的虚拟关节选项卡

第三个选项卡如图 7 所示，需要定义机器人手臂的规划群组。规划群组顾名思义是需要一起进行规划的一组关节，以实现特定的连杆或末端执行器上的给定目标。这里，需要定义两个规划群组：一个是机械臂本身，另一个是夹持器。然后分别对机械臂位姿和夹持器动作进行规划。

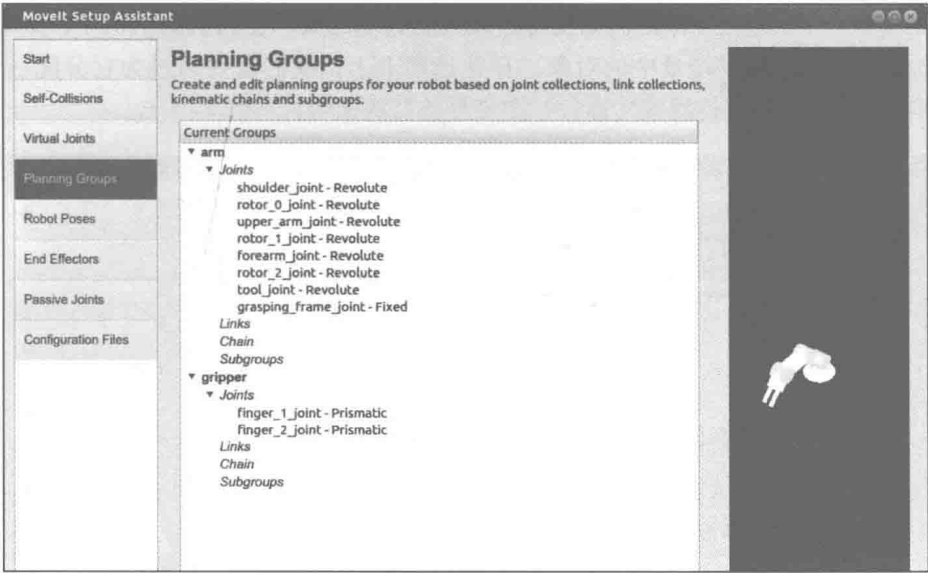


图 7 MoveIt! 设置助手的规划群组选项卡

第四个选项卡如图 8 所示，可以定义机器人已知的位姿以便以后能够引用。这些预定义的位姿也被称为群组状态。如图所示，我们已经建立了两个不同的位姿：机器人的初始位置，

它对应于“储存”机械臂的位置以及夹持器位置，顾名思义，应该让机器人来抓握场景中的对象。实际情况下，设置已知的位姿有很多好处，例如，通常需要一个规划时的初始位置、一个在容器里安全储存机械臂的位置，甚至一组可以随时间变化与位置精度相对应的已知位置。

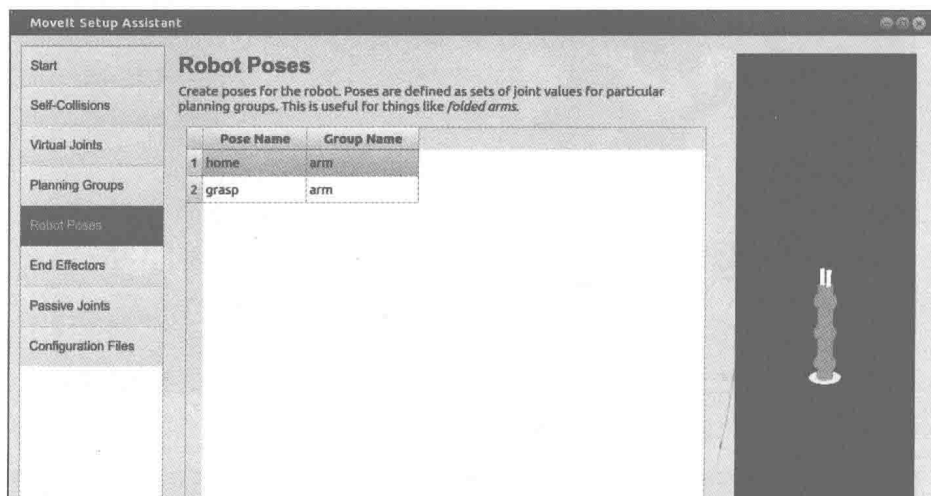


图 8 MoveIt! 设置助手的机器人位姿选项卡

第五个选项卡定义机器人手臂末端执行器，如图 9 所示。如我们先前讨论的，机械臂通常有末端执行器用于执行操作，例如夹持器或某些其他工具。在我们的示例中，末端是夹持器，这使机器人能够抓取场景中的对象。在此选项卡上，我们需要通过为它分配一个名称、一个规划群组 and 含末端执行器的父连杆，定义夹持器的末端执行器。

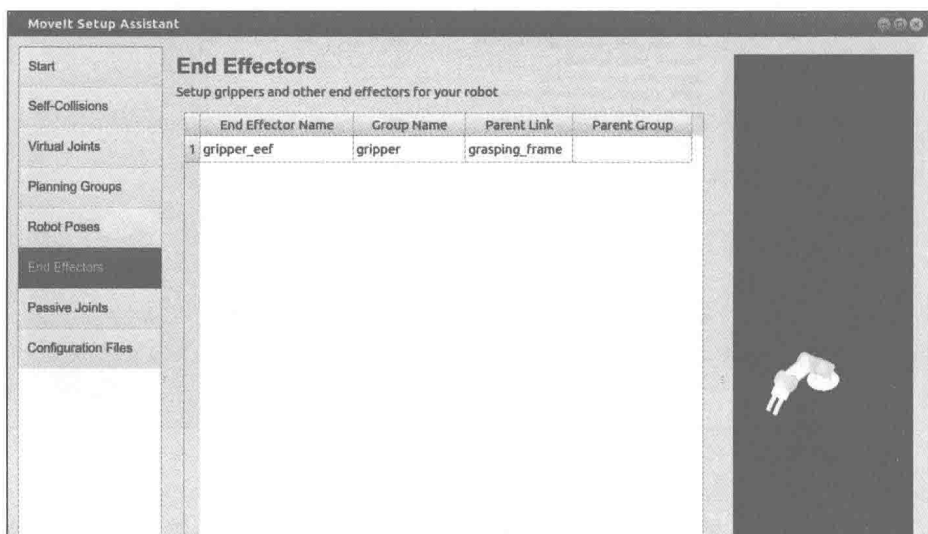


图 9 MoveIt! 设置助手的末端执行器选项卡

第六个选项卡如图 10 所示，是定义不能驱动关节的一个可选的配置步骤。这些关节的一个重要特点是 MoveIt! 不需要规划它们，模块也不需要发布关于它们的信息。机器人中被动关节的一个例子可能是脚轮（caster），但由于所有被动关节已经被定义为固定的关节，所以这里我们会跳过此步骤，最终的运动规划将会产生相同的效果。

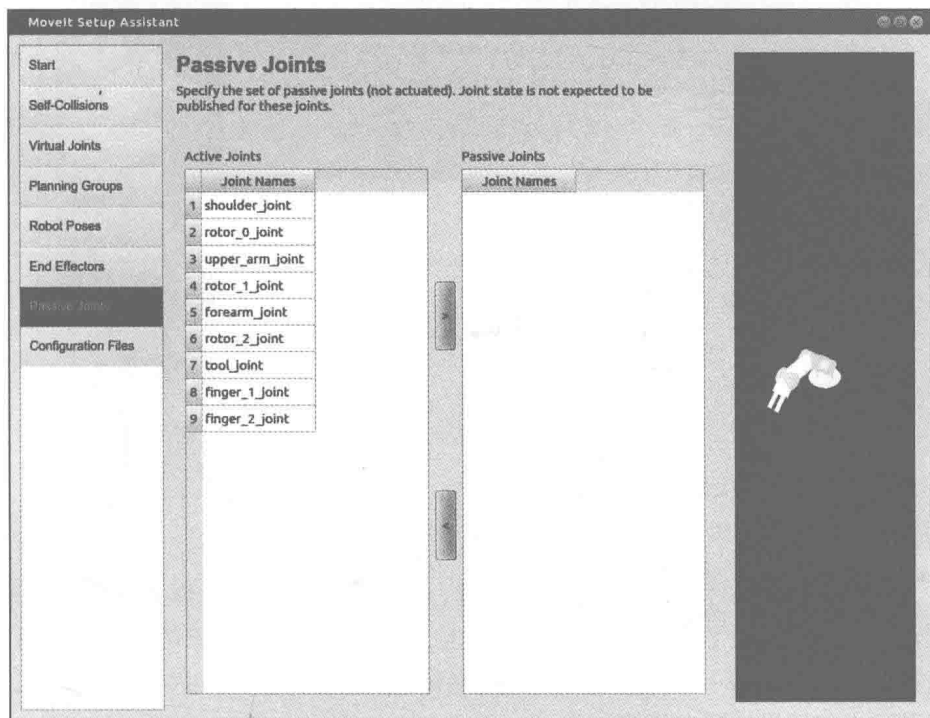


图 10 MoveIt! 设置助手的被动关节选项卡

如图 11 所示，设置助手的最后一步是生成配置文件。在此步骤中唯一需要做的是提供 MoveIt! 创建的配置功能包路径。配置功能包包含大部分我们从 MoveIt! 正确启动控制机器人手臂所需要 launch 和配置文件。

10.2.3 集成到 rviz

MoveIt! 提供了一个非常有用和完整的 rviz 插件，使用户能够执行一些操作，如规划不同的目标、在场景中添加和移除对象等。设置助手通常会创建若干启动文件，其中还包括一个演示，负责启动 MoveIt! 以及仿真控制器、rviz 和插件等。通过运行下面的命令启动示例：

```
$ roslaunch rosbook_arm_moveit_config demo.launch
```

一旦 rviz 启动，将会出现运动规划面板以及可视化的机械臂。需要重点考虑的是规划选项卡和场景对象选项卡。在规划选项卡中，用户将能够规划不同的目标位置，执行它们，并设置一些常见的规划选项。在场景对象选项卡中，可以在规划的场景中插入和删除对象。

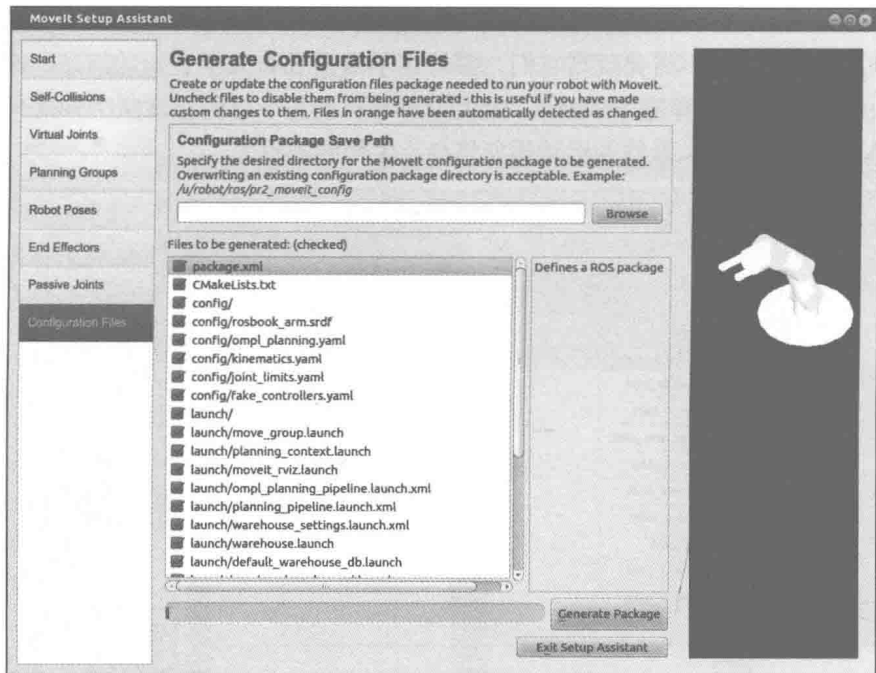


图 11 MoveIt! 设置助手的生成配置文件选项卡

图 12 是规划选项卡，以及分别以白色和橙色可视化显示的机械臂。前者是机械臂的当前状态，而后者是由用户设定的目标位置。这里，可使用查询面板中的工具生成目标位置。一旦对目标状态满意，接下来一步或是通过可视化的方式演示手臂如何移动，或既包括执行可视化运动，也包括机械臂本身的运动。

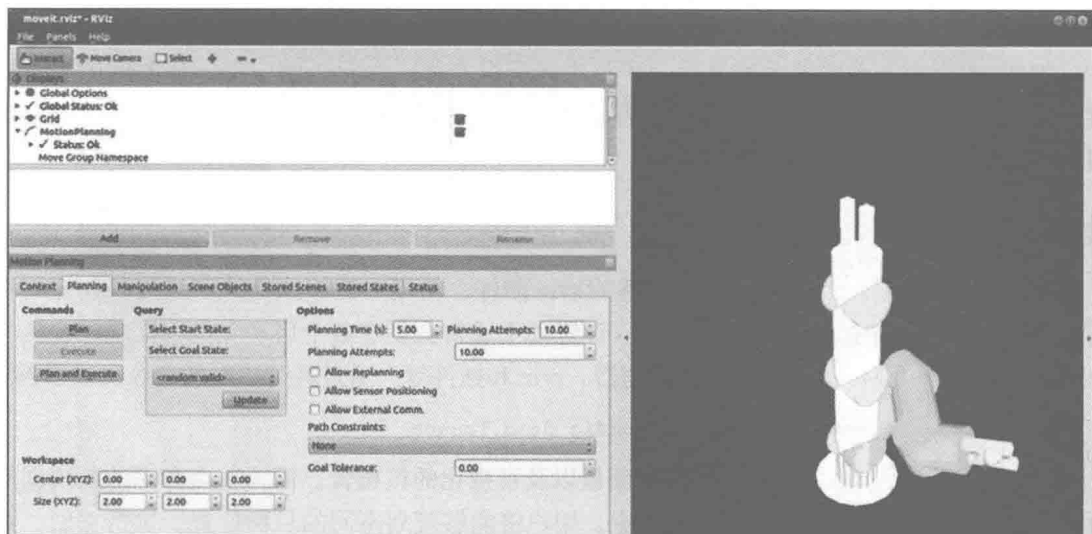


图 12 rviz 插件中的规划选项卡和目标位置可视化

其他选项，如时间规划和尝试次数规划，可以为复杂的目标进行相应的调整，但在大多数示例的情况下不需要更改这些参数。另一个重要参数是定义目标误差，它定义了机械臂需要到达的目标位置与已经达到的位置之间的距离有多远。

可能会出于某些兴趣来规划随机目标，rviz 插件提供了另一个层面的规划。如图 13 所示，可视化机械臂的末端执行器上有一个标记，此标记允许我们定位机械臂末端，以及绕各个轴进行旋转。你现在可以使用此标记来设置机械臂朝向更有趣的配置。



图 13 在 rviz 插件中使用标记来设置目标位置

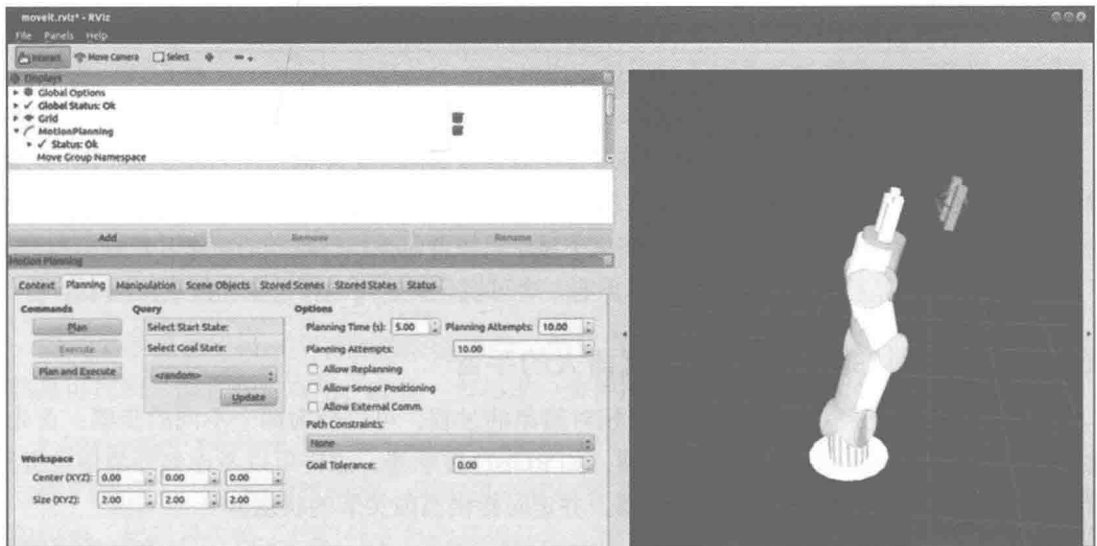


图 14 在 rviz 插件中标记在边界外情况

在许多情况下, 放置标记的规划可能根本没有产生相应的运动, 机械臂显示还在原来的位置, 但标记和机械臂的末端执行器却在别的不同的位置上。这种情况的示例可以在图 14 中看到, 它通常发生在目标位置超出机械臂的运动范围 (如自由度不够、太多约束等)。同样, 当机械臂放置的状态与场景中的物体或本身产生碰撞时, 机械臂将以红色显示碰撞的区域。最后, 图 15 显示了由 MoveIt! 插件的可视化界面提供的不同选项。

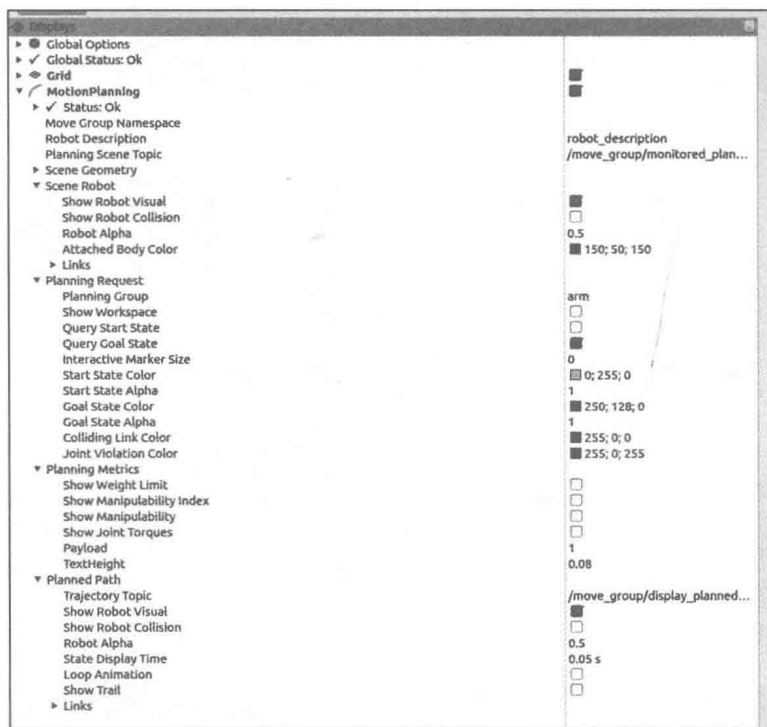


图 15 rviz 插件的运动规划插件选项

显而易见, 所有这些选项提供了调整可视化效果以及添加更多信息的一种方法。用户可能想要修改的其他有趣选项还有轨迹主题 (Trajectory Topic), 就是发布可视化轨迹的主题, 以及查询初始状态 (Query Start State), 这也将从中显示机械臂将要执行规划的状态。在大多数情况下, 初始状态通常是机械臂的当前状态, 但有一个可视化提示可以帮助我们发现算法中问题。

10.2.4 集成到 Gazebo 或实际机器人的手臂

将 MoveIt! 集成到 Gazebo 中是一个相对简单的过程, 可以分为两个不同的步骤: 首先, 我们需要提供 MoveIt! 所需的所有传感器, 如 RGBD 传感器, 这样可以考虑到环境信息进行运动规划, 其次, 还需要提供一个控制器, 并定时提供当前关节的状态。

当在 Gazebo 中创建一种传感器时, 它如正常传感器一样, 只产生与系统交互所需的数据。这个数据接着被 MoveIt! 使用, 过程与实际传感器产生数据的方式完全相同, 用于生成

规划场景中的碰撞物件。传感器通过 MoveIt! 进行处理的过程将在本章后面解释。

关于机械臂（臂和夹持器）的定义，使用 Xacro 文件所提供的 URDF 描述，如同 ROS 中所使用的任何一个机器人。由于运动规划需要为控制器发布消息，在这种情况下使用 MoveIt! 我们需要配置机械臂关节控制器如 JointTrajectoryController。在这种情况下本章用到的机械臂需要两个这种类型的控制器：一个用于臂，另一个用于夹持器。控制器配置文件分别包括 launch 和 config 的 YAML 文件，在 `rosbook_arm_controller_configuration` 和 `rosbook_arm_controller_configuration_gazebo` 功能包中。

ROS 控制提供了这种类型的控制器。为此，我们需要一个 RobotHardware 接口使机械臂在 Gazebo 或实际硬件中真正动起来。在 Gazebo 中和在实际的机械臂上的具体实现是不同的，这里我们只提供前者的示例。本章中使用的 `Rosbook_arm_hardware_gazebo` 功能包中有机械臂 RobotHardware 的 C++ 实现，这是通过接口实现的，于是我们创建一个继承它的新类。然后，关节均通过写入所需的目标位置（使用位置控制）和读取的实际位置进行处理，还包括每个关节的速度和力控制。为了简单起见，我们省略了对于理解 MoveIt! 不需要的细节。然而，如果机械臂的数量变化较大，即使它是通用的并能够从机器人描述自动检测关节的数量，其实现方式也必须进行改变。

10.3 简单的运动规划

rviz 插件提供了一个非常有趣的与 MoveIt! 进行交互的机制，但由于它缺乏自动化，可能会被认为相当具有局限性甚至很麻烦。为了充分发挥 MoveIt! 中的功能，已开发了几个应用程序接口（API），这使我们能够执行一系列的操作，如运动规划、访问机器人的模型和修改规划的场景等。

在下面的内容中，我们将通过几个例子介绍如何执行不同类型的简单运动规划。我们将开始规划一个单一的目标，然后规划一个随机目标，接着进行预定义群组的状态的规划，最后解释如何改进我们与 rviz 相互作用的代码段。

为了简化解释，提供了一组启动文件启动全部所需的功能。其中最重要的一个负责启动 Gazebo、MoveIt! 和机械臂的控制器：

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_empty_world.launch
```

安装助手提供了另一个有趣的启动文件，用于启动 rviz 和运动规划插件。rviz 这款插件虽然是可选的，但它非常有用，在这一节将进一步使用：

```
$ roslaunch rosbook_arm_moveit_config moveit_RViz.launch config:=true
```

我们提供一定数量的代码段，用于进一步解释说明。代码段放在 `rosbook_arm_snippets` 功能包中。代码段包除了代码外不包含其他任何东西，通过调用 `roslaunch` 而不是跟往常一样使用 `roslaunch` 启动代码段。

从典型的 ROS 初始化开始, 本节中每个代码段都遵循相同的模式, 本章不再赘述。初始化之后, 我们需要定义对那些将要执行运动规划的规划群组。在我们的示例中只有两个规划群组, 机械臂和夹持器, 这里我们只关注臂。这将实例化一个规划群组接口, 并负责与 MoveIt! 进行交互, 如下:

```
moveit::planning_interface::MoveGroup plan_group("arm");
```

在规划群组接口实例化后, 通常用一些代码决定规划的目标, 它对应本节包括的每种类型的目标。在目标确定后, 需要传达 MoveIt! 使它执行。下面的代码段负责创建规划和使用规划群组接口去请求 MoveIt! 执行运动规划, 如果成功, 将执行:

```
moveit::planning_interface::MoveGroup::Plan goal_plan;
if (plan_group.plan(goal_plan))
{
    ...
    plan_group.move();
}
```

10.3.1 规划单个目标

规划单个目标, 从字面上理解只需要将目标本身提供给 MoveIt!。目标由 geometry_msgs 功能包的 Pose 消息表示。需要设定方向和位姿。在此例中, 通过执行手动的规划和检查臂的状态得到了这一目标。在实际情况下, 目标可能将会根据机械臂的期望位置进行设置:

```
geometry_msgs::Pose goal;
goal.orientation.x = -0.000764819;
goal.orientation.y = 0.0366097;
goal.orientation.z = 0.00918912;
goal.orientation.w = 0.999287;
goal.position.x = 0.775884;
goal.position.y = 0.43172;
goal.position.z = 2.71809;
```

为了这个特定的目标, 还可以设置误差。我们都知道 PID 控制并非很准确, 可能导致 MoveIt! 认为控制目标没有完成。更改目标误差可使系统考虑到控制的不精确性, 以较高的误差区间到达路径点:

```
plan_group.setGoalTolerance(0.2);
```

最后, 只需要设置规划群组目标位姿, 然后由这一节前面所示的代码段规划和执行:

```
plan_group.setPoseTarget(goal);
```

可以用下面的命令运行这个代码段, 机械臂应该正常地调整自身位置:

```
$ rosrun rosbook_arm_snippets move_group_plan_single_target
```

10.3.2 规划一个随机目标

有效地规划一个随机目标可以通过执行两个步骤实现: 首先, 我们需要创建随机目标

本身，然后检查其有效性。如果有效性得到证实，那么就可以继续像往常一样请求目标。否则，我们将取消（虽然我们可以重试，直到我们找到一个有效的随机目标）。为了验证目标的有效性，我们需要调用由 MoveIt! 为此特定目的提供的服务。像往常一样，要执行一个服务调用，就需要一个服务客户端：

```
ros::ServiceClient validity_srv = nh.serviceClient<moveit_
msgs::GetStateValidity>("/check_state_validity");
```

一旦服务客户端设置完成，就需要创建随机目标。要做到这一点，我们需要创建一个包含随机位置的机器人状态对象，但为了简化过程，我们可以通过获取机器人的当前状态对象来启动，如下：

```
robot_state::RobotState current_state = *plan_group.getCurrentState();
```

接着设置当前机器人状态对象到随机的位置，但这样做，我们需要为这个机器人状态提供关节模型群组。关节模型群组可使用如下所示已创建的机器人状态对象获取：

```
current_state.setToRandomPositions(current_state.
getJointModelGroup("arm"));
```

至此，我们已经有想要验证的一个等待使用的服务客户端以及一个机器人随机状态对象。我们将创建一对消息：一个用于请求，另一个用于响应。使用一个应用程序接口转换函数以随机的机器人状态填写请求消息并请求服务调用：

```
moveit_msgs::GetStateValidity::Request validity_request;
moveit_msgs::GetStateValidity::Response validity_response;

robot_state::robotStateToRobotStateMsg(current_state, validity_
request.robot_state);
validity_request.group_name = "arm";

validity_srv.call(validity_request, validity_response);
```

当服务调用完成后，我们可以检查响应消息。如果状态显示无效，我们将直接停止运行模块，否则我们将会继续。如前所述，在这一点上我们可以重试，直到我们得到一个有效的随机状态。这对于读者而言是一个简单的练习：

```
if (!validity_response.valid)
{
    ROS_INFO("Random state is not valid");
    ros::shutdown();
    return 1;
}
```

最后，我们将会使用规划群组界面为刚刚创建的机器人设置目标状态，然后由 MoveIt! 像往常一样进行规划和执行：

```
plan_group.setJointValueTarget(current_state);
```

可以用下面的命令运行这个代码段，这会使机械臂重新调整自身位姿到一个随机的配置上：

```
$ rosrn rosbook_arm_snippets move_group_plan_random_target
```

10.3.3 规划预定义的群组状态

正如我们在配置生成步骤中所标注的, 在最初导入机械臂时, MoveIt! 提供预定义群组状态的概念, 之后可用于放置机器人为一个预定义的位姿。访问预定义的群组状态需要创建一个机器人的状态对象作为目标。为了做到这一点, 最好的办法是通过从规划群组接口获取机械臂当前状态来启动, 如下:

```
robot_state::RobotState current_state = *plan_group.getCurrentState();
```

一旦我们获得了当前状态, 就可以通过下面的调用将它设置为预定义的群组状态, 使用需要进行修改的模型群组以及预定义群组状态的名称:

```
current_state.setToDefaultValues(current_state.  
getJointModelGroup("arm"), "home");
```

最后, 将使用机械臂新的状态作为新的目标, 让 MoveIt! 像往常一样负责规划和执行:

```
plan_group.setJointValueTarget(current_state);
```

可以用下面的命令来运行这个代码段, 机械臂将重新调整来实现预定义的群组状态:

```
$ rosrn rosbook_arm_snippets move_group_plan_group_state
```

10.3.4 显示目标的运动

MoveIt! 提供一组消息用来与可视化信息进行通信, 从根本上讲, 为规划的路径提供这些消息是为了得到机械臂如何移动到达其目标的良好可视化效果。像往常一样, 需要用广播的方式通过主题进行通信:

```
ros::Publisher display_pub = nh.advertise<moveit_  
msgs::DisplayTrajectory>("/move_group/display_planned_path", 1, true);
```

我们需要发布的消息包括轨迹的起始状态和轨迹本身。为了获得这类信息, 需要先使用规划组接口执行规划, 然后使用创建的规划, 我们可以继续填入消息:

```
moveit_msgs::DisplayTrajectory display_msg;  
display_msg.trajectory_start = goal_plan.start_state_;  
display_msg.trajectory.push_back(goal_plan.trajectory_);  
display_pub.publish(display_msg);
```

消息填入后, 将它发布到正确的主题, rviz 可视化界面就会显示机械臂将要执行的轨迹。需要重点注意的是, 在执行规划的调用时, 它还将显示相同类型可视化效果, 如果运动轨迹显示两次, 不要混淆。

10.4 考虑碰撞的运动规划

让读者知道 MoveIt! 提供了考虑碰撞情况的运动规划可能是有趣的, 我们将介绍如何将

物体添加到可能与我们机械臂发生碰撞的规划场景中。首先，将解释如何将基本对象添加到规划的场景中，即使在我们的场景中不存在一个真正的物体，它也允许我们执行规划。这是相当有趣的。之后，将介绍如何从场景中删除这些物体。最后，将解释如何添加 RGBD 传感器，它会基于真实环境（或模拟）的对象产生点云，从而使我们的运动规划更有趣和实用。

10.4.1 将对象添加到规划场景中

在开始添加对象时，我们需要有一个规划的场景。这只有在 MoveIt! 正运行的时候才可能，因此第一步是启动 Gazebo、MoveIt!、控制器和 rviz。这是由于规划场景只存在于 MoveIt! 中。rviz 需要可视化包含在内的对象。为了启动所有需要的模块，需要运行以下命令：

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_empty_world.launch
$ roslaunch rosbook_arm_moveit_config moveit_RViz.launch config:=true
```

这段代码开始实例化规划场景接口对象，该对象可用于对规划场景本身执行操作：

```
moveit::planning_interface::PlanningSceneInterface current_scene;
```

下一步是创建我们想要通过规划场景接口发送的碰撞对象消息。首先我们需要为碰撞对象提供一个名称，这将唯一标识该对象并将使我们能够对它执行操作。例如一旦当我们完成后，将它从场景中删除：

```
moveit_msgs::CollisionObject box;
```

```
box.id = "rosbook_box";
```

下一步是提供对象本身的属性。通过一个固体原始的信息，设定正在创建对象的类型，并根据对象的类型设定其属性。在我们的示例中，我们直接创建一个方盒（box），它基本上有三个维度：

```
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.2;
primitive.dimensions[1] = 0.2;
primitive.dimensions[2] = 0.2;
```

若要继续，我们需要提供在规划场景中方盒的位姿。由于我们需要可能产生碰撞的情况，所以将方盒放置在靠近机械臂的位置上。这个位姿本身由标准几何消息功能包的位姿消息设定：

```
geometry_msgs::Pose pose;
pose.orientation.w = 1.0;
pose.position.x = 0.7;
pose.position.y = -0.5;
pose.position.z = 1.0;
```

向消息中添加初始的位姿，然后设定我们想要执行的操作并将其添加到规划场景中：

```
box.primitives.push_back(primitive);
box.primitive_poses.push_back(pose);
box.operation = box.ADD;
```

最后，我们添加碰撞对象到其消息向量并调用规划场景接口中的 `addCollisionObjects` 方法，负责通过适当的主题发送需要的消息，以确保在当前规划场景中创建对象：

```
std::vector<moveit_msgs::CollisionObject> collision_objects;
collision_objects.push_back(box);
```

```
current_scene.addCollisionObjects(collision_objects);
```

与前面相同，我们可以通过在终端中运行下面命令测试该代码段。一旦当该对象添加到规划场景中，就要保证 `rviz` 可视化界面已经运行，否则用户将无法看到该对象：

```
$ rosrun rosbook_arm_snippets move_group_add_object
```

结果如图 16 所示，一个简单、绿色的方盒位于机械臂的目标位置和当前位置之间的路径上：

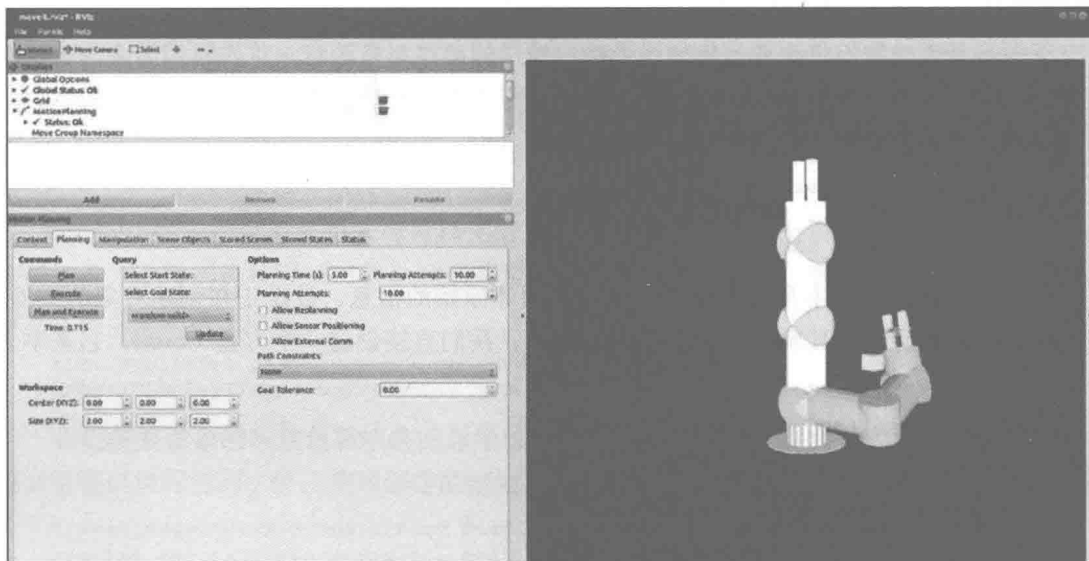


图 16 在 `rviz` 中的场景碰撞对象

10.4.2 从规划的场景中删除对象

从规划场景删除已添加的对象是一个非常简单的过程。按照前面的示例进行相同的初始化，只需要创建一个包含移除对象 ID 的字符串向量，并调用规划场景界面功能中的 `removeCollisionObjects` 函数：

```
std::vector<std::string> object_ids;
object_ids.push_back("rosbook_box");
current_scene.removeCollisionObjects(object_ids);
```

可以通过运行下面的命令来测试该代码段，这将从规划场景中删除通过前面的代码段创建的对象：

```
$ rosrun rosbook_arm_snippets move_group_remove_object
```

另外，也可以使用 rviz 插件中的场景对象选项卡从场景中删除任何对象。

10.4.3 应用点云进行运动规划

应用点云进行运动规划似乎比它看起来要简单得多。我们主要需要考虑的是提供一个点云输入并告诉 MoveIt! 应用点云执行规划。在本章中，我们已经在 Gazebo 仿真中设置好可以提供点云数据的 RGBD 传感器。通过启动下面的命令开始这个示例：

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch  
$ roslaunch rosbook_arm_moveit_config moveit_RViz.launch config:=true
```

用户可能已经注意到 Gazebo 仿真现在似乎在世界场景中包含了几个对象。这些对象由 RGBD 传感器扫描，由此产生的点云数据发布到 /rgb_camera/depth/points 主题上。在这种情况下我们需要做的是告诉 MoveIt! 去哪里获得这类信息以及该信息的格式是什么。需要修改的第一个文件是：

```
rosbook_arm_moveit_conifg/config/sensors_rgbd.yaml
```

这个文件将用于存储 RGBD 传感器的信息。在这个文件中，需要告诉 MoveIt! 需要哪个插件来管理点云数据，以及设定传感器插件本身的一些其他参数。在这种情况下，该插件使用 Octomap 更新器（Octomap Updater），这将由点云数据产生 octomap，缩减采样像素，并发布生成的云。在这一步设立了一个插件，将向 MoveIt! 提供足够信息，应用点云去考虑可能发生碰撞的规划：

```
sensors:  
- sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater  
  point_cloud_topic: /rgb_camera/depth/points  
  max_range: 10  
  padding_offset: 0.01  
  padding_scale: 1.0  
  point_subsample: 1  
  filtered_cloud_topic: output_cloud
```

你可能会疑惑这个文件本身只不过是一个配置文件。下一步需要执行的是加载此配置文件到环境中，这样 MoveIt! 就知道我们已经添加了新的传感器。为了做到这一点，还需要修改下面的 XML 文件：

```
$ rosbook_arm_moveit_conifg/launch/rosbook_arm_moveit_sensor_manager.  
launch.xml
```

在此 XML 文件中，可以设定将使用的传感器插件的一些参数，例如云的分辨率和参考坐标。需要重点注意的是一些参数可能是多余的或者可以省略。最后，需要在配置文件添加一条加载配置文件到环境中的命令：

```
<launch>
  <rosparam command="load" file="$(find rosbook_arm_moveit_config)/
config/sensors_rgbd.yaml" />
</launch>
```

运行我们增加的新命令后的结果如图 17 所示。此时，可以看到 Gazebo 仿真和 rviz 可视化效果。rviz 仿真中包含了点云数据和已经执行的一些手动运动规划，成功地应用点云避免了碰撞。

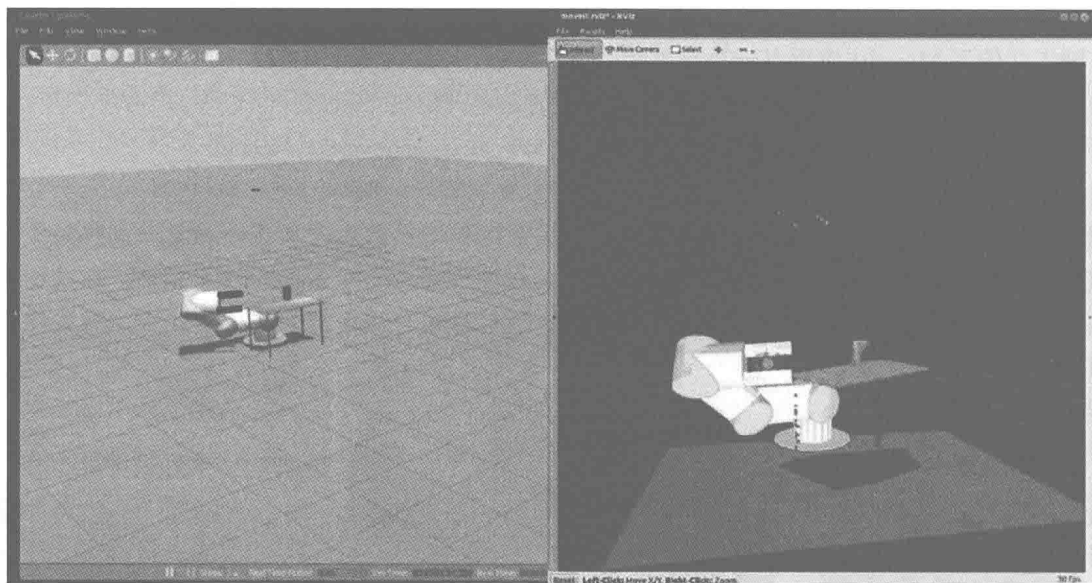


图 17 Gazebo 仿真 (左), rviz 中的点云 (右)

10.5 抓取和放置任务

在本节中，我们要解释如何用机械臂执行一个很常见的应用程序或任务。抓取和放置任务包括拿起一个目标对象：抓住它，并放置到另一个位置。在这里，我们假设对象最初位于支撑面上，支撑面可以是平面或曲面，例如桌子，很容易推广到更复杂的环境。至于要抓取的对象，考虑将其近似为方盒，这样当使用夹持器抓取时就非常简单。对于更复杂的对象，将需要更好的夹持器，甚至一只机械手。

在下面的小节中，我们首先描述如何设置规划场景，这是 MoveIt! 除了机械臂本身外还需要标识的对象。这些对象在运动规划时需要考虑避障，可以被拿起或者抓握。为了简化问题，我们将省略感知的部分，但我们将解释它是如何实现和集成的。一旦定义了规划的场景，我们将描述如何使用 MoveIt! 的应用程序接口完成抓取和放置的任务。最后，我们将解释如何在演示模式下，使用虚拟控制器运行此任务，这样我们就不需要实际的机器人系统（不论在 Gazebo 中或在一个真正机器人上）。我们还将向你展示当仿真机械臂与环境中仿真对

象交互时，如何在 Gazebo 中真正看到仿真机械臂的运动。

10.5.1 规划的场景

我们要做的第一件事是定义对象，因为 MoveIt! 在环境中使机械臂在交互中避碰需要这类信息，并参照它们执行某些操作。在这里，我们将考虑的场景如图 18 所示。

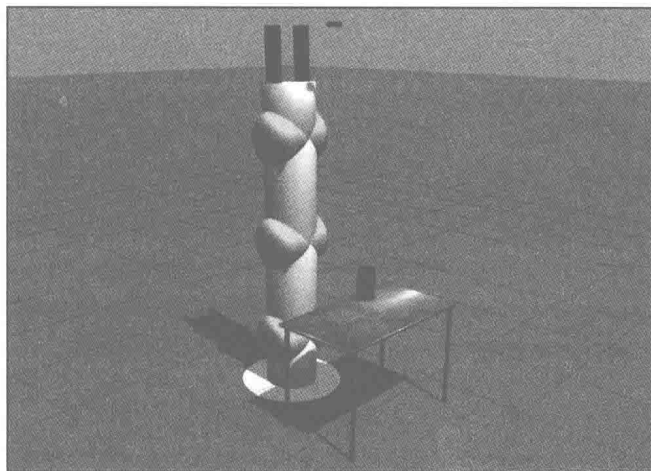


图 18 在 Gazebo 中的机械臂和对象环境

这个场景包括机器人手臂的夹持器与 RGB-D 传感器。然后还有一张桌子和一个可乐罐，分别是平坦的支撑面和圆柱形物体。你可以用下面的命令在 Gazebo 中启动这个场景：

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch
```

这个场景只是实际使用情况的一个简单示例模型。然而，我们仍然要告诉 MoveIt! 关于规划场景的信息。这时，它只知道关于机械臂的信息。我们要告诉它桌子和可乐罐。这可以通过两种方式实现，一是通过使用三维感知算法，它通过 RGB-D 传感器获取点云数据，或者以编程方式，通过一些基本的基元设定位姿和对象的形状。我们将看到如何用下面的方法定义规划场景。

执行抓取和放置任务的代码是 `pick_and_place.py`，这个 Python 程序位于 `rosbook_arm_pick_and_place` 功能包的脚本文件夹中。在 `CokeCanPickAndPlace` 类的 `__init__` 方法是创建规划场景的重要组成部分：

```
self._scene = PlanningSceneInterface()
```

在下面的小节中，我们将在这个规划场景中添加桌子和可乐罐。

1. 要抓取的目标对象

在这种情况下，要抓取的目标对象是可乐罐。它是一个圆柱形的物体，我们可以近似为一个方盒，它是 MoveIt! 规划场景应用程序接口的基本元素之一：

```
# Retrieve params:
self._grasp_object_name = rospy.get_param('~grasp_object_name', 'coke_
can')

# Clean the scene:
self._scene.remove_world_object(self._grasp_object_name)

# Add table and Coke can objects to the planning scene:
self._pose_coke_can = self._add_coke_can(self._grasp_object_name)
```

规划场景中的物体收到一个唯一的字符串标识符。这里，`coke_can` 是可乐罐的标识符。我们从场景来删除它以避免重复，然后添加到场景中。`_add_coke_can` 方法定义了可乐罐的位姿和形状的尺寸：

```
def _add_coke_can(self, name):
    p = PoseStamped()
    p.header.frame_id = self._robot.get_planning_frame()
    p.header.stamp = rospy.Time.now()

    p.pose.position.x = 0.75 - 0.01
    p.pose.position.y = 0.25 - 0.01
    p.pose.position.z = 1.00 + (0.3 + 0.03) / 2.0

    q = quaternion_from_euler(0.0, 0.0, 0.0)
    p.pose.orientation = Quaternion(*q)

    self._scene.add_box(name, p, (0.15, 0.15, 0.3))

    return p.pose
```

这里重要的部分是 `add_box` 方法将对象 `box` 添加到我们先前创建的规划场景中。设定方盒的名称、位姿和尺寸，这里，还需要将其与 Gazebo 场景中的桌子和可乐罐匹配。我们还需要设置规划框架的 `frame_id` 和时间采样 `now`。为了使用规划框架，我们需要 `RobotCommander`，它是 `MoveIt!` 命令机械臂执行程序接口：

```
self._robot = RobotCommander()
```

2. 支撑面

用与创建一个方盒相同的方式，创建一个桌子对象。我们直接删除之前所有的对象，并添加桌子。这里，对象名称是桌子 (`table`):

```
# Retrieve params:
self._table_object_name = rospy.get_param('~table_object_name',
'table')

# Clean the scene:
self._scene.remove_world_object(self._table_object_name)

# Add table and Coke can objects to the planning scene:
self._pose_table = self._add_table(self._table_object_name)
```

`_add_table` 方法添加桌子到规划场景中:

```
def _add_table(self, name):
    p = PoseStamped()
    p.header.frame_id = self._robot.get_planning_frame()
    p.header.stamp = rospy.Time.now()

    p.pose.position.x = 1.0
    p.pose.position.y = 0.0
    p.pose.position.z = 1.0

    q = quaternion_from_euler(0.0, 0.0, numpy.deg2rad(90.0))
    p.pose.orientation = Quaternion(*q)

    self._scene.add_box(name, p, (1.5, 0.8, 0.03))

    return p.pose
```

我们运行以下命令可以在 rviz 中可视化规划场景物体:

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch
$ roslaunch rosbook_arm_moveit_config moveit_RViz.launch config:=true
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch
$ rosrun rosbook_arm_pick_and_place pick_and_place.py
```

这实际是运行整个抓取和放置任务,我们将在之后继续解释。当开始 `pick_and_place.py` 程序之后,你将看到方盒、建模后的桌子和以绿色显示的可乐罐,完美匹配 RGB-D 传感器感知的点云,如图 19 所示。

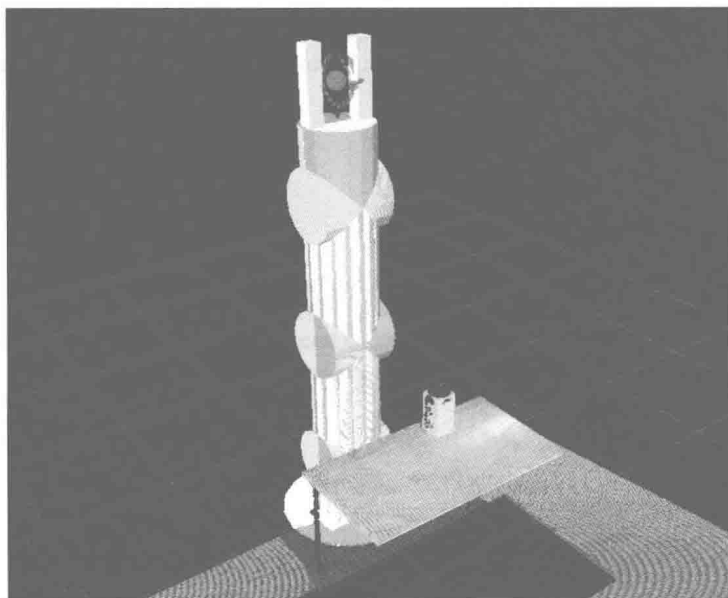


图 19 由 RGB-D 传感器感知的点云环境

10.5.2 感知

可以通过感知支撑面来代替手动将对象添加到规划场景。在这种情况下，桌子可以作为点云检测到的一个水平面。一旦桌子被识别，可以通过原始的点云获取目标对象，可以将其近似为圆柱体或方盒。我们将使用与之前相同的方法将方盒添加到规划场景。但在这种情况下，对象的位姿和尺寸（以及分类）将来自 3D 感知和分割算法的输出。

这种由 RGB-D 传感器提供的点云感知和分割可以使用相关概念和算法轻松完成。然而在某些情况下，可能出现精度不足而导致无法正确抓取对象。感知可以通过在对象上放置基准标记来帮助完成抓握，例如 ArUco (<http://www.uco.es/investiga/grupos/ava/node/26>)，它有 ROS 封装，可以查看网页 https://github.com/pal-robotics/aruco_ros。

在这里我们手动设置规划场景，感知部分留给你去探索。正如我们看到的要抓取由代码手动定义的目标对象和支撑面，通过与 rviz 的点云数据进行比较对应，直到匹配效果很好。

10.5.3 抓取

现在，我们已经定义了场景中的目标对象，需要生成抓握位姿来把它拿起来。要达到这个目的，我们使用 `moveit_simple_grasps` 功能包中的抓握生成服务，源码在 https://github.com/davetcoleman/moveit_simple_grasps 下。

对 ROS hydro 而言，虽然在 Ubuntu 中有可用的 Debian 功能包，但它不能正确地支持任何机器人。出于这个原因，我们需要补丁版本，直到以下 pull 请求被接受。可以从 https://github.com/davetcoleman/moveit_simple_grasps/pull/16 下载补丁的版本。

因此，我们需要运行以下命令来将修补的分支添加到我们的工作空间（在工作空间的 `src` 文件夹）：

```
$ wstool set moveit_simple_grasps --git git@github.com:efernandez/moveit_
simple_grasps.git -v server_params
$ wstool up moveit_simple_grasps
```

使用下列命令编译这个文件：

```
$ cd ..
$ cactin_make
```

现在我们可以运行抓握生成服务，如下所示（别忘了源 `devel/setup.bash`）：

```
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch
```

在本例中，抓握生成服务需要抓握数据配置如下：

```
base_link: base_link

gripper:
end_effector_name: gripper

# Default grasp params
joints: [finger_1_joint, finger_2_joint]

pregrasp_posture:          [0.0, 0.0]
```

```

pregrasp_time_from_start: &time_from_start 4.0

grasp_posture:            [1.0, 1.0]
grasp_time_from_start:    *time_from_start

postplace_time_from_start: *time_from_start

# Desired pose from end effector to grasp [x, y, z] + [R, P, Y]
grasp_pose_to_eef:        [0.0, 0.0, 0.0]
grasp_pose_to_eef_rotation: [0.0, 0.0, 0.0]

end_effector_parent_link: tool_link

```

这基本上定义了我们将要使用夹持器抓取的对象以及抓取前、后的位姿。

现在我们需要一个操作客户端来查询抓握位姿。这在 `pick_and_place.py` 程序内完成，在尝试抓握目标对象之前进行。使用以下代码创建操作客户端：

```

# Create grasp generator 'generate' action client:
self._grasps_ac = SimpleActionClient('/moveit_simple_grasps_server/
generate', GenerateGraspsAction)
if not self._grasps_ac.wait_for_server(rospy.Duration(5.0)):
    rospy.logerr('Grasp generator action client not available!')
    rospy.signal_shutdown('Grasp generator action client not
available!')
    return

```

在 `_pickup` 方法中，我们使用下列代码获取夹持器的位姿：

```
grasps = self._generate_grasps(self._pose_coke_can, width)
```

在这里，宽度参数设定为要抓取对象的宽度。`_generate_grasps` 方法执行以下操作：

```

def _generate_grasps(self, pose, width):
    # Create goal:
    goal = GenerateGraspsGoal()

    goal.pose = pose
    goal.width = width

    # Send goal and wait for result:
    state = self._grasps_ac.send_goal_and_wait(goal)
    if state != GoalStatus.SUCCEEDED:
        rospy.logerr('Grasp goal failed!: %s' % self._grasps_
ac.get_goal_status_text())
        return None

    grasps = self._grasps_ac.get_result().grasps

    # Publish grasps (for debugging/visualization purposes):
    self._publish_grasps(grasps)

    return grasps

```

总结一下，它将发送一个 `actionlib` 目标去获取一组用于定位目标位姿（通常在对象质心）的一套抓握位姿。本书中提供的代码，对一些选项进行了说明，但它们只能查询特定类型的抓握，如某些角度或指向向上或向下。函数的输出是所有抓取位姿，然后将尝试进行抓取动作。多种抓握位姿可以增加一次成功抓取的可能性。

出于可视化和调试的目的，抓握生成服务所提供的抓握位姿也是 `PoseArray` 使用 `_publish_grasps` 方法发布。我们可以像以前一样看到它们在 `rviz` 中运行整个抓取和放置任务的过程：

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch
$ roslaunch rosbook_arm_moveit_config moveit_RViz.launch config:=true
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch
$ rosrn rosbook_arm_pick_and_place pick_and_place.py
```

运行 `pick_and_place.py` 程序几秒钟后，我们将看到在目标对象上有多个箭头，对应于尝试把它拿起来的抓取位姿。如图 20 所示。

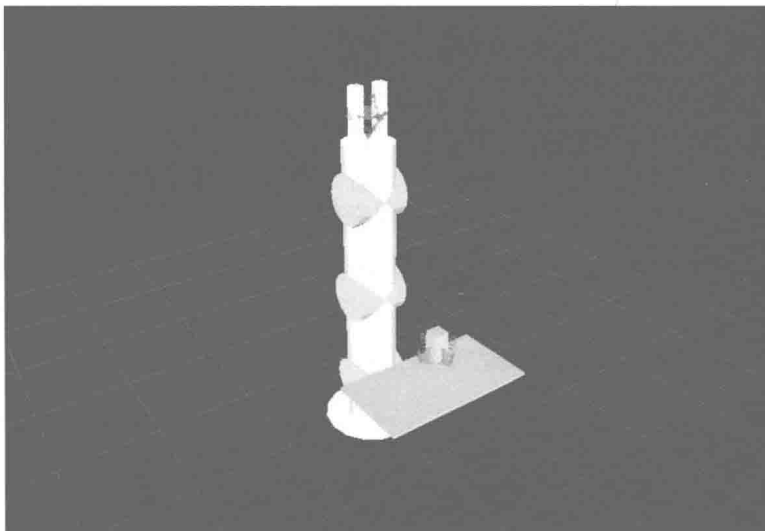


图 20 抓取位姿的可视化

10.5.4 抓取操作

一旦有了抓握的位姿，我们可以使用 `MoveIt!` 的 `/pickup` 操作服务发送所有这些目标。和前面一样，我们将创建操作客户端：

```
# Create move group 'pickup' action client:
self._pickup_ac = SimpleActionClient('/pickup', PickupAction)
if not self._pickup_ac.wait_for_server(rospy.Duration(5.0)):
    rospy.logerr('Pick up action client not available!')
    rospy.signal_shutdown('Pick up action client not available!')
    return
```

然后，我们将尝试多次抓取可乐罐直到成功：

```
# Pick Coke can object:
while not self._pickup(self._arm_group, self._grasp_object_name,
self._grasp_object_width):
    rospy.logwarn('Pick up failed! Retrying ...')
    rospy.sleep(1.0)
```

在 `_pickup` 方法中，我们为 MoveIt! 创建了一个抓取目标，然后如之前所述，生成抓握位姿：

```
# Create and send Pickup goal:
goal = self._create_pickup_goal(group, target, grasps)

state = self._pickup_ac.send_goal_and_wait(goal)
if state != GoalStatus.SUCCEEDED:
    rospy.logerr('Pick up goal failed!: %s' % self._pickup_ac.get_
goal_status_text())
    return None

result = self._pickup_ac.get_result()

# Check for error:
err = result.error_code.val
if err != MoveItErrorCodes.SUCCESS:
    rospy.logwarn('Group %s cannot pick up target %s!: %s' % (group,
target, str(moveit_error_dict[err])))

    return False

return True
```

发送目标，并使用状态来检查机械臂是否已经拿起对象。在 `_create_pickup_goal` 方法创建拿起的目标，如下所示：

```
def _create_pickup_goal(self, group, target, grasps):
    # Create goal:
    goal = PickupGoal()

    goal.group_name = group
    goal.target_name = target

    goal.possible_grasps.extend(grasps)

    # Configure goal planning options:
    goal.allowed_planning_time = 5.0

    goal.planning_options.planning_scene_diff.is_diff = True
    goal.planning_options.planning_scene_diff.robot_state.is_diff = True
    goal.planning_options.plan_only = False
    goal.planning_options.replan = True
    goal.planning_options.replan_attempts = 10

    return goal
```

目标需要规划群组（这里为 arm）和目标名称（这里为 coke_can）。然后，设置所有可能的抓握和一些规划选项，其中包括是否允许按需要增加规划时间。

当目标对象成功地拿起时，我们将看到它附加到夹持器，对应的方盒以紫色的颜色显示，如图 21 所示（注意，它可能看起来像放错了位置的影子夹持器，毕竟它只是一个生成的可视化部件）。

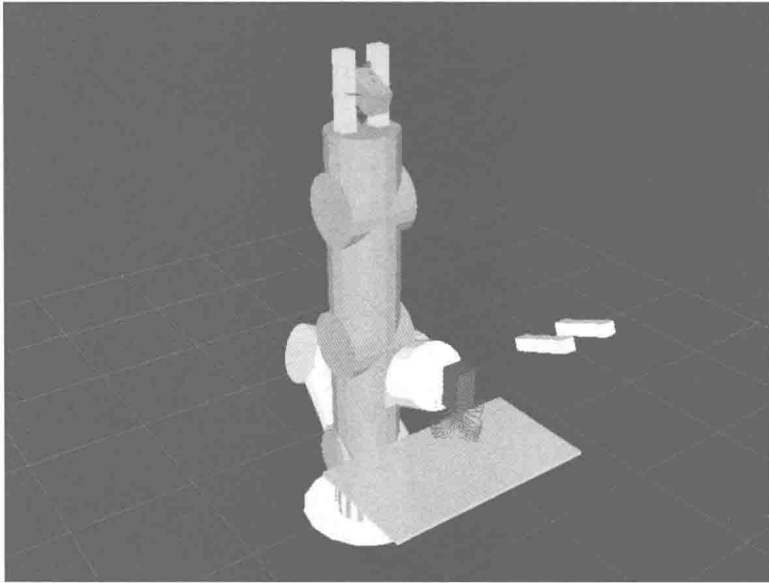


图 21 机械臂抓取一个对象

10.5.5 放置操作

当拿起对象后，机械臂将继续进行放置的操作。MoveIt! 提供 /place 操作服务，所以第一步创建一个操作客户端发送一个期望位置的放置目标，以便放置拿起的对象：

```
# Create move group 'place' action client:
self._place_ac = SimpleActionClient('/place', PlaceAction)
if not self._place_ac.wait_for_server(rospy.Duration(5.0)):
    rospy.logerr('Place action client not available!')
    rospy.signal_shutdown('Place action client not available!')
    return
```

然后，我们将尝试放置对象，直到最终设法做到：

```
# Place Coke can object on another place on the support surface
(table):
while not self._place(self._arm_group, self._grasp_object_name, self._
pose_place):
    rospy.logwarn('Place failed! Retrying ...')
    rospy.sleep(1.0)
```

`_place` 方法使用下面的代码：


```
def _place(self, group, target, place):
    # Obtain possible places:
    places = self._generate_places(place)

    # Create and send Place goal:
    goal = self._create_place_goal(group, target, places)

    state = self._place_ac.send_goal_and_wait(goal)
    if state != GoalStatus.SUCCEEDED:
        rospy.logerr('Place goal failed!: ' % self._place_ac.get_goal_
            status_text())
        return None

    result = self._place_ac.get_result()

    # Check for error:
    err = result.error_code.val
    if err != MoveItErrorCodes.SUCCESS:
        rospy.logwarn('Group %s cannot place target %s!: %s' % (group,
            target, str(moveit_error_dict[err])))

    return False

return True
```

方法生成了多个可能的对象放置位置，创建放置目标并将其发送。然后，它检查结果来验证对象是否已被放置。为了放置对象，我们可以使用一个单一的放置位姿，但一般最好多提供几个选项。这里，我们使用 `_generate_places` 方法，在给定位位置生成不同角度的放置方式。当放置方式生成后，它们将以 `PoseArray` 发布，所以可以看到它们以蓝色箭头显示，如图 22。

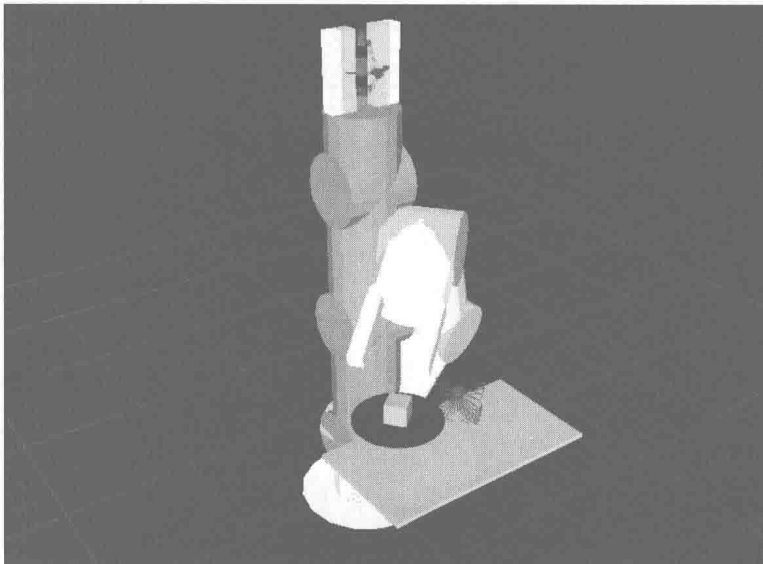


图 22 放置位姿的可视化

一旦获得位置，`_create_place_goal` 方法创建一个放置目标的代码如下：

```
def _create_place_goal(self, group, target, places):
    # Create goal:
    goal = PlaceGoal()

    goal.group_name          = group
    goal.attached_object_name = target

    goal.place_locations.extend(places)

    # Configure goal planning options:
    goal.allowed_planning_time = 5.0

    goal.planning_options.planning_scene_diff.is_diff = True
    goal.planning_options.planning_scene_diff.robot_state.is_diff =
True
    goal.planning_options.plan_only = False
    goal.planning_options.replan = True
    goal.planning_options.replan_attempts = 10

    return goal
```

简单地说，放置目标有群组（这里为 `arm`）和目标对象（这里为 `coke_can`），对应夹持器和位置有多个放置方式（位姿）。此外，还提供了几个规划选项，包括允许的规划时间，可以按需要增加。当放置对象时，我们将在桌子上看到方盒再次以绿色显示，接着机械臂会抬起来，如图 23 所示。

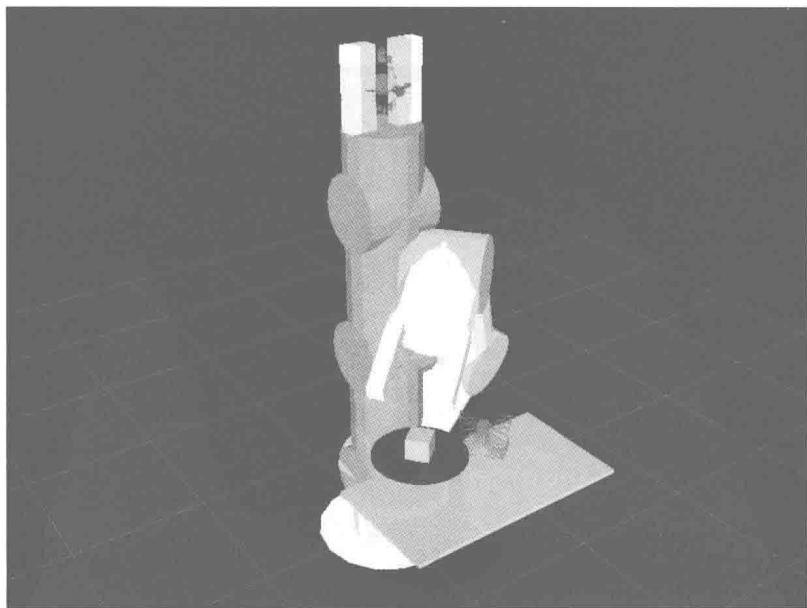


图 23 放置完对象后的机械臂

10.5.6 演示模式

我们可以在没有感知的演示模式下完成整个抓取和放置任务，也就是没有实际运行的 Gazebo 仿真或真实机器人手臂。在此模式下，我们将使用虚拟控制器移动手臂，一旦获得规划方案，MoveIt! 将执行抓取和放置的操作，包括抓握对象。可以直接在实际控制器上使用相同的代码。

在演示模式下运行抓取和放置，运行以下命令：

```
$ roslaunch rosbook_arm_moveit_config demo.launch  
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch  
$ rosrn rosbook_arm_pick_and_place pick_and_place.py
```

特别是第一个启动文件，它直接打开 rviz，装载虚拟控制器而不是 Gazebo 中的机器人手臂。运行上述命令后，机械臂移动做抓取和放置任务的一些截图如图 24 所示。

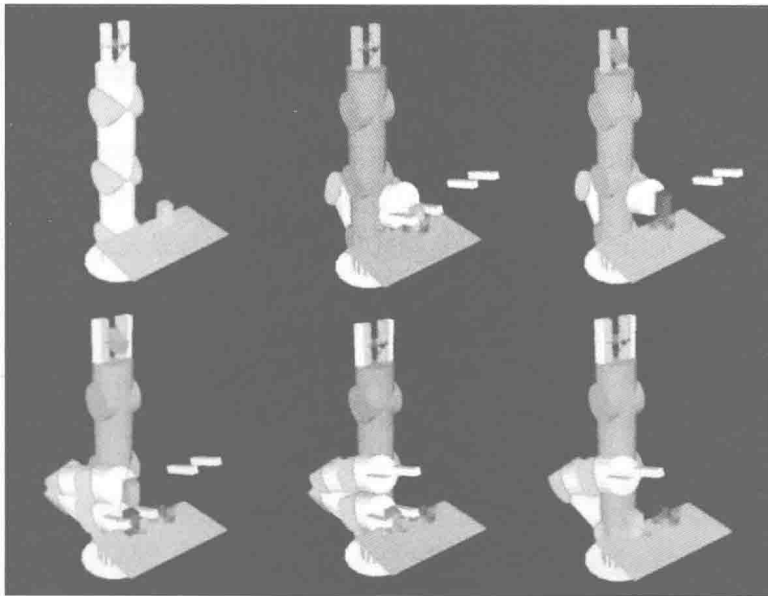


图 24 在演示模式下手臂做抓取和放置的任务

10.5.7 在 Gazebo 中仿真

在演示模式下使用相同的代码，不管是在仿真（Gazebo）还是实际硬件平台上，我们实际上都可以移动真正的控制器。这个接口是相同的，所以使用一个真正的手臂或在 gazebo 中仿真是完全相同的。这里，关节会真实地移动和抓取，会让夹持器接触并抓住物体（可乐罐）。这就要求在 Gazebo 中合理地定义对象和夹持器，使其正常工作。

在这种情况下要运行前面所示的抓取和放置任务的命令是：

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch
```

```
$ roslaunch rosbook_arm_moveit_config moveit_RViz.launch config:=true
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch
$ rosrun rosbook_arm_pick_and_place pick_and_place.py
```

启动的文件只有第一部分相对于演示模式进行了修改，用它替换演示模式下的 demo.launch。这里，在 Gazebo 中机械臂的环境包含桌子、可乐罐以及 RGB-D 摄像头等。然后，moveit_rviz.launch 文件打开 rviz 和 MoveIt! 插件并提供与 demo.launch 相同的接口。此时，当 pick_and_place.py 程序运行时，机械臂在 Gazebo 中移动。

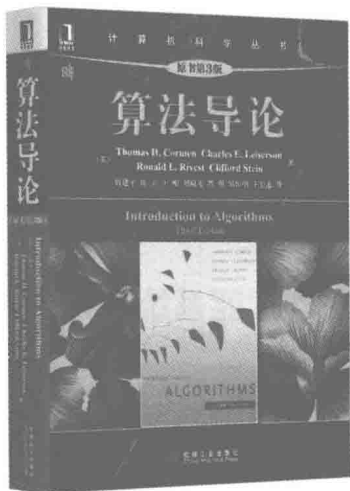
10.6 本章小结

本章涵盖了将一个机械臂集成到 MoveIt! 和 Gazebo 中所涉及的大部分内容，讲述了如何在真实仿真环境中使用机械臂。MoveIt! 为我们提供了一类使用规划求解逆运动学模型 (IK) 进行机械臂运动规划的非常简单和简洁的工具，以及便于使用的丰富文档。但鉴于该体系结构的复杂性，只有当正确理解了 MoveIt!、机器人的传感器和执行器之间所有不同的相互作用时，才能正常工作。

我们已经了解了 MoveIt! 应用程序接口中不同的高级别对象，详细介绍需要涵盖一整本书的内容。为了避免在理解一个 API 如何完整地执行非常简单的操作时浪费时间，本书中采取的办法是一直把内容局限于非常简单的运动规划以及在规划场景人工创建对象并和 RGB-D 传感器中生成点云进行交互。

最后，对如何执行目标对象的抓取和放置任务提供了一个非常详细的讲解。虽然这不是机械臂的唯一目的，但可能是你喜欢的一个尝试，并且是非常常见的工业机器人功能。使用 MoveIt! 运动规划和 3D 感知，可以在复杂动态环境下实现这样的任务。对于特定用途的机械臂，需要对 MoveIt! 的体系结构和 API 进一步学习和深化，这样可以使你更加深入地理解 MoveIt! 的功能。

推荐阅读



算法导论（原书第3版）

作者：Thomas H. Cormen 等著

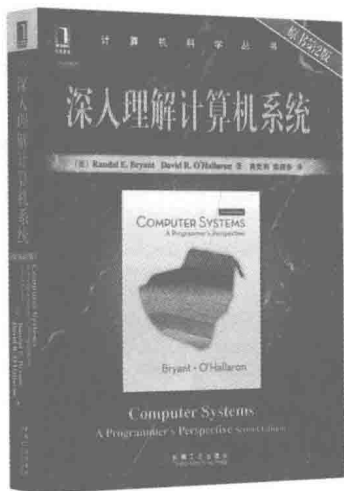
ISBN: 978-7-111-40701-0 定价：128.00元



C程序设计语言（第2版·新版）

作者：Brian W. Kernighan 等著

ISBN: 978-7-111-12806-0 定价：30.00元



深入理解计算机系统（原书第2版）

作者：Randal E. Bryant 等著

ISBN: 978-7-111-32133-0 定价：99.00元

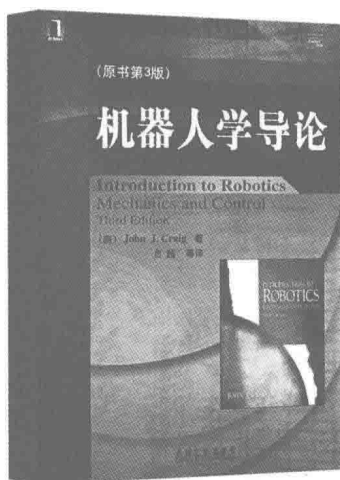


计算机组成与设计：硬件/软件接口（原书第5版）

作者：David Patterson, John Hennessy

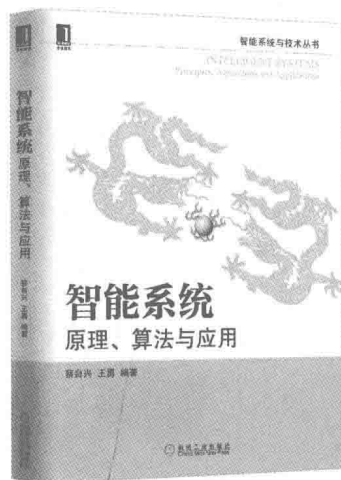
ISBN: 978-7-111-50482-5 定价：99.00元

推荐阅读



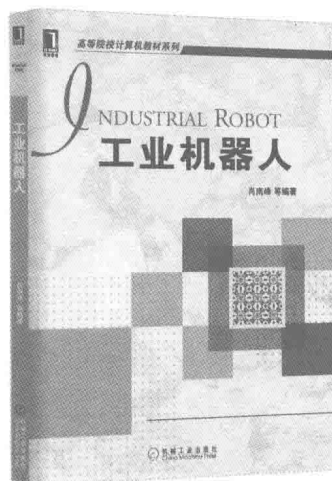
机器人学导论（原书第3版）

作者：John J. Craig ISBN: 978-7-111-18681-8 定价：42.00元



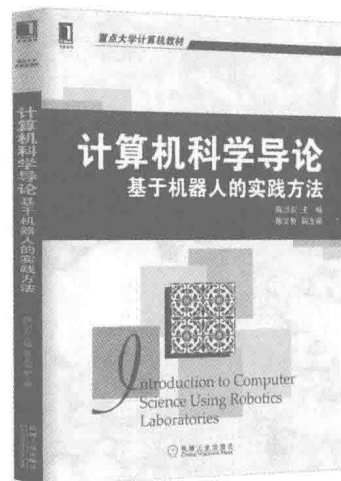
智能系统：原理、算法与应用

作者：蔡自兴等 ISBN: 978-7-111-47200-1 定价：59.00元



工业机器人

作者：肖南峰等 ISBN: 978-7-111-35333-1 定价：30.00元



计算机科学导论：基于机器人的实践方法

作者：陈以农等 ISBN: 978-7-111-43588-4 定价：35.00元

机器人操作系统一站式

本书从入门级内容开始介绍，包括传感器集成、建模、仿真、计算机视觉、导航算法等。然后，探索主题、消息和节点等概念。接下来，介绍机器人如何使用高清摄像头观测或使用距离传感器导航避障。

第2版新增了ROS Hydro，不仅介绍如何创建、可视化和处理来自不同传感器的点云信息，还展示如何使用MoveIt! 控制和规划包含多个关节的机械臂的运动。

为了充分利用这本书，读者应有C++编程基础、GNU/Linux系统知识和计算机科学的一般技能。不需要ROS经验背景，这本书可以带你从零开始。

通过阅读本书，你将学到

- 安装一个完整的ROS Hydro系统
- ROS功能包和综合功能包的创建及其实时使用和调试
- ROS节点的编译、处理和调试
- 设计三维机器人模型并使用Gazebo在虚拟环境中进行仿真
- 在机器人上生成和调整导航功能包集
- 将不同的传感器如激光测距仪、Arduino和Kinect集成到机器人上
- 可视化和处理来自不同传感器的点云信息
- 使用MoveIt!控制和规划包含多个关节的机械臂的运动

[PACKT]
PUBLISHING

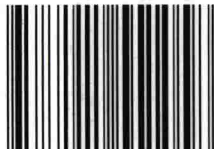
投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

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: 自动化\嵌入式

ISBN 978-7-111-55105-8



9 787111 551058 >

定价: 69.00元

[General Information]

书名=ROS机器人程序设计 原书第2版

作者=恩里克·费尔南德斯

页数=296

SS号=14136979

DX号=

出版日期=2016.11

出版社=北京机械工业出版社