HZ BOOKS
华章科技

[PACKT]
PUBLISHING

对于非专业人员，你将从本书了解如何安装ROS如何开始使用ROS的基本工具和框架中不同的功能。
对于专业人员，你将从本书学会如何使用硬件，如何将你的算法应用到现实环境中，从而创建一个满足你所有需求的功能齐全的机器人

机器人设计与制作系列

ROS机器人高效编程

（原书第3版）

Effective Robotics Programming with ROS,Third Edition

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

张瑞雷　刘锦涛　译

机械工业出版社
China Machine Press

机器人设计与制作系列

ROS机器人高效编程（原书第3版）

Effective Robotics Programming with ROS，Third Edition

（西）阿尼尔·马哈塔尼（Anil Mahtani） 等著

张瑞雷 刘锦涛 译

ISBN：978-7-111-57846-8

本书纸版由机械工业出版社于2017年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

推荐序一

2006年，在好奇心的驱使下，一帮人走在一起，组建了一个机器人研究实验室Willow Garage。他们利用开源软件吸引他人，使人们加入一个创造个人机器人的宏伟计划中。机器人操作系统（ROS）正是这一宏伟计划的一部分。

ROS打开了一个潘多拉魔盒，可是很多人还没有做好准备，还完全没有意识到是怎么回事，就不得不与ROS牵连在一起，卷入到一个洪流中。刘锦涛博士和张瑞雷博士将本书翻译成中文，帮助大家突破语言障碍，从而在洪流中更好地奋勇前进。

刘锦涛博士和张瑞雷博士都是易科（Exbot）机器人实验室的成员，是机器人技术普及的积极推动者。从2010开始，易科机器人实验室利用互联网、社交网络、博客，积极开展机器人技术和ROS的在线教育、互动问答，使上万人受益。

2013年，当我回到国内开始我的职业生涯时，第一件事就是寻找机器人相关的研究者和爱好者，也是那时，通过易科机器人实验室建立的QQ群，结识了后来一帮志同道合的老师、学生和朋友。

2015年，我们实验室组织了全国第一届机器人操作系统暑期学校，希望有更多的人通过线下的互动促进交流，激发合作的热情，碰撞出创业的激情。我们的活动信息也通过易科机器人实验室建立的社交网络传递到全国的各个角落。

在本书再版之际，我想，对所有热爱科技，热爱机器人技术，热爱这块我们所赖以生存的土地的人来说，无论未来多么不可预测，只要像刘锦涛博士和张瑞雷博士这样，不懈地努力，大家团结在一起，都会成为洪流中的勇士。

张新宇博士

华东师范大学智能机器人运动与视觉实验室负责人

机器人操作系统（ROS）暑期学校创办人

推荐序二

记得第一次接触ROS的时候我还在学校做研究，是当时与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。得知其特色及相关工具后，我非常兴奋，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刚处在起步阶段，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合适作为机器人敏捷开发工具，算法及控制等代码都能很容易复用，减少了很多重复性的工作。但奈何当时的功能包不多，而且系统对运算资源要求高，最终也没有在当时的机器人项目中使用上。

由于其开源性以及对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，甚至基于ROS进行开发。相信这个趋势会一直持续下去并且蔓延到全球各地。而我亦深深体会到国内对ROS的关注也在近年有显著的上升。

几年前，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上学习ROS的相关知识，完全没有中文数据可以查看。幸好在国内也有不少有心人积极推动国内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文章，丰富ROS的中文资源，使学习ROS变得更方便。

我与本书译者通过共同举办ROS国内培训课程而结缘。过去一年我们一起推动的国内线下ROS实战培训课程星火计划已遍布全国，渐见成效。他在推动ROS在国内发展方面也有着举足轻重的地位，运营着国内著名的ROS交流社区——易科机器人实验室（exbot.net）。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，是一本ROS入门必看书。希望本书能帮助你快速进入ROS的世界，探索ROS的精彩。

林天麟博士

NXROBO创始人&CEO

译者序

机器人的时代已经到来！机器人正在变得越来越灵活、智能。机器人已经从传统的工业应用开始加速进入千家万户，正从方方面面改变着人们的工作和生活，例如，扫地机器人能在清扫过程中自主绘制室内地图并智能规划路径，京东的包裹小车已经开始在校园中穿梭并投身到快递服务的行业中，这样的智能机器人已经越来越多。

那么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，这样就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，这套系统对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）用于进行技术分享与交流，我们的队伍也在不断发展壮大。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版补充了点云和MoveIt!方面的内容。第3版则对ROS版本进行了升级，采用目前ROS最新长期支持的版本Kinetic进行介绍；并针对ROS的最新进展，继续完善，增加了Docker和设计开发真实机器人的示例；同时对章节结构进行了调整。第

3版涵盖了使用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，还应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的开源精神和日益完善的合作开发框架，很多问题会逐步迎刃而解。唯一迫切需要的就是，期待你加入到机器人的设计、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3版与第2版的重叠部分主要沿用了第2版中的翻译，个别词汇根据习惯进行了修改。具体来说，张瑞雷对书中内容进行梳理补充，刘锦涛对全书进行了修改润色和统稿整理。

我们将会在books.exbot.net发布本书的其他相关资源。

前言

本书第3版全面地介绍了ROS和各种工具。ROS是一个先进的机器人操作系统框架，目前已有数百个研究团体和公司将其应用在机器人行业中。更重要的是，对于机器人技术的非专业人士和学生来说，它也相对容易上手。在本书中，你将了解如何安装ROS，如何使用ROS的基本工具和框架中不同的功能。

在阅读本书的过程中无须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购买得起的设备，同时涵盖了在机器人研究中最典型的传感器和执行器。

最后，展示ROS具有使整个机器人在实际或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通过Gazebo仿真环境集成它。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。本书将带你从不同方面探索如何创建机器人，例如使用计算机视觉或点云分析传感器感知世界，使用强大的导航功能包集在环境中实现导航，甚至能够用MoveIt!包控制机械臂与周围环境交互。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，我们衷心希望你能全面了解ROS在开发机器人系统时所提供的无限可能性。

主要内容

第1章介绍安装ROS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 Kinetic。这一章还会说明如何从Debian包安装或从源代码进行编译安装，以及在虚拟机、Docker和ARM CPU中安装。

第2章讨论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rosbag和rqt_bag记录并回放消息。

第4章介绍在ROS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等传感器，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5章是关于ROS导航功能包集中的其中一章。该章介绍如何为方便机器人使用导航功能包集进行初始化配置。然后用几个例子对导航功能包集进行说明。

第6章延续第5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7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。

第8章介绍ROS与现实世界如何连接。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9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，就可以使用ROS的标定工具标定摄像头。该章会详细介绍和说明什么是图像管道，讨论如何使用集成了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10章将展示如何在ROS节点中使用点云库（Point Cloud Library，PCL）。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。

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 16.04 LTS。这样你可以根据第1章的内容安装ROS Kinetic。

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，最好有足够的外围接口，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包），以便从本书提供的源代码中复制软件库。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如何帮助开发人员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到一些新东西。当然，只有前3章是纯粹为初学者准备的，所以那些已经使用过ROS的人可以跳过这三章直接阅读后面的章节。

源代码和彩色图片下载

本书源代码可以从华章官网www.hzbook.com下载。

作者简介

Anil Mahtani是一名主要从事水下机器人工作研发的计算机科学家。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，这个大学生团队设计开发了一个自主水下航行器并参加了2012年的欧洲学生自主水下航行器设计挑战赛（Student Autonomous Underwater Challenge-Europe，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2015年，他加入SecureWorks公司，任职软件工程师，在那里他应用相关知识和技术开发入侵检测和预防系统。

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、数据结构、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和电子学感兴趣，并且开发了一些Arduino库回馈社区。

首先，我要感谢家人和朋友的支持，他们总是在我最需要的时候帮助我。我还要感谢我的女友Alex的耐心支持，她是我灵感的源泉。最后，我要感谢我的同事Ihor Bilyy和Dan Good，在我软件工程师职业生涯中他们以专业的方式教会我很多知识。

Luis Sánchez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开始开发声学和计算机视觉系统，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，是一个跨多学科的工程师。他对工作负责，自制力强，并承担一个团队领导者的责任，这在euRathlon比赛中充分展现了出来。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出版社出版的《Learning ROS for Robotics Programming》的相关工作以及第2版的撰写工作。

首先，我要感谢Aaron、Anil以及Enrique邀请我参与编写这本书。同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，他们满腔热忱地准备和引导教学机器人及技术项目，与他们共同工作的时光也非常开心。

最后，我要感谢家人和女友对我参与的每个项目的大力支持和鼓励。我以此书献给他们。

Enrique Fernández具有计算机工程博士学位和机器人学研究背景。他的博士论文解决了自主水下滑翔器（AUG）的路径规划问题，他还研究了SLAM、感知、视觉、控制等机器人学课题。在读博士期间，他加盟了赫罗纳大学的CIRS/ViCOROB水下机器人研究中心，在那里他为AUV开发了视觉SLAM和INS模块。他在2012年参加了SAUC-E并获奖，在2013年作为合作者参与了SAUC-E。

攻读博士学位期间，Enrique在机器人顶级会议和期刊上发表了多篇论文，其中包括国际机器人和自动化会议（International Conference of Robotics and Automation，ICRA）。他也合作编写了一些ROS书籍和章节。

之后，Enrique作为SLAM工程师在2013年6月加盟PAL Robotics公司。在那里，他开发了用于REEM、REEM-C仿人型机器人的ROS软件，也继续为开源社区（主要是ROS控制软件库）做贡献，目前仍是其中一名维护人员。在2015年，他加盟Clearpath Robotics公司的自主系统部门，从事感知算法开发相关工作。他曾经在通用电气公司（General Electric）和约翰迪尔（John Deere）等多家大型工业公司的设施中负责部署工业移动机器人OTTO 1500和OTTO 100软件的运行。

我要感谢本书的合著者，感谢他们为完成本书所付出的努力以及提供了无数示例的代码。我还要感谢拉斯帕尔马斯大学研究组和水下机器人研究中心（Center of Underwater Robotics Research，CIRS/ViCOROB）的研究小组成员。我也要感谢在PAL Robotics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。此外，我还要感谢在Clearpath Robotics的同事们，在这里我掌握了ROS并参与了为工业4.0销售的24/7全天候运行自动驾驶机器人的软件开发。最后，我要感谢我的家人和朋友的帮助与支持，特别是Eva。

Aaron Martinez是数字化制造领域的计算机工程师、企业家和专家。他于2010年在拉斯帕尔马斯大学的IUCTC（Instituto Universitario de Cienciasy Tecnologias Ciberneticas）完成硕士论文。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。获得学位后，他参加了在奥地利林茨约翰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。

我要感谢我的女友，她在我写这本书时支持我并且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、机械以及工程组织，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第一个机器人的机会。同他们一起工作，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。

审校者简介

Lentin Joseph是印度Qbotics Labs（http://www.qboticslabs.com）的创始人兼首席执行官、作家、企业家、电子工程师、机器人爱好者、机器视觉专家、嵌入式程序员。

他在印度喀拉拉的联邦理工学院（FISAT）获电子学和通信工程学士学位。在工程项目的最后一年，他制作了一个可以与人交互的社交机器人（http://www.technolabsz.com/2012/07/social-robot-my-final-year.html）。项目取得了巨大的成功，被视觉和印刷媒体多次报道。该机器人的主要特点是可以与人交流并智能回复，同时具有一定的图像处理能力，如面部、动作和颜色检测。整个项目使用Python编程语言实现。他对机器人、图像处理和Python的兴趣从此开始。

毕业后，他在一家专门从事机器人和图像处理的创业公司工作了3年。同时，他学习了主流的机器人软件平台，如机器人操作系统（ROS）、V-REP、Actin（机器人仿真工具），以及图像处理库，如OpenCV、OpenNI和PCL。他还了解Arduino和Tiva Launchpad上的机器人三维设计和嵌入式编程。

在积累3年的工作经验后，他创立了一家名叫Qbotics Labs的新公司，主要从事研究工作，在机器人和机器视觉等领域开发一些优秀的产品。他负责维护个人网站（http://www.lentinjoseph.com）和一个名为technolabsz的技术博客（http://www.technolabsz.com）。他在科技博客上发布作品。他也是印度PyCon2013的演讲者，主题是“使用Python的学习机器人”（Learning Robotics using Python）。

Lentin是《Learning Robotics using Python》（更多内容参考http://learn-robotics.com）和《Mastering ROS for Robotics Programming》（更多内容参考http://mastering-ros.com）的作者，这两本书都由Packt出版社出版。第一本书的主题是使用ROS和OpenCV构建自主移动机器人。这本书是在ICRA 2015上推出的，并在ROS博客Robohub、OpenCV、Python网站以及其他相关论坛上推广。第二本书是掌握机器人操作系统（ROS）的工具书，也在ICRA 2016上推出，它是最畅销的ROS书籍之一。

作为ICRA 2016的一部分，Lentin及其团队获得了HRATC 2016挑战

赛的冠军，同时他也是ICRA 2015挑战赛HRATC决赛的入围者（http://www.icra2016.org/conference/challenges/）。

第1章　ROS入门

欢迎开始阅读本书第1章。本章将介绍如何安装ROS，它是一种新的标准机器人系统软件框架。本书是基于ROS　Hydro/Indigo的《ROS机器人程序设计》（原书第2版）[1]一书的升级版。通过ROS，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。得益于开源精神，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正在不断进步完善。

本书包含如下内容：

·在特定版本的Ubuntu系统下安装ROS Kinetic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

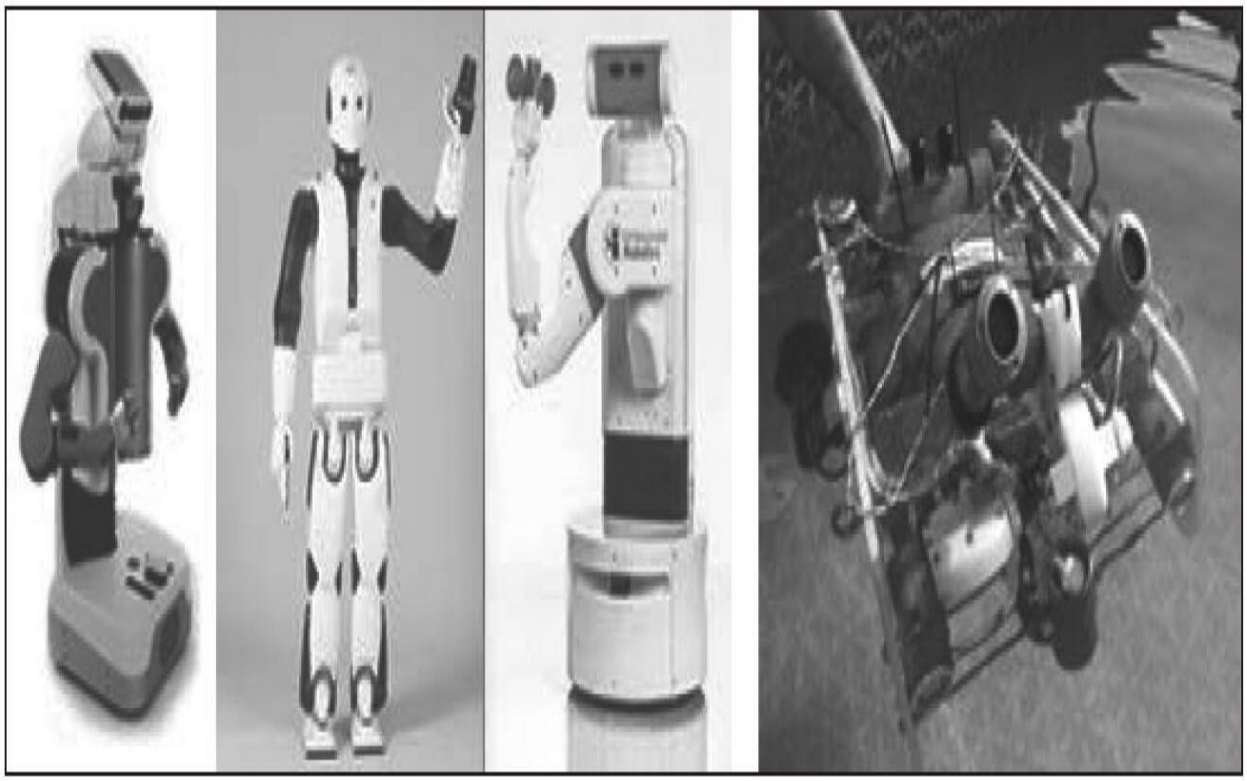
本章主要介绍怎样在Ubuntu系统中安装完整版本的ROS　Kinetic。Ubuntu不但能够全面支持ROS，而且是ROS官方推荐的操作系统。当然，也可以在其他的操作系统中安装ROS。这本书使用的Ubuntu版本是15.10（Wily　Werewolf），可以在http://releases.ubuntu.com/15.10免费下载安装。注意，也可以按照如下示例步骤使用Ubuntu 16.04（Xenial）版本，同时，对于BeagleBone Black的安装我们使用Ubuntu Xenial。

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的基本思想是无须改动就能够在不同的机器人上复用

代码。基于此，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，这避免了重复劳动。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，Willow Garage继续开发ROS，如今开源机器人基金会（Open Source Robotics Foundation，OSRF）开始接管ROS及其相关工程（如Gazebo）的维护工作，也包括新功能的开发。

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中应用。一些完全支持ROS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。最后一个是由拉斯帕尔马斯大学开发的水下机器人，代码尚未公布。可以在http://wiki.ros.org/Robots找到很多这样的例子。



ROS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），然后为ROS提供测量值以开发机器人应用程序。

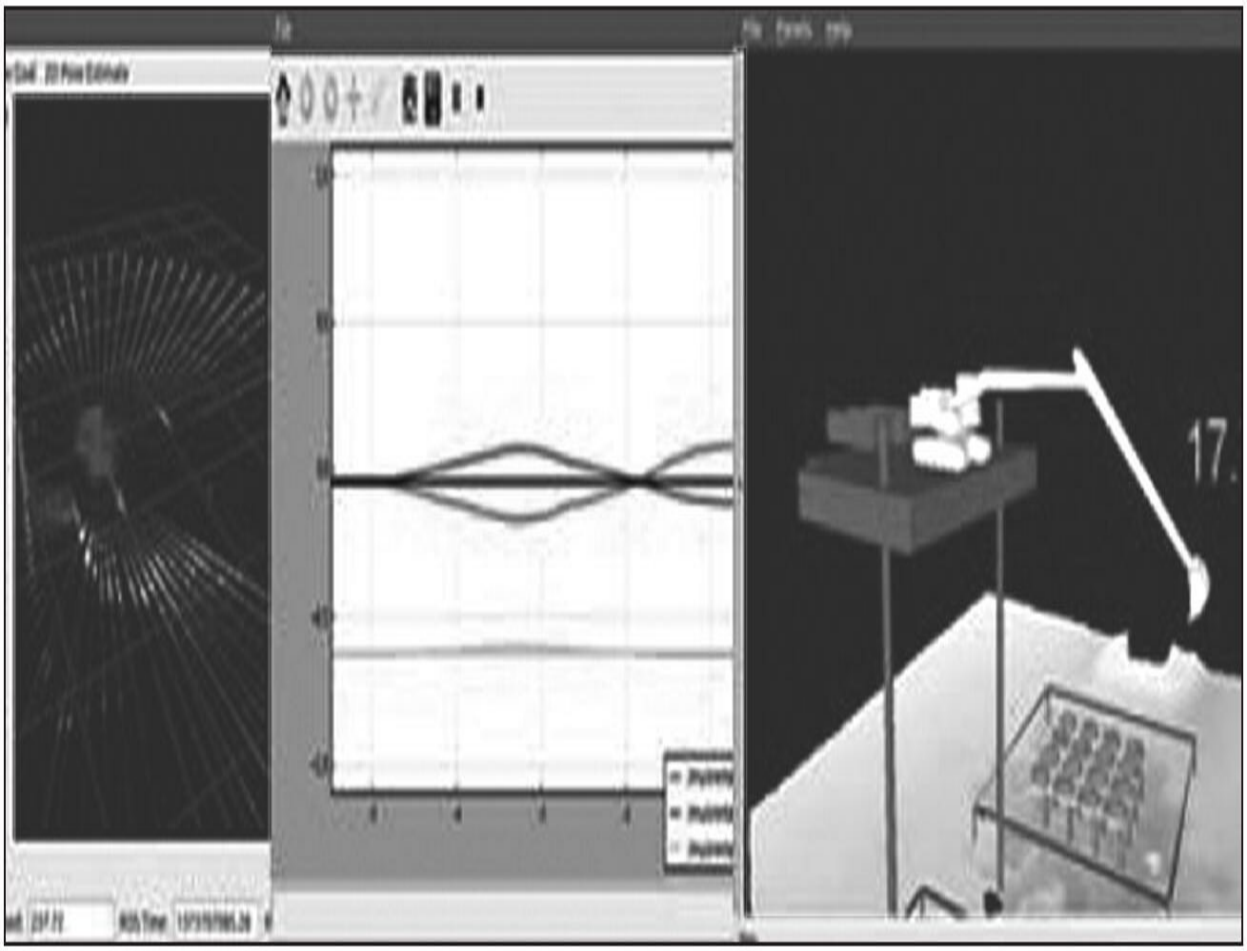
ROS提供了一个标准的操作系统环境，包括硬件抽象、底层设备控制、通用功能的实现、进程间消息转发和使用catkin和cmake管理功能包等。

它基于一个集中式拓扑的图结构，其中处理节点与其他节点之间在通信图网络上接收和发布的信息。节点是任意进程，它从传感器读取数据、控制执行器，或运行用于在环境中自主映射或导航的高级复杂的机器人或视觉算法。

*-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相关联的，如导航库和rviz可视化界面都基于这个软件库。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。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rviz和rqt_plot工具。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；在截图中可以看到真实机器人的三维显示。

ROS是一个基于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。而贡献的*-ros-pkg包则遵循不同的开源协议。

用ROS可以做更多工作。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，在本书中，我们使用的版本是Kinetic，因为这个版本在写作本书时最新。下面会介绍如何安装Kinetic版本的ROS。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以使用其中的ROS，或者下载已安装ROS的虚拟机的方法。

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

[1] 该书由机械工业出版社引进并出版，ISBN：978-7-111-55105-8。——编辑注

1.1 PC安装教程

假设你已经安装了Ubuntu 15.10系统。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、创建的文件夹等。如果需要学习这些工具，可以在网上找到很多相关的资源，也可以参考与这些主题相关的图书。

1.2 使用软件库安装ROS Kinetic

去年，ROS网页更新了设计风格和内容的组织。可以看到如下网页截图：

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。也可以找到博客、新闻和其他功能。

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

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

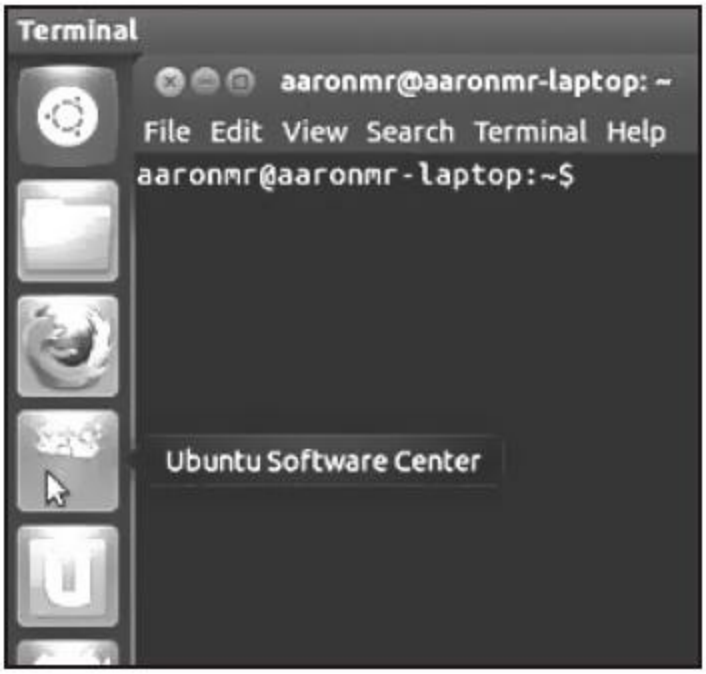
所以这里使用软件库安装ROS，下面将开始在系统中配置Ubuntu软件库。

1.2.1 配置Ubuntu软件库

在本节中，你将学习安装ROS Kinetic的步骤。这个过程基于官方安装页面的内容，链接地址是http://wiki.ros.org/kinetic/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，请单击打开桌面左侧的Ubuntu软件中心（Ubuntu Software Center），如下图所示。



单击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。（选择合适的国家的服务器下载源软件）。

通常情况下，这些选项都是默认选中的，因此这一步骤不会遇到什么问题。

1.2.2 添加软件库到sources.list文件中

在这一步中，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 Kinetic。虽然可以使用任何一个版本，但是推荐使用15.10版本来学习本书的示例。请牢记，Kinetic在Wily Werewolf（15.10）、Xenial Xerus（16.04）下可以正常工作。使用下面的命令添加软件库：

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

 提示：下载示例代码

本书的前言中提到了下载代码包的详细步骤。请参考。

代码包在GitHub上的https://github.com/rosbook/effective_robotics_programming_with_ros处。

在https://github.com/PacktPublishing/也有其他代码包。

一旦添加了正确的软件库，操作系统就知道在哪里下载程序，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 -
```

现在我们能够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，根据用途不同有4种安装方式。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desktop-full（桌面完整安装）。这将安装ROS、rqt工具、rviz可视化环境（3D）、通用机器人库、2D（如stage plan）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、激光或RGBD摄像头（深度摄像头）：

```
$ sudo apt-get install ros-kinetic-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，其中包括ROS、rqt工具、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-kinetic-*功能包集）：

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

·如果你只是想尝试一下，请安装ROS-base。通常建议将ROS-base安装在机器人上，尤其是在机器人没有屏幕和人机界面并且只能TTY远

程登录的情况下。它只安装具有编译和通信库的ROS包，而没有任何的GUI工具。在BeagleBone Black（BBB）中，你将使用如下命令：

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

·最后，无论选择哪一个选项进行安装，都可以安装个别的/特定的ROS功能包集（将PACKAGE替换成给定功能包集的名称）：

```
$ sudo apt-get install ros-kinetic-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。出于同样的原因，ROS中的一些核心组件也需要rosdep，因此rosdep默认安装在ROS中。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，需要执行以下脚本。如果你还安装了另一个ROS发行版，每次需要通过调用脚本来使用它，因为这个脚本会直接配置你的环境。在此我们使用的是ROS Kinetic的脚本，如果你想尝试其他发行版，只需要用indigo或jade代替kinetic即可：

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

如果你在命令行窗口中输入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，需要再次对于setup.bash执行source命令，以配置环境变量，从而使得系统能够找到ROS功能包。否则输入roscore或其他ROS命令，会出现无法工作的情况。这是因为你需要再一次执行脚本来配置环境变量，其中包括ROS的安装路径，以及正确编译新代码的其他功能包和额外路径。

这个问题很容易解决，你只需要在.bashrc脚本文件最后添加脚本，这样，当你开始新命令行窗口时，该脚本将执行并配置环境。

.bashrc文件在用户的home文件夹（/home/USERNAME/.bashrc）下。每次用户打开终端，这个文件会加载命令行窗口或终端的配置。所以可以添加命令或进行配置以方便用户使用。出于这个原因，我们将在.bashrc文件结束时添加脚本，以避免每次打开一个新终端时都要重复输入命令。这用下面命令完成：

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

要使配置生效，必须使用下面的命令执行这个文件，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多个ROS发行版，在这种情况下也许需要切换不同的发行版。因为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。

例如，在.bashrc文件下面可能有这么几行代码：

```
...
source /opt/ros/indigo/setup.bash
source /opt/ros/jade/setup.bash
source /opt/ros/kinetic/setup.bash
...
```

在这种情况下，ROS Kinetic版本将执行。所以必须确保将要运行的版本是文件中的最后一个。建议只导入单个setup.bash。

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，使用它不需要掌握Python。接下来的章节将介绍如何使用这个工具：

运行以下命令在Ubuntu中安装这个工具：

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

这就完成了！你已经在你的系统完成了一个完整的ROS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：roscore和turtlesim。

如果你想做相同的事，在不同命令行窗口中分别输入以下命令：

```
$ roscore

$ rosrun turtlesim turtlesim_node
```

如果一切正常，你将看到下图所示的界面：

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，而无须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，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，能够通过快速重启虚拟机解决，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，最新的版本是4.3.12。可以从http://download.virtualbox.org/virtualbox/4.3.12/下载Linux版本，对于Windows系统，能够从http://download.virtualbox.org/virtualbox/4.3.12/VirtualBox4.3.12-93733-Win.exe下载可用版本。

一旦安装完成，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 Kinetic的Ubuntu镜像文件。可以在OSBOXES通过以下链接下载它：http://www.osboxes.org/ubuntu/。然后可以按照之前的描述直接安装ROS Kinetic。Ubuntu 15.10镜像下载地址为：http://sourceforge.net/projects/osboxes/files/vms/vbox/Ubuntu/15.10/Ubu 64bit.7z/download。

这将下载一个.7z文件。在Linux中，可以使用下面命令解压缩：

```
$ 7z x Ubuntu_15.10-64bit.7z
```

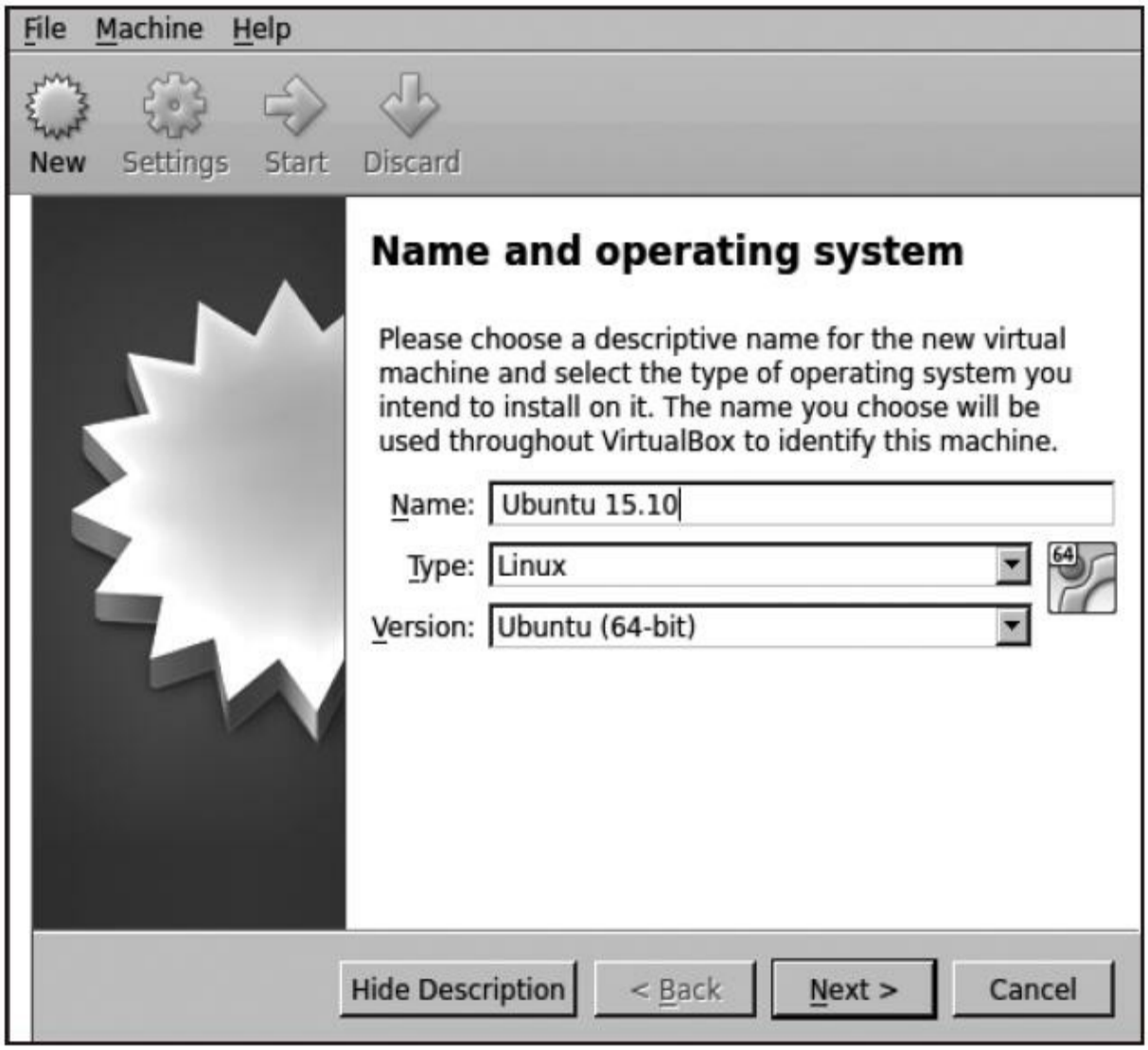
如果.7z命令没有安装，使用下面的命令进行安装：

```
$ sudo apt-get install p7zip-full
```

虚拟机文件在64-bit文件夹中，名为Ubuntu 15.10 Wily(64bit).vdi。

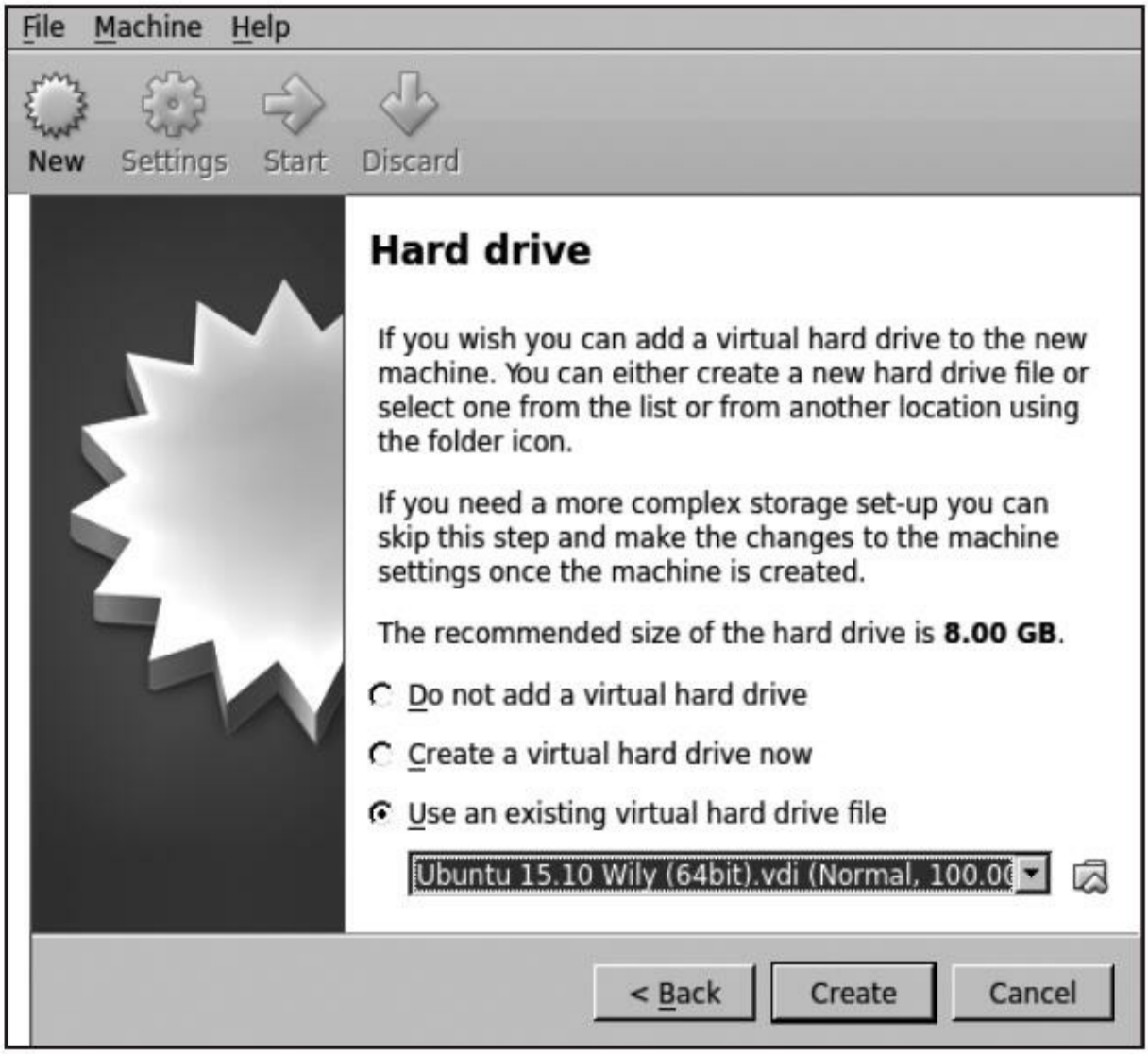
1.3.2 创建虚拟机

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并单击New图标。我们将新建一个虚拟机并使用之前下载的Ubuntu 15.10 Wily(64bit).vdi，这个硬盘镜像已经安装好Ubuntu 15.10了。设置虚拟机名称、类型和版本，如下图所示。

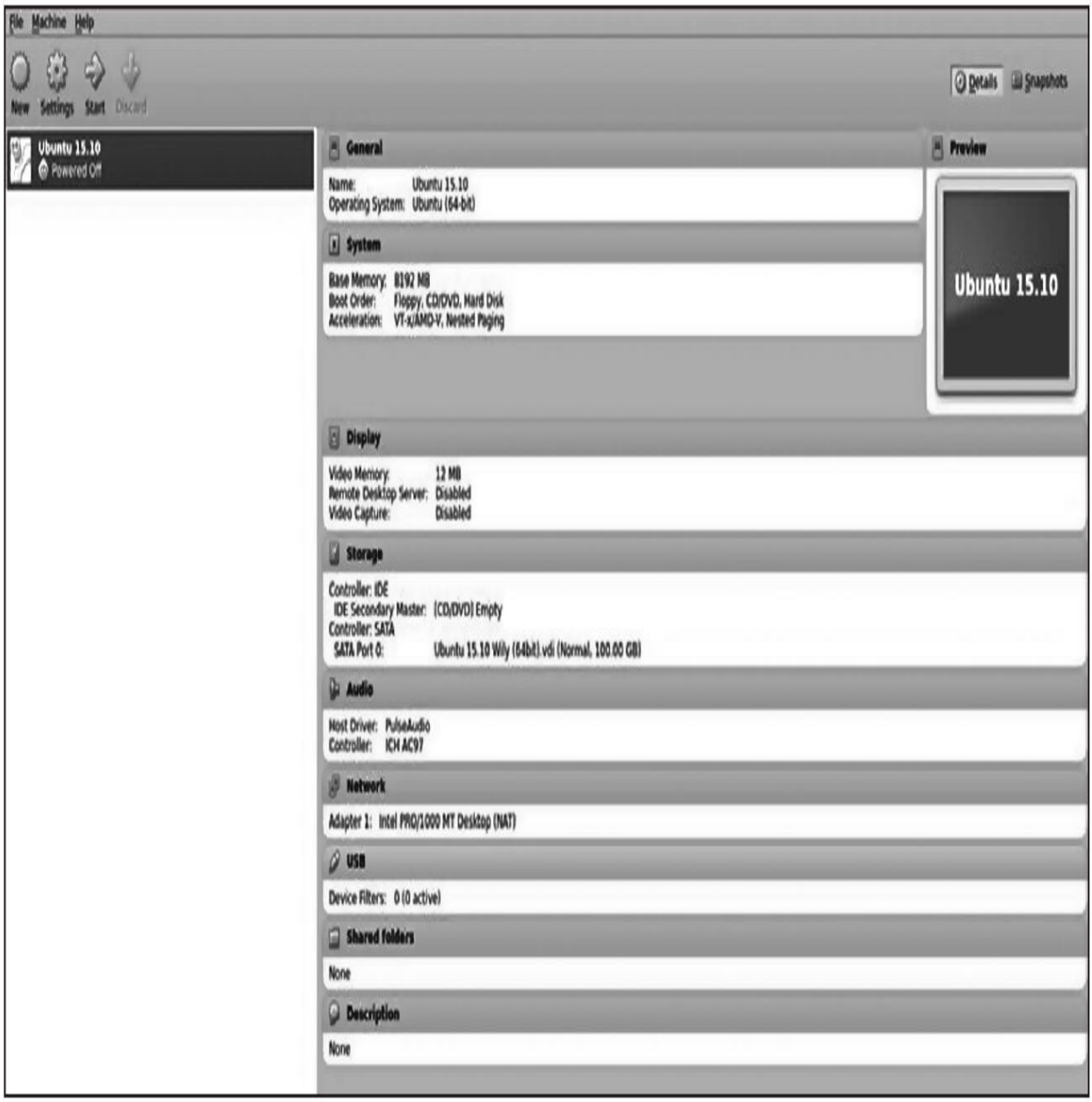


在下一个窗口中，可以配置新虚拟机的参数。保持默认配置并且仅仅改变虚拟机的名称。这个名称帮助我们区分不同的虚拟机。对于内存（RAM），当然多多益善，不过8GB应当够用了。

对于硬盘，使用已经下载的镜像Ubuntu 15.10 Wily(64bit).vdi，如下图所示。



完成这些之后，可以单击Start按钮启动虚拟机。注意，在启动前确认一下选择正确的虚拟机。在该示例中，只有一个，不过你可能有多个。



当虚拟机启动后，你将看到另一个窗口，如下图所示。这是已经安装好ROS的Ubuntu 15.10系统（使用osboxes.org密码登录）。

当完成这些步骤后，如同你在正常的计算机上按照之前的步骤安装ROS Kinetic一样，并且有了一个本书可以使用的完整版本的ROS Kinetic。你可以运行所有将会使用的示例和元功能包。遗憾的是，VirtualBox在使用部分实际外接设备的时候会有问题，并且可能无法使用这个ROS Kinetic镜像完成第4章中给出的例子。

1.4 通过Docker镜像使用ROS

Docker是一个开放平台，方便发布应用程序和完整的系统。在某些方面，它类似于虚拟机，但它更快、更灵活。有关详细信息，请参阅https://www.docker.com或https://dockerproject.org。

1.4.1 安装Docker

在Ubuntu中安装，只须运行下面命令：

```
$ sudo apt-get install docker.io
```

1.4.2 获取和使用ROS Docker镜像和容器

Docker镜像就像虚拟机或已经配置好的系统。用户只需要在提供这样镜像的服务器上下载它们。主服务器是Docker hub，网址https://hub.docker.com。在那可以搜索到不同系统和配置的Docker镜像。在这里，将使用已经发布的ROS Kinetic镜像。所有ROS Docker镜像都在官方网站的ROS镜像仓库中列出，网址为https://hub.docker.com/_/ros/。使用下面的命令获取ROS容器镜像：

```
$ docker pull ros
```

有可能会看到如下错误：

```
~$ docker pull ros
FATA[0000] Post http:///var/run/docker.sock/v1.18/images/create?fromImage=ros%3
Alatest: dial unix /var/run/docker.sock: permission denied. Are you trying to c
onnect to a TLS-enabled daemon without TLS?
```

可以通过更新系统或尝试将用户添加到docker组以解决此问题：

```
$ sudo usermod -a -G docker $(whoami)
```

你将会看到多个Docker镜像同时下载。每个镜像有不同的哈希名。下载需要一些时间，特别是在网速慢的情况下。完成后，将看到类似下图的内容：

```
~$ docker pull ros
latest: Pulling from ros
808ef855e5b6: Pull complete
267903aa9bd1: Pull complete
d28d8a6a946d: Pull complete
ab035c88d533: Pull complete
0b409bfffca0: Pull complete
aa8ec2450c6b: Pull complete
fea18d173ca4: Pull complete
5c9bb5cbe512: Pull complete
ae87b758dd0d: Pull complete
9cadeb3affd3: Pull complete
9c28b2d84bd7: Pull complete
0c7cd879039b: Pull complete
e8530b0325b8: Pull complete
8ab2cb273ccb: Pull complete
c7411052df49: Pull complete
ec05b0e2ef74: Pull complete
c366f9bb95b3: Pull complete
e795c4487953: Pull complete
Digest: sha256:078fbd221da8a3126eff2e283655f5a58e0342de272e38ef94631a1017568b86
Status: Downloaded newer image for ros:latest
```

使用下面的命令及对应的标签获取ROS Kinetic发行版：

```
$ docker pull ros:kinetic-robot
```

尽管你不需要了解，但镜像和容器是由Docker默认存储在/var/lib/docker文件夹下的。

容器下载完成后，可以使用以下命令以交互方式运行：

```
$ docker run -it ros
```

这就像在Docker容器中输入会话。此命令将从主镜像创建一个新容器。在里面有一个已经安装ROS Kinetic的完整Ubuntu系统。可以安装其他功能包，并像常规系统一样运行ROS节点。使用docker ps-a，可以检查所有可用的容器及其镜像来源。

我们必须在容器内设置ROS环境，以便使用ROS。也就是说，必须运行下面的命令：

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

Docker容器可以通过docker stop命令从其他终端停止，也可以使用docker rm进行删除。Docker还允许配置容器使用网络，以及将主机文件夹作为卷挂载到其中。除此之外，它还支持Python API等许多其他功能。所有这些都可以在官方文档网站https://docs.docker.com上找到。尽管如此，理论上docker run应该足够了，像通常的机器一样，甚至可以使用它的名字，通过SSH进入运行的Docker容器。还可以在另一个终端中使用下面的命令打开运行的容器（其中NAME是Docker容器的名称，可以使用docker ps-a）：

```
$ docker exec -it NAME bash
```

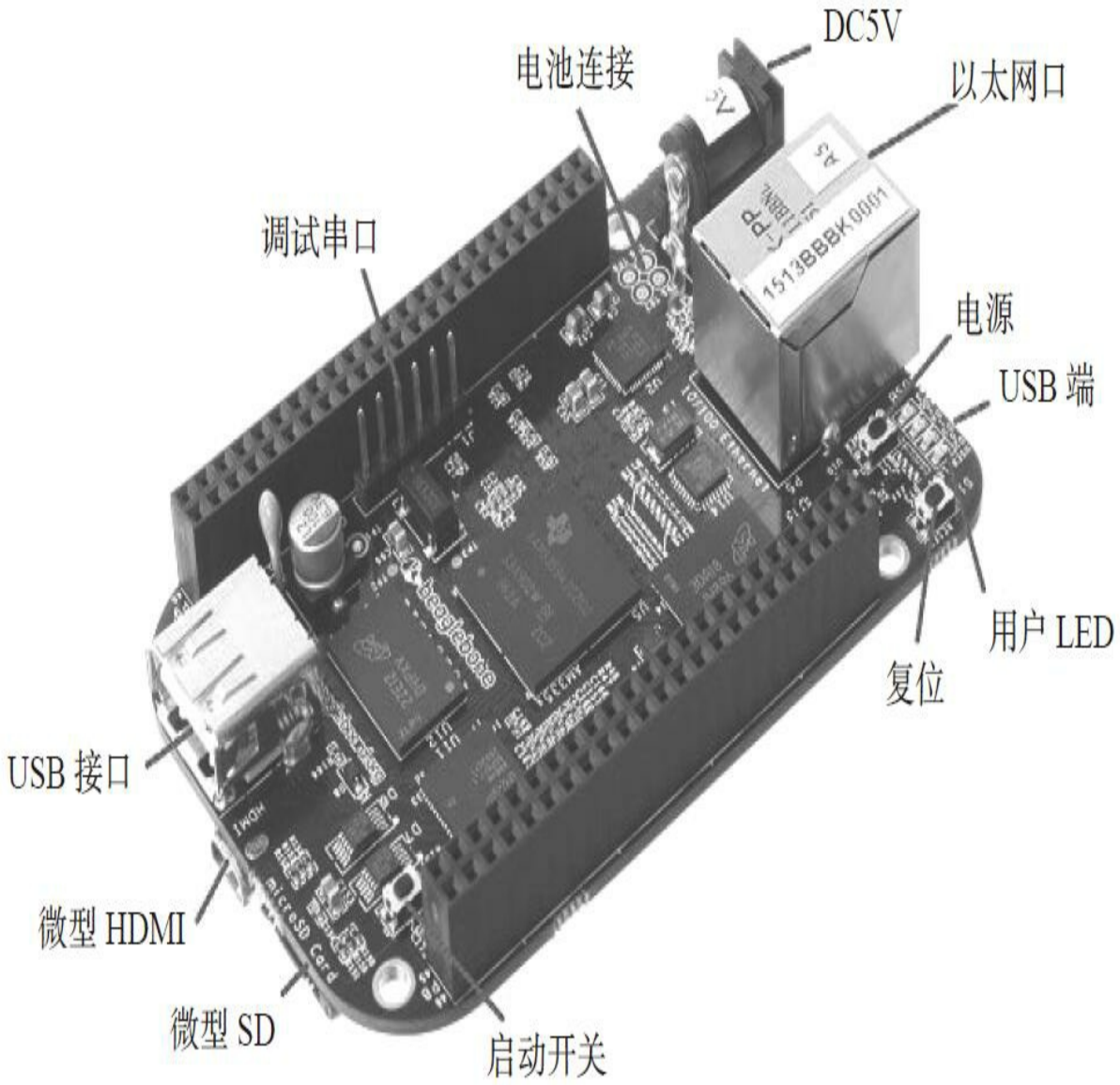
还可以使用docker build创建自己的Docker镜像，并指定在Dockerfile中安装镜像文件。甚至可以通过docker push在线发布它们，将它们贡献给社区或者简单地共享你的工作配置。本书有一个可用的Docker镜像和构建它的Dockerfile，可以通过在Dockerfile所在的文件夹中运行docker build找到它。此Docker镜像基本上是带有本书代码的ROS Kinetic的扩展。下载和安装的说明与其他代码在GitHub软件库中。

1.5 在BeagleBone Black上安装ROS Kinetic

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发行版的小型团队维护。他们的愿景是开发稳定且用户友好的操作系统。

考虑到社区的开发人员需要一个具有一些通用输入/输出（general purpose input/output，GPIO）引脚的车载计算机设备，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连接，以及两个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 ARM 16.04 Xenial armhf的镜像。

目前已经有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/kinetic/Installation/Angstrom。除此之外，由于这个发行版更常用，我们选择在Ubuntu ARM上安装ROS，此外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计算机等移动设备领域蓬勃发展。除了ARM cortex的运算性能不断增强，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，比如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等）也已经上市。

1.5.1 准备工作

将ROS安装到Beaglebone Black上，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。关于Beaglebone Black和Ubuntu ARM的更多信息可以在网站、论坛和书中找到。

首先，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 16.04 Xenial armhf：

```
$ wget https://rcn-ee.com/rootfs/2016-10-06/elinux/ubuntu-16.04.1-
console-armhf-2016-10-06.tar.xz
```

也可以在网址https://rcn-ee.com/rootfs上找到更新的版本。这个版本是官方文档http://elinux.org/BeagleBoardUbuntu中提及的版本之一。

下载镜像后，将其安装到microSD卡上。首先使用以下命令解压镜像：

```
$ tar xf ubuntu-16.04.1-console-armhf-2016-10-06.tar.xz
$ cd ubuntu-16.04.1-console-armhf-2016-10-06
```

插入一个至少2GB的microSD卡到计算机的读卡器，然后使用以下命令安装Ubuntu镜像：

```
$ sudo ./setup_sdcard.sh --mmc DEVICE --dtb BOARD
```

在前面的脚本中，DEVICE是microSD卡在系统中所处的设备，例如/dev/sdb，BOARD是开发板名。对于BeagleBone Black，它是beaglebone。因此，假设microSD卡位于/dev/mmcblk0中，并且使用的是BeagleBone Black，则命令如下所示：

```
$ sudo ./setup_sdcard.sh --mmc /dev/mmcblk0 --dtb beaglebone
```

如果不清楚分配给microSD卡的设备名，可以使用下面的命令：

```
$ sudo ./setup_sdcard.sh --probe-mmc
```

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。

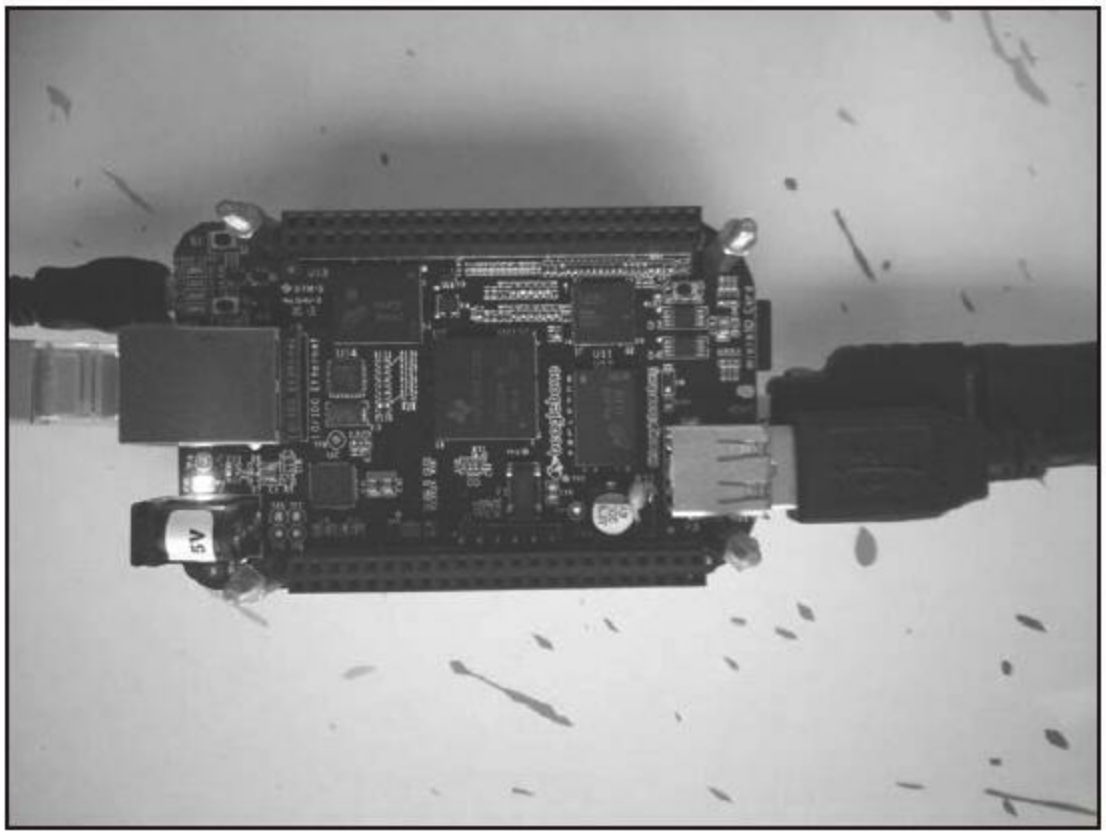
另一个简单的方法是使用网线并运行DHCP客户端来获取IP地址。

```
$ sudo dhclient eth0
```

为此，需要使用microSD卡启动BeagleBone Black。需要按住S2按钮，然后才能使用DC或USB连接器开启开发板。几分钟后，系统将显示登录提示符。使用用户ubuntu和密码temppwd（默认值）登录，然后运行上面的DHCP客户端命令，并连接网络。然后，可以检查分配的IP地址（查看inet addr：value）：

```
$ ifconfig eth0
```

在这里的设置中，已经将BeagleBone Black连接到以下设备（如下图所示）。



·HDMI（带microHDMI适配器）电缆可在网络设置期间查看屏幕上的终端提示符。之后我们可以利用SSH协议进入开发板。

·通过USB连接的键盘

·通过microUSB连接器供电

·以太网网线以访问互联网

网络配置完成后，应该安装ROS所需的功能包、程序和库。现在网络已经启动，也可以用SSH协议登录（假设分配给它的IP地址是192.168.1.6）：

```
$ ssh ubuntu@192.168.1.6
```

我们将按照http://wiki.ros.org/indigo/Installation/UbuntuARM中的说明进行操作，但需要进行更改才能使用ROS Kinetic（请注意，网络上仍是indigo）。第一步是设置软件库源，以便可以安装ROS：

```
$ sudo vi /etc/apt/sources.list
```

按如下方式添加软件库源：

```
deb http://ports.ubuntu.com/ubuntu-ports/ xenial main restricted universe
multiverse

#deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial main restricted
universe multiverse

deb http://ports.ubuntu.com/ubuntu-ports/ xenial-updates main restricted
universe multiverse

#deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial-updates main
restricted universe multiverse

#Kernel source (repos.rcn-ee.com) : https://github.com/RobertCNelson/
linux-stable-rcn-ee

#

#git clone https://github.com/RobertCNelson/linux-stable-rcn-ee

#cd ./linux-stable-rcn-ee

#git checkout `uname -r` -b tmp

#

deb [arch=armhf] http://repos.rcn-ee.com/ubuntu/ xenial main

#deb-src [arch=armhf] http://repos.rcn-ee.com/ubuntu/ xenial main
```

然后，运行以下命令更新源：

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

用于BeagleBone Black的操作系统在microSD卡上配置的空间为1～4GB。这个存储空间是非常有限的，如果我们想要使用大部分的ROS Kinetic包，可能就不够用了。为了解决这个问题，可以使用空间更大的SD卡，借助重新分区，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_mi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。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：

```
$ sudosu
```

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PC安装过程，这些内容比较熟悉。主要区别是当在BeagleBone Black上安装ROS时，不能安装ROS　full-desktop，必须单独安装每一个包。

1.5.2 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，将配置源列表，这基于安装在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 16.40 Xenial armhf，它也是Ubuntu ARM最受欢迎的版本。运行以下命令安装Ubuntu armhf软件库：

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

1.5.3 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，并且无人在未经所有者授权的情况下修改过代码或程序：

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-
key 0xB01FA116
```

1.5.4 安装ROS功能包

在安装ROS功能包之前，必须更新系统以避免出现库依赖的问题。

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

这部分安装在BeagleBone Black上略有不同。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。

可以尝试安装ROS-base，它称为ROS Bare Bones。ROS-base会安装ROS功能包以及编译库和通信库，但不包括GUI工具（出现提示时输入（Y）并按回车键）：

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

可以使用下面的命令来安装指定的ROS包：

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

如果需要查找在BeagleBone Black中可用的ROS功能包，可以运行下面的命令：

```
$ apt-cache search ros-kinetic
```

例如，下面的包为ROS正常工作的基础（已作为ros-base依赖项安装），可以使用apt-get install单独安装：

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

$ sudo apt-get install ros-kinetic-roslaunch

$ sudo apt-get install ros-kinetic-rosparam

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

虽然从理论上讲BeagleBone Black并不支持所有的ROS包，但实际上我们已经能够将在PC上开发的整个项目移植到BeagleBone Black中。我们成功尝试了很多包，只有安装rviz没有实现，确实不推荐在其上运行。

1.5.5 为ROS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。这可使你轻松地安装库并解决准备编译的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
```

1.5.6 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，这样它们就会在命令行窗口启动时自动加载：

```
$ echo "source /opt/ros/kinetic/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窗口中配置环境，运行如下命令：

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

1.5.7 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。如果要安装它，可以在Ubuntu中使用下面的命令行：

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

1.5.8 BeagleBone Black基本ROS示例

作为一个基本示例，可以在BeagleBone Black上的一个终端上运行ROS内核：

```
$ roscore
```

从另一个终端发布一个位姿消息（注意，可以在geometry_msgs/Pose后按Tab Tab，它会自动补全消息字段，然后需要更改默认值）：

```
$ rostopic pub /dummy geometry_msgs/Pose
Position:
x: 1.0
y: 2.0
z: 3.0
Orientation:
x: 0.0
y: 0.0
z: 0.0
w: 1.0 -r 10
```

现在，在笔记本电脑（在同一个网络中）上，可以将ROS_MASTER_URI设置为指向BeagleBone Black（在本例中为IP 192.168.1.6）：

```
$ export ROS_MASTER_URI=http://192.168.1.6:11311
```

现在你应该可以在笔记本电脑上看到BeagleBone Black发布的位姿：

```
$ rostopic echo -n1 /dummy
Position:
x: 1.0
y: 2.0
z: 3.0
Orientation:
x: 0.0
y: 0.0
z: 0.0
w: 1.0
---
```

如果使用PoseStamped，甚至可以在rviz中显示它。

此刻，可以通过http://wiki.ros.org/BeagleBone查询多个项目，以及使用Ångström操作系统替代Ubuntu的另一个安装选项，但目前它不支持ROS Kinetic。

1.6 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 Kinetic。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，还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，都无须重新安装操作系统，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。

第2章　ROS架构及概念

一旦完成了ROS的安装，你肯定会想“好了，我已经安装完成了，那么下一步要做什么呢？”在本章我们将学习ROS架构及它的组成。然后，我们会开始创建节点和包，并使用ROS自带的TurtleSim示例。

ROS的架构经过设计并划分成了三部分，每一部分都代表一个层级的概念：

·文件系统级（Filesystem level）

·计算图级（Computation Graph level）

·社区级（Community level）

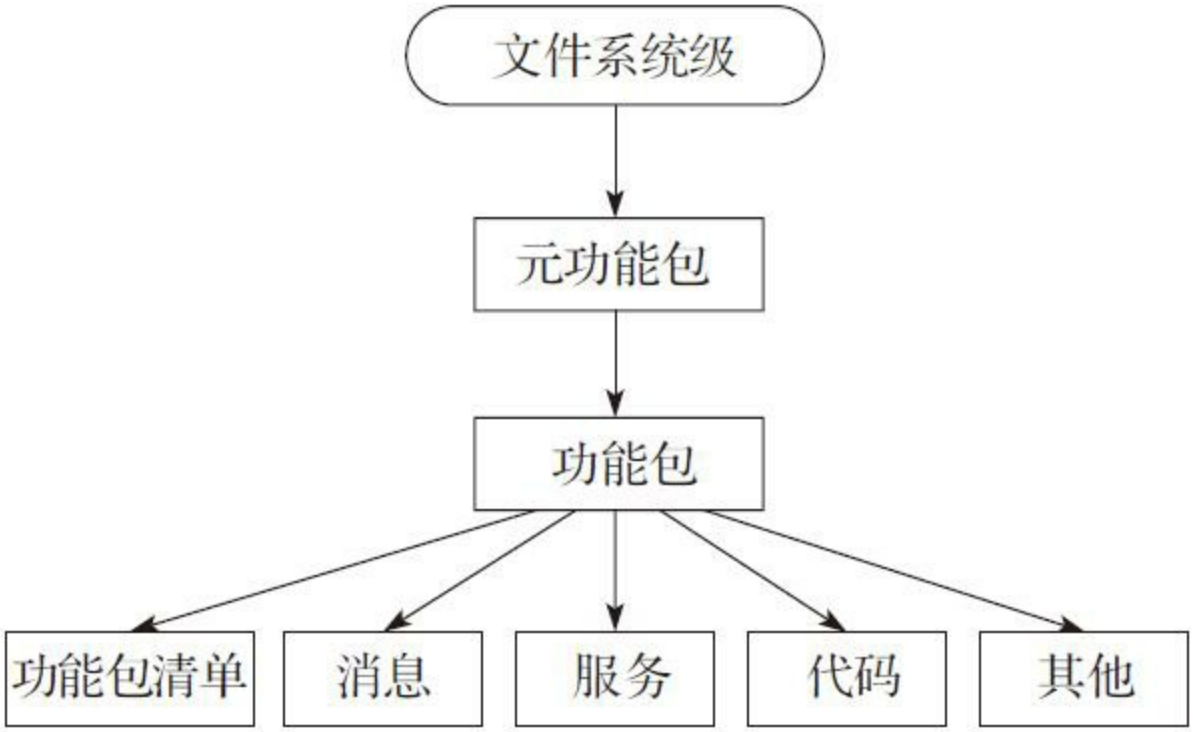
第一级是文件系统级。在这一级，我们会使用一组概念来解释ROS的内部构成、文件夹结构，以及工作所需的核心文件。

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，和大部分开源软件工程一样，有一个强大的社区支持，不仅提高了初学者理解复杂软件的能力，还解决了最常见的问题，正是由于开源社区的大力支持，ROS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同时认识到管理工程及其依赖的价值。ROS文件系统的主要目标是将项目构建的过程集中化，同时提供足够的灵活性和工具来分散之间的依赖性。



与其他操作系统类似，一个ROS程序的不同组件要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）：功能包构成ROS中的原子级。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运行时进程（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、许可证、依赖关系、编译标志等的信息。包清单由一个名为package.xml的文件管理。

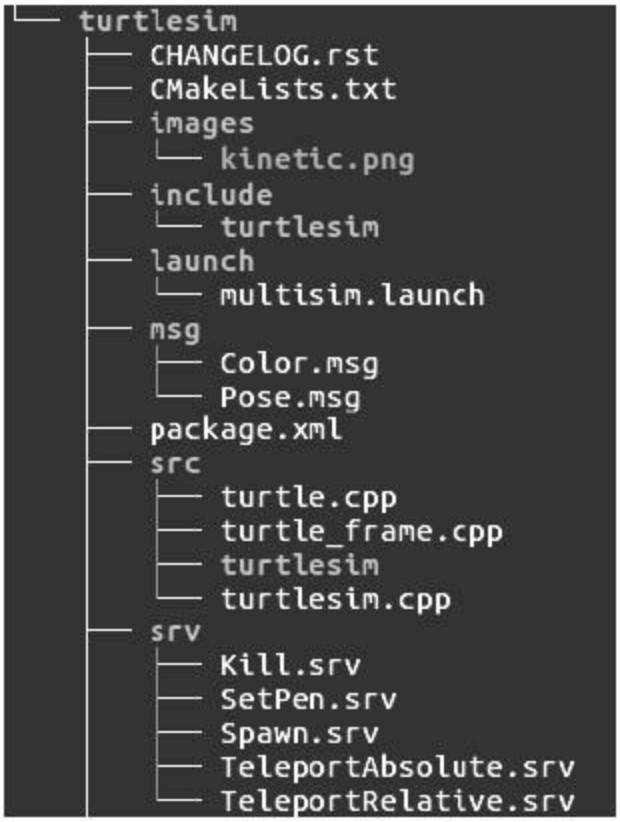
·元功能包（Metapackage）：如果你希望将几个具有某些功能的包组织在一起，那么你将会使用一个元功能包。在ROS Fuerte中，这种包的组织形式称为功能包集（Stack）。为了保持ROS简洁，功能包集被移除，现在使用元功能包实现这个功能。在ROS中，存在大量不同用途的元功能包，例如导航功能包集。

·元功能包清单（Metapackage manifest）：元功能包清单（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有很多标准类型的消息。消息类型的说明存储在my_package/msg/MyMessageType.msg中。

·服务类型（Service（srv）type）：服务描述说明存储在my_package/srv/MyServiceType.srv中，为ROS中由每个进程提供的服务定义请求和响应数据结构。

在下面的截图中，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。需要注意的是，截图显示了这些文件的一个简短列表，真正的功能包会包含更多内容。


turtlesim
CHANGELOG.rst
CMakeLists.txt
images
kinetic.png
include
turtlesim
launch
multisim.launch
msg
Color.msg
Pose.msg
package.xml
src
turtle.cpp
turtle_frame.cpp
turtlesim
turtlesim.cpp
srv
Kill.srv
SetPen.srv
Spawn.srv
TeleportAbsolute.srv
TeleportRelative.srv

2.1.1 工作空间

概言之，工作空间就是一个文件夹，其中包含功能包，功能包又包含源文件和环境或工作空间，从而提供编译这些功能包的一种方式。当你想同时编译不同的功能包时它非常有用，并且是集中化所有开发的一种好方式。

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（Source space）：在源空间（src文件夹）中，放置了功能包、项目、复制的包等。在这个空间中，最重要的一个文件是CMakeLists.txt。当在工作空间中配置包时，src文件夹中有CMakeLists.txt因为cmake调用它。这个文件是通过catkin_init_workspace命令创建的。

·编译空间（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（Development（devel）space）：devel文件夹用来保存编译后的程序，这些是无须安装就能用来测试的程序。一旦项目通过测试，就可以安装或导出功能包从而与其他开发人员分享。

用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，这两个命令编译出可执行文件。

ROS的另一个有趣的特性是它的覆盖（overlay）。当你正在使用ROS功能包（例如Turtlesim）时，可以使用安装版本，也可以下载源文件并编译它来使用你修改后的版本。

ROS允许使用你自己版本的功能包去替代安装版本。如果你正在升级已安装的功能包，这是非常有用的。或许此时你并不理解它的作用，但无须担心，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

包指的是一种特定结构的文件和文件夹组合。这种结构如下所示。

·include/package_name/：此目录包含了需要的库的头文件。

·msg/：如果开发需要非标准的消息，请把文件放在这里。

·scripts/：其中包括Bash、Python或任何其他脚本语言的可执行脚本。

·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希望的方式组织它。

·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包。

·catkin_create_pkg：使用此命令创建一个新的功能包。

·catkin_make：使用此命令来编译工作空间。

·rosdep：使用此命令安装功能包的系统依赖项。

·rqt_dep：此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）的插件。选择一个包并查看依赖关系。

要在文件夹和功能包之间移动文件，ROS提供了非常有用的rosbash功能包，其中包含了一些非常类似于Linux命令的命令。下面是一些示例。

·roscd：此命令用于更改目录，类似于Linux中的cd命令。

·rosed：此命令用来编辑文件。

·roscp：此命令用于从功能包复制文件。

·rosd：此命令列出功能包的目录。

·rosls：此命令列出功能包下的文件，类似于Linux中的ls命令。

文件package.xml必须在每个功能包中，它用来说明此包相关的各类信息。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包或元功能包。

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方便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中代码所需要的包。以下截图是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 元功能包

如前所述，元功能包（或简称元包）是一些只有一个文件的特殊包，这个文件就是package.xml。它不包含其他文件，如代码等。

元功能包用于指代其他按照类似功能特性分组的包，例如导航功能包集、ros_tutorials等。

使用迁移的特定规则，可以将ROS Fuerte中的功能包和功能包集转换为Kinetic和catkin。具体参考http://wiki.ros.org/catkin/migrating_from_rosbuild。

在下图中，可以看到在ros_tutorials元功能包中package.xml的内容。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，在下图中也可以看到这些标记。

```
<?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元功能包，可以使用下面的命令：

```
$ rosstack find ros_tutorials
```

显示路径为：/opt/ros/kinetic/share/ros_tutorials。

通过下面的命令查看里面的代码：

```
$ vim /opt/ros/kinetic/ros_tutorials/package.xml
```

记住，Kinetic使用元功能包，不是功能包集，但rosstack find命令也可用于查找元功能包。

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。如果创建了一条新的消息，那么就要把它放到功能包的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作为扩展名。

消息类型必须具有两个主要部分：字段（field）和常量（constant）。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，如叫作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ROS消息所使用的很多标准数据类型。

基本类型	序列化	C++	Python
bool(1)	无符号 8 位 int	uint8_t(2)	bool
int8	有符号 8 位 int	int8_t	int
uint8	无符号 8 位 int	uint8_t	int(3)
int16	有符号 16 位 int	int16_t	int
uint16	无符号 16 位 int	uint16_t	int
int32	有符号 32 位 int	int32_t	int
uint32	无符号 32 位 int	uint32_t	int
int64	有符号 64 位 int	int64_t	long
uint64	无符号 64 位 int	uint64_t	long
float32	32 位 IEEE float	float	float
float64	64 位 IEEE float	double	float
string	ascii string (4)	std::string	string
time	secs/nsecs 有符号 32 位 ints	ros::Time	rospy.Time
duration	secs/nsecs 有符号 32 位 ints	ros::Duration	rospy.Duration

ROS消息中的一种特殊数据类型是标头类型，它主要用于添加时间、坐标系和序列号等。标头类型还允许对消息进行编号。通过在标头类型内部附加信息，我们可以知道是哪个节点发出的消息，或者可以添加对于用户透明的功能以及一些能够被ROS处理的功能。

标头类型包含以下字段：

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

可以通过以下命令查看消息的结构：

```
$ rosmsg show std_msgs/Header
```

我们将在后续的章节中看到，正是通过标头类型才能够记录当前机器人运行的时间戳和坐标系。

在ROS中有一些处理消息的工具。例如rosmsg命令行工具能够输出消息定义信息，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/子目录下的.srv文件中。

要调用服务，需要使用该功能包的名称及服务名称。例如，可以将sample_package1/srv/sample1.srv文件称为sample_package1/sample1服务。

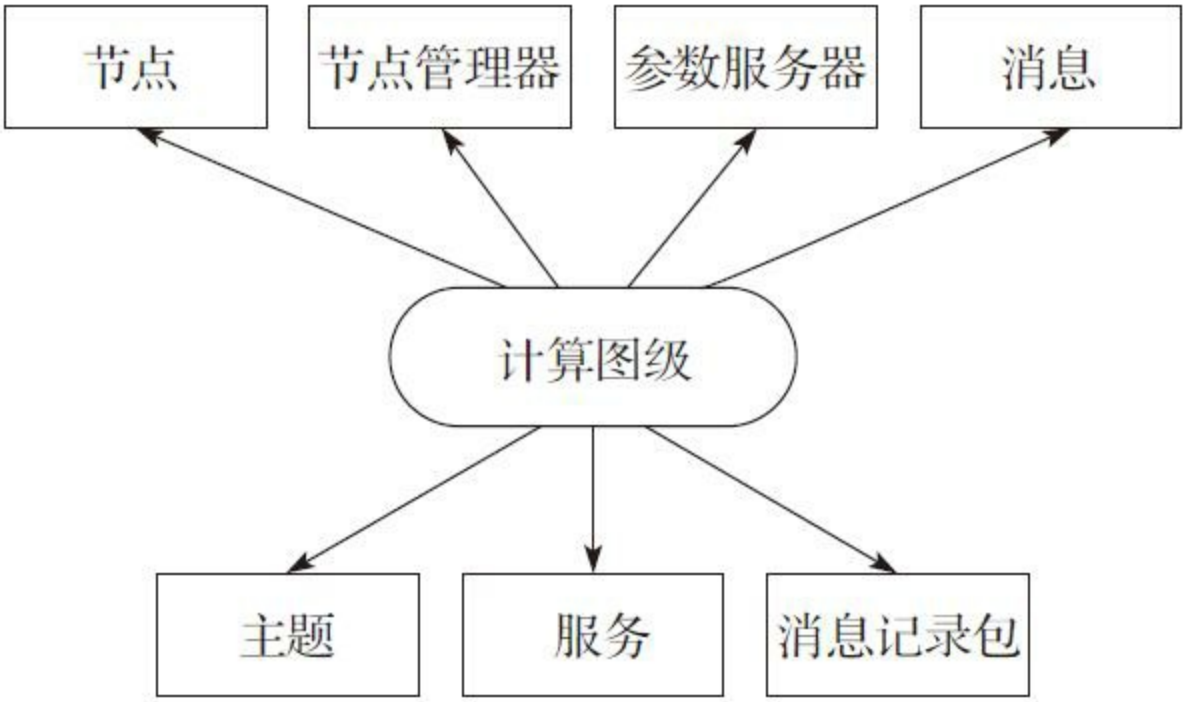
ROS中有一些在服务上执行某些操作的工具。rossrv工具能输出服务说明、.srv文件所在的包，并可以找到使用某一服务类型的源文件。

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。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，获取其他节点发送的信息，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。你最好让众多节点都具有单一的功能，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。如果在整个ROS中没有节点管理器，就无法与节点、服务、消息等通信。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密钥存储在一个系统的核心位置。通过参数，就能够在运行时配置节点或改变节点的工作任务。

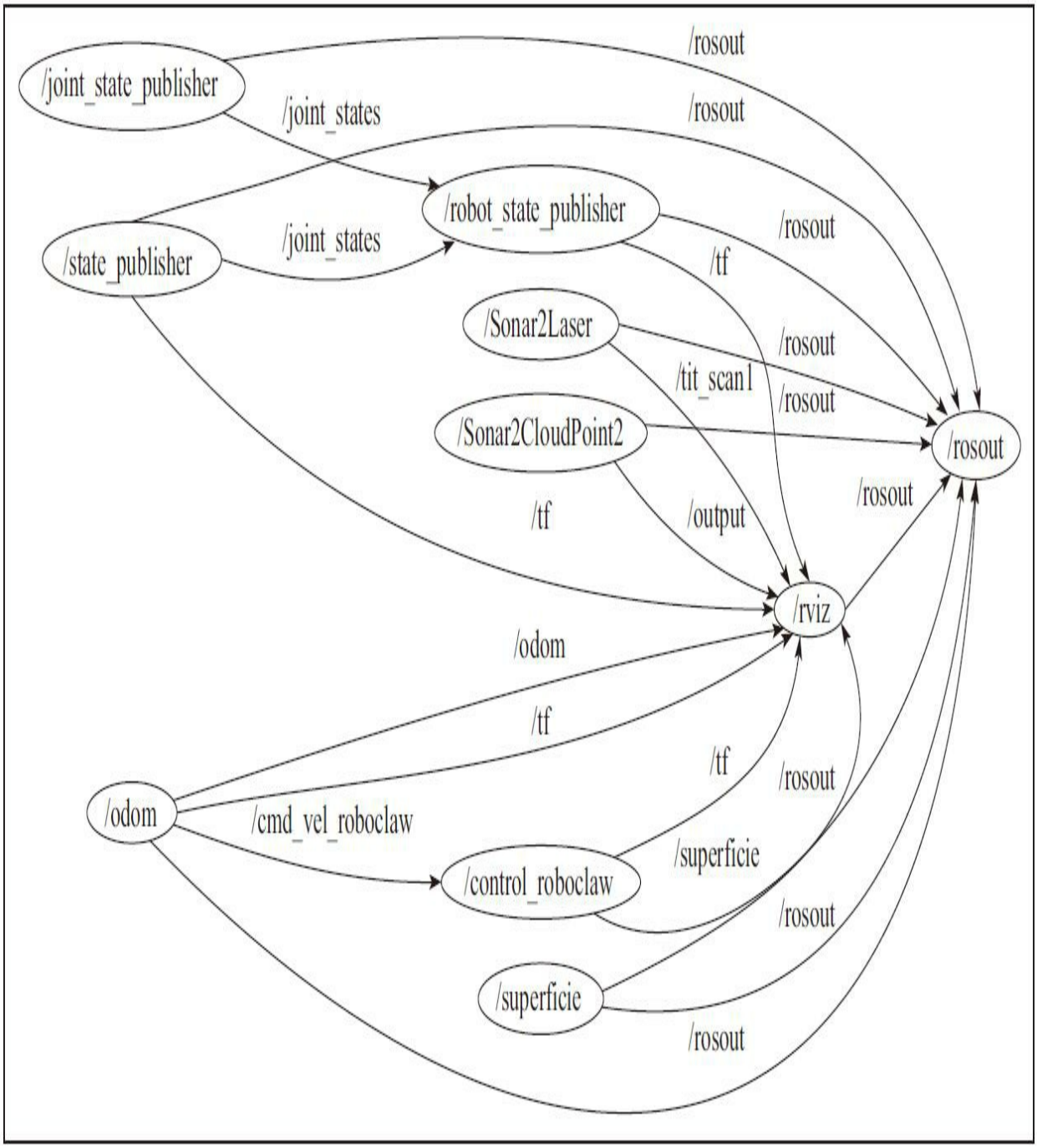
·消息（message）：节点通过消息完成彼此的沟通。消息包含一个节点发送到其他节点的信息数据。ROS中包含很多种标准类型的消息，同时也可以基于标准消息类型开发自定义类型的消息。

·主题（topic）：每个消息都必须有一个名称以便被ROS网络分发。当一个节点发送数据时，我们就说该节点正在向主题发布主题。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，而不需要任何其他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，完全无须知晓对方的存在。主题的名称必须是唯一的，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。用于创建该图的工具是rqt_graph，在第3章中将学习到更多的相关知识。



这些概念在ros-comm软件库中实现。

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（动态加载节点）——内部可通信的多个节点。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二义性。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提供了处理节点和显示节点信息的工具，如rosnode。rosnode是一个用于显示节点信息的命令行工具，例如列出当前正在运行的节点。支持的命令如下所示。

·rosnode info NODE：输出当前节点信息。

·rosnode kill NODE：结束当前运行节点或发送给定信号。

·rosnode list：列出当前活动节点。

·rosnode machine hostname：列出某一特定计算机上运行的节点或列出主机名称。

·rosnode ping NODE：测试节点间的连通性。

·rosnode cleanup：将无法访问节点的注册信息清除。

在接下来的章节中，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在启动该节点时更改参数。此功能

使我们能够改变节点名称、主题名称和参数名称。我们无须重新编译代码就能重新配置节点，这样就可以在不同的场景中使用该节点。

一个改变主题名称的例子如下所示：

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

此命令将主题名称从topic1改为/level1/topic1。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：

```
$ rosrun 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，如下所示。

·__name：为节点名称保留的一个特殊关键字。

·__log：为记录节点中日志文件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（topics）是节点间用来传输数据的总线。通过主题进行消息传输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。一个主题可以有多个订阅者，也可以有多个发布者，但是用不同的节点发布同样的主题时要慎重，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ROS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中的主题可以使用TCP/IP和UDP传输。基于TCP传输称为TCPROS，它使用TCP/IP长连接。这是ROS默认的传输方式。

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

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：输出主题的信息，例如其消息类型、发布者、订阅者。

·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，即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以RPC方式获得应答时，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，服务类型是包名和.srv文件名的组合。例如chapter2_tutorials/srv/chapter2_srv1.srv文件的服务类型是chapter2_tutorials/chapter2_srv1。

ROS关于服务的命令行工具有两个：rossrv和rosservice。我们可以通过rossrv看到有关服务数据结构的信息，并且与rosmsg具有完全相同的用法。

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，从而将信息发送到另一个节点。消息具有简单的数据结构，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中遵循以下标准命名方式：包名/文件名.msg。例如，std_msgs/msg/String.msg的消息类型是std_msgs/String。

ROS使用命令行工具rosmsg来获取有关消息的信息。常用参数如下所示。

·rosmsg show：显示一条消息的字段。

·rosmsg list：列出所有消息。

·rosmsg package：列出包中的所有消息。

·rosmsg packages：列出所有具有该消息的包。

·rosmsg users：搜索使用该消息类型的代码文件。

·rosmsg md5：显示一条消息的MD5求和结果。

2.2.5 消息记录包

消息记录包（bag）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。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。借助它可以播放、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，可以使用此功能来调试算法。

要使用消息记录包文件，可以使用以下ROS工具。

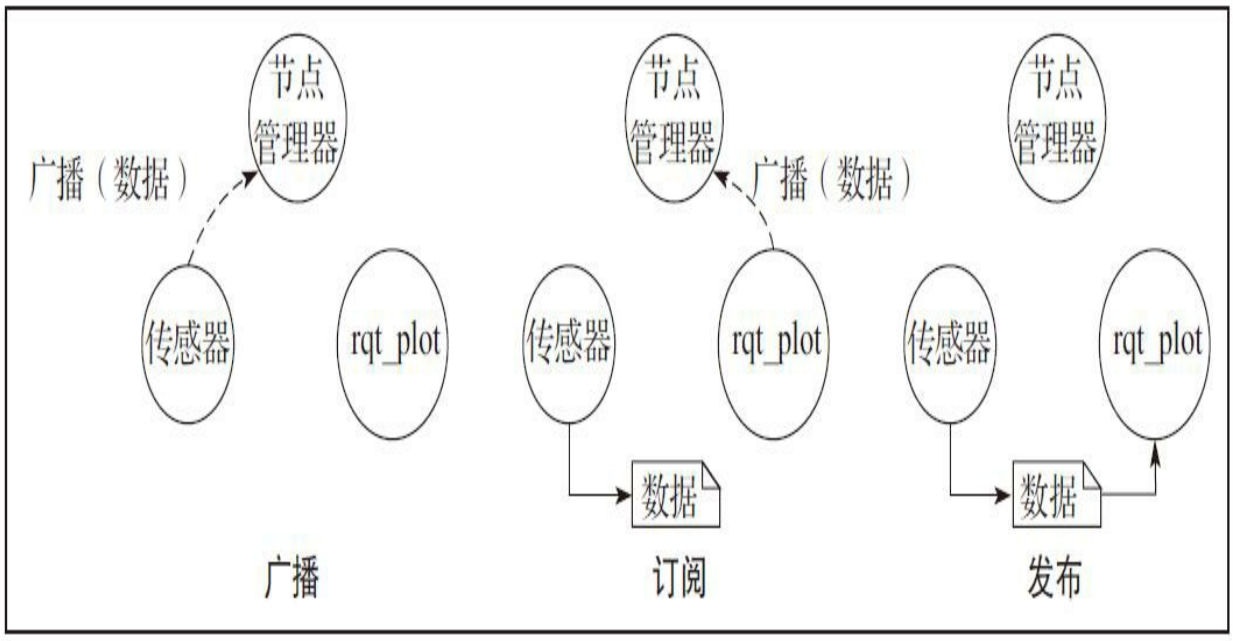
·rosbag：用来录制、播放和执行其他操作。

·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（ROS master）为ROS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（Parameter Server）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，其中包括以下类型。

·32位整数

·布尔值

·字符串

·双精度浮点

·ISO 8601日期

·列表

·基于64位编码的二进制数据

ROS中关于参数服务器的工具是rosparam。其支持的参数如下所示。

·rosparam list：列出了服务器中的所有参数。

·rosparam get parameter：获取参数值。

·rosparam set parameter value：设置参数值。

·rosparam delete parameter：删除参数。

·rosparam dump file：将参数服务器保存到一个文件中。

·rosparam load file：加载参数文件到参数服务器中。

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和知识。这些资源包括以下几个。

·发行版（Distribution）：ROS发行版是可以独立安装、带有版本号的一系列元功能包。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依赖于代码存储库的联网，在这里不同的机构能够发布和分享各自的机器人软件组件。

·ROS维基（ROS Wiki）：ROS Wiki是用于记录有关ROS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、编写教程以及做其他事情。

·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）：可以看到定期更新、照片和新闻，网址是http://www.ros.org/news。

2.4 ROS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，你将看到一些用于练习的示例，以及关于创建包、使用节点、使用参数服务器和通过Turtlesim移动仿真机器人的例子。

2.4.1 ROS文件系统导览

我们通过命令行工具来浏览ROS的文件系统。我们将会解释最常用的部分。

为了获得功能包和功能包集的信息，比如，其路径、依赖关系等，我们将使用rospack、rosstack命令进入功能包和功能包集，并列出其中的内容。

例如，如果你想要找turtlesim包的路径，可以使用以下命令：

```
$ rospack find turtlesim
```

将获得以下信息：

```
/opt/ros/kinetic/share/turtlesim
```

同样，如果你想要找到你已经在系统中安装过的某个元功能包，示例如下：

```
$ rosstack find ros_comm
```

你将获得到ros-comm元功能包的路径，如下：

```
/opt/ros/kinetic/share/ros_comm
```

要获得功能包或功能包集下面的文件列表，需要使用：

```
$ rosls turtlesim
```

之前命令的输出如下所示：

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

更改当前工作目录，尤其是进入某个文件夹，可以使用roscd命令完成：

```
$ roscd turtlesim
$ pwd
```

将获得以下新路径：

```
/opt/ros/kinetic/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。

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/kinetic/share:/opt/ros/kinetic/stacks
```

我们将要创建的文件夹位于~/dev/catkin_ws/src/中。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
```

当创建工作空间文件夹后，里面并没有功能包，只有CMakeList.txt文件。下一步是编译工作空间，为此，可使用下面的命令：

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

现在，如果输入ls命令，可以看到以上命令创建的新文件夹，分别是build和devel文件夹。

为完成配置，使用下面的命令：

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

这一步只重新加载了setup.bash文件。如果关闭并打开一个新的命令行窗口，也将得到同样的效果。你应该已经在你的~/.bashrc文件末尾加入了该命令行，因为我们在第1章用过。如果没有，可以使用下面的命令添加它：

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

2.4.3 创建ROS功能包和元功能包

就像之前所说，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中创建新的功能包：

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

$ catkin_create_pkg chapter2_tutorials std_msgs roscpp
```

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

```
catkin_create_pkg [package_name] [dependency1] ... [dependencyN]
```

这些依赖项包括以下几个。

·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，可以使用rospack、roscd和rosls命令来获取新的功能包信息。下面是可以执行的一些操作。

·rospack profile：此命令用于通知用户ROS中新添加的内容。在安装任何新功能包之后使用它。

·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。当编译功能包的时候，主要是代码的编译过程，不仅包括用户添加的代码，还包括由消息和服务生成的代码。

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，必须在workspace文件夹中运行catkin_make命令。如果在其他文件夹中这样做，命令无法执行，下面是一个例子：

```
$ roscd chapter2_tutorials/

$ catkin_make
```

当在chapter2_tutorials文件夹中试图用catkin_make编译功能包时，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
```

当在catkin_ws文件夹中执行catkin_make命令时，则会编译成功。最后，如果编译单个功能包，可使用如下格式的命令：

```
$ catkin_make --pkg <package name>
```

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，一旦编译，这些可执行文件就位于devel空间中。要学习和了解有关节点的知识，要使用一个名为turtlesim的典型功能包进行练习。

如果你进行了ROS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-kinetic-ros-tutorials
```

在开始之前，重要的是，打开终端并且使用如下命令启动roscore：

```
$ roscore
```

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

```
$ rosnode
```

你会获得一个可接受参数清单，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，可使用以下命令：

```
$ rosnode list
```

你会看到运行的节点仅有/rosout。这是正常的，因为这个节点总是通过roscore启动。

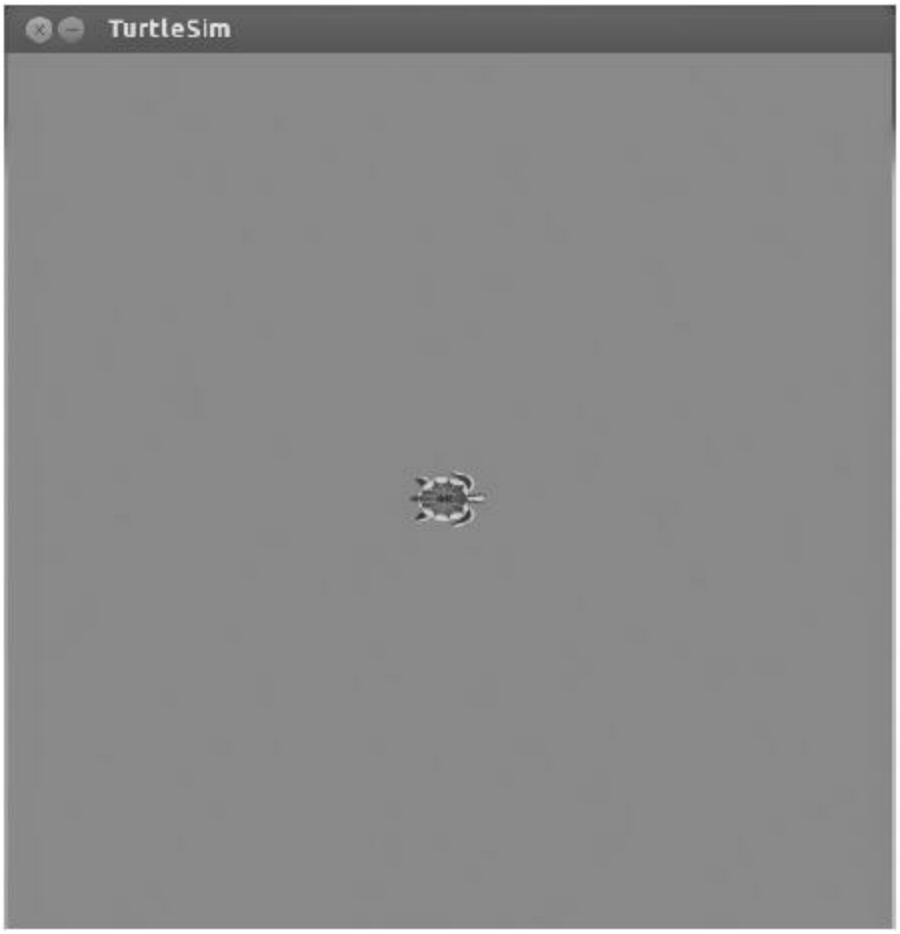
使用rosnode工具提供的参数可以获得此节点的所有信息。也可以使用下列命令获得更详细的信息：

```
$ rosnode info

$ rosnode ping

$ rosnode machine

$ rosnode kill

$ rosnode cleanup
```

现在要用rosrun命令启动一个新的节点，如下所示：

```
$ rosrun turtlesim turtlesim_node
```

我们看到出现了一个新的窗口，窗口中间有一只小海龟，如下图所示。



如果我们再查看节点列表，会看到出现了一个新的节点，叫作/turtlesim。可以通过使用rosnode info nameNode命令查看节点信息。

使用以下命令，可以看到很多能用于程序调试的信息：

```
$ rosnode info /turtlesim
```

上一个命令行输出如下信息：

```
--------------------------------------------------------------------------------
Node [/turtlesim]
Publications:
 * /turtle1/color_sensor [turtlesim/Color]
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/pose [turtlesim/Pose]

Subscriptions:
 * /turtle1/cmd_vel [geometry_msgs/Twist]

Services:
 * /turtle1/teleport_absolute
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
 * /reset
 * /spawn
 * /clear
 * /turtle1/set_pen
 * /turtle1/teleport_relative
 * /kill


contacting node http://daneel:38674/ ...
Pid: 3881
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
 * topic: /turtle1/cmd_vel
    * to: /teleop_turtle (http://daneel:44645/)
    * direction: inbound
    * 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。

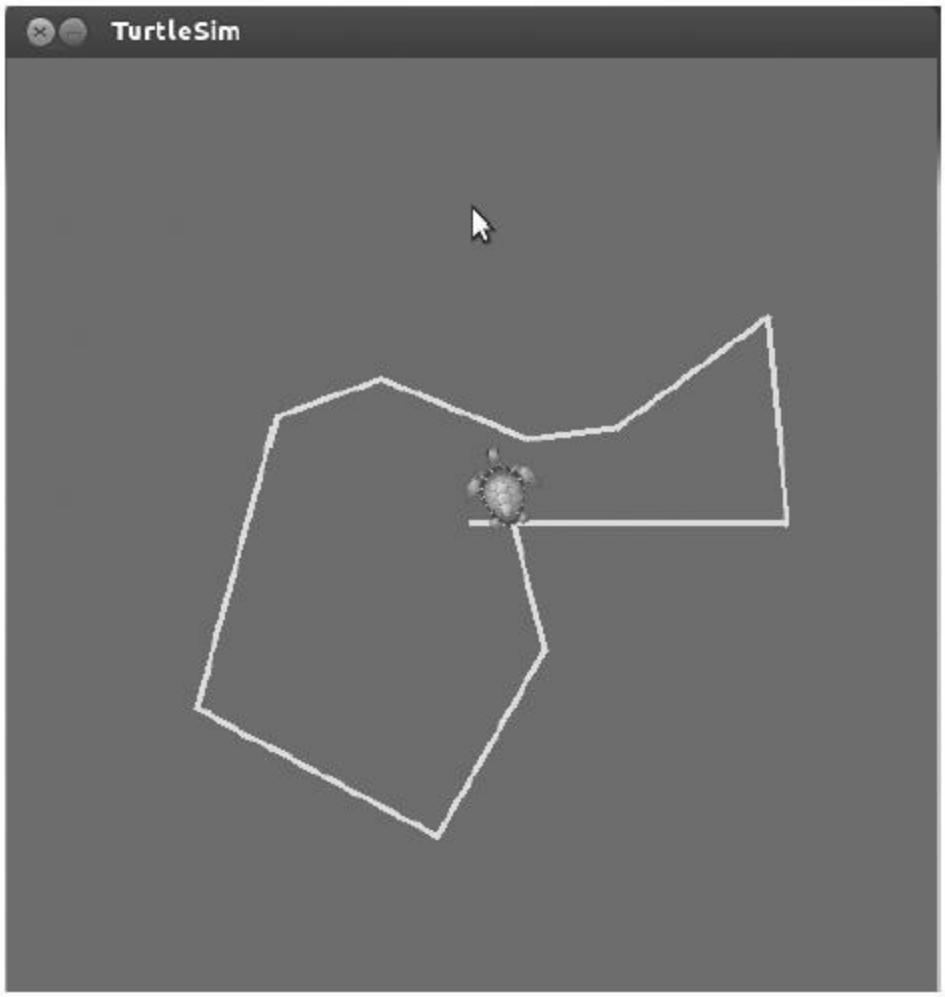
·rostopic info TOPIC：输出活动主题的信息。

·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，可以使用箭头键移动海龟，如下图所示。

```
--------------------------------------------------------------------------------
Node [/teleop_turtle]
Publications:
 * /turtle1/cmd_vel [geometry_msgs/Twist]
 * /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
 * /teleop_turtle/get_loggers
 * /teleop_turtle/set_logger_level


contacting node http://daneel:44645/ ...
Pid: 4156
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
 * topic: /turtle1/cmd_vel
    * to: /turtlesim
    * direction: outbound
    * transport: TCPROS
```

turtle_teleop_key执行时，为什么小海龟会移动呢？

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Publications部分有一个主题叫/turtle1/cmd_vel[geometry_msgs/Twist]；在第二个节点的Subscriptions部分有一个主题叫/turtle1/cmd_vel[geometry_msgs/Twist]：

```
$ rosnode info /teleop_turtle
```

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。可以使用以下命令行查看主题清单：

```
$ rostopic list
```

输出如下：

```
/rosout
  /rosout_agg
/turtle1/colour_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输出结果：

```
---
linear:
x: 0.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 2.0
---
```

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输出结果：

```
Geometry_msgs/Twist
```

如果你想要看到消息字段，可以使用以下命令：

```
 $ rosmsg show geometry_msgs/Twist
```

你会看到类似如下的输出结果：

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

这些工具非常有用，因为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，如下图所示。


TurtleSim

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。

·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
```

要调用服务，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。

现在尝试其他的服务，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
```

输入下面的命令：

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

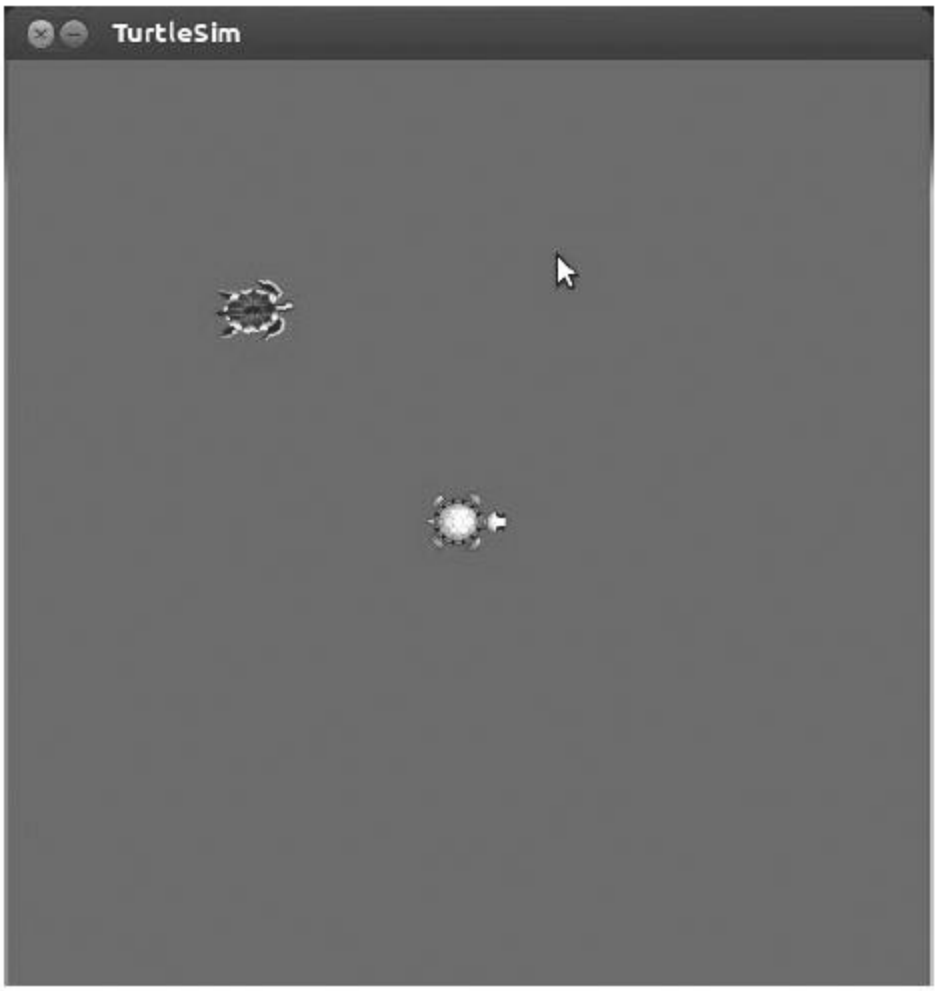
将会看到如下类似的结果：

```
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）以及新海龟的名称：

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

我们会获得下面的结果：


TurtleSim

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：将参数转储到一个文件中。

·rosparam delete parameter：删除参数。

·rosparam list：列出参数名。

例如，查看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
/rosdistro
/roslaunch/uris/host_aaronmr_laptop__60878
/rosversion
/run_id
```

上面的背景（background）参数是turtlesim节点的参数。这些参数

可以改变窗口的颜色，窗口最初为蓝色。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，可以保存或加载参数服务器的内容。

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文件中或者从存储库中下载它：

```
#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。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包含了要使用的消息类型：

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

此刻，初始化该节点并设置其名称，请记住，该名称必须是唯一的：

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

下面是进程的处理程序，它允许我们与环境交互：

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

将节点实例化成发布者，并将所发布主题和类型的名称告知节点管理器。名为message，第二个参数是缓冲区的大小。如果主题发布数据的速度较快，那么将缓冲区设置为至少存放1000条消息，如下所示：

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

在本例中，发送数据的频率设置为10Hz：

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

当按Ctrl+C组合键或ROS停止所有节点时，ros::ok()行会停止该节点：

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

在这里，创建一个消息变量，变量的类型必须符合发送数据的要求：

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

继续发布消息，本例中，使用之前发布器的定义发布消息：

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

spinOnce函数负责处理所有ROS内部事件和操作，例如读取订阅的消息。spinOnce在主循环中执行一次迭代允许用户执行操作，与spin函数不同，spinOnce在不中断的情况下运行主循环：

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

最后，按照10Hz的频率将程序挂起。

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

现在创建另一个节点。将下面的代码复制到example1_b.cpp文件中或者从存储库中下载它：

```
#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。要包含头文件和主题所使用的消息类型。

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

下面的函数类型是回调，每次该节点收到一条消息时都将调用此函数。该函数可以处理数据。在本示例中，我们将收到的数据在命令行窗口中输出：

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

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条消息，处理消息的回调函数为messageCallback：

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

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

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

2.4.10 编译节点

当使用chapter2_tutorials包时，需要自行编辑CMakeLists.txt文件。可以使用你喜欢的编辑器或直接使用rosed工具。这里将会在Vim编辑器下打开这个文件：

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

将以下命令行复制到文件的末尾处：

```
include_directories(
include
  ${catkin_INCLUDE_DIRS}
)

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

add_dependencies(example1_a
chapter2_tutorials_generate_messages_cpp)
add_dependencies(example1_b
chapter2_tutorials_generate_messages_cpp)

target_link_libraries(example1_a ${catkin_LIBRARIES})
target_link_libraries(example1_b ${catkin_LIBRARIES})
```

现在使用catkin_make工具来编译包和全部的节点：

```
$ cd ~/dev/catkin_ws/
$ catkin_make --pkg chapter2_tutorials
```

如果在你的电脑上还没有启动ROS，需要首先调用：

```
$ roscore
```

可以使用rosnode list命令检查ROS是否运行：

```
$ rosnode list
```

然后，在不同的命令行窗口下分别运行两个节点：

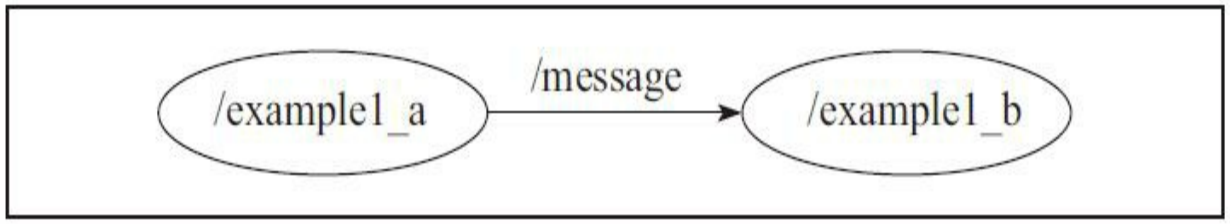
```
$ rosrun chapter2_tutorials example1_a
```

```
$ rosrun 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 ]
...
```

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。



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

```
$ rosnode list

$ rosnode info /example1_a

$ rosnode info /example1_b

$ rostopic list

$ rostopic info /message

$ rostopic type /message

$ rostopic bw /message
```

2.4.11 创建msg和srv文件

在这一节中，我们将会学习如何在节点中创建msg和srv文件。它们是用于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创建所需的代码，以便msg和srv文件能够被节点使用。

第一步，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消息的节点。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，在文件中添加以下行：

```
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，用下面的命令进行编译：

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

为了检查编译是否成功，使用下面的rosmsg命令：

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

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

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

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

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

首先，按下面的方式从chapter2_tutorials包中打开package.xml文件夹：

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

找到下面的行并取消注释：

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

使用以下命令打开CMakeLists.txt：

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

找到下面的行，取消注释，并改为正确的数据：

```
catkin_package(
  CATKIN_DEPENDS message_runtime
)
```

为了生成消息，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部分如下所示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
 )
```

使用如下rossrv命令测试编译是否成功：

```
$ 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：一个服务器和一个客户端。

在chapter2_tutorials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：这些行包含必要的头文件和我们创建的srv文

件：

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

这个函数会对3个变量求和，并将计算结果发送给其他节点：

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

在这里，创建服务并在ROS中发布广播。

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

在第2个文件example2_b.cpp中，添加以下代码：

```
#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：以add_3_ints为名称创建一个服务的客户端。

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

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

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

这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。

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

为了编译新节点，在CMakeList.txt文件中添加以下行：

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

add_dependencies(example2_a
chapter2_tutorials_generate_messages_cpp)
add_dependencies(example2_b
chapter2_tutorials_generate_messages_cpp)
target_link_libraries(example2_a ${catkin_LIBRARIES})
target_link_libraries(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
```

并且你会看到如下输出结果：

```
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 launch文件

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中执行。想象一下，如果在每一个命令行窗口中启动20个节点会是多么恐怖的一件事情！

通过launch文件可以在命令行窗口中方便地实现以上任务，只需要启动后缀名为.launch的配置文件。

为了练习这个例子，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>
```

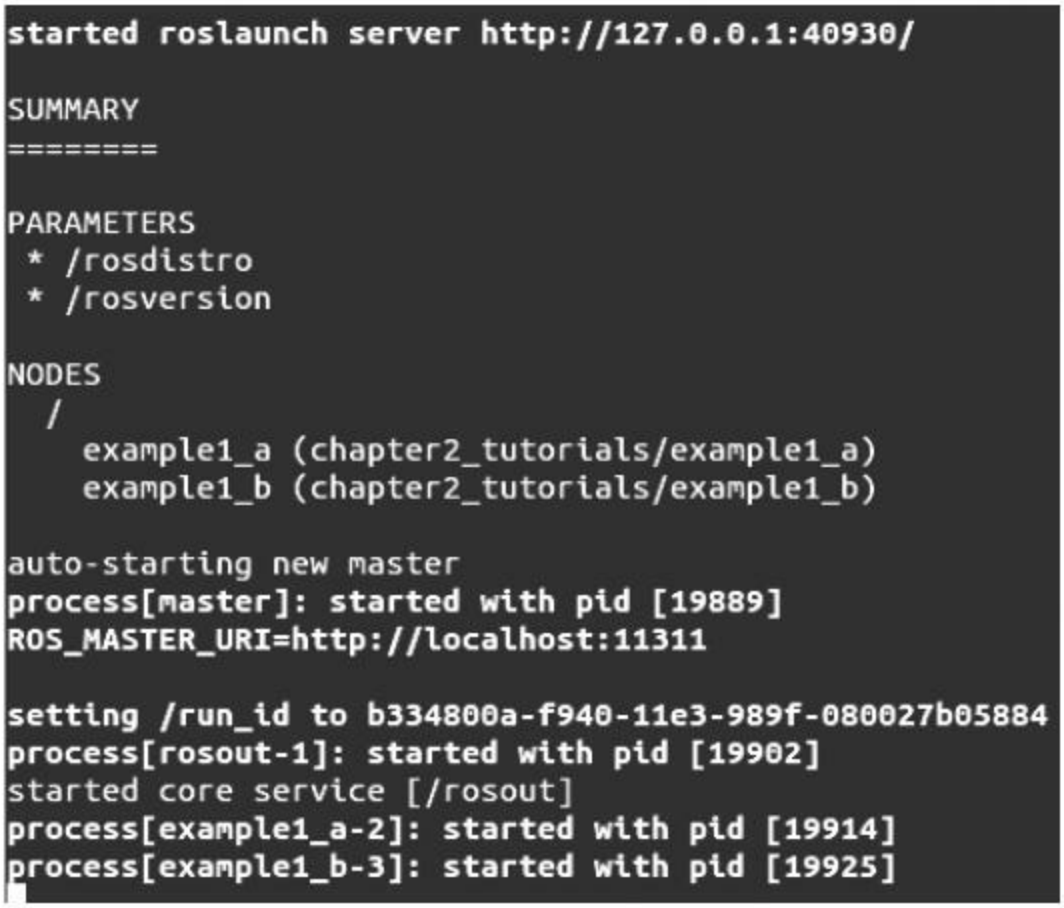
这是个简单的例子，根据需要，也可以编写非常复杂的文件。例如，为了控制一个完整的机器人，如PR2或Robonaut，包括真实的机器人和在ROS中仿真的机器人。

这个文件包括launch标签，在标签内部可以看到node标签。这个node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，即本章最前面的两个例子。如果你还记得，就是节点example1_a发送消息到节点example1_b。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输出结果：



运行的节点在上面的截图中列出，可以通过下面的命令查看运行的节点：

```
$ rosnode list
```

可以看到如右图所示的三个节点：

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

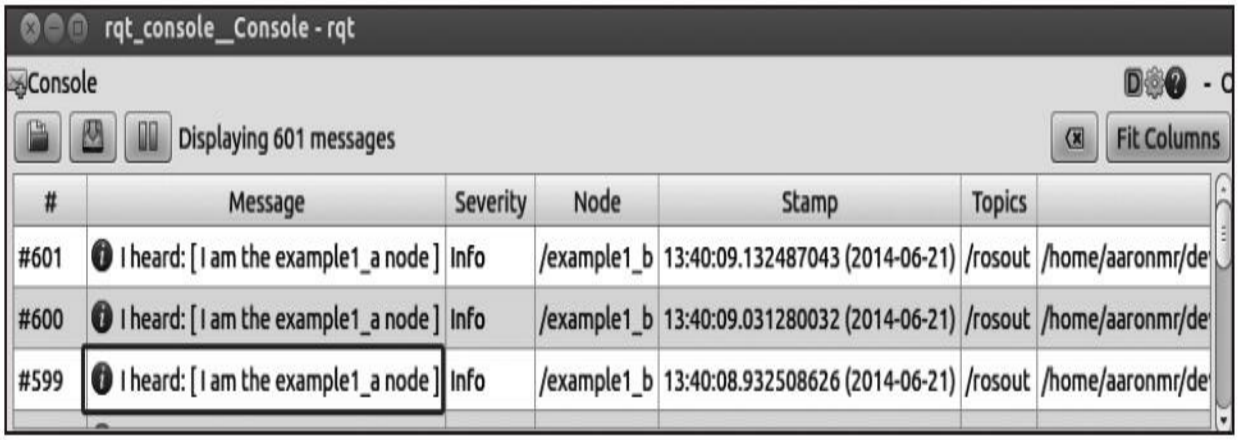
当执行启动文件时，并不需要在roscore命令前启动，roslaunch会启动它。

还记得节点example1_b会在屏幕上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ROS_INFO输出消息。当在shell中只运行一个节点时，可以看到它。但是当运行启动文件时，则看不到它。

现在，为了看到信息，可以运行rqt_console实用程序。接下来的章节详细介绍该实用程序。运行以下命令：

```
$ rqt_console
```

从下面的截图中可以看到example1_b发送的消息：



在框中，可以看到节点发送的消息以及来源文件的路径。

2.4.14 动态参数

ROS的另一个功能是动态重配置实用程序。通常情况下，当你正在编写一个新节点时，以仅你可以更改的数据初始化节点内的变量。如果你想从节点外部动态地改变这些值，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实用程序。

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文件中。

为了使用动态重配置，需要写一个配置文件并保存在包的cfg文件夹中。创建一个文件夹和新文件，如下所示：

```
$ 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, "High 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：以上代码初始化ROS并导入参数生成器：

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

from dynamic_reconfigure.parameter_generator_catkin import *
```

以上代码初始化参数生成器，通过它可以开始用下面的代码行添加参数：

```
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)
```

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：一个参数简短描述。

·default：节点启动时的默认值。

·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min和max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。本书的示例代码主要使用C++编写，但有时会使用Python代码段。

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命令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，加入下面的代码：

```
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，如下所示：

```
$ 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>::
  CallbackType 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、参数服务器以及先前创建的config文件。

callback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配置文件中的参数相同：

```
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);
}
```

初始化服务器，忽略chapter2_Config配置文件：

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

dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>::
CallbackType f;

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

现在，向服务器发送callback函数。当服务器得到重新配置请求时，它会调用callback函数。

一旦完成以上步骤，就需要在CMakeLists.txt文件中添加如下代码：

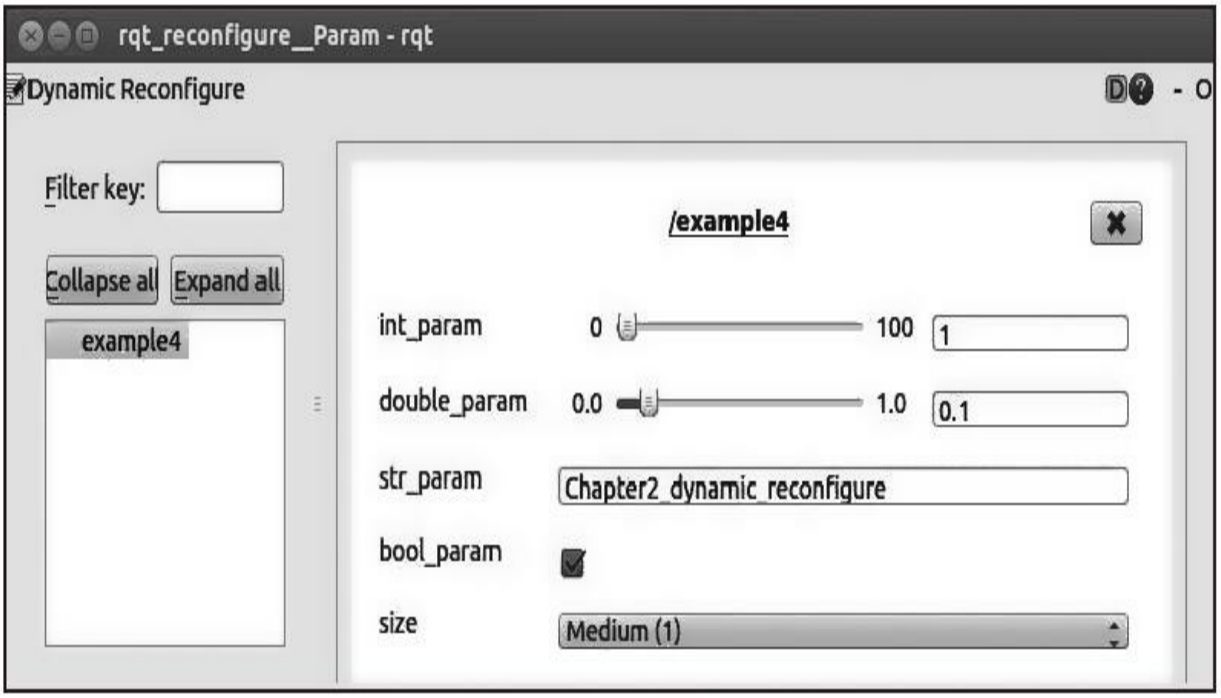
```
add_executable(example4 src/example4.cpp)

add_dependencies(example4 chapter2_tutorials_gencfg)

target_link_libraries(example4 ${catkin_LIBRARIES})
```

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

```
$ roscore
$ rosrun chapter2_tutorials example4
$ rosrun rqt_reconfigure rqt_reconfigure
```

在执行最后一行命令后，你会看到一个新窗口，通过它可以动态重

配置节点的参数，如下图所示。



每当通过滑块、复选框等调整参数时，都可以在shell中正在运行的节点中看到这些改变，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，可以更高效地开发和测试节点。配合硬件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。

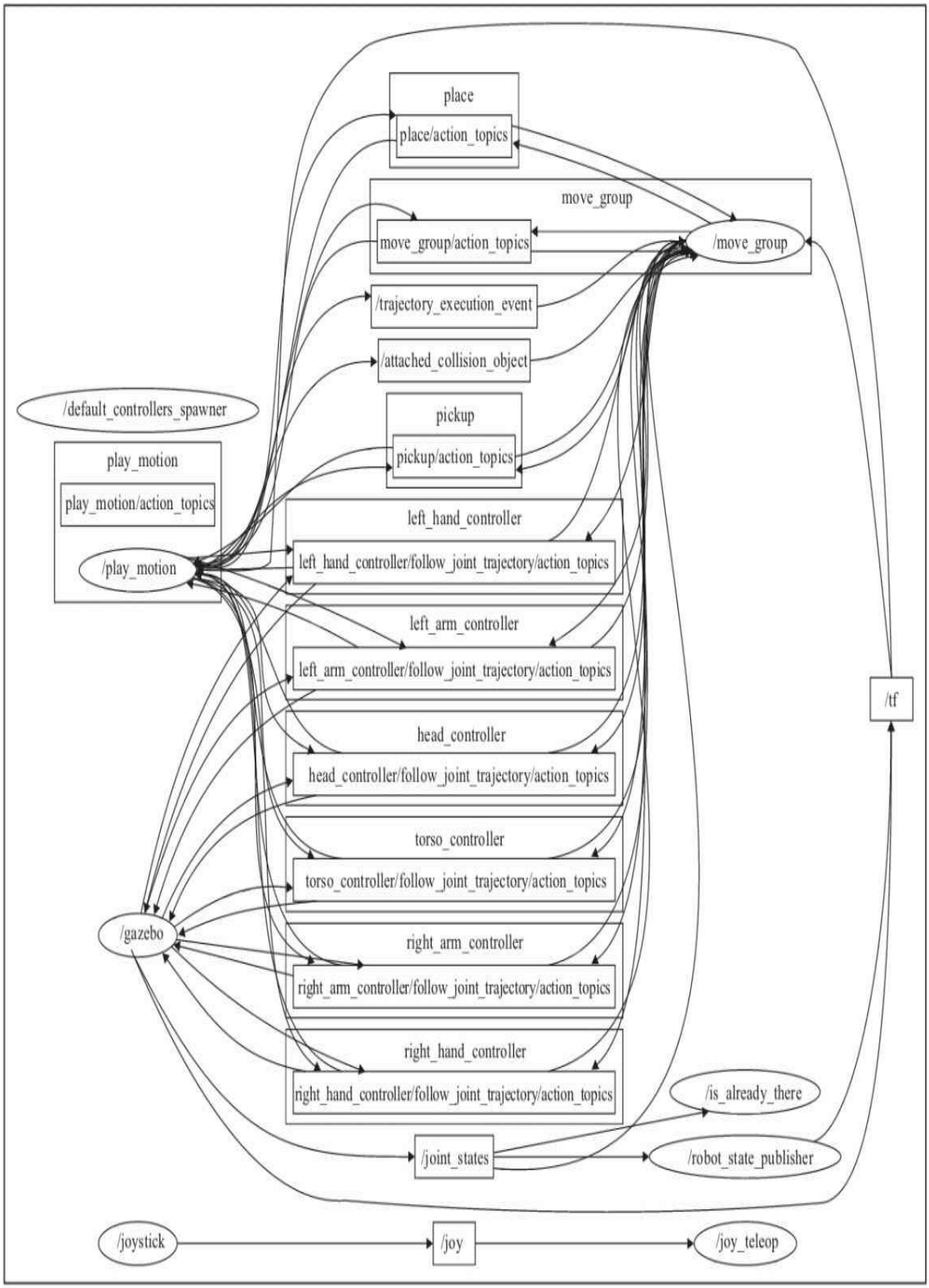
各位读者最好在继续后续章节的学习之前，对这些概念及示例进行练习，因为在后面的章节里，我们将假定你已经熟悉所有的概念及其用途。

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，了解ROS现在的操作是否正确，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

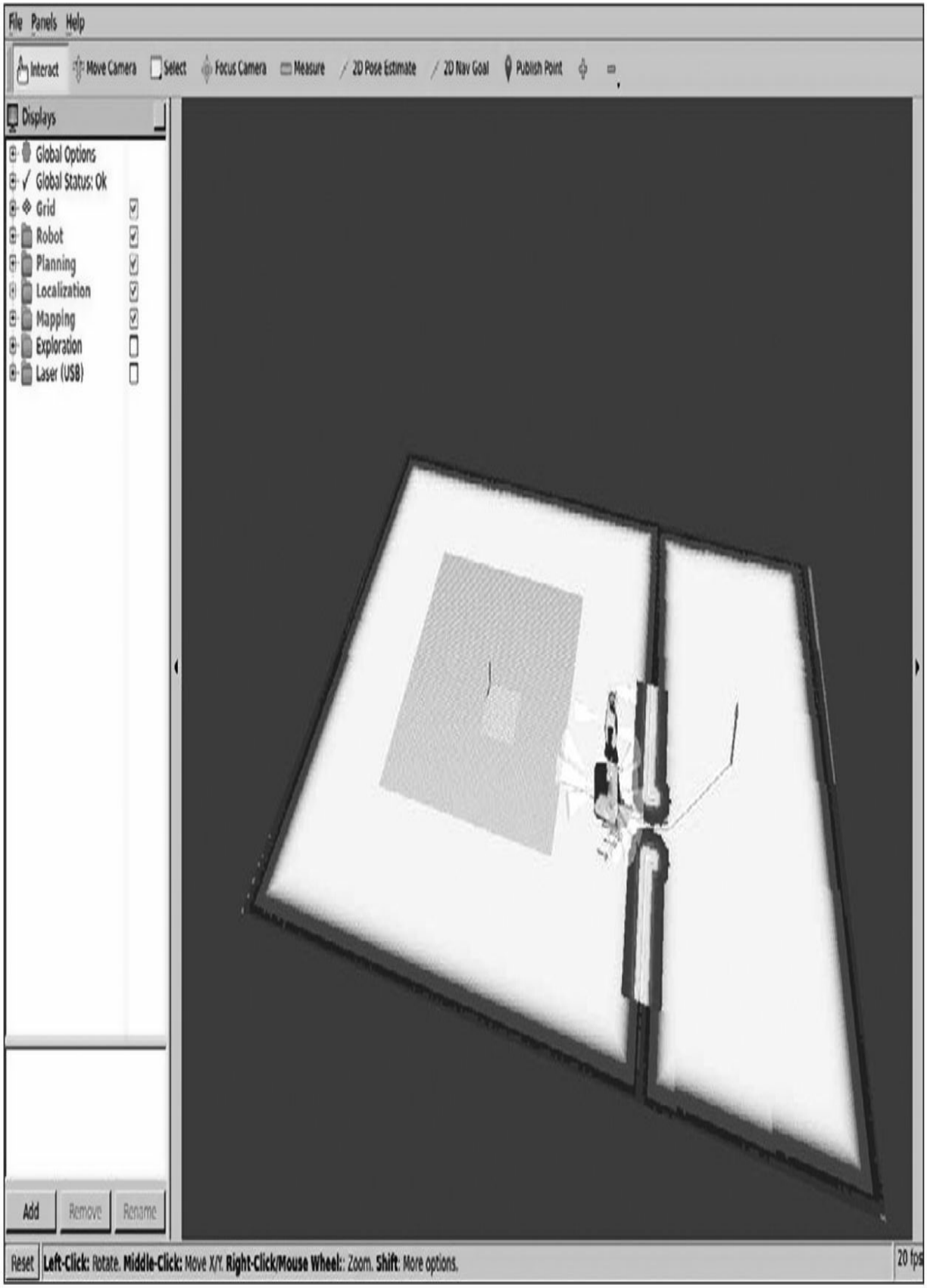
ROS附带了大量功能强大的工具，帮助用户和开发人员可视化和调试代码，以便检测并解决软硬件问题。其中包括消息日志系统（类似log4cxx）、诊断消息、可视化以及检测工具。这些工具展示了哪些节点正在运行和它们是如何互连的。


place
place/action_topics
move_group
move_group/action_topics
/move_group
/trajectory_execution_event
/attached_collision_object
/default_controllers_spawner
pickup
pickup/action_topics
play_motion
play_motion/action_topics
/play_motion
left_hand_controller
left_hand_controller/follow_joint_trajectory/action_topics
left_arm_controller
left_arm_controller/follow_joint_trajectory/action_topics
/tf
head_controller
head_controller/follow_joint_trajectory/action_topics
torso_controller
torso_controller/follow_joint_trajectory/action_topics
/gazebo
right_arm_controller
right_arm_controller/follow_joint_trajectory/action_topics
right_hand_controller
right_hand_controller/follow_joint_trajectory/action_topics
/is_already_there
/joint_states
/robot_state_publisher
/joystick
/joy
/joy_teleop

本章还会展示如何用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，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器，以及预存储动作的play_motion节点。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。


File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Global Status: Ok
Grid
Robot
Planning
Localization
Mapping
Exploration
Laser (USB)
Add
Remove
Rename
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
20 fps

可以使用以下命令运行上图的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http://wiki.ros.org/Robots/REEM网页的使用说明。

本章后续几节将介绍可视化和调试方面的内容：如何在ROS中调试代码，如何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可视化消息。介绍如何通过列表来检测ROS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。将介绍以有向图形式显示主题和节点的rqt_graph，以及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，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。还展示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。此外，还解释如何使用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，以及如何在一个GUI中排列它们。

大部分rqt工具可以通过在终端输入名称运行，例如rqt_console，但有时它不行，必须使用rosrun rqt_reconfigure rqt_reconfigure，它始终可行。注意，名字虽然是重复的，但实际上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）。可以用任何标准工具（如gdb）进行调试。同样可以用memcheck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：

```
$ rosrun chapter3_tutorials example1
```

很遗憾，不能以下面的方式通过gdb启动命令：

```
$ gdb rosrun chapter3_tutorials example1
```

接下来的几节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，也可以诊断基本和罕见问题。然后，将讨论ROS自检工具，测试节点间无效的连接。因此，本章的概述是自下而上的，实际的问题诊断方式是自上向下的。

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 Kinetic和catkin功能包中，节点的可执行文件在工作空间的devel/lib/<package>文件夹下。例如，为了在gdb中运行chapter3_tutorials功能包中的example1节点，需要按如下步骤进行，首先切换到工作空间文件夹下（/home/<user>/book_ws）：

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

如果已经运行过catkin_make install，也可使用下面的命令导航到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
```

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

一旦roscore在运行，就可以通过按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: /home/luis/devel/catkin_ws/devel/lib/chapter3_tutorials/exampl
e1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff170d700 (LWP 6618)]
[New Thread 0x7ffff0f0c700 (LWP 6619)]
[New Thread 0x7fffebfff700 (LWP 6620)]
[New Thread 0x7fffeb7fe700 (LWP 6625)]
[DEBUG] [1476313631.940149636]: This is a simple DEBUG message!
[DEBUG] [1476313631.940214159]: This is a DEBUG message with an argument: 3.1400
00
[DEBUG] [1476313631.940246937]: This is DEBUG stream message with an argument: 3
.14
[Thread 0x7fffeb7fe700 (LWP 6625) exited]
[Thread 0x7ffff170d700 (LWP 6618) exited]
[Thread 0x7ffff0f0c700 (LWP 6619) exited]
[Thread 0x7fffebfff700 (LWP 6620) exited]
[Inferior 1 (process 6613) exited normally]
(gdb)
```

3.1.2 在ROS节点启动时调用gdb调试器

在许多情况下，需要一个launch文件负责启动节点，如下：

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

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。根据需要设置断点，按C键或R键启动节点并调试。在节点崩溃时，这可以得到回溯（backtrace，bt）。

3.1.3 在ROS节点启动时调用valgrind分析节点

此外，可以使用相同的属性把节点附加到诊断工具上。例如，可以使用memcheck启动valgrind（可以访问http://valgrind.org获取详细信息）来检测程序的内存泄漏情况，并使用callgrind执行性能分析。与调用gdb的方式相反，现在无须重新启动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，在gdb会话中会用到）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-c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$ROS_HOME已有一个core目录会防止core文件转储。因此，为了创建名称和路径为$ROS_HOME/core.PID的core文件转储，必须运行如下命令：

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

3.2 日志消息

通过消息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消息输出，每条消息都有自己的名称，并根据相应条件输出消息，甚至可调。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 输出日志消息

ROS自带了大量能够输出日志消息的函数和宏。它支持如消息（或日志）级别、条件触发消息和STL流、可调等诸多功能。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（在这里定义日志记录的API）：

```
#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.
```

所有输出的消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），这两个值放在实际消息之前的方括号中。时间戳以公历时间计时，代表自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，可以按照以下代码输出变量val对应的浮点数值：

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

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

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

请注意，我们没有指定任何流，因为API通过重定向到cout/cerr、一个文件或这两者完成这些工作。

3.2.2 设置调试消息级别

ROS有5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）

这些名称是输出消息的函数的一部分，它们遵循以下语法：

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

每一种消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
INFO in white
WARN in yellow
ERROR in red
FATAL in purple
```

每个消息级别用于不同的目的。在这里，有以下建议。

·DEBUG：只在调试时用，此信息不应出现在部署的应用中，仅用于测试目的。

·INFO：应有的标准消息，说明重要步骤或节点所正在执行的操作。

·WARN：提醒一些错误、缺失或者不正常，但程序仍能运行。

·ERROR：提示错误，尽管节点仍然可以在这里恢复，但它们对节点的行为设置了一定的期望。

·FATAL：这些消息通常表示阻止节点继续运行的错误。

3.2.3 为特定节点配置调试消息级别

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。其中有些在编译时设定，有些消息甚至在没有给定级别时进行编译，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工具来实现这一功能。

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。

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或更高级别的调试消息，在源代码中加入下面的代码：

```
#define ROSCONSOLE_MIN_SEVERITY ROSCONSOLE_SEVERITY_ERROR
```

或者，在CMakeLists.txt中使用以下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的文件夹和一个名为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，设置ROSCONSOLE_CONFIG_FILE环境变量指向我们的文件。可以使用一个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，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 消息命名

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，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函数如下面的代码所示（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，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消息，如下列代码所示：

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

3.2.5 按条件显示消息与过滤消息

按条件显示（conditional）消息是指仅当满足给定的条件时才输出的消息。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消息。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）消息在本质上与按条件显示消息类似，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。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：

```
structMyLowerFilter : public ros::console::FilterBase {
  MyLowerFilter(const double&val ) : value( val ) {}
  inline virtual boolisEnabled() { return value < 0.; }
  double value;
  };

MyLowerFilterfilter_lower(val );

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

3.2.6 显示消息的方式——单次、可调以及其他组合

可以控制消息的显示次数。通过使用ROS_<LEVEL>[_STREAM]_ONCE[_NAMED]可以让消息只输出一次。

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

这段代码也来自example2节点，只会显示消息1次。

然而，有时候在迭代中以一定频率显示消息更好。这就需要可调消息。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，它将会每间隔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消息，命名消息、按条件显示消息、单次/可调消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消息与其他节点的提示消息区别开。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开头的ROS_*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。

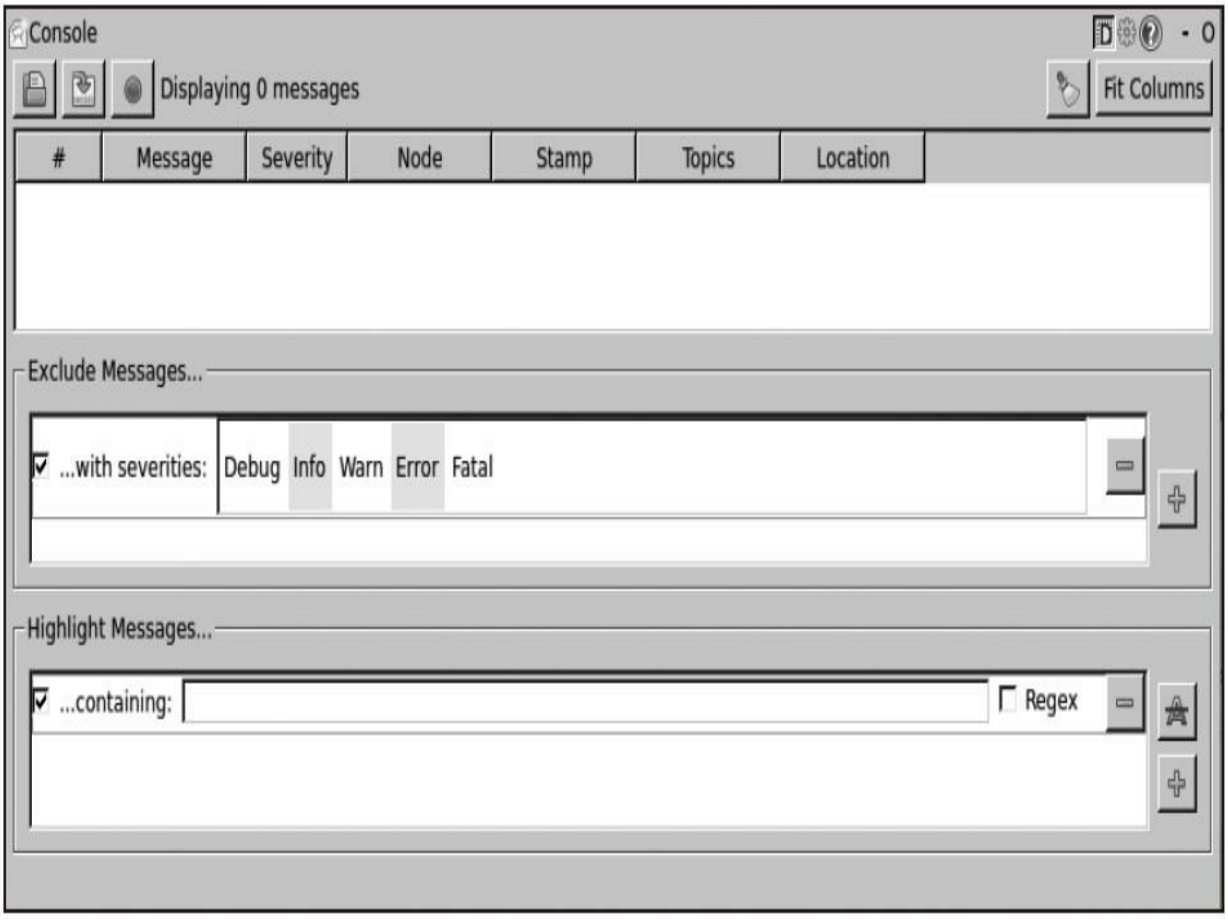
3.2.7 使用rqt_console和rqt_logger_level在运行时修改调试级别

ROS为管理日志信息提供了一系列工具。在ROS Kinetic中有两个独立的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对日志消息进行可视化、过滤和分析。

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

```
$ rosrun rqt_console rqt_console
```

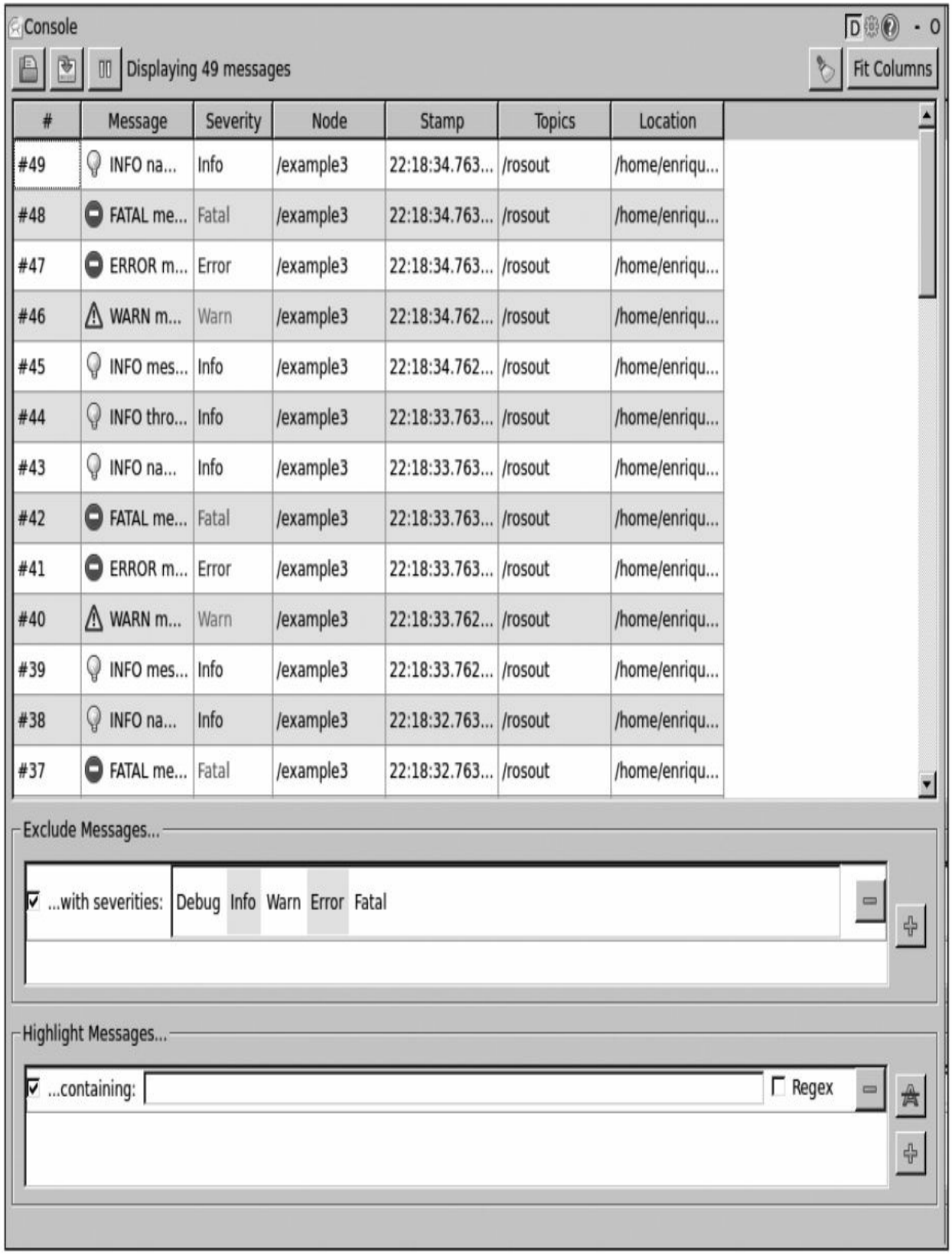
将看到如下窗口：



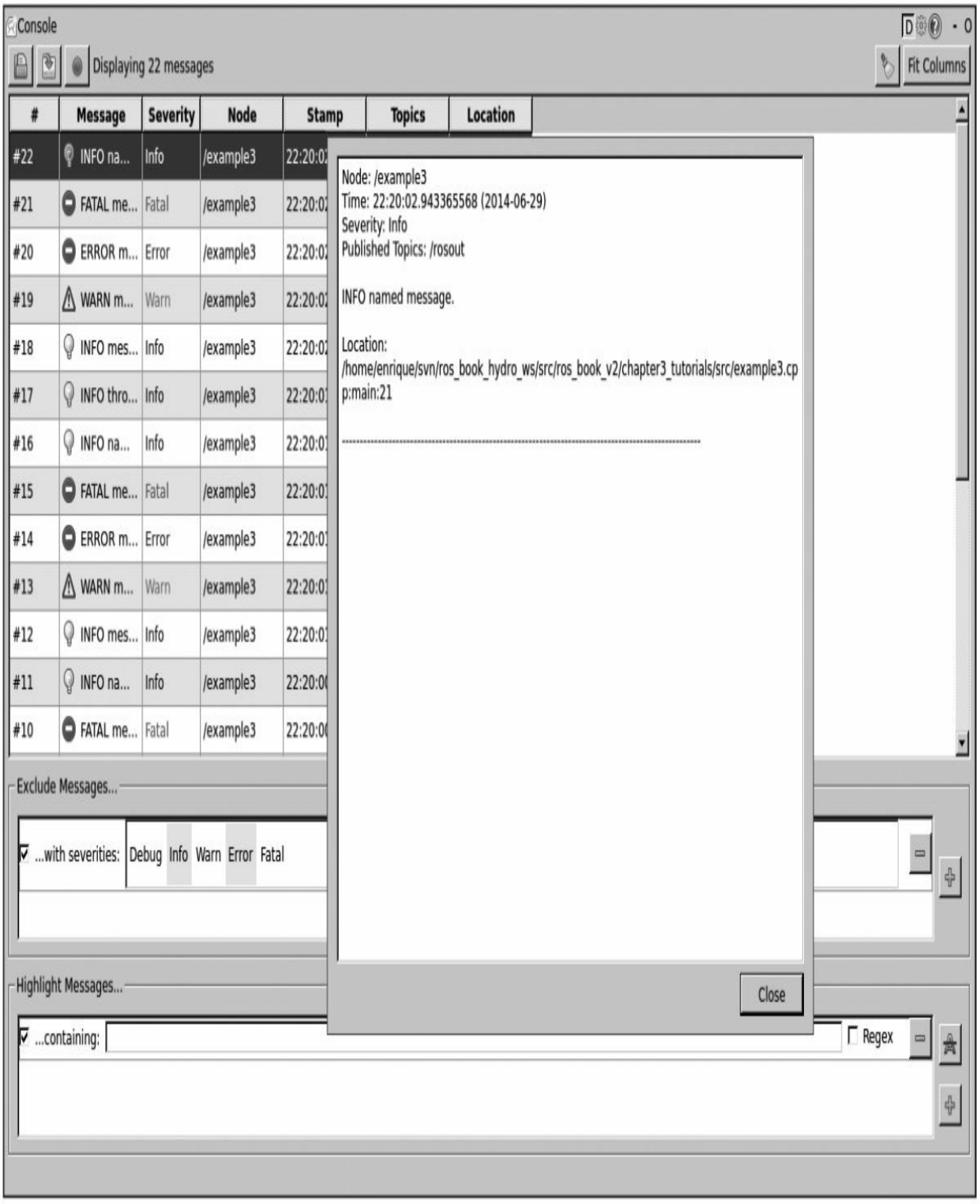
运行下面的节点：

```
$ rosrun chapter3_tutorials example3
```

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

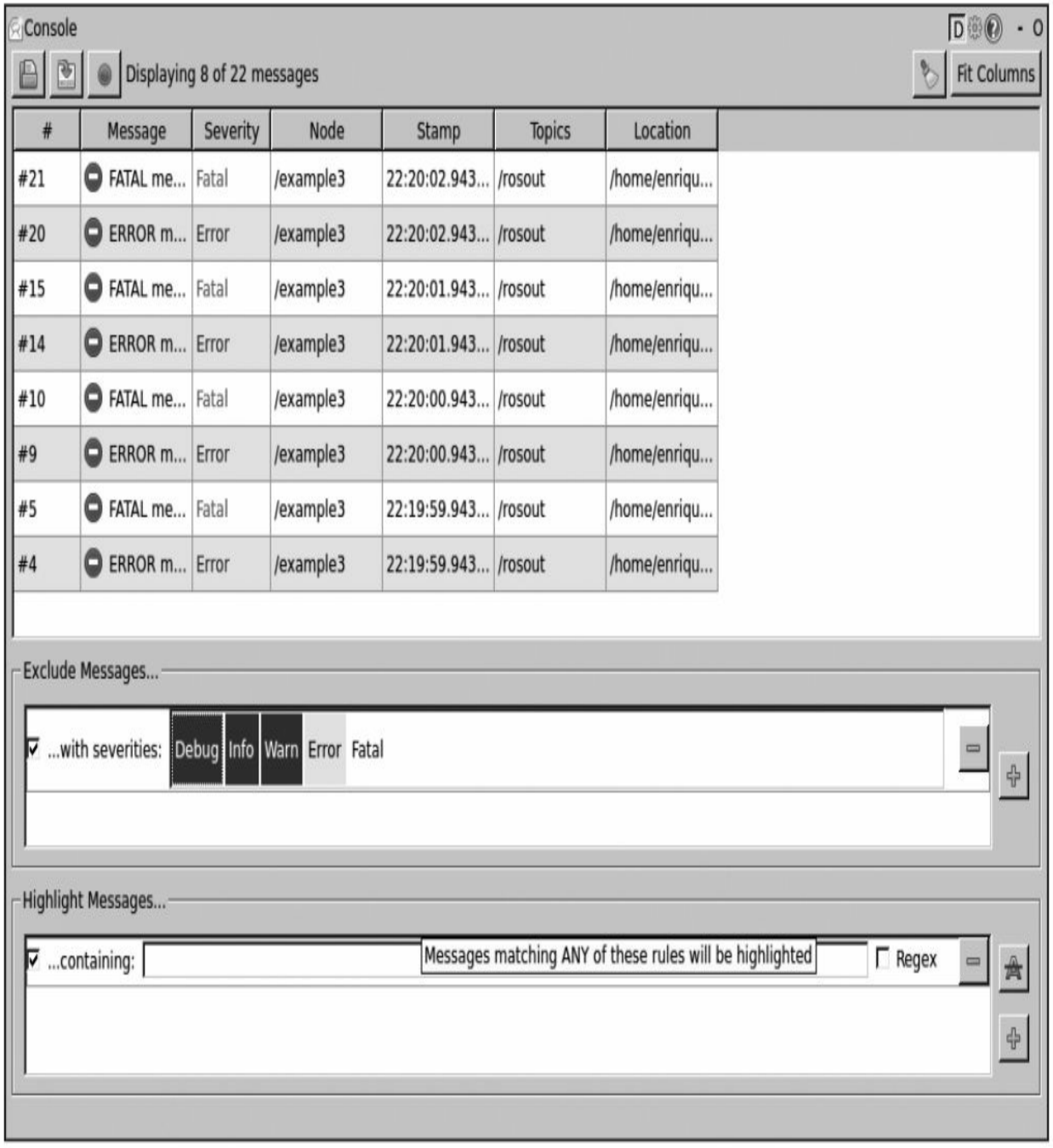

Console
Displaying 49 messages
Fit Columns
Message Severity Node Stamp Topics Location
#49 INFO na... Info /example3 22:18:34.763... /rosout /home/enriqu...
#48 FATAL me... Fatal /example3 22:18:34.763... /rosout /home/enriqu...
#47 ERROR m... Error /example3 22:18:34.763... /rosout /home/enriqu...
#46 WARN m... Warn /example3 22:18:34.762... /rosout /home/enriqu...
#45 INFO mes... Info /example3 22:18:34.762... /rosout /home/enriqu...
#44 INFO thro... Info /example3 22:18:33.763... /rosout /home/enriqu...
#43 INFO na... Info /example3 22:18:33.763... /rosout /home/enriqu...
#42 FATAL me... Fatal /example3 22:18:33.763... /rosout /home/enriqu...
#41 ERROR m... Error /example3 22:18:33.763... /rosout /home/enriqu...
#40 WARN m... Warn /example3 22:18:33.762... /rosout /home/enriqu...
#39 INFO mes... Info /example3 22:18:33.762... /rosout /home/enriqu...
#38 INFO na... Info /example3 22:18:32.763... /rosout /home/enriqu...
#37 FATAL me... Fatal /example3 22:18:32.763... /rosout /home/enriqu...
Exclude Messages...
...with severities: Debug Info Warn Error Fatal
Highlight Messages...
...containing:
Regex

在rqt_console下，消息按类别进行收集和显示，如通过时间戳、消息类型、严重级别以及产生这些消息的节点等。可以通过单击Resize columns按钮自动调整格式。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。



交互窗口可以暂停、保存、读取和载入保存的日志消息。我们可以清理消息列表并过滤它们。在ROS Kinetic下，除了过滤掉的消息外，消

息有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，可以设置突出显示的过滤器，如下图所示。

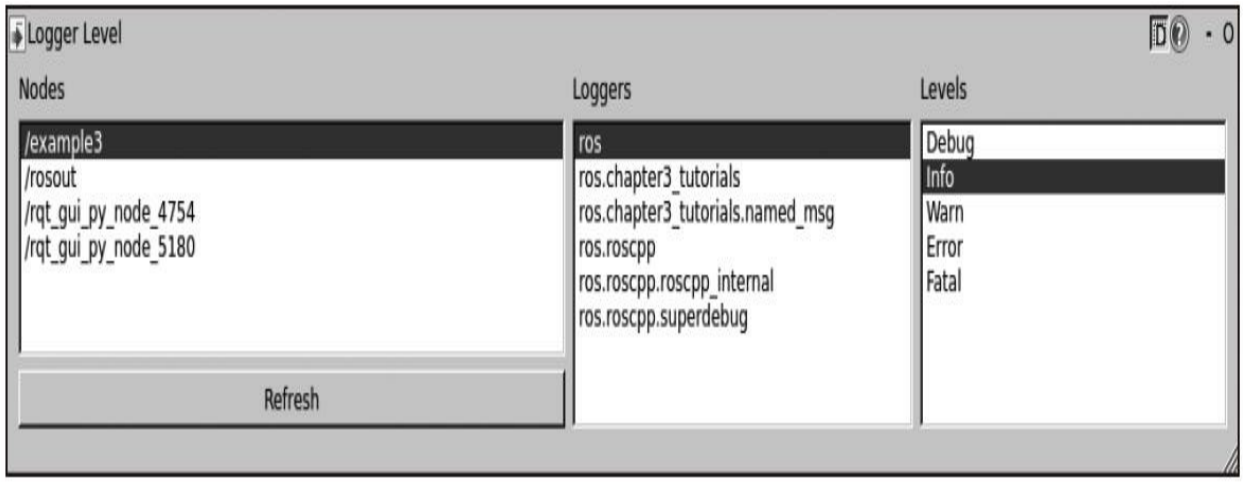


举一个例子，上图通过排除法仅显示FATAL和ERROR级别的消息。

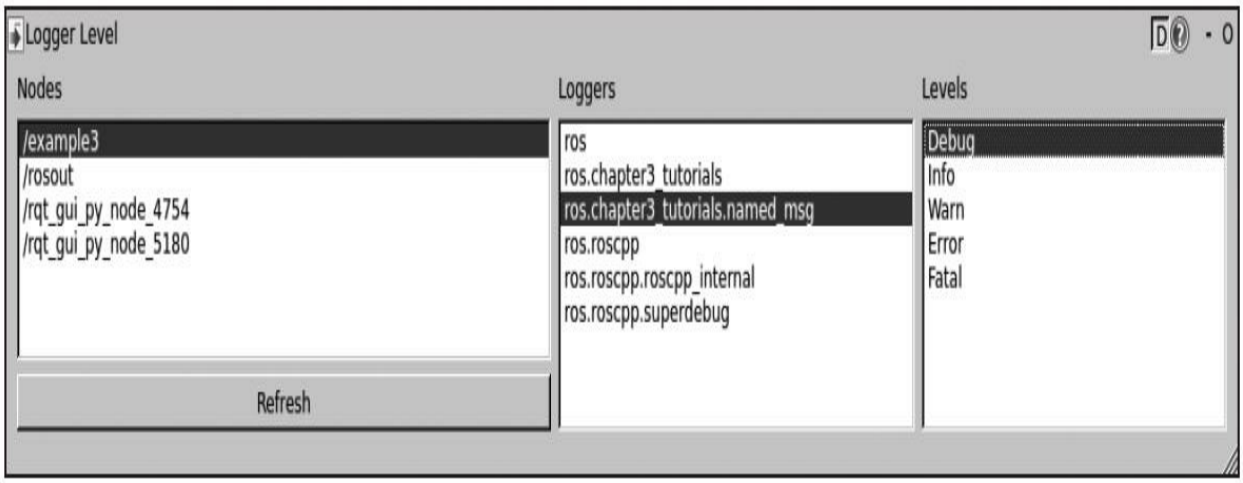
为了设置日志记录器的严重级别，必须运行以下命令：

```
$ rosrun rqt_logger_level rqt_logger_level
```

在这里，可以选择节点，然后指定日志记录器，最后是严重级别。一旦修改它，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，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，这些内部日志记录器与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之间发布消息。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的应用。

为了检测节点、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。

获得完整的列表	命　　令
正在运行的节点	`rosnode list`
所有正在运行节点的主题	`rostopic list`
所有正在运行节点的服务	`rosservice list`
服务器上的参数	`rosparam list`

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rosmsg show获取特定主题和字段发出的消息类型。

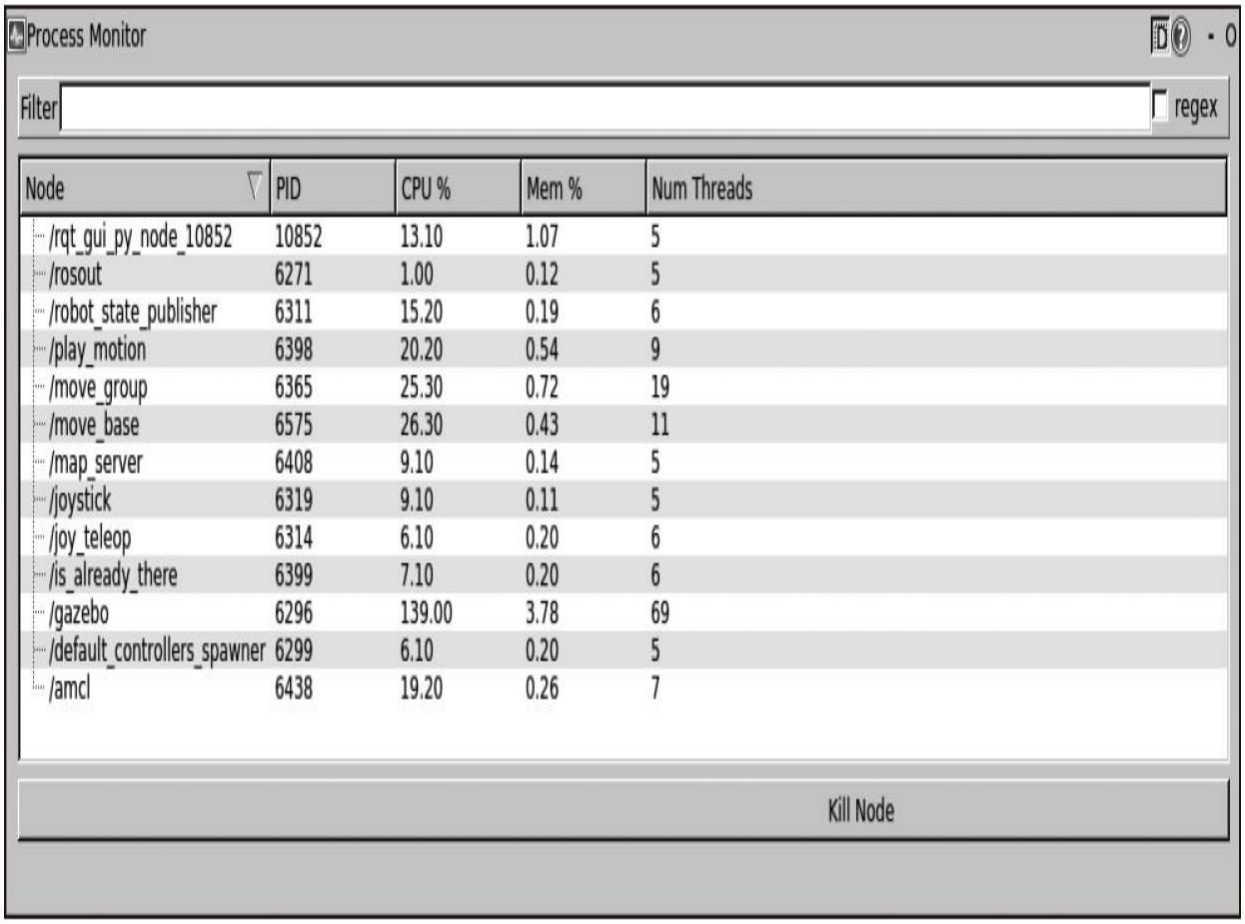
所有这些命令可以结合bash命令（如grep）寻找所需的节点、主题、服务或参数。例如，可以使用以下命令启动目标主题：

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

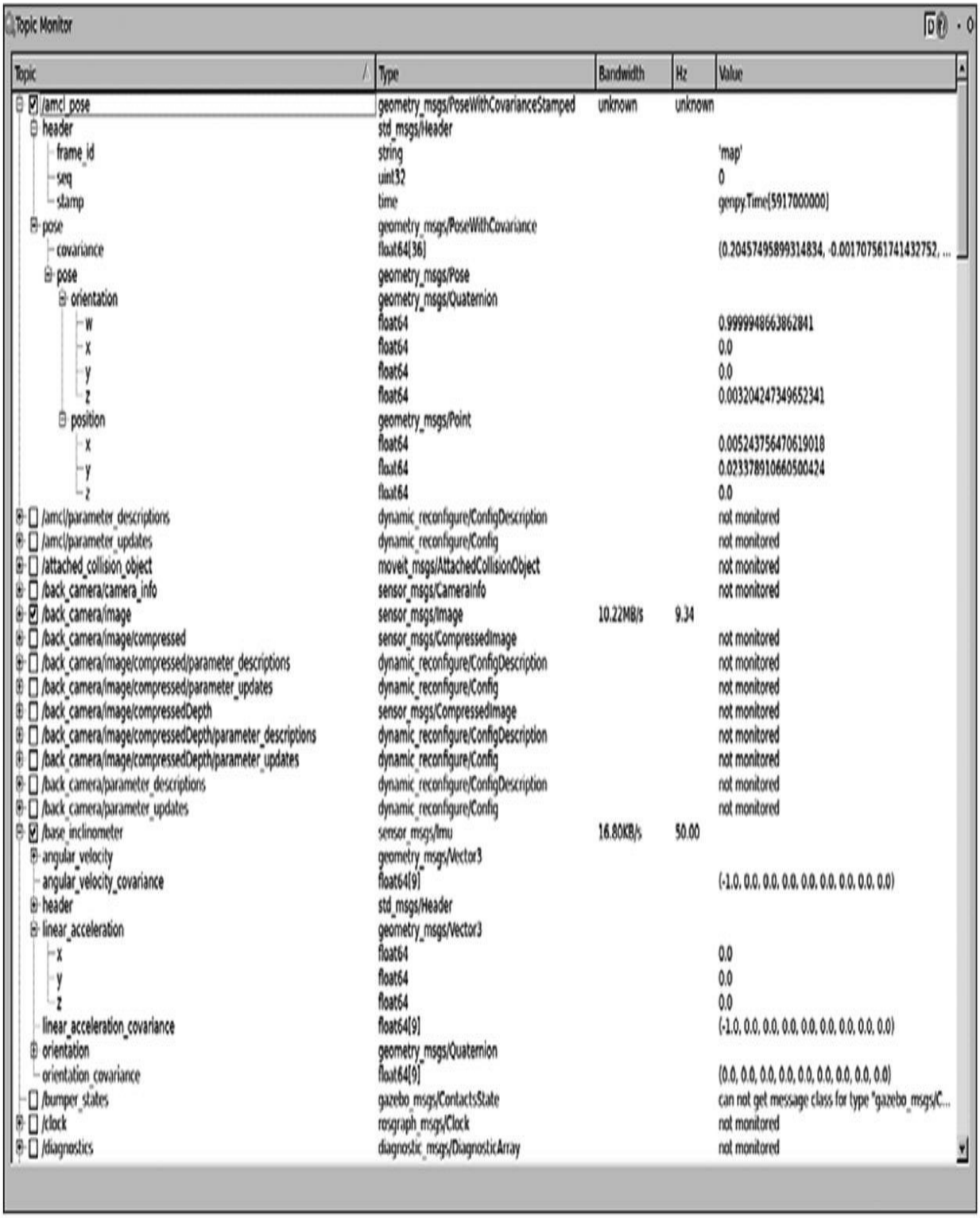
bash命令grep在文件列表或标准输出中查找文本或模式，如本例所示。

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）的窗口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。以运行的导航功能包集的REEM仿真为例，如下图所示。

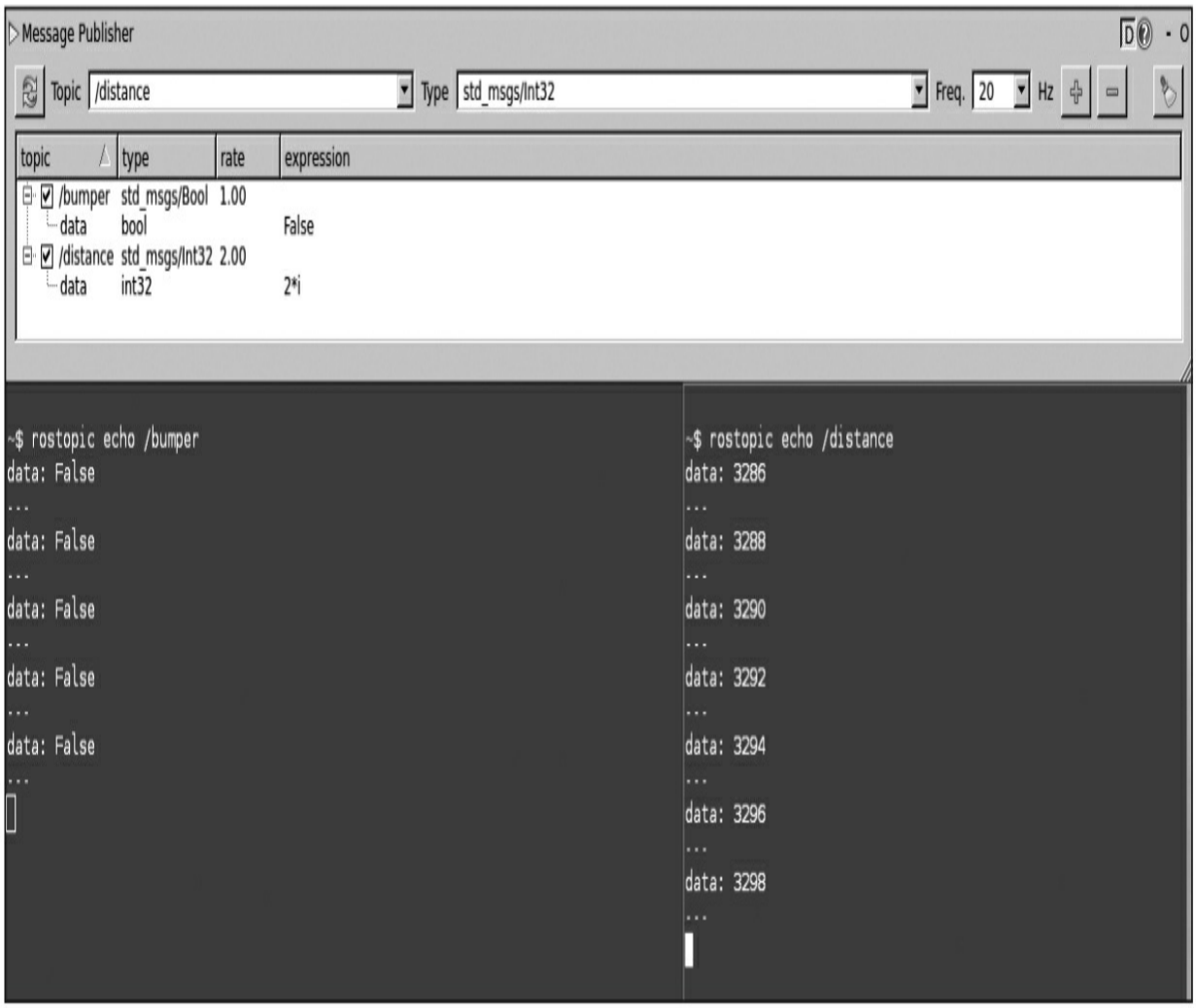


另一方面，rqt_topic显示主题调试信息，包括发布者、接收者、发布速率和发布的消息。可以查看消息字段并选择你想要订阅的主题以分析带宽和速率（Hz），以及查看最新发布的消息。注意，锁定的主题通常不会持续发布，所以不会看到任何关于它们的信息，如下图所示。

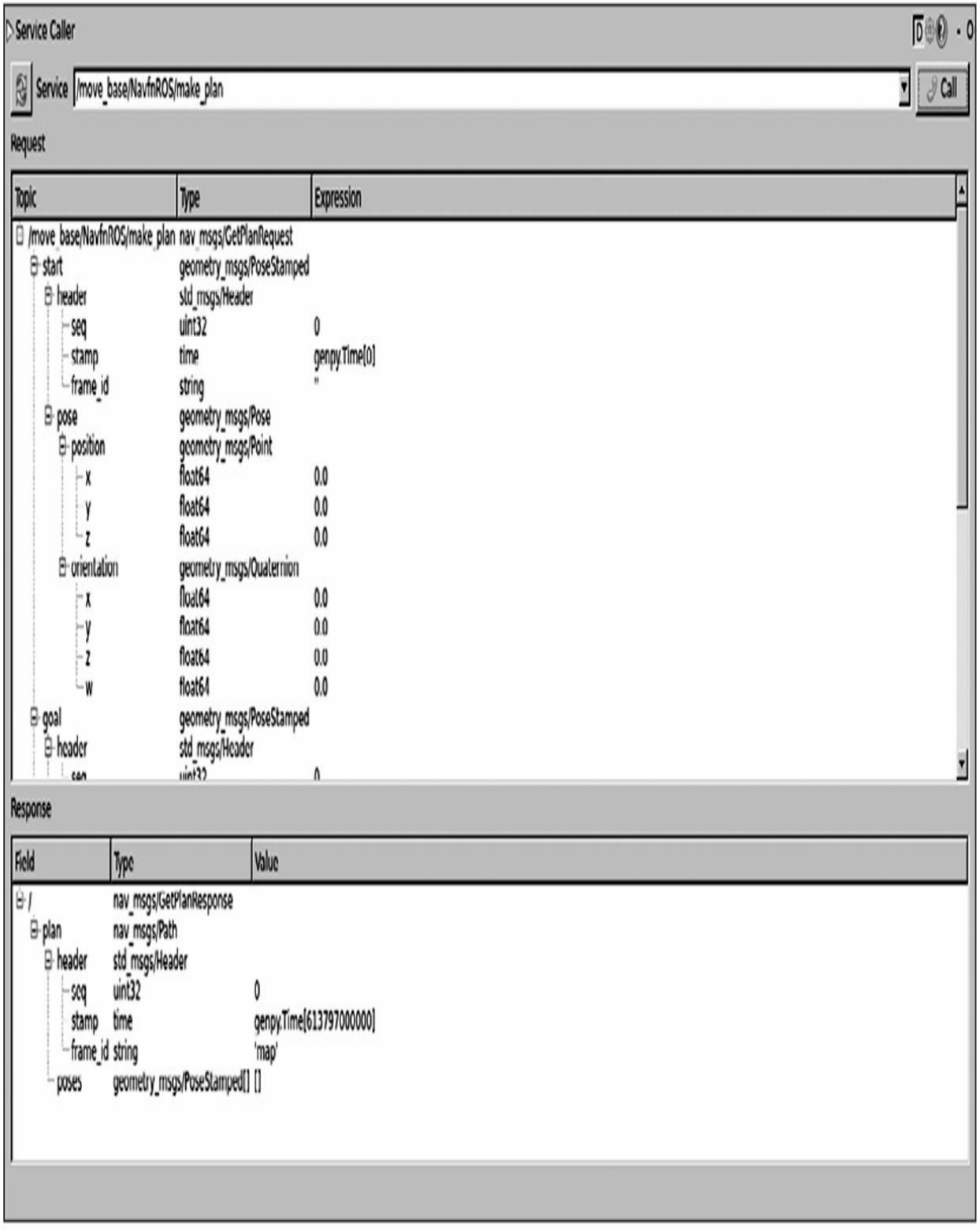


同样，rqt_publisher允许我们在一个界面中管理rostopic　pub命令的多个实例。它还支持Python发布的消息和固定值的表达式。在下图中，

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）。



注意，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。单击Call按钮之后，我们将获得响应消息。对于本例，我们使用运行的导航功能包集的REEM仿真，如下图所示。


Service Caller
Service
/move_base/NavfnROS/make_plan
Call
Request
Topic
Type
Expression
/move_base/NavfnROS/make_plan nav_msgs/GetPlanRequest
start geometry_msgs/PoseStamped
header std_msgs/Header
seq uint32 0
stamp time genpy.Time[0]
frame_id string ''
pose geometry_msgs/Pose
position geometry_msgs/Point
x float64 0.0
y float64 0.0
z float64 0.0
orientation geometry_msgs/Quaternion
x float64 0.0
y float64 0.0
z float64 0.0
w float64 0.0
goal geometry_msgs/PoseStamped
header std_msgs/Header
Response
Field
Type
Value
/ nav_msgs/GetPlanResponse
plan nav_msgs/Path
header std_msgs/Header
seq uint32 0
stamp time genpy.Time[613797000000]
frame_id string 'map'
poses geometry_msgs/PoseStamped[] []

使用rqt_graph在线检测节点的状态图

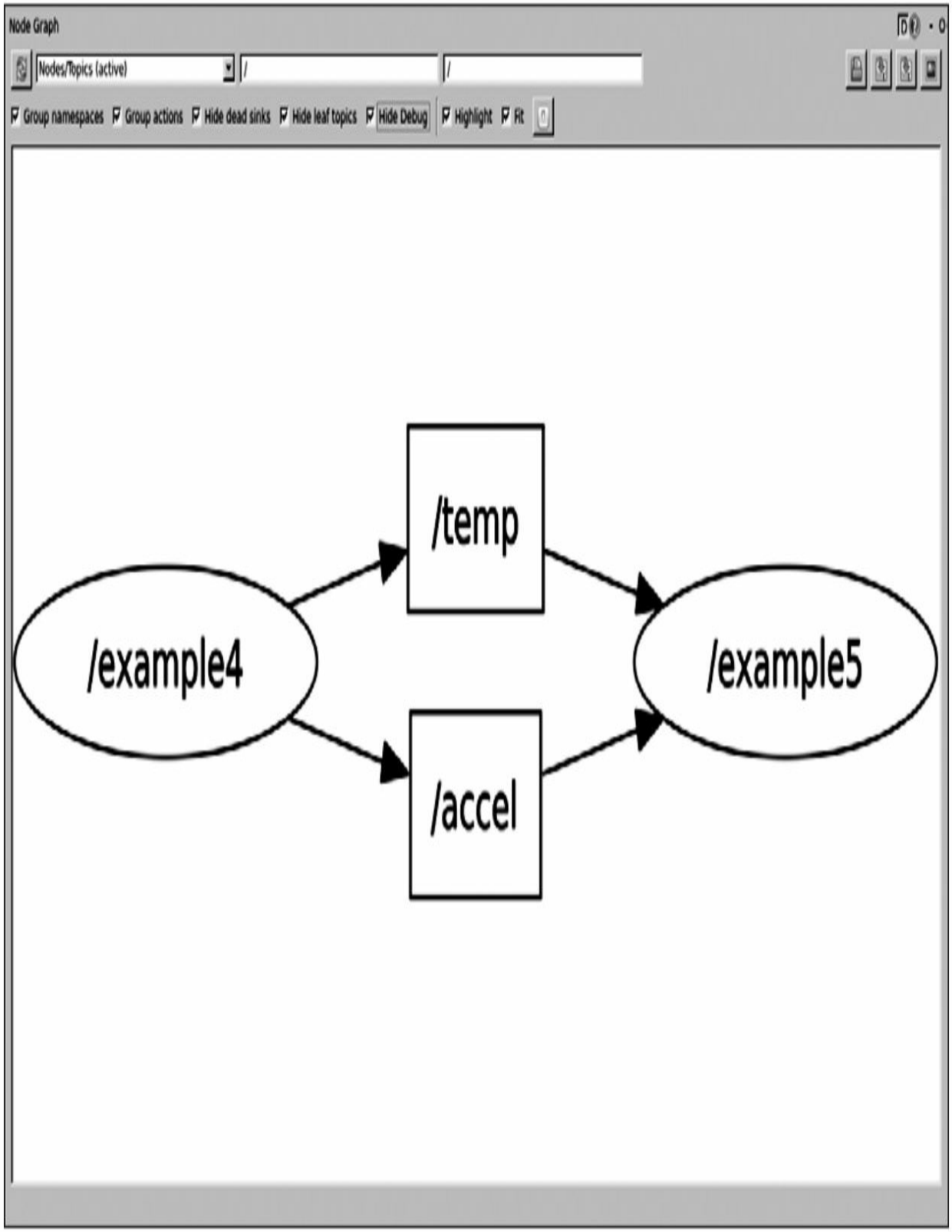
可以用有向图来显示ROS会话的当前状态，其中运行的节点是图中的节点，边为发布者-订阅者在这些节点与主题间的连接。此图形由rqt_graph动态绘制。

```
$ 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：

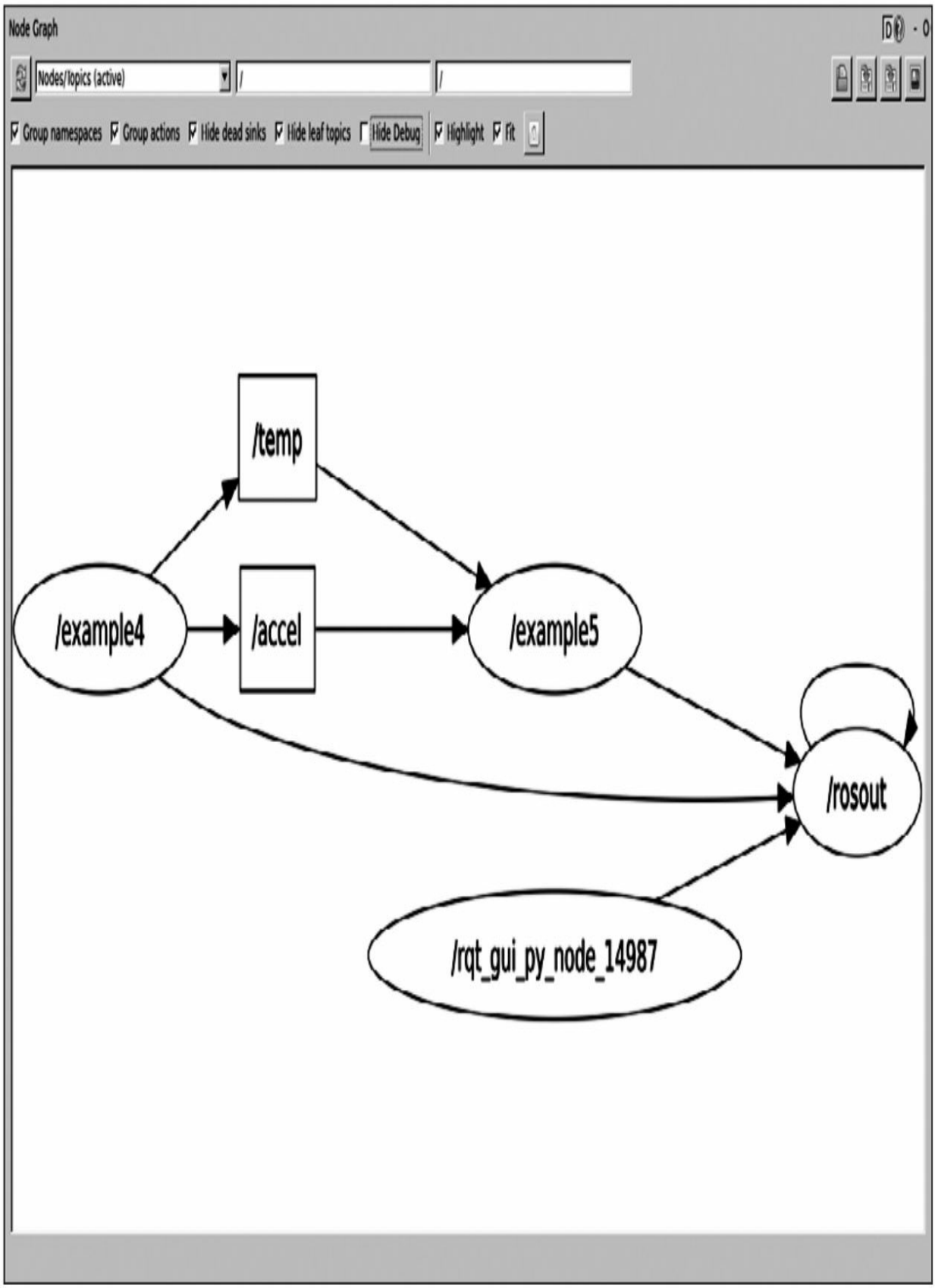
```
$ roslaunch chapter3_tutorials example4_5.launch
```

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temp和accel主题相连。由于勾选了

Hide Debug复选框，因此我们不会看到ROS服务器节点rosout以及rosout主题发布到诊断聚合器（diagnostic aggregator）上的日志消息，如我们之前做的那样。取消勾选这个选项就可以显示该节点和主题的调试消息，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，因为图可以通过隐藏节点简化。同时，ROS Kinetic中相同命名空间的节点被分组，例如图像管道的节点。


Node Graph
Nodes/topics (active)
/
/
Group namespaces
Group actions
Hide dead sinks
Hide leaf topics
Hide Debug
Highlight
Fit
/temp
/example4
/accel
/example5
/rosout
/rqt_gui_py_node_14987

当系统出现问题时，节点会一直（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，可以启用统计功能查看在主题边上显示的消息速率、带宽，以及写入速率和行速率。必须在运行rqt_graph之前设置参数来使信息有效：

```
$ rosparam set enable_statistics true
```

但是这里不仅需要启用统计功能。为了能得到需要的信息，rqt_topic应用需要启动并勾选适当的主题。还应注意，在启动相关节点之前，可能需要启用该参数。

3.4 设置动态参数

如果一个节点实现了一个动态重配置参数服务器，在工作中就可以使用rqt_reconfigure立即进行修改。运行下面的代码，它实现一个带有几个参数的动态重配置服务器（见功能包的cfg文件夹中的cfg文件）：

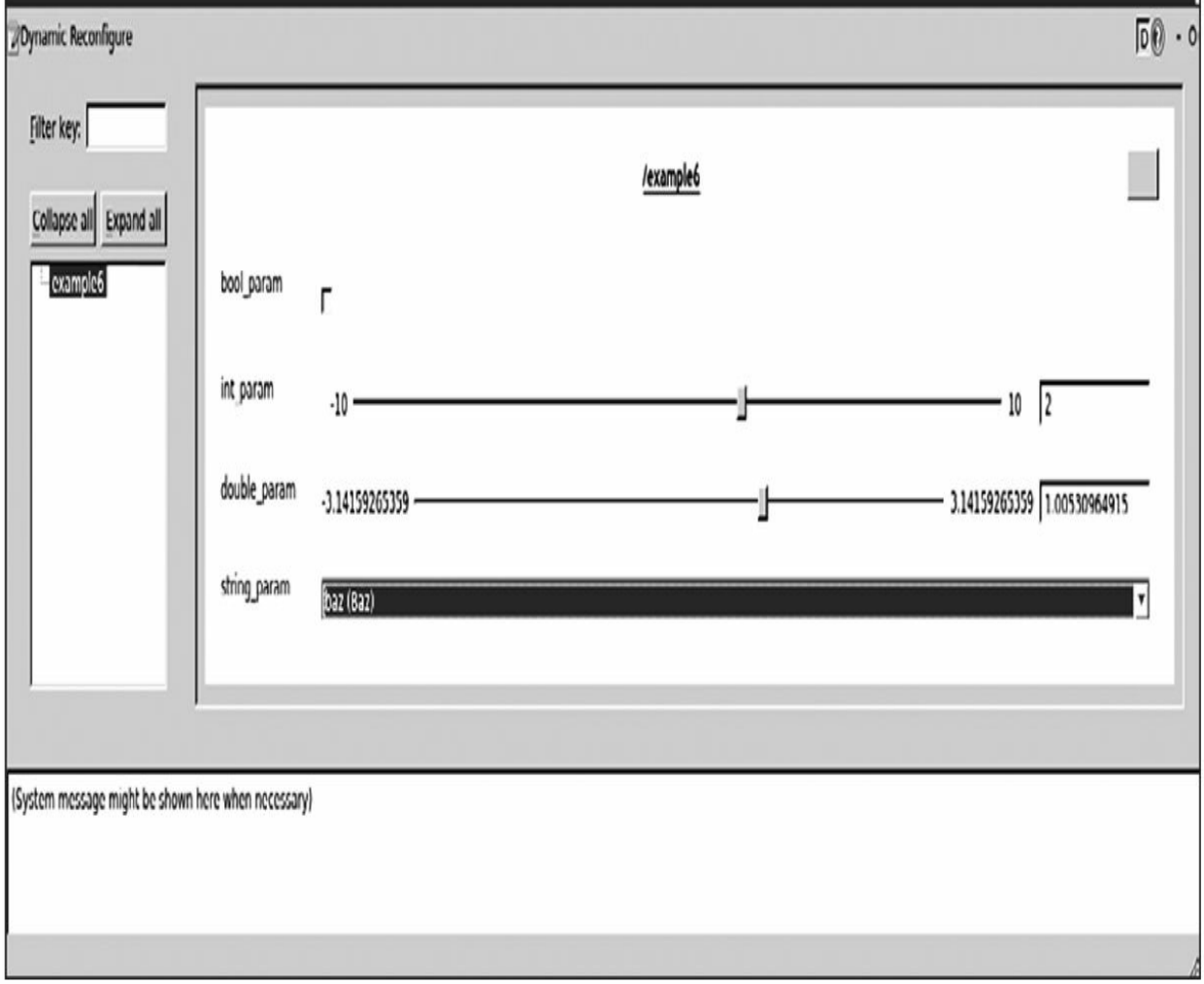
```
$ roslaunch chapter3_tutorials example6.launch
```

使用下面的命令启动动态重配置服务器，打开GUI：

```
$ rosrun 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。这些参数更改会立刻生效，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。

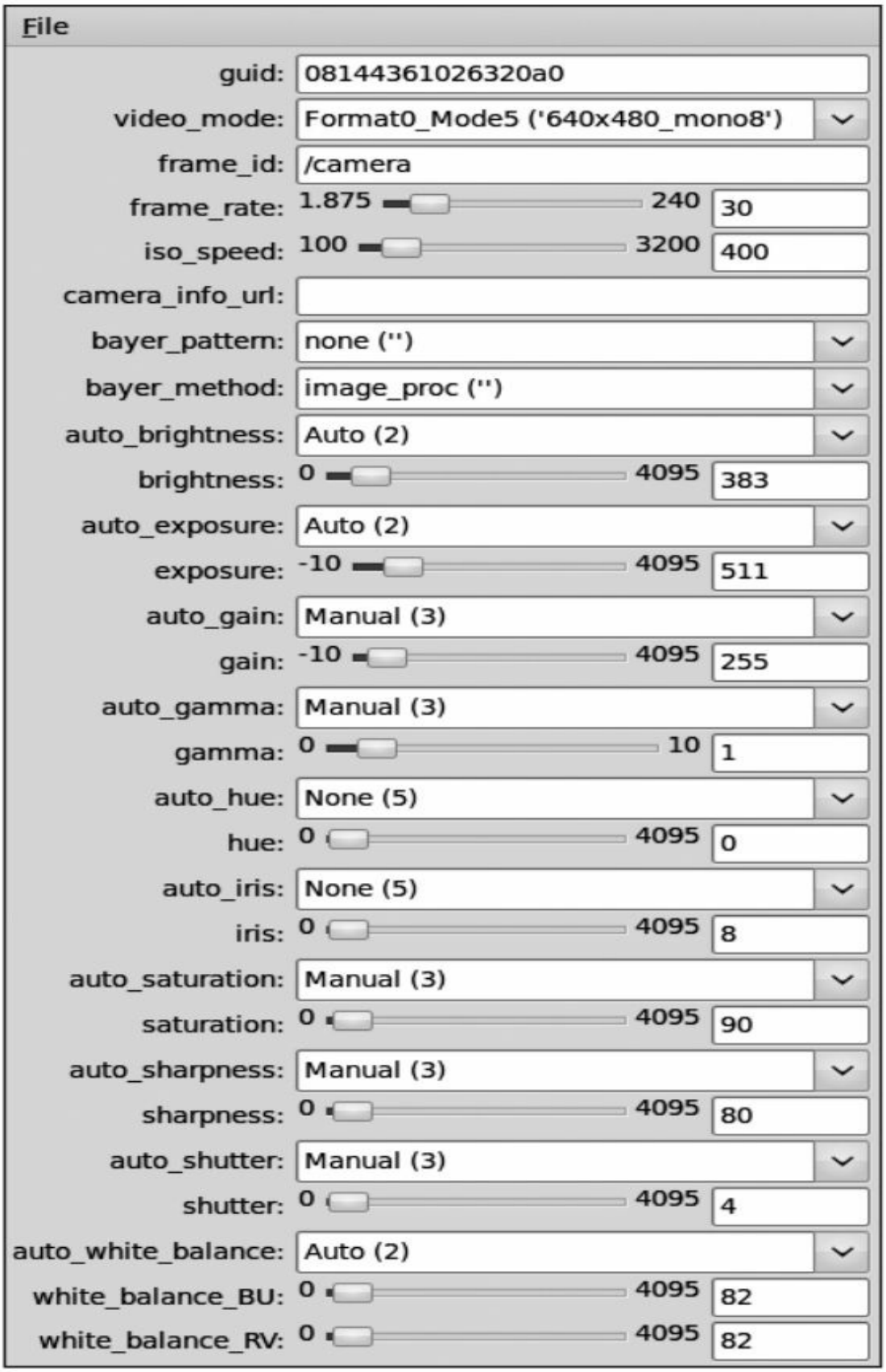
```
string = Foo
[ INFO] [1404155692.740704020]: New configuration received with level = 3:
bool   = 0
int    = 2
double = 1.00531
string = Foo
[ INFO] [1404155699.438546902]: New configuration received with level = 0:
bool   = 0
int    = 2
double = 1.00531
string = Bar
[ INFO] [1404155713.895218175]: New configuration received with level = 0:
bool   = 0
int    = 2
double = 1.00531
string = Baz
```



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程序上得以应用。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有Hokuyo激光测距仪的hokuyo_node驱动程序或FireWire camera1394驱动程序。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摄像头驱动程序（IEEE 1394，a和b）：

```
$ rosrun camera1394 camera1394_node
```

当摄像头运行时，可以用rqt_reconfigure配置其参数，将会看到类似下图的界面。


File
guid:
08144361026320a0
video_mode:
Format0_Mode5 ('640x480_mono8')
frame_id:
/camera
frame_rate:
1.875
240
30
iso_speed:
100
3200
400
camera_info_url:
bayer_pattern:
none ('')
bayer_method:
image_proc ('')
auto_brightness:
Auto (2)
brightness:
0
4095
383
auto_exposure:
Auto (2)
exposure:
-10
4095
511
auto_gain:
Manual (3)
gain:
-10
4095
255
auto_gamma:
Manual (3)
gamma:
0
10
1
auto_hue:
None (5)
hue:
0
4095
0
auto_iris:
None (5)
iris:
0
4095
8
auto_saturation:
Manual (3)
saturation:
0
4095
90
auto_sharpness:
Manual (3)
sharpness:
0
4095
80
auto_shutter:
Manual (3)
shutter:
0
4095
4
auto_white_balance:
Auto (2)
white_balance_BU:
0
4095
82
white_balance_RV:
0
4095
82

请注意，第9章将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

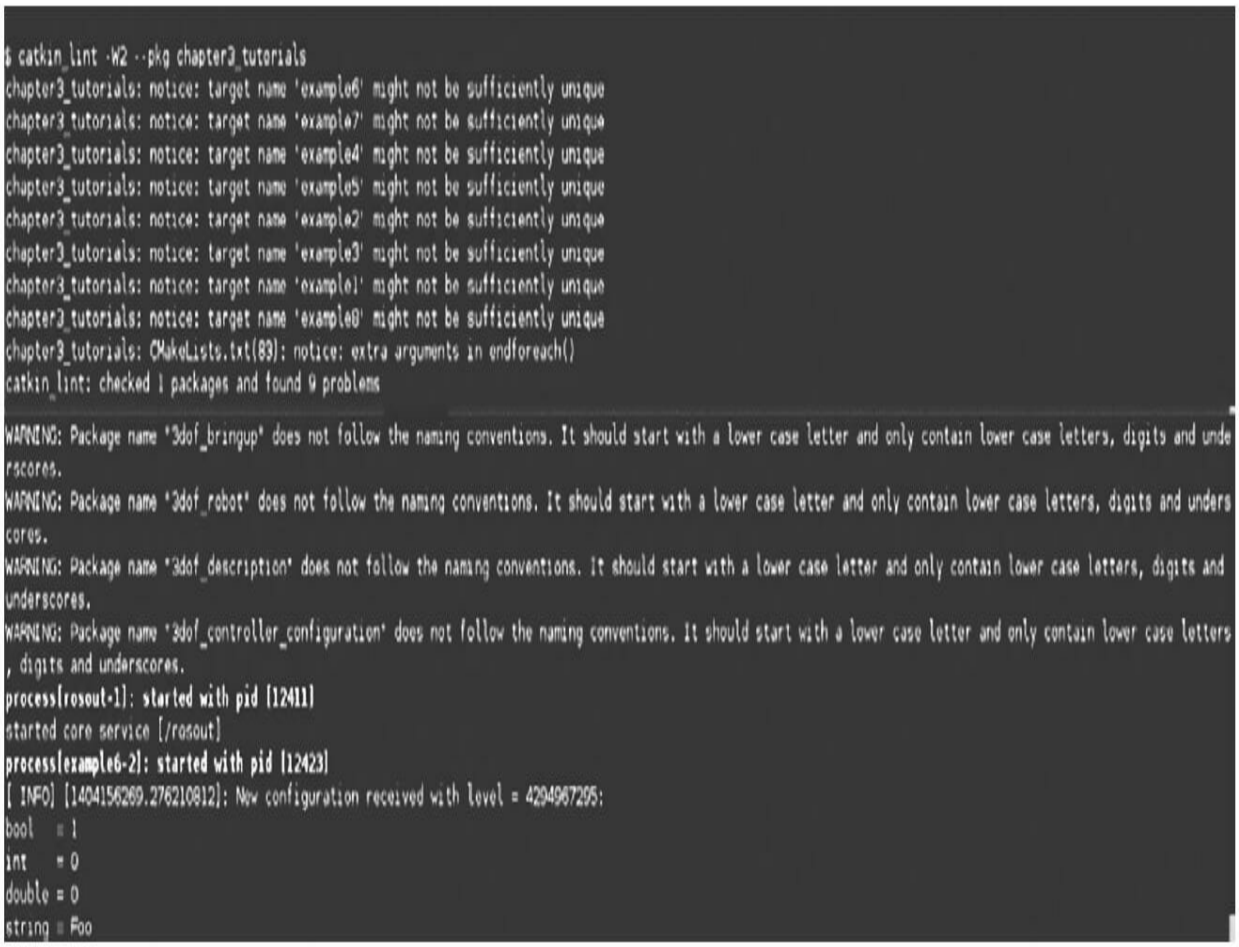
ROS还提供了另外一些工具来检测给定功能包中所有元件的潜在问题。使用roscd移动到你想要分析的功能包路径下，然后运行roswtf。对于chapter3_tutorials，可以获得以下输出。请注意，如果你运行了一些代码，也可以用ROS图进行分析。运行roslaunch chapter3_tutorials example6.launch命令，输出如下图所示。

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潜在问题，然后就像上面的示例一样，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，它可以帮助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。



请注意，可能需要单独安装catkinlint。它通常包含在python-catkin-lint包中。

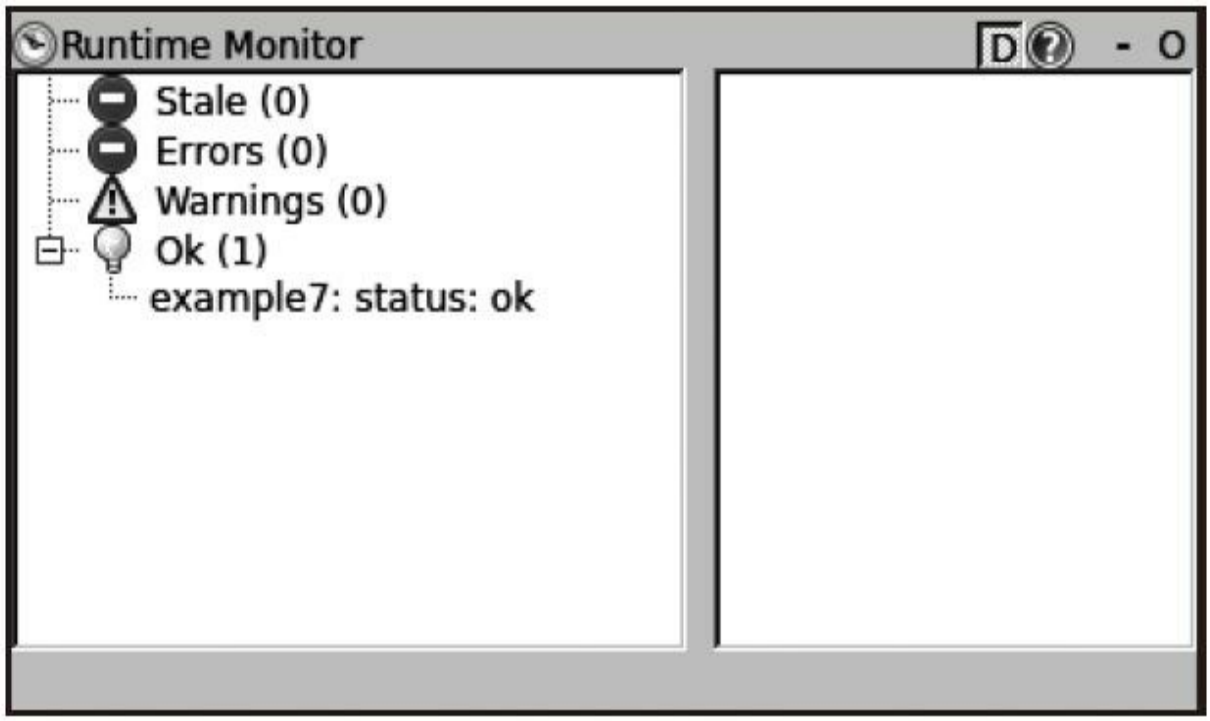
3.6 可视化节点诊断

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/DiagnositcStatus的消息类型，它允许我们指定一个级别（OK、WARN、ERROR）、名称、消息和硬件ID以及diagnostic_msgs/KeyValue列表，该列表对应成对的键（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

$ rosrun rqt_runtime_monitor rqt_runtime_monitor
```

前面的命令输出如下结果。



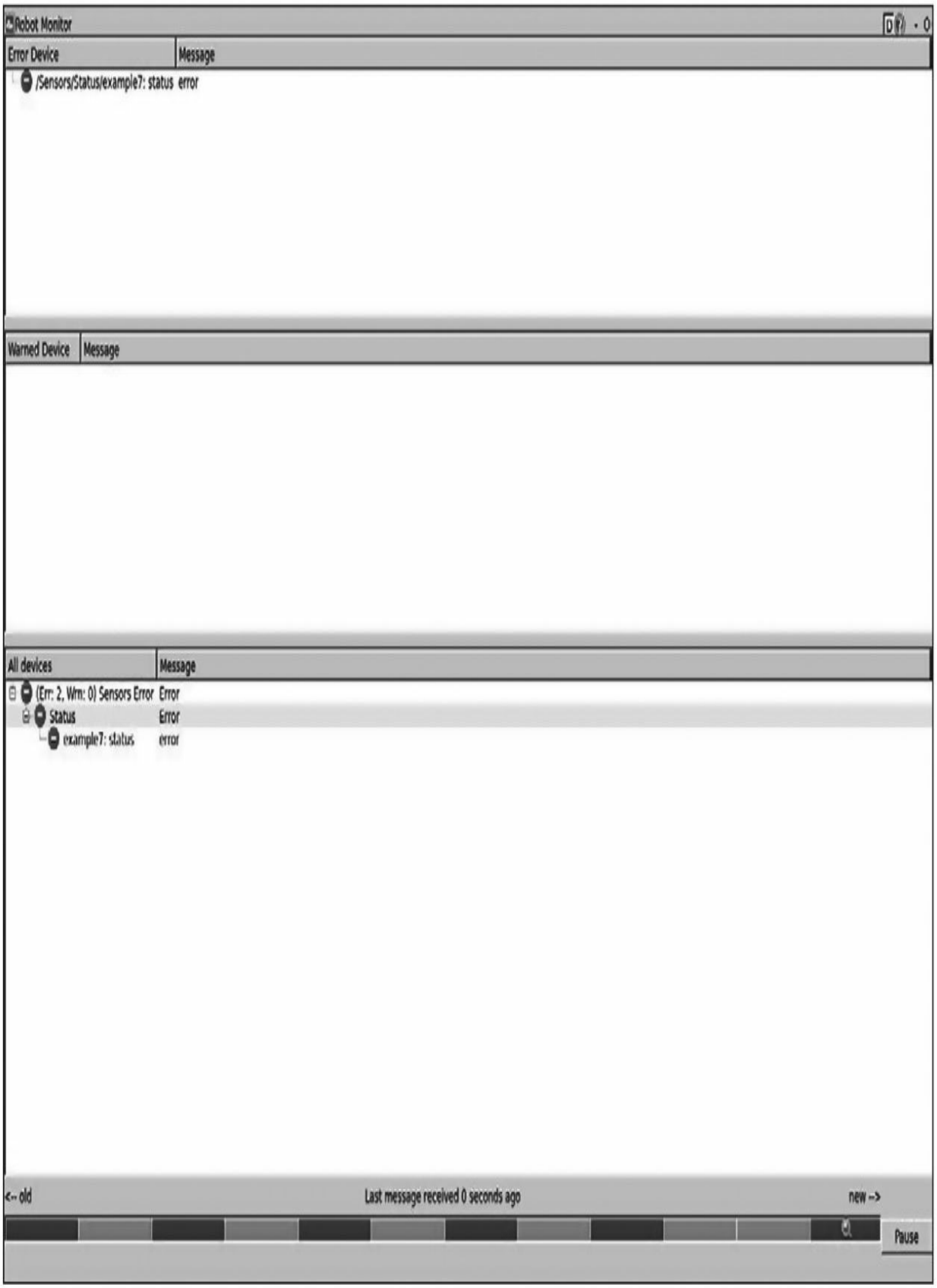
当系统很大时，可以使用diagnostic_aggregator汇总诊断信息。在diagnostics_agg里处理和归类diagnostics主题的消息并重新发布。这些汇总诊断消息通过rqt_robot_monitor实现可视化。诊断汇总器通过一个配置文件进行配置，例如下面的这一个（参看chapter3_tutorials中的config/diagnostic_aggregator.yaml文件），并使用AnalyzerGroup定义不同的analyzers（分析器）：

```
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：

```
$ rosrun rqt_robot_monitor rqt_robot_monitor
```

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


Robot Monitor
Error Device
Message
/Sensors/Status/example7: status error
Warned Device
Message
All devices
Message
(Err: 2, Wrn: 0) Sensors Error
Error
Status
Error
example7: status
error
<-- old
Last message received 0 seconds ago
new -->
Pause

3.7 绘制标量数据图

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，非标量数据图也可以绘制，但是要分别在不同的标量域里进行。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，有专门的可视化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，它允许我们在结构化消息中指定多个字段（当然使用相当简明的方式），也可以在GUI中手动添加或删除主题和字段。

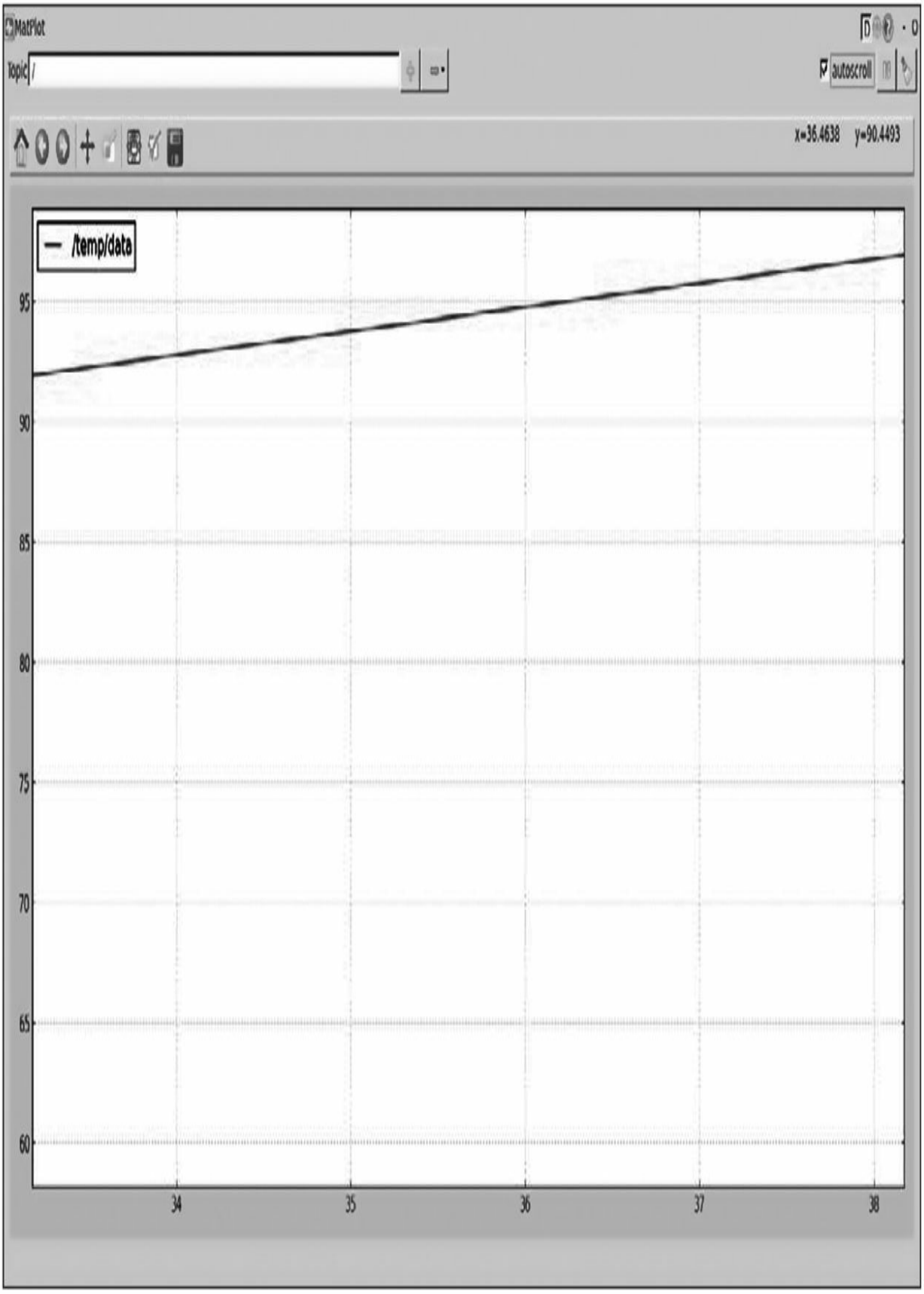
为了能够实际展示rqt_plot工具，使用example4节点，它在两个不同的主题中分别发布一个标量和一个矢量（非标量），这两个主题分别是温度（temp）和加速度（accel）。在这些消息中的值是随机生成的，所以它们没有实际意义，仅用于示范曲线绘制。首先以下面的命令运行节点：

```
$ 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，运行下面的命令：

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

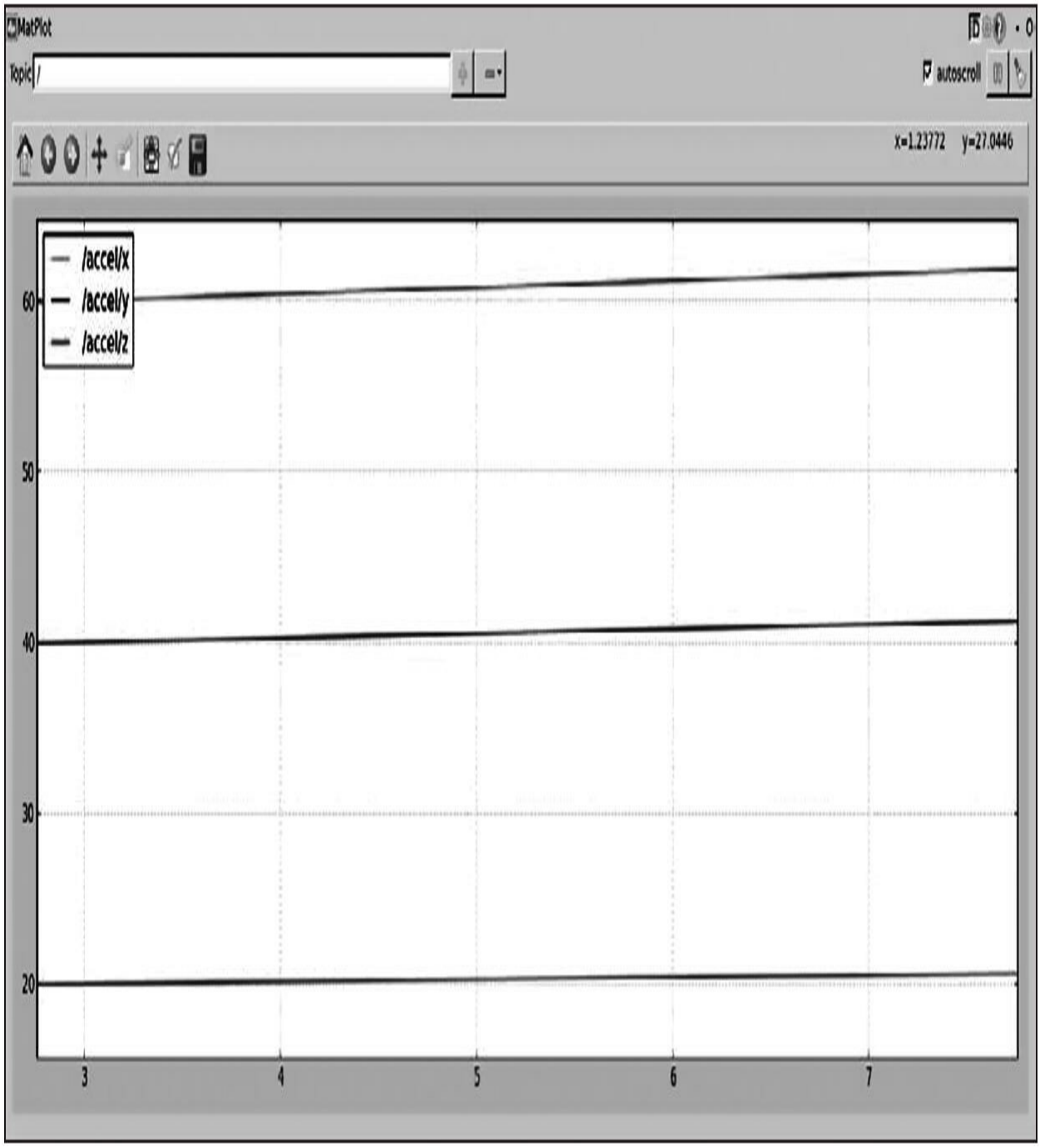
只要节点正常运行，我们就能看到曲线图随输入消息一直变化，如下图所示。


MatPlot
Topic /
autoscroll
x=36.4638 y=90.4493
/temp/data
95
90
85
80
75
70
65
60
34
35
36
37
38

在示例节点提供的accel主题里，我们看到一个Vector3消息（可以通过rostopic type/accel来查看），我们可以在一个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。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：

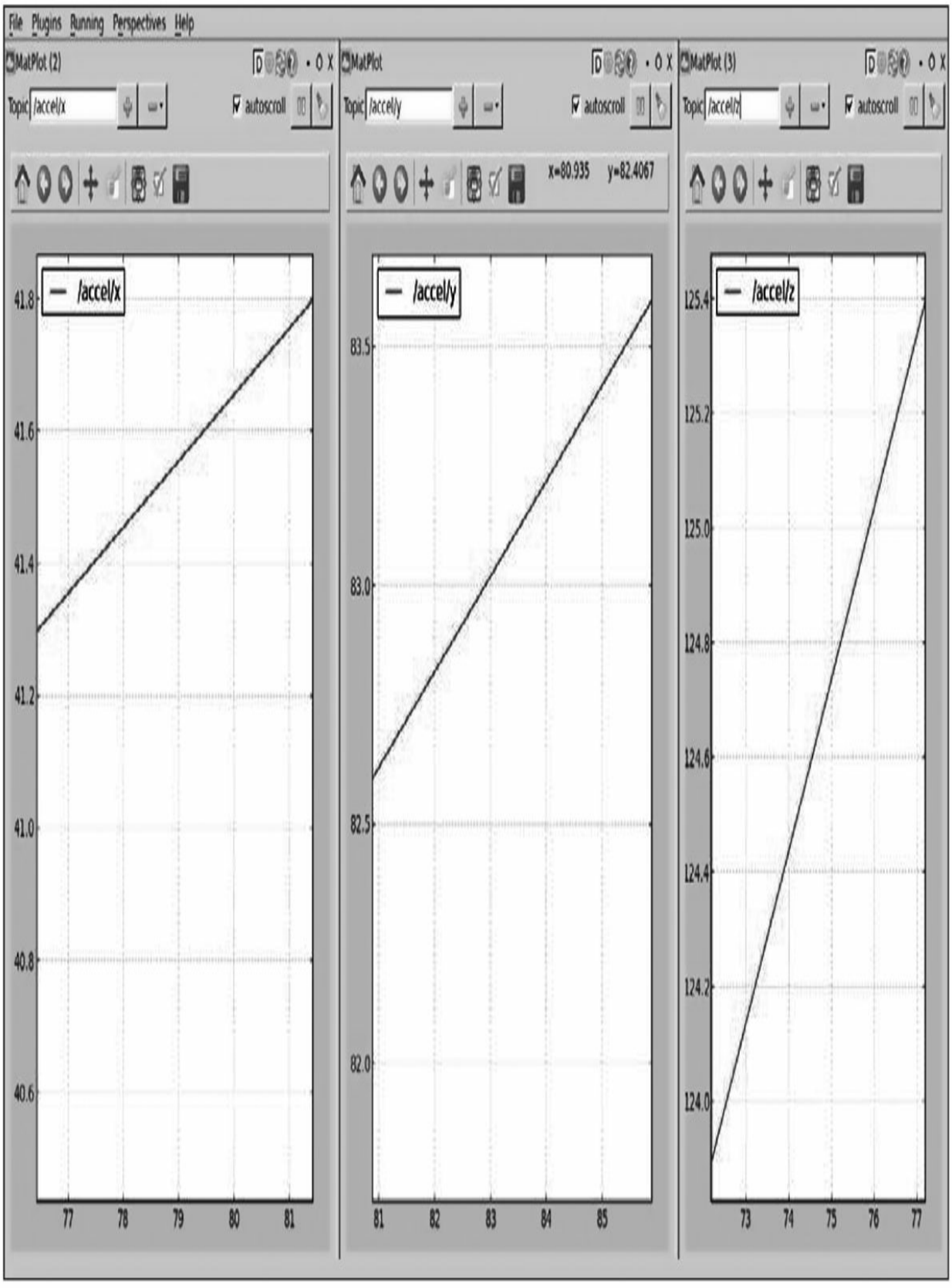
```
$ rosrun rqt_plot rqt_plot /accel/x:y:z
```

绘制的曲线图如下：

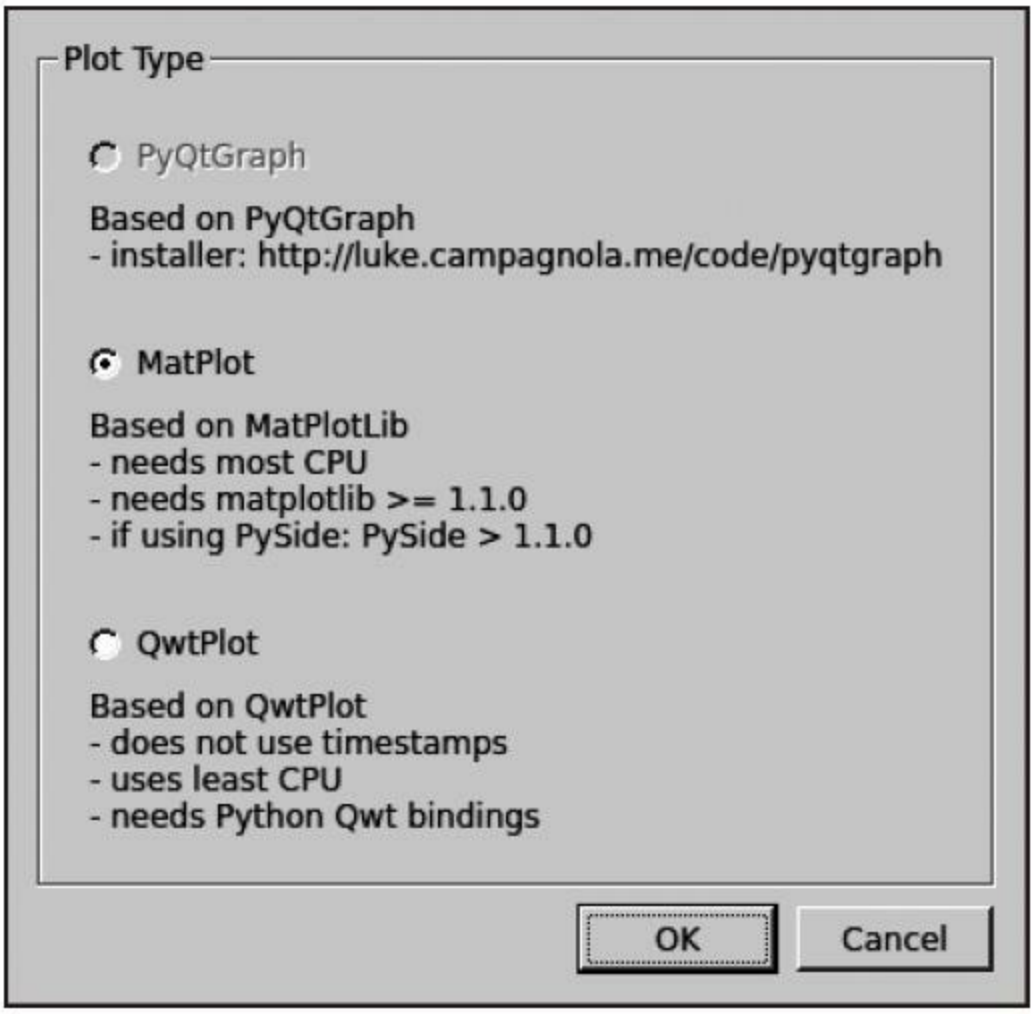


还可以将每个字段分开绘制。由于rqt_plot不直接支持这个功能，因此我们需要使用rqt_gui将三个图手动分开，如下图所示。

```
$ rosrun rqt_gui rqt_gui
```


File Plugins Running Perspectives Help
MatPlot (2)
Topic /accel/x
autoscroll
/accel/x
41.8
41.6
41.4
41.2
41.0
40.8
40.6
77
78
79
80
81
MatPlot
Topic /accel/y
autoscroll
x=80.935 y=82.4067
/accel/y
83.5
83.0
82.5
82.0
81
82
83
84
85
MatPlot (3)
Topic /accel/z
autoscroll
/accel/z
125.4
125.2
125.0
124.8
124.6
124.4
124.2
124.0
73
74
75
76
77

rqt_plot的GUI支持三种绘图前端。现在，可以使用更快的QT前端，同时它支持更多的时间序列。可以单击configuration（配置）按钮选择。



3.8 图像可视化

在ROS中，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，例如你笔记本电脑上的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图像转换为ROS Image消息，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，但是下面的章节将会介绍如何在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。

```
~$ 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

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：

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

这里使用了image_view节点，在窗口中展示了给定主题（使用image参数）的图像，如下图所示。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。可以通过在窗口中单击右边的按钮将当前帧保存在硬盘里，通常会保存在home目录下或者~/.ros目录下。

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

```
$ rosrun rqt_image_view rqt_image_view
```

前面命令的结果如下图所示。

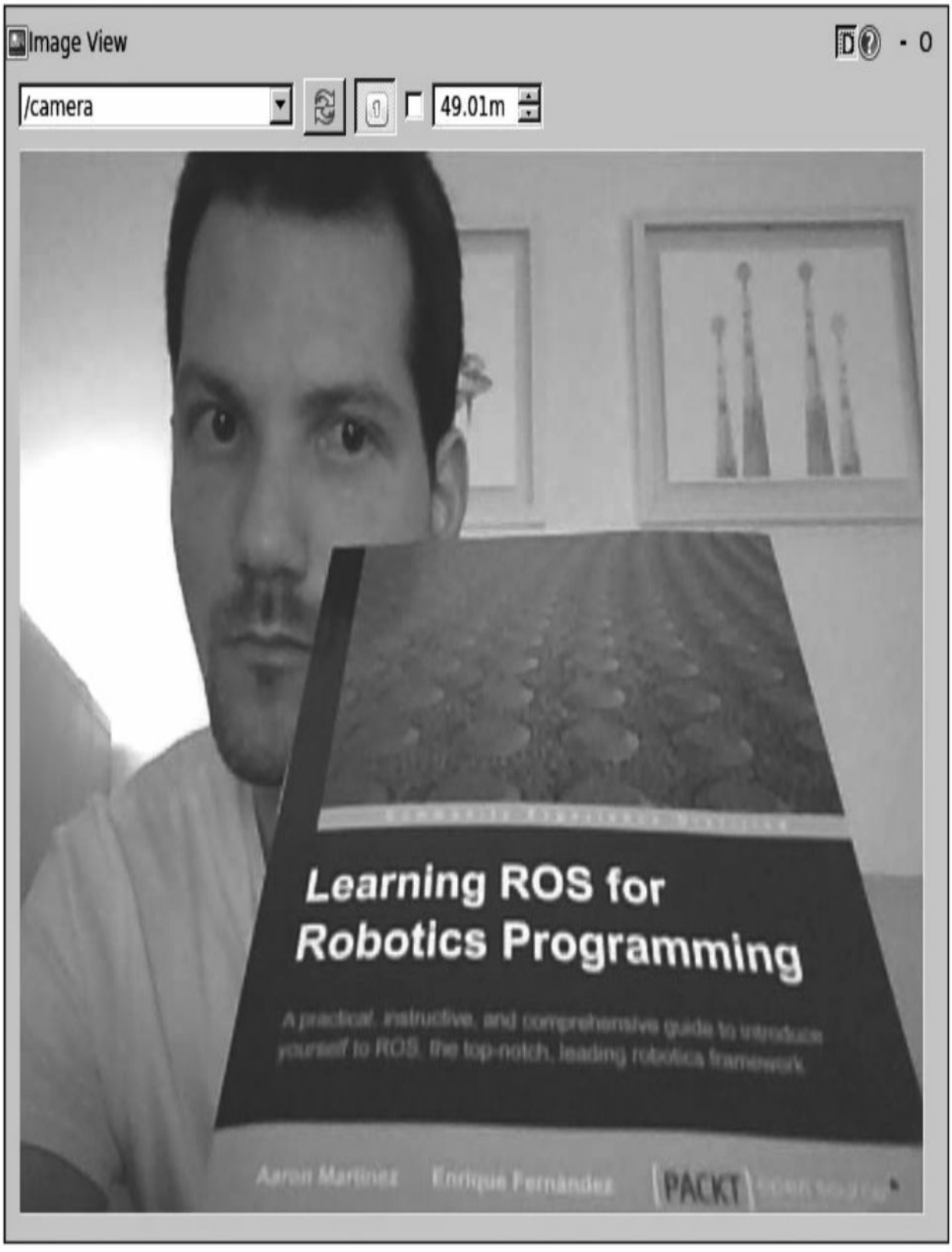


Image View
/camera
49.01m
Learning ROS for
Robotics Programming
A practical, instructive, and comprehensive guide to introduce
yourself to ROS, the top-notch, leading robotics framework
Aaron Martinez
Enrique Fernández
PACKT

ROS提供了一个基于OpenCV校准API的摄像头校准界面。第9章将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）的相关内容，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

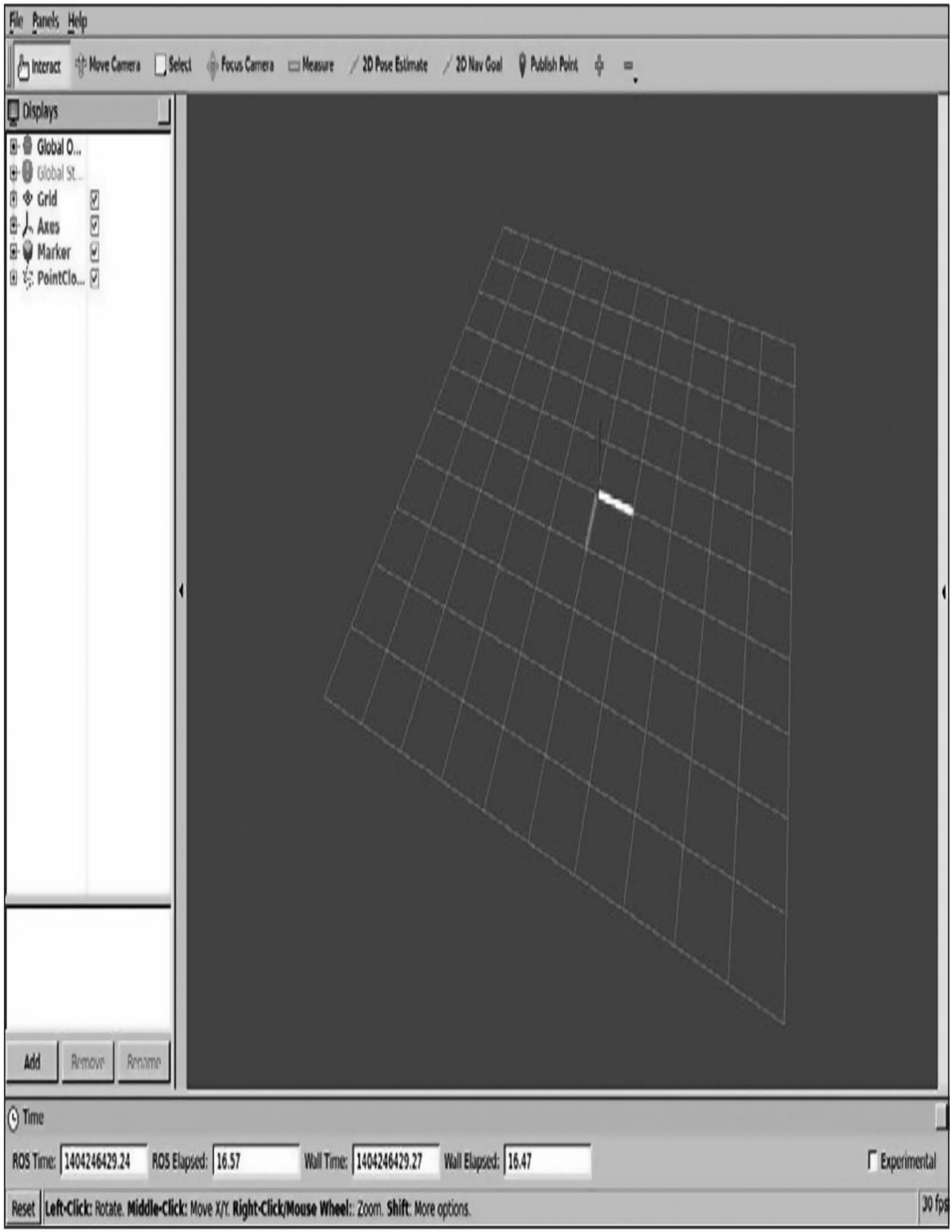
正如我们在前面章节中所见，有很多设备（例如双目摄像头、3D激光和Kinect传感器等）能够提供3D数据。它们通常使用点云格式（组织好的或未组织的）。基于上述原因，能够使用工具实现3D数据可视化就非常有用了。本节将介绍ROS中的rviz或rqt_rviz工具。它集成了能够完成3D数据处理的OpenGL界面，能够将传感器数据在模型化世界（world）中展示，过程是先使用传感器坐标系读取测量值，再将这些读数按照之间的相对位置在正确的位置绘制。

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 Kinetic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。


File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global O...
Global St...
Grid
Axes
Marker
PointClo...
Add
Remove
Rename
Time
ROS Time: 1404246429.24
ROS Elapsed: 16.57
Wall Time: 1404246429.27
Wall Elapsed: 16.47
Experimental
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
30 fps

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。本例中的配置和布局都存储在了config/example9.rviz文件中，这可以通过单击File|Open Config菜单项加载。

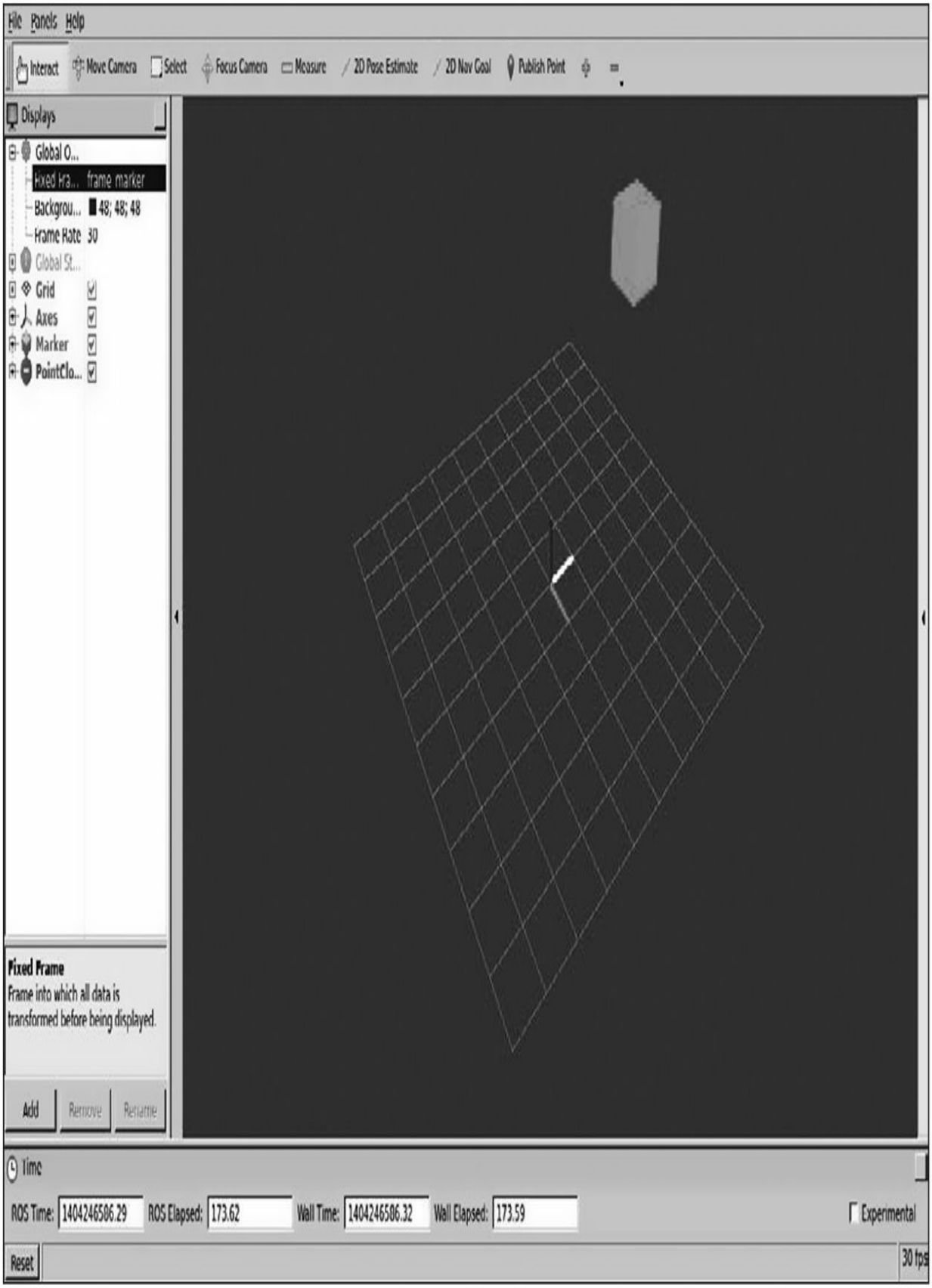
在Displays区域之下有一个Add按钮，它允许通过主题或类型添加更多的参数项。同时，注意，这里还有一些全局选项，它们基本上是用于设定固定坐标系的工具，因为坐标系是可以移动的。其次，还有Axes（轴）和Grid（网格），它们可以作为各个参数项的参照物。在本例中，对于example9节点，我们将会看到Marker（标记）和PointCloud2（点云）。

最后，在状态栏上有时间相关的信息提示，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。Views菜单提供了不同的查看类型，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俯视查看，其他菜单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Interact（交互）、Move（移动）、Measure（测量）等，以及一些插件。

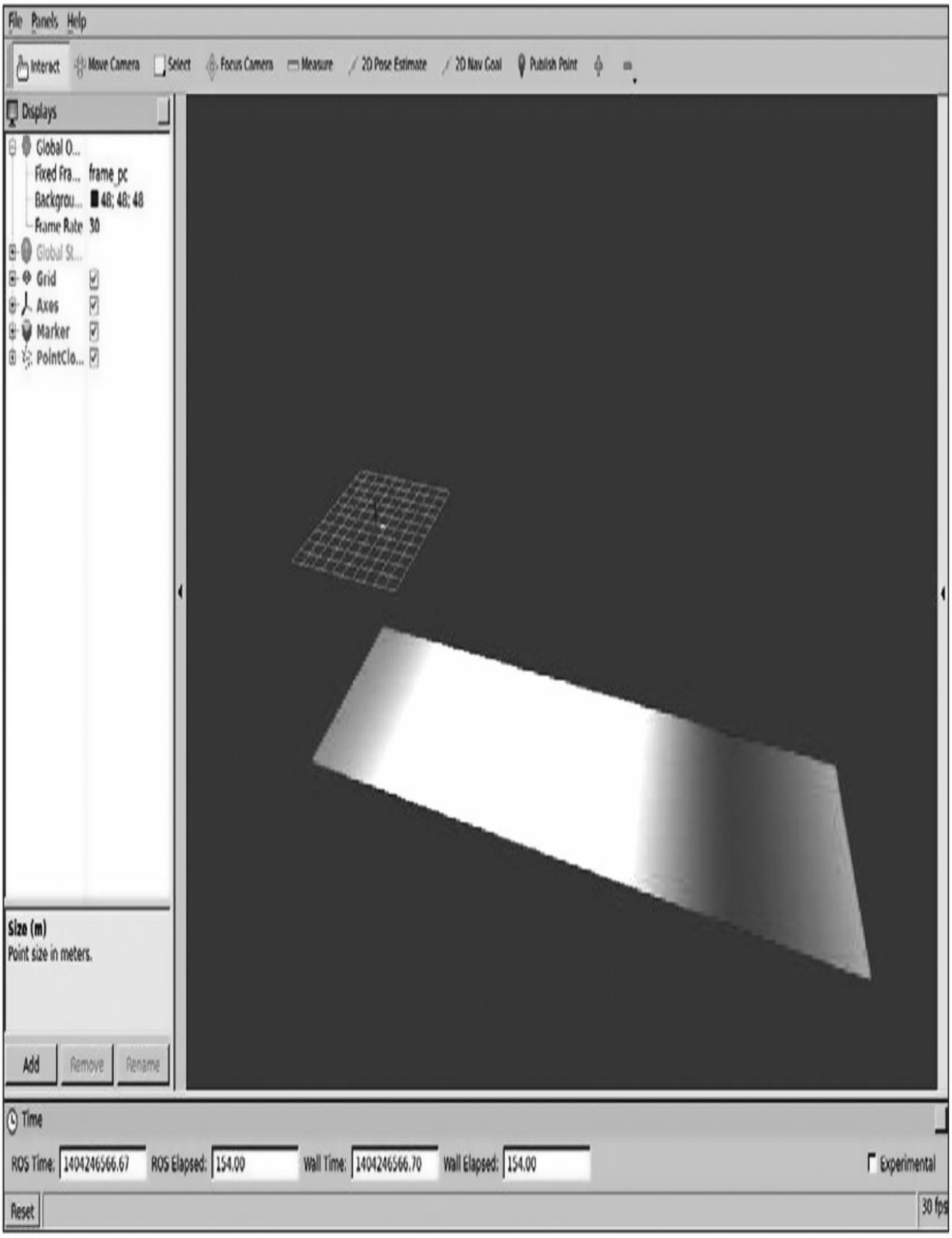
现在使用以下命令运行example9节点：

```
$ 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中的坐标标记（frame_marker）。我们将会看到红色方块标记（见彩插1）在移动，如下图所示。


File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global O...
Fixed Fra... frame marker
Backgrou... 48; 48; 48
Frame Rate 30
Global St...
Grid
Axes
Marker
PointClo...
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Remove
Rename
Time
ROS Time: 1404246586.29
ROS Elapsed: 173.62
Wall Time: 1404246586.32
Wall Elapsed: 173.59
Experimental
Reset
30 fps

类似地，如果设置Fixed Frame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。

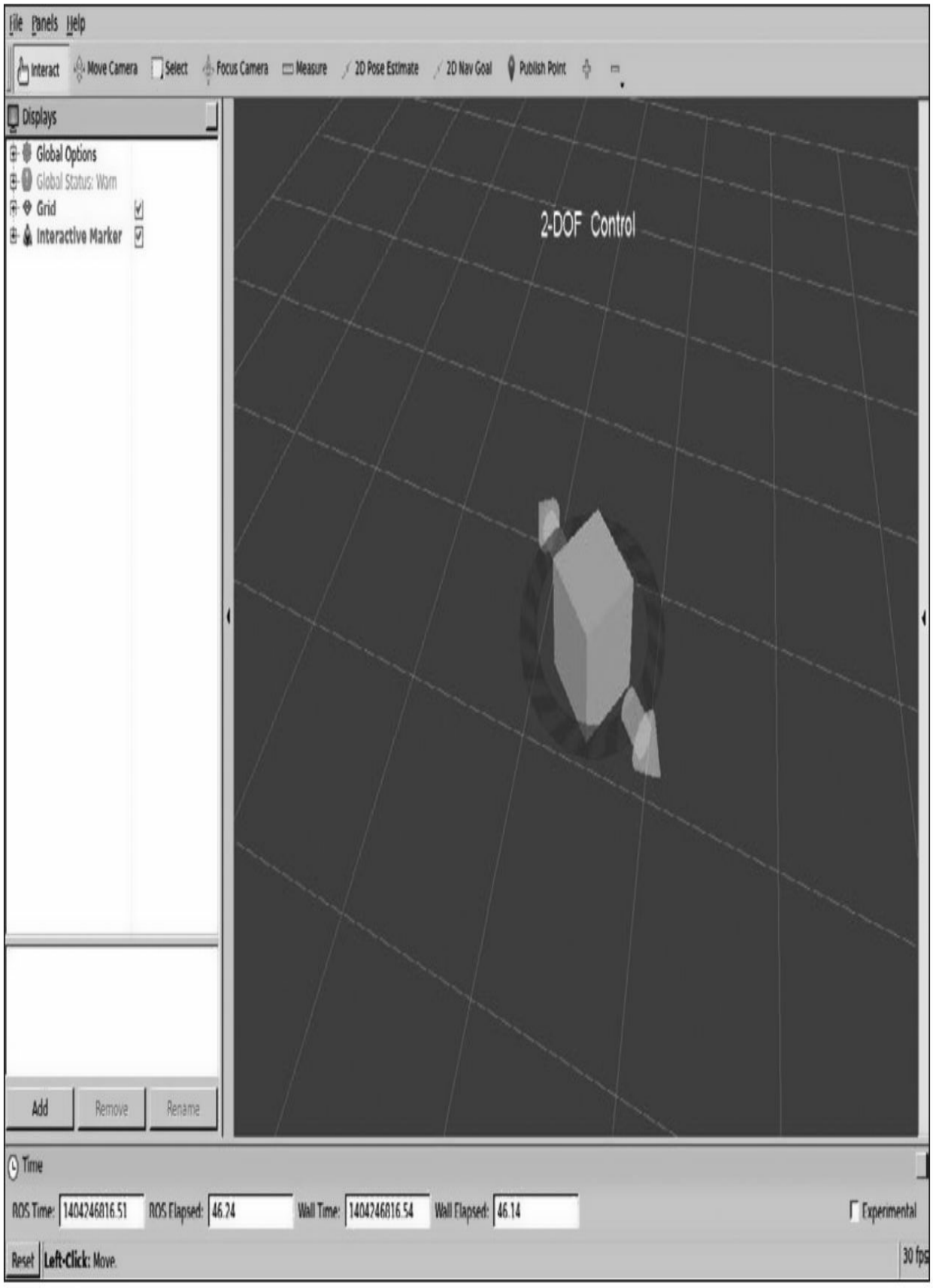
对于导航功能包集，将会在后面几章进行详细介绍。我们会使用多种数据类型，例如Odometry（画出机器人的里程位姿），Path（画出机器人所走过的路径），物体Pose，带有机器人位姿估计的粒子云PoseArray，使用Occupancy Grid Map（OGM）的Map，以及costmaps（这是ROS Kinetic中的Map类型，在ROS Kinetic之前是GridCell）。

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，并将每个元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而移动。

基本元素也可以表示，例如机器人足迹的Polygon、各种不同的标记Markers，它们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）。使用下面的命令，运行example10节点查看简单的交互式标记：

```
$ 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，例如机器人的关节：


File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Global Status: Warn
Grid
Interactive Marker
2-DOF Control
Add
Remove
Rename
Time
ROS Time: 1404246816.51
ROS Elapsed: 46.24
Wall Time: 1404246816.54
Wall Elapsed: 46.14
Experimental
Reset
Left-Click: Move.
30 fps

3.9.2 主题与坐标系的关系

如果数据从现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以检测环境中的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中，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，加盖戳记的消息还要附上frame_id（坐标系标签）。frame_id用于区分消息所属的坐标系。

然而，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树都会有一个base_link作为根坐标系（或者地图，如果运行导航功能包集）。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现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：

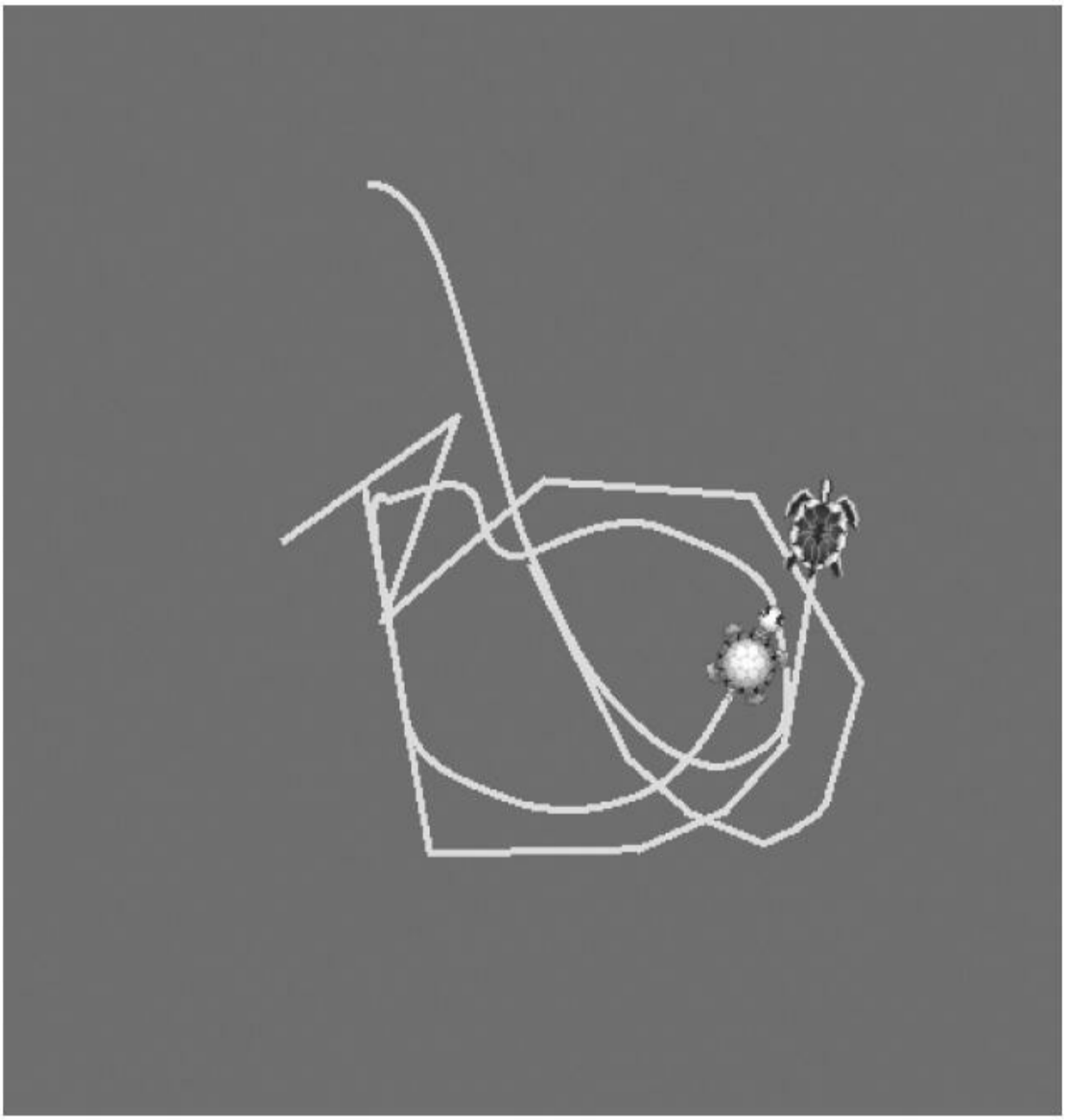
```
$ roslaunch turtle_tf turtle_tf_demo.launch
```

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tf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，这很可能是因为在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：

```
$ rosrun turtle_tf turtle_tf_listener
```

现在你应该会看到一个带有两只小海龟的窗口（小海龟的图标不同），一只小海龟跟着另一只。可以使用键盘上的上下左右四个方向键控制小海龟。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。下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。



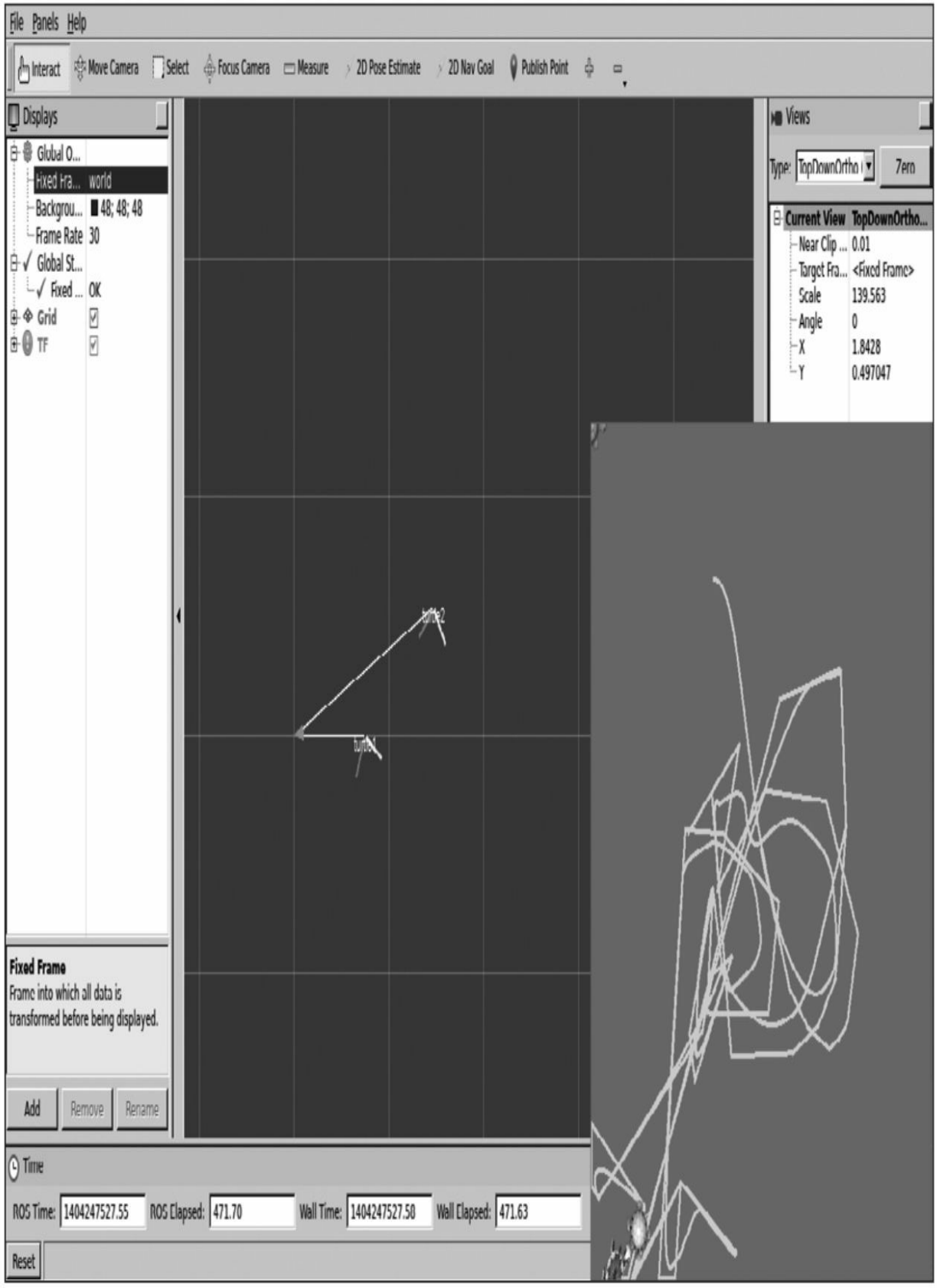
每一只小海龟都有自己的坐标系，可以在rqt_rviz中查看它们：

```
$ rosrun rqt_rviz rqt_rviz
```

现在，先放下turtlesim窗口，我们先来看当使用方向键移动小海龟时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根坐标系的子坐标系。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（见彩插2）。同时，设定world的视图（view）类型为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，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。


File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global O...
Fixed Fra... world
Backgrou... 48; 48; 48
Frame Rate 30
Global St...
Fixed ... OK
Grid
TF
Views
Type: TopDownOrtho
Zero
Current View TopDownOrtho...
Near Clip ... 0.01
Target Fra... <Fixed Frame>
Scale 139.563
Angle 0
X 1.8428
Y 0.497047
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Remove
Rename
Time
ROS Time: 1404247527.55
ROS Elapsed: 471.70
Wall Time: 1404247527.50
Wall Elapsed: 471.63
Reset

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，或者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实验却不是容易的事情，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（bag）文件来保存消息，并包含消息的所有字段参数和时间戳。这允许离线回放实验并模拟真实的状态，包括消息的时间延迟。甚至ROS工具能够非常有效地在处理高速数据流时以充足的数据组织方式来实现这一切。

下一节会解释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

消息记录包文件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，不仅是标头里的时间戳，还包括消息记录包文件中的功能包。用于记录的时间戳和标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。

消息记录包文件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，它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，只需要在使用rosbag record命令记录时使用-j参数，你会在下一节看到具体示例。

每个消息与发布它的主题都被记录下来。因此，可以指定某个主题进行记录，或者直接选择全部（使用-a参数）。然后，回放消息记录包文件时，同样可以通过指明特定主题的名称来选择消息记录包文件中全部主题的某个特定子集进行回放。

3.10.2 使用rosbag在消息记录包文件中记录数据

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：

```
$ rosrun chapter3_tutorials example4
```

现在有两个选择。第一个是记录所有的主题：

```
$ rosbag record -a
```

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

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

默认情况下，当运行前面的命令时，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日期作为文件名的消息记录包文件中。一旦完成实验，只需要在运行的终端中按Ctrl+C组合键。下面就是一个记录会话和消息记录包文件结果的示例：

```
[ 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，其中包括消息记录包文件的大小、记录的持续时间、分割文件为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，我们能以兆字节（MB）为单位增加记录缓冲区（-b）大小。缓冲区大小默认值是256MB，但在某些数据存储速率特别高的情况下也可能需要吉字节（GB）级的缓冲区（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，需要在launch文件中添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文件直接运行rosbag的时候，消息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

既然我们有了一个消息记录包文件，就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消息记录包示例），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
```

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

通过rostopic list命令自动查看主题列表，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，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。

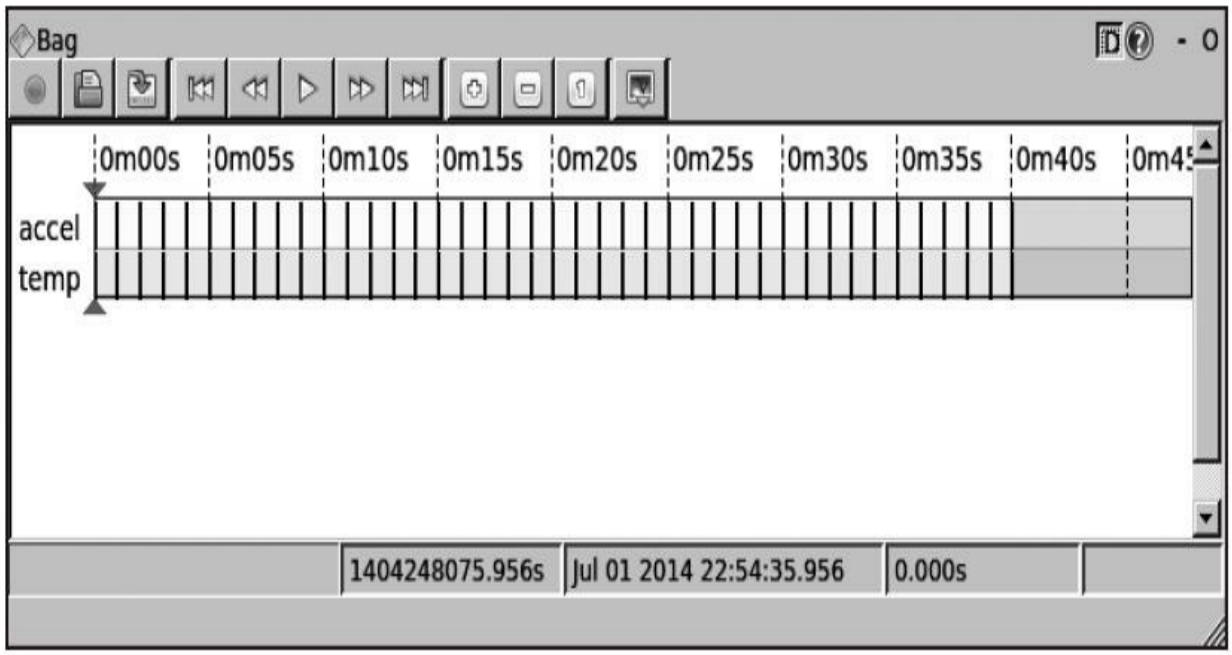
3.10.4 查看消息记录包文件的主题和消息

这里有两个主要的方法来查看消息记录包文件内部的数据。第一个非常简单。只需要输入rosbag info<bag_file>，结果如下图所示。

```
~$ 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        [da5909fbe378aeaf85e547e830cc1bb7]
topics:      /accel   41 msgs    : geometry_msgs/Vector3
             /temp    41 msgs    : std_msgs/Int32
```

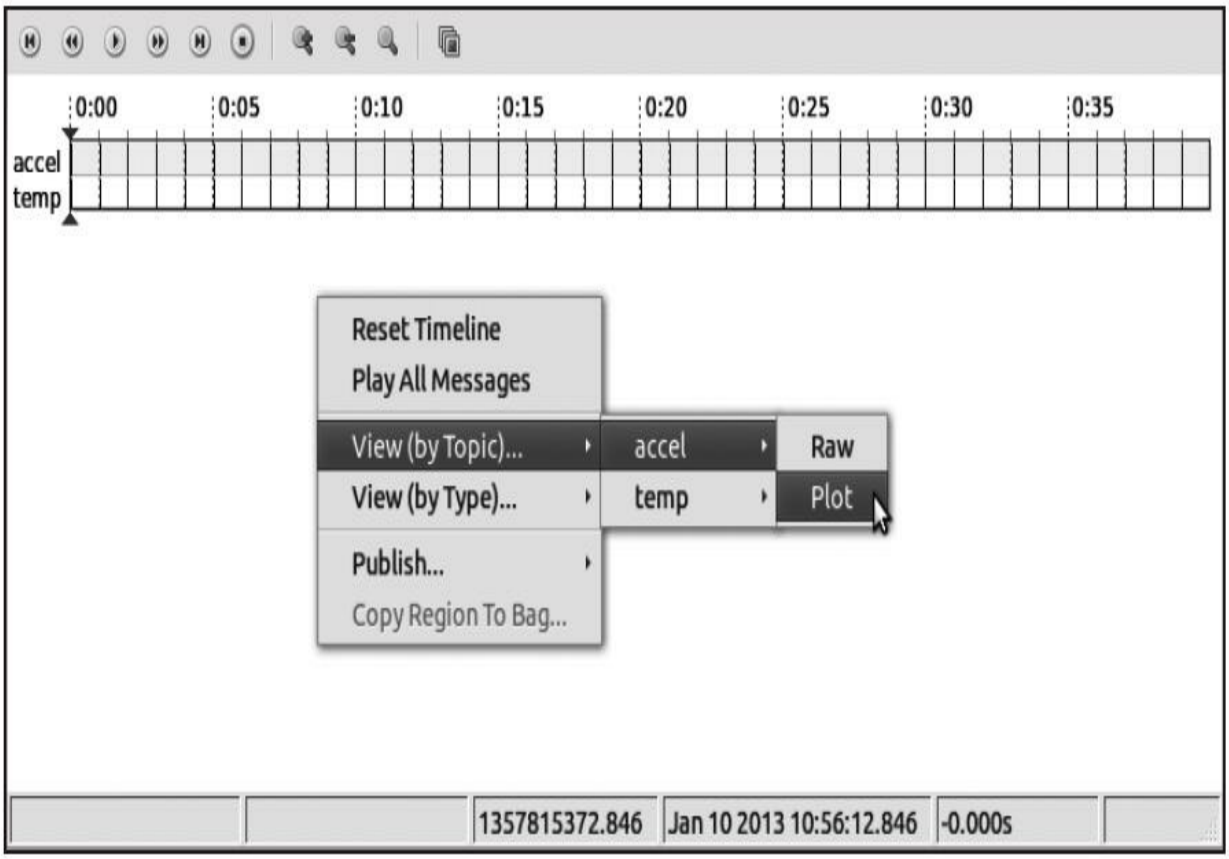
我们有消息记录包文件本身的信息，例如创建的日期、持续时间、文件大小，以及内部消息的数量和文件的压缩格式（如果有压缩）。然后，我们还会有文件内部数据类型的列表。最后有主题的列表，并它们对应的名称、消息数量和类型。

第二种检查消息记录包文件的方法则非常强大。这个GUI工具叫作rqt_bag。它不仅具有GUI，还允许我们回放消息记录包文件、查看图像（如果有）、绘制标量数据图和消息的RAW结构。我们只需要传递消息记录包文件的名称，就会看到如下图所示的界面（使用之前的消息记录包文件）。

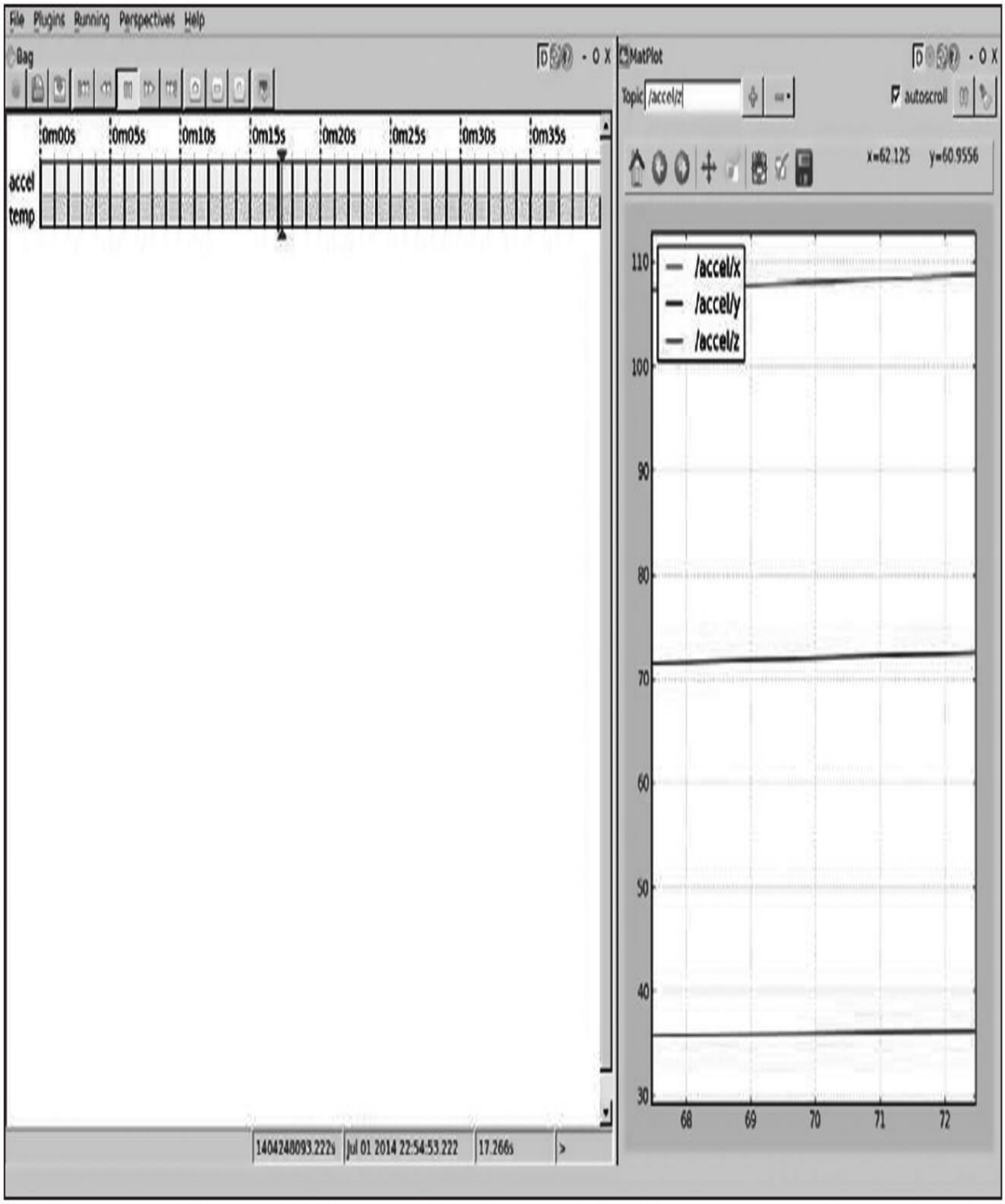


我们有一个能用于所有主题的时间条。对于每一个消息都会带有这个标记。在本例中，能够在时间条中使用缩略图查看消息。

在下图中，我们能够看到如何调用Raw、Plot和Image指令（如果是Image类型的主题），来查看消息记录包文件中的主题。在时间条上右击就会弹出这个菜单。



作为补充，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，使用Pubulish All和播放查看绘图。对于/accel主题，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，单击+按钮/图标添加所有字段。请注意，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，它只是简单显示回放和发布的数据：



需要注意的是，由于rxbag的特点，至少要单击一次play按钮才能够对数据进行绘制。我们还能够播放、暂停、停止和移动到文件头或者文件结尾。这里图像都很简单，而且会出现一个带有当前坐标系和配置选项的窗口，能够将这些内容作为图像文件存储在硬盘上。

3.11 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。使用下面的列表对本章介绍的工具进行说明（ROS Kinetic的rqt工具以及对应被替代的之前的ROS版本）：

ROS Kinetic rqt 工具	替代者（ROS Fuerte 或者之前版本）
rqt_console 与 rqt_logger_level	rxconsole
rqt_graph	rxgraph
rqt_reconfigurerqt_reconfigure	dynamic_reconfigurereconfigure_gui
rqt_plot	rxplot
rqt_image_view	image_view
rqt_bag	rxbag

在ROS Kinetic中，这些可以单独运行的插件甚至更多，比如shell（rqt_shell）、主题的发布者（rqt_publisher）、消息类型的查看器（rqt_msg）等，本章涵盖了这些重要的插件。甚至rviz都有一个名为rqt_rviz的插件，它还能够集成在最新的rqt_gui界面中。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所示：

```
$ 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、调试错误、可视化结果等大量实用工具。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定的概念和工具。

现在你知道了如何在代码中包含具有不同详细程度的日志消息。这能够帮助你调试节点错误。为了实现这个目的，还需要使用ROS中包含的其他一些强大工具，例如rqt_console界面。另外，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工具查看节点状态图。

对于可视化工具，你已学习了使用rqt_plot绘制标量数据图。它能够以特定变量更加直观的分析方式发布节点。类似地，你能够查看更加复杂的数据类型（非标量数据）。这包括分别使用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rosbag工具学会了记录和回放会话中的消息，还知道了如何使用rqt_bag查看消息记录包文件的内容。这允许你基于以往的经验记录数据和使用AI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　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如下内容：

·创建机器人的3D模型

·为机器人提供运动、物理限制、惯性和其他物理特性

·在机器人3D模型上添加仿真传感器

·在仿真环境中使用此模型

4.1 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常规工作，它们在ROS中通过URDF文件实现。

标准化机器人描述格式（Unified Robot Description Format，URDF）是一种用于描述机器人及其部分结构、关节、自由度等的XML格式文件。每次在ROS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。

4.2 创建第一个URDF文件

我们将在下面几节中建立的机器人是一种最常见的移动机器人，它有四个轮子和一个带有夹持器的手臂。

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chapter4_tutorials/robot1_description/urdf文件夹下创建一个新文件并命名为robot1.urdf，并将下面的代码复制到文件中：

```
<?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>
```

这些URDF代码是基于XML的，这种格式并不强制缩进，但还是建议这么做。因此，建议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）。

4.2.1 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：link（连接）和joint（关节）。

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）以及原点。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（fixed、revolute、continuous、floating或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轮轴，那么可以设置其为转动关节（revolute）。

为了检查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
```

此命令的输出如下图所示。

```
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
```

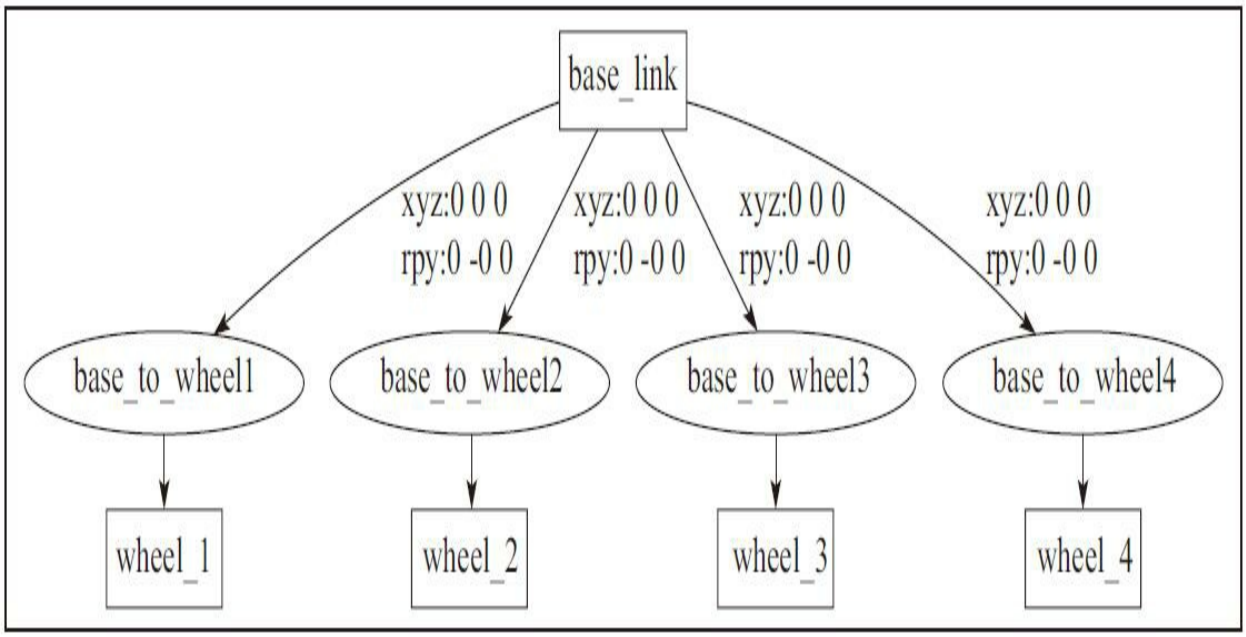
如果你希望以图形的方式来查看它，那么可以使用urdf_to_graphiz命令工具：

```
$ urdf_to_graphiz robot1.urdf
```

此命令将生成两个文件：origins.pdf和origins.gv。可以使用evince打开文件：

```
$ evince origins.pdf
```

然后你会在输出文件中看到下图。



4.2.2 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显示，以及它关节的运动。

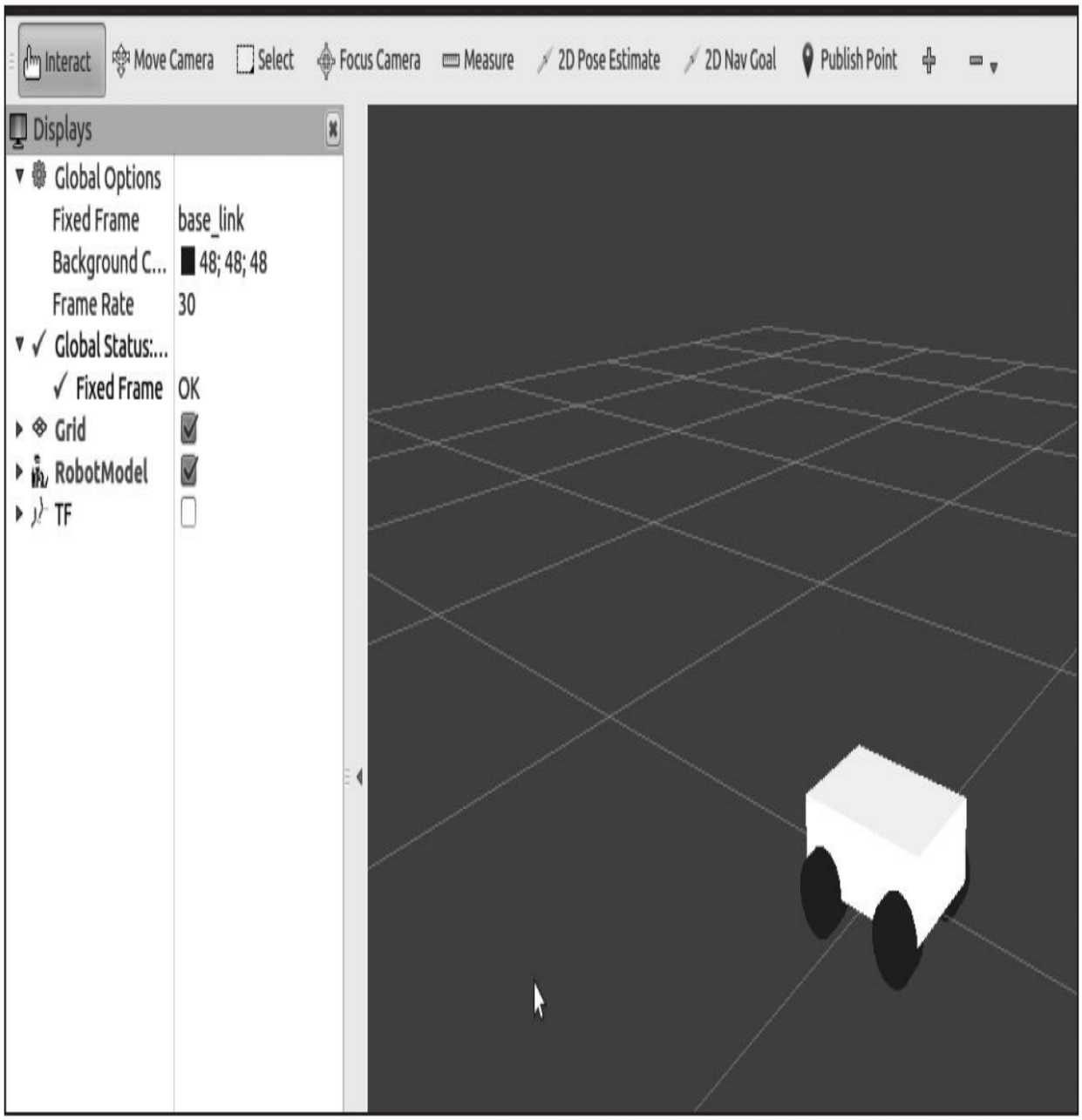
接着在robot1_description/launch文件夹下创建display.launch文件，并在文件中输入以下代码：

```
<?xml version="1.0"?>


<launch>
  <arg name="model" />
  <arg name="gui" default="False" />
  <param name="robot_description" textfile="$(arg model)" />
  <param name="use_gui" value="$(arggui)" />
  <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>
```

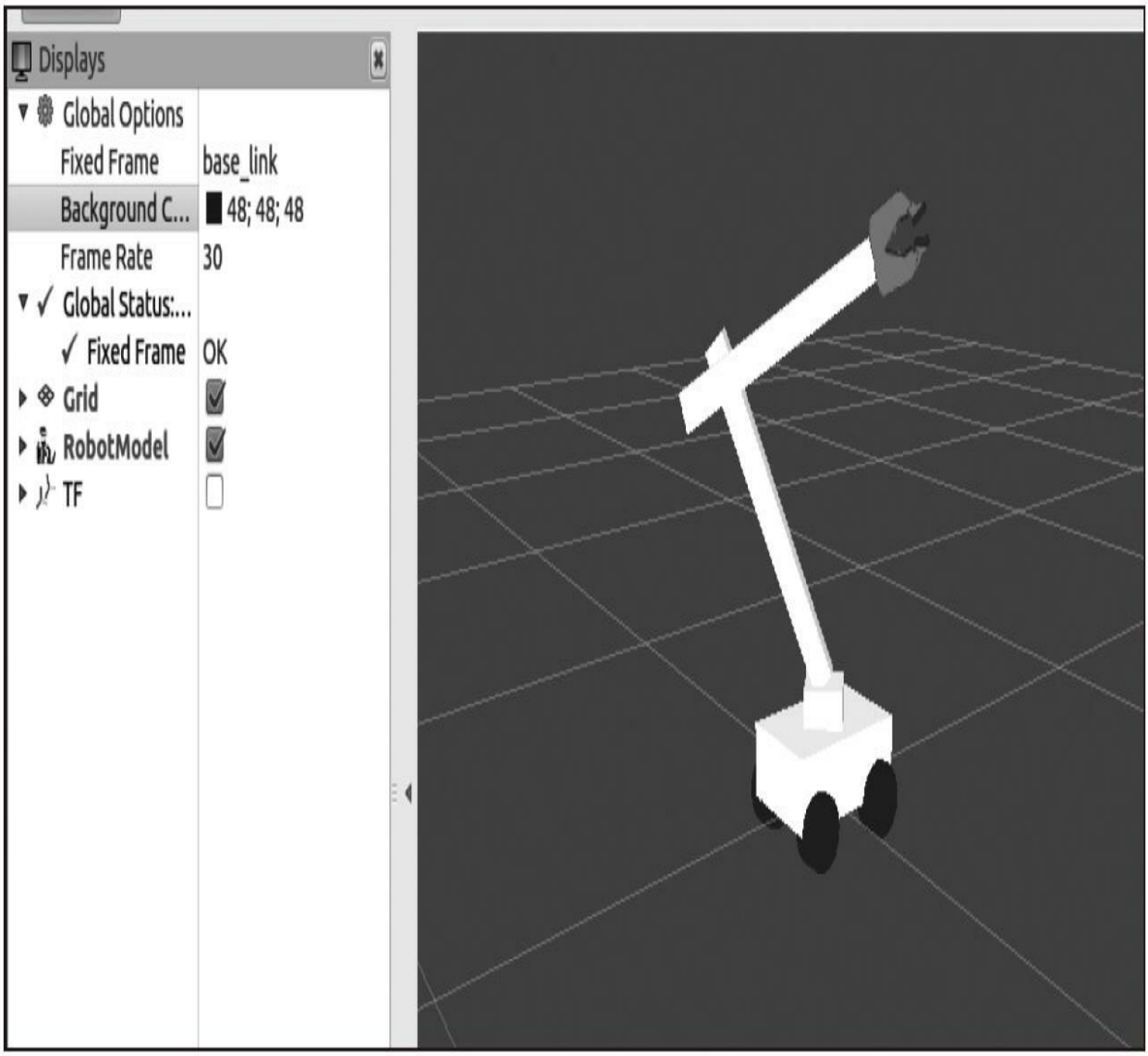
使用如下命令启动它：

```
$ roslaunch robot1_description display.launch model:="'rospack 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，可以在chapter4_tutorials/robot1_description/urdf/robot1.urdf文件中找到最终设计模型，如下图所示。

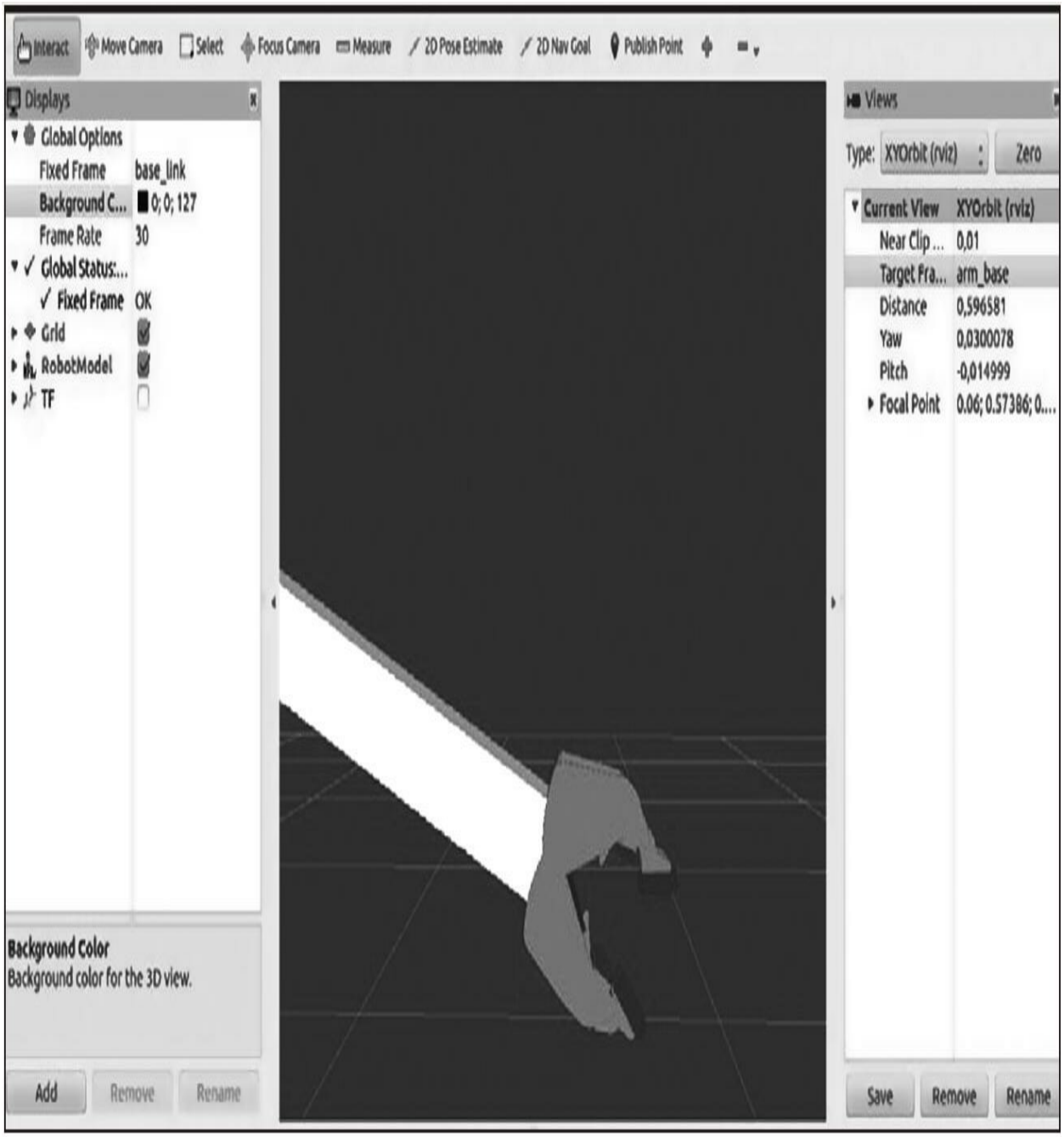

Displays
Global Options
Fixed Frame
base_link
Background C...
48; 48; 48
Frame Rate
30
Global Status:...
Fixed Frame
OK
Grid
RobotModel
TF

4.2.3 加载网格到机器人模型中

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）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，使用PR2机器人的夹持器。在下面的代码中你会看到是如何使用它的：

```
<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。


Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
base_link
Background C...
0; 0; 127
Frame Rate
30
Global Status:...
Fixed Frame
OK
Grid
RobotModel
TF
Background Color
Background color for the 3D view.
Add
Remove
Rename
Views
Type:
XYOrbit (rviz)
Zero
Current View
XYOrbit (rviz)
Near Clip ...
0,01
Target Fra...
arm_base
Distance
0,596581
Yaw
0,0300078
Pitch
-0,014999
Focal Point
0.06; 0.57386; 0....
Save
Remove
Rename

4.2.4 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，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，如我们在arm_1_to_arm_base上所使用的，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axis xyz="100"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，旋转关节的单位是弧度，移动关节的单位是米），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极限值是否合适，一种好的办法就是使用Join_State_Publisher图形化用户界面（GUI）运行rviz：

```
$ roslaunch robot1_description display.launch model:="'rospack find
robot1_description'/urdf/robot1.urdf" gui:=true
```

我们会看到另一个窗口中的rviz界面，窗口中有很多滑块的，其中每个滑块都能控制一个关节。

4.2.5 物理和碰撞属性

如果你想在Gazebo或者其他仿真软件中进行机器人仿真，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，需要首先设定重量才能够计算惯性等。

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，使用简单的几何形状比实际的网格模型更容易进行碰撞计算。相比简单的几何形状，在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>
```

对于其他的连接臂也是如此。记住，要为所有连接添加collision和inertial元素，因为如果你不这样做，Gazebo将无法使用这些模型。

你能够在robot1_description/urdf/robot1_physics.urdf中查看包含所有参数的完整文件。

4.3 xacro——一种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。想象一下如果添加摄像头、腿或者其他几何部件，那么这个文件将会开始急剧增大，并且维护这些代码将会变得非常困难。

Xacro（XML Macros的简写）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以创建相同的结构，如更多的手臂和腿。

为了开始使用xacro，需要指定一个命名空间，以便文件能够正确地解析。例如，下面是一个有效的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。

4.3.1 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，若需要改变一个值，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" />
```

为了使用这些变量，现在只需要使用${name_of_variable}引用将旧值更新为下列新值：

```
${name_of_variable}:
<cylinder length="${length_wheel}" radius="${radius_wheel}"/>
```

4.3.2 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，需要先做好参数设计。

4.3.3 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.pyrobot1.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，为了保证书写的命令更加简洁，还建议在同一个文件夹下工作。

4.3.4 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
  doubleinc= 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::TransformStampedodom_trans;
  sensor_msgs::JointStatejoint_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)
    // (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;
}
```

下面分析如何对代码进行改进以获得这些运动。

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，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::TransformStampedodom_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，所以创建一条带有7个字段的消息：

```
sensor_msgs::JointStatejoint_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，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
  $(arg model)" />
  <param name="use_gui" value="$(arggui)"/>
  <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>
```

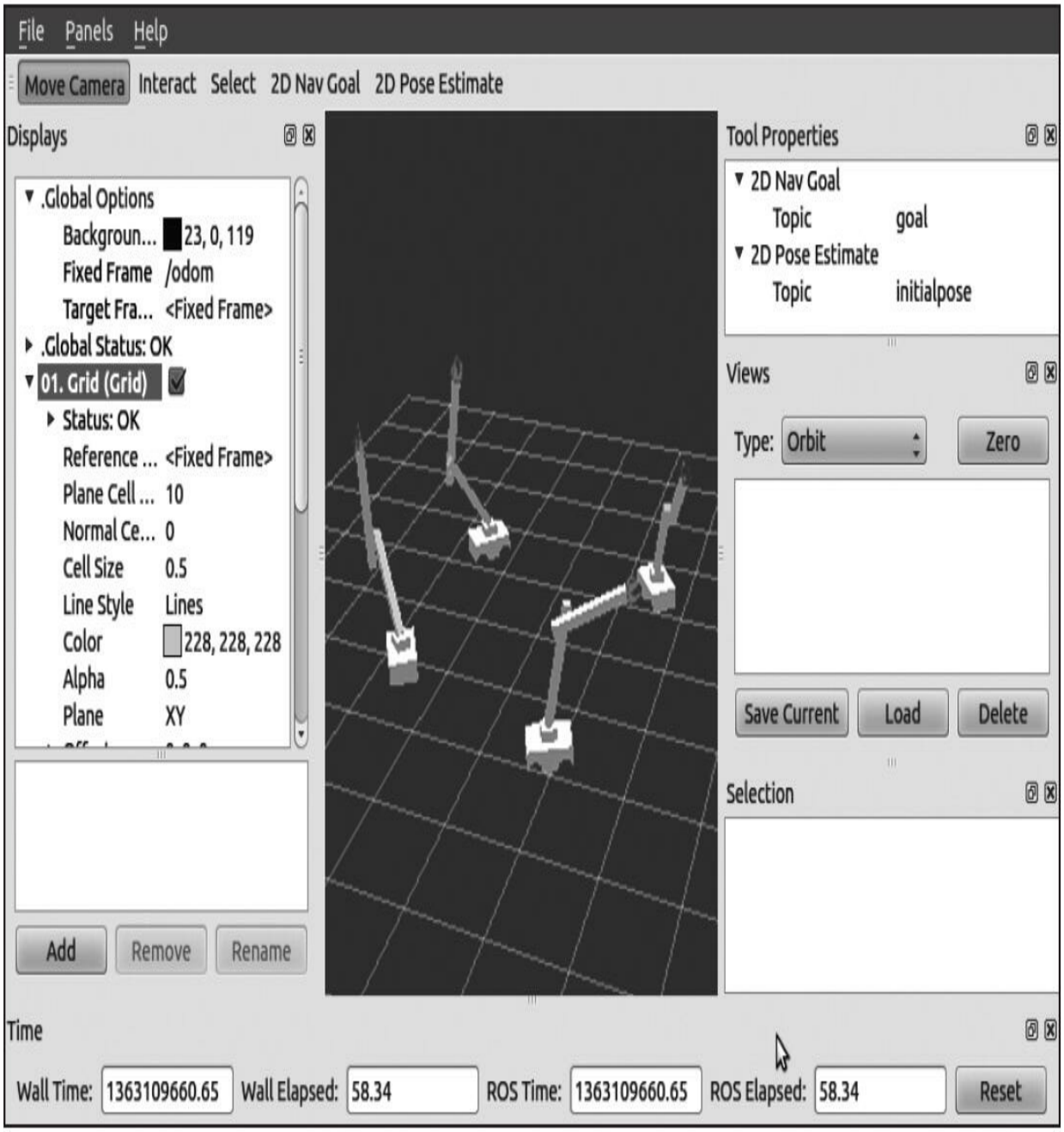
启动该节点前，必须安装以下功能包：

```
$ sudo apt-get install ros-kinetic-map-server

$ sudo apt-get install ros-kinetic-fake-localization

$ cd ~/catkin_ws && catkin_make
```

使用以下命令，启动带有完整模型的新节点。我们将会在rviz中看到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"
```

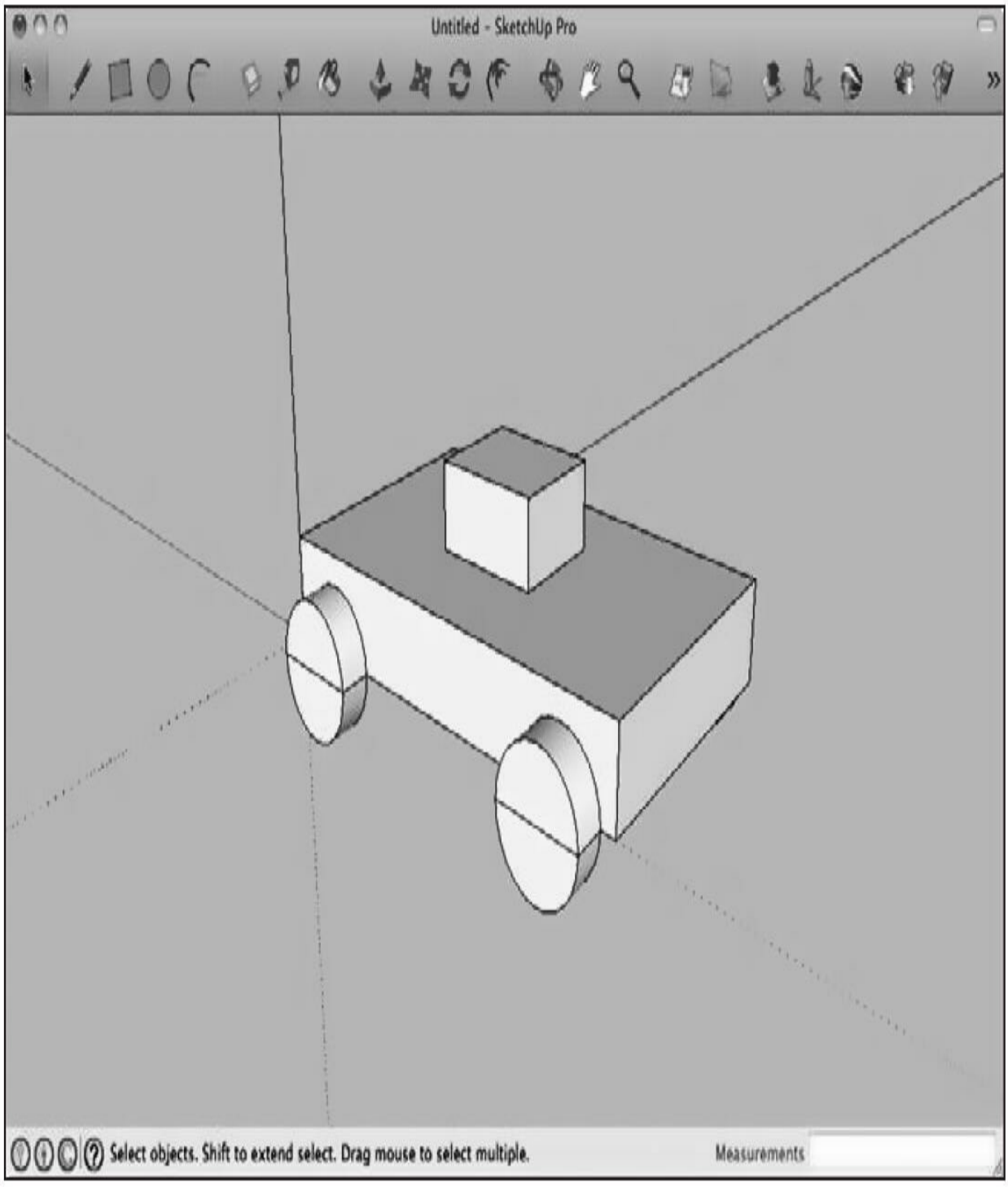
在下图中，你能看到4个窗口的组合，它们分别显示了我们通过节点捕捉的机器人运动。如果你能认真观察它，就能看到手臂移动及其圆形的轨迹。



4.3.5 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并发布简单模型，生成urdf文件，以及在rviz中查看模型。请注意，SketchUp只能在Windows系统或Mac系统下运行。本节的模型是在Mac中开发的，而不是在Linux中。

首先，需要在电脑中安装SketchUp。安装好之后，创建一个与下图类似的模型。



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)。

将文件命名为bot.dae，并将这个文件保存在robot1_description/meshes文件夹下。

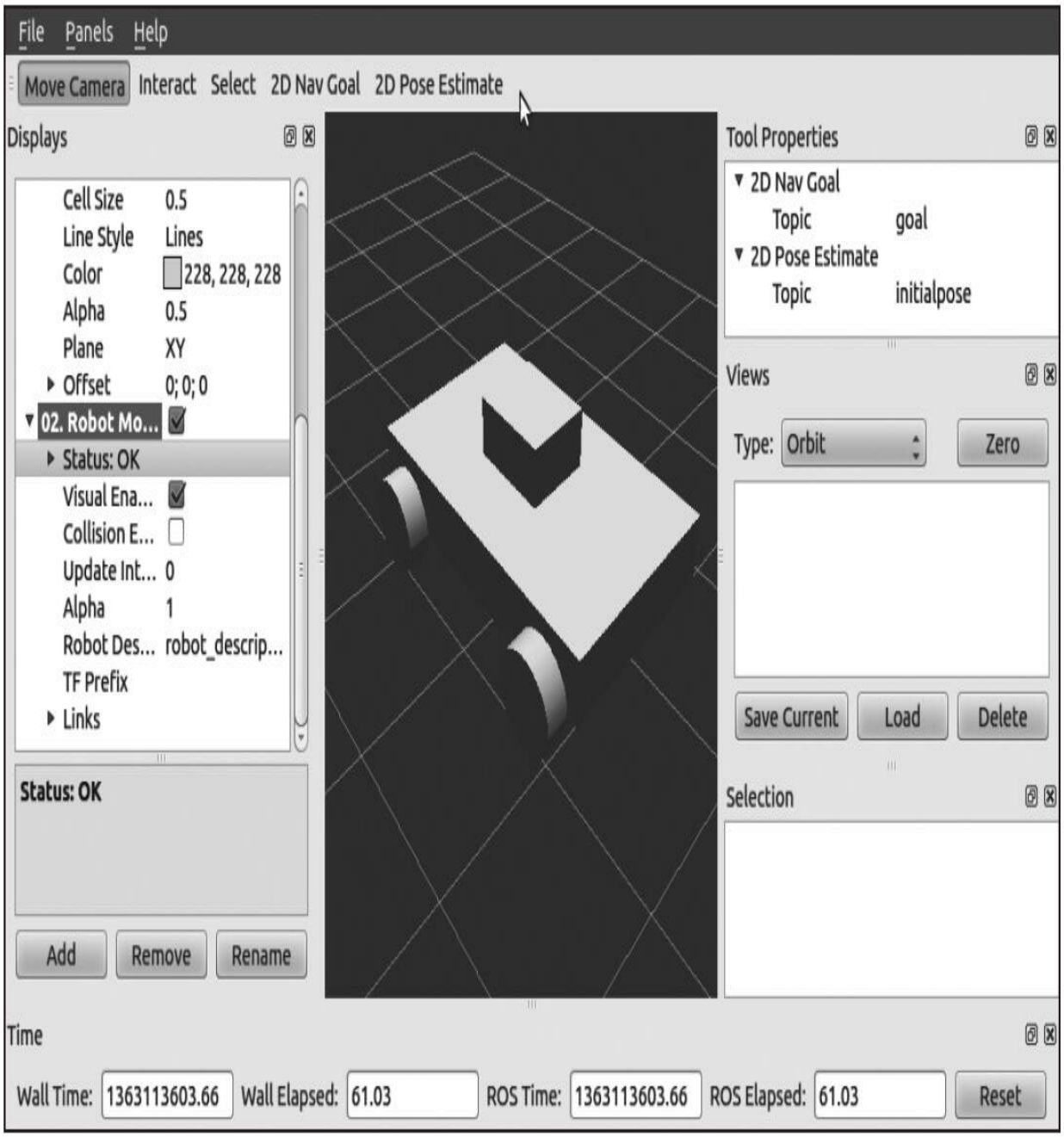
现在，为了能使用3D模型，我们将在robot1_description/urdf文件夹下创建一个以dae.urdf为名的新文件。在文件中添加以下代码：

```
<?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，可以通过命令<mesh scale="0.0250.0250.025"filename="package://robot1_description/meshes/bot.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。


File Panels Help
Move Camera Interact Select 2D Nav Goal 2D Pose Estimate
Displays
Cell Size 0.5
Line Style Lines
Color 228, 228, 228
Alpha 0.5
Plane XY
Offset 0; 0; 0
02. Robot Mo...
Status: OK
Visual Ena...
Collision E...
Update Int... 0
Alpha 1
Robot Des... robot_descrip...
TF Prefix
Links
Status: OK
Add Remove Rename
Tool Properties
2D Nav Goal
Topic goal
2D Pose Estimate
Topic initialpose
Views
Type: Orbit
Zero
Save Current Load Delete
Selection
Time
Wall Time: 1363113603.66
Wall Elapsed: 61.03
ROS Time: 1363113603.66
ROS Elapsed: 61.03
Reset

4.4 在ROS中仿真

要在ROS中对机器人进行仿真，需要使用Gazebo。

Gazebo（http://gazebosim.org/）是一种适用于复杂室内和室外环境的多机器人仿真环境。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，生成实际传感器的反馈以及物体之间的物理交互。

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，如何加载一个激光传感器和一个摄像头，并使机器人模型像真的机器人一样移动。

4.4.1 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通信：

```
$ sudo apt-get install ros-kinetic-gazebo-ros-pkgsros-kinetic-Gazebo-ros-
control
```

接下来将打开Gazebo GUI。如果一切正常，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 & rosrun gazebo_ros Gazebo
```

为了在Gazebo中导入机器人模型，需要先完成URDF模型。要在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。能够在chapter4_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。

要启动这所有的东西，将要创建一个新的.launch文件。在chapter4_tutorials/robot1_gazebo/launch文件夹下创建一个名为gazebo.launch的新文件，并添加以下代码：

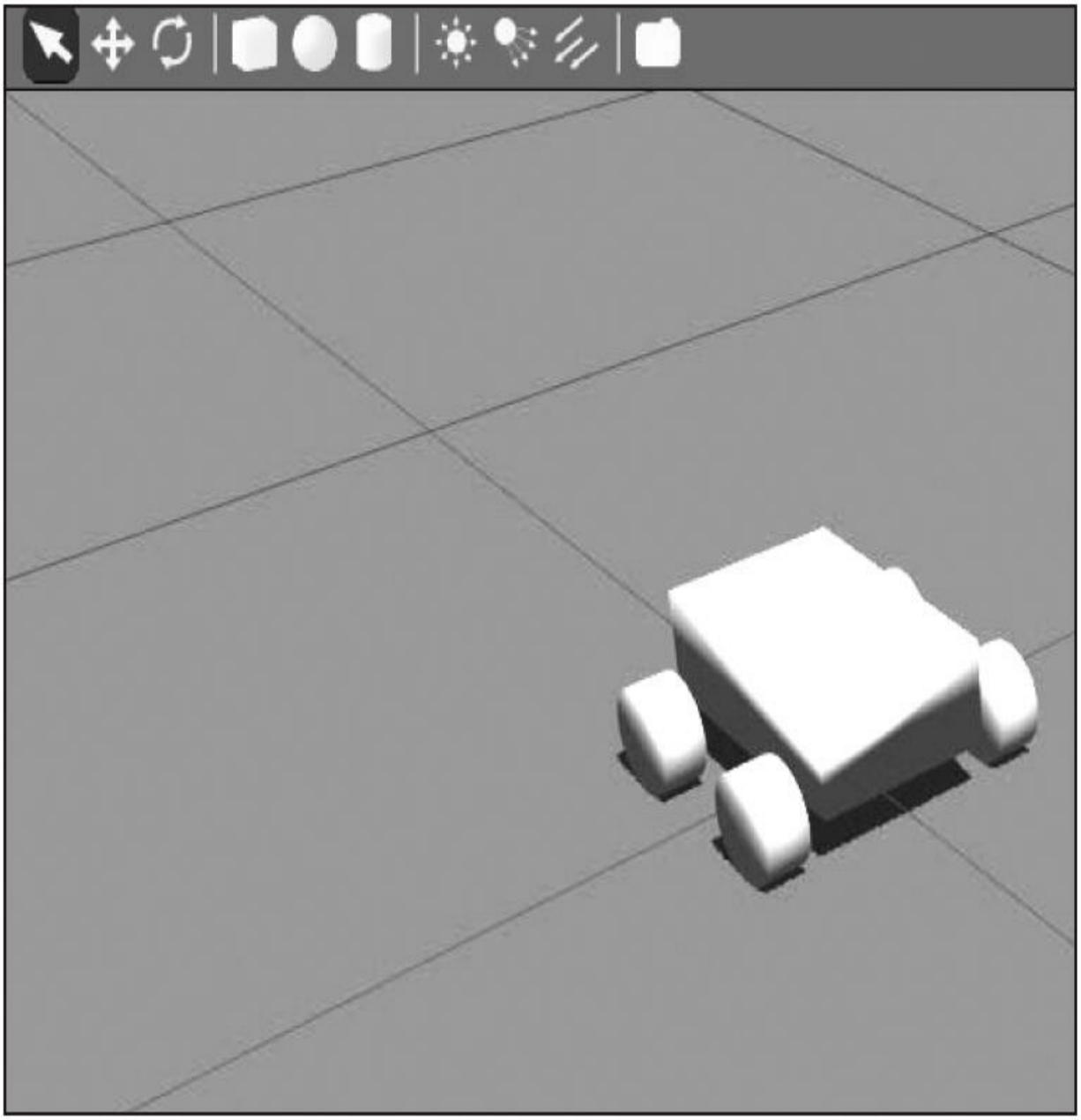
```
<?xml version="1.0"?>

<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="$(arg debug)" />
    <arg name="gui" value="$(arggui)" />
    <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 -
  paramrobot_description -z 0.05" />
</launch>
```

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，可单击play按钮来运行它。祝贺你！这是你在虚拟世界中迈出的第一步。

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.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>
```

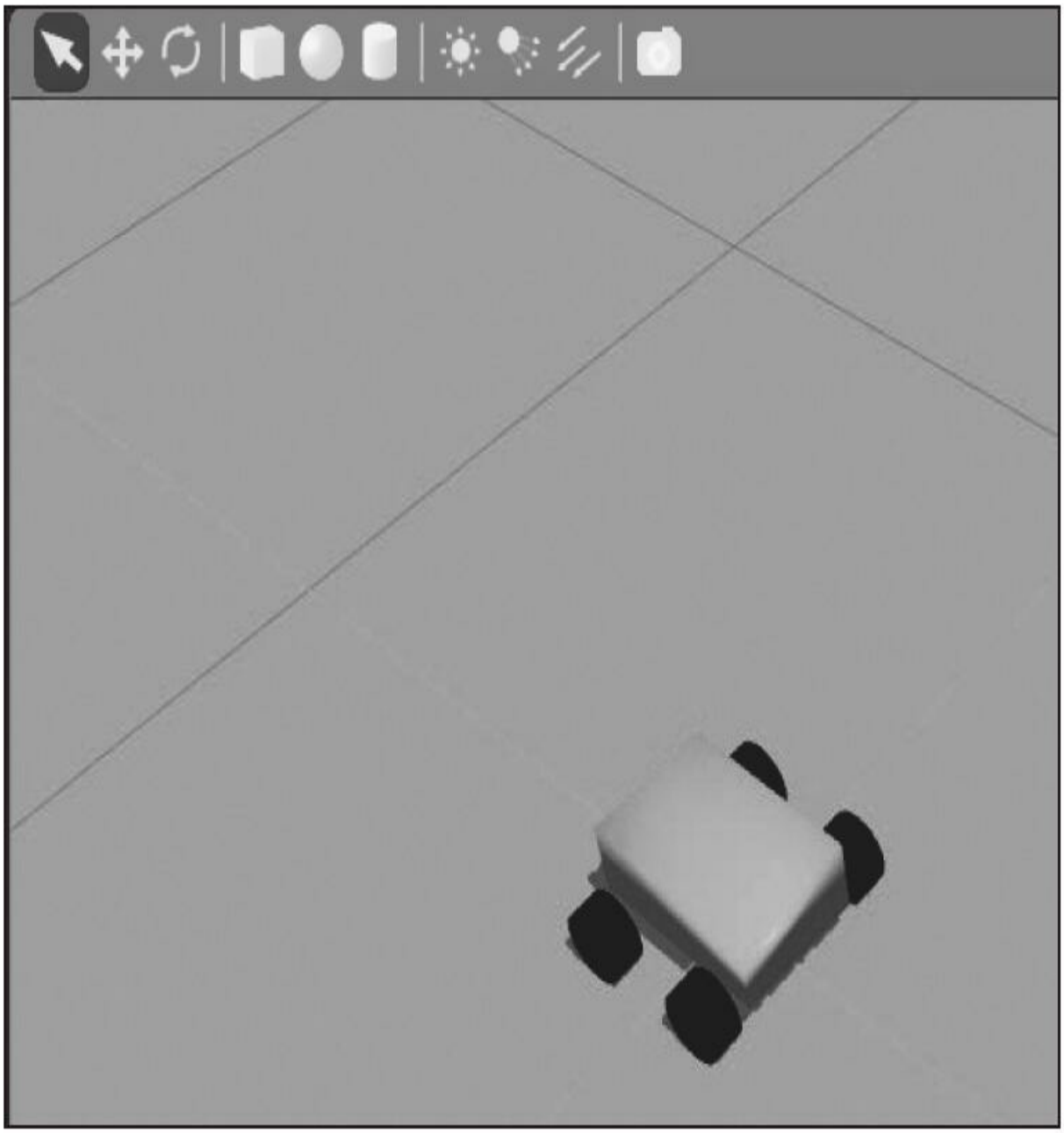
将robot1_description/urdf/robot1_base_01.xacro文件另存为robot1_base_02.xacro，并添加以下代码：

```
<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。



4.4.2 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传感器。这个传感器将会成为机器人的一个新部件。因此，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，并有一个代表摄像头的红色立方体，其他章曾经介绍过它。

我们会从gazebo_ros_demos功能包中调用激光。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文件中增加这些代码行来为机器人添加Hokuyo激光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>
    <massvalue="1e-5" />
    <originxyz="0 0 0" rpy="0 0 0" />
    <inertiaixx="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>
```

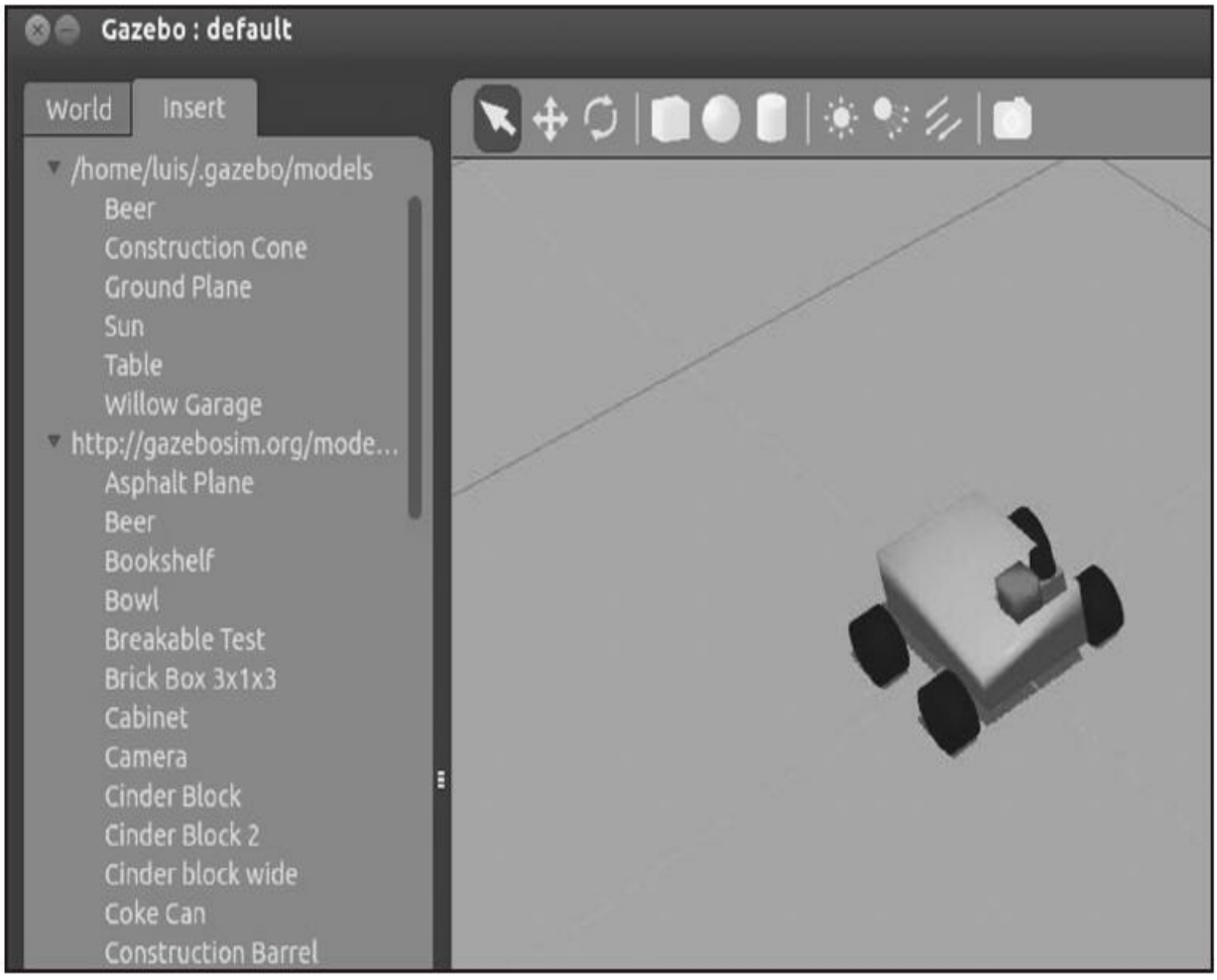
使用以下命令启动新的模型：

```
$ roslaunch robot1_gazebo gazebo.launch model:="'rospack find robot1_
description'/urdf/robot1_base_03.xacro"
```

你将会看到附带激光模块的机器人。

采用类似的方法，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）。请查阅这些文件！

在下图中，你可以看到机器人模型为带有Hokuyo激光和模拟摄像头模型的红色立方体（见彩插3）。



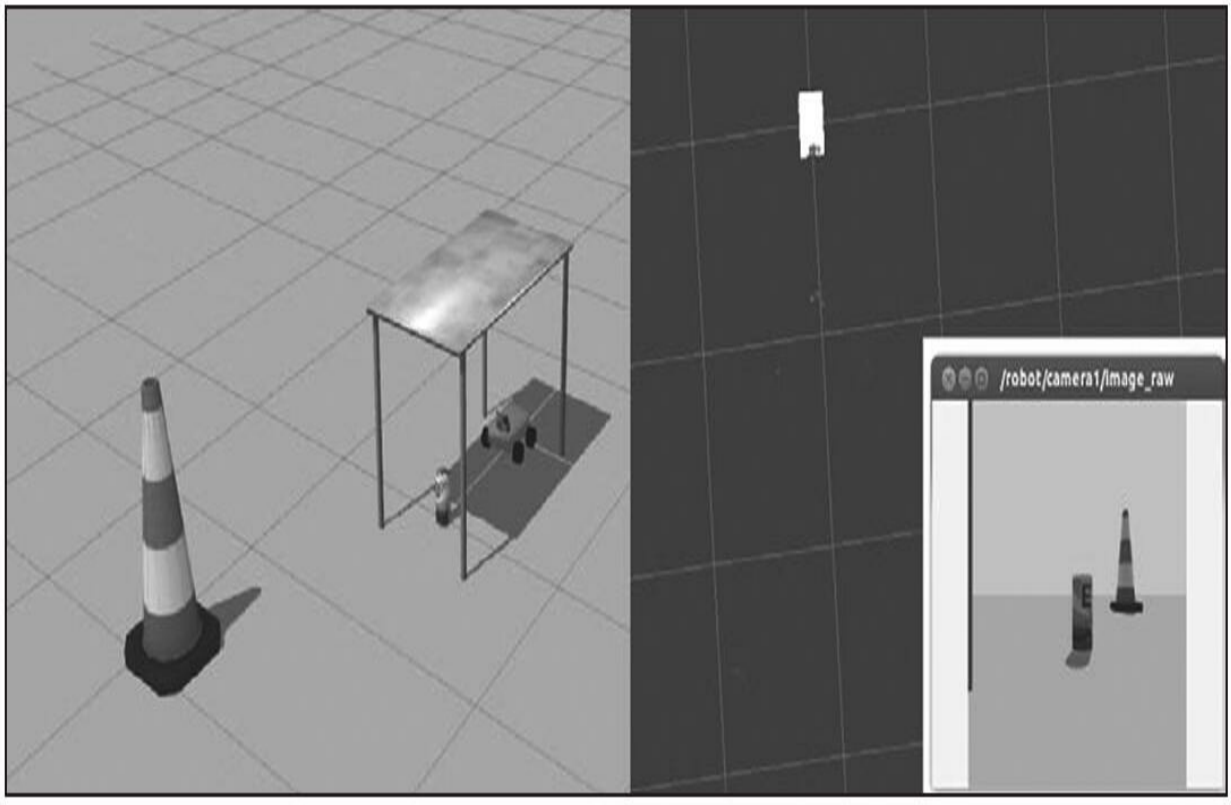
注意，这个激光会像真实的激光一样产生“真实”的传感器数据。你能够通过rostopic echo命令看到这些数据：

```
$ 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，可以在终端中写入以下指令：

```
$ rosrun 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及激光数据，最后是摄像头图像的可视化。



4.4.3 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。

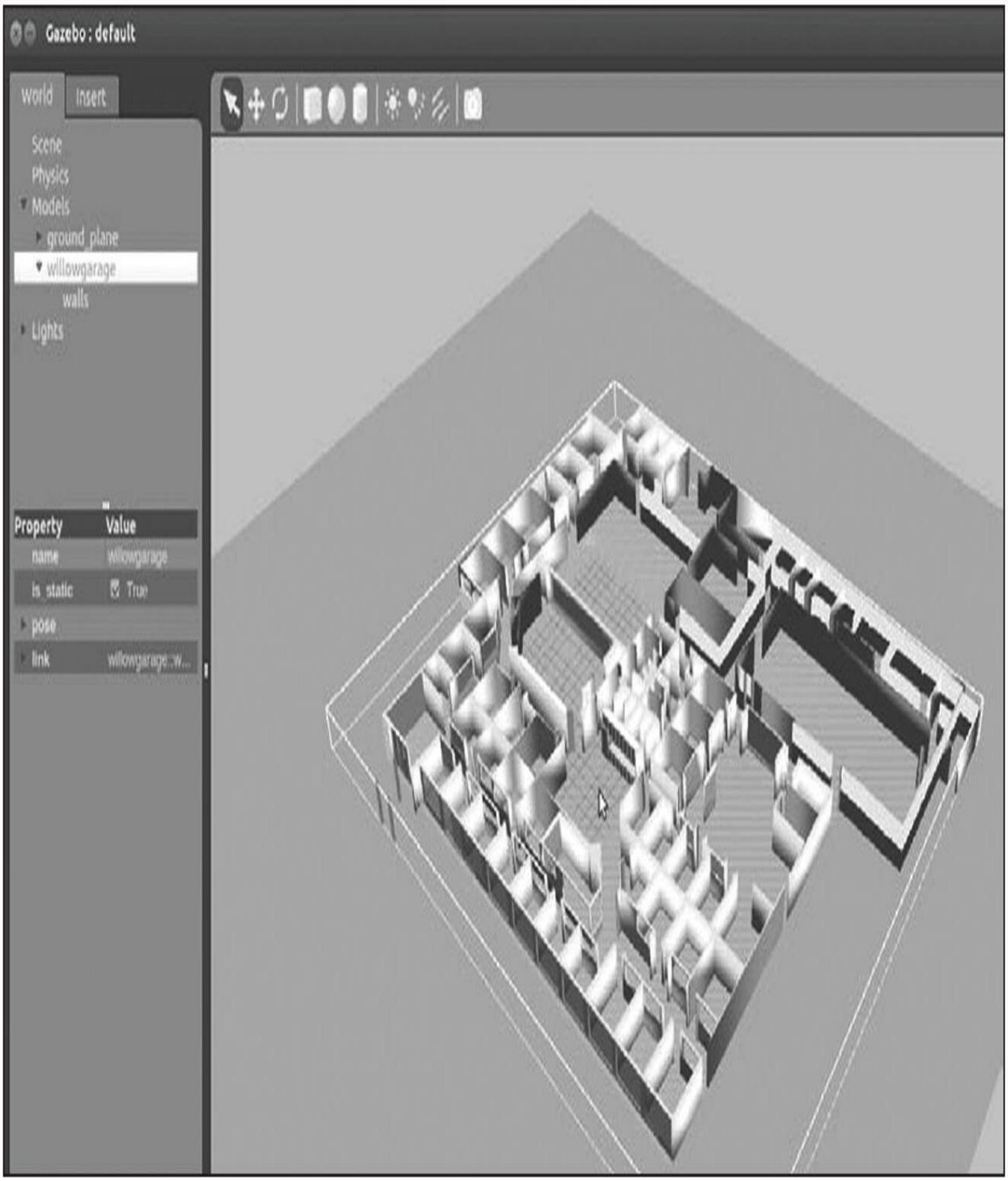
这个3D模型保存在gazebo_worlds功能包中。如果你没有安装这个功能包，请在进行以下步骤之前安装它。

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办公室。这个办公室只有墙壁。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官方网站上查看是否支持你的显卡。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，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。



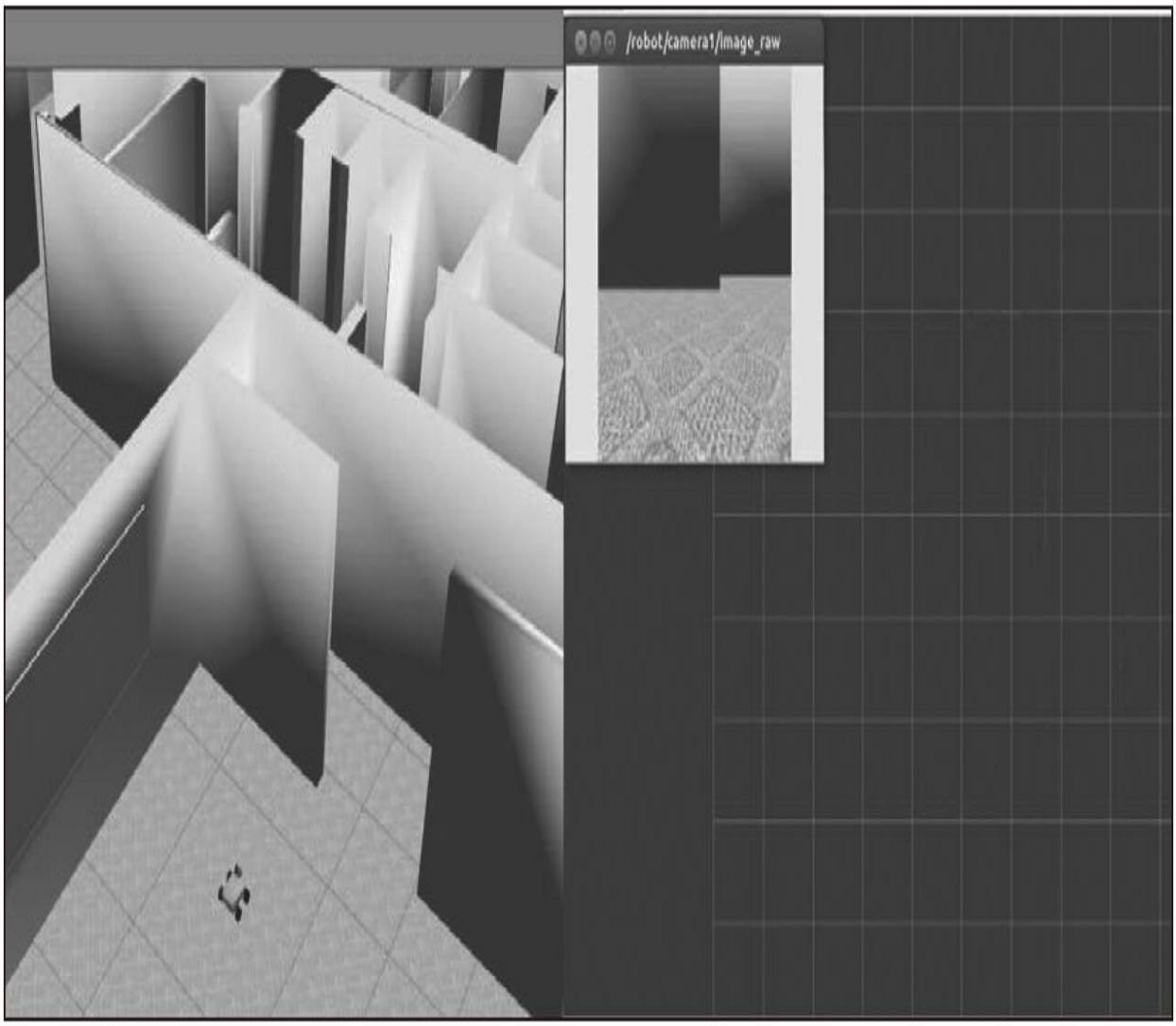
现在要做的是创建一个新的.launch文件来同时加载地图和机器人。为实现这个功能，在robot1_gazebo/launch文件夹下创建一个名为gazebo_wg.launch的文件，并添加以下代码：

```
<?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 -paramrobot_description -z 0.05"/>


</launch>
```

现在运行带有激光的模型文件：

```
$ roslaunch robot1_gazebo gazebo_wg.launch
```

你会在Gazebo GUI中看到机器人和地图。下一步是在仿真环境的虚拟世界中命令机器人移动和接收其传感器的仿真读数。


/robot/camera1/image_raw

4.4.4 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。和前面的激光一样，Gazebo也已经有了滑移（skid）驱动的实现，我们能够直接使用它移动机器人。

为使用此控制器，只需要在模型文件中增加以下代码：

```
<gazebo>
  <plugin name="skid_steer_drive_controller"
```

```
    filename="libgazebo_ros_skid_steer_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，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名称发布命令来控制机器人。此时，当发布一个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chapter4_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，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，它发布/cmd_vel主题。

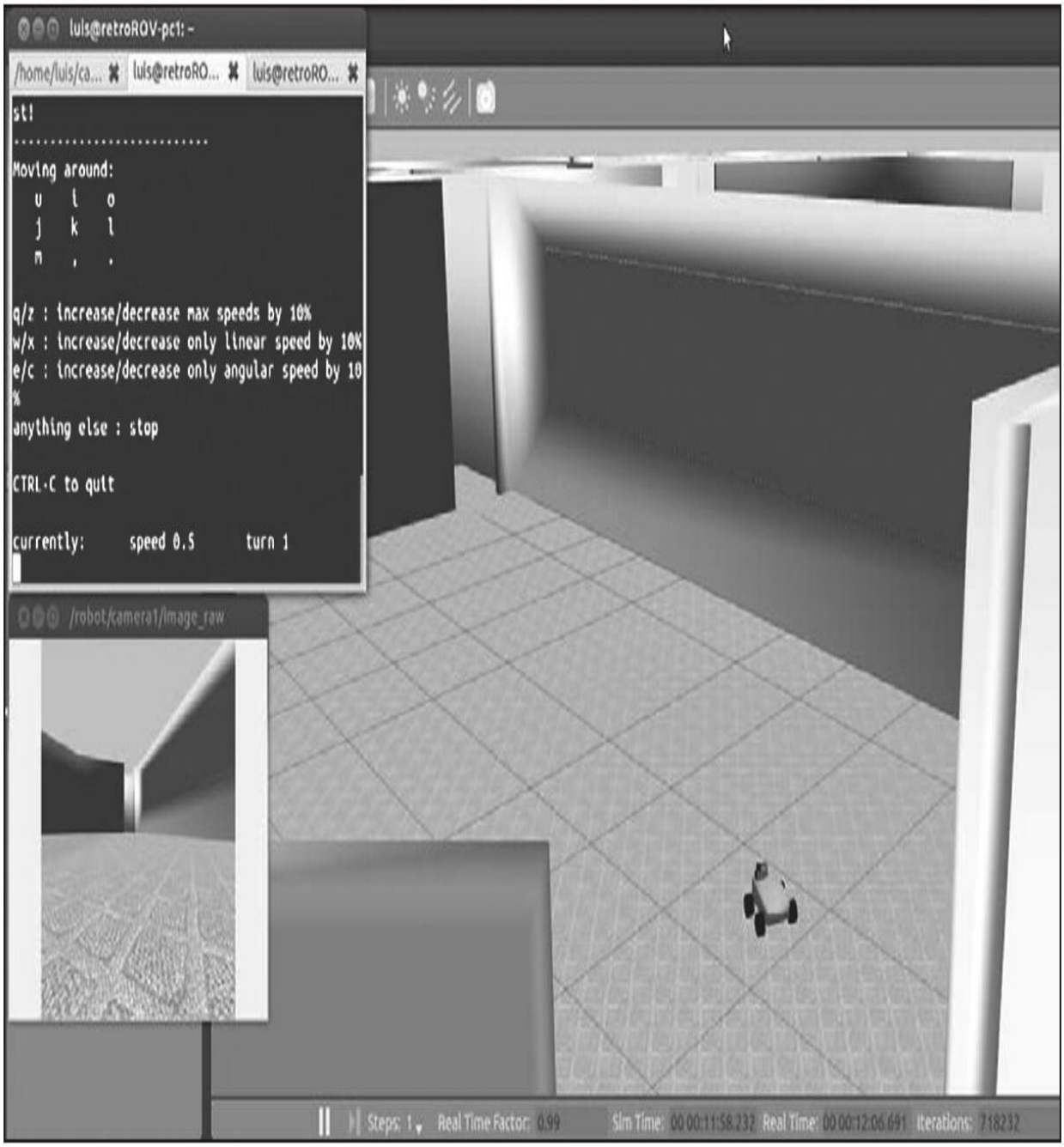
在终端中运行以下命令以安装此功能包：

```
$ sudo apt-get install ros-kinetic-teleop-twist-keyboard
$ rosstack profile
$ rospack profile
```

然后，运行节点，如下所示：

```
$ 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，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，你可以驾驶机器人穿越Willow Garage的办公室。你可以观察到激光数据和摄像头显示的图像。

4.5 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算法的行为，这也是仿真工具存在的原因。

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环境。这个仿真环境被ROS社区广泛使用并且它能够在仿真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，特别是我们已经有了抓取器并添加了传感器，例如一个激光测距仪和一个摄像头。

因此，在仿真环境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，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。

可以从http://www.ros.org/wiki/Robots找到ROS所支持的机器人列表。关于Gazebo的教程见http://gazebosim.org/tutorials。

第5章　导航功能包集入门

前面几章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最强大的特性之一——导航，它能够让你的机器人自主运动。

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，我们会学习在自己的机器人上配置导航功能包集所必需的步骤。在下一章中，将会学习在仿真的机器人上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。特别是，本章将会介绍以下内容。

·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。

·tf库，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是一个跟踪坐标系的库。

·创建一个激光驱动程序或对其进行仿真。

·计算并发布里程计（odometry）数据，以及Gazebo是如何提供结果的。

·基础控制器和如何为机器人创建基本控制器。

·用ROS执行同步定位与地图构建（Simultaneous Localization And Mapping，SLAM）。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。使用导航包集里的自适应蒙特卡罗定位（adaptive Monte Carlo localization，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。

5.1 ROS导航功能包集

为了能够理解导航功能包集，你应该把它看作一套使用机器人传感器和里程计功能的算法，以便使用标准的消息来控制机器人。它可以毫无难度地移动机器人到期望的位置（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，机器人需要满足一些条件。

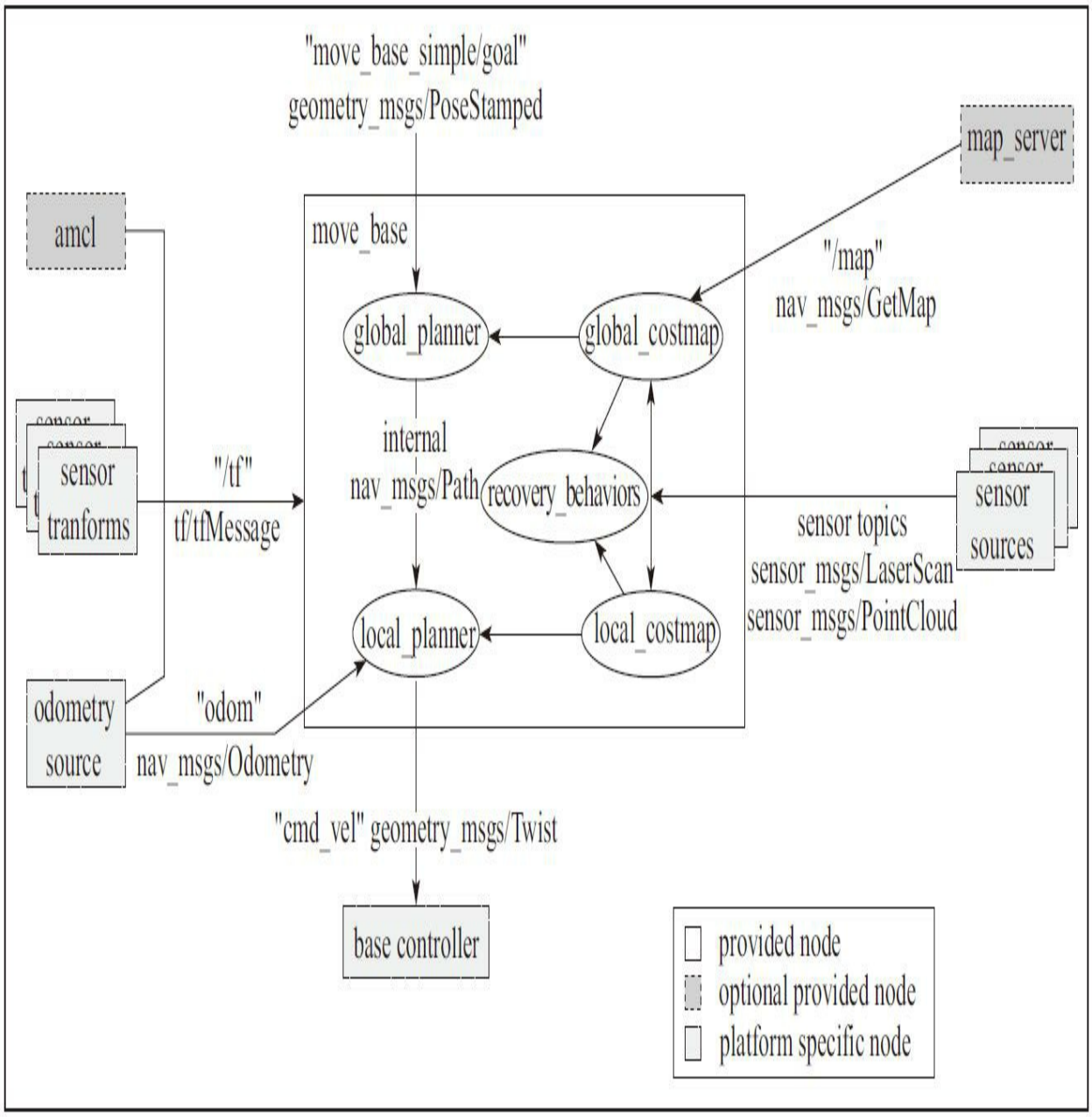
·导航功能包集只能够处理双轮差动（非完整系统）驱动和全向轮（完整系统）驱动的机器人。机器人的形状必须是方形或者矩形。当然，双足机器人也可以使用它做一些事情，如机器人定位等，但不能横向移动机器人。

·需要机器人能发布关于所有关节和传感器位置之间关系的信息。

·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来完成地图构建和定位。或者，你能生成一些相当于激光或者声呐的数据，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，需要你根据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。

5.2 创建变换

导航功能包集需要知道传感器、轮轴和关节的位置。

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。也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。

得益于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量的新坐标系到坐标变换树中。

一旦插入和创建完成，我们就能轻松地知道激光相对于base_link值或者相对于轮子的位置。我们唯一需要做的就是调用tf库并进行坐标变换。

5.2.1 创建广播器

让我们用一些简单的代码测试一下。在chapter5_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。

5.2.2 创建侦听器

在chapter5_tutorials/src文件夹下以tf_listener.cpp为名称创建一个新的文件，并将以下代码复制到文件中：

```
#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，还要添加一行代码到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
$ rosrun chapter5_tutorials tf_broadcaster
$ rosrun chapter5_tutorials tf_listener
```

不要忘记每次在运行示例前要先运行roscore，然后会看到如下消息：

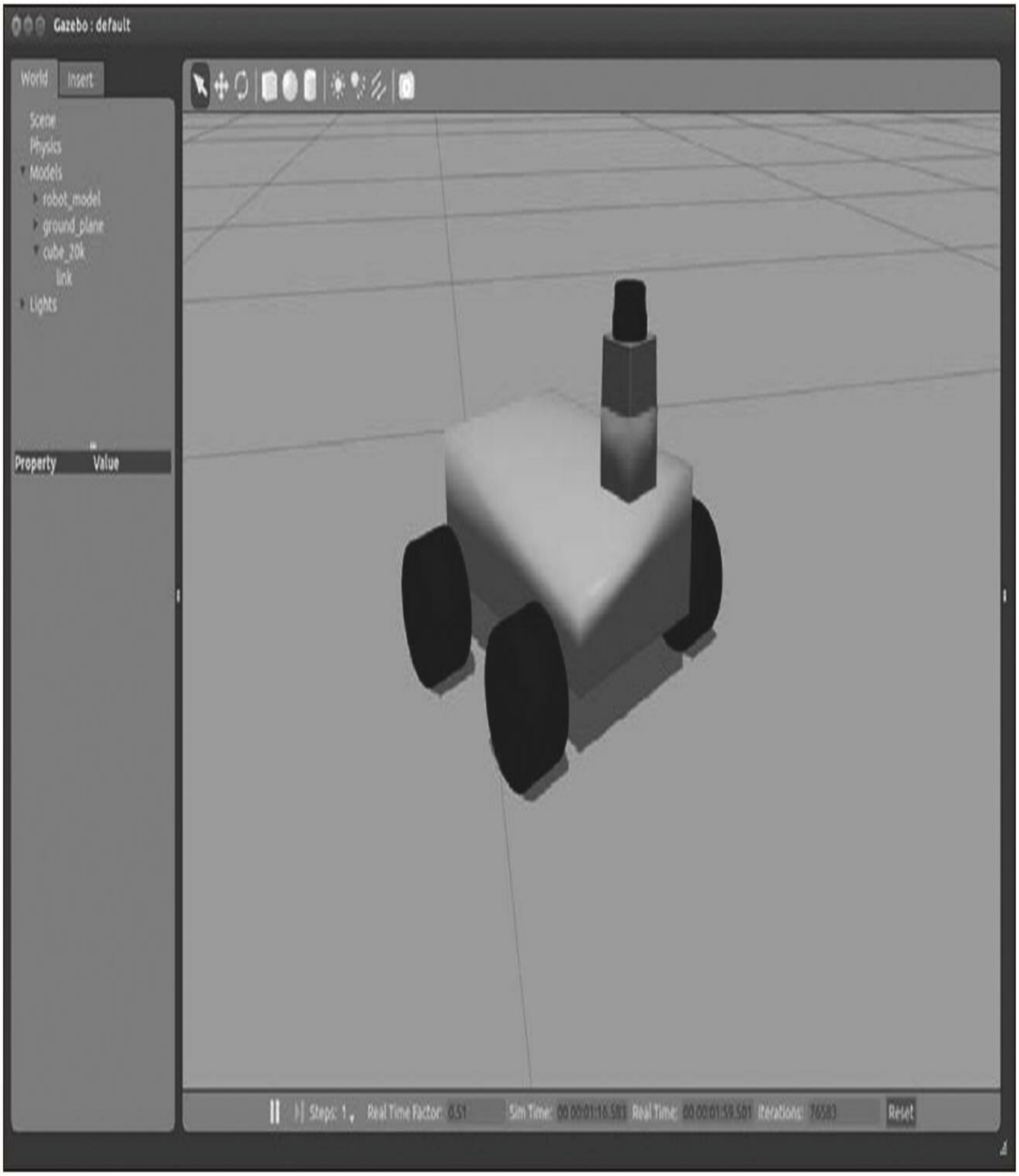
```
[ INFO] [1368521854.336910465]: base_laser: (1.00, 2.00. 0.00) ----->
base_link: (1.10, 2.00, 0.20) at time 1368521854.33
[ INFO] [1368521855.336347545]: base_laser: (1.00, 2.00. 0.00) ----->
base_link: (1.10, 2.00, 0.20) at time 1368521855.33
```

这表示你在节点上发布的点若以base_laser坐标系为参照，则位于（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库自动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。

使用第4章中的模型，可以再添加一个激光，比如放在机器人（base_link）的后面。



机器人系统必须知道这个新激光的具体位置，包括和轮子、墙面之间的相对位置，只有这样才能够避免碰撞。通过tf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。如果想要添加一个传感器或者机器人组件都非常方便，因为tf库会自动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tf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，并准确无误地移动机器人。

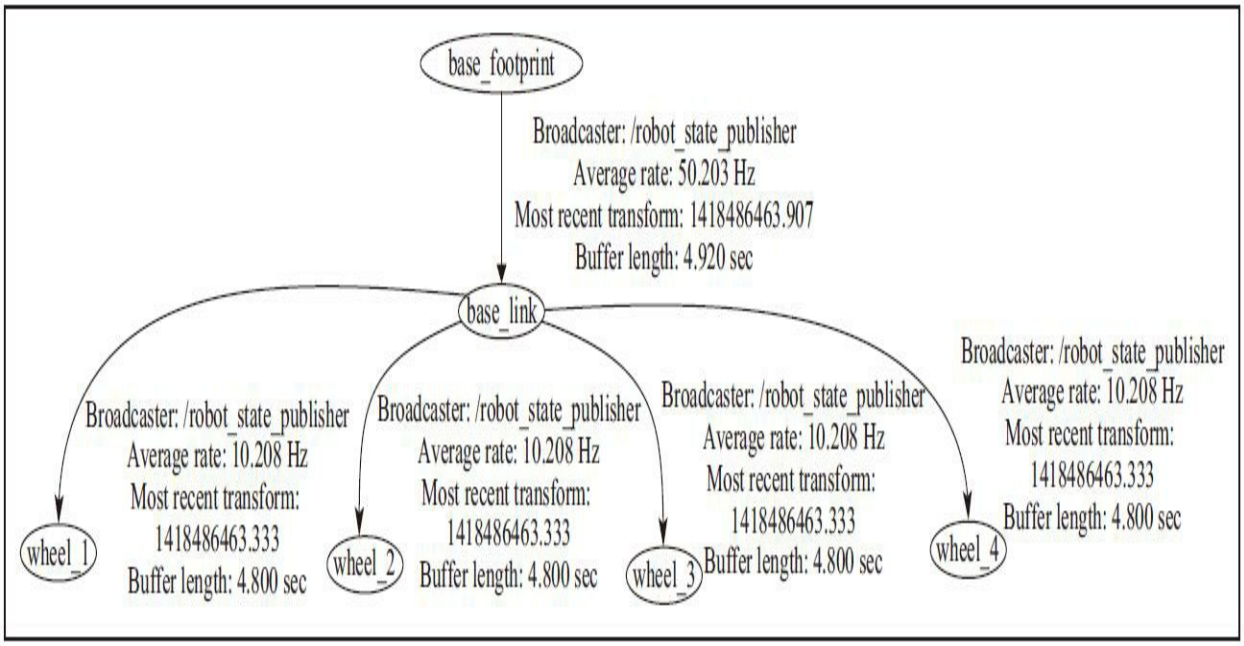
在开始编写代码进行组件配置之前，你要记住，首先要有完整的机器人几何外形的定义文件（URDF文件）。而且你可能已经忘记了，我们甚至使用robot_state_publisher功能包发布过机器人的坐标变换树。在第4章中我们首次使用它。这样，其实你已经有了一个能够应用导航功能包集的机器人配置。基于以上原因，没有必要在这里再对机器人配置进行介绍和说明了。

5.2.3 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 chapter5_tutorials gazebo_map_robot.launch model:="'rospack
find chapter5_tutorials'/urdf/robot1_base_01.xacro"

$ rosrun tf view_frames
```

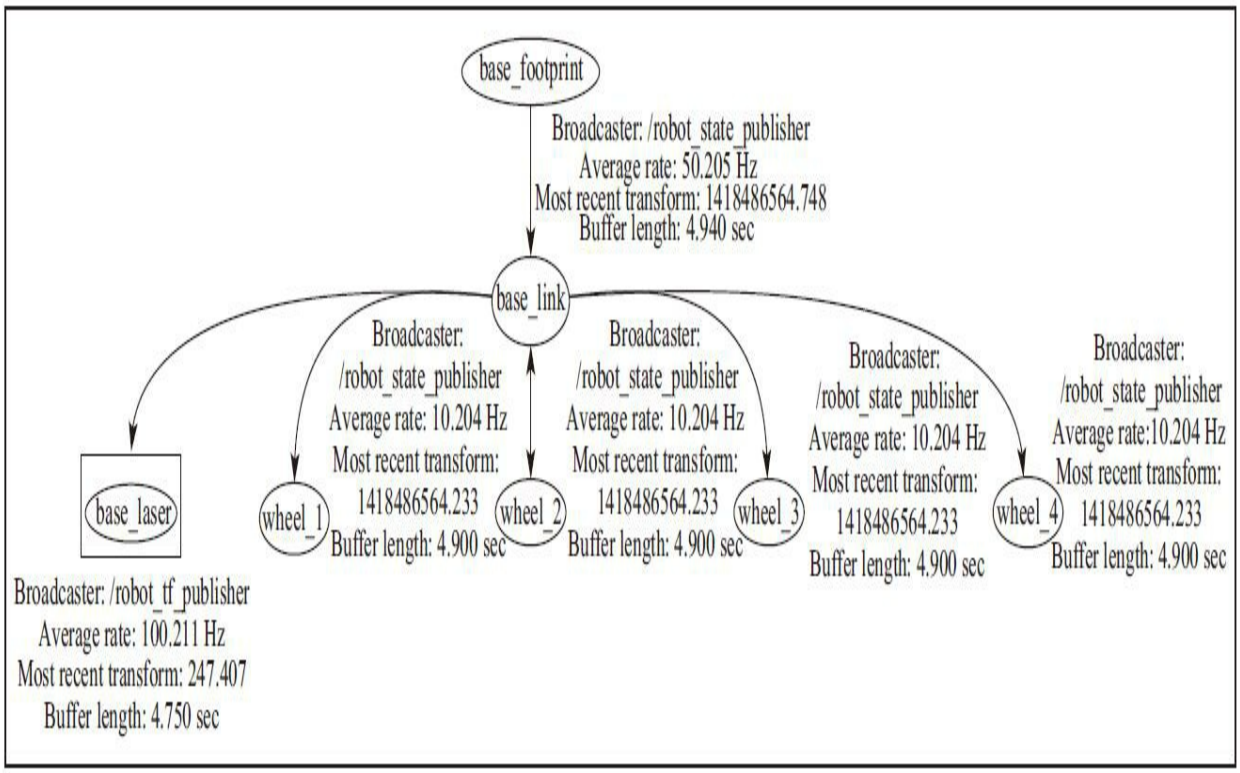
所生成的坐标系结构如下图所示。



现在，如果再一次运行tf_broadcaster和rosrun tf view_frames命令，你会看到你通过代码创建的坐标系：

```
$ rosrun chapter5_tutorials tf_broadcaster

$ rosrun tf view_frames
```

所生成的坐标系结构如下图所示。


base_footprint
Broadcaster: /robot_state_publisher
Average rate: 50.205 Hz
Most recent transform: 1418486564.748
Buffer length: 4.940 sec
base_link
base_laser
Broadcaster: /robot_tf_publisher
Average rate: 100.211 Hz
Most recent transform: 247.407
Buffer length: 4.750 sec
wheel_1
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 1418486564.233
Buffer length: 4.900 sec
wheel_2
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 1418486564.233
Buffer length: 4.900 sec
wheel_3
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 1418486564.233
Buffer length: 4.900 sec
wheel_4
Broadcaster: /robot_state_publisher
Average rate:10.204 Hz
Most recent transform: 1418486564.233
Buffer length: 4.900 sec

5.3 发布传感器信息

机器人可以使用很多类型的传感器来感知世界，可以编写对应的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传感器。所以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进行导航。记住，这个激光是在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激光了。这是因为在.urdf文件中配置过tf树，而且激光也能够使用正确的格式发布数据。

如果使用一个真实的激光，ROS很可能已经包含了它的驱动程序。请参考第8章中如何在ROS中使用Hokuyo激光的相关内容。但你还是有可能用到一个ROS没有提供官方驱动程序的激光，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。下一节会专门介绍相关功能的实现，并提供一个示例模板。

首先，请记住消息的类型是sensor_msgs/LaserScan。使用以下命令：

```
$ rosmsg 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
```

创建激光节点

现在在chapter5_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();
  }
}
```

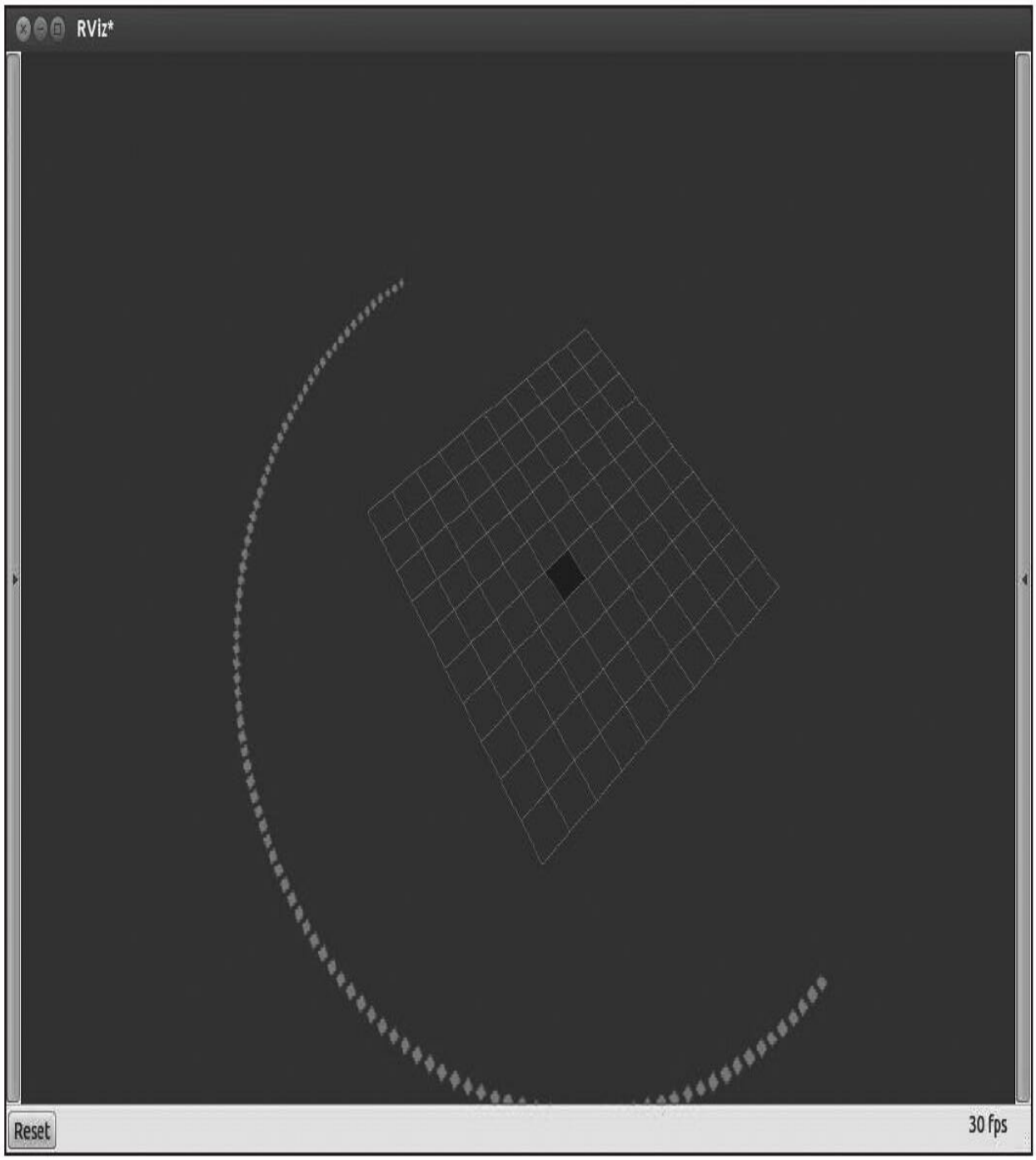
正如你所见，我们将要以scan为名创建一个新的主题，其消息格式为sensor_msgs/LaserScan。你应该对这种消息类型比较熟悉，它将在第8章介绍。主题的名称必须是唯一的。当配置导航功能包集时，需要选择这个主题来进行导航。以下代码显示了如何使用正确的名称创建主题：

```
ros::Publisher scan_pub =
n.advertise<sensor_msgs::LaserScan>("scan", 50);
```

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坐标系变换中。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。



通过这个示例模板，可以使用任何激光，而无须考虑ROS中是否有它的驱动程序。你只需要用你激光上的真实数据来替代模板中的虚假数据。

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。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。

5.4 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。在本例中，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。可以使用以下指令查看消息的数据结构：

```
$ rosmsg show nav_msgs/Odometry
```

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。

位姿信息中包含两个结构，一个显示了欧氏坐标系中的位置，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偏移。

速度信息包含两个结构，一个是线速度，另一个是角速度。对于这里的机器人，我们经常使用的是线速度x和角速度z。使用线速度x是为了知道机器人是向前移动还是向后移动。使用角速度z是为了知道机器人是向左转还是向右转。

```
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
```

因为里程其实就是两个坐标系之间的位移，所以我们就有必要发布两个坐标系之间的坐标变换信息。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。

5.4.1 Gazebo如何获取里程数据

正如在其他关于Gazebo的示例中所见的，这里的机器人在仿真环境中的移动和现实世界中的机器人是相似的。机器人使用的驱动程序插件是diffdrive_plugin。我们曾经在第4章中配置过这个插件。当时，在Gazebo中创建了机器人并通过这个插件驱动它。

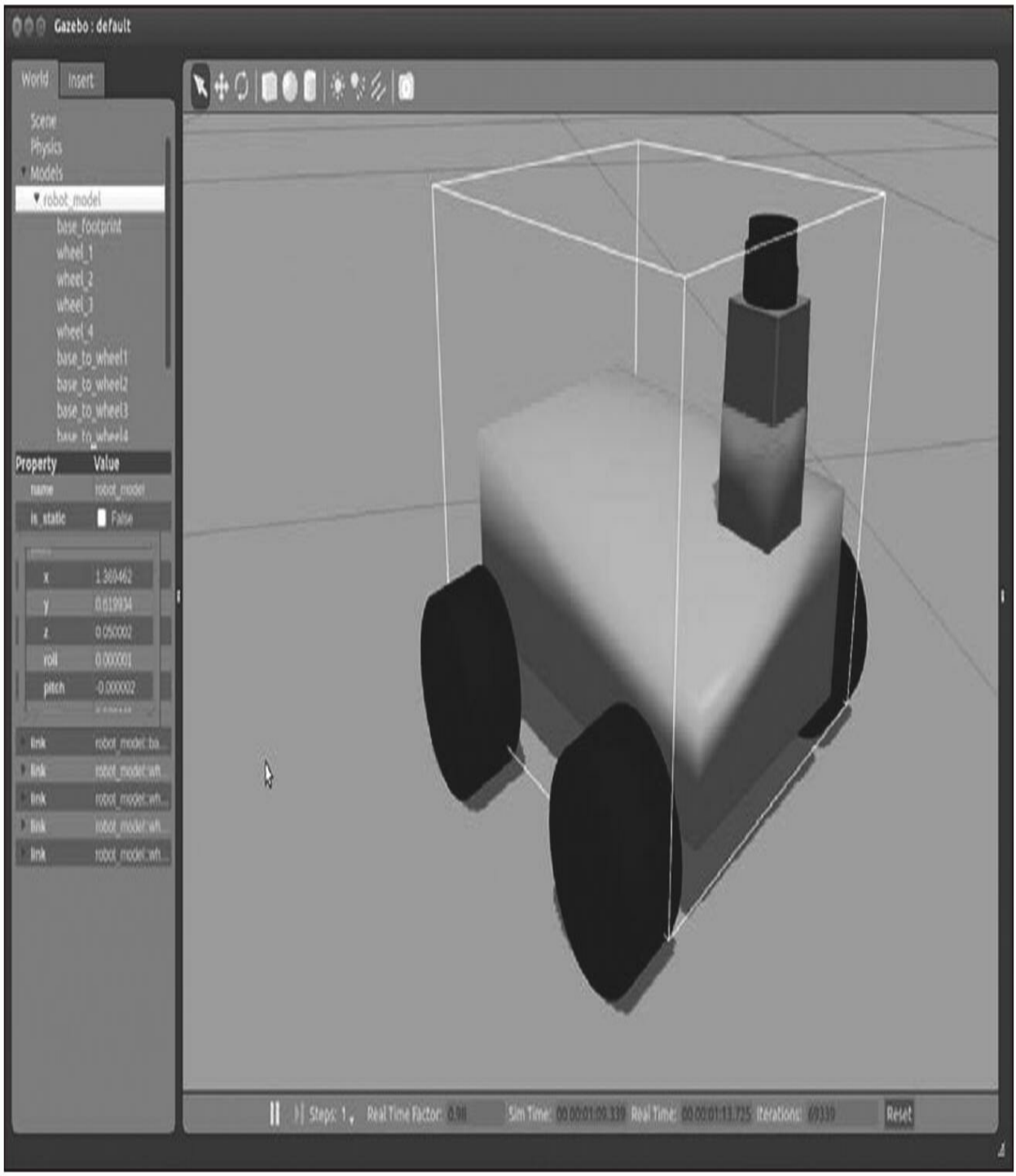
这个驱动程序就会发布机器人在仿真环境中的里程信息，所以我们并不需要为Gazebo再编写任何代码。

在Gazebo中执行示例机器人，并查看里程数据。在命令行窗口中输入以下命令：

```
$ roslaunch chapter5_tutorials gazebo_xacro.launch model:="'rospack find
robot1_description'/urdf/robot1_base_04.xacro"

$ rosrun 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，如果你单击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）。单击位姿，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。可以单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。

正如你所看到的，Gazebo在机器人移动时会发布里程数据。我们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/rossimulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_plugins/src/gazebo_ros_skid_steer_drive.cpp。

这个文件有大量代码，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为里程设定主题的名称（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。这会在后面的小节中进行具体介绍。

一旦懂得了Gazebo如何和在哪里获取里程数据，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。

5.4.2 使用Gazebo创建里程数据

要清楚地知道Gazebo的内部原理及如何创建里程数据，就需要看一下diffdrive_plugin.cpp文件。下载地址：https://github.com/ros-simulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_plugins/src/gazebo_ros_skid_steer_drive.cpp。

Load函数会完成对主题订阅者的注册，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_);
  …
  …
}
```

当消息收到时，线速度和角速度都存储在内部变量中以便于之后的运行操作：

```
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，如下面的函数所示：

```
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，它使用基于机器人运动学模型的更多公式来估计机器走过的路径。正如代码所示，需要知道机器人的车轮直径（wheel diameter）和车轮间距（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控制Gazebo中仿真机器人的方式。

5.4.3 创建自定义里程数据

在chapter5_tutorials/src文件夹下以odometry.cpp为名创建一个新的文件，并加入以下代码以发布里程数据：

```
#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);
  ...

  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::createQuaternionMsgFromYaw(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.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。在此示例中，基础坐标系base_footprint将会相对于odom坐标系移动：

```
geometry_msgs::TransformStamped odom_trans;
odom_trans.header.frame_id = "odom";
odom_trans.child_frame_id = "base_footprint";
```

在这段代码中，还会生成机器人的位姿信息。然后根据线速度和角速度还能够计算一段时间后机器人的理论位置：

```
double dt = (current_time - last_time).toSec();
double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
double delta_th = vth * dt;

x += delta_x;
y += delta_y;
th += delta_th;

geometry_msgs::Quaternion odom_quat;
odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);
```

在这个坐标变换结构体中，将只给x和rotation字段赋值，因为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两个结构体。为pose结构体中的x、y和orientation字段分别赋值。在twist结构体中，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，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})
```

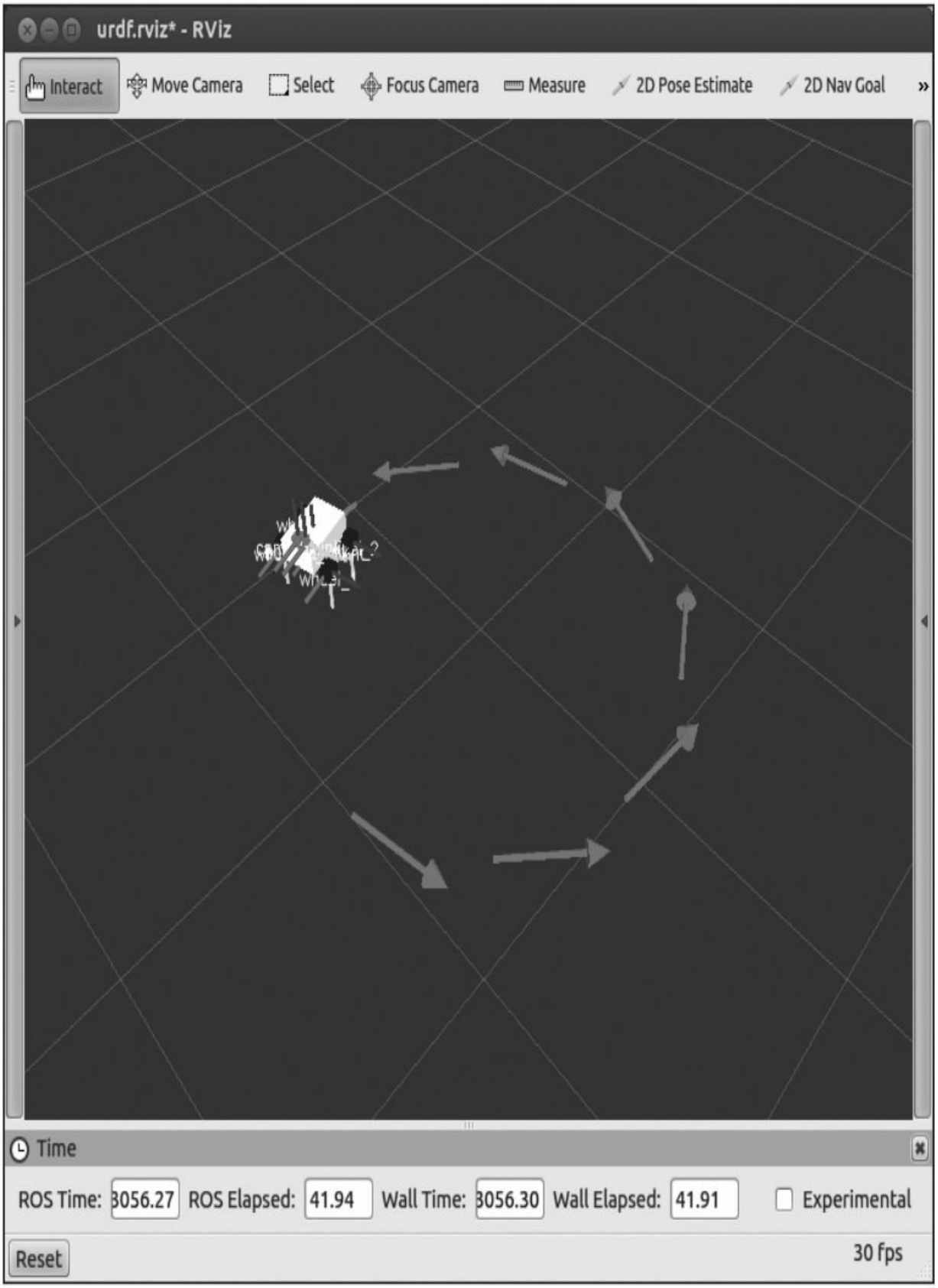
编译完功能包之后，不使用Gazebo而只使用rviz来可视化机器人模型及其运动。运行以下命令：

```
$ roslaunch chapter5_tutorials display_xacro.launch model:="'rospack find
chapter5_tutorials'/urdf/robot1_base_04.xacro"
```

使用以下命令运行odometry节点：

```
$ rosrun chapter5_tutorials odometry
```

输出如下图所示。


urdf.rviz* - RViz
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Time
ROS Time: 3056.27
ROS Elapsed: 41.94
Wall Time: 3056.30
Wall Elapsed: 41.91
Experimental
Reset
30 fps

在rviz的屏幕上，能看到机器人沿着一些红色的箭头移动（见彩插4），并穿过不同的背景网格。机器人穿过网格说明了系统正在为机器人发布新的tf坐标系变换。红色的箭头是图形化表示的odometry消息。如果一直运动下去，正如代码中定义的那样，你会看到机器人围绕着一个圆形不断地运动。

5.5 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元素，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，因此必须自己编写针对移动平台的基础控制器。

机器人通过geometry_msgs/Twist类型的消息进行控制。这个类型正是我们之前看到的Odometry消息所使用的。

所以基础控制器必须订阅名称为cmd_vel的主题，必须生成正确的线速度和角速度命令来驱动平台。

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

现在我们先复习一下消息的结构。在命令行窗口内输入以下命令查看消息的具体结构：

```
$ rosmsg show geometry_msgs/Twist
```

这个命令的输出结果如下所示。

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。

对于这里的机器人，只需要使用线速度x和角速度z。这是因为机器人基于差动轮驱动平台，驱动它的两个电动机只能够让机器人前进、后退或者转向。

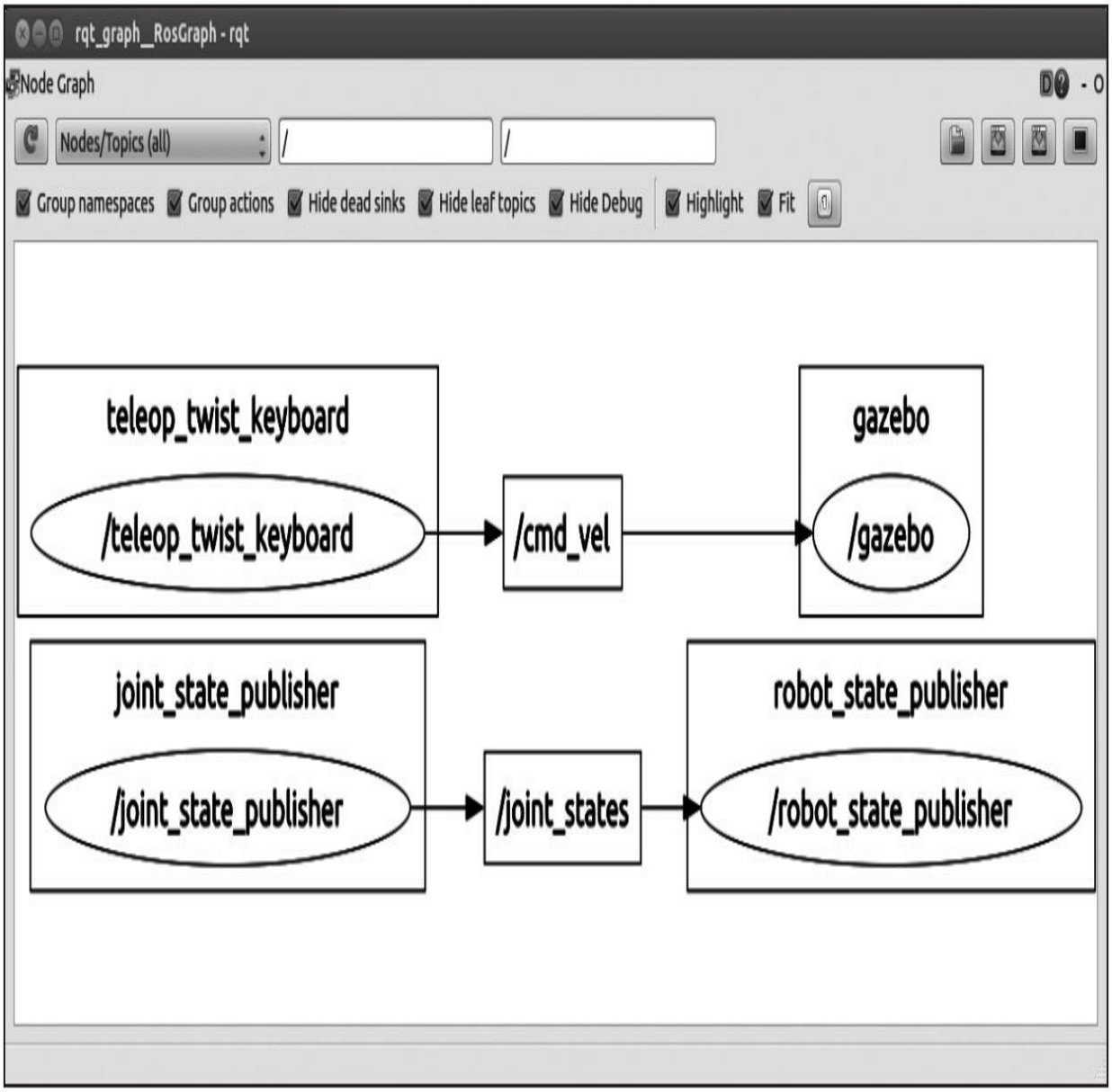
我们先在Gazebo中对这个机器人进行仿真，用于机器人运动/仿真的基础控制器在驱动程序中实现。这也就意味着在Gazebo中不必为机器人实际创建一个基础控制器。

在本节中，还会学习实现一个实际的物理机器人的基础控制器的示例。在此之前，先在Gazebo中运行机器人，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 chapter5_tutorials gazebo_xacro.launch model:="'rospack find
robot1_description'/urdf/robot1_base_04.xacro"

$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

在所有节点都启动和正常运行之后，使用节点状态图命令rqt_graph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teleop节点生成的cmd_vel主题。

在Gazebo仿真环境中，正运行着差动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。

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chapter5_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;
```

在这部分代码中，声明了全局变量，包括一些库以计算里程数据，并且进行空间变换来定位机器人。现在，应该创建一个回调函数接收速度命令。应用一些运动学方程，可以将速度命令与机器人每个轮子的速度相关联。

```
void cmd_velCallback(const geometry_msgs::Twist &twist_aux)
{
  geometry_msgs::Twist twist = twist_aux;
  double vel_x = twist_aux.linear.x;
  double vel_th = twist_aux.angular.z;
  double right_vel = 0.0;
  double left_vel = 0.0;

  if(vel_x == 0){
    // turning
    right_vel = vel_th * width_robot / 2.0;
    left_vel = (-1) * right_vel;
```

```
  }else if(vel_th == 0){
    // forward / backward
    left_vel = right_vel = vel_x;
  }else{
    // moving doing arcs
    left_vel = vel_x - vel_th * width_robot / 2.0;
    right_vel = vel_x + vel_th * width_robot / 2.0;
  }
  vl = left_vel;
  vr = right_vel;
}
```

在main函数中，将编写一个循环，该循环将使用来自编码器的数据更新机器人的实际速度，并计算里程数据。

```
int main(int argc, char** argv){
  ros::init(argc, argv, "base_controller");
  ros::NodeHandle n;
  ros::Subscriber cmd_vel_sub = n.subscribe("cmd_vel", 10,
  cmd_velCallback);
  ros::Rate loop_rate(10);

  while(ros::ok())
  {

    double dxy = 0.0;
    double dth = 0.0;
    ros::Time current_time = ros::Time::now();
    double dt;
    double velxy = dxy / dt;
    double velth = dth / dt;

    ros::spinOnce();
    dt = (current_time - last_time).toSec();;
    last_time = current_time;

    // calculate odomety
    if(right_enc == 0.0){
  distance_left = 0.0;
  distance_right = 0.0;
}else{
  distance_left = (left_enc - left_enc_old) / ticks_per_meter;
  distance_right = (right_enc - right_enc_old) / ticks_per_meter;
}
```

一旦知道每个轮子所经过的距离，就可以计算距离dxy的增量和角度dth的增量从而更新机器人位置。

```
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;
}
```

在x和y中计算机器人的位置，对于2D机器人平台的基础控制器，将假设z为常量0。对于机器人方位，可以假定俯仰和滚转也等于0，并且只需更新偏航角。

```
    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。第8章会通过一个具有轮子和编码器的真实机器人平台给出一个功能齐全的示例。

5.6 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传感器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chapter5_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>
```

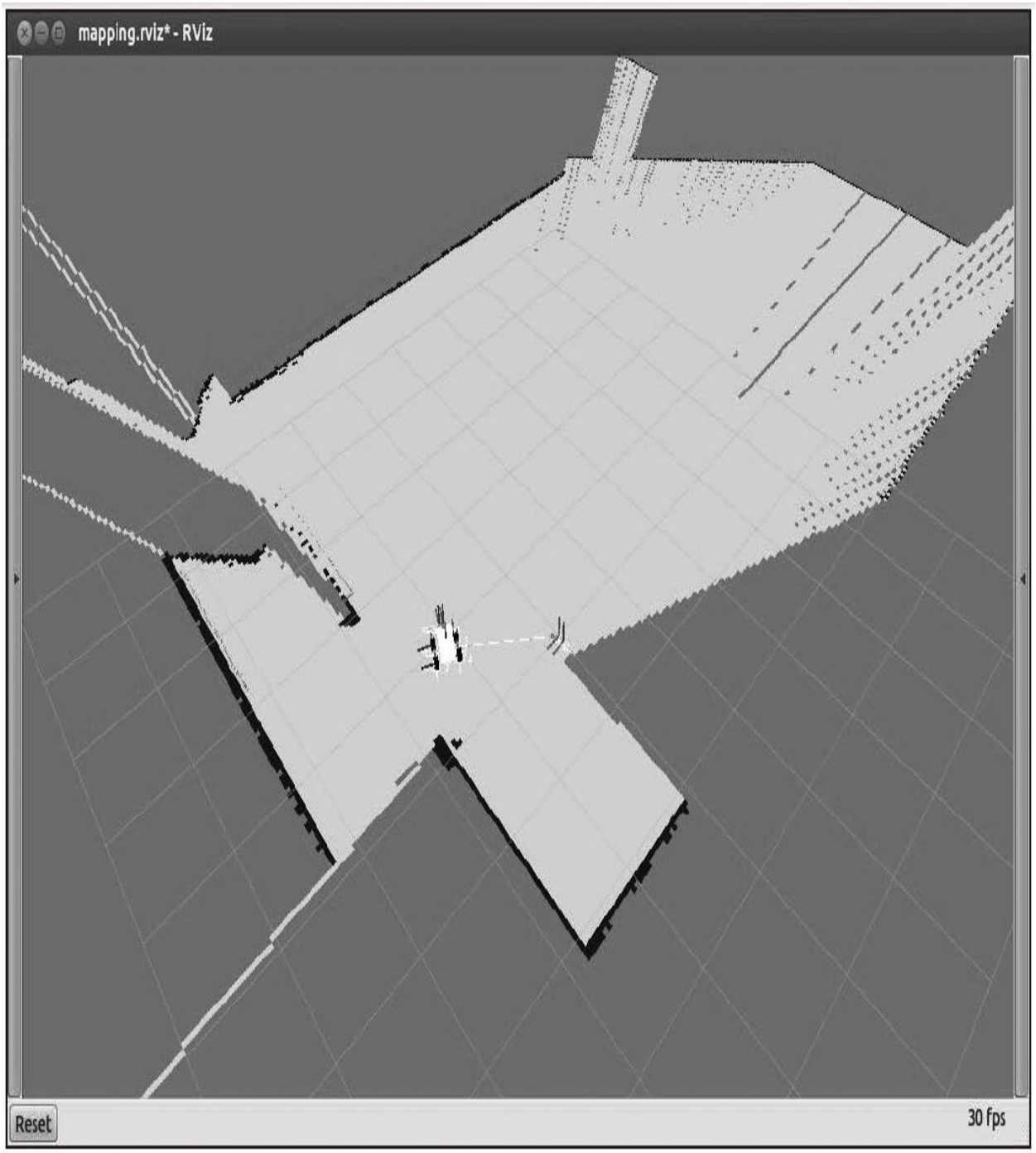
```
<!-- start robot state publisher -->
<node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen" >
<param name="publish_frequency" type="double" value="50.0" />
</node>
<node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
args="-urdf -param robot_description -z 0.1 -model robot_model"
respawn="false" output="screen" />
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find
chapter5_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配置文件和通过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teleop节点来移动机器人：

```
$ roslaunch chapter5_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。机器人使用激光扫描和里程计来构建OGM。



5.6.1 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，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。可以使用以下命令来保存地图：

```
$ rosrun map_server map_saver -f map
```

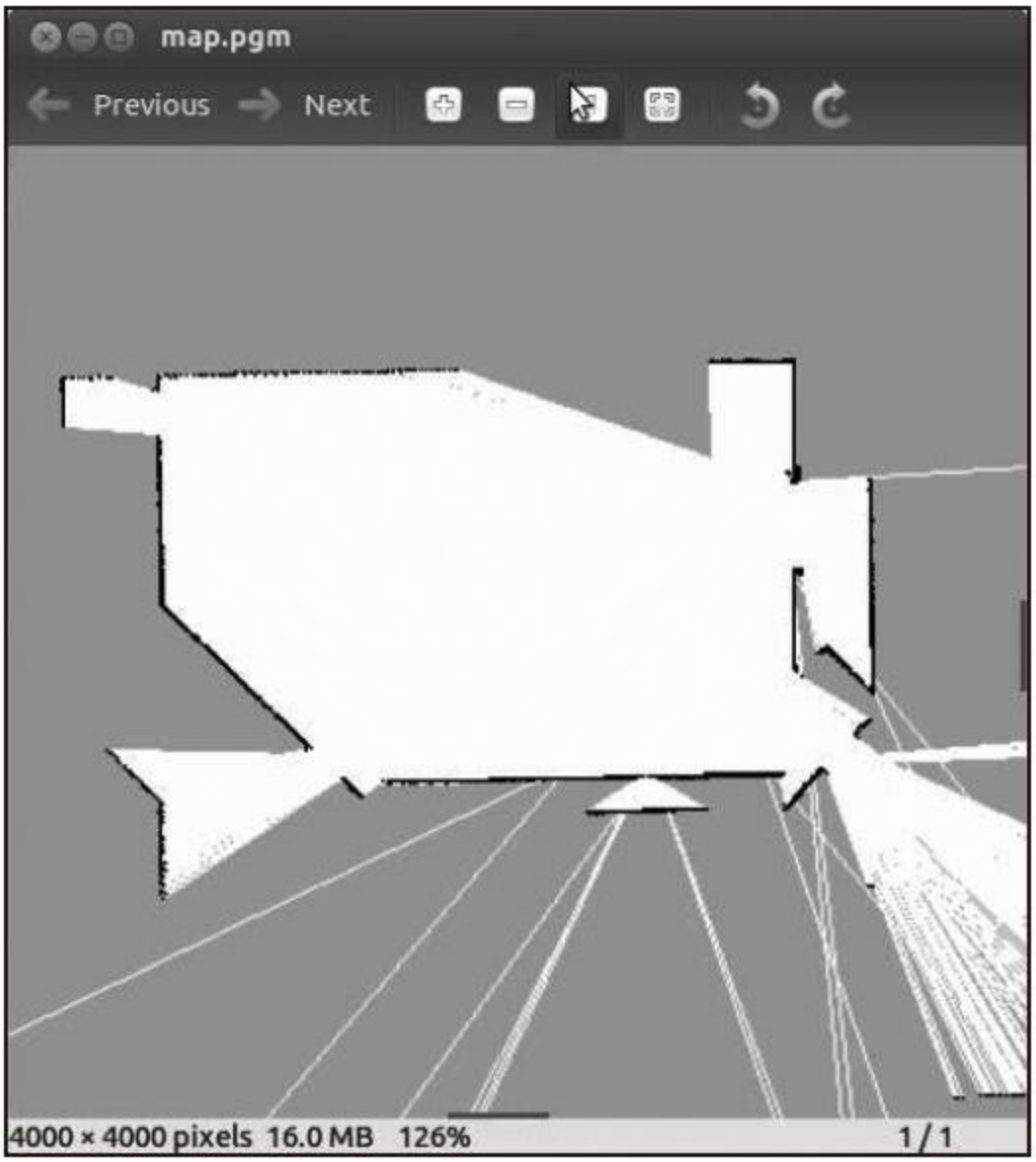
命令输出如下：

```
[ 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，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：


map.pgm
Previous
Next
4000 × 4000 pixels 16.0 MB 126%
1 / 1

5.6.2 使用map_server加载地图

当想要使用你自己的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
```

但是为了简化工作，也可以在chapter5_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 chapter5_tutorials)/maps/map.yaml" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
  chapter5_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 chapter5_tutorials gazebo_map_robot.launch model:="'rospack
find chapter5_tutorials'/urdf/robot1_base_04.xacro"
```

然后，你就会在rviz中看到机器人和地图。为了能够获取机器人的位置，导航功能包集会使用地图服务器发布的地图数据和激光的读数。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）节点中实现的粒子滤波估计机器人的位置时有帮助。

在下一章中，我们会学习关于地图的更多知识和更多有用的工具。

5.7 本章小结

在本章中，为了使用导航功能包集，你逐步学习如何逐步配置机器人。介绍了应用导航功能包集对机器人平台的需求，即需要使用平面激光、差动轮式机器人、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同的，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，在满足需要的基础之上，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6章　导航功能包集进阶

我们已经创建了功能包、节点、机器人的3D模型等。在第5章中，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。

之前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5章所学的知识，并继续进行程序开发。

·理解导航功能包集及其工作方式。

·配置所有必要文件。

·创建启动文件以启动导航功能包集。

让我们开始吧。

6.1 创建功能包

正确创建功能包，并不是仅仅把文件放到一起，而是要为机器人创建的功能包添加依赖项。例如，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
```

但是在本例中，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 chapter6_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。

6.2 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>
```

这个启动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。如果配置的传感器要更多的参数，那么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协助导航的多个传感器。在这样的情况下，可以为这些传感器添加更多的节点，并在创建启动文件时，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、底盘控制、机器人移动和定位等必需的功能。第5章在不同程度介绍过这些内容。正如在其他小节一样，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使用真实机器人时使用，而对于现在的示例，下面这个启动文件包括了所有我们需要的。

在chapter6_tutorials/launch文件夹下以chapter6_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
```

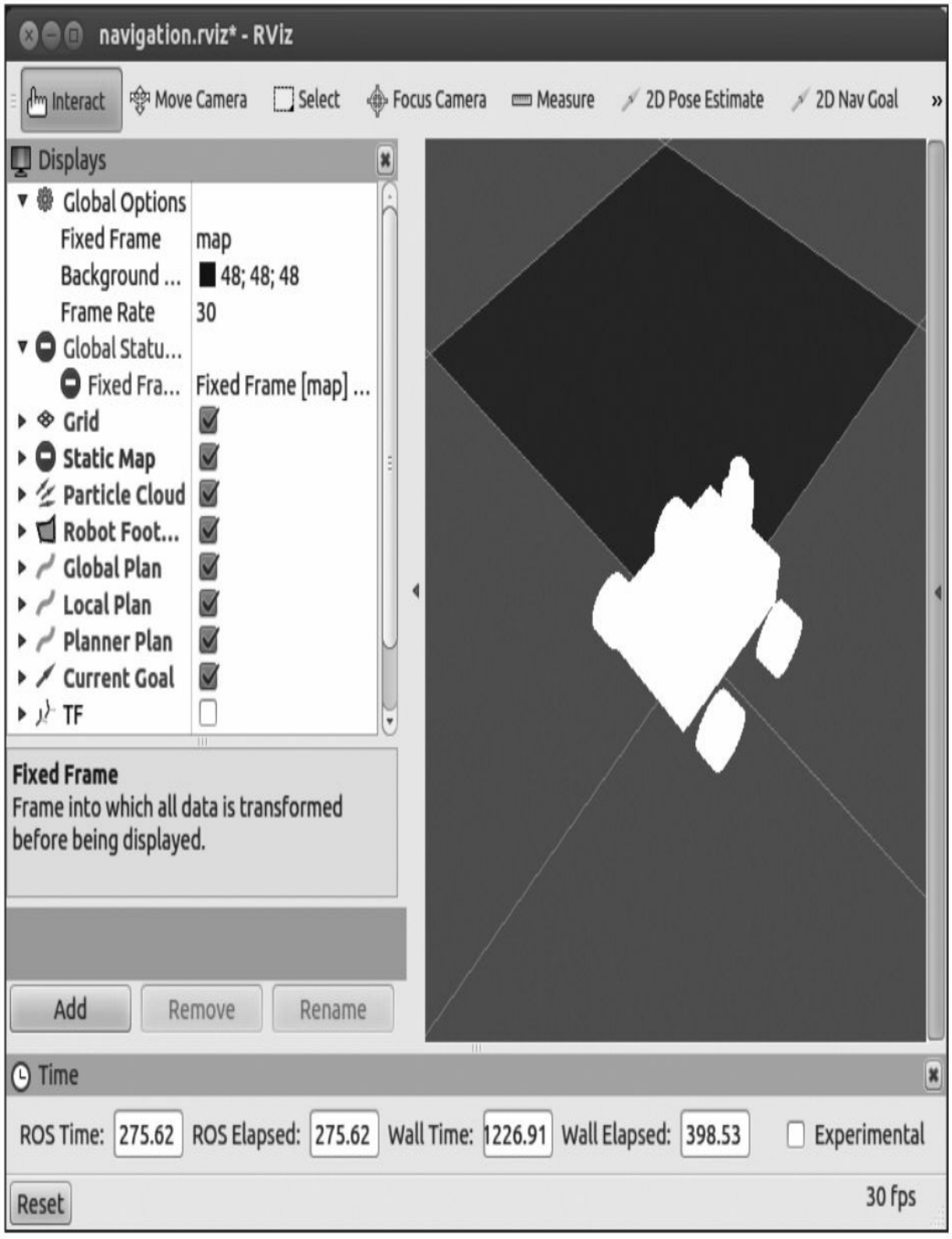
```
  robot1_description)/urdf/robot1_base_04.xacro"/>
  <param name="robot_description" command="$(find xacro)/xacro.py
  $(arg model)" />
  <node name="joint_state_publisher" pkg="joint_state_publisher"
  type="joint_state_publisher" ></node>
  <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
  name="robot_state_publisher" output="screen" />
  <node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
  args="-urdf -paramrobot_description -z 0.1 -model robot_model"
  respawn="false" output="screen" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
  chapter6_tutorials)/launch/navigation.rviz" />
</launch>
```

这个启动文件其实和前面几章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：

```
$ ros launch chapter6_tutorials chapter6_configuration_gazebo.launch
```

你会看到如下窗口。



请注意，在前面的截屏中，在没有进行任何配置时，有一些字段显

示出红色、蓝色和黄色的底纹（见彩插5）。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看所有的主题。

6.3 配置全局和局部代价地图

现在我们将要开始配置导航功能包集和启动它时所有必需的文件。为了开始进行配置，首先我们将会学习什么是代价地图（costmap）和代价地图的作用。机器人将使用两种导航算法在地图中移动——全局（global）导航和局部（local）导航。

·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4m×4m的方形窗户。

这些导航算法通过使用代价地图来处理地图中的各种信息。全局代价地图用于全局导航，局部代价地图用于局部导航。

代价地图的参数用于配置算法计算行为。它们也有最基本的通用参数，这些参数会保存在共享的文件中。

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。

6.3.1 基本参数的配置

从最基本的参数讲起。在chapter6_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物信息。前者主要用于障碍物的探测。在本例中，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，可以通过修改参数设计一个激进或保守的行为。

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。

在本例中，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用于在代价地图中添加和清除障碍。例如，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。

6.3.2 全局代价地图的配置

下一个待配置的文件是全局代价地图。在chapter6_tutorials/launch文件夹下以global_costmap_params.yaml为名创建一个新文件，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。在本例中，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。

6.3.3 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chapter6_tutorials/launch文件夹下以local_costmap_params.yaml为名创建一个新文件，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。在机器人运动过程中，参数rolling_window用于保持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（以赫兹为单位）。在执行空间清理操作前，planner_patiente参数配置规划器寻找一条有效路径的等待时间（以秒为单位）。

能够通过width、height和resolution参数来配置代价地图的尺寸和分

辨率。这些参数都是以米为单位的。

6.3.4 底盘局部规划器配置

一旦完成代价地图的配置，就能够开始进行底盘规划器的配置了。这个底盘规划器会产生一个速度命令来移动机器人。在chapter6_tutorials/launch文件夹下以base_local_planner_params.yaml为名创建一个新文件，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机器人的最大和最小速度，同时也设定了加速度。

当你使用的是一台完整约束的平台（holonomic platform）时，参数holonomic_robot就应设为true。在本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，那么它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，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，差分驱动机器人平台就是一个非全向移动机器人平台。

6.4 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，还要创建一个启动文件。在chapter6_tutorials/launch文件夹下以move_base.launch为名创建一个新文件，并添加以下代码：

```
<launch>
  <!-- Run the map server -->
  <node name="map_server" pkg="map_server" type="map_server"
  args="$(find chapter6_tutorials)/maps/map.yaml" output="screen"
  />
  <include file="$(find amcl)/examples/amcl_diff.launch" />
  <node pkg="move_base" type="move_base" respawn="false"
  name="move_base" output="screen">
    <rosparam file="$(find
    chapter6_tutorials)/launch/costmap_common_params.yaml"
    command="load" ns="global_costmap" />
    <rosparam file="$(find
    chapter6_tutorials)/launch/costmap_common_params.yaml"
    command="load" ns="local_costmap" />
    <rosparam file="$(find
    chapter6_tutorials)/launch/local_costmap_params.yaml"
    command="load" />
    <rosparam file="$(find
    chapter6_tutorials)/launch/global_costmap_params.yaml"
    command="load" />
    <rosparam file="$(find
    chapter6_tutorials)/launch/base_local_planner_params.yaml"
    command="load" />
  </node>
</launch>
```

请注意，在这个文件中会启动之前创建的所有文件。还会启动一个

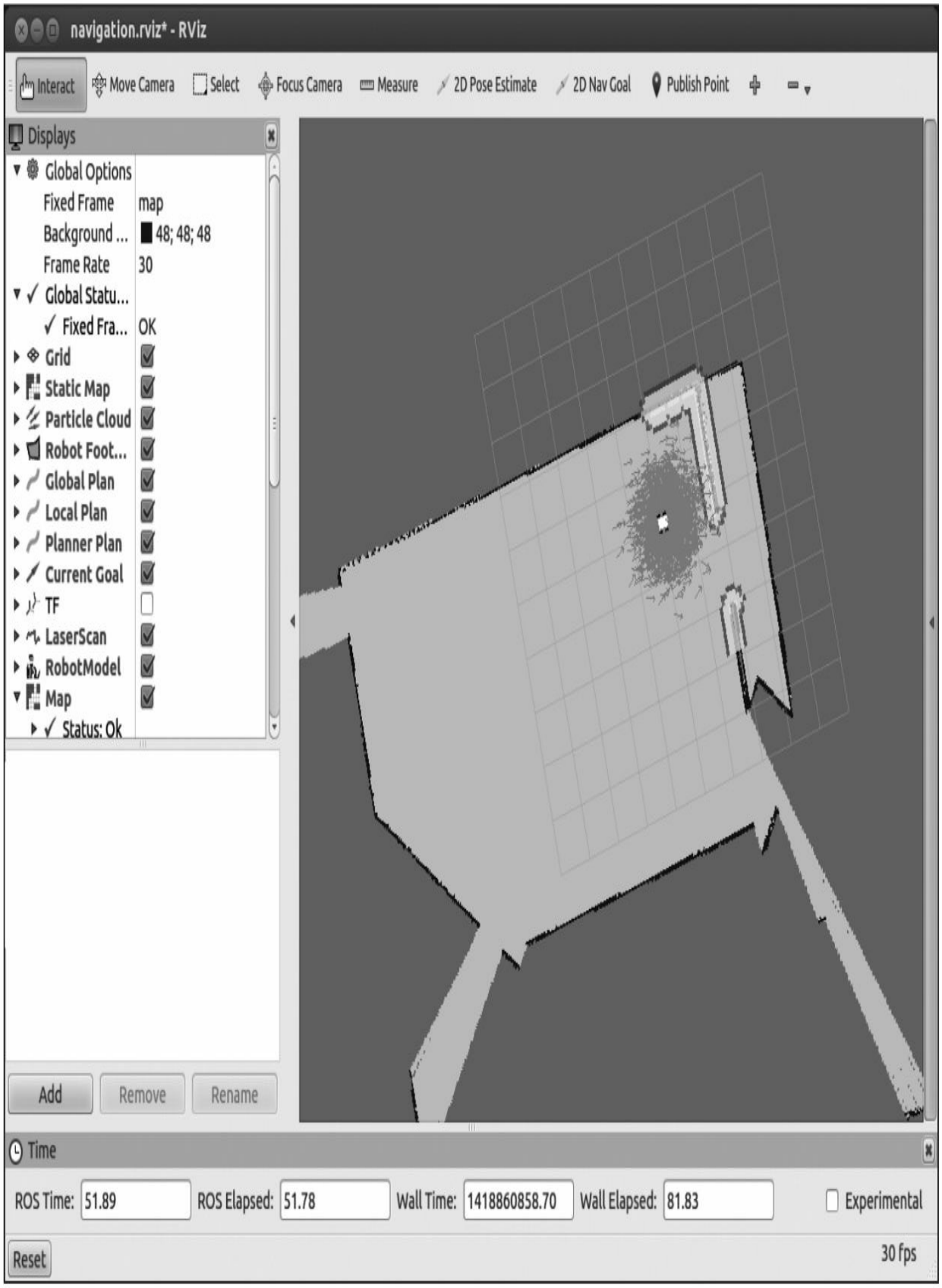
带有第5章中创建的地图的地图服务器和amcl节点。

我们将会使用的这个amcl节点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，那么请参见第5章并创建一个新地图。

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，需要先启动chapter6_configuration_gazebo.launch文件：

```
$ roslaunch chapter6_tutorials move_base.launch
```

然后你会看到以下窗口。


navigation.rviz* - RViz
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
map
Background ...
48; 48; 48
Frame Rate
30
Global Statu...
Fixed Fra...
OK
Grid
Static Map
Particle Cloud
Robot Foot...
Global Plan
Local Plan
Planner Plan
Current Goal
TF
LaserScan
RobotModel
Map
Status: Ok
Add
Remove
Rename
Time
ROS Time:
51.89
ROS Elapsed:
51.78
Wall Time:
1418860858.70
Wall Elapsed:
81.83
Experimental
Reset
30 fps

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（见彩插6），这说明导航功能包集已经正常启动。

正如前面所讲，后面的小节会介绍导航功能包集中用于可视化所有主题必需的配置选项。

6.5 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。本节会展示如何在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。

6.5.1 2D位姿估计

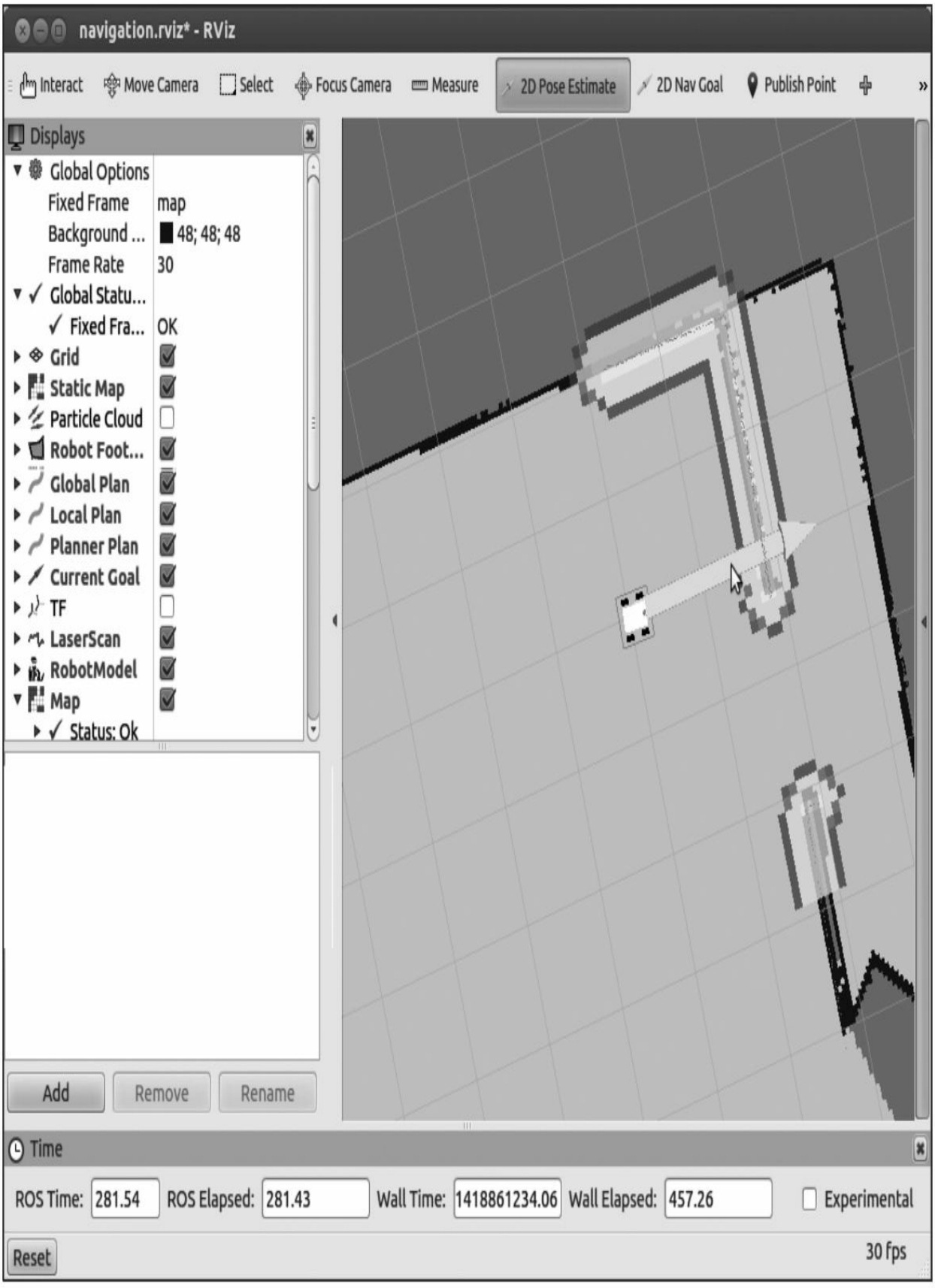
2D位姿估计（2D pose estimate，快捷键P）允许用户通过设定机器人在实际环境中的位姿，初始化导航功能包集的定位系统。

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。单击2D Pose Estimate按钮，并单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位姿。

·Topic：initialpose

·Type：geometry_msgs/PoseWithCovarianceStamped

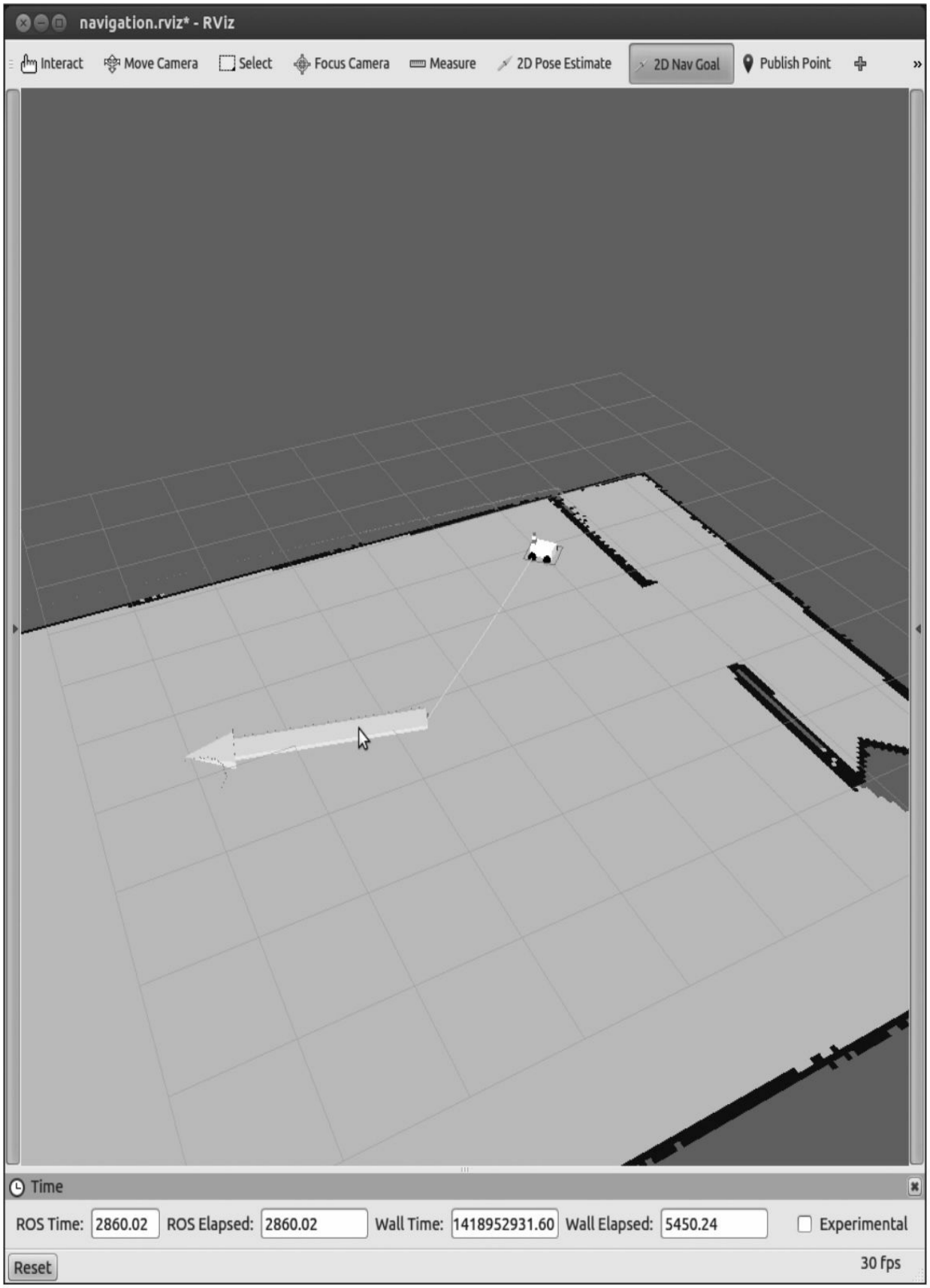

navigation.rviz* - RViz
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
map
Background ...
48; 48; 48
Frame Rate
30
Global Statu...
Fixed Fra...
OK
Grid
Static Map
Particle Cloud
Robot Foot...
Global Plan
Local Plan
Planner Plan
Current Goal
TF
LaserScan
RobotModel
Map
Status: Ok
Add
Remove
Rename
Time
ROS Time:
281.54
ROS Elapsed:
281.43
Wall Time:
1418861234.06
Wall Elapsed:
457.26
Experimental
Reset
30 fps

6.5.2 2D导航目标

2D导航目标（2D nav goal，快捷键G）允许用户为机器人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。在下图中你会看到具体配置方法。单击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


navigation.rviz* - RViz
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Time
ROS Time: 2860.02
ROS Elapsed: 2860.02
Wall Time: 1418952931.60
Wall Elapsed: 5450.24
Experimental
Reset
30 fps

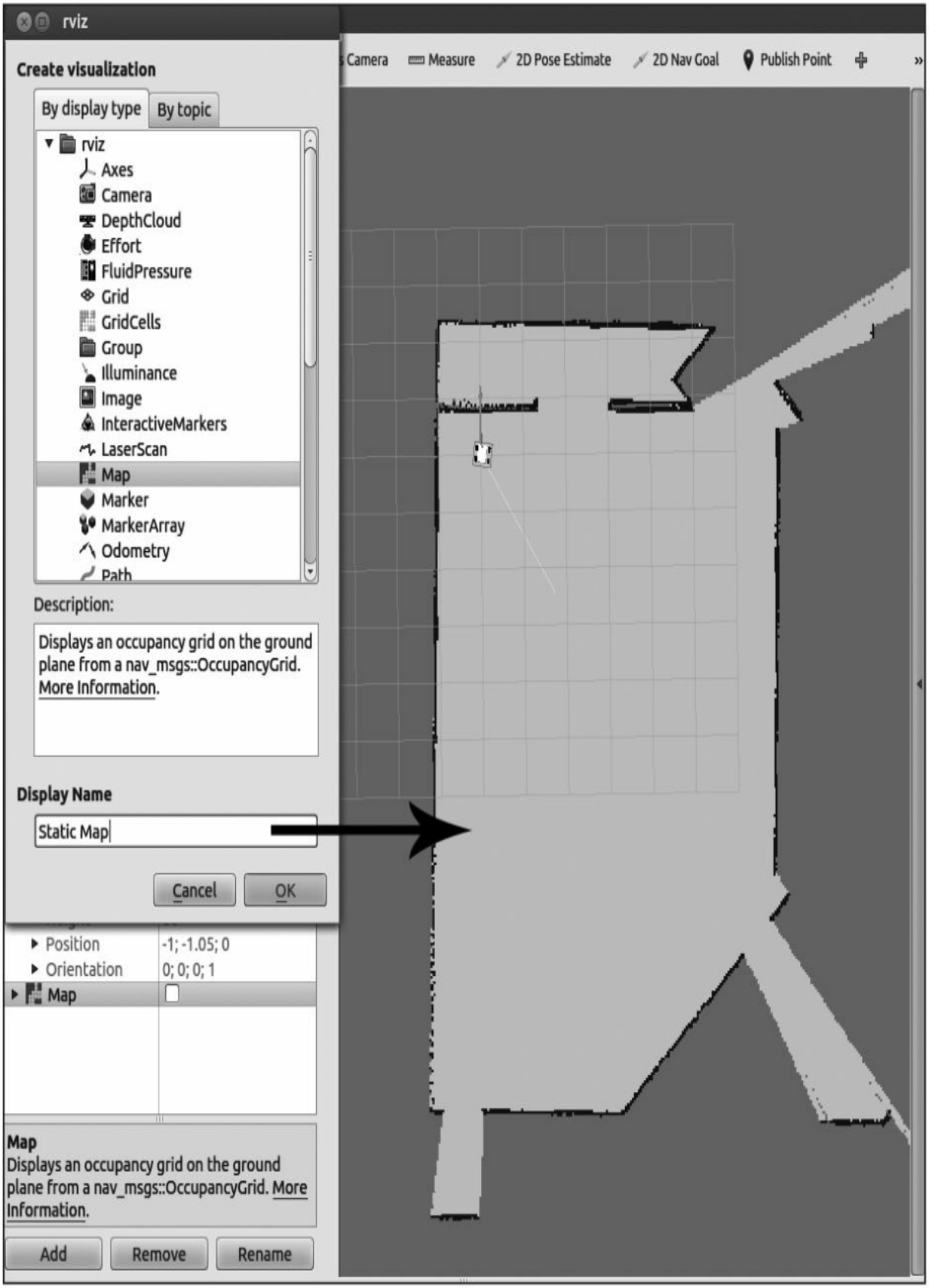
6.5.3 静态地图

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，就会看到在第5章中获取的地图。

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

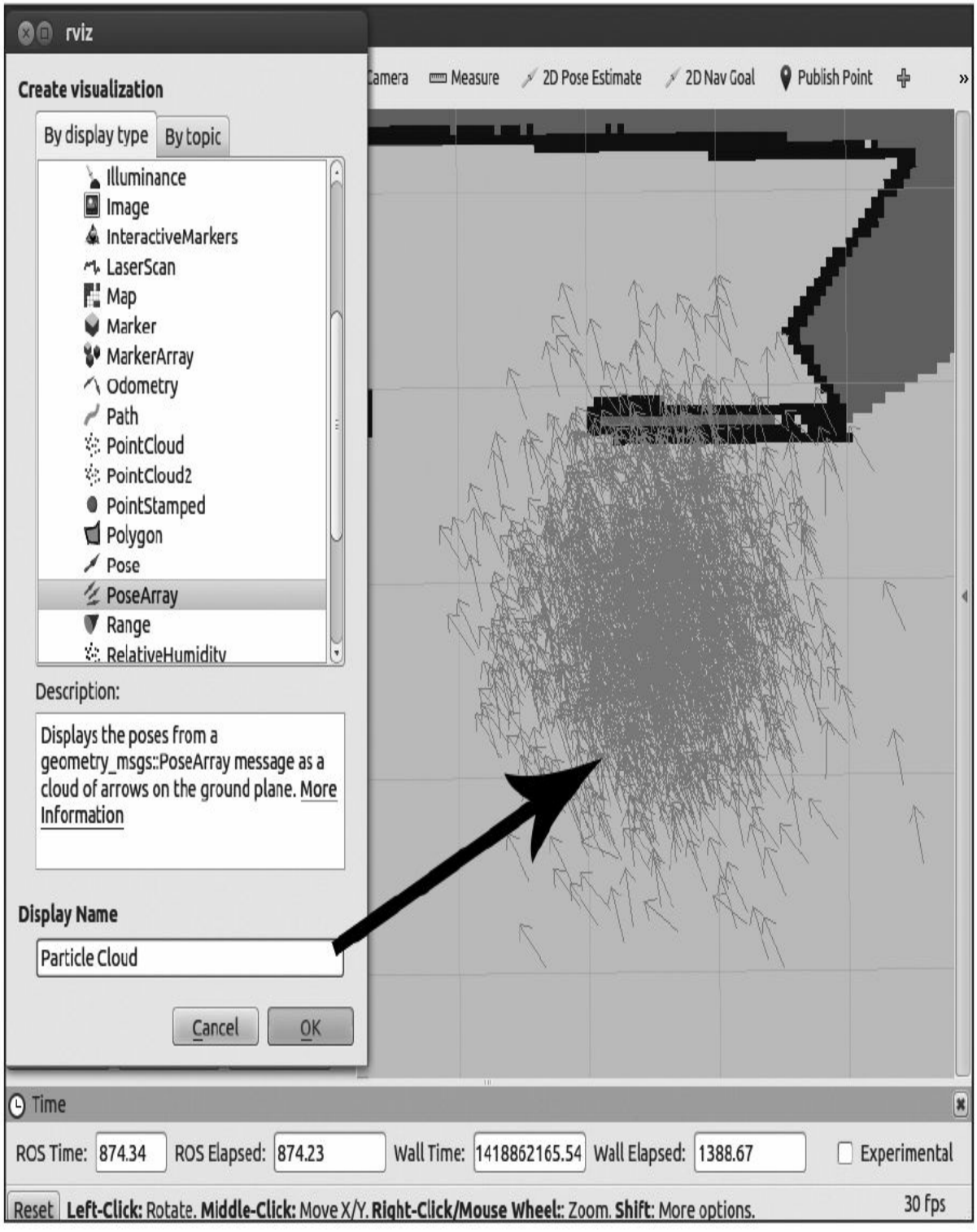

rviz
Create visualization
By display type
By topic
rviz
Axes
Camera
DepthCloud
Effort
FluidPressure
Grid
GridCells
Group
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
Description:
Displays an occupancy grid on the ground plane from a nav_msgs::OccupancyGrid. More Information.
Display Name
Static Map
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Position
-1; -1.05; 0
Orientation
0; 0; 0; 1
Map
Map
Displays an occupancy grid on the ground plane from a nav_msgs::OccupancyGrid. More Information.
Add
Remove
Rename

6.5.4 粒子云

下图显示了用于机器人定位系统的粒子云。点云的分布表示在定位系统中机器人位姿的不确定性。分散的点云代表较高的不确定性，聚集的点云代表较低的不确定性。在本例中，你将为机器人添加如下点云。

·Topic：particlecloud

·Type：geometry_msgs/PoseArray

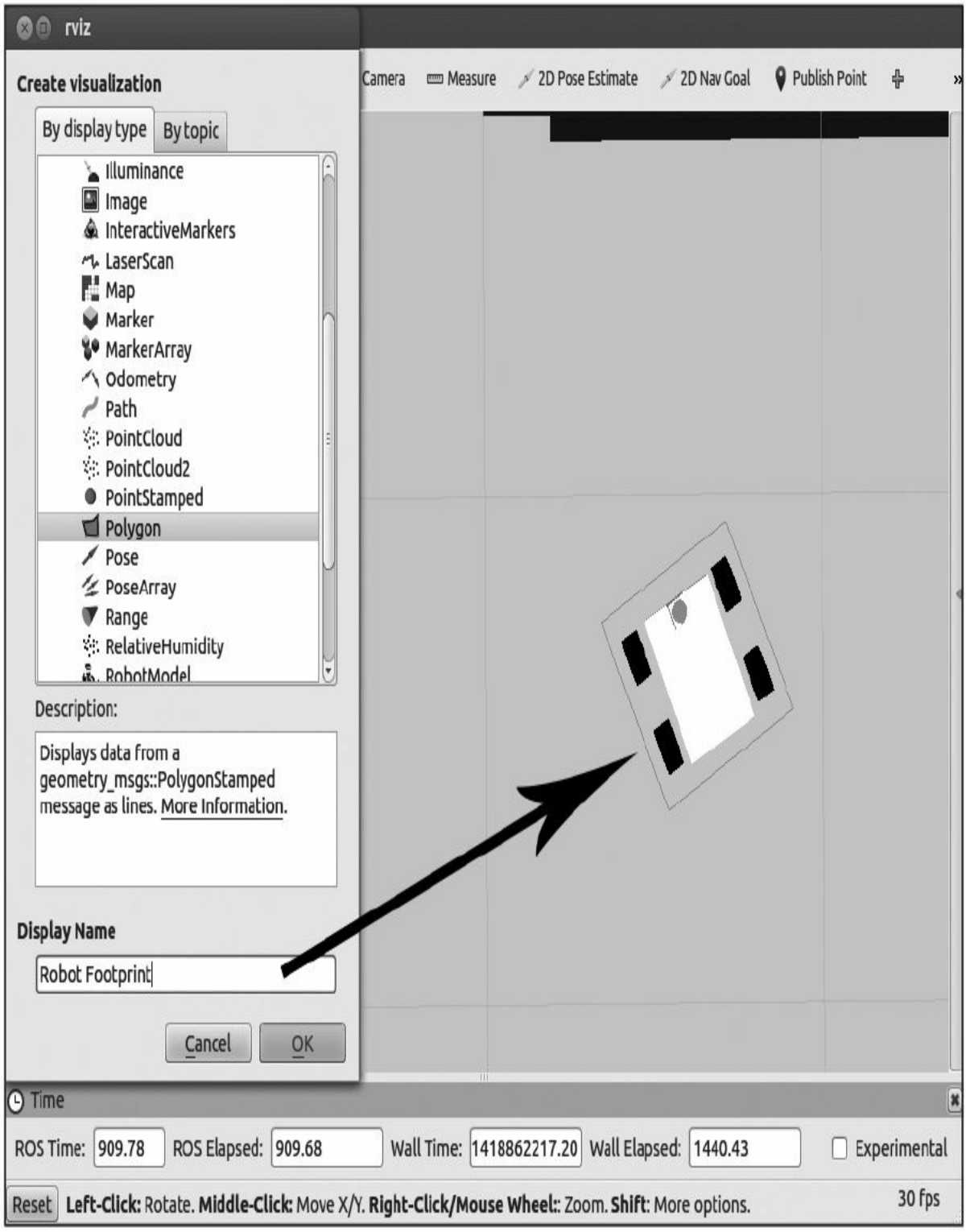

rviz
Create visualization
By display type
By topic
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
PointCloud
PointCloud2
PointStamped
Polygon
Pose
PoseArray
Range
RelativeHumidity
Description:
Displays the poses from a geometry_msgs::PoseArray message as a cloud of arrows on the ground plane. More Information
Display Name
Particle Cloud
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Time
ROS Time: 874.34
ROS Elapsed: 874.23
Wall Time: 1418862165.54
Wall Elapsed: 1388.67
Experimental
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
30 fps

6.5.5 机器人占地空间

下图显示了机器人的占地空间。在本例中，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

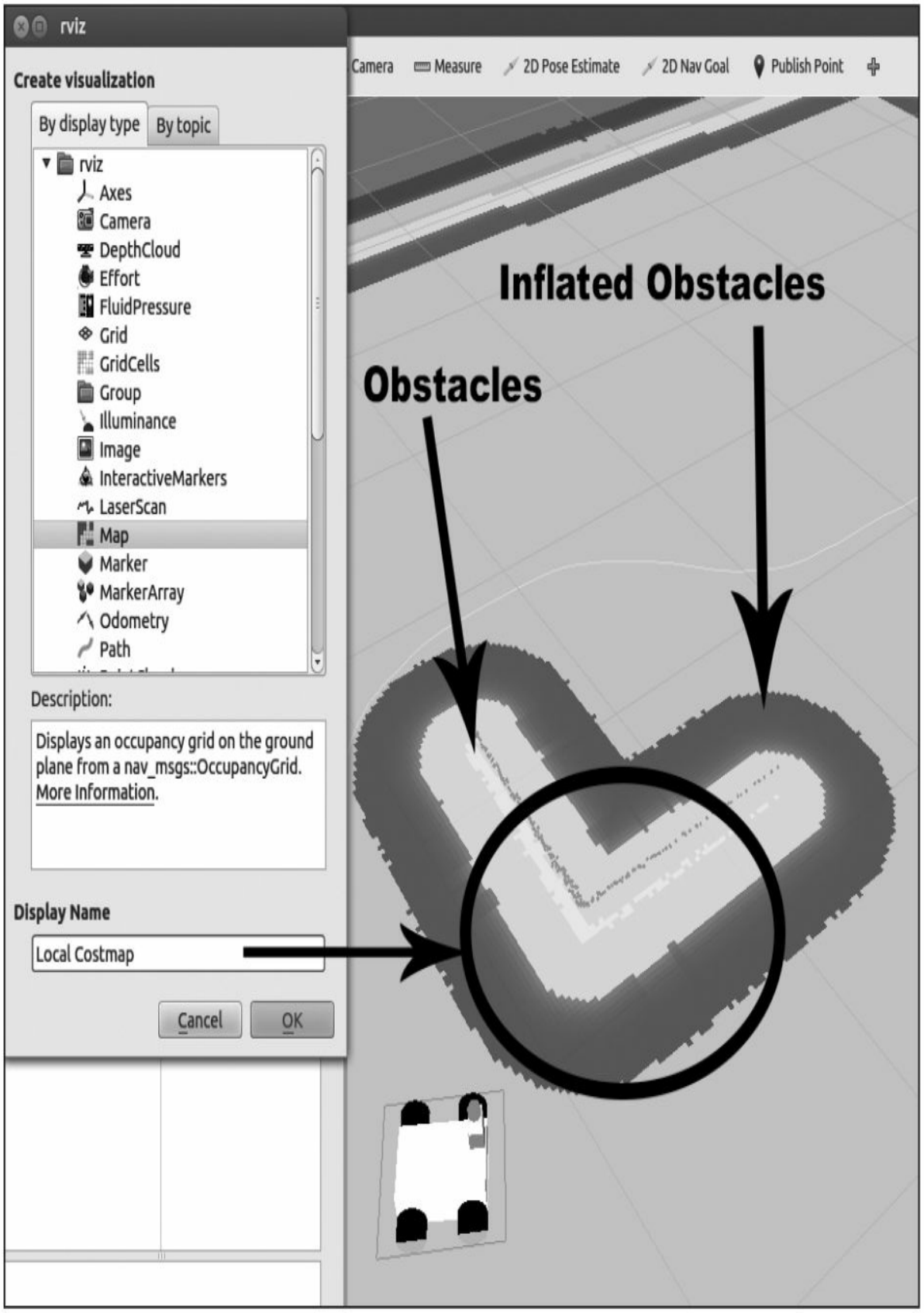

rviz
Create visualization
By display type
By topic
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
PointCloud
PointCloud2
PointStamped
Polygon
Pose
PoseArray
Range
RelativeHumidity
RobotModel
Description:
Displays data from a geometry_msgs::PolygonStamped message as lines. More Information.
Display Name
Robot Footprint
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Time
ROS Time: 909.78
ROS Elapsed: 909.68
Wall Time: 1418862217.20
Wall Elapsed: 1440.43
Experimental
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
30 fps

6.5.6 局部代价地图

接下来显示导航包集的局部代价地图（如下图所示，见彩插7）。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

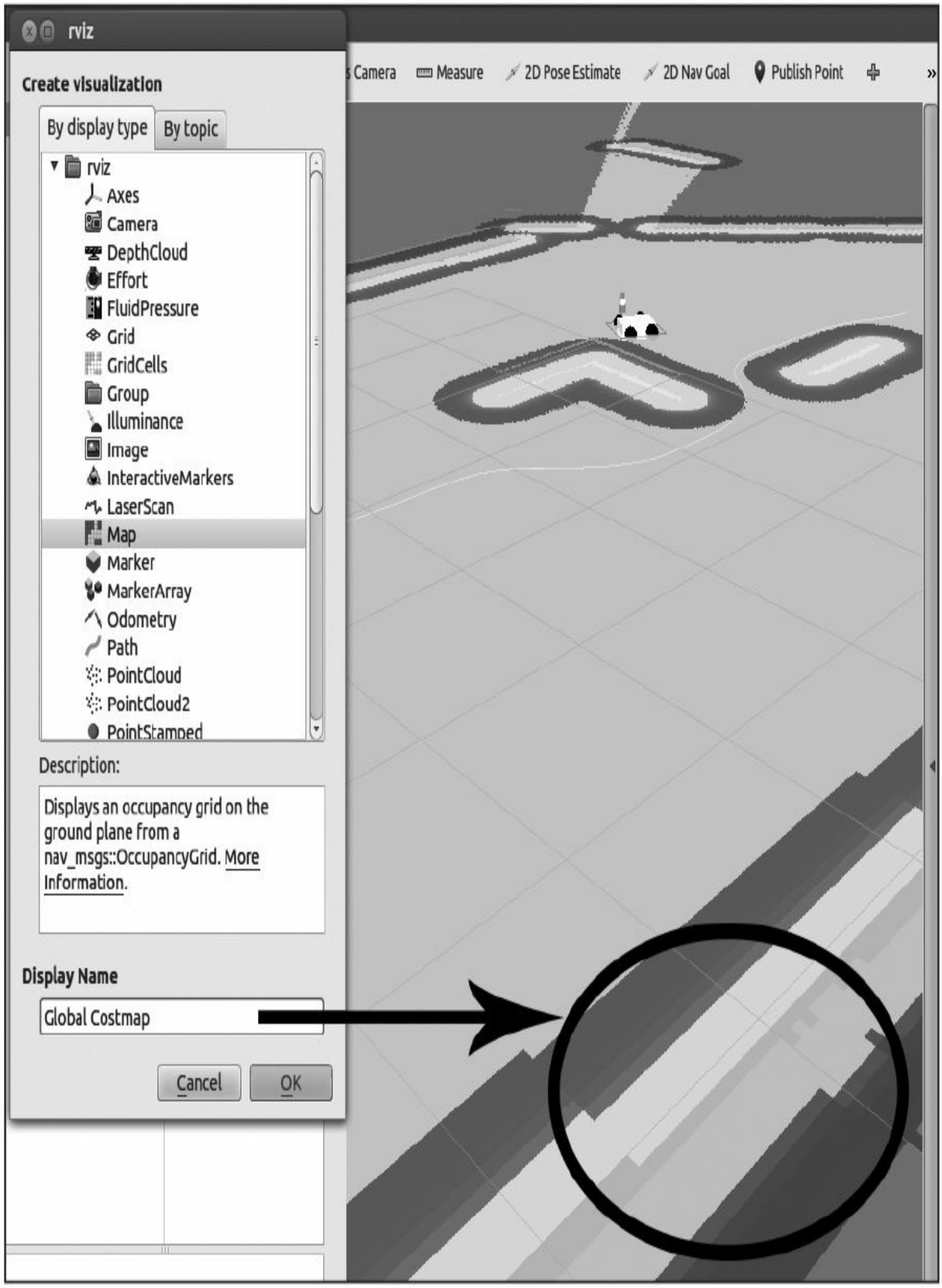

rviz
Create visualization
By display type
By topic
rviz
Axes
Camera
DepthCloud
Effort
FluidPressure
Grid
GridCells
Group
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
Description:
Displays an occupancy grid on the ground plane from a nav_msgs::OccupancyGrid.
More Information.
Display Name
Local Costmap
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Inflated Obstacles
Obstacles

6.5.7 全局代价地图

接下来显示导航包集的全局代价地图（如下图所示，见彩插8）。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

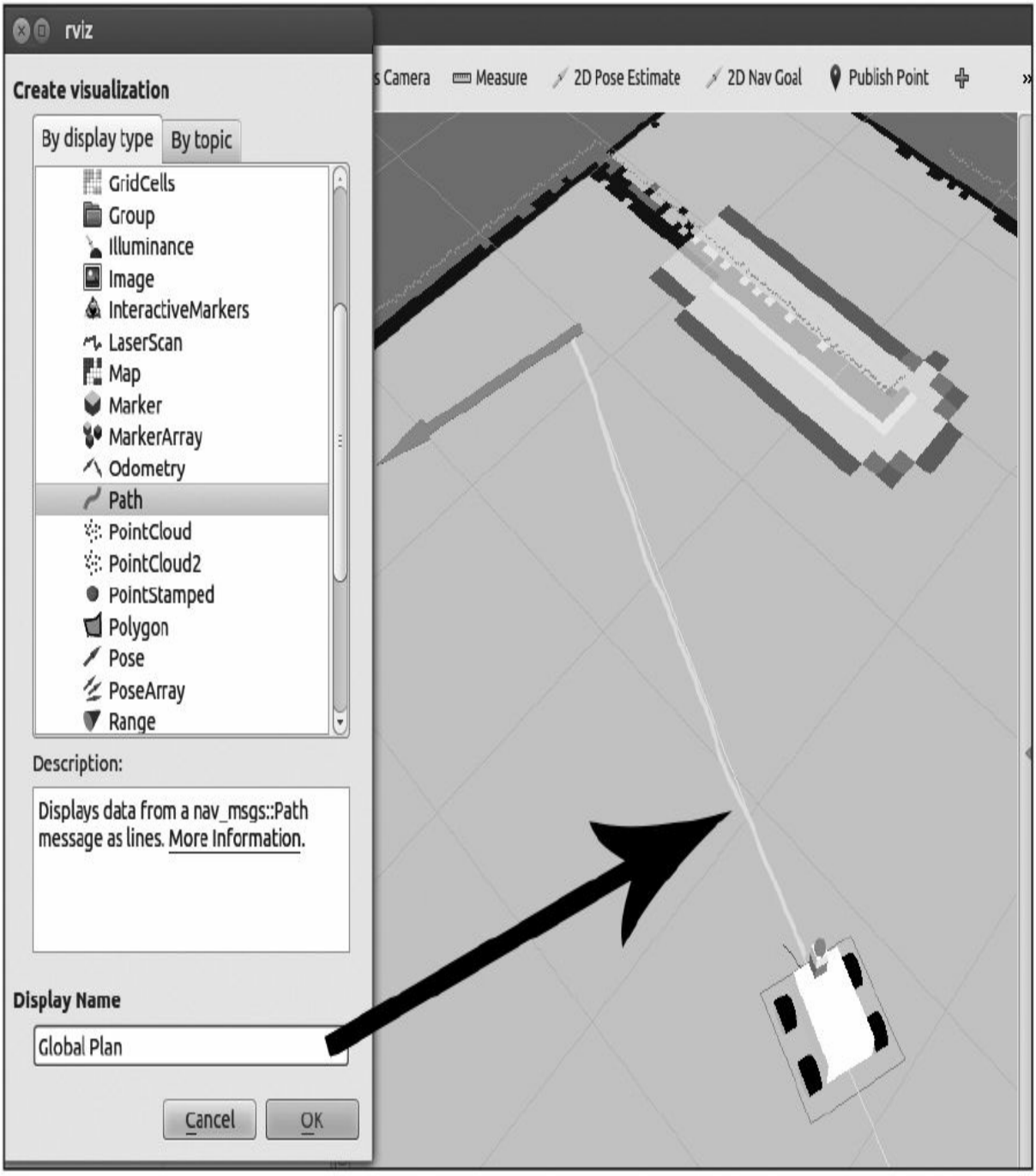

rviz
Create visualization
By display type
By topic
rviz
Axes
Camera
DepthCloud
Effort
FluidPressure
Grid
GridCells
Group
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
PointCloud
PointCloud2
PointStamped
Description:
Displays an occupancy grid on the ground plane from a nav_msgs::OccupancyGrid. More Information.
Display Name
Global Costmap
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point

6.5.8 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（见彩插9）。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

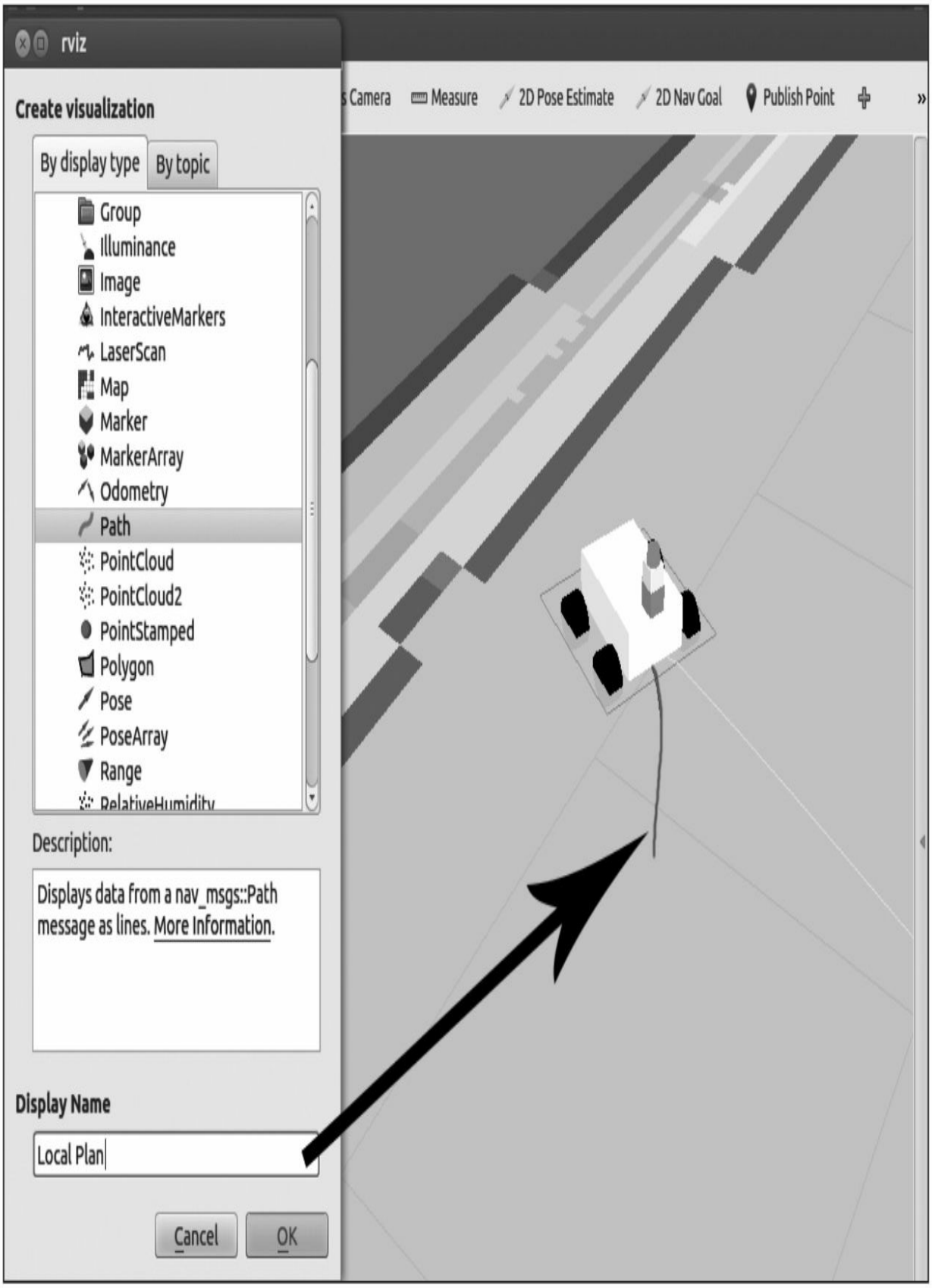

rviz
Create visualization
By display type
By topic
GridCells
Group
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
PointCloud
PointCloud2
PointStamped
Polygon
Pose
PoseArray
Range
Description:
Displays data from a nav_msgs::Path message as lines. More Information.
Display Name
Global Plan
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point

6.5.9 局部规划

下图显示了机器人执行着由局部规划器生成的速度命令以及将会形成的运动轨迹。你能看到其显示为蓝色的线条（见彩插10）。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

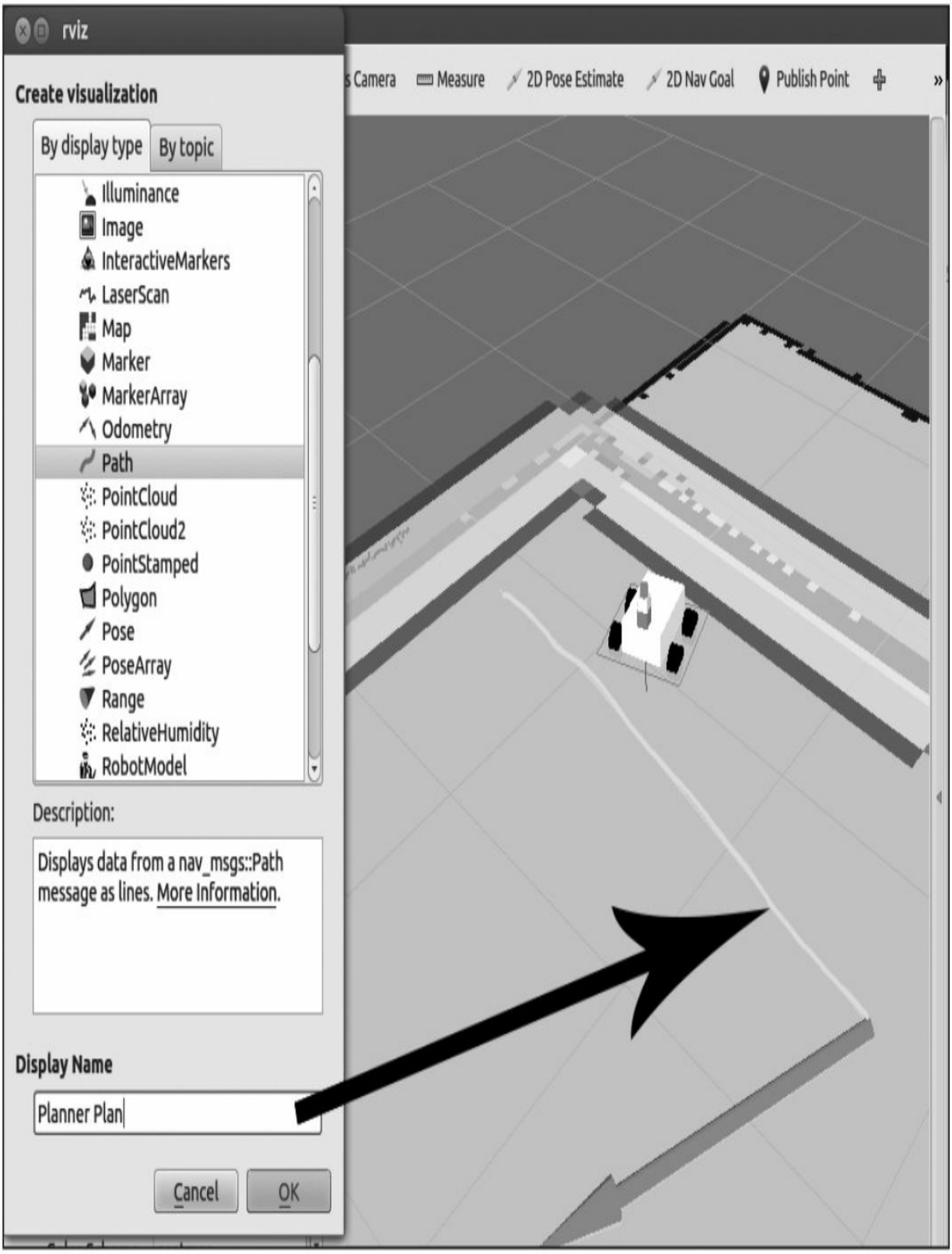

rviz
Create visualization
By display type
By topic
Group
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
PointCloud
PointCloud2
PointStamped
Polygon
Pose
PoseArray
Range
Description:
Displays data from a nav_msgs::Path message as lines. More Information.
Display Name
Local Plan
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point

6.5.10 规划器规划

下图显示了由全局规划器计算的完整规划。你能看到这些线条和全局规划中的线条很类似。

·Topic：NavfnROS/plan

·Type：nav_msgs/Path

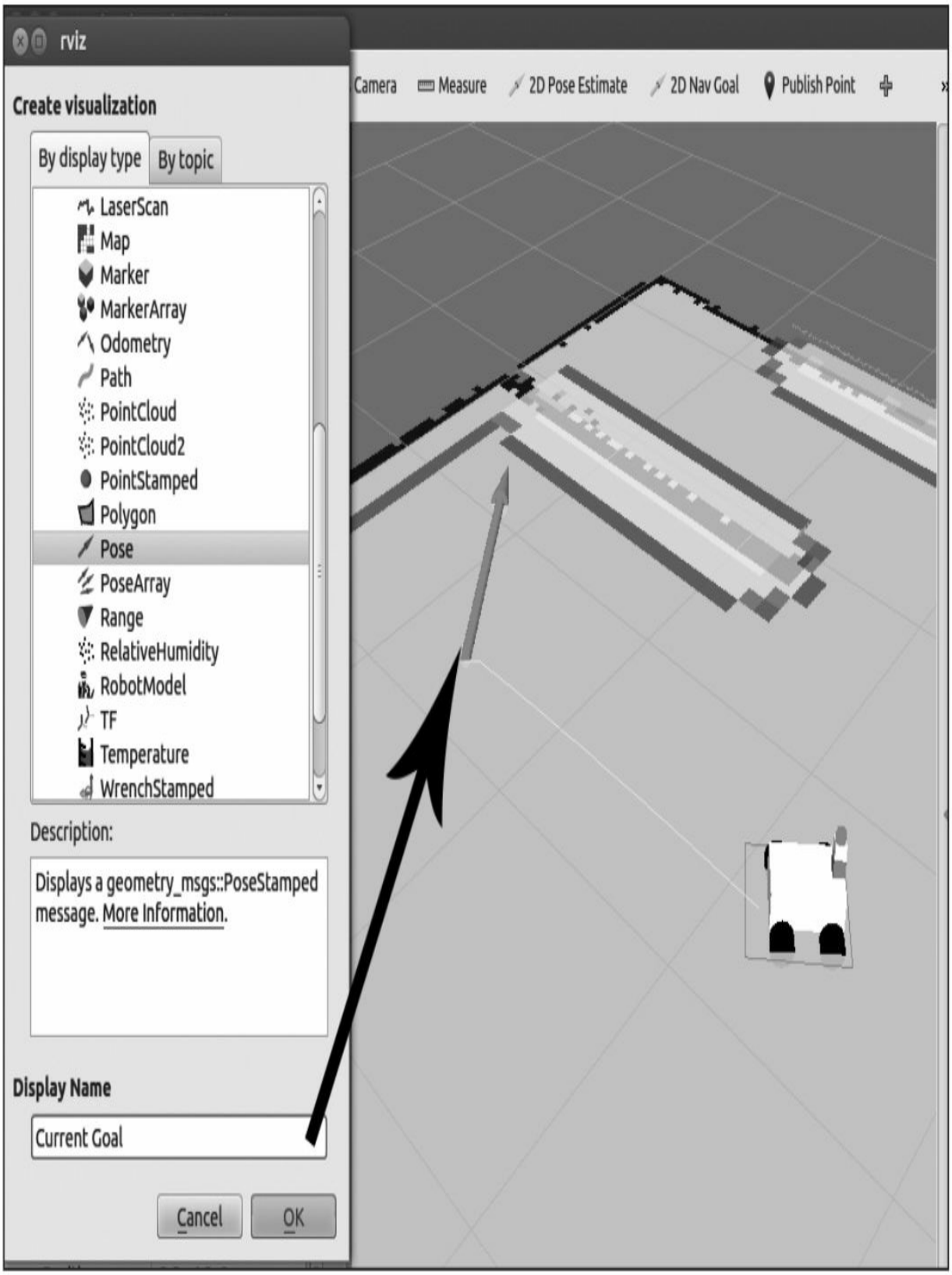

rviz
Create visualization
By display type
By topic
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
PointCloud
PointCloud2
PointStamped
Polygon
Pose
PoseArray
Range
RelativeHumidity
RobotModel
Description:
Displays data from a nav_msgs::Path message as lines. More Information.
Display Name
Planner Plan
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point

6.5.11 当前目标

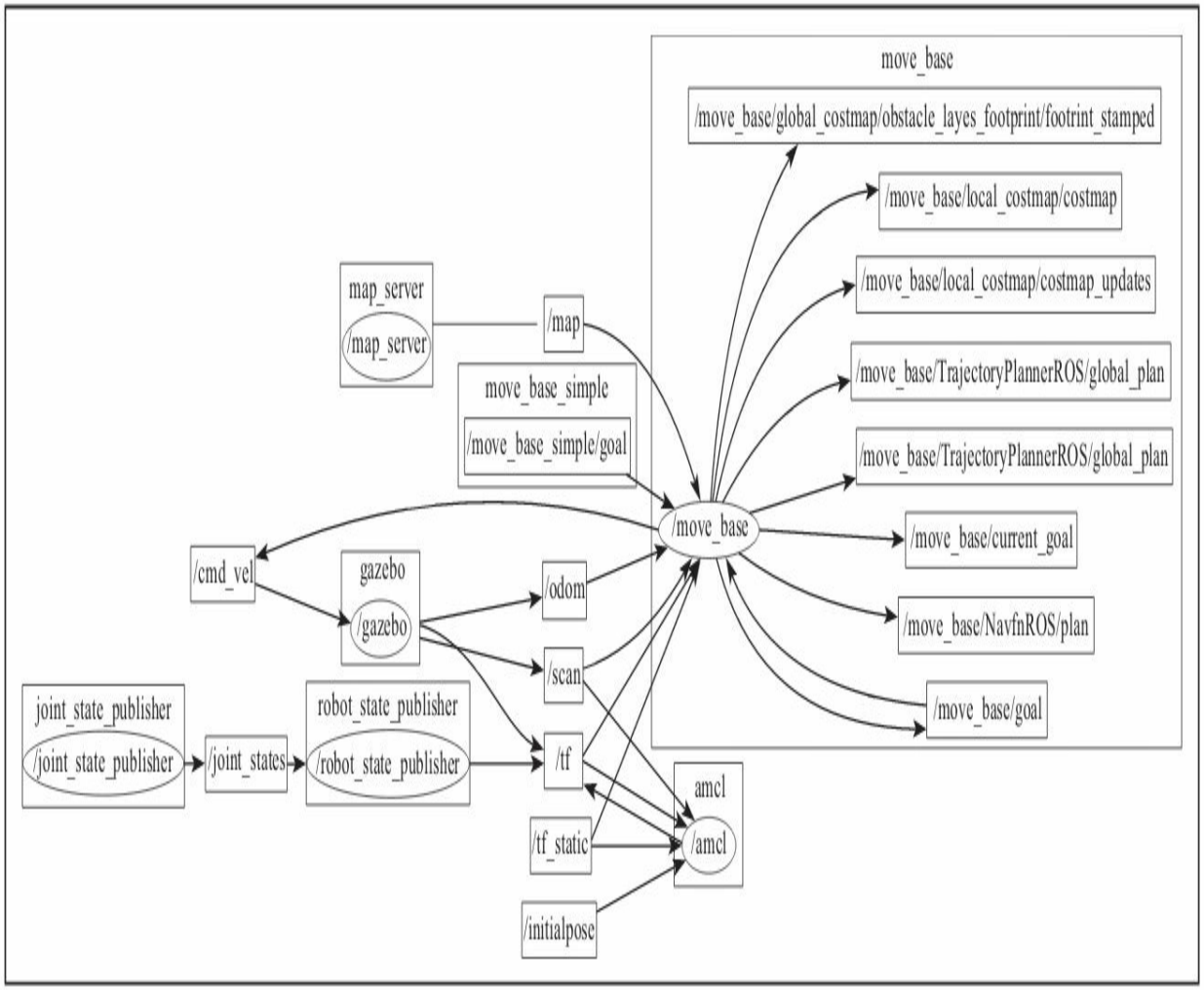
下图显示了导航功能包集正在实现的导航位姿目标。它是一个红色的箭头（见彩插11），你重新配置一个新的2D导航目标后它才会出现。它用于了解机器人最终的导航目的地。

·Topic：current_goal

·Type：geometry_msgs/PoseStamped


rviz
Create visualization
By display type
By topic
LaserScan
Map
Marker
MarkerArray
Odometry
Path
PointCloud
PointCloud2
PointStamped
Polygon
Pose
PoseArray
Range
RelativeHumidity
RobotModel
TF
Temperature
WrenchStamped
Description:
Displays a geometry_msgs::PoseStamped message. More Information.
Display Name
Current Goal
Cancel
OK
Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point

这些可视化数据是你需要在rviz中查看的所有导航功能包集。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。现在我们查看整个系统的总览图。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。



6.6 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（Adaptive Monte Carlo Localization，AMCL）算法完成机器人定位。AMCL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中的具体实现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AMCL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，你会找到该算法的更详细介绍：http://www.ros.org/wiki/amcl以及http://www.probabilistic-robotics.org/。

AMCL节点主要使用激光扫描和激光地图。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基于激光生成的地图，传递消息并完成位姿估计的计算。我们要先针对ROS提供的各个初始化参数，完成AMCL算法粒子滤波器的初始化。如果你没有设定初始位姿，AMCL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。所以建议在rviz中通过2D Pose Estimate按钮来设定初始位姿。

然后需要引用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" />
    <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用于配置激光的类型。在本例中，将参数设置为likelihood_field。但在算法中，也可以设置为beam激光。

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（而无须在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并且能改变的信息。

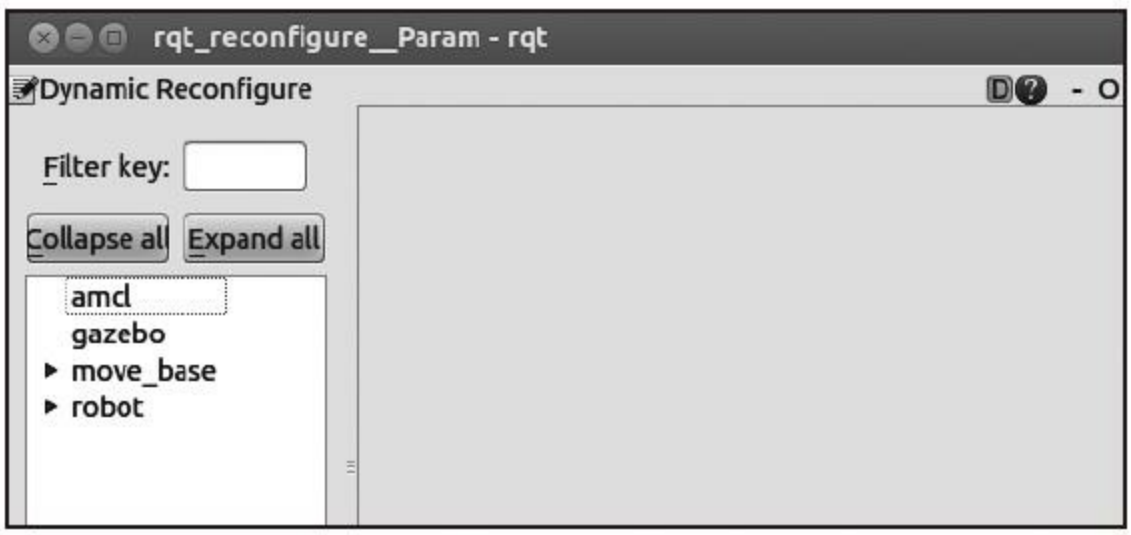
6.7 使用rqt_reconfigure修改参数

通过使用rqt_reconfigure在不重新启动仿真的情况下来对参数值进行修改，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：

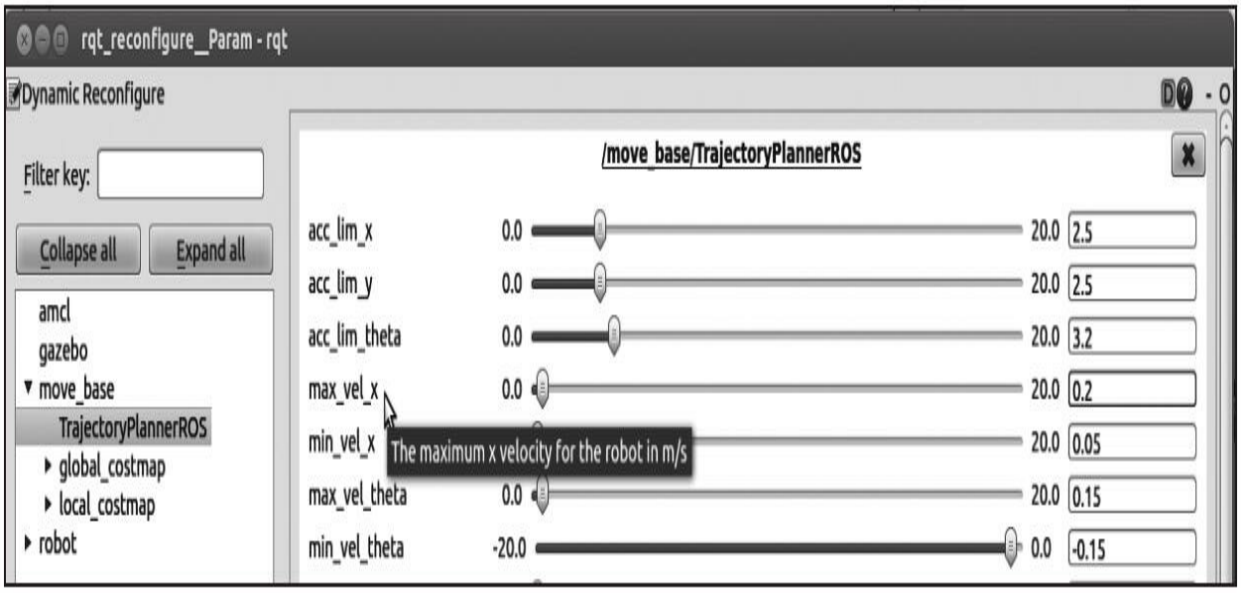
```
$ rosrun rqt_reconfigure rqt_reconfigure
```

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。

将鼠标指针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。

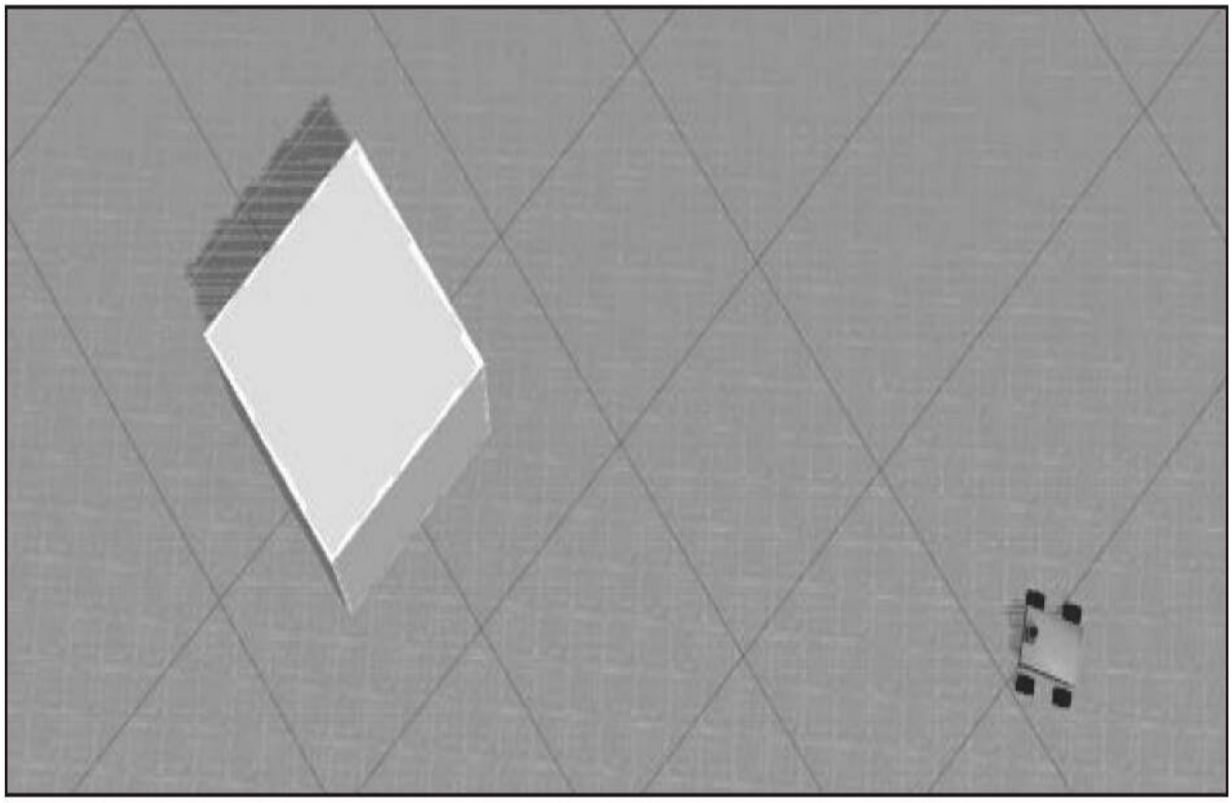

rqt_reconfigure__Param - rqt
Dynamic Reconfigure
Filter key:
Collapse all
Expand all
amcl
gazebo
move_base
TrajectoryPlannerROS
global_costmap
local_costmap
robot
/move_base/TrajectoryPlannerROS
acc_lim_x 0.0 20.0 2.5
acc_lim_y 0.0 20.0 2.5
acc_lim_theta 0.0 20.0 3.2
max_vel_x 0.0 20.0 0.2
min_vel_x 20.0 0.05
The maximum x velocity for the robot in m/s
max_vel_theta 0.0 20.0 0.15
min_vel_theta -20.0 0.0 -0.15

6.8 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，在仿真环境中在机器人路径的中央添加一个大的箱子。导航功能包集会检测到新的障碍物并自动创建新的路径。

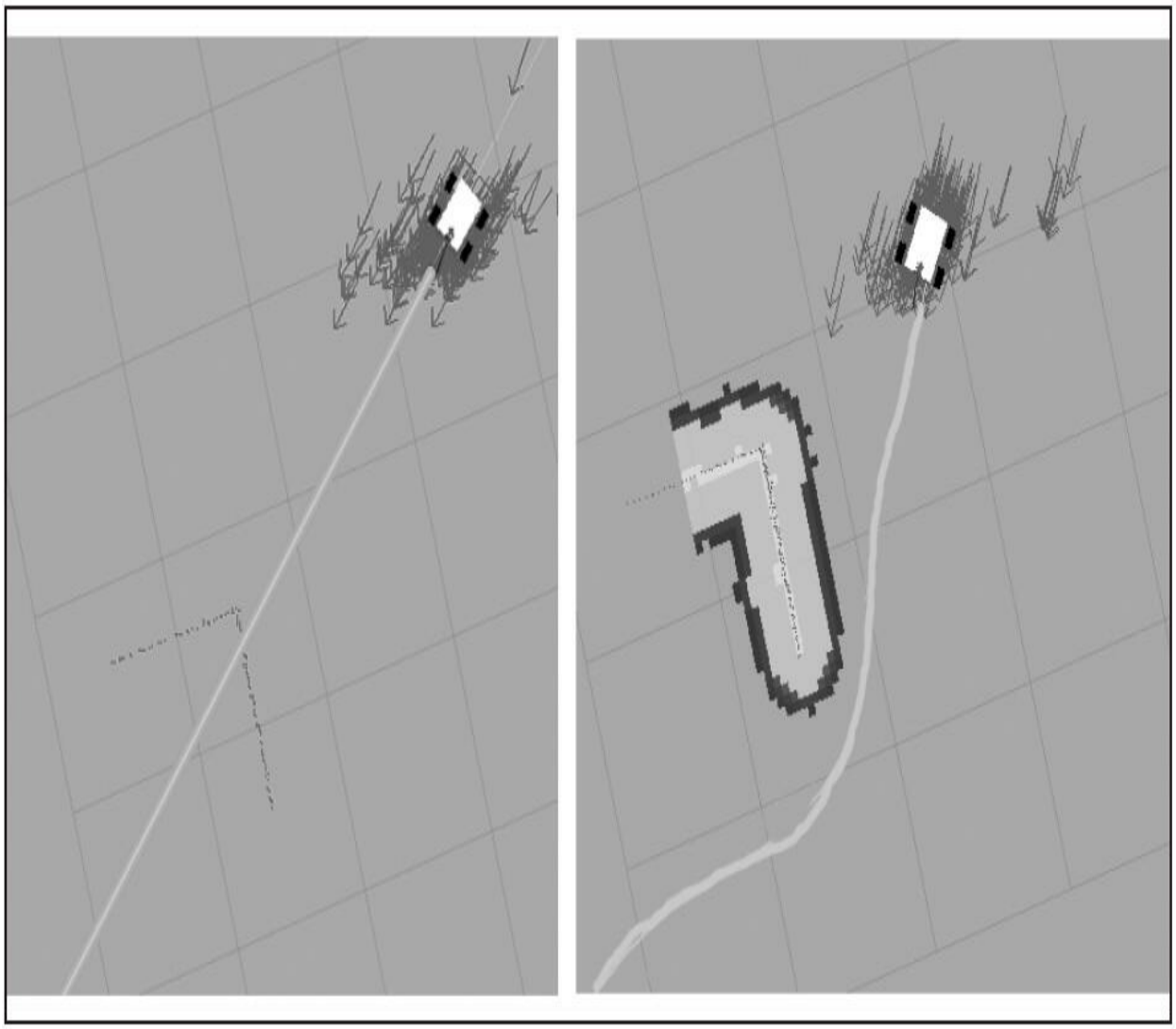
在下图中，你能够看到我们添加了一个物体。Gazebo有一些预定义的3D物体。它们可以在仿真中与移动机器人、手臂、类人机器人等一起使用。

要查看这些物体的清单，可以单击Insert model。选择一个物体并单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4m×4m的一个区域）。



6.9 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。这样即使我们不在计算机前操作rviz，它也会根据预先的指令运动。

现在来学习如何使用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，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，使用激光进行扫描等。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，那么你可能希望具有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chapter6_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>

typedefactionlib::SimpleActionClient<move_base_msgs::
MoveBaseAction>MoveBaseClient;

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

  MoveBaseClientac("move_base", true);

  while(!ac.waitForServer(ros::Duration(5.0))){
    ROS_INFO("Waiting for the move_base action server");
  }

  move_base_msgs::MoveBaseGoal goal;

  goal.target_pose.header.frame_id = "map";
  goal.target_pose.header.stamp = ros::Time::now();

  goal.target_pose.pose.position.x = 1.0;
  goal.target_pose.pose.position.y = 1.0;
  goal.target_pose.pose.orientation.w = 1.0;

  ROS_INFO("Sending goal");
  ac.sendGoal(goal);

  ac.waitForResult();
  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
  ROS_INFO("You have arrived to the goal position");
  else{
    ROS_INFO("The base failed for some reason");
  }
  return 0;
}
```

在CMakeList.txt文件中添加以下文件来为程序编译生成可执行文件：

```
add_executable(sendGoalssrc/sendGoals.cpp)
target_link_libraries(sendGoals ${catkin_LIBRARIES})
```

现在，使用以下命令编译功能包：

```
$ catkin_make
```

下一步就是使用启动文件启动仿真环境并测试新程序。使用下面的指令来启动所有节点和配置：

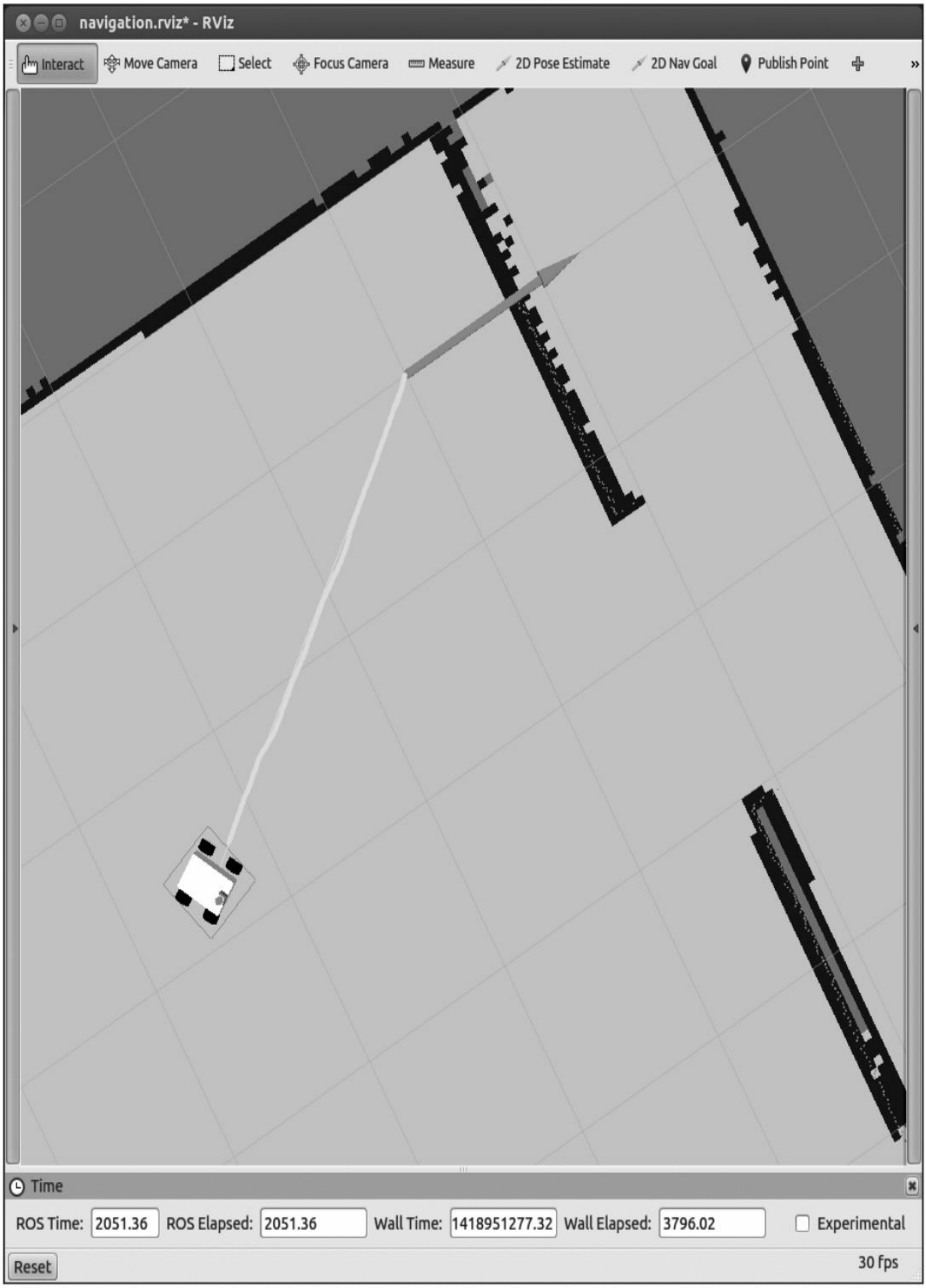
```
$ roslaunch chapter6_tutorials chapter6_configuration_gazebo.launch
```

```
$ roslaunch chapter6_tutorials move_base.launch
```

一旦配置好2D位姿估计，使用以下命令在一个新的命令行窗口中启动sendGoal节点：

```
$ rosrun chapter6_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（见彩插12）。


navigation.rviz* - RViz
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Time
ROS Time: 2051.36
ROS Elapsed: 2051.36
Wall Time: 1418951277.32
Wall Elapsed: 3796.02
Experimental
Reset
30 fps

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消息：

```
[ 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。

6.10 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机器人了。无论你使用的是真正的机器人还是仿真机器人，只须使用导航功能包集，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定位机器人的算法，以便轻松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恰当地使用这些参数。能否正确使用参数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7章　使用MoveIt!

ROS中有针对机器人进行移动操作的一套工具——MoveIt!。主页http://moveit.ros.org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。除了后台功能之外，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。

本章首先介绍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中使用MoveIt!插件。

7.1 MoveIt!体系结构

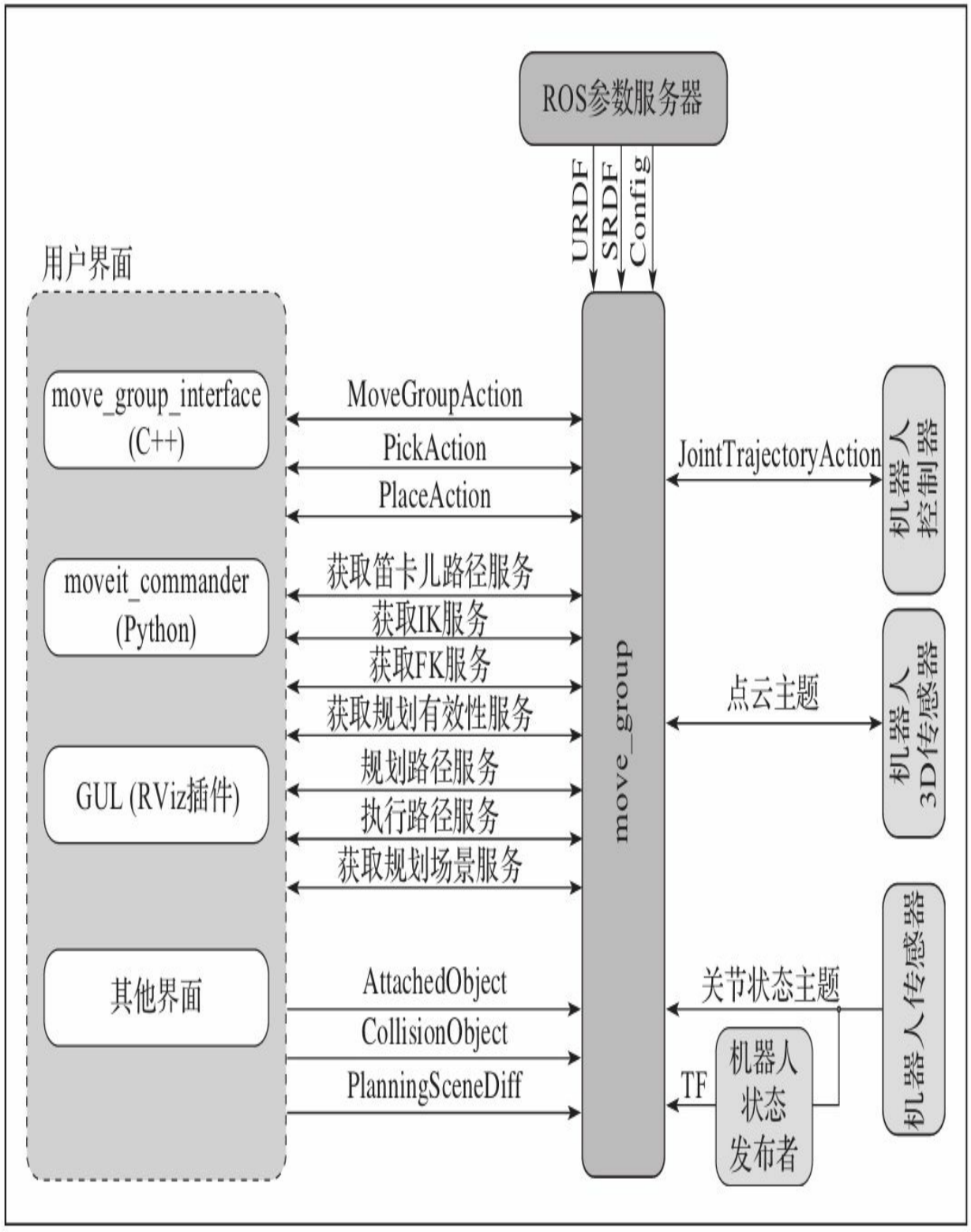
MoveIt!体系结构如下图所示，内容选自官方文档的概念部分（http://moveit.ros.org/documentation/concepts/）。这里简要介绍主要概念。要安装MoveIt!，只需要运行这个命令：

```
$ sudo apt-get install ros-kinetic-moveit-full
```

另外，也可以通过在它相应的工作空间中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
```

下图是MoveIt!体系结构框图。



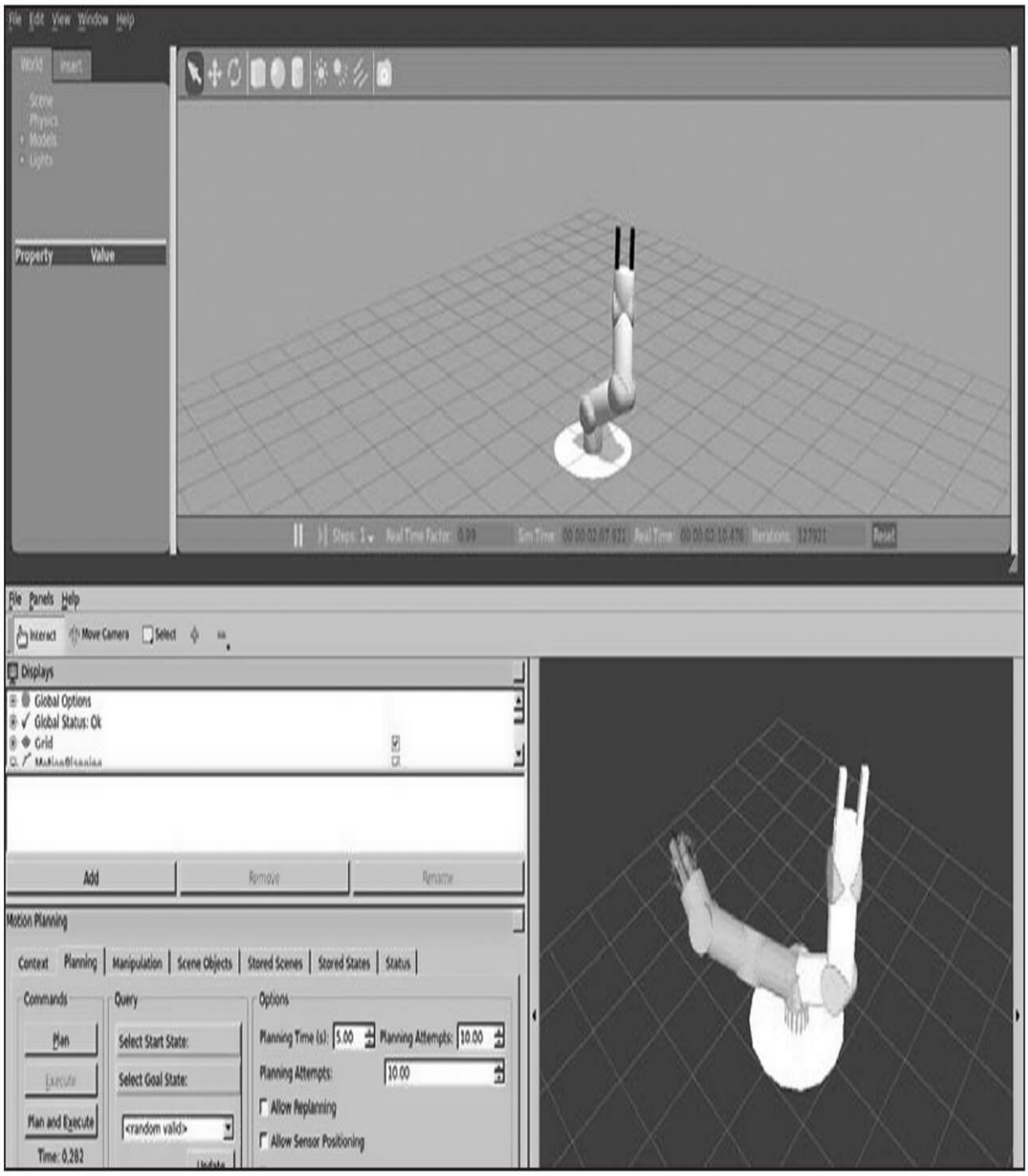
MoveIt!的体系结构框图

体系结构的核心是move_group元件。其主要思想是，先依据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。这类算法包含与物体交互的场景以及该群组的关节特性。

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）上规划执行。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程序发布。需要注意的是，MoveIt!提供了一个GUI向导去定义给定机器人的关节群组，它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，就可以和它连接并通信。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进行执行等，如下图所示。



集成到Gazebo中的MoveIt!机械臂仿真

7.1.1 运动规划

运动规划（Motion planning）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物发生碰撞——包括自身连杆或由传感器感知（一般为点云，point clouds）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）。

当把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，找到使移动机械臂的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。例如，夹持器拿起物体的运动学约束条件，包括位置、方向、可视角度以及用户指定的约束等。

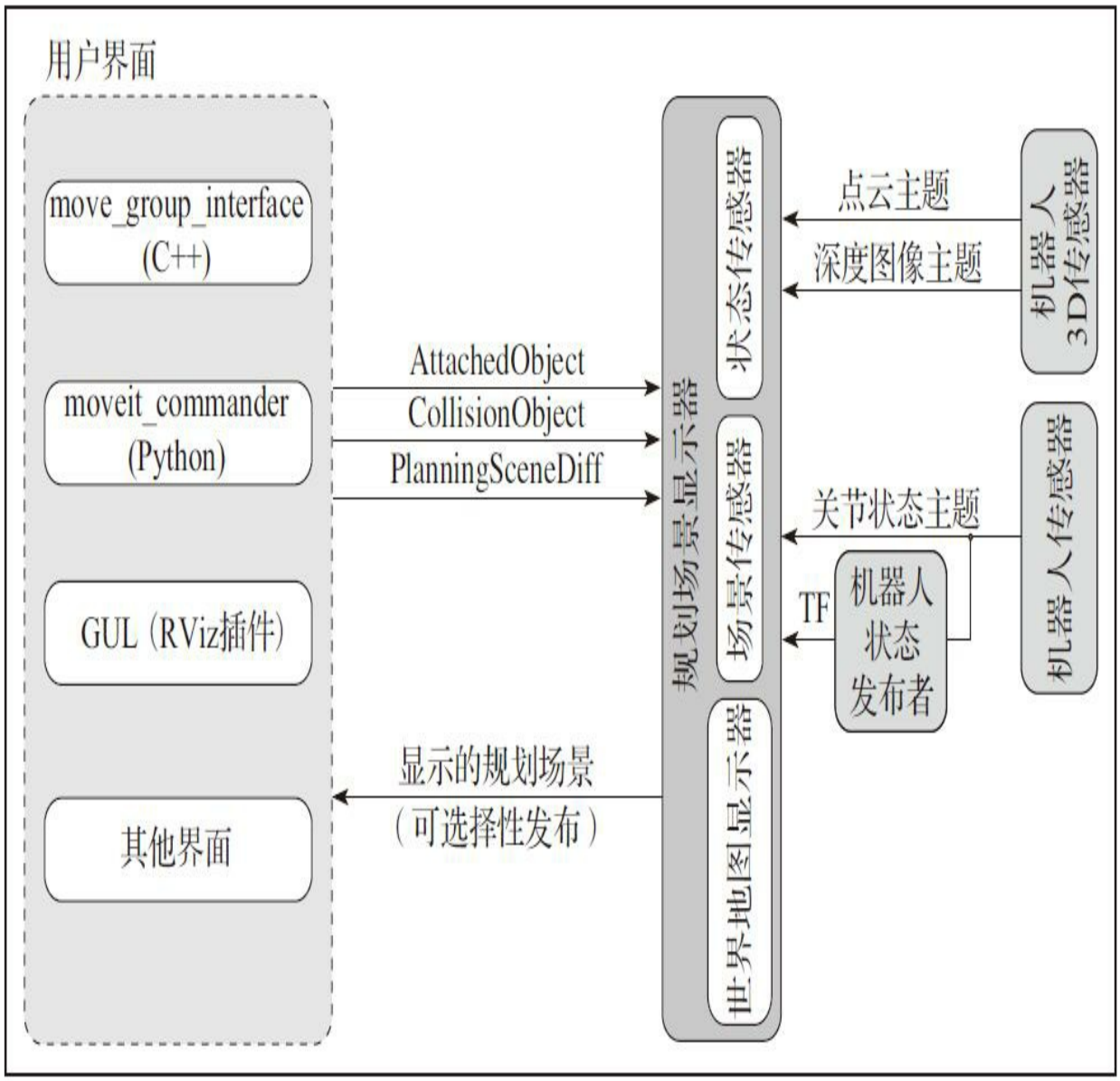
运动规划的结果是得到一条能使机械臂移动到预定目标位置的轨迹，这条轨迹需要避免碰撞，并且满足关节层速度和加速度的约束条件。

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。

7.1.2 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，由下图所示的规划场景显示器实现。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，其中显示joint_state、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。

规划场景框图如下图所示。



MoveIt!规划场景框图

7.1.3 世界几何结构显示器

世界几何结构显示器采用了占有地图显示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。为了生成三维表示方式，MoveIt!支持不同的传感器插件去感知环境，主要通过两种方式：点云和深度图像。

7.1.4 运动学

正向运动学及其雅可比矩阵（Jacobians）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，该解算器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。

7.1.5 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。

7.2 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。

7.2.1 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，我们提供了一组功能包，其中包含所有必要的配置、机器人描述、启动脚本和模块等来将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chapter7_tutorials文件夹下的软件库中，提供了下列软件包。

·chapter7_tutorials：这个软件库相当于一个容器，其中包含了将在本章中使用的功能包。这种结构通常需要元功能包告诉catkin这些包是松散关联的。因此，这个功能包是资源库的元功能包。

·rosbook_arm_bringup：这个功能包集中了控制器和MoveIt!的启动。它可以用于调出实际或者仿真中的机器人。

·rosbook_arm_controller_configuration：这个功能包包括移动机械臂需要加载的控制器的启动文件。这些轨迹控制器（JointTrajectoryController）用于支持MoveIt!运动规划。

·rosbook_arm_controller_configuration_gazebo：这个功能包包含关节轨迹控制器的配置。此配置也包括在Gazebo中控制机械臂需要的PID值。

·rosbook_arm_description：这个功能包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，其中包含Gazebo启动文件（也负责启动MoveIt!和仿真环境）和控制器，以及负责运行需要的启动文件（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Gazebo中的关节。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：这个功能包通过MoveIt!设置助手生成，其中包含MoveIt!和RViz插件所需的大部分启动文件，以及MoveIt!的一些配置文件。

·rosbook_arm_snippets：除了抓取和放置示例之外，此功能包包含本章中使用的所有代码段。

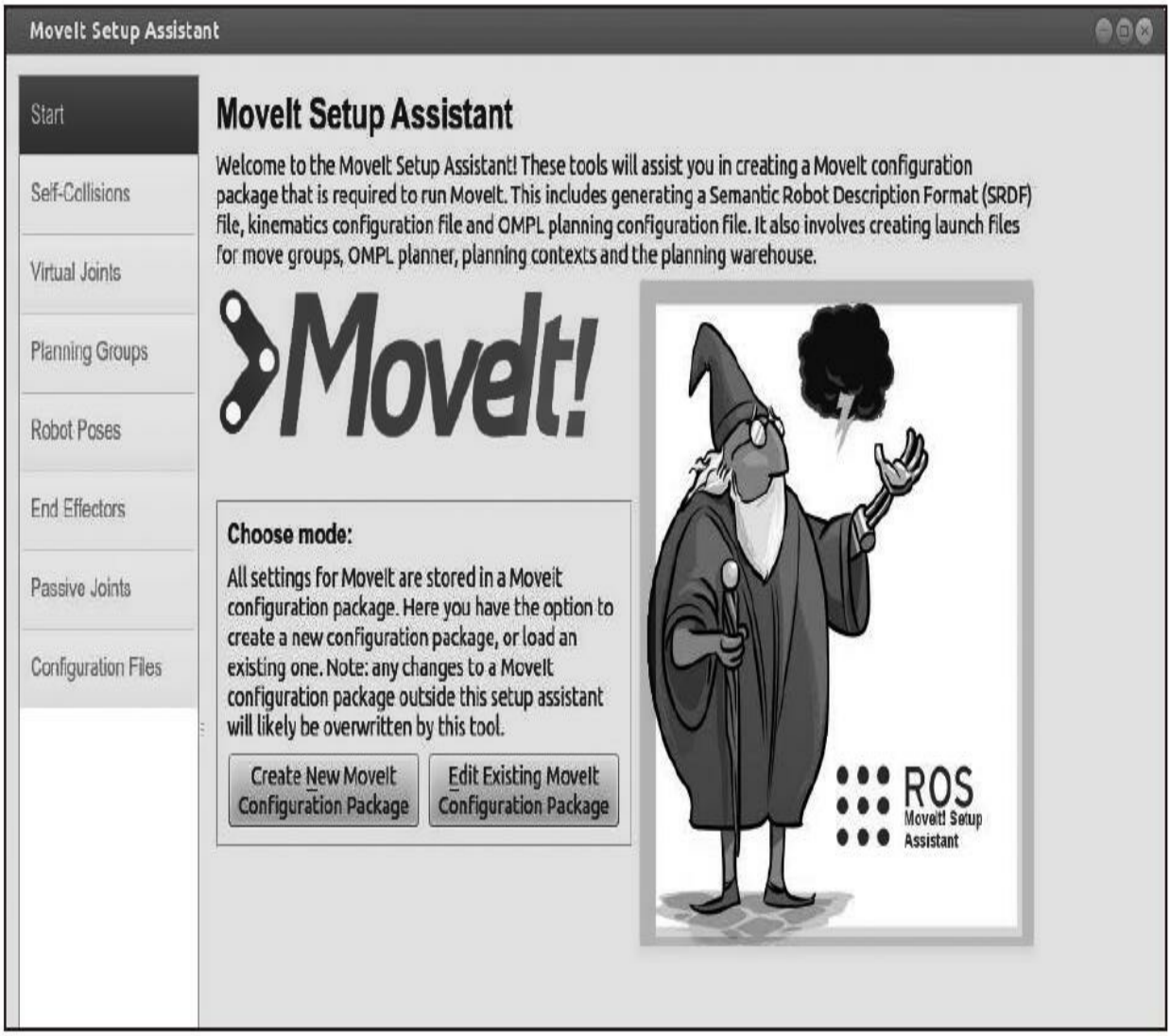
·rosbook_arm_pick_and_place：此功能包是本书中最庞大和最复杂的示例，包含一个如何使用MoveIt!完成对象的抓取和放置的文档。

7.2.2 使用设置助手生成一个MoveIt!功能包

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（Setup Assistant）基于用户提供的信息负责生成全部的配置文件和启动（launch）脚本。一般而言，这是开始使用MoveIt!最简单的方法。同时还会生成几个演示性的启动脚本，系统可以在无真实机械臂或仿真的情况下运行。

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
```

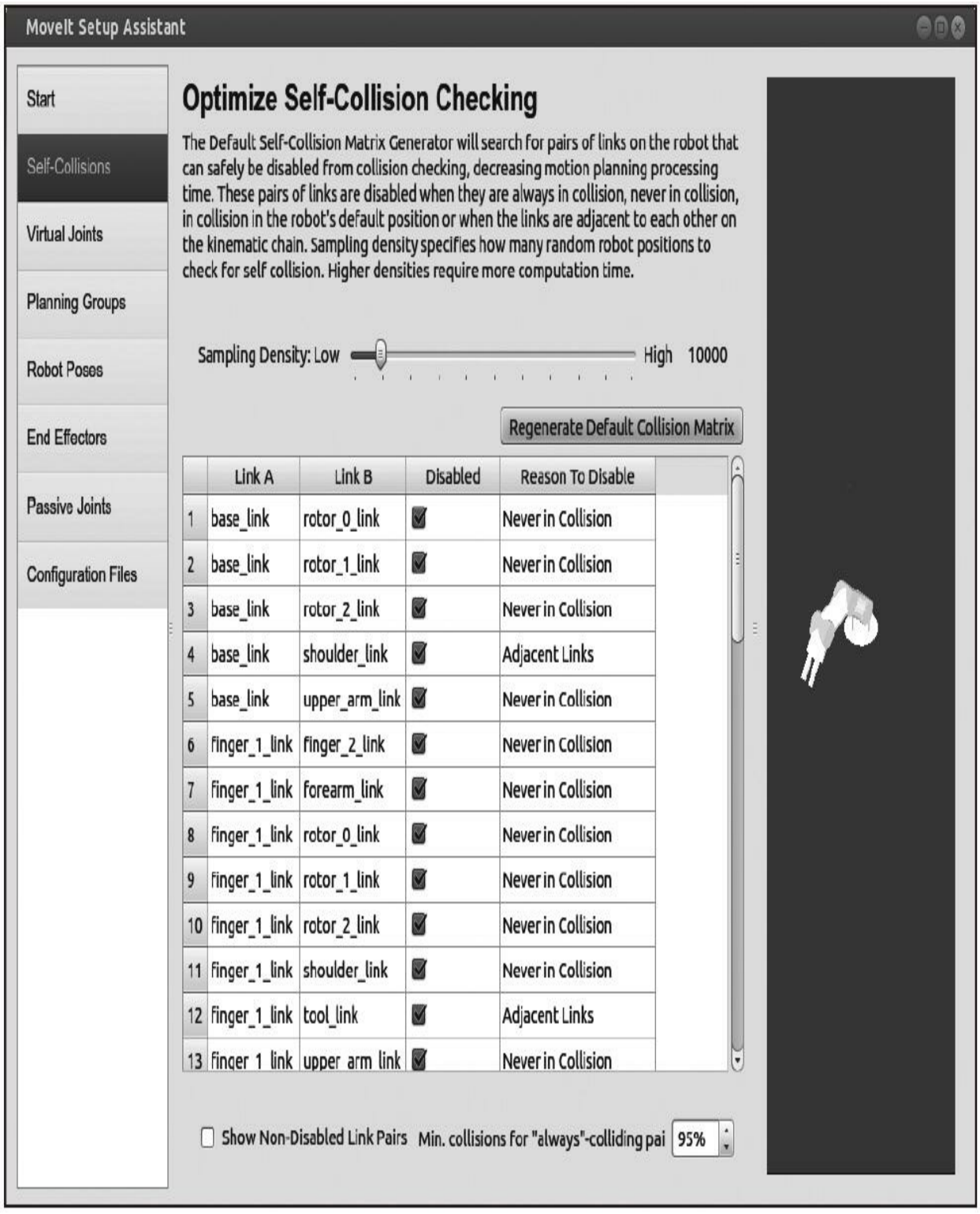


MoveIt!设置助手的初始界面

当命令执行后，会出现一个类似上图所示的窗口。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。一旦单击该按钮，配置助手便会请求机器人手臂的URDF或COLLADA模型，对于这里的机械臂，模型位于：rosbook_arm_description/robots/rosbook_arm_base.urdf.xacro的软件库功能包中。

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。当加载机器人描述文件时，用户需要浏览每个选项卡并添加所需的信息。第一个选项卡如下图所示，它用来生成自碰撞矩阵。

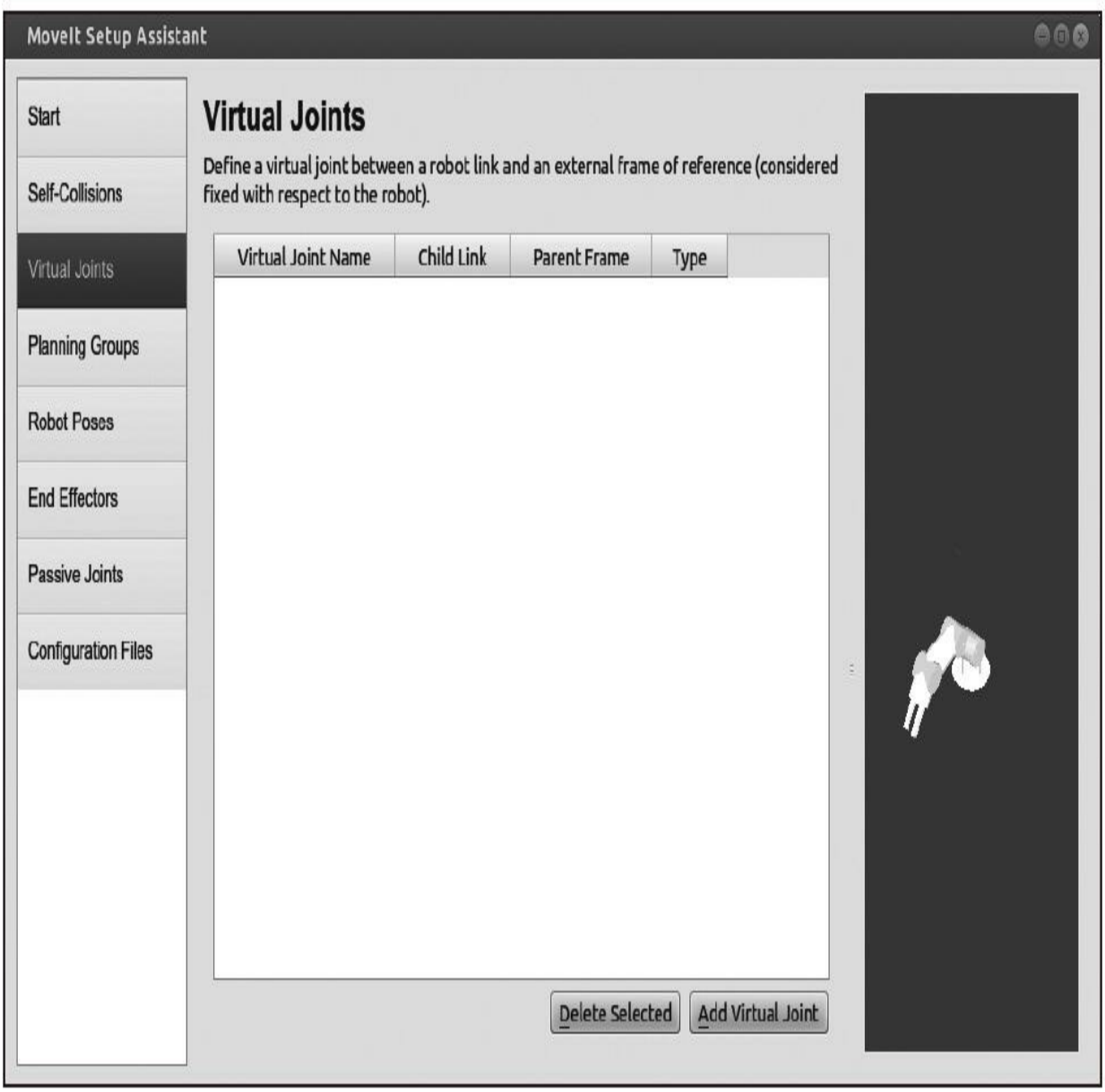
幸运的是，对于用户而言，这一过程可以通过简单地设置采样密度（或使用默认值）并单击Regenerate Default Collision Matrix（重新生成默认碰撞矩阵）按钮自动完成。为了提高运动规划器的性能，碰撞矩阵包含了有关连杆碰撞的信息。下图给出了详细的说明。



MoveIt!设置助手的Self-Collision选项卡

第二个选项卡如下图所示，用于分配机器人的虚拟关节。虚拟关节

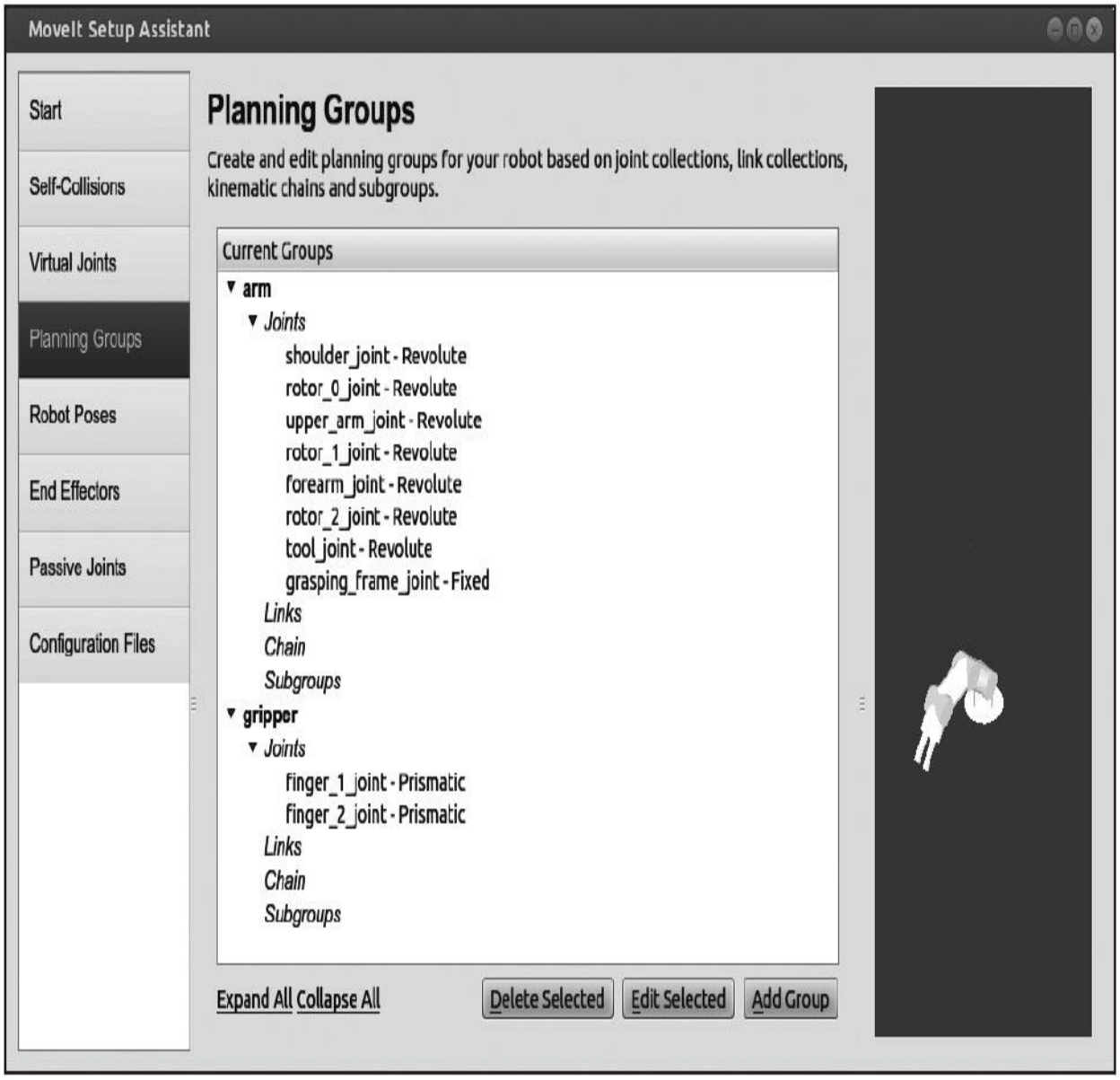
用于连接世界环境和机械臂，它随着机器人位姿的变化而改变，但在机械臂底座不会移动的特定情况下，不需要虚拟关节。当机械臂不固定在某一个地方时，就需要虚拟关节，例如，当机械臂在移动平台上的顶端时，就需要一个对应里程计的虚拟关节，这是因为base_link（底座）相对于里程计（odom）框架移动。



MoveIt!设置助手上的Virtual Joints选项卡

第三个选项卡如下图所示，需要定义机器人手臂的规划群组。顾名思义，规划群组是需要一起进行规划的一组关节，以实现特定的连杆或

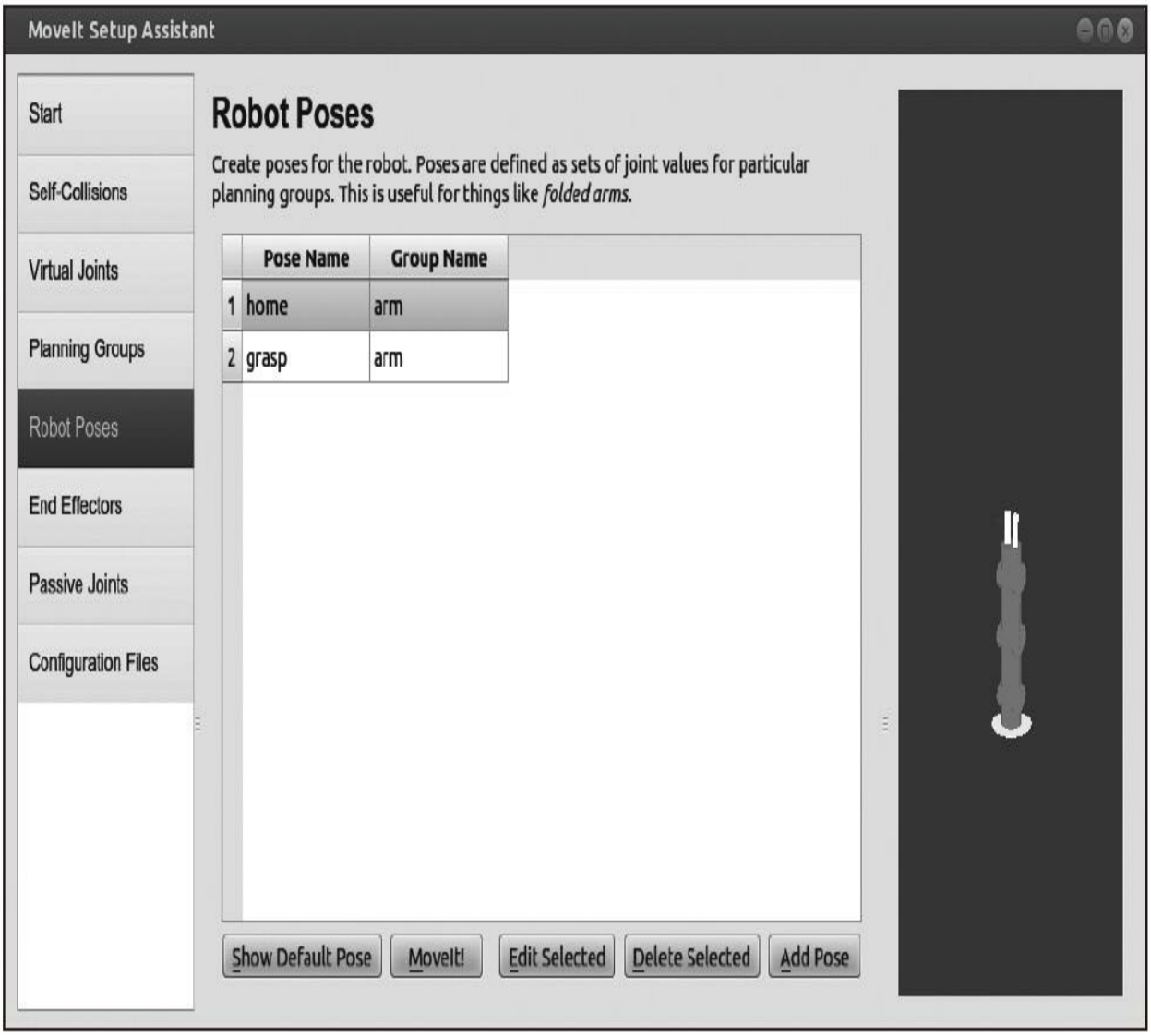
末端执行器上的给定目标。这里，需要定义两个规划群组：一个是机械臂本身，另一个是夹持器。然后分别对机械臂位姿和夹持器动作进行规划。



MoveIt!设置助手的Planning Groups选项卡

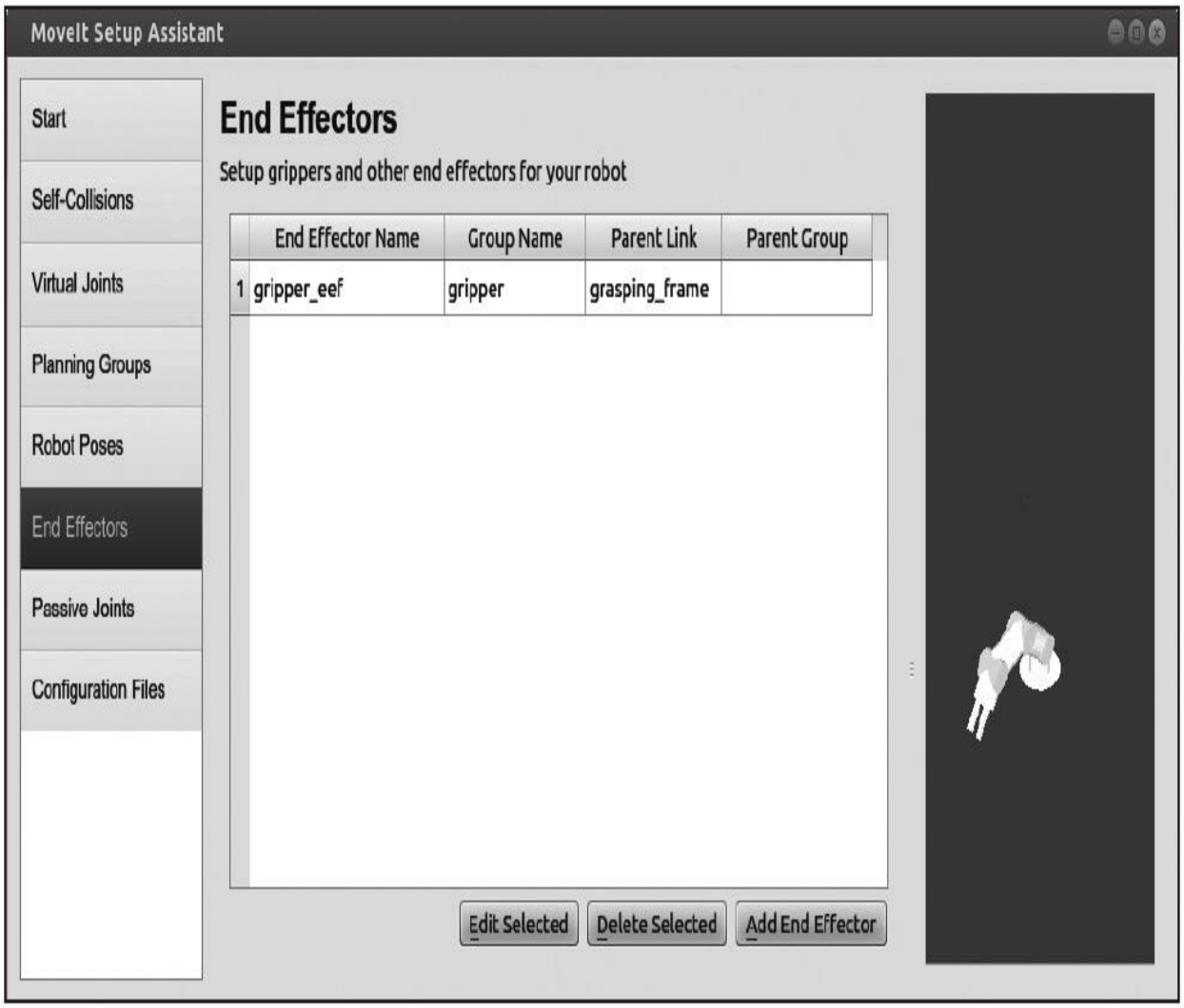
第四个选项卡如下图所示，可以定义机器人已知的位姿以便以后能够引用。这些预定义的位姿也称为群组状态。如图所示，我们已经建立了两个不同的位姿：机器人的初始位置，它对应于机械臂的“储存”位置以及夹持器位置，顾名思义，应该让机器人来抓握场景中的对象。实际情况下，设置已知的位姿有很多好处，例如，通常需要一个规划时的初

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。



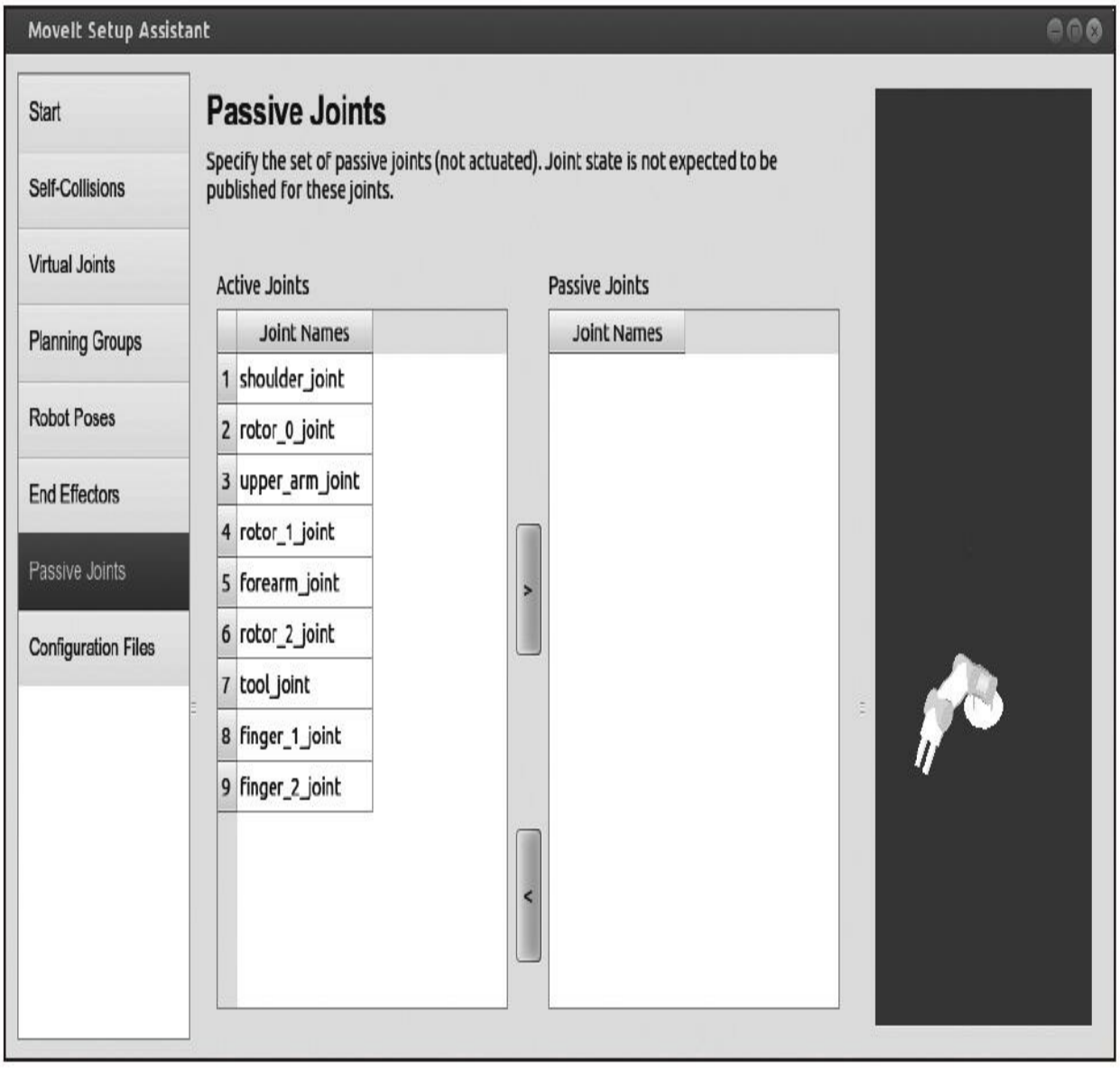
MoveIt!设置助手的Robot Poses选项卡

第五个选项卡定义机器人手臂的末端执行器，如下图所示。如先前讨论的，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。在本例中，末端执行器是夹持器，这使机器人能够抓取场景中的对象。在此选项卡上，需要通过为它分配一个名称、一个规划群组和含末端执行器的父连杆，定义夹持器的末端执行器。



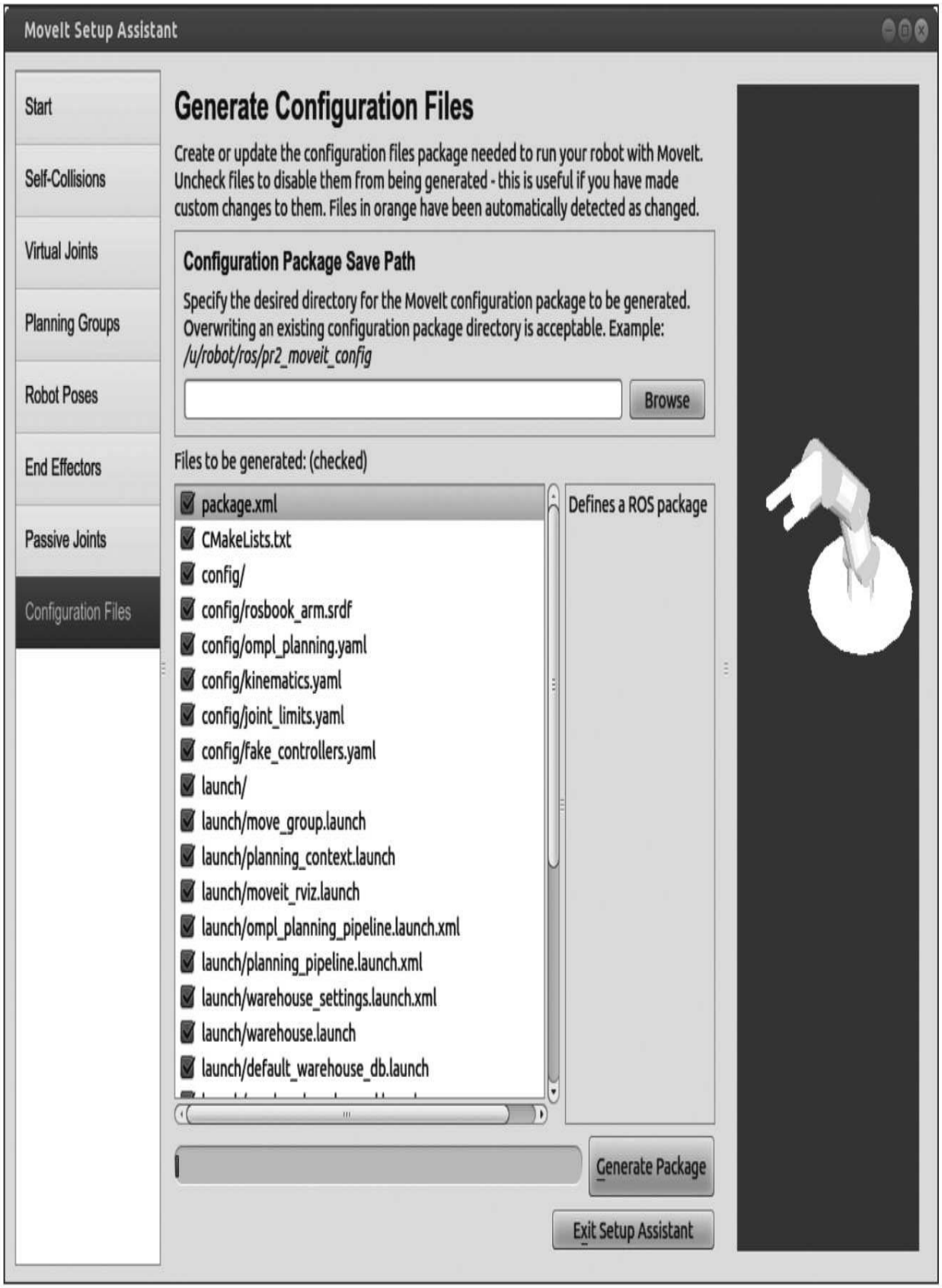
MoveIt!设置助手的End Effectors选项卡

如下图所示，第六个选项卡是定义不能驱动的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），但因为所有被动关节已经定义为固定的关节，所以这里会跳过此步骤，最终的运动规划将会产生相同的效果。



MoveIt!设置助手的Passive Joints选项卡

如下图所示，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事是提供MoveIt!创建的配置功能包的路径。配置功能包包含大部分从MoveIt!开始正确控制机器人手臂所需要的启动和配置文件。


MoveIt Setup Assistant
Start
Self-Collisions
Virtual Joints
Planning Groups
Robot Poses
End Effectors
Passive Joints
Configuration Files
Generate Configuration Files
Create or update the configuration files package needed to run your robot with MoveIt. Uncheck files to disable them from being generated - this is useful if you have made custom changes to them. Files in orange have been automatically detected as changed.
Configuration Package Save Path
Specify the desired directory for the MoveIt configuration package to be generated. Overwriting an existing configuration package directory is acceptable. Example: /u/robot/ros/pr2_moveit_config
Browse
Files to be generated: (checked)
package.xml
CMakeLists.txt
config/
config/rosbook_arm.srdf
config/ompl_planning.yaml
config/kinematics.yaml
config/joint_limits.yaml
config/fake_controllers.yaml
launch/
launch/move_group.launch
launch/planning_context.launch
launch/moveit_rviz.launch
launch/ompl_planning_pipeline.launch.xml
launch/planning_pipeline.launch.xml
launch/warehouse_settings.launch.xml
launch/warehouse.launch
launch/default_warehouse_db.launch
Defines a ROS package
Generate Package
Exit Setup Assistant

MoveIt!设置助手的Generate Configuration Files选项卡

尽管源代码中已经提供了由设置助手生成的配置文件，但是我们依然建议你完成这一过程，这一点非常重要，并依据喜好替换提供的功能包，它是参考源中其余启动脚本和配置文件完成的。

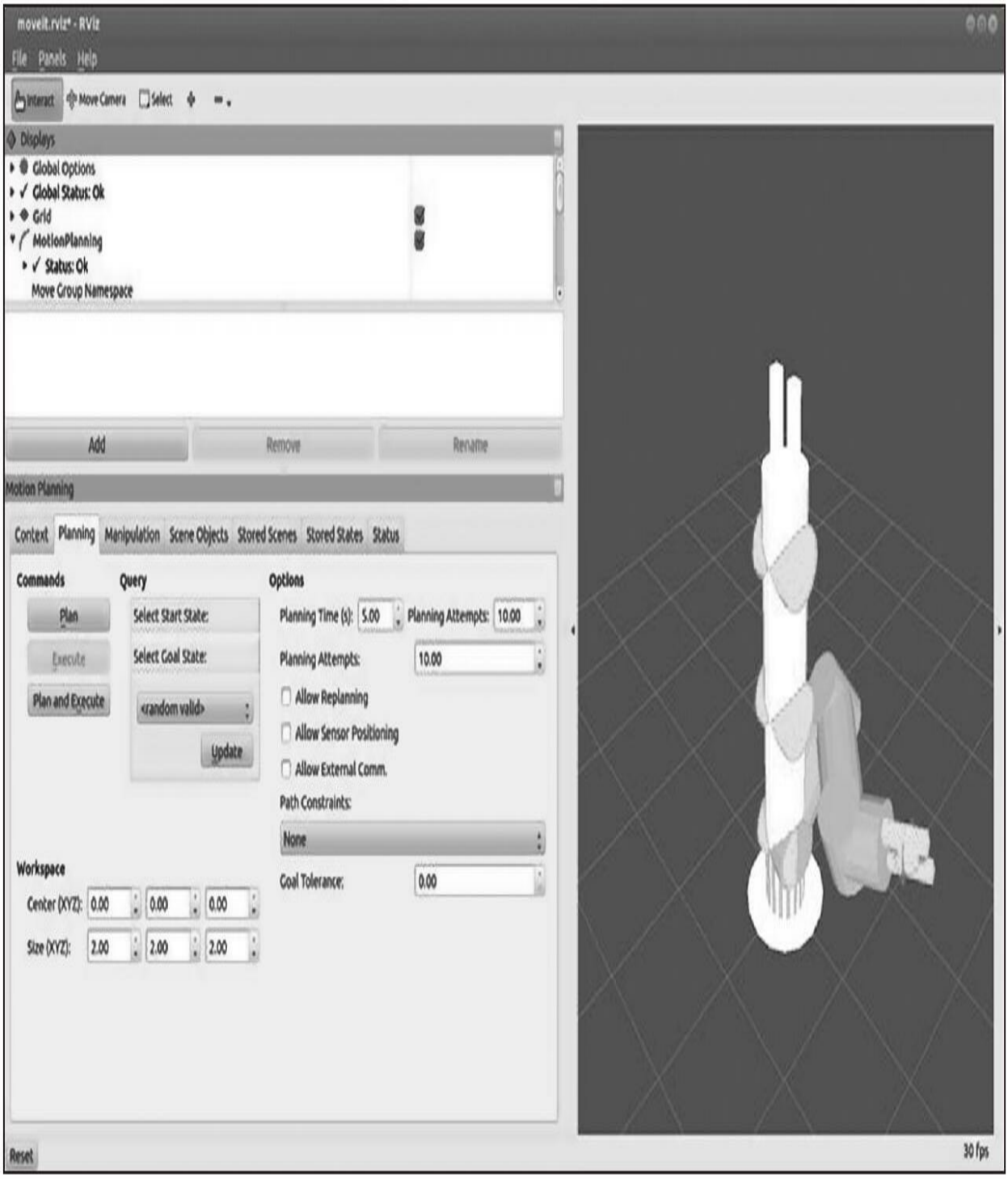
7.2.3 集成到RViz中

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derno文件，它负责启动MoveIt!以及仿真控制器、RViz和插件等。通过运行下面的命令启动演示示例：

```
$ 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Planning（规划）选项卡和Scene objects（场景对象）选项卡。在Planning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Scene objects选项卡中，可以在规划的场景中插入和删除对象。

下图是Planning选项卡，以及分别以白色和橙色可视化显示的机械臂（见彩插13）。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Query面板中的工具生成目标位置。一旦对目标状态满意，下一步或者通过可视化的方式演示手臂如何移动，或者执行它既可视化运动也移动机械臂本身。

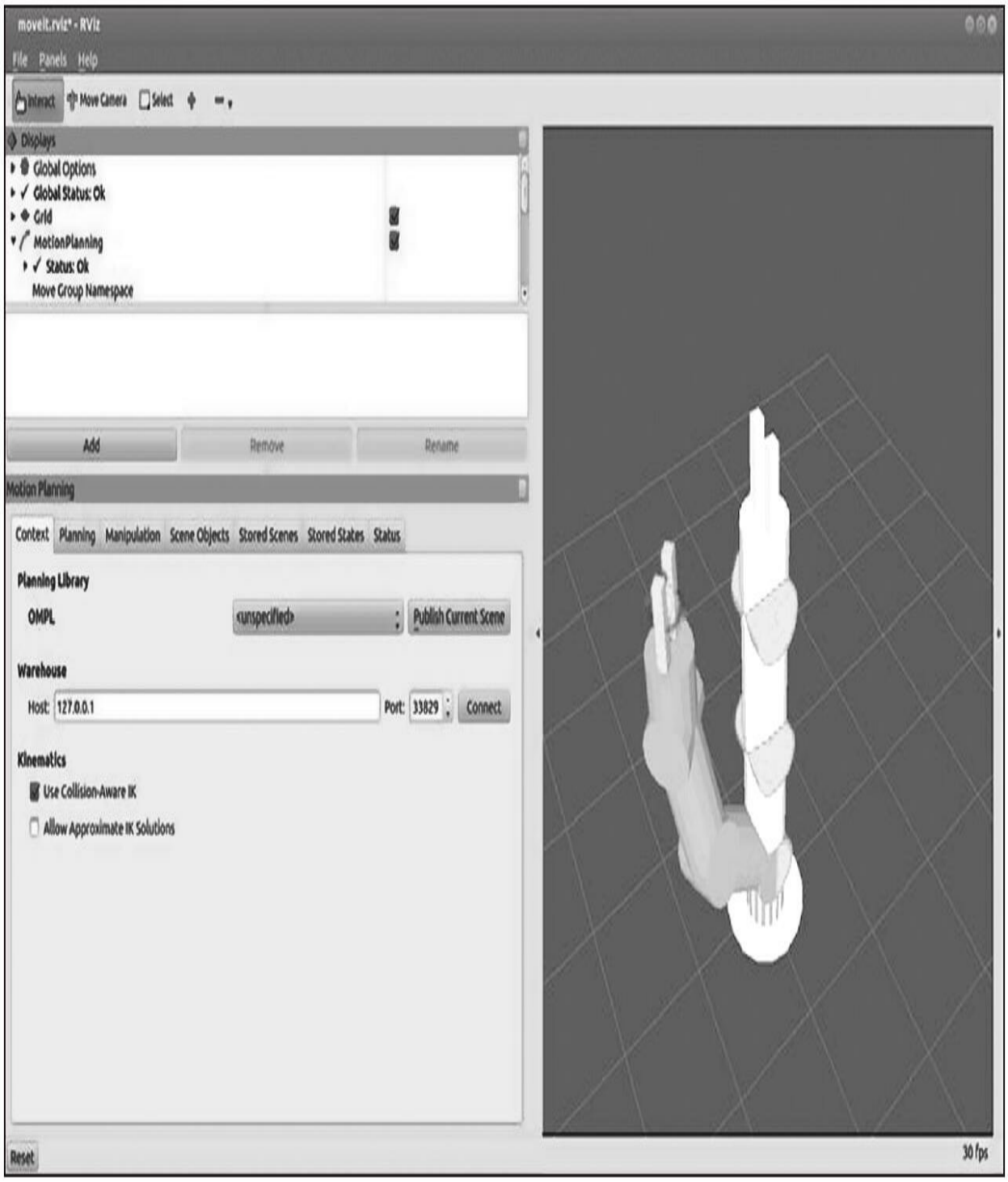


RViz插件中的Planning选项卡和目标位置可视化

其他选项（如时间规划和规划尝试次数）可以为复杂的目标进行相应的调整，但在大多数演示情况下不需要更改这些参数。另一个重要参数是目标容差（tolerance），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。

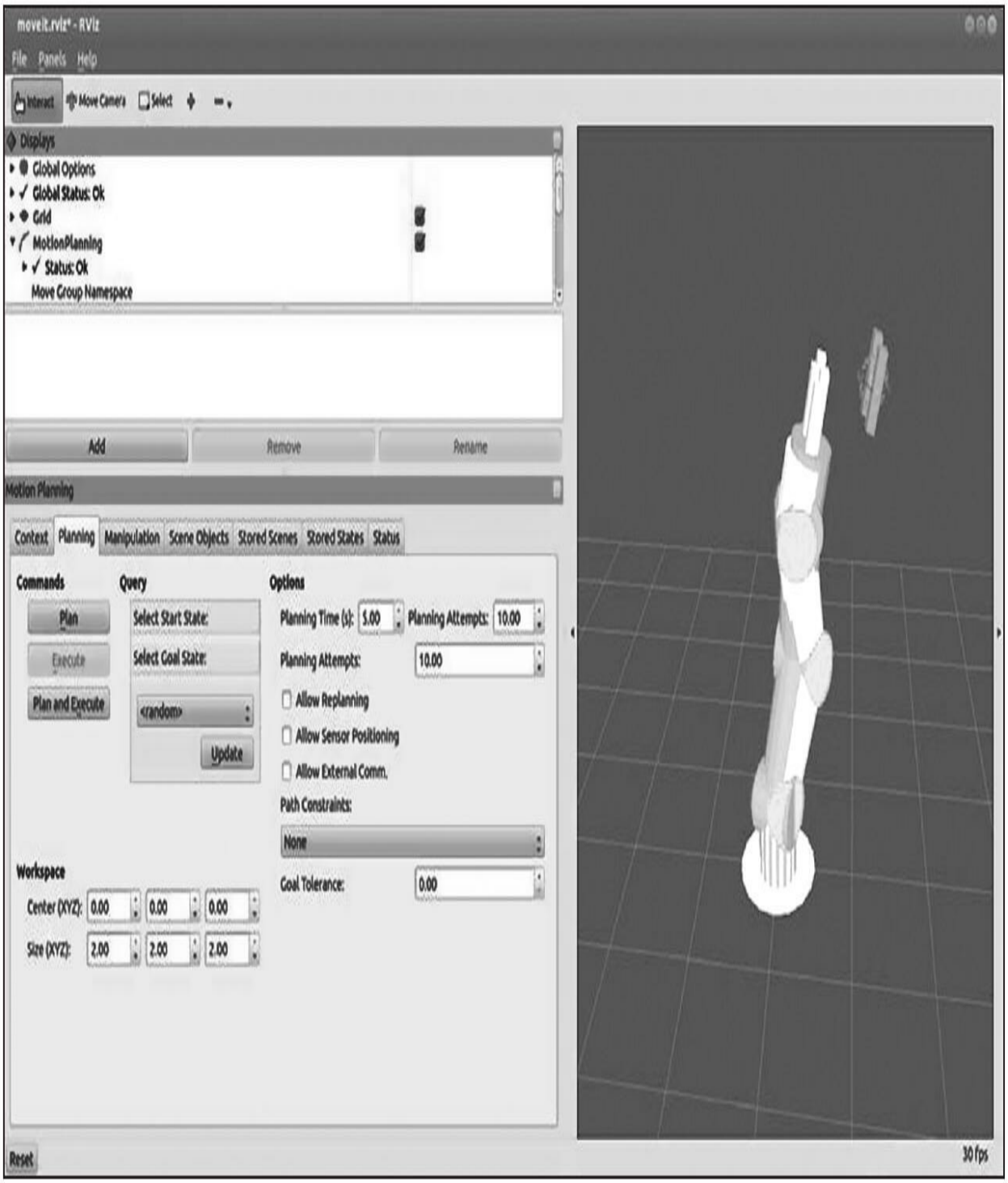
可能会出于某些兴趣来规划随机目标，RViz插件提供了另一个层面的规划。如下图所示，可视化机械臂的末端执行器上有一个标记，此标记允许我们定位机械臂的末端执行器，以及绕各个轴进行旋转。现在可以使用此标记来指定机械臂更有趣的配置。



在RViz插件中使用标记来设置目标位置

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在不同的位置上。这种情况的示例可以在下图中看到，它通常发生在目标位置超出机

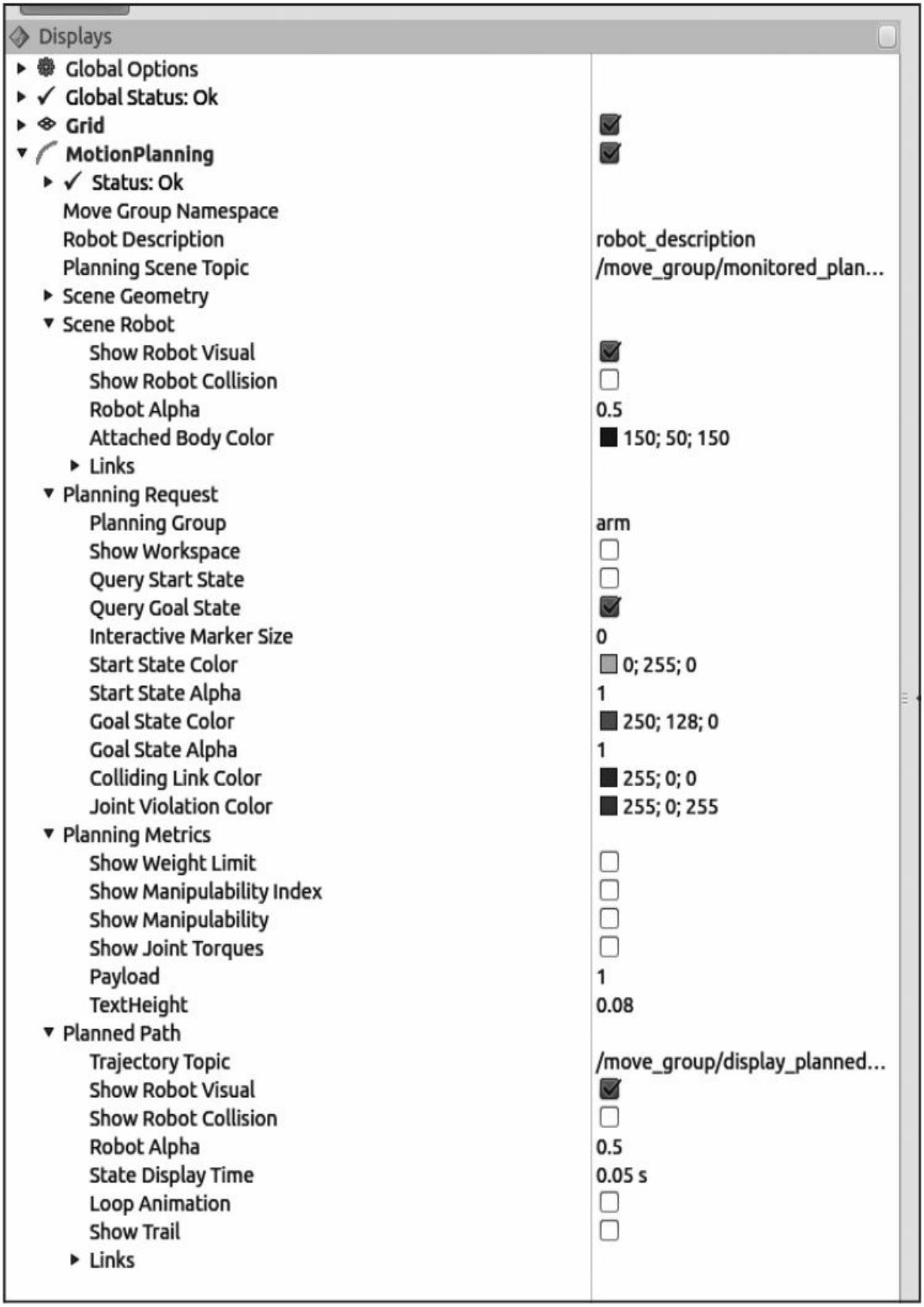
械臂的运动范围（如自由度不够、太多约束等）。



在RViz插件中标记在边界外的情况

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，下图显示了由MoveIt!插件的可视化界面提供的不同选项。


Displays
Global Options
Global Status: Ok
Grid
MotionPlanning
Status: Ok
Move Group Namespace
Robot Description
robot_description
Planning Scene Topic
/move_group/monitored_plan...
Scene Geometry
Scene Robot
Show Robot Visual
Show Robot Collision
Robot Alpha
0.5
Attached Body Color
150; 50; 150
Links
Planning Request
Planning Group
arm
Show Workspace
Query Start State
Query Goal State
Interactive Marker Size
0
Start State Color
0; 255; 0
Start State Alpha
1
Goal State Color
250; 128; 0
Goal State Alpha
1
Colliding Link Color
255; 0; 0
Joint Violation Color
255; 0; 255
Planning Metrics
Show Weight Limit
Show Manipulability Index
Show Manipulability
Show Joint Torques
Payload
1
TextHeight
0.08
Planned Path
Trajectory Topic
/move_group/display_planned...
Show Robot Visual
Show Robot Collision
Robot Alpha
0.5
State Display Time
0.05 s
Loop Animation
Show Trail
Links

RViz插件中的Motion Planning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Trajectory Topic（轨迹主题），这就是发布可视化轨迹的主题，以及Query Start State（查询初始状态），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的问题。

7.2.4 集成到Gazebo或实际机械臂中

将MoveIt!集成到Gazebo中是一个相对简单的过程，它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RGB-D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，如正常传感器一样，它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，因此在这种情况下使用MoveIt!我们需要配置机械臂关节的控制器，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，分别在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，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，其实现方式也必须改变。

7.3 简单的运动规划

RViz插件提供了一个与MoveIt!进行交互的非常有趣的机制，但由于它缺乏自动化，可能会认为它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。

在下一节中，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。我们将开始规划单个目标，然后规划一个随机目标，接着进行预定义群组状态的规划，最后解释如何改进我们与RViz交互的代码段。

为了简化解释，提供了一组启动文件启动全部所需的功能。其中最重要的一个启动文件负责启动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，它用于启动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rosrun（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，我们需要定义那些将要执行运动规划的规划群组。在本例中只有两个规划群组，即机械臂和夹持器，但这里我们只关注臂。这将实例化一个规划群组接口，它负责与MoveIt!进行交互，如下：

```
moveit::planning_interface::MoveGroupplan_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，需要传达给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，将执行它：

```
moveit::planning_interface::MoveGroup::Plan goal_plan;
if (plan_group.plan(goal_plan))
{
  .plan_group.move();
}
```

7.3.1 规划单个目标

要规划单个目标，从字面上理解只需要将目标本身提供给MoveIt!。目标由geometry_msgs功能包中的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实现了这一目标。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，还可以设置容差。我们都知道PID控制并非很准确，它可能使MoveIt!认为控制目标没有完成。更改目标容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
```

7.3.2 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，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，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::ServiceClientvalidity_srv =
nh.serviceClient<moveit_msgs::GetStateValidity>("/check_state_vali
dity");
```

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，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，可以通过获取机器人的当前状态对象来启动，如下：

```
robot_state::RobotStatecurrent_state =
*plan_group.getCurrentState();
```

接着为当前机器人状态对象设置随机的位置，但要这样做，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已创建的机器人状态对象获取：

```
current_state.setToRandomPositions(current_state.getJointModelGrou
p("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API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，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，此刻，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：

```
$ rosrun rosbook_arm_snippets move_group_plan_random_target
```

7.3.3 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，之后可用于使机器人处于一个预定义的位姿。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::RobotStatecurrent_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，这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：

```
$ rosrun rosbook_arm_snippets move_group_plan_group_state
```

7.3.4 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从而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::DisplayTrajectorydisplay_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的可视化效果，如果运动轨迹显示两次，不要混淆。

7.4 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，本节将介绍如何将物体添加到可能与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传感器输入，它会基于真实环境（或模拟）的对象产生点云，从而使运动规划更有趣和实用。

7.4.1 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需要可视化包含在其中的对象。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::PlanningSceneInterfacecurrent_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消息，设定正在创建的对象的类型，并根据对象的类型设定其属性。在本例中，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，需要在规划场景中提供方盒的位姿。因为我们需要产生可能碰撞的情况，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中的位姿消息设定：

```
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;
```

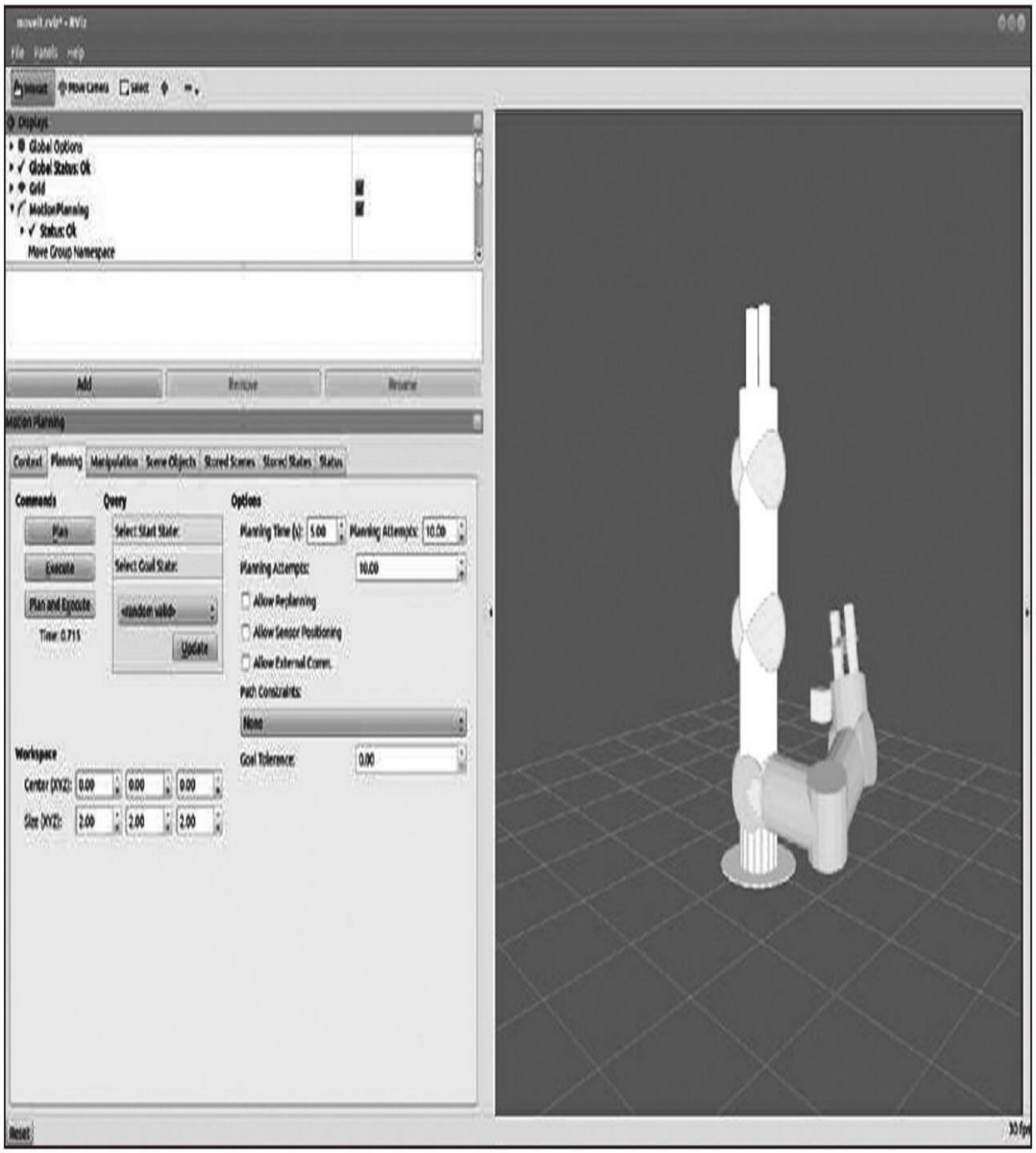
最后，添加碰撞对象到其消息向量中并调用规划场景接口中的addCollisionObjects方法。它负责通过适当的主题发送需要的消息，以确保在当前规划场景中创建对象：

```
std::vector<moveit_msgs::CollisionObject>collision_objects;
collision_objects.push_back(box);

current_scene.addCollisionObjects(collision_objects);
```

与前面相同，可以通过在终端中运行下面的命令测试该代码段。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
```

结果如下图所示，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（见彩插14）。



在RViz中的场景碰撞对象

7.4.2 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待移除对象ID的字符串向量，并调用规划场景界面中的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Scene objects选项卡从场景中删除任何对象。

7.4.3 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，把由此产生的点云数据发布到/rgbd_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，它将向MoveIt!提供足够的信息，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: /rgbd_camera/depth/points
  max_range: 10
  padding_offset: 0.01
  padding_scale: 1.0
  point_subsample: 1
  filtered_cloud_topic: output_cloud
```

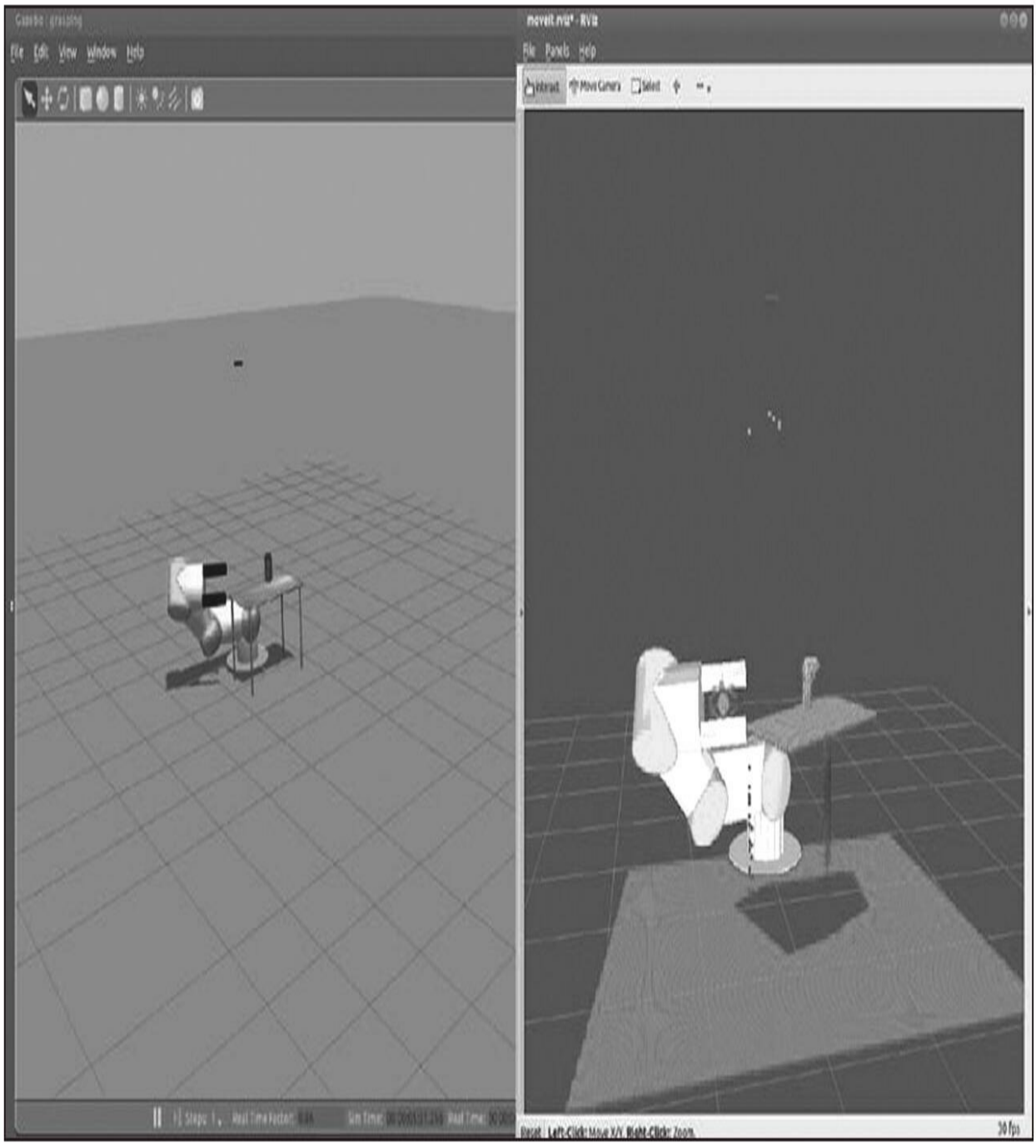
你可能会疑惑这个文件本身只不过是一个配置文件。下一步需要执行的是加载此配置文件到环境中，这样MoveIt!就知道我们已经添加了新的传感器。为了做到这一点，还需要修改下面的XML文件：

```
$ rosbook_arm_moveit_config/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>
```

运行增加的新命令后的结果如下图所示。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，这成功地应用点云避免了碰撞。



Gazebo仿真（左），RViz中的点云（右）

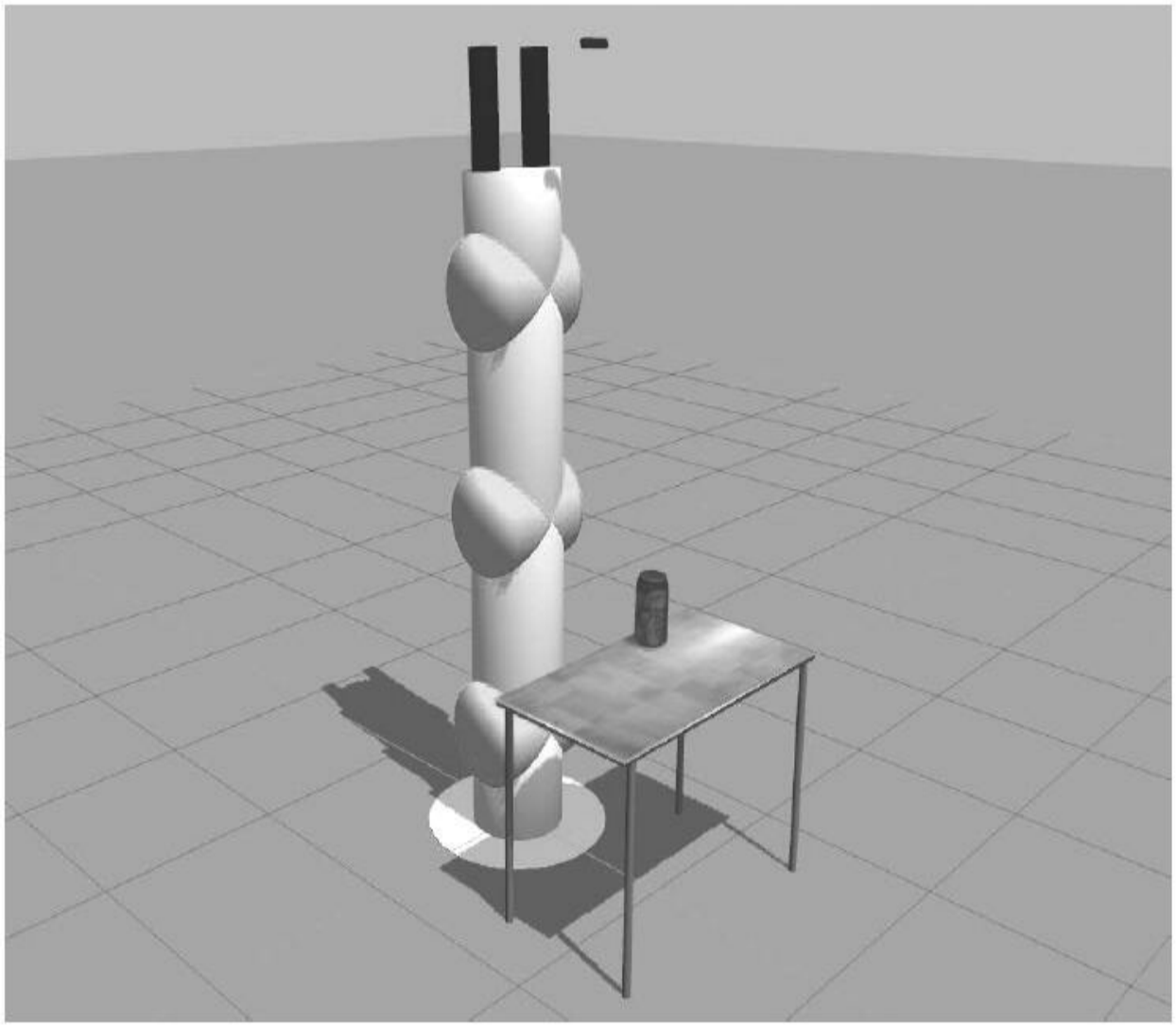
7.5 抓取和放置任务

本节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，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，但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。

下面的小节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API完成抓取和放置的任务。最后，我们将解释在演示模式下，如何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。

7.5.1 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下图所示。



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。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。

7.5.2 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，它可以近似为一个方盒，它是MoveIt!规划场景API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))

    returnp.pose
```

这里重要的部分是add_box方法将对象box添加到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和时间戳now。为了使用规划框架，需要RobotCommander，它是MoveIt!命令机械臂执行程序的接口：

```
self._robot = RobotCommander()
```

7.5.3 支撑面

用和创建方盒相同的方式，创建桌子对象。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)
```

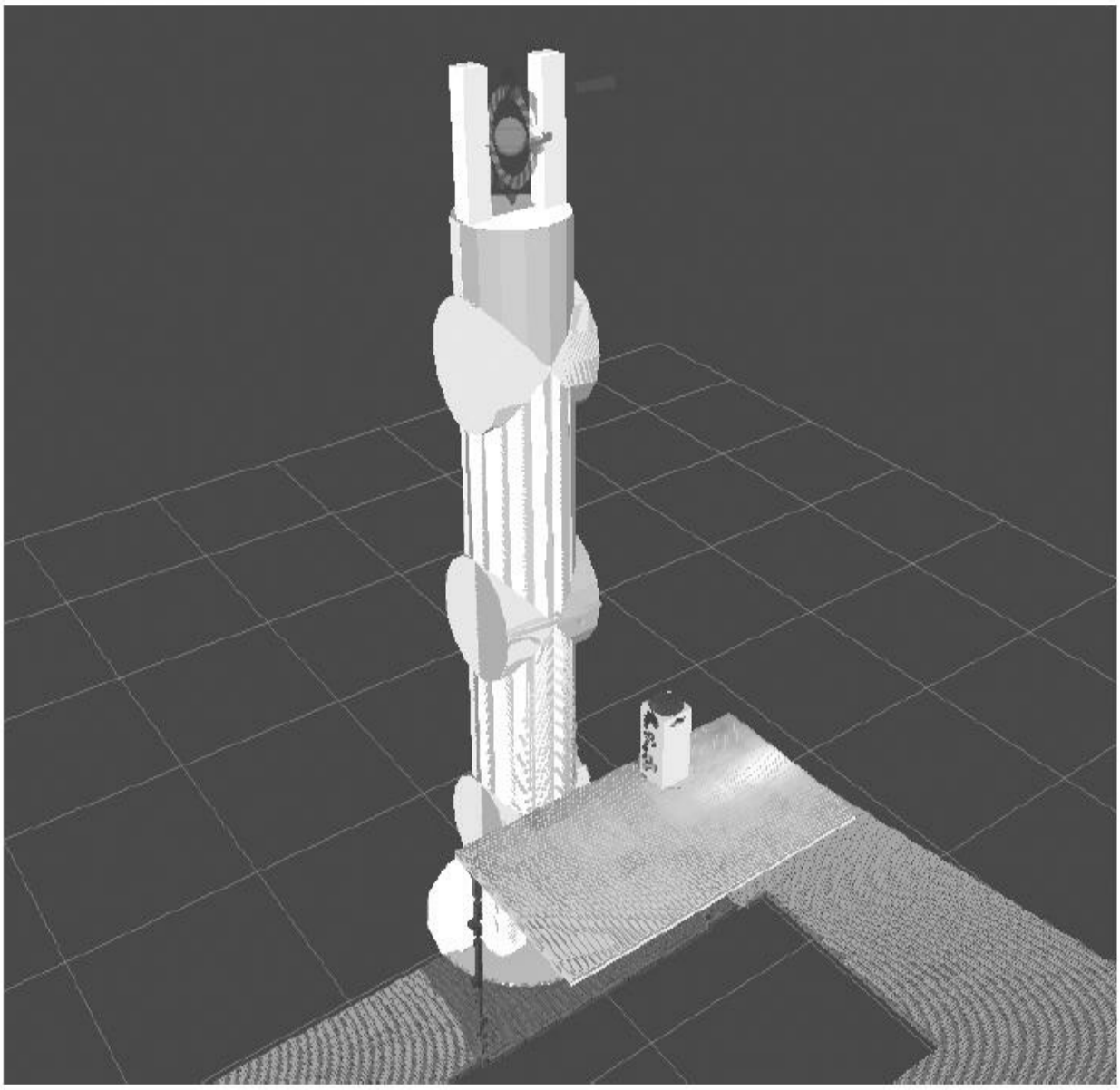
_add_table方法添加桌子到规划场景中：

```
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))

    returnp.pose
```

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
```

这实际上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（见彩插15），完美匹配RGB-D传感器感知的点云，如下图所示。



场景中RGB-D传感器获取的点云信息

7.5.4 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中。在这种情况下，桌子可以作为点云上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中。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。通过在对象上放置基准标记可以帮助感知完成抓握，例如ArUco（http://www.uco.es/investiga/grupos/ava/node/26），它有ROS封装，这可以查看网页https://github.com/pal-robotics/aruco_ros。

在这里手动设置规划场景，感知部分留给你去探索。正如我们看到的，通过与RViz的点云数据比较对应关系，直到匹配效果很好，可以在代码中手动定义待抓取的目标对象和支撑面。

7.5.5 抓取

现在，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，使用moveit_simple_grasps功能包中的抓握生成服务器，源码在https://github.com/davetcoleman/moveit_simple_grasps下。

遗憾的是，对于ROS Kinetic没有可用的debian功能包。因此，需要运行以下命令来将修补的分支（kinetic-devel）添加到工作空间中（在工作空间的src文件夹下）：

```
$ wstool set moveit_simple_grasps --git https://github.com/davetcoleman/
moveit_simple_grasps.git -v kinetic-devel

$ wstool up moveit_simple_grasps
```

使用下列命令编译这个文件：

```
$ cd ..

$ caktin_make
```

现在可以运行抓握生成服务器，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，抓握生成服务器需要如下抓握数据配置：

```
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
```

这基本上定义了用于抓取对象的夹持器以及抓取前、后的位姿。

现在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，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：

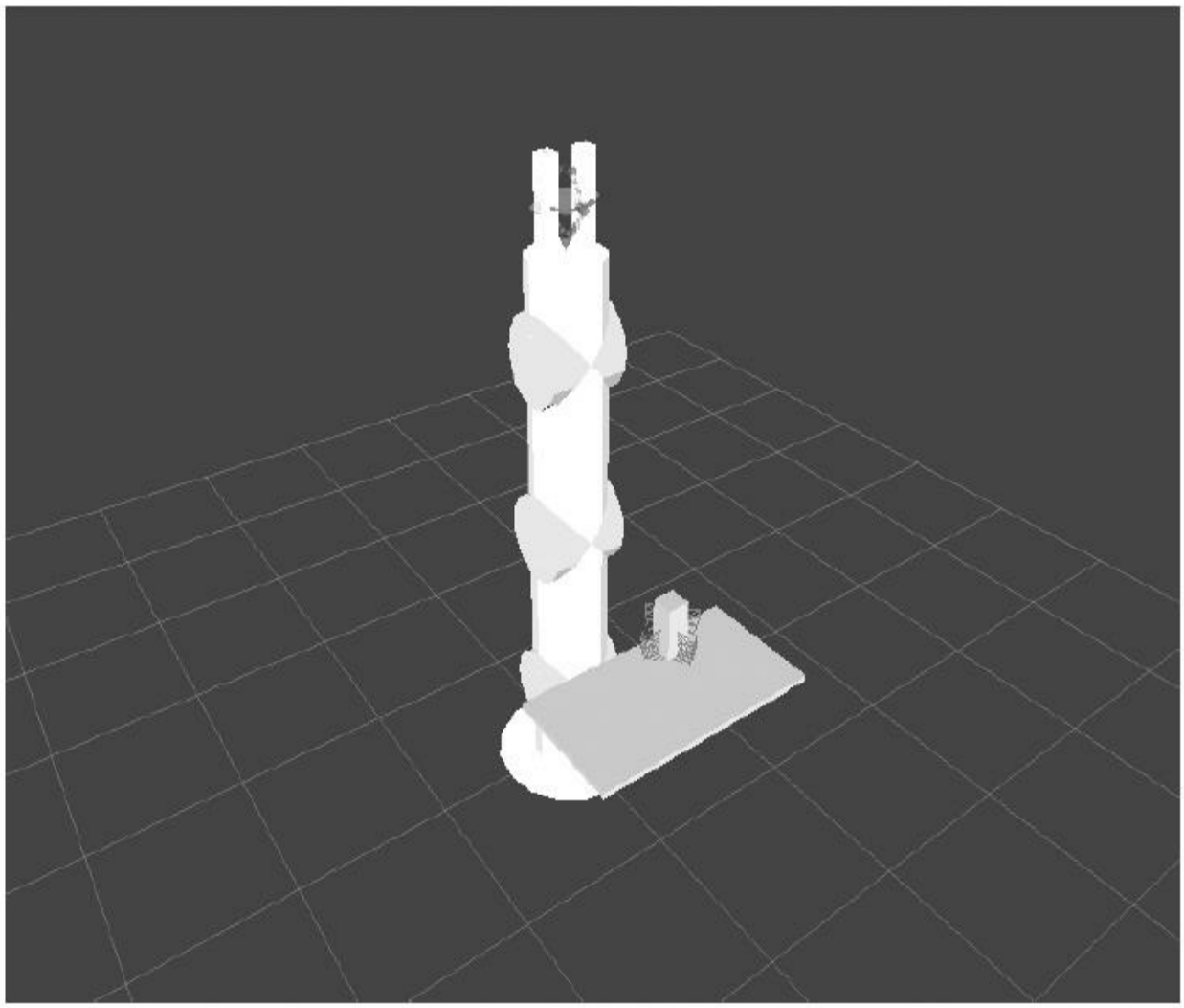
```
def _generate_grasps(self, pose, width):
    # Create goal:
    goal = GenerateGraspsGoal()

    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，但它们只能用于查询特定类型的抓握，如某些角度或指向上或指向下。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，抓握生成服务器所提供的抓握位姿也使用_publish_grasps方法作为PoseArray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

$ rosrun 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，它们对应于尝试把它拿起来的抓取位姿，如下图所示。



抓取位姿的可视化

7.5.6 抓取操作

一旦有了抓握的位姿，就可以使用MoveIt!的/pickup操作服务器发送所有这些目标。和前面一样，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，多次尝试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，为MoveIt!创建了一个抓取目标，然后如前所述，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
```

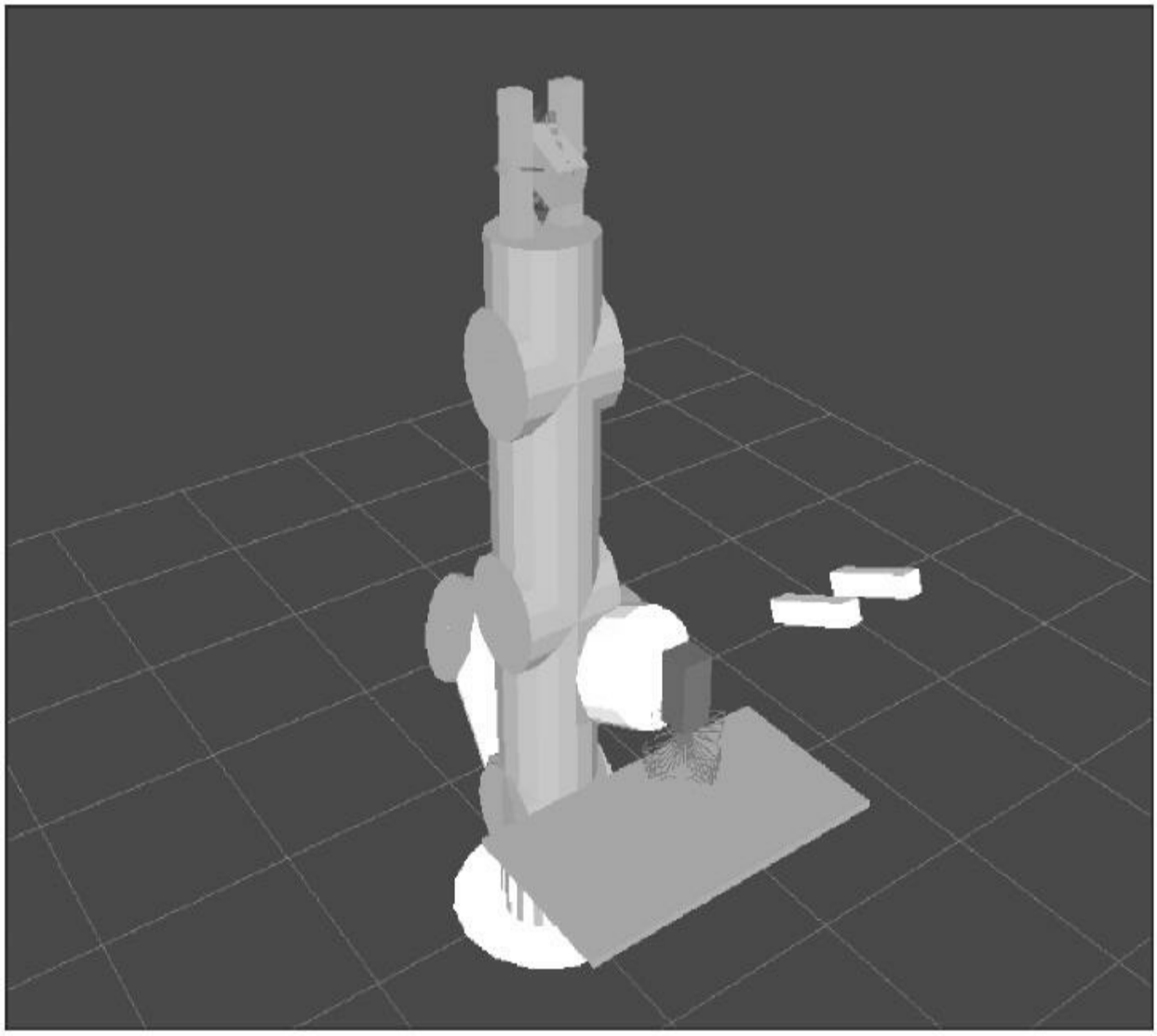
发送目标，并使用状态来检查机械臂是否已经拿起对象。在_create_pickup_goal方法中创建拿起的目标，如下所示：

```
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中，在界面中对应的方盒以紫色的颜色显示，如下图所示（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）。



机械臂抓取一个对象

7.5.7 放置操作

当拿起对象后，机械臂将继续进行放置操作。MoveIt!提供/place操作服务器，所以第一步包括创建一个操作客户端以发送一个期望位置中的放置目标，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，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)
```

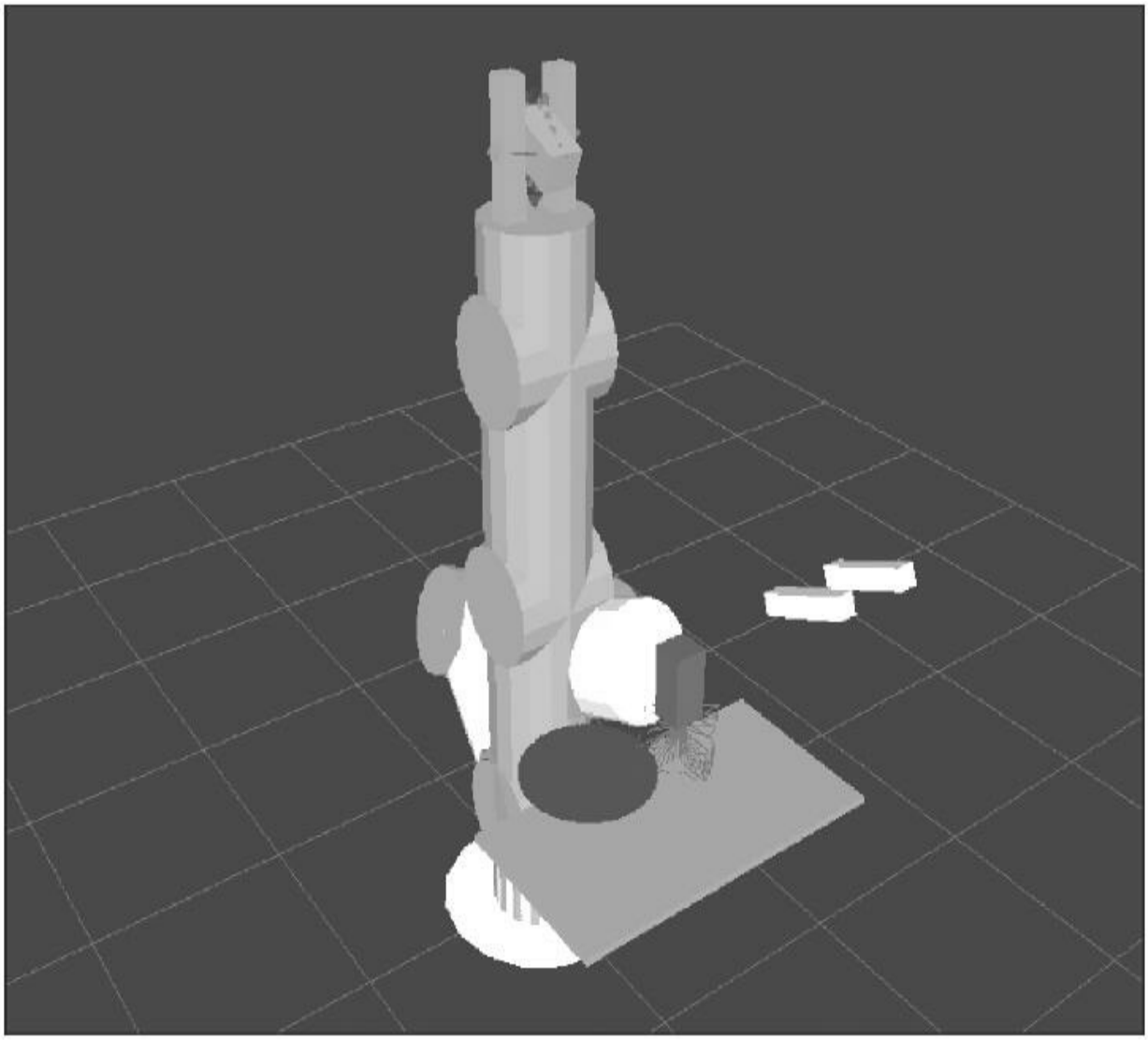
_place方法：

```
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
```

该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放置。为了放置对象，可以使用单个放置位姿，但一般最好多提供几个选项。这里，使用_generate_places方法，在给定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（见彩插16），如下图所示。



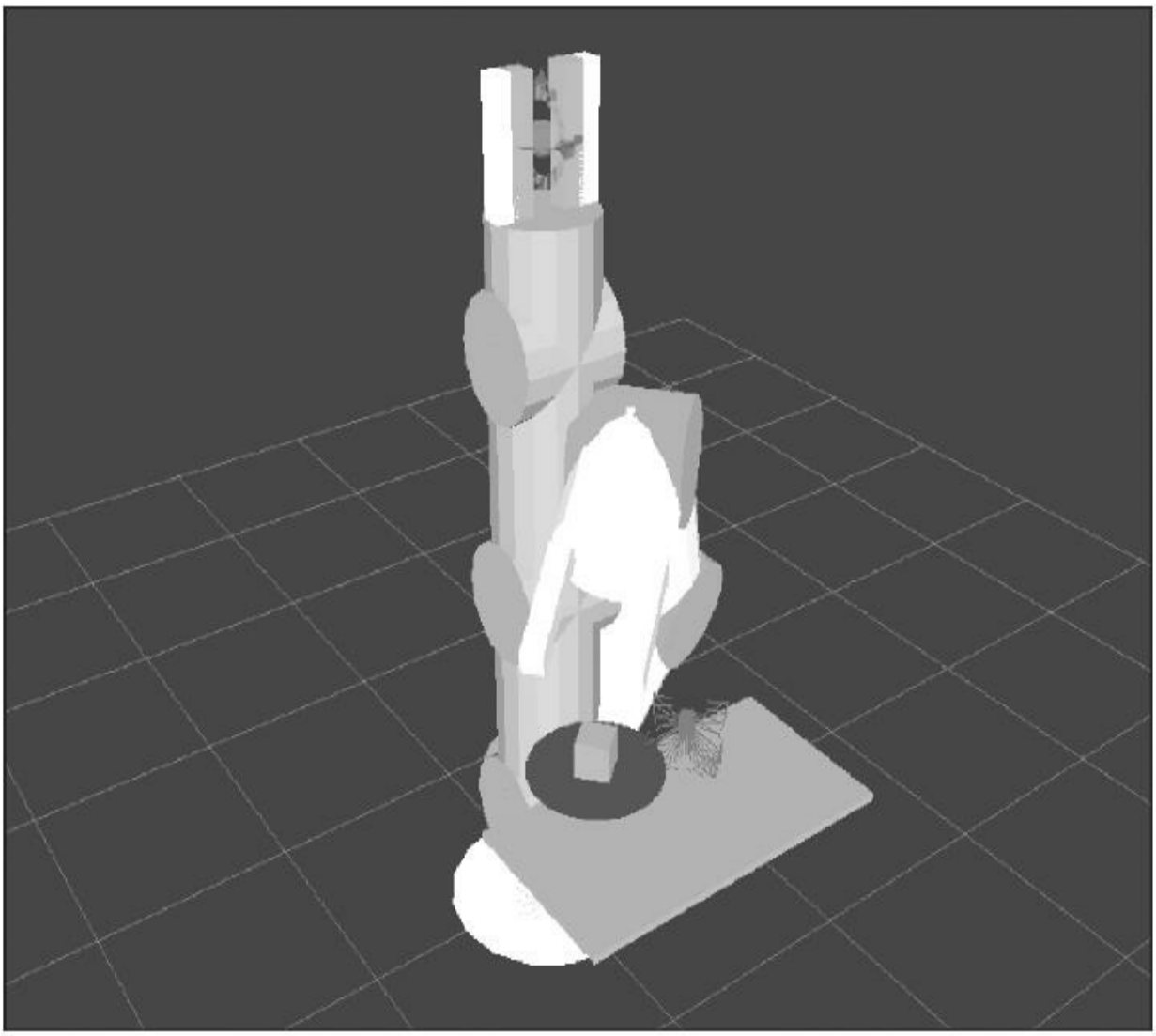
放置位姿的可视化

一旦获得位置，_create_place_goal方法创建一个放置目标，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，它可以按需要增加。当放置对象时，我们将在桌子上看到方盒再次以绿色显示（见彩插17），接着机械臂会抬起来，如下图所示。



放置完对象后的机械臂

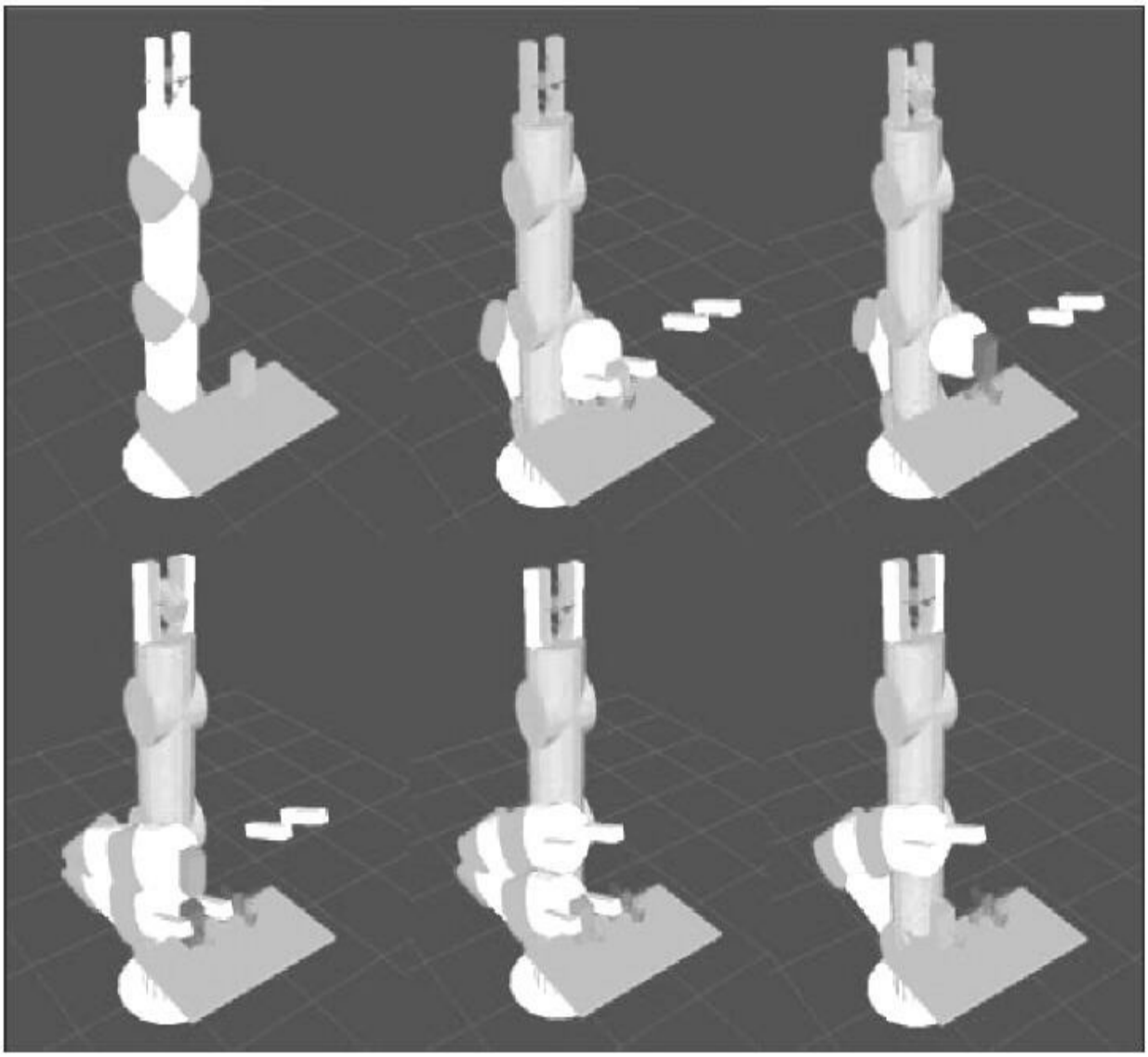
7.5.8 演示模式

在演示模式下，在无人觉察的情况下（也就是，没有在Gazebo仿真或者实际的机器臂上实际执行的情况下），也可以完成整个抓取和放置任务。在这种模式下，一旦Moveit！发现运动计划，虚假控制器就会移动机器臂完成抓取和放置操作，包括抓住物体。同样的模式可以直接用于实际的控制器。

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_moveit_config demo.launch

$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch

$ rosrunros book_arm_pick_and_place pick_and_place.py
```

特别是第一个启动文件，它直接打开RViz，加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下图所示。



在演示模式下手臂做抓取和放置的任务

7.5.9 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中机器臂的环境包含桌子、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。

7.6 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（IK）解算器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，才能使它正常工作。

我们已经了解了MoveIt!API中不同的高级元素，详细介绍它们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与规划场景中人工创建的对象和生成点云的RGB-D传感器进行交互。

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，但它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，可以在复杂的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。

第8章　在ROS下使用传感器和执行器

当你想到机器人时，可能会想起人形机器人，它有手臂、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。首先，我们将学习如何使用ROS来控制DIY机器人平台。然后，将学习如何将机器人实验室中常用的传感器和执行器连接起来。

 注意：在http://www.ros.org/wiki/Sensors有一个很长的设备清单，上面都是ROS支持的设备。

传感器和执行器分成不同的类别：测距仪、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。

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

·使用Arduino连接更多的传感器和执行器

·如Kinect之类的3D传感器、2D激光测距仪、GPS和伺服电动机

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。使用这些设备，将其连接来构建一个小型机器人平台。

8.1 使用游戏杆或游戏手柄

可以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。要安装这些功能包到Ubuntu系统中，首先执行下面的命令：

```
$ sudo apt-get install ros-kinetic-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，将游戏手柄连接到电脑。使用如下代码，检查游戏手柄是否能够被识别：

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

我们会看到下面的输出：

```
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
```

我们的罗技（Logitech　F710）游戏杆有8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中测试它的功能。为了达到测试的目的，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.
```

8.1.1 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，使用下面的命令：

```
$ rostopic echo /joy
```

然后我们看到每次发出的消息：

```
header:
  seq: 33
  stamp:
    secs: 1480289803
    nsecs: 599782892
  frame_id: ''
axes: [-0.0, -0.2219386249780655, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

你会看到两个主要的向量：一个代表轴向输入，另一个代表按钮。很明显，这些向量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，在命令行窗口中输入下面的指令：

```
$ 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：

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

然后你会看到：

这就是你在使用游戏杆时将会使用的消息结构。在下一节中，你将会学到如何写一个订阅游戏杆主题的节点，以及如何生成用于移动机器人模型的指令。

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32[] axes
int32[] buttons
```

8.1.2 使用游戏杆数据移动机器人模型

现在，我们将创建一个节点，它从joy_node获取数据，并发布主题控制机器人模型。

这时，需要创建一个类似我们在前面章节中所看到的机器人三维模型和一个带有里程计的节点。在软件库中，查找chapter8_tutorials/urdf文件夹下的urdf模型，并在src文件夹下名为c8_odom.cpp的文件中检查里程计代码。

在该代码中，使用初始位置（0,0,0）并使用geometry_msgs::Twist中的速度信息来计算里程。计算每个车轮的速度，并使用差速驱动运动学原理更新位置。在运行roscore命令后，在终端中键入下面的命令可以运行里程计节点：

```
$ rosrun chapter8_tutorials c8_odom
```

要了解此节点中使用的主题，可以使用下面的命令显示主题列表：

```
$ rosnode info /odom
```

你会看到以下输出，其中/cmd_vel是该节点发布的主题：

```
--------------------------------------------------------------------------------
Node [/odom]
Publications:
 * /odom [nav_msgs/Odometry]
 * /rosout [rosgraph_msgs/Log]
 * /tf [tf2_msgs/TFMessage]

Subscriptions:
 * /cmd_vel [unknown type]

Services:
 * /odom/get_loggers
 * /odom/set_logger_level


contacting node http://daneel:35582/ ...
Pid: 30375
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
```

现在我们需要知道主题的类型。请使用下面的命令查看它：

```
$ rostopic type /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：

```
$ rosmsg 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chapter8_tutorials/src文件夹下创建一个新的文件c8_teleop_joy.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;
float max_linear_vel = 0.2;
float max_angular_vel = 1.5707;

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()
{ i_velLinear = 1;
  i_velAngular = 0;
  n.param("axis_linear",i_velLinear,i_velLinear);
  n.param("axis_angular",i_velAngular,i_velAngular);
  pub = n.advertise<geometry_msgs::Twist>("/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 = max_angular_vel*joy->axes[0];
  vel.linear.x = max_linear_vel*joy->axes[1];
  pub.publish(vel);
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "c8_teleop_joy");
  TeleopJoy teleop_joy;
  ros::spin();
}
```

现在解释上面的代码。在main函数中，创建TeleopJoy类的一个实例。

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


  TeleopJoy teleop_joy;
}
```

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()
{ i_velLinear = 1;
  i_velAngular = 0;
  n.param("axis_linear",i_velLinear,i_velLinear);
  n.param("axis_angular",i_velAngular,i_velAngular);
  pub = n.advertise<geometry_msgs::Twist>("/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。使用vel名称创建一个新的变量，它用于发布数据。使用为最大速度赋值的系数，把游戏杆轴向输入的值赋给vel变量。在这部分里，你能够在发布数据之前对接收到的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 = max_angular_vel*joy->axes[0];
  vel.linear.x = max_linear_vel*joy->axes[1];
  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c8_teleoop_joy节点：

```
<?xml version="1.0" ?>
<launch>
  <node pkg="chapter8_tutorials" type="c8_teleop_joy"
  name="c8_teleop_joy" />
  <param name="axis_linear" value="1" type="int" />
  <param name="axis_angular" value="0" type="int" />
  <node respawn="true" pkg="joy" type="joy_node" name="joy_node">
    <param name="dev" type="string" value="/dev/input/js0" />
    <param name="deadzone" value="0.12" />
  </node>
</launch>
```

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。

```
$ roslaunch chapter8_tutorials chapter8_teleop_joy.launch
```

可以使用rostopic　echo检查代码运行是否正常，当移动操纵杆时，将发布/cmd_vel主题。

现在，我们将准备一个更大的launch文件，用来接收手柄指令可视化机器人模型。将下面的代码复制到chapter8_tutorials/launch文件夹的新文件chapter8_tutorials_robot_model.launch中：

```
<?xml version="1.0"?>
<launch>
  <arg name="model" />
  <arg name="gui" default="False" />
  <param name="robot_description" textfile="$(find
  chapter8_tutorials)/urdf/robot2.urdf" />
  <param name="use_gui" value="$(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" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
  chapter8_tutorials)/config/config.rviz" />
  <node name="c8_odom" pkg="chapter8_tutorials" type="c8_odom" />
  <node name="joy_node" pkg="joy" type="joy_node" />
  <node name="c8_teleop_joy" pkg="chapter8_tutorials"
  type="c8_teleop_joy" />
</launch>
```

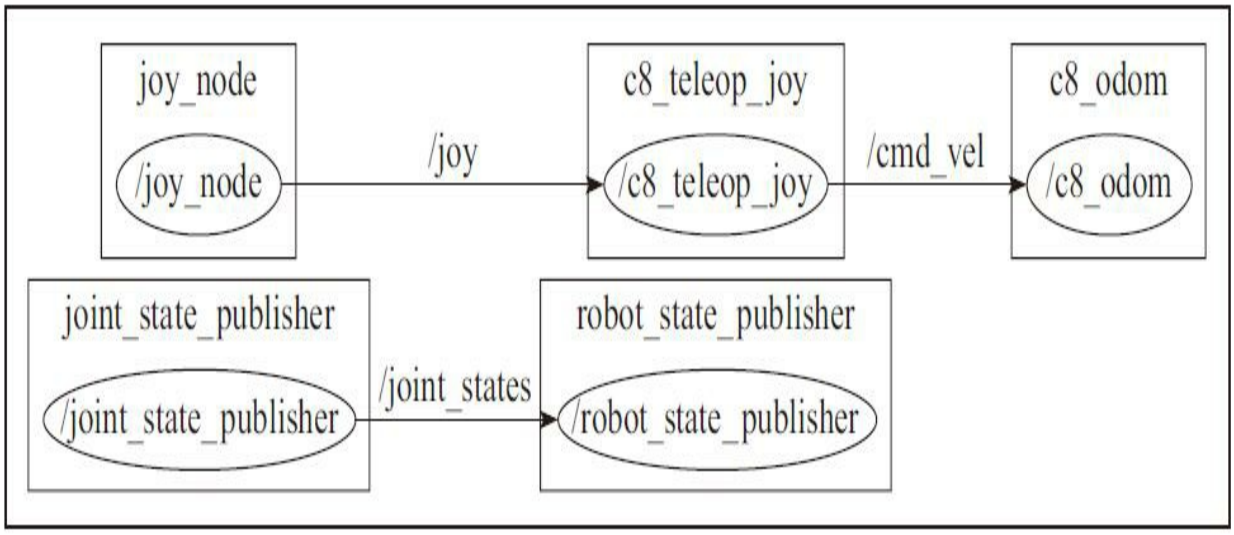
注意，在launch文件中，有4个不同的节点：c8_teleop_joy、joy_node、c8_odom、joint_state_publisher和rviz。在launch文件中注意到机器人模型名为robot2.urdf，在chapter8_tutorials/urdf中可找到它。

使用下面的命令启动该示例：

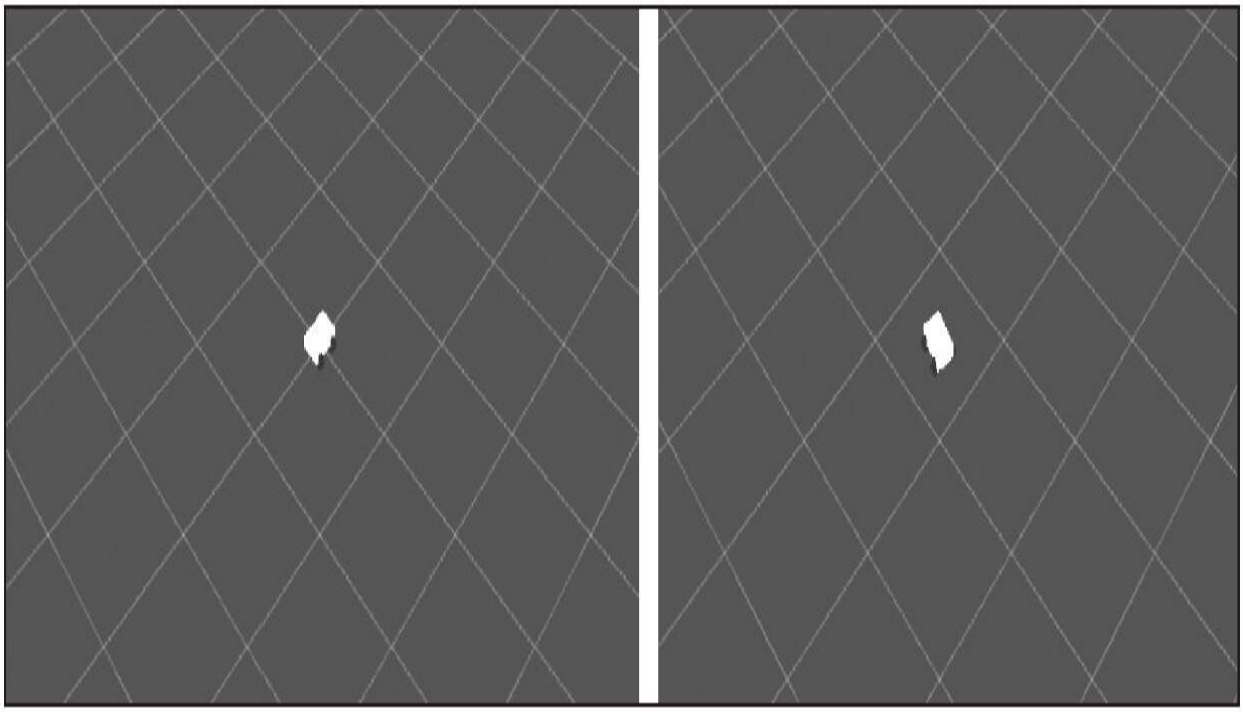
```
$ roslaunch chapter8_tutorials chapter8_robot_model.launch
```

可以通过使用rosnode列表和rostopic列表检查运行的节点和主题列

表是否正常。如果想以图形方式查看，使用rqt_graph。



如果一切成功，应该在RViz可视化界面中看到一个机器人模型，可以使用游戏杆控制它。

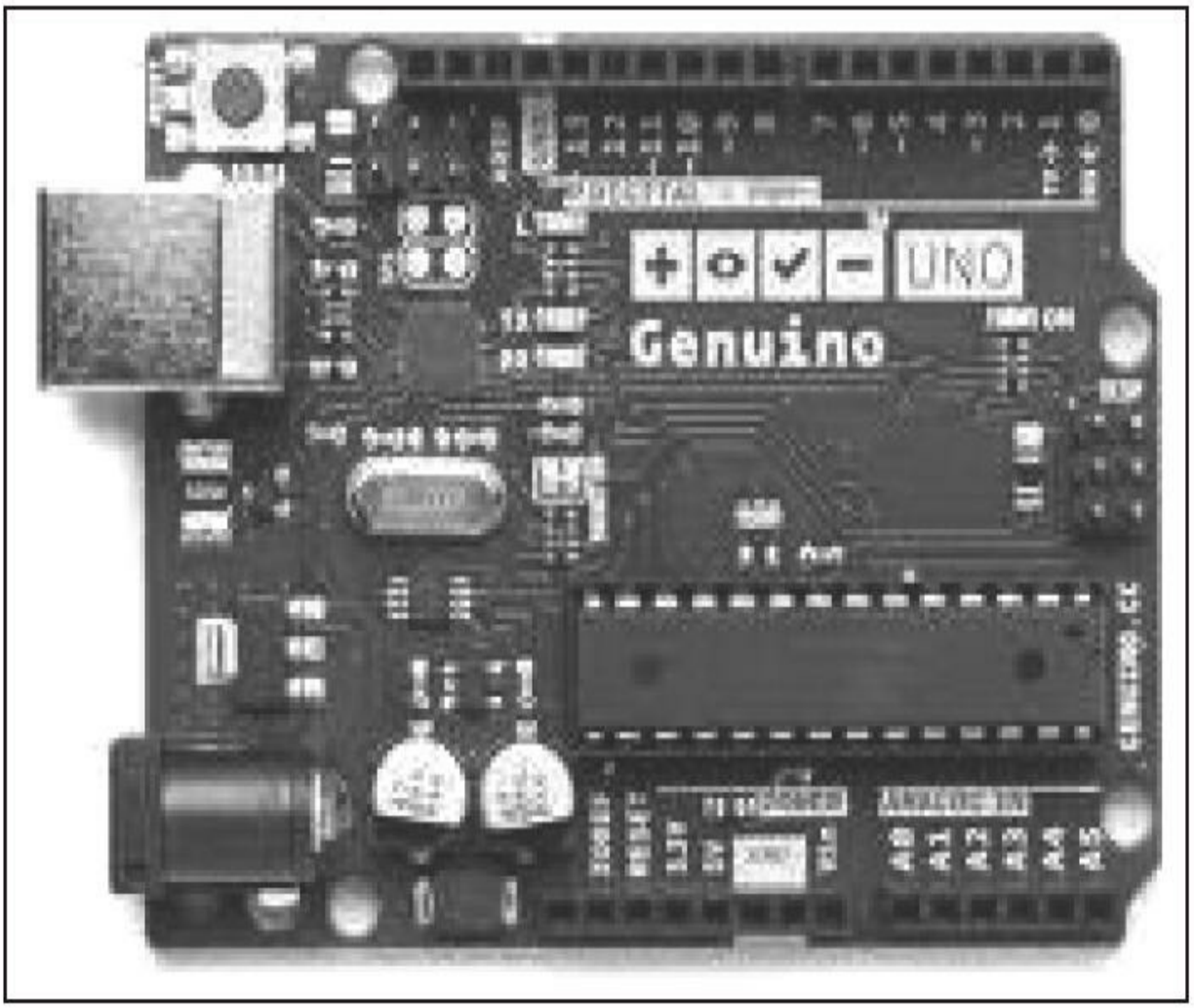


在下一节中，我们将学习如何在ROS中使用Arduino。然后，将把游戏杆示例节点连接到Arduino节点来控制真正的机器人电动机。

8.2 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。它适用于艺术家、设计人员、爱好者和所有喜爱创造交互式物体和环境的人。

下图是一个Arduino开发板。



ROS能通过rosserial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。可以通过rosserial使用大量支持串口连接的设备，例如GPS、伺服控制器等。

首先，输入下面的命令，安装功能包：

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

然后，对于catkin工作空间，复制rosserial软件库到其中。用下面的代码创建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
```

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ROS Kinetic，Arduino核心都在rosserial_arduino功能包中。可以在www.arduino.cc下载最新版本的Arduino IDE。

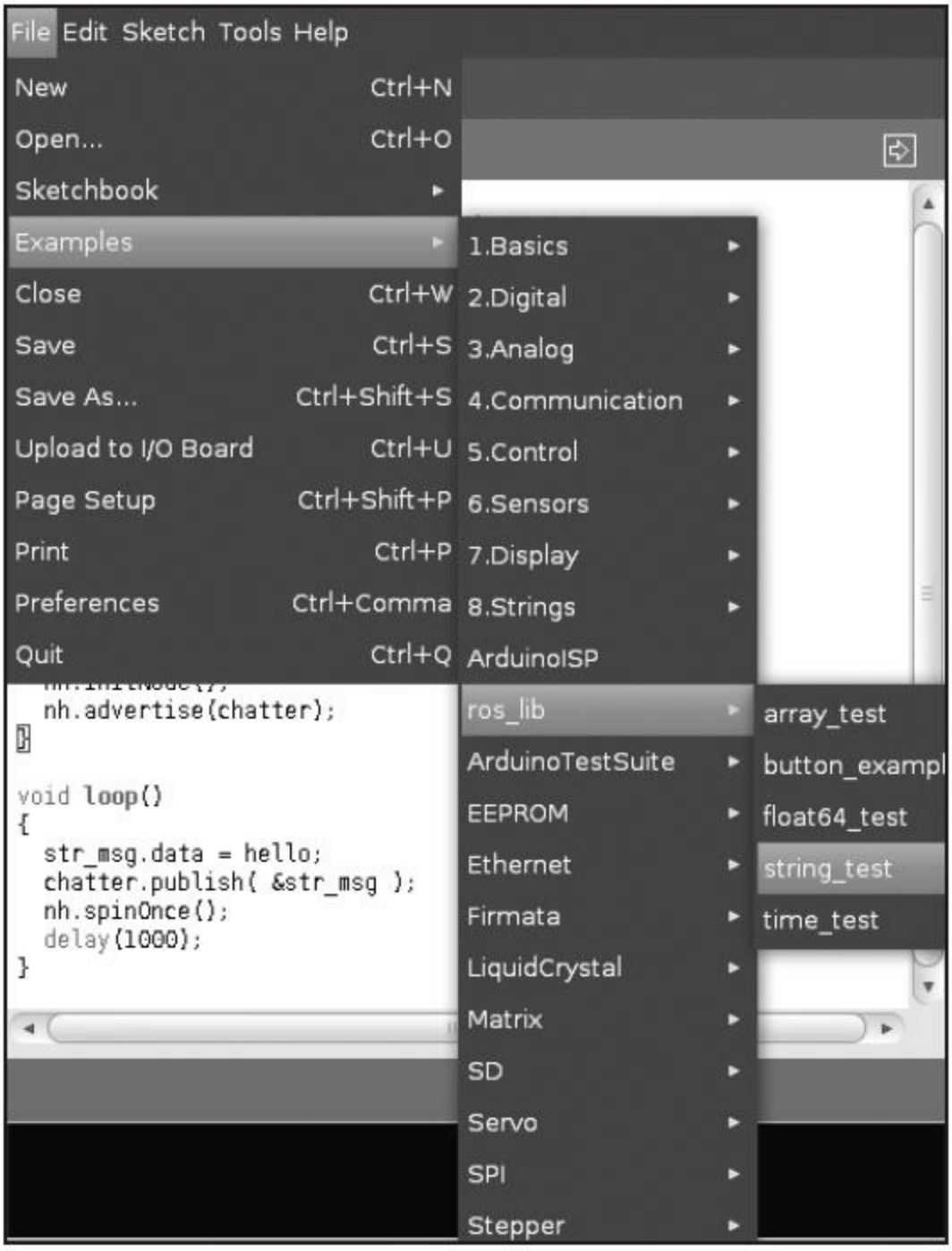
注意新版本的Arduino IDE，在home文件夹下的sketchbook文件夹已经更名为Arduino。

一旦你将所有功能包和IDE都安装好，就能够从rosserial功能包复制ros_lib到sketchbook/libraries文件夹下。这个文件夹会在计算机运行Arduino IDE之后创建。然后，需要运行make_libraries.py：

```
$ cd ~/Arduino/libraries
$ sudo rm -r roslib
$ rosrun rosserial_arduino make_libraries.py .
```

8.2.1 创建使用Arduino的示例程序

现在我们将从IDE上传示例代码到Arduino中。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rosserial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类型的信息。

下面的代码都保存在c8_arduino_string.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] = "chapter8_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，通常用于配置开发板，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。

```
$ rosrun rosserial_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的数据：

```
data: chapter8_tutorials
```

最后的示例是从Arduino向计算机发送数据。现在我们将会使用一个示例，其中Arduino会订阅一个主题并改变连接到13号引脚上的LED状态。示例的名称是blink，可以从Arduino IDE的菜单栏上直接查找File|Examples|ros_lib|Blink。

下面的代码段已经保存在c8_arduino_led.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：

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

现在，如果你想改变LED的状态，可以使用rostopic pub命令发布新的状态：

```
$ 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已经发光，那么它会灭掉。为了再次改变状态，需要再次发布主题：

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


publishing and latching message for 3.0 seconds
```

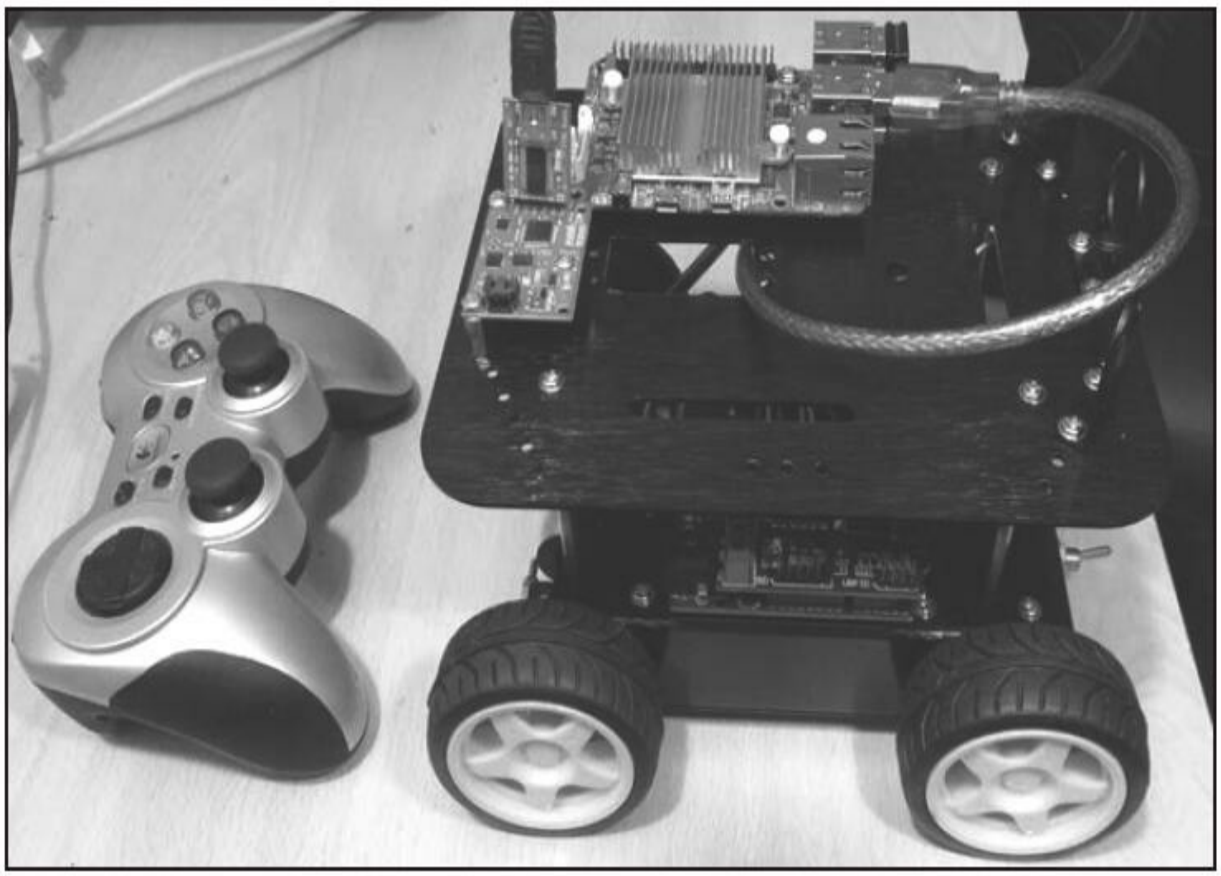
现在你能够在ROS中使用Arduino的所有设备了。这非常有用，因为你可以使用各种廉价的传感器和执行器装备你的机器人了。

 注意：当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，最好慎重选择你所用的设备。

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

8.2.2 由ROS和Arduino控制的机器人平台

现在，我们已经了解了如何使用Arduino与ROS。在本节中，将连接第一批执行器到ROS。Arduino用户有很多低成本的套件，可以选择不同套件制作由ROS控制的机器人。在本节中，将使用4×4机器人机箱套件。

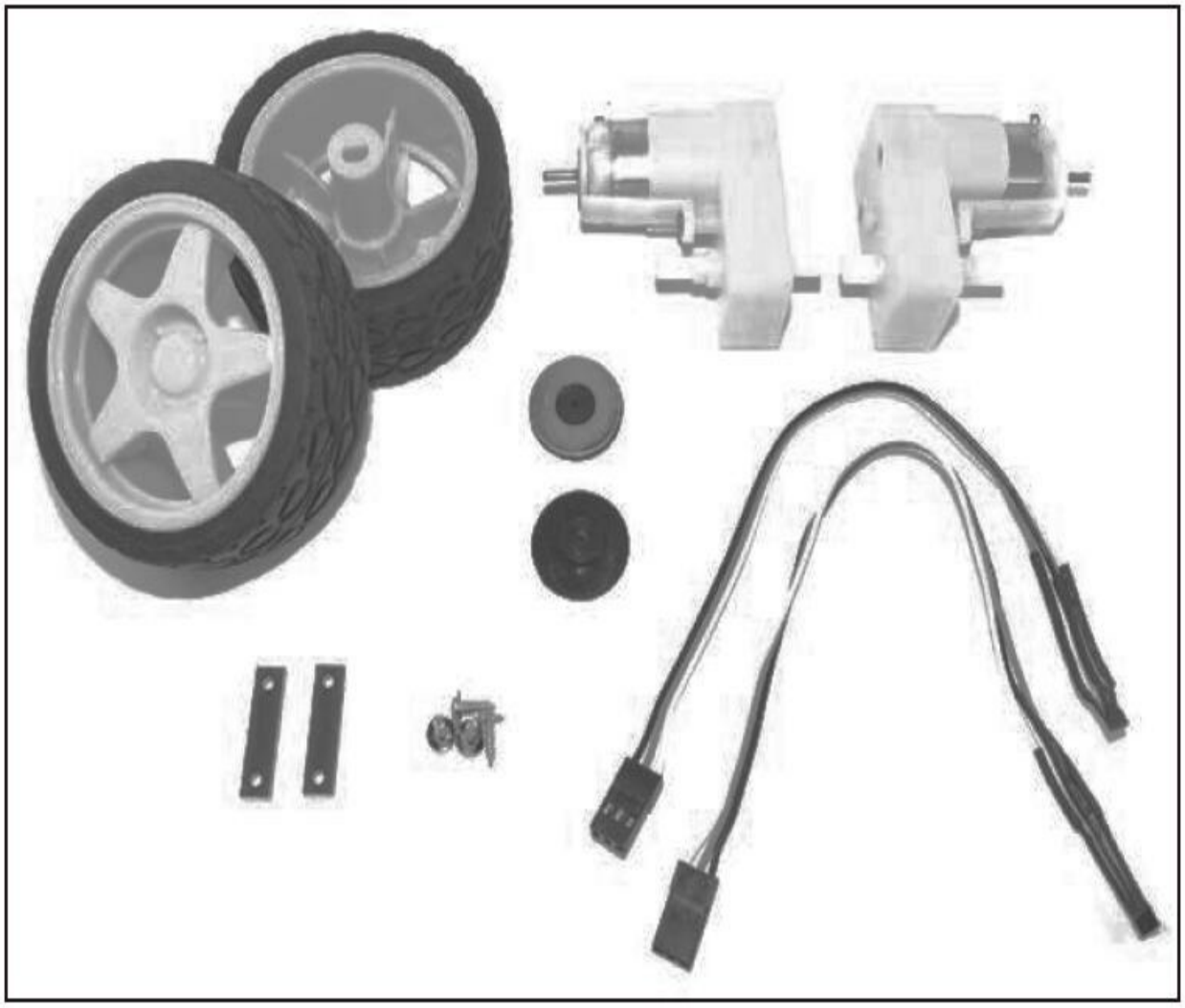


4×4机器人平台与Odroid C1、Arduino、9自由度Razor IMU和游戏杆

为了在机器人中使用ROS，不仅需要Arduino，还需要安装ROS的嵌入式计算机。有很多与ROS兼容的ARM开发板，如Dragonboard、Raspberry Pi或BeagleBone Black。这里，用Odroid C1并安装Ubuntu Xenial和ROS Kinetic。在本章中，我们将学习如何将不同的传感器连接到该平台来计算里程计，如惯性测量单元（Inertial Measurement Unit，IMU）或某些车轮编码器。

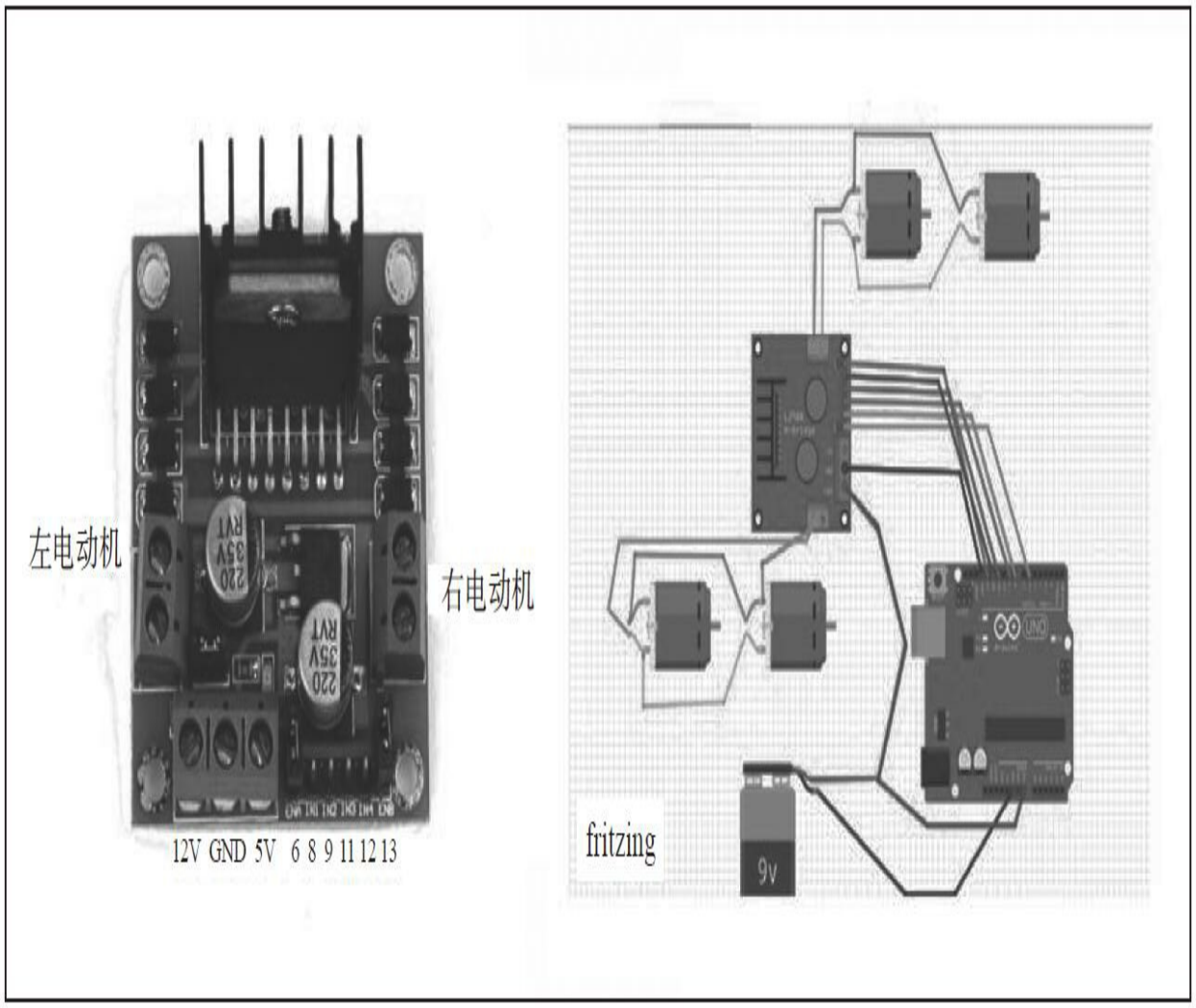
机器人平台有4个被控电动机。为此，将使用带有电动机控制器的

Arduino。有很多电动机控制器，它们具有和Arduino的接口。这里将使用非常通用的电路板，它基于L298N双电动机驱动器。下图显示了具有两个车轮、两个电动机和两个磁性编码器的一个套件。



1.使用Arduino将机器人电动机连接到ROS

由于我们有四个电动机，因此将平台同一侧电动机连接到相同的输出。将数字信号连接到L298N板上控制电动机的行为。



每个通道需要三个要控制的信号。IN1和IN2是用于设定电机旋转方向的数字信号。ENA用于使用Arduino脉宽调制（Pulse Width Modulation，PWM）信号控制电机功率。该信号将控制左电动机。对于右电动机，IN3、IN4和ENB是控制信号。在前面提到的图中，有一个连线图。注意，将Arduino的线路连接到L298N，并编程INx信号。将电动机连接到L298N电动机控制器，并将控制信号连接到Arduino。

可以启动一个程序来控制电动机，在chapter8_tutorials/src/robot_motors.ino中有全部代码。我们将使用Arduino IDE来实现这个功能。首先，声明依赖关系并定义连接到L298N电动机控制器的Arduino引脚：

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

#define ENA 6
#define ENB 11
#define IN1 8
#define IN2 9
#define IN3 12
#define IN4 13
```

然后，将声明两个回调函数来激活每个电动机。当接收到用于左轮的命令消息时，根据命令的符号来设定电动机的旋转方向。数字信号IN1和IN2设置电动机向前或向后旋转。ENA是控制电动机电压的PWM信号；该信号能够调节电动机的速度。右轮命令的回调类似于左轮：

```
void cmdLeftWheelCB( const std_msgs::Int16& msg)
```

```
{
  if(msg.data >= 0)
  {
    analogWrite(ENA,msg.data);
    digitalWrite(IN1, LOW);
    digitalWrite(IN2, HIGH);
  }
  else
  {
    analogWrite(ENA,-msg.data);
    digitalWrite(IN1, HIGH);
    digitalWrite(IN2, LOW);
  }
}

void cmdRightWheelCB( const std_msgs::Int16& msg)
{
  if(msg.data >= 0)
    {
    analogWrite(ENB,msg.data);
    digitalWrite(IN3, LOW);
    digitalWrite(IN4, HIGH);
  }
  else
  {
    analogWrite(ENB,-msg.data);
    digitalWrite(IN3, HIGH);
    digitalWrite(IN4, LOW);
  }
}
```

然后，声明两个订阅，该主题命名为cmd_left_wheel，使用std::msgs::Int16类型，以及之前的回调函数：

```
ros::Subscriber<std_msgs::Int16> subCmdLeft("cmd_left_wheel",
cmdLeftWheelCB );
ros::Subscriber<std_msgs::Int16>
subCmdRight("cmd_right_wheel",cmdRightWheelCB );
```

在Arduino的setup()函数中，对应的引脚设置为输出，初始化nd节点并开始订阅该主题：

```
void setup() {
  // put your setup code here, to run once:
  pinMode(ENA, OUTPUT);
  pinMode(ENB, OUTPUT);
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
  analogWrite(ENA,0);
```

```
  analogWrite(ENB, 0);
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, HIGH);

  nh.initNode();
  nh.subscribe(subCmdRight);
  nh.subscribe(subCmdLeft);
}
```

loop()函数如下：

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

现在将代码上传到开发板，并在终端运行Arduino节点：

```
$ roscore
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

在其他终端，可以手动发布一些命令到左电动机和右电动机：

```
$ rostopic pub /cmd_right_wheel std_msgs/Int16 "data: 190"

$ rostopic pub /cmd_left_wheel std_msgs/Int16 "data: -100"
```

该命令信号范围在-255和255之间，当使用零值时，电动机停止。

既然将机器人电动机连接到ROS，就可以用游戏杆控制它们。可以统一速度的主题，采用差分驱动运动学方程和geometry_msgs::Twist主题。这样就可以将标准消息发送到机器人的里程计节点，该节点与我们使用的机器人类型无关。例如使用c8_teleop_joy节点发送该类型的主题，就可以用游戏杆来控制车轮。

新建一个名为robot_motors_with_twist.ino的sketch文件。在复制上面的代码前，在该文件开始添加一个新的头文件以便于使用geometry_msgs::Twist：

```
#include <geometry_msgs/Twist.h>
float L = 0.1; //distance between wheels
```

然后，包括一个新的订阅者及其回调函数：

```
void cmdVelCB( const geometry_msgs::Twist& twist)
{
  int gain = 4000;
  float left_wheel_data = gain*(twist.linear.x -
  twist.angular.z*L);
  float right_wheel_data = gain*(twist.linear.x +
  twist.angular.z*L);
  {
    analogWrite(ENA,abs(left_wheel_data));
    digitalWrite(IN1, LOW);
```

```
    digitalWrite(IN2, HIGH);
  }
  else
  {
    analogWrite(ENA,abs(left_wheel_data));
    digitalWrite(IN1, HIGH);
    digitalWrite(IN2, LOW);
  }
  if(right_wheel_data >= 0)
  {
    analogWrite(ENB,abs(left_wheel_data));
    digitalWrite(IN3, LOW);
    digitalWrite(IN4, HIGH);
  }
  else
  {
    analogWrite(ENB,abs(left_wheel_data));
    digitalWrite(IN3, HIGH);
    digitalWrite(IN4, LOW);
  }
}
ros::Subscriber<geometry_msgs::Twist> subCmdVel("cmd_vel",
cmdVelCB);
```

现在，将代码上传到Arduino开发板。为了更容易地运行所有节点，我们将创建一个Launch文件管理游戏杆、RViz可视化程序中的机器

人模型和Arduino节点。

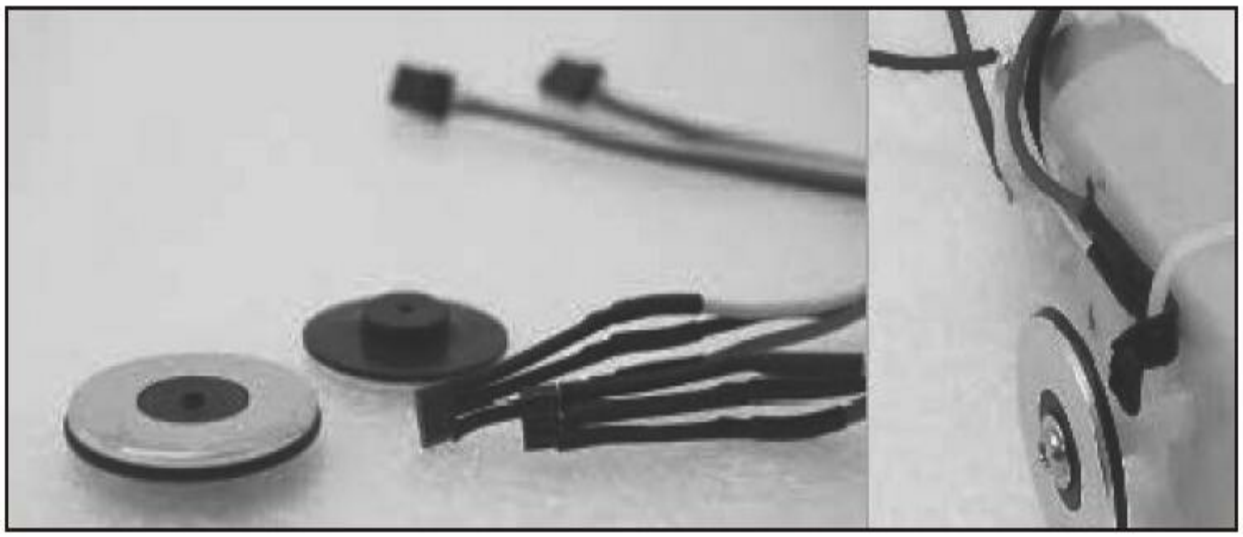
可以在终端中使用以下命令启动示例：

```
$ roslaunch chapter8_tutorials chapter8_tutorials robot_model_with_
motors.launch

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

现在，可以用游戏杆控制机器人，还可以在RViz可视化程序中看到机器人模型的移动。如果在一个没有显示器的嵌入式ARM电脑上测试代码，可以注释掉Launch文件中的rviz节点。

2.连接编码器到机器人

Dagu编码器的每个轮盘由两部分组成：一个具有8极（4个北极和4个南极）的磁盘以及一个霍尔效应传感器。磁盘安装在轮轴上，霍尔效应必须以最大不超过3毫米的距离面对轮盘。



Dagu编码器：传感器和磁盘，安装示例

将编码器安装在车轮上后，就可将其连接到Arduino。霍尔效应传感器直接焊接到三根电线：黑线、红线和白线。

黑线是传感器的地线，必须连接到Arduino地，红色为电源线，连接+5V电压。白线是信号线，信号是二进制的，当磁盘转动并且不同的磁极面对传感器时，其值从高电平变为低电平。

因此，将信号线连接到Arduino数字引脚是合乎逻辑的。此外，根据电动机的转数以及电动机高速旋转编码器的极数/步数，编码器的信号可能也会变化非常快。建议将编码器信号连接到Arduino中断引脚，以便更快地读取信号变化。

对于Arduino/Genuino Uno，中断引脚是引脚2和引脚3。将编码器连接到该引脚。为了检查编码器的工作原理并学习如何使用Arduino中断，将使用以下代码：

```
#include <std_msgs/Float32.h>
#include <TimerOne.h>
#define LOOP_TIME 200000
#define left_encoder_pin 3
#define right_encoder_pin 2
```

我们声明新变量来统计编码器的每个脉冲并创建主题以发布轮速：

```
unsigned int counter_left=0;
unsigned int counter_right = 0;

ros::NodeHandle nh;
std_msgs::Float32 left_wheel_vel;
ros::Publisher left_wheel_vel_pub("/left_wheel_velocity",
&left_wheel_vel);

std_msgs::Float32 right_wheel_vel;
ros::Publisher right_wheel_vel_pub("/right_wheel_velocity",
&right_wheel_vel);
```

必须创建两个函数来处理编码器的中断。每当在右侧或左侧编码器引脚检测到变化时，都会增加计数：

```
void docount_left() // counts from the speed sensor
{
  counter_left++; // increase +1 the counter value
}

void docount_right() // counts from the speed sensor
{
  counter_right++; // increase +1 the counter value
}
```

定时器用于发布每个车轮的速度。使用编码计数器、车轮半径和定时器持续时间来计算speed:void timerIsr()：

```
{
  Timer1.detachInterrupt(); //stop the timer

  //Left Motor Speed
  left_wheel_vel.data = counter_left;
  left_wheel_vel_pub.publish(&left_wheel_vel);
  right_wheel_vel.data = counter_right;
  right_wheel_vel_pub.publish(&right_wheel_vel);
  counter_right=0;
  counter_left=0;
  Timer1.attachInterrupt( timerIsr ); //enable the timer
}
```

最后，在setup函数中，把编码器引脚声明为INPUT_PULLUP。这意味着Arduino将把它们视为输入，将在内部将这些引脚连接一个上拉电阻：

```
//Setup for encoders
pinMode(right_encoder_pin, INPUT_PULLUP);
pinMode(left_encoder_pin, INPUT_PULLUP);
Timer1.initialize(LOOP_TIME);
attachInterrupt(digitalPinToInterrupt(left_encoder_pin),
docount_left, CHANGE); // increase counter when speed sensor pin
goes High
attachInterrupt(digitalPinToInterrupt(right_encoder_pin),
docount_right, CHANGE); // increase counter when speed sensor pin
goes High
```

此外，发布的消息将会在setup中宣传：

```
nh.advertise(left_wheel_vel_pub);
nh.advertise(right_wheel_vel_pub);
Timer1.attachInterrupt( timerIsr ); // enable the timer
```

现在可以将代码上传到开发板中，并运行Arduino节点：

```
$ roscore
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

在另一个终端中输入下面的命令：

```
$ rostopic list
```

将看到如下：

现在，可以使主题回显并转动车轮，或者可以运行rqt_graph。此外，还可以使用rostopic hz查看各消息发布的频率：

```
/cmd_left_wheel
/cmd_right_wheel
/diagnostics
/left_wheel_velocity
/right_wheel_velocity
/rosout
/rosout_agg
```

```
subscribed to [/left_wheel_velocity]
average rate: 5.008
        min: 0.198s max: 0.201s std dev: 0.00127s window: 5
average rate: 5.004
        min: 0.197s max: 0.201s std dev: 0.00146s window: 10
^Caverage rate: 5.006
        min: 0.197s max: 0.201s std dev: 0.00156s window: 14
luis@daneel:~$ rostopic hz /right_wheel_velocity
subscribed to [/right_wheel_velocity]
average rate: 5.007
        min: 0.198s max: 0.201s std dev: 0.00095s window: 5
average rate: 5.011
        min: 0.198s max: 0.201s std dev: 0.00083s window: 10
average rate: 5.009
        min: 0.198s max: 0.201s std dev: 0.00078s window: 15
^Caverage rate: 5.007
        min: 0.198s max: 0.201s std dev: 0.00081s window: 16
```

与之前操作/cmd_vel类似，现在可以在geometry_msgs::Twist中统一这两个主题。要创建一个新的geometry_msgs::Twist消息以发送平台的速度和该消息的主题发布者：

```
geometry_msgs::Twist sensor_vel;
ros::Publisher sensor_vel_pub("/sensor_velocity", &sensor_vel);
```

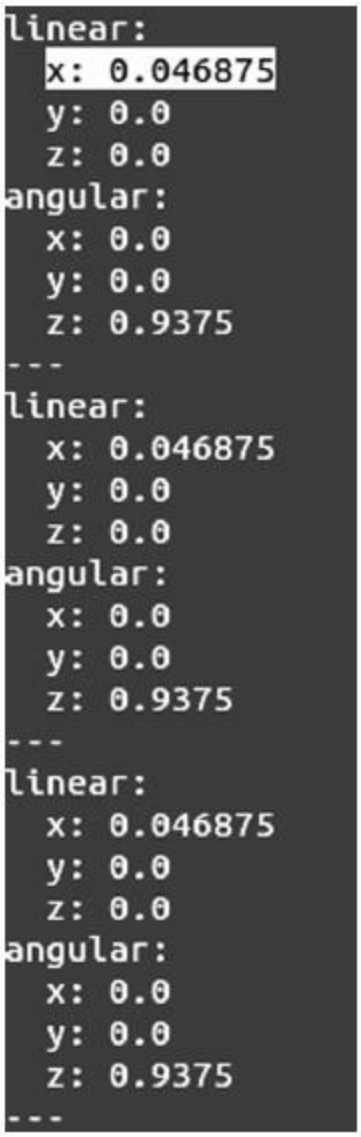
在timerISR()函数中，使用运动学方程计算线速度和角速度：

```
sensor_vel.linear.x = radius*(left_wheel_vel.data +
right_wheel_vel.data)/2;
sensor_vel.linear.y = 0;
sensor_vel.linear.z = 0;
sensor_vel.angular.x = 0;
sensor_vel.angular.y = 0;
sensor_vel.angular.z = radius*(left_wheel_vel.data +
right_wheel_vel.data)/L;
sensor_vel_pub.publish(&sensor_vel);
```

然后，可以通过修改里程计节点来订阅机器人编码器传感器在/sensor_vel主题中发布的实际里程计数据：

可以使用下面的命令查看示例：

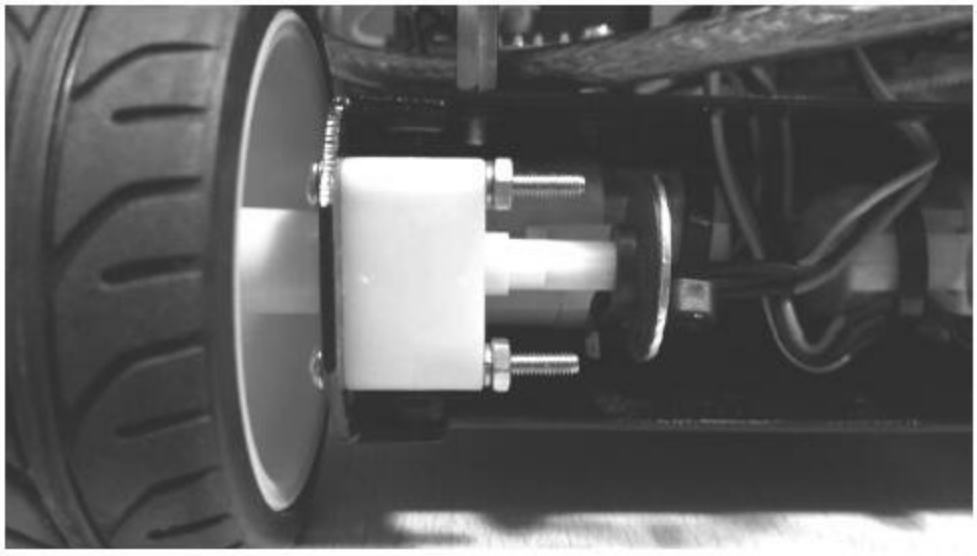
```
$ roslaunch chapter8_tutorials chapter8_robot_encoders.launch
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```



3.控制车轮速度

现在，可以使用游戏杆、手动发布/cmd_vel主题或使用导航算法控制机器人的车轮。此外，通过编码器数据可以推算小车的里程计并估计其位置。但是如何确保/sensor_velocity数据接近/cmd_vel命令呢？

如果查看/cmd_vel命令和/sensor_velocity的数据，会发现它们是不相等的。可以在/cmd_vel命令中尝试设置不同的增益以根据真实速度调整速度命令，但这不是最佳的方式。该解决方案需要考虑电源、小车的重量或机器人行驶的地板表面。



平台上加载的车轮、电动机和编码器

现在是进一步深入探索控制算法的时候了。当我们设计一个速度可控的机器人时，通过/cmd_vel和编码器传感器的速度反馈，就可以在机器人平台上实现闭环控制算法。例如，这里应用PID算法。因此，尝试在Arduino平台上实现自己的控制算法。如果你完成这里的操作，将一定能够实现它。

在代码库中，你会发现一个PID闭环控制算法的例子，如果不愿亲自动手开发控制算法不妨尝试运行该示例。

8.3 使用9自由度低成本IMU

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

在本节中，我们将学习如何使用低成本的传感器——9自由度（Degree of Freedom，DOF）惯性测量模块。该传感器具有加速度计（x3）、磁力计（x3）、气压计（x1）和陀螺仪（x3）。9DoF Razor IMU和9DoF传感器模块是可用于机器人项目的低成本IMU。这两个板子有一个HMCL5883磁力计、一个ADXL345和ITG3200陀螺仪。旧版本有不同的磁力计。两块板之间的主要区别是，该Razor IMU包含一个微控制器ATMega328，因此俯仰、滚动和偏航从原始数据计算得到，并通过TTL发送。为了连接Razor IMU，需要一个3.3V TTL到USB的转换器。传感器模块只有三个传感器，具有I2C通信。

注意，9DoF Razor IMU的微控制器是ATMega328，这与Arduino UNO使用的控制器相同，因此可以使用Arduino IDE轻松地更新固件或开发自己的代码。也可以通过自己的Arduino来使用I2C控制传感器模块。

本节介绍的传感器成本约为70美元，由于价格低廉被很多项目采用。现在，Sparkfun将其升级为一个新的9DOF IMU Razor M0，价格在50美元左右，链接为https://www.sparkfun.com/products/14001。

你可以在下图中看到9DoF Razor IMU传感器。它很小，主要有以下部分组成。

9DoF Razor IMU

这块板子具有以下传感器。

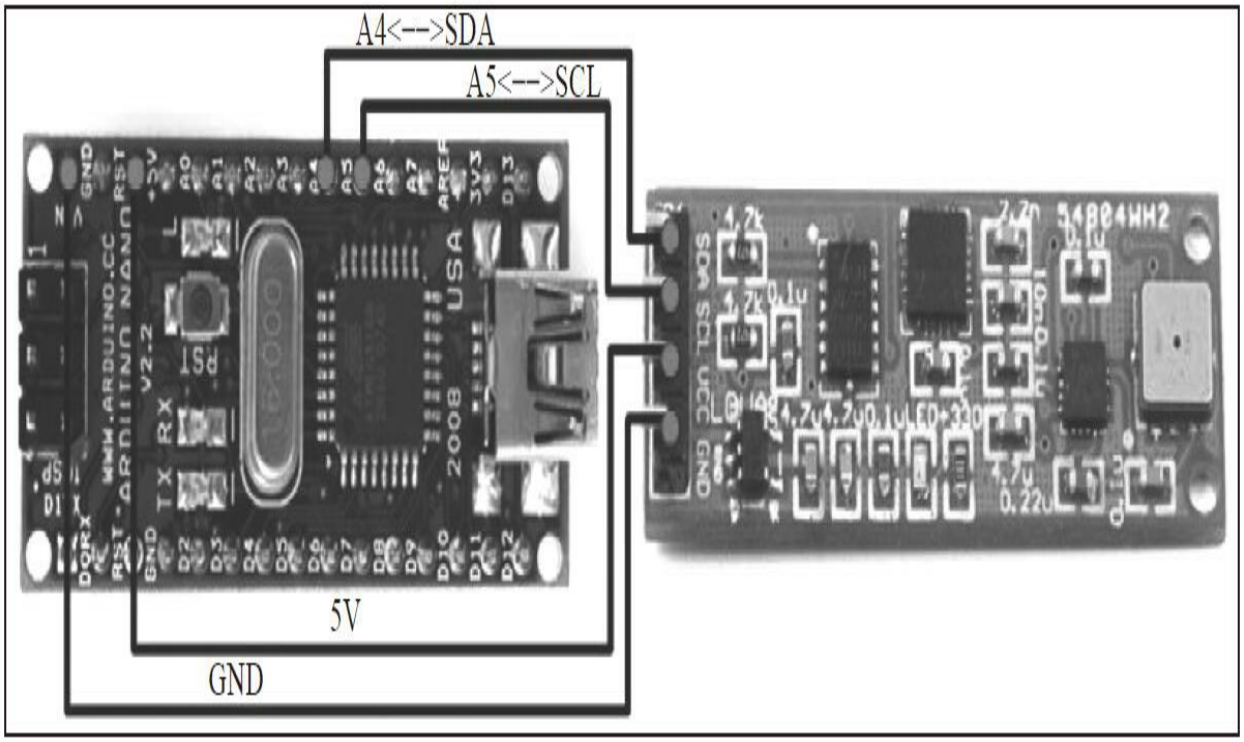
·ADXL345：这是一个高精度（13位）三轴加速度计，最高测量范围为±16g。这个传感器广泛地用于移动设备终端。它能够测量静态的重力加速度用于倾角测量，同时也能够测量由于运动和冲击造成的动态加速度。

·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：这是用于先进移动应用的高精度大气压力传感器。它性能强大，具有最高0.03hPa（百帕）的绝对精度和3μA的超低功耗。

·L3G4200D：这是高精度（16位）三轴陀螺仪，测量范围高达2000度每秒（d/s）。这个陀螺仪能够同时对三个轴的转动量进行测量。

如前所述，这块板子可通过I2C协议进行控制。我们能够使用Arduino来控制它。在下图中，能看到Arduino和IMU两块板子的连接方式：



Anduino Nano和传感器模块

对于Arduino UNO，只需要按图中Arduino Nano一样连接相同引脚即可。唯一需要做的事情是连接4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（Serial Data Line，SDL）需要连接到模拟量4号引脚，串口时钟（Serial Clock，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。

8.3.1 安装Razor IMU ROS库

在安装ROS的Razor IMU库之前，需要使用下面的命令安装可视化的python：

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

然后，切换到工作空间的/src文件夹下，并且从GitHub的Kristof Robot复制razor_imu_9dof库到工作空间的/src文件夹，编译它，代码如下：

```
$ cd ~/dev/catkin_ws/src
$ git clone https://github.com/KristofRobot/razor_imu_9dof.git
$ cd ..
$ catkin_make
```

需要安装Arduino IDE，这在8.2解释过。打开Arduino IDE的/dev/catkin_ws/src/Razor_AHRS/Razor_AHRS.ino。现在，选择硬件选项。在代码中搜索本部分，并取消正确代码的注释：

```
/*****************************************************************/
/********** USER SETUP AREA! Set your options here! *************/
/*****************************************************************/
```

```
// HARDWARE OPTIONS
/*****************************************************************/
// Select your hardware here by uncommenting one line!
//#define HW__VERSION_CODE 10125 // SparkFun "9DOF Razor IMU"
version "SEN-10125" (HMC5843 magnetometer)
#define HW__VERSION_CODE 10736 // SparkFun "9DOF Razor IMU"
version "SEN-10736" (HMC5883L magnetometer)
//#define HW__VERSION_CODE 10183 // SparkFun "9DOF Sensor Stick"
version "SEN-10183" (HMC5843 magnetometer)
//#define HW__VERSION_CODE 10321 // SparkFun "9DOF Sensor Stick"
version "SEN-10321" (HMC5843 magnetometer)
//#define HW__VERSION_CODE 10724 // SparkFun "9DOF Sensor Stick"
version "SEN-10724" (HMC5883L magnetometer)
```

这里使用的是9DoF Razor IMU版本SEN-10736。现在，可以保存并上传代码。在Arduino IDE中慎重选择正确的处理器。对于我的Razor IMU，选择Arduino Pro或Pro Mini，并且选择3.3V和8MHz的ATMega作为处理器。

在razor_imu功能包中，需先准备名为my_razor.yaml的配置文件。可以复制my_razor.yaml的默认配置：

```
$ roscd razor_imu_9dof/config
$ cp razor.yaml my_razor.yaml
```

可以打开文件并检查其配置是否与你的配置相匹配。现在最重要的是正确设置端口，弄清楚你是否为正在使用传感器模块和Arduino。我这里是默认的：

```
port: /dev/ttyUSB0
```

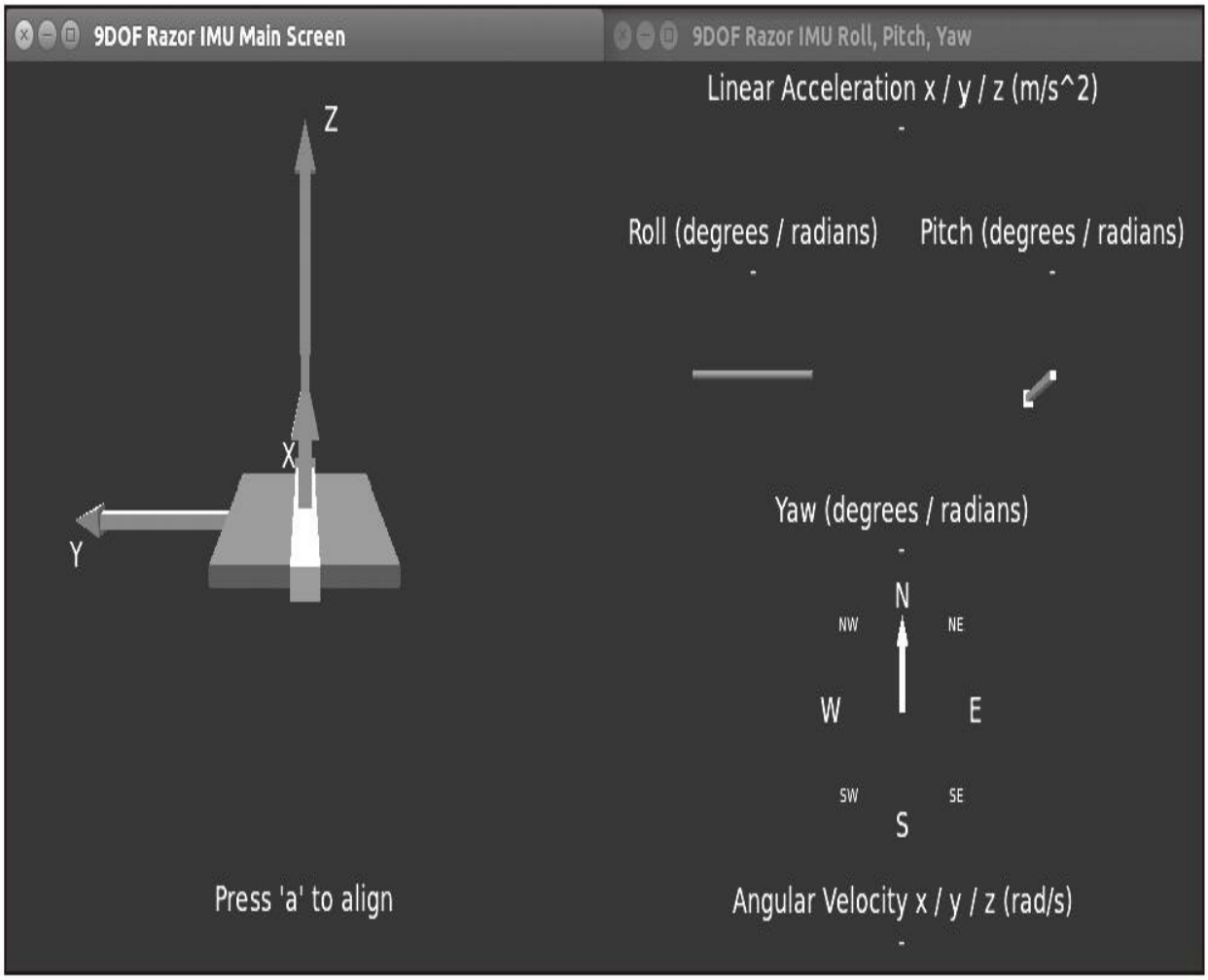
请注意，如果你校准过IMU，则可以使用此文件中的所有校准参数

获取正确的方向测量。

现在通过下面的启动命令使用razor_imu或模块：

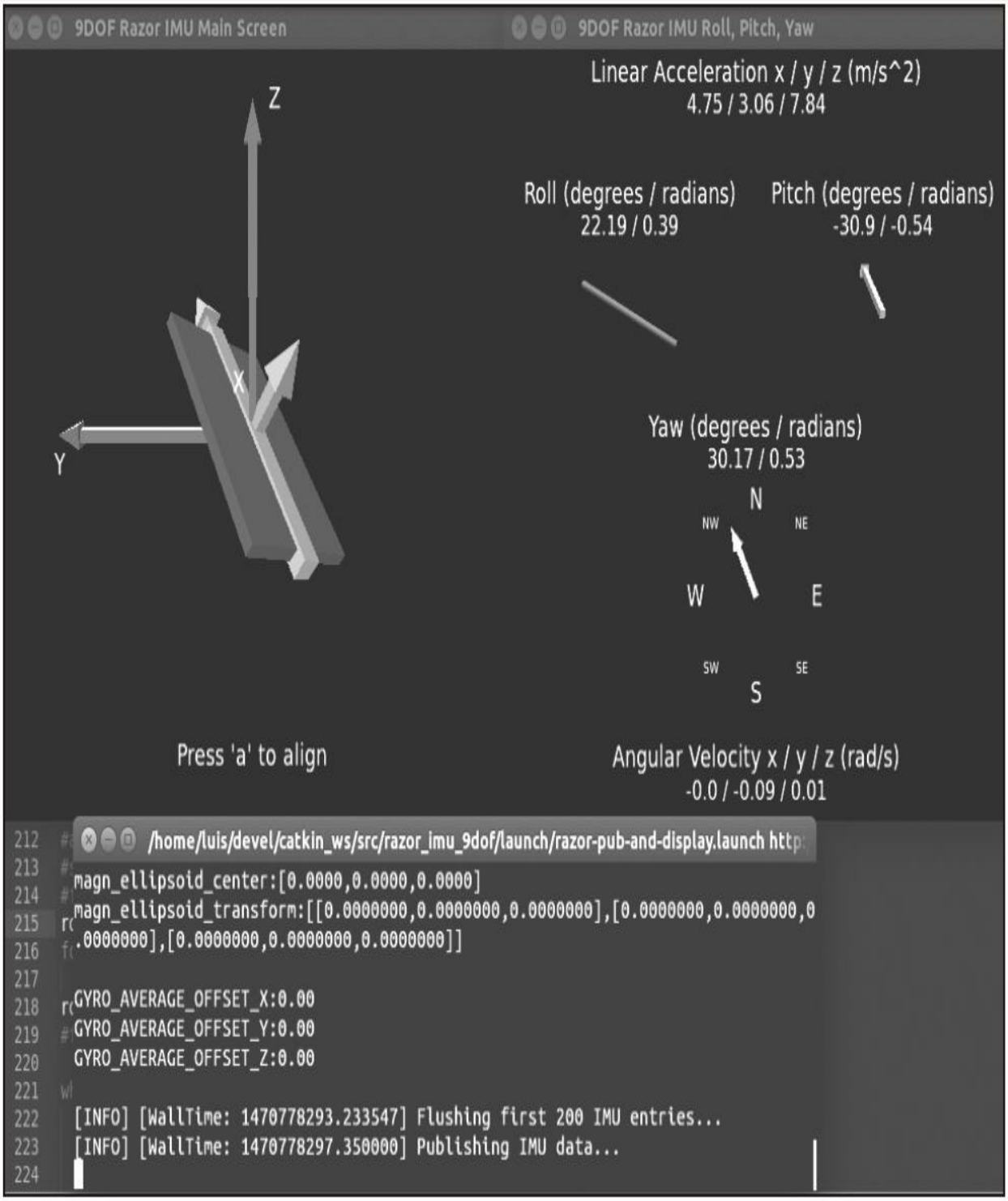
```
$ roslaunch razor_imu_9dof razor-pub-and-display.launch
```

必将出现两个窗口，一个3D图用来表示IMU的姿态，一个2D图形包括倾斜、俯仰和偏航。



Razor IMU显示器

几秒钟后，传感器开始传输msgs::IMU消息，如果你移动传感器，会看到参数也会随之发生变化。



具有不同方位的Razor IMU

8.3.2 Razor如何在ROS中发送数据

如果一切正常，则可以使用rostopic命令来查看主题列表：

```
$ rostopic list
```

节点将发布三个主题。我们将在本节使用/imu/data。首先，将看到该主题发送的类型和数据。使用下面的命令查看类型和字段：

```
$ rostopic type /imu

$ rosmsg show sensor_msgs/Imu
```

/IMU的主题是sensor_msg/IMU。这些字段用于指示方向、加速度和速度。在本例中，将使用orientation（方向）字段。可以使用下面的命令查看实际发送数据的消息：

```
$ rostopic echo /imu
```

你将看到类似于下图的内容：

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
float64[9] orientation_covariance
geometry_msgs/Vector3 angular_velocity
  float64 x
  float64 y
  float64 z
float64[9] angular_velocity_covariance
geometry_msgs/Vector3 linear_acceleration
  float64 x
  float64 y
  float64 z
float64[9] linear_acceleration_covariance
```

```
header:
  seq: 43264
  stamp:
    secs: 1480621387
    nsecs: 926049947
  frame_id: base_imu_link
orientation:
  x: -0.664401936806
  y: 0.459286679427
  z: -0.562455021343
  w: 0.176833711254
orientation_covariance: [0.0025, 0.0, 0.0, 0.0, 0.0025, 0.0, 0.0, 0.0, 0.0025]
angular_velocity:
  x: -0.02
  y: -0.04
  z: -0.0
angular_velocity_covariance: [0.02, 0.0, 0.0, 0.0, 0.02, 0.0, 0.0, 0.0, 0.02]
linear_acceleration:
  x: 5.8836
  y: -7.60960921875
  z: -2.98087078125
linear_acceleration_covariance: [0.04, 0.0, 0.0, 0.0, 0.04, 0.0, 0.0, 0.0, 0.04]
---
```

如果观察orientation（方向）字段，将看到4个变量，而不是所期望的三个变量。这是因为在ROS中，空间方向用四元数表示。你可以在互联网上找到关于这个简洁而明确的方位表示的很多文章。

8.3.3 创建一个ROS节点以使用机器人中的9DoF传感器数据

现在，在ROS中我们已经使低成本IMU工作了，然后将创建一个订阅Razor IMU中imu/data主题的新节点。例如，可以使机器人模型根据IMU的数据呈现俯仰、横滚和方位变化。所以，创建一个基于c8_odom.cpp的新节点。将这个文件命名为c8_odom_with_imu.cpp。

现在要做一些修改。首先，将包括sensor_msgs/IMU头文件以便能够订阅IMU主题：

```
#include <sensor_msgs/Imu.h>
```

另外，需要一个全局变量以存储IMU数据：

```
sensor_msgs::Imu imu;
```

然后，创建一个回调函数以获取IMU数据：

```
void imuCallback(const sensor_msgs::Imu &imu_msg)
{
  imu = imu_msg;
}
```

在main函数中，需要声明/imu_data主题的一个订阅者：

```
ros::Subscriber imu_sub = n.subscribe("imu_data", 10,
imuCallback);
```

要分配正确的IMU方向数据给每个odom变换：

```
odom_trans.transform.rotation.x = imu.orientation.x;
odom_trans.transform.rotation.y = imu.orientation.y;
odom_trans.transform.rotation.z = imu.orientation.z;
odom_trans.transform.rotation.w = imu.orientation.w;
```

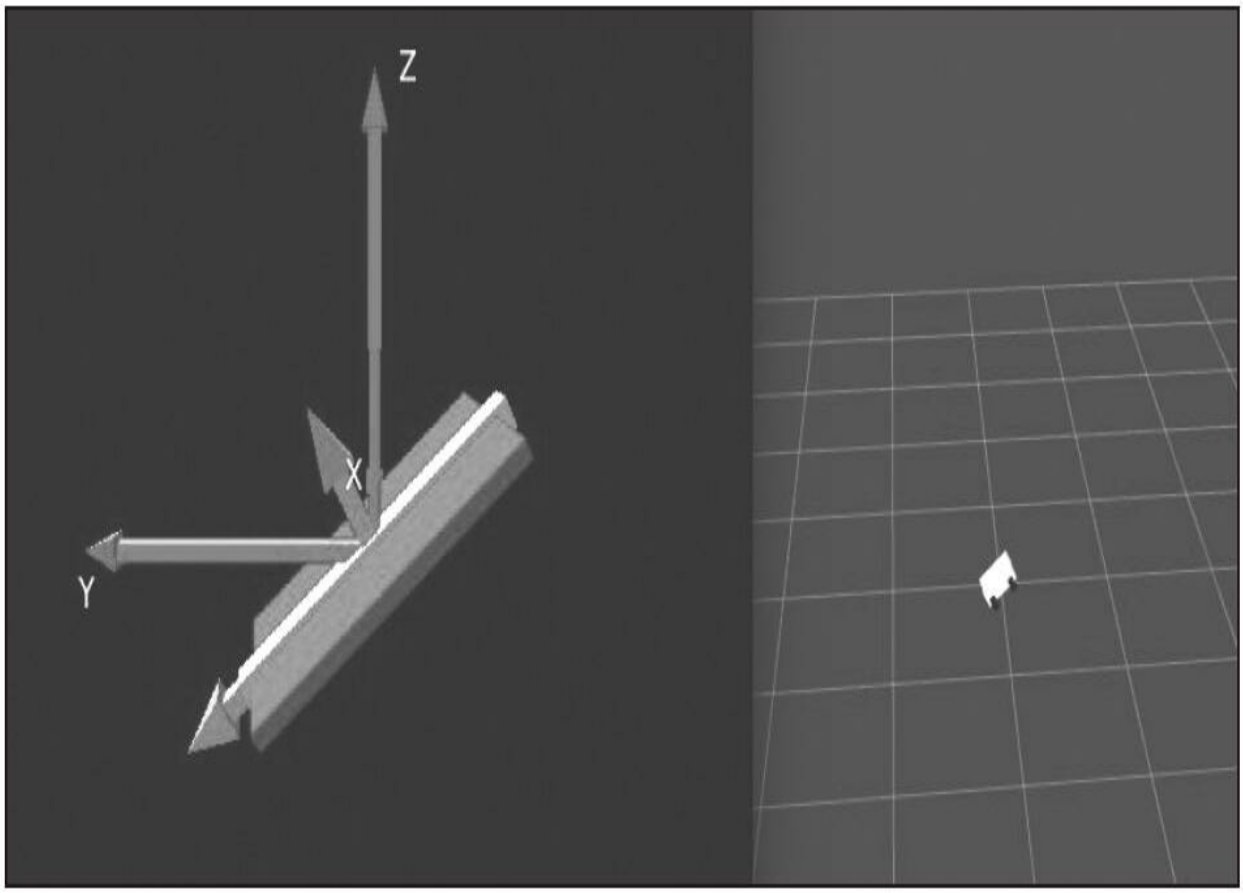
此外，必须将机器人的姿态分配给IMU获取的方向数据：

```
odom.pose.pose.orientation.x = imu.orientation.x;
odom.pose.pose.orientation.y = imu.orientation.y;
odom.pose.pose.orientation.z = imu.orientation.z;
odom.pose.pose.orientation.w = imu.orientation.w;
```

在创建这个文件并编译它后，就可以运行这个例子：

```
$ roslaunch razor_imu_9dof razor-pub-and-display.launch
$ roslaunch chapter8_tutorials chapter8_robot_imu.launch
```

你将在RViz可视化程序中看到Razor IMU显示器和机器人模型。机器人模型和Razor IMU的姿态和方向一致。



Razor IMU显示器和RViz中的机器人模型

8.3.4 使用机器人定位来融合传感器数据

这时，我们有机器人并有不同的传感器对机器人进行定位，如轮编码器和9DoF Razor IMU。我们将安装一个新的功能包以融合这些传感器中的数据，从而改进机器人位置估计。功能包robot_localization使用扩展卡尔曼滤波器（Extended Kalman Filter，EKF）通过来自多个传感器的数据计算新的估计值。

在终端中键入以下命令安装此功能包：

```
$ sudo apt-get install ros-kinetic-robot-localization
```

当安装这个功能包之后，下一步就是学习如何使用它。所以，在功能包中找到名为ekf_template.launch的启动文件：

```
$ rosed robot_localization ekf_template.launch
```

打开文件后，我们将看到以下代码：

```
<launch>
  <node pkg="robot_localization" type="ekf_localization_node"
  name="ekf_se" clear_params="true">

    <rosparam command="load" file="$(find
    robot_localization)/params/ekf_template.yaml" />
    <!-- Placeholder for output topic remapping
<remap from="odometry/filtered" to=""/>
-->
  </node>
</launch>
```

我们将看到用于启动ekf_localization_node和加载存储在ekf_template.yaml中的一些参数的代码。yaml文件类似于如下格式：

```
#Configuation for robot odometry EKF
#
frequency: 50

odom0: /odom
odom0_config: [false, false, false,
false, false, false,
true, true, true,
false, false, true,
false, false, false]
odom0_differential: false

imu0: /imu_data
imu0_config: [false, false, false,
false, false, true,
false, false, false,
false, false, true,
true, false, false]
imu0_differential: false

odom_frame: odom
base_link_frame: base_footprint
world_frame: odom
```

在这个文件中，定义了IMU的主题、里程计的主题和base_link_frame的名称。EKF滤波器仅使用来自IMU和里程计主题的数据，这通过在6×3矩阵中填写true或false实现。

将这个yaml文件命名为robot_localization.yaml并保存在chapter8_tutorials/config中，同时将创建一个新的启动文件在该章的launch文件夹下，名为robot_localization.yaml，如下：

```
<launch>
  <node pkg="robot_localization" type="ekf_localization_node"
  name="ekf_localization">
    <rosparam command="load" file="$(find
    chapter8_tutorials)/config/robot_localization.yaml" />
  </node>
</launch>
```

这时，我们完成了机器人的代码，可以测试所有的功能：

```
$ roscore
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
$ roslaunch razor_imu_9dof razor-pub-and-display.launch
$ roslaunch chapter8_tutorials chapter8_robot_encoders.launch
$ roslaunch chapter8_tutorials robot_localization
```

8.4 使用IMU——Xsens MTi

在下面的图片中，你会看到Xsens MTi，它是本节的主角。

Xsens MTi

Xsens IMU是可以在机器人中找到的一种典型的惯性传感器。在本节中，我们将会学习Xsens MTi在ROS中的使用方式以及如何使用传感器发布的主题。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中使用多种IMU设备，例如之前使用的Razor IMU。本节中，我们将使用Xsens IMU。它需要安装正确的驱动程序才能正常工作。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程序。

 注意：MicroStrain 3DM-GX2 IMU的驱动程序参见http://www.ros.org/wiki/microstrain_3dmgx2_imu。

带有Wii Motion Plus的Wiimote的驱动程序参见http://www.ros.org/wiki/wiimote。

为了使用这个设备，需要使用xsens_driver。可以通过下面的命令安装它：

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

使用下面的命令，也需要同时安装两个功能包，因为下载的驱动程序需要依靠它们运行。

```
$ rosstack 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，输入下面的命令：

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

无须任何更改，这个驱动程序将直接检测USB端口和波特率。

Xsens如何在ROS中发送数据

如果一切正常，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。要查看类型和字段，需要使用下面的命令行：

```
$ rostopic type /imu/data

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

/imu/data主题是sensor_msg/Imu。这些字段用于指明方向、加速度和速度。在本例中，将会使用orientation字段。先查看一个消息以了解发送数据的实例。可以通过下面的命令来查看：

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

你将看到类似如下的输出结果：

```
---
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orientation字段，就会看到4个变量，而不是大家通常认为的3个。这是因为在ROS中，特殊的方向需要用四元数来表示。你能够从网上找到关于这种简明又无歧义的方向表达方式的很多文献。

可以在rviz run中观察IMU的方向并添加imu显示类型：

```
$ rosrun rviz rviz
```

8.5 GPS的使用

全球定位系统（GPS）是一种基于空间的卫星系统，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的信息。你必须与4个GPS卫星之间有畅通的通道才能获得有效的数据。

通过GPS获得的数据符合由NMEA建立的多个通信标准并遵循一个具有不同句子类型的协议。在这里，可以得到接收器的位置信息。

 注意：关于NMEA消息类型的更多信息，可访问 http://www.gpsinformation.org/dale/nmea.htm。

GPS中最有趣的信息包含在GGA中，它们为当前的fix数据提供GPS的3D位置。下面是一个句子示例和每一个字段的解释：

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
Where:
GGA Global Positioning System Fix Data
123519 Fix taken at 12:35:19 UTC
4807.038,N Latitude 48 deg 07.038' N
01131.000,E Longitude 11 deg 31.000' E
1 Fix quality: 0 = invalid
1 = GPS fix (SPS)
2 = DGPS fix
3 = PPS fix
4 = Real Time Kinematic
5 = Float RTK
6 = estimated (dead reckoning) (2.3 feature)
7 = Manual input mode
8 = Simulation mode
08 Number of satellites being tracked
0.9 Horizontal dilution of position
545.4,M Altitude, Meters, above mean sea level
46.9,M Height of geoid (mean sea level) above WGS84
ellipsoid
(empty field) time in seconds since last DGPS update
(empty field) DGPS station ID number
*47 the checksum data, always begins with *
```

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动力学（RTK）模式下，第二个GPS在一个已知的位置向第一个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。可以发现使用TTL、RS232的设备，它们通过USB适配器很容易与计算机相连。这一节，我们将用到廉价的GPS（EM-406a）和一个精确的系统，如在RTK模式下的GR-3　Topcon。在相同的驱动程序下，我们将会看到可以从这两个设备中得到经度、纬度和高度。

EM-406a and Topcon GPS

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

```
$ sudo apt-get install ros-kinetic-nmea-gps-driver

$ rosstack profile & rospack profile
```

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

```
$ rosrun 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。如果我们想在ROS下用它们，必须像下面的命令行中那样声明_baud参数。

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

8.5.1 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驱动程序将用sensor_msgs/NavSatFix消息发送GPS的状态信息。

```
$ rostopic type /fix

sensor_msgs/NavSatFix
```

/fix主题是sensor_msg/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，这个方法称为墨卡托方位法（UTM）。

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
---
```

8.5.2 创建一个使用GPS的工程示例

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。可以在chapter8_tutorials的c8_fixtoUTM.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, "fixtoUTM");
  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。

通过这种方式可以在ros::Subscriber gps_sub=n.subscribe("fix",10,gpsCallBack)main函数中订阅/fix主题。

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，并将发布一个名为/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。

可以在启动GPS驱动程序之后使用下面的命令来验证一下代码：

```
$ rosrun chapter8_tutorials c8_fixtoUTM
```

可以使用GPS数据来改进机器人定位，通过使用卡尔曼滤波器将里

程计、IMU与NavSatFix数据进行融合。

8.6 使用激光测距仪——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。用于实现这个目的的传感器是LIDAR。这个传感器专门用于测量机器人和物体之间的距离。

Hokuyo URG-041x

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LIDAR。本章的传感器选用Hokuyo URG-04lx测距仪。可以通过以下链接获取更加详细的信息：http://www.hokuyo-aut.jp/。Hokuyo测距仪是能够实现机器人实时导航和地图创建的设备：

标准的Hokuyo URG-04lx是机器人领域最常用的低成本测距仪。它具有极佳的分辨率而且非常容易上手。要使用它，首先需要安装激光的

驱动程序：

```
$ sudo apt-get install ros-kinetic-hokuyo-node

$ rosstack profile && rospack profile
```

一旦驱动程序安装完毕，就应检查全部内容是否安装正确。连接激光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
```

当连接激光时，系统能够查看它，因为上述命令行的结果是以下输出：

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

在该示例中，需要重新配置激光设备以便ROS能够访问和使用它，也就是说，需要获取适当的访问权限：

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

使用下面的命令行检查重新配置是否成功：

```
$ 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上。在一个命令行窗口中启动roscore，在另一个命令行窗口中执行以下命令：

```
$ rosrun hokuyo_node hokuyo_node
```

如果一切正常，你会看到以下输出：

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

8.6.1 了解激光如何在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
```

然后你会在输出中看到以下主题：

```
/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信息的消息类型：

```
sensor_msgs/LaserScan
```

使用下面的命令查看消息的数据结构：

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

如果想了解关于激光工作原理和发送哪些数据的更多信息，可以使

用rostopic命令来查看实时消息：

```
$ rostopic echo /scan
```

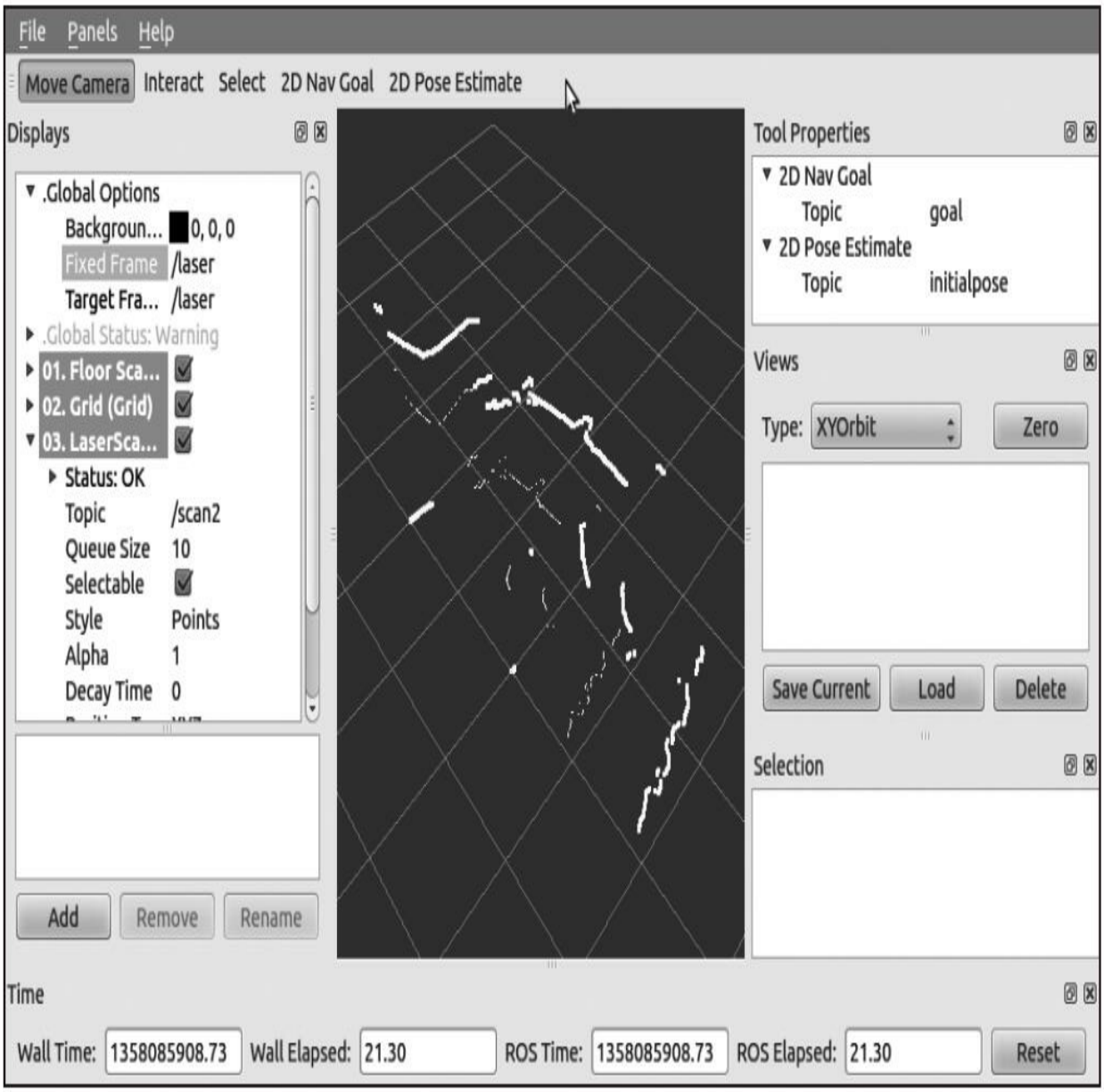
然后，会看到激光发送的消息如下：

```
---
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：

```
$ rosrun rviz rviz -d 'rospack find chapter8_tutorials'/config/laser.rviz
```

下图显示了消息的图形化展示：



RViz中的激光可视化

你能够从屏幕上看出轮廓。如果移动激光传感器，你会看到轮廓持续变化。

8.6.2 访问和修改激光数据

现在，我们将会创建一个节点以获取激光数据，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/chapter8_tutorials/src文件夹下将以下代码段复制到c8_laserscan.cpp文件中：

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

#include<stdio.h>
using namespace std;
class Scan2{
  public:
  Scan2();
  private:
  ros::NodeHandle n;
  ros::Publisher scan_pub;
  ros::Subscriber scan_sub;
  void scanCallBack(const sensor_msgs::LaserScan::ConstPtr&
  scan2);
};
Scan2::Scan2()
{
  scan_pub = n.advertise<sensor_msgs::LaserScan>("/scan2",1);
  scan_sub = n.subscribe<sensor_msgs::LaserScan>("/scan",1,
  &Scan2::scanCallBack, this);
}

void Scan2::scanCallBack(const sensor_msgs::LaserScan::ConstPtr&
scan2)
{
  int ranges = scan2->ranges.size();
  //populate the LaserScan message
  sensor_msgs::LaserScan scan;
  scan.header.stamp = scan2->header.stamp;
  scan.header.frame_id = scan2->header.frame_id;
  scan.angle_min = scan2->angle_min;
  scan.angle_max = scan2->angle_max;
  scan.angle_increment = scan2->angle_increment;
  scan.time_increment = scan2->time_increment;
  scan.range_min = 0.0;
  scan.range_max = 100.0;
  scan.ranges.resize(ranges);
  for(int i = 0; i < ranges; ++i)
  {
    scan.ranges[i] = scan2->ranges[i] + 1;
  }
  scan_pub.publish(scan);
}
int main(int argc, char** argv)
{
  ros::init(argc, argv, "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主题。第二个主题具有激光中的原始数据，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。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。

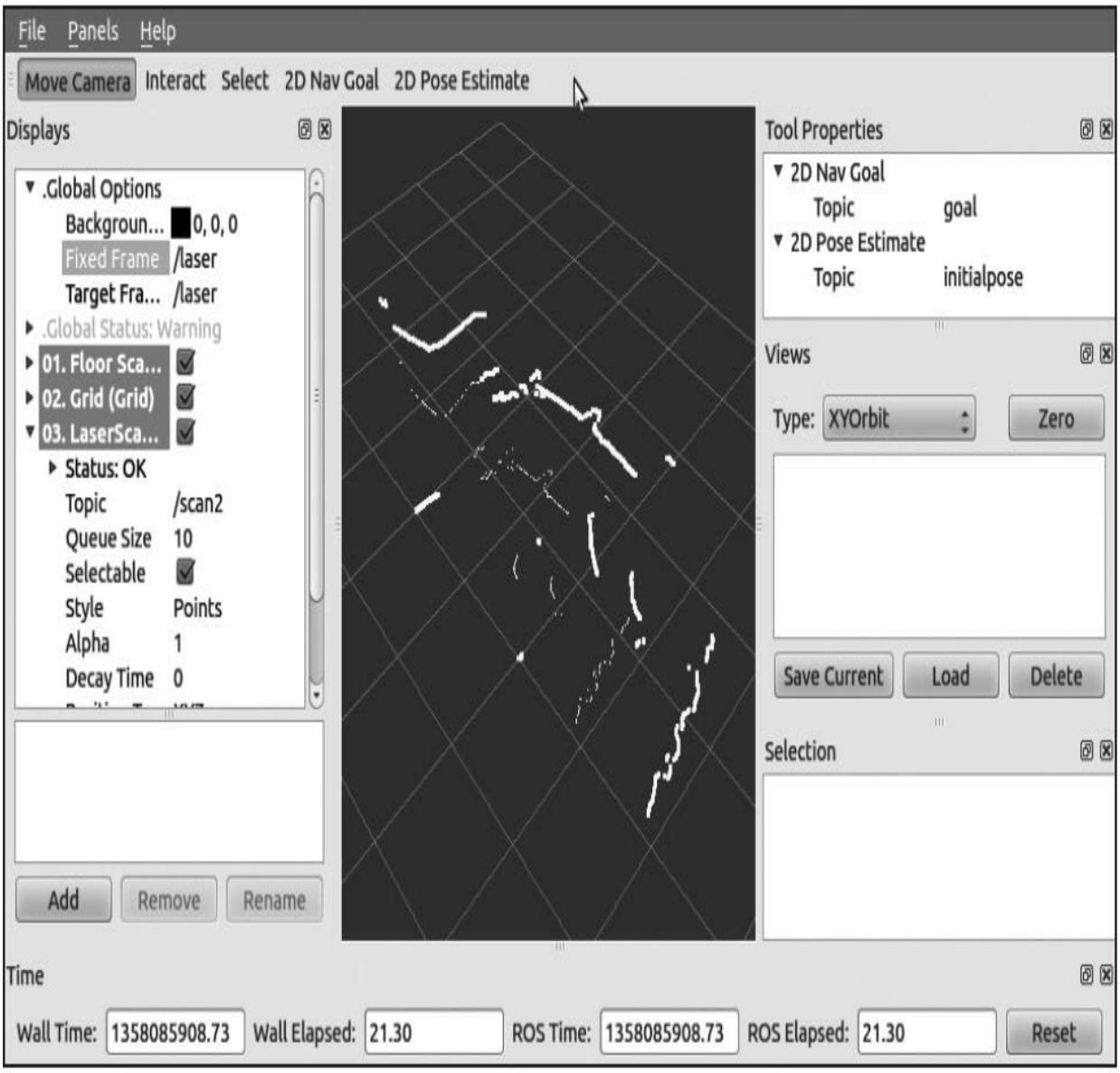
```
void Scan2::scanCallBack(const sensor_msgs::LaserScan::ConstPtr&
scan2)
{
  ...
  sensor_msgs::LaserScan scan;
  scan.header.stamp = scan2->header.stamp;
  ...
  ...
  scan.range_max = 100.0;
  scan.ranges.resize(ranges);
  for(int i = 0; i < ranges; ++i){
    scan.ranges[i] = scan2->ranges[i] + 1;
  }

  scan_pub.publish(scan);
}
```

8.7 创建launch文件

为了启动这一切，需要创建一个launch文件chapter8_laserscan.launch：

```
<launch>
  <node pkg="hokuyo_node" type="hokuyo_node" name="hokuyo_node"/>
  <node pkg="rviz" type="rviz" name="rviz"
  args="-d $(find chapter8_tutorials)/config/laser.rviz"/>

  <node pkg="chapter8_tutorials" type="c8_laserscan"
  name="c8_laserscan" />
</launch>
```

现在，如果启动chapter8_laserscan.launch文件，会启动三个节点：hokuyo_node、rviz和c8_laserscan。你将会在RViz可视化的界面内看到两条激光轮廓。绿色的轮廓线（见彩插18）是新数据产生的，如下图所示。


File Panels Help
Move Camera Interact Select 2D Nav Goal 2D Pose Estimate
Displays
.Global Options
Backgroun... 0, 0, 0
Fixed Frame /laser
Target Fra... /laser
.Global Status: Warning
01. Floor Sca...
02. Grid (Grid)
03. LaserSca...
Status: OK
Topic /scan2
Queue Size 10
Selectable
Style Points
Alpha 1
Decay Time 0
Add Remove Rename
Tool Properties
2D Nav Goal
Topic goal
2D Pose Estimate
Topic initialpose
Views
Type: XYOrbit Zero
Save Current Load Delete
Selection
Time
Wall Time: 1358085908.73 Wall Elapsed: 21.30 ROS Time: 1358085908.73 ROS Elapsed: 21.30 Reset

8.8 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电视机附近用于放置XBOX 360的架子上。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（它是一个红外色斑投影仪）和一个单色CMOS传感器配合工作，以获取物体的深度信息并转换为3D数据。

·用于分离玩家的声音和室内噪声的多阵列麦克风。

Kinect

在ROS中，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，需要安装功能包和驱动程序。使用下面的命令

行来安装它们：

```
$ sudo apt-get install ros-kinetic-openni-camera ros-kinetickinetic-
openni-launch
$ rosstack profile && rospack profile
```

一旦功能包和驱动程序安装完成，插入Kinect传感器，我们就能运行节点并开始使用它。在命令行窗口中，启动roscore。在另外一个命令行窗口中运行下面的指令：

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

如果一切正常，你不会看到任何错误消息。

8.8.1 Kinect如何发送和查看传感器数据

现在试一下我们能够使用这些节点做什么事情。使用以下命令列出已创建的主题：

```
$ 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，将会使用image_view功能包。在一个命令行窗口中运行以下指令：

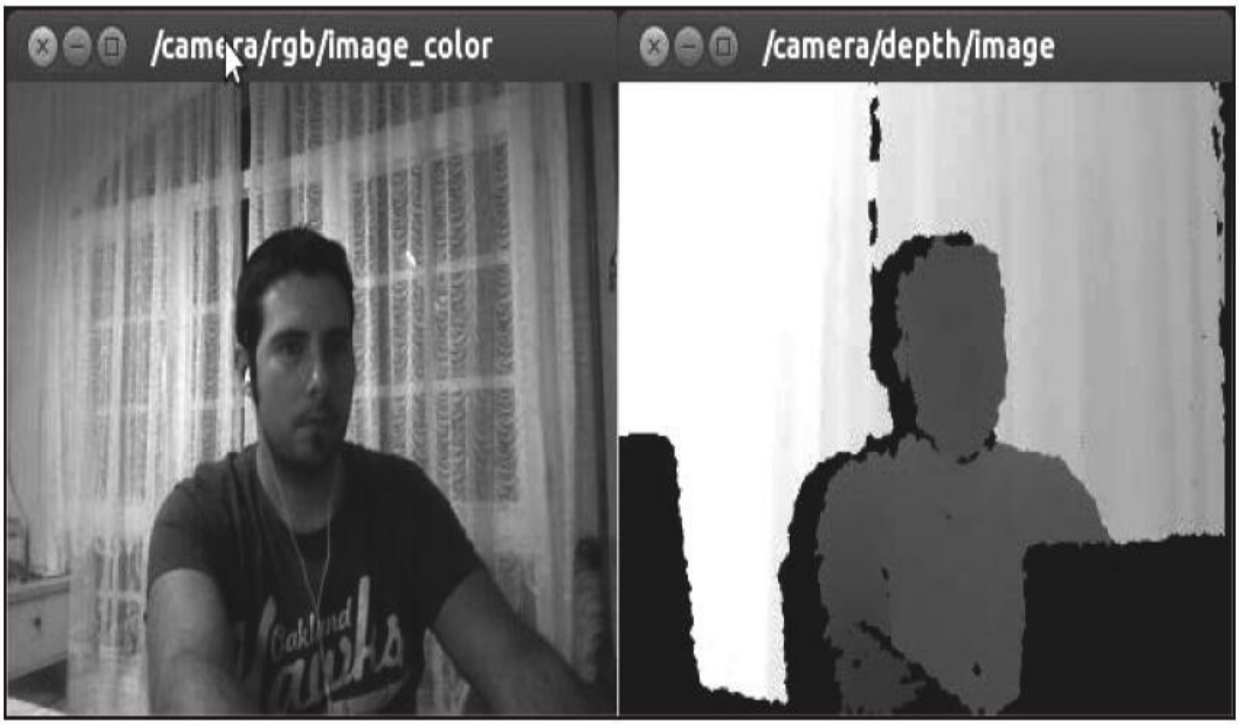
```
$ rosrun image_view image_view image:=/camera/rgb/image_color
```

注意，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：



Kinect中的RGB和深度图像

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。可以在以下主题中找到此类数据：/camera/depth/points、/camera/depth_registered/points等。

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

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

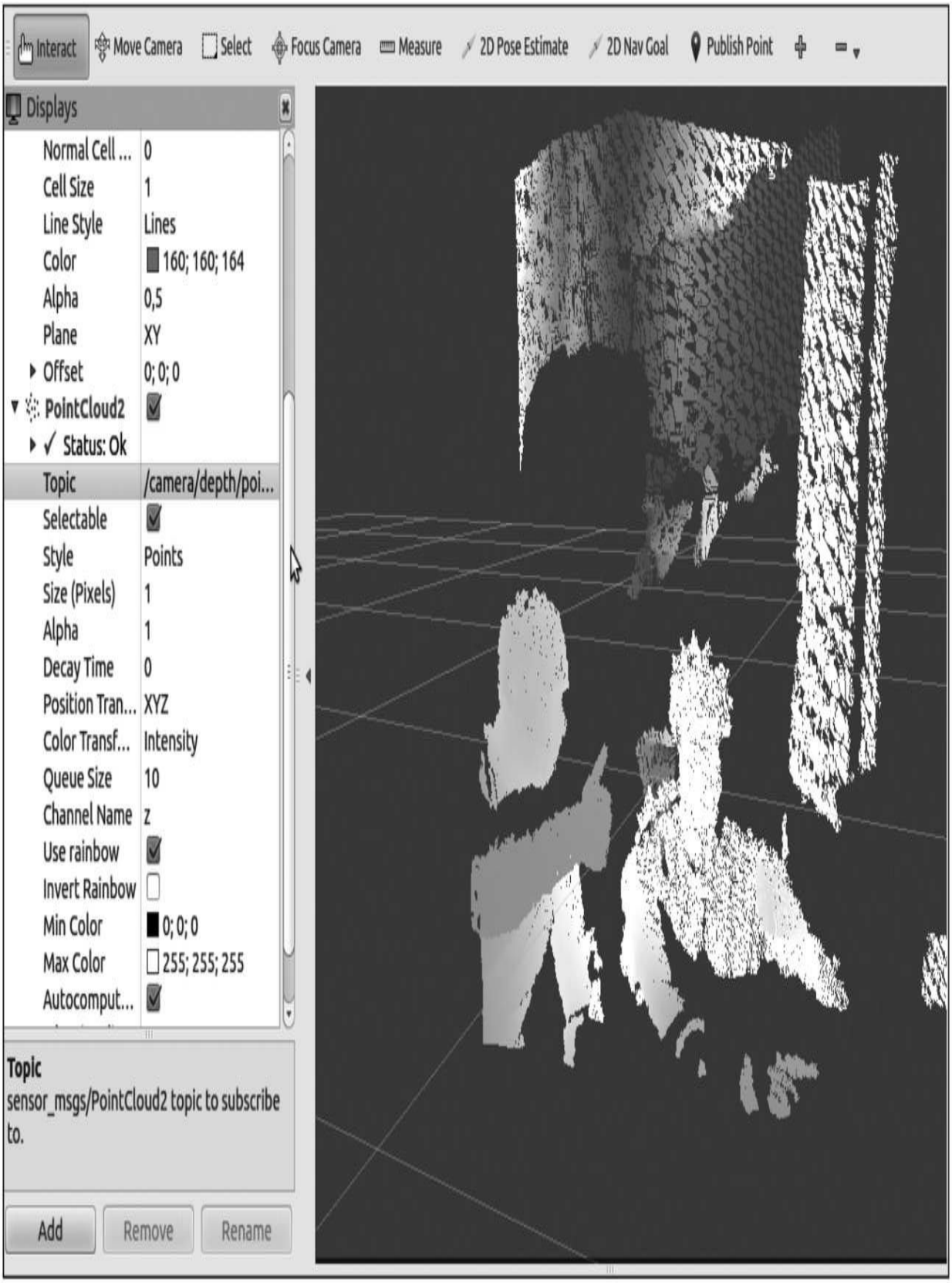
 注意：如果想查看消息的官方说明，请访问 http://ros.org/doc/api/sensor_msgs/html/msg/PointCloud2.html。

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：

```
$ rosrun rviz rviz
```

单击Add按钮，按显示类型订阅主题，并选择PointCloud2。一旦添加PointCloud2显示类型，就必须选择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。


Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Normal Cell ... 0
Cell Size 1
Line Style Lines
Color 160; 160; 164
Alpha 0,5
Plane XY
Offset 0; 0; 0
PointCloud2
Status: Ok
Topic /camera/depth/poi...
Selectable
Style Points
Size (Pixels) 1
Alpha 1
Decay Time 0
Position Tran... XYZ
Color Transf... Intensity
Queue Size 10
Channel Name z
Use rainbow
Invert Rainbow
Min Color 0; 0; 0
Max Color 255; 255; 255
Autocomput...
Topic
sensor_msgs/PointCloud2 topic to subscribe to.
Add
Remove
Rename

Kinect中的3D点云数据

8.8.2 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少原始数据中点的数量，从而减少采样的数据。

在chapter8_tutorials/src文件夹下创建一个新文件c8_kinect.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 dataSize
  pub.publish (cloud_filtered);
}

int main (int argc, char** argv)
{
  // Initialize ROS
  ros::init (argc, argv, "c8_kinect");
  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 ();
}
```

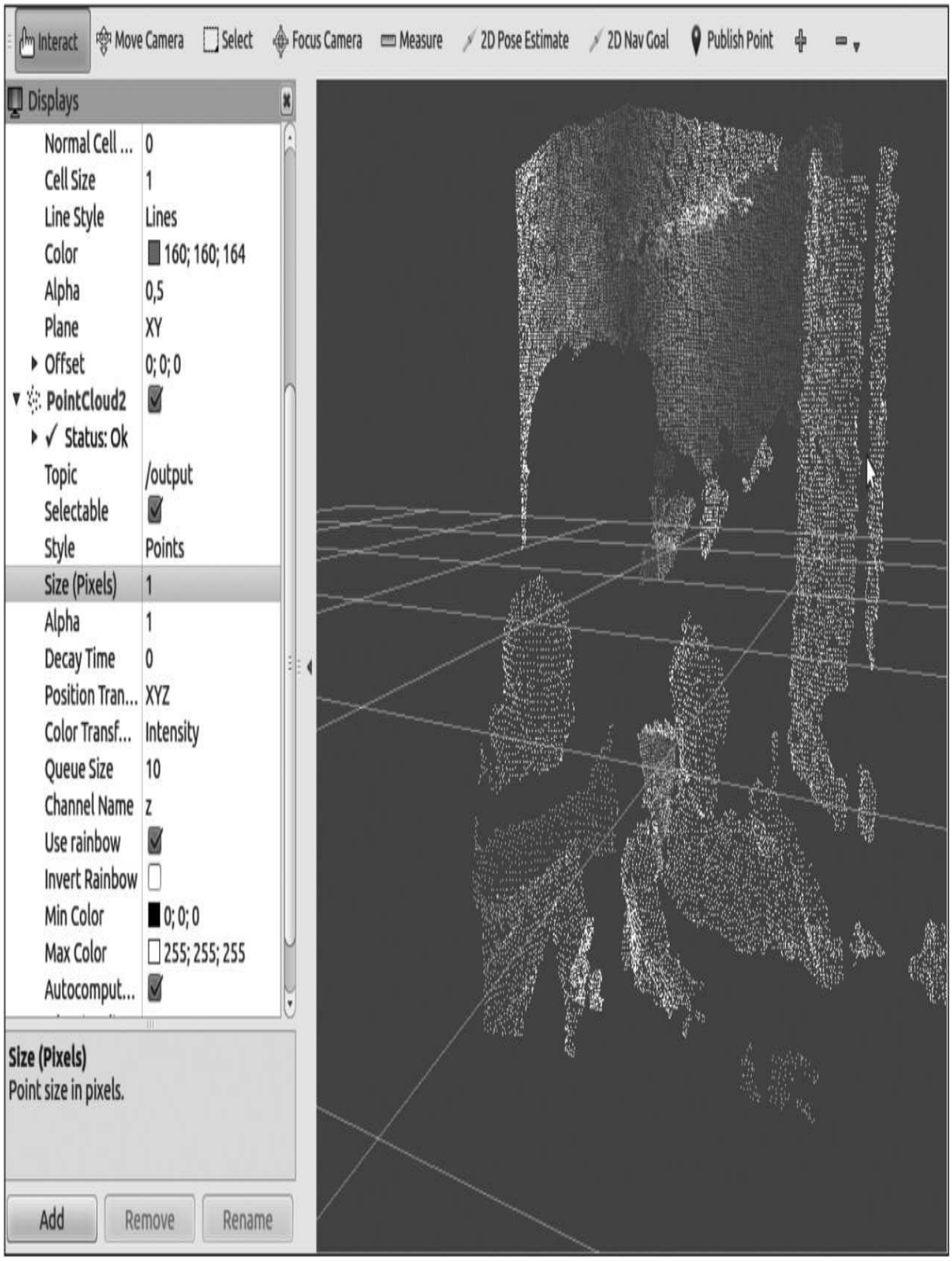
这个例子基于点云库（Point Cloud Library，PCL）的教程。

 注意：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。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，会很容易发现分辨率比原有的数据低了不少，如下图所示。


Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Normal Cell ... 0
Cell Size 1
Line Style Lines
Color 160; 160; 164
Alpha 0,5
Plane XY
Offset 0; 0; 0
PointCloud2
Status: Ok
Topic /output
Selectable
Style Points
Size (Pixels) 1
Alpha 1
Decay Time 0
Position Tran... XYZ
Color Transf... Intensity
Queue Size 10
Channel Name z
Use rainbow
Invert Rainbow
Min Color 0; 0; 0
Max Color 255; 255; 255
Autocomput...
Size (Pixels)
Point size in pixels.
Add
Remove
Rename

减少采样数据后的3D点云数据

在rviz中，你能够看到一个消息中包含的点数。对于原始数据，我们能看到点的数量是219075。而新的点云中，数量只有16981。可见数据大幅减少。

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。

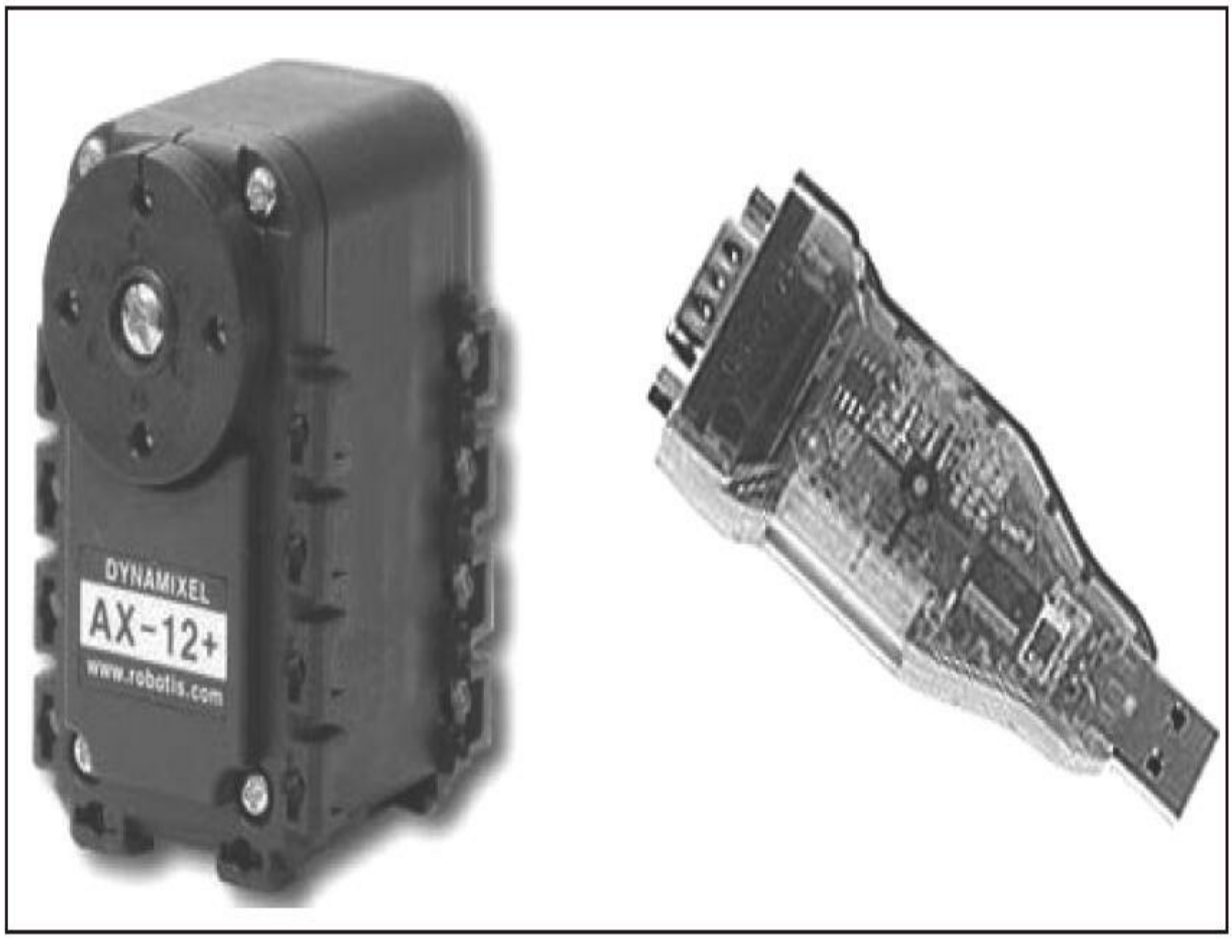
8.9 使用伺服电动机——Dynamixel

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 servomotor）。

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。



Dynamixel AX-12+电动机

首先，我们将会安装必需的功能包和驱动程序。在命令行窗口输入下面的命令：

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

$ rosstack profile && rospack profile
```

一旦必要的功能包和驱动程序安装完毕，将转换器连接到计算机并检查它能否检测到。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，就启动节点并试一试这个伺服电动机。

在命令行窗口中启动roscore，然后在另外一个窗口中输入下面的命令：

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

如果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.
```

8.9.1 Dynamixel如何发送和接收运动命令

一旦启动了controller_manager.launch文件，就可以查看主题列表。记住，请使用下面的命令行查看主题：

```
$ 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。在本例中，仅仅有一个ID为6的电动机。然而，不能使用这些主题来驱动电动机，所以需要加载另一个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，在主题清单中会出现两个新主题，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，需要输入下面的命令行：

```
$ rostopic type /tilt_controller/command


std_msgs/Float64
```

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28.6478898°的位置上。

8.9.2 创建和使用伺服电动机示例

现在，我们将展示如何使用节点来驱动电动机。在/chapter8_tutorials/src文件夹下创建一个新的c8_dynamixel.cpp文件，并输入下面的代码：

```
#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, "c8_dynamixel");
  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°～180°之间不断运动。这是一个非常简单的例子，但你能够使用它进行复杂的运动或者控制更多的电动机。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。

8.10 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交互的唯一办法。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，如关于仿真的章节所述，测距激光是导航算法中最有用的传感器。因为它是实现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刷新率的数据。

第9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，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摄像头的驱动程序，但是对于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，本章将讨论其中的一些，此外还提供了一个使用OpenCV视频捕捉API的驱动程序，该驱动程序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多个摄像头的装置。我们将会进一步介绍这一切是如何集成在系统中的。例如，双目视觉也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。

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），我们能够实现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。这里不会深入探讨OpenCV　API，因为这超出了本书的范围。但是，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，通过一个示例，简单介绍如何在节点中使用OpenCV，具体是用特征检测、描述符提取和匹配方法计算两幅图像之间的单应性。此外，本章末尾的一个教程会展示如何在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。也会讲述其他视觉里程计库，例如fovis，以及一些关于如何使用它们和如何提高RGBD传感器效果的建议，如Kinect。甚至还有传感器融合或在单目视觉方面的一些信息。

9.1 ROS摄像头驱动程序支持

不同的摄像头及其不同的使用方式将在下面的章节中介绍。实质上，对FireWire和USB摄像头进行了区分。

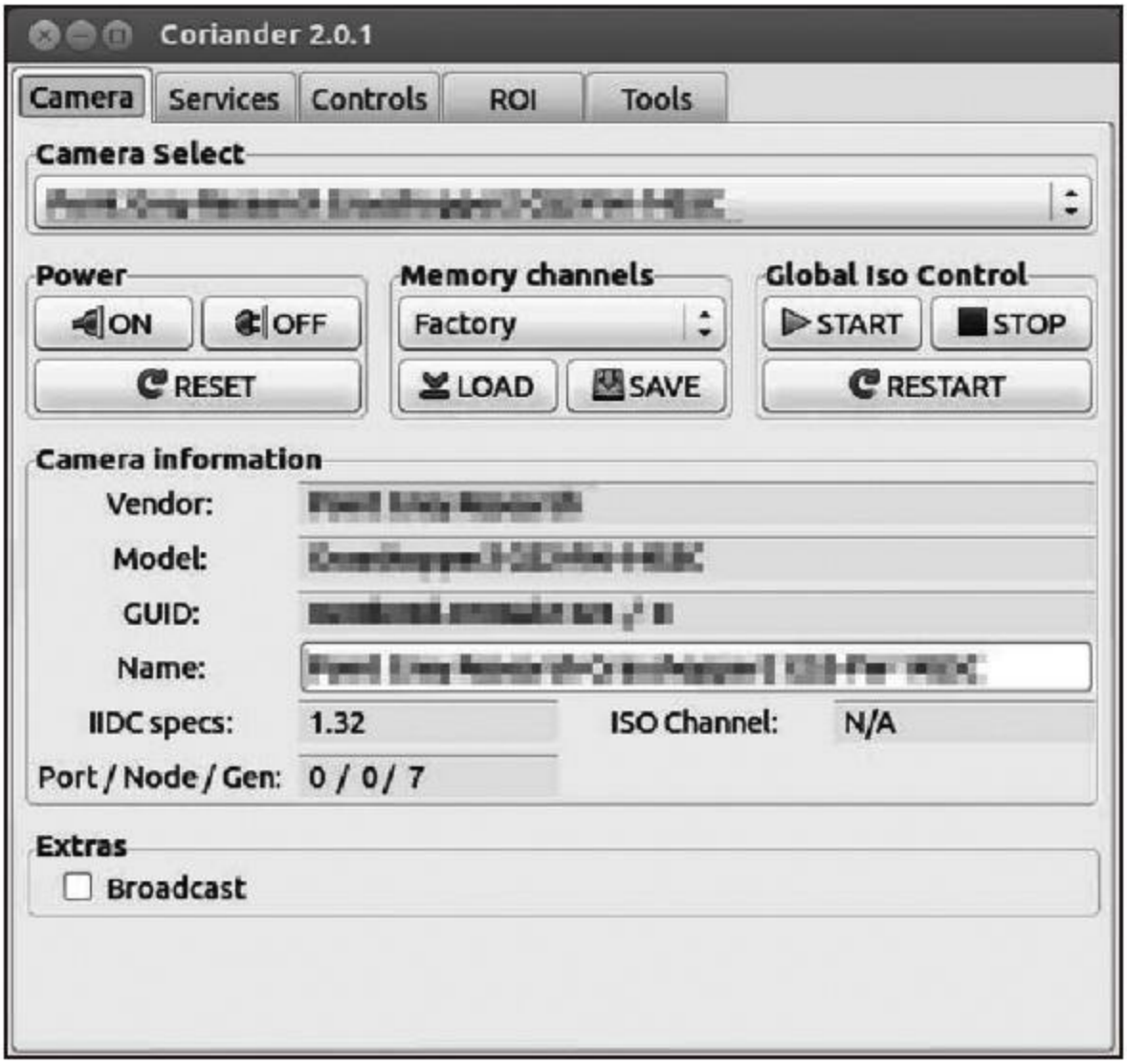
第一步是将摄像头与计算机连接，运行驱动程序并检查它在ROS中获取的图像。其实在使用ROS之前，最好使用其他工具来检查摄像头是否被操作系统正确识别，在这里可以使用Ubuntu发行版。我们将从FireWire摄像头讲起，因为ROS对它的支持更好，然后再学习USB摄像头。

9.1.1 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，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，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。

既然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摄像头驱动程序（需要roscore运行）。通过以下命令安装摄像头驱动程序功能包：

```
$ sudo apt-get install ros-kinetic-camera1394
```

如果在ROS Kinetic上它还不可用，需要在工作区用源码进行编译，代码可以从https://github.com/ros-drivers/camera1394获取，直接复制源文件到工作区并编译。

使用下面的命令运行摄像头驱动程序：

```
$ 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。

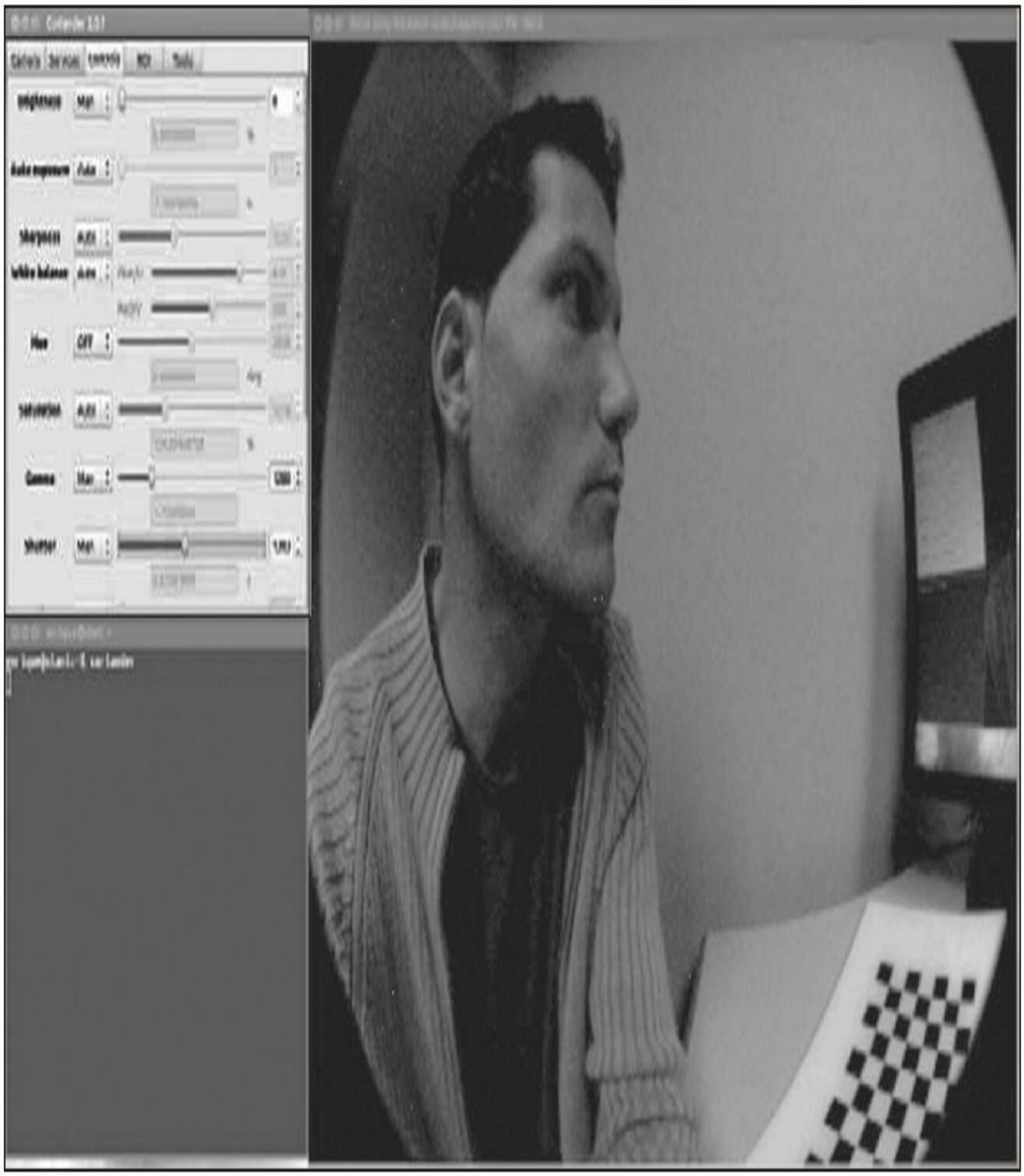
FireWire摄像头支持的参数在camera1394/config/firewire_camera/format7_mode0.yaml文件中列出并赋值，如下所示：

```
guid: 00b09d0100ab1324 # (defaults to first camera on bus)
iso_speed: 800 # IEEE1394b
video_mode: format7_mode0 # 1384x1036 @ 30fpsbayer 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)
```

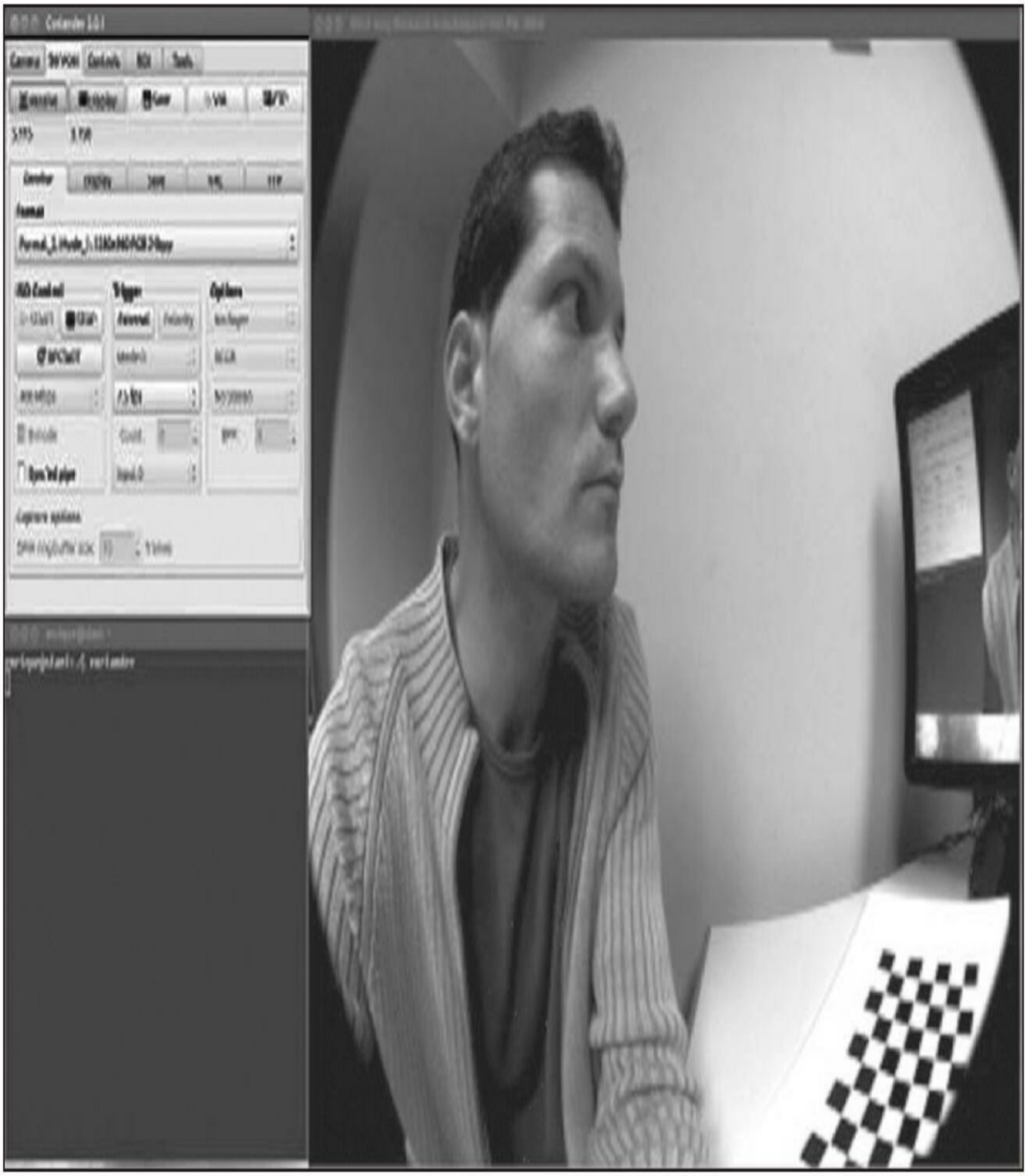
```
sharpness: 1000
auto_shutter: 3 # Manual (3)
#shutter: 1000 # = 10ms
shutter: 1512 # = 20ms (1/50Hz), max. in30fps
auto_white_balance: 3 # Manual (3)
white_balance_BU: 820
white_balance_RV: 520
frame_id: firewire_camera
camera_info_url:
package://chapter5_tutorials/calibration/firewire_camera/
calibration_firewire_camera.yaml
```

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，具体配置取决于所处环境的光线条件，而且你必须适应它们。如果使用coriander或rqt_reconfigure界面还是比较容易做到的。在下图中，我们能看到在coriander中的特定配置和输出图像：

```
$ rosrun rqt_reconfigure rqt_reconfigure /camera
$ coriander
```



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="$(find
  chapter5_tutorials)/config/firewire_camera/format7_mode0.yaml"
  />
  <!-- Camera driver -->
  <node pkg="camera1394" type="camera1394_node"
  name="camera1394_node">
    <rosparam file="$(argparams)" />
  </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驱动程序。然后，它会运行图像管道以获得使用去拜耳化算法（Debayer algorithm）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，只需要在其参数文件中设定好GUID，并运行以下命令：

```
$ roslaunch chapter9_tutorials firewire_camera.launch view:=true
```

然后，你还能够使用rqt_reconfigure进行动态地配置。

9.1.2 USB摄像头

现在我们将会学习对USB摄像头进行相同的配置。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电脑，那么它可能是内置的网络摄像头）。

需要注意的是，有两个主要的选项能用于ROS的USB摄像头驱动程序。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-kinetic-usb-cam
```

然后，运行以下命令：

```
$ roslaunch chapter5_tutorials usb_cam.launch view:=true
```

直接运行rosrun 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-kinetic-gscam
```

接着运行下面的命令：

```
$ roslaunch chapter5_tuturials gscam.launch view:=true
```

对于usb_cam，这个launch文件运行了rosrun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 ! ffmpegcolorspace
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。

9.1.3 使用OpenCV制作USB摄像头驱动程序

虽然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摄像头驱动程序。驱动程序是通过调用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，还允许我们修改它的一些参数。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，可以进行更多的控制。可以在ROS中改变摄像头配置，并查看如何发布摄像头的图像和信息。根据摄像头图像的读取方式，有两种使用OpenCV实现摄像头驱动程序的方法。第一个是根据每秒钟给定的图像帧（Frames Per Second，FPS）进行轮询；第二个是能够为这类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，因此它在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，这样我们无须承担使用计时器和回调函数所造成的时间损失。在FPS较小时，计时器方法和轮询方法就差不多了，而前者的代码更清晰。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。为了节省时间，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程序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，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无缝地发布图像，其中包括各种压缩图像格式和在ROS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图像消息的转换，其中有灰度/颜色（grayscale/color）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程序的类。它的属性如下：

```
ros::NodeHandle nh;
image_transport::ImageTransport it;
image_transport::Publisher pub_image_raw;

cv::VideoCapture camera;
cv::Mat image;
cv_bridge::CvImagePtr frame;

ros::Timer timer;

intcamera_index;
int fps;
```

按照常规，需要节点句柄。然后，需要用于以任何可能的格式发送图像的ImageTransport对象。在代码中，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，使用OpenCV来捕捉图像/帧（images/frames）。在捕捉帧的时候，直接使用cv_brigde帧。它是CvImagePtr类型的，通过它可以直接访问其图像字段。

最后，有计时器和用于使驱动程序工作的基本摄像头参数。这可能是最基本的驱动程序。这些参数包括摄像头索引，也就是/dev/video?设备的编号，例如，0代表/dev/video0。这个摄像头索引被传递给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。

这个驱动程序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，打开摄像头，如果无法打开则需要中止程序。注意，必须在属性构造函数内完成这些工作，如下所示，这里camera_index作为参数进行传递：

```
camera(camera_index )
```

然后，读取fps参数并计算计时器周期。这些参数最终用于创建计时器并设定capture回调函数。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。

capture回调函数读取和发布的图像如下所示：

```
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，并检查是否实际上捕捉到帧。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：

```
$ rosrun chapter9_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文件格式通常是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：设定传感器模数转换器（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：决定图像的曝光，也就是设定图像的亮度。通常通过调整增益和快门速度（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9.7.2节也会用到它。

·camera_info_url：提供到摄像头信息的路径，所保存的内容主要是摄像头标定信息。

然后，在驱动程序中，通过以下语句使用动态再配置服务器：

```
#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 ) );
```

setCallback构造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类来配置摄像头的属性。其中包括了部分前面提到的参数。下面以frame_width参数为例进行说明：

```
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。注意，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程序的再配置。

为了能设定摄像头的信息，ROS有一个camera_info_manager库，它能够帮助我们完成这些工作。简而言之，使用以下代码：

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

我们用它来获取CameraInfo消息。现在，在计时器中的capture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，我们为第一版驱动程序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）从管理器获取摄像头信息，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程序：

```
$ 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的所有动态再配置参数。

你能够使用rostopic list、rostopic hz/camera/image_raw和image_view（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，我们将会在下面的小节中分别解释每一项工作。

9.2 ROS图像

ROS提供了sensor_msgs::Image消息在节点之间发送图像。然而，通常需要一个数据类型或对象来操作这些图像，以便做一些有用的工作。最常用的库是OpenCV，所以ROS提供了一个桥接类，用于在将OpenCV中图像与ROS中图像之间进行来回转换。

假设有一个OpenCV图像，那么它会是一个cv::Mat图像。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去共享或者复制图像。然而，如果可能，通过cv_bridge在CvImage类中使用OpenCV image字段更容易。这正是我们在摄像头驱动程序中的做法，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;
```

最后在某个时刻设置OpenCV图像，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。下一节介绍这些内容。

使用image_transport发布图像

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程序中的设计是一样的。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传输类型。

能够通过rostopic info命令来查看它们。使用下面的命令安装所有插件：

```
$ sudo apt-get install ros-kinetic-image-transport-plugins
```

在代码中，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摄像头驱动程序中的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进行发布了：

```
pub_image_raw.publish( frame->toImageMsg() );
```

9.3 ROS中的OpenCV库

ROS提供了与OpenCV（最广泛使用的开源计算机视觉库）的非常简单的集成。但是，它不会为ROS Kinetic提供一个特定的Debian，因此将迫使用户使用独立的版本。除此之外，它允许你在系统上使用最新的OpenCV库，并提供了在ROS中使用OpenCV的其他集成工具。下面几节将介绍如何安装OpenCV和其他工具。

9.3.1 安装OpenCV 3.0

在ROS Kinetic中，可以开始使用OpenCV 3.0，与以前的版本相比，其中某些软件包对OpenCV 2.*有依赖关系或与3.0有兼容性问题。

安装遵循安装Ubuntu软件包的标准工作流程，因此只须执行以下操作：

```
$ sudo apt-get install libopencv-dev
```

或者，还可以从库中安装ROS功能包：

```
$ sudo apt-get install ros-kinetic-opencv3
```

9.3.2 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>
```

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，必须在target_link_libraries中增加${OpenCV_LIBS}（在chapter9_tutorials的CMakeLists.txt中）。

在节点的cpp文件中，要包含所需的所有OpenCV库。例如，使用以下声明来包含highgui.hpp文件：

```
#include <opencv2/highgui/highgui.hpp>
```

现在，你能够在代码中使用任何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。

9.4 使用rqt_image_view显示摄像头输入的图像

第3章介绍了如何在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，如以下代码所示：

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

在这里最重要的是通过使用图像传输，我们能够选择不同的主题来查看图像，并在必要时使用压缩格式。还有在双目视觉中，使用rqt rviz还能查看由差分图像生成的点云。这会在后续章节中进行介绍。

9.5 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系数。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，后面会介绍这部分内容。而最终的结果就是我们获取了真实环境的3D信息。

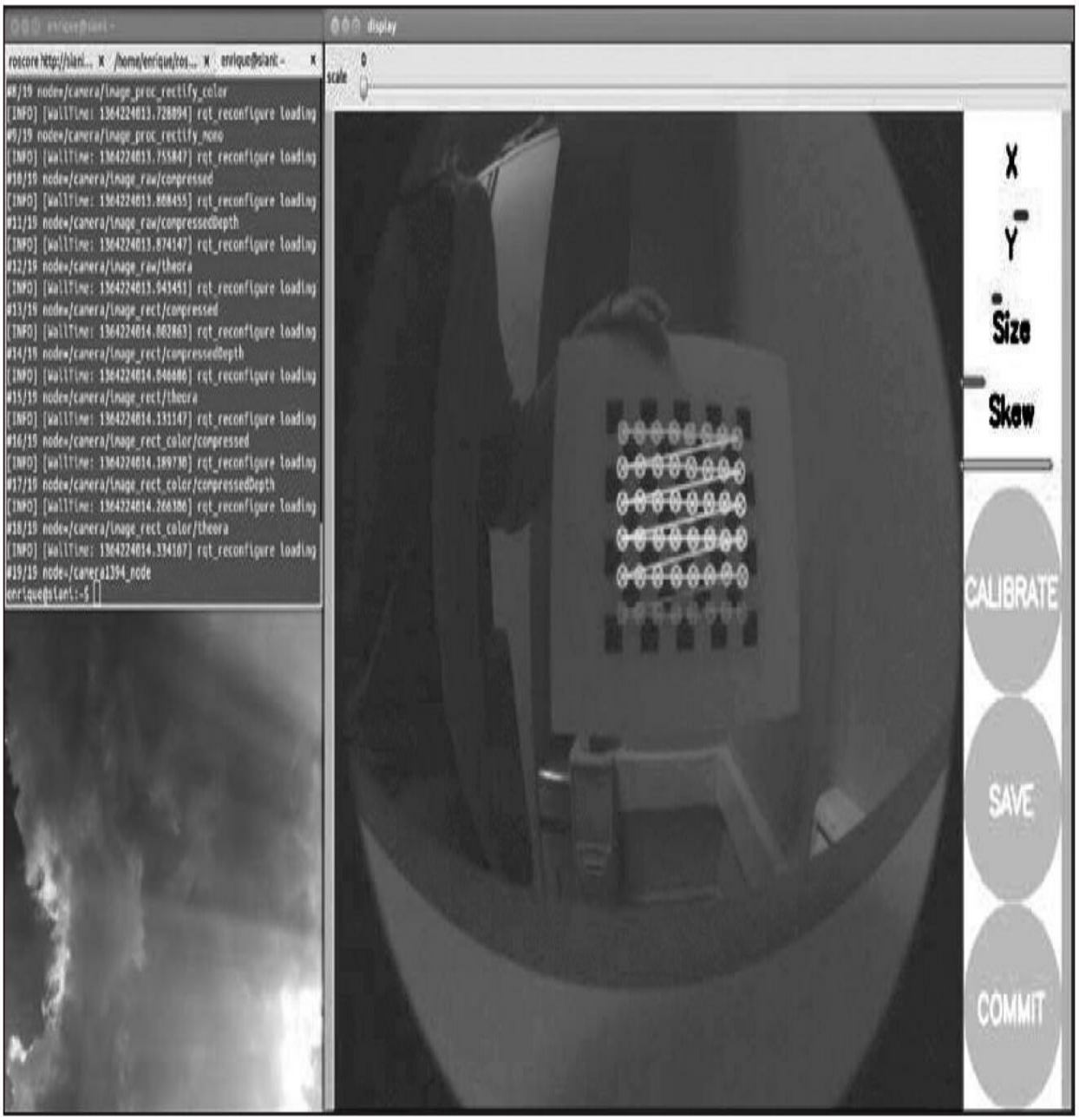
标定是通过使用一种已知的标定图案，并对图案的多种角度的视图进行辨识来实现的。最典型的图案是棋盘，也有以一列圆圈或者非对称的圆圈作为图案的，但要注意，当视图倾斜时，圆圈看起来成了椭圆。检测算法会获取棋盘上每个单元格的内角点，并用它们来估计摄像头的内参数和外参数。简单讲，外参数是摄像头的姿态，或者换句话说，就是当摄像头保持在某个固定位置时，图案相对于摄像头的位姿。我们最需要的是内参数，因为它不会发生改变，这就能在摄像头处于任何位姿时使用，能够测量图像的距离，修正图像的失真，也就是纠正图像。

9.5.1 如何标定摄像头

运行摄像头驱动程序，就能够使用ROS的标定工具完成摄像头标定。在此过程中，摄像头驱动程序提供了CameraInfo消息和camera_info_set服务。这个服务允许设定保存标定结果的文件的路径。之后每次使用摄像头时，标定信息就能够被图像管道加载。FireWire摄像头的camera1394驱动程序恰好满足这些先决条件。要标定FireWire摄像头，需要运行以下命令：

```
$ roslaunch chapter5_tutorialscalibration_firewire_camera_ chessboard.
launch
```

这会打开一个GUI，并自动选择标定图案的视图。它在右上角会提供一些指示条来展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。然后，scale表示标定范围从近到远（直到可以进行探测的最大距离）。最后，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，如下图所示。



你能看到每次图案重叠的时候标定算法还是能找到这些点。这些视图会自动选择并覆盖一定数量的不同视图。这样你在指示的引导下能够让指示条在某个时刻从左到右完全变绿。理论上，有两个视图就足够了，但是实际使用中一般都需要10个左右。而这个界面会捕捉更多（30～40个）。你应该避免过快的移动速度，因为模糊的图像对于检测来说是非常不利的。一旦这个工具收集了足够的视图，它就会允许你进行标定，也就是说，根据标定图案的视图中找到的点对针孔摄像头模型进行求解和系统优化，如下图所示。

然后，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服务来向摄像头提交标定结果。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_colour" />
  <remap from="camera" to="camera" />
</node>
```

标定工具只需要该图案的特征（这里方格数量和大小分别是8×6个和0.030平方米）、image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，还可以使用image_raw来替代image_color主题。

一旦保存了标定结果（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。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。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，根据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。在这里的功能包中，这个文件是calibration/firewire_camera/calibration_firewire_camera.Yaml，它由参数文件引用：

```
camera_info_url:
package://chapter5_tutorials/calibration/firewire_camera/calibrati
on_firewire_camera.yaml
```

现在，使用图像管道再次运行摄像头。如果正确标定摄像头，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，建议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_cal

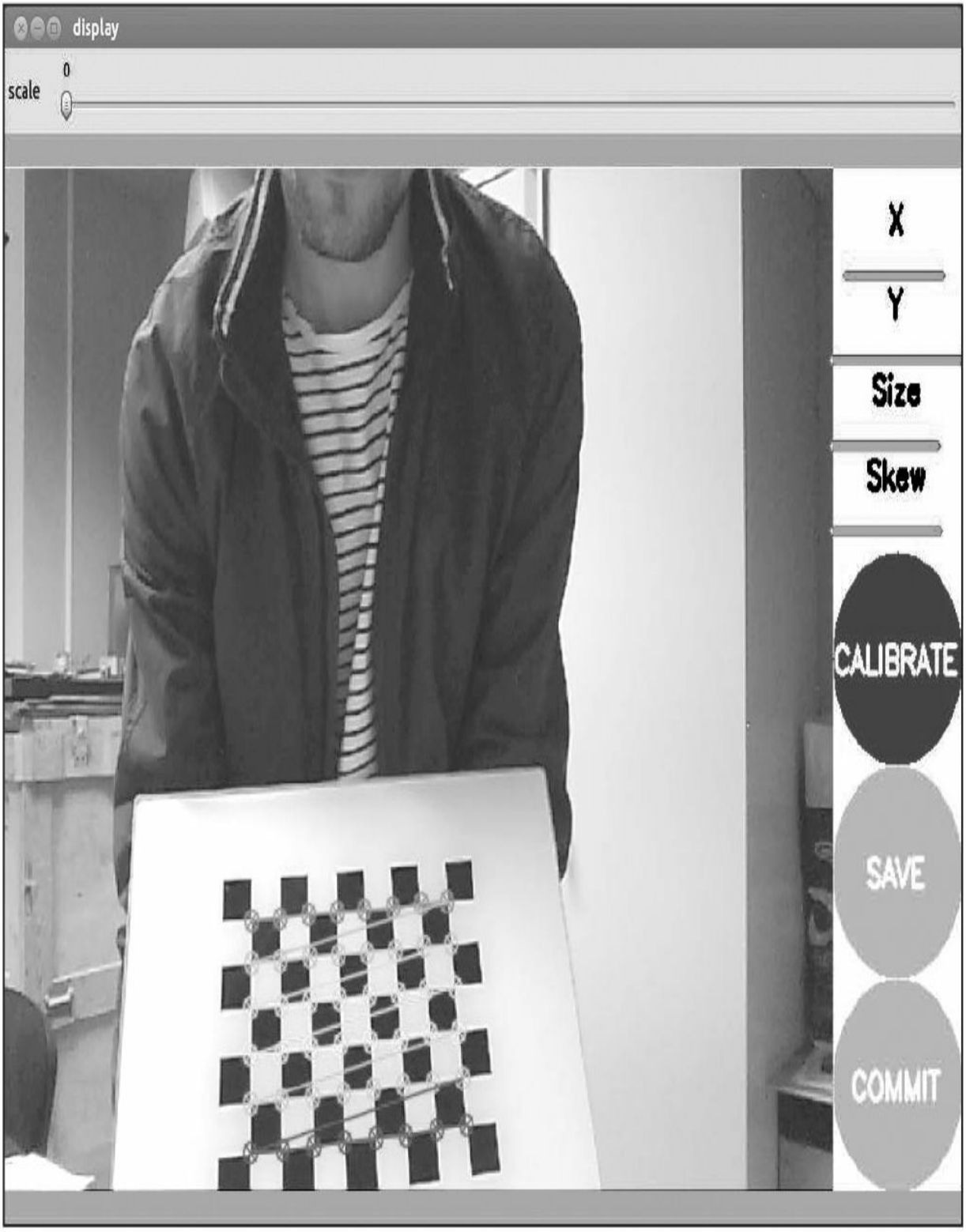
最后，还可以使用下面的launch文件试着应用不同的标定图案，如圆形和不对称圆形（查看https://raw.githubusercontent.com/opencv/opencv/05b15943d6a42c99e5f92这都针对FireWire摄像头：

```
roslaunchchapter5_tutorialscalibration_firewire_camera_circles.lau
nch
roslaunchchapter5_tutorialscalibration_firewire_camera_acircles.la
unch
```

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摄像头驱动程序也只这样做。对于USB摄像头驱动程序，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：

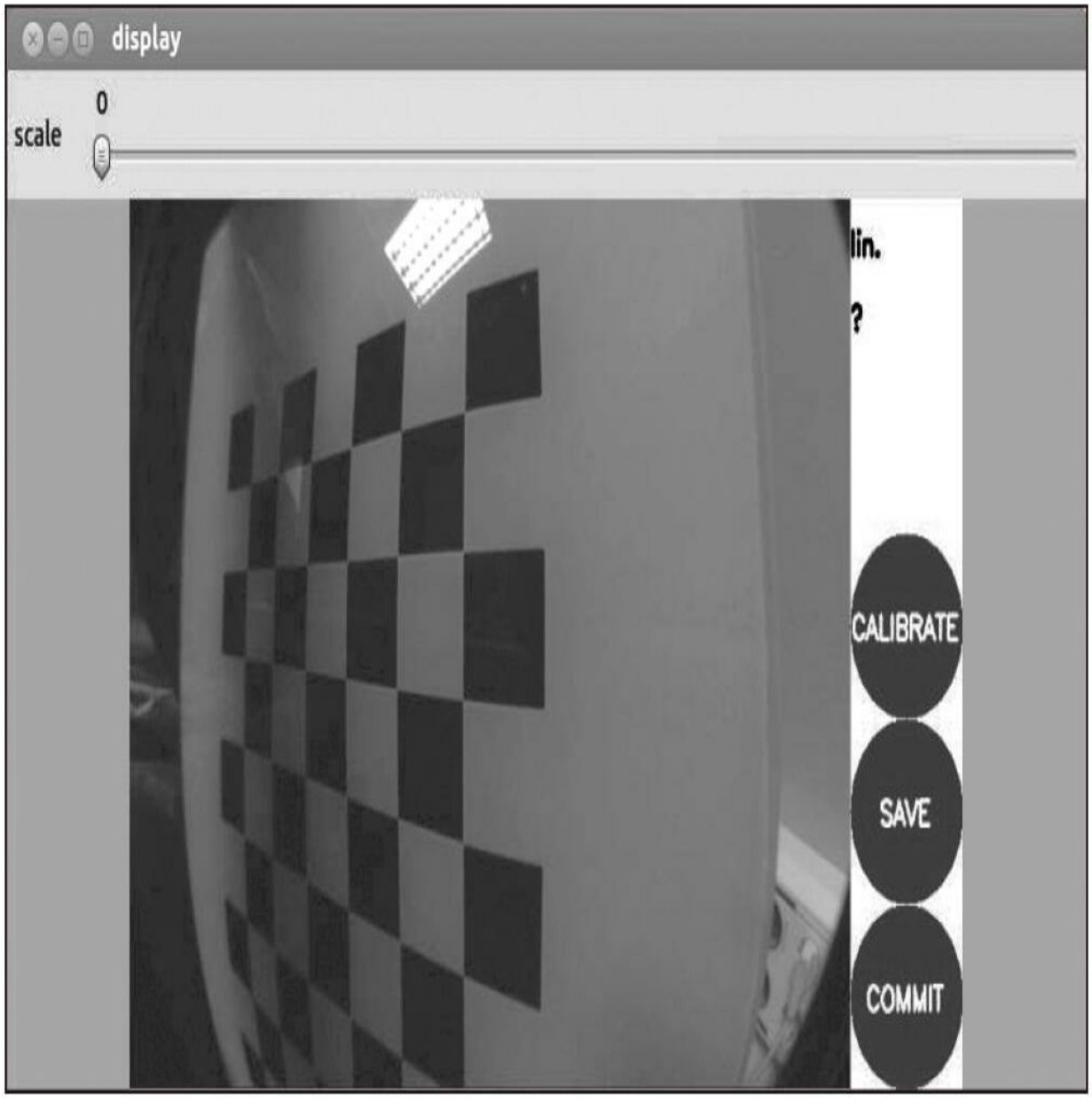
```
$ roslaunch chapter5_tutorials camera.launch calibrate:=true
```

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。



单击CALIBRATE（标定）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，允许保存（SAVE）标定数据并提交（COMMIT）到摄像头配置文件中，而且不需要做更多的设置。



9.5.2 双目标定

下面的章节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程序，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程序。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。

基线大约是12cm，这个参数在双目摄像头标定过程中也要计算进去。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。这样就总能把它们分配给/dev/video0和/dev/video1设备。当然，如果0被占用，就是1和2。另外，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它：

```
$ 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。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 : MJPEGioctl: 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)
```

如果支持MJPEG，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下你能够检查这个功能并测试你的摄像头。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双目视觉驱动程序基于前面章节介绍过的USB摄像头驱动程序。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。能够运行它并使用以下命令查看图像：

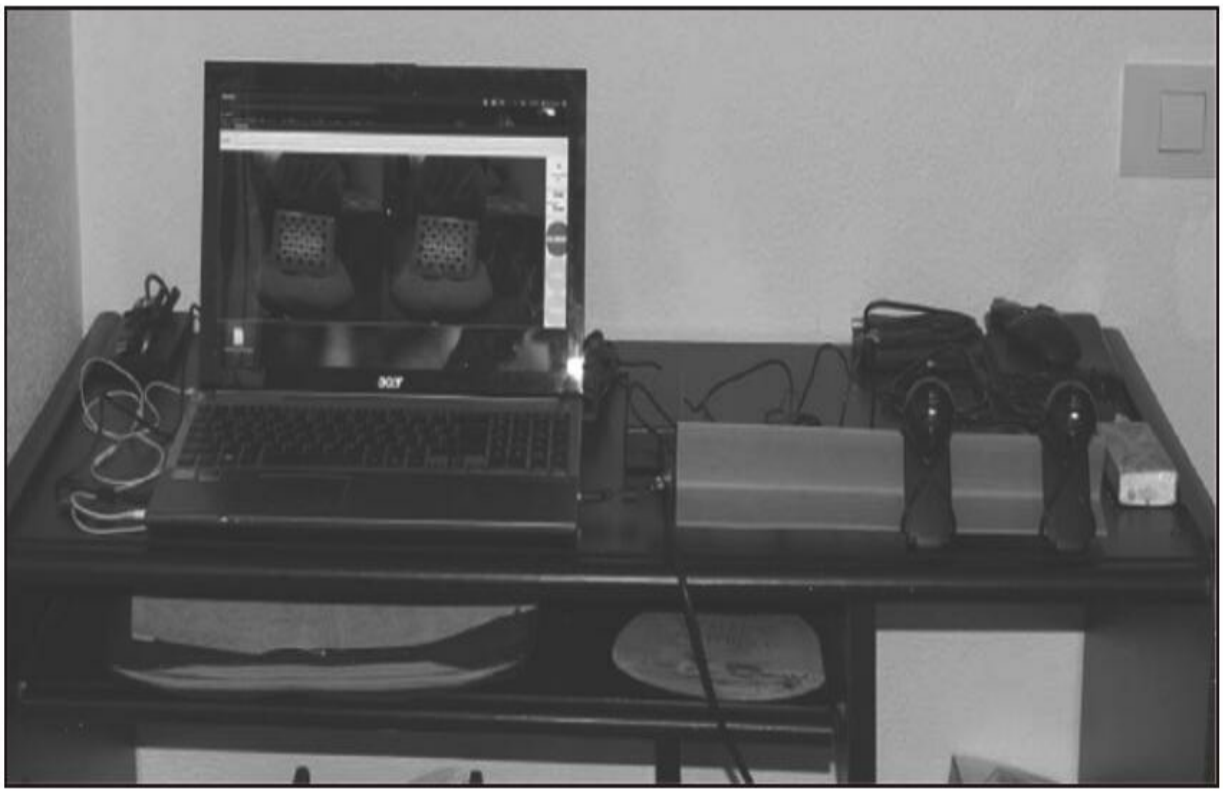
```
$ roslaunch chapter5_tutorials camera_stereo.launch view:=true
```

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，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，有：

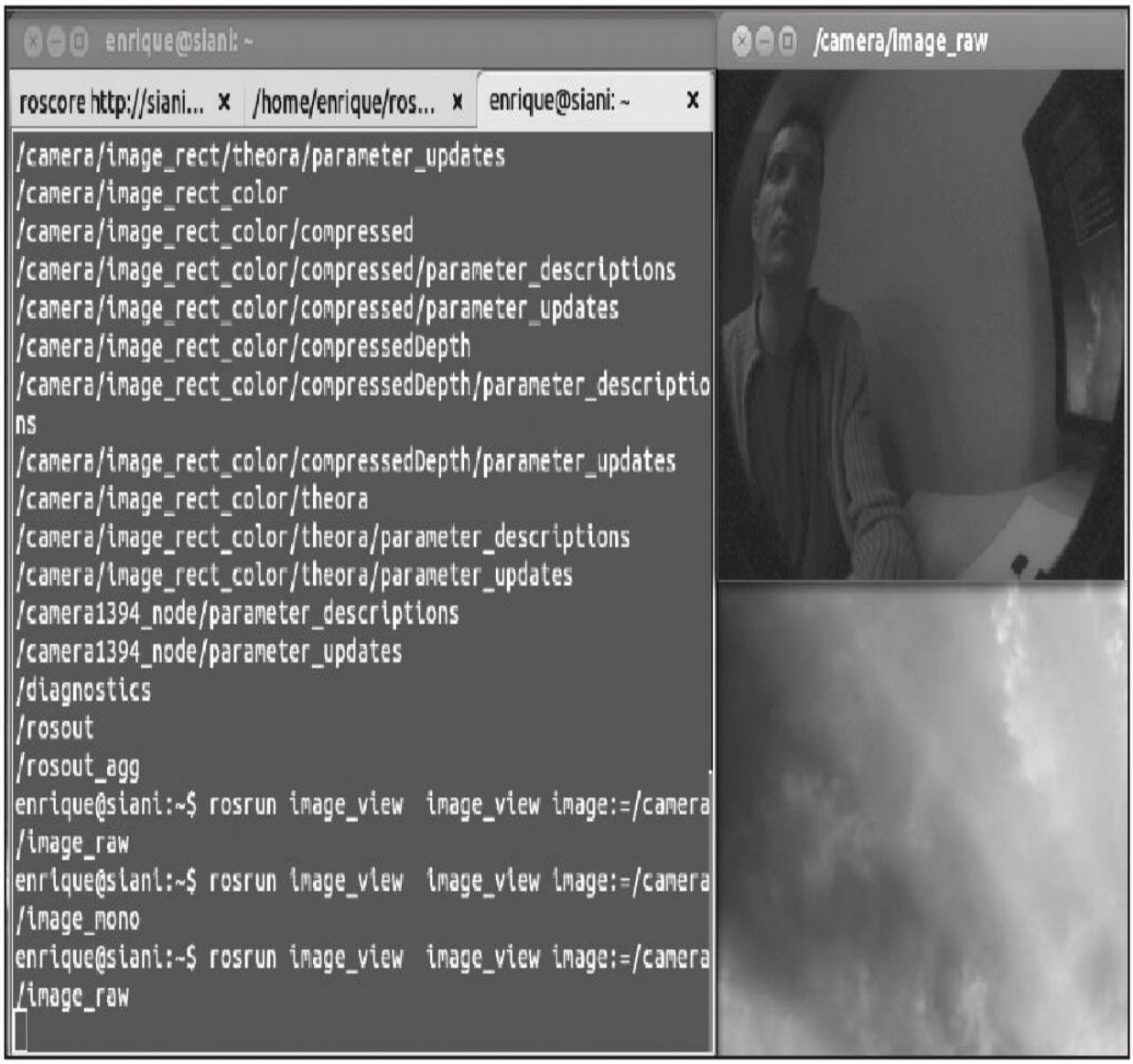
```
camera_info_url_left:
package://chapter5_tutorials/calibration/camera_stereo/
${NAME}.yaml
camera_info_url_right:
package://chapter5_tutorials/calibration/camera_stereo/
${NAME}.yaml
```

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、失真模型系数以及透视与投影矩阵，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。

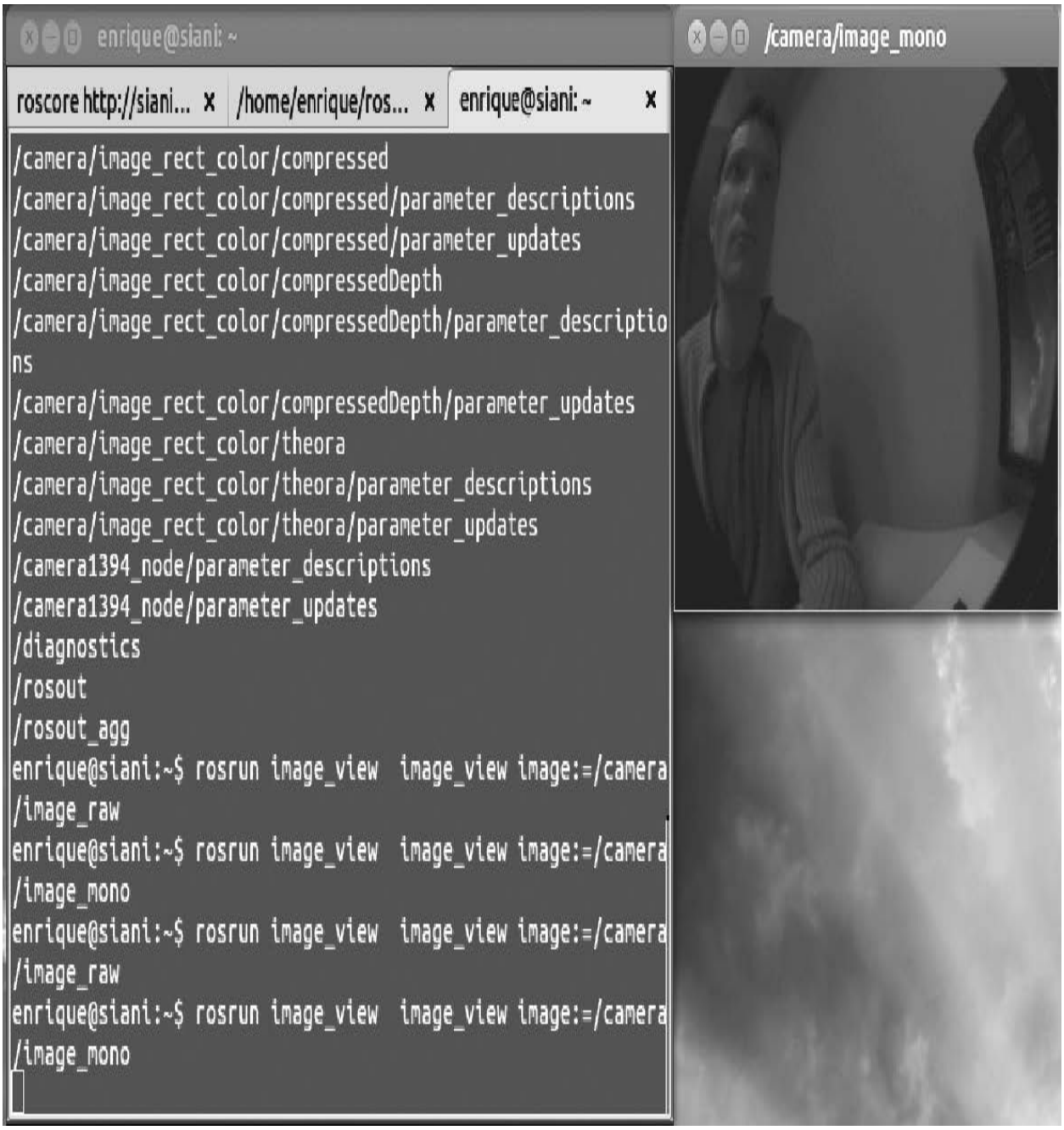
9.6 ROS图像管道

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系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关于摄像头的更多主题。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：

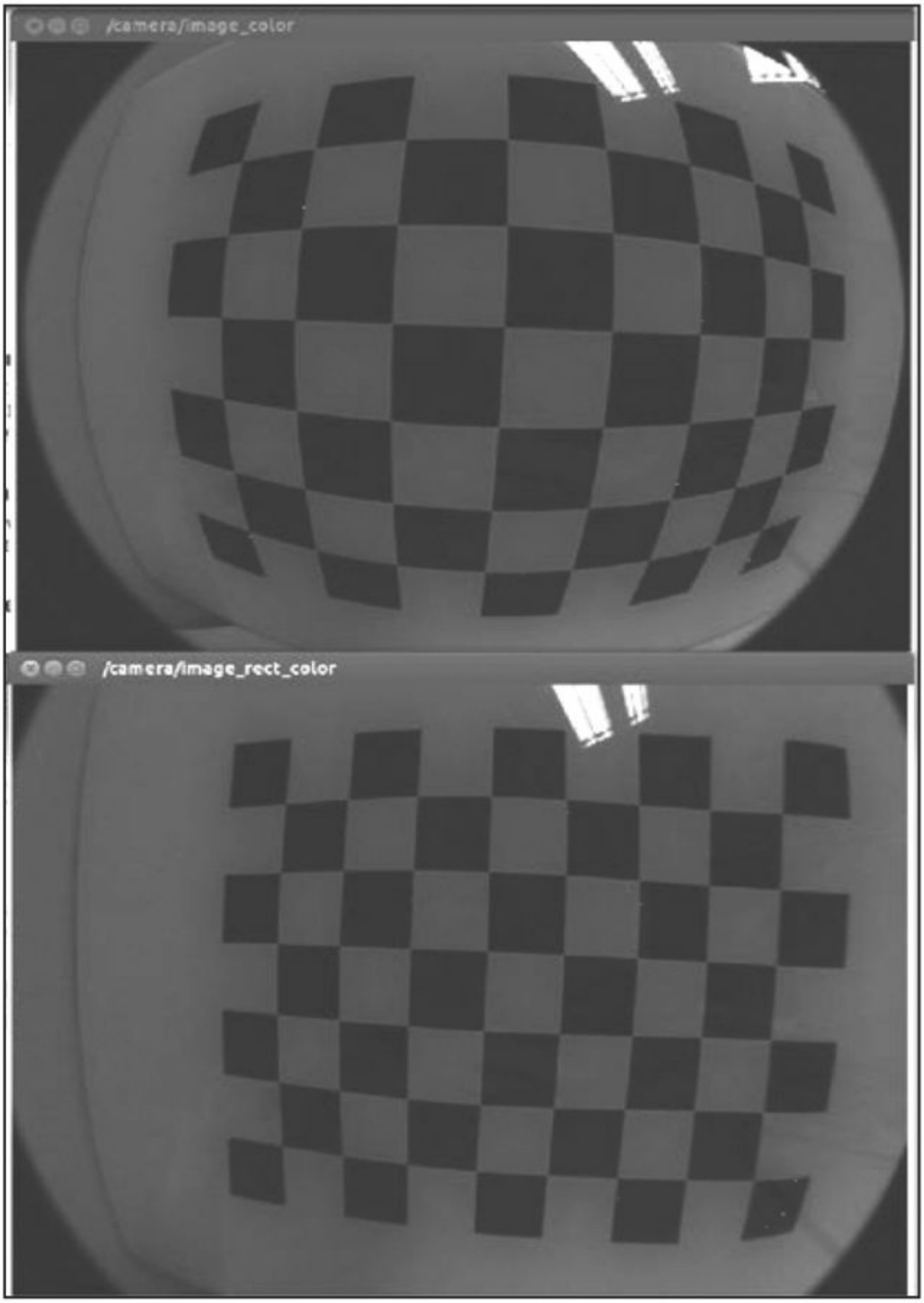


在这种情况下，单色等同于以下原始图像。请注意，对于性能良好的相机，原始图像通常对相机传感器上的单元采用拜耳模式编码，因此它不同于单色。



最后，如果相机实际上支持颜色，显示以下彩色图像，它与单色不同，否则它也将会是单色图像：

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，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：


/camera/image_color
/camera/image_rect_color

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：

```
$ roslaunch chapter5_tutorials camera.launch view:=true
```

它也能用来查看其他的主题。但是对于一些摄像头，RAW图像已经是彩色的了。如果要查看修正后的图像，也可以用image_view或者rqt_image_view查看image_rect_color主题，或者修改launch文件。通过image_proc节点可以激活这些主题，只需要在launch文件中加入以下代码：

```
<node ns="$(arg camera)" pkg="image_proc" type="image_proc"
name="image_proc"/>
```

用于双目摄像头的图像管道

对于双目摄像头，左右摄像头都是相同的。然而，也有一些专门针对它们开发的可视化工具，这样我们就能够使用左右图像来计算和查看视差图像。其主要算法使用双目视觉标定和左右图像纹理估计每个像素的深度，也就是视差图像。为了获得优秀的计算结果，必须调整计算这种图像的算法。在下图中我们看到左、右和视差图像，也就是使用stereo_image_proc的rqt_reconfiguire，在其中节点为双目图像建立了图像管道。在launch文件中只需要：

```
<node ns="$(arg camera)" pkg="stereo_image_proc"
      type="stereo_image_proc"
      name="stereo_image_proc" output="screen">
  <rosparam file="$(argparams_disparity)"/>
</node>
```

如下图所示，它需要能够设定rqt_reconfigure的视差参数，并与rosparam dump/stereo/stereo_image_proc一起保存：

我们有一套调整好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。另外，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，但这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，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 ReprojectionRMS Error: 1.278
Pixels
Linearity RMS Error: 1.542 Pixels ReprojectionRMS Error: 1.368
Pixels
Linearity RMS Error: 1.437 Pixels ReprojectionRMS Error: 1.112
Pixels
Linearity RMS Error: 1.455 Pixels ReprojectionRMS Error: 1.035
Pixels
Linearity RMS Error: 2.210 Pixels ReprojectionRMS Error: 1.584
Pixels
Linearity RMS Error: 2.604 Pixels ReprojectionRMS Error: 2.286
Pixels
Linearity RMS Error: 0.611 Pixels ReprojectionRMS Error: 0.349
Pixels
```

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，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。但在本例中，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。

9.7 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（包括安装和使用的所有步骤）。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。只需要按照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，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，它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，还有一个更值得探索的工具就是RoboEarth（http://www.roboearth.org）。它允许我们对物体进行探测和3D建模，并将其存储在一个全球性的数据库

中。这个数据库允许全世界所有机器人（或人）访问。存储模型可以是2D或3D的，它可以用于识别相似的物体或它们的视点，也就是说，识别摄像头/机器人看到的物体。RoboEarth（机器人地球）计划已经集成在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/roboearth）。当然，最新版本的ROS还不正式支持该计划。

·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。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。我们还介绍如何安装它们，目前需要从源代码中完成，因为这些软件包不正式支持在ROS Kinetic中使用。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。

9.7.1 视觉里程计

视觉里程计（Visual odometry）是指使用视觉估计移动机器人或传感器相对位移的算法的名称。在环境中机器人或传感器相对于初始位姿不断积累里程数据可以得到全局的位姿估计，但是需要注意的是，位姿估计会累积漂移，因为每个相对位移的误差都在不断地积累并且无边界增长。为了解决这个问题，需要一个全局定位（global localization）或闭环检测（loop closure detection）算法。这是视觉SLAM系统的组件之一，但它也需要视觉里程计来创建位姿估计的参考估计。

9.7.2 使用viso2实现视觉里程计

为了使用viso2，需要切换到catkin工作空间（~/dev/catkin_ws）并使用如下命令：

```
$ cdsrc
$ wstoolinit
$ wstool set viso2 --git git://github.com/srv/viso2.git
$ wstool update
```

现在通过下面的命令进行编译：

```
$ cd ..
$ catkin_make
```

一旦编译完成，需要使用下面的命令设置环境变量：

```
$ sourcedevel/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。双目摄像头驱动程序已经安装好，后面的小节会介绍它是如何工作的。

9.7.3 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，用于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，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caliration_tf_publisher.launch文件：

```
<include file="$(find
camera_pose_calibration)/blocks/calibration_tf_publisher.launch">
  <arg name="cache_file"
  value="/tmp/camera_pose_calibration_cache.bag"/>
</include>
```

现在，运行以下命令：

```
$ roslaunch 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它：

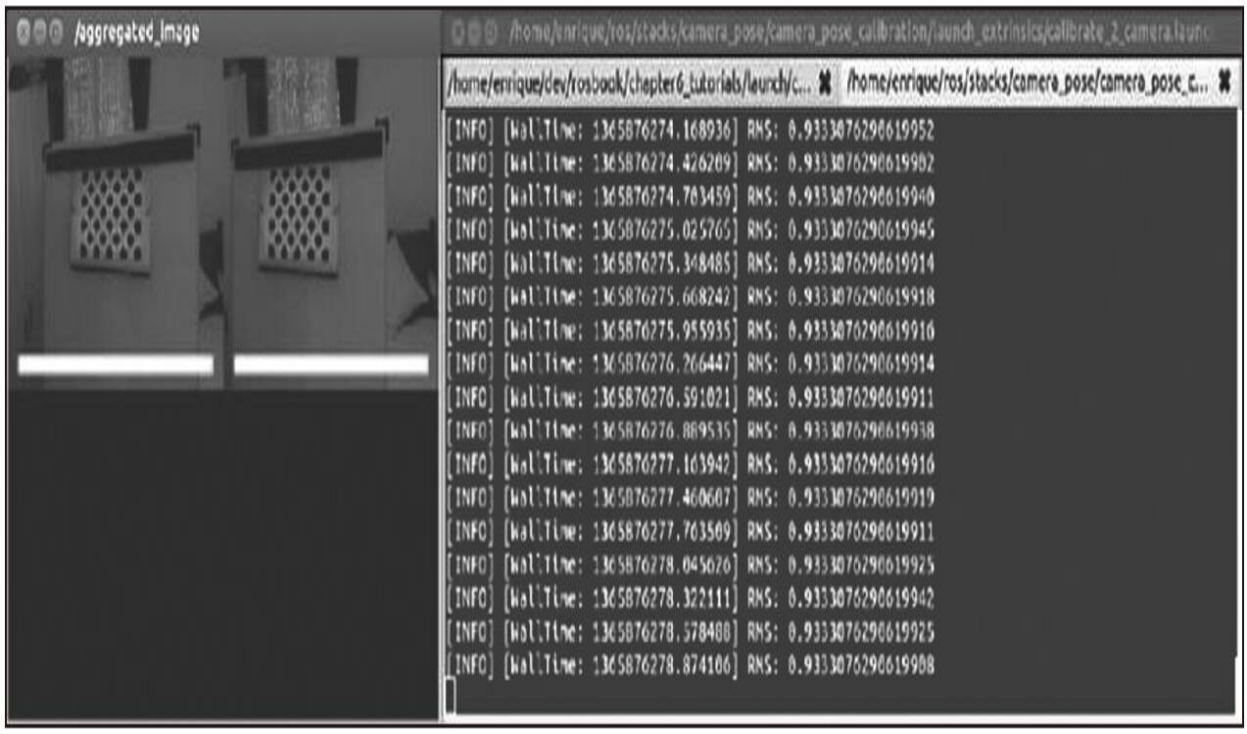
```
$ roslaunch camera_pose_calibration calibrate_2_ camera.launch camera1_
ns:=/stereo/left camera2_ns:=/stereo/right checker_rows:=6 checker_
cols:=8 checker_size:=0.03
```

通过这个命令，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，

在界面中，一些指示条才会从左到右完全变绿。

由于摄像头本身就有很多噪声，因此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。

然后就能进行标定，如下图所示。



同时，这也会创建从左摄像头到右摄像头的tf。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对于双目摄像头更简单实用的方法。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，它使用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="$(arg
  camera)_optical_link"
  args="$(argoptical_translation) $(argoptical_rotation) /$(arg
  camera) /$(arg camera)_optical 100"/>
</launch>
```

这个文件包含在双目摄像头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.launchtf:=true
```

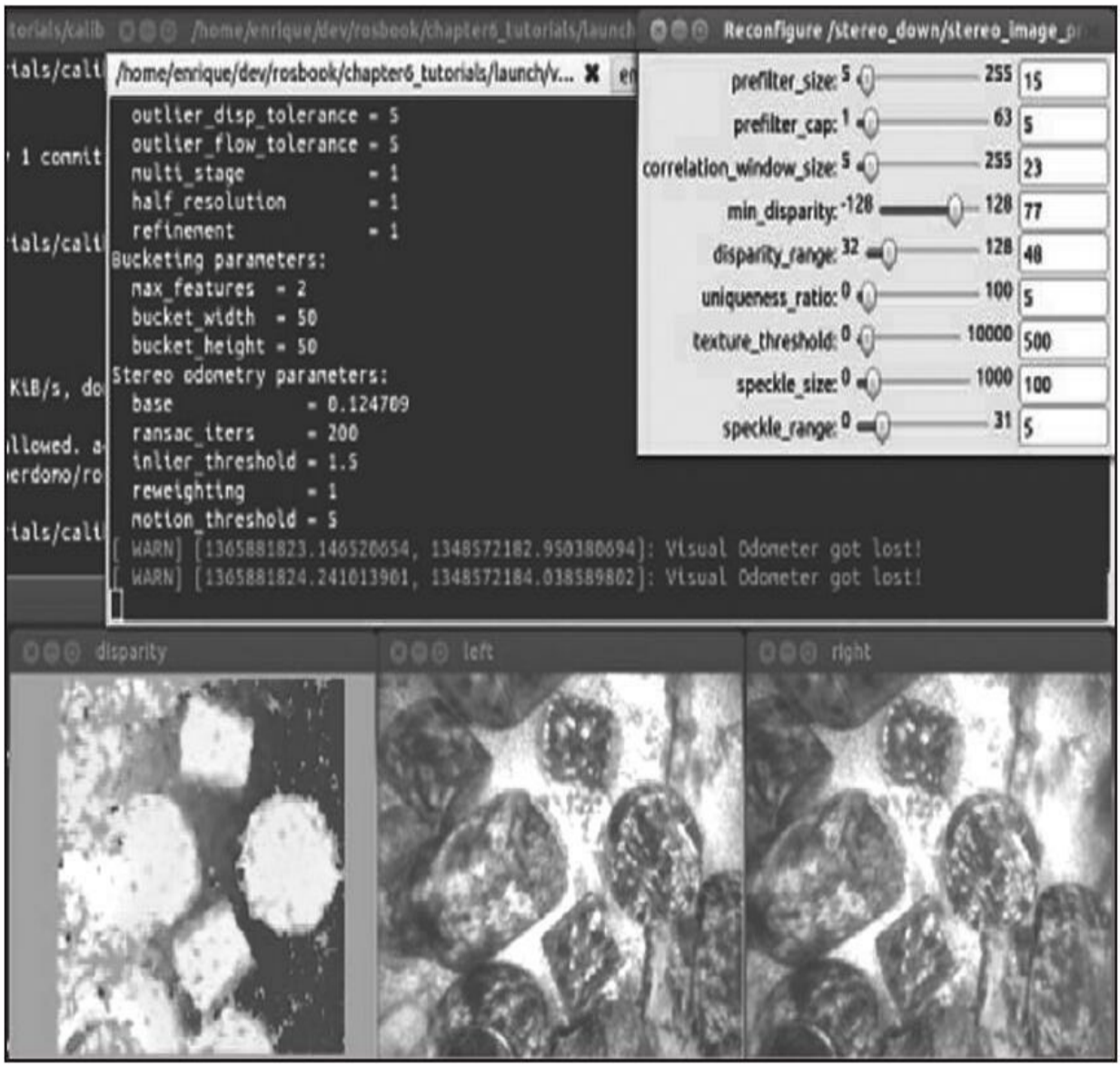
然后你就能够使用rqt_rviz查看tf数据是否发布成功。我们将会在下接下来的viso2在线演示中介绍。也可以使用rqt_tf_tree列出它们的树状结构（见第3章）。

9.7.4 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，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，以及一个rqt_reconfigure界面用来配置视差算法。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：



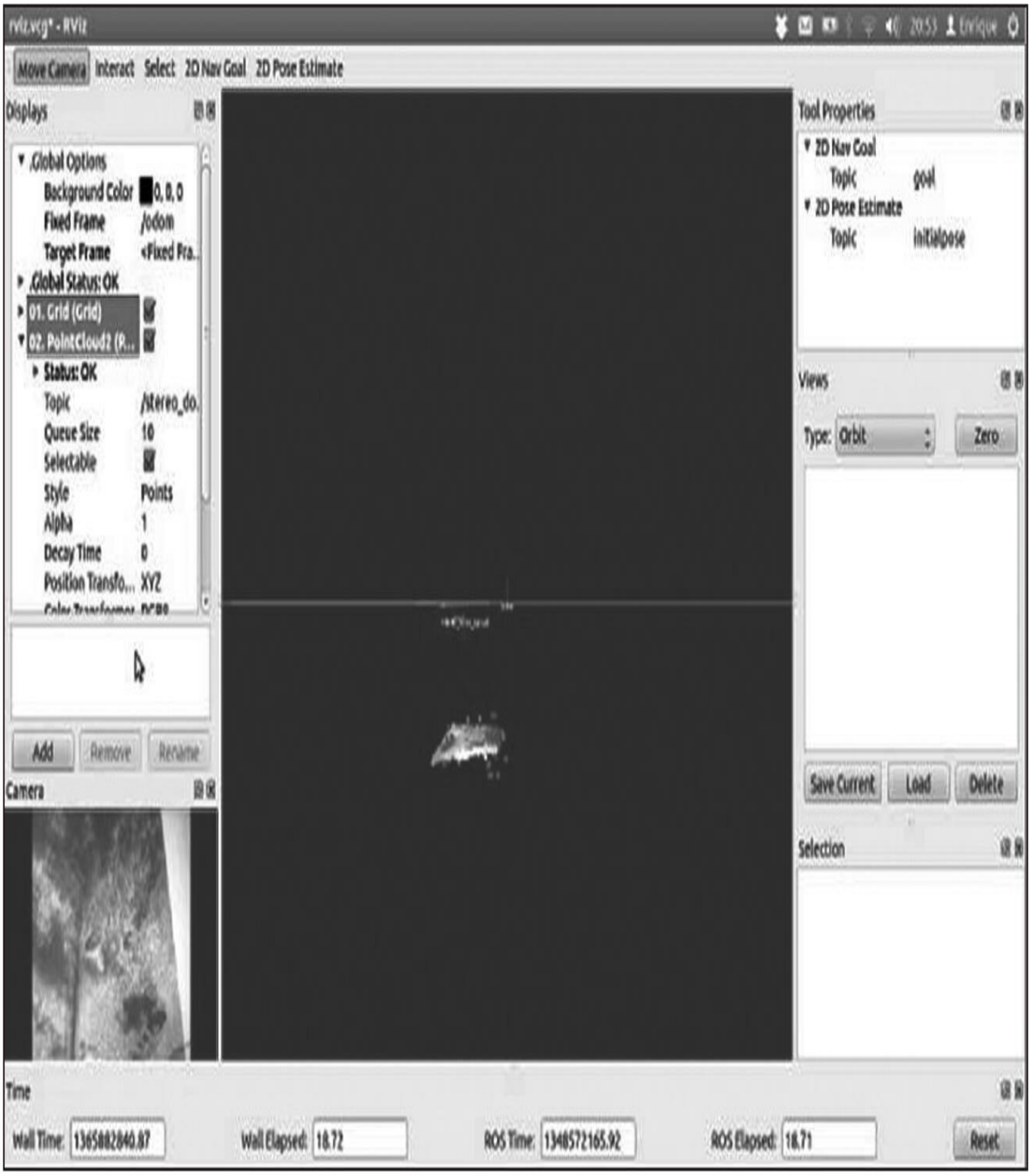
运行以下命令来启动双目里程计并在rqt_rviz中查看结果：

```
$ roslaunch chapter5_tutorials viso2_demo.launch odometry:=true
rviz:=true
```

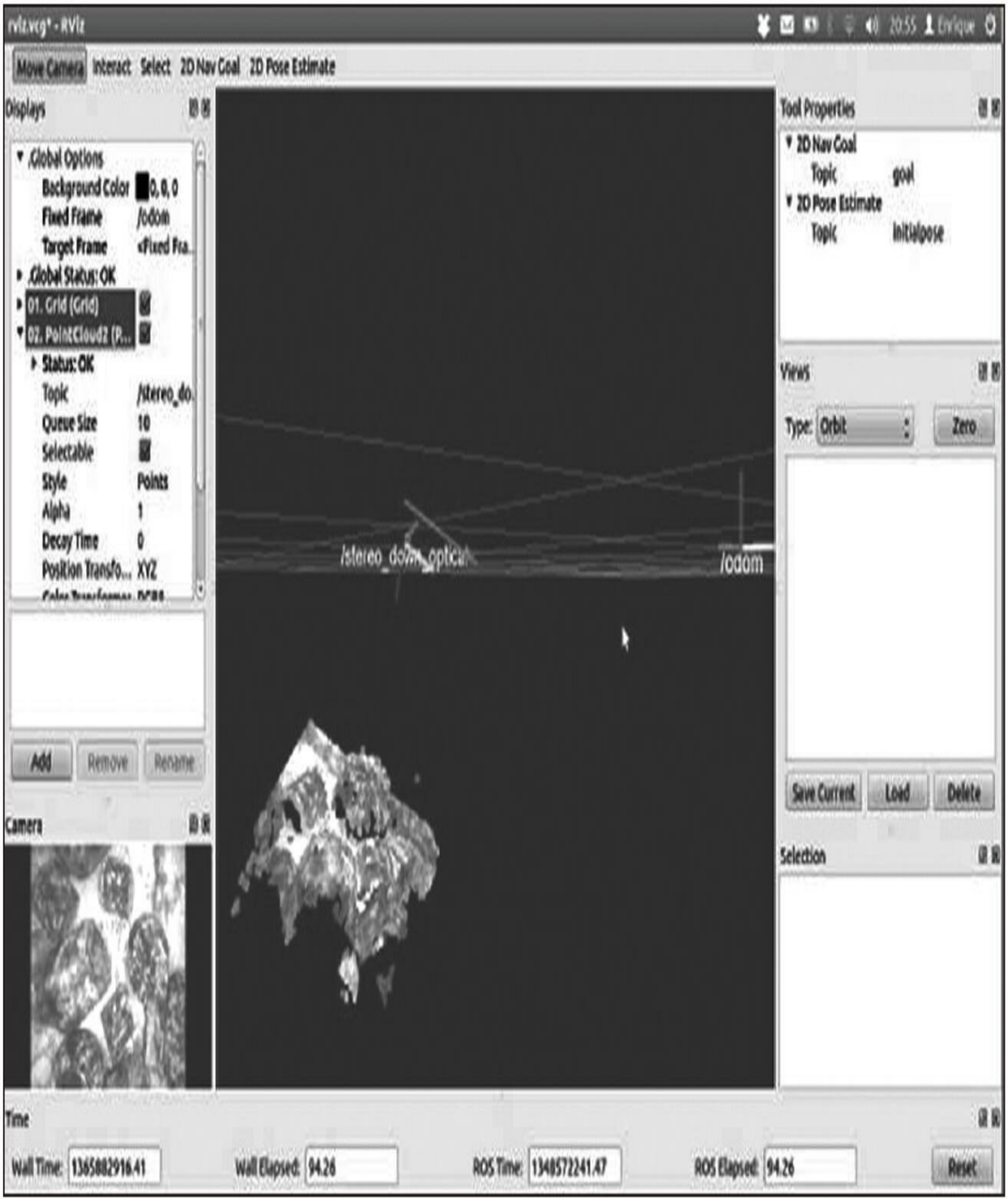
请注意，在config/viso2_demo/rviz.cfg中为rqt_rviz提供了足够的配置。

该配置文件——launch文件会自动加载。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，还显示了在双目视觉里程计中用

来进行摄像头位姿估计的/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。

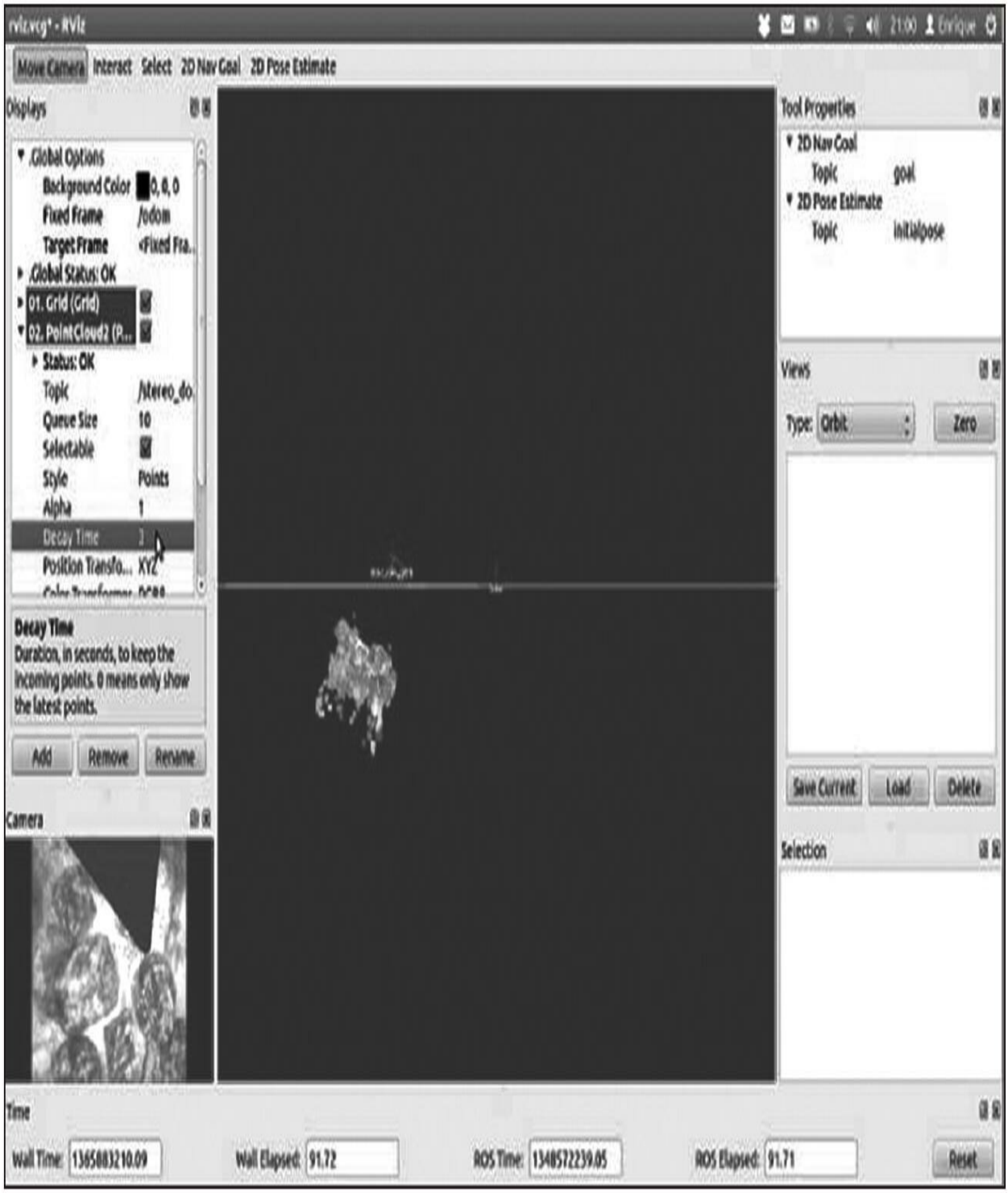


随着新帧的出现，该算法能够创建一个3D重建，如下面的截图所示，我们可以看到海底上岩石的高度：



如果设置三秒钟的衰减时间，则连续帧中的不同点云将一起显示，

因此可以看到海底的地图。记住，由于视觉里程计算法的漂移，地图将包含一些错误：

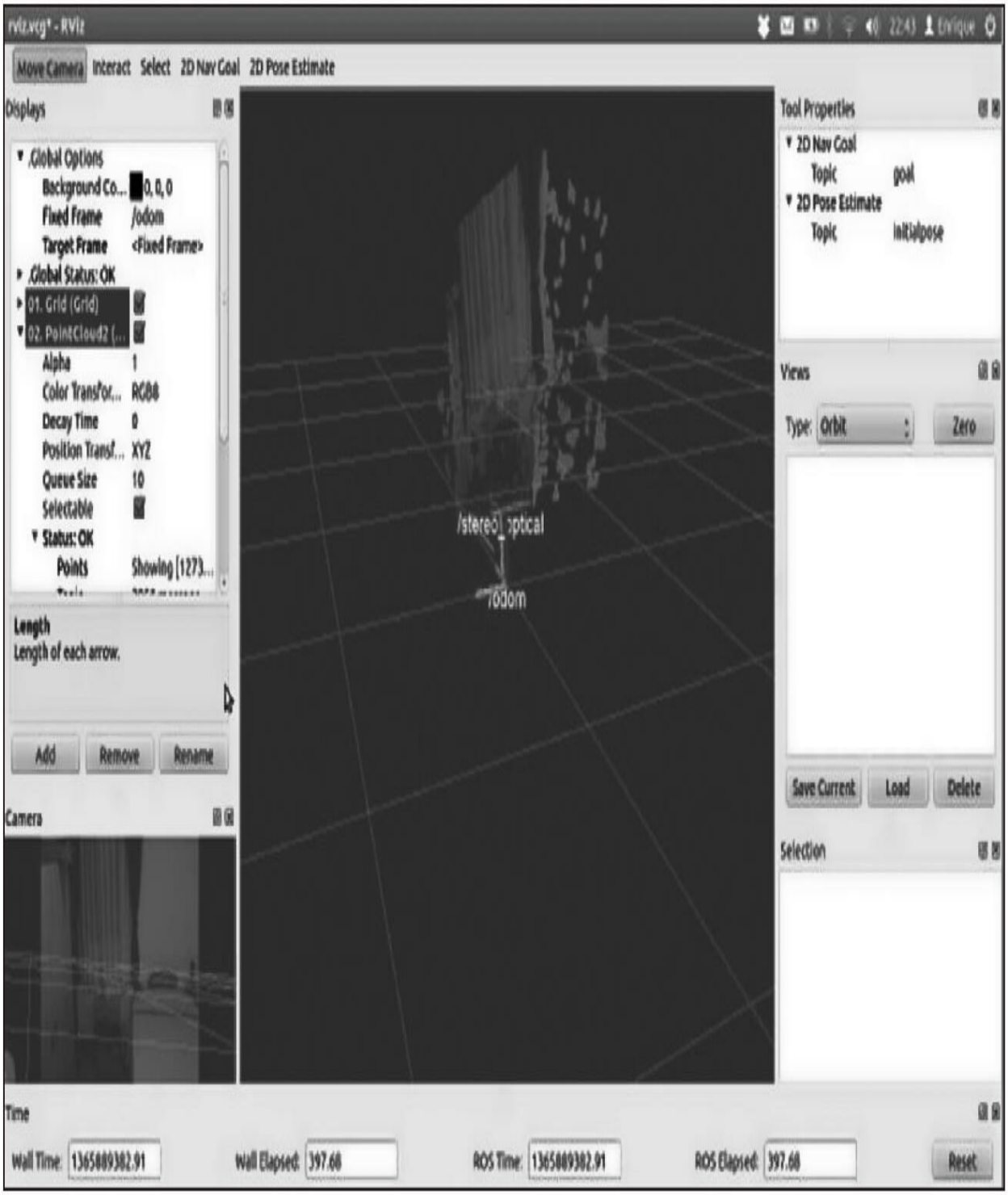


9.7.5 使用低成本双目摄像头运行viso2

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消息。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。也可以试着调一下视差算法的参数。


rviz.vcg* - RViz
Move Camera
Interact
Select
2D Nav Goal
2D Pose Estimate
Displays
Global Options
Background Co...
0, 0, 0
Fixed Frame
/odom
Target Frame
<Fixed Frame>
Global Status: OK
01. Grid (Grid)
02. PointCloud2 (...
Alpha
1
Color Transfor...
RGB8
Decay Time
0
Position Transf...
XYZ
Queue Size
10
Selectable
Status: OK
Points
Showing [1273...
Length
Length of each arrow.
Add
Remove
Rename
Camera
/stereo_optical
/odom
Tool Properties
2D Nav Goal
Topic
goal
2D Pose Estimate
Topic
initialpose
Views
Type:
Orbit
Zero
Save Current
Load
Delete
Selection
Time
Wall Time:
1365089302.91
Wall Elapsed:
397.68
ROS Time:
1365089302.91
ROS Elapsed:
397.68
Reset

9.8 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。

9.8.1 安装fovis

由于fovis没有提供Debian功能包，因此需要在catkin工作空间中输入下面的命令（使用与chapter5_tutorials相同的工作空间）：

```
$ cdsrc
$ git clone https://github.com/srv/libfovis.git
$ git clone https://github.com/srv/fovis.git
$ cd ..
$ catkin_make
```

这会复制两个软件库，使我们能够在ROS中集成fovis软件。注意，原始代码托管在Google代码项目（http://fovis.github.io/）上。

一旦编译成功，在使用软件前先使用下面的命令设置环境：

```
$ sourcedevel/setup.bash
```

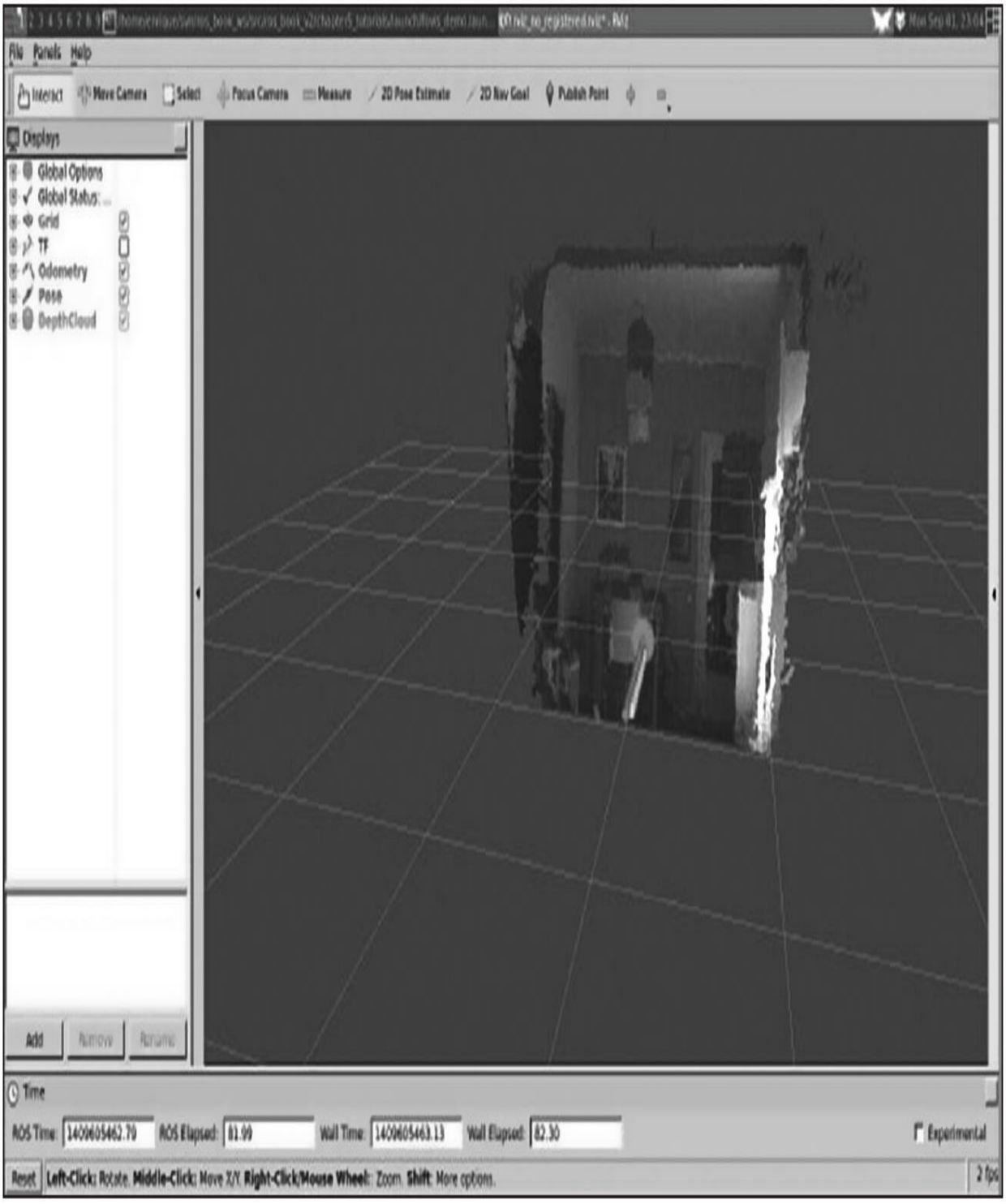
9.8.2 用Kinect RGBD深度摄像头运行fovis

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。

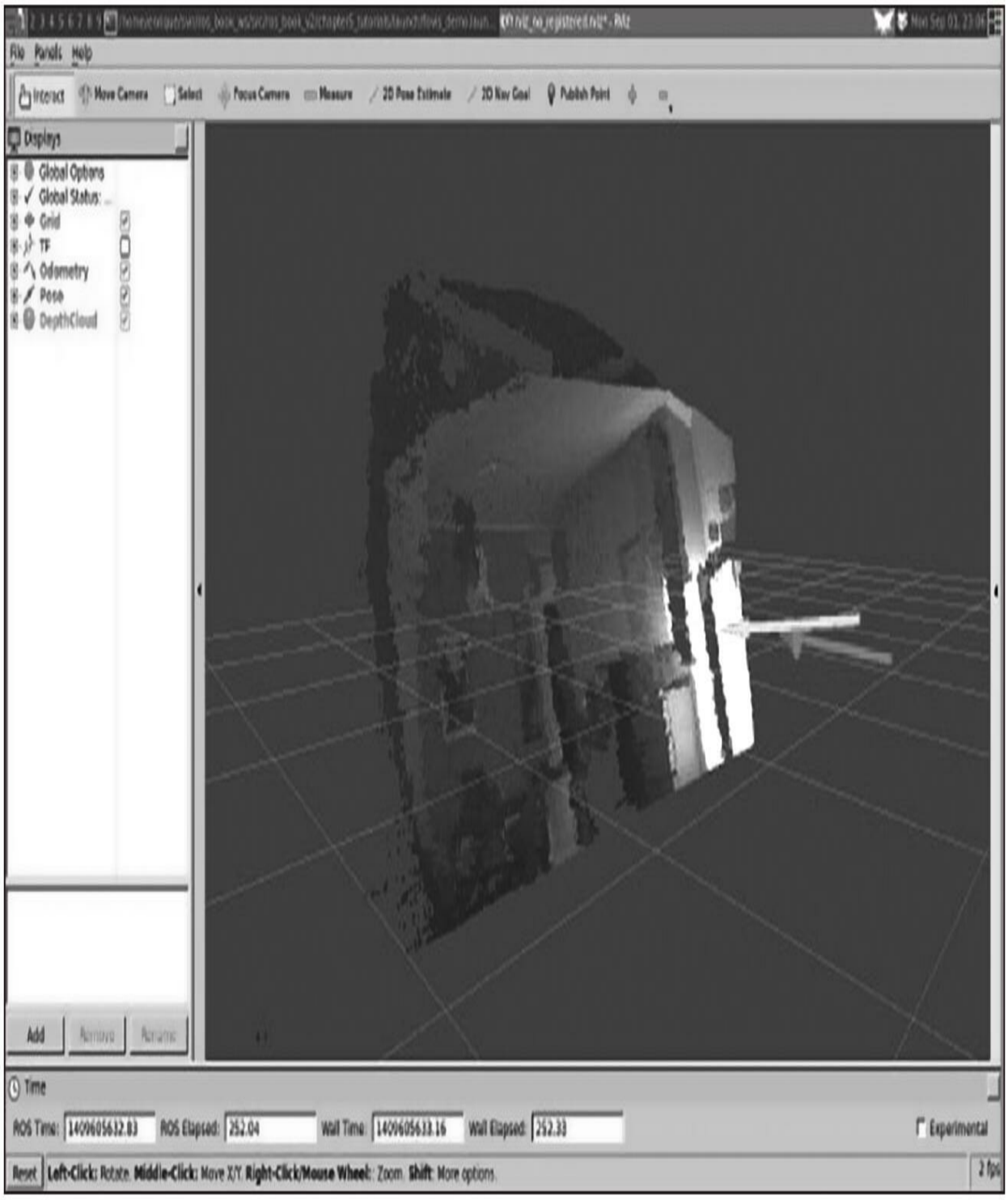
只需要启动Kinect RGBD摄像头驱动程序和fovis。为了方便起见，提供一个launch文件同时运行两者。

```
$ 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、两个指示里程的箭头和摄像头的当前位置：



随着摄像头移动，你可以看到箭头指示摄像头位姿（如上图所示）。因为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，但这会依据计算资源使原来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。

尽管如此，仍然可以在RGB摄像头上使用throttle来使用注册的版本。这可由所提供的launch文件自动运行。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，可以尝试软件配准模式（sw_registered），调节RGB摄像头消息速率在2.5Hz，可以在fovis_sw_registered.launch进行修改，如下所示：

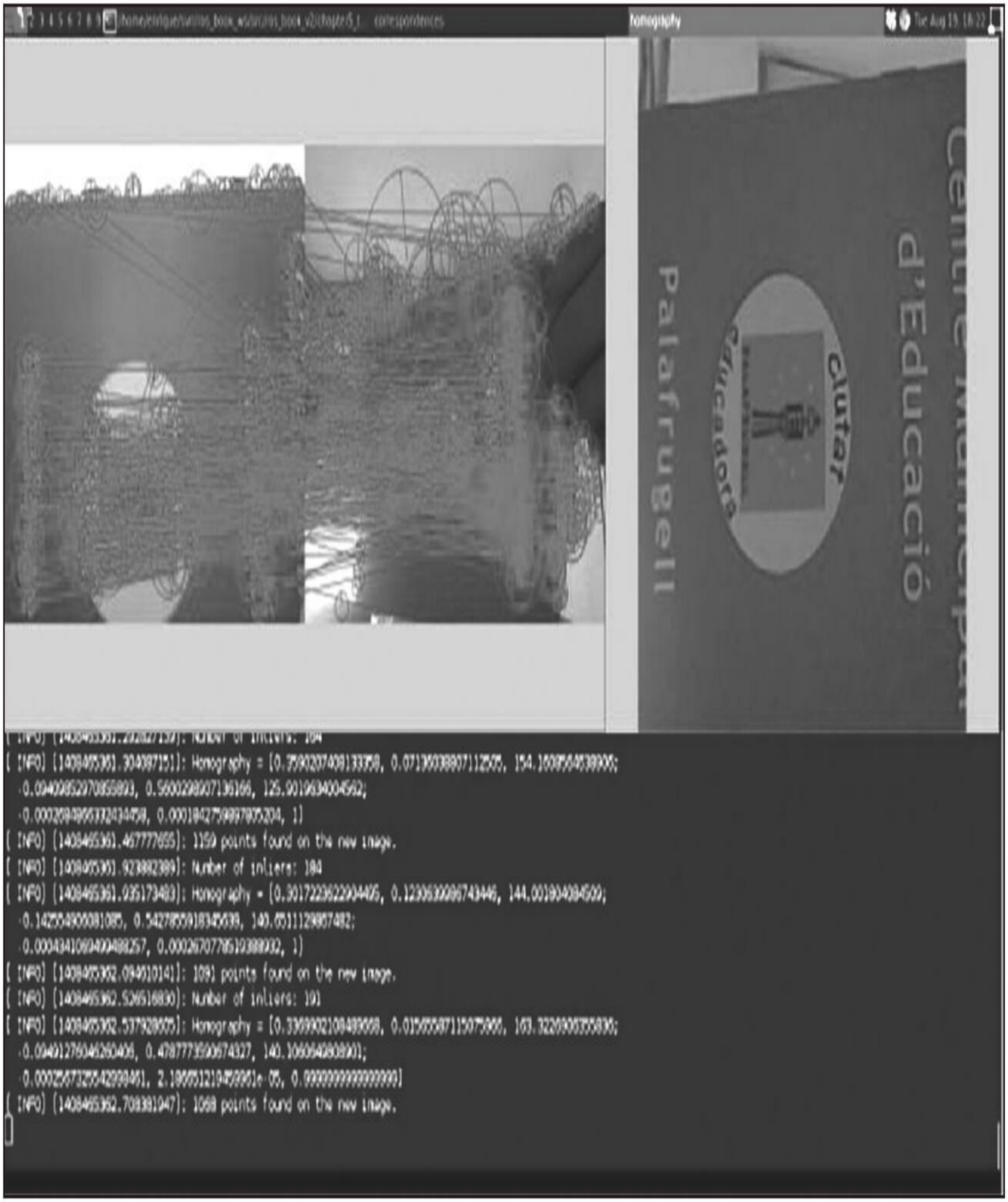
```
$ roslaunch chapter5_tutorials fovis_demo.launch mode:=sw_registered
```

9.9 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（webcam）启动并采集帧，检测特征（默认SURF），提取每一个的描述符，使用基于Flann的交叉确认过滤器（cross-check filter）对比从第一帧图像中提取的匹配信息。一旦程序匹配，计算单应性矩阵H。使用H，可以将新一帧的图像变形至原始帧，如下图所示（顶部是匹配点和通过H变形后的图像，终端中是相应的纯文本）：



Homography
correspondences
d'Educació
Palafrugell
ciutat
educadora
[INFO] [1408465361.304087151]: Homography = [0.359020740913358, 0.07136038807112505, 154.1608564538906;
-0.0940985297085893, 0.5600298007136166, 125.9019634004562;
-0.000268496573243458, 0.0001842759897805204, 1]
[INFO] [1408465361.467777655]: 1159 points found on the new image.
[INFO] [1408465361.923882389]: Number of inliers: 184
[INFO] [1408465361.935173483]: Homography = [0.3017223622904405, 0.1230630086743446, 144.001804084509;
-0.1425548060081085, 0.5427855918345638, 140.6511129867482;
-0.000434106940488257, 0.0002670778519388932, 1]
[INFO] [1408465362.094610141]: 1091 points found on the new image.
[INFO] [1408465362.526516830]: Number of inliers: 191
[INFO] [1408465362.537928605]: Homography = [0.3369902108489668, 0.01565587115075966, 163.3226908355836;
-0.09491276046260406, 0.4787773590674327, 140.1060640808901;
-0.000256732554299461, 2.186551219459961e-05, 0.9999999999999999]
[INFO] [1408465362.708381947]: 1068 points found on the new image.

9.10 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摄像头驱动程序示例。

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中与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双目摄像头的示例。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，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10章　点云

机器人学的工具中，点云（point　cloud）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（Point Cloud Library，PCL）提供了大量数据类型和数据结构，不但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可视化程序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。

10.1 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，它是存储所有点的容器。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建自己的算法。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。

10.1.1 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，并且定义了云所要存储的点类型。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；它只存储了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，它非常有用。还有与此相似的其他两种标准的点数据类型：一是pcl::InterestPoint，它有一个字段存储强度（strength）；二是pcl::PointWithRange；它有一个字段用来存储距离（视点到采样点），而不是亮度或强度。

·pcl::PointXYZRGBA：这种点类型存储3D信息，也存储颜色（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PointXYZRGB-Normal和PointXYZINormal，顾名思义，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用已经可用的类型，理论上还可以使用用户定义的语法正确的类型。

10.1.2 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以及常用的默认值的机制。

为了解决这个问题，PCL的开发者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类层次结构。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，重用这些算法，并且它允许用户通过存取器轻松地为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);
```

只有在库中有要求时才遵守这个方法，所以可能有例外，比如I/O功能就不受同样要求的约束。

10.1.3 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。这个消息用来传递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：这个消息类型存储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（即顶点和多边形）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，用于描述一个多边形。等价的PCL类型是pcl::Vertices。

·pcl_msgs::ModelCoefficients：这个消息类型存储了一个模型的不同系数，例如描述一个平面需要的4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必须用消息类型替代。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 semantics）。

10.2 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软件包：

```
$ catkin_create_pkg chapter10_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 chapter10_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并且正确编译的节点：

```
#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<pcl::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中，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和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})
```

完成最后一步后，功能包可以通过在工作空间的根目录中调用catkin_make来编译。

10.2.1 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伪随机点组成的PCL点云。这个PCL点云最终会通过一个称为/pcl_output的主题定期发布出去。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chapter10_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，这个消息类型用来存储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，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，所以高度仅为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，所以下一步和最后一步是将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可视化程序中可视化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：

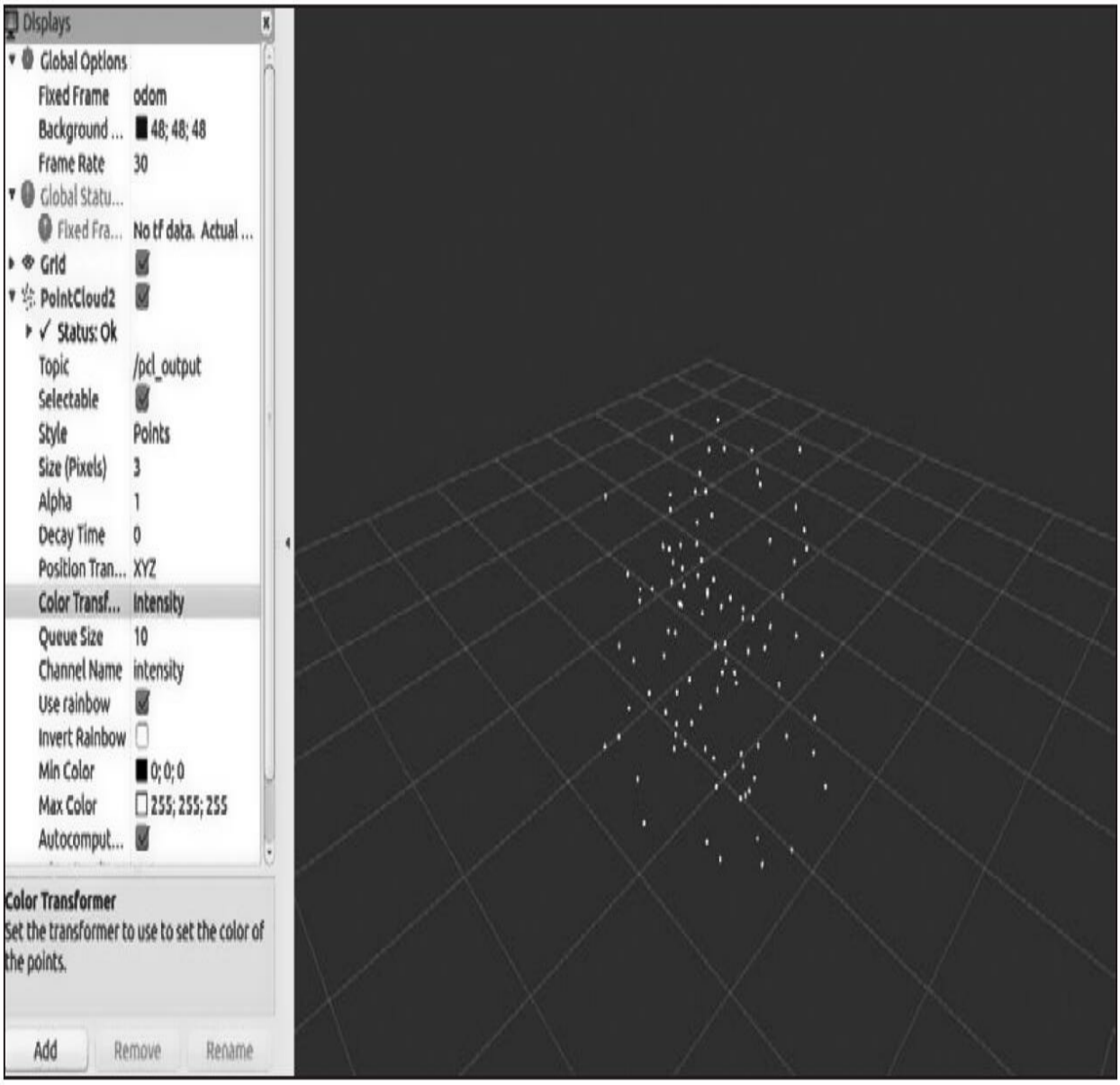
```
rosrun chapter10_tutorials pcl_create
```

为了可视化点云，必须用下面的命令来运行RViz可视化程序：

```
$ rosrun rviz rviz
```

一旦rviz加载好，就可以通过单击Add按钮并添加pcl_output主题来

增加一个PointCloud2对象。读者必须确保在Global Options部分中将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，如下图所示。



10.2.2 加载和保存点云到硬盘中

PCL提供了一些标准的文件格式去加载和存储点云到硬盘中，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的点列表。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chapter10_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，它包含了加载和存储点云到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，用随机点人工填充了点云，在这里则是从硬盘中加载它们：

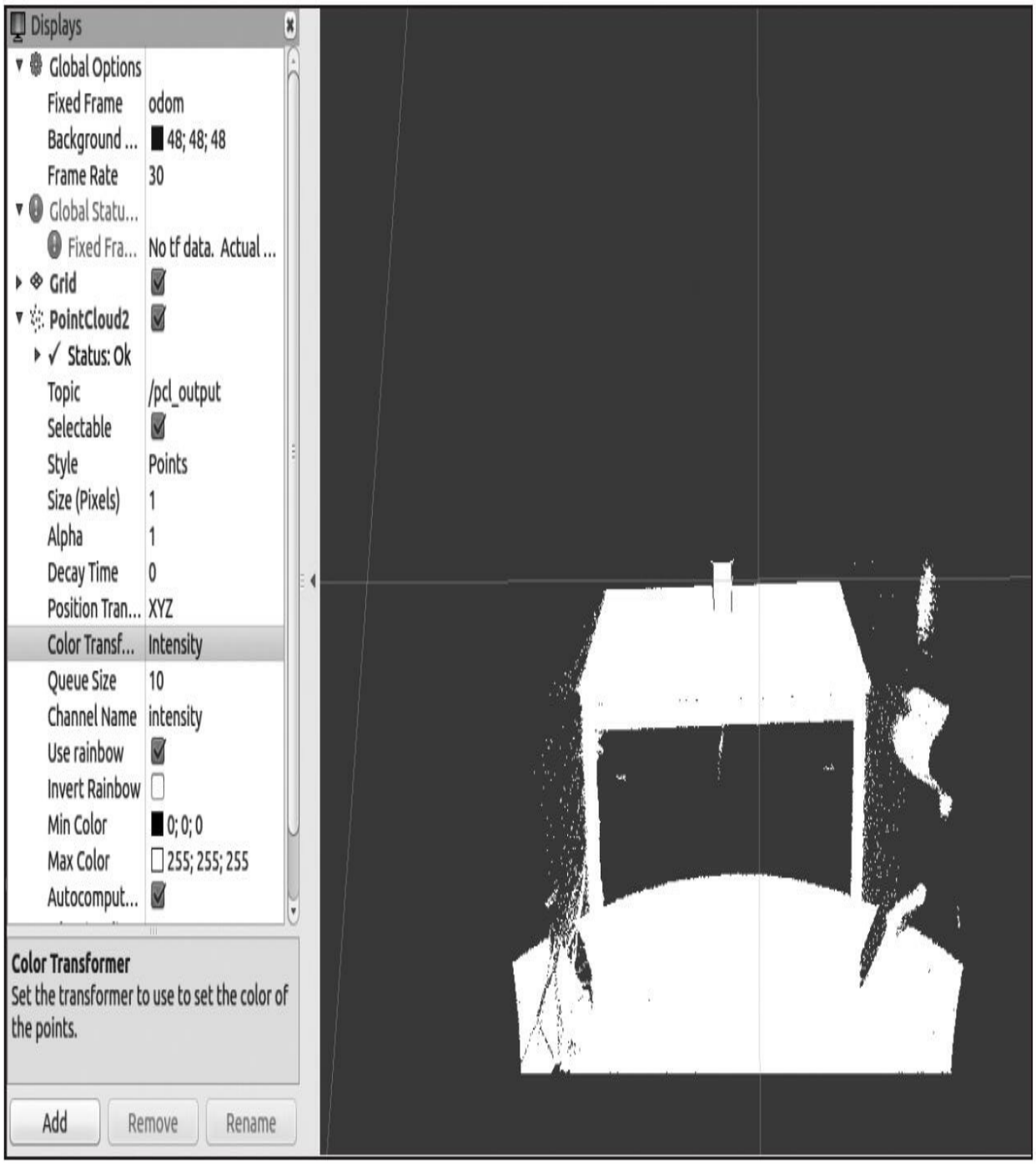
```
pcl::io::loadPCDFile ("test_pcd.pcd", cloud);
```

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 chapter10_tutorials/data

$ rosrun chapter10_tutorials pcl_read
```

如同前一示例，通过RViz可视化程序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，目标是订阅一个sensor_msgs/PointCloud2主题，并且将接收的点云存储到一个文件中。这个代码可以在chapter10_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，所以它们可以连接在一起测试：

```
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格式存储到硬盘上，但是它也可以另存为二进制格式，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，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，分别运行roscore、pcl_read节点和pcl_write节点：

```
$ roscore

$ roscd chapter10_tutorials/data && rosrun chapter10_tutorials pcl_read

$ roscd chapter10_tutorials/data && rosrun chapter10_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chapter10_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。

10.2.3 可视化点云

PCL提供了几种方法来可视化点云。第一种并且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chapter10_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，对于初始化列表中遗漏的对象可以自动调用它。在所有的事情都正确地初始化后，用一个与窗口名字相对应的非常简单的字符串来隐式初始化云句柄。订阅者将设置接收pcl_output主题，并且设置一个定时器，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，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，运行pcl_read示例和数据源，比如一个提示，使

用下面的命令：

```
$ roscd chapter10_tutorials/data
$ rosrun chapter10_tutorials pcl_read
```

最后，在第三个终端中，运行下面的命令：

```
$ rosrun chapter10_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，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，也包括很多视区和颜色空间。在chapter10_tutorials的源代码文件夹中提供了一个称为pcl_visualize2.cpp的文件，它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功能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；结合shift键，可以平移图

像，结合control键，可以旋转图像。最后，只要按下H键就可以在当前终端打印帮助文档，如下面的截图所示。

```
$ rosrun chapter6_tutorials pcl_visualize
| 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
```

10.2.4 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。本节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chapter10_tutorials功能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点云，这样就可以将两个示例连接到一起，以便在后续的算法中产生更好的结果。这个示例可以在chapter10_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_downsampled",
        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;
}
```

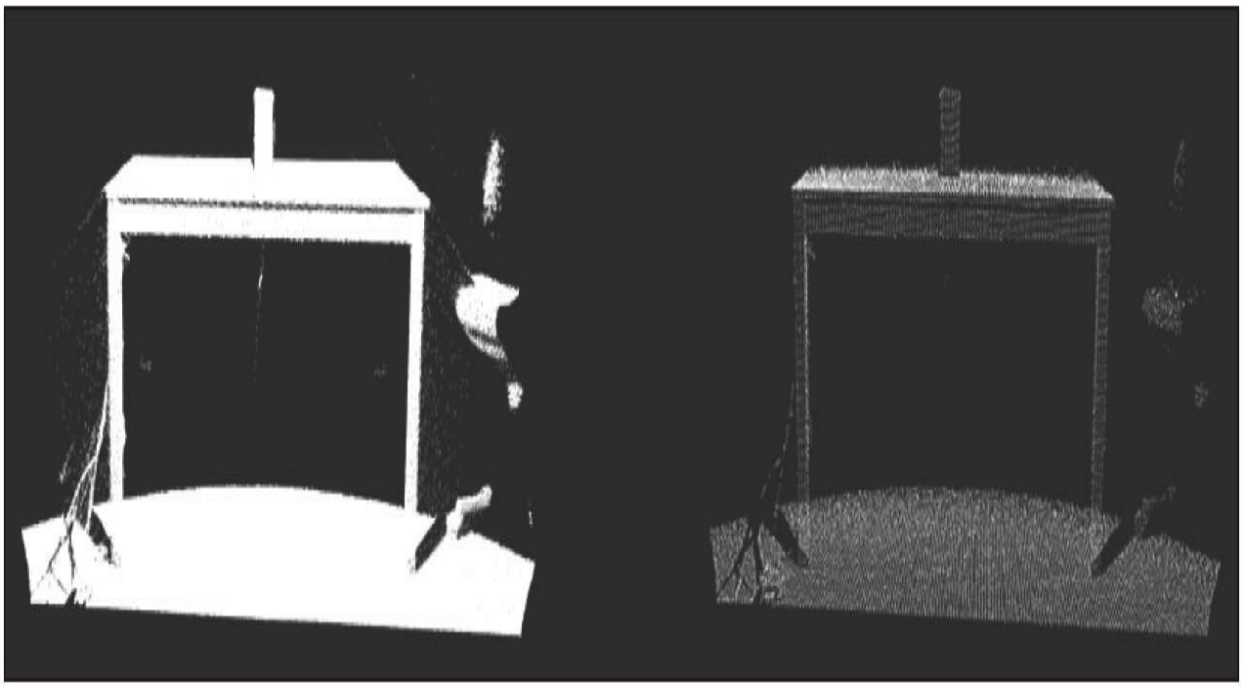
这个示例与前面的一样，不同的是主题的订阅者和发布者，在这个

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
```

在第二个终端中，运行pcl_read示例和一个数据源：

```
$ roscd chapter10_tutorials/data
$ rosrun chapter10_tutorials pcl_read
```

在第三个终端中，运行滤波的示例，它会产生pcl_filtered图像供缩减采样的示例使用：

```
$ rosrun chapter10_tutorials pcl_filter
```

最后，在第四个终端中，运行缩减采样的示例：

```
$ rosrun chapter10_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，但是你也许需要调整订阅的主题。

10.2.5 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位置和另一个点云的开始位置一样简单。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，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，ICP）的算法来执行配准和匹配。我们会在下面的示例中使用这个算法，这个示例可以在chapter10_tutorials功能包的源文件夹中找到，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值小于特定的阈值。这个阈值可以通过setTrans-formationEpsilon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），它可以通过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：

```
pcl::toROSMsg(cloud_aligned, output);
pcl_pub.publish(output);
```

为了运行这个示例，需要参照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 chapter10_tutorials/data
$ rosrun chapter10_tutorials pcl_read
```

在第三个终端中，运行滤波示例：

```
$ rosrun chapter10_tutorials pcl_filter
```

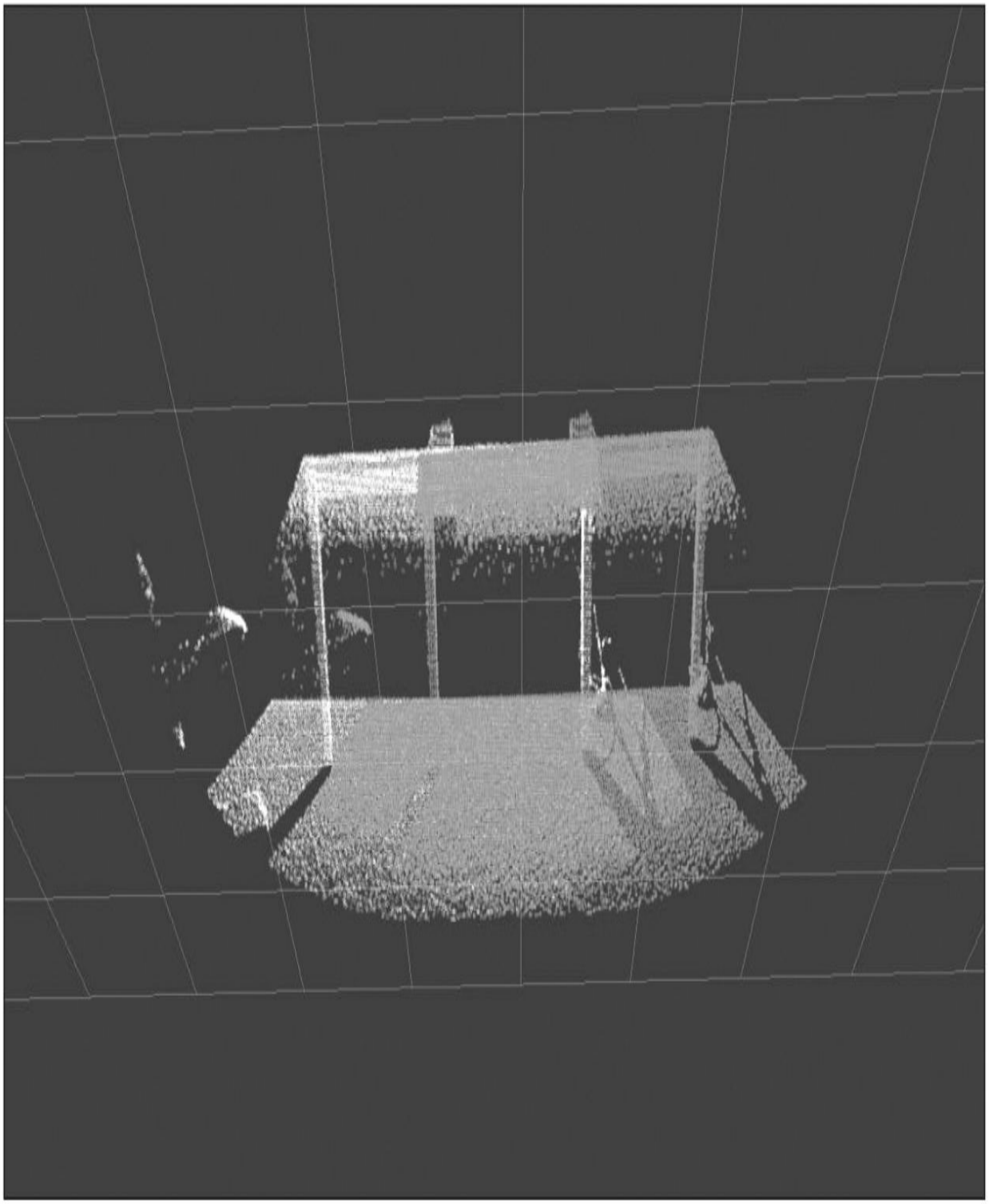
在第四个终端中，运行缩减采样的示例：

```
$ rosrun chapter10_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_Downsampled主题：

```
$ rosrun chapter10_tutorials pcl_matching
```

可以从rviz中获取的以下图像中查看最后的结果。在界面中，蓝色的点云是从PCD文件中获取的原始点云，而白色的点云是从ICP算法中获取的对应点云。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。



10.2.6　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存储数据，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，八叉树大致上表示在一个树结构中每个节点有8个子节点，并且子节点可以用来分割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chapter10_tutorials功能包的源文件夹中找到，它的文件名为pcl_partitioning.cpp：

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主题上。这个处理程序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。该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，可以创建一个新点云，其中只包含属于这个分区的点：

```
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，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 chapter10_tutorials/data
$ rosrun chapter10_tutorials pcl_read
```

在第三个终端中，运行滤波示例：

```
$ rosrun chapter10_tutorials pcl_filter
```

在第四个终端中，运行缩减采样的示例：

```
$ rosrun chapter10_tutorials pcl_downsampling
```

最后，运行这个示例：

```
$ rosrun chapter10_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，所以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。

10.3 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，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，或者一个球体，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。

在本节的示例中，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chapter10_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>("planar_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。顾名思义，Model-Coefficients消息存储了一个数学模型的系数，PointIndices存储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消息滤波器来实现：

```
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RANSAC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，转换通过标准函数执行，但是你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的回调函数：

```
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

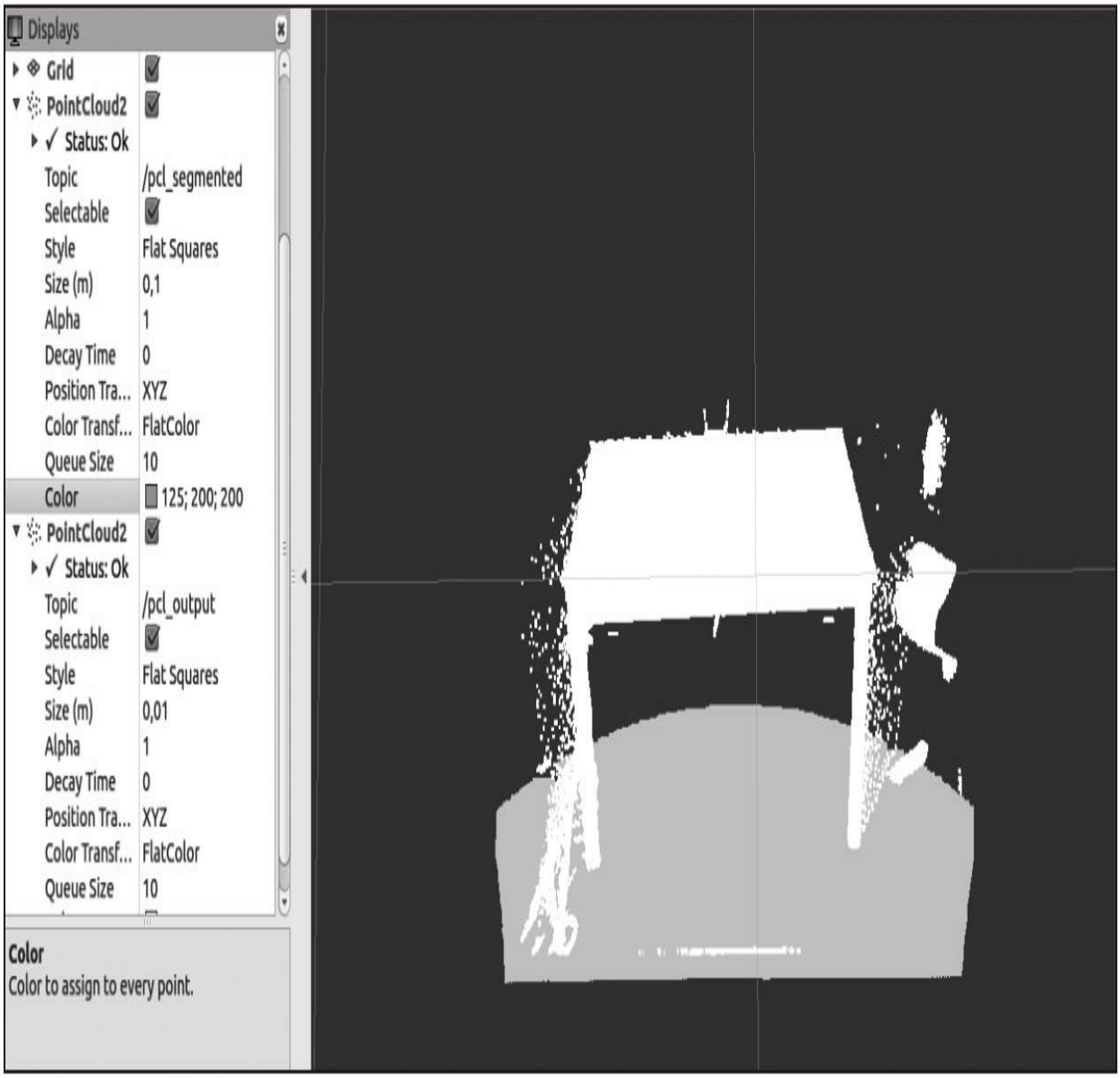
的点放到一个新的点云中来轻松实现：

```
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。在界面中，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。


Displays
Grid
PointCloud2
Status: Ok
Topic /pcl_segmented
Selectable
Style Flat Squares
Size (m) 0,1
Alpha 1
Decay Time 0
Position Tra... XYZ
Color Transf... FlatColor
Queue Size 10
Color 125; 200; 200
PointCloud2
Status: Ok
Topic /pcl_output
Selectable
Style Flat Squares
Size (m) 0,01
Alpha 1
Decay Time 0
Position Tra... XYZ
Color Transf... FlatColor
Queue Size 10
Color
Color to assign to every point.

10.4 本章小结

本章探讨了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。不管如何，考虑到点云处理的计算成本，任何架构设计都不可避免地与所使用系统的计算能力相关。

这里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。

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。

Table of Contents