



◎ 机器人领域的权威指南，涵盖使用ROS进行整体机器人设计、碰撞、传感器和定位的
高级技术，工具和编程技巧
适合进阶学习者或进行相关领域的研究、无经验的初学者

[index]



精通ROS机器人编程

(第2版)

Mastering ROS for Robotics Programming
Second Edition

〔美〕伦纳特·约瑟夫 (Lenert Joseph) 编

〔美〕乔纳森·卡多纳 (Jonathan Cardona) 等译
高群平 郭志杰 等译



【PACKT】

ROS进阶实践指南，通过使用ROS进行复杂的机器人设计、建模、仿真和实测的一本机器人设计与制作系列实用技术、工具和编程技巧。

使用先进的开源机器人平台进行书中大部分示例，完成任何种类的项目。



精通ROS机器人编程

(第2版)

Mastering ROS for Robotics Programming
Second Edition

【译】 莱恩·约瑟夫 (Lentin Joseph) /
【译】 约纳森·卡卡切 (Jonathan Cacace) / 编
张鹏宇 张志杰 等译

机械工业出版社
China Machine Press

版权信息

书名：精通ROS机器人编程(原书第2版)
作者：（印）郎坦·约瑟夫（Lentin Joseph）
排版：小暑暑
出版社：机械工业出版社
出版时间：2019-02-01
ISBN：9787111621997

— · 版权所有 侵权必究 · —

译者序

背景

未来20年，中国的科技将逐步从模仿、跟随、追赶，过渡到引领世界，成为创新引擎。构建科技核心竞争力的唯一之路是人才的培养。

正是意识到了机器人相关人才的重要性和紧迫性，我们从2015年开始举办全国“机器人操作系统（ROS）及其应用暑期学校”。2015年有近200人参与，现场非常火爆，很多人坐在地板上听，甚至有人站在门口听。2016年的ROS暑期学校，网上有500人报名，现场有100多人报名。加上2017年300人左右的规模，2018年400多人的规模，四年时间有千余人参加过我们的ROS暑期学校。除此之外，易科机器人实验室、ROS小课堂、泡泡机器人、ROS星火计划、古月居的在线课程至少影响了4000~5000人。这样的趋势如果能够保持下去，再过5年，就会形成3万~5万人的学习团体。现在机器人产业的窗口逐渐打开，随着产业的发展、需求的推动、资本的介入，将形成强大的牵引力。现在有不少机器人公司，其中一些公司一开始只有2~3人，逐渐发展成5~6人，然后20~30人，如果他们能够坚持下去，也有可能发展成100~200人的规模。如果300~500人中有一位杰出代表愿意带领大家，3万~5万人中就有100位这样的代表。在国家层面的推动下，通过建立一批机器人学院和人工智能学院，15年后，将形成10万左右的机器人、人工智能工程师队伍，另外还有50万左右相关学科领域的工程师辅助。如果了解信息产业的发展过程，就会比较清楚，现在基本上每

个大学都有计算机学院、信息学院或软件学院，这样的过程也会发生在机器人、人工智能方向上。

从地理位置的角度来讲，以上海为中心的长江三角洲和以深圳、香港、广东、珠海、东莞、中山等城市构建的大湾区将是未来机器人、人工智能领域的科技创新中心。江浙沪一带有很多经济发展很好的城市，而且聚集了很多有名的大大学。最主要的是这些大学的毕业生很多都会留下来，聚集在这里；行业内的企业也会聚集在这里。

ROS正是影响整个机器人产业的灵魂，很多人还没有做好准备，还没有意识到是怎么回事，就不得不与ROS牵连在一起，卷入一场巨大的洪流中。

学习ROS的10大理由

过去几年里，ROS在工业上的应用取得非常好的发展势头。这一方面得益于工业机器人领域长期的积累，另一方面得益于ROS内在的优势。若干年后，ROS毫无疑问将成为机器人工工程师的必备技能，机器人领域的从业人员没有人能回避ROS。下面列出学习ROS的10大理由。

1) ROS是开源免费的

ROS于2007年诞生于斯坦福大学人工智能实验室，2008年至2013年期间由柳树车库研发，并逐渐形成开源生态系统。ROS使用BSD许可证，可以免费用于研究和商业项目，任何人或者任何组织都可以对ROS的源码进行修改，并且可以选择开源，也可以选择不开源。

2) 大量算法在ROS中已经实现

ROS的最大魅力在于其集成了许多先进的机器人算法，同时为非常多的工具提供了接口。ROS中基于物理的仿真环境Gazebo可以模拟机器人运动学、动力学分析以及环境中的很多物理特性。又如，ROS可以与计算机视觉算法OpenCV进行通信。另外，点云库PCL实现了大量点云相关的通用算法和高效数据结构，包括点云获取、滤波、分

割、配准、检索、特征提取、识别、追踪、曲面重建、可视化等。运动规划库MoveIt!可以提供有效的规划算法，用于多种机器人的运动规划。MoveIt!模块也已经被整合到了ROS框架中。

3) 学习使用ROS的人越来越多

ROS提供松散耦合的基础通信中间件、业界先进的算法库、统一标准的软件接口、完善的教程、庞大的社区等，使ROS的学习对入门者越来越友好。而对于ROS开发者而言，ROS提供的业内领先的资源可以让他们轻松地搭建起自己的机器人系统，极大地缩减了开发周期。对研究人员而言，ROS提供的仿真环境可以用来方便地验证新算法，而与ROS相配套的硬件系统则可以轻松地在现实情况中验证算法是否有效，从而不必纠结于软硬件平台的搭建。基于这些特点，ROS自诞生之日起就开始迅速发展。在世界各地开展了形式多样的ROS学习活动，包括我们前面提过的连续四年举办的全国“机器人操作系统(ROS)及其应用暑期学校”。

4) ROS已成为机器人系统基础软件的标准

随着机器人应用浪潮的掀起，越来越多的人力和物力投入到了这片看似美妙的汪洋大海中。与之伴随的是，机器人应用软件的蓬勃发展。与传统工业机器人厂商不同，ROS自诞生之初就尝试建立机器人系统基础软件的标准，而不是在各自的自留地内封闭式发展。这背后的深层原因是，解决机器人软件问题是一个极其庞大而复杂的工程，应该把精力放在解决新的问题上，而不是重复地造基础软件的轮子。经过多年的发展，ROS也实现了其创造之初被寄予的期望，已然成为广泛使用的机器人系统基础软件的标准，在学术界得到了广泛使用，在工业界也开始崭露头角。

5) ROS有较好的易用性，而且越来越好

ROS从一开始连基本的维基页面都不完善，到现在拥有大量的教程和完善的多语言维基，一步一步地提高易用性。

但是ROS的开发者们没有止步于此，在经历了ROS版本的巨大成功之后，OSRF团队开始进行ROS 2的研究。OSRF重新编写了核心系统，主要目标是改善在不同组件之间交换信息的分布式计算机系统。ROS 2将会更好地支持小型嵌入式设备，支持多机器人协同工作并增加系统实时性。

6) 模块化设计，节点独立性，使协作开发非常方便

ROS使用分布式处理框架。采用若干类型的通信，包括基于服务的同步RPC（远程过程调用）通信、基于话题的异步数据流通信，还有参数服务器上的数据存储。这些特点使可执行文件能单独设计，并且在运行时松散耦合。程序可以封装到软件包和软件包集中，以便共享和分发。ROS还支持代码库的协同，这使得协作亦能被分发。

7) 大量硬件设备、传感器、驱动器和机器人平台支持ROS

目前ROS支持上百种机器人平台，包括著名的PR2机器人、UR5机械臂等。在这些机器人平台上，使用ROS可以方便地进行机器人开发。

同时，ROS可以很好地支持各种类型的传感器，包括各种摄像头（如Kinect深度摄像头、单目摄像头、双目摄像头等）和雷达（如激光雷达、超声波雷达等）。由于ROS是基于Linux的，所以在大部分Linux的平台上也可以运行ROS，如ROS可以在ARM核心板BeagleBoard上运行。所以ROS提供对嵌入式平台的固有支持。

8) 你的竞争对手已经开始使用ROS

ROS的蓬勃发展，让它成为新兴的资本宠儿。截至2015年，全球约有1.5亿美元的风险投资注入与ROS相关的项目。可以看到，目前机器人创业公司大多数会将自己的原型方案架设在ROS上，并且新兴的机器人厂商也以支持ROS作为一个主要的卖点。

与此同时，一些传统的厂商也开始加入ROS的阵营。ROS Industrial是一个致力于把ROS推向工业界的ROS分支，目前已经得到

3M、ABB、BMW、Ford、Boeing、Siemens等传统强势企业的支持。

在学术界，ROS的使用更为广泛。在DARPA机器人挑战赛中有18支参赛队伍使用ROS，包括NASA的Robonaut。

9) ROS能让你快速启动机器人项目的开发

开发一个机器人项目是一个综合性问题，异常复杂。从软件层面来看，它包含了最底层的硬件抽象（各种设备驱动等）、设备控制、通信、上层算法、应用系统等。任何一个公司都不可能独立完整地解决一切问题。经过多年的努力，ROS提供了一整套软硬件解决方案，开发者们可以专注于解决特定的问题，而无须花费精力构建整个平台。只需要一台可以运行Ubuntu的电脑，就可以进行机器人开发的测试仿真。ROS支持上百种机器人平台，让实际的机器人测试也变得十分方便。

10) ROS的未来很光明

人类社会正在快速向前发展，机器人领域也一直在不断革新。作为一个开源的基础机器人软件系统，ROS毫无疑问会成为这场变革的助推剂。我们看到，越来越多的初创公司开始转向ROS，并利用ROS提供的完善的软件系统快速地搭建起机器人平台。越来越多的传统大公司也注意到了这一变化，他们的产品也开始支持ROS。

所有这一切，都预示着ROS有光明的前途。

接受翻译任务

也正是在这样的背景下，我们清楚地知道我们这些与ROS有牵连的人，正在参与一项影响人类未来20年的科技变革。2018年6月，我正筹备在深圳举办全国“机器人操作系统（ROS）及其应用暑期学校”，机械工业出版社华章公司的刘锋老师联系到我，希望我牵头翻译本书。我们了解到各家出版社已经陆续翻译出版了一些关于ROS的图书，我们对比了这些书，认为本书对高级用户学习ROS非常有帮助。于是我欣然接受了刘锋老师委托的翻译任务，并联系ROS小课堂

的创始人张志杰先生一起参与翻译。我们期待这样的图书能够引领ROS的初学者进阶到中高级学者的行列。同时，我们对以往的一些翻译做了认真的梳理和推敲，规范了一些ROS术语的中文译名，并期待这样的名称既能表达原作者的意图，也能符合中国读者的阅读和理解习惯。

在整个翻译过程中，我翻译了第1章，张志杰翻译了第2章及第9~13章，刘彬翻译了第3章和第4章，邹玉龙翻译了第5章、第6章及第14章，郭宗芝翻译了第7章、第8章及第15章。我与张志杰，及一些朋友负责审阅了所有章节。最后，由我对全书进行了修改润色，统稿整理。从这次翻译的经历，我们都体会到科学技术图书的翻译是一项巨大的挑战。当译稿终于要交付的时候，所有人都松了一口气。所有参与翻译的译者都付出了大量的时间和精力，所有这些努力都无法用金钱来衡量。面对一项伟大的科技变革，任何付出都是值得的。

我还要特别感谢家人对我翻译本书的支持。

张新宇
2018年11月

前言

机器人操作系统（Robot Operating System，ROS）本质是机器人的中间件，可帮助开发人员编写机器人应用程序，现已被机器人公司、研究机构和大学广泛采用。本书主要介绍ROS框架的高级概念，特别适合已经熟悉ROS基本概念的读者。但是，第1章还是简要介绍了ROS的基本概念，以帮助初学者快速入门。读者将了解新机器人的软件包生成、机器人建模、设计等，以及利用ROS框架对机器人进行仿真、与ROS连接。这里使用的是高级的仿真软件和ROS工具，这些工具可以辅助机器人导航、控制和传感器可视化。最后，读者将学习如何使用ROS底层控制器、小节点（nodelet）和插件等重要概念。读者可以使用普通计算机来运行本书中的大部分示例，而无须任何特殊硬件。但是，在本书的某些章节中将讨论如何使用特殊硬件将ROS与外部传感器、驱动器和I/O开发板一起使用。

本书是按照如下顺序来组织的。在介绍了ROS的基本概念之后，开始讨论如何对机器人进行建模和仿真。Gazebo和V-REP仿真器将用于控制机器人建模并与之交互，还将用于连接机器人与MoveIt!和ROS导航软件包。然后讨论ROS插件、控制器和小节点。最后，本书还会讨论如何将Matlab和Simulink软件与ROS连接。

本书的读者对象

本书为有激情的机器人开发人员或想充分利用ROS功能的研究人员而写。本书也适合已经熟悉典型机器人应用的用户，或者想要进阶学习ROS，学习如何建模、构建、控制自己机器人的用户。如果你想轻松

理解本书的所有内容，强烈建议你先掌握GNU/Linux和C++编程的基础知识。

本书主要内容

第1章介绍ROS的核心基本概念。

第2章解释如何使用ROS软件包。

第3章讨论两类机器人的设计：一类是七自由度（7-DOF）机械臂，另一类是差速驱动机器人。

第4章讨论7-DOF机械臂、差速轮式机器人和ROS控制器的仿真，这些控制器有助于控制Gazebo中的机器人关节。

第5章介绍如何使用V-REP仿真器和vrep_plugin将ROS连接到仿真场景中。然后讨论7-DOF机械臂和差速移动机器人的控制。

第6章介绍如何使用ROS MoveIt!和导航软件包集的现有功能，如机器人机械臂运动控制和机器人自主导航。

第7章展示ROS中的一些高级概念，例如ROS pluginlib、小节点和Gazebo插件。本章还将讨论每个概念的功能和应用，并通过一个示例介绍其工作原理。

第8章展示如何为类似于PR2的机器人编写基本的ROS控制器。在创建控制器后，我们将使用Gazebo中的PR2来运行控制器。与此同时，我们还将看到如何为RViz创建插件。

第9章讨论一些硬件组件（如传感器和执行器）与ROS的接口。我们将看到如何使用I/O开发板（如Arduino、树莓派和Odroid-XU4）与ROS的传感器通信。

第10章讨论如何使用ROS连接各种视觉传感器，并利用开源计算机视觉库（OpenCV）和点云库（PCL）与AR Markers一起对其进行编程。

第11章介绍如何构建具有差速驱动器配置的自主移动机器人，并将其与ROS连接。

第12章讨论MoveIt!的功能，例如碰撞规避，使用3D传感器进行感知、抓握、拾取和放置。在此之后，我们将学习如何将机器人控制器硬件与MoveIt!连接。

第13章讨论如何将Matlab和Simulink软件与ROS连接。

第14章帮助大家了解和安装ROS中的ROS-Industrial软件包。我们将学习如何为工业机器人开发MoveIt! IKFast插件。

第15章讨论如何在Eclipse IDE中设置ROS开发环境，并介绍ROS中的最佳实战经验与调试技巧。

如何学习本书

为了运行本书中的示例，你需要一台运行Linux系统的计算机。这里推荐安装的Linux发行版是Ubuntu 16.04，Debian 8也可以。建议计算机配置至少有4GB的内存和一款性能优异的处理器（Intel i系列）来执行Gazebo仿真和图像处理算法。

读者甚至可以在Windows系统上托管的虚拟机或VMware软件上安装Linux系统，在这样的虚拟系统中也可以工作。但这种选择的缺点是需要更多的计算资源来运行示例，并且当ROS与真实的硬件连接时，可能会遇到各种问题。

本书中使用的ROS版本是Kinetic Kame。需要的其他软件有V-REP仿真器、Git、Matlab和Simulink。

最后，某些章节要求读者将ROS与商用硬件连接，例如I/O开发板（Arduino、Odroid和树莓派）、视觉传感器（Kinect/Asus Xtion Pro）和驱动器。你必须购买这些硬件才能运行本书中的一些示例，但这并不是学习ROS必须要购买的。

下载示例代码

你可以从www.packtpub.com通过个人账号下载本书的示例代码，也可以访问华章图书官网<http://www.hzbook.com>，通过注册并登录个人账号下载。

本书的代码也在GitHub上托管，网址为<https://github.com/PacktPublishing/Mastering-ROS-for-Robotics-Programming-Second-Edition>。如果代码有更新的话，会将更新提交到GitHub代码仓库中。

除此之外，我们还在<https://github.com/PacktPublishing/>中提供了其他书籍和视频，这里还有一些其他代码，有兴趣就去看看吧！

下载彩图

我们还提供了一个PDF文件，其中包含本书中使用的所有截图和样图。可以在此处下载：

http://www.packtpub.com/sites/default/files/downloads/MasteringROSforRoboticsProgrammingSecondEdition_ColorImages.pdf。

排版约定

在本书中使用了许多排版约定。

代码格式如下：

```
<launch>
  <group ns="/">
    <param name="rosversion" command="rosversion roslaunch" />
    <param name="rosdistro" command="rosversion -d" />
    <node pkg="rosout" type="rosout" name="rosout" respawn="true"/>
  </group>
</launch>
```

命令行输入或输出格式如下：

```
$ rostopic list
```

```
$ cd
```

粗体: 表示你看到的是新术语、重要单词或句子。例如，菜单或对话框中的单词会出现在文本中。下面就是一个示例：在主工具栏上，选择“File|Open Workspace”，然后选择代表ROS工作区的目录。

 警告或重要说明标志。

 提示和技巧标志。

作者简介

朗坦·约瑟夫 (Lentin Joseph) 是一名来自印度的作家、创业者。他是印度Qbotics实验室的创始人兼CEO，在机器人领域已经有7年的从业经验，主要研究方向包括机器人操作系统 (ROS) 、OpenCV和PCL等。

朗坦已经出版了3本书《Learning Robotics using Python》《Mastering ROS for Robotics Programming》和《ROS Robotics Projects》。

目前，他在印度攻读硕士学位，并在美国卡内基 - 梅隆大学 (CMU) 机器人学院从事研究工作。

乔纳森·卡卡切 (Jonathan Cacace) 于1987年12月13日出生于意大利。在意大利那不勒斯腓特烈二世大学获计算机科学硕士学位及信息与自动化工程博士学位。目前，乔纳森是那不勒斯腓特烈二世大学PRISMA实验室的博士后，主要研究工业机器人和服务机器人，曾经开发了几款基于ROS且集成了机器人感知控制的机器人应用。

“非常感谢我的朋友、父母和所有陪我一路走来的人，是你们让我的生活变得丰富多彩。”

译者简介

张新宇，副教授，现为华东师范大学“智能机器人运动与视觉实验室”负责人。毕业于浙江大学，获本科、硕士、博士学位。博士毕业后，在韩国梨花女子大学图形学与虚拟现实研究中心从事博士后研究，后受聘为讲师和研究教授。曾受聘美国北卡罗来纳州立大学教堂山分校计算机系研究科学家，从事机器人、虚拟现实方面的研究。

2013年回国后立即着手筹建华东师范大学“智能机器人运动与视觉实验室”，将研究内容定位于机器人运动规划、计算机视觉、虚拟现实、基于物理的计算机模拟等方向。2015年年底创建华东师范大学“创客空间”，推动创客教育与创新活动。2016年年底负责筹建华东师范大学“虚拟现实与智能计算实验室”。2017年，领导成立面向华东师范大学的学生社团“机器人俱乐部”和“虚拟现实俱乐部”，推动相关技术的科普工作。2015~2018年连续四年举办全国“机器人操作系统（ROS）及其应用暑期学校”，四年间共有近两百所高校，上千名本科生、硕士、博士生，以及机器人企业的开发人员参加暑期学校的学习，授课视频在线点击超过10万次。2017年创立中国机器人操作系统（ROS）教育基金会，与国内外同行共同推进机器人操作系统在中国的宣传普及与产业应用。

张志杰，2016年3月创立ROS小课堂，致力于ROS教育的非盈利组织。ROS小课堂一直以开源、免费的形式提供优质的ROS学习资源，在优酷视频平台（<http://i.youku.com/rostutorials>）上提供ROS学习视频，目前已有70多部ROS学习视频，累计播放量9万余次。ROS小课堂

同时还注册了微信公众号，并拥有ROS小课堂的独立网站
www.corvin.cn。ROS小课堂已经连续两年（2017年和2018年）被邀请参加全国“机器人操作系统（ROS）及其应用暑期学校”，在更大的舞台上与大家分享更多更好的ROS学习经验。张志杰还受邀参加北京钢铁侠科技有限公司组织的ROS分享活动，与田之博特合作开发和销售ROS2GO便携系统。ROS小课堂将不断努力，持续为国内的ROS普及与教育贡献一份微薄之力。

第1章

ROS简介

本章将介绍ROS的基本概念，如ROS节点管理器、ROS节点、ROS参数服务器、ROS消息、ROS服务，并讨论如何安装ROS，以及如何从ROS节点管理器开始学习。

本章将介绍以下内容：

- 为什么要学习ROS?
- 为什么有些人愿意选择ROS，有些人不愿意选择ROS?
- 理解ROS的文件系统，及ROS的计算“图”的概念。
- 理解ROS框架的基本构成。
- 从ROS节点管理器开始学习。

1.1 为什么要学习ROS

机器人操作系统（Robot Operating System, ROS）是一个灵活的软件框架，为机器人软件开发提供了丰富的工具和软件库，可以帮助人们提高开发机器人的效率。ROS尤其侧重机器人软件系统中的消息传递、分布式计算、代码复用和最新的算法实现。

ROS项目起源于2007年的Switchyard项目，当时由Morgan Quigley负责开发（见<http://wiki.osrfoundation.org/morgan>）。

Switchyard是斯坦福大学STAIR机器人项目的一个子项目。ROS后期的主要开发是在柳树车库（Willow Garage）进行的（见<https://www.willowgarage.com>）。

ROS社区发展非常迅速，世界上有许多用户和开发者涉足其中。多数高端机器人公司都在将其软件移植到ROS平台上。这一趋势也影响了工业机器人领域，该领域的公司正把软件开发从自己的专用软件平台迁移到ROS框架下。

由于在工业机器人领域进行了大量的研究，过去几年里，ROS在工业中的应用取得了非常好的发展势头。ROS-Industrial可以把ROS框架的先进功能扩展到制造业。最近几年，ROS应用与日俱增，相应能够创造大量与ROS相关的就业岗位。若干年后，ROS将成为机器人工程师必备的技能。

1.2 在机器人开发中，人们为什么更愿意选择ROS

设想我们要搭建一个自主移动的机器人，以下是我们放弃其他机器人开发平台（如Player、YARP、Orocos、MRPT等），选择ROS的一些主要原因：

- 高端功能：ROS自带现成的算法。比如，即时定位与地图构建（Simultaneous Localization and Mapping, SLAM）和自适应蒙特卡罗定位（Adaptive Monte Carlo Localization, AMCL），这两个软件包可用于移动机器人的自主导航。MoveIt! 软件包可以用于机械臂的运动规划。这些功能可以非常方便地直接应用于机器人软件中。这些功能在ROS中的实现方式也是最好的，为已有的功能编写新代码就像“重复发明轮子”。^[1]除此之外，这些功能是高度可配置的，允许用户根据需要对各功能的参数进行微调。
- 大量的工具：ROS集中了大量的工具软件，如调试、可视化、仿真等。其中有一些功能很强的开源工具，如用于调试的rqt_gui、可视化的RViz、仿真用的Gazebo等。其他机器人开发框架很少有如此全的工具。
- 对高端传感器和执行器的支持：ROS包含了机器人领域中各类传感器和执行器的驱动程序和软件接口。高端传感器包括Velodyne激光雷达、激光雷达扫描设备、Kinect体感设备等。还有DYNAMIXEL伺服驱动器这样的执行机构。这样，我们就可以非常方便地将这些设备与ROS连接起来。

- 平台内操作：ROS的消息传递机制允许在不同节点间进行通信。我们可以用C++或C语言对节点进行编程，当然也可以用Python或Java语言。只要这一语言带有ROS客户端库，我们就可以用这种语言对节点进行编程。这样的灵活性在其他机器人框架中是没有的。
- 模块化：大多数独立运行的机器人应用程序都可能出现这样的问题：如果主代码的某一个线程崩溃，整个机器人系统就会停止运行。ROS的情形则不一样，每个进程对应一个节点，如果某一个节点崩溃，整个系统仍然可以运行。除此之外，如果某个传感器或马达出现故障，ROS提供了鲁棒的方法来重启操作。
- 资源并发处理：通过两个以上的进程处理硬件资源，一直以来就是件麻烦事。设想一下，从相机读取一幅图像，既要做人脸检测，又要做运动检测。我们有两种处理方式：一种是写一段可以处理两个任务的代码，另一种是写两段代码，用两个线程并行运行。如果有两个以上的任务，应用系统就变得复杂起来，调试也不容易。在ROS中，我们则可以利用ROS驱动程序发布的ROS话题（topic）访问硬件设备。无论ROS节点数量的多少，都可以同时订阅来自相机驱动程序发布的图像消息。不同节点可以执行不同的功能。这种机制可以降低计算复杂性，提高整个系统的可调试能力。
- 活跃社区：当我们选择一个软件库或软件框架，特别是来自开源社区时，使用之前考虑的主要因素之一是软件技术支持和开发者社区。有些开源工具可能有较好的技术支持，而很多是没有的。在ROS中，技术支持社区很活跃，还有一个门户网站 (<http://answers.ros.org>)，处理来自用户的技术问题。目前，ROS社区开发者人数保持稳定的增长。

选择ROS的原因还有很多，不仅限于以上几点。

下面，我们不妨看看人们不愿意使用ROS的各种原因。这里列出了几种原因。

[1] 即重复已有的工作，而不是集中注意力来解决新问题。——译者注

1.3 为什么有些人不愿意选择ROS

在此，我们总结了人们不愿意选择ROS的几个原因：

- 难学：ROS可能很难学。需要在短期内迅速掌握大量新技术，学习者要不断熟悉很多新概念，才能从ROS框架中有所收获。
- 仿真不易：ROS的主要仿真平台是Gazebo，即使Gazebo的运行没有问题，但要上手使用Gazebo进行仿真却不容易。Gazebo仿真器没有内置的编程环境，所有仿真必须在ROS下编程完成。与Gazebo比较，其他仿真环境（如V-REP、Webots）有内置的编程环境，可以直接对机器人进行原型设计与编程。而且，这些仿真环境有丰富的图形界面（GUI）工具集，支持各类机器人，并且提供了ROS接口。虽然这些相应的工具都是各自平台专用的，但效果也都不错。使用Gazebo和ROS学习机器人仿真难度很高，导致有些人干脆不用ROS。
- 机器人建模难：ROS采用URDF格式对机器人进行建模，URDF是一种基于XML的机器人描述方法。简而言之，我们需要用URDF标签描述机器人模型。而在V-REP中，可以直接利用GUI构建3D机器人模型，而且还可以导入机器人网格模型。在ROS中，可以用SolidWorks插件，把3D模型从SolidWorks转换为URDF。但如果使用其他3D CAD工具，就没有这样的插件。在ROS中学习对一个机器人建模非常耗时，与其他仿真器比较，使用URDF标记构建机器人也很耗时。
- 本身的不足：当前的ROS版本本身也有不足之处。例如，缺少对实时应用系统开发的支持；实现健壮的多机器人分布式应用系统的复杂性很高。

· 商用机器人产品：把ROS部署到一个商用产品中，还需要考虑更多的因素。例如，代码质量。为了便于后期的代码维护，ROS源代码遵循一定的编程规范和编程实践经验。在商用产品开发中，我们必须检查这些代码是否达到商用产品所需的质量水平。我们可能还需要做些额外的工作，进一步提高代码质量。ROS中的大部分代码都是由大学的研究人员提供的，水平良莠不齐。如果我们对ROS代码质量不满意，最好自己重写代码。这样的代码就只适合于特定的机器人，根据需求，可以只使用ROS核心功能模块。

现在我们大概了解了什么时候必须选择ROS，什么时候不必选择ROS。如果你的机器人确实需要用ROS，那么我们现在就详细讨论。我们先看ROS的核心概念。ROS中主要有三个级别：文件系统、计算图、社区。我们将逐一简要介绍。

1.4 理解ROS的文件系统

ROS不仅是一个开发框架，它提供了各种工具和软件库，还提供了类似OS的功能（如硬件抽象、软件包管理、开发人员工具链等），因此我们可以将ROS视为一种超操作系统（meta-operating system）。与真正的操作系统一样，ROS文件以特定的方式组织在硬盘上，如图1-1所示。

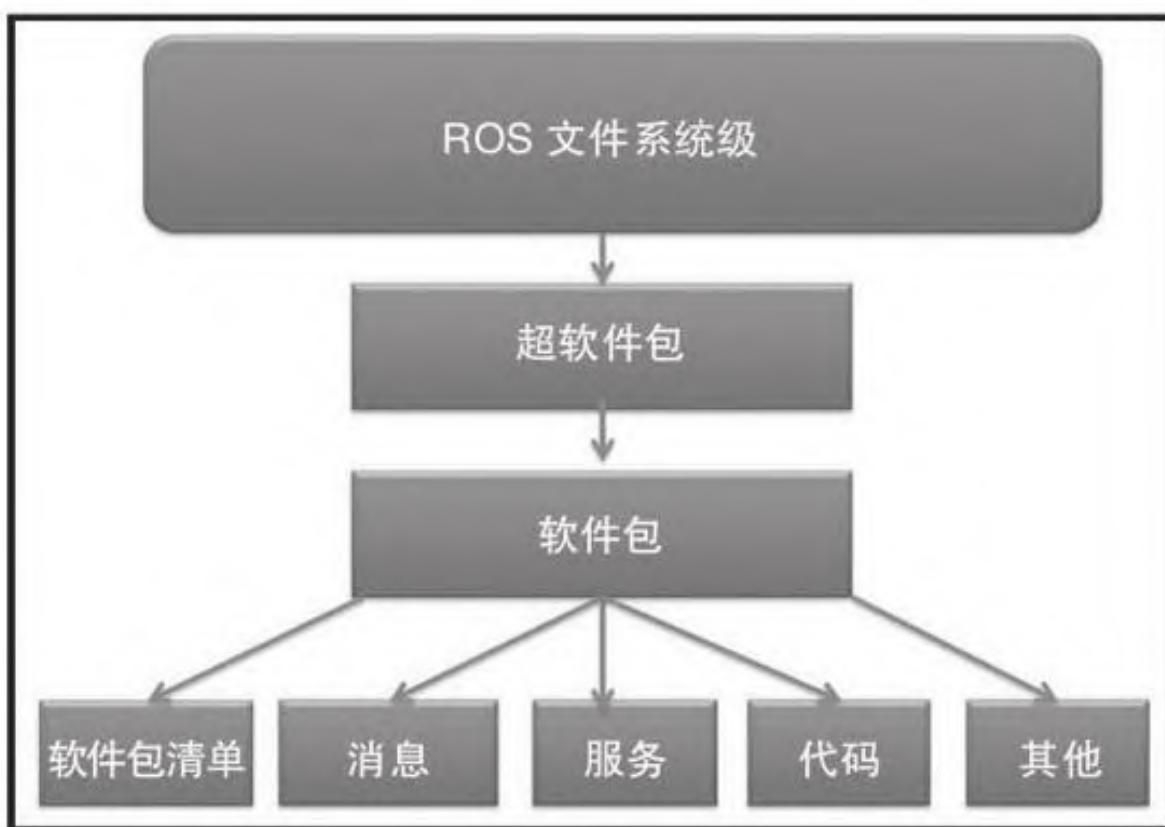


图1-1 ROS文件系统级

以下是文件系统中每个模块的说明：

- 软件包：ROS软件包是ROS框架中最基本的单位，它们包含一个或多个ROS程序（称为节点）、库、配置文件等。它们组合在一起作为一个单元。软件包是ROS软件中基本的构成项和发布项。
- 软件包清单：软件包清单文件位于软件包中，包含软件包基本信息、作者、许可协议、依赖项、编译标志等信息。ROS软件包内的 package.xml 文件就是该包的清单文件。
- 超软件包：超软件包指的是一个或多个相关的软件包以松散的形式组合在一起。本质上讲，超软件包是一种虚拟软件包，不包含任何源代码，也不包含常见软件包中的各种文件。
- 超软件包清单：超软件包清单类似于软件包清单，区别在于它可能包含其他软件包，即那些运行时需要的依赖项，并且声明了一个 export（导出）标识。
- 消息 (.msg)：ROS消息是从一个ROS进程发送到另一个进程的一种信息。我们可以在一个软件包的msg文件夹中定义消息 (my_package/msg/MyMessageType.msg)，其中“.msg”是消息文件的扩展名。
- 服务 (.srv)：ROS服务是一种进程间的请求/应答方式。应答和请求的数据类型可以在一个软件包的srv文件夹内定义 (my_package/srv/MyServiceType.srv)。
- 软件包仓库：大多数ROS软件包采用版本控制系统（Version Control System, VCS）进行维护，如Git、Subversion（svn）、Mercurial（hg）等。在一个版本控制系统下的一组软件包称为软件包仓库，仓库中的软件包可以通过catkin自动发布工具bloom进行对外发布。

图1-2是一个软件包的文件和文件夹的组织示意图，后面几节将介绍如何创建这个软件包。

```
ros_pkg
├── action
│   └── demo.action
├── CMakeLists.txt
├── include
│   └── ros_pkg
│       └── demo.h
├── msg
│   └── message.msg
├── src
│   └── demo.cpp
└── srv
    └── service.srv
```

图1-2 一个示例软件包的文件结构

1.4.1 ROS软件包

ROS软件包的典型结构如图1-3所示。

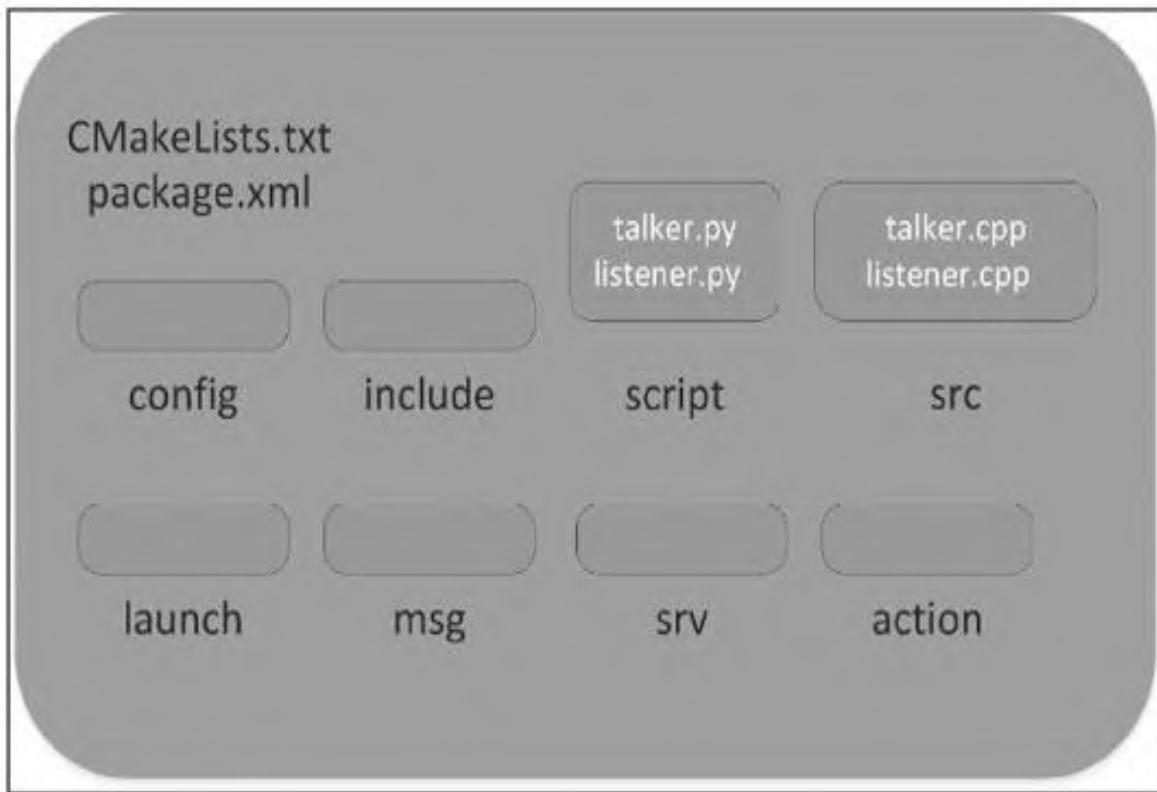


图1-3 C++ ROS软件包的典型结构

下面简单介绍每个文件夹和文件的用途：

- `config`: ROS软件包用到的所有配置文件都在此文件夹中。此文件夹由用户创建，一般习惯将其命名为`config`，将配置文件放置其中。
- `include/package_name`: 此文件夹包含了ROS软件包用到的头文件和库文件。
- `script`: 可执行的Python脚本放在此文件中。在图1-3中，我们可以看到两个示例脚本。
- `src`: 此文件夹存放C++源代码。

- launch: 此文件夹存放启动一个或多个ROS节点的启动文件。
- msg: 此文件夹包含用户定制化消息的定义。
- srv: 此文件夹包含各种服务的定义。
- action: 此文件夹包含动作文件。下一章将介绍这类文件的详细内容。
- package.xml: 这是软件包清单文件。
- CMakeLists.txt: 此文件包含编译软件包的各类指令。

为了能创建、修改、使用ROS软件包，我们需要先了解一些常用命令。下面是一些使用ROS软件包的常用命令：

- catkin_create_pkg: 此命令用于创建一个新软件包。
- rospack: 此命令用于获取软件包相关的信息。
- catkin_make: 此命令用于在工作区中编译软件包。
- rosdep: 此命令将安装一个软件包所需的系统依赖项。

为了能使用软件包，ROS提供了一个类似bash的命令，将其命名为rosbash (<http://wiki.ros.org/rosbash>)，可用于浏览和操作ROS软件包。下面是若干rosbash命令：

- roscd: 此命令用软件包的名称、软件包集的名称或特殊位置信息切换当前所在文件夹。如将软件包的名称作为参数传给该命令，系统将切换到那个软件包所在的文件夹。
- roscp: 此命令用于从软件包中复制文件。
- rosed: 此命令用于使用vim编辑器编辑一个文件。
- rosrun: 此命令用于在软件包内运行一个可执行文件。

图1-4显示了常见的软件包内package.xml文件的定义。

```
<?xml version="1.0"?>
<package>
  <name>hello_world</name>
  <version>0.0.1</version>
  <description>The hello_world package</description>
  <maintainer email="jonathan.cacace@gmail.com">Jonathan Cacace</maintainer>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <export>
  </export>
</package>
```

图1-4 软件包的package.xml文件结构

package.xml文件包括软件包名称、软件包版本、软件包描述、作者详细信息、编译软件包的依赖项和实时运行时的依赖项等。

<build_depend></build_depend>标签包含了编译软件包源码所必需的其他软件包。<run_depend></run_depend>标签中包含的是软件包节点运行时必需的依赖项。

1.4.2 ROS超软件包

超软件包（metapackage）是ROS的一种特殊的软件包，仅包含一个文件，即package.xml文件。超软件包不包含像普通软件包那样的文件夹和文件。

超软件包只是简单地将多个普通软件包组合在一起，形成一个逻辑软件包。在超软件包的package.xml文件中，有一对<export></export>标签，如下所示：

```
<export>
  <metapackage/>
</export>
```

此外，超软件包没有catkin中常见的<buildtool_depend>依赖项，仅有<run_depend>依赖项，这些依赖项就是在超软件包中被组合在一起的那些软件包。

ROS导航（navigation）软件包集是超软件包的一个很好的例子。如果已安装了ROS和导航软件包，可以利用下面的命令切换到导航超软件包所在的文件夹：

```
$ roscd navigation
```

使用喜欢的文本编辑器打开package.xml文件（在此我们用gedit）：

```
$ gedit package.xml
```

这个文件比较长，图1-5显示了一个简化版。

```
<?xml version="1.0"?>
<package>
  <name>navigation</name>
  <version>1.14.0</version>
  <description>
    A 2D navigation stack that takes in information from odometry, sensor
    streams, and a goal pose and outputs safe velocity commands that are sent
    to a mobile base.
  </description>
  ...
  <url>http://wiki.ros.org/navigation</url>
  ...
  <buildtool_depend>catkin</buildtool_depend>
  <run depend>amcl</run depend>
  ...
  <export>
    <metapackage/>
  </export>
</package>
```

图1-5 超软件包的package.xml文件结构

1.4.3 ROS消息

ROS节点可以读/写不同类型的数据。这些数据使用一种简化的消息描述语言来表示，也被称为ROS消息。根据这些数据类型的描述与说明，可以为其他语言的消息类型生成相应的源码。

ROS消息对数据类型的描述与说明存储在当前软件包msg文件夹下的“.msg”文件中。尽管ROS框架提供了大量用于特定机器人的消息，但开发人员依然可以在其节点内定义自己的消息类型。

消息的定义可以由两种类型组成：字段（field）和常量（constant）。字段又可以分为字段类型和字段名。字段类型是传输消息的数据类型，字段名即它的名称。常量在消息文件中定义了一个常量值。

下面是消息定义的示例：

```
int32 number
string name
float32 speed
```

第一部分是字段类型，第二部分是字段名。字段类型即数据类型，字段名可用于访问消息中的值。例如，我们可以使用msg.number访问消息中number的值。

下表列出了可以在消息中直接使用的部分内置数据类型：

主要类型	序列化结果	C++ 类型	Python 类型
bool (1)	8位无符号整型	uint8_t (2)	bool
int8	8位有符号整型	int8_t	int
uint8	8位无符号整型	uint8_t	int (3)
int16	16位有符号整型	int16_t	int
uint16	16位无符号整型	uint16_t	int
int32	32位有符号整型	int32_t	int
uint32	32位无符号整型	uint32_t	int
int64	64位有符号整型	int64_t	long
uint64	64位无符号整型	uint64_t	long
float32	32位单精度浮点数	float	float
float64	64位双精度浮点数	double	float
string	字符串类型	std::string	string
time	秒/纳秒 32位无符号整型	ros::Time	rospy.Time
duration	秒/纳秒 32位有符号整型	ros::Duration	rospy.Duration

还有一些为了满足特定应用需求的消息，例如通用几何信息（geometry_msgs）或传感器信息（sensor_msgs）互换。还有一种特殊类型的ROS消息，称为消息头。消息头可以附加一些特殊信息，如时间戳、参考坐标（或frame_id），以及序列号。使用消息头，我们将获得带编号的消息，可以更清楚地知道当前的消息是由谁发送的。消

息头中的信息主要用于发送数据，如机器人关节变换（TF）数据。下面是消息头的示例：

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

`rosmsg`命令工具可用于检查消息头和字段类型。下面的命令有助于查看具体某一消息的消息头：

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

这行命令会输出示例中的消息头。后面的章节，我们将进一步了解`rosmsg`命令，并学习如何使用自定义消息。

1.4.4 ROS服务

ROS服务是节点间的一种请求/应答式的通信方式。一个节点发送请求，等待来自另外一个节点的应答。这种请求/应答式的通信方式也采用ROS的消息机制。

与使用“.msg”文件的消息定义方式类似，我们必须在一个名为“.srv”的文件中定义服务。该文件必须存储在软件包的srv文件夹中。与消息定义类似，服务描述语言用于定义ROS服务类型。

下面是一种ROS服务的描述格式：

```
#Request message type
string str
---
#Response message type
string str
```

第一部分是请求的消息类型，然后是分隔符。第二部分是应答的消息类型。在本示例中，请求/应答的数据类型都是字符串(string)类型。

1.5 理解ROS的计算图

ROS中的计算是通过进程网络实现的，每个进程为一个ROS节点。这样的计算网络称为计算图（graph）。计算图中的主要概念有：ROS节点（node）、ROS节点管理器（master）、参数服务器（parameter server）、消息（message）、话题（topic）、服务（service）和消息记录包（bag）。图中的每个概念都以不同的方式对此图做出贡献。

与ROS通信相关的软件包包括核心客户端库工具（如rosCPP、rosPython），以及话题、节点、参数、服务等概念的实现。这些软件包都包含在一个名为ros_comm的软件包集中（http://wiki.ros.org/ros_comm）。

该软件包集还包括诸如rostopic、rosparam、rosservice、rosnode之类的工具来呼应前面的概念。

ros_comm软件包集包含了ROS通信的中间件软件包，这些软件包统称为ROS图层（Graph Layer），如图1-6所示。

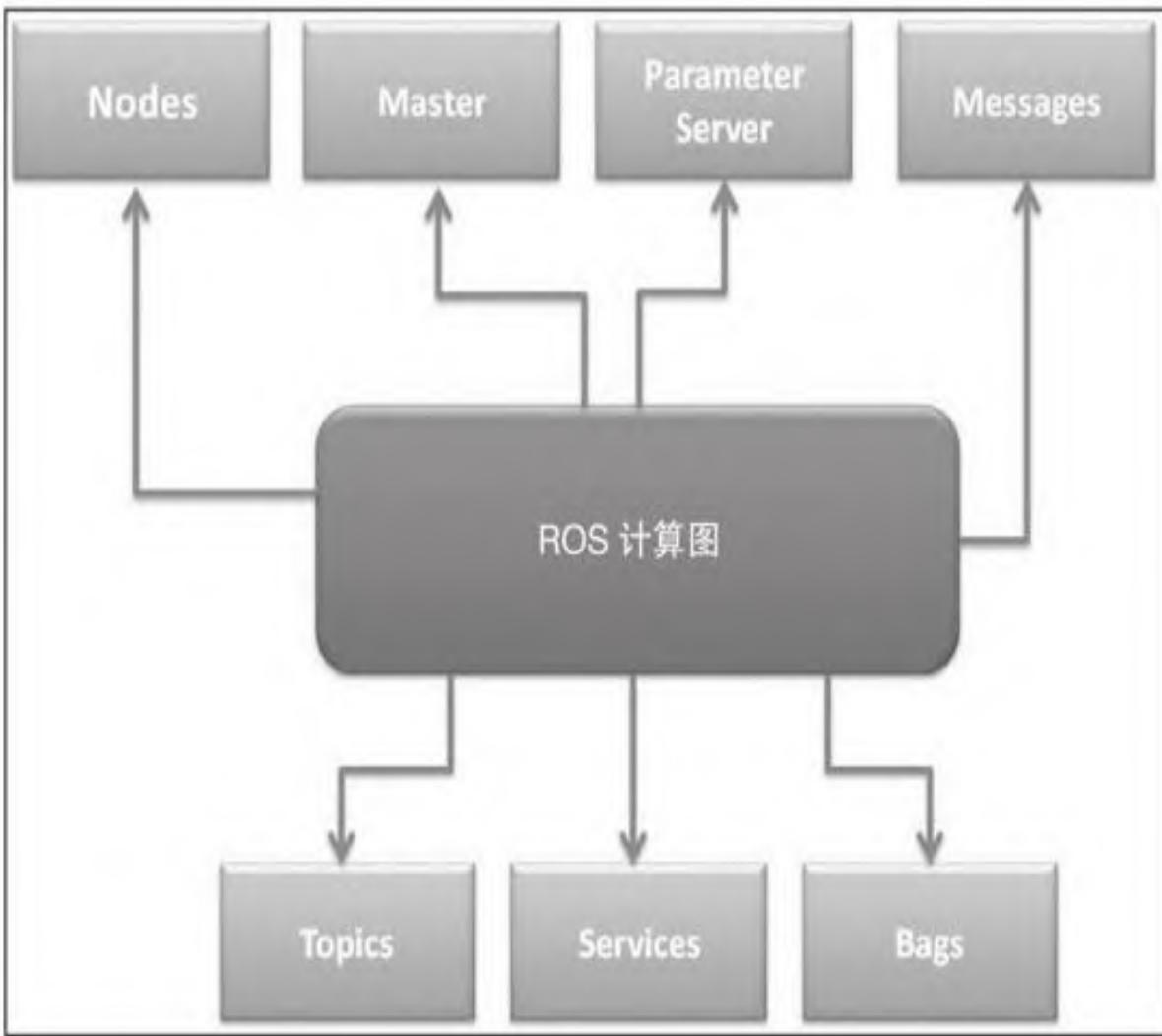


图1-6 ROS图层的结构

以下是图中每个概念的简要介绍：

- 节点 (Node)：节点对应于执行计算任务的进程。每个ROS节点都使用ROS客户端库来编写。使用客户端库提供的API，我们可以实现ROS的各种功能，例如节点间通信。当一个机器人的不同节点间交换信息时，这种节点间的通信就显得尤为有用。利用这种ROS通信方式，节点间可以相互通信并交换数据。ROS节点的作用之一是构建多个简单的进程，而不是一个具有所有功能的大进程。因此ROS节点结构简单，易于调试。

- 节点管理器 (Master)：节点管理器用于所有其他节点的名称注册和查找。如果没有ROS节点管理器，节点将无法找到对方，也无法交换消息或请求服务。在分布式系统中，节点管理器仅运行在某一台电脑上，而其他远程节点通过与节点管理器通信来找到彼此。
- 参数服务器 (Parameter Server)：用户可以利用参数服务器将数据统一存储在一个地方。所有节点都可以访问和修改这些参数值。参数服务器是ROS节点管理器的一部分。
- 消息 (Message)：节点间通过消息实现相互通信。消息是一种包含字段类型的数据结构，它可以保存一组数据，也可以将数据发送到其他节点。ROS消息支持一些标准的数据类型（如整型、浮点型、布尔类型等）。我们还可以利用这些标准的数据类型构建自己的消息类型。
- 话题 (Topic)：ROS中的每条消息都使用被称为“话题”的特定总线进行传输。当节点通过话题发送消息时，我们可以说该节点正在发布话题。当某一节点通过话题接收消息时，我们可以说该节点正在订阅话题。发布节点和订阅节点不知道彼此的存在。我们甚至可以订阅一个没有任何发布者的话题。简而言之，消息的产生和获取是分离的。每个话题都有一个唯一的名称，任何节点都可以访问此话题并通过它发送数据，前提是它们具有正确的消息类型。
- 服务 (Service)：在某些机器人应用系统中，发布/订阅的通信模型不一定合适。例如，在某些情况下，我们需要一种请求/应答的交互方式，即其中一个节点可能向另一个节点请求执行某一个快速过程（如请求一些快速的计算）。基于ROS服务的交互方式就像远程过程调用。
- 日志 (Logging)：ROS提供了一个用于存储数据的记录系统，如记录传感器数据。这些数据可能很难收集，但对于开发和测试机器人算法来说又是必不可少的。当涉及复杂的机器人时，消息记录包对于ROS开发是非常有用的。

图1-7展示了节点间如何使用话题相互通信。话题显示在矩形中，节点用椭圆表示。此图中不包括消息和参数信息。这种类型的图可以

通过调用名为rqt_graph (http://wiki.ros.org/rqt_graph) 的工具来生成。

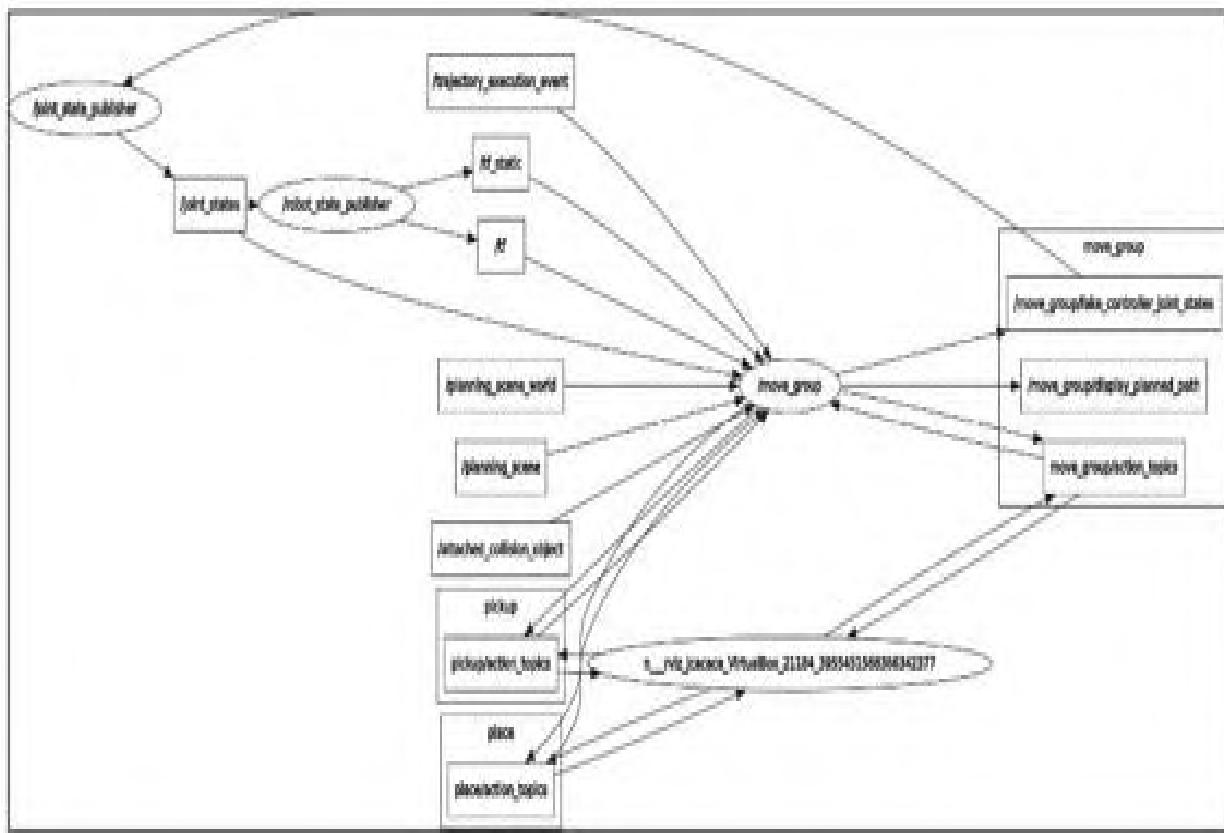


图1-7 基于话题的节点间的通信图

1.5.1 ROS节点

ROS节点是使用ROS客户端库（如roscpp或rospy）编写的执行某种计算的进程。一个节点可以利用ROS话题、ROS服务和ROS参数与其他节点通信。

一个机器人可能包含很多节点。例如，一个节点处理来自摄像头的图像，一个节点处理来自机器人的串口数据，一个节点则可以用来计算里程计信息等。

使用节点可以提高系统的容错能力。即使有一个节点崩溃，整个机器人系统仍然可以运转。与整体代码相比，由于每个节点只处理单个功能，节点机制还可以降低代码复杂性并降低调试难度。

所有正在运行的节点都应该有唯一的名称，以便区别系统中的其他节点。例如，/camera_node可以是广播相机图像的节点名称。

ROS中的小工具rosbash与ROS节点对应。其中，`rosnode`命令可用于获取有关ROS节点的信息。以下是`rosnode`的用法：

- `$ rosnode info [node_name]`: 打印节点的相关信息。
- `$ rosnode kill [node_name]`: 强行终止一个正在运行的节点。
- `$ rosnode list`: 列出正在运行的节点。
- `$ rosnode machine [machine_name]`: 列出运行在某个机器上或多个机器人上的节点。
- `$ rosnode ping`: 检查节点的连通性。
- `$ rosnode cleanup`: 清除无法访问的节点注册信息。

后面，我们会用roscpp编写节点示例，讨论当使用ROS话题、ROS服务、ROS消息和actionlib等功能时，ROS节点的工作机制。

1.5.2 ROS消息

ROS节点通过向话题发布消息来实现彼此通信。如我们前面讨论过的，消息是一种包含字段类型的简单数据结构。ROS消息支持标准的基本数据类型，也支持这些基本数据类型的组合。

节点还可以使用服务来交换信息。服务也是一种消息。服务消息的定义位于srv文件内。

可以使用以下方法访问消息定义。例如，要访问 std_msgs/msg/String.msg，可以使用std_msgs/String。如果使用的是roscpp客户端，必须在字符串消息定义的地方包含 std_msgs/String.h头文件。

除了消息数据类型外，ROS还使用MD5校验来确认发布者和订阅者是否交换相同的消息数据类型。

ROS有一个内置的工具叫rosmsg，可以获取有关ROS消息的信息。下面列出与rosmsg相关的命令参数：

- \$ rosmsg show [message]: 显示消息的描述。
- \$ rosmsg list: 列出所有的消息。
- \$ rosmsg md5 [message]: 显示一个消息的md5sum信息。
- \$ rosmsg package [package_name]: 列出一个软件包中的所有消息。
- \$ rosmsg packages [package_1] [package_2]: 列出包括某消息的软件包。

1.5.3 ROS话题

ROS话题是ROS节点交换消息的特定总线。话题的发布和订阅可以是匿名的，也就是说消息的产生和消息的获取是分离的。ROS节点无须知道哪个节点在发布话题，哪个节点在订阅话题。它们只是按名称来找话题，并检查发布者和订阅者之间的消息类型是否匹配。

话题的通信方式是单向的。如果我们要实现一个请求/应答式的通信，就必须采用ROS服务。

ROS节点利用话题进行通信，采用了基于TCP/IP的数据传输方式，即TCPROS。此方法是ROS默认使用的传输方式。另一种通信类型是UDPROS，这种方式是一种低延迟、非可靠的传输，仅适用于远程操作。

ROS话题的辅助工具可用于获取与ROS话题有关的信息。下面是此命令的语法：

- \$ rostopic bw /topic: 显示给定话题使用的网络带宽。
- \$ rostopic echo /topic: 以可读的方式打印出给定话题的内容。可以使用“-p”选项以csv格式打印数据。
- \$ rostopic find /message_type: 根据给定的消息类型查找相应的话题。
- \$ rostopic hz /topic: 显示给定话题的发布频率。
- \$ rostopic info /topic: 打印一个活动话题的相关信息。
- \$ rostopic list: 列出ROS系统中所有活动的话题。
- \$ rostopic pub /topic message_type args: 向具有指定消息类型的话题发布消息。
- \$ rostopic type /topic: 显示给定话题的消息类型。

1.5.4 ROS服务

如果我们需要一个请求/应答式的通信方式，就必须采用ROS服务。ROS话题的通信方式是单向的，因此无法实现这种双向的通信方式。ROS服务主要应用于分布式系统。

ROS服务由一对消息定义。请求和应答的数据类型都定义在一个名为srv的文件中。所有的srv文件都存储在软件包内的srv文件夹中。

在ROS服务中，一个节点作为ROS服务提供者（服务器），另一个节点（客户端）可以向这个节点请求服务。如果服务提供者完成了服务，将把结果返回给服务客户端。比如，一个节点可以计算给定的两个数的和，并以ROS服务的形式实现此功能。系统中的其他节点则可以通过该节点提供的服务，请求计算给定的两数之和。与之不同的是，话题用于传递连续的数据流。

可以通过如下方式访问ROS的服务定义。比如，`my_package/srv/Image.srv`可以被`my_package/Image`访问。

在ROS服务中，还有一项MD5校验和比较，用于检查节点。如果校验值匹配，服务器才对请求的客户端做出响应。

有两个ROS辅助工具可用于获得关于ROS服务的信息。一个工具为`rossrv`，类似于`rosmsg`，可用于获得有关服务类型的信息。另一个工具为`rosservice`，用于列出并查询正在运行的ROS服务。

下面说明如何用`rosservice`获取正在运行的ROS服务的相关信息：

- `$ rosservice call /service args`: 使用给定的参数调用服务。
- `$ rosservice find service_type`: 查找给定服务类型的服务。
- `$ rosservice info /services`: 显示服务的相关信息。
- `$ rosservice list`: 列出系统中正在运行的服务。
- `$ rosservice type /service`: 显示给定服务的服务类型。

- `$ rosservice uri /service`: 显示服务的ROSRPC URI。

1.5.5 ROS消息记录包

ROS消息记录包文件用于存储来自话题和服务的消息数据。扩展名为“.bag”的文件用于表示ROS消息记录包文件。

ROS消息记录包文件由rosbag命令生成。该命令订阅一个或若干个ROS话题，并在接收到话题的消息数据时将其存储在文件中。一旦话题被记录下来，该文件就可以回放这些话题，也可以重映射现有的话题。

rosbag的主要应用是数据记录。机器人的数据可以被记录下来，也可以对其进行可视化和离线处理。

rosbag命令用于处理“.bag”文件。以下是记录和回放数据记录包文件的命令：

- \$ rosbag record [topic_1] [topic_2] -o [bag_name]: 该命令用于将给定的话题记录到命令参数设定的消息记录包文件中。我们还可以使用参数-a记录所有的话题。
- \$ rosbag play [bag_name]: 该命令用于回放现有的消息记录包文件。

有关此命令的详细信息，可参考链接：
<http://wiki.ros.org/rosbag/CommandLine>。

图形化界面的工具rqt_bag可用于处理ROS消息记录包文件的记录和回放。关于rqt_bag的更多内容，可参考链接：
http://wiki.ros.org/rqt_bag。

1.5.6 ROS节点管理器

ROS节点管理器非常像一个DNS服务器，将唯一的名称和ID与我们系统中活跃的ROS元素关联起来。当一个节点在ROS系统中启动时，它就会查找ROS节点管理器，并在其中注册节点的名称。因此，ROS节点管理器记录了目前ROS系统中运行的所有节点的详细信息。当节点的任何信息发生变化时，它会生成回调并使用最新的详细信息进行更新。为了将节点彼此连接，这些节点的详细信息就非常有用。

当节点开始发布话题时，节点将话题的详细信息（如名称和数据类型）提供给ROS节点管理器。ROS节点管理器将检查是否有其他节点订阅了同一话题。如果有节点订阅了同一话题，ROS节点管理器会将发布者的节点详细信息共享给订阅节点。获取节点详细信息后，这两个节点将使用基于TCP/IP套接字的TCPROS协议进行互连。两个节点连接后，ROS节点管理器就不再起作用了。我们可以根据需要，停止发布者节点或停止订阅者节点。停止任何一个节点，都会再次检查ROS节点管理器。ROS服务也采用相同的方法。

节点使用ROS客户端程序（如roscpp和 rospy）编写。这些客户端程序使用基于XML远程过程调用（XMLRPC）的API与ROS节点管理器进行交互，这些API充当ROS系统API的后端。

ROS_master_URI环境变量包含ROS节点管理器的IP和端口。使用这个环境变量，ROS节点可以找到ROS节点管理器。如果该环境变量错误，节点之间将不会有通信。在单机系统中使用ROS时，可以使用本机的IP或localhost。但是在分布式网络中，计算发生在不同的物理机上，我们应该以合适的方式定义ROS_master_URI。只有这样，远程节点才能找到彼此并相互通信。在分布式系统中，只需要一个ROS节点管理器。该节点管理器应该运行在某一台计算机上，所有其他计算机都可以正确ping这台计算机，从而确保远程ROS节点可以访问ROS节点管理器。

图1-8展示了ROS节点管理器如何与发布者节点和订阅者节点进行交互。发布者节点发布了一个字符串类型话题，消息为Hello World，订阅者节点订阅了此话题。

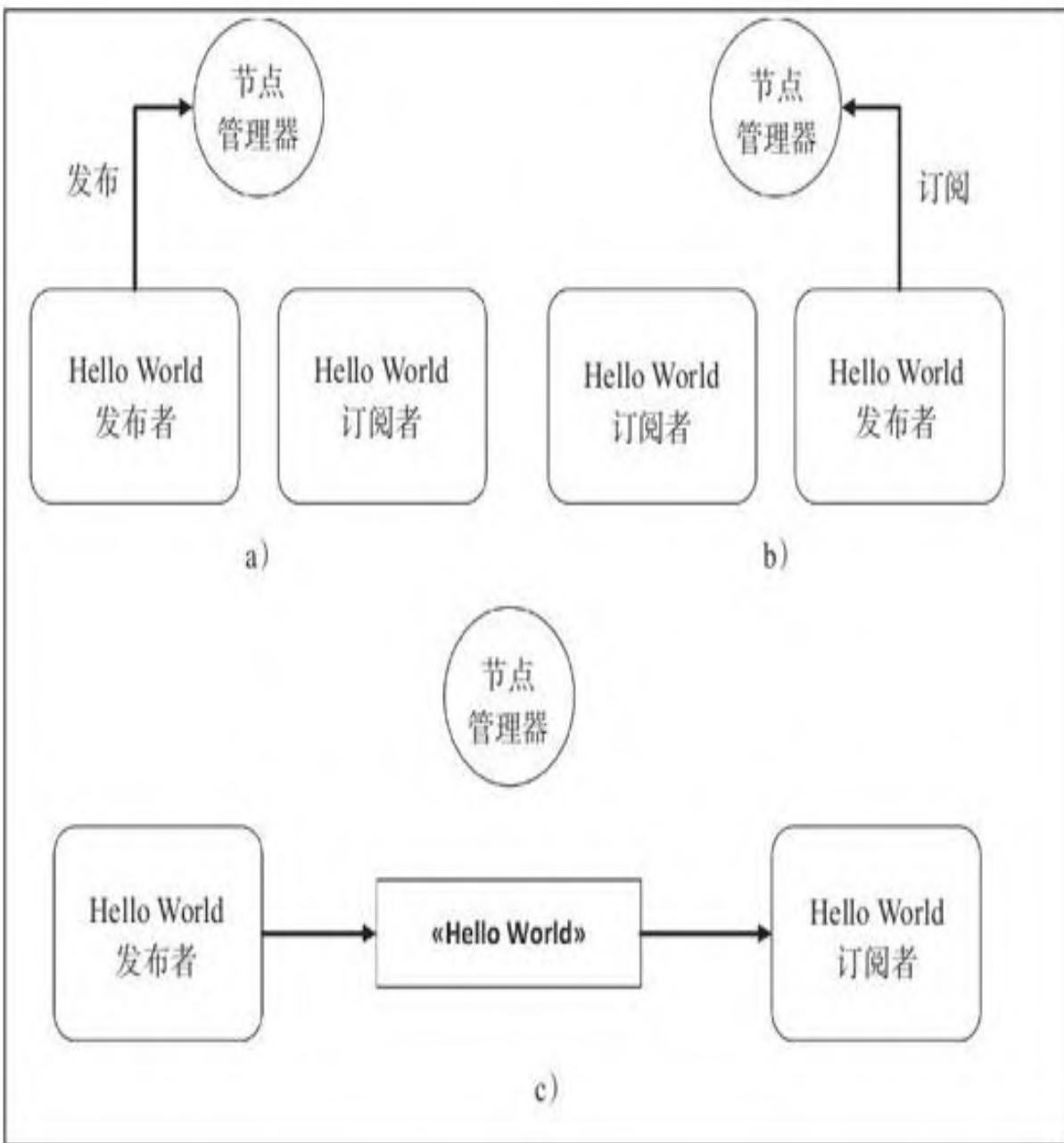


图1-8 ROS节点管理器和“Hello World”发布者和订阅者间的通信

当发布者节点开始在某一个话题中发布Hello World消息时，ROS节点管理器将获取该节点的话题和详细信息。ROS节点管理器将检查是否有其他节点订阅了同一话题。如果当时没有节点订阅该话题，节点间保持未连接状态。如果发布者节点和订阅者节点同时运行，ROS节点管理器会将发布者的详细信息推送到订阅者，这样两个节点将连接，并可以通过ROS消息交换数据。

1.5.7 应用ROS参数

对一个机器人进行编程时，我们可能必须定义机器人参数，如机器人控制器中的P、I、D参数。随着参数数量的增加，我们可能需要将这些参数存储在文件中。有时两个或多个程序需要共享这些参数。在这种情况下，ROS提供了一个参数共享服务器，所有的ROS节点都可以从该服务器访问参数。节点还可以通过参数服务器读取、写入、修改、删除参数值。

我们可以把这些参数存储在一个文件中，并将它们加载到参数服务器中。服务器可以存储各种各样的数据类型，甚至可以存储字典。用户还可以设置参数的范围，如该参数是否仅可以被某个节点访问，还是可以被所有节点访问。

ROS参数服务器支持如下的XMLRPC数据类型：

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

我们还可以在参数服务器上存储字典。如果参数数量很多，我们可以使用YAML文件保存它们。下面就是YAML文件参数定义的示例：

```
/camera/name : 'nikon' #string type  
/camera/fps : 30      #integer  
/camera/exposure : 1.2 #float  
/camera/active : true #boolean
```

`rosparam`工具用于从命令行获取和设置ROS参数。下面是使用ROS参数的命令：

- `$ rosparam set [parameter_name] [value]`: 该命令将为指定参数设置一个值。
- `$ rosparam get [parameter_name]`: 该命令将获取指定参数的参数值。
- `$ rosparam load [YAML file]`: ROS参数可以存储在YAML文件中，用该命令可以将YAML文件加载到参数服务器。
- `$ rosparam dump [YAML file]`: 该命令将现有的ROS参数转储到YAML文件中。
- `$ rosparam delete [parameter_name]`: 该命令删除指定的参数。
- `$ rosparam list`: 该命令列出所有的参数名称。

利用`dynamic_reconfigure`软件包(http://wiki.ros.org/dynamic_reconfigure)，在节点运行过程中，使节点参数可以动态更新。

1.6 ROS的社区

ROS社区资源有助于ROS软件和知识的分享。ROS社区的各种资源如下：

- 发行版（Distribution）：与Linux的发行版类似，ROS发行版是一系列带版本号的、可安装的超软件包的集合。ROS的发行版不仅能使ROS软件的安装更加容易，而且还能在一个发布版本中维持ROS系列软件的版本一致性。
- 软件仓库（Repository）：ROS依赖于开源代码仓库的统一网络，不同的机构能够在这样的系统中开发、发布各自的机器人软件。
- ROS维基（ROSWiki）：ROS维基社区是用于发布并记录有关ROS内容的主要论坛。任何人都可以注册账户并上传自己的文档、发布更正信息、提供软件更新、编写教程等。
- Bug提交系统（Bug ticket system）：如果你发现ROS现有软件中有问题或者想增加一个新功能，就可以用ROS提供的这个资源实现。
- 邮件列表（Mailing list）：ROS用户邮件列表是关于ROS最新资讯的主要交流渠道，当然也可以像论坛一样，进行提问。
- ROS问答（ROSAnswer）：针对ROS的问题，用户可以使用这个资源去提问题。如果你在这个网站提问，其他用户就可能看到，帮助提供解决方法。

- 博客 (Blog) : ROS博客会定期更新与ROS社区相关的新闻、照片、视频 (<http://www.ros.org/news>) 。

1.7 学习ROS需要做哪些准备

在开始学习ROS之前，尤其是练习本书中的代码之前，先确保已经安装如下软件：

- Ubuntu 16.04 LTS / Ubuntu 15.10 / Debian 8：ROS官方只支持Ubuntu和Debian两类操作系统。我们更愿意使用Ubuntu的LTS版本（即Ubuntu 16.04）。
- ROS Kinetic的完整桌面安装版：在安装ROS的完整桌面版时，我们推荐ROS Kinetic版。以下链接为读者提供了最新ROS的安装说明：
<http://wiki.ros.org/kinetic/Installation/Ubuntu>，从仓库列表中选择ros-kinetic-desktop-full软件包。

运行ROS节点管理器和ROS参数服务器

在运行任何ROS节点之前，我们应该先启动ROS节点管理器和ROS参数服务器。我们可以使用一个名为`roscore`的命令来启动ROS节点管理器和ROS参数服务器，该命令会启动以下程序：

- ROS 节点管理器
- ROS 参数服务器
- `rosout` 日志节点

`rosout`节点负责收集来自其他ROS节点的日志消息，并存储在一个日志文件中。还可以把收集到的日志消息转播到其他话题。

“/rosout”话题由ROS节点利用ROS客户端库（如`roscpp`或`rospy`）进行发布，并且`rosout`节点订阅了此话题，该节点在名为

“/rosout_agg”的话题中转播这些消息。“/rosout_agg”话题汇集了所有日志消息。roscore命令是运行任何其他ROS节点之前的先决条件。

下面是在Linux终端上运行roscore的命令：

图1-9的屏幕截图显示了在终端中运行roscore命令时的输出信息：

```
$ roscore

... logging to /home/jcacace/.ros/log/d6cdf7da-6667-11e7-a0a0-0800278bc65c/roslaunch-robot-31486.log
Checking log directory for disk usage. This may take awhile. 1
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://robot:35683/
ros_comm version 1.12.7 2

SUMMARY
=====

PARAMETERS
* /rosdistro: kinetic
* /rosversion: 1.12.7 3

NODES

auto-starting new master
process[master]: started with pid [31498]
ROS_MASTER_URI=http://robot:11311/ 4

setting /run_id to d6cdf7da-6667-11e7-a0a0-0800278bc65c
process[rosout-1]: started with pid [31511]
started core service [/rosout] 5
```

图1-9 运行roscore时终端显示的信息

下面是在终端上运行roscore时各部分的解释：

- 第1部分，我们可以看到在“`~/.ros/log`”文件夹中创建了一个日志文件，用于从ROS节点收集日志。此文件可用于调试。
- 第2部分，该命令启动一个名为`roscore.xml`的ROS启动文件。文件启动时，它会自动启动ROS节点管理器（`rosmaster`）和ROS参数服务器。`roslaunch`命令是一个Python脚本，只要它尝试执行启动文件，它就会启动ROS节点管理器（`rosmaster`）和ROS参数服务器。此部分还显示了ROS参数服务器的地址和端口。
- 第3部分，我们可以看到显示在终端的`rosdistro`和`rosversion`等参数。运行`roscore.xml`，这些参数就会展现出来。我们将在下面进一步了解`roscore.xml`的详细信息。
- 第4部分，我们可以看到ROS节点管理器（`rosmaster`）是用`ROS_master_URI`启动的，之前我们将其定义为一个环境变量。
- 第5部分，我们可以看到`rosout`节点已启动，开始订阅“`/rosout`”话题，并将其转播到“`/rosout_agg`”。

以下是`roscore.xml`文件的内容：

```
<launch>
<group ns="/">
  <param name="rosversion" command="rosversion roslaunch" />
  <param name="rosdistro" command="rosversion -d" />
  <node pkg="rosout" type="rosout" name="rosout" respawn="true"/>
</group>
</launch>
```

在执行`roscore`命令时，首先检查命令行参数，从而获得节点管理器的新端口号。如果得到端口号，它就开始监听新的端口，否则，将使用默认端口号。然后将此节点管理器的端口号和名为`roscore.xml`的启动文件一起传递给`roslaunch`系统。`roslaunch`系统是在Python模块中实现的，它解析端口号，并启动`roscore.xml`文件。

在roscore文件中，可以看到ROS参数和节点被封装在<group></group>标记中，并带有标识为“/”的命名空间。<group></group>标记说明此标记内的所有节点具有相同的设置。

两个参数rosversion和rosdistro，分别存储了rosversion roslaunch和rosversion -d两个命令的输出结果，此处用到的command标签是ROSParam标签的一部分。command标签将运行上面提到的命令，并将命令的输出存储在这两个参数中。

ROS节点管理器（rosmaster）和参数服务器通过使用ROS_master_URI地址在roslaunch模块内执行。这是在Python模块的roslaunch中实现的。ROS_master_URI要连接的ROS节点管理器（ROS master）的环境变量，它是由IP地址和要监听的端口号组成的。端口号可以根据roscore命令中给定的端口号进行更改。

查看roscore命令的输出

让我们查看运行roscore后创建的ROS话题和ROS参数。下面的命令会在终端上列出活动ROS话题：

```
$ rostopic list
```

下面列出了活动的ROS话题。根据我们前面的讨论，rosout节点订阅/rosout话题。这保存了来自ROS节点的所有日志消息，并且/rosout_agg将转播日志消息：

```
/rosout  
/rosout_agg
```

下面的命令列出了运行roscore时ROS系统中所有可用的参数：

```
$ rosparam list
```

这里提到的参数具有发布名称、版本、roslaunch服务器的地址和run_id，其中run_id是与某一特定的roscore运行相关联的唯一ID：

```
/rosdistro  
/roslaunch/uris/host_robot_virtualbox_51189  
/rosversion  
/run_id
```

可以使用以下命令查看运行roscore时生成的ROS服务列表：

```
$ rosservice list
```

运行的服务列表如下所示：

```
/rosout/get_loggers  
/rosout/set_logger_level
```

这些ROS服务是为每个ROS节点生成的，用于设置日志级别。

1.8 习题

学习完本章后，现在你应该能够回答以下问题：

- 我们为什么要学ROS?
- ROS较其他机器人软件平台，有什么不同？
- ROS框架的基本构成是什么？
- roscore的工作机制是什么？

1.9 本章小结

在机器人领域，ROS现在是一款非常流行的软件框架。如果读者愿意在机器人领域施展拳脚，成为一名机器人工程师，掌握ROS将是必备的本领。这一章，我们介绍了ROS的基本概念，讨论了学习ROS的必要性，ROS与其他软件框架的优劣。我们学习的基础概念包括：ROS节点管理器、ROS参数服务器、`roscore`，并介绍了`roscore`的工作原理。下一章，我们将主要介绍ROS软件包，讨论ROS的通信系统的一些实际例子。

第2章

ROS编程入门

在讨论了ROS节点管理器、参数服务器和roscore的基础知识之后，现在我们可以开始创建并编译生成ROS软件包了。在本章中，我们将创建不同的ROS节点来实现ROS通信系统。通过学习ROS软件包，我们还将更新ROS节点、话题、消息、服务和动作库（actionlib）的概念。

本章将介绍以下内容：

- 创建、编译和运行ROS软件包。
- 使用标准和自定义的ROS消息。
- 学习ROS服务和动作库。
- 维护和发布你的ROS软件包。
- 为ROS软件包创建一个维基（wiki）页面。

2.1 创建一个ROS软件包

ROS软件包是ROS系统的基本单位。我们可以创建，然后编译生成一个ROS软件包，并将其向公众发布。当前我们使用的ROS发布版本是Kinetic。我们使用catkin编译系统来编译生成ROS软件包。编译系统主要负责将用户的源码生成“目标（target）”（可执行文件或库文件）。在较老的ROS发行版本中，例如Electric和Fuerte，使用rosbuild来编译生成软件包。由于rosbuild存在各种缺陷，所以catkin应运而生了。catkin基本上基于跨平台编译器（Cross Platform Make, CMake）。它有很多优点，例如可以将软件包移植到另一个操作系统（如Windows）上。如果操作系统支持CMake和Python，就可以轻松地将基于catkin的软件包移植到该系统上。

使用ROS软件包的第一个要求是创建ROS的catkin工作区。安装好ROS后，创建一个名为catkin_ws的catkin工作区：

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

为了编译工作区，我们需要获取ROS的环境变量，以便访问ROS系统提供的功能：

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

切换到前面创建的源码文件夹src中：

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

初始化新的catkin工作区：

```
$ catkin_init_workspace
```

即使现在工作区中没有软件包，我们也可以编译工作区。使用以下命令切换到工作区文件夹中：

```
$ cd ~/catkin_ws
```

下面使用catkin_make编译工作区：

```
$ catkin_make
```

执行完上面这条命令后，在catkin工作区中将生成devel和build文件夹。各种安装文件位于devel文件夹中。要将创建的ROS工作区添加到ROS环境变量中，我们需要获取其中的一个安装文件setup.bash。此外，在每次使用以下命令启动新的bash会话时，我们都可以获取此工作区的配置文件：

```
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

配置好catkin工作区后，我们就可以创建自己的软件包了，其中将包含示例节点，用来演示ROS话题、消息、服务和动作库的工作方式。catkin_create_pkg命令就是用来创建ROS软件包的。我们将用它创建各种ROS概念的演示示例。

切换到catkin工作区的src文件夹后，使用以下命令创建软件包：

```
$ catkin_create_pkg package_name [dependency1] [dependency2]
```



源码文件夹：所有的ROS软件包（无论是从头创建的还是从其他代码库中下载的）都必须放在ROS工作区的src文件夹中，否则ROS系统将无法识别它们，从而导致无法编译。

下面是创建ROS示例软件包的命令：

```
$ catkin_create_pkg mastering_ros_demo_pkg roscpp std_msgs  
actionlib actionlib_msgs
```

软件包中的依赖关系如下：

- roscpp：这是ROS的C++实现，一个ROS客户端库，为C++开发人员提供API，使用ROS话题、服务和参数等生成ROS节点。包含该依赖的原因是我们要编写的是一个C++实现的ROS节点。任何使用C++节点代码的ROS软件包都必须添加此依赖项。
- std_msgs：该软件包包含了基本的ROS原始数据类型，例如整型、浮点型、字符串、数组等。我们可以在节点中直接使用这些数据类型，而无须定义新的ROS消息。
- actionlib：该超软件包提供了在ROS节点中创建可抢占任务的接口。我们在这个软件包中创建了基于actionlib的节点，所以我们需要包含该

软件包来创建ROS节点。

· actionlib_msgs：该软件包包含了与动作服务器和动作客户端交互所需的标准消息定义。

创建了软件包后，我们也可以通过编辑CMakeLists.txt和package.xml这两个文件来手动添加其他依赖项。如果成功创建了软件包，我们将收到以下信息，如图2-1所示。

```
Created file mastering_ros_v2_pkg/package.xml
Created file mastering_ros_v2_pkg/CMakeLists.txt
Created folder mastering_ros_v2_pkg/include/mastering_ros_v2_pkg
Created folder mastering_ros_v2_pkg/src
Successfully created files in /home/jcacace/mastering_ros_v2_pkg. Please
adjust the values in package.xml.
```

图2-1 创建ROS软件包时的终端信息

创建了这个软件包后，可以使用catkin_make命令来编译生成软件包，但是它不会增加任何节点。我们必须在catkin工作区的根路径下执行此命令。以下是我们编译生成空ROS软件包的命令：

```
~/catkin_ws $ catkin_make
```

成功编译生成软件包后，我们可以将节点源码添加到工作区下的src文件夹中。

在CMake的build文件夹中主要包含了节点的可执行文件，该节点的源码位于catkin工作区的src文件夹中。devel文件夹主要包含了在编译过程中生成的bash脚本、头文件和可执行文件。我们可以看到使用catkin_make创建并编译生成ROS节点的过程。

2.1.1 学习ROS话题

话题是两个节点间通信的基本方式。在本节，我们将学习话题的工作原理。下面我们将创建两个ROS节点，一个发布话题，另一个订阅该话题。进入`mastering_ros_demo_pkg`文件夹中，在`/src`源码文件夹中，`demo_topic_publisher.cpp`和`demo_topic_subscriber.cpp`是我们将要讨论的两个源码文件。

2.1.2 创建ROS节点

我们要讨论的第一个节点是demo_topic_publisher.cpp。此节点将在名为/numbers的话题上发布一个整型数值。我们可以将下面的代码复制到新软件包中或使用代码仓库中的现有文件：

以下是完整的代码：

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>
int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_topic_publisher");
    ros::NodeHandle node_obj;
    ros::Publisher number_publisher =
    node_obj.advertise<std_msgs::Int32>("/numbers",10);
    ros::Rate loop_rate(10);
    int number_count = 0;
    while (ros::ok())
    {
        std_msgs::Int32 msg;
        msg.data = number_count;
        ROS_INFO("%d",msg.data);
        number_publisher.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++number_count;
    }
    return 0;
}
```

以下是上述代码的详细说明：

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>
```

`ros/ros.h`是ROS的主要头文件。如果我们想在代码中使用`roscpp`客户端API的话，就必须包含此头文件。`std_msgs/Int32.h`是整型数据类型的标准消息定义的头文件。

这里我们通过话题发送整型数据，所以我们需要一个消息类型来处理这些整型数据。`std_msgs`包含了基本数据类型的标准消息定义。`std_msgs/Int32.h`包含了整型消息的定义。

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

该段代码将初始化一个带有名称的ROS节点。需要注意的是，ROS节点名应该是唯一的。所有的ROSC++节点代码都必须包含此行代码。

```
ros::NodeHandle node_obj;
```

这将创建一个`NodeHandle`对象，用于与ROS系统进行通信。

```
ros::Publisher number_publisher =
node_obj.advertise<std_msgs::Int32>("/numbers", 10);
```

这将创建一个话题发布者，该话题是`std_msgs::Int32`消息类型，话题名是`/numbers`。第二个参数是缓冲区大小，它表示在发送消息之前可以将多少消息放到缓冲区中。如果数据发送的频率很高，该参数也需要设置得很大。

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

这是用来设置发送数据的频率。

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

这是一个无限while循环，当我们按下Ctrl+C时，它才会退出。如果出现一个中断，ros::ok()函数将返回0，这样就可以中断该while循环。

```
std_msgs::Int32 msg;  
msg.data = number_count;
```

第一行创建了一个整型ROS消息，第二行为这条消息分配了整型数据。在这里，data是msg对象的字段名。

```
ROS_INFO("%d",msg.data);
```

这将打印消息数据。该行用于输出ROS消息的日志。

```
number_publisher.publish(msg);
```

这将发布消息到/numbers话题上。

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

此命令将读取和更新所有的ROS话题。一个节点如果没有spin()或spinOnce()函数的话是不会发布消息的。

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

这行能提供必要的延时使发布消息的频率达到10Hz。

讨论了发布者节点之后，现在我们可以讨论订阅者节点了，即demo_topic_subscriber.cpp。你可以将下面的源码复制到一个新文件中或者使用已有的文件。

以下是订阅者节点的源码定义：

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>
void number_callback(const std_msgs::Int32::ConstPtr& msg) {
    ROS_INFO("Received [%d]", msg->data);
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "demo_topic_subscriber");
    ros::NodeHandle node_obj;
    ros::Subscriber number_subscriber =
node_obj.subscribe("/numbers", 10, number_callback);
    ros::spin();
    return 0;
}
```

以下是代码的解析：

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>
```

这是订阅者所需的头文件。

```
void number_callback(const std_msgs::Int32::ConstPtr& msg) {
    ROS_INFO("Received [%d]", msg->data);
}
```

这是一个回调函数，只要有数据发送到/nodes话题上，它就会被自动调用执行。当有数据发送到此话题上时，该函数将调用并提取消息中的数值，然后将其打印在控制台上。

```
ros::Subscriber number_subscriber =  
node_obj.subscribe("/numbers", 10, number_callback);
```

这里定义了一个订阅者，我们给出了订阅所需的话题名称、缓冲区大小和要执行的回调函数。我们将订阅/nodes话题，回调函数已经在前面看到过了。

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

这是一个无限循环，节点将在此步骤中一直等待。只要有数据被发送到话题上，此代码就会立刻执行相应话题的回调函数。只有当按下Ctrl+C键时，该节点才会退出。

2.1.3 编译生成节点

我们必须编辑软件包中的CMakeLists.txt文件才能编译和构建源码。切换到mastering_ros_demo_pkg以查看现有的CMakeLists.txt文件。此文件中的下面这段代码将负责构建这两个节点：

```
include_directories(  
    include  
    ${catkin_INCLUDE_DIRS}  
    ${Boost_INCLUDE_DIRS}  
)  
  
#This will create executables of the nodes  
add_executable(demo_topic_publisher src/demo_topic_publisher.cpp)  
add_executable(demo_topic_subscriber src/demo_topic_subscriber.cpp)  
  
#This will generate message header file before building the target  
add_dependencies(demo_topic_publisher  
    mastering_ros_demo_pkg_generate_messages_cpp)  
add_dependencies(demo_topic_subscriber  
    mastering_ros_demo_pkg_generate_messages_cpp)  
  
#This will link executables to the appropriate libraries  
target_link_libraries(demo_topic_publisher ${catkin_LIBRARIES})  
target_link_libraries(demo_topic_subscriber ${catkin_LIBRARIES})
```

我们可以创建一个新的CMakeLists.txt文件，然后添加前面的代码，这样也可以编译这两个节点。

catkin_make命令是用来编译生成软件包的。首先我们需要切换到工作区根文件夹中：

```
$ cd ~/catkin_ws
```

像下面这样来编译生成mastering_ros_demo_package软件包：

```
$ catkin_make
```

或者我们可以使用前面的命令来编译整个工作区，也可以使用-DCATKIN_WHITELIST_PACKAGES选项。使用此选项可以设置一个或多个要编译的软件包：

```
$ catkin_make -DCATKIN_WHITELIST_PACKAGES="pkg1,pkg2,..."
```

注意，必须还原此配置才能编译其他软件包或整个工作区。我们可以使用以下命令来完成这个操作：

```
$ catkin_make -DCATKIN_WHITELIST_PACKAGES=""
```

如果编译完成，我们就可以执行这个节点了。首先需要启动roscore：

```
$ roscore
```

现在，在两个终端中分别运行这两个命令。运行发布者节点：

```
$ rosrun mastering_ros_demo_package demo_topic_publisher
```

运行订阅者节点：

```
$ rosrun mastering_ros_demo_package demo_topic_subscriber
```

我们可以看到如图2-2所示的输出。

The screenshot shows two terminal windows side-by-side. The left window is titled 'catkin@robot: ~\$ rosrun mastering_ros_demo_pkg demo_topic_publisher' and displays the following log entries:

```
[INFO] [1500276155.757000857]: 0  
[INFO] [1500276155.857052842]: 1  
[INFO] [1500276155.957062454]: 2  
[INFO] [1500276156.057095824]: 3  
[INFO] [1500276156.157097268]: 4  
[INFO] [1500276156.257505796]: 5  
[INFO] [1500276156.357532737]: 6
```

The right window is titled 'catkin@robot: ~\$ rosrun mastering_ros_demo_pkg demo_topic_subscriber' and displays the following log entries:

```
[INFO] [1500276156.057591945]: Received [3]  
[INFO] [1500276156.157553762]: Received [4]  
[INFO] [1500276156.257991575]: Received [5]  
[INFO] [1500276156.358834728]: Received [6]  
[INFO] [1500276156.457377162]: Received [7]  
[INFO] [1500276156.557647552]: Received [8]  
[INFO] [1500276156.658285212]: Received [9]
```

图2-2 运行话题的发布者和订阅者节点

图2-3显示了节点间是如何相互通信的。我们可以看到demo_topic_publisher节点向/numbers话题发布信息，然后demo_topic_subscriber节点订阅这个话题：

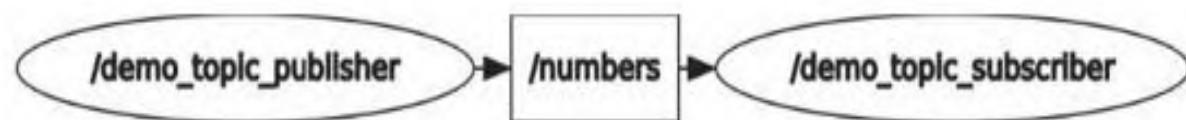


图2-3 发布者和订阅者节点间的通信图

我们可以使用rosnode和rostopic工具来调试和理解这两个节点的工作方式：

- \$ rosnode list: 这将列出所有活动的节点。
- \$ rosnode info demo_topic_publisher: 这将获取发布者节点的相关信息。

- `$ rostopic echo /numbers`: 这将显示发送到/`numbers`话题上的数据。
- `$ rostopic type /numbers`: 这将打印/`numbers`话题的消息类型。

2.2 添加自定义的msg和srv文件

在本节中，我们将学习如何在当前软件包中创建自定义的消息和服务。消息的定义存储在`.msg`文件中，服务的定义存储在`.srv`文件中。这些定义告知ROS从节点发送的数据类型和数据名称。当添加自定义消息时，ROS会将其定义转换为等价的C++代码，这样我们就可以将这些代码包含在节点中。

首先我们来介绍消息定义。这些消息定义必须写在`.msg`文件中，且必须保存在软件包内的`msg`文件夹中。现在我们创建一个名为`demo_msg.msg`的消息文件，定义如下：

```
string greeting
int32 number
```

到目前为止，我们一直使用的都是标准消息定义。现在，我们已经可以创建自定义的消息了，并学会如何在我们的代码中使用它们。

第一步是编辑当前软件包中的`package.xml`文件，取消注释：
`<build_depend>message_generation</build_depend>`和
`<exec_depend>message_runtime</exec_depend>`。

编辑当前的`CMakeLists.txt`并添加`message_generation`行，如下所示：

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  actionlib
  actionlib_msgs
  message_generation
)
```

取消下面这些行代码的注释并添加自定义消息文件：

```
add_message_files(
  FILES
    demo_msg.msg
)
## Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
    std_msgs
    actionlib_msgs
)
```

完成这些步骤后，我们就可以编译和生成软件包了：

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

要检查消息生成是否正确，我们可以使用rosmsg命令来检查：

```
$ rosmsg show mastering_ros_demo_pkg/demo_msg
```

如果执行命令后显示的内容和定义一样，那说明创建自定义消息的过程是正确的。

如果我们想测试自定义的消息，可以使用名为demo_msg_publisher.cpp和demo_msg_subscriber.cpp的自定义消息类型生成发布者和订阅者来测试节点。在mastering_ros_demo_pkg/1文件夹中可以获取到这些代码。

我们可以通过在CMakeLists.txt中添加以下代码来测试自定义消息：

```
add_executable(demo_msg_publisher src/demo_msg_publisher.cpp)
add_executable(demo_msg_subscriber src/demo_msg_subscriber.cpp)

add_dependencies(demo_msg_publisher
mastering_ros_demo_pkg_generate_messages_cpp)
add_dependencies(demo_msg_subscriber
mastering_ros_demo_pkg_generate_messages_cpp)

target_link_libraries(demo_msg_publisher ${catkin_LIBRARIES})
target_link_libraries(demo_msg_subscriber ${catkin_LIBRARIES})
```

使用catkin_make来编译生成软件包，并使用以下命令测试这些节点。

- 运行roscore：

```
$ roscore
```

· 启动自定义消息的发布者节点：

```
$ rosrun mastering_ros_demo_pkg demo_msg_publisher
```

· 启动自定义消息的订阅者节点：

```
$ rosrun mastering_ros_demo_pkg demo_msg_subscriber
```

发布者节点将发布一条带有字符串和一个整数的消息，订阅者节点订阅该话题并打印这些内容。输出内容和通信图如图2-4所示。

jacacae@robot: \$ rosrun mastering_ros_demo_pkg demo_msg_publisher	jacacae@robot: \$ rosrun mastering_ros_demo_pkg demo_msg_subscriber
[INFO] [1500276387.166778705]: 0	[INFO] [1500276387.467496528]: Received greeting [hello world]
[INFO] [1500276387.166801438]: hello world	[INFO] [1500276387.467579254]: Received [3]
[INFO] [1500276387.267694471]: 1	[INFO] [1500276387.567331442]: Received greeting [hello world]
[INFO] [1500276387.267855187]: hello world	[INFO] [1500276387.567382312]: Received [4]
[INFO] [1500276387.3688983935]: 2	[INFO] [1500276387.668345874]: Received greeting [hello world]
[INFO] [1500276387.368898128]: hello world	[INFO] [1500276387.668564167]: Received [5]
[INFO] [1500276387.466853659]: 3	[INFO] [1500276387.768672445]: Received greeting [hello world]
[INFO] [1500276387.466933039]: hello world	[INFO] [1500276387.768753221]: Received [6]

图2-4 运行使用自定义消息的发布者和订阅者

节点间通信的话题名为/dem0_msg_topiC。图2-5是两个节点之间的通信图：

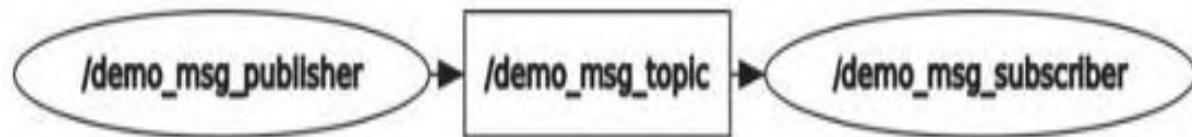


图2-5 消息发布者和订阅者之间的通信图

接下来，我们可以在软件包中添加srv文件。首先在当前软件包的文件夹中创建一个名为srv的新文件夹，然后添加一个名为demo_srv.srv的srv文件。该文件的定义如下：

```
string in
---
string out
```

这里的请求（Request）和应答（Response）都是字符串类型。

下一步，我们需要取消package.xml文件中的以下两行注释，和我们在ROS消息中所做的一样：

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

打开CMakeLists.txt文件并在catkin_package()中添加message_runtime：

```
catkin_package(
    CATKIN_DEPENDS roscpp rospy std_msgs actionlib actionlib_msgs
    message_runtime
)
```

我们需要遵循与生成ROS消息相同的过程来生成服务。除此之外，我们还需要取消其他部分注释，如下所示：

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

做了这些修改后，我们就可以使用`catkin_make`来编译生成软件包了。使用下面的命令验证整个过程是否正确：

```
$ rossrv show mastering_ros_demo_pkg/demo_srv
```

如果我们看到输出内容与服务文件中定义的内容一样，我们就可
以确认创建自定义服务的过程是正确的。

2.3 使用ROS服务

在本节中，我们将创建两个ROS节点，它们使用我们已经自定义了的服务。我们创建的服务节点可以将一个字符串消息作为请求发送到服务器上，服务器节点将反馈另一条消息作为应答。

进入mastering_ros_demo_pkg/src文件夹，找到名为demo_service_server.cpp和demo_service_client.cpp的节点代码。

demo_service_server.cpp是服务器节点源码，定义如下：

```
#include "ros/ros.h"
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
#include <sstream>
using namespace std;

bool demo_service_callback(mastering_ros_demo_pkg::demo_srv::Request &req,
                           mastering_ros_demo_pkg::demo_srv::Response &res) {
    std::stringstream ss;
    ss << "Received Here";
    res.out = ss.str();
    ROS_INFO("From Client [%s], Server says [%s]", req.in.c_str(), res.out.c_str());
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_service_server");
    ros::NodeHandle n;
    ros::ServiceServer service = n.advertiseService("demo_service",
                                                     demo_service_callback);
    ROS_INFO("Ready to receive from client.");
    ros::spin();
    return 0;
}
```

让我们看看代码的解释：

```
#include "ros/ros.h"
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
#include <sstream>
```

这里包含了ros/ros.h头文件，它是编写ROS CPP节点代码必须要包含的头文件。mastering_ros_demo_pkg/demo_srv.h头文件是生成的一个头文件，它包含我们服务的定义，我们可以在代码中直接使用它。sstream.h是用于获取字符串流的头文件。

```
bool demo_service_callback(mastering_ros_demo_pkg::demo_srv::Request &req,
                           mastering_ros_demo_pkg::demo_srv::Response &res)
{
}
```

这是在服务器节点上收到请求时执行的回调函数。服务器可以从客户端接收消息类型为`mastering_ros_demo_pkg::demo_srv::Request`的请求，也可以发送`mastering_ros_demo_pkg::demo_srv::Response`类型的应答。

```
std::stringstream ss;
ss << "Received Here";
res.out = ss.str();
```

在这段代码中，将字符串数据“Received Here”传递给服务的Response实例。在这里，out是我们在demo_srv.srv中给出的应答的字段名称。这个应答将反馈到服务客户端节点：

```
ros::ServiceServer service = n.advertiseService("demo_service",  
demo_service_callback);
```

这里创建了一个名为demo_service的服务，并在请求到达此服务时执行回调函数。回调函数名为demo_service_callback，我们在上一节中学过。

接下来，我们学习demo_service_client.cpp是如何工作的。

以下是此段代码的定义：

```
#include "ros/ros.h"
#include <iostream>
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
#include <sstream>
using namespace std;

int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_service_client");
    ros::NodeHandle n;
    ros::Rate loop_rate(10);
    ros::ServiceClient client =
n.serviceClient<mastering_ros_demo_pkg::demo_srv>("demo_service");
    while (ros::ok())
    {
        mastering_ros_demo_pkg::demo_srv srv;
        std::stringstream ss;
        ss << "Sending from Here";
        srv.request.in = ss.str();
        if (client.call(srv))
        {
            ROS_INFO("From Client [%s], Server says
[%s]", srv.request.in.c_str(), srv.response.out.c_str());
        }
        else
        {
            ROS_ERROR("Failed to call service");
            return 1;
        }

        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

我们来解释一下该代码：

```
ros::ServiceClient client =  
n.serviceClient<mastering_ros_demo_pkg::demo_srv>("demo_service");
```

此行代码将创建一个服务客户端，其消息类型为
`mastering_ros_demo_pkg::demo_srv`，它可以与名为`demo_service`的
ROS服务器进行通信。

```
mastering_ros_demo_pkg::demo_srv srv;
```

此行代码将创建一个新的服务对象实例。

```
std::stringstream ss;  
ss << "Sending from Here";  
srv.request.in = ss.str();
```

将名为“`Sending from Here`”的字符串填充到请求实例中。

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

这会将服务调用发送到服务器上。如果发送成功，将打印应答和
请求。如果失败，则什么都不做。

```
ROS_INFO("From Client [%s], Server says  
[%s]", srv.request.in.c_str(), srv.response.out.c_str());
```

如果收到了应答，将打印请求和应答。

在讨论了这两个节点之后，现在我们讨论如何编译生成这两个节点。将以下代码添加到CMakeLists.txt就可以来编译生成这两个节点了：

```
add_executable(demo_service_server src/demo_service_server.cpp)
add_executable(demo_service_client src/demo_service_client.cpp)

add_dependencies(demo_service_server
mastering_ros_demo_pkg_generate_messages_cpp)
add_dependencies(demo_service_client
mastering_ros_demo_pkg_generate_messages_cpp)

target_link_libraries(demo_service_server ${catkin_LIBRARIES})
target_link_libraries(demo_service_client ${catkin_LIBRARIES})
```

我们可以使用下面的命令来编译该代码：

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

要启动节点，首先需要执行roscore命令，然后再使用下面的命令来启动节点，结果如图2-6所示：

```
$ rosrun mastering_ros_demo_pkg demo_service_server
$ rosrun mastering_ros_demo_pkg demo_service_client
```

```
jcacace@robot:~/catkin_ws$ rosrun mastering_ros_demo_pkg demo_service_server
[INFO] [1499857954.849054844]: Ready to receive from client.
[INFO] [1499857956.626780527]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857956.727580536]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857956.827664441]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857956.933545057]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.027340860]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.127714980]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.227157798]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.328243221]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.427351564]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.527108113]: From Client [Sending from Here], Server says [Received Here]

jcacace@robot:~$ rosrun mastering_ros_demo_pkg demo_service_client
[INFO] [1499857956.627200681]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857956.727860599]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857956.828064716]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857956.934237703]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.027558745]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.127958080]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.227397212]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.328513872]: From Client [Sending from Here], Server says [Received Here]
[INFO] [1499857957.427616100]: From Client [Sending from Here], Server says [Received Here]
```

图2-6 运行ROS服务客户端和服务端节点

我们可以使用的`rosservice`命令如下：

- `$ rosservice list`: 这将列出当前系统中的ROS服务。
- `$ rosservice type /demo_service`: 这将打印/dem0_service的消息类型。
- `$ rosservice info /demo_service`: 这将打印/dem0_service相关的信息。

2.3.1 使用ROS动作库

在ROS服务中，我们实现了两个节点间请求/应答式的交互，但是如果应答需要花费太多时间或者服务器没有完成指定的工作，那么我们就必须等待它完成。在等待请求的动作结束的过程中，主应用程序将被阻塞。此外，我们也可以通过调用客户端来监视远程进程的执行进度。这种情形下，我们应该使用actionlib来实现应用。ROS中的另一种用法是：如果请求没有像我们预期的那样按时完成，我们就可以抢占正在运行的请求，并发送另一个请求。actionlib软件包提供了实现这类抢占任务的标准方法。actionlib经常用在机器人手臂导航和移动机器人导航中。下面让我们来看看如何实现动作服务器和动作客户端。

与ROS服务一样，在actionlib中，我们也必须初始化动作规范。这个动作的规范存储在以.action结尾的文件中。该文件必须保存在ROS软件包内的action文件夹中。action文件包含以下各部分内容。

- Goal（目标）：动作客户端可以发送一个必须由动作服务器来执行的目标。这就类似于ROS服务中的请求。例如，如果机器人手臂关节想从45度转动到90度，那么这里的目标就是90度。
- Feedback（反馈）：动作客户端向动作服务器发送目标后，将开始执行回调函数。反馈只是简单地给出回调函数内当前操作的进度。通过使用反馈，我们可以获得当前任务的进度。在前面的例子中，机器人手臂关节必须移动到90度，在这种情况下，反馈就是机械臂从45度转动到90度之间的中间值。
- Result（结果）：完成目标后，动作服务器将发送完成的最终结果，它可以是计算结果或者一个确认。在前面的例子中，如果关节转动到90度，则完成了目标任务，结果就是可以表示目标完成的任何形式。

在此，我们讨论一个演示动作服务器和动作客户端。动作客户端将发送一个数字作为目标。动作服务器收到目标后，将从0开始计数，每秒加1，加到给定数字。如果在给定的时间内完成计数累加，它将发送结果，否则，该任务将被客户端抢占。这里的反馈是计算的进度。该任务的动作文件如下，动作文件名为Demo_action.action：

```
#goal definition
int32 count
---
#result definition
int32 final_count
---
#feedback
int32 current_number
```

这里的count值是目标，服务器必须从0开始增加到该值。
final_count是结果，即任务完成后最终的结果值。current_number是反馈值（即当前的计数值），表示任务的进度。

进入mastering_ros_demo_pkg/src文件夹，你就可以看到动作服务器节点的源码文件demo_action_server.cpp和动作客户端节点的源码文件demo_action_client.cpp。

创建ROS动作服务器

在本节中，我们将讨论demo_action_server.cpp。这个动作服务器将接收一个整型目标值。当服务器得到此目标值后，它将从0开始计数，直到该值。如果计数完成了，它将成功地完成动作。如果在完成任务之前被抢占，动作服务器将寻找另一个目标。

这段代码有点长，所以我们就只讨论这段代码的重要部分：

首先我们来看头文件：

```
#include <actionlib/server/simple_action_server.h>
#include "mastering_ros_demo_pkg/DemoAction.h"
```

第一个头文件是用于实现动作服务器节点的标准动作库。第二个头文件是由存储的action文件生成的。它包含了我们的动作定义。

```
class Demo_actionAction  
{
```

这个类包含了动作服务器的定义。

```
actionlib::SimpleActionServer<mastering_ros_demo_pkg::Demo_actionAction>  
as;
```

这是使用我们自定义的动作消息类型创建的一个简单的动作服务器实例。

```
mastering_ros_demo_pkg::Demo_actionFeedback feedback;
```

创建一个反馈实例以便在操作过程中可以发送反馈。

```
mastering_ros_demo_pkg::DemoActionResult result;
```

创建一个结果实例来发送最终的结果。

```
Demo_actionAction(std::string name) :  
    as(nh_, name, boost::bind(&Demo_actionAction::executeCB, this, _1),  
    false),  
    action_name(name)
```

这是一个动作构造函数，这里创建的动作服务器还包含了一些参数，例如Nodehandle、action_name和executeCB，其中executeCB是所有动作完成后的回调函数。

```
as.registerPreemptCallback(boost::bind(&Demo_actionAction::preemptCB,
this));
```

这行代码是当动作被抢占时注册一个回调函数。preemptCB是动作客户端发出抢占请求时执行的回调函数名。

```
void executeCB(const mastering_ros_demo_pkg::Demo_actionGoalConstPtr
&goal)
{
    if(!as.isActive() || as.isPreemptRequested()) return;
```

这是回调函数的定义。当动作服务器接收到目标值后就会执行该回调函数。它只有在检查了动作服务器的活动状态和是否已经被抢占后，才会执行该回调函数。

```
for(progress = 0 ; progress < goal->count; progress++){
    //Check for ros
    if(!ros::ok()){
        break;
    }
}
```

这个循环在目标值到达之前将会一直执行。它将不断地发送当前任务的进度作为反馈。

```
if(!as.isActive() || as.isPreemptRequested()){
    return;
}
```

在此循环内部，将检查动作服务器是处于活动状态还是处于被抢占状态。如果未处于活动状态，或者处于被抢占状态，该函数将返回。

```
if(goal->count == progress){  
    result.final_count = progress;  
    as.setSucceeded(result);  
}  
}
```

如果当前值到达了目标值，那么它将发布最终的结果。

```
Demo_actionAction demo_action_obj(ros::this_node::getName());
```

在main()函数中，我们创建了一个Demo_actionAction的实例，它将启动动作服务器。

创建ROS动作客户端

在本节中，我们将讨论动作客户端的工作方式。

demo_action_client.cpp是动作客户端节点的源文件，它负责发送目标值，即作为目标的一个数值。客户端从命令行的参数中获取目标值。客户端的第一个命令行参数是目标值，第二个参数是此任务的完成时间。

该目标值将被发送到服务器上，客户端将一直等待，直到到达指定的时间。时间以秒为单位。在等待指定时间后，客户端将检查任务是否完成，如果未完成，客户端将抢占该动作。

客户端代码有点长，因此我们仅讨论代码的重要部分：

```
#include <actionlib/client/simple_action_client.h>  
#include <actionlib/client/terminal_state.h>  
#include "mastering_ros_demo_pkg/Demo_actionAction.h"
```

在动作客户端中，我们需要包含actionlib/client/simple_action_client.h来获取动作客户端的API，这些API是用来实现动作客户端的。

```
actionlib::SimpleActionClient<mastering_ros_demo_pkg::Demo_actionAction>
ac("demo_action", true);
```

这行代码将创建一个动作客户端实例。

```
ac.waitForServer();
```

如果在系统上没有运行任何动作服务器，那么这行代码将等待无限长时间。只有在系统上运行了动作服务器时才会退出。

```
mastering_ros_demo_pkg::Demo_actionGoal goal;
goal.count = atoi(argv[1]);
ac.sendGoal(goal);
```

创建一个目标的实例，并从第一个命令行参数获取需要发送的目标值。

```
bool finished_before_timeout =
ac.waitForResult(ros::Duration(atoi(argv[2])));
```

此行将等待服务器的结果，一直到指定的等待秒数。

```
ac.cancelGoal();
```

如果还没有完成，该动作将被强占。

2.3.2 编译ROS动作服务器和客户端

在src文件夹中创建了这两个源码文件后，我们必须编辑 package.xml 和 CMakeLists.txt 才能编译生成这两个节点。

package.xml 文件应该包含消息生成和运行时的软件包，就像我们在创建ROS服务和消息中所做的那样。

我们必须在CMakeLists.txt 中包含Boost库才能编译生成这些节点。此外，我们也必须把为此示例编写的动作文件添加进去：

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  actionlib
  actionlib_msgs
  message_generation
)
```

我们需要在 find_package() 中添加 actionlib、actionlib_msgs 和 message_generation：

```
## System dependencies are found with CMake's conventions
find_package(Boost REQUIRED COMPONENTS system)
```

我们需要将 Boost 添加为系统依赖：

```
## Generate actions in the 'action' folder
add_action_files(
  FILES
  Demo_action.action
)
```

我们需要在`add_action_files()`中添加`.action`文件：

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

我们需要在`generate_messages()`中添加`actionlib_msgs`：

```
catkin_package(
  CATKIN_DEPENDS roscpp rospy std_msgs actionlib actionlib_msgs
  message_runtime
)

include_directories(
  include
  ${catkin_INCLUDE_DIRS}
  ${Boost_INCLUDE_DIRS}
)
```

我们必须在`include_directories`中包含`Boost`：

```
##Building action server and action client

add_executable(demo_action_server src/demo_action_server.cpp)
add_executable(demo_action_client src/demo_action_client.cpp)

add_dependencies(demo_action_server

mastering_ros_demo_pkg_generate_messages_cpp)
add_dependencies(demo_action_client
mastering_ros_demo_pkg_generate_messages_cpp)

target_link_libraries(demo_action_server ${catkin_LIBRARIES} )
target_link_libraries(demo_action_client ${catkin_LIBRARIES})
```

在catkin_make编译后，我们需要使用下面的命令来运行这些节点：

- 运行roscore：

```
$ roscore
```

- 启动action server节点：

```
$rosrun mastering_ros_demo_pkg demo_action_server
```

- 启动action client节点：

```
$ rosrun mastering_ros_demo_pkg demo_action_client 10 1
```

这些过程的输出如图2-7所示。

```
jcacace@robot:~/catkin_ws$ rosrun mastering_ros_demo_pkg demo_action_client 10 1
[ INFO] [1499861037.958432848]: Waiting for action server to start.
[ INFO] [1499861038.206812461]: Action server started, sending goal.
[ INFO] [1499861038.207104961]: Sending Goal [10] and Preempt time of [1]
[ INFO] [1499861039.209897255]: Action did not finish before the time out.
jcacace@robot:~/catkin_ws$
```

```
jcacace@robot:~
```

```
jcacace@robot:~$ rosrun mastering_ros_demo_pkg demo_action_server
[ INFO] [1499861036.234953391]: Starting Demo Action Server
[ INFO] [1499861038.209617808]: /demo_action is processing the goal 10
[ INFO] [1499861038.209949156]: Setting to goal 0 / 10
[ INFO] [1499861038.413934495]: Setting to goal 1 / 10
[ INFO] [1499861038.609803856]: Setting to goal 2 / 10
[ INFO] [1499861038.809718825]: Setting to goal 3 / 10
[ INFO] [1499861039.009985643]: Setting to goal 4 / 10
[ INFO] [1499861039.210416071]: Setting to goal 5 / 10
[ WARN] [1499861039.210567039]: /demo_action got preempted!
```

图2-7 运行ROS动作服务器和客户端

2.4 创建启动文件

ROS中的启动（launch）文件在启动多个节点时非常有用。在前面的示例中，我们看到最多有两个ROS节点，但想象一下，如果我们需要为一个机器人启动10个或20个节点的情形。如果我们在终端中逐个启动每个节点将是很麻烦的事。相反，我们可以在一个launch文件中基于XML格式编写所有的节点，可以使用roslaunch命令解析此文件，然后启动其中的所有节点。

roslaunch命令将自动启动ROS的节点管理器和参数服务器。因此，我们不需要再单独启动roscore命令和单个节点了。如果我们使用launch文件，那么所有的操作都将在一个命令中完成。

下面我们开始创建launch文件。进入软件包的文件夹中并创建一个名为demo_topic.launch的新启动文件，它将启动两个节点，分别是发布和订阅整型数值的ROS节点。我们需要将launch文件保存在软件包内的launch文件夹中：

```
$ rosdep mastering_ros_demo_pkg  
$ mkdir launch  
$ cd launch  
$ gedit demo_topic.launch
```

粘贴下面的内容到文件中：

```
<launch>
  <node name="publisher_node" pkg="mastering_ros_demo_pkg"
    type="demo_topic_publisher" output="screen"/>

  <node name="subscriber_node" pkg="mastering_ros_demo_pkg"
    type="demo_topic_subscriber" output="screen"/>
</launch>
```

我们讨论一下代码中的内容。<launch></launch>标签是launch文件中的根元素。所有的定义都在这对标签的内部。

<node>标签指明了要启动的节点：

```
<node name="publisher_node" pkg="mastering_ros_demo_pkg"
  type="demo_topic_publisher" output="screen"/>
```

<node>中的name标签表示节点的名称，pkg是软件包的名称，type是我们要启动的可执行文件的名称。

在创建了demo_topic.launch启动文件后，我们就可以使用下面的命令来启动它：

```
$ rosrun mastering_ros_demo_pkg demo_topic.launch
```

如果启动成功的话，我们将获得如图2-8所示的输出。

```
started roslaunch server http://robot:34091/  
  
SUMMARY  
=====
```

PARAMETERS

- * /rosdistro: kinetic
- * /rosversion: 1.12.7

NODES

- /
 - publisher_node (mastering_ros_demo_pkg/demo_topic_publisher)
 - subscriber_node (mastering_ros_demo_pkg/demo_topic_subscriber)

auto-starting new master
process[master]: started with pid [10348]
ROS_MASTER_URI=http://localhost:11311

图2-8 启动demo_topic.launch文件时的终端信息

我们可以使用下面的命令来查看节点列表：

```
$ rosnode list
```

我们还可以使用一个名为rqt_console的GUI工具来查看日志消息，并调试节点：

```
$ rqt_console
```

我们可以在此工具中看到由这两个节点生成的日志，如图2-9所示。

The screenshot shows the rqt_console application window titled "rqt_console_Console - rqt". The main area displays a table of log messages with the following columns: #, Message, Severity, Node, Stamp, Topics, and Location. There are three rows of data:

#	Message	Severity	Node	Stamp	Topics	Location
#1552	Received [878]	info	/subscriber_node	12:12:37.961994162 (2015-10-17)	/rosout	/home/robot/mastering_robots_ws/
#1551	628	info	/publisher_node	12:12:37.981201394 (2015-10-17)	/numbers /rosout	/home/robot/mastering_robots_ws/
#1550	Received [877]	info	/subscriber_node	12:12:37.062119738 (2015-10-17)	/rosout	/home/robot/mastering_robots_ws/

图2-9 使用rqt_console工具查看日志

2.5 话题、服务和动作库的应用

话题 (topic)、服务 (service) 和动作库 (actionlib) 可应用于不同的场景。我们知道话题是单向通信的，服务是具备请求/应答功能的双向通信，动作库是ROS服务的一种变化形式，我们可以在需要时取消在服务器上运行的进程。

下面是我们使用这些方法的一些场景。

- 话题：流式传输的连续数据流（传感器数据）。例如，流式传输数据以远程控制机器人，发布机器人里程计信息，从相机发布视频流等。
- 服务：执行能快速结束的程序。例如，保存传感器的标定参数，保存机器人在导航过程中生成的地图或加载参数配置文件等。
- 动作库：执行时间长、任务复杂、需要管理反馈的动作时使用。例如，向某个目的地导航时，或进行路径规划时。

可以从下面的Git仓库下载此项目的完整源码。完整的下载命令如下：

```
$ git clone https://github.com/jocacace/mastering_ros_demo_pkg.git
```

2.6 维护ROS软件包

大多数ROS软件包都使用版本控制系统（Version Control System, VCS）来维护，例如Git、Subversion(svn)、Mercurial(hg)等。共享公共VCS的一组软件包集合可称为代码仓库。可以使用名为blooom的catkin自动化发布工具来发布代码仓库中的软件包。大多数的ROS软件包都是以BSD许可的形式发布的。全球各地都有活跃的开发人员为ROS平台贡献代码。维护软件包代码对所有的软件都是很重要的，尤其是开源应用程序。这些开源软件由社区的开发人员维护，并提供支持。如果想要维护并接受其他开发人员的贡献代码，那么为我们的软件包创建版本控制系统是必不可少的。

前面的软件包已经在GitHub中更新了，可以在
https://github.com/jocacace/mastering_ros_demo_pkg查看该项目的源码。

2.7 发布ROS软件包

在GitHub中创建了ROS软件包后，我们就可以正式发布软件包了。在ROS中提供了名为bloom (<http://ros-infrastructure.github.io/bloom/>) 的工具来发布ROS软件包，在网站中提供了详细的操作步骤。Bloom是一个自动化发布工具，旨在从源代码项目中生成特定平台版本的软件包，但是Bloom需要与catkin一起使用。

发布软件包的前提条件如下：

- 安装Bloom工具
- 为当前软件包创建一个Git仓库
- 为发布版本创建一个空的Git仓库

在Ubuntu中使用下面的命令安装Bloom：

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

为当前软件包创建一个Git仓库，将其作为上传仓库。在这里，我们已经在https://github.com/jocacace/mastering_ros_demo_pkg创建了一个代码仓库。

在Git中为发布软件包创建一个空的仓库。此仓库被称为release仓库。我们创建了一个名为demo_pkg-release的软件包。

在完成这些前提条件后，我们就可以开始创建软件包的发布版本了。进入本地代码库mastering_ros_demo_pkg，并将软件包中的代码推送到Git上。然后在此本地代码库中打开一个终端并执行下面的命令：

```
$ catkin_generate_changelog
```

此命令的目的是在本地代码库中创建CHANGELOG.rst文件。在执行完此命令后，它将显示以下选项：

Continue without -all option [y/N]. 在这里输入 y

它将在本地代码库中创建一个CHANGELOG.rst文件。

在创建了日志文件后，我们就可以通过提交这些更改来更新Git仓库：

```
$ git add -A  
$ git commit -m 'Updated CHANGELOG.rst'  
$ git push -u origin master
```

2.7.1 准备发布ROS软件包

在本节中，我们将检查软件包是否包含更改日志、版本等信息。下面这条命令可以让我们在发布软件包前进行所有检查，从而保持内容一致。

这条命令需要在软件包的本地代码库中执行：

```
$ catkin_prepare_release
```

如果当前没有版本信息，那么该命令将设置一个版本标签，然后向上游代码库提交更改。

2.7.2 发布软件包

下面这条命令将启动发布。该命令的语法如下：

```
bloom-release --rosdistro <ros_distro> --track <ros_distro> repository_name  
$ bloom-release --rosdistro kinetic --track kinetic mastering_ros_demo_pkg
```

当执行这条命令时，它将跳转到 rosdistro (<https://github.com/ros/rosdistro>) 软件包仓库中获取软件包的详细信息。在ROS的rosdistro软件包中包含一个索引文件，其中包含了ROS中所有软件包的列表。目前，我们的软件包没有索引，因为这是第一个版本，但我们可以将我们的软件包详细信息添加到名为distributions.yaml的索引文件中。

当rosdistro中没有软件包的引用时，将显示如图2-10所示的信息。

```
ROS Distro index file associate with commit '43659b6409dcb545fd3d25c6d977f195cdf  
f886a'  
New ROS Distro index url: 'https://raw.githubusercontent.com/ros/rosdistro/43659  
b6409dcb545fd3d25c6d977f195cdff886a/index.yaml'  
Specified repository 'mastering_ros_demo_pkg' is not in the distribution file lo  
cated at 'https://raw.githubusercontent.com/ros/rosdistro/43659b6409dcb545fd3d25  
c6d977f195cdff886a/kinetic/distribution.yaml'  
Could not determine release repository url for repository 'mastering_ros_demo_pk  
g' of distro 'kinetic'  
You can continue the release process by manually specifying the location of the  
RELEASE repository.  
To be clear this is the url of the RELEASE repository not the upstream repositor  
y.  
For release repositories on GitHub, you should provide the 'https://' url which  
should end in '.git'.  
Here is the url for a typical release repository on GitHub: https://github.com/r  
os-gbp/rviz-release.git  
==> Looking for a release of this repository in a different distribution...  
A previous distribution, 'indigo', released this repository.  
Release repository url [https://github.com/qboticslabs/demo\_pkg-release.git]: ht  
tps://github.com/jocacace/demo_pkg-release.git
```

图2-10 当rosdistro中没有软件包引用时的消息

我们需要提供发布的仓库，见图2-10。在本例中，URL是
https://github.com/jocacace/demo_pkg-release，如图2-11所示。

在接下来的步骤中，发布向导将询问仓库名称、上游仓库名、URL等。我们可以提供这些选项需要的内容。最后，向rosdistro提交一个合并请求（Pull Request），如图2-12所示。

```
Given track 'kinetic' does not exist in release repository.  
Available tracks: []  
Create a new track called 'kinetic' now [Y/n]? Y  
Creating track 'kinetic'...  
Repository Name:  
upstream  
    Default value, leave this as upstream if you are unsure  
<name>  
    Name of the repository (used in the archive name)  
['upstream']: mastering_ros_demo_pkg  
Upstream Repository URI:  
<uri>  
    Any valid URI. This variable can be templated, for example an svn url  
    can be templated as such: "https://svn.foo.com/foo/tags/foo-:{version}"  
    where the :{version} token will be replaced with the version for this releas  
e.  
[None]: https://github.com/jocacace/mastering_ros_demo_pkg.git
```

图2-11 输入发布仓库的URL

```
==> Pulling latest rosdistro branch
remote: Counting objects: 99872, done.
remote: Compressing objects: 100% (38/38), done.
remote: Total 99872 (delta 35), reused 48 (delta 20), pack-reused 99809
Receiving objects: 100% (99872/99872), 29.62 MiB | 4.71 MiB/s, done.
Resolving deltas: 100% (64655/64655), done.
From https://github.com/ros/rosdistro
 * branch           master    -> FETCH HEAD
==> git reset --hard 43659b6409dcb545fd3d25c6d977f195cdff886a
HEAD is now at 43659b6 Merge pull request #15521 from trainman419/bloom-diagnos
tcs-32
==> Writing new distribution file: kinetic/distribution.yaml
==> git add kinetic/distribution.yaml
==> git commit -m "mastering_ros_demo_pkg: 0.0.3-0 in 'kinetic/distribution.yaml
' [bloom]"
[bloom-mastering_ros_demo_pkg-0 763d941] mastering_ros_demo_pkg: 0.0.3-0 in 'kin
etic/distribution.yaml' [bloom]
 1 file changed, 6 insertions(+)
==> Pushing changes to fork
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 458 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://7454b673dc9f5564070690111b8f170187884d73:x-oauth-basic@github.com/joc
acace/rosdistro.git
 * [new branch]      bloom-mastering_ros_demo_pkg-0 -> bloom-mastering_ros_demo_
pkg-0
<== Pull request opened at: https://github.com/ros/rosdistro/pull/15526
```

图2-12 向rosdistro发送一个合并请求

此软件包的合并请求可在如下链接中查看：

[https://github.com/ros/rosdistro/pull/15526。](https://github.com/ros/rosdistro/pull/15526)

如果合并请求被接受了，将被合并到`kinetic/distribution.yaml`文件中。该文件包含ROS中所有软件包的索引。

图2-13显示了当前软件包在`kinetic/distribution.yaml`中的索引。

```
6 亲和性 kinetic/distribution.yaml

diff --git a/kinetic/distribution.yaml b/kinetic/distribution.yaml
@@ -3531,6 +3531,12 @@ repositories:
3531 3531     release: release/kinetic/{package}/{version}
3532 3532     url: https://github.com/MarvelmindRobotics/marvelmind_nav-release.git
3533 3533     version: 1.0.6-0
+ 3534     + mastering_ros_demo_pkg:
+ 3535     +   release:
+ 3536     +   tags:
+ 3537     +   release: release/kinetic/{package}/{version}
+ 3538     +   url: https://github.com/jocacace/mastering_ros_demo_pkg.git
+ 3539     +   version: 0.0.3-0
3534 3540     mav_comm:
3535 3541     release:
3536 3542     packages:
```

图2-13 ROSKinetic的distribution.yaml文件

至此，我们就可以确认此软件包已被发布并正式添加到ROS索引中了。

2.7.3 为ROS软件包创建维基页面

ROS维基允许用户创建自己的主页来展示他们的软件包、机器人或传感器。ROS的官方维基页面是wiki.ros.org。现在，我们将为软件包创建一个维基页面。

i 下载示例代码：你可以从以下网址下载章节代码：

https://github.com/jocacace/master-ing_ros_2nd_ed.git，此代码库包含本书中所有章节的代码。

第一步是使用你的电子邮件在维基中注册。进入wiki.ros.org，然后单击Login按钮，如图2-14所示。



图2-14 在ROS维基中找到登录选项

点击了Login后，你就可以注册了。如果你已经注册过，那么就可以直接使用你的账号登录。登录后，点击维基页面右侧的用户名链接，如图2-15所示。

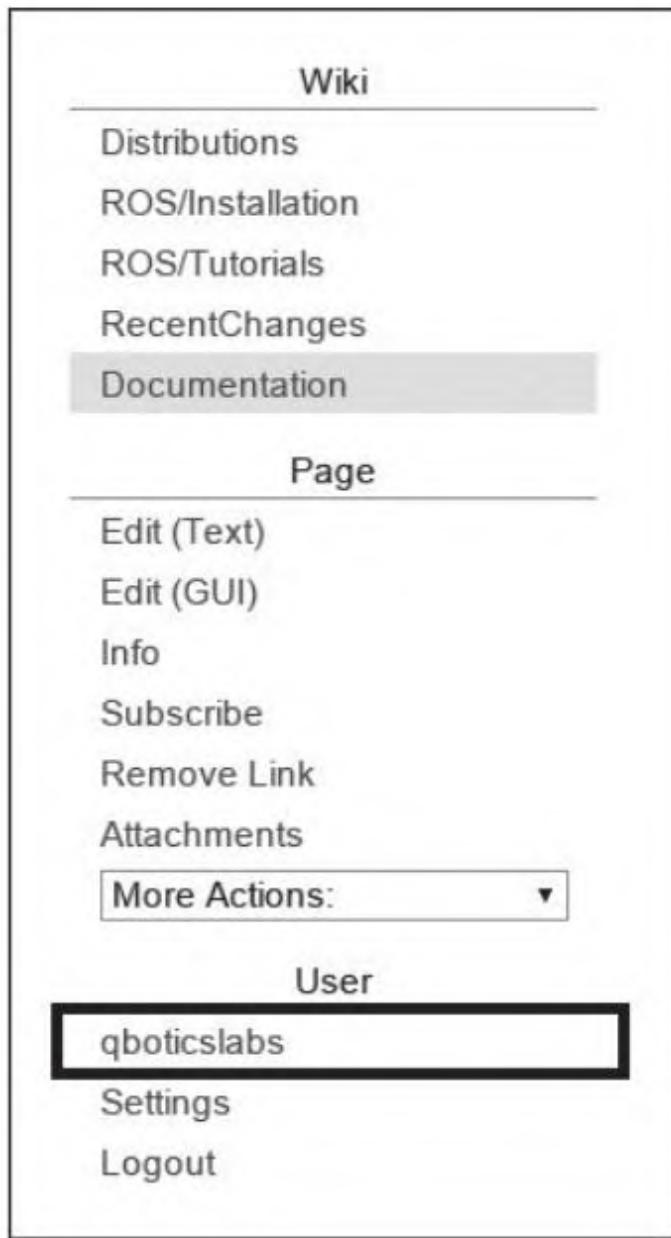


图2-15 从ROS维基中找到用户账户按钮

点击了此链接后，你就可以为自己的软件包创建主页了。你会看到一个带GUI的文本编辑器来填写内容。图2-16显示了我们为此软件包创建的页面。

qbotticslabs

ⓘ Thank you for your changes. Your attention to detail is appreciated.

[Clear message](#)

Mastering Robotics using ROS

Package Summary

A demo package which has example codes demonstrating topic, service, custom messages and actionlib

- Maintainer: Lentin Joseph <qbotticslabsAT gmail DOT com>
- Author : Lentin Joseph < qbotticslabs AT gmail DOT com>
- License : BSD
- Source : git [git](https://github.com/qbotticslabs/mastering_ros_demo_pkg.git)

1. Installation

You can use git clone to install package.

Wiki

Distributions

ROSInstallation

ROS/Tutorials

RecentChanges

Documentation

qbotticslabs

Page

[Edit \(Text\)](#)

[Edit \(GUI\)](#)

[info](#)

[Subscribe](#)

[Add Link](#)

[Attachments](#)

[More Actions](#) ▾

User

qbotticslabs

[Settings](#)

[Logout](#)

图2-16 创建一个新维基页面

2.8 习题

学习完本章后，现在你应该能够回答以下问题：

- ROS节点之间支持哪种通信协议？
- rosrun和roslaunch命令有何区别？
- ROS话题和ROS服务在运行时有何不同？
- ROS服务和ROS动作库的运行有何不同？

2.9 本章小结

在本章中，我们提供了ROS节点的不同示例，其中实现了ROS的话题、服务和动作等功能。我们还讨论了如何使用自定义和标准消息来创建与编译ROS软件包。演示了每个概念的工作原理后，我们将软件包上传到GitHub，然后为该软件包创建了一个维基页面。

下一章，我们将讨论使用URDF和xacro对ROS机器人进行建模，然后讲解如何设计机器人模型。

第3章

在ROS中为3D机器人建模

机器人制造的第一个阶段是设计和建模。这里我们可以使用CAD软件对一个机器人进行设计和建模，例如AutoCAD、SOLIDWORKS和Blender。机器人建模的主要目的之一就是仿真。

机器人仿真工具可以检查机器人设计中的关键缺陷，也可以确认机器人在进入制造阶段之前是否可以正常工作。

虚拟机器人模型必须具有真实硬件的所有特性。机器人的外形可能与实际的机器人相似，也可能与实际的机器人不相似，但它必须是抽象的，应具有实际机器人的所有物理特性。

在本章中，我们将讨论两类机器人的设计。一个是七自由度（7 Degrees of Freedom, 7-DOF）机械手臂，另一个是差速驱动机器人。在接下来的章节中，我们将讨论仿真，如何构建真实硬件以及与ROS的接口。

如果我们计划创建机器人的3D模型并使用ROS对其进行仿真，那么你需要了解一些有助于机器人设计的ROS软件包。出于各种不同的原因，在ROS中为机器人创建一个模型是很重要的。例如，我们可以使用这个模型来仿真并控制机器人，将其可视化，或者使用ROS工具获取机器人的结构及其运动的信息。

ROS有一个标准的超软件包robot_model，用于设计和创建机器人模型。它由一组软件包组成，其中一些软件包被称为urdf、kdl_parser、robot_state_publisher和collada_urdf。这些软件包可以帮助我们创建具有真实硬件准确特征的3D机器人模型描述。

本章将介绍以下内容：

- 用于机器人建模的ROS软件包
- 为机器人描述创建ROS软件包
- 使用URDF理解机器人建模
- 使用xacro理解机器人建模
- 将xacro转换为URDF

- 为一个7-DOF机器人机械臂创建机器人描述
- 关节状态发布和机器人状态发布协同工作
- 创建一个差速轮式机器人描述

3.1 机器人建模的ROS软件包

ROS提供了一些很有用的软件包，可以用来创建3D机器人模型。

在本节中，我们将讨论一些普遍用于创建机器人模型的重要的ROS软件包：

- urdf：机器人建模最重要的ROS软件包是urdf软件包。这个软件包包含一个用于统一机器人描述格式（Unified Robot Description Format, URDF）的C++解析器，它是一个表示机器人模型的XML文件。还有一些其他不同的组件来组成urdf：
- urdf_parser_plugin：这个软件包实现了写入URDF数据结构的方法；
- urdfdom_headers：此组件提供了使用urdf解析器的核心数据结构头文件；
- collada_parser：这个软件包通过解析Collada文件来填充数据结构；
- urdfdom：此组件通过解析URDF文件来填充数据结构；
- collada-dom：这是一个独立组件，可以使用Maya、Blender和Softimage等3D计算机图形软件对Collada文档进行转换。

我们可以使用URDF来定义机器人模型、传感器和工作环境，使用URDF解析器对其进行解析。我们只能使用URDF描述一个类似树状连杆结构的机器人，也就是说，机器人会有刚性连杆，并通过关节连接。我们无法用URDF表达柔性连杆。URDF由特殊的XML标签构成，我们可以

使用解析器程序解析这些XML标签以进行进一步处理。在后面的部分我们会使用URDF进行建模：

- joint_state_publisher：在使用URDF设计机器人模型时，该工具非常有用。这个软件包包含一个名为joint_state_publisher的节点，该节点读取机器人模型描述，查找所有关节，并利用GUI滑动条将关节值向所有非固定关节发布。用户可以用该工具与机器人的每个关节进行交互，也可以使用RViz进行可视化。在设计URDF时，用户可以使用此工具验证每个关节的旋转和平移。我们将在后面详细讨论joint_state_publisher节点及其用法。
- kdl_parser：运动学和动力学库（Kinematic and Dynamics Library，KDL）是一个ROS软件包，它包含解析器工具，可以从URDF表示中构建KDL树。运动学树可以用来发布关节的状态，也可以用来表示机器人的正/逆运动学。
- robot_state_publisher：该软件包可以读取当前机器人的关节状态，并发布由URDF构建的运动学树中每个机器人连杆的3D姿态。机器人的3D姿态以ROS tf(transform)的形式发布。tf发布了机器人各坐标系之间的关系。
- xacro：Xacro是XML Macros的缩写，xacro可理解为URDF加上若干插件。xacro包含一些插件，让URDF变得更简短、可读性更好，并且可以用于构建复杂的机器人描述。利用ROS相应的工具，可以在需要的时候将xacro转换为URDF。我们将在以后的章节学习xacro及其用法。

3.2 利用URDF理解机器人建模

我们已经讨论了urdf软件包。在本节中，我们将进一步研究URDF XML标签，它有助于进行机器人建模。我们必须创建一个文件，并在其中编写机器人的每个连杆和关节之间的连接关系，并用.urdf扩展名保存文件。

URDF可以表示机器人的运动学和动态描述，机器人的视觉以及机器人的碰撞模型。

下面是组成URDF机器人模型常用的URDF标签：

- link：link标签表示机器人的单个连杆。使用此标签，我们可以为机器人的一个连杆及其属性建模。该模型包括大小、形状和颜色，甚至可以导入一个3D网格来表示机器人的连杆。我们还可以提供连杆的动态属性，如惯性矩阵和碰撞属性。

语法如下：

```
<link name="
```

图3-1是单个连杆的表示。visual区域代表机器人的真实连杆，将真实连杆包围的面积是collision区域。collision部分包围了真实连杆，可以在与真实连杆发生碰撞之前检测到碰撞。

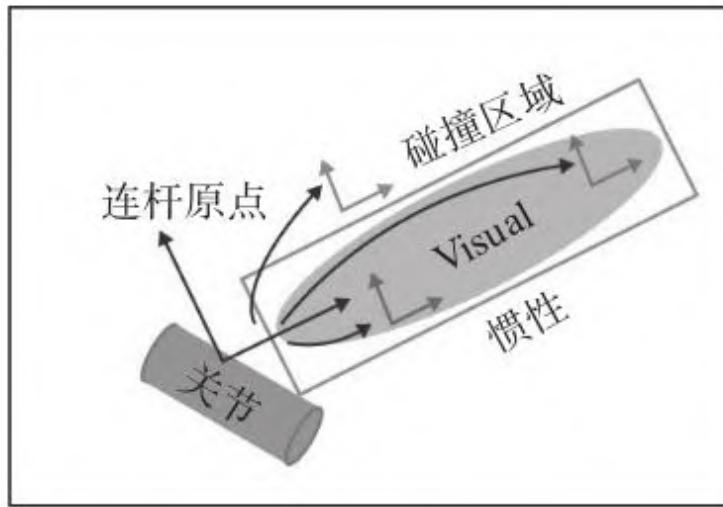


图3-1 URDF连杆的可视化

· joint: joint标签表示机器人关节。我们可以指定关节的运动学和动力学，并设定关节运动和速度的极限。joint标签支持不同类型的关节，如旋转关节 (revolute)、无限位旋转关节 (continuous)、滑动关节 (prismatic)、固定关节 (fixed)、浮动关节 (floating) 和平面关节 (planar)。

语法如下：

```
<joint name="
```

在两个连杆之间会形成一个URDF关节。第一个被称为父连杆 (parent)，第二个被称为子连杆 (child)。图3-2是一个关节及其连杆的示意图：

- robot: 这个标签封装了用URDF表示的整个机器人模型。在robot标签内，我们可以定义机器人的名字、各个连杆，以及机器人的各关节。

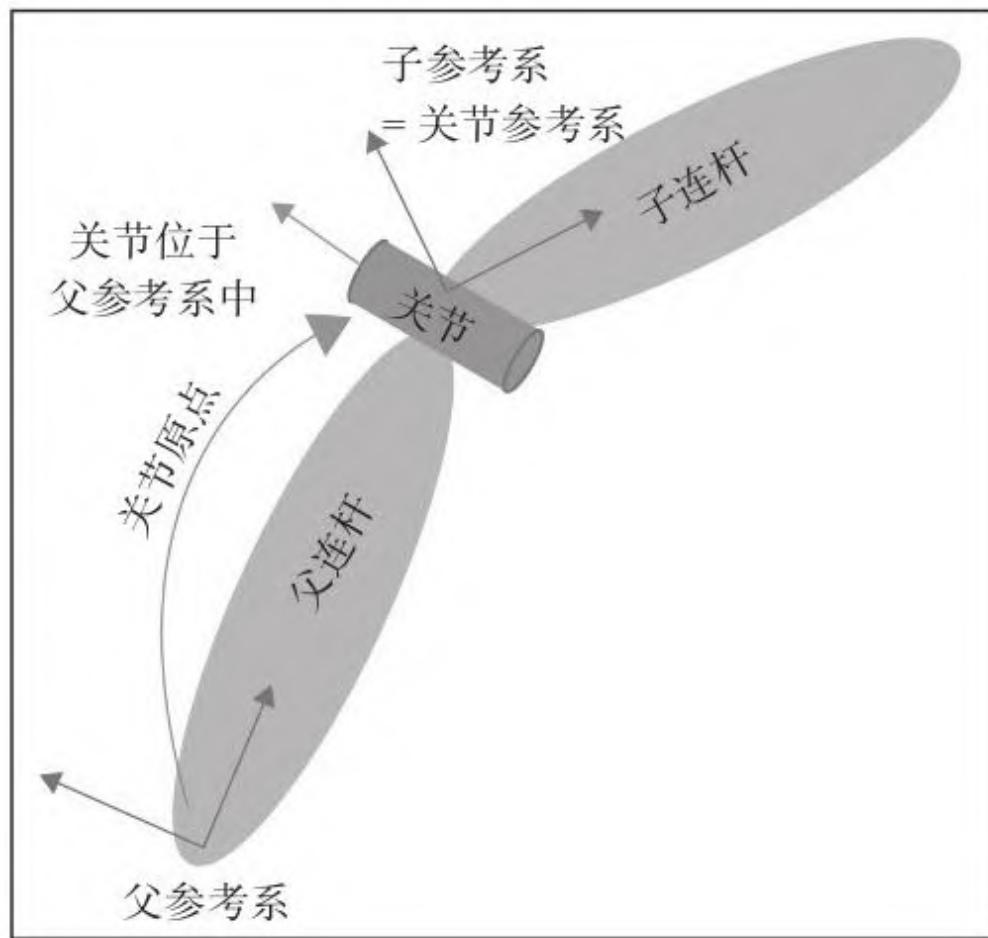


图3-2 URDF关节的可视化

语法如下：

```
<robot name="">
  <link> .... </link>
  <link> ..... </link>

  <joint> ..... </joint>
  <joint> ..... </joint>
</robot>
```

机器人模型由连接的连杆和关节组成。图3-3是一个机器人模型的可视化效果。

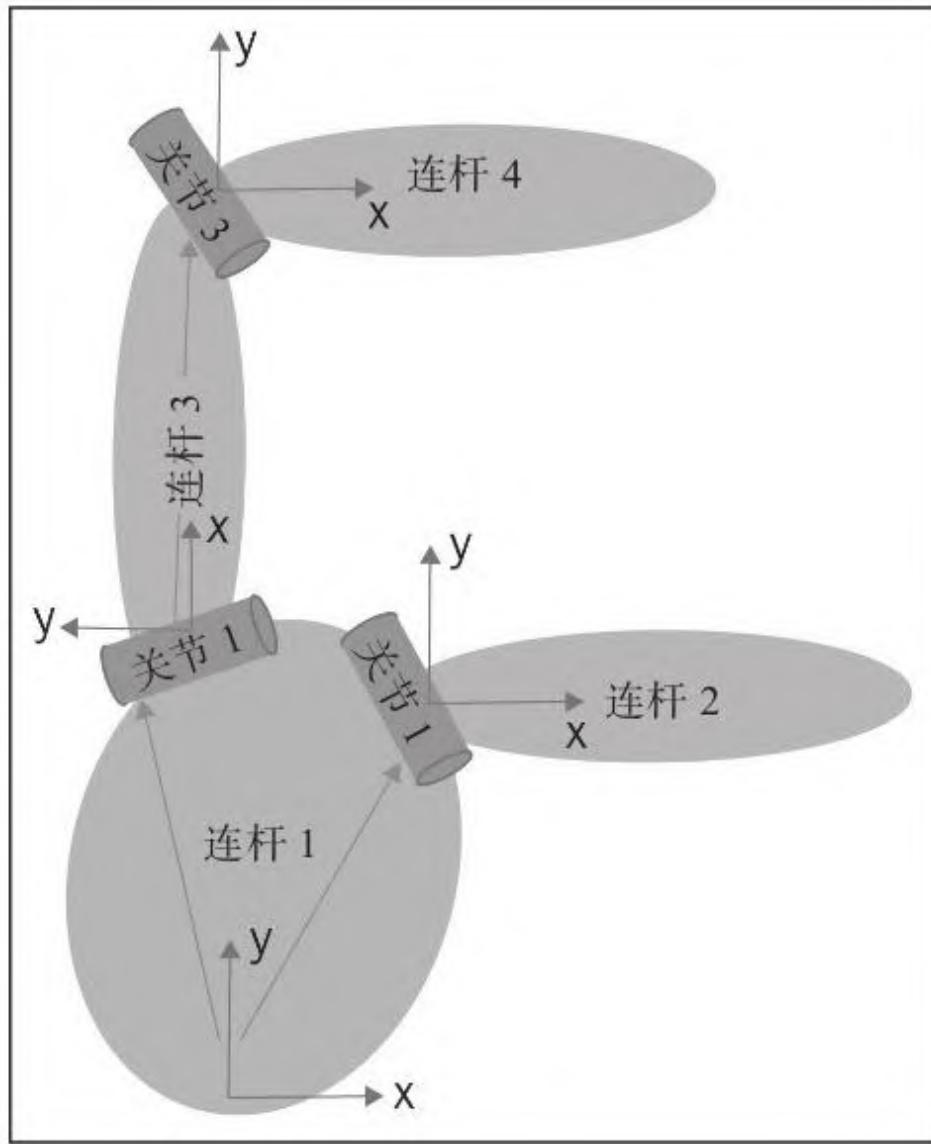


图3-3 由连杆和关节构成的机器人模型的可视化

· gazebo：当我们在URDF中包含Gazebo仿真器的仿真参数时，使用这个标签。我们可以用这个标签包含gazebo插件、gazebo材料属性等。以下是一个使用gazebo标签的例子：

```
<gazebo reference="link_1">
  <material>Gazebo/Black</material>
</gazebo>
```

你可以在<http://wiki.ros.org/urdf/XML>中找到更多的URDF标签。

3.3 为机器人描述创建ROS软件包

为机器人创建URDF文件之前，让我们在catkin工作区中使用如下命令创建一个ROS软件包来保存机器人模型：

```
$ catkin_create_pkg mastering_ros_robot_description_pkg roscpp tf  
geometry_msgs urdf rviz xacro
```

该软件包主要依赖于urdf和xacro软件包。如果这些软件包尚未安装在你的系统上，可以使用软件包管理器来安装它们：

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

我们可以在这个软件包中创建机器人的URDF文件，并创建启动文件以在RViz中显示创建的URDF。完整的软件包可以在下面的Git库中找到，你可以下载Git库，参考该软件包的实现，或者你也可以从本书附带的源码中获得该软件包：

```
$ git clone  
https://github.com/jocacace/mastering\_ros\_robot\_description\_pkg.git
```

在为这个机器人创建URDF文件之前，让我们先在软件包文件夹下创建三个文件夹：urdf、meshes和launch。urdf文件夹可以用来保存

我们将要创建的urdf和xacro文件。meshes文件夹保存我们需要包含在urdf文件中的网格模型。launch文件夹用来保存ROS的启动文件。

3.4 创建我们的第一个URDF模型

在学习了URDF及其重要的标签之后，我们可以使用URDF开始创建一些基本的模型。我们设计的第一个机器人模型是pan-and-tilt机械结构，如图3-4所示。

该机械结构有三个连杆和两个关节。底座连杆是不动的，其他所有连杆都安装在上面。第一个关节可以沿轴平移，第二个连杆安装在第一个连杆上，可以在轴上倾斜。这个系统中的两个关节都是旋转关节：

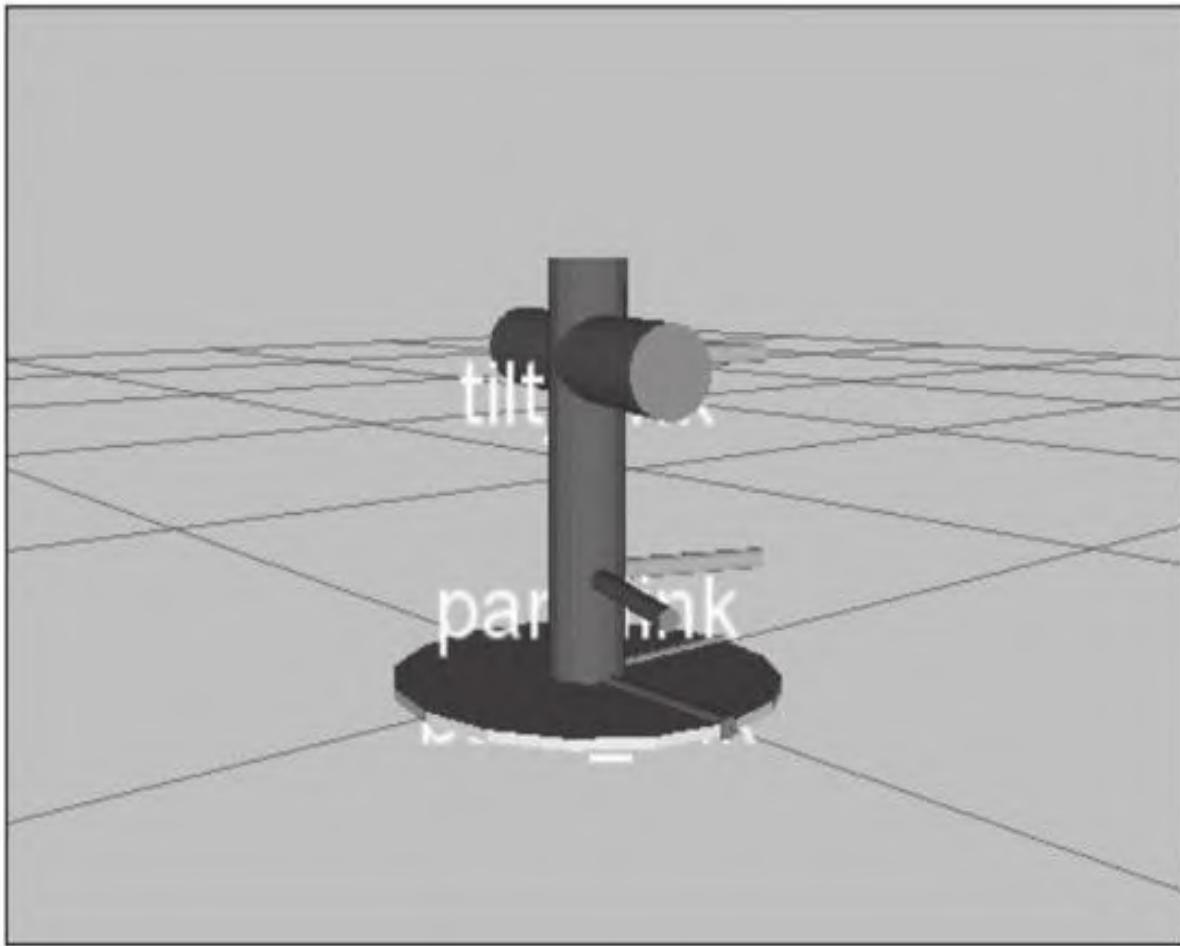


图3-4 在RViz中显示pan-and-tilt模型

让我们看看这个模型的URDF代码。切换到
`mastering_ros_robot_description_pkg/urdf`文件夹并打开
`pan_tilt.urdf`:

```
<?xml version="1.0"?>
<robot name="pan_tilt">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.01" radius="0.2"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <material name="yellow">
        <color rgba="1 1 0 1"/>
      </material>
    </visual>
  </link>
  <joint name="pan_joint" type="revolute">
    <parent link="base_link"/>
    <child link="pan_link"/>
    <origin xyz="0 0 0.1"/>
    <axis xyz="0 0 1" />
  </joint>
  <link name="pan_link">
    <visual>
      <geometry>
        <cylinder length="0.4" radius="0.04"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0.09"/>
      <material name="red">
        <color rgba="0 0 1 1"/>
      </material>
    </visual>
```

```
</link>
<joint name="tilt_joint" type="revolute">
  <parent link="pan_link"/>
  <child link="tilt_link"/>
  <origin xyz="0 0 0.2"/>
  <axis xyz="0 1 0" />
</joint>
<link name="tilt_link">
  <visual>
    <geometry>
<cylinder length="0.4" radius="0.04"/>
    </geometry>
    <origin rpy="0 1.5 0" xyz="0 0 0"/>
    <material name="green">
      <color rgba="1 0 0 1"/>
    </material>
  </visual>
</link>
</robot>
```

3.5 详解URDF文件

当检查代码时，我们可以在描述的顶部添加一个`<robot>`标签：

```
<?xml version="1.0"?>
<robot name="pan_tilt">
```

`<robot>`标签定义了我们将要创建的机器人的名称。在这里，我们将机器人命名为`pan_tilt`。

如果我们查看`<robot>`标签定义之后的部分，我们可以看到`pan-and-tilt`机械结构中连杆和关节的定义：

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.01" radius="0.2"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <material name="yellow">
      <color rgba="1 1 0 1"/>
    </material>
  </visual>
</link>
```

前面的这段代码是pan-and-tilt机械结构的base_link的定义。<visual>标签描述了连杆的可见外观，它将在机器人仿真中显示出来。我们可以用这个标签来定义连杆的几何形状（圆柱、立方体、球体或网格模型）以及连杆的材质（颜色和纹理）。

```
<joint name="pan_joint" type="revolute">
  <parent link="base_link"/>
  <child link="pan_link"/>
  <origin xyz="0 0 0.1"/>
  <axis xyz="0 0 1" />
</joint>
```

在前面的代码片段中，我们定义了一个具有唯一名称和关节类型的关节。在这里我们使用的关节类型是旋转关节（revolute），父连杆和子连杆分别是base_link和pan_link。在此标签内还指定了关节原点。

将前面的URDF代码保存为pan_tilt.urdf，并使用以下命令检查urdf是否包含错误：

```
$ check_urdf pan_tilt.urdf
```

check_urdf命令将解析urdf标签并显示错误（如果有的话）。如果一切正常，它将输出如下内容：

```
robot name is: pan_tilt
----- Successfully Parsed XML -----
root Link: base_link has 1 child(ren)
  child(1): pan_link
    child(1): tilt_link
```

如果我们想以图形化的方式查看机器人连杆和关节的结构，我们可以使用一个名为urdf_to_graphiz的命令行工具：

```
$ urdf_to_graphiz pan_tilt.urdf
```

这个命令将生成两个文件：pan_tilt.gv和pan_tilt.pdf。我们可以使用以下命令来查看此机器人的结构：

```
$ evince pan_tilt.pdf
```

我们将得到如图3-5的输出。

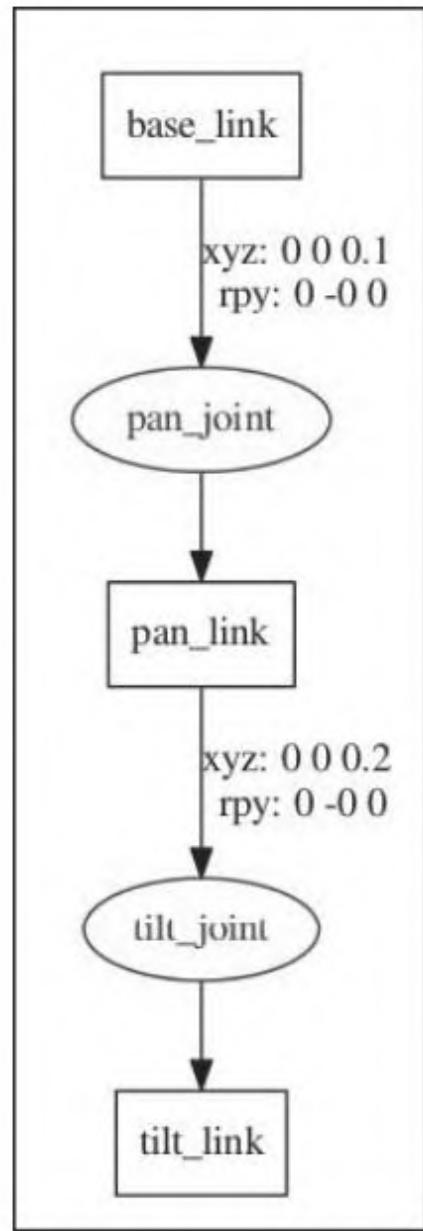


图3-5 pan-and-tilt机械结构的关节和连杆图

3.6 在RViz中可视化机器人3D模型

设计好URDF后，可以在RViz上查看它。我们可以创建一个view_demo.launch启动文件，并将下面的代码存放到该文件中，然后将该文件放入launch文件夹。切换到mastering_ros_robot_description_pkg/launch文件夹中来获取代码：

```
<launch>
  <arg name="model" />
  <param name="robot_description" textfile="$(find
mastering_ros_robot_description_pkg)/urdf/pan_tilt.urdf" />
  <param name="use_gui" value="true"/>
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mastering_ros_robot_description_pkg)/urdf.rviz" required="true" />
</launch>
```

我们可以使用以下命令来启动模型：

```
$ rosrun mastering_ros_robot_description_pkg view_demo.launch
```

如果一切正常，我们将在RViz中看到这个pan-and-tilt机械结构，如图3-6所示。

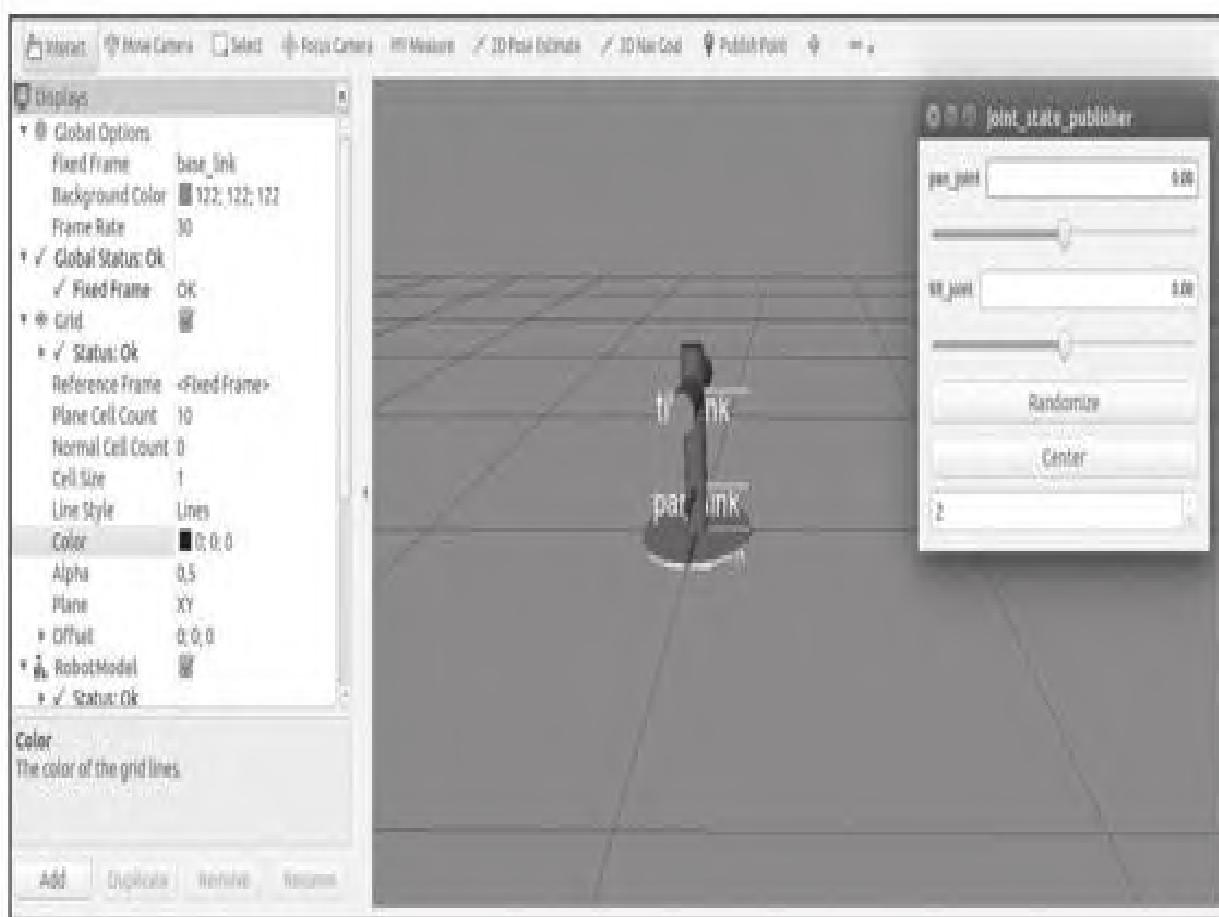


图3-6 pan-and-tilt机械结构的关节层

与pan-and-tilt关节的交互

我们可以看到一个附加的GUI窗口和RViz一起弹出，它包含控制平移（pan）关节和倾斜（tilt）关节的滑动条。这个GUI称为关节状态发布者节点，来自joint_state_publisher软件包。

```
<node name="joint_state_publisher" pkg="joint_state_publisher"
      type="joint_state_publisher" />
```

我们可以使用此语句将这个节点包含在启动文件中。同时需要在joint标签内提及pan-and-tilt的活动极限范围：

```
<joint name="pan_joint" type="revolute">
  <parent link="base_link"/>
  <child link="pan_link"/>
  <origin xyz="0 0 0.1"/>
  <axis xyz="0 0 1" />
  <limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
  <dynamics damping="50" friction="1"/>
</joint>
```

<limit effort="300" velocity="0.1" lower="-3.14" upper="3.14" />定义了受力极限、速度、旋转角的限位值。effort是该关节所能支撑的最大力度受力。lower和upper表示转动关节旋转范围的上下限位值（以弧度为单位），滑动关节移动距离的上下界（以米为单位）。velocity为关节的最大速度：

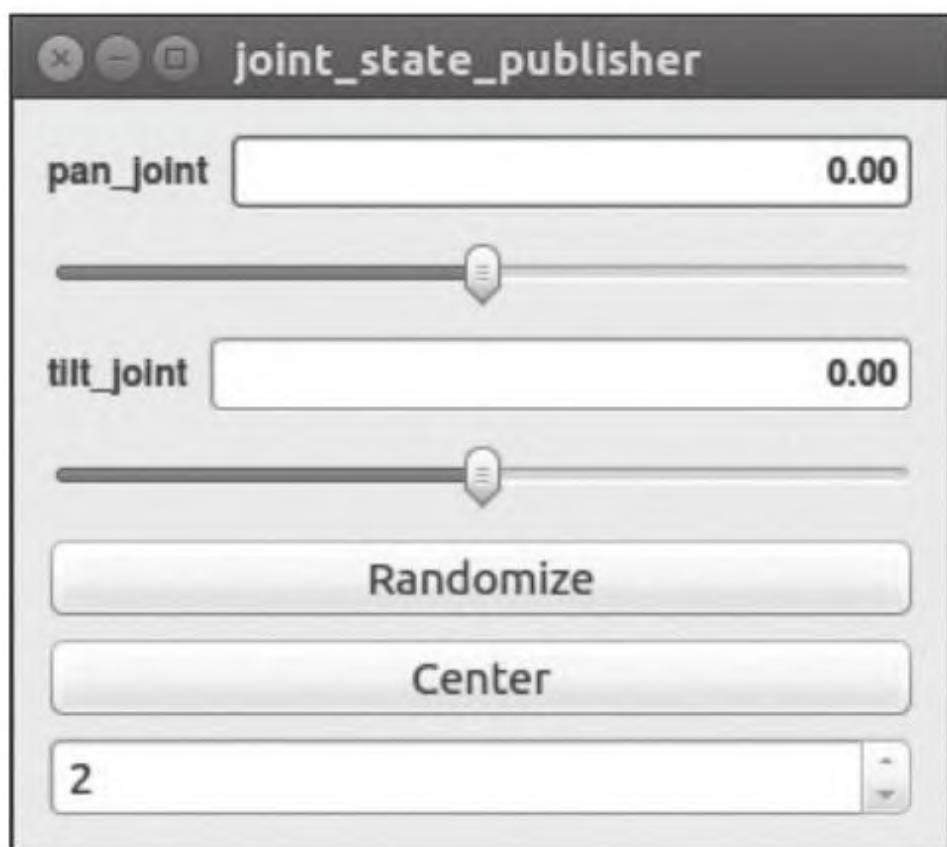


图3-7 pan-and-tilt机械结构的关节状态发布者

图3-7显示了关节状态发布者（joint_state_publisher）的GUI窗口，显示了滑动条，方框中显示了当前关节值。

3.7 向URDF模型添加物理属性和碰撞属性

在机器人仿真器中（如Gazebo或V-REP），对机器人仿真之前，我们需要定义机器人连杆的物理属性，如几何形状、颜色、质量、惯性，以及连杆的碰撞属性。

只有在机器人模型中定义所有这些属性，才能得到良好的仿真效果。URDF提供了标签来包含所有这些参数，在base_link的代码片段中包含了这些属性值，如下所示：

```
<link>
.....
<collision>
    <geometry>
        <cylinder length="0.03" radius="0.2"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
</collision>

<inertial>
    <mass value="1"/>
    <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
.....
</link>
```

这里，我们将碰撞几何定义为圆柱体，将质量定义为1kg，我们还设置了连杆的惯性张量。

每个连杆都需要设置碰撞（collision）和惯性（inertial）参数，否则Gazebo将无法正确加载机器人模型。

3.8 利用xacro理解机器人建模

当我们创建复杂的机器人模型时，URDF的灵活性将会降低。URDF缺少的主要特性是简单性、可重用性、模块化和可编程性。

如果有人想在机器人描述中重复使用URDF代码块10次，他可以复制并粘贴10次。如果有一个选项可以使用这个代码块并可以在不同的设置下复制多个副本，那么该选项在创建机器人描述时将非常有用。

URDF是一个单独的文件，我们不能在它里面包含其他的URDF文件。这降低了代码的模块化特性。所有代码都必须放在一个文件中，这会降低代码的简单性。

此外，如果在描述语言中存在一些可编程性，如添加变量、常量、数学表达式和条件语句，那么就会更具用户友好性。

使用xacro的机器人模型将满足所有这些条件。xacro的一些主要特点如下：

- 简化URDF：xacro是URDF的高级版本。它在机器人描述中创建宏并重用宏。这可以减少代码长度。此外，它还可以包含来自其他文件的宏，使代码更简单、更易读和更模块化。
- 可编程性：xacro语言在其描述中支持简单的编程语句。有变量、常量、数学表达式、条件语句等使描述更加智能和高效。

我们可以说xacro是URDF的升级版本，在需要的时候，我们可以利用ROS工具将xacro定义转换为URDF。

我们来讨论如何用xacro创建与pan-and-tilt相同的描述。转到
mastering_ros_robot_description_pkg/urdf文件夹，文件名是
pan_tilt.xacro。我们需要对xacro文件使用.xacro扩展，而不是
.urdf。以下是xacro代码的说明：

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="pan_tilt">
```

这两行指定了解析xacro文件所需的所有xacro文件的命名空间。
在指定命名空间后，我们需要添加xacro文件的名称。

3.8.1 使用属性

使用xacro，我们可以在xacro文件中声明常量或属性，即以名称表示的值，这些常量或属性可以在代码中的任何地方使用。常量的主要用途是，避免在连杆和关节上提供硬编码的值，而是保持一些常量，这样就更容易更改这些值，而不是查找并替换这些硬编码的值。

这里给出了一个使用属性的例子。我们声明了基座连杆和平移连杆的长度和半径。因此，在这里很容易改变尺寸而不是改变每个尺寸的值：

```
<xacro:property name="base_link_length" value="0.01" />
<xacro:property name="base_link_radius" value="0.2" />

<xacro:property name="pan_link_length" value="0.4" />
<xacro:property name="pan_link_radius" value="0.04" />
```

我们可以通过下面的定义，使用变量的值来替换硬编码的值：

```
<cylinder length="${pan_link_length}"
radius="${pan_link_radius}" />
```

在这里，旧值0.4被替换为{pan_link_length}，0.04被替换为{pan_link_radius}。

3.8.2 使用数学表达式

我们可以使用基本操作（如+、-、*、/、一元求负运算和括号）在\${} 中构建数学表达式。目前还不支持指数和模数。下面是一个代码中使用简单的数学表达式的例子：

```
<cylinder length="${pan_link_length}"  
radius="${pan_link_radius+0.02}"/>
```

3.8.3 使用宏

xacro的一个主要特性就是它支持宏（macro）。我们可以使用宏来减少复杂定义的长度。这是一个我们在惯性代码中使用的宏定义：

```
<xacro:macro name="inertial_matrix" params="mass">
  <inertial>
    <mass value="${mass}" />
    <inertia ixx="0.5" ixy="0.0" ixz="0.0"
              iyy="0.5" iyz="0.0" izz="0.5" />
  </inertial>
</xacro:macro>
```

这里，宏被命名为inertial_matrix，它的参数是mass。mass参数可以在惯性定义中使用\${mass}。我们可以用一行命令来替换每个惯性码，如下所示：

```
<xacro:inertial_matrix mass="1"/>
```

与URDF相比，xacro定义提高了代码可读性并减少了行数。接下来，我们将学习如何将xacro转换为URDF文件。

3.9 将xacro转换为URDF

设计完xacro文件之后，我们可以使用以下命令将其转换为URDF文件：

```
$ rosrun xacro xacro pan_tilt.xacro --inorder > pan_tilt_generated.urdf
```

最近在ROS中引入了--inorder选项以增加转换工具的功能。它允许我们按阅读顺序处理文件，比旧版本的ROS添加了更多功能。

我们可以在ROS启动文件中使用下面的命令将xacro转换为URDF，并将其作为robot_description的参数：

```
<param name="robot_description" command="$(find xacro)/xacro --inorder  
$(find mastering_ros_robot_description_pkg)/urdf/pan_tilt.xacro"  
/>
```

我们可以通过制作启动文件来查看pan-and-tilt的xacro文件，并使用以下命令来启动它：

```
$ roslaunch mastering_ros_robot_description_pkg  
view_pan_tilt_xacro.launch
```

3.10 为7-DOF机械臂创建机器人描述

现在，我们可以使用URDF和xacro创建一些复杂的机器人了。我们要创建的第一个机器人是一个7-DOF机械臂，它是一个有着多个串联连杆的机械臂。7-DOF手臂在运动学上是冗余的，这意味着它有更多的关节和自由度来到达它的目标位置和姿态。这个冗余机械臂的优点是我们可以对所需的目标位置和姿态有更多关节配置。它将提高机器人运动的灵活性和通用性，并能在机器人工作空间内实现有效的无碰撞运动。

让我们开始创建7-DOF机械臂。机器人手臂的最终输出模型如图3-8所示（图中还标注了机器人的各个关节和连杆）。

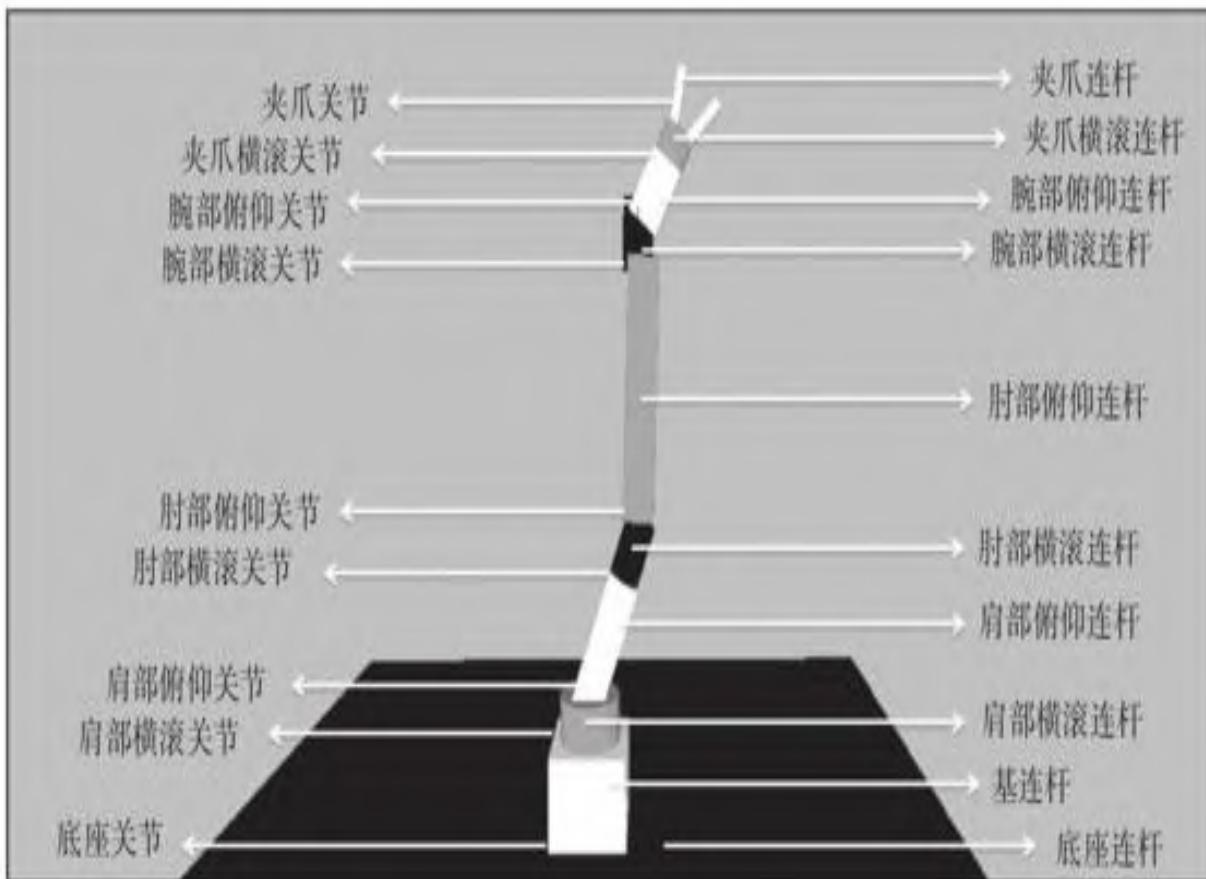


图3-8 7-DOF机械臂的关节和连杆

前面的这个机械臂用xacro格式描述。我们可以从代码库中获取该描述文件。我们可以转到下载的软件包中的urdf文件夹中，并打开seven_dof_arm.xacro文件。我们将复制描述并将其粘贴到当前软件包中，然后开始讨论这个机械臂描述的主要部分。

3.10.1 机械臂规格

以下是该7-DOF机械臂的规格：

- 自由度： 7
- 臂长： 50cm
- 工作半径： 35cm
- 连杆数： 12
- 关节数： 11

3.10.2 关节类型

以下是包含关节名称和机器人类型的关节列表：

关节号	关节名	关节类型	角的范围(度)
1	bottom_joint	固定	-
2	shoulder_pan_joint	转动	-150° ~ 114°
3	shoulder_pitch_joint	转动	-67° ~ 109°
4	elbow_roll_joint	转动	-150° ~ 41°
5	elbow_pitch_joint	转动	-92° ~ 110°
6	wrist_roll_joint	转动	-150° ~ 150°
7	wrist_pitch_joint	转动	92° ~ 113°
8	gripper_roll_joint	转动	-150° ~ 150°
9	finger_joint1	滑动	0 ~ 3cm
10	finger_joint2	滑动	0 ~ 3cm

我们使用上述规格设计了机械臂的xacro文件，接下来进行xacro手臂文件的解析。

3.11 解析7-DOF机械臂的xacro模型

我们将在这个机器人上定义10个连杆和9个关节，在机器人夹爪中定义2个连杆和2个关节。

从xacro的定义开始查看：

```
<?xml version="1.0"?>
<robot name="seven_dof_arm" xmlns:xacro="http://ros.org/wiki/xacro">
```

因为我们正在编写xacro文件，为了解析该文件，应该提一下xacro的命名空间。

3.11.1 使用常量

在xacro中使用常量可以让机器人的描述更简短、更可读。在这里，我们定义了每个连杆的角度 - 弧度换算系数、PI值、长度、高度和宽度：

```
<property name="deg_to_rad" value="0.01745329251994329577"/>
<property name="M_PI" value="3.14159"/>
<property name="elbow_pitch_len" value="0.22" />
<property name="elbow_pitch_width" value="0.04" />
<property name="elbow_pitch_height" value="0.04" />
```

3.11.2 使用宏

在以下代码中定义宏，避免重复并使代码更简短。下面是在这段代码中使用的宏：

```
<xacro:macro name="inertial_matrix" params="mass">
  <inertial>
    <mass value="${mass}" />
    <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="0.5" iyz="0.0"
    izz="1.0" />
  </inertial>
</xacro:macro>
```

这是惯性矩阵宏的定义，其中我们用mass作为其参数。

```
<xacro:macro name="transmission_block" params="joint_name">
  <transmission name="tran1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="${joint_name}">
      <hardwareInterface>PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
      <hardwareInterface>PositionJointInterface</hardwareInterface>
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
</xacro:macro>
```

在这段代码中，我们可以看到使用transmission（传动）标签的定义。

transmission标签将关节与执行机构相关联。它定义了我们在某一关节中使用的传动类型，马达的类型及其参数。它还定义了我们与

ROS控制器通信时使用的硬件接口的类型。

3.11.3 包含其他xacro文件

我们可以通过使用xacro: include标签包含传感器的xacro定义来扩展xacro的功能。下面的代码片段展示了如何在机器人xacro中包含传感器定义：

```
<xacro:include filename="$(find  
mastering_ros_robot_description_pkg)/urdf/sensors/xtion_pro_live.urdf.xacro  
"/>
```

在这里，我们包含了一个叫作Asus Xtion pro的传感器的xacro定义，当解析xacro文件时，这个定义将被展开。

使用“\$(find
mastering_ros_robot_description_pkg)/urdf/sensors/xtion_pro_live.urdf.xacro”，我们就可以访问传感器的xacro定义，其中find是查找当前mastering_ros_robot_description_pkg软件包的位置。

我们将在第11章“差速驱动移动机器人”中讨论有关视觉处理更多的内容。

3.11.4 在连杆中使用网格模型

我们可以在连杆中插入一个基本的形状，或者使用mesh标签插入一个网格模型文件。下面的例子展示了如何在视觉传感器中插入一个网格模型：

```
<visual>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <geometry>
    <mesh filename=
"package://mastering_ros_robot_description_pkg/meshes/sensors/xtion_pro_liv
e/xtion_pro_live.dae"/>
  </geometry>
  <material name="DarkGrey"/>
</visual>
```

3.11.5 使用机器人夹爪

机器人夹爪用于抓取和放置物体。夹爪属于简单的连接类型，它有2个关节，每个关节都是滑动关节。以下是一个夹爪关节的joint定义：

```
<joint name="finger_joint1" type="prismatic">
  <parent link="gripper_roll_link"/>
  <child link="gripper_finger_link1"/>
  <origin xyz="0.0 0 0" />
  <axis xyz="0 1 0" />
    <limit effort="100" lower="0" upper="0.03" velocity="1.0"/>
    <safety_controller k_position="20"
      k_velocity="20"
      soft_lower_limit="${-0.15 }"
      soft_upper_limit="${ 0.0 }"/>
  <dynamics damping="50" friction="1"/>
</joint>
```

在这里，夹爪的第一个关节由gripper_roll_link和gripper_finger_link1构成，第二个关节由gripper_roll_link和gripper_finger_link2构成。

图3-9显示了夹爪的关节是如何连接在gripper_roll_link上的。

3.11.6 在RViz中查看7-DOF机械臂

讨论完机器人模型后，我们可以在RViz中查看设计好的xacro文件，并使用关节状态发布者（joint state publisher）节点控制每个关节，使用机器人状态发布者（robot state publisher）节点发布机器人状态。

可以使用名为view_arm.launch的启动文件执行上述任务。该文件位于此软件包的launch文件夹内：

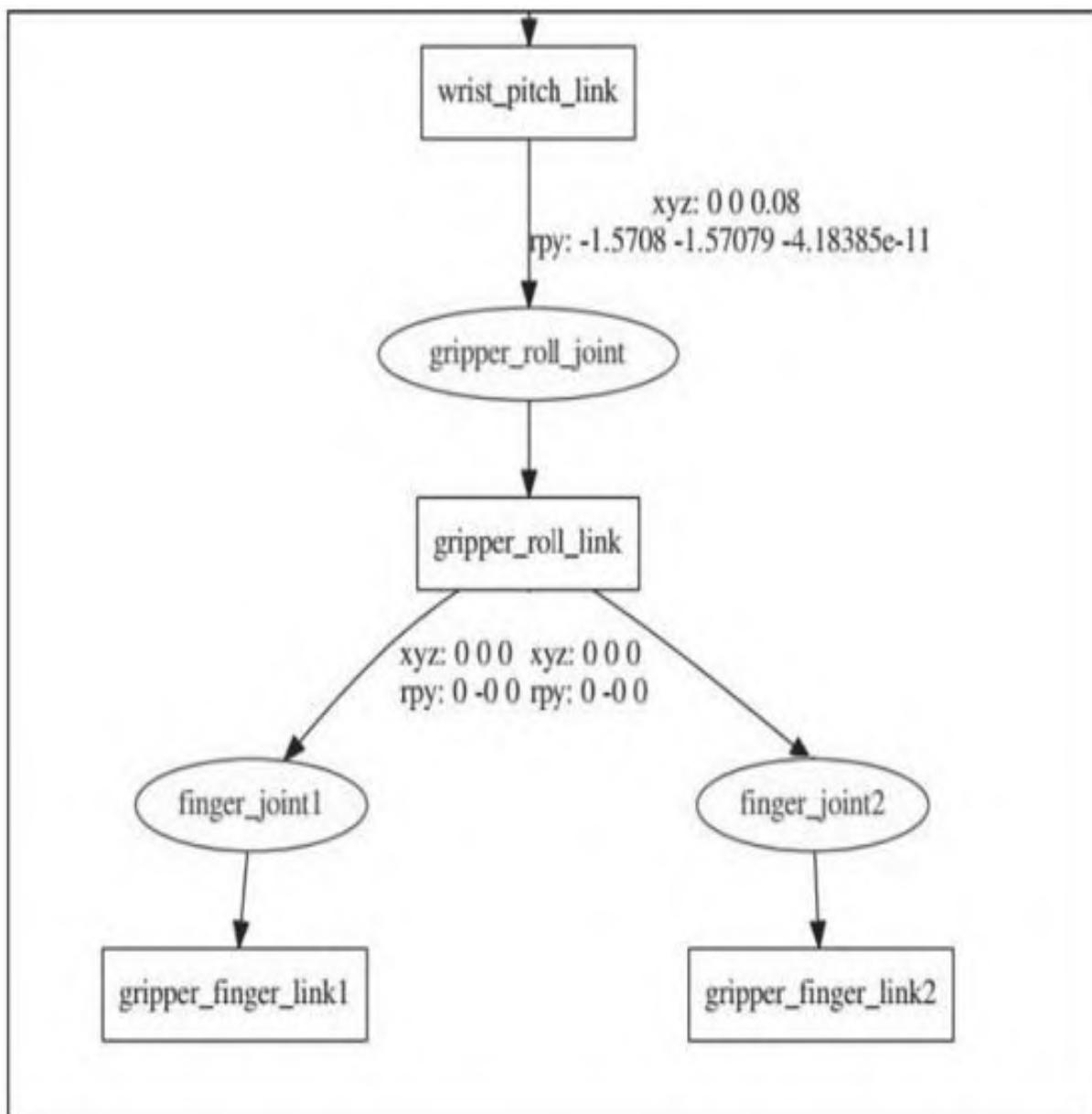


图3-9 7-DOF度机械臂的末端执行器连接图

```
<launch>
  <arg name="model" />

  <!-- Parsing xacro and loading robot_description parameter -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder
$(find mastering_ros_robot_description_pkg)/urdf/ seven_dof_arm.xacro" />

  <!-- Setting gui parameter to true for display joint slider, for getting
joint control -->
  <param name="use_gui" value="true"/>

  <!-- Starting Joint state publisher node which will publish the joint
values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />

  <!-- Starting robot state publish which will publish current robot joint
states using tf -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />

  <!-- Launch visualization in rviz -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mastering_ros_robot_description_pkg)/urdf.rviz" required="true" />
</launch>
```

在launch文件夹中创建以下启动文件，并使用catkin_make命令编译软件包。使用以下命令启动URDF：

```
$ rosrun mastering_ros_robot_description_pkg view_arm.launch
```

该机器人将在RViz上显示，且同时打开了关节状态发布者（joint state publisher）的GUI：

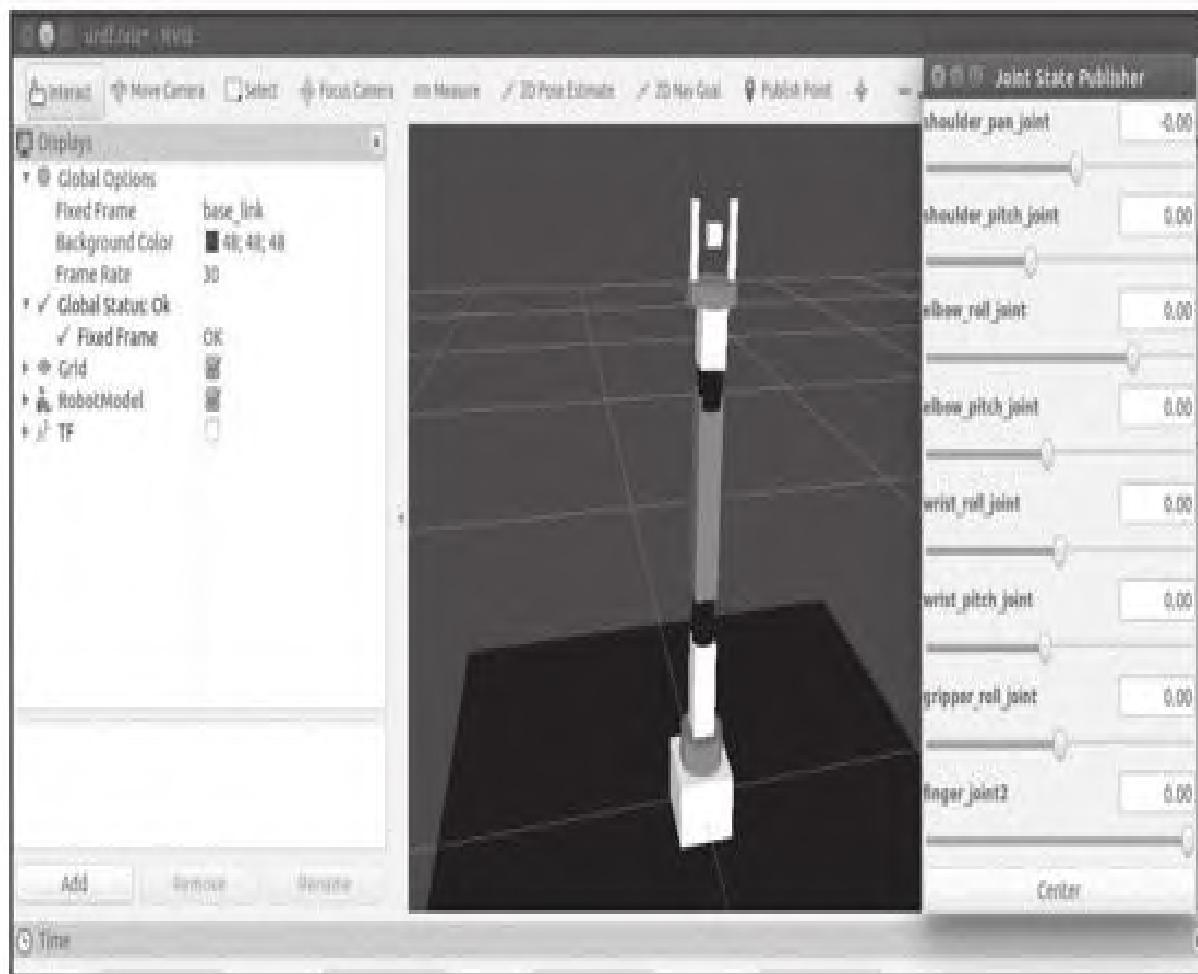


图3-10 在RViz中查看7-DOF机械臂和joint_state_publisher

我们可以与关节滑块进行交互并移动机器人的关节。接下来，我们将讨论关节状态发布者能做些什么。

理解关节状态发布者

关节状态发布者（joint state publisher）是一个ROS软件包，常用于与机器人的每个关节进行交互。该软件包包含joint_state_publisher节点，该节点将从URDF模型中找到非固定关节，并以sensor_msgs/JointState的消息格式发布每个关节的关节状态值。

在前面的启动文件view_arm.launch中，我们启动了joint_state_publisher节点并将一个名为use_gui的参数设置为true，如下所示：

```
<param name="use_gui" value="true"/>

<!-- Starting Joint state publisher node which will publish the joint
values -->
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
```

如果我们将use_gui设置为true，那么joint_state_publisher节点将显示一个基于滑动条控件的窗口来控制每个关节。关节的下限位值和上限位值将取自joint标签内的limit标签相关联的上下限位数据。前面的截图显示了RViz中的机器人模型，以及用于更改机器人关节位置的用户界面，其中use_gui参数的设置为true。

我们可以在[http://wiki.ros.org/joint state publisher](http://wiki.ros.org/joint%20state%20publisher)找到更多关于关节状态发布者的信息。

理解机器人状态发布者

机器人状态发布者（robot state publisher）软件包可以将机器人的状态发布到tf。此软件包订阅了机器人的关节状态，并使用URDF模型的运动表示来发布每个连杆的3D姿态。我们可以在启动文件中使用以下代码来实现机器人状态发布者节点：

```
<!-- Starting robot state publish which will publish tf -->
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
```

在前面的启动文件view_arm.launch中，我们启动了这个节点来发布机械臂的tf。我们可以通过点击RViz上的tf选项可视化机器人的变换，如图3-11所示。

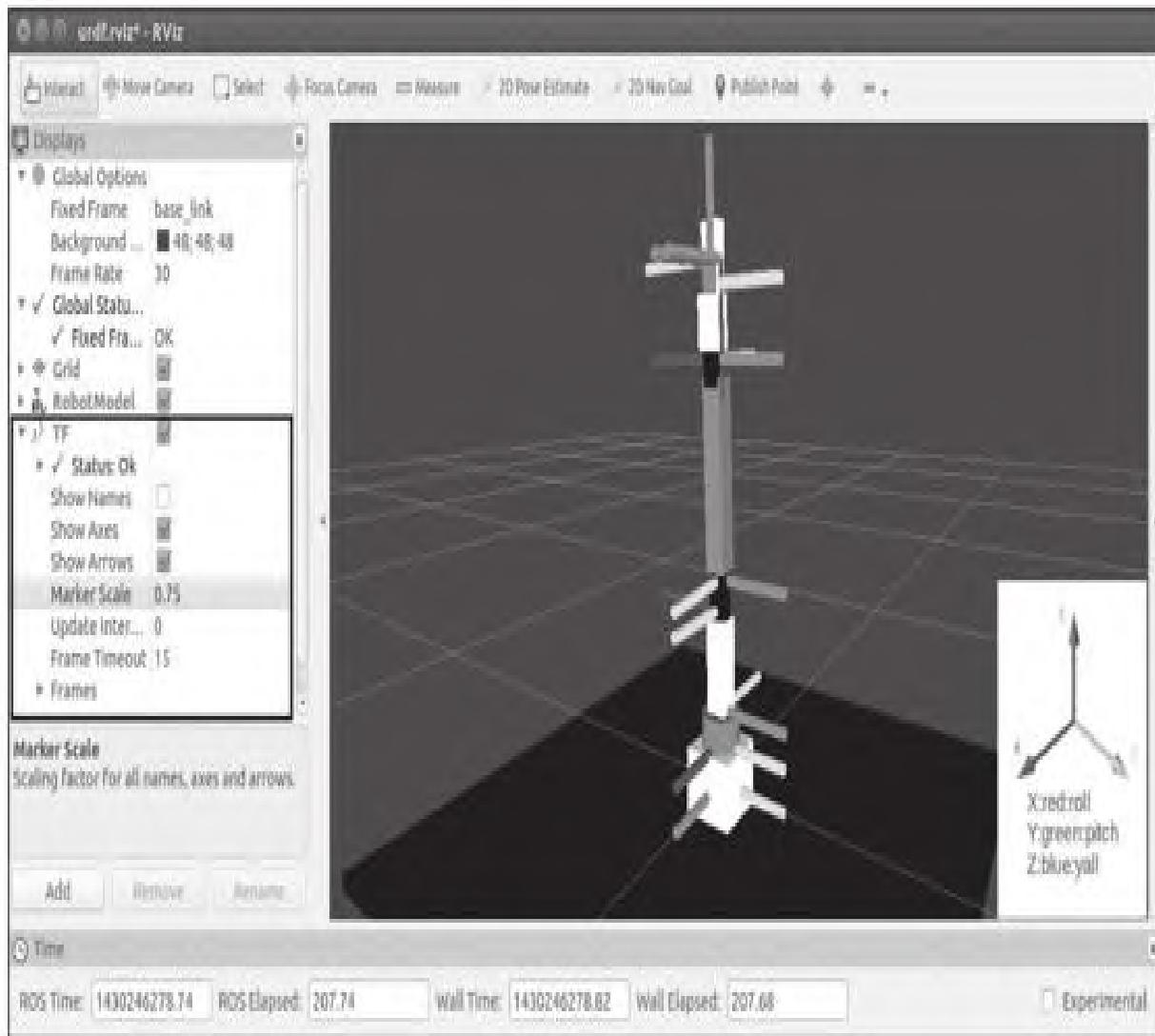


图 3-11 RViz 中 7-DOF 机械臂的 tf 视图

关节状态发布者软件包和机器人状态发布者软件包都是与ROS桌面安装程序一起安装的。

创建了7-DOF机械臂的机器人描述后，我们来讨论如何制作一个差速轮式移动机器人。

3.12 为差速驱动移动机器人创建机器人模型

差速轮式机器人在机器人底盘的两端安装两个轮子，整个底盘由一个或两个脚轮支撑。轮子将通过调节速度来控制机器人的移动速度。如果两个马达以相同的速度运行，轮子会向前或向后移动。如果一个轮子的速度比另一个慢，机器人就会转向低速的一边。如果我们想把机器人转动到左边，就减小左轮的速度，反之亦然。

底盘上有两个辅助轮，称为脚轮，它支撑着机器人并根据主轮的移动自由旋转。

这个机器人的URDF模型存放在下载的ROS软件包中。最终的机器人模型如图3-12所示：

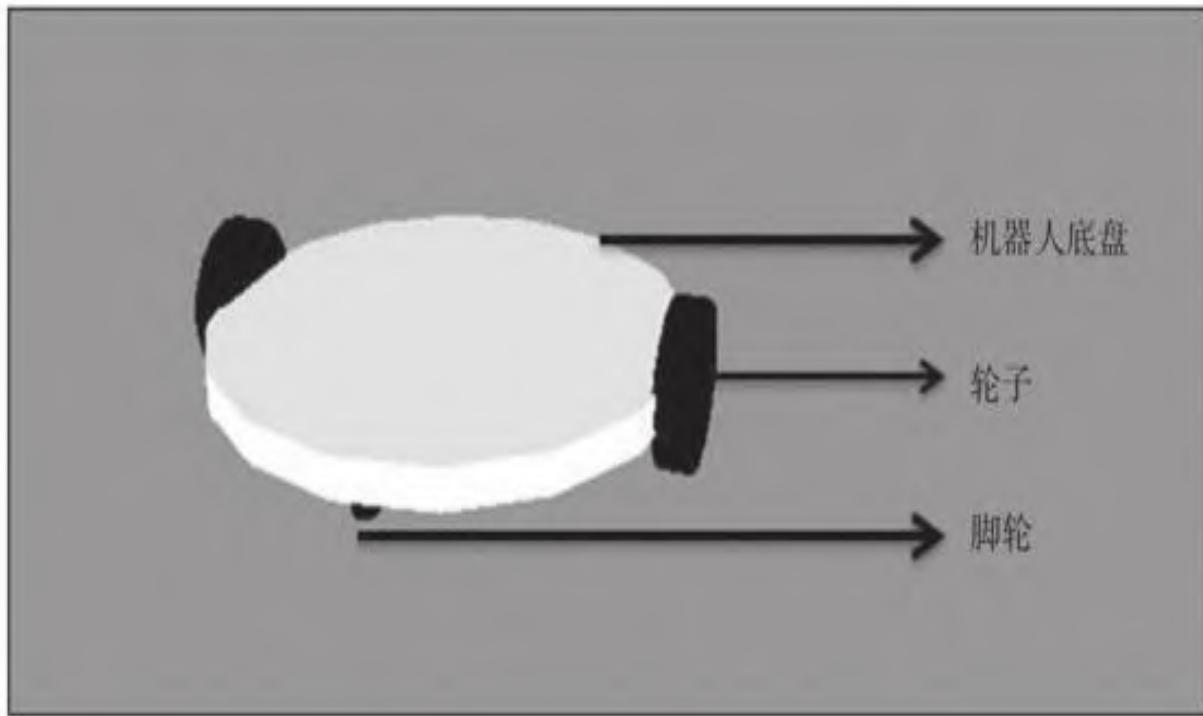


图3-12 差速驱动移动机器人

前面的机器人有5个关节和5个连杆。2个主要的关节将轮子连接到机器人上。其余3个是固定关节，2个用于将支撑脚轮连接到机器人主体上，1个用于将底盘连接到机器人主体上。图3-13是该机器人的连接图。

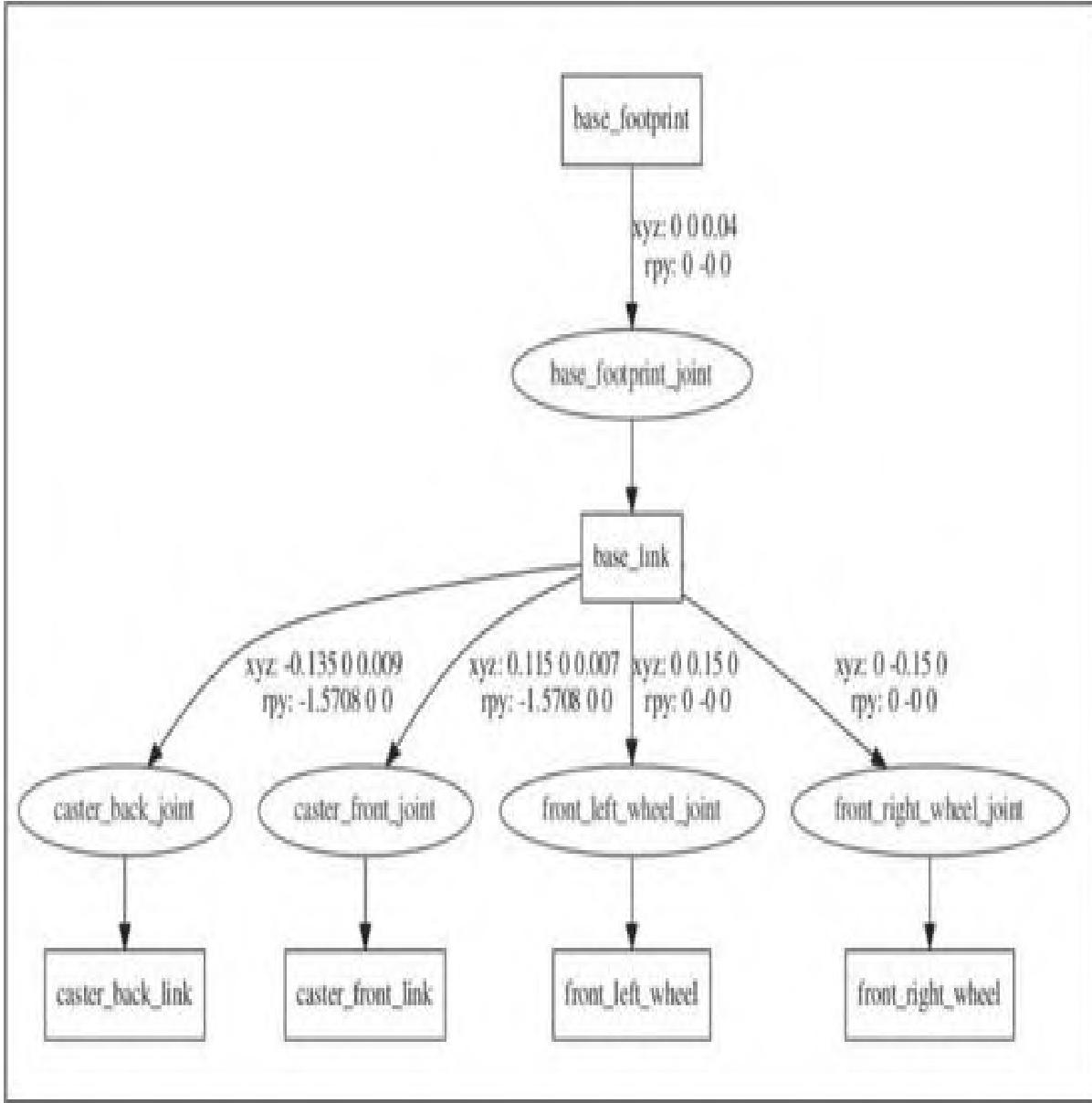


图3-13 差速驱动机器人的连杆和关节连接图

我们来看看URDF文件中的部分重要代码。这个URDF文件名为 `diff_wheeled_robot.xacro`，位于下载的ROS软件包内的 `urdf` 文件夹中。

这里给出了URDF文件的第一部分。机器人被命名为 `differential_wheeled_robot`，还包含一个名为 `wheel.urdf.xacro` 的URDF文件。`xacro`文件包含轮子的定义及其传动方式。如果我们使用该

xacro文件，就可以避免为两个轮子写两套定义。因为两个轮子在形状和大小上是相同的，所以我们采用xacro的定义：

```
<?xml version="1.0"?>
<robot name="differential_wheeled_robot"
xmlns:xacro="http://ros.org/wiki/xacro">

<xacro:include filename="$(find
mastering_ros_robot_description_pkg)/urdf/wheel.urdf.xacro">
```

轮子的定义在wheel.urdf.xacro中给出。我们可以指定轮子是否必须放在左边、右边、前面或后面。使用这个宏，我们最多可以创建4个轮子，但目前我们只需要2个：

```
<xacro:macro name="wheel" params="fb lr parent translateX translateY
flipY"> <!--fb : front, back ; lr: left, right -->
<link name="${fb}_${lr}_wheel">
```

我们还指定了仿真所需的Gazebo参数。这里提到的是与轮子相关的Gazebo参数。我们可以用gazebo:reference标签说明摩擦系数和刚度系数：

```
<gazebo reference="${fb}_${lr}_wheel">
  <mu1 value="1.0"/>
  <mu2 value="1.0"/>
  <kp value="10000000.0" />
  <kd value="1.0" />
  <fdir1 value="1 0 0"/>
  <material>Gazebo/Grey</material>
  <turnGravityOff>false</turnGravityOff>
</gazebo>
```

我们为轮子定义的关节是连续转动关节，因为在轮子关节中没有任何限值。这里的parent link是机器人底盘，child link是每个轮子：

```
<joint name="${fb}_${lr}_wheel_joint" type="continuous">
  <parent link="${parent}" />
  <child link="${fb}_${lr}_wheel" />
  <origin xyz="${translateX} *
```

我们还需要设定每个轮子的transmission标签。该轮子的宏定义如下：

```
<!-- Transmission is important to link the joints and the controller --
->
<transmission name="${fb}_${lr}_wheel_joint_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="${fb}_${lr}_wheel_joint" />
  <actuator name="${fb}_${lr}_wheel_joint_motor">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>

  </transmission>
</xacro:macro>
</robot>
```

在diff_wheeled_robot.xacro中，我们可以用以下命令来使用wheel.urdf.xacro中定义的宏：

```
<wheel fb="front" lr="right" parent="base_link" translateX="0"
translateY="-0.5" flipY="-1"/>
<wheel fb="front" lr="left" parent="base_link" translateX="0"
translateY="0.5" flipY="-1"/>
```

使用前面的几行代码，我们定义了机器人底盘的左侧和右侧的轮子。机器人底盘是圆柱形的，如上图所示。这里给出了惯性计算的宏定义。xacro代码段将使用圆柱的质量、半径和高度来计算惯性张量，代码如下：

```
<!-- Macro for calculating inertia of cylinder -->
<macro name="cylinder_inertia" params="m r h">
  <inertia ixx="${m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0"
            iyy="${m*(3*r*r+h*h)/12}" iyz = "0"
            izz="${m*r*r/2}" />
</macro>
```

这里给出了启动文件的定义，可用于在RViz中显示该机器人模型。启动文件命名为view_mobile_robot.launch：

```
<launch>
  <arg name="model" />
  <!-- Parsing xacro and setting robot_description parameter -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder
$(find mastering_ros_robot_description_pkg)/urdf/diff_wheeled_robot.xacro"
/>
  <!-- Setting gui parameter to true for display joint slider -->
  <param name="use_gui" value="true"/>
  <!-- Starting Joint state publisher node which will publish the joint
values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
  <!-- Starting robot state publish which will publish tf -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
  <!-- Launch visualization in rviz -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mastering_ros_robot_description_pkg)/urdf.rviz" required="true" />
</launch>
```

与机械臂URDF文件比较，唯一区别是名称不同，其他部分都是一样的。

我们可以使用下面的命令查看移动机器人：

```
$ roslaunch mastering_ros_robot_description_pkg view_mobile_robot.launch
```

在RViz中机器人的截图如图3-14所示。

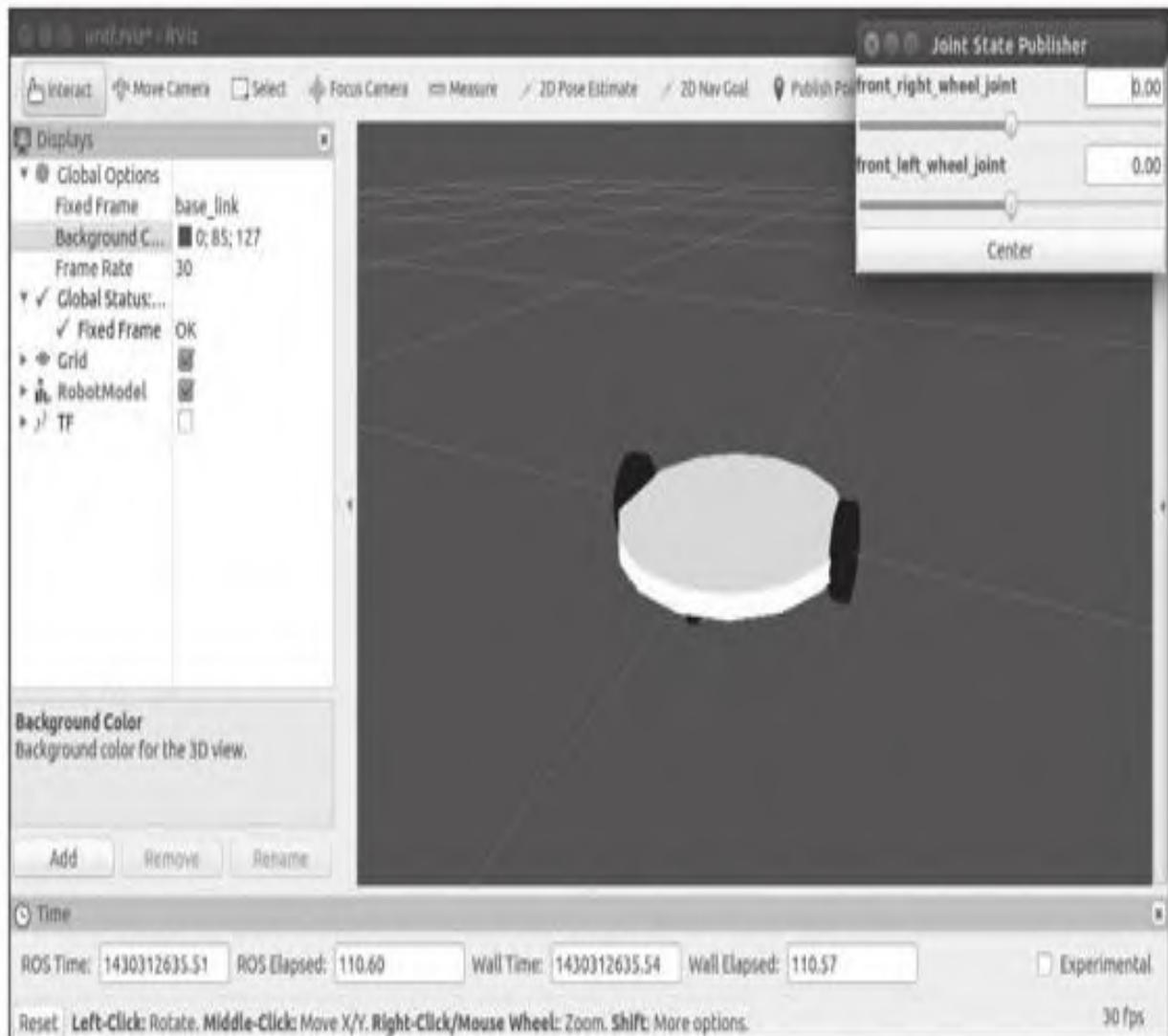


图3-14 在RViz中可视化移动机器人与关节状态发布者

3.13 习题

- 在ROS中用于机器人建模的软件包是什么？
- 用于机器人建模的重要的URDF标签是什么？
- 使用xacro而不是URDF的原因是什么？
- 关节状态发布者和机器人状态发布者软件包的功能各是什么？
- URDF中transmission标签的功能是什么？

3.14 本章小结

在本章中，我们主要讨论了机器人建模的重要性，以及如何在ROS中对机器人进行建模。然后我们更多地讨论了robot_model超软件包，以及包含在robot_model中的各种软件包，如urdf、xacro和joint_state_publisher。我们讨论了URDF、xacro和将要使用的主要URDF标签。我们还在URDF和xacro中创建了一个示例模型，并讨论了两者之间的差异。在此之后，我们又创建了一个复杂的7-DOF机械臂，并查看了关节状态发布者（joint state publisher）和机器人状态发布者（robot state publisher）两个软件包的使用方法。在本章的最后，我们回顾了使用xacro的差速驱动移动机器人的设计过程。在下一章中，我们将使用Gazebo来仿真这些机器人。

第4章

使用ROS和Gazebo进行机器人仿真

设计了机器人的3D模型之后，下一步就是仿真了。机器人仿真将让你了解机器人在虚拟环境中的工作情况。

我们将使用Gazebo (<http://www.gazebosim.org/>) 仿真器来仿真7-DOF机械臂和移动机器人。

Gazebo是一种多机器人仿真器，可用于室内外机器人仿真。我们可以仿真复杂的机器人、机器人传感器和各种3D物体。Gazebo已经在其模型仓库

(https://bitbucket.org/osrf/gazebo_models/) 中提供了各种流行的机器人、传感器和各种3D物体的仿真模型。我们可以直接使用这些模型，而无须创建新的模型。

Gazebo在ROS中有良好的接口，包含ROS中Gazebo的所有控制。我们可以在没有ROS的情况下安装Gazebo，若要实现ROS到Gazebo的通信，我们则必须安装ROS-Gazebo的接口。

在本章中，我们将讨论7-DOF机械臂和差速轮式机器人的仿真，还将讨论在Gazebo中帮助控制机器人关节的ROS控制器。

本章将介绍以下内容：

- 了解机器人仿真和Gazebo
- 在Gazebo中仿真机械臂模型
- 仿真带有rgb-d传感器的机械臂
- 在Gazebo中使用ROS控制器控制机器人关节
- 在Gazebo中仿真差速轮式机器人
- 在Gazebo中遥控移动机器人

4.1 使用Gazebo和ROS仿真机械臂

在前一章中，我们设计了一个7-DOF机械臂。在本节中，我们将在Gazebo中用ROS来仿真机器人。

在开始使用Gazebo和ROS之前，我们应该安装以下各软件包：

```
$ sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-msgs  
ros-kinetic-gazebo-plugins ros-kinetic-gazebo-ros-control
```

在ROS Kinetic版本中安装的默认版本是Gazebo 7.0。使用到的各软件包如下：

- gazebo_ros_pkgs：它包含用于将ROS与Gazebo连接的封装和工具
- gazebo-msgs：它包含ROS与Gazebo交互的消息和服务的数据结构
- gazebo-plugins：它包含用于传感器、执行结构的Gazebo插件
- gazebo-ros-control：它包含用于在ROS和Gazebo之间通信的标准控制器

安装后，请使用以下命令检查Gazebo是否安装正确：

```
$ roscore & rosrn gazebo_ros gazebo
```

这条命令将打开Gazebo的GUI。如果有了Gazebo仿真器，我们可以继续为Gazebo开发7-DOF机械臂的仿真模型。

4.2 为Gazebo创建机械臂仿真模型

我们可以通过添加仿真参数来更新现有的机器人描述，从而创建一个机械臂仿真模型。

我们可以使用以下命令来创建仿真机械臂所需的软件包：

```
$ catkin_create_pkg seven_dof_arm_gazebo gazebo_msgs gazebo_plugins  
gazebo_ros gazebo_ros_control mastering_ros_robot_description_pkg
```

也可以在以下Git库中获得完整的软件包。你可以从代码仓库下载，参考该软件包的实现，或者也可以从本书附带的源码中获得该软件包：

```
$ git clone https://github.com/jocacace/seven_dof_arm_gazebo.git
```

可以在seven_dof_arm.xacro文件中看到机器人的完整仿真模型，它放在mastering_ros_robot_description_pkg/urdf/文件夹中。

文件包含了URDF标签，这对于仿真是必要的。我们将定义碰撞、惯性、传动、关节、连杆，以及Gazebo。

我们可以使用seven_dof_arm_gazebo软件包来启动现有的仿真模型，它有一个名为seven_dof_arm_world.launch的启动文件。文件定义如下：

```
<launch>

    <!-- these are the arguments you can pass this launch file, for example
paused:=true -->
    <arg name="paused" default="false"/>
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="debug" default="false"/>

    <!-- We resume the logic in empty_world.launch -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="debug" value="$(arg debug)" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="paused" value="$(arg paused)" />
        <arg name="use_sim_time" value="$(arg use_sim_time)" />
        <arg name="headless" value="$(arg headless)" />
    </include>

    <!-- Load the URDF into the ROS Parameter Server -->
    <param name="robot_description" command="$(find xacro)/xacro --inorder
'$(find mastering_ros_robot_description_pkg)/urdf/seven_dof_arm.xacro'" />

    <!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
    <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen"
    args="-urdf -model seven_dof_arm -param robot_description"/>
</launch>
```

启动下面的命令并检查你获得的内容：

```
$ rosrun seven_dof_arm_gazebo seven_dof_arm_world.launch
```

可以在Gazebo中看到机械臂，如图4-1所示。如果得到这个输出，没有任何错误，说明成功了。

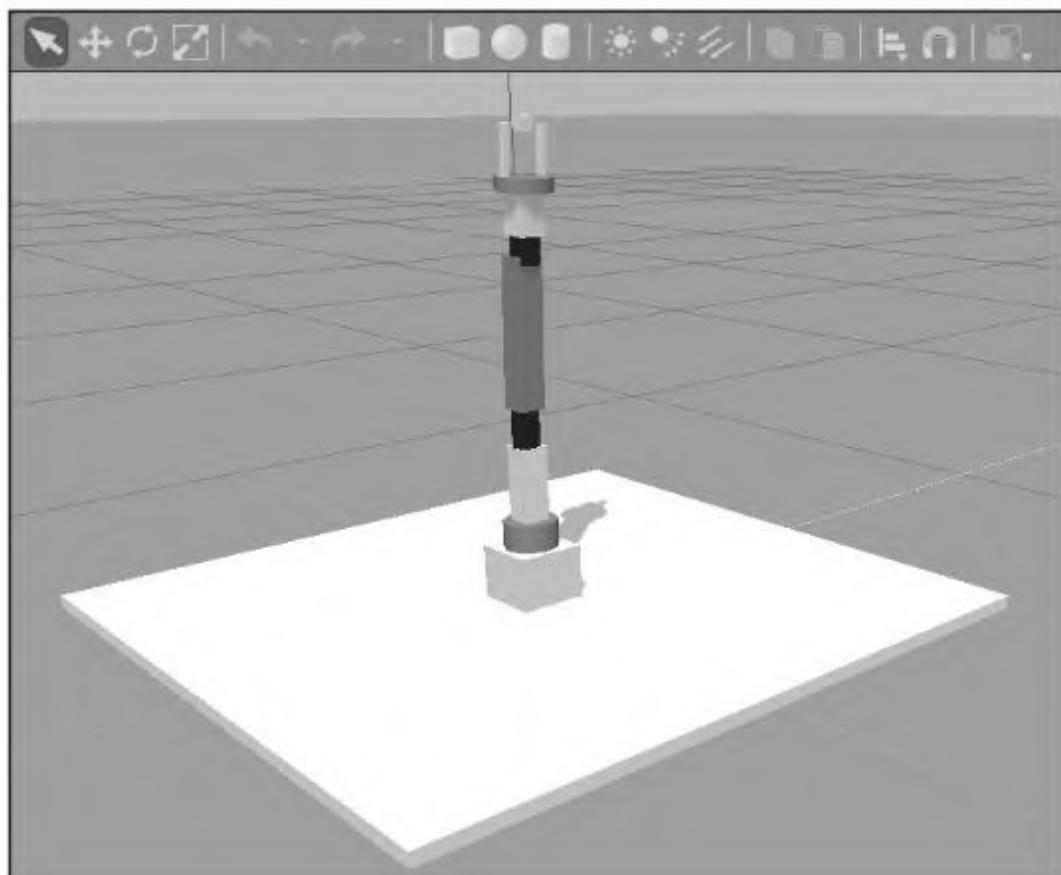


图4-1 在Gazebo中仿真7-DOF机械臂

让我们来详细地讨论一下机器人的仿真模型文件。

4.2.1 为Gazebo机器人模型添加颜色和纹理

在机器人仿真中我们可以看到每个连杆都有不同的颜色和纹理。在xacro文件中，下面的标签可以为机器人的连杆提供纹理和颜色：

```
<gazebo reference="bottom_link">
  <material>Gazebo/White</material>
</gazebo>
<gazebo reference="base_link">
  <material>Gazebo/White</material>
</gazebo>
<gazebo reference="shoulder_pan_link">
  <material>Gazebo/Red</material>
</gazebo>
```

4.2.2 添加transmission标签来驱动模型

为了使用ROS控制器来驱动机器人，我们需要定义<transmission>（传动）标签来连接执行机构和关节。以下是为传动定义的宏：

```
<xacro:macro name="transmission_block" params="joint_name">
  <transmission name="trans1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="${joint_name}">
      <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
</xacro:macro>
```

在这里<joint name="">是连接驱动器的关节。<type>标签是传动类型。目前，仅支持简单的传动transmission_interface/SimpleTransmission。<hardwareInterface>标签是要加载的硬件接口类型（位置、速度或力度）。在该示例中，使用了位置控制硬件接口。这个硬件接口由gazebo_ros_control插件加载，在下一节中我们将看到这个插件。

4.2.3 添加gazebo_ros_control插件

在添加传动标签之后，我们应该在仿真模型中添加gazebo_ros_control插件来解析传动标签并分配适当的硬件接口和控制管理器。以下代码将gazebo_ros_control插件添加到xacro文件：

```
<!-- ros_control plugin -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/seven_dof_arm</robotNamespace>
  </plugin>
</gazebo>
```

在这里，`<plugin>`标签指定了要加载的插件名是`libgazebo_ros_control.so`。可以将`<robotNamespace>`标签作为机器人的名称，如果我们没有指定名称，它将从URDF自动加载机器人的名称。我们还可以在参数服务器(`<robotParam>`)上指定控制器刷新速率(`<control-Period>`)，`robot_description`(URDF)的位置以及机器人硬件接口的类型(`<robotSimType>`)。默认的硬件接口可以是`JointStateInterface`、`EffortJointInterface`或`VelocityJointInterface`。

4.2.4 在Gazebo中添加3D视觉传感器

在Gazebo中，我们可以仿真机器人的运动和物理特征，也可以对传感器进行仿真。

要在Gazebo中创建一个传感器，我们就必须对Gazebo中传感器的行为进行建模。Gazebo中有一些预先创建的传感器模型，可以直接在代码中使用而无须编写新模型。

这里，我们在Gazebo中添加了一个名为Asus Xtion Pro模型的3D视觉传感器（通常称为RGB-D传感器）。传感器模型已经在gazebo_ros_pkgs/gazebo_plugins的ROS软件包中实现，在我们的ROS系统中应该已经安装了它。

Gazebo中的每个模型都可以作为Gazebo-ROS插件实现，可以将其插入URDF文件来加载。

以下是我们如何在seven_dof_arm_with_rgbd.xacro机器人的xacro文件中把Gazebo定义和Xtion Pro的物理机器人模型包含进来的：

```
<xacro:include filename="$(find  
mastering_ros_robot_description_pkg)/urdf/sensors/xtion_pro_live.urdf.xacro  
"/>
```

在xtion_pro_live.urdf.xacro文件内部，我们可以看到以下几行：

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="$(find
mastering_ros_robot_description_pkg)/urdf/sensors/xtion_pro_live.gazebo.xac
ro"/>
  .....
  <xacro:macro name="xtion_pro_live" params="name parent *origin
*optical_origin">
  .....
  <link name="${name}_link">
    .....
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <mesh
filename="package://mastering_ros_robot_description_pkg/meshes/sensors/xtio
n_pro_live/xtion_pro_live.dae"/>
      </geometry>
      <material name="DarkGrey"/>
    </visual>
  </link>

</robot>
```

在这里，我们可以看到它包含另一个名为
xtion_pro_live.gazebo.xacro的文件，该文件包含Xtion Pro在
Gazebo中的完整定义。

我们还可以看到一个名为xtion_pro_live的宏定义，其中包含
Xtion Pro的完整模型定义，包括连杆和关节：

```
<mesh  
filename="package://mastering_ros_robot_description_pkg/meshes/sensors/xtio  
n_pro_live/xtion_pro_live.dae"/>
```

在宏定义中，我们将导入一个Asus Xtion Pro的网格文件，该文件将在Gazebo中显示为相机连杆。

在 mastering_ros_robot_description_pkg/urdf/sensors/xtion_pro_live.gazwbo.xacro 文件中，我们可以设置Xtion Pro的Gazebo-ROS插件。在这里，我们将插件定义为宏，并支持RGB和深度相机。以下是插件的定义：

```
<plugin name="${name}_frame_controller"  
filename="libgazebo_ros_openni_kinect.so">  
    <alwaysOn>true</alwaysOn>  
    <updateRate>6.0</updateRate>  
    <cameraName>${name}</cameraName>  
    <imageTopicName>rgb/image_raw</imageTopicName>  
  
</plugin>
```

Xtion Pro的插件文件名是 libgazebo_ros_openni_kinect.so，我们可以定义插件参数，例如相机名称、图像话题等。

4.3 仿真装有Xtion Pro的机械臂

现在我们已经了解了Gazebo中相机插件的定义，我们可以使用以下命令启动完整的仿真：

```
$ roslaunch seven_dof_arm_gazebo seven_dof_arm_with_rgbd_world.launch
```

我们可以看到一个机器人模型，在机械臂顶部装有一个传感器，如图4-2所示。

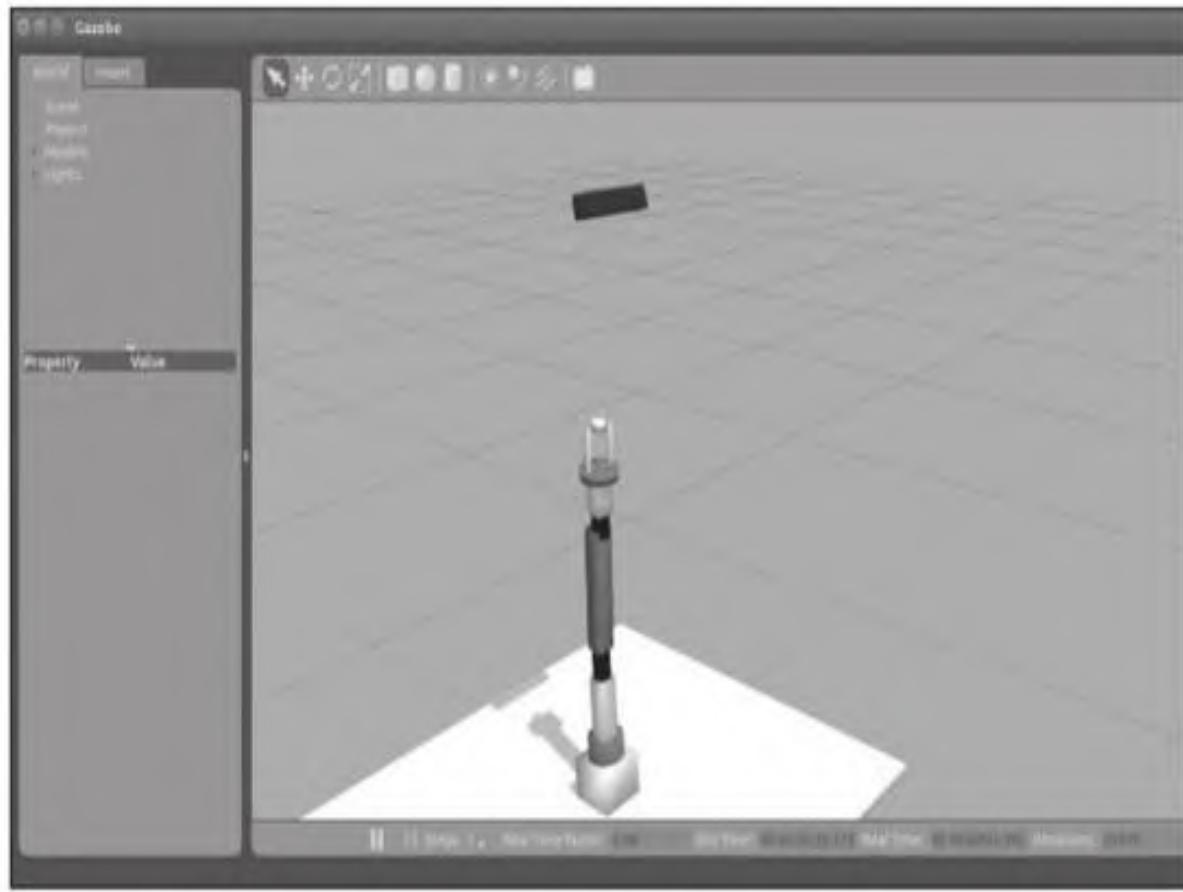


图4-2 在Gazebo中仿真装有Asus Xtion Pro的7-DOF机械臂

现在我们可以使用仿真的RGB-D传感器，就像它直接连在我们的计算机上一样。因此我们可以检查它是否提供了正确的图像输出。

可视化三维传感器数据

使用上述命令启动仿真后，我们可以检查由传感器插件生成的话题，如图4-3所示。

```
jcacace@robot:~$ rostopic list
/rgbd_camera/depth/image_raw
/rgbd_camera/ir/image_raw
/rgbd_camera/rgb/image_raw
```

图4-3 由Gazebo生成的RGB-D图像话题

让我们使用名为image_view的工具来查看3D视觉传感器的图像数据：

- 查看RGB原始图像：

```
$ rosrun image_view image_view image:=rgbd_camera/rgb/image_raw
```

- 查看IR原始图像：

```
$ rosrun image_view image_view image:=rgbd_camera/ir/image_raw
```

- 查看深度图像

```
$ rosrun image_view image_view image:=rgbd_camera/depth/image_raw
```

图4-4是包含所有这些图像的截图。

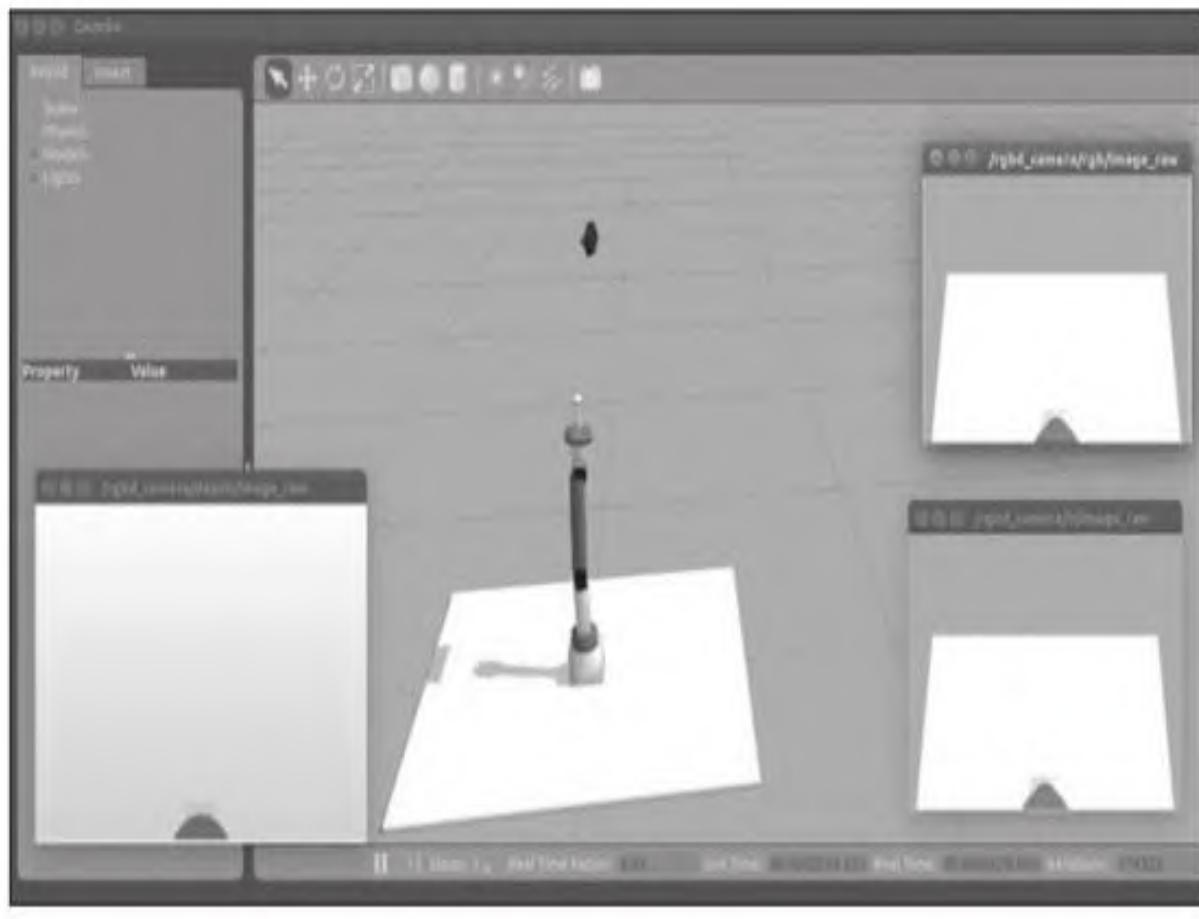


图4-4 在Gazebo中查看RGB-D传感器的图像

我们还可以在RViz中查看此传感器的点云数据。

使用以下命令启动RViz：

```
$ rosrun rviz rviz -f /rgbd_camera_optical_frame
```

添加一个PointCloud2显示类型，并将Topic设置为/rgbd_camera/depth/points。我们将得到如图4-5所示的点云视图。

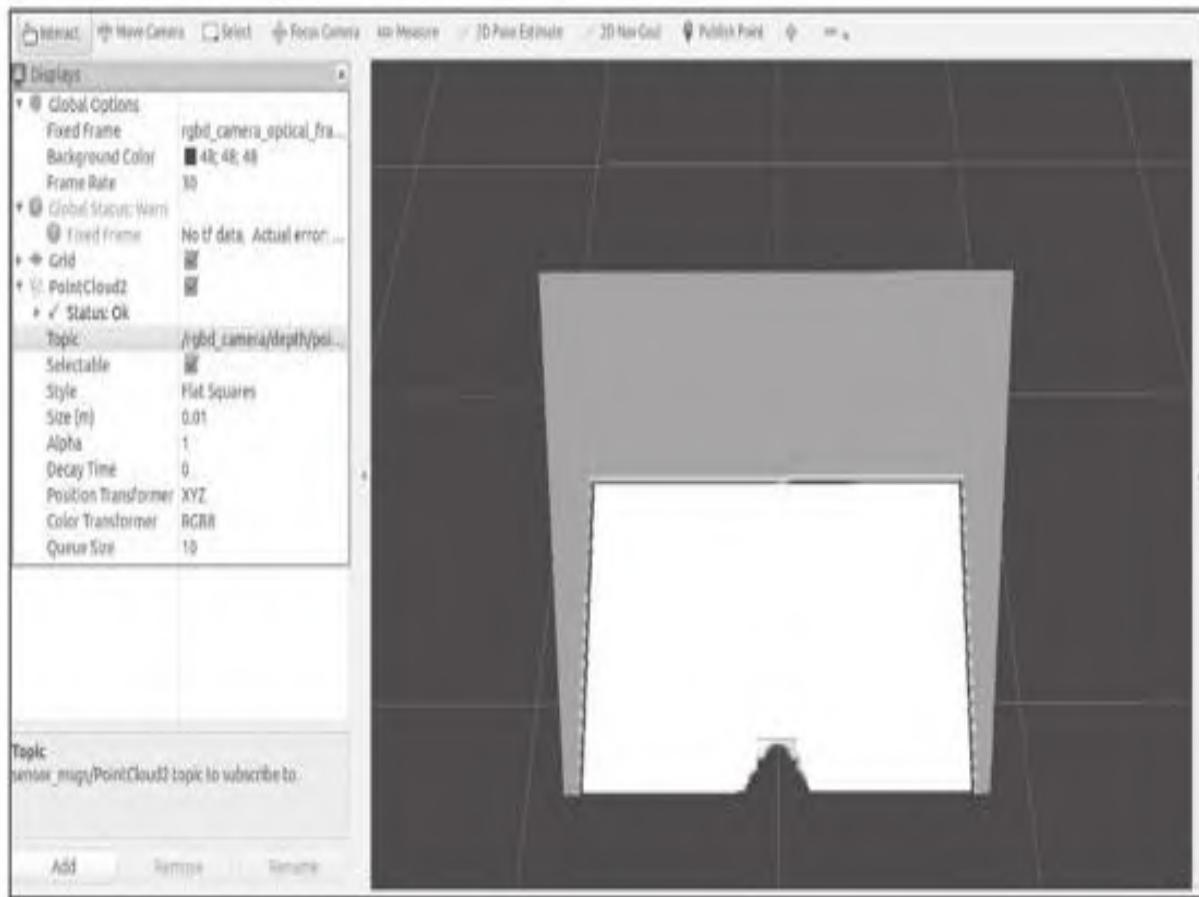


图4-5 在RViz中查看RGB-D传感器点云数据

4.4 在Gazebo中使用ROS控制器

在本节中，我们将讨论如何在Gazebo中让机器人的每个关节运动。

为了让关节动起来，我们需要分配一个ROS控制器。尤其是，我们需要为每个关节连上一个与transmission标签内指定的硬件接口兼容的控制器。

ROS控制器主要由一套反馈机构组成，可以接收某一设定点，并用执行机构的反馈控制输出。

ROS控制器使用硬件接口与硬件交互。硬件接口的主要功能是充当ROS控制器与真实或仿真硬件之间的中介，根据ROS控制器生成的数据来分配资源控制它。

在本机器人中，我们定义了位置控制器、速度控制器、力控制器等。这些ROS控制器是由名为ros_control的一组软件包提供的。

为了正确理解如何为机械臂配置ROS控制器，我们需要理解它的概念。我们将进一步讨论ros_control软件包、不同类型的ROS控制器以及ROS控制器如何与Gazebo仿真交互。

4.4.1 认识ros_control软件包

ros_control软件包实现了机器人控制器、控制管理器、硬件接口、不同的传输接口和控制工具箱。ros_controls软件包由以下各独立的软件包组成：

- control_toolbox：这个软件包包含了通用模块(PID和Sine)，可供所有控制器使用；
- controller_interface：这个软件包包含了控制器的接口（interface）基类；
- controller_manager：这个软件包提供了加载（load）、卸载（unload）、启动（start）和停止（stop）等控制器的基础架构；
- controller_manager_msgs：这个软件包提供了控制管理器的消息和服务定义；
- hardware_interface：这个软件包包含了硬件接口的基类；
- transmission_interface：这个软件包包含了传动（transmission）接口的接口类（差速、四杆联动、关节状态、位置和速度）。

4.4.2 不同类型的ROS控制器和硬件接口

让我们看看包含标准ROS控制器的ROS软件包列表：

- joint_position_controller：这是关节位置控制器的简单实现。
- joint_state_controller：这是一个发布关节状态的控制器。
- joint_effort_controller：这是关节力（强度）控制器的实现。

以下是ROS中常用的一些硬件接口：

- Joint Command Interface：将命令发送到硬件。
- Effort Joint Interface：发送effort命令。
- Velocity Joint Interface：发送velocity命令。
- Position Joint Interface：发送position命令。
- Joint State Interface：从执行器编码器检索关节状态。

4.4.3 ROS控制器如何与Gazebo交互

让我们看看ROS控制器是如何与Gazebo进行交互的。图4-6显示了ROS控制器、机器人硬件接口、仿真器/真实硬件的互连：

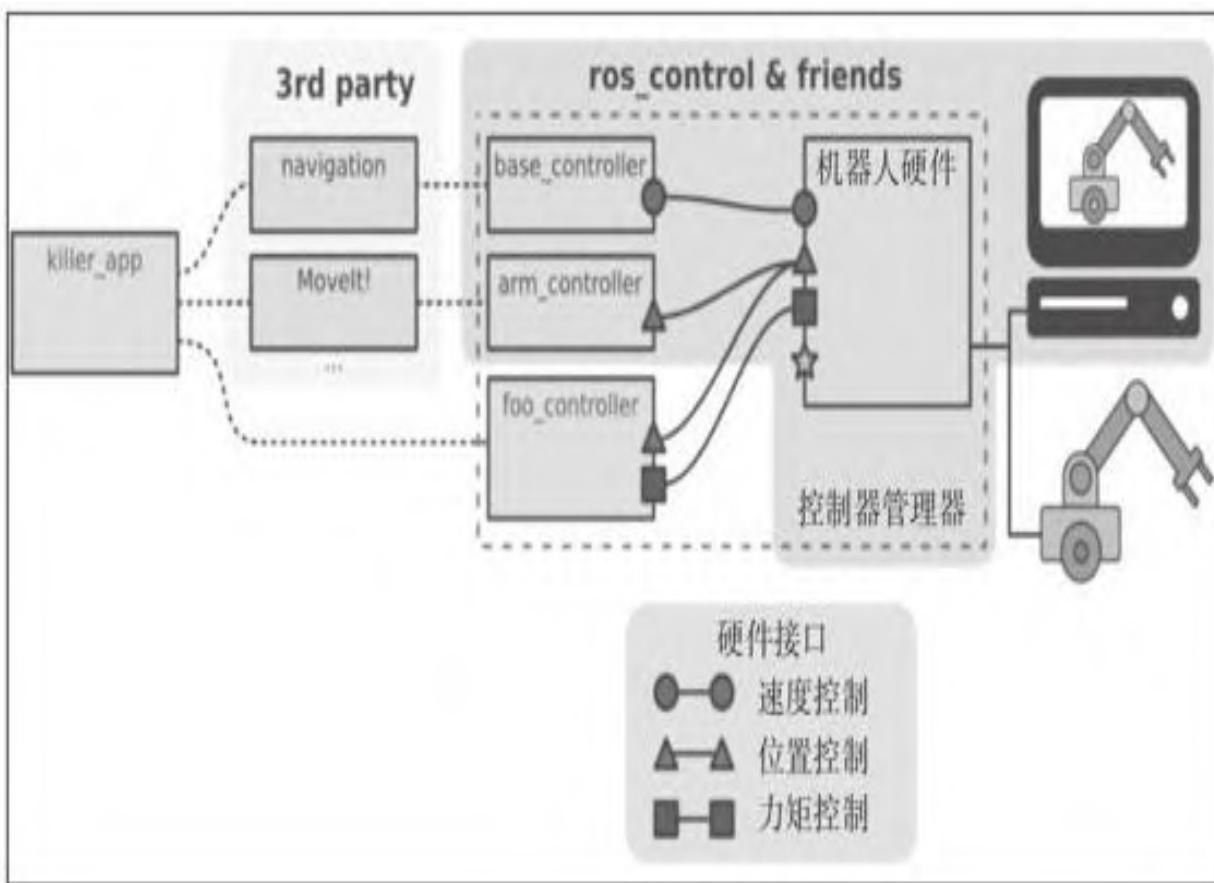


图4-6 ROS控制器与Gazebo的接口

我们可以看到第三方工具navigation和MoveIt! 软件包。这些软件包可以为移动机器人控制器和机械臂控制器提供目标位置（即设定点）。这些控制器可以将位置、速度或驱动力发送到机器人的硬件接口上。

硬件接口将每个资源分配给控制器，并将值发送给每个资源。机器人控制器与机器人硬件接口之间的通信如图4-7所示。

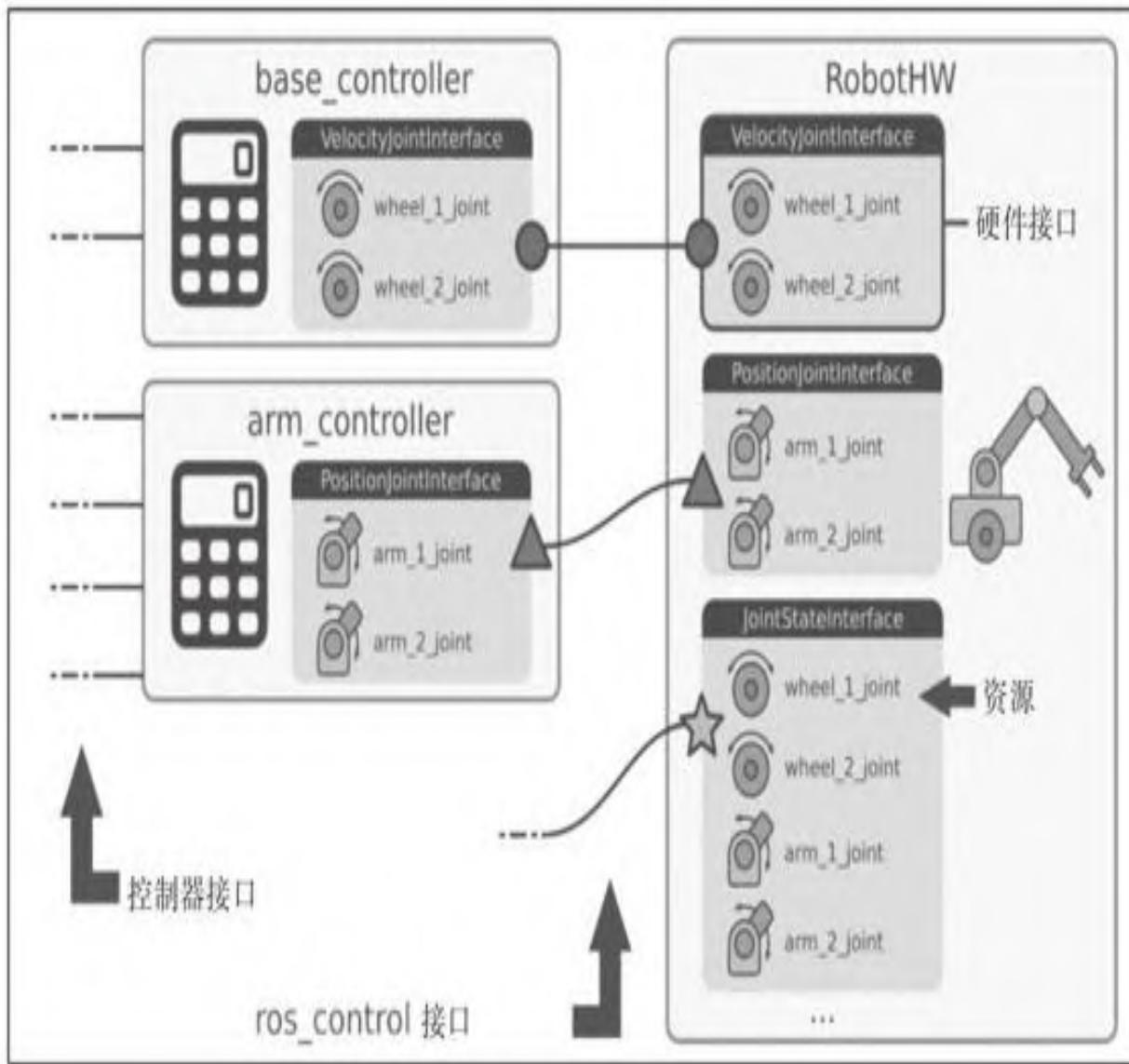


图4-7 ROS控制器和硬件接口的示意图

硬件接口与实际硬件和仿真分离。来自硬件接口的值可以馈送到 Gazebo 进行仿真或馈送到实际硬件本身。

硬件接口是机器人及其抽象硬件的软件表示。硬件接口的资源是执行机构、关节和传感器。有些资源是只读的，比如关节状态、IMU、力-扭矩传感器等，有些资源是可读可写的，比如位置、速度和关节驱动力。

4.4.4 将关节状态控制器和关节位置控制器连接到手臂

将机器人控制器连接到每个关节是一项简单的任务。第一项任务就是为两个控制器编写配置文件。

关节状态控制器将发布手臂的关节状态，而且关节位置控制器可以接收每个关节的目标位置并可以让每个关节运动。

我们将在seven_dof_arm_gazebo/config文件夹下找到控制器的配置文件seven_dof_arm_gazebo_control.yaml。

以下是配置文件的定义：

```
seven_dof_arm:
  # Publish all joint states -----
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50
  # Position Controllers -----
  joint1_position_controller:
    type: position_controllers/JointPositionController
    joint: shoulder_pan_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint2_position_controller:
    type: position_controllers/JointPositionController
    joint: shoulder_pitch_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint3_position_controller:
    type: position_controllers/JointPositionController
    joint: elbow_roll_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint4_position_controller:
    type: position_controllers/JointPositionController
    joint: elbow_pitch_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint5_position_controller:
    type: position_controllers/JointPositionController
    joint: wrist_roll_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint6_position_controller:
    type: position_controllers/JointPositionController
    joint: wrist_pitch_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint7_position_controller:
    type: position_controllers/JointPositionController
    joint: gripper_roll_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
```

我们可以看到所有的控制器都位于命名空间seven_dof_arm中，第一行代表关节状态控制器，它将以50Hz的频率发布机器人的关节状态。

其余的控制器是关节位置控制器，它被分配给前7个关节，而且还定义了PID增益。

4.4.5 在Gazebo中启动ROS控制器

如果控制器配置准备就绪，我们就可以创建一个启动文件，该文件将启动所有控制器并进行Gazebo仿真。

进入seven_dof_arm_gazebo/launch文件夹并打开seven_dof_arm_gazebo_control.launch文件：

```
<launch>
  <!-- Launch Gazebo -->
  <include file="$(find
seven_dof_arm_gazebo)/launch/seven_dof_arm_world.launch" />
```

```
<!-- Load joint controller configurations from YAML file to parameter
server -->
<rosparam file="$(find
seven_dof_arm_gazebo)/config/seven_dof_arm_gazebo_control.yaml"
command="load"/>

<!-- load the controllers -->
<node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false"
output="screen" ns="/seven_dof_arm" args="joint_state_controller
joint1_position_controller
joint2_position_controller
joint3_position_controller
joint4_position_controller
joint5_position_controller
joint6_position_controller
joint7_position_controller"/>

<!-- convert joint states to TF transforms for rviz, etc -->
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher"
respawn="false" output="screen">
  <remap from="/joint_states" to="/seven_dof_arm/joint_states" />
</node>

</launch>
```

该启动文件可以启动手臂的Gazebo仿真，加载控制器配置信息、关节状态控制器和关节位置控制器，最后运行机器人状态发布者（负责发布关节状态和tf）。

检查运行此启动文件后生成的控制器话题：

```
$ rosrun seven_dof_arm_gazebo seven_dof_arm_gazebo.launch
```

如果命令执行成功，我们可以在终端看到如图4-8的消息。

```
[ INFO] [1503389354.607765795, 0.155000000]: Loaded gazebo_ros_control.  
[INFO] [1503389354.726844, 0.274000]: Controller Spawner: Waiting for service controller_manager/switch_controller  
[INFO] [1503389354.728599, 0.276000]: Controller Spawner: Waiting for service controller_manager/unload_controller  
[INFO] [1503389354.730271, 0.277000]: Loading controller: joint_state_controller  
[INFO] [1503389354.812192, 0.355000]: Loading controller: joint1_position_controller  
[INFO] [1503389354.896451, 0.433000]: Loading controller: joint2_position_controller  
[INFO] [1503389354.905462, 0.442000]: Loading controller: joint3_position_controller  
[INFO] [1503389354.914256, 0.451000]: Loading controller: joint4_position_controller  
[INFO] [1503389354.921849, 0.458000]: Loading controller: joint5_position_controller  
[INFO] [1503389354.928891, 0.466000]: Loading controller: joint6_position_controller  
[INFO] [1503389354.935862, 0.473000]: Loading controller: joint7_position_controller  
[INFO] [1503389354.944609, 0.482000]: Controller Spawner: Loaded controllers: joint_state_controller, joint1_position_controller, joint2_position_controller, joint3_position_controller, joint4_position_controller, joint5_position_controller, joint6_position_controller, joint7_position_controller  
[INFO] [1503389354.947569, 0.485000]: Started controllers: joint_state_controller, joint1_position_controller, joint2_position_controller, joint3_position_controller, joint4_position_controller, joint5_position_controller, joint6_position_controller, joint7_position_controller
```

图4-8 启动7-DOF机械臂的ROS控制器时的终端消息

图4-9是我们运行该启动文件时从控制器中生成的话题：

```
/seven_dof_arm/joint1_position_controller/command  
/seven_dof_arm/joint2_position_controller/command  
/seven_dof_arm/joint3_position_controller/command  
/seven_dof_arm/joint4_position_controller/command  
/seven_dof_arm/joint5_position_controller/command  
/seven_dof_arm/joint6_position_controller/command  
/seven_dof_arm/joint7_position_controller/command
```

图4-9 ROS控制器生成的位置控制器命令话题

4.4.6 控制机器人的关节运动

完成以上步骤后，我们就可以开始对每个关节进行控制了。

要在Gazebo中控制机器人关节运动，我们需要使用
std_msgs/Float64类型的消息将所需的关节值发布到关节位置控制器
命令话题上。

图4-10展示了控制第4个关节运动到1.0度的位置：

```
$ rostopic pub /seven_dof_arm/joint4_position_controller/command  
std_msgs/Float64 1.0
```

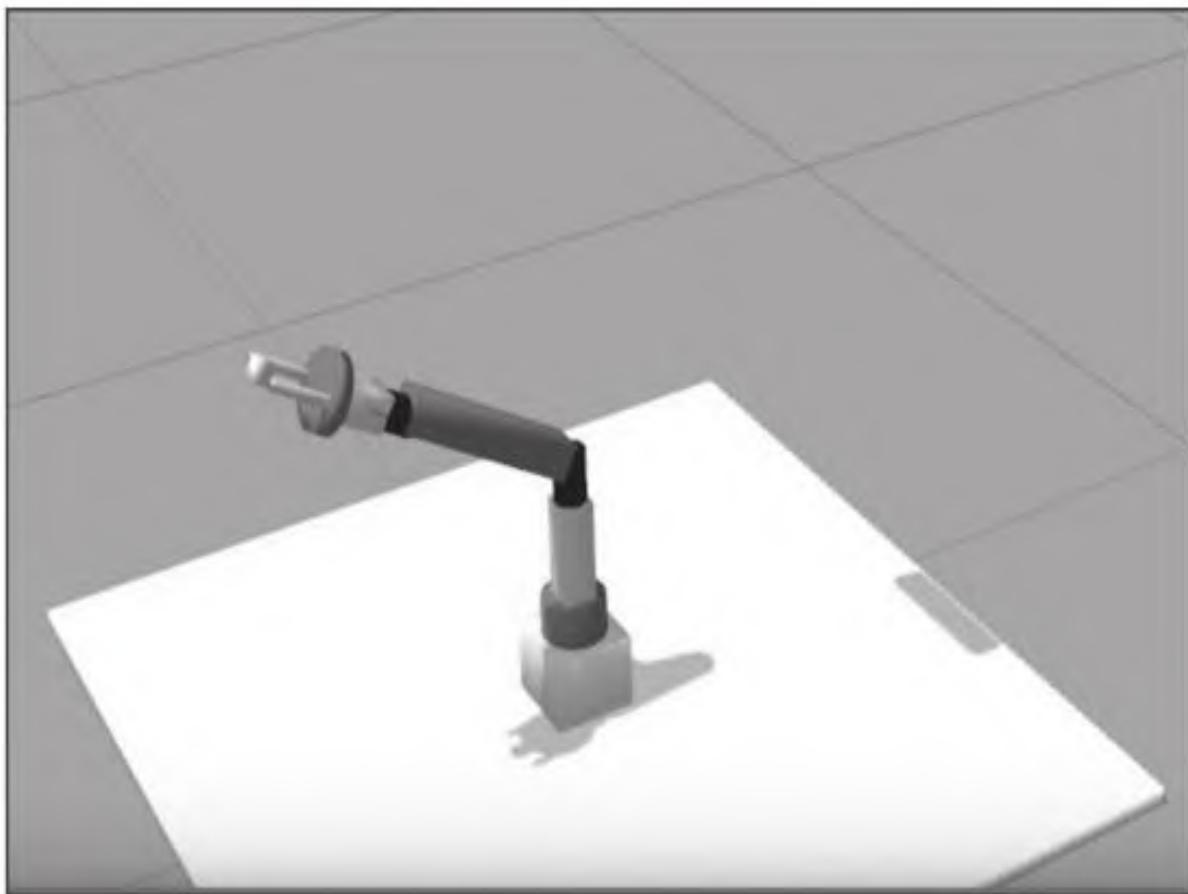


图4-10 在Gazebo中控制手臂关节运动

我们还可以用以下命令查看机器人的关节状态：

```
$ rostopic echo /seven_dof_arm/joint_states
```

4.5 在Gazebo中仿真差速轮式机器人

我们已经学习了机械臂的仿真。在本节，我们会对前一章设计的差速轮式机器人进行仿真。

你可以在mastering_ros_robot_description_pkg/urdf文件夹中获取diff_wheeled_robot.xacro移动机器人的描述文件。

让我们创建一个启动文件，在Gazebo中生成仿真模型。就像我们对机械臂所做的那样，我们可以创建一个ROS软件包，用seven_dof_arm_gazebo软件包的相同依赖项启动Gazebo仿真，从下面的Git库中下载整个软件包，或者从本书附带的源代码中获取软件包：

```
$ git clone https://github.com/jocacace/diff_wheeled_robot_gazebo.git
```

进入diff_wheeled_robot_gazebo/launch文件夹，并提取diff_wheeled_gazebo.launch文件。以下是启动文件的定义：

```
<launch>

    <!-- these are the arguments you can pass this launch file, for example
paused:=true -->
    <arg name="paused" default="false"/>
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="debug" default="false"/>

    <!-- We resume the logic in empty_world.launch -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="debug" value="$(arg debug)" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="paused" value="$(arg paused)" />
        <arg name="use_sim_time" value="$(arg use_sim_time)" />
        <arg name="headless" value="$(arg headless)" />
    </include>

    <!-- urdf xml robot description loaded on the Parameter Server-->
    <param name="robot_description" command="$(find xacro)/xacro --inorder
'$(find
mastering_ros_robot_description_pkg)/urdf/diff_wheeled_robot.xacro'" />

    <!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
    <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen"
args="-urdf -model diff_wheeled_robot -param robot_description"/>

</launch>
```

我们可以使用以下命令来启动此文件：

```
$ roslaunch diff_wheeled_robot_gazebo diff_wheeled_gazebo.launch
```

你将在Gazebo中看到如图4-11所示的机器人模型。如果你看到了这个模型，说明你已经成功地完成了第一阶段的仿真：

仿真成功后，接下来我们将激光雷达添加到机器人中。

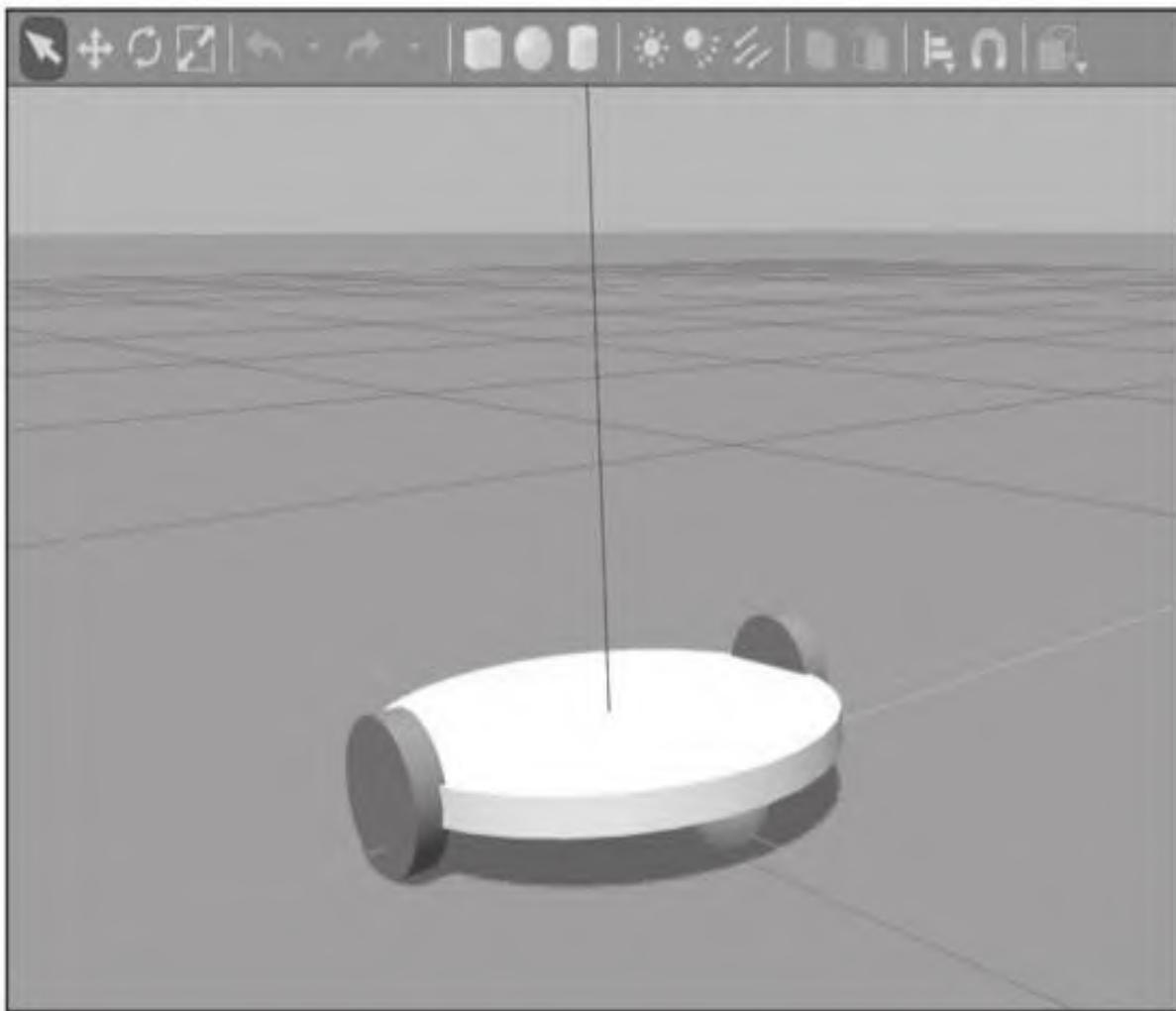


图4-11 在Gazebo中的差速轮式机器人

4.5.1 将激光雷达添加到机器人中

我们在机器人顶部添加了激光雷达来执行高级操作，比如用该机器人进行自主导航或地图构建。在这里，为了将激光雷达添加到机器人中，我们应该将以下代码添加到diff_wheeled_robot.xacro文件中：

```
<link name="hokuyo_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="${hokuyo_size} ${hokuyo_size} ${hokuyo_size}" />
    </geometry>
    <material name="Blue" />
  </visual>
</link>
<joint name="hokuyo_joint" type="fixed">
  <origin xyz="${base_radius - hokuyo_size/2} 0
${base_height+hokuyo_size/4}" rpy="0 0 0" />
  <parent link="base_link"/>
  <child link="hokuyo_link" />
</joint>
<gazebo reference="hokuyo_link">
  <material>Gazebo/Blue</material>
  <turnGravityOff>false</turnGravityOff>
  <sensor type="ray" name="head_hokuyo_sensor">
    <pose>${hokuyo_size/2} 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>10.0</max>
<resolution>0.001</resolution>
</range>
</ray>
<plugin name="gazebo_ros_head_hokuyo_controller"
filename="libgazebo_ros_laser.so">
<topicName>/scan</topicName>
<frameName>hokuyo_link</frameName>
</plugin>
</sensor>
</gazebo>
```

本节中，我们使用名称为libgazebo_ros_laser.so的GazeboROS插件来仿真激光雷达。完整的代码可以在diff_wheeled_robot_with_laser.xacro描述文件中找到，该文件位于mastering_ros_robot_description_pkg/urdf/文件夹中。

在仿真环境中添加一些物体，这样我们就可以查看激光雷达的数据。在这里，我们在机器人周围添加了一些圆柱体，可以在图4-12中看到相应的激光视图。

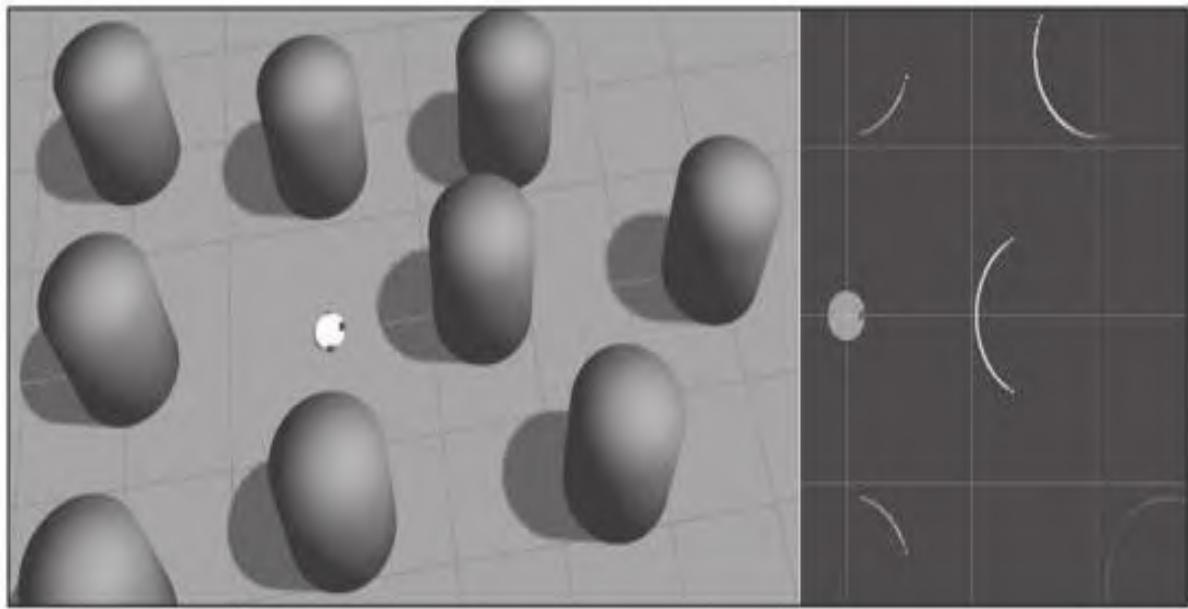


图4-12 差速驱动的机器人位于Gazebo物体中间

激光雷达插件将激光数据（sensor_msgs/LaserScan）发布到/scan话题中。

4.5.2 在Gazebo中控制机器人移动

我们正在使用的是一个差速机器人，配有2个轮子和2个脚轮。该机器人的完整特性应该作为Gazebo-ROS插件来仿真。幸运的是，基本的差速驱动的插件已经实现。

要在Gazebo中控制机器人移动，我们需要添加一个名为 libgazebo_ros_diff_drive.so 的Gazebo-ROS插件，从而可以生成该机器人的差速驱动动作。

以下是该插件的定义及其参数的完整代码片段：

```
<!-- Differential drive controller -->
<gazebo>
  <plugin name="differential_drive_controller"
filename="libgazebo_ros_diff_drive.so">

    <rosDebugLevel>Debug</rosDebugLevel>
    <publishWheelTF>false</publishWheelTF>
    <robotNamespace>/</robotNamespace>
    <publishTf>1</publishTf>
    <publishWheelJointState>false</publishWheelJointState>
    <alwaysOn>true</alwaysOn>
    <updateRate>100.0</updateRate>

    <leftJoint>front_left_wheel_joint</leftJoint>
    <rightJoint>front_right_wheel_joint</rightJoint>

    <wheelSeparation>${2*base_radius}</wheelSeparation>
    <wheelDiameter>${2*wheel_radius}</wheelDiameter>
    <broadcastTF>1</broadcastTF>
    <wheelTorque>30</wheelTorque>
    <wheelAcceleration>1.8</wheelAcceleration>
    <commandTopic>cmd_vel</commandTopic>
    <odometryFrame>odom</odometryFrame>
    <odometryTopic>odom</odometryTopic>
    <robotBaseFrame>base_footprint</robotBaseFrame>

  </plugin>
</gazebo>
```

在该插件中，我们可以提供一些参数，如机器人的车轮关节（关节应该是连续转动型的）、车轮间距、车轮直径、里程计话题等。

控制机器人移动的一个重要参数是：

```
<commandTopic>cmd_vel</commandTopic>
```

该参数是插件的速度指令话题，是ROS中一个Twist类型的消息（sensor_msgs/Twist）。我们可以将Twist消息发布到/cmd_vel话题中，我们就可以看到机器人开始从它的位置移动。

4.5.3 在启动文件中添加关节状态发布者

添加了差速驱动插件之后，我们需要将关节状态发布者加入到现有的启动文件中，或者我们也可以创建一个新的启动文件。你可以在 diff_wheeled_robot_gazebo/launch 下看到更新后的最终启动文件 diff_wheeled_gazebo_full.launch。

启动文件包含关节状态发布者，这有助于在 RViz 中可视化显示。以下是在此启动文件中为关节状态发布者添加的额外代码：

```
<node name="joint_state_publisher" pkg="joint_state_publisher"
      type="joint_state_publisher" ></node>
      <!-- start robot state publisher -->

<node pkg="robot_state_publisher" type="robot_state_publisher"
      name="robot_state_publisher" output="screen" >
    <param name="publish_frequency" type="double" value="50.0" />
</node>
```

4.6 添加ROS遥控节点

ROS遥控（teleop）节点通过接收键盘的输入来发布ROSTwist命令。在该节点中，我们可以生成线速度和角速度，而且已经有一个标准的遥控节点实现，我们可以重用该节点。

遥控节点是在diff_wheeled_robot_control软件包中实现的。脚本文件夹包含diff_wheeled_robot_key节点，它就是遥控节点。和前面一样，你可以从本书提供的代码中获得这个软件包，也可以从以下链接来下载：

```
$ git clone https://github.com/jocacace/diff_wheeled_robot_control.git
```

要想成功编译和使用该软件包，你可能需要安装joy_node软件包：

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

下面是名为keyboard_teleop的启动文件，用来启动遥控节点：

```
<launch>
  <!-- differential_teleop_key already has its own built in velocity
smoother -->
  <node pkg="diff_wheeled_robot_control" type="diff_wheeled_robot_key"
name="diff_wheeled_robot_key"  output="screen">

    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="/cmd_vel"/>
  </node>
</launch>
```

让我们开始控制机器人运动。

使用以下命令启动具有完整仿真设置的Gazebo：

```
$ rosrun diff_wheeled_robot_gazebo diff_wheeled_gazebo_full.launch
```

启动遥控节点：

```
$ rosrun diff_wheeled_robot_control keyboard_teleop.launch
```

启动RViz可视化机器人状态和激光数据：

```
$ rosrun rviz rviz
```

在RViz中添加Fixed Frame: /odom，添加Laser Scan，话题设置为/scan以查看激光扫描数据，添加Robot model来查看机器人模型。

在遥控终端中，我们可以使用一些按键（U、I、O、J、K、L、M、“,”、“.”）进行方向调整，其他键（q、z、w、x、e、c、K、空格键）进行速度调整。图4-13显示机器人使用遥控在Gazebo中移动及其在RViz中的可视化。

我们可以从Gazebo工具栏上选择基本物体，并添加到机器人环境中，也可以在左边的面板上添加在线库中的物体：

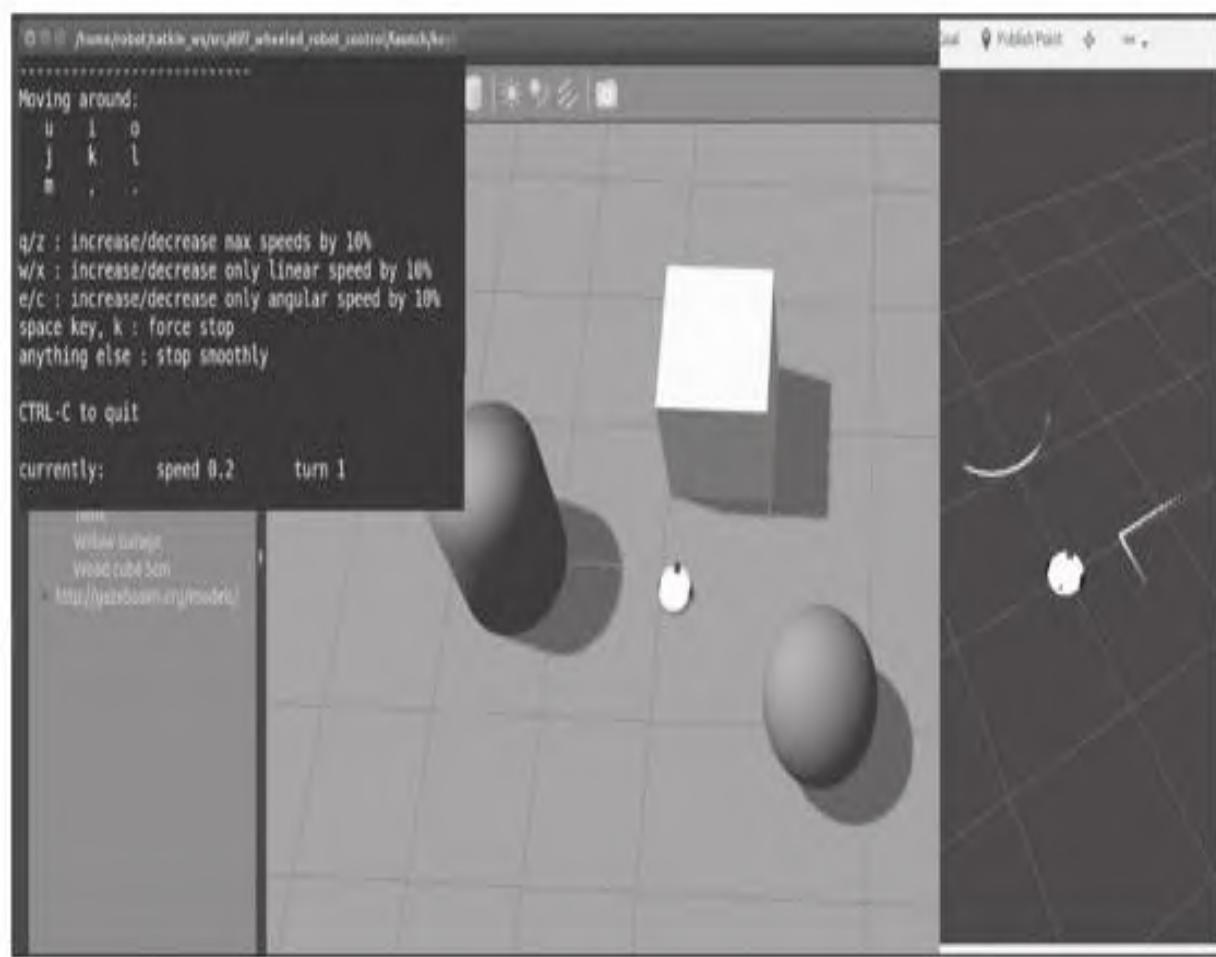


图4-13 在Gazebo中遥控差速驱动机器人

只有当我们按下遥控节点终端内相应的按键时，机器人才会移动。如果该终端处于不活动状态，按下按键机器人不会移动。如果一切正常，我们可以使用机器人来探索该区域并在RViz中可视化激光数据。

4.7 习题

- 为什么要进行机器人仿真?
- 如何在Gazebo仿真环境中添加传感器?
- 有哪些不同类型的ROS控制器和硬件接口?
- 如何在Gazebo仿真环境中移动机器人?

4.8 本章小结

设计好机器人之后，下一阶段就是仿真。仿真有很多用途，可以验证机器人的设计，同时，在没有真正硬件的情况下我们也可以使用机器人。在某些情况下，我们需要在没有机器人硬件的情况下工作。在所有这些情况下，仿真器都很有用。

在本章中，我们尝试仿真了两类机器人，一类是7-DOF机械臂，另一类是差速轮式移动机器人。我们从机械臂开始，讨论了在Gazebo启动机器人所需的额外Gazebo标签。我们讨论了如何在仿真环境中添加3D视觉传感器。随后，我们创建了一个启动文件，启动Gazebo来仿真机械臂，并讨论了如何为每个关节添加控制器。

与机械臂一样，我们为Gazebo仿真创建了URDF，并为激光雷达和差速驱动机制添加了必要的Gazebo-ROS插件。完成仿真模型之后，我们使用自定义的启动文件启动仿真。最后，我们讨论了如何使用遥控节点来移动机器人。

我们可以从下面的链接了解ROS支持的机械臂和移动机器人的更多信息：<http://wiki.ros.org/Robots>。

下一章，我们将学习如何使用另一个著名的机器人仿真软件：V-REP。

第5章

用ROS和V-REP进行机器人仿真

学习了如何使用Gazebo进行机器人仿真后，这一章我们将讨论如何使用另一个强大且著名的仿真软件：虚拟机器人实验平台（Virtual Robot Experimentation Platform, V-REP，<http://www.coppeliarobotics.com>）。

V-REP是由Coppelia Robotics开发的支持多平台的机器人仿真器。针对常见的工业机器人和移动机器人，V-REP提供了许多可直接使用的仿真模型，通过专用的API，各种不同功能可以非常容易集成并组合在一起。另外，V-REP可以通过通信接口与ROS一起运行。该接口让我们可以通过话题（topic）和服务（service）来控制仿真场景和机器人。与Gazebo一样，V-REP可以作为独立的软件来使用，但是需要安装外部的插件与ROS协同运行。

在本章中，我们将学习如何安装设置V-REP仿真器，并安装ROS通信工具，讨论一些初级代码以了解其工作原理。我们将展示如何使用服务和话题与V-REP进行交互，以及如何使用URDF文件来导入和连接新的机器人模型。最后，我们将讨论如何与从V-REP模型数据库中导入的常见移动机器人进行交互，并用附加的传感器增强其功能。

本章将介绍以下内容：

- 安装带有ROS的V-REP
- 理解vrep_plugin
- 使用ROS与V-REP交互
- 使用URDF文件导入机器人模型
- 实现ROS接口来仿真V-REP中的机械臂
- 使用V-REP控制一个移动机器人
- 为仿真机器人增加传感器

5.1 安装带有ROS的V-REP

在开始使用V-REP之前，我们需要在系统中安装它，并且编译相关的ROS包来建立ROS和仿真场景之间的通信桥梁。V-REP是一种跨平台的软件，可用于不同的操作系统，如Windows、macOS和Linux。它由Coppelia Robotics GmbH开发，并且随附免费教育版和商业版使用许可。可以从Coppelia Robotics公司的下载网页（<http://www.coppeliarobotics.com/downloads.html>）下载最新版本的V-REP仿真器，选择Linux版本的V-REP PRO EDU软件。

在本章，我们将使用V-REP的v3.4.0版本。你可以在任何文件夹中使用以下命令来下载此版本（如果网站上可以下载）：

```
$ wget  
http://coppeliarobotics.com/files/V-REP_PRO_EDU_V3_4_0_Linux.tar.gz
```

下载完成后，提取存档：

```
$ tar -zxvf V-REP_PRO_EDU_V3_4_0_Linux.tar.gz
```

为了方便访问V-REP资源，可以设置一个环境变量（VREP_ROOT）指向V-REP的主文件夹：

```
$ echo "export VREP_ROOT=/path/to/v_rep/folder >> ~/.bashrc"
```

V-REP提供以下模式用于从外部应用控制仿真机器人：

- Remote API: V-REP远程API由若干函数组成，这些函数可以由C/C++、Python、Lua或者Matlab开发的外部应用调用。远程API与V-REP交互使用套接字通信。为了将ROS和仿真场景连接起来，可以将V-REP远程API集成到你的C++或者Python节点中。可以在Coppelia Robotics网站上找到V-REP中所有可调用的远程API：

<http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctions.htm>。

要使用远程API，就必须同时实现客户端和服务器端：

- V-REP客户端：客户端位于外部应用中。它可在ROS节点中实现，也可在由V-REP支持的编程语言所编写的标准程序中实现。

- V-REP服务器端：服务器端由V-REP脚本实现。它允许仿真器接收外部的数据来与仿真场景进行交互。

- RosPlugin: V-REP的ROS插件实现了一个高级抽象，可直接将仿真物体与场景和ROS通信系统连接起来。使用此插件，你可以自动使用订阅的消息，并发布来自场景物体的话题，从而获取信息或控制仿真机器人。

- RosInterface: 最新版的V-REP中引入了RosInterface，在以后的版本中该接口将替换RosPlugin。与RosPlugin不同，该模块复制C++ API函数，从而允许ROS与V-REP通信。

这本书中，我们将讨论如何使用RosPlugin与V-REP进行交互。首先要做的是编译ROS通信接口。我们需要在ROS的工作区中添加两个软件包：`vrep_comm`和`vrep_plugin`。你可以从以下GitHub库中下载整个软件包：

```
$ git clone https://github.com/jocacace/vrep_common.git  
$ git clone https://github.com/jocacace/vrep_plugin.git
```

使用`catkin_make`命令编译该软件包。如果一切顺利，编译后将创建`vrep_plugin`共享库：`libv_repExtRos.so`。该文件在ROS工作区的

devel/lib文件夹中。要使V-REP可以使用此库，我们需要将其复制到vrep的主文件夹中：

```
$ cp devel/lib/libv_repExtRos.so $VREP_ROOT
```

该库允许V-REP在启动时与roscore的活动实例建立连接，并将其转换成与整个ROS框架连接的ROS节点。所以要连接V-REP和ROS，必须在启动V-REP之前执行roscore实例。为了测试一切运行良好，先运行roscore，然后启动V-REP软件：

```
$ roscore & $VREP_ROOT/vrep.sh
```

在启动阶段，将加载系统中安装的所有V-REP插件。我们可以检测vrep_plugin是否被加载，如图5-1所示。

```
jcacace@robot:~$ $VREP_ROOT/vrep.sh
Using the default Lua library.
Loaded the video compression library.
Add-on script 'vrepAddOnScript-addOnScriptDemo.lua' was loaded.
Simulator launched.
Plugin 'BubbleRob': loading...
Plugin 'BubbleRob': load succeeded.
Plugin 'Collada': loading...
Plugin 'Collada': load succeeded.
Plugin 'RemoteApi': load succeeded.
Plugin 'Ros': loading...
Plugin 'Ros': load succeeded.
```

图5-1 V-REP启动时加载插件

此外，启动V-REP程序后，将发布一个新的话题，包含仿真状态信息。通过列出活动的话题，我们可以检查/vrep/info话题是否已发布。如图5-2所示，此消息提供了仿真状态的信息（仿真器是否运行，仿真时间信息）。

```
---  
headerInfo:  
  seq: 823  
  stamp:  
    secs: 1504261442  
    nsecs: 363384144  
    frame_id: ''  
simulatorState:  
  data: 1  
simulationTime:  
  data: 41.1496582031  
timeStep:  
  data: 0.0500000007451  
---
```

图5-2 /vrep/info消息的内容

为了探究vrep_plugin的功能，我们可以查看位于vrep_demo_pkg/scene文件夹下的plugin_publisher_subscriber.ttt场景。要打开该场景，请使用主下拉菜单并选择相应的条目：File|Open Scene。此仿真是基于旧版本V-REP提供的rosTopicPublisherAndSubscriber1.ttt场景。

打开此场景后，应显示出仿真窗口，如图5-3所示。

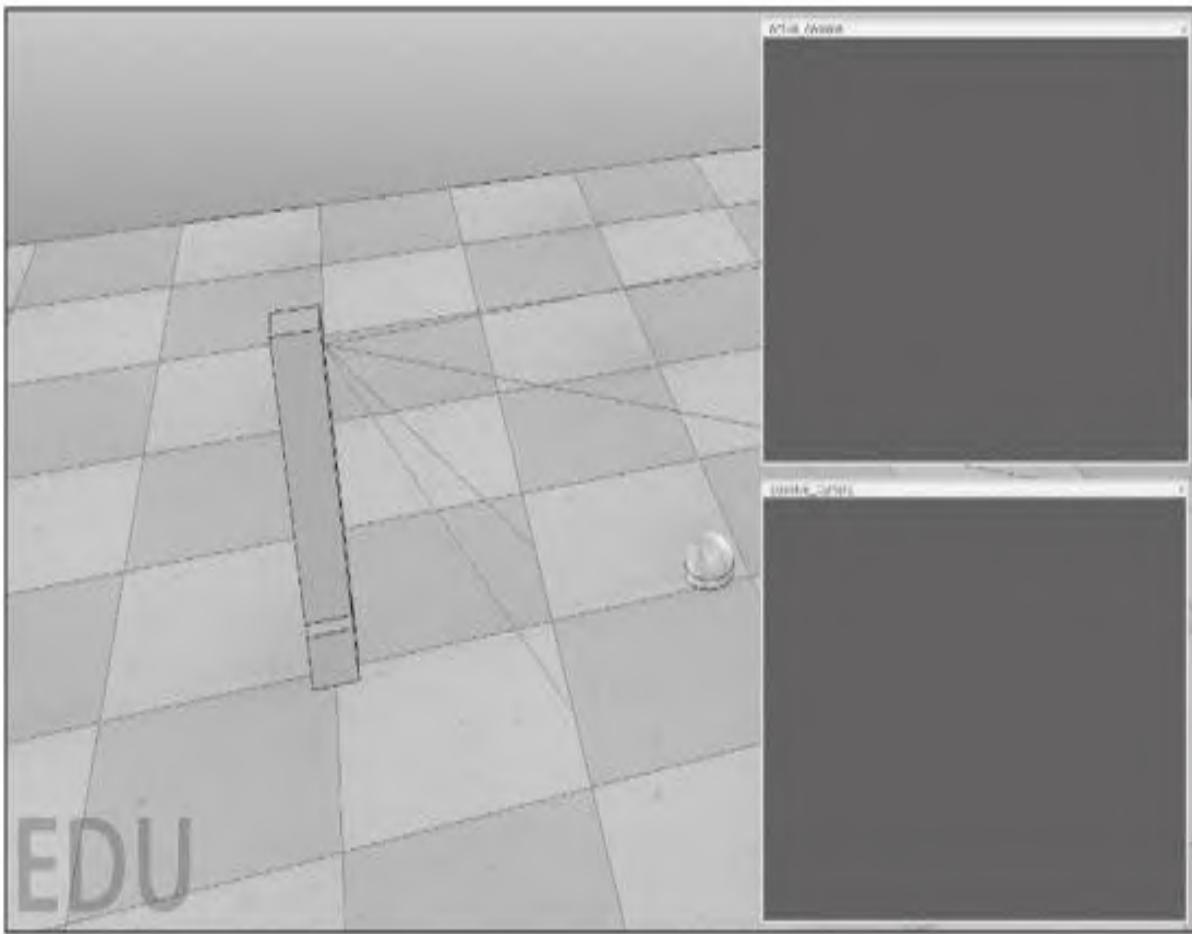


图5-3 plugin_publisher_subscriber.ttt仿真场景

在这个场景中，机器人配备两台相机。一台主动相机从环境获取图像，将视频流发布到特定的话题上。另外一台被动相机只是从同一个话题中获取视频流。我们可以在V-REP界面的主菜单栏按下play按钮。

仿真器启动后，将发生如下变化，如图5-4所示。

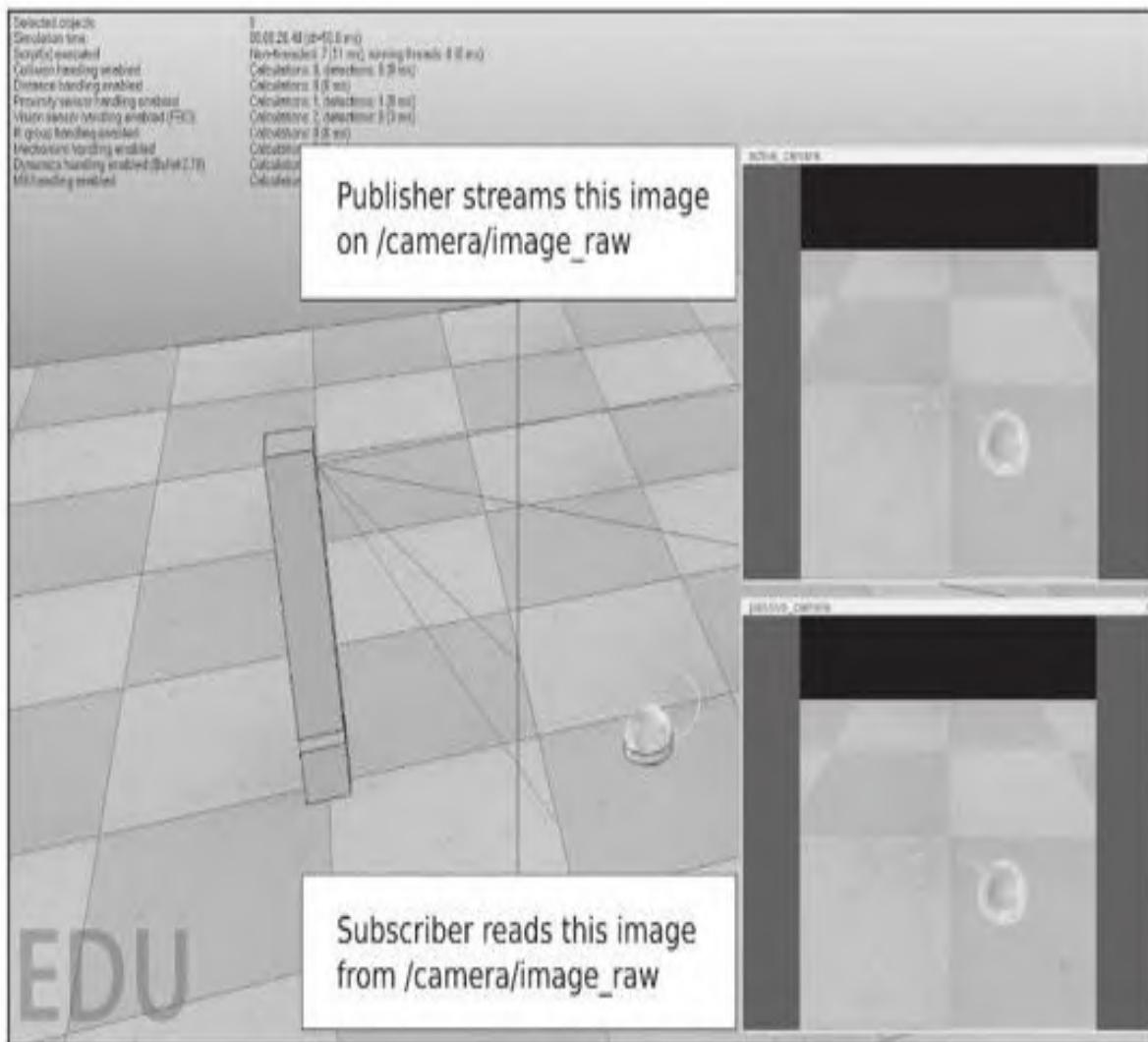


图5-4 发布的图像和订阅示例

在此仿真中，被动相机显示了从主动相机发布的图像，这是直接从ROS软件框架获取视觉数据。我们也可以使用image_view软件包可视化V-REP发布的视频流：

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

5.2 理解vrep_plugin

vrep_plugin是V-REP API框架的一部分。即使插件已经正确安装到了系统上，如果roscore没有运行，加载操作也会失败。如果仿真场景需要vrep_plugin，可是由于roscore没有在运行仿真器之前运行，或者roscore没有安装到系统上，就会弹出一个错误窗口来提示用户，如图5-5所示。

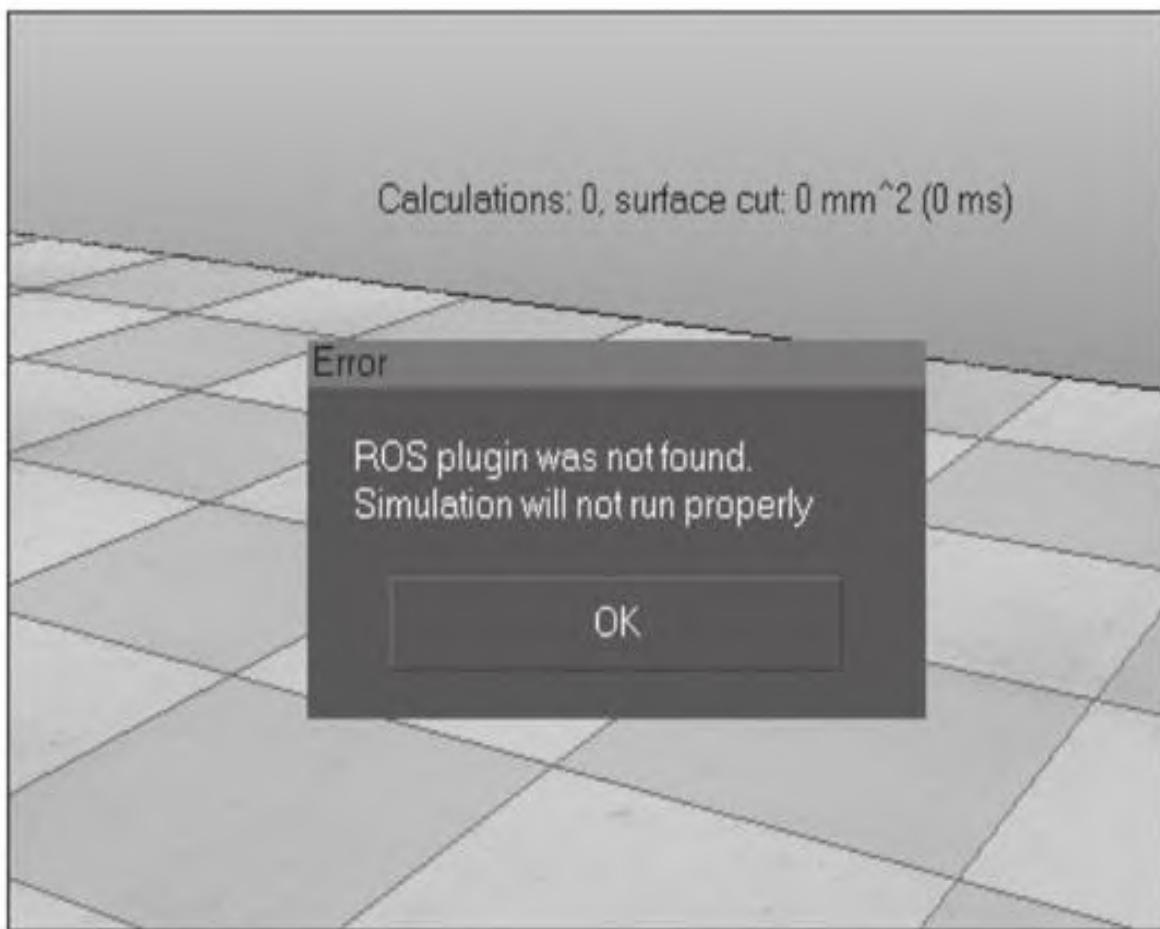


图5-5 没有运行roscore的情况下使用vrep_plugin时会显示错误

正确加载vrep_plugin并启动V-REP后，V-REP将以一个名称为/vrep的ROS节点方式运行。通过如下命令可以显示该节点，节点列表如图5-6所示。

```
$ rosnode list
```

```
jcacace@robot:~$ rosnode list  
/rosout  
/vrep
```

图5-6 运行V-REP后的ROS节点列表

其他ROS节点可以通过以下方式与V-REP通信：

- vrep_plugin提供ROS服务。启动V-REP后可以使用不同的服务来控制仿真场景或其状态。
- vrep_plugin在启用后可以订阅或发布话题。像正常的ROS节点一样，仿真模型可以通过话题来进行通信。

我们可以使用服务与V-REP进行交互。让我们建立一个包含以下依赖项的ROS软件包：

```
$ catkin_create_pkg vrep_demo_pkg roscpp vrep_common std_msgs geometry_msgs
```

或者从GitHub获取整个软件包：

```
$ git clone https://github.com/jocacace/vrep_demo_pkg.git
```

5.2.1 使用ROS服务与V-REP交互

在第一个例子中，我们将使用ROS服务来启动和停止仿真场景。为此，我们必须分别调用/vrep/simRosStartSimulation 和/vrep/simRosStopSimulation服务。我们将讨论位于 vrep_demo_pkg/src文件夹下的start_stop_scene.cpp文件中的源码：

```
#include "ros/ros.h"
#include "vrep_common/simRosStartSimulation.h"
#include "vrep_common/simRosStopSimulation.h"

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

    ros::init( argc, argv, "start_stop_vrep_node");
    ros::NodeHandle n;
    ros::ServiceClient start_vrep_client =
n.serviceClient<vrep_common::simRosStartSimulation>("/vrep/simRosStartSimulation");
    vrep_common::simRosStartSimulation start_srv;

    ros::ServiceClient stop_vrep_client =
n.serviceClient<vrep_common::simRosStopSimulation>("/vrep/simRosStopSimulation");
    vrep_common::simRosStopSimulation stop_srv;

    ROS_INFO("Starting Vrep simulation...");
    if( start_vrep_client.call( start_srv ) ) {
        if( start_srv.response.result == 1 ) {
            ROS_INFO("Simulation started, wait 5 seconds before stop
it!");
            sleep(5);
            if( stop_vrep_client.call( stop_srv ) ) {
                if( stop_srv.response.result == 1 ) {
                    ROS_INFO("Simulation stopped");
                }
            }
            else
                ROS_ERROR("Failed to call /vrep/simRosStopSimulation
service");
        }
    }
    else
        ROS_ERROR("Failed to call /vrep/simRosStartSimulation service");
}
```

下面我们来看一下代码的解析：

```
ros::ServiceClient start_vrep_client =
n.serviceClient<vrep_common::simRosStartSimulation>("/vrep/simRosStartSimulation");
vrep_common::simRosStartSimulation start_srv;

ros::ServiceClient stop_vrep_client =
n.serviceClient<vrep_common::simRosStopSimulation>("/vrep/simRosStopSimulation");
vrep_common::simRosStopSimulation stop_srv;
```

这里，与在第1章中看到的一样，我们声明了服务客户端对象。这些服务分别与vrep_common::simRosStartSimulation和vrep_common::simRosStopSimulation消息类型进行通信。这些服务在返回启动/停止操作的结果（成功或失败）时，无须任何输入值。如果启动操作执行没有错误，我们可以在一段时间后停止仿真。如下面的代码所示：

```
ROS_INFO("Starting Vrep simulation...");  
if( start_vrep_client.call( start_srv ) ) {  
    if( start_srv.response.result != -1 ) {  
        ROS_INFO("Simulation started, wait 5 seconds before stop it!");  
        sleep(5);  
        if( stop_vrep_client.call( stop_srv ) ) {  
            if( stop_srv.response.result != -1 ) {  
                ROS_INFO("Simulation stopped");  
            }  
        }  
        else  
            ROS_ERROR("Failed to call /vrep/simRosStopSimulation  
service");  
    }  
}  
}  
else  
    ROS_ERROR("Failed to call /vrep/simRosStartSimulation service");
```

现在我们可以使用另外一个服务/vrep/simRosAddStatusbarMessage将消息发布到V-REP的状态栏。我们可以将之前的代码做如下改进，文件start_stop_scene_with_msg.cpp的内容如下：

```
int cnt = 0;
while( cnt++ < 5 ) {
    std::stringstream ss;
    ss << "Simulation will stop in " << 6-cnt << " seconds";
    msg_srv.request.message = ss.str();
    if( !msg_client.call( msg_srv ) ) {
        ROS_WARN("Failed to call /vrep/simRosAddStatusbarMessage
service");
    }
    sleep(1);
}
```

这里用到的simRosAddStatusbarMessage服务是用来显示停止仿真之前剩余的秒数。我们可以通过编译和运行start_stop_scene_with_msg.cpp节点来测试此功能。

```
$ rosrun vrep_demo_pkg start_stop_scene_with_msg
```

图5-7显示了该节点在V-REP窗口的输出。

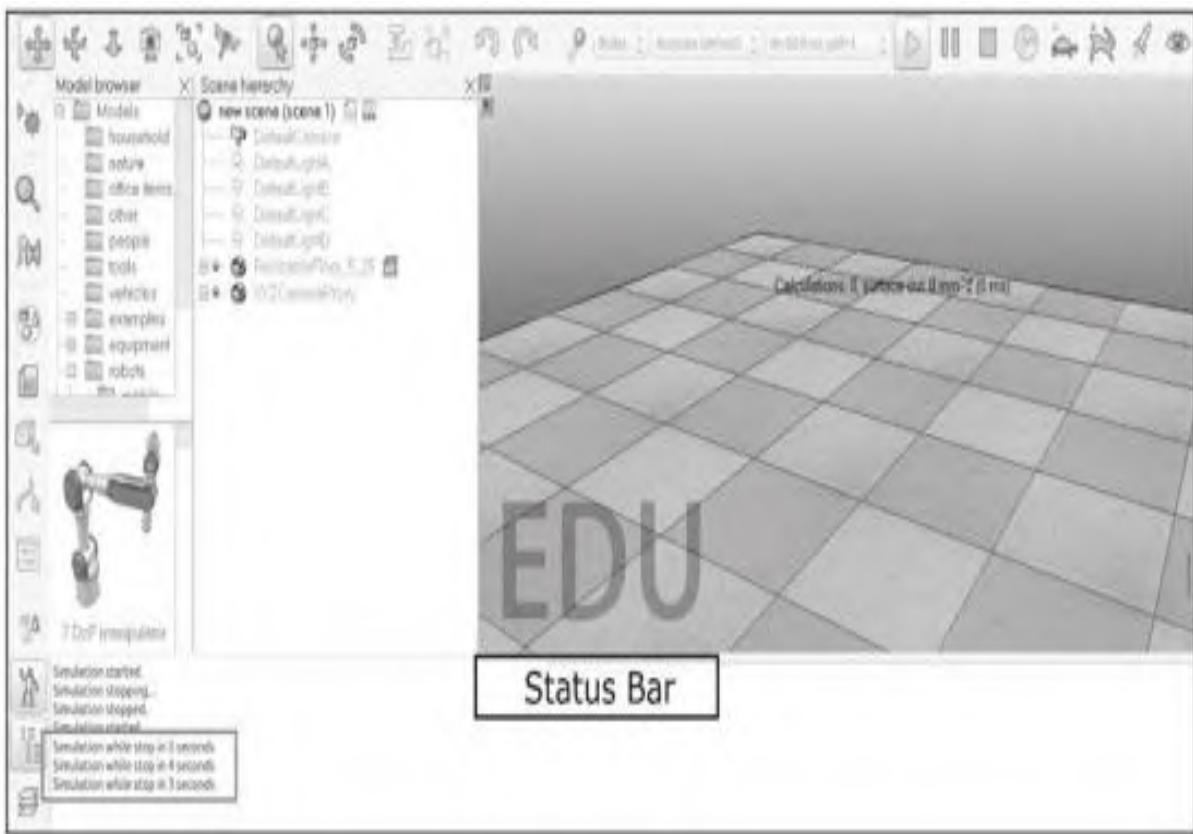


图5-7 通过vrep_plugin向V-REP状态栏发送消息

5.2.2 使用ROS话题与V-REP交互

现在我们将讨论如何用话题与V-REP进行通信。当我们想将消息发送到仿真中的对象或者接收机器人产生的数据时，这个方法是非常有用的。当V-REP启动且服务启用时，话题通信仅在需要时才会发生，并初始化仿真场景中的发布者和订阅者。

对仿真场景进行编程最常用的方法是利用Lua脚本。场景中的每个对象都可以与一个脚本相关联。该脚本在仿真开始时自动调用，并在仿真时间内循环执行。

在下一个示例中，我们将创建一个包含两个对象的场景。其中一个对象将被编程为从特定话题接收浮点型数据，而另一个对象将复制前一个话题上发布的相同数据。

在Scene hierarchy面板上选择下拉菜单，选择选项Add|Dummy。我们可以建立两个对象dummy_publisher和dummy_subscriber，并为每个对象关联一个脚本。在创建的对象上点击鼠标右键，选择选项Add|Associated child script|Non-threaded，如图5-8所示。

或者我们可以打开位于vrep_demo_pkg/scene文件夹内的demo_publisher_subscriber.ttt文件直接加载仿真场景。让我们看一下与dummy_subscriber对象相关联的脚本内容：

```
if (sim_call_type==sim_childscriptcall_initialization) then
    local moduleName=0
    local moduleVersion=0
    local index=0
    local pluginNotFound=true
    while moduleName do
```

```
moduleName,moduleVersion=simGetModuleName(index)
if (moduleName=='Ros') then
    pluginNotFound=false
end
index=index+1
end

if (pluginNotFound) then
simDisplayDialog('Error','ROS plugin was not found.&&nSimulation will not
run properly', sim_dlgstyle_ok, false ,nil,{0.8,0,0,0,0,0},{0.5,0,0,1,1,1})
else
--Enable subscriber to /vrep_demo/float_in topic
simExtROS_enableSubscriber("/vrep_demo/float_in",1,simros_strmcmd_set_float
_signal ,-1,-1,"in")
end
end
if (sim_call_type==sim_childscriptcall_actuation) then
end
if (sim_call_type==sim_childscriptcall_sensing) then
end
if (sim_call_type==sim_childscriptcall_cleanup) then
end
```

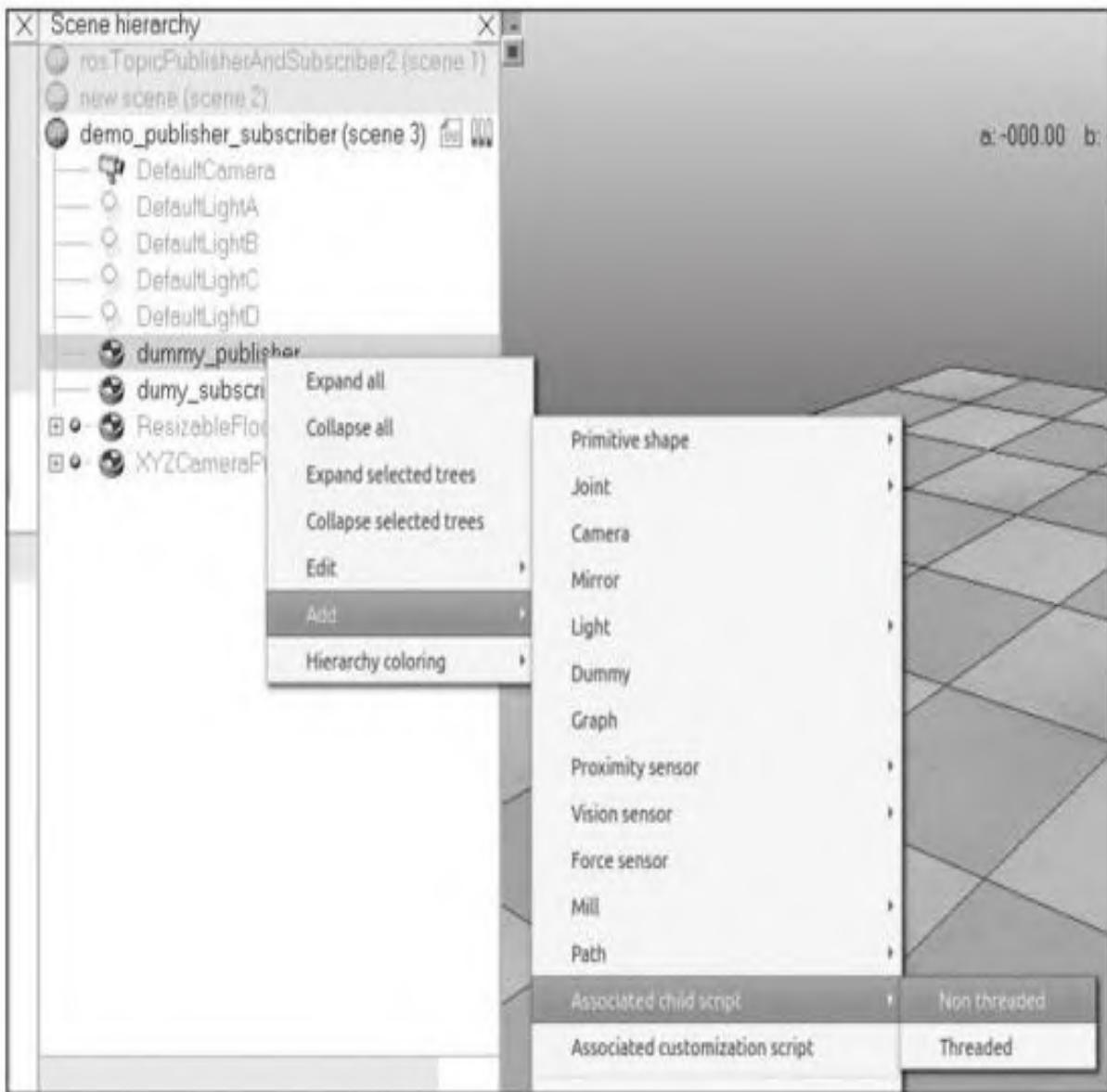


图 5-8 将一个非线程脚本关联到V-REP对象

关联到V-REP对象的每个Lua脚本都包含如下四个部分：

- sim_childscriptcall_initialization：该部分仅在仿真第一次启动时执行；
- sim_childscriptcall_actuation：该部分在仿真时以同样的帧周期循环调用，用户可以在此添加控制机器人动作的代码；

- sim_childscriptcall_sensing: 该部分在每个仿真步骤的传感器探测阶段都会执行;
- sim_childscriptcall_cleanup: 该部分仅在仿真结束前被调用。

让我们看看前面代码的解析:

```
if (sim_call_type==sim_childscriptcall_initialization) then
    local moduleName=0
    local moduleVersion=0
    local index=0
    local pluginNotFound=true
    while moduleName do
        moduleName,moduleVersion=simGetModuleName(index)
        if (moduleName=='Ros') then
            pluginNotFound=false
        end
        index=index+1
    end
```

在初始化部分，我们将检查系统中是否已经安装了vrep_plugin，如果没有安装则会显示错误；

```
simExtROS_enableSubscriber("/vrep_demo/float_in",1,simros_strmcmd_set_float
_signal,-1,-1,"in")
```

这将激活/vrep_demo/float_in话题上的输入浮点数的订阅者功能。simExtROS_enableSubscriber函数需要如下参数：话题名称、所需的队列大小、要传输的类型以及三个启用参数。这些参数指明了数据将被应用到的项目。例如，我们想要设置关节对象的位置，第一个参数是对象的句柄，其他参数将不会被使用。在示例中，我们希望将从话题接收的数据保存到变量in中。

下面我们来看与dummy_publisher对象关联的脚本内容：

```
if (sim_call_type==sim_childscriptcall_initialization) then
    -- Check if the required plugin is there (libv_repExtRos.so or
    libv_repExtRos.dylib):
        local moduleName=0
        local moduleVersion=0
        local index=0
        local pluginNotFound=true
        while moduleName do
            moduleName,moduleVersion=simGetModuleName(index)
            if (moduleName=='Ros') then
                pluginNotFound=false
            end
            index=index+1
        end
        if (pluginNotFound) then
            simDisplayDialog('Error','ROS plugin was not found.&&nSimulation
will not run properly', sim_dlgstyle_ok, false, nil,
{0.8,0,0,0,0,0},{0.5,0,0,1,1,1})
        else
            simExtROS_enablePublisher("/vrep_demo/float_out",
1,simros_strmcmd_get_float_signal, -1,-1,"out")
        end
```

```
    end
end

if (sim_call_type==sim_childscriptcall_actuation) then
    --Get value of input signal and publish it on /vrep_demo/float_out topic
    data = simGetFloatSignal('in')
    if( data ) then
        simSetFloatSignal("out",data)
        simAddStatusbarMessage(data)
    end
end
if (sim_call_type==sim_childscriptcall_sensing) then
    -- Put your main SENSING code here
end
if (sim_call_type==sim_childscriptcall_cleanup) then
    -- Put some restoration code here
End
```

下面对代码进行解析：

```
simExtROS_enablePublisher("/vrep_demo/float_out",
1,simros_strmcmd_get_float_signal, -1,-1,"out")
```

这段代码是在检查vrep_plugin是否安装之后，我们将启用浮点数据发布者。在这段代码后，该脚本会连续地发布out变量中的数据：

```
if (sim_call_type==sim_childscriptcall_actuation) then
    --Get value of input signal and publish it on /vrep_demo/float_out topic
    data = simGetFloatSignal('in')
    if( data ) then
        simSetFloatSignal("out",data)
        simAddStatusbarMessage(data)
    end
end
```

最后我们将out变量的值设置为从/vrep_demo/float_in话题中接收的值，并将其存储在in变量中。注意这里的in和out变量是从场景的所有脚本中都能访问的特殊全局变量。这种变量在V-REP中被称为信号。

在仿真器运行后，我们可以使用以下命令检查一切是否运行正常，在输入话题上发布所需的数据，并监控输出话题的输出：

```
$ rostopic pub /vrep_demo/float_in std_msgs/Float32 "data: 2.0" -r 12
$ rostopic echo /vrep_demo/float_out
```

5.3 使用V-REP和ROS仿真机械臂

在上一章，我们用Gazebo导入和仿真了第3章中设计的7-DOF机械臂。这里我们用V-REP来做同样的事情。仿真7-DOF机械臂的第一步是将它导入到仿真场景中。V-REP允许使用URDF文件导入新的机器人。因此我们要将机械臂的xacro模型转换为一个URDF文件。将生成的URDF文件保存到vrep_demo_pkg软件包存放URDF文件的文件夹中：

```
$ rosrun xacro xacro seven_dof_arm.xacro -inorder >  
/path/to/vrep_demo_pkg/urdf/seven_dof_arm.urdf
```

现在我们可以使用URDF导入插件来导入机器人模型。在主界面上的下拉列表中选择选项：Plugins|URDF import。在按下导入按钮后，在对话框中选择默认导入选择。最后选择要导入的文件，接着在场景中将显示7-DOF机器臂，如图5-10所示。

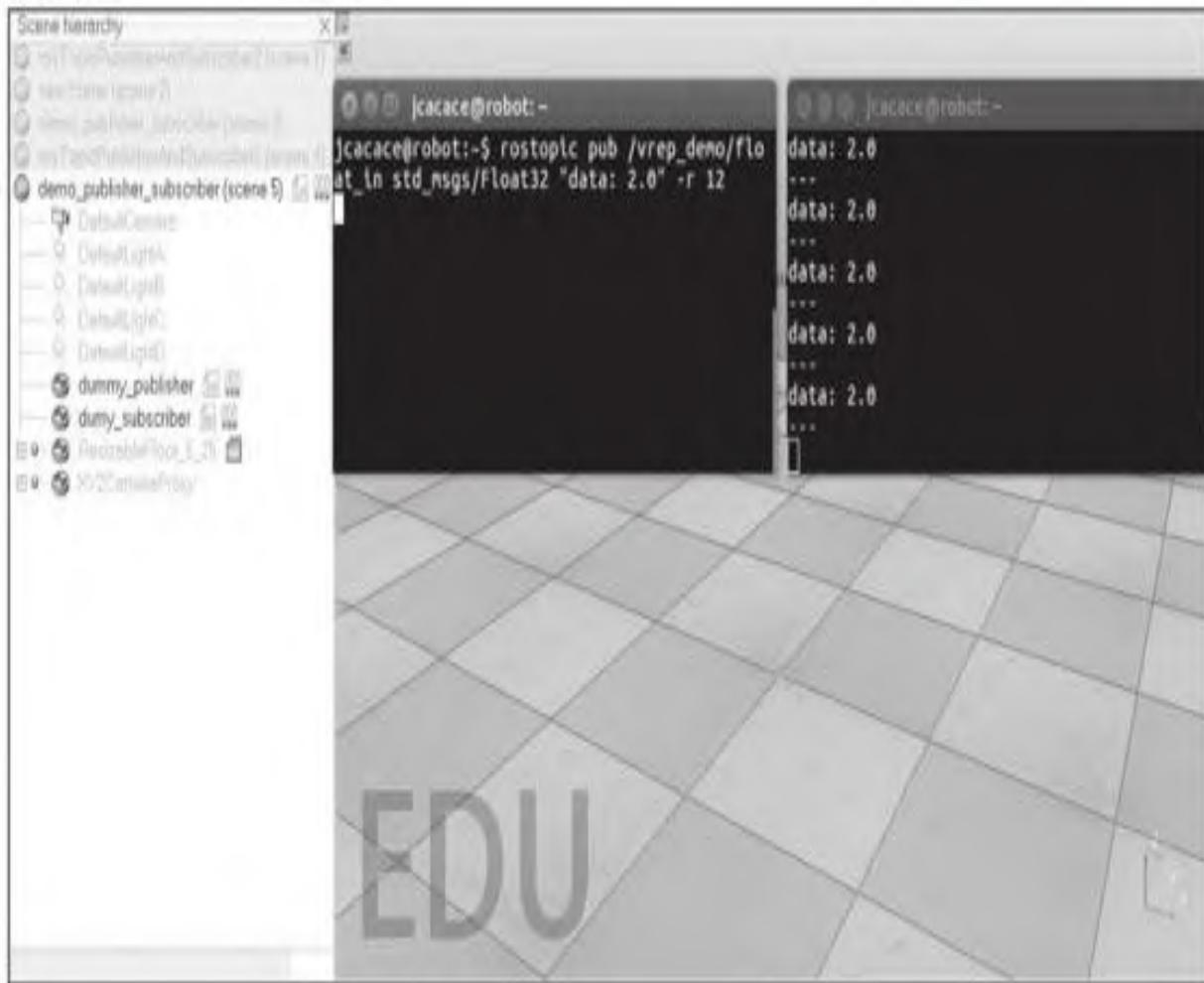


图 5-9 运行V-REP发布者和订阅者示例

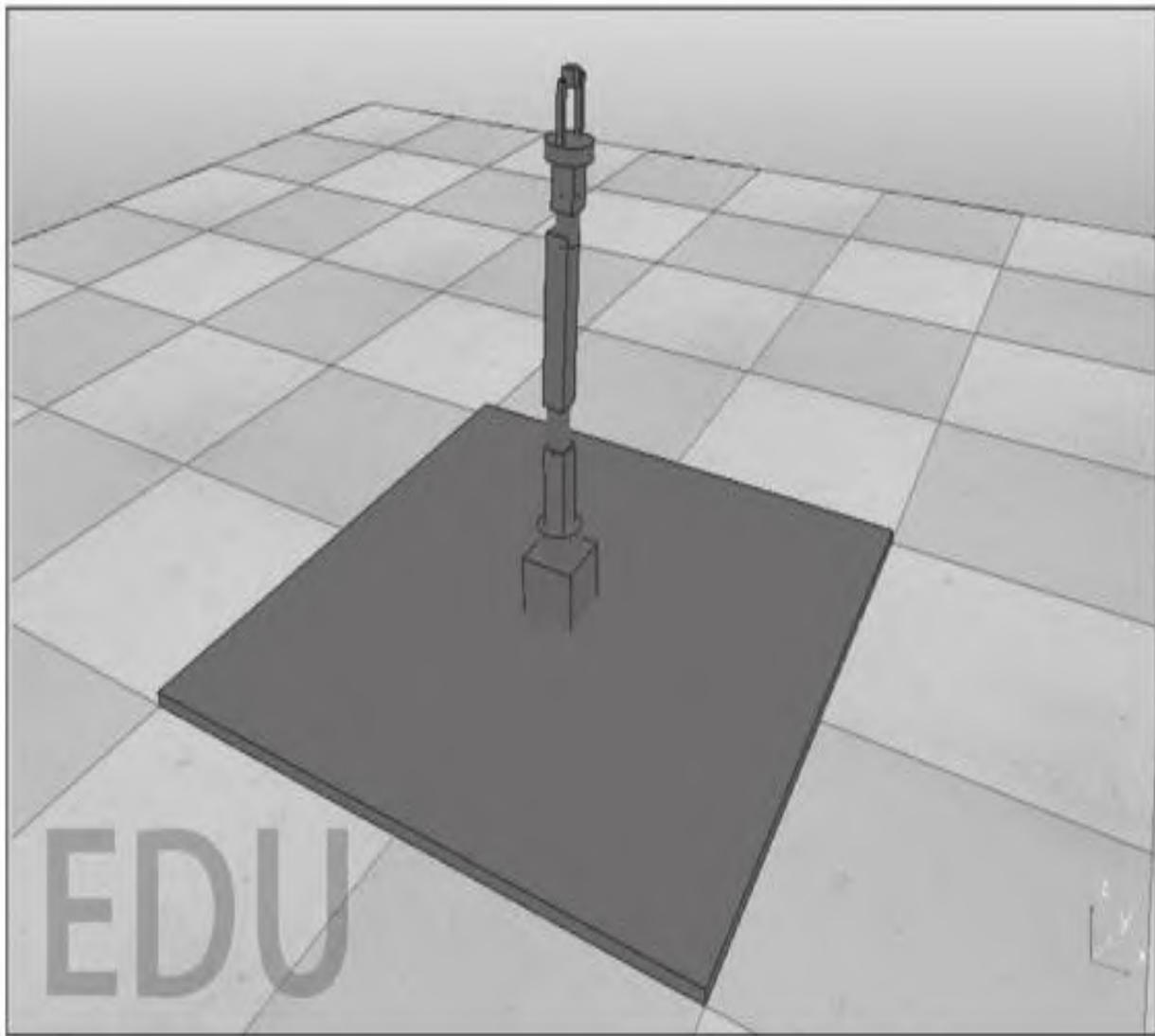


图5-10 在V-REP中仿真7-DOF机械臂

现在机器人的所有部件都导入场景中了。在scene hierarchy面板中我们可以看到URDF文件中定义的机器人关节和连杆的集合。

即使机器人模型已被正确导入，我们仍不能立即对它进行控制。为了启动机器人，我们需要在Joint Dynamic Properties面板中开启所有机器人马达。如果马达被禁用，我们就不能在场景中移动机器人了。若要启用一个关节的马达，我们要打开Scene object properties面板，在面板的主下拉菜单中选择选项：Tools|Scene object properties。你也可以在scene hierarchy面板中双击对象图标来打开此对话框。在这个新窗口中，我们可以打开动态属性对话

框，启用马达和关节的控制环路，并选择控制类型。默认方式下马达是通过PID来控制的，如图5-11所示。

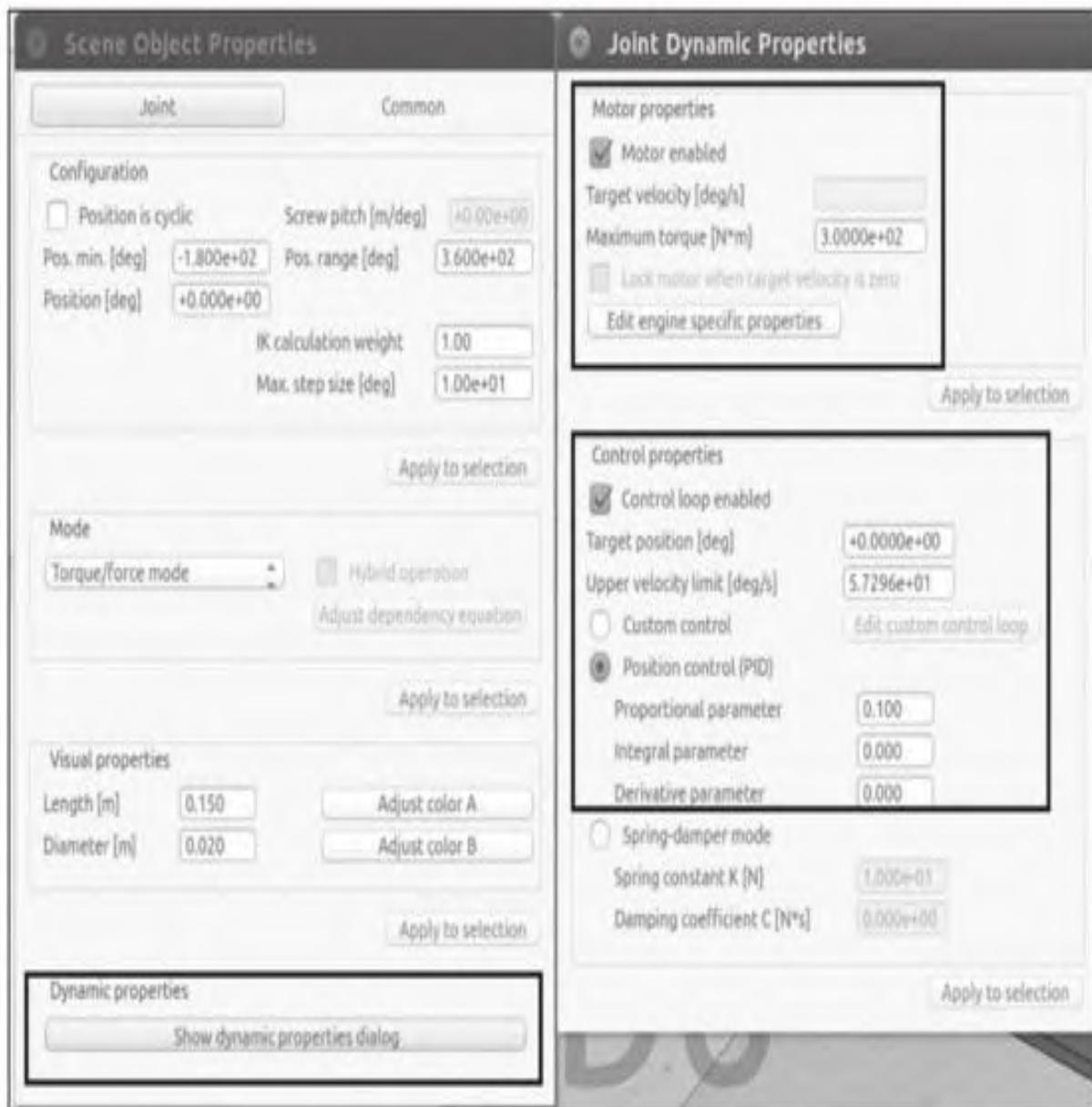


图5-11 场景对象属性和关节动态属性对话框

为了提高控制环路的性能，应适当调整PID增益。为所有机器人关节启用马达和控制环路后，我们可以检查是否所有功能都已经被正确配置。启动仿真并在Scene Object Properties面板中设置一个目标位置。

图5-12是将第4个关节移动到1.0弧度的示例。

在V-REP关节控制器中增加ROS接口

在本节，我们将学习如何将7-DOF机械臂与vrep_plugin插件连接，以便用流式传输其关节状态并通过话题接收控制输入。如前面的示例所示，选择机器人的一个部件（如base_link_respondable）并建立一个Lua脚本来管理V-REP和ROS之间的通信。

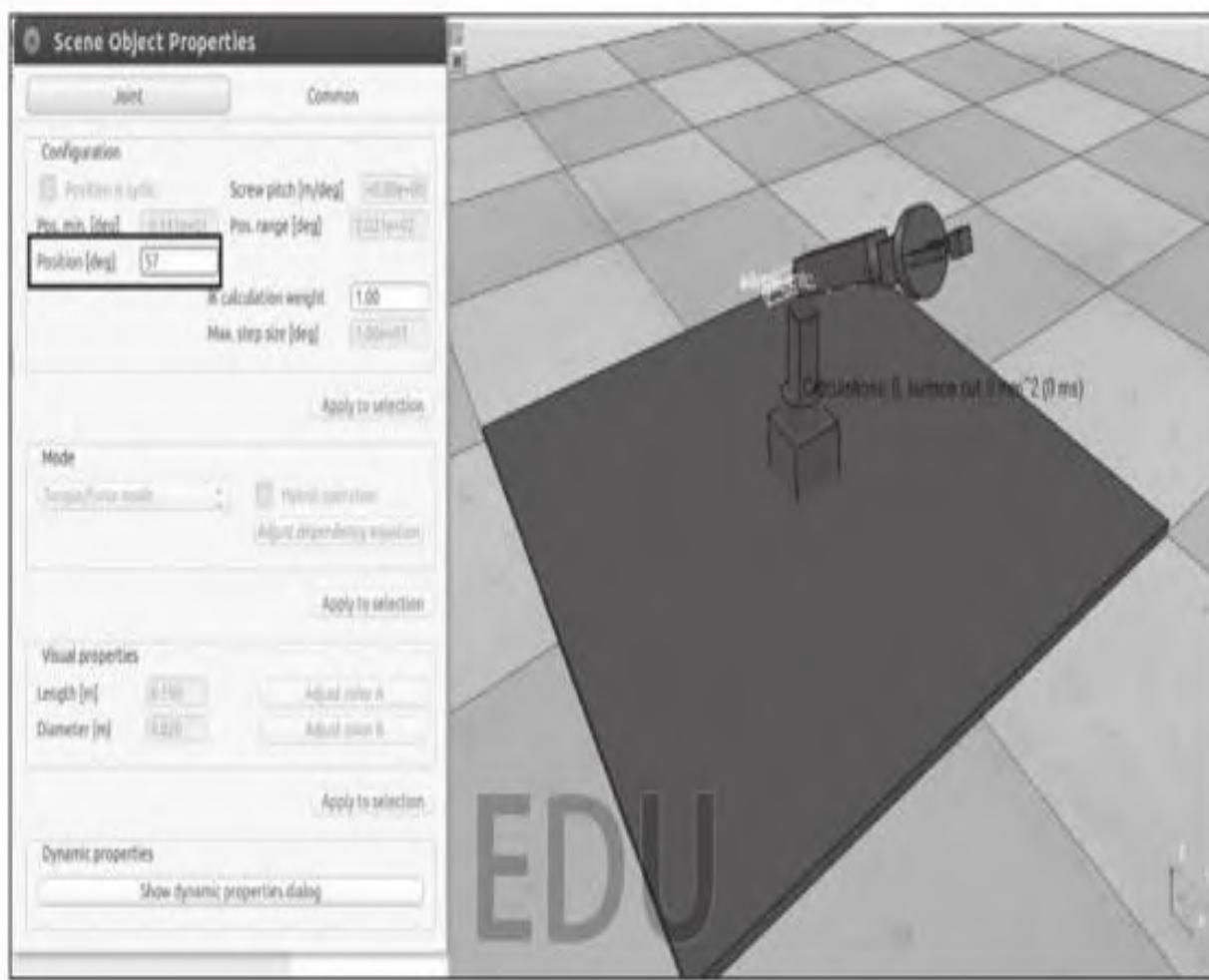


图5-12 在V-REP对象属性对话框中控制机械臂的关节的运动

下面是脚本的源代码：

```
if (sim_call_type==sim_childscriptcall_initialization) then
    -- Check if the required plugin is there (libv_repExtRos.so or
    libv_repExtRos.dylib):
    local moduleName=0
    local moduleVersion=0
    local index=0
    local pluginNotFound=true

    while moduleName do
        moduleName,moduleVersion=simGetModuleName(index)
        if (moduleName=='Ros') then
            pluginNotFound=false
        end
        index=index+1
    end

    if (pluginNotFound) then
        -- Display an error message if the plugin was not found:
        simDisplayDialog('Error','ROS plugin was not found.&&nSimulation
will not run
properly',sim_dlgstyle_ok,false,nil,{0.8,0,0,0,0,0},{0.5,0,0,1,1,1})
    else
        -- Retrive the handle of all joints
        shoulder_pan_handle=simGetObjectHandle('shoulder_pan_joint')
        shoulder_pitch_handle=simGetObjectHandle('shoulder_pitch_joint')
        elbow_roll_handle=simGetObjectHandle('elbow_roll_joint')
        elbow_pitch_handle=simGetObjectHandle('elbow_pitch_joint')
        wrist_roll_handle=simGetObjectHandle('wrist_roll_joint')
        wrist_pitch_handle=simGetObjectHandle('wrist_pitch_joint')
        gripper_roll_handle=simGetObjectHandle('gripper_roll_joint')

        -- Enable joint publishing
```

```
simExtROS_enablePublisher('/vrep_demo/seve_dof_arm/shoulder_pan/state', 10,
simros_strmcmd_get_joint_state,shoulder_pan_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/shoulder_pitch/state',
10, simros_strmcmd_get_joint_state,shoulder_pitch_handle,0,'')
simExtRS_enablePublisher('/vrep_demo/seven_dof_arm/elbow_roll/state', 10,
simros_strmcmd_get_joint_state,elbow_roll_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/elbow_pitch/state', 10,
simros_strmcmd_get_joint_state,elbow_pitch_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/wrist_roll/state', 10,
simros_strmcmd_get_joint_state,wrist_roll_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/wrist_pitch/state', 10,
simros_strmcmd_get_joint_state,wrist_pitch_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/gripper_roll/state',
10, simros_strmcmd_get_joint_state,gripper_roll_handle,0,'')
--Enable joint subscriber
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/shoulder_pan/ctrl',
10, simros_strmcmd_set_joint_target_position, shoulder_pan_handle, 0, '')
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/shoulder_pitch/ctrl',
10, simros_strmcmd_set_joint_target_position, shoulder_pitch_handle, 0, '')
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/elbow_roll/ctrl', 10,
simros_strmcmd_set_joint_target_position, elbow_roll_handle, 0, '')
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/elbow_pitch/ctrl', 10,
simros_strmcmd_set_joint_target_position, elbow_pitch_handle, 0, '')
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/wrist_roll/ctrl', 10,
simros_strmcmd_set_joint_target_position, wrist_roll_handle, 0, '')
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/wrist_pitch/ctrl', 10,
simros_strmcmd_set_joint_target_position, wrist_pitch_handle, 0, '')
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/gripper_roll/ctrl',
10, simros_strmcmd_set_joint_target_position, gripper_roll_handle, 0, '')
end
end
if (sim_call_type==sim_childscriptcall_cleanup) then
end
if (sim_call_type==sim_childscriptcall_sensing) then
end
if (sim_call_type==sim_childscriptcall_actuation) then
end
```

下面我们来对代码进行解析。在检查了vrep_plugin是否正确安装后，我们为每个机械臂的关节初始化了一个对象句柄：

```
shoulder_pan_handle=simGetObjectHandle('shoulder_pan_joint')
shoulder_pitch_handle=simGetObjectHandle('shoulder_pitch_joint')
elbow_roll_handle=simGetObjectHandle('elbow_roll_joint')
elbow_pitch_handle=simGetObjectHandle('elbow_pitch_joint')
wrist_roll_handle=simGetObjectHandle('wrist_roll_joint')
wrist_pitch_handle=simGetObjectHandle('wrist_pitch_joint')
gripper_roll_handle=simGetObjectHandle('gripper_roll_joint')
```

这里我们使用了simGetObjectHandle函数，其参数是我们想处理的scene hierarchy面板中出现的对象名称。现在我们就可以启动关节状态发布者了：

```
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/shoulder_pan/state', 1,
simros_strmcmd_get_joint_state,shoulder_pan_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/shoulder_pitch/state',
1, simros_strmcmd_get_joint_state,shoulder_pitch_handle,0,'')
simExtRS_enablePublisher('/vrep_demo/seven_dof_arm/elbow_roll/state', 1,
simros_strmcmd_get_joint_state,elbow_roll_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/elbow_pitch/state', 1,
simros_strmcmd_get_joint_state,elbow_pitch_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/wrist_roll/state', 1,
simros_strmcmd_get_joint_state,wrist_roll_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/wrist_pitch/state', 1,
simros_strmcmd_get_joint_state,wrist_pitch_handle,0,'')
simExtROS_enablePublisher('/vrep_demo/seven_dof_arm/gripper_roll/state', 1,
simros_strmcmd_get_joint_state,gripper_roll_handle,0,'')
```

这段代码使用simExtROS_enablePublisher函数和simros_strmcmd_get_joint_state参数，使得V-REP可以使用sensor_msgs::JointState消息类型流式传输通过其对象句柄（函数中的第4个参数）指定的关节状态，如图5-13所示。

```
---  
header:  
  seq: 11900  
  stamp:  
    secs: 1504564905  
    nsecs: 995165677  
    frame_id: ''  
  name: ['elbow_roll_joint']  
  position: [-3.712777470354922e-06]  
  velocity: [-0.0002352813316974789]  
  effort: [-0.7412756085395813]  
---
```

图5-13 vrep_plugin发布的关节状态

```
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/shoulder_pan/ctrl',  
10, simros_strmcmd_set_joint_target_position, shoulder_pan_handle, 0, '')  
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/shoulder_pitch/ctrl',  
10, simros_strmcmd_set_joint_target_position, shoulder_pitch_handle, 0, '')  
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/elbow_roll/ctrl', 10,  
simros_strmcmd_set_joint_target_position, elbow_roll_handle, 0, '')  
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/elbow_pitch/ctrl', 10,  
simros_strmcmd_set_joint_target_position, elbow_pitch_handle, 0, '')  
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/wrist_roll/ctrl', 10,  
simros_strmcmd_set_joint_target_position, wrist_roll_handle, 0, '')  
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/wrist_pitch/ctrl', 10,  
simros_strmcmd_set_joint_target_position, wrist_pitch_handle, 0, '')  
simExtROS_enableSubscriber('/vrep_demo/seven_dof_arm/gripper_roll/ctrl',  
10, simros_strmcmd_set_joint_target_position, gripper_roll_handle, 0, '')
```

最后我们启用机械臂来获取用户的控制输入。使用
`simros_strmcmd_set_joint_target_position`命令调用
`simExtROS_enableSubscriber`函数，使机械臂能够订阅一组浮点数据流。获取的数据将自动通过对象句柄应用在指定的关节上。

像通常一样，我们可以通过给机器人的关节设置一个目标位置来测试每项功能是否一切正常：

```
$ rostopic pub /vrep_demo/seven_dof_arm/wrist_pitch/ctrl std_msgs/Float64  
"data: 1.0"
```

5.4 在V-REP下仿真差速轮式机器人

仿真了机械臂后，现在我们将讨论如何建立一个仿真场景来控制移动机器人。在本例中，我们将使用一种在V-REP中已实现了的仿真模型。为了从V-REP数据库中导入模型，需要在Model Browser面板上选择所需的模型类和对象，然后将其拖入场景中。

这里我们将仿真Pioneer 3dx，一种作为研究平台的常见的轮式移动机器人之一，如图5-14所示。

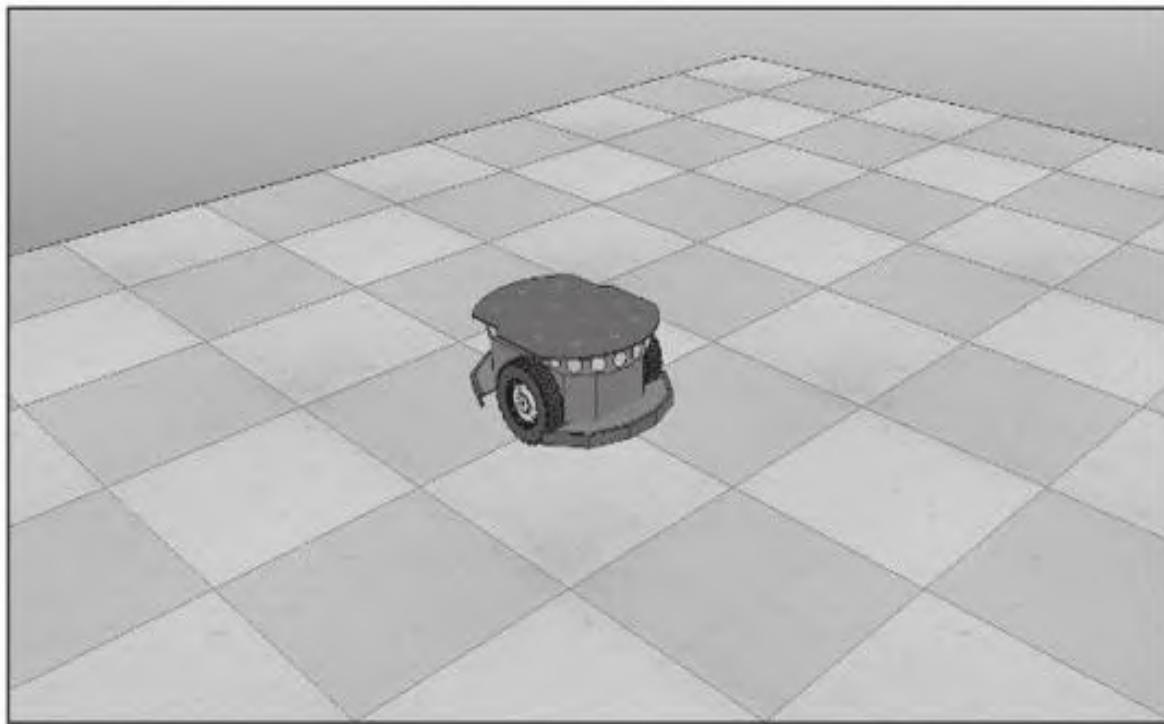


图5-14 在V-REP仿真场景中的Pioneer 3dx

默认情况下，Pioneer机器人配有16个超声波传感器，包括朝前的和朝后的。在下一部分，我们将讨论如何为机器人配备其他的传感器。

为了使用ROS驱动机器人，我们要添加一个脚本来接收所需的线速度和角速度，并将其转化为车轮速度。我们可以使用相同的脚本通过ROS话题启用传感器的数据流。

下面是Scene Hierarchy面板中与Pioneer_p3dx对象关联的全部脚本代码。此代码的一部分已经和V-REP仿真器一起发布了，当Pioneer_p3dx导入到仿真场景中时就已经存在了：

```
if (sim_call_type==sim_childscriptcall_initialization) then
    usensors={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}
    sonarpublisher={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}
    for i=1,16,1 do
        usensors[i]=simGetObjectHandle("Pioneer_p3dx_ultrasonicSensor"..i)
        sonarpublisher[i] = simExtROS_enablePublisher('/sonar'..i , 0,
            simros_strmcmd_read_proximity_sensor, usensors[i], -1, '')
    end

    motorLeft=simGetObjectHandle("Pioneer_p3dx_leftMotor")
    motorRight=simGetObjectHandle("Pioneer_p3dx_rightMotor")

--Input
    simExtROS_enableSubscriber('/linear_vel', 0,
        simros_strmcmd_set_float_signal, -1, -1, 'vl')
    simExtROS_enableSubscriber('/angular_vel', 0,
        simros_strmcmd_set_float_signal, -1, -1, 'va')
```

```

--output
robotHandler=simGetObjectHandle('Pioneer_p3dx') -- body position
odomPublisher=simExtROS_enablePublisher('/odometry',1,simros_strmcmd_get_od
om_data,robotHandler,-1,'')
axes_length = 0.331;
wheel_radius = 0.0970;
end

if (sim_call_type==sim_childscriptcall_cleanup) then
end

if (sim_call_type==sim_childscriptcall_actuation) then
    local v_l = simGetFloatSignal( 'vl' )
    local v_a = simGetFloatSignal( 'va' )

    if not v_l then
        v_l = 0.0
    end

    if not v_a then
        v_a = 0.0
    end
    local v_left = 0.0
    local v_right = 0.0
    v_left = ( 1/wheel_radius)*(v_l-axes_length/2*v_a)
    v_right = ( 1/wheel_radius)*(v_l+axes_length/2*v_a)

    simSetJointTargetVelocity(motorLeft,v_left)
    simSetJointTargetVelocity(motorRight, v_right)
end

```

下面是对代码的解析：

```
for i=1,16,1 do
    usensors[i]=simGetObjectHandle("Pioneer_p3dx_ultrasonicSensor"..i)
    sonarpublisher[i] = simExtROS_enablePublisher('/sonar'..i , 0,
        simros_strmcmd_read_proximity_sensor, usensors[i], -1, '')
    end
    motorLeft=simGetObjectHandle("Pioneer_p3dx_leftMotor")
    motorRight=simGetObjectHandle("Pioneer_p3dx_rightMotor")
```

这里我们流式传输超声波数据并初始化要控制的车轮的处理程序：

```
simExtROS_enableSubscriber('/linear_vel', 0,
    simros_strmcmd_set_float_signal, -1, -1, 'vl')
    simExtROS_enableSubscriber('/angular_vel', 0,
    simros_strmcmd_set_float_signal, -1, -1, 'va')

robotHandler=simGetObjectHandle('Pioneer_p3dx') -- body position
odomPublisher=simExtROS_enablePublisher('/odometry',1,simros_strmcmd_get_odom_data,robotHandler,-1,'')
```

这里允许机器人从ROS话题中获取所需的笛卡儿速度并流式传输其里程信息。`simros_strmcmd_set_float_signal`命令用来从`/linear_vel`和`/angular_vel`话题中读取一个浮点数据。而`simros_strmcmd_get_odom_data`命令通过`nav_msgs::Odom`消息启用机器人的里程计数据流。

在脚本的驱动部分，我们可以计算车轮的速度：

```
local v_l = simGetFloatSignal( 'vl' )
local v_a = simGetFloatSignal( 'va' )
```

这段代码从输入话题中找到所需的线速度和角速度的信号值：

```
if not v_l then  
    v_l = 0.0  
end  
if not v_a then  
    v_a = 0.0  
end
```

由于默认情况下会使用空值创建信号，因此建议在使用它们之前检查是否已初始化：

```
v_left = (1/wheel_radius)*(v_l-axes_length/2*v_a)  
v_right = (1/wheel_radius)*(v_l+axes_length/2*v_a)  
simSetJointTargetVelocity(motorLeft, v_left)  
simSetJointTargetVelocity(motorRight, v_right)
```

最后，根据要求的控制输入，我们可以计算并设置驱动机器人所需的车轮速度。在我们的仿真中，轮子是由简单的关节表示的。我们可以用simSetJointTargetVelocity控制它运动，该函数为关节设置所需的目标速度。

运行仿真之后，用户应该能够读取超声波数据和机器人里程计计算出来的位置，并且可以设置线速度和角速度：

```
$ rostopic pub /linear_vel std_msgs/Float32 "data: 0.2"
```

上面的命令将机器人的线速度设置为0.2m/s。

5.4.1 在V-REP中添加激光传感器

机器人仿真器除了可以仿真机器人，还有一个重要功能是可以仿真各种传感器。现在我们给Pioneer_3dx添加一个激光传感器。V-REP提供了不同的视觉、惯性和接近传感器。这些可以在components|sensors的Model Browser面板上选择。

要给机器人添加一个激光传感器，请选择Hokuyo_URG_04LX_UG01_ROS，并拖曳到场景中。将这个传感器放置在机器人小车顶层很有用，可用于在机器人的坐标系中定位。导入该传感器之后，应该会在Scene Hierarchy中显示出来，如图5-15所示。

我们可以通过工具栏中的Object/Item position/orientation窗口定位机器人小车上的传感器位置。该激光传感器可以用sensor_msgs::LaserScan消息在/front_scan话题上传输激光数据。启动仿真后，我们还可以在RViz中查看激光传感器生成的数据，如图5-16所示。

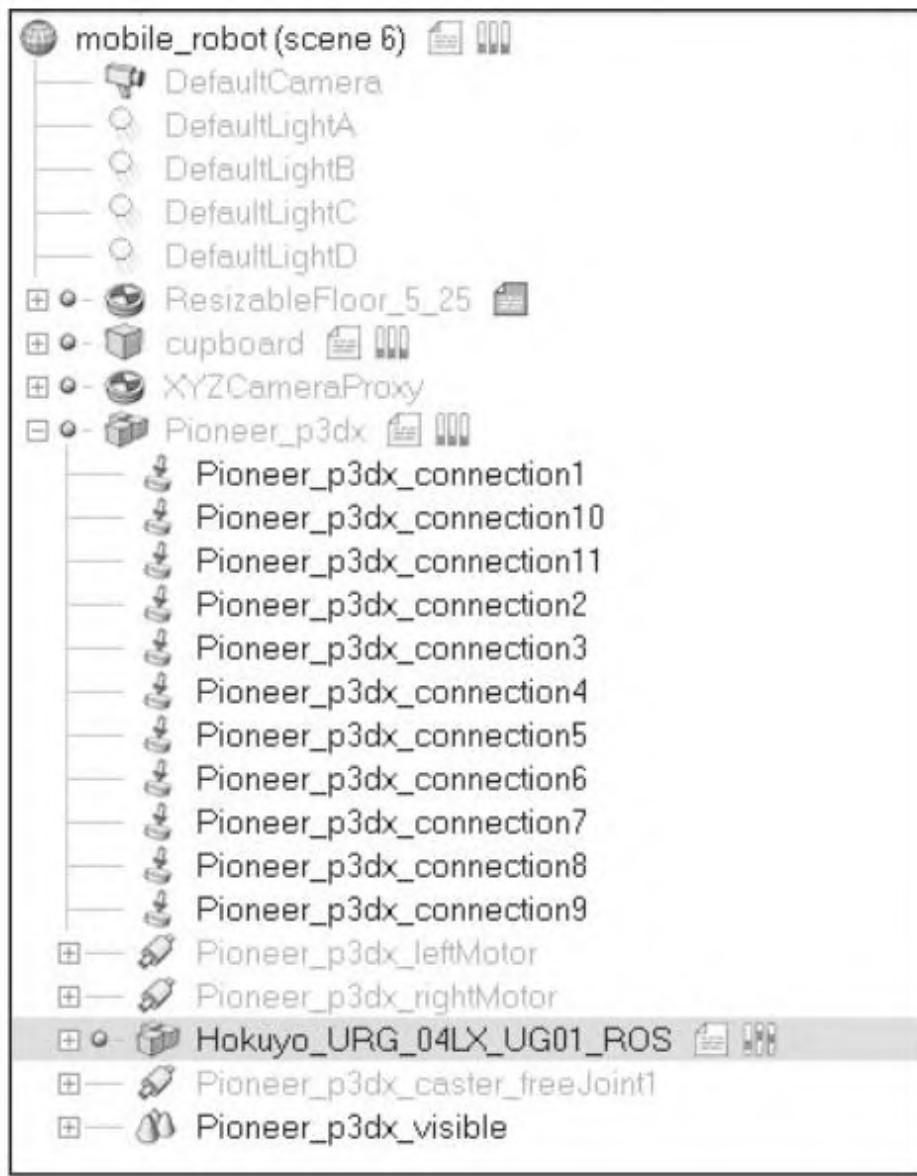


图5-15 Pioneer配有Hokuyo激光传感器时的Scene Hierarchy面板

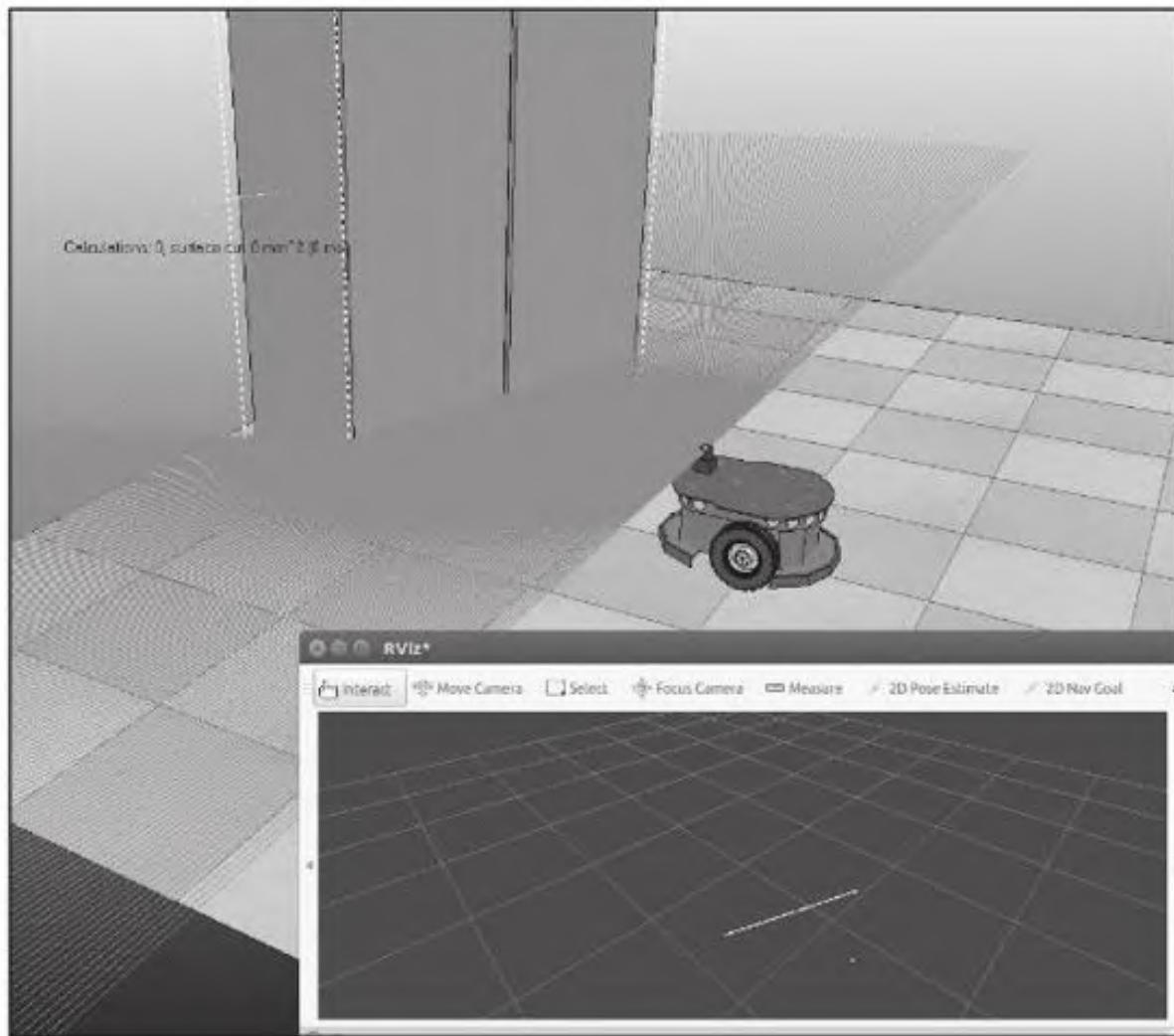


图5-16 在V-REP和RViz中查看的激光传感器数据

5.4.2 在V-REP中添加3D视觉传感器

在本节，我们将为移动机器人添加另一种传感器：RGB-D传感器，和第4章中使用过的传感器一样。V-REP已经预先创建了这种传感器的模型。与激光传感器不一样的是它没有直接与ROS集成。因此，我们要修改与此传感器关联的脚本，以便通过话题传输数据。从Model Browser面板上选择Kinect模型，然后将其拖放到机器人部件中。按照以下方式将Kinect放在所需的位置和相关脚本中：

```
if (sim_call_type==sim_childscriptcall_initialization) then
    depthCam=simGetObjectHandle('kinect_depth')
    depthView=simFloatingViewAdd(0.9,0.9,0.2,0.2,0)
    colorCam=simGetObjectHandle('kinect_rgb')
    colorView=simFloatingViewAdd(0.69,0.9,0.2,0.2,0)
    glass=simGetObjectHandle('kinect_glass')
    kinect=simGetObjectHandle('kinect')
end

if(sim_call_type==sim_childscriptcall_sensing) then
    rgbTopicName=simExtROS_enablePublisher('/rgb/image_raw',1,simros_strmcmd_get_vision_sensor_image,colorCam,0,'')
    DepthTopicName=simExtROS_enablePublisher('/depth/image_raw',1,simros_strmcmd_get_vision_sensor_image,depthCam,0,'')
    infoTopicName=simExtROS_enablePublisher('/cameraInfo',1,simros_strmcmd_get_vision_sensor_info,colorCam,0,'')

end

if (sim_call_type==sim_childscriptcall_cleanup) then
end
```

在sim_childscriptcall_sensing部分，我们添加了传输RGB-D传感器生成的数据代码。这些数据包含彩色图像、深度图像以及相机标

定的信息。我们可以使用下面的命令运行仿真：

```
$ rosrun image_view image_view image:=/rgb/image_raw
```

我们得到的图像如图5-17所示。

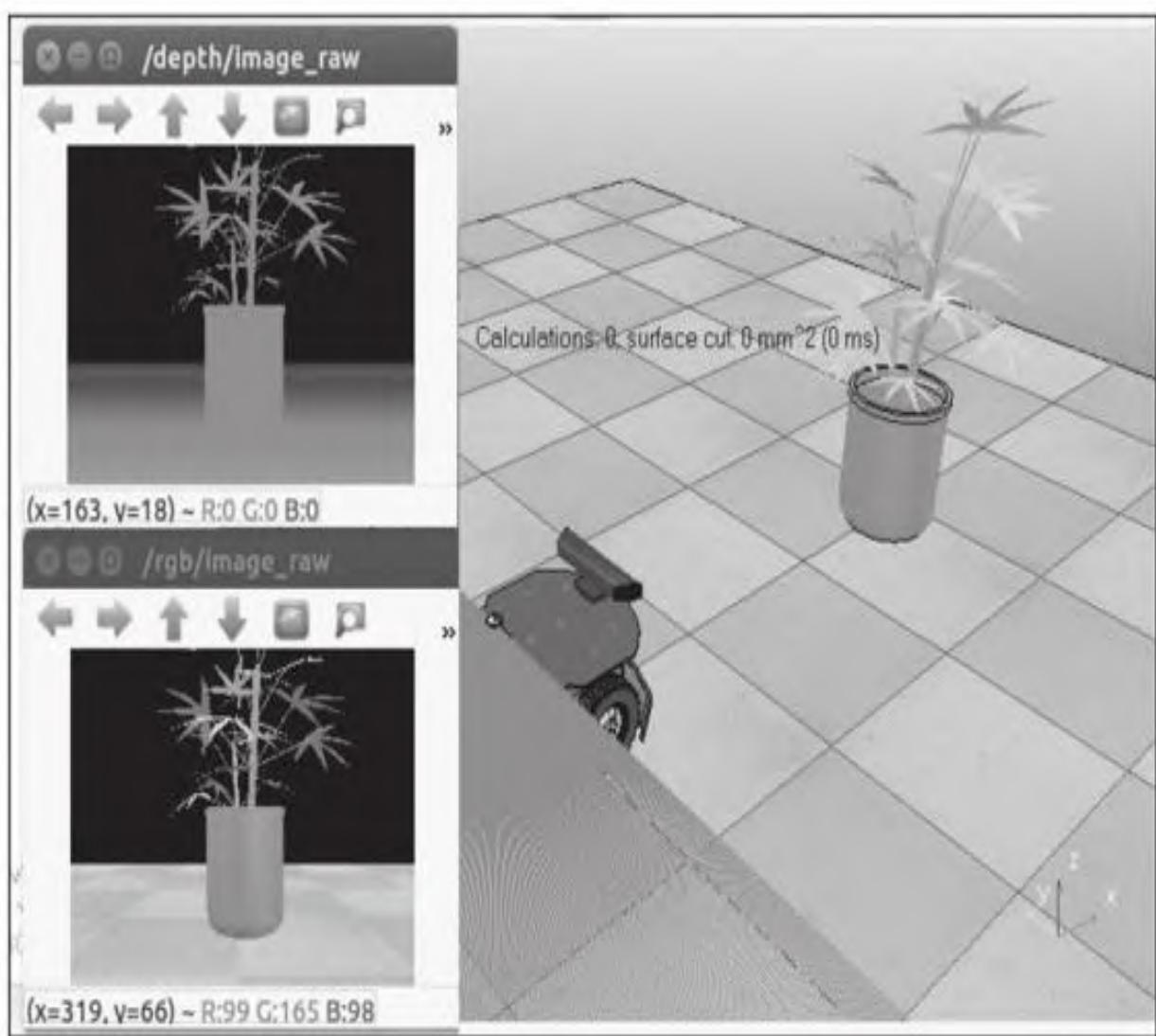


图5-17 在V-REP中查看由Kinect传感器生成的RGB和深度图像

现在我们仿真了一种差速轮式移动机器人，它配备了与ROS集成的各种传感器。此机器人模型保存在vrep_demo_pkg软件包内的

`mobile_robot.ttt`仿真场景中。

5.5 习题

现在我们应该能够回答以下问题：

- V-REP和ROS是如何进行通信的？
- 可以用何种方式让ROS控制V-REP仿真？
- 如何在V-REP中导入新的机器人模型并将它们与ROS进行集成？
- 如何将新的V-REP模型传感器与ROS进行集成？
- 如何在V-REP仿真控制机器人移动？

5.6 本章小结

在本章，我们主要使用另一种仿真软件V-REP重复了在前一章中用Gazebo进行的一些操作。V-REP是一个支持多平台的仿真软件，集成了不同的技术，功能多样。与Gazebo相比，V-REP表现得更容易让新手使用。

在本章，我们仿真了两类机器人，一类是由URDF文件导入的前一章中设计的7-DOF机械臂，另一类是由V-REP仿真模型提供的一个常见的差速轮式机器人。我们还学习了如何用ROS连接和控制我们模型的机器人关节，以及如何用话题控制差速移动机器人运动。此外，我们还讨论了在仿真场景中如何添加不同类型的传感器，通过添加激光雷达和3D视觉传感器改进仿真移动机器人的设备。最后，我们讨论了如何将V-REP中仿真的RGB-D传感器与ROS框架连接起来。

下一章，我们将学习如何将机械臂与ROS MoveIt!软件包连接起来，并将移动机器人与导航（navigation）软件包集连接起来。

第6章

ROS MoveIt!与导航软件包集

前面几章，我们一直在讨论机械臂和移动机器人的设计与仿真。我们在Gazebo中使用ROS控制器控制机械臂的每个关节，并在Gazebo中使用teleop节点控制机器人移动。

在本章，我们将讨论运动规划（motion planning）问题。通过人工直接控制关节让机器人运动可能非常困难，尤其当我们想要对机器人的运动增加位置约束和速度约束的时候。同样地，驱动移动机器人运动并规避障碍物时，也需要规划其运动路径。因此，我们使用ROS MoveIt!和导航软件包集（navigation stack）解决这些问题。

MoveIt!是一组在ROS中进行移动操作的软件包工具的集合。官方网站 (<http://moveit.ros.org/>) 包含了各种文档，采用MoveIt!的机器人列表，以及各种示例（用来演示拾取、放置和抓取，使用逆向运动学进行简单的运动规划）。

MoveIt!包含最先进的软件来进行运动规划、操作、3D感知、运动学、碰撞检测、控制和导航。除了命令行界面，MoveIt!拥有非常好的图形界面，用以将新的机器人连接到MoveIt!。此外，还有一个RViz插件用于在RViz内进行运动规划。我们还将看到如何使用MoveIt!的C++ API对我们的机器人进行运动规划。

接下来是导航软件包集，这是一组强大的工具和软件库，主要用于移动机器人导航。导航软件包集包含可以立即使用的导航算法，尤其是用于差速轮式机器人的。使用这些软件包我们可以让机器人自主控制，这是我们在导航软件包集中看到的最后一个概念。

在本章的第一部分，我们将进一步讨论MoveIt!软件包、安装和架构。讨论了MoveIt!的主要概念之后，我们将学习如何为我们的机械臂创建一个MoveIt!软件包，该软件包将为我们的机器人提供能感知碰撞的路径规划。使用该软件包，我们可以在RViz内进行运动规划（逆向运动学），还可以与Gazebo或真实机器人连接来执行规划的路径。

讨论了接口之后，我们将进一步讨论导航软件包集，并了解如何使用即时定位与地图构建（Simultaneous Localization And Mapping, SLAM）和自适应蒙特卡罗定位（Adaptive Monte Carlo Localization, AMCL）执行自主导航。

6.1 安装MoveIt!

下面让我们来安装MoveIt!。安装过程非常简单，只有一个命令。使用如下命令，我们就可以安装MoveIt!核心库，这是一套针对ROS Kinetic的插件和规划器的集合：

```
$ sudo apt-get install ros-kinetic-moveit ros-kinetic-moveit-plugins ros-kinetic-moveit-planners
```

MoveIt!架构

让我们从MoveIt!及其架构开始。理解MoveIt!的架构有助于我们编程，并有助于将机器人与MoveIt!进行交互。我们将快速地看一看MoveIt!的架构和一些重要的概念，然后开始对机器人进行编程，并与MoveIt!交互。

MoveIt!的架构如6-1所示，官方网址为
<http://moveit.ros.org/documentation/concepts>。

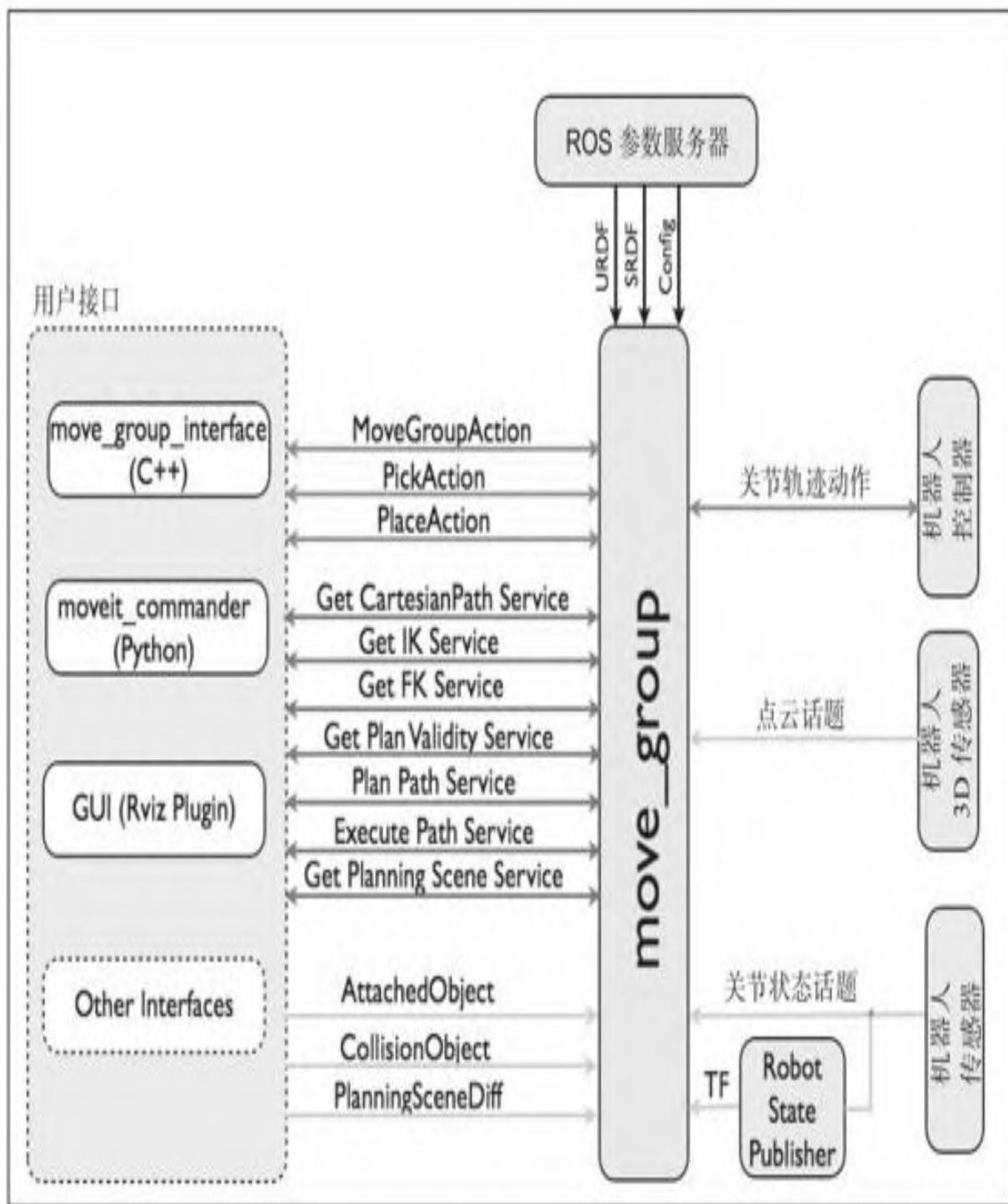


图6-1 MoveIt!架构图

move_group节点

我们可以认为move_group就是MoveIt!的核心。因为该节点作为机器人的各个部件的集合，并且根据用户的需求提供动作和服务。

在架构图中可以清楚地看出，move_group节点用话题和服务的形式搜集了机器人的信息，例如点云、关节状态、坐标变换（tf）。

MoveIt!从参数服务器收集了机器人的运动学数据，例如robot_description(URDF)、语义机器人描述格式（Semantic Robot Description Format, SRDF）及配置文件。在我们为机器人生生成MoveIt!软件包的同时也生成了SRDF文件和配置文件。配置文件包含设置关节约束、感知、运动学、末端执行器等方面的参数文件。当我们讨论为机器人生生成MoveIt!软件包时将看到这些文件。

当MoveIt!可以获取关于机器人及其配置的所有信息时，我们可以说它已经配置好了，我们就可以从用户界面开始控制机器人了。我们可以使用MoveIt!的C++或者Python API接口向move_group节点发指令以执行一些动作，如拾取/放置、逆向运动学（IK）和正向运动学（FK）。在使用RViz运动规划插件时，我们可以在RViz图形界面向机器人发指令。

正如我们讨论过的，move_group节点是一个集合。它没有直接运行任何类型的运动规划算法，而是将所有功能作为插件连接起来。有运动学求解器，运动规划等插件，我们可以通过这些插件来扩展功能。

在运动规划之后，生成的运动轨迹可以使用FollowJointTrajectoryAction接口与机器人的控制器进行交互。这是一个动作接口，通过该接口动作服务器在机器人上运行，move_node节点负责初始化一个动作客户端，该动作客户端与该服务器通信并在真实机器人或Gazebo仿真器中执行这些规划的轨迹。

在讨论MoveIt!的最后，我们将学习如何将MoveIt!和RViz图形界面连接到Gazebo。图6-2所示的截图显示了一个从RViz控制的机械臂，其运动轨迹在Gazebo中执行。

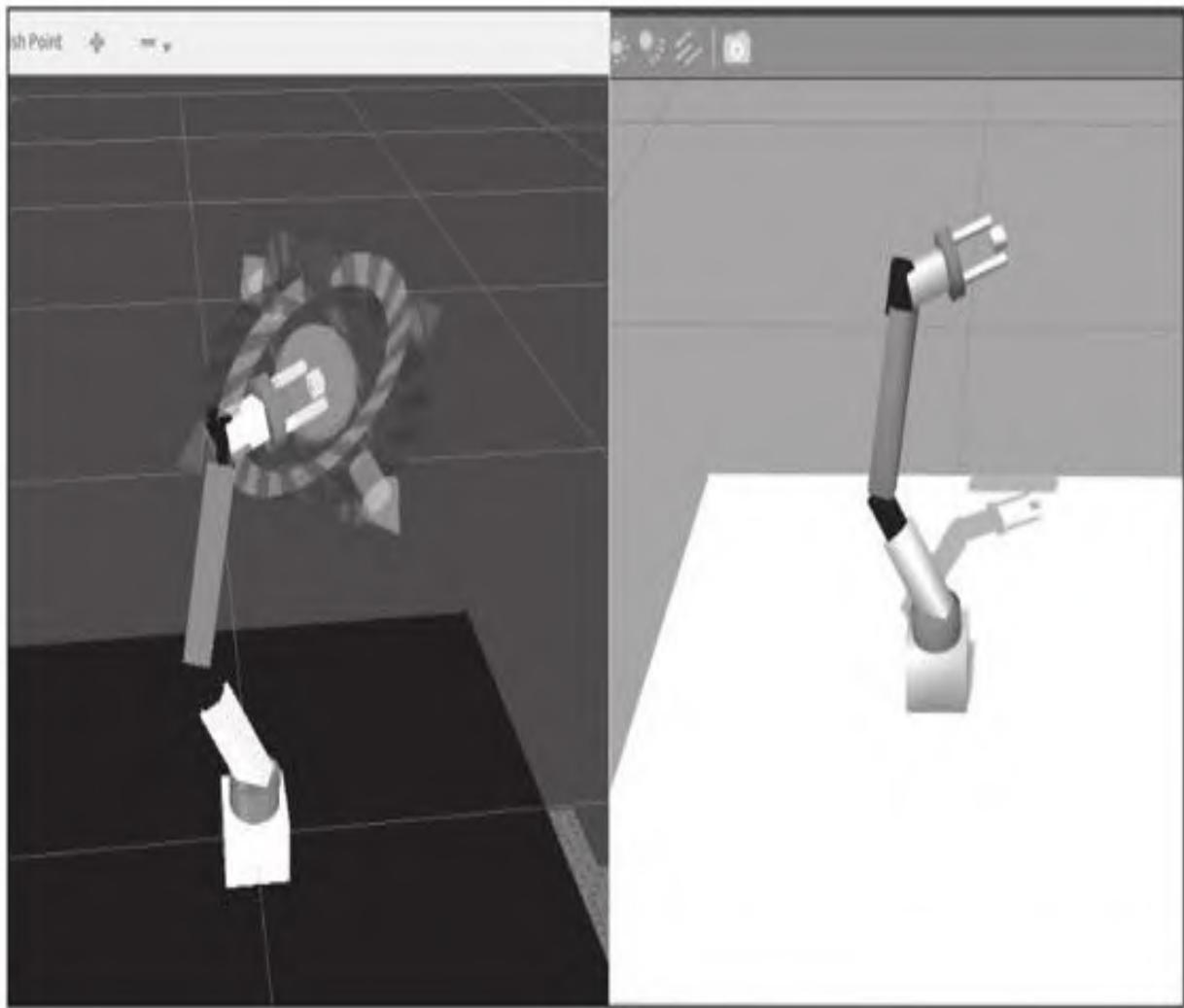


图6-2 在Gazebo中执行来自RViz GUI的轨迹

用MoveIt!进行运动规划

假如我们知道了机器人的初始姿态，机器人的期望目标姿态，机器人的几何描述以及周围环境的几何描述，那么路径规划就是寻找将机器人逐步从初始姿态移动到目标姿态的最佳路径，在运动过程中不能碰到障碍物，也不能与自身的机器人连杆碰撞。

在这里，机器人的外观几何形状由URDF文件描述。我们还可以为机器人环境创建描述文件，并用机器人上的激光或视觉传感器为其操作空间构建地图，避免在执行规划的路径时与出现的静态或动态的障碍物发生碰撞。

在机械臂的例子中，运动规划器需要找出一个可以让机器人的所有连杆都不与环境发生碰撞的轨迹（由每个关节的关节空间组成），也要避免自碰撞（机器连杆之间的碰撞），还要避免超过关节的极限。

MoveIt!可以通过插件接口与运动规划器进行交互。我们只需通过更改插件即可使用任意的运动规划器。此方法具有高度可扩展性，因此我们可以使用此接口来尝试自定义的运动规划器。`move_group`节点通过ROS动作和服务与运动规划器插件进行交互。`move_group`节点默认的规划器是OMPL (<http://ompl.kavrakilab.org/>)。

为了启动运动规划，我们需要向运动规划器发送运动规划请求，以指定我们具体的规划要求。规划需求可以是设定末端执行器的目标姿态，例如用于拾取和放置的操作。

我们可以为运动规划器设置附加运动学约束。下面是MoveIt!中的一些内置约束：

- 位置约束：这些约束限制连杆的位置。
- 方向约束：这些约束限制连杆的方向。
- 可见性约束：这些约束限制连杆上的点在区域中的可见性（传感器的视野）。
- 关节约束：这些约束限制关节的限值。
- 用户指定的约束：利用这些约束，用户可以使用回调函数定义自己的约束。

利用这些约束，我们可以发送运动规划请求，规划器将根据请求生成适当的轨迹。`move_group`节点将从运动规划器中生成适当的轨迹，该轨迹将遵循所有约束。最终的规划路径将发送给机器人的关节轨迹控制器。

运动规划请求适配器

规划请求适配器（Planning Request Adapter）有助于预处理运动规划请求并对运动规划响应进行后续处理。预处理请求的一个用途是它能帮助纠正关节状态的非法情况，另外在后续处理时，它还可以将规划器生成的路径转换为以时间为参数的轨迹。下面是MoveIt!中一些默认的规划请求适配器：

- FixStartStateBounds：如果关节状态轻微超出了关节的限制，则此适配器可以将关节限值固定在限制范围内。
- FixWorkspaceBounds：该适配器指定了用于规划多维数据集大小的工作空间为10m*10m*10m。
- FixStartStateCollision：如果现有的关节位姿（configuration）处于碰撞状态，该适配器将采样生成新的无碰撞位姿。它通过一个名为jiggle_factor的因子更改位姿来进行新配置。
- FixStartStatePathConstraints：该适配器将在机器人的初始姿态不符合路径约束时使用。这里，它将寻找满足路径约束的近似姿态并将此姿态当作初始状态。
- AddTimeParameterization：该适配器通过添加速度和加速度约束来参数化运动规划。

MoveIt! 规划场景

术语“规划场景”是用来表示机器人周围的世界并存储机器人本身的状态。在move_group内的规划场景监视器维护着规划场景的表示。move_group节点由另一个称作世界几何结构监测器的部分组成，该部分通过机器人的传感器和用户的输入构建世界的几何结构。

规划场景监视器从机器人读取joint_states话题，从世界几何结构监视器读取传感器信息和世界几何结构。世界场景监视器从占用地图监视器中读取数据，该监视器使用3D感知构建周围环境的3D表示，称为Octomap。Octomap可以由点云生成，这些点云由点云占用地图更新插件来处理，深度图像由深度图像占用地图更新器处理。图6-3显示了MoveIt!官方维基中规划场景的表示

(<http://moveit.ros.org/documentation/concepts/>)。

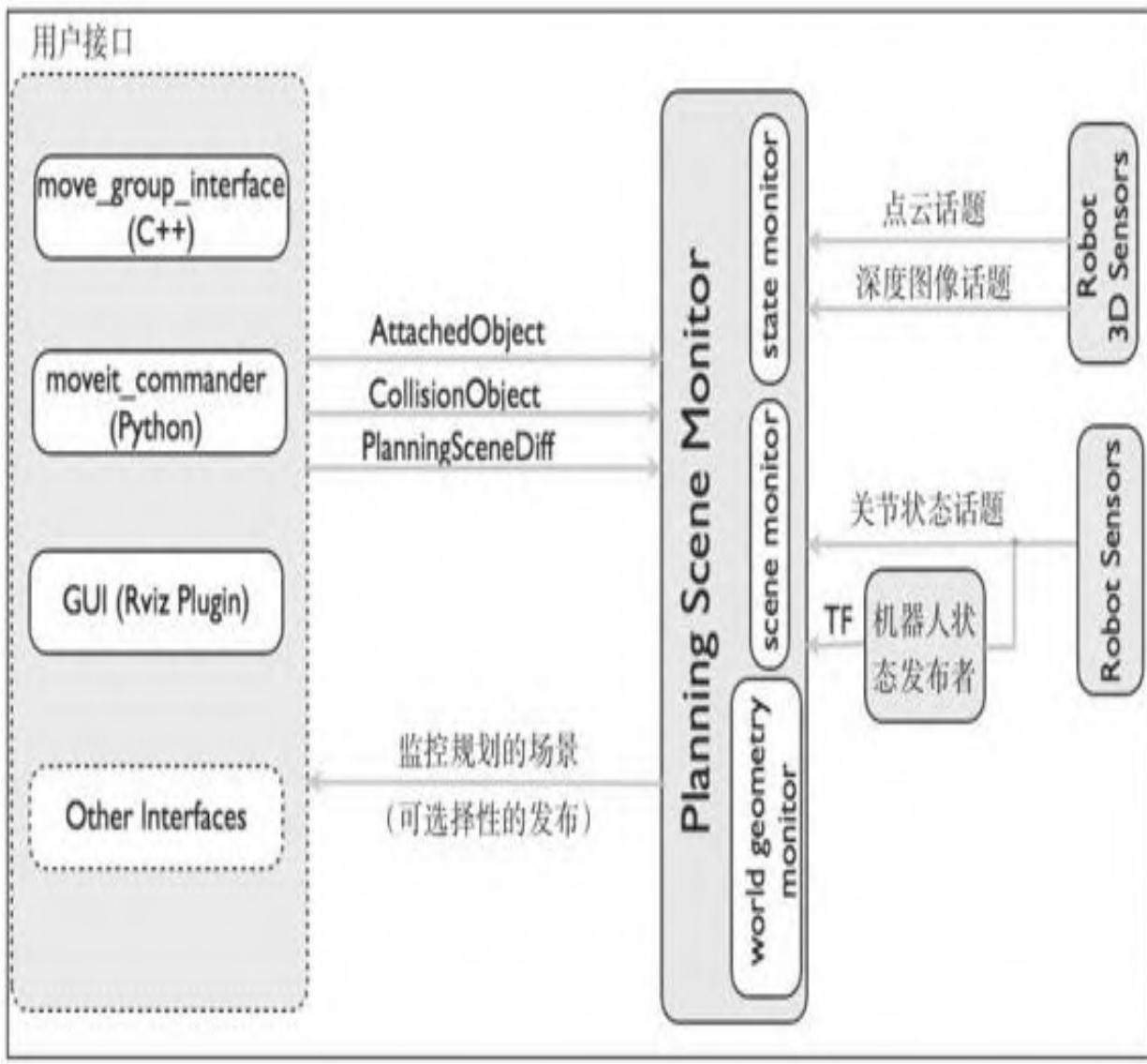


图6-3 MoveIt!规划场景概览图

MoveIt!处理运动学

MoveIt!使用机器人插件为切换逆向运动学算法提供了极大的灵活性。用户可以将自己的IK解算器编写为MoveIt!的插件，并在需要时替换默认的解算器插件。MoveIt!中的默认IK解算器是基于雅可比行列式的数值解算器。

与解析解算器相比，数值解算器可能需要更多时间来处理IK。这个名为IKFast的软件包可以用来生成用于使用解析方法来求解IK的C++代码，该方法可用于不同类型的机器人执行器，且DOF小于6的时候

表现更好。此C++代码可以用一些ROS工具转换为MoveIt!插件。我们将在接下来的章节中介绍这个过程。

正向运动学和计算雅克比矩阵已经集成到了MoveIt! RobotState类中，因此我们不需要使用插件来解决FK问题。

MoveIt!碰撞检测

MoveIt!中的CollisionWorld对象用来查找规划场景内的碰撞。该场景使用Flexible Collision Library (FCL) 软件包作为后端。

MoveIt!支持多种不同类型对象的碰撞检测，比如网格物体，基本形状如立方体、圆柱、圆锥、球体和Octomap。

碰撞检测是运动规划期间计算成本高昂的任务之一。为了减少计算资源的消耗，MoveIt!提供了允许碰撞的矩阵（Allowed Collision Matrix, ACM）。它包含一个二进制数值，确定是否要检测两个物体之间的碰撞。如果矩阵中的数值是1，意味着无须为相应的两个物体检测碰撞。我们可以将离得较远彼此间不可能发生碰撞的两个物体的该值设置为1。这样优化ACM可以减少避免碰撞规避所需的总计算量。

讨论了MoveIt!的基本概念之后，现在我们可以讨论如何将一个机械臂连接到MoveIt!。为了在MoveIt!中连接机械臂，我们需要满足图6-1中看到的组件。move_group节点主要需要一些参数，例如URDF，SRDF，配置文件和关节状态话题，以及来自机器人进行运动规划的tf。

MoveIt!提供了一个叫作配置助手（Setup Assistant）的图形界面工具来生成所有的这些元素。接下来的部分描述了使用配置助工具配置的过程。

6.2 使用配置助手工具生成MoveIt!配置软件包

MoveIt!配置助手是一个图形化的用户工具，用来将任何机器人安装配置到MoveIt!中。简单地说，这个工具可以生成SRDF配置文件，启动文件和从机器人URDF模型生成的脚本，这是配置move_group节点所必需的。

SRDF文件详细描述了有关机械臂关节、末端执行器关节、虚拟关节和碰撞连杆对的详细信息，这些都是可以用配置助手工具进行MoveIt!配置过程中的配置。

配置文件包含了运动学解算器、关节约束、控制器等方面的详细信息。这些信息也在配置过程中进行配置和保存。

使用生成的机器人配置软件包，我们可以在没有真实机器人和仿真界面的情况下在RViz中进行运动规划。

下面启动配置界面，我们就可以看到构建机械臂配置软件包的分步过程。

6.2.1 第1步：启动配置助手工具

我们可以用下面的命令来启动MoveIt!配置助手工具：

```
$ rosrun moveit_setup_assistant setup_assistant.launch
```

这将打开一个窗口并提供两个选项：Create New MoveIt! Configuration Package或者Edit Existing MoveIt! Configuration Package。这里我们正在创建一个新配置软件包，所以我们选择第一个选项。如果我们已经有一个MoveIt!配置软件包了，那么我们就可以选择第二项。

再点击Create New MoveIt! Configuration Package按钮，将出现一个新的窗口，如图6-4所示。

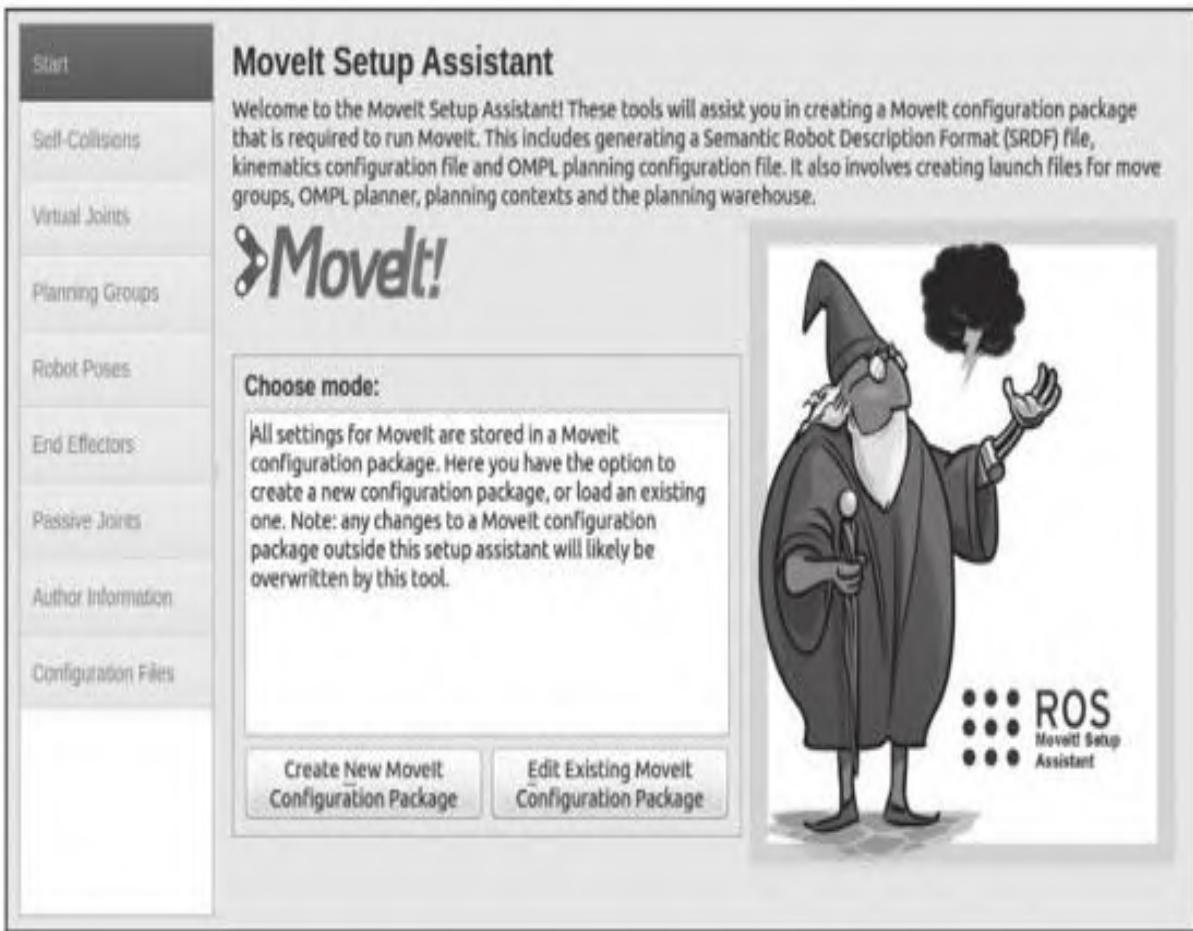


图6-4 MoveIt!配置助手

在此步骤中，该助手会提示需要提供新机器人的URDF模型。这里我们点击Browse按钮，选择该路径下的mastering_ros_robot_description_pkg/urdf/seven_dof_arm.urdf文件。选择该文件后，点击Load按钮来加载URDF。我们可以将机器人模型作为纯URDF或者xacro文件，如果我们提供xacro模型，那么该工具将在内部转换为URDF模型。

如果机器人模型解析成功，我们将在窗口中看到机器人的模型，如图6-5所示。



图6-5 配置助手工具成功解析机器人模型

6.2.2 第2步：生成自碰撞矩阵

现在我们可以查看该窗口中的所有面板来正确配置我们的机器人。在Self-Collisions选项中，MoveIt!查询机器人中可以安全禁用碰撞检测的连杆对。这样可以减少处理时间。此工具对每个连杆对进行分析，并将每对连杆对分类为始终处于碰撞状态（Always in Collision）、永远不会碰撞（Never in Collision）、默认处于碰撞状态（Default in Collision）、禁用相邻连杆（Adjacent Links Disabled）、有时处于碰撞状态（Sometimes in Collision）。该工具将对发生碰撞的连杆禁用碰撞检测。图6-6展示了Self-Collisions窗口。

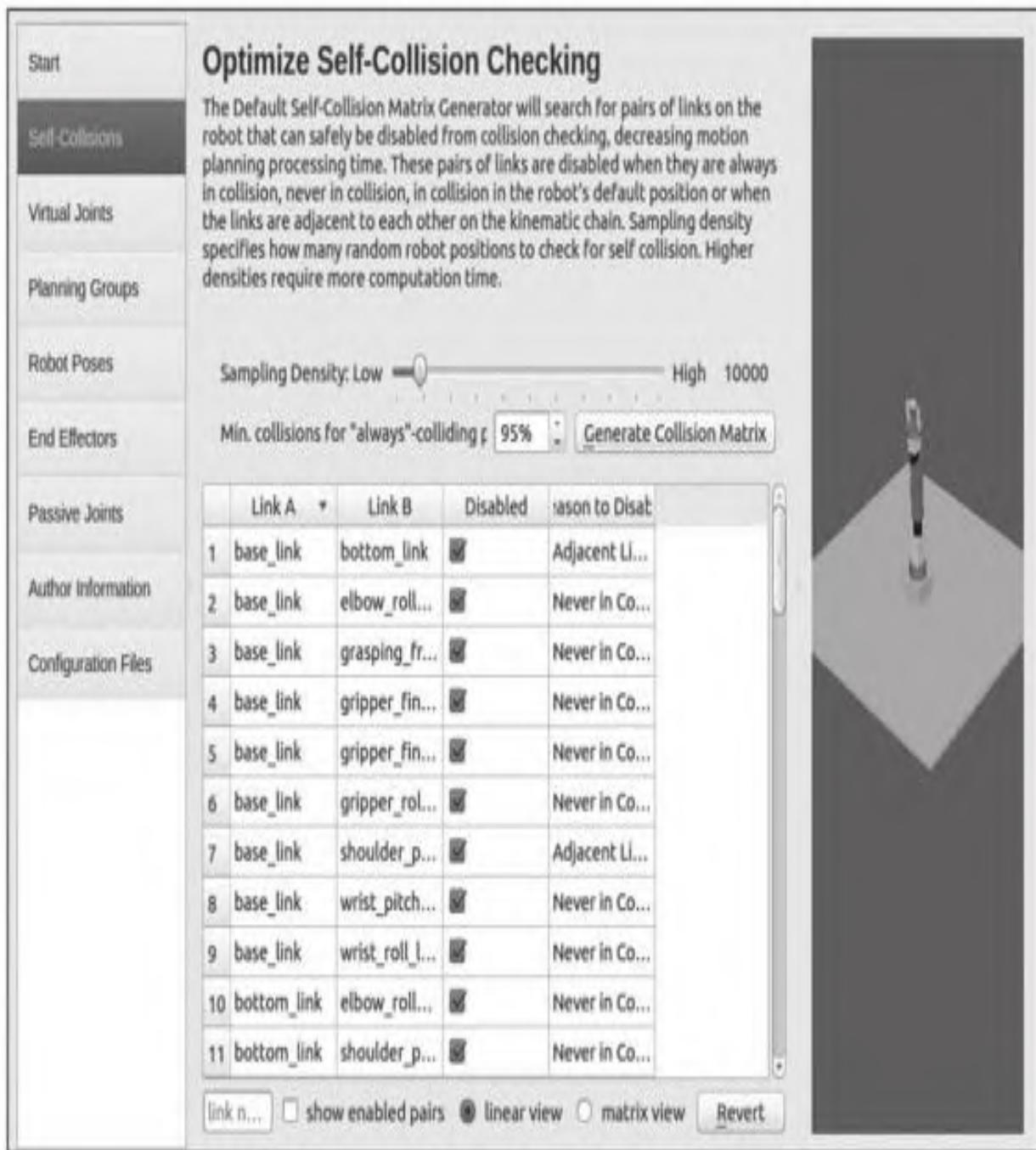


图6-6 重新生成自碰撞 (Self-Collisions) 矩阵

采样密度是检查自碰撞的随机位置的数量。如果采样密度高，则计算量较大，但自碰撞机率会较小。采样密度的默认值为10 000。我们可以通过点击Regenerate Default Collision Matrix按钮看到禁用的连杆对。列出禁用的连杆对需要几秒钟时间。

6.2.3 第3步：增加虚拟关节

虚拟关节（Virtual Joints）将机器人与世界连接起来。对非移动的静态机器人，虚拟关节不是必须设置的。当机械臂的底座非固定时，我们需要虚拟关节。当一个机械臂固定在移动机器人上时，我们就需要定义一个相对于里程计（odom）的虚拟关节。

对于我们的机器人，我们不需要创建虚拟关节。

6.2.4 第4步：添加规划组

规划组（Planning Group）基本上就是机械臂中的一组关节/连杆。该规划组一起进行规划以实现到连杆或末端执行器的目标位置。我们需要创建两个规划组，一个用于机械臂，另一个用于夹爪。

点击左侧的Planning Groups选项卡，然后再点击Add Group按钮。你将看到如图6-7的界面，其中包含了机械臂组的各项设置：

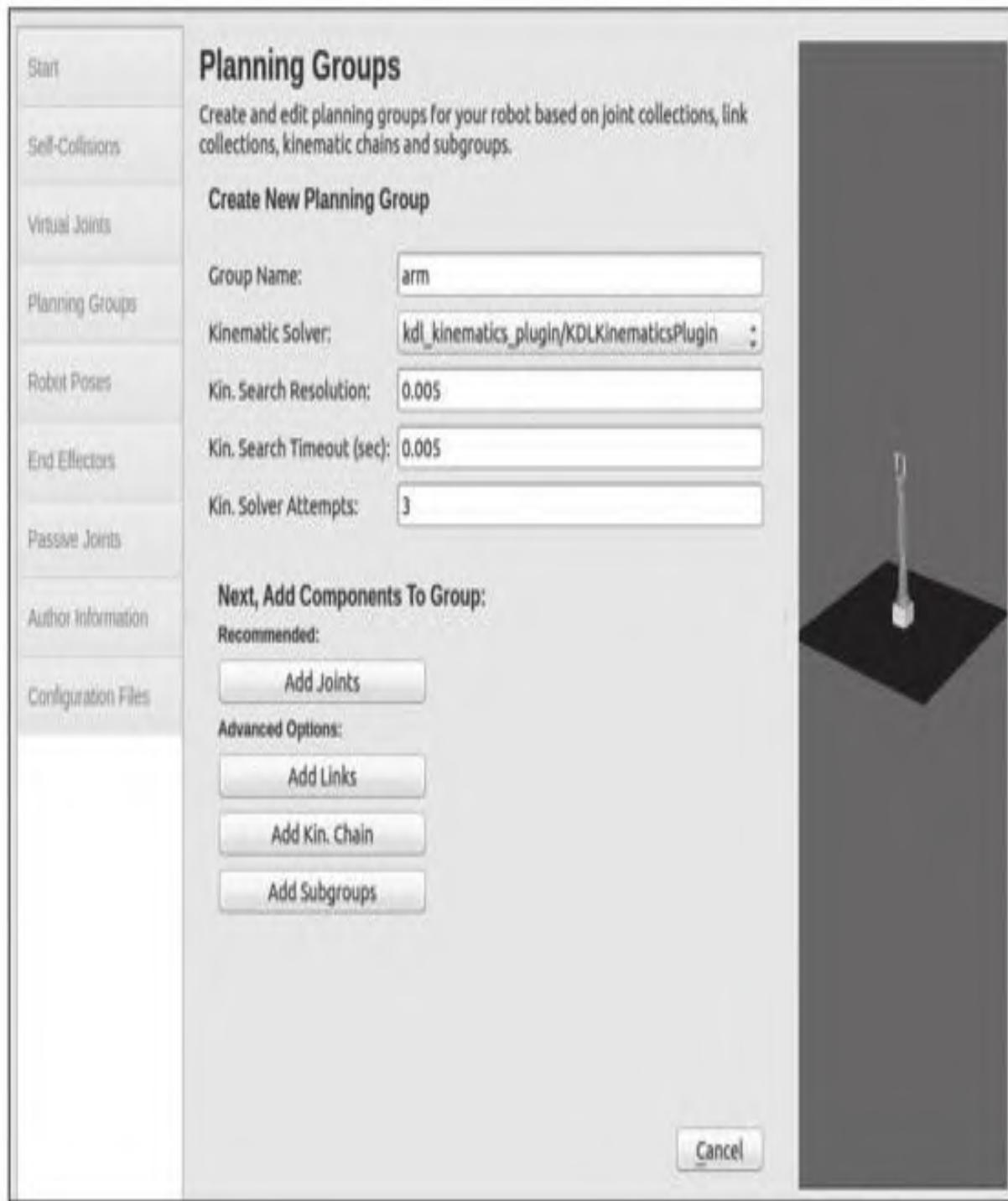


图6-7 添加机械臂规划组

在这里，我们将Group Name设置为arm，Kinematic Solver设置为kdl_kinematics_plugin/KDLKinematicsPlugin，这是MoveIt!默认的数值IK解算器。我们可以将其他参数设置为默认值。此外，我们还可

以在规划组中用不同的方法添加元素。例如，我们可以指定组的关节，添加其连杆，或直接指定运动链。

在arm组内，首先我们需要添加一个运动链，从第一个连杆base_link开始至连杆grasping_frame。

添加一个名为gripper的组，我们不需要再为gripper组提供运动解算器。在该组内，我们可以增加gripper的关节和连杆。这些设置如图6-8所示。

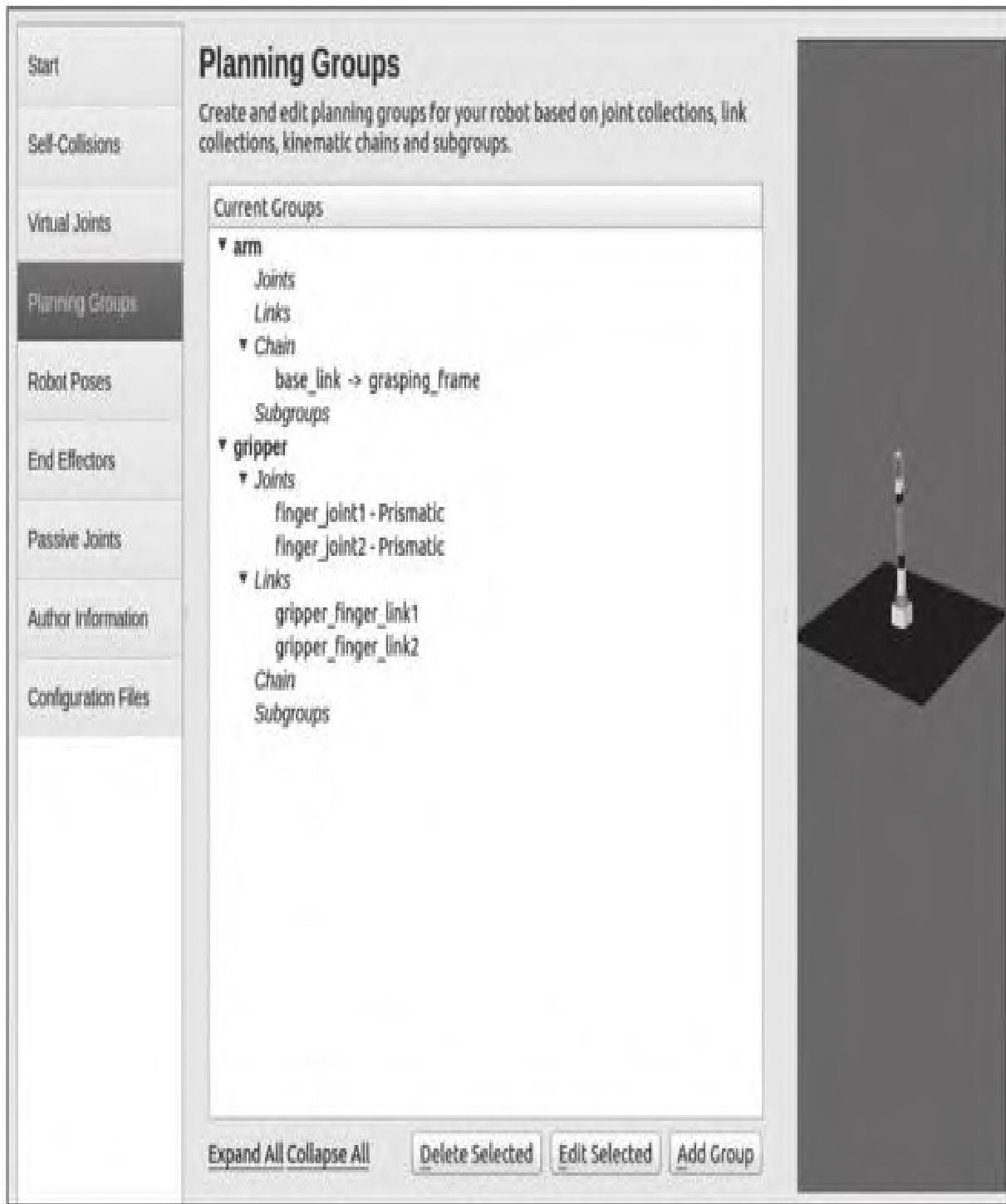


图6-8 添加arm和gripper规划组

6.2.5 第5步：添加机器人姿态

在此步骤中，我们可以在机器人配置中添加一些固定的姿态。例如，我们可以在此步骤中指定起始位置或者拾取/放置的位置。优点是我们在用MoveIt! API编程时可以直接调用这些姿态，这些姿态也叫作组状态，它们在拾取/放置和抓取操作中有很多的应用。机器人可以毫无困难地切换到固定姿态。

6.2.6 第6步：设置机器人的末端执行器

在此步骤中，我们需要给机器人的末端执行器命名并分配末端执行器组、父连杆和父级组。

我们可以给这个机器人添加任意数量的末端执行器。在我们的例子中，它是一个专门用来执行拾取/放置操作的夹爪。

点击Add End Effector按钮并将末端执行器命名为robot_eef，将规划组设置为grripper（前面我们已创建过），父连杆设置为grasping_frame，父级组设置为arm，如图6-9所示。

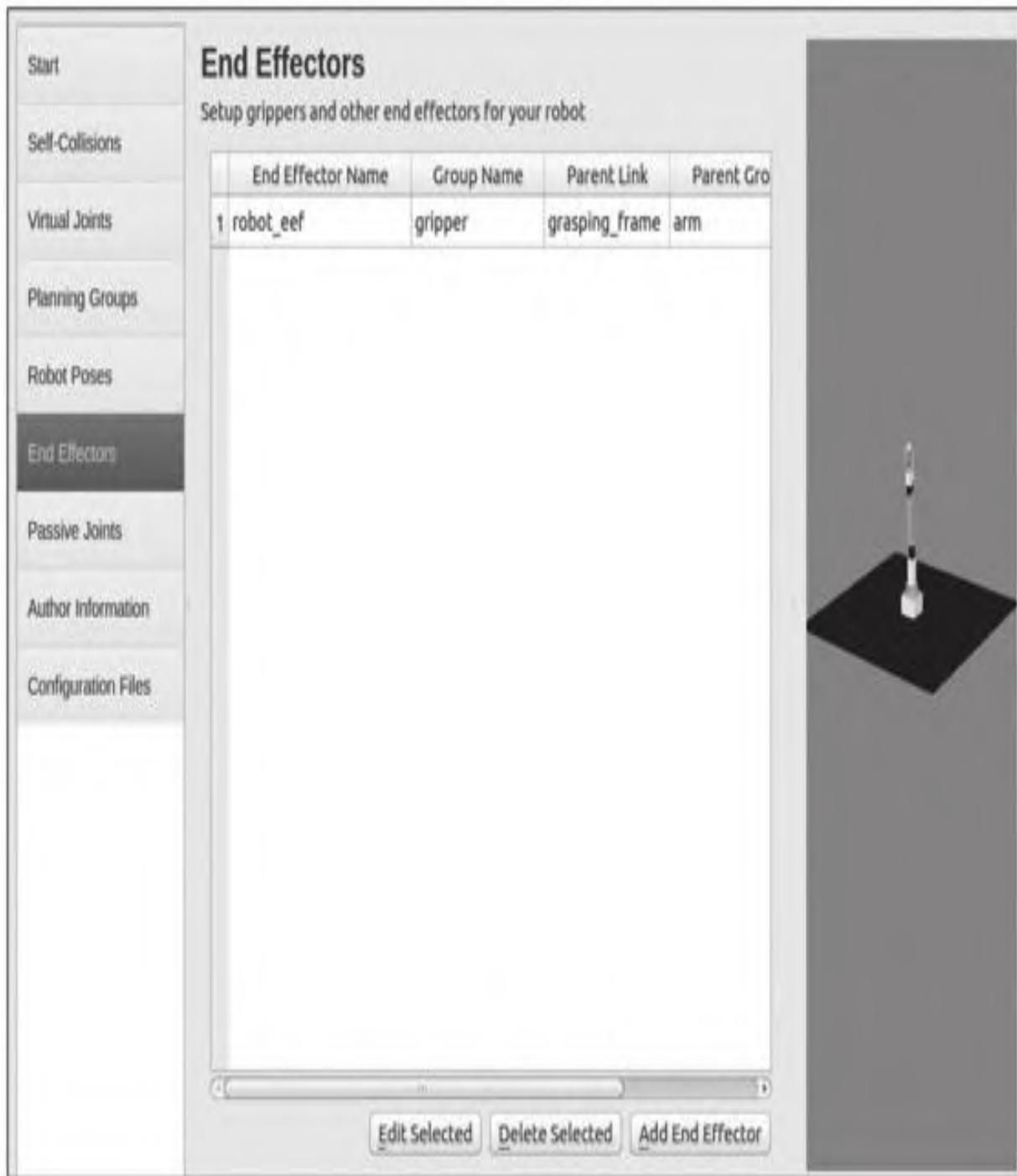


图 6-9 添加末端执行器

6.2.7 第7步：添加被动关节

在此步骤中，我们可以指定机器人中的被动关节。被动关节是指关节没有任何执行机构。脚轮是一个被动关节的例子。规划器将在运动规划时忽略这些类型的关节。

6.2.8 第8步：作者信息

在此步骤中，机器人模型的作者可以增加catkin所需的个人信息，包括作者的名字和邮件地址，以便将该模型发布到ROS社区。

6.2.9 第9步：生成配置文件

我们基本上算是完成了。现在我们到了最后一步，即生成配置文件。在此步骤中，该工具将生成包含与MoveIt!接口交互所需文件的配置软件包。

点击Browse按钮选择用来保存由配置助手工具生成的配置文件的文件夹。在这里我们可以看到文件在一个名为seven_dof_arm_config的文件夹中创建。你可以将配置软件包命名为机器人名称加_config或_generated这样的后缀。

点击Generate Package按钮后，它将在指定的文件夹中生成配置文件。

如果该过程成功，我们可以点击Exit Setup Assistant退出配置助手工具。

图6-10所示的截图展示了生成配置文件的过程。

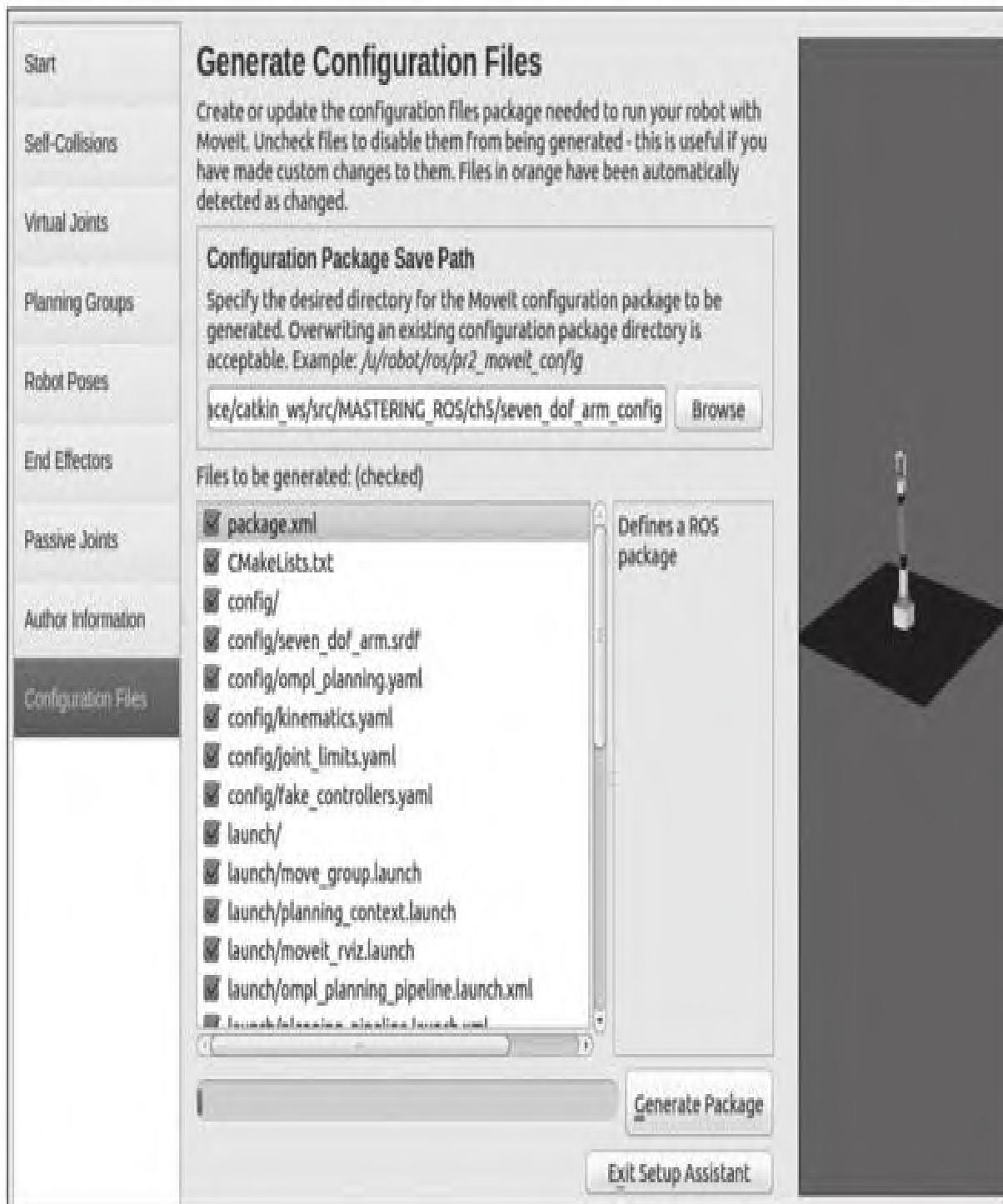


图6-10 生成MoveIt!配置软件包

生成MoveIt!配置软件包后，我们可以将它复制到我们的catkin工作区。在下一节，我们将使用此软件包。像往常一样，创建的机器人

模型可以从以下GitHub代码仓库下载或从本书附带的源代码中获取：

```
$ git clone https://github.com/jocacaco/seven_dof_arm_config
```

6.3 使用MoveIt!配置软件包在RViz中进行机器人运动规划

MoveIt!为RViz提供了一个插件，可以建立新的规划场景（在该场景中，机器人运作、生成运动规划、添加新物体），显示规划的输出结果，还可直接与可视化的机器人进行交互。

MoveIt!配置软件包由配置文件和启动文件组成，用于在RViz中启动运动规划。在软件包中有一个演示的启动文件，用于进一步了解该软件包中的所有功能。

下面是运行演示启动文件的命令：

```
$ rosrun seven_dof_arm_config demo.launch
```

如果一切正常，我们将看到RViz加载了MoveIt!提供的MotionPlanning插件，如图6-11所示。

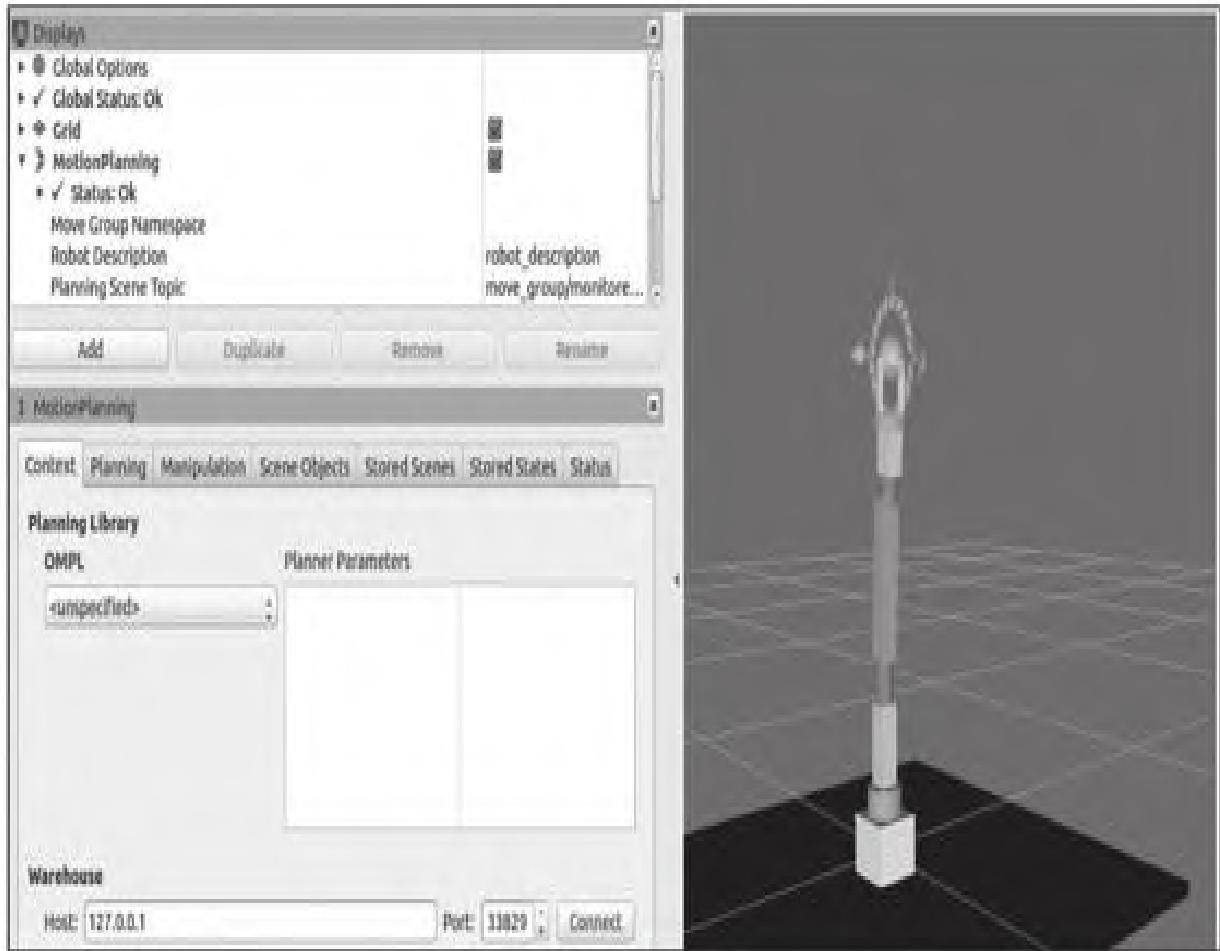


图6-11 MoveIt!-RViz插件

6.3.1 使用RViz运动规划插件

图6-11中，在屏幕的左边，我们可以看到RViz运动规划插件被加载了。在Motion Planning窗口中有多个选项卡，例如Context、Planning等。默认的选项卡是Context。我们看到默认的Planning Library为OMPL。这表明MoveIt!成功地加载了运动规划库。如果未加载的话，我们则无法执行运动规划。

图6-12是Planning选项卡，这是一个经常使用的选项卡。在这个选项卡中可以设定起始状态（Start State）、目标状态（Goal State）、路径规划（Plan）和执行（Execute）规划的路径。图6-12为Planning选项卡的GUI。

我们可以在Query面板下指定机械臂的开始状态和目标状态。点击Plan按钮后，我们就可以规划从起始状态到目标状态的路径，如果规划成功的话，我们就可以执行规划的路径。一般默认情况下，执行都是在伪控制器上完成的。我们可以将这些控制器更改为轨迹控制器，以便在Gazebo或真实机器人中执行规划的轨迹。

我们可以使用机械臂夹爪上的交互式标记来设置机器人末端执行器的起始位置和目标位置。我们可以拖动和旋转标记的姿态，并且如果规划有解，我们将看到一个橙色的机械臂。在某些情况下，即使末端执行器的标记姿态移动了，机械臂也不移动。如果机械臂没有到达标记的位置，我们就可以假设该姿态中没有IK解。我们可能需要更多的自由度来到达那里或者连杆之间可能存在一些碰撞。

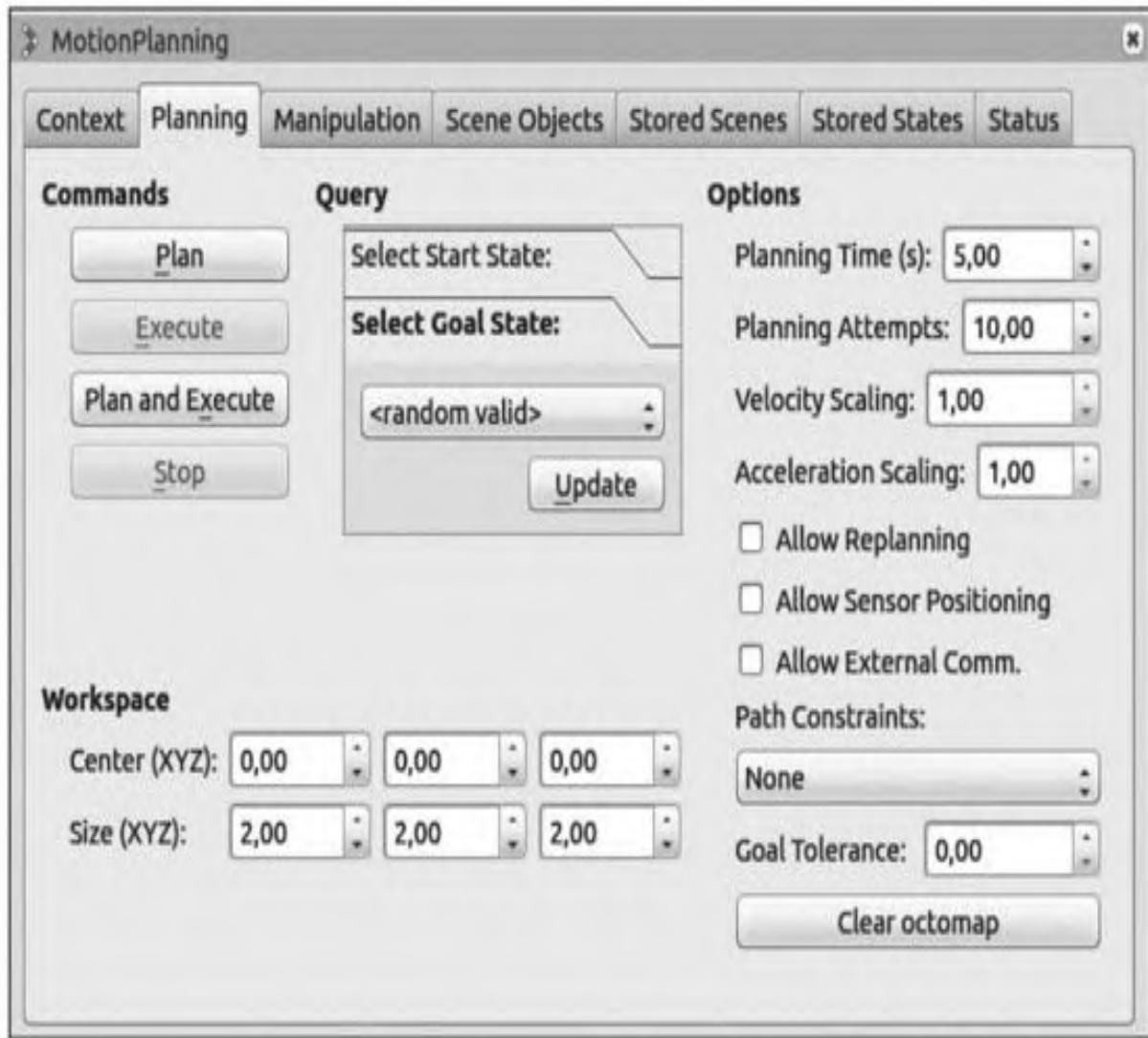


图6-12 MoveIt!-RViz的Planning选项卡

图6-13展示了有效的目标姿态和无效的目标姿态。

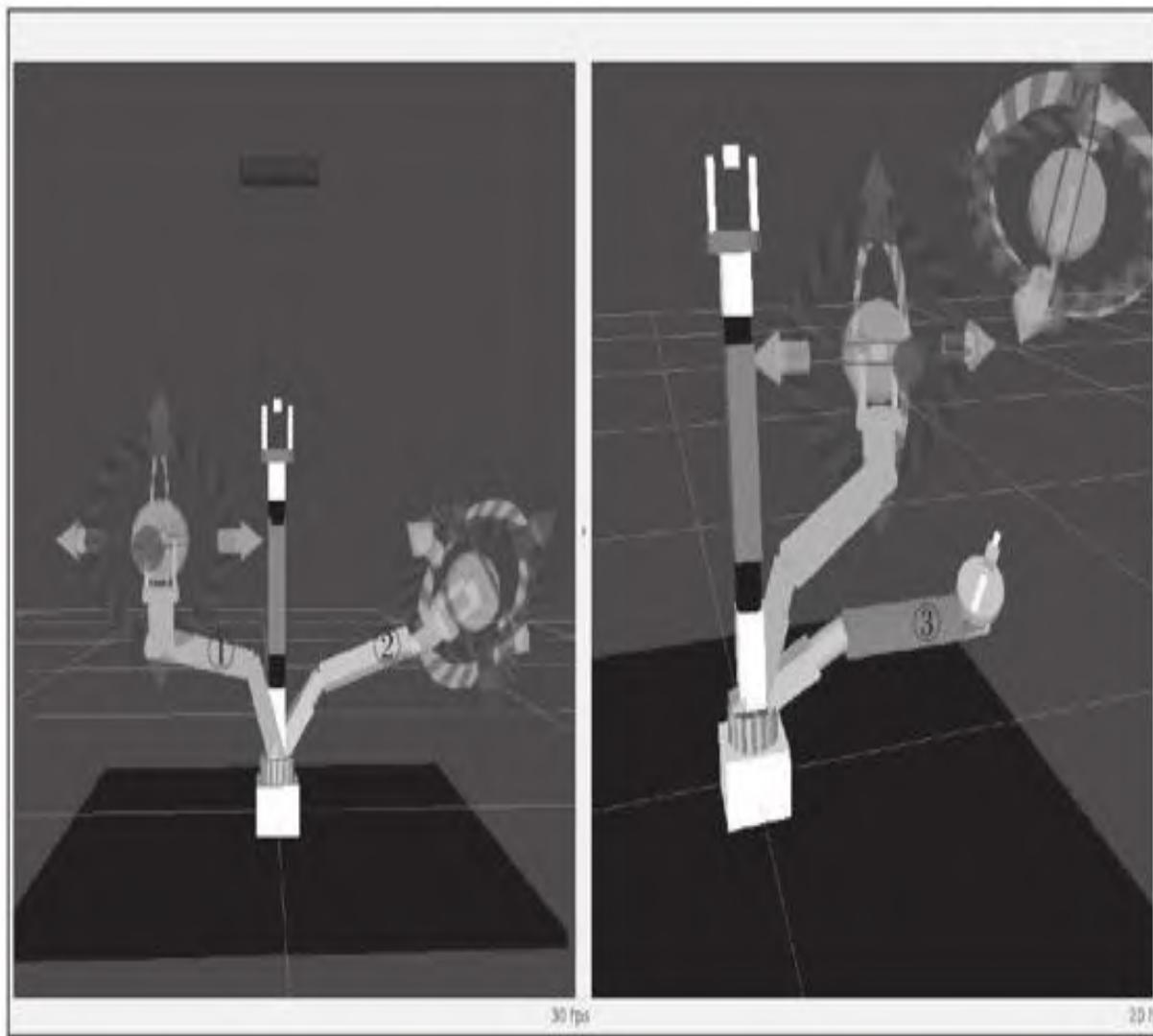


图6-13 RViz中机器人的有效姿态和无效姿态

绿色机械臂表示起始位置（见图6-13中①所示），橙色机械臂表示目标位置（见图6-13中②所示）。在图6-13的左图中，如果我们按下Plan按钮，MoveIt!将会从起始位置到目标位置规划出一条路径。在图6-13的右图中，我们可以观察到两件事。首先，橙色机械臂的一段关节是红色的（见图6-13中③所示），这表示目标姿态处于自碰撞的状态。其次，观察末端执行器标记，它与真实的末端执行器距离很远并且也变红了。

我们也可以在起始状态和目标状态中使用random valid（随机有效）选项进行快速的运动规划。如果我们选择目标状态为random

`valid`（即随机有效状态），并按下Update按钮，那么将会生成随机有效的目标姿态。点击Plan按钮后，我们可以观察到运动规划过程。

我们可以使用MotionPlanning插件中的多个选项自定义RViz的显示效果。图6-14就是这个插件的选项。

<input type="checkbox"/> Interact	<input type="checkbox"/> Move Camera	<input type="checkbox"/> Select	<input type="checkbox"/>	= ▾
Displays				
► ⚙ Grid	<input checked="" type="checkbox"/>			
▼ ➔ MotionPlanning	<input checked="" type="checkbox"/>			
► ✓ Status: Ok				
Move Group Namespace				
Robot Description				robot_description
Planning Scene Topic				planning_scene
▼ Scene Geometry				
Scene Name				(noname)+
Show Scene Geometry	<input checked="" type="checkbox"/>			
Scene Alpha				0.9
Scene Color				<input type="checkbox"/> 50; 230; 50
Voxel Rendering				Occupied Voxels
Voxel Coloring				Z-Axis
Scene Display Time				0.2
▼ Scene Robot				1
Show Robot Visual	<input checked="" type="checkbox"/>			
Show Robot Collision	<input type="checkbox"/>			
Robot Alpha				0.5
Attached Body Color				<input type="checkbox"/> 150; 50; 150
► Links				
► Planning Request				2
► Planning Metrics				
▼ Planned Path				
Trajectory Topic				/move_group/display_planned_path
Show Robot Visual	<input checked="" type="checkbox"/>			
Show Robot Collision	<input type="checkbox"/>			
Robot Alpha				0.5
State Display Time				0.05 s
Loop Animation	<input checked="" type="checkbox"/>			
Show Trail	<input checked="" type="checkbox"/>			
► Links				3
Add	Remove	Rename		

图6-14 RViz中MotionPlanning插件的设置

第一个标记的区域是Scene Robot，它将显示机器人模型；如果未选中，我们就看不到任何机器人模型。第二个标记的区域是Trajectory Topic，RViz在这里获取可视化的轨迹。如果我们想要为运动规划设置动画并想要显示运动轨迹，那么我们就应该启用此选项。

插件设置中的其他部分如图6-15所示。



图6-15 MotionPlaning插件中的规划请求设置

在图6-15中，我们可以看到Query Start State和Query Goal State选项。这些选项可以显示机械臂的起始姿态和目标姿态，就如我们在图6-13中看到的那样。Show Workspace将机器人周围的方形工作空间（所处世界的几何形状）可视化。可视化可以帮助我们调试运动规划算法，并详细了解机器人运动行为的细节。

在下一节中，我们将学习如何将MoveIt!配置软件包与Gazebo连接。这将在Gazebo中执行MoveIt!生成的轨迹。

6.3.2 MoveIt!配置软件包与Gazebo的接口

我们已经使用Gazebo仿真了机械臂及其控制器。如我们在MoveIt!架构中提到的那样，为了将MoveIt!中的机器臂连接到Gazebo，我们还需要一个具有FollowJoint-Trajectory-Action接口的轨迹控制器。

下面是将MoveIt!连接到Gazebo的过程：

第1步：为MoveIt!编写控制器配置文件

第1步是创建一个配置文件，用来与Gazebo中来自MoveIt!的轨迹控制器进行通信。我们需要在seven_dof_arm_config软件包的config文件夹中创建名为controller.yaml的控制器配置文件。

下面是controllers.yaml定义的一个示例：

```
controller_manager_ns: controller_manager
controller_list:
  - name: seven_dof_arm/seven_dof_arm_joint_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: true
    joints:
      - shoulder_pan_joint
      - shoulder_pitch_joint
      - elbow_roll_joint
      - elbow_pitch_joint
      - wrist_roll_joint
      - wrist_pitch_joint
      - gripper_roll_joint
```

```
- name: seven_dof_arm/gripper_controller
  action_ns: follow_joint_trajectory
  type: FollowJointTrajectory
  default: true
  joints:
    - finger_joint1
    - finger_joint2
```

控制器配置文件包含两个控制器接口的定义，一个用于机械臂，另一个用于夹爪。控制器使用的动作类型为FollowJointTrajectory，动作命名空间为follow_joint_trajectory。我们必须列出每组下的关节。default:true表明它将使用默认控制器，它是MoveIt!中用来与一组关节进行通信的主要控制器。

第2步：创建控制器启动文件

接下来我们必须创建一个名为seven_dof_arm_moveit_controller_manager.launch的新启动文件，它可以启动轨迹控制器。该文件名称是以机器人的名称开头，后面跟上_moveit_controller_manager结尾。

下面就是启动文件：

```
seven_dof_arm_config/launch/seven_dof_arm_moveit_controller_manager.launch的定义：
```

```
<launch>
  <!-- Set the param that trajectory_execution_manager needs to find the
controller plugin -->
  <arg name="moveit_controller_manager"
default="moveit_simple_controller_manager/MoveItSimpleControllerManager" />
  <param name="moveit_controller_manager" value="$(arg
moveit_controller_manager)" />

  <!-- load controller_list -->
  <arg name="use_controller_manager" default="true" />
  <param name="use_controller_manager" value="$(arg
use_controller_manager)" />

  <!-- Load joint controller configurations from YAML file to parameter
server -->
  <rosparam file="$(find seven_dof_arm_config)/config/controllers.yaml"/>
</launch>
```

该启动文件启动了MoveItSimpleControllerManager并加载了 controllers.yaml 中定义的关节轨迹控制器。

第3步：为Gazebo创建控制器配置文件

创建了MoveIt!文件后，我们必须创建Gazebo控制器配置文件和启动文件。

创建一个名为 trajectory_control.yaml 的新文件。该文件包含了需要与 Gazebo 一起加载的 GazeboROS 控制器列表。

你可以从第4章中创建的 seven_dof_arm_gazebo 软件包的 config 文件夹下获取此文件。

以下是该文件的定义：

```
seven_dof_arm:  
    seven_dof_arm_joint_controller:  
        type: "position_controllers/JointTrajectoryController"  
        joints:  
            - shoulder_pan_joint  
            - shoulder_pitch_joint  
            - elbow_roll_joint  
            - elbow_pitch_joint  
            - wrist_roll_joint  
            - wrist_pitch_joint  
            - gripper_roll_joint  
  
        gains:  
            shoulder_pan_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}  
            shoulder_pitch_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}  
            elbow_roll_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}  
            elbow_pitch_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}  
            wrist_roll_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}  
            wrist_pitch_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}  
            gripper_roll_joint: {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}  
  
gripper_controller:  
    type: "position_controllers/JointTrajectoryController"  
    joints:  
        - finger_joint1  
        - finger_joint2  
    gains:  
        finger_joint1: {p: 50.0, d: 1.0, i: 0.01, i_clamp: 1.0}  
        finger_joint2: {p: 50.0, d: 1.0, i: 0.01, i_clamp: 1.0}
```

在这里我们创建了
position_controllers/JointTrajectoryController，它具有跟arm和

gripper通信的FollowJointTrajectory动作接口。我们还定义了与每个关节相关的PID增益，这样就可以提供一个平滑的运动。

第4步：为Gazebo轨迹控制器创建启动文件

创建配置文件后，我们可以与Gazebo一起加载控制器。为此我们必须创建一个启动文件，这样就可以用单个命令同时启动Gazebo、轨迹控制器和MoveIt!的接口。

启动文件seven_dof_arm_bringup_moveit.launch中包含了启动所有这些命令的定义：

```
<launch>
  <!-- Launch Gazebo -->
  <include file="$(find
seven_dof_arm_gazebo)/launch/seven_dof_arm_world.launch" />

  <!-- ros_control seven dof arm launch file -->
  <include file="$(find
seven_dof_arm_gazebo)/launch/seven_dof_arm_gazebo_states.launch" />

  <!-- ros_control trajectory control dof arm launch file -->
  <include file="$(find
seven_dof_arm_gazebo)/launch/seven_dof_arm_trajectory_controller.launch" />

  <!-- moveit launch file -->
```

```
<include file="$(find  
seven_dof_arm_config)/launch/moveit_planning_execution.launch" />  
  
<node name="joint_state_publisher" pkg="joint_state_publisher"  
type="joint_state_publisher">  
    <param name="/use_gui" value="false"/>  
    <rosparam  
param="/source_list">[/move_group/fake_controller_joint_states]</rosparam>  
    </node>  
</launch>
```

该启动文件在Gazebo中生成机器人模型，发布关节状态，连接位置控制器，连接轨迹控制器；最后在MoveIt！软件包内启动moveit_planning_execution.launch文件，这样与RViz一起启动MoveIt!节点。如果MotionPlanning插件没有默认加载的话，我们要在RViz中加载一下。

我们可以在RViz中启动运动规划，然后使用下面的命令在Gazebo中执行规划的路径：

```
$ rosrun seven_dof_arm_gazebo seven_dof_arm_bringup_moveit.launch
```

注意，在正确启动规划场景前，我们需要使用下面的命令来安装MoveIt!所需的一些软件包，这样才能正常使用ROS控制器：

```
$ sudo apt-get install ros-kinetic-joint-state-controller ros-kinetic-  
position-controllers ros-kinetic-joint-trajectory-controller
```

安装了前面的软件包后，我们就可以启动规划场景了。这将启动RViz和Gazebo，我们可以在RViz中进行运动规划。运动规划后，点击Execute按钮将轨迹发送给Gazebo控制器，如图6-16所示。

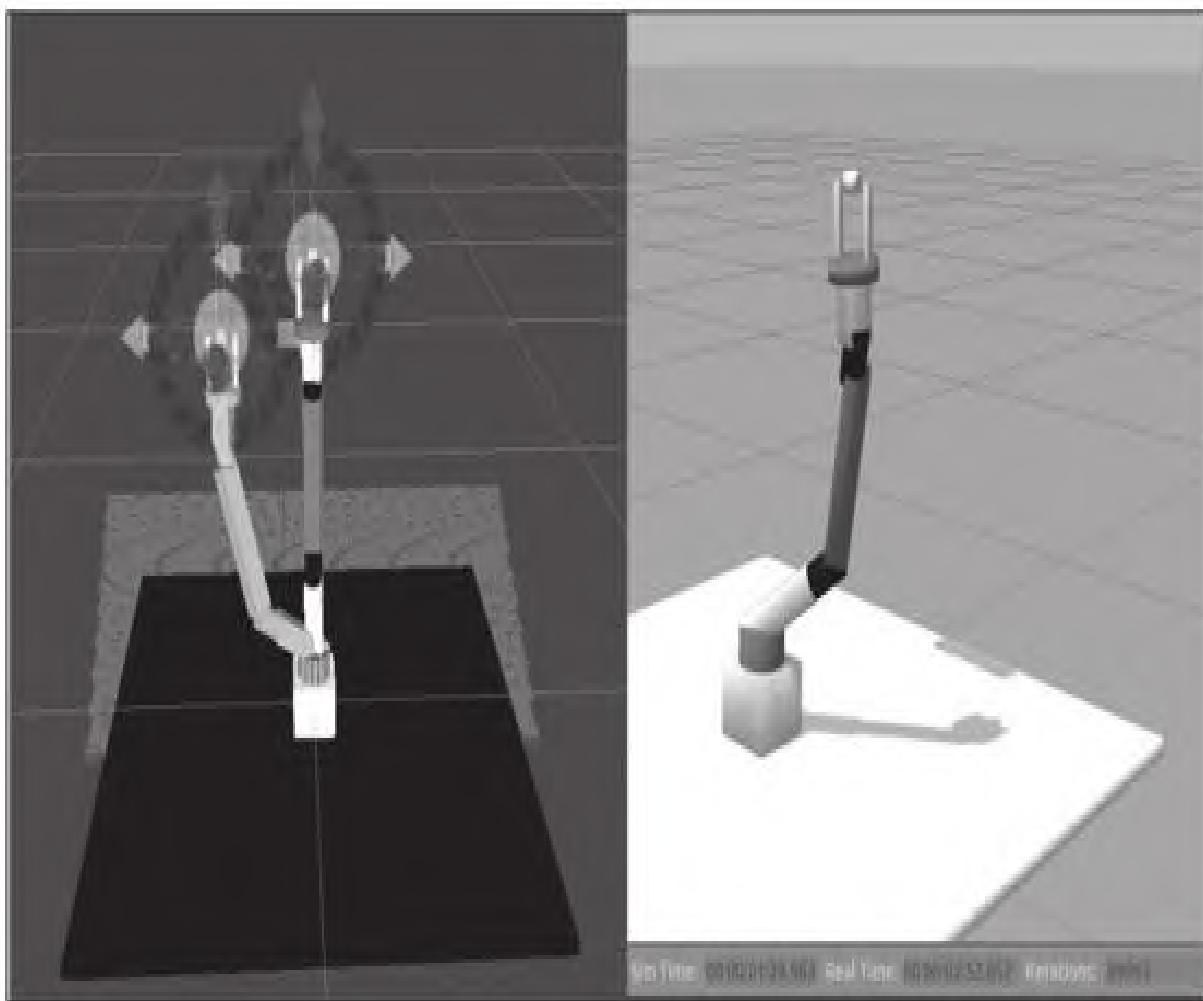


图6-16 Gazebo轨迹控制器执行来自MoveIt!的轨迹

第5步：调试Gazebo-MoveIt!接口

在本节中，我们将讨论Gazebo-MoveIt!接口中的一些常见问题和调试技术。

如果在Gazebo中轨迹没有执行，首先列出所有话题：

```
$ rostopic list
```

如果Gazebo控制器启动正确，我们将获得各关节轨迹话题，如图6-17所示。

```
/seven_dof_arm/gripper_controller/command  
/seven_dof_arm/gripper_controller/follow_joint_trajectory/cancel  
/seven_dof_arm/gripper_controller/follow_joint_trajectory/feedback  
/seven_dof_arm/gripper_controller/follow_joint_trajectory/goal  
/seven_dof_arm/gripper_controller/follow_joint_trajectory/result  
/seven_dof_arm/gripper_controller/follow_joint_trajectory/status  
/seven_dof_arm/gripper_controller/state  
/seven_dof_arm/joint_states  
/seven_dof_arm/seven_dof_arm_joint_controller/command  
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/cancel  
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/feedback  
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/goal  
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/result  
/seven_dof_arm/seven_dof_arm_joint_controller/follow_joint_trajectory/status  
/seven_dof_arm/seven_dof_arm_joint_controller/state  
/tf  
/tf_static  
/trajectory_execution_event
```

图6-17 来自Gazebo-ROS轨迹控制器的话题

我们可以看到gripper组和arm组的follow_joint_trajectory。如果控制器未准备就绪，轨迹将不会在Gazebo中执行。

此外，在加载启动文件时可以检查终端的输出信息。

```
[1505806707.153599116, 0.343000000]: Added FollowJointTrajectory controller for seven_dof_ar  
ller  
[1505806707.153740538, 0.343000000]: Returned 2 controllers in list  
[1505806707.205783246, 0.347000000]: Trajectory execution is managing controllers  
'move_group/ApplyPlanningSceneService'...  
'move_group/ClearOctomapService'...  
'move_group/MoveGroupCartesianPathService'...  
'move_group/MoveGroupExecuteTrajectoryAction'...  
'move_group/MoveGroupGetPlanningSceneService'...  
'move_group/MoveGroupKinematicsService'...  
'move_group/MoveGroupMoveAction'...  
'move_group/MoveGroupPickPlaceAction'...  
'move_group/MoveGroupPlanService'...  
'move_group/MoveGroupQueryPlannersService'...  
'move_group/MoveGroupStateValidationService'...  
  
[1505806835.903571251, 36.978000000]: arm[RRTkConfigDefault]: Starting planning with 1 state  
astructure  
[1505806835.994742622, 36.997000000]: arm[RRTkConfigDefault]: Created 21 states  
[1505806836.036028021, 37.004000000]: arm[RRTkConfigDefault]: Created 38 states  
[1505806836.038435520, 37.005000000]: ParallelPlan::solve(): Solution found by one or more t  
41 seconds
```

图6-18 终端信息显示轨迹执行成功

在图6-18中，第一部分显示了MoveItSimpleControllerManager可以连接到Gazebo控制器。如果它无法连接到控制器，在输出信息中会显示不能连接到控制器。第二部分显示了一个成功的运动规划。如果运动规划不成功，MoveIt!将不会给Gazebo发送轨迹信息。

下一节，我们将讨论ROS导航软件包集，并学习导航软件包集与Gazebo仿真接口进行交互时的必要条件。

6.4 理解ROS导航软件包集

ROS导航软件包的主要目的是将机器人从初始位置移动到目标位置，在移动过程中不会与周围环境发生任何碰撞。ROS导航软件包附带了几种导航相关的算法实现，它们可以帮助我们轻松实现移动机器人的自主导航。

用户只需要提供机器人的目标位置和来自轮子编码器、IMU、GPS等传感器获得的测量数据，激光扫描器的数据流，以及其他传感器数据流，例如激光雷达数据或来自Kinect等传感器的3D点云数据。导航软件包的输出是控制移动的速度命令，这些命令将控制机器人移动到目标位置。

导航软件包集包含了一些标准算法的实现，例如SLAM、A*(star)、Dijkstra、amcl等，可以直接用于我们的应用程序。

6.4.1 ROS导航硬件的要求

ROS导航软件包集被设计成一个通用的软件包集。机器人需要满足一些基本的硬件要求。下面就是这些要求：

- 导航软件包在差速驱动和完整约束（即机器人的总DOF等于机器人可控的DOF）方面表现更好。此外，应通过以下形式的速度命令控制移动机器人：x:velocity（线速度）、y:velocity（线速度）和theta:velocity（角速度）。
- 机器人需要配备视觉（RGB-D）或者激光传感器，用来构建周围环境的地图。
- 导航软件包集在方形或圆形的移动平台上工作将表现得更好。当然它也可以在任意形状的平台上工作，但是性能无法得到保证。

图6-19是从ROS网站

(<http://wiki.ros.org/navigation/Tutorials/RobotSetup>) 上获取的导航软件包集的基本组成框图。我们可以看到每个功能块的用途以及如何为自定义机器人配置导航软件包集。

由导航软件包集的框图可知，为了配置自定义机器人的导航软件包，我们需要提供与导航软件包集接口交互的功能。下面就是作为导航软件包集输入功能模块的说明：

- 里程计数据源：机器人的测距数据给出了机器人相对于起始位置的距离。主要的测距数据源来自轮子编码器、IMU、2D/3D相机（视觉测距）。里程计数据需要以nav_msgs/Odometry的消息形式发送到导航软件包集。里程计信息主要用于控制机器人的位置和移动速度。里程计数据是导航软件包集必需的输入数据。
- 传感器数据源：我们必须给导航软件包集提供激光扫描数据或者点云数据来扫描机器人周围的环境。这些数据与里程计数据一起构建机器人的全局和局部代价地图。这里使用的主要传感器是激光雷达或者

Kinect 3D传感器。数据需要封装成sensor_msgs/LaserScan或者sensor_msgs/PointCloud的形式。

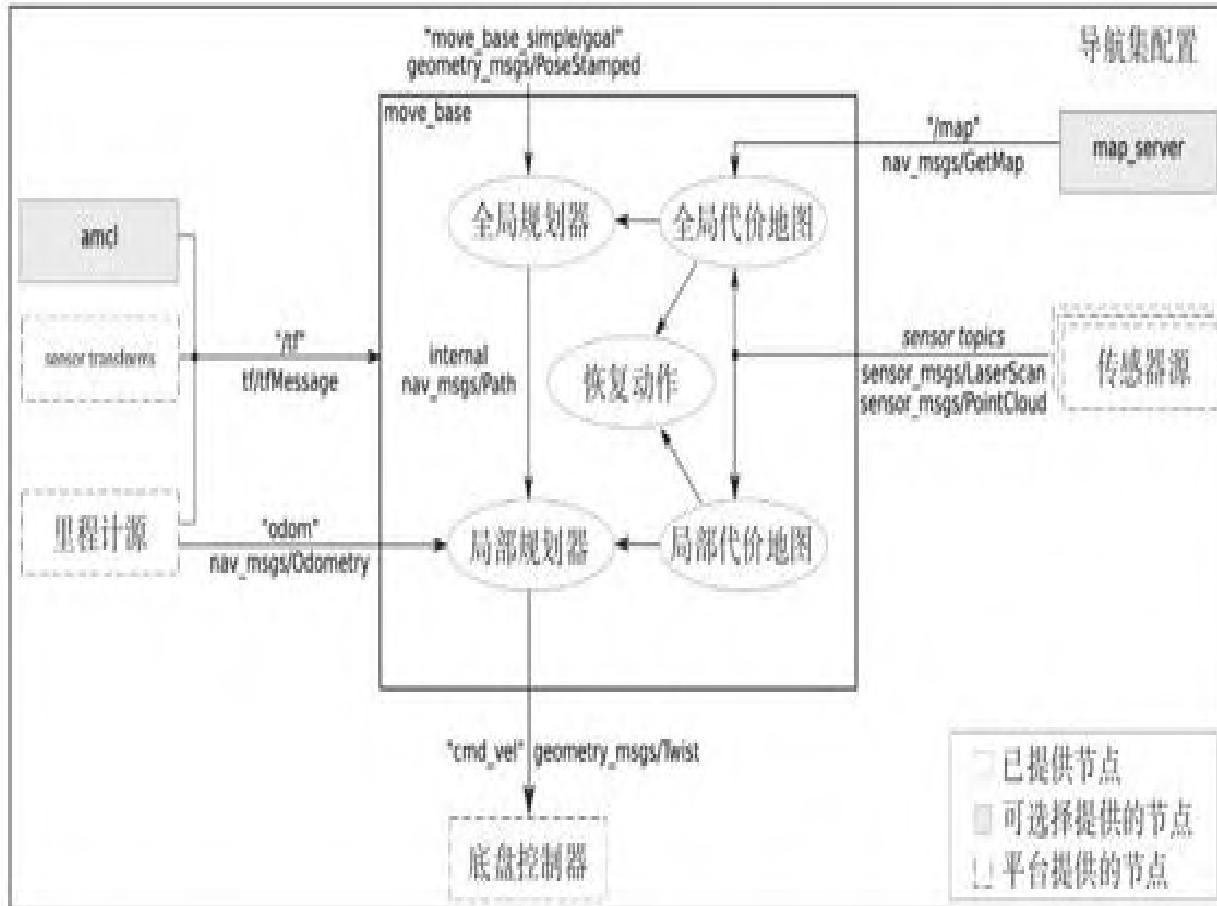


图6-19 导航软件包集的框图

- 传感器坐标变换/tf: 机器人需要使用ROS tf来发布机器人坐标系的变换关系。
- 底盘控制器: 底盘控制器的主要功能是转换导航软件包集的输出数据, 该消息是geometry_msgs/Twist类型的消息, 它将会被转换成机器人的相应的马达速度。

导航软件包集的可选节点有amcl和地图服务器, 它们可以帮助机器人定位以及保存/加载机器人的地图。

6.4.2 使用导航软件包

在使用导航软件包集之前，我们已经讨论了MoveIt!和move_group节点。在导航软件包集中也有一个类似move_group的节点，它被称为move_base节点。在图6-19中我们清楚地看到move_base节点从传感器、tf和里程计中获取输入数据。这与我们在MoveIt!中看到的move_group节点很类似。

下面我们进一步学习move_base节点。

理解move_base节点

move_base节点来自名为move_base的软件包。该软件包的主要功能是在其他导航节点的帮助下将机器人从当前位置移动到目标位置。该软件包中的move_base节点将全局规划和局部规划连接在一起进行路径规划。如果机器人在移动时被某些障碍困住动不了，那么它将开始调用rotate-recovery软件包，此时需要将全局代价地图和局部代价地图结合起来。

move_base节点基本上实现了SimpleActionServer。它以geometry_msgs/PoseStamped消息的形式发送目标位置信息。我们可以使用SimpleActionClient节点向此节点发送目标位置信息。

move_base节点从名为move_base_simple/goal的话题订阅目标位置信息。正如图6-19中所示，该话题是导航软件包集的一个输入数据。

当此节点接收到一个目标位置后，它将连接到如global_planner、local_planner、recovery_behavior、global_costmap和local_costmap之类的组件，以生成输出结果(geometry_msgs/Twist类型的速度指令)，并将其发送给底盘控制器，从而控制机器人移动到目标位置。

下面是move_base节点连接的所有软件包列表：

- global-planner：该软件包提供了函数库和节点，用来在地图上从机器人的当前位置到目标位置规划一条最优路径。该软件包实现的路径

查找算法有A*、Dijkstra等，这些算法用于查找从机器人当前位置到目标位置的最短路径。

- local-planner: 该软件包的主要功能是使用全局规划路径中的部分局部路径为机器人提供导航。局部规划将获取里程计数据、传感器数据，然后给机器人控制器发送合适的速度命令来完成全局规划路径的一部分。局部规划软件包是轨迹滑出（Trajectory Rollout）算法和动态窗口算法（Dynamic Window Algorithms, DWA）的具体实现。
- rotate-recovery: 该软件包通过执行360度旋转来帮助机器人从本地障碍中恢复。
- clear-costmap-recovery: 该软件包也是用来帮助机器人从本地障碍中恢复。但它是通过将当前导航软件包集使用的代价地图转换为静态地图后清除代价地图来达到摆脱障碍的目的。
- costmap-2D: 该软件包的主要用途是建立机器人周围环境地图。机器人只能通过地图来规划路径。在ROS中，我们创建2D或3D的栅格地图，用栅格单元表示周围的环境。每个栅格都有一个概率值，表示该区域是否被占据。costmap-2D软件包可以通过订阅激光扫描或者点云的传感器值来建立周围环境的栅格地图。这里的全局代价地图用于全局路径规划，局部代价地图用于局部路径规划。

下面是与move_base节点交互的其他软件包：

- map-server: 该软件包保存和加载costmap-2D软件包生成的地图。
- AMCL: AMCL (Adaptive Monte Carlo Localization) 是一种在地图上定位机器人位置的方法。这种方法使用粒子滤波器在概率论的帮助下跟踪机器人相对于地图的位置。在ROS系统中，AMCL是通过接收sensor_msgs/LaserScan消息来建立地图的。
- gmapping: 该软件包是Fast SLAM算法的一种实现，它使用激光扫描数据和里程计数据构建一个2D占据栅格地图。

讨论了导航软件包集的每个功能模块后，让我们来看看它们是如何具体工作的。

使用导航软件包集

在上一小节中，我们认识了ROS导航软件包集中每个模块的功能。接下来我们来看整个系统是如何工作。机器人需要发布一个合适的里程计数据、`tf`信息和激光传感器的数据，并且需要一个底盘控制器和周围环境的地图。

如果所有需要都满足，我们就可以开始使用导航软件包了。

在地图上定位

机器人将要执行的第一步就是在地图上进行定位。`AMCL`就是机器人用来在地图上定位的软件包。

发送目标和路径规划

获取了机器人的当前位置后，我们就可以向`move_base`节点发送一个目标位置。然后，`move_base`节点将此目标位置发送给全局规划器。该规划器将规划一条从当前机器人位置到目标位置的路径。

这个规划是根据地图服务器提供的全局代价地图进行规划的。全局规划器将此路径发送给局部规划器。最后，局部规划器将执行全局规划的每段路径。

局部规划器从`move_base`节点获取里程计信息和传感器数据，并为机器人找出一个无碰撞的局部规划。局部规划器从局部代价地图获取信息，局部代价地图用来监视机器人周围的障碍物。

碰撞恢复行为

全局代价地图和局部代价地图与激光扫描数据关联在一起。如果机器人在某处卡住了，那么导航软件包将触发恢复行为节点，例如通过清除代价地图恢复或者通过原地旋转恢复。

发送速度命令

局部规划器以包含线速度和角速度的twist消息
(geometry_msgs/Twist) 的形式向机器人底盘控制器发送生成的控制
移动命令。机器人底盘控制器负责将twist消息转换为相对应的马达速
度。

6.5 安装ROS导航软件包集

ROS桌面完整安装不会安装ROS导航软件包集。我们必须使用下面的命令来单独安装导航软件包集：

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

安装了导航软件包后，下面我们来学习如何构建机器人周围环境的地图。在这里我们使用前一章中讨论过的差速轮式机器人。该机器人满足了导航软件包集的所有（3个）要求。

6.6 使用SLAM构建地图

ROS gmapping软件包是开源SLAM算法OpenSLAM (<https://www.openslam.org/gmapping.html>) 的修改版本。该软件包包含一个名为slam_gmapping的节点，即SLAM算法的实现。该软件包利用激光扫描数据和移动机器人姿态辅助构建一个2D占据栅格地图。

SLAM的基本硬件要求是机器人要有里程计数据，以及水平安装于机器人顶部的激光雷达。本章的机器人已经满足了这些要求。我们可以按照下面的过程用gmapping软件包生成机器人环境的2D地图。

使用gmapping操作之前，我们需要先用下面的命令安装gmapping：

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

6.6.1 为gmapping创建启动文件

为gmapping创建启动文件时的主要任务是为slam_gmapping节点和move_base节点设置相应参数。slam_gmapping节点是ROS gmapping软件包中的核心节点。slam_gmapping节点订阅激光数据(sensor_msgs/LaserScan)和tf数据，并将占据栅格地图数据作为输出(nav_msgs/OccupancyGrid)发布。该节点具有高度可配置性，并且我们可以对参数进行微调以改进地图精度。这些参数可以在网址<http://wiki.ros.org/gmapping>中查看。

接下来我们要配置的另一个节点是move_base节点。我们需要配置的主要参数是全局和局部代价地图参数、局部规划器和move_base参数。这些参数的列表很长，所以我们是在几个YAML文件中来表示这些参数的。这些参数文件都包含在diff_wheeled_robot_gazebo软件包中的param文件夹下。

下面就是该机器人使用的gmapping.launch文件。该启动文件存放 在diff_wheeled_robot_gazebo/launch文件夹中：

```
<launch>
  <arg name="scan_topic" default="scan" />

  <!-- Defining parameters for slam_gmapping node -->

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
output="screen">
    <param name="base_frame" value="base_footprint"/>
    <param name="odom_frame" value="odom"/>
    <param name="map_update_interval" value="5.0"/>
    <param name="maxUrange" value="6.0"/>
    <param name="maxRange" value="8.0"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="lskip" value="0"/>
    <param name="minimumScore" value="100"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>
    <param name="linearUpdate" value="0.5"/>
    <param name="angularUpdate" value="0.436"/>
```

```
<param name="temporalUpdate" value="-1.0"/>
<param name="resampleThreshold" value="0.5"/>
<param name="particles" value="80"/>
<param name="xmin" value="-1.0"/>
<param name="ymin" value="-1.0"/>
<param name="xmax" value="1.0"/>
<param name="ymax" value="1.0"/>

<param name="delta" value="0.05"/>
<param name="llsamplerange" value="0.01"/>
<param name="llsamplestep" value="0.01"/>
<param name="lasamplerange" value="0.005"/>
<param name="lasamplestep" value="0.005"/>
<remap from="scan" to="$(arg scan_topic)"/>
</node>

<!-- Defining parameters for move_base node -->

<node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
  <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/costmap_common_params.yaml" command="load"
ns="global_costmap" />
  <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/costmap_common_params.yaml" command="load"
ns="local_costmap" />
  <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/local_costmap_params.yaml" command="load"
/>
  <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/global_costmap_params.yaml" command="load"
/>
  <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/base_local_planner_params.yaml"
command="load" />
  <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/dwa_local_planner_params.yaml"
command="load" />
  <rosparam file="$(find
diff_wheeled_robot_gazebo)/param/move_base_params.yaml" command="load" />
</node>

</launch>
```

6.6.2 在差速驱动机器人上运行SLAM

为了构建地图，我们需要编译ROS软件包：`diff_wheeled_robot_gazebo`，并运行`gmapping.launch`启动文件。下面就是启动地图构建过程的命令。

通过使用Willow Garage地图开始机器人仿真：

```
$ rosrun diff_wheeled_robot_gazebo diff_wheeled_gazebo_full.launch
```

使用如下命令运行`gmapping`的启动文件：

```
$ rosrun diff_wheeled_robot_gazebo gmapping.launch
```

如果`gmapping`启动文件运行正常，我们将在终端上看到如图6-20的输出。

```
[ INFO] [1505810240.049575967, 15.340000000]: Loading from pre-hydro parameter style
[ INFO] [1505810240.168699314, 15.381000000]: Using plugin "static_layer"
[ INFO] [1505810240.384469019, 15.449000000]: Requesting the map...
[ INFO] [1505810240.663457937, 15.552000000]: Resizing costmap to 288 X 608 at 0.050000 m/pix
[ INFO] [1505810240.871384865, 15.658000000]: Received a 288 X 608 map at 0.050000 m/pix
[ INFO] [1505810240.897210021, 15.656000000]: Using plugin "obstacle_layer"
[ INFO] [1505810240.913185546, 15.660000000]: Subscribed to Topics: scan bump
[ INFO] [1505810241.183408917, 15.714000000]: Using plugin "inflation_layer"
[ INFO] [1505810241.592248141, 15.851000000]: Loading from pre-hydro parameter style
[ INFO] [1505810241.730240828, 15.900000000]: Using plugin "obstacle_layer"
[ INFO] [1505810241.978042290, 16.015000000]: Subscribed to Topics: scan bump
[ INFO] [1505810242.124180243, 16.057000000]: Using plugin "inflation_layer"
[ INFO] [1505810242.504991688, 16.191000000]: Created local_planner dwa_local_planner/DWAPlannerROS
[ INFO] [1505810242.518319734, 16.198000000]: Sim period is set to 0.20
[ INFO] [1505810244.343111055, 16.967000000]: Recovery behavior will clear layer obstacles
[ INFO] [1505810244.546680028, 17.020000000]: Recovery behavior will clear layer obstacles
[ INFO] [1505810244.697982461, 17.046000000]: odom received!
```

图6-20 gmapping过程中的终端信息

在当前环境中使用键盘人工遥控机器人遍历整个环境。机器人只有在覆盖了整个区域后才能给出整个环境的地图：

```
$ roslaunch diff_wheeled_robot_control keyboard_teleop.launch
```

图6-21展示的是当前机器人的Gazebo视图。在机器人周围环境中存在一些障碍物。

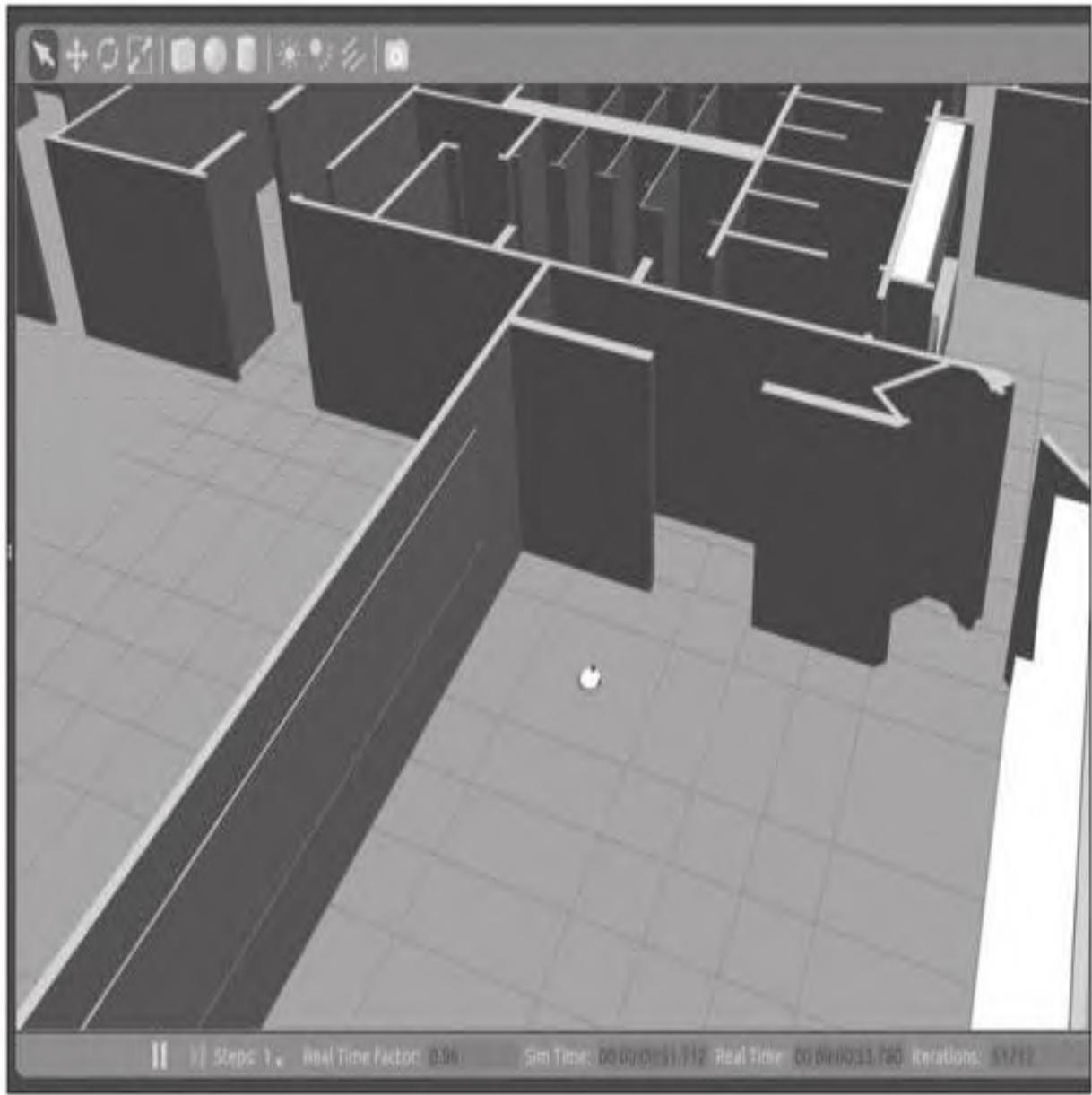


图6-21 使用Willow Garage地图仿真机器人

我们可以启动RViz并添加一个名为Map的显示类型，然后设置话题名为/map。

我们可以通过使用键盘遥控机器人在虚拟测试环境中移动，同时我们将看到根据周围环境构建地图的过程。图6-22显示了在RViz中看到的环境地图。

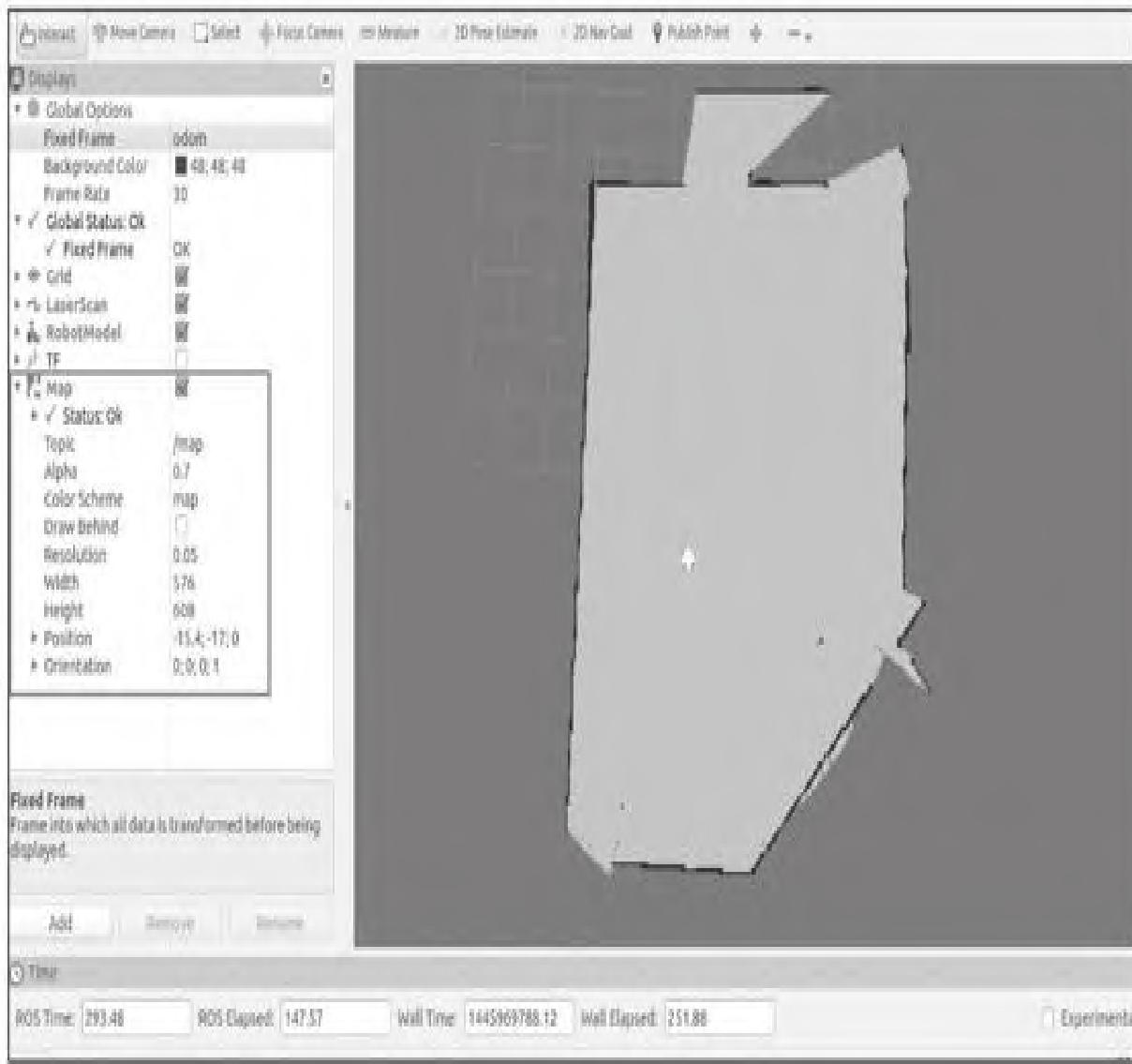


图6-22 在RViz中查看构建完成的房间地图

我们可以使用下面的命令保存构建的地图。该命令将监听地图话题并将地图保存为图像格式：

```
$ rosrun map_server map_saver -f willo
```

这里的willo是保存的地图文件名。地图文件被保存为两个文件：一个是YAML文件，其中包含了地图的元数据和图像名称；第二个是图

像本身，其中包含了占据栅格地图的编码数据。图6-23是正确运行该命令过程的截图。

```
jcacace@robot:~$ rosrun map_server map_saver -f willo
[INFO] [1505810794.895750258]: Waiting for the map
[INFO] [1505810795.117276658, 21.621000000]: Received a 288 X 608 map @ 0.050 m/pix
[INFO] [1505810795.119888038, 21.621000000]: Writing map occupancy data to willo.pgm
[INFO] [1505810795.138065942, 21.632000000]: Writing map occupancy data to willo.yaml
[INFO] [1505810795.138632329, 21.632000000]: Done
```

图6-23 保存地图时的终端截图

图6-24显示了保存地图编码的图像。如果机器人提供了精确的里程计数据，我们将得到与其所处环境类似的精确地图。精确的地图将有助于路径规划，最终可以提高自主导航的精度。

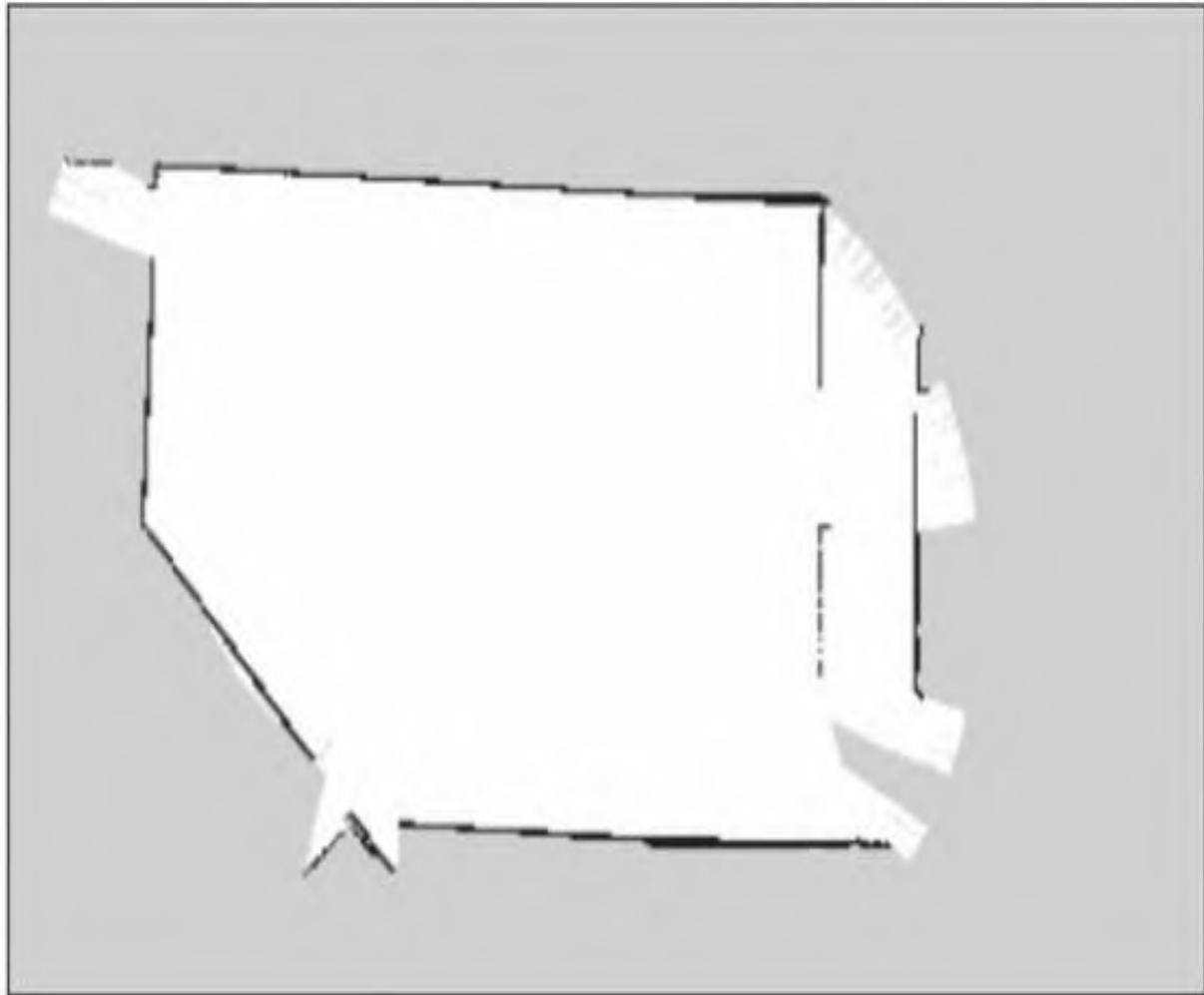


图6-24 保存的地图

下面是在这个静态的地图中进行定位和导航的过程。

6.6.3 使用amcl和静态地图实现自主导航

ROS的amcl软件包提供节点用于在静态地图中对机器人进行定位。amcl节点将订阅激光扫描数据，基于激光扫描的地图和来自机器人的tf信息。amcl节点将估计机器人在地图上的姿态，然后发布相对于地图的估计位置。

如果我们根据激光扫描数据创建一个静态地图。机器人可以使用amcl和move_base节点从地图上的任意位置开始进行自主导航。第一步就是创建启动amcl节点的启动文件。amcl节点是高度可配置的。我们可以用许多参数配置它。参数的列表可以在ROS软件包网站（<http://wiki.ros.org/amcl>）上找到。

6.6.4 创建amcl启动文件

下面将给出一个典型的amcl启动文件。amcl节点在 amcl.launch.xml文件中配置，该文件位于 diff_wheeled_robot_gazebo/launch/include软件包中。move_base节点也可以在move_base.launch.xml文件中单独配置。用map_server节点，我们可以在此加载gmapping过程创建的地图文件：

```
<launch>

    <!-- Map server -->
    <arg name="map_file" default="$(find
diff_wheeled_robot_gazebo)/maps/test1.yaml"/>
    <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />

    <include file="$(find
diff_wheeled_robot_gazebo)/launch/includes/amcl.launch.xml">

        <arg name="initial_pose_x" value="0"/>
        <arg name="initial_pose_y" value="0"/>
        <arg name="initial_pose_a" value="0"/>

    </include>

    <include file="$(find
diff_wheeled_robot_gazebo)/launch/includes/move_base.launch.xml"/>

</launch>
```

下面是amcl.launch.xml的代码片段。这个文件比较长，因为我们必须给amcl节点配置大量参数：

```
<launch>
  <arg name="use_map_topic" default="false"/>
  <arg name="scan_topic"      default="scan"/>
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>

  <node pkg="amcl" type="amcl" name="amcl">
    <param name="use_map_topic"           value="$(arg use_map_topic)"/>
    <!-- Publish scans from best pose at a max of 10 Hz -->
    <param name="odom_model_type"        value="diff"/>
    <param name="odom_alpha5"            value="0.1"/>
    <param name="gui_publish_rate"       value="10.0"/>
    <param name="laser_max_beams"         value="60"/>
    <param name="laser_max_range"        value="12.0"/>
```

创建了该启动文件后，我们可以按照下面的步骤启动amcl节点：

在Gazebo中开始机器人仿真：

```
$ roslaunch diff_wheeled_robot_gazebo diff_wheeled_gazebo_full.launch
```

用如下命令加载amcl启动文件：

```
$ roslaunch diff_wheeled_robot_gazebo amcl.launch
```

如果amcl启动文件加载正确，终端将显示如图6-25所示的消息。

如果amcl工作正常，我们可以用RViz控制机器人移动到地图上指定的位置，如图6-26所示。图中，箭头表示目标位置。我们必须在RViz中启用LaserScan、Map和Path可视化插件来查看激光扫描数据、

全局/局部代价地图以及全局/局部路径。使用RViz中的2D NavGoal按钮我们就可以控制机器人移动到指定的位置。

机器人将规划到达目标位置的路径，并且向机器人控制器发送速度命令以到达那个位置：

```
[ INFO] [1505821904.100025792, 139.365000000]: Using plugin "static_layer"
[ INFO] [1505821904.277281445, 139.434000000]: Requesting the map...
[ INFO] [1505821904.489128458, 139.541000000]: Resizing costmap to 512 X 480 at 0.050000 m/pix
[ INFO] [1505821904.667453907, 139.643000000]: Received a 512 X 480 map at 0.050000 m/pix
[ INFO] [1505821904.675176688, 139.648000000]: Using plugin "obstacle_layer"
[ INFO] [1505821904.681719452, 139.648000000]: Subscribed to Topics: scan bump
[ INFO] [1505821904.813327088, 139.699000000]: Using plugin "inflation_layer"
[ INFO] [1505821905.081866940, 139.802000000]: Using plugin "obstacle_layer"
[ INFO] [1505821905.194340020, 139.871000000]: Subscribed to Topics: scan bump
[ INFO] [1505821905.323469494, 139.903000000]: Using plugin "inflation_layer"
[ INFO] [1505821905.674954354, 140.036000000]: Created local_planner dwa_local_planner/DWAPlannerROS
[ INFO] [1505821905.689447045, 140.040000000]: Sim period is set to 0.20
[ INFO] [1505821907.560275254, 141.046000000]: Recovery behavior will clear layer obstacles
[ INFO] [1505821907.785016235, 141.138000000]: Recovery behavior will clear layer obstacles
[ INFO] [1505821907.949123108, 141.197000000]: odom received!
```

图6-25 执行amcl时的终端截图

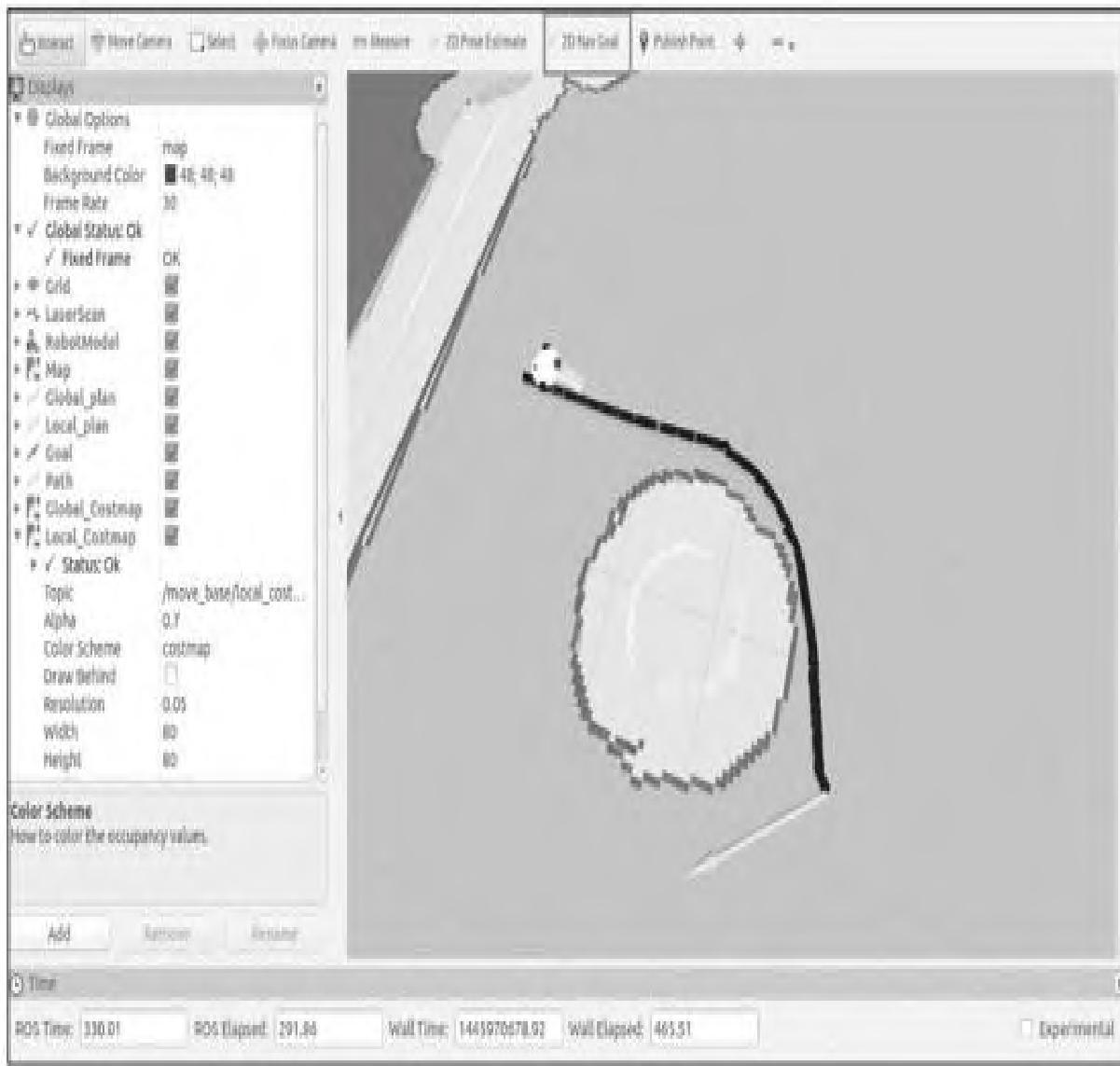


图6-26 使用amcl和地图进行自主导航

在图6-26中，我们可以看到在机器人的运动路径中放置了一个随机障碍物，然后机器人就重新规划了一条可以避开障碍物的路径。

我们可以在RViz上添加Pose Array（将话题设置为`/particle_cloud`）来观察机器人周围的粒子云。图6-27展示了机器人周围的amcl粒子。

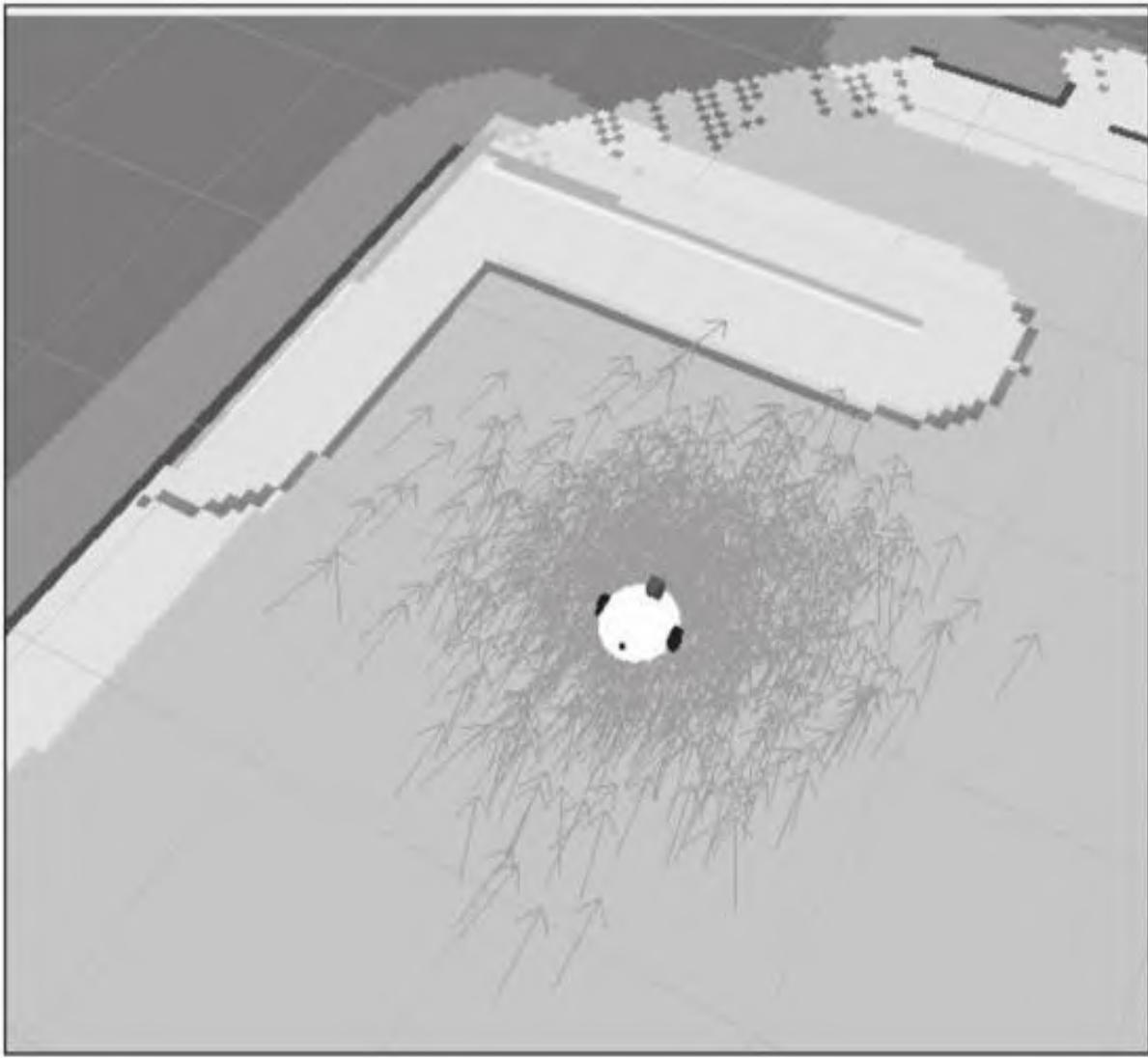


图6-27 amcl粒子云

6.7 习题

- MoveIt!软件包的主要用途是什么？
- MoveIt!软件包中move_group节点为何如此重要？
- 导航软件包集中move_base节点的用途是什么？
- SLAM和amcl软件包的功能是什么？

6.8 本章小结

本章简要概述了MoveIt!和ROS导航软件包集，并使用机械臂和移动底盘在Gazebo中仿真演示了其功能。本章以MoveIt!概述开始，讨论了有关MoveIt!的详细概念。讨论完MoveIt!，我们将MoveIt!和Gazebo进行连接。然后，在Gazebo上执行MoveIt!规划的轨迹。

另一部分内容是关于ROS导航软件包集。我们同样讨论了其概念和工作原理。之后，我们尝试将Gazebo中的机器人与导航软件包集进行连接，并使用SLAM构建地图。完成SLAM后，我们用amcl和静态地图进行自主导航。

下一章，我们将讨论pluginlib、小节点和控制器。

第7章

使用pluginlib、小节点和Gazebo插件

前一章，我们讨论了机械臂和移动机器人在ROS MoveIt!和导航软件包集中的接口及仿真。在本章，我们将介绍ROS中的一些高级概念，例如ROS pluginlib软件包、小节点（nodelet）和Gazebo插件。我们将讨论每个概念的功能和应用，并通过示例了解其工作原理。在前面章节中，我们已经使用Gazebo插件获取了在Gazebo仿真器中的传感器数据和机器人行为数据。在这一章节中，我们将学习如何创建这样的一个插件。我们还将讨论一种改进的ROS节点，称作ROS小节点（nodelet）。ROS中的这些功能是使用pluginlib的插件架构实现的。

本章将介绍以下内容：

- 理解pluginlib
- 使用pluginlib实现一个示例插件
- 理解ROS小节点
- 实现一个小节点示例
- 理解并创建一个Gazebo插件

7.1 理解pluginlib

在计算机领域，插件是很常用的术语。插件是一种模块化的软件，可以在现有应用软件的基础上增加一些新的功能。插件的优点是我们不需要在主应用中编写所有功能。相反，我们只需要在主应用中建立一个软件架构，并能接受新的插件。通过这种方法，我们可以任意扩展软件功能。

我们也需要为自己的机器人应用安装一些插件，当我们开始为机器人开发复杂的ROS应用程序时，插件将是我们扩充应用功能的一个不错的选择。

ROS系统提供了pluginlib插件框架，该框架可以动态地加载或卸载插件。插件可以是一个库，也可以是一个类。pluginlib代表一组C++库，该库可以帮助我们编写插件，也可以在需要时加载或卸载某一插件。

Plugin文件是一组运行时库，如共享对象库（. so）或动态链接库（. DLL）。它们是在不链接到主应用程序代码的情况下编译生成的。插件是与主应用软件没有任何依赖关系的独立实体。

插件的主要优点是我们可以在不对主应用代码做太多修改的情况下扩展应用软件的功能。

我们可以使用pluginlib创建一个简单的插件，并且可以看到使用ROS pluginlib创建一个插件所涉及的所有步骤。

在这里，我们将使用pluginlib创建一个简单的计算器应用。我们通过使用插件来增加计算器的各个功能。

使用pluginlib为计算器应用创建插件

与编写一段简单的代码相比，使用插件创建一个计算器应用是一项略微烦琐的任务。然而，此示例的主要目的是展示如何在不修改主应用代码的情况下为计算器添加新功能。

在本示例中，我们将看到一个计算机应用程序，它可以加载插件来执行每项功能。在这里，我们只实现了它的主要功能，如加法、减法、乘法和除法。我们可以通过为每个操作编写单独的插件来扩展其功能。

在继续创建插件定义之前，我们可以参考一下在 `pluginlib_calculator` 文件夹中的计算器代码。

我们将创建一个名为`pluginlib_calculator`的ROS软件包来构建这些插件和主计算器应用程序。

图7-1展示了如何在`pluginlib_calculator` ROS软件包中组织计算器插件和应用程序。

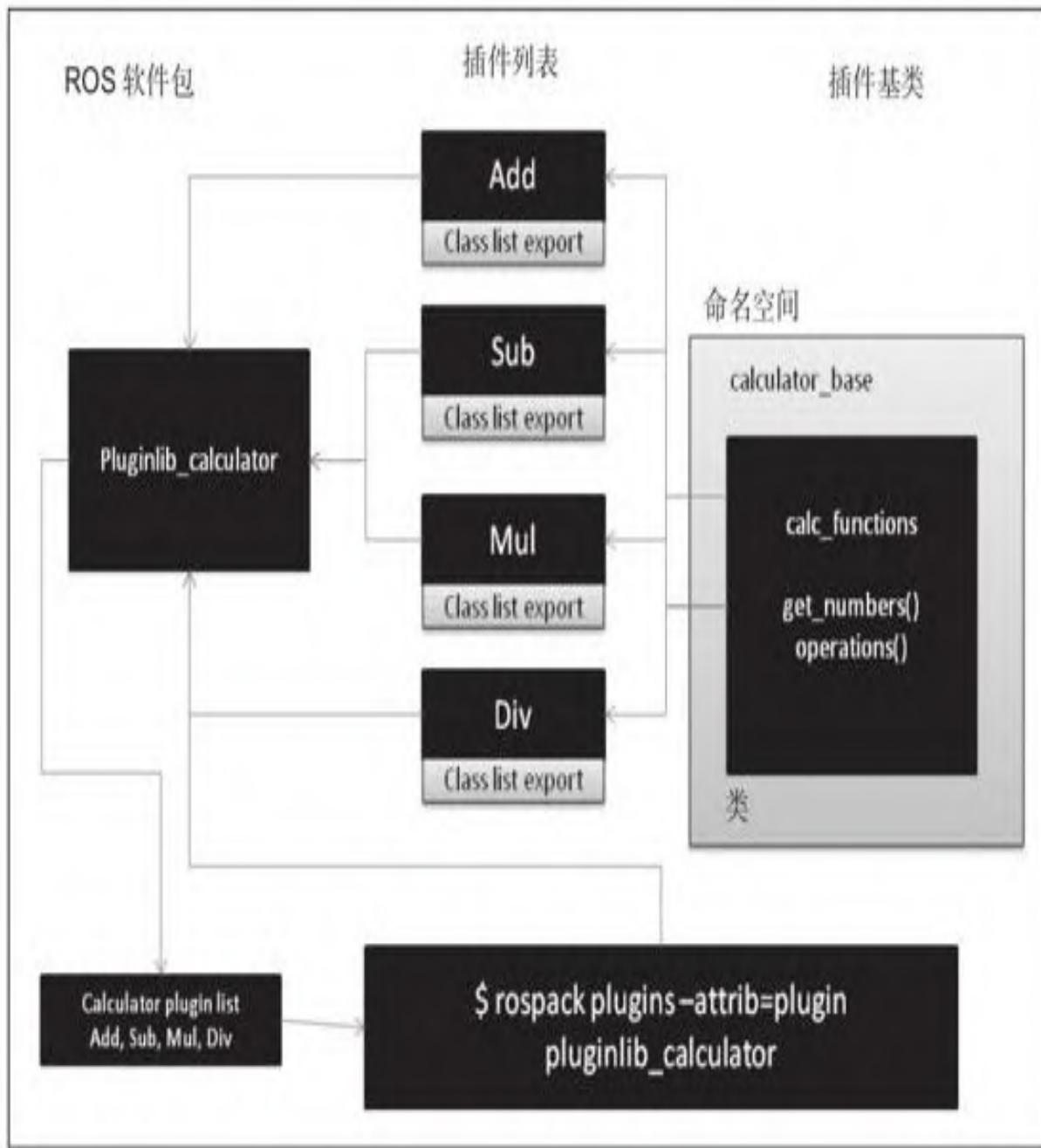


图7-1 计算器应用程序中的插件架构

我们可以看到计算器的插件列表和一个名为`calc_functions`的插件基类。该插件基类实现了这些插件所需的通用功能。

这就是我们创建ROS软件包并开始为主计算器应用程序开发插件的方法。

使用pluginlib_calculator软件包

为了快速入手，我们可以使用现有的ROS插件包
pluginlib_calculator。

如果想从头开始创建该软件包，我们可以使用下面的命令：

```
$ catkin_create_pkg pluginlib_calculator pluginlib roscpp std_msgs
```

该软件包主要依赖的就是pluginlib。为了编译生成该插件，我们会讨论该软件包中的主要源码文件。然而，你可以从本书提供的代码中获取插件代码或者从下面的链接中下载：

```
$ git clone https://github.com/jocacace/plugin_calculator
```

第1步：创建calculator_base头文件

文件calculator_base.h保存在
pluginlib_calculator/include/pluginlib_calculator文件夹中。该文件的主要用途是声明插件中常用的函数或方法：

```
namespace calculator_base
{
class calc_functions
{
```

在这段代码中，我们声明了一个名为calc_functions的类，它封装了插件使用的函数。这个类包含在calculator_base命名空间中。我们可以在此命名空间中添加更多类来扩展此基类的功能。

```
virtual void get_numbers(double number1, double number2) = 0;  
virtual double operation() = 0;
```

这些都是在calc_function类中实现的主要方法。get_number()函数可以检索到两个数字作为计算器的输入，而operation()函数定义了我们想要执行的数学运算。

第2步：创建calculator_plugins头文件

calculator_plugins.h文件保存在pluginlib_calculator/include/pluginlib_calculator文件夹中，该文件的主要用途是定义计算器插件的完整功能，这些插件可以被命名为Add、Sub、Mul及Div。下面给出了这段代码的解析：

```
#include <pluginlib_calculator/calculator_base.h>  
#include <cmath>  
  
namespace calculator_plugins  
{  
    class Add : public calculator_base::calc_functions  
    {
```

该头文件包括了calculator_base.h，它用于获取计算器的基本函数。每个插件都被定义为一个类，它继承了calculator_base.h类中的calc_functions类：

```
class Add : public calculator_base::calc_functions
{
public:
Add()
{
    number1_ = 0;
    number2_ = 0;
}

void get_numbers(double number1, double number2)
{
    try{

        number1_ = number1;
        number2_ = number2;
    }

    catch(int e)
    {
        std::cerr<<"Exception while inputting numbers"<<std::endl;
    }
}

double operation()
{
    return(number1_+number2_);
}

private:
    double number1_;
    double number2_;
};

};
```

在这段代码中，我们可以看到继承的函数get_numbers()和operations()的定义。函数get_numbers()用于检索两个数字输入参数的计算。函数operations()执行想要的运算，在当前情况下，它用于执行加法运算。我们也可以在这个头文件中看到其他所有插件的定义。

第3步：用calculator_plugins.cpp导出插件

为了动态地加载这个插件，我们必须使用一个特定的宏PLUGINLIB_EXPORT_CLASS来导出每个类。这个宏必须存在于由插件类组成的任何CPP文件中。我们已经定义了插件类，并且在这个文件中，我们将仅定义宏语句。

从pluginlib_calculator/src文件夹中可以找到calculator_plugins.cpp文件，以下是导出每个插件的方法：

```
#include <pluginlib/class_list_macros.h>
#include <pluginlib_calculator/calculator_base.h>
#include <pluginlib_calculator/calculator_plugins.h>

PLUGINLIB_EXPORT_CLASS(calculator_plugins::Add,
calculator_base::calc_functions);
```

在PLUGINLIB_EXPORT_CLASS中，我们需要提供插件的类名和基类。

第4步：用calculator_loader.cpp实现插件加载器

插件加载器节点负责加载每个插件，将数字输入到每个插件后从插件中获取结果。我们可以从pluginlib_calculator/src文件夹中找到calculator_loader.cpp文件。

以下是此代码的解析：

```
#include <boost/shared_ptr.hpp>
#include <pluginlib/class_loader.h>
#include <pluginlib_calculator/calculator_base.h>
```

这些是加载插件必需的头文件；

```
pluginlib::ClassLoader<calculator_base::calc_functions>
calc_loader("pluginlib_calculator", "calculator_base::calc_functions");
```

pluginlib提供了ClassLoader类，它位于class_loader.h中，用于在运行时加载类。我们需要为加载器和计算器基类提供一个名称作为参数。

```
boost::shared_ptr<calculator_base::calc_functions> add =
calc_loader.createInstance("pluginlib_calculator/Add");
```

这将使用ClassLoader对象创建一个add类的实例；

```
add->get_numbers(10.0,10.0);
double result = add->operation();
```

这些代码用于提供输入并在插件实例中执行操作。

第5步：创建插件描述文件：calculator_plugins.xml

生成计算器加载器代码之后，下一步我们必须在名为Plugin Description File的XML文件中描述此软件包内的插件列表。插件描述文件包含了软件包所含插件的所有信息，例如类的名称、类的类型、基类等。

插件描述文件是一个基于插件软件包的重要文件，因为它有助于ROS系统自动查找、加载插件。它还包含诸如插件描述之类的信息。

下面的代码展示了软件包中名为calculator_plugins.xml的插件描述文件，该文件与CMakeLists.txt和package.xml文件一起存储。你可以从软件包本身中获取此文件。

以下是此文件的解析：

```
<library path="lib/libpluginlib_calculator">
  <class name="pluginlib_calculator/Add" type="calculator_plugins::Add"
    base_class_type="calculator_base::calc_functions">
    <description>This is a add plugin.</description>
  </class>
```

此段代码是Add插件的代码，它定义了插件的库路径、类名、类的类型、基类以及描述信息。

第6步：使用ROS软件包系统注册插件

为了让pluginlib在ROS系统中查找到所有基于插件的软件包，我们应该在package.xml中导出插件描述文件。如果不包含此插件，ROS系统将无法找到软件包内的插件。

在这里，我们在package.xml中添加export标签，如下所示：

```
<export>
  <pluginlib_calculator plugin="${prefix}/calculator_plugins.xml" />
</export>
```

要正确使用此导出命令，我们应该在package.xml中插入以下代码：

```
<build_depend>pluginlib_calculator</build_depend>
<run_depend>pluginlib_calculator</run_depend>
```

无论是在编译时还是在运行时，当前这个软件包应该直接依赖其自身。

第7步：编辑CMakeLists.txt文件

与其他普通的ROS节点的另一个区别就是CMakeLists.txt文件中包含的编译指令。为了编译计算器插件和加载器节点，我们应该在CMakeLists.txt文件中添加下面几行：

```
## pluginlib_tutorials library
add_library(pluginlib_calculator src/calculator_plugins.cpp)
target_link_libraries(pluginlib_calculator ${catkin_LIBRARIES})
## calculator_loader executable
add_executable(calculator_loader src/calculator_loader.cpp)
target_link_libraries(calculator_loader ${catkin_LIBRARIES})
```

到目前为止我们几乎完成了所有的配置，现在是时候使用catkin_make命令来编译软件包了。

第8步：查询软件包中的插件列表

如果软件包编译正常，那么我们就可以执行加载器了。下面的命令将查询包含在软件包中的插件：

```
$ rospack plugins --attrib=plugin pluginlib_calculator
```

如果一切都被正确编译的话，我们将得到如图7-2的结果。

```
jcacace@robot:~$ rospack plugins --attrib=plugin pluginlib_calculator
pluginlib_calculator /home/jcacace/catkin_ws/src/MASTERING_ROS/ch6/pluginlib_calculator/calculator_
plugins.xml
```

图7-2 插件查询结果

第9步：运行插件加载器

启动roscore后，我们可以使用以下命令来执行calculator_loader：

```
$ rosrun pluginlib_calculator calculator_loader
```

图7-3展示了此命令的输出结果，用来检查是否一切都正常。这个加载器将两个输入都设置为10.0，之后我们使用插件得到了正确的结果，如图7-3所示。

```
jcacace@robot:~$ rosrun pluginlib_calculator calculator_loader
[ INFO] [1506769896.353657043]: Triangle area: 20.00
[ INFO] [1506769896.353796789]: Subtracted result: 0.00
[ INFO] [1506769896.353853201]: Multiplied result: 100.00
[ INFO] [1506769896.353886772]: Division result: 1.00
```

图7-3 插件加载器节点的结果

下一节，我们将介绍一个新概念：小节点（nodelet），并讨论如何实现小节点。

7.2 理解ROS小节点

小节点（nodelet）是特殊的ROS节点，旨在以有效的方式在同一进程中运行多个算法，它以线程的形式执行每个进程。线程节点可以有效地相互通信而不会使网络过载，两个节点之间没有复制传输。这些线程节点也可以与外部的节点通信。

正如pluginlib那样，我们在小节点中也可以动态地将每个类作为一个插件进行加载，它具有独立的命名空间。每个被加载的类都可以充当单独的节点，这些节点位于一个名为小节点的独立进程中。

当节点之间传输的数据量非常大时，我们可以使用小节点。例如：从3D传感器或者从相机传输数据时。

接下来，我们将学习如何创建一个小节点。

创建一个小节点

在本节中，我们将创建一个简单的小节点，它可以订阅一个名为/msg_in的字符串话题，并在/msg_out话题上发布相同的字符串（std_msgs/String）。

第1步：为小节点创建软件包

我们可以使用下面的命令创建一个名为nodelet_hello_world的软件包：

```
$ catkin_create_pkg nodelet_hello_world nodelet roscpp std_msgs
```

另外，我们也可以使用nodelet_hello_world文件夹中现有的软件包，或者从下面的链接下载：

```
$ git clone https://github.com/jocacace/nodelet_hello_world
```

这里，这个软件包的主要依赖是nodelet软件包，它提供了用于构建ROS小节点的API函数。

第2步：创建hello_world.cpp小节点

现在，我们将编写小节点代码。首先在软件包中创建一个名为src的文件夹，并创建一个名为hello_world.cpp的源码文件。

你可以从nodelet_hello_world/src文件夹中获取现成的代码。

第3步：hello_world.cpp的解析

下面就是代码的解析：

```
#include <pluginlib/class_list_macros.h>
#include <nodelet/nodelet.h>
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <stdio.h>
```

这些是代码的头文件。我们应该包含class_list_macros.h和nodelet.h来访问pluginlib API和小节点API。

```
namespace nodelet_hello_world
{
    class Hello : public nodelet::Nodelet
    {
```

在上面的代码中，我们创建了一个名为Hello的小节点类，它继承了一个标准nodelet基类。所有的小节点类都应该从nodelet基类继承，并可以使用pluginlib来动态加载。在这里，Hello类将用于动态加载：

```
virtual void onInit()
{
    ros::NodeHandle& private_nh = getPrivateNodeHandle();
    NODELET_DEBUG("Initialized the Nodelet");
    pub = private_nh.advertise<std_msgs::String>("msg_out", 5);
    sub = private_nh.subscribe("msg_in", 5, &Hello::callback, this);
}
```

这是小节点的初始化函数。该函数应该是非阻塞的或者不应该处理非常重要的任务。在该函数内部，我们分别在msg_out和msg_in话题创建了节点句柄对象、话题发布者和订阅者。当执行小节点时，有一些宏可以打印调试信息。在这里，我们使用NODELET_DEBUG宏在控制台中打印调试信息。订阅者使用一个名为callback()的回调函数，该函数位于Hello类中。

```
void callback(const std_msgs::StringConstPtr input)
{
    std_msgs::String output;
    output.data = input->data;
    NODELET_DEBUG("Message data = %s", output.data.c_str());
    ROS_INFO("Message data = %s", output.data.c_str());
    pub.publish(output);
}
```

在callback()函数中，它将打印来自/msg_in话题的消息，并发布在/msg_out话题上。

```
PLUGINLIB_EXPORT_CLASS(nodelet_hello_world::Hello, nodelet::Nodelet);
```

在这里，我们将Hello作为动态加载的插件导出。

第4步：创建插件描述文件

与pluginlib示例一样，我们必须在nodelet_hello_world软件包中创建一个插件描述文件。该插件描述文件（hello_world.xml）如下所示：

```
<library path="libnodelet_hello_world">
  <class name="nodelet_hello_world/Hello" type="nodelet_hello_world::Hello"
    base_class_type="nodelet::Nodelet">
    <description>
      A node to republish a message
    </description>
  </class>
</library>
```

第5步：在package.xml文件中添加导出标签

我们需要在package.xml文件中添加导出（export）标签，并添加编译和运行的依赖项：

```
<export>
  <nodelet plugin="${prefix}/hello_world.xml"/>
</export>

<build_depend>nodelet_hello_world</build_depend>
<run_depend>nodelet_hello_world</run_depend>
```

第6步：编辑CMakeLists.txt

我们需要在CMakeLists.txt文件中添加补充代码，以编译小节点软件包。下面是补充的代码。你可以从已有的软件包中获取完整的CMakeLists.txt文件：

```
## Declare a cpp library
add_library(nodelet_hello_world
    src/hello_world.cpp
)

## Specify libraries to link a library or executable target against
target_link_libraries(nodelet_hello_world
    ${catkin_LIBRARIES}
)
```

第7步：编译并运行小节点

执行完以上步骤之后，我们可以使用catkin_make来编译软件包，如果编译成功的话，我们就可以生成共享对象文件libnodelet_hello_world.so，用于表示插件。

运行小节点的第一步就是启动小节点管理器。小节点管理器是一个C++可执行程序，它监听ROS服务并动态加载小节点。我们可以运行独立的管理器，也可以将其嵌入运行的节点中。

下面的命令将启动一个小节点管理器：

启动roscore：

```
$ roscore
```

使用下面的命令启动小节点管理器：

```
$ rosrun nodelet nodelet manager __name:=nodelet_manager
```

如果小节点管理器运行成功的话，我们将收到如图7-4所示的一条消息。

```
jcacace@robot:~$ rosrun nodelet nodelet manager __name:=nodelet_manager  
[ INFO] [1506775149.019457792]: Initializing nodelet with 2 worker threads.
```

图7-4 运行小节点管理器

启动了小节点管理器之后，我们可以使用下面的命令启动小节点：

```
$ rosrun nodelet nodelet load nodelet_hello_world/Hello nodelet_manager  
__name:=nodelet1
```

运行完上述命令，小节点联系管理器实例化一个nodelet_hello_world/Hello小节点实例，将其命名nodelet1。图7-5所示截图展示了我们加载小节点时的消息。

```
jcacace@robot:~/catkin_ws$ rosrun nodelet nodelet load nodelet_hello_world/Hello  
[ INFO] [1506776968.889742876]: Loading nodelet /nodelet1 of type nodelet_hello_
```

图7-5 运行小节点

运行了小节点之后，生成的话题列表如图7-6所示。

```
jcacace@robot:~$ rostopic list  
/nodelet1/msg_in  
/nodelet1/msg_out  
/nodelet_manager/bond  
/rosout  
/rosout_agg
```

图7-6 小节点的话题列表

我们可以通过向/nodelet1/msg_in话题发布一个字符串，然后检查我们是否在nodelet1/msg_out中接收到相同的消息来测试该节点。

下面的命令是向/nodelet1/msg_in话题发布一个字符串，结果如图7-7所示。

```
$ rostopic pub /nodelet1/msg_in std_msgs/String "Hello"
```

```
jcacace@robot:~$ rostopic pub /nodelet1/msg_in std_msgs/  
/String "Hello"  
publishing and latching message. Press ctrl-C to termin  
ate  
● ● ● jcacace@robot:~  
jcacace@robot:~$ rostopic echo /nodelet1/msg_out  
data: Hello
```

图7-7 用小节点发布和订阅

我们可以输出msg_out话题中内容来确认代码是否工作正常。

在这里，我们已经看到Hello()类的单个实例被创建为一个节点。我们可以在此小节点中以不同的节点名字创建多个Hello()类的实例。

第8步：创建小节点的启动文件

我们可以编写启动文件来加载小节点类的多个实例。下面的启动文件将加载两个小节点：test1和test2。我们可以将启动文件保存为hello_world.launch：

```
<launch>

<!-- Started nodelet manager -->

<node pkg="nodelet" type="nodelet" name="standalone_nodelet"
args="manager" output="screen"/>

<!-- Starting first nodelet -->

<node pkg="nodelet" type="nodelet" name="test1" args="load
nodelet_hello_world/Hello standalone_nodelet" output="screen">
</node>

<!-- Starting second nodelet -->

<node pkg="nodelet" type="nodelet" name="test2" args="load
nodelet_hello_world/Hello standalone_nodelet" output="screen">
</node>

</launch>
```

可以使用下面的命令启动上面的启动文件：

```
$ rosrun nodelet_hello_world hello_world.launch
```

如果启动成功的话，将在终端中显示如图7-8所示消息。

```
[ INFO] [1506951118.603857685]: Loading nodelet /test2 of type nodelet_hello_world/Hello to manager  
standalone_nodelet with the following remappings:  
[ INFO] [1506951118.606768479]: Loading nodelet /test1 of type nodelet_hello_world/Hello to manager  
standalone_nodelet with the following remappings:  
[ INFO] [1506951118.610320371]: waitForService: Service [/standalone_nodelet/load_nodelet] has not  
been advertised, waiting...  
[ INFO] [1506951118.613444334]: waitForService: Service [/standalone_nodelet/load_nodelet] has not  
been advertised, waiting...  
[ INFO] [1506951118.627001318]: Initializing nodelet with 2 worker threads.  
[ INFO] [1506951118.632595864]: waitForService: Service [/standalone_nodelet/load_nodelet] is now a  
available.  
[ INFO] [1506951118.634985422]: waitForService: Service [/standalone_nodelet/load_nodelet] is now a  
available.
```

图7-8 启动Hello()类的多个实例

这里显示的就是话题和节点的列表。我们可以看到实例化了两个小节点，我们也可以看到它们的话题，如图7-9所示。

这些话题是由Hello()类的多个实例生成的。我们可以使用rqt_graph工具查看这些小节点之间的内在联系。打开rqt_gui的命令如下：

```
jcacace@robot:~$ rostopic list  
/rosout  
/rosout_agg  
/standalone_nodelet/bond  
/test1/msg_in  
/test1/msg_out  
/test2/msg_in  
/test2/msg_out  
jcacace@robot:~$ rosnode list  
/rosout  
/standalone_nodelet  
/test1  
/test2
```

图7-9 由Hello()类的多个实例生成的话题

```
$ rosrun rqt_gui rqt_gui
```

我们可以通过下面的选项Plugins | Introspection | Node Graph来加载Node Graph插件，如图7-10所示。

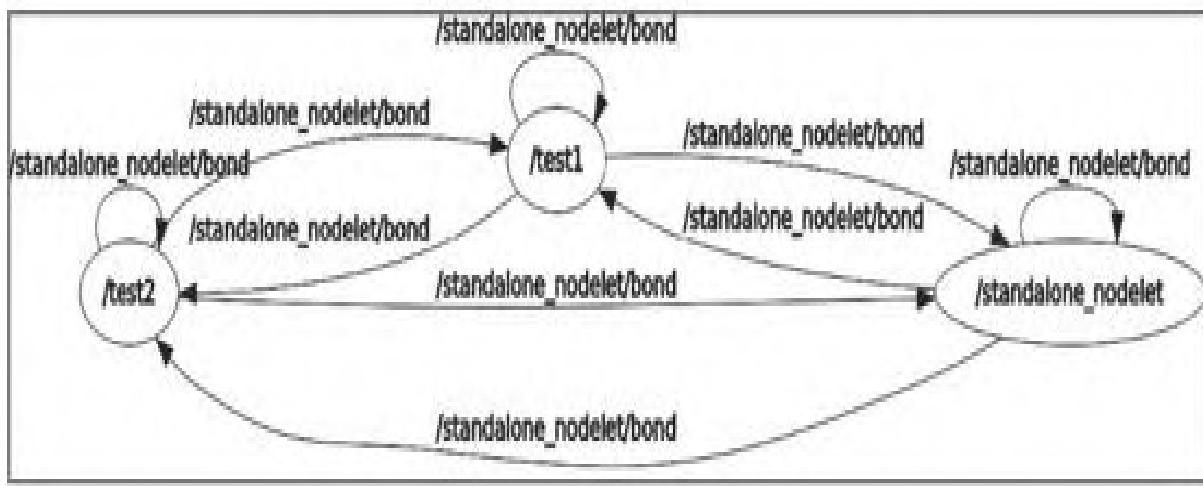


图7-10 小节点的双节点示例

或者你也可以直接加载rqt_node插件：

```
$ rosrun rqt_graph rqt_graph
```

7.3 理解Gazebo插件

Gazebo插件可以帮助我们控制机器人模型、传感器、地图环境属性，甚至Gazebo的运行方式。与pluginlib和小节点一样，Gazebo插件是一组C++代码，它可以从Gazebo仿真器中动态加载或卸载。

使用插件，我们就可以访问Gazebo的所有组件。插件独立于ROS系统，因此它可以分享给那些不使用ROS系统的人们。我们可以将插件大致分类为：

- world插件（world plugin）：使用world插件，我们可以控制Gazebo中特定的地图属性。我们还可以使用该插件更改物理引擎、光线和其他世界属性。
- 模型插件（model plugin）：这个模型插件附加在Gazebo的特定模型上，并且可以控制其属性。使用该插件还可以控制一些参数，例如模型的关节状态，关节的控制等。
- 传感器插件（sensor plugin）：传感器插件用于在Gazebo中为传感器建模，例如相机、IMU等。
- 系统插件（system plugin）：系统插件随Gazebo启动时一起启动。用户可以使用此插件控制Gazebo中与系统相关的功能。
- 可视插件（visual plugin）：可以使用可视插件访问和控制任何Gazebo组件的可视属性。

使用Gazebo插件进行开发之前，我们可能需要安装一些软件包。在ROS Kinetic中需要安装的Gazebo版本是7.0，因此你可能需要在

Ubuntu中使用如下命令安装开发包：

```
$ sudo apt-get install libgazebo7-dev
```

Gazebo插件是独立于ROS系统的，因此我们不需要ROS库来编译生成插件。

创建一个简单的world插件

本节我们将讨论一个简单的Gazebo World插件，并尝试在Gazebo中构建和加载它。

在任意一个文件夹中创建一个名为gazebo_basic_world_plugin的文件夹，然后创建一个名为hello_world.cc的CPP文件：

```
$ mkdir gazebo_basic_world_plugin && cd gazebo_basic_world_plugin  
$ nano hello_world.cc
```

Hello_world.cc的定义如下：

```
//Gazebo header for getting core gazebo functions
#include <gazebo/gazebo.hh>

//All gazebo plugins should have gazebo namespace

namespace gazebo
{

    //The custom WorldpluginTutorials is inheriting from standard
    worldPlugin. Each world plugin has to inheriting from standard plugin type.

    class WorldPluginTutorial : public WorldPlugin
    {

        public: WorldPluginTutorial() : WorldPlugin()
        {
            printf("Hello World!\n");
        }

        //The Load function can receive the SDF elements
        public: void Load(physics::WorldPtr _world, sdf::ElementPtr _sdf)
        {

    };

    //Registering World Plugin with Simulator
    GZ_REGISTER_WORLD_PLUGIN(WorldPluginTutorial)
}
```

在上述代码中使用的头文件是<gazebo/gazebo.hh>。该头文件包含了Gazebo中的核心函数。其他头文件如下所示：

- gazebo/physics/physics.hh：这是用于访问物理引擎参数的Gazebo头文件。
- gazebo/rendering/rendering.hh：这是用于处理渲染参数的Gazebo头文件。
- gazebo/sensors/sensors.hh：这是处理传感器的头文件。

在代码的最后，我们必须使用下面的语句导出插件。

GZ_REGISTER_WORLD_PLUGIN(WorldPluginTutorial) 宏会注册并将插件导出一个world插件。以下宏可用于注册传感器、模型等：

- GZ_REGISTER_MODEL_PLUGIN：这是Gazebo机器人模型的导出宏
- GZ_REGISTER_SENSOR_PLUGIN：这是Gazebo传感器模型的导出宏
- GZ_REGISTER_SYSTEM_PLUGIN：这是Gazebo系统的导出宏
- GZ_REGISTER_VISUAL_PLUGIN：这是Gazebo可视化的导出宏

设置完代码之后，我们可以编辑CMakeLists.txt来编译源代码。下面是CMakeLists.txt文件的内容：

```
$ nano gazebo_basic_world_plugin/CMakeLists.txt

cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

set(CMAKE_CXX_FLAGS "-std=c++0x ${CMAKE_CXX_FLAGS}")

find_package(Boost REQUIRED COMPONENTS system)
include_directories(${Boost_INCLUDE_DIRS})
link_directories(${Boost_LIBRARY_DIRS})

include (FindPkgConfig)
if (PKG_CONFIG_FOUND)
  pkg_check_modules(GAZEBO gazebo)
endif()
include_directories(${GAZEBO_INCLUDE_DIRS})
link_directories(${GAZEBO_LIBRARY_DIRS})

add_library(hello_world SHARED hello_world.cc)
target_link_libraries(hello_world ${GAZEBO_LIBRARIES} ${Boost_LIBRARIES})
```

创建一个build文件夹用于存储共享对象：

```
$ mkdir build && cd build
```

切换到build文件夹，执行以下命令来编译源代码：

```
$ cmake ../
$ make
```

编译好代码之后，我们将得到一个名为libhello_world.so的共享对象文件。我们必须将此共享对象的路径导出至GAZEBO_PLUGIN_PATH中，并将其添加到.bashrc文件中：

```
export  
GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:/path/to/gazebo_basic_world_plugin  
/build
```

设置完Gazebo插件路径并重新加载bashrc文件后，我们就可以在URDF或SRDF文件中使用它了。下面是一个名为hello.world的示例world文件，其中包含了以下插件：

```
$ nano gazebo_basic_world_plugin/hello.world  
  
<?xml version="1.0"?>  
<sdf version="1.4">  
  <world name="default">  
    <plugin name="hello_world" filename="libhello_world.so"/>  
  </world>  
</sdf>
```

该文件同样也位于本书提供的代码以及下面的Git代码仓库中：

```
$ git clone https://github.com/jocacace/gazebo_basic_world_plugin
```

运行Gazebo服务器并加载该示例world文件：

```
$ cd gazebo_basic_world_plugin  
$ gzserver hello.world --verbose
```

```
[cacace@robot:~/catkin_ws/src/MASTERING ROS/ch6/gazebo_basic_world_plugins$ gzserver hello.world --verbose
Gazebo multi-robot simulator, version 7.0.0
Copyright (C) 2012-2016 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org

[Msg] Waiting for master.
[Msg] Connected to gazebo master @ http://127.0.0.1:11345
[Msg] Publicized address: 10.0.2.15
Hello World!
```

图7-11 Gazebo中world插件打印“hello world!”

图7-11展示了Gazebo world插件打印出的“Hello World!”。我们将从Gazebo仓库中查看到不同的Gazebo插件代码。

我们可以查看<https://bitbucket.org/osrf/gazebo>网页，并浏览源代码，如图7-12所示，打开examples文件夹，再打开plugins文件夹。

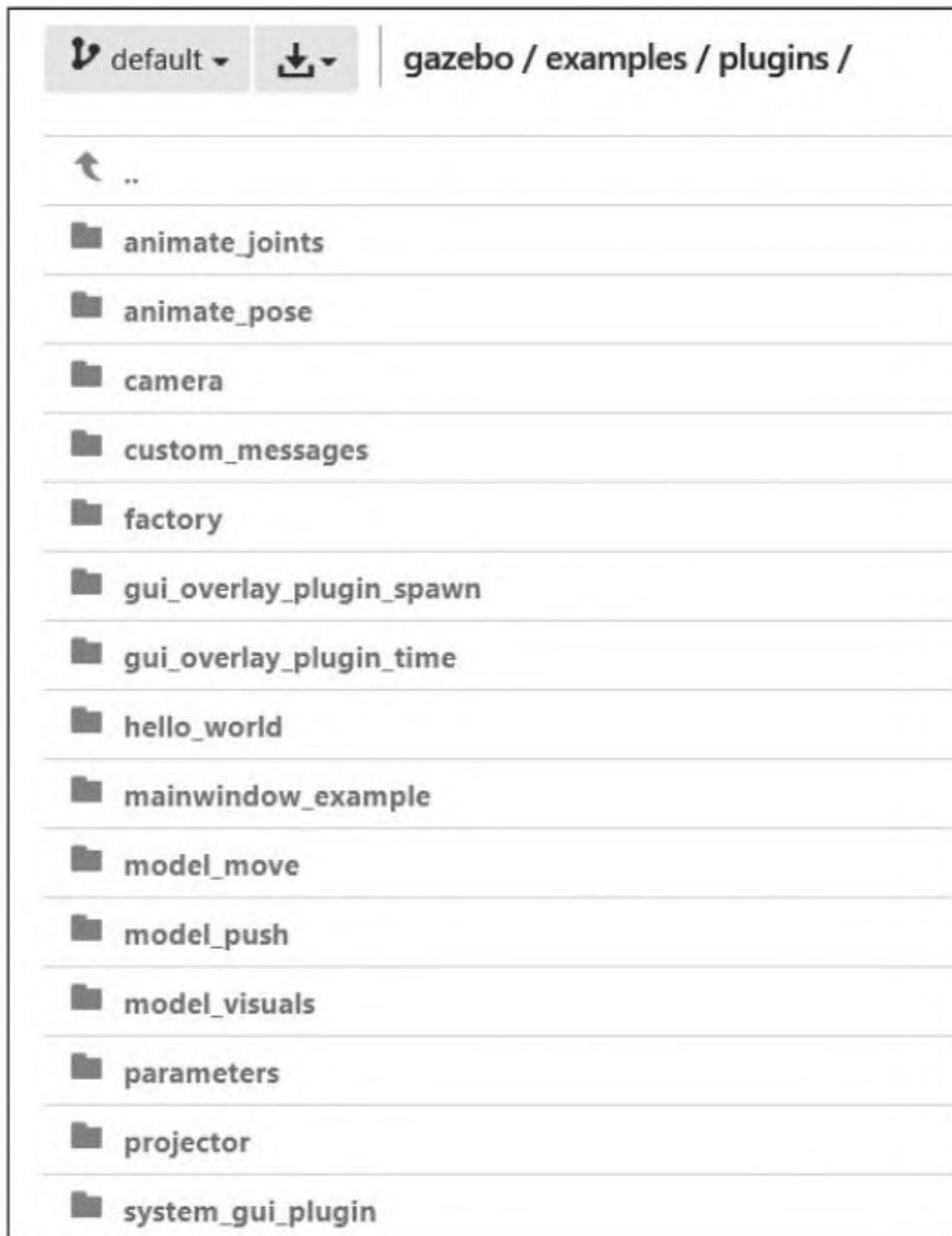


图7-12 仓库中的Gazebo插件列表

7.4 习题

- 什么是pluginlib，它的主要应用是什么？
- 小节点的主要应用是什么？
- 有哪些不同类型的Gazebo插件？
- Gazebo中模型插件的作用是什么？

7.5 本章小结

在本章，我们介绍了一些高级概念，例如pluginlib、小节点和Gazebo插件，它们都可用于向复杂的ROS应用添加更多的功能。我们讨论了pluginlib的基础知识，学习了一个使用示例。介绍pluginlib之后，我们又讨论了ROS小节点。ROS小节点广泛用于高性能的应用程序中。此外，我们还研究了一个使用ROS小节点的示例。最后，我们讨论了Gazebo插件，它们主要用于向Gazebo仿真器添加功能。

下一章，我们将进一步讨论RViz插件和ROS控制器。

第8章

ROS控制器和可视化插件编程

在上一章，我们讨论了pluginlib、小节点和Gazebo插件。在ROS系统中pluginlib是创建插件的基础库，并且可以在小节点和Gazebo插件中使用相同的库。在本章，我们将继续学习基于pluginlib的概念，例如ROS控制器和RViz插件。我们已经在第4章使用过ROS控制器，并且多次使用过一些标准控制器，比如关节状态控制器、位置控制器和轨迹控制器等。

在本章，我们将学习如何为通用机器人编写一个基本的ROS控制器。我们还将为在前面章节中开发的7-DOF机器人实现一个所需的控制器，并将其运行在Gazebo仿真器中。RViz插件可以为RViz扩展很多功能，我们将在本章介绍如何创建一个基本的RViz插件。本章将要详细讨论的内容如下：

- 了解开发ROS控制器所需的软件包
- 配置ROS控制器的开发环境
- 理解ros_control软件包
- 编写并运行一个基本的ROS控制器
- 编写并运行一个RViz插件

接下来让我们学习如何开发一个ROS控制器。第一步要了解构建自定义控制器所需的依赖包。

用于开发所有机器人通用控制器的主要软件包集都包含在ros_control中。这是pr2_mechanism的一个重写版本，包括一些有用的库，这些库用于为使用旧ROS系统的PR2机器人 (<https://www.willowgarage.com/pages/pr2/overview>) 编写底层控制器。在ROS Kinetic中，pr2_mechanism已经被ros_control软件包集取代了。下面是一些有用的软件包的简介，有助于我们编写机器人控制器：

- ros_control：该软件包可以把关节状态数据（直接从机器人的执行机构获得）和想要的设置点作为输入，然后生成输出并发送至马达。这些输出信息通常表示为关节位置、速度或力等。
- controller_manager：控制器管理器可以加载和管理多个控制器，也可以在实时兼容的循环中工作。

- controller_interface：这是控制器基类包，所有自定义控制器都应从该包继承控制器基类。如果控制管理器继承于该软件包，它将仅加载控制器。
- hardware_interface：该软件包是控制器和机器人硬件之间的接口。使用这个接口，控制器可以直接访问硬件组件。
- joint_limits_interface：该软件包允许我们设置关节限值，以安全地使用机器人。关节限值也可以在机器人的URDF中设置，而该软件包不同于URDF，因为它允许我们另外指定加速度和抖动限值。除此之外，也可以使用该软件包覆盖在URDF模型中包含的位置、速度和力等信息。发送给硬件的指令都会依据指定的关节限值进行过滤。
- realtime_tools：操作系统支持实时操作，该软件包会包含一组可以从硬实时线程使用的工具。这些工具目前只提供了实时发布者，可以向ROS话题实时发布消息。

因为我们已经在第4章使用过ros_control，所以我们的系统应该已经安装了所有需要的软件包。如果没有安装，要运行该软件包，我们应该先从Ubuntu/Debian库中安装以下ROS软件包：

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

编写ROS控制器之前，最好先了解一下ros_control软件包集中每个软件包的使用方法。

8.1 理解ros_control软件包集

ros_control软件包集包含用于编写ROS底层控制器的软件包。我们要讨论的第一个软件包是控制器接口软件包。

8.1.1 controller_interface软件包

如果我们想要实现基本的ROS底层控制器就必须从该软件包继承一个名为**controller_interface::Controller**的基类。该基类包含四个基本函数：`init()`、`start()`、`update()`、`stop()`。`Controller`类的基本结构如下：

```
namespace controller_interface
{
    class Controller
    {
        public:
            virtual bool init(hardware_interface *robotHW,
                               ros::NodeHandle &nh);
            virtual void starting();
            virtual void update();
            virtual void stopping();
    };
}
```

该控制器类的工作流程如图8-1所示。

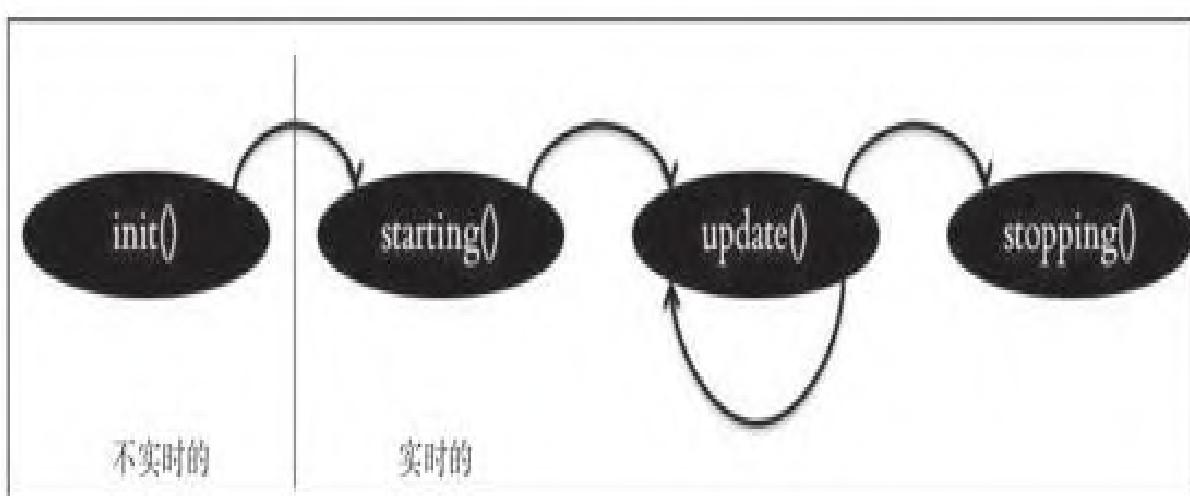


图8-1 ROS底层控制器类的工作流程

初始化控制器

加载控制器时执行的第一个函数是init()。init()函数不会启动运行控制器，它只是将控制器初始化。在启动控制器之前，初始化的过程可能需要花费一些时间。init函数的声明如下：

```
virtual bool init(hardware_interface *robotHW, ros::NodeHandle &nh);
```

函数的参数如下：

- hardware_interface *robotHW：该变量表示控制器用于开发特定的硬件接口。ROS包含了一系列已经实现的硬件接口，例如：
 - 关节命令接口（作用力、转速和位置）。
 - 关节状态接口。
 - 驱动器状态接口。
- 我们甚至可以创建自己的硬件接口，在下一个示例中，我们将使用位置关节接口。
- ros::NodeHandle &nh：控制器可以读取机器人的配置信息，甚至可以使用NodeHandle来发布话题。
- 初始化函数init()仅在控制器管理器加载控制器时执行一次。如果init()函数调用失败，它将在控制器管理器中被卸载。我们还可以编写一个自定义消息，当init()函数运行出错时，可以输出该消息。

启动ROS控制器

starting()函数仅在更新或运行控制器之前执行一次。
starting()函数的声明如下：

```
virtual void starting();
```

控制器也可以在重新启动时调用starting()函数而不被卸载掉。

更新ROS控制器

update()函数是使控制器保持活动状态的最重要函数。默认情况下update()函数包含的代码将以1 000Hz的频率被执行。这意味着控制器每毫秒执行一次该函数：

```
virtual void update();
```

停止控制器

本函数将在控制器停止时被调用。stopping()函数将会作为最后一次update()调用执行，并且只执行一次。stopping()函数不会执行失败也不会返回任何值。下面是stopping()函数的声明：

```
virtual void stopping();
```

8.1.2 控制器管理器

controller_manager软件包可以加载和卸载指定的控制器。控制器管理器还能确保控制器不会设置小于或大于关节安全限制的目标值。控制器管理器还能以默认的100Hz频率在/joint_state(sensor_msgs/JointState)话题中发布关节的状态信息。图8-2展示了控制器管理器的基本工作流程。

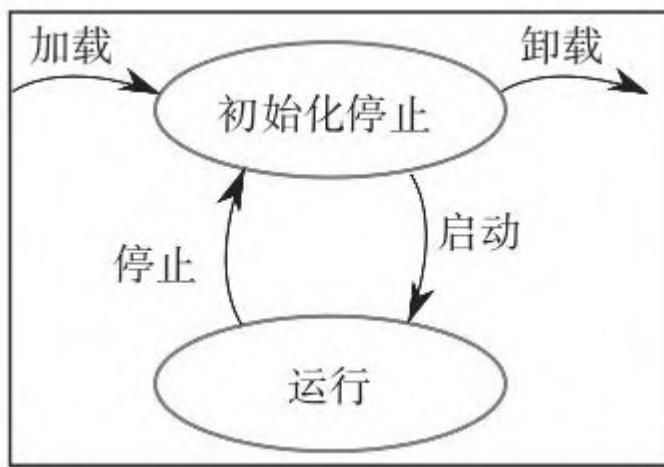


图8-2 控制器管理器的基本工作流程

控制器管理器可以加载和卸载一个插件。当控制器被控制器管理器加载时，它将首先被初始化，但它不会开始运行。

控制器被加载之后，我们可以启动或者停止控制器。当我们启动控制器时，它会运行；当我们停止控制器后，它会停止。停止并不意味着它被卸载。但是控制器管理器一旦卸载控制器，我们将无法再访问控制器。

8.2 使用ROS编写一个基本的关节控制器

首先确保已经安装了编写ROS控制器的必要软件包。而且我们已经讨论了控制器的一些基本概念。现在我们可以开始为自己创建一个控制器软件包了。

我们将开始开发一个控制器，它可以控制机器人关节，并让机器人以正弦方式移动。特别是，让7-DOF机械臂的第一个关节以正弦方式运动。

构建控制器的过程与我们之前学习的其他插件的开发过程类似。下面给出了创建ROS控制器的步骤列表：

- 1) 创建一个包含必要依赖项的ROS软件包；
- 2) 用C++编写控制器代码；
- 3) 注册或导出C++类，将其作为插件；
- 4) 在XML文件中定义插件；
- 5) 编辑CMakeLists.txt和package.xml文件，用于导出插件；
- 6) 为控制器编写配置信息；
- 7) 使用控制器管理器加载控制器。

8.2.1 第1步：创建控制器软件包

第1步就是创建一个包含必要依赖项的ROS软件包。可以用下面的命令创建一个名为mycontroller的控制器软件包：

```
$ catkin_create_pkg my_controller roscpp pluginlib controller_interface
```

我们可以直接从本书提供的my_controller文件夹中得到现有的软件包，或者直接从下面的git库中下载：

```
$ git clone https://github.com/jocacace/my_controller.git
```

8.2.2 第2步：创建控制器头文件

我们将从my_controller/src/文件夹中得到头文件my_controller.h。下面给出的正是my_controller.h头文件中的定义部分。如前所述，在这个头文件中，我们将实现controller_interface::Controller类中包含的函数：

```
#include <controller_interface/controller.h>
#include <hardware_interface/joint_command_interface.h>
#include <pluginlib/class_list_macros.h>

namespace my_controller_ns {

    class MyControllerClass: public
controller_interface::Controller<hardware_interface::PositionJointInterface
>
{
public:
    bool init(hardware_interface::PositionJointInterface* hw,
ros::NodeHandle &n);
    void update(const ros::Time& time, const ros::Duration& period);
    void starting(const ros::Time& time);
    void stopping(const ros::Time& time);

private:
    hardware_interface::JointHandle joint_;
    double init_pos_;
};

}
```

在上面的代码中，我们可以看到控制器类MyControllerClass，它继承了基类controller_interface::Controller。我们还可以看到

Controller类中的每个函数在我们的类中都被重载了。

8.2.3 第3步：创建控制器源文件

在软件包中创建一个名为src的文件夹，并创建一个名为my_controller_file.cpp的C++文件，该文件就是前面头文件中的类的定义。

下面给出了my_controller_file.cpp的定义，必须把它保存在src文件夹中：

```
#include "my_controller.h"
namespace my_controller_ns {
// 控制器初始化
bool MyControllerClass::init(hardware_interface::PositionJointInterface*
hw, ros::NodeHandle &n)
{
// 获取要控制的关节对象
    std::string joint_name;
    if( !nh.getParam( "joint_name", joint_name ) ) {
        ROS_ERROR("No joint_name specified");
        return false;
    }
    joint_ = hw->getHandle("shoulder_pan_joint");
    return true;
}
// 启动控制器
void MyControllerClass::starting(const ros::Time& time) {
    //Get initial position to use in the control procedure
    init_pos_ = joint_.getPosition();
}

// 控制器运行
void MyControllerClass::update(const ros::Time& time, const ros::Duration&
period)
{
// shoulder_pan_joint 关节执行正弦运动
double dpos = init_pos_ + 10 * sin(ros::Time::now().toSec());
    double cpos = joint_.getPosition();
    joint_.setCommand( -10*(cpos-dpos)); // 对选定的关节应用命令
    joint_
    //---
}

// 控制器退出
void MyControllerClass::stopping(const ros::Time& time) { }

// 注册插件 : PLUGINLIB_EXPORT_CLASS(my_namespace::MyPlugin,
base_class_namespace::PluginBaseClass)
PLUGINLIB_EXPORT_CLASS(my_controller_ns::MyControllerClass,
controller_interface::ControllerBase);
```

8.2.4 第4步：控制器源文件解析

在本节中，我们可以看到代码每个部分的解析：

```
/// 非实时控制器初始化
```

```
bool MyControllerClass::init(hardware_interface::PositionJointInterface*
hw, ros::NodeHandle &n)
{
```

上述代码为控制器init()函数的定义。该函数在控制器被控制器管理器加载的时候调用。在init()函数内部，我们创建了一个机器人状态的实例化参数(hw)和NodeHandle的实例。我们还得到了关节接口控制器的管理器。在我们的示例中，我们在my_controller.yaml文件中定义了要控制的关节，将关节名称加载到了ROS参数服务器中。该函数还将返回控制器初始化是否成功。

```
if( !nh.getParam( "joint_name", joint_name ) ) {
ROS_ERROR("No joint_name specified");
return false;
}
```

这段代码将为所需关节创建一个关节状态对象。这是hardware_interface类的一个实例化。joint_name就是我们用来依附在控制器上的所需关节。

```
/// 控制器实时启动
void MyControllerClass::starting(const ros::Time& time)
{
    init_pos_ = joint_.getPosition();
}
```

加载控制器后，下一步就是启动它。只有我们启动控制器时，这些预定义的函数才会被执行。在该函数中，它将检测关节的当前位置，并将其值存储到init_pos_变量中。

```
/// 控制器实时更新回路
void MyControllerClass::update(const ros::Time& time, const ros::Duration&
period)
{
    // shoulder_pan_joint 关节执行正弦运动
    double dpos = init_pos_ + 10 * sin(ros::Time::now().toSec());
    double cpos = joint_.getPosition();
    joint_.setCommand(-10*(cpos-dpos)); // 对选定的关节应用命令

    //---
}
```

这是控制器的update函数，它将控制关节以正弦方式持续移动。

8.2.5 第5步：创建插件描述文件

我们可以定义一个插件定义文件，下面给出了具体的代码。该插件文件以controller_plugins.xml为名被保存在软件包文件夹中：

```
<library path="lib/libmy_controller_lib">
    <class name="my_controller_ns/MyControllerClass"
          type="my_controller_ns::MyControllerClass"
          base_class_type="controller_interface::ControllerBase" />
</library>
```

8.2.6 第6步：更新package.xml文件

我们需要更新package.xml文件，配置controller_interface的plugin参数指向controller_plugins.xml文件：

```
<export>
  <controller_interface plugin="${prefix}/controller_plugins.xml" />
</export>
```

8.2.7 第7步：更新CMakeLists.txt文件

当完成所有上述操作之后，我们就可以修改软件包中的CMakeLists.txt文件：

```
## my_controller_file library
add_library(my_controller_lib src/my_controller.cpp)
target_link_libraries(my_controller_lib ${catkin_LIBRARIES})
```

8.2.8 第8步：编译控制器

修改好CMakeLists.txt后，我们就可以使用catkin_make命令编译我们的控制器。编译完成后需要使用rospack命令检查该控制器是否已被配置为插件，检查命令如下所示：

```
$ rospack plugins --attrib=plugin controller_interface
```

执行完此命令后，将列出与controller_interface相关的所有控制器。如果执行一切正确的话，输出结果可能如图8-3所示。

```
jcacace@robot:~$ rospack plugins --attrib=plugin controller_interface
my_controller /home/jcacace/catkin_ws/src/MASTERING ROS/ch7/my_controller/controller_plugins.xml
joint_trajectory_controller /opt/ros/kinetic/share/joint_trajectory_controller/ros_control_plugins.xml
position_controllers /opt/ros/kinetic/share/position_controllers/position_controllers_plugins.xml
effort_controllers /opt/ros/kinetic/share/effort_controllers/effort_controllers_plugins.xml
diff_drive_controller /opt/ros/kinetic/share/diff_drive_controller/diff_drive_controller_plugins.xml
joint_state_controller /opt/ros/kinetic/share/joint_state_controller/joint_state_plugin.xml
```

图8-3 检查控制器是否已被配置为插件的运行结果

8.2.9 第9步：编写控制器配置文件

正确安装控制器后，我们就可以对其进行配置并运行它。首先创建控制器的配置文件，该文件由控制器类型、关节名称、关节限值等组成。该配置文件被保存为YAML文件，并需要保存在软件包中。我们创建了一个名为`my_controller.yaml`的YAML文件，下面给出相应定义：

```
#File loaded during Gazebo startup
my_controller_name:
  type: my_controller_ns/MyControllerClass
  joint_name: elbow_pitch_joint
```

该文件是控制器的配置文件。具体讲，该文件包含了控制器类型和要传递给控制器的参数集，控制器类型由控制器源码编译的类名表示。在本例中，就是需要控制的关节的名称。

8.2.10 第10步：编写控制器的启动文件

用来展示该控制器工作过程的关节是seven_dof_arm机器人的shoulder_pan_joint。创建YAML文件后，我们就可以在启动文件夹中创建启动文件，该文件可以加载控制器配置文件并运行控制器。该启动文件名为my_controller.launch，具体如下所示：

```
<?xml version="1.0" ?>
<launch>
  <include file="$(find my_controller)/launch/seven_dof_arm_world.launch"
/>
  <rosparam file="$(find my_controller)/my_controller.yaml"
command="load"/>
  <node name="my_controller_spawner" pkg="controller_manager"
type="spawner" respawn="false"
  output="screen" args="my_controller_name"/>
</launch>
```

在下面的代码中，我们将解释该启动文件：

```
<launch>
  <include file="$(find my_controller)/launch/seven_dof_arm_world.launch"
/>
```

在这里，我们运行Gazebo仿真器，启动7-DOF机械臂的修改版本：

```
<rosparam file="$(find my_controller)/my_controller.yaml" command="load"/>
```

然后，加载开发的控制器。

最终，生成了控制器：

```
<node name="my_controller_spawner" pkg="controller_manager"
type="spawner" respawn="false"
output="screen" args="my_controller_name"/>
```

用这种方式，`controller_manager`将运行`args`列表中指定的控制器。在本例中，只有`my_controller_name`是通过控制器实现的`init()`、`start()`和`update()`函数执行的。

8.2.11 第11步：在Gazebo中运行控制器和7-DOF机械臂

创建了控制器启动文件之后，我们就应该在我们的机器人上进行测试了。可以使用下面的命令启动Gazebo仿真：

```
$ roslaunch my_controller my_controller.launch
```

当我们启动仿真时，与机器人关联的所有控制器也会启动。控制器的目的是控制控制器中定义的seven_dof_arm的elbow_pitch_joint关节进行运动。如果一切正常，机器人的肘部将开始以图8-4所示的正弦方式移动。

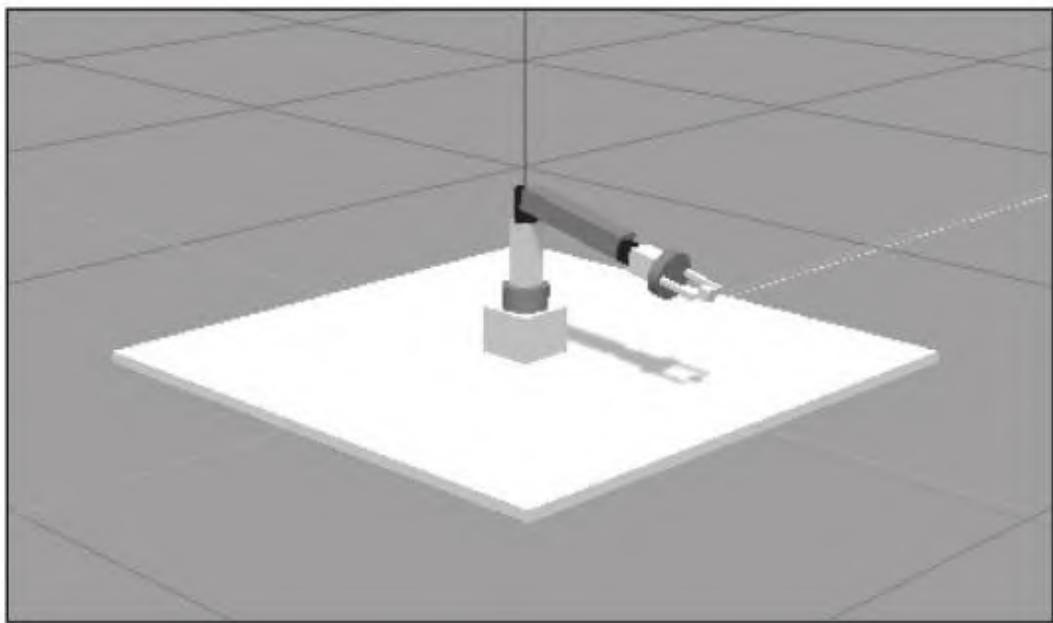


图8-4 机器人的肘部以正弦方式移动

如果还有其他控制器也在控制这个关节，我们的控制器将无法正常工作。为了避免这种情况，我们需要停止控制机器人同一关节的控制器。`controller_manager`通过发布一系列服务来管理机器人的控制器。例如，我们可以使用以下命令检查系统中加载的控制器的状态：

```
$ rosrun controller_manager controller_manager/list_controllers
```

此命令执行后的输出结果如图8-5所示。

```
jcacace@robot:~$ rosrun controller_manager controller_manager/list_controllers
controller:
  -
    name: my_controller_name
    state: running
    type: my_controller_ns/MyControllerClass
    claimed_resources:
      -
        hardware_interface: hardware_interface::PositionJointInterface
        resources: ['elbow_pitch_joint']
```

图8-5 检查系统中加载的控制器的状态

从上图中，可以看到我们的控制器（my_controller_name）正在运行。我们可以使用/controller_manager/switch_controller服务来停止它，如图8-6所示。

```
jcacace@robot:~$ rosservice call /controller_manager/switch_controller "start_controllers:  
''  
stop_controllers:  
- 'my_controller_name'  
strictness: 0"  
ok: True  
jcacace@robot:~$ rosservice call /controller_manager/list_controllers  
controller:  
-  
  name: my_controller_name  
  state: stopped  
  type: my_controller_ns/MyControllerClass  
  claimed_resources:  
  -  
    hardware_interface: hardware_interface::PositionJointInterface  
    resources: ['elbow_pitch_joint']
```

图8-6 停止控制器

在这个例子中，考虑到我们正在使用gazebo_ros_control插件运行我们的控制器。该插件表示我们的机器人在仿真场景中的硬件接口。对于真实的机器人，我们应该编写自己的硬件接口，将控制数据应用于机器人的执行机构上。

总而言之，ros_control可以为任何类型的机器人实现一组标准的通用控制器，例如effort_controllers、joint_state_controllers、position_controllers和velocity controllers。我们已经在第3章使用了ros_control中的这些ROS控制器。ros_control目前仍处于开发阶段。在这里，我们使用ros_control为7-DOF机械臂机器人开发了一个简单的专用的位置控制器。最后，还可以在ros_control的维基页面查询新控制器的信息，具体网址为：https://github.com/ros-controls/ros_control/wiki。

8.3 理解ROS可视化工具(RViz)及其插件

RViz工具是ROS的官方3D可视化工具。通过此工具可以查看来自传感器的几乎所有类型的数据。RViz会在ROS桌面完整版安装时一起被安装。让我们启动RViz并查看RViz中的基本组件。

启动roscore:

```
$ roscore
```

启动rviz:

```
$ rviz
```

RViz GUI的重要部分已在图中标记出来，每个部分的具体用法将在图8-7中展示出来。

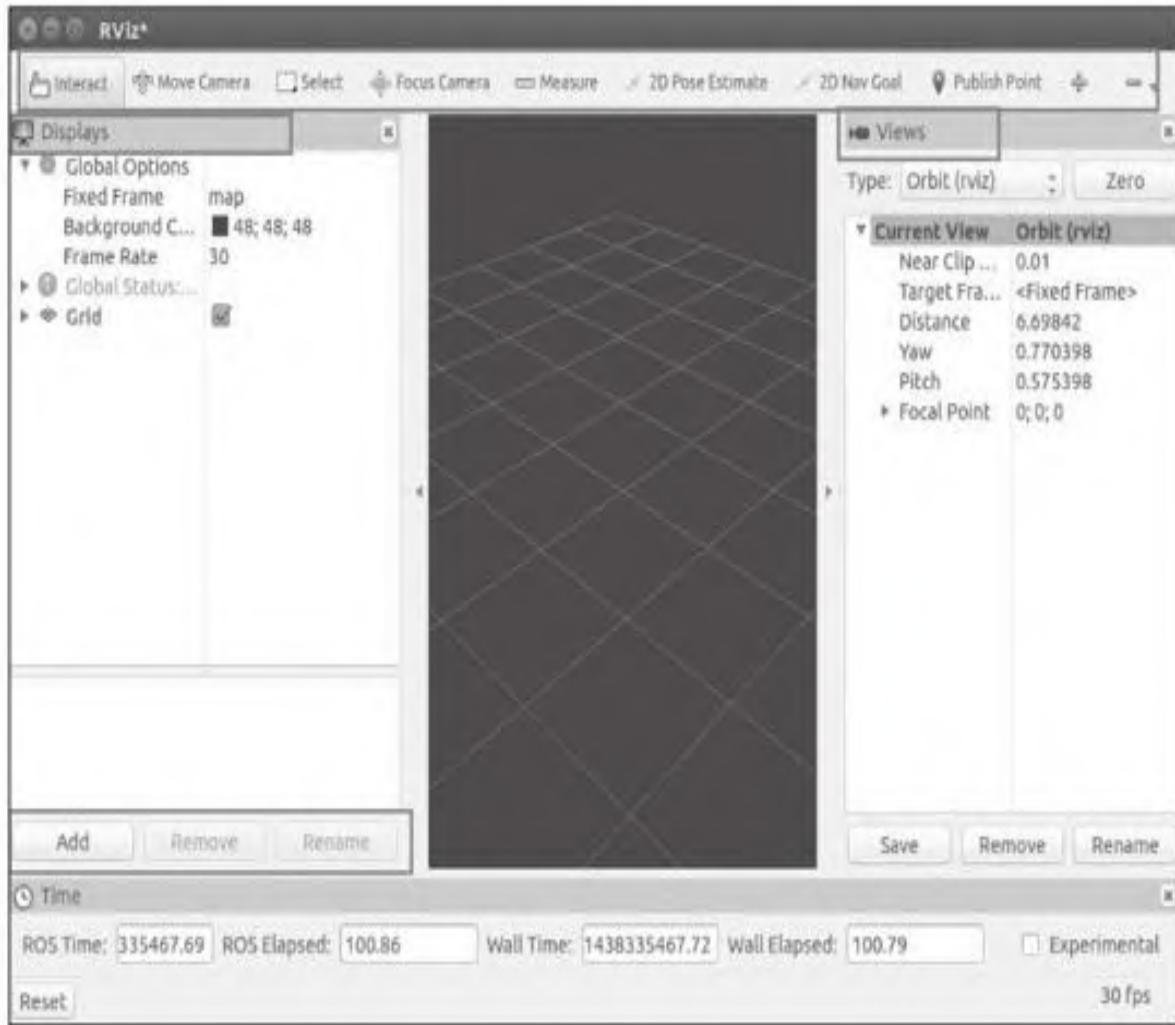


图8-7 RViz GUI各部分的具体用法

8.3.1 Displays面板

RViz的左侧面板被称作Displays面板。Displays面板包含RViz的显示插件列表及其属性。显示插件的主要用途是可视化不同类型的ROS消息，主要是RViz 3D视角中的传感器数据。在RViz中已经有很多显示插件可以查看来自相机的图像，也可以查看3D点云、激光扫描、机器人模型、tf等。我们可以通过点击左侧的Add按钮来添加所需的插件。我们也可以编写自己的显示插件并将其添加到面板上。

8.3.2 RViz工具栏

在RViz工具栏中有一组用于操作3D视图的工具。工具栏位于RViz的顶部。这些工具可以用来与机器人模型交互、调整相机视图、设置导航目标以及设置机器人2D姿态估计等。我们也可以在工具栏中以插件形式添加自定义的工具。

8.3.3 Views面板

Views面板位于RViz的右侧。通过使用Views面板，我们可以保存3D视角的不同视图，并可以通过加载已保存的配置来切换到相应的视图模式下。

8.3.4 Time面板

Time面板展示了仿真花费的时间，如果有一个仿真器与RViz一起运行的话，这将很有用。我们也可以在这个面板上重置RViz的初始设置。

8.3.5 可停靠面板

前面的工具栏和面板都属于可停靠面板。我们也可以创建自己的可停靠面板作为RViz的插件。我们将创造一个可停靠面板，上面有一个RViz插件用来遥控机器人移动。

8.4 编写用于遥控操作的RViz插件

在本章中，我们设计了一个遥控界面，如图8-8所示，我们可以在界面上手动输入遥控话题、线速度、角速度。

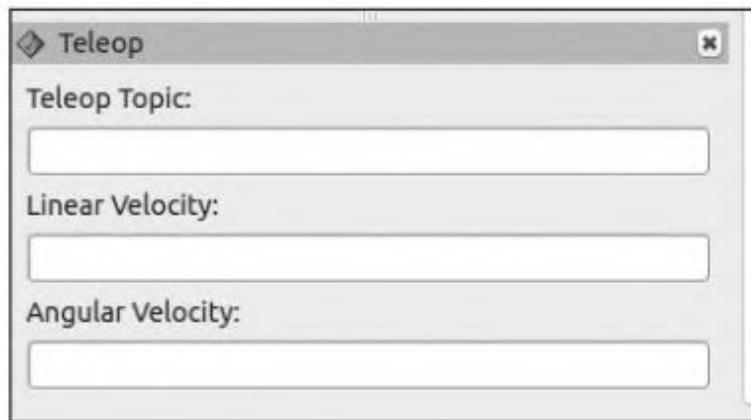


图8-8 遥控界面

下面是如何构建此插件的详细过程。

创建RViz插件的方法

在开始创建这个插件之前，我们应该知道如何创建。创建ROS插件的标准方法也适用于此插件。不同之处在于RViz插件是基于GUI的。RViz是使用名为Qt的GUI框架编写的，因此我们需要在Qt中创建GUI，并使用Qt的接口函数，这样我们就可以获取GUI值并将它们发送到ROS系统。

以下步骤描述了遥控RViz插件的工作方式：

- 1) 可停靠面板将具有Qt GUI界面，用户可以通过GUI输入遥控的话题、线速度、角速度等。
- 2) 使用Qt信号（signal）和槽（slot）机制从GUI获取用户输入，并使用ROS的订阅和发布方法来发布这些值。（Qt信号和槽机制是Qt中可用的触发器调用技术。当GUI字段生成了信号/触发器时，它就可以调用槽或函数，例如回调机制。）
- 3) 在这里，我们也可以使用相同的步骤来创建一个前面讨论过的插件。

现在我们来看看创建该插件的分步过程。

第1步：创建RViz插件软件包

让我们创建一个用于遥控插件的软件包：

```
$ catkin_create_pkg rviz_telop_commander roscpp rviz std_msgs
```

该软件包主要依赖于RViz软件包。因为RViz是用Qt库构建的，因此我们不需要在软件包中包含额外的Qt库。在Ubuntu16.04版本，我们需要使用Qt5库。

第2步：创建RViz插件头文件

在src文件夹中创建一个名为teleop_pad.h的头文件。你也可以从现有的软件包中获取源代码。该头文件包含这个插件所需的类和方法的声明。

下面是该头文件的解析：

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

上面的代码是构建这个插件所需的头文件。我们需要ROS的头文件来发布teleop话题，需要`<rviz/panel.h>`来获取RViz的基类，用于创建一个新的面板。

```
class TeleopPanel: public rviz::Panel  
{
```

这是一个插件类，它继承自`rviz::Panel`基类。

```
Q_OBJECT  
public:
```

这个类使用了Qt信号和槽机制，它是Qt中`QObject`的子类。在这种情形下，我们应该使用`Q_OBJECT`宏。

```
TeleopPanel( QWidget* parent = 0 );
```

这就是`TeleopPanel()`类的构造函数，我们正在将`QWidget`类初始化为0。我们使用`TeleopPanel`类中的`QWidget`实例实现`teleop`插件的GUI。

```
virtual void load( const rviz::Config& config );  
virtual void save( rviz::Config config ) const;
```

这些显示了如何覆盖`rviz::Panel`函数以保存和加载RViz的配置文件。

```
public Q_SLOTS:
```

在这条语句之后，我们就可以定义一些公共的Qt槽机制。

```
void setTopic( const QString& topic );
```

当我们在GUI中输入话题名称并按下回车键时，将调用这个槽并以给定的名称创建一个话题发布者。

```
protected Q_SLOTS:  
void sendVel();  
void update_Linear_Velocity();  
void update_Angular_Velocity();  
void updateTopic();
```

当我们更改现有的话题名称时，这些都是用于发送速度、更新线速度和角速度以及更改话题名称的受保护槽机制。

```
QLineEdit* output_topic_editor_;  
QLineEdit* output_topic_editor_1;  
QLineEdit* output_topic_editor_2;
```

我们正在创建Qt QLineEdit对象，这样就能在插件中创建三个文本输入框，用于接收话题名称、线速度及角速度。

```
ros::Publisher velocity_publisher_;
ros::NodeHandle nh_;
```

上面是发布者（publisher）对象，及用于发布话题和处理ROS节点的节点句柄（NodeHandle）对象。

第3步：创建RViz插件定义

在本步中，我们将创建包含插件定义的C++主文件。该文件即 teleop_pad.cpp，可以从src文件夹中获取该文件。

该文件的主要功能如下：

- 它充当Qt GUI元素的容器，例如QLineEdit来接收文本输入。
- 它使用ROS发布者发布速度指令。
- 它可以保存和恢复RViz的配置文件。

下面是该代码每个部分的解析：

```
TeleopPanel::TeleopPanel( QWidget* parent )
: rviz::Panel( parent )
, linear_velocity_( 0 )
, angular_velocity_( 0 ) {
```

这是构造函数，它用QWidget初始化rviz::Panel，并将线速度和角速度设置为0。

```
QVBoxLayout* topic_layout = new QVBoxLayout;
topic_layout->addWidget( new QLabel( "Teleop Topic:" ) );
output_topic_editor_ = new QLineEdit;
topic_layout->addWidget( output_topic_editor_ );
```

这将在面板上添加一个新的QLineEdit小工具，用于处理话题名称。与之类似，另外两个QLineEdit小工具用于处理线速度和角速度。

```
QTimer* output_timer = new QTimer( this );
```

这将创建一个Qt定时器对象，用于更新发布速度话题的函数。

```
connect( output_topic_editor_, SIGNAL( editingFinished() ), this, SLOT(
updateTopic() ) );
connect( output_topic_editor_, SIGNAL( editingFinished() ), this, SLOT(
updateTopic() ) );
connect( output_topic_editor_1, SIGNAL( editingFinished() ), this, SLOT(
update_Linear_Velocity() ) );
connect( output_topic_editor_2, SIGNAL( editingFinished() ), this, SLOT(
update_Angular_Velocity() ) );
```

这将Qt信号与槽机制相连接。在这里，当editingFinished()函数返回true时触发信号，这里的槽即函数updateTopic()。当在QtLineEdit中编辑完成并按下回车键时，这个信号就会被触发，并执行相应的槽函数。在这里，该槽函数将用插件文本框中的值来设置话题名称、角速度和线速度的值。

```
connect( output_timer, SIGNAL( timeout() ), this, SLOT( sendVel() ) );
output_timer->start( 100 );
```

当Qt定时器超时，它就会产生一个信号。该定时器每100 ms超时一次，并执行名为sendVel()的槽函数，用以发布速度话题。

我们可以在本节后看到每个槽的定义。这段代码写得非常清楚。最后，我们可以看到下面的代码，用于将其导出为插件：

```
#include <pluginlib/class_list_macros.h>
PLUGINLIB_EXPORT_CLASS(rviz_telop_commander::TeleopPanel, rviz::Panel)
```

第4步：创建插件描述文件

下面给出plugin_description.xml的定义：

```
<library path="lib/librviz_telop_commander">
  <class name="rviz_telop_commander/Teleop"
        type="rviz_telop_commander::TeleopPanel"
        base_class_type="rviz::Panel">
    <description>
      A panel widget allowing simple diff-drive style robot base control.
    </description>
  </class>
</library>
```

第5步：向package.xml文件添加导出标签

我们必须修改package.xml文件以包含插件的描述信息。下面是修改后的package.xml：

```
<export>
  <rviz plugin="${prefix}/plugin_description.xml"/>
</export>
```

第6步：编辑CMakeLists.txt

我们需要在CMakeLists.txt定义中另外添加一些代码，如下所示：

```
find_package(Qt5 COMPONENTS Core Widgets REQUIRED)
set(QT_LIBRARIES Qt5::Widgets)
catkin_package(
    LIBRARIES ${PROJECT_NAME}
    CATKIN_DEPENDS roscpp
        rviz
)

include_directories(include
    ${catkin_INCLUDE_DIRS}
    ${Boost_INCLUDE_DIRS}
)

link_directories(
    ${catkin_LIBRARY_DIRS}
    ${Boost_LIBRARY_DIRS}
)

add_definitions(-DQT_NO_KEYWORDS)

QT5_WRAP_CPP(MOC_FILES
    src/teleop_pad.h
    OPTIONS -DBOOST_TT_HAS_OPERATOR_HPP_INCLUDED -
    DBOOST_LEXICAL_CAST_INCLUDED
)

set(SOURCE_FILES
    src/teleop_pad.cpp
    ${MOC_FILES}
)
add_library(${PROJECT_NAME} ${SOURCE_FILES})
target_link_libraries(${PROJECT_NAME} ${QT_LIBRARIES}
${catkin_LIBRARIES})
```

可以从下面的Git库中下载完整的软件包：

```
$ git clone https://github.com/jocacace/rviz_teleop_commander.git
```

第7步：编译和加载插件

创建了这些文件之后，我们就可以使用catkin_make来编译生成软件包。如果编译成功的话，我们就可以在RViz中加载插件了。启动RViz后，通过点击Menu Panel | Add New Panel来加载面板，这样我们就能得到如图8-9所示的面板。



图8-9 加载面板

如果我们从列表中加载Teleop插件，我们将得到如图8-10所示的面板。

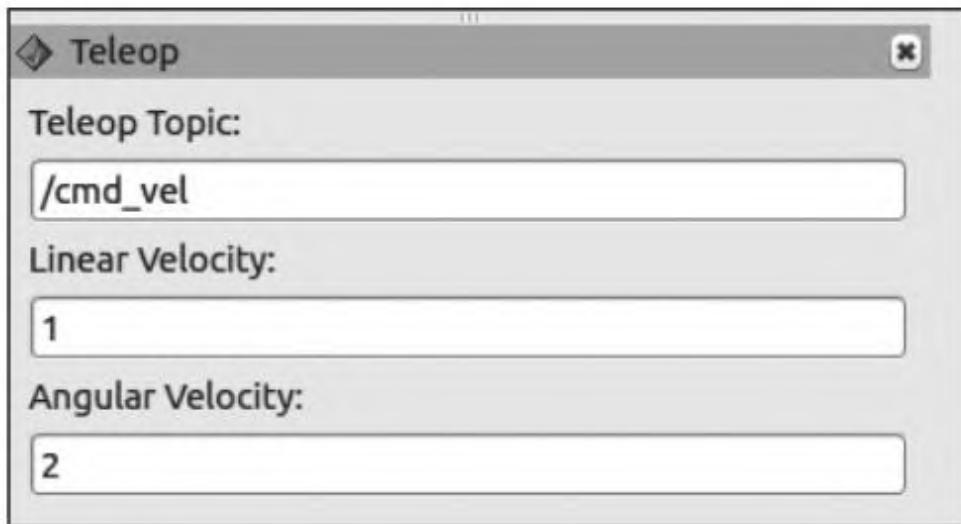


图8-10 加载Teleop插件

我们可以在Teleop Topic（话题名称）文本框中输入名称，在Linear Velocity（线速度）和Angular Velocity（角速度）文本框中设置相应的值，这样我们就能重复Teleop Topic并得到话题内容了，如图8-11所示。

```
jcacace@robot:~/catkin_ws$ rostopic echo /cmd_vel
linear:
  x: 1.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0
---
linear:
  x: 1.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0
---
linear:
  x: 1.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0
```

图8-11 话题内容

8.5 习题

- 在ROS中编写底层控制器需要哪些系列软件包?
- 在ROS控制器内部发生了哪些不同的过程?
- ros_control软件包集包含哪些主要软件包?
- RViz插件有哪些不同类型?

8.6 本章小结

在本章，我们讨论了如何为ROS可视化工具(RViZ)创建插件，并编写了基本的ROS控制器。我们曾经使用过ROS中的默认控制器，而在本章中，我们开发了一个定制化的控制器，用于控制关节运动。编辑生成并测试完控制器后，我们测试了RViz插件。在RViz中，我们生成了一个新的遥控面板，可以手动输入话题名称，还可以在面板中输入线速度和角速度。该面板可用于控制机器人，而无须启动另一个遥控节点。下一章，我们将讨论如何连接I/O开发板，以及如何在嵌入式系统中运行ROS。

第9章

将ROS与I/O开发板、传感器、执行机构连接

前几章我们讨论了ROS中的几种插件框架。本章将讨论传感器、马达驱动器等硬件组件的ROS接口。利用I/O主控板（如Arduino、树莓派、Odroid-XU4），我们将讨论如何让ROS与传感器通信，还会讨论智能驱动器（如DYNAMIXEL）与ROS间的接口。

本章将介绍以下内容：

- 理解Arduino-ROS接口。
- 安装Arduino-ROS接口软件包。
- Arduino-ROS接口实例：Chatter-Talker、LED闪烁/按钮、ADXL335加速度计、超声波测距传感器、里程计发布者等。
- 非Arduino板与ROS的接口。
- 在Odroid-XU4和树莓派2上安装/配置ROS。
- ROS与树莓派和Odroid GPIO通信。
- ROS与DYNAMIXEL驱动器接口。

9.1 理解Arduino-ROS接口

我们不妨先了解Arduino是什么？Arduino是当今市面上最流行的开源开发板之一。Arduino的巨大成功在于其易于编程、硬件性价比高。大部分Arduino开发板采用Atmel8~32位微控制器，时钟频率范围为8MHz~84MHz。Arduino可以用来快速设计机器人原型，Arduino主要作为传感器和驱动器的接口，使用UART协议与PC主机进行通信，如接收PC端发来的用户指令，并将传感器数据发送到PC主机。

市面上有不同类型的Arduino开发板，我们需要根据机器人的特点选择一款适合自己项目的Arduino开发板。我们先了解一下现有的面向初级、中级和高级用户的Arduino开发板，如图9-1所示。

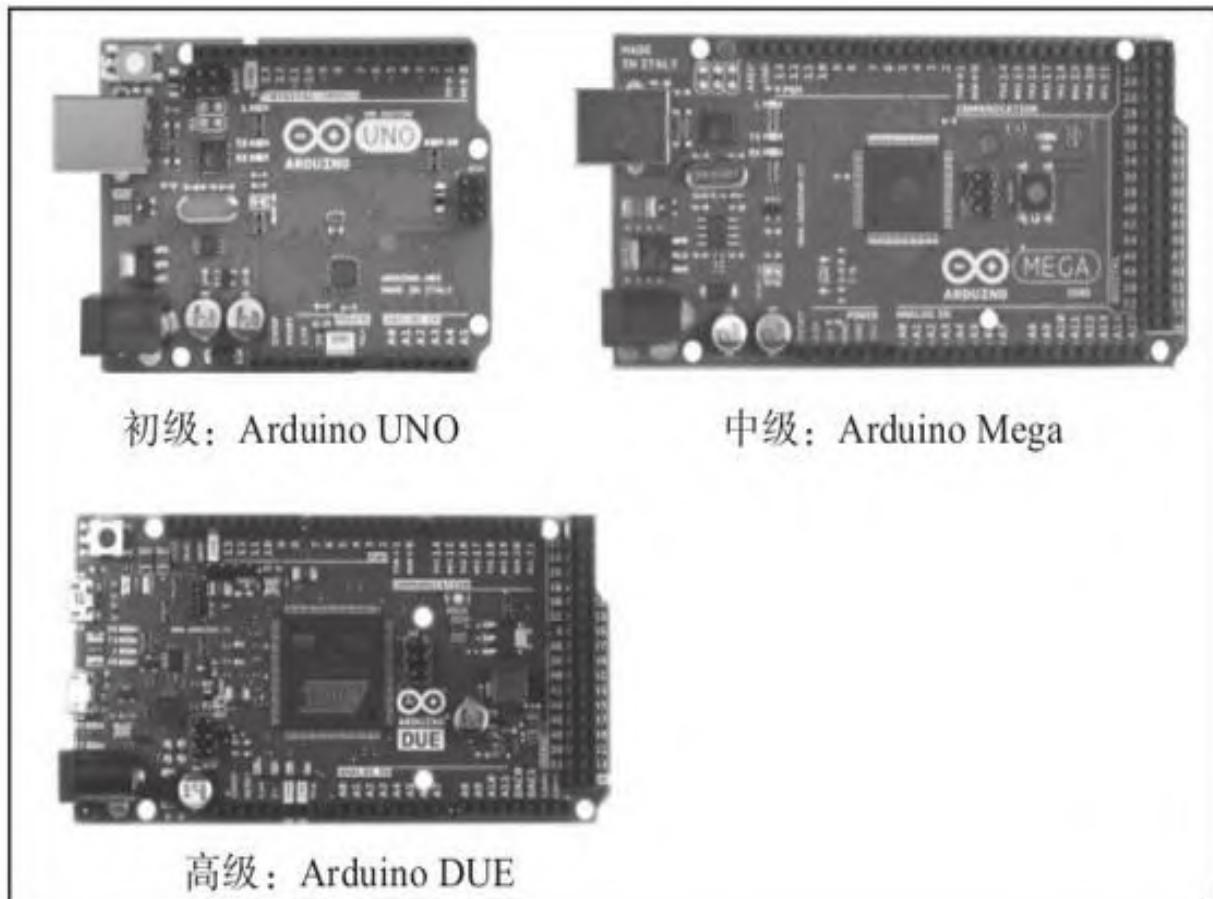


图9-1 不同版本的Arduino开发板

在下面的表格中，我们将简要介绍每个Arduino开发板的规格及其可以在哪里使用：

开发板	Arduino UNO	Arduino Mega 2560	Arduino Due
处理器	Atmega328P	Atmega2560	ATSAM3X8E
工作 / 输入电压	5V/7-12V	5V/7-12V	3.3V/7-12V
CPU 频率	16MHz	16MHz	84MHz
模拟输入 / 输出引脚	6/0	16/0	12/2
数字 IO/PWM 引脚	14/6	54/15	54/12
EEPROM[KB]	1	4	-
SRAM[KB]	2	8	96
闪存 [KB]	32	256	512
USB	方口	方口	2 个 Micro 口
UART	1	4	4
应用	简单机器人和传感器接口	中级机器人开发应用	高端机器人开发应用

我们来看如何将Arduino连接到ROS。

9.2 Arduino-ROS接口是什么

PC主机和机器人I/O控制板之间的通信基本都是通过UART协议实现的。当两个设备相互通信时，每个设备端都应该有程序负责为当前的设备解析串口指令。我们可以实现自己的一套逻辑，从而在PC和控制板之间接收/发送数据。因为没有标准库来实现这种通信，所以每个I/O控制板的接口代码是不同的。

Arduino-ROS接口是Arduino开发板和PC主机之间标准的通信方式。目前，这种接口是Arduino专用的。当然也可以编写自定义节点来实现与其他I/O开发板的通信。

我们可以在Arduino IDE中使用ROS的C++ API，就像在PC上编程那样，对Arduino开发板编程。有关接口软件包的详细信息将在下面介绍。

9.2.1 理解ROS中的rosserial软件包

rosserial软件包是ROS的一套标准通信协议，用于实现ROS与字符设备（如串口、套接字等）之间的通信。rosserial协议可以将标准ROS消息和服务数据类型转换为嵌入式设备上对等的数据类型，它还可以通过多路复用来自字符设备的串口数据，支持多话题通信。串口数据通过在数据中添加帧头和帧尾作为数据包来发送，数据包的结构如图9-2所示。

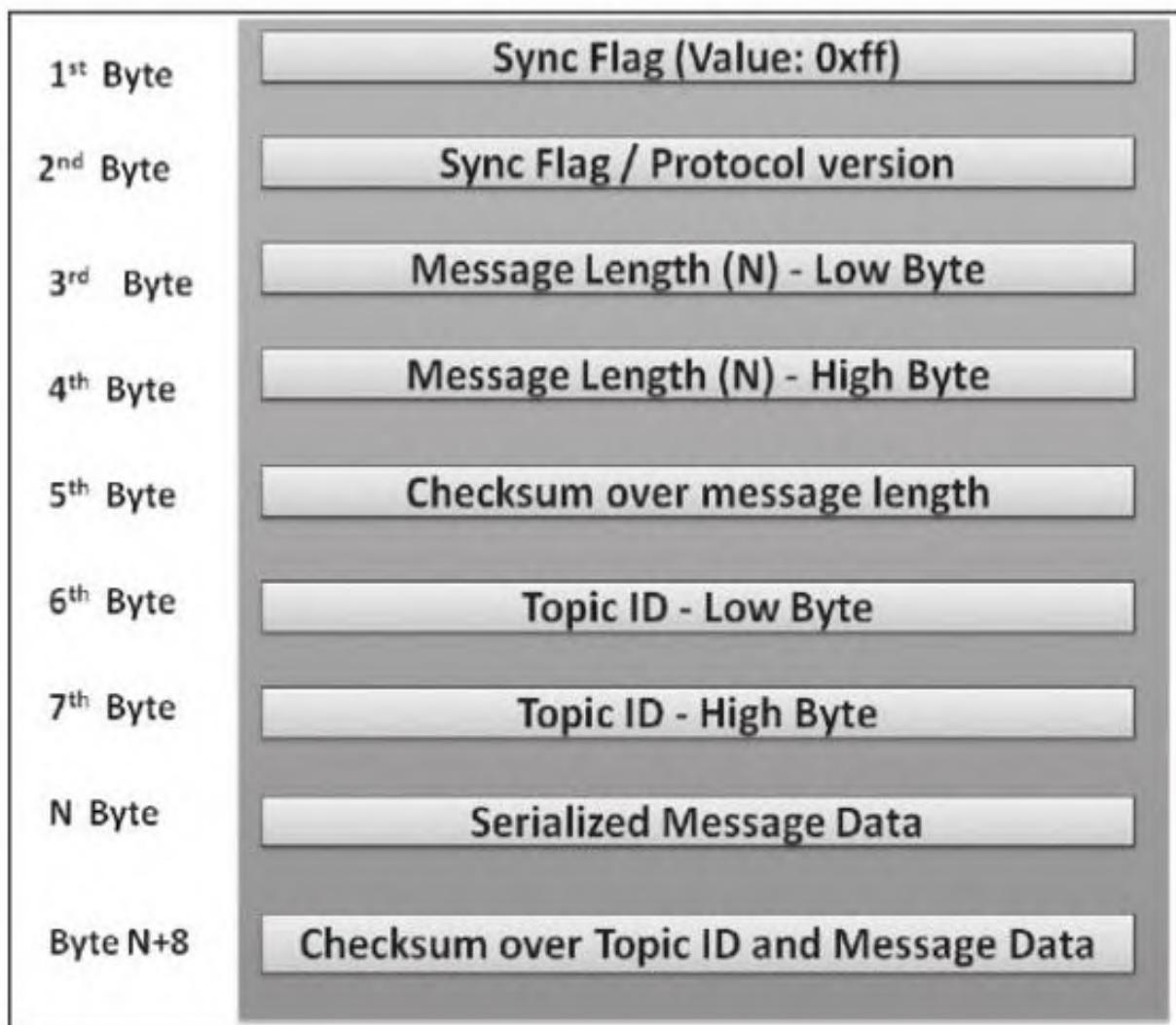


图9-2 rosserial数据包的结构

每个字节的功能如下：

- Sync Flag（同步标识）：这是软件包的第一个字节，总是0xff。
- Sync Flag（同步标识）/Protocol version（协议版本）：该字节在ROSGroovy版本中是0xff，此后修改为0xfe。
- Message Length（消息长度）：这是该数据包的总长度（两个字节）。
- Checksum over message length（消息长度的校验和）：这是消息长度的校验和，为了校验数据包是否损坏。
- Topic ID（话题ID）：这是为每个话题分配的ID，其中0~100是为系统相关的功能保留的标识。
- Serialized Message data（序列化消息数据）：这是与每个话题相关的数据。
- Checksum of Topic ID and data（话题标识和消息数据的校验和）：这是话题标识和它相应的串口数据的校验和，为了校验软件包是否损坏。

使用如下公式计算长度的校验和：

校验和=255-((话题标识低字节+话题标识高字节+...消息数据字节)%256)

利用ROS客户端库（如roscpp、 rospy和roslisp），可以开发运行于各类设备上的ROS节点。利用ROS客户端库的某端口，可以在嵌入式设备（如Arduino和基于嵌入式Linux的开发板）上运行ROS节点。这样的库就是rosserial_client。使用rosserial_client库，我们就能在Arduino、嵌入式Linux平台和Windows上开发ROS节点。下面列出了每个平台上的rosserial_client库：

- rosserial_arduino：该库适用于Arduino平台，如Arduino UNO、Leonardo、Mega，还有用于高级机器人项目中的Due系列。

- rosserial_embeddedlinux：该库支持嵌入式Linux平台，例如VEXPro、Chumby闹钟、WRT54GL路由器等。

- rosserial_windows：这是一个可以在Windows平台使用的客户端库。

在PC端，我们需要一些其他软件包来解析串口消息，并从rosserial_client客户端库中将其转换为准确的话题。下面的软件包可以帮助我们解码串口数据：

- rosserial_python：这是推荐的PC端节点，用于处理来自设备的串口数据。接收节点的代码完全用Python编写。

- rosserial_server：这是PC端rosserial的C++实现，内置的功能比rosserial_python要少，但是却能用于对性能要求较高的应用程序中。

我们重点关注运行在Arduino中的ROS节点。首先，我们将学习如何安装rosserial软件包，然后再讨论如何在Arduino IDE中安装rosserial_arduino库。

在Ubuntu 16.04上安装rosserial软件包

要在Ubuntu 16.04上使用Arduino，我们必须安装rosserial ROS软件包，然后配置Arduino客户端，安装与ROS通信所需的库。我们可以使用以下命令在Ubuntu上安装rosserial软件包：

1) 使用apt-get安装rosserial二进制软件包：

```
$ sudo apt-get install ros-kinetic-rosserial-arduino ros-kinetic-rosserial-embeddedlinux ros-kinetic-rosserial-windows ros-kinetic-rosserial-server ros-kinetic-rosserial-python
```

2) 要在Arduino中安装rosserial_client客户端库（即ros_lib库），我们必须下载最新的Arduino IDE（32/64位Linux版本）。

Arduino IDE下载链接为：

<https://www.arduino.cc/en/main/software>。在这里，我们将下载Linux 64位版本并将Arduino IDE文件夹复制到Ubuntu桌面。为了运行

Arduino，还需要Java执行环境的支持。如果未安装Java执行环境，使用如下命令来安装：

```
$ sudo apt-get install java-common
```

3) 安装好Java执行环境后，使用如下命令切换到arduino文件夹：

```
$ cd ~/Desktop/arduino-1.8.5
```

4) 使用如下命令启动Arduino：

```
$ ./arduino
```

图9-3显示的是Arduino IDE窗口。

sketch_oct21a | Arduino 1.8.5

File Edit Sketch Tools Help

sketch_oct21a

```
void setup() {
  // put your setup code here, to run once:
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

图9-3 Arduino IDE

5) 点击File|Preference来配置Arduino的sketchbook文件夹（用来存储你的程序的文件夹）。Arduino IDE将程序(sketch)存储到此位置。我们在用户主文件夹下创建了一个名为Arduino1的文件夹，并将该文件夹设置为Sketchbook文件夹所在位置。

我们可以在Arduino1文件夹中看到一个名为libraries的文件夹，使用如下命令切换到该文件夹：

```
$ cd ~/Arduino1/libraries/
```

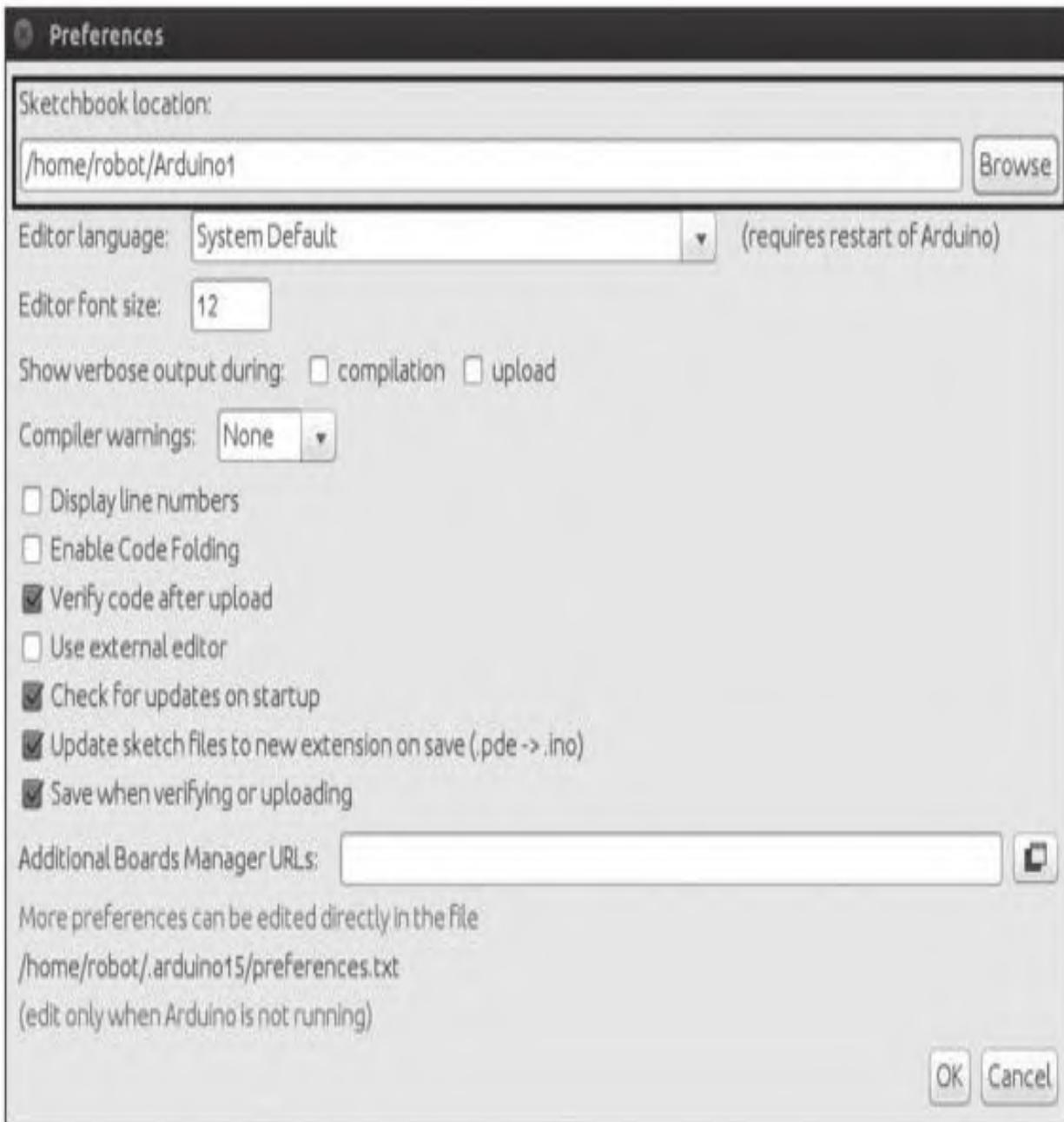


图9-4 Arduino IDE的首选项

如果没有libraries文件夹的话，可以新建一个。切换到此文件夹后，我们可以使用make_libraries.py脚本生成ros_lib，该脚本在rosserial_arduino软件包中。ros_lib是Arduino的rosserial_client库，它在Arduino IDE环境中提供ROS客户端API：

```
$ rosrun rosserial_arduino make_libraries.py
```

rosserial_arduino是arduino的ROS客户端，可以使用UART进行通信，也可以像ROS节点那样，发布话题、服务、tf等。

make_libraries.py脚本可以生成ROS消息和服务的封装，并针对Arduino数据类型进行优化。然后将ROS消息和服务转换为Arduino C/C++的等价代码，如下所示：

- 转换ROS消息：

```
ros_package_name/msg/Test.msg --> ros_package_name::Test
```

- 转换ROS服务：

```
ros_package_name/srv/Foo.srv --> ros_package_name::Foo
```

例如，如果包含头文件#include <std_msgs/UInt16.h>，我们就可以实例化std_msgs::UInt16类型的数字了。

如果make_libraries.py脚本运行正常，可以在libraries文件夹中生成一个ros_lib文件夹。重启Arduino IDE后，我们可以看到如图9-5所示的ros_lib例子。

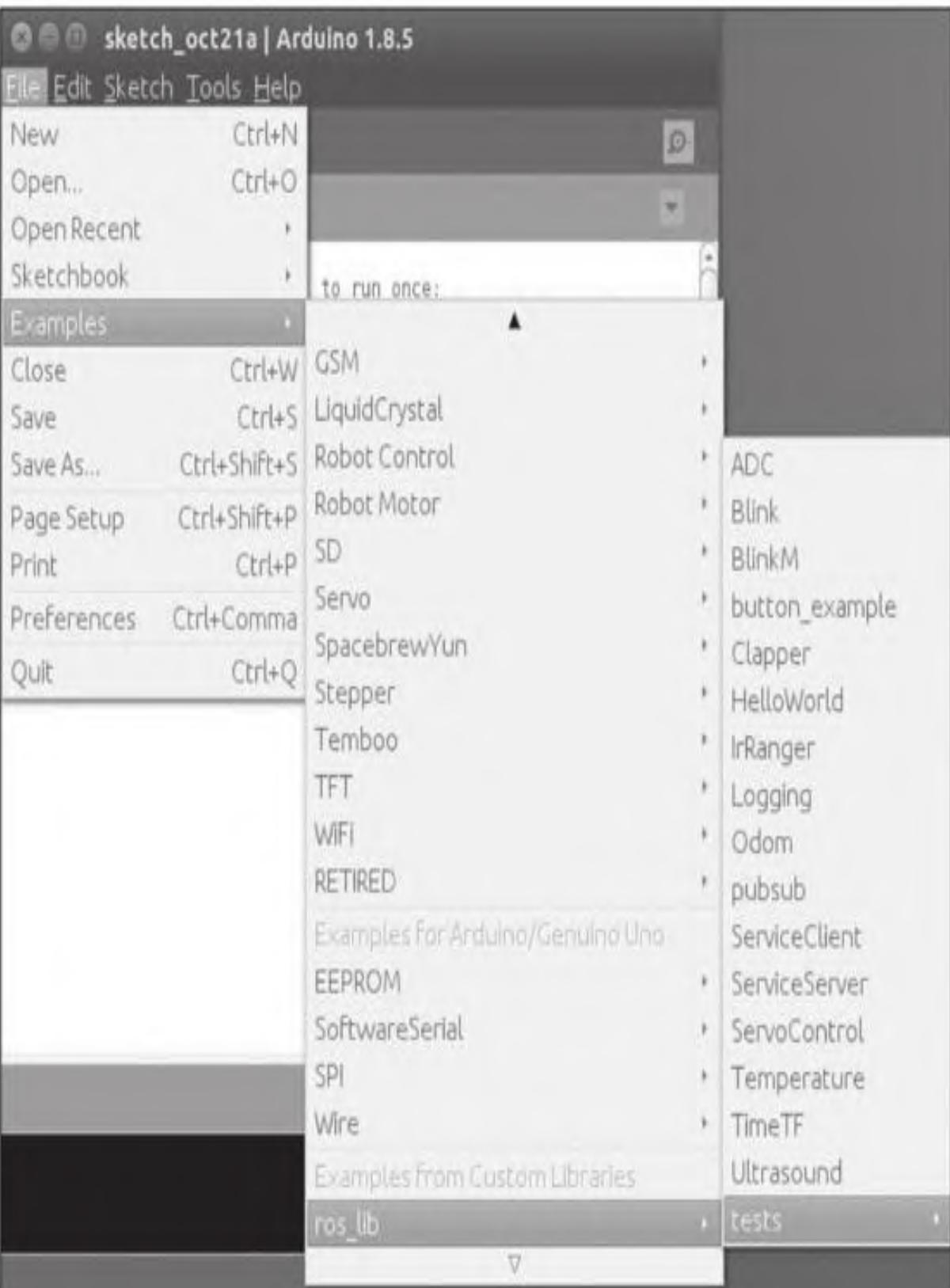


图9-5 Arduino下ros_lib示例列表

我们可以测试其中任意的示例程序，确保它可以正常编译，这样就能确认ros_lib APIs工作正常。下面来讨论编译ROS Arduino节点所必需的APIs。

9.2.2 理解Arduino中的ROS节点API

下面是ROS Arduino节点的基本结构，以及每行代码的功能：

```
#include <ros.h>

ros::NodeHandle nh;

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

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

使用如下代码在Arduino中创建NodeHandle：

```
ros::NodeHandle nh;
```

注意，应该在setup()函数之前声明Nodehandle，从而保证变量nh具有全局作用域。实例节点的初始化则在setup()函数中完成：

```
nh.initNode();
```

设备启动时，Arduino中的setup()函数只执行一次。注意，只能在串口设备上创建单个节点。

在loop()函数中，需使用如下代码执行一次ROS回调：

```
nh.spinOnce();
```

像其他ROS客户端库一样，可以在Arduino中创建订阅者（Subscriber）和发布者（Publisher）对象。下面是定义订阅者和发布者的过程。

以下代码展示如何在Arduino中定义一个订阅者对象：

```
ros::Subscriber<std_msgs::String> sub("talker", callback);
```

在此，我们定义了一个订阅者，并订阅了一个String类型的消
息。当一个String类型的消息到达talker话题时，就会运行
callback()回调函数。下面就是一个处理String类型数据的回调函数
实例：

```
std_msgs::String str_msg;

ros::Publisher chatter("chatter", &str_msg);

void callback ( const std_msgs::String& msg){
    str_msg.data = msg.data;

    chatter.publish( &str_msg );
}
```

注意，callback()、Subscriber和Publisher需要在setup()函数
之前定义，以确保全局作用域。在此，我们定义const
std_msgs::String &msg来接收String类型的数据。

以下代码演示如何在Arduino中定义一个发布者对象：

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

以下代码演示如何发布string类型消息：

```
chatter.publish( &str_msg );
```

定义了发布者和订阅者对象之后，需要在setup()函数内使用如下代码来初始化它们：

```
nh.advertise(chatter);
nh.subscribe(sub);
```

在Arduino中有打印日志的ROS API，下面就是支持不同类型日志打印的API：

```
nh.logdebug("Debug Statement");
nh.loginfo("Program info");
nh.logwarn("Warnings.");
nh.logerror("Errors..");
nh.logfatal("Fatalities!");
```

我们可以使用ROS的内置函数来提取Arduino中当前的ROS时间信息（如当前时间和时间间隔）。

- 获取ROS当前时间：

```
ros::Time begin = nh.now();
```

- 将ROS当前时间转换为浮点秒数：

```
double secs = nh.now().toSec();
```

- 以秒为单位创建持续时间间隔：

```
ros::Duration ten_seconds(10, 0);
```

9.2.3 ROS-Arduino发布者和订阅者实例

Arduino-ROS接口的第一个实例是chatter和talker。用户将一条String类型的消息发送至talker话题，随后，Arduino也将同样的消息发送至chatter话题。下面是在Arduino上实现的ROS节点，详细解释如下：

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

// 创建 NodeHandle 对象
ros::NodeHandle nh;

// 定义字符串类型变量
std_msgs::String str_msg;

// 定义发布者
ros::Publisher chatter("chatter", &str_msg);
// 定义回调函数
void callback ( const std_msgs::String& msg){

    str_msg.data = msg.data;
    chatter.publish( &str_msg );
}

// 定义订阅者
ros::Subscriber<std_msgs::String> sub("talker", callback);

void setup()
{
    // 初始化节点
```

```
nh.initNode();
// 开始进行发布和订阅
nh.advertise(chatter);
nh.subscribe(sub);
}

void loop()
{
    nh.spinOnce();
    delay(3);
}
```

编译上面的代码，并将其上传到Arduino开发板中。在上传代码前，需要在Arduino IDE中选择你使用的Arduino开发板型号，以及设备端口号。

在Tools|Boards菜单选择开发板型号，在Tools|Ports菜单选择开发板的设备端口号。本书作者使用的是Arduino Mega开发板。

编译并上传代码后，在PC主机上启动ROS桥接节点，连接Arduino和PC主机（见下面的命令行）。在执行此命令前，请确保已经将Arduino连接到PC主机上了：

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

本例中，我们在/dev/ttyACM0端口上运行了serial_node.py。我们可以列出/dev文件夹下的所有设备以找到端口号。注意，使用此端口需要root权限。本例中，可以使用以下命令更改权限，从而在所需端口上读写数据：

```
$ sudo chmod 666 /dev/ttyACM0
```

在此，我们使用`rosserial_python`节点作为ROS桥接节点。我们需要指定参数的设备名和波特率，默认通信波特率为57600。我们可以根据应用的实际要求更改波特率。`serial_node.py`的用法，可参考http://wiki.ros.org/rosserial_python。若ROS节点和Arduino节点之间的通信正常，将得到如图9-6所示的信息。

```
[INFO] [WallTime: 1438880620.972231] ROS Serial Python Node
[INFO] [WallTime: 1438880620.982245] Connecting to /dev/ttyACM0 at 57600 baud
[INFO] [WallTime: 1438880623.117417] Note: publish buffer size is 512 bytes
[INFO] [WallTime: 1438880623.118587] Setup publisher on chatter [std_msgs/String]
[INFO] [WallTime: 1438880623.132048] Note: subscribe buffer size is 512 bytes
[INFO] [WallTime: 1438880623.132745] Setup subscriber on talker [std_msgs/String]
```

图9-6 运行`rosserial_python`节点

当PC主机上开始运行`serial_node.py`时，它将发送`query`包的串口数据包以获取话题数据（如话题的数量、名称和类型）。这些话题是从Arduino节点接收到的。我们已经了解过用于ArduinoROS通信的串口数据包的结构。图9-7给出的是从`serial_node.py`发送到Arduino的`query`包的数据结构。



图9-7 query包的数据结构

这个query包的数据结构包含Sync Flag（同步标识）、ROS Version（版本）、Length（长度）、MD5（MD5校验）、Topic ID（话题ID）等字段。当在Arduino上接收到query包时，会做出响应，回复一条带有话题信息的消息，包含话题名称、类型、长度和数据等内容。图9-8就是一个来自Arduino常见的响应数据包。

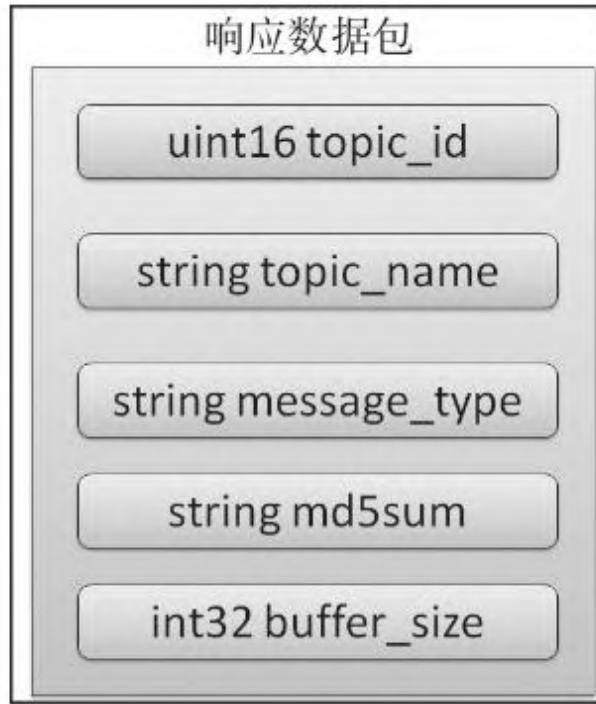


图9-8 响应数据包的结构

如果没有对query包进行响应，将再次发送query包。通信同步是利用ROS时间进行保证的。

从图9-6我们可以看出，当我们运行serial_node.py时，为发布者和订阅者分配了512字节的缓冲区。分配的缓冲区大小取决于我们正在使用的微控制器上可用的RAM的大小。下表列出了不同Arduino控制器的缓冲区大小，我们可以改变ros.h中的BUFFER_SIZE宏定义来调整这些值。

AVR型号	缓冲区大小	发布者 / 订阅者
ATMEGA168	150字节	6/6
ATMEGA328P	280字节	25/25
其他	512字节	25/25

在Arduino中，对使用ROS的float64数据类型有一些限制。它被截断为32位数据类型。另外，当我们使用string数据类型时，使用unsigned char型指针可以节省内存。

运行serial_node.py后，我们使用如下命令可以得到话题列表：

```
$ rostopic list
```

我们可以看到刚才生成的chatter和talker话题。使用如下命令向talker话题发送一条简单的消息：

```
$ rostopic pub -r 5 talker std_msgs/String "Hello World"
```

它将以5Hz的频率将“Hello World”这条消息发送到talker话题上。

用下面的命令可以查看chatter话题中的内容，我们将看到与我们刚发布的消息一样的信息：

```
$ rostopic echo /chatter
```

9.2.4 Arduino-ROS接口实例——LED灯闪烁/按按钮

本例中，我们可以将LED灯和按钮连接到Arduino上进行控制，并与ROS进行通信。当按钮被按下时，Arduino节点将向pushed话题中发送一条值为True的消息，与此同时，也会点亮Arduino板上的LED灯。

图9-9就是本实例中用到的电路图。

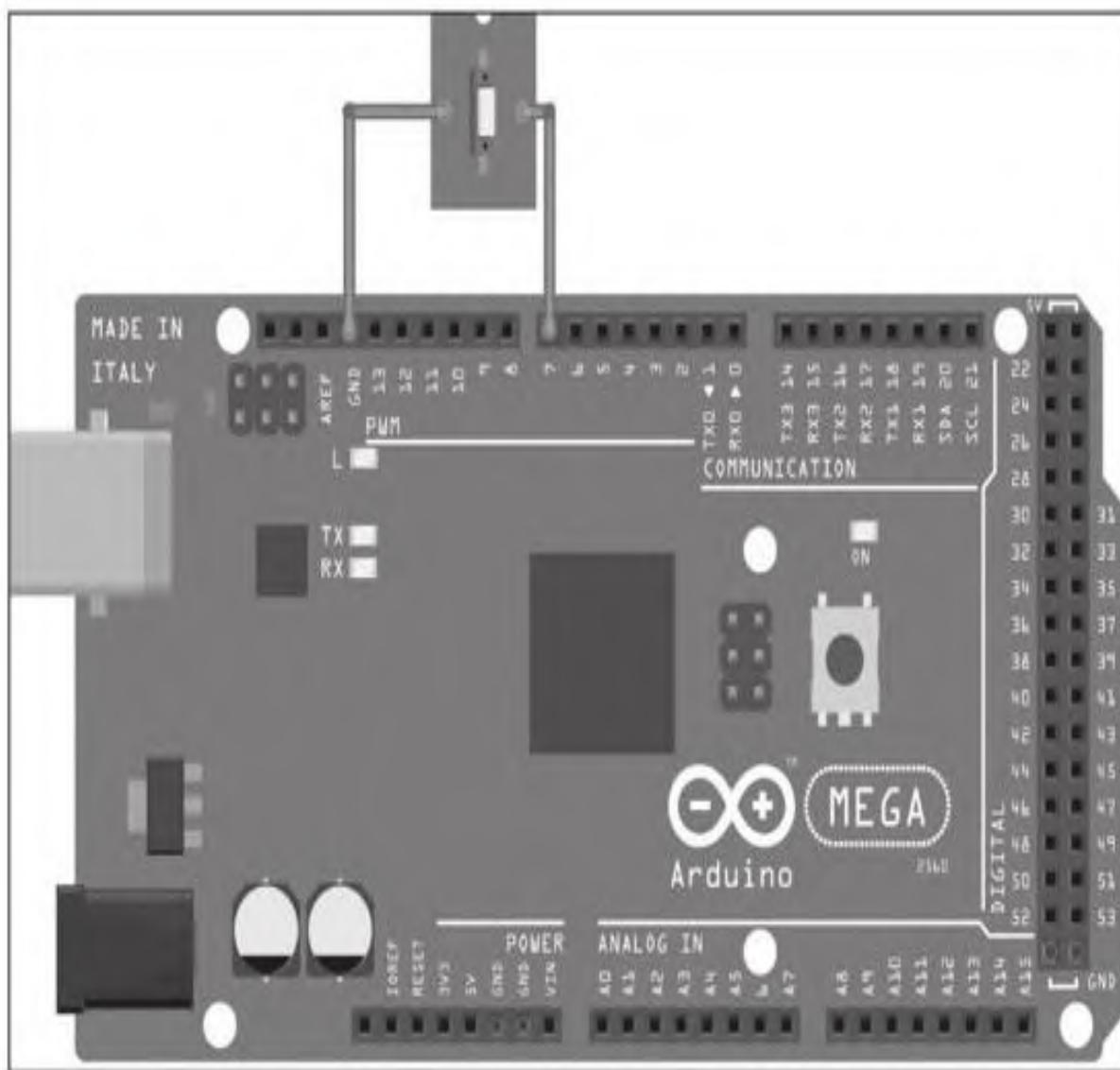


图9-9 将按钮连接到Arduino

```
/*
 * Button Example for Rosserial
 */

#include <ros.h>
#include <std_msgs/Bool.h>

// 节点句柄
ros::NodeHandle nh;

// 按钮发布的布尔消息
std_msgs::Bool pushed_msg;

// 定义一个名为 pushed 的话题发布者
ros::Publisher pub_button("pushed", &pushed_msg);
```

```
// LED 和按钮的引脚定义
const int button_pin = 7;
const int led_pin = 13;

// 用于处理抖动的变量
//https://www.arduino.cc/en/Tutorial/Debounce

bool last_reading;
long last_debounce_time=0;
long debounce_delay=50;
bool published = true;

void setup()
{
    nh.initNode();
    nh.advertise(pub_button);
    // 初始化 LED 引脚为输出模式
    // 初始化按钮引脚为输入模式
    pinMode(led_pin, OUTPUT);
    pinMode(button_pin, INPUT);
    // 使能按钮的上拉电阻
    digitalWrite(button_pin, HIGH);
    // 这个按钮是一个普通的按钮
    last_reading = ! digitalRead(button_pin);
}

void loop()
{
    bool reading = ! digitalRead(button_pin);
    if (last_reading!= reading){
        last_debounce_time = millis();
        published = false;
    }
    // 如果按钮值在抖动延时后还没有改变，我们认为按钮是正常的

    if ( !published && (millis() - last_debounce_time) >
debounce_delay) {
        digitalWrite(led_pin, reading);
        pushed_msg.data = reading;
        pub_button.publish(&pushed_msg);
        published = true;
    }

    last_reading = reading;
    nh.spinOnce();
}
```

上面的代码解决了按键抖动问题，只有在按下的按钮释放后才改变按钮的状态。将这部分代码上传到Arduino后，可以使用以下命令与ROS进行交互：

启动roscore：

```
$ roscore
```

启动serial_node.py：

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

通过打印pushed话题中的内容，可得知按钮按下的状态：

```
$ rostopic echo pushed
```

按下按钮，我们会看到如图9-10所示的输出。

```
---  
data: False  
---  
data: True  
---  
data: False  
---  
data: False  
---  
data: True  
---
```

图9-10 按Arduino按钮时的输出结果

9.2.5 Arduino-ROS接口实例——ADXL 335加速度计

本实例将在Arduino Mega板的ADC引脚上连接ADXL 335加速度计，并使用ROS工具rqt_plot绘制数据的图形。

图9-11显示了ADXL 335和Arduino之间的电路连接图。

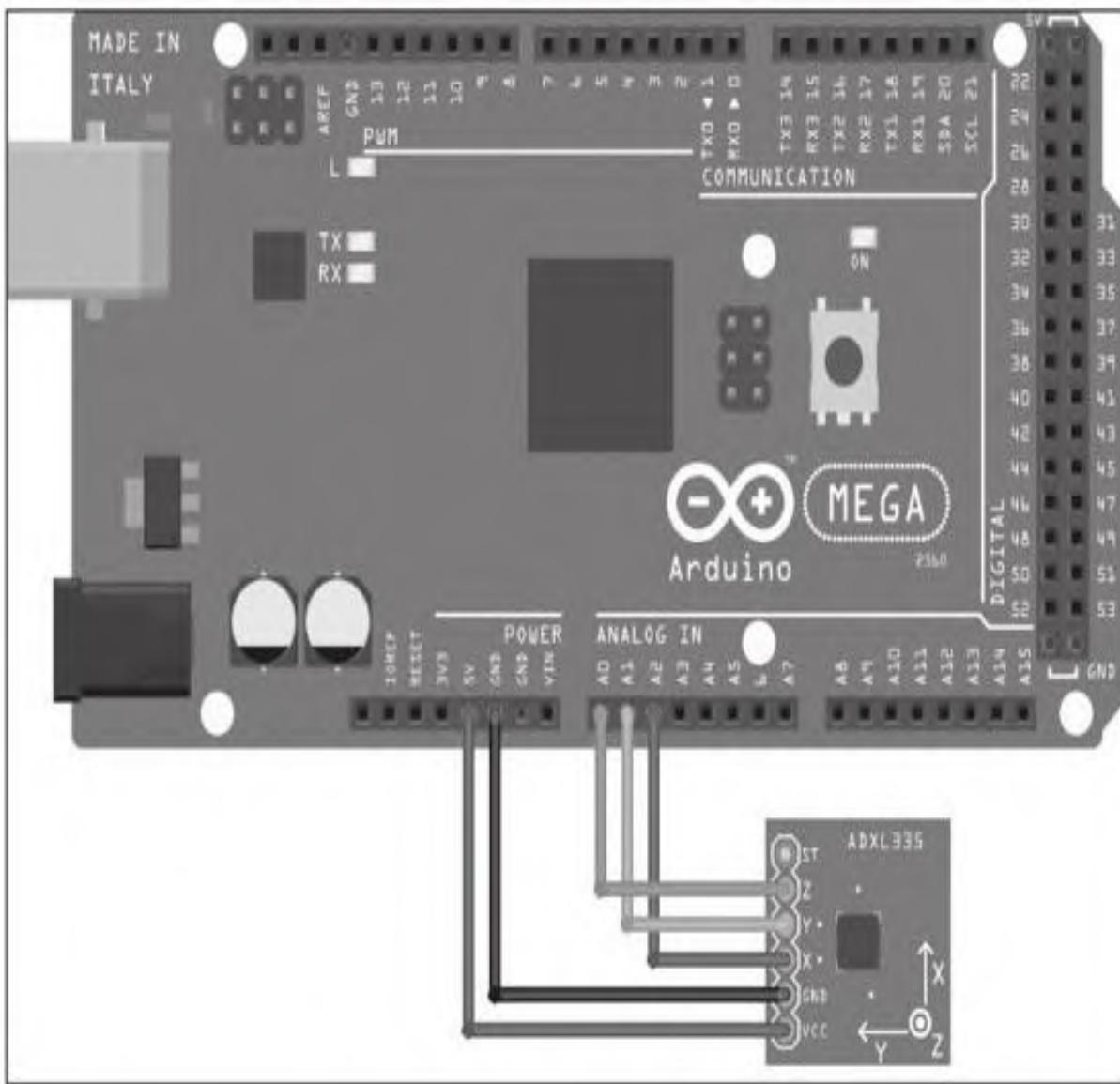


图9-11 Arduino-ADXL 335连接图

ADXL 335是一款模拟加速度计，可以方便地连接到ADC端口，然后读取数值结果。下面是将ADXL 335连接到Arduino ADC的一段嵌入式代

码:

```
#if (ARDUINO >= 100)
#include <Arduino.h>
#else
#include <WProgram.h>
#endif
#include <ros.h>
#include <rosserial_arduino/Adc.h>

const int xpin = A2;                      // x-axis of the accelerometer
const int ypin = A1;                      // y-axis
const int zpin = A0;                      // z-axis (only on 3-axis models)

ros::NodeHandle nh;

// 创建一条 adc 消息
rosserial_arduino::Adc adc_msg;

ros::Publisher pub("adc", &adc_msg);

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

// 我们计算读取到的模拟值的平均值，用来消除噪声
int averageAnalog(int pin){
    int v=0;
    for(int i=0; i<4; i++) v+= analogRead(pin);
    return v/4;
}

void loop()
{
    // 在 ADC 消息中插入 ADC 的读值
    adc_msg.adc0 = averageAnalog(xpin);
    adc_msg.adc1 = averageAnalog(ypin);
    adc_msg.adc2 = averageAnalog(zpin);

    pub.publish(&adc_msg);

    nh.spinOnce();

    delay(10);
}
```

上述代码将X、Y、Z三个轴的ADC值发布到了/adc话题上。这段代码用rosserial_arduino::Adc消息处理ADC值。我们可以使用rqt_plot工具来绘制这些值。

下面的命令在同一个图中绘制出了三个坐标轴的值：

```
$ rqt_plot adc/adc0 adc/adc1 adc/adc2
```

图9-12即是ADC的三个通道折线图。

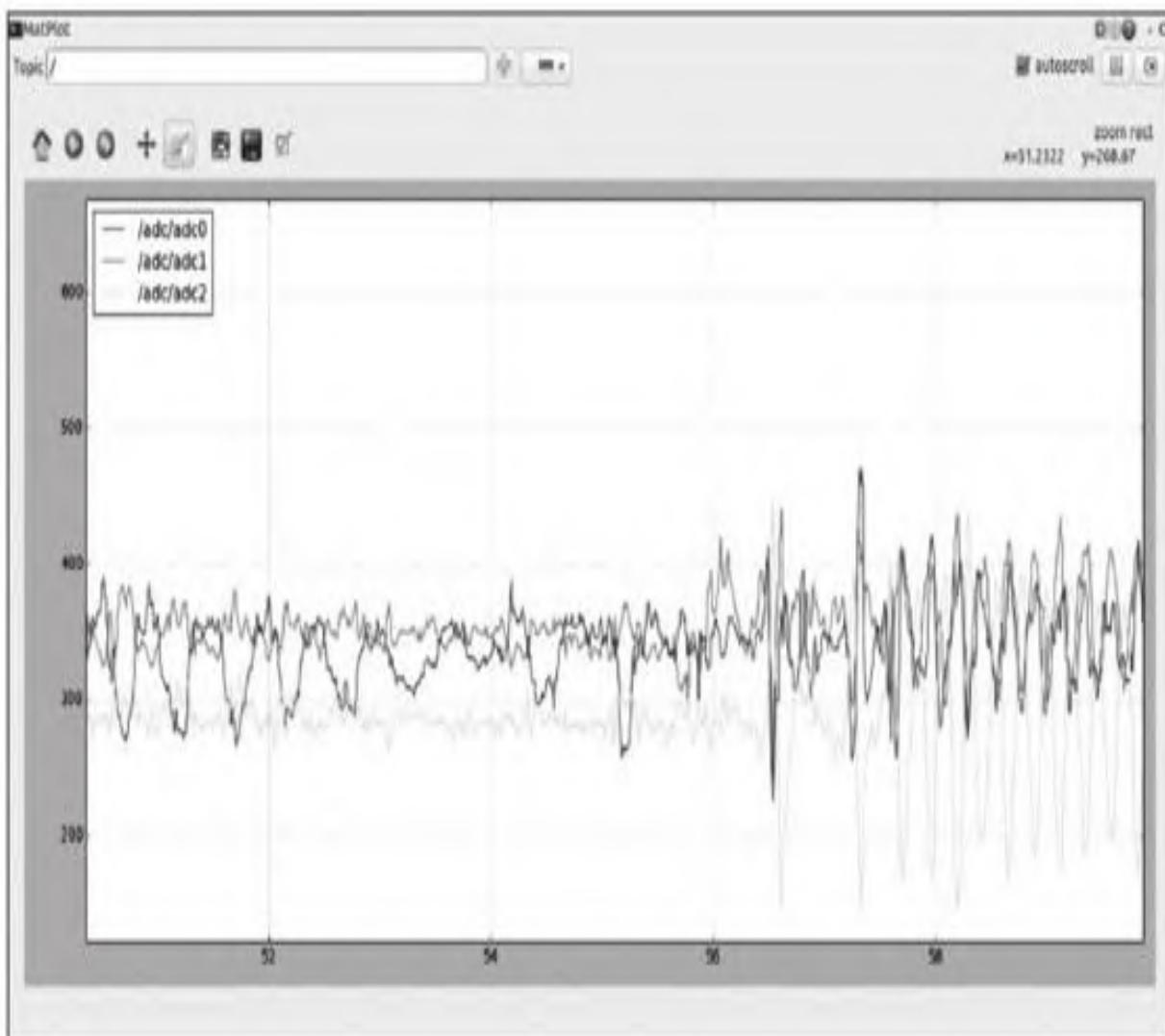


图9-12 使用rqt_plot绘制ADXL 335的值

9.2.6 Arduino-ROS接口实例——超声波测距传感器

测距传感器不仅是机器人中很实用的传感器之一，也是其中较便宜的传感器。超声波传感器有两个引脚，用于处理输入和输出信号，分别称为Echo和Trigger。本例使用的是HC-SR04超声波测距传感器，如图9-13所示。

超声波传感器由两部分构成：超声波发射器和超声波接收器。超声波测距传感器的工作原理如下：超声波传感器的trigger引脚上发送一个短时间的触发脉冲，超声波信号被发射到机器人所处的环境中。发射器发射的一部分超声波信号遇到障碍物后，被反射回传感器。反射回的超声波被接收器接收到后，生成一个输出信号，该信号的接收时间与反射回的超声波信号接收所需的时间成比例关系。

超声波测距传感器的测距公式

以下公式用于计算从超声波测距传感器到障碍物之间的距离：

$$\text{距离} = \text{速度} \times \text{时间}/2$$

海平面声速=343m/s或34 300cm/s

因此，距离=17 150×时间（单位cm）

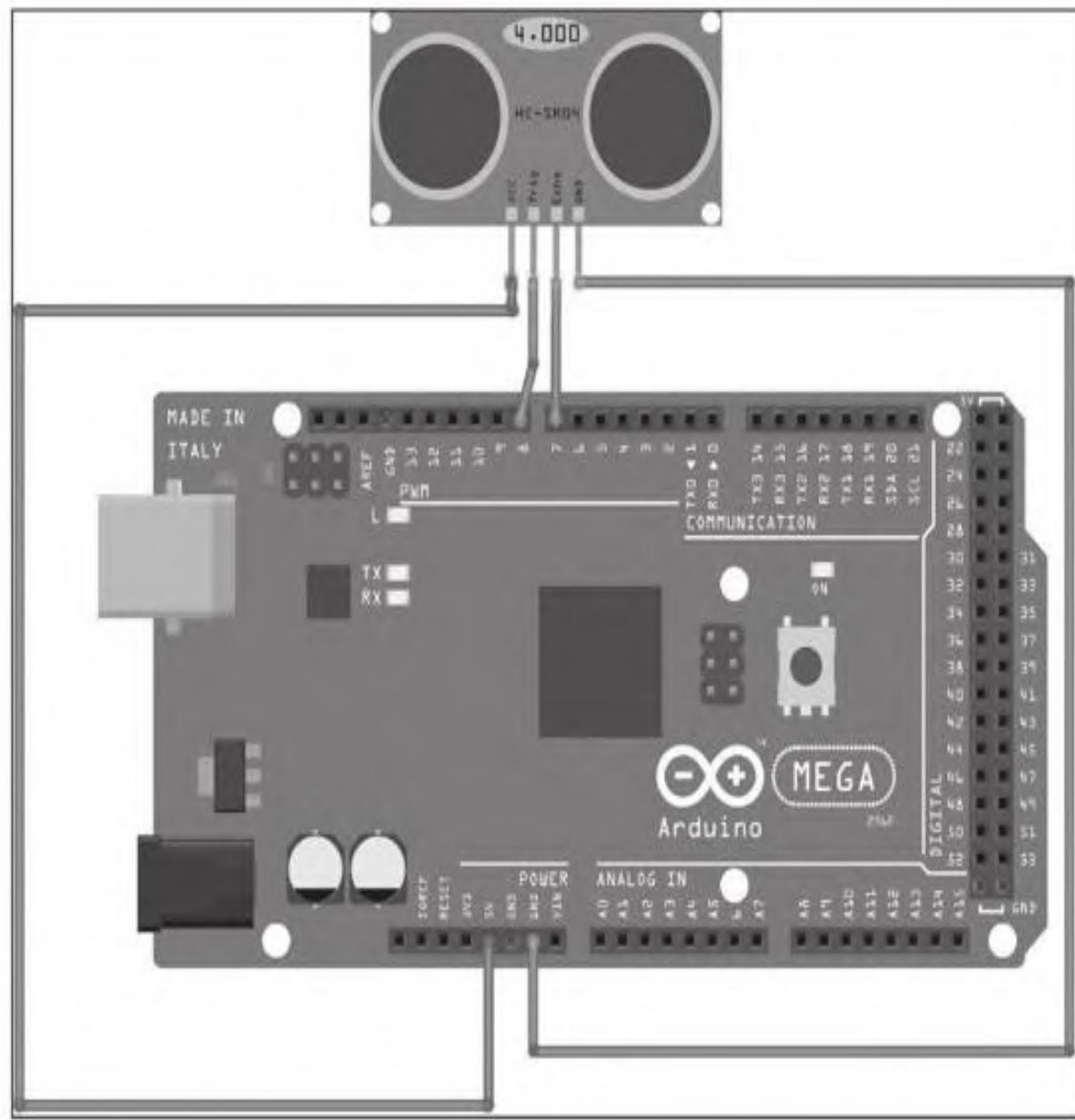


图9-13 HC-SR04超声波测距传感器与Arduino的连接图

我们可以用脉冲信号的持续时间来计算与障碍物之间的距离。下面是用超声波测距传感器测距的代码。需要定义一个使用range类型的ultrasound话题，然后向该话题中发送一个测距值：

```
#include <ros.h>
#include <ros/time.h>
#include <sensor_msgs/Range.h>

ros::NodeHandle nh;

#define echoPin 7
#define trigPin 8

int maximumRange = 200; // 所需的最大范围
int minimumRange = 0; // 所需的最小范围
long duration, distance; // 用于计算距离的持续时间

sensor_msgs::Range range_msg;
ros::Publisher pub_range( "/ultrasound", &range_msg);

char frameid[] = "/ultrasound";

void setup() {
    nh.initNode();
    nh.advertise(pub_range);
    range_msg.radiation_type = sensor_msgs::Range::ULTRASOUND;
    range_msg.header.frame_id = frameid;
    range_msg.field_of_view = 0.1; // fake
```

```
range_msg.min_range = 0.0;
range_msg.max_range = 60;
pinMode(trigPin, OUTPUT);
pinMode(echoPin, INPUT);
}

float getRange_Ultrasonic(){

    int val = 0;

    for(int i=0; i<4; i++) {
        digitalWrite(trigPin, LOW);
        delayMicroseconds(2);

        digitalWrite(trigPin, HIGH);
        delayMicroseconds(10);
        digitalWrite(trigPin, LOW);
        duration = pulseIn(echoPin, HIGH);
        // 根据声音的速度来计算距离 (厘米)
        val += duration;
    }
    return val / 232.8 ;
}

long range_time;

void loop() {
/* The following trigPin/echoPin cycle is used to determine the
distance of the nearest object by bouncing soundwaves off of it. */

    if ( millis() >= range_time ){
        int r =0;

        range_msg.range = getRange_Ultrasonic();
        range_msg.header.stamp = nh.now();
        pub_range.publish(&range_msg);
        range_time = millis() + 50;
    }
    nh.spinOnce();
    delay(50);
}
```

我们可以使用如下命令来绘制测距值：

启动roscore：

```
$ roscore
```

启动serial_node.py：

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

使用rqt_plot绘制测距值图：

```
$ rqt_plot /ultrasound
```

如图9-14所示，中间那条线表示测距传感器当前的测距值range。上面的那条线是max_range，下面的那条线是min_range。

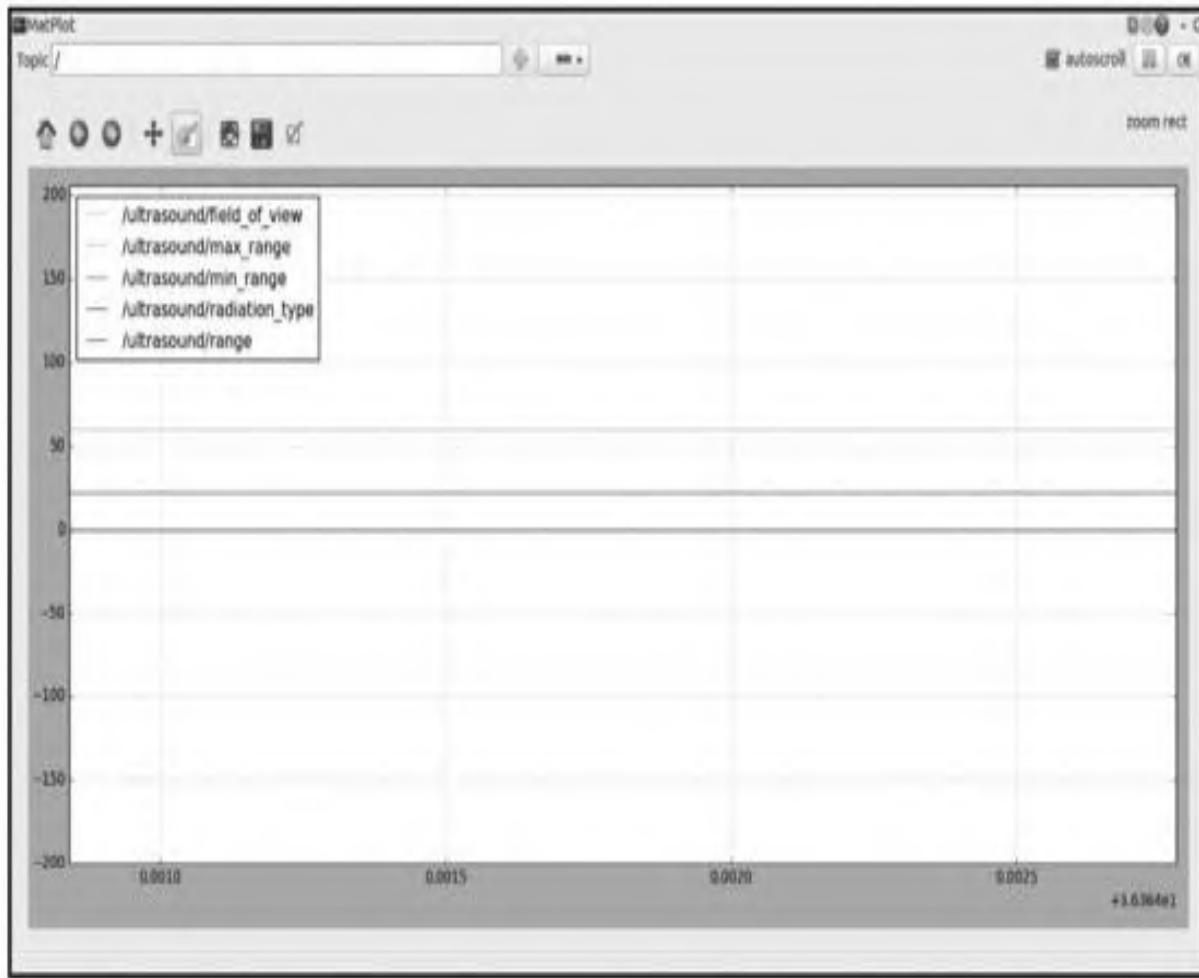


图9-14 绘制超声波传感器的测距值

9.2.7 Arduino-ROS接口实例——里程计发布者

在这个例子中，我们将了解如何将一条odom消息，从一个Arduino节点发送到PC主机上。本实例可以用于机器人计算odom，并将其作为输入发送到ROS navigation软件包集（注：用于机器人导航）。马达编码器可用于计算odom，然后将计算得到的odom发送到PC主机上。本例中，我们将学习如何为一个做圆周运动的机器人发送其里程计信息，且暂不考虑马达编码器的值：

```
/*
 * rosserial Planar Odometry Example
 */

#include <ros.h>
#include <ros/time.h>
#include <tf/tf.h>
#include <tf/transform_broadcaster.h>

ros::NodeHandle nh;
// 坐标变换广播对象
geometry_msgs::TransformStamped t;
tf::TransformBroadcaster broadcaster;

double x = 1.0;
double y = 0.0;
double theta = 1.57;

char base_link[] = "/base_link";
char odom[] = "/odom";
```

```
void setup()
{
    nh.initNode();
    broadcaster.init(nh);
}

void loop()
{
    // 圆周运动
    double dx = 0.2;
    double dtheta = 0.18;

    x += cos(theta)*dx*0.1;
    y += sin(theta)*dx*0.1;
    theta += dtheta*0.1;

    if(theta > 3.14)
        theta=-3.14;
    // odom->base_link 的坐标变换
    t.header.frame_id = odom;
    t.child_frame_id = base_link;
    t.transform.translation.x = x;
    t.transform.translation.y = y;
    t.transform.rotation = tf::createQuaternionFromYaw(theta);
    t.header.stamp = nh.now();
    broadcaster.sendTransform(t);
    nh.spinOnce();
    delay(10);
}
```

上传代码后，运行roscore和rosserial_node.py，我们就可以在RViz中查看tf和odom信息了。如图9-15所示，我们能看到odom的箭头在RViz中一直做圆圈运动。

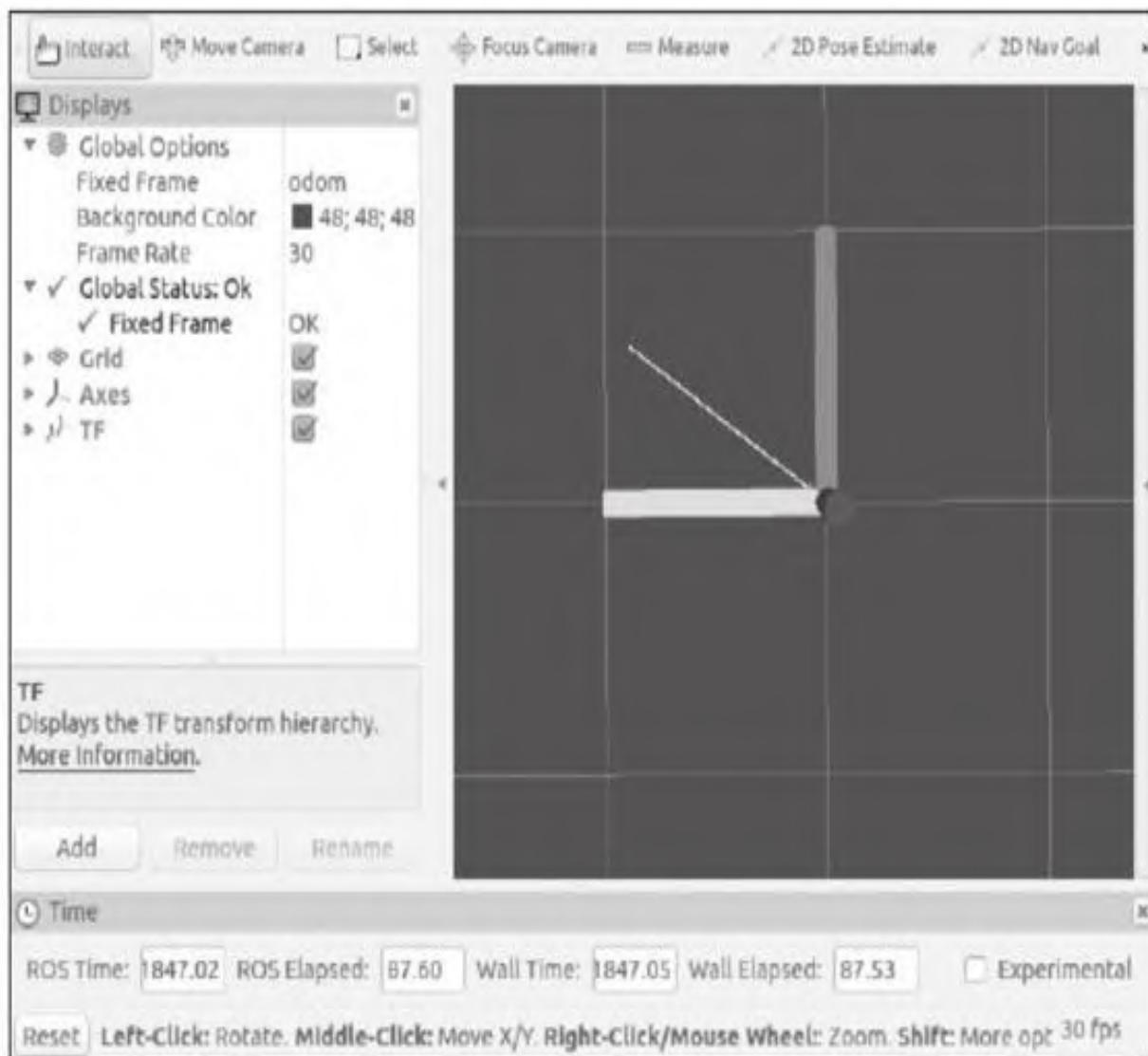


图9-15 可视化来自Arduino的odom数据

9.3 非Arduino开发板与ROS接口

Arduino是机器人中常用的开发板，但是如果我们想要使用比Arduino更强大的开发板该怎么办呢？在这种情况下，我们可能就需要为开发板编写驱动程序了，从而可以将串口消息转换成话题。

在第11章中，我们将学习用Python驱动节点，将一个非Arduino开发板（Tiva C Launchpad开发板）与ROS相连，使用ROS构建和连接差速驱动的移动机器人，并介绍了如何将一个真正的移动机器人连接到ROS上，这个机器人使用的就是Tiva C Launchpad开发板。

9.3.1 在Odroid-XU4和树莓派2上配置ROS

Odroid-XU4和树莓派2都是单板计算机，外形小巧，跟一张信用卡差不多大。这些单板计算机可以安装在机器人上，我们可以在它们上面安装ROS系统。

Odroid-XU4和树莓派2的主要规格比较如下：

设备	Odroid-XU4	树莓派 2
CPU	2.0 GHz Quad core ARM Cortex-A15 CPU 由三星生产	900 MHz quad core ARM Cortex A7 CPU 由博通生产
GPU	Mali-T628 MP6 GPU	VideoCore IV
内存	2GB	1GB
存储器	SD 卡插槽或 eMMC 模块	SD 卡插槽
外部接口	2 x USB 3.0, 1 x USB 2.0, micro HDMI, Gigabit Ethernet	4 x USB, HDMI, Ethernet, 3.5 mm audio jack
操作系统	Android,Ubuntu/Linux	Raspbian,Ubuntu/Linux,Windows 10
内部接口	GPIO, SPI, I2C, RTC (Real Time Clock) backup battery connector	Camera interface (CSI), GPIO, SPI, I2C, JTAG
价格	\$59	\$35

图9-16是一片Odroid-XU4开发板。

Odroid开发板是由Hard Kernel公司制造的，官方网址是
http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825。

Odroid-XU4是Odroid系列功能最强大的开发板。此外，它还有较便宜和性能稍低的开发板，如Odroid-C1、Odroid-C2。所有这些开发板都支持ROS。

最流行的单板计算机之一恐怕要属树莓派了。树莓派是由位于英国的树莓派基金会制造的（访问<https://www.raspberrypi.org>）。

图9-17是一张树莓派2开发板。



图9-16 Odroid-XU4开发板

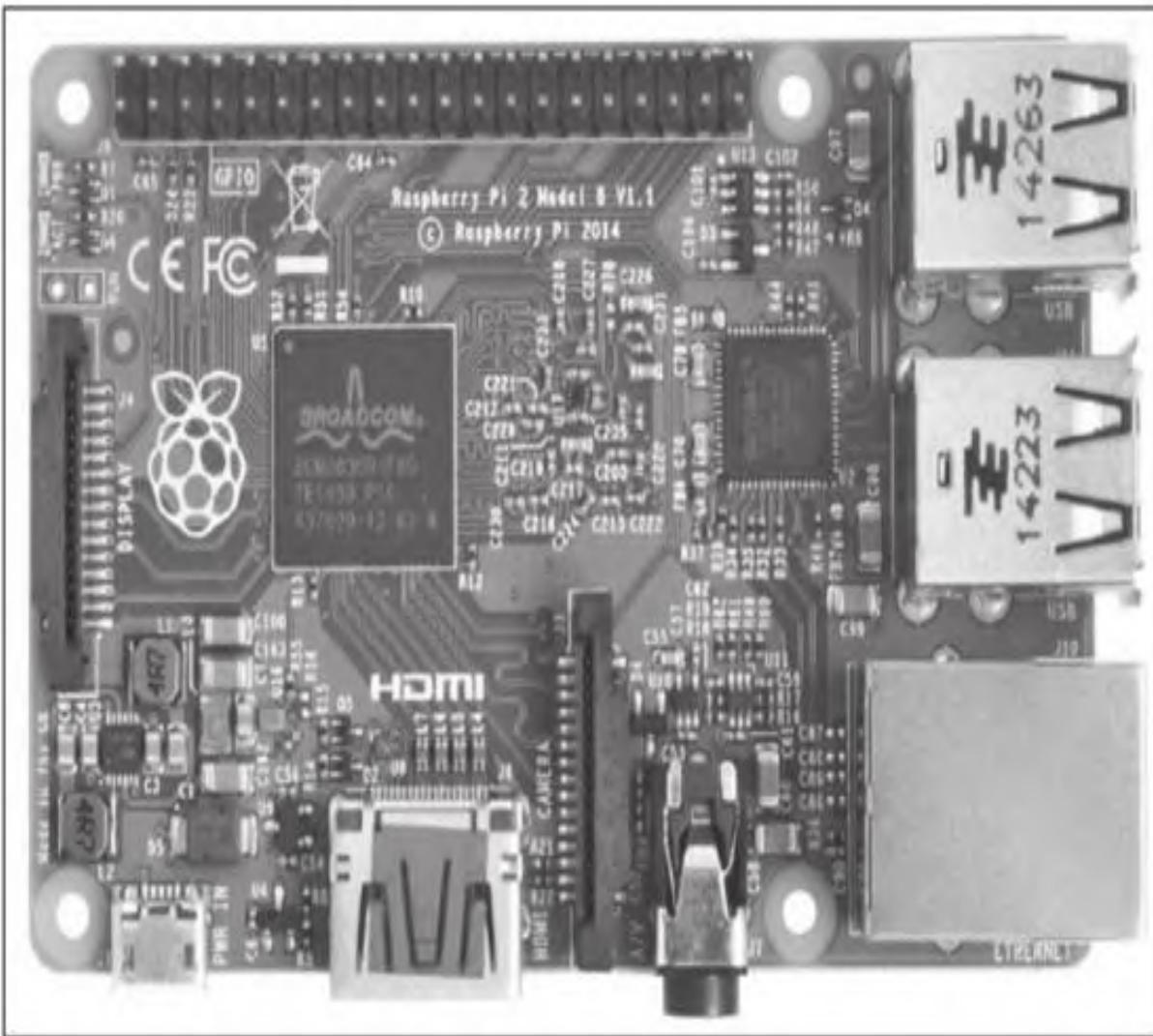


图9-17 树莓派2开发板

可以在Odroid上安装Ubuntu和Android。当然也可以安装一些非官方发布的Linux版本，例如Debian mini、Kali Linux、Arch Linux和Fedora，以及其他一些支持库，例如ROS、OpenCV、PCL等。为了在Odroid上安装ROS，我们可以安装一个全新的Ubuntu系统，然后像在PC上安装标准桌面版一样来安装ROS，或者也可以直接下载一个已经安装好的ROS，面向Odroid非官方Ubuntu版本镜像。

可以从http://de.eu.odroid.in/ubuntu_16.04lts/下载为Odroid开发板制作的Ubuntu 16.04版本镜像。你也可以为Odroid-XU4下载想要的其他内核版本（例如，ubunt-16.04-mate-odroid-xu4-20170731.img.xz）。这个文件包含预先安装的Ubuntu镜像。

Odroid-XU4支持的其他操作系统可以在下面这个维基页面上查到<http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4>。树莓派2官方的操作系统镜像位于<https://www.raspberrypi.org/downloads/>。

树莓派基金会支持的官方操作系统是Raspbian和Ubuntu。一些非官方发布，基于这些操作系统的镜像，预装了ROS。本书使用的Raspbian Jessie镜像，预装了ROS（<https://www.raspberrypi.org/downloads/>）。ROS安装维基页面为<http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Kinetic%20on%20the%20Raspberry%20Pi>。

如何在Odroid-XU4和树莓派2上安装一个操作系统镜像

我们可以下载Odroid的Ubuntu版本镜像和树莓派版本的Jessie镜像，然后可以将其安装在一个micro SD卡上，SD卡最好是16GB。将micro SD卡格式化为FAT32文件系统，或者用SD卡适配器或USB存储卡读卡器连接到电脑上。

我们既可以在Windows也可以在Linux下安装系统，操作系统的安装流程如下。

在Windows中安装

在Windows中，有一个专门为Odroid设计的Win32DiskImage工具，你可以从下面的网址下载该工具：

http://dn.odroid.com/DiskImager_ODROID/Win32DiskImager-odroid-v1.3.zip。

使用管理员权限运行Win32 Disk Imager工具。选择下载的镜像，选择内存卡驱动器，然后将镜像写入到驱动器中。

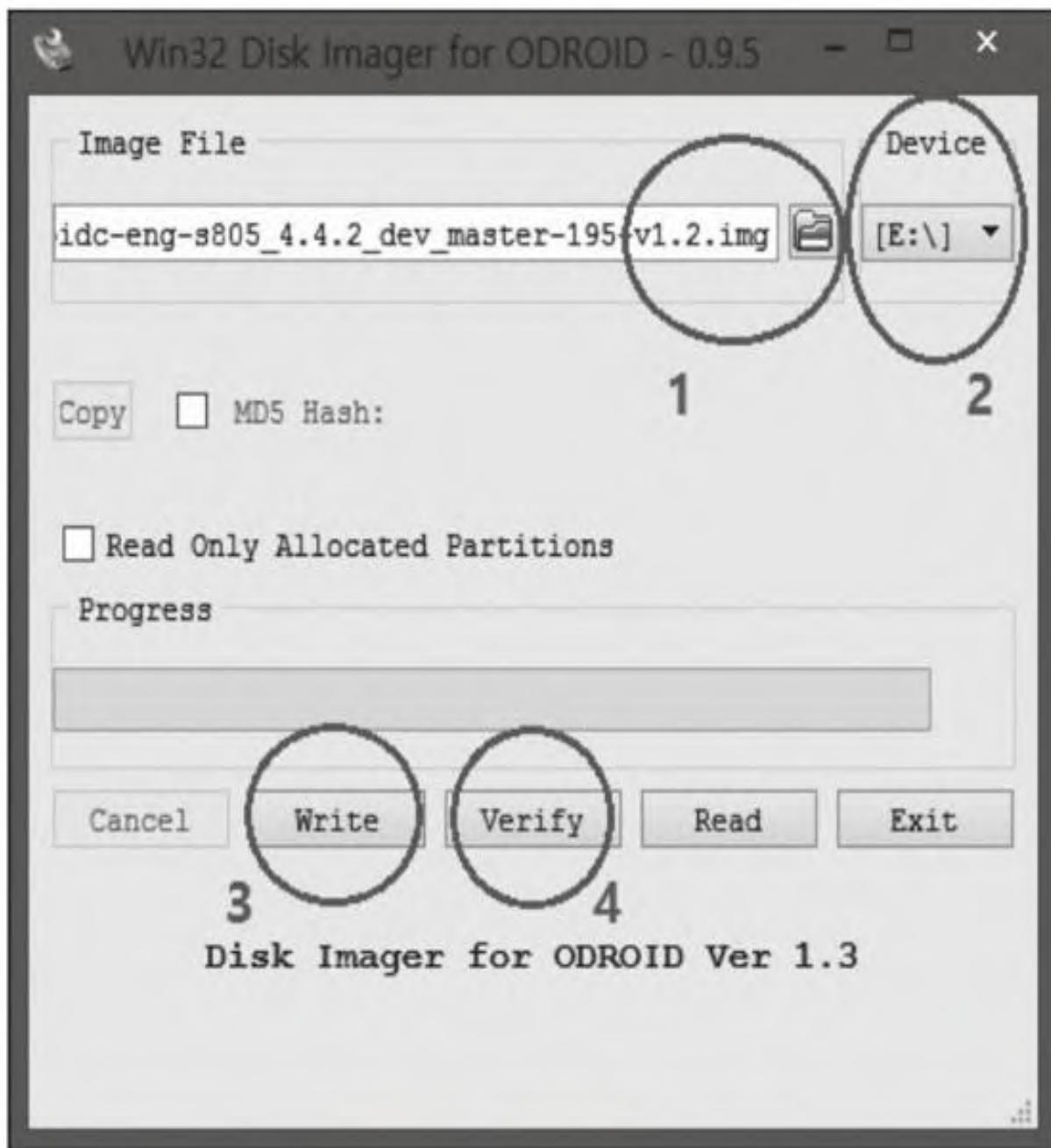


图9-18 Odroid版本的Win32 Disk Imager工具

完成该操作后，我们可以将micro SD卡插到Odroid上，然后启动支持ROS的操作系统。我们也可以用同样的工具在树莓派2上安装Raspbian镜像。从链接<http://sourceforge.net/projects/win32diskimager/>，我们能下载Win32 Disk Imager工具，从而将Raspbian镜像写到micro SD卡。

在Linux上安装

在Linux中，有一个叫磁盘转储（Disk Dump， dd）的工具。此工具可以帮助我们将镜像复制到SD卡上。dd是一个命令行工具，所有Ubuntu/Linux系列的操作系统中都有该工具。插入microSD卡，格式化为FAT32文件系统，最后使用下面的命令将镜像写入到micro SD卡中。

dd工具默认没有进度条来显示复制进度。为了显示进度条，可以安装一个叫pv的管道查看器工具：

```
$ sudo apt-get install pv
```

安装pv后，就可以使用下面的命令将镜像文件安装到micro SD卡上。注意，需要确保存储镜像的路径与终端（Terminal）中的路径一致，并留意micro SD卡设备名，例如mmcblk0、sdb、sdd等。可以用dmesg命令得到设备名：

```
$ dd bs=4M if=image_name.img | pv | sudo dd of=/dev/mmcblk0
```

image_name.img是镜像的名称，/dev/mmcblk0是SD卡设备名，bs=4M表示块大小。如果块大小设置为4M，dd将每次从镜像中读取4M字节并写入SD卡设备中。完成镜像文件复制操作后，将SD卡插到Odroid或树莓派上，就可以启动操作系统了。

PC主机连接Odroid-XU4和树莓派2

我们可以像使用普通电脑那样，使用Odroid-XU4和树莓派2，如在HDMI端口连接显示屏，在USB接口上连接键盘和鼠标。这是使用Odroid和树莓派最简单的方式。

大多数机器人项目，开发板是安装在机器人上的，因此我们无法将显示屏和键盘连接到开发板上。将这些开发板连接到PC上，可以采用下面几种方法。如果能将这些开发板连上网络，就比较好办了。用以下方法可以将这些开发板连上网络，然后，就可以使用SSH协议远程连接到开发板上：

- 使用Wi-Fi路由器和便携Wi-Fi适配器联网后，通过SSH远程连接：该方法需要一台联网用的Wi-Fi路由器和一个Wi-Fi适配器。PC主机和开发板都连接到同一个网络，这样每个联网设备都将有自己的IP地址，然后可以使用该地址进行通信。
- 使用网络热点直接连接：我们可以共享网络连接，然后通过Dnsmasq使用SSH进行通信（Dnsmasq是一个免费的DNS转发软件，用很少的系统资源提供DHCP服务）。使用Dnsmasq，可以共享笔记本电脑的Wi-Fi网络，然后将开发板连接到笔记本电脑的以太网接口。这样的连接方式适用于不动的机器人。

第一种方法很容易配置，它就像将两台PC连到同一个网络一样。第二种方法是将开发板的网络接口连接到笔记本电脑上。这种方法可以在静止不动机器人上使用，开发板和笔记本电脑通过SSH进行通信。这种方法还可以共享网络访问功能。本章使用第二种方法与ROS通信。

为Odroid-XU4和树莓派2配置网络热点

下面演示了如何在Ubuntu中创建网络热点，并共享Wi-Fi网络连接。

在网络设置中，找到Edit Connections...，然后点击Add添加一个新的网络连接，如图9-19所示。

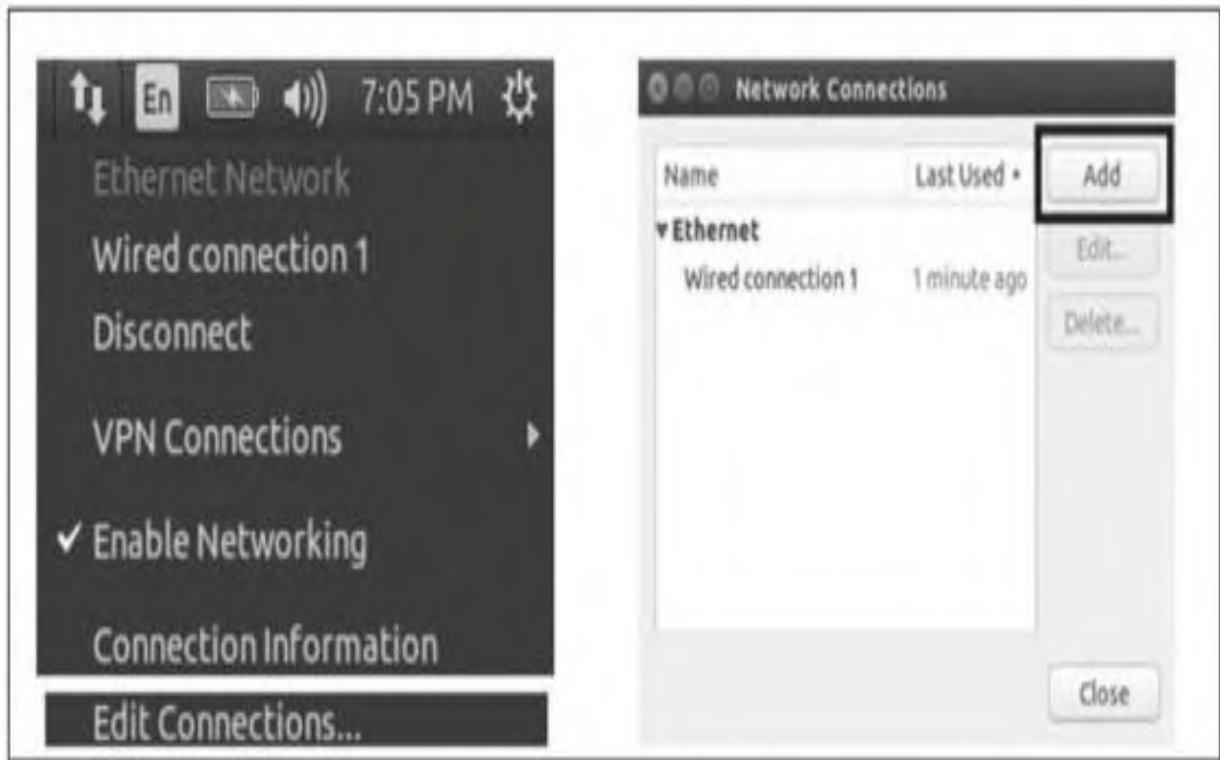


图9-19 在Ubuntu中配置网络连接

创建一个Ethernet网络连接，然后在IPv4设置中将Method选项更改为Shared to other computers，并将连接名称命名为Share，如图9-20所示。

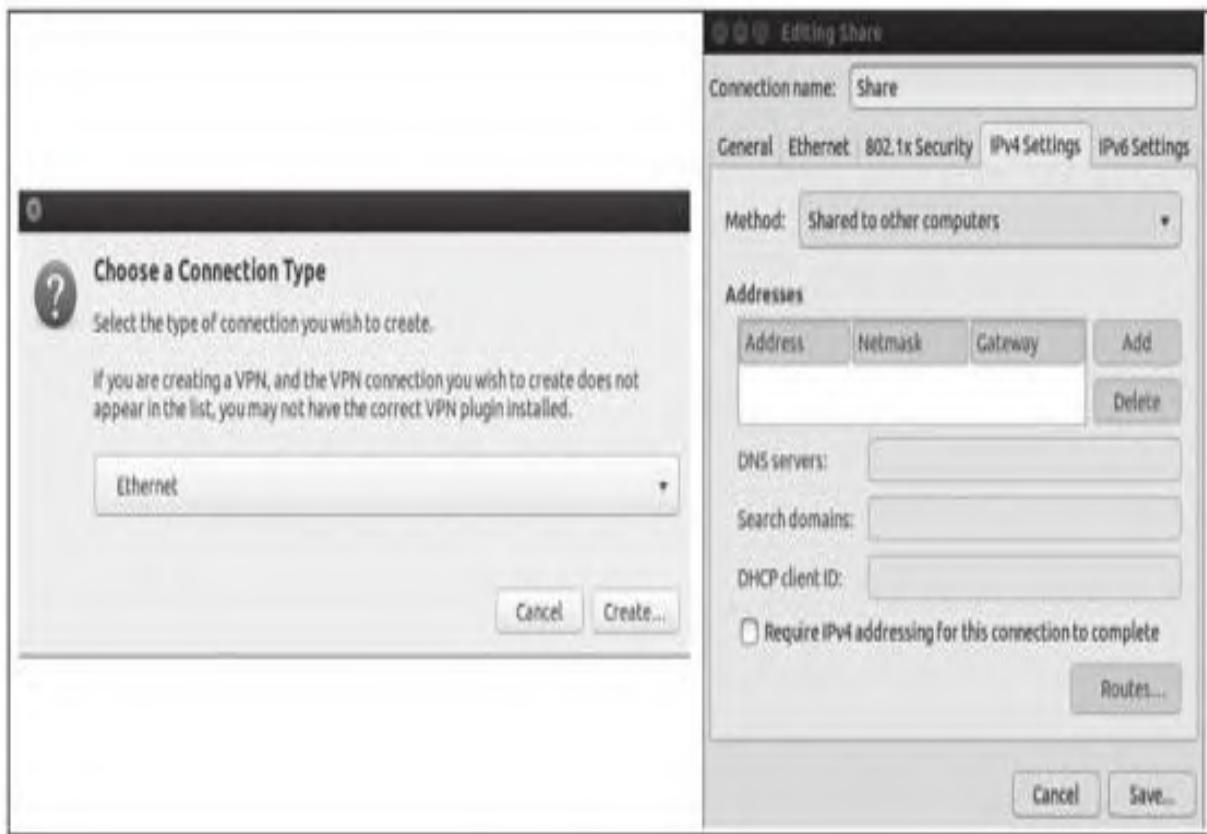


图9-20 创建一个共享网络连接

插入micro SD卡，上电启动Odroid或树莓派，然后将开发板的以太网接口连接到PC主机上。当开发板启动后，将看到开发板会自动连接到共享网络。

可以使用下面的命令与开发板进行通信：

使用Odroid：

```
$ ssh odroid@ip_address  
password is odroid
```

使用树莓派2：

```
$ ssh pi@ip_adress  
password is raspberry
```

用SSH连接开发板后，我们可以像PC一样，在开发板上启动roscore和大部分的ROS命令。在此，我们结合两个实例来看看如何使用这些开发板。一个实例是LED灯闪烁，另一个是处理按钮操作。我们使用Wiring Pi库来处理Odroid和树莓派的GPIO^[1]引脚。

Odroid和树莓派具有相同的引脚布局，大多数树莓派的GPIO库都移植到了Odroid，这样让编程变得更加简单。本章采用Wiring Pi库来对GPIO编程。Wiring Pi库可以通过C++ API接口来访问开发板的GPIO。

下面章节将介绍如何在Odroid和树莓派2上安装Wiring Pi库。

在Odroid-XU4上安装Wiring Pi库

下面演示如何在Odroid-XU4上安装Wiring Pi库，这个版本的Wiring Pi库是专为树莓派2定制的：

```
$ git clone https://github.com/hardkernel/wiringPi.git  
$ cd wiringPi  
$ sudo ./build
```

Odroid-XU4有42个引脚，分别位于两个不同的扩展头CON10和CON11上，如图9-21所示。

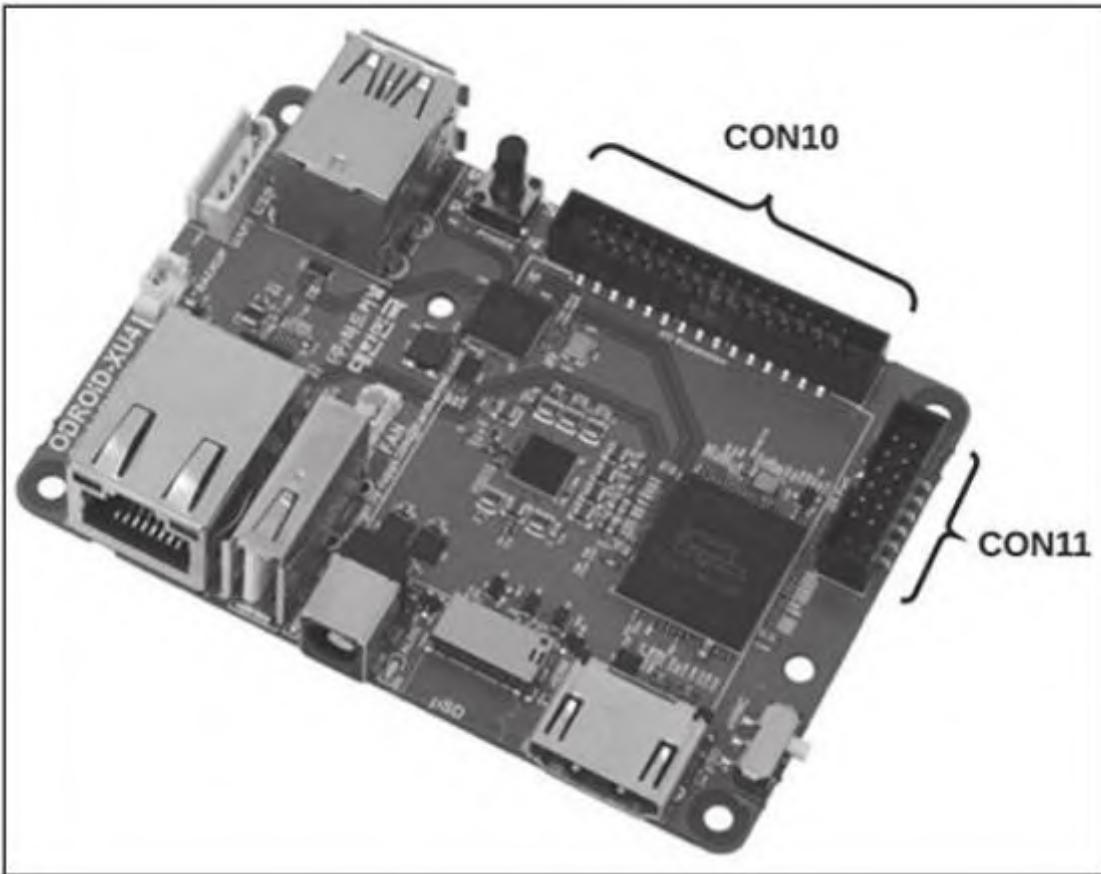


图9-21 Odroid-XU4上的CON10和CON11扩展头

图9-22给出了采用Odroid-XU4（CON10）扩展头，用于机器人连接的Wiring Pi引脚布局。

ODROID XU4 Pin Layout (CON10)							
WiringPi GPIO#	Name(GPIO#)	Label	HEADER		Label	Name(GPIO#)	WiringPi GPIO#
		5V0	1	2	GND		
	ADC_0.AIN0	AIN0	3	4	#173	UART0_RTS	1
0	UART_CTS	#174	5	6	#171	UART0_RXD	16
12	MOSI_SPI1	#192	7	8	#172	UART0_TxD	15
13	MISO_SPI1	#191	9	10	#189	CLK_SPI1	14
10	CSN_SPI1	#190	11	12	PRWON		
2	GPIO	#21	13	14	#210	SCL_I2C	9
7	GPIO	#18	15	16	#209	SDA_I2C	8
3	GPIO	#22	17	18	#19	GPIO	4
22	GPIO	#30	19	20	#28	GPIO	21
26	GPIO	#29	21	22	#31	GPIO	23
	ADC_0.AIN3	AIN3	23	24	#25	GPIO	11
5	SCL_I2C	#23	25	26	#24	GPIO	6
27	SDA_I2C	#33	27	28	GND	GND	
		1V8	29	30	GND	GND	

图9-22 Odroid-XU4, CON10的引脚布局

图9-23给出的是采用Odroid-XU4 (CON11) 扩展头，用于机器人连接的Wiring Pi引脚布局。

ODROID XU4 Pin Layout (CON11)							
WiringPi GPIO#	Name(GPIO#)	Label	HEADER		Label	Name(GPIO#)	WiringPi GPIO#
		5V0	1	2	GND		
		1V8	3	4	#173	SDA_I2C_5	30
	GPIO	#34	5	6	#171	SCL_I2C_5	31
	SCLK_I2S_0	#225	7	8	#172	GND	
	CDCLK_I2S_0	#226	9	10	#189	SDO_I2S_0	
	LRCK_I2S_0	#227	11	12	PRWON	SDI_I2S_0	

图9-23 Odroid-XU4, CON11的引脚布局

在树莓派2上安装Wiring Pi库

下面的操作用于在树莓派2上安装Wiring Pi库：

```
$ git clone git clone git://git.drogon.net/wiringPi
$ cd wiringPi
$ sudo ./build
```

图9-24显示了树莓派2和Wiring Pi的引脚列表。

P1: The Main GPIO connector							
WiringPi Pin	BCM GPIO	Name	Header	Name	BCM GPIO	WiringPi Pin	
		3.3v	1 2	5v			
8	Rv1:0 - Rv2:2	SDA	3 4	5v			
9	Rv1:1 - Rv2:3	SCL	5 6	0v			
7	4	GPIO7	7 8		14	15	
		0v	9 10		15	16	
0	17	GPIO0	11 12	GPIO1	18	1	
2	Rv1:21 - Rv2:27	GPIO2	13 14	0v			
3	22	GPIO3	15 16	GPIO4	23	4	
		3.3v	17 18	GPIO5	24	5	
12	10	MOSI	19 20	0v			
13	9	MISO	21 22	GPIO6	25	6	
14	11	SCLK	23 24	CE0	8	10	
		0v	25 26	CE1	7	11	
WiringPi Pin	BCM GPIO	Name	Header	Name	BCM GPIO	WiringPi Pin	

P5: Secondary GPIO connector (Rev. 2 Pi only)							
WiringPi Pin	BCM GPIO	Name	Header	Name	BCM GPIO	WiringPi Pin	
		5v	1 2	3.3v			
17	28	GPIO8	3 4	GPIO9	29	18	
19	30	GPIO10	5 6	GPIO11	31	20	
		0v	7 8	0v			
WiringPi Pin	BCM GPIO	Name	Header	Name	BCM GPIO	WiringPi Pin	

图9-24 树莓派2的引脚列表

下面是树莓派2的ROS实例。

[1] General Purpose I/O Ports，意思为通用输入/输出端口。——译者注

9.3.2 用ROS控制树莓派2上的LED灯闪烁

这是一个LED灯闪烁的简单实例。将LED灯连接到Wiring Pi上的第一个引脚（即开发板上的引脚12）。将LED的阴极连接到引脚GND，阳极连接到引脚12。图9-25是树莓派2连接LED的电路图。

使用下面的命令可以创建ROS的测试软件包：

```
$ catkin_create_pkg ros_wiring_example roscpp std_msgs
```

你可以在ros_wiring_examples文件夹下找到这个软件包。

创建一个src文件夹，然后在该文件夹下创建一个名为blink.cpp的源码文件：

```
#include "ros/ros.h"
#include "std_msgs/Bool.h"
#include <iostream>

//Wiring Pi header
#include "wiringPi.h"

// Wiring PI 的第1个引脚
```

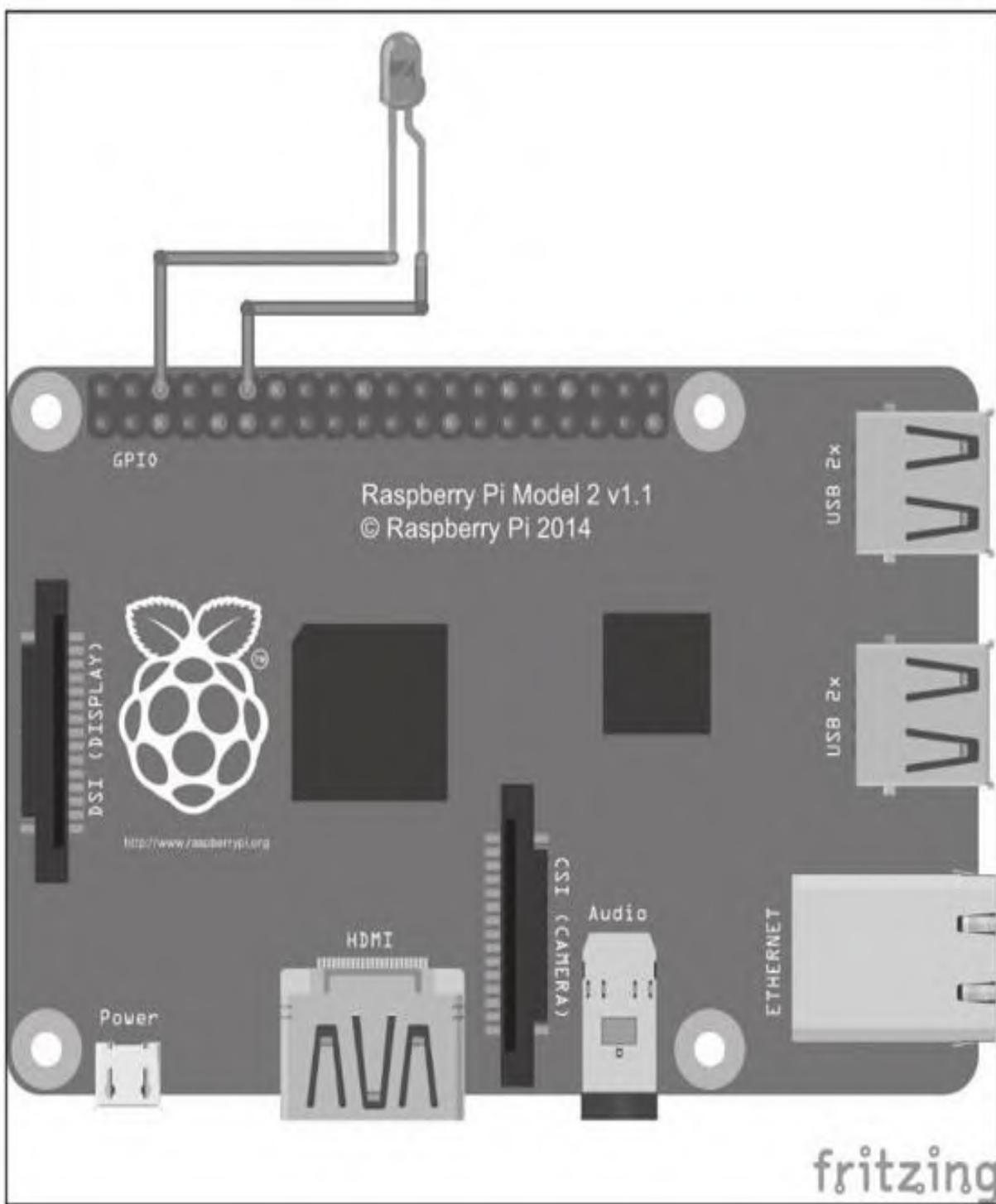


图9-25 用树莓派2控制LED灯闪烁

```
#define LED 1

// 回调函数根据话题值来控制 LED 的亮灭
void blink_callback(const std_msgs::Bool::ConstPtr& msg)
{
    if(msg->data == 1){
        digitalWrite (LED, HIGH) ;
        ROS_INFO("LED ON");
    }
    if(msg->data == 0){
        digitalWrite (LED, LOW) ;
        ROS_INFO("LED OFF");
    }
}

int main(int argc, char** argv)
{
    ros::init(argc, argv,"blink_led");
    ROS_INFO("Started Raspberry Blink Node");
    // 配置 WiringPi
    wiringPiSetup () ; // 配置 LED 引脚为输出模式
    pinMode(LED, OUTPUT);
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("led_blink",10,blink_callback);
    ros::spin();
}
```

该段代码订阅了一个名为led_blink的布尔类型的话题。如果向该话题发布消息1，就会点亮LED灯。如果发布消息0，这个LED灯将会熄灭。

9.3.3 在树莓派2上使用ROS测试按钮和LED灯闪烁

下面的实例可以处理一个按钮输入。当按下按钮时，这段代码将向led_blink话题发布消息，并使LED灯闪烁。当弹起按钮时，LED灯将会熄灭。该LED灯连接到引脚12和GND上，而按钮连接到引脚11和GND上。图9-26显示了本例中的连接电路图。Odroid的连接电路图也是一样的。

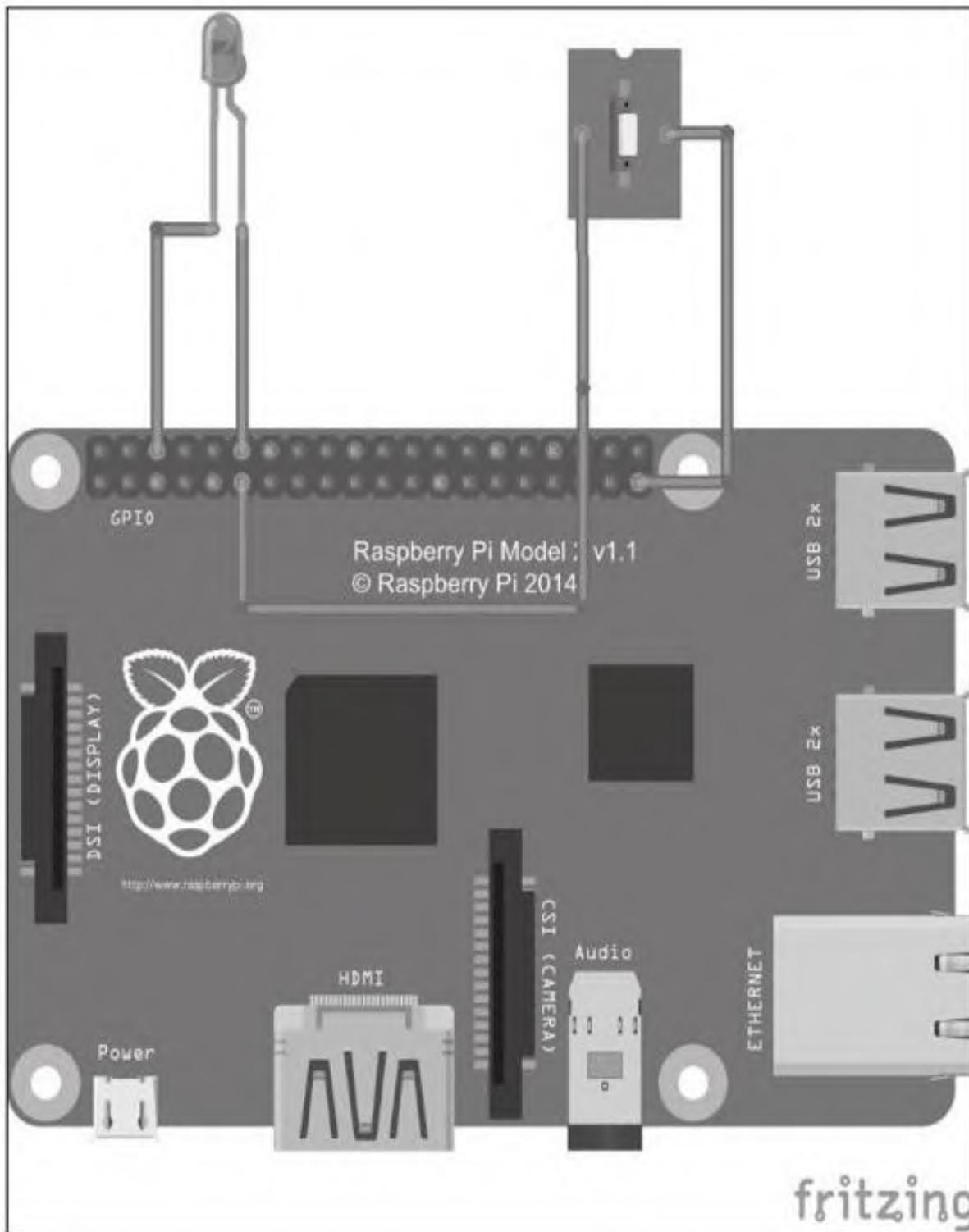


图9-26 树莓派2上LED和按钮的连接方式

下面是连接LED和按钮的代码。这段代码保存于src文件夹下的button.cpp源码文件中：

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

#include <iostream>
#include "wiringPi.h"

//Wiring PI 1
```

```
#define BUTTON 0
#define LED 1

void blink_callback(const std_msgs::Bool::ConstPtr& msg)
{
    if(msg->data == 1) {
        digitalWrite (LED, HIGH) ;
        ROS_INFO("LED ON");
    }

    if(msg->data == 0) {
        digitalWrite (LED, LOW) ;
        ROS_INFO("LED OFF");
    }
}

int main(int argc, char** argv)
{
    ros::init(argc, argv,"button_led");
    ROS_INFO("Started Raspberry Button Blink Node");

    wiringPiSetup () ;

    pinMode(LED, OUTPUT);
    pinMode(BUTTON, INPUT);
    pullUpDnControl(BUTTON, PUD_UP); // 使能按钮的上拉电阻

    ros::NodeHandle n;
    ros::Rate loop_rate(10);

    ros::Subscriber sub = n.subscribe("led_blink",10,blink_callback);
    ros::Publisher chatter_pub = n.advertise<std_msgs::Bool>("led_blink",
10);

    std_msgs::Bool button_press;
    button_press.data = 1;

    std_msgs::Bool button_release;
    button_release.data = 0;

    while (ros::ok())
    {
        if (!digitalRead(BUTTON)) // 如果按钮被按下，将返回 True
        {
            ROS_INFO("Button Pressed");
            chatter_pub.publish(button_press);
        }
        else
```

```
{  
  
    ROS_INFO("Button Released");  
    chatter_pub.publish(button_release);  
  
}  
  
ros::spinOnce();  
loop_rate.sleep();  
  
}  
}
```

接下来给出的是编译这两个例子的CMakeLists.txt文件。软件包中包含Wiring Pi代码，因此需要包含Wiring Pi库，我们在CMakeLists.txt文件中添加这个库：

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_wiring_examples)

find_package(catkin REQUIRED COMPONENTS
    roscpp
    std_msgs
)

find_package(Boost REQUIRED COMPONENTS system)

// Wiring Pi 的头文件路径
set(wiringPi_include "/usr/local/include")

include_directories(
    ${catkin_INCLUDE_DIRS}
    ${wiringPi_include}
)

// Wiring Pi 的库文件路径
LINK_DIRECTORIES("/usr/local/lib")

add_executable(blink_led src/blink.cpp)

add_executable(button_led src/button.cpp)

target_link_libraries(blink_led
    ${catkin_LIBRARIES} wiringPi
)

target_link_libraries(button_led
    ${catkin_LIBRARIES} wiringPi
)
```

用`catkin_make`编译好项目后，我们就可以运行这些实例了。我们需要root权限才能执行基于Wiring Pi的程序。

9.3.4 在树莓派2上运行示例

到此，项目已经编译好了，但在运行之前，我们还需要对树莓派2做一些设置。可以通过SSH登录到树莓派2上完成这些设置。

我们需要将下面各行添加到root用户的**.bashrc**文件中。编辑root用户的**.bashrc**文件：

```
$ sudo -i  
$ nano .bashrc
```

将下面几行添加到该文件的末尾：

```
source /opt/ros/indigo/setup.sh  
source /home/pi/catkin_ws/devel/setup.bash  
export ROS_MASTER_URI=http://localhost:11311
```

我们可以通过另一个终端登录到树莓派2上，并运行下面的命令来执行**blink_demo**程序：

在终端启动**roscore**：

```
$ roscore
```

在另一个终端，以**root**身份运行可执行文件：

```
$ sudo -s  
# cd /home/odroid/catkin_ws/build/ros_wiring_examples  
#./blink_led
```

启动blink_led节点后，在另一个终端向led_blink话题发布1：

将LED设置为点亮（ON）状态：

```
$ rostopic pub /led_blink std_msgs/Bool 1
```

将LED设置为熄灭（OFF）状态：

```
$ rostopic pub /led_blink std_msgs/Bool 0
```

在另一个终端运行LED按钮节点：

```
$ sudo -s  
# cd /home/odroid/catkin_ws/build/ros_wiring_examples  
#./button_led
```

当按下按钮，我们可以看到LED灯在闪烁。我们也可以输出led_blink话题来检查按钮的状态：

```
$ rostopic echo /led_blink
```

9.4 将DYNAMIXEL驱动器连接到ROS

DYNAMIXEL是当前市场上最新的智能驱动器之一，由一家名为Robotis的公司制造。DYNAMIXEL舵机有多种版本可供选择，图9-27就是其中的一些版本。

这些智能驱动器全部支持ROS，并且也提供了完整的文档。

DYNAMIXEL的官方ROS维基页面是
http://wiki.ros.org/dynamixel_controllers/Tutorials。



图9-27 不同型号的DYNAMIXEL舵机

9.5 习题

- 各种rosserial软件包的不同之处在哪里？
- rosserial_arduino的主要功能是什么？
- rosserial协议是如何工作的？
- Odroid和树莓派开发板的主要区别是什么？

9.6 本章小结

本章介绍了如何将I/O开发板与ROS相连，并添加传感器。我们讨论了最流行的Arduino-ROS接口I/O开发板和连接基本组件（如LED、按钮、加速度计、超声波传感器等）。了解了Arduino-ROS接口后，我们又讨论了如何在树莓派2和Odroid-XU4上配置ROS。基于ROS和Wiring Pi库，我们还在Odroid和树莓派中演示了几个简单的实例。最后，我们简单看了看ROS下的智能驱动器DYNAMIXEL。

第10章

用ROS对视觉传感器编程、OpenCV、PCL

前一章，我们讨论了如何在ROS中使用开发板连接各种传感器和驱动器。本章，我们将讨论如何在ROS中连接各种视觉传感器，然后利用开源计算机视觉库（Open Source Computer Vision，OpenCV）和点云库（Point Cloud Library，PCL）进行编程。机器视觉是机器人进行物体操控和导航的一个重要内容。目前市场上有许多2D/3D的视觉传感器，而且大多数传感器在ROS中都有驱动程序。我们将讨论新型视觉传感器与ROS接口，以及如何用OpenCV和PCL进行编程。最后，我们将讨论用基准标记库开发基于视觉的机器人应用。

本章将介绍以下内容：

- ROS、PCL和OpenCV的集成。
- 在ROS中使用USB摄像头。
- 学习如何校准相机。
- 在ROS中使用RGB-D传感器。
- 在ROS中使用激光扫描仪。
- 在ROS中使用增强现实标记。

10.1 理解ROS-OpenCV开发接口软件包

OpenCV是流行的开源实时计算机视觉库，主要用C/C++编写。OpenCV基于BSD许可，无论在学术研究还是在商业应用都是免费的。OpenCV可以用C/C++、Python、Java进行编程，而且支持多种平台（如Windows、Linux、MacOS X、Android、iOS）。OpenCV内含大量的计算机视觉相关的API，可以用于开发计算机视觉应用程序。OpenCV库的官方网站为<http://opencv.org/>。

OpenCV通过名为vision_opencv的ROS软件集与ROS连接。vision_opencv包含两个重要的软件包，用于将OpenCV和ROS互联。它们分别是：

- cv_bridge：该软件包包含一个算法库，提供将OpenCV下的cv::Mat类型图像转换为ROS图像消息(sensor_msgs/Image)的API，当然反向转换的API也有。总之，它可以作为OpenCV和ROS之间的桥梁。只要我们想向另一个节点发送ROS图像类型消息，我们就可以用OpenCV的API来进行图像处理，并将其转换成ROS图像类型。我们将在后续章节讨论如何进行这种转换。
- image_geometry：在使用相机之前，我们需要做的一个操作是相机校准。image_geometry软件包包含使用C++和Python编写的算法库，可用校准参数来校正图像的几何形状。该软件包用类型名为sensor_msgs/CameraInfo的消息处理校准参数，然后将其传送给OpenCV的图像校正函数。

10.2 理解ROS-PCL开发接口软件包

点云数据可以定义为某一坐标系统下的一组数据点。在3D环境中，点云数据有x、y、z三个坐标。PCL则是用于处理2D/3D图像和点云数据的开源项目。

与OpenCV一样，它也基于BSD许可，可免费用于商业和学术研究。OpenCV也是跨平台的软件包，支持Linux、Windows、MacOS、Android/iOS等操作系统。

PCL由大量标准算法组成，如滤波、分割、特征提取等，可用于实现不同类型的点云应用开发。PCL的官方网站为：

<http://pointclouds.org/>。

点云数据可以通过Kinect、华硕Xtion Pro、英特尔Real Sense等传感器获取。然后，我们就可以在机器人项目中使用这些数据，如机器人操控和抓取物体。PCL与ROS紧密集成，用于处理来自各类传感器的点云数据。perception_pcl软件集是PCL的ROS接口。该软件集可以将来自ROS的点云数据转换成PCL的数据类型，反之亦然。

perception_pcl软件集包含以下软件包：

- pcl_conversions：该软件包提供了将PCL数据类型转换为ROS消息的API，反之亦然。
- pcl_msgs：该软件包包含了在ROS定义的PCL相关消息。PCL相关消息包含：
 - ModelCoefficients
 - PointIndices

- PolygonMesh
- Vertices
- pcl_ros：该软件包是ROS与PCL之间的桥梁，包含将ROS消息与PCL数据类型桥接的工具和节点。
- pointcloud_to_laserscan：该软件包的主要功能是将3D点云数据转换为2D激光扫描数据。该软件包有助于将便宜的3D视觉传感器（如Kinect和华硕Xtion Pro）变成激光扫描仪。激光扫描数据主要用于2D-SLAM，可应用于机器人导航。

安装ROSperception软件包

我们将安装一个叫perception的软件包，它是ROS中的一个超软件包，包含所有与视觉相关的软件包，例如OpenCV、PCL等：

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

ROSperception软件包集包含以下各类ROS软件包：

- image_common：该软件包包含处理ROS图像的常用功能。这个超软件包由下列软件包组成 (http://wiki.ros.org/image_common)：
 - image_transport：该软件包可以在发布和订阅图像时压缩图像，这样可以节省带宽 (http://wiki.ros.org/image_transport)。压缩算法包括JPEG/PNG压缩和Theora视频流压缩。另外，还可以将自定义的压缩方法添加到image_transport中。
 - camera_calibration_parsers：该软件包包含一个从XML文件读/写相机校准参数的程序。该软件包主要用于相机驱动程序访问校准参数。
 - camera_info_manager：该软件包包含一个保存、恢复、加载校准信息的程序，主要由相机驱动程序调用。

- polled_camera: 该软件包包含从轮询相机驱动程序（例如，prosilica_camera）请求图像的接口。
- image_pipeline: 该超软件包包含一组软件包，处理来自相机驱动的原始图像，可以实现各种图像处理，如校准、畸变消除、立体视觉处理、深度图像处理等。以下软件包包含在该超软件包中（http://wiki.ros.org/image_pipeline）。
 - camera_calibration: 校准是建立3D环境与2D相机图像之间映射关系的重要工具之一。该软件包中的工具为ROS提供了单目和立体相机校准。
 - image_proc: 该软件包中的节点介于相机驱动节点和视觉处理节点之间，可用来处理相机参数校准，矫正失真的原始图像并将其转换为彩色图像。
 - depth_image_proc: 该软件包用于处理来自Kinect和3D视觉传感器的深度图像，由若干个节点和小节点组成。深度图像可以被小节点处理，生成点云数据。
 - stereo_image_proc: 该软件包的节点可以处理一对相机图像，如消除畸变。与image_proc软件包的不同之处在于stereo_image_proc可以处理两个相机的立体视觉，生成点云数据和视差图像。
 - image_rotate: 该软件包包含可以旋转输入图像的节点。
 - image_view: 这是ROS中查看消息话题的简单工具，还可以用于浏览立体图像和视差图像。
 - image_transport_plugins: 该软件包包含ROS图像传输的插件，用于发布和订阅ROS图像，并将图像压缩至不同程度，或将视频进行编码，从而减少传输带宽和传输延时。
 - laser_pipeline: 这是一组可以处理激光数据的软件包，例如滤波和转换3D笛卡尔数据并组装这些数据以形成点云。laser_pipeline集包含以下各种软件包：

- `laser_filters`: 该软件包用于对原始激光数据进行去噪，去除机器人本体范围内的激光点，还可以去除激光数据中的假数值。
- `laser_geometry`: 对激光数据进行去噪后，须将其距离值和角度值转换成笛卡儿坐标下的坐标值，同时还要考虑激光扫描仪的旋转和倾斜角。
- `laser_assembler`: 该软件包可以将激光扫描数据融合成3D点云数据或2.5D扫描数据。
- `perception_pcl`: 这是PCL-ROS接口的ROS软件包集。
- `vision_opencv`: 这是OpenCV-ROS接口的ROS软件包集。

10.3 在ROS中连接USB相机

我们可以在ROS中使用普通的相机或笔记本电脑的相机。总的来说，没有具体的ROS软件包来安装和使用相机。如果在Ubuntu/Linux系统中可以使用相机的话，也许在ROS中也可以使用。当连接上相机后，可以检查是否已经创建了/dev/videoX的设备文件，另外还可以用Cheese、VLC或其他类似的应用程序检查。可以在<https://help.ubuntu.com/community/Webcam>中找到Ubuntu支持的相机列表。

我们可以用下面的命令找到系统中存在的视频设备：

```
$ ls /dev/ | grep video
```

如果看到video的一个输出信息，你就可以确认USB相机可以正常使用。

确保Ubuntu支持这款相机后，我们就可以用下面的命令安装ROS的相机驱动程序usb_cam：

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

我们可以使用源码来安装最新版本的usb_cam软件包。该软件包在GitHub上的网址是：https://github.com/bosch-ros-pkg/usb_cam。

在usb_cam软件包中包含usb_cam_node节点，它是USB相机的驱动程序。运行该节点之前需要设置一些参数。我们可以使用配置的参数来运行这个ROS节点。下面的usb_cam-test.launch文件可以使用配套的参数来启动USB相机驱动程序：

```
<launch>
  <node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
    <param name="video_device" value="/dev/video0" />
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap"/>
  </node>
  <node name="image_view" pkg="image_view" type="image_view"
        respawn="false" output="screen">

    <remap from="image" to="/usb_cam/image_raw"/>
    <param name="autosize" value="true" />
  </node>
</launch>
```

该启动文件将使用usb_cam_node节点加载启动/dev/video0视频设备，分辨率为 640×480 。这里的像素格式是YUV (<https://en.wikipedia.org/wiki/YUV>)。初始化usb_cam_node节点后，会启动image_view节点来显示从相机获取的原始图像。我们可以使用下面的命令来运行上面的启动文件：

```
$ roslaunch usb_cam usb_cam-test.launch
```

我们将得到下面的输出信息，并看到图像窗口，如图10-1所示。

```
[ INFO] [1509310151.685693448]: Using transport "raw"
[ INFO] [1509310151.851576979]: using default calibration URL
[ INFO] [1509310151.851731568]: camera calibration URL: file:///home/jcacace/.ros/camera_info/head_camera.yaml
[ INFO] [1509310151.851937275]: Unable to open camera calibration file [/home/jcacace/.ros/camera_info/head_camera.yaml]
[ WARN] [1509310151.852013709]: Camera calibration file /home/jcacace/.ros/camera_info/head_camera.yaml not found.
[ INFO] [1509310151.852111773]: Starting 'head_camera' (/dev/video0) at 640x480 via mmap (yuyv) at 30 FPS
[ WARN] [1509310152.108112434]: unknown control 'focus_auto'
```



图10-1 使用图像查看工具查看USB相机输出

图10-2显示了各节点发布的话题。这里有raw、compressed和theora编解码等话题。

```
/image_view/output
/image_view/parameter_descriptions
/image_view/parameter_updates
/rosout
/rosout_agg
/usb_cam/camera_info
/usb_cam/image_raw
/usb_cam/image_raw/compressed
/usb_cam/image_raw/compressed/parameter_descriptions
/usb_cam/image_raw/compressed/parameter_updates
/usb_cam/image_raw/compressedDepth
/usb_cam/image_raw/compressedDepth/parameter_descriptions
/usb_cam/image_raw/compressedDepth/parameter_updates
/usb_cam/image_raw/theora
/usb_cam/image_raw/theora/parameter_descriptions
/usb_cam/image_raw/theora/parameter_updates
```

图10-2 USB相机输出的话题列表

我们可以使用下面的命令在另一个窗口中显示图像：

```
$ rosrun image_view image_view image:=/usb_cam/image_raw
```

就如在话题列表中看到的，由于安装了image_transport软件包，因此可以使用多种方式来发布图像，即未压缩和压缩的。当通过网络将图像发送到其他ROS节点或将该话题的视频数据保存在日志文件bagfiles中时，压缩格式将非常有用，因为占用的硬盘空间很小。为了从远程主机或单个日志文件bagfile使用日志文件保存的压缩图像，我们需要用image_transport软件包中的republish节点以未压缩的格式重新发布图像。

```
$ rosrun image_transport republish [input format] in:=<in_topic_base>  
[output format] out:=<out_topic>
```

例如：

```
$ rosrun image_transport republish compressed in:=/usb_cam/image_raw  
[output format] out:=/usb_cam/image_raw/republished
```

i 在前面的示例中需要注意：我们使用原始图像的话题作为输入（/usb_cam/image_raw），而不是它的压缩图像话题（/usb_cam/image_raw/compressed）。

现在我们已经学会了如何从相机中获取图像，下面开始学习如何校准相机。

10.4 ROS与相机校准

与其他传感器一样，相机也需要校准以校正相机内部参数造成的图像失真，并根据相机坐标确定世界坐标。

导致图像畸变的主要原因是径向畸变和切向畸变。使用相机校准算法，我们就可以对这些参数进行建模，还可以通过计算相机校准矩阵，完成相机坐标到真实世界坐标的转换，该矩阵包含相机焦距和投影中心。

可以用经典的黑白棋盘、对称圆形图案、不对称圆形图案等方法对相机进行校准。根据不同的图案，我们用不同的方程求得校准参数。我们通过校准工具检测图案，并将检测到的每个图案视为一个新方程。当校准工具检测到足够多的图案时，就可以计算出相机的最终参数。

ROS提供名为camera_calibration的软件包（http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration）来进行相机校准。该软件包是图像处理软件包集（image_pipeline）的一部分，可以对单目相机、立体相机进行校准，甚至可以对3D传感器（如Kinect、华硕Xtion Pro）进行校准。

校准之前，首先下载ROS维基页面中提到的黑白棋盘格图案，然后打印出来并粘贴在一个纸板上。这将是用于校准的图案，该校准板有 8×6 个校准点，每个方格为108mm宽。

运行了usb_cam启动文件加载相机启动后，我们就可以运行ROS相机校准节点，并用话题/usb_cam/image_raw的原始图像进行校准。下面的命令指定必备的参数来运行校准节点：

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

该命令会弹出一个校准窗口，当将校准板对准相机时，开始检测，我们将看到如图10-3所示信息。

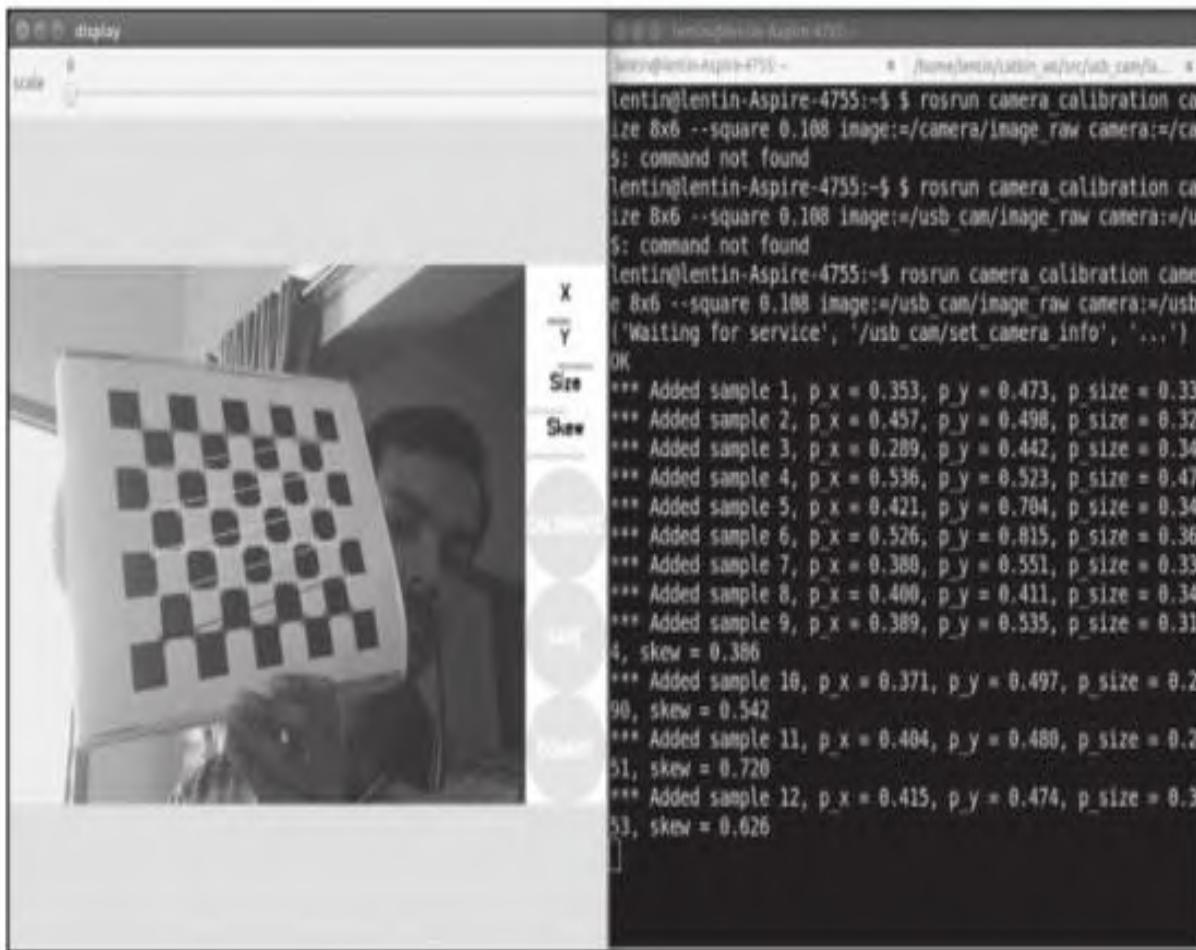


图10-3 ROS相机校准

沿着X方向和Y方向移动校准板。当校准节点获得足够多的采样点时，窗口上的CALIBRATE按钮会处于激活状态。当我们按下CALIBRATE按钮，将使用这些采样点计算相机参数。计算过程需要花一些时间。计算完成后，窗口上的SAVE和COMMIT按钮将被激活，如图10-4所示。如果我们按下SAVE按钮，校准参数将被保存在/tmp文件夹下的文件

中。当我们按下COMMIT按钮，校准参数将被保存到`./ros/camera_info/head_camera.yaml`文件中。

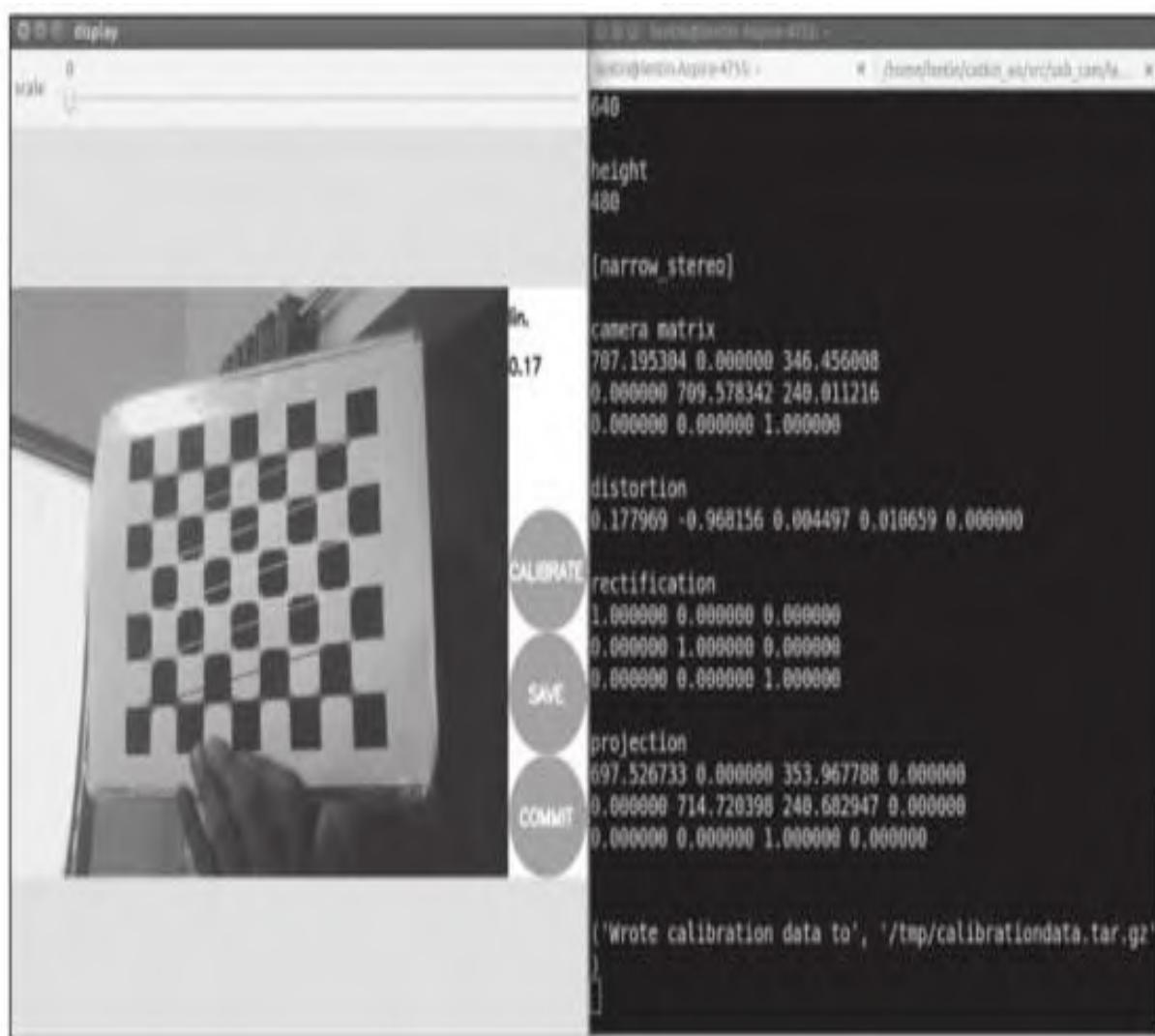


图10-4 生成相机校准参数文件

重启相机节点后，会看到相机节点加载了YAML校准参数文件。生成的校准参数文件如下所示：

```
image_width: 640
image_height: 480
camera_name: head_camera
camera_matrix:
rows: 3
cols: 3
data: [707.1953043273086, 0, 346.4560078627374, 0, 709.5783421541863,
240.0112155124814, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
rows: 1
cols: 5
data: [0.1779688561999974, -0.9681558538432319, 0.004497434720139909,
0.0106588921249554, 0]
rectification_matrix:
rows: 3
cols: 3
data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:

rows: 3
cols: 4
data: [697.5267333984375, 0, 353.9677879190494, 0, 0, 714.7203979492188,
240.6829465337159, 0, 0, 0, 1, 0]
```

10.4.1 使用cv_bridge在ROS和OpenCV之间转换图像

本节，我们将学习如何在ROS图像消息（`sensor_msgs/Image`）和OpenCV图像数据（`cv::Mat`）之间进行转换。这种转换主要依赖ROS中的`cv_bridge`软件包，它是`vision_opencv`软件包集的一部分。在`cv_bridge`软件包中，`CvBridge`库来执行这种转换。我们可以在代码中用`CvBridge`库执行这种转换。图10-5显示了如何在ROS和OpenCV之间完成转换。

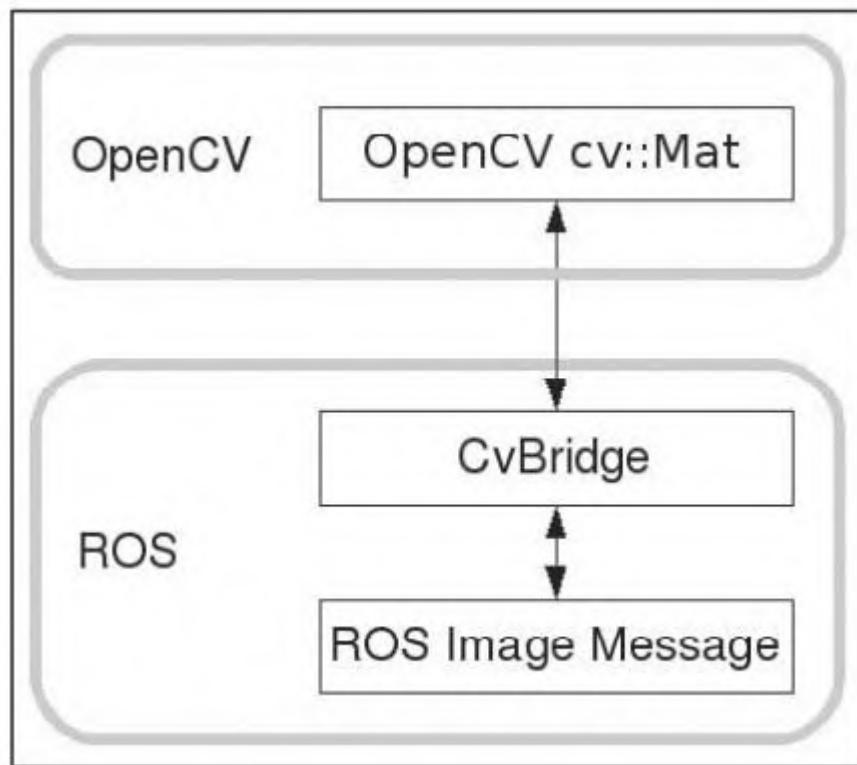


图10-5 用CvBridge库完成图像格式转换

在这里，`CvBridge`库充当ROS消息与OpenCV图像互相转换的桥梁，我们将通过下面的例子理解如何在ROS和OpenCV之间进行转换的。

10.4.2 使用ROS和OpenCV进行图像处理

在本节中，我们将看到一个示例，该示例使用cv_bridge从相机中获取图像并且使用OpenCV API来转换和处理图像。下面是该示例的工作流程：

- 1) 从相机节点的/usb_cam/image_raw (sensor_msgs/Image) 话题订阅图像。
- 2) 用CvBridge将ROS图像转换为OpenCV图像类型。
- 3) 用OpenCV的API处理图像，找到图像的边缘。
- 4) 将边缘检测的OpenCV图像转换为ROS图像消息并将其发布到/edge_detector/processed_image话题上。

下面来一步一步地介绍该示例的操作步骤：

步骤1：创建一个ROS软件包

可以从随书提供的代码中找到已创建好的cv_bridge_tutorial_pkg软件包，也可以用下面的命令创建一个新软件包：

```
$ catkin_create_pkg cv_bridge_tutorial_pkg cv_bridge image_transport roscpp  
sensor_msgs std_msgs
```

该软件包主要依赖cv_bridge、image_transport和sensor_msgs。

步骤2：创建源码文件

可以从cv_bridge_tutorial_pkg/src文件夹下获取本实验的源码文件sample_cv_bridge_node.cpp。

步骤3：代码说明

下面是完整的代码解释：

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

此段代码用image_transport软件包发布和订阅ROS中的图像。

```
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
```

这两个头文件包含了CvBridge类以及与图像编码相关的函数。

```
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
```

这两个头文件包含了OpenCV图像处理模块和GUI模块，分别提供图像处理和GUI的API。

```
image_transport::ImageTransport it_;
public:
    Edge_Detector()
        : it_(nh_)
    {
        // 订阅输入视频和发布输出视频
        image_sub_ = it_.subscribe("/usb_cam/image_raw", 1,
            &ImageConverter::imageCb, this);

        image_pub_ = it_.advertise("/edge_detector/processed_image", 1);
```

我们来仔细研究这行代码image_transport::ImageTransport it_。这行代码创建了一个ImageTransport对象实例，用于发布和订阅ROS图像。ImageTransport的API信息在下面介绍。

使用image_transport发布和订阅图像

ROS图像的传输与ROS中发布者和订阅者非常相似，它发布和订阅图像，并且带有相机信息。虽然我们可以使用`ros::Publisher`来发布图像数据，但是使用图像传输是发送图像数据更有效的方式。

图像传输的API是由`image_transport`软件包提供的。使用这些API，我们可以用不同的压缩格式来传输图像。例如，我们可以传输未压缩图像，也可以传输JPEG/PNG压缩格式图像；或者在单独话题中，用Theora压缩格式来传输图像；还可以通过插件添加其他不同的传输格式。默认情况下，我们可以看到用压缩格式和Theora格式的传输。

```
image_transport::ImageTransport it_;
```

在这行代码中，我们创建了一个`ImageTransport`类的实例。

```
image_transport::Subscriber image_sub_;
image_transport::Publisher image_pub_;
```

在这行代码中，用`image_transport`对象声明了订阅者和发布者对象，用于订阅和发布图像数据。

```
image_sub_ = it_.subscribe("/usb_cam/image_raw", 1,
                           &ImageConverter::imageCb, this);
image_pub_ = it_.advertise("/edge_detector/processed_image", 1);
```

上面的代码显示了如何实现订阅和发布图像数据。

```
    cv::namedWindow(OPENCV_WINDOW);
}
~Edge_Detector()
{
    cv::destroyWindow(OPENCV_WINDOW);
}
```

上面的代码中cv::namedWindow()是一个OpenCV函数，用于创建GUI窗口，显示图像。该函数的参数是GUI的窗口名。在析构函数中，根据窗口名来销毁GUI窗口。

使用cv_bridge将OpenCV转换为ROS图像

这是一个图像的回调函数，它用CvBridge的API将ROS图像消息转换为OpenCV下的cv::Mat类型数据。下面是在ROS和OpenCV之间进行转换的代码：

```
void imageCb(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;
    namespace enc = sensor_msgs::image_encodings;

    try
    {
        cv_ptr = cv_bridge::toCvCopy(msg,
sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("cv_bridge exception: %s", e.what());
    }

    return;
}
```

要想启动CvBridge，我们首先需要创建一个CvImage的实例。下面的代码即创建了一个CvImage的指针：

```
cv_bridge::CvImagePtr cv_ptr;
```

CvImage是由cv_bridge提供的一个类，由OpenCV图像、编码方式、ROS消息头等信息组成。利用CvImage，可以很方便地在ROS图像和OpenCV之间完成转换。

```
cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
```

我们可以用两种方式来处理ROS图像消息：使用图像的副本或共享图像数据。使用图像副本时，我们可以处理图像。但是如果使用共享

图像的指针的话，我们就不能修改图像数据。我们可以用`toCvCopy()`函数创建ROS图像副本，用`toCvShare()`函数得到图像指针。在这些函数中，我们需要提到ROS消息和编码类型。

```
if (cv_ptr->image.rows > 400 && cv_ptr->image.cols > 600){  
    detect_edges(cv_ptr->image);  
    image_pub_.publish(cv_ptr->toImageMsg());  
}
```

这段代码从`CvImage`实例中提取图像及其属性，并从该实例中访问`cv::Mat`对象。这段代码只检查图像的行和列是否在特定范围内。如果在特定范围内，则调用另一个方法`detect_edges (cv::Mat)`。该方法处理图像并显示边缘检测的结果图像。

```
image_pub_.publish(cv_ptr->toImageMsg());
```

上面这行代码在转换为ROS图像消息后发布边缘检测图像。在这里，我们用`toImageMsg()`函数将`CvImage`实例转换为ROS图像消息。

图像边缘检测

将ROS图像转换为OpenCV类型后，将调用`detect_edges (cv::Mat)`函数进行图像的边缘检测，具体使用下述OpenCV的内置函数：

```
cv::cvtColor( img, src_gray, CV_BGR2GRAY );  
cv::blur( src_gray, detected_edges, cv::Size(3,3) );  
cv::Canny( detected_edges, detected_edges, lowThreshold,  
lowThreshold*ratio, kernel_size );
```

这里，调用`cvtColor()`函数将RGB图像转换为灰色图像，调用`cv::blur()`函数对图像进行模糊处理。最后，用Canny边缘检测器提取

图像中的边缘。

显示原始图像和边缘检测图像

我们用OpenCV的imshow()函数显示图像数据，该函数包含窗口名称和图像名称：

```
cv::imshow(OPENCV_WINDOW, img);
cv::imshow(OPENCV_WINDOW_1, dst);
cv::waitKey(3);
```

步骤4：编辑CMakeLists.txt文件

下面是CMakeLists.txt文件的内容。因为本例需要OpenCV的支持，所以要包含OpenCV头文件的路径，并将源代码与OpenCV库路径连接起来：

```
include_directories(
    ${catkin_INCLUDE_DIRS}
    ${OpenCV_INCLUDE_DIRS}
)

add_executable(sample_cv_bridge_node src/sample_cv_bridge_node.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(sample_cv_bridge_node
    ${catkin_LIBRARIES}
    ${OpenCV_LIBRARIES}
)
```

步骤5：编译并运行示例

使用catkin_make编译软件包后，我们就可以使用下面的命令来运行这个节点：

1) 启动相机节点：

```
$ roslaunch usb_cam usb_cam-test.launch
```

2) 运行cv_bridge节点：

```
$ rosrun cv_bridge_tutorial_pkgs sample_cv_bridge_node
```

3) 如果一切正常的话，我们将看到两个窗口，如图10-6所示。第一个窗口显示原始图像，第二个窗口显示处理后的边缘检测图像。



图10-6 原始图像和边缘检测图像

10.5 在ROS中连接Kinect与华硕Xtion Pro

目前为止，我们使用过的相机都只能提供周围环境的2D信息。为了获得关于环境的3D信息，必须使用3D视觉传感器或测距仪，例如激光雷达。我们在本章讨论的3D视觉传感器有Kinect、华硕Xtion Pro、英特尔Real Sense和Kokuyo激光雷达。

我们首先讨论的两个传感器是Kinect和华硕Xtion Pro，如图10-7所示。这两个设备都运行在Linux系统中，并且需要OpenNI（Open Natural Interaction）驱动库的支持。OpenNI是3D视觉设备和应用软件之间的中间件。OpenNI驱动已经被集成到ROS中，我们可以使用下面的命令来安装这些驱动程序。这些软件包有助于连接OpenNI支持的设备，例如Kinect和华硕Xtion Pro：



图10-7 上：Kinect；下：华硕Xtion Pro

```
$ sudo apt-get install ros-kinetic-openni-launch ros-kinetic-openni2-launch
```

上面的命令将安装OpenNI驱动程序，并加载RGB/depth视频流的启动文件。成功安装这些软件包后，我们可以使用下面的命令来启动驱动程序：

```
$ roslaunch openni2_launch openni2.launch
```

该启动文件用ROS小节点从设备获取原始数据，并转换为有用的数据，例如3D点云、视差图像、深度图像、RGB图像等。

除了OpenNI驱动程序，还有另一个叫lib-freenect的驱动程序。该驱动程序常用的启动文件位于rgbd_launch软件包中。这个软件包包含freenect和openni驱动程序常用的启动文件。

可以用RViz显示由OpenNI ROS驱动程序生成的点云。用下面的命令启动RViz：

```
$ rosrun rviz rviz
```

将固定坐标系设置为/camera_depth_optical_frame，然后添加PointCloud2显示页，并将其话题名设置为/camera/depth/points。图10-8显示了来自红外相机的未配准点云，也就是说，它也许与RGB相机完全匹配，且仅使用深度相机来生成点云。

可以使用下面的命令来启动Dynamic Reconfigure（动态配置）图形界面工具（如图10-9），然后启用点云的配准：

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

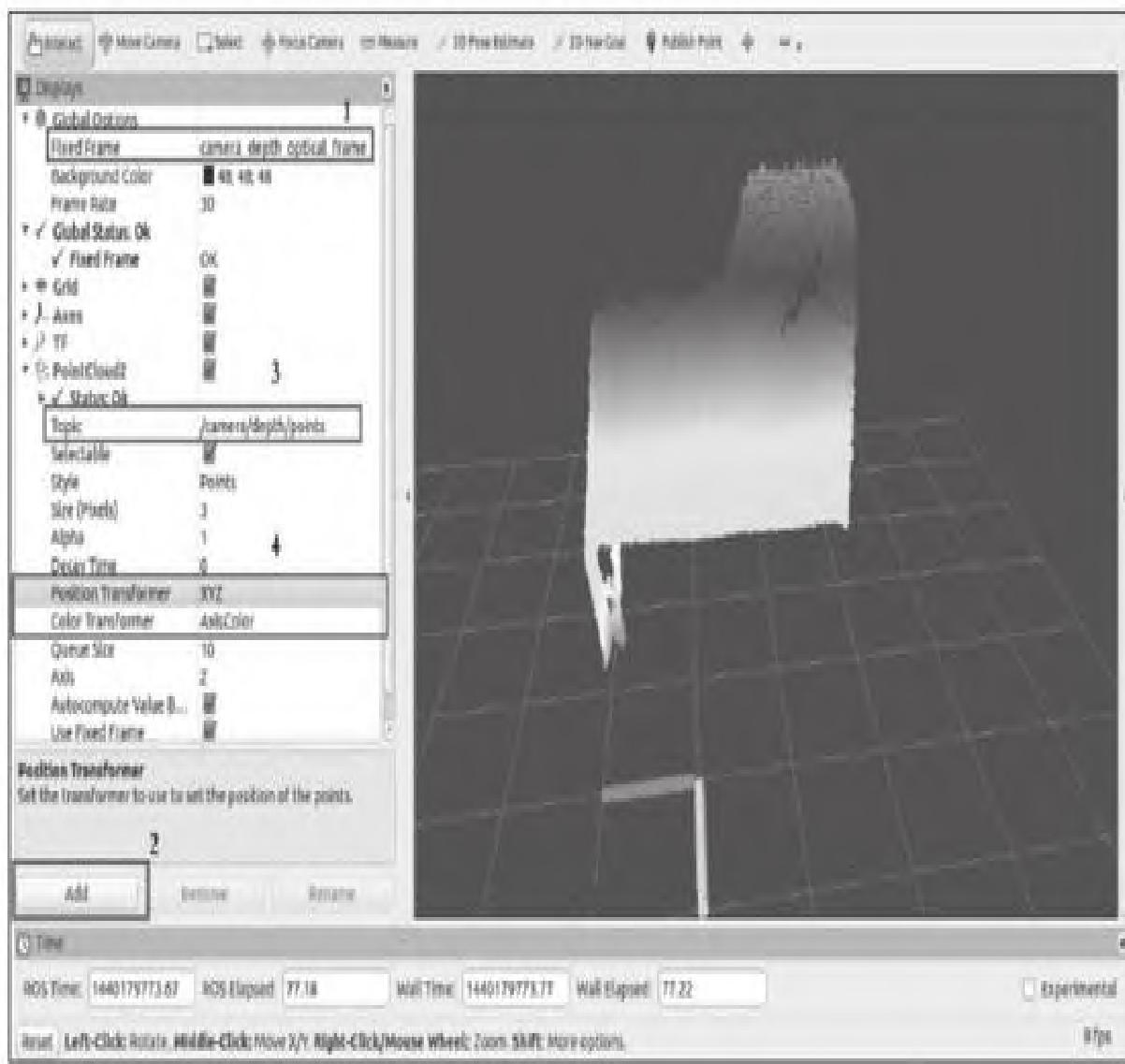


图10-8 RViz中显示未配准的点云

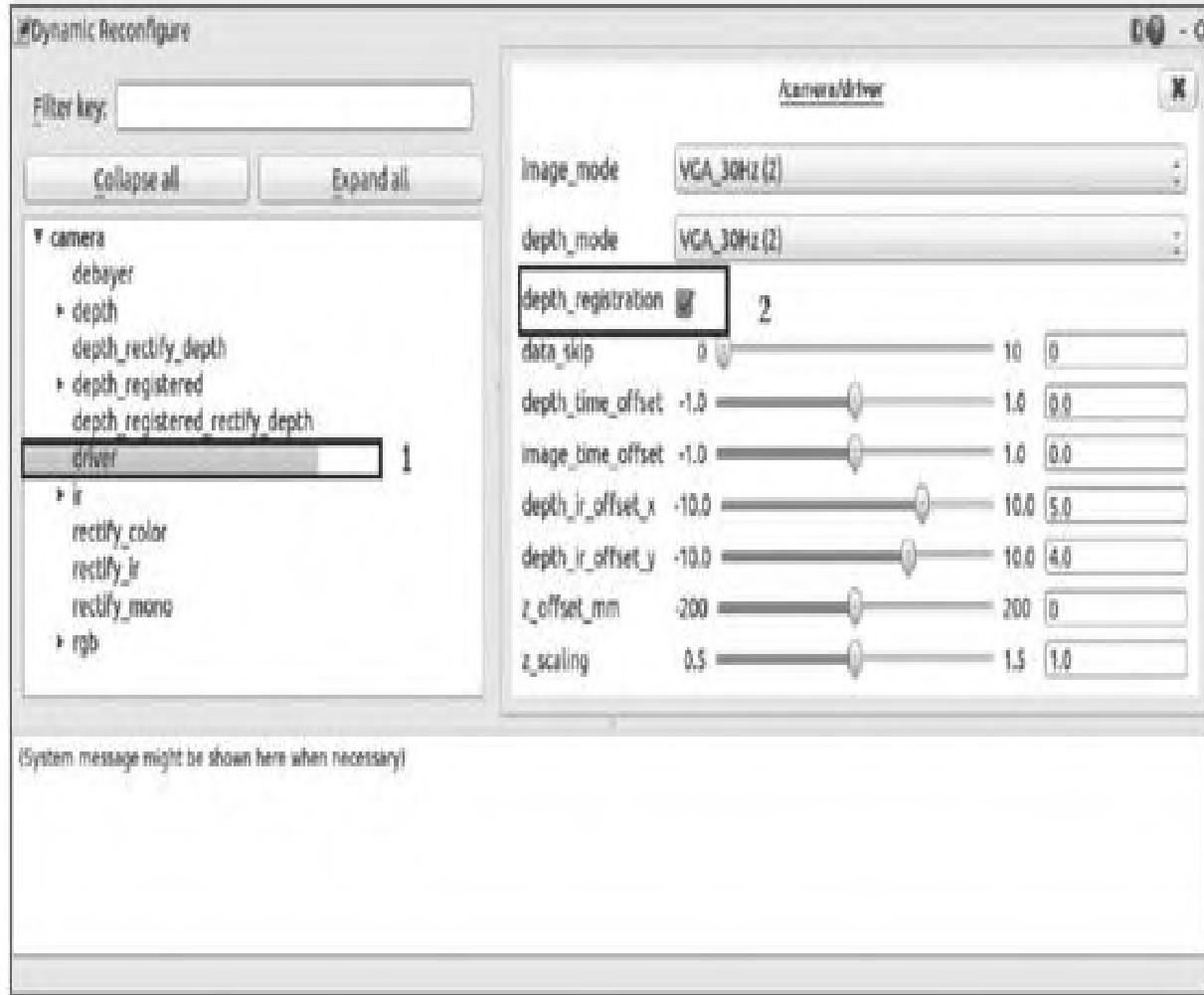


图10-9 Dynamic Reconfigure（动态配置）图形界面工具

点击camera|drive，选中depth_registration。在RViz中将点云话题切换成/camera/depth_registered/points，将Color Transformer切换到RGB8。我们将在RViz中看到图10-10所示的配准点云数据。配准点云从深度相机和RGB相机同时获取信息来生成点云。

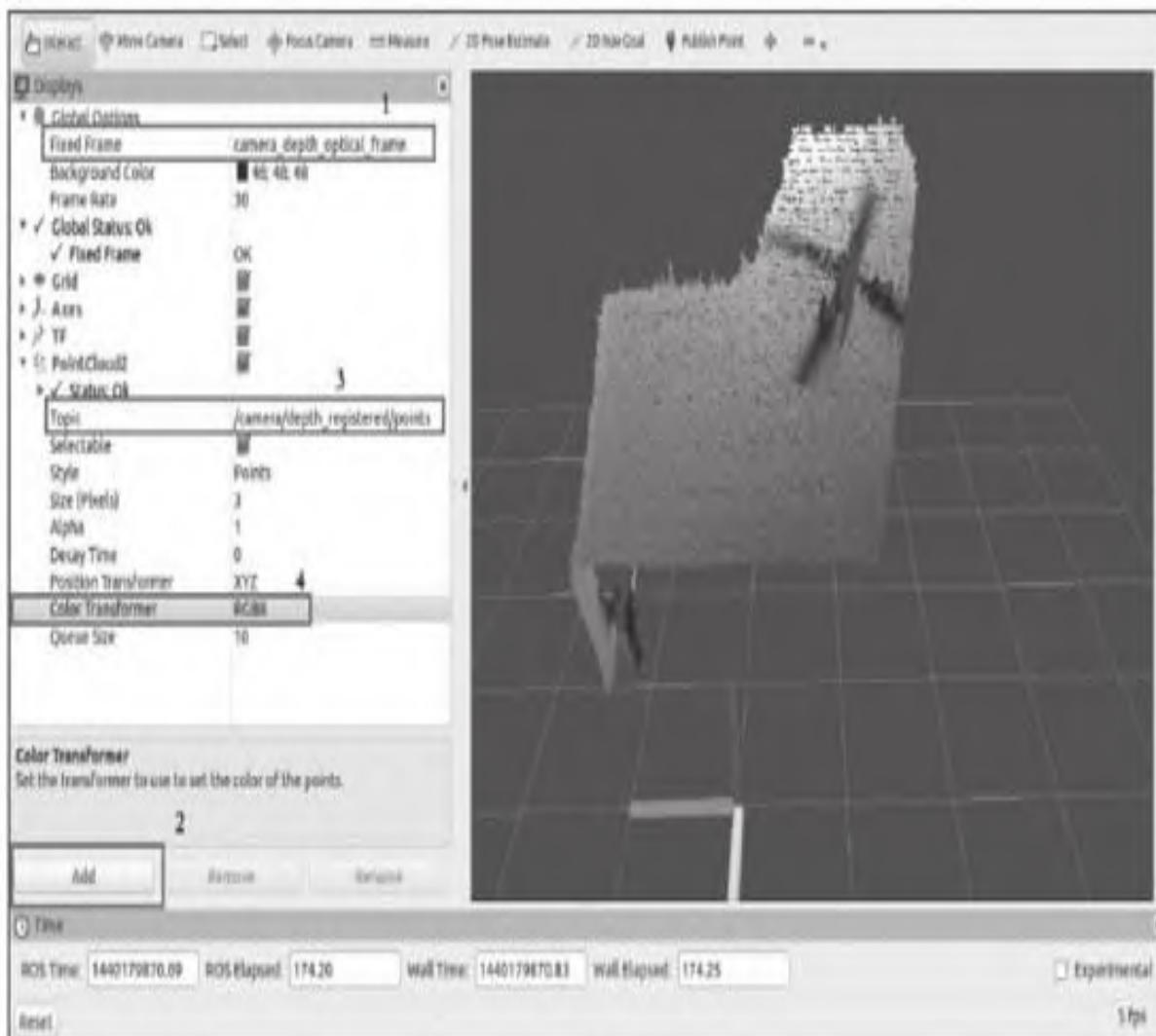


图 10-10 显示配准点云

10.6 将英特尔Real Sense相机与ROS连接

Real Sense是来自英特尔的一款新的3D深度传感器。到目前为止，这款传感器已经发布了不同型号（如F200、R200、SR30...）。为了将ROS中与RealSense传感器相连接，我们首先必须安装librealsense库。

用下面的命令从地址

<https://github.com/IntelRealSense/librealsense> 下载RealSense库：

```
$ git clone https://github.com/IntelRealSense/librealsense.git
```

然后按照以下步骤进行安装：

1) 安装libudev-dev, pkg-config和libgtk-3:

```
$ sudo apt-get install libudev-dev pkg-config libgtk-3-dev
```

2) 安装g1fw3:

```
$ sudo apt-get install libg1fw3-dev
```

3) 进入librealsense根文件夹，运行下面命令：

```
$ mkdir build && cd build  
$ cmake ..  
$ make  
$ sudo make install
```

安装了RealSense库之后，我们可以安装ROS软件包来启动传感器数据流。我们可以通过下面的命令从Ubuntu/Debian软件包管理器安装：

```
$ sudo make install ros-kinetic-realsense
```

或者我们也可以从Git库中直接下载软件包，然后编译工作区：

```
$ git clone https://github.com/intel-ros/realsense.git
```

现在，我们可以使用本例中的启动文件来启动传感器，然后打开RViz（图10-11）查看realsense的颜色和深度数据：

```
roslaunch realsense_camera sr300_nodelet_rgbd.launch
```

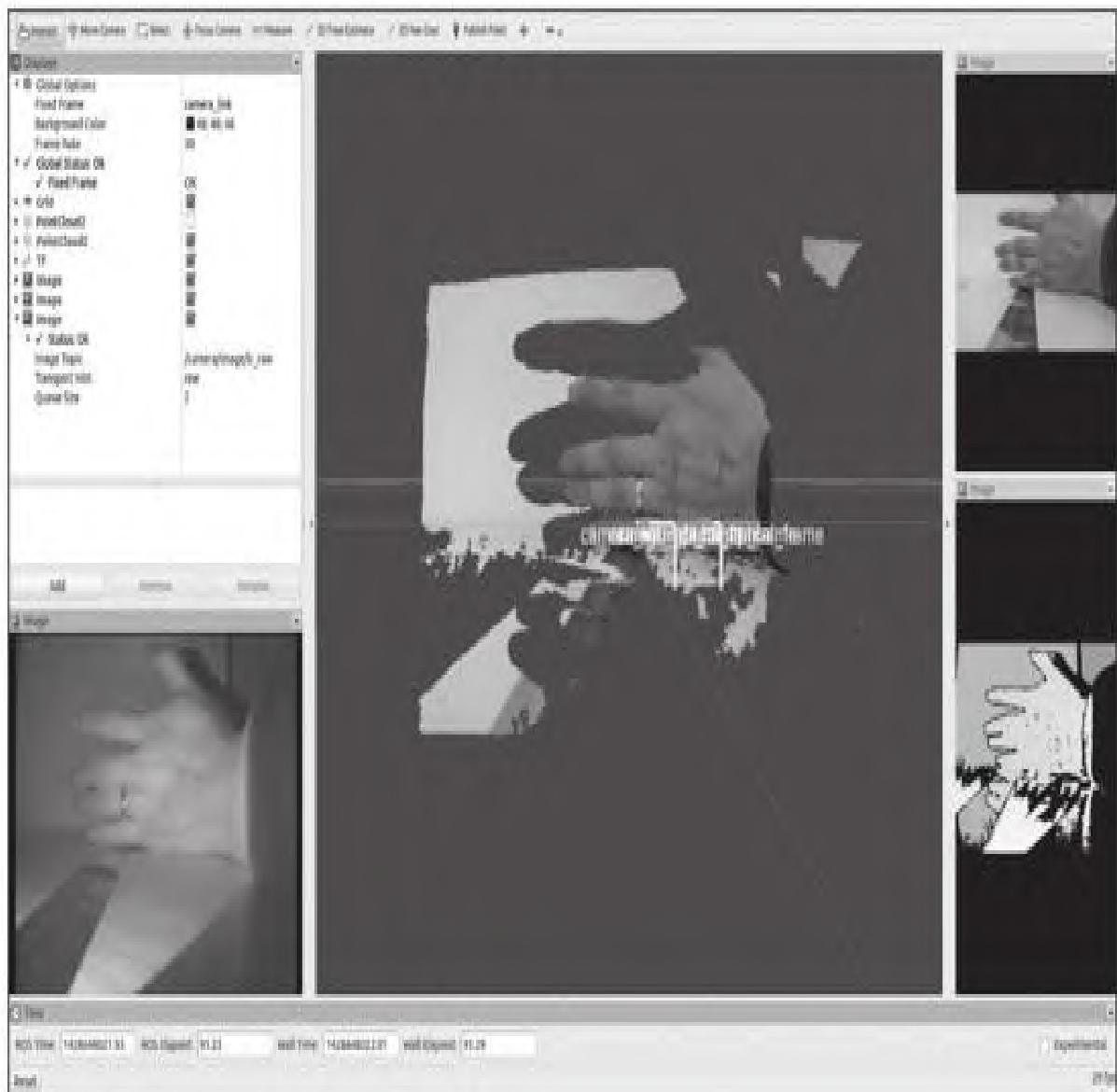


图10-11 在RViz中查看英特尔Real Sense数据

下面这些是Real Sense驱动生成的话题：

```

sensor_msgs::PointCloud2
/camera/depth/points           point cloud without RGB
/camera/depth_registered/points point cloud with RGB

sensor_msgs::Image
/camera/image/rgb_raw          raw image for RGB sensor
/camera/image/depth_raw         raw image for depth sensor
/camera/image/ir_raw            raw image for infrared sensor

```

使用点云转换激光扫描的软件包

3D视觉传感器的一个重要应用就是模拟激光雷达的功能。在使用自主导航算法时，我们需要用到激光雷达的数据，例如SLAM算法。我们可以用3D视觉传感器模拟一个假的激光雷达。我们可以将一层点云数据/深度数据转换成激光雷达测距数据。在ROS中，我们有一系列软件包可以完成从点云到激光雷达数据的转化：

- `depthimage_to_laserscan`: 该软件包可以从视觉传感器获取深度数据，还可以根据提供的参数生成2D激光雷达扫描数据。该节点的输入参数是深度数据和相机参数校准参数。转换成激光扫描数据后，它将在`/scan`话题上发布激光扫描数据。这些节点参数包括`scan_height`、`scan_time`、`range_min`、`range_max`和输出帧的ID。该软件包的官方ROS维基页面可以参考网址：http://wiki.ros.org/depthimage_to_laserscan。
- `pointcloud_to_laserscan`: 上面的软件包得到深度图像，该软件包有所不同，它可以将点云数据转换成2D激光扫描数据。该软件包的官方ROS维基页面可以参考网址：http://wiki.ros.org/pointcloud_to_laserscan。

上述第一个软件包适用于普通的应用。然而，如果传感器安装时有一定倾斜角度的话，最好还是使用第二个软件包。第一个软件包的处理流程比第二个软件包少。在此，我们用`depthimage_to_laserscan`软件包来转换激光扫描数据。我们可以用下面的命令安装`depthimage_to_laserscan`和`pointcloud_to_laserscan`:

```
$ sudo apt-get install ros-kinetic-depthimage-to-laserscan ros-kinetic-pointcloud-to-laserscan
```

我们可以用下面的软件包从OpenNI设备获取深度数据，再将其转换为2D激光扫描数据。

创建一个用于执行转换的软件包：

```
$ catkin_create_pkg fake_laser_pkg depthimage_to_laserscan nodelet roscpp
```

创建一个launch文件夹，并在该文件夹中创建一个start_laser.launch启动文件。可以从fake_laser_pkg/launch文件夹中找到这个软件包和文件：

```
<launch>
  <!-- "camera" should uniquely identify the device. All topics are
       pushed down
       into the "camera" namespace, and it is prepended to tf frame
       ids. -->
  <arg name="camera"      default="camera"/>
  <arg name="publish_tf" default="true"/>

  . . .
  . . .

  <group if="$(arg scan_processing)">
    <node pkg="nodelet" type="nodelet"      name="depthimage_to_laserscan"
          args="load      depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg
          camera)/$(arg camera)_nodelet_manager">
      <!-- Pixel rows to use to generate the laserscan. For each
          column, the scan will return the minimum value for those pixels
```

```
centered vertically in the image. -->
    <param name="scan_height" value="10"/>
    <param name="output_frame_id" value="/$(arg
camera)_depth_frame"/>
    <param name="range_min" value="0.45"/>
    <remap from="image" to="$(arg camera)/$(arg
depth)/image_raw"/>
    <remap from="scan" to="$(arg scan_topic)"/>
    .
    .
</launch>
```

下面这段代码会启动一个小节点将深度数据转换成激光扫描数据：

```
<node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg
camera)/$(arg camera)_nodelet_manager">
```

启动这个文件后，我们就可以在RViz中查看激光扫描数据。

用下面的命令来启动这个文件：

```
$ rosrun fake_laser_pkg start_laser.launch
```

我们可以RViz查看数据，如图10-12所示。

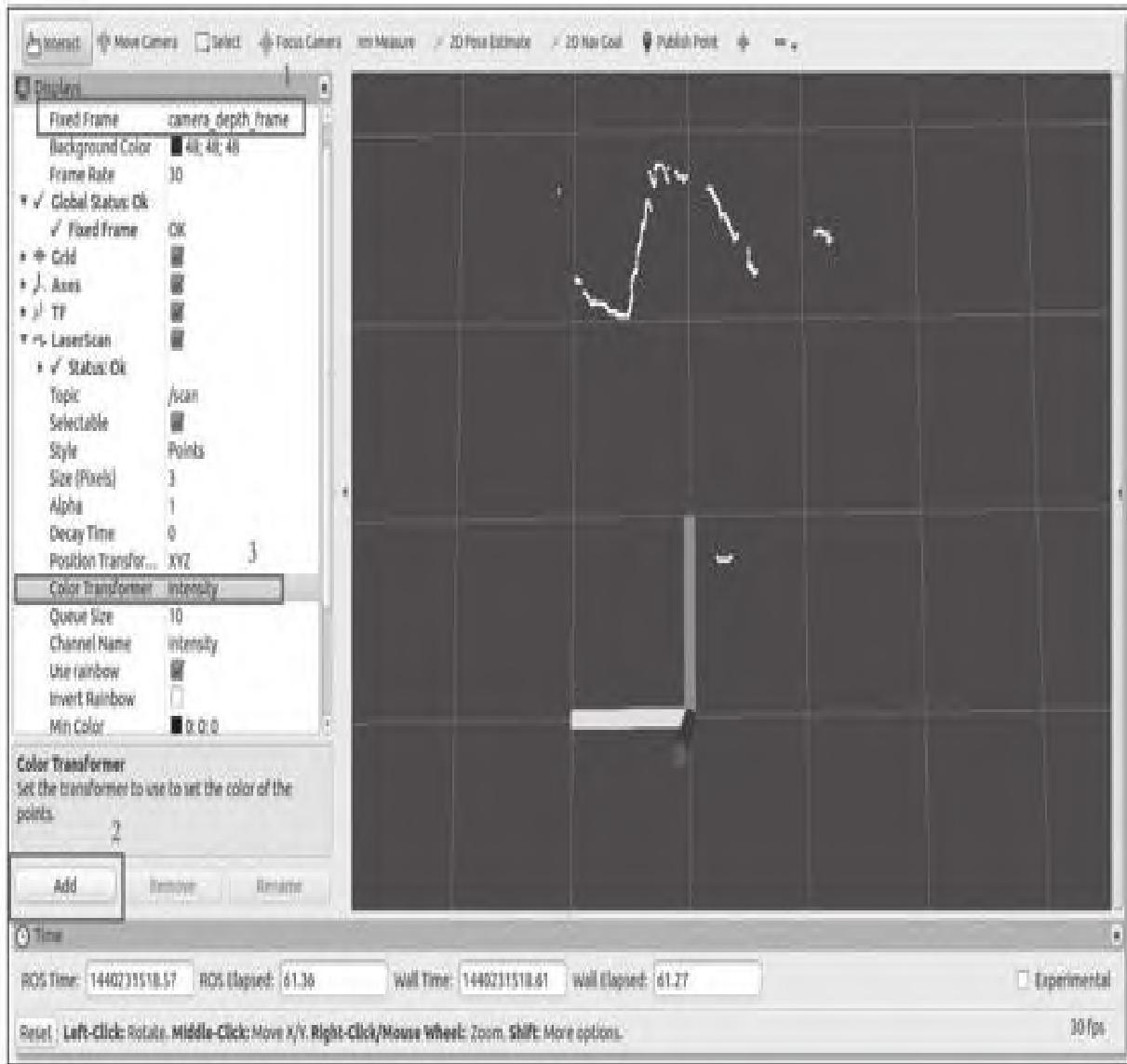


图10-12 在RViz中查看激光扫描数据

将Fixed Frame设置为camera_depth_frame，然后点击Add LaserScan，设置话题名为scan。我们可以在视图窗口看到激光扫描数据。

10.7 在ROS中连接Hokuyo激光雷达

在ROS中我们可以使用不同测距范围的激光雷达，如图10-13所示。市面上最受欢迎的激光雷达之一是Hokuyo（<http://www.robotshop.com/en/hokuyo-utm-03lx-laser-scanning-rangefinder.html>）。



图10-13 不同系列的Hokuyo激光雷达

一种常用的Hokuyo激光雷达型号是UTM-30LX，如图10-14所示。这个传感器测距快且准确，非常适合机器人应用。该设备有一个USB 2.0接口用于通讯，并且具有高达30米范围内的毫米分辨率，扫描的角度范围约是270度。



图10-14 Hokuyo UTM-30LX激光雷达

在ROS中已有支持这些激光雷达的驱动程序，
urg_node (http://wiki.ros.org/urg_node) 是其中之一。

通过下面的命令，可以安装这个软件包：

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

当设备连接到Ubuntu系统时，会创建一个名为ttyACMx的设备。可以在终端使用dmesg命令来找到该设备名。使用下面的命令更改USB设备权限：

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

用下面的hokuyo_start.launch启动文件启动激光扫描设备：

```
<launch>
  <node name="urg_node" pkg="urg_node" type="urg_node" output="screen">
    <param name="serial_port" value="/dev/ttyACM0"/>
    <param name="frame_id" value="laser"/>
    <param name="angle_min" value="-1.5707963"/>
    <param name="angle_max" value="1.5707963"/>
  </node>
  <node name="rviz" pkg="rviz" type="rviz" respawn="false" output="screen">
    args="-d $(find hokuyo_node)/hokuyo_test.vcg"
  </node>
</launch>
```

该启动文件启动/dev/ttyACM0设备节点，并获取激光扫描数据。可在RViz窗口中查看到激光扫描数据，如图10-15所示。

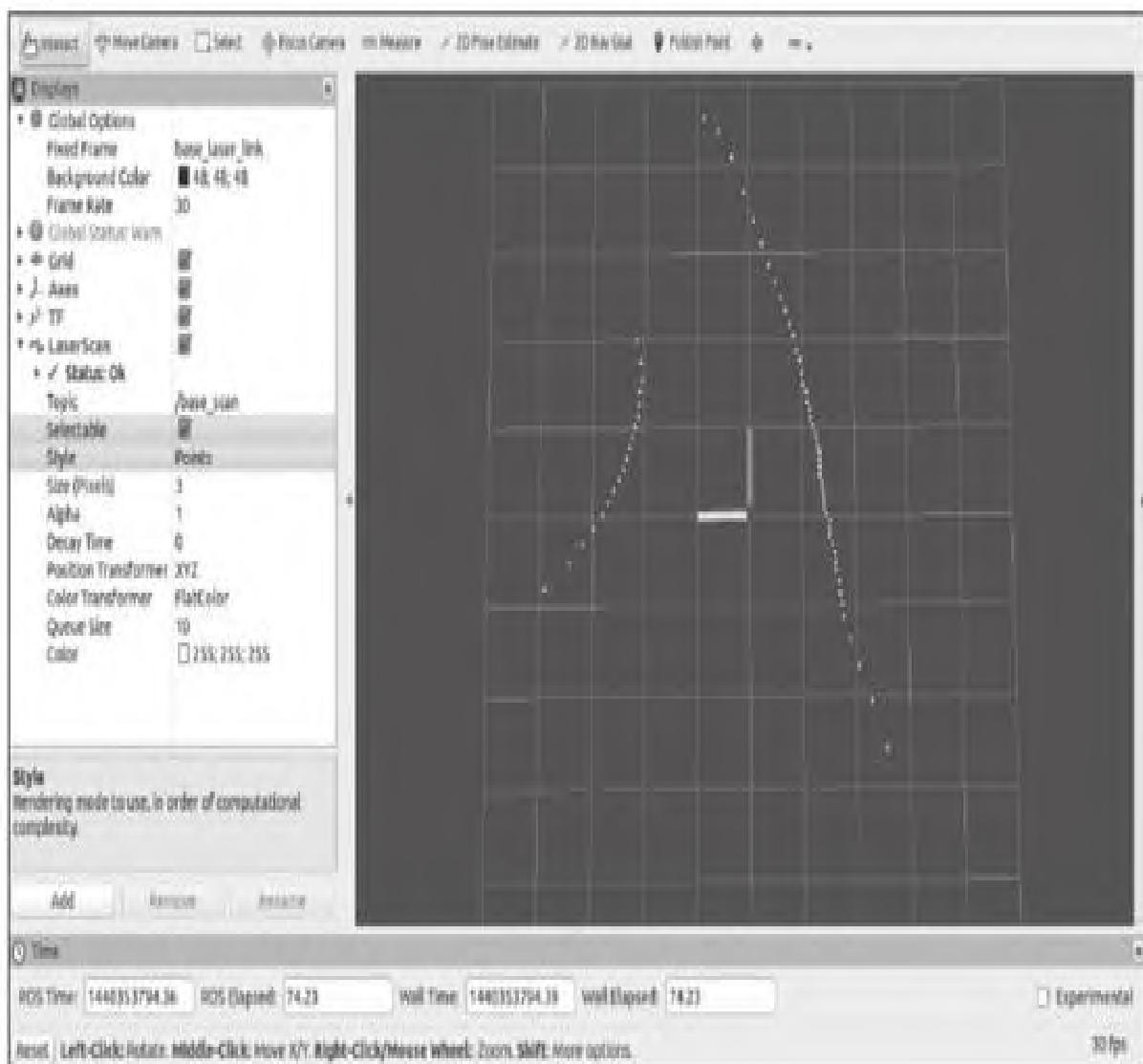


图10-15 在RViz中查看Hokuyo激光扫描数据

10.8 处理点云数据

我们可以处理来自Kinect或其他3D传感器的点云数据，然后用于执行各种任务，如3D物体检测和识别、避障、3D建模等。本节，我们将了解一些基本功能：使用PCL库及其ROS接口。我们将讨论下面这些示例：

- 如何在ROS中发布点云
- 如何订阅和处理点云
- 如何将点云数据写入PCD文件
- 如何从PCD文件读取和发布点云

10.8.1 如何发布点云

本例中，我们将了解如何用sensor_msgs/PointCloud2类型的消息发布点云数据。本程序用PCL中的API来处理和创建点云，并将PCL点云数据转换为PointCloud2消息类型。

可以从pcl_ros_tutorial/src文件夹中找到这段示例代码
pcl_publisher.cpp：

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

// 点云头文件
#include <pcl/point_cloud.h>
// 头文件中包含了 PCL 与 ROS 直接互相转换的函数
#include <pcl_conversions/pcl_conversions.h>

// sensor_msgs 中包含点云头文件
#include <sensor_msgs/PointCloud2.h>

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

    ROS_INFO("Started PCL publishing node");

    ros::NodeHandle nh;

    // 为点云创建发布者对象

    ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
    ("pcl_output", 1);

    // 创建点云对象
    pcl::PointCloud<pcl::PointXYZ> cloud;

    // 创建一个 sensor_msg 点云对象

    sensor_msgs::PointCloud2 output;

    // 插入点云数据
    cloud.width = 50000;
    cloud.height = 2;
    cloud.points.resize(cloud.width * cloud.height);

    // 在点云中插入随机点

    for (size_t i = 0; i < cloud.points.size (); ++i)
    {
        cloud.points[i].x = 512 * rand () / (RAND_MAX + 1.0f);
        cloud.points[i].y = 512 * rand () / (RAND_MAX + 1.0f);
        cloud.points[i].z = 512 * rand () / (RAND_MAX + 1.0f);
    }

    // 将点云数据转换为 ROS 消息
    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "point_cloud";

    ros::Rate loop_rate(1);
```

```
while (ros::ok())
{
    // 发布点云数据
    pcl_pub.publish(output);
    ros::spinOnce();
    loop_rate.sleep();
}

return 0;
}
```

像下面这样，创建PCL点云：

```
// 创建一个点云对象
pcl::PointCloud<pcl::PointXYZ> cloud;
```

创建点云后，我们在点云中插入一些随机点。用下面的函数将PCL点云转换为ROS消息：

```
// 将点云数据转换为 ROS 消息
pcl::toROSMsg(cloud, output);
```

转换为ROS消息后，我们就可以在/`pcl_output`话题上简单地发布数据。

10.8.2 如何订阅和处理点云

在本示例中，我们将了解如何在/pcl_output话题上订阅已经创建的点云。订阅点云后，我们用PCL中一个叫VoxelGrid类的滤波器，通过保持输入点云的同质心的方式来降采样。可以从软件包的src文件夹中找到本例的源码pcl_filter.cpp：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
//Vortex filter header
#include <pcl/filters/voxel_grid.h>

// 创建用于处理点云数据的类
class cloudHandler
{
public:
    cloudHandler()
    {
        // 订阅发布的pcl_output话题
        // 这个话题可以根据点云源来修改

        pcl_sub = nh.subscribe("pcl_output", 10, &cloudHandler::cloudCB, this);
        // 创建发布者来过滤点云数据
        pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_filtered",
1);
    }
    // 创建点云的回调函数
    void cloudCB(const sensor_msgs::PointCloud2& input)
    {
        pcl::PointCloud cloud;
```

```
pcl::PointCloud<pcl::PointXYZ> cloud_filtered;

sensor_msgs::PointCloud2 output;
pcl::fromROSMsg(input, cloud);

// 创建 VoxelGrid 对象
pcl::VoxelGrid<pcl::PointXYZ> vox_obj;
// 配置 voxel 对象的输入点云
vox_obj.setInputCloud (cloud.makeShared());
// 配置滤波器的参数，如叶大小
vox_obj.setLeafSize (0.1f, 0.1f, 0.1f);
// 执行滤波器并复制点云滤波变量
vox_obj.filter(cloud_filtered);
pcl::toROSMsg(cloud_filtered, output);
output.header.frame_id = "point_cloud";

pcl_pub.publish(output);
}

protected:
ros::NodeHandle nh;
ros::Subscriber pcl_sub;
ros::Publisher pcl_pub;
};

main(int argc, char** argv)
{
    ros::init(argc, argv, "pcl_filter");
    ROS_INFO("Started Filter Node");
    cloudHandler handler;
    ros::spin();
    return 0;
}
```

这段代码订阅了/`pcl_output`点云话题，并用`VoxelGrid`滤波器进行滤波，然后在/`cloud_filtered`话题上发布了滤波后的点云。

10.8.3 将点云数据写入点云数据（PCD）文件

我们可以用下面这段代码，将点云保存到PCD文件中，在src文件夹中将源码文件命名为pcl_write.cpp：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
// 用于写入 PCD 文件的头文件
#include <pcl/io/pcd_io.h>

void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::fromROSMsg(input, cloud);

    // 将数据保存为 test.pcd 文件
    pcl::io::savePCDFileASCII ("test.pcd", cloud);
}
```

```
main (int argc, char **argv)
{
    ros::init (argc, argv, "pcl_write");
    ROS_INFO("Started PCL write node");

    ros::NodeHandle nh;
    ros::Subscriber bat_sub = nh.subscribe("pcl_output", 10, cloudCB);

    ros::spin();

    return 0;
}
```

10.8.4 从PCD文件中读取并发布点云

下面这段代码读取一个了PCD文件，然后在pcl_output话题上发布了点云。该段代码在src文件夹中的pcl_read.cpp源码文件中：

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/io/pcd_io.h>

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

    ROS_INFO("Started PCL read node");

    ros::NodeHandle nh;
    ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
    ("pcl_output", 1);

    sensor_msgs::PointCloud2 output;
    pcl::PointCloud<pcl::PointXYZ> cloud;

    // 加载 test.pcd 文件
    pcl::io::loadPCDFile ("test.pcd", cloud);

    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "point_cloud";

    ros::Rate loop_rate(1);
    while (ros::ok())
    {
        // 发布在 pcd 文件内部的点云数据
        pcl_pub.publish(output);
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

我们可以创建一个名为pcl_ros_tutorial的ROS软件包来编译这些示例：

```
$ catkin_create_pkg pcl_ros_tutorial pcl pcl_ros roscpp sensor_msgs
```

我们也可以使用已编译好的软件包。

在src文件夹中创建上述示例源码文件pcl_publisher.cpp, pcl_filter.cpp和pcl_write.cpp和pcl_read.cpp。

创建CMakeLists.txt编译所有的源码：

```
## Declare a cpp executable
add_executable(pcl_publisher_node src/pcl_publisher.cpp)
add_executable(pcl_filter src/pcl_filter.cpp)
add_executable(pcl_write src/pcl_write.cpp)
add_executable(pcl_read src/pcl_read.cpp)

target_link_libraries(pcl_publisher_node
    ${catkin_LIBRARIES}
)
target_link_libraries(pcl_filter
    ${catkin_LIBRARIES}
)
target_link_libraries(pcl_write
    ${catkin_LIBRARIES}
)
target_link_libraries(pcl_read
    ${catkin_LIBRARIES}
)
```

用catkin_make编译这个软件包，我们可以用下面的命令运行pcl_publisher_node节点，并在RViz中查看点云：

```
$ rosrun rviz rviz -f point_cloud
```

图10-16是来自pcl_output话题的点云。

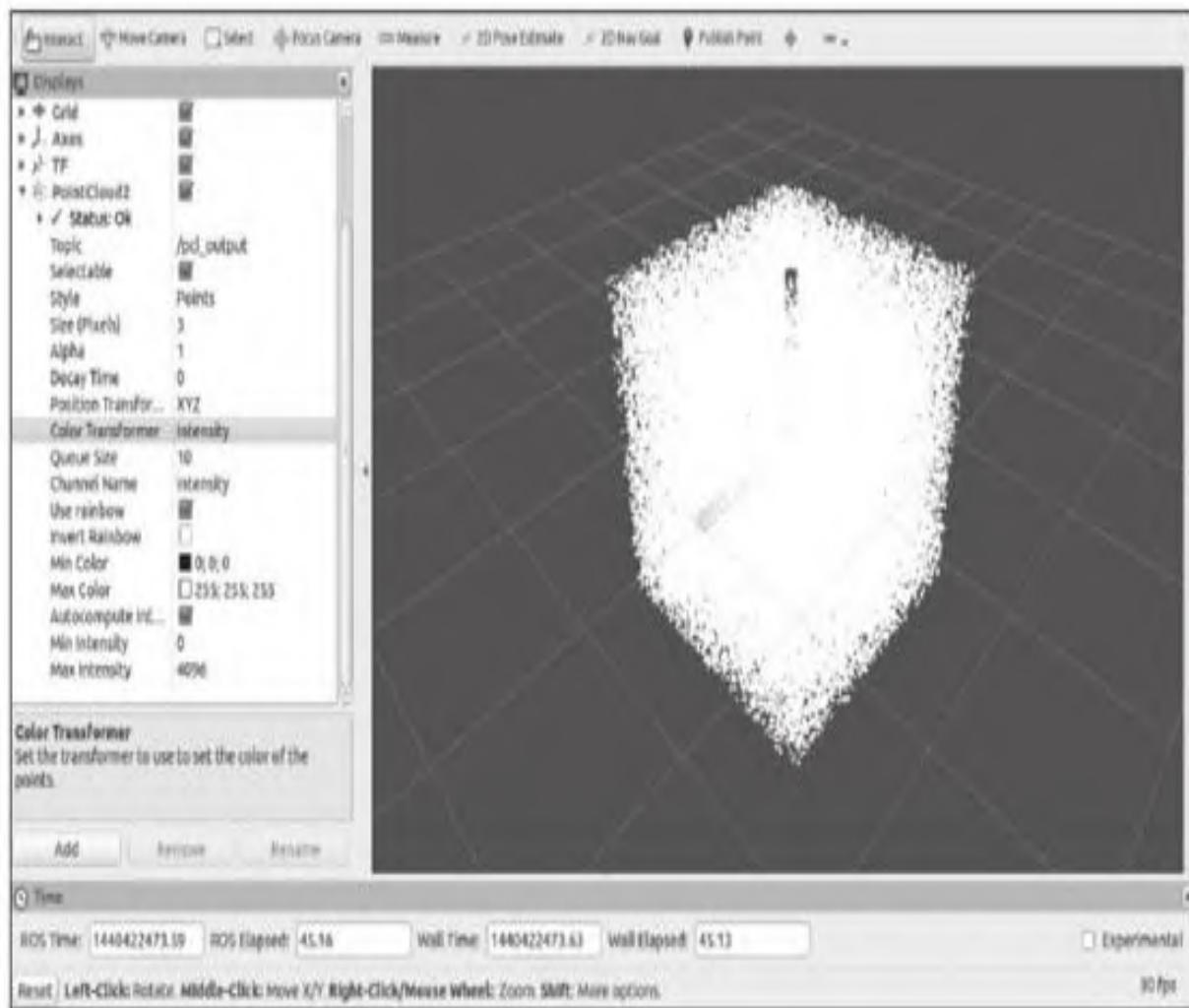


图10-16 RViz中的PCL点云

可以运行pcl_filter节点来订阅pcl_output点云话题，然后进行voxel grid滤波。图10-17是 pcl_filtered话题的输出，即降采样的结果。

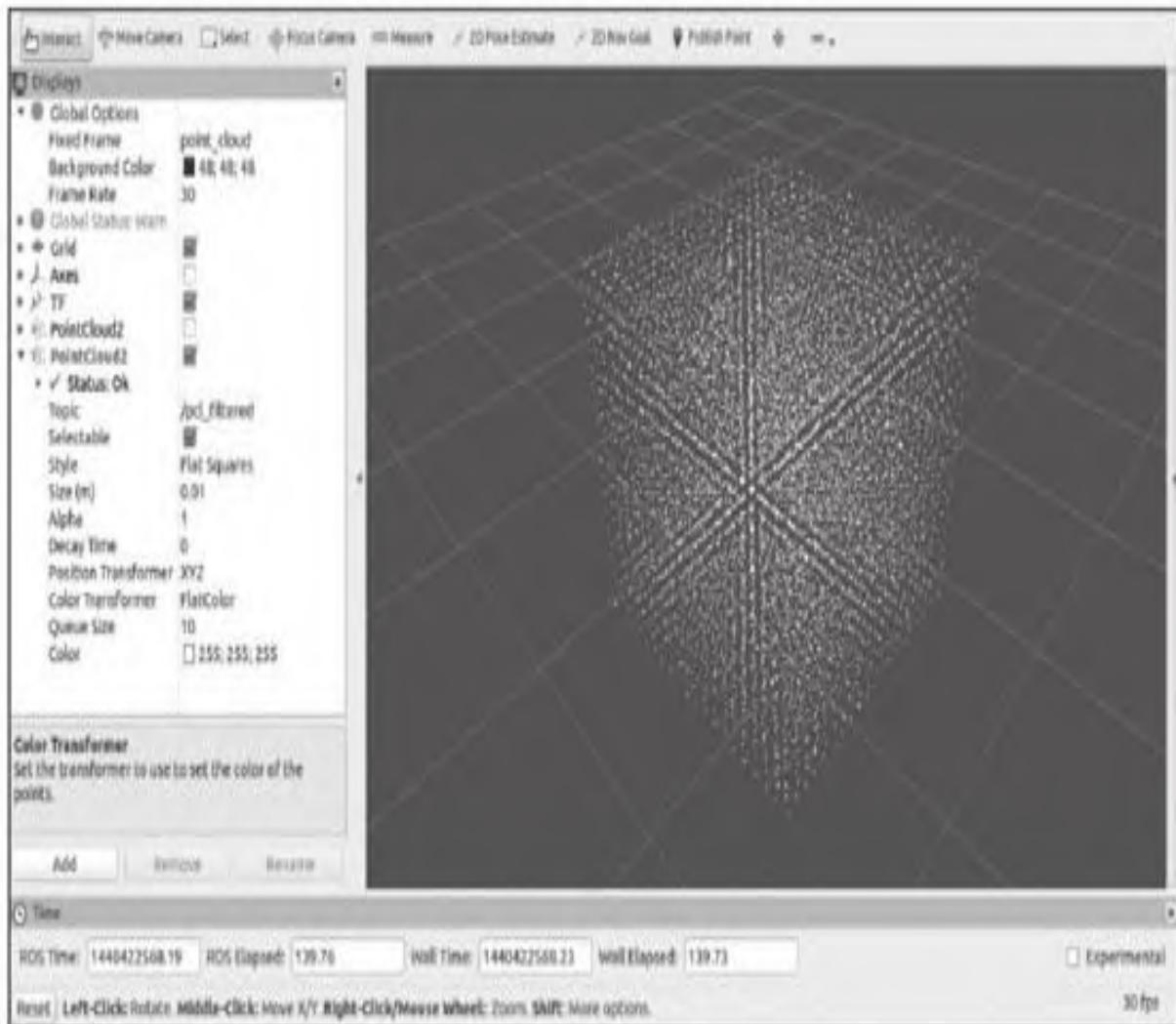


图10-17 RViz中滤波后的点云

可以用pcl_write节点来编写pcl_output点云，用pcl_read节点来读取和发布点云。

10.9 物体姿态估计与AR标记检测

本例中，我们将学习如何利用基准标记，让机器人与其所处的周围环境进行交互。为了与环境中的任意物体进行交互，机器人需要依靠其视觉传感器来识别和定位这些物体。物体姿态估计是所有机器人和计算机视觉应用系统中的一个重要功能。但是，能够在现实环境中实现高效的物体识别和姿态估计是很不容易的，并且在多数情况下，一台相机不足以确定物体的3D姿态。

确切地讲，仅使用一个固定相机不可能得到一个场景的空间深度信息。因此，通常会利用AR（Augmented Reality，增强现实）标记来简化物体姿态估计。AR标记通常由人工合成的方形图像表示，由宽的黑色外边框和内部黑白图案组成，从而可以确定唯一的标识符，如图10-18所示。黑色的外边框有助于图像的快速检测，而二进制表示的黑白编码则可以进行标识符的确认，同时应用错误检测和错误纠正技术。

利用这些标记的主要优势是，每个基准图像都可以被程序检测到，且都是唯一的标识。利用标记字典，机器人能够检测并识别大量的物体，并与之进行交互。另外，还可以通过检测标记的大小来估计给定对象相对于视觉传感器的距离。这些标记通常被称为AR标记，因为这种技术已被广泛应用于增强现实，允许在视频中呈现虚拟信息或虚拟对象。目前，使用AR标记技术已经开发出了不同形式的程序，很多已经提供了ROS的接口，如ARToolKit (https://github.com/artoolkit/ar-tools/ar_tools) 和ArUco (一个微型的增强现实库) (<https://www.uco.es/investiga/grupos/ava/node/26>)。下面，我们将讨论如何使用AprilTag视觉基准系统来检测和定位标记，并将其与ROS连接，从而可以估计物体的三维位置和姿态。



图10-18 增强现实标记

使用Apriltag和ROS

Apriltag (<https://april.eecs.umich.edu/software/apriltag.html>) 是专门为机器人应用设计的基准系统。Apriltag计算得到的AR标记姿态精度非常高，所以非常适合开发机器人应用。可以从下面的库中获取Apriltag的代码：

```
$ git clone https://github.com/RIVeR-Lab/apriltags_ros.git
```

现在你可以编译ROS工作区来构建apriltags软件包，及其在ROS下的移植版本apriltags_ros。使用Apriltag时，需要注意以下事项：

- 视频流：通过订阅sensor_msgs/Image话题来获取视频数据。
- 相机校准：如前面所示，通过订阅sensor_msgs/CameraInfo话题来获取校准数据。
- 标记描述：配置要检测的标记，特别是它的ID、标记的大小，以及与其姿态相关的坐标系，这些都必须设置好。
- 标记：打印出用于检测的标记。Apriltag已经提供了5种不同编码（16h5、25h7、25h9、36h9、36h11）的标记集，并提供了png文件供直接打印。这些标记都可以在apriltags软件包下的apriltags/tags文件夹中找到。

配置完apriltags_ros并启动后，它将发布在坐标系场景中检测到的所有标记的姿态。

图10-19显示的这些话题都是由apriltags节点发布的。

通过查看发布在/tag_detections_image话题上的图像数据，可以以图形化的方式看到详细处理流程和检测到的标记，每个基准标记都被高亮显示，并给出了ID值。

检测到的标记的姿态被发布在/tag_detections话题上，该话题具有apriltags_ros/AprilTag-DetectionArray类型。所有检测到的标记的ID值和姿态都会发布到该话题中。与图10-20所示坐标系相对应的话题的内容如图10-21所示。

```
/tag_detections
/tag_detections_image
/tag_detections_image/compressed
/tag_detections_image/compressed/parameter_descriptions
/tag_detections_image/compressed/parameter_updates
/tag_detections_image/compressedDepth
/tag_detections_image/compressedDepth/parameter_descriptions
/tag_detections_image/compressedDepth/parameter_updates
/tag_detections_image/theora
/tag_detections_image/theora/parameter_descriptions
/tag_detections_image/theora/parameter_updates
/tag_detections_pose
/tf
```

图10-19 由AprilTag发布的ROS话题列表

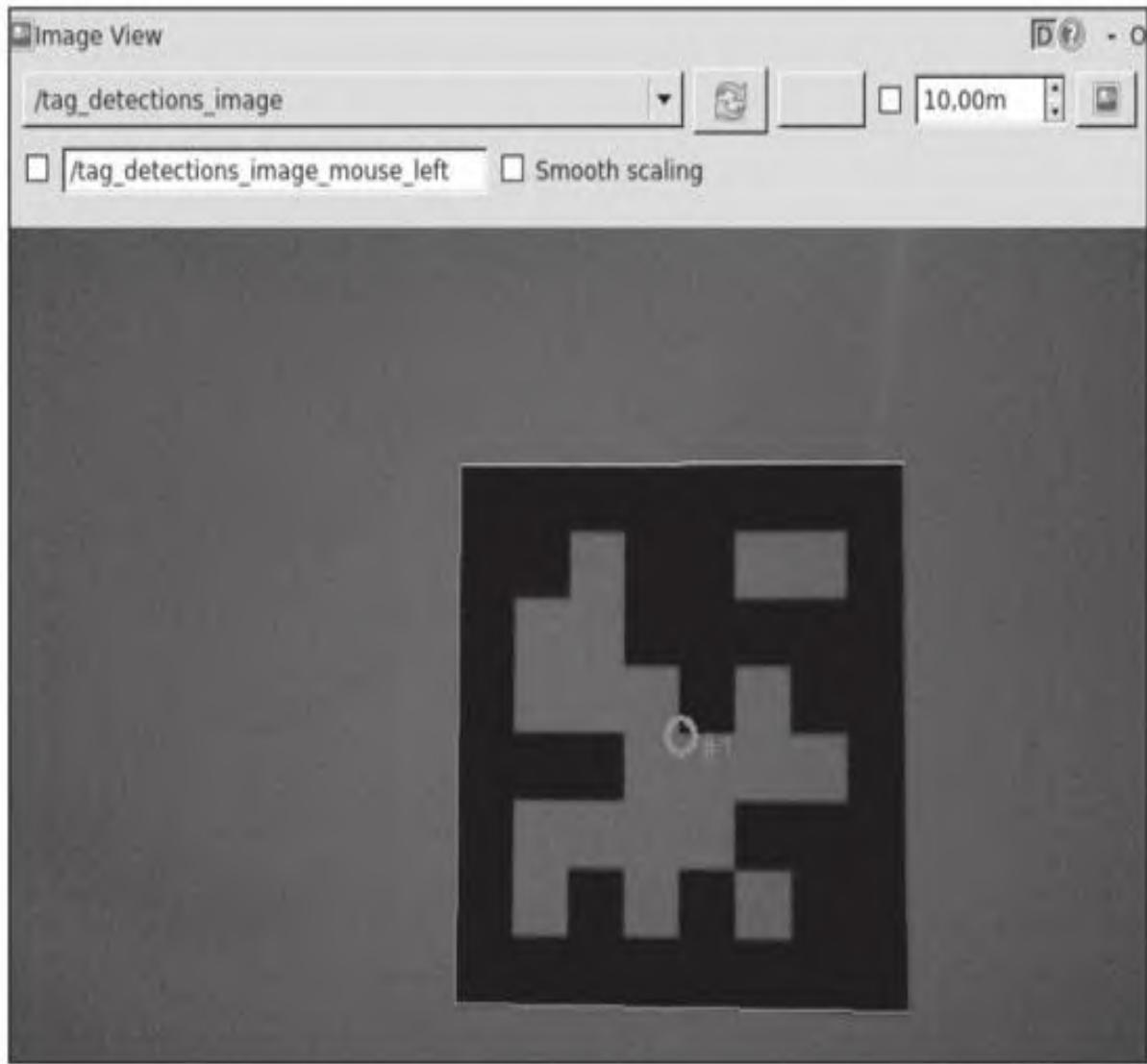


图10-20 ROS下AprilTag的图形化输出结果

```
detections:
  -
    id: 1
    size: 0.08
    pose:
      header:
        seq: 55709
      stamp:
        secs: 1510415864
        nsecs: 148304216
      frame_id: camera_rgb_optical_frame
    pose:
      position:
        x: 0.0201272971812
        y: -0.02393358631
        z: 0.383437954847
      orientation:
        x: 0.713140734773
        y: -0.681737860948
        z: 0.153311144456
        w: 0.0562092015923
  -
```

图10-21 ROS下AprilTag检测到标记的位置和姿态

现在我们来讨论如何用apriltags得到三个物体的位置。校准过视觉系统后，就像前面介绍的那样，我们可以配置并启动apriltags_ros节点。当然，也可以使用随书提供的ROS软件包apriltags_ros_demo来运行这个演示，或者直接从下面的Git库中下载：

```
$ git clone https://github.com/jocacace/apriltags_ros_demo
```

该软件包中有配置好的启动文件，用于启动apriltags。需要特别注意下面两个主要文件：

- bags：包含object.bag包文件，它可以通过ROS话题进行视频数据和相机校准信息的流式发布。

- launch：包含apriltags_ros_objects.launch启动文件，可以播放objects.bag包文件，还能启动apriltags_ros节点，用于识别三种不同的标记。

要配置apriltags_ros节点，我们必须按照下面的方式来修改启动文件：

```
<node pkg="apriltags_ros" type="apriltag_detector_node"
name="apriltag_detector" output="screen">

<remap from="image_rect" to="/camera/rgb/image_raw" />
<remap from="camera_info" to="/camera/rgb/camera_info" />

<param name="image_transport" type="str" value="compressed" />

<param name="tag_family" type="str" value="36h11" />

<rosparam param="tag_descriptions">[
  {id: 6, size: 0.035, frame_id: mastering_ros},
  {id: 1, size: 0.035, frame_id: laptop},
  {id: 7, size: 0.035, frame_id: smartphone}
]
</rosparam>
</node>
```

在该启动文件中，我们可以用remap命令设置相机用到的话题，如视频流和图像话题：

```
<remap from="image_rect" to="/camera/rgb/image_raw" />
<remap from="camera_info" to="/camera/rgb/camera_info" />
```

另外，我们需要告知apriltags_ros节点，哪些标记必须检测。首先，我们需要指定编码类型：

```
<param name="tag_family" type="str" value="36h11" />
```

最后，按照下面的方式指定标记的ID、标记的大小（米），以及每个标记所对应的坐标系名：

```
<rosparam param="tag_descriptions">[  
    {id: 6, size: 0.035, frame_id: mastering_ros},  
    {id: 1, size: 0.035, frame_id: laptop},  
    {id: 7, size: 0.035, frame_id: smartphone}  
]</rosparam>
```

注意，标记的大小是由其黑色外边框的边长表示的。该参数可以让AR标记检测算法，估计标记与相机之间的距离。因此，当需要用这个方法获得精确的姿态估计时，需要特别留意该参数。

本例中，我们要检测三个不同的标记，ID分别是6、1、7，大小都是3.5厘米。每个标记都与一个坐标系关联，可以在RViz中使用tf来查看。可以在apriltags_ros_demo/launch文件夹中找到本例完整的启动文件。

可以直接用提供的启动文件运行本示例。第一次运行时会生成一个bagfile日志文件，该日志文件包含一个场景视频流，场景中有三个物体、三个附加的标记，以及相机校准信息。

```
$ rosrun apriltags_ros_demo apriltags_ros_objects.launch
```

现在，你可以通过apriltags_ros话题来输出这些物体的姿态信息，或者在RViz中进行可视化，如图10-22所示。

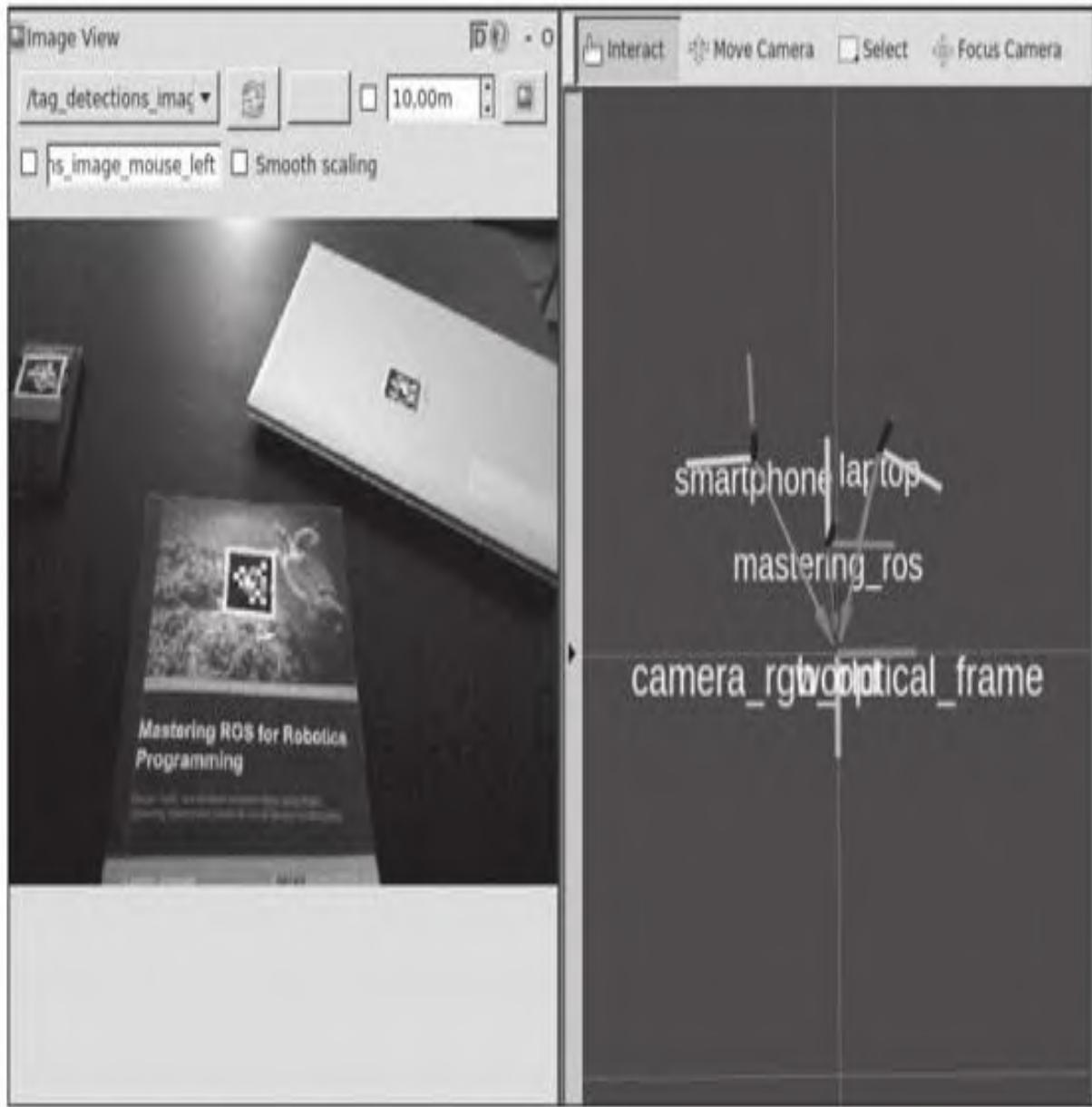


图10-22 使用AR标记和ROS跟踪多个对象

10.10 习题

- 在vision_opencv软件包集中有哪些软件包?
- 在perception_pcl软件包集中有哪些软件包?
- cv_bridge软件包的功能是什么?
- 如何将PCL点云转换为ROS消息?
- 如何用ROS进行分布式计算?
- AR标记的主要优点是什么?

10.11 本章小结

本章介绍了视觉传感器及其如何在ROS中编程。我们了解了用于连接相机和3D视觉传感器的接口软件包集：`vision_opencv`和`perception_pcl`。我们学习了这些软件包集中的每个软件包及其功能。我们使用ROS的`cv_bridge`软件包了解了相机接口及图像处理的接口。讨论了`cv_bridge`后，我们研究了各种3D视觉传感器和激光雷达与ROS的接口。之后，我们又学习了如何用PCL库和ROS处理来自这些传感器的数据。在本章的最后部分，我们演示了如何使用基准标记来轻松地估计物体的姿态信息。下一章，我们将学习ROS中机器人的硬件接口。

第11章

在ROS中构造与连接差速驱动移动机器人

在前一章，我们讨论了ROS的机器人视觉。本章，我们将学习如何制作差速驱动的自主移动机器人，以及如何将其与ROS连接。我们将了解如何为该机器人配置ROS导航软件包集（navigation stack），利用SLAM和AMCL使其自主运动。本章旨在让你了解如何制作定制的移动机器人及其与ROS的接口。

本章将介绍以下内容：

- 介绍DIY自主移动机器人Chefbot
- 用ROS为Chefbot开发底盘控制器和里程计节点
- 为Chefbot配置导航软件包集并了解AMCL
- Chefbot的仿真
- 导航软件包集与ROS节点交互

本章讨论的第一个主题是如何制作一个DIY自主移动机器人，开发其固件代码，并利用ROS导航软件包集进行控制。这个叫Chefbot的机器人（见图11-1）是本书作者（Lentin Josep）编写的另一本名为《Learning Robotics using Python》的书中的一部分，由Packt出版社出版（<http://learn-robotics.com>）。这本书讨论了如何一步一步地制作Chefbot机器人。在本章，我们将涉及该机器人硬件的简要介绍，学习如何配置ROS导航软件包集，如何用SLAM和AMCL进行自主导航，并优化配置参数。我们已经在第6章讨论过ROS的导航软件包集，用Gazebo对差速驱动机器人进行了仿真，并运行了SLAM和AMCL。本章的第一部分，必须有一个Chefbot机器人，才能学习这部分的教程。本章第一部分讨论的概念将用于第二部分的仿真机器人。



图11-1 Chefbot机器人原型

11.1 Chefbot DIY移动机器人及其硬件配置

我们在第6章讨论了移动机器人与ROS导航软件包接口的一些必备条件。下面回顾一下这些条件：

- 测距源：机器人需要发布自身相对于起始位置的距离/位置信息。提供距离信息的必备硬件是车轮编码器、IMU或2D/3D相机（视觉测距）。
- 传感器源：需要激光雷达或可以作为激光雷达的视觉传感器。激光扫描数据在使用SLAM进行地图构建的过程中至关重要。
- 用tf进行传感器坐标系变换：机器人需要利用ROS的坐标变换功能(tf)发布传感器和其他机器人组件的坐标变换。
- 底盘控制器：底盘控制器是一个ROS节点，它可以将来自导航软件包集的twist消息转换为相应的马达速度。

我们需要检查机器人中现有的零配件，并确定是否满足导航软件包集的要求。机器人中现有以下配件：

- 带正交编码器的Pololu直流减速马达(<https://www.pololu.com/product/1447>)：该马达的参数分别是12 V, 80 RPM, 18 kg-cm扭矩。空载运行时的电流为300 mA, 堵转电流是5 A。该马达连接到正交编码器后，经过变速器每转可输出8400个计数。马达编码器计数是机器人的一个测距源。

- Pololu马达驱动器 (<https://www.pololu.com/product/708>) : 这是可以连接Pololu马达的双路马达驱动器，可支持高达30 A的电流和5.5 V~16 V的马达电压。
- Tiva C Launchpad控制器 (<http://www.ti.com/tool/ek-tm4c123gxl>) : 该机器人配有Tiva C LaunchPad控制器，用于连接马达、编码器、传感器。此外，也可以从PC接收控制命令，并根据命令向马达发送适当的控制信号。该控制器可以充当机器人的嵌入式控制板。Tiva C Launchpad板的运行频率是80 MHz。
- MPU 6050 IMU: 该机器人使用的IMU是MPU 6050，它含有加速度计、陀螺仪和数字运动处理器 (Digital Motion Processor, DMP)。该运动处理器可以在板上运行传感器融合算法，并且提供横滚角、俯仰角和偏航角的精确结果。可以用IMU的输出配合轮子编码器计算里程。
- Xbox Kinect/华硕Xtion Pro: 这些是3D视觉传感器，可用于模拟激光雷达。从这些传感器生成的点云可以转换为激光扫描数据并用于导航。
- 英特尔NUC PC: 这是英特尔的迷你PC，我们需要为其安装好Ubuntu和ROS。PC连接上Kinect和LaunchPad就能获取到传感器值和里程信息。PC上运行的程序可以计算机器人的tf，并运行导航软件包集和相关的软件包，如SLAM和AMCL。该PC安装在机器人内部。

根据上述硬件列表，检查ROS导航软件包集所需的所有要求是否全部满足。图11-2显示了该机器人的关系框图。

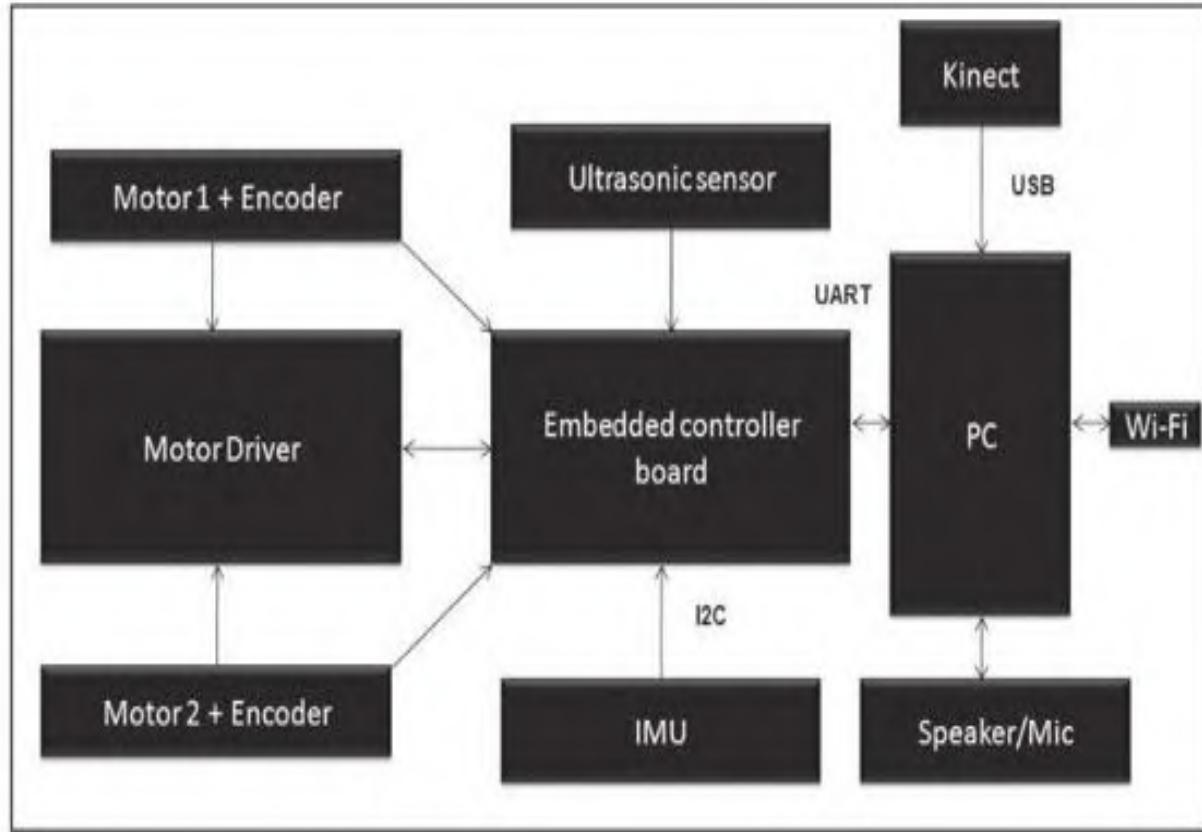


图11-2 Chefbot机器人的关系框图

这个机器人采用的嵌入式控制板是Tiva C LaunchPad。所有的传感器和驱动器都连接到该控制板上，再连接到英特尔NUC PC主机上以接收更高级别的指令。该控制板与PC通过UART协议进行通信，与IMU进行I2C通信。Kinect用USB与PC连接，所有其他传感器都用GPIO引脚来连接。机器人各组件的详细连接图如图11-3所示。

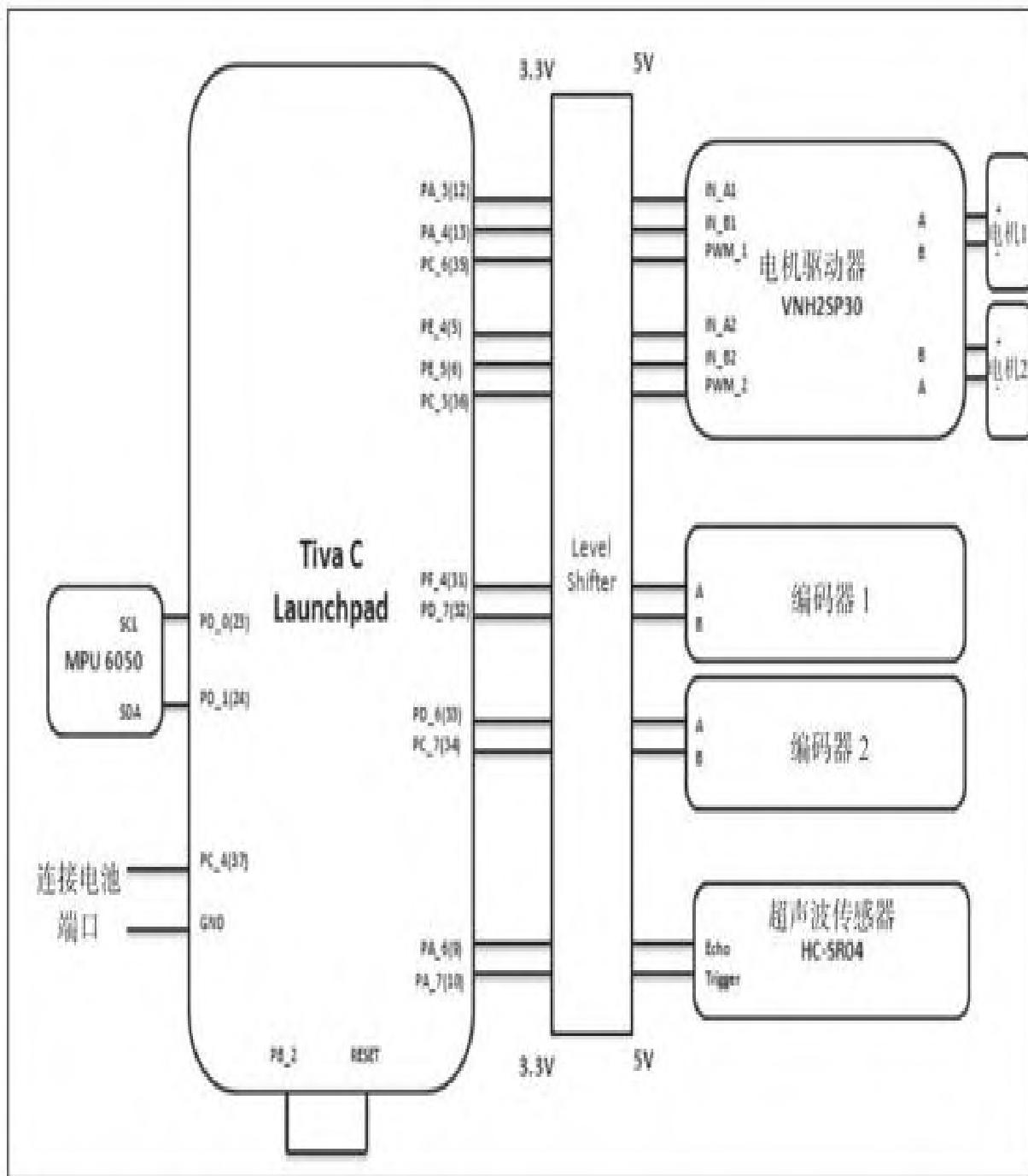


图11-3 Chefbot机器人的连接图

11.1.1 使用Energia IDE来烧写Chefbot固件

硬件连接好后，就可以用Energia IDE (<http://energia.nu/>) 在Launchpad上编程了。在PC上配置好Energia IDE（首选操作系统为Ubuntu）后，我们就可以将机器人固件烧写到控制板上了。使用下面的命令来获取固件代码和ROS接口软件包：

```
$ git clone https://github.com/qboticslabs/Chefbot_ROS_pkg
```

源码文件夹下包含一个名为tiva_c_energia_code的文件夹，该文件夹中有可烧写到控制板上的固件代码，代码可在Energia IDE中编译。该固件代码不仅可以读取编码器、超声波传感器和IMU值，还可以接收控制马达转速指令。在这里将讨论固件代码的重要部分。

LaunchPad的编程语言和Arduino是相同的。这里我们用Energia IDE在控制板上编程，实际由Arduino IDE来编译。下面这段代码是setup()函数的定义，其功能是以115 200的波特率启动串口通信，并配置马达编码器引脚、马达驱动器、超声波测距传感器、IMU的引脚。此外，通过此代码我们还配置了一个额外引脚，用于重置LaunchPad：

```
void setup()
{
    // 初始化串口波特率为 115 200bps
    Serial.begin(115200);
    // 配置编码器
    SetupEncoders();
    // 配置电机
    SetupMotors();
    // 配置超声波传感器
    SetupUltrasonic();
    // 配置 MPU 6050 传感器
    Setup_MP6050();
    // 配置重置引脚
    SetupReset();
    // 配置消息
    Messenger_Handler.attach(OnMessageCompleted);
}
```

在loop()函数中，不断地轮询传感器值并通过串口发送数据，同时不断轮询传入串口的数据以获取机器人的控制指令。下面的常用协议用于通过串口（UART）将每个传感器数据从LaunchPad发送到PC主机。

LaunchPad到PC的串口数据发送协议

编码器的协议如下：

e<space><left_encoder_ticks><space><right_encoder_ticks>

超声波传感器的协议如下：

u<space><distance_in_centimeter>

IMU的协议如下：

i<space><value_of_x_quaternion><space><value_of_y_quaternion>
<space><value_of_z_quaternion><space><value_of_w_quaternion>

PC到Launchpad的串口数据发送协议

马达的协议如下：

s<space><pwm_value_of_motor_1><space><pwm_value_of_motor_2>

重置设备的协议如下：

r<space>

我们可以用名为miniterm.py的命令行工具检查LaunchPad的串口数据。该工具可以查看来自设备的串口数据。安装python-serial软件包时就已经安装了这个脚本，该软件包是和rosserial-python Debian 软件包一起安装的。以下命令将显示机器人控制器的串口值：

```
$ miniterm.py /dev/ttyACM0 115200
```

我们将得到如图11-4所示截屏中显示的数值。

```
b      0.00
t      66458239      0.05
e      0      0
u      10
s      0.00      0.00
i      -0.68     -0.47     -0.40     0.40
b      0.00
t      66511681      0.05
e      0      0
u      10
s      0.00      0.00
i      -0.68     -0.47     -0.40     0.40
b      0.00
t      66566051      0.05
e      0      0
u      10
s      0.00      0.00
i      -0.68     -0.47     -0.40     0.40
b      0.00
t      66620423      0.05
e      0      0
u      10
s      0.00      0.00
```

图11-4 使用miniterm.py检查串口数据

11.1.2 讨论Chefbot的ROS软件包接口

确认过开发板的串口数据后，我们就可以安装Chefbot的ROS软件包了。Chefbot软件包有以下文件和文件夹：

- chefbot_bringup：该软件包包含Python脚本、C++节点和启动文件，主要用于发布机器人的里程计信息和tf，执行gmapping和AMCL。它包含用于从LaunchPad读写数据的Python/C++节点。它可以将编码器计数值转换为tf，并将twist消息转化为马达指令。它还包含PID节点，用于处理来自马达的速度控制指令。
- chefbot_description：该软件包包含Chefbot的URDF模型文件。
- chefbot_simulator：该软件包包含可以在Gazebo中仿真机器人的启动文件。
- chefbot_navig_cpp：该软件包包含这些节点的C++实现，这些节点在chefbot_bringup中已经作为Python节点实现过了。

从前面的GitHub库中下载Chefbot源码后，就可以编译工作区了。若要成功编译库中的软件包，必须使用下面的命令来安装一些依赖项：

```
$ sudo apt-get install ros-kinetic-depthimage-to-laserscan ros-kinetic-kobuki-gazebo-plugins ros-kinetic-robot-pose-ekf ros-kinetic-yocs-cmd-vel-mux ros-kinetic-move-base-msgs ros-kinetic-openni-launch ros-kinetic-kobuki-description ros-kinetic-gmapping ros-kinetic-amcl ros-kinetic-map-server
```

下面的启动文件将启动机器人的里程计和tf发布节点：

```
$ roslaunch chefbot_bringup robot_standalone.launch
```

图11-5显示了由该启动文件启动的各节点，以及它们之间如何互相连接。

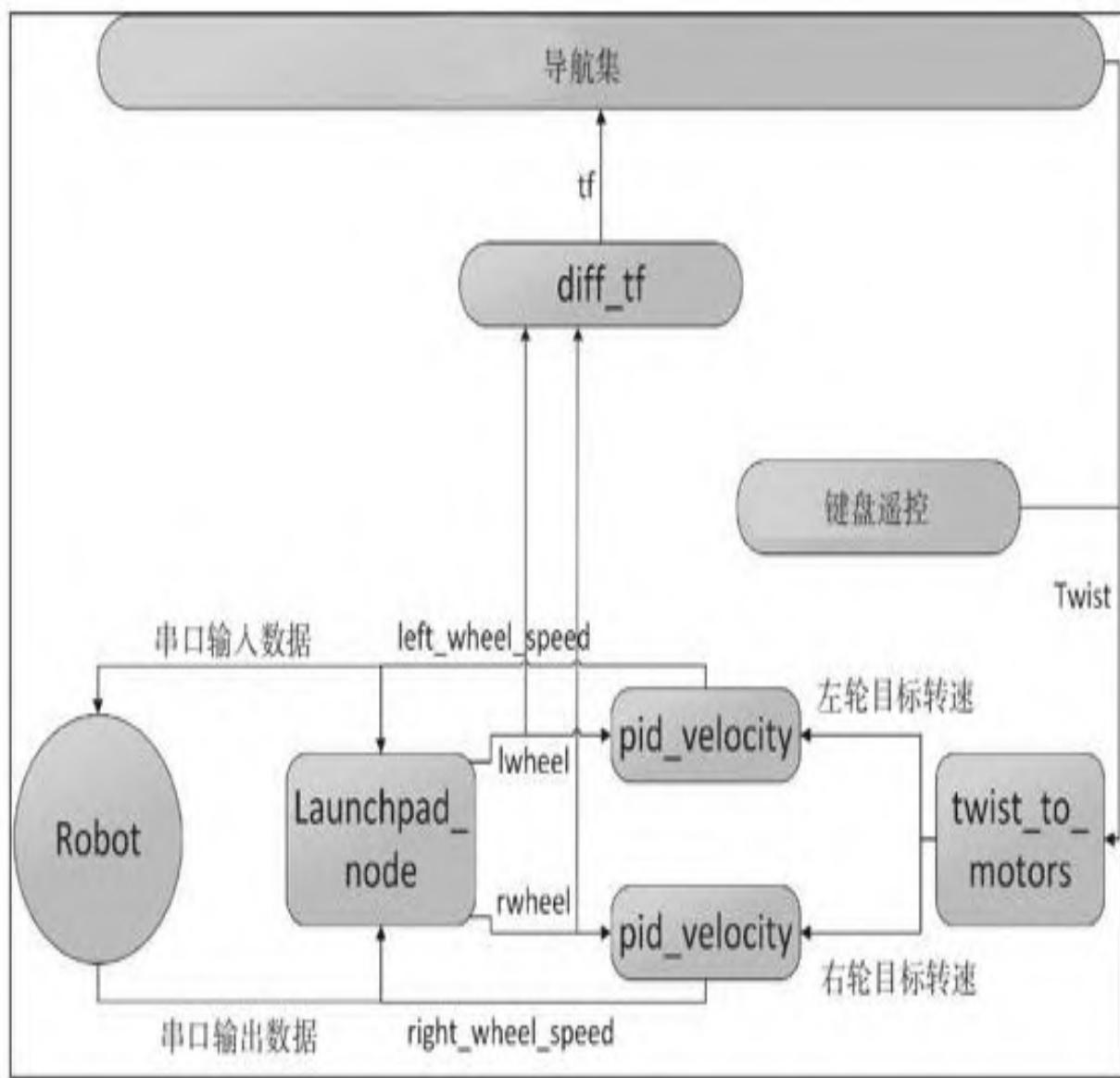


图11-5 Chefbot的各节点关系图

接下来解释由此启动文件启动的各节点及其运行方式：

- `launchpad_node.py`: 我们知道Chefbot机器人将Tiva C LaunchPad作为其控制器。该节点充当机器人控制器和ROS之间的桥梁。该节点的基本

功能是从LaunchPad接收串口数据，并将每个传感器的数据转换为ROS话题。该节点还充当LaunchPad的ROS驱动程序。

- `twist_to_motors.py`: 该节点可以将`geometry_msgs/Twist`消息转化为马达的目标速度。它订阅了来自遥控节点或来自ROS导航软件包集的速度指令，并发布`lwheel_vtarget`和`rwheel_vtarget`。
- `pid_velocity.py`: 该节点订阅了`twist_to_motors`节点的`wheel_vtarget`话题和`launchpad_node`节点的编码器计数话题。我们需要为机器人的每个轮子启动一个PID节点，如图11-5所示。该节点最终将为每个马达生成相应的马达控制指令。
- `diff_tf.py`: 该节点订阅两个马达的编码器计数，然后计算里程计信息，最后为导航软件包集发布`tf`信息。

运行`robot_standalone.launch`后生成的话题列表如图11-6所示。

```
lentin@lentin-Aspire-4755:~$ rostopic list
/battery_level
/cmd_vel_mux/input/teleop
 imu/data
/joint_states
/left_wheel_speed
/lwheel
/lwheel_vel
/lwheel_vtarget
/odom
/qw
/qx
/qy
/qz
/right_wheel_speed
/rosout
/rosout_agg
/rwheel
/rwheel_vel
/rwheel_vtarget
/serial
/tf
/ultrasonic_distance
```

图11-6 执行robot_standalone.launch时生成的话题列表

下面就是robot_standalone.launch文件的内容：

```
<launch>
  <arg name="simulation" default="$(optenv TURTLEBOT_SIMULATION false)"/>
  <param name="/use_sim_time" value="$(arg simulation)"/>

  <!-- URDF robot model -->
  <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find
chefbot_description)/urdf/chefbot_base.xacro'" />
  <param name="robot_description" command="$(arg urdf_file)" />

  <!-- important generally, but specifically utilised by the current app
manager -->
  <param name="robot/name" value="$(optenv ROBOT turtlebot)"/>
  <param name="robot/type" value="turtlebot"/>
```

```

<!-- Starting robot state publisher -->
<node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher">
    <param name="publish_frequency" type="double" value="5.0" />
</node>

<!-- Robot parameters -->
<rosparam param="base_width">0.3</rosparam>
<rosparam param="ticks_meter">14865</rosparam>

<!-- Starting launchpad_node -->
<node name="launchpad_node" pkg="chefbot_bringup"
type="launchpad_node.py">
    <rosparam file="$(find chefbot_bringup)/param/serial.yaml"
command="load" />
</node>

<!-- PID node for left motor , setting PID parameters -->
<node name="lpid_velocity" pkg="chefbot_bringup" type="pid_velocity.py"
output="screen">
    <remap from="wheel" to="lwheel"/>
    <remap from="motor_cmd" to="left_wheel_speed"/>
    <remap from="wheel_vtarget" to="lwheel_vtarget"/>
    <remap from="wheel_vel" to="lwheel_vel"/>
    <rosparam param="Kp">400</rosparam>
    <rosparam param="Ki">100</rosparam>
    <rosparam param="Kd">0</rosparam>
    <rosparam param="out_min">-1023</rosparam>
    <rosparam param="out_max">1023</rosparam>
    <rosparam param="rate">30</rosparam>
    <rosparam param="timeout_ticks">4</rosparam>
    <rosparam param="rolling_pts">5</rosparam>
</node>

<!-- PID node for right motor, setting PID parameters -->
<node name="rpid_velocity" pkg="chefbot_bringup" type="pid_velocity.py"
output="screen">
    <remap from="wheel" to="rwheel"/>
    <remap from="motor_cmd" to="right_wheel_speed"/>
    <remap from="wheel_vtarget" to="rwheel_vtarget"/>
    <remap from="wheel_vel" to="rwheel_vel"/>
    <rosparam param="Kp">400</rosparam>
    <rosparam param="Ki">100</rosparam>
    <rosparam param="Kd">0</rosparam>
    <rosparam param="out_min">-1023</rosparam>
    <rosparam param="out_max">1023</rosparam>
    <rosparam param="rate">30</rosparam>
    <rosparam param="timeout_ticks">4</rosparam>
    <rosparam param="rolling_pts">5</rosparam>
</node>

<!-- Starting twist to motor and diff_tf nodes -->
<node pkg="chefbot_bringup" type="twist_to_motors.py"
name="twist_to_motors" output="screen"/>
<node pkg="chefbot_bringup" type="diff_tf.py" name="diff_tf">

```

```
    output="screen"/>  
  
  </launch>
```

运行robot_standalone.launch后，可以用下面的命令在RViz中显示机器人：

```
$ rosrun chefbot Bringup view_robot.launch
```

我们将看到如图11-7所示的机器人模型。

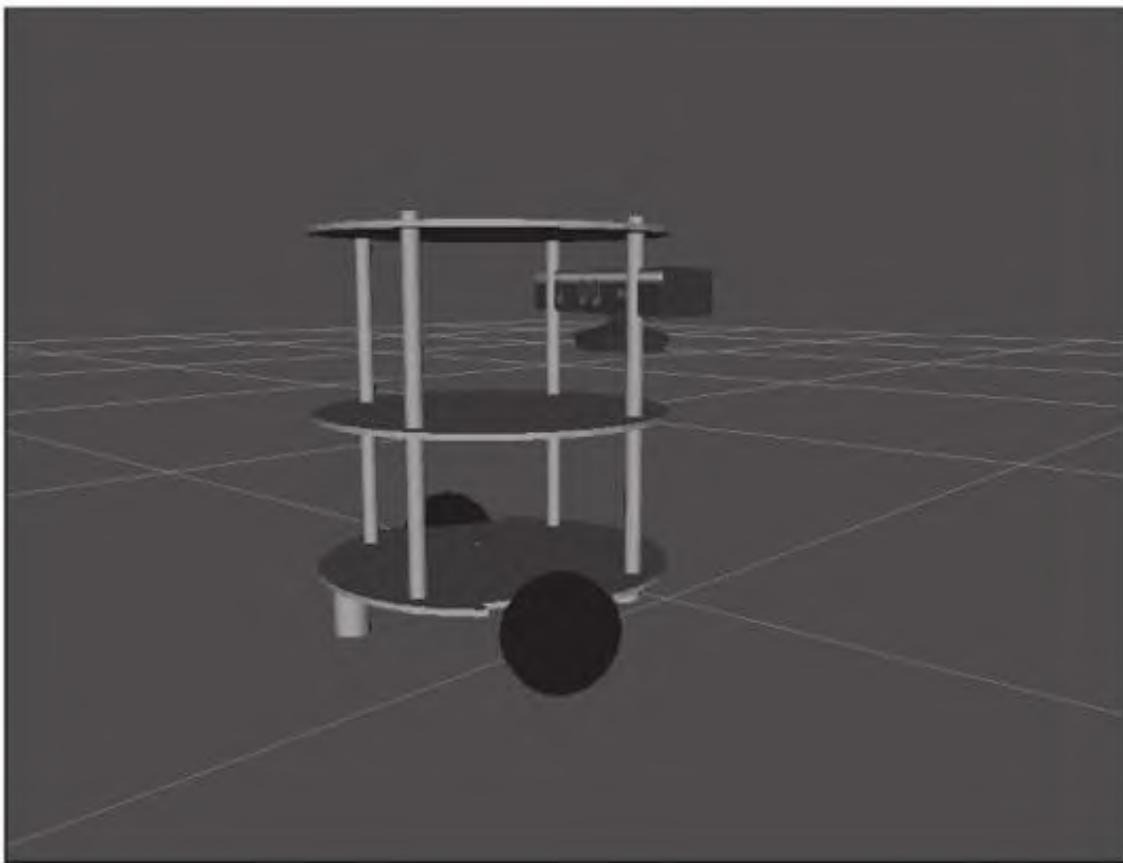


图11-7 用真实机器人数据显示的机器人模型

启动teleop键盘遥控节点后，我们就可以控制机器人运动：

```
$ roslaunch chefbot_bringup keyboard_teleop.launch
```

利用键盘控制机器人运动，我们会看到机器人可以四处移动。如果在RViz中启用机器人的tf，我们就可以看到里程计信息，如图11-8所示。

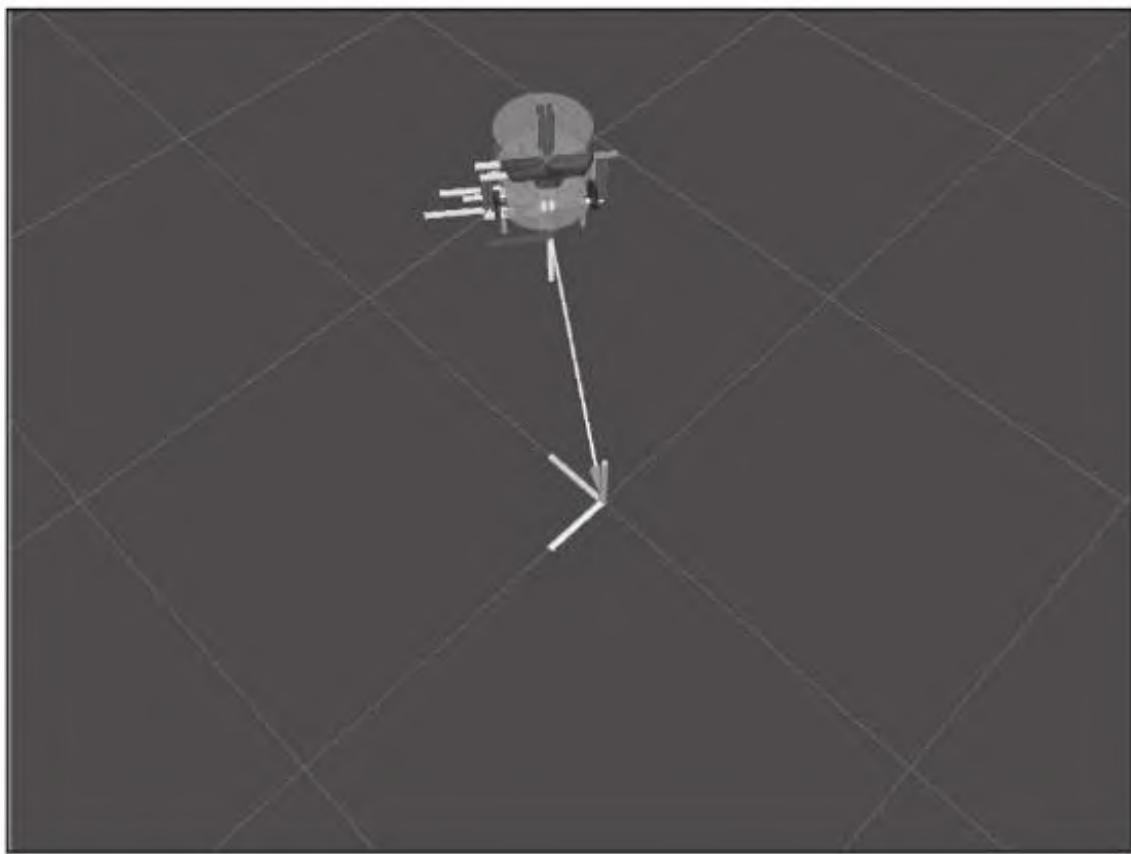


图11-8 查看机器人里程计

我们可以使用rqt_graph工具来查看各节点间的连接图：

```
$ rqt_graph
```

至此，我们已经讨论了Chefbot机器人的ROS接口。Chefbot机器人的代码完全由Python实现。但是仍然有一些节点是用C++实现的，这些

节点用来从编码器计数计算里程计信息和从*twist*消息生成马达控制指令。

11.1.3 从编码器计数计算里程计信息

本节将介绍diff_tf.py节点的C++实现，该节点订阅了编码器计数，并计算里程计数据，然后发布机器人里程计数据和tf数据。我们来看看这段代码的C++实现，该源码文件diff_tf.cpp位于chefbot_navig_cpp软件包中的src文件夹内。

接下来讨论这段代码的重要片段，并加以解释。下面这段代码是Odometry_calc类的构造函数。该类包含了计算里程计信息的定义。这段代码定义了左右轮编码器的订阅者及odom值的发布者：

```
Odometry_calc::Odometry_calc(){

    // 初始化节点中使用的变量
    init_variables();

    ROS_INFO("Started odometry computing node");

    // 订阅左右轮的编码器值
    l_wheel_sub = n.subscribe("/lwheel",10, &Odometry_calc::leftencoderCb,
    this);
    r_wheel_sub = n.subscribe("/rwheel",10, &Odometry_calc::rightencoderCb,
    this);

    // 创建里程计信息的发布者
    odom_pub = n.advertise<nav_msgs::Odometry>("odom", 50);

    // 获取该节点的参数
    get_node_params();
}
```

下面这段代码是用来计算里程计信息的。它利用机器人编码器、机器人底盘宽度，以及编码器每米的计数值来计算机器人的移动距离

和转动角度。计算得到移动距离和转动角度后，我们就可以利用标准差速驱动机器人的运动模型计算最终的x、y、theta值：

```
if ( now > t_next) {  
  
    elapsed = now.toSec() - then.toSec();  
  
    if(enc_left == 0){  
        d_left = 0;  
        d_right = 0;
```

```

    }
    else{
        d_left = (left - enc_left) / ( ticks_meter);
        d_right = (right - enc_right) / ( ticks_meter);
    }
    enc_left = left;
    enc_right = right;

    d = (d_left + d_right ) / 2.0;

    th = ( d_right - d_left ) / base_width;
    dx = d /elapsed;

    dr = th / elapsed;

    if ( d != 0){

        x = cos( th ) * d;
        y = -sin( th ) * d;

        // 计算机器人的最后位置
        x_final = x_final + ( cos( theta_final ) * x - sin(
theta_final ) * y );
        y_final = y_final + ( sin( theta_final ) * x + cos(
theta_final ) * y );

    }
    if( th != 0)
        theta_final = theta_final + th;
}

```

用上面的代码得到机器人的位置和方向后，我们将odom值赋给odom消息头和tf消息头，它们将在/odom和/tf话题上发布：

```
geometry_msgs::Quaternion odom_quat ;  
  
odom_quat.x = 0.0;  
odom_quat.y = 0.0;  
odom_quat.z = 0.0;  
  
odom_quat.z = sin( theta_final / 2 );  
odom_quat.w = cos( theta_final / 2 );  
  
// 首先，我们使用 tf 发布转换  
geometry_msgs::TransformStamped odom_trans;  
odom_trans.header.stamp = now;  
odom_trans.header.frame_id = "odom";  
odom_trans.child_frame_id = "base_footprint";  
  
odom_trans.transform.translation.x = x_final;  
odom_trans.transform.translation.y = y_final;  
odom_trans.transform.translation.z = 0.0;  
odom_trans.transform.rotation = odom_quat;  
  
// 发送变换  
odom_broadcaster.sendTransform(odom_trans);
```

```
// 接下来，我们将通过 ROS 发布里程计信息
nav_msgs::Odometry odom;
odom.header.stamp = now;
odom.header.frame_id = "odom";

// 设置位置信息
odom.pose.pose.position.x = x_final;
odom.pose.pose.position.y = y_final;
odom.pose.pose.position.z = 0.0;
odom.pose.pose.orientation = odom_quat;

// 设置速度信息
odom.child_frame_id = "base_footprint";
odom.twist.twist.linear.x = dx;
odom.twist.twist.linear.y = 0;
odom.twist.twist.angular.z = dr;

// 发布消息
odom_pub.publish(odom);
```

11.1.4 根据ROS twist消息计算马达转速

本节将讨论twist_to_motor.py的C++实现。该节点将twist消息(geometric_msgs/Twist)转换为马达的目标转速。该节点订阅的话题来自teleop节点或导航软件包集，然后它会向两个马达发布目标转速。该目标转速被馈送到PID节点，然后PID节点向每个马达发送适合的指令。CPP文件名为twist_to_motor.cpp，位于chefbot_navig_cpp/src文件夹内：

```
TwistToMotors::TwistToMotors()
{
    init_variables();
    get_parameters();

    ROS_INFO("Started Twist to Motor node");
    cmd_vel_sub = n.subscribe("cmd_vel_mux/input/teleop", 10,
&TwistToMotors::twistCallback, this);
    pub_lmotor = n.advertise<std_msgs::Float32>("lwheel_vtarget", 50);

    pub_rmotor = n.advertise<std_msgs::Float32>("rwheel_vtarget", 50);
}
```

下面这段代码是twist消息的回调函数。将线速度X赋值给dx，Y赋值给dy并将角速度Z赋值给dr：

```
void TwistToMotors::twistCallback(const geometry_msgs::Twist &msg)
{
    ticks_since_target = 0;
    dx = msg.linear.x;
    dy = msg.linear.y;
    dr = msg.angular.z;

}
```

得到 dx 、 dy 和 dr 后，我们就可以利用下面的方程组计算马达的转速：

$$\begin{aligned} dx &= (l + r) / 2 \\ dr &= (r - l) / w \end{aligned}$$

在这里， l 和 r 是左右车轮的转速， w 是底盘的宽度。下面这段代码实现了上述方程组。计算了车轮的转速后，它们将分别被发布到`lwheel_vtarget`和`rwheel_vtarget`话题上：

```
right = ( 1.0 * dx ) + (dr * w /2);
left = ( 1.0 * dx ) - (dr * w /2);
std_msgs::Float32 left_;
std_msgs::Float32 right_;

left_.data = left;
right_.data = right;

pub_lmotor.publish(left_);
pub_rmotor.publish(right_);

ticks_since_target += 1;

ros::spinOnce();
```

运行机器人的launch文件以启动C++节点

下面的命令可以启动robot_standalone.launch文件，它使用的是C++节点：

```
$ rosrun chefbot_navig_cpp robot_standalone.launch
```

11.1.5 为Chefbot机器人配置导航软件包集

设置好里程计节点、底盘控制器节点和PID节点后，我们需要配置导航软件包集，这样就可以执行SLAM和自适应蒙特卡罗定位（Adaptive Monte Carlo Localization, AMCL）来构建地图，确定机器人的位置，并进行自主导航。在第6章我们已经了解了导航中的基本软件包。要想构建当前环境的地图，我们需要重点配置两个节点：执行SLAM的gmapping节点和move_base节点。我们还需要配置导航功能中的全局规划器、局部规划器、全局代价地图、局部代价地图。首先让我们来看gmapping节点的配置。

11.1.6 配置gmapping节点

gmapping节点是执行SLAM的软件包（见<http://wiki.ros.org/gmapping>）。

该软件包中的gmapping节点主要订阅和发布下述话题：

以下是订阅的话题：

- tf (tf/tfMessage) : 与Kinect、机器人底盘和里程计有关的机器人的坐标变换。
- scan (sensor_msgs/LaserScan) : 创建地图所需的激光扫描数据。

以下是发布的话题：

- map (nav_msgs/OccupancyGrid) : 发布占用栅格地图数据。
- map_metadata (nav_msgs/MapMetaData) : 有关占用栅格地图的基本信息。

gmapping节点可使用各种参数自由配置。gmapping节点的参数在chefbot_bringup/launch/include/gmapping.launch.xml文件中定义。以下是该文件的代码片段及用途：

```
<launch>
  <arg name="scan_topic" default="scan" />

  <!-- Starting gmapping node -->
  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
    output="screen">

  <!-- Frame of mobile base -->
  <param name="base_frame" value="base_footprint"/>
  <param name="odom_frame" value="odom"/>
  <!-- The interval of map updation, reducing this value will speed of map
  generation but increase computation load -->
  <param name="map_update_interval" value="5.0"/>
  <!-- Maximum usable range of laser/kinect -->
  <param name="maxUrange" value="6.0"/>
  <!-- Maximum range of sensor, max range should be > maxUrange -->
  <param name="maxRange" value="8.0"/>
  <param name="sigma" value="0.05"/>
  <param name="kernelSize" value="1"/>
</node>
</launch>
```

通过微调这些参数，我们就能提高gmapping节点建图的准确性。

接下来给出gampping节点的启动文件。它位于
chefbot_bringup/launch/includes/gampping_demo.launch。该启动
文件会启动openni_launch文件和depth_to_laserscan节点，还会将深
度图像数据转换为激光扫描数据。启动了Kinect节点后，它将启动
gmapping节点和move_base节点：

```
<launch>
  <!-- Launches 3D sensor nodes -->
  <include file="$(find chefbot_bringup)/launch/3dsensor.launch">
    <arg name="rgb_processing" value="false" />
    <arg name="depth_registration" value="false" />
    <arg name="depth_processing" value="false" />
    <arg name="scan_topic" value="/scan" />
  </include>

  <!-- Start gmapping nodes and its configurations -->
  <include file="$(find
chefbot_bringup)/launch/includes/gmapping.launch.xml"/>

  <!-- Start move_base node and its configuration -->
  <include file="$(find
chefbot_bringup)/launch/includes/move_base.launch.xml"/>
</launch>
```

11.1.7 配置导航软件包集

另一个需要配置的节点是move_base。配置move_base节点时，我们也要配置全局规划器和局部规划器，以及全局代价地图和局部代价地图。首先我们来看一下加载这些配置文件的启动文件。该启动文件（chefbot_bringup/launch/includes/move_base.launch.xml）会加载move_base、planner和costmap的所有参数：

```
<launch>
  <arg name="odom_topic" default="odom" />
  <!-- Starting move_base node -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">

  <!-- common parameters of global costmap -->
  <rosparam file="$(find
chefbot_bringup)/param/costmap_common_params.yaml" command="load"
ns="global_costmap" />

  <!-- common parameters of local costmap -->
  <rosparam file="$(find
chefbot_bringup)/param/costmap_common_params.yaml" command="load"
ns="local_costmap" />

  <!-- local cost map parameters -->
  <rosparam file="$(find
chefbot_bringup)/param/local_costmap_params.yaml" command="load" />

  <!-- global cost map parameters -->
  <rosparam file="$(find
chefbot_bringup)/param/global_costmap_params.yaml" command="load" />

  <!-- base local planner parameters -->
  <rosparam file="$(find
chefbot_bringup)/param/base_local_planner_params.yaml" command="load" />

  <!-- dwa local planner parameters -->
  <rosparam file="$(find
chefbot_bringup)/param/dwa_local_planner_params.yaml" command="load" />

  <!-- move_base node parameters -->
  <rosparam file="$(find chefbot_bringup)/param/move_base_params.yaml"
command="load" />

  <remap from="cmd_vel" to="/cmd_vel_mux/input/navi"/>
  <remap from="odom" to="$(arg odom_topic)"/>
</node>
</launch>
```

现在我们来了解每个配置文件及其参数。

local_costmap和global_costmap的常见配置

本节将讨论局部代价地图和全局代价地图的常见参数。可以利用机器人周围的障碍物来创建代价地图。通过优化微调代价地图的参数，可以提高地图生成的精度。Chefbot机器人使用自定义的costmap_common_params.yaml文件。该配置文件包含全局代价地图和局部代价地图的通用参数，位于chefbot_bringup/param文件夹中。有关代价地图(costmap)常用参数的进一步信息，请查看http://wiki.ros.org/costmap_2d/flat:

```
#The maximum value of height which has to be taken as an obstacle
max_obstacle_height: 0.60
#This parameters set the maximum obstacle range. In this case, the robot
will only look at obstacles within 2.5 meters in front of robot
obstacle_range: 2.5
#This parameter helps robot to clear out space in front of it upto 3.0
meters away given a sensor reading
raytrace_range: 3.0
#If the robot is circular, we can define the robot radius, otherwise we
need to mention the robot
footprint
robot_radius: 0.45
#footprint: [[-0.,-0.1],[-0.1,0.1], [0.1, 0.1], [0.1,-0.1]]
#This parameter will actually inflate the obstacle up to this distance from
the actual obstacle. This can be taken as a tolerance value of obstacle.
The cost of map will be same as the actual obstacle up to the inflated
value.
inflation_radius: 0.50
#This factor is used for computing cost during inflation
cost_scaling_factor: 5
#We can either choose map type as voxel which will give a 3D view of the
world, or the other type, costmap which is a 2D view of the map. Here we
are opting voxel.
map_type: voxel
#This is the z_origin of the map if it voxel
origin_z: 0.0
#z resolution of map in meters
z_resolution: 0.2
#No of voxel in a vertical column
z_voxels: 2
#This flag set whether we need map for visualization purpose
publish voxel map: false
#A list of observation source in which we get scan data and its parameters
observation_sources: scan
#The list of scan, which mention, data type of scan as LaserScan, marking
and clearing indicate whether the laser data is used for marking and
clearing costmap.
scan: {data_type: LaserScan, topic: scan, marking: true, clearing: true,
min_obstacle_height: 0.0, max_obstacle_height: 3}
```

讨论了常用参数后，现在我们来看全局代价地图的配置。

配置全局代价地图参数

下面是构建全局代价地图所需的主要配置。代价地图的参数定义存储在chefbot_bringup/param/global_costmap_params.yaml文件中。以下是该文件的定义及用法：

```
global_costmap
  global_frame: /map
  robot_base_frame: /base_footprint
  update_frequency: 1.0
```

```
publish_frequency: 0.5
static_map: true
transform_tolerance: 0.5
```

这里的global_frame是/map，它是代价地图的坐标系。
robot_base_frame参数是/base_footprint，它是代价地图需要参考的机器人底盘的坐标系。update_frequency是代价地图运行其主循环刷新的频率。代价地图的发布频率publish_frequency是0.5。如果使用现有的地图，我们必须将static_map设置为true。否则，我们需要将其设置为false。transform_tolerance是执行坐标变换需达到的频率。如果坐标变换未以该频率刷新，机器人将停止工作。

配置局部代价地图参数

下面是该机器人的局部代价地图的配置。该配置文件位于chefbot_bringup/param/local_costmap_params.yaml：

```
local_costmap:  
  global_frame: odom  
  robot_base_frame: /base_footprint  
  update_frequency: 5.0  
  publish_frequency: 2.0  
  static_map: false  
  rolling_window: true  
  width: 4.0  
  height: 4.0  
  resolution: 0.05  
  transform_tolerance: 0.5
```

global_frame、robot_base_frame、publish_frequency、static_map与全局代价地图是一样的。rolling_window参数使代价地图以机器人为中心。如果我们将此参数设置为true，那么我们将以机器人为中心创建代价地图。地图宽度、长度和分辨率即代价地图的宽度、长度和分辨率。下一步就可以配置基础局部规划器了。

配置基础局部规划器参数

局部规划器的主要功能是根据ROS节点发送的目标来计算控制移动的速度。该文件主要包含与速度和加速度等有关的配置。机器人的局部规划器配置文件位于

chefbot_bringup/param/base_local_planner_params.yaml。该文件的定义如下：

```
TrajectoryPlannerROS
# Robot Configuration Parameters, these are the velocity limit of the robot
max_vel_x: 0.3
min_vel_x: 0.1
#Angular velocity limit
max_vel_theta: 1.0
min_vel_theta: -1.0
min_in_place_vel_theta: 0.6
#These are the acceleration limits of the robot
acc_lim_x: 0.5
acc_lim_theta: 1.0
# Goal Tolerance Parameters: The tolerance of robot when it reach the goal
position
yaw_goal_tolerance: 0.3
```

```
xy_goal_tolerance: 0.15
# Forward Simulation Parameters
sim_time: 3.0
vx_samples: 6
vtheta_samples: 20
# Trajectory Scoring Parameters
meter_scoring: true
pdist_scale: 0.6
gdist_scale: 0.8
occdist_scale: 0.01
heading_lookahead: 0.325
dwa: true
# Oscillation Prevention Parameters
oscillation_reset_dist: 0.05
# Differential-drive robot configuration : If the robot is holonomic
configuration, set to true other vice set to false. Chefbot is a non
holonomic robot.
holonomic_robot: false
max_vel_y: 0.0
min_vel_y: 0.0
acc_lim_y: 0.0
vy_samples: 1
```

配置DWA局部规划器参数

DWA规划器是ROS中的另一种局部规划器。它的配置与前面的局部规划器的配置几乎一样。该规划器位于`chefbot_bringup/param/dwa_local_planner_params.yaml`。在这里我们既可以用前面的局部规划器也可以用DWA局部规划器。

配置move_base节点参数

move_base节点需要做一些配置工作。move_base节点的配置文件位于param文件夹中。下面是`move_base_params.yaml`的内容：

```
#This parameter determine whether the cost map need to shutdown when
move_base in inactive state
shutdown_costmaps: false
#The rate at which move base run the update loop and send the velocity
commands
controller_frequency: 5.0
#Controller wait time for a valid command before a space-clearing
operations
controller_patience: 3.0
#The rate at which the global planning loop is running, if it is 0, planner
only plan when a new goal is received
planner_frequency: 1.0
#Planner wait time for finding a valid path before the space-clearing
operations
planner_patience: 5.0
#Time allowed for oscillation before starting robot recovery operations
oscillation_timeout: 10.0
#Distance that robot should move to be considered which not be oscillating.
Moving above this distance will reset the oscillation_timeout
oscillation_distance: 0.2
# local planner - default is trajectory rollout
base_local_planner: "dwa_local_planner/DWAPlannerROS"
```

我们已经讨论了导航软件包集，以及gmapping节点和move_base节点的大部分参数，现在我们可以运行gmapping节点来演示如何构建地图。

启动机器人的tf节点和底盘控制器节点：

```
$ rosrun chefbot Bringup robot_standalone.launch
```

使用以下命令启动gmapping节点：

```
$ rosrun chefbot Bringup gmapping_demo.launch
```

gmapping_demo.launch启动了OpenNI驱动，也启动了将深度数据转换为激光扫描数据的节点，从而可以构建3D传感器的数据流。另外，用必备的参数启动了gmapping节点和move_base节点。现在我们可以启动teleop（遥控）节点让机器人移动，从而构建当前环境的地图。下面的命令将启动遥控机器人移动的teleop节点：

```
$ rosrun chefbot Bringup keyboard_teleop.launch
```

可以在RViz中查看正在构建的地图，使用下面的命令来执行该操作：

```
$ rosrun chefbot Bringup view_navigation.launch
```

在一个普通的房间内测试这个机器人，我们将该机器人移动到房间内的所有区域。如果遍历一遍整个房间，我们将得到一张地图，如图11-9所示。

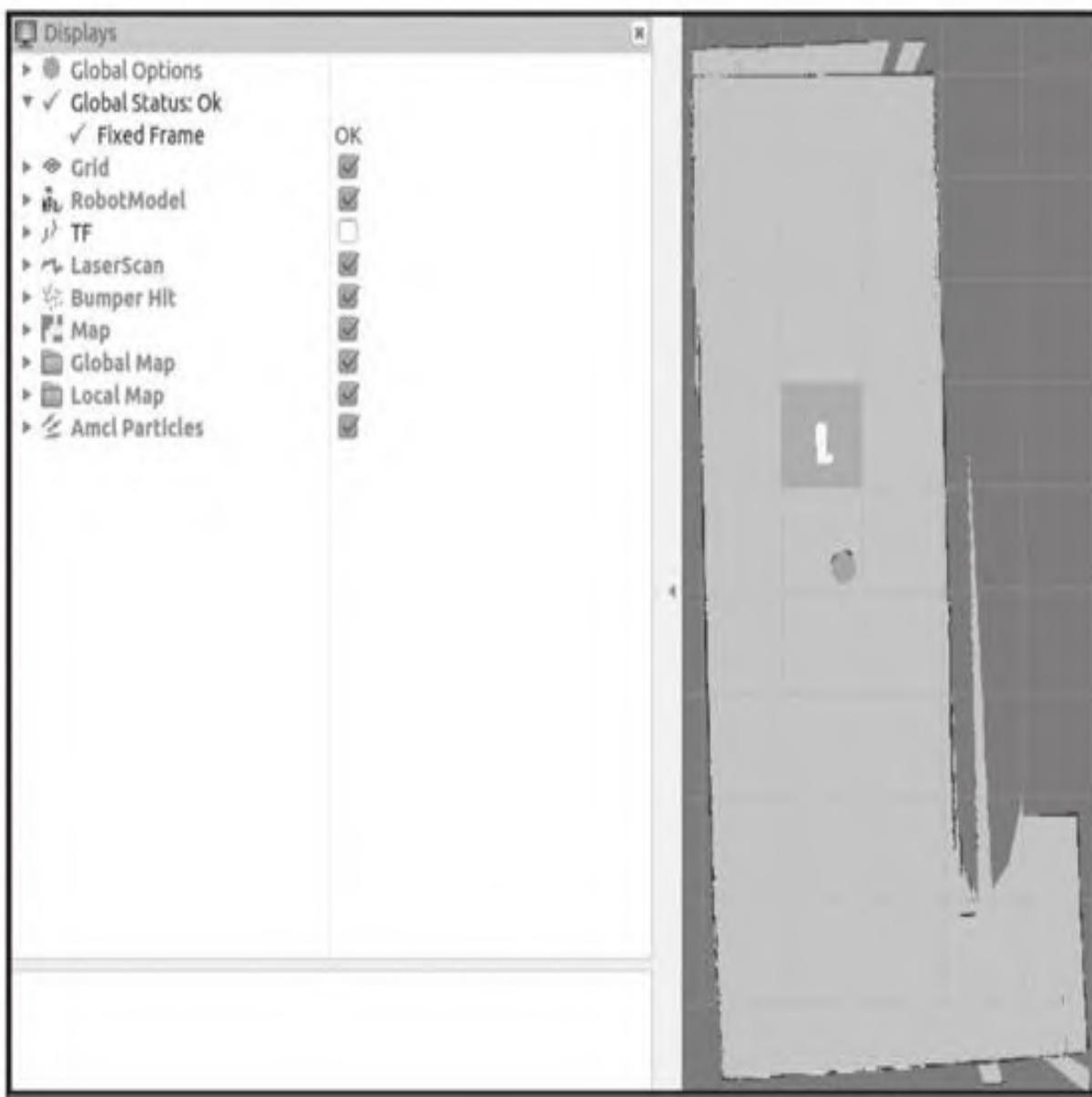


图11-9 在RViz中显示gmapping构建的地图

地图构建完成后，我们可以使用下面的命令保存地图：

```
$ rosrun map_server map_saver -f /home/lentin/room
```

ROS中的map_server软件包包含map_server节点，该节点将当前地图的数据作为一个ROS服务。该软件包提供了名为map_saver的命令行

工具，用来保存地图。

当前地图被保存为两个文件：room.pgm和room.yaml。第一个文件是地图数据，第二个文件是元数据，包含地图文件名及其参数。图11-10所示的截图显示了使用map_server工具生成地图的过程，该地图被保存在当前用户的home文件夹中。

```
lentin@lentin-Aspire-4755:~$ rosrun map_server map_saver -f room
[ INFO] [1441544530.992319268]: Waiting for the map
[ INFO] [1441544531.226293214]: Received a 2560 X 2336 map @ 0.010 m/pix
[ INFO] [1441544531.226483203]: Writing map occupancy data to room.pgm
[ INFO] [1441544531.497796388, 101.846000000]: Writing map occupancy data to room.yaml
[ INFO] [1441544531.498148723, 101.846000000]: Done
```

图11-10 保存地图时的终端消息

下面是room.yaml文件的内容：

```
image: room.pgm
resolution: 0.010000
origin: [-11.560000, -11.240000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

各参数的定义如下：

- image：这个图像包含占据栅格数据。该数据相对于该YAML文件提及的原点，可以是绝对值也可以是相对值。
- resolution：该参数是地图的分辨率，单位是米/像素。
- origin：这是地图左下角像素的2D姿态（x、y、yaw），其中偏角是逆时针为正（yaw=0表示没有旋转）。

- negate: 该参数可以反转地图的白/黑，自由空间/占据空间。
- occupied_thresh: 这是决定像素是否被障碍物占据的阈值。如果占据概率大于该阈值，则被视为完全占据。
- free_thresh: 若地图上的像素被障碍物占据的概率小于以上阈值，则被视为自由空间。构建完环境地图后，我们可以退出所有终端，重新运行下面的命令来启动AMCL。

启动AMCL节点之前，我们先了解AMCL的配置和主要功能。

11.1.8 理解AMCL

构建好环境地图后，接下来需要实现是机器人定位。机器人需要在生成的地图上确定自己的位置。在第6章中，我们已经接触过AMCL。本节，我们将了解amcl软件包的更多细节，以及在Chefbot机器人中用到的amcl启动文件。AMCL实现了机器人在2D环境中的概率定位技术。该算法采用粒子滤波，实现了机器人在已知地图中的姿态跟踪。想了解更多关于定位的技术，请参阅Thrun的著作《Probabilistic Robotics》（<http://www.probabilistic-robotics.org/>）。AMCL算法是在AMCL ROS软件包（<http://wiki.ros.org/amcl>）中实现的。该软件包有一个AMCL节点，订阅了scan（sensor_msgs/LaserScan）、tf（tf/tfMessage）、初始姿态（geometry_msgs/PoseWithCovarianceStamped）、map（nav_msgs/OccupancyGrid）等话题。

处理完传感器数据后，AMCL发布了
amcl_pose（geometry_msgs/PoseWithCovarianceStamped）、
particlecloud（geometry_msgs/PoseArray）、tf（tf/Message）等消息。

amcl_pose是处理后的机器人姿态估计，其中，粒子云是由滤波器维护的姿态估计集。

如果没有设置机器人的初始姿态，粒子将位于原点附近。我们可以在RViz中利用2D Pose estimate选项按钮设置机器人的初始姿态。我们不妨看看该机器人使用的amcl启动文件。下面就是启动amcl的启动文件amcl_demo.launch的主要内容：

```
<launch>
  <rosparam command="delete" ns="move_base" />
  <include file="$(find chefbot_bringup)/launch/3dsensor.launch">
    <arg name="rgb_processing" value="false" />
    <arg name="depth_registration" value="false" />
    <arg name="depth_processing" value="false" />
    <!-- We must specify an absolute topic name because if not it will be
        prefixed by "$(arg camera)". -->
    <arg name="scan_topic" value="/scan" />
  </include>

  <!-- Map server -->
  <arg name="map_file" default="$(find
turtlebot_navigation)/maps/willow-2010-02-18-0.10.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />

  <arg name="initial_pose_x" default="0.0"/> <!-- Use 17.0 for willow's map
in simulation -->
  <arg name="initial_pose_y" default="0.0"/> <!-- Use 17.0 for willow's map
in simulation -->
  <arg name="initial_pose_a" default="0.0"/>

  <include file="$(find chefbot_bringup)/launch/includes/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <include file="$(find
chefbot_bringup)/launch/includes/move_base.launch.xml"/>

</launch>
```

该启动文件启动了与3D传感器相关的节点，还启动了提供地图数据的地图服务节点，执行定位的amcl节点，以及控制机器人移动的move_base节点。

amcl完整的启动参数在另外一个名为amcl.launch.xml的文件中，位于chefbot_bringup/launch/include文件夹中。下面是该文件的定义：

```
<launch>
  <arg name="use_map_topic" default="false"/>
  <arg name="scan_topic" default="scan"/>
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>

  <node pkg="amcl" type="amcl" name="amcl">
    <param name="use_map_topic" value="$(arg use_map_topic)"/>
    .....
    .....

    <!-- Increase tolerance because the computer can get quite busy -->
    <param name="transform_tolerance" value="1.0"/>
    <param name="recovery_alpha_slow" value="0.0"/>
    <param name="recovery_alpha_fast" value="0.0"/>
    <param name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <param name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <param name="initial_pose_a" value="$(arg initial_pose_a)"/>
    <remap from="scan" to="$(arg scan_topic)"/>
  </node>
</launch>
```

想了解这些参数的详细信息，可以参考ROS amcl软件包的维基页面。我们将学习如何在现有的地图上对机器人进行定位和路径规划。

使用下面的命令启动机器人硬件节点：

```
$ rosrun chefbot_bringup robot_standalone.launch
```

使用下面的命令运行amcl的启动文件：

```
$ rosrun chefbot_bringup amcl_demo.launch  
map_file:=/home/lentin/room.yaml
```

我们可以使用下面的命令启动RViz，然后控制机器人移动到地图上特定的位置：

```
$ rosrun chefbot_bringup view_navigation.launch
```

图11-11是RViz的截屏。

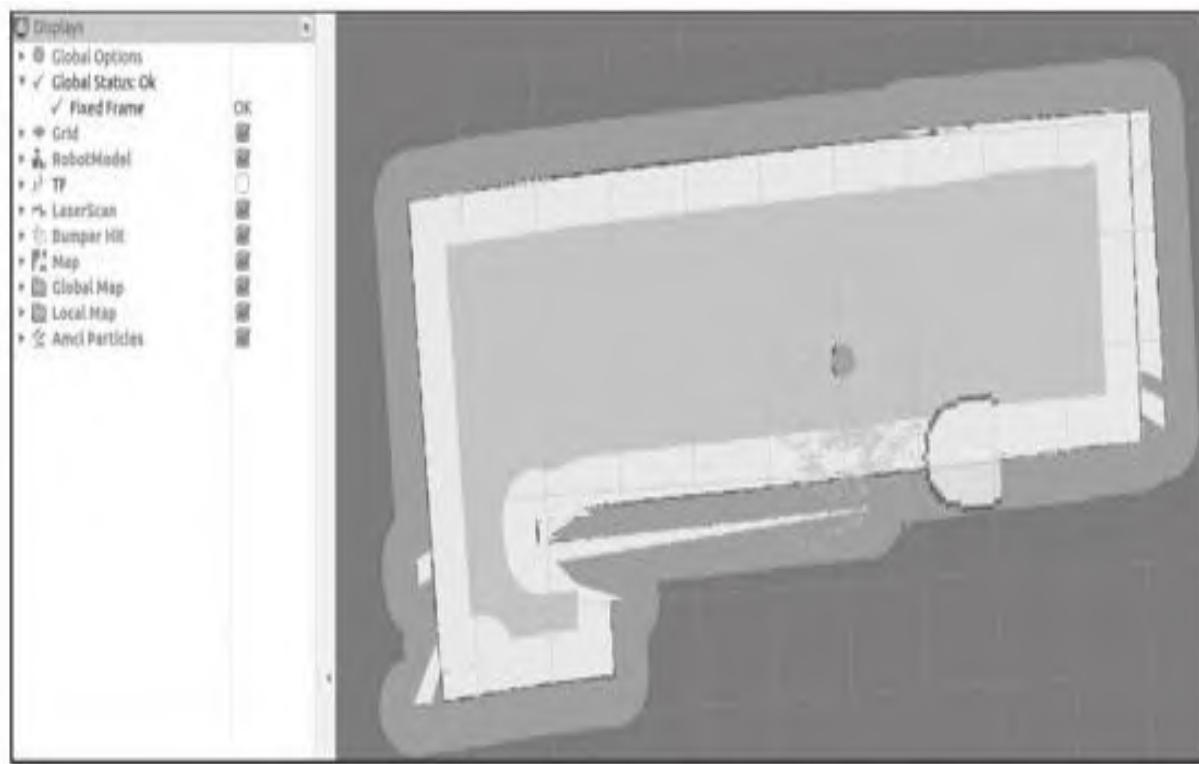


图11-11 用AMCL实现机器人自主导航

下一节，我们将了解RViz中各选项的含义，并了解如何在地图上控制机器人。

11.1.9 在RViz中使用导航功能

我们将探索RViz中的各种GUI选项以可视化显示导航中的每个参数。

2D姿态估计（Pose Estimation）按钮

在RViz中的第一步就是在地图上设置机器人的初始位置。如果机器人能够在地图上自定位，则无须设置初始位置。否则，我们必须使用RViz中的2D Pose Estimate按钮来设置初始位置，如图11-12所示。

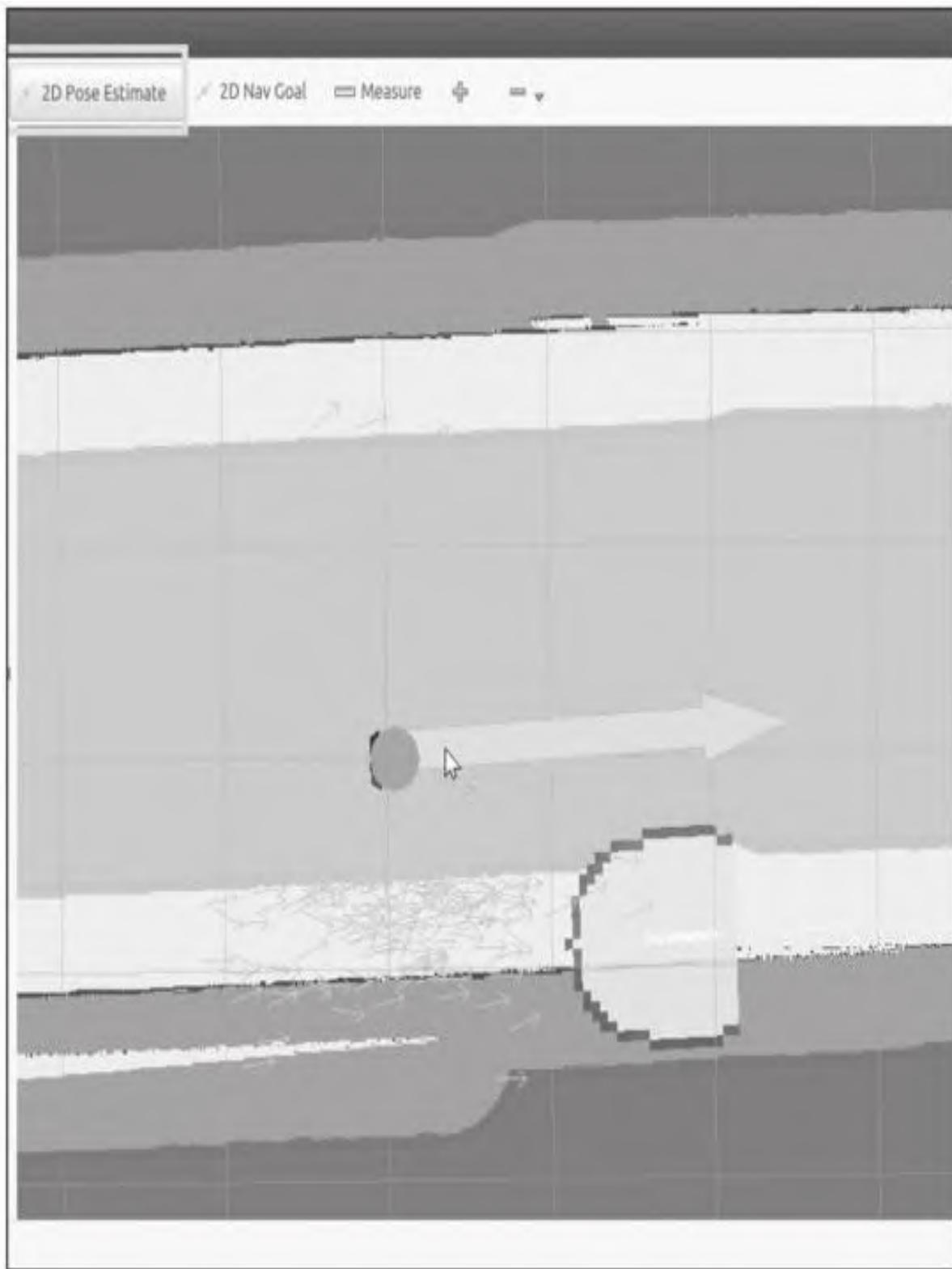


图11-12 RViz中的2D Pose Estimate按钮

按下2D Pose Estimate按钮并用鼠标左键选择一个机器人的姿态，如图11-12所示。检查机器人的实际姿态与机器人在RViz中的模型是否一致。设置好姿态后，我们就可以开始机器人的路径规划了。机器人周围的箭头云是AMCL的粒子云。粒子的数量越多，意味着机器人位置的不确定性越高。如果粒子云数量较少，意味着不确定性很低，机器人几乎可以确定其自身位置了。用来处理机器人初始姿态的话题是：

- 话题名：initialpose
- 话题类型：geometry_msgs/PoseWithCovarianceStamped

粒子云可视化

机器人附近的粒子云可通过PoseArray选项来显示。这里，RViz显示的PoseArray话题是/particlecloud。注意，这里的PoseArray类型被改名为Amcl Particles（见图11-13）：

- 话题名：/particlecloud
- 话题类型：geometry_msgs/PoseArray

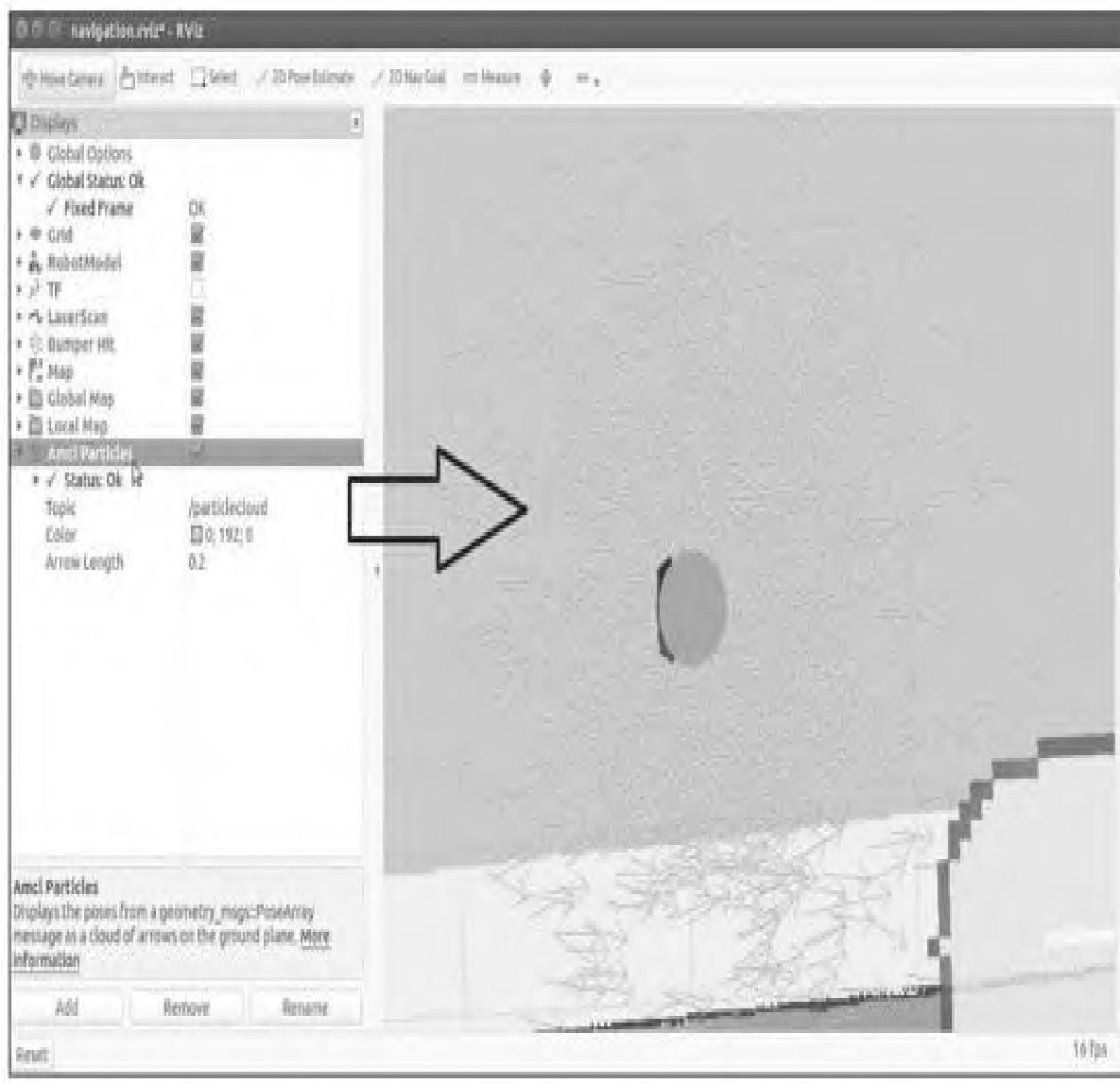


图11-13 可视化AMCL粒子云

2D Nav Goal按钮

通过RViz中的2D Nav Goal按钮，可以向ROS导航软件包集中的move_base节点发送一个目标位置。我们可以从RViz的顶部面板中选择该按钮，然后用鼠标左键点击地图来指定目标位置。该目标位置将发送至move_base节点，用于将机器人移动到该位置，如图11-14所示：

- 话题名：move_base_simple/goal
- 话题类型：geometry_msgs/PoseStamped

显示静态地图

静态地图是我们为map_server节点提供的地图。随后，map_server节点会通过/map话题提供静态地图服务：

- 话题名：/map
- 话题类型：nav_msgs/GetMap

图11-15就是RViz中的静态地图。

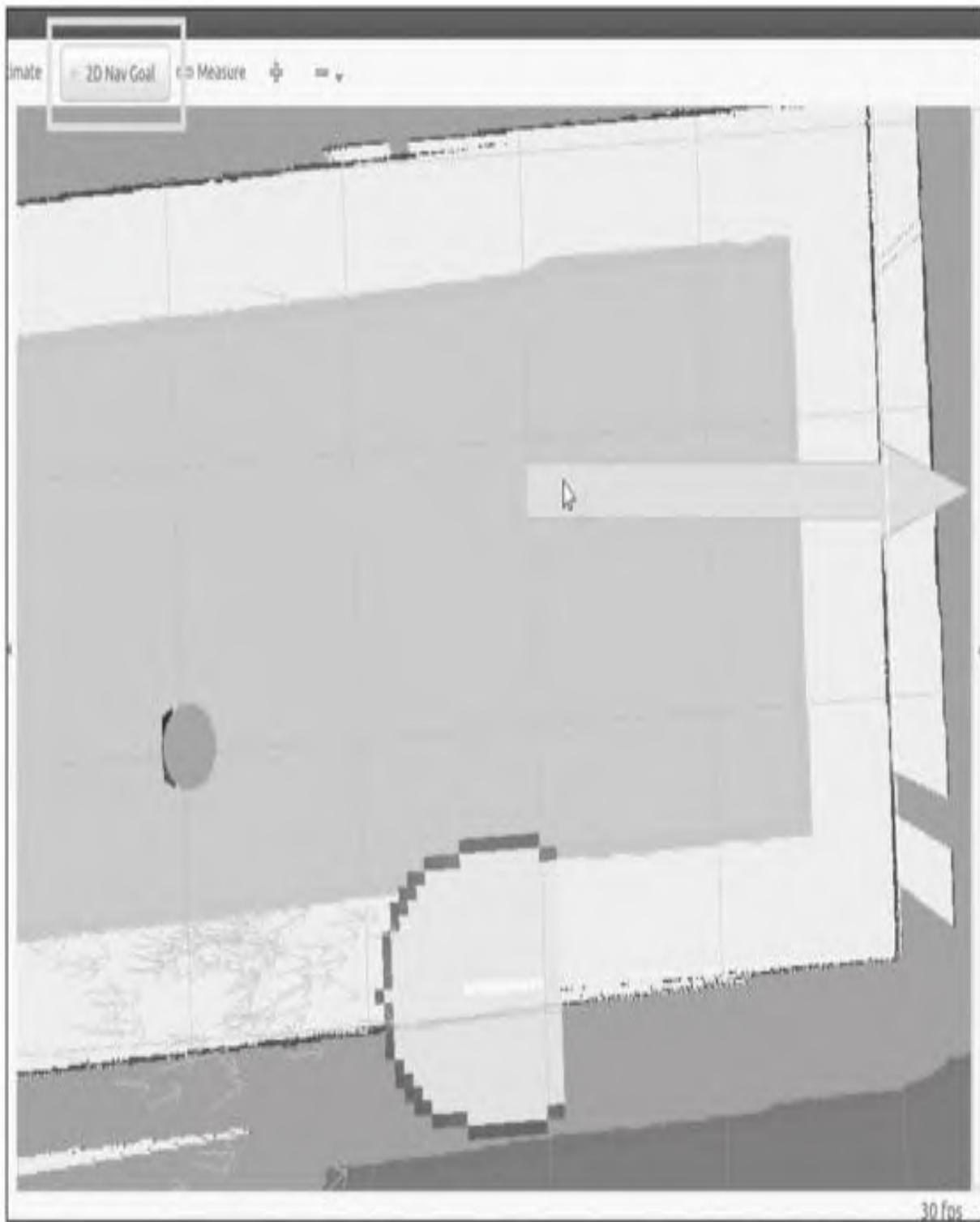


图11-14 在RViz中使用2D Nav Goal按钮设置机器人的目标位置

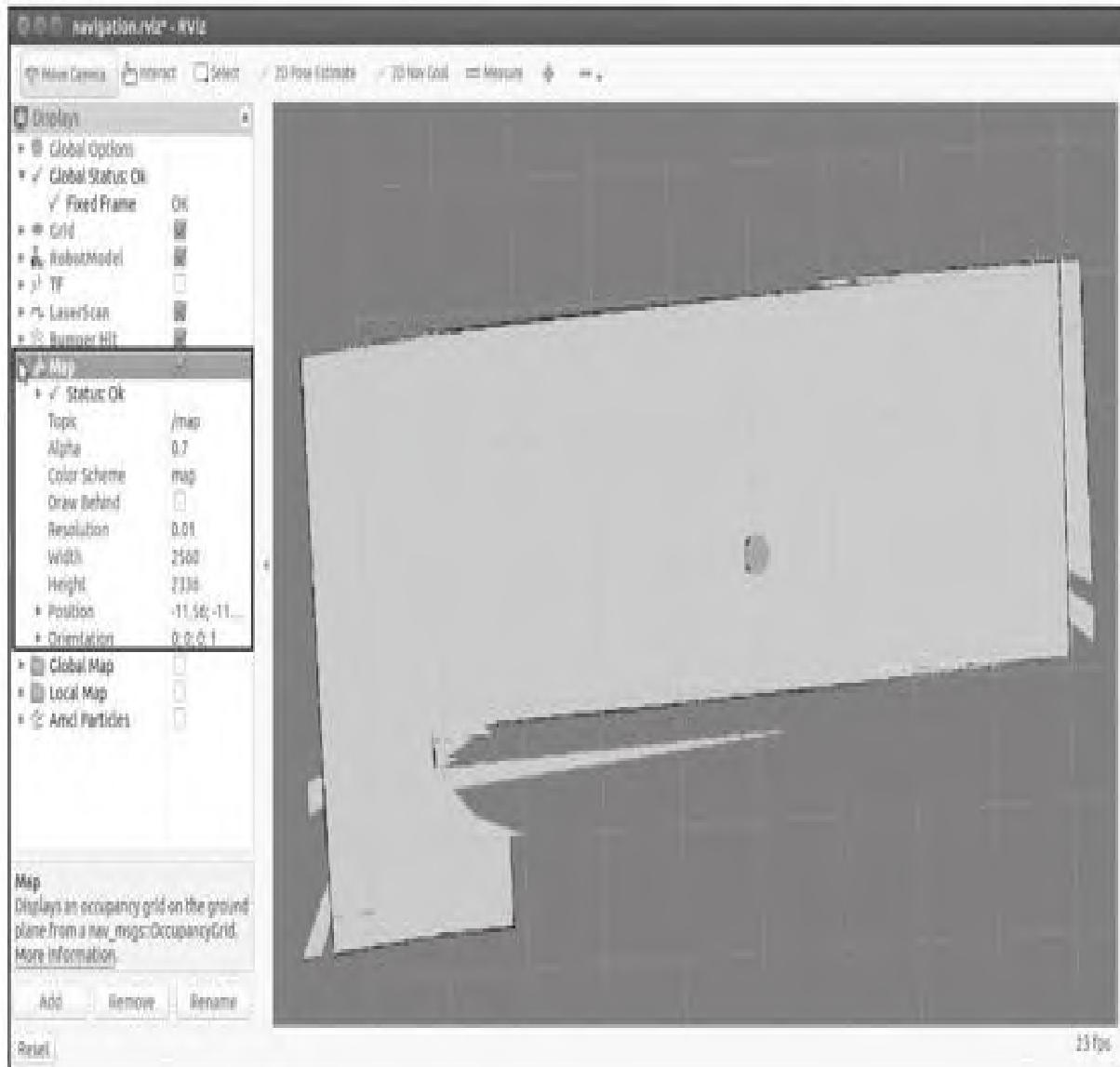


图11-15 在RViz中可视化静态地图

显示机器人足迹

在配置文件`costmap_common_params.yaml`中我们已经定义了机器人的足迹（即轮廓）。该机器人是圆形的，半径为0.45米。在RViz中可以使用多边形（Polygon）来显示机器人足迹。下面是机器人模型的圆形足迹及其发布的话题（见图11-16）：

- 话题
名：/move_base/global_costmap/obstacle_layer_footprint/footprint_stamped

- 话题
名：/move_base/local_costmap/obstacle_layer_footprint/footprint_stamped
- 类型：geometry_msgs/Polygon

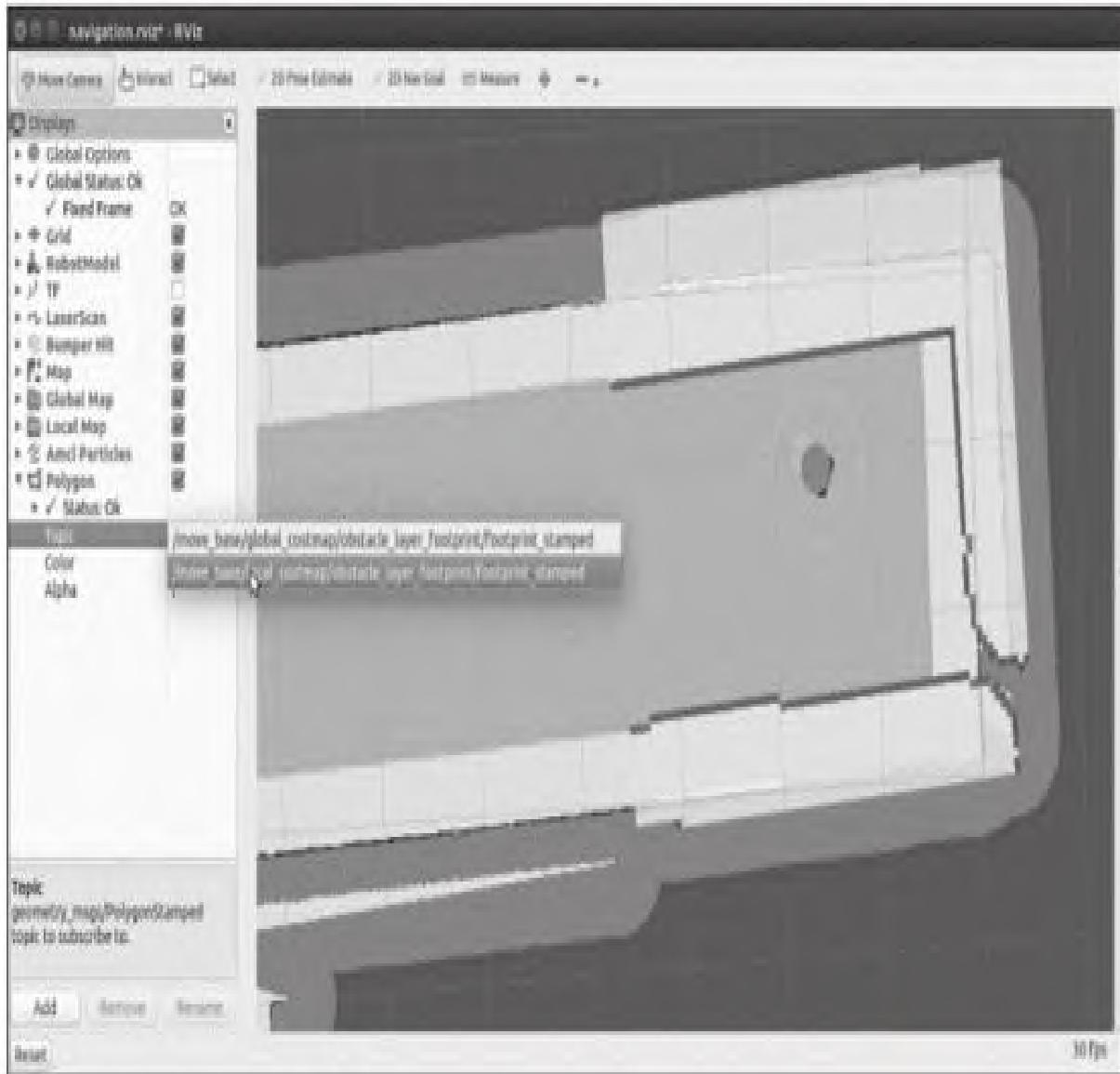


图11-16 在RViz中显示机器人的全局和局部足迹

显示全局和局部代价地图

图11-17所示的RViz截图显示了局部代价地图和全局代价地图，以及真实的障碍物和膨胀的障碍物。每张地图的显示类型都是地图本身：

- 局部代价地图话题: /move_base/local_costmap/costmap
- 局部代价地图话题类型: nav_msgs/OccupancyGrid
- 全局代价地图话题: /move_base/global_costmap/costmap
- 全局代价地图话题类型: nav_msgs/OccupancyGrid

为了避免与真实障碍物碰撞，根据配置文件中的设置，将真实障碍物向外膨胀了一定距离，被称为膨胀的障碍物。机器人只会在膨胀障碍物之外的空间进行路径规划。障碍物膨胀是一种避免与真实障碍物碰撞的技术。

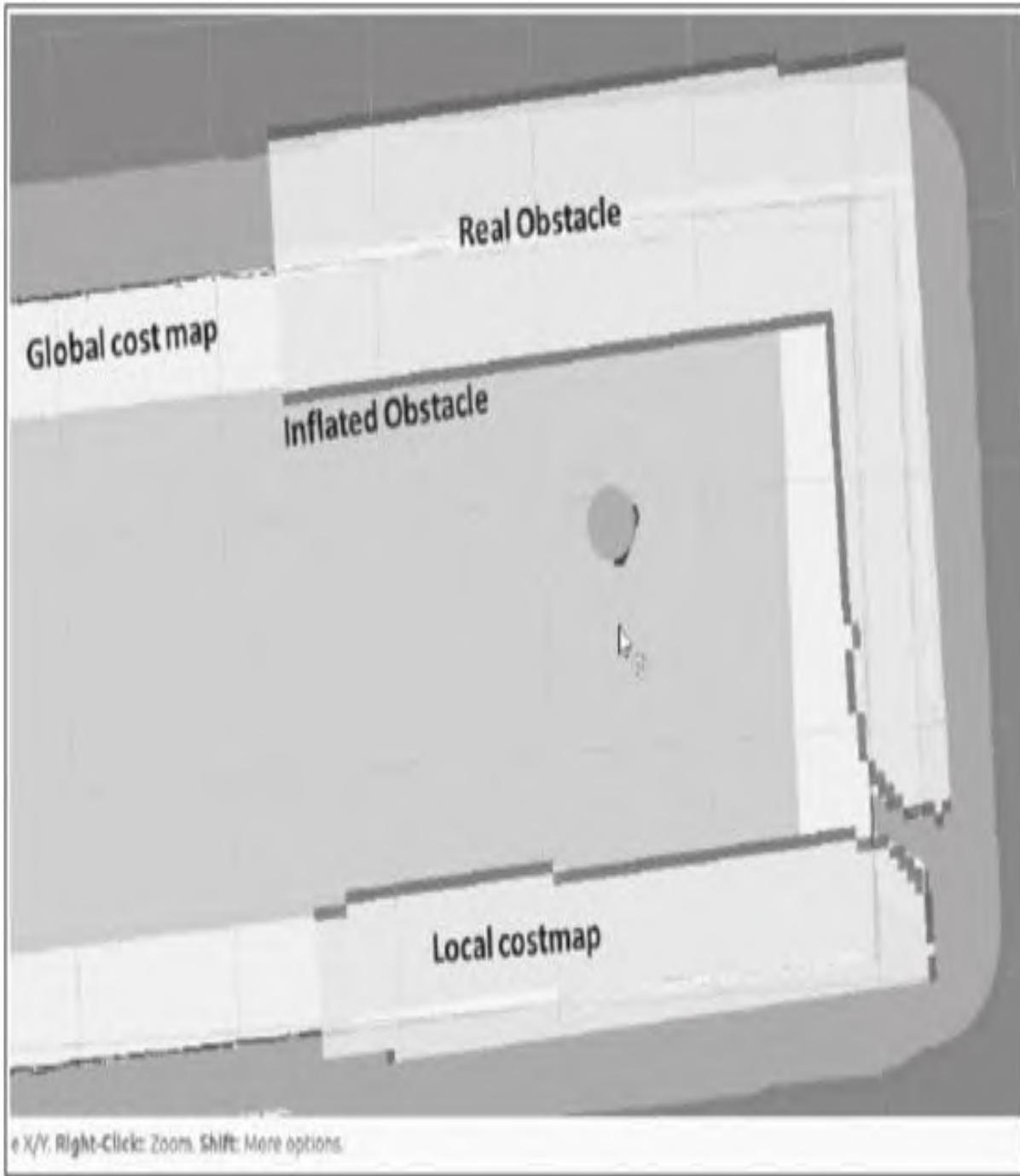


图11-17 全局地图和局部地图的可视化，以及RViz真实的障碍物和膨胀的障碍物

显示全局规划路径、局部规划路径和规划器规划路径

图11-18显示了全局规划路径、局部规划路径以及规划器规划路径。规划器规划路径和全局规划路径都代表要到达目标的完整规划。

局部规划路径是遵循全局规划路径的一种短期路径。若遇到障碍物，全局规划路径和规划器规划路径会随之改变。在RViz中，可以使用Path选项来显示这些规划的路径：

- 全局规划路径话题：/move_base/DWAPlannerROS/global_plan
- 全局规划路径话题类型：nav_msgs/Path
- 局部规划路径话题：/move_base/DWAPlannerROS/local_plan
- 局部规划路径话题类型：nav_msgs/Path
- 规划器规划路径话题：/move_base/NavfnROS/plan
- 规划器规划路径话题类型：nav_msgs/Path

当前目的地

可以用2D Nav Goal按钮或者利用ROS客户端节点，设置机器人的当前导航目的地。图11-19中的红色箭头指示了机器人的当前目的地：

- 话题名：/move_base/current_goal
- 话题类型：geometry_msgs/PoseStamped

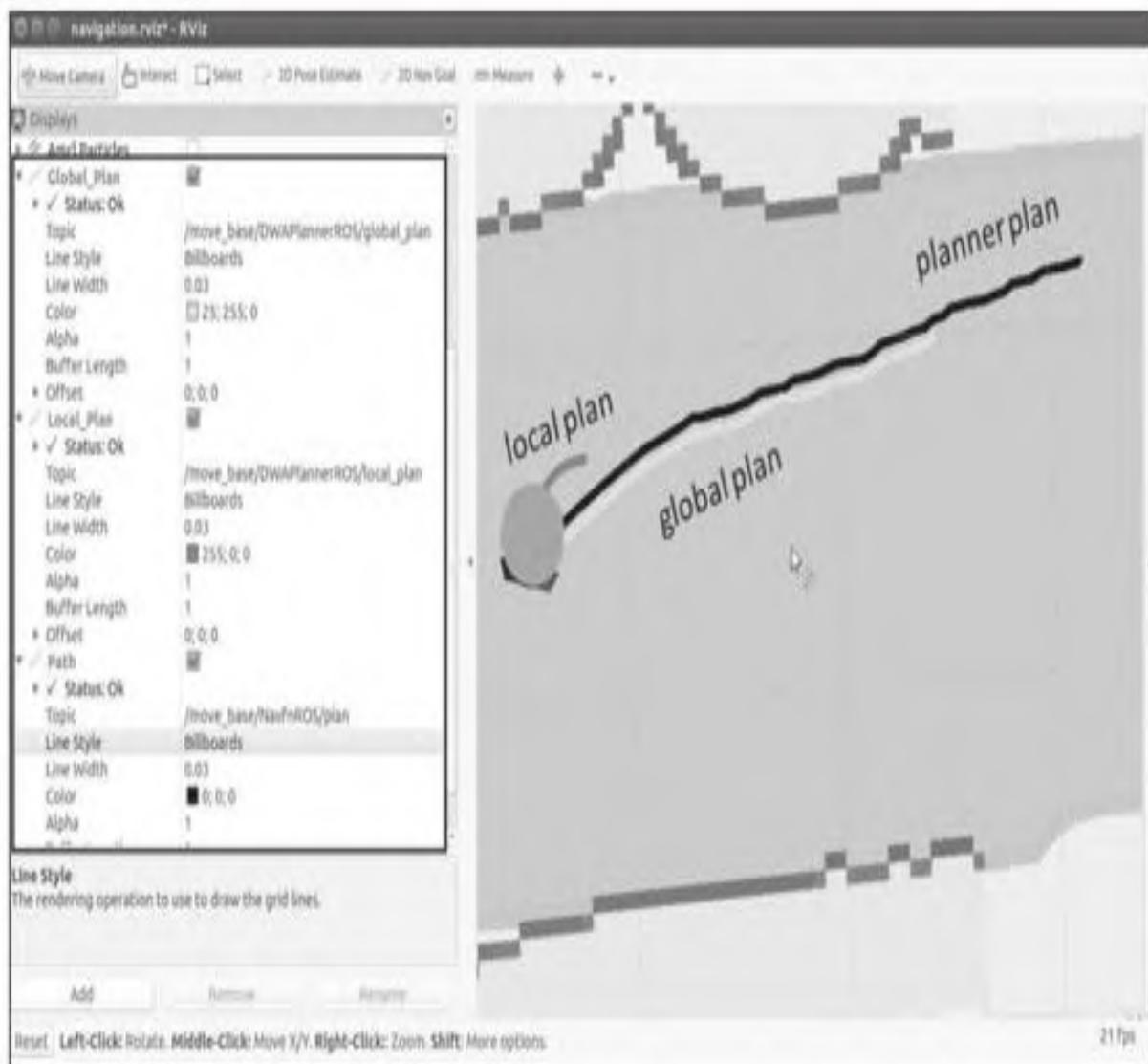


图11-18 在RViz中可视化全局规划路径、局部规划路径和规划器规划路径

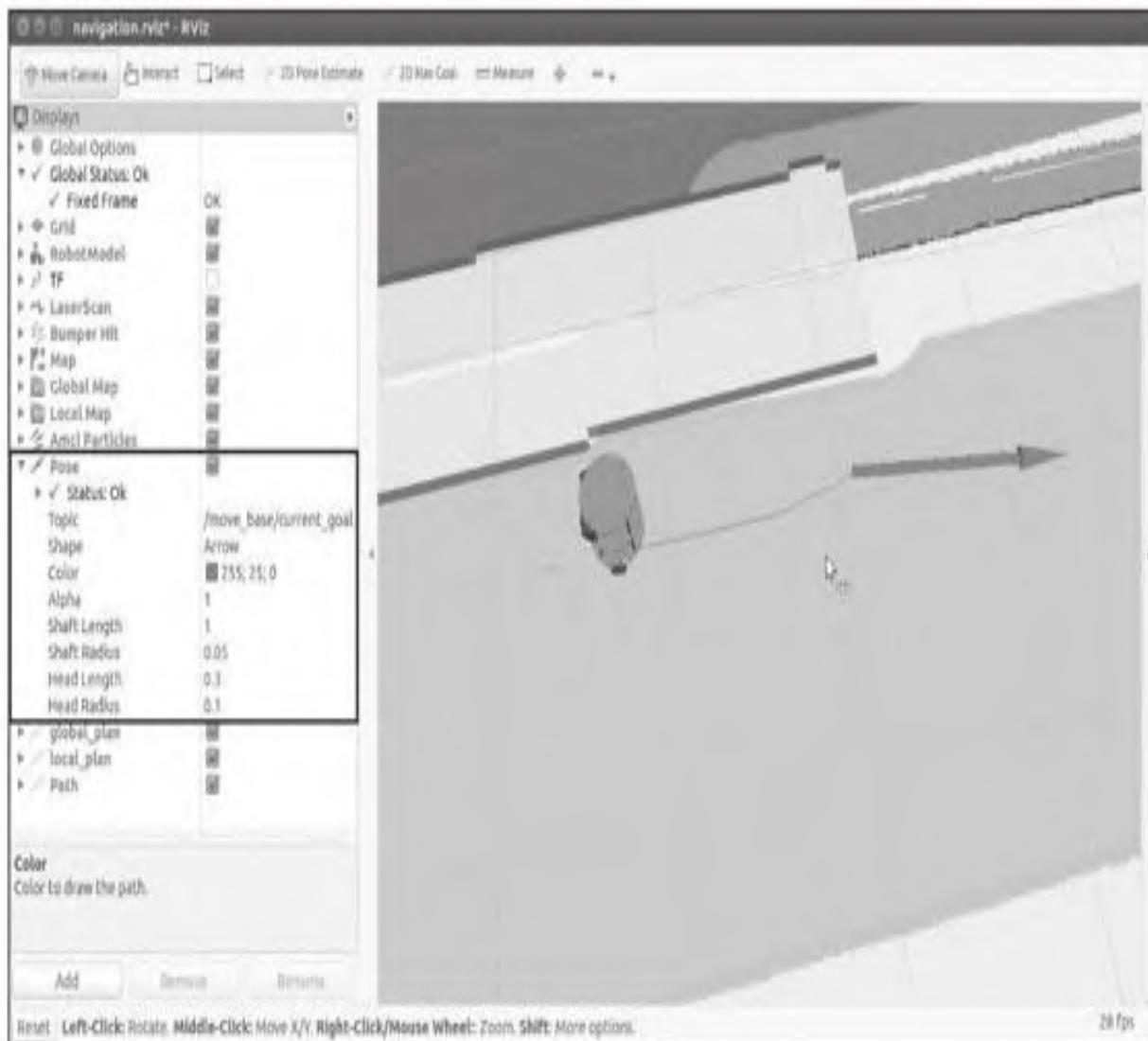
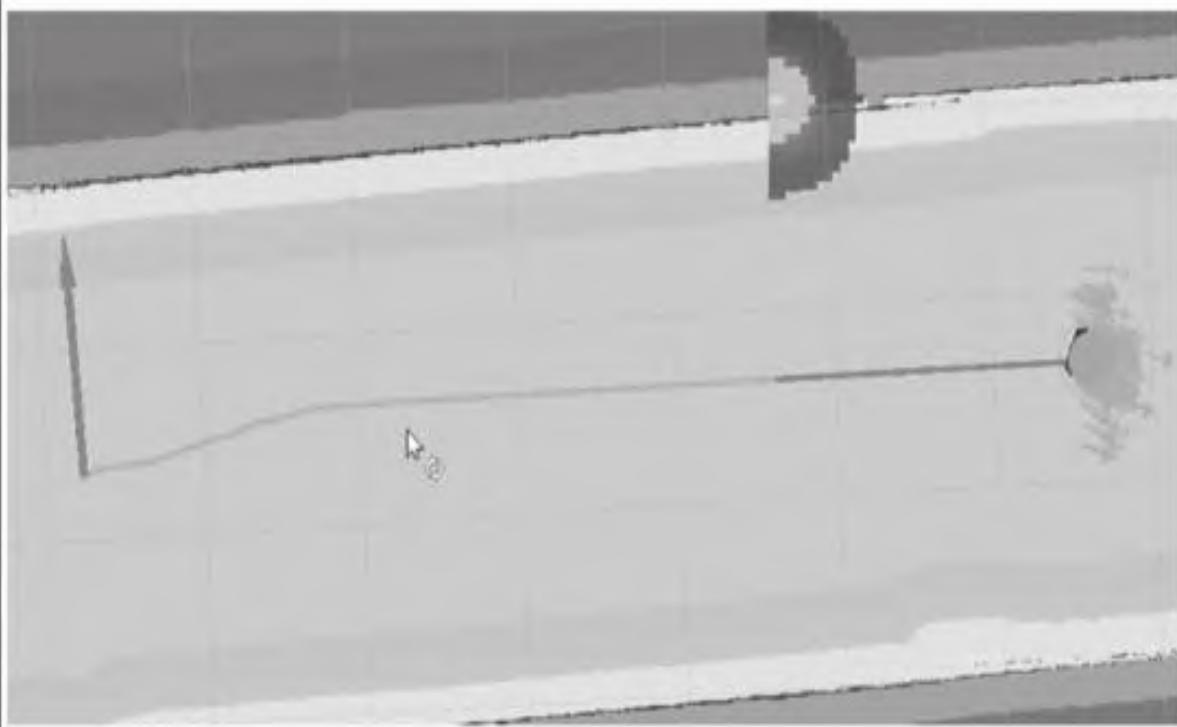


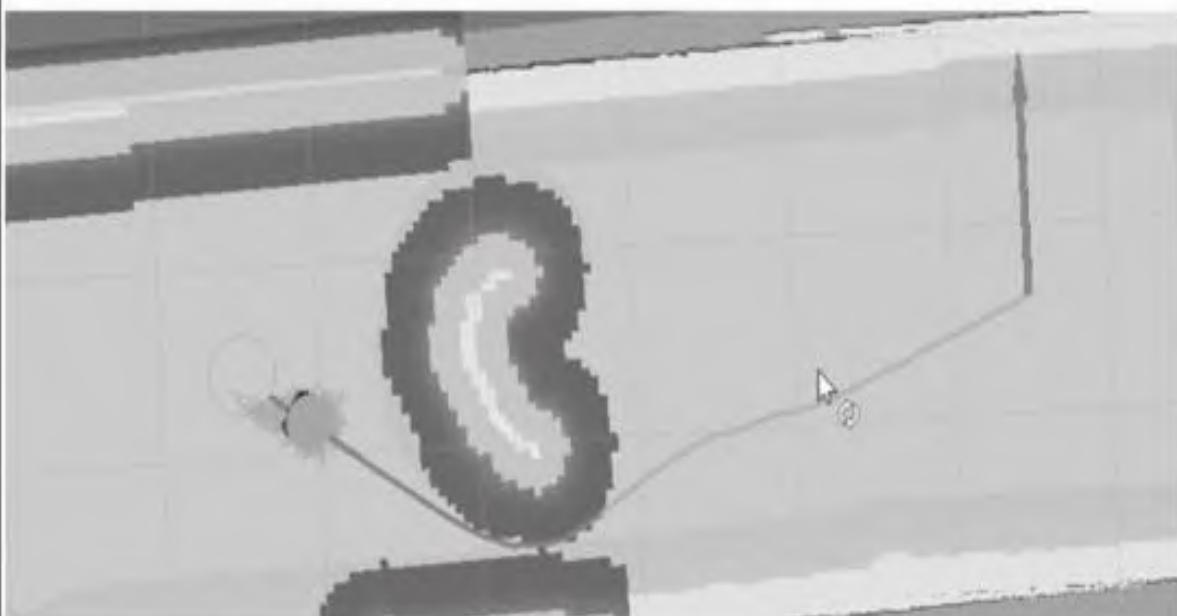
图11-19 显示机器人的目的地

11.1.10 导航软件包中避障

导航软件包集可用来让机器人在运动中躲避随机出现的障碍物。在下面这个场景中，我们在机器人规划好的路径上放置一个动态障碍物。特别说明的是，图11-20的上半部分显示的是在没有任何障碍物的地图上进行路径规划。一旦在机器人的规划路径上放置一个动态障碍物，我们可以看到一条规避了障碍物的路径被规划出来，如图11-20的下半部分所示。



没有障碍物



有障碍物

图11-20 在RViz中显示避障功能

11.1.11 Chefbot机器人仿真

chefbot_gazebo仿真软件包是随chefbot_bringup软件包一起发布的，这样我们就可以在Gazebo中进行机器人仿真了。我们将学习如何模仿测试硬件时用的房间，建造一个类似的仿真环境。首先，在Gazebo中创建一个虚拟房间。

在Gazebo中创建虚拟房间

在Gazebo中创建一个房间，保存为语义描述格式（Semantic Description Format, SDF），并将其加载到Gazebo环境中。

在一个空场景中启动Gazebo，并加载Chefbot机器人：

```
$ rosrun chefbot_gazebo chefbot_empty_world.launch
```

该命令在Gazebo空场景中打开Chefbot模型。我们可以用墙壁、窗户、门和楼梯创建一个房间。

Gazebo里有一个Building Editor选项。我们可以从菜单Edit|Building Editor中启动该编辑器。然后，我们将在Gazebo的窗口看到该编辑器：

可以通过单击Gazebo左侧面板中的Add Wall选项来添加墙壁，如图11-21所示。在Building Editor中，我们还可以通过点击鼠标左键来绘制墙壁。我们可以看到，在编辑器中添加墙壁后，会在Gazebo中创建一个真实的3D墙壁。我们正在创建一个与我们测试真实机器人那样类似的房间布局。通过Save As（另存为）选项或者点击Done（完成）按钮，会弹出一个窗口，用来保存这个文件。该文件将以.sdf的格式保存，我们将此示例保存为final_room.sdf。

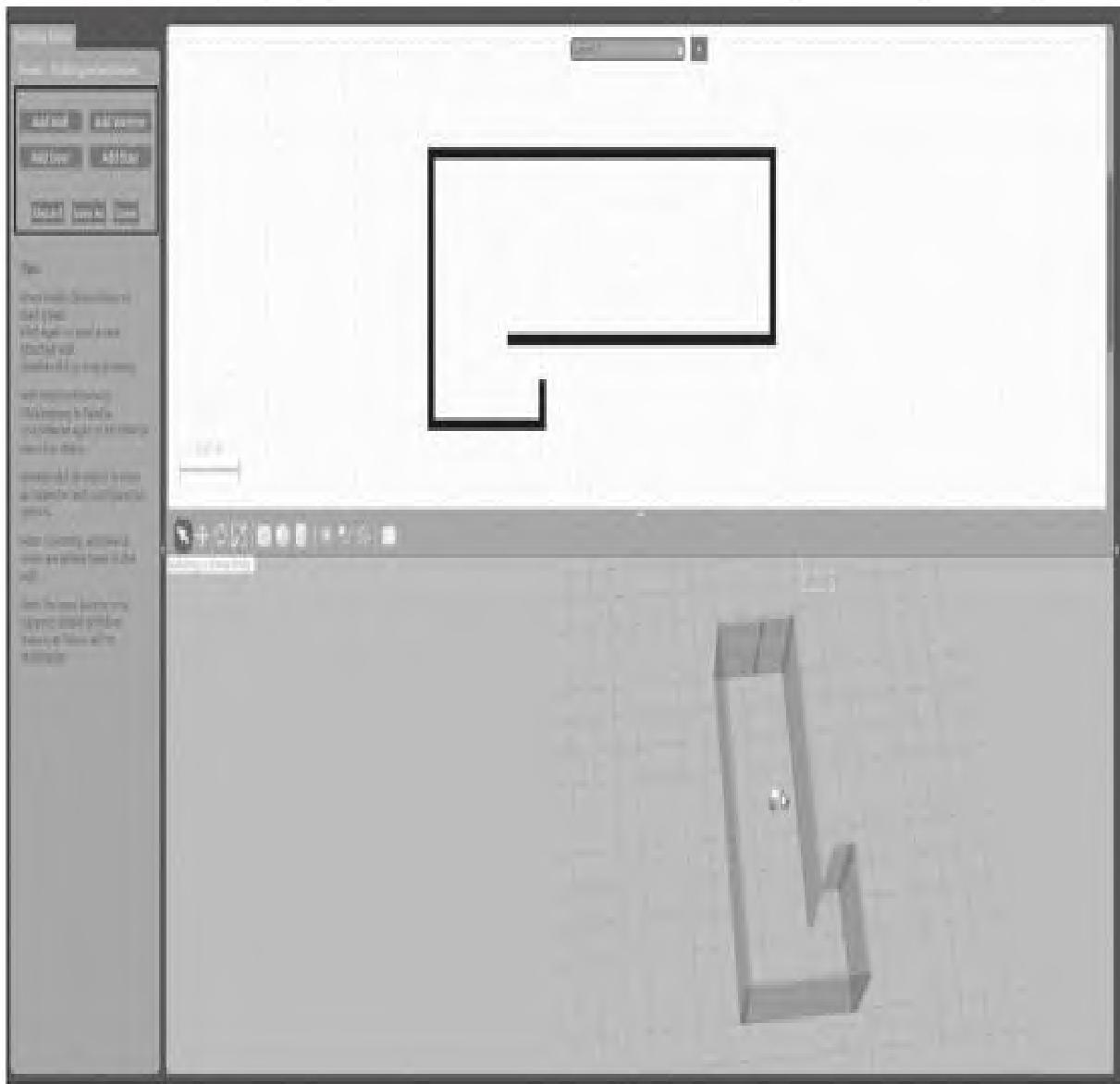


图11-21 在Gazebo中创建墙壁

保存了房间文件后，我们可以将房间模型添加到gazebo models文件夹中，这样我们就可以在任何仿真环境中使用该模型了。

将模型文件添加到Gazebo模型文件夹中

下面就是将模型文件添加到Gazebo文件夹中的过程。

Gazebo的默认模型文件夹位于`~/.gazebo/models`。我们可以在此创建一个名为`final_room`的文件夹，然后将`final_room.sdf`复制到该

文件夹中。另外，还需要创建一个名为model.config的文件，它包含模型文件的详细信息。该文件的定义如下：

```
<?xml version="1.0"?>
<model>
    <!-- 在 Gazebo 中显示的模型名称 -->
    <name>Test Room</name>
    <version>1.0</version>
    <!-- Model file name -->
    <sdf version="1.2">final_room.sdf</sdf>

    <author>
        <name>Lentin Joseph</name>
        <email>qboticslabs@gmail.com</email>
    </author>

    <description>
        A test room for performing SLAM
    </description>
</model>
```

模型添加到模型文件夹后，重启Gazebo，然后我们就可以在Insert选项卡中看到名为Test Room的模型，如图11-22所示。在model.config文件中将此模型命名为Test Room，该名称就显示在这个列表中了。我们可以像图11-22所示的那样，选择此模型并将其添加到视图窗口中。

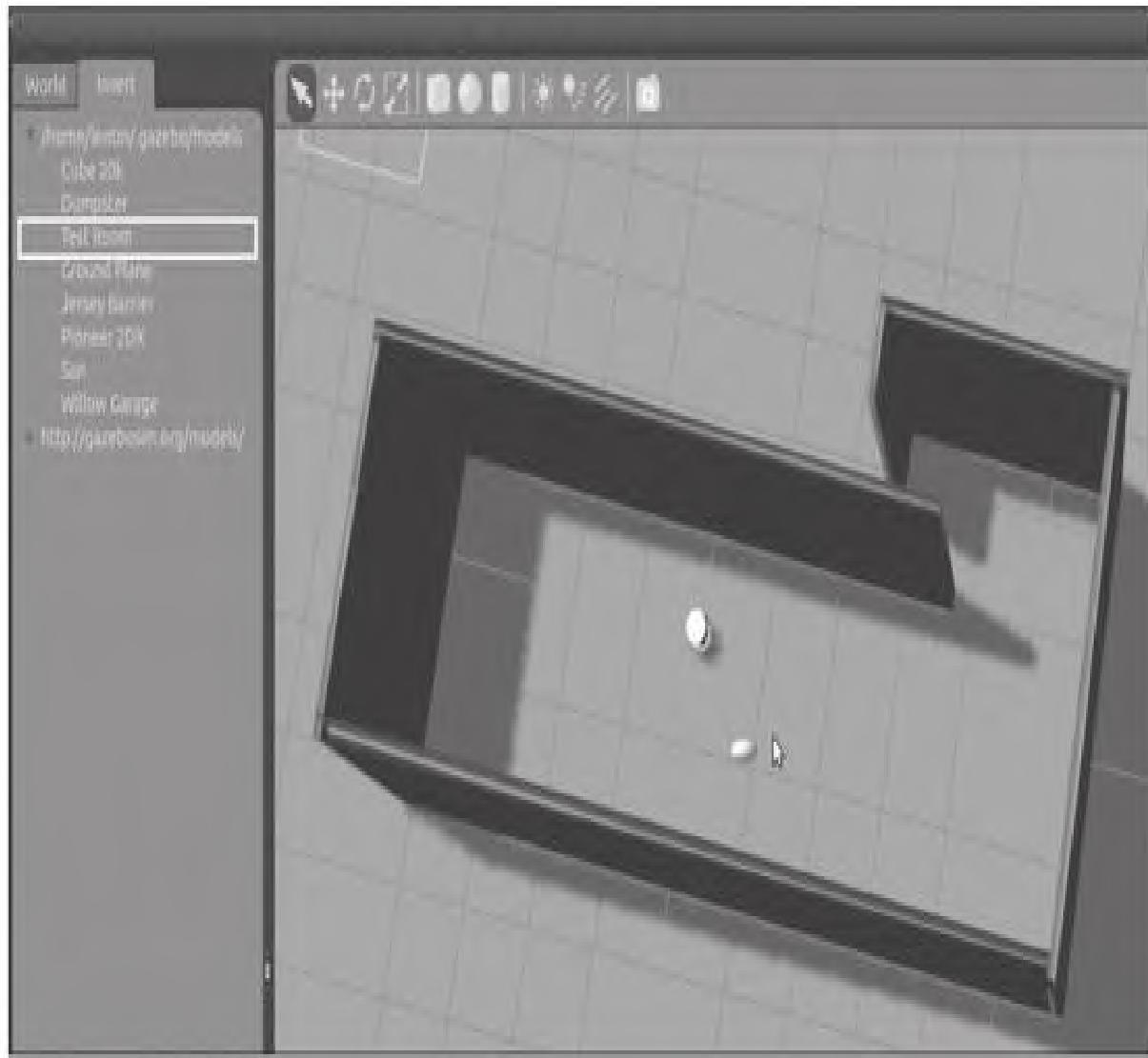


图11-22 在Chefbot机器人仿真环境中添加墙壁

将模型添加到视图窗口后，保存当前仿真环境的配置。从Gazebo的菜单中选择File，然后选择Save world as选项。将文件命名为test_room.sdf，保存到ROS软件包chefbot_gazebo下的worlds文件夹中，然后在chefbot_empty_world.launch文件中添加该文件，并将此启动文件另存为chefbot_room_world.launch文件。该启动文件内容如下：

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="use_sim_time" value="true"/>
  <arg name="debug" value="false"/>

  <!-- Adding world test_room.sdf as argument -->
  <arg name="world_name" value="$(find
chefbot_gazebo)/worlds/test_room.sdf"/>
</include>
```

保存了该启动文件后，运行文件chefbot_room_world.launch即可与机器人真实环境那样进行仿真。我们可以将Gazebo系统提供的障碍物添加到仿真环境中。

现在不用从chefbot_bringup软件包中运行robot_standalone.launch文件来启动机器人硬件了，我们可以运行chefbot_hotel_world.launch启动文件，构建出一个更复杂的机器人仿真环境，并获得odom和tf的仿真数据：

```
$ rosrun chefbot_gazebo chefbot_hotel_world.launch
```

其他操作（例如SLAM和AMCL）与我们的硬件操作相同。接下来的启动文件用来在仿真器中执行SLAM和AMCL。要用gmapping和AMCL程序执行建图和定位，我们需要使用下面的命令。

在仿真器中运行SLAM：

```
$ rosrun chefbot_gazebo gmapping_demo.launch
```

运行遥控（teleop）节点：

```
$ roslaunch chefbot_bringup keyboard_teleop.launch
```

在仿真器中运行AMCL：

```
$ roslaunch chefbot_gazebo amcl_demo.launch
```

在仿真环境中让机器人四处运动，保存生成的地图：

```
$ rosrun map_server map_saver -f /tmp/hotel
```

11.1.12 从ROS节点向导航软件包集发送一个目的地

我们已经知道了如何在RViz中使用2D Nav Goal按钮将目标位置发送给机器人，从而让机器人从A点移动到B点。现在我们将学习如何用actionlib客户端和ROS C++ API来控制机器人。下面是用于与move_base节点通信的示例包和节点。

move_base节点是SimpleActionServer。如果导航任务需要很长一段时间才能完成，我们可以向机器人发送取消命令，从而取消到目标位置的任务。

下面就是为move_base节点编写的代码SimpleActionClient，它可以从命令行发送x、y和theta参数。下面这段代码位于chefbot_bringup/src文件夹中，源码文件名是send_robot_goal.cpp：

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_broadcaster.h>
#include <sstream>
#include <iostream>
// 定义一个新的 SimpleActionClient
move_base_msgs::MoveBaseAction
typedef
actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;

int main(int argc, char** argv){
    ros::init(argc, argv, "navigation_goals");
    // 初始化 move_base 客户端
    MoveBaseClient ac("move_base", true);
    // 等待服务器启动
    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting for the move_base action server");
    }
    // 定义 move base 目标
    move_base_msgs::MoveBaseGoal goal;

    // 在目标动作中设置目标坐标系 id 和时间
    goal.target_pose.header.frame_id = "map";
    goal.target_pose.header.stamp = ros::Time::now();

    // 从命令行中获取姿态参数，否则将使用一套默认的姿态参数
    try{
        goal.target_pose.pose.position.x = atof(argv[1]);
        goal.target_pose.pose.position.y = atof(argv[2]);
        goal.target_pose.pose.orientation.w = atof(argv[3]);
    }
    catch(int e){
        goal.target_pose.pose.position.x = 1.0;
        goal.target_pose.pose.position.y = 1.0;
        goal.target_pose.pose.orientation.w = 1.0;
    }
    ROS_INFO("Sending move base goal");

    // 发送目标值
    ac.sendGoal(goal);
    ac.waitForResult();

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

将下面两行添加到CMakeLists.txt，然后编译这个节点：

```
add_executable(send_goal src/send_robot_goal.cpp)
target_link_libraries(send_goal ${catkin_LIBRARIES})
```

用catkin_make编译软件包，并在Gazebo中使用下面这些命令来测试该客户端是否工作正常：

启动有一个房间的Gazebo仿真：

```
$ rosrun chefbot_gazebo chefbot_room_world.launch
```

用生成的地图启动amcl节点：

```
$ rosrun chefbot_gazebo amcl_demo.launch map_file:=/tmp/hotel.yaml
```

启动RViz进行导航：

```
$ rosrun chefbot_bringup view_navigation.launch
```

运行发送目标位置的节点，然后向move_base发送目标位置：

```
$ rosrun chefbot_bringup send_goal 1 0 1
```

在RViz中用2D Nav Goal按钮在地图上设置所需的机器人姿态。通过打印/move_base/goal话题的内容就可以得知在RViz中设置的目标姿态。我们可以将这些值作为send_goal节点的命令行参数。

11.2 习题

- 使用ROS导航软件包集的基本要求是什么？
- 使用ROS导航软件包集需要的主要配置文件是什么？
- ROS中的AMCL软件包是如何工作的？
- 将目标姿态发送到导航软件包集的方法有哪些？

11.3 本章小结

本章主要介绍了DIY自主移动机器人与ROS导航软件包集的接口，介绍了该机器人的基本情况、必备的硬件组件，以及连接关系框图。我们了解了机器人的固件代码，以及如何将其烧写到真正的机器人上。烧写了固件后，我们学习了如何将它与ROS连接，然后学习了连接机器人LaunchPad控制器的Python节点及其他节点，如将twist消息转换为马达转速，将编码器计数值转换为odom和tf的节点。

讨论了Chefbot机器人节点间相互连接的方法后，我们又介绍了用于计算里程计和底盘控制器节点的C++代码。讨论了这些节点后，我们了解了ROS导航软件包集的详细配置。我们还运行了gmapping和AMCL，并学习了RViz中导航选项的详细介绍。我们还介绍了在Chefbot的仿真中用导航软件包集来避障。模仿真实机器人环境，在Gazebo中建立了一个类似的仿真环境，并在其中执行了SLAM和AMCL。本章的最后，我们学习了如何用actionlib将目标姿态发送到导航软件包集中。

第12章

探索ROS-MoveIt!的高级功能

在上一章中，我们介绍了ROS导航和移动机器人的硬件接口。在本章中，我们将介绍ROS-MoveIt!的高级功能，例如避障、使用3D传感器进行感知、抓取、拾取和放置。在此之后，我们将学习如何将机器人的操控硬件与MoveIt!连接。

本章将介绍以下内容：

- 使用MoveIt! 的C++ API进行机械臂的运动规划避障
- 在MoveIt!和Gazebo中进行感知
- 用MoveIt!操控对象
- 了解机器人硬件DYNAMIXELROS伺服控制器的接口
- 将基于DYNAMIXEL的7-DOF的机械臂与ROS MoveIt!连接

在第4章和第6章，我们都讨论了MoveIt!以及如何在Gazebo中对机械臂进行仿真，如何用MoveIt!进行运动规划。在本章，我们将学习MoveIt!的一些高级功能以及如何将真实的机械臂与ROS MoveIt!连接。

我们要讨论的第一个主题是如何用MoveIt!的C++ API对我们的机器人进行运动规划。

12.1 使用move_group的C++接口进行运动规划

在第6章，我们讨论了如何与机械臂进行交互，如何用MoveIt!的RViz运动规划插件进行路径规划。在本节，我们将学习如何使用move_group的C++ API通过编程方式控制机器人运动。用RViz可以进行运动规划，当然也可以通过move_group的C++ API用编程方式实现运动规划。

使用C++ API的第一步是创建一个依赖MoveIt!的ROS软件包。你可以从本书提供的源码中获取现有的seven_dof_arm_test软件包，或者也可以从下面的代码库中下载：

```
$ git clone https://github.com/jocacace/seven_dof_arm_test.git
```

我们也可以用下面的命令创建同样的软件包：

```
$ catkin_create_pkg seven_dof_arm_test catkin cmake_modules  
interactive_markers moveit_core moveit_ros_perception  
moveit_ros_planning_interface pluginlib roscpp std_msgs
```

12.1.1 使用MoveIt! C++ API规划随机路径

我们将学习的第一个示例是用MoveIt! C++ API进行随机运动规划。你可以从src文件夹中找到源码文件test_random.cpp。下面是代码及其每行的注释。当我们执行这个节点时，它将规划出一条随机路径并执行：

```
#include <moveit/move_group_interface/move_group_interface.h>
int main(int argc, char **argv)
{
    ros::init(argc, argv, "move_group_interface_demo",
    ros::init_options::AnonymousName);
    // 启动一个ROS自旋线程
    ros::AsyncSpinner spinner(1);
    spinner.start();
    // 连接到一个运行中的move_group节点的运行实例
    //move_group_interface::MoveGroup group("arm");
    moveit::planning_interface::MoveGroupInterface group("arm");
    // 指定一个随机目标位置
    group.setRandomTarget();
    // 开始规划然后移动到指定目标位置
    group.move();
    ros::waitForShutdown();
}
```

为了编译该源码，我们需要将下面几行代码添加到CMakeLists.txt文件中。你也可以从已有的软件包中获取完整的CMakeLists.txt文件：

```
add_executable(test_random_node src/test_random.cpp)
add_dependencies(test_random_node seven_dof_arm_test_generate_messages_cpp)
target_link_libraries(test_random_node
${catkin_LIBRARIES})
```

我们用`catkin_make`编译该软件包。首先检查`test_random.cpp`是否可以正常编译，如果源码编译正常，我们就可以进行测试了。

下面的命令将启动`RViz`，同时将加载带有运动规划插件的7-DOF机械臂：

```
$ rosrun seven_dof_arm_config demo.launch
```

在`RViz`中移动末端执行器以检查是否一切正常。

使用下面的命令运行该C++节点以规划出一个随机位置：

```
$ rosrun seven_dof_arm_test test_random_node
```

图12-1显示的是`RViz`的输出。机械臂将随机选择一个IK有效的位姿作为目标点，并从当前位置到目标位置进行运动规划。

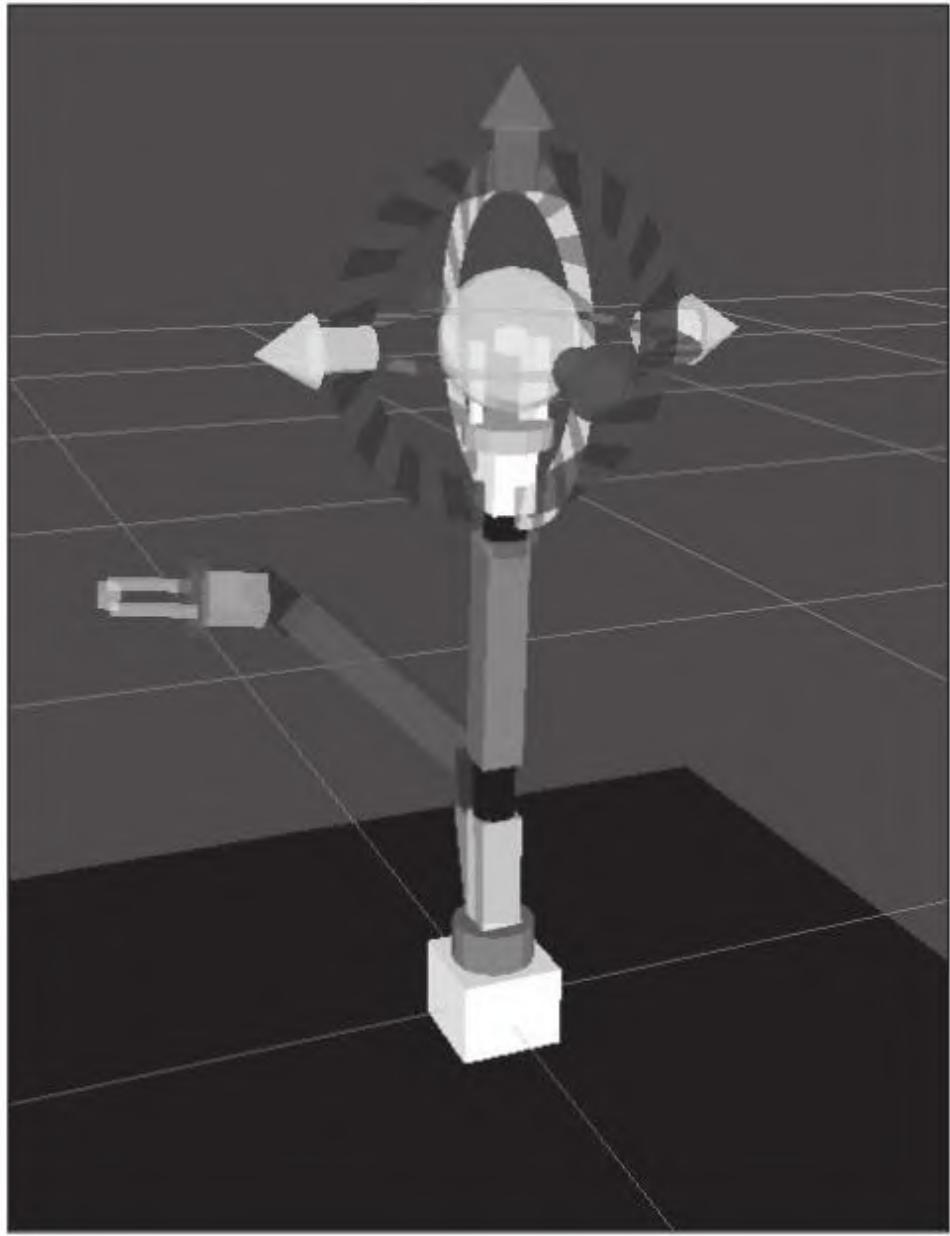


图12-1 用move_group API进行随机运动规划

12.1.2 使用MoveIt! C++ API规划自定义路径

在前面的例子中我们学习了随机运动规划。在本节中，我们将学习如何控制机器人末端执行器移动到指定的目标位置。下面的示例 `test_custom.cpp` 将实现这个目标：

```
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

int main(int argc, char **argv)
{
    // ROS 初始化
    ros::init(argc, argv, "move_group_interface_tutorial");
    ros::NodeHandle node_handle;
    ros::AsyncSpinner spinner(1);
    spinner.start();
    sleep(2.0);
    // Move group 配置
    moveit::planning_interface::MoveGroupInterface group("arm");
    moveit::planning_interface::PlanningSceneInterface
    planning_scene_interface;
    ros::Publisher display_publisher =
    node_handle.advertise<moveit_msgs::DisplayTrajectory>("/move_group/display_
planned_path", 1, true);
```

```
moveit_msgs::DisplayTrajectory display_trajectory;
ROS_INFO("Reference frame: %s", group.getEndEffectorLink().c_str());
// 配置目标位姿
geometry_msgs::Pose target_pose1;
target_pose1.orientation.w = 0.726282;
target_pose1.orientation.x= 4.04423e-07;
target_pose1.orientation.y = -0.687396;
target_pose1.orientation.z = 4.81813e-07;
target_pose1.position.x = 0.0261186;
target_pose1.position.y = 4.50972e-07;
target_pose1.position.z = 0.573659;
group.setPoseTarget(target_pose1);
// 动作规划
moveit::planning_interface::MoveGroupInterface::Plan my_plan;
moveit::planning_interface::MoveItErrorCode success =
group.plan(my_plan);
ROS_INFO("Visualizing plan 1 (pose goal) %s", success.val ? "" :"FAILED");
// 等待一会以便 Rviz 显示路径
sleep(5.0);
ros::shutdown();
}
```

下面多出的几行代码是为了编译源码：

```
add_executable(test_custom_node src/test_custom.cpp)
add_dependencies(test_custom_node seven_dof_arm_test_generate_messages_cpp)
target_link_libraries(test_custom_node
${catkin_LIBRARIES})
```

下面是执行自定义节点的命令：

```
$ rosrun seven_dof_arm_test test_custom_node
```

图12-2所示截屏显示了test_custom_node节点的执行结果。

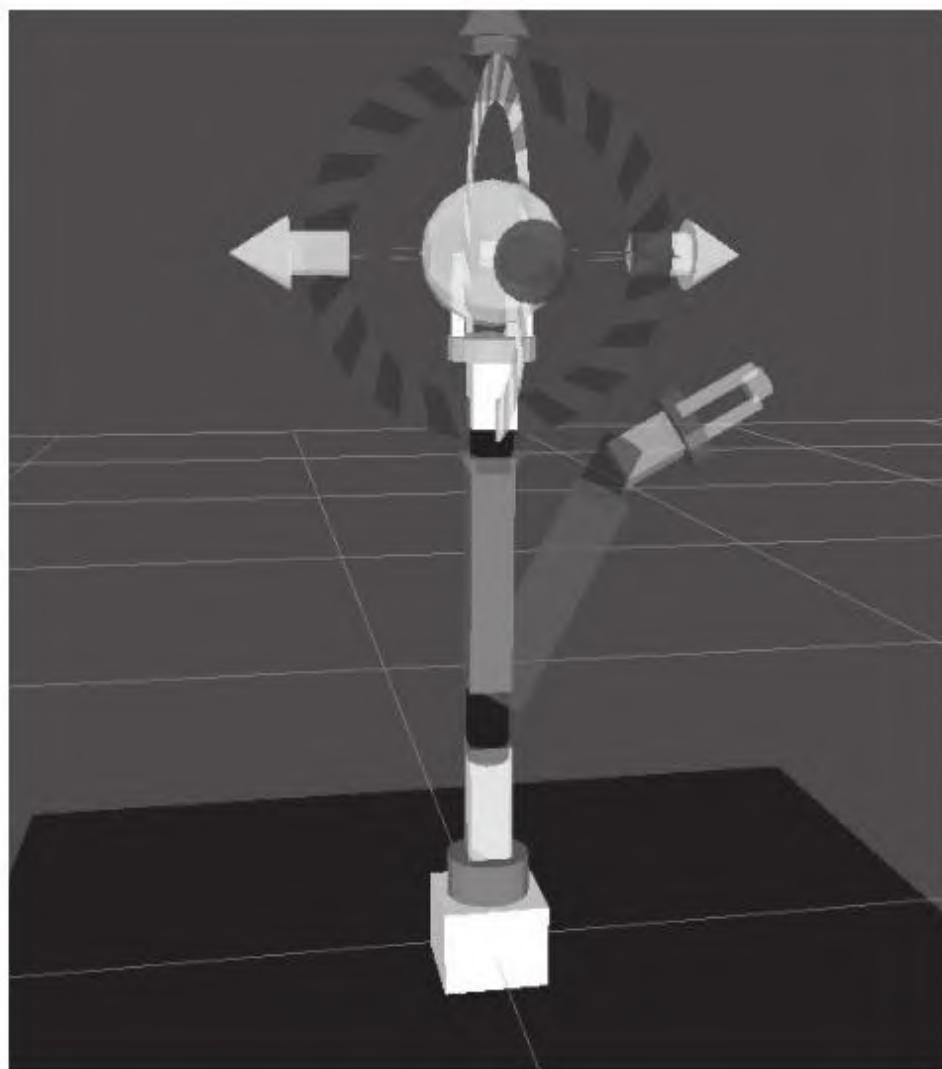


图12-2 用MoveIt! C++ API进行自定义动作规划

12.2 使用MoveIt!进行机械臂的碰撞检测

在运动规划和IK求解算法中，碰撞检测及避障是MoveIt!同时进行的重要任务之一。MoveIt!可以利用内置的碰撞检测库（Flexible Collision Library, FCL）

（http://gamma.cs.unc.edu/FCL/fcl_docs/webpage/generated/index.html）处理自碰撞以及与环境的碰撞检测。FCL是一个实现各种碰撞检测和避障算法的开源项目。MoveIt!利用FCL的强大功能，用`collision_detection::CollisionWorld`类处理规划场景内的碰撞。MoveIt!的碰撞检测包括各种对象，如网格、基本形状（立方体和圆柱体），及OctoMap。其中，OctoMap（<http://octomap.github.io/>）库实现了一个3D占据栅格（称为八叉树），它包含相关环境中障碍物的概率信息。MoveIt!软件包可以用3D点云信息构建OctoMap，也可以直接将OctoMap提供给FCL进行碰撞检测。

与运动规划一样，碰撞检测也是计算密集型的。我们可以用允许碰撞矩阵（Allowed Collision Matrix, ACM）的参数微调两个物体（如机器人连杆和环境）之间的碰撞检测。如果在ACM中将两个连杆之间的碰撞值设置为1，则不会进行任何碰撞检测。我们可以用该矩阵设置距离较远的连杆，这样我们就可以通过优化这个矩阵来优化碰撞检测过程。

12.2.1 向MoveIt!添加碰撞对象

我们可以向MoveIt!的规划场景中添加一个碰撞对象，这样我们就能看到运动规划的工作原理。对于添加碰撞对象，我们可以使用可直接导入MoveIt!接口的网格文件，当然也可通过MoveIt! API编写的ROS节点来添加。

首先我们讨论如何用ROS节点添加碰撞对象：

1) 在seven_dof_arm_test/src文件夹中的节点源码文件add_collision_object.cpp中，我们将启动一个ROS节点并创建一个moveit::planning_interface::PlanningSceneInterface对象，该对象可以访问MoveIt!中的规划场景并可对当前场景执行任何操作。我们将增加5秒的休眠时间以等待planningSceneInterface对象实例化：

```
moveit::planning_interface::PlanningSceneInterface  
current_scene;  
sleep(5.0);
```

2) 下一步，我们需要创建一个碰撞对象消息

moveit_msgs::CollisionObject的实例。该消息将被发送到当前的规划场景中。在这里，我们为一个圆柱体创建一个碰撞对象消息，这个消息以seven_dof_arm_cylinder形式给出。当将该对象添加到规划场景中时，该对象的名称即是其ID：

```
moveit_msgs::CollisionObject cylinder;  
cylinder.id = "seven_dof_arm_cylinder";
```

3) 创建了碰撞对象消息后，我们必须定义一种其他类型的消息shape_msgs::SolidPrimitive，该消息用于定义我们正在使用的基本形状及这些形状的属性。在该示例中，我们正在创建一个圆柱体对象，如下面的代码所示。我们必须定义该对象的类型，大小参数、圆柱体的宽度和高度：

```
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.CYLINDER;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.6;
primitive.dimensions[1] = 0.2;
primitive.dimensions[2] = 0.2;
```

4) 创建了物体模型的消息后，我们必须创建一个`geometry_msgs::Pose`消息来定义该对象的姿态。我们应该定义一个尽可能接近机器人的姿态。在规划场景中创建对象后，我们也可以改变其姿态：

```
geometry_msgs::Pose pose;
pose.orientation.w = 1.0;
pose.position.x = 0.0;
pose.position.y = -0.4;
pose.position.z = -0.4;
```

5) 定义好碰撞对象的姿态后，我们需要添加定义过的基本形状对象及其相对于碰撞检测对象的姿态。我们需要执行的操作是添加规划场景：

```
cylinder.primitives.push_back(primitive);
cylinder.primitive_poses.push_back(pose);
cylinder.operation = cylinder.ADD;
```

6) 在这一步中，创建一个名为`collision_objects`的向量(`vector`)，它的类型是`moveit_msgs::CollisionObject`，将碰撞对象插入到该向量中：

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

7) 我们将用下面的代码将碰撞对象向量添加到当前规划场景中。PlanningSceneInterface类中的addCollisionObjects()将对象添加到规划场景中：

```
current_scene.addCollisionObjects(collision_objects);
```

8) 下面这些是在CMakeLists.txt中用于编译的代码：

```
add_executable(add_collision_object
src/add_collision_object.cpp)
add_dependencies(add_collision_object
seven_dof_arm_test_generate_messages_cpp)
target_link_libraries(add_collision_object
${catkin_LIBRARIES})
```

让我们看看这个节点在RViz中如何使用MoveIt!运动规划插件：

- 运行seven_dof_arm_config软件包中的demo.launch文件以测试该节点：

```
$ rosrun seven_dof_arm_config demo.launch
```

- 然后添加下面的碰撞对象：

```
$ rosrun seven_dof_arm_test add_collision_object
```

当我们运行add_collision_object节点时，会显示出一个圆柱体，我们可以像图12-3那样移动碰撞对象。将碰撞对象成功添加到规划场景中后，它将在Scene Objects选项卡中列出。我们可以单击该对象并修改其姿态。针对机器人的任意连杆，我们都可以为其添加新的碰撞模型。这里有一个Scale选项可以缩放碰撞模型：

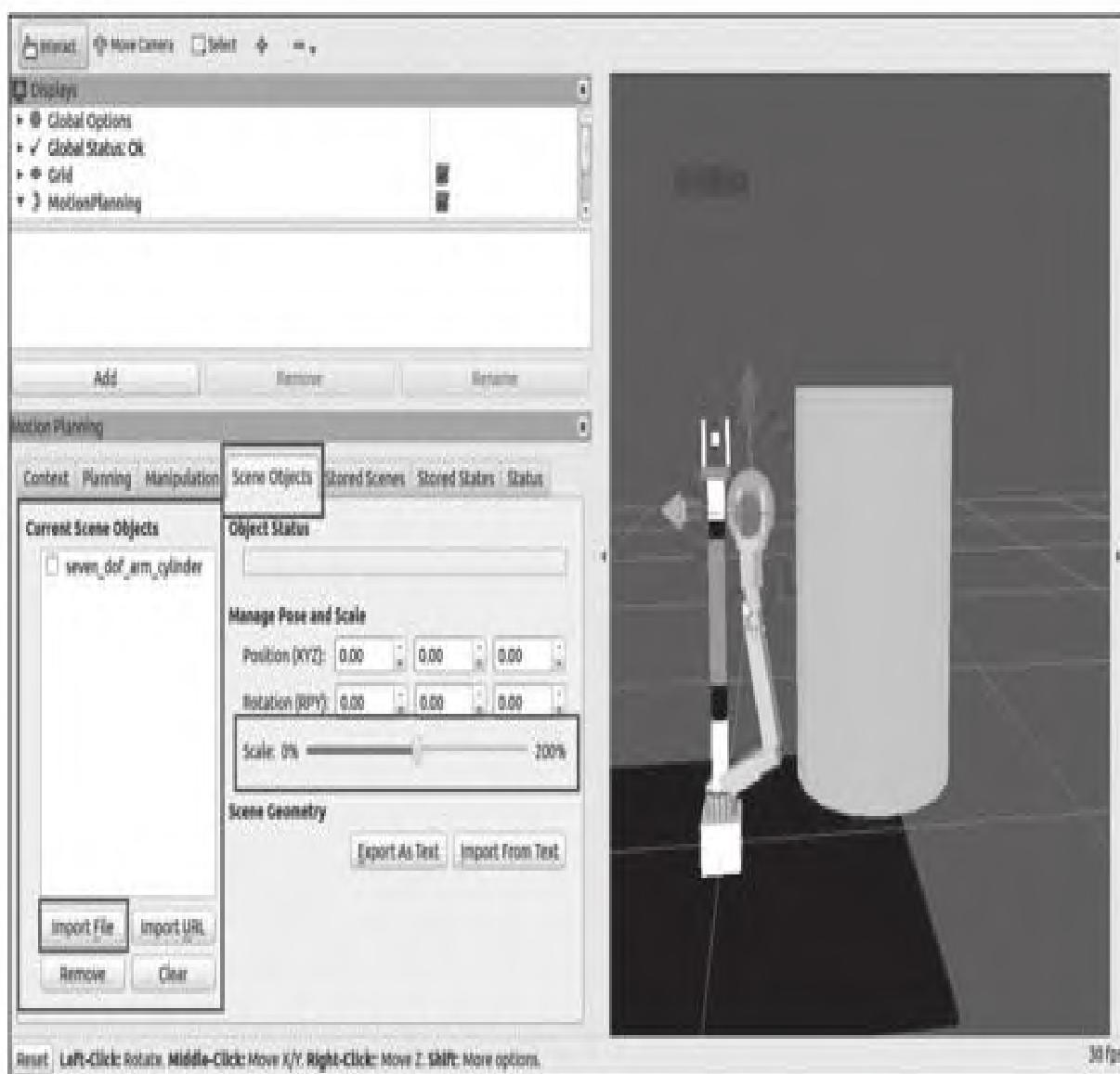


图12-3 用MoveIt! C++ API将碰撞对象添加到RViz

RViz运动规划插件也提供了可以向规划场景中导入3D网格模型的选项卡。点击Import File按钮可以进行模型文件的导入。图12-4展示了一个导入的立方体网格DAE文件，该文件是与规划场景中的圆柱一起导入的。我们可以用Scale滑动条放大碰撞物体，也可以用Manager Pose选项设置想要的姿态。当我们把机械臂末端执行器移动到这些碰撞物体上时，MoveIt!将检测到碰撞。MoveIt!不仅可以检测到自碰撞还可以检测到与环境之间碰撞。

图12-4就是一张与环境碰撞的截图。

当机械臂接触到物体时，碰撞的连杆会变红。在自碰撞中，碰撞的连杆也会变红。当然，我们可以在运动规划插件的设置中修改碰撞的颜色。

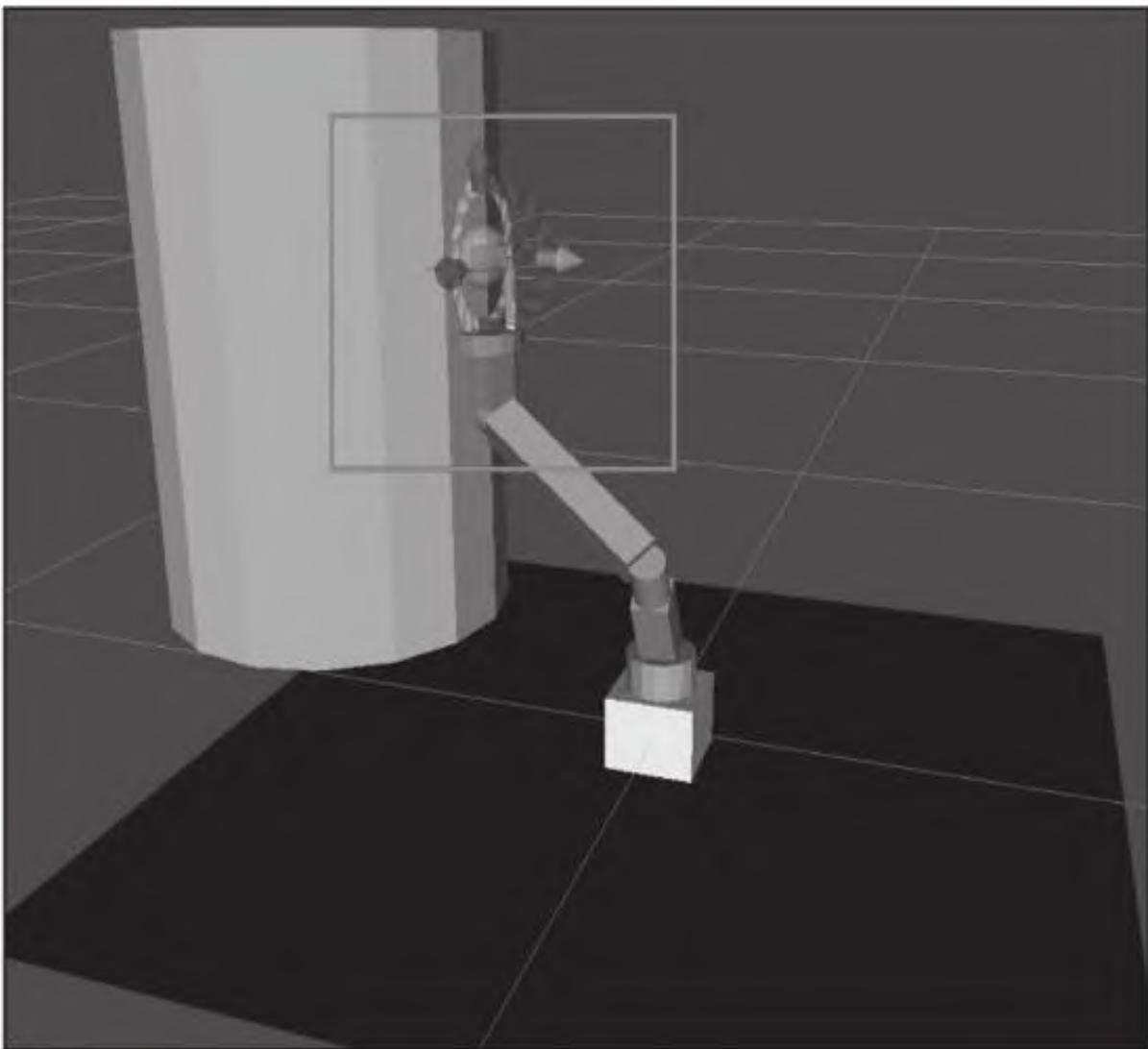


图12-4 可视化碰撞连杆

12.2.2 从规划场景中移除碰撞对象

从规划场景中移除碰撞物体是很容易的。我们必须创建一个`moveit::planning_interface::PlanningSceneInterface`的对象，就像我们在前面的例子中所做的那样，然后再加一些延时：

```
moveit::planning_interface::PlanningSceneInterface current_scene;  
sleep(5.0);
```

接下来，我们需要创建一个包含碰撞物体ID的字符串向量（vector）。在这里，碰撞对象ID是`seven_dof_arm_cylinder`。将该字符串赋值给这个向量后，我们将调用`removeCollisionObjects(object_ids)`，它可以从规划场景中将碰撞物体移除：

```
std::vector<std::string> object_ids;  
object_ids.push_back("seven_dof_arm_cylinder");  
current_scene.removeCollisionObjects(object_ids);
```

这里的代码位于
`seven_dof_arm_test/src/remove_collision_object.cpp`。

12.2.3 向机器人连杆上添加一个碰撞对象

了解了如何从规划场景中插入和移除对象后，现在我们讨论如何在机器身上附加或分离这些物体。ROS MoveIt!的这一重要特性使我们能够操作这些对象。实际上，将物体附着到机器人上后，被抓取物体也被添加到避障空间内。通过这种方式，机器人可以在工作空间中自由移动、避障、抓取物体。我们将要讨论的代码位于 seven_dof_arm_test/src/attach_detach_objs.cpp 文件中。创建了 moveit::planning_interface::PlanningSceneInterface 对象后，如前面的示例所示，我们必须初始化一个 moveit_msgs::AttachedCollisionObject 对象，完善相关信息，如哪些场景对象将附着到机器人的指定连杆上：

```
moveit_msgs::AttachedCollisionObject attached_object;
attached_object.link_name = "grasping_frame";
attached_object.object = grasping_object;
current_scene.applyAttachedCollisionObject( attached_object );
```

在该示例中，附着到机器人连杆的 `grasping_object` 是在 `add_collision_object.cpp` 示例中已使用的那个。当对象成功附着到机器人上时，MoveIt! 中显示的颜色将从绿色变为紫色，并随着机器人的运动一起移动。要从机器人本体中分离一个对象，我们应该调用目的对象的 `applyAttachedCollisionObject` 函数，并将其操作从 ADD 改为 REMOVE：

```
grasping_object.operation = grasping_object.REMOVE;
attached_object.link_name = "grasping_frame";
attached_object.object = grasping_object;
```

12.2.4 使用MoveIt! API检查自碰撞

我们已经了解了如何在RViz中检测碰撞，但是如果我们在ROS节点中获取碰撞信息该如何做呢？在本节，我们将讨论如何在ROS代码中获取机器人的碰撞信息。该示例可以检查自碰撞和环境碰撞，还可以得知哪些连杆发生了碰撞。示例源码文件名为check_collision.cpp，位于seven_dof_arm_test/src文件夹中。该代码是PR2 MoveIt!机器人教程中碰撞检测示例代码的修改版本。在此代码中，下面的代码段是将机器人的运动模型加载到规划场景中：

```
robot_model_loader::RobotModelLoader  
robot_model_loader("robot_description");  
robot_model::RobotModelPtr kinematic_model = robot_model_loader.getModel();  
planning_scene::PlanningScene planning_scene(kinematic_model);
```

要在机器人的当前状态下测试自碰撞，我们可以创建
`collision_detection::CollisionRequest`和
`collision_detection::CollisionResult`这两个类的实例，实例名称
分别为`collision_request`和`collision_result`。创建好这些对象后，
将其传递给MoveIt!碰撞检测函数
`planning_scene.checkSelfCollision()`，它可以在`collision_result`
对象中给出碰撞结果，我们可以打印详细信息，如下代码所示：

```
planning_scene.checkSelfCollision(collision_request, collision_result);  
ROS_INFO_STREAM("1. Self collision Test: "<< (collision_result.collision ?  
"in" : "not in")  
<< " self collision");
```

如果想测试某个特定组中的碰撞，我们可以通过上面提到的
`group_name`来实现，如下面的代码所示。这里的`group_name`是`arm`：

```
collision_request.group_name = "arm";
current_state.setToRandomPositions();

//Previous results should be cleared
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("3. Self collision Test(In a group): "<<
(collision_result.collision ? "in" : "not in"));


```

想要执行完整的碰撞检测，我们必须使用函数 `planning_scene.checkCollision()`。在该函数中需要指定当前机器人状态和ACM矩阵。

以下就是用该函数进行完整碰撞检测的代码片段：

```
collision_detection::AllowedCollisionMatrix acm =
planning_scene.getAllowedCollisionMatrix();
robot_state::RobotState copied_state = planning_scene.getCurrentState();
planning_scene.checkCollision(collision_request, collision_result,
copied_state, acm);
ROS_INFO_STREAM("6. Full collision Test: "<< (collision_result.collision ?
"in" : "not in")
<< " collision");
```

我们可以用下面的命令启动动作规划示例：

```
$ rosrun seven_dof_arm_config demo.launch
```

运行碰撞检测节点：

```
$ rosrun seven_dof_arm_test check_collision
```

你将得到一份报告，如图12-5所示。机器人现在没有碰撞，如果有碰撞的话，它会发送碰撞报告信息。

```
[ INFO] [1512837566,744018279]: 1. Self collision Test: not in self collision
[ INFO] [1512837566,744073739]: 2. Self collision Test(Change the state): in
[ INFO] [1512837566,744108096]: 3. Self collision Test(In a group): in
[ INFO] [1512837566,744122925]: 4. Collision points valid
[ INFO] [1512837566,744167799]: 5. Self collision Test: in self collision
[ INFO] [1512837566,744179527]: 6 . Contact between: elbow_pitch_link and wrist_pitch_link
[ INFO] [1512837566,744227589]: 6. Self collision Test after modified ACM: not in self collision
[ INFO] [1512837566,744262790]: 6. Full collision Test: not in collision
```

图12-5 碰撞检测信息

12.3 使用MoveIt!和Gazebo处理视觉

到目前为止，在MoveIt!中，我们只使用过一个机械臂。在本节，我们将学习如何将3D视觉传感器数据与MoveIt!连接。该传感器可以用Gazebo进行仿真，或者也可以用openni_launch软件包直接连接RGB-D传感器（如Kinect或Xtion Pro）。这里，我们将用Gazebo来仿真。我们将在MoveIt!中添加传感器，用于视觉辅助执行拾取和放置操作。我们将在Gazebo中创建一张桌子和一个待抓取物体，用于拾取和放置操作。我们将添加两个自定义模型，分别是Grasp_Object和Grasp_Table。示例模型存放在seven_dof_arm_test软件包中的模型文件夹下，需要将其复制到~/.gazebo/models文件夹下，以便可以从Gazebo访问这些模型。下面的命令将在Gazebo中启动机器人的机械臂和华硕Xtion Pro的仿真：

```
$ rosrun seven_dof_arm_gazebo seven_dof_arm_grasping.launch
```

该命令将打开带有机械臂关节控制器的Gazebo和用于3D视觉传感器的Gazebo插件。我们可以添加一张桌子和一个待抓取物体到仿真器中，如图12-6所示，我们只需简单地点击并将它们拖动到工作区即可。我们可以创建任何类型的桌子和对象。图12-6中显示的对象仅用于演示。我们可以编辑模型的SDF文件，这样就可以改变模型的大小和形状。

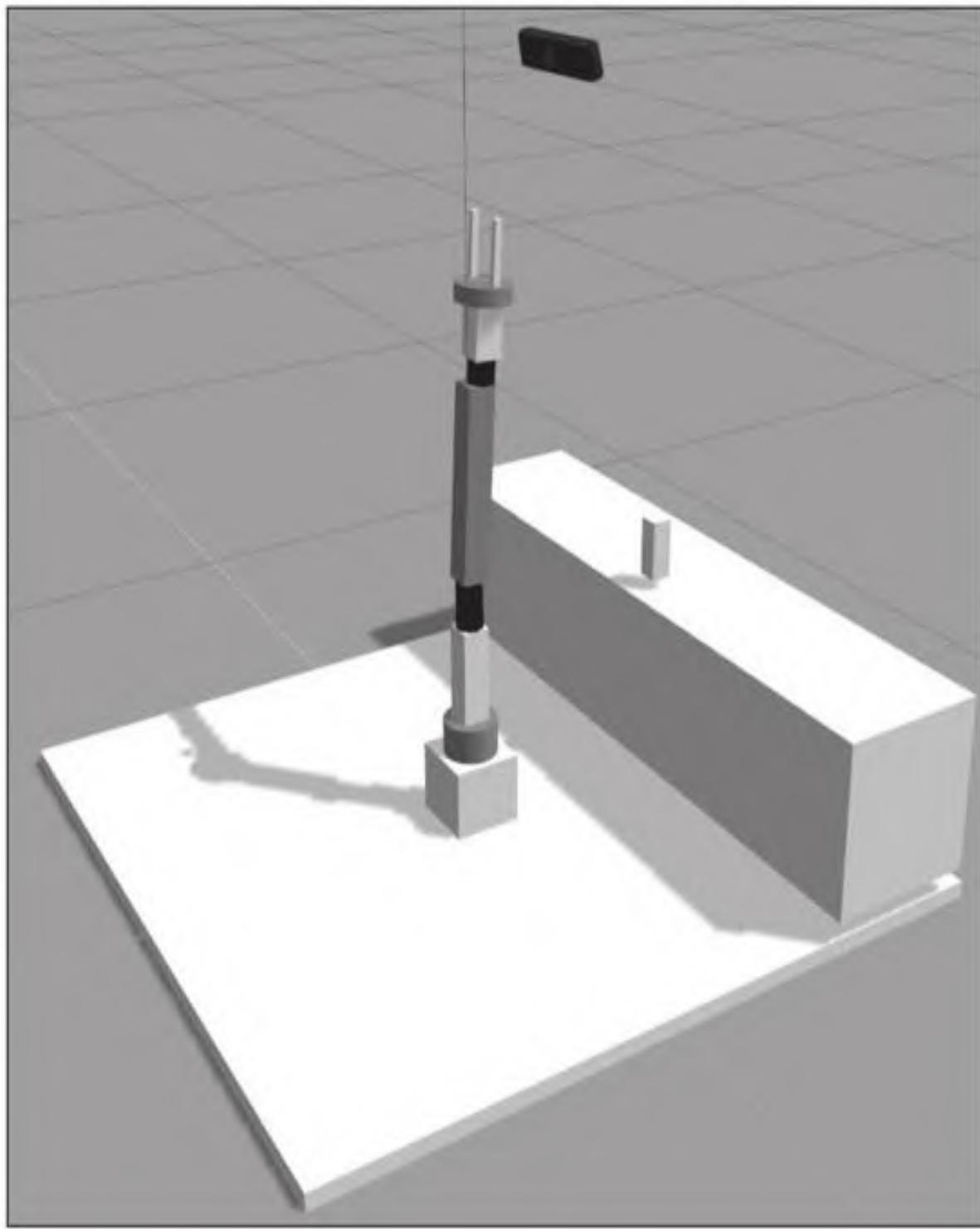


图12-6 在Gazebo中的机械臂和待抓取对象

启动仿真环境后，查看生成的话题：

```
$ rostopic list
```

确保我们可以获取到RGB-D相机话题，如图12-7显示的代码段所示。

我们可以用下面的命令在RViz中查看点云：

```
$ rosrun rviz rviz -f base_link
```

图12-8显示的是生成输出的点云信息。

```
/rgbd_camera/depth/camera_info
/rgbd_camera/depth/image_raw
/rgbd_camera/depth/points
/rgbd_camera/ir/camera_info
/rgbd_camera/ir/image_raw
/rgbd_camera/ir/image_raw/compressed
/rgbd_camera/ir/image_raw/compressed/parameter_descriptions
/rgbd_camera/ir/image_raw/compressed/parameter_updates
/rgbd_camera/ir/image_raw/compressedDepth
/rgbd_camera/ir/image_raw/compressedDepth/parameter_descriptions
/rgbd_camera/ir/image_raw/compressedDepth/parameter_updates
/rgbd_camera/ir/image_raw/theora
/rgbd_camera/ir/image_raw/theora/parameter_descriptions
/rgbd_camera/ir/image_raw/theora/parameter_updates
/rgbd_camera/parameter_descriptions
/rgbd_camera/parameter_updates
/rgbd_camera/rgb/camera_info
/rgbd_camera/rgb/image_raw
/rgbd_camera/rgb/image_raw/compressed
/rgbd_camera/rgb/image_raw/compressed/parameter_descriptions
/rgbd_camera/rgb/image_raw/compressed/parameter_updates
/rgbd_camera/rgb/image_raw/compressedDepth
/rgbd_camera/rgb/image_raw/compressedDepth/parameter_descriptions
/rgbd_camera/rgb/image_raw/compressedDepth/parameter_updates
/rgbd_camera/rgb/image_raw/theora
/rgbd_camera/rgb/image_raw/theora/parameter_descriptions
/rgbd_camera/rgb/image_raw/theora/parameter_updates
/rgbd_camera/rgb/points
```

图12-7 列出RGB-D传感器话题

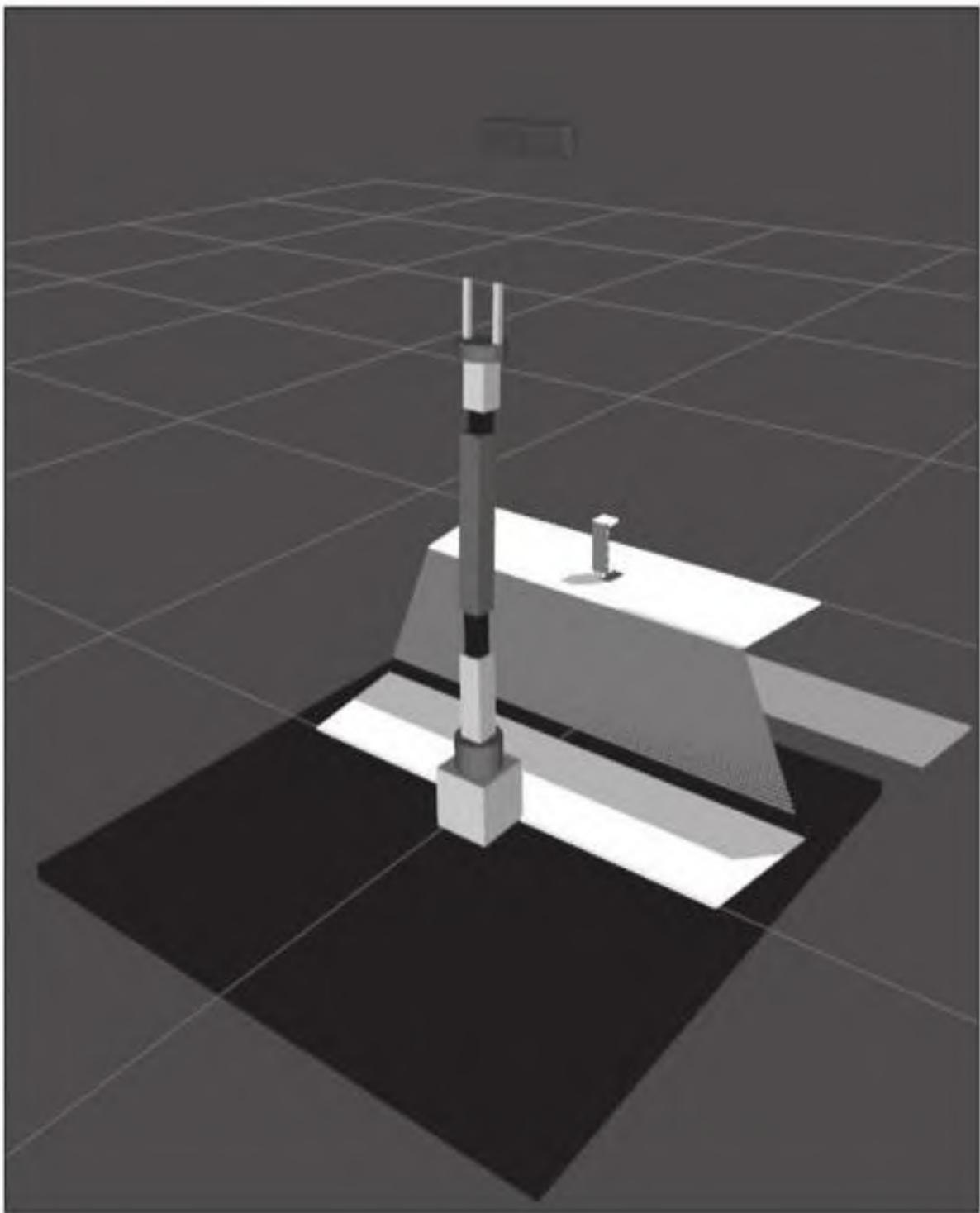


图12-8 在RViz中可视化点云数据

确认了来自Gazebo插件的点云数据后，我们必须将一些文件添加到这个机械臂的MoveIt!配置软件包中，即seven_dof_arm_config，用

于将点云数据从Gazebo复制到MoveIt!规划场景中。

机器人的环境可以用八叉树来表示（<https://en.wikipedia.org/wiki/Octree>），可以用Octo-Map库来构建，我们已在上一节中学过。OctoMap作为一个MoveIt!的插件，被称为Occupancy Map Updater插件，它可以从不同类型的传感器输入来更新八叉树，例如从点云和来自3D视觉传感器的深度图像。目前，有下面这些用于处理3D数据的插件：

- PointCloudOccupancymap Updater:该插件能以点云的形式作为输入(sensor_msgs/PointCloud2)。
- DepthImageOccupancymapUpdater:该插件能以深度图像的形式作为输入(sensor_msgs/Image)。

第一步是为这些插件编写配置文件。这些配置文件包含机器人用到的插件及其参数属性。用到第一个插件的配置文件可以在seven_dof_arm_config/config文件夹中找到，文件名为sensor_kinect_pointcloud.yaml。

该文件的定义如下：

```
sensors:  
- sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater  
  point_cloud_topic: /rgbd_camera/depth/points  
  max_range: 10  
  padding_offset: 0.01  
  padding_scale: 1.0  
  point_subsample: 1  
  filtered_cloud_topic: output_cloud
```

参数的解释如下：

- sensor_plugin: 该参数指定机器人使用的插件名称。

下面就是sensor_plugin的各参数：

- point_cloud_topic: 该插件将监听此话题来获取点云数据。
- max_range: 以米为单位的距离限值，大于该范围的点将不被处理。
- padding_offset: 当滤波的点云包含机器人连杆时，该值将考虑机器人连杆及其附加的对象。
- padding_scale: 自滤波时也会考虑该值。
- point_subsample: 如果刷新过程很慢，则可以对点云进行二次采样。如果该值被设置为大于1，我们将忽略该点云不进行处理。
- filtered_cloud_topic: 这是最终滤波后的点云话题。我们可以通过该话题来得到处理过的点云。它主要在调试的时候使用。

如果使用DepthImageOctomapUpdater插件，我们需要另外一个不同的配置文件。在示例机器人中虽然没有使用该插件，但是我们仍可以看看它的用法和属性：

```
sensors:  
- sensor_plugin: occupancy_map_monitor/DepthImageOctomapUpdater  
  image_topic: /head_mount_kinect/depth_registered/image_raw  
  queue_size: 5  
  near_clipping_plane_distance: 0.3  
  far_clipping_plane_distance: 5.0  
  skip_vertical_pixels: 1  
  skip_horizontal_pixels: 1  
  shadow_threshold: 0.2  
  padding_scale: 4.0  
  padding_offset: 0.03  
  filtered_cloud_topic: output_cloud
```

参数的解释如下：

- sensor_plugin：该参数指定机器人使用的插件名称。

下面就是sensor_plugin的各参数：

- image_topic：传输图像的话题。
- queue_size：这是订阅的深度图像话题的消息队列大小。
- near_clipping_plane_distance：这是距传感器的最小有效距离。
- far_clipping_plane_distance：这是距传感器的最大有效距离。
- skip_vertical_pixels：这是从图像的上部和下部忽略的像素数。如果我们设置为5，那么将从图像的上部和下部跳过5列。
- skip_horizontal_pixels：在水平方向上跳过的像素数。
- shadow_threshold：在某些情形下，机器人连杆的下方可能会出现点云。这是由填充造成的。shadow_threshold参数删除距离大于该阈值的点云。

讨论了OctoMap刷新插件及其属性后，我们来看看启动该插件及其参数的启动文件。我们需要创建的第一个文件位于seven_dof_arm_config/launch文件夹中，名为seven_dof_arm_moveit_sensor_manager.launch。下面就是该文件的定义。启动文件基本上加载了插件的参数：

```
<launch>
  <rosparam command="load" file="$(find
  seven_dof_arm_config)/config/sensors_kinect_pointcloud.yaml" />
</launch>
```

我们需要编辑的下一个文件是sensor_manager.launch，它位于launch文件夹中。该文件的定义如下：

```
<launch>
  <!-- This file makes it easy to include the settings for sensor managers
  -->

  <!-- Params for the octomap monitor -->
  <!-- <param name="octomap_frame" type="string" value="some frame in
which the robot moves" /> -->
  <param name="octomap_resolution" type="double" value="0.015" />
  <param name="max_range" type="double" value="5.0" />

  <!-- Load the robot specific sensor manager; this sets the
moveit_sensor_manager ROS parameter -->

  <arg name="moveit_sensor_manager" default="seven_dof_arm" />
  <include file="$(find seven_dof_arm_config)/launch/${arg
moveit_sensor_manager}_moveit_sensor_manager.launch.xml" />

</launch>
```

下面这行代码被注释掉了，因为它仅适用于机器人是动态的情形。在我们的示例中，机器人是静态的。如果它固定于一个移动机器人上，我们就可以设置该机器人的坐标系为`odom`或`odom_combined`。

```
<param name="octomap_frame" type="string" value="some frame in which the
robot moves" />
```

下面的参数是OctoMap的分辨率，用于在RViz中进行可视化显示（单位是米）。超出距离限值（`max_range`）的值将被丢弃：

```
<param name="octomap_resolution" type="double" value="0.015" />
<param name="max_range" type="double" value="5.0" />
```

现在接口已经配置完成，我们可以用下面的命令测试MoveIt!的接口。用下面的命令启动Gazebo感知周围环境，然后添加所需的桌子和待抓取对象模型：

```
$ rosrun seven_dof_arm_gazebo seven_dof_arm_grasping.launch
```

启动有传感器支持的MoveIt!规划器：

```
$ rosrun seven_dof_arm_config moveit_planning_execution.launch
```

现在RViz有了传感器支持。我们就可以在机器人前面看到OctoMap了，如图12-9所示。

使用MoveIt!操作对象

对物体进行操作是机械臂的主要用途之一。在机器人的工作空间范围内拾取物体并将其放置到不同位置，这种能力在工业和研究应用中非常有用。拾取的过程也被称为抓取（grasping），这是一个很复杂的任务，因为需要许多约束才能以适当的方式来抓取对象。人类可以利用他们的智能来处理抓取操作，但是机器人需要一定的规则来处理抓取操作。约束之一是力量的控制。末端执行器应该能够调整抓取物体的抓取力，同时要求抓住物体时不能对物体造成任何形变。另外，需要以最佳抓握姿态来抓取物体，该抓握姿态应该考虑其形状和姿态。MoveIt!没有提供任何内置功能以找到最佳接近或抓取姿态来抓取物体。出于这个原因，在本书中，我们将首先讨论如何使用一种已知的抓握姿态来拾取和放置物体。后面，我们将展示抓握姿态检测（Grasping Pose Detector，GPD）软件包，一个能够根据点云检测6-DOF抓握姿态的ROS软件包。

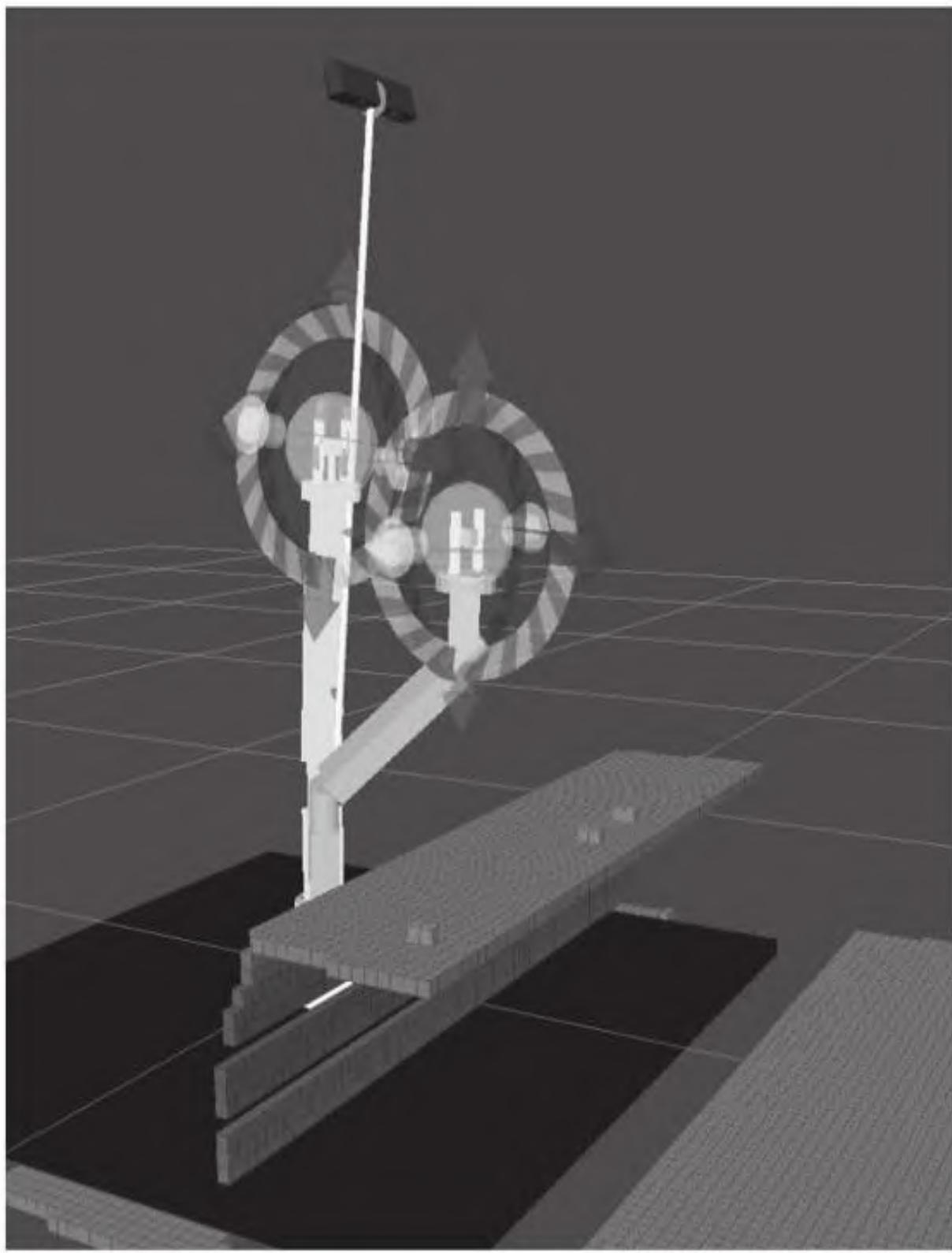


图12-9 在RViz中可视化octomap

12.4 使用MoveIt!执行拾取和放置任务

我们可以用各种方式来进行拾取和放置（Pick-and-place）。其中一种方式是使用预定义的关节序列值。在这种方式下，我们将待抓取对象放置在已知位置上，然后提供关节值或正向运动学来将机器人移动到该位置。另外一种方式是在没有任何视觉信息引导的情况下使用逆向运动学进行。在这种方式中，通过求解IK来命令机器人移动到相对于其基坐标系的笛卡儿位置上。通过这种方式，机器人可以到达该位置并拾取物体。还有一种方式是使用外部传感器，例如通过视觉传感器计算拾取和放置的位置。在这种方式中，视觉传感器用于识别物体的位置，然后对该物体进行运动学求解以使机械臂到达其位置。当然，使用视觉传感器需要开发出具备鲁棒性的算法来执行对象的识别和跟踪，并且能够计算抓取该对象的最佳抓握姿态。但在本节中，我们想要演示一个拾取和放置运动序列，通过定义接近和抓取位置来拾取对象，然后将其放置在其工作空间内的另一个位置上。我们可以和Gazebo一起使用这个例子，或者仅使用MoveIt!的demo界面。该示例的完整源码位于seven_dof_arm_test/src/pic_place.cpp文件中。正如我们看到的，首先初始化规划场景：

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

ros::NodeHandle node_handle;
ros::AsyncSpinner spinner(1);
spinner.start();
moveit::planning_interface::MoveGroupInterface group("arm");
moveit::planning_interface::PlanningSceneInterface
planning_scene_interface;
sleep(2);
moveit::planning_interface::MoveGroupInterface::Plan my_plan;
const robot_state::JointModelGroup *joint_model_group =
group.getCurrentState()->getJointModelGroup("arm");
```

然后，我们必须创建机器人的工作环境，将桌子和待抓取对象放在该场景中：

```
moveit::planning_interface::PlanningSceneInterface current_scene;
geometry_msgs::Pose pose;
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.03;
primitive.dimensions[1] = 0.03;
primitive.dimensions[2] = 0.08;
moveit_msgs::CollisionObject grasping_object;
grasping_object.id = "grasping_object";
pose.orientation.w = 1.0;
pose.position.y = 0.0;
pose.position.x = 0.33;
pose.position.z = 0.35;
grasping_object.primitives.push_back(primitive);
grasping_object.primitive_poses.push_back(pose);
grasping_object.operation = grasping_object.ADD;
grasping_object.header.frame_id = "base_link";
primitive.dimensions[0] = 0.3;
primitive.dimensions[1] = 0.5;
primitive.dimensions[2] = 0.32;
moveit_msgs::CollisionObject grasping_table;
grasping_table.id = "grasping_table";
pose.orientation.w = 1.0;
pose.position.y = 0.0;
pose.position.x = 0.46;
pose.position.z = 0.15;
grasping_table.primitives.push_back(primitive);
grasping_table.primitive_poses.push_back(pose);
grasping_table.operation = grasping_table.ADD;
grasping_table.header.frame_id = "base_link";
std::vector<moveit_msgs::CollisionObject> collision_objects;
collision_objects.push_back(grasping_object);
collision_objects.push_back(grasping_table);
//--- 向场景中发布对象
current_scene.addCollisionObjects(collision_objects);
```

现在规划场景已经配置好了，我们就可以请求机器人向工作空间中预先配置好的位置运动，这样就可以让末端执行器靠近物体并执行拾取操作：

```
//--- 接近
geometry_msgs::Pose target_pose;

target_pose.orientation.x = 0;
target_pose.orientation.y = 0;
target_pose.orientation.z = 0;
target_pose.orientation.w = 1;
target_pose.position.y = 0.0;
target_pose.position.x = 0.32;
target_pose.position.z = 0.35;
group.setPoseTarget(target_pose);
group.move();

//--- 抓取
target_pose.position.y = 0.0;
target_pose.position.x = 0.34;
target_pose.position.z = 0.35;
group.setPoseTarget(target_pose);
group.move();
```

如果抓取成功了，为了将被抓取的物体放置于工作空间的另一个位置，我们可以将该物体粘附到机器人的末端执行器上：

```
//--- 附加物体到机器人上
moveit_msgs::AttachedCollisionObject attached_object;
attached_object.link_name = "grasping_frame";
attached_object.object = grasping_object;
current_scene.applyAttachedCollisionObject( attached_object );
//--- 放置
target_pose.position.y = -0.1;
target_pose.position.x = 0.34;
target_pose.position.z = 0.4;
group.setPoseTarget(target_pose);
group.move();
//---
target_pose.orientation.x = -1;
target_pose.orientation.y = 0;
target_pose.orientation.z = 0;
target_pose.orientation.w = 0;
target_pose.position.y = -0.1;
target_pose.position.x = 0.34;
target_pose.position.z = 0.35;
group.setPoseTarget(target_pose);
group.move();
```

最后，我们必须将物体从机器人的夹爪中移除：

```
grasping_object.operation = grasping_object.REMOVE;
attached_object.link_name = "grasping_frame";
attached_object.object = grasping_object;
current_scene.applyAttachedCollisionObject( attached_object );
```

可以启动MoveIt!的demo来运行此示例：

```
$ roslaunch seven_dof_arm_config demo.launch
```

运行拾取和放置的程序：

```
$ rosrun seven_dof_arm_test pick_place
```

图12-10是抓取过程的截图。

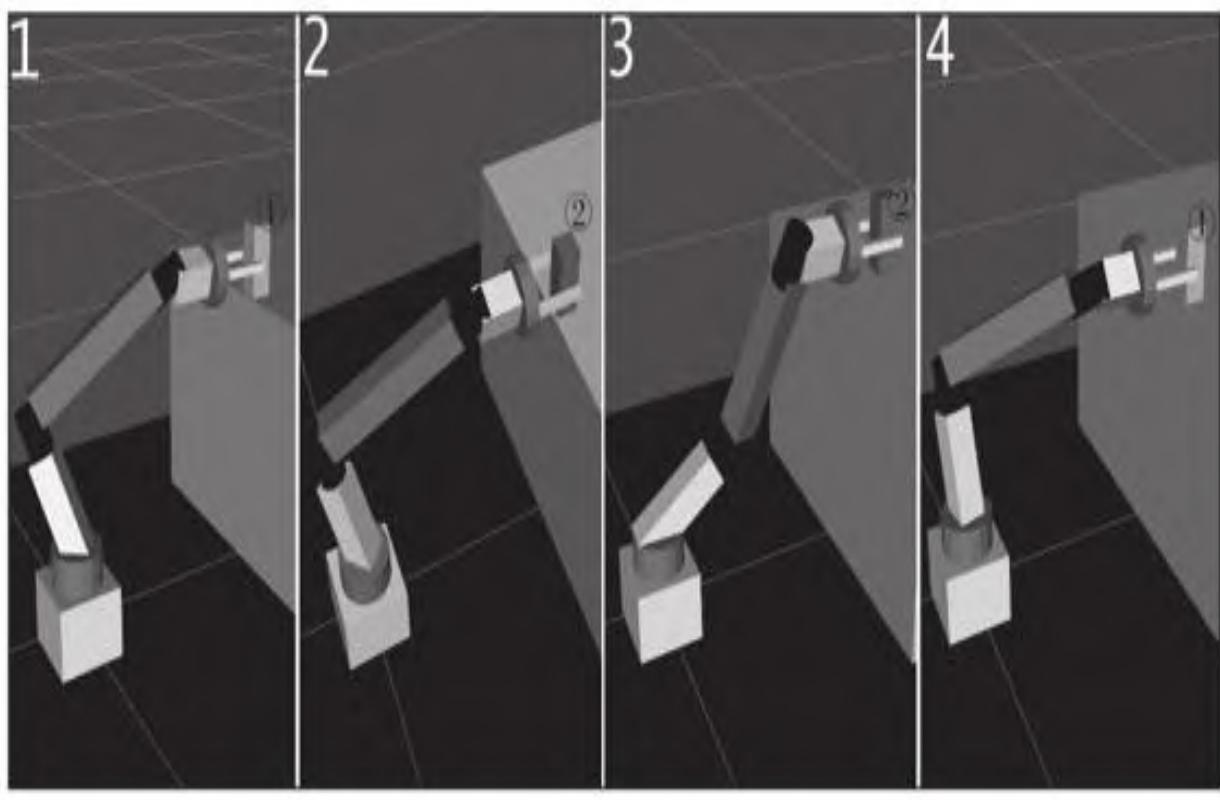


图12-10 用MoveIt!拾取和放置的顺序

接下来解释抓取过程中的各个步骤：

- 在第一步中，我们可以看到一个绿色方块（见图12-10中①所示），这是一个机器人将要抓取的物体。我们用pick_and_place节点在规划场

景中创建了该物体。节点的第一个操作就是让机器人的末端执行器接近待抓取物体。

- 接近物体后，生成可以抓取物体的有效轨迹。抓取完成后，绿色方块将被夹在机器人的夹爪上，并将其颜色改为紫色（见图12-10中②所示）。
- 拾取方块后，机器人将其运送到工作空间的另一个位置，然后将其放在工作台上。如果目的位置上存在有效的IK，夹爪将抓着物体在规划的轨迹上运动。
- 最后，将物体放在桌子上，然后机器人的夹爪松开。

另外一种执行拾取和放置任务的方法是使用MoveIt!提供的操作。启动MoveIt!后，有两个动作服务会启动：

- pickup：此动作将接收moveit_msgs::PickupGoal消息，前提是需要指定待抓取对象及可能抓取的配置列表。这些配置存放在moveit_msgs::Grasp中，在接近、抓取过程中，我们都必须设置完整的关节位置和末端执行器位置。
- place：该动作会将物体放置在一个平面上。它需要moveit_msgs::PlaceGoal类型的消息，该消息需要指定可能被抓取的物体及其被放置的位置。

使用MoveIt!虽然可以确保拾取和放置任务的成功，但却需要大量预先规划的信息，这使得它们难以在高级复杂和动态机器人应用中使用。

12.4.1 使用GPD计算抓握姿态

在本节中，我们将介绍一种ROS软件包：抓取姿态检测器（Grasp Pose Detector）（<https://github.com/atenpas/gpd>），它能够检测双指夹爪的6-DOF抓握姿态，例如我们的seven_dof_arm的夹爪。抓取姿态的检测需要使用3D点云，因此我们可以用机器人的深度传感器来找到物体抓取姿态。该软件包利用深度学习和GPU来计算检测得到场景中所有物体的抓取姿态。若要下载该软件包，只需下载下面的代码库：

```
$ git clone https://github.com/atenpas/gpd.git
```

因为要利用GPU的并行计算，并且要使用该软件包，所以需要一个Nvidia的显卡。编译GPD必须安装下面这些库：

```
$ sudo apt-get install libgflags-dev libprotobuf-dev liblmdb-dev  
libleveldb-dev libsnappy-dev libatlas3-base
```

另外，GPD用Caffe框架（<https://github.com/BVLC/caffe>）实现深度学习功能。要想安装该软件，需要遵循以下流程：

```
$ git clone https://github.com/BVLC/caffe.git  
$ cd caffe && mkdir build && cd build  
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=/usr ..  
$ make && sudo make install
```

如果一切顺利的话，我们可以创建一个ROS工作空间来编译GPD软件包。编译完成后，我们可以用下面的命令在示例数据集上测试GPD：

```
$ rosrun gpd tutorial0.launch
```

在出现的GUI中，按下r键使视图居中，然后按下q键退出GUI并加载下一个可视化窗口。它显示了点云测试集的几个抓取姿态，如图12-11所示。

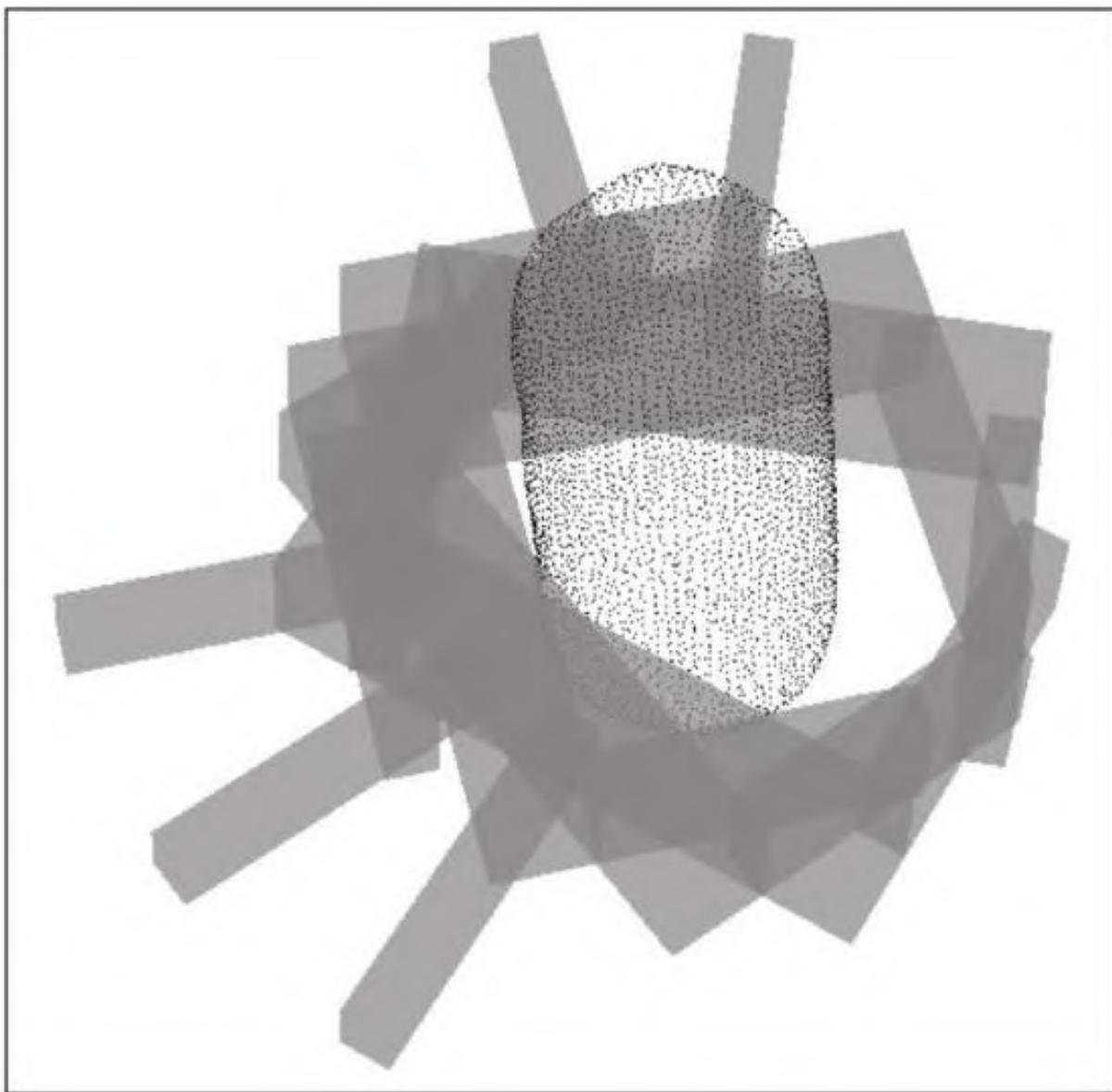


图12-11 在点云测试集上生成抓取姿态

为了在我们的机器人系统中测试GPD软件包，我们可以使用Gazebo中包含待抓取物体的仿真场景：

```
$ rosrun seven_dof_arm_gazebo seven_dof_arm_grasp.launch
```

生成仿真场景后，我们就可以用下面的命令来启动GPD软件：

```
$ rosrun seven_dof_arm_test gpd.launch
```

我们可以在RViz中显示由GPD生成的可能抓取姿态，如图12-12所示。

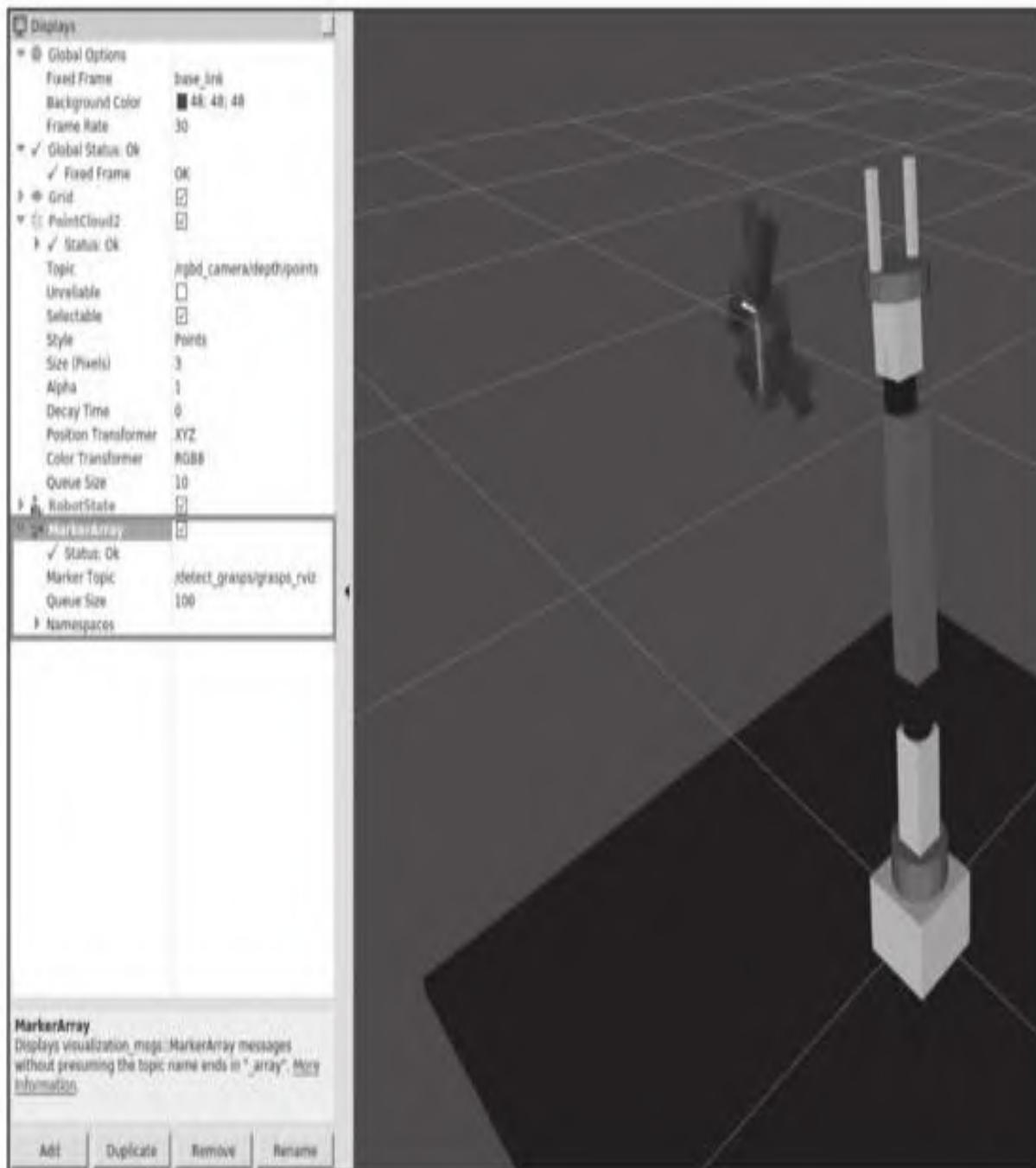


图12-12 在规划场景中物体的抓取姿态

在下面的代码中，我们将看到GPD启动文件的部分内容：

```
<!-- Load hand geometry parameters -->
<include file="$(find gpd)/launch/hand_geometry.launch">
    <arg name="node" value="detect_grasps" />
</include>
```

首先，我们必须通知系统末端执行器的几何形状。在 `hand_geometry` 启动文件中，包含了定义夹爪宽度和深度的参数列表：

```
<param name="cloud_topic" value="/rgbd_camera/depth/points" />
<param name="num_samples" value="100" />
<param name="num_threads" value="4" />
<param name="num_selected" value="10" />
```

在这里，我们定义了点云的话题名和用于检测抓取姿态的样本数量。当然，更多的样本需要消耗更多的计算时间。最后，`num_selected` 参数指定了需要输出的抓取姿态数量。这些姿态都会有得分，得分最高的位于列表的顶部。

选择了一个抓取姿态后，你就可以从 `/detect_grasps/clustered_grasps` 话题中获取有关放置机械手的位置信息。在此消息中，将发布抓取配置的列表，该列表定义了抓取操作期间末端执行器的位置和姿态。特别需要注意时，该消息包含以下字段：

- `bottom`: 表示机械手底部中心的3D点。
- `top`: 表示机械手顶部中心的3D点。
- `surface`: 表示物体表面中心位置的3D点。
- `approach`: 表明靠近抓取方向的3D向量。
- `binormal`: 表明抓取表面方向的3D向量。

- axis: 垂直于靠近方向向量的3D向量。
- width: 可以抓取物体的夹爪的打开宽度。
- score: 抓握姿态得分。

不幸的是，目前还没有提供GPD和MoveIt!之间的直接接口。

12.4.2 在Gazebo和真实机器人上执行拾取和放置动作

在MoveIt!演示中执行的抓取序列使用的是伪控制器（fake controller）。我们可以将轨迹发送给真实的机器人或Gazebo。在Gazebo中，我们可以启动抓取场景来执行此操作。

在真实的硬件中，唯一的区别是我们需要为手臂创建关节轨迹控制器。其中一种常用的硬件控制器是DYNAMIXEL控制器。我们将在下一节进一步了解DYNAMIXEL控制器。

12.5 理解用于机器人硬件接口的DYNAMIXEL ROS伺服控制器

到目前为止，我们已经学习了用Gazebo仿真的MoveIt!接口。在本节，我们将学习如何替换Gazebo并将一个真实的机器人与MoveIt!连接。我们来讨论DYNAMIXEL伺服舵机和ROS控制器。

12.5.1 DYNAMIXEL伺服舵机

DYNAMIXEL伺服舵机是一种用于高端机器人应用的智能、高性能执行机构。这些舵机由一家名为ROBOTIS (<http://en.robotis.com/>) 的韩国公司制造。这些舵机在机器人爱好者中非常受欢迎，因为它们可以提供出色的位置和扭矩控制，并且还能提供各种反馈，例如位置、速度、温度和电压等。它们的一个有用功能是它们可以用菊花链的方式来组网。此功能在多关节系统中非常有用，例如机器人手臂、人形机器人和蛇形机器人等。该舵机可以用ROBOTIS提供的USB转DYNAMIXEL控制器的设备来直接连接到PC上。该控制器有USB接口，当它插入PC中时，它将作为一个虚拟COM端口。我们可以将数据发送到该端口上，并在内部将RS 232协议转换为TTL和RS 485标准。给DYNAMIXEL加电后，连接上USB转DYNAMIXEL控制器就可以使用它了。DYNAMIXEL舵机支持TTL和RS 485标准。图12-13显示了DYNAMIXEL舵机，它们是MX-106和USB转DYNAMIXEL控制器。



图12-13 DYNAMIXEL舵机和USB转DYNAMIXEL控制器

目前市面上有多种系列的DYNAMIXEL舵机。例如MX-28, 64和Rx-28, 64, 106等。图12-14是DYNAMIXEL舵机用USB转DYNAMIXEL控制器连接PC的示意图。

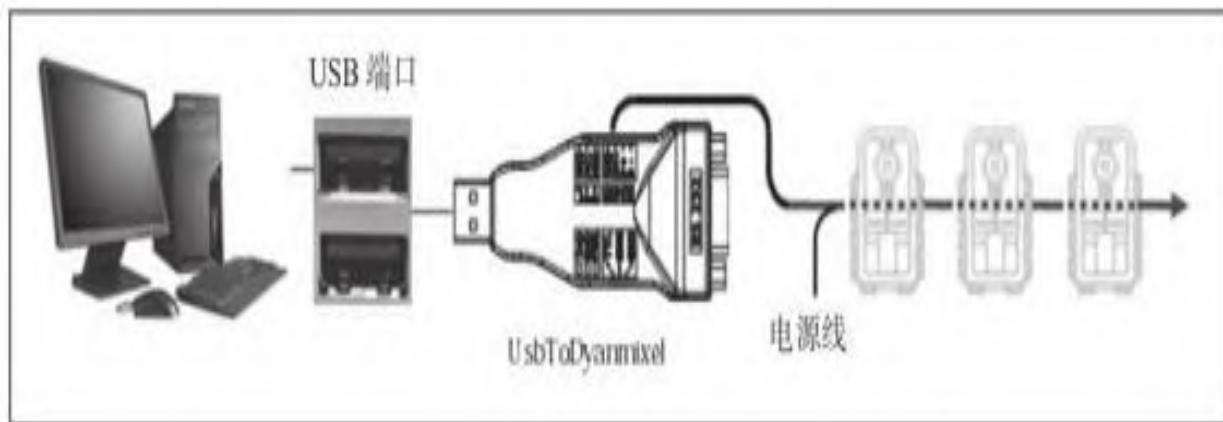


图12-14 DYNAMIXEL舵机用USB转DYNAMIXEL控制器连接PC

多个DYNAMIXEL舵机可以按顺序（或菊花链）连接在一起，如图12-14所示。在每个DYNAMIXEL控制器内都有一个固件设置。我们可以在控制器内分配舵机的ID、关节限值、位置限值、位置命令、PID值、电压限值等。DYNAMIXEL提供了ROS驱动程序和控制器代码，可以在http://wiki.ros.org/dynamixel_motor网址中查询。

12.5.2 DYNAMIXEL-ROS接口

用于连接DYNAMIXEL马达的ROS软件包集是dynamixel_motor。该软件包集包含了一些DYNAMIXEL马达的接口，例如MX-28、MX64、MX-106、RX-28、RX64、EX106、AX-12，及AX-18。该软件包集包含下面这些软件包：

- dynamixel_driver：该软件包是DYNAMIXEL的驱动程序，可以在PC与DYNAMIXEL进行底层IO通信。该驱动程序包包含前面提到的系列马达的硬件接口，并且可以通过该软件包对DYNAMIXEL进行读写操作。该软件包被上层软件包调用，例如dynamixel_controllers。少数情形下，用户才需要直接使用该软件包。
- dynamixel_controllers：这是一个上层调用的软件包，用该软件包，我们可以为机器人的每个DYNAMIXEL关节创建一个ROS控制器。该软件包包含一个可配置的节点、服务和脚本，可用于启动、停止和重启一个或多个控制器插件。在每个控制器中，我们可以设置速度和扭矩。每个DYNAMIXEL控制器都可以用ROS参数进行设置，也可以通过加载YAML文件来修改参数。dynamixel_controllers软件包支持位置、扭矩和轨迹控制器。
- dynamixel_msgs：该软件包包含在dynamixel_motor软件包集中使用的消息定义。

12.6 7-DOF机械臂与ROS MoveIt!

在本节，我们将讨论一款叫作COOL arm-5000的7-DOF机械臂，它由一家名为ASIMOVE Robotics (<http://asimovrobotics.com/>) 的公司制造。如图12-15所示。这款机器人使用的是DYNAMIXEL舵机。我们将看到如何使用dynamixel_controllers软件包将基于Dynamixel的机械臂连接到ROS。

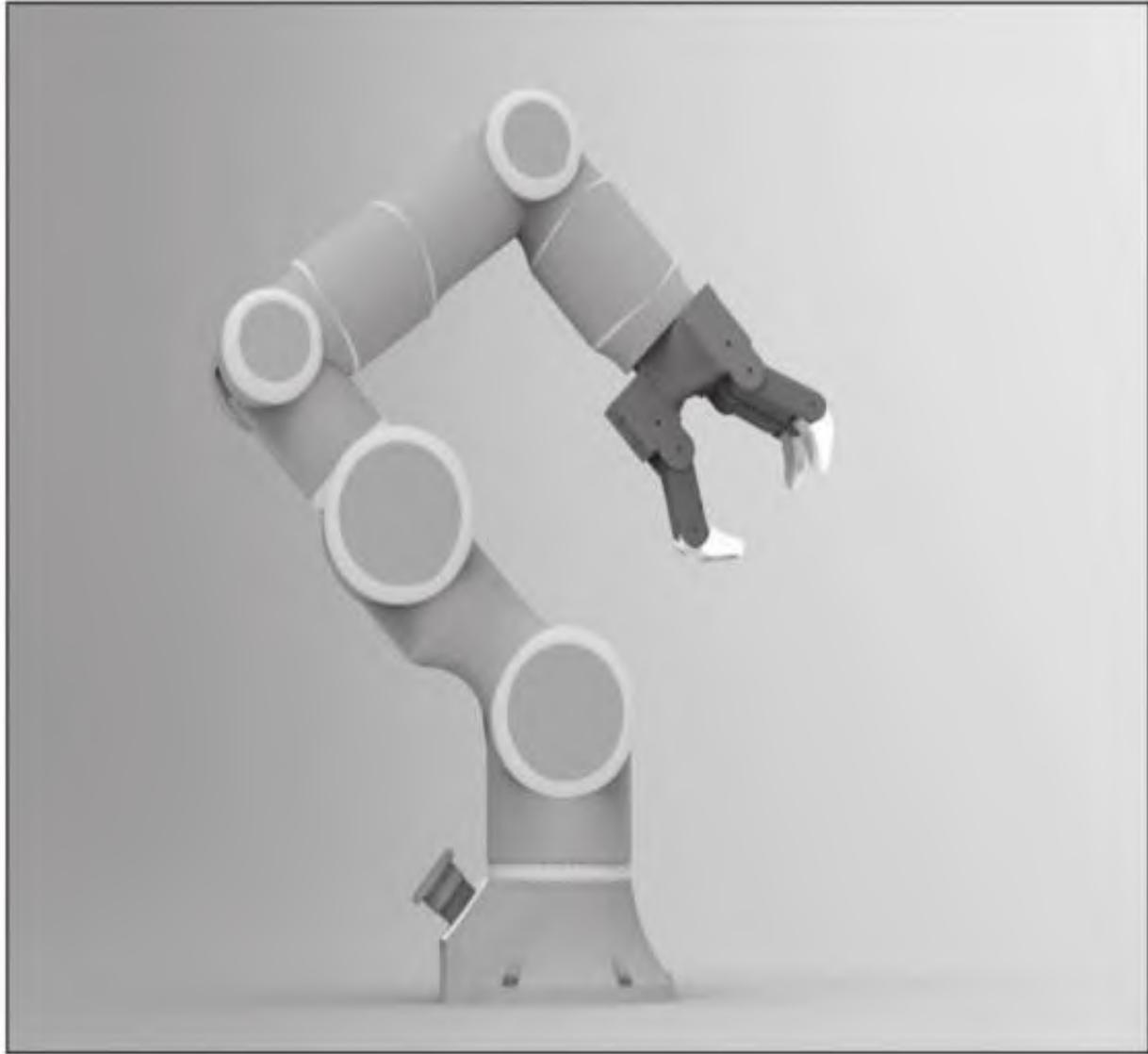


图12-15 COOL机械臂

COOL机械臂与ROS和MOveIt!完美兼容，主要用于教育和研究。下面是该机械臂的详细规格：

- 自由度：7
- 执行机构类型：DYNAMIXEL MX-64和MX-28
- 关节列表：肩部横滚角、肩部俯仰角、肘部横滚角、肘部俯仰角、手腕偏航角、手腕俯仰角和手腕横滚角

- 有效负载: 5 kg
- 工作范围: 1米
- 工作空间: 2.09 立方米
- 重复精度: +/-0.05mm
- 3指夹爪

12.6.1 为COOL机械臂创建一个控制器软件包

第一步是为COOL机械臂创建一个控制器软件包，用于连接ROS。该软件包可以在本书附带的代码中找到。创建软件包之前，我们需要安装dynamixel_controllers软件包：

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

下面这条命令将创建带有依赖项的控制器软件包。该软件包的重要依赖项是dynamixel_controllers软件包：

```
$ catkin_create_pkg cool5000_controller roscpp rospy dynamixel_controller  
std_msgs sensor_msgs
```

下一步是为每个关节创建配置文件。配置文件名是cool5000.yaml，其中包含每个控制器的名称、类型及其参数的定义。我们可以在cool5000_controller/config文件夹中看到该配置文件。我们必须为这个机械臂的7个关节创建参数。下面是该配置文件的代码片段：

```
joint1_controller:  
    controller:  
        package: dynamixel_controllers  
        module: joint_position_controller  
        type: JointPositionController  
    joint_name: joint1  
    joint_speed: 0.1  
motor:  
    id: 0  
    init: 2048  
    min: 320  
    max: 3823  
joint2_controller:  
    controller:  
        package: dynamixel_controllers  
        module: joint_position_controller  
        type: JointPositionController  
    joint_name: joint2  
    joint_speed: 0.1  
motor:  
  
id: 1  
init: 2048  
min: 957  
max: 3106
```

在控制器配置文件中涉及关节名称、控制器软件包、控制器类型、关节转速、马达ID、初始位置以及关节的最小和最大限值。我们可以根据需要连接多个马达，然后通过把它们包含在配置文件中来创建控制器参数。下一个要创建的是joint_trajectory控制器的配置文件。MoveIt!只能在机器人具有FollowJointTrajectory动作服务器时

才能连接。配置文件cool5000_trajectory_controller.yaml位于cool5000_controller/config文件夹中，其定义如以下代码所示：

```
cool5000_trajectory_controller:  
  controller:  
    package: dynamixel_controllers  
    module: joint_trajectory_action_controller  
    type: JointTrajectoryActionController  
  joint_trajectory_action_node:  
    min_velocity: 0.0  
    constraints:  
      goal_time: 0.01
```

创建了JointTrajectory控制器后，我们需要创建一个joint_state_aggregator节点，该节点用于整合和发布机械臂关节的状态。你可以从cool5000_controller/src文件夹中找到该节点的源码joint_state_aggregator.cpp。该节点的功能是订阅每个dynamixel::JointState类型消息的控制器的控制状态，并将控制器的每个消息都整合进sensor_msgs::JointState消息中，然后在/joint_states话题中发布。该消息是所有DYNAMIXEL控制器的关节状态的聚合。下面是joint_state_aggregator.launch的定义，它运行joint_state_aggregator节点及其参数，位于cool5000_controller/launch文件夹中：

```
<launch>
  <node name="joint_state_aggregator" pkg="cool5000_controller"
    type="joint_state_aggregator" output="screen">
    <rosparam>
      rate: 50
      controllers:
        - joint1_controller
        - joint2_controller
        - joint3_controller
        - joint4_controller
        - joint5_controller
        - joint6_controller
        - joint7_controller
        - gripper_controller
    </rosparam>
  </node>
</launch>
```

我们可以用位于 launch 文件夹中的 cool5000_controller.launch 来启动整个控制器。该启动文件将启动 PC 与 DYNAMIXEL 马达直接通信，并且还将启动控制管理器。控制管理器的参数包括串口号、波特率、马达 ID 范围和刷新频率：

```
<launch>

    <!-- Start the Dynamixel motor manager to control all cool5000 servos --
->

    <node name="dynamixel_manager" pkg="dynamixel_controllers"
type="controller_manager.py" required="true" output="screen">
        <rosparam>
            namespace: dxl_manager
            serial_ports:
                dynamixel_port:
                    port_name: "/dev/ttyUSB0"
                    baud_rate: 1000000
                    min_motor_id: 0
                    max_motor_id: 6
                    update_rate: 20
        </rosparam>
    </node>
```

下一步，通过读取控制器的配置文件来启动控制器生成器：

```
<!-- Load joint controller configuration from YAML file to parameter
server -->
<rosparam file="$(find cool5000_controller)/config/cool5000.yaml"
command="load"/>

<!-- Start all Cool Arm joint controllers -->
<node name="controller_spawner" pkg="dynamixel_controllers"
type="controller_spawner.py"
    args="--manager=dxl_manager
          --port dynamixel_port
          joint1_controller
          joint2_controller
          joint3_controller
          joint4_controller
          joint5_controller
          joint6_controller
          joint7_controller
          gripper_controller"
    output="screen"/>
```

在代码的另一部分，它将从控制器配置文件启动JointTrajectory控制器：

```
<!-- Start the cool5000 arm trajectory controller -->
<rosparam file="$(find
cool5000_controller)/config/cool5000_trajectory_controller.yaml"
command="load"/>
<node name="controller_spawner_meta" pkg="dynamixel_controllers"
type="controller_spawner.py"
    args="--manager=dxl_manager
          --type=meta
```

```
--type=meta
cool5000_trajectory_controller
joint1_controller
joint2_controller
joint3_controller
joint4_controller
joint5_controller
joint6_controller"
output="screen"/>
```

下面这部分将从cool5000_description软件包中启动
joint_state_aggregator节点和机器人描述：

```
<!-- Publish combined joint info -->
<include file="$(find
cool5000_controller)/launch/joint_state_aggregator.launch" />
<param name="robot_description" command="$(find xacro)/xacro.py '$(find
cool5000_description)/robots/cool5000.xacro'" />
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" output="screen">
  <rosparam param="source_list">[joint_states]</rosparam>
  <rosparam param="use_gui">FALSE</rosparam>
</node>
```

以上就是COOL机械臂控制器软件包的全部内容了。接下来，我们需要在COOL机械臂的MoveIt!配置软件包中配置控制器，该软件包是cool5000_moveit_config。

12.6.2 COOL机械臂的MoveIt!配置

第一步是配置`controllers.yaml`，它位于`cool5000_moveit_config/config`文件夹中。该文件的定义如下。目前，我们只专注于移动机械臂，而不处理夹具控制。所以配置只包含了机械臂的关节组：

```
controller_list:  
  - name: cool5000_trajectory_controller  
    action_ns: follow_joint_trajectory  
    type: FollowJointTrajectory  
    default: true  
    joints:  
      - joint1  
      - joint2  
      - joint3  
      - joint4  
      - joint5  
      - joint6  
      - joint7
```

下面是`cool5000_description_moveit_controller_manager.launch.xml`文件的定义，该文件位于`cool5000_moveit_config/launch`文件夹中：

```
<launch>  
<!--
```

```
Set the param that trajectory_execution_manager needs to find the
controller plugin
-->
<arg name="moveit_controller_manager"
default="MoveIt_simple_controller_manager/MoveItSimpleControllerManager"/>

<param name="MoveIt_controller_manager" value="$(arg
MoveIt_controller_manager)"/>

<!-- load controller_list -->

<rosparam file="$(find cool5000_moveit_config)/config/controllers.yaml"/>
</launch>
```

配置了MoveIt!后，我们就可以使用它了。为机械臂提供一个适当的电源，然后用USB将DYNAMIXEL与PC连接。我们将看到生成了一个串口设备，它可能是/dev/ttyUSB0或/dev/ttyACM0。根据设备名来修改启动文件中的端口名称。

用下面的命令启动cool5000机械臂控制器：

```
$ rosrun cool5000_controller cool5000_controller.launch
```

启动RViz演示，并开始进行路径规划。如果我们按下Execute按钮，轨迹将在机械臂上执行：

```
$ rosrun cool5000_moveit_config 5k.launch
```

随机姿态（在RViz中显示）和COOL机械臂如图12-16所示。



图12-16 COOL-ARM-5000原型及其MoveIt!可视化

12.7 习题

- FCL库在MoveIt!中的作用是什么？
- MoveIt!如何构建环境的OctoMap？
- 机械臂在抓住一个物体后如何避开障碍物？
- ROS中的GPD软件包的主要作用是什么？
- DYNAMIXEL马达的主要特点是什么？

12.8 本章小结

在本章中，我们探讨了MoveIt!的一些高级功能，以及如何将其与真实的硬件连接。本章首先讨论了使用MoveIt!进行碰撞检查，学习了如何使用MoveIt! API添加碰撞对象，还学习了如何将物体直接导入到规划场景中。我们讨论了用MoveIt! API来碰撞检测的ROS节点。了解了碰撞后，我们为MoveIt!增加了感知功能。我们将仿真点云数据连接到MoveIt!，并在MoveIt!中创建了OctoMap。紧接着我们讨论的主题是如何执行拾取和放置操作来操控场景中的物体。我们用一个ROS软件包从物体的点云中自动生成抓取姿态。讨论了这些后，我们开始讨论使用DYNAMIXEL马达及其ROS控制器硬件的MoveIt!接口。最后，我们学习了一个真正的机械臂（COOL机械臂），了解了它与MoveIt!的接口。该机械臂完全使用DYNAMIXEL控制器来构建。下一章，我们将讨论如何将Matlab（一款世界著名的数值计算环境）与ROS连接。

第13章

在MATLAB和Simulink中使用ROS

在前面的章节中，我们讨论了如何通过由C++实现的ROS节点仿真和控制机器人。在本章，我们将学习如何用MATLAB（一款功能强大的软件）创建ROS节点，通过几个工具箱提供算法和硬件的连接，可以为地面车辆、操作器、仿人机器人，开发自主机器人应用。此外，MATLAB集成了Simulink（基于模型设计的框图环境），可以通过图形编辑器实现我们的控制程序。在本章，我们还将讨论如何用Simulink实现机器人应用。

本章的第一部分是对MATLAB和机器人系统工具箱（Robotic System Toolbox）的简要介绍。学习了如何在ROS和MATLAB之间交换数据后，我们将为差速驱动的移动机器人Turtlebot设计一个避障系统，演示在系统开发中使用机器人系统工具箱中已有的组建来减少工作量是多么简单。我们将在本章的第二部分介绍Simulink，首先以初始模型为例，然后讨论一个发布者和订阅者模型来演示Simulink和ROS之间的通信接口。最后，在Simulink中开发一个用于调试Turtlebot机器人方向的控制系统，并在Gazebo仿真器中进行测试。

本章将介绍以下内容：

- 学习使用MATLAB、MATLAB-ROS以及机器人系统工具箱。
- 在MATLAB中使用ROS话题。
- 使用MATLAB和Gazebo开发机器人应用程序。
- 使用Simulink和Simulink-ROS接口。
- 在Simulink和Gazebo中开发一个控制系统。

13.1 学习使用MATLAB与MATLAB-ROS

MATLAB (MATrix LABoratory) 是一个多平台数值计算环境，被工业、大学和研究中心广泛使用。MATLAB诞生于数学软件，但现在它为不同领域提供了许多附加软件包，例如控制设计、绘图、图像处理和机器人等。MATLAB是MathWorks的产品，不是免费软件。免费许可证通常是分发给学生和学术机构的。你可以在Windows、GNU/Linux和macOS中使用MATLAB。启动之后，MATLAB的主窗口显示其默认布局，如图13-1的截图所示。



图13-1 MATLAB的默认主窗口及其布局

该窗口包括三个主要面板：

- Current Folder（当前文件夹）：这里显示本地文件。

- Command Window (命令窗口) : 这里输入MATLAB命令或运行MATLAB脚本。

- Workspace (工作区) : 显示命令窗口或MATLAB脚本中的数据。

在命令窗口中，你可以执行相关数学命令，可以创建标量，并在工作区中显示。你也可以在命令窗口中查看MATLAB函数文档。事实上，所有内置的MATLAB代码都有支持文档，包括函数输入、输出和调用语法的示例和描述。你可以用doc或help命令访问这些文档。第一条命令会打开一个包含文档的外部窗口，第二条命令会在命令窗口中显示文档。让我们看看如何获取有关平均函数的文档：

```
>> doc mean
```

你也可以使用此命令：

```
>> help mean
```

13.1.1 机器人系统工具箱和ROS-MATLAB接口入门

除了由MATLAB的默认安装提供的标准功能外，还有几个外部工具箱可以访问其他实用程序和库。为了实现ROS和MATLAB之间的通信，我们需要ROS-MATLAB的接口，它是机器人系统工具箱（Robotic System Toolbox, <https://it.mathworks.com/products/robotics.html>）的一部分。该工具箱提供了几种算法，可帮助我们开发自动化机器人应用，例如路径规划器、避障方法、状态估计、运动学和动力学算法。

此外，该工具箱实现了MATLAB和ROS之间的接口，使得开发人员能够在真实机器人和机器人仿真器上测试并移植他们的应用程序。你可以在MATLAB的安装过程中从软件包列表中添加机器人系统工具箱（如图13-2），也可以从工具箱网站上购买：

利用机器人系统工具箱，我们可以将MATLAB转换为真正的ROS节点，该节点能够与系统的其他节点交换信息，并能用话题和服务直接控制仿真或真实的ROS机器人。图13-3描述了MATLAB和ROS之间的连接。将MATLAB连接到ROS节点管理器后，它可以从机器人或其他ROS节点获取数据。MATLAB本身就可以初始化ROS节点管理器，以便管理与网络节点的通信，或者它也可以连接到另一个远程的ROS节点管理器，就像ROS网络中的其他资源一样。此外，在应用程序的最终版本中，我们不必在我们的计算机上允许MATLAB来执行它，我们可以将开发的应用程序像典型的C++节点那样来部署。

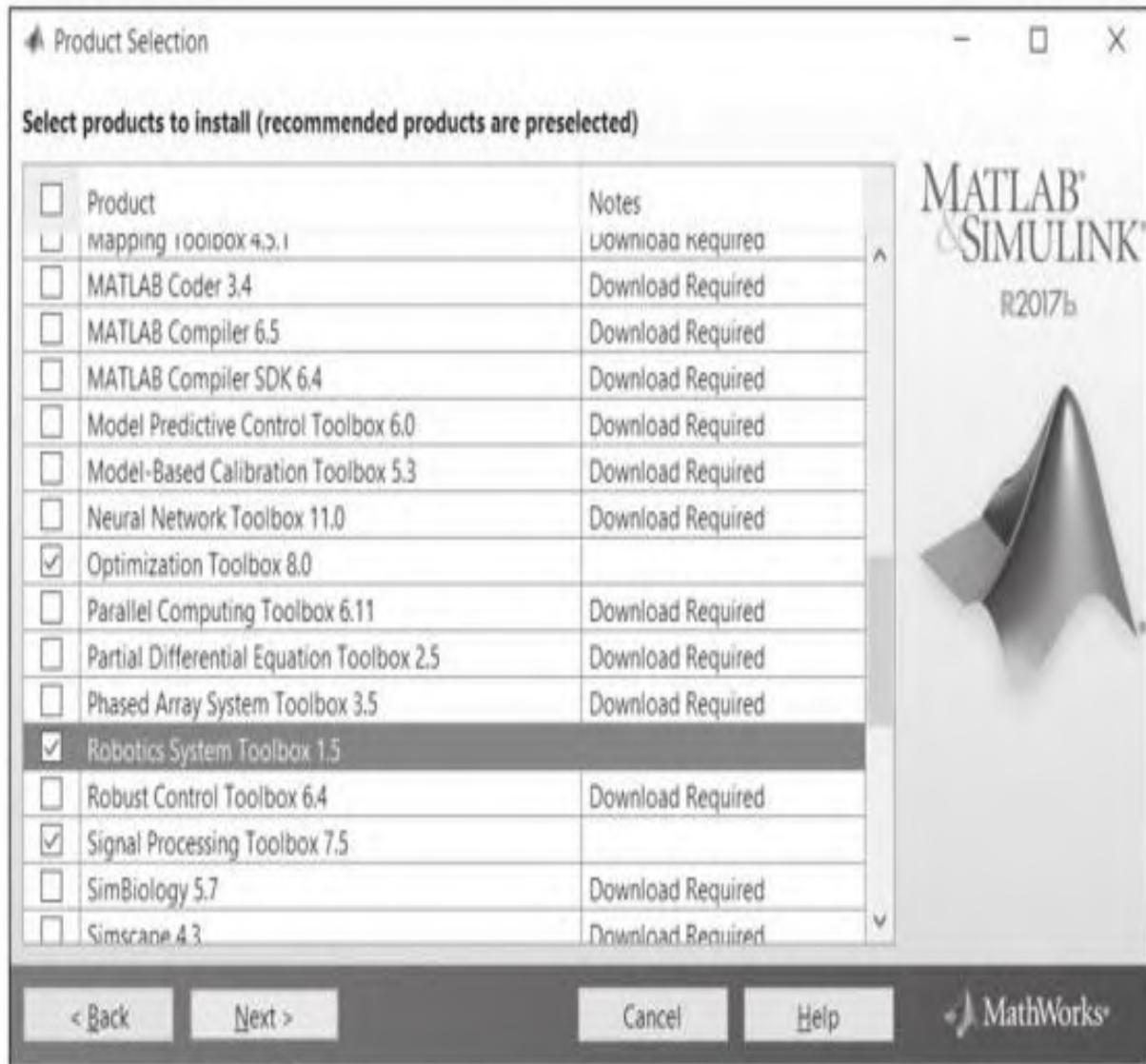


图13-2 在MATLAB的安装过程中选择机器人系统工具箱

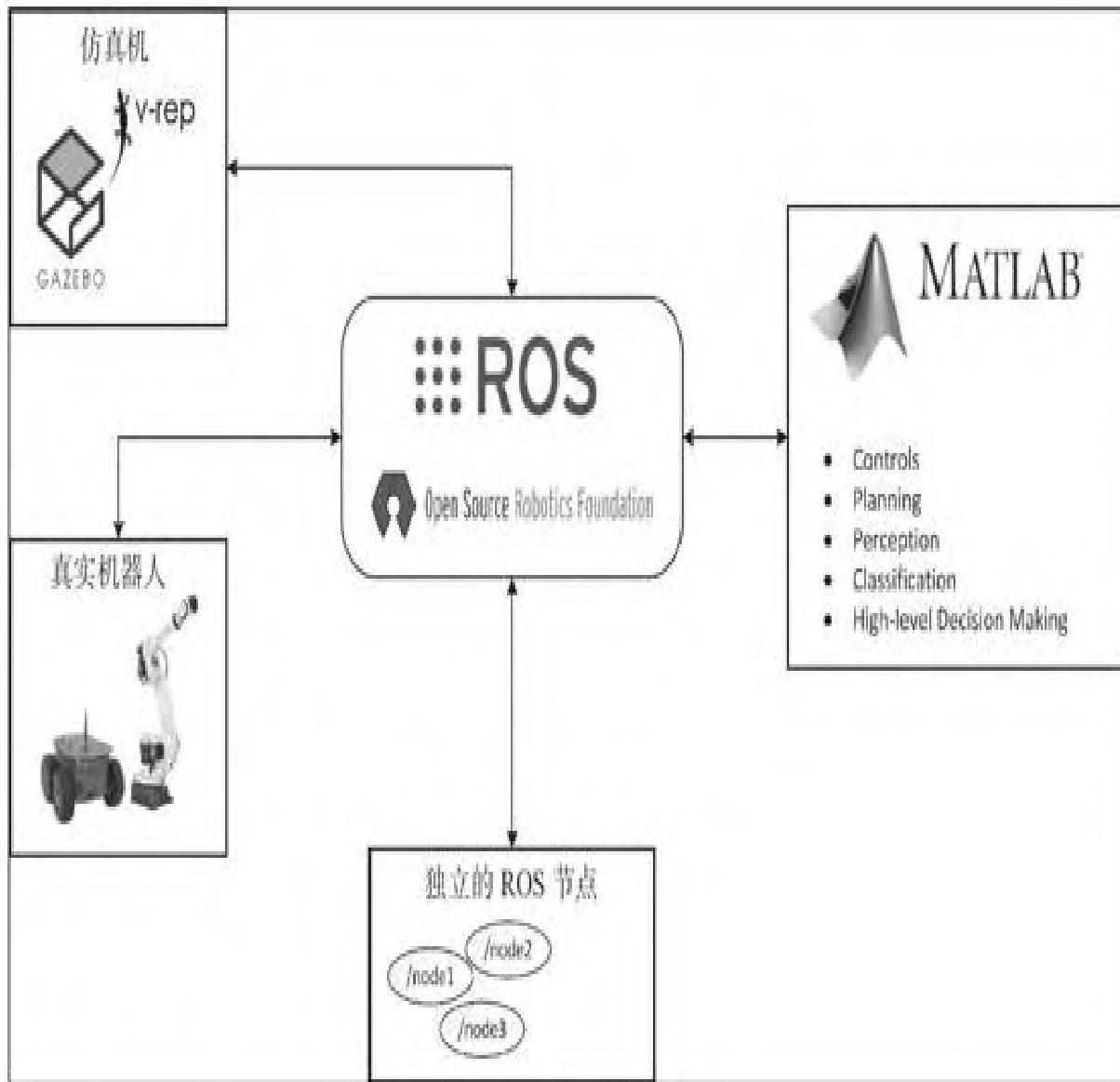


图13-3 ROS-MATLAB的接口模式

通过安装机器人系统工具箱，我们可以访问若干个与Linux下使用的命令相同的ROS命令。要列出这些命令，可以在命令窗口中输入下面的命令行：

```
>> help robotics.ros
```

该命令的输出如图13-4所示。

```
>> help robotics.ros

ros (Robot Operating System)
  rosinit          - Initialize the ros system
  rosshutdown      - Shut down the ros system

  rosmessage       - Create a ros message
  rospublisher     - Create a ros publisher
  rossubscriber    - Create a ros subscriber
  rossvcclient     - Create a ros service client
  rossvcserver      - Create a ros service server
  rosactionclient   - Create a ros action client
  rostype           - View available ros message types

  rosaction         - Get information about actions in the ros network
  rosmsg             - Get information about messages and message types
  rosnode            - Get information about nodes in the ros network
  rosservice         - Get information about services in the ros network
  rostopic           - Get information about topics in the ros network

  rosbag             - Open and parse a rosbag log file
  rosparam           - Get and set values on the parameter server
  rosrate            - Execute loop at fixed frequency using ros time
  rostf              - Receive, send, and apply ros transformations

  rosduration        - Create a ros duration object
  rostime             - Access ros time functionality

ros functionality is part of Robotics System Toolbox.
Type "help robotics" for more information.
```

图13-4 ROS-MATLAB接口命令

我们可以用rosinit命令初始化ROS-MATLAB接口，用rosshutdown命令停止该接口。默认情况下，rosinit命令在Matlab中创建ROS节点管理器，还将启动matlab_global_node节点与ROS的通信网络。使用rosnode list命令将初始化roscore，之后我们可以看到正在运行的ROS节点，如图13-5所示。

```
>> rosinit
Initializing ROS master on http://DESKTOP-40TG18P:11311/.
Initializing global node /matlab_global_node_16208 with NodeURI http://DESKTOP-40TG18P:61762/
>> rosnode list
/matlab_global_node_16208
```

图13-5 ROS-MATLAB接口的默认初始化

使用ROS-MATLAB接口的默认配置，我们必须使用运行MATLAB的计算机的IP地址在ROS网络上的另一个节点上设置ROS_MASTER_URI环境变量。如果在Windows上运行MATLAB，可以使用下面的命令轻松获取计算机的IP地址：

```
>> !ipconfig
```

如果是在Linux上运行MATLAB，则可以使用下面的命令：

```
>> !ifconfig
```

在Windows系统中该命令的输出如图13-6的截图所示。

```
Scheda LAN wireless Wi-Fi:
```

```
Suffisso DNS specifico per connessione: lan
```

```
Indirizzo IPv6 locale rispetto al collegamento . : fe80::cc11:c374:70f8:a4c4%11
```

```
Indirizzo IPv4 . . . . . : 192.168.1.130
```

```
Subnet mask . . . . . : 255.255.255.0
```

```
Gateway predefinito . . . . . : 192.168.1.254
```

图13-6 在Windows下的MATLAB运行lipconfig命令

我们也可以直接将MATLAB连接到活动的ROS网络中。在这种情况下，我们必须告知ROS-MATLAB接口运行ROS节点管理器的计算机或机器人的IP地址是什么。可以通过下面的命令来操作：

```
>> setenv('ROS_MASTER_URI', 'http://192.168.1.131:11311')
```

```
>> setenv('ROS_MASTER_URI', 'http://192.168.1.131:11311');
>> rosinit
The value of the ROS_MASTER_URI environment variable, http://192.168.1.131:11311, will be used to connect
Initializing global node /matlab_global_node_75920 with NodeURI http://192.168.1.130:61991/
>> rosnode list
/matlab_global_node_75920
/roscout
```

图13-7 在ROS外部网络初始化ROS-MATLAB接口

在下一节，我们将使用话题回调函数初始化ROS-MATLAB接口并且直接从MATLAB脚本中获取数据。

13.1.2 ROS话题和MATLAB回调函数

在本节，我们将讨论如何用MATLAB脚本发布和订阅ROS消息。我们分析的第一个脚本定义了一个典型的模板，用于开发机器人的控制循环。首先，订阅一个输入话题，接着我们将其转发至输出话题上，并持续一段时间。完整的源代码保存在talker.m中，在随本书提供的代码中可以找到，或者也可以从下面的git库中下载：

```
$ git clone https://github.com/jocacace/ros_matlab_test
```

让我们来看看talker.m脚本的内容：

```
ros_master_ip = 'http://192.168.1.5:11311';
matlab_ip = '192.168.1.13';
rosinit(ros_master_ip, 'NodeHost', matlab_ip);
pause(2) % wait a bit the roscore initialization

talker_sub = rossubscriber( '/talker' );
[chatter_pub, chatter_msg] = rospublisher('/chatter','std_msgs/String');
r = rosrate(2); % 2 Hz loop rate

for i = 1:20
    data = talker_sub.LatestMessage;
    chatter_msg.Data = data.Data;
    send(chatter_pub, chatter_msg);
    waitfor(r);
end
rosshutdown
```

让我们来看看该脚本是如何工作的：

```
ros_master_ip = 'http://192.168.1.5:11311';
matlab_ip = '192.168.1.13';
rosinit(ros_master_ip, 'NodeHost', matlab_ip);
```

在前面的代码中，我们初始化了MATLAB-ROS节点。在本例中，我们想将MATLAB连接到外部的ROS网络中，使其能够读取和写入话题数据。因此，我们应该导出ROS_MASTER_URI和ROS_HOSTNAME环境变量。在系统配置的基础上更改IP地址：

```
talker_sub = rossubscriber('/talker');
[chatter_pub, chatter_msg] = rospublisher('/chatter','std_msgs/String');
```

然后，发布/chatter话题（初始化为std_msgs/String类型），订阅/talker话题：

```
data = talker_sub.LatestMessage;
send(chatter_pub, chatter_msg);
```

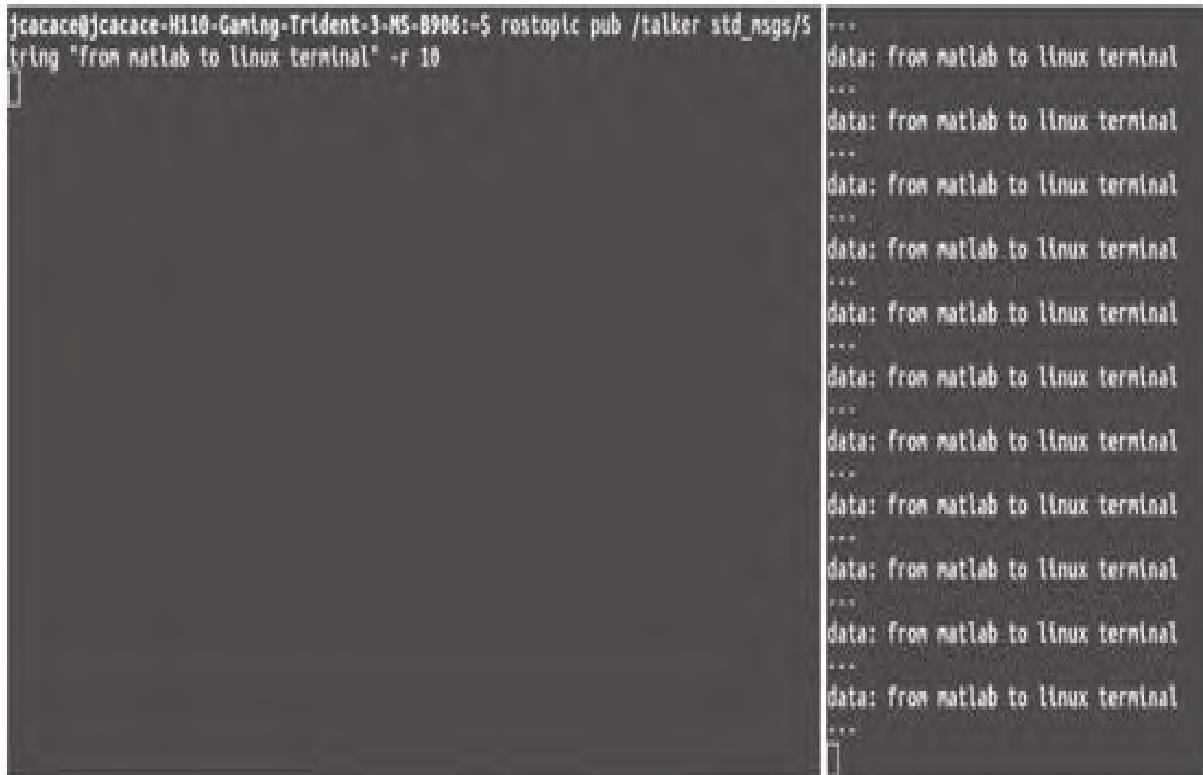
最后，我们用LatestMessage函数获取输入话题的最新消息，同时在/chatter话题上发布消息。

此时，你可以使用与MATLAB计算机对于同一网络中的Linux计算机，用命令行向/talker话题发布所需的消息，并可视化发布在/chatter话题上的消息。在运行MATLAB脚本之前，请确保在要发布消息的计算机上已经正确配置好ROS_HOSTNAME变量，以便MATLAB能够接收已发布的数据。

现在，你可以在命令窗口中输入脚本名称来运行它：

```
>> talker
```

如果一切都已正确设置，那么在Linux电脑上的输出应该如图13-8所示。



```
jcacace@jcacace-H110-Gaming-Trident-3-MS-B906:~$ rostopic pub /talker std_msgs/String "from matlab to linux terminal" -r 10
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
```

图13-8 MATLAB与ROS通信

在前面的脚本中定义了一个典型的模板来实现自主机器人的控制循环。我们可以定义每次收到新消息时调用的回调函数，而不是不断地查询话题上收到的最新消息。通过这种方式，我们可以编写更复杂的控制循环来处理机器人的行为，可以从ROS话题异步接收多个消息。在下一个例子中，我们将ROS-MATLAB连接到Gazebo，仿真Turtlebot机器人并用MATLAB绘制其激光传感器的值。

要运行Gazebo仿真器，我们将使用turtlebot_gazebo软件包：

```
$ rosrun turtlebot_gazebo turtlebot_world.launch
```

启动Gazebo后，就会发布不同的话题，其中就包括/scan。在这个例子中，我们需要下面这些MATLAB功能：

- plot_laser.m：初始化订阅所需激光雷达话题的ROS-MATLAB接口，并以所需的帧率绘制激光数据。
- get_laser.m：接收并存储激光雷达的数据。

让我们看看plot_laser脚本的代码：

```
function plot_laser()
    global laser_msg;
    ros_master_ip = 'http://192.168.1.5:11311';
    matlab_ip = '192.168.1.13';
    rosinit(ros_master_ip, 'NodeHost', matlab_ip);
    pause(2)
    laser_sub = rossubscriber('/scan', @get_laser );
    r = rosrate(2); % 2 Hz loop rate
    for i=1:50
        plot(laser_msg,'MaximumRange',7)
        waitfor(r);
    end
    rosshutdown
    close all
end
```

设置好ROS-MATLAB接口后，我们将初始化订阅激光雷达话题的订阅者：

```
laser_sub = rossubscriber('/scan', @get_laser );
```

这行代码中，get_laser函数处理/scan话题中的数据。要在不同的MATLAB脚本之间交换数据，我们需要使用一个全局变量：

```
global laser_msg;
```

最后，我们绘制25秒的激光雷达数据：

```
plot(laser_msg, 'MaximumRange', 7)
```

现在让我们看一下get_laser函数的代码：

```
function get_laser(~, message)
    global laser_msg;
    laser_msg = message;
end
```

在这个函数中，我们只保存了激光雷达的数据。

启动了Gazebo仿真后，我们可以运行MATLAB脚本：

```
>> plot_laser
```

输出显示在场景中物体的默认位置，如图13-9所示。

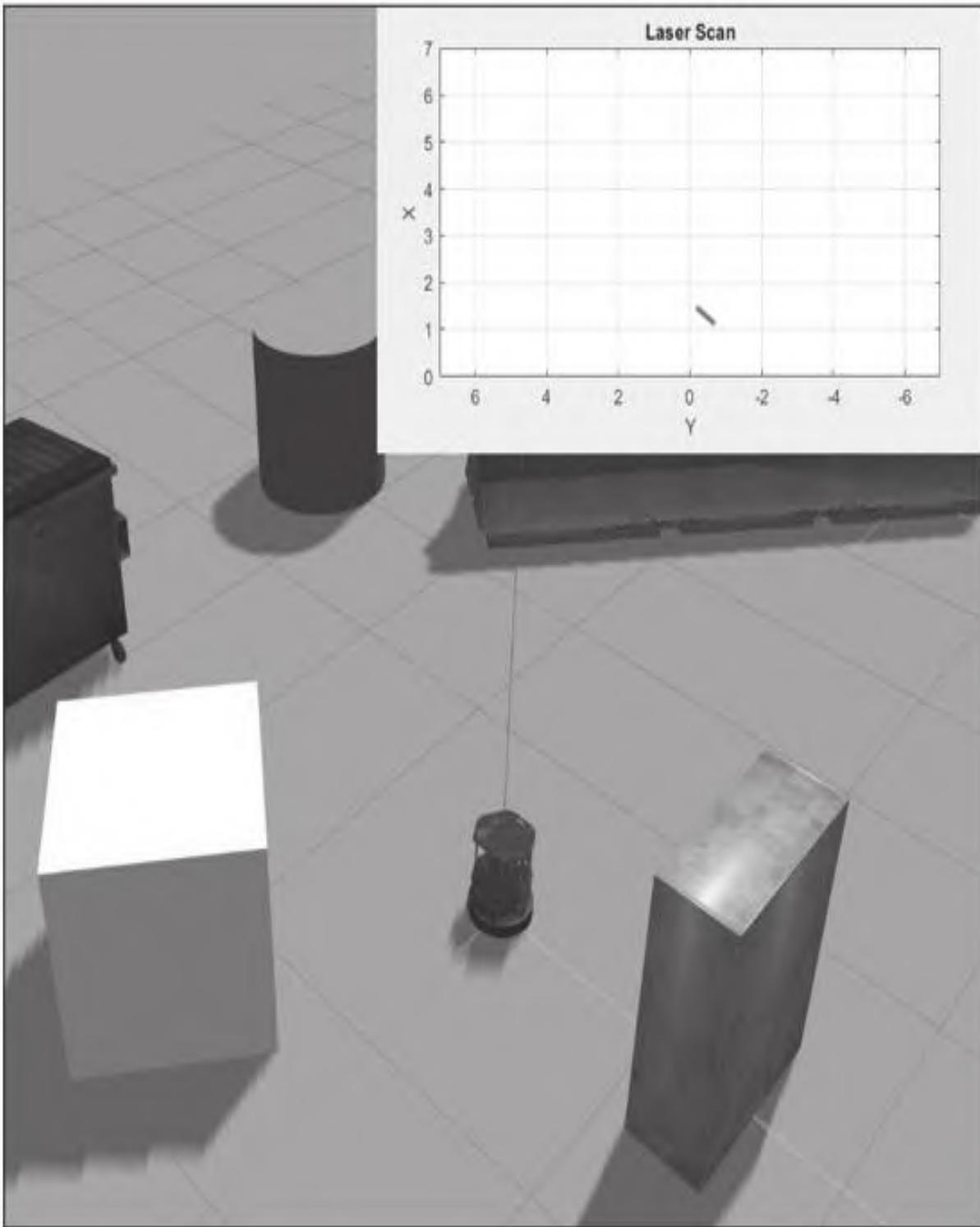


图13-9 在MATLAB中绘制Gazebo激光雷达数据

13.1.3 为Turtlebot机器人设计一个避障系统

到目前为止，我们仅在MATLAB中使用ROS话题来交换数据。在本节，我们将演示用MATLAB和机器人系统工具箱为移动机器人创建机器人应用程序是多么容易。我们将为差速驱动机器人设计一个避障系统，使Turtlebot机器人能够在拥挤的环境中移动而不会碰到任何障碍物。我们将提供一个MATLAB脚本，它将控制机器人的速度来产生随机运动。同时，机器人传感器的激光雷达数据将用于避开障碍物。为了实现这样的行为，我们将依靠矢量直方图（Vector Field Histogram, VFH）算法，该算法根据距离传感器读数计算机器人的无障碍转向方向。该算法已由robotics.VectorFieldHistogram类中的机器人系统工具箱提供。最后，导航一段时间后，将用MATLAB的绘图函数显示一些日志数据。这些可以帮助开发人员调试应用。

我们将要讨论的脚本的完整源代码可以在
vh_obsacle_avoidance.m源文件中找到：

```

function vfh_obstacle_avoidance(ros_master_ip, matlab_ip )
    rosinit(ros_master_ip, 'NodeHost', matlab_ip);
    pause(2);
    laserSub = rossubscriber('/scan');
    odomSub = rossubscriber('/odom');
    [velPub, velMsg] = rospublisher('/mobile_base/commands/velocity');
    vfh = robotics.VectorFieldHistogram;
    vfh.DistanceLimits = [0.05 1];
    vfh.RobotRadius = 0.1;
    vfh.MinTurningRadius = 0.2;
    vfh.SafetyDistance = 0.1;
    r_max = 6.28;
    r_min = 0.0;
    ob_dist = [];
    odom_vel = [];
    rate = robotics.Rate(10);
    odom_vel_x = [];
    odom_vel_z = [];
    odom_pos_x = [];
    odom_pos_y = [];

    while rate.TotalElapsedTime < 5
        % Get laser scan data
        laserScan = receive(laserSub);

        odom = receive(odomSub);
        ranges = double(laserScan.Ranges);
        angles = double(laserScan.readScanAngles);
        odom_vel_x = [odom_vel_x, odom.Twist.Twist.Linear.X];
        odom_vel_z = [odom_vel_z, odom.Twist.Twist.Angular.Z];
        odom_pos_x = [odom_pos_x, odom.Pose.Pose.Position.X];
        odom_pos_y = [odom_pos_y, odom.Pose.Pose.Position.Y];

        targetDir = (r_max-r_min).*rand();
        % Call VFH object to computer steering direction
        steerDir = vfh(ranges, angles, targetDir);
        ob_dist = [ob_dist, min(ranges) ];
        % Calculate velocities

```

```

if ~isnan(steerDir) % If steering direction is valid
    desiredV = 0.2;
    w = exampleHelperComputeAngularVelocity(steerDir, 1);
else % Stop and search for valid direction
    desiredV = 0.0;
    w = 0.5;
end
velMsg.Linear.X = desiredV;
velMsg.Angular.Z = w;
velPub.send(velMsg);
waitfor(rate);
end
rossshutdown
figure(1);
plot( ob_dist, 'red-' );
legend('obstacle distance');
ylabel( 'm' );
grid on;
title('obstacle distance');
figure(2);
plot( odom_vel_x, 'red' );
legend('Forward velocity');
ylabel( 'm/s' );
grid on
title('forward velocity');
figure(3);
plot( odom_vel_z, 'blue' );
legend('Angular velocity');
ylabel( 'rad/s' );
grid on
title('angular velocity');
figure(4)
plot( odom_pos_x, odom_pos_y, 'red' );
xlabel('x');
ylabel('y');
title('path');
grid on
end

```

让我们来解释上面的脚本：

```
function vfh_obstacle_avoidance(ros_master_ip, matlab_ip )  
rosinit(ros_master_ip, 'NodeHost', matlab_ip);  
    pause(2) % wait a bit the roscore initialization
```

通过指定运行roscore的计算机的IP地址和运行MATLAB计算机的IP地址来调用该脚本。通过这种方式，我们可以初始化ROS-MATLAB接口：

```
laserSub = rossubscriber('/scan');  
[velPub, velMsg] = rospublisher('/mobile_base/commands/velocity');  
odomSub = rossubscriber('/odom');
```

现在我们订阅激光扫描信息，然后声明变量来发布控制机器人的命令。rospublisher函数返回实例化的发布者，velPub是发布velMsg类型消息的发布者。此外，我们订阅了机器人的里程计以跟踪其在运动过程中的速度。

现在我们准备实例化VFH对象以实现我们的避障系统：

```
vfh = robotics.VectorFieldHistogram;
```

VFH算法需要一些参数，特别是下面这些：

- DistanceLimits：激光读数的范围，用二维向量指定激光雷达可测量的最大最小范围。
- RobotRadius：指定机器人的尺寸（以米为单位）。
- MinTurningRadius：机器人的最小转弯半径（以米为单位）。

- SafetyDistance：机器人和障碍物之间允许的最大空间：

```
vfh.DistanceLimits = [0.05 1];  
vfh.RobotRadius = 0.1;  
vfh.MinTurningRadius = 0.2;  
vfh.SafetyDistance = 0.1;
```

现在我们准备启动允许机器人运动的控制循环。首先，我们需要定义控制循环的频率：

```
rate = robotics.Rate(10);
```

接下来，描述了运动控制循环。我们希望该控制循环可以执行一段时间。我们用rate.TotalElapsedTime记录已经过的时间。该函数返回创建对象后经过的时间（以秒为单位）。在控制循环内部，我们将从激光扫描话题中读取传感器数据：

```
while rate.TotalElapsedTime < 50  
    laserScan = receive(laserSub);  
        ranges = double(laserScan.Ranges);  
        angles = double(laserScan.readScanAngles);
```

targetDir指定机器人运动的角度方向，其值以弧度来表示，机器人前方被视为零弧度。如上所述，示例中的目标方向是在每个控制循环中随机计算的：

```
targetDir = (r_max-r_min).*rand();
```

然后，我们可以调用vfh方法，根据激光扫描数据和实际所需的运动方向来计算得到无障碍的转向：

```
steerDir = vfh(ranges, angles, targetDir);
```

如果存在有效的转向方向，我们需要计算旋转速度发送给机器人以驱动它。为此，我们将使用以下函数：

```
w = exampleHelperComputeAngularVelocity(steerDir, 1);
```

该函数将返回差速驱动机器人的角速度，以rad/s表示。此外，函数的第二个参数表示最大速度值，以便得到计算的极限值。最后，绘制机器人在其运动过程中检测到的障碍物的最小距离、执行路径以及驱动的角度和前进速度：

```
figure(1);
plot( ob_dist, 'red-' );
figure(2);
plot( odom_vel_x, 'red' );
figure(3);
plot( odom_vel_z, 'blue' );
figure(4)
plot( odom_pos_x, odom_pos_y, 'red');
```

为了测试这个例子，首先需要在我们想运行roscore的计算机上启动Turtlebot仿真环境：

```
$ rosrun turtlebot_gazebo turtlebot_world.launch
```

然后在执行MATLAB脚本时，必须使用我们的ROS网络的正确的IP地址：

```
>> vfh_obstacle_avoidance( '192.168.1.105', '192.168.1.130' )
```

机器人将在图13-8所示的相同环境中进行导航，MATLAB脚本的输出示例如图13-10中各截图所示。

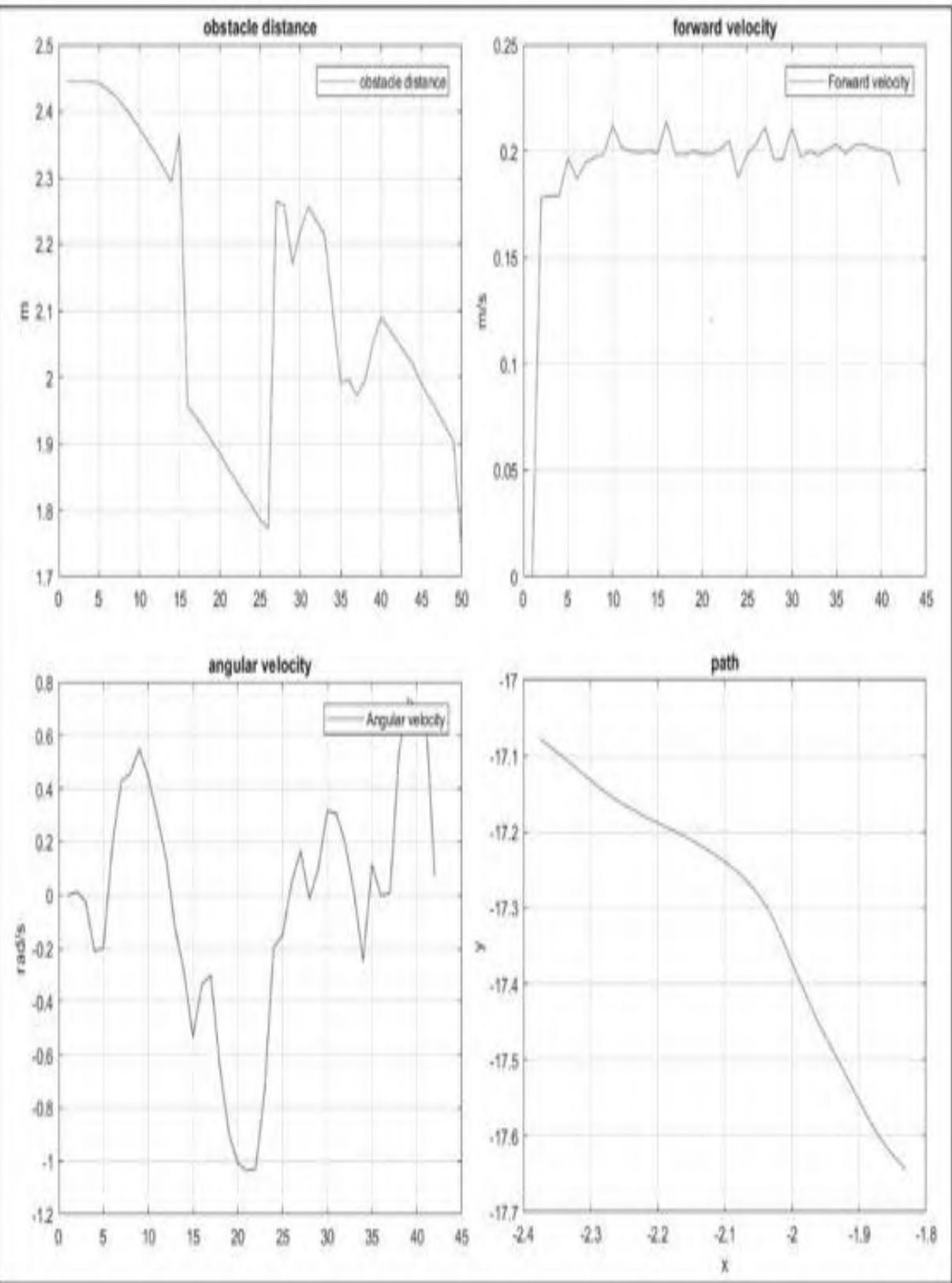


图13-10 用MATLAB打印函数来绘制日志：左上：最小障碍物距离，右上：线性前
向速度，左下角：角速度，右下角：执行路径

13.2 学习使用ROS与Simulink

在前几节，我们讨论了如何使用MATLAB与ROS进行交互。在本节，我们将学习使用MATLAB的另一个强大工具：Simulink。Simulink是一个用于建模、仿真和分析动态系统的图形化编程环境。我们可以用Simulink创建系统模型并仿真其随时间的行为变化。

在本节，我们将从ROS框架开始创建第一个简单的系统。我们还将讨论如何使用Simulink开发ROS应用。

13.2.1 在Simulink中创建波形信号积分器

要创建一个新系统，让我们从打开Simulink开始。我们可以在命令窗口中输入以下命令来打开它：

```
>> Simulink
```

然后，你应该选择创建一个新的空白模型。要创建新系统，我们必须导入构成它所需的Simulink模块。可以从库浏览器（Library Browser）直接将这些模块拖放到模型窗口中。要打开库浏览器，请从模型面板工具条中选View|Library Browser。对于我们的第一个系统，我们需要四个模块：

- 正弦波（Sine Wave）：这会产生一个正弦信号，代表系统的输入。
- 积分器（Integrator）：它集成了输入信号。
- 总线生成器（Bus Creator）：它在一个信号中组合多个信号。
- 示波器（Scope）：以图形方式显示输入信号。

导入这些模块后，你的模型面板应该看起来像图13-11这样。

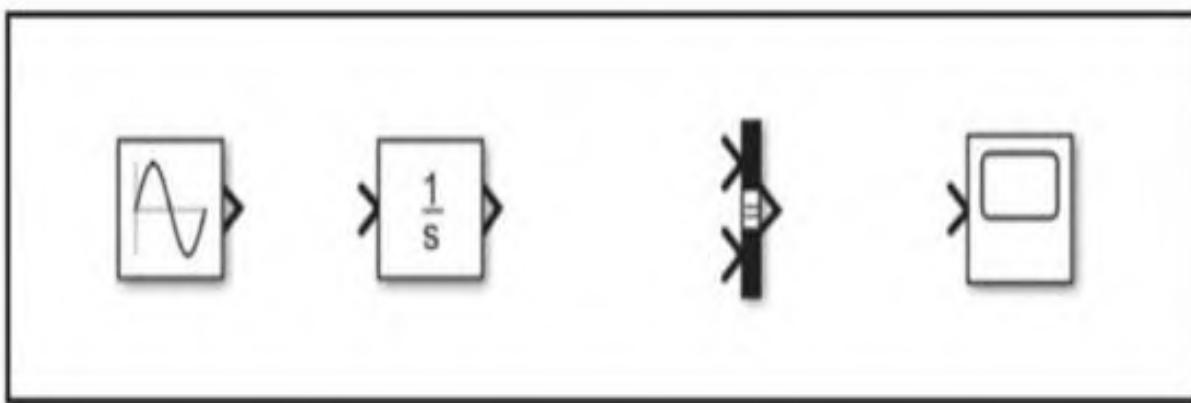


图13-11 Simulink中的正弦波、积分器、总线生成器和示波器模块

一些模块必须使用正确的参数来配置。例如，正弦波模块需要生成正弦信号的幅度和频率。要设置这些值，我们可以通过双击所需的模块来查看这些参数。为了使系统正常工作，我们需要正确连接 simulink 模块，如图13-12所示。

现在已经连接好了模型组件，我们可以仿真我们系统的行为。首先，我们应该配置仿真的持续时间，设置开始和停止仿真时间。打开 Simulation|Model Configuration Parameters 窗口，然后插入所需的值。在我们的示例中，我们设置 Start time: 0 和 Stop time: 10.0，如图13-13所示。

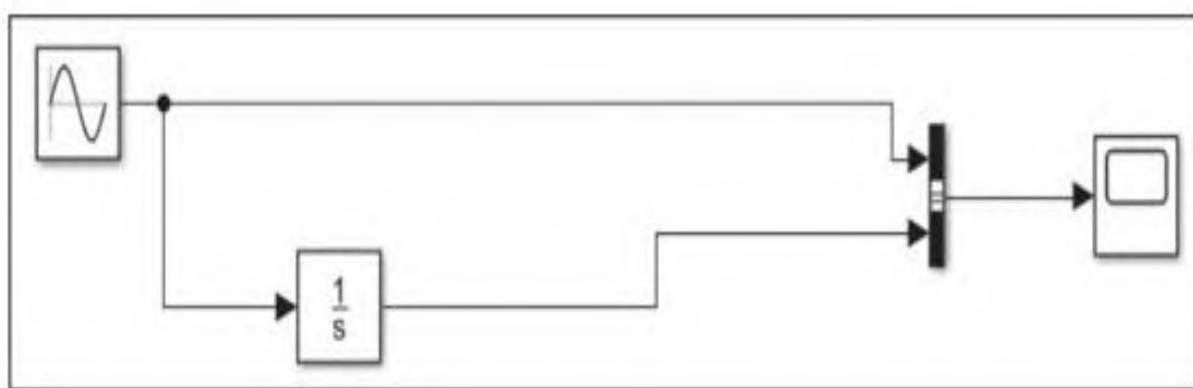


图13-12 正弦信号积分器

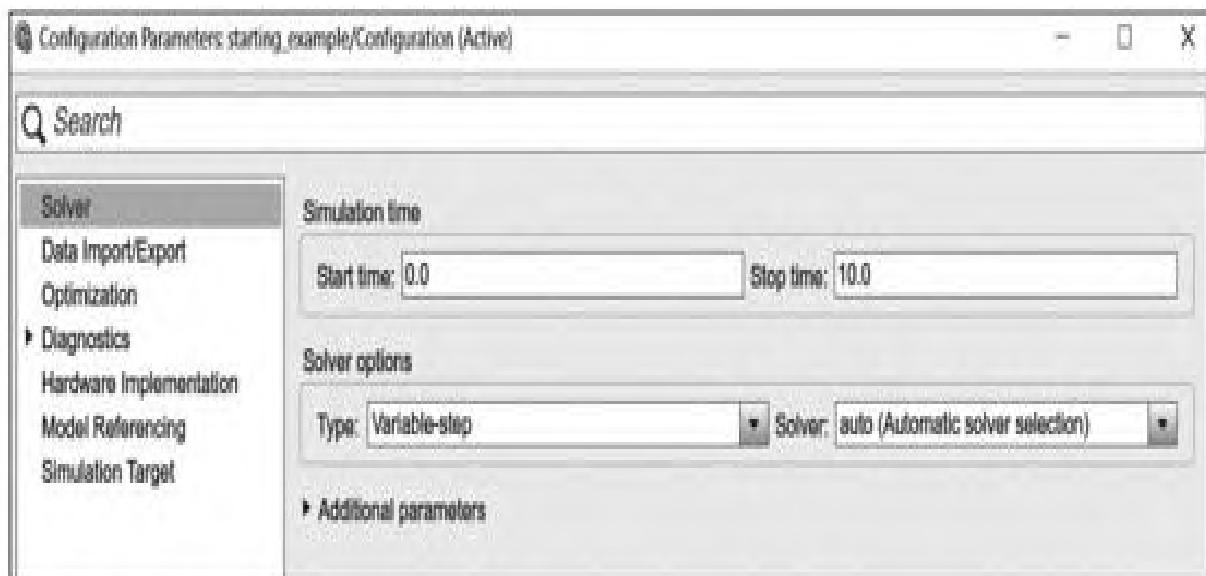


图13-13 我们的系统的仿真时间

现在，我们可以通过按模型面板工具条中的播放按钮，同时双击示波器模块Scope模块来查看输出，如图13-14所示。

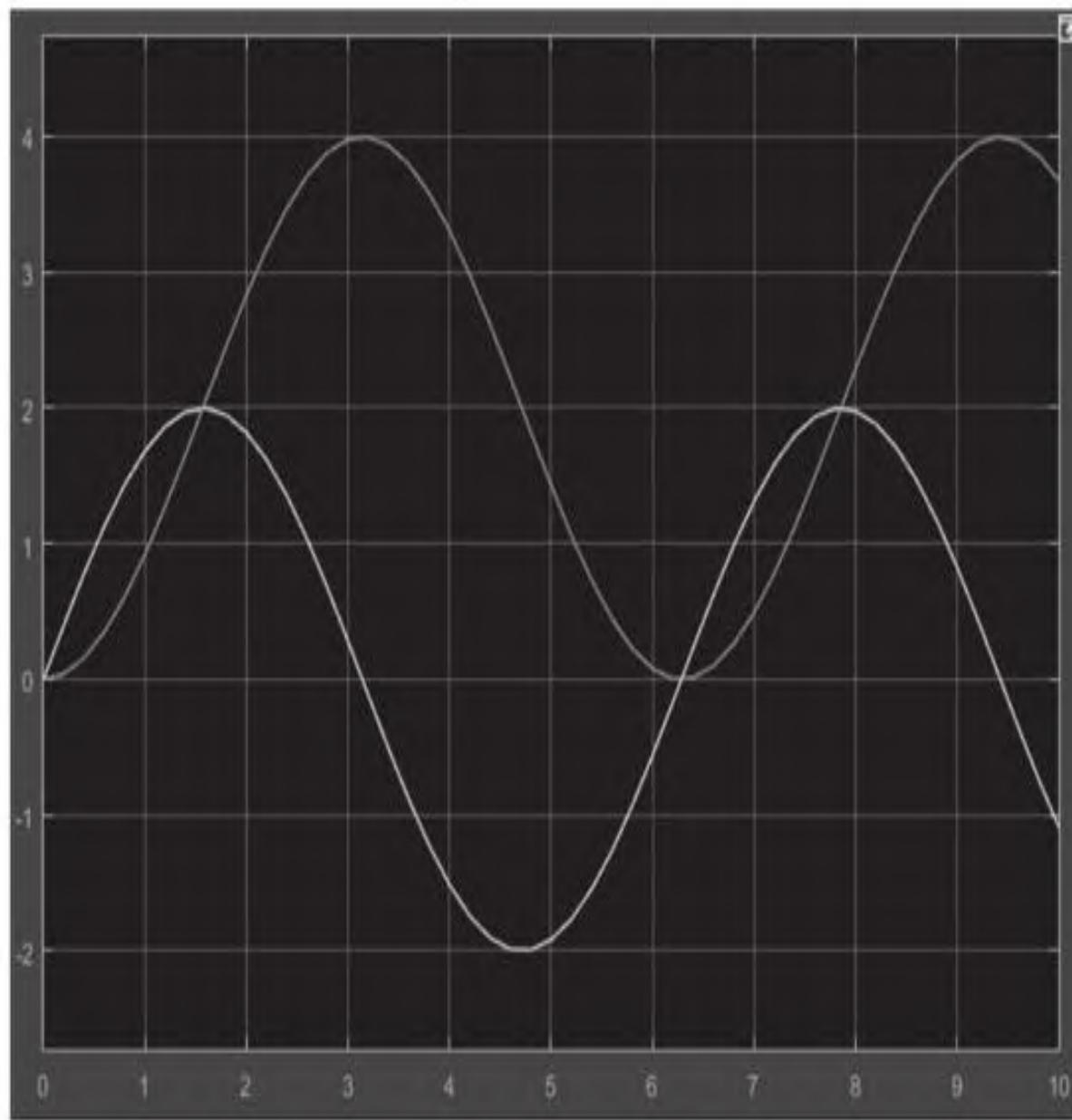


图13-14 正弦和积分信号

注意，即使我们插入10秒的仿真时间，Simulink也不会实时工作，只会仿真增量时间步长。这样，仿真期间的有效时间将非常短。

本示例中提出的模型可以在本书的源码文件中找到：
ros_matlab_test/staring_example.mdl。

13.2.2 在Simulink中使用ROS消息

Simulink对ROS的支持允许我们创建能连接到ROS网络中其他节点上的系统。这种支持包括通过话题发送和接收消息的Simulink模块库。当我们开始仿真开发的模型时，Simulink将尝试连接到ROS网络中，该网络可以在同一台计算机或在另一台远程计算机上运行Simulink。一旦建立好连接后，Simulink将与ROS网络进行消息交换，直到仿真结束。正如我们在上一节中所做的那样，我们将首先展示如何使用ROS话题读取和写入数据，然后讨论如何创建一个更复杂的系统来控制在Gazebo中仿真的Turtlebot机器人。

让我们来创建两个不同的Simulink模型。在一个模型中，我们将创建一个消息发布者，而在另一个模型中，我们将实现一个简单的订阅者。这些模型可以在ros_matlab_test源码文件夹中找到，它们分别是publisher.mdl和subscriber.mdl。

13.2.3 在Simulink中发布ROS消息

要在Simulink中发布ROS消息，我们主要需要两个模块：

- Publish：该模块可以在ROS网络中发布消息。通过配置模块参数，我们可以制定话题名称和消息类型。
- Blank message：该模块可以创建具有指定消息类型的空白消息。

让我们看看如何连接这些模块以在/`position`话题上发布`geometry_msgs/Twist`类型的消息。通过从库浏览器中导入空白消息模块并双击它来配置消息的类型。从模块的参数窗口中，我们可以点击Select按钮从列表中选择ROS消息类型，如图13-15的截图所示。

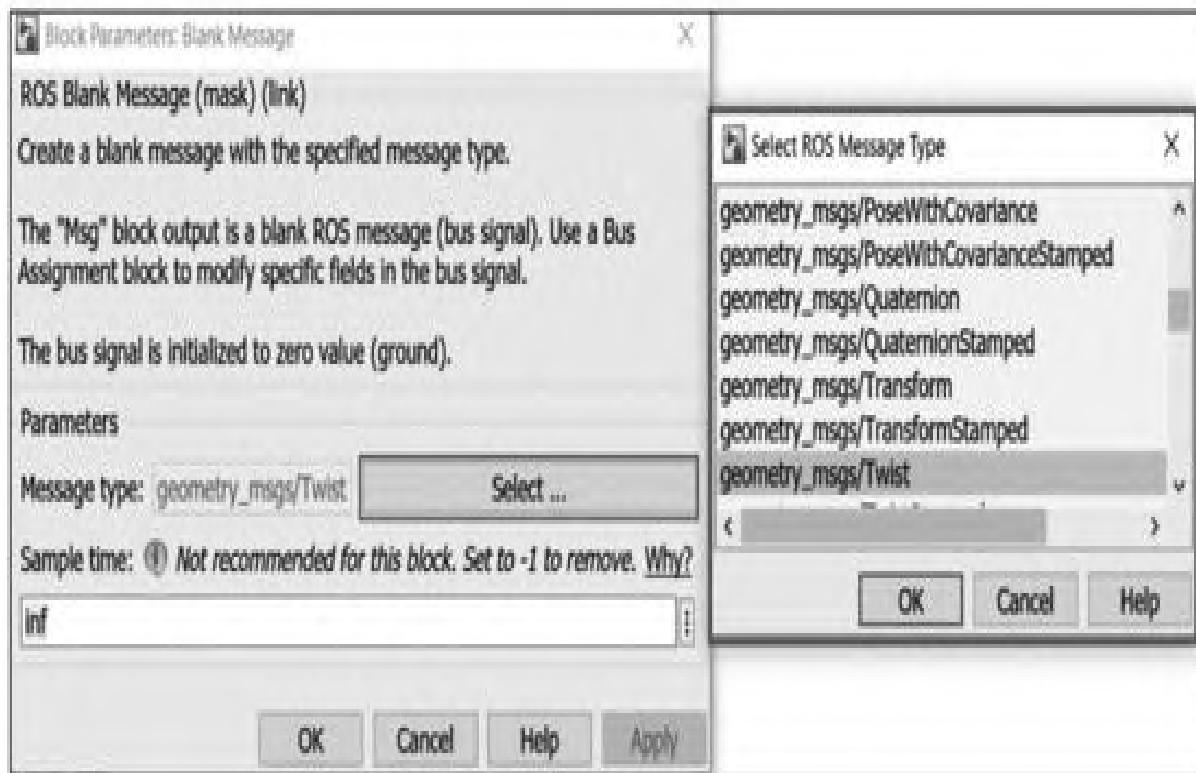


图13-15 在Simulink的ROS空白消息模块中配置参数

现在我们准备导入ROS的发布模块：将模块拖放到模型中，双击它来配置话题名称和消息类型。在话题源中选择Specify your own，这

样就能指定你想要的话题名。在话题栏中输入`/position`。正如我们看到的，我们可以选择要发布的消息类型，如图13-16的截图所示。

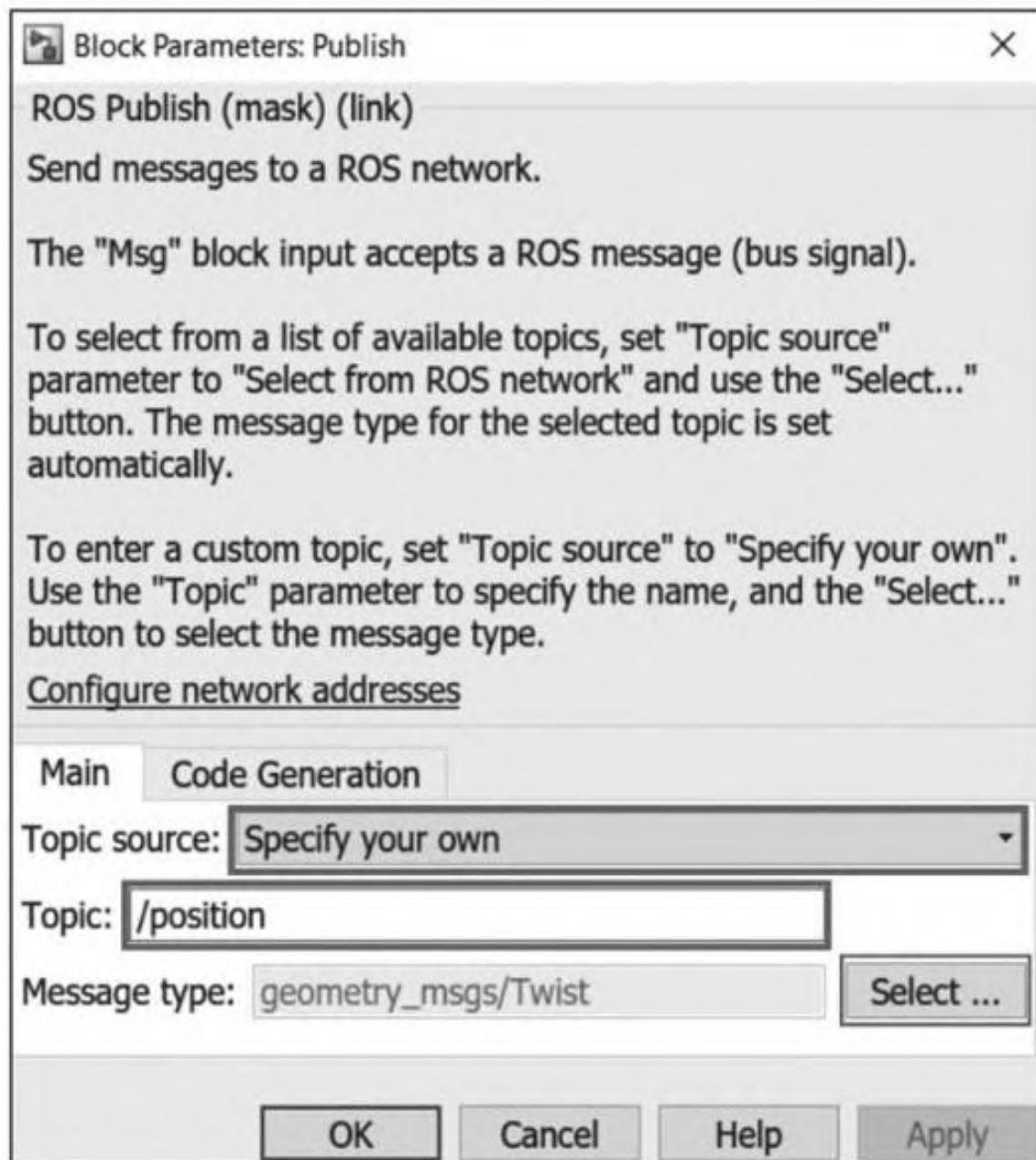


图13-16 在Simulink的ROS发布模块中配置参数

现在，我们必须在将其发布到ROS网络之前填写ROS消息字段。我们使用另外两个Simulink组件来完成这项工作。第一个是Sine Wave（正弦波），产生的正弦信号已经在第一个Simulink示例中使用

过了。第二个是信号总线分配。实际上，ROS消息在Simulink环境中被表示为总线信号，允许我们使用总线信号模块来管理其字段。将空白消息模块的输出端口连接到BusAssignment模块的输入端口。将BusAssignment模块的输出端口连接到ROS发布模块的输入端口。然后双击BusAssignment模块来配置总线信号参数。你应该看到X、Y和Z（包含了geometry_msgs/Twist消息的信号）排列在图13-17的左侧列表中。删除右边列表中的元素，并在左侧列表中选择消息的线性部分的X和Y信号，点击Select>>，然后点击OK来关闭模块。在这里，我们只设置了Twist消息的线性部分的前两部分。

完成总线分配模块的参数配置并接受所选输入信号的值后，模块的形状将改变。现在，我们应该将期望的值分配给这些组件。我们可以使用正弦波模块来实现这一点，就像我们前面的例子中所做的那样，拖放两个正弦信号发生器，将它们连接到总线分配模块。最后的模型如图13-18所示。

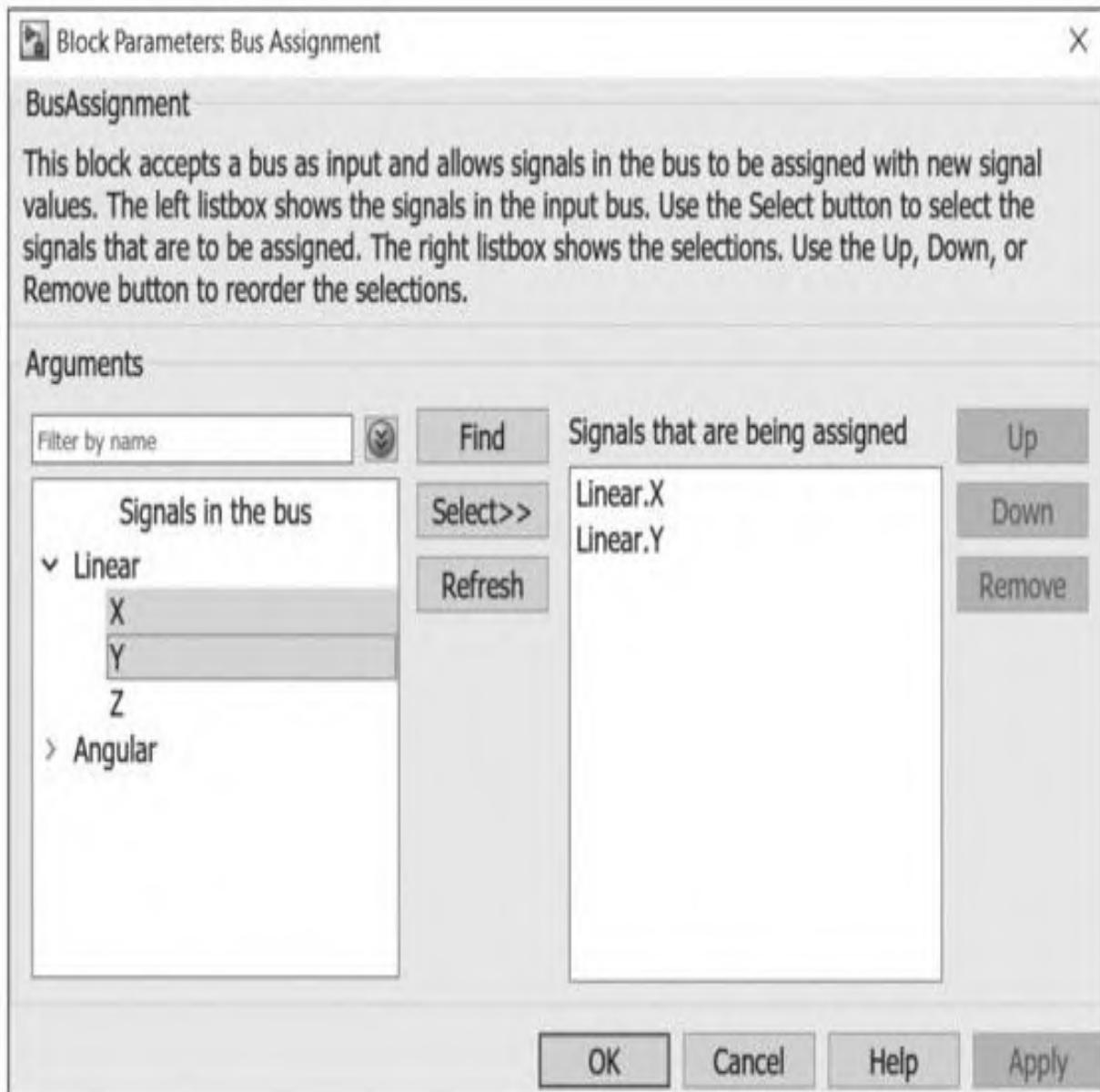


图13-17 geometry_msgs/Twist消息的总线配置

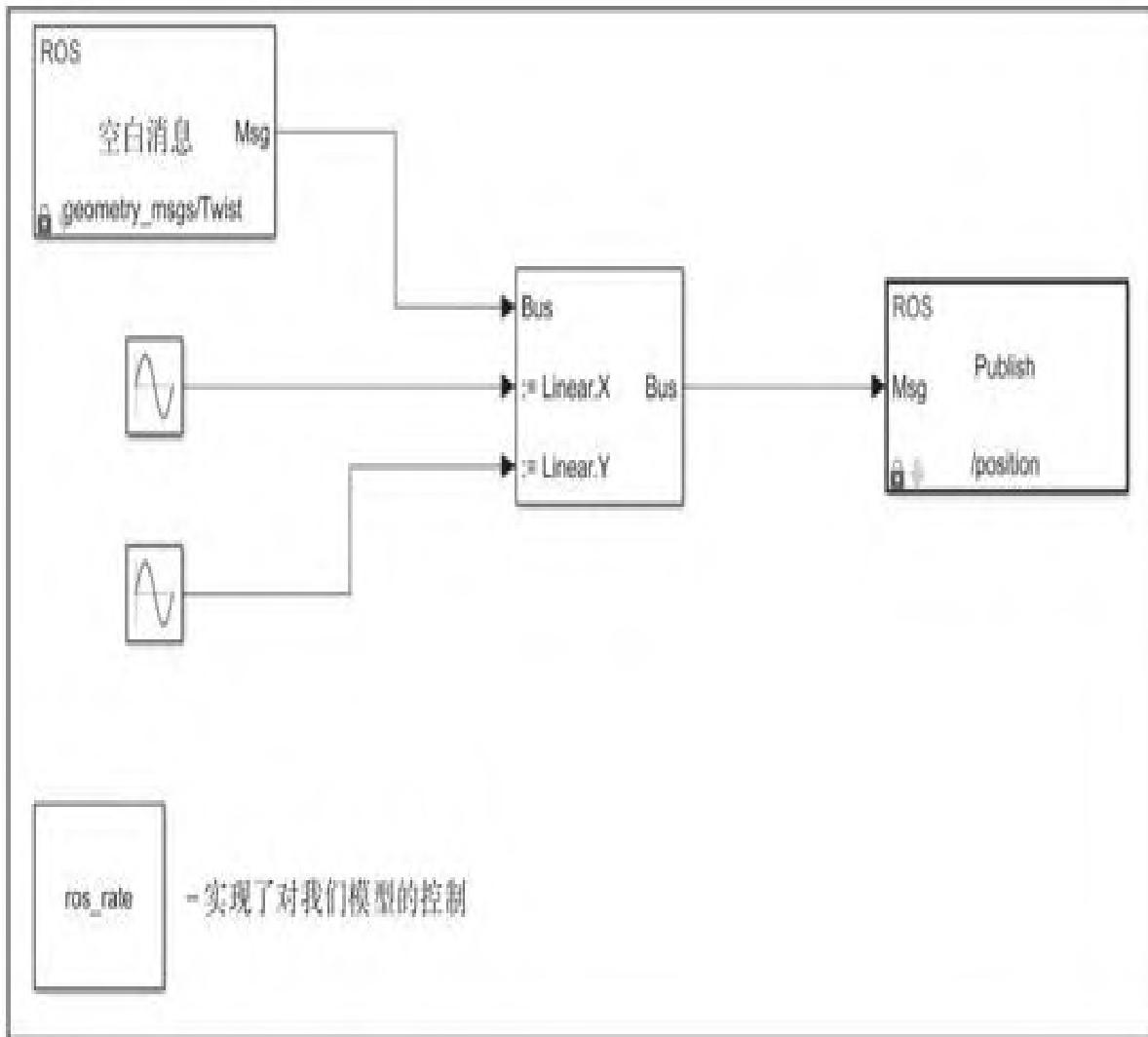


图13-18 Simulink的发布者模型

在我们的Simulink发布者模块中还包含一个额外的模块：
ros_rate。在模型执行期间，我们需要该模块来仿真实时的动作，实现ROS的频率机制。事实上，如果没有这个模块，该节点的执行频率将非常高，它将以最大频率来发布ROS消息。**ros_rate**模块是一个称为MATLAB System的特殊模块，它允许我们实例化并调用matlabclass对象。将该模块导入系统模型后，我们应该选择要调用的System object name或者创建一个新的模块。

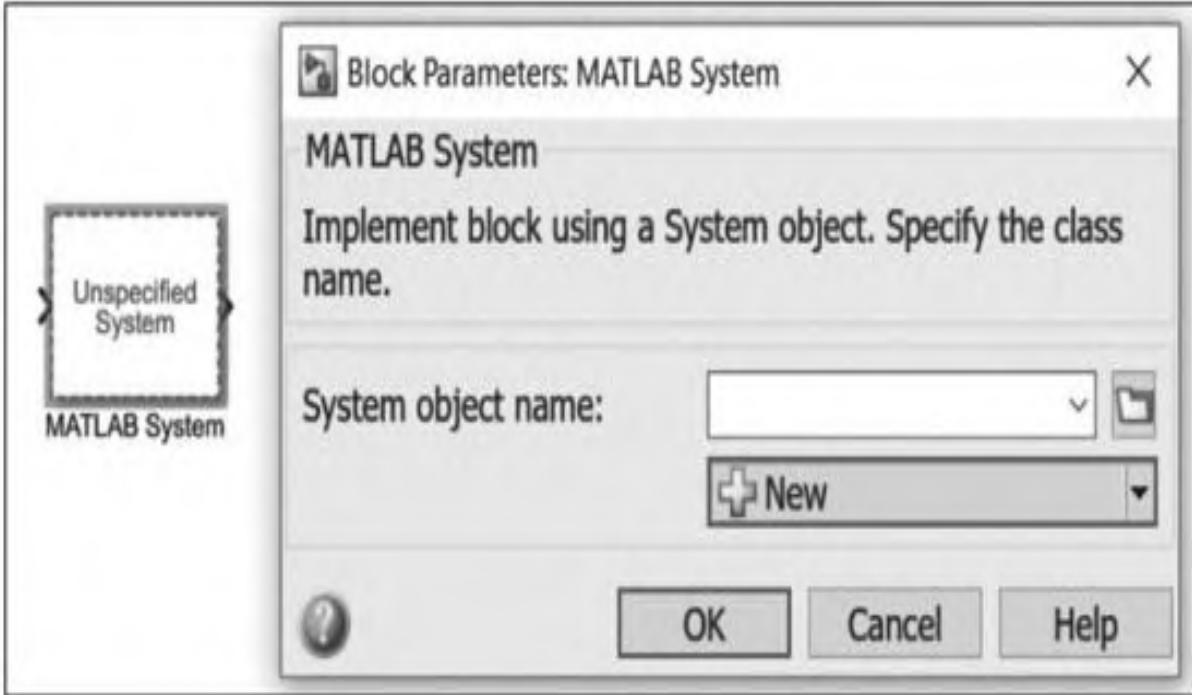


图13-18 Simulink的发布者模型

ros_rate模块的代码位于ros_rate.m源文件中：

```
classdef ros_rate < matlab.System
    % Public, tunable properties
    properties
        RATE;
    end
    properties(DiscreteState)
    end
    % Pre-computed constants
    properties(Access = private)
        rateObj;
    end
    methods(Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing constants
            obj.rateObj = robotics.Rate(obj.RATE);
        end
        function stepImpl(obj)
            obj.rateObj.waitFor();
        end
        function resetImpl(obj)
        end
    end
end
```

在这段代码中，我们定义了ros_rate类，它有两个对象：指定循环频率的rate和用来实现robotics.Rate机制的rateObj。这个类最重要的方法是在仿真开始时调用setupImpl(obj)方法，它用于初始化类的成员变量，以及在每一步都调用stepImpl(obj)方法，以便调节仿真的执行时间。

现在我们的模型已经完成了，我们需要设置仿真的停止时间(stop time)为inf，即设置一个永不结束的仿真时间。通过这种方式，我们可以在需要时使用停止按钮来终止仿真。现在我们可以开始仿真并读取在/position话题上发布的内容了。

13.2.4 在Simulink中订阅ROS话题

要订阅ROS话题，我们只需要（Subscribe）订阅者模块，如图13-20所示。在这种情况下，我们也必需配置要读取的消息类型和话题名称。让我们选择/position话题，以便读取发布者Simulink模块发送到ROS网络的数据。订阅者模块有两个输出：IsNew，一个用来定义是否收到新消息的布尔型信号；Msg，其中包含接收到的消息：

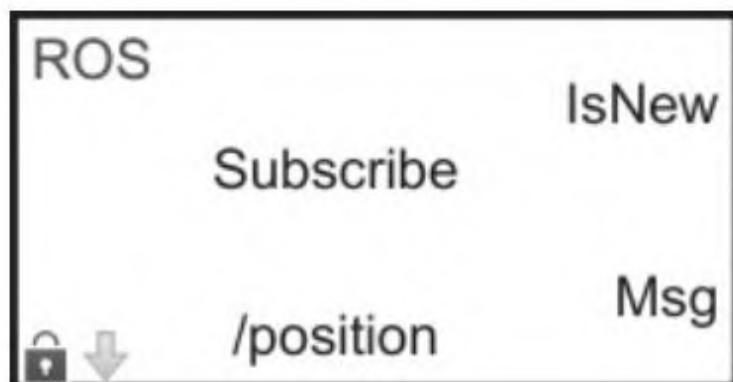


图13-20 Simulink订阅者模块

在发布者模型中，如果我们使用总线创建者在一条消息中聚合多个数据，那么我们就需要拆分消息的数据。为此，我们将使用具有一个输入和两个输出的总线选择模块：twist消息的线性部分的X和Y字段。要创建该模块，请对其进行配置，以使所选信号仅包含twist消息的Linear. X和Linear. Y部分，如图13-21所示。

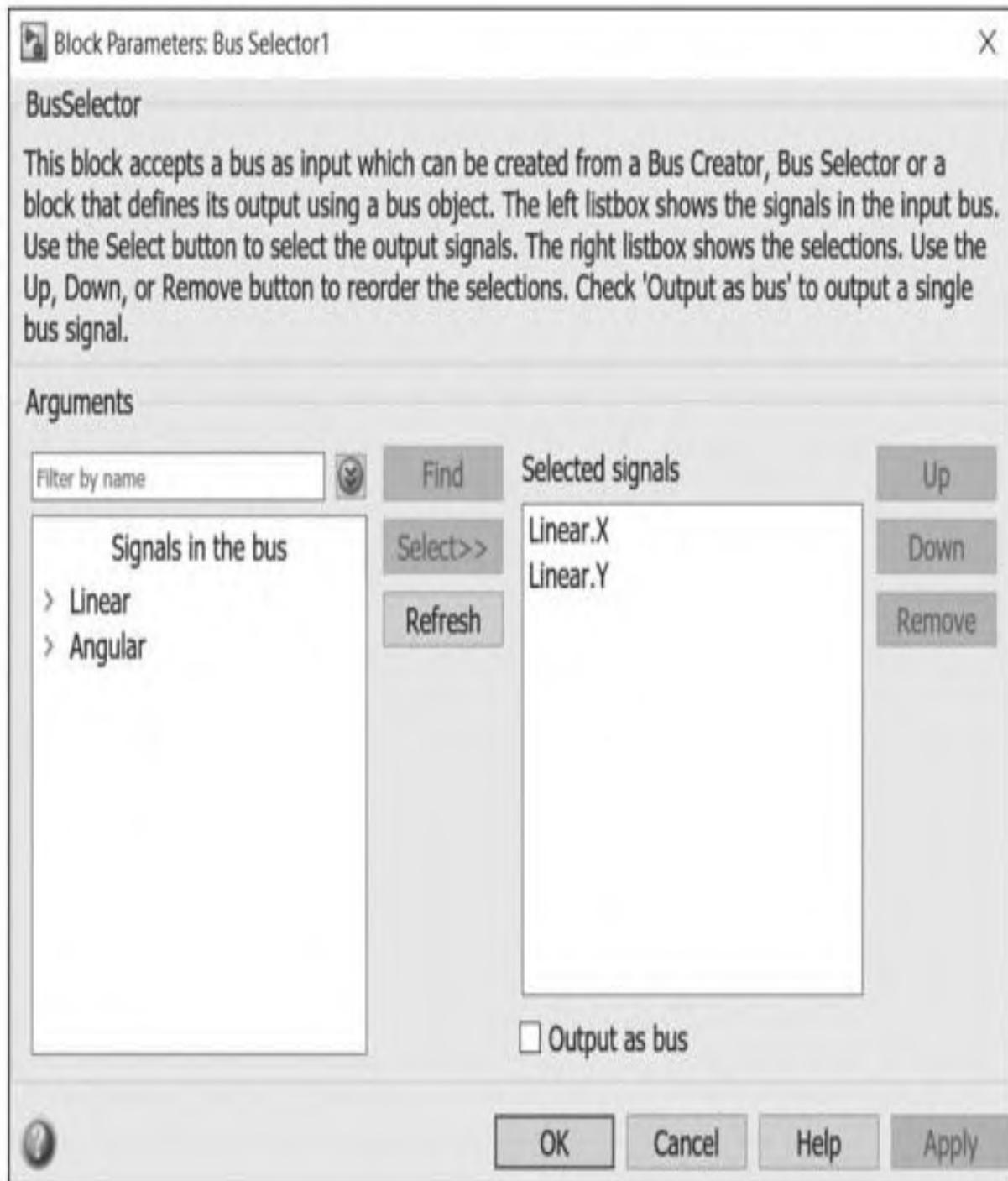


图 13-21 总线选择模块

在我们的实现中，我们将总线选择器包含在子系统中，这是另一种可以启用/禁用使能端口的模块。通过这种方式，我们可以将订阅者模块的 IsNew 字段链接到子系统，并仅在收到新消息时启用其输出。要

探索子系统的内容，只需要双击它就足够了，就像任何其他模块一样。最后，我们可以添加两个示波器模块来绘制子系统的输出。最终的连接模型如图13-22所示。

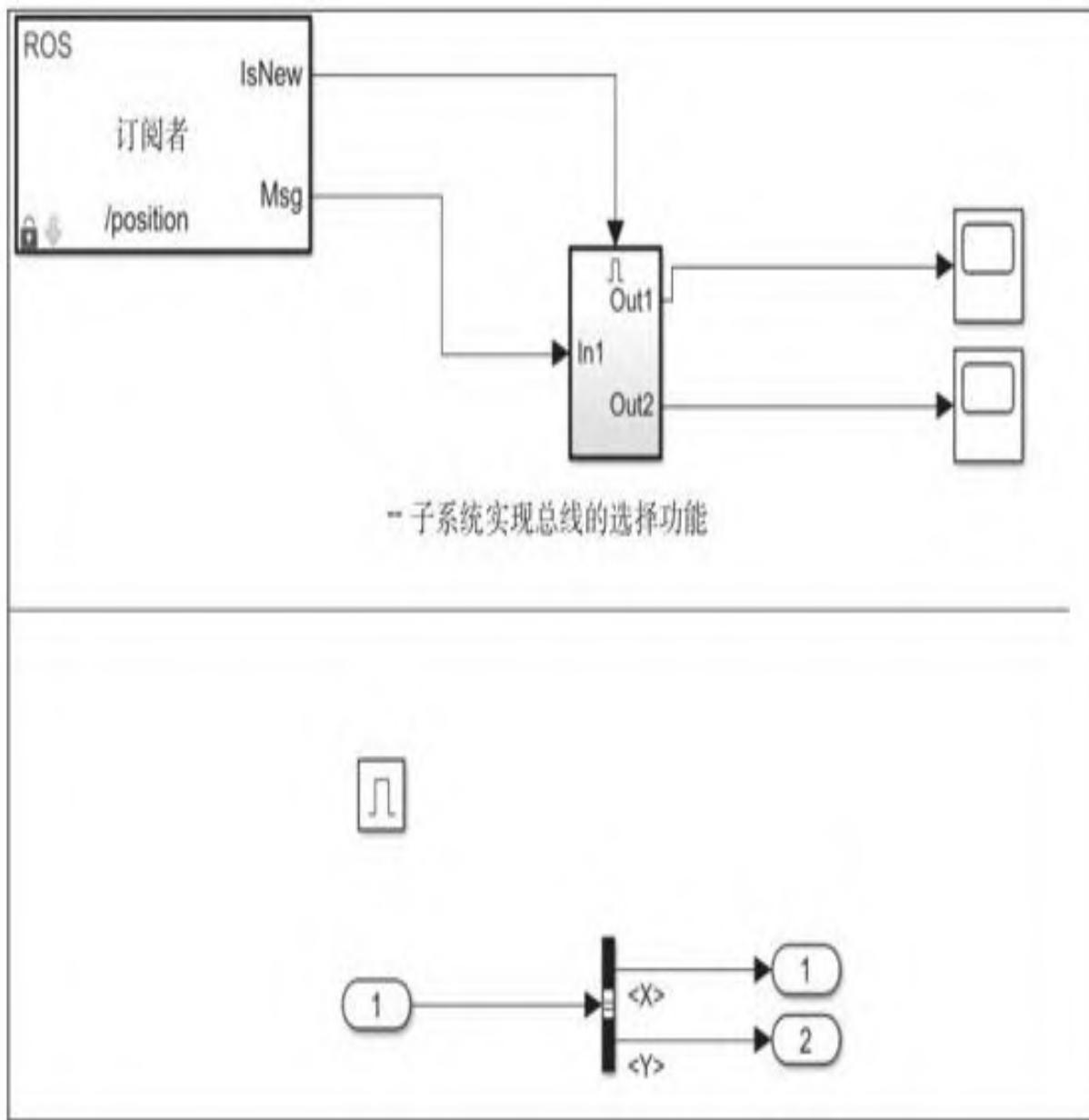


图13-22 订阅者系统模型

现在我们可以运行发布者和订阅者系统，并检查示波器模块上的输出。

13.3 用Simulink开发一个简单的控制系统

现在我们已经学会了如何连接Simulink和ROS，我们可以尝试实现一个更加复杂的系统，从而控制真实或仿真机器人。我们将继续使用在Gazebo中仿真的Turtlebot机器人，我们将学习如何控制其方向以使其到达想要到达的位置。换句话说，我们将实现一个控制系统，该系统将使用其里程计来测量得到机器人的方向，将该值与所需方向进行比较并获得方向误差。我们将用PID控制器计算速度，以控制机器人达到最终所需方向，最终将方向误差设置为零。该控制器在Simulink中已经存在了，因此不需要我们自己实现它。下面我们开始讨论我们模型的所有元素。

用/odom消息表示系统的输入，其中包含关于机器人的实际姿态及其速度的信息，以及常数模块，它用来指定Turtlebot的期望方向。

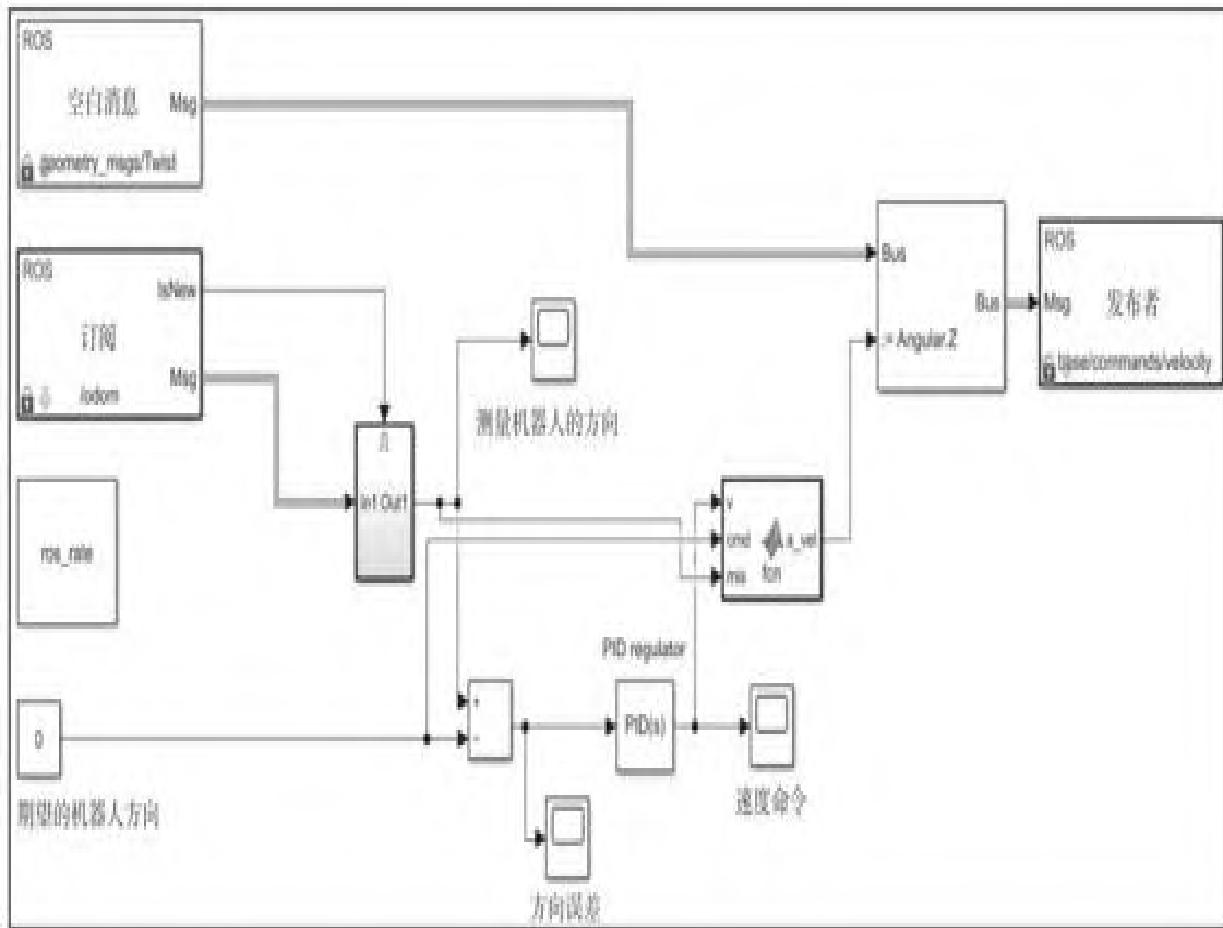


图13-23 Simulink 中的Turtlebot方向控制模型

我们的模型所做的第一件事就是从`/odom`消息中估计方向。通过在每个时间步长中对角速度进行积分来估计机器人的朝向。我们用MATLAB的功能模块控制`/odom`消息的速度值以丢弃噪声测量值。为了集成速度数据，我们使用Simulink提供的Integrator模块。

同样，我们将该部分包含在子系统中，如图13-24所示。

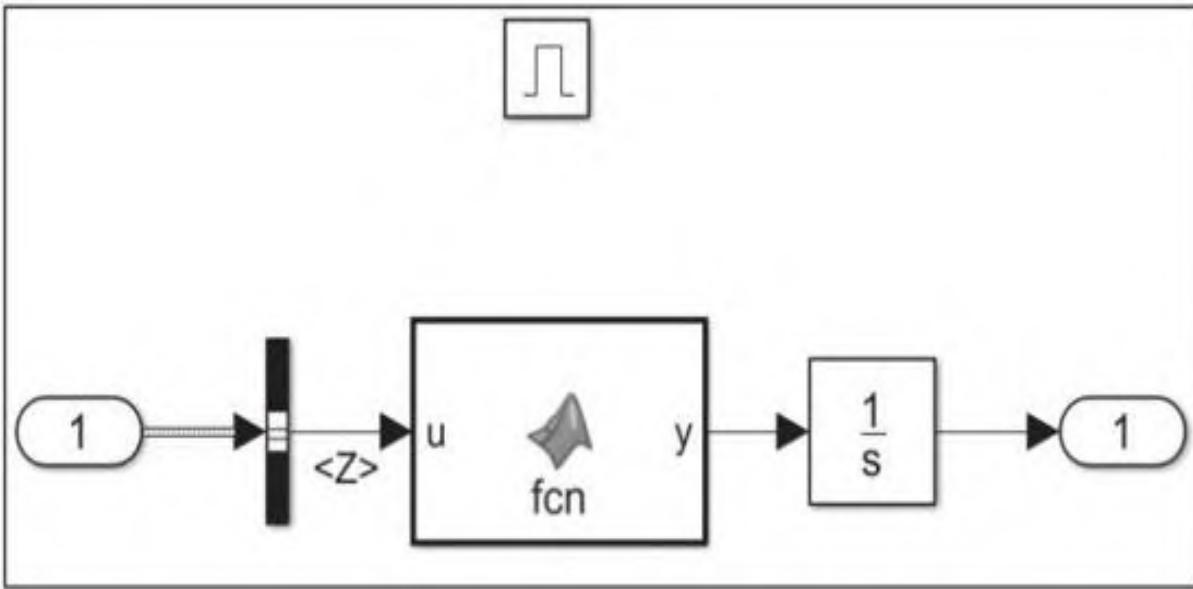


图13-24 MATLAB函数模块

MATLAB功能模块允许开发人员将他们自己的MATLAB函数转换为Simulink模块。本例中，函数代码如下：

```
function y = fcn(u)
    y = 0.0;
    if abs( u ) > 0.01
        y = u;
    end
end
```

我们从接收到的twist消息中提取Angular. Z值，该值指定了相对于Z轴的角速度，表示机器人的旋转方向。我们考虑的噪声低于0.01 rad/s。

现在我们知道了如何旋转机器人，我们可以通过使用Simulink的求和（sum）模块来考虑期望的方向以计算方向误差。为了更改为期望方向，我们可以双击模块来配置其参数。

最后，我们可以实现我们的机器人控制器。对于scope模块，我们将使用PID控制器，这是最常用的带反馈的控制回路机制之一。这种控

制器广泛应用于工业和大学中，它适用于各种应用。它不断尝试最小化输入误差，应用基于比例、积分和微分项的控制输出。在模型中拖放该控制器后，它对输入数据的响应将取决于可以从模块属性中正确调整的P、I和D项（增益）。最后，我们必须在/base/commands/velocity话题上发布由PID控制器生成的数据，并在Gazebo仿真中驱动机器人。和平时一样，我们可以在开始仿真后查看scope模块是如何降低误差的。

在应用计算出来的速度之前，我们使用另一个MATLAB功能模块来设置速度的正负。实际上，考虑到速度的正负，机器人将在两个不同的方向上旋转：负速度将使机器人沿顺时针方向旋转，而正速度将使机器人沿逆时针方向旋转。在我们的例子中，我们希望选择能够使机器人更快地朝其前进方向转动的方向：

```
function a_vel = fcn(v, cmd, mis)
    a_vel = 0;
    if (mis < cmd )
        a_vel = abs(v);
    elseif ( mis > cmd )
        a_vel = -abs(v);
    end
end
```

该功能模块接收的输入包括计算出的速度、控制命令和驱动机器人的方向。当测量的方向低于指令方向时，机器人必须沿顺时针方向旋转，否则沿逆时针方向旋转。

配置Simulink模型

现在，我们的模型已经完全连接好了，我们只需要配置和仿真它。首先，我们需要导入ros_rate模块以同步Simulink仿真。在这种情况下，更高的帧速率将保证更好的行为，因此你可以双击ros_rate模块并将速率设置为100Hz。然后，从模型窗口的主菜单中点击 Simulation|Model Configuration Parameters来打开Model Configuration Parameters，或者直接使用快捷键Ctrl+E。建议使用

Fixed-step size指定想要的步长（我们可以使用0.01秒），如图13-25所示。

现在，模型已经配置完成，我们可以进行仿真了。正如上一个例子，使用下面的命令来启动Turtlebot仿真：

```
$ rosrun turtlebot_gazebo turtlebot_world.launch
```

然后按下播放按钮开始Simulink仿真。在Gazebo中，你将看到机器人在尝试移动到指定的方向上，与此同时，你也可以看到scope面板监视方向误差和生成的速度指令，如图13-26所示。



图13-25 模型配置参数

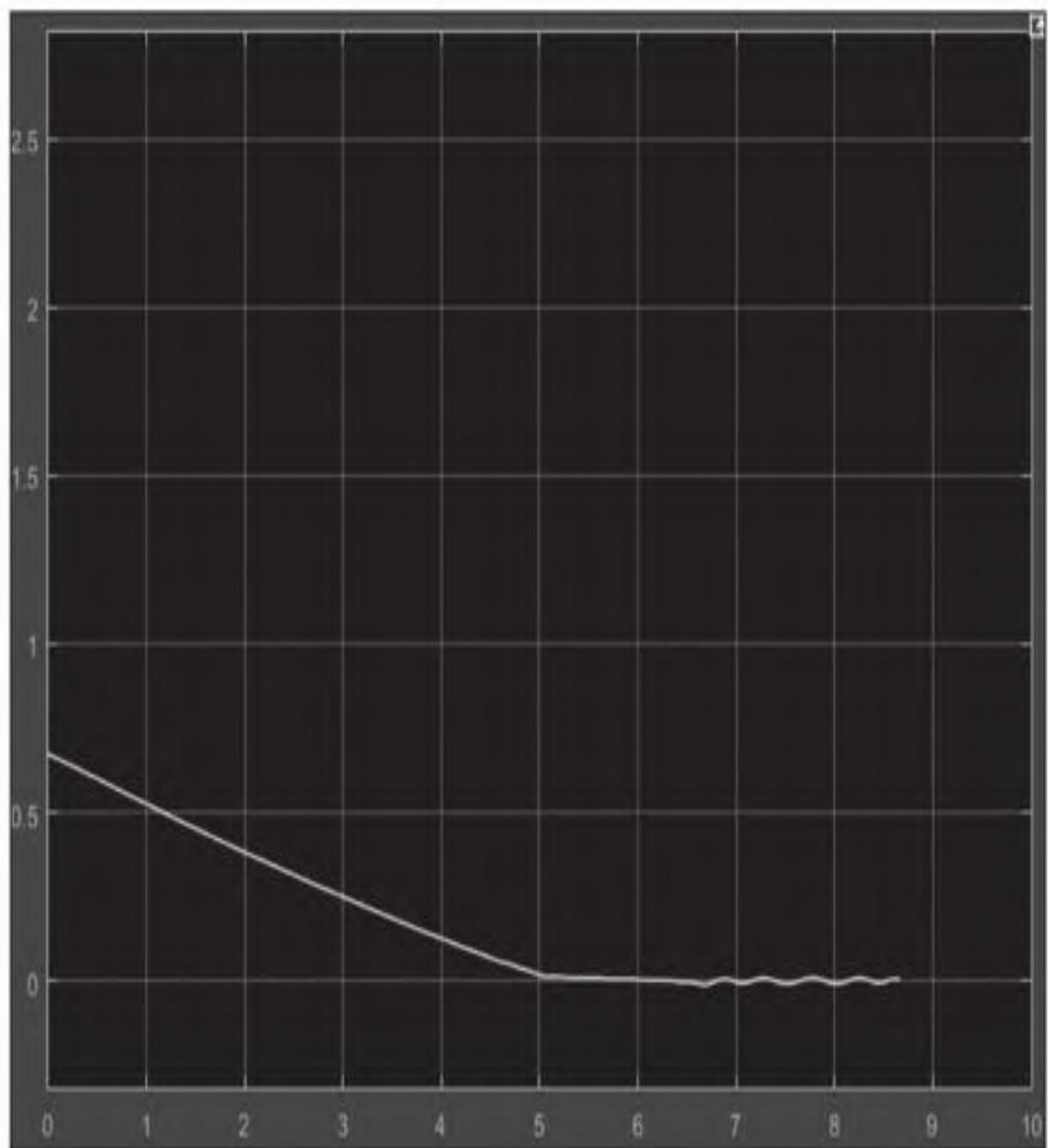


图13-26 方向误差变化

13.4 习题

- MATLAB和机器人系统工具箱是什么？
- 如何将MATLAB与ROS网络连接起来？
- 为什么MATLAB可用于开发机器人应用？
- Simulink是什么？
- PID控制器是什么，如何使用Simulink实现它？

13.5 本章小结

在本章，我们学习了如何使用MATLAB开发简单或复杂的机器人应用，以及如何将MATLAB与运行在同一台计算机上的或运行于ROS网络的ROS节点连接。我们讨论了如何处理MATLAB中的话题，通过复用MATLAB工具箱中已有的功能，为差速驱动机器人开发一个简单的避障系统。然后，我们介绍了Simulink，这是一个基于图形的程序编辑器，它允许开发人员来实现、仿真和验证他们的动态系统模型。我们学习了如何在ROS网络中获取和设置数据，以及如何开发一个控制Turtlebot机器人方向的简单控制系统。下一章，我们将介绍ROS-Industrial（一个将工业机器人操作器连接到ROS的ROS软件包），以及如何用ROS的强大功能（如MoveIt!、Gazebo和RViz等）来控制它。

第14章

ROS与工业机器人

到目前为止，我们主要讨论的是ROS与个人机器人和研究型机器人的接口，但是机器人被大量应用的主要领域之一是工业制造。ROS支持工业机器人吗？是否有公司用ROS解决制造工艺流程中的问题？ROS-Industrial软件包利用其功能强大的工具集，如MoveIt!、Gazebo、RViz，为连接并控制工业机器人提供了解决方案。

本章将介绍以下内容：

- 认识并开展使用ROS-Industrial
- 为工业机器人创建URDF文件并将其与MoveIt!连接
- 使用UR机械臂和ABB机械臂的MoveIt! 配置
- 了解ROS-Industrial机器人支持的软件包
- 了解ROS-Industrial机器人客户端和驱动软件包
- 使用IKfast算法和MoveIt! IKFast插件
- 让我们从ROS-Industrial的简要概述开始学习。

14.1 理解ROS-Industrial软件包

ROS-Industrial（图14-1）是将ROS软件的很多高级功能扩展到制造流程中的工业机器人。ROS-Industrial包含许多软件包，用于帮助我们控制工业机器人。这些软件包是基于BSD（legacy）/Apache 2.0 许可授权的，其中包含适用于工业硬件的标准解决方案的软件库、驱动程序和工具。ROS-Industrial现在由ROS-Industrial联盟负责规划发展。ROS-Industrial(ROS-I)的官方网站为
<http://rosindustrial.org/>。



图14-1 ROS-Industrial的标志

14.1.1 ROS-Industrial的目标

ROS-Industrial开发的主要目标如下：

- 将ROS的优势与现有的工业技术相结合，进一步探索ROS在制造业中的先进功能。
- 为工业机器人应用开发出可靠而强大的软件。
- 提供一种简单的方法来进行工业机器人的研发。
- 创建由工业机器人研究人员和专业人员支持的庞大社区。
- 提供工业级的ROS应用，并成为工业相关应用的一站式解决方案。

14.1.2 ROS-Industrial简史

2012年，为了在工业制造中使用ROS，由Yaskawa Motoman Robotics (<http://www.motoman.com/>)、Willow Garage (<https://www.willowgarage.com/>) 和SwRI (Southwest Research Institute) (<http://www.swri.org/>) 联合创建了ROS-Industrial开源项目。ROS-I由Shaun Edwards在2012年1月创立。

2013年3月，SwRI领导建立了ROS-I Consortium Americas，德国的Fraunhofer IPA领导建立了ROS-I Consortium Europe。

14.1.3 ROS-Industrial优点

让我们来看看ROS-I为社区带来的好处：

- 探索ROS的功能：ROS-Industrial软件包与ROS框架相关联，因此我们可以在工业机器人中使用ROS的所有功能。使用ROS，我们可以为每个机器人创建自定义的IK解算器，并通过2D/3D感知来实现对物体的操作。
- 即刻能用的应用：ROS接口能使机器人实现高级感知，以便处理拾取和放置复杂对象的操作。
- 简化机器人编程：ROS-I去除了机器人的示教和路径规划功能，而是自动计算给定目标点的无碰撞最佳路径。
- 开源：ROS-I是开源软件，允许商业使用且不带任何附加限制。

14.2 安装ROS-Industrial软件包

可以使用软件包管理器来安装ROS-Industrial软件包或者从源码安装。如果我们已经安装了ros-kinetic-desktop-full，我们就可以用下面的命令在Ubuntu上安装ROS-Industrial软件包：

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

上面的命令将安装ROS-Industrial软件包的核心软件包。
industrial-core软件包集包含下面的ROS软件包：

- industrial-core：该软件包集包含支持工业机器人系统的软件包和库文件。软件包包含了用来与工业机器人控制器和工业机器人仿真器通信的节点，还为工业机器人提供了ROS控制器。
- industrial_deprecated：该软件包包含一些被弃用的节点、启动文件等。该软件包中的文件将在下一个ROS版本中被删除。因此，我们应该在这些内容被删除之前找到这些文件的代替版本。
- industrial_msgs：该软件包包含与ROS-Industrial软件包相关的消息定义。
- simple_message：这是ROS-Industrial软件包的一部分，它是一种标准消息协议，包含用于与工业机器人控制器之间通信的简单消息框架。
- industrial_robot_client：该软件包包含一个通用的机器人客户端，用来连接工业机器人控制器。该控制器上运行了工业机器人服务器并且可

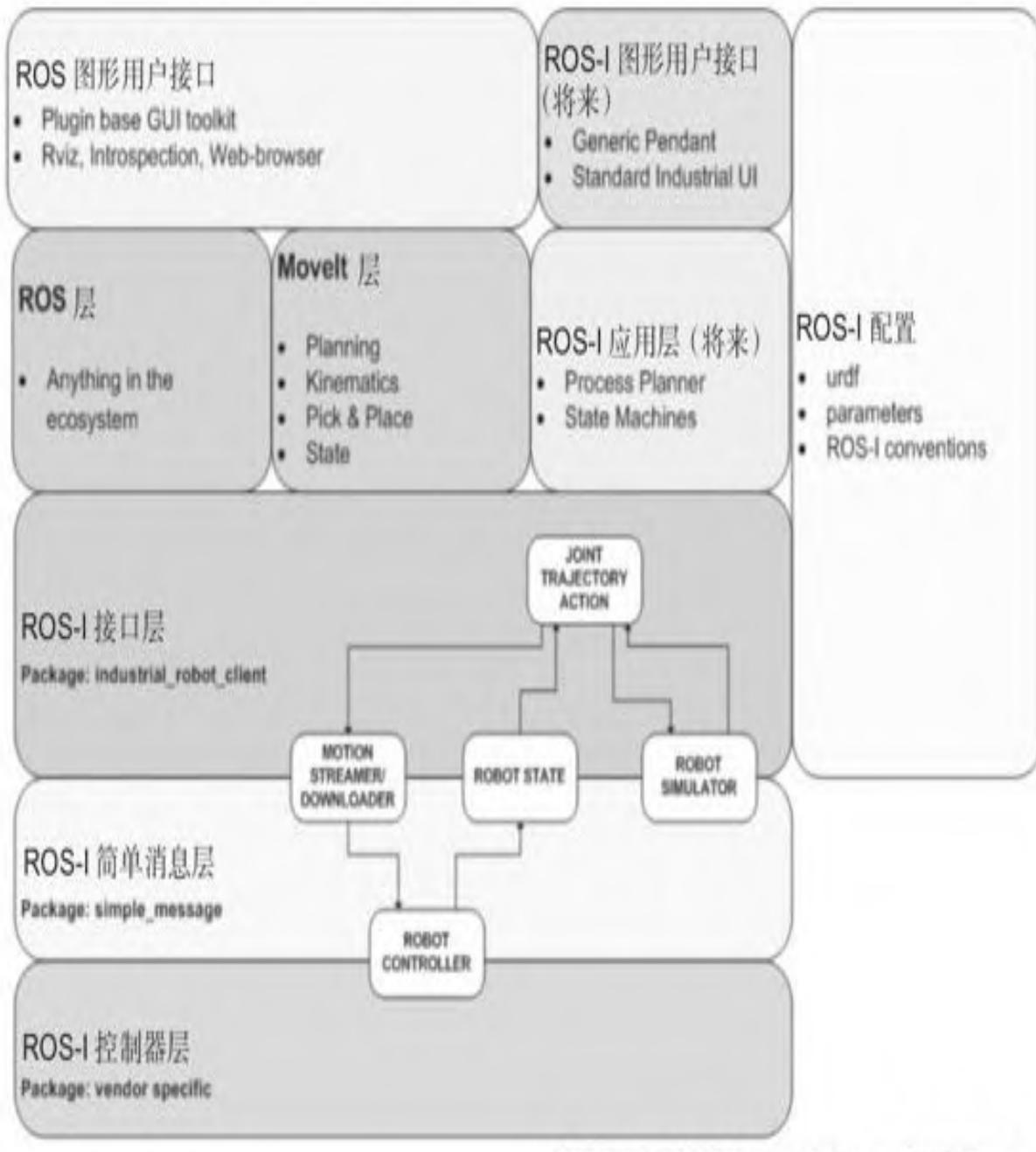
以使用简单的消息协议进行通信。

- `industrial_robot_simulator`: 该软件包仿真了工业机器人控制器，这个控制器遵循ROS-Industrial驱动程序标准。使用该仿真器，我们可以对工业机器人进行仿真和可视化。
- `industrial_trajectory_filters`: 该软件包包含用于过滤轨迹的库和插件。这些库和插件将被发送给机器人控制器。

14.3 ROS-Industrial软件包框图

图14-2是ROS-I软件包的一个简单框图，它们组织在ROS框架上。我们可以在ROS层顶部看到ROS-I层。为了更好地理解，每一层都有一个简要的描述。图14-2取自ROS-I维基网页（<http://wiki.ros.org/Industrial>）：

- ROSGUI：该层包含基于ROS插件的GUI工具层，该工具层由诸如RViz、rqt_gui等工具组成。
- ROS-I GUI：这些GUI是与工业机器人交互的标准工业界面，它们可能在将来被实现。
- ROSLayer：该层是所有通信发生的基础层。
- MoveIt Layer：MoveIt!层为工业机械臂在运动规划、运动学和拾取与放置方面提供了直接的解决方案。
- ROS-I Application Layer：该层由工业过程规划器组成，用来规划制造什么，如何制造以及制造过程中需要哪些资源。
- ROS-I Interface Layer：该层由工业机器人客户端组成，它可以使用简单的消息协议与工业机器人控制器相连接。
- ROS-I Simple Message Layer：该层是工业机器人的通信层，它是一组标准协议，用于将数据从机器人客户端发送到控制器，反之亦然。
- ROS-I Controller Layer：该层由供应商特定的工业机器人控制器组成。



ROS-Industrial High Level Architecture - Rev 0.02.vsd

图14-2 ROS-Industrial框图

讨论了基本概念之后，我们开始使用ROS-Industrial将工业机器人与ROS连接起来。首先，我们将展示如何创建工业机器人的URDF模型以及如何为其创建合适的MoveIt!配置。然后，我们将讨论如何控制真实的和仿真的UR和ABB工业机械臂，并分析ROS-I包的所有必要元素。最后，我们将用IKfast算法和插件加速MoveIt!的运动学计算过程。

14.4 为工业机器人创建URDF

为普通机器人和工业机器人创建URDF文件的方法是相同的，但是工业机器人在URDF建模时需要严格遵守一些标准。这些标准如下所示：

- 简化URDF设计：URDF文件应该是简单易读的，并且只需要重要的标签。
- 设计通用化：为不同供应商的工业机器人开发通用的设计公式。
- URDF模块化：URDF需要使用XACRO宏进行模块化设计，使其可以很容易地包含在大型的URDF文件中。

下面是在ROS-I中设计URDF时的主要区别：

- 碰撞感知：工业机器人IK规划器具有碰撞感知功能，因此URDF应该包含每个连杆精确的碰撞3D网格模型。机器人中的每个连杆都需要用合适的坐标系导出STL或DAE格式的模型。ROS-I遵循的坐标系是x轴指向前方，当每个关节处于零位置时，z轴指向上方。还需注意的是，如果关节的原点与机器人的基座重合，变换将更简单。如果我们把机器人的关节置于零位置（原点），这样可以简化机器人的设计。在ROS-I中，用于视觉的网格模型文件要求是高精度的，但是用于碰撞的网格文件不要求高度精确，因为越精确就需要更多的时间执行碰撞检测。为了删除网格模型细节，我们可以使用像MeshLab (<http://meshlab.sourceforge.net/>) 之类的工具，使用选项(Filters -> Remeshing,Simplification and Reconstruction -> Convex Hull) 进行调整。

- URDF关节约束：每个机器人关节的方向仅限于单个旋转，即在三个方向（横滚、俯仰和偏航）值中只能有一个值。
- Xacro宏：在ROS-I中，整个执行器部分都用xacro写成宏。我们可以在另一个宏文件中添加这个宏的实例，该文件可用于生成URDF文件。我们还可以在同一个文件中包含其他末端执行器的定义。
- 标准框架：在ROS-I中，base_link应该是第一个连杆，并且tool0 (tool-zero) 应该是末端执行器的连杆。此外，base坐标系应该与机器人控制器的base坐标系匹配。在多数情况下，从base到base_link的变换被认为是固定的。

为工业机器人构建xacro文件后，我们可以将其转换为URDF文件并用下面的命令对其进行验证：

```
$ rosrun xacro xacro -inorder -o <output_urdf_file> <input_xacro_file>
$ check_urdf <urdf_file>
```

接下来，我们将讨论为工业机器人创建MoveIt!配置时的差异。

14.5 为工业机器人创建MoveIt!配置

除了某些标准约定外，为工业机器人创建MoveIt!配置的过程与创建其他普通机器人是一样的。下面的过程清楚地概述了这些标准约定：

- 1) 使用下面的命令启动MoveIt!配置助手：

```
$ rosrun moveit_setup_assistant setup_assistant.launch
```

- 2) 从机器人描述文件夹中加载URDF，或者将xacro转换为URDF并加载到配置助手。

- 3) 创建一个自碰撞（Self-Collisions）矩阵，采样密度约为80 000。该值可以增加机械臂的碰撞检测能力。

- 4) 增加虚拟关节（Virtual Joints）矩阵，如图14-3所示。这里的虚拟关节名和父框架的名称可以是任意的。

- 5) 在下一步中，我们将增加操作器（manipulator）和末端执行器（endeffector）的规划组（Planning Groups）。这里的组名也可以是任意的。默认的插件是KDL，即使在完成MoveIt!配置后我们也可以更改它，如图14-4所示。

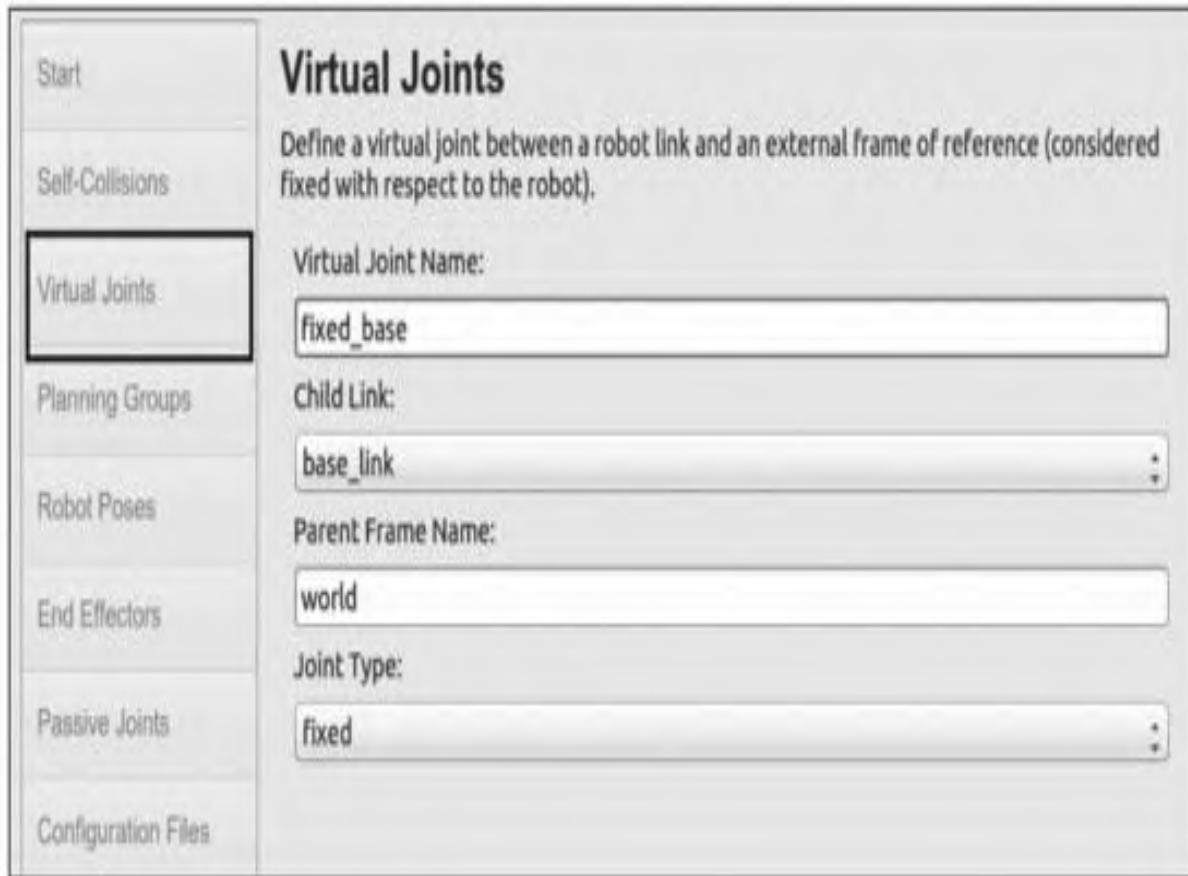


图14-3 增加MoveIt!-虚拟关节

Start
Self-Collisions
Virtual Joints
Planning Groups
Robot Poses
End Effectors
Passive Joints
Configuration Files

Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

Edit Planning Group 'manipulator'

Group Name:	<input type="text" value="manipulator"/>
Kinematic Solver:	<input type="text" value="kdl_kinematics_plugin/KDLKinematicsPlugin"/> :
Kin. Search Resolution:	<input type="text" value="0.005"/>
Kin. Search Timeout (sec):	<input type="text" value="0.005"/>
Kin. Solver Attempts:	<input type="text" value="3"/>

Start
Self-Collisions
Virtual Joints
Planning Groups
Robot Poses
End Effectors
Passive Joints
Configuration Files

Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

Edit Planning Group 'endeffect'or'

Group Name:	<input type="text" value="endeffector"/>
Kinematic Solver:	<input type="text" value="None"/> :
Kin. Search Resolution:	<input type="text" value="0.005"/>
Kin. Search Timeout (sec):	<input type="text" value="0.005"/>
Kin. Solver Attempts:	<input type="text" value="3"/>

图14-4 在MoveIt!中创建规划组

6) 这是规划组，即操作器加上末端执行器的配置，如图14-5所示。

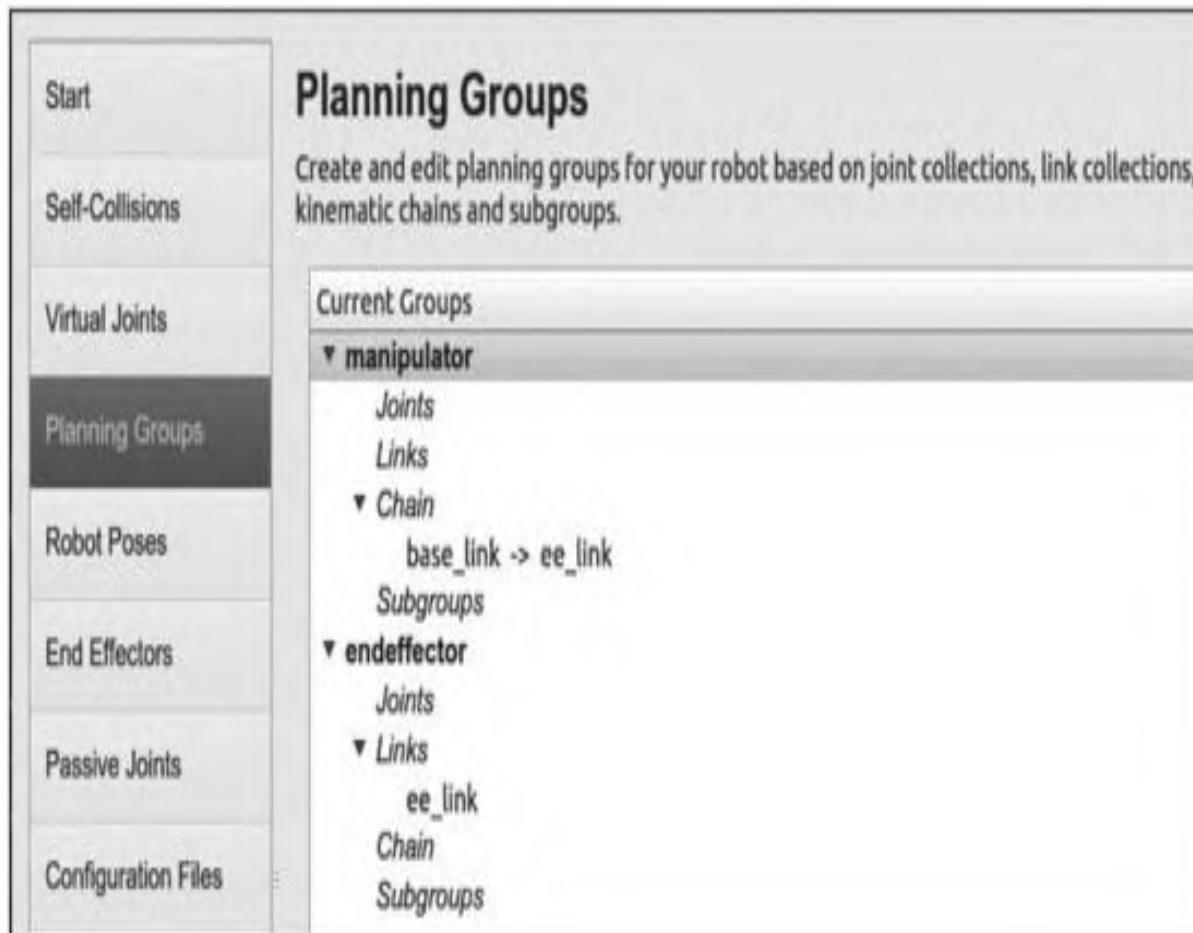


图14-5 在MoveIt!中规划操作器和末端执行器组

7) 我们可以指定机器人姿态，如初始姿态、直立姿态等。这个设置是可选的。

8) 我们可以指定末端执行器（End Effectors），如图14-6所示，这也是一个可选设置。

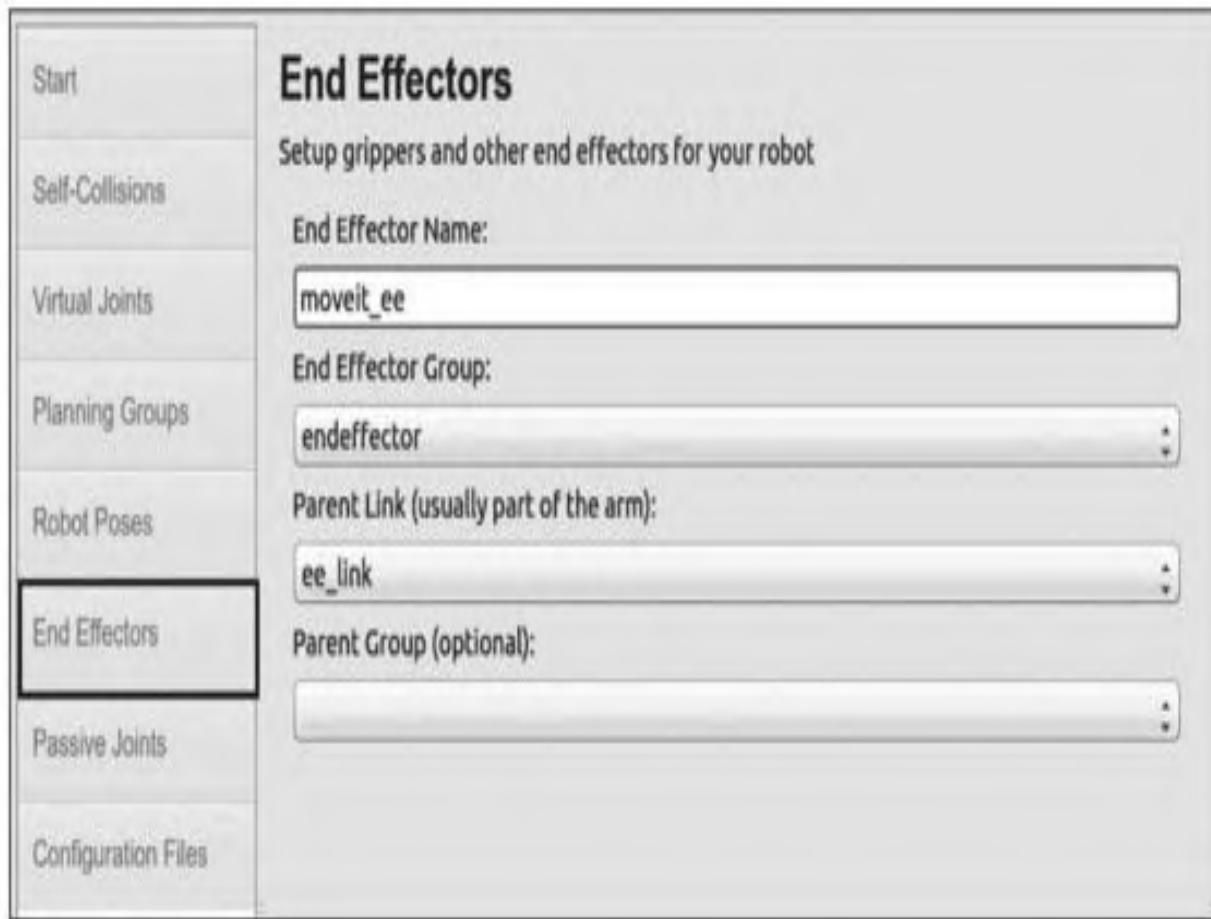


图14-6 在MoveIt!配置助手中设置末端执行器

9) 设置了末端执行器后，我们可以直接生成配置文件。需要注意的是moveit-config软件包应该命名为<robot_name>_moveit_config，其中robot_name是URDF文件的名称。此外，如果我们想将生成的配置软件包移动到另一台电脑上，我们需要编辑setup_assistant文件。该文件位于moveit软件包中。我们应该将绝对路径改为相对路径。下面是abb_irb2400机器人的示例。我们应该在这个文件中使用URDF和SRDF的相对路径，如下所示：

```
moveit_setup_assistant_config:  
    URDF:  
        package: abb_irb2400_support  
        relative_path: urdf/irb2400.urdf  
    SRDF:  
        relative_path: config/abb_irb2400.srdf  
CONFIG:  
    generated_timestamp: 1402076252
```

14.5.1 更新MoveIt!配置文件

创建了MoveIt!配置后，我们需要更新MoveIt!软件包内config文件夹中的controllers.yaml文件。下面是controllers.yaml的示例：

```
controller_list:
- name: ""
  action_ns: follow_joint_trajectory
  type: FollowJointTrajectory
  joints:
    - shoulder_pan_joint
    - shoulder_lift_joint
    - elbow_joint
    - wrist_1_joint
    - wrist_2_joint
    - wrist_3_joint
```

我们还应该更新关于关节信息的joint_limits.yaml文件。下面是joint_limit.yaml的代码片段：

```
joint_limits:
  shoulder_pan_joint:
    has_velocity_limits: true
    max_velocity: 2.16
    has_acceleration_limits: true
    max_acceleration: 2.16
```

我们还可以通过编辑kinematics.yaml文件来更改运动解算器插件。编辑完所有配置文件后，我们需要编辑controller manager launch文件

(<robot>_moveit_config/launch/<robot>_moveit_controller_manager.launch)。

这里是controller manager.launch文件的示例：

```
<launch>
  <rosparam file="$(find ur10_moveit_config)/config/controllers.yaml"/>
  <param name="use_controller_manager" value="false"/>
  <param name="trajectory_execution/execution_duration_monitoring"
value="false"/>
  <param name="moveit_controller_manager" value=
"moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
</launch>
```

创建了控制管理器后，我们需要创建<robot>_moveit_planning_execution.launch文件。下面是该文件的一个示例：

```
<launch>
  <arg name="sim" default="false" />
  <arg name="limited" default="false"/>
  <arg name="debug" default="false" />

  <!-- Remap follow_joint_trajectory -->
  <remap if="$(arg sim)" from="/follow_joint_trajectory"
    to="/arm_controller/follow_joint_trajectory"/>

  <!-- Launch moveit -->
  <include file="$(find ur10_moveit_config)/launch/move_group.launch">
    <arg name="limited" default="$(arg limited)"/>
    <arg name="debug" default="$(arg debug)" />
  </include>
</launch>
```

14.5.2 测试MoveIt!配置

编辑好MoveIt!中的配置和启动文件后，我们就可以启动机器人仿真来检查MoveIt!配置是否工作正常。前提是需要保证ros-industrial-simulator软件包已正确安装。下面是测试一个工业机器人的步骤：

- 1) 启动机器人仿真器。
- 2) 使用如下命令启动MoveIt!的规划执行启动文件：

```
$ rosrun roslaunch <robot>_moveit_config  
moveit_planning_execution.launch
```

- 3) 打开RViz，使用Plan and Execute按钮加载RViz运动规划插件。我们可以在仿真机器人上进行运动轨迹的规划和执行。

14.6 安装UR机械臂的ROS-Industrial软件包

UR机器人（Universal Robots, <http://www.universal-robots.com/>）是一家位于丹麦的工业机器人制造商。该公司主要生产3款机器人：UR3、UR5和UR10。机器人如图14-7所示。

这些机器人的规格如下表所示：

机器人	UR-3	UR-5	UR-10
工作半径	500 mm	850 mm	1300 mm
负载	3 kg	5 kg	10 kg
重量	11 kg	18.4 kg	28.9 kg
占地半径	118 mm	149 mm	190 mm

在下一节中，我们将安装UR机器人软件包并用MoveIt!接口在Gazebo中对工业机器人进行仿真。



图14-7 UR-3、UR-5和UR-10机器人

安装UR机器人的ROS接口

我们可以用Ubuntu/Debian软件包管理器安装UR机器人软件包：

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

或者，我们可以直接从下面的代码库中下载这些软件包：

```
$ git clone https://github.com/ros-industrial/universal_robot.git
```

UR机器人软件包包括：

- ur_description: 该软件包包含了UR-3、UR-5和UR-10的机器人描述和Gazebo描述。
- ur_driver: 该软件包包含了客户端节点，它可与UR-3、UR-5和UR-10机器人的硬件控制器通信。
- ur_bringup: 该软件包由启动文件组成。它用来启动与机器人硬件控制器的通信，以使真实机器人启动。
- ur_gazebo: 该软件包包含了UR-3、UR-5和UR-10的Gazebo仿真。
- ur_msgs: 该软件包包含了用于不同UR节点之间通信的ROS消息。
- urXX_moveit_config: 这些是UR机器人操作器的moveit配置文件。每种类型的机械臂都有一个软件包 (ur3_moveit_config、ur5_moveit_config 和ur10_moveit_config) 。
- ur_kinematics: 该软件包包含了用于UR-3、UR-5和UR-10的运动解算器插件。我们可以在MoveIt!中使用该解算器插件。

安装和编译UR机器人软件包后，我们就可以用下面的命令启动Gazebo中的UR-10机器人进行仿真，如图14-8所示。

```
$ rosrun ur_gazebo ur10.launch
```

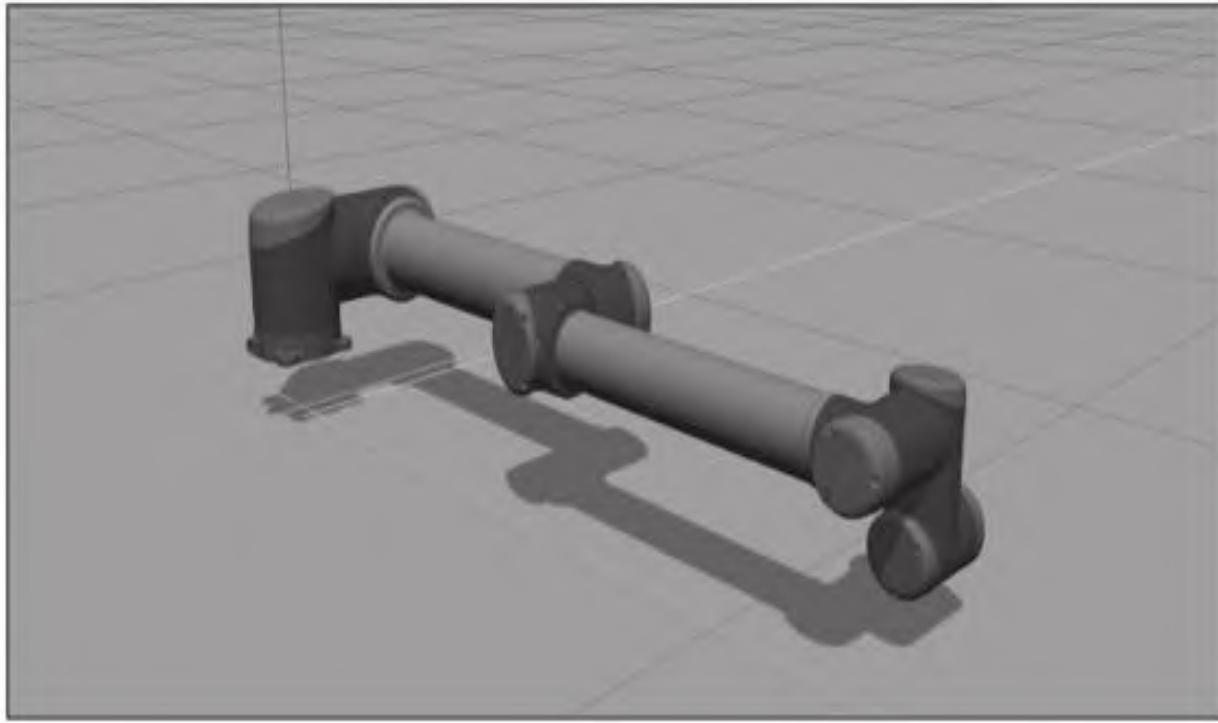


图14-8 Gazebo中的UR-10仿真

我们可以看到用于连接MoveIt!软件包的机器人控制器配置文件。下面的YAML文件定义了JointTrajectory控制器。它位于ur_gazebo/controller文件夹中，文件名称为arm_controller_ur10.yaml：

```
arm_controller:  
  type: position_controllers/JointTrajectoryController  
  joints:  
    - shoulder_pan_joint  
    - shoulder_lift_joint  
    - elbow_joint  
    - wrist_1_joint  
    - wrist_2_joint  
    - wrist_3_joint  
  constraints:  
    goal_time: 0.6  
    stopped_velocity_tolerance: 0.05  
    shoulder_pan_joint: {trajectory: 0.1, goal: 0.1}  
    shoulder_lift_joint: {trajectory: 0.1, goal: 0.1}  
    elbow_joint: {trajectory: 0.1, goal: 0.1}  
    wrist_1_joint: {trajectory: 0.1, goal: 0.1}  
    wrist_2_joint: {trajectory: 0.1, goal: 0.1}  
    wrist_3_joint: {trajectory: 0.1, goal: 0.1}  
  stop_trajectory_duration: 0.5  
  state_publish_rate: 25  
  action_monitor_rate: 10
```

14.7 理解UR机械臂的MoveIt!配置

UR机械臂的MoveIt!配置位于每个moveit_config软件包的config文件夹下（例如，UR-10的配置是ur10_moveit_config）。

下面是UR-10的controller.yaml的定义：

```
controller_list:
- name: ""

action_ns: follow_joint_trajectory
type: FollowJointTrajectory
joints:
- shoulder_pan_joint
- shoulder_lift_joint
- elbow_joint
- wrist_1_joint
- wrist_2_joint
- wrist_3_joint
```

在同样的路径下，我们可以找到运动配置文件：kinematics.yaml。该文件指定了用于机械臂的IK解算器。UR-10机器人的运动配置文件的内容如下所示：

```
#manipulator:  
#  kinematics_solver: ur_kinematics/UR10KinematicsPlugin  
#  kinematics_solver_search_resolution: 0.005  
#  kinematics_solver_timeout: 0.005  
#  kinematics_solver_attempts: 3  
manipulator:  
    kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin  
    kinematics_solver_search_resolution: 0.005  
    kinematics_solver_timeout: 0.005  
    kinematics_solver_attempts: 3
```

在launch文件夹中的ur10_moveit_controller_manager.launch的定义如下。该启动文件负责加载轨迹控制器配置和启动轨迹控制管理器：

```
<launch>  
  <rosparam file="$(find ur10_moveit_config)/config/controllers.yaml"/>  
  <param name="use_controller_manager" value="false"/>  
  <param name="trajectory_execution/execution_duration_monitoring"  
  value="false"/>  
  <param name="moveit_controller_manager"  
  value="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>  
</launch>
```

讨论了这些文件后，下面让我们看看如何用MoveIt!进行运动规划，并用Gazebo进行仿真：

1) 启动带有关节轨迹控制器的UR-10仿真：

```
$ rosrun ur_gazebo ur10.launch
```

2) 启动MoveIt!节点来进行运动规划。我们可以通过设置
sim:=true，仅在仿真器中测试MoveIt!：

```
$ rosrun ur10_moveit_config  
ur10_moveit_planning_execution.launch sim:=true
```

3) 使用MoveIt!可视化插件启动RViz，如图14-9所示：

```
$ rosrun ur10_moveit_config moveit_rviz.launch config:=true
```

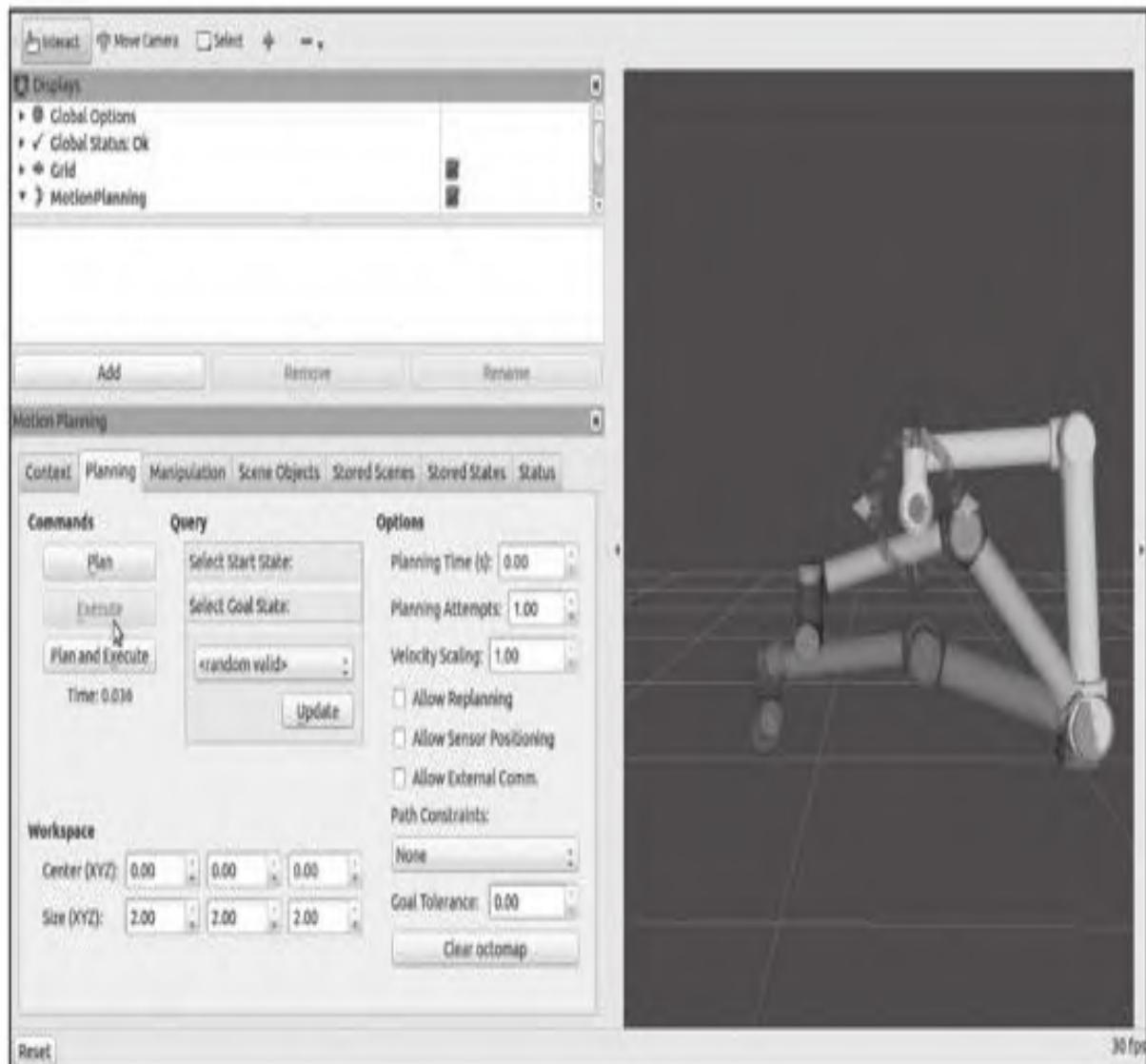


图14-9 RViz中UR-10的运动规划

我们可以移动机器人末端执行器的位置并用Plan按钮进行路径规划。当我们按下Execute按钮或Plan and Execute按钮时，轨迹将被发送到仿真机器人中，然后在Gazebo环境中执行运动。

14.8 使用真实的UR机器人和ROS-I

在Gazebo仿真中测试了我们的控制算法后，我们就可以在真实的UR机械臂上执行操作任务。使用仿真机器人和真实机器人执行轨迹的主要区别在于，我们需要启动驱动器来与机械臂控制器通信，进一步设置所需关节位置的驱动器。

UR机械臂的默认驱动程序是随着ROS-I的ur_driver软件包一起发布的。该驱动程序已成功通过v1.5.7至v1.8.2的系统版本测试。UR机器人控制器的最新版本为v3.2，因此ROS-I驱动程序的默认版本可能不完全兼容。对于这些系统的较新版本（v3.x及更高版本），我们建议使用非官方的ur_modern_driver软件包。

从下面的Git库中下载ur_modern_driver：

```
$ git clone ur10_moveit_config ur10_moveit_planning_execution.launch  
sim:=true
```

下载了这个软件包后，我们需要编译工作区才能使用该驱动程序。

下一步是配置UR机器人硬件使其可以从我们的计算机来控制它。首先，我们可以通过其教学器来启用机器人的网络功能。进入Robot->Setup Network Menu菜单，选择与我们的网络兼容的配置。如果你希望机器人具有固定的网络地址，你需要选择Static Address选项并手动输入所需的地址信息。如果你不想设置固定网络地址就选择DHCP选项，然后应用配置。设置好IP地址后，可以通过ping机器人控制器来检查连接状态：

```
$ ping IP_OF_THE_ROBOT
```

如果控制器对ping命令有回复，则表明连接已经成功建立，我们就可以开始控制机器人了。

如果你的UR机器人系统的版本低于v3.x，我们可以通过执行下面的命令来启动它：

```
$ rosrun ur_driver IP_OF_THE_ROBOT [reverse_port:=REVERSE_PORT]
```

将IP_OF_THE_ROBOT替换为分配给机器人控制器的IP地址。然后，我们就可以使用下面的脚本来测试机器人的运动了：

```
$ rosrun ur_driver IP_OF_THE_ROBOT [reverse_port:=REVERSE_PORT]
```

当使用高于v3.x的系统时，我们可以使用ur_modern_driver软件包提供的启动文件：

```
$ rosrun ur_modern_driver ur10_bringup.launch robot_ip:=IP_OF_THE_ROBOT  
[reverse_port:=REVERSE_PORT]
```

下一步是用MoveIt!控制机器人：

```
$ rosrun ur10_moveit_config ur5_moveit_planning_execution.launch
```

```
$ rosrun ur10_moveit_config moveit_rviz.launch config:=true
```

注意，对于某些机器人配置，MoveIt!在完全关节限制下寻找路径时可能会很难。还有另一个对关节限制的较低版本。只需使用启动命令中的参数`limited`即可启动该操作模式：

```
$ roslaunch ur10_moveit_config ur5_moveit_planning_execution.launch  
limited:=true
```

我们已经知道了如何仿真和控制一个UR机器人。在下一节中，我们将使用ABB机器人。

14.9 ABB机器人的MoveIt!配置

我们将使用两种最受欢迎的ABB工业机器人型号：IRB 2400和IRB 6640。图14-10展示了这两款机器人的图片及其规格。

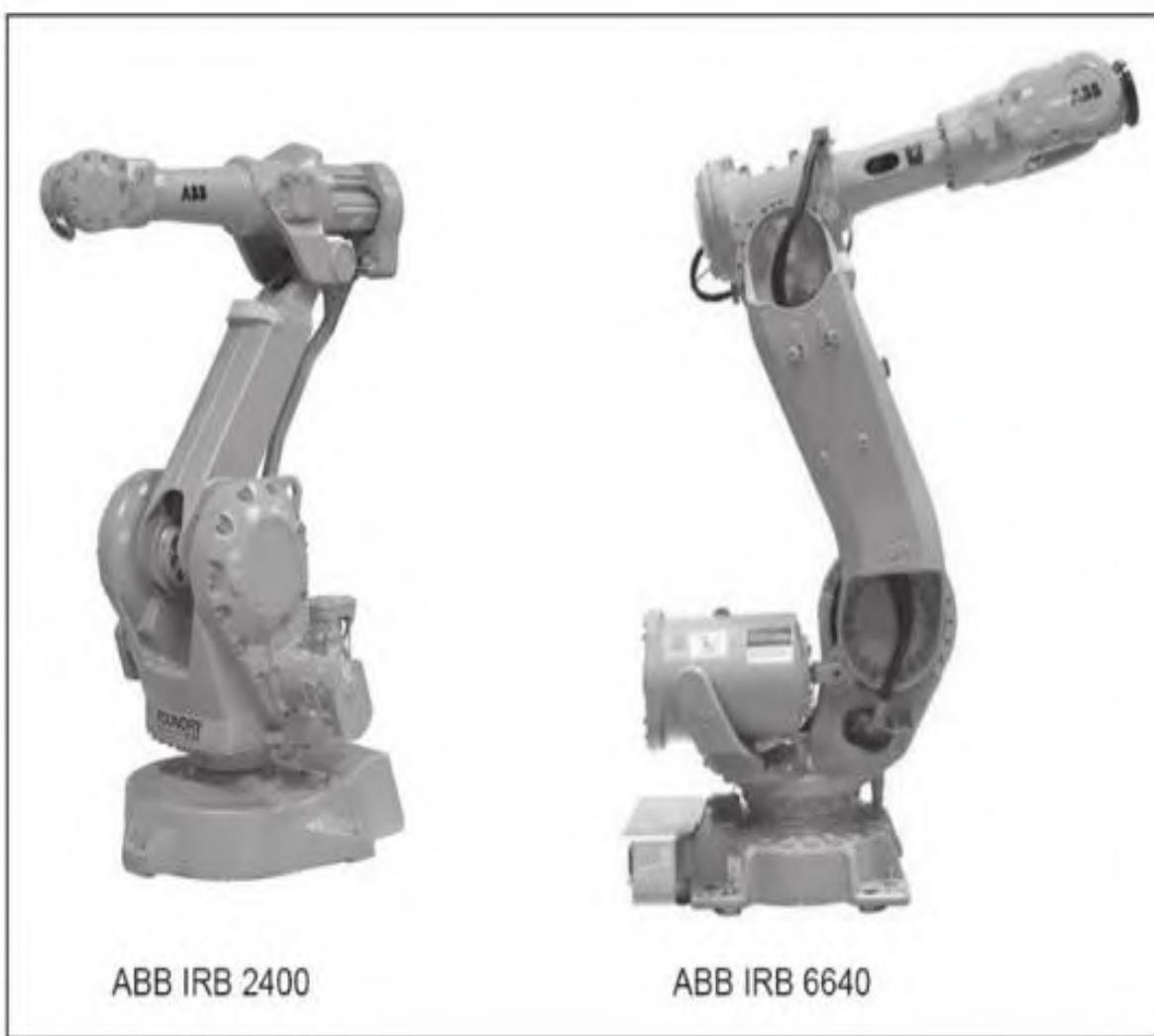


图14-10 ABB IRB 2400和IRB 6640

这两款机械臂的规格如下表所示：

机器人	IRB 2400-10	IRB 6640-130
工作半径	1.55 m	3.2 m
负载	12 kg	130 kg
重量	380 kg	1310 ~ 1405 kg
机器人底座	732 x 600 mm	1107 x 720 mm

要使用ABB软件包，需要将机器人的ROS软件包下载到catkin工作区。我们可以使用下面的命令来执行此任务：

```
$ git clone https://github.com/ros-industrial/abb
```

然后用catkin_make编译软件包。或者我们也可以用Ubuntu/Debian软件包管理系统安装这个软件包。但是为了遵循本章的提示教程，建议在ROS工作区中下载ABB源码。下面的命令将安装完整的ABB机器人软件包：

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

使用如下命令在RViz中启动ABB IRB 6640进行运动规划：

```
$ roslaunch abb_irb6640_moveit_config demo.launch
```

这时RViz窗口将打开，我们就可以在RViz中启动运动规划了，如图14-11所示。

另一个受欢迎的ABB机器人型号是IRB 2400，如图14-12所示。我们可以使用下面的命令在RViz中启动机器人：

```
$ roslaunch abb_irb2400_moveit_config demo.launch
```

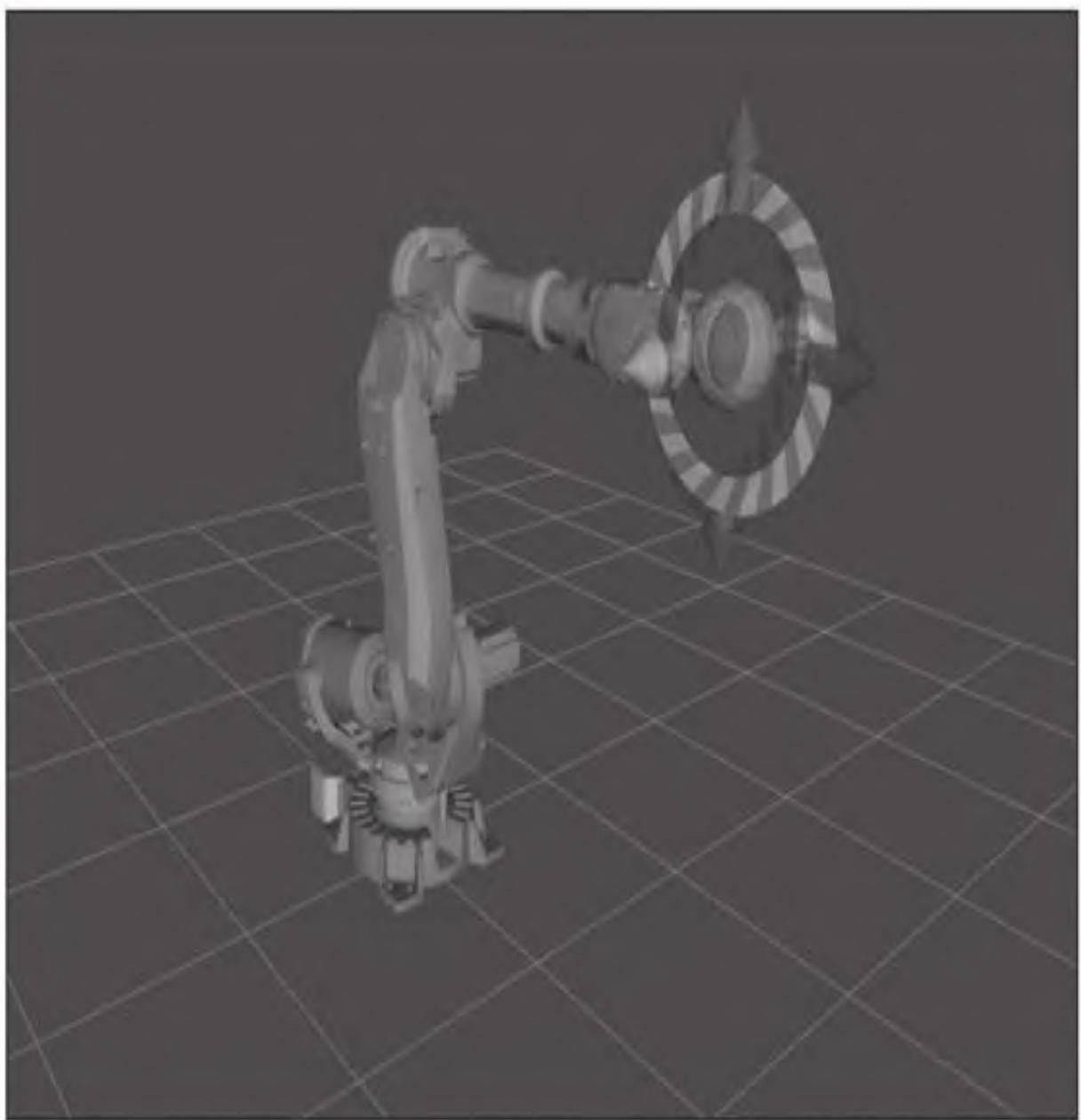


图14-11 ABB IRB 6640的运动规划

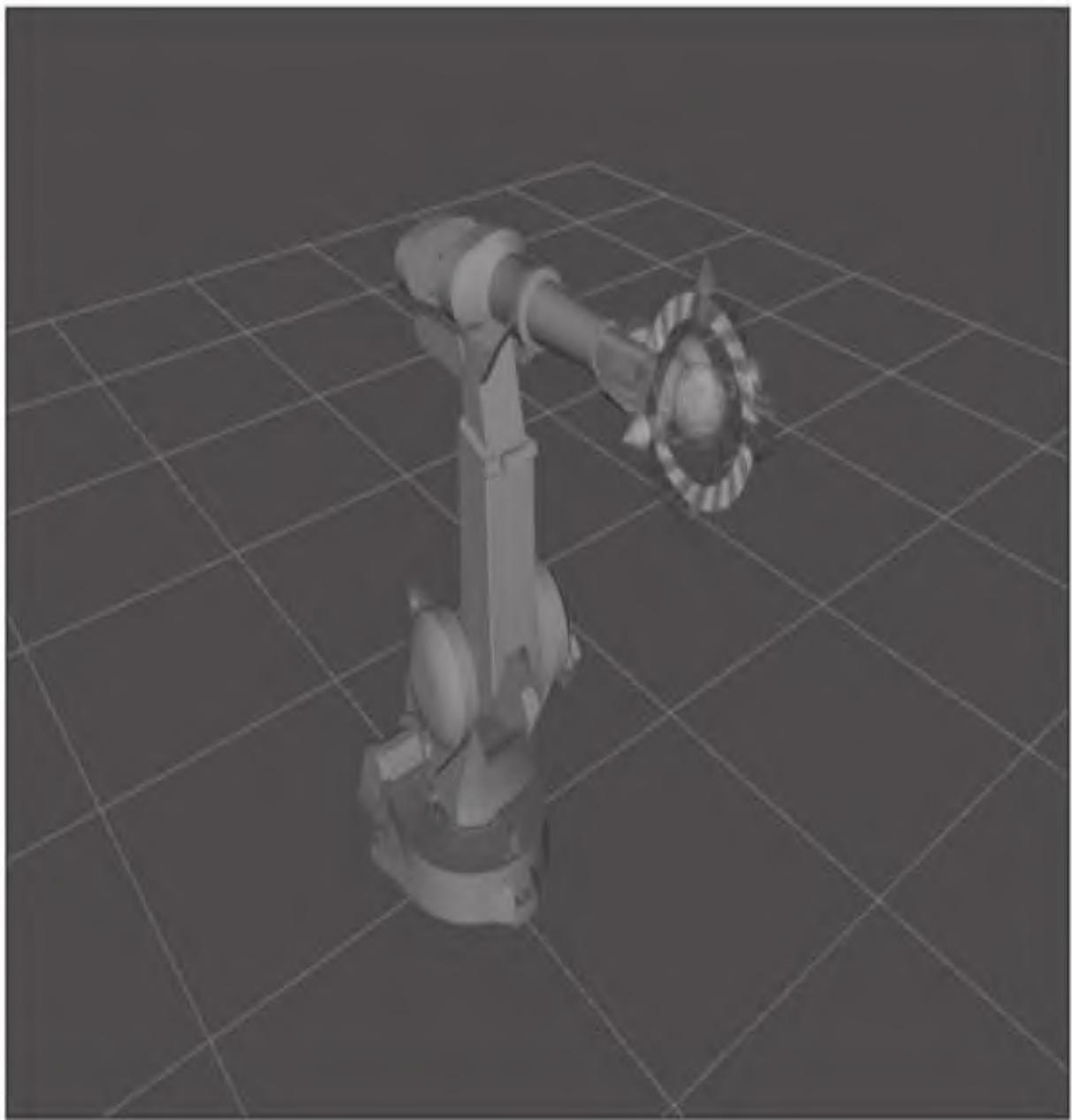


图14-12 ABB IRB 2400的运动规划

14.10 ROS-Industrial机器人支持软件包

ROS-I机器人支持软件包是工业机器人遵循的新惯例。这些支持软件包的目的是使不同供应商的各种工业机器人的ROS软件包的维护标准化。使用标准的方式将文件存放在支持软件包中，我们访问里面的文件就不会有任何困惑。我们可以展示ABB机器人的支持软件包，也可以看到相关文件夹、文件及其用法。

我们已经下载了ABB机器人软件包，在这个文件夹中我们可以看到三个支持软件包，分别用于支持三款ABB机器人。这里我们以ABB IRB 2400的支持软件包abb_irb2400_support为例。这是ABB工业机器人IRB 2400的支持软件包。下面列出了软件包中的文件夹和文件：

- config：与文件夹的名字一样，它包含了关节名称的配置文件，RViz 配置和机器人模型的配置文件。
- joint_names_irb2400：该配置文件在config文件夹中，其中包含了ROS 控制器使用的机器人的关节名称。
- launch：该文件夹包含了机械臂启动文件的定义。这些文件遵循所有 工业机器人的通用规范。
- load_irb2400.launch：该文件只在参数服务器上加载robot_description。根据机器人的复杂性，可以增加xacro文件的数量。该文件将所有xacro 文件加载到单个启动文件中。我们可以简单包含这个启动文件，而不是在其他启动文件中添加robot_description的单独代码。
- test_irb2400.launch：该启动文件可以显示已加载的URDF。我们可以在RViz中对URDF进行检查和确认。该启动文件包含了前面的启动文

件，并且启动了joint_state_publisher和robot_state_publisher节点。这两个节点有助于在RViz中与用户进行交互。这样，无须真实硬件即可运行。

- robot_state_visualize_irb2400.launch：该启动文件通过使用适当参数运行ROS-Industrial驱动软件包中的节点，并可视化真实机器人的当前状态。通过运行RViz和robot_state_publisher节点就可以可视化机器人的当前状态了。该启动文件需要真实的机器人或仿真接口。与该启动文件一起提供的主要参数之一是工业控制器的IP地址。另外注意，控制器需要运行一个ROS-Industrial服务器节点。
- robot_interface_download_irb2400.launch：该启动文件用于启动从工业机器人控制器到ROS的双向通信。用工业机器人客户端节点报告机器人的状态（robot_state节点），订阅关节命令话题并发送关节位置给控制器（joint_trajectory node）。该启动文件也需要访问仿真或真实的机器人控制器，但是要提供工业控制器的IP地址。控制器也需要运行ROS-Industrial服务器程序。
- urdf：该文件夹包含了机器人模型的标准的xacro文件集。
- irb2400_macro.xacro：这是特定机器人的xacro定义。它不是一个完整的URDF，只是操作器部分的宏定义。我们可以将该文件包含在另一个文件中，并且创建这个宏的实例。
- irb2400.xacro：这是顶层的xacro文件，它创建了前面部分所讨论的宏的实例。该文件不包括除机器人宏之外的任何其他文件。这个xacro文件将被加载到我们前面讨论过的load_irb240.launch文件中。
- irb2400.urdf：这是使用xacro工具从前面的xacro文件生成的URDF。当工具或软件包无法直接加载xacro时，将使用该文件。这是机器人顶层的URDF。
- meshes：包含了用于可视化和碰撞检测的网格模型。
- irb2400：该文件夹包含了特定机器人的网格模型文件。

- visual: 该文件夹包含了用于可视化的STL文件。
- collision: 该文件夹包含了用于碰撞检测的STL文件。
- tests: 该文件夹包含了测试启动文件以测试前面所有的启动文件。
- rosrun _test.xml: 该启动文件用于测试所有的启动文件。

RViz中ABB机器人的可视化

创建机器人模型后，我们可以用test_irb2400.launch文件对其进行测试。下面的命令将启动对ABB IRB 2400机器人的测试界面：

```
$ rosrun abb_irb2400_support test_irb2400.launch
```

它将在RViz中显示带有关节状态发布者节点的机器人模型，如图14-13所示。

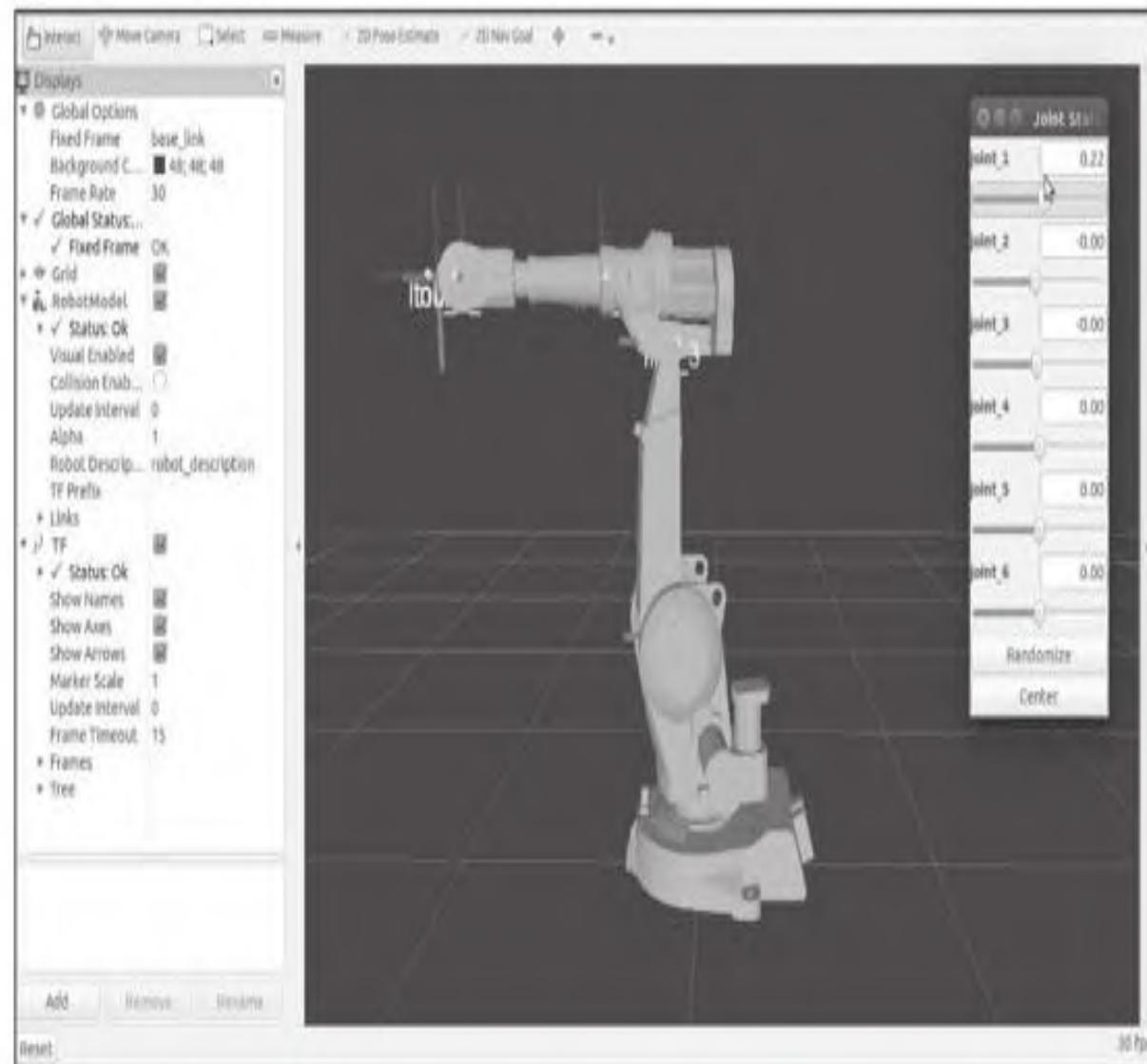


图14-13 在RViz中可视化显示ABB IRB 2400

我们可以通过调节关节状态发布者的滑动条数值来调整机器人的关节。使用这个测试界面，我们还可以确认URDF设计是否正确。

14.11 ROS-Industrial机器人客户端软件包

工业机器人客户端节点负责将机器人位置/轨迹数据从ROSMoveIt！发送给工业机器人控制器。工业机器人客户端将轨迹数据转换为simple_message，并用simple_message协议与机器人控制器通信。运行有服务器和客户端节点的工业机器人控制器将与该控制器相连接并开始与该服务器通信。

设计工业机器人客户端节点

industrial_robot_client软件包包含用来实现工业机器人客户端节点的各种类定义。客户端应具有的主要功能包括从机器人控制器更新机器人当前的状态，以及向控制器发送关节位置消息。有两个负责获取机器人的状态和发送关节位置数据的主要节点如下：

- robot_state节点：该节点负责发布机器人的当前位置、状态等信息。
- joint_trajectory节点：该节点订阅了机器人的命令话题，并且通过简单消息协议将关节位置命令发送给机器人控制器。

图14-14给出了工业机器人客户端提供的API列表。

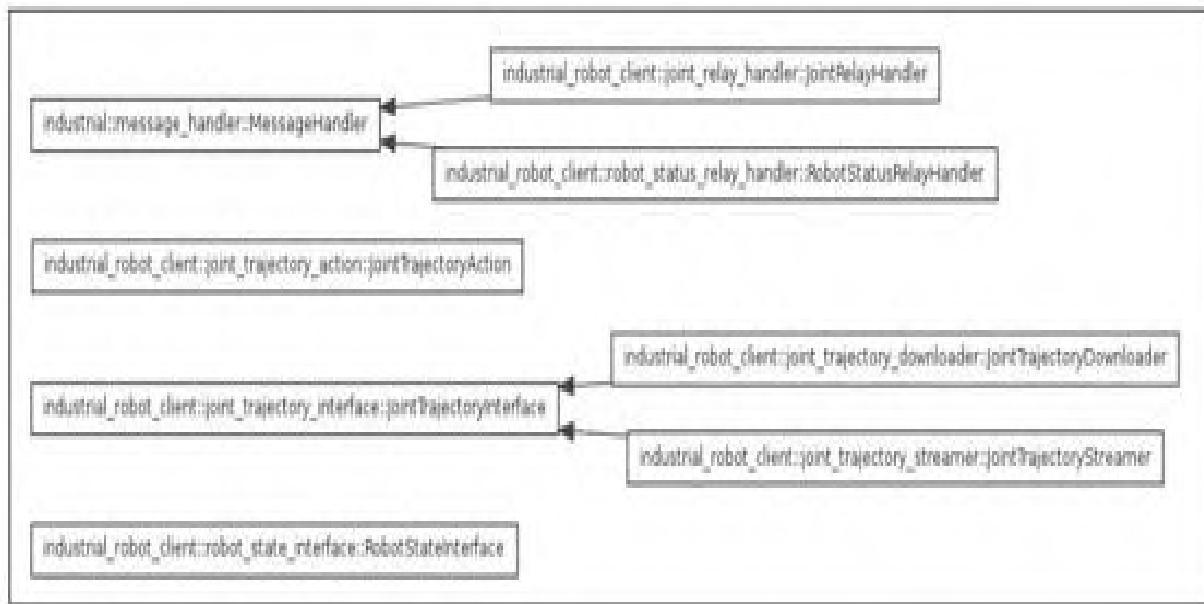


图14-14 工业机器人客户端API列表

我们可以简要地介绍这些API及其功能，如下所示：

- RobotStateInterface：从机器人控制器接收到位置数据后，该类包含了按一定的时间间隔发布当前机器人位置和状态的方法。
- JointRelayHandler：RobotStateInterface类封装了一个名为MessageManager的类。其作用是监听simple_message机器人连接并用Messagehandler处理每个消息。JointRelay-Handler的功能就是一个MessageHandler，其作用是向joint_states话题中发布关节位置信息。
- RobotStatusRelayHandler：这是另一个MessageHandler，可以向robot_status话题中发布当前的机器人状态信息。
- JointTrajectoryInterface：该类包含当接收到ROS轨迹命令时将机器人的关节位置发送到控制器的方法。
- JointTrajectoryDownloader：该类派生于JointTrajectoryInterface类，它实现了一个名为send_to_robot()的方法。该方法将整个轨迹作为消息序列发送到机器人控制器。只有从客户端获取到所有的序列后，机器人控制器才会执行机器人中的轨迹。

- JointTrajectoryStreamer: 除了实现send_to_robot()方法外，这个类与前面的类基本相同。该方法在单独的线程中向控制器发送单独的关节数据。每个位置命令仅在执行现有命令之后才发送。从机器人方面看，需要一个缓冲区来接收位置信息以使运动更加平滑。

工业机器人客户端内的节点如下所示：

- robot_state: 该节点基于RobotStateInterface运行，它可以发布当前机器人的状态。
- motion_download_interface: 该节点运行JointTrajectoryDownloader，它将按顺序将轨迹下载到控制器中。
- motion_streaming_interface: 该节点运行JointTrajectoryStreamer，它将用线程并行发送关节位置信息。
- joint_trajectory_action: 该节点提供一个基本的actionlib接口。

14.12 ROS-Industrial机器人驱动软件包

在本节中，我们将讨论工业机器人驱动软件包。如果我们以ABB机器人为例，它就是一个名为abb_driver的软件包。该软件包负责与工业机器人控制器通信。该软件包包含了工业机器人客户端和可以与控制器通信的启动文件。我们可以查看abb_driver/launch文件夹中的内容。下面就是名为robot_interface.launch的启动文件的定义：

```
<launch>
  <!-- robot_ip: IP-address of the robot's socket-messaging server -->
  <arg name="robot_ip" />

  <!-- J23_coupled: set TRUE to apply correction for J2/J3 parallel linkage
  -->
  <arg name="J23_coupled" default="false" />

  <!-- copy the specified arguments to the Parameter Server, for use by
  nodes below -->
  <param name="robot_ip_address" type="str" value="$(arg robot_ip)"/>
  <param name="J23_coupled" type="bool" value="$(arg J23_coupled)"/>

  <!-- robot_state: publishes joint positions and robot-state data
      (from socket connection to robot) -->
  <node pkg="abb_driver" type="robot_state" name="robot_state"/>
```

```

<!-- motion_download_interface: sends robot motion commands by
DOWNLOADING path to robot
                                (using socket connection to robot) -->

<node pkg="abb_driver" type="motion_download_interface"
name="motion_download_interface"/>

<!-- joint_trajectory_action: provides actionlib interface for high-level
robot control -->
<node pkg="industrial_robot_client" type="joint_trajectory_action"
name="joint_trajectory_action"/>
</launch>
```

该启动文件为ABB机器人提供了基于socket的连接，它使用的是标准的ROS-Industrial simple_message协议。该启动文件启动了几个节点，它们可以提供基本的机器人通信和高级的actionlib支持：

- robot_state：这将发布当前的关节位置和机器人状态数据。
- motion_download_interface：通过向机器人发送运动点来控制机器人运动。
- joint_trajectory_action：这是控制机器人运动的actionlib接口。

它们的用法如下：

```
$ robot_interface.launch robot_ip:=IP_OF_THE_ROBOT [J23_coupled:=false]
```

我们可以看看
abb_irb6600_support/launch/robot_interface_download_irb6640.launch文件，这是ABB IRB 6640型号的驱动程序。下面的代码给出了启动文件的定义。上述驱动程序启动文件包含在该启动文件中。在其他ABB型号的支持软件包中，使用同样的驱动程序，只不过是关节配置参数文件不同：

```
<launch>
  <arg name="robot_ip" />
  <arg name="J23_coupled" default="true" />

  <rosparam command="load" file="$(find
abb_irb2400_support)/config/joint_names_irb2400.yaml" />

  <include file="$(find abb_driver)/launch/robot_interface.launch">
    <arg name="robot_ip" value="$(arg robot_ip)" />
    <arg name="J23_coupled" value="$(arg J23_coupled)" />
  </include>
</launch>
```

前面的文件是robot_interface.launch (abb_driver) 机器臂的具体版本：

- IRB 2400提供的默认值： -J23_coupled=true
- 用法： robot_interface_download_irb2400.launch robot_ip:=

我们需要运行驱动程序的启动文件来与真实机器人控制器通信。对于ABB机器人IRB 2400，我们可以用下面的命令来启动机器人控制器和ROS客户端的双向通信：

```
$ rosrun abb_irb2400_support robot_interface_download_irb2400.launch
robot_ip:=IP_OF_THE_ROBOT
```

启动驱动程序后，我们就可以使用MoveIt!接口开始运动规划了。还应注意，需要在启动机器人驱动程序之前配置好ABB机器人并找到机器人控制对应的IP。

14.13 理解MoveIt! IKFast插件

ROS中默认的数值IK解算器是KDL。KDL通常在DOF>6时使用。对DOF≤6的机器人，我们可以使用解析解算器，它比数值解算器（如KDL）快得多。大部分工业机械臂DOF≤6，因此为每个机械臂制作一个解析解算器插件更合适。虽然机器人也可以用KDL解算器进行求解，但是如果我们想要一个快速的IK解算方法，我们可以选择像IKFast这类的模块，为MoveIt!生成基于解析解算器的插件。我们可以查看机器人（例如UR或ABB）中存在的IKFast插件软件包：

- ur_kinematics：该软件包包含UR机器人公司的UR-5和UR-10的IKFast解算器插件。
- abb_irb2400_moveit_plugins/irb2400_kinematics：该软件包包含用于ABB机器人IRB 2400的IKFast解算器插件。

我们可以通过这些程序为MoveIt!构建一个IKFast插件。当我们为定制的工业机械臂创建IK解算器插件时将非常有用。下面让我们看看如何为工业机器人ABB IRB 6640创建MoveIt! IKFast插件。

14.14 为ABB IRB 6640机器人创建MoveIt! IKFast插件

我们已经学习了ABB IRB 6640机器人的MoveIt!软件包。该机器人使用了默认的数值解算器。在本节中，我们将讨论如何利用IKFast生成IK解算器插件。IKFast是Rosen Diankov开发的OpenRAVE运动规划软件中提供的强大的逆运动解算器。在本节末尾，我们可以用我们自定义的逆运动学插件来运行该机器人的MoveIt!演示。

简而言之，我们将为ABB IRB 6640机器人创建一个MoveIt! IKFast插件。可以在MoveIt!设置向导中选择此插件，或者在moveit-config软件包的config/kinematics.yaml文件中设置它。

14.14.1 开发MoveIt! IKFast插件的前提条件

下面是我们用于开发MoveIt! IKFast插件的配置：

- Ubuntu 16.04 LTS x86_64 bit
- ROS-kinetic desktop-full
- Open-Rave 0.9

14.14.2 OpenRave和IKFast模块

OpenRave是一组在真实应用中开发、测试和部署运动规划算法命令行和GUI工具集。IKFast是OpenRave中的一个模块，它是一个机器人运动学编译器。OpenRave是由一位名为Rosen Diankov的机器人研究者开发的。IKFast编译器可以解析地解决机器人的逆运动学问题，并生成优化且独立的C++文件，这些文件可以部署在我们的代码中来解决IK问题。IKFast编译器生成IK的解析解速度，它比KDL提供的数值解算器要快得多。IKFast编译器可以处理任意数量的DOF，但是实际上它非常适合DOF≤6的情况。IKFast是一个Python脚本，它的参数为IK类型、机器人型号、基本连杆的关节位置和末端执行器等。

下面是IKFast支持的主要IK类型：

- Transform 6D：该末端执行器能够实现6D变换。
- Rotation 3D：该末端执行器能够实现3D旋转。
- Translation 3D：该末端执行器原点能够实现3D平移。

MoveIt! IKFast

MoveIt!的ikfast软件包中包含了用OpenRave源代码文件生成的运动解算器插件的工具。我们将使用该工具来为MoveIt!生成IKFast插件。

安装MoveIt! IKFast软件包

下面的命令将在ROSKinetic中安装moveit-ikfast软件包：

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

在Ubuntu 16.04上安装OpenRave

在Ubuntu16.04上安装OpenRave很繁琐。接下来我们将按照下面的过程使用源码安装OpenRave：

1) 将源代码下载至某一文件：

```
$ git clone --branch latest_stable  
https://github.com/rdiankov/openrave.git
```

2) 为了编译源代码，我们需要安装以下软件包：

boost、Python开发软件包和NumPy：

```
$ sudo apt-get install libboost-python-dev  
$ python python-dev python-numpy ipython python-pip
```

3) 安装Python（科学版），及其软件包来处理符号数学问题。注意，为了与OpenRave和Ubuntu 16.04配合使用，建议使用Python v0.7.1：

```
$ sudo apt-get install python-scipy  
$ pip install -v sympy==0.7.1
```

4) 安装开源的资源导入库来处理3D文件格式：

```
$ sudo apt-get install libassimp-dev assimp-utils python-pyassimp
```

5) 安装Qt4 GUI工具集：

```
$ sudo apt-get install libsoqt4-dev
```

6) 要安装collada文件处理软件包需要从下面的Git库中下载其源码：

```
$ git clone https://github.com/rdiankov/collada-dom.git
```

7) 进入collada-dom文件夹，创建一个编译文件夹，然后开始编译该软件：

```
$ cd collada-dom && mkdir build && cd build  
$ cmake ..  
$ make && sudo make install
```

8) 现在，我们将看到如何安装cmake-gui，这样就能根据CMakeLists.txt配置来生成Makefile文件。OpenRave项目是基于CMake的，所以我们需要这个工具来生成Makefile文件。

```
$ sudo apt-get install cmake-qt-gui
```

安装OpenRave的第一步就是从CMakeLists.txt文件生成Unix的Makefile：在OpenRave的下载源码文件夹中创建一个build文件夹，然后打开cmake-gui来配置和构建Makefile。首先配置好源代码和build文件夹路径，如图14-15所示，配置好所有选项后，取消对MATLAB和Octave接口的支持。

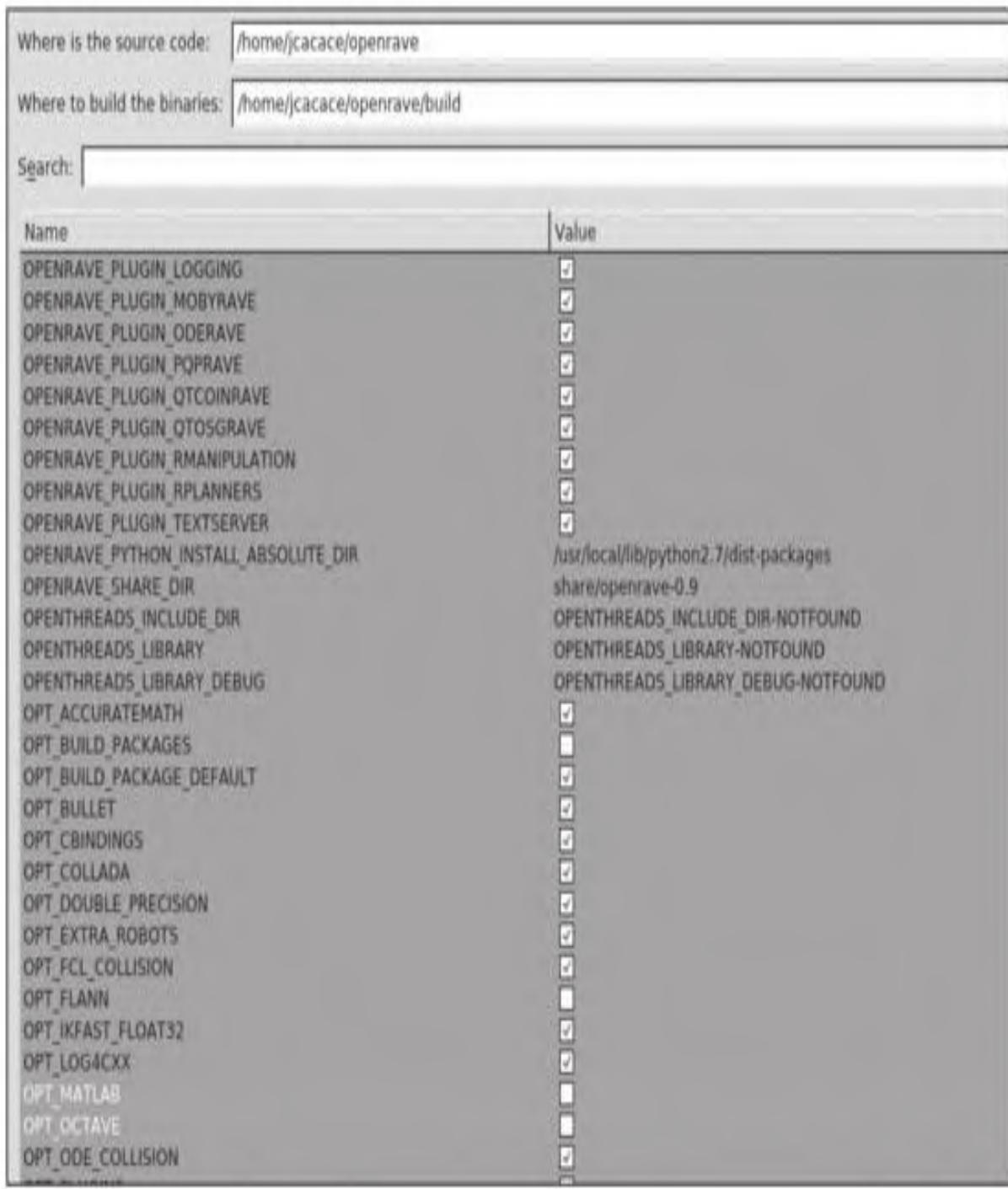


图14-15 用cmake-gui配置OpenRave

单击Generate按钮在选定的build文件夹中生成Makefile文件。然后切换到build文件夹中，用下面的命令编译代码，并进行安装：

```
$ make  
$ sudo make install
```

安装完OpenRave后，执行下面的命令来检查OpenRave是否工作正常：

```
$ openrave
```

如果工作一切正常，会打开一个3D视图界面。

14.15 为使用OpenRave，创建机器人的COLLADA文件

在本节中，我们将讨论如何使用OpenRave的URDF机器人模型。首先，我们将学习如何将collada文件（.dae）格式转换为URDF。然后，该文件将用来生成IKFast源文件。如果要将URDF模型转换为collada文件，我们可以使用名为collada_urdf的ROS软件包。

我们将使用ABB IRB 6640机器人模型，该模型的文件名为irb6640.urdf，它保存在/urdf文件夹下的abb_irb6600_support软件包中。或者，你也可以从随书发布的ikfast_demo文件夹中获取该文件。将该文件复制到你的工作文件夹中并运行下面的命令来进行转换：

```
$ roscore && rosrun collada_urdf urdf_to_collada irb6640.urdf irb6640.dae
```

命令输出collada文件格式的机器人模型。

i 在大多数情况下，该命令会执行失败，因为大多数的URDF文件包含STL网格模型，它们可能无法按照预期转换为DAE。如果机器人网格模型是DAE格式的，它将会运行成功。如果命令执行失败，可以执行下面的步骤：

用下面的命令安装MeshLab工具，以查看和编辑网格模型：

```
$ sudo apt-get install meshlab
```

在MeshLab中打开abb_irb6600_support/meshes/irb6640/visual中的网格文件，然后将它导出为具有相同文件名的DAE格式。

编辑irb6640.urdf文件并将STL扩展中的显示网格更改为DAE格式。该工具仅用来处理网格模型的可视化效果，因此我们将获得一个最终的DAE模型。

我们可以在OpenRave中使用下面的命令打开irb6640.dae文件：

```
$ openrave irb6640.dae
```

然后我们将在OpenRave中看到机械臂模型，如图14-16所示。



图14-16 在OpenRave中查看ABB IRB 6640

我们可以用下面的命令检查机器人的连杆信息：

```
openrave-robot.py irb6640.dae --info links
```

我们将得到如下格式的机器人的连杆信息：

name	index	parents
base_link	0	
base	1	base_link
link_1	2	base_link
link_2	3	link_1
link_4	5	link_3
link_5	6	link_4
link_6	7	link_5
tool0	8	link_6
link_cylinder	9	link_1
link_piston	10	link_cylinder

14.16 为IRB 6640机器人生成IKFast CPP文件

获取连杆信息后，我们可以开始生成逆运动解算器的源文件，以处理机器人的IK。本教程需要的所有文件都可以在本书提供的ikfast_demo源代码文件夹中找到。或者，你可以从下面的Git库中下载该代码：

```
$ git clone https://github.com/jocacace/ikfast_demo.git
```

用下面的命令为ABB IRB 6640机器人生成IK解算器：

```
$ python `openrave-config --python-dir`/openravepy/_openravepy_ikfast.py -  
-robot=irb6640.dae --iktype=transform6d --baselink=1 --eelink=8 --  
savefile=ikfast61.cpp
```

上面的命令生成了一个名为ikfast61.cpp的CPP文件，其中IK类型是transform6d，baselink的位置是1，末端执行器连杆是8。我们将机器人的DAE文件作为机器人参数。

在将该代码与MoveIt!一起使用之前，我们可以用ikfastdemo.cpp示例程序对其进行测试。这个ikfastdemo.cpp已被修改为包含ikfast61.cpp，你可以从头文件列表中看到：

```
#define IK_VERSION 61
#include "output_ikfast61.cpp"
```

编译示例源码文件：

```
$ g++ ikfastdemo.cpp -lstdc++ -llapack -o compute -lrt
```

上面的命令生成了一个名为compute的可执行文件。如果在不带参数的情况下运行它，程序将显示使用菜单。使用下面的命令给定一组关节角度值来获取正运动学解：

```
$ ./compute fk j0 j1 j2 j3 j4 j5
```

这里的j0、j1、j3、j4、j5是以弧度表示的关节角度值。要想测量IKFast算法对一组随机关节角度所取的平均时间，请使用下面的命令：

```
$ ./compute iktiming
```

现在我们已经成功创建了逆运动解算器的CPP文件。接下来我们就可以用这个源码文件创建MoveIt! IKFast插件了。

创建MoveIt! IKFast插件

创建MoveIt! IKFast插件的过程很简单，不需要编写代码。整个过程都可以用一些工具来生成。我们唯一要做的就是创建一个空的ROS软件包。下面是创建一个插件的过程：

创建一个空的软件包，它的名字应包含机器人名称和型号。该软件包将用插件生成工具转换为最终的插件软件包。

```
$ catkin_create_pkg abb_irb6640_moveit_plugins
```

然后用catkin_make命令编译工作区。你也可以从下面的库中下载ROS软件包：

```
$ git clone  
https://github.com/jocacace/abb_irb6640_moveit_plugins.git
```

工作区编译好后，复制ikfast.h文件到abb_irb6640_moveit_plugins/include文件夹下。

复制先前在软件包中创建的ikfast61.cpp文件，将其重命名为abb_irb6640_manipulator_ikfast_solver.cpp。该文件名由机器人名称、型号、机器人类型等组成。这种命名方式是生成工具所要求的。

执行了这些步骤后，在IK解算器CPP文件所在的路径下打开两个终端。在其中一个终端运行roscore命令。在另一个终端，进入create软件包并输入插件创建命令，如下所示：

```
$ rosrun moveit_kinematics create_ikfast_moveit_plugin.py abb_irb6640  
manipulator abb_irb6640_moveit_plugins  
abb_irb6640_manipulator_ikfast_solver.cpp
```

由于URDF文件和SRDF文件中指定的机器人名称不匹配，该命令可能会执行失败。要解决这个问题，我们需要更改SRDF文件中的机器人名称。该文件放在abb_irb6640_moveit_config/config文件夹中。你需要将这个文件第7行的<robot name=“abb_irb6640_185_280”>改为<robot name=“abb_irb6640”>。或者简单地将该文件用ikfast_demo文件夹中的文件替换。

moveit_ikfastROS软件包包含用来生成插件的create_ikfast_moveit_plugin.py脚本。第一个参数是带有型号的机

器人名称，第二个参数是机器人的类型，第三个参数是我们之前创建的软件包名称，第四个参数是IK解算器CPP文件的名称。这个工具需要abb_irb6640_moveit_config软件包才能工作。它将用给定的机器人名称来搜索这个软件包。因此，如果机器人的名称有误，这个工具将会提示一个找不到机器人moveit软件包的错误。

如果创建成功，终端上将显示如图14-17所示信息。

```
IKFast Plugin Generator
Loading robot from 'abb_irb6640_moveit_config' package ...
Creating plugin in 'abb_irb6640_moveit_plugins' package ...
found 1 planning groups: manipulator
found group 'manipulator'
found source code generated by IKFast version 750435529

created plugin file at '/home/jcacace/ros_ws/src/MASTERING_ROS/ch13/abb_irb6640_moveit_plugins/src/abb_irb6640_manipulator_ikfast_moveit_plugin.cpp'

created plugin definition at: '/home/jcacace/ros_ws/src/MASTERING_ROS/ch13/abb_irb6640_moveit_plugins/abb_irb6640_manipulator_ikfast_moveit_plugin_description.xml'

overwrote CMakeLists file at '/home/jcacace/ros_ws/src/MASTERING_ROS/ch13/abb_irb6640_moveit_plugins/CMakeLists.txt'

modified package.xml at '/home/jcacace/ros_ws/src/MASTERING_ROS/ch13/abb_irb6640_moveit_plugins/package.xml'

modified kinematics.yaml at /home/jcacace/ros_ws/src/abb_irb6640_moveit_config/config/kinematics.yaml

created update plugin script at /home/jcacace/ros_ws/src/MASTERING_ROS/ch13/abb_irb6640_moveit_plugins/update_ikfast_plugin.sh
```

图14-17 成功为MoveIt!创建IKFast插件的终端信息

从这些消息中可以看出创建插件后，
abb_irb6640_moveit_config/config/kinematics.yaml文件被更新了。将
abb_irb6640_manipulator_kinematics/IKFastKinematicsPlugin作为运动解算器。下面的代码中显示了更新后的文件版本：

```
manipulator:  
  kinematics_solver:  
    abb_irb6640_manipulator_kinematics/IKFastKinematicsPlugin  
    kinematics_solver_search_resolution: 0.005  
    kinematics_solver_timeout: 0.005  
    kinematics_solver_attempts: 3
```

现在你可以再次编译工作区来安装插件，并开始使用新的IKFast插件来操作机器人，启动演示场景命令如下：

```
$ rosrun abb_irb6640_moveit_config demo.launch
```

14.17 习题

下面是一些常见的问题，它们可以帮助你更好地学习和理解本章内容：

- 使用ROS-Industrial软件包的主要好处是什么？
- ROS-I在设计工业机器人的URDF时要遵守的惯例是什么？
- ROS的支持软件包有什么作用？
- ROS的驱动软件包有什么作用？
- 为什么我们在工业机器人中需要使用IKFast插件，而不是默认的KDL插件？

14.18 本章小结

在本章，我们讨论了工业机器人的一个ROS新接口，称为ROS-Industrial。我们学习了开发工业机器人软件包的基本概念，以及如何在Ubuntu上安装这些软件包。安装完成后，我们学习了这些软件包集的框图，并讨论了如何为工业机器人开发URDF模型，以及创建MoveIt!接口。详细介绍了这些主题后，我们安装了UR和ABB的一些工业机器人软件包。我们还学习了MoveIt!软件包的结构，然后开始学习ROS-Industrial支持软件包。我们详细地讨论了如何使用工业机器人客户端以及如何创建MoveIt! IKFast插件等概念。最后，我们将开发的插件应用于ABB机器人中。

下一章，我们将介绍ROS软件开发中的调试方法与最佳实战技巧。

第15章

调试方法与最佳实战技巧

在本章，我们将讨论如何在ROS中安装集成开发环境（Integrated Development Environment, IDE），ROS最佳实战技巧与经验，以及ROS的调试方法。这是本书的最后一章，在我们开始开发ROS之前，了解一些代码编写的标准方法会很有帮助。

本章将介绍以下内容：

- 在ROS中使用RobWare Studio IDE
- ROS最佳实战技巧与经验
- 在ROS中使用C++进行最佳编码实战经验
- ROS中的重要调试方法

在我们准备ROS编程之前，安装一个ROSIDE是大有好处的。虽然为ROS安装IDE不是必需的，但是有这样一个开发环境可以节省开发者的时间。IDE可以提供字词自动补全功能，还能提供可以简化编程的编译和调试工具。虽然我们可以使用任何一种文本编辑器，如Sublime和VIM，或者只用Gedit进行ROS编程，但是如果你打算在ROS中开发一个大项目，选择IDE将会很有用处。因此，我们将在本章重点介绍RoboWare Studio，这是一个专为ROS开发而设计的IDE。它使ROS开发变得直观、简单且易于管理。除用于编程的工具之外，RoboWare还提供了很多有用的工具来管理ROS的工作区，ROS节点的创建、处理和编译，以及支持运行ROS的工具。

15.1 在Ubuntu中安装RoboWare Studio

Linux系统中已经有一些IDE了，例如NetBeans (<https://netbeans.org>)、Eclipse (www.eclipse.org)、QtCreator (https://wiki.qt.io/Qt_Creator)，适用于不同的编程语言。为了在IDE中编译和运行ROS程序，必须设置好ROS环境。所有的IDE可能都有一个配置文件来配置环境，但是在ROS获取的shell中运行IDE，是避免不一致的最简单方式。在本节，我们将讨论如何将RoboWare Studio IDE与ROS配合使用。可以在<http://wiki.ros.org/IDEs>中找到能够搭配ROS其他IDE的详细列表。

RoboWare Studio是专为ROS开发而设计的IDE，支持ROSKinetic版本。安装非常简单，不需要额外的配置即可自动检测并加载ROS环境。RoboWare Studio有许多无须配置即可使用的功能，可帮助ROS开发人员创建应用程序，如创建ROS软件包的图形界面，源文件（以及服务和消息文件），以及列出节点和软件包。

15.1.1 安装和卸载RoboWare Studio

为了安装RoboWare Studio，我们需要下载安装文件。可以从网址 <http://www.roboware.me/> 下载最新版本的软件，双击下载完的.deb文件，用软件包管理器GUI打开并安装，或者在终端使用下面的命令来安装：

```
$ cd /path/to/deb/file/  
$ sudo dpkg -i roboware-studio_<version>_<architecture>.deb
```

若想卸载该软件，可以使用下面的命令：

```
$ sudo apt-get remove roboware-studio
```

15.1.2 RoboWare Studio入门

安装好RoboWare后，可以从命令行启动它：

```
$ roboware-studio
```

打开RoboWare的主窗口。根据图15-1所示的列表，我们将讨论RoboWare界面的主要组件。



图15-1 RoboWare Studio 用户界面

- 1) EXPLORER (资源管理器) 窗口：该面板显示ROS工作区src文件夹的内容，在该面板中，你可以查看所有的ROS软件包。
- 2) NODE (节点)：在该面板，你可以访问工作区内所有编译好的节点。节点都被包含在软件包下，可以用该面板直接运行节点。
- 3) EDITOR (编辑器)：在该面板，你可以编辑软件包的源码。

4) TERMINAL (终端) 与OUTPUT (输出)：该面板允许开发者使用集成在IDE中的Linux终端，并在编译过程中检查可能的错误。

在开始编辑源码之前，我们应该在RoboWare中导入ROS工作区，如图15-2所示。在主工具栏中，选择File|Open Workspace，然后选择ROS工作区的文件夹。这样，位于src文件夹中的所有软件包都将显示在资源管理器中。

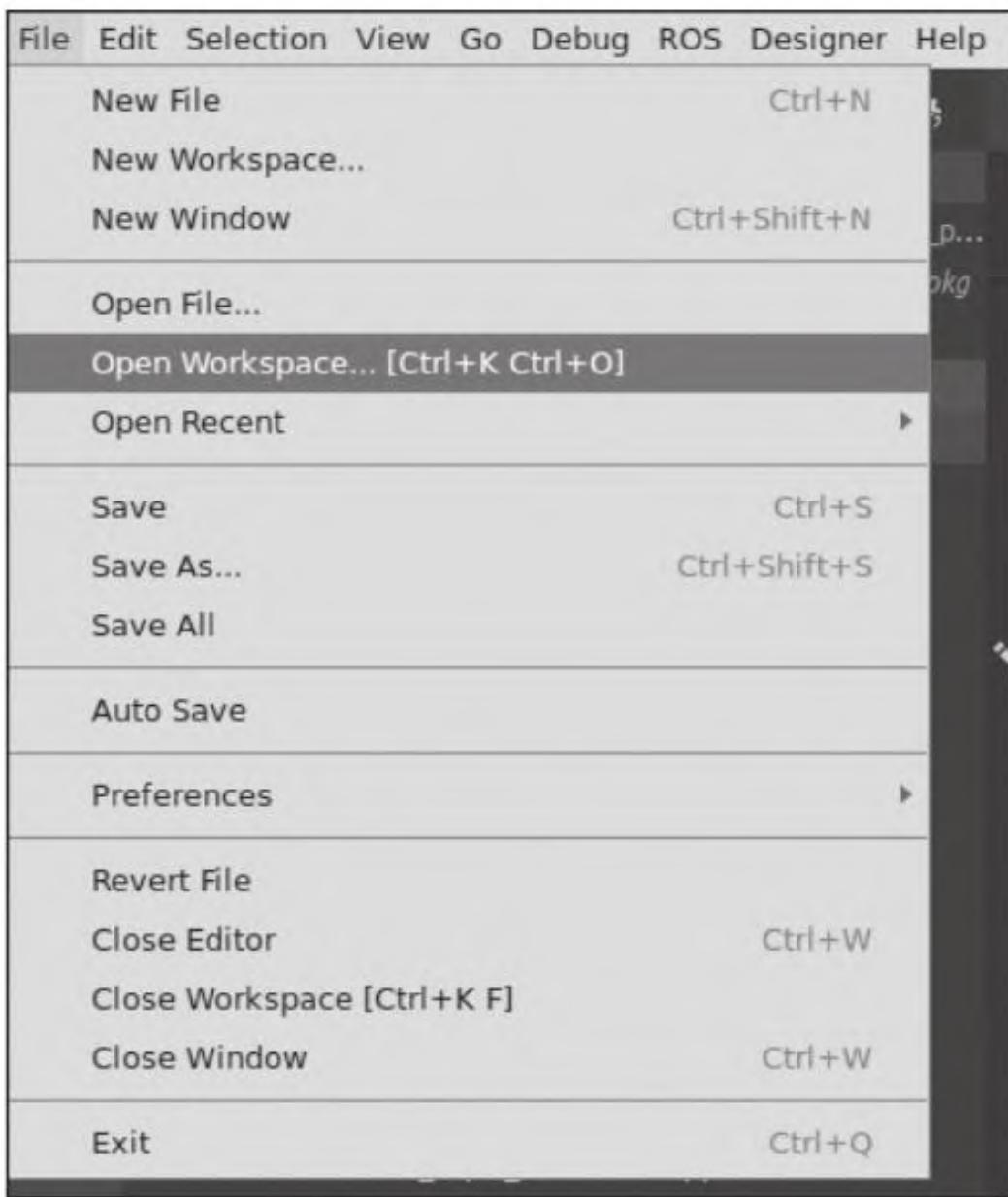


图15-2 在RoboWare中导入ROS工作区

15.1.3 在RoboWare Studio中创建ROS软件包

如前所述，RoboWare Studio允许开发者直接从用户界面管理ROS项目，而无须使用Linux终端或编辑CMakeLists.txt文件。在本节中，我们将讨论如何在RoboWare中创建和处理ROS软件包。

创建基于CPP可执行程序的ROS软件包，需要如下步骤：

1) 创建软件包。在资源管理器窗口的ROS工作区中的src文件夹上右击，然后选择AddROSPackage，并输入软件包的名称。这样就能创建一个新的ROS软件包了。在我们的例子中，我们正在创建一个叫作roboware_package的软件包，如图15-3所示。

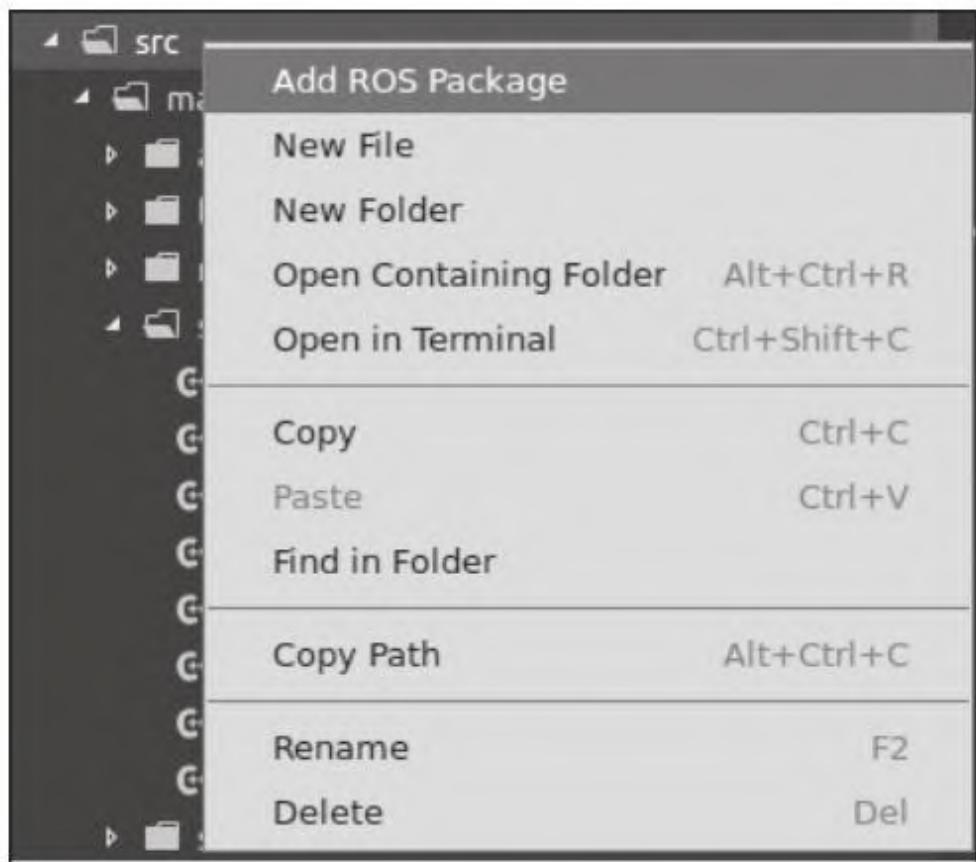


图15-3 在RoboWare Studio中添加ROS软件包

2) 创建源代码文件夹。在资源管理器窗口中右击软件包的名称，然后选择Add Src Folder，如图15-4所示。

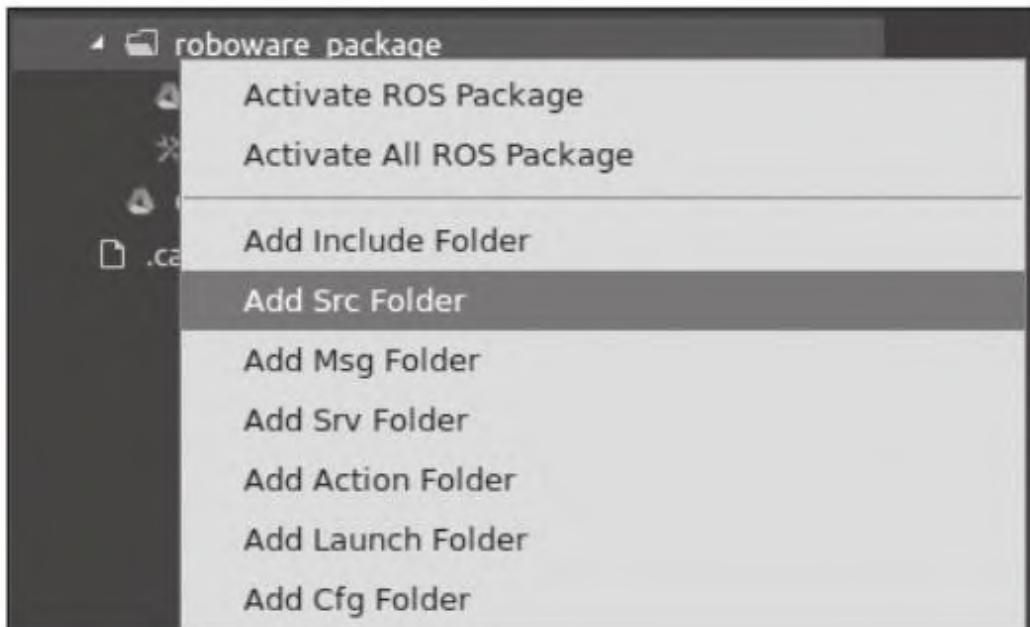


图15-4 在软件包中添加存放源代码的文件夹

3) 创建源码文件，在创建的src文件夹上右击，然后选择Add CPP File。输入源码文件名称后，RoboWare将询问你该文件是一个系统库文件还是一个可执行文件，在这里选择Executable（可执行文件），如图15-5所示。

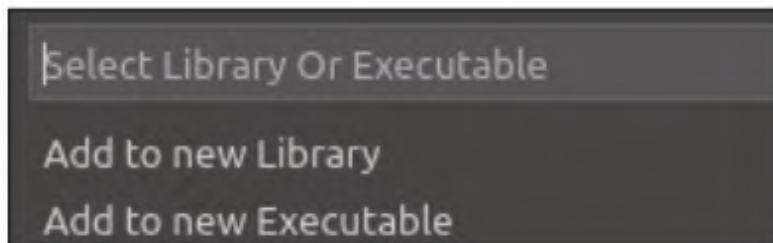


图15-5 选择CPP源码文件的类型

4) 添加软件包的依赖项。为软件包添加依赖项的操作是在资源管理器窗口右击软件包的名字，然后选择Edit catkinROSPackage Dependencies。在该输入栏中输入我们需要的依赖项列表。例如，我们可以添加roscpp和std_msgs依赖项，如图15-6所示。

```
roscpp std_msgs
```

Edit catkin ROS Package Dependencies list, Separated by space. (Press 'Enter' to confirm or 'E...

图15-6 指定ROS包的依赖项

在这4步操作过程中，RoboWare Studio将修改CMakeLists.txt文件，这样就能编译所需的可执行文件了。现在我们查看roboware_package更新后的CMakeLists.txt文件：

```
cmake_minimum_required(VERSION 2.8.3)
project(roboware_package)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs)
find_package(catkin REQUIRED COMPONENTS roscpp)
catkin_package()
include_directories( include ${catkin_INCLUDE_DIRS} )
add_executable(roboware
    src/roboware.cpp
)
add_dependencies(roboware ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(roboware
    ${catkin_LIBRARIES}
)
```

从生成的CMakeLists.txt文件中可以看到，已成功添加可执行文件和附加的库。同样的，我们还可以添加ROS消息、服务、动作等。现在我们准备在源文件中编写一些代码然后编译它。

15.1.4 在RoboWare Studio中编译ROS工作区

针对本地与远程编译和部署的ROS软件包，RoboWare Studio同时支持发行版本和调试版本。在本教程中，我们将配置RoboWare，从而可以编译本地开发模式的发行版本。要选择编译模式，可以直接利用资源管理器面板的下拉菜单，如图15-7所示。

要编译工作区，可以使用主工具栏的ROS|Build entry（或者使用快捷键Ctrl+Shift+B）。编译过程的输出将显示在Output面板中。

默认情况下，RoboWare Studio会编译工作区中的所有软件包（使用catkin_make命令）。为了手动指定一个或多个软件包来编译，我们可以在指定的软件包上右击，然后选定Activate-ROS-Package来激活它。

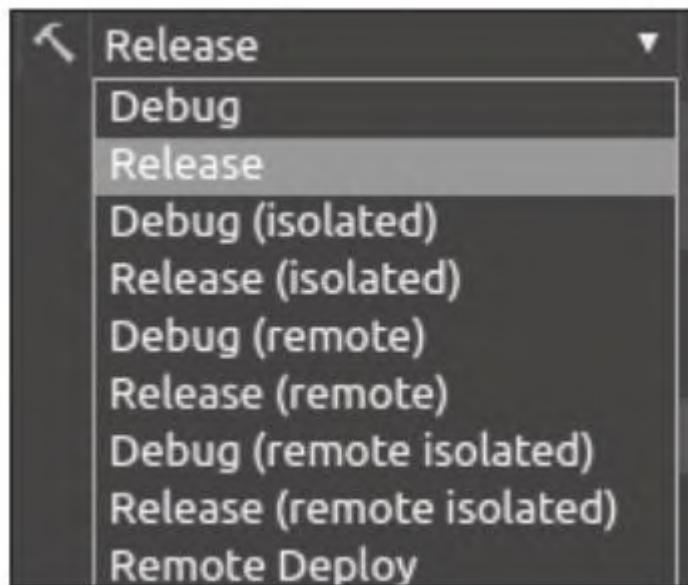


图15-7 选择RoboWare Studio的编译配置

这样的话，当我们单击build按钮时，只会编译已被激活的软件包，而那些未被激活的软件包会用删除线标记出来，如图15-8所示。

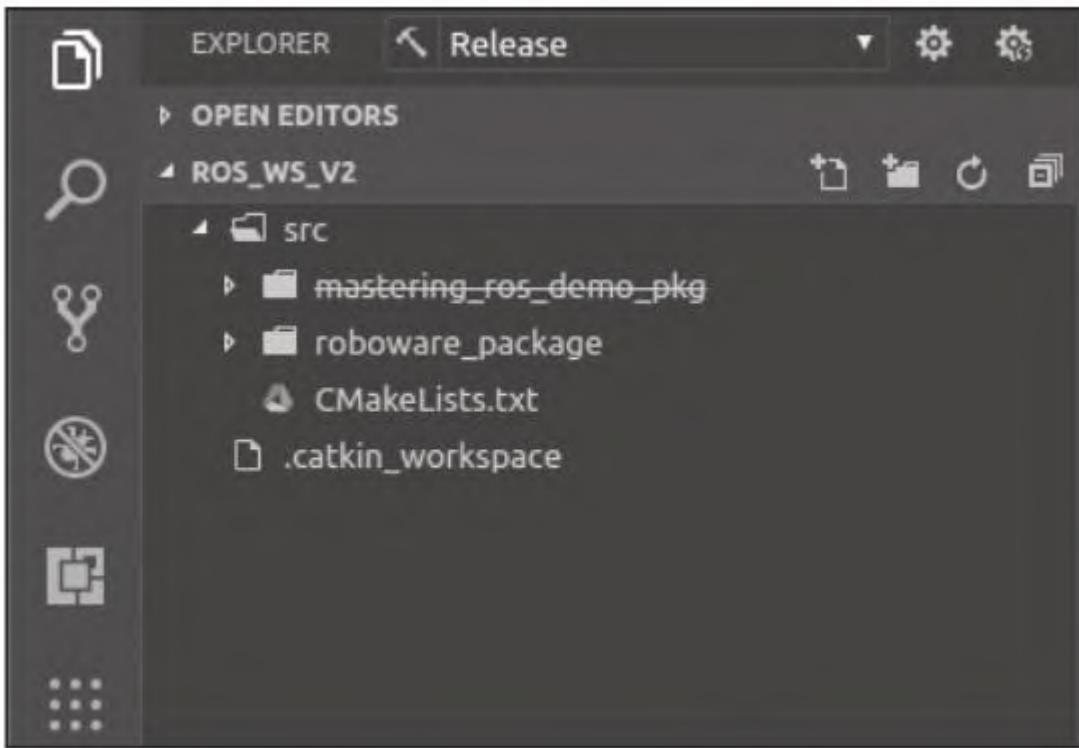


图15-8 显示已激活和未激活的软件包

我们可以通过在资源管理器窗口选择Activate All ROS Package来激活所有的软件包配置。

15.1.5 在RoboWare Studio中运行ROS节点

你可以通过使用roslaunch和rosrun命令来运行ROS节点。

首先，我们应该为我们的软件包创建一个启动文件。为了执行该操作，需要在软件包上右击，并选择Add Launch Folder来创建一个名为launch的文件夹。然后，在启动文件夹上右击并选择Add Launch File来添加新文件。当编辑好启动文件后，我们只需在启动文件上右击，然后选择Run Launch File即可，如图15-9所示。

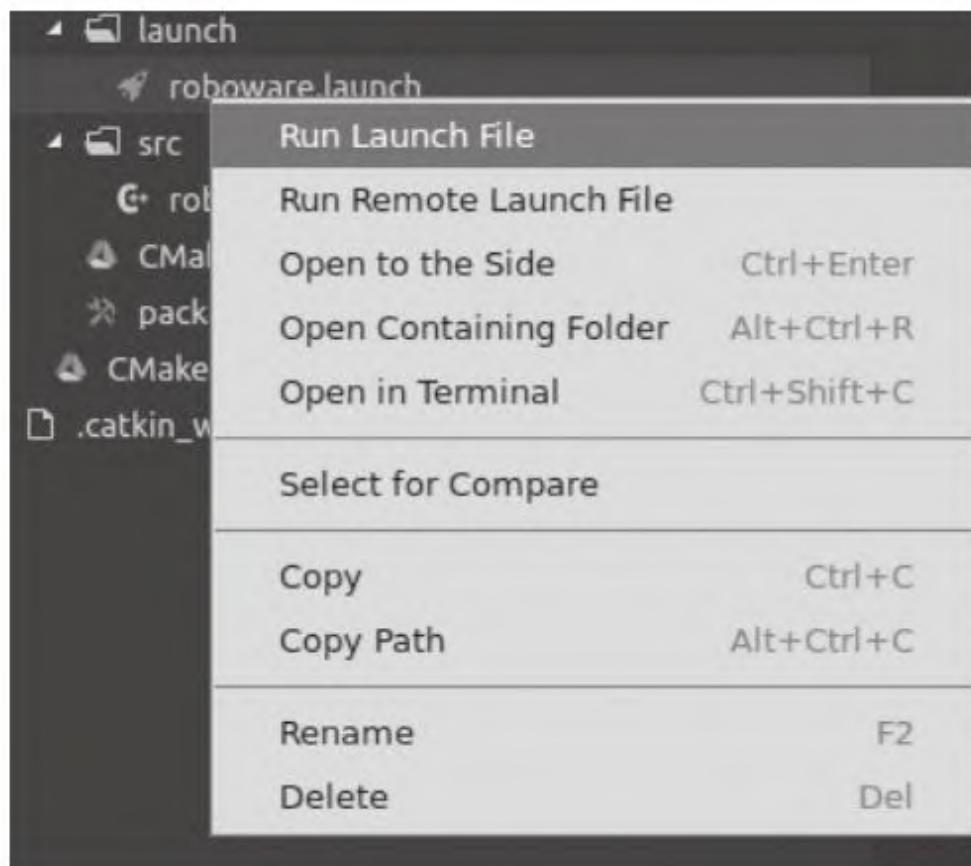


图15-9 在RoboWare Studio中运行roslaunch命令

要使用rosrun命令来执行ROS节点，我们必须从节点列表中选择要运行的可执行文件。这将打开节点窗口，进而允许我们在该节点上执行不同的操作。例如，点击Run this File将开始执行节点，如图15-10所示。

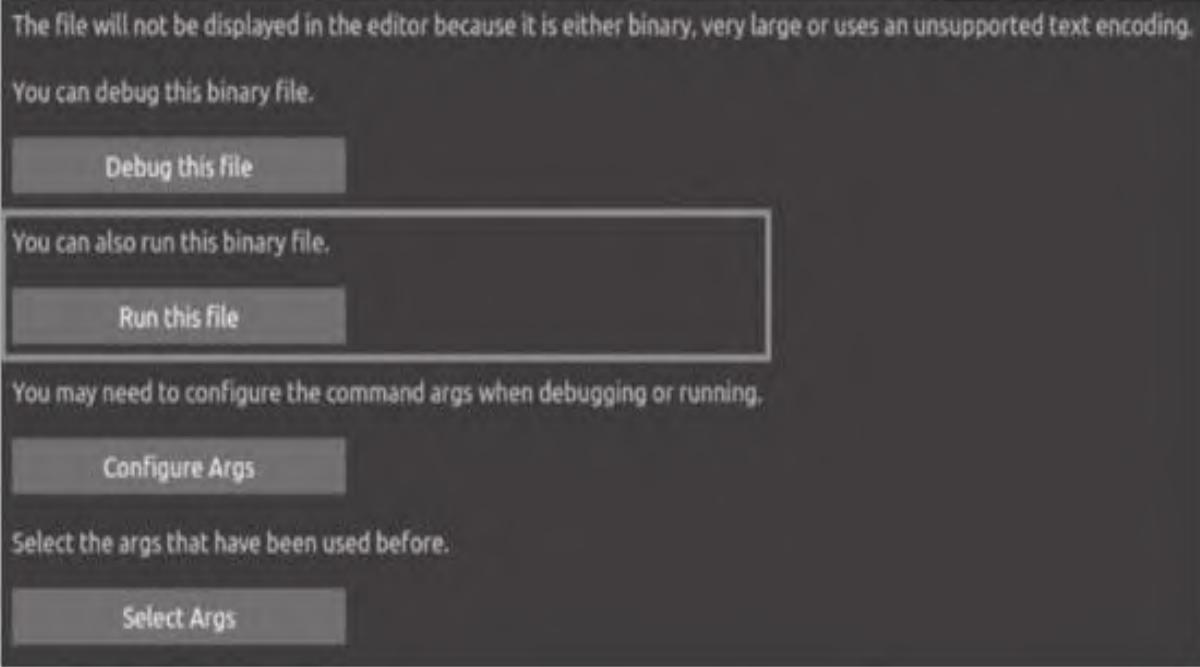


图15-10 使用rosrun运行ROS节点

用户可以在调试控制台窗口查看节点的输出信息。

15.1.6 在RoboWare界面启动ROS工具

RoboWare Studio允许开发者运行一些常用的ROS工具。要使用这些工具，可以在RoboWare的顶部工具栏中，点击ROS菜单展开如图15-11所示的下拉菜单。



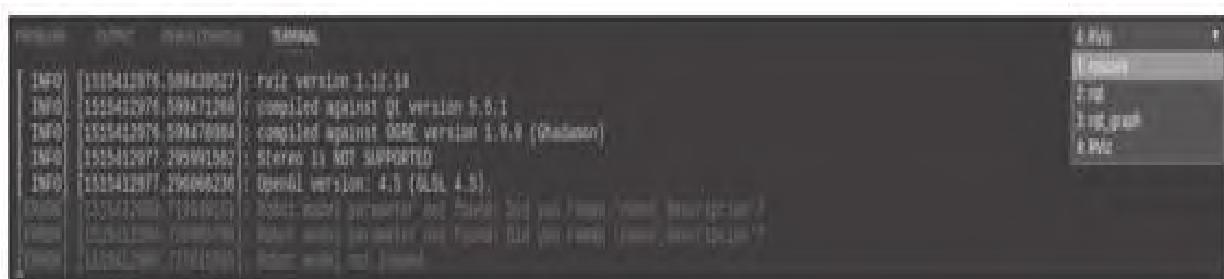
图15-11 ROS工具列表

你可以从该菜单上直接运行roscore或访问这些常用的工具：

- RViz
- rqt
- rqt-reconfigure
- rqt-graph

除此之外，你可以直接在文件编辑器中打开.bashrc文件并手动修改系统配置。另外，你也可以通过选择Run Remote roscore选项来运行远程的roscore。

你可以从终端窗口监测这些ROS命令的执行情况，每个ROS工具会打开一个新终端，一些可视化工具会打开外部窗口。



The screenshot shows a terminal window with several lines of text output. The text includes timestamped messages such as '1967 [1333411287]: 000420927' and various ROS-related status messages like 'ROS version 3.12.10', 'compiled against Qt version 5.6.1', 'status is NOT SUPPORTED', and 'OpenAL version: 4.3 (API: 4.3)'. On the right side of the terminal window, there is a vertical scroll bar and a small menu or status bar with icons.

图15-12 ROS终端窗口

15.1.7 处理活动的ROS话题、节点和服务

要在特定的时间查看系统中活动的ROS话题、节点和服务，请单击左侧边栏的ROS图标。随着roscore形成的信息遍历列表显示在每个框中，我们可以通过单击话题名称而显示每条ROS消息的内容，如图15-13所示。

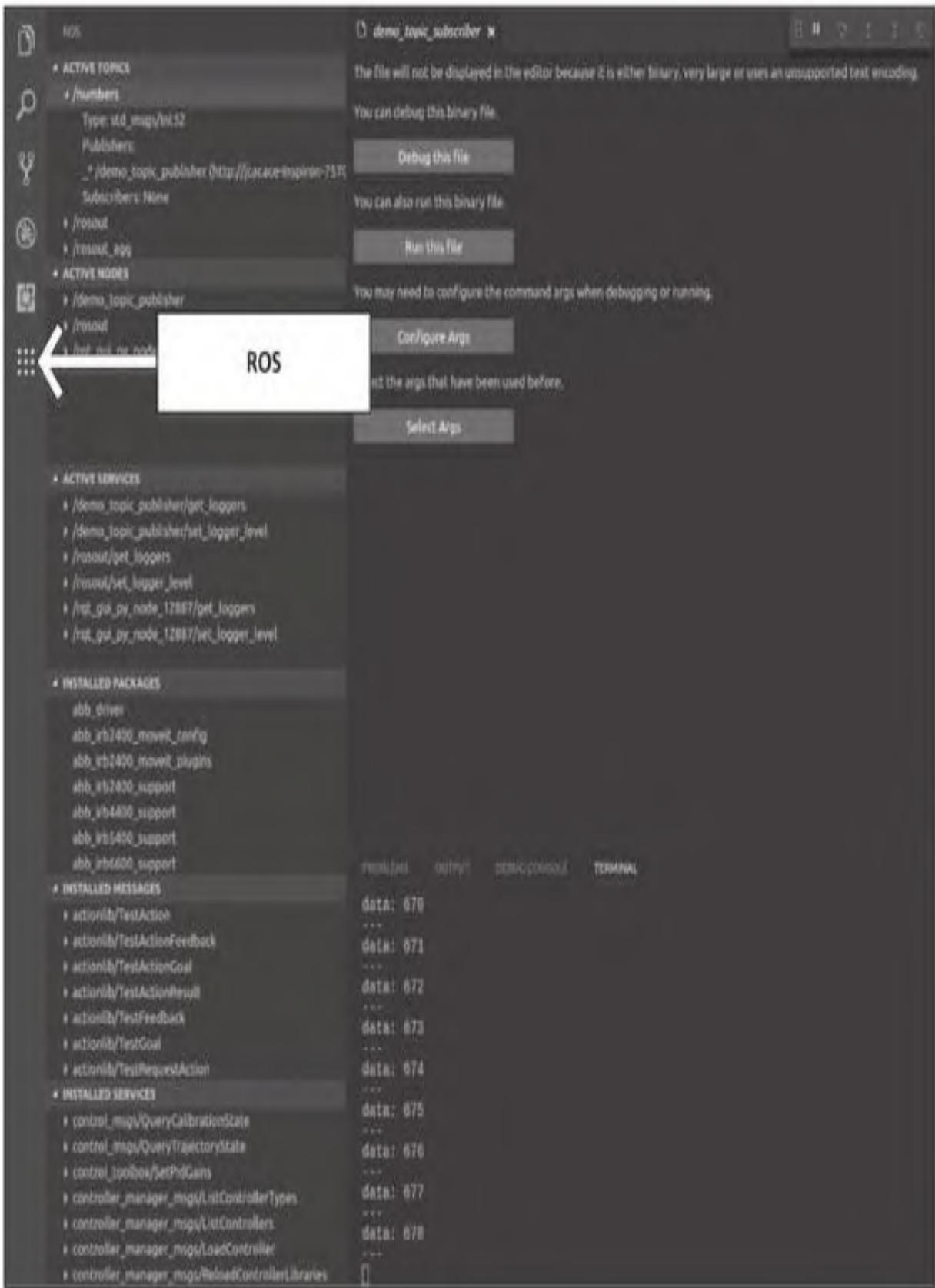


图15-13 RoboWare Studio中的ROS面板

在图15-13所示的示例中，我们选择显示发布在/number话题中 std_msgs::Int32类型的数据。

我们甚至可以在RoboWare中录制和回放ROS日志文件。为了录制日志文件，请单击Active Topics面板旁边的RecordROSTopic按钮，如图15-14所示。单击此按钮，系统中所有活动的话题都将被记录下来。生成的日志文件将保存在工作区的根文件夹下，命名为 yyyy_MM_dd_HH_mm_ss.bag 格式。要停止录制，必须在终端窗口中使用 Ctrl+C。如果你想记录一个或者多个话题，按Ctrl键然后逐一选择它们，最后单击rosbag记录按钮。

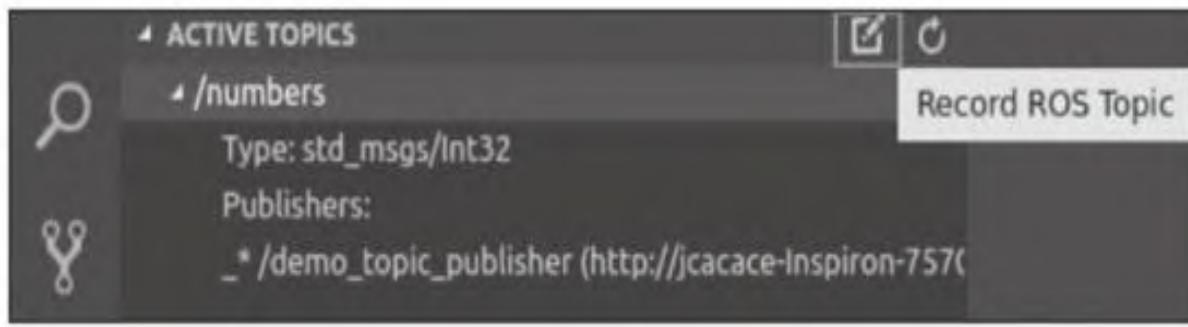


图15-14 记录日志文件

要回放日志文件，你可以在资源管理器窗口右击日志文件名称，然后单击Play Bag File，如图15-15所示。

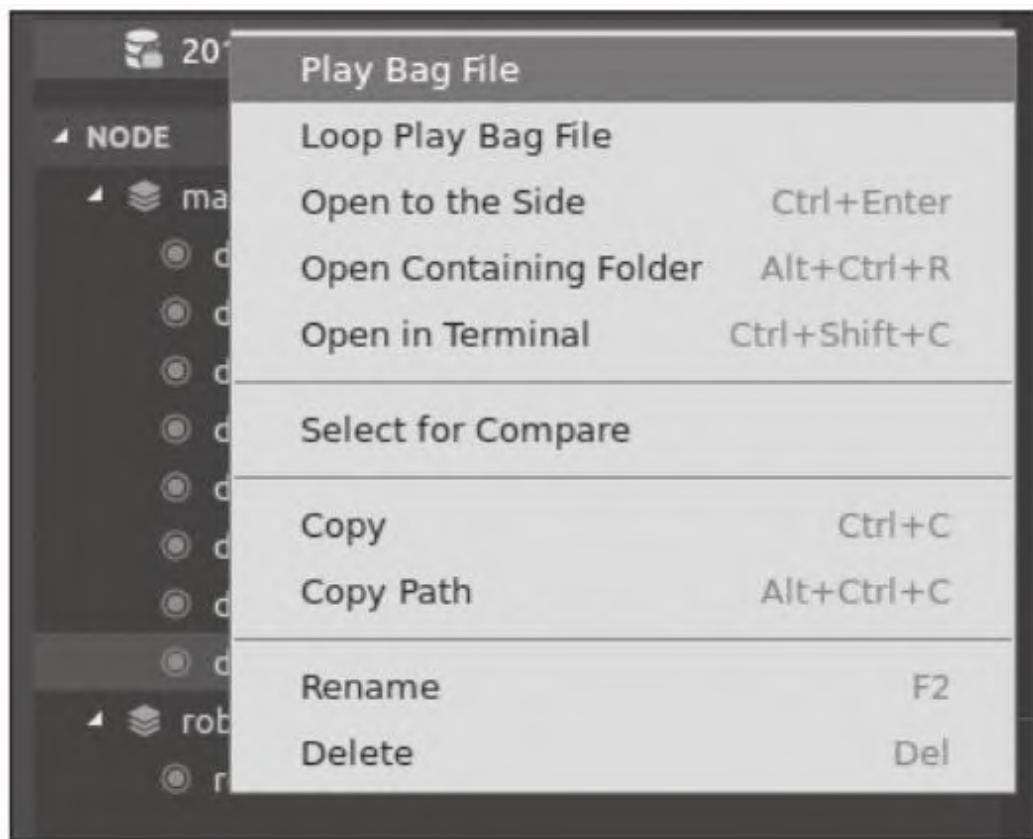


图15-15 回放日志文件

15.1.8 使用RoboWare工具创建ROS节点和类

RoboWare Studio提供了一个向导来创建C++、Python类及ROS节点。要创建ROS节点，请按照下面操作来执行：

- 1) 在软件包名上右击，然后选择Add C++ROSNode或Add PythonROSNode；
- 2) 输入软件包的名称；
- 3) 默认情况下，将创建两个源文件：一个发布者节点和一个订阅者节点的示例。例如，如果你输入chatter作为包名，则会创建一个chatter_pub.cpp文件和一个chatter_sub.cpp文件；
- 4) 编译软件包。CMakeLists.txt文件已根据新创建的节点进行了更新。你也可以根据需要删除发布者或者订阅者。这时，CMakeLists.txt文件也会自动更新。

除了ROS节点，我们还可以用以下方法来创建C++类：

- 1) 在软件包名称上右击；
- 2) 选择Add C++ Class；
- 3) 输入类的名称，例如roboware_class；
- 4) 在include文件夹下将创建roboware_class.h的头文件，同时在src文件夹下将创建roboware_class.cpp的源码文件；
- 5) 选择一个可执行文件来链接刚创建的类，以便将类导入软件包的另一个ROS节点中；
- 6) CMakeLists.txt将自动更新。

15.1.9 RoboWare Studio中的ROS软件包管理器

在RoboWare Studio界面中，我们可以通过ROS软件包管理器面板安装或浏览可用的ROS软件包。要访问此面板，请单击左侧栏的ROSPackages Manager图标。RoboWare将自动检测正在使用的ROS版本以及已安装在ROS软件包路径中的软件包列表。

在此面板中，我们可以浏览ROS仓库中的可用软件包，并且可以在软件包和超软件包之间进行选择。你还可以单击软件包名直接在RoboWare Studio中查看它的维基页面，而且还可以很方便地安装或卸载选定的软件包，如图15-9所示。



图15-16 RoboWare Studio的ROS软件包管理器

15.2 ROS的最佳实战技巧与经验

本节将简要介绍在开发ROS时可以遵循的最佳实战技巧与经验。ROS提供有关其QA（质量保证）流程的详细教程。QA流程是一份详细的开发人员指南，其中包括C++和Python代码风格指南，命名约定等。首先，我们来讨论ROS的C++编码风格。

ROS C++编码风格指南

ROS C++节点遵循的编码风格，使代码具有更好可读性、更易于调试、更易于维护。如果代码风格好，就很容易重用并且对现有代码也很有帮助。在本节，我们将简单学习一些常用的编码风格。

ROS中使用的标准命名约定

在这里我们用HelloWorld来演示我们在ROS中使用的命名模式：

- HelloWorld：这个名字以大写字母开头，每个新单词都以大写字母开头，单词间没有任何空格或下划线。
- helloWorld：在这个命名约定中，第一个字母是小写的，但是后面的单词将以大写字母开头，单词间没有空格。
- hello_world：这个仅包含小写字母。单词间用下划线分隔。
- HELLO_WORLD：所有字母都是大写的。单词间用下划线分隔。

下面是ROS中每个组件遵循的命名约定：

- 软件包、话题/服务、文件、库：这些ROS组件遵循hello_world模式。
- 类/类型：遵循HelloWorld命名约定，例如：class ExampleClass。
- 函数/方法：函数遵循helloWorld命名约定，函数参数命名遵循hello_world模式，例如：void exampleMethod(int sample_arg)。
- 变量：通常情况下，变量遵循hello_world模式。
- 常量：常量遵循HELLO_WORLD模式。
- 成员变量：类中的成员变量遵循hello_world模式，并在尾部添加了下划线后缀，例如：int sample_int_。
- 全局变量：遵循hello_world约定，在头部添加g_作为前缀，例如：int g_samplevar。
- 命名空间：遵循hello_world命名模式。

代码许可协议

我们应该在代码顶部添加许可声明。ROS是一个开源的软件框架，它是BSD许可。下面就是一个许可的代码片段，它必须插入到代码的顶部。你可以从主代码库的任何ROS节点获得许可协议。你还可以从https://github.com/ros/ros_tutorials查看ROS教程中的源代码：

```
*****  
* Software License Agreement (BSD License)  
*
```

```
* Copyright (c) 2012, Willow Garage, Inc.  
* All rights reserved.  
*  
* Redistribution and use in source and binary forms, with or without  
* modification, are permitted provided that the following conditions  
* are met:  
*****
```

有关ROS中各种许可协议的进一步信息，请参阅
<http://wiki.ros.org/Developers%20Guide#Licensing>。

ROS编码格式

开发代码时需要注意的一件事就是它的格式。编码格式的基本注意事项之一是在ROS中的每个代码块都由两个空格分开。下面就是一个展示代码格式的片段：

```
if(a < b)  
{  
    // 做一些操作  
}  
else  
{  
    // 做其他操作  
}
```

下面是一个ROS标准样式的示例代码片段：

```
#include <boost/tokenizer.hpp>
#include <moveit/macros/console_colors.h>
#include <moveit/move_group/node_name.h>

static const std::string ROBOT_DESCRIPTION = "robot_description"; //  
name of the robot description (a param name, so it can be changed  
externally)

namespace move_group
{

class MoveGroupExe
{
public:

    MoveGroupExe(const planning_scene_monitor::PlanningSceneMonitorPtr& psm,  
bool debug) :  
    node_handle_("~")  
{  
    // 如果用户想禁用路径的执行，他们就可以设置 ROS 参数为 false  
    bool allow_trajectory_execution;  
    node_handle_.param("allow_trajectory_execution",  
    allow_trajectory_execution, true);

    context_.reset(new MoveGroupContext(psm, allow_trajectory_execution,  
debug));
}

// 开始配置功能
```

```
    configureCapabilities();  
}  
  
~MoveGroupExe()  
{
```

控制台输出

尽量避免在ROS节点中使用printf或cout打印调试信息。

我们可以用rosconsole (<http://wiki.ros.org/rosconsole>) 在ROS节点中打印调试消息，而不是printf或cout函数。rosconsole还提供了带时间戳的输出消息，自动记录打印的消息，并提供了5种不同级别的信息显示。有关详细的编码样式，可以参考<http://wiki.ros.org/CppStyleGuide>。

15.3 ROS软件包中的最佳实战技巧与经验

下面是创建和维护软件包时必须考虑的要点：

- 版本控制：ROS支持使用Git、Mercurial和Subversion进行版本控制。我们可以在GitHub和BitBucket上托管我们的代码。大部分ROS软件包都是放在GitHub上的。
- 打包：在ROScatkin软件包内有一个package.xml，该文件应包含作者姓名，软件包描述信息和许可。下面就是一个package.xml的示例：

```
<?xml version="1.0"?>
<package>
  <name>roscpp_tutorials</name>

  <version>0.6.1</version>

  <description>
    This package attempts to show the features of ROS step-by-step,
    including using messages, servers, parameters, etc.
  </description>

  <maintainer email="dthomas@osrfoundation.org">Dirk Thomas</maintainer>

  <license>BSD</license>

  <url type="website">http://www.ros.org/wiki/roscpp\_tutorials</url>
  <url type="bugtracker">https://github.com/ros/ros\_tutorials/issues</url>
  <url type="repository">https://github.com/ros/ros\_tutorials</url>
  <author>Morgan Quigley</author>
```

15.4 ROS中的重要调试技巧

现在我们将介绍使用ROS过程中遇到的一些常见问题，以及如何解决这些问题的技巧。

在ROS系统中用于发现问题的内置工具之一是rosout。rosout是一个命令行工具，可用于检查如下领域的问题：

- 环境变量配置
- 软件包或超软件包配置
- 启动文件
- 节点话题通信图

使用rosout

当我们想检查ROS软件包中的问题时，进入软件包内输入rosout即可。我们还可以通过输入以下命令来检查我们的ROS系统中的问题：

```
$ rosout
```

该命令将生成关于系统运行状态的报告，例如，在ROS主机名配置错误时，将生成如图15-17所示的报告。

```
Loaded plugin tf,tfwtf
=====
Static checks summary:

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

WARNING ROS_HOSTNAME may be incorrect: ROS_HOSTNAME [192.168.2.23] resolves to [192.168.2.23], which does
not appear to be a local IP address ['127.0.0.1', '192.168.1.7'].

=====
ROS Master does not appear to be running.
Online graph checks will not be run.
ROS_MASTER_URI is [http://192.168.2.2:11311]
```

图15-17 当ROS主机名配置错误时rosrun的输出

我们还可以在启动文件中运行rosrun来查看潜在的问题：

```
$ rosrun <file_name>.launch
```

rosrun的维基页面是<http://wiki.ros.org/rosrun>。

下面是使用ROS过程中的一些常见错误：

- 问题1：

错误消息：Failed to contact master
at [localhost:11311]. Retrying...，如图15-18所示。

```
jcacace@jcacace-Inspiron-7570:~$ rosrun roscpp_tutorials talker
[ERROR] [1515175271.173829991]: [registerPublisher] Failed to contact ma
ster at [localhost:11311]. Retrying...
```

图15-18 无法与master节点通信的错误消息

解决方案：当在执行ROS节点却没有运行roscore命令或者检查ROS主配置时，会出现此消息。

· 问题2：

错误消息： Could not process inbound connection:topic types do not match，如图15-19所示。

```
jcacace@jcacace-Inspiron-7570:~$ rostopic pub /chatter std_msgs/Int32 "data: 1"
publishing and latching message. Press ctrl-C to terminate
[WARN] [1515176143.614150]: Could not process inbound connection: topic types do not
match: [std_msgs/String] vs. [std_msgs/Int32][{'topic': '/chatter', 'tcp_nodelay': '0'
, 'md5sum': '992ce8a1687cec8c8bd883ec73ca41d1', 'type': 'std_msgs/String', 'callerid'
: '/listener'}]
```

图15-19 消息不匹配的警告信息

解决方案：当话题的消息类型不匹配时会发生这种警告信息，因此我们需要确保发布和订阅的话题有相同的话题消息类型。

· 问题3：

错误消息： Couldn't find executables，如图15-20所示。

```
jcacace@jcacace-Inspiron-7570:~$ rosrun roscpp_tutorials taker
[rosrun] Couldn't find executable named taker below /opt/ros/kinetic/sha
re/roscpp_tutorials
```

图15-20 找不到可执行文件

解决方案：有很多原因可能导致该错误的出现。一个可能的错误是在命令行中输入了错误的可执行文件名或ROS软件包中没有发现可执行文件名称。在这种情况下，我们应该检查一下在CMakeLists.txt文件中生成的可执行文件的名称。

· 问题4：

错误消息： roscore command is not working，如图15-21所示。

```
jcacace@jcacace-Inspiron-7570:~$ roscore
^C... logging to /home/jcacace/.ros/log/5a62571a-f2d2-11e7-9514-9cda3ea0
e939/roslaunch-jcacace-Inspiron-7570-6141.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

图15-21 roscore命令运行不正常

解决方案：可能导致roscore命令挂起的原因是ROS_IP和ROS_MASTER_URI的配置错误。当我们在多台计算机上运行ROS时，每台计算机都必须用自己的IP地址来设置ROS_IP，并将ROS_MASTER_URI设置为运行roscore的计算机的IP地址。如果该IP配置错误，roscore将无法正常运行。因此，解决该问题的方法就是检查这些环境变量配置是否正确。

- 问题5：

错误消息：Compiling and linking errors，如图15-22所示。

```
Base path: /home/jcacace/ros_ws
Source space: /home/jcacace/ros_ws/src
Build space: /home/jcacace/ros_ws/build
Devel space: /home/jcacace/ros_ws/devel
Install space: /home/jcacace/ros_ws/install
#####
##### Running command: "make cmake_check_build_system" in "/home/jcacace/ros_ws/build"
#####
#####
##### Running command: "make -j8 -l8" in "/home/jcacace/ros_ws/build"
#####
[ 50%] Linking CXX executable /home/jcacace/ros_ws/devel/lib/linking_error_test/linking_error
CMakeFiles/linking_error.dir/src/linking_error.cpp.o: In function `main':
/home/jcacace/ros_ws/src/linking_error_test/src/linking_error.cpp:7: undefined reference to `ros::init(int&, char**, std::cxx_11::basic_string<char>, std::char_traits<char>, std::allocator<char> > const&, unsigned int)'
collect2: error: ld returned 1 exit status
linking_error_test/CMakeFiles/linking_error.dir/build.make:104: recipe for target '/home/jcacace/ros_ws/devel/lib/linking_error_test/linking_error' failed
make[2]: *** [/home/jcacace/ros_ws/devel/lib/linking_error_test/linking_error] Error 1
CMakeFiles/Makefile2:493: recipe for target 'linking_error_test/CMakeFiles/linking_error.dir/all' failed
make[1]: *** [linking_error_test/CMakeFiles/linking_error.dir/all] Error 2
Makefile:138: recipe for target 'all' failed
make: *** [all] Error 2
Invoking "make -j8 -l8" failed
```

图15-22 编译和链接错误

解决方案：如果CMakeLists.txt没有编译ROS节点所需的依赖项，那么将会显示这个错误。我们必须检查package.xml和CMakeLists.txt的软件包依赖资源。在这里，我们是通过注释掉roscpp依赖项来产生的该错误，如图15-23所示。

```
cmake_minimum_required(VERSION 2.8.3)
project(linking_error_test)

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

图15-23 缺少依赖项的CMakeLists.txt文件

其他一些调试技巧还可以从ROS维基页面上查到，网址为
<http://wiki.ros.org/ROS/Troubleshooting>。

15.5 习题

- 为什么我们在使用ROS时需要IDE?
- 在ROS中常用的命名约定是什么?
- 为什么在创建软件包时文档工作很重要?
- rosytic命令有什么作用?

15.6 本章小结

在本章，我们学习了如何使用RoboWare Studio IDE，如何在IDE中设置ROS开发环境，如何创建节点和软件包，以及如何管理ROS数据。在RoboWare中配置好ROS后，我们又讨论了一些ROS的最佳实战技巧与经验，例如，在创建ROS软件包时的命名约定、编码风格等。然后我们又学习了ROS的调试技巧。在调试方面，我们讨论了使用ROS时需要牢记的各种调试技巧。