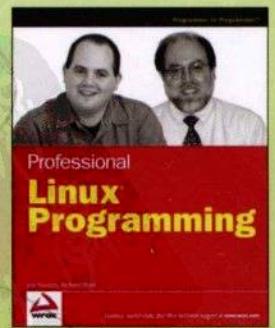


Professional Linux Programming

Linux 高级程序设计

[英] Jon Masters 著
[美] Richard Blum 译
陈健 等 译

- 畅销书《Linux 程序设计（第3版）》后使你更上一层楼的经典著作
- 著名开源技术社区LUPA强烈推荐
- 全面阐述现代Linux程序设计的技术和方法



人民邮电出版社
POSTS & TELECOM PRESS

“本书不是一本适合Linux初学者的指南，有经验的Linux程序员都能从中受益。它深入地阐述了Linux程序设计过程中所涉及的重要知识、技巧和常用工具，让你能更透彻地理解：‘作为一位现代Linux程序员，你究竟需要什么。’”

——著名开源技术社区LUPA (<http://www.lupaworld.com/>) 强烈推荐

“本书出色地为其他平台的程序员揭示了Linux程序设计的复杂本质，而且特别强调了内核开发。为作者喝彩！”

——Linux Magazine杂志

Professional Linux Programming Linux高级程序设计

读了《Linux 程序设计（第3版）》之后还不过瘾？本书将为你献上一顿饕餮大餐！

本书是Linux程序设计领域内的经典著作，涵盖了各种常用的和最重要的Linux程序设计的技术和方法。书中蕴含了作者的宝贵经验，提供了大量的最佳实践。无论你是有开发经验的Linux程序员，还是从其他平台转到Linux上的专业开发者，都能通过本书学到最新的Linux平台开发技术，迅速成为现代Linux程序员。



Jon Masters 著名Linux内核工程师，目前效力于Red Hat公司。13岁取得计算机科学学士学位，创造了英国记录。他精通Linux内核引擎、Unix系统管理、基于Linux的嵌入式系统开发，而且在网络、安全等领域也颇有造诣。目前正在负责维护Module-init-tools——Linux官方的一个工具包，包含所有与Linux内核交互的工具。他还是*Linux User & Developer*、*Linux Magazine*等著名杂志的专栏作家。



Richard Blum 毕业于美国普度大学电气工程专业，资深程序员，精通多种编程语言。除本书外，他还著有*Professional Assembly Language*等经典著作，深受读者好评。

图灵Linux/Unix系列图书阅读路线图



本书相关信息请访问：图灵网站 <http://www.turingbook.com>

读者/作者热线：(010)88593802

反馈/投稿/推荐信箱：contact@turingbook.com

计算机/程序设计/Linux

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-17910-4



9 787115 179104 >

ISBN 978-7-115-17910-4/TP

定价：59.00 元

TP316.81/163

2008

TURING 图灵程序设计丛书 Linux/Unix系列

Professional Linux Programming

Linux

高级程序设计

[英] Jon Masters 著
[美] Richard Blum 著
陈健等 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Linux 高级程序设计 / (英) 美斯特 (Masters, J.),
(美) 布卢 (Blum, R.) 著; 陈健等译. —北京: 人民邮
电出版社, 2008.7

(图灵程序设计丛书)

书名原文: Professional Linux Programming

ISBN 978-7-115-17910-4

I. L… II. ①美…②布…③陈… III. Linux 操作系统—
程序设计 IV. TP316.89

中国版本图书馆 CIP 数据核字 (2008) 第 046781 号

内 容 提 要

本书是 Linux 程序设计领域的一部力作, 讲解了大量程序员需要掌握的关键知识点, 包括 Linux 开发中的基本工具、Linux 系统编程、Linux 桌面开发以及 Linux 与 Web 开发。书中包括大量有益的经验之谈和富于启发的示例。

本书主要针对已有一定 Linux 开发经验或者从其他平台转到 Linux 平台的专业程序员, 同样也适合想更多了解系统以解决实际问题的 Linux 使用者。

图灵程序设计丛书

Linux 高级程序设计

-
- ◆ 著 [英] Jon Masters [美] Richard Blum
 - 译 陈 健 等
 - 责任编辑 杨福川
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
 - 邮编 100061 电子函件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 三河市海波印务有限公司印刷
 - 新华书店总店北京发行所经销
 - ◆ 开本: 800×1000 1/16
 - 印张: 25.25
 - 字数: 660 千字 2008 年 7 月第 1 版
 - 印数: 1~4 000 册 2008 年 7 月河北第 1 次印刷
 - 著作权合同登记号 图字: 01-2007-0946 号
 - ISBN 978-7-115-17910-4/TP
-

定价: 59.00 元

读者服务热线: (010) 88593802 印装质量热线: (010) 67129223

反盗版热线: (010) 67171154

版 权 声 明

Original edition, entitled *Professional Linux Programming* by Jon Masters, Richard Blum, published by Wiley Publishing, Inc. Copyright © 2007 by Wiley Publishing, Inc.

All rights reserved. This translation published under license.

The Wrox Brand trade dress is a trademark of Wiley Publishing, Inc. in the United States and/or other countries. Used by permission.

Translation edition published by Posts & Telecom Press Copyright © 2008 .

本书简体中文版由 Wiley Publishing, Inc. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

Wrox 商标是 Wiley 出版社在美国及其他国家使用的商标，使用经过许可。

版权所有，侵权必究。

译者序

与市面上其他的Linux编程类图书不同，本书是一本专门介绍Linux最新编程技术的图书，它没有像其他图书那样介绍很多Linux和其他类UNIX系统共有的内容，而是专门为Linux程序员所撰写的。

本书涉及面非常广，从基本工具和技术的使用到LAMP技术的介绍，从系统底层内核的剖析到网络、数据库编程，从GNOME桌面环境到图形、音频和游戏编程，可以说涵盖了现代Linux编程的各个方面。作者使用简炼的笔法勾勒出了Linux编程的全景。我在翻译本书时深深为作者在Linux方面的广博学识和深刻见解所折服。

由于本书涉及面非常广泛而篇幅又不长，所以作者采用的是技术概述、要点介绍、示例阐述的方式来组织本书的内容，对每一方面的内容的最新发展动态和技术要点都进行了介绍，并给出了进一步学习和研究所需要参考的内容的链接，但并不进行事无巨细的阐述，要求读者在阅读本书时在感兴趣的内容上参考大量其他的资料以进行进一步的学习。对如今这个信息极为丰富的时代来说，读者缺乏的并不是技术资料和文档，而是如何选择这些资料和更有效地进行学习以跟上Linux编程的最新技术进展，相信本书会为读者打开通向现代Linux编程的大门。

在翻译过程中我对原书中的一些明显错误进行了更正，为了帮助读者理解，还对一些名词、概念进行了译注，但因为本书涉及面非常广，有些领域也并非我所擅长，所以译文中的错误在所难免，真诚地希望读者能提出宝贵意见，以便在本书重印时进行修订。

最后，感谢浙江Linux专业委员会副主任兼著名开源技术社区LUPA (www.lupaworld.com) 的负责人邵炜先生的强烈推荐。同时，还要感谢人民邮电出版社的编辑，正是因为他们始终如一的鼓励和支持，本书的翻译工作才得以顺利地完成。

陈健
2007年冬于南京大学

前 言

Linux近几年来有了很大的发展，已从一个不起眼的小玩意发展到在越来越多的《财富》500强公司中发挥巨大作用。从人们使用的手机到最大型的超级计算机集群，几乎都在使用Linux内核和为Linux编译的软件。但究竟什么是Linux？是什么使得它和目前市场上其他的类UNIX操作系统区别开来的呢？最重要的是，如何才能在我们的软件项目中充分利用Linux的强大功能和广泛使用的自由/开放源码软件（Free, Libre, and Open Source Software，简称FLOSS^①）带来的变革呢？

本书的目的就是为了讨论这些问题以及其他问题。写作本书的目的源自于读者的这样一种需求，即究竟是什么使得Linux如此独一无二，但本书并不是一本适合Linux初学者的指南，因为这样的书早已在市场上存在了。这些年来，作为一位专业的Linux程序员，我们发现一起工作的很多技术精湛的软件工程师都缺乏或没有Linux编程方面的经验。其中一些工程师一直在寻找与本书类似的图书，但最后总是失望而归。为了让读者不再遭受这样的挫折，本书将帮助读者理解Linux社区的强大意义、已确立的软件开发模型和Linux世界中处理事务的方式。

有许多图书声称是专为Linux编程而写的，其中有许多书确实非常出色，但它们往往过于集中地介绍Linux简单继承自其前辈的内容。在本书中你不会发现这些内容，本书不是一本只介绍Linux和其他老版本UNIX系统共有内容的图书，而是一本介绍现代Linux操作系统的图书。本书不仅仅是另外一本UNIX编程类图书，它试图解释为什么Linux这么成功，并向读者展示在这个主题上被其他图书一笔带过或完全忽略的系统中的某些部分。

在本书中，你将学习到是什么推动了Linux的开发过程。你将了解各种各样常被Linux开发人员使用的工具——编译器、调试器和软件配置管理工具，以及这些工具是如何用来构建应用软件、工具甚至Linux内核自身的。你将学习到Linux系统中使其与其他类UNIX系统真正区分开来的特有组件，你还将深入研究Linux系统的内部工作机理，以便更好地理解作为新一代Linux开发人员你所需要扮演的角色。

你将学习一些新颖的开发方法，包括虚拟化技术的使用和交叉编译的使用（一种为不同的兼容平台编译软件的手段）。你还将学习对于一个没有国界的社区来说软件国际化的重要性——Linux是真正国际性的，它的用户也是如此。最后，你将通过为热门的LAMP（Linux、Apache、MySQL、Perl/Python）组合编写软件来学习Linux在现代因特网上的广泛用途。Linux所包含的内容远不只是Linux内核，作为一位Linux开发人员，意识到这一点是非常重要的。

最重要的是，本书将为未来进一步学习打下基础。通过对推动Linux开发的关键主题的深刻讨论，我们将为你打开通向自由/开放源码软件项目世界的大门。在阅读本书之后，你将能更好地明白你究竟

① 2000年，Rishab Ghosh在荷兰创造了FLOSS这个词，libre是法语中的“自由”一词。——译者注

需要了解什么，你并不会在本书中找到所有的答案，但你将具备自己发现这些答案的能力。不论你是使用Linux编写自由软件还是参与一个大型商业软件项目，你都将在阅读本书中有所收获。

读者对象

本书为两类不同的读者服务。首先，本书面向的是准备转向Linux开发平台的程序员，这类读者已经熟悉C编程语言，并理解了编译器、链接器和调试器等基本概念。他们有可能已看过这方面的介绍性图书，例如，Wrox的*Beginning Linux Programming*（Wiley 2004）^①，但却缺乏实践经验。

对于那些从事专业Linux软件开发的新手而言，本书的内容安排非常有利于学习。你可以按顺序逐章阅读全书，也可以有选择地跨过与内核相关的章节（第7~9章），而把学习重点放在每天的项目中都会用到的与更高层次应用程序和工具相关的章节。你還将在本书中找到工具链（Toolchain）、可移植性和有特定用途的SCM（Software Configuration Management，软件配置管理）的背景知识。

对于那些已在他们的日常生活中使用Linux并且想要深入了解一个典型的Linux系统的内部工作原理，而又不需要开发软件的Linux爱好者、管理人员和其他有关人员来说，本书也包含他们感兴趣的内容。现代Linux系统是如何进行硬件检测的？为什么Linux内核不提供设备驱动程序模块？Linux是如何支持国际化的？许多类似的问题都可以在本书中找到答案。

对于那些已经有Linux使用经验的读者来说，并不需要阅读本书的全部内容，但可能也会在每一章中发现一些新鲜和有趣的内容。我们通常会在脚注和注释中包括一些你可能在以前没有遇到过的示例和建议，其中包括从其他人的经验中获取的轶事和教训。你可能会选择投入更多的精力在本书后面讨论Linux内核、桌面和LAMP的章节中。

最后，不管你是一位对Linux或UNIX有基本了解同时又希望开扩视野的微软的Windows开发人员，还是一位从过去的岁月走过来的执着的UNIX程序员，希望了解是什么使得Linux如此成功，本书都会对你有所帮助。

主要内容

本书涵盖了各种各样用于Linux软件开发和软件本身的技术，包括现代UNIX、类UNIX和Linux系统的背景知识，从一个平台到另一个平台的软件可移植性以及有助于在现代的Linux软件发行中实现这一目标的工具。你将学习到如何通过网络接口、图形化用户环境、复杂的基于Web的现代LAMP组合来与Linux系统进行交互，甚至将学习到如何扩展Linux内核本身。在本书中你将学习到的是现代Linux的开发技术。

本书的内容反映了写作时技术的最新发展水平，但软件的版本却在不断变化。因此，本书讨论的大多数主题并不要求使用某个特定版本的工具、源代码或发行包。如果需要，我们会在书中指出，否则，你可以假设书中的示例可以运行在任一个你所使用的最新的Linux发行版本上。

组织结构

本书大致分为4个部分。在第一部分中，你将了解一些基本的工具和技术，目的是让你（作为一

^① 中文版《Linux程序设计》（第3版）已由人民邮电出版社出版。——编者注

位专业的Linux程序员）的生活更加轻松。你将了解GNU工具链、软件可移植的重要性和软件国际化的需求，以及许多其他的主题，这些内容能帮你提高软件项目开发的效率。你可能想先阅读这些内容，并会经常查阅它。

本书的第二部分介绍了一个典型的Linux系统的底层部分，即传统的系统编程的主题，包括网络、数据库概念和Linux内核。可以通过阅读这些内容来更好地理解你所感兴趣的主題，但你并不需要学习这些章节中的所有內容，特別是本书中关于Linux内核部分的内容，因为本书并不是一本Linux内核编程类的图书，但通过对这些内容的学习确实会让你更想继续深入学习。

在本书的第三部分中，你将了解一些更高级的概念，如GNOME桌面环境及其数量庞大的软件库。你将学习自由桌面项目（Free Desktop Project），并有机会利用Gstreamer库的强大功能编写一个简单的CD播放器应用程序，该函数库被用于现代GNOME桌面多媒体应用程序的开发。你将会发现有多少代码可以通过软件复用来实现，并获得一些编写自己的GNOME软件的深刻体会。

本书的最后一章专门介绍LAMP。通过基于商品化的软件栈并使用Linux、Apache、MySQL和Perl/Python来编译，LAMP允许你只使用自由开放源码的软件就可以编写功能非常强大的Web应用程序。该章将介绍这些组件中的每一部分并提供一些使用示例。

排版约定

为了帮助您更好地理解本书的内容并清楚发生了什么，我们贯穿全书使用了一些排版约定。

像这样的文本框用于放置重要的、不应被忘记的信息，这些信息和上下文的内容直接相关。

与当前讨论内容有关的技巧、提示、诀窍会单独列出并以楷体显示。

文本的格式如下所示。

- 在介绍新的术语和重要的词汇时以**黑体字**显示它们。
- 以类似“Ctrl+A”的格式来显示组合键。
- 以类似的格式来显示文本中的文件名、URL和代码。
- 以两种不同的方式来显示代码：

在代码示例中，以灰色背景来高亮显示新的和重要的代码。

对当前上下文并不重要的或已经显示过的代码不会以灰色高亮显示。

源代码

当你阅读本书中的示例时，可能会选择手工敲入所有代码或使用随本书附带的源代码文件。本书中使用的所有源代码都可以通过网址<http://www.wrox.com>下载^①。访问该网址后，只需简单地定位本书的书名（使用搜索栏或书名列表），然后点击图书详细页面中的Download Code（下载代码）链接就可以获得本书的所有源代码。

^① 源代码也可以在图灵网站www.turing book.com本书网页免费注册下载。——编者注

下载了源代码之后，只需使用你所喜欢的压缩工具解压缩它即可。此外，你也可以访问Wrox的主代码下载页面<http://www.wrox.com/dynamic/books/download.aspx>来查看本书的代码和所有Wrox出版的其他图书的代码。

勘误

我们已尽最大努力来确保本书的文字或代码中没有错误。然而，没有人是完美的，错误在所难免。如果你发现了本书中的错误，如拼写错误或错误的代码段，我们将非常感谢您的反馈意见。通过向我们发送勘误表，你不仅节省了其他读者困惑的时间，同时也有助于我们提供更高质量的信息。

要找到本书的勘误页，请访问<http://www.wrox.com>并使用搜索栏或书名列表来定位本书。然后，在本书的详细页面中，点击Book Errata（图书勘误）链接。在勘误页中，你将会看到由Wrox编辑发表的关于本书的所有勘误。一个包括每本图书勘误表的完整图书列表也可以通过网址www.wrox.com/misc-pages/booklist.shtml访问到。

如果在本书的勘误页中没有发现你所找到的错误，你可以通过访问网址www.wrox.com/contact/techsupport.shtml并填写上面的表格来向我们发送你所找到的错误。我们将核查该信息，如果它是正确的，我们就会发送消息到本书的勘误页并在本书的后续版本中更正该问题。

p2p.wrox.com

如果你希望和本书的作者以及其他进行讨论，请加入p2p.wrox.com上的P2P论坛。这个论坛是一个基于Web的系统，你可以在上面发表与Wrox图书以及相关技术有关的文章，并与其他读者和技术人员进行交流。这个论坛提供了订阅功能，当有新的文章发表时，论坛会将你所选择感兴趣的主題通过E-mail发送给你。Wrox的作者、编辑、其他的行业专家和读者都加入了这些论坛。

你将会在网址<http://p2p.wrox.com>上发现许多不同的论坛，它们不仅对你阅读本书有益，而且还将帮助你开发自己的应用程序。要加入这些论坛，请遵循如下步骤：

- (1) 访问p2p.wrox.com并点击Register（注册）；
- (2) 阅读使用条款并点击Agree（同意）；
- (3) 填写加入论坛所必需的信息以及你希望提供的一些可选信息，然后点击Submit（提交）；
- (4) 将收到一封电子邮件，邮件内容描述了如何验证你的账号并完成加入论坛的流程。

不加入P2P，你也可以阅读论坛中的文章，但为了发表你自己的文章，你必须加入论坛。

在加入论坛之后，你就可以发表新的文章和回复别人发表的文章。你可以在任何时间通过Web阅读文章。如果你想要将某个特定论坛上新发表的文章通过E-mail发送给你，请点击论坛列表中该论坛名称前面的Subscribe（订阅）链接。

如果你想了解使用Wrox P2P的更多信息，请阅读P2P FAQ，它包含针对论坛软件工作情况以及许多针对P2P和Wrox图书的常见问题的解答。要阅读这份FAQ，请点击任意一个P2P页面上的FAQ链接。

致谢

值我生日之际，我坐在这里写这篇致谢。在去年的无数个漫漫长夜中，我伏案安排进度，制定计

划，偶尔甚至亲自完成一部分写作。当我承诺写作本书时，我并没有完全意识到完成这样一本书所需要的工作量和所需要克服的困难。我刚开始写作本书时还住在伦敦城外，在写作期间，我已决定离开英国，不到一年之后，我在位于美国马萨诸塞州的剑桥新家完成了本书。过去的一年不论对我个人而言，还是对我的职业而言，都发生了很多的变化，但在我的朋友和家人的支持下，我顺利地克服了这些困难。

首先，我要感谢Wiley出版社中和我一起工作的这个团队——Debra、Adaobi、Kit、Howard和Carol，以及许多其他参与本书出版工作的人员。我要特别感谢Kit Kemper，他忍受了我的写作时间表并使它正好在结束之前完成，而Debra Williams-Cauley从一开始就坚信这个写作计划是一个很好的主意。Howard Jones作为我的编辑帮助我保证内容的准确，出色地完成了他的工作。本书的诞生离不开在我的好朋友——我在共振仪器公司（后来的牛津仪器公司）的前任老板Malcolm Buckingham和Jamie McKendry的启发下产生的灵感，他们过去经常抱怨缺乏Linux专用的编程图书。同样地，如果没有来自我的几位好朋友——Kat、David Goodwin、Matthew Walton和Chris Aillon的贡献，本书也不会出现。谢谢你们，也谢谢Richard Blum，当我已显然不能按时完成本书时，你出现并加入了我们的团队，你做了大量的工作，我真的非常感谢你。

这一路上，还离不开来自我的幸福家庭的帮助——我的父母Paula和Charles、我的姐姐Hannah Wrigley和Holly、我的姐夫Joe，偶尔还有由我的祖母启发的灵感。我还受益于其他与我最要好的朋友，由于人数太多，这里就不一一列出了，但我要特别提到下面这些朋友：Hussein Jodiyawalla、Johannes Kling、Ben Swan、Paul Sladen、Markus Kobler、Tom Hawley、Sidarshan Guru Ratnavelli、Chris和Mad Ball（还有他的猫Zoe）、Emma Maule、John和Jan Buckman、Toby Jaffey和Sara、Sven Thorsten-Dietrich、Bill Weinberg、Daniel James、Joe Casad、Andrew Hutton和Emilie。我还要特别感谢我在Red Hat公司的所有朋友，我的老板和所有其他努力工作的同事，是你们使得我们的公司成为这个世界上最好的工作场所。Red Hat公司真正理解Linux开发的内涵，我非常感谢拥有这样一个优越的工作环境，它以真正的Linux社区精神鼓励参与这样一个项目——谢谢你们，你们太棒了！

最后，我要感谢来自Karin Worley的友谊，你为我提供了充分的机会在这个项目的最后阶段争取更多的时间。Karin，如果没有最近进入我的生活的这一新的幸福感，我不能确信是否能顺利地完成本书。

Jon Masters
剑桥，马萨诸塞

非常感谢Wiley出版社中为这个项目做出突出贡献的伟大团队。感谢组稿编辑Kit Kemper给我这个机会参与本书的写作。还要感谢加工编辑Howard Jones使一切步上正轨，并帮助使本书更加得体。我还要感谢Waterside产品公司的Carole McClendon为我安排了这一次机会，并一直在我的写作期间给予帮助。

最后，我要感谢我的父母Mike和Joyce Blum在我的成长过程中给予我的奉献和支持，感谢我的妻子Barbara和女儿Katie Jane、Jessica的爱、忍耐和理解，特别是当我在进行写作时。

Richard Blum

目 录

第 1 章 Linux 简介	1	
1.1 Linux 发展简史	1	
1.1.1 GNU 项目	2	
1.1.2 Linux 内核	2	
1.1.3 Linux 发行版	3	
1.1.4 自由软件与开放源码	4	
1.2 开发起步	5	
1.2.1 选择一个 Linux 发行版	5	
1.2.2 安装 Linux 发行版	7	
1.2.3 沙盒和虚拟化技术	13	
1.3 Linux 社区	13	
1.3.1 Linux 用户组	14	
1.3.2 邮件列表	14	
1.3.3 IRC	14	
1.3.4 私有社区	14	
1.4 关键差别	15	
1.4.1 Linux 是模块化的	15	
1.4.2 Linux 是可移植的	15	
1.4.3 Linux 是通用的	15	
1.5 本章总结	16	
第 2 章 工具链	17	
2.1 Linux 开发过程	17	
2.1.1 使用源代码	18	
2.1.2 配置本地环境	18	
2.1.3 编译源代码	19	
2.2 GNU 工具链的组成	20	
2.3 GNU 二进制工具集	29	
2.3.1 GNU 汇编器	29	
2.3.2 GNU 连接器	30	
2.3.3 GNU objcopy 和 objdump	31	
2.4 GNU Make	33	
2.5 GNU 调试器	34	
2.6 Linux 内核和 GNU 工具链	37	
2.6.1 内联汇编	37	
2.6.2 属性标记	38	
2.6.3 定制连接器脚本	38	
2.7 交叉编译	39	
2.8 建立 GNU 工具链	40	
2.9 本章总结	41	
第 3 章 可移植性	42	
3.1 可移植性的需要	42	
3.2 Linux 的可移植性	44	
3.2.1 抽象层	44	
3.2.2 Linux 发行版	45	
3.2.3 建立软件包	49	
3.2.4 可移植的源代码	61	
3.3 硬件可移植性	78	
3.3.1 64 位兼容	78	
3.3.2 字节序中立	79	
3.3.3 字节序的门派之争	81	
3.4 本章总结	81	
第 4 章 软件配置管理	83	
4.1 SCM 的必要性	83	
4.2 集中式开发与分散式开发	84	
4.3 集中式工具	85	
4.3.1 CVS	85	
4.3.2 Subversion	93	
4.4 分散式工具	96	
4.4.1 Bazaar-NG	96	
4.4.2 Linux 内核 SCM	99	
4.5 集成化 SCM 工具	102	
4.6 本章总结	104	

2 目 录

第 5 章 网络编程	105	7.2 内核概念	174
5.1 Linux 套接字编程	105	7.2.1 一句警告	175
5.1.1 套接字	105	7.2.2 任务抽象	175
5.1.2 网络地址	107	7.2.3 虚拟内存	179
5.1.3 使用面向连接的套接字	108	7.2.4 不要恐慌	182
5.1.4 使用无连接套接字	114	7.3 内核编程	182
5.2 传输数据	117	7.4 内核开发过程	185
5.2.1 数据报与字节流	117	7.4.1 git: 傻瓜内容跟踪器	185
5.2.2 标记消息边界	121	7.4.2 Linux 内核邮件列表	187
5.3 使用网络编程函数库	123	7.4.3 “mm”开发树	189
5.3.1 libCurl 函数库	123	7.4.4 稳定内核小组	189
5.3.2 使用 libCurl 库	124	7.4.5 LWN: Linux 每周新闻	189
5.4 本章总结	129	7.5 本章总结	190
第 6 章 数据库	130	第 8 章 内核接口	191
6.1 持久性数据存储	130	8.1 什么是接口	191
6.1.1 使用标准文件	130	8.2 外部内核接口	192
6.1.2 使用数据库	131	8.2.1 系统调用	193
6.2 Berkeley DB 软件包	133	8.2.2 设备文件抽象	197
6.2.1 下载和安装	133	8.2.3 内核事件	210
6.2.2 编译程序	134	8.2.4 忽略内核保护	211
6.2.3 基本数据处理	134	8.3 内部内核接口	215
6.3 PostgreSQL 数据库服务器	143	8.3.1 内核 API	215
6.3.1 下载和安装	144	8.3.2 内核 ABI	216
6.3.2 编译程序	145	8.4 本章总结	217
6.3.3 创建一个应用程序数据库	145	第 9 章 Linux 内核模块	218
6.3.4 连接服务器	147	9.1 模块工作原理	218
6.3.5 执行 SQL 命令	150	9.1.1 扩展内核命名空间	220
6.3.6 使用参数	157	9.1.2 没有对模块兼容性的保证	221
6.4 本章总结	160	9.2 找到好的文档	221
第 7 章 内核开发	161	9.3 编写 Linux 内核模块	223
7.1 基本知识	161	9.3.1 开始之前	223
7.1.1 背景先决条件	161	9.3.2 基本模块需求	223
7.1.2 内核源代码	162	9.3.3 日志记录	226
7.1.3 配置内核	165	9.3.4 输出的符号	227
7.1.4 编译内核	168	9.3.5 分配内存	228
7.1.5 已编译好的内核	171	9.3.6 锁的考虑	236
7.1.6 测试内核	172	9.3.7 推迟工作	243
7.1.7 包装和安装内核	174	9.3.8 进一步阅读	251

9.4 分发 Linux 内核模块	252	12.1.1 什么是 D-Bus	300
9.4.1 进入上游 Linux 内核	252	12.1.2 D-Bus 基础	301
9.4.2 发行源代码	252	12.1.3 D-Bus 方法	304
9.4.3 发行预编译模块	253	12.2 硬件抽象层	308
9.5 本章总结	253	12.2.1 使硬件可以即插即用	308
第 10 章 调试	254	12.2.2 HAL 设备对象	311
10.1 调试概述	254	12.3 网络管理器	316
10.2 基本调试工具	255	12.4 其他自由桌面项目	317
10.2.1 GNU 调试器	255	12.5 本章总结	318
10.2.2 Valgrind	263	第 13 章 图形和音频	319
10.3 图形化调试工具	264	13.1 Linux 和图形	319
10.3.1 DDD	264	13.1.1 X 视窗	319
10.3.2 Eclipse	267	13.1.2 开放式图形库	321
10.4 内核调试	269	13.1.3 OpenGL 应用工具包	321
10.4.1 不要惊慌！	269	13.1.4 简单直接媒介层	322
10.4.2 理解 oops	270	13.2 编写 OpenGL 应用程序	322
10.4.3 使用 UML 进行调试	272	13.2.1 下载和安装	323
10.4.4 一件轶事	275	13.2.2 编程环境	323
10.4.5 关于内核调试器的注记	276	13.2.3 使用 GLUT 库	324
10.5 本章总结	276	13.3 编写 SDL 应用程序	336
第 11 章 GNOME 开发者平台	277	13.3.1 下载和安装	336
11.1 GNOME 函数库	277	13.3.2 编程环境	337
11.1.1 Glib	277	13.3.3 使用 SDL 库	337
11.1.2 GObject	277	13.4 本章总结	347
11.1.3 Cairo	278	第 14 章 LAMP	348
11.1.4 GDK	278	14.1 什么是 LAMP	348
11.1.5 Pango	278	14.1.1 Apache	349
11.1.6 GTK+	278	14.1.2 MySQL	349
11.1.7 libglade	279	14.1.3 PHP	349
11.1.8 GConf	279	14.1.4 反叛平台	350
11.1.9 GStreamer	279	14.1.5 评价 LAMP 平台	350
11.2 建立一个音乐播放器	280	14.2 Apache	351
11.2.1 需求	280	14.2.1 虚拟主机	352
11.2.2 开始：主窗口	280	14.2.2 安装和配置 PHP 5	353
11.2.3 建立 GUI	282	14.2.3 Apache Basic 认证	353
11.3 本章总结	299	14.2.4 Apache 与 SSL	354
第 12 章 自由桌面项目	300	14.2.5 SSL 与 HTTP 认证的整合	355
12.1 D-BUS：桌面总线	300	14.3 MySQL	355

14.3.1 安装 MySQL	355	14.4.7 参数处理	377
14.3.2 配置和启动数据库	356	14.4.8 会话处理	378
14.3.3 修改默认密码	356	14.4.9 单元测试	378
14.3.4 MySQL 客户端接口	356	14.4.10 数据库和 PHP	380
14.3.5 关系数据库	357	14.4.11 PHP 框架	380
14.3.6 SQL	357	14.5 DVD 库	381
14.3.7 关系模型	359	14.5.1 版本 1：开发者的噩梦	381
14.4 PHP	362	14.5.2 版本 2：使用 DB 数据层的 基本应用程序	382
14.4.1 PHP 语言	362	14.5.3 版本 3：重写数据层，添加 日志记录和异常	385
14.4.2 错误处理	369	14.5.4 版本 4：应用模板框架	388
14.4.3 异常错误处理	370	14.6 本章总结	390
14.4.4 优化技巧	371		
14.4.5 安装额外的 PHP 软件	375		
14.4.6 日志记录	376		

Linux 简介

为 Linux 编写软件的一个最大障碍是弄明白 Linux 是什么和它不是什么。不同的人对 Linux 有着不同的理解。虽然如今大多数用户都随意地将整个基于 Linux 的系统称为 Linux，但从技术角度来说，Linux 本身是由芬兰人 Linus Torvalds 编写的一个操作系统内核。在短短几年时间里，Linux 迅速发展并被全球一些最大的企业和最强大的计算机用户所广泛接受。

Linux 现在已成为一个提供高收益和企业质量的操作系统。它既用于一些最大型的超级计算机，也用在许多你根本不会想到的底层由 Linux 支持的最小型装置中。然而，这样一个在现代计算机领域中流行的大品牌却并不属于任何一家公司。Linux之所以会这么成功，是因为有数以千计来自世界各地的开发者在坚持不懈地努力来完善它。这些开发者和你一样，对编写高质量的软件有浓厚的兴趣，并从 Linux 社区中获取他人的经验。

不管 Linux 对你意味着什么，你选择本书是因为你想了解更多的有关如何成为一位专业 Linux 程序员的知识。当你准备开始这次学习之旅时，你将发现如果你对不同版本的 Linux 系统有所了解，知道如何开始对它们进行开发，并且清楚 Linux 开发和目前市场上其他流行平台的开发有何不同，将会对你的学习有很大的帮助。如果你已是一位 Linux 专家，那么只需略读本章即可。如果你想成为一位 Linux 专家，本章将会为你提供一些有用的指导。

在本章中，你将站在专业程序员的角度来学习什么是 Linux 以及 Linux 发行版的各个组件是如何组合在一起的。你将学习 Linux 系统上所用的大多数自由/开放源码软件（FLOSS）的开发过程，并找到大量提供开放源码革命动力的在线社区。最后，你还将了解到 Linux 与你之前遇到的其他操作系统的一些不同之处——我们将在本书的其余部分介绍更多这方面的内容。

1.1 Linux 发展简史

Linux 有一个非常多样而有趣的历史，它的历史可能比你最初想象的要早得多。事实上，Linux 的继承历史跨越了 30 年，可以从 20 世纪 70 年代最早的 UNIX 系统算起。这一事实不只是与执着的 Linux 狂热者有关，对读者来说也很重要，因为它至少让你对目前接触到的现代 Linux 系统的独特历史有一个大致的了解。通过介绍这些历史能让你更好地理解将 Linux 和目前市场上的其他操作系统区分开来的细节特征，并有助于使 Linux 的开发更加有趣。

Linux 本身的工作最早始于 1991 年夏天，但早在 Linux 存在之前，就有了 GNU 项目。这个项目已花费了 10 多年的时间来创建很多必要的自由软件组件，其目的就是为了能创建一个完全自由的操作系统，如 Linux。如果没有 GNU 项目，就不会诞生 Linux；同样，如果没有 Linux，你可能也不会立刻去阅

读GNU项目。这两个项目彼此之间互相受益，正如你将要在本书所要讨论的主题中发现的那样。

1.1.1 GNU项目

1983年，那时的Richard Stallman（也称为RMS）还在麻省理工学院的人工智能实验室工作。直到那时为止，许多软件应用程序还是以源代码的形式提供，或有源代码可用，以便用户在必要的时候对自己的系统进行修改。但从那时开始，已存在一种日益增长的趋势，即软件厂商只发行二进制版本的软件应用程序。软件的源代码很快变成了公司的“商业机密”，并受到高度保护——开放源码的开发者通常将这些源代码称为“秘笈”。

GNU项目最初的目标是通过使用必要的工具从源代码开始创建一个自由的类UNIX操作系统。该项目花了10多年的时间创建了所需的大多数工具，包括GCC编译器、GNU emacs文本编辑器和数十个其他的工具和文档。其中许多工具都以它们的高品质和丰富的功能而闻名，例如GCC和GNU调试器。

GNU享有许多早期的成就，但它在20世纪80年代缺少了一个最关键的组件。它没有自己的内核，即操作系统的内核，而是需要用户在已有的商业操作系统，如专有的UNIX上安装GNU工具。虽然这并不会对许多在他们自己的专用系统上使用GNU工具的用户造成什么影响，但如果GNU项目没有自己的内核，它就不是一个完整的项目。在Linux出现之前，针对是否开发这样一个内核（如发展中的GNU HURD）的激烈争论长期以来一直存在着。

Linux从来没有真正成为Richard Stallman所设想的GNU操作系统的一部分。事实上，尽管Linux已成为新一代用户和开发者的宠儿，并且是迄今为止更受欢迎的内核，GNU项目还是一直主张在其概念性的GNU系统中使用GNU HURD微内核。尽管如此，仍然会偶尔看到在提及一个完整的Linux系统时使用术语“GNU/Linux”，这是对在构建和运行任何一个现代Linux系统中扮演重要角色的众多GNU工具的认可。

1.1.2 Linux内核

Linux内核的诞生远晚于GNU项目本身，在Richard Stallman发表他的最初宣言之后的10多年后它才出现。在此之前，已有一些可供选择的操作系统被开发出来，包括HURD微内核（在狂热的内核开发者社区之外只赢得了有限的大众关注）和由Andrew Tanenbaum编写的用于教学目的的Minix微内核。但由于种种原因，在Linux初次登台之前没有一个系统在这一段最好的时机中获得一般计算机用户的广泛认可。

就在那时，一位在赫尔辛基大学读书的年轻的芬兰学生正受制于Minix操作系统中很多他认为不合理的地方^①。于是，他开始专门为他的（当时很高级的）AT 386微机设计自己的操作系统。此人就是Linus Torvalds，他将继续领导这个已创造了整个Linux产业的项目并激励着新一代。

Linus在1991年夏天开发出Linux的最初版本后，就在Usenet新闻组comp.os.minix中发表了如下的公告：

日期：25 Aug 91 20:57:08 GMT

组织：赫尔辛基大学

^① 这些问题持续了数年之久，并成为早期Minix和Linux新闻组中大多数争论的主要话题。但在以后的几年中，争论渐渐平息，因为Linux已在市场上占据主导地位，而Minix及其后继者则继续为那些构思未来操作系统设计的人扮演学术研究的角色。

所有使用minix的人们——我正在为386(486)AT微机开发一个(免费的)操作系统(只是个人爱好,它不会像gnu那样庞大和专业)。我从4月份开始酝酿该系统,现在已进入准备阶段。我需要任何喜欢或不喜欢minix的朋友的反馈意见,因为我的操作系统与它有些类似(同样的文件系统物理布局(由于某些实际原因)以及其他方面)。

我已将bash(1.08)和gcc(1.40)移植到该系统中,并且它们看起来可以正常工作。这意味着在短短几个月内我就能够在该系统中做一些实际的事情,我很想知道大多数人都希望增加哪些功能。欢迎大家提出任何建议,但我无法保证一定会实现它们。

尽管Linus一开始很谦逊,但人们对Linux内核的兴趣却在全球迅速扩大。不久之后,Linux就推出了多个新版本,并且一个日益增长的用户群体(他们基本上都是开发人员,因为即使是简单地安装Linux也需要大量的专业技术)正在为Linux解决各种技术难题并将一些新的构思和想法付诸实现。现在的许多大名鼎鼎的Linux开发者都是在当时加入这一行业的。他们享受着能够在一个现代的完全免费的类UNIX系统上工作的乐趣,而不用忍受像其他系统那样的设计复杂性。

Linux的开发者利用许多已有的GNU工具来构建Linux内核并为它开发新的特性。事实上,在Linux出现后不久,就有越来越多的人对它发生了兴趣,Minix用户也开始切换到Linux系统上来工作——这些事情最终导致在Minix创造者Andrew Tanenbaum和Linus Torvalds之间发生了一系列著名的“口水战”。Tanenbaum至今仍坚持认为Linux的设计根本不如Minix。从理论上来说,这可能是事实,但对其他现代操作系统而言,可以说也存在着同样的问题。

读者可以从Peter H. Salus所著的*A Quarter Century of UNIX*(Addison-Wesley, 1994)中了解到更多有关Linux和其他类UNIX操作系统历史继承的信息。

1.1.3 Linux 发行版

随着Linux内核越来越受欢迎,人们希望Linux系统也能为那些对其内部编程机理没有深入了解的用户提供很好的服务。为了创建这样一个可用的Linux系统,需要的不仅仅只是一个Linux内核。事实上,如今一个普通的Linux桌面系统为了能提供从系统加电到具备丰富功能的图形桌面环境(如GNOME),它利用了成千上万个独立的软件程序。

当Linux第一次发布时,它还没有那么多丰富的软件可用。事实上,Linus开始时只有一个应用程序——GNU Borne Again SHell(bash)。那些曾经以受限的“单用户”模式(只运行一个bash shell)启动过Linux或UNIX系统的用户都知道这是一种什么体验。Linus使用一个单一的bash命令行shell对早期的Linux做了大量的测试,但是,即便这样一个单一的bash也不能直接运行在Linux系统上。它首先需要经过移植或修改才能从一个已有的系统(如Minix)转换到Linux系统上运行。

随着越来越多的人开始使用Linux并为Linux开发软件,那些有耐心编译和安装软件的用户已有大量的软件可用。但随着时间的流逝,人们发现以一种从无到有的方式来构建每一个Linux系统显然是一种不能忍受、不可复加的梦魇,除了最有热情的用户以外,它将阻止所有其他用户体验Linux所提供的功能。解决方法是使用Linux发行版的形式或将预先创建好的软件集以及Linux内核以软盘(或之后的光盘)形式提供给广泛的潜在用户群。

早期的Linux发行版只是简单地为那些不想自己从无到有构建整个系统的用户提供便利。它并没有跟踪系统上已安装了哪些软件或对软件的安全删除以及新软件的添加做出处理。直到包(package)

管理软件如Red Hat的RPM和Debian的dpkg的出现，才使得不具备详尽专业知识的普通用户安装Linux系统成为可能。当你在本书后续章节中学习如何为Linux发行版构建自己的Linux软件包时，将会了解到更多有关软件包管理的内容。

现代的Linux发行版的规模和大小各不相同，并且针对不同的市场。有些版本针对的是常规的桌面型Linux用户；有些版本针对的是企业级用户，他们要求操作系统具备可扩展性和健壮的性能；有些版本甚至是专为嵌入式设备如PDA、手机和机顶盒设计的。尽管各种Linux发行版都有各自不同的包装方式，但它们通常都有用户可以利用的共性。例如，大多数发行版都力争在一定程度上与事实上的可兼容Linux环境标准“Linux标准化规范（LSB）”兼容。

1.1.4 自由软件与开放源码

Richard Stallman启动了GNU项目并成立了自由软件基金会作为一个非营利组织来负责监督该项目。他还编写了第一版的通用公共许可证（GPL）——为Linux系统编写的大部分软件都使用GPL许可证。GPL本身是一个很有意思的文档，因为它的目的不是限制你使用GPL授权的软件，而是保护用户和开发者获得源代码的权利^①。

GPL条款允许你修改Linux内核和其他GPL许可的自由软件，作为回报，你应该公布这些修改，以便其他用户使用它们（或将它们集成到指定软件的下一个正式版本中）。例如，GPL允许你修复一个重要的应用程序如Open Office的bug，或为GNOME桌面系统上的totem多媒体播放器添加定制音频文件支持。GPL带给开发者很大的灵活性，你可以出于任何目的来使用Linux，只要你将自己的修改也同样地提供给其他用户即可。这是关键的一点——即GPL试图保持开发过程的开放性。

对Richard Stallman来说，遗憾的是，英语中还没有一个可以和法语单词libre（英语单词liberty中自由的含义）完全相当的单词，所以很多人混淆了自由软件和免费软件的概念。事实上，许多自由软件都是完全免费的，但也有一些公司通过销售GPL许可的软件（包括它的自由发布的源代码）来赚钱。他们并不是通过软件本身来赚钱，而是当软件出现故障时，通过提供各种技术支持和附加的专业服务来赚钱。

为了减少对“自由软件”一词理解上的混乱，人们提出了术语“开放源码”，这个术语成为了20世纪90年代的流行词汇。与自由软件不同，开放源码并不特指GPL许可的软件，而是指一种对包括源代码的软件（使之可以被他人调整、调试和改进）的普遍需求，即使该源代码是在一个比GPL限制更严格的许可证下授权的。因此，有更多的软件虽然不是自由软件，但从技术上却满足开放源码的定义。

当你要修改现有的拥有GPL许可的软件时，理解GPL究竟对这项工作有哪些要求是非常重要的。虽然你不一定在自己的程序中使用GPL许可，但你必须尊重已这么做的其他人的权利。在因特网上有许多潜在侵害GPL许可的例子——通常是由于公司不清楚在对软件（如Linux内核）做出修改时需要将这些修改提供给用户。当然，你并不想成为下一个这样的案例，所以请始终确保你和你的同事都了解GPL，并尽早决定你准备如何利用它开展工作。

^① 在写作本书时，GPL正在第三次重大改写中。新版本很可能会成为最具争议的自由软件许可证之一。它包括对专利和其他技术许可的规定，试图取缔数字版权管理（被Richard Stallman称为“数字限制管理”）和大量其他的規定。

1.2 开发起步

作为一个Linux开发人员，你所需要迈出的第一步是为今后的任务准备好你的工具。这意味着你需要有一个合适的开发系统，以便编译和测试自己的Linux程序。虽然几乎任何合理的工作站都可以满足要求（起码在一开始是这样），但当你在编译了许多软件之后，你可能就会选择一个更高性能的机器以便节省编译时间。没有什么比不断等待大型软件编译完成更让人沮丧的了。但是，在你会跑之前，先学会走总是好的。

需要在这里强调的是，本书的作者并不准备推荐读者安装或使用特定的Linux发行版。市面上存在着许多优秀的Linux发行版，促使你选择和支持某一个特定的版本而不是其他版本的原因取决于企业营销和社区兴趣。尽管如此，先选择一些知名的发行版（至少在一开始）以便你在出现错误时可以更好地从一个非常活跃的开发者社区得到帮助还是很有必要的。

你可以通过一些公正的网站，如www.distrowatch.com来跟踪现代Linux发行版的当前发展趋势。Distrowatch网站还提供针对每一个发行版的有用的信息资源。

1.2.1 选择一个Linux发行版

在写作本书的时候，已有超过300种Linux发行版在全世界范围内使用，而且这个数字几乎每天都在增加。因为一般的Linux发行版所携带的大部分（即便不是全部）软件都遵循GNU的通用公共许可证（GPL），所以几乎任何人都可以利用这些软件并将它们打包放入自己的发行版中。这鼓励了人们进行主动尝试和试验，但对那些决定为300多种不同的发行版制作软件包的人们来说，它也很快会增加管理支持的难度。

作为一个软件开发人员，幸运的是，你需要支持的大多数Linux用户使用的仅仅是一小部分流行的Linux发行版。那些没有使用这些知名发行版的用户所使用的版本很可能也是基于它们的。因为对于较新的发行版来说，将其建构在一部分现有用户的适当需求之上是一件很寻常的事情。显而易见的是，与使用某个流行版本的成千上万用户相比，使用某个特定专家的Linux发行版的100个用户不一定能获得与前者相同程度的支持。

下面列出的是目前较受欢迎的10个Linux发行版：

- ❑ Debian GNU/Linux
- ❑ Fedora（以前称为Fedora Core）
- ❑ Gentoo Linux
- ❑ Mandriva Linux
- ❑ Red Hat Enterprise Linux（RHEL）
- ❑ Slackware Linux
- ❑ OpenSuSE
- ❑ SuSE Linux Enterprise Server（SLES）
- ❑ Ubuntu

1. Red Hat的Linux发行版

Red Hat曾经制作了被称为Red Hat Linux（RHL）的Linux发行版。这个发行版一直发展到了9.0版，在这之后，Red Hat推出了被称为Red Hat Enterprise Linux的商业产品。与此同时，Fedora社区的Linux

发行版成了供那些喜欢使用一个没有商业支持的完全开放源码的Linux的用户可选择的一个版本。Fedora在桌面用户和爱好者中非常受欢迎，并且被自由软件开发者以及商业厂商广泛使用——当然，商业厂商还需要继续努力，在企业版Linux中测试和验证他们的软件。

要了解更多有关Red Hat的信息，请访问www.redhat.com。Fedora项目有它单独的网站www.fedoraproject.org。

2. Novell的Linux发行版

Novell公司于2004年收购了SuSE并获得对SuSE Linux的完全控制权。与此同时，各种销售和品牌决策也影响了Novell对未来Linux产品的命名。和Red Hat一样，Novell也为他们的操作系统提供了一个社区版本——OpenSUSE。它由一个日益增长的用户社区维护，他们帮助发掘新的技术，而这些技术最终可能会反馈到商业SuSE Linux企业服务器的下一版本中。Red Hat和Novell通常被认为是市场上最大的两个商业Linux厂商。

要了解更多有关Novell和SuSE的信息，请访问www.novell.com。OpenSuSE项目有它单独的网站www.opensuse.org。

3. Debian和Ubuntu GNU/Linux

Debian存在的历史和Red Hat、SuSE一样长久，并且拥有大量的核心支持者。作为一个完全由社区维护的发行版，它的动力并不来自于某个特定公司的商业目标，而只是简单地追求艺术的境界。虽然Debian过去的开发周期的确太长——主版本的升级通常需要花费多年的时间，但这确实是一个值得称道的目标。过去几年已经产生了多种Debian的派生产品，包括Progeny Linux，它是最早尝试制作商业版本的Debian发行版之一。

Mark Shuttleworth一手创建了Thwate安全咨询公司，他通过该公司的业务创造了大笔财富，其业务发展部分依赖于Debian系统。因此，他积极参与Debian社区，并于2004年创办了Ubuntu项目。Ubuntu基于Debian，但它的目的并不是要取代Debian。相反，Ubuntu项目的目标是提供稳定的版本发布周期，并将产品化后的Debian放入发行版中以提供给用户。支持Ubuntu开发的Canonical公司为这一过程开发了多种工具，包括在本书后面会提到的Launchpad和Rosetta。

要了解更多有关Debian GNU/Linux的信息，请访问www.debian.org。Ubuntu项目有它单独的网站www.ubuntulinux.org。

4. Linux发行版分类

Linux发行版根据它们的目标的不同，大致可以分成三个不同的类别。这些目标分别为：派生自另一个流行的发行版；针对易于使用而设计；为那些有更高需求的用户而设计。例如，一般桌面用户不会因为一时心血来潮而去重建他（或她）的整个Linux发行版，而有些服务器管理员则乐于享有去除每一个可能影响机器性能因素的权力和灵活性。

请记住Linux为我们带来了很大的灵活性——如果有人能想出一种使用Linux的方法并为此创建一个新的发行版，别人可能早已着手实现它了。

● 基于RPM的发行版

基于RPM的发行版之所以这么命名是因为它们使用Red Hat的RPM软件包管理工具来制作和分发发行版中的单个组件。早在1995年秋，RPM就成为了Linux最早的软件包管理工具之一。它很快被诸如SuSE Linux等其他Linux发行版采用。RPM由此从Red Hat软件包管理程序（Red Hat Package Manager）更名为RPM软件包管理程序（RPM Package Manager）以反映如今RPM工具独立开发这一现状，但仍

有许多使用RPM的发行版继续利用Red Hat发行版中的管理工具。

基于RPM的发行版，如Red Hat的Enterprise Linux（RHEL）和Novell的SuSE Linux Enterprise Server（SLES）占据了如今全球使用的商业Linux中的大多数。如果你正在为企业编写软件，则一定要确保你的软件支持如上所述的基于PRM的发行版。你并不需要购买这些发行版的企业版本来进行日常的软件开发。相反，你可以使用它们的社区维护版本，如Fedora（派生自Red Hat Linux）或OpenSuSE Linux发行版。

- 派生自Debian的发行版

正如你将在本书后面看到的，派生自Debian的发行版基于Debian Linux发行版和诸如apt这样的软件包管理工具。Debian的dpkg软件包管理工具的编写和PRM的最初工作大约是在同一时间开始的，但由于不同的设计决策和设计哲学，这两个工具沿着各自独立的轨道向前发展。Debian以它构成了各种社区和商业Linux发行版的基础而闻名。

Debian是一个由社区维护的Linux发行版，它由非营利组织SPI（Software in the Public Interest）管理。从最早的版本开始，人们就有了定制Debian系统和为某一特定需求发布Debian派生版本的兴趣。Debian派生版本最鲜明的例子之一就是Ubuntu Linux发行版，它旨在通过规范的发布周期来获得广泛的采用，并通过对整体开发的掌控以达到某些特定目标。

- 源代码发行版

Linux发行版并不需要基于某个公共的软件包管理系统。许多发行版只使用很少的软件包管理或根本没有使用软件包管理，而是将软件组件封装在单独的档案文件中。此外，有些发行版还要求你在安装系统时进行编译，这可能是出于应用（或理论）的需求，但这种发行版通常仅限于非常特殊的Linux市场。

构建自源代码的发行版如Gentoo的设计目的是便于使用；同时，通过为每个安装好的系统本地定制软件以提供高性能。如果需要，Gentoo可以使用一个名为portage的系统来实现下载和构建每个单独的软件应用程序的自动化。但需要记住的是，第一次需要使用Open Office并指示portage来为你安装它时，portage可能需要花费数个小时的时间来完成这项工作。

如果为大众市场制作应用程序，通常就不必关心基于源代码的发行版。因为大多数用户宁愿使用流行的商业发行版或经过标准化软件包装过程处理的社区发行版。这减少了软件支持的难度，也使得你的生活变得更加轻松。如果你对Gentoo Linux有兴趣，不要忘记访问它的网站www.gentoo.org。

- 自己动手

正如你将在本书后面看到的，你完全可以从各个组件开始构建自己的Linux发行版。有几个原因可能会促使你这样做——好奇心、需要更大的灵活性和定制能力，等等。事实上，如今市场上许多嵌入式Linux设备都是由生产该设备的厂商完全从头开始构建的。毋庸讳言，我们通常不鼓励大家在没有熟悉Linux发行版所需要的内部软件包、软件和工具之前就尝试构建自己的发行版。

如果你想从头开始构建自己的Linux发行版，可以参考“Linux From Scratch”（Linux从零开始）项目，它是一个自助式的指南，其网址是www.linuxfromscratch.org。你还可以尝试一些自动化的发行版构建工具，如PTXdist，其网址是<http://ptxdist.sf.net>。

1.2.2 安装Linux发行版

决定好使用哪个Linux发行版后，接下来需要安装至少一台用于开发的机器。但需要注意的是，

在着手开发时，你并不需要为今后可能要支持的每一个发行版单独安装系统。反之，你应该选择一个在开发和运行软件过程中感到舒服的发行版。以后，你可以将该软件移植到其他可能需要的发行版中。不要忘记还有一些虚拟化产品如Xen和VMware可以极大地缓解测试压力，因为你可以将任意一个现代Linux发行版安装到它自己的虚拟沙盒中，而不影响已安装的系统。

本章稍后的部分会介绍一些在线组织和资源的链接，你可以在那里与他人讨论如何选择Linux发行版，并就安装过程中遇到的问题提问。

1. 获取Linux

大多数现代Linux发行版都以CD或DVD介质的方式提供，或以CD或DVD映像文件（ISO）的方式供用户从因特网上下载。Linux发行版通常会利用镜像站点来分担庞大用户群通过其高速链路下载CD或DVD映像所带来的负载。你可以通过从地理位置上靠近你的镜像站点下载ISO映像文件来为减轻服务器负担做出一份贡献。通过这样做，你就不会因为大量的下载而导致国际链路不必要的阻塞——请记住，Linux就其本质来说，是国际性的。

不要忘记使用BitTorrent，作为一种利用点对点技术的软件，它可以提高下载速度。Linux发行版遵循GPL的条款，它允许自由地重新发布，因此有许多用户通过设置BitTorrent tracker来加速下载，而实际上在提高自己的下载速度的同时也帮助其他人提高了下载速度——请从该软件的网址获得对这方面内容的解释。

预先警告一下，即使通过现代的高速因特网连接，下载一个特定的Linux发行版也需要花费数小时。如果你不想等待如此长的时间来下载多个CD或DVD映像，通常也可以选择使用在线安装的方式。这一过程所需要的时间也比较长，但你只需要安装那些你选择的软件包，因此安装程序就不用先获得全部的数据了。要执行在线安装，首先需要在厂商的网站上寻找容量较小的网络安装CD映像。它的容量一般不超过100 MB，因此下载速度会非常快，而且它也允许用户进行完整的安装。

当然，也可以选择购买现成的盒装产品以节省下载的时间和避免刻录光碟的麻烦。选购商业Linux发行版时，请求助本地Linux供应商。以后需要其他的软件时，他们也可以帮上大忙。所以，如果可能，你可以利用这个机会与他们建立联系。你可能还会发现你的本地Linux用户组与某些Linux厂商有特惠供应关系，厂商们会向爱好者提供相应的产品。

2. 决定安装时的软件包选择

大多数现代Linux发行版的安装过程都是很平稳和轻松的，它只需要你回答几个问题。告诉安装程序你想给Linux机器起的名字、机器的网络配置以及其他若干细节，它马上就会在你的机器上安装许多优秀的软件。如今一个常规的Linux安装所需要的过程就是这么简单——当然是远离了那个自己从头开始构建系统的年代了。

大多数安装程序在设置一个普通的Linux桌面系统或服务器系统时，一般不会自动包括开发工具。尤其对于流行的Linux发行版，如Red Hat、SuSE或Ubuntu来说，一般不会预先安装GNU工具链和相关的编译工具。你需要在安装系统时修改默认的软件包选择，以包括通常标为“开发工具”（development tool）或有类似名称的软件包。要做到这一点，你需要根据你所使用的Linux发行版的具体版本情况选择定制安装选项。请查看相应的安装文档以获得指导。

图1-1显示的是在安装Fedora Core 5系统时选择开发软件包的画面。

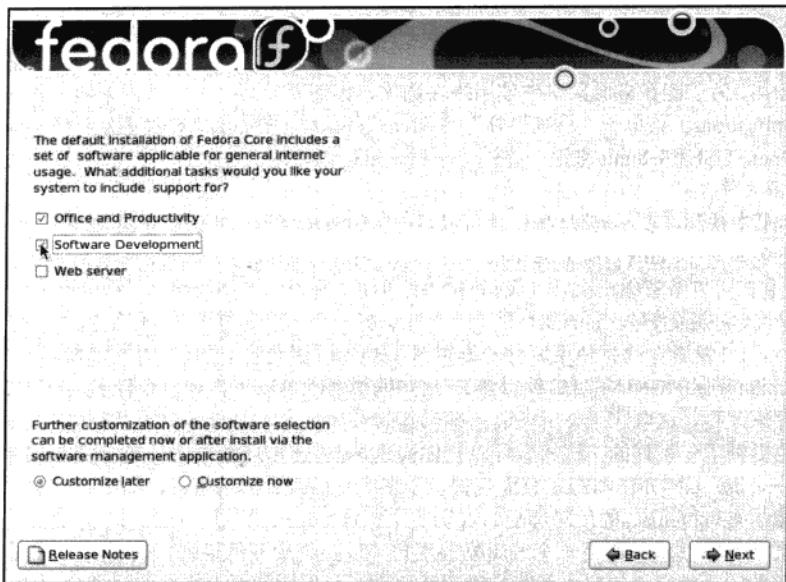


图1-1 选择Fedora Core的开发软件包

如果你在安装系统时错过了添加开发工具的机会，还可以在系统安装完以后再添加它们。这通常可以通过使用发行版自带的图形化软件包管理工具来完成。图形化软件包管理工具，如yumex（Fedora）、YaST（SuSE）和synaptic（Ubuntu）以组的方式提供相关的软件包，并简化了确定哪些组件需要安装的过程。如果这一切你都没有做，那么当你尝试执行本书中的一些示例代码时，将会遇到一些奇怪的错误，此时请查看到底少安装了哪些工具。

3. 设置开发环境

新安装的Linux系统通常会自动运行一个图形化桌面环境。如今大多数Linux系统都会选择GNOME或KDE图形化桌面（有些系统也会同时安装这两种桌面，让你选择其中之一来使用）。虽然本书尽量保证没有任何偏向，但也不可能在一本书中对所有的技术都做同一程度的介绍。所以我们在特别谈论与桌面相关的话题时，会侧重于GNOME桌面环境。

GNOME是Fedora和Ubuntu项目所使用的默认图形化桌面环境。不过，不管你个人或组织的喜好是什么，它们的界面看起来都差不多。你会很快找到管理工具（位于系统菜单）以及那些由你的发行版预装的开发工具。现在已有多个发行版默认安装了Eclipse IDE开发环境——如果你熟悉其他一些图形化开发工具，如Windows上的Microsoft Visual Studio，就会发现它是一个良好的开端。

- 找到终端

和其他UNIX系统一样，Linux系统建立在许多不同的工具和软件之上，并通过它们的共同合作来完成工作。虽然在过去几年中，图形化桌面环境已变得越来越流行，但完全通过系统终端的方式来完成日常的软件源文件编辑和软件编译仍然是一件很平常的事情。尽管你可以使用如Eclipse这样的图形化开发环境，但了解如何以命令行的方式来完成工作也是很有必要的。

在阅读本书时，你会发现书中的大多数示例代码都会包括一些简单的命令，你可以在命令行上运

行这些命令来编译软件。你通常可以通过“系统”(system)菜单来找到可用的终端，在某些系统中，也可以通过在桌面上单击鼠标右键从弹出的菜单中选择“打开终端”(Open Terminal)命令来查找。在Fedora系统中，为了能在桌面菜单中使用终端选项，你需要安装一个额外的系统软件包（使用“应用程序”(Applications)菜单→“系统工具”(System Tools)菜单中的“软件更新”(Software Updater)工具，对于OpenSUSE和Ubuntu来说，终端选项默认就可以使用。

● 编辑源文件

你将在本书中找到许多示例源代码，你可以运行它们或按照自己的想法修改它们。在后续的章节中，你还将了解更多如何在Linux系统上编译软件的知识。在本书相应的网站上，你也可以找到许多示例程序，通过下载它们可以节省你每次敲入代码的时间。但是，你显然希望能够尽早地开始编写自己的程序。因此，我们建议你在开始开发Linux软件之前，首先选择一个你感觉用起来很舒服的文本编辑器。

大多数Linux开发者会选择使用流行的编辑器，如vim（源自古老的UNIX vi编辑器）或GNU emacs（Richard Stallman开创的GNU项目的编辑器）。这些编辑器既可以工作在命令行上，也可以作为一个图形化应用程序来运行，这取决于你所安装的编辑器的具体版本。它们都提供了丰富的功能以提高工作效率，同时也提供了一系列的文档和教程以让你快速掌握它们的使用方法。对于那些喜欢使用图形化编辑器的用户来说，GNOME和KDE桌面也提供了多个功能强大的编辑器。

值得注意的是vi和emacs的敌对历史。在历史上，vi和emacs的用户相互排斥，使用一个编辑器的用户通常都会非常不喜欢使用另一个编辑器的用户（以及使用其他编辑器的用户）。人们在邮件列表中时不时挑起的关于编辑器的口水战其实没有任何意义，但因特网上销售的泾渭分明的vi和emacsT恤以及其他商品则显示了一些人是如何严肃地对待这场编辑器之战的。试图准确地理解为什么人们这么在意这个问题并不是一个好主意——你所需要做的只是去适应它。

不管你选择了哪种文本编辑器，都请不要尝试使用字处理软件如Open Office writer或abiword来编辑程序的源代码。虽然在技术上这么做也是可行的，但这些工具通常会破坏源代码，甚至会在编辑文本文件时尝试嵌入各种RTF格式，从而会迷惑你稍后用来编译软件的编译工具。

图1-2和图1-3显示了使用vim和emacs文本编辑器编辑源文件的例子。

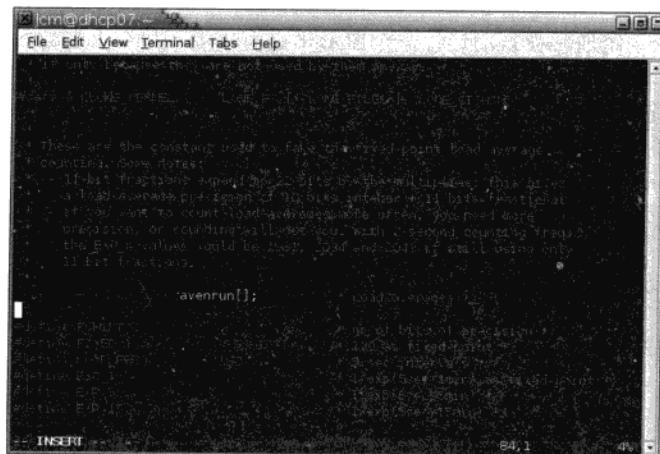


图1-2 使用vim编辑源文件

```

#define CLONE_CHILD_SETTID      0x01000000      /* set the TID in the child */
#define CLONE_STOPPED          0x02000000      /* Start in stopped state */

/*
 * List of flags we want to share for kernel threads,
 * if only because they are not used by them anyway.
 */
#define CLONE_KERNEL    (CLONE_FS | CLONE_FILES | CLONE_SIGHAND)

/*
 * These are the constant used to fake the fixed-point load-average
 * counting. Some notes:
 * - 11 bit fractions expand to 22 bits by the multiplies: this gives
 *   a load-average precision of 10 bits integer + 11 bits fractional
 * - if you want to count load-averages more often, you need more
 *   precision, or rounding will get you. With 2-second counting freq,
 *   the EXP_n values would be 1981, 2034 and 2043 if still using only
 *   11 bit fractions.
 */
extern unsigned long avenrun[];           /* Load averages */

#define FSHIFT            11                  /* nr of bits of precision */
#define FIXED_1            (1<<FSHIFT)        /* 1.0 as fixed-point */
#define LOAD_FREQ          (5*HZ)             /* 5 sec intervals */
#define EXP_I              1884                /* 1/exp(5sec/1min) as fixed-point */
#define EXP_S              2014                /* 1/exp(5sec/5min) */
#define EXP_15             2037                /* 1/exp(5sec/15min) */

#define CALC_LOAD(load,exp,n) \
    load *= exp; \
    load += n*(FIXED_1-exp); \
    load >= FSHIFT;

extern unsigned long total_forks;
extern int nr_threads;
%% sched.h      (C Abbrev)--L75-- 5--

```

图1-3 使用emacs编辑源文件

● 使用root账号

为避免偶发的意外系统损坏——如核心系统文件的删除和系统工具的意外删除等，你最好以普通用户的身份来完成日常工作。普通用户拥有对他（或她）的家目录（在/home目录下）的完全访问权，并可以轻松地编译和测试大多数普通应用软件。使用普通用户的身份已足以完成大多数开发任务，但有时候你也需要用到管理员（root）账号，比如修改全局系统配置、安装测试软件以及在通常情况下完成任务。

你并不需要总是以root身份来使用系统，或使用root账号重新登录系统。当需要使用root账号时，我们推荐使用sudo工具。sudo允许用户以root身份运行一个命令，而不需要冒着始终以超级用户的权力来使用系统的风险。令人惊讶的是，作为root用户，一次不经意的错误的命令输入就可能意外地导致整个系统崩溃。因此，大多数开发者在一般情况下都只使用他们自己的账号。

在使用sudo命令之前，你必须确认你的普通用户账号已列在文件/etc/sudoers中。例如，可以在该文件中增加如下一行来授予“jcm”用户使用sudo的权限。

```
jcm    ALL=(ALL) ALL
```

这授予了jcm以root用户身份在任何机器上（大多数情况下只会涉及本机，但如果你是在一个局域网中，请询问IS或IT部门的人员）运行任何命令的权限。要想真正以root权限运行一个命令，你可以像下面这样来使用sudo命令：

```
$ sudo whoami  
root
```

如果这是你第一次输入该命令，系统首先会要求你输入密码。

当你第一次使用该命令时，`sudo`会首先向你发出警告，并说明作为一个`root`用户可能会给系统带来的危险，然后会要求你输入密码，这个密码可能并不一定是你的登录密码。在有些发行版中，`sudo`被设置为在默认情况下要求输入`root`账号的密码，而另外一些发行版则要求你输入自己的登录密码以使用`sudo`工具。你需要查看你的发行版文档，或使用UNIX的`man`命令来了解更多这方面的信息。

如果你是在一个企业环境中工作，别忘记通知你的IT或IS部门，你需要以管理权限访问开发用的机器。这将省去以后的很多麻烦，除非他们能够在你每次需要使用`root`账号时都能特别地支持你。

4. 开发版本

Linux发行版通常都有一个对应的开发版本，它紧跟发行版的开发现状。开发版本的更新频率要远高于它们对应的稳定版本。发行版的稳定版本通常每隔6~18个月才会更新一次，而开发版本的更新频率甚至可能是每天一次。三大发行版——Red Hat、SuSE和Ubuntu——都有每日更新的不稳定的开发版本。通常情况下，你不需要去使用这些版本，但对它们有所了解还是有必要的，所以在这里我们将对它们进行快速地概览。

这里对现代Linux发行版的开发版本的介绍只是为了满足你的兴趣以及出于教学的目的。虽然它们可能会对你的某些开发决策有所帮助，但你不应直接在这些开发版本上编译或开发软件产品。因为它们的版本变化非常频繁，因此要想获得可重现的结果非常困难。而且对于开发版本来说，发生彻底的系统损坏也并不鲜见。

Red Hat将Fedora的不稳定版本称为rawhide。你可以在Fedora的网站上找到它，但与直接安装rawhide相比，通常你是通过在最新的稳定版本中执行一次“`yum update`”（在将目录`/etc/yum.repos.d`中的YUM开发软件库取消注释之后）来进行切换。Rawhide中呈现的是一派繁忙的景象，你常常可以从中察觉到在Fedora的下一版本中可能会做出哪些改进，而这些改进最终甚至可能在将来的某个时刻影响到对Red Hat企业产品的改进。

Novell将它的OpenSuSE不稳定版本称为Factory。你可以在OpenSuSE的网站上找到它，其安装方式是通过使用可以下载得到的网络启动映像文件。如果要执行网络安装，需要小心地参照安装说明来操作。因为这需要在启动过程的一开始就改变多种选项——甚至在启动YaST安装程序之前。你也可以使用YUM（请阅读在线文档）来进行升级，不过这一功能在写作本书时刚推出不久。Factory的更新周期也有一定规律，其使用的技术最终将会融入到SuSE Linux企业服务器中。

和Debian一样，Ubuntu也有一个不稳定的版本。每当它包含的软件包更新时，这个版本也会随之升级。因此，可能会发生系统在某一特定时刻不能及时升级到最新的不稳定版本的情况。与这里提到的其他发行版不同，Ubuntu和Debian提供它们的发行版的临时测试版本，这个版本中包含了在不稳定版本中已证实可用的软件包。通过修改文件`/etc/apt/sources.list`，可以使用命令“`apt-get upgrade`”将系统升级到不稳定版本。

1.2.3 沙盒和虚拟化技术

在熟悉Linux开发并更富有冒险精神之后，你可能想拥有一个沙盒，这样就可以不用冒着可能破坏系统中已安装的文件的风险，来测试你的一些想法（或完全没有价值的东西）了。尤其是当你以后决定编写自己的Linux内核设备驱动程序或修改关键的系统组件时，更需要这么做，因为你不会希望在一台你需要始终保持稳定运行的机器上做这些工作。许多黑客都利用旧电脑来达到这一目的。

通过使用虚拟化技术如VMware、qemu和Xen可以获得大量的测试用虚拟机。你可以整日在上面闲逛而不用去购买任何额外的硬件。使用虚拟化技术不仅是一个很好的节约成本的想法，而且当你需要建立一个标准的测试环境并和你的同事分享构思时，它也非常实用。大多数虚拟化技术允许你对以特定方式配置的系统建立“快照”(snapshot)，这样你就可以通过一些网络存储区域存储它或将它发送给你的同事。

1. VMware

VMware允许通过其专有的图形化软件来管理大量的虚拟机。通过它配置一个新的虚拟机并在其中安装一个Linux发行版是一件轻而易举的事情。使用VMware，你可以在不改变机器上其他软件的情况下轻松地安装各种不同的基于PC的Linux发行版。当你想要存储预先配置好的测试用机器或尝试一些不依赖于某些特定硬件设备的实验性特点时，使用它是非常适合的。但是，如果要测试定制的Linux内核设备驱动程序，它就不太适合了。

要想了解更多有关VMware的信息，请访问www.vmware.com。

2. Qemu

Qemu是一项开放源码的虚拟化技术，它可用于运行Linux。它完全免费，但与专有软件如VMware相比，功能更为有限。通过使用qemu的命令行工具，可以创建一个虚拟机环境，然后在其中安装Linux发行版。因为qemu遵循GNU通用公共许可证，所以你可以修改它并为它添加有趣的新功能。正如你将在本书后面看到的，可能的修改包括定制虚拟化硬件设备，这样你就可以编写自己的设备驱动程序，而不需要冒着影响正常开发环境稳定性的风险。

要想了解更多有关qemu的信息，请访问该项目的网站www.qemu.org。

3. Xen

在过去的几年中，人们对被称为Xen的虚拟化技术越来越感兴趣。在写作本书的时候，Xen几乎每隔一周就会成为新闻头条。和VMware一样，Xen能够在一个虚拟化环境中运行任何一种操作系统。但与VMware不同的是，Xen是一款自由软件，并且目前市面上有各种针对它的图形化配置工具可以使用。大多数最新的Linux发行版都会在某种程度上对编译和配置Xen虚拟化环境给予特定的支持，以便用户测试自己的软件。

要想了解更多有关Xen的信息，请访问该项目的网站www.cl.cam.ac.uk/Research/SRG/netos/xen。

1.3 Linux 社区

当为Linux开发软件时，要认识到的一件最重要的事情是你并不孤独。在全球存在着一个庞大的开发者社区，他们中的许多人乐于和你交流经验或者会在你陷入困境时帮助你。你可以通过许多不同的方式来与这些Linux用户和开发者取得联系，而且你也希望能在遇到第一次挫折之前加入这个大型的社区。

除了加入Linux社区以外，你或许还想开始阅读一本内容不错的杂志。历史最悠久的要算*Linux Journal* (www.linuxjournal.com)，但在写作本书时，全球差不多已有几十种这方面的杂志了。

1.3.1 Linux 用户组

无论你身处何方，你都会发现周围有很多其他的Linux用户。大部分城镇或城市里都有他们自己的Linux用户组（Linux user group, LUG），它由当地的来自各行各业的Linux爱好者组成。其中一些成员可能和你一样对Linux不太熟悉，并且希望能从那些已加入社区10年甚至更长的老用户那里获取建议。事实上，一些Linux用户组的存在时间已远远超过了10年。

你可以通过因特网找到更多有关本地Linux用户组的信息。只需在Google的专用Linux搜索引擎 www.google.com/linux 中输入城镇名或城市名即可。

1.3.2 邮件列表

如今全球的大多数Linux开发者都通过电子邮件列表来交流新的想法、交换软件补丁和参与广泛的讨论。针对许多不同主题的邮件列表数目是如此之多，从讨论Linux内核中最小的子系统到讨论整个Linux发行版，甚至在地球最偏远的地区都会有某种类型的邮件列表，以至于在本书中不可能对它们进行全部的介绍。我们鼓励你通过你的本地Linux用户组和你所使用的Linux发行版的销售商所提供的用户列表来加入邮件列表作为你的起点。

在本书中，你将找到很多邮件列表和其他类似资源的链接，它们将帮助你更多地参与或更好地理解某一特定的主题。

1.3.3 IRC

开发人员经常希望能以一种比邮件列表所能提供的更加互动的方式来参与讨论。Internet在线聊天系统（Internet Relay Chat, IRC）促进了这一过程，它允许你加入全球IRC网络中的各种频道，许多拥有邮件列表的组织同时都会有一个某种类型的IRC频道以实现邮件列表讨论。你可以通过IRC网络如Freenode、OFTC等找到大量的在线资源。这些IRC网络都已被预先设置进大多数现代Linux发行版的图形化IRC客户端（如xchat）中。

1.3.4 私有社区

Linux开发者社区并不像你想象的那样总是公开的。全球也存在一些封闭的组织，它们是为促进针对某一特定技术的真正核心开发者之间的讨论而成立的。这些组织有时也会举办一些专题活动，但更多的是使用非公开的邮件列表和IRC网络进行交流。尽管存在着这些私人俱乐部，但Linux确实有着一个开放的开发过程，认识到这一点是很重要的。

私有组织的一个例子是那些遍布全球的安全组织，他们对Linux软件中发现的错误进行迅速地修复。在找到安全漏洞并和厂商以及其他涉及的第三方协同推出修复版本之前，他们不会公布工作的细节。这有助于减少安全事件的数量。如果你发现了Linux软件中的安全漏洞，请通过适当的频道来报告它。

要记住，Linux没有阴谋。

1.4 关键差别

Linux与你以前遇到的其他操作系统不同。大多数操作系统都是由一个由技术精湛的人员组成的小型团队通过多年的设计完成的。这些设计人员将规格文档交给软件工程师，由他们来实现设计的内容。Linux并不属于这些封闭团队。虽然的确有许多厂商在半封闭的大门之后从事Linux技术的开发，但Linux的核心开发过程是对所有人开放的——它毫不遮丑。

拥有一个开放的开发过程意味着全世界的开发者都可以仔细研究Linux中各种特征的实现并提出他们各自的修改方案。因此，受益于这种“多眼球”的可扩展性，Linux允许任何人为它做出贡献——这大大超过了即使最大的专有软件公司所拥有的资源。拥有一个开放的开发过程还意味着Linux系统的核心不会接受不完美的方案。每个人都会犯错误，但Linux社区对它可以接受的改进通常都是非常挑剔的。

1.4.1 Linux 是模块化的

一个典型的Linux系统是由许多小组件构建而成的，它们共同合作以构成一个大的整体。与微软的Windows操作系统不同，Linux明确地将所有的功能（甚至最小的功能）都分解到一个个单独的专用工具中。例如，有的工具专用于设置系统时钟，有的用来控制声卡的音量，有的专用于个人的网络操作，等等。通过查看任何一个Linux系统中的标准目录/bin和/usr/bin，你就可以看到许多这样的专用工具。

和许多老一辈的UNIX系统一样，Linux建立在KISS（保持简单、傻瓜，keep it simple, stupid）准则之上。它遵循的是“要做一件事，就做好这件事”，而不是让一个大型的单一程序承担过多的功能。与微软的Windows操作系统不同，Linux系统被设计为易于修改。你被鼓励以任何你所选择的方式来定制系统，这也是自由和开放源码软件的观点。

在本书中，你将看到许多你可能之前都没有见过的工具的信息。不要惊慌，你将很快发现应该求助于哪些人，利用正确的资源和社区组织以及使用每个Linux系统自带的文档通常已足够帮你走出困境。许多所谓的专家事实上只是技巧纯熟的Google搜索者，他们知道如何搜索到能够帮助他们解决问题的信息。

1.4.2 Linux 是可移植的

正如你将在本书第3章中发现的那样，Linux本身可以说是目前最易移植的操作系统之一。Linux既有针对最小的嵌入式设备如手机、PDA、数码录像机（DVR）和机顶盒的发行版，同时也有支持大型机或用于处理人类基因组的超级计算机的发行版。为Linux编写的软件在设计时通常都考虑到了可移植性，作为Linux发行版的一部分，它必须能够自动地为各种不同的目标系统编译，所以在开发的早期就考虑到可移植性问题是一件很平常的事情。

当为Linux编写软件时，你需要时刻考虑你的决定将来是否会影响软件的可移植性。你是否需要它运行在64位系统上？你是否总是拥有一个基于GNOME的全功能的图形化桌面环境？或是否可能有人会将你的软件用在一个资源受限的嵌入式设备上？这些问题都是你需要考虑的，以免将来遇到不必要的麻烦。

1.4.3 Linux 是通用的

Linux内核本身力争尽可能地通用。这意味着内置的可扩展性的相同的源代码既可以运行在最小

的装置中，也可以运行在最大的大型机上。由于在编写软件时已考虑到灵活性，所以我们并不需要对软件做任何根本性的调整以支持各种目标系统。当然，某些功能可以针对特定的系统做出调整，但核心算法不会变。对绝大多数现代Linux发行版所附带的软件来说，也同样如此。

尽可能保持通用性并允许用户自己来决定他们如何使用Linux系统是Linux之所以会这么成功的原因之一。你应该总是从用户的角度来考虑他们会怎么使用你的软件，并避免在软件中加入一些不必要的设计限制。你并不需要支持那些以非常规方式使用你的软件的用户，但你也不应人为地强加一些限制，除非它是绝对必要的。

1.5 本章总结

在本章中，你学习了Linux的相关知识。你了解到术语Linux在不同的上下文中有着不同的含义。从技术上来说，Linux指由Linus Torvalds编写的核心操作系统内核，它由全世界数以千计技术精湛的开发人员共同维护。但Linux也可以用于指由建立在Linus的原始内核之上的软件构成的发行版。这个发行版包括数以千计的工具、现代图形化桌面环境和用户期望在一个完整的现代操作系统中找到的许多其他组件。

过去10年中，Linux的成功离不开社区的努力，是它使得Linux如此受欢迎，并成为除了大型专有UNIX系统和目前市场上其他操作系统之外用户的另一个操作系统选择。你现在知道了应该如何加入你的本地Linux用户组以及如何与世界各地的Linux开发者取得联系，他们乐于在你成长为专家的路上给予你帮助。你还了解了Linux和其他操作系统的许多不同之处。

工具链

工具链（toolchain）是在每一个大型开放源码项目（包括Linux内核本身）背后默默支撑的力量。它们由一组必要的工具和软件构成，用于编译和调试从最小的工具软件到你可以想象的最复杂的具有Linux内核特征的各种软件。如果你曾经编写过Linux程序，那么你很可能已用过了GNU编译器集（GCC），但要完成一个优秀的应用程序，要做的事情可比简单的编译源代码多得多，你需要借助一个完整的工具集来做到这一点，这套工具集通常被称为工具链。

工具链中包括编译器、连接器、汇编器以及调试器——用于跟踪所有程序（除了那些非常简单的程序）中的不可避免的错误。此外，还有各种其他的工具用于在必要的时候控制应用程序的二进制代码——例如，将Linux内核的二进制代码转换为机器的启动映像。绝大多数的Linux应用程序都使用GNU的工具链来编译，该工具链由GNU工程中发行的工具构成。

本章将向你介绍GNU工具链中的各种工具，以及其他一些相关的工具——它们也被Linux开发人员用于编译和调试应用程序。这些工具包含了许多非标准的特性，它们常被Linux应用程序以及内核使用。你将学习如何使用GNU工具链并熟悉它们的一些更高级的特性。在阅读完本章之后，你将具备编译和调试应用程序的能力，并将熟悉诸联汇编的概念和GNU二进制工具集（binutils）等的强大功能。

2.1 Linux 开发过程

要充分了解Linux中用于编译软件的每个工具，首先必须对整个软件开发过程以及工具链中每个工具的设计目的有一个高层次的理解。只有这样才能在本章后面实验这些工具时能够更容易地将这些概念应用到它们的高级应用中去。

现代Linux软件由大量单个的组件构成，它们在编译时被合并到少量的可执行程序和其他文件中。这包括应用程序的可执行文件本身，以及许多配套资源、文档和用于源代码管理（SCM）和版本控制的额外数据。这些单个的文件可能被包装进一个单独的面向特定发行版的可安装软件包中，然后将该软件包发送给用户。

当然，某些应用程序可能并没有设计为最终用户可安装，这通常发生在嵌入式设备或其他OEM解决方案中。在这些情况下，运行Linux的整个系统只是作为一个大型产品的一部分，你仍然以标准方式编写和编译软件，但需要将包装过程替换为一个自动化过程以将软件安装到目标设备上。

在本节中，你将学习如何使用源代码、如何获取它，以及如何将一组程序源代码配置为一个可工作的开发树。你还将开始研究一个通用的编译过程，Linux系统中大多数的应用程序都使用这一过程，你在编写自己的应用程序时也需要熟悉它。

2.1.1 使用源代码

开发人员很少直接与最终用户可安装的软件包打交道。相反，他们使用的是源代码软件包或包含应用程序源代码以及重新编译应用程序所必需的额外脚本和配置数据的打包文件。这些外部支持脚本和配置数据用于自动判断软件需求以及那些在指定Linux目标环境中使用该软件所必需具备的特性。

Linux软件通常以两种方式分发给开发人员。第一种（也是最基本的一种）方式是源代码打包文件，它常被称为tarball文件（这是因为标准的压缩打包文件的后缀名为.tar.gz或.tar.bz2）。源代码打包文件在使用标准过程编译它之前必须首先被解包，这可以通过使用合适的工具如命令行的tar命令或图形化打包文件管理工具（由你的图形化桌面环境提供）来完成。

第二种分发源代码给开发人员的方式是通过SCM工具。这些工具将获取源代码、跟踪本地修改、提交补丁或源代码的修改版本到中心库或更高一层开发人员的过程变得自动化。这些工具的选择将在本书后面进行讨论。如果你确实要使用SCM，你就必须遵循在开发团队的成员之间约定好的步骤和方法。

你将发现本书所提供的大多数示例代码都存放在Linux的普通tar打包文件中，当然，你也可以很容易地将这些示例代码输入到自己的SCM并跟踪你对它们所作的改动（见第4章“软件配置管理”）。要解开包含本章示例代码的打包文件，你可以使用如下所示的Linux tar命令：

```
tar xvfj toolchains.tar.bz2
```

"xvfj"选项标记告诉标准的tar命令释放给定的以bzip2格式压缩的tar打包文件并进行验证。如果文件是存储在老式的但仍然被广泛使用的gzip格式压缩的tar打包文件（通常使用.tar.gz后缀名）中，你就需要使用"xvfz"选项标记来解压缩它。

2.1.2 配置本地环境

我们所提供的示例项目打包文件中包含一个顶层目录和其他子目录。顶层目录中的README文件和INSTALL文件描述了（重新）编译软件的标准过程。这一过程首先需要配置本地编译环境，然后才是真正地运行命令来编译一个可执行应用程序。

当为Linux开发软件时，配置本地编译环境是一个必要的步骤，因为Linux可以运行在各种不同的目标平台上。每个不同的硬件环境可能都有其特定的限制。例如，内存的存储格式是大头字节序（big endian）还是小头字节序（little endian），或对要运行在指定目标处理器上的可执行程序有特殊的要求。而且每个不同的Linux发行版都有各自不同的软件环境——工具和系统函数库的版本随发行版的不同而不同。

因为只有极少数的Linux（或其他UNIX）环境是相同的，所以有必要通过一套工具以一种可移植和抽象的方式来处理这些差别。我们感兴趣的第一个工具是GNU Autoconf，它通过其生成的configure脚本来体现其功能。当执行configure脚本文件时，它将自动判断系统上是否已安装了必需的编译工具，并判断其版本是否有一些特殊的需求。

你可以执行如下命令来获得configure脚本的使用帮助：

```
$ ./configure --help
`configure' configures hello_world 1.0 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...
```

为简洁起见，对命令的输出做了删减。

根据configure脚本自身的配置情况，它的输出列出了各种可能的选项和值。其中大多数选项在这里都没有必要使用，因为本例中的configure脚本只是用来判断你是否正确地安装了必需的GNU工具链来运行本章中的代码。本书后面的示例，如下一章中的一些示例将利用Autoconf中更强大的功能，并讨论其内部细节。

请运行本章源代码包中包括的configure脚本来配置示例代码：

```
./configure
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
configure: creating ./config.status
```

configure脚本将自动判断你是否已安装了必需的GNU工具，并通过实际编译一个非常简单的test程序来验证它们是否可以正常工作。你将在本书的后面学习如何编写自己的configure脚本。

根据所使用的工作站的具体环境的不同，输出可能与上面列出的有所不同。如果在这一阶段出现任何错误，就说明你没有在系统上安装适当的工具。许多Linux发行版不再在它们的安装过程中自动安装完整的GNU工具链，所以需要你来安装这些缺少的开发工具。发行商通常会在安装过程中提供一个“开发”（Development）选项，选择它就会安装正确的开发工具（请参考第1章以了解更多关于如何获得一个基于正确配置的Linux系统的信息）。

2.1.3 编译源代码

示例源代码中的README文件对示例代码的作用进行了概述，它然后指向内容更详细的INSTALL文件，该文件描述了如何编译和安装示例源代码到你的工作站上。与所有其他Linux软件一样，文档还说明了当你不需要该软件时，应该如何删除它。

你可以使用如下命令来编译和测试示例代码：

```
$ cd src
$ make
$ ./hello
Hello, World!
```

和大多数类UNIX系统一样，Linux上的大多数软件是在GNU make的控制下进行编译的。这个便利的工具读取Makefile文件，并使用它来确定调用命令的顺序以及从指定的源代码（存储在示例源代码目录的src子目录下）产生可执行程序的步骤。GNU make通过几条特殊的指令就能理解如何编译大多数简单的可执行程序，但这也并不是绝对的。

make调用命令的正确顺序是由在Makefile文件中指定的依赖信息确定的。在本例中，源代码文件hello.c将被自动编译为可执行程序hello，然后你就可以执行它。make允许你根据期望的动作定义

多个不同的make目标。例如，你可以使用命令"make clean"或"make distclean"来清理已准备好分发给开发人员的源代码。

示例Makefile文件包含如下所示的依赖规则：

```
all: hello  
  
hello: hello.c
```

没有必要告诉GNU make应该如何将源文件hello.c编译成一个可执行程序。因为它清楚一个C语言源文件依赖关系意味着需要调用GNU C编译器和其他一些必需的工具。事实上，GNU make在默认情况下支持各种文件后缀名，它通过这些后缀名来确定针对不同类型源代码所应采用的默认规则。

在某些情况下，Makefile文件可能也是由其他自动化工具产生的。为了产生可以广泛移植到各种不同类型的目标Linux机器上的软件，我们通常会将Makefile文件的生成作为configure脚本生成过程的一部分来完成。Makefile然后会自动使用由configure脚本检测到的特定GNU工具并根据在配置过程中其他一些选项的情况采取操作，例如使用一个特定的交叉编译器来为运行在一个不同处理器架构下的目标编译软件。

自由软件基金会建议总是使用这种自动配置工具，因为它们降低了日常Linux软件编译的复杂度。而且作为这样使用的结果，你也不需要了解编译和使用大多数Linux应用程序所需的资源。更重要的是，对于那些想要从源代码开始编译应用程序的用户来说，这个道理也同样适用。

2.2 GNU 工具链的组成

上一节介绍了几乎通用的命令序列"configure, make"，它用于编译大多数的Linux软件。正如你看到的，通过将GNU Autoconf和GNU make进行适当的结合，也许只需使用数条短短的命令就可以将源代码编译为一个应用程序。你将在本章后面发现我们可以通过使用一个图形化桌面环境如Eclipse将整个编译周期简化为点击一个按钮来完成。

自动化是软件开发的一个重要组成部分，因为它不仅简化了重复编译周期，而且通过清除出错的可能性增强了再现性。许多开发人员（包括作者）都是好不容易才懂得尽可能地简化Linux软件的编译过程是多么的重要。他们往往在浪费了一整天去调试根本就不存在的bug之后才发现这其实是由编译过程过于复杂所产生的副作用。

虽然应该尽量使用手边的自动化工具，但仅仅依赖它们并不是一个好的编程习惯。对GNU工具链中的每个组成部分都有一个大概的了解是非常重要的，这样你就可以正确地处理自动化工具不能帮助你的情况了。因此，本章的其余部分将侧重于介绍隐藏在编译过程之后的每个工具，告诉你它们是什么做的，以及它们是如何纳入GNU工具集整体的。

GNU 编译器集

GNU编译器集（其前身为GNU C编译器）诞生于1987年。当时Richard Stallman（GNU项目的创办人）想要创建一个编译器，它可以满足他定义的“自由软件”概念，并可用来编译GNU项目发布的其他软件。GNU C编译器迅速在自由软件社区中流行开来，而且以其健壮性和可移植性而闻名。它已成为许多集成开发工具的基础，被世界各地的发行商应用在Linux和其他操作系统之上。

GCC已不再是主要针对GNU项目自身的软件的小型C语言编译器了。如今，它已支持了许多不同的语言，包括C、C++、Ada、Fortran、Objective C，甚至还有Java。事实上，现代Linux系统除了可以

自豪地炫耀那些由GNU工具直接支持的语言以外，它还支持大量其他语言。日益流行的脚本语言Perl、Python和Ruby，以及正在不断发展的mono 可移植C#实现的确有助于冲淡人们对Linux编程的传统看法，但这完全是另外一个问题了。

本章重点介绍GCC作为C编译器的用法。Linux内核和许多其他自由软件以及开放源码应用程序都是用C语言编写并使用GCC编译的。因此，即便你不准备直接在项目中使用GCC或将GCC用于其他用途，你最好也能对本章所介绍的概念有一定程度的理解。通过它们自己的文档和第三方在线资源的帮助，你应该能将本章所介绍的一般概念应用到其他类似的语言工具中。

除了其广泛的语言支持以外，GCC受到许多开发人员的喜爱还因为它已被移植到广泛的硬件目标（基于不同处理器架构的机器）上。标准的GCC版本支持超过20种微处理器，包括用在许多工作站和服务器中的Intel IA32 (“x86”）和AMD64处理器。GCC还支持高端的SPARC和POWER/PowerPC处理器，以及越来越多地专用嵌入式微处理器——用于使用Linux的小装置和其他设备。只要商业上允许，GCC可能还可以为它们编译代码。

1. 编译单个源文件

正如你已看到的，在Linux上编译软件的过程涉及许多工作在一起的不同工具。GCC是一个非常灵活和强大的现代C编译器，但不管怎么说，它终究只是一个C编译器。它非常擅于解析C源文件并为特定的处理器输出汇编语言代码，但由于它缺少一个针对某一特定机器的内置汇编器或连接器，所以它自身并无法真正产生一个可工作的可执行程序。

这个设计是有意而为之的。它遵循了UNIX哲学，即要做一件事，就把这件事做好。GCC支持各种不同的目标处理器，但它需要依赖外部工具来完成汇编和连接工作，具体步骤是首先由汇编器将已编译的源代码转换为目标代码，然后由连接器将它连接进一个合适的容器文件中，该文件可以被特定的Linux目标机器真正装载并执行。

虽然GCC自身并不能产生一个可工作的可执行程序，但它知道要真正实现这一目标还需要哪些GNU工具的帮助。作为GCC前端的gcc驱动程序，知道一个C语言源文件必须经过GNU汇编器和连接工具的汇编和连接，因此，当你递交给它一个C语言源文件要求编译时，它在默认情况下就会执行这些额外步骤。

为了进行测试，你可以创建“Hello World”程序：

```
/*
 * Professional Linux Programming - Hello World
 */
```

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    printf("Hello, World!\n");
    exit(0);
}
```

使用如下命令编译并测试这个代码：

```
$ gcc -o hello hello.c
$ ./hello
Hello, World!
```

在默认情况下，gcc驱动程序会执行所有必需的步骤以将hello.c源文件编译为一个可执行二进制程序。这包括调用作为GCC内的一部分的真正的C编译器（cc1），以及调用从GNU C编译器的输出中实际产生可执行代码的外部GNU汇编器和连接器工具。由于历史原因，在默认情况下产生的可执行程序名为a.out，但你通常可以通过gcc的“-o”选项来指定自己的可执行程序名称，如上例所示。

2. 编译多个源文件

大多数应用程序是基于多个单个源代码文件的，它们被单独编译，然后连接在一起以构成最终的可执行程序。这既简化了开发过程并允许不同的团队开发一个项目的不同部分，同时也鼓励适当地进行代码复用。在本书的后面，你将学习到Linux内核是如何由数以千计的单个源文件构成，以及图形化GNOME桌面应用程序是如何通过许多单独的组件编译而成的，但就目前而言，我们首先来学习如何将两个源文件编译为一个单独的程序。

gcc驱动程序不仅懂得如何将单个源文件编译为一个可执行程序，而且它通过适当地调用GNU连接器，还可以将多个不同的目标文件连接在一起。当你指定多个目标文件（.o）作为GCC的输入参数时，GCC会自动将它们连接成一个单独的可执行输出文件。为了对此进行测试，你可以创建一个由两个单独的源文件组成的程序，它们将被连接成一个单独的可执行文件。

源文件message.c包含一个简单的消息打印函数：

```
#include <stdio.h>

void goodbye_world(void)
{
    printf("Goodbye, World!\n");
}
```

本例中的goodbye_world函数构成了一个简单的软件库，它可以被其他源代码调用。这个函数依赖标准的C库例程printf来真正执行必需的底层IO以在终端上打印一条消息。C函数库printf将在稍后的连接阶段调入程序。由于这段代码并没有构成一个完整的程序（没有main函数），所以如果你试图像在前一个例子中那样编译并连接它，GCC将会报错，如下所示：

```
$ gcc -o goodbye message.c
/usr/lib/gcc/i486-linux-gnu/4.0.3/../../../../lib/crt1.o: In function
`_start':../sysdeps/i386/elf/start.S:115: undefined reference to `main'
collect2: ld returned 1 exit status
```

相反，我们需要告诉GCC不要执行任何额外的连接操作，而是使用GNU汇编器将源文件转换为目标代码后就结束。因为在这种情况下连接器并没有被执行，所以输出的目标文件不会包含作为一个Linux程序在被装载和执行时所必须包含的信息，但它可以在以后被连接到一个程序。

使用gcc的“-c”标记来编译支持库代码：

```
gcc -c message.c
```

这将告诉gcc驱动程序调用它内部的C编译器并将其输出传递给外部的GNU汇编器。这一过程的输出结果是一个名为message.o的文件，它包含适合连接到一个较大程序的已编译目标代码。

为了在一个较大的程序中使用消息函数goodbye_world，可以创建一个简单的示例包裹程序，它包含一个调用goodbye_world的main函数。

```
#include <stdlib.h>

void goodbye_world(void);

int main(int argc, char **argv)
{
    goodbye_world();
    exit(0);
}
```

这个文件将外部消息打印函数声明为一个不带参数的void函数。如你所知，这样的声明是必要的，否则GCC将假设任何未声明的外部函数为integer类型并隐含地将它们声明为这样。这可能会导致在编译时产生不必要的警告。你通常会遵循最佳的编程习惯，就像你以前做的那样，将这些定义放入自己的头文件中。

使用GCC编译这个包裹程序：

```
gcc -c main.c
```

现在你有了两个目标文件：message.o和main.o。它们包含能够被你的Linux工作站执行的目标代码。要从这个目标代码创建Linux可执行程序，你需要再一次调用GCC来执行连接阶段的工作：

```
gcc -o goodbye message.o main.o
```

GCC认识目标代码的.o后缀名，并知道应该如何为你调用外部GNU连接器。请记住GCC在默认情况下将把所有可执行文件命名为a.out，所以你需要在命令行中指定可执行程序名。在成功将多个源文件编译并连接进单个可执行文件后，你就可以以正常的方式来执行这个程序了。

```
./goodbye
Goodbye, World!
```

前面这些单独的步骤也可以简化为一个命令，这是因为GCC对如何将多个源文件编译为一个可执行程序有内置的规则。

```
gcc -o goodbye message.c main.c
./goodbye
Goodbye, World!
```

3. 使用外部函数库

GCC常常与包含标准例程的外部软件库结合使用，这些软件库为在Linux上运行的C程序提供必需的功能。这实际上是C编程语言设计上的一个副作用。作为一个特点鲜明的语言，它甚至连一些基本的任务，如I/O、读写文件或终端显示窗口也要依赖于某些标准的外部库例程来执行。

几乎每一个Linux应用程序都依赖于由GNU C函数库GLIBC所提供的例程。这个函数库提供了基本的I/O例程，如printf以及上例中的exit函数，后者用于请求Linux内核正常终止程序（事实上，不论你是否明确地调用它，像exit这样的函数都会被调用——我们在本书后面讨论Linux内核时还会介绍更多方面的内容）。

正如你将在以后章节中看到的那样，GNU C函数库构成Linux内核之上的一层薄层并提供了许多有用的例程，如果这些例程都由Linux内核自身来提供，付出的代价将相当昂贵（根据代码效率和增加的复杂性）。事实上，当你要求GCC执行编译任务时，GCC在默认情况下将假设GLIBC将被包括进

你的程序。由于这一过程发生在连接阶段，所以还需要你在应用程序源代码中添加函数库的头文件以提供库函数自身的原型定义。

给GNU C函数库的这一特殊待遇是由其几乎普遍的实用性以及几乎所有应用程序都会使用到它这一事实决定的。虽然我们很少会在编译源代码时不使用GLIBC，但我们的的确可以这样做。事实上，Linux内核就完全没有使用GLIBC，而是在内核中包含了它自己对许多标准C库函数的简化实现，这是因为GLIBC依赖于Linux内核来代表它执行各种服务。

通过遵循一些简单的准则，你就可以为Linux应用程序创建自己的软件库。我们将在这里对此进行解释，首先创建一个简单的程序trig.c，它以一个给定的角度值来计算各种三角函数值：

```
/*
 * Professional Linux Programming - Trig Functions
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX_INPUT 25

int main(int argc, char **argv)
{
    char input[MAX_INPUT];
    double angle;

    printf("Give me an angle (in radians) ==> ");
    if (!fgets(input, MAX_INPUT, stdin)) {
        perror("an error occurred.\n");
    }
    angle = strtod(input, NULL);

    printf("sin(%e) = %e\n", angle, sin(angle));
    printf("cos(%e) = %e\n", angle, cos(angle));
    printf("tan(%e) = %e\n", angle, tan(angle));

    return 0;
}
```

这个程序依赖于由系统数学库（作为Linux系统上GLIBC软件包的一部分——附带提一下，当你使用GCC编译普通应用程序时，它不会被自动包含）提供的外部数学函数。因此，在连接阶段，我们需要告诉GCC在它搜索库函数的路径中应该包括这个外部函数库：

```
gcc -o trig -lm trig.c
```

注意GCC的"-lm"选项，它告诉GCC查看系统提供的数学库（libm）。因为Linux和UNIX的系统函数库通常以"lib"为前缀，所以我们假设它存在。真正的函数库位置随系统的不同而不同，但它一般会位于目录/lib或/usr/lib中，在这些目录中还有数以百计的其他必需的系统函数库，你可能在后面会用到它们。

● 共享函数库与静态函数库

Linux系统上的函数库分为两种不同的类型：共享的和静态的。后者继承自往昔的UNIX系统，那

时所有的软件库都静态连接到使用这些库中例程的代码。每次当应用程序和静态连接的函数库一起编译时，任何引用的库例程中的代码都会被直接包含进最终的二进制程序。由于每个可执行程序都将包含标准例程的副本，所以导致它的容量较大。

现代Linux（和UNIX）系统在大多数情况下使用共享函数库。共享函数库和它对应的静态函数库包含相同的例程，但这些例程并不是直接插入每个要连接它的程序。相反，共享函数库包含每个库例程的单一全局版本，它在所有应用程序之间共享。这一过程背后所涉及的机制相当复杂，但主要依靠的是现代计算机的虚拟内存能力，它允许包含库例程的物理内存安全地在多个独立用户程序之间共享。

使用共享函数库不仅减少了文件的容量和Linux应用程序在内存中覆盖的区域，而且它还提高了系统的安全性。如今，软件中新的安全漏洞每天都会被发现，相应的补丁修复出来得也一样的快。如果安全问题存在于系统函数库中，那么大量使用这些函数库的应用程序将会突然成为你的安全梦魇。通过尽可能地使用共享函数库资源，你就可以在不对软件进行重新编译的情况下确保它是最新的——只需一个全局的函数库新就可以自动修复应用程序。

使用共享函数库的另一个好处是：一个被许多不同程序同时调用的共享函数库很可能会驻留在内存中，以在需要使用它时被立即使用，而不是位于磁盘的交换分区中。这有助于进一步减少一些大型Linux应用程序的装载时间。

● 一个共享函数库的例子

创建共享函数库的过程是比较简单的。事实上，你可以重新编译前面的多源文件示例代码以使用共享函数库，而不是将不同的源文件静态连接为一个单独的可执行程序。由于共享函数库会同时被许多不同的应用程序使用，所以我们需要以一种“位置无关”的方式来编译共享函数库代码（即，它可以在任意内存位置被装载并仍然可以执行）。

使用GCC的“-fPIC”选项重新编译源文件message.c：

```
gcc -fPIC -c message.c
```

“PIC”命令行标记告诉GCC产生的代码不要包含对函数和变量具体内存位置的引用，这是因为现在还无法知道使用该消息代码的应用程序会将它连接到哪一段内存地址空间。这样编译输出的文件message.o可以被用于建立共享函数库，我们只需使用gcc的“-shared”标记即可：

```
gcc -shared -o libmessage.so message.o
```

你可以让前面例子中的包裹程序main.c使用这里创建的共享函数库。与将消息打印函数连接到最终可执行程序goodbye不同，GCC可以通知连接器使用共享函数库资源libmessage.so：

```
gcc -o goodbye -lmessage -L. main.o
```

注意，我们使用“-lmessage”标记来告诉GCC驱动程序在连接阶段引用共享函数库libmessage.so。“-L.”标记告诉GCC函数库可能位于当前目录（包含程序源文件的目录）中，否则GNU的连接器会查找标准系统函数库目录，在本例的情况下，就找不到可用的函数库了。

你所建立的任何共享函数库都可以像你的Linux系统上安装的其他函数库一样使用，但它必须被安装到正确的位置以使这样的使用完全自动化。当有使用共享函数库的应用程序被加载时，现代Linux系统所提供的运行时动态连接器/加载器-ld-linux会被自动调用并在标准系统目录/lib和/usr/lib（当

然在许多系统中，我们也可以使用配置文件/etc/ld.so.conf来进行修改）下查找函数库。当你发布应用程序时，你需要在软件安装过程中将一些必需的函数库安装到系统可以在运行时自动搜索到的目录中。根据系统具体情况的不同，你可能还需要再次运行命令ldconfig。

你可以使用命令ldd来发现一个特定应用程序需要使用的函数库。ldd搜索标准系统函数库路径并显示一个特定程序使用的函数库版本。我们尝试将ldd使用到上面的例子中：

```
$ ldd goodbye
    linux-gate.so.1 => (0xfffffe000)
    libmessage.so => not found
    libc.so.6 => /lib/tls/libc.so.6 (0xb7e03000)
    /lib/ld-linux.so.2 (0xb7f59000)
```

库文件libmessage.so不能在任何一个标准搜索路径中找到，而且系统提供的配置文件/etc/ld.so.conf也没有包含一个额外的条目来指定包含该库文件的目录。因此，如果我们运行这个程序，它将产生如下所示的输出：

```
$ ./goodbye_shared
./goodbye_shared: error while loading shared libraries: libmessage.so: cannot open
shared object file: No such file or directory
```

如果你仅仅是想测试这个示例，那不用修改你的标准Linux系统设置或将库文件libmessage.so安装到系统函数库目录中，你只需设置一个环境变量LD_LIBRARY_PATH即可，它可以包含额外的函数库搜索路径。运行时连接器将搜索这些额外路径以发现没有在标准路径中找到的函数库。

首先设置一个适当的LD_LIBRARY_PATH，然后运行示例代码：

```
$ export LD_LIBRARY_PATH=`pwd`
$ ldd goodbye_shared
    linux-gate.so.1 => (0xfffffe000)
    libmessage.so => /home/jcm/PLP/src/toolchains/libmessage.so (0xb7f5b000)
    libc.so.6 => /lib/tls/libc.so.6 (0xb7e06000)
    /lib/ld-linux.so.2 (0xb7f5e000)
$ ./goodbye
Goodbye, World!
```

4. GCC选项

GCC拥有数不清的命令行选项标记，它们几乎可以被用于控制编译过程中的每一个方面以及GCC可能会依赖的任何外部工具的操作。通常不会在编译应用程序时指定太多的选项，但当在自己的项目中使用GCC时，你会很快习惯使用GCC提供的各种调试和警告选项。

GCC选项可以被分为如下几类：

- 一般选项
- 语言选项
- 警告级别
- 调试
- 优化
- 硬件选项

● 一般选项

一般选项包括指定GCC产生的可执行文件名以及是否需要GCC完成整个编译过程或在它完成基本编译之后就终止。你已看到可以在命令行上指定“-c”标记来告诉GCC只执行编译和汇编而不执行连接。我们还可以通过指定“-s”标记来让GCC在执行完编译之后就停止并输出汇编语言代码。

● 语言选项

特定语言选项允许你控制GCC对当前源文件所使用的编程语言的相关标准的解释。在使用C语言的情况下，GCC默认会使用它自己对ANSI C的扩展版本（GNU89或GNU99），这个版本支持一些受到程序员欢迎的较为宽松的约定，但可能并不严格符合官方的C89或C99语言规范。你可以使用这些特定语言选项来指定语言行为。几个比较常见的选项如表2-1所示：

表2-1 常见的可指定语言行为的语言选项

-ansi	禁止使用与C90规范不兼容的某些GCC特征，如关键字asm和typeof（在本章后面将介绍更多这方面的内容）
-std	指定选项“-std=c89”将告诉GCC使用C89 ANSI C语言规范。默认的GNU89包括GNU对C89标准的各种扩展。最新的GCC版本支持C99和其扩展版本GNU99
-fno-builtin	GCC在默认情况下包含一些常用函数如memcpy，甚至printf的内置版本，它们比外部函数库中对应的函数更有效。你可以通过指定这个选项来禁止使用这些函数的内置版本

除了一些极端情况（例如，你正在编写自己的C函数库）或你正在运行自己的语言验证测试外，你通常不需要改变语言选项。

● 警告级别

GCC提供了各种警告级别，它们可以帮助追查某些类型的不良编程习惯，或当你误用了某些你已告诉GCC不要提供的语言特征时，它们会向你发出警告。几个比较常见的警告选项如表2-2所示：

表2-2 常见的警告选项

-pedantic	这个选项要求GCC严格解释有关的C语言标准，并在程序没有遵循使用的标准时发出警告。另一个相关的选项“-pedantic-errors”将在编译时将这些警告信息强制为错误的编译结果
-Wformat	这是许多相关选项中的一个，它要求GCC查看每个函数调用以确定它们是否可能造成运行时问题。“-Wformat”将检查printf系列函数的错误使用，而“-Wformat-security”将对一些潜在的安全漏洞发出警告
-Wall	这启用了绝大多数警告选项，并将产生冗长的输出

养成在编译自己的程序时总是使用“-Wall”选项的习惯将对你大有帮助。此外，我们还建议你尽可能考虑使用“-pedantic-errors”选项，因为这将有助于在许多常见问题发生之前孤立它们，同时还有助于确保你的代码是尽可能接近标准规范的。

一个使用了额外选项的编译示例如下所示：

```
gcc -o hello -Wall -pedantic-errors hello.c
```

● 调试

GCC提供了各种选项来方便你对应用程序的调试，正如你将在本章后面看到的一样。其中主要的是“-g”选项，它将在可执行程序的调试数据段中包含调试信息。这个信息可以被GDB用于执行源代

码级别的调试以及方便在本书后面提到的其他更高级的调试方式。

为了在编译前面的应用程序时提供有用的调试信息以及启用适当的警告，你可以使用如下命令：

```
gcc -g -o hello -Wall -pedantic-errors hello.c
```

● 优化

GCC能够对它所编译的代码执行各种优化。可以使用的选项包括数字优化级别“`-O0`”到“`-O3`”——从不进行优化到要求GCC尽可能地执行优化。根据优化级别的不同，它们分别会启用或禁用各种额外选项。

优化并不是没有代价的。它通常会大大增加编译时间和编译过程中编译器所需要的内存资源。它也不能保证不增加最终可执行文件的大小，这是因为它通常会将循环或函数展开，使它们以内联的方式循环而不是通过函数调用，这样做将显著地提高性能。如果你想对可执行文件的大小进行优化，可以使用“`-Os`”选项标记。

要注意的是，有些优化是必要的。GCC默认不会使用内联函数^①（在本章后面会介绍更多这方面的内容），除非我们在调用它时使用一组适当的优化标记。但在编译Linux内核时，我们可能需要选择降低使用的优化级别以避免引起问题。真正要降低优化级别的原因之一是想避免GCC为了在现代处理器上更有效地执行指令而对指令重新排序，虽然这样做是有益的，但指令的重新排序将使得一些调试更加困难，对非常大型和复杂的程序（如Linux内核）尤其如此。

● 硬件选项

GCC是一个可移植性非常强的C编译器，它支持多种不同的硬件目标。一个硬件目标是一种类型的机器，其特性由安装在其中的微处理器类型决定。你的工作站很可能是基于Intel兼容处理器的，该处理器有各种不同的模型，每种模型又各自具备不同的能力。当编译代码时，你的GCC版本在默认情况下使用你的Linux发行版厂商所配置的处理器模型。

如果你编译的代码只用在自己的计算机上（或类似类型的机器上），你并不需要设置任何硬件选项以让编译的代码成功执行。但是，当你使用交叉编译器或希望编译的代码能够在类型完全不同的机器上执行时，情况就不同了。

根据所用硬件的不同，你可以使用表2-3中列出的部分或全部选项：

表2-3 GCC的硬件选项

<code>-march</code>	要求GCC针对特定的CPU模型生成代码，其中将包含特定模型的指令
<code>-msoft-float</code>	要求GCC不要使用硬件浮点运算指令，而是依靠函数库调用来执行浮点运算。当为缺乏硬件浮点运算支持的嵌入式设备编译软件时，这是最常用的选项
<code>-mbig-endian</code> <code>-mlittle-endian</code>	指定硬件目标使用的是大头字节序还是小头字节序模式。这仅适用于特定类型的目标处理器（如PowerPC），因为它可以同时工作在两种模式下。而大多数Intel CPU使用的是小头字节序
<code>-mabi</code>	有些处理器可以支持各种不同的应用程序二进制接口（ABI），这可能需要你指定一个非默认的ABI版本。当一个处理器系列同时支持32位和64位扩展时，也会出现类似的情况（用不同的选项），我们需要在编译时指定它

^① 内联一个函数意味着将一个被调用函数的可执行代码全部插入到函数调用的位置。由于并没有执行函数调用，所以我们也不用承受调用它所需要付出的代价，尤其当我们调用一个非常小的函数时更是如此。

附带说明一下，一些开发人员（和越来越多的渴望将他们的机器性能用到极致的爱好者）会使用GCC的硬件选项标记来改变默认的处理器目标或设置特定于他们本地CPU模型的其他选项。这样做，他们可以让本地编译的应用程序获得最大程度的性能提高，付出的代价是降低了程序的可移植性。但是，如果你从未打算发布你的应用程序，或你是一个嵌入式Linux开发者，希望能利用一切可能的硬件性能优化，那这并不是一个问题。

● 更详细的文档

介绍GCC可用选项的完整文档见本地GCC安装的在线帮助。要阅读GCC文档，你可以从任何终端窗口使用Linux的man命令或GNU的info工具。传统的man页面包含了对命令选项的概述，你可以使用如下命令来查看：

```
man gcc
```

自由软件基金会通常并不鼓励在自己的项目中使用man页面，而是建议使用它认为更灵活的GNU的info系统。可以使用如下命令通过GNU info来查看扩展的文档：

```
info gcc
```

2.3 GNU 二进制工具集

GNU GCC是任何Linux开发环境中可见的也是最明显的一个组成部分，但它却非常依赖于一些外部工具来实际代表Linux开发者执行有用的工作。这些外部工具许多是由GNU的二进制工具集提供的，它是一组工具的集合，用于产生和控制Linux中的二进制应用程序代码。

每个二进制工具都有其特定的目的并出色地完成某一项任务，这遵循了标准的UNIX哲学。有一些工具的用途较为明显，但对于编译大型复杂的软件项目（如Linux内核）来说，每一个工具都是必需的。成千上万的开发人员每天都会依赖这些工具来完成工作，但其中许多人可能甚至都没有意识到这一点。即便你不会直接使用这些工具，但了解它们以及它们所提供的一般功能还是非常重要的。

2.3.1 GNU 汇编器

GNU汇编器——它也许是自然的对应GNU编译器集的工具了。它负责把已编译的C代码（以汇编语言形式表示）转换为能够在某一特定目标处理器上执行的目标代码。GNU as支持许多不同种类的微处理器，其中包括Linux工作站常用的Intel IA32（大部分人称它为“x86”）系列。

安装在你的工作站上的GNU as版本是由Linux发行商针对你的系统所基于的特定处理器类型预配置好的，当然你也可以安装其他配置的版本（更多信息请见2.7节）。但无论你安装的是什么配置的汇编器，它在所有平台上的表现都是类似的。

为了对汇编器的使用进行尝试，你可以要求GCC输出前面的Hello World示例的汇编代码，然后将它汇编为可执行目标代码。要产生一些示例汇编代码，请在编译源文件hello.c时指定“-S”选项标记。

```
gcc -S hello.c
```

GCC将把你的程序的汇编语言版本输出到文件hello.s。下面是运行在一个Intel工作站上的GCC 4.0所产生的输出：

```

.file    "hello.c"
.section     .rodata
.LC0:
.string  "Hello, World!"
.text
.globl main
.type   main, @function
main:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $8, %esp
    andl   $-16, %esp
    movl   $0, %eax
    addl   $15, %eax
    addl   $15, %eax
    shrll  $4, %eax
    sall   $4, %eax
    subl   %eax, %esp
    movl   $.LC0, (%esp)
    call    puts
    movl   $0, %eax
    leave
    ret
.size   main, .-main
.ident  "GCC: (GNU) 4.0.3 20060115 (prerelease) (Debian 4.0.2-7)"
.section     .note.GNU-stack, "",@progbits

```

要学习在Linux上使用GNU的一些有趣的事情，你并不需要理解上面列出的每一行汇编语言代码的含义。目前有许多优秀的关于Linux上Intel汇编语言编程的参考书，例如*Professional Assembly Language* (Wrox)。你也可以使用Linux来为许多其他体系结构编写汇编语言代码。

在上面列出的汇编语言代码中，请注意源代码是如何通过“`.section`”命令分为多个段落的。其中每一个段落将在稍后由GNU连接器及其相关的连接器脚本构成程序可执行代码中的各个部分。全局符号也在汇编语言代码中进行了标注，它包括对外部库函数的调用，如`puts`:

```
call puts
```

函数`puts`由GNU C语言函数库提供。这个函数并不是程序的一部分，但如果以后它通过标准系统函数库连接到程序（例如，当调用GNU连接器来产生可运行的可执行程序文件），程序就可以使用它了。

你可以使用GNU汇编器`as`来编译源代码`hello.s`，如下所示：

```
as -o hello.o hello.s
```

这将产生文件`hello.o`，它包含针对指定汇编语言源文件的可执行目标代码。请注意，你现在还不能在自己的Linux工作站上实际地运行这个文件，这是因为它还没有被GNU连接器处理，所以它还没有包含任何Linux系统在试图加载并开始代表你执行应用程序时所必需的额外信息。

2.3.2 GNU 连接器

连接是在Linux系统上产生可工作的可执行程序的一个重要阶段。要建立一个可执行程序，源代码必须首先被编译、汇编，然后被连接到一个可以被目标Linux系统理解的标准容器格式（standard container format）。在Linux和现代UNIX/类UNIX系统中，这个容器格式是ELF (Executable and Linking

Format, 可执行和连接格式)。它既是已编译目标代码和应用程序的文件格式选择, 也是GNU ld的格式选择。

Linux应用程序存储在ELF文件中, 它由许多段落构成。其中包括程序自身的代码段和数据段以及各种和应用程序本身相关的元数据。如果没有经过适当的连接阶段, 一个可执行程序就不会包含足够的额外数据以让Linux运行时装载器可以成功地装载并执行该程序。虽然程序代码可能已存放在一个特定的目标代码文件中, 但这并不足以让它成为一个有用的程序。

除了产生可以在Linux系统上运行的可执行程序以外, 连接器还负责确保任何必需的环境设置代码已位于每个要在Linux目标机器上装载并运行的可执行文件的正确位置。在使用GNU C编译代码的情况下, 这个启动代码包含在文件crtbegin.o(位于GCC安装目录中)中, 在编译程序时它将自动被连接到应用程序。另一个类似的文件crtend.o则提供了在应用程序终止时执行清理退出任务的代码。

连接器的操作

GNU连接器在处理各种目标代码文件并产生所要求的输出时遵循一系列被称为连接器脚本的预编写命令。你的工作站应该已在诸如/usr/lib/ldscripts这样的目录下安装了一些脚本。每当ld需要创建一个特定类型的Linux可执行文件时, 它就会使用其中的脚本。你可以查看一下安装在自己系统中的这些脚本以了解隐藏在每个Linux程序背后的复杂性。

在使用GCC编译普通应用程序时, 你通常不需要直接调用GNU连接器ld。它的操作相当复杂并且最终决定于你所安装的GCC具体版本和各种外部软件库的位置和版本。如果你想要修改Linux内核的内存布局(当你想为自己的硬件设施创建一个自己的Linux版本时), 你就需要理解连接器是如何使用连接器脚本的。

在本书后面讨论Linux内核时, 我们将介绍更多有关连接器使用的内容。

2.3.3 GNU objcopy 和 objdump

GNU二进制工具集中包括几个工具, 它们是专用于控制和将二进制目标代码从一种格式转换为另一种格式的。这些工具被称为objcopy和objdump, 很多在Linux机器上使用的底层软件在其编译阶段都非常依赖于这些工具, 甚至一些普通应用程序在调试时也需要用到它们。

工具objcopy可以用于从一个文件拷贝目标代码到另一个文件, 并在这一过程中执行各种转换。通过使用objcopy, 我们可以在不同的目标代码格式之间进行自动的转换并操纵这一过程中的内容。objdump和objcopy都建立在非常灵活的bfd二进制操纵函数库之上。这一函数库被许多便利的工具所利用, 它们可以以任何你想象的方式来操纵二进制目标文件。

工具objdump用于方便查看可执行文件的内容, 并通过执行各种任务使得这一可视化过程更加容易。通过使用objdump, 你可以检查本章前面使用的Hello World示例代码的内容。

```
$ objdump -x -d -S hello
```

这些命令开关要求objdump显示二进制文件hello的所有头(-x), 试图反汇编任一可执行段落的内容(-d)并将程序的源代码与其对应的反汇编代码混合显示(-S)。最后一个选项只会在某些情况下产生可读的输出。这通常需要使用完整的调试信息(使用GCC命令的“-g”开关)来编译源文件, 并且没有进行任何GCC指令调度优化(为了增加可读性)。

objdump产生的输出可能与下面的输出结果类似。这里，你可以看到在一个运行Linux的PowerPC平台上通过objdump输出的Hello World示例程序头部分的内容：

```

hello:      file format elf32-powerpc
hello
architecture: powerpc:common, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x100002a0

Program Header:
  PHDR off    0x00000034 vaddr 0x10000034 paddr 0x10000034 align 2**2
    filesz 0x00000100 memsz 0x00000100 flags r-x
  INTERP off   0x00000134 vaddr 0x10000134 paddr 0x10000134 align 2**0
  STACK off   0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
    filesz 0x00000000 memsz 0x00000000 flags rwx

Dynamic Section:
  NEEDED      libc.so.6
  INIT        0x10000278
  FINI        0x100007ac
  HASH        0x10000164
  PLTGOT     0x100108fc

Version References:
  required from libc.so.6:
    0xd696910 0x00 02 GLIBC_2.0

Sections:
Idx Name          Size    VMA       LMA       File off  Algn
  0 .interp       0000000d 10000134 10000134 00000134 2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version 0000000a 10000228 10000228 00000228 2**1
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init         00000028 10000278 10000278 00000278 2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .text         0000050c 100002a0 100002a0 000002a0 2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .fini        00000020 100007ac 100007ac 000007ac 2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE

```

ELF头表明这是一个32位的PowerPC目标文件。因为Linux是一个可移植的操作系统，所以作者在各种不同的Linux平台（如PowerPC）上编写和测试本书中的示例。由于ELF本身也是一种跨平台的二进制标准，所以在objdump的输出中没有提及Linux。理论上来说，你可以在任何一台遵循同一个ABI（Application Binary Interface，应用程序二进制接口）并有能力装载ELF文件及其函数库的PowerPC操作系统中运行这段代码。

如果我们使用gcc命令的“-g”调试标记来编译源代码，那么objdump的输出还可以包含应用程序源代码的完整反汇编代码。下面是在一台IA32（x86）Linux工作站上的objdump的部分输出结果，它显示了main函数的反汇编代码和对应的源代码：

```

int main(int argc, char **argv)
{
    8048384:      55                      push   %ebp
    8048385:      89 e5                  mov    %esp,%ebp
    8048387:      83 ec 08              sub    $0x8,%esp
    804838a:      83 e4 f0              and    $0xfffffff0,%esp
    804838d:      b8 00 00 00 00        mov    $0x0,%eax
    8048392:      83 c0 0f              add    $0xf,%eax
    8048395:      83 c0 0f              add    $0xf,%eax
    8048398:      c1 e8 04              shr    $0x4,%eax
    804839b:      c1 e0 04              shl    $0x4,%eax
    804839e:      29 c4                  sub    %eax,%esp

    printf("Hello, World!\n");
80483a0:      c7 04 24 e8 84 04 08    movl   $0x80484e8,(%esp)
80483a7:      e8 fc fe ff ff          call   80482a8 <puts@plt>

    return 0;
80483ac:      b8 00 00 00 00        mov    $0x0,%eax
}
80483b1:      c9                      leave
80483b2:      c3                      ret

```

ELF文件的结构已超出了本书介绍的范围，但是，如果想要理解Linux应用程序的二进制文件布局，至少应该熟悉objdump工具的使用。在objdump的文档和在线帮助中还有更多关于它的用法示例。objdump的输出可以被用于帮助某些调试过程或满足你的好奇心。你在稍后将会看到GDB也可以提供类似的信息，但有时（尽管这种情况很少）GDB可能并不适用。

2.4 GNU Make

正如前面已指出的，Linux机器上的大多数软件是通过GNU make来编译的。从本质上来说，GNU make只是一个简单的依赖性跟踪工具，它遵循一系列的规则以确定对一个大型项目中每个单独源文件必须执行的动作。当你在本章前面学习组成GNU工具链的其他各种编译工具时，你已看到几个简单的Makefile规则。

下面是一个更复杂的Makefile文件，它可用于编译迄今为止的所有示例代码。

```

# Makefile to build Toolchains examples

CFLAGS := -Wall -pedantic-errors

all: hello goodbye trig

clean:
    -rm -rf *.o *.so hello goodbye trig

hello:

goodbye: main.o message.o

trig:
    $(CC) $(CFLAGS) -lm -o trig trig.c

```

Makefile文件针对可能的目标定义了一系列的规则。其中包括实际编译三个示例（Hello、Goodbye

和Trig)的规则,以及定义如何在发布他们的软件给其他开发者之前清理源代码的规则。一条规则是由一个命名标签后跟随一个冒号和一个依赖关系名单组成的。这个依赖关系名单必需首先得到满足,然后才能考虑结果是否成功。如果make通过规则本身不能自动确定如何去做,在该规则之后就需要跟随一些具体应该使用的命令。

GNU Make的规则可以定义的内容非常广泛,从琐碎的源文件依赖到复杂的层次依赖均可,但它们最终总是可以被细分为简单的命令序列。除了定义规则以外,make还支持定义变量、条件和许多其他你可能期望从一个普通编程语言中获得的特征。GNU Make还理解许多标准变量,如\$CFLAGS(这个标记将被传递给C编译器)和\$CC(默认设置为gcc)。

在上面的Makefile文件中,默认动作(all)试图通过使用针对每个示例程序的规则来编译这三个程序——Hello、Goodbye和Trig。因为hello规则没有定义任何依赖关系,所以make将自动假定它需要将源文件hello.c编译为可执行文件hello。对于goodbye规则来说,make在连接两个依赖文件到一个最终可执行程序之前首先需要自动编译这两个依赖文件。

Trig示例程序的编译过程与其他程序的编译过程稍有不同。在trig规则之后没有指定任何依赖关系,但在这条空规则的下一行给出了一个编译器命令。GNU Make将把它作为一个shell命令来执行,并替换放在括号中的变量扩展。因此,\$(CC)将成为gcc,而\$(CFLAGS)将扩展为在Makefile文件开头定义的标记内容。要注意的是,当给出这样一个明确的命令时,我们有必要同时包含CFLAGS和CC。

你可以通过阅读安装在本地工作站上的在线文档或通过访问自由软件基金会的网址来了解更多有关Make规则和如何使用GNU Make的信息。但更好的方式是下载一个大型的自由软件项目的源代码,通过阅读它所使用的Makefile文件,你将很快弄清楚GNU make可以使用的复杂规则集。

在John R. Levine所著的非常幽默的图书*Linkers and Loaders*(Morgan Kaufman, 2000)中,你将找到更多有关ELF文件格式的设计和实现以及标准工具如GNU连接器操作的内容。

2.5 GNU 调试器

GNU调试器(GDB)是Linux开发者的工具箱中功能最强大的工具之一。GDB不仅是一个数以千计的用户每天都在使用的灵活的工具,而且它也是世界上最普及的调试器之一。这在某种程度上也是因为有数众多的发行商选择在他们的产品中使用GDB而不是自己从头开始重新实现它的功能。所以,不论你从哪里购买工具,它都很有可能在其内部提供某种形式的GDB。

作为一个独立的软件包,GDB通常作为现代Linux发行版中的任意一个开发工具的一部分安装到你的系统中。使用GDB非常简单,我们以前面的三角函数程序作为一个简单的示例,可以通过在gdb中运行应用程序来调试它,使用的命令如下所示(请确认程序在编译时使用了gcc的“-g”调试标记):

```
$ gdb trig
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

(gdb)
```

GDB显示其默认提示符并等待你的命令。首先使用list命令来查看这次调试会话的一部分源代码：

```
(gdb) list
8      #include <math.h>
9
10     #define MAX_INPUT 25
11
12     int main(int argc, char **argv)
13     {
14         char input[MAX_INPUT];
15
16         printf("Give me an angle (in radians) ==> ");
17     }
```

此时程序还没有运行，你可以使用run命令来运行它。在gdb中运行一个程序之前，你最好能在代码中至少插入一个断点。这样做之后，当程序运行到源代码特定的某一行时，GDB将暂停程序的运行以允许你执行任何有助于你调试的查询。人们习惯于在main函数的入口插入第一个程序断点，然后在程序中的其他位置插入其他断点。

在示例代码中插入一个断点，然后运行它，如下所示：

```
(gdb) break main
Breakpoint 1 at 0x8048520: file trig.c, line 17.
(gdb) run
Starting program: /home/jcm/PLP/src/toolchains/src/trig

Breakpoint 1, main (argc=1, argv=0xbfd87184) at trig.c:17
printf("Give me an angle (in radians) ==> ");
```

GDB将在trig程序运行到main函数的第一条指令时暂停程序，在本例中，这条指令位于源文件trig.c的第17行。除了在程序运行到断点后GDB自动显示的对printf函数的调用以外，我们还可以使用list命令来显示main函数中的源代码。每一行代码可以使用step和next命令来单步执行——前者会在执行每一条机器指令之后暂停，后者的执行方式类似，但它不会进入外部函数的内部，而是把函数调用语句当作一条普通语句来执行。

我们使用next命令执行printf语句，并在用户通过命令行输入角度之后停止，如下所示：

```
(gdb) next
18             if (!fgets(input, MAX_INPUT, stdin)) {
(gdb)
Give me an angle (in radians) ==> 3.14
angle = strtod(input, NULL);
```

GDB在它调用库函数strtod()将字符串转换为双精度浮点数之前暂停了程序的运行。你可以使用GDB的print命令在调用这个函数前后查看angle变量的值：

```
(gdb) print angle
$1 = -4.8190317876499021e-39
(gdb) next
23             printf("sin(%e) = %e\n", angle, sin(angle));
(gdb) print angle
$2 = 3.1400000000000001
```

在默认情况下，程序的输入和输出与GDB的命令输入共用同一个终端。你可以使用GDB的`tty`命令来重定向程序的输入/输出到一个特定的Linux终端。例如，在图形化桌面环境中，你可以在一个X终端窗口中调试应用程序，并将它的输入和输出传递到另一个终端窗口。你可以使用`shell`命令行中同名的`tty`命令来查找一个X终端的终端号。

最后，你可以使用`continue`命令（简写为`c`）让程序一直运行，直到它终止（或遇到另一个断点）：

```
(gdb) c
Continuing.
sin(3.140000e+00) = 1.592653e-03
cos(3.140000e+00) = -9.999987e-01
tan(3.140000e+00) = -1.592655e-03
.
Program exited normally.
```

无论何时，当你需要GDB的使用帮助时，你只需输入“`help`”就可以获得一个帮助选项，它根据命令的类别将它们分在不同的部分中。

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

GDB的另一个常见用途是用于调试程序的核心转储（`core dump`）——当应用程序崩溃时包含应用程序当时状态的文件，它用来确定崩溃的具体情况，这和飞机失事后使用的飞机黑匣子记录器非常类似。每当应用程序犯了错误，如非法存取内存或访问NULL指针指向的内容时，Linux将强制终止程序的运行并自动生成这些核心文件。

你选择的Linux发行版在默认情况下可能并不会创建核心转储文件。在这种情况下，你可能需要修改系统设置来要求它这样做，具体细节请参考你的发行商的系统文档。

Linux内核本身会通过在目录`/proc`中的只读文件`kcore`的不断更新来显示其内部状态。你可以对这个伪核心转储使用`gdb`来获得你的Linux工作站的当前状态的一些有限的信息。例如，你可以查看从Linux内核的角度看到的当前时间，它是从系统启动开始计算的统计时器滴答数，如下所示：

```
# sudo gdb /lib/modules/`uname -r`/build/vmlinux /proc/kcore
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

Core was generated by `BOOT_IMAGE=linux ro root=303'.
#0 0x00000000 in ?? {}
(gdb) print jiffies
$1 = 72266006
```

因为jiffies的值在调用GDB时已被缓存，所以，如果试图重新显示该值，将获得相同的结果——需要重启GDB来确认这个值已改变了。

这个GDB命令依赖于目录/lib/modules中的当前内核版本子目录下的一个符号连接（该符号连接指向的目录用于编译内核）并要求GDB使用文件/proc/kcore中输出的伪内核文件映像。如果你没有在你的工作站上编译过内核，那么由于在符号连接所指向的目录中不存在vmlinux文件，你将不能运行这个命令。在本书的后面你还将学习到更多有关编译和调试Linux内核的内容。

GDB可以使用的命令比在本章中简要介绍的这些命令多得多。事实上，要想完整地涵盖GDB包含的所有功能，需要一本比本书厚得多的图书来介绍。我们鼓励你使用GDB进行实验，在本书的其余部分，我们也会在需要的时候再次使用它。

2.6 Linux 内核和 GNU 工具链

GNU工具链被世界上无数的软件项目所使用，但几乎没有哪个软件项目像Linux内核那样淋漓尽致地使用了这些工具所提供的功能。正如你将在本书后面看到的那样，Linux内核极度依赖于GCC中的许多语言扩展以及许多其他常用系统工具的GNU扩展。不依赖GNU工具链来编译Linux内核在理论上是可行的，但在实践中很少这样做。

2.6.1 内联汇编

底层Linux内核代码会经常使用GNU C编译器中的一个扩展以支持内联汇编代码。从字面上看，这就是汇编语言代码，它插入到构成大部分内核的普通C函数中。虽然内核的目的是为了尽可能地可移植——因此它大多是由平台独立的C语言代码所编写——但有些操作只能用机器特定的指令来执行。内联汇编有助于实现这一过程。

下面是一个简单的C函数示例，其中就包含了一些内联汇编语言代码：

```
void write_reg32(unsigned long address, unsigned long data) {
    __asm__ __volatile ("stw %0,0(%1); eieio"
                       : // no output registers here
                       : "r" (data), "r" (address));
}
```

这个函数有两个参数——一个地址和一些数据。在`__asm__`属性标记之后的PowerPC汇编语言将传递给函数的数据存储在指定的内存地址中，最后以特殊的机器特定的指令（`eieio`）结束，该指令强制硬件提交写入某些硬件设备的值，而不论该值是什么。`__volatile__`标记告诉GCC不要试图优化这段示例代码，因为它必须按照它编写的顺序来执行。

这个示例代码需要一些机器特定的处理器寄存器在汇编语言代码执行之前存放`data`和`address`变量。请注意，示例代码中的`r`标记表明`data`和`address`变量将只能被内联汇编语言代码段读取而不能被它更新。如果内联汇编语言命令需要改变它的输入寄存器，我们就需要使用`w`标记使GCC知道这些寄存器将要被修改。

你并不需要关注本例中所使用的具体汇编语言，而应该留意内联汇编所使用的一般格式：

```
__asm__ ( Machine specific assembly instructions
          : Machine specific registers affected by the instruction
          : Machine specific registers required as inputs
```

你将在Linux内核的源代码中找到更多关于内联汇编的例子。

2.6.2 属性标记

Linux内核非常依赖GCC的属性。这些属性是一些和源代码一起使用的特殊标记，它们向GNU C编译器提供额外的信息以告诉它应该如何特别地处理内核源代码。属性包括`nonnull`和`noreturn`，前者告诉编译器这个函数不能使用NULL参数，后者告诉编译器这个函数完全可以不返回。

下面是Linux内核中的一个属性定义的例子：

```
#define module_init(initfn)
    static inline initcall_t __inititest(void) \
    { return initfn; }
    int init_module(void) __attribute__((alias(#initfn)));
```

上面的代码定义了`module_init`函数，它被每一个Linux内核模块（LKM）所使用，当模块第一次被装载时，它用来声明将要运行的函数。在本例的众多参数中，`alias`属性用于为模块初始化函数设置一个别名`#initfn`。今后每当一个头文件需要提及这个函数时，它都可以使用宏`#initfn`来代替完整的函数名。

Linux内核函数经常会利用`section`属性，它用于指定某些函数必须被放置在Linux内核的二进制文件的特定段中。内核还会用到`alignment`属性，它强制对变量声明进行精确的内存对齐。出于对高性能的要求，Linux内核中的变量的内存对齐通常是非常严格的。这有助于确保有效地在主存中传输某些数据。

属性可以帮助各种代码分析工具（如Linus Torvald的sparse源代码检查器）来推测Linux内核中所使用的特定函数和变量的附加信息。例如，标记了`nonnull`属性标记的函数参数将被检查它是否曾经被传递了NULL值。

2.6.3 定制连接器脚本

Linux内核和其他底层软件非常依赖于连接器脚本来创建有着特定二进制映像布局的可执行文件。之所以这样做有很多重要的理由，其中一个理由是我们希望在内核二进制映像中将某些相关的内核特征分组到逻辑段中。例如，Linux内核中的某些函数有`_init`或`_initdata`标记。这些标记都被定义为GCC的属性，它们的作用是将这类函数分组到内核中的一个特殊段。

一旦Linux内核启动成功，它就不再需要有init之类的标记的代码和数据了，因此用于存储它们的内存也将被释放。之所以能实现是因为这类代码和数据都被存储在内核的一个特殊段中，它可以作为一大块可回收的内存被释放。这些特殊段在最终内核二进制映像中的物理位置由专门的内核连接器脚本确定，它在其中包含了这样的考虑。

有时候，精确的二进制映像布局是由运行内核的目标机器的硬件决定的。许多现代微处理器期望在内核映像的精确偏移位置找到某些底层的操作系统函数。这包括硬件异常处理函数（用于响应某些同步或异步处理器事件的代码）以及许多用于其他用途的函数。由于Linux内核映像是在机器的物理内存的精确偏移位置被装载的，所以我们可以借助于连接器技巧来满足这一需求。

Linux内核针对每一个它要支持的体系结构使用一个特定的连接器脚本。请查看本地Linux内核源代码目录中arch/i386/kernel/vmlinux.lds文件的内容，你将看到如下的片段：

```
.text : AT(ADDR(.text) - (0xC0000000)) {
*(.text)
.= ALIGN(8); __sched_text_start = .; *(.sched.text) __sched_text_end = .;
.= ALIGN(8); __lock_text_start = .; *(.spinlock.text) __lock_text_end = .;
.= ALIGN(8); __kprobes_text_start = .; *(.kprobes.text) __kprobes_text_end = .;
```

这些条目执行各种不同的任务，详细的解释请看GNU的连接器文档。在本例中，它指定所有内核代码从内存的虚拟地址0xC000_0000开始，而且几个特定符号必须按8字节边界对齐。完整的连接器脚本是非常复杂的，因为许多平台对Linux内核的要求非常复杂的。

2.7 交叉编译

Linux上的大多数软件开发都发生在与最终运行该软件的机器的类型相同的机器上。一个工作在Intel IA32 (x86) 工作站上的开发者将为拥有同类型基于Intel工作站的顾客（或自由软件用户）编写应用程序。

但对于嵌入式Linux开发者来说，事情就不是这么容易了。他们必须为各种不同类型的机器开发软件，每一种机器可能是在一个完全不同的处理器架构上运行Linux。而且对于面向个人的设备如PDA和手机而言，由于它们没有提供足够的处理能力或存储空间，所以直接在它们之上进行软件开发是不可行的。相反，我们采用的方法是在更强大的Linux主机上对软件进行交叉编译。

交叉编译（或交叉建立）是这样一种过程，它在一种机器结构下编译的软件将在另一种完全不同的机器结构下执行。一个常见的例子是在基于Intel的工作站上为运行在基于ARM、PowerPC或MIPS的目标设备编译软件。幸运的是，GNU工具使得这一过程所面临的困难要比听起来小得多。

GNU工具链中的一般工具通常都是通过在命令行上调用命令（如gcc）来执行的。在使用交叉编译的情况下，这些工具将根据它编译的目标而命名。例如，要使用交叉工具链为PowerPC机器编译简单的Hello World程序，你可以运行如下所示的命令：

```
$ powerpc-eabi-gcc -o hello hello.c
```

请注意交叉编译的目标powerpc-eabi是如何成为特定工具名前缀的。虽然工具的准确名称在很大程度上取决于它所面对的特定目标设备，但类似的命名约定也适用于交叉工具链中的其他工具。许多嵌入式Linux厂商销售各种你可以使用的设备专用工具链，当然你也可以根据下一节中的示例建立自己的工具。

除了直接使用交叉工具链以外，你可能经常会发现自己需要编译那些大型的基于自动编译工具如GNU Autoconf的已有项目。在这种情况下，你需要告诉已有的configure脚本，你要为某一特定的目标交叉编译一些软件，你将发现表2-4中的选项标记很有用。

表2-4 交叉编译时的选项标记

--build	配置为以指定的主机类型进行编译
--host	交叉编译运行在指定主机类型上的程序

2.8 建立 GNU 工具链

你无疑已经认识到GNU工具链是由许多单个的组件构成的，它们彼此独立开发但又必须协同工作以实现特定的目标。这些单个的GNU工具本身也必须在某个时候被从因特网上提供的源代码编译为可执行文件。这是一个非常费时的过程，Linux发行商已花了很多时间来解决它，并已为你的本地Linux工作站完成了这一任务，所以你就不需要再做一次了。

但也有时候现有的工具都不适合于手头的任务。有很多原因会导致出现这种情况，包括：

- 需要一个比发行商提供的工具更新的版本。
- 为一个特定的或不常见的目标交叉编译应用程序。
- 为单个工具修改特定的编译时选项。

在这些情况下，你需要自己从头开始获得并编译GNU工具链或为此付费购买需要的工具。从头开始编译一个工具链是一件非常困难和费时的过程，因为你需要针对不同的目标处理器为工具打上许多补丁，而且还要考虑到工具自身之间的相互依赖关系。为了建立一个工具链，你需要将下面列出的单个组件组合到一起：

- GCC
- binutils
- GLIBC
- GDB

为了建立一个能够编译一般用户程序的GCC，情况会变得更糟。因为你需要一个预编译的GNU C函数库，但为了获得一个这样的函数库，你需要先有一个编译器。为此，整个建立过程实际上被分成多个阶段，首先建立一个最小的GCC以便编译GLIBC（它本身还需要一份Linux内核头文件的拷贝），然后再重建一个能够编译有用程序的工具链。

顺便提一下，如果只是建立一个用于编译Linux内核或其他底层嵌入式应用程序的工具链，那就不需要用到GLIBC，因此整个建立过程会变得简单一些，但付出的代价是建立的工具链只能用于编译定制内核或为特定的嵌入式板编写一些特定的底层固件。如果这就是你所需要的，那么事情将变得容易一些。

幸运的是，近年来，工具链的建立过程已变得更加容易，这要感谢一些自动化脚本的出现，它们旨在帮助你为特定应用程序自动建立一个定制的工具链。迄今为止，最受欢迎的脚本是Dan Kegel的crosstool。许多爱好者和厂商都以它为基础来建立自己的定制工具链，并且它可以免费从网上获得。如果你也使用它，将省去很多麻烦。

你可以通过Kegel的网站<http://kegel.com/crosstool>获得crosstool。它是一个压缩文件，你必须首先

将它解包，解包产生的目录中还包括为每一个GNU工具链支持的CPU类型编写的演示脚本。你可以通过修改这些演示脚本来为你所需要的工具链创建一个合适的配置脚本。例如，你可以在基于Intel IA32(x86)的工作站上创建一个可以为PowerPC目标编译应用程序的交叉工具链。或者，你也可以用crosstool为与你的Linux工作站的结构相同的机器建立普通的工具链。

crosstool将从各自相应的项目网站下载单个工具链源代码的过程变得自动化，并自动为期望的目标组合打上必要的补丁。然后crosstool将按照正确的顺序编译工具链的各个部分。在结束时，你还可以选择运行标准的回归测试（它作为GNU工具链源代码的一部分发行）。

2.9 本章总结

全世界成千上万的用户、开发人员和厂商都非常依赖于GNU工具链中的每个工具。他们每天都在使用GNU工具为各种操作系统编译许多不同类型的应用程序。在Linux上，GCC及其相关的工具几乎被用于建立从Linux内核自身到最精致的图形化桌面环境的所有软件。

在本章中，我们介绍了GNU工具链中的每个工具——GCC、binutils、gdb等。我们还介绍了这些工具的一些高级的功能，以及它们在大型Linux软件开发项目中的实际应用。我们还特别关注了一些不常见的项目（如Linux内核）的需求，它依赖于GNU工具链所拥有的一些独特的特点。

使用GNU工具完成任务的一个关键优势是它的灵活性，但除了那些最高级的开发人员以外，对于大多数人来说，它们的功能总是显得非常神秘。当然，我们也可以通过图形化开发工具来完成本章所介绍的许多操作，但对这些工具有一个整体的了解还是很有必要的，因为有时候图形化工具可能会让你失望或它不能解决你手头上的任务。

第3章

可移植性

为 现代Linux系统进行软件开发时，软件和硬件的可移植性是需要考虑的一个非常重要的问题。但可移植性究竟意味着什么？你如何才能实现软件预期的灵活性，从而使得拥有各种不同硬件和软件平台的用户都可以在对软件源代码不做修改或只做极少修改的情况下使用你的软件呢？这些问题常常困扰着开发人员。本章的目的就是为了解决在使用Linux时普遍面临的各种硬件和软件可移植性问题。

本章大致被分为两部分。前半部分介绍纯软件的可移植性。在这种情况下，软件可移植性指的是对软件可以运行在来自不同厂商的各种Linux发行版之上或甚至来自同一厂商的不同版本之上的一种需求。最理想的情况是，所有软件都可以被自动安装，而不需要用户干涉，尤其是当该软件属于一个大型项目或是Linux发行版的一部分时更需要如此。通过使用本章介绍的工具和技术，你将能够使得你所编写的软件更具可移植性，可以运行在如今的各种Linux发行版之上。

本章的后半部分着重介绍为不同的硬件平台编写软件。现代硬件的机器结构相当复杂，虽然本章不会尝试阐述现代计算机的组织结构，但本章会试图解决几个根本性的问题，如果你打算为运行Linux的不同硬件平台编写软件，这些问题会影响到你。你还将学习大头字节序和小头字节序计算机、编写同时适用于32位和64位机器的代码，并简要触及将Linux内核作为Linux系统中一种可移植硬件抽象层形式的需要。

在阅读完本章之后，你将会对为最终用户编写软件更具信心。虽然你并不会在本章中学习到所有你需要知道的内容，但当你在开发自己软件的过程中遇到这些问题时，你将会为考虑和学习许多其他实际的可移植性问题做好准备。请记住这样一条准则：软件编写没有什么硬性限制。但你应该记住：要考虑你的决定会如何影响其后的软件可移植性。

3.1 可移植性的需要

可移植性并不是一个新概念。从最初的电脑开始，人们就希望编写的软件能够尽可能地被各种计算机所使用，而不需要在这一过程中对软件进行大的修改。要定义何谓可移植性，我们需要考虑开发一个复杂的现代软件所要花费的平均时间。如果该软件原本是为某一特定的软件或硬件环境所编写的，那么我们可能需要在不进行大改动的情况下将它移植到其他类似的环境中。

软件具备可移植性的更正式的解释是：如果将一个软件移植并使其适应新环境所需要

付出的努力小于从头开始重建同一个软件所需要付出的努力，那我们就说这个软件具备可移植性。

可移植软件可以追溯到数字计算机诞生的那一天。早期的计算机由插板和可以手工编程的开关构成。一个“程序”是由手工安装到专用机器中经过特别配置的电线组成的。在那个年代（在第一个汇编语言程序出现之前），还没有出现我们如今已很熟悉的许多概念，如型号、微处理器系列等。

随着时间的推移，硬件变得越来越复杂，而用户希望在购买机器上拥有足够的灵活性，无论他购买的是什么类型的机器，它都可以运行已有的软件。所以在1964年，IBM公司公布了System/360机器。它是第一款系列兼容计算机。这些大型机器虽然彼此结构不同，但它们却可以执行同一段代码、运行相同的程序。这是第一次实现一台机器上的软件可以在不做修改的情况下运行在同一系列的另一台机器上。

随着电脑变得更加强大和复杂，编程语言从本质上也变得更加抽象。曾经用汇编语言（或手工机器代码）编写过的程序改为使用更高级的语言如C、Pascal或FORTRAN来编写。这些编程语言不仅易于被开发者学习和使用，而且它们还允许将开发者编写的代码从一台机器移植到另一台机器。在机器之间移植一个应用程序仅需要修改和机器硬件相关的代码并对它进行重新编译。这虽然很费时，但却朝正确方向迈出的一步。

高级编程语言的不断发展使得诸如UNIX这样的复杂操作系统的开发成为可能。UNIX之所以后来被看作是一个可移植的操作系统，就是因为它使用C编程语言（该语言也是由UNIX创始人发明的）进行了重写。因为我们可以为许多不同的计算机编写C编译器，所以对UNIX操作系统的移植是完全可能的。整个过程虽然很繁琐和复杂，但它比从头开始重写整个系统要容易得多，而且它还在很大程度上避免了对单个应用程序的改写，这在以前是决无可能的。

UNIX是一个强大的操作系统，它为复杂的用户应用程序提供了坚实的软件平台。它引入了一些目前仍然被广泛应用的基本概念和抽象，并极大地改进了编写可移植用户软件的过程。通过对机器中的物理硬件进行抽象，一个程序员可以编写针对UNIX系统的代码，而不是针对来自某一特定厂商的特定机器的代码。当然，其结果是不同的厂商都试图在他们的UNIX供给上超越其他厂商，这导致了“UNIX战争”。

在20世纪80年代后期和20世纪90年代，UNIX市场逐渐（有时也较为动荡）平静下来，面对来自非UNIX厂商日益激烈的竞争，不同的UNIX厂商开始携手合作以提供一个共同的UNIX标准。通过标准化UNIX，软件开发人员可以更容易地编写可移植软件，使得软件可以运行在许多不同的UNIX系统之上。这一标准化工作直接导致了可移植操作系统接口（POSIX）标准的产生。POSIX其后又被（可自由获取的）单一UNIX规范（SUS）所补充。

软件可移植性在过去十年里又有了飞跃的发展。在20世纪90年代，Sun公司的绿色计划（后来导致了Java的产生）寻求创建一个完整的虚拟机，用于帮助解决当时在各种不同的嵌入式软件平台上存在的问题。通过使用Java虚拟机，软件只需一次编译就可以在移植了虚拟机的任意平台之上运行。Java虚拟机（JVM）和Java编程语言现已成为新一代程序员熟知的概念。

随着时间的推移，可移植性的概念已发生了改变。以前，只要软件可以在来自同一厂商的各种机器上使用，我们就称该软件是可移植的，但现在我们期望Linux（和UNIX）软件可以在各种不同的机

器、发行版、配置等情况下都可以运行。如今，一个为基本的32位Intel架构的计算机所编写的软件可以在不做修改的情况下在Itanium、PowerPC或SPARC机器上重新编译被认为是理所当然的。本章的其余部分将向你介绍使这一切在Linux上成为现实的概念。

3.2 Linux的可移植性

可移植性对不同的用户和Linux软件开发人员意味着不同的含义。有些人更关心平台之间的可移植性，希望他们的软件可以运行在Linux、其他类型的UNIX和诸如Sun Solaris或Apple Mac OS X等类UNIX系统上。其他人则更关心Linux发行版之间的可移植性，希望他们的软件可以运行在众多的Linux发行版之上。还有一些人对两种情况都关心，而且希望他们的软件可以自由运行在来自不同厂商发布的各种不同类型硬件平台的32位和64位系统上。

通过使用可移植函数库并充分利用现有的各种自动配置工具编写的经过精心设计的Linux软件将可以满足上述每一项需求。即使你最初的设计目标并不是希望你的软件拥有广泛的可移植性，但想远一点总不是一件坏事。一个不可避免的事实是软件有时会超越其最初的设计目的和生命周期。你可能希望能减轻那些今后需要维护你的软件的用户的负担。

3.2.1 抽象层

任何Linux系统的内核都是Linux内核。它的标准版本本身已支持超过24种不同的硬件平台架构。当你意识到一个普通的PC工作站只是i386架构的一个变体的时候，你已开始看到它所涉及的规模。Linux除了其令人惊异的高度硬件可移植性以外，它还提供了众多的设备驱动程序以支持各种连接到系统上的周边设备。从某种程度来说，Linux内核的主要目的就是为了提供在这些不同的机器硬件之上的一个可移植软件抽象。

当为Linux编写软件时，将使用到依赖于Linux内核中特殊特征的标准化系统函数库。函数库负责提供常用的软件例程，而内核则负责处理繁琐的硬件细节。这样一来，你就不需要在编写普通应用程序时留意机器的底层硬件了。当然，如果你正在建立定制设备或需要挖掘机器的所有潜能，就需要在应用程序中突破这一抽象以实现一些更大的目标。请看图3-1以了解一个应用程序是如何与一个运行Linux的典型系统结合的。

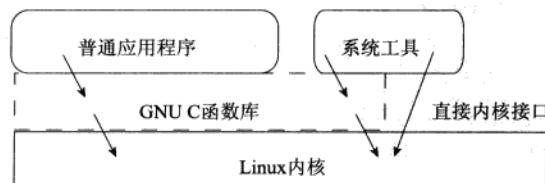


图3-1 应用程序与运行Linux的典型系统相结合的图示

人生是一连串的取舍，Linux开发同样如此。与特定的硬件和软件环境相比，抽象程度越高，性能将会越低。不过，现代计算机都很便宜，一般用户所拥有的计算机性能都远远超出他们的最低需求。但是，如果你是为资源受限系统进行开发，你可能就需要稍微改变一下这些规则。例如，在你所创建

的一个嵌入式设备上，你（当然）会假定它是一个特定的硬件平台并在保持一定程度的设计灵活性范围内使用一切可能的技巧来创建产品^①。

你可能并不需要担心机器中的硬件，但你却必须关注在Linux内核之上提供给你的软件环境。正是在这里，不同的Linux发行版（甚至同一发行版的不同版本）以及单个系统软件和函数库的版本将给你带来很多麻烦。这里提到的第二点非常重要——即函数库的版本会引发问题，我们将在本章后面介绍更多有关GNU自动化工具（Autotools）的内容时讨论它。

3.2.2 Linux 发行版

虽然Linux本身就具备高度的可移植性，但对大多数用户和开发者来说，他们更关心的是发行版的可移植性，而不是软件将运行在哪一个硬件平台上。因为他们毕竟使用普通Linux PC工作站的可能性要远远大于使用一个专用的集群设备（如果你确实使用的是集群设备，那么你可能会将该集群设备看作一个特定的平台需求，而不那么关心发行版的可移植性了，这一观点即使在今天也是正确的）。如今存在许多Linux发行版，它们互相之间都存在一些细微的差别。

我们已在本书的第1章中介绍了Linux的发行版。你已了解到目前市面上存在各种不同的Linux发行版。它们被世界各地的用户所使用，其中既有商业的发行版（如Red Hat和SuSE），也有非商业的发行版（如Gentoo和派生自Debian的Ubuntu和Knoppix）。虽然它们之间有许多不同之处，但这些发行版从根本上来说，仅仅是由发行商提供（或支持）的软件包的集合。

1. 软件包

对大多数发行版而言，一个软件包就是一个基本的原子单位。它通常是一个单独的软件或在一起提供的一组（少量的）相关工具。软件包通常基于“上游”的Linux社区项目并和它分享类似的名称。例如，大多数现代Linux发行版都提供一个名为“moduleinit-tools”的软件包（取代旧的“modutils”软件包），它包含的就是由上游的同名的Linux内核项目所提供的工具集。

每个不同的应用程序和工具（或对明显相关工具的一个简单集合）都很可能会以软件包的形式提供给用户，在软件包中还会包含文档之类的描述性数据和最重要的依赖信息。通过发行商提供的汇总已知软件包状态的数据库和软件包中的依赖信息，Linux发行版的软件包管理软件可以自动解决软件包之间的依赖关系。

依赖关系的解决方案需要Linux发行版的软件包管理程序（一个系统提供的工具，用于安装、删除和管理软件包）确定在安装一个特定的软件包之前必须先安装哪些其他的软件包。我们通常将复杂的软件包依赖关系层次构建为一个逻辑树，安装一个软件包可能需要首先安装十个或更多的其他软件包，对于一个新安装的只使用了最小配置的系统来说尤其如此。

图3-2显示了一个依赖关系树的例子：

① 一个常见的用于嵌入式设备的技巧是基于标准的平面32位内存映射（在这种内存映射方式下，硬件仅仅是内存中的另一个地址），它通过/dev/mem设备从用户空间mmap()映射外设内存。然后普通应用程序代码（尽管还需要拥有root特权）就可以在不使用内核驱动程序的情况下存取硬件。虽然这样做并不优雅，但却很有效，所以它被广泛使用。

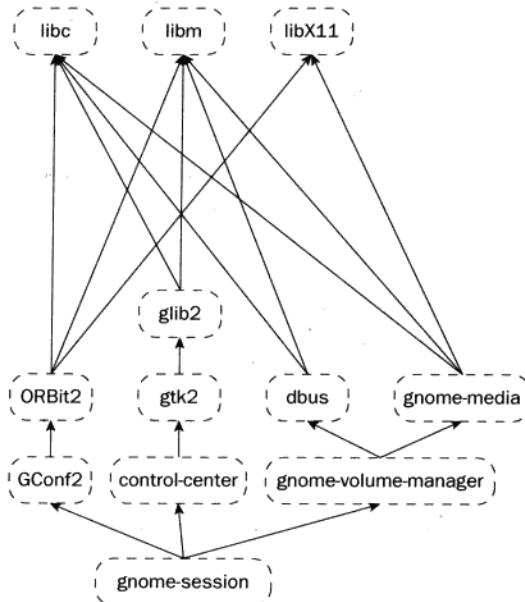


图3-2 依赖关系树

现代包管理软件能够自动处理必需的依赖软件包的获取和安装，而不需要用户手动做这些事情（与将一堆杂乱的文件整合到一起相比，这是一个根本性的优势）。但是，这一过程也并非总是如你期望的那样清晰。不同的发行版在其软件包的命名约定上都有细微的不同，在依赖关系和软件包冲突的处理方式上也有微妙的差异。所以，在不同的发行版上对你的软件进行实际的测试仍是无法替代的。

不同的Linux发行版所携带的各种由发行商所提供的软件包依赖于不同的系统函数库版本，从而导致软件的配置也各有不同。事实上，即便是同一个发行版的不同版本之间也存在着差异。正是由于发行版之间的这些差异，如果你认为Linux对第三方软件开发者来说是一个梦魇也是可以理解的。而且这有时候可能是事实，在其他操作系统平台上的开发者深知选择Linux所会受到的限制。简言之，生活中没有什么是完美的，所以当发行版问题出现时，你不得不去处理它。

当你在本章后面使用RPM和Debian的软件包管理程序建立示例软件包时，你将学习到更多有关软件包的内容。这两个最受欢迎的软件包管理系统覆盖了目前市场上绝大多数的Linux发行版。如果你能够让软件很好地与这两个系统配合，那么你就可以应付今后出现的任何流行的发行版。

2. Linux标准化规范

虽然目前还没有真正的标准Linux系统存在，但已有一些事实上的标准（以及数个正式标准）用来确定普通文件系统的位置、函数库的有效性、配置文件和应用程序以及其他一些事项，其中就包括流行的Linux标准化规范（LSB）。LSB依据的是POSIX和单一UNIX规范，但它也增加了对计算机图形、桌面和许多其他领域的考虑。LSB主张基于已有的标准而不是重新开始设计。你可以通过www.freestandards.org获得最新版本的LSB（及其示例）。

Linux标准化规范的目标是在符合LSB特定版本要求的指定主机架构上，在不同的发行版和同一发行版的不同版本之间提供真正的二进制兼容。LSB并不是专用于Linux的，从理论上来说，任何现代操作系统都可以实现某一特定程度的LSB（虽然在实践中没有这样做的）。因为LSB是一个二进制ABI（应用程序二进制接口）标准，所以LSB兼容的软件包应该（理论上）可以被广泛地移植到许多其他系统上，而且一切都应该可以正常工作，但现实情况可能有点不同。

LSB使用RPM软件包作为其可移植软件包格式概念的基础，并对LSB兼容的软件包的命名和内部内容提供了各种约束。本章的目的是使你编写的软件大致符合标准（我们将在稍后介绍如何测试应用程序是否兼容LSB），但是，如果你想对LSB兼容的软件包装能做什么和不能做什么有一个完整的了解，就需要查看专门针对LSB当前标准的相关图书，你可以在LSB的网站上找到它们。

● LSB不是灵丹妙药

唉，人生永远不会像你期望的那样简单，在使用LSB达到普遍可移植性的路途上存在着很多的障碍。不仅如此，LSB还是一个不断发展的规范，并得到现有Linux发行版不同程度的支持。每个厂商都支持某一特定级别的LSB，并可能已被认证为在发行版的某一特定配置情况下满足各种LSB需求。由此可见，LSB并不是一个可以解决所有跨发行版问题并让你作为一个开发者的生活永远免于痛苦的灵丹妙药。从长远来看，LSB真正提供的是使可移植性变得简单的机会。

事实上，不同程度的标准的存在并不意味着当一些琐碎的差别出现时你不必处理它们。为了使假设尽可能的简单，应该避免使用你并没有提供并且在其他Linux系统上也不可用的发行版特定的系统功能。如果你一定要支持某一特定的Linux发行版，那么通常可以通过由厂商提供的版本文件来检测系统中安装的Linux版本并据此采取相应的行动。

我们将在本章后面使用RPM软件包管理工具建立一个示例软件包时介绍更多有关LSB的内容。在建立该示例软件包的过程中，我们将讨论用于遵循LSB的某些标准并对这些标准进行解释。不过，本书的目的并不是要取代在LSB网站上提供的最新的文档和例子。

3. Linux厂商扮演的角色

Linux厂商参与的是一个艰苦的游戏，它不断跟踪上游社区软件的版本并在决定发行版中包含什么软件包时必须考虑到发展趋势和利益的变化。这并不是一个容易参与的游戏，也不可能面面俱到。这就是为什么有这么多不同发行版的原因之一，正如音频爱好者需要系统支持一定程度的实时音频处理能力，而许多普通的桌面用户则完全不关心这方面的功能一样。

厂商的工作是一个不断妥协的过程。它需要根据大多数用户的需求预先包装一套软件并交付给用户。有着特殊需求的用户可能会在系统安装时找到他们需要的可选软件包，也可能找不到。如果是后一种情况，他们就需要到别处去寻找或得到社区的支持。我们继续讲上面的音频的例子，许多高端Linux音频用户现在使用被称为JACK的音频系统。它提供的系统函数库对高端音频开发人员非常有用，但它却并不会被许多Linux发行版所安装。

当编写自己的应用程序时，需要在设计中考虑将依赖于哪些外部软件和函数库、哪些软件将由自己交付（如有必要）、如何支持不满足应用程序需求的发行版。有一些限制是绝对的，例如，如果应用程序需要最新的Linux实时（RT）支持，那么用户就必须有一个已包含支持该特征的内核的发行版。因为代表用户来升级这样一个基本的系统软件通常是不合理的。

一些开发人员和厂商所面对的典型问题包括：

- 软件包和函数库的有效性。

- 预先安装软件的版本。
- 发行版专用的配置。

对于一个软件开发人员来说，其中的每一条都会引发一个不同的问题。虽然有各种方法来处理这些问题，但我们通常采用的是支持最小公分母的方法。如果只依赖于在大多数Linux发行版中已存在一段时间的公认特征，那么遇到的问题可能比依赖于一些较新的特征要少得多，这些较新的特征在开始被发行版厂商采纳时使用的是不同的方式，而且表现方式也各有不同。

● 关于质量

虽然任何人都可以发布一个定制的Linux发行版，但发行版质量仍然是我们需要考虑的一个重要方面。主要的Linux发行商和社区项目都为他们所提供的Linux系统的稳定性付出了相当多的努力，并对Linux系统做了大量的测试以确保系统在各方面的性能。许多其他的项目也可能品质较高，但它们却不一定能保证做了同等程度的测试，从根本上来说这是一个资源问题。作为一位开发人员，支持哪一种商业和社区维护的Linux发行版以及选择哪一个具体的版本是由你自己来决定的。

不管你决定支持哪一种Linux发行版，都不要歧视诸如Debian这样的非商业Linux发行版——仅仅因为它们没有主要Linux发行商的支持。有好几个非常受欢迎的Linux发行版都几乎完全由社区来支持，它们特别受爱好者和家庭用户的欢迎。此外，还有大量商业用户也在使用它们。你可以根据你的软件的目标受众和你的资源情况来做出一个合理的决定。编写良好的软件几乎不需要进行重建就可以移植到其他的Linux环境中。

4. 基于RPM的发行版

几种流行的Linux发行版中的软件包都是以RPM软件包格式存储的。RPM最初代表的是Red Hat软件包管理程序（Red Hat Package Manager），但如今它已被许多其他的发行版所使用，因此它的全称又被改名为RPM软件包管理程序（RPM Package Manager，它创造了一种递归的缩写——这是自由软件开发者所喜欢的一种消遣方式）。RPM是LSB使用的一种标准的软件包格式，它被Linux开发人员广泛采用，一些并没有用它作为正式系统软件包管理程序的发行版（如Debian、Ubuntu）也在一定程度上支持它。

RPM并不一定是最好的软件包格式，但它却是历史最悠久和最受欢迎的软件包格式之一。因为其普遍的接受性，你肯定想使用RPM软件包格式来包装自己的应用程序（除非你使用的是Debian GNU/Linux和Ubuntu这样的系统）。正如你后面将学习到的，在其他类型的Linux发行版中存在一些强大的工具如alien，它们允许这些系统在不同程度上支持RPM。

很少有用户会直接在命令行上调用RPM。大多数人会选择使用某些类型的图形化安装工具，只需点击一下软件包，其他工作就可以交给软件包管理程序来完成了。为此，存在各种在RPM之上的高级工具。在Fedora系统上最广泛使用的工具之一是yum，它类似于Debian上的apt，yum本身并不是一个软件包装系统，而是一个自动化的前端，它有能力解决在安装过程中可能出现的RPM依赖问题。其他基于RPM的发行版者有它们自己的软件包管理程序——请参考它们的文档。

5. 派生自Debian的发行版

基于Debian的发行版（包括Ubuntu和它的所有变体）本身并不使用RPM。Debian也从未打算转向RPM，这主要是因为它们喜欢使用自己的软件包管理系统dpkg中的一些高级的功能。派生自Debian的发行版依赖于dpkg、更高层的软件包解析工具apt和synaptic等图形化软件包管理程序。在本章中包括

一段对Debian软件包建立过程的简短描述，但是，如果你想了解更多这方面的内容（包括详细的HOWTO指南），请访问www.debian.org。

值得注意的是，大多数Debian项目的文档质量都非常的高。虽然RPM可能是一个流行的事实标准，但Debian dpkg的使用也非常简单。

6. 源代码发行版

源代码级别的发行版并不倾向于使用非常广泛的软件包管理系统，但也有例外。Gentoo就是这样一个例子，它是一个强大的基于源代码的发行版，它在直接操作源代码这一基本概念之上加上了软件包的概念。老的发行版（如Slackware）更喜欢直接操作二进制tarball文件和其他档案文件。

本章并不会直接讨论源代码级别的发行版，因为当你直接和未包装的软件打交道时，你一般不会太关注它的可移植性问题。如果你想了解更多有关如何在一个纯基于源代码的发行版中建立软件包的信息，你可以访问GNU项目的网站www.gnu.org，上面有一些正在为此工作的项目小组的示例。

7. 自己动手

有时你可能有必要创建自己的Linux发行版。我们一般不鼓励这样做，因为用户不可能仅仅为了获取一个软件而去安装一个Linux发行版。话虽如此，有些软件厂商确实提供专用的独立系统，它携带定制的Linux发行版，并预先整合了所有必要的软件。如果你决定了要采取这一特别的步骤，请务必让你的Linux环境基于一个流行的发行版。如果有必要，有些发行版还提供明确的指令用于创建你自己的派生发行版。

创建自己的Linux环境的一个更为常见的原因发生在专用Linux设备上。资源严重受限的嵌入式系统通常都需要使用常用Linux软件、工具和函数库的特定精简版本。有些嵌入式Linux厂商销售的产品就旨在解决这一问题，但仍有不少人喜欢在创建他们自己的环境所获得的灵活性（当然也要付出更多的努力）和对这一环境中的一切事物所拥有的精确控制。

不仅仅是嵌入式设备使用它们自己的Linux发行版，大型金融机构为了便于审计或大规模部署（在这种情况下，存储空间理所当然将受到特别的关注）偶尔也会创建自己的精简版Linux。有时人们也会为了提高性能或科学计算而对发行版进行定制。在这种情况下，优化将成为一个十分现实的问题。这些定制的Linux环境大部分都基于流行的发行版。

如果决定创建自己的Linux发行版，那么请访问www.linuxfromscratch.org。你将在该网站上看到一个完整的指南，它将告诉你哪些是文件系统中必需的系统函数库和工具，并告诉你如何将它们整合到一起。这的确是一个很有教育意义（如果不是非常耗时）的过程，但在大多数情况下我们并不鼓励这样做。

不要因为我们上面说的这些话你就感到气馁而放弃追求你的好奇心——只是你要记住，
大多数用户宁愿使用一个已经过多年测试并被广泛采用的发行版，而不会愿意安装一个诸如
“Larry的不可侵犯的Linux 9000”这样的系统。

3.2.3 建立软件包

大多数发行版的软件包都是通过上游源代码（社区软件的源代码发布）自动建立（使用这些源代码包中所包含的详细编译说明）的。厂商为软件提供自己的补丁是十分常见的，这些补丁将修改软件包中所包含软件的功能（在编译时），以使它们符合特定发行版的标准，系统配置文件的位置和默认

参数都是在这时考虑的，补丁也可以仅仅是简单的修复已知的错误并提供稳定性。如果你不是一个大型社区项目工作，而是单独工作或与第三方公司合作，那么你不太可能需要应用很多定制的补丁。

当包装自己的软件时，你很可能会使用我们上面讨论的两个主要软件包格式（RPM或Debian package）之一，但你并不仅限于使用这两种包装机制。你有可能认为自己的软件已充分自包含了，因此交给用户的软件包中只需包含一个简单的tarball文件，但你却不得不自己处理软件的更新和冲突。在目前流行的Linux软件中，已有几个采取这种做法的著名示例。Tarball发行版在与图形化安装程序捆绑在一起时尤其受欢迎——与严格的跟踪软件包相比，也许这并不是一个好想法，但它确实相当受欢迎。

几个设计用来为多个平台提供软件的“通用安装器”工具选择将Linux软件包装为一个单独的可执行程序，它同时包含安装程序和软件本身的tarball文件，然后由安装程序来处理对已安装版本的检测，它通常会忽略系统的软件包数据库。这并不是我们积极鼓励去做的一件事情，因为它破坏了系统的软件包管理机制。

1. 建立RPM软件包

RPM是使用最广泛的Linux软件包格式之一，你极有可能在Linux工作站上遇到（或已经遇到）RPM（即使你使用的并不是Red Hat或SuSE的派生系统）。事实上，RPM软件包在旧的tarball文件之后得到了普遍的使用，你可以使用它来把自己的源代码、二进制文件、函数库、文档和其他应用程序文件包装到一个单独的逻辑单元中，并将它分发给用户和其他开发人员。

● 编写Spec文件

所有的RPM软件包都是根据在spec（规格）文件中包含的一系列指令建立的。spec文件描述了RPM的建立过程：首先获取程序的源代码，然后根据发行版版本的情况给软件打上相应的补丁，针对特定的编译环境进行编译，最后生成一个单独的软件包文件。该软件包能够在一个（可能兼容LSB的）目标系统上被安装和删除。spec文件中的一些段落如下所示：

- **通用头信息：**包括软件包的名称、用途摘要、版本、许可证和源代码最初来源。软件包的名称是非常重要的，因为LSB对你可以在一般发行版中使用的有效名称做了限制。LSB兼容的非厂商提供的软件包必须以lsb-为前缀，其后跟随一个由LANANA^①分配的名字。如果你选择遵循LSB标准，就需要牢记这一点。你将在LSB规范本身中找到更多这方面的内容。
- **Prep：**以%prep标注的段落指明为了准备要编译的源代码所必须采取的行动。RPM已知道应该如何解开源代码包并按顺序应用补丁（源代码和补丁的编号从零开始，如果只有一个文件，则不加编号），但在这里你可以管理和接管这一过程。
- **Build：**以%build标注的段落告诉RPM应该如何编译源代码。这可能只是简单地调用GNU make（可能还会调用GNU configure和其他GNU的自动化工具），也可能比较复杂，尤其当项目的规模大到和官方的Linux内核一样的程度时，我们可能必须在编译之前正确执行各种额外的步骤。

① Linux名称与编号分配机构（Linux Assigned Names And Numbers Authority）。其名字的构造大致仿造如IANA（the Internet Assigned Number Authority，因特网编号分配机构）这样的机构。它作为Linux社区的公共服务由kernel.org中的工作人员负责。LANANA不仅维护分配的软件包LSB名称，而且还维护Linux内核和其他Linux社区项目所使用的主要编号列表。

- **Clean:** 以%clean标注的段落告诉RPM在完成它的编译后应该如何执行清理工作。这通常只是通过简单的调用rm命令来删除临时编译目录，如有必要，你也可以实现更复杂的处理。
- **Post:** 以%post标注的段落包含一系列的命令，它们将在软件包安装之后被执行。这包括启动新添加系统服务的命令、执行安装后更新的命令或只是提醒用户软件需要额外的配置。例如一款著名的虚拟化产品会在这一时刻编译它自己的内核模块。
- **Files:** 以%files标注的段落告诉RPM哪些目录和文件将被安装。你没有必要列出每一个单独的文件，如果应用程序将文件安装到标准的系统位置，就可以使用如%{_bindir}/*这样的宏来指向“我的应用程序中，在目录bin下的所有文件”。不要忘记设置%defattr，它设置文件的默认属性和拥有者，这样RPM就知道在安装文件时应该给它们分配什么权限。
- **Changelog:** 以%changelog标注的段落含义相当明显。它包含一个自由格式的文本区域以对这个RPM所做的修改进行简要介绍。有的开发者倾向于在这里列出对应用程序做出的所有改动，而有的开发者则仅仅在这里列出针对特定Linux发行版对应用程序的RPM软件包版本所做出的改动。
- **Extras:** 你还可以在自己的RPM spec文件中包含许多其他的可选段落（请查看相关文档以获得更详细的介绍）。例如，当在系统中卸载软件包时，你可能需要实现一些特殊的处理，如完全卸载需要先停止相关的系统服务。

查看一个实际的RPM示例后，可能就一切都清楚了。下面是一个简单的应用程序（Hello World——一个经典的、依然最好的示例程序）的典型RPM spec文件内容：

```
Name: hello
Summary: A simple hello world program
Version: 1.0
Release: 1
License: GPL
Group: Applications/Productivity
URL: http://p2p.wrox.com/
Source0: %{name}- %{version}.tar.gz
Patch0: hello-1.0-hack.patch
Buildroot: %{_tmpdirpath}/%{name}-buildroot

%description
This is a simple hello world program which prints
the words "Hello, World!" except we've modified it
to print the words "Goodbye, World!" via a patch.

%prep
%setup -q
%patch0 -p1

%build
make

%install
rm -rf %{buildroot}

%makeinstall
```

```
%clean
rm -rf %{buildroot}

%post
echo "The files are now installed, here we do anything else we need to."
echo "You should never actually print output in normal use since users"
echo "with regular graphical package management software won't see it."

%preun
if [ "$1" = 0 ];
then
    echo "The package is being removed, here we cleanup, stop services before"
    echo "we pull the files from underneath, etc."
fi

%postun
if [ "$1" -ge "1" ];
then
    echo "The package is still on the system, so restart dependent services"
fi

%files
%defattr(-,root,root)
%{_bindir}/*

%changelog
* Mon Feb 21 2006 Jon Masters <jcm@jonmasters.org>
- initial release of hello with added goodbye
```

示例spec文件包含了我们感兴趣的一些段落。请注意它是如何被分成不同的逻辑段落以描述一个典型建立过程的不同阶段的。首先是通用头(general header)，它描述了软件包及其许可证和它的源代码来源。接着是对软件包的一段描述(用户可通过它来决定是否安装这个软件包)，最后是一系列的指令，它们用于解开源代码并针对特定的目标架构将源代码编译为可执行二进制代码。

- 建立一个示例RPM

本章所带的源代码中包括一个RPM示例。你将在第3章的子目录rpm中找到如下的文件：

- hello-1.0-1.spec
- hello-1.0-hack.patch
- hello-1.0.tar.gz

这三个文件就是建立一个可安装的RPM软件包所需要的全部文件。RPM软件包的内容和建立指令由前面描述的hello-1.0-1.spec文件指定。这个spec文件告诉RPM构建过程，它必须首先解开hello-1.0.tar.gz文件中的源代码，然后打上一个针对这个RPM软件包的补丁hello-1.0-hack.patch。可以使用这个补丁来添加任何必要的修改，以针对它将使用的特定目标软件环境包装你的软件。例如，发行版会经常使用自己的(不同的)配置文件路径为“上游”软件打上补丁。如果你遵循这些步骤，它将使你的生活更加轻松。

- 为你的源代码打上补丁

示例代码中的补丁非常简单，因为它仅仅替换了原文件hello-1.0.tar.gz中的欢迎字符串。我

们通过将源文件拷贝到一个新目录中并修改文件hello.c来产生补丁文件:

```
$ cd portability/rpm  
$ tar xvzf hello-1.0.tar.gz  
$ cp -ax hello-1.0 hello-1.0_patched
```

示例中修改的代码如下所示:

```
int main(int argc, char **argv)  
{  
  
    printf("Goodbye, World!\n");  
  
    return 0;  
}
```

最后, 你可以通过在示例源代码的顶层目录中使用GNU diff工具来生成合适的补丁文件:

```
$ diff -urN hello-1.0 hello-1.0_patched  
diff -urN hello-1.0/hello.c hello-1.0_patched/hello.c  
--- hello-1.0/hello.c 2006-02-21 01:06:31.000000000 +0000  
+++ hello-1.0_patched/hello.c 2006-03-18 21:59:46.000000000 +0000  
@@ -8,7 +8,7 @@  
 int main(int argc, char **argv)  
{  
  
-    printf("Hello, World!\n");  
+    printf("Goodbye, World!\n");  
  
    return 0;  
}
```

● 配置构建环境

RPM默认情况下会在目录/usr/src/rpm下寻找它所需要的任何文件。这是现有的Red Hat以及SuSE派生发行版所使用的标准位置, 但对于要建立自己的软件包的个人用户来说, 这个位置并不一定合适。其中一个原因是, 上述目录是一个系统级别的目录, 它通常由root用户所拥有(或有类似的权限限制), 所以你最好(通常也是这样)使用家目录或一个专用于开发的目录来构建RPM软件包。要使用家目录, 你需要对你的构建环境稍作修改。

你可以通过编辑家目录中的rpmmacros文件来配置RPM构建环境:

```
$ vi ~/.rpmmacros
```

你至少需要告诉RPM到哪里去寻找必需的源文件以及将产生的软件包放置到哪里。rpmmacros文件还允许针对整个构建过程设置许多灵活的扩展, 更多的信息请查看在线(*man*手册页和因特网)文档。

下面是一个很小的示例文件:

```
%_home %(echo $HOME)  
%_topdir %(home)/rpm
```

你还需要创建相应的目录, RPM将在构建过程中在这些目录中寻找文件。在你的终端上使用以下命令来完成这项工作:

```
$ mkdir -p $HOME/rpm/{SPECS,SOURCES,BUILD,SRPMS,RPMS}
```

mkdir命令的-p选项将自动创建指定路径中所有尚不存在的目录。

然后，将示例文件分别拷贝到你的本地RPM环境中的合适位置。

```
$ cp hello-1.0-1.spec $HOME/rpm/SPECS
$ cp hello-1.0.tar.gz $HOME/rpm/SOURCES
$ cp hello-1.0-hack.patch $HOME/rpm/SOURCES
```

● 启动构建过程

要正确地开始构建过程，你只需针对适当的spec文件调用rpmbuild命令即可。对于派生自Debian的发行版以及其他非RPM的发行版，这个命令可能稍有不同。例如，在笔者使用的一台机器上，使用的就是rpm而不是rpmbuild。下面针对的是标准的基于RPM的发行版，正确的命令调用方式及其输出如下所示：

```
$ rpmbuild -ba /home/jcm/rpm/SPECS/hello-1.0-1.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.15663
+ umask 022
+ cd /home/jcm/rpm/BUILD
+ cd /home/jcm/rpm/BUILD
+ rm -rf hello-1.0
+ /bin/gzip -dc /home/jcm/rpm/SOURCES/hello-1.0.tar.gz
+ tar -xf -
+ STATUS=0
+ '[' 0 -ne 0 ']'
+ cd hello-1.0
+ echo 'Patch #0 (hello-1.0-hack.patch):'
Patch #0 (hello-1.0-hack.patch):
+ patch -p1 -s
+ exit 0
Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.15663
+ umask 022
+ cd /home/jcm/rpm/BUILD
+ cd hello-1.0
+ make
cc     hello.c    -o hello
+ exit 0
Executing(%install): /bin/sh -e /var/tmp/rpm-tmp.86515
+ umask 022
+ cd /home/jcm/rpm/BUILD
+ cd hello-1.0
+ rm -rf /var/tmp/hello-buildroot
+ make prefix=/var/tmp/hello-buildroot/usr exec_prefix=/var/tmp/hello-buildroot/usr
bindir=/var/tmp/hello-buildroot/usr/bin sbindir=/var/tmp/hello-buildroot/usr/sbin
sysconfdir=/var/tmp/hello-buildroot/usr/etc datadir=/var/tmp/hello-
buildroot/usr/share includedir=/var/tmp/hello-buildroot/usr/include
libdir=/var/tmp/hello-buildroot/usr/lib libexecdir=/var/tmp/hello-
buildroot/usr/libexec localstatedir=/var/tmp/hello-buildroot/usr/var
sharedstatedir=/var/tmp/hello-buildroot/usr/com mandir=/var/tmp/hello-
buildroot/usr/share/man infodir=/var/tmp/hello-buildroot/usr/info install
mkdir -p /var/tmp/hello-buildroot/usr/bin
cp hello /var/tmp/hello-buildroot/usr/bin
+ /usr/lib/rpm/b�-compress
+ /usr/lib/rpm/b�-strip
```

```
+ /usr/lib/rpm/bp-strip-comment-note
Processing files: hello-1.0-1
Finding Provides: (using /usr/lib/rpm/find-provides)...
Finding Requires: (using /usr/lib/rpm/find-requires)...
PreReq: /bin/sh /bin/sh /bin/sh rpmlib(PayloadFilesHavePrefix) <= 4.0-1
rpmlib(CompressedFileNames) <= 3.0.4-1
Requires(interp): /bin/sh /bin/sh /bin/sh
Requires(rpmlib): rpmlib(PayloadFilesHavePrefix) <= 4.0-1
rpmlib(CompressedFileNames) <= 3.0.4-1
Requires(post): /bin/sh
Requires(preun): /bin/sh
Requires(postun): /bin/sh
Requires: libc.so.6 linux-gate.so.1 libc.so.6(GLIBC_2.0)
Wrote: /home/jcm/rpm/SRPMs/hello-1.0-1.src.rpm
Wrote: /home/jcm/rpm/RPMS/i386/hello-1.0-1.i386.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.17511
+ umask 022
+ cd /home/jcm/rpm/BUILD
+ cd hello-1.0
+ rm -rf /var/tmp/hello-buildroot
exit 0
```

这一过程最终将在 \$HOME/rpm/PRMS/i386 目录下产生一个正确的二进制 RPM 软件包 hello-1.0-1.i386.rpm。如果你的系统使用 RPM 作为其软件包管理程序，那么你还可以安装这个软件包（如有必要，可以将 i386 替换为你的开发机器所使用的架构——例如 ia64、s390x 或 ppc64）。

```
$ sudo rpm -ivh $HOME/rpm/RPMS/i386/hello-1.0-1.i386.rpm
$ hello
Goodbye, World!
```

请参考你的发行版的手册（和 RPM 文档）以获得更多在你的 Linux 环境中安装、删除和配置软件包的指导。

● 在非基于 RPM 的系统中安装 RPM 软件包

有些系统（特别是 Debian）不是基于 RPM 的，它未必会允许你直接安装 RPM 软件包。例如在 Debian 系统中，如果你尝试使用 RPM 来安装软件包，将产生如下的错误信息：

```
$ sudo rpm -ivh /home/jcm/rpm/RPMS/i386/hello-1.0-1.i386.rpm
rpm: To install rpm packages on Debian systems, use alien. See README.Debian.
error: cannot open Packages index using db3 - No such file or directory (2)
error: cannot open Packages database in /var/lib/rpm
```

请注意上面提到的 alien。它是 Debian 提供的一个工具，用于在不同的软件包格式之间执行自动转换。alien 可以将 RPM 软件包转换为 Debian 的软件包并执行一些其他的操作（这些操作在线手册中都有说明）。虽然 alien 并不完善（因而只能部分替代真正的 Debian 软件包），但当 Debian 和 Ubuntu 用户^①没有自身发行版提供的软件包可用时，它可以幫助这些用户摆脱困境。

可以使用 alien 命令将 hello-1.0-1.i386.rpm 软件包转换为等价的 Debian 软件包。你可以通过下面这个例子来自己测试 alien：

^① 那些使用派生自 Debian/Ubuntu 发行版的用户。

```
$ fakeroot alien /home/jcm/rpm/RPMS/i386/hello-1.0-1.i386.rpm
```

alien命令在默认情况下将假定它被要求将一个指定的RPM软件包转换为Debian软件包，并执行转换工作。alien默认还需要拥有root特权（为了创建一些在软件包构建过程中需要的文件），但你可以通过使用fakeroot包裹alien调用的方法来解决这一问题。fakeroot临时越过标准C库函数并让alien进程（及其所有子进程）认为它们拥有root特权。

注意，当使用fakeroot时并没有授予alien真正的root权限，但alien会认为它可以创建由root拥有的文件和目录。这样做就已经足够了，因为任何被创建的文件都将被立刻放入软件包的一个档案中，它们并不需要以系统的root用户身份存在于磁盘上。因此，fakeroot对于这些有限的情况（如所有需要的仅仅是一个权利的幻觉）是非常适用的。如果你需要给一个进程授予真正的root特权，那么就应该使用sudo命令来代替。你的Linux发行版应该有该命令的相关文档。

2. Debian软件包

Debian软件包由dpkg软件包管理工具制作。与RPM一样，dpkg遵循由一个单独的文本文件所指定的一系列指令。在Debian系统中，这个文件的文件名为control（对应RPM中的SPEC文件），它所包含的信息类型大部分都和你在前面的RPM示例中看到的一样。在我们所提供的示例Debian软件包中包含一个control文件，你将在本节中学习如何建立它。

创建一个单独的Debian软件包的过程是非常简单易懂的，这要感谢Debian的自动化帮助脚本。在本例中，给定一个已有的源代码项目，我们可以通过在命令行中使用dh_make来创建一个新的软件包。例如，你可以将已有的hello-1.0.tar.gz拷贝到一个新的目录中，并以它作为一个全新的Debian软件包的基础。

将hello-1.0.tar.gz拷贝到一个工作目录中，并对它进行解包：

```
$ tar xvfz hello-1.0.tar.gz
```

在新释放出的目录中，你将看到熟悉的示例源代码。你可以通过在该目录中调用dh_make脚本来创建一个Debian软件包：

```
$ dh_make -e jcm@jonmasters.org -f ./hello-1.0.tar.gz
```

虽然dh_make有许多可以使用的选项标记（请查看它的手册页以获得一个冗长的描述），但你只需要使用其中的两个选项就可以开始工作了。在上面的命令中，“-e”告诉dh_make软件包作者的电子邮件地址是jcm@jonmasters.org，而“-f”选项指定了当前源代码树的最初未修改源代码的位置。

在运行dh_make时，你会被问到你想创建哪种类型的软件包：

```
Type of package: single binary, multiple binary, library, kernel module or cdb? [s/m/l/k/b]
```

因为你是要创建一个单独的二进制软件包，所以选择选项“s”（其他类型更为复杂，详细信息请查看它们的相关文档）。下面是一个典型的dh_make调用的输出：

```
$ dh_make -e jcm@jonmasters.org -f ../hello-1.0.tar.gz
Type of package: single binary, multiple binary, library, kernel module or cdbs?
[s/m/l/k/b] s

Maintainer name : Jon Masters
Email-Address   : jcm@jonmasters.org
Date            : Mon, 01 Jan 2007 14:54:00 +0000
Package Name    : hello
Version         : 1.0
License          : blank
Type of Package : Single
Hit <enter> to confirm:
Done. Please edit the files in the debian/ subdirectory now. You should also
check that the hello Makefiles install into $DESTDIR and not in / .
```

如果一切按计划进行，你将很快在现有的项目源代码目录中获得一个新的debian目录。当你在项目源代码目录中执行ls命令时，你将看到如下的输出：

```
$ ls
Makefile debian hello hello.c
```

Debian所做的所有神奇的事情都发生在debian目录中。在该目录中，你将发现dh_make已为你创建了许多标准文件：

README.Debian	dirs	init.d.ex	preinst.ex
changelog	docs	manpage.1.ex	prerm.ex
compat	emacsen-install.ex	manpage.sgml.ex	rules
conffiles.ex	emacsen-remove.ex	manpage.xml.ex	watch.ex
control	emacsen-startup.ex	menu.ex	
copyright	hello-default.ex	postinst.ex	
cron.d.ex	hello.doc-base.EX	postrm.ex	

幸运的是，你并不需要在创建一个标准软件包时修改其中的许多文件。但你需要修改control文件以提供你的特定软件包的相关信息。由dh_make所创建的control文件只提供相当基本的内容：

```
Source: hello
Section: unknown
Priority: optional
Maintainer: Jon Masters <jcm@jonmasters.org>
Build-Depends: debhelper (>= 4.0.0)
Standards-Version: 3.6.2

Package: hello
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: <insert up to 60 chars description>
             <insert long description, indented with spaces>
```

你的第一个任务就是修改该信息以提供类似于你在先前的RPM SPEC文件中提供的元数据。请注意，你并不需要指定这个软件包的来源，因为你已位于源代码树中（debian子目录就在你自己的应用程序的源代码目录中——这是Debian的事情）。

下面是针对这个例子修改过的control文件的内容：

```

Source: hello
Section: devel
Priority: optional
Maintainer: Jon Masters <jcm@jonmasters.org>
Build-Depends: debhelper (>= 4.0.0)
Standards-Version: 3.6.2

Package: hello
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: A simple Hello World example program.
This program changes the world in many ways. Firstly, it uses the Debian
package management system to demonstrate how you can create your very own
Debian packages. But there's so much more. Call now! Operators are standing
by for /your/ call. And if you call right now, your order will be doubled!

```

Source和Package行分别指定软件包和应用程序自身的名字。Source行指的是新创建的Debian源代码软件包的名字，而不是目前正在使用的程序源代码的位置。Section的含义类似于RPM软件包中的Group。Debian项目定义了各种标准组，你需要查看它们的文档以获得正确的组名称的进一步示例（和RPM的情况一样，不要随便给组命名——这将使得它的运行情况很差，并导致用户最终向你发出抱怨）。control文件的最后一段对这个软件包的功能的冗长描述，说明了它是如何有助于改变世界的（不论好坏）。

● Debian软件包的依赖关系

到目前为止，你也许想知道与使用RPM相比，使用Debian软件包的好处是什么（除了为使用相应的工具，需要你为自己的程序源代码添加Debian文件和目录以外）？Debian软件包真正强大的功能是通过其轻量级的apt（Advanced Package Tool，高级软件包管理工具）软件包管理工具^①来呈现的。apt使得Debian可以自动解决可能存在的任何依赖关系，它通过安装必备的软件包并确保避免软件包之间的冲突来达到这一点。当然，apt做到所有这一切也离不开软件包管理者的帮助。

Debian软件包的control文件可以包含许多不同类型的依赖关系。下面这些例子来自官方文档：

- **Depends:** 以Depends:开头的行指定必须先安装的软件包，以便顺利安装给定的软件包。这些都是单纯的依赖关系，它们用于避免造成严重的系统损坏。从理智上来讲，你当然不会希望自己犯一点点的错误。
- **Suggests:** 以Suggests:开头的行描述了那些对顺利操作一个特定的软件包并不是必需的软件包，但它们的存在将在某种程度上增强软件的功能。例如，如果用户有这样的需求，你可能想有一个（但不是必需）一个图形化工具来帮助将字符串Hello World格式化为霓虹灯效果的标记。
- **Conflicts:** 以Conflicts:开头的行描述了不能与这个特定的软件包同时安装的软件包。例如，如果你要提供一个邮件服务器，你可能会判断Debian默认的exim邮件服务器（及其配置脚本）将与你自己的邮件服务器软件相冲突。
- **Provides:** Debian有时会定义虚拟软件包或不存在的软件包，它提供了一种友好的方式来指向某一特定类型的软件，可能有许多可用的软件都属于这一类型。这有助于用户寻找一个现有

^① 它有着超级牛力。不要问，这是Debian的事情。在Debian系统中执行命令“apt-get --help”，输出结果的最下面一行显示的就是“It has super cow powers”（它有着超级牛力），这可以理解为是apt的一句广告词，或是—个彩蛋。——译者注

的软件包或一个著名的软件包在最新的Debian版本中被转移到了哪里，而不需要用户理解这背后的策略。

在线文档中还提供了更多这方面的例子，在下一节的结尾将提供其细节。即使在最坏的情况下，它也是一个很好的枕边读物。

● 构建Debian软件包

Debian软件包的构建过程与RPM不尽相同。但与RPM一样的是，为了成功的构建一个Debian软件包，你需要修改程序的Makefile文件，使得软件被安装到一个临时目录层次中，而不是安装到你的文件系统的根目录下。如果你查看本例中使用的Makefile文件，你将看到所有的文件都被安装到相对于\${DESTDIR}的目录下，该变量是在构建软件包时由dpkg定义的。

下面是一个取自示例软件包的dpkg Makefile文件内容：

```
all: hello

install:
    mkdir -p ${DESTDIR}
    cp hello ${DESTDIR}
```

一旦你已成功地准备好程序源代码，你就可以开始构建自己的Debian软件包了。在顶层目录（即debian子目录的上一层目录）中运行下面的命令：

```
$ dpkg-buildpackage -rfakeroot
```

类似于rpmbuild，dpkg-buildpackage将首先清理程序源代码，然后构建应用程序并从中产生一个Debian软件包。和前面一样，请注意fakeroot的使用，它是用于欺骗包构建软件，让它认为它是以完全的root权限在运行。之所以能够这样做是因为在构建软件包时并不是真正需要向标准系统文件位置写入文件，但在自包含软件包的构建过程中，可能需要明确地创建由root用户所拥有的文件。请记住，fakeroot并没有真正授予任何root特权。

一个成功的dpkg-buildpackage运行将产生以下输出：

```
$ dpkg-buildpackage -rfakeroot
dpkg-buildpackage: source package is hello
dpkg-buildpackage: source version is 1.0-1
dpkg-buildpackage: source changed by Jon Masters <jcm@jonmasters.org>
dpkg-buildpackage: host architecture i386
  fakeroot debian/rules clean
  dh_testdir
  dh_testroot
  rm -f build-stamp configure-stamp
  /usr/bin/make clean
make[1]: Entering directory `/home/jcm/PLP/src/portability/dpkg/hello-1.0'
make[1]: *** No rule to make target `clean'. Stop.
make[1]: Leaving directory `/home/jcm/PLP/src/portability/dpkg/hello-1.0'
make: [clean] Error 2 (ignored)
  dh_clean
  dpkg-source -b hello-1.0
dpkg-source: building hello using existing hello_1.0.orig.tar.gz
dpkg-source: building hello in hello_1.0-1.diff.gz
dpkg-source: building hello in hello_1.0-1.dsc
  debian/rules build
  dh_testdir
```

```

touch configure-stamp
dh_testdir
/usr/bin/make
make[1]: Entering directory `/home/jcm/PLP/src/portability/dpkg/hello-1.0'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/jcm/PLP/src/portability/dpkg/hello-1.0'
#docbook-to-man debian/hello.sgml > hello.1
touch build-stamp
fakeroot debian/rules binary
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs
/usr/bin/make install DESTDIR=/home/jcm/PLP/src/portability/dpkg/hello-
1.0/debian/hello
make[1]: Entering directory `/home/jcm/PLP/src/portability/dpkg/hello-1.0'
mkdir -p /home/jcm/PLP/src/portability/dpkg/hello-1.0/debian/hello
cp hello /home/jcm/PLP/src/portability/dpkg/hello-1.0/debian/hello
make[1]: Leaving directory `/home/jcm/PLP/src/portability/dpkg/hello-1.0'
dh_testdir
dh_testroot
dh_installchangelogs
dh_installdocs
dh_installeamples
dh_installman
dh_link
dh_strip
dh_compress
dh_fixperms
dh_installdeb
dh_shlibdeps
dh_gencontrol
dpkg-gencontrol: warning: unknown substitution variable ${misc:Depends}
dh_md5sums
dh_builddeb
dpkg-deb: building package `hello' in `../hello_1.0-1_i386.deb',
signfile hello_1.0-1.dsc
gpg: skipped "Jon Masters <jcm@jonmasters.org>": secret key not available
gpg: [stdin]: cleartext failed: secret key not available

```

请注意，这个构建过程的输出结果将出现在软件包源代码的上一层目录中。所以，如果你在家目录中解开hello-1.0.tar.gz，你就可以直接在家目录中找到输出的软件包文件hello_1.0-1_i386.deb。除了这个二进制软件包以外，你还将看到创建了其他文件。

```
hello_1.0-1.diff.gz hello_1.0-1.dsc hello_1.0-1.dsc.asc
```

这些文件可以被其他开发者用于重新生成二进制软件包。

你会看到在前面的示例的构建过程中产生了一些警告信息。第一个发现的问题是GNU Make抱怨它不能清理源代码。这个问题很好解决，你只需确保在每个Makefile文件中增加一个clean目标，就可以保证在需要的时候你的项目可以被清理。第二个发现的问题是在构建软件包时，用户没有安装GPG（或有GPG可用）。正确分发的Debian软件包只要有可能，就应包含GPG（PGP）签名，如果你选

择允许包含签名，在软件包构建过程中它就将为你执行电子签名。

● Debian 开发者

我们将日常工作在Debian Linux发行版上的人们和负责官方Debian发行版中的软件包的人称为Debian开发者（Debian Developer），或亲切地简称为“DD”。Debian不同于许多其他的发行版，这不仅表现在其使命的声明，还表现在它的开发者社区。要想成为一个Debian开发者是比较困难的，这通常需要各种辅导措施并经历一个正式的程序。

好消息是你并不需要成为一个Debian开发者才能为Debian发行版发布软件包（同样你也不需要成为一个官方的Ubuntu开发者）。既然你不太可能是或将是一位正式认可的开发者，你将只能通过自己的网站来提供软件包。如果你制作了apt仓库（apt repository），你生成的软件包并不会自动显示在用户的apt搜索软件包列表中——需要明确地告诉apt才行^①。

● 进一步信息

Debian软件包管理工具非常复杂并在不断发展着，所以需要参考“Debian New Maintainer’s Guide”（可在Debian的网站www.debian.org上找到）以获得更多信息。Debian维护自己的一套发行版标准和规则以构建有效的软件包。再次重申，软件的对应网站是获得最新参考资料的最佳位置。

3.2.4 可移植的源代码

为Linux编写的软件不仅应该在运行于相同硬件平台上的发行版之间尽可能实现二进制兼容，还应该可以在大多数已提供正确的函数库和其他必备条件的系统中成功编译。编译过程必须小心地确定不同软件环境之间的差异。这既适用于不同Linux发行版之间的情况，也适用于Linux和其他商业UNIX环境之间的情况。

为辅助实现不同编译环境之间的源代码级的可移植性，人们创建了GNU的自动化工具（Autotools）。Autoconf、Autoheader、Libtool、Automake和许多其他的脚本和工具一起工作并共同构成GNU的编译系统。每个工具会自动运行一系列的测试以获得对用户所处硬件和软件环境的理解，然后再确定是否有可能在这一特定的软件环境中编译应用程序。

当准备好合适的源代码集之后，就可以使用你可能早已熟悉的“configure, make, make install”序列来编译软件了。下一节只是对GNU自动化工具的介绍。更多信息请查看自动化工具的相关图书，你可以在<http://sources.redhat.com/autobook>上免费获得它。

1. GNU自动化工具

也许通过一个实例的讲解你会更容易理解GNU自动化工具是如何工作的。在本例中，我们使用一个简单的程序hello.c来显示一个消息。它已被修改为使用一个库（libhello）函数print_message来执行实际的消息打印操作。此外，这个程序还试图确定编译它的机器使用的是大头字节序还是小头字节序，如果机器是大头字节序，则打印一个可选的欢迎消息（字节序将稍后在本章进行介绍——请继续阅读以发现更多有关乔纳森·斯威夫特如何影响计算机科学的内容）。

下面是hello.c示例程序的源代码：

^① 通过修改/etc/apt/sources文件可以添加你提供的仓库。

```
#include "hello.h"

int main(int argc, char **argv)
{
    print_message("Hello World!\n");

#ifdef WORDS_BIGENDIAN
    printf("This system is Big Endian.\n");
#endif

    return 0;
}
```

这个程序使用了一个名为libhello的函数库，它只包含一个消息打印函数：

```
void print_message(char *msg)
{
    printf("The message was %s\n", msg);

}
```

● 建立自动化工具项目

编译这个示例程序和函数库需要若干步骤。其中每个步骤都依赖于GNU自动化工具中的一个不同部分，我们将在后续小节中对它们一一进行解释。当你在上一章中学习如何使用GNU工具链来为自己编译代码和函数库时，你已看到过类似的示例代码。现在，你将学习如何依靠强大的GNU编译工具来为你执行这些操作——如果你愿意的话。请记住GNU自动化工具本身是非常复杂的，但我们并没有在这里涵盖它们的所有复杂性。如果你想了解更多的细节，请查看GNU自动化工具的文档或参考GNU自动化工具图书的在线拷贝。

你需要在示例目录portability中准备一个宏文件aclocal.m4，该文件将根据你的configure.in文件中的内容包含许多用于automake的定义。所幸的是，你可以运行aclocal工具来帮助你完成这一工作：

```
$ aclocal
```

使用autoconf处理configure.in文件并生成一个configure脚本：

```
$ autoconf
```

使用autoheader以准备好config.h.in文件，当此后我们在目标系统中运行configure脚本时，它将用于生成有用的头文件config.h：

```
$ autoheader
```

为libtool准备好函数库，它们将被libtool自动编译：

```
$ libtoolize
```

最后，使用automake生成Makefile.in文件，使得其后对configure工具脚本的调用能够为这个项目生成Makefile文件。

```
$ automake --add-missing
automake: configure.in: installing './install-sh'
automake: configure.in: installing './mkinstalldirs'
automake: configure.in: installing './missing'
automake: configure.in: installing './config.guess'
automake: configure.in: installing './config.sub'
automake: Makefile.am: installing './INSTALL'
automake: Makefile.am: installing './COPYING'
```

这些阶段现在看来都非常复杂和难以处理，但没有关系，我们很快就会对其中的每一个阶段进行解释。当阅读完本章后，你可能会有更多有关GNU自动化工具的疑问没有办法在本章中找到答案，但至少你已知道存在这些工具，并且当你建立自己的项目时，知道到哪里去查找进一步的资料，这已成功了一半。

你可以通过调用新创建的configure脚本来启动一个配置周期：

```
$ ./configure
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
checking for a BSD-compatible install... /usr/bin/ginstall -c
checking whether build environment is sane... yes
checking whether make sets $(MAKE)... yes
checking for working aclocal-1.4... found
checking for working autoconf... found
checking for working automake-1.4... found
checking for working autoheader... found
checking for working makeinfo... found
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for a sed that does not truncate output... /bin/sed
checking for egrep... grep -E
checking for ld used by gcc... /usr/bin/ld
checking if the linker (/usr/bin/ld) is GNU ld... yes
checking for /usr/bin/ld option to reload object files... -r
checking for BSD-compatible nm... /usr/bin/nm -B
checking whether ln -s works... yes
checking how to recognise dependent libraries... pass_all
checking how to run the C preprocessor... gcc -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
```

```
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking dlfcn.h usability... yes
checking dlfcn.h presence... yes
checking for dlfcn.h... yes
checking for g++... g++
checking whether we are using the GNU C++ compiler... yes
checking whether g++ accepts -g... yes
checking how to run the C++ preprocessor... g++ -E
checking for g77... g77
checking whether we are using the GNU Fortran 77 compiler... yes
checking whether g77 accepts -g... yes
checking the maximum length of command line arguments... 32768
checking command to parse /usr/bin/nm -B output from gcc object... ok
checking for objdir... .libs
checking for ar... ar
checking for ranlib... ranlib
checking for strip... strip
checking if gcc static flag works... yes
checking if gcc supports -fno-rtti -fno-exceptions... no
checking for gcc option to produce PIC... -fPIC
checking if gcc PIC flag -fPIC works... yes
checking if gcc supports -c -o file.o... yes
checking whether the gcc linker (/usr/bin/ld) supports shared libraries... yes
checking whether -lc should be explicitly linked in... no
checking dynamic linker characteristics... GNU/Linux ld.so
checking how to hardcode library paths into programs... immediate
checking whether stripping libraries is possible... yes
checking if libtool supports shared libraries... yes
checking whether to build shared libraries... yes
checking whether to build static libraries... yes
configure: creating libtool
 appending configuration tag "CXX" to libtool
checking for ld used by g++... /usr/bin/ld
checking if the linker (/usr/bin/ld) is GNU ld... yes
checking whether the g++ linker (/usr/bin/ld) supports shared libraries... yes
checking for g++ option to produce PIC... -fPIC
checking if g++ PIC flag -fPIC works... yes
checking if g++ supports -c -o file.o... yes
checking whether the g++ linker (/usr/bin/ld) supports shared libraries... yes
checking dynamic linker characteristics... GNU/Linux ld.so
checking how to hardcode library paths into programs... immediate
checking whether stripping libraries is possible... yes
 appending configuration tag "F77" to libtool
checking if libtool supports shared libraries... yes
checking whether to build shared libraries... yes
checking whether to build static libraries... yes
checking for g77 option to produce PIC... -fPIC
checking if g77 PIC flag -fPIC works... yes
checking if g77 supports -c -o file.o... yes
checking whether the g77 linker (/usr/bin/ld) supports shared libraries... yes
checking dynamic linker characteristics... GNU/Linux ld.so
checking how to hardcode library paths into programs... immediate
```

```

checking whether stripping libraries is possible... yes
checking for gcc... (cached) gcc
checking whether we are using the GNU C compiler... (cached) yes
checking whether gcc accepts -g... (cached) yes
checking for gcc option to accept ANSI C... (cached) none needed
checking for ANSI C header files... (cached) yes
checking whether byte ordering is bigendian... no
checking for unistd.h... (cached) yes
checking for stdlib.h... (cached) yes
checking stdio.h usability... yes
checking stdio.h presence... yes
checking for stdio.h... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating include/config.h
config.status: executing default-1 commands

```

=====
Thanks for testing this example. You could add some output here
and refer to any AC environment variables if needed.

请注意configure是如何报告它所执行的每个测试的状态的。你可以看到很多类似于checking for stdio.h usability（检查stdio.h可用性）的输出，它们表明configure正在为你执行许多测试。运行完configure之后，你会看到在项目源代码目录的include子目录中创建了一个config.h文件，其内容如下所示：

```

/* include/config.h. Generated by configure. */
/* include/config.h.in. Generated from configure.in by autoheader. */

/* Define to 1 if you have the <dlfcn.h> header file. */
#define HAVE_DLFCN_H 1

/* Define to 1 if you have the <inttypes.h> header file. */
#define HAVE_INTTYPES_H 1

/* Define to 1 if you have the <memory.h> header file. */
#define HAVE_MEMORY_H 1

/* Define to 1 if you have the <stdint.h> header file. */
#define HAVE_STDINT_H 1

/* Define to 1 if you have the <stdio.h> header file. */
#define HAVE_STDIO_H 1

/* Define to 1 if you have the <stdlib.h> header file. */
#define HAVE_STDLIB_H 1

/* Define to 1 if you have the <strings.h> header file. */
#define HAVE_STRINGS_H 1

```

```

/* Define to 1 if you have the <string.h> header file. */
#define HAVE_STRING_H 1

/* Define to 1 if you have the <sys/stat.h> header file. */
#define HAVE_SYS_STAT_H 1

/* Define to 1 if you have the <sys/types.h> header file. */
#define HAVE_SYS_TYPES_H 1

/* Define to 1 if you have the <unistd.h> header file. */
#define HAVE_UNISTD_H 1

/* Name of package */
#define PACKAGE "1.9"

/* Define to the address where bug reports for this package should be sent. */
#define PACKAGE_BUGREPORT "jcm@jonmasters.org"

/* Define to the full name of this package. */
#define PACKAGE_NAME "hello"

/* Define to the full name and version of this package. */
#define PACKAGE_STRING "hello 1.0"

/* Define to the one symbol short name of this package. */
#define PACKAGE_TARNAME "hello"

/* Define to the version of this package. */
#define PACKAGE_VERSION "1.0"

/* Define to 1 if you have the ANSI C header files. */
#define STDC_HEADERS 1

/* Version number of package */
#define VERSION ""

/* Define to 1 if your processor stores words with the most significant byte
   first (like Motorola and SPARC, unlike Intel and VAX). */
/* #undef WORDS_BIGENDIAN */

```

每个#define声明对应于配置过程中的一次成功测试。hello.c源代码可以使用这些定义，而且它确实在判断是否应打印一个额外的欢迎消息时这样做了。头文件hello.h也使用这些定义来准确地判断应包含哪些头文件：

```

#ifndef HELLO_H
#define HELLO_H 1

#if HAVE_CONFIG_H
# include <config.h>
#endif

#if HAVE_STDIO_H
# include <stdio.h>
#endif

```

```
#if STDC_HEADERS
# include <stdlib.h>
#endif

#if HAVE_UNISTD_H
# include <unistd.h>
#endif

#endif /* HELLO_H */
```

你可能会觉得这种做法过于迂腐，但当你需要调整自己的源代码以尽可能地适应当地编译环境时，你将使用输出的config.h文件作为一种处理方式。例如，当将源代码移植到其他类UNIX系统中时，你可能会发现缺少一些内存分配函数或它们并没有按照你所期望的方式工作。在这种情况下，你就可以使用configure脚本来判断哪些标准函数库功能丢失了，并在必要的时候提供自己的替代实现。请查看GNU Autoconf文档以获得它所支持的测试和功能的完整列表。

一旦配置好软件，你就可以使用GNU make以正常的方式来编译它了：

```
$ make
Making all in src
make[1]: Entering directory `/home/jcm/PLP/src/portability/autotools/src'
/bin/sh ../libtool --mode=compile gcc -DHAVE_CONFIG_H -I. -I..../include -g -O2 -c hello_msg.c
mkdir .libs
gcc -DHAVE_CONFIG_H -I. -I..../include -g -O2 -Wp,-MD,.deps/hello_msg.pp -c
hello_msg.c -fPIC -DPIC -o .libs/hello_msg.o
hello_msg.c: In function 'print_message':
hello_msg.c:7: warning: incompatible implicit declaration of built-in function
'printf'
gcc -DHAVE_CONFIG_H -I. -I..../include -g -O2 -Wp,-MD,.deps/hello_msg.pp -c
hello_msg.c -o hello_msg.o >/dev/null 2>&1
/bin/sh ../libtool --mode=link gcc -g -O2 -o libhello.la -rpath /usr/local/lib
hello_msg.lo
gcc -shared .libs/hello_msg.o -Wl,-soname -Wl,libhello.so.0 -o
.libs/libhello.so.0.0.0
(cd .libs && rm -f libhello.so.0 && ln -s libhello.so.0.0.0 libhello.so.0)
(cd .libs && rm -f libhello.so && ln -s libhello.so.0.0.0 libhello.so)
ar cru .libs/libhello.a hello_msg.o
ranlib .libs/libhello.a
creating libhello.la
(cd .libs && rm -f libhello.la && ln -s ../libhello.la libhello.la)
gcc -DHAVE_CONFIG_H -I. -I..../include -g -O2 -c hello.c
/bin/sh ../libtool --mode=link gcc -g -O2 -o hello hello.o libhello.la
gcc -g -O2 -o .libs/hello hello.o ./libs/libhello.so
creating hello
make[1]: Leaving directory `/home/jcm/PLP/src/portability/autotools/src'
make[1]: Entering directory `/home/jcm/PLP/src/portability/autotools'
make[1]: Nothing to be done for 'all-am'.
make[1]: Leaving directory `/home/jcm/PLP/src/portability/autotools'
```

对make install的快速调用（具备适当的root权限）将把二进制程序hello及其必需的函数库安装为/usr/local/bin/hello（默认位置，如果你不想将它安装到你的开发用机器的主文件系统中，

可以通过configure.in或configure脚本的--prefix参数进行定制,例如,你可以选择\$HOME/bin):

```
$ sudo make install
Making install in src
make[1]: Entering directory `/home/jcm/PLP/src/portability/autotools/src'
make[2]: Entering directory `/home/jcm/PLP/src/portability/autotools/src'
/bin/sh ../mkinstalldirs /usr/local/lib
/bin/sh ../libtool --mode=install /usr/bin/ginstall -c libhello.la
/usr/local/lib/libhello.la
/usr/bin/ginstall -c .libs/libhello.so.0.0.0 /usr/local/lib/libhello.so.0.0.0
(cd /usr/local/lib && { ln -s -f libhello.so.0.0.0 libhello.so.0 || { rm -f
libhello.so.0 && ln -s libhello.so.0.0.0 libhello.so.0; }; })
(cd /usr/local/lib && { ln -s -f libhello.so.0.0.0 libhello.so || { rm -f
libhello.so && ln -s libhello.so.0.0.0 libhello.so; }; })
/usr/bin/ginstall -c .libs/libhello.lai /usr/local/lib/libhello.la
/usr/bin/ginstall -c .libs/libhello.a /usr/local/lib/libhello.a
ranlib /usr/local/lib/libhello.a
chmod 644 /usr/local/lib/libhello.a
PATH="$PATH:/sbin" ldconfig -n /usr/local/lib
-----
Libraries have been installed in:
/usr/local/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `--LIBDIR'
flag during linking and do at least one of the following:
- add LIBDIR to the `LD_LIBRARY_PATH' environment variable
during execution
- add LIBDIR to the `LD_RUN_PATH' environment variable
during linking
- use the `--rpath' linker flag
- have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
```

```
/bin/sh ../mkinstalldirs /usr/local/bin
/bin/sh ../libtool --mode=install /usr/bin/ginstall -c hello
/usr/local/bin/hello
/usr/bin/ginstall -c .libs/hello /usr/local/bin/hello
make[2]: Nothing to be done for `install-data-am'.
make[2]: Leaving directory `/home/jcm/PLP/src/portability/autotools/src'
make[1]: Leaving directory `/home/jcm/PLP/src/portability/autotools/src'
make[1]: Entering directory `/home/jcm/PLP/src/portability/autotools'
make[2]: Entering directory `/home/jcm/PLP/src/portability/autotools'
make[2]: Nothing to be done for `install-exec-am'.
make[2]: Nothing to be done for `install-data-am'.
make[2]: Leaving directory `/home/jcm/PLP/src/portability/autotools'
make[1]: Leaving directory `/home/jcm/PLP/src/portability/autotools'
```

此时,你可以尝试运行这个程序:

```
$ hello
```

```
hello: error while loading shared libraries: libhello.so.0: cannot open shared
object file: No such file or directory
```

系统找不到刚刚安装的libhello.so.0，为什么？因为动态连接器需要刷新它的缓存，在该缓存中有已知的函数库信息。连接器知道列在文件/etc/ld.so.conf中的目录并将在重建其缓存时包括某些标准目录（如本例中使用的目录/usr/local/lib）。你可以通过运行ldconfig来达到重建缓存的目的：

```
$ sudo ldconfig
```

这样做之后，hello程序就可以正常运行了：

```
$ hello
The message was Hello World!
```

祝贺你完成了你的第一个GNU自动化工具项目。现在你已沿着这条道路开始学习了，在阅读完后面小节中的解释性资料之后，你就可以开始自己进行试验了。请使用GNU自动化工具所提供的文档来指导你的试验。

如果你建立的项目之后将成为GNU的一部分（即成为自由软件基金会的官方项目），你还需要确保示例项目中包含几个重要文件。每个GNU项目必须包含一个AUTHORS文件——它包含每一个参与这个项目的作者的信息、一个NEWS文件——跟踪项目的新闻、一个README文件——提供项目的基本信息、一个ChangeLog文件——跟踪每个项目代码版本的发布。

2. GNU Autoconf

GNU Autoconf用于产生configure脚本，该脚本在编译应用程序之前运行。configure脚本可以运行一系列开发者要求的测试以确定该软件是否可以在一个特定环境中编译。例如，它可以确定你的机器上安装的（GNU或非GNU）C编译器的版本，并确保某些标准C头文件已准备好。它也可能会做一些更复杂的事情——GNU autoconf有着很大的灵活性。

GNU configure可以用于建立包含一个特定系统动态信息的头文件，这些动态信息可用于源代码测试。例如，一个编译周期通常会创建一个头文件config.h，它包含许多#define声明对应已经经过测试的特征。如果GNU configure确定本地软件环境适合编译软件，那么这些定义将有助于增加软件的灵活性，因为它们允许在必要的时候进行条件代码编译。

你很快会意识到需要使用GNU configure，因为它将极大地简化了你在Linux上的开发工作。

下面是一个configure.in的示例文件，configure脚本就是通过它生成的：

```
# This is a sample configure.in that you can process with autoconf in order
# to produce a configure script.

# Require Autoconf 2.59 (version installed on author's workstation)
AC_PREREQ([2.59])

AC_INIT([hello], [1.0], [jcm@jonmasters.org])

# Check the system
AC_CANONICAL_SYSTEM

AM_CONFIG_HEADER(include/config.h)
```

```

AC_CONFIG_SRCDIR(src/hello.c)
AM_INIT_AUTOMAKE([1.9])
AM_PROG_LIBTOOL

AC_PROG_CC
AC_HEADER_STDC
AC_C_BIGENDIAN
AC_CHECK_HEADERS([unistd.h stdlib.h stdio.h])
AC_OUTPUT(Makefile src/Makefile)

# Pretty print some output
AC_MSG_RESULT([])
AC_MSG_RESULT([=====
=====
=====
])
AC_MSG_RESULT([])
AC_MSG_RESULT([Thanks for testing this example. You could add some output
here])AC_MSG_RESULT([and refer to any AC environment variables if needed.])
AC_MSG_RESULT([])
AC_MSG_RESULT([=====
=====
=====
])

```

你可以看到这个文件较短，而且只包含几个命令（实际上都是一些宏，许多工具都是通过M4宏处理器建立的，但并不影响这里的讨论）。configure.in文件以命令AC_PREREQ开头，它要求Autoconf的最低版本是2.59。因此，用于生成configure脚本的系统必须安装了GNU Autoconf 2.59。其后的AC_INIT用于告诉Autoconf软件包的名称、版本和作者的电子邮件地址。随后是一连串以AC开头的命令，它们直接确定哪些特征和测试需要包含在configure脚本中。

你很快会了解到configure.in文件中其他行的含义。

这个示例文件中的主要Autoconf命令是：

- ❑ AC_CANONICAL_SYSTEM：由Autoconf来确定是为主机编译还是为另一个目标系统编译，并在必要时处理交叉编译^①。
- ❑ AC_CONFIG_SRCDIR：测试是否存在文件src/hello.c，如果没有发现它，将大声抱怨。这用于确保用户是在正确的目录位置下运行正确的脚本，等等。
- ❑ AC_PROG_CC：测试是否存在C编译器（例如，如果系统中安装了GCC，它将检测到它）。
- ❑ AC_HEADER_STDC：测试标准C头文件的可用性（在/usr/include目录中）。
- ❑ AC_C_BIGENDIAN：确定机器的字节序（详见3.3节）——如果需要，这是你可以添加的众多定制测试中的一个。
- ❑ AC_CHECK_HEADERS：指定要测试的额外的系统头文件。
- ❑ AC_OUTPUT：指定GNU Autoconf运行之后将输出的文件列表。在本例中，GNU Autoconf将导致在顶层目录和子目录src中生成Makefile文件。

configure脚本从configure.in文件中自动生成并基于可移植的shell代码。这一点非常重要，因为要想知道任一给定的Linux或UNIX系统安装的是哪种shell是非常困难（或不可能）的。虽然几乎所有的Linux系统都会安装bash，但既然存在这种可能性，我们还是值得提供这样的灵活性。除非你决

^① GNU Autoconf将自动处理对交叉编译环境的检测，但这可能需要特定版本的GNU 编译器集（GCC）和GNU工具链中的相关部分的支持以使得在一个系统中编译的软件可以运行在另一个不同类型的计算机平台上。

定编写自己的configure测试，否则你不需要担心要编写自己的可移植shell代码（如果真的需要，请查看Autoconf文档）。

3. GNU Automake

Automake用于建立Makefile文件，这样软件就可以使用常规的GNU make调用来轻松的编译，正如你在前面例子中看到的那样。GNU Automake将makefile.am作为其输入文件，在我们的示例项目中有两个这样的文件。第一个文件在项目的顶层目录中，它只是简单地告诉GNU Automake进入src子目录：

```
# A simple GNU Automake example
```

```
SUBDIRS      = src
```

第二个Makefile.am文件在src子目录中，这次有了更多有用的内容。

```
# A simple GNU Automake example
```

```
bin_PROGRAMS      = hello
```

```
hello_SOURCES      = hello.c
```

```
hello_LDADD      = libhello.la
```

```
lib_LT_LIBRARIES      = libhello.la
```

```
libhello_la_SOURCES      = hello_msg.c
```

请注意，这个文件的格式非常简单。你并不需要告诉GNU Automake C编译器或在编译程序源代码时通常需要的C编译器的普通编译标记，事实上，它不需要知道除了要编译的源代码和该源代码的依赖关系以外的任何事情——GNU的自动化工具将为你处理其余的一切。这个例子将产生的程序的名称是hello，所以在Makefile.am文件中有一行bin_PROGRAMS，它就告知Automake该事件。Hello_SOURCES行告知Automake编译hello所需要的源文件。

这个Makefile.am示例文件的最后三行用于GNU Libtool，我们将在下一节对它们进行解释。

4. GNU Libtool

Libtool被设计用来减轻在许多不同的Linux和类UNIX系统（你的项目源代码编译所基于的平台）中编译静态和共享函数库所带来的麻烦。在其他GNU自动化工具和GNU Libtool之间有着紧密的结合，所以你并不需要直接使用GNU Libtool。你只需要在configure.in中添加如前面例子中的一行内容（在这里重复列出以方便你阅读）就足够了。

```
AM_PROG_LIBTOOL
```

这将使得产生的GNU configure脚本知道要启用GNU Libtool。此外，你还需要告诉GNU Automake你的函数库，正如前面的Makefile.am文件所显示的。

```
hello_LDADD      = libhello.la
```

```
lib_LT_LIBRARIES      = libhello.la
```

```
libhello_la_SOURCES      = hello_msg.c
```

这三行告诉GNU Automake hello需要一个额外的和库文件libhello连接的阶段，并提供了函数库本身的编译依赖关系。

GNU Libtool除了可以像本例中那样通过间接调用的方式执行以外，它还可以执行更多的操作。我

们鼓励你进一步研究GNU Libtool文档以通过更多的例子来了解这个可移植函数库生成工具的强大功能。

5. 国际化

当你在自己特定的工作站上进行开发时，请记住Linux的使用遍布全球。发行版的受欢迎程度在不同的地理位置有着很大的差别。有时候，这受某个特定发行版的起源的影响。例如，SuSE Linux在德国和欧洲大陆非常受欢迎，这是因为它起源于德国（尽管目前属于Novell的一部分），而Red Hat历来在美国深受欢迎。这一差别随时间的流逝渐渐变得模糊，但仍然存在——世界上不同的地区流行着不同的发行版。

有时候，整个国家会标准化其本地发行版，因为这些发行版有着很高的本地化水平并支持多种不同的语言。如今，Linux对国际化有相当好的支持，这部分归功于开放国际化倡议（Open Internationalisation Initiative，或称为OpenI18N）这样的标准化工作组，OpenI18N来自自由标准组（Free Standards Group，发布LSB标准的同一组织，开放国际化倡议构成LSB标准的一部分）。你将在www.openi18n.org上找到更多有关OpenI18N规范的内容。

尽管发行版可能具备对国际化的良好支持，但重要的是，只要有可能，就应好好地利用这个能力。为达到这个目的，Canonical公司（Ubuntu Linux发行版的制作者）最近发起了一个名为Rosetta（www.launchpad.net）的项目，它允许各种能力的人（程序员和非程序员）通过Web界面快速地发布针对流行Linux应用软件的语言翻译。

● 语言和Locale

Linux的第一次发布是十多年前的事情了，早期的发行版只包含一个内核和几个简单的工具，最初是为大部分西方（说英语）用户编写的。这在当时非常好，但如今的系统几乎完全是根据用户与系统的交互而构成的一个独立世界。正如你将在本书后面学习到的，如GNOME这样的图形化桌面环境允许在一个现代Linux桌面上呈现一个非常强大且易于使用的最终用户体验。

但是，一个不支持你的（或更为重要的，你的用户的）语言的易于使用的界面能有多好呢？是locale发挥作用的时候了。人们做了大量的工作以提供对广泛语言、各个国家和其他一些用户期望能够针对他们的地理位置所作调整^①的巨大支持。大多数这些工作最初都是由美国（和其他国家）的大型企业所赞助，因为他们需要把他们的软件销售到全世界。如今，由于出现了强大的翻译工具（如由Rosetta提供的），任何人在任何地方都可以做出自己的贡献。

Linux通过使用locale的概念来处理世界各地不同的地区。你可以通过在任一终端上运行下面的命令来确定你的Linux系统上当前可用的locale：

```
$ locale -a
```

现代Linux发行版中支持的locale列表可能相当多也可能非常少。例如，除非你选择安装了许多可选的locale，否则你的系统可能只包含你所处的当前地区所需的locale。你可以参考厂商提供的文档以获得他们所支持的locale的更详细信息。

下面是在作者的机器上可用的locale列表：

^① 例如，你可能期望你的货币和日期显示格式根据你所处位置而发生变化。这些问题并没有在本章中进行具体的讨论，但为了使Linux发行版可移植到世界各地不同的地区，它们确实属于我们需要考虑的问题中的一部分。

```
$ locale -a
en_GB
en_GB.iso88591
en_GB.iso885915
en_GB.utf8
en_US
en_US.iso88591
en_US.iso885915
en_US.utf8
```

一个正式的locale标识符包含的字符串格式如下所示：

```
[language[_territory][.codeset][@modifier]]
```

在上面的列表中，你可以看到这个测试系统支持英国英语（由于种种原因，国际上将它称为大不列颠，简写为GB）和美国英语。每个可用的locale都包括对不同字符编码的支持，其中就有我们将在下一节详细介绍的UTF-8 Unicode字符集。

你可能已注意到在安装你的Linux发行版时出现的locale，虽然安装过程可能并不是以明确的方式要求你设置适当的locale，而是将它放入一些简单的问题中，如询问你的地理位置和语言选择。你可能也需要为你的用户提供同样层次的抽象，以避免他们不得不去理解locale的具体含义。

● 字符集的使用

在过去的20年中，人们制定了一套针对字符编码和表示方式的国际标准。软件厂商已厌倦了实现他们自己的字符编码系统来处理自己产品的多语言版本，他们共同合作以产生一个通用的解决方案，该方案可以普遍适用于处理未来软件不断增长的对语言本地化的需求。这一工作的主要成果之一就是Unicode标准。

除非你在过去的10年中与西藏的僧侣生活在一起，否则你很可能已在自己的软件中遇到过（和使用过）Unicode。使用Unicode字符为Linux编写软件还是比较简单的，但你需要牢记Linux（和其他类UNIX系统一样）最初是基于8位（单字节）US ASCII^①字符编码（甚至更老的编码方式）的，因此，有许多系统工具还不能正确处理Unicode。但这并不是一个大问题，因为我们通常还可以在Linux系统中使用UTF-8编码来存储这些字符。

UTF-8解决了在Linux中使用Unicode可能出现的许多问题。Unicode多字节编码会包含一些特殊的单字节字符，如“\0”（NULL）或“/”，它们在文本文件或文件路径中的使用会引发严重的问题。例如，NULL字符在Linux和类UNIX系统的C函数库中有着特定的含义，它是字符串的结束标记，如果在标准C库函数要处理的Unicode字符串中使用这个字符，就会产生问题。而UTF-8以向后兼容的方式

^① 用于信息交换的美国标准代码。严格来说，ASCII是7位字符编码标准，最初基于电报码。虽然这个标准已有二十年没有更新，但它仍然被广泛使用，尤其是在UNIX世界中。

(ASCII是UTF-8的有效子集) 使用多个单字节进行存储。

UTF-8根据字符本身的编码改变其宽度(即其内存覆盖区域)。那些适合于现有基于7位的ASCII字符集的字符将继续表示为一个单字节,而其他字符则需要一个转义字符来表明存在一个多字节编码的UTF-8字符。UTF-8字符最大长度为6个字节,但你并不用担心UTF-8字符构建的确切方式——你将通过一个易于使用的内部表示,利用库函数来对其进行转换。

你将在本书后面发现,为图形化桌面环境编写软件涉及使用已有的对Unicode非常好的内部支持的函数库。但这并不应该降低你对本节内容的兴趣,因为不论是对桌面软件还是对桌面以外的软件,本节的大部分内容仍然是很恰当的。此外,能够了解隐藏在事物背后的原理对你也很有帮助。

● 宽字符

Unicode是一种用来表示字符集的编码系统,通过使用它,不同的应用程序可以处理不同形状和大小的字符串。从内部来看,一个程序可以使用它所喜欢的任何数据类型来存储数据。但在大多数情况下,程序员通过使用比技术要求略大的内存来减轻他们的负担,因为这样有利于更好地处理字符串和字符。因此,在内存中使用多个字节来存储每个字符是可以接受的,即使使用如UTF-8这样的可变长度编码也可以导致更少的整体内存使用。

现代Linux系统(特别是遵循ISO C90标准的系统)支持宽字符(wide character)概念。一个宽字符(wchar_t)是一个多字节数据类型,它用于在内部表示(例如)一个Unicode字符串中的每一个字符。在大多数系统中,一个wchar_t有4字节宽,或在大多数32位工作站有一个内存字宽。使用宽字符需要包含系统头文件wchar.h,一些老版本的Linux发行版可能没有该头文件。

下面是一个使用宽字符的例子。在这个例子中,你可以看到与使用printf打印一个老式的char字符串相比,如何打印一个使用wchar_t的宽字符串:

```
/* Test program for wide characters
 * Note than use of strlen is merely for demonstrative purposes. You should
 * never use strlen with untrusted strings!
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>

int main(int argc, char **argv)
{
    char *sc = "short characters";
    wchar_t *wc = L"wider characters";

    printf("%ls are more universally useful than %s, ",
           wc, sc);

    printf("but they do use more space (%d as opposed to %d bytes).\n",
           wcslen(wc)*sizeof(wchar_t), strlen(sc));

    exit(0);
}
```

你可以看到wchar_t数据类型用于表示一个宽字符串以及存在标准C库函数的宽字符版本。在本例中，标准C库函数strlen的宽字符串版本wcslen用于计算宽字符串中包含的字符数（与strlen返回的字节数相对应），然后该字符数将与不透明^①的wchar_t数据类型的内部长度相乘。

下面是前面的示例在一个测试用工作站上的输出结果：

```
$ ./wide_characters
wider characters are more universally useful than short characters, but they do use
more space (64 as opposed to 16 bytes).
```

示例代码使用了老式的printf C库函数，但它通过给普通的%s格式字符串加上“l”前缀修饰符来告诉printf使用宽字符。宽字符的声明也是通过在一个标准字符串声明的前面加上“L”修饰符，正如你在wc宽字符串的声明中看到的那样。注意，如果你希望对屏幕上显示字符的宽度的控制有更好的灵活性，你可以使用printf的宽字符版本wprintf。老式的printf函数将它的宽度修饰符看作为字节宽度，这将引发严重的输出对齐问题。

要想获得更多有关宽字符的信息，请查看单一UNIX规范的全文，其中包括了支持宽字符显示和转换的API函数的完整列表。但你需要注意并非单一UNIX规范的所有部分都被非Linux系统所支持，要实现真正的可移植Unicode，你可能需要更进一步约束自己，例如只使用那些由ISO C90标准定义的函数和表示方法。这有几分灰色区域的样子——跨越UNIX/类UNIX和非UNIX的真正Unicode可移植性。

你还可以参考你的Linux系统所带的手册页。输入命令man -k wide characters作为起点。最后，你也许想试试Ulrich Drepper的iconv工具及其库函数，将它作为一种在不同字符编码之间进行转换的简易方式。你的发行版应该在系统的标准位置提供这个工具及其文档。

● 说用户的语言

限制你的软件在全球推广的最大阻力之一来自于以用户自己的母语向用户提供有用输出的需求。除非自然语言的处理真正出现（到那个时候，计算机可能都会自己编写软件了），否则你就需要为你的应用程序与外界（通过用户）沟通的每一个主要输出对话提供翻译。所幸的是，函数库gettext可以帮助你减轻痛苦，而且你还可以通过将翻译外包给那些有语言天赋的人以获得额外的好处。

库函数gettext可以简单地利用一个普通的ASCII文本字符串（按照惯例用英语书写）并使用它作为某一软件对应翻译表格的索引。每一个语言翻译被存储在.po（可移植对象）文件中，它可以与你的软件一起分发。一些应用程序可能包含数十个保存在目录/usr/share/locale/<language>_<region>中的可用翻译文件（显然，Rosetta项目还会进一步增加它的数量）。例如，下面的例子将把用户指向en_US和en_GB locale^②。

下面这个简单的例子可以针对不同的用户输出不同的欢迎信息，这取决于他们所居住的地区（通过环境变量LANG设置的默认locale）：

^① 不透明意味着其内部表示是有意对用户隐藏的。宽字符究竟占用了多少内存并没有太大关系，除非我们需要执行这样的计算。

^② 一个有点做作的例子，但作为一个生活在英国的英国作者，有时你会有一种需要在两种语言之间进行翻译的错觉。尽管有点做作，但这个例子已达到其目的。

```

/* A sample gettext program
 */

#include <stdio.h>

#include <locale.h>      /* LC_ environment variables */
#include <libintl.h>      /* Internationalized library */
#include <stdlib.h>

#define _(String) gettext (String)
#define gettext_noop(String) String
#define N_(String) gettext_noop (String)

int main(int argc, char **argv)
{
    setlocale(LC_ALL, "");

    bindtextdomain("gettext_example", ".");
    textdomain("gettext_example");

    printf("%s\n", gettext("Hello, World!"));

    return 0;
}

```

这段代码首先包含各种标准头文件（其中有头文件libintl.h），然后定义了一些由gettext文档建议的宏，该文档可以在自由软件基金会的网站上找到。对setlocale的调用使得其后的gettext调用将它的翻译选择基于用户可能设置的或还没有设置的各种环境变量。值得指出的是，用户的发行版可能已设置了这些环境变量——通常是基于在安装系统时选择的locale。尽管如此，在本例中，你可以通过设置环境变量LANG来声明一个特定的locale以暂时覆盖默认的locale，从而可以测试不同的应用程序行为。

为了使用gettext，需要进行一些初始化。你必须告诉gettext库函数将要使用的翻译文件所处的位置。通常情况下，它们将位于/usr/share/locale目录下的各种应用程序子目录中（你将在系统中发现数以百计的类似的子目录），但为了使示例代码尽可能的简单，这个例子将在程序可执行文件本身所在目录的子目录中查找翻译文件。bindtextdomain调用告诉gettext翻译文件的位置，而textdomain调用则会强制当前应用程序使用这些翻译。

每当示例程序调用gettext时，如果有针对该特定字符串的翻译，就会返回该翻译字符串。如果没有找到合适的翻译，就将使用默认文本来代替。在本例的情况下，它将默认打印字符串Hello, World，除非存在一个合适的翻译。你可以使用xgettext工具来创建一个翻译(po)文件：

```
$ xgettext -ao gettext_example.po gettext_example.c
```

这将创建一个名为gettext_example.po的文件，它包含示例程序中的每一个潜在的翻译字符串。在本例中，你只需要提供一个针对Hello, World字符串的翻译，所以文件内容可以大幅度地删减。为了举例，这个程序支持的头两个翻译文件将针对美国和英国用户。为此，需要制作两份po文件的拷贝——分别针对这两个国家。

下面的gettext_example.po文件针对的是喝美国咖啡的核心程序员：

```
# gettext_example
# US Translations

#: gettext_example.c:18
msgid "Hello, World!"
msgstr "cup of coffee?"
```

而英国用户得到的是一个略有不同的问候语，更倾向于喝茶的用户：

```
# gettext_example
# UK Translations

#: gettext_example.c:18
msgid "Hello, World!"
msgstr "cup of tea?"
```

这些文件在默认情况下并不会被选择，因为它们还没有在正确的路径下（根据针对gettext所进行的初始化调用）。为了测试这个例子，我们需要创建正确的目录结构：

```
$ mkdir -p en_GB/LC_MESSAGES en_US/LC_MESSAGES
```

你需要使用msgfmt工具将po文本文件转换为机器可读的二进制翻译文件。例如，为了产生一个可用的en_US二进制翻译文件，需要执行如下命令：

```
$ msgfmt -o en_US/LC_MESSAGES/gettext_example.mo gettext_example_en_US.po
```

在实际使用中，你的应用程序支持的每一种语言所对应的gettext_example.mo文件都应位于/usr/share/locale目录中，与其他软件程序所提供的许多已有的翻译文件在一起。在使用美国英语的情况下，这个翻译文件就应在/usr/share/locale/en_US/LC_MESSAGES目录中。请查看你的现有系统已提供了多少翻译文件，以防你需要使用其中的一个文件。你将会为已有这么多日常应用程序已被翻译而感到惊讶。

为测试这个程序，需要将环境变量LANG输出为en_US或en_GB（或任何你想测试的locale）。你将看到类似下面的输入/输出。

设置美国locale：

```
$ export LANG=en_US
$ ./gettext_example
cup of coffee?
```

设置英国locale：

```
$ export LANG=en_GB
$ ./gettext_example
cup of tea?
```

一般系统都会支持许多locale（有一些locale很少会被支持，例如确实存在克林贡（Klingon）^①语言locale，但你的系统很可能不支持它）。像本节这样提供多种英语的翻译是没有什么意义的，这个例子的目的仅仅是告诉你，当你为自己的国家或地区使用非英语翻译时可以做到哪些。为了解更多有关gettext使用方面的信息，请查看在线手册页以及自由软件基金会的网站。

^① 即便如此，但克林贡语言没有针对英文“hello”的官方单词，因此对这个例子的直译将大致变成如“你想要什么？”（What do you want?）这样比较生硬的翻译。——译者注

- 非C/C++的编程语言

你需要注意更高级的语言（如Perl和Python）实现了它们自己的内部Unicode处理机制，与在Linux的系统级别所使用的方法截然不同。你可以继续使用和在其他操作系统平台上一样的方式来编写你的Perl和Python程序。你可能还想单独了解图形化桌面环境中相关问题的解决方法——请阅读本书的第11章以了解在现代图形化桌面环境中如何实现国际化的更多信息。

3.3 硬件可移植性

可移植的Linux软件对不同的人有着不同的含义。Linux是一个高度灵活的操作系统，虽然还有很多人仍然认为Linux指的是运行在标准PC上、针对Intel IA32（x86）处理器的Linux，但其实它本身已被移植到了各种不同的硬件平台上。事实上，大多数（即便不是全部）第三方软件厂商在为“Linux”发行软件时并未充分地进行区分。这一营销策略是由你决定的，但你知道这其中是有差别的。你的软件是否需要支持比最受欢迎的平台更多的平台是一个商业决策^①。

Linux内核中有许多不同的内部API、头文件和其他支持代码，它们以一种尽可能可移植的方式来抽象和管理硬件平台之间的许多不同之处。当你开始在本书后面的章节中探索Linux内核的内部结构时，你将了解更多内核在这方面的能力。但硬件可移植性并不仅仅是影响Linux内核的问题，普通用户应用程序也需要关注自己的字节序问题——在不同机器之间以及在Linux机器和外部世界之间。

3.3.1 64位兼容

至此，如果在阅读本章内容的过程中有一件事给你留下了深刻印象，那就应该是Linux实现的广泛性。通过使用不同类型的软件配置，Linux几乎可以支持所有种类的硬件系统。随着时间的推移，Linux在所谓的“大铁家伙”（Big Iron）机器（来自如IBM和SGI公司的“有房子这么大”的大型机器）上也日益受到欢迎，而这些机器通常至少都是64位的系统。当然，如今你已不需要通过这样的“大铁家伙”来让Linux运行在64位的计算机上，现在许多的硬件平台和下一代的硬件平台都已基于64位技术了。

但64位的真正含义是什么呢？在现代计算机系统的术语中，它通常意味着基于LP64数据模型的计算机。计算机科学家会告诉你：这类机器中的长整型（long）和指针数据类型使用64位表示，而其他数据类型可能使用32位表示（或事实上，使用一些其他表示——例如用于128位浮点数运算或向量计算）。

在确保你的代码兼容64位时所面对的主要问题是不安全的类型转换。在32位机器上的指针通常具备和一个整数（integer）相同的长度，因此执行如下这样可怕的语句已成为一件比较普遍的事情：

```
int i = (int)some_pointer;
```

这不是一个纯净的想法，对64位机器来说更不是一个好主意，因为你将得到一个被截断的指针值，在此之后它可能会破坏程序的数据结构。基于这些原因，人们将void *类型追加到C标准中，并由GCC支持：

```
void *p = some_pointer;
```

^① 也有可能受到可用硬件的限制。当Linux社区项目需要在更少见的平台上测试他们的软件时，通常会面临很大的问题，尽管越来越多的厂商会在他们的用户对某一个大型工具感兴趣并提出移植需求时给予Linux社区项目帮助。

这可以确保将一个指针安全地存储在一些通用的类型中，而不需要将它的类型先转换为一个整型(int)。当你转移到64位机器上并习惯于做出某些假设时，你可能还会遇到许多其他类似的将会破坏程序的情况（除了你自己进行尝试和测试以外，没有其他真正的好方法了）。测试并确保你的代码可以运行在64位平台上是一个底线，同时它也是一个发现并纠正可能已悄悄混进现有代码中的不良做法的很好的方式。你只需在64位开发用机器上进行简单的测试就可以避免许多麻烦。

3.3.2 字节序中立

“他还有充分的理由认为，你骨子里是个大头(Big-Endian)派。叛逆开始总是先在心里盘算，然后才公开行动，因此他指控你是叛徒。”

——《格列佛游记》(约拿旦·斯威夫特)

endian(字节序)的概念非常有趣，如果你之前从未遇到过它更是如此。从本质上说，它可以归结为(endian是双关语——继续阅读即可看出)决定选择哪条路，或在现代计算机中，多字节数据结构在内存中应该如何存储。在使用4字节整数字的现代32位或64位计算机系统中，整数在内存中的存储有两种方式，一种是以最重要字节优先方式存储，另一种是以最不重要字节优先方式存储。

如图3-3所示，整数1234可以按内存地址的升序依次存储在计算机内存中(大头字节序)，或逆序存储(小头字节序)。



图3-3 整数在使用4字节的现代32位或64位计算机的内存中的存储方式

究竟使用哪一种存储约定并没有关系，只要内存的访问方式是一致的即可。处理器在内部被设计为处理以其中的一种格式存储的数据，所以它并不会产生错误的结果。这里的底线是不论哪种存储格式都不应影响执行的任何计算的结果。

既然如此，那为什么我们还说字节序是一个重要的问题呢？请看下面的代码：

```
/*
 * Test the endianess of this machine
 */

#include <stdio.h>
#include <stdlib.h>

static inline int little_endian() {
    int endian = 1;
    return (0 == (*(char *)&endian));
}

void broken_endian_example() {

    union {
        int i;
        char j[sizeof(int)];
    } test = {0xdeadbeef};
    int i = 0;
```

```

        for (i=0;i<sizeof(int);i++) {
            printf("test.j[%d] = 0x%x\n",
                   i, test.j[i]);
    }

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

    printf("This machine is ");
    little_endian() ? printf("little") : printf("big");
    printf(" endian\n");

    printf("This program was build on a machine that is: ");
#if BYTE_ORDER == BIG_ENDIAN
    printf("big endian\n");
#else
#endif
#if BYTE_ORDER == LITTLE_ENDIAN
    printf("little endian\n");
#else
    printf("something weird\n");
#endif
#endif

    printf("and here's a silly example...\n");
    broken_endian_example();

    exit(0);
}

```

你可以用标准方式编译这段代码：

```
$ gcc -o endian endian.c
```

当在IBM POWER5 大头字节序系统中运行这段代码时，将产生如下输出：

```

$ ./endian
This machine is little endian
This program was build on a machine that is: big endian
and here's a silly example...
test.j[0] = 0xde
test.j[1] = 0xad
test.j[2] = 0xbe
test.j[3] = 0xef

```

但当它运行在x86 (IA32) 小头字节序的PC工作站上时，将产生如下输出：

```

$ ./endian
This machine is big endian
This program was build on a machine that is: little endian
and here's a silly example...
test.j[0] = 0xfffffffef
test.j[1] = 0xffffffffbe
test.j[2] = 0xfffffffffad
test.j[3] = 0xfffffffffde

```

很显然，程序在两种不同类型的硬件平台上产生了不同的输出。这一结果的产生是因为系统级别的语言（如C语言）允许你对存储在内存中的数据结构进行底层的访问。通过将一个联合（union）中的32位整数看作一个个单独的8位字符，使得我们有可能以这样的一种方式访问内存元素，从而可以看出两台机器之间字节序的不同。需要注意的是，以这样的一种方式访问内存效率很低。

现代计算机通常在内部通常基于RISC处理器概念的一些变体。它们有着天生的内存字节序并有经过优化的内存访问粒度。老式苹果笔记本中的32位PowerPC微处理器被优化为在任一给定操作中都通过内存读/写32位（或更大）的数据——如果你试图读写一个字节的数据，处理器实际上将执行一个完整的32位操作并将结果截断以返回给你想要的字节。如果你要的数据不是自然的32位对齐的，那么情况会更糟。

这里的底线是一定要小心。你应该经常想想，当你的程序运行在一个与你的开发机器所用字节序不同的机器上时，你的指针运算和类型转换是否会影响程序的正常运行。在不同字节序的机器上测试你的代码以确保程序中没有意外的漏洞，并使用字节序特定的函数包裹你的代码以解决不同字节序的问题。为实现透明的可移植性，你可以使用GNU自动化工具的测试以实现函数的自动替换。

3.3.3 字节序的门派之争

术语 **endian**来自约拿旦·斯威夫特的《格列佛游记》^①。在故事中，一个岛屿上的居民为争论究竟应该从小头还是大头打破煮鸡蛋而发生战争。小头派和大头派各自都完全确信他们的方法才是打开鸡蛋的唯一正确的方法，不理解为什么别人不能看到现实。这个故事不仅本身很有趣，而且它还有助于你去理解计算机科学家眼中的字节序问题——它只不过是另一个无谓的争论而已。

endian（字节序）之所以会引发门派之争是因为开发人员都对计算机应采用何种方法存储整数有他们自己的个人看法，他们会拒绝接受使用其他的方法。事实上，许多现代RISC计算机允许以任一种格式存储数据——只要它知道应使用哪一种。PowerPC就是一个支持这种概念的现代计算机架构的例子。从内部来看，大多数PowerPC处理器是大头字节序（它们继承自IBM^②），但是，如果被置于小头字节序模式，则它将自动执行大头和小头字节序之间的转换。

笔者往往倾向于在调试硬件时使用大头字节序，这是因为物理内存位置将以一种自然的顺序包含数据，使得数据更易读。但几乎在所有其他的情况下，大头字节序和小头字节序并没有实质的差别——你只需知道在特定情况下使用哪种（或应该使用哪种）字节序即可。要注意，在网络操作中，你需要在网络字节序和本地字节序之间进行转换——请阅读本书第5章以获得更多示例。

3.4 本章总结

本章介绍了与编写可移植软件相关的各种主题。严格来说，笔者将可移植性看作为“以这样的一种方式编写软件，使得将编写的软件移植到一个新环境中所付出的努力小于从头开始重建同一个软件所付出的努力”。这一定义允许对编写可移植软件的意义做出一个更加广泛的解释。正如你已发现的，编写软件不仅仅是认真的编写源代码。

现代Linux系统被包装成各种发行版，每个发行版所具备的能力各有不同。为了帮助你制作可以

① 类似的图书经常会在本书中被引用，但实际上作者很少阅读他们。

② 传统上，IBM以“大头字节序”著称，而Intel则以发明“小头字节序”闻名。

运行在各种发行版上的预包装软件，人们已做出了多种努力，例如LSB——Linux标准化规范。但有些时候，你还是不得不自己处理发行版之间的问题。当然，可移植性也不仅仅是指发行版的可移植性，现代Linux软件应该可以在各种目标机器架构和平台上编译和运行——在本章中你已发现了一些可以帮助你的工具。

在阅读完本章后，你应该感到有能力为各种发行版、硬件和软件平台和世界各地使用不同语言的用户编写软件。当然，你并不会在本章中学到你所需要的一切，但现在应该有了一个很好的基础，你可以通过自身的实践在这之上建立自己的知识。你可以阅读一些在线的国际化指南，通过帮助Rosetta项目，在编程时参与和学习。

第4章

软件配置管理

编写现代软件是一个复杂而费时的过程，其中充满了许多漫长而重复的编译周期、错误修复和质量保证。开发人员需要随时掌握整个软件项目的状态以确保在编译和测试过程中所获得的结果的可重现性，并保持整体的生产率。因此，软件配置管理（software configuration management，简写为SCM）构成了软件开发整体过程中的一个很重要的部分。选择合适的SCM解决方案成为一个重要的、可能也是代价很高的过程——尤其当你需要在项目的中期切换到使用新的工具时。

Linux开发者使用两种常见的、但形式却截然相反的SCM。第一种SCM基于更为传统的、集中式开发模型，在这种模型中，开发者修改单个的源文件并将他们的改动传递给中央系统以便保持源代码的一致性，并管理哪些用户可以在指定时间内使用哪些源文件。第二种SCM是真正分布式的，它不依赖于任何集中式的管理，更适合于大型的、地域分布广泛的项目。

在本章中，你将了解可用于Linux的各种类型的SCM并对多种SCM的使用方法进行试验。你并不会在本章中学习到这些工具的所有内容，但你将在阅读完本章后获得足够的知识以找出你所欠缺的知识。此外，你还将了解集成工具（如Eclipse）是如何通过SCM增强其自身功能的。阅读完本章之后，你可能会对在使用Linux过程中遇到的各种不同的SCM工具有所了解，并对使用其他各种商业SCM工具所必须掌握的知识有了一个更广泛的理解。

4.1 SCM的必要性

你过去可能很少遇到软件配置管理这一术语，但与这一更现代的术语相关的过程我们已在实践中使用了很多年，类似这样的情况经常会发生。SCM对不同的人有着不同的含义，在本书中，它指对软件开发及其相关活动的跟踪和控制。

一些与SCM相关的广泛活动包括：

- 将源文件签入/签出共享版本库。
- 与其他开发者同步。
- 维护版本（和分支）。
- 管理编译。
- 集成软件开发过程。

大多数SCM解决方案都涉及某种形式的源代码版本库。在传统意义上，一个版本库构成整个项目的一个中央存储，开发人员通过它签入和签出对源文件的修改。开发人员必须首先建立一个工作空间用于本地的修改，然后签出他们需要修改的文件。他们对源文件的修改将经过测试，一旦被认为是令人满意的，则这些改动将被再次签回版本库。

一个使用SCM的开发者通常会和一个或多个其他的开发者共同工作，后者可能工作在同一项目的其他部分（或在某些情况下，甚至会工作在同一个文件上）。因此，一个SCM工具的主要功能之一就是管理这些个别的改动并在任何时候都向开发人员呈现一致的版本库视图。由于修改（有时也被称为修改集）经常发生，所以版本库需要不断更新以反映这些改动，而开发人员可以通过查看最近的修改以确定他们是否必须更新自己的本地项目视图。

一旦一个开发周期完成，一个典型的SCM工具允许你将版本库的特定状态标记为“准备好发布”。这是定期版本之外的一个版本。每当源文件被修改并签回版本库时都会产生一个定期版本。许多工具还支持分支概念以允许同时存在同一个项目的多个不同的派生版本。例如，当一个项目必须支持多个不同的平台，而且由不同的小组进行开发以支持这些不同的硬件或软件平台时，这就有可能会发生。

正如你已看到的，SCM用于保持对一个软件项目的控制。不论对开发人员、项目经理还是其他参与软件项目的人员来说，能够看到整个开发过程的进展是非常重要的。一个好的SCM解决方案应尽可能地自然和易于使用，从而可以鼓励开发人员更好的利用现有的工具。没有什么比必须每天重复使用一些缓慢而笨拙的工具更让人沮丧的事情了。Linux内核社区使用的git SCM就是这样的一个很好的工具，我们将在本章的稍后部分对它进行介绍——就它所提供的功能来说，它既快速又轻巧。

4.2 集中式开发与分散式开发

当在Linux系统中使用SCM工具时，你将要面对的第一个决定并不是选择哪一个工具，而是选择哪一种类型的工具。在传统的集中式解决方案中，你将使用一个中央版本库并使用一个工具如CVS（concurrent versioning system，并发版本控制系统）来跟踪对中央版本库的签入和签出（当工作要求你处理一个项目中的特定源文件时）。在这种情况下，你通常需要接管项目中某一特定的部分的所有权以避免在同一时间对项目源代码造成多个冲突的修改。

近年来，人们把重点放在了分散式SCM上。这促使了一些强大工具的产生，如Linus Torvald的Linux内核SCM工具git（“傻瓜内容跟踪器”）。在分散式SCM环境中，并没有一个单一的中央版本库——开发人员必须通过它管理对一个项目的所有修改。相反，每一个开发人员维护他们自己的项目树，他们可以自由地做出任何他们想要的修改。这些修改将合并到一个上游维护者的项目树中，而冲突解决过程将帮助处理冲突的修改。

这两种方法各有利弊。集中式系统在一个非常强调组织结构的环境中工作良好，在这一环境中，服务器全天提供服务，开发人员必须遵循严格的程序以递交他们的修改。分散式SCM解决方案生来就缺乏组织性，但它却有在系统中没有单点故障的好处，提供了可以在任何时候任何地点进行修改的能力，以及将多个不同开发者的贡献智能地合并到项目中的能力。

你究竟应该使用哪一种工具取决于你的本地开发环境。如果你只是作为一个单独的开发者或为一个公司中的一个小团队工作，那么你也许更应该安装并使用一个集中式的SCM解决方案。这种环境有一些固有的限制，比如两个开发者不太可能在同一时间处理完全相同的问题（大多数公司都没有能力在这一高度上复制工作），因此一个开发人员可以轻松地在他处理某一特定源文件的时段内接管该文件的所有权。

如果你参与了一个较大的软件项目，不论它是商业的还是属于更广泛的Linux社区，你可能都需要考虑使用一个分散式的SCM解决方案以使得每个开发者尽可能地独立工作。Linux内核就是一个这样的项目，它采用分散式SCM进行管理，而且越来越多的其他项目也都选择了分散式解决方案——这

使得代码更加自由和开放，从而可以让开发者同时展开工作。

4.3 集中式工具

在Linux系统中常用的集中式SCM工具有很多种。事实上，由于这样的工具实在太多，所以我们不可能在这里对它们进行一一介绍。所幸的是，大多数集中式工具都有着类似的行为并分享一套共同的核心功能，当然，它们在某些细节上还是有差别的。你可以自己尝试一下这些工具，并判断在集中式环境中哪一个工具更适合于你使用。

这里并不存在“正确”或“错误”的选择（虽然对此事存在着许多不同的观点），因为在一定程度上它属于个人品味的范畴（包括公司策略）——你个人感觉哪个工具用得很舒服。因此，本节将选择CVS和Subversion作为广泛使用的集中式SCM工具的例子进行介绍，它们被数以千计的Linux开发者所使用，而且这两个工具还管理着世界各地的一些非常大型的项目。

4.3.1 CVS

项目网站：www.nongnu.org/cvs。

CVS（并发版本控制系统）是当今Linux开发人员所使用的最流行的SCM工具之一。它同时也是目前最古老的工具之一，它与其他的一些非常古老的版本控制系统都有关联，例如最初在20世纪80年代开发的RCS（Revision Control System，版本控制系统）。CVS基于中央版本库，版本库中包含的每个源代码项目都受CVS的控制。你使用cvs驱动命令与版本库交互并管理项目中的源文件。

几乎所有的Linux发行版都在它们的开发软件包中包含CVS。如果你没有在你的工作站中找到CVS，请检查你是否在安装过程中安装了必需的开发工具，或使用另外一台工作站进行试验。

与许多在它之前出现的工具不同，CVS支持并发开发（CVS中的“C”即Concurrent，“并发”的意思）。这意味着签出某一特定源文件并开始处理它的开发者不会阻止其他开发者进行相同的工作。以前的版本控制系统需要开发人员明确地标记他们打算编辑某一特定的源文件，从而可以锁住该文件以防止它被其他开发人员修改。我们至少可以说，如果有人锁定版本库中的文件后出去休假了，这是一件多么让人沮丧的事情。因此，当CVS第一次出现在人们的视线中时，真可谓是一项革命性的成果。

当然，CVS并没有什么魔法。为了支持并发开发，它也进行了一些折中。两个开发人员确实可以同时编辑同一个文件，而且可以同时将他们对同一个文件的修改签入版本库。但由于CVS是不可能具备什么“巫术”的，所以它提供了一个冲突解决机制以允许开发人员将他们的修改和其他开发人员的修改进行比较以决定如何成功地完成修改的整合。

为使读者快速掌握CVS，下面几节将包含一些关于如何创建和管理CVS版本库的有用信息。虽然你并不需要创建自己的CVS版本库进行试验，但如果你这样做了，你将拥有更大的自由度并且可以安全地尝试你的想法而不会对其他用户带来问题。除了建立你自己的版本库以外，因为许多大型的开放源码项目都是使用CVS进行管理的，所以你还可以使用由你所喜爱的开放源码项目所提供的CVS服务器。

热门的Source Forge网站（www.sourceforge.net）被许多开放源码项目所使用，它为每一个设立在该网站上的项目提供了CVS服务器设施并提供一个指南以说明你如何作为一个匿名的远程用户使用

这些设施。虽然你不能作为一个匿名用户将修改签入CVS版本库，但你可以进行一些试验。

1. 创建一个CVS版本库

要创建一个自己的CVS版本库，可以使用下面的命令：

```
$ cvs -d /usr/local/cvsroot init
```

这告诉CVS在目录/usr/local/cvsroot下创建一个新的版本库。需要注意的是目录/usr/local通常被保护，只有超级用户具有写权限。由于你通常不会以root用户身份登录你的工作站，所以也许你可以使用sudo命令以代替直接以root用户身份登录来创建版本库并修改目录cvsroot的权限（使用chown命令或为CVS访问中央版本库创建一个共享的用户组），或者你为版本库选择另一个目录路径。

将CVS版本库保存在哪里并没有关系——如果你想要有一个自己的CVS版本库用于试验，你可以使用家目录下的一个子目录。

名字cvsroot是非常重要的。它是CVS管理的版本库的根目录，它将包含由CVS控制的每一个项目。所有CVS的操作都是相对于这个目录的，所以为了避免每次调用命令时都需要告诉CVS版本库的根目录，将环境变量CVSROOT输出是一个好主意。

```
$ export CVSROOT=/usr/local/cvsroot
```

2. 创建一个新的CVS项目

一旦创建好CVS版本库并设置了CVSROOT，你就可以创建一个新的由CVS管理的项目或将现有项目导入到版本库中，把它置于CVS的控制之下。一般情况下，你已经创建了一个项目并希望将它导入现有的CVS版本库，正如我们在这里所举的例子一样。你可以使用任何现有的项目作为例子，例如第2章中的示例代码。使用以下命令将目录toolchains签入你的新CVS版本库：

```
$ cd toolchains
$ cvs import -m "toolchains example" plp/toolchains plp start
```

你将看到如下所示的CVS输出（为简洁而进行了删减）：

```
cvs import: Importing /usr/local/cvsroot/plp/toolchains/src
N plp/toolchains/src/Makefile
N plp/toolchains/src/hello.c
N plp/toolchains/src/hello
I plp/toolchains/src/main.o
N plp/toolchains/src/message.c
N plp/toolchains/src/hello.s
N plp/toolchains/src/main.c
I plp/toolchains/src/message.o
N plp/toolchains/src/trig
I plp/toolchains/src/libmessage.so
N plp/toolchains/src/trig.c
N plp/toolchains/src/goodbye_shared
```

```
No conflicts created by this import
```

请注意CVS在导入的每个文件名的左边显示的“N”或“I”。该代码显示了版本库中该文件的状态。“N”表示这个文件是在这次导入中新创建的，而在目标文件如main.o、

message.o和libmessage.so左边出现的“**I**”表示CVS已注意到这些是目标代码文件，它将忽略它们。通过特殊的.cvsignore文件你可以控制哪些文件应该被CVS忽略，你可以在以后的具体项目目录中创建该文件。要想了解更多细节，请查看CVS文档。

上面的CVS命令将toolchains项目导入到由环境变量\$CVSROOT指定的CVS版本库中。项目源代码本身存放在CVS版本库的\$CVSROOT/plp/toolchains目录中，这样可以允许其他相关的项目在稍后阶段一起放入同一个plp目录。上面命令中的plp的作用是作为这个项目的厂商标记，而start指定了一个初始的版本标记。你并不需要立刻使用这两个额外的标记，但因为CVS在创建新版本库时需要使用它们，所以我们在上面的命令中包括了它们。

在默认情况下，当签入任何源文件到版本库中时，CVS将启动一个编辑器（通常由环境变量\$EDITOR决定），让你有机会针对这次特定的签入输入一些说明性的文字。通过在上面命令中使用“-m”命令行标记，你可以告诉CVS使用文本toolchains example作为这次特定签入的信息，从而避免了启动一个外部文本编辑器。图形化的CVS工具（包括对开放源码Eclipse IDE的扩展）通常会调用它们内置的签入编辑器。

3. 从CVS中签出源文件

一旦一个项目已被导入到CVS版本库中，你就需要在处理源文件之前先把它签出。要做到这一点，首先创建一个新的工作目录并确保你的环境变量CVSROOT已被正确的设置（指向先前创建版本库的位置），然后在你的工作目录中（签出的源文件就将保存在该目录中）运行以下命令：

```
$ cvs co plp/toolchains
```

你还可以使用cvs驱动命令的“-d”标记来覆盖CVSROOT变量——请参考CVS手册页以获得这方面的示例，在手册页中还包括对访问方法的修改。例如，我们可以告诉CVS它必须使用Kerberos安全票据来认证对你的安全公司的CVS服务器的访问，或告诉CVS它必须使用SSH来和坐落在地球另一端的CVS服务器进行通信。CVS的功能非常强大，只要你去利用它。

在要求CVS从中央项目版本库中签出一些源文件之后，你将看到类似下面的输出，目录被创建和文件被签出：

```
cvs checkout: Updating plp/toolchains/src
U plp/toolchains/src/Makefile
U plp/toolchains/src/goodbye_shared
U plp/toolchains/src/hello
U plp/toolchains/src/hello.c
U plp/toolchains/src/hello.s
U plp/toolchains/src/main.c
U plp/toolchains/src/message.c
U plp/toolchains/src/trig
U plp/toolchains/src/trig.c
```

CVS的签出进程创建了目录plp，在该目录下包含示例项目源文件的toolchains目录。请注意在文件名左边的“U”标记表示从版本库中签出的版本更新了本地文件，在本例中就创建了这些文件。

要签出项目的最新版本，例如包括其他开发者所作修改的版本，可以运行以下命令：

```
$ cvs update
```

在中央版本库中与你本地文件不同的任何文件都会被下载。你在本地所做的修改在此时将被标记出来以便你解决冲突。在试图签入项目之前先要求CVS签出项目通常是一个好习惯，因为这可以确保你的修改基于CVS版本库中最新的项目版本。你将在下一节学习到更多有关如何将你对项目的修改签入CVS的内容。

4. 签入修改

修改单个源文件并将它签回中央版本库对CVS来说是非常容易的。作为一个例子，我们对hello.c源文件进行了修改（阴影显示的部分）：

```
/*
 * Professional Linux Programming - Hello World
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    printf("CVS is a Software Configuration Management tool!\n");
    return 0;
}
```

首先像平常一样使用make命令编译和测试这个程序，然后使用以下命令将修改的文件签回CVS版本库：

```
$ cvs commit
```

在本例中，由于没有在CVS命令行中使用-m标记，cvs将自动调用一个编辑器，你可以在编辑器中输入一段针对所做修改的简短说明（例如，“silly message added to hello world example at the request of the Bob in support”，其含义是“应用户Bob的要求，在hello world示例程序中添加了一条无聊的消息”）并保存。

CVS将产生如下所示的输出，并完成它的提交动作：

```
cvs commit: Examining .
/usr/local/cvsroot/plp/toolchains/src/hello,v  <- hello
new revision: 1.2; previous revision: 1.1
/usr/local/cvsroot/plp/toolchains/src/hello.c,v  <- hello.c
new revision: 1.2; previous revision: 1.1
```

5. 添加、删除文件和目录

在默认情况下，CVS只会跟踪那些它知道的文件。它不会因为你的项目的本地版本发生了改变就自动在版本库中添加或删除文件。因此，你需要使用CVS的add和rm命令在CVS项目中添加、删除文件和目录。下面就是一个例子，它创建一个新目录并删除一些旧的cruff^①：

^① 随着软件的发展，以及经历了修改bug和更新的若干周期，它的部分代码已不再使用，但仍然保留在源代码中，这种代码称为cruff。cruff的尺寸范围可由一两行无用代码到整个源文件模块。由于很难识别cruff，所以去除它往往很困难。——译者注

```
$ mkdir rev2
$ cvs add rev2
Directory /usr/local/cvsroot/plp/toolchains/src/rev2 added to the repository
$ cd ..
$ rm README
$ cvs rm README
cvs remove: scheduling 'README' for removal
cvs remove: use 'cvs commit' to remove this file permanently
```

要注意，直到你执行下一次CVS提交动作，版本库中对文件或目录的删除才会真正发生：

```
$ cvs commit
cvs commit: Examining .
cvs commit: Examining src
/usr/local/cvsroot/plp/toolchains/README,v  <-- README
new revision: delete; previous revision: 1.1.1.1
```

6. 查看CVS历史

CVS维护着记录其活动的详尽日志以及签入版本库的文件的每一个版本之间的所有差异。为了查看一个指定版本库的历史，你可以使用CVS的history命令。例如，可以使用以下命令来查看针对文件hello.c所做出的所有修改：

```
$ cvs history -c hello.c
M 2006-02-12 01:52 +0000 jcm 1.2 hello.c plp/toolchains/src ==
~/cvs/plp/toolchains/src
M 2006-02-12 02:19 +0000 jcm 1.3 hello.c plp/toolchains/src ==
~/cvs/plp/toolchains/src/cvs1/plp/toolchains/src
M 2006-02-12 02:29 +0000 jcm 1.4 hello.c plp/toolchains/src ==
~/cvs/plp/toolchains/src/cvs2/plp/toolchains/src
M 2006-02-12 02:30 +0000 jcm 1.5 hello.c plp/toolchains/src ==
~/cvs/plp/toolchains/src/cvs1/plp/toolchains/src
```

CVS的history命令可以使用的选项标记记录在CVS手册中，你也可以通过在线的man手册和GNU的info文档进行查找。

7. 冲突解决

作为一个并发的开发工具，CVS真正强大的功能是它处理冲突的能力。当两个开发者试图修改同一个文件并相继（这里总是会存在着一个顺序问题，即使两个开发者同时提交他们的修改，因为在实际执行签入操作时，版本库将被锁住）将他们的修改签回中央版本库时，冲突就会发生。这种冲突状态仅仅通过自动化的行动是不能轻易解决的。

因为对同一源文件的修改可能涉及微妙的语义变化（ABI破坏、锁定级别的改变以及一些更加恼人的问题），这些问题不能被自动解决，因此CVS将进入一个冲突解决模式，请你协助它进行处理。

为测试CVS的冲突解决特征，你可以创建两个测试目录并使用前面已介绍过的命令签出示例项目的两份单独的拷贝。然后修改两个hello.c文件以分别打印不同的文本字符串。在未经版本更新的前提下，相继签入该文件。第一个签入将成功，但第二个签入将产生如下的信息：

```
$ cvs commit
cvs commit: Examining .
cvs commit: Up-to-date check failed for 'hello.c'
cvs [commit aborted]: correct above errors first!
```

CVS知道在上一次签出项目拷贝之后中央版本库已被修改，而你现在又试图将修改提交回版本库。为修复这一状况，你必须先运行一个更新命令以将你的本地项目和版本库中的项目进行同步：

```
$ cvs update
cvs update: Updating .
RCS file: /usr/local/cvsroot/plp/toolchains/src/hello.c,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into hello.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in hello.c
C hello.c
```

更新进程注意到你对文件hello.c做出了修改并试图通过修改该文件的本地拷贝以帮助你解决这一冲突：

```
/*
 * Professional Linux Programming - Hello World
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    <<<<< hello.c
        printf("goodbye!\n");
    =====
        printf("hello!\n");
    >>>>> 1.3

    return 0;
}
```

你现在可以纠正这一冲突并像前面一样再次签入你的修改了。

如果你发现你经历了过多的冲突并认为做出一些限制是正当的，那么你可以使用CVS来锁定文件以防止它被修改。虽然我们并不推荐你对每一次签出都这样做，但你可能会发现这是一个比较容易的选择以保证你在某个特定场合是唯一一个可以修改某个特定文件的开发者。请查看CVS文档中的CVS admin命令以了解更多有关使用CVS锁定文件的信息和示例。

8. 标记和分支

CVS提供一个机制以允许你在某个阶段给你的项目命名一个特定的修订版本号。在项目开发中，当你需要参考应用程序的某一个特定开发版本，但又没有发布它或由于其他类似原因时，这一机制就非常有用。要创建一个修订版本，可以使用CVS的tag命令：

```
$ cvs tag dev_rev123
```

CVS将标记工作目录中的每一个文件并产生如下所示的输出：

```
cvs tag: Tagging src
T src/Makefile
T src/goodbye_shared
T src/hello
T src/hello.c
T src/hello.s
T src/main.c
T src/message.c
T src/trig
T src/trig.c
```

除了项目中的每个文件都被标记为属于一个特定名称的版本（在本例中，是`dev_rev123`）以外，其他都没有改变。

除了标注以外，CVS还提供一个更强大的机制用于标识项目中明显的分支点。分支是一个很方便的机制，它用于冻结项目的某一特定版本并采用平行开发。你可能会在发布应用程序的一个新版本之后采用这一方法。虽然你想继续开发你的项目，但你仍然希望保留软件在发布时的状态，所以你会在此时对软件进行分支。

要创建一个CVS分支，你可以使用`rtag`命令来指定版本库的位置：

```
$ cvs rtag -b release_1 p1p/toolchains
```

这将立刻创建分支`release_1`而不用执行CVS提交命令。

分支能让你在CVS中保持对老版本软件的跟踪，因为你需要在今后的一段时间内继续支持该版本。虽然我们可能不会再在软件项目的老版本分支上进行最新的开发，但可以有一个单独的团队继续提供老版本的支持和不定期的更新。例如，修复发行版本中的一个紧急的安全漏洞。

你可能想知道分支和修订版本之间有什么实际的差别。在许多开发环境（尤其是在企业）中，分支用于主要产品版本，而修订版本在项目每次进行测试编译时都会产生。在一个CVS版本库中可能会有成千上万个修订版本，但只会有少数几个分支——这就是这两者之间的本质区别。

要了解关于分支和修订版本的更多信息，请查看CVS文档。

9. 分布式开发

到现在为止，本节中所给出的示例都是有关处理本地CVS版本库的，它与使用该版本库的开发人员位于同一台机器上。这种操作模式非常适合在一个共享的Linux服务器环境中使用，但当开发人员相距很远^①，必须远程连接中心服务器时，这种模式就不太适合了。

有两种简单的方法可以远程使用CVS服务器：

- 通过SSH连接访问CVS。
- 通过远程pserver访问CVS。

你会选择上述两种解决方案中的哪一种在很大程度上取决于你的本地开发环境和你所参与项目的策略。在一个不可信任的环境中，如通过因特网进行工作，我们就鼓励使用SSH服务器而不是pserver。SSH服务器引入了更强的说明性并在开发者和远程版本库服务器之间提供了进行加密的安全连接。

^① 这里要考虑的不仅仅是时区问题，而是影响社区Linux项目的各种问题——会议、休假、时区以及不同的工作习惯。在一个大型项目中，开发人员随时都会签入修改的源文件而不会考虑你是否正在享受你喜欢的国定假日。

- 通过SSH连接访问CVS

你以前很可能已经用过SSH，因为它是通过网络远程连接Linux服务器的首选机制。通过SSH访问CVS是一个非常简单的过程，你只需要在包含CVS版本库的远程SSH服务器上注册一个账号即可，然后你就可以通过输出一个环境变量来启用SSH。

```
$ export CVS_RSH=ssh
```

这告诉CVS使用SSH协议连接版本库服务器。

环境变量CVSROOT当然也需要更新以反映新的远程版本库位置。它所使用的格式如下所示：

```
$ export CVSROOT=username@hostname:/path/to/cvsroot
```

- 通过远程pserver访问CVS

对于那些不使用SSH协议的开发者，CVS提供了不太安全的CVS pserver。它通常被包含可被公众访问的版本库的开放源码项目所使用，因为这些项目不可能为每一个可能的开发者在中央版本库服务器上建立一个账号。这样做主要是出于安全考虑，因为给通过因特网连接的不可信的、未知的第三方以登录系统的信任是非常不可取的。

你将使用格式为:pserver:cvs@pserver:/path/to/cvsroot的CVSROOT来连接到远程pserver。然后你需要使用命令cvs login来连接和认证：

```
$ cvs login
```

现在你就可以像以前一样使用常规的CVS命令了，这里没有“退出”(log out)的概念。

作为一个使用pserver的例子，你可以在一台安装了CVS的Linux工作站上使用以下命令来签出流行的Samba项目的源代码。

```
$ export CVSROOT=:pserver:cvs@pserver.samba.org:/cvsroot  
$ cvs login  
$ cvs co samba
```

10. 局限性

近年来，CVS的一些局限性逐渐开始体现出来。首先，项目修改的跟踪基于每个文件而不是每次的改动。开发人员通常会在修复一个漏洞或实现一个新的软件功能时修改多个文件，所以最好能将所有这些修改作为一个单独的修改集来查看。在CVS中，要想统计出有哪些文件在一系列修改中被改动是非常麻烦和困难的。虽然也有方法可以解决这一问题（例如使用大量的修订版本标记），但这并不十分完美。

也许CVS最严重的缺陷是它不能将修改的提交处理为一个原子操作。你可能理所当然地认为，如果一个开发者在运行一个CVS提交命令的同时有另一个开发者在执行一个CVS更新命令或签出源文件，这个更新操作将看到所有的版本库修改或什么也看不到。但不幸的是，在CVS中，情况并非如此。两个开发者可能会重叠他们的活动，此时如果想弄清楚哪里出了问题将会非常麻烦。虽然这并不是会经常出现的情况，但你需要提防它。

CVS的最后一个问题是它不能很好地处理文件更名。由于修改和特定的文件是绑定在一起的，所以如果你对其中的文件进行更名，CVS将会变得非常“不开心”。对这一问题有不同的解决方法，其中包括手工修改版本库结构、将更名文件作为新文件签入并删除旧文件，等等。对长寿的CVS的最后

一击来自于它的主要开发者，他们现在认为CVS的各种问题已根深蒂固且难以修复，所以已开发了Subversion作为其替代。但即便如此，CVS很可能还会存在许多年。

4.3.2 Subversion

项目网站：<http://subversion.tigris.org>。

正如你在上一节中看到的，Subversion是CVS的一个替代产品。它解决了许多困扰CVS原本的设计问题，同时还引入了更多的灵活性。与CVS一样，Subversion被设计用于支持一个集中的、共享的开发者版本库，但值得引起注意的是，现在有一个完全独立的项目svk，它在Subversion之上进行了扩展以支持非集中式的开发。因此，如果你不能完全确定使用哪一种形式的开发并希望能够规避风险，那么Subversion提供了一个很好的过渡解决方案。

与CVS不同，Subversion不仅保存文件内容的版本信息，还保存与目录、拷贝和更名相关的元数据。因此，它可以处理困扰CVS用户很多年的棘手的文件更名问题。此外，Subversion中的提交动作是原子化的，在整个提交动作成功之前不会有部分提交动作生效。因此，当正好有两个开发者试图（或稍有偏差）同时从同一个版本库执行提交和更新动作时，不会产生开发者驱动的竞争条件。

因为CVS和Subversion是如此相似^①，所以本节并不想像介绍CVS那样详细介绍Subversion。如果你想了解Subversion更多的日常使用方法，请参考其在线文档，毫无疑问它将比本书的内容更新。Subversion是一个在不断发展的项目，所以你最好直接通过其项目网站了解其最新的开发状况。

要注意的是，和CVS不同，默认情况下Subversion可能没有在你的Linux工作站上被安装。

如果你发现不能在你的系统中找到这里描述的工具，请访问你所使用的Linux发行版的网站以获得可能已准备好的任何更新。此外，你也可以直接访问Subversion的网站，该网站提供了针对许多不同Linux发行版的软件包。

1. 创建Subversion版本库

要创建一个新的Subversion版本库，可以使用svnadmin命令：

```
$ svnadmin create /usr/local/subrepos
```

这个新的版本库与其对应的CVS版本库非常类似，除了版本库的元数据现在是存放在BerkeleyDB文件中而非CVS所使用的老式的RCS文件中以外，一般情况下，你并不需要担心版本库的内部文件类型和结构。

2. 创建新的Subversion项目

将一个现有项目导入Subversion版本库比导入CVS版本库更简单。虽然Subversion也可以使用标记和分支，但它不再需要在项目创建时指定它们。相反，你只需要告诉Subversion一个项目所在的目录，你希望将Subversion控制置于该项目中。

```
$ svn import toolchains file:///usr/local/subrepos/trunk
```

位置/usr/local/subrepos是任意选择的——你可以选择一个适合于你的系统的位置来放置Subversion版本库。

Subversion在执行初始导入时将产生如下所示的输出：

^① 正如我们已说过的，CVS和Subversion共享了许多相同的开发者，所以这不足为奇。

```

Adding      toolchains/src
Adding (bin) toolchains/src/hello
Adding      toolchains/src/hello.c
Adding      toolchains/src/hello.s
Adding      toolchains/src/message.c
Adding      toolchains/src/main.c
Adding (bin) toolchains/src/trig
Adding      toolchains/src/trig.c
Adding (bin) toolchains/src/libmessage.so
Adding      toolchains/src/Makefile
Adding (bin) toolchains/src/goodbye_shared

```

注意，这与CVS的行为非常相似，除了现在你可以清楚地看到哪个文件是二进制文件（在括号中的bin）和哪个文件是常规的程序源代码（或至少不是二进制文件）以外。

3. 从Subversion中签出源文件

将源文件从Subversion版本库中签出的命令看上去也很熟悉：

```
$ svn checkout file:///usr/local/subrepos/trunk toolchains
```

这个命令的输出如下所示：

```

A   toolchains/src
A   toolchains/src/hello
A   toolchains/src/hello.c
A   toolchains/src/hello.s
A   toolchains/src/main.c
A   toolchains/src/message.c
A   toolchains/src/trig
A   toolchains/src/trig.c
A   toolchains/src/libmessage.so
A   toolchains/src/Makefile
A   toolchains/src/goodbye_shared
A   toolchains/README
Checked out revision 1.

```

4. 签入修改

为了对Subversion的签入过程进行试验，我们对先前的hello.c示例代码再次进行简单的修改：

```

/*
 * Professional Linux Programming - Hello World
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    printf("Subversion rocks!\n");
    return 0;
}

```

此外，我们修改源文件message.c以将其打印的消息改变为更适合于这个例子的消息：

```
#include <stdio.h>

void goodbye_world(void)
{
    printf("Look! Subversion uses proper changesets!\n");
}
```

你可以通过Subversion的status命令找出你编辑过的文件：

```
$ svn status
M      hello.c
M      message.c
```

Subversion知道你已在一次事务处理中修改了这两个文件，并准备好在提交这些修改时将它们绑在一起作为一个完整的修改集。

● 修改集

我们在前面的对CVS设计缺陷的讨论中说过CVS是基于文件的，它不能将修改组合成一个关联实体以代表在一次开发周期中的所有修改。在CVS中，要想看到为了修复一个漏洞或添加一个新功能所做的所有修改并不是一件容易的事情，但这并没有成为困扰其继承者Subversion的问题。为了证明这一点，请使用commit命令将你的修改合并到Subversion版本库中。

```
$ svn commit
Sending      src/hello.c
Sending      src/message.c
Transmitting file data ..
Committed revision 1.
```

Subversion的log命令可以用于查看提交给版本库的所有修改。例如，可以使用以下命令来查看在修订版本1中修改的文件：

```
$ svn log -r 1 -v
-----
r1 | jcm | 2006-02-12 05:19:46 +0000 (Sun, 12 Feb 2006) | 2 lines
Changed paths:
  M /trunk/src/hello.c
  M /trunk/src/message.c

checked in a bunch of files
```

你可以清楚地看到相关的修改已被组合到一个实体中，被称为toolchains示例项目的修订版本1。

5. 分布式开发

Subversion并不使用传统的CVS pserver方式来与更广泛的开发者共享项目。相反，Subversion使用最新的基于HTTP的WebDAV/DeltaV协议进行网络通信，并通过Apache Web服务器提供对版本库的访问。由于依赖于Apache作为其服务器，所以Subversion可以通过Apache使用普通Web页面可以使用的任何认证和压缩功能，而不需要提供自己的实现。

从一个普通开发者的角度来看，远程Subversion版本库的使用方法和其对应的CVS非常相似。例如，要通过Subversion签出Samba项目的源文件，可以使用以下命令：

```
$ svn co svn://svnanon.samba.org/samba/trunk samba-trunk
```

你不需要登录或指定任何额外的选项，因为Subversion会负责其余的一切。

4.4 分散式工具

分散式SCM工具在过去的几年中已变得非常流行，尤其是在Linux社区项目中。因为开发人员往往分散在世界各地，并同时工作在许多不同的问题上——这通常远远超过了单个人或强制每个活动必须通过一个中心服务器可以轻松跟踪的程度。此外，当你携带笔记本参加一个会议并突然有了编写代码的激情时，集中式SCM解决方案往往不能很好的工作。

由于存在着太多的分散式SCM工具，所以我们不可能在像本书这样的图书中介绍所有这些工具（毕竟这个主题需要整本书来介绍）。当你阅读本书时，很可能已有一个全新的、革命性的工具正在被发明——git的迅速开发就是一个例子。幸运的是每个工具的行为和使用方法都非常相似，你使用一个工具所学到的概念可以很快地应用到其他工具上。

大多数分散式工具继续使用版本库——尽管它们可能不再使用这一术语，并且开发者的日常活动与他们在使用集中式SCM工具时的活动非常相似。在集中式SCM解决方案中通常只有一个远程版本库的本地拷贝，但在分散式SCM环境中，同一个版本库有许多不同的拷贝是很常见的——每一份拷贝针对你参与的一个不同的开发线路（或分支）。

两种SCM解决方案最大的差别是在对上游版本库提交动作的处理上。在分散式SCM的情况下，提交过程实际发生在两个开发者之间。一个开发者或者是将修改主动推送给另一方，或者（更有可能）要求另一方引入已修改版本库的拷贝并进行合并。一个特定的分散式SCM真正的实力是其智能化处理源代码树合并的能力，这些源代码树可能共享一个共同的祖先，但各自都经过了大量的开发。

很明显工具的分散并不会减弱对集中式项目管理的需要。毕竟，如果缺乏某种形式的指导，就不会产生某个特定项目的正式版本。分散式SCM并不会设法解决如何管理一个大型而复杂的问题——它们只是使得开发者的日常代码编写变得更容易。通过一系列的“引入”动作将开发者的修改集成到一个正式版本中的任务需要一点耐心和几个小时的空余时间来让一切同步。

两个常见的提供分散式解决方案的工具是Bazaar-NG（基于GNU Arch）和Linus Torvalds的Linux内核SCM工具git。前者已发展成为一个通用的解决方案，它被Canonical（Ubuntu）的开发者大量使用，他们还使用Bazaar来管理许多其他项目。后者是专门为Linux内核开发的，其设计的初衷是为了纯粹的速度——git是现有最快的SCM工具之一。你将在本章后面学到更多有关git的内容，但我们先将在下一节简要介绍Bazaar。

4.4.1 Bazaar-NG

Bazaar-NG（bzr）继承自Bazaar 1.x，它本身试图在名为GNU Arch的SCM之上建立一个易于操作的层。GNU Arch是一个功能强大的SCM工具，但对最终用户和开发者来说，它并不易于使用^①。相反，GNU Arch侧重于解决和分散式SCM技术相关的有趣的计算机科学问题。Bazaar之所以变得相当受欢迎是因为它在设计中考虑到了开发者必须经常使用它。Bazaar和Arch不尽相同，它们之间曾经出现过分裂，但仍保留了大致的兼容。

GNU Arch和Bazaar-NG之间的差别集中在可用性上，例如：

^① 和许多其他GNU项目一样，想法很好，但开发的最初目的更多地是为了理论兴趣。

- Bazaar可以在大多数命令中接受URL。我们可以在**baz get**命令中直接指定远端位置，而不需要传递额外的选项标记。
- Bazaar不需要被告知应该使用哪一个版本库，如果没有指定，只要有可能，它就会使用当前的本地源代码树。
- 如果在合并时发生冲突，整个项目树将被标识为有冲突。除非这样的冲突被完全解决，否则像**commit**这样的命令将不会被执行。

1. 配置Bazaar

许多分散式SCM工具必然需要某种形式的唯一用户标识符以便跟踪每个开发者所作的修改，并当这些修改在开发者之间进行传递时使用一个唯一的标记来标识它们。虽然有一些其他的解决方法，但Bazaar使用电子邮件地址作为标识符，并将它应用到你在Bazaar的控制下所做的所有工作中。

用户信息保存在配置文件**bazaar.conf**中，该文件本身位于家目录下的**.bazaar**子目录中 (`$HOME/.bazaar/bazaar.conf`)。你可以为bazaar配置一个合适的电子邮件地址，如下例所示：

```
[DEFAULT]
email=jcm@jonmasters.org
```

现在当Bazaar代表它所配置的用户账号控制版本库和分支时，它将使用默认标识符jcm@jonmasters.org。

2. 创建一个新的Bazaar分支

为了学习Bazaar的使用，你可以在自己的工作站上将现有的**toolchains**示例代码拷贝到一个新创建的工作目录中，并使用命令**bzr init**将该目录置于Bazaar的控制之下：

```
$ bzr init
```

这将在你的项目源代码目录中创建**.bzr**目录，该目录用于Bazaar跟踪对你的项目的修改。

要想真正地跟踪一个文件或将文件置于Bazaar中，你需要使用命令“**bzr add**”来添加它们：

```
$ bzr add .
```

Bazaar在添加项目中的每个文件时将产生如下输出：

```
added src/Makefile
added src/hello.c
added src/hello
added src/message.c
added src/hello.s
added src/main.c
added src/trig
added src/trig.c
added src/goodbye_shared
ignored 4 file(s) matching "*.o"
ignored 2 file(s) matching "*.so"
If you wish to add some of these files, please add them by name.
```

除了创建自己的版本库以外，你还可以通过指定一个现有Bazaar版本库的URL来让Bazaar下载它。

请注意，Bazaar在添加操作中自动忽略了几个目标文件和库文件。和CVS一样，Bazaar知道几个标准的文件后缀名，并将在执行类似扫描的操作中忽略它们——大多数人不会在SCM系统中跟踪二进制预编译函数库版本，如果在某些情况下需要这样做也应该非常谨慎（例如，当使用基于SCM的某种

类型的自动化工具来跟踪不同函数库版本之间的ABI改动时)。

3. 签入修改

虽然Bazaar是一个分散式SCM解决方案，但它支持将修改提交到你的本地项目分支的概念。当你在一个受到Bazaar控制的项目中编辑文件时，这些修改并不会自动成为你所创建的Bazaar分支的一部分。为了维护Bazaar中的修改记录，你需要执行一个提交操作。你可以对hello.c示例代码略做修改并将它签入Bazaar版本库进行试验：

```
$ bzr commit -m "modified hello.c"
Committed revision 1.
```

与Subversion以及其他最新的SCM工具一样，Bazaar以修改集为单位，所以对在一次修改周期中所修改的源文件集，你将只看到一个修订版本。你可以使用命令bzr log来查看你的版本库历史，这和我们之前讨论的工具用法基本一样。

```
$ bzr log -v
-----
revno: 2
committer: Jon Masters <jcm@jonmasters.org>
branch nick: toolbaz
timestamp: Sun 2006-02-12 09:55:34 +0000
message:
    changeset
modified:
    src/hello.c
    src/trig.c
```

4. 合并Bazaar分支

当使用由Bazaar管理的项目时，你将自己的修改放入本地分支并希望整合由其他Bazaar分支所做的修改，这时你可以使用命令bzr merge。这个命令使用另一个Bazaar版本库的位置作为参数并试图将你的本地修改和其他版本库的修改合并。例如，你可以创建现有Bazaar分支的第二份拷贝，同时修改两份拷贝，然后将这两个源代码树合并到一起——这样做虽然有一些牵强，但已足以模拟现实生活中的合并。

要拷贝你的现有Bazaar分支，可以使用branch命令：

```
$ bzr branch toolchains toolchains1
```

接下来，为了测试解决冲突的方法，同时修改两个项目并使用bzr commit命令提交这些修改。现在你可以尝试使用merge命令来合并这两个分离的源代码树：

```
$ bzr merge ../toolbaz1
bzr: WARNING: Diff3 conflict encountered in
    /home/jcm/toolchains1/src/hello.c
1 conflicts encountered.
```

在本例中，Bazaar发现两个版本库对同一个文件hello.c所做的修改互不兼容。为解决这一问题，Bazaar除了修改原hello.c文件并对不兼容的部分进行了注释以外，它还创建了两个额外的文件hello.c.BASE（原hello.c）和hello.c.OTHER（合并候选文件），修改后的hello.c文件如下所示：

```

/*
 * Professional Linux Programming - Hello World
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    <<<<<< TREE
        printf("Bazaar1!\n");
    =====
        printf("Bazaar2!\n");
    >>>>> MERGE-SOURCE

    return 0;
}

```

你的任务就是确定哪个修改需要保留，并清理hello.c文件和删除（或更名）hello.c.BASE和hello.c.OTHER文件，否则Bazaar将不允许你执行合并，而且许多其他管理命令在冲突存在时也会暂时不能使用。

4.4.2 Linux 内核 SCM

项目网站：www.kernel.org/pub/software/scm/git。

在过去的几年中，Linux内核开发一直使用的是一个私有SCM工具BitKeeper。这个工具被免费提供给内核开发者社区，它被视为特别负责了近年来开发者所完成的许多成果。通过使用分散式SCM工具BitKeeper，开发者可以快速有效地使用极其大量的修改集（许多修改集会同时包含数十个文件）而不用担心其他SCM解决方案将会面对的性能问题。

Linux内核是一个极其庞大和活跃的软件项目，它有数以百计的捐助者，此外还有更多的测试者和第三方在为它工作。在写作本书的时候，Linux内核任何一个一个月的总修改集大小已达到30MB左右，这已超过Linux内核在早期经过许多年开发后的总容量，而现在它却只占一个月的开发量。毫无疑问你可以想象得到，有许多工作都投入到内核开发的周期中。

由于各种复杂的原因，对私有的BitKeeper解决方案的使用在2005年4月突然终止了，这使得内核开发者不得不迫切地寻找它的替代者。由于没有一个SCM可以让所有的开发者（主要是Linus Torvalds，Linux内核的创造者）都满意，所以Linus Torvalds决定消失几天去编写自己的SCM。不到一个星期git就诞生了。git是一个非常快速的基于对象的SCM，它可以用于管理大型项目（如Linux内核）并且效率非常高。现在已有越来越多的非内核项目也在使用它。

git的发展速度非常惊人。在2005年4月初，还没有人听说过git，然而在短短几个月内，它已被广泛使用并成为管理内核版本的“官方”工具。在git的最初版本出现大概几个星期后，就产生了各种对应的其他工具。首先出现的是“易于使用的”前端，然后是图形化版本库浏览器，最后是今天能得到的一整套工具。

因为git主要是用来给内核开发者使用的，所以如果你要使用它，也许就是为了这个目的。本节对涉及从头创建一个版本库的过程做了简要的说明，但当你使用Linux内核时，在大多数时候，你将通

过拷贝一个现有的提供给公众的git源代码树来对内核源代码进行修改。

在写作本书的时候，git还没有成为大多数发行版的标准安装包，但它可能很快就将成为发行版中的一个标准功能。在这段时间内，你可以通过其网站获得最新版本的git，它同时以源代码和各种预包装软件包的形式提供。

1. 一切都是对象

对git来说，一切都是对象。因为git确实是一个非常快速的基于对象的文件系统，而这正好也提供了一定层次的软件配置管理。当Linus最初创建git的时候，他对如何解决快速地在不同地点之间克隆内核树或取出一个特定修改集中所有的修改而不用为了完成这一过程等待整整一个下午很感兴趣。这也是为什么现有的SCM不适合于管理Linux内核的一个主要原因——它们对于内核开发者来说都太慢了。

当一个文件被签入git版本库时，它被存储为它的当前内容的一个SHA1哈希值。git在你的项目中的.git目录下使用了一个非常有效的分类目录层次，所有对象都位于.git/objects目录中。每次将文件的修改签入版本库时，git都会将这个文件的一份新拷贝保存为一个全新的对象——旧的对象被丢弃。这是一个有意的设计决策，其目的是为加快开发者的速度，但同时对于每一个签入版本库的修改集来说，它比其他常见的SCM解决方案要多消耗（少量的）一些磁盘空间。

2. git与Cogito

git包含各种不同的形式。标准的git发行版由100多个简单的工具组成，它们共同向开发者提供必要的功能。你可能从一开始接触git就觉得它很好用，但对于那些不太习惯的开发者来说，他们可以使用各种git的前端工具，这些工具旨在为开发者提供更舒适的开发体验。在写作本书的时候，这些工具中最受欢迎的是Cogito。在随后的例子中，你可以通过阅读该工具的在线文档来轻松地将git命令替换为适当的Cogito命令。

3. 创建新的git版本库

你可以通过在现有代码的基础上创建自己的版本库来开始学习git的使用。为了测试git SCM，我们再次使用第2章中的示例源代码。这次，将该目录放置到一个新的位置，然后通过这些源代码创建一个新的git版本库：

```
$ cd toolchains
$ git-init-db
defaulting to local storage area
$ git-add .
$ git commit
```

在执行commit命令之后将要求你输入一段说明文字。在新打开的编辑器窗口中，你可以看到在这次提交操作中将会被改变的文件列表。在本例中，git告诉你版本库中的每一个文件都将受到影响，这是因为版本库刚刚被创建。在你输入一段提交信息之后，git将确认这次提交。

```
Committing initial tree 370eeb1d55789681888d29c4bb85c21ef6cd6317H
```

git创建了一个新对象370eeb1d55789681888d29c4bb85c21ef6cd6317H，它代表源代码树被创建时的初始状态。此外，这些动作还导致git创建.git目录并填充其内容以及其内部对象数据库的内容。

4. 克隆已有的git版本库

大多数时候，在使用git（如果你一直都在使用git）时使用的都是别人提供的git版本库。这可能是

一个特定的开发者源代码树，但你更有可能使用的是它的一个本地拷贝。例如，官方的Linux内核源代码是通过git管理的。你将对它做出一些修改，然后将这些修改发送回（或要求他们从你这里引入修改）上游的项目社区，由他们来处理内核的发布。

为了获得一份最新Linux内核的本地拷贝以便进行git试验，你可以创建一个新目录，然后使用git pull命令引入官方Linux内核的git树。

```
$ mkdir linux-2.6
$ cd linux-2.6
$ git pull rsync://rsync.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

你现在在目录linux-2.6中有了一个完整的Linux 2.6内核的拷贝。

你还可以使用git clone命令克隆你的本地git树以创建另一份拷贝：

```
$ git clone toolchains toolchains1
defaulting to local storage area
Packing 38 objects
```

如果你想对一套源代码进行许多不同的试验，同时又希望能将这些修改分开，那么这个命令将特别有用。

5. 通过git管理源文件

如果你在本地git版本库中修改了一个文件，并希望将这个修改签回git（以便它可以管理你的本地git版本库树），可以使用命令git-update-index和git-commit：

```
$ git-update-index src/hello.c
$ git-commit
```

要想看到针对一个特定版本库的所有提交列表，可以使用git-whatchanged命令：

```
diff-tree 78edf00... (from c2d4eb6...)
Author: Jon Masters <jcm@perihelion.(none)>
Date:   Sun Feb 12 11:18:43 2006 +0000

modified hello

:100644 100644 b45f4f6... d1378a6... M  src/hello.c

diff-tree c2d4eb6... (from f77bb2b...)
Author: Jon Masters <jcm@perihelion.(none)>
Date:   Sun Feb 12 11:10:14 2006 +0000

foo

:100644 100644 3c13431... b45f4f6... M  src/hello.c
```

6. git 对调试的帮助

SCM工具不仅仅用于一些常见的用途，尤其当你调试一个和Linux内核一样庞大的项目时更是如此。人们通常会在遇到问题时发送一个错误报告给LKML（Linux内核邮件列表），描述在某种特定配置下的某一特定版本的Linux内核当运行在一个特定机器的特定环境中时会失败（其实许多错误报告远远没有这么详细——但内核开发者一直期望能接收到这样书写的错误报告）。

当一个特定的Linux内核版本被认为“有问题”时，我们通常需要开始一个调试过程以发现报告错误的用户可以正常使用的最后一个内核是哪一个版本。在最后一个可以正常工作的内核和有问题的内核之间通常还会存在多个版本。因此，调试过程将使用二分搜索算法——尝试各种内核，直到发现引入有问题修改集的内核版本为止。

由于这个二分搜索的调试方法是如此常用，因此git专门包含了一个特定的功能来缩短这一过程花费的时间并减轻内核开发者的痛苦。git bisect命令允许你告诉git已知的好版本和坏版本。首先：

```
$ git bisect start
```

然后告诉git坏内核和好内核的版本：

```
$ git bisect bad v2.6.16
```

```
$ git bisect good v2.6.15
```

git将发现刚好介于2.6.15和2.6.16中间的提交，并在本地目录中放置对应的内核源代码版本，然后你可以编译、开机测试并使用git bisect good或git bisect bad命令将它标记为好版本或坏版本。这一过程将不胜其烦地持续进行直至git最终发现引入这一错误行为的确切补丁为止。

4.5 集成化SCM工具

现在已有各种工具试图将你在本章学习到的不同的SCM功能整合到一起。集成化工具通常是理想的，因为它们消除了在开发软件时必须明确调用SCM工具的负担，从而解放了你的时间，使得你可以集中精力处理那些真正需要关注的问题——例如像开发代码等事情。

Eclipse

那些基于强大的开放源码Eclipse框架的图形化工具提供了一个极好的机会将本章中描述的所有SCM功能整合到一个可视化的IDE工具中。你已知道Eclipse的功能有多强大，我们要感谢人们为它编写的各种各样的模块，而且作为一个开发者，如果你需要某个功能，那很可能已有人至少试图为它创建一个eclipse模块了。这对SCM工具也是完全适用的，就像语言支持和其他功能一样。

Eclipse为CVS版本库提供了现成的支持，此外还有一些第三方的插件支持Subversion（称为Subclipse）和其他各种商业SCM工具。在写作本书的时候，GNU Arch和Linux内核SCM工具git的插件也正在开发中。无论你决定标准化哪一个SCM，它都很可能已有一个对应的Eclipse插件。

CVS版本库视图

Eclipse的CVS repository perspective（CVS版本库视图）可以通过perspectives菜单找到。一旦加载，eclipase将在左边的Repositories窗格中显示已知CVS版本库的列表，它最初是空的。要添加一个新的CVS版本库，请在CVS Repositories窗格上点击鼠标右键，选择“New”，然后选择“Repository Location”（版本库位置）。

此时将出现一个与图4-1类似的对话框。

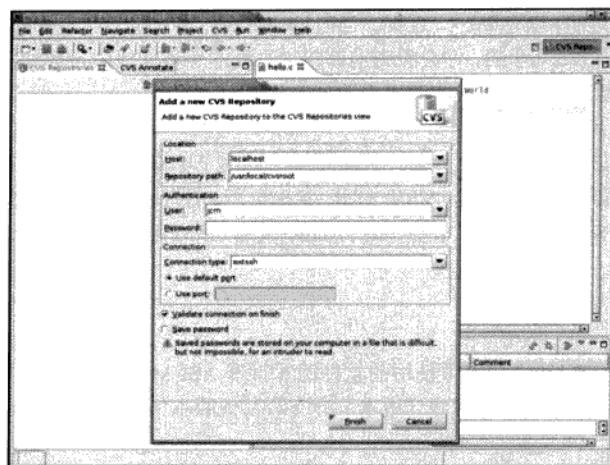


图4-1 加载后的CVS版本库视图

你应该在对话框中为你的CVS服务器输入适当的信息，然后点击“Finish”将它添加为一个新的CVS版本库，该版本库随后就将显示在Eclipse的“CVS Repositories”窗格中。

- 将一个项目置于CVS控制之下

CVS版本库的一个典型应用就是将一个现有的Eclipse项目置于CVS的控制之下。你可以按照以下步骤来完成这一工作。首先切换回C/C++ perspective（或你先前选择的任何一个语言视图），然后右键点击左边窗格中的项目名称。选择“Team”，然后选择“Share Project”。此时会自动加载一个向导，它列出了先前配置的CVS版本库。你从列表中选择一个正确的CVS版本库并完成向导。

此时将出现一个类似于图4-2的对话框。

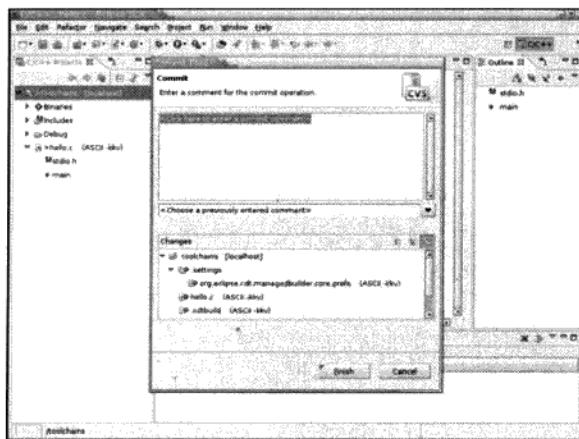


图4-2 将现有的Eclipse项目置于CVS控制之下

提交项目到CVS版本库以完成这一过程。当Eclipse主窗口再次出现时，你将看到项目和以前完全一样，但它现在是通过CVS共享的。你会看到当前CVS的版本号和其他的元数据显示在文件名的右边，所以我们可以很容易地跟踪项目的现状而不用四处查找。

- 控制CVS版本库

当右键点击Eclipse项目中的源文件时，你将在弹出的菜单中看到一组新的条目。通过这个菜单，你现在可以方便地将文件修改提交给CVS版本库、签出更新、处理标记和分支。事实上，你可以在CVS视图中执行所有的CVS命令。要想了解更多这方面的信息，请查看Eclipse的在线文档和你的Eclipse版本的手册。

4.6 本章总结

本章介绍了几个可在Linux系统中使用的软件配置管理（SCM）工具，它们可用于你自己的软件项目中。你学习了管理源代码和处理修订版本、分支和其他版本信息的过程，同时也发现了其中的一些工具提供的灵活性。

SCM工具可以被分为专门针对分散式软件开发的工具和不属于该类型的工具。你学习了应该根据不同项目的特定需求选择不同的工具。Linux内核开发者对现有的SCM工具都不满意，因此他们决定编写自己的SCM。

编写自己的SCM是非常极端的一步，但其中所包含的观点还是一致的——每个人都有他自己的个人喜好和对开发过程中所用软件和工具的不同倾向。一些人喜欢较简单的工具，而另外一些人则喜欢通过Eclipse插件使用一个图形化的前端。你应该在自己的项目中使用哪一个SCM工具最终还是由你来决定的。

第5章

网 络 编 程

如今，使你的Linux工作站或服务器联网是非常普通的。对应用程序来说，使它具备网络能力几乎已成为绝对必要的事情。知道如何在网络设备之间传递数据已成为一个专业Linux程序员所必须掌握的一项重要技能。

本章将介绍如何使用两种不同的方法为Linux程序增加网络功能。第一种方法介绍了如何执行原始套接字编程^①，它直接调用Linux网络子系统。使用这个方法，你可以编写任何通信协议以在任何网络设备之间进行通信。

第二种方法介绍了如何在应用程序中使用预包装的网络编程函数库。预包装的网络函数库向你提供已准备好的网络编程函数。你只需要调用库函数，通过常用的网络协议与网络设备进行交互即可。

5.1 Linux 套接字编程

UNIX操作系统带来了许多新的特征，它们改变了编程世界中的许多概念。其中一个特征就是文件描述符。一个文件描述符提供了一个文件对象的编程接口。因为在UNIX系统中几乎每一个对象都被定义为文件，所以一个文件描述符可以被用于在UNIX系统中的许多不同对象之间发送和接收数据。这使得UNIX（和现在的Linux）程序员的生活变得轻松了很多。无论你试图访问的是哪一种类型的设备，其使用的编程模型都是一样的。

从4.2BSD UNIX版本开始，网络访问也被定义为使用文件描述符。通过网络向一个远程设备发送数据就如同发送数据给一个本地文件一样简单。Linux利用网络文件描述符以允许程序访问网络子系统。本节将介绍如何利用Linux网络编程的不同特性。

5.1.1 套接字

在Linux网络编程中，你并不需要直接访问网络接口设备来发送和接收数据包。相反，我们通过创建一个中间的文件描述符来处理网络编程接口。那些诸如数据应该从哪个网络接口设备发送出去以及它应该如何发送的细节都交给Linux操作系统来处理。

用于指代网络连接的特殊文件描述符被称为套接字(socket)。套接字定义了一个特定的通信域(如网络连接或Linux进程间的通信(IPC)管道)、一个特定的通信类型(如stream或datagram)和一个特定的协议(如TCP或UDP)。一旦套接字被创建，必须将该套接字绑定到一个特定的网络地址和端口(对

^① 这里所说的原始套接字编程指的并不是使用SOCK_RAW方式(见表5-2中的解释)进行网络编程，而指的是直接使用套接字的系统调用进行网络编程。——译者注

于服务器应用程序)或一个远程的网络地址和端口(对于客户端应用程序)。在套接字被绑定之后,它就可以被用于通过网络发送和接收数据了。图5-1是一个通过套接字进行网络通信的例子。

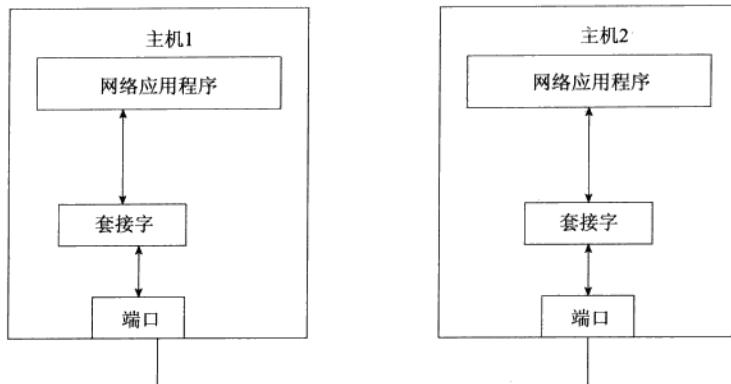


图5-1 通过套接字进行网络通信

套接字扮演的是系统中应用程序和网络接口之间的中间人的角色,所有的网络数据都要通过套接字传递。Linux提供了C语言函数socket来创建新的套接字:

```
int socket(int domain, int type, int protocol)
```

socket()函数返回一个套接字描述符,该描述符可以用于通过网络发送和接收数据(后面还会提到)。用于创建套接字的三个参数分别定义了通信域、类型和使用的协议。可以使用的domain值如表5-1所示:

表5-1 可用于创建套接字的常见通信域

值	说 明
PF_UNIX	UNIX IPC通信
PF_INET	IPv4因特网协议
PF_INET6	IPv6因特网协议
PF_IPX	Novell协议
PF_NETLINK	内核用户接口设备
PF_X25	ITU-T X.25/ISO-8208协议
PF_AX25	业余无线电AX.25协议
PF_ATMPVC	访问原始ATM PVC
PF_APPLETALK	Appletalk协议
PF_PACKET	底层数据包接口

对于IP通信,我们应使用PF_INET。参数type定义了在指定通信域中传输数据包所使用的网络通信类型。可以使用的type值如表5-2所示:

表5-2 常见的网络通信域类型

值	说 明
SOCK_STREAM	使用面向连接通信的数据包
SOCK_DGRAM	使用无连接通信的数据包
SOCK_SEQPACKET	使用面向连接的固定最大长度的数据包
SOCK_RAW	使用原始IP数据包
SOCK_RDM	使用一个可靠的数据包层，但不保证数据包按序到达

对于IP通信来说，最常用的两个*type*值是：SOCK_STREAM（用于面向连接通信）和SOCK_DGRAM（用于无连接通信）。

用于创建套接字的参数*protocol*的值取决于你所选择的*type*值。大多数*type*值（如SOCK_STREAM和SOCK_DGRAM）只能使用其默认的*protocol*值（例如，针对SOCK_STREAM的TCP和针对SOCK_DGRAM的UDP）。要指定默认的*protocol*，你可以将参数*protocol*设置为零而不需要设置一个正常的*protocol*值。

使用这些准则，在Linux中创建一个套接字用于网络通信是相当简单的：

```
int newsocket;
newsocket = socket(PF_INET, SOCK_STREAM, 0);
```

这个例子创建了一个标准的TCP套接字，它使用面向连接的字节流来传输数据给远程主机。但创建套接字本身并没有定义套接字将连接到哪个主机，这方面的内容我们将在后面进行介绍。

一旦套接字被创建，你就可以使用socket函数返回的文件描述符来引用该套接字，在前面的例子中，这个文件描述符就是变量newsocket。

5.1.2 网络地址

当套接字被创建之后，它必须被绑定到一个网络地址/端口组合。Linux套接字系统使用IP地址和TCP或UDP端口的方式是网络编程中最令人困惑的部分之一。它使用一个特殊的C语言结构来指定地址信息。

sockaddr结构包含两个成员：

- sa_family：地址族（定义为short类型）
- sa_data：设备的地址（定义为14个字节长）

sa_data成员的设计目的是允许sockaddr结构引用许多不同类型的网络地址。因为这个原因，所以这个14字节长的地址难以被直接使用。相反，Linux提供了一个专用于IP的地址结构sockaddr_in，它包含以下这些成员：

- sin_family：地址族（定义为short类型）
- sin_port：端口号（定义为short类型）
- sin_addr：地址（定义为long类型的IP地址，4个字节长）
- sin_data：8字节的填充

对IP通信来说，sin_family成员应该被设置为AF_INET，以表明它使用的是因特网地址族。sin_port和sin_addr成员的值可能与你期望的稍有不同。

在Linux系统中引用地址和端口的最大问题之一就是字节序。由于不同的系统以不同的顺序表示多个字节（高比特优先或低比特优先），因此Linux提供了相应的函数调用以确保地址和端口的值保持

一致的格式。`htons()`函数将`short`类型的值从主机字节序转换为网络字节序。因为端口号的值就是`short`类型的，所以这个函数可用于为`sin_port`成员转换端口号的值。

有几种不同的方法可以将IP地址转换为网络字节序。最简单的方法是使用inet_addr()函数。这个函数将一个以点分十进制表示的IP地址字符串转换为long类型的网络字节序。如果IP地址以域名的形式表示，那么你就必须先使用gethostbyname()函数来获得与该域名对应的IP地址。

通过使用这些函数，用于获得一个主机的IP地址/端口组合的代码如下所示：

```
struct sockaddr_in myconnection;
memset(&myconnection, 0, sizeof(myconnection));
myconnection.sin_family = AF_INET;
myconnection.sin_port = htons(8000);
myconnection.sin_addr.s_addr = inet_addr("192.168.1.1");
```

创建`sockaddr_in`变量`myconnection`之后，调用`memset()`以确保该变量中的所有成员都为0是一个好习惯。在这之后，对每一个成员进行定义。请注意成员`sin_addr`也使用了它的一个成员来定义地址。成员`s_addr`用于表示IP地址。我们使用`inet_addr()`函数将字符串形式的IP数字地址转换为`s_addr`所使用的IP地址结构。

现在你知道了应该如何定义IP地址/端口组合，你可以将套接字绑定到一个IP地址并开始传递数据了。套接字接口将根据套接字是面向连接的还是无连接的而使用不同的函数调用。以下各节将介绍如何使用这两种方法。

5.1.3 使用面向连接的套接字

对于面向连接的套接字（使用SOCK_STREAM类型）来说，它使用TCP协议来建立两个IP地址端点之间的会话（连接）。TCP协议保证两个端点之间的数据的交付（除非出现网络故障）。在连接的建立过程中，双方将有几次数据包交换的开销，但一旦连接建立，数据就可以可靠地在设备之间进行传输。

要创建一个面向连接的套接字，在服务器程序和客户端程序中必须要分别调用一系列的函数。图 5-2 显示了在这两类程序中分别要调用的函数序列。

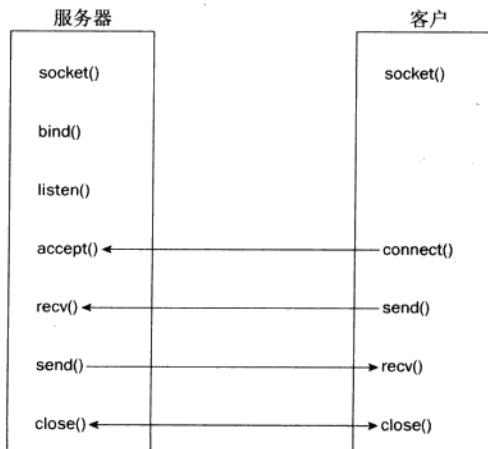


图5-2 创建面向连接的套接字时在服务器程序和客户端程序必须调用的函数序列

以下几节详细说明了服务器程序和客户端程序的不同之处。

1. 服务器函数

对于服务器程序来说，创建的套接字必须被绑定到一个本地IP地址和端口号以用于TCP通信。Linux的bind()函数就是用来完成该工作的。

```
int bind(int socket, sockaddr *addr, int length);
```

参数socket使用了socket()函数的返回值。参数addr使用sockaddr类型的地址/端口组合来定义本地网络连接。由于服务器通常只接受对其自身IP地址的连接，所以这里使用的是本地设备的IP地址和分配给应用程序的TCP端口号。如果你不知道本地系统的IP地址，你可以使用INADDR_ANY值以允许套接字绑定到系统中的所有本地地址上。参数length定义了sockaddr结构的长度。bind函数的一个实例如下所示：

```
struct sockaddr_in myaddr;
bind(s, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
```

bind()函数在失败时返回-1，成功时返回0，这可以通过标准的if语句来检测。

在将套接字绑定到一个地址和端口号之后，服务器程序必须准备好接受来自远程客户端的连接请求。这是一个分为两步的过程。

首先，程序必须调用listen()以开始监听客户端的连接，然后调用accept()来真正地接受客户端的连接。listen()函数的原型如下：

```
int listen(int socket, int backlog);
```

正如预期的那样，参数socket使用的是socket()函数创建的套接字描述符。参数backlog设置的是系统可接受的等待处理的连接队列长度。如果这个值被设置为2，系统在指定的端口就可以同时接受两个独立客户端的连接请求。其中一个连接请求将被立刻处理，而另一个将被放置到连接队列中等待第一个连接请求处理完成。如果此时第三个连接请求到达，系统将拒绝该连接请求，因为连接队列的长度已达到backlog设置的值^①。

在listen()函数调用之后，accept()函数必须被调用以等待进入的连接。accept()函数是一个阻塞式函数^②。程序的执行将暂停在accept()函数调用上直到有一个客户端的连接请求到达。accept()函数的原型如下：

```
int accept(int socket, sockaddr *from, int *fromlen);
```

现在，你应该知道参数socket的含义了。参数from和fromlen分别指向sockaddr地址结构和其长度。客户端的远程地址信息就保存在这个结构中，所以你可以在必要的时候访问它。

当一个连接请求被接受后，accept()函数将返回一个新的套接字描述符。这个新的套接字被用于与远程客户通信。由socket()函数创建的原来的套接字仍然可以被用来继续监听更多的客户端连接。

^① 对于参数backlog的解释原书并不十分精确，实际上当第一个连接请求被处理时，它已从连接队列中删除，所以第三个连接请求是可以接受的。而且Linux内核维护的实际上是两个连接队列：未完成连接队列和已完成连接队列。关于backlog的更详细的解释请见《UNIX网络编程 卷1》的第4章。

^② 也被称为慢系统调用。——译者注

一旦一个连接请求被接受，服务器就可以使用新的套接字描述符向客户端发送数据或接收来自客户端的数据，`send()`和`recv()`函数的原型如下：

```
int send(int socket, const void *message, int length, int flags)
int recv(int socket, void *message, int length, int flags)
```

参数`socket`再次使用了`accept()`函数为这次连接返回的新打开的套接字。参数`message`要么指向包含发送数据的缓冲区，要么指向准备接收数据的空缓冲区。参数`length`指出缓冲区的大小，而参数`flags`指出是否需要使用某些特殊的标记（例如你是否想在TCP数据包中将数据标记为紧急数据）。对于正常的TCP通信来说，参数`flags`应设置为0。

`send()`函数不会阻塞程序的执行^①。缓冲区中的数据被发送到系统中底层的TCP发送缓冲区，然后函数调用将返回。有可能在`send()`函数中定义的缓冲区中的数据不会全部被发送出去。`send()`函数调用返回的整数值将表示有多少字节的数据已被发送到TCP发送缓冲区。确认这个返回值匹配缓冲区的大小以确保所有的数据都已被发送出去是非常重要的。

`recv()`函数是一个阻塞式函数。程序的执行将暂停直到`recv()`函数从远程客户端接收到数据或者远程客户端明确地断开TCP会话。如果客户端断开TCP会话，`recv()`函数将返回0。如果客户端发送一个数据包，`recv()`函数将接收到的数据放入定义的缓冲区并返回接收到的字节数。

在设计客户端和服务器应用程序时，同步发送和接收函数是非常重要的。如果服务器和客户端同时等待在`recv()`函数调用上，它们将产生死锁，并且不会有通信发生。

下面是一个典型的服务器示例。代码清单5-1中的程序`tcpserver.c`演示了一个简单的服务器，它在TCP端口8000上接受客户端的连接并向客户端返回它接收到的任何数据。

代码清单5-1 一个简单的TCP服务器

```
/*
 * Professional Linux Programming - Simple TCP server
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(argc, argv)
{
    int s, fd, len;
    struct sockaddr_in my_addr;
    struct sockaddr_in remote_addr;
    int sin_size;
    char buf[BUFSIZ];

    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_addr.s_addr = INADDR_ANY;
```

^① 当套接字是阻塞的并且套接字的TCP发送缓冲区容不下应用程序缓冲区中的所有数据时，`send()`函数将阻塞。

——译者注

```

my_addr.sin_port = htons(8000);

if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("socket");
    return 1;
}

if (bind(s, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) < 0)
{
    perror("bind");
    return 1;
}

listen(s, 5);
sin_size = sizeof(struct sockaddr_in);
if ((fd = accept(s, (struct sockaddr *)&remote_addr, &sin_size)) < 0)
{
    perror("accept");
    return 1;
}
printf("accepted client %s\n", inet_ntoa(remote_addr.sin_addr));
len = send(fd, "Welcome to my server\n", 21, 0);
while ((len = recv(fd, buf, BUFSIZ, 0)) > 0)
{
    buf[len] = '\0';
    printf("%s\n", buf);
    if (send(fd, buf, len, 0) < 0)
    {
        perror("write");
        return 1;
    }
}
close(fd);
close(s);
return 0;
}

```

需要注意的是，程序tcpserver.c使用了几个头文件。这些头文件定义了用于处理网络连接的函数和数据类型，它们必须包含在所有的网络程序中。

程序tcpserver.c首先使用socket()函数创建了一个套接字，然后将套接字绑定到主机所有本地地址的TCP端口8000。接着listen()函数被调用以监听新的连接请求，在拒绝新的连接请求之前最多允许有5个连接请求排在队列中等待处理。

当listen()函数调用之后，服务器程序调用accept()函数等待客户端连接请求的到达。当一个连接请求到达后，accept()函数将返回一个新的套接字描述符以代表这个新的远程客户端连接。原来的套接字描述符仍然被用于接受其他客户端的连接请求。

然后新的客户端连接将被用于发送和接收数据。服务器首先发送一个欢迎信息给客户。接着，服务器进入一个while循环以等待来自客户端的数据。如果接收到数据，它将在主控台上显示这些数据并将这些数据返回给客户端。请注意，当接收到数据时，recv()函数返回的缓冲区长度被用于在缓冲区中放置一个NULL结束符。这确保了每个接收到的字符串都被正确地终止了。

当远程客户端关闭这个连接时，`recv()`函数将返回0，它用于退出while循环。

编译程序`tcpserver.c`并不需要使用一些特殊的函数库，你只需要像编译其他C语言程序一样编译这个程序，然后运行它。

```
$ cc -o tcpserver tcpserver.c
$ ./tcpserver
```

`tcpserver`程序将启动，然后等待客户端的连接。为了测试这个服务器，我们在Linux系统上（或在网络中的另一个Linux系统上）启动另一个会话，然后telnet到服务器的8000端口。如果你是在同一个系统上进行连接，你可以使用特殊的`localhost`网络地址来指向本机。

```
$ telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^>'.
Welcome to my server
test
test
This is another test.
This is another test.
goodbye.
goodbye.
^]
telnet> quit
Connection closed.
$
```

服务器接受了这个连接，并发送了它的欢迎信息，这个信息显示在`telnet`程序中。在这之后，服务器返回你输入的任何文本。为了结束这个会话，我们输入`Ctrl-[`，然后输入`quit`。客户将终止这个TCP会话，服务器程序也终止了（因为`recv()`函数返回0）。

2. 客户端函数

在面向连接的应用程序中，客户端必须连接到一个特定的主机地址和端口。如果你知道远程服务器的IP地址，可以使用`inet_addr()`函数将它转换为正确的格式：

```
remote_addr.sin_family = AF_INET;
remote_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
remote_addr.sin_port = htons(8000);
```

在客户端程序中，你必须使用`socket()`函数来建立正确的通信套接字，但在套接字创建之后，我们使用`connect()`函数来代替`bind()`函数。`connect()`函数将使用你提供的`sockaddr`地址/端口号组合与远程网络设备建立连接。

```
int connect(int socket, sockaddr *addr, int addrlen);
```

参数`socket`再次使用`socket()`函数返回的值，而参数`addr`指向一个`sockaddr`结构，在该结构中包含了你想建立连接的远程网络设备的IP地址和TCP端口号。

如果`connect()`函数失败，它将返回-1。如果它成功了，则客户端已连接到服务器，客户端就可以在套接字描述符上使用标准的`send()`函数和`recv()`函数与服务器传输数据了。

显示在清单5-2中的程序`tcpclient.c`演示了一个简单的客户端应用程序。

代码清单5-2 一个简单的TCP客户端

```

/*
 *
 * Professional Linux Programming - Simple TCP client
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(argc, argv)
{
    int s, len;
    struct sockaddr_in remote_addr;
    char Buf[BUFSIZ];

    memset(&remote_addr, 0, sizeof(remote_addr));
    remote_addr.sin_family = AF_INET;
    remote_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    remote_addr.sin_port = htons(8000);

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket");
        return 1;
    }

    if (connect(s, (struct sockaddr *)&remote_addr, sizeof(struct sockaddr)) < 0)
    {
        perror("connect");
        return 1;
    }

    printf("connected to server\n");
    len = recv(s, buf, BUFSIZ, 0);
    buf[len] = '\0';
    printf("%s", buf);
    while(1)
    {
        printf("Enter string to send: ");
        scanf("%s", buf);
        if (!strcmp(buf, "quit"))
            break;
        len = send(s, buf, strlen(buf), 0);
        len = recv(s, buf, BUFSIZ, 0);
        buf[len] = '\0';
        printf(" received: %s\n", buf);
    }

    close(s);
    return 0;
}

```

程序tcpclient.c是设计用来和前面的程序tcpserver.c通信的。它创建了一个套接字并试图和本机的TCP端口8000建立连接。如果连接建立成功，将显示一个消息，然后调用recv()函数接收服务器发送过来的欢迎信息。当通过网络连接接收字符串时，不要忘记给该字符串附加一个NULL终止符：

```
len = recv(s, buf, BUFSIZ, 0);
buf[len] = '\0';
printf("%s", buf);
```

在接收到欢迎信息之后，客户程序进入一个while循环，要求用户输入要发送的字符串，然后调用send()函数将该字符串发送给服务器。scanf()函数对于产品级的程序并不是一个好的选择，因为它容易遭受到缓冲区溢出的攻击，但对于这次测试来说，使用它就够了。

因为服务器将返回它接收到的任何数据，所以客户将使用recv()函数来接收服务器返回的字符串。我们在使用printf()函数打印字符串之前再次使用NULL字符终止了它。

当客户端程序发现用户输入了字符串quit，它将关闭客户端与服务器的连接，这同时也迫使服务器程序终止。

3. 关闭连接

在程序tcpserver.c和tcpclient.c中，你可能已注意到我们在程序的结尾使用标准的close()函数调用关闭了创建的套接字。虽然这完全可以接受，但套接字还有另外一个函数可以更精确地完成这一工作。

套接字描述符可以使用shutdown()函数来指定通信会话终止的方式。该函数的原型如下：

```
shutdown(int socket, int how)
```

shutdown()函数使用参数how来让你决定如何优雅地关闭连接。可使用的选项如下：

- 0——不再接收数据包。
- 1——不再发送数据包。
- 2——不再发送或接收数据包。

通过选择0或1，你可以禁止套接字接收或发送更多的数据，但允许套接字继续将剩余的数据发送完或继续接收剩余的数据。当连接处理完所有这些数据之后，我们可以调用close()函数终止连接而不会造成任何数据丢失。

5.1.4 使用无连接套接字

在面向连接的套接字编程中，客户端使用connect()函数与服务器建立TCP连接。无连接套接字并不会在网络设备之间创建一个专用的连接。相反，数据被“扔到网络中”，并希望它们能够自己到达指定的目的地。无连接套接字的好处是传输数据的开销很小，它不需要认真地跟踪每一个数据包，这极大地降低了处理数据包所需的开销。正因为如此，无连接套接字比面向连接套接字有更好的吞吐量^①。

无连接套接字使用套接字类型SOCK_DGRAM实现。它通常使用用户数据报协议（UDP）在网络设备之间传递数据。由于无连接套接字不需要建立连接，所以服务器和客户端程序看起来很相似。

要发送UDP消息，套接字不需要使用bind()函数。我们使用sendto()函数来定义数据和它要发送到的目的地：

^① 实际上，在涉及大量数据的传输时，面向连接套接字通常比无连接套接字有更好的吞吐量，因为TCP协议会将多个小数据包合并在一起发送。

```
int sendto(int socket, void *msg, int msglen, int flags,
          struct sockaddr *to, int tolen)
```

sendto() 函数使用socket()函数创建的套接字来定义参数socket，其他参数分别为要发送的消息、消息的长度、一些特殊标记、目标地址（使用sockaddr结构）和地址结构的长度。

要接收UDP消息，套接字必须使用bind()函数绑定到一个UDP端口以监听进入的数据包，这和TCP连接一样。在bind()函数调用之后，数据包可以使用recvfrom()函数来接收：

```
int recvfrom(int socket, void *msg, int msglen, int flags,
            struct sockaddr *from, int fromlen)
```

由于recvfrom()函数是无连接的，所以它可以接收来自任何网络设备的数据。发送主机的网络地址信息保存在结构from中，而接收到的消息保存在缓冲区msg中。

代码清单5-3中的程序udpserver.c演示了如何建立一个无连接的服务器程序。

代码清单5-3 一个简单的UDP服务器

```
/*
 *
 * Professional Linux Programming - Simple UDP server
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    int s, fd, len;
    struct sockaddr_in my_addr;
    struct sockaddr_in remote_addr;
    int sin_size;
    char buf[BUFSIZ];

    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_addr.s_addr = INADDR_ANY;
    my_addr.sin_port = htons(8000);

    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket");
        return 1;
    }

    if (bind(s, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) < 0)
    {
        perror("bind");
        return 1;
    }
```

```

    sin_size = sizeof(struct sockaddr_in);
    printf("waiting for a packet...\n");
    if ((len=recvfrom(s,buf,BUFSIZ,0,(struct sockaddr *)&remote_addr,
    &sin_size)) < 0)
    {
        perror("recvfrom");
        return 1;
    }

    printf("received packet from %s:\n",
           inet_ntoa(remote_addr.sin_addr));
    buf[len] = '\0';
    printf(" contents: %s\n", buf);

    close(s);
    return 0;
}

```

程序udpserver.c创建了一个套接字并将它绑定到UDP端口8000。接着，它使用recvfrom()函数等待进入的数据包。请注意，此时任何网络设备都可以发送数据给该服务器的UDP端口8000，因为套接字并没有连接到一个特定的主机。当数据包到达时，发送设备的网络地址被提取出来，服务器显示数据包中的数据，然后关闭套接字。

代码清单5-4中的程序udpclient.c演示了如何发送一个UDP数据包给远程网络设备。

代码清单5-4 一个简单的UDP客户端

```

/*
 *
 * Professional Linux Programming - Simple UDP client
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    int s, len;
    struct sockaddr_in remote_addr;
    char buf[BUFSIZ];

    memset(&remote_addr, 0, sizeof(remote_addr));
    remote_addr.sin_family = AF_INET;
    remote_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    remote_addr.sin_port = htons(8000);

    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket");
        return 1;
    }

```

```

    }

    strcpy(buf, "This is a test message");
    printf("sending: '%s'\n", buf);
    if ((len = sendto(s, buf, strlen(buf), 0, (struct sockaddr *)&remote_addr,
sizeof(struct sockaddr))) < 0)
    {
        perror("sendto");
        return 1;
    }

    close(s);
    return 0;
}

```

程序udpclient.c创建了一个UDP套接字，并使用sendto()函数发送消息给指定的主机。为了测试这些程序，我们在系统中启动程序udpserver，然后在同一个系统或另一个网络系统中运行程序udpclient。你将在服务器上看到发送的消息。

5.2 传输数据

虽然知道如何与远程设备建立一个套接字连接是很有帮助的，但它并不是网络编程的全部内容。在一个受控的环境中从一个主机发送简单的消息给另一台主机尽管令人印象深刻，但却不是非常有用。你很可能希望使用网络编程跨越大型网络（甚至可能是因特网）传递有意义的数据，但这是最需要慎重对待的情况。在网络编程中，有许多事情都可能会导致错误的结果。本节就将介绍其中的一些陷阱及其解决方法。

5.2.1 数据报与字节流

TCP和UDP通信之间最核心的差异是通过网络发送数据的方法。UDP协议以消息的形式发送数据，这个消息被称为数据报。每次使用sendto()函数发送一个数据报时，它作为一个单独的UDP数据包发送到网络。接收程序接收到这个UDP数据包并使用一次单独的recvfrom()函数调用来处理它。UDP并不能保证数据包将到达目的地，但一旦数据包到达了，它将保证由sendto()函数发送的数据将被recvfrom()函数调用完全接收（这里假设发送数据的长度满足标准UDP数据包的要求）。

但是，TCP通信并不是这样处理数据发送的。TCP套接字使用字节流技术而不是数据报技术。由send()函数发送的数据将被放入一个字节流中通过网络传播给接收主机。接收主机可以使用recv()函数读取字节流中的部分数据，但并没有办法控制读取字节流中哪一部分的数据。你不能保证通过一次recv()函数调用就可以读取由send()函数发送的所有数据。

解释这一原则最好的方法就是创建一个程序来测试它。代码清单5-5中的程序badserver.c是这个演示程序的服务器部分。

代码清单5-5 一个错误的TCP服务器

```

/*
 *
 * Professional Linux Programming - A bad TCP server
 *
 */

```

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(argc, argv)
{
    int i, s, fd, len;
    struct sockaddr_in my_addr;
    struct sockaddr_in remote_addr;
    int sin_size;
    char buf[BUFSIZ];

    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_addr.s_addr = INADDR_ANY;
    my_addr.sin_port = htons(8000);

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket");
        return 1;
    }

    if (bind(s, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) < 0)
    {
        perror("bind");
        return 1;
    }

    listen(s, 5);
    sin_size = sizeof(struct sockaddr_in);
    if ((fd = accept(s, (struct sockaddr *)&remote_addr, &sin_size)) < 0)
    {
        perror("accept");
        return 1;
    }
    printf("accepted client %s\n", inet_ntoa(remote_addr.sin_addr));
    len = send(fd, "Welcome to my server\n", 21, 0);
    for (i = 0; i < 5; i++)
    {
        len = recv(fd, buf, BUFSIZ, 0);
        buf[len] = '\0';
        printf("%s\n", buf);
    }

    close(fd);
    close(s);
    return 0;
}

```

现在你应该能够看懂程序badserver.c中的大部分内容了。它使用标准函数创建套接字、将套接字绑定到本地地址、监听连接并接受新的连接请求。在新的连接请求被接受之后，服务器发送欢迎信

息，然后进入循环等待接收来自客户端的5个消息。每一个消息以单独一行的形式显示在主控台上。当接收到这5个消息之后，连接被关闭。

代码清单5-6中的程序badclient.c是这个演示程序的客户端部分。

代码清单5-6 一个错误的TCP客户端

```
/*
 *
 * Professional Linux Programming - A bad TCP client
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(argc, argv)
{
    int i, s, fd, len;
    struct sockaddr_in remote_addr;
    int sin_size;
    char buf[BUFSIZ];

    memset(&remote_addr, 0, sizeof(remote_addr));
    remote_addr.sin_family = AF_INET;
    remote_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    remote_addr.sin_port = htons(8000);

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket");
        return 1;
    }

    if (connect(s, (struct sockaddr *)&remote_addr, sizeof(struct sockaddr)) < 0)
    {
        perror("connect");
        return 1;
    }

    printf("connected to server\n");
    len = recv(s, buf, BUFSIZ, 0);
    buf[len] = '\0';
    printf("%s", buf);

    len = send(s, "test message1", 13, 0);
    len = send(s, "test message2", 13, 0);
    len = send(s, "test message3", 13, 0);
    len = send(s, "test message4", 13, 0);
    len = send(s, "test message5", 13, 0);

    close(s);
    return 0;
}
```

程序badclient.c连接到程序badserver.c，接收欢迎信息，然后连续发送5个测试消息以匹配badserver期望接收到的5个消息。

编译这两个程序，如果有可能，就在你的网络中的两个不同的系统中运行它们（记住将程序badclient.c中的s_addr值修改为运行程序badserver的系统的IP地址）。

尝试在几个不同的时间运行这个程序，并注意badserver的输出。下面是在作者的系统中badserver的输出：

```
$ ./badserver
accepted client 10.0.1.37
test message1
test message2test message3test message4test message5
```

\$

发生了什么？很显然，并不是所有的消息都放在独立的数据包中。服务器通过第一个recv()函数调用收到客户端发送的第一个消息，这工作得很好，但接下来客户端发送的四个消息却被第二个recv()函数调用作为一个消息接收下来。剩下的三次recv()函数调用没有输出任何数据，因为远程客户端已关闭了这个连接。

这个示例生动地说明了TCP的一个基本原则，它常常被网络编程的初学者忽略。TCP中没有消息边界的概念。当消息使用send()函数发送并使用recv()函数接收时，它们不一定在发送和接收时使用相同的边界，因为TCP有处理数据的内部缓冲区。

在Linux系统中，由于TCP数据包可能会发生重传，所以应用程序发送给套接字的数据会在Linux系统内部缓存。所以，如果远程设备请求重传数据包，系统只需从套接字缓冲区中提取出数据包的数据并重传它即可，而不需要去打扰应用程序了。但这样做的缺点是套接字缓冲区经常会在所有数据发送出去之前就被填满了，这个过程如图5-3所示。

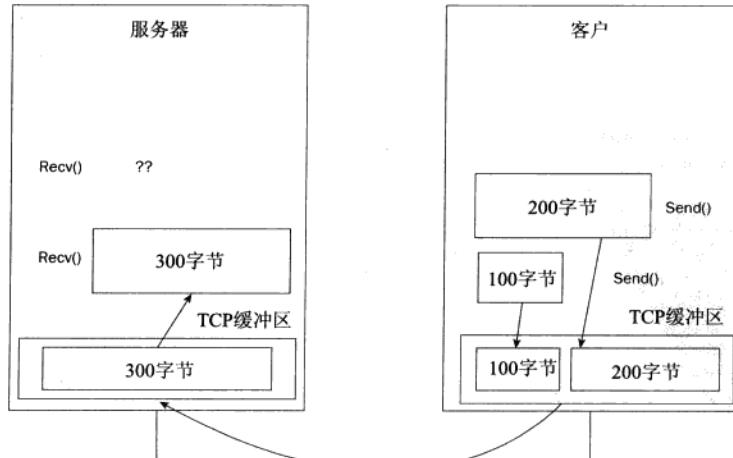


图5-3 当远程设备请求重传数据包时系统处理数据包的过程

两个send()函数将数据放到其内部套接字缓冲区中以等待被发送到远程设备。数据可能以两个数据包、一个数据包或很多数据包的方式发送到远程设备，我们无法控制底层的套接字进程处理缓存数据的方式。TCP可以保证的是所有发送的数据都将出现在远程设备上。

在远程设备一端，到达的数据包在套接字缓冲区中重组。当recv()函数被调用时，在缓冲区中的数据都将被发送给应用程序（根据recv()函数提供的缓冲区大小）。如果recv()函数指定的缓冲区大小足够接收套接字缓冲区中的所有数据，那么所有的数据都将发送给应用程序处理。因此，由两次send()函数调用发送的数据可以被一次recv()函数调用读取。

5.2.2 标记消息边界

由于TCP没有提供分隔消息的方式，所以必须在你的网络程序中自己提供它。有两种常用的方法用来界定网络字节流中的消息：

- 使用消息长度；
- 使用消息结束标记。

下面两小节将分别介绍这两种方法。

1. 使用消息长度

第一种方法利用消息长度来确定消息的边界。如果接收主机在消息到达之前已知道消息的长度，它就知道如何处理到达的数据了。在数据传输中，消息长度的使用有两种方法：

- 每个消息的长度都相同；
- 每个消息中都包含其长度的定义。

通过使得每个消息的长度都相同，你就可以准确地知道recv()函数应该从内部缓冲区中取走多少数据。

代码清单5-7中的程序betterserver.c通过限制recv()函数可以从内部缓冲区中取走的字节数来解决程序badserver.c的问题。

代码清单5-7 解决TCP服务器badserver的问题

```
/*
 *
 * Professional Linux Programming - A better TCP server
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(argc, argv)
{
    int i, s, fd, len;
    struct sockaddr_in my_addr;
    struct sockaddr_in remote_addr;
    int sin_size;
    char buf[BUFSIZ];
```

```

memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = INADDR_ANY;
my_addr.sin_port = htons(8000);

if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("socket");
    return 1;
}

if (bind(s, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) < 0)
{
    perror("bind");
    return 1;
}

listen(s, 5);
sin_size = sizeof(struct sockaddr_in);
if ((fd = accept(s, (struct sockaddr *)&remote_addr, &sin_size)) < 0)
{
    perror("accept");
    return 1;
}
printf("accepted client %s\n", inet_ntoa(remote_addr.sin_addr));
len = send(fd, "Welcome to my server\n", 21, 0);
for (i = 0; i < 5; i++)
{
    len = recv(fd, buf, 13, 0);
    buf[len] = '\0';
    printf("%s\n", buf);
}

close(fd);
close(s);
return 0;
}

```

请注意for循环中的recv()函数限制了其可以读取的字节数:

```
len = recv(fd, buf, 13, 0)
```

因为客户端发送的每个消息长度都是13个字节, 所以这就保证了每个recv()函数调用处理的数据量不会超过一个消息的长度。编译并运行程序betterserver, 当使用程序badclient发送数据时, 服务器端将产生以下输出:

```
$ ./betterserver
accepted client 10.0.1.37
test message1
test message2
test message3
test message4
test message5
$
```

无论运行这个程序多少次，它都将产生相同的结果^①。现在，服务器可以清楚地将每个消息分开了。有时候，在应用程序中总是发送长度相同的消息并不是很有效率。消息通常都是大小不一的。选择一个过大的通用消息长度会导致消息在发送到网络中时需要许多填充数据。

为了解决这个问题，程序员通常使用可变长消息并在发送实际消息给远程设备之前先发送消息的长度。接收主机必须首先读取消息长度，然后进入循环以接收数据，直到一个完整的消息被接收为止。一旦消息中的所有字节都被接收到了，程序就重置以接收另一个消息长度。

2. 使用消息标记

除了使用消息长度以外，第二种方法是使用消息结束标记来处理消息。结束标记是一个预先确定的字符（或一组字符），它用于界定一个消息的结尾。当接收程序接收数据时，它必须扫描到达的数据以找到结束标记。一旦发现结束标记，就将这个完整的消息传递给程序来处理。很显然，选择的消息结束标记不能出现在消息的数据部分。

对于使用ASCII字符集的许多系统来说，标准的消息结束标记是回车换行符。当主机接收到每个数据包时，它必须检查数据包中的每一个字节。如果它不是消息结束标记，就把它添加到临时缓冲区中。当发现消息结束标记时，临时缓冲区中的数据就被传输到永久缓冲区中以便使用。然后，临时缓冲区被清空以用于重组下一个消息。

5.3 使用网络编程函数库

编写原始套接字代码是开发网络应用程序的最健壮的方法，但它并不是唯一的方法。当你编写自己的客户端/服务器协议时，数据如何在网络设备之间进行传输都在你的控制之中。但是，在今天的网络世界中，已有许多设计好的网络协议用来处理任何类型的数据传输。与从头开始编写你的应用程序需要的每一个网络协议相反，你可以使用已设计好的网络协议在系统之间交换数据。每当你要使用这样的标准协议时，都很可能已有一个免费的网络编程函数库包含了实现该协议的代码。

这些网络编程函数库为许多常用的网络协议如FTP、HTTP和Telnet提供了应用程序编程接口（API）。你只需要使用指定的目标地址和数据调用一个函数就可以发送数据了。这使得你可以把注意力集中在应用程序代码上而不是网络编程的细节上，从而极大地减轻了你的负担。

本节说明了如何使用libCurl网络编程函数库。libCurl是一个较为流行的软件包，它提供了简单的FTP、HTTP和Telnet网络编程函数。

5.3.1 libCurl 函数库

Curl软件包是一个开放源码项目，它为流行的网络协议提供了命令行工具。你只需执行一个简单的命令行程序就可以和远程服务器进行网络通信。网络连接所需的所有参数（如主机名、用户ID和密码）都在命令行程序上定义。这些命令行工具可以在任何shell脚本程序以及Perl和Python程序中使用。Curl的命令行工具支持FTP、安全FTP（FTPS）、TFTP、HTTP、HTTPS、Telnet、DICT、FILE和LDAP网络协议。

libCurl是Curl的姊妹项目。libCurl是一个标准的函数库，它提供了用于实现Curl支持的所有网络协议的C语言API函数。这使得C和C++程序员可以轻松地在他们的应用程序中整合网络协议。

^① 当服务器接收到的字节数小于13个字节时，程序并不能产生预期的结果，正确的解决方法应该是循环读取客户端发送的数据直到接收到需要的字节数。

本节介绍了如何在Linux系统中下载并安装libCurl函数库。

1. 下载libCurl

Curl的网址是curl.haxx.se。它包含Curl及其姊妹项目libCurl的文档、教程和下载。Curl软件包以源代码和二进制两种形式提供下载。二进制软件包包含针对特定操作系统平台预编译的应用程序。如果你下载针对你的平台的二进制软件包，请确认它包含了libCurl函数库（在下载页面中有说明）。

如果你喜欢自己进行编译，可以下载最新的Curl源代码软件包。它包含libCurl函数库，并将在安装时自动编译它们。在写作本书的时候，最新的源代码软件包的版本是curl-7.15.5，它以多种压缩格式提供下载。

2. 安装libCurl

在下载了源代码软件包之后，必须先将源代码解压缩到一个工作目录中。如果下载的是.tar.gz文件，就可以使用tar命令来完成这一工作。

```
$ tar -zxvf curl-7.15.5.tar.gz
```

这个命令将自动解压缩软件包并将源代码释放到目录curl-7.15.5中。

Curl软件包利用常用的Linux工具集来编译可执行文件和库文件。configure程序用于配置需要的选项并确定系统的编译环境。进入主目录curl-7.15.5并执行configure命令：

```
$ ./configure
```

完成配置过程之后，可以使用标准的make命令来编译可执行文件和库文件。这个过程涉及三个步骤：编译、测试和安装。前两个步骤可以以普通用户的身份执行，但为了安装文件，你必须拥有超级用户特权：

```
$ make  
$ make test  
$ su  
Password:  
$ make install
```

在安装过程结束之后，libCurl库文件位于/usr/local/lib目录中。并不是所有的Linux发行版都会在查找库文件的默认目录列表中包含这个目录。请确认这个目录包含在文件/etc/ld.so.conf中。如果没有，将它添加到该文件中。

5.3.2 使用 libCurl 库

libCurl库的设计目的既灵活又易于使用。为满足这些需求，对每一个C语言函数都提供了两种格式：

- 简易的
- 多重的

简易格式的函数提供了一些简单的功能用于在网络设备之间建立同步连接。由于这些都是同步连接，所以同一时间只能有一个连接是活跃的。而多重格式的函数提供了一些更高级的功能用于实现更高层次的控制。多重格式的函数还允许在网络设备之间建立异步连接，它们允许你在一个线程中同时执行多个连接。

要编译libCurl应用程序，你必须在应用程序中包含libCurl头文件并将程序与libCurl库连接。libCurl头文件位于/usr/local/include/curl目录中。你可以在程序中使用相对于curl目录的相

对路径来包含头文件：

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

在编译程序时，你必须指定/usr/local/include目录以及curl库文件：

```
$ cc -I /usr/local/include -o test test.c -lcurl
```

这个命令将源文件test.c编译为可执行程序test。

下面几小节将介绍一些libCurl的简易格式的函数，并说明如何在应用程序中使用它们来连接到常用的网络资源。

1. 连接到网络资源

libCurl函数库提供了许多不同的函数来处理网络连接，其主要功能是通过一个CURL对象句柄来处理的。CURL对象用于跟踪连接、处理用于建立连接的参数以及提供连接返回的数据。

在调用任何函数之前，Curl函数库必须先使用curl_easy_init()函数进行初始化：

```
CURL *curl;
curl = curl_easy_init();
if (!curl)
{
    perror("curl");
    return 1;
}
```

如果初始化成功，CURL对象将包含一个新的句柄，它可用于与网络资源建立连接。这个连接的参数使用curl_easy_setopt()函数来定义。该函数的原型如下：

```
curl_easy_setopt(curl, opt, value)
```

参数curl就是已初始化的CURL对象。参数opt是要定义的连接选项，参数value是该选项的值。有大量的选项可以用来修改网络连接参数，一些较常见的选项列在表5-3中。

表5-3 用于修改网络连接参数的常见选项

选 项	说 明
CURLOPT_URL	指定要连接的URL
CURLOPT_USERPWD	指定用于连接的“用户名:密码”组合
CURLOPT_PROXY	指定用于连接的“代理服务器:端口”
CURLOPT_VERBOSE	指定是否启用详细(verbose)模式，启用(1)，关闭(0)
CURLOPT_HEADER	指定是否需要显示协议头信息，显示(1)
CURLOPT_WRITEFUNCTION	指定处理已接收数据的函数指针。否则，已接收的数据将显示在标准输出上
CURLOPT_WRITEDATA	传递给已定义写函数的数据指针
CURLOPT_READFUNCTION	指定处理待发送数据的函数指针
CURLOPT_READDATA	传递给已定义读函数的数据指针

在定义了连接参数之后，使用curl_easy_perform()函数开始连接。这个函数使用CURL对象指针作为其唯一的参数，并返回一个CURL code对象以表示其执行情况。

在使用完CURL对象之后，你最好使用curl_easy_cleanup()函数将它从内存中删除。

代码清单5-8中的程序getweb.c演示了一个基本的libCurl应用程序：

代码清单5-8 使用libCurl函数库

```
/*
 *
 * Professional Linux Programming - A simple libCurl program
 *
 */
#include <stdio.h>
#include <curl/curl.h>

int main(int argc, char *argv[])
{
    CURL *curl;
    CURLcode res;
    curl = curl_easy_init();
    if(!curl)
    {
        perror("curl");
        return 1;
    }
    curl_easy_setopt(curl, CURLOPT_URL, argv[1]);
    curl_easy_setopt(curl, CURLOPT_PROXY, "webproxy:8080");
    res = curl_easy_perform(curl);

    curl_easy_cleanup(curl);
    return 0;
}
```

这个程序说明了使用libCurl进行网络编程是多么容易。只使用了短短几行代码，这个程序就通过libCurl函数库连接到了一个远程Web站点，并显示了获取的Web页面。如果你的网络通过代理服务器来访问Web站点，你需要将选项CURLOPT_PROXY的值修改为符合你的网络设置情况，否则，你需要删除这个选项行。

使用以下命令来编译源文件getweb.c：

```
$ cc -I /usr/local/include -o getweb getweb.c -lcurl
```

在完成编译之后，使用一些Web页面来测试这个新的程序。你可以指定一个Web站点的完整URL（如`http://curl.haxx.se`），你也可以仅仅指定Web站点的地址部分（如`curl.haxx.se`）。libCurl函数库将试图通过URL的名称来确定要使用的默认协议。

如果连接成功，Web页面的HTML代码将显示在你的主控台上：

```
$ ./getweb http://www.google.com
<html><head><meta http-equiv="content-type" content="text/html; charset=ISO-8859-1"><title>Google</title><style><!--
body,td,a,p,.h{font-family:arial,sans-serif}
.h{font-size:20px}
.q{color:#00c}
--></style>
<script>
```

```

<!--
function sf(){document.f.q.focus();}
// -->
</script>
</head><body bgcolor="#ffffff text="#000000 link="#0000cc vlink="#551a8b alink="#ff0000
onLoad=sf() topmargin=3 marginheight=3><center><table border=0 cellspacing=0
cellpadding=0 width=100%><tr><td align=right nowrap><font size=-1><a href="/
url?sa=p&pref=ig&pval=3&q=http://www.google.com/ig%3Fhl%3Den&sig=__yvmOvIrk79QYmDkr
JAeuY08jTmo">Personalized Home</a>&ampnbsp|&ampnbsp<a href="https://www.google.com/
accounts/Login?continue=http://www.google.com/&hl=en">Sign in</a></font></td></
tr><tr height=4><td><img alt="" width=1 height=1></td></tr></table><br><br>
<form action=/search name=f><table border=0 cellspacing=0 cellpadding=4><tr><td
nowrap><font size=-1><b>Web</b>&ampnbsp&ampnbsp&ampnbsp&ampnbsp<a class=q
href="/imghp?hl=en&ie=UTF-8&tab=wi">Images</a>&ampnbsp&ampnbsp&ampnbsp&ampnbsp&ampnbsp<a class=q
href="http://video.google.com/?hl=en&ie=UTF-8&tab=wv">Video<a style="text-
decoration:none"><sup><font
color=red>New!</font></sup></a></a>&ampnbsp&ampnbsp&ampnbsp&ampnbsp<a class=q
href="http://news.google.com/nwshp?hl=en&ie=UTF-
8&tab=wn">News</a>&ampnbsp&ampnbsp&ampnbsp&ampnbsp&ampnbsp<a class=q
href="/maps?hl=en&ie=UTF-8&tab=wl">Maps</a>&ampnbsp&ampnbsp&ampnbsp&ampnbsp<b><a
href="/intl/en/options/" class=q onclick="this.blur();return
togDisp(event);">more&nbsp;&raquo;</a></b><script><!--
function togDisp(e){stopB(e);var elems=document.getElementsByName('more');for(var
i=0;i<elems.length;i++){var obj=elems[i];var
dp="";if(obj.style.display==""){dp="none";}obj.style.display=dp;}return false;}
function stopB(e){if(!e)e=window.event;e.cancelBubble=true;}
document.onclick=function(event){var
elems=document.getElementsByName('more');if(elems[0].style.display ==
""){togDisp(event);}}
//-->
</script><style><!--
.cb{margin:.5ex}
--></style>
<span name=more id=more
style="display:none;position:absolute;background:#fff;border:1px solid
#369;margin:-.5ex 1.5ex;padding:0 0 .5ex .8ex;width:16ex;line-height:1.9;z-
index:1000" onclick="stopB(event);"><a href="#" onclick="return togDisp(event);"><img
border=0 src=/images/x2.gif width=12 height=12 alt="Close menu" align=right
class=cb></a><a class=q href="http://books.google.com/bkshp?hl=en&ie=UTF-
8&tab=wp">Books</a><br><a class=q
href="http://froogle.google.com/frghp?hl=en&ie=UTF-8&tab wf">Froogle</a><br><a
class=q href="http://groups.google.com/grphp?hl=en&ie=UTF-
8&tab=wg">Groups</a><br><a href="/intl/en/options/" class=q><b>even more
&raquo;</b></a><span></span></font></td></tr></table><table cellspacing=0
cellpadding=0><tr><td width=25%>&ampnbsp</td><td align=center><input type=hidden
name=hl value=en><input type=hidden name=ie value="ISO-8859-1"><input
maxlength=2048 size=55 name=q value="" title="Google Search"><br><input type=submit
value="Google Search" name=btnG><input type=submit value="I'm Feeling Lucky"
name=btnI></td><td valign=top nowrap width=25%><font size=-2>&ampnbsp&ampnbsp&ampnbsp<a
href=/advanced_search?hl=en>Advanced Search</a><br>&ampnbsp&ampnbsp&ampnbsp<a
href=/preferences?hl=en>Preferences</a><br>&ampnbsp&ampnbsp&ampnbsp<a
href=/language_tools?hl=en>Language
Tools</a></font></td></tr></table></form><br><br><font size=-1><a

```

```

    href="/intl/en/ads/">Advertising&nbsp;Programs</a> - <a href=/services/>Business
Solutions</a> - <a href=/intl/en/about.html>About Google</a></font><p><font size=-
2>&copy;2006 Google</font></p>
$
```

我们只使用了短短几行代码就下载了完整的www.google.com的Web页面的HTML代码并显示了它。你能想象如果在应用程序中使用原始套接字完成这一工作需要多少行代码吗？

2. 保存接收到的数据

前面的例子表明在默认情况下，libCurl将获取的数据显示在标准输出上。对于许多程序来说，这并不是你所希望的结果。相反，你希望能够在应用程序中处理接收到的数据，然后向客户端显示恰当的信息。libCurl函数库提供了一个选项以帮助你完成这一工作。

CURLOPT_WRITEDATA选项允许你定义一个流，接收到的数据将传递给这个流而不是发送到标准输出。你可以在流中执行任何你想要的数据处理，并控制在应用程序中显示哪些数据。

如果你想在将接收到的数据直接发送给流之前对数据进行处理，可以使用 CURLOPT_WRITEFUNCTION选项。它允许你定义一个回调函数，当接收到数据时，libCurl将调用该函数。在回调函数中，你可以在将数据发送给由 CURLOPT_WRITEDATA选项定义的流之前对数据进行处理。

代码清单5-9中的程序getheaders.c演示了如何定义一个文件流来保存通过一个HTTP连接获取到的数据。获取的数据被分别保存到两个文件中，一个保存HTTP header数据，一个保存HTTP body数据。

代码清单5-9 使用libCurl保存获取的Web数据

```

/*
 *
 * Professional Linux Programming - Storing retrieved data
 */
#include <stdio.h>
#include <curl/curl.h>

int main(int argc, char **argv)
{
    CURL *curl;
    CURLcode result;
    FILE *headerfile, *bodyfile;

    headerfile = fopen("header.txt", "w");
    bodyfile = fopen("body.txt", "w");
    curl = curl_easy_init();
    if (!curl)
    {
        perror("curl");
        return 1;
    }
    curl_easy_setopt(curl, CURLOPT_URL, argv[1]);
    curl_easy_setopt(curl, CURLOPT_WRITEHEADER, headerfile);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, bodyfile);

    curl_easy_setopt(curl, CURLOPT_VERBOSE, 1);
```

```

result = curl_easy_perform(curl);
curl_easy_cleanup(curl);

fclose(headerfile);
fclose(bodyfile);
return 0;
}

```

程序getheaders.c定义了两个文件流：headerfile和bodyfile。 CURLOPT_WRITEHEADER选项用于提取HTTP header数据，并将它放入文件header.txt中。HTTP body数据被放入文件body.txt中。在编译和运行这个程序之后，我们可以在文件header.txt中看到程序的输出结果：

```

$ cat header.txt
HTTP/1.1 200 OK
Date: Sun, 08 Oct 2006 17:14:32 GMT
Server: Server
Set-Cookie: skin=noskin; path=/; domain=.amazon.com; expires=Sun, 08-Oct-2006
17:14:32 GMT
x-amz-id-1: 00Z0BGDZQZ7VKNNKKTQ7E
x-amz-id-2: FPU2ju1jPdhsebipLULkH6d0uAXtsDBP
Set-cookie: session-id-time=11608956001; path=/; domain=.amazon.com; expires=Sun
Oct 15 07:00:00 2006 GMT
Set-cookie: session-id=002-7554544-2886459; path=/; domain=.amazon.com; expires=Sun
Oct 15 07:00:00 2006 GMT
Vary: Accept-Encoding,User-Agent
Content-Type: text/html; charset=ISO-8859-1
nnCoection: close
Transfer-Encoding: chunked

$

```

与www.amazon.com进行的HTTP会话所获取的HTTP header信息被保存到文件header.txt中，而HTML页面的内容则被保存到文件body.txt中。

5.4 本章总结

在如今的网络世界中，一个专业Linux程序员必须知道如何将Linux应用程序与网络相连。这一工作既可以使用套接字编程来完成，也可以利用标准的开放源码网络函数库来完成。套接字编程提供了与网络交互的最灵活的方式，但它需要你手工编写通过网络传输数据的所有必需的功能。你必须首先确定使用什么协议与远程网络设备通信，选择标准协议还是实现你自己的协议。服务器和客户端应用程序有着不同的编程需求，你通常需要为每一方设计不同的逻辑以确保它们能正确通信。

使用网络函数库极大地简化了网络编程。有许多开放源码函数库提供了针对C和C++应用程序以及许多Linux脚本语言的标准网络协议实现。libCurl函数库是一个广受欢迎的开放源码函数库，它通过使用简单的C语言函数调用为FTP、HTTP、Telnet和其他标准网络协议提供了发送和接收数据的简易接口。知道如何使用这些标准网络函数库可以节省你的时间，使得你不必手工编写每个网络协议，从而使得你作为一个网络程序员的生活变得更加轻松。

第6章

数 据 库

当一个Linux应用程序在运行时，它所使用的所有数据都驻留在系统内存中。一旦它停止运行，内存中的数据就会丢失。当你再次运行该程序时，你将不能访问到上一次运行中所使用的数据。大多数数据处理的应用程序都需要将数据提供给应用程序的不同会话期（也称为数据的持久性）。信息的持久性存储对任何必须处理历史数据（如员工记录、库存信息或班级成绩）的应用程序都是至关重要的。

本章介绍了两种用于在Linux程序中实现数据持久性的方法。第一种方法利用了内置的数据库引擎。在应用程序中包含一个内置的数据库引擎将为你提供简单的持久性数据功能而不用承担使用一个独立的数据库服务器所需要的管理和开销。第二种方法利用了一个全功能的开放源码数据库服务器。通过使用一个数据库服务器，用户可以在网络的任何一个位置，甚至通过因特网访问你的数据。

6.1 持久性数据存储

持久性数据存储的关键是可以快速的获取存储的数据。现代数据库理论运用许多技术和方法来提供快速的数据检索。有许多流派阐述了数据库应该如何被设计、配置和操作。幸运的是，Linux提供的产品涵盖了范围广泛的解决方案。

Linux应用程序通常使用下述三种方法之一来实现数据持久性：

- 标准的文件读写；
- 内置数据库引擎；
- 外部数据库服务器。

当创建一个必须存储数据的Linux应用程序时，需要考虑每一种方法的利弊以选择合适的方法。

6.1.1 使用标准文件

数据持久性最基本的形式是将数据保存到系统硬盘上的标准文件中。在Linux系统中，你可以将数据元素写入文件，并在下次运行程序时访问同一个数据元素。你可以使用Linux操作系统来控制对数据文件的访问权，使得未经授权的用户或应用程序不能看到存储的数据。

在大多数Linux应用程序中，对文件的使用是一个多步骤的过程。图6-1显示了与文件交互所需的步骤。

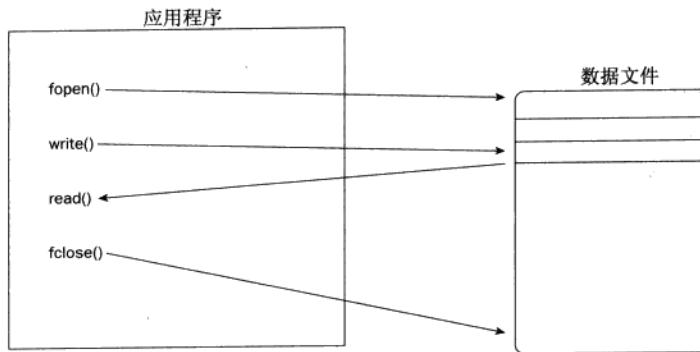


图6-1 在Linux应用程序中使用文件的步骤

大多数Linux编程语言提供了用于打开文件、写入数据、读取数据和关闭文件的函数。你必须在应用程序中编写合适的函数以从文件中存储和获取数据。

使用标准文件来实现数据持久性的缺点是性能。虽然我们可以轻松地创建一个文件并向它写入数据，但在文件中查找数据却是一个挑战。在包含1 000个记录的数据文件中查找某一个记录可能意味着在找到你需要的数据之前需要读取1 000个记录。为了解决这个问题，你必须使用数据库的数据持久性解决方法。

6.1.2 使用数据库

现代数据库理论提供了许多方法来提高查询数据库中数据的性能。与将数据记录以其创建的顺序保存到文件中不同，一个数据库系统可以基于值、关键字来对数据记录进行排序，甚至可以创建独立的文件来包含关键字和指向完整记录的指针。基于唯一的关键字来对数据进行排序有利于数据库快速地检索存储的信息，这比扫描每一个记录以发现需要的数据要快得多。

实现一个数据库系统的主要技巧在编码中。创建适当的逻辑以实现一个数据库需要编写大量的代码。关于如何在一个应用程序中实现数据库逻辑大致分为两种流派。

一个流派的想法是使用函数库在应用程序中集成简单的数据库功能，如图6-2所示：

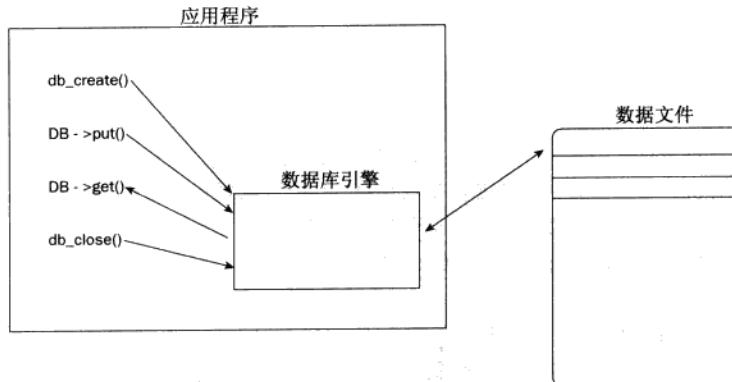


图6-2 在应用程序中实现数据库逻辑的流派之一——使用函数在应用程序中集成简单的数据库功能

数据库引擎函数库提供了用于创建数据库文件、在数据库中插入、修改和删除数据记录、在数据库中查询数据元素的函数。通过在程序代码中实现数据库功能，我们就不需要使用一个必须进行单独管理的独立的数据库服务器了，因为所有的数据库功能都包含在应用程序中。当然，这也意味着需要由应用程序来控制必需的数据文件和访问这些文件。

使用内置数据库引擎的弊端出现在多用户环境中。当只运行一个应用程序时，在必要的时候访问和更新数据库文件完全没有问题。但如果同时有多个用户试图访问和更新数据库文件，而且他们各自都使用自己的数据库引擎，那么数据库文件中数据的完整性将出现问题。我们需要有一个单独的实体来控制对数据的访问，并确定何时可以将更新的数据提供给其他用户。

这就是数据库服务器发挥作用的时候了。数据库服务器提供了用于处理数据库中数据的所有逻辑。应用程序必须与数据库服务器交互才能访问数据库中的数据。数据库服务器可以控制哪些数据可以呈现给用户，以及从用户那里接收到的数据如何存储在数据库中。一个典型的数据库服务器环境如图6-3所示：

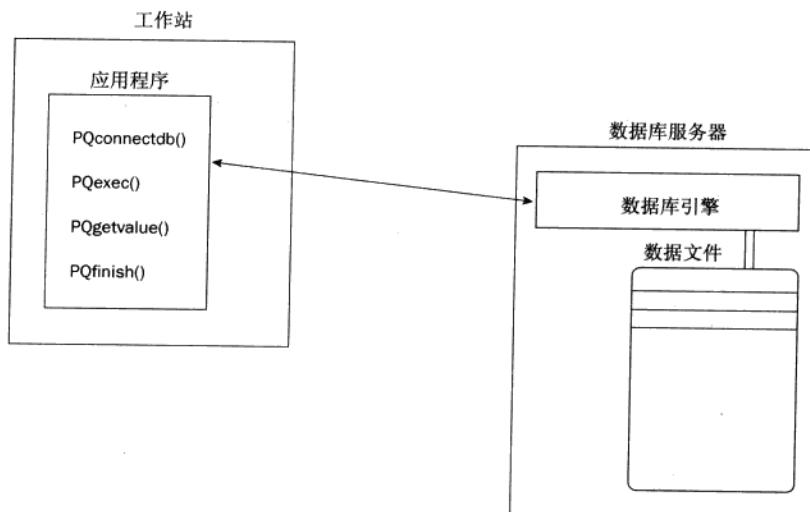


图6-3 典型的数据库服务器环境

为了与数据库服务器交互，每个应用程序必须再次使用函数库。函数库中包含了用于连接到数据库服务器、插入、修改和删除数据库中的数据以及查询包含在数据库中数据的函数。因为有各种客户端需要与数据库服务器连接和交互，所以人们创建了许多标准来简化这一过程。其中一个标准就是标准查询语言（Standard Query Language, SQL）。这是一个用于向数据库服务器发送命令和检索结果的标准协议。

以下几节将介绍两个流行的Linux数据库产品。Berkeley DB产品演示了如何在应用程序中使用一个内置的数据库引擎，而PostgreSQL产品演示了使用中央数据库服务器的方法。

6.2 Berkeley DB 软件包

迄今为止，最受欢迎和最有用的内置数据库引擎软件包是开放源码的Berkeley DB软件包。它在一个简单的、易于使用的函数库中提供了许多高级的数据库功能，该函数库可以连接到你的C或C++应用程序中。

不幸的是，Berkeley DB产品具有多变的生命周期。正如它的名字所暗示的，它一开始只是作为一个简单的数据库项目用于美国加州大学伯克利分校的BSD操作系统4.4版本。由于它越来越受欢迎，在1996年，Netscape公司希望作者能够发布一个通用的产品。为此还单独成立了一个Sleepycat软件公司来继续对从BSD UNIX发行版中分离的Berkeley DB软件进行开发。SleepyCat同时以开放源码许可证（被称为Sleepycat许可证）和用于支持商业应用的商业许可证发布该软件。

最近，Oracle公司收购了SleepyCat软件公司。Oracle作为一个主要的商业数据库厂商，将负责对Berkeley DB开放源码产品进行维护。

以下几节将介绍如何下载、编译、安装Berkeley DB软件包以及如何在Linux应用程序中使用它。

6.2.1 下载和安装

因为Berkeley DB是一个非常流行的软件包，所以许多Linux发行版都包含了它（有些发行版甚至会在默认情况下安装它）。你既可以选择安装针对你的发行版的预编译软件包（如果它尚未被安装），你也可以通过Oracle网站下载、编译和安装最新版本的Berkeley DB软件。

如果你选择安装预编译软件包，请遵循你的Linux发行版的标准软件包安装流程。请确认你同时安装了库文件和开发文件。它们通常包装在两个单独的软件包中。

你可以从Oracle网站上下载最新版本的Berkeley DB。它的网址是www.oracle.com/technology/products/berkeley-db/index.html。在该页中，单击页面右上角的Download链接进行下载。

在写作本书的时候，Berkeley DB的最新版本是4.5.20。它以各种不同的配置和软件包格式提供。为了在Linux系统上安装该软件，请选择.tar.gz软件包，有没有AES加密都可以。

将下载的文件保存在你的Linux系统的一个工作目录中。因为下载的是源代码软件包，所以它必须被释放和编译。要释放源代码，请使用适合你所下载的软件包格式的合适的解压缩工具：

```
$ tar -zxvf db-4.5.20.tar.gz
```

这个命令在当前目录下创建了子目录db-4.5.20，并将源代码释放到该子目录中。

Berkeley DB软件包的编译方式与其他你可能使用过的开放源码产品有所不同。为了编译源代码，你必须首先进入主目录中正确的build_xxx子目录。对于Linux系统来说，这个子目录就是build_unix目录。你必须在该目录中运行位于dist目录下的configure程序。整个过程如下所示：

```
[db-4.5.20]$ cd build_unix
[build_unix]$ ../dist/configure
```

configure工具检查系统环境，并产生编译程序所需的文件。当它运行结束后，你就可以在build_unix目录中运行make命令了：

```
[build_unix]$ make
```

当编译结束后，你需要运行“make install”命令将库文件和开发文件安装到你的系统中：

```
[build_unix]# make install
```

请记住这个步骤必须以.root用户身份来执行。一旦库文件和开发文件安装好，你就可以开始编程了。

6.2.2 编译程序

在默认情况下，Berkeley DB库文件和开发文件将被安装到/usr/local/BerkeleyDB.4.5目录中。这个顶层目录包含了以下几个目录：

- ❑ bin: 包含用于检查和修复Berkeley DB数据库文件的工具。
- ❑ docs: 包含C API和工具的HTML文档。
- ❑ include: 包含用于编译Berkeley DB应用程序的C和C++头文件。
- ❑ lib: 包含用于编译和运行Berkeley DB应用程序的共享库文件。

为了运行使用Berkeley DB函数库的应用程序，你必须将lib目录添加到系统的库文件目录列表中。/etc/ld.so.conf文件包含了Linux搜索库文件的目录列表。你可以以.root用户身份将/usr/local/BerkeleyDB.4.5/lib目录添加到该文件的目录列表中（不要删除该文件中任何已有的目录，否则可能影响你的系统的正常运行）。一旦你修改了/etc/ld.so.conf文件，你必须运行ldconfig命令（以.root用户身份）来更新系统。

现在你的系统已准备好运行Berkeley DB应用程序了。为了编译使用Berkeley DB函数的应用程序，你必须告诉编译器目录和库文件所在的目录，并且还必须向链接器指定db函数库。下面是一个编译Berkeley DB应用程序的例子：

```
$ cc -I/usr/local/BerkeleyDB.4.5/include -o test test.c -L/usr/local/BerkeleyDB.4.5  
-ldb
```

参数-I用于指定编译程序所需头文件的位置，而参数-L用于指定将应用程序链接到一个可执行程序所需库文件的位置。

6.2.3 基本数据处理

现在已在系统中装载并配置了Berkeley DB函数库，你已准备好在应用程序中使用它们了。本节将介绍在程序中使用Berkeley DB函数来利用数据库所需的步骤。

1. 打开和关闭数据库

在访问Berkeley DB数据库文件之前，你必须先打开一个数据库文件。当使用完数据库文件之后，必须关闭它，否则可能会造成数据的损坏。我们使用一个DB句柄来控制对数据库文件的访问。DB句柄是通过db_create()函数创建的：

```
int db_create(DB **dbp, DB_ENV *env, u_int32_t flags)
```

参数dbp指定用于访问数据库文件的DB句柄。参数env指定数据库文件打开时所处的环境。如果它的值为NULL，则该数据库被认为是一个独立的数据库，并且所有指定的设置都只用于该文件。你也可以为它指定一个DB_ENV值以将这个数据库和其他数据库放入一个环境中。为这个环境所做的任何设置（如设置文件锁）都将应用到在该环境中创建的所有数据库文件上。对于运行在Linux环境中的数据库来说，参数flags应该设置为0。

db_create()函数在成功时返回0，失败时返回非零值。

一旦创建了DB句柄，我们就可以使用open()函数来打开一个新的或已有的数据库文件：

```
int DB->open(DB *db, DB_TXN *txnid, const char *file, const char *database,
               DBTYPE type, u_int32_t flags, int mode)
```

`open()` 函数使用由 `db_create()` 函数创建的DB句柄`db`。如果这个函数调用属于一个事务处理的一部分，那么参数`txnid`指定一个打开的事务对象。参数`file`和`database`分别指定用于数据库的系统文件的文件名和存储在该文件中的数据库的名字。Berkeley DB 允许你在一个物理文件中存储多个数据库。如果文件中只有一个数据库，那么你可以为参数`database`指定NULL值，否则，你必须使用单独的`open()`函数调用来打开每一个数据库。

如果你要创建一个新的数据库，你必须通过参数`type`指定数据库的类型。目前支持的数据库类型有：

- `DB_BTREE`: 一个有序的平衡树数据库结构，使用关键字值来排序数据。
- `DB_HASH`: 一个扩展线性哈希表，使用关键字的哈希值来排序数据。
- `DB_QUEUE`: 一个定长记录队列，使用逻辑记录号作为关键字。
- `DB_RECNO`: 一个定长或变长记录队列，使用逻辑记录号作为关键字。

`DB_BTREE` 和 `DB_HASH` 类型通过对输入数据库文件中的数据进行及时的排序以提供对数据的快速访问。当然，这也意味着新数据的存储需要花费更长的时间，因为数据必须被放置到文件的特定位置。对于特大型数据库而言，`DB_HASH` 方法的执行效率被认为比 `DB_BTREE` 方法要高。

`DB_QUEUE` 和 `DB_RECNO` 类型通常被用于那些基于逻辑记录号来排序的数据。它们使用逻辑记录号作为关键字值。记录只能通过记录号来获取。不过，在这些类型的数据库中记录的插入和检索都非常快。如果应用程序需要快速地数据存储和检索，而不需要进行排序，那么这些类型将最适合于你。

如果你是打开一个已有的数据库，也可以为参数`type` 指定 `DB_UNKNOWN` 值。使用这个值之后，Berkeley DB 将试图自动确定这个已有数据库的类型。如果它不能自动识别数据库的类型，`open()` 函数调用将失败。

参数`flags` 指定数据库文件打开的方式，它可以使用的值见表6-1。

表6-1 在指定数据库文件打开方式时参数`flags`可使用的值

标记	说明
<code>DB_AUTO_COMMIT</code>	使用一个事务来打开数据库文件。如果调用成功，打开操作将是可恢复的，否则，不会创建数据库文件
<code>DB_CREATE</code>	如果数据库文件不存在，则创建它
<code>DB_DIRTY_READ</code>	对数据库的读操作可能请求返回已修改但还未保存的数据
<code>DB_EXCL</code>	如果数据库已存在，则返回一个错误
<code>DB_NOMMAP</code>	不将数据库文件映射到内存中
<code>DB_RDONLY</code>	以只读方式打开数据库
<code>DB_THREAD</code>	使得返回的DB句柄可以被多线程使用
<code>DB_TRUNCATE</code>	截断数据库文件，清除文件中已有的所有数据库

`flags` 的值可以使用逻辑或符号 | 结合。例如，你可以指定标记 `DB_CREATE | DB_EXCL`，它将创建一个新的数据库文件，如果该文件已存在，则打开失败。在默认情况下，`DB_CREATE` 标记将打开一个已有的数据库。

参数`mode` 指定文件的UNIX系统文件模式。如果将它的值指定为0则表示该文件的拥有者和组具备读写权限，而其他人没有任何权限。这个值可以使用八进制UNIX文件模式设置方法来指定（如0664）。

下面这个例子创建了一个DB句柄，并打开一个新的数据库：

```
DB *dbp;
int ret;

ret = db_create(&dbp, NULL, 0);
if (ret != 0)
{
    perror("create");
    return 1;
}

ret = dbp->open(dbp, NULL, "test.db", NULL, DB_BTREE, DB_CREATE, 0);
if (ret != 0)
{
    perror("open");
    return 1;
}
```

在上面这一小段代码片断中，我们使用db_create()函数创建了DB句柄dbp，然后使用它创建（或打开）数据库文件test.db，这个数据库使用B-Tree数据库格式和默认的拥有者和组的权限。

2. 添加新数据

现在已有了一个打开的数据库文件，你可以把一些数据放进去了。完成这一工作的函数是put()。它的原型如下：

```
int DB->put(DB *db, DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags)
```

参数db和txnid的含义与open()函数中的一样，它们分别指定DB句柄和用于这个函数的事务。

参数key和data指定数据库中用于存储数据的关键字/数据对。

DBT结构包含一个data成员和一个size成员。因此，成员key.data包含记录的关键字值，而成员data.data包含记录的数据值。

参数flags指定新数据放入数据库文件的方式。对于DB_QUEUE和DB_RECNO数据库来说，可以使用标记DB_APPEND将记录添加到数据库文件的结尾。而对于DB_BTREE和DB_HASH数据库来说，可以使用标记DB_NODUPDATA以保证不会向数据库中添加关键字/数据对相同的记录。标记DB_NOOVERWRITE用于阻止覆盖关键字相同的记录。

关键字/数据对的结合可能会引起误解。许多人认为这限制了他们只能在记录中使用单个的数据值。其实数据值可以是你需要的任何数据类型，包括结构。你可以创建一个C语言结构来包含应用程序需要的所有数据元素。C语言结构的每一个实例将作为一个单独的数据值存储到数据库文件中。

代码清单6-1中的程序newemployee.c演示了新数据的添加。

代码清单6-1 添加记录到Berkeley DB数据库

```
/*
 *
 * Professional Linux Programming - Adding a new employee record
 *
 */
#include <stdio.h>
#include <db.h>
```

```
#define DATABASE "employees.db"

int main()
{
    DBT key, data;
    DB *dbp;
    int ret;
    struct data_struct {
        int empid;
        char lastname[50];
        char firstname[50];
        float salary;
    } emp;

    ret = db_create(&dbp, NULL, 0);
    if (ret != 0)
    {
        perror("create");
        return 1;
    }

    ret = dbp->open(dbp, NULL, DATABASE, NULL, DB_BTREE, DB_CREATE, "0");
    if (ret != 0)
    {
        perror("open: ");
        return 1;
    }

    while(1)
    {
        printf("Enter Employee ID: ");
        scanf("%d", &emp.empid);
        if (emp.empid == 0)
            break;
        printf("Enter Last name: ");
        scanf("%s", &emp.lastname);
        printf("Enter First name: ");
        scanf("%s", &emp.firstname);
        printf("Enter Salary: ");
        scanf("%f", &emp.salary);

        memset(&key, 0, sizeof(DBT));
        memset(&data, 0, sizeof(DBT));

        key.data = &(emp.empid);
        key.size = sizeof(emp.empid);
        data.data = &emp;
        data.size = sizeof(emp);

        ret = dbp->put(dbp, NULL, &key, &data, DB_NOOVERWRITE);
        if (ret != 0)
        {
            printf("Employee ID exists\n");
        }
    }
}
```

```

    }
}

dbp->close(dbp, 0);
return 0;
}

```

程序newemployee.c创建了一个DB句柄，并将文件employees.db作为一个独立的数据库打开（注意，.db文件后缀名的使用并不是必须的，但我们通常使用它来识别Berkeley DB数据库文件）。接着，我们使用一个while()循环来询问用户新雇员的信息。emp结构用于数据库中的数据元素。它包含一个整数的雇员ID、字符串lastname和firstname以及浮点数的薪水值。我们将从用户处获取的这些值放入emp结构。

在使用DBT对象key和data之前，调用memset函数将它们的内存位置清空是一个好习惯。这避免了游离的比特进入数据元素的内容中。接下来，key值被设置为雇员ID，data值被设置为整个emp结构。一旦一切就绪，我们就执行put()函数，使用DB_NOOVERWRITE标记以确保当你试图添加使用相同雇员ID的记录时，该函数调用将失败。注意，如果你要更新一个已有雇员的记录，只需删除DB_NOOVERWRITE标记即可。

当输入的新雇员ID为0时，将退出while()循环。然后我们调用close()函数来正确关闭数据库文件。

编译这个应用程序（记得要使用正确的头文件和库文件），然后运行它。你应该可以往数据库中添加雇员记录了。当你结束程序时，你可以在目录中看到新的employees.db数据库文件。

3. 检索数据

当数据被放入数据库文件后，你很可能会在某个时候想取回它。get()函数就用于完成这个工作：

```
int DB->get(DB *db, DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags)
```

和其他函数一样，参数db指定一个打开的数据库DB句柄，参数txnid指定一个已有的事务。参数key和data是用于放置关键字值和数据值的DBT对象。在检索数据的情况下，参数key指定要检索的关键字值，而参数data在函数执行完成时将被对应的数据值填充。

参数flags指定在检索数据时要执行的动作。如果将它设置为0，则只获取数据元素，而不执行任何其他动作。它可以使用的值如表6-2所示：

表6-2 当指定检索数据时要执行的动作时，参数flags可使用的值

标记	说明
DB_CONSUME	在DB_QUEUE类型的数据库中，获取位于队列头的记录数据，同时从数据库中删除该记录
DB_CONSUME_WAIT	和DB_CONSUME功能一样，除了当队列为空时，它将等待记录的出现
DB_GET_BOTH	仅当数据库中的关键字值和数据值都匹配给定的参数时，才返回它们
DB_SET_RECNO	从数据库中同时获取关键字值和数据值，而不仅仅是数据值。它只用于DB_BTREE类型的数据库
DB_DIRTY_READ	读取已修改但还未提交的数据
DB_MULTIPLE	返回多个数据元素。如果data.size的值不足够大，它将被设置为包含所有数据元素所需的大小
DB_RMW	在执行检索时获取写锁而不是读锁。用于事务处理

代码清单6-2中的程序getemployee.c演示了如何从数据库文件中检索数据。

代码清单6-2 从Berkeley DB数据库中检索记录

```
/*
 *
 * Professional Linux Programming - Retrieving an employee record
 *
 */
#include <stdio.h>
#include <db.h>

#define DATABASE "employees.db"

int main()
{
    DBT key, data;
    DB *dbp;
    int ret;
    struct data_struct {
        int empid;
        char lastname[50];
        char firstname[50];
        float salary;
    } emp;

    ret = db_create(&dbp, NULL, 0);
    if (ret != 0)
    {
        perror("create");
        return 1;
    }

    ret = dbp->open(dbp, NULL, DATABASE, NULL, DB_BTREE, DB_CREATE, 0);
    if (ret != 0)
    {
        perror("open: ");
        return 1;
    }

    while(1)
    {
        printf("Enter Employee ID: ");
        scanf("%d", &emp.empid);
        if (emp.empid == 0)
            break;

        memset(&key, 0, sizeof(DBT));
        memset(&data, 0, sizeof(DBT));
        key.data = &(emp.empid);
        key.size = sizeof(emp.empid);
        data.data = &emp;
        data.ulen = sizeof(emp);
```

```

data.flags = DB_DBT_USERMEM;

ret = dbp->get(dbp, NULL, &key, &data, 0);
if (ret != 0)
{
    printf("Employee ID does not exist\n");
} else
{
    printf(" Employee: %d - %s,%s\n", emp.empid, emp.lastname,
emp.firstname);
    printf(" Salary: $%.2lf\n", emp.salary);
}
}

dbp->close(dbp, 0);
return 0;
}

```

代码清单6-2首先演示了用于创建DB句柄和打开employees.db数据库文件的标准方法。然后在while()循环中要求用户输入要检索的雇员ID。这里再次使用memset函数来清除key和data的值，并将输入的雇员ID放入key.data。data则被设置为data_struct结构的一个空实例。

data中还包含一个用于指定数据处理方式的标记。DB_DBT_USERMEM标记告诉Berkeley DB这个程序使用它自己的内存区域来处理返回的数据。

当get()函数完成后，返回的数据被放入emp结构并显示在主控台上。while()循环则继续执行直到用户输入为0的雇员ID为止。

4. 删除数据

另一个有用的功能是从数据库文件中删除记录。del()函数用于这个目的：

```
int DB->del(DB *dbp, DB_TXN *txnid, DBT *key, u_int32_t flags)
```

删除的记录基于在函数调用中提供的key值。在当前Berkeley DB版本中，参数flags还未使用，它应被设置为0。

代码清单6-3中的程序delemployee.c演示了如何使用del()函数来删除employees.db数据库中一个指定的记录。

代码清单6-3 从Berkeley DB数据库中删除一个记录

```

/*
 *
 * Professional Linux Programming - removing an employee record
 *
 */
#include <stdio.h>
#include <db.h>

#define DATABASE "employees.db"

int main()
{
    DBT key;

```

```

DB *dbp;
int ret;
struct data_struct {
    int empid;
    char lastname[50];
    char firstname[50];
    float salary;
} emp;

ret = db_create(&dbp, NULL, 0);
if (ret != 0)
{
    perror("create");
    return 1;
}

ret = dbp->open(dbp, NULL, DATABASE, NULL, DB_BTREE, DB_CREATE, 0);
if (ret != 0)
{
    perror("open: ");
    return 1;
}

while(1)
{
    printf("Enter Employee ID: ");
    scanf("%d", &emp.empid);
    if (emp.empid == 0)
        break;

    memset(&key, 0, sizeof(DBT));
    key.data = &(emp.empid);
    key.size = sizeof(emp.empid);

    ret = dbp->del(dbp, NULL, &key, 0);
    if (ret != 0)
    {
        printf("Employee ID does not exist\n");
    } else
    {
        printf(" Employee %d deleted.\n", emp.empid);
    }
}

dbp->close(dbp, 0);
return 0;
}

```

现在你应该可以看懂这个程序的大部分内容了。用户指定要删除的雇员ID，其值被存储在key.data中，然后我们执行del()函数。如果删除成功，该函数将返回0。如果这个关键字在数据库中不存在，del()函数将返回值DB_NOTFOUND。

5. 使用游标

`get()`函数只能用来检索已知关键字值的数据值。有时候，你可能需要在不知道关键字值的情况下从数据库文件中提取数据。

Berkeley DB使用一个被称为游标的数据库概念以允许你遍历数据库中包含的记录。一个游标对象用于指向数据库中的一个记录。你可以在数据库中向前或向后移动游标以获得下一个或前一个记录。一旦游标指向了一个记录，你就可以获取该记录数据、删除游标指向的记录或在数据库中的该位置放置一个新的记录。

一个DBC游标对象是通过为数据库创建的标准DB句柄创建的，我们使用`cursor()`函数来创建它：

```
int DB->cursor(DB *db, DB_TXN *txnid, DBC **cursorp, u_int32_t flags);
```

`cursor()`函数创建一个游标指针并将它分配给DBC对象指针`cursorp`。一旦创建了游标对象，我们就可以使用`c_get()`、`c_put()`或`c_del()`函数在数据库文件中游标指向的位置获取、放置或删除一个记录了。

当你需要列出数据库中包含的所有记录时，拥有遍历数据库中记录的功能就显得非常方便了。代码清单6-4中的程序`listemployees.c`演示了如何在数据库文件中通过游标来列出数据库中包含的所有记录。

代码清单6-4 从Berkeley DB数据库中获取多个记录

```
/*
 *
 * Professional Linux Programming - listing all employee records
 *
 */
#include <stdio.h>
#include <db.h>

#define DATABASE "employees.db"
int main()
{
    DBT key, data;
    DBC *cursor;
    DB *dbp;
    int ret;
    struct data_struct {
        int empid;
        char lastname[50];
        char firstname[50];
        float salary;
    } emp;

    ret = db_create(&dbp, NULL, 0);
    if (ret != 0)
    {
        perror("create");
        return 1;
    }
```

```

ret = dbp->open(dbp, NULL, DATABASE, NULL, DB_BTREE, DB_CREATE, 0);
if (ret != 0)
{
    perror("open: ");
    return 1;
}

ret = dbp->cursor(dbp, NULL, &cursor, 0);
if (ret != 0)
{
    perror("cursor: ");
    return 1;
}

memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));
data.data = &emp;
data.size = sizeof(emp);
data.flags = DB_DBT_USERMEM;

while(ret = cursor->c_get(cursor, &key, &data, DB_NEXT) == 0)
{
    printf("%d - %s,%s  $%.2lf\n", emp.empid, emp.lastname, emp.firstname,
    emp.salary);
}

cursor->c_close(cursor);
dbp->close(dbp, 0);
return 0;
}

```

当创建了DB句柄并打开employee.db数据库文件之后，我们创建了一个用于遍历数据库文件的游标对象cursor。在每次迭代中，我们使用c_get()函数从数据库记录中同时提取出key和data的DBT值。

c_get()函数包括一个标记以表明如何在数据库中移动游标。它有很多可用的选项，最常见的有DB_NEXT、DB_PREV、DB_FIRST、DB_LAST和DB_NEXE_DUP（用于在数据库中找到重复的关键字/数据对）。

当从记录中提取出key和data值之后，我们使用一个data_struct结构来提取保存在data元素中的雇员信息。因为程序使用自己的内存位置，所以我们必须为data元素使用DB_DBT_USERMEM标记。

当数据库中的最后一个记录被获取之后，c_get()函数将返回一个非0值，导致程序退出while()循环。因为程序已不再使用游标，所以它在关闭数据库文件之前调用c_close()函数来关闭游标并释放它使用的内存。

6.3 PostgreSQL 数据库服务器

PostgreSQL数据库服务器是最流行的开放源码数据库之一。它支持许多在商业数据库中常见的数据库功能，如存储过程、视图、触发器、函数和热备份。

当然，使用数据库服务器的缺点是它必须和应用程序分开来管理。通常一个组织会为此专门雇佣

数据库管理员，管理员的唯一职责就是确保数据库服务器的正常运行。但使用一个数据库服务器的优点是你可以将公司中的所有应用程序要使用的所有数据都放到一个服务器中，并且它还得到持续的支持。

为数据库服务器环境编程和使用内置的数据库引擎并没有太大的差别。与打开一个数据库文件不同，你必须与数据库服务器（它运行在本地系统上或运行在远程服务器上）建立连接。一旦你建立了一个连接，就可以通过发送SQL命令来插入、删除或更新数据库表格中的数据了。你还可以通过发送SQL命令来查询表格中的数据。数据库服务器向你的程序返回查询结果（称为结果集），我们然后可以遍历结果集以提取出每条记录数据。

6.3.1 下载和安装

许多Linux发行版都以可安装软件包的形式包括PostgreSQL产品。如果你的Linux发行版中包括该软件，那么你需要遵循发行版的软件安装指南来安装PostgreSQL。

如果你想安装最新版本的PostgreSQL，你可以下载针对你的Linux发行版的源代码软件包或合适的二进制安装软件包。访问PostgreSQL的主站www.postgresql.org，然后点击页面顶部的Download链接。

下载页面包含一个到FTP浏览页面的链接，在该浏览页面中你可以看到各种版本的软件包。文件夹binary中包含针对Red Hat Linux的RPM软件包。文件夹source中包含各种版本的源代码软件包。在写作本书时，最新的版本是8.1.5。

如果你下载的是源代码软件包，你必须使用tar命令将源代码释放到一个工作目录中：

```
$ tar -zxf postgresql-8.1.5.tar.gz
```

这创建了目录postgresql-8.1.5，所有待编译的源代码都在该目录中。进入该目录，然后运行configure和gmake命令来编译可执行文件（PostgreSQL使用GNU gmake工具来代替标准的make工具）。

```
[postgresql-8.1.5]$ ./configure
...
[postgresql-8.1.5]$ gmake
...
All of PostgreSQL successfully made. Ready to install.
[postgresql-8.1.5]$ su
[postgresql-8.1.5]# gmake install
```

在编译和安装了PostgreSQL数据库服务器软件之后，你必须为PostgreSQL服务器创建一个特殊的用户账号。为了避免安全性问题，PostgreSQL服务器不会以root用户身份来运行，它必须以一个普通用户身份运行。标准的做法是创建一个名为postgres的账号，并授予他访问数据库目录的权限。

```
# adduser postgres
# mkdir /usr/local/pgsql/data
# chown postgres /usr/local/pgsql/data
```

PostgreSQL数据库目录可以位于Linux系统中的任何位置，上面的目录只是PostgreSQL推荐的一个位置。请确保无论你将data目录放置到哪里，该目录只能被postgres用户访问。

在启动PostgreSQL服务器程序之前，你必须初始化默认的PostgreSQL数据库。你可以使用initdb命令来完成这个工作（以postgres用户身份）：

```
$ su - postgres
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

initdb程序为默认的PostgreSQL数据库（即postgres）创建必需的文件。当它执行完毕后，/usr/local/pgsql/data目录将包含用于数据库的文件。你现在可以使用pg_ctl命令来启动PostgreSQL服务器了（再次以postgres用户身份）：

```
$ /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
```

pg_ctl命令用于停止、启动和重新载入PostgreSQL服务器配置。你必须使用-D参数来指定数据库目录的位置。你还可以使用-l参数来指定服务器日志文件的文件名。请确保postgres用户对日志文件有写权限。你可以通过查看系统当前运行进程以了解服务器运行的状况。你应该可以看到系统中运行着几个不同的PostgreSQL进程：

```
28949 pts/1    S      0:00 /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/d
28951 pts/1    S      0:00 postgres: writer process
28952 pts/1    S      0:00 postgres: stats buffer process
28953 pts/1    S      0:00 postgres: stats collector process
```

进程postmaster是服务器的主控制器。它将在必要的时候启动其他进程。

6.3.2 编译程序

和Berkeley DB函数库一样，你必须配置你的Linux系统以创建和运行使用PostgreSQL函数库的应用程序。PostgreSQL C语言函数库被称为libpq。该库文件位于/usr/local/pgsql/lib目录中。为了使在你的系统中的PostgreSQL应用程序可以正常运行，你必须将这个目录添加到/etc/ld.so.conf文件中，并以root用户身份运行ldconfig程序。

为了编译使用libpq函数库的C程序，你必须指定头文件和libpq库文件的位置：

```
$ cc -I/usr/local/pgsql/include -o test test.c -L/usr/local/pgsql/lib -lpq
```

参数-I用于指定编译程序所需头文件的位置，参数-L用于指定将应用程序连接为一个可执行程序所需库文件的位置。

6.3.3 创建一个应用程序数据库

当你安装了PostgreSQL软件并在系统中运行它之后，你就可以开始为应用程序创建数据库、模式和表格了。

命令行程序psql提供了一个访问PostgreSQL服务器的简单接口。你必须指定登录ID（使用参数-U）和密码来连接到数据库。在默认情况下，管理员登录ID是postgres，没有密码。

```
$ /usr/local/pgsql/bin/psql -U postgres
Welcome to psql 8.1.5, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help with psql commands
      \g or terminate with semicolon to execute query
      \q to quit

postgres=#
```

你现在已连接到默认的数据库postgres，并拥有全部的特权。

postgres数据库包含了保存统计信息的系统表格。我们建议你为自己的应用程序创建一个单独的数据库。可以使用SQL命令CREATE DATABASE来创建一个新的数据库：

```
postgres=# create database test;
CREATE DATABASE
postgres=#

```

如果命令执行成功，你将看到一个简短的信息。否则，你将看到一个较长的错误信息对问题进行说明。

为了为下面的示例程序做好准备，你应该在新创建的数据库test中创建一个表格。首先，你必须使用元命令\c连接到新的数据库，然后在该数据库中创建表格：

```
postgres=# \c test
You are now connected to database "test".
test=# create table employee (
test(# empid int4 primary key not null,
test(# lastname varchar,
test(# firstname varchar,
test(# salary float4);
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey" for
table "employee"
CREATE TABLE
test=#

```

新的表格已被创建。为数据库的访问专门创建一个用户也是一个好习惯，这样应用程序就不需要使用postgres用户身份来登录了。新的用户应该被授予访问新创建的表格的特权：

```
test=# create role management with nologin;
CREATE ROLE
test=# create role earl in role management;
CREATE ROLE
test=# alter role earl login password 'bigshot' inherit;
ALTER ROLE
test=# grant all on employee to management;
GRANT
test=#

```

PostgreSQL同时提供了组角色和登录角色，登录角色用于允许个人登录数据库，组角色包含登录角色并被授予访问数据库对象的特权。这样你就可以根据用户在组织中职能的改变将他的登录角色转移到不同的组角色中。上面这个例子创建了名为management的组角色，并授予它对employee表格的所有特权。它还创建了一个名为earl的登录角色，并将它添加到management组角色中。现在你可以以earl角色身份登录并访问在test数据库中的employee表格了：

```
$ psql test -U earl -W
Password:
Welcome to psql, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms

```

```

\b for help with SQL commands
\? for help on internal slash commands
\g or terminate with semicolon to execute query
\q to quit

test=> select * from employee;
empid | lastname | firstname | salary
-----+-----+-----+
(0 rows)

test=>

```

要想直接连接到test数据库，你必须在psql命令行上指定它。同样地，要想以earl登录角色登录，必须指定参数-U和-W。在输入earl的密码（设置为bigshot）之后，你就可访问数据库了。现在你已为开始编程做好准备了。

6.3.4 连接服务器

libpq函数库提供了5个函数（具体如表6-3所示）用于建立和关闭一个到PostgreSQL服务器的连接。

表6-3 libpq函数库提供的5个用于启动和关闭到PostgreSQL服务器的连接的函数

函 数	说 明
Pqconnectdb(const char *coninfo)	使用coninfo中的参数建立到一个PostgreSQL服务器的连接，并等待响应
PqconnectStart(const char *coninfo)	使用coninfo中的参数以非阻塞模式建立到一个PostgreSQL服务器的连接
PqconnectPoll(PGconn *conn)	检查未决的非阻塞连接尝试conn的状态
Pgfinish(PGconn *conn)	关闭（结束）一个已建立的PostgreSQL服务器会话conn
Preset(PGconn *conn)	通过关闭先前的会话conn并使用相同的参数建立一个新的会话来重置与PostgreSQL服务器的会话。这个命令将等待来自服务器的响应

PQconnectdb()函数用于与PostgreSQL服务器建立一个新的连接。它的原型如下：

```
PGconn *PQconnectdb(const char *conninfo)
```

PQconnectdb()函数返回一个指向PGconn数据类型的指针。这个值将用于后续的使用这个连接发送命令给服务器的所有函数。

conninfo常数字符串指针定义用于连接PosgreSQL服务器的值。这些值与你已使用的用于连接PostgrSQL服务器的值并没有太大的区别。正如预期的那样，我们可以使用几个标准的参数来定义连接，如表6-4所示：

表6-4 用于定义连接的标准参数

连接参数	说 明
host	PostgreSQL服务器的DNS主机名或IP地址
hostaddr	PostgreSQL服务器的IP地址
port	PostgreSQL服务器的TCP端口号

(续)

连接参数	说 明
dbname	这次会话连接的数据库
user	用于登录服务器的登录角色
password	登录角色的密码
connect_timeout	建立连接的最大等待时间（单位为秒）。0表示一直等待
options	发送给服务器的命令行选项
sslmode	设置SSL优先级：disable不使用SSL；allow将和服务器首先尝试非SSL连接；require只使用SSL连接
service	设置服务名以指定在pg_service.conf配置文件中的额外参数

当在字符串中列出多个参数时，每个参数对设置必须以一个或多个空格符分开。下面是一个创建新连接的例子：

```
const char *conninfo;
Pconn *conn;
conninfo = "host = 127.0.0.1 dbname = test user = earl password = bigshot";
conn = PQconnectdb(conninfo);
```

这个例子使用环回地址127.0.0.1连接到运行在本地系统上的PostgreSQL服务器。它使用登录角色earl与数据库test建立会话。

PQconnectdb()函数是一个阻塞式函数。它将停止（阻塞）程序的执行直到该函数执行完毕。如果在连接字符串中指定的PostgreSQL服务器不能连接，你将不得不在控制权返回程序之前等待connect_timeout秒。对于事件驱动的应用程序来说，这有时会引起问题。因为程序阻塞在一个连接上时，它将不能响应如单击鼠标或键盘输入等事件。

为了解决这个问题，你可以使用非阻塞式的连接函数。PQconnectStart()函数使用与PQconnectdb()相同的连接字符串信息，并与指定的PostgreSQL服务器建立相同类型的连接。但PQconnectStart()不会等待连接建立成功或失败。

程序将在PQconnectStart()函数执行之后立刻继续执行。为了确定连接是成功还是失败，你必须调用PQconnectPoll()函数。这个函数将返回连接尝试的状态，它可以返回的状态值如下：

- CONNECTION_STARTED: 等待连接建立。
- CONNECTION_AWAITING_RESPONSE: 等待来自PostgreSQL服务器的响应。
- CONNECTION_SSL_STARTUP: 协商SSL加密方案。
- CONNECTION_AUTH_OK: 认证成功，等待服务器完成连接。
- CONNECTION_SETENV: 协商会话参数。
- CONNECTION_MADE: 连接建立，等待命令。

如果连接还没有成功或失败了，你必须继续轮询连接，直到连接建立或它最终失败为止。

在与PostgreSQL服务器建立连接之后，你可以使用几个函数来检查连接的状态以及用于建立连接的连接参数（如表6-5所示）。

表6-5 检查连接状态以及用于建立连接的连接参数的函数

函 数	说 明
PQdb(PGconn *conn)	返回连接的数据库名
PQuser(PGconn *conn)	返回用于连接的登录角色
PQpass(PGconn *conn)	返回登录角色的连接密码
PQhost(PGconn *conn)	返回连接的服务器主机名
PQport(PGconn *conn)	返回连接的TCP端口号
PQoptions(PGconn *conn)	返回用于建立连接的任何命令行选项
PQstatus(PGconn *conn)	返回服务器连接的状态
PQtransactionStatus(PGconn *conn)	返回服务器的事务状态
PQparameterStatus(PGconn *conn, const char *param)	返回服务器的一个参数param的当前设置
PQprotocolVersion(PGconn *conn)	返回PostgreSQL后端协议的版本
PQserverVersion(PGconn *conn)	返回以整数值表示的服务器版本
PQerrorMessage(PGconn *conn)	返回最近产生的服务器错误信息
PQsocket(PGconn *conn)	返回与服务器连接的套接字的文件描述符
PBackendPID(PGconn *conn)	返回处理此连接的服务器进程的PID
PQgetssl(PGconn *conn)	如果没有使用SSL，则返回NULL，否则返回一个连接使用的SSL结构

所有的状态函数都使用PQconnectdb()函数返回的PGconn值来确定连接。大多数状态函数都返回一个指向状态信息字符串的指针。下面是一个例子：

```
char *user;
user = PQuser(conn);
```

字符指针user指向用于建立连接的登录角色。PQstatus()函数是一个例外，它返回的是一个ConnStatusType数据类型，它有两个定义的值：

- CONNECTION_OK 表示连接成功；
- CONNECTION_BAD 表示连接失败。

代码清单6-5中的程序version.c演示了如何连接到一个数据库，检查连接的状态并提取PostgreSQL服务器的版本。

代码清单6-5 连接到PostgreSQL数据库

```
/*
 *
 * Professional Linux Programming - connecting to a database
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

int main(int argc, char **argv)
{
    const char *conninfo;
```

```

const char *serverversion;
PGconn *conn;
const char *paramtext = "server_version";

conninfo = "hostaddr = 127.0.0.1 dbname = test user = earl password = bigshot";

conn = PQconnectdb(conninfo);
if (PQstatus(conn) != CONNECTION_OK)
{
    printf("Unable to establish connection: %s",
           PQerrorMessage(conn));
    return 1;
} else
{
    printf("Connection established!\n");
    serverversion = PQparameterStatus(conn, paramtext);
    printf("Server Version: %s\n", serverversion);
}
PQfinish(conn);
return 0;
}

```

这个小小的例子演示了用C语言与PostgreSQL服务器连接并交互的所有基本要素。它使用创建的登录角色earl连接到本机上的数据库test。

在尝试连接之后，我们使用PQstatus()函数来测试连接的状态。如果连接建立成功，PQparameterStatus()函数调用将从PostgreSQL服务器那里获得参数server_version的值。

要编译这个程序，你必须在程序中包括PostgreSQL libpq头文件libpq-fe.h，并与libpq库文件进行连接：

```

$ cc -I/usr/local/pgsql/include -o version version.c -L/usr/local/pgsql/lib -lpq
$ ./version
Connection established!
Server Version: 8.1.5
$ 

```

这个程序如预期的那样正常工作。下一节将介绍更多的可以在服务器上执行的高级函数。

6.3.5 执行SQL命令

与PostgreSQL服务器建立连接之后，你很可能希望能在PostgreSQL服务器上执行SQL命令。libpq函数库中包含了相当多的命令执行函数如表6-6所示：

表6-6 libpq函数库中包含的命令执行函数

函数	说 明
PQexec(PGconn *conn, const char *command)	提交一个字符串命令给服务器并等待结果
PQexecParams(PGconn *conn, const char *command, int nParams, const Oid *paramTypes, const char *paramValues, const int *paramLengths, const int *paramFormats, int resultFormat)	提交一个包含参数的命令给服务器并等待结果。该参数是根据其他参数的长度和格式来定义的。这个命令还指定了查询结果的格式（0表示文本格式，1表示二进制格式）
PQprepare(PGconn *conn, const char *name, const char *query, int nParams, const Oid *paramTypes)	使用连接conn向服务器提交字符串query中包含的命令以创建一个准备好的语句。命令可能会使用参数，参数由PQexecPrepared()函数定义。准备好的语句通过name引用

(续)

函数	说明
PQexecPrepared(PGconn *conn, const char *name, int nParams, const char *paramValues, const int paramLengths, const int paramFormats, int resultFormat)	提交一个请求以执行前面准备好的语句name。你可以指定参数和查询结果的格式(0表示文本格式,1表示二进制格式)
PQresultStatus(PGresult *result)	返回一个已执行命令的结果状态
PQresStatus(ExecStatuwsType status)	把PQresultStatus返回的值转换为一个字符串结果状态
PQresultErrorMessage(PGresult *result)	返回一个已执行命令的错误信息字符串,或在没有错误时返回一个NULL值
PQclear(PGresult *result)	清除(释放)结果状态result的存储空间

PQexec()函数用于在PostgreSQL服务器上执行SQL命令的基本函数。libpq函数库使用一个函数来同时执行查询和非查询SQL命令。所有命令的执行结果都通过PGresult数据类型对象返回:

PGresult *PQexec(PGconn *conn, const char *command)

PQexec()函数需要两个参数。第一个参数是与服务器建立连接时创建的PGconn对象。

第二个参数是一个字符串,它包含了要在服务器上执行的SQL命令。

当在程序中执行PQexec()函数时,命令被发送给服务器,然后程序等待服务器的响应。响应结果被放到一个PGresult数据对象中。因为不同的SQL命令有着不同类型的输出,所以这个数据对象必须有能力处理许多种可能性。

这个数据对象必须首先被PQresultStatus()函数检查以确定命令的结果状态和这个命令的输出类型。可能返回的结果状态有:

- PGRES_COMMAND_OK: 命令执行成功,但它不返回结果集
- PGRES_TUPLES_OK: 命令执行成功并返回一个结果集(甚至可能是一个空的结果集)
- PGRES_EMPTY_QUERY: 发送给服务器的命令是空的
- PGRES_BAD_RESPONSE: 无法理解服务器的响应
- PGRES_NONFATAL_ERROR: 服务器产生了一个通知或警告
- PGRES_FATAL_ERROR: 服务器产生了一个致命的错误信息

这些值定义在libpq函数库中,你可以直接在代码中对它们进行检查:

```
PGresult *result;
result = PQexec(conn, "SELECT lastname from employee");
if (PQresultStatus(result) != PGRES_TUPLES_OK)
{
    ...
}
```

如果命令返回的结果是一个结果集,有许多辅助函数(如表6-7所示)可以用于确定返回的数据以及如何处理它。

表6-7 用于确定返回数据及其处理方式的辅助函数

函数	说明
PQntuples(PGresult *res)	返回结果集res中的记录数(元组数)
PQnfields(PGresult *res)	返回结果集res中的字段数(数据域的个数)
PQfname(PGresult *res, int column)	返回与指定的结果集res和字段编号column相关联的字段名

(续)

函数	说明
PQfnumber(PGresult *res, const char *colname)	返回与指定的结果集res和字段名colname相关联的字段编号
PQftable(PGresult *res, int column)	返回我们抓取字段column所在表的OID
PQgetvalue(PGresult *res, int rec, int column)	返回结果集res中记录rec的字段column的值
PQgetisnull(PGresult *res, int rec, int column)	测试结果集res中记录rec的字段column是否为NULL,如果是则返回1,否则返回0
PQgetlength(PGresult *res, int rec, int column)	返回结果集res中记录rec的字段column的值的长度(单位为字节)

这些函数对于从PGresult对象返回的结果集中挑选数据是非常有用的。程序处理结果集数据的方式取决于返回数据（如果有的话）的类型。以下几小节将说明如何处理在libpq中执行SQL命令所返回的不同类型的数据。

1. 不返回数据的命令

SQL命令如INSERT、UPDATE和DELETE不返回数据，但会返回一个表示命令是否执行成功的状态码。在这种情况下，你必须检查结果状态是否是PGRES_COMMAND_OK，因为它不会返回元组（记录）。

代码清单6-6中的程序update.c演示了这个原则：

代码清单6-6 向PostgreSQL数据库中插入记录

```
/*
 *
 * Professional Linux Programming - insert a new record
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

int main(int argc, char **argv)
{
    const char *conninfo;
    PGconn *conn;
    PGresult *result;
    char *insertcomm;

    conninfo = "hostaddr = 127.0.0.1 dbname = test user = earl password = bigshot";
    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK)
    {
        printf("Unable to establish connection: %s",
               PQerrorMessage(conn));
        return 1;
    } else
    {
        insertcomm = "INSERT into employee values (100, 'Test', 'Ima', 35000)";
        result = PQexec(conn, insertcomm);
        if (PQresultStatus(result) != PGRES_COMMAND_OK)
```

```

    {
        printf("Problem with command: %s\n", PQerrorMessage(conn));
        PQclear(result);
        PQfinish(conn);
        return 1;
    }
    PQclear(result);
}
PQfinish(conn);
return 0;
}

```

程序update.c使用登录角色earl与运行在本地系统上的PostgreSQL服务器的test数据库建立连接。在连接建立之后，我们调用PQexec()函数在服务器上执行一个简单的INSERT SQL命令。

这个命令的结果状态通过使用PQresultStatus()函数和PGRES_COMMAND_OK值来检查。如果命令没有成功，我们调用PQerrorMessage()函数以显示由服务器产生的错误信息。最后，调用PQclear()函数清除PGresult对象使用的内存。清理内存是一个好习惯，尤其当你在另一个命令中复用PGresult对象时更是如此。

编译程序update.c并在PostgreSQL系统上运行它。如果INSERT命令执行成功，屏幕上不会有任何显示（这不是太让人兴奋）。但你查看employee表格时将看到新添加的记录。如果再次运行这个程序，你将看到一个错误信息，它显示雇员ID关键字值已存在。

```

$ ./update
ERROR: duplicate key violates unique constraint "employee_pkey"
Problem with command: ERROR: duplicate key violates unique constraint
"employee_pkey"

$ 

```

正如预期的那样，记录没有被添加。

2. 返回数据的命令

对于返回数据的SQL命令，你必须使用PQgetvalue()函数来获取返回的信息。该函数将获取的数据值放入一个字符串数据类型中，而不管字段值的实际数据类型是什么。对于整数或浮点数值，你可以使用标准的C函数（如用于整数值的atoi()或用于浮点数值的atof()）将字符串值转换为适当的数据类型。

PQgetvalue()函数允许你以任何顺序从结果集中检索数据，并不需要在结果集中逐一向前搜索每条记录。它的原型如下：

```
char *PQgetvalue(PGresult *result, int record, int column)
```

为了检索数据，你必须在这个函数中指定结果集result、所需的记录号record和字段编号column。记录号和字段编号都是从0开始。对于只产生一个记录的命令来说，记录号总是0。

代码清单6-7中的程序getvals.c演示了如何从一个SQL函数的结果集中获取并转换数据。

代码清单6-7 在PostgreSQL数据库中查询记录

```

/*
 *
 * Professional Linux Programming - retrieving values from queries
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

int main(int argc, char **argv)
{
    const char *conninfo;
    PGconn *conn;
    PGresult *result;
    char *time, *pi;
    float fpi;

    conninfo = "hostaddr = 127.0.0.1 dbname = test user = earl password = bigshot";
    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK)
    {
        printf("Unable to establish connection: %s",
               PQerrorMessage(conn));
        return 1;
    } else
    {
        result = PQexec(conn, "SELECT timeofday()");
        if (PQresultStatus(result) != PGRES_TUPLES_OK)
        {
            printf("Problem with command1: %s\n", PQerrorMessage(conn));
            PQclear(result);
            return 1;
        }
        time = PQgetvalue(result, 0, 0);
        printf("Time of day: %s\n", time);
        PQclear(result);

        result = PQexec(conn, "Select pi()");
        if (PQresultStatus(result) != PGRES_TUPLES_OK)
        {
            printf("Problem with command: %s\n", PQerrorMessage(conn));
            PQclear(result);
            return 1;
        }
        pi = PQgetvalue(result, 0, 0);
        fpi = atof(pi);
        printf("The value of pi is: %lf\n", fpi);
        PQclear(result);
    }

    PQfinish(conn);
    return 0;
}

```

在建立连接之后，我们调用PQexec()函数发送一个标准的SELECT SQL命令给服务器。请注意，为了检查结果集的状态，你必须使用PGRES_TUPLES_OK值，因为这个查询返回一个结果集。接着我们使用记录号0和字段号0来调用PQgetvalue()函数以获取结果集，因为这个查询只返回一个数据值。在第一个命令实例中，timeofday() PostgreSQL函数返回一个字符串值，它可以直接被放入一个字符串变量中。

在调用PQclear()函数重置了PQresult值之后，我们调用PQexec()函数发送另一个查询命令，并再次使用PQgetvalue()函数获取查询的文本结果集。这次，因为预期的数据类型是浮点数，所以我们使用atof()C语言函数将字符串值转换为一个浮点数变量。

运行这个程序将产生如下结果：

```
$ ./getvals
Time of day: Wed Oct 18 19:27:54.270179 2006 EDT
The value of pi is: 3.141593
$
```

3. 处理字段数据

程序getvals.c比较简单，因为我们知道结果集中只有一个字段的数据，从结果集中提取单个记录数据是非常简单的。但对于较复杂的结果集，必须确定返回的记录数和结果集中数据字段的顺序。如果在结果集中有多个记录，你必须遍历结果集，读取所有的数据记录。

为了确定结果集中的记录数和字段数，你可以分别使用PQntuples()函数和PQnfields()函数。

```
int recs;
int cols;
result = PQexec(conn, "SELECT * from employee");
recs = PQntuples(result);
cols = PQnfields(result);
```

一旦知道了结果集中的记录数和字段数，就可以轻松地遍历结果集以提取每个数据项了。

```
for (i = 0; i < recs; i++)
{
    ...
}
```

在提取字段数据时有一件事你必须小心。请记住PQexec()函数以文本格式返回所有的数据项。这意味着，如果你打算在程序中以另一种数据类型来使用表格数据，你必须使用标准的C语言函数将字符串转换为适当的数据类型：

- atoi()用于将字符串转换为一个整数值；
- atof()用户将字符串转换为一个浮点数值。

代码清单6-8中的程序getemployees.c演示了如何从结果集中提取每个字段数据元素。

代码清单6-8 从PostgreSQL数据库中获取多个记录

```
/*
 *
 * Professional Linux Programming - list all employees
 *
 */
#include <stdio.h>
#include <stdlib.h>
```

```
#include "libpq-fe.h"

int main(int argc, char **argv)
{
    const char *conninfo;
    PGConn *conn;
    PGResult *result;
    char *empid, *lastname, *firstname, *salary;
    float salaryval;
    int i;

    conninfo = "hostaddr = 127.0.0.1 dbname = test user = earl password = bigshot";

    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK)
    {
        printf("Unable to establish connection: %s",
               PQerrorMessage(conn));
        return 1;
    } else
    {
        result = PQexec(conn, "Select * from employee order by empid");
        if (PQresultStatus(result) != PGRES_TUPLES_OK)
        {
            printf("Problem with query: %s",
                   PQerrorMessage(conn));
            return 1;
        } else
        {
            printf("ID      Name          Salary\n");
            printf("-----\n");
            for(i = 0; i < PQntuples(result); i++)
            {
                empid = PQgetvalue(result, i, 0);
                lastname = PQgetvalue(result, i, 1);
                firstname = PQgetvalue(result, i, 2);
                salary = PQgetvalue(result, i, 3);
                salaryval = atof(salary);
                printf("%s %s $%.2lf\n", empid, lastname,
                       firstname, salaryval);
            }
            PQclear(result);
        }
    }
    PQfinish(conn);
    return 0;
}
```

开始时，程序getemployees.c和往常一样，连接到示例数据库，使用PQexec()函数发送一个简单的查询命令。结果集中的记录数通过PQntuples()函数来获取，然后程序进入一个for循环来遍历结果集中的记录。

在for循环中，每一次迭代处理结果集中的一个记录。对于每个记录，我们使用PQgetvalue()

函数来提取它的每个字段数据值。这个函数使用结果集、记录号（由for循环的计数值控制）和每个数据元素的字段编号作为其参数。

当然，这需要你知道查询命令将产生哪些字段以及它们的顺序。如果你不知道这些信息，可以使用PQfname()函数来找出字段编号和字段名称的对应关系。

每个字段数据值被分配给一个字符串变量，因为PQexec()只返回字符串值。由于salary字段值是一个浮点数据类型，所以我们使用atof()C语言函数将字符串转换为一个浮点数值。

```
$ ./getemployees
ID      Name        Salary
-----
100    Test, Ima    $35000.00
101    Test, Another $75000.00
$
```

6.3.6 使用参数

在许多情况下，我们需要使用同一个SQL命令，但提供不同的数据元素以执行多个查询。与重写每个PQexec()命令相比，可以使用PQexecParams()函数和一个被称为参数的特性来完成这个功能。参数允许你在SQL命令中应放置普通数据值的位置使用变量。每个变量由美元符号 (\$) 和数字编号（1为第1个变量，2为第2个变量，依此类推）组成。下面是一个使用参数的例子：

```
SELECT * from employee where lastname = $1 and firstname = $2
```

变量只用于数据值，不能为表格或字段名使用变量。PQexecParams()函数的完整格式如下：

```
PGresult *PQexecParams(Pgconn *conn, const char *command,
                      int nparms,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat)
```

正如预期的那样，必须声明函数中所用变量的值。在指定要使用的连接之后，你必须定义变量值。第一个要定义的值是nparms，它指定了所用变量的个数。接着是paramTypes的值。这个值是一个包含每个变量类型的OID的数组。也可以给它指定一个NULL值，这将强制PostgreSQL服务器为这个数据值所对应的字段使用默认的数据类型。

接下来的三个值分别指定变量值 (paramValues)、它们的长度 (paramLengths) 和变量的格式 (paramFormats)。这三个值都是数组。数组中的第一个元素指的是变量\$1，第二个元素是变量\$2，依此类推。

每个变量的格式值可以是文本模式 (0) 或二进制模式 (1)。如果paramFormats的值是NULL，那么PostgreSQL将假设所有的变量都是文本模式。如果变量的格式是文本模式，PostgreSQL将假设该变量是文本（或字符）数据类型，并将变量值转换为SQL命令需要的数据类型。

因此，你可以以文本字符串的形式指定一个整数或浮点数值，并将格式值设置为0，PostgreSQL将自动为你完成转换工作。如果你为变量使用文本模式，还可以将长度变量设置为NULL，因为PostgreSQL将自动确定文本字符串变量的长度。

如果将值的格式类型设置为二进制模式 (1)，就必须使用适当的数据类型来指定变量值，并以字节为单位指定值的长度。

结果格式的值 (resultFormat) 也很有趣。它允许你设置结果值的数据类型。它可以是文本模式 (0) 或二进制模式 (1)。如果使用文本模式，结果集将以文本值的形式返回，和PQexec()函数一样。对于二进制值必须执行适当的数据类型转换。如果使用二进制模式，结果集数据将以适当的二进制格式（整数或浮点数）返回。目前，结果格式值只能是两者之一，不能混合搭配文本和二进制模式的数据元素。

对PQexecParams()函数的设置容易使人困惑。对于初学者来说，最容易的做法是同时将输入和输出格式设置为文本模式，这样做减少了你必须收集并传递给函数的信息量。

代码清单6-9中的程序empinfo.c演示了如何同时为输入和输出数据值使用文本模式的参数。

代码清单6-9 在PostgreSQL程序中使用参数

```
/*
 *
 * Professional Linux Programming - get employee info
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include "libpq-fe.h"

int main(int argc, char **argv)
{
    const char *conninfo;
    PGconn *conn;
    PGresult *result;
    char *paramValues[1];
    const char *query;
    char empid[4];
    char *lastname, *firstname, *salary;
    float salaryval;

    conninfo = "hostaddr = 127.0.0.1 dbname = test user = earl password = bigshot";
    conn = PQconnectdb(conninfo);
    if (PQstatus(conn) != CONNECTION_OK)
    {
        printf("Unable to establish connection: %s", PQerrorMessage(conn));
        return 1;
    } else
    {
        query = "SELECT * from employee where empid = $1";
        while(1)
        {
            printf("\nEnter employee id: ");
            scanf("%3s", &empid);
            if (!strcmp(empid, "0"))
                break;

            paramValues[0] = (char *)empid;
            result = PQexecParams(conn, query, 1, NULL,
                                  (const char * const *)paramValues, NULL, NULL, 0);
        }
    }
}
```

```

if (PQresultStatus(result) != PGRES_TUPLES_OK)
{
    printf("Problem: %s", PQerrorMessage(conn));
    PQclear(result);
    PQfinish(conn);
    return 1;
} else
{
    if (PQntuples(result) > 0)
    {
        lastname = PQgetvalue(result, 0, 1);
        firstname = PQgetvalue(result, 0, 2);
        salary = PQgetvalue(result, 0, 3);
        salaryval = atof(salary);
        printf(" Employee ID: %s\n", empid);
        printf(" Name: %s,%s\n", lastname, firstname);
        printf(" Salary: %.2lf\n", salaryval);
        PQclear(result);
    } else
        printf(" Employee ID not found\n");
}
}

PQfinish(conn);
return 0;
}

```

paramValues指针被声明为一个数组。因为在SQL命令中只使用了一个参数，所以数组中只需要一个值。程序然后声明了一个查询字符串，它为WHERE子句中的数据元素使用了一个参数变量\$1。

接下来，程序进入一个无穷的while()循环以不断读取用户输入的值，直到用户输入一个表示停止的值为止。程序使用scanf C语言函数以允许用户输入雇员id值，从而在数据库中进行查找。

在用户输入需要的变量值之后，它被分配给paramValues数组，接着程序声明了PQexecParams()函数。因为输入值是文本模式，所以paramLengths和paramFormats值被设置为NULL。而且，resultFormat值也被设置为0以表明我们希望输出结果集数据也是文本模式。

在执行PQexecParams()函数之后，我们使用PQgetvalue()提取结果集中的数据。因为输出数据格式被设置为文本，所以salary值必须使用atof()C语言函数转换为浮点数。

编译程序之后，我们可以运行它以查看特定雇员的信息。输入0可以结束程序：

```

$ ./empinfo

Enter employee id: 100
Employee ID: 100
Name: Test, Ima
Salary: $35000.00

Enter employee id: 103
Employee ID not found

Enter employee id: 0
$
```

6.4 本章总结

在今天的使用环境中, Linux应用程序必须能够存储信息以供今后检索。有几种方法可用于在Linux应用程序中提供数据的持久性。其中最流行的方法需要使用数据库以允许对数据的快速存储和检索。有两种流行的方法用于在Linux应用程序中实现数据库功能。

Berkeley DB应用程序允许你直接在应用程序中编写数据库引擎。我们可以使用简易的库函数从简单的数据文件中插入、删除和检索数据。Berkeley DB函数库支持有序平衡树、哈希表、队列和记录号数据存储方式。

另一种方法使用一个外部数据库服务器来控制对数据库文件的访问。PostgreSQL开放源码数据库在一个免费数据库中提供了许多商业特性。它提供了libpq函数库, C和C++应用程序可以使用该函数库来访问数据库服务器。所有的访问被发送给服务器, 由服务器来控制哪些数据可以被访问。服务器集成了所有的数据库引擎功能。应用程序发送命令给服务器并获取由服务器返回的结果。通过使用一个中央数据库服务器, 应用程序可以很容易的在多个用户之间共享数据, 即使这些用户分散在网络的各处。

内核开发

为 Linux 系统开发软件有时需要对操作系统内核本身的内部设计有更深入的了解，尤其当你需要编写 Linux 设备驱动程序时更是如此。当然，还有一些其他原因也可能会促使你更深入地钻研 Linux 内核。有些开发者需要将 Linux 内核移植到新的硬件平台、修复错误、添加新功能、解决性能瓶颈或改善其在大型（或小型）系统中的可扩展性，他们理所当然需要剖析决定 Linux 运行机制的核心算法。

有时，人们并不是需要以任何方式来改变内核。Linux 内核是一个非比寻常的程序示例，它并不遵循你在编写应用程序代码时所使用的准则。当编写内核代码时，你需要明确处理所有的内存分配——没有人会为你释放内存——而且你需要应对非常实际的编程错误或可能导致整个系统崩溃的死/活锁。因此，许多人只是对他们所依赖的内核的内部设计抱有特别的好奇心。

无论你是因为什么原因对 Linux 内核的内部设计感兴趣，本章都将为你奠定坚实的基础。在本章中，你将学习编写内核与编写常规用户应用程序有何不同。你还将学习如何编译 Linux 内核的最新开发版本、如何跟踪公共邮件列表以及如何向社区提交你最初的补丁。本章的重点是使读者对 Linux 内核有广泛的了解，这将有助于你对后续章节的学习。

7.1 基本知识

开发 Linux 内核与专业 Linux 程序员其他的日常活动有些不同。Linux 内核不是一个常规的用户程序，它并不以通常的方式开始和结束。要想在不冒着导致整个系统崩溃危险的前提下在内核中测试自己的想法并不是一件容易的事情，而且当程序出错时，对它的调试也非常困难。尽管有这些潜在的缺陷，许多人还是乐意参与这个开发过程——当然也包括你。当你开始参与 Linux 内核的开发时，你会发现这一过程是非常值得的。

不论对 Linux 内核抱有什么样的个人兴趣，在开始学习之前你都需要掌握几件基本的事情。你需要快速地精通内核的配置和编译、测试补丁的应用以及最终结果的包装。这些工作都应该在你开始编写新代码之前掌握。在没有掌握基本知识之前就想进行具体的开发并希望一切都能正常工作是不切实际的。同样，你还需要了解开发过程的各个方面。Linux 内核的开发过程将在本章的最后进行介绍。

7.1.1 背景先决条件

在开始开发 Linux 内核之前，你需要满足几个背景先决条件。首先，你需要有一台开发机器以便在不干扰你的工作站正常操作的前提下测试你的内核代码。虚拟机在许多情况下工作得很好^①，但确

^① 从这个意义上说，人们并不指望虚拟机和真实机器是同一台机器。但使用一台基于虚拟硬件的虚拟机（不管是怎么实现的）永远不会和使用一台真实的机器完全一样。为此，我们认为投资一点小钱购买一台已用了3~5年的旧机器还是值得的。

实不要在你编写常规代码的同一台机器上进行内核开发，因为你可能会在内核开发中出现错误并导致整个系统崩溃，难道你真的希望每隔10分钟就重启一次系统吗？使用一台旧的测试机器的做法应更为可取。

你应该配备一个拥有较多内存（内核编译是内存密集型操作，如果减少内存，它的编译时间将会更长）和尽可能快速的CPU（你可以证明这样购买是合理的或你可以买得起）的开发机器。不要过分强调处理器的速度，更为重要的是这台机器不会将每一秒钟都花费在读写磁盘上，内存是主要的瓶颈所在，请你确保有足够的可用资源。如果你是在一家商业公司工作，请以提高生产率为由申请一台新的机器——你决不会错的。

为Linux内核开发代码是一个磁盘密集型的操作。你需要很大的空间来存储所有的测试内核，如果你想要通过工具（我们将在本章的后面对它进行介绍）来跟踪上游内核的每日更新，你甚至需要比这更多的空间。作为一个指导原则，一个完整编译的Linux 2.6内核可能需要多达2 GB的空间来存储源代码和二进制目标文件——如果你决定启用所有可能的配置选项（不要这样做——这将极大地增加编译时间），需要的空间将会更多。如果你的开发树还想跟踪上游开发的每日更新，所需的空间还会增加。如今磁盘的价格已经越来越便宜，所以你一定要确保有充足的空间用于内核开发。

如果你参与内核开发并工作在一台PC工作站上，你的工作站至少需要拥有一个2 GHz的处理器、1 GB以上的内存（除去其他系统开销）和一个大型的专用存储区域用于所有的测试内核编译，它们所占用的空间可以很轻易地达到50 GB甚至更多。幸运的是，这些需求对于如今的用户来说都不是难以达到的——这也是为什么它们构成了这个建议的基础的原因。当然，我们也可以使用一台老式的386机器或一台Sun IPX^①来编译Linux内核——如果你能够花费一个星期的时间来编译内核并有无限的耐心。做不到这一点，就请使用一台满足建议要求的机器。

7.1.2 内核源代码

整个Linux内核源代码树大约由10 000个头文件、9 000个C语言源文件、800个硬件体系结构相关的汇编语言文件和其他各种链接器、Makefile文件以及用于将所有这一切整合到一起的辅助基础设施组成。要想完全明白Linux内核源代码树的全部内容几乎是不可能的——而且也没有人懂内核的所有内容。这并不重要，重要的是你理解源代码树的布局并知道到哪里去查找影响内核工作的部分。

事实上的Linux内核源代码“官方”网站是kernel.org。你将在该网站上找到所有发布过的Linux内核的拷贝（包括它们源代码的签名校验和，除了那些仅因为其历史价值而存在的早期内核以外），以及成千上万由使用各种机器的开发者所贡献的补丁。不论何时，当你需要找到和内核相关的事物时，这个网站都是你应该最先光顾的地方，你将在该网站上找到许多指向其他信息的链接。

在kernel.org网站的首页上，你将看到指向最新官方“稳定版本”Linux内核的链接，在写作本书时它的格式是2.6.x.y（x指的是主修订版本号，而可选的y是指由“稳定版本”内核小组应用的专用安全（或其他非常紧急的）勘误版本号）。你还将看到指向最新-mm开发内核的链接和有关其他旧内核的信息。不过不要过于费心去了解为什么还没有2.7版本的内核或什么时候会开始2.8稳定版本内核的开发。没有人知道更多这方面的信息。

① 这让作者回忆起了他十几岁时在一台老式386机器上编译测试内核的时光。那时候，编译内核是你在晚上睡觉前开始做的事情。对于Sun IPX来说，它需要花费超过3天的时间来安装Debian，所以编译内核总是一件很痛苦的事情。

为了开始内核开发，你需要从kernel.org网站的首页上直接下载最新的2.6.x.y版本的内核。请下载有着最高修订版本号（例如kernel 2.6.18）的.tar.bz2^①内核（它大约有40 MB大小），并使用以下命令将它释放到一个新的工作目录中。

```
$ tar xvzf linux-2.6.18.tar.bz2
linux-2.6.18
linux-2.6.18/COPYING
linux-2.6.18/CREDITS
linux-2.6.18/Documentation
linux-2.6.18/Documentation/00-INDEX
linux-2.6.18/Documentation/BUG-HUNTING
linux-2.6.18/Documentation/Changes
linux-2.6.18/Documentation/CodingStyle
...
...
```

为简洁起见，上面的命令列表已被截短——完整的文件列表将包含成千上万的文件。

请特别留意Documentation目录中的文件。并非所有这些文件的内容都和当前的内核版本一样是最新的——如果你有时间，并且也试图用任何方式帮助解决内核中的一些问题，那么你可以帮忙改进这些文件的内容（如果你这样做，2.6版内核的维护者Andrew Morton等人将非常感谢你）。但是，这个目录确实是一个你获取关于内核额外信息的好地方。

你将在该目录下发现一篇很好的关于内核编码风格指南的文档CodingStyle，以及如何提交关于内核中许多内部实现的纯技术方面的缺陷和概述文档的指南。当然，你也可以随时编写自己的文档。

1. 浏览内核源代码

Linux内核源代码总是拥有如下这样的顶层目录结构：

arch/	crypto/	include/	kernel/	mm/	scripts/
block/	Documentation/	init/	lib/	net/	security/
COPYING	drivers/	ipc/	MAINTAINERS	README	sound/
CREDITS	fs/	Kbuild	Makefile	REPORTING-BUGS	usr/

这些目录在过去的10多年中逐渐有机地结合在一起，但大量有趣的核心内核功能仍保留在arch、kernel和mm目录中。如今大多数的内核驱动程序已位于drivers目录中，但还有几个明显例外，例如，sound和net目录还保留在顶层目录中，这主要是由于各种历史原因造成的。表7-1对每一个顶层目录所包含的内容进行了概述。

表7-1 顶层目录所包含的内容

目 录 名	目 录 内 容
arch	硬件体系结构相关文件。包含的体系结构有arm、i386、powerpc等。这是底层启动代码以及支持处理器特征所需关键代码所在的位置。这些文件主要是用汇编语言编写的。请查看汇编文件head.S以找到底层内核入口
block	支持硬盘等块设备。这个目录包含用于表示I/O操作请求的结构bio和bio_vec的Linux内核2.6版本的实现，以及绝大多数在Linux内核中支持硬盘所需的代码

① 我们已在第2章～第4章介绍tarball文件时对.bz2文件进行了解释，但你将在kernel.org网站上找到它在Linux内核中使用的一个说明。

(续)

目 录 名	目 录 内 容
crypto	加密库函数。在美国法律修改之前，在一个可能出口海外的内核中使用美国的强加密算法是不可能的。而这在不久前有所改变，现在强加密算法已被编译进内核
Documentation	Linux内核提供的文档。请记住，没有得到商业资助的文档小组在维护这些文档，所以它们并不总是符合当前内核的现状。只要你记住这一点，你就可以很好地利用它们
drivers	针对各种可能设备的设备驱动程序
fs	Linux内核支持的每种文件系统的实现代码
include	Linux内核提供的头文件，分为体系结构相关的arch头文件和通用的linux头文件。我们将在本章的后面提及include/linux/sched.h头文件
init	在非常底层的体系结构相关代码运行结束之后负责初始化内核的高层启动代码。内核调用这个目录中的底层函数以建立内核虚拟内存（分页）支持，开始运行进程（第一个进程或任务是init），最终exec()进入init进程以开始运行常规的Linux启动脚本和第一个用户程序
ipc	实现在Linux内核中使用的各种进程间通信（IPC）原语的代码位置
kernel	高层核心内核函数。这个目录包含Linux进程或任务的调度程序，以及各种支持代码，如printk()通用例程，它允许内核开发者在内核运行时发送消息给内核日志进程(klogd根据配置情况将接收到的每条信息记录到/var/log目录中)
lib	放置库例程，如CRC32校验和计算和用于内核中各处的各种高层库例程代码
mm	用于支持虚拟内存的Linux实现的内存管理例程。它们与体系结构相关的代码结合使用
net	包含Linux内核支持的每个网络标准的实现代码。它并不包含网络设备驱动程序，这些驱动程序位于drivers/net目录或类似的位置中
scripts	放置用于配置和编译Linux内核的各种脚本
security	Linux包括对SELinux（安全增强型Linux）功能的支持，它原本是由美国的国家安全局（NSA）开发的。这些代码虽已经过严格的审核，但因其本质，偶尔还是会成为媒体关注和恐惧的对象
sound	包含最新Linux内核使用的ALSA（高级Linux音频体系结构）音频子系统的实现代码。旧的被称为OSS的音频系统正在逐渐被淘汰
usr	包含用于建立初始化内存盘（几乎被所有的Linux厂商用来引导它们的发行版环境）的各种支持工具，等等

2. 内核版本

Linux内核以往遵循一个简单的数字命名系统。在美好的往昔岁月中，内核的版本以a.b.c的形式命名，其中“a”表示主版本号，到目前只产生过两个主版本号；“b”表示次版本号，如果它是奇数，表示这是一个开发版本的内核，如果它是偶数，则表示这是一个稳定版本的内核；“c”表示一个特定内核开发系列的发行版本号。例如，以前我们可以通过内核的版本号2.3.z来确定它是一个不稳定的开发版本，而2.4.z版本的内核则是稳定的（虽然一些早期的2.4版内核有虚拟内存方面的问题需要解决）。但是，现在内核已不再用这样一种简单的方式命名了。

继Linux内核2.4版本推出之后，人们将大量的工作投入2.5系列内核的开发。这一系列内核对阻塞I/O和可扩展性做了大量的改进^①，并且还进行了大量其他的各种开发。在经过一段相当长的开发时

^① 这是第一个真正提供了与大型计算机公司进行重要开发合作的版本系列，这些公司因客户的需求，需要在其大型机上运行Linux内核。

间之后，Linux内核的2.6版本最终发布并迅速被誉为Linux开发历史中的一种突破（这的确是事实）。从Linux内核的2.6版本开始，Linus Torvalds改变了数字式的命名方式。

以前，Linux内核开发的一个问题是需要花费大量的时间将一个新的特征加入到内核的稳定版本中。将一个位于开发版本中的特征加入稳定版本大致需要花费6~12个月的时间，而有的特征甚至从来没有在稳定版本中出现过。例如，Linux 2.4内核系列对许多现代外围设备缺乏很好的支持，虽然2.5内核可以迅速地获得对这些设备的支持，但2.6内核却花费了一年多的时间才能支持这些特征。许多厂商因此将新的特征加入旧的内核中，但这种情况肯定不是理想的。

人们认为在2.6内核的使用期间，内核的开发过程一直很顺利，因此没有必要启用一个新的开发内核。开发者彼此也对整个过程很满意，所以Linux内核的开发将在2.6内核的整个生命周期中继续在主线Linux内核中进行，而这也正是目前Linux内核开发所处的情况。除非出现了Linux2.7内核或开始了3.0内核的开发，否则这一现状将继续维持。

3. 厂商内核

Linux内核被世界各地许多不同的个人和组织广泛使用。他们中的许多都从Linux的发展和普及中获得了好处（也有一些组织乐于看到对Linux的支持）。因此，许多组织都在开发他们自己的Linux内核版本。例如，你几乎肯定会在自己的Linux工作站上运行一个标准的Linux发行版。这个发行版就可能会携带它自己的内核——一个“厂商内核”，它将包括厂商认为对改善用户整体体验有帮助的所有附加特性和补丁。厂商将对它的官方内核提供支持。

虽然厂商内核非常有用，因为他们为了支持特定的系统和工作负荷对内核进行了额外的开发和测试，但当你开始自己的内核开发时，应谨慎使用厂商内核。你应尽可能快地抛弃厂商内核而改为使用上游的“官方”Linux内核源代码，只有这样才能融入内核开发社区，并提出有用的问题。如果你试图使用厂商内核来编写自己设想的设备驱动程序，没有人会向你伸出援手，你的收件箱很快就会提醒你注意这个事实。

当然也有例外的情况，一些厂商销售的产品中包括了针对特定类型系统的Linux内核，通常是针对需要特定内核集和补丁来充分利用硬件环境的嵌入式设备。如果你是为一些奇特的新设备编写设备驱动程序，你当然希望最终能支持这一领域中的各种厂商内核。但是，如果你期望从其他许多内核开发者那里得到帮助——他们已准备好并愿意和你交谈（你当然希望是这样），你应该从一个全新的上游内核源代码开始你的内核开发。

7.1.3 配置内核

在一个刚解压缩释放的Linux内核源代码目录中，你将看到类似于上一节中显示的内核源代码顶层目录结构，但现在你还不能真正开始编译Linux内核。只有当你确定内核需要支持哪些特征之后，你才能开始编译内核。Linux有一个非常灵活的配置选项集，它允许你启用或禁用某些核心功能以及许多可能在你的系统中并不需要的可选设备驱动程序。你需要在一开始时花费一些时间浏览内核的配置选项，并注意查看大多数配置选项提供的“help screen”（帮助屏幕）信息。

为了配置Linux内核，请进入刚刚释放内核源代码所建立的目录，并使用基于文本的配置工具menuconfig来配置内核，如下所示：

```
$ make menuconfig
```

Linux内核配置为一个给定的体系结构提供了一套合理的默认配置，所以你可能对内核呈现的基本配置已感到很满意。但你需要做的一件事情是确保你的测试机器所要支持的任何硬件设备的驱动程序都已在内核配置中启用，否则，当你测试新内核时，内核将启动失败。不要因为有这么多配置选项而感到灰心，你需要经过几次尝试才能搞清楚它们之间的关系，而且你也只有经过这样一个艰难的过程才能真正掌握内核的配置。

对配置系统的浏览可能需要一点时间来习惯，但你最终肯定会熟悉它。如果你不清楚应该做什么了，请使用帮助选项来理解菜单系统的不同部分是如何结合在一起的。如果你不习惯使用图形化配置工具，那你可以直接手工编辑内核配置——它不过是一个填满选项的文本文件，其中的选项使用在编译时定义的C语言常量和宏来定义。内核配置的文本表示被保存在内核顶层目录下的.config文件中。但与直接手工编辑这个文件相比，我们更建议你使用make config命令。

如果你决定手工编辑.config文件，我们强烈建议你运行make oldconfig命令来清楚地输入修改，如果你希望使用目前运行在你的系统中的厂商提供的内核配置作为内核配置的起点，你也同样需要这样做。厂商提供的内核配置通常位于它们的Linux内核开发软件包中。

下面是使用make config命令进行内核配置的一个示例：

```
$ make config
scripts/kconfig/conf arch/powerpc/Kconfig
#
# using defaults found in .config
#
*
* Linux Kernel Configuration
*
64-bit kernel (PPC64) [N/y/?]
*
* Processor support
*
Processor Type
> 1. 6xx/7xx/74xx (CLASSIC32)
  2. Freescale 52xx (PPC_52xx)
  3. Freescale 82xx (PPC_82xx)
  4. Freescale 83xx (PPC_83xx)
  5. Freescale 85xx (PPC_85xx)
  6. AMCC 40x (40x)
  7. AMCC 44x (44x)
  8. Freescale 8xx (8xx)
  9. Freescale e200 (E200)
choice[1-9]:
Altivec Support (ALTIVEC) [Y/n/?]
Symmetric multi-processing support (SMP) [N/y/?]
*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers (EXPERIMENTAL) [Y/n/?]
*
```

```
* General setup
*
Local version - append to kernel release (LOCALVERSION) []
...
```

为简洁起见，我们对输出进行了精简。

请注意每个选项是如何依次显示的——你只需要依次对数百个（或上千个）可能的配置选项回答“y”（是）或“n”（否），具体的配置选项将根据体系结构、平台和你已选择的其他选项有所不同。如果你喜欢这样做，或真的想确保你检查了每个可能的选项，那你可以采用这样一种老式的方法。但在几年之后，你将会感到很高兴，可以用到下面介绍的这些替代命令。

对于那些不喜欢 curses（基于文本）图形的用户来说，他们可能更倾向于使用内核编译系统中的 make xconfig 命令。这个命令将向你显示一个为 X 视窗系统编写的图形化配置工具^①。或者，如果你已购买了一个商业 Linux 内核开发产品（例如，一个使用 Eclipse 图形化开发环境 IDE 的产品），那么你将可以使用一个更复杂的内核配置工具集。但不管你使用的是哪一种配置工具，其最终产生的结果都是一样的——一个填满选项的文本文件 .config。

一旦配置好了内核，配置信息将被保存到内核配置目录下的 .config 文件中。你需要备份该文件以便以后每次解开一个新内核或使用 make mrproper 命令清理内核源代码之后使用。一旦有了一个可以正常工作的内核配置，就可以将它输入到新的内核源代码配置文件中使用，只需将它拷贝为新的内核源代码目录下的 .config 文件，并运行 make oldconfig 命令以导入配置文件即可。当从一个内核版本转移到另一个内核版本时，内核开发者通常都喜欢使用这个机制，因为它避免了对内核的重新配置^②。

下面是从一个有效的内核配置文件 .config 中摘录的一小部分内容：

```
# ...
# Automatically generated make config: don't edit
# Linux kernel version: 2.6.17
# Mon Jun 19 18:48:00 2006
#
# CONFIG_PPC64 is not set
CONFIG_PPC32=y
CONFIG_PPC_MERGE=y
CONFIG_MMU=y
CONFIG_GENERIC_HARDIRQS=y
CONFIG_RWSEM_XCHGADD_ALGORITHM=y
CONFIG_GENERIC_HWEIGHT=y
CONFIG_GENERIC_CALIBRATE_DELAY=y
CONFIG_PPC=y
CONFIG_EARLY_PRINTK=y
```

1. 厂商内核配置

根据所使用的 Linux 发行版的不同，你可能会发现系统在 /boot 目录中提供了默认的内核配置选项

-
- ① 很显然，你需要运行 X 来使用这个命令。如果你是在一个安装了 X 视窗系统的远程机器上使用 SSH 进行连接，请使用 ssh 命令的 “-X” 选项标记来将配置界面转发到远程 X 视窗。
 - ② 除非一个新的选项被添加进来，在这种情况下，你只会看到那些被新添加的或被修改的选项。请注意对配置选项的修改可能会移除你默认启用的 IDE/SATA（或甚至可能是 SCSI）磁盘支持而导致系统不能启动。

集，如配置文件config-2.x.y-a.b_FC6（在使用Fedora内核的情况下），而且它还可以被轻松地导入到新的内核源代码配置文件中。大多数发行版都提供了描述它们的官方内核编译过程的文档，并会告诉你到哪里去查找用于编译内核的配置文件。很显然，这个配置文件可能并不会完全包括你正在编译的内核的所有特征，但没有关系，make oldconfig命令将尽其最大努力帮你解决这个问题。

在一些最新的Linux内核中，你将发现内核配置可以在/proc/config.gz文件中找到。虽然这消耗了一些内核空间，但在开发内核时，在内核配置中启用这个选项是一个好主意，因为这样就可以非常容易地看到用于编译特定测试内核的配置选项。一个在/proc目录中的压缩内核配置可以很轻易地被安装到一个新的内核源代码中，你只需使用gunzip命令并将解压缩后的结果文件拷贝到新的内核源代码目录中即可。

2. 为交叉编译进行配置

当Linux内核需要在一个处理能力比我们所使用的工作站低很多的嵌入式设备上时，使用交叉编译环境来编译内核是十分常见的。在这种情况下，使用交叉编译不仅节省时间，而且在编译内核时并不需要熟悉你所使用的编译机器——可以选择你能够访问的最快的机器，而不用在编译时关心该机器的体系结构或平台。

为了交叉编译Linux内核，我们需要设置环境变量CROSS_COMPILE以使它的值作为编译过程中你将使用的gcc和二进制工具集命令的前缀。例如，如果你有一个通过如powerpc-eabi-405-gcc这样的命令来支持PowerPC处理器的GNU工具链，就需要在执行本章介绍的内核配置和编译步骤之前，将环境变量CROSS_COMPILE设置为powerpc-eabi-405-。还需要确保环境变量ARCH被设置为适当的编译架构，如powerpc：

```
$ export CROSS_COMPILE="powerpc-eabi-405-"
$ ARCH=powerpc make menuconfig
$ ARCH=powerpc make
```

Linux 2.6 Kbuild系统经过优化以支持各种编译目标。你会看到它明确地显示每个文件是使用交叉编译器编译的还是使用主机编译器编译的。但仅仅在几年之前，使用交叉编译器编译一个Linux内核还被认为只是一小群嵌入式设备爱好者所玩的一个不寻常的实验性游戏。

7.1.4 编译内核

编译Linux内核其实非常简单，只需使用GNU make命令即可，然后内核的Kbuild系统（它位于make之上）将对编译整个内核系统所需的工具做出适当的调用。一次成功的编译可能需要花费较长的时间（在一个非常慢的机器上需要几个小时，即使在一个非常快的工作站上也需要几十分钟——在一些拥有巨量内存的非常大型和速度极快的机器上大约需要7秒钟）。所以，在通常情况下，不论内核源代码树处于什么样的状态，编译过程都会充分利用可以使用的所有资源。

经常性地进行编译和测试是一个好主意。现在有许多回归测试套件可以使用，但即便是一次简单的编译和启动周期也足以让你知道你的做法是否正确。那些幸运的拥有自动化编译和测试系统的用户将因此受益，当然这也给Linux厂商带来了额外的收入。

在要求编译系统开始编译内核之前，你需要先拥有一个已配置好的Linux内核。使用make命令启动编译过程，如下所示（只显示了编译过程的开始的几行信息）：

```
$ make
CHK      include/linux/version.h
UPD      include/linux/version.h
SYMLINK  include/asm -> include/asm-powerpc
SPLIT    include/linux/autoconf.h -> include/config/*
CC       arch/powerpc/kernel/asm-offsets.s
GEN      include/asm-powerpc/asm-offsets.h
HOSTCC   scripts/genksyms/genksyms.o
SHIPPED  scripts/genksyms/lex.c
...
```

为简洁起见，对命令的输出进行了精简。

你将清楚地看到在编译的每个阶段都执行了哪种类型的动作，这要感谢Kbuild系统将动作类型添加到每一行输出的开头。命令的输出默认使用的是一种新的简短显示模式，它没有显示每个阶段所使用的完整的编译命令。如果你想获得更详细的输出信息，需要为make命令加上V=1选项。

下面这个例子与前面的例子使用了相同的命令，不过这次启用了冗长模式。

```
$ make V=1
rm -f .kernelrelease
echo 2.6.17-rc6 > .kernelrelease
set -e; echo '  CHK      include/linux/version.h'; mkdir -p include/linux/;      if [ `echo -n "2.6.17-rc6" | wc -c` -gt 64 ]; then echo "'2.6.17-rc6' exceeds 64
characters' >&2; exit 1; fi; (echo '#define UTS_RELEASE "\'2.6.17-rc6\'"; echo
#define LINUX_VERSION_CODE `expr 2 \'* 65536 + 6 \\'* 256 + 17`; echo '#define
KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))'; ) <
/data/work/linux_26/linus_26/Makefile > include/linux/version.h.tmp; if [ -r
include/linux/version.h ] && cmp -s include/linux/version.h
include/linux/version.h.tmp; then rm -f include/linux/version.h.tmp; else echo '
UPD      include/linux/version.h'; mv -f include/linux/version.h.tmp
include/linux/version.h; fi
  CHK      include/linux/version.h
if [ ! -d arch/powerpc/include ]; then mkdir -p arch/powerpc/include; fi
ln -fsn /data/work/linux_26/linus_26/include/asm-ppc arch/powerpc/include/asm
make -f scripts/Makefile.build obj=scripts/basic
  SPLIT    include/linux/autoconf.h -> include/config/*
mkdir -p .tmp_versions
rm -f .tmp_versions/*
make -f scripts/Makefile.build obj=.
mkdir -p arch/powerpc/kernel/
  gcc -m32 -Wp,-MD,arch/powerpc/kernel/.asm-offsets.s.d -nostdinc -isystem
/usr/lib/gcc/ppc64-redhat-linux/4.1.0/include -D__KERNEL__ -Iinclude -include
include/linux/autoconf.h -Iarch/powerpc -Iarch/powerpc/include -Wall -Wundef -
Wstrict-prototypes -Wno-trigraphs -fno-strict-aliasing -fno-common -Os
-fomit-frame-pointer -g -msoft-float -pipe -Iarch/powerpc -ffixed-r2 -mmultiple -
mno-alitvec -funit-at-a-time -mstring -mcpu=powerpc -Wa,-maltivec
-Wdeclaration-after-statement -Wno-pointer-sign      -D"KBUILD_STR(s)="#$" -
D"KBUILD_BASENAME=KBUILD_STR(asm_offsets)"
-D"KBUILD_MODNAME=KBUILD_STR(asm_offsets)" -fverbose-asm -S -
arch/powerpc/kernel/asm-offsets.s arch/powerpc/kernel/asm-offsets.c
```

```

mkdir -p include/asm-powerpc/
(set -e; echo "#ifndef __ASM_OFFSETS_H__"; echo "#define
__ASM_OFFSETS_H__"; echo "/* DO NOT MODIFY.*/"; echo " * This
file was generated by /data/work/linux_26/linus_26/Kbuild"; echo " *"; echo " */";
echo ""; sed -ne "/^-->/{s:^-->\([^\n]*\)\[\\$#\]*\[^\n]*\)\ \(.*):#define \1 \2 /* \3
*/; s:->::; p;}" arch/powerpc/kernel/asm-offsets.s; echo ""; echo "#endif" ) >
include/asm-powerpc/asm-offsets.h
make -f scripts/Makefile.build obj=scripts
make -f scripts/Makefile.build obj=scripts/genksyms
gcc -Wp,-MD,scripts/genksyms/.genksyms.o.d -Wall -Wstrict-prototypes -O2 -fomit-
frame-pointer -c -o scripts/genksyms/genksyms.o scripts/genksyms/genksyms.c
gcc -Wp,-MD,scripts/genksyms/.lex.o.d -Wall -Wstrict-prototypes -O2 -fomit-frame-
pointer -Iscripts/genksyms -c -o scripts/genksyms/lex.o scripts/genksyms/lex.c
...

```

正如你所看到的，Kbuild系统使用了冗长输出选项后给出的输出结果确实非常详细，所以你通常在编译内核时不会启用冗长模式（否则，在这么多输出信息中，你很容易陷入只见树木不见森林的状态）。如果命令失败了，可以执行make help命令来查看一系列的make目标、可选标记和对可用编译选项的简要说明。

下面是make help命令输出的编译选项列表：

```

$ make help
Cleaning targets:
clean           - remove most generated files but keep the config
mrproper        - remove all generated files + config + various backup files

Configuration targets:
config          - Update current config utilising a line-oriented program
menuconfig      - Update current config utilising a menu based program
xconfig         - Update current config utilising a QT based front-end
gconfig         - Update current config utilising a GTK based front-end
oldconfig       - Update current config utilising a provided .config as base
randconfig      - New config with random answer to all options
defconfig       - New config with default answer to all options
allmodconfig   - New config selecting modules when possible
allyesconfig   - New config where all options are accepted with yes
allnoconfig    - New config where all options are answered with no

Other generic targets:
all              - Build all targets marked with [*]
* vmlinux        - Build the bare kernel
* modules        - Build all modules
modules_install - Install all modules to INSTALL_MOD_PATH (default: /)
dir/             - Build all files in dir and below
dir/file.[ois]   - Build specified target only
dir/file.ko     - Build module including final link
rpm              - Build a kernel as an RPM package
tags/TAGS       - Generate tags file for editors
cscope          - Generate cscope index
kernelrelease   - Output the release version string
kernelversion   - Output the version stored in Makefile

Static analysers

```

```

checkstack      - Generate a list of stack hogs
namespacecheck - Name space analysis on compiled kernel

Kernel packaging:
rpm-pkg          - Build the kernel as an RPM package
binrpm-pkg       - Build an rpm package containing the compiled kernel
                   and modules
deb-pkg          - Build the kernel as an deb package
tar-pkg          - Build the kernel as an uncompressed tarball
targz-pkg        - Build the kernel as a gzip compressed tarball
tarbz2-pkg       - Build the kernel as a bzip2 compressed tarball

Documentation targets:
Linux kernel internal documentation in different formats:
xmldocs (XML DocBook), psdocs (Postscript), pdfdocs (PDF)
htmldocs (HTML), mandocs (man pages, use installmandocs to install)

Architecture specific targets (powerpc):
* zImage           - Compressed kernel image (arch/powerpc/boot/zImage.*)
install          - Install kernel using
                   (your) ~bin/installkernel or
                   (distribution) /sbin/installkernel or
                   install to ${INSTALL_PATH} and run lilo
*_defconfig      - Select default config from arch/powerpc/configs

cell_defconfig    - Build for cell
g5_defconfig     - Build for g5
iseries_defconfig - Build for iseries
maple_defconfig   - Build for maple
mpc834x_sys_defconfig - Build for mpc834x_sys
mpc8540_ads_defconfig - Build for mpc8540_ads
pmac32_defconfig - Build for pmac32
ppc64_defconfig   - Build for ppc64
pseries_defconfig - Build for pseries

make V=0|1 [targets] 0 => quiet build (default), 1 => verbose build
make O=dir [targets] Locate all output files in "dir", including .config
make C=1  [targets] Check all c source with $CHECK (sparse)
make C=2  [targets] Force check of all c source with $CHECK (sparse)

Execute "make" or "make all" to build all targets marked with [*]
For further info see the ./README file

```

正如你所看到的，选项列表相当全面并值得研究。

7.1.5 已编译好的内核

一旦内核编译成功完成，输出的内核将被放置到一个架构（和平台）相关的位置。在Intel IA32 (x86) 计算机上，这个默认位置是有点不寻常的arch/i386/boot/bzImage文件。这样做是有历史原因的，但你很快就会习惯它了。这个文件可以用来替换（最好能和旧的内核文件并存）已有的 /boot/vmlinuz-2.6.x.y文件。此后，你需要将这个文件拷贝到正确的位置并更新系统的启动加载器（如Intel机器上的lilo或grub，当然还有许多其他的启动加载器）来使用新的内核。

不论二进制内核输出文件位于系统中的什么位置，在你可以测试——启动新内核之前，你还需要准备一些支持文件。内核编译过程将在内核源代码的顶层目录中创建一个名为System.map的文件，你需要保存它并最终将它拷贝到正确的位置，一般是放在公认的/boot目录中。System.map文件包含刚刚编译的内核中的每个公开输出符号的拷贝。这些符号在调试时偶尔会用到，系统工具top在确定它的一个可选输出显示列的内容时也要用到这些符号——即一个指定进程当前正在内核的哪一个函数中休眠，这里指的是WCHAN输出显示列。

应该对刚编译好的内核二进制文件和与之相应的System.map文件进行备份，你可能还想备份在配置内核时包含你所选择配置选项的.config文件。同时对目前正在运行的内核也需要进行备份，以免你编译的测试内核不能正常启动系统。

不要马上删除内核编译目录，它还包含需要放到你的测试系统中的所有内核模块。关于内核模块的更详细内容请参见下一节。

7.1.6 测试内核

在一个典型的Linux工作站上，对内核进行测试非常简单，只需要先将二进制内核映像（在将它更名为符合vmlinuz-2.6.x.y语法格式的文件名后）、System.map文件和可选的在编译过程中使用的config文件拷贝到正确的位置。然后，你需要编辑Linux启动加载器的配置文件以确保系统在默认情况下启动这个新内核。在很久以前，每次写入新内核之后，我们还需要重新运行启动加载器的配置程序才行（因为启动加载器依赖于磁盘上内核的硬编码位置——每次写入文件时，该位置都会发生改变，即使最后一次写入的是同一个文件也是如此）。

如今，大多数常规的基于PC的Linux系统使用GRUB启动加载器，它有一个名为/boot/grub/menu.lst的配置文件，可以通过编辑该文件来修改默认的启动内核。基于PowerPC的工作站和服务器通常会使用/etc/yaboot.conf文件来重新配置yaboot启动加载器。而专用的嵌入式设备通常使用Das U-Boot通用启动加载器来直接启动Linux（它支持直接从闪存中启动Linux）。不论你的系统应该采取什么样的适当动作，在真正启动新内核之前通知Linux是非常容易的——在启动加载器配置中保留几个旧的内核入口以预备退路也是非常容易的。

大多数情况下，当测试Linux内核时，建议你建立一个单独的测试机器。在可能的情况下它应该运行一个与你的开发机器类似的Linux发行版。最好的做法是你有一个NFS服务器（例如，在你的开发工作站上），并且它为开发机器和测试机器输出一个共享文件系统。然后你就可以轻松地将新内核拷贝给测试机器，而不必使用刻录CD、USB存储设备或老式的软盘。

可以通过在线文档或（希望如此）你的发行版文档找到NFS服务器配置的信息。有各种指南介绍了这一主题，包括内核中的源代码文档，所以在这里我们就不详细介绍它了。使用NFS启动系统确实是测试内核的最佳途径之一——除非你真的非常需要在系统启动时（而不是在启动之后）拥有一个有着本地文件系统的本地磁盘。

你可以使用如下命令将新内核和System.map文件拷贝到你的测试系统中的正确位置：

```
$ cp /mnt/nfs_on_dev_box/linux-2.6.x.y/arch/i386/boot/bzImage /boot/vmlinuz-2.6.x.y
$ cp /mnt/nfs_on_dev_box/linux-2.6.x.y/System.map /boot/System.map-2.6.x.y
```

一旦内核准备好了，你就需要使用make modules_install命令来安装系统所需的任何内核模块

(以及刚编译的其他任何文件)。这个命令将把新编译的内核模块拷贝到标准的目录位置 /lib/modules/2.6.x.y，并为它们的使用做好准备(在你再次运行depmod之后，有时该命令会在系统启动时自动运行以使得模块在机器重启之后就可以使用，但这取决于发行版的选择)。

如果在测试机器上使用了通过网络装载的文件系统，而且在该机器本身上还没有安装任何可装载的内核模块，并且你希望改变内核模块安装的位置，那么你就要给make命令传递一个可选的环境变量 INSTALL_MOD_PATH以设置合适的安装路径。如果你在一台共享(或输出)你的测试机器的根NFS文件系统的机器上编译内核，这一点就非常重要，因为在这种情况下，你当然不希望将内核模块安装到开发机器上。

下面这个例子将内核模块安装到你的本地系统的/lib/modules目录中(使用sudo命令以避免以root用户身份登录系统)：

```
$ sudo make modules_install
Password:
INSTALL arch/powerpc/oprofile/oprofile.ko
INSTALL crypto/aes.ko
INSTALL crypto/anubis.ko
INSTALL crypto/arc4.ko
INSTALL crypto/blowfish.ko
INSTALL crypto/cast5.ko
INSTALL crypto/cast6.ko
INSTALL crypto/crc32c.ko
INSTALL crypto/crypto_null.ko
INSTALL crypto/deflate.ko
...
...
```

这个命令的输出在显示了几个模块之后就被截断了。一个完整的配置模块集可能包含数百个模块，总容量超过150MB。

可以运行depmod以使得modprobe命令可以装载这些内核模块——这样你就不需要对一个具体的.ko模块文件使用更特殊(也更不灵活)的insmod/rmmod命令了。modprobe命令还会自动解决模块之间的依赖关系。

下面显示的是如何针对新安装的Linux 2.6内核模块正确地运行depmod命令：

```
$ depmod -a 2.6.x.y
```

上面命令中的2.6.x.y应被替换为你刚编译的内核的具体版本号。

初始化内存盘：initrd

许多现代Linux发行版都依赖于一个初始化内存盘来启动系统。这个初始化内存盘的目的是使得内核可以被配置得尽可能的模块化，这样我们就可以在系统启动过程中，在内核需要装载某些模块之前，使用位于初始化内存盘映像中的脚本预先装载Linux内核模块。由此可见，所谓初始化内存盘不过是一个专门创建的二进制文件，它包含用于内存盘的一个文件系统映像，同时还包含各种脚本和必须在系统启动之前被装载的模块(通常是针对磁盘等的模块)。现代的启动加载器都支持代表Linux内核将这个文件装载进内存并做好使用准备。

许多发行版由于假设存在该初始化内存盘，所以，如果该内存盘不存在，这些发行版将不能正常启动。在这样的系统中，你将在启动加载器的配置中看到对内存盘的引用——例如在/boot/grub/menu.lst

文件中。幸运的是，制作initrd非常容易，这要感谢现代Linux发行版提供的脚本。在大多数发行版中，一个名称为mkinitrd的脚本文件会为你制作initrd。不管是什么样的情况，你的Linux发行版都很可能会提供一个描述制作initrd所需步骤的文档。

可以使用mkinitrd命令来制作initrd，你需要向它提供initrd的名称和针对的内核版本，或参考该命令的UNIX手册页（man initrd）。请看下面的例子：

```
$ mkinitrd /boot/initrd-2.6.x.y.img 2.6.x.y
```

你显然需要以root用户身份来制作initrd（或使用sudo工具来运行该命令）。这不仅是因为mkinitrd通常需要拥有对/boot目录的写权限，而且因为它必须在一个普通文件之上环回装载一个虚拟文件系统以产生内存盘（取决于具体的进程和文件系统，这有可能不是必须的）。如果你现在还不能完全理解它的内部原理也没有关系——你需要花一些时间来弄明白它，因为内核开发者多年来一直在对Linux内核的编译过程进行研究。

下一步，包装和安装内核。

7.1.7 包装和安装内核

现在，你应该对内核的配置和编译过程有了足够的认识，可以开始自己编译内核了，但你如何才能为一般的Linux发行版包装这些内核呢？答案是，你通常不会这样做——因为没有理由这样做，厂商会负责为他们的发行版提供最新的内核，而其他第三方不太可能随便安装一个你发送给他们的内核，除非有充足的理由这样做^①。不过，有时候，你可能想包装一个内核供自己使用。

大多数发行版都提供一个RPM spec文件（对于Debian来说，是一个可以自己使用的用于建立Debian软件包的规则集）以包含针对这个发行版的配置和编译过程。在使用Fedora Linux和OpenSUSE Linux的情况下，这个过程其实非常简单。你只需安装内核的源代码RPM软件包，对用于编译内核的源文件做出一些修改，然后重新建立内核RPM即可。

安装一个预包装的内核可以减轻开发人员在开发机器和生产机器上的负担，但对经常性的内核测试并不适用——直接拷贝测试内核要比这种方法快得多。

7.2 内核概念

祝贺你！现在，你应该可以充满希望地开始配置和编译自己的测试内核了。你已对内核的配置过程有了一个整体的认识，并且对Linux使用的内核目录布局（至少）有了一个高层次的理解。同时，你也明白了拥有一个独立的测试机器的重要性，你可以将在开发机器上编译的内核放到测试机器上进行测试。现在，是时候开始真正深入了解内核代码以获得乐趣了。

内核是任何操作系统的内核。它扮演着资源仲裁者这一独一无二的角色，其设计目的是避免所有运行在内核之上的应用程序之间的互相干扰。为了减轻编写普通应用程序的程序员的负担——他们必须为Linux系统编写大量的代码，内核还负责提供一个特殊的多进程抽象。这个抽象使得每个应用程序产生一种自己拥有整个Linux系统的错觉。作为一种结果，应用程序通常不需要互相担心彼此的运

^① 这种想法隐藏着一个安全梦魇。决不要从一个你不会授予root访问权限的用户那里安装一个内核——这两者实际上是一回事，尤其当内核是位于一个软件包中，而且该软件包还有它自己的安装前/安装后运行脚本时更是如此，因为这些脚本将以root用户身份来运行，或者你也可以针对这种情况使用一台测试机器。

行状态，除非它们需要进行交互。

应用程序通常不会关心在使用完自己公平分享的处理器时间片后如何放弃处理器，或当自己在等待键盘输入时是否有其他应用程序也会进行同样的等待，或是否会有许多其他事情阻碍程序的运行，而是由Linux内核来负责确保每个应用程序获取一定的CPU时间、对外围资源的合理访问以及它有资格访问的任何其他资源。内核通过对这些功能的实现为困难的计算机科学问题提供了许多创新的解决方案。

在本节中，你将学习一些开发Linux内核必须知道的概念。

7.2.1 一句警告

初次进行内核开发的开发人员常犯的一个错误是认为Linux内核是一个全知全能的神秘程序，它对系统的运作拥有完全的控制权，并以某种监督角色的身份一直在后台运行着。这一想法的一部分内容确实是真的——内核的确在系统中扮演着监督角色，而且它确实拥有对计算机中硬件的最终控制权。但内核并不是一个程序，它是一个例程集，偶尔被调用来为用户的请求和其他任务服务。

Linux内核代码通常只在响应直接来自用户程序的请求时才运行。当然，有一组例程专门用来处理底层的硬件中断——来自计算机硬件设备的异步请求，如帮助Linux跟踪时间流逝的系统定时器，以及一些其他偶尔运行的特殊代码，但并不存在一个一直在运行的神奇的内核程序。你不应该将内核看作一个特别的程序，它更像是一个有特权的库例程集，随时准备接受调用。

了解了这一点之后，你需要学习一些Linux内核所用的基本抽象。在下面几页中，你会发现任务抽象和虚拟内存如何简化从事内核和应用程序开发的程序员的工作。在你开始编写自己的内核代码之前，理解这些概念是非常重要的——内核邮件列表中的文章显示这些基本概念经常被人误解，从而导致产生各种各样的问题。

7.2.2 任务抽象

Linux内核建立在基本的任务抽象之上。它允许每个应用程序独立于其他程序运行——除非这些程序的开发者刻意在程序中引入交互。任务封装了在Linux系统中运行的应用程序的完整状态，包括这个任务所需的所有资源（打开的文件、分配的内存，等等）、处理器的状态和其他细节。任务由`task_struct`结构表示，该结构定义在内核源代码的`include/linux/sched.h`头文件中。

任务结构通常在一个现有程序`fork()`产生一个进程时被分配。不过，它也可以在一个程序创建新线程时被分配。Linux并不像其他操作系统那样有一个专门针对线程的内部抽象，而是以大致相同的方式来对待进程和线程。对Linux内核来说，一个线程就是一个与现有任务共享相同内存和其他系统资源的任务结构，但它有自己的处理器状态。任务还可以针对被称为内核线程的特殊线程来分配——我们将在这章后面介绍更多这方面的内容。

任务一旦被创建，它将一直存在，直到它成功结束或被一个信号杀掉（`kill`）或试图执行一个非法操作（如访问未分配给这个任务的内存位置）为止。Linux负责调度每个任务，它将根据系统所用的调度策略来保证每个任务都能公平地分享处理器时间。每当任务被调度出处理器时，任务结构中的特殊域将被用于保存当时的处理器状态以便当任务再次获取处理器时间时，它可以恢复运行。

用户在谈论Linux系统中运行的程序时，通常不使用术语“任务”，而是使用一个运行程序中的“进程”和“线程”。你将在本章后面看到在内核中这些不同术语之间的内部区别。用户通常使用如`top`和

ps这样的工具来查看系统中运行的进程列表。一个典型的Linux桌面系统中可能同时会有数百个进程在并发执行，每个用户应用程序对应一个进程，GNOME桌面系统对应一些进程，还有一些进程用于其他各种用途。

在内核代码中，`current`宏总是指向处理器当前执行的任务。当你需要引用“现在运行的任务”时，你总是可以使用它。

1. 调度器

Linux调度器是所有Linux系统的核心。这段内核代码负责跟踪给予任务系统CPU时间的频率以及每次分配多少时间——当前任务的时间片或量子。调度是对系统为满足用户交互所产生的各种差别的强烈需求的一种巨大的折中，它为用户应用程序提供最大限度的处理吞吐量，同时又能与其他各种系统活动在各种复杂的记账环境中保持协调一致。

你可以利用内核中的`schedule()`函数来手工调用调度器。当在一个长期运行的代码路径中为系统调用（我们将在下一节和第8章中介绍更多有关系统调用的内容）服务时，我们偶尔会使用它来主动放弃CPU，将CPU让给另一个任务。然后，Linux可能会根据各种因素选择运行另一个任务并将当前任务置于睡眠状态，直到稍后的某个时间为止。这里的关键点是“稍后的某个时间”，因为你不能确定调度器什么时候会再次唤醒一个阻塞的任务——除非你因`schedule()`调用而间接导致阻塞在一个特殊事件上。这是一件相当复杂的事情，你需要花一些时间来理解它。

显然，调用`schedule()`函数很可能会导致当前任务在此刻被阻塞或停止运行，所以只有当确实存在一个当前任务可以阻塞时你才可以明确地调用调度器。更重要的是，许多内核的内部函数可能会因为副作用而导致对调度器的调用，而这可能并不是你所期望的，有时候它还会导致灾难性的后果。当我们在本章后面讨论不同的内核上下文及其对模块的影响时，你还将学习到更多有关为什么这是一个潜在的严重问题的原因。

当编写内核代码时，你通常不需要担心Linux调度器的内部活动——调度器的内部设计是那些开发调度器本身的开发者需要关心的事情，它不在本书介绍的范围内（专门介绍Linux内核设计的图书会讨论这方面的内容）。不过，你确实需要认识到Linux系统必须支持许多并发执行的任务以向Linux系统的用户提供一种多处理的错觉。只有当调度器被调用得足够频繁才能提供这样一种能力以允许其他任务能够公平分享系统处理器的时间（如果系统有一个以上的处理器）。

Linux内核现在越来越具备可抢占性。内核抢占意味着一个更重要的任务在准备好之后可以尽可能快地开始运行，而不用等待另一个任务中的任何已有的活动执行完毕。但内核中的抢占并非总是可能的，例如，如果在内核中执行的代码在其临界区中持有一个自旋锁，它就不能被阻塞，以免日后造成系统死锁。因为长期阻止抢占从而导致整个系统吞吐量的大量延迟是不可接受的，所以应尽可能地将临界区的大小降到最小。

2. 系统调用

内核和任务之间的关系类似于软件函数库和应用程序之间的关系。用户程序通过一个被称为系统调用的已定义的软件接口与内核进行通信。它们通过改变使用的系统调用编号以请求Linux内核代表它们执行特定的操作。当内核完成该操作之后（在内核对请求执行了健全性检查以确保它满足了所有的安全考虑之后），将把执行结果放入一个适当的响应中以返回给应用程序。

系统调用通常不是由应用程序直接调用的。当你编写软件时，使用的是在本书中多次提及的标准GNU C函数库。C函数库为应用程序提供了一系列实用函数，反过来它们又依赖于系统调用来执行它们的部分操作。例如，向一个打开的文件句柄写入文本就依赖于write()系统调用，它被包裹在同名的C库函数中。C库函数负责设置系统调用并获取返回代码。

如系统C函数库这样的函数库提供了许多额外的功能，这些功能并非由内核通过系统调用直接提供，但在函数库中却非常常见。系统调用的代价被认为相对较高，因为它们的操作需要花费较多的时间。可以对应用程序使用time命令，这样就可以看到该程序有多少时间花费在了内核中（通过系统调用），所以我们最好只在必要的时候才进行系统调用。为此，GNU C函数库提供了一组函数（以“f”开头的函数，如fopen/fclose/fread/fwrite），它们对常用的文件操作进行了缓存以避免频繁调用内核来读写每一个字符。类似的优化在函数库中很常见。

你可以通过在命令行中使用strace命令来跟踪一个正在运行的程序中的系统调用。该命令将显示程序中的所有系统调用、它们的返回值和一个特定程序接收到的所有信号。它还将显示由其他系统进程活动或因某些事件的发生而导致Linux内核发送给该程序的任何信号。

我们在一个普通的Linux终端上使用以下命令在strace中运行一个简单的Linux命令ls（为了可读性，我们对输出格式进行了对齐）：

```
$ strace ls
execve("/bin/ls", ["ls"], /* vars */) = 0
brk(0)                                = 0x1002b000
mmap(NULL, 4096, PROT_READ...)        = 0x30000000
open(".", O_RDONLY|O_DIRECTORY...)    = 3
fstat64(3, {st_mode=S_IFDIR|0755, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC)       = 0
getdents64(3, /* 3 entries */, 4096)   = 80
getdents64(3, /* 0 entries */, 4096)   = 0
close(3)                               = 0
write(1, "foo.txt\n", foo.txt)         = 8
close(1)                               = 0
munmap(0x30001000, 4096)              = 0
exit_group(0)                          = ?
Process 3533 detached
```

输出显示了程序如何使用系统调用execve()来启动一个新的进程，然后又如何调用brk()和mmap()来扩展进程的内存地址空间（本章后面将介绍更多有关地址空间的内容）。最后，程序通过调用open()、getdents64()和使用其他文件操作来获取当前目录中的每个文件的信息。

上面strace命令的输出结果经过编辑后删除了一些用来建立共享库的额外系统调用和其他的与本次讨论内容无关的程序操作。

3. 执行上下文

在Linux系统中，任务可以在各种特权上下文中执行。一个普通的用户程序不能直接与系统硬件交互，也不能干扰系统中其他运行程序的操作，也就是说，它们运行在一个非特权用户上下文中，这里所说的用户既可能是系统中的一个普通用户，也可能是系统中的超级用户root。root用户并不直接拥有对机器中硬件的某种特殊权力，它只能通过其具备的可信任能力来要求内核为它执行某些特殊操作（如关机、杀掉任务和其他一些超级用户的操作）。

普通用户程序通过系统调用机制频繁地进入内核以执行各种操作。当进入内核时，系统在被称为内核上下文的特权上下文中执行代码，也就是说，程序进入了内核态。在这个高级状态中，代码可以直接和硬件设备交互，并操纵敏感的内核数据结构以满足内核给出的各种请求。所有的内核代码都在这个特权状态中执行，这通常是通过特殊的处理器特性来实现的。

● 进程上下文与中断上下文

在标准的Linux系统中还有另一组执行上下文，即进程上下文和中断上下文。进程上下文的含义正如它听起来那样，这意味着内核将代表一个用户进程运行一些内核代码（通常是由于一个系统调用），例如，它可能会阻塞当前任务，或将当前任务置于睡眠状态以等待磁盘中出现一些数据。在这种情况下，系统调度器会选择另一个任务在这期间运行。当引发前一个任务阻塞的条件不复存在时，该睡眠任务将被唤醒，并从保存它的状态开始继续运行。

能够阻塞当前任务所带来的实际效果是在大多数情况下能够在内核上下文中执行各种操作。可以自由地调用可能会导致当前任务睡眠的内核函数，调用可能需要系统将一些数据置换到磁盘的内存分配函数中，或调用其他各种操作而不用顾虑其对系统其余部分的影响。大部分内核代码都假定每当你调用某些特定例程时，它们可能会导致当前任务的阻塞。

有时，我们不能阻塞当前任务，因为根本没有一个当前任务可以阻塞。每当一个中断在运行或发生另一个异步事件从而导致内核代码不是代表一个用户任务在运行，而是为了一些更通用的系统目标在运行时，就会产生这种情况。在这种情况下，从技术上来说，阻塞是不可能的（它将导致内核崩溃），即使可以这样做，由于系统通常需要在中断上下文中有非常快的响应，因此它不能等待内存被释放/数据从磁盘中读取，或等待发生在任务阻塞之后的其他操作完成，所以这也是不可行的。

不能阻塞所带来的实际效果是有时候你只能使用有限的内核函数或只能使用这些函数功能的一个有限子集。例如，在一个负责处理某些特殊硬件设备的硬件中断的中断处理程序中，直接调用内核函数kmalloc()来分配内存而不给该函数传递一个GFP_ATOMIC标记是一个非常不好的习惯，因为只有使用了该标记，kmalloc函数才肯定不会阻塞，在这种情况下，如果它不能满足内存分配的需求，它将使用一个特殊的保留内存池，或当它实在没有办法在不阻塞的情况下提供足够的内存时，将返回一个错误代码。

4. 内核线程

并非所有的任务都代表普通用户程序执行。在一个普通系统中存在着一些特殊的特权任务。这些特殊任务被称为内核线程，它们拥有在一个进程环境中运行特权内核代码的独特能力。每当内核程序员需要一个可独立调度的实体（它偶尔能够处理一些数据并在被再次唤醒之前返回睡眠状态）时，他就会使用内核线程。

虽然有许多其他方法来调度Linux内核中延迟的工作，但只有内核线程可以执行一个普通用户进程中的全方位的操作。内核代码只被允许在一个进程上下文中阻塞（通过调用一个可能会导致当前任务睡眠以等待数据到达的函数，如内存分配函数等），而内核线程就提供了这样一个上下文。使用内核线程作为一种手段在内核中提供某些类型的重量级处理是非常普遍的。例如，内核调页守护进程(kswapd)就作为一个内核线程来运行，它偶尔会调整可用内存并将未用的数据交换到系统的缓存空间中。

可以通过在ps命令的输出中查找以方括号括起的进程来找出系统中运行的内核线程，它们通常以字母k开头。例如，在一个典型的Linux系统中，可能会在ps命令的输出中看到如下所示的内核线程（输

出结果经过编辑删除了非内核线程的其他进程)。

```
$ ps fax
 PID TTY      STAT   TIME  COMMAND
  2 ?        SN      0:00  [ksoftirqd/0]
  3 ?        S       0:00  [watchdog/0]
  4 ?        S<     0:02  [events/0]
  5 ?        S<     0:00  [khelper]
  6 ?        S<     0:00  [kthread]
 30 ?        S<     0:00  \_ [kblockd/0]
 34 ?        S<     0:00  \_ [khubd]
 36 ?        S<     0:00  \_ [kseriod]
 86 ?        S       0:00  \_ [pdflush]
 87 ?        S       0:00  \_ [pdflush]
 89 ?        S<     0:00  \_ [aio/0]
 271 ?       S<     0:00  \_ [kpsmoused]
 286 ?       S<     0:00  \_ [kjournald]
1412 ?       S<     0:00  \_ [kmirrord]
1437 ?       S<     0:00  \_ [kjournald]
1446 ?       S<     0:00  \_ [kjournald]
1448 ?       S<     0:00  \_ [kjournald]
2768 ?       S<     0:00  \_ [kauditfd]
 88 ?        S       0:02  [kswapd0]
1106 ?       S       0:00  [khpsbpkt]
1178 ?       S       0:00  [knodemgrd_0]
2175 ?       S<     0:00  [krfcommnd]
```

正如你看到的那样，即使在一个普通的桌面系统中也运行着许多内核线程。随着系统变得更加庞大以及添加了更多的处理器，系统将会产生越来越多的内核线程来帮助它管理各项事务，涉及的范围包括将一个进程从一个处理器迁移到另一个处理器和平衡输入输出等各项工作。

7.2.3 虚拟内存

由Linux内核管理的每个任务都有它们自己的内存环境，也被称为地址空间。每个进程的地址空间通过被称为虚拟内存的基本抽象与其他进程的地址空间分开。虚拟内存使得进程不需要关心系统中其他应用程序的内存使用情况，同时它也在用户程序和Linux内核之间构成一部分屏障，这要归功于系统MMU(内存管理单元)中的各种电路。正如你将在本节中看到的那样，虚拟内存确实是一个非常强大的抽象，但也许在一开始有点难以理解——希望本节对它的介绍对你有所帮助。

虚拟内存允许每个程序独立于其他程序进行编译和连接。每个程序都可以拥有一个范围广泛的内存空间，地址从零开始一直到一个特定系统或单独处理器存在的上限(由管理员定义的每个用户可以使用的资源上限通常被用来设置一个指定进程的实际限制)。在大多数32位Linux系统中，每个进程地址空间的上限是3GB，这允许每个用户进程的大小达到3GB^①。在所需内存受到各种系统资源限制的情况下，每个进程可以对其自己的地址空间进行随意的操作，尽管在实际情况中不同的事实标准确实会影响进程可以进行的操作。例如，对进程中的栈、程序代码和数据等都有事实上的标准化位置。

^① 可以使用3GB的地址空间并不意味着可以在应用程序代码中分配3GB的堆。事实上，普通进程的地址空间大小受到各种因素的限制，这有助于解释为什么即使只需要运行消耗中等程度内存的程序，64位的机器也更受欢迎。

1. 页抽象

现代计算机系统使用被称为页的单位来说明内存的使用情况。一个页只是一组连续的内存地址，它的容量通常都非常小——在Linux系统中常见的是4K字节大小的页，但其确切的大小将根据机器使用的体系结构和平台而有所不同。页构成了虚拟内存概念实现的关键部分。系统中的物理内存被分为数以千计的页（它们根据所选择页的大小在内存中对齐——例如，在4K字节的边界对齐），这些页由一系列内核中的列表来跟踪空闲页列表、已分配页列表，等等。然后，在系统MMU中的虚拟内存硬件负责在页面级别将虚拟内存地址转换为物理地址。

当一个正在运行的进程需要读取存储在指定虚拟地址（如0x10 001 234）中的数据时，该虚拟地址会先由系统中的虚拟内存硬件通过从该虚拟地址中提取出适当的页帧并查找对应的物理页，从而将它转换为一个物理地址——所有这一切对应用程序都是透明的（在许多方面对内核也是如此）。以上的地址为例，如果页面大小为4K，虚拟页帧号（PFN）0x10 001将首先被映射到系统内存中某处的物理页，然后在该物理页的偏移位置0x234对数据进行任何内存读写操作。

系统虚拟内存硬件并不能通过一些神奇的过程就知道虚拟地址和物理地址之间的具体转换关系。内核维护了一系列硬件表，当MMU需要执行转换时，它会在必要的时候直接咨询这些表。实际上Linux本身也实现了各种页跟踪机制和列表以在必要的时候用来更新硬件表。当编写普通内核代码时，你并不需要担心页管理过程的精确机制，但你必须意识到页抽象，特别是用于在Linux内核中表示页的page结构。

2. 内存映射

每当在Linux机器中装载程序时，Linux内核的ELF解析器会从二进制文件中提取出程序代码、数据和其他程序段在进程地址空间中装载的地址。你已在本书的第2章中学习了如何使用GNU工具链通过程序的二进制文件来确定程序的各项细节，包括它应该被装载到的内存地址。作为任何应用程序启动过程的一部分，动态连接器负责确保任何共享函数库也将被动态映射到进程地址空间中所需的地址。

正在运行的程序可以利用mmap()系统调用来改变它们的内存映射。这个特殊的系统调用可以将一个特定的文件映射到指定的内存范围来读、写或执行。mmap()用于映射共享函数库，并作为一种快速机制用于直接处理数据文件，而不是调用C函数库中的文件读写例程来处理文件。mmap()还用于映射匿名内存——没有磁盘上任何真实文件支持的内存，即为各类一般程序使用而调用如malloc()这样的C库例程分配的内存。

可以使用/proc/pid/maps接口来查看Linux系统中执行的任何程序的地址空间。使用一个工具（如cat）来读取/proc目录中某个数字目录下的maps文件的内容可以看到进程号为这个数字的进程的地址空间信息。例如，要查看bash shell的信息，就打开一个终端窗口并执行以下命令（bash将把\$\$替换为它自己的进程ID）：

```
$ cat /proc/$$/maps
00100000-00103000 r-xp 00100000 00:00 0
0f494000-0f49f000 r-xp 00000000 03:0b 983069      /lib/libnss_files-2.4.so
0f49f000-0f4ae000 ---p 0000b000 03:0b 983069      /lib/libnss_files-2.4.so
0f4ae000-0f4af000 r--p 0000a000 03:0b 983069      /lib/libnss_files-2.4.so
0f4af000-0f4b0000 rw-p 0000b000 03:0b 983069      /lib/libnss_files-2.4.so
0f4c0000-0f4c3000 r-xp 00000000 03:0b 987547      /lib/libdl-2.4.so
0f4c3000-0f4d2000 ---p 00003000 03:0b 987547      /lib/libdl-2.4.so
0f4d2000-0f4d3000 r--p 00002000 03:0b 987547      /lib/libdl-2.4.so
0f4d3000-0f4d4000 rw-p 00003000 03:0b 987547      /lib/libdl-2.4.so
0f5c0000-0f71b000 r-xp 00000000 03:0b 987545      /lib/libc-2.4.so
```

```

0f71b000-0f72a000 ---p 0015b000 03:0b 987545 /lib/libc-2.4.so
0f72a000-0f72e000 r--p 0015a000 03:0b 987545 /lib/libc-2.4.so
0f72e000-0f72f000 rw-p 0015e000 03:0b 987545 /lib/libc-2.4.so
0f72f000-0f732000 rw-p 0f72f000 00:00 0
0fb0000-0fb0c5000 r-xp 00000000 03:0b 987568 /lib/libtermcap.so.2.0.8
0fb0c5000-0fb0d4000 ---p 00005000 03:0b 987568 /lib/libtermcap.so.2.0.8
0fb0d4000-0fb0d5000 rw-p 00004000 03:0b 987568 /lib/libtermcap.so.2.0.8
0ff0000-0ffd0000 r-xp 00000000 03:0b 987544 /lib/ld-2.4.so
0ffed000-0ffee000 r--p 0001d000 03:0b 987544 /lib/ld-2.4.so
0ffee000-0ffe0000 rw-p 0001e000 03:0b 987544 /lib/ld-2.4.so
10000000-100c6000 r-xp 00000000 03:0b 950282 /bin/bash
100d5000-100dc000 rw-p 000c5000 03:0b 950282 /bin/bash
100dc000-100e0000 rwxp 100dc000 00:00 0
100eb000-100f3000 rw-p 000cb000 03:0b 950282 /bin/bash
100f3000-10135000 rwxp 100f3000 00:00 0 [heap]
30000000-30001000 rw-p 30000000 00:00 0
30001000-30008000 r--s 00000000 03:0b 262661 /usr/lib/gconv/gconv-modules.cache
30008000-3000a000 rw-p 30008000 00:00 0
3001b000-3001c000 rw-p 3001b000 00:00 0
3001c000-3021c000 r--p 00000000 03:0b 199951 /usr/lib/locale/locale-archive
7fa8b000-7faa0000 rw-p 7fa8b000 00:00 0 [stack]

```

正如你看到的，即使对于一个简单的应用程序而言，它的地址空间看起来也很复杂。在输出中，每行在左边显示了一个内存地址范围，其后是与该地址范围关联的权限，最后是支持该内存范围的磁盘文件。在输出中，你可以看到每个共享函数库被映射到bash进程的低端内存地址，然后是/bin/bash程序代码，最后是位于地址空间高端的栈（因为栈是向下生长的，所以栈将向它下方的大片未映射的区域扩展）。

● 内核内存映射

程序并不能将它们的地址空间中的所有可用内存位置都供自己使用。32位Linux系统通常对用户的地址空间有3GB上限的限制（64位系统将根据特定平台的约定而有所不同），这使得每个进程都有1GB的内存不能被进程使用。但这些内存并没有被浪费，而是被Linux内核使用。内核这么做是为了能够提供比每次产生系统调用时都拥有它自己的内存地址空间有更好的性能表现。

当在现代Linux系统中发生一个系统调用时，处理器将切换到更高的特权级以执行内核代码，这些内核代码位于用户进程的高端地址空间中。内核代码使用它自己的内核栈并独立于它提供服务的程序而运行，不过它仍然保留对整个进程地址空间的完全访问权并可以自由操纵其中的数据——这方便了通过系统调用在内核和进程之间传送缓冲数据，以及在必要的时候对程序中的其他数据进行访问。

特殊的硬件保护机制将阻止用户应用程序访问内核数据，尽管在老版本的Linux内核中偶尔存在的漏洞会允许一个程序欺骗内核以获得对特权内核内存的访问。尽管如此，这种对进程地址空间的优化在Linux和其他操作系统中是非常常见的，所以它仍然维持现状。有一些针对内核的特殊补丁可以为内核提供它自己的地址空间，而且有一些发行版内核也使用了它，但它大约会给系统增加10%的性能开销。

3. 任务地址空间

在内核中，内存以各种形式存在并通过各种详尽的列表来说明，它所涉及的其他各种机制由于过于复杂而不适合在本章的范围中进行讨论。每个任务通过其task_struct结构中的mm成员来说明它自己的内存使用情况。mm_struct结构又包含一组指向其他结构的指针，这些结构用于说明任务中映射的每个地址范围——每个结构对应你在前面/proc/pid/maps输出中看到的一个地址范围。任何一个任务的完整地址空间都可以变得非常复杂，这还是在没有考虑使用更加高级用法的情况下。

每当一个任务结构被Linux内核创建，它的`mm`成员就被设置为指向一个新创建的`mm_struct`结构，该结构存储了有关任务地址空间的信息。每当内核需要告诉MMU一个指定任务的正确内存范围时，它就会查询这个结构。内核还将通过查询`mm_struct`结构以确定是否应告诉MMU允许对某个指定的内存位置进行读、写或执行操作。该结构还用于确定一个特定的内存范围是否需要在使用之前从磁盘交换进内存。

查询任务的`mm_struct`结构的一种常见情况是在发生缺页时。你已知道Linux系统使用页来表示虚拟内存的基本单位，所以页也是内存访问的基本单位。任何试图对一个进程中的目前还不可用的地址中的数据进行读、写或执行操作（它已被交换到磁盘上以释放内存空间；或属于磁盘上某个文件的一部分，但只有在实际需要的时候才会被装载进内存，等等）都将导致缺页。系统的缺页处理程序将使用一个任务中的`mm`数据结构来确定内存访问是否合法，如果合法就产生特定的内存页。

并不是所有的任务都有分配给它们自己使用的`mm_struct`结构。就像一个进程中的线程有它们自己的`task_struct`结构，但它们却共享同一个内存地址空间（当创建一个新的任务结构时，给`clone()`系统调用传递一个特殊的参数以保存这个指针）一样，内核线程甚至在开始还没有它们自己的`mm_struct`结构。可以通过查看内核源代码以找到更多有关`mm_struct`结构的信息，它也定义在`include/linux/sched.h`头文件中。

7.2.4 不要恐慌

这一节在很短的篇幅内^①简要介绍了很多概念并涵盖了许多内容。请记住，这些内容都比较高深，没有人能立刻全部理解它们。这个简要概述的目的不是通过指出你在进行内核开发时必须要了解非常多的知识从而让你不敢追求你对内核开发的兴趣，而是帮助你在开始内核开发之前先从30 000英尺的高度俯瞰内核的一些关键思想。后面几章将向你介绍一些更有意义的例子，但即便如此，一本书也只能在其有限的篇幅中介绍这么多内容了。

通过对一些实际问题的研究，你将学到比任何书本可以教给你的还要多得多的知识。当然，你肯定会犯一些错误（这是肯定的），但你也将从中学到更多。请把这些记在心里，本章的其余部分将介绍内核开发过程并向读者提供一些在线资源和你可以寻求帮助的讨论组的链接。

7.3 内核编程

Linux内核由许多单独的组件构成，它们必须相互协作以使系统作为一个整体可以正常地工作。Linux内核有别于许多其他操作系统内核，它既不是一个微内核（由许多彼此之间完全相互隔离的模块化组件构成，有一个很小的可信任核心），也不是一个纯的单内核（内置一切功能，在运行时缺乏灵活性）。事实上，Linux吸取了过去30年开发UNIX内核的经验。从这个意义上来说，它是一个巨大的单内核，但同时它也为运行时扩展添加了强大的模块系统。

在内核中，任何代码都可以与系统的任何其他部分交互。它们不能得到你的任何保护，正如俗话说的那样，“Linux允许你立刻搬起石头砸自己的脚”——但这却提供了极大的灵活性，几乎使得你可以毫无阻碍地做任何事情。内核几乎完全由C语言编写，其中只使用了少量的汇编语言代码来支持特定的底层操作，因为它们必须使用与体系结构（或平台）相关的汇编代码。这样做的目的就是为了使

^① 双关语。原文是“in a small amount of page”，这里翻译为“在很短的篇幅内”，但它也可以解释为“在少量的内存页中”，所以作者在这里注明它是双关语。——译者注

编写的内核尽可能地具备可移植性。

Linux并没有计划使用C++编写内核，在所有体系结构的内核中执行浮点运算也并非易事——最好不要这样做。事实上，有很多事情绝对不能在内核中做。例如，不应该尝试在内核中打开并操作一个文件——是的，你可以这么做，但在用户空间中将更容易做到这一点，它通过一个适当的接口就可以将数据提供给内核。当你开始Linux内核的开发时，你将很快学到一些显而易见的可以做和不可以做的事情。

可装载模块

开始Linux内核开发最简单的方法之一就是开发可装载Linux内核模块（LKM），这些模块通过在运行时提供新的功能和特征以扩展核心内核的功能。但是，我们并不能在运行时扩展内核每个方面的功能——你不能真正地很轻易地接触到核心内核功能，如调度器、虚拟内存子系统等，不过还有很多事情可以通过模块来做到。模块的另外一个优势是它是一个自包含的实体，它可以相对比较容易地从一个内核移植到另一个内核。

在本章中，你将看到一个简单的模块示例以了解编译和开发一个Linux内核模块的过程。在接下来的几章中，你将学习如何从零开始开发一个强大的内核模块——为一个已有的Linux系统添加对新硬件的支持和其他有用的功能。但你不能在还没有学会爬之前就开始想跑，所以你首先需要了解从源代码开始编译一个测试模块的过程。

示例模块

下面这个非常简单的示例在装载和卸载该模块时会向内核的环缓冲区（内核日志）写一条欢迎信息。它并没有做任何特别有用的事情，这并不是它存在的目的，但它确实演示了如何建立一个内核模块并在运行的系统中装载它。

plp示例模块由单个文件plp.c构成：

```
/*
 * plp.c - An example kernel module
 *
 * Copyright (C) 2006 Jon Masters <jcm@jonmasters.org>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation.
 *
 */

#include <linux/module.h>
#include <linux/init.h>

int __init hello_init(void)
{
    printk("plp: hello reader!\n");
    return 0;
}

void __exit hello_exit(void)
{
    printk("plp: thanks for all the fish!\n");
}
```

```

}

/* Module Metadata */

MODULE_AUTHOR("Jon Masters <jcm@jonmasters.org>");

MODULE_DESCRIPTION("PLP example");
MODULE_LICENSE("GPL");

module_init(hello_init);
module_exit(hello_exit);

```

请注意这个示例模块代码的某些地方。首先，它没有main()函数，因为它并不是一个普通程序，你不能使用对gcc前端的常规调用来编译它并期望能够执行最终的二进制文件。相反，模块定义了内核可以在特定时间使用的一些入口。所有的模块都有初始化和退出函数，模块通过调用宏module_init()和module_exit()来将它们注册到内核——这两个宏通过修改ELF格式的模块以在一个特殊段中提供函数的信息，当内核装载模块时将查询该段。

你会看到这个模块调用了三个宏来设置作者和许可证信息，在模块中这么做是非常重要的。有许多理由可以解释为什么理解Linux内核及其模块发布所遵循的许可证是一个好主意，但除此之外，如果你没有定义MODULE_LICENSE，模块将“污染”内核并导致该模块被设置一个特殊标记，这样开发者将认为可能无法获得模块源代码以帮助调试。你当然不希望这样，所以应该设置一个适当的许可证——例如，GNU GPL。

编译示例模块非常容易，这要感谢Linux 2.6内核的编译系统Kbuild。示例源代码包含一个简单的Kbuild文件，它只包含下面一行内容：

```
obj-m      := plp.o
```

这个文件在要求内核编译系统处理当前目录时被读取并允许内核找到模块源文件plp.c。为了真正地编译模块，你需要有一个刚编译的内核树——仅有内核源代码是没有用的，你需要有一个编译好的内核树以便进行相应模块的编译（厂商可能是一个例外——他们提供一个特别修改过的内核头文件集以进行模块的编译，但这并不通用）。有了这样一个刚编译的内核树后，你可以在模块源代码目录下使用以下命令进行模块的编译：

```
make -C /usr/src/linux-2.6.x.y modules M=$PWD
```

其中的/usr/src/linux-2.6.x.y是你要编译的模块所对应的内核源代码的目录。

如果正确地输入了命令，并且在系统中也正确地设置了模块源代码，你将看到如下所示的由Kbuild系统输出的编译信息：

```
$ make -C /usr/src/linux-2.6.x.y modules M=$PWD
make: Entering directory `/usr/src/linux-2.6.x.y'
CC [M]  /home/jcm/PLP/src/kerneldev/plp.o
Building modules, stage 2.
MODPOST
CC      /home/jcm/PLP/src/kerneldev/plp.mod.o
LD [M]  /home/jcm/PLP/src/kerneldev/plp.ko
make: Leaving directory `/usr/src/linux-2.6.x.y'
```

这个编译过程将最终输出一个可装载的内核模块plp.ko。你可以通过使用标准的Linux modinfo系统工具来找出这个模块的更多信息以及用于编译该模块的源代码的校验和：

```
$ /sbin/modinfo plp.ko
filename:      plp.ko
author:        Jon Masters <jcm@jonmasters.org>
description:   PLP example
license:       GPL
vermagic:     2.6.15 preempt gcc-4.1
depends:
srcversion:   C5CC60784CB4849F647731D
```

你可以通过使用`/sbin/insmod`和`/sbin/rmmod`命令并向它们传递内核模块的完整路径以装载和卸载这个模块。要注意的是，除非你在系统主控台上装载和卸载这个模块，否则你将看不到模块的输出信息——你可以使用`dmesg`命令或查询你的系统日志。如果你还是无法看到内核的输出信息，请询问你的Linux厂商以了解它们的日志处理方式。

下面是一个在系统主控台上装载和卸载示例模块的例子：

```
$ /sbin/insmod plp.ko
plp: hello reader!
$ /sbin/rmmod plp
plp: thanks for all the fish!
```

取决于你所使用的系统，你也许还可以将可装载内核模块`plp.ko`安装到标准的系统模块目录`/lib/modules`中。要做到这一点，使用`make`的`modules_install`目标：

```
make -C /usr/src/linux-2.6.x.y modules M=$PWD modules_install
```

然后你就可以使用`/sbin/modprobe`工具以常规方式来装载和卸载模块了：

```
$ /sbin/modprobe plp
$ /sbin/modprobe -r plp
```

`modprobe`负责找出模块所在位置并装载和卸载它。它还将满足示例模块可能需要的任何依赖关系——装载该模块依赖的其他模块。你并不需要担心多重依赖内核模块，除非你正在开发一些非常大型的内核项目——在这种情况下，你也许早已知道这里介绍的所有内容了。

7.4 内核开发过程

上游Linux内核社区的开发过程与其他自由软件和开放源码开发项目的开发过程有所不同。首先，与许多其他项目相比，Linux是一个非常大型的项目，它有广泛的贡献者和更加广泛的关注度。Linux开发是一个国际化的事务，它有遍布世界各地和位于各个时区的众多开发者。英语作为它的半官方语言用在所有的邮件列表、源代码注释和文档中，但你也需要允许母语是非英语的用户参与其中。

内核开发以疯狂的补丁活动、稳定性测试、版本发行、“稳定”内核小组推出的任何必要的发行后更新这样一个周期在循环。内核开发由三个关键的基础部件驱动——git内核SCM工具、LKML邮件列表和一个名字叫Andrew Morton的家伙。每个Linux内核开发者都可以通过kernel.org网站跟踪Linus个人的上游git树，他们可以通过git树跟踪那些可能会构成Linux内核未来版本一部分的补丁以及测试内核当前的开发状态。Andrew Morton维护他的-mm内核树作为递交给Linus的补丁的集结区。

7.4.1 git：傻瓜内容跟踪器

Linux内核开发社区使用一个被称为git的自动化SCM（软件配置管理）工具。我们已在本书的第4

章对git进行了介绍，但本节将提供一些特定于内核的注意事项。git已迅速被采纳成为内核社区的事实上的开放源码SCM工具——当人们不得不放弃前一个内核开发所使用的SCM工具时，Linus经过几周编写了这个工具，它可能是如今被广泛使用的最精巧和最有用的SCM工具之一。从根本上来说，git实际上是一个非常快速的“文件系统”和数据库，这两者被合并到一个普通的应用程序中。它可以快速地跟踪许多源代码目录。

你可以通过你的Linux发行版获得一份git的拷贝。如果它没有被安装，请查找git的预包装版本以避免你从源代码开始自己编译它的麻烦。使用git需要完成的第一项任务是克隆一个上游Linux内核树作为参考。大多数内核开发社区都使用Linus Torvalds的git树作为参考。你可以在一个终端上使用以下命令来克隆Linus的git树：

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
linus_26
```

这将创建一个Linus的git源代码树的新拷贝并将它命名为linus_26。一旦克隆完成，你将在linus_26目录中看到类似于常规内核的源代码树。你可以在源代码树的目录中使用git pull命令来更新该树。例如：

```
$ git pull
Unpacking 3 objects
100% (3/3) done
* refs/heads/origin: fast forward to branch 'master' of
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6
  from ce221982e0bef039d7047b0f667bb414efece5af to
427abfa28afedffadfc9dd8b067eb6d36bac53f
Auto-following refs/tags/v2.6.17
Unpacking 1 objects
100% (1/1) done
* refs/tags/v2.6.17: storing tag 'v2.6.17' of
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6
Updating from ce221982e0bef039d7047b0f667bb414efece5af to
427abfa28afedffadfc9dd8b067eb6d36bac53f
Fast forward
 Makefile |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

你可以在本书第4章中阅读更多有关如何使用git的信息，或者你也可以查询Linux内核的网站，它包含使用git进行内核开发的一个快速入门指南。你可以在git的网站www.kernel.org/git上找到关于git的更详细的内容。

gitk：图形化git树浏览器

git是一个非常强大的SCM工具，遍布世界各地的Linux内核开发者每天都在使用它。尽管git包含一整套命令用于查询当前git树以获得有用的信息，但是有时候，如果有一个图形化的工具可以帮助你以可视化的方式查看git树将会更方便。因为这个原因，人们编写了gitk。gitk表示使用由tcl语言编写的图形化工具来管理某个特定树，它可以在大多数Linux系统中运行。gitk工具允许你轻松地浏览一个git版本库（如在本例中的内核版本库）的改动。每个修改集、合并和删除都可以被轻易地识别，它们还包括与之关联的元数据——如作者姓名和电子邮件地址。请安装针对你的特定系统的gitk软件包。

你可以进入一个由git管理的内核源代码目录并运行命令gitk来尝试使用它。该命令将产生一个如

图7-1所示的图形化界面。

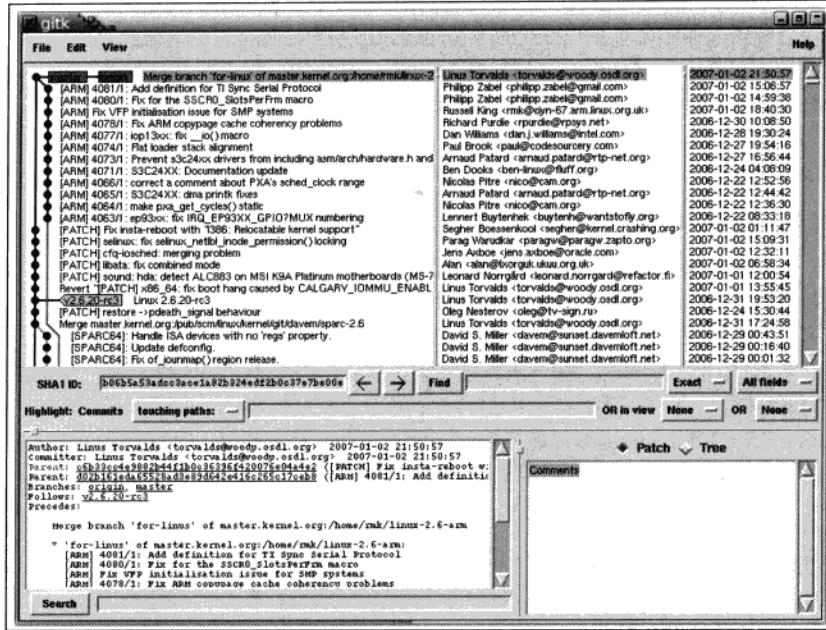


图7-1 运行gitk命令来尝试使用由git管理的内核源代码目录时产生的图形化界面

7.4.2 Linux 内核邮件列表

Linux内核开发围绕着一个内核黑客的核心小组（数十人以内），他们出没在LKML（Linux内核邮件列表，kernel.org的另一个产物）上。LKML是一个公开的邮件列表，人们在上面公开讨论可能会进入内核的新特征、发布补丁、公布新版本、时不时还会爆发很多激烈的口水战。LKML上永远不会有平淡的日子，虽然有些人希望有这一天。

你可能想尽可能快地订阅LKML以开始跟踪这一邮件列表上的信件——更多与订阅有关的信息请查看kernel.org网站。注意，你不一定非要直接订阅邮件列表才能跟踪其中的信件。有一些很好的信件资料存档网站可以使用，其中一个网站是lkml.org，它还提供了各种统计数据。对于那些喜欢使用Usenet feed的用户来说，gmane.org为LKML读者提供了一个免费的新闻网关。此外，你还可以在某些Linux出版物中找到许多邮件列表活动的摘要。

除非你已经浏览了信件资料存档并对邮件列表有初步的认识，否则请不要发送信件给LKML。许多问题反复在LKML中出现，并迅速地激怒了LKML的许多成员^①。

1. 提交Linux内核补丁

Linux内核开发在很大程度上依赖于每天发送到Linux内核邮件列表中的许多补丁。虽然也存在一

^① 本书作者好不容易才意识到这一点，所以最好不要让历史在你身上重演。

些其他方式可以与其他开发者分享代码，但简陋的补丁仍然是开发者首选的机制，因为它不仅提供了一种简单的方式来共享代码，而且同时还在邮件档案中保留了谁做了什么的审计线索——这是一件好事情，特别是当有些人偶尔会对某些特性的最初起源做出不同的声明时更是如此。

如何制作补丁已在本书的前面介绍过。对于Linux内核来说，你需要提交的补丁应该位于一个全新的内核源代码和包含你目前正在开发的特性的内核源代码之间。例如，假定你有一个新的froozebar魔幻显卡，你决定在你的开发内核树中为它编写一个驱动程序。一旦这个驱动程序准备好给公众使用，你就需要生成一个使用统一diff格式的补丁，该补丁位于这个开发内核和一个全新内核源代码之间：

```
diff -urN linux-2.6.x.y.orig linux-2.6.x.y_froozebar >froozebar.patch
```

这个命令的输出结果是一个位于原有内核源代码和新修改的内核源代码之间的一个补丁。内核补丁总是在内核源代码顶层目录的上一级目录中制作的，这是为了避免今后将补丁应用到其他内核树时产生任何歧义。补丁一般应首先通过LKML提交给广泛的Linux内核社区供其审议，除非有一个非常明显的维护者（查阅MAINTAINERS文件）。应首先咨询，然后由类似于“快闪暴走族”^①的随机组成的特定小组进行讨论。

当你发送一个补丁到Linux内核邮件列表时，你必须在邮件中遵循一个标准的礼节。首先在邮件主题中以单词[PATCH]开头，然后在邮件主体中对特定问题进行一个简短的介绍，在“Signed off by”行之后是补丁内容。下面是一个有理想格式的示例，当你提交自己的补丁时，应遵循这个示例的格式。

```
Subject: [PATCH] New froozebar driver for froozybaz-branded graphics cards
```

```
This driver adds new support for the exciting features in the froozebar chipset as
used in froozybaz-branded graphics cards. It's not perfect yet since at the moment
the image is displayed upside down, in reverse video, and the kernel crashes a few
minutes after loading this driver. But it's a start.
```

```
Signed off by: A developer <a.developer@froozybazfansunite.org>
```

```
diff -urN linux-2.6.x.y.orig/drivers/video/froozebar.c linux-
2.6.x.y_new/drivers/video/froozebar.c
--- linux-2.6.x.y.orig/drivers/video/froozebar.c      1970-01-01
01:00:00.000000000 +0100
+++ linux-2.6.x.y_new/drivers/video/froozebar.c      2006-06-19 09:41:10.000000000
+0100
@@ -0,0 +1,4 @@
+/*
+ * froozebar.c - New support for froozybaz branded graphics cards
+ *
+ */
...
```

2. 内核开发新手的邮件列表

如果你是一个Linux内核开发的新手（或即便你不是新手），你可能会有兴趣参与kernelnewbies（内

^①“快闪暴走族”原文为flash mob，它指的是一群会用网络、手机等互相沟通、串联并参与特定族群活动和做出实际行动的人。——译者注

核新手) 社区的讨论。kernelnewbies.org网站是一个非常受欢迎的基于wiki的网站，任何人都可以通过编辑和使用它来改善自己理解Linux内核和内核中一般文档的水平。除了这个网站以外，该社区还有一个邮件列表，参与者既有新手，也有非常有经验的内核专家。

如果你有一个不能非常确信是否适合在LKML中发表的问题，同时你又觉得不适合询问一般的Linux用户，如你的本地Linux用户组，那么你可以尝试发送信件给kernelnewbies邮件列表。你可能会为有那么多其他用户也有类似的问题而感到惊喜。只是不要向他们询问二进制驱动程序或你最喜欢的“星际迷航”的情节。

● 内核看门人

就像kernelnewbies组一样，kernel janitor(内核看门人)用于帮助那些想要对内核做出贡献的人们，但janitor中并不总是Linux内核社区的新手。内核看门人邮件列表是一个针对各种烦人(和一些不怎么烦人的)的问题的通用交换所，这些问题需要在Linux内核中解决，但它们可能不会得到常规内核开发社区的重视。你可以通过访问janitor.kernelnewbies.org网站以帮助解决问题并在这一过程中进行学习。你还可以在该网站上找到它的邮件列表。

7.4.3 “mm”开发树

Andrew Morton构成Linux内核开发过程的一个基石，就像Linus和其他少数保持内核不断发展的开发者一样。但Andrew值得特别提及，因为他维护着“-mm”内核树并且他是Linux 2.6内核的官方维护者。如果你想让你的内核补丁进入上游Linux内核，它很可能会在某一环节上经过Andrew(又名akpm)之手。与使用开发内核树相比，Andrew的“-mm”内核树被证明对于那些想要对还未进入内核的实验性补丁进行测试的人们来说是一个好地方。

你可以通过Linux内核网站kernel.org来跟踪Andrew的“-mm”内核。Andrew并不经常使用SCM系统如git，而是使用普通补丁和tarball文件，但他的确有他自己的用于快速应用和测试补丁的系统quilt。

你可以在GNU的“Savannah”开放源码跟踪网站<http://savannah.nongnu.org/projects/quilt>上找到更多有关Andrew的quilt补丁跟踪工具的信息。

7.4.4 稳定内核小组

“-stable”小组负责当内核中偶尔出现一些令人厌恶的安全漏洞以及当必须进行常规内核版本的升级以解决一些非常严重的错误时，对现有内核版本进行更新。该小组因增强新内核版本发行后的稳定性而成立并负责维护2.6.x.y内核版本中的数字“y”。与花费六个星期或更长的时间等待新版本相比，“-stable”小组通过部署简单的补丁来改善内核的性能。最近该小组遇到了很多困难，尤其是各种运行在内核之上的自动化代码覆盖工具在内核中发现了一些相当低级的错误。

稳定内核小组一直与安全小组密切合作，并且总是在新版本中集成任何最新的安全补丁。如果你想帮助稳定和安全内核小组，可以密切注意内核中的任何令人厌恶的错误和安全漏洞并通过他们的电子邮件security@kernel.org来通知他们。千万不要在还没有通知他们的情况下就公开安全漏洞——你应该给他们一些时间来准备做出适当的响应(并希望他们能给出一些补丁)。

7.4.5 LWN: Linux 每周新闻

没有哪一个介绍Linux内核开发的章节不会提及Linux每周新闻。多年来，Jonathan Corbet、Rebecca

Sobol和其他Linux每周新闻的工作人员坚持不懈的报导发生在Linux社区的最新新闻。LWN运行一个每周新闻公告，它包括一周的内核开发摘要并对内核的未来走向提供各种独特的见解。此外，LWN还提供Linux内核在版本更新之后内部API改动的完整列表。

将LWN的网址<http://www.lwn.net>添加到你的书签中——这样做肯定是值得的。

7.5 本章总结

本章在相对较短的篇幅中涵盖了很多主题。这样做的目的是希望能够尽可能全面地对内核开发过程进行概述。虽然你不需要立刻理解Linux内核或其开发过程的每一个方面，但对任务、虚拟内存和内核如何融入一个系统等概念有总的了解还是非常重要的。如果你对内核各个方面是如何结合在一起以构成一个可以正常工作的系统有一个最基本的理解，你将发现它使得你编写内核代码变得容易了许多。

如果你不具备自己配置和编译内核的能力，那本章中提供的信息将无用武之地。所以，本章对在实际机器上编译、安装和测试内核背后的过程做了基本的介绍。此外，为了让你对内核有一个全面的了解，本章还对内核的交叉编译、配置管理和其他主题做了补充说明。希望本章除能为后续章节奠定基础，在后面几章中，你将应用本章介绍的一些概念到现实的例子中。

第8章将介绍内核接口，介绍内核的最后一章——第9章将详细介绍如何建立你自己的内核模块。

内核接口

在上一章中，你学习了如何开发软件来扩展或修改Linux内核，并且知道了Linux内核从根本上来说仅仅是一个有用的（有特权的）库例程集，可以在程序中利用它们来代表你执行一些操作。你还学习了如何编写自己的Linux内核模块。上一章中的简短示例演示了在用户程序（用户层）和内核之间的一些限定的接口方式。

本章将在上一章的基础上解释在Linux内核中存在什么类型的接口以及在Linux内核和其他用户级应用程序之间的接口。术语“接口”的含义并不是非常明确，但在本章中，它既指内核和用户之间的各种接口，也指内核自身的内部API。这两者都将在本章中进行讨论，本章还会对一些关键概念进行解释，例如缺乏一个稳定的内核ABI将影响Linux内核模块的编写。

在阅读完本章之后，你将对内核如何融入一个典型的Linux系统有更好的理解。即使你不准备编写自己的内核模块，本章中提供的信息也将有助于你理解一些工具的作用，如udev动态设备文件系统守护进程和消息是如何在系统的底层传递的。这些内容对那些需要建立他们自己的Linux系统的用户以及那些偶尔需要在定制系统中改动正式接口的用户将特别有用。

8.1 什么是接口

术语“接口”可以有很多不同的含义。在本章中，它指用来定义Linux系统不同部分相互交互的软件接口（API）。应用程序可以合法地与Linux内核交互的方式只有那么几种——例如，使用系统调用、读写文件或读写套接字。已定义接口的存在减轻了每个开发人员的负担，在这方面Linux也是一样（使生活更加轻松）。

在Linux内核中有大量不同类型的接口。本章将集中讨论其中的两种接口：在Linux内核和用户应用程序之间的外部接口和Linux内核自身各个部分之间的内部接口。后者还涉及对缺乏一个标准内核ABI以及它将如何影响编写Linux内核模块的解释，对于那些必须将他们的内核模块移植到大量其他目标Linux系统的开发人员来说尤其如此。

这两种不同类型的接口（外部和内部）可以归纳如下：

- 在Linux内核和用户空间（用户应用程序）之间的外部接口定义了某些操作必须通过的软件壁垒。接口（如系统调用）为发送给内核的请求和返回的响应提供一个已定义的机制。netlink套接字提供一个进入内核的接口，通过它可以接收消息、更新路由表和执行其他各种可能的操作。这些接口不能被轻易改变，因为大多数用户应用程序依赖于它们保持一致^①。

^① 一件在线销售的伟大Linux内核黑客T恤通过下面这句话传递了这一讯息：“The Linux kernel: breaking user space since 1991.”（Linux内核：自1991年开始就一直在破坏用户空间）。这里的关键是破坏用户空间总是不可接受的，它只能在对一个预期的改动做了预告之后才能发生。

- 内部接口由提供给可装载内核模块的输出内核函数集构成（即所谓的“公开输出”的内核符号，如`printf()`函数）。例如，`printf()`函数为任何一个有消息记录需求的可装载内核模块提供消息记录的能力。一旦内核被编译，它就将包含这些函数的二进制版本，所有后续的内核模块都必须建立在它的基础之上。

你之前可能已遇到过API，它们作为编程工具（如一个图形化视窗系统）或标准系统函数库的一部分。与它们不同的是Linux内核的可外部访问的接口（如系统调用）无法被轻易的改动而不对整个系统造成长久的影响。当一个普通软件库被更新时，考虑到兼容性，该软件库的老版本仍然可以保留在系统中。但这对Linux内核来说是不可能的，所以对外部可见的改动必须谨慎控制。由此可见，最终要想删除所有过时的接口往往需要经过很长一段时间才能完成。

未定义接口

有时候，在现实世界中，事情并不是它们看起来那么简单，也有破坏规则的情况发生。例如，在某些平台上，我们可以在一个用户应用程序中手工访问物理系统内存^①。这个过程绕过了正常的内核对哪个设备位于该特定内存地址的处理，所以我们必须非常小心地使用它。例如X视窗系统在历史上就一直渴望能直接访问图形硬件。

还有一些其他方法也可以绕过系统的标准接口。当使用Linux时，你并没有被限制只能遵循规则——尤其当你使用的是一个完全在你控制之下的定制设计的系统时更是如此。大量的嵌入式设备都带有非标准的修改以使开发人员的工作变得更轻松。可以这么做并不意味着我们鼓励你进行定制的修改，但你应该知道都有哪些可能性。

你绝不应该欣然接受绕过正规系统接口的机会。正规系统接口的目的是使得你的工作更加轻松并提供互操作性——就像在高速公路上张贴的时速限制。

8.2 外部内核接口

外部内核接口是那些普通用户应用程序用来与Linux内核通信的接口。它们包括系统调用、UNIX文件抽象和过去几年中专门针对Linux实现的其他各种现代内核事件机制。你不仅完全依赖于内核来使得你的应用程序可以正常工作，而且完全依赖于它来提供一个稳定的环境以使得你定制的驱动程序可以通过它与应用软件通信。

外部接口如系统调用定义了非常死板的API，它们不能被轻易修改。事实上，Linux内核中任何被认为是面向用户的部分都不能在未经公告和小心处理的情况下被修改。例如，内核社区在几年前投票决定删除一个旧的设备文件系统（它负责在`/dev`目录下创建文件，就像现在的`udev`），但直到现在，社区才认为可以安全地从内核中删除这个过时的文件系统而不会对那些仍然依赖于它的用户造成不利的影响^②。

-
- ① 对某些受限硬件系统来说，这可能是其设计“特征”，其他系统也可以通过特殊的受控权限来实现它。但不管属于哪一种情况，有时候为了获得相对的性能提升确实有必要这样做——不通过常规的内核接口可以节省一些开销。
 - ② 即便如此，这也是因为新的接口是一个可兼容的替换并已使用了足够长的一段时间才能这么做。有时，厂商会在接口已从“上游”官方Linux内核中删除之后再将它们添加回内核以避免强迫他们的客户升级原有的软件。我们再次声明，这种类型的基本系统破坏是被非常小心地控制的。

外部接口的数量也许比你认为的还要多。现代Linux系统不仅有设备文件系统，还有已过时的procfs（/proc）、新的sysfs（/sys）和更多其他的文件系统。上述每一种接口都可能被用户软件所依赖——例如，module-init-tools（它提供模块加载和模块管理系统工具modprobe、insmod、rmmod和modinfo，等等）就依赖于/proc目录中的特定文件并将随着时间的流逝更多地依赖于/sys目录中的各种文件。只是在近期（引入sysfs之后），用户、开发人员和厂商才真正体会到添加更多此类的机制来与Linux内核交互所引入的复杂性。

这里的底线是：当添加新的内核接口时，要考虑到它可以被长期使用。如果你提议一个看上去没有经过深思熟虑的新接口，你将会从Linux内核社区得到非常消极的反应——他们不希望这一接口从现在开始只能使用5年的时间。

8.2.1 系统调用

也许最常用的内核接口就是系统调用了。在系统正常运行甚至在明显空闲的情况下，一个Linux工作站平均每秒钟也将产生数以千计的系统调用。所有的系统守护进程、桌面应用程序和图形化应用程序都依赖于对内核提供功能的频繁调用，如获取当前时间。也许你自己从未有这种感觉，但即便是你的桌面音量控制器都将频繁的查询当前音频混音器的状态。

用户程序可以通过系统调用这样一种抽象来请求内核代表它们执行某些特权操作。这些操作不能直接授权给用户程序来安全地执行，因为它们需要直接访问硬件（这将破坏用户空间的底层硬件抽象），如对整个系统放开，将导致它们被用户滥用。系统调用的典型例子包括open()一个文件、mmap()一些匿名内存（被标准内存分配例程（如malloc）所使用，以及用于更复杂的内存管理）或其他你可能原来认为是完全由GNU C函数库提供的功能。

事实上，许多系统C库函数只是实际执行请求任务的系统调用的包裹函数。C库函数用于提供一个执行系统调用的简单方式并执行任何不需要内核支持的轻量级任务。例如，当调用GNU C函数库中的exec()函数族将当前程序代码替换为一个新的程序并开始执行该新程序时，你实际上只调用了一个底层的系统调用，C函数库使用一些包装逻辑将它包裹以使你的工作变得更轻松。

在某些情况下，你可能期望原来直接由Linux内核处理的功能可以由系统C函数库更有效地完成而不会对系统造成任何整体的损害。每当执行一个系统调用时，一个潜在昂贵的（根据计算、内存访问、时间和其他东西来度量）从用户态到内核态的上下文切换就必须被执行^①（正如上一章已经说明的）。如果这是可以避免的，系统吞吐量就将增加。例如，一些系统不需要使用系统调用从CPU那里获得高精度的时间戳。

1. 跟踪系统调用

你可以使用工具如strace来查看系统中的系统调用，正如你在本书前面看到的那样。在strace中运行的应用程序所产生的每个系统调用（strace本身也使用ptrace()系统调用来执行跟踪——一个极为丑陋的代码技巧，本书没有介绍它，你最好也避免这样使用）以及传递给它的任何参数都将被显示。当你在strace中运行ls命令时，每当ls产生一个系统调用（包括用于建立该程序的C库函数中的系统调用），你都将看到相应的输出。

^① 用于缓存用户空间虚拟内存映射的被称为TLBs（变换索引缓冲区）的内部CPU硬件通常需要在此时被清空，而当用户进程再次运行时，其他类似的活动将导致额外的开销。与其他操作系统相比，Linux有着较低的系统调用开销，但它决非不存在。所以，减少系统调用有助于增加吞吐量。

为了对任何系统命令执行strace跟踪，只需使用strace包裹该程序：

```
$ strace program_name
```

当ls命令打开当前目录以列出该目录中包含的文件时，下面的open()系统调用成为一个很好的系统调用例子：

```
open(".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
```

当ls命令调用C库函数opendir()时，就会产生这个open()调用。opendir()需要打开代表目录内容的底层文件并列出文件列表。open()调用本身是在一个C函数库的桩函数中实现的，该桩函数代表程序产生合适的系统调用。这允许系统调用API的平台相关智能被限制到C函数库针对每个平台的单一部分从而避免对每个系统调用进行复制。open命令手册页的第2部分^①列出了open()调用一种可能的原型：

```
int open(const char *pathname, int flags);
```

从open函数的原型可以看出，很明显它期望得到一个要打开的磁盘上文件或目录的路径名以及一些标记。在上面的例子中，这些标记指定当前目录应该以只读方式打开、使用非阻塞IO（换句话说，即如果后续的IO操作不能立刻完成，它们不会阻塞而是返回一个错误以表明IO操作应该在稍后继续尝试）并使用Linux相关的O_DIRECTORY标记以表明除非这个路径名代表一个磁盘上的目录，否则就不打开该文件。这个系统调用返回一个新的文件描述符（3）。

- 将控制转移到内核：sys_open

对系统GLIBC函数库（例如/lib/libc.so.6）中实现的open函数的调用最终将导致一个平台相关的处理，它将控制转移到负责提供真正open()系统调用的特定内核函数中。例如，在PowerPC平台中，系统调用实际上是使用一个特殊的sc处理器指令实现的。而Intel IA32（x86）平台则使用一个特殊的处理器异常编号0x80（128）来实现同样的功能^②。无论实际使用的是哪种硬件相关的过程，其结果都是导致上下文切换到内核的sys_open()函数：

```
asmlinkage long sys_open(const char __user *filename, int flags, int mode)
{
    long ret;

    if (force_o_largefile())
        flags |= O_LARGEFILE;

    ret = do_sys_open(AT_FDCWD, filename, flags, mode);
    /* avoid REGPARM breakage on x86: */
    prevent_tail_call(ret);
    return ret;
}
EXPORT_SYMBOL_GPL(sys_open);
```

^① 正如你可能已知道的，Linux手册页根据类型被分为几个部分。每当有人引用open（2）或其他类似的手册时，你就知道他们指的是open手册页的第2部分。更多有关手册部分的信息请参考你的系统文档或查看man命令的帮助。

^② 新的Intel IA32（x86）机器实际上有一个更为现代的实现方式，它巧妙地使用了用户空间函数所处地址空间顶端的内存映射页。例如，GNU C函数库可以使用与调用函数库本身中普通函数一样的方式来调用进内核。桩代码使用新的SYSENTER处理器指令而不是sc来映射进每个用户空间进程。

真正提供open()系统调用功能的是更长（也更复杂）的do_sys_open()函数，它由内核源代码中的fs/open.c提供。如果想详细了解当打开一个文件或目录进行处理时究竟发生了什么，可以查看该文件。注意，sys_open()函数通过预处理的asmlinkage宏（说明这个函数是直接通过底层陷入处理器异常代码调用的）向GNU C编译器指定额外的连接信息，而EXPORT_SYMBOL_GPL(sys_open)的作用是将这个函数提供给外部内核模块。

大多数系统调用在内核中都是通过以sys_开头的函数实现的，这已成为一种事实上的用于确认一个系统调用的标准方式。你将在下一节中学习系统调用表如何提供必要的信息以将特定的系统调用编号和相应的函数调用对应起来。

2. 系统调用表

每个不同的Linux架构都有它们自己的通过系统调用进入内核的方法。常用的方法涉及对一个特殊机器指令的调用（通过内联汇编代码）。这将使得处理器进入其特权内核模式并导致对要执行的正确软件例程进行硬件辅助或软件辅助查找。这通常取决于系统调用编号，而该编号通常是在产生系统调用时被传递到一个处理器寄存器中。各种底层硬件在这一地方都有着很多灵活的实现方式。

在IA32（x86）架构中，系统调用是在系统调用表(syscall_table.S)中定义的，该文件位于内核源代码的arch/i386/kernel目录中。其他架构所使用的方法也类似（请参考针对你的特定架构的内核源代码以了解所使用方法的精确细节^①）。IA32系统调用表将系统调用编号5定义为open系统调用，因此，表中的第6个条目^②就包含一个指向sys_open()函数的指针。当产生一个系统调用时，调用编号将作为系统调用表中的一个直接偏移量用于调用相应的函数。

下面是Intel（x86）机器上系统调用表的摘录：

```
ENTRY(sys_call_table)
    .long sys_restart_syscall      /* 0 - old "setup()" system call,
                                         used for restarting */

    .long sys_exit
    .long sys_fork
    .long sys_read
    .long sys_write
    .long sys_open                /* 5 */
    .long sys_close
    .long sys_waitpid
    .long sys_creat
    .long sys_link
    .long sys_unlink              /* 10 */
```

在写作本书的时候，总共有317个系统调用被定义，当然对于不同的Linux架构来说，确切的数字将有所不同（理论上，甚至在基于特定架构的不同平台之间也会有区别——尽管这将是非常不寻常的），而且随着时间的推移，这一数字通常会逐渐增加。正如你可以看到的，新的调用如那些用于支持kexec()接口（它用于现代Linux内核崩溃转储的分析）的调用被附加到表格的结尾。系统调用编号从不会被复用，只是它们所对应的系统调用会被产生合适警告的哑函数所替换。

你可以在用户程序中通过使用由C函数库提供的一些简单的宏来手工调用在内核的系统调用表中

^① 在Linux内核源代码的arch目录中查找你的特定架构。

^② 原文为“the fifth entry”（第5个条目），但系统调用编号从0开始，所以这里应为“第6个条目”。——译者注

定义的任何系统调用。通常并不需要这么做，因为C函数库将把产生系统调用作为其标准功能的一部分。但有时候，C函数库可能还不知道出现了一个新的系统调用（尤其是当你自己编写了一个系统调用并将它添加到系统调用表中时）。在这种情况下，你最好使用简单宏的方式，或者从源代码开始重建你的C函数库。

下面是一个直接调用time系统调用的例子，它使用C函数库头文件来抽象在一个特定平台上使用的精确系统调用过程：

```
/*
 * Manually call the sys_time system call.
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <syscall.h>
#include <unistd.h>
#include <linux/unistd.h>

#define __NR_time 13 /* normally defined in <asm/unistd.h> */
_syscall1(long, time, time_t *, tloc);

int main(int argc, char **argv)
{
    time_t current_time;
    long seconds;

    seconds = time(&current_time);

    printf("The current time is : %ld.\n", seconds);

    exit(0);
}
```

系统调用编号首先被定义并分配给宏`__NR_time`。这个宏以随后的`_syscall1`宏所要求的格式被定义。`_syscall1`宏在Linux的头文件`unistd.h`中定义（请查看该头文件以了解具体的细节）。`syscall1`宏用于定义一个系统调用，它有一个参数并返回一个结果，当然还存在许多其他的宏分别针对拥有不同参数数目的系统调用。这个宏将被扩展为一个桩函数，该函数包含产生实际系统调用所需的最小数量的内联汇编代码。

在这个例子中，`time()`系统调用的用户空间桩函数使用如下的宏来定义：

```
_syscall1(long, time, time_t *, tloc);
```

在用户程序中随后对`time()`函数的调用将导致上述系统调用被执行并返回一个长整型的结果。

3. 实现自己的系统调用

你通常不会想实现自己的Linux系统调用。在你自己的可装载内核模块和类似的程序中可以使用更好的机制来与内核交互（继续阅读以了解详情）。事实上，如果没有一个非常好的理由（一个内核社区很可能接受的理由），如果你希望自己的系统调用能够被广泛提供，最好不要编写自己的系统调用。

本节的内容主要是考虑到对系统调用介绍的完整性——你需要理解这些内容以对Linux内核的核心进行重大的修改——但是，系统调用不应该在没有令人信服的理由以及没有上游社区的协调以确保唯一性的情况下被添加。与其编写一个新的系统调用，还不如阅读后面介绍接口（如Linux sysfs）的几节以使用其他机制。

请注意，当你实现自己的系统调用时，实际上修改了内核的二进制ABI，这可能会破坏未来与其他Linux内核的兼容性。也就是说，如果你要将一个新的系统调用编号319添加到系统调用表中，你需要确保今后没有人会将该编号用于其他系统调用——不断地修改应用程序代码以跟上系统调用编号的改变实在不是一件有趣的事情。

4. vsyscall优化

为完整性考虑，你应该注意添加到现代Linux内核中的vsyscall优化。在过去，Linux总是要使用C函数库的包裹函数中一个特殊的处理器指令来产生系统调用，而现代的硬件提供了多种开销较小的替代方案。现代内核中的vsyscall能力允许大多数平台上的所有用户应用程序依赖于内核来采用最合适进入机制。

使用vsyscall的应用程序在它们地址空间的高端有一个额外的内存映射页，应用程序与Linux内核直接共享该页。当程序启动并创建一个新进程时，内核使用正确的（并且更重要的是，最优化的）指令集来产生这个页，这个指令集是切换到内核并执行系统调用所必需的。在许多情况下，这可能只涉及特殊的处理器指令，但在某些有着更现代和更有效进入机制（如Intel IA32[x86]上的SYSENTER指令）的平台上，Linux使用这些机制来代替特殊的处理器指令。

你并不需要担心vsyscall或Intel的SYSENTER指令，你只需隐约意识到它的存在即可，因为它是由Linux内核直接处理的。要想了解更多有关vsyscall和sysenter的信息，请参考你感兴趣的特定Linux架构的kernel子目录或查看各种在线编写的介绍它们的文章。

8.2.2 设备文件抽象

Linux和其他类UNIX系统依赖于文件的基本抽象来作为代表许多不同的操作系统底层功能和设备的原语。正如人们常说的：“一切皆是文件”。事实上，Linux系统中的大多数设备都是由文件系统中的/dev目录下的文件所代表。在现代Linux发行版中，这个目录通常作为一个内存文件系统如tmpfs的装载点。这允许设备节点（一类用于代表底层物理硬件设备的特殊文件）在系统运行时被自由创建。

你可以使用ls命令来查看Linux系统中当前提供的设备文件：

```
$ ls -l /dev/
crw-rw---- 1 root audio      14,   4 2006-06-13 19:41 audio
crw-rw---- 1 root audio      14,  20 2006-06-13 19:41 audio1
lrwxrwxrwx  1 root root       3 2006-06-13 19:41 cdrom -> hdc
lrwxrwxrwx  1 root root       3 2006-06-13 19:41 cdrw -> hdc
crw----- 1 root root       5,   1 2006-06-13 19:41 console
lrwxrwxrwx  1 root root      11 2006-06-13 19:41 core -> /proc/kcore
drwxr-xr-x  3 root root      80 2006-06-13 19:41 cpu
brw-rw---- 1 root cdrom     22,   0 2006-06-13 19:41 hdc
brw-rw---- 1 root disk      33,   0 2006-06-13 19:41 hde
brw-rw---- 1 root disk      33,   1 2006-06-13 19:41 hde1
brw-rw---- 1 root disk      33,   2 2006-06-13 19:41 hde2
brw-rw---- 1 root disk      33,   3 2006-06-13 19:41 hde3
```

这里的输出仅仅显示了Linux系统中的一部分典型的设备文件。

设备文件有助于更容易地和以更一致的方式来表示底层硬件。设备文件（或“节点”）的类型只有几种，它们根据所代表的硬件设备类型的总体特征进行区分。通过使用文件，我们可以打开设备并使用常规的IO函数来与底层硬件设备交互。例如，你可以打开并读取一个磁盘文件，就如同整个硬盘是一个巨大的文件一样。对这一规则只有很少的例外情况^①。

历史上，Linux系统在/dev目录中有着数以万计（或成千上万）的文件来代表可能的设备。但如今，当Linux侦测到新的设备并加载了支持该设备的驱动程序后，设备文件守护进程如udev会自动创建相应的设备节点（文件）。这减少了不必要的设备文件的数量（它们本身还需要占用少量的硬盘空间）。你将在本书的第12章中了解到更多有关udev的内容，你還将在该章中学习D-BUS和Utopia项目以及它们是如何有助于简化一般系统通知过程。

1. 字符文件

在上面ls命令的输出中，第一列为c的文件就是字符设备。这包括总是按序读写（一次一个字符）的任何设备。你也许可以从字符设备读写整个缓冲区的数据，但底层的操作都是按序完成的。典型的字符设备包括键盘和鼠标，这是因为它们代表硬件的瞬时状态——你不会从鼠标输入中读取一个随机的位置。

在上面的ls命令列出的设备文件中，你可以看到几个音频设备。它们允许应用程序使用标准的Linux音频API通过系统的声卡播放声音。从根本上说，虽然你可以通过读取这些设备之一来记录一个声音并通过再次将它写回设备来回放声音。但在现实中，你需要设置声卡以启用输入的记录，但这里的关键是应用程序只需要担心几个特殊文件的处理即可。

- 创建一个字符设备

你编写的第一个Linux内核模块可能就会在某处涉及一个字符设备。人们传统上一直认为字符设备易于实现，并认为它是学习一般驱动程序编写的一个好方法。但在linux 2.6内核中，字符设备比过去要复杂得多，这有许多非常好的原因，尤其是当我们需要考虑设备的动态分配以及对用于支持每个字符设备的cdev结构的处理时更是如此。在21世纪，要想彻底摆脱存在于旧内核中的许多特殊代码（hack）已不再可能。

请看下面这个简单的示例代码：

```
/*
 * char.c - A simple example character device.
 *
 * Copyright (C) 2006 Jon Masters <jcm@jonmasters.org>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation.
 *
 */
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/major.h>
```

^① 一个明显的例外是Linux网络设备（netdevs），它们不以文件方式表示。

```

#include <linux/blkdev.h>
#include <linux/module.h>
#include <linux/cdev.h>

#include <asm/uaccess.h>

/* function prototypes */

static int char_open(struct inode *inode, struct file *file);
static int char_release(struct inode *inode, struct file *file);
static ssize_t char_read(struct file *file, char __user *buf,
                        size_t count, loff_t *ppos);

/* global variables */

static struct class *plp_class;      /* pretend /sys/class */
static dev_t char_dev;              /* dynamically assigned at registration. */
static struct cdev *char_cdev;       /* dynamically allocated at runtime. */

/* file_operations */

static struct file_operations char_fops = {
    .read     = char_read,
    .open     = char_open,
    .release  = char_release,
    .owner    = THIS_MODULE,
};

/*
 * char_open: open the phony char device
 * @inode: the inode of the /dev/char device
 * @file: the in-kernel representation of this opened device
 * Description: This function just logs that the device got
 *              opened. In a real device driver, it would also
 *              handle setting up the hardware for access.
 */
static int char_open(struct inode *inode, struct file *file)
{
    /* A debug message for your edutainment */
    printk(KERN_INFO "char: device file opened.\n");
    return 0;
}

/*
 * char_release: close (release) the phony char device
 * @inode: the inode of the /dev/char device
 * @file: the in-kernel representation of this opened device
 * Description: This function just logs that the device got
 *              closed. In a real device driver, it would also handle
 *              freeing up any previously used hardware resources.
 */

```

```

static int char_release(struct inode *inode, struct file *file)
{
    /* A debug message for your edutainment */
    printk(KERN_INFO "char: device file released.\n");
    return 0;
}

/*
 * char_read: read the phony char device
 * @file: the in-kernel representation of this opened device
 * @buf: the userspace buffer to write into
 * @count: how many bytes to write
 * @ppos: the current file position.
 * Description: This function always returns "hello world"
 * into a userspace buffer (buf). The file position is
 * non-meaningful in this example. In a real driver, you
 * would read from the device and write into the buffer.
 */

static ssize_t char_read(struct file *file, char __user *buf,
                       size_t count, loff_t *ppos)
{
    char payload[] = "hello, world!\n";
    ssize_t payload_size = strlen(payload);

    if (count < payload_size)
        return -EFAULT;

    if (copy_to_user((void __user *)buf, &payload, payload_size))
        return -EFAULT;

    *ppos += payload_size;
    return payload_size;
}

/*
 * char_init: initialize the phony device
 * Description: This function allocates a few resources (a cdev,
 * a device, a sysfs class...) in order to register a new device
 * and populate an entry in sysfs that udev can use to setup a
 * new /dev/char entry for reading from the fake device.
 */
static int __init char_init(void)
{
    if (alloc_chrdev_region(&char_dev, 0, 1, "char"))
        goto error;

    if (0 == (char_cdev = cdev_alloc()))
        goto error;
}

```

```

kobject_set_name(&char_cdev->kobj, "char_cdev");
char_cdev->ops = &char_fops; /* wire up file ops */
if (cdev_add(char_cdev, char_dev, 1)) {
    kobject_put(&char_cdev->kobj);
    unregister_chrdev_region(char_dev, 1);
    goto error;
}

plp_class = class_create(THIS_MODULE, "plp");
if (IS_ERR(plp_class)) {
    printk(KERN_ERR "Error creating PLP class.\n");
    cdev_del(char_cdev);
    unregister_chrdev_region(char_dev, 1);
    goto error;
}
class_device_create(plp_class, NULL, char_dev, NULL, "char");

return 0;

error:
printk(KERN_ERR "char: could not register device.\n");
return 1;
}

/*
 * char_exit: uninitialized the phony device
 * Description: This function frees up any resource that got allocated
 * at init time and prepares for the driver to be unloaded.
 */
static void __exit char_exit(void)
{
    class_device_destroy(plp_class, char_dev);
    class_destroy(plp_class);
    cdev_del(char_cdev);
    unregister_chrdev_region(char_dev, 1);
}

/* declare init/exit functions here */

module_init(char_init);
module_exit(char_exit);

/* define module meta data */

MODULE_AUTHOR("Jon Masters <jcm@jonmasters.org>");
MODULE_DESCRIPTION("A simple character device driver for a fake device");
MODULE_LICENSE("GPL");

```

这个示例驱动程序char.c通过module_init()和module_exit()声明了两个公开的入口点。它们用于处理驱动程序的装载和卸载,我们将在稍后对它们进行详细解释。这个驱动程序还利用MODULE_宏来定义作者和许可证并提供了对该驱动程序功能的一个非常简短的描述。当你将该驱动程序编译为

一个内核模块char.ko并使用modinfo命令显示该模块的信息时，这段描述将在该命令的输出中显示。最后，请注意这个模块是完全按照我们在上一章讨论的内核编码风格进行编写的——为代码添加详细的注释总是一个好主意。

- 编译驱动程序

为了试验这个示例驱动程序，你需要使用内核的kbuild系统来编译它。这个例子是在Linux内核版本2.6.15.6上进行测试的（本书中所有其他的内核示例也都是在这个内核版本上测试的），但它也应该可以运行在你可能安装的任何最新的Linux 2.6内核上（如果不行，你可以试着针对最新的内核版本对它进行修改，并将你的补丁通过E-mail发给出版者！）。正如我们在上一章中说明的，/lib/modules/2.6.15.6/build目录是对实际内核源代码目录的一个符号链接，所以你总是可以通过对uname命令^①的快速调用找出你所运行内核的源代码所在位置。

下面是编译Linux内核模块的正确方式：

```
$ make -C /lib/modules/`uname -r`/build modules M=$PWD
make: Entering directory `/data/work/linux_26/linux-2.6.15.6'
CC [M] /home/jcm/PLP/src/interfaces/char/char.o
Building modules, stage 2.
MODPOST
CC       /home/jcm/PLP/src/interfaces/char/char.mod.o
LD [M]   /home/jcm/PLP/src/interfaces/char/char.ko
make: Leaving directory `/data/work/linux_26/linux-2.6.15.6'
```

在成功编译（希望如此）之后，驱动程序char.ko将被创建。与上一章一样，你可以使用modinfo命令来显示这个驱动程序的更多信息：

```
$ /sbin/modinfo char.ko
filename:      char.ko
author:        Jon Masters <jcm@jonmasters.org>
description:   A simple character device driver for a fake device
license:       GPL
vermagic:     2.6.15.6 preempt K7 gcc-4.0
depends:
srcversion:   B07816C66A436C379F5BE7E
```

使用insmod命令在当前运行的内核中装载该驱动程序：

```
$ sudo insmod char.ko
```

由于驱动程序char.ko不在标准路径/lib/modules下，所以不能使用系统级的命令modprobe来装载该模块。这意味着必须使用insmod并指定.ko内核模块文件的具体位置——用户肯定不会习惯在他们自己的系统中这样做。与其使用insmod，还不如将模块安装到系统标准路径/lib/modules中以提供普遍的使用。

你可以通过重复上面的编译步骤，并将make的目标modules替换为modules_install来将驱动程序安装到/lib/modules目录中：

```
$ export INSTALL_MOD_DIR=misc
$ sudo make -C /lib/modules/`uname -r`/build modules_install M=$PWD
```

^① 正如你可能已经知道的，uname-r将返回正在使用的内核的版本号。针对正在运行的内核的模块位于/lib/modules/`uname-r`目录中，符号连接子目录build也位于该目录中。

```
make: Entering directory `/data/work/linux_26/linux-2.6.15.6'
  INSTALL /home/jcm/PLP/src/interfaces/char/char.ko
make: Leaving directory `/data/work/linux_26/linux-2.6.15.6'
$ sudo depmod -a
```

请注意`INSTALL_MOD_DIR`应该被设置为你希望驱动程序被安装到的内核模块目录下的位置——`misc`是一个好的选择（例如，`/lib/modules/2.6.15.6/misc`）。这样设置并不是必须的，但它有助于将你自己的模块与系统提供的标准模块分开。不要忘记运行`depmod`命令来更新`modprobe`使用的`module.dep`数据库。`depmod`命令的用法如上所示。

不论你使用什么方法（`insmod`或`modprobe`）装载了模块，你都希望能确定它是否已被正确地装载。为了确认这一点，请检查内核环缓冲区的输出，在本例的情况下，可以看到很多对象注册信息（你的系统有可能没有这么多输出信息，这取决于在运行内核的内核配置中所使用的冗长设置）。可以使用`dmesg`命令来查看内核环缓冲区：

```
$ dmesg|tail
[18023332.980000] kobject char: registering. parent: <NULL>, set: module
[18023332.980000] kobject_hotplug
[18023332.980000] fill_kobj_path: path = '/module/char'
[18023332.980000] kobject_hotplug: /sbin/udevsend module seq=1021 HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin ACTION=add DEVPATH=/module/char SUBSYSTEM=module
[18023332.996000] subsystem plp: registering
[18023332.996000] kobject plp: registering. parent: <NULL>, set: class
[18023332.996000] kobject char: registering. parent: plp, set: class_obj
[18023332.996000] kobject_hotplug
[18023332.996000] fill_kobj_path: path = '/class/plp/char'
[18023332.996000] kobject_hotplug: /sbin/udevsend plp seq=1022 HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin ACTION=add DEVPATH=/class/plp/char SUBSYSTEM=plp
```

除此之外，这个驱动程序确实工作了！可以尝试对新创建的`/dev/char`设备文件使用`cat`命令（以root身份执行，因为设备文件的默认权限不太可能包括普通用户账号）：

```
$ sudo cat /dev/char
hello, world!
hello, world!
hello, world!
```

你将看到来自设备文件的一个永不停止的`hello, world!`流。此外，在读取设备之后，可以看到内核环缓冲区中也包含了一个活动日志：

```
$ dmesg|tail
[18023447.656000] char: device file opened.
[18023449.544000] char: device file released.
```

每当`/dev`目录下的设备文件自身被打开和关闭时，它所对应的驱动程序就使用`printk()`日志函数记录这些信息。因此内核日志缓冲区中的输出为调试提供了最低形式的有用信息，但对于真正的驱动程序来说，它们可能并不想在日志中记录这些信息。

当你测试完毕后，不要忘记卸载这个驱动程序：

```
$ rmmod char
```

或使用modprobe来卸载：

```
$ modprobe -r char
```

应该可以看到这个示例Linux内核模块向内核环缓冲区产生了适当的输出。

- 关于动态设备的注记

尽管你将在本章的后面以及本书中学到更多有关udev和动态设备管理的内容，特别是驱动程序如何能够自动在/dev目录中注册它自己的动态设备条目。但当你编译了驱动程序并实际装载它之后，你可能希望通过/sys目录查看针对这个简单的字符驱动程序所新创建的sysfs的class类（显然，只有在装载了这个驱动程序之后，它才是可见的，它是对驱动程序中数据结构的一个伪文件系统表示）。

```
$ ls /sys/class/plp/char/
dev uevent
$ cat /sys/class/plp/char/dev
254:0
```

当驱动程序被装载后，/sys/class/plp目录被创建（由驱动程序）并产生一个针对由该驱动程序提供的char设备的子目录。当这些目录都被创建后，一个消息被发送给系统的udev动态设备守护进程，该进程将读取dev文件并确定要创建一个名为/dev/char的新设备，它的主设备号是254，次设备号是0——这就是/dev/char产生的过程，也是更多Linux内核将要采用的方法。以后不会再有硬编码的设备存在了。

你决不要在自己的模块中为新创建的设备节点使用硬编码的设备号。

- 驱动程序初始化

这个示例驱动程序是由modprobe或insmod装载的，然后内核负责运行初始化函数char_init()。这个示例Linux内核模块通过使用module_init()宏将该函数声明为模块的初始化函数。驱动程序首先尝试分配一个新的设备号——用于代表由该驱动程序支持的一类设备的主设备号。这个主设备号将出现在/dev/char设备文件中。所有的分配工作都是由alloc_chrdev_region()处理的，在这个例子中，它只分配一个设备号。

一旦分配了文件系统设备节点的设备号，一个相应的cdev结构将为内核提供字符设备及其属性的内存中表示。cdev包含一个kobject，它用于引用计数有多少内核的不同部分和任务引用（以某种方式使用）cdev。为cdev中嵌入的kobject关联一个名称char_cdev是为了提高可读性和有助于今后的调试。在旧的内核中，它还将显示在/sys/cdev目录中——在从sysfs中移除多余的cruft^①之前。

char_cdev指向的cdev结构还包含一个指向文件操作char_fops的指针，它与针对设备文件的各种不同类型的IO相关联。这个驱动程序只实现了几个文件操作——打开、释放和读取设备文件。其他未实现的文件操作企图（例如，ioctl或写操作）将返回一个错误。然后我们使用cdev_add将已分配的cdev与主设备号关联，从而使得设备和操作对应起来。

最后，这个示例驱动程序调用class_create()函数并在/sys/class/plp目录中创建一个新的“plp”sysfs的class类以容纳通过sysfs输出的驱动程序的所有属性（构成内核和用户空间之间的部分接口）。其中一个属性是dev文件条目，它由class_device_create()调用创建——它触发了用户

^① cruft的含义见本书第4章的解释。

空间的udev守护进程创建/dev/char设备节点。不要过于担心这些都是如何发生的——它非常复杂，只有很少的主流内核驱动程序真正搞清楚了它！

注意，错误是由goto语句处理的。尽管Edsger Dijkstra^①曾经说过goto语句现在被认为是有害的，但它仍然在内核代码中被大量使用——但只用在错误处理中。事实上，如果你需要在一个函数中尽可能多地复用错误处理代码并且希望能减轻内存分配或释放部分分配的资源，使用错误标签和goto语句将更清晰。

- 驱动程序的卸载

天下无不散之筵席，对示例模块来说也是一样。当我们调用modprobe或rmmod命令从内核中移除驱动程序时，char_exit()负责在模块被移除之前释放sysfs的class类、先前分配的cdev结构，并从系统中取消对主设备节点的注册。注意cdev_del()不得不应付有人仍然在试图使用这个设备的情况——它将一直等待直到cdev的引用计数变为0，然后才会真正地从内存中释放底层结构。

- 驱动程序的有效载荷

这个示例模块的目的是提供一个简单的字符设备，可以从该设备读取到一个永不停止的测试数据。这是通过一个定制的char_read()函数来实现的，它填充一个用户空间提供的数据缓冲区并返回写入的数据量。char_read()函数实际上是通过file_operations (fops) 结构挂上的，该结构由字符设备驱动程序的cdev结构中的ops成员所指向。写入用户缓冲区中的实际数据必须通过一个专用函数copy_to_user来完成以确保该用户缓冲区在写入时是在内存中^②。

我们鼓励你试验这个示例代码并扩展其功能。为什么不可以再设备被打开时（在char_open()函数中）分配一个大的内存缓冲区并使用它来存储从该设备读取或写入的数据呢？你可以在内核源代码目录的子目录drivers/char/中找到一些使用字符设备的有用的例子。

2. 块设备

在前面的/dev文件列表中，第一列为b的文件就是块设备。这包括完全随机寻址的任何设备，如磁盘驱动器。你想从一个普通磁盘上读取什么数据都没有关系，因为你总是可以指定你想要从哪里开始读取数据。磁盘驱动器由文件/dev/hda或/dev/sda来代表。在可分区块设备中的每个分区也由一个文件来代表，如/dev/hda1。

当编写块设备驱动程序时，你并不像在字符设备中那样直接处理用户IO，而是通过实现一系列函数以允许内核以一种有效的方式来执行面向块的IO（这是一个块设备的当务之急）。当用户需要在一个块设备上装载文件系统或执行一些其他类型的块设备相关活动（如对设备进行分区并建立一个文件系统）时，内核负责将该块设备呈现给用户。

创建一个自己的块设备驱动程序需要付出艰苦的努力并且需要你对Linux的块设备层有比本章除外介绍给你的还要多的了解。如果你对编写块设备驱动程序有兴趣，那么你需要查看Linux内核源代码以找出其他磁盘/块设备驱动程序的例子——为什么不了解一下你的磁盘控制器的驱动程序是如何实现对包含你的文件系统的块设备的抽象呢？你可能还想查看一些我们前面提到过的第三方的资

^① 荷兰计算机科学家，1972年图灵奖获得者，他最早提出“goto是有害的”，他在算法和算法理论、编译器、操作系统等诸多方面都做出了杰出贡献。——译者注

^② 由于用户空间可以自由地换入和换出，所以没有办法保证当内核想要写入某个用户空间时，它一定是在内存中。而且，也没有办法保证内核能够直接对一个用户空间缓冲区执行写入操作，所以你真的需要使用copy_to_user函数。

源——尤其是*Linux Device Drivers*一书。

3. 一切都是文件吗

Linux尽其努力来维护文件抽象，这样做确实很有意义的。但在现实世界中，要想使用文件抽象来表示Linux系统中的每一个底层设备既不实际也不可能。每一条规则都有例外，特别当遇到非常不寻常的设备或具有独特性能需求的软件栈时更是如此——如Linux网络设备和每一个Linux系统都使用的包过滤栈。

Linux中唯一被广泛采用而没有使用某种形式的文件抽象来表示的设备就是网络设备（但其他一些类UNIX操作系统以及更激进的系统（如Plan 9）则以文件来代表网络设备）。Linux网络栈使用特殊的接口（如netlink）和特殊的系统调用来建立和拆除网络配置。这意味着你不能使用cat这样的命令来轻易地读取原始网络数据包，但有工具可以做这件事情^①。

4. 机制与策略

实现机制而非策略是Linux内核程序员的基本目标。例如，内核并不需要知道只有Bob可以在9点到5点之间读写系统的声卡。内核需要做的只是实现一个机制为/dev/audio和/dev/dsp声卡接口添加文件权限，其他系统工具（如PAM函数库）可以为它们设置适当的权限。这样一来，今后对策略的改变就是小事一桩了。

有时候，虽然使用了文件抽象，但有些需要的功能不能通过文件操作来表示。ioctl()系统调用（或最近使用的特殊的sysfs文件）就是用于对设备进行特殊的控制。它们被设计用来弥补在使用普通文件IO操作来控制设备时偶尔带来的不足。例如，告诉磁带驱动器回绕磁带就不是一个典型的文件操作（没有标准的tape_rewind()系统调用或C库函数），所以它在内核中通过ioctl()系统调用来实现。ioctl()直接在设备文件自身上被执行。

每当你改变声卡混频器的音量时，你就会看到一个使用ioctl的实例——一个特殊的ioctl()被用于通知声卡驱动程序你希望修改混频器，而另一个特殊的ioctl被用于确定当前设置的声音级别。要想了解更多的信息，请参考在现有Linux内核驱动程序中的ioctl()实例或搜索因特网。

5. Procs

传统上，procfs伪文件系统是在系统调用之外进入Linux内核的主命令和控制接口。各种用户应用程序和工具都可以通过/proc来确定当前系统状态或调整系统参数，所有的工作都是通过读写简单的文件来完成。procfs一开始只是作为一个简单的机制用于查询正在运行进程的状态（这也是procfs名称的来历），但它很快就因其功能可以被轻易扩展而被过度使用。

可以通过列出/proc目录的内容来查看procfs文件系统中的典型条目：

	1	2115	27205	31915	4773	5703	7337	9271	kmsg
1063	22311	27207	31918	4918	5727	7339	9521	loadavg	
115	23732	27209	31921	4994	5730	7341	9804	locks	
116	24831	27210	31926	5	5732	7344	acpi	mdstat	
13266	2493	27456	31929	5003	5745	7352	asound	meminfo	
13278	24933	27457	32020	5012	5747	7360	buddyinfo	misc	
13286	25065	28597	32024	5019	5749	7379	bus	modules	
13289	25483	2870	32183	5026	5764	7381	cmdline	mounts	
13330	26136	28711	32186	5045	5781	7383	config.gz	mtrr	

^① libpcap函数库提供了一个通用的方法，它足以在大多数Linux系统上做到这一点。

13331	26410	28749	32189	5116	5784	7385	cpuinfo	net
14193	26557	2982	32193	5149	5814	7390	crypto	partitions
1727	2677	3	32195	5154	5816	7395	devices	pci
1738	26861	30191	32199	5162	5826	7397	diskstats	schedstat
1744	26952	30230	32221	5170	5884	7399	dma	scsi
1831	26962	30234	32226	5190	5906	7400	dri	self
19074	26965	30237	32236	5197	5915	7403	driver	slabinfo
19077	26966	30240	32280	5216	5918	7417	execdomains	stat
19078	26969	30270	32287	5256	5921	7429	fb	swaps
19162	26984	30273	32288	5263	5924	7601	filesystems	sys
19176	26997	3045	32416	5266	5927	7605	fs	sysrq-trigger
1929	27007	31079	32420	5267	6	8	ide	sysvipc
1935	27030	31118	3792	5269	6589	889	interrupts	tty
19411	27082	31415	3837	5290	704	9	iomem	uptime
19607	27166	31446	3848	5376	7253	9000	ioports	version
2	27174	31490	3849	5382	731	9016	irq	vmnet
20113	27180	31494	4	5388	7329	9023	kallsyms	vmstat
20270	27185	31624	4121	5397	7330	9024	kcore	zoneinfo
20278	27187	31799	4131	5401	7332	9025	keys	
20670	27193	31912	4260	5531	7335	9029	key-users	

你可以查询一个特定系统的相关信息，如处理器模型：

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id      : AuthenticAMD
cpu family     : 6
model          : 4
model name     : AMD Athlon(tm) processor
stepping        : 4
cpu MHz         : 1402.028
cache size     : 256 KB
fdtbug         : no
hltbug         : no
f00f_bug       : no
coma_bug       : no
fpu             : yes
fpu_exception   : yes
cpuid level    : 1
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 sep mttr pge mca cmov pat
pse36 mmx fxsr syscall mmxext 3dnowext 3dnow
bogomips       : 2806.44
```

你还可以调整各种系统参数，如启用IP转发（以root用户身份）：

```
$ echo 1 >/proc/sys/net/ipv4/ip_forward
```

可以轻松地执行这些操作使得procfs对那些仅仅需要一个进入内核的简单接口的内核开发人员有非常大的吸引力。例如，/proc/meminfo和/proc/slabinfo被无数的黑客用来确定内核中大量内存分配和管理功能的当前性能。当然，procfs最初的设计目的就是提供进程的信息，所以它可以做到这一点。

查看init进程（进程号为1）的相关信息：

```
$ sudo ls /proc/1
attr      cwd      fd      mem      oom_score  seccomp  statm  wchan
auxv     environ  loginuid  mounts   root      smaps   status
cmdline  exe      maps      oom_adj  schedstat stat      task
```

你可以看到包含指向该进程所使用当前文件描述符的目录 (fd)、用于调用该命令的命令行 (cmdline) 和该进程的内存映射 (maps)。/proc 目录中许多额外的信息由安装在一个典型 Linux 工作站或其他系统中的各种工具来提取。所涉及的各种操作要求用户拥有适当的安全能力以正确的执行它们，这通常包括需要使用一个 root 用户账号（除非使用了另一个 Linux 安全系统并对 root 用户的活动也进行了限制）。

• 创建 procfs 条目

procfs 传统上为任何想要创建 proc 文件的用户提供了足够低的进入门槛。事实上，像 `create_proc_read_entry()` 这样的函数使得这一过程比以前更容易。你可以在 `include/linux/proc_fs.h` 头文件中找到全系列的 procfs 函数及对每个函数的简短注释。但需要牢记的是，procfs 已不再像以前那样流行——但它对小数据集仍然非常适用。

下面的示例代码在 /proc 目录中创建了一个名为 plp 的可读条目，当我们读取该条目时，它将返回信息 hello, world!：

```
/*
 * procfs.c - Demonstrate making a file in procfs.
 *
 * Copyright (C) 2006 Jon Masters <jcm@jonmasters.org>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation.
 *
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>

/* function prototypes */

static int procfs_read_proc(char *page, char **start, off_t off,
                           int count, int *eof, void *data);
/* global variables */

static struct proc_dir_entry *procfs_file;

/*
 * procfs_read_proc: populate a single page buffer with example data.
 * @page: A single 4K page (on most Linux systems) used as a buffer.
 * @start: beginning of the returned data
 * @off: current offset into proc file
 * @count: amount of data to read
 * @eof: eof marker
 * @data: data passed that was registered earlier
 */
```

```
static int procfs_read_proc(char *page, char **start, off_t off,
                           int count, int *eof, void *data)
{
    char payload[] = "hello, world!\n";
    int len = strlen(payload);

    if (count < len)
        return -EFAULT;

    strncpy(page,payload,len);

    return len;
}

/*
 * procfs_init: initialize the phony device
 * Description: This function allocates a new procfs entry.
 */
static int __init procfs_init(void)
{
    procfs_file = create_proc_read_entry("plp", 0, NULL,
                                         procfs_read_proc, NULL);

    if (!procfs_file)
        return -ENOMEM;
    return 0;
}

/*
 * procfs_exit: uninitialized the phony device
 * Description: This function frees up the procfs entry.
 */
static void __exit procfs_exit(void)
{
    remove_proc_entry("plp", NULL);
}

/* declare init/exit functions here */

module_init(procfs_init);
module_exit(procfs_exit);

/* define module meta data */

MODULE_AUTHOR("Jon Masters <jcm@jonmasters.org>");
MODULE_DESCRIPTION("A simple driver populating a procfs file");
MODULE_LICENSE("GPL");
```

这个例子的结构和前面的关于Linux内核模块的例子非常相似，所以我们就不在这里对它进行详细的分析了。但有一点需要指出的是procfs基于页进行工作。这意味着当你写出数据时，你在任一时间最多只能使用4K（这与体系结构和平台相关）的缓冲区。试图通过procfs呈现很多数据一点也不有趣——而且也不是它设计的目的。一个现成的理由就是/proc目录中的大多数现有文件都是非常小的文本文件。此外，4KB的大小对于任何人来说都应该足够了。

• 编译驱动程序

这个驱动程序的编译过程与前面字符设备示例的编译过程非常相似。首先，你需要进入模块源代码目录并调用内核的kbuild系统来编译驱动程序：

```
$ make -C /lib/modules/`uname -r`/build modules M=$PWD
make: Entering directory `/data/work/linux_26/linux-2.6.15.6'
CC [M]  /home/jcm/PLP/src/interfaces/procfs/procfs.o
Building modules, stage 2.
MODPOST
CC      /home/jcm/PLP/src/interfaces/procfs/procfs.mod.o
LD [M]  /home/jcm/PLP/src/interfaces/procfs/procfs.ko
make: Leaving directory `/data/work/linux_26/linux-2.6.15.6'
```

然后你可以装载并测试这个驱动程序：

```
$ sudo insmod procfs.ko
$ cat /proc/plp
hello, world!
```

6. Sysfs

现代Linux系统包括一个新的被称为sysfs的用户可见的内核接口。sysfs最初是作为一种代表Linux内核电源管理子系统状态的机制被创建的。通过在/sys目录下的设备树中表示物理设备，系统不仅可以跟踪一个设备的当前电源状态（开、关、休眠，等等），而且还可以跟踪它与系统中其他设备之间的关系。起初，这样做只是为了让Linux能够确定各种操作之间的顺序。

例如，假设你有一个USB设备B，它连接到USB总线A。在理想情况下，内核不会试图在任何可能连接到USB总线A上的设备断电之前就将A断电。但如果内核不了解系统中物理总线的拓扑结构，Linux就可能会在连接到该总线上的设备的驱动程序还没来得及将这些设备置于静止状态之前关掉总线。在连接设备的驱动程序没有做好充分准备之前这么做可能不仅会导致数据丢失，还会导致内核崩溃。

sysfs在表现不同系统设备之间的关联方面做了非常好的工作，但它自身却遭受了特征蠕动（feature creep^①）。因为其底层实现和一个被称为kobject的内核中的数据结构有着密切关系——kobject用于Linux内核中的引用计数和许多其他功能——而且因为sysfs条目可以被轻易创建，所以它现在已被用在许多与其最初目的并不直接相关的额外功能上。

8.2.3 内核事件

有各种机制可以用来在内核和用户空间之间发送消息。其中一个较为有趣的是内核事件层，它大概在内核版本2.6.10出现的时候被添加进来。事件以消息的形式存在并被传递到用户空间，事件指向一个特定的sysfs路径（一个特定的sysfs kobject）。用户空间将根据存在的任何规则来处理消息——通常这是一个守护进程（如udev动态设备进程）的工作。

^① “feature creep”有时也理解为“需求漂移”。

内核事件的实现方式与原先的方案相比已经有所改变。最初，事件可以被动态定义，但现在kobject_uevent.h头文件定义了一个可被调用的标准事件列表——例如，KOBJ_ADD用于告诉用户空间刚刚新增了一些新的硬件（或驱动程序），它们可能需要一些额外的处理。

当你在本书后面阅读FreeDesktop.org项目时，你将学到更多有关标准系统事件的内容。现在，你可以通过查看内核源代码以找到更多这方面的示例。如果你想了解内核事件的实现方式，需要查看netlink套接字类型的使用。

8.2.4 忽略内核保护

有时候，为了提高效率或仅仅是为了完成一些原本不可能的事情，我们需要忽略由Linux内核及其抽象提供的各种保护。对那些定制设计的系统来说这是一种典型的情况，因为它们既没有必要担心普通用户，也没有必要向系统整合的问题妥协，系统整合已由你所需的修改提供了。许多嵌入式Linux系统在适当的地方利用了这样的黑客行为。

我们可以直接与物理硬件设备通信而不需要像其他应用程序那样使用一个内核设备驱动程序或底层的设备文件抽象。但系统对直接mmap()硬件时可以做的事情有一些限制：

- 并不是所有的硬件都可以在所有的平台上以这种方式进行内存映射。我们通常在支持块映射的32位嵌入式Linux环境中使用mmap()的硬件。
- 如果不为你的内核打上实验性的用户空间中断信号路由补丁，要想处理中断或将它们传递（路由）到一个用户空间程序是不可能的。依赖于中断驱动IO的硬件肯定需要内核的帮助。
- 你不能保证在用户空间中发生的硬件交互的及时性，即使你以可能的最高优先级使用了最新的低延迟（“实时”）补丁，你的程序仍然会受到内核内部活动的干扰。

如果你能够在这些限制下工作，下面的例子演示了如何通过/dev/mem设备节点打开并映射一段物理内存范围：

```
/*
 * peekpoke.c
 * Jon Masters <jcm@jonmasters.org>
 */

#include <linux/stddef.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define MMAP_FILE "/dev/mem" /* physical direct. */
#define MMAP_SIZE 4096      /* 4K. */

/* #define DEBUG 1 */

int use_hex = 0;

void display_help(void);
```

```

int valid_flag(char *flag_str);
void peek(char *address_str);
void poke(char *address_str, char *value_str);
unsigned long *map_memory(unsigned long address);

void display_help() {

    printf("Usage information: \n"
    "\n"
    "  peekpoke [FLAG] ADDRESS [DATA]\n"
    "\n"
    "Valid Flags: \n"
    "\n"
    "  -x      Use Hexadecimal.\n");
    exit(0);
}

int valid_flag(char *flag_str) {

    if (strncmp(flag_str, "-x", 2) == 0) {
        use_hex = 1;
    #ifdef DEBUG
        printf("DEBUG: using hexadecimal.\n");
    #endif
        return 1;
    }

    #ifdef DEBUG
        printf("DEBUG: no valid flags found.\n");
    #endif
        return 0;
}

void peek(char *address_str) {

    unsigned long address = 0;

    unsigned long offset = 0;
    unsigned long *mem = 0;

    #ifdef DEBUG
        printf("DEBUG: peek(%s).\n", address_str);
    #endif

    if (use_hex) {
        sscanf(address_str, "0x%lx", &address);
        /* printf("hexadecimal support is missing.\n"); */
    } else {
        address = atoi(address_str);
    }

    #ifdef DEBUG
        printf("DEBUG: address is 0x%x.\n", address);
    #endif

    offset = address - (address & ~4095);
}

```

```

address = (address & ~4095);

#ifndef DEBUG
printf("DEBUG: address is 0x%x.\n",address);
printf("DEBUG: offset is 0x%x.\n",offset);
#endif

mem = map_memory(address);

printf("0x%lx\n",mem[offset]);

}

void poke(char *address_str, char *value_str) {

unsigned long address = 0;
unsigned long value = 0;

unsigned long offset = 0;
unsigned long *mem = 0;

#ifndef DEBUG
printf("DEBUG: poke(%s,%s).\n",address_str,value_str);
#endif

if (use_hex) {
    sscanf(address_str,"0x%lx",&address);
    sscanf(value_str,"0x%lx",&value);
    /* printf("hexadecimal support is missing.\n"); */
} else {
    address = atoi(address_str);
    value = atoi(value_str);
}

#ifndef DEBUG
printf("DEBUG: address is 0x%x.\n",address);
printf("DEBUG: value is 0x%x.\n",value);
#endif

offset = address - (address & ~4095);
address = (address & ~4095);

#ifndef DEBUG
printf("DEBUG: address is 0x%x.\n",address);
printf("DEBUG: offset is 0x%x.\n",offset);
#endif

mem = map_memory(address);

mem[offset] = value;

}

unsigned long *map_memory(unsigned long address) {

int fd = 0;

```

```

unsigned long *mem = 0;

#ifndef DEBUG
    printf("DEBUG: opening device.\n");
#endif

if ((fd = open(MMAP_FILE,O_RDWR|O_SYNC)) < 0) {
    printf("Cannot open device file.\n");
    exit(1);
}

if (MAP_FAILED == (mem = mmap(NULL, MMAP_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED,
fd, address))) {
    printf("Cannot map device file.\n");
    exit(1);
}

return mem;
} /* map_memory */

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

/* test we got a sensible invocation. */

switch(argc) {

case 0:
    printf("Impossibility Reached.\n");
    exit(1);
case 1:
    display_help();
    break;
case 2:
    peek(argv[1]);
    break;
case 3:
    if (valid_flag(argv[1])) {
        peek(argv[2]);
    } else {
        poke(argv[1],argv[2]);
    }
    break;
case 4:
    if (valid_flag(argv[1])) {
        poke(argv[2],argv[3]);
    } else {
        printf("Sorry that feature is not supported.\n");
        display_help();
    }
    break;
default:
    printf("Sorry that option is not supported.\n");
    display_help();
    break;
}
}

```

```

}
exit(0);

} /* main */

```

如果你能够访问一个嵌入式参考板，可以尝试使用本例来点亮驻留在GPIO区域中的LED（例如，向位于0xe1000000的虚构的控制寄存器写入数据）：

```
$ ./peekpoke -x 0xe1000000 1
```

假定向控制寄存器写入1将点亮LED，而写入0将关闭LED，那么你可以通过使用一些简单的shell命令来让LED每秒闪亮一次：

```

$ while true; do
    ./peekpoke -x 0xe1000000 1
    sleep 1
    ./peekpoke -x 0xe1000000 0
done

```

这个示例程序的关键部分在map_memory()函数中。该函数接受一个指定的物理内存地址并尝试通过mmap()调用将它映射到程序的虚拟地址空间中。我们首先使用一个特殊的MMAP_FILE标记来打开设备文件/dev/mem，然后调用mmap()在新打开的文件描述符(open函数的返回值)上建立一个需要的映射。peek()和poke()函数只是对映射内存范围的读取和写入进行了简单的包装，这个程序还支持一些命令标记。

8.3 内部内核接口

你现在应该对用户应用程序和Linux内核通信的各种接口方式有一个总体的了解，但Linux内核还有其另外一面，即构成提供给第三方驱动程序（如你可能会编写的驱动程序）的API的那些函数之间的内部交互。在Linux内核中有着数以千计的函数，但输出的只是它们的一个子集——本节的目的就是对提供给你的那些API进行快速的浏览。

要注意的是，任何一本图书只能涵盖这么多的内容，尤其对于像本书这样的并非专门介绍内核编程的图书更是如此。要想更详细地了解内核函数，你需要查看下面这些文档：在线参考资料、邮件列表或图书——如*Understanding the Linux Kernel*和*Linux Device Drivers*。然而，没有哪个资源比内核源代码自身更有用，它们比任何图书可以提供的信息都要新。

8.3.1 内核 API

Linux内核包含了数以千计的函数，它们分布在数以百计的源文件中。可装载模块所使用的公开API的产生归功于遍布在内核源代码中的EXPORT_SYMBOL宏和EXPORT_SYMBOL_GPL宏的使用。前者为第三方模块提供了一个可用的符号，而后者为输出的函数附加了限制，它们不能提供给未使用GPL许可证的模块。在不陷入法律争论的前提下，我们只想说，你总是有充足的理由编写使用GPL许可证的内核模块。

虽然没有官方正式公布的标准化Linux内核API，甚至不能保证它们没有危害，但内核社区和任何其他人一样都不喜欢主要API改动所带来的影响——所以他们尽可能避免这样做。现在有各种工具用于帮助理清楚这些函数都被定义在内核的哪个位置以及它们是如何结合在一起工作的，如LXR——Linux

源代码交叉引用工具（Linux source Cross Reference）——解析Linux内核源代码并建立一个基于Web的函数、结构和其他数据类型的列表。Coywolf负责在网站<http://www.sosdg.org/~coywolf/lxr>上维护一个上游的LXR。

在一个给定Linux内核中公开输出的函数集（及其位置）被列在Module.symvers文件中，该文件位于已编译的内核源代码目录下。当编译启用版本控制功能的源代码树以外的模块时将会查询该文件，该文件和标准的System.map文件的结合还可以用于定位正在运行的内核中的任何你感兴趣的符号。

Linux每周新闻（LWN）的工作人员维护一个针对Linux内核的最新的API改动列表，可以通过访问位于<http://www.lwn.net>上的内核页面以找到这个列表。

8.3.2 内核ABI

一旦Linux内核被编译，它就会生成一个公开输出符号集（输出给可装载内核模块）的二进制版本，它被称为内核ABI——应用程序二进制接口或kABI。在上游内核中并不存在官方的Linux API或ABI。这意味着像kABI这样的术语仅指Linux内核的特定编译版本所提供的特定接口集。如果将内核升级到新的版本，kABI可能会有非常大的不同。

每当你针对一个Linux内核编译一个模块时，各种依赖于kABI的信息将被添加到模块中，这一点你可能并没有意识到。当你在Linux内核中使用模块版本控制（modversion）时更是如此。在这种情况下，模块还将包含用于编译它的内核接口的校验和信息。任何具备一个兼容kABI的内核都可以使用该驱动程序。虽然不能保证兼容性，但这个机制可以允许一个模块被用在与用来编译它的内核接近的内核版本中。

Linux并不需要一个稳定的上游kABI。这一想法已多次被提出，微软在它的Windows驱动程序模型中有一个稳定的ABI集，那在Linux中也设计一个类似的东西是不是一个好主意呢？从理论上说，这确实是一个好主意。但一个稳定kABI的缺点是使得我们难以以效率为名来修改它或干脆重写旧的黑客代码。微软（和其他一些公司）被强制要求在重写代码时提供兼容层，而这将迅速导致代码变得非常复杂。开发人员可能需要这样的一个机制，但它并不适合官方Linux内核。

虽然上游内核开发者对稳定的kABI并不感兴趣，但厂商却并非如此。尽管大多数组织都希望他们的Linux内核驱动程序最终出现在官方的上游内核中，但现实情况是，有很多“源代码树以外”（out-of-tree）的驱动程序并不会很快地出现在上游内核中。因此，厂商通常希望能保持内核ABI的兼容性从而可以确保源代码树以外的驱动程序可以在尽可能广泛的各种安装系统中被编译和使用。

内核模块软件包

两个较大的Linux厂商——Novell（OpenSuSE）和Red Hat（Fedora Core）已经开始在软件包装机制的软件包级别利用kABI跟踪技术。这些系统通常会提取用于模块版本控制的同一个信息并将它用在软件包依赖关系中。例如，在Fedora Core系统中，内核软件包可能会包含以下一些额外的kABI依赖关系（为简洁起见，进行了删减）：

```
kernel(kernel) = 329af1d5c0ddc4ac20696d3043b4f3e1cbd6a661
kernel(drivers_media_video_bt8xx) = 727583b2ef41a0c0f87914c9ef2e465b445d1cde
kernel(crypto) = d5a2dff1c7bb7a101810ce36149f4b9f510780c4
kernel(drivers_cpufreq) = 4bd57712e2f886b4962a159bcd28fccef34f88f86
kernel(sound_i2c_other) = c2e9f7ce21f9465661ec7cbff8ed0b18545b3579
kernel(net_802) = 887bebcd345ad585ecff84f3beb9b446e82934785
kernel(fs_configfs) = 844ef6e77e9da3e17f8cfe038d454265cabcl7e6
```

相关kABI依赖关系被组合进校验和中（为了避免书写8 000条依赖关系），这些校验和可以在稍后阶段被第三方提供模块所使用。如果校验和匹配，则表示第三方驱动程序可以放在标准化位置供内核使用。我们可以使用各种系统脚本来不断更新可用的兼容模块。

Fedora和OpenSuSE都公布了用于编译添加了这些依赖关系的驱动程序的指令，当针对这些发行版包装一个驱动程序时，你需要了解它们。有关此类模块包装方式的更多信息请查询社区网站<http://www.kerneldrivers.org>（由本书作者维护）。

8.4 本章总结

在本章中，你学习了Linux内核的接口。了解了存在着多种标准化接口，用户程序必须依赖于它们来与内核通信。这些接口必须在一段相当长的时间内保持一致，因为让所有应用程序都能针对外部内核接口的改动而进行相应的修改需要花费许多年的时间。

本章起着承上启下的作用，我们在本章中以Linux内核呈现的内部接口和外部接口的视角解释了一些重要的概念。在第9章中，你将运用在这几章中学到的知识并使用提供给Linux内核程序员的标准工具来帮助调试驱动程序。

第9章

Linux内核模块

在上两章中，你学习了Linux内核的开发过程，了解了上游内核社区并知道了内核是一个不断发展的项目。上一章的目的只是为了向你解释Linux究竟有多少会变化的需求（moving target）^①（这是很复杂的事情，但不用担心，在本书结束的时候没有课程测验），而本章将为你开始编写自己的Linux内核模块做好准备。这些模块通常被称为Linux内核驱动程序，因为大多数模块都用于支持特定的设备。

在第7章中已编写了一个简单的Linux内核模块plp，这个简单的可装载Linux内核模块只是输出一条欢迎信息到内核的环缓冲区（使用dmesg命令显示它），它演示了编写一个模块所涉及的基本概念，但没有进行深入的探讨。本章基于前面章节中的示例并介绍了更多的概念，这些概念都是你在编写实际使用的Linux内核模块时所需要掌握的。

并非所有的内核开发都能以第三方模块的形式完成。内核中许多部分的修改不能只通过可装载内核模块来实现，它们需要改动核心代码而不是简单地装载模块。虽然你首先将以可装载模块的形式来体验内核开发，但直到本章的结束你也不会改动核心内核算法。要做到这一点，你需要分配大量的时间来进一步学习并参与社区讨论。请记住，这不是一件简单的事情，你不可能在一个星期内就掌握它。

9.1 模块工作原理

Linux内核通过使用可装载内核模块来支持动态运行时扩展。可动态装载的Linux内核模块（或LKM）类似于你可以获取的针对Windows、一些专有UNIX（例如Sun Solaris）、Mac OS X和其他操作系统的驱动程序。LKM是针对一组特定的Linux内核源代码集建立的，它被部分链接到一个带有.ko后缀名的模块目标文件中，然后我们使用特殊的系统调用将它们装载到一个正在运行的Linux内核中。

当一个模块被装载进内核时，内核将为该模块分配足够的内存并从ELF的“modinfo”段^②中提取信息（该信息在模块被编译时添加到模块中）以便满足模块可能具有的任何其他需求。在装载模块时，模块最终将被链接到一个运行的内核版本中，该内核版本满足其所依赖的任何输出内核函数，而且它公开输出的符号也将成为运行内核的一部分。最后，模块成为内核的一部分。

你很可能现在就在依赖于可装载内核模块。如果你运行的是一个标准的Linux发行版，它几乎肯定会支持可装载内核模块。系统将装载一些模块以支持在系统启动时检测到的各种硬件，模块是由设备管理器如udev（它维护/dev文件系统以及为新检测到的设备装载模块）自动装载的。

你可以使用lsmod命令列出系统装载的模块：

① 在软件工程中，所谓“moving target”，就是指随着开发的进展，软件的需求可能不断发生变化。

② 正如第2章中介绍的那样，Linux使用ELF作为所有现代用户程序事实上的二进制格式标准（取代旧的、过时的a.out格式）。ELF只是为二进制数据（包括程序）定义了一个标准的容器格式，它将程序分成许多不同的段。

```
$ /sbin/lsmod
Module           Size  Used by
procfs          1732   0
char            2572   0
raw             7328   0
tun             9600   0
snd_rtctimer    2636   0
vmnet          35172  13
vmmon          111244  0
capability     3336   0
commoncap      5504   1 capability
thermal         11272   0
fan              3588   0
button          5200   0
processor       19712  1 thermal
ac               3652   0
battery         7876   0
8250_pnp        8640   0
8250            25492  3 8250_pnp
serial_core    18944  1 8250
floppy          59972   0
pcspkr          1860   0
rtc              10868  1 snd_rtctimer
usbnet          14024   0
sd_mod          16704   2
eth1394         18376   0
ohci1394        33268   0
ieee1394        292568  2 eth1394,ohci1394
emu10k1_gp      3072   0
ehci_hcd        32008   0
usb_storage     64128   1
scsi_mod        94760  2 sd_mod,usb_storage
ohci_hcd        19652   0
via686a         16712   0
i2c_isa          3968  1 via686a
uhci_hcd        31504   0
usbcore         120452  6 usbnet,ehci_hcd,usb_storage,ohci_hcd,uhci_hcd
parport_pc      38276   0
parport         33608  1 parport_pc
shpchp          44512   0
pci_hotplug    11780  1 shpchp
tsdev            6336   0
md_mod          64916   0
dm_mod          54552   0
psmouse         36420   0
ide_cd          38212   0
cdrom           37152  1 ide_cd
snd_cmipci      32288   1
gameport        12808  3 emu10k1_gp,snd_cmipci
snd_opl3_lib     9472   1 snd_cmipci
snd_mpu401_uart 6720   1 snd_cmipci
snd_emu10k1_synth 7040   0
snd_emux_synth  36160  1 snd_emu10k1_synth
```

```

snd_seq_virmidi      6464  1 snd_emux_synth
snd_seq_midi_emul    6784  1 snd_emux_synth
snd_seq_oss          32960 0
snd_seq_midi         7328  0
snd_seq_midi_event   6400  3 snd_seq_virmidi,snd_seq_oss,snd_seq_midi
snd_seq              50512 8
snd_emux_synth,snd_seq_virmidi,snd_seq_midi_emul,snd_seq_oss,snd_seq_midi,snd_seq_midi_event
snd_emu10k1          119652 5 snd_emu10k1_synth
snd_rawmidi          21600 4
snd_mpu401_uart,snd_seq_virmidi,snd_seq_midi,snd_emu10k1
snd_seq_device        7628  8
snd_opl3_lib,snd_emu10k1_synth,snd_emux_synth,snd_seq_oss,snd_seq_midi,snd_seq,snd_emu10k1,snd_rawmidi
snd_ac97_codec       94304  1 snd_emu10k1
snd_pcm_oss          50784  1
snd_mixer_oss         17728 3 snd_pcm_oss
snd_pcm               83460 4 snd_cmipci,snd_emu10k1,snd_ac97_codec,snd_pcm_oss
snd_timer             22404 5
snd_rtctimer,snd_opl3_lib,snd_seq,snd_emu10k1,snd_pcm
snd_ac97_bus          2176  1 snd_ac97_codec
snd_page_alloc        8904  2 snd_emu10k1,snd_pcm
snd_util_mem          3776  2 snd_emux_synth,snd_emu10k1
snd_hwdep             7840  3 snd_opl3_lib,snd_emux_synth,snd_emu10k1
snd                 47844 20
snd_cmipci,snd_opl3_lib,snd_mpu401_uart,snd_emux_synth,snd_seq_virmidi,snd_seq_oss,
snd_seq,snd_emu10k1,snd_rawmidi,snd_seq_device,snd_ac97_codec,snd_pcm_oss,snd_mixer_oss,snd_pcm,snd_timer,snd_hwdep
soundcore            8224  4 snd
r128                 46912 1
drm                  68308 2 r128
agpgart              29592 1 drm
8139too              23424 0
mii                  5120  1 8139too

```

这已经是不少了，但还只是针对一台普通的桌面电脑。

`lsmod`命令的输出显示了在这个Linux系统上共装载了77个模块。其中一些模块输出到内核命名空间中的符号没有被其他模块使用（第三列为0），而其他模块则构成了模块栈的复杂层次——例如下面这两种情况：ALSA音频驱动程序同时支持安装的两个声卡（Creative Labs emu10k芯片组和C-Media cmipci芯片组）（如果你支持将这些符号编译到你的内核中）以及使用各种已安装的USB模块。

9.1.1 扩展内核命名空间

可装载Linux内核模块通常会输出供Linux内核其他部分使用的新函数（你将在本章后面了解其工作原理）。这些新符号及其在内核映像中的地址将通过`procfs`的`/proc/kallsyms`显示（如果你支持将这些符号编译到你的内核中）。下面就是从一个运行内核的`/proc/kallsyms`中摘录的信息。首先，我们看符号表的开头：

```
$ cat /proc/kallsyms | head
c0100220 T _stext
c0100220 t rest_init
c0100220 T stext
```

```
c0100270 t do_pre_smp_initcalls
c0100280 t run_init_process
c01002b0 t init
c0100430 t try_name
c0100620 T name_to_dev_t
c01008e0 t calibrate_delay_direct
c0100a60 T calibrate_delay
```

下面显示的是内核符号表的结尾。因为新增加的模块符号被添加到符号表的结尾，所以你可以在下面的输出中看到一些模块符号（模块名列在方括号中——在本例中是MII以太网设备管理和控制的标准模块）：

```
$ cat /proc/kallsyms | tail
e0815670 T mii_check_media      [mii]
e08154f0 T mii_check_gmii_support [mii]
c023d900 U capable            [mii]
e0815600 T mii_check_link      [mii]
c011e480 U printk             [mii]
e08155a0 T mii_nway_restart    [mii]
c032c9b0 U netif_carrier_off   [mii]
e0815870 T generic_mii_ioctl   [mii]
c032c970 U netif_carrier_on    [mii]
e0815000 T mii_ethtool_gset    [mii]
```

可装载模块一直列到内核符号表的结尾，并在方括号中显示相应的模块名称，如上面的[mii]。你还可以通过符号表输出中第一列的地址来轻松地识别可装载内核模块。因为Linux内核通常链接到虚拟内存的3 GB空间以上，所以内置符号出现在3 GB (0xc0000000) 地址的附近，而装载的模块位于内核分配的内存中，在本例中位于0xe0000000地址以上，因为模块的内存是由内核在运行时分配的。

上面的示例Linux系统故意“污染”内核以向读者强调一点：这个运行内核装载了一个非GPL许可的模块以支持一个流行的商业虚拟机产品。对现代图形芯片组来说，这也是一个常见的例子。这类模块受到Linux内核社区的广泛抵制，一般来说，你会希望在自己的测试机器上删除这类内核模块——否则，你将得不到来自内核社区的任何支持。

9.1.2 没有对模块兼容性的保证

与其他操作系统不同，Linux（故意地）没有标准的驱动程序模型可以供你建立自己的驱动程序以保证它在不同Linux系统之间的兼容性。这意味着Linux针对模块源代码公开的情况进行了优化（通常情况也是如此——许多人认为非GPL许可的驱动程序侵害了内核开发人员的权利，所以在编写模块时不要尝试发布二进制驱动程序是一个非常好的主意）。

一般来说，你不应该假定针对你的桌面Linux内核编译的模块可以以预编译二进制格式的形式在另一台Linux机器上运行。这些模块通常需要在运行它的每一台不同的Linux机器上从源代码开始重新编译。但有一个明显的例外情况存在——厂商内核彼此之间通常是兼容的——但你不能指望它。

9.2 找到好的文档

Linux开发新手（尤其是那些开发内核的人员）所面临的最大问题之一就是找到一份好的API文档，

它能够给出开发人员必须使用的成千上万的函数的说明，以使得他们可以充分利用Linux内核提供的功能。除非你每天都在开发Linux内核，否则你不可能在此刻马上对Linux内核有非常深入的了解，所以一份好的文档将构成你的工作的一个重要部分。

正如你可能已猜到的那样，问题是没有这样的好文档存在。事实上，Linux内核一直在不断发展，所以即便有好的文档，当你阅读它时，它可能也已经过时了。在本书中，你将注意到作者并没有试图为Linux内核编写文档。相反，本书阐述内核的本质并提供范例以演示概念，你可以在自己的程序中应用这些概念。编写新的内核模块的最终文档来源于上游内核中现有的模块^①。

Linux 内核手册页

所幸的是，确实存在一个很好的API文档来源，它就是Linux内核手册页。这个文档集试图尽可能保持与Linux内核新版本一致，可以从kernel.org主网站的<http://www.kernel.org/pub/linux/docs/manpages>页面中找到它和许多其他资源。你还可以使用make mandocs命令从Linux内核源代码编译这些内核手册页：

```
$ make mandocs
MAN Documentation/DocBook/usb.9
Writing struct_usb_ctrlrequest.9 for refentry
Writing struct_usb_host_endpoint.9 for refentry
Writing struct_usb_interface.9 for refentry
etc.
```

使用make installmandocs命令安装它们以使它们可以供man命令使用。

你的Linux发行版可能有这些Linux内核手册页和普通手册页的预编译软件包，这样甚至连自己编译的步骤都省去了。作为这方面的一个示例，请在Debian、Ubuntu及其派生版本中查找“linux-manual”软件包。

下面显示的是如何查看vmalloc()内核内存分配函数的手册页：

```
$ man vmalloc
VMALLOC(9)                               LINUX                               VMALLOC(9)

NAME
    vmalloc - allocate virtually contiguous memory

SYNOPSIS
    void * vmalloc (unsigned long size);

ARGUMENTS
    size    allocation size

DESCRIPTION
    Allocate enough pages to cover size from the page level allocator and
    map them into contiguous kernel virtual space.

    For tight control over page level allocator and protection flags use
    __vmalloc instead.
```

^① 即由<http://www.kernel.org>网站提供的Linux内核。

DESCRIPTION

Allocate enough pages to cover size from the page level allocator and map them into contiguous kernel virtual space.

For tight control over page level allocator and protection flags use `_vmalloc` instead.

Kernel Hackers Manual

July 2006

VMALLOC(9)

当你开始开发内核时，可能会发现这些手册非常有用。也请你不要忘记去帮助编写那些因缺乏文档而让人沮丧的内核部分的文档。如果人人都能参与内核文档的弥补工作，其他开发人员的工作将变得更加轻松。

9.3 编写 Linux 内核模块

在第7章中为Linux内核开发了一个非常简单的示例程序。该示例程序演示了内核模块是如何编译和装载到一个（二进制兼容的）运行内核中的。通过使用内核的`printf()`函数，模块可以在其被装载进内核和从内核中卸载时进行记录。你现在应该掌握了编译内核和为该内核编译模块的基本过程。如果你错过或略过了第7章，请回头阅读该章并确保你已掌握了这些内容。

本节旨在涵盖编写有用的内核模块所涉及的一些要素，同时使你具备利用现有资源进行自我学习的能力。要想在一章的范围内解释编写内核代码所涉及的所有错综复杂的内容是不可能的（实际上，即使使用一本书来解释也不够——这需要实践），但通过将本章学习到的内容和对现有代码的研究以及上面提到的文档结合起来，你应该能够编写一些相当复杂的Linux内核模块了。

最后，你并不是只能编写与上游主线Linux内核分开的可装载模块。Linux所涉及的范围比你仅可以为之编写驱动程序的内核要大得多，它有整个社区的支持。Linux的目标始终是尽可能快地将新的代码放入主线内核中。同样地，你将发现有些自己想做的事情不能通过模块来实现，而需要改变“核心内核”代码自身。但本书并不特别包含这些内容，因为对核心内核的修改涉及许多方面，这最好留给专门讨论这一主题的专著。

9.3.1 开始之前

你已在第7章中学习了如何设置适当的开发和测试环境。请记住，即使是最有经验的Linux工程师也会犯错误或出现偶然的输入错误，并做其他一些愚蠢的事情，从而导致一个不稳定的测试环境。一旦失去了对内核中的指针和其他资源的跟踪，那么它们将永远消失。同样地，当写入内存时，你也可以毫无阻碍地破坏整个系统内存的内容。

这里的底线是，如果在编写内核代码的机器上对代码进行测试，很可能经常导致系统崩溃并不断地重启机器，所以请不要这样做。可以准备一台真正过时的旧机器来测试的代码，如果以后需要为某些特定硬件设备进行开发，可以将它们添加到一台合适的测试机器上。强烈建议在开发机器上使用网络共享来将内核和模块提供给测试机器——有关如何做到这一点的细节请参考在线参考资料。

这是很重要的一点，值得再次重申：切记不能期望测试机器保持稳定。

9.3.2 基本模块需求

每个Linux内核模块都需要一个已定义的入口点和出口点。入口点和出口点表明将要在模块装载

和卸载时调用的函数。此外，每个模块还应该定义作者、版本号并提供对其功能的描述，从而使用户、管理员和开发人员可以轻松地使用标准的module-init-tools工具（如modinfo）来获得已编译驱动程序模块的信息（正如在第7章、第8章中演示的那样）。

下面显示的是一个可以建立的最小的可装载模块示例：

```
/*
 * plp_min.c - Minimal example kernel module.
 */

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>

/* function prototypes */

static int __init plp_min_init(void);
static void __exit plp_min_exit(void);

/*
 * plp_min_init: Load the kernel module into memory
 */

static int __init plp_min_init(void)
{
    printk("plp_min: loaded");
    return 0;
}

/*
 * plp_min_exit: Unload the kernel module from memory
 */

static void __exit plp_min_exit(void)
{
    printk("plp_min: unloading");
}

/* declare init/exit functions here */

module_init(plp_min_init);
module_exit(plp_min_exit);

/* define module meta data */

MODULE_AUTHOR("Jon Masters <jcm@jonmasters.org>");
MODULE_DESCRIPTION("A minimal module stub");

MODULE_ALIAS("minimal_module");
MODULE_LICENSE("GPL");
MODULE_VERSION("0:1.0");
```

使用通常的方式来编译这个模块：

```
$ make -C `uname -r`/build modules M=$PWD
```

针对编译好的模块运行modinfo命令并查看它的输出：

```
$ /sbin/modinfo plp_min.ko
filename:      plp_min.ko
author:        Jon Masters <jcm@jonmasters.org>
description:   A minimal module stub
alias:         minimal_module
license:       GPL
version:      0:1.0
vermagic:     2.6.15.6 preempt K7 gcc-4.0
depends:
srcversion:   295F0EFEC45D00AB631A26C
```

注意输出中显示了模块版本以及针对该版本而生成的一个校验和。请养成总是为你的模块添加版本号的习惯，因为这样做可以使得那些必须包装它们的工作人员可以使用工具如modinfo来自动获取这个信息。

1. module.h

所有的Linux内核模块都必须包括<linux/module.h>以及其他一些它们需要的头文件。module.h定义了一些宏，内核编译系统使用它们给已编译的模块文件添加必需的元数据。可以通过查找编译过程中生成的以.mod.c为后缀名的C源文件来了解这些宏在编译过程中是如何使用的。这些源文件使用编译器指令详细标注以告诉GNU工具链如何将产生的模块连接到一个可以连接到内核中的特殊的ELF文件。还可以在该源文件中看到一个特殊的module结构定义。

表9-1显示了头文件中提供的一些特殊的宏和函数。这个表中显示的宏和函数应该在每个模块中的适当位置使用。它们都应该被视为函数（即便有些事实上是宏），并且通常都被插入到模块源文件的结尾。

表9-1 头文件中提供的一些特殊的宏和函数

宏/函数名	说 明
MODULE_ALIAS (_alias)	为用户空间提供模块的别名。它允许装载和卸载模块的工具认识这个别名。例如，intel-agp模块和许多其他PCI驱动程序一样，它指定由该驱动程序支持的一系列PCI设备ID，用户空间的工具可以使用这些设备ID来自动找到驱动程序 MODULE_ALIAS("pci:v00001106d00000314sv*sd*bc06sc00i00*")
MODULE_LICENSE (_license)	这个模块使用的许可证。注意，一些内核函数只提供给拥有被认可的自由许可证的LKM。例如，定义一个使用GPL许可证的模块： MODULE_LICENSE("GPL")
MODULE_AUTHOR	以姓名和电子邮件地址的形式指定模块的作者。例如： MODULE_AUTHOR("Jon Masters jcm@jonmasters.org")
MODULE_AUTHOR	(简要) 指定LKM的目的和功能。例如，在使用模块toshiba_acpi的情况下： MODULE_DESCRIPTION("Toshiba Laptop ACPI Extras Driver")

(续)

宏/函数名	说 明
module_param, MODULE_PARM_DESC	module_param 定义模块的装载参数（它既可以在insmod/modprobe命令行上指定，也可以在配置文件/etc/modules.conf中指定。参数包括名字、类型和对应sysfs文件的访问权限。例如，ndb（网络块设备）驱动程序以如下的方式定义了一个参数及其描述：
MODULE_VERSION	以[<epoch>:]<version>[-extra-version]>的形式指定模块的版本。epoch 和extra版本可以省略或根据需要使用。例如，iscsi_tcp模块定义了如下的版本：

2. init.h

Linux内核模块（LKM）还应包括<linux/init.h>。这个头文件包含了大量的定义，它们用于支持内核的启动以及支持通过module_init和module_exit宏插入和删除模块。你应该总是使用这两个宏来定义模块的入口点和出口点，因为当建立的内核不支持可装载模块时，这两个宏还将负责在内核中静态建立入口点和出口点函数。

因此你将从本书的示例模块中发现入口点和出口点函数是以如下方式定义的：

```
static int __init kmem_init(void);
static void __exit kmem_exit(void);
```

注意__init和__exit属性的使用，它们也定义在头文件init.h中，它们用于标记只在模块装载和卸载时使用的函数，如果内核是静态建立的，那么当内核正常启动以后，它将自动释放这些函数所用的内存。如果模块是以“正常”方式在内核运行时装载和卸载的，那么这些属性除了对源代码构成一个有用的文档注释形式以外就没有其他任何效果了。

9.3.3 日志记录

Linux内核通过/proc/kmsg提供日志记录机制，日志可以通过工具（如klogd）从用户空间读取。如果查看你的Linux系统，就会发现klogd进程一直在后台默默地运行着。klogd打开/proc/kmsg并将接收到的所有内核消息传递给标准的syslog守护进程，由该进程将日志信息记录到你的Linux发行版定义的系统日志文件中。内核消息记录的确切位置要根据Linux发行版和该消息所关联的优先级来确定——高优先级的消息可能会直接显示在系统主控台上。

在内核中，消息使用printf()函数发送给内核日志：

```
printf("my_module: This message is logged.");
```

这个函数在被调用时也可以有一个优先级设置（否则使用默认设置的优先级）：

```
printf(KERN_INFO "This is a regular message.");
printf(KERN_EMERG "The printer is on fire!");
```

优先级只是另一个用于确定消息将如何处理的字符串形式的宏。这些优先级以及实际的printf

函数本身都在<linux/kernel.h>中定义。例如，在2.6.15.6内核中，定义了如下的优先级：

```
#define KERN_EMERG      "<0>"      /* system is unusable          */
#define KERN_ALERT       "<1>"      /* action must be taken immediately */
#define KERN_CRIT        "<2>"      /* critical conditions          */
#define KERN_ERR         "<3>"      /* error conditions             */
#define KERN_WARNING     "<4>"      /* warning conditions           */
#define KERN_NOTICE      "<5>"      /* normal but significant condition */
#define KERN_INFO         "<6>"      /* informational                */
#define KERN_DEBUG        "<7>"      /* debug-level messages          */
```

从内部来看，`printk()`是一个神奇的函数。它的神奇在于它保证始终可以正常工作，无论它在
哪里被调用、身处什么样的上下文、持有什么样的锁都一样。它之所以可以工作得这么好是因为它将
消息写入一个（相当大的）环缓冲区，该缓冲区将在它被完全填满之前定期地刷新。这样做的目的是
为了允许即便在最恶劣的错误状况下也能进行日志记录，但也意味着当有非常多的消息被发送导致
`klogd`来不及读取时，确实有可能发生日志消息丢失的情况。

在某些架构中甚至还存在`printk`的早期版本，它允许你甚至在非常底层的代码中也可以
将日志记录到一个串行口主控台上——如果你对此关心，请检查一下你的系统。

9.3.4 输出的符号

Linux内核中充满着成千上万的函数和全局数据结构。如果你查看一下位于`/boot`目录中的
`System.map-kernelversion`文件或在内核编译过程中生成的`System.map`文件，你就会看到你正在
处理多少个符号。曾几何时，这些符号对内核代码的每一个部分都是全局可见的，但随着时间的推移，
人们已做出努力来降低不必要输出的符号对命名空间污染的程度。

你可以通过`Module.symvers`文件看到对可装载内核模块可见的符号，该文件也是在内核编译过
程中生成的，它位于内核编译目录的根目录中（例如，在`/lib/modules/`uname -r`/build`符号链
接目录中），当编译内核模块时，内核编译系统使用它来自动解决模块目标文件中需要表示的任何符
号依赖关系，该目标文件将被运行时模块装载机制使用。

在一个典型的系统中，`Module.symvers`文件中定义的符号可能超过4 000个：

```
$ cat /lib/modules/`uname -r`/build/Module.symvers | wc -l
4555
```

内核使用宏（如`EXPORT_SYMBOL`）将这些符号和数据结构明确地提供给内核的其他部分使用。内
核定义了两个这样的宏。

1. EXPORT_SYMBOL

`EXPORT_SYMBOL`宏允许在一个模块中定义的内核符号可以在该模块外使用。你会经常在构成一个
大型驱动程序栈的子系统和驱动程序代码中看到它，因为每个单独的模块都需要外部向它提供特定的
函数。尽管随着时间的推移，更多的输出符号随着新功能的出现而被添加，但人们仍然一直在不断努
力减少输出的符号数。

2. EXPORT_SYMBOL_GPL

Linux内核开发社区中很大一部分人认为由于Linux内核使用的是GNU通用公共许可证，所以它也
应该被延伸到内核模块。既然所有的内核模块都是Linux内核的扩展，那么它们就应该都遵循GPL许
可证。无论现实的法律情况如何，人们正在不断努力增加这个宏的使用。

`EXPORT_SYMBOL_GPL`将一个符号提供给可装载模块，但条件是这个模块遵循一个兼容的许可证。实际上，那些没有遵循兼容许可证的模块看不到以这种方式输出的符号。虽然Linux社区有一个象征性的保证，即他们不会不输出某些在过去一直提供的广泛使用的符号。如果你编写的是GPL许可的代码，就不必担心这一点了。

3. 不要挑战系统

在阅读本节时，你可能会感到困惑。既然内核可以访问它愿意访问的任何内存（Linux并不是一个多层次的微内核），模块或其他内核代码当然应该可以在内核的范围内做任何它们想做的事情。如果模块想要访问一个未输出的符号，它只需要越过一些额外的障碍即可——计算偏移量或使用其他一些合适的技巧就足以到达你需要到达的任何内存位置，这样做对不对呢？

请永远不要这么做。某些内核代码就有这样的问题，它只会带来麻烦。

内核可以自由地访问任何它想访问的内存，所以我们确实可以在内核的任何位置访问任何数据或函数。然而，如果一个函数或数据结构没有被输出，就不应该使用它。新的内核程序员往往决定利用黑客技术使用原本不能从模块中访问到的数据结构（系统调用表就是一个这样的示例）。不应该使用这些未输出的符号，因为它们没有被合适的锁或其他机制保护。

我们以系统调用表为例。在2.5-2.6内核开发周期中，人们抱怨系统调用表没有被输出，从而导致内核模块不能在运行时添加它们自己的系统调用。在过去，图书和其他参考资料对此都有建议的修改方案作为一种添加或修改系统调用的方法，特别是使用允许拦截的模块，或使用那些在运行时为一个系统调用补上安全漏洞的模块。

不幸的是，修补系统调用表将引发令人厌恶的副作用，这不仅可能发生在修补过程中，也可能发生在当移除模块并必须确定如何恢复表项时。如果`sys_open`在模块运行时被覆盖，你不能确定当做出修补的模块被移除时，你是否可以恢复`sys_open`函数，因为该函数可能被修改了两次，或有可能当该系统调用被移除的时刻，正好它被调用——引发一个无效的指针在内核中被解引用。

系统调用表和其他结构从全局内核命名空间中移除并取消输出有非常正确的理由，你不应该在可装载模块中使用它们。如果你需要的符号没有被输出，请在公开邮件列表中申请要求重新输出它之前首先考虑为什么会这样。如果你不这样考虑，那么它最终将给你带来麻烦。

9.3.5 分配内存

Linux内核和用户空间的程序不一样，它不能利用标准库函数（如`malloc`）。此外，`malloc`实际上依赖于内核来完成它的底层内存分配，并且在必要的时候它会频繁地调用进内核来获取更多的内存以支持其内部桶内存分配结构。大多数时候，你大概从不会考虑应用程序的内存是从哪里来的，它总是在那里等待你的申请。但当你开发Linux内核时，你将发现事情不是那么简单了。

Linux内核根据请求的内存类型、请求的时间和其他各种准则采用不同的方法来管理其内部内存。你通常使用两种分配函数之一：`kmalloc`或`vmalloc`。在本节中，你将发现这两个函数之间的区别并将学习如何使用宝贵的内核内存。你还将了解更多有关更高级分配机制的内容。

1. `kmalloc`

你通常会在自己的模块中使用的最原始的内存分配函数是`kmalloc`。它是一个底层的物理页面级分配器，它保证其分配的所有内存中物理内存中是连续的。`kmalloc`使用一个`size`参数和一组标记并

返回一个指向分配内存的指针（每当由于各种原因导致分配不能完成时，它将返回NULL）：

```
pointer = kmalloc(SIZE_IN_BYTES, ALLOCATION_FLAGS)
```

可以分配的内核内存大小是十分有限的，它不应该超过几千个字节，因为你所依赖的系统启动以来的各种活动并没有占用足够连续的物理内存。一旦结束使用内存，必须调用`kfree()`将内存返还给内核：

```
kfree(pointer)
```

如果不将内存返还给内核，它就将永远消失。内核中并没有运行一个垃圾回收器，而且当模块从内核中移除时，内核也不会自动清理它所使用的内存——内存将消失并且不能被再次使用，直到系统重启为止。所以你一定要确认你已正确地跟踪了你的驱动程序使用的所有内存，否则系统将慢慢出现内存饿死的状况。

应该为内存的分配指定下列标记之一（标记定义在头文件`<linux/mm.h>`中）：

- GFP_ATOMIC

内存分配将被立刻执行而不会睡眠，这个标记通常在不可能睡眠的情况下使用——例如，在内核中断处理程序中。内存要么被立刻分配（如需要，将从一个特别的保留区域中分配），要么内存分配将失败而不会导致阻塞。

- GFP_KERNEL

正常的内核内存分配应该使用这个标记来执行。它由`_GFP_WAIT`、`_GFP_IO`和`_GFP_FS`结合构成。这组标记允许内核阻塞并等待更多的内存可用。它允许内存来自对文件系统缓存的同步刷新或来自其他IO活动。该标记的一个变体`GFP_NOFS`用在各种文件系统代码中，因为这些代码不可能重新装入内核以刷新文件系统缓存。

- GFP_USER

每当分配的内存将返回给用户空间时，就应该使用这个标记。它将把分配的内存清零，其他方面与上面的`GFP_KERNEL`很相似。

在分配内存时还有其他各种分配标记可以使用。请查询`<linux/mm.h>`及其包含的`<linux/gfp.h>`以找到可用标记集——虽然你几乎可以肯定不使用它们。

2. `vmalloc`

很多时候，并不需要一个大型而连续的物理系统内存块，而是需要一个足够大的缓冲区用于在内核内部使用。`vmalloc`函数就是用于满足内核代码的这一需求的，它还允许更大的内存分配，但分配的空间在物理内存中不是连续的。这意味着许多想要处理连续内存（例如在不涉及复杂的描述内存中许多单个缓冲区的分散-收集链表的常规DMA操作中）的设备将无法正确处理`vmalloc()`分配的内存。

你可以以下方式使用`vmalloc()`来分配内存：

```
pointer = vmalloc(SIZE_IN_BYTES)
```

使用`vfree()`释放内存：

```
vfree(pointer)
```

在使用`vmalloc`时，无法拥有和使用`kmalloc`时一样程度的控制权（所以，在某些情况下，不应该使用`vmalloc`）。`vmalloc`的典型用途包括在模块启动时分配大型内部缓冲区或每当一个设备被打开以执行一些新操作时分配较小的缓冲区。内核负责分配物理内存以支持连续的虚拟内存。

3. plp_kmem: 内核内存缓冲区

你可以通过编写一个简单的允许用户读写一个内存缓冲区的内核模块来测试各种内存分配函数。这个模块需要分配一个大型缓冲区用于支持存储，它应该使用vmalloc函数来做到这一点，因为分配的内存并不需要在物理内存中是连续的，但却需要大于kmalloc可以分配的有限内存。

下面的示例代码在/dev目录下提供了一个名为plp_kmem的新设备，它代表一个你可以读写数据的（默认）大小为1MB的内存缓冲区。这个模块使用内核函数vmalloc来分配一个大小为PLP_KMEM_BUFSIZE(1MB)字节的大型缓冲区：

```
if (NULL == (plp_kmem_buffer = vmalloc(PLP_KMEM_BUFSIZE))) {
    printk(KERN_ERR "plp_kmem: cannot allocate memory!\n");
    goto error;
}
```

请注意上面错误处理代码的使用，这是因为内核中所有的内存分配都可能会失败。在plp_kmem_init函数中除了使用vmalloc进行内存分配以外，还有许多其他函数在执行时也会间接地导致内存的分配。例如，它调用了一个名为class_create的函数（创建条目/sys/class/plp_kmem已在上一章介绍的动态设备注册机制中使用），这个函数在其内部可能需要使用kmalloc执行小内存分配：

```
plp_kmem_class = class_create(TTHIS_MODULE, "plp_kmem");
if (IS_ERR(plp_kmem_class)) {
    printk(KERN_ERR "plp_kmem: Error creating class.\n");
    cdev_del(plp_kmem_cdev);
    unregister_chrdev_region(plp_kmem_dev, 1);
    goto error;
}
```

因为当系统处于沉重的内存压力下时，class_create函数的内部操作可能会失败，所以上面的代码中放置了更多的错误处理代码。请记住内核与普通程序不同，后者中的内存泄漏通常不会造成系统崩溃。在内核中，用于内核其他部分的所有内存分配和数据结构的注册都必须在错误路径处理中被释放。内核中频繁地使用goto语句来简化错误处理，因为这样一来，函数可以在必要的时候直接跳转到常见的错误处理代码块。

一旦内核模块完成了它的工作并且要求将它从运行内核中卸载，该模块分配的内存也必须返回给内核。示例模块通过调用vfree()函数来完成这一工作：

```
vfree(plp_kmem_buffer);
```

在示例代码中，你看到的一些结构将直到本书的后面才会得到解释——但你不需要为此担心，因为设备的创建、删除和sysfs都已得到详细的解释。

```
/*
 * plp_kmem.c - Kernel memory allocation examples.
 */

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/vmalloc.h>

#include <linux/fs.h>
#include <linux/major.h>
#include <linux/blkdev.h>
```

```

#include <linux/cdev.h>
#include <asm/uaccess.h>
#define PLP_KMEM_BUFSIZE (1024*1024) /* 1MB internal buffer */
/* global variables */

static char *plp_kmem_buffer;

static struct class *plp_kmem_class; /* pretend /sys/class */
static dev_t plp_kmem_dev; /* dynamically assigned char device */
static struct cdev *plp_kmem_cdev; /* dynamically allocated at runtime. */

/* function prototypes */

static int __init plp_kmem_init(void);
static void __exit plp_kmem_exit(void);

static int plp_kmem_open(struct inode *inode, struct file *file);
static int plp_kmem_release(struct inode *inode, struct file *file);
static ssize_t plp_kmem_read(struct file *file, char __user *buf,
                           size_t count, loff_t *ppos);
static ssize_t plp_kmem_write(struct file *file, const char __user *buf,
                           size_t count, loff_t *ppos);

/* file_operations */

static struct file_operations plp_kmem_fops = {
    .read      = plp_kmem_read,
    .write     = plp_kmem_write,
    .open      = plp_kmem_open,
    .release   = plp_kmem_release,
    .owner     = THIS_MODULE,
};

/*
 * plp_kmem_open: Open the kmem device
 */
static int plp_kmem_open(struct inode *inode, struct file *file)
{
#ifdef PLP_DEBUG
    printk(KERN_DEBUG "plp_kmem: opened device.\n");
#endif

    return 0;
}

/*
 * plp_kmem_release: Close the kmem device.
 */
static int plp_kmem_release(struct inode *inode, struct file *file)

```



```

{
    size_t bytes = count;
    loff_t fpos = *ppos;
    char *data;

    if (fpos >= PLP_KMEM_BUFSIZE)
        return -ENOSPC;

    if (fpos+bytes >= PLP_KMEM_BUFSIZE)
        bytes = PLP_KMEM_BUFSIZE-fpos;

    if (0 == (data = kmalloc(bytes, GFP_KERNEL)))
        return -ENOMEM;

    if (copy_from_user((void *)data, (const void __user *)buf, bytes)) {
        printk(KERN_ERR "plp_kmem: cannot read data.\n");
        kfree(data);
        return -EFAULT;
    }

#ifdef PLP_DEBUG
    printk(KERN_DEBUG "plp_kmem: write %d bytes to device, offset %d.\n",
           bytes, (int)fpos);
#endif

    memcpy(plp_kmem_buffer+fpos, data, bytes);

    *ppos = fpos+bytes;

    kfree(data);
    return bytes;
}

/*
 * plp_kmem_init: Load the kernel module into memory
 */
static int __init plp_kmem_init(void)
{
    printk(KERN_INFO "plp_kmem: Allocating %d bytes of internal buffer.\n",
           PLP_KMEM_BUFSIZE);

    if (NULL == (plp_kmem_buffer = vmalloc(PLP_KMEM_BUFSIZE))) {
        printk(KERN_ERR "plp_kmem: cannot allocate memory!\n");
        goto error;
    }

    memset((void *)plp_kmem_buffer, 0, PLP_KMEM_BUFSIZE);

    if (alloc_chrdev_region(&plp_kmem_dev, 0, 1, "plp_kmem"))
        goto error;

    if (0 == (plp_kmem_cdev = cdev_alloc()))
        goto error;

    kobject_set_name(&plp_kmem_cdev->kobj, "plp_kmem_cdev");
}

```

```

plp_kmem_cdev->ops = &plp_kmem_fops; /* file up fops */
if (cdev_add(plp_kmem_cdev, plp_kmem_dev, 1)) {
    kobject_put(&plp_kmem_cdev->kobj);
    unregister_chrdev_region(plp_kmem_dev, 1);
    goto error;
}

plp_kmem_class = class_create(THIS_MODULE, "plp_kmem");
if (IS_ERR(plp_kmem_class)) {
    printk(KERN_ERR "plp_kmem: Error creating class.\n");
    cdev_del(plp_kmem_cdev);
    unregister_chrdev_region(plp_kmem_dev, 1);
    goto error;
}
class_device_create(plp_kmem_class, NULL, plp_kmem_dev, NULL, "plp_kmem");

printk(KERN_INFO "plp_kmem: loaded.\n");

return 0;

error:
printk(KERN_ERR "plp_kmem: cannot register device.\n");
return 1;
}

/*
 * plp_kmem_exit: Unload the kernel module from memory
 */

static void __exit plp_kmem_exit(void)
{
    class_device_destroy(plp_kmem_class, plp_kmem_dev);
    class_destroy(plp_kmem_class);
    cdev_del(plp_kmem_cdev);
    unregister_chrdev_region(plp_kmem_dev, 1);

    vfree(plp_kmem_buffer);

    printk(KERN_INFO "plp_kmem: unloading.\n");
}

/* declare init/exit functions here */

module_init(plp_kmem_init);
module_exit(plp_kmem_exit);

/* define module meta data */

MODULE_AUTHOR("Jon Masters <jcm@jonmasters.org>");
MODULE_DESCRIPTION("Demonstrate kernel memory allocation");

MODULE_ALIAS("memory_allocation");
MODULE_LICENSE("GPL");
MODULE_VERSION("0:1.0");

```

● 编译和测试示例模块

你可以使用如下命令来编译这个示例模块：

```
$ make -C /lib/modules/`uname -r`/build modules M=$PWD
make: Entering directory `/usr/src/kernels/2.6.17-1.2517.fc6-ppc'
CC [M]  /home/jcm/modules/plp_kmem/plp_kmem.o
Building modules, stage 2.
MODPOST
CC      /home/jcm/modules/plp_kmem/plp_kmem.mod.o
LD [M]  /home/jcm/modules/plp_kmem/plp_kmem.ko
make: Leaving directory `/usr/src/kernels/2.6.17-1.2517.fc6-ppc'
```

使用下面的命令将编译好的内核模块装载到运行内核中：

```
$ /sbin/insmod ./plp_kmem.ko
```

你将在内核日志中看到一些行为的记录：

```
$ dmesg|tail
plp_kmem: Allocating 1048576 bytes of internal buffer.
plp_kmem: loaded.
```

这个模块还在/dev/plp_kmem中注册了并创建了一个设备，你可以对它执行读写操作：

```
$ echo "This is a test" >/dev/plp_kmem
$ cat /dev/plp_kmem
This is a test
```

试图向这个设备写入过多的数据将导致一个预期的错误：

```
$ cat /dev/zero >/dev/plp_kmem
cat: write error: No space left on device
```

一旦完成对它的使用，就可以卸载该设备：

```
$ /sbin/rmmod plp_kmem
```

模块把卸载信息记录到内核日志中：

```
$ dmesg|tail
plp_kmem: unloading.
```

关于udev的注记

在下一章中，你将学习/dev/plp_kmem中的动态设备节点是如何在装载模块时被创建以及如何在卸载模块时被删除的。动态设备文件系统的一个副作用是它将针对它不认识的设备节点使用默认权限——没有人将plp_kmem告诉给udev！这样做的结果导致它有一些非常严格的默认权限。在Debian系统中，这些默认权限可能是：

```
$ ls -l /dev/plp_kmem
crw-rw---- 1 root root 253, 0 2006-08-14 13:04 /dev/plp_kmem
```

而Fedora系统可能更严格：

```
$ ls -l /dev/plp_kmem
crw----- 1 root root 253, 0 2006-08-14 13:04 /dev/plp_kmem
```

你可以通过明确告诉udev这个新模块来解决这一问题。要做到这一点，请将如下内容添加到新文件/etc/udev/rules.d/99-plp.rules中：

```
# Add a rule for plp device
KERNEL=="plp_*", OWNER="root" GROUP="root", MODE="0666"
```

9.3.6 锁的考虑

Linux内核提供了人们期望在任何编程环境中看到的常用的锁原语。其中最主要的两个锁原语是自旋锁和信号量（在使用二元操作模式时，它也被称为互斥量）。因为每一种锁原语针对的都是不同的问题，所以当需要保护数据以避免它被同时访问时，典型的内核模块会同时利用它们两个（以及其他更复杂的锁原语）。

1. 信号量

信号量用于控制对指定资源的并发使用，它也被称为是休止锁（sleeping lock）。一个信号量允许你确保只有一个（或指定数目的）用户在某一特定时刻访问一个指定资源。每当需要进入一个代码的临界区时，可以对一个指定的信号量调用down()以对一些数据执行必要的操作，然后调用up()释放这个信号量以让等待该信号量的下一个任务可以被再次唤醒。

显然，信号量只适合在内核中睡眠的地方使用。这使得它们非常适用于那些服务于用户系统调用的函数，因为当用户任务无法获取它请求的资源时，内核可以简单地将该任务置于睡眠状态。一旦另一个用户（通常是另一个任务）完成对受保护数据执行的一些操作并释放信号量后，正在等待的睡眠任务就可以被唤醒并可以自己控制信号量了。

信号量定义在头文件<asm/semaphore.h>中，它可以以如下方式声明：

```
struct semaphore my_sem;
```

你通常使用的一种被称为互斥量（mutex）的特殊形式的信号量。互斥量在同一时间只能被一个任务持有，而完整的信号量实际上可以在需要的时候被初始化以处理大量的任务。你可以使用如下的宏来声明和初始化一个互斥量（然后就可以使用它了）：

```
DECLARE_MUTEX(my_lock);
```

信号量必须通过down()调用来获得：

```
down(&my_sem);
```

或（最好）通过down_interruptible()调用：

```
down_interruptible(&my_sem);
```

你通常会用down_interruptible()调用，因为它不像down()调用那样阻塞发往当前任务的信号。当任务正在等待获取信号量时，如果down_interruptible()调用被一个传递给该任务的信号中断，它将返回一个正数值。你通常希望如此，否则，如果模块驱动的物理设备出现了硬件故障，它将被无限期阻塞，这意味着该任务永远不会再次被唤醒了。因为它不会响应信号，所以该任务也将不能被杀掉。

信号量使用up()调用释放：

```
up(&my_sem);
```

下面是一个声明和使用信号量的示例：

```
static DECLARE_MUTEX(my_lock);
down(&my_lock);
/* perform some data manipulation */
up(&my_lock);
```

2. 自旋锁

Linux 内核还提供另一种被称为自旋锁的锁原语。自旋锁类似于信号量，它也用于防止对一个指定资源的并发访问，但自旋锁并不是休止锁——它主要用于保护临界区代码以防止它被运行在另一个处理器上的代码干扰，而不是用于限制用户对共享资源的访问。当一个函数试图获取自旋锁而失败时，它就将不停地旋转（非常快速地循环执行处理器的NOP操作），而此时在另一个处理器上执行的任务正持有该锁。

自旋锁不应被长时间持有。它是一种保护单个数据或结构以避免其在某个成员被操纵时被并发访问的好方法，但它同时也会导致系统中其他处理器进入一个紧密的循环并等待获取被其他函数持有的同一个锁。随着时间的推移自旋锁本身也在不断的发展，人们正在开展工作以解决其对实时应用程序性能的影响。自旋锁定义在头文件<linux/spinlock.h>中，它可以以如下方式声明：

```
struct spinlock my_lock;
spin_lock_init(&my_lock);
```

自旋锁通过spin_lock()函数获取：

```
spin_lock(&my_lock);
```

并通过spin_unlock()函数释放：

```
spin_unlock(&my_lock);
```

这取决于你是否还希望避免在获得自旋锁时受运行在其他处理器上的中断的影响，你可能会调用自旋锁函数的禁用中断的特殊变体：

```
unsigned long flags;
spin_lock_irqsave(&my_lock, flags);
spin_unlock_irqrestore(&my_lock, flags);
```

flags变量必须在函数内部声明并且不能被传递给其他任何函数。你不需要担心flags变量的内容，因为它是自旋锁实现在其内部使用来跟踪本地中断状态并用于确定是否真的在机器上启用或禁用硬件中断（因为它可能已被针对另一个锁的自旋锁函数调用禁用了，所以我们有必要在再次启用中断之前核实情况并非如此）。

3. plp_klock：为plp_kmem添加用户跟踪

前面的示例演示了一些非常有用的内存分配机制，但它有一个重大的缺陷。当有多个用户试图同时读写设备时，这个问题就暴露出来了。可以使用包括在模块源代码的在线版本中的reader.sh和writer.sh示例脚本来进行测试。你很快会看到许多不同的用户破坏一个共享资源的内容（如前面示例中的大型缓冲区）是多么的容易。

如果能将对模块缓冲区设备的访问限制为同一时间内只能有一个真正的用户可以访问，将是非常理想的。因为在没有设置某种形式互斥的情况下，如果有两个用户试图对同一个共享缓冲区进行读写操作将很快导致共享数据的损坏。这个问题本身并不是一个典型的“锁”问题，而是一个共享资源的问题，每当独立的个人用户共享设备时都会遇到这个问题。但是，要解决这个问题，你需要使用一些新的锁原语。在本节中，你将学习如何基于前面的plp_kmem建立一个plp_klock示例。

首先，你需要跟踪哪个系统用户已打开设备并正在使用它。你将在本书后面学到更多有关文件系统活动的内容，但现在你只需知道模块中的open函数被传递了一个struct file指针即可，可以通过它提取当前的用户ID。file->f_uid将包含系统用户的UID，root用户的UID总是0，而大多数普通系统的UID是从500开始分配的。设备的当前用户将被保存在一个全局变量中。

全局变量plp_klock_owner将包含打开设备的当前系统用户的UID，而plp_klock_count将用于统计设备被打开的次数。这里假定同一个系统用户多次打开设备是安全的——只需对plp_klock_open函数中的语义进行简单的调整就可以改变这一假定。但引入新的全局变量也带来一个问题——它们没有得到保护。

4. plp_klock的锁考虑

将在不同代码路径中同时使用的共享全局数据必须使用某种形式的锁来保护。在本例中，我们使用一个信号量来保护操纵plp_klock_owner和plp_klock_count的临界区代码。这个信号量是在进入plp_klock_open函数对试图打开设备的当前用户ID进行测试之前获取的（为方便阅读，我们在下面的代码中删除了调试代码）：

```
down(&plp_klock_sem);

if ((plp_klock_owner != file->f_uid) &&
    (plp_klock_count != 0)) {
    up(&plp_klock_sem);
    return -EBUSY;
}

plp_klock_count++;
plp_klock_owner=file->f_uid;

up(&plp_klock_sem);
```

注意，在错误处理中必须释放已获得的任何锁，因为一个错误的发生并不意味着内核就不应该继续正常操作并从错误中优雅地恢复。如果设备正被另一个用户使用，那么任何试图打开该设备的尝试都将导致错误-EBUSY的返回。应用程序知道这是一个临时错误，它是由于设备正忙于执行另一个操作而不能在那一时刻及时提供给程序造成的。

```
/*
 * plp_klock.c - Add locking to the previous kmem example.
 */

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/vmalloc.h>

#include <linux/fs.h>
#include <linux/major.h>
#include <linux/blkdev.h>
#include <linux/cdev.h>

#include <asm/uaccess.h>

#define PLP_KMEM_BUFSIZE (1024*1024) /* 1MB internal buffer */
```

```

/* global variables */

static char *plp_klock_buffer;

static struct class *plp_klock_class; /* pretend /sys/class */
static dev_t plp_klock_dev;          /* dynamically assigned char device */
static struct cdev *plp_klock_cdev;  /* dynamically allocated at runtime. */

static unsigned int plp_klock_owner; /* dynamically changing UID. */
static unsigned int plp_klock_count; /* number of current users. */

/* Locking */

static DECLARE_MUTEX(plp_klock_sem); /* protect device read/write action. */

/* function prototypes */

static int __init plp_klock_init(void);
static void __exit plp_klock_exit(void);

static int plp_klock_open(struct inode *inode, struct file *file);
static int plp_klock_release(struct inode *inode, struct file *file);
static ssize_t plp_klock_read(struct file *file, char __user *buf,
                           size_t count, loff_t *ppos);
static ssize_t plp_klock_write(struct file *file, const char __user *buf,
                           size_t count, loff_t *ppos);

/* file_operations */

static struct file_operations plp_klock_fops = {
    .read        = plp_klock_read,
    .write       = plp_klock_write,
    .open        = plp_klock_open,
    .release     = plp_klock_release,
    .owner       = THIS_MODULE,
};

/*
 * plp_klock_open: Open the ksem device
 */
static int plp_klock_open(struct inode *inode, struct file *file)
{
    down(&plp_klock_sem);

    if ((plp_klock_owner != file->f_uid) &&
        (plp_klock_count != 0)) {

#ifdef PLP_DEBUG
        printk(KERN_DEBUG "plp_klock: device is in use.\n");
#endif
        up(&plp_klock_sem);
    }
}

```

```
        return -EBUSY;
    }

    plp_klock_count++;
    plp_klock_owner=file->f_uid;

#ifndef PLP_DEBUG
    printk(KERN_DEBUG "plp_klock: opened device.\n");
#endif

    up(&plp_klock_sem);
    return 0;
}

/*
 * plp_klock_release: Close the ksem device.
 */
static int plp_klock_release(struct inode *inode, struct file *file)
{
    down(&plp_klock_sem);
    plp_klock_count--;
    up(&plp_klock_sem);

#ifndef PLP_DEBUG
    printk(KERN_DEBUG "plp_klock: device closed.\n");
#endif

    return 0;
}

/*
 * plp_klock_read: Read from the device.
 */
static ssize_t plp_klock_read(struct file *file, char __user *buf,
                             size_t count, loff_t *ppos)
{
    size_t bytes = count;
    loff_t fpos = *ppos;
    char *data;

    down(&plp_klock_sem);

    if (fpos >= PLP_KMEM_BUFSIZE) {
        up(&plp_klock_sem);
        return 0;
    }

    if (fpos+bytes >= PLP_KMEM_BUFSIZE)
        bytes = PLP_KMEM_BUFSIZE-fpos;

    if (0 == (data = kmalloc(bytes, GFP_KERNEL))) {
```

```

        up(&plp_klock_sem);
        return -ENOMEM;
    }

#endif PLP_DEBUG
printk(KERN_DEBUG "plp_klock: read %d bytes from device, offset %d.\n",
       bytes,(int)fpos);
#endif

memcpy(data,plp_klock_buffer+fpos,bytes);

if (copy_to_user((void __user *)buf, data, bytes)) {
    printk(KERN_ERR "plp_klock: cannot write data.\n");
    kfree(data);
    up(&plp_klock_sem);
    return -EFAULT;
}

*ppos = fpos+bytes;

kfree(data);
up(&plp_klock_sem);
return bytes;
}

/*
 * plp_klock_write: Write to the device.
 */
static ssize_t plp_klock_write(struct file *file, const char __user *buf,
                               size_t count, loff_t *ppos)
{
    size_t bytes = count;
    loff_t fpos = *ppos;
    char *data;

    down(&plp_klock_sem);

    if (fpos >= PLP_KMEM_BUFSIZE) {
        up(&plp_klock_sem);
        return -ENOSPC;
    }

    if (fpos+bytes >= PLP_KMEM_BUFSIZE)
        bytes = PLP_KMEM_BUFSIZE-fpos;

    if (0 == (data = kmalloc(bytes, GFP_KERNEL))) {
        up (&plp_klock_sem);
        return -ENOMEM;
    }

    if (copy_from_user((void *)data, (const void __user *)buf, bytes)) {
        printk(KERN_ERR "plp_klock: cannot read data.\n");
        kfree(data);
    }
}

```

```

        up (&plp_klock_sem);
        return -EFAULT;
    }

#ifdef PLP_DEBUG
    printk(KERN_DEBUG "plp_klock: write %d bytes to device, offset %d.\n",
           bytes,(int)fpos);
#endif

    memcpy(plp_klock_buffer+fpos,data,bytes);

    *ppos = fpos+bytes;

    kfree(data);
    up(&plp_klock_sem);
    return bytes;
}

/*
 * plp_klock_init: Load the kernel module into memory.
 */

static int __init plp_klock_init(void)
{
    printk(KERN_INFO "plp_klock: Allocating %d bytes of internal buffer.\n",
           PLP_KMEM_BUFSIZE);

    if (NULL == (plp_klock_buffer = vmalloc(PLP_KMEM_BUFSIZE))) {
        printk(KERN_ERR "plp_klock: cannot allocate memory!\n");
        goto error;
    }

    memset((void *)plp_klock_buffer, 0, PLP_KMEM_BUFSIZE);

    if (alloc_chrdev_region(&plp_klock_dev, 0, 1, "plp_klock"))
        goto error;

    if (0 == (plp_klock_cdev = cdev_alloc()))
        goto error;

    kobject_set_name(&plp_klock_cdev->kobj, "plp_klock_cdev");
    plp_klock_cdev->ops = &plp_klock_fops; /* file up fops */
    if (cdev_add(plp_klock_cdev, plp_klock_dev, 1)) {
        kobject_put(&plp_klock_cdev->kobj);
        unregister_chrdev_region(plp_klock_dev, 1);
        goto error;
    }

    plp_klock_class = class_create(THIS_MODULE, "plp_klock");
    if (IS_ERR(plp_klock_class)) {
        printk(KERN_ERR "plp_klock: Error creating class.\n");
        cdev_del(plp_klock_cdev);
        unregister_chrdev_region(plp_klock_dev, 1);
        goto error;
    }
}

```

```

}

class_device_create(plp_klock_class, NULL, plp_klock_dev, NULL, "plp_klock");

printf(KERN_INFO "plp_klock: loaded.\n");

return 0;

error:
printf(KERN_ERR "plp_klock: cannot register device.\n");
return 1;
}

/*
 * plp_klock_exit: Unload the kernel module from memory.
 */

static void __exit plp_klock_exit(void)
{
    class_device_destroy(plp_klock_class, plp_klock_dev);
    class_destroy(plp_klock_class);
    cdev_del(plp_klock_cdev);
    unregister_chrdev_region(plp_klock_dev, 1);

    vfree(plp_klock_buffer);

    printf(KERN_INFO "plp_klock: unloading.\n");
}

/* declare init/exit functions here */

module_init(plp_klock_init);
module_exit(plp_klock_exit);

/* define module meta data */

MODULE_AUTHOR("Jon Masters <jcm@jonmasters.org>");
MODULE_DESCRIPTION("Demonstrate kernel memory allocation");

MODULE_ALIAS("locking_example");
MODULE_LICENSE("GPL");
MODULE_VERSION("0:1.0");

```

9.3.7 推迟工作

有时候，内核代码希望将一些操作延迟，或将它们推迟到可以被更轻松处理的时候再执行。内核这么做可能出于许多不同的原因，但是，如果能够在系统不太忙的时候调度一些函数来执行，并能够抽出一些空闲时间来进行日常管理操作将是非常有用的。中断处理程序就是推迟工作的一个好示例，因为它们通常会暂时中止系统执行其他任务。为了避免不必要的大量开销，中断处理程序推迟执行它们的一些工作。

总的来说，在Linux内核中有两种类型的延迟或推迟工作的函数。第一类是那些在类似中断的上

下文中执行工作的函数，它们与所有用户进程完全分离并处于一个相当受限的环境中。它们虽然非常快（如果你需要非常频繁地或以非常低的开销执行大量工作，它们将非常有用），但是许多用户却喜欢在进程上下文中（更重量级）使用推迟工作的函数。

请记住，当在进程上下文中执行工作时，可以调用可能引起重新调度的函数而不用担心当调度实际发生时可能造成的系统锁死。例如，可以夺取一个信号量，或使用不同于GFP_ATOMIC的其他标记（GFP_ATOMIC显然可以在甚至最严格的资源受限情况下工作——这就是问题的终结）来调用kmalloc，或完成许多在过去不可能做的其他事情。请在可能的情况下尽量使自己的工作变得更容易。

1. 工作队列

调度推迟工作的最简单方法之一就是使用Linux 2.6内核中的工作队列接口。工作队列实质上是一个用于推迟执行单个函数的机制，它与其他许多支持函数（其中一些函数在这里进行了说明）一起定义在头文件<linux/workqueue.h>中。为了使用工作队列，必须先编写一个函数，它将在每次工作队列运行时被执行。它的原型如下：

```
void my_work(void *data);
```

然后，需要调用DECLARE_WORK宏来创建一个可调度的工作实体：

```
DECLARE_WORK(my_work_wq, my_work, data);
```

这个宏定义一个名称为my_work_wq的workqueue_struct新结构，并将它设置为每当被调用时就执行函数my_work。第三个参数data是一个可选的void指针参数，每次当my_work函数需要使用特定数据完成工作时，可以使用该参数指向该数据。只有调用下面的某个schedule_work函数时，这个工作才真正运行。

```
int schedule_work(my_work_wq);
```

当系统稍微空闲时，my_work函数将被立刻执行。如果想对内核考虑运行你的工作函数之前的等待时间进行限制，可以使用schedule_delayed_work函数来增加一个延迟：

```
int schedule_delayed_work(my_work_wq, delay)
```

延迟应该以jiffies值进行指定。例如，要延迟10秒，可以使用HZ宏来自动获得延迟的jiffies值，即 $10 * \text{HZ}$ 。

2. 内核线程

虽然工作队列很好，但它们通常在单内核线程的上下文中运行，而该内核线程还被其他的作为调度实体的工作元件所使用。可以通过定义自己的内核线程来更好地完成工作，该线程只提供给自己的工作使用，其他工作都不能使用。这样一来，你可以拥有一个自己的内核线程，它不时地醒来以执行一些数据处理，然后睡眠，直到再次需要它。要设置一个内核线程，你可以使用定义在头文件<linux/kthread.h>中的kthread API函数，它们使内核线程更易于处理。

- kthread_create

内核线程可以通过辅助函数轻松地创建。该函数使用将被新创建的线程运行的普通函数的函数名称作为其参数。在下面的示例中，新创建的内核线程将用于运行plp_kwork_kthread函数，它有一个NULL参数并被称为plp_work_kthread。kthread_create函数返回一个新的task_struct结构：

```
(struct task_struct *)plp_kwork_ts = kthread_create(plp_kwork_thread, NULL,
"plp_work_kthread");
```

你还可以调用kthread_run变体，它将创建内核线程并启动它：

```
(struct task_struct *)plp_kwork_ts = kthread_run(plp_kwork_thread, NULL,
"plp_work_kthread");
```

要通知一个内核线程停止运行，调用kthread_stop函数并向它传递一个内核线程的任务结构指针：

```
kthread_stop(plp_kwork_ts);
```

它将一直阻塞直到内核线程结束运行。为了加快这一过程，线程函数应该不时地检查kthread_should_stop()函数返回的值。下面是一个简单的内核线程函数的示例，它每10秒钟苏醒一次，处理数据，然后继续睡眠10秒钟——除非它被一个信号唤醒，例如由模块的另一部分调用kthread_stop()所发送的信号：

```
static int plp_kwork_kthread(void *data)
{
    printk(KERN_INFO "plp_kwork: kthread starting.\n");

    for(;;) {
        schedule_timeout_interruptible(10*HZ);

        if (!kthread_should_stop()) {

            /* If nobody has the device open, reset it after
             * a period of time.
            */

            down(&plp_kwork_sem);
            if (!plp_kwork_count) {
                plp_kwork_owner = 0;
                memset((void *)plp_kwork_buffer, 0,
                       PLP_KMEM_BUFSIZE);
            }
            up(&plp_kwork_sem);

        } else {
            printk(KERN_INFO "plp_kwork: kthread stopping.\n");
            return 0;
        }
    }
}
```

这个示例函数检查是否有人正在使用由前面的kmem示例定义的内存缓冲区。如果虚拟设备没有被使用，那么内存将使用memset重置。

3. 内核线程示例

下面的示例对前面的kmem示例和klock示例进行了修改以支持内核线程，它将一直监视设备，如果设备没有被使用，它将重置内存缓冲区：

```
/*
 * plp_kwork.c - Example of scheduling deferred work via kernel threads.
 */
```

```

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/vmalloc.h>

#include <linux/fs.h>
#include <linux/major.h>
#include <linux/blkdev.h>
#include <linux/cdev.h>

#include <linux/kthread.h>

#include <asm/uaccess.h>
#define PLP_KMEM_BUFSIZE (1024*1024) /* 1MB internal buffer */

/* global variables */

static char *plp_kwork_buffer;

static struct class *plp_kwork_class; /* pretend /sys/class */
static dev_t plp_kwork_dev; /* dynamically assigned char device */
static struct cdev *plp_kwork_cdev; /* dynamically allocated at runtime. */

static unsigned int plp_kwork_owner; /* dynamically changing UID. */
static unsigned int plp_kwork_count; /* number of current users. */

static struct task_struct *plp_kwork_ts; /* task struct for kthread. */

/* Locking */

static DECLARE_MUTEX(plp_kwork_sem); /* protect device read/write action. */

/* function prototypes */

static int __init plp_kwork_init(void);
static void __exit plp_kwork_exit(void);

static int plp_kwork_open(struct inode *inode, struct file *file);
static int plp_kwork_release(struct inode *inode, struct file *file);
static ssize_t plp_kwork_read(struct file *file, char __user *buf,
                           size_t count, loff_t *ppos);
static ssize_t plp_kwork_write(struct file *file, const char __user *buf,
                           size_t count, loff_t *ppos);

static int plp_kwork_kthread(void *data);

/* file_operations */

static struct file_operations plp_kwork_fops = {
    .read        = plp_kwork_read,
    .write       = plp_kwork_write,
    .open        = plp_kwork_open,
    .release     = plp_kwork_release,
    .owner       = THIS_MODULE,
};

```

```
};

/*
 * plp_kwork_open: Open the ksem device
 */
static int plp_kwork_open(struct inode *inode, struct file *file)
{
    down(&plp_kwork_sem);

    if ((plp_kwork_owner != file->f_uid) &&
        (plp_kwork_count != 0)) {
        #ifdef PLP_DEBUG
            printk(KERN_DEBUG "plp_kwork: device is in use.\n");
        #endif

        up(&plp_kwork_sem);
        return -EBUSY;
    }

    plp_kwork_count++;
    plp_kwork_owner=file->f_uid;

    #ifdef PLP_DEBUG
        printk(KERN_DEBUG "plp_kwork: opened device.\n");
    #endif

    up(&plp_kwork_sem);
    return 0;
}

/*
 * plp_kwork_release: Close the ksem device.
 */
static int plp_kwork_release(struct inode *inode, struct file *file)
{
    down(&plp_kwork_sem);

    plp_kwork_count--;
    up(&plp_kwork_sem);

    #ifdef PLP_DEBUG
        printk(KERN_DEBUG "plp_kwork: device closed.\n");
    #endif

    return 0;
}

/*
 * plp_kwork_read: Read from the device.
 */

```

```

static ssize_t plp_kwork_read(struct file *file, char __user *buf,
                             size_t count, loff_t *ppos)
{
    size_t bytes = count;
    loff_t fpos = *ppos;
    char *data;

    down(&plp_kwork_sem);

    if (fpos >= PLP_KMEM_BUFSIZE) {
        up(&plp_kwork_sem);
        return 0;
    }

    if (fpos+bytes >= PLP_KMEM_BUFSIZE)
        bytes = PLP_KMEM_BUFSIZE-fpos;

    if (0 == (data = kmalloc(bytes, GFP_KERNEL))) {
        up(&plp_kwork_sem);
        return -ENOMEM;
    }

#ifdef PLP_DEBUG
    printk(KERN_DEBUG "plp_kwork: read %d bytes from device, offset %d.\n",
           bytes,(int)fpos);
#endif

    memcpy(data,plp_kwork_buffer+fpos,bytes);

    if (copy_to_user((void __user *)buf, data, bytes)) {
        printk(KERN_ERR "plp_kwork: cannot write data.\n");
        kfree(data);
        up(&plp_kwork_sem);
        return -EFAULT;
    }

    *ppos = fpos+bytes;

    kfree(data);
    up(&plp_kwork_sem);
    return bytes;
}

/*
 * plp_kwork_write: Write to the device.
 */

static ssize_t plp_kwork_write(struct file *file, const char __user *buf,
                             size_t count, loff_t *ppos)
{
    size_t bytes = count;
    loff_t fpos = *ppos;
    char *data;

```

```

down(&plp_kwork_sem);

if (fpos >= PLP_KMEM_BUFSIZE) {
    up(&plp_kwork_sem);
    return -ENOSPC;
}

if (fpos+bytes >= PLP_KMEM_BUFSIZE)
    bytes = PLP_KMEM_BUFSIZE-fpos;

if (0 == (data = kmalloc(bytes, GFP_KERNEL))) {
    up(&plp_kwork_sem);
    return -ENOMEM;
}

if (copy_from_user((void *)data, (const void __user *)buf, bytes)) {
    printk(KERN_ERR "plp_kwork: cannot read data.\n");
    kfree(data);
    up(&plp_kwork_sem);
    return -EFAULT;
}

#ifndef PLP_DEBUG
printf(KERN_DEBUG "plp_kwork: write %d bytes to device, offset %d.\n",
       bytes, (int)fpos);
#endif

memcpy(plp_kwork_buffer+fpos,data,bytes);

*ppos = fpos+bytes;

kfree(data);
up(&plp_kwork_sem);
return bytes;
}

/*
 * plp_kwork_init: Load the kernel module into memory
 */
static int __init plp_kwork_init(void)
{
    printk(KERN_INFO "plp_kwork: Allocating %d bytes of internal buffer.\n",
           PLP_KMEM_BUFSIZE);

    if (NULL == (plp_kwork_buffer = vmalloc(PLP_KMEM_BUFSIZE))) {
        printk(KERN_ERR "plp_kwork: cannot allocate memory!\n");
        goto error;
    }

    memset((void *)plp_kwork_buffer, 0, PLP_KMEM_BUFSIZE);

    if (alloc_chrdev_region(&plp_kwork_dev, 0, 1, "plp_kwork"))
        goto error;
}

```

```

if (0 == (plp_kwork_cdev = cdev_alloc()))
    goto error;

kobject_set_name(&plp_kwork_cdev->kobj, "plp_kwork_cdev");
plp_kwork_cdev->ops = &plp_kwork_fops; /* file up fops */
if (cdev_add(plp_kwork_cdev, plp_kwork_dev, 1)) {
    kobject_put(&plp_kwork_cdev->kobj);
    unregister_chrdev_region(plp_kwork_dev, 1);
    goto error;
}

plp_kwork_class = class_create(THIS_MODULE, "plp_kwork");
if (IS_ERR(plp_kwork_class)) {
    printk(KERN_ERR "plp_kwork: Error creating class.\n");
    cdev_del(plp_kwork_cdev);
    unregister_chrdev_region(plp_kwork_dev, 1);
    goto error;
}
class_device_create(plp_kwork_class, NULL, plp_kwork_dev, NULL, "plp_kwork");

plp_kwork_ts = kthread_run(plp_kwork_kthread, NULL, "plp_kwork_kthread");
if (IS_ERR(plp_kwork_ts)) {
    printk(KERN_ERR "plp_kwork: can't start kthread.\n");
    cdev_del(plp_kwork_cdev);
    unregister_chrdev_region(plp_kwork_dev, 1);
    goto error;
}

printk(KERN_INFO "plp_kwork: loaded.\n");

return 0;

error:
printk(KERN_ERR "plp_kwork: cannot register device.\n");
return 1;
}

/*
 * plp_kwork_exit: Unload the kernel module from memory
 */
static void __exit plp_kwork_exit(void)
{
    kthread_stop(plp_kwork_ts);

    class_device_destroy(plp_kwork_class, plp_kwork_dev);
    class_destroy(plp_kwork_class);
    cdev_del(plp_kwork_cdev);
    unregister_chrdev_region(plp_kwork_dev, 1);

    vfree(plp_kwork_buffer);
}

```

```

    printk(KERN_INFO "plp_kwork: unloading.\n");
}

static int plp_kwork_kthread(void *data)
{
    . . .

    printk(KERN_INFO "plp_kwork: kthread starting.\n");

    for(;;) {
        schedule_timeout_interruptible(10*HZ);

        if (!kthread_should_stop()) {

            /* If nobody has the device open, reset it after
             * a period of time.
            */

            down(&plp_kwork_sem);
            if (!plp_kwork_count) {
                plp_kwork_owner = 0;
                memset((void *)plp_kwork_buffer, 0,
                       PLP_KMEM_BUFSIZE);
            }
            up(&plp_kwork_sem);

        } else {
            printk(KERN_INFO "plp_kwork: kthread stopping.\n");
            return 0;
        }
    }
    return 0;
}

/* declare init/exit functions here */

module_init(plp_kwork_init);
module_exit(plp_kwork_exit);

/* define module meta data */

MODULE_AUTHOR("Jon Masters <jcm@jonmasters.org>");
MODULE_DESCRIPTION("Demonstrate kernel memory allocation");

MODULE_ALIAS("kthread_example");
MODULE_LICENSE("GPL");
MODULE_VERSION("0:1.0");

```

9.3.8 进一步阅读

前几节对Linux内核中的不同API作了简要的介绍。讲解了内核中的某些部分是如何对内存分配、锁原语和推迟工作处理提供支持的。当然，Linux内核中包含的API比你在本章中看到的要多得多，但

我们希望本章除能使你体验内核操作的感觉并受到鼓舞，你还可以通过参考与这一主题相关的其他图书和在线资源以学习更多这方面的内容。

本章没有涉及的一个问题是开发特定硬件设备的驱动程序，因为我们难以判断你在阅读本书时有什么样的硬件，而且我们也不可能在一章的内容中涵盖所有的方案。你可以在诸如《Linux设备驱动程序》这样的书中找到特定硬件的示例，Linux每周新闻网站www.lwn.net提供了该书的在线版本，你还可以看看该网站对内核的评论。

此外，一些在线教程利用虚拟机技术（如由gemu和UML提供的技术）来合成虚拟设备，以使得你可以编写自己的驱动程序。你可以在与本书相关的网站上找到一些示例，但不管怎么做，高质量驱动程序示例的最好来源还是Linux内核自身。请查看Linux内核源代码目录下的“drivers”子目录，你将找到许多简单的示例模块。

9.4 分发 Linux 内核模块

有许多不同的方法可以将自己的模块提供给第三方，最明显的方式就是直接将补丁发送给Linux内核邮件列表，但这不可能总是切实可行的，所以本节将讨论几种与Linux内核发布驱动程序相独立的可能方法——至少在内核将模块内置之前。

9.4.1 进入上游 Linux 内核

任何人编写Linux内核模块的最终目标都是希望它能进入上游Linux内核。这意味着模块成为主流Linux内核的一部分并由你维护，同时还可以得到其他任何希望提供修复、更新或提出改善模块建议的人的帮助。近年来，内核树之外的模块一直在不断更新以反映Linux内核开发的变化步伐。当上游内核修改了它的API后，模块必须进行相应的修改，你必须忍受上游内核开发人员的决定。

随着时间的推移，内核输出给可装载模块的接口已越来越少。内核开发人员很少关心是否完全删除一个只由内核树之外的已知模块使用的API。例如，在写作本书时，人们一直争论是否要将LSM（Linux安全模块）构架从内核中删除，因为它只有一个内核中的用户（而在给其他已知的内核树之外的用户发送了多次请求之后，他们并没有将代码发送给上游内核）。这里的底线是，只要有可能，最安全的方式就是使你的代码进入上游内核——即使你只是该代码的唯一使用者！

使你的模块进入上游内核决不是一件容易的事情，你必须首先通过“Christoph”^①的测试。这意味着你首先必须将代码补丁发送到Linux内核邮件列表（LKML）并等待其他内核开发者的意见。某些开发者在各种议题上被公认为固执己见，他们很可能会批评你的代码，此时你应该谦逊一些并认真听取他们的意见，因为他们的意见往往是正确的。这些意见并不是针对你个人的——他们尽力确保Linux内核拥有着最好的质量，所以他们对内核代码非常的苛求。

9.4.2 发行源代码

对那些还没有将他们的模块内置到Linux内核中的开发人员来说，另一个好的解决方案是通过一个网站提供模块的源代码并允许第三方使用我们在前面几章中讨论过的指令和编译过程来自己编译驱动程序。请记住，你需要确保你的模块在用户所用的各种内核中都可以使用。这意味着你需要针对一些不太常用的厂商内核验证你的模块的可用性。

^① Christoph指Christoph Hellwig，德国人，Linux内核核心编程人物。——译者注

许多较有经验的用户乐意遵循指示来编译模块，但对于那些新手来说就比较困难，所以你可能会有兴趣了解DKMS项目。DKMS支持在目标环境中自动编译驱动程序，一个主要的计算机厂商已在这方面开展了许多有趣的工作。有关DKMS的更多信息请访问网站<http://linux.dell.com/dkms>。

9.4.3 发行预编译模块

越来越多的Linux发行版为预包装驱动程序提供基础架构。它们可能没有正式支持内核树之外的驱动程序，但如果你还不能使你的内核模块进入上游内核，那么你可以按照你的Linux发行版提供的软件包装指南来包装你的模块。一些著名的Linux公司甚至联合起来开发支持在软件包级别验证模块符号版本的工具——允许你同时在多个兼容内核中使用同一个模块，这会有助于减少预包装驱动程序需要的更新次数。

欲了解更多有关包装预编译内核模块的信息，请联系你的厂商。

9.5 本章总结

在本章中，你学习了Linux内核模块（LKM），了解了它们是如何通过大量的函数和宏构建起来的。你学会了如何使用GNU工具链编译LKM并将它们连接进目标文件以扩展Linux内核的功能。你还了解了一些提供给大多数可装载内核模块的标准Linux内核API，并通过一个基于虚拟内存的缓冲区设备看到了应用这些API的一些示例。

虽然本章不可能涵盖编写模块的所有复杂性，但你现在应对编写模块所涉及的概念比较熟悉，并具备继续学习Linux内核（特别是模块开发）的能力。在后面的章节中，你将学习如何与Linux内核沟通，诸如udev和D-BUS这样的工具如何能够接收消息并显示发生在系统中的各种事件，内核在其中起的又是什么作用。

第 10 章

调试

在本书中，你将学到各种可用于Linux系统的技术。你将研究本书提供的各种示例并（希望）自行测试其中的一些示例。但你很可能会在学习的过程中至少遇到一次或两次程序错误——这是开发任何计算机软件所必然面临的一个问题。虽然Linux是由来自世界各地的编程高手共同开发的，但没有软件是完美无暇的。

在本章中，你将学习一些可有助于你为Linux系统编写代码的调试技术。本章分为三个部分：首先，你将学习gdb命令行GNU调试工具以及它的一些强大指令集的用法；其次，你将通过使用诸如ddd和Eclipse这样的工具来学习基于gdb的更高层的图形化调试方法；最后，我们将使用UML（用户模式Linux）内核作为一个示例来深入研究内核调试。

在学习完本章后，你将对Linux程序员可以使用的各种调试工具有一个很好的了解。你将掌握这些工具的基本用法并具备通过自己的实践继续学习的能力。请牢记调试本质是一项复杂而艰巨的任务，它需要你投入很多年的时间才能完全掌握。千万不要指望仅仅阅读完本章你就会成为一位调试专家——你需要将这些技术付诸实践。

10.1 调试概述

几乎可以肯定你在调试应用程序上有过一些经验或经历。你既可能是一位程序员，也可能是一位经历过程不以预期方式运行而遭受挫折的用户。软件缺陷出现的原因各种各样（事实上，对它的研究涉及一整个学科），但不管原因是什么，其结果往往是导致用户产生负面体验^①，而且，不管哪种情况都需要某种形式的软件补丁。我们首先面临的困难（通常）是准确找出产生错误的原因。

常见类型的错误有很多种，如琐碎的“差一错误”(off by one error)，无论你使用哪种操作系统，它们都有可能发生并将导致应用程序产生某些不当行为。但还有很多错误是特定于Linux和UNIX系统的，这些错误你可能不太熟悉。我们将在这里对它们进行简要的介绍，但是，如果你想对它们有一个真正全面的了解，需要购买一本专门介绍这方面内容的优秀的学术图书。在任何情况下，当你为Linux编写代码时，请你牢记来自UNIX合作者Brian W. Kernighan的名言：

“调试代码的难度是首次编写这些代码的两倍，因此，如果你在编写代码的时候就已经发挥了全部的聪明才智，那么按照常理，你将无法凭借自己的智慧去调试这些代码。”

内存管理

Linux和类UNIX系统中最常见（而且最烦人）的错误发生区域之一在每个应用程序的内存处理中。错误的内存分配、没有释放内存、使用已释放的内存以及其他许多典型的错误会在各种平台的程序中

^① 例如ATM自动取款机由于软件故障而送给你一个“馅饼”。

频繁地发生并需要我们花费大量的时间来发现它们。幸运的是，在普通应用程序中，错误的内存分配并不会对整个系统造成影响，这归功于Linux内核对虚拟内存环境的处理。

当为Linux编写软件时，将遇到的两种典型的内存相关错误是：

- **段错误：**每当程序试图访问超出其合法数据存储区域之外的内存时，这个丑陋的错误就将发生。Linux内核之所以能够检测到这些非法访问是因为其虚拟内存子系统在发生这类访问时负责处理底层（坏）页故障^①。应用程序将被强行终止，并将根据系统的配置来确定是否要创建一个包含程序状态的核心转储文件。
- **内存泄漏：**这个错误在任何编程环境中都普遍存在。但是，因为内核为应用程序提供了一个虚拟内存环境，所以普通程序中的内存泄漏不会导致整个系统崩溃。一旦应用程序崩溃或终止，它泄漏的内存就可以再次提供给其他应用程序。系统管理员通常会设置一些资源限制以限制每个程序可以使用的内存量。

前一种错误可能是你会遇到的最烦人的错误了，但它总比允许这样的错误内存使用要好。大多数导致程序崩溃的内存相关问题同时也会留下足够的线索，你所需要的只是花费一点时间在gdb中发现这些线索。但有时候，一些细微的内存误用可能未被发现，此时就是诸如Valgrind（稍后讨论）这样的工具发挥作用的时候了，它将监控运行时程序的执行。

10.2 基本调试工具

Linux程序员一直以来都依赖于一小组基本调试工具集，许多更复杂的调试工具都是基于这些基本工具建立的。GNU调试器（gdb）是迄今为止用于跟踪和调试应用程序的最流行的调试工具，但如果你想做比仅仅插入几个断点并使用一些测试数据以监视应用程序运行状况更多的事情，它并不是你唯一的选择。例如，近年来出现的一些系统分析工具（如systemtap和frysk）就将为应用程序的性能增强提供帮助。

10.2.1 GNU 调试器

GDB是如今最广为人知的著名的自由和开放源码软件之一。它被大量GNU软件项目以及众多与GNU没有关联但却希望能有一个高质量调试器的第三方软件所使用。事实上，许多第三方工具合并gdb并将其作为它们的调试功能的基础，即便它们在gdb之上建立了各级图形化抽象。你很可能已遇到过GDB——但可能根本没有意识到这一点。

GDB建立在任何调试器都有两个组成部分这一基本概念之上。首先，GDB的底层处理单独进程或线程的启动和关闭、跟踪代码执行以及在运行代码中插入和删除断点。GDB支持大量不同的平台和机制以在各种架构上实现这些（看似简单的）操作。其具体的功能可能会受底层硬件功能的影响而偶尔有所变动。

GDB甚至支持连接到Abatron BDI2000等硬件调试设备，它是一个专用的CPU级调试器，它通过停止和启动CPU自身可以用于远程调试运行在另一个系统上的Linux内核。与采用其他一些拙劣的调试器设计技术重新实现一个调试器相比，GDB更受欢迎。如果你是一个嵌入式开发者，你可能会遇到这种用法。

^① 实际上，当发生页故障时，内核将试图提供不在页中（当时也不在交换空间中）的数据。因为还有其他一些没有在这里介绍的原因和复杂性也会导致页故障的产生——但生活就是这样。

基于调试应用程序所必需的底层功能之上的是对可被程序员以有效方式使用的更高层接口的需求。GDB提供了一个带有一套标准命令的接口，而不管它在何种硬件之上运行。这可能是它为什么这么流行的原因之一——你只需要学习一次核心命令即可。

1. 为GDB的使用编译代码

GNU调试器以一种看起来好像很简单的命令行工具gdb的形式呈现。它期望以一个程序名作为它的参数或在执行之前被要求装载一个程序。然后它将启动调试器环境并等待指令。在你明确地告诉gdb在调试器环境中开始程序的执行之前，程序将不会开始执行。

作为示例，我们创建一个简单的Hello World程序用于gdb的使用：

```
/*
 * hello.c - test program.
 */

#include <stdio.h>

void print_hello()
{
    printf("Hello, World!\n");
}

int main(int argc, char **argv)
{
    print_hello;
    return 0;
}
```

这个示例代码使用了一个函数来显示欢迎信息，因为这将导致主程序至少进行一次函数调用。当使用调试器时，你将注意到一个新的堆栈帧被创建并可以看到函数调用的效果，这要比仅仅使用一个标准的测试程序看到更多的东西——关键是，在示例中使用函数调用是有原因的，因为你将在后面对GDB的功能进行测试时利用到这一点。

你可以使用GCC来编译这个示例程序：

```
$ gcc -o hello -g hello.c
```

请注意在GCC命令行中指定的-g标记。它告诉GCC必须在最终产生的可执行文件中添加额外的调试符号和源代码级信息，与使用普通二进制文件相比，GDB可以为前者提供更详细的信息。GDB的确需要这些额外信息（它们通常从Linux二进制文件中剥离以节省空间）使其能够提供源文件行号和其他相关信息。如果你在尝试编译下面的示例时不在最终可执行文件中加入调试信息，你将很快看到它们在输出上的差异。

除了-g标记以外，你可能还想在一些平台（特别是那些因性能原因而对机器代码指令调度进行优化的平台）上提供其他的GCC命令标记。带有精简指令集的现代非x86处理器就属于这一情况，因为现在的趋势是在编译器中进行优化而不是对机器指令集进行复杂化。这意味着在实践中关闭指令的重新调度将导致有意义的GDB输出。请参考前面第2章中对GCC命令行标记的说明。

2. 启动GDB

一旦应用程序使用正确的命令标记进行了编译，就可以通过调用gdb命令将hello程序装载到GDB中，如下所示：

```
$ gdb ./hello
GNU gdb Red Hat Linux (6.3.0.0-1.122rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "ppc-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb)
```

GDB将hello ELF二进制文件装载到内存中并为程序的运行设置一个环境，然后它将同时在程序的二进制文件和连接进程序的任何库文件中查找有用的符号。run命令用于正常运行程序：

```
(gdb) run
Starting program: /home/jcm/PLP/src/debugging/hello
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x100000
Hello, World!

Program exited normally.
```

一切都很好，但仅仅可以从头到尾运行一个程序并不是特别有用。这时就是GDB的强大命令集发挥作用的时候了。其中一部分命令我们将在下面单独讨论，你还可以通过GDB内置的帮助系统以及现有的系统文档（和GDB专著）找到完整的命令集以及它们的使用说明。GDB的帮助系统将命令集分成如下几类：

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

例如，如果你想获得控制程序执行的命令列表，可以在提示符中输入help running。请花一些时间浏览GDB中可以使用的命令集，这样你就不用在今后需要使用的时候频繁地查找命令名称了。

- 设置断点

仅仅用GDB来运行程序并不是特别令人兴奋，除非你还可以在程序运行时控制程序的执行，此时就是break命令发挥作用的时候了。通过在运行时将一个断点插入程序，你可以在程序指令到达一个特定点时让GDB停止程序的执行。在程序执行开始时（即在main函数的入口）设置这样的一个断点是很常见的（甚至是惯例）：

```
(gdb) break main
Breakpoint 1 at 0x1000047c: file hello.c, line 17.
Now that a breakpoint has been inserted, program execution will cease immediately
upon entry into the main function. You can test this by re-issuing the run command
to GDB:
```

你还可以使用break命令的简写形式：

```
(gdb) b main
```

它将达到同样的效果。

然后使用run命令运行程序：

```
(gdb) run
Starting program: /home/jcm/PLP/src/debugging/hello
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x100000

Breakpoint 1, main (argc=1, argv=0x7fed6874) at hello.c:17
10         printf("Hello, World!\n");
```

请注意GDB是如何在指定的断点处暂停程序的执行的，然后将由你来决定是继续运行还是执行另一个操作。你可以使用step和next命令继续向前执行程序。step命令将继续执行程序直至到达程序源代码的新的一行，而stepi命令只向前执行一条机器指令（对处理器来说，源代码中的一行语句可能意味着许多条机器代码指令）。

next命令和nexti变体的行为与上面的两个命令类似，但它们在遇到函数调用时不会进入函数内部——这样你在调试代码时就不需要担心函数调用了。这个命令非常有用，因为C函数库很可能并不具备和你正在调试的应用程序一样级别的调试信息。因此，跟踪进入标准C库函数可能并没有什么意义，即使你有兴趣想了解这些函数实现的更多信息。

请注意，从技术上来说，在main函数的入口插入一个断点并不会在程序最开始处停止程序的执行，因为运行在Linux系统中的所有基于C语言的程序都会在运行时利用GNU C函数库来为调用main函数做好安排。因此，在到达断点时，已有许多额外的库函数执行了大量设置工作。

3. 显示数据

可以在GDB中使用print命令查询程序中存储的数据，例如，可以显示通过run命令传递给示例程序的可选参数。传递给run命令的参数将作为argv参数列表传递给被调用程序。下面显示了当我们使用添加的参数调用一个简单程序时，GDB产生的输出：

```
(gdb) break main
Breakpoint 1 at 0x1000047c: file hello.c, line 17.
(gdb) run foo baz
Starting program: /home/jcm/PLP/src/debugging/hello foo baz
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x100000

Breakpoint 1, main (argc=3, argv=0x7f8ab864) at hello.c:17
10          printf("Hello, World!\n");
(gdb) print argv[0]
$1 = 0x7f8ab9d8 "/home/jcm/PLP/src/debugging/hello"
(gdb) print argv[1]
$2 = 0x7f8ab9fa "foo"   .
(gdb) print argv[2]
$3 = 0x7f8ab9fe "baz"
(gdb) print argv[3]
$4 = 0x0
```

你可以看到程序的参数列表，在`argv`列表的最后是NULL字符。示例程序是否使用这个参数列表并没有关系——它一直存在并且你可以调用GDB的`print`命令来返回它的值，如上面的输出所示。

4. Backtrace

GDB提供了许多有用的堆栈帧管理命令，并且包括了几个专门用于查看程序是如何到达它目前所在的位置的命令。其中最有用的一个命令是`backtrace`（简写为`bt`），它可以用于查看程序运行到当前位置之前所有的堆栈帧情况。下面是一个由`backtrace`命令提供信息的示例：

```
(gdb) break main
Breakpoint 1 at 0x100004b4: file hello.c, line 17.
(gdb) run
Starting program: /home/jcm/PLP/src/debugging/hello
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x100000

Breakpoint 1, main (argc=1, argv=0x7ff96874) at hello.c:17
17          print_hello();
(gdb) step
print_hello () at hello.c:10
10          printf("Hello, World!\n");
(gdb) bt
#0  print_hello () at hello.c:10
#1  0x100004b8 in main (argc=1, argv=0x7ff96874) at hello.c:17
```

在上面的GDB会话中，可以明显地看到程序进入`print_hello`函数并如何导致由`bt(backtrace)`命令列出的两个堆栈帧的产生。第一个（也是最里层的一个）是当前堆栈帧，它由`print_hello`函数使用，而外层堆栈帧被全局函数`main`用来保存它在调用`print_hello`之前的局部变量。

5. 一个有错误的示例

到目前为止，你已看到了一些在GDB环境中使用命令的示例，但你还没有试图调试一个包含实际错误的真正程序！下面的示例将向你显示如何使用GDB来完成日常的调试任务——定位一个错误的指针引用、在程序崩溃后进行回溯（`backtrace`）以及其他相关的动作。

你可以通过下面的简单的示例程序开始GDB的调试之旅。在源文件`buggy.c`中的代码定义了一个

`linked_list`结构并使用头优先链表插入算法分配了10个元素。但不幸的是，链表中的`data`元素被分配到未分配的内存中（没有使用预分配的内存调用`strncpy`）。更糟的是，程序没有释放已分配的内存——我们将在本节后面介绍更多有关检测内存泄漏的内容。

下面是一个简单的有错误的程序：

```
/*
 * buggy.c - A buggy program.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct linked_list {
    struct linked_list *next;
    char *data;
};

int main(int argc, char **argv)
{
    struct linked_list *head = NULL;
    struct linked_list *tmp = NULL;
    char *test_string = "some data";
    int i = 0;

    for (i=0;i<10;i++) {
        tmp = malloc(sizeof(*tmp));
        strncpy(tmp->data,test_string,strlen(test_string));
        tmp->next = head;
        head = tmp;
    }

    return 0;
}
```

你可以使用GCC编译并运行这个程序，如下所示：

```
$ gcc -o buggy -g buggy.c
$ ./buggy
Segmentation fault
```

正如你看到的，程序因一个错误而不幸崩溃了，这意味着它试图访问已分配数据存储区之外的内存。通常情况下，出现这种情况是因为一个指向非法位置而不是合法内存位置的指针被解引用（通常没有很简单的方法来知道存储在某个内存位置中的数字是否是一个真正的指针，知道时可能为时已晚）。在本例中，出现错误的原因是因为代码试图将数据写入一个内存还没有被分配的字符串。

你可以将这个有错误的程序装载到GDB中，然后再次运行它：

```
$ gdb ./buggy
GNU gdb Red Hat Linux (6.3.0.0-1.122rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "ppc-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb) run
Starting program: /home/jcm/PLP/src/debugging/buggy
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x100000

Program received signal SIGSEGV, Segmentation fault.
0x0f65a09c in strcpy () from /lib/libc.so.6
```

这次，程序的崩溃被GDB检测到了，并且它已准备好对程序进行调试。你现在可以使用bt（backtrace）命令找出当程序由于一个错误的内存解引用而接收到一个致命信号时程序到底做了些什么。简写的bt命令可以加快你的输入：

```
(gdb) bt
#0 0x0f65a09c in strcpy () from /lib/libc.so.6
#1 0x10000534 in main (argc=1, argv=0x7f9c5874) at buggy.c:26
```

根据上面的backtrace命令的输出，我们可以看出buggy源程序的第26行代码导致对C库函数strcpy的调用，此时程序崩溃了。很明显，我们需要查看源程序中的那一行代码以找出问题所在：

```
strcpy(tmp->data,test_string,strlen(test_string));
```

这行代码将test_string拷贝到链表元素tmp的数据成员中，但data成员事先并没有被初始化，所以一个随机的内存位置被用来存放该字符串。这类的内存误用将很快导致一个预料中的而且可能不可避免的程序崩溃。我们所需要做的只是在strcpy之前进行一次简单的malloc调用（或使用一个调用malloc的字符串拷贝函数）。在本例的情况下，你只需设置data成员指向静态字符串test_string就可以避免任何字符串拷贝操作。

要解决这个错误，你可以将示例源代码中的strcpy调用替换为：

```
tmp->data = test_string;
```

这个程序现在可以正常编译和运行了：

```
$ gcc -o buggy -g buggy.c
$ ./buggy
```

6. 调试核心转储文件

传统上，UNIX和类UNIX系统在程序崩溃时会进行核心转储或提供一个程序状态的二进制输出。如今，很多Linux发行版关闭了普通用户的core转储文件创建功能以节省磁盘空间（让这些普通用户可能一无所知的核心转储文件散乱的分布在磁盘中既浪费空间也让用户感到不安）。正常情况下，一个特定的Linux发行版将使用ulimit（用户限制）命令来控制核心转储文件的创建。

你可以使用ulimit命令来查看当前用户限制:

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
max nice                (-e) 0
file size               (blocks, -f) unlimited
pending signals          (-i) 4096
max locked memory        (kbytes, -l) 32
max memory size          (kbytes, -m) unlimited
open files              (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
max rt priority          (-r) 0
stack size               (kbytes, -s) 10240
cpu time                 (seconds, -t) unlimited
max user processes        (-u) 4096
virtual memory            (kbytes, -v) unlimited
file locks               (-x) unlimited
```

正如你看到的那样，核心转储文件的大小被设置为0（禁用）。可以通过给ulimit命令的-c可选标记传递一个新值来重置这个值。它应该被设置为以磁盘数据块为单位的最大核心转储文件大小。下面就是一个重置核心转储文件大小来为上面的示例程序服务的示例（为简洁起见，对输出进行了精简）：

```
$ ulimit -c 1024
$ ulimit -a
core file size          (blocks, -c) 1024
```

请注意，在你的Linux发行版上，这样做可能还不足以重置核心转储文件的大小，这要取决于你的Linux系统的具体配置情况。

当设置了适当的核心转储文件大小之后，重新运行示例程序将导致发生一个真正的核心转储：

```
$ ./buggy
Segmentation fault (core dumped)
```

你将在当前目录下看到一个新的核心转储文件：

```
$ ls
buggy  buggy.c  core.21090  hello  hello.c
```

值得注意的是，在这个特定Linux发行版的情况下，核心转储文件的文件名包含了运行原程序的进程号。这个可选的特征可能并没有在你的Linux发行版所提供的内核配置中启用。

GDB可以读取核心转储文件并基于该文件开始一个调试会话。由于核心转储文件是由一个不再运行的程序产生的，所以并不是所有的gdb命令都可以使用——例如，试图在一个不再运行的程序中执行步进操作是没有意义的。但核心转储文件还是非常有用的，因为你可以以离线方式调试它们，所以你的用户可以将核心转储文件以及他们的本地机器环境通过电子邮件发送给你^①，以方便你进行远程调试。

^① 或者你也可以通过编写一个工具来代表用户向你返回有用错误报告，正如GNOME bug buddy等工具所做的那样，这可能更会让你的软件获得成功。

你可以针对示例核心转储文件运行GDB:

```
$ gdb ./buggy core.21090
GNU gdb Red Hat Linux (6.3.0.0-1.122rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "ppc-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

Failed to read a valid object file image from memory.
Core was generated by `./buggy'.
Program terminated with signal 11, Segmentation fault.

warning: svr4_current_sos: Can't read pathname for load map: Input/output error

Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld.so.1...done.
Loaded symbols for /lib/ld.so.1
#0 0x0f65a09c in strncpy () from /lib/libc.so.6
```

然后，你可以使用我们前面介绍的相同的方法继续调试会话。只需要记住程序流控制命令不能被使用——因为程序早已停止运行。

10.2.2 Valgrind

Valgrind是一个运行时诊断工具，它可以监视一个指定程序的活动并通知你在你的代码中可能存在的各种各样的内存管理问题。它类似于老式的Electric Fence工具（该工具将标准的内存分配函数替换为自己的函数以提高诊断能力），但被认为更容易使用并且在多个方面都提供了更丰富的功能——而且现在大多数主流Linux发行版都提供了该工具，所以在你的系统中使用它不需要花费太多时间，你只需安装它的软件包即可。

一个典型的Valgrind运行可能如下所示：

```
$ valgrind ./buggy
==2090== Memcheck, a memory error detector.
==2090== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.
==2090== Using LibVEX rev 1658, a library for dynamic binary translation.
==2090== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==2090== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==2090== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==2090== For more details, rerun with: -v
==2090==
==2090== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 1)
==2090== malloc/free: in use at exit: 80 bytes in 10 blocks.
==2090== malloc/free: 10 allocs, 0 frees, 80 bytes allocated.
==2090== For counts of detected errors, rerun with: -v
==2090== searching for pointers to 10 not-freed blocks.
==2090== checked 47,492 bytes.
```

```

==2090==
==2090== LEAK SUMMARY:
==2090==   definitely lost: 80 bytes in 10 blocks.
==2090==   possibly lost: 0 bytes in 0 blocks.
==2090==   still reachable: 0 bytes in 0 blocks.
==2090==   suppressed: 0 bytes in 0 blocks.
==2090== Use --leak-check=full to see details of leaked memory.

```

输出显示有80个字节的内存程序结束时丢失了。通过指定leak-check选项，我们可以找到这个泄漏的内存来自哪里：

```

==2101== 80 (8 direct, 72 indirect) bytes in 1 blocks are definitely lost in loss
record 2 of 2
==2101==    at 0xFF7BBE0: malloc (vg_replace_malloc.c:149)
==2101==    by 0x100004B0: main (buggy.c:25)

```

你应该养成习惯在可能的情况下使用诸如Valgrind这样的工具来对发现和修复内存泄漏以及其他编程错误的过程进行自动化。因为这里只对Valgrind进行了肤浅的介绍，所以你需要查看它的在线文档以更全面的了解其功能。事实上，越来越多的开放源码项目都依赖于Valgrind作为其回归测试（任何一个具有相当规模的软件项目的一个重要组成部分）的一部分。

自动化代码分析

有越来越多的第三方工具可以用于执行自动化代码分析，寻找软件中各种典型类型的缺陷。这类代码覆盖工具一般提供静态、动态或混合形式的代码分析。这意味着工具可能只是检查源代码以确定潜在的缺陷，或它可能试图钩入其他一些进程，以获取确定软件中缺陷可能存在位置所必需的数据。

基于斯坦福大学的checker的商业代码分析工具Coverity经常被用在Linux系统中。它钩入编译过程并提取大量有用的信息，这些信息可用于发现很多潜在的问题。事实上，Coverity为越来越多的开放源码项目提供免费代码分析。它甚至还发现了Linux内核中相当多的以前未被发现的错误。这些问题被发现后立即得到解决。

静态代码分析的一个比较有趣的用途是查找源代码中是否有非法使用GPL代码的情况。Blackduck软件就提供了这样一个工具，它可以帮助你扫描你的大型软件项目，以查找借用自开放源码项目的源代码，并确定处理方法。这对兼容性测试以及其他你的法律团队可能会提醒你进行的活动将非常有用。

10.3 图形化调试工具

尽管你可以在大多数（即便不是全部）Linux调试任务中使用GDB，但与长时间坐在GDB命令行前面相比，许多人还是更愿意使用诸如DDD或Eclipse这样的图形化工具。从各方面来看，大多数非常花哨的图形化调试工具不过是建立在GDB基础上的一个抽象，所以选择哪一种图形化工具完全属于个人爱好。

本节将介绍两个这样的工具，当然还存在其他许多这样的工具（包括GDB的前端GNU insight），它们也被各种开发团队所使用。

10.3.1 DDD

数据显示调试器或DDD最初作为一个大学工程项目的一部分被编写，但它很快就因为其友好、易

于使用的界面以及它强大的数据可视化功能而在社区中变得非常流行。DDD实际上只是一个GDB的直接驱动程序，GDB运行在底层窗口，而源代码和数据显示则显示在上层窗口中。DDD应该作为你的Linux发行版的一部分被预安装。如果它没有被制作成软件包，你也可以通过源代码编译它。

你可以通过在命令行上输入ddd或通过点击相应的菜单选项来启动DDD。当它被装载后，你需要告诉DDD打开哪个程序。点击File菜单，选择“Open Program”，然后将有错误的示例程序装入调试器。如果你已正确地编译示例代码（告诉GCC将调试信息留在二进制文件中），那么除了反汇编和状态信息以外，程序的源代码（如果有的话）也将出现在源代码窗口中。

DDD图形化界面的主窗口如图10-1所示。

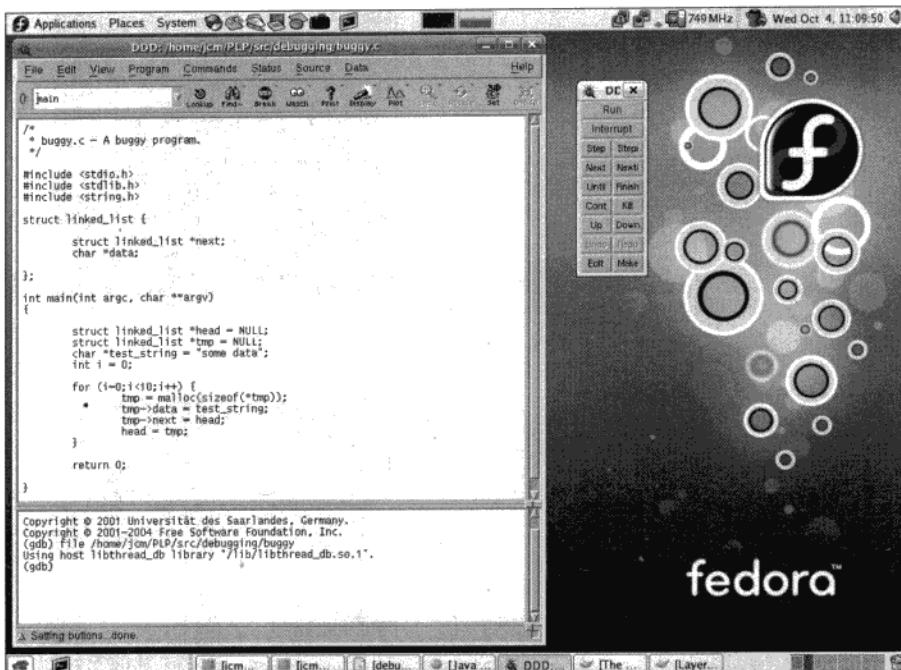


图10-1 DDD图形化界面的主窗口

可以轻松地在程序中插入或删除断点，只需用鼠标右键单击程序中相应的代码。在弹出的上下文菜单中将包括用于添加断点或删除任何已定义断点以及继续程序执行直至到达鼠标光标所在位置（另一种替代明确设置断点的方法）的选项。图10-2显示了在一个有错误程序的main函数入口处设置断点的情况。

既然DDD是一个数据显示调试器，它的数据显示功能当然是很棒的。事实上，通过点击View，然后选择“Data Window”，就可以打开另一个窗口，在该窗口中，各种类型的数据都可以可视化显示，如图10-3所示。在数据窗口中，只需简单地插入预定义的表达式来定义你想要查看的数据即可，例如tmp·data用来显示包含在临时链表元素tmp中的字符串值。

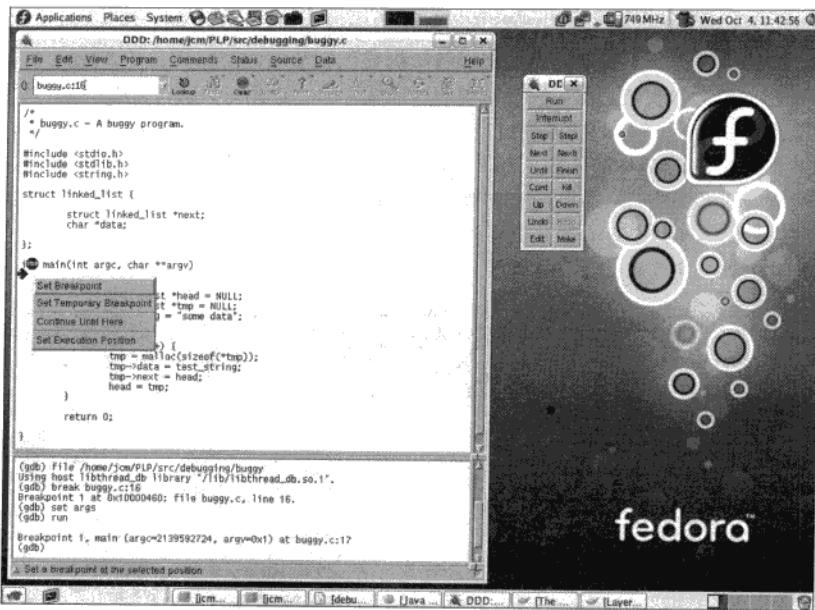


图10-2 在有错误程序的main函数入口处设置断点

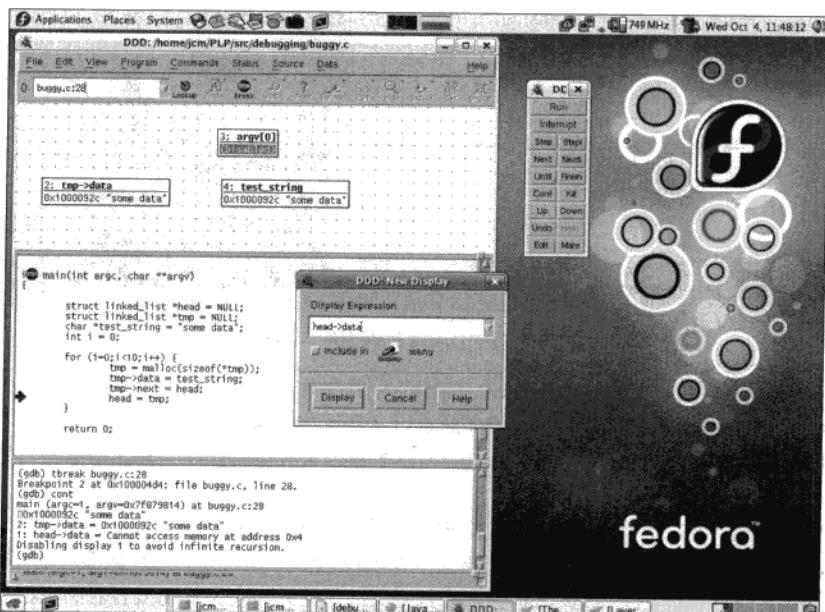


图10-3 各种数据在数据窗口中的可视化显示

10.3.2 Eclipse

如今，每个人都希望能尽可能有效地使用诸如Eclipse这样的集成工具。Eclipse实际上又包括了另一个你可以在该环境中直接使用的GNU调试器（GDB）的图形化前端。首先，通过在命令行中输入eclipse或点击桌面菜单中相应的选项启动eclipse。然后选择“New Project wizard”并创建一个名为buggy的项目。接着，通过点击File→New→Source File创建一个名称为buggy.c的C源文件，并将示例文件的内容输入，如图10-4所示。

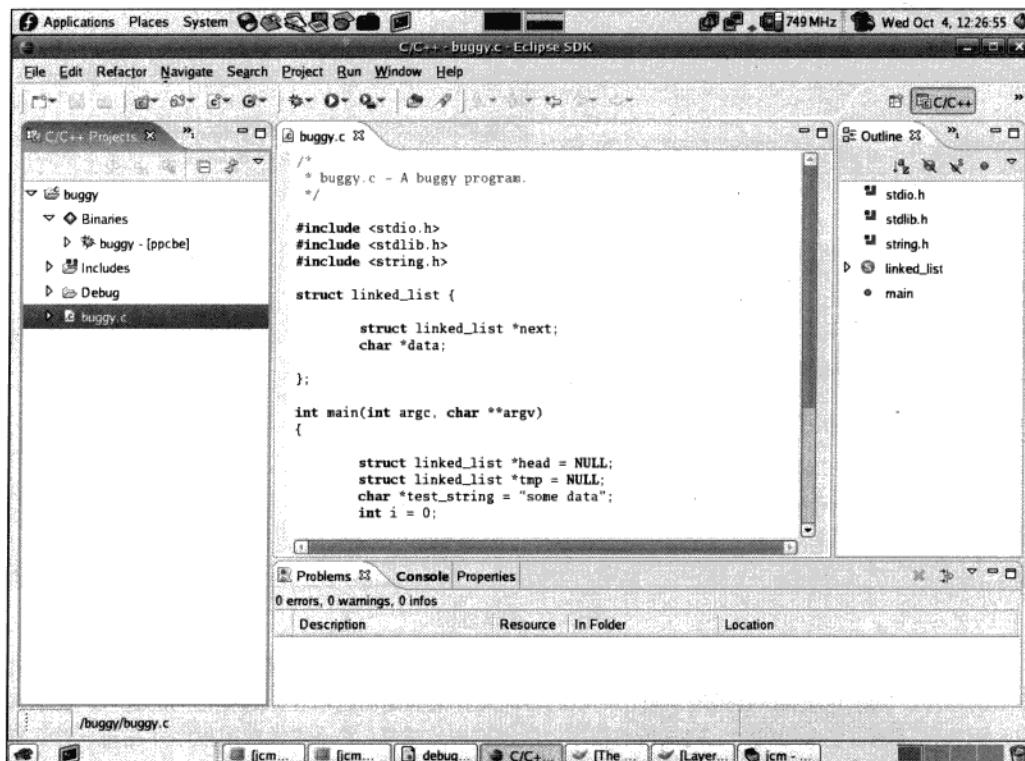


图10-4 将示例中的内容输入到新建的项目中

每当在Eclipse中保存源文件时，Eclipse将自动编译该源文件，但你也可以使用Project菜单中的选项来达到同样的效果（例如，如果在Eclipse中禁用了自动工作空间编译功能）。你可以在左边窗格的Binaries标题下看到列出的二进制输出文件。它的类型是ppcbe，因为它是在一个大头字节序的PowerPC笔记本电脑（一个运行Linux的Apple Macintosh电脑）上编译的，但输出结果将根据你所选择的平台而有所不同。

为了开始调试有错误的程序，你需要切换到调试视图，如图10-5所示。Eclipse主窗口的右上角提供了用来切换视图的图标。你应该可以在该列表中找到调试视图的选项，除了该选项以外，列表中还

会提供厂商或第三方Eclipse软件包提供的其他选项（它们可能为你提供了大量的“增值”功能）。

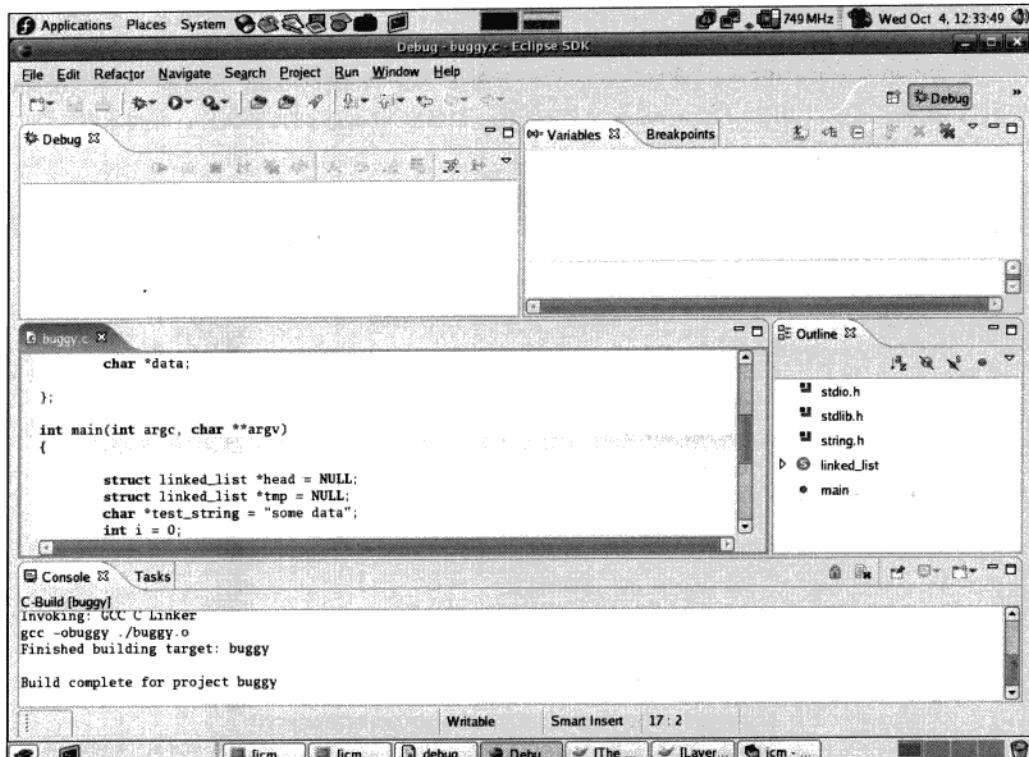


图10-5 调试视图

为了开始调试程序，从Run菜单中选择Debug，将获得一个选项列表，可以根据是调试一个本地机器上的程序、执行远程调试还是使用输出的核心转储文件来确定问题所在来选择不同的选项。在本例中，因为你只是想在本地机器上调试当前程序，所以你应该选择C/C++本地应用程序并在向导对话框中定义一个新的调试配置。

当你创建了一个正确的调试配置后，向导对话框中的Debug按钮将被激活，现在你可以对应用程序开始真正的调试了。点击该按钮将导致程序进入调试模式，程序将启动（带有你在向导对话框中指定的选项）并自动在main函数的入口插入一个断点。

你应该会看到一个类似于图10-6所示的画面。

现在你可以像在GDB和DDD中一样继续调试应用程序了。关于Eclipse调试及其各种选项的更多信息请查看在线Eclipse IDE文档，你将从中学到一些更高级的功能，包括对嵌入式Linux设备远程调试的支持——通过Eclipse控制整台机器。

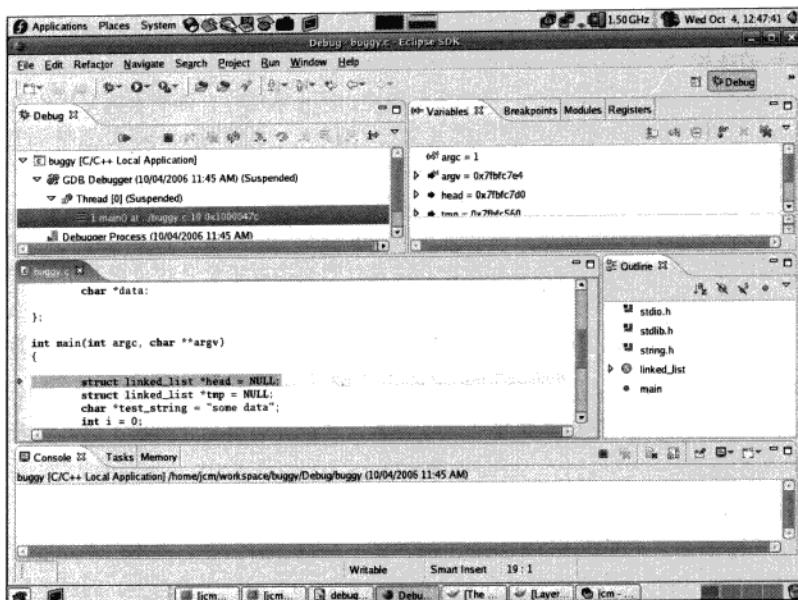


图10-6 进入调试模式

10.4 内核调试

在前面几节中，你学习了如何对运行在Linux内核控制之下的普通应用程序进行调试。对于这种情况，你将直接受益于由Linux内核提供的众多基础支持功能。Linux内核提供了trace系统调用用于处理普通程序的跟踪，它监视错误的内存处理并通过许多其他的调度和进程管理抽象来避免应用程序运行出错，而这一切你通常都视为理所当然。

但从根本上来说Linux内核只是一个复杂的软件，和其他所有的复杂的软件一样，它容易碰到一些令人不快的错误。而且当内核出现错误时，你会很快知道——因为即便是内核中的小错误也会对整个系统的稳定性造成非常坏的影响。所幸的是，Linux内核是一个编写良好并被专业维护的软件项目，有成千上万的人不断地工作以消除Linux内核中的这类错误。

本节将简要介绍内核调试，但由于其自身的复杂性，你可能希望能在走上内核调试的漫长路途之前咨询更多的在线参考资料。

10.4.1 不要惊慌！

根据内核错误的特性及其发生的准确位置（在核心内核中、设备驱动程序中或其他上下文中），系统可能可以继续运行或无法继续运行。系统通常在损失一些功能的情况下（通常会产生一个非常丑陋的消息被记录到系统日志中）仍然可以继续运行并产生一个被称为oops的错误^①。oops将导致当前任

^① 有些企业版Linux厂商将他们的内核定制为当发生一个oops时就使内核崩溃。这意味着即使是一个轻微的内核错误也将导致系统立刻崩溃——这样做的好处是避免了内核错误可能在今后带来的内核数据的损坏、文件系统的破坏以及其他更大的隐形损害。

务（进程）被杀死，该任务所使用的所有资源将处于不太一致的状态。此时，应该尽可能快地重启一个产生oops错误的系统。

在系统无法继续安全运行的情况下，内核将立即停止其正常运行并强制系统突然死亡。这种情况被称为内核panic，UNIX和类UNIX内核在历史上（甚至如今）一直使用panic函数强制系统崩溃。虽然一些数据可能会在panic之后丢失（先前没有提交给磁盘的数据，包括无效磁盘缓冲），但系统已安全阻止了进一步的（实际上更为严重的）数据损坏。这是一种从安全角度考虑的折中。

请注意，你的系统也许可以通过使用最近引入到Linux 2.6内核中的kexec和kdump机制来支持内核崩溃转储（旧系统使用各种替代方法）。你应该查询发行版文档以了解是否可以利用这个机制来保存内核panic产生的崩溃输出——这比手工记录崩溃输出内容或以数码照片（是的，真的是这么做）拍摄在内核崩溃时显示在屏幕上的内核调试信息要好得多。有时候，它甚至是获取重要诊断数据的唯一途径^①。

如果你的系统安装了panic X视窗屏幕保护程序，可以通过激活它来看到各种各样的Linux和其他UNIX以及类UNIX系统中的panic屏幕（欺骗你的朋友和同事，使他们误认为你的系统经常会崩溃）。

10.4.2 理解oops

每当Linux内核遇到一个非致命（但仍然非常严重）的错误时，就会产生一个oops。在这种情况下，一个内核栈回溯、寄存器内容以及其他相关信息将被打印到系统主控台上，同时也被记录到系统日志中。请注意，你的X视窗系统显示并不是系统的主控台。所以，如果你使用的是图形化系统，当oops发生在主控台上时，你可能永远也看不到它的实时输出，但你仍然可以通过系统日志看到其内容。

下面是一个示例oops，它来自最近发送给LKML（Linux内核邮件列表）的一封邮件：

```
Unable to handle kernel NULL pointer dereference at virtual address 0000025c
printing eip:
d118143f
*pde = 00000000
Oops: 0000 [#!]
PREEMPT
Modules linked in: saa7134 i2c_prosavage i2c_viaopro ehci_hcd uhci_hcd
usbcore 8250_pci 8250 serial_core video_buf compat_ioctl32 v4l2_common
v4l1_compat ir_kbd_i2c ir_common videodev via_agp agpgart snd_via82xx
snd_ac97_codec snd_ac97_bus snd_mpu401_uart snd_rawmidi snd_seq_device
snd_rtctimer snd_pcm_oss snd_pcm snd_timer snd_page_alloc
snd_mixer_oss snd_soundcore it87 hwmon_vid hwmon i2c_isa video fan
button thermal processor via_rhine uinput fuse md5 ipv6 loop rtc
pcspkr ide_cd cdrom dm_mod
CPU: 0
EIP: 0060:[<d118143f>] Not tainted VLI
EFLAGS: 00010246 (2.6.16-rc4 #1)
EIP is at saa7134_hwinit2+0x8c/0xe2 [saa7134]
eax: 00000000 ebx: cc253000 ecx: 00000000 edx: 0000003f
esi: cc253000 edi: cdc36400 ebp: cc2530d8 esp: c05d3e44
```

^① 当你使用X视窗系统时尤其如此。因为X配置你的显卡以实现图形输出，所以内核通常不能将有意义的数据传送到显示器。对于那些不带显示器的设备而言，情况就更加复杂，而那些使用真正帧缓冲设备的系统（通常是在非Intel x86平台上）经常会在其图形显示中将内核输出叠加在一起。

```

ds: 007b    es: 007b    ss: 0068
Process modprobe (pid: 7451, threadinfo=c05d2000 task=cb24fa70)
Stack: <0>cc253000 cc253000 00000000 d1181ae7 cc253000 d1180fdc
24000000 cc2530d8
        cc253000 00000003 d118b1cd ffffffed cdc36400 d1199920 d119994c c02188ad
        cdc36400 d1198ef4 d1198ef4 d1199920 cdc36400 d119994c c02188ea d1199920
Call Trace:
[<d1181ae7>] saa7134_initdev+0x332/0x799 [saa7134]
[<d1180fdc>] saa7134_irq+0x0/0x2d6 [saa7134]
[<c02188ad>] __pci_device_probe+0x5f/0x6d
[<c02188ea>] pci_device_probe+0x2f/0x59
[<c026b26a>] driver_probe_device+0x93/0xe5
[<c026b325>] __driver_attach+0x0/0x69
[<c026b38c>] __driver_attach+0x67/0x69
[<c026a747>] bus_for_each_dev+0x58/0x78
[<c026b3b4>] driver_attach+0x26/0x2a
[<c026b325>] __driver_attach+0x0/0x69
[<c026acba>] bus_add_driver+0x83/0xd1
[<c026b883>] driver_register+0x61/0x8f
[<c026b80e>] klist_devices_get+0x0/0xa
[<c026b818>] klist_devices_put+0x0/0xa
[<c0218b33>] __pci_register_driver+0x54/0x7c
[<c011be9a>] printk+0x17/0x1b
[<d1182229>] saa7134_init+0x4f/0x53 [saa7134]
[<c0137d75>] sys_init_module+0x138/0x1f7
[<c0102f65>] syscall_call+0x7/0xb
Code: 24 04 e8 68 aa f9 ee 89 1c 24 e8 d8 81 00 00 89 1c 24 e8 74 3e
00 00 83 bb d0 00 00 00 01 ba 3f 00 00 00 75 a9 8b 8b d4 00 00 00 <8b>
81 5c 02 00 00 a9 00 00 01 00 75 0e a9 00 00 04 00 74 2e ba

```

乍看起来，这个oops似乎相当复杂，但我们可以比较容易地对它进行分解。首先，注意开头的消息：Unable to handle kernel NULL pointer dereference at virtual address 0000025c（不能处理在虚拟地址0000025c处解引用的内核NULL指针）。这相当于在内核空间中的一个段错误，但这次无法通过操纵未分配内存或跟随某处的NULL指针来挽救内核。在本例中，EIP（指令指针）位于0xd118143f，随后内核将它巧妙地转换为：

```
EIP is at saa7134_hwinit2+0x8c/0xe2
```

这意味着内核正在执行函数saa7134_hwinit2中0x8c字节处的代码，该函数的编译形式长度为0xe2字节。为了能够定位到源代码中准确的代码行，可以使用工具objdump从内核文件vmlinu中将该函数的反汇编代码进行转储，正如我们在本书第2章中介绍的那样。通常情况下，只需知道执行了哪个函数以及了解了来自oops消息中的栈回溯和当前寄存器上下文就足够解决问题了。

oops的输出确认这是自系统启动以来产生的第一个oops，并且内核在编译时带有抢占式内核支持（请看“底层Linux”以了解进一步的细节）。内核还提供了在产生oops时已装载进内核的完整模块列表。一个典型的Linux系统中会有很多这样的模块，因为如今的趋势是朝着对驱动程序和其他可以被模块化的非基本核心内核架构不断模块化的方向发展。

接下来，oops输出描述了系统中安装的CPU的类型和特性，以及在产生oops时它们的寄存器内容。从这个Intel IA32（x86）处理器的输出中，可以看到寄存器EAX包含NULL值，内核试图使用它作为一

个指针值，从而导致该oops的发生。内核告诉我们当前任务是modprobe，因此我们可以合理地推断出这个oops是在驱动程序saa7134被装载后立刻发生的。

oops最后输出的是一个（可能冗长的）栈回溯，它显示了内核是如何到达产生错误的位置的。从显示的回溯信息中可以很明显地看出，saa7134_init注册了一个新的PCI设备驱动程序，该驱动程序立即产生了一个对模块PCI探查函数的调用，而在saa7134_initdev函数的某处，一个NULL指针的解引用导致了oops的发生。如果这一切现在看来还不是很明显，也不用担心——你需要花费一些时间来阅读oops消息以掌握它。

1. 不要相信随后的oops

请牢记，oops是在Linux内核“发生了错误”后产生的。这意味着你真的不希望继续运行系统并且还期待系统能够保持稳定，或希望随后的oops输出能够富有含义（它可能会受到第一个oops之后的不稳定系统环境的影响）。有时候，在系统被锁死或崩溃之前捕获有用的输出并非易事。在这种情况下，投资购买一个空调制解调器电缆并将你的系统设置为远程记录主控台输出还是非常值得的。你可以在各种在线资源中找到这方面的示例。

2. 向上游报告错误

如果你不能自己找出产生oops的原因并希望将内核错误报告给上游的维护者，需要首先查看位于你的Linux内核源代码顶层目录中的REPORTING-BUGS文件的内容。该文件描述了你应该遵循的正确步骤以及发送邮件的格式。注意，如果你使用的是一个旧内核（请联系你的供应商）或使用了专有驱动程序，你也许不能从内核社区获得支持。

请注意，当你使用专有的非GPL驱动程序时，内核社区可以分辨出来。很多时候，人们会在递交给上游的内核错误报告中将二进制模块列表移除，但内核社区在这一点上是足够聪明的。他们能够看出来（并可能在今后不再提供支持）。如今一些发行版甚至会修改oops报告的输出格式以使得内核开发者更加难以掩盖这一信息。

10.4.3 使用UML进行调试

调试真正的硬件是一个艰巨的任务，特别是如今存在着这么多的平台和架构。即便是提供一个可以实际解决一小部分硬件的示例也决非易事——至少在UM Linux、gemu等工具以及其他内核虚拟化技术出现之前是这样。现在，通过使用模拟器，我们已可以安全地调试某些类型的内核问题而不用冒着造成系统崩溃的危险了。

现在有各种类型的模拟技术可用于在各种模拟环境中运行Linux内核。其中最简单的一种方法是使用用户模式（UM）内核，这是因为它现在已内置在主流Linux内核中并可以在编译一个新的Linux内核时作为它的编译时选项。用户模式Linux（UML）将Linux内核作为一个普通应用程序来运行。它使用信号来模拟硬件中的时钟和中断，它并不直接和真正的硬件通信。尽管如此，它仍然是一个Linux内核并且对测试某些内核功能来说非常有用。

注意，你不能使用UML来测试自己的Linux内核硬件设备驱动程序——这并不是它的设计目的——但你可以跟踪内核执行并测试纯软件功能。例如，可以用它来测试最新的纯软件文件系统。

编译一个UML内核

UM Linux内核是一种虚拟Linux架构，它必须在内核编译时指定。你首先需要从kernel.org网站上下载一个全新的Linux内核源代码拷贝并准备一些磁盘空间，然后使用GNU工具链来编译和测试UM内核。

首先调用tar命令在一个新目录中解开Linux内核源代码：

```
$ tar xvfj linux-2.6.18.tar.bz2
```

接下来，配置新的内核源代码，指定UM架构以取代你的真实系统架构（通常是自动检测的）：

```
$ make ARCH=um menuconfig
```

注意，Linux内核目标架构是通过ARCH环境变量指定的。

选择你想要在UM内核中使用的编译选项，然后开始编译：

```
$ make ARCH=um
```

这个编译过程的输出结果是一个名称为linux的普通Linux ELF可执行文件。你可以像运行其他任何普通应用程序一样运行这个特殊的Linux内核：

```
$ ./linux
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...missing
Checking for tmpfs mount on /dev/shm...OK
Checking PROT_EXEC mmap in /dev/shm...OK
Checking for the skas3 patch in the host:
  - /proc/mn...not found
  - PTRACE_FAULTINFO...not found
  - PTRACE_LDT...not found
UML running in SKAS0 mode
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...missing
Linux version 2.6.18 (jcm@panic) (gcc version 3.3.5 (Debian 1:3.3.5-13)) #1 Wed Oct
4 16:00:55 BST 2006
Built 1 zonelists. Total pages: 8192
Kernel command line: root=98:0
PID hash table entries: 256 (order: 8, 1024 bytes)
Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
Memory: 30116k available
Mount-cache hash table entries: 512
Checking for host processor cmov support...Yes
Checking for host processor xmm support...No
Checking that host ptys support output SIGIO...Yes
Checking that host ptys support SIGIO on close...No, enabling workaround
Using 2.6 host AIO
NET: Registered protocol family 16
NET: Registered protocol family 2
IP route cache hash table entries: 256 (order: -2, 1024 bytes)
TCP established hash table entries: 1024 (order: 0, 4096 bytes)
TCP bind hash table entries: 512 (order: -1, 2048 bytes)
TCP: Hash tables configured (established 1024 bind 512)
TCP reno registered
```

```

Checking host MADV_REMOVE support...MADV_REMOVE failed, err = -22
Can't release memory to the host - memory hotplug won't be supported
mconsole (version 2) initialized on /home/jcm/.uml/INfrHO/mconsole
Host TLS support detected
Detected host type: i386
VFS: Disk quotas dquot_6.5.1
Dquot-cache hash table entries: 1024 (order 0, 4096 bytes)
io scheduler noop registered
io scheduler anticipatory registered (default)
io scheduler deadline registered
io scheduler cfq registered
TCP bic registered
NET: Registered protocol family 1
NET: Registered protocol family 17
Initialized stdio console driver
Console initialized on /dev/tty0
Initializing software serial port version 1
Failed to open 'root_fs', errno = 2
VFS: Cannot open root device "98:0" or unknown-block(98,0)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(98,0)

EIP: 0073:[<40043b41>] CPU: 0 Not tainted ESP: 007b:40152fbc EFLAGS: 00000246
    Not tainted
EAX: 00000000 EBX: 00001594 ECX: 00000013 EDX: 00001594
ESI: 00001591 EDI: 00000000 EBP: 40152fc8 DS: 007b ES: 007b
08897b10: [<0806cddd8>] show_regs+0xb4/0xb6
08897b3c: [<0805aa84>] panic_exit+0x25/0x3f
08897b4c: [<0807e41a>] notifier_call_chain+0x1f/0x3e
08897b6c: [<0807e4af>] atomic_notifier_call_chain+0x11/0x16
08897b80: [<08071a8a>] panic+0x4b/0xd8
08897b98: [<08049973>] mount_block_root+0xff/0x113
08897bec: [<080499d8>] mount_root+0x51/0x56
08897c00: [<08049aab>] prepare_namespace+0xce/0xfa
08897c10: [<0805644d>] init+0x73/0x12a
08897c24: [<08066be8>] run_kernel_thread+0x45/0x4f
08897ce0: [<0805aeee0>] new_thread_handler+0xc3/0xf5
08897d20: [<fffffe420>] _etext+0xf7e72405/0x0

<3>Trying to free already-free IRQ 2
Trying to free already-free IRQ 3

```

正如你看到的，因为内核无法装载一个根文件系统，所以它崩溃了。你可以下载一份伪Linux文件系统的拷贝用于UML，但对于Linux内核的简单实验来说，这并不是必须的（因此没有在这里进行介绍）。内核确实崩溃并产生输出这一事实显示了UML的使用就和现实的内核使用一样（每当Linux内核不能装载其根文件系统并且没有其他选择时，它就会崩溃）。

UM Linux可以在GDB中运行，我们需要使用debug选项来调用它。首先启动GDB：

```

$ gdb ./linux
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are

```

```
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
```

然后启动UM Linux，给它传递参数debug（否则将难于对它进行调试，因为UM Linux默认使用信号处理程序来处理内部中断——这个选项将改变UM Linux的默认行为以使它使用另一种替代机制来与其自身通信）：

```
(gdb) run debug
Starting program: /home/jcm/linux-2.6.18/linux debug
'debug' is not necessary to gdb UML in skas mode - run
'gdb linux' and disable CONFIG_CMDLINE_ON_HOST if gdb
doesn't work as expected
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...missing
Checking for tmpfs mount on /dev/shm...OK
Checking PROT_EXEC mmap in /dev/shm/...OK
Checking for the skas3 patch in the host:
- /proc/mn...not found
- PTRACE_FAULTINFO...not found
- PTRACE_LDT...not found
UML running in SKAS0 mode

Program received signal SIGUSR1, User defined signal 1.
0x40043b41 in kill () from /lib/tls/libc.so.6
```

尝试在高层C语言启动代码的开始处插入一个断点，该启动代码存在于所有Linux内核中。要做到这一点，在start_kernel函数上设置一个断点即可。

你可以在内核运行到start_kernel之后跟踪内核的启动过程：

```
(gdb) break start_kernel
Breakpoint 1 at 0x80493fd: file init/main.c, line 461.
(gdb) continue
Continuing.

Breakpoint 1, start_kernel () at init/main.c:461
461          smp_setup_processor_id();
```

你当然还可以做更多比单纯跟踪内核启动更强大的事情。通过在一个调试器中运行内核，可以插入断点并充分利用GDB的所有功能，而所有这一切都不需要使用一个运行在实际机器上的昂贵的硬件调试器。当你需要对内核的功能进行扩展和增强时，我们鼓励你花费一些时间来研究UM Linux。

10.4.4 一件轶事

关于本章的一件有趣的事是，它帮助我找到了内核中的一个错误。本书的原稿是使用Open Office文本编辑器撰写的，但一些后期处理工作是使用Microsoft Word模板来完成的，我们使用CrossOver office来运行Microsoft Word（CrossOver是wine的商业版本，后者是一个自由的Windows模拟器，它允许各种Windows应用程序未经修改就可以在图形化Linux环境中运行）。

看起来Microsoft Office喜欢在它被装载时打开音频设备（不论出于什么原因）。wine应该愉快地接

受模拟Windows API调用的任务并将结果传递给Word应用程序，至少，事情应该是这么发生的。但实际情况是，声卡驱动程序将试图初始化它的并不存在的MIDI接口，并在知道实际上并没有安装这样一个接口时变得非常不愉快。这将导致每次当Microsoft Word被装载时都会产生一次硬锁死。

硬锁死是最严重的内核问题类型之一。它们自动发生，对此你没有什么可以做的（除了重启或尝试一些较新的Linux内核死锁检测补丁集），而且你可能不会有太多有用的日志可以进行检查。但是，如果稍加审慎地思考并在合适的位置放置printk语句，我们就可以找出在内核崩溃之前它究竟做了些什么。一旦问题被发现，那么修复声卡驱动程序并不会花太长的时间。

现在，我们就可以安全地装载Microsoft Windows应用程序而不会导致系统崩溃了。

10.4.5 关于内核调试器的注记

“我不认为内核开发是一件‘很容易的’事情。我不赞成通过单步调试代码来发现错误的做法。我不认为系统的额外可见度是一件必要的好事”。

当Linus Torvalds（Linux内核的创造者）在几年前写下上述这段话时，他是在重复他已在之前讲过很多次的观点——他认为在许多情况下，内核调试器的害处比它的好处要多。他的观点似乎是说内核开发者应该对代码非常了解，他们不需要使用调试器——他们应该只需要很少的有用提示（诸如一个oops报告）就可以凭借其本能感知到错误的位置。

更重要的是，内核调试器通常会改变内核操作所处的环境。因为它们会改变内核的工作方式，所以一个正确的观点是调试器会引入本来并不存在的问题，或导致一个本应发生的错误却因为特定的环境而不能被检测出来。如果不考虑这种情况发生的可能性很小，使用一个调试器在某些情况下还可能会改变它所调试的对象。

当然，并非每个人都赞同不应该有内核调试器这样一种观点。虽然像Linus这样足够聪明的人可以仅使用最少的帮助就能找出问题所在是值得赞扬的，但对于其他许多开发者来说，一个调试器还是有相当大用处的。基于这个原因，有各种第三方补丁和项目用于修改内核以使得它可以被远程调试，例如，通过网络使用GDB（GDB over the network），使用一个空的调制解调器电缆或甚至使用FireWire。

注意，针对Linux的内核调试器都是非标准的，所以它们没有在本书中进行介绍。不过你可以在因特网上找到大量这类项目的信息，如KGDB。这些由第三方编写的项目通常都提供了必需的功能。

10.5 本章总结

在本章中，你学习了如何在Linux系统中调试应用程序。你学习了GDB及其图形化前端DDD、Eclipse以及一些其他项目。你还了解了可能发生在Linux系统中的某些典型类型的错误并看到了一个简单的调试会话的处理过程。最后，你学习了一些内核调试的知识，如何破译和提交oops，为什么Linux内核自身没有内置一个完整的内核调试器。

你现在应该感到自己完全有能力去更多地了解如GDB这样的调试工具了。

GNOME开发者平台

你可能很熟悉或至少知道流行的GNOME桌面环境。它通常被看作另一个类似的同样也很流行的K桌面环境（KDE）的竞争对手。GNOME是开放源码项目的旗舰级产品之一，它也是使得GNU/Linux系统能够被普通用户所接受的最佳方法之一。

GNOME桌面环境基于被称为GNOME开发者平台的函数库栈。构成GNOME用户体验的应用程序如面板、会话管理器和Web浏览器都被称为GNOME桌面。因为GNOME平台和桌面是严格按照每6个月的时间表同时发布的，所以其发展理念是渐进的改善而不是激进的改变和重写。但是，一些通常被视为平台的一部分的函数库并不按照这个时间表发布——Glib和GTK+有它们自己的时间表，因为它们并非GNOME项目的一部分，即便它们有许多相同的核心开发人员。

本章通过建立一个简单的音乐播放器程序来介绍GNOME开发者平台，该程序只使用了平台提供的库函数。在学习完本章之后，你应该为进一步研究函数库和GNOME应用程序的开发做好了准备。我们还将列出一些可能的对音乐播放器进行改善方法，这需要你进行一些研究以发现合适的解决方案。本章假定你对面向对象编程有一个基本的了解。

在开始编程之前，首先了解一些在GNOME开发者平台中最重要的函数库还是很有必要的。

11.1 GNOME 函数库

GNOME系统提供了很多函数库。成功编写GNOME程序的一个诀窍是只使用你的应用程序需要的函数库。GNOME函数库以一种分层方式进行交互——底层核心函数库向GNOME桌面提供一个接口并成为更高层图形化函数库建立的基础。

本节显示了用于创建音乐播放器程序的GNOME函数库。这里显示的每个函数库都是基于在它之前介绍的部分或全部函数库的。一个工作在GNOME平台上的开发者应该只使用那些必需的函数库，但为了能和GNOME桌面完全整合，所需要的函数库可能相当多。

11.1.1 Glib

GNOME平台的核心是Glib。它从本质上来说是到C标准函数库的一个“推动器”，Glib提供了对许多程序都有用处的可移植数据类型，如Unicode字符串和链表。它还提供了基本算法和可移植方法，它们用于完成诸如文件系统路径构建、解析命令行选项和确定用户家目录等任务。

Glib最初属于GTK+的一部分，但现在被拆分为一个单独的函数库使之能够用于不需要GUI的程序。

11.1.2 GObject

GObject基于Glib，它使用C语言提供了一个单根继承的面向对象系统。虽然这个函数库的实现很复杂，并且其新类的创建即使与C++相比也过于繁琐，但基于GObject的函数库的使用还是比较一致和

简单的。GObject位于它自己的函数库中，但它和Glib放在同一个软件包中发布。

GObject的某些复杂性源于它的一个设计原则：可以轻松地以任何语言来创建对基于GObject函数库的绑定。因此，大多数GNOME平台的函数库都已针对大量的语言提供了绑定，包括C++、C#（以及其他.NET语言）、Perl、Python、Ruby和Haskell。这些绑定能够充分地利用语言本身的面向对象和内存管理功能。

11.1.3 Cairo

Cairo是一个图形函数库，其API模仿了PostScript的绘图模型。Cairo提供了在任何输出设备上绘制高质量、分辨率独立的图形的一个比较简单的方法。它具有多个输出后端，既可以使用Xlib或Windows图形模型在屏幕上进行简单的绘图，也可以生成PostScript或PDF用于打印。因此，从绘图代码的角度来看，Cairo应用程序在很大程度上是平台无关的。

关于Linux上的Cairo后端的一个正在开展的有趣工作是Glitz函数库，它使用硬件加速Open GL命令来实现所有的绘图操作。对于那些带有3-D图形硬件设备的许多现代桌面系统来说，这提供了极大加速其GNOME桌面中的复杂图形处理速度的潜力。

11.1.4 GDK

GDK（GIMP绘图工具包）是一个图形函数库，GTK+使用它来绘制其窗口部件。因为它作为GTK+软件包的一部分发行，所以它并不适合单独使用。从GTK+的2.8版本开始，GDK基于Cairo实现其功能，当应用程序需要绘制自己的图形时，应优先考虑使用Cairo的绘图命令。

11.1.5 Pango

Pango是一个精致的文本渲染系统，它用于布置以各种国际化语言书写的文本。GTK+使用它来渲染用户界面中的所有文本，它还支持各种屏幕显示增强功能，如抗锯齿和亚像素渲染。Pango依赖于其他一些函数库（如Freetype和Fontconfig）来提供实际的字形生成和字体选择，它自己的代码主要用于进行文本布局。

11.1.6 GTK+

最初的GTK函数库是为GIMP（GNU图像处理程序）图像编辑器项目开发的，该项目需要一个带有自由许可证的工具包来替换其最初使用的商业Motif工具包，后者被认为不适合用于该项目。GTK+（即GIMP工具包）是这一努力的最终产品，由于它自身取得相当大的成功，因此它成为了一个单独的项目并被选定为GNOME桌面的基础。从那以后，它经过了大量的开发，尤其是GTK+ 2.0的开发。从2.0版本开始，GTK+的所有版本都是API和ABI兼容的，这允许在不破坏现有应用程序的情况下进行函数库的升级——尽管许多新的应用程序使用了仅在最新版本的GTK+中添加的功能。这一策略至少在GTK+ 3.0的开发之前不会改变，GTK+ 3.0将是GTK+开发中的一个里程碑，但目前还不在开发者的计划日程中。因为目前的GTK+ API相当成功，所以预计在今后一段时间内以兼容API添加的方式来进行升级就足以满足用户的需求。

GTK+的主要特点包括其非常简单易用的基于GObject的API、丰富的构件选择、完整的Unicode支持和高度的可移植性。它被设计为可以很容易地被其他语言绑定，而且现在已有针对许多语言的、成熟的并且很成功的GTK+绑定，包括C++的gtkmm、C#和.NET/Mono的GTK#、Ruby的ruby-gtk，甚至还有针对PHP的GTK+绑定。

虽然也有GNOME特定的用户界面函数库如libgnomeui，但GTK+提供了用于任何GNOME应用程序的大部分构件，并且这个比例将随着开发者逐步地将所有普遍适用的构件移入GTK+以简化平台栈而不断增加。

为了有助于实现用户界面的一致性和实现国际化，GTK+提供了丰富的固化词条（stock item）选择：用于构建菜单和工具栏中常用条目的图标、文字和适当的快捷键，它们将自动遵循当前主题和语言的设置。虽然应用程序的其余部分仍然需要进行翻译，但固化词条轻松地处理了很多通用情况并有助于确保GNOME应用程序用户界面的一致性。

11.1.7 libglade

libglade是一个帮助构建用户界面的函数库。它读取由Glade界面设计程序生成的XML文件并使用它们在程序运行时建立应用程序的用户界面，这避免了必须编写大量代码以用于建立复杂应用程序用户界面的麻烦。它还使得我们可以更容易地在开发过程中改变用户界面，因为布局的调整可以轻松地在Glade中完成，而不需要修改应用程序的代码，除非你需要修改直接由应用程序代码操纵的窗口部件。

由于其非常有用而且很受欢迎，libglade可望被集成到即将到来的GTK+ 2.10版本中^①。而这项工作的开始早在GtkUIManager类被引入GTK+ 2.6时就初见端倪，该类允许基于XML描述来构建菜单系统。

11.1.8 GConf

GConf是GNOME的集中配置系统。它有点类似于Windows的注册表，它所存储的数据数据库中的配置键被应用程序用于保存它们的设置和其他信息。这些信息默认以磁盘上的XML文件树形式表示，在GNU/Linux系统中，其默认全局位置通常位于/usr/share目录中，而每个用户的个人设置在他们自己的家目录中。这些文件并不希望直接被用户编辑修改，但如果有必要，它们可以通过一个文本编辑器进行修改。

访问GConf的主要方式是通过应用程序使用的函数库API，尽管我们也可以使用GNOME桌面提供的配置编辑器来访问GConf。GConf应该只用于保存少量信息——如果应用程序需要保存大量数据，你应该使用自己的数据文件。

GConf还为管理员提供了锁定某些应用程序偏好的功能，这对大型多用户的部署（如一个企业）是非常有用的。使用基于GConf配置的应用程序可以支持这些功能而不需要编写大量的代码。

11.1.9 GStreamer

作为一个流媒体框架，GStreamer是GNOME的音频和视频处理的基础。GStreamer将每个声音片或视频帧作为一个单独的元素处理。它是基于将提供的元素组合到管道中进行处理的想法进行操作的。每个管道提供了必要的处理步骤以播放声音片或显示视频帧。

GStreamer允许程序员轻松地在他们的应用程序中建立支持各种编解码器和输出设备的视频或音频回放。它还可以用来完成媒体的编码、转码和其他任务。元素也可用来检测文件类型、读取元数据和引入各种效果。

在写作本书时，GStreamer的最新稳定版本是0.10系列，它被用在本章和GNOME 2.14版本中。目

^① GTK+ 2.10已于2006年7月3日正式发布，译者翻译本书时GTK+的最新版本是2.12.3。

前GStreamer的API还不稳定，所以本章显示的代码可能需要在GStreamer走向最终的1.0版本并有了一个稳定API之后进行修改。

11.2 建立一个音乐播放器

本章的示例应用程序是一个简单的音乐播放器。虽然它一次只能够打开和播放一个文件，但其完整的程序功能还包括文件类型检测、艺术家和标题元数据的读取以及在音轨中进行搜索的能力。它演示了GTK+编程的基本思想、标准文件选择器和About对话框的使用。在本章的结尾有一些改善这个音乐播放器的建议，对于那些希望更多地了解GTK+编程的人来说，我们相信这是一个很好的起点。

11.2.1 需求

首先，我们需要系统安装有GTK+和GStreamer，以及它们的依赖软件。大多数Linux发行版都提供了它们的软件包，而且它们的源代码也很容易获得并都有一个标准的GNU自动化工具编译过程。本章的代码需要GTK+ 2.8.0或更高的版本（因为它使用了在2.8系列中才引入的一些功能），以及GStreamer 0.10.0或其后与它的API兼容的版本，和一些适当的GStreamer插件用于媒体播放——具体需要哪些插件将根据所用媒体文件格式和系统的声卡驱动而有所不同，但gst-plugins-good软件包将为在大多数系统中播放Ogg Vorbis文件提供所需的一切。

GStreamer的API可能会在0.11版本以及以后的版本中进行修订，但其核心概念不大可能发生变化，因此只需查询更新的GStreamer文档应该就可以对这里的代码进行改写了。

使用二进制软件包的发行版用户需要确保安装了必需的GTK+和GStreamer开发软件包。在许多发行版中，你只需安装gtk+-devel和gstreamer-devel软件包就足够了。

11.2.2 开始：主窗口

解释GTK+编程背后的基本原则的最佳方式是通过示例代码。首先创建一个目录用于音乐播放器项目，然后编辑一个名称为main.h的新文件，它包含如下内容：

```
#ifndef MAIN_H
#define MAIN_H

#include <gtk/gtk.h>

#endif
```

除了包括全局GTK+头文件以外，它没有做任何其他事情，该头文件将引入对GTK+工具包中所有类和函数的声明。另一种方法是只包括工具包中需要的那部分头文件，带有许多源文件的大型项目可能会采取这种做法。因为，如果程序并没有使用工具包中的太多内容，这种做法将减少编译时间。

现在创建源文件main.c。下面的代码已足够创建和显示一个空窗口了：

```
#include "main.h"

static GtkWidget *main_window;

static void
destroy(GtkWidget *widget, gpointer data)
{
```

```

    gtk_main_quit();
}

int main(int argc, char *argv[]) {
    /* Initialise GTK+ */
    gtk_init(&argc, &argv);

    /* Create a window */
    main_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    /* Set main window title */
    gtk_window_set_title(GTK_WINDOW(main_window), "Music Player");

    /* Connect the main window's destroy handler */
    g_signal_connect(G_OBJECT(main_window), "destroy",
        G_CALLBACK(destroy), NULL);

    /* Make everything visible */
    gtk_widget_show_all(GTK_WIDGET(main_window));

    /* Start the main loop */
    gtk_main();

    return 0;
}

```

这几乎是一个GTK+应用程序可以包含的最小代码片段了。关于这个工具包的一些概念将在这里进行介绍。`gtk_init()`设置函数库并处理命令行参数中的GTK+知道的参数，同时修改`argc`和`argv`变量以使得该应用程序的程序员不需要担心已被处理过的命令行参数。

窗口本身是由`gtk_window_new()`创建的。该函数是一个对象构造器，它返回一个指向新`GtkWindow`对象的指针。命名约定在这里非常重要，因为GTK+中的所有对象构造器和方法都要遵循它——名字的第一部分是类名，这里是`GtkWindow`（对于函数名来说，我们将它写为`gtk_window`，以将它和类本身的数据类型区分开来），然后是方法名。`gtk_window_new()`接受一个参数，它是一个常量，它指定了要创建的窗口类型。`GTK_WINDOW_TOPLEVEL`是一个正常的应用程序窗口，它可以通过窗口管理器关闭、最小化和调整大小。

`gtk_window_set_title()`是`GtkWindow`的一个方法，它如你期望的那样设置窗口的标题。对于所有非构造器的方法来说，它们的第一个参数必须是一个有适当类型的对象——它要么是一个该方法所属类的实例，要么是一个继承自该方法所属类的类的实例。`main_window`是一个`GtkWindow`对象，所以它也可以直接传递给`gtk_window_set_title()`。宏`GTK_WINDOW()`确保该函数看到的确实是一个`GtkWindow`对象，如果它不能执行适当的转换，它就将产生一个错误。在本例中，虽然这个宏并不是必需的，但当你想使用这个方法来设置`GtkDialog`的标题时，就需要使用它了。

所有的类都有这样的转换宏，它们在所有的GTK+代码中被广泛地使用。

`g_signal_connect()`设置一个信号处理函数。GTK+工具包的设计中的一个基本概念就是信号和回调函数——回调函数是当其所连接的信号被发出时要调用的函数。信号是针对诸如按钮被点击、鼠标移动、键盘焦点发生变化以及其他许多事件发送的。在这里，我们所讨论的信号是`GtkWindow`的“`destroy`”信号，当X发现窗口将被关闭时（例如，当用户按下标题栏中的关闭按钮）就会发送该信号。

将该信号连接到`destroy()`函数可确保对这个窗口的关闭将导致程序的退出，因为`destroy`函数将调用`gtk_main_quit()`，而后者将如你所期望的那样导致程序的退出。

`gtk_main()`开始GTK+的主循环。它将阻塞直到主循环结束，在用户和应用程序代码之间的任何进一步的交互都将通过信号处理函数的调用来完成。

编译

为了编译这段代码，我们需要创建`Makefile`文件，该文件使用`pkg-config`程序来获取针对GTK+和GStreamer的正确的编译器和连接器标记：

```
CFLAGS=`pkg-config --cflags gtk+-2.0 gstreamer-0.10`
LIBS=`pkg-config --libs gtk+-2.0 gstreamer-0.10`

all: player

player: main.o
        gcc -o player main.o $(LIBS)

%.o: %.c
        gcc -g -c $< $(CFLAGS)
```

编译程序并运行所产生的可执行文件`player`。关闭窗口将根据“`destroy`”信号处理函数的内容导致程序终止。

11.2.3 建立 GUI

现在你有了一个窗口，但是，如果没有其他构件来提供有用的用户界面，它是没有什么用处的。GTK+中的窗口是`GtkBin`的子类，它是一种只能包含一个子构件的容器对象。这听起来可能并不是特别有用，但GTK+还提供了各种可以容纳多个构件的`GtkBin`子类。因此，GTK+中的窗口布局总是在一个`GtkWindow`中添加一个多构件容器来构成的。这些容器中最最有用的是`GtkVBox`、`GtkHBox`和`GtkTable`。`GtkVBox`允许任意数量的构件垂直叠放。`GtkHBox`的功能相同，但提供的是水平排列。`GtkTable`是上述两者的结合，它允许构件在一个灵活的网格中进行排放。

这三种布局构件几乎提供了所有必要的布局功能。需要注意的是，GTK+很少处理对象的绝对位置或大小——容器和其中的构件将调整大小以适应可用空间。不过用户可以改变这类大小调整的一些约束以获得所需的效果，如果你确实需要精确的像素布局，那么`GtkFixed`可作为你最后的选择。

你应该首先选择灵活的盒状模型布局，因为它可以更容易地应付不同的屏幕尺寸和用户可能使用的各种主题。

GUI设计的最终效果看上去类似于图11-1。

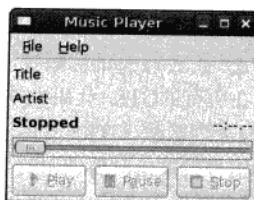


图11-1 GUI设计的最终效果

图11-1显示了在装载文件进行播放之前的音乐播放器窗口。从上往下，依次是菜单栏，用于标题和艺术家元数据的标签，显示播放器当前状态的标签，它的旁边是一个显示经过时间的标签，然后是一个在当前文件中进行搜寻的滑动条，最后是三个用于控制播放的按钮。这个截图表现的是GTK+的ClearLook主题，它也是GNOME 2.14中的默认主题。经过GNOME软件包制作者和最终用户定制的主题可能会导致窗口显示有所不同，但基本的布局是不会改变的。

1. 顶层容器

窗口布局的顶层容器将是一个GtkVBox，因为窗口可看作是一系列构件的依次叠放。你需要为今后将要创建和引用的构件添加一些新的全局变量。

```
static GtkWidget *play_button;
static GtkWidget *pause_button;
static GtkWidget *stop_button;
static GtkWidget *status_label;
static GtkWidget *time_label;
static GtkWidget *seek_scale;
static GtkWidget *title_label;
static GtkWidget *artist_label;
```

在main()之前添加一个用于构建GUI的函数：

```
GtkWidget *build_gui() {
    GtkWidget *main_vbox;
    GtkWidget *status_hbox;
    GtkWidget *controls_hbox;

    /* Create the main GtkVBox. Everything else will go inside here */
    main_vbox = gtk_vbox_new(0, 6);
```

这里创建了一个GtkVBox。gtk_vbox_new的参数影响构件是如何在组装盒中排放的。第一个参数指明组装盒中的构件是否都必须具有相同的高度，一个false值表示构件可能根据它们的需求有不同的高度。第二个参数指明在构件之间的以像素为单位的间距。6个像素是在GNOME人性化界面指南中推荐的间距值。

2. 文本标签

你现在可以添加显示歌曲标题和艺术家名字的标签，这些信息是从歌曲的元数据中提取出来的：

```
title_label = gtk_label_new("Title");
gtk_misc_set_alignment(GTK_MISC(title_label), 0.0, 0.5);
gtk_box_pack_start(GTK_BOX(main_vbox), title_label, FALSE, FALSE, 0);
```

GtkMisc是GtkLabel的父类之一，它包含的机制使你可以控制GtkLabel中的文本的对齐。对齐参数的取值范围从0到1，本例的情况是水平左对齐（0.0）和垂直居中对齐（0.5）。默认情况是水平和垂直都居中对齐（0.5, 0.5），但这样做在本例的窗口布局中并不好看。

gtk_box_pack_start()告诉主GtkVBox将title_label放入组装盒的开始位置（即顶部），当然它会排在它之前使用pack_start放入容器的构件之后。在本章中没有用到的gtk_box_pack_end()是从相反的方向放入构件，如果你希望当包含的容器被扩展时构件可以从容器的中间某处分离，那么可以使用这个函数。

gtk_box_pack_start()的最后三个参数，事实上也是gtk_box_pack_end()的参数表明了被

封装构件分配空间的方式。这三个参数中的第一个参数被称为expand，它影响构件所占空间的扩充。如果有额外的空间分配给包含构件的组装盒，参数expand指这个构件所占的容器段是否可以再分配更多空间。当参数expand被设置为TRUE时，额外的空间将平均分配给所有的构件。第二个参数被称为fill，当有额外空间被分配给构件段时，该参数表明构件是应该扩展以填满该空间还是这个空间由容器进行填白。第三个参数给出构件段之间要保留的填白像素值。

艺术家标签以类似的方式创建：

```
artist_label = gtk_label_new("Artist");
gtk_misc_set_alignment(GTK_MISC(artist_label), 0.0, 0.5);
gtk_box_pack_start(GTK_BOX(main_vbox), artist_label, FALSE, FALSE, 0);
```

还有两个标签：一个显示播放器的状态，一个显示目前音轨的经过时间。因为这两个标签都很短，所以将它们并排放入一个GtkHBox中：

```
status_hbox = gtk_hbox_new(TRUE, 0);
```

注意，对于这个组装盒来说，构件的大小应该相同，所以给这个构造函数传递TRUE值。在本例中，并不需要设置构件之间的间距。

```
gtk_box_pack_start(GTK_BOX(main_vbox), status_hbox, FALSE, FALSE, 0);

status_label = gtk_label_new("<b>Stopped</b>");
gtk_label_set_use_markup(GTK_LABEL(status_label), TRUE);
gtk_misc_set_alignment(GTK_MISC(status_label), 0.0, 0.5);
gtk_box_pack_start(GTK_BOX(status_hbox), status_label, TRUE, TRUE, 0);
```

现在你应该看出来，与手工编码相比，使用libglade来生成GUI非常有吸引力。对于一个与本例的规模相似的项目来说，手工编码还是可以接受的，但对于较大的应用程序来说，这很快就会变得难以接受。请注意，在这个代码中，标签的文本包含了HTML风格的标记。gtk_label_set_use_markup()允许你很容易地在一个标签中显示粗体字。

另一种实现粗体字的方法是修改标签的字体风格，但是，如果你只想要修改一个属性，这种方法似乎就过于复杂了。虽然这种方法使得你不需要在每次设置标签文本时都包括粗体标记，并且当有多个字体属性需要改变时确实也推荐使用它，而且它还可以在整个程序的生命周期中保持不变，但标记更适用于当格式需求的变化比标签文本的更新更频繁的情况，而且它还允许标签为其内容的不同部分显示不同的字体风格。

```
time_label = gtk_label_new("---.--");
gtk_misc_set_alignment(GTK_MISC(time_label), 1.0, 0.5);
gtk_box_pack_start(GTK_BOX(status_hbox), time_label, TRUE, TRUE, 0);
```

这段代码中唯一值得注意的就是，time_label中文本的对齐方式被设置为右对齐，而不是迄今为止我们一直做的左对齐，因为这个标签需要和窗口的右侧对齐。

3. 滑动条控制

下一个要添加到GUI中的构件是滑动条，它用于在音轨中进行搜寻或显示播放进度。这是由GtkHScale构件提供的（正如你可能期望的那样，同样有GtkVScale构件）。这些构件是GtkScale的子类，我们可以在GtkScale中找到大多数比例相关（scale-specific）的方法，而该类又是GtkRange

的一个子类，GtkRange包含了其他与维持取值范围和当前位置相关的有用功能。

```
seek_scale = gtk_hscale_new_with_range(0, 100, 1);
gtk_scale_set_draw_value(GTK_SCALE(seek_scale), FALSE);
gtk_range_set_update_policy(GTK_RANGE(seek_scale),
    GTK_UPDATE_DISCONTINUOUS);
gtk_box_pack_start(GTK_BOX(main_vbox), seek_scale, FALSE, FALSE, 0);
```

HScale及其同类vScale有许多不同的构造函数。这里选择的构造函数在指定的范围内建立构件，其他构造函数使用默认范围，还有一些构造函数将构件与现有的GtkRange对象相关联。这里使用的是从0~100的范围，它允许我们将当前曲目的播放进度以百分比的方式表示，而不需要在打开一个长度不同的曲目时改变范围参数。第三个参数表示范围的粒度——这里使用的单位是非常适合的。

gtk_scale_set_draw_value()用于告诉比例构件不要在滑块的旁边以数字形式显示其当前位置。最后一个设定修改的是构件的更新方式，它指定何时比例构件会发送值改变信号以表明用户拖动了滑块或其他方式操纵了滑块。GTK_UPDATE_DISCONTINUOUS指定只有当用户结束拖动时才发送该信号。其他选项还包括在用户拖动时发送信号，这更适合用于颜色选择器中的滑动条。

4. 控制按钮

这个GUI的最后一部分是封装了播放控制按钮的HBox。

```
controls_hbox = gtk_hbox_new(TRUE, 6);
gtk_box_pack_start_defaults(GTK_BOX(main_vbox), controls_hbox);
```

gtk_box_pack_start_defaults()简单地将程序员从必须提供三个用于调整大小和间隔的参数中解放出来，如果他们对默认值感到满意，就无需提供这三个参数，而在本例中，默认值表现得已经很好了。

```
play_button = gtk_button_new_from_stock(GTK_STOCK_MEDIA_PLAY);
gtk_widget_set_sensitive(play_button, FALSE);
gtk_box_pack_start_defaults(GTK_BOX(controls_hbox), play_button);
```

上面的代码创建了第一个按钮。GTK+ 2.8针对常见的媒体播放功能（如播放、暂停和停止）引入了固化词条，它们可被用于自动建立带有正确的图标、标题和提示的按钮。更让人高兴的是，固化词条将自动使用用户的语言设置（假设已安装了相应的翻译文件）。这意味着，如果你想让你的应用程序本地化，你只需要在程序中翻译任何非固化的函数即可。对于大型应用程序来说，虽然依旧有大量的内容需要翻译，但通过使用固化词条，可以确保至少常见的功能都会被翻译。固化词条还将遵循用户的图标主题偏好，这将使得你的应用程序更容易融入周围的桌面环境。

按钮的“敏感”属性被设置为FALSE，这将导致它显示为灰色并且不能响应鼠标点击或键盘事件。当程序刚启动时，这个状态对于按钮来说很适合，因为在用户选择文件之前还没有文件可以播放。

其余的按钮都是以类似的方式建立的，但为了使在曲目暂停时暂停按钮能停留在按下的状态，它使用的是GtkToggleButton。GTK+使程序员必须依命行事。GTK+ 2.8中并没有gtk_toggle_button_new_from_stock()构造函数，所以按钮在被构造时需要首先将其标签设置为所需的固化ID，然后必须调用gtk_button_set_use_stock()（继承自GtkButton）将按钮设置为固化按钮。这也是GtkButton的new_from_stock()构造函数所使用的过程，实际上这个构造函数只是一个便于使用的包裹函数而已。

```

pause_button =
    gtk_toggle_button_new_with_label(GTK_STOCK_MEDIA_PAUSE);
gtk_button_set_use_stock(GTK_BUTTON(pause_button), TRUE);
gtk_widget_set_sensitive(pause_button, FALSE);
gtk_box_pack_start_defaults(GTK_BOX(controls_hbox), pause_button);

stop_button = gtk_button_new_from_stock(GTK_STOCK_MEDIA_STOP);
gtk_widget_set_sensitive(stop_button, FALSE);
g_signal_connect(G_OBJECT(stop_button),
    "clicked",
    G_CALLBACK(stop_clicked),
    NULL);
gtk_box_pack_start_defaults(GTK_BOX(controls_hbox), stop_button);

```

这个函数最后返回一个GtkVBox。与所有Gobject对象一样，GTK+构件是引用计数的，由于我们使用指针来访问它们，所以对象本身是不会被销毁的，除非它的引用计数变为0。大多数时候，引用计数的具体减少情况并不是太重要，因为大多数构件都会在应用程序关闭时被销毁。但是，如果构件在程序运行时被创建和销毁，我们确实需要小心，因为错误的引用计数可能会导致内存泄漏和段错误。

```

    return main_vbox;
}

```

在main()函数中的gtk_widget_show_all()调用之前添加如下一行：

```
gtk_container_add(GTK_CONTAINER(main_window), build_gui());
```

上面的语句将包含窗口布局的VBox放入窗口中。已有的gtk_widget_show_all()调用将显示它，在编译和运行该程序后，应该可以看到大部分用户界面了。

5. 菜单

这个程序还缺少一个组成部分：菜单栏。传统的UI设计观点认为在本例这样的小程序中，菜单栏是没有什么用处的，但我们想通过它来演示如何使用GTK+的方法来构建菜单。

建立菜单的方法有好几种，但对于最新版本的GTK+来说，使用Gtk.UIManager对象来构建基于动作的菜单和工具栏是首选方案。Gtk.UIManager有点类似于libglade，尽管它在一个更加受限的范围内，它通过由程序员提供的XML来生成菜单结构。每个菜单项都关联一个GtkAction对象，该对象描述菜单项的标题、图标和提示，以及当它被点击时应该做什么。GtkAction对象还可用于生成工具栏按钮，当菜单项的启用/禁用状态必须与工具栏中相应项保持一致时，该对象能提供很多便利，因为它的状态改变会自动更新相应菜单和工具栏项目的状态。

我们的第一项任务是描述我们将创建的菜单项的行为。在build_gui()函数定义之前添加一个GtkActionEntry结构的数组：

```

static GtkActionEntry mainwindow_action_entries[] = {
    { "FileMenu", "NULL", "_File" },
    {
        "OpenFile",
        GTK_STOCK_OPEN,
        "_Open",
        "<control>O",
        "Open a file for playback",
    }
};

```

```

    G_CALLBACK(file_open)
},
{
    "QuitPlayer",
    GTK_STOCK_QUIT,
    "_Quit",
    "<control>Q",
    "Quit the music player",
    G_CALLBACK(file_quit)
},
{
    "HelpMenu", "NULL", "_Help" ),
{
    "HelpAbout",
    GTK_STOCK_ABOUT,
    "_About",
    "",
    "About the music player",
    G_CALLBACK(help_about)
}
);

```

这里定义了两种不同类型的ActionEntry。第一种类型代表的是菜单而不是菜单项——在本例中是FileMenu和HelpMenu。它们只提供菜单的标题。其他成员则更为详细，并通过动作名、图标、标签、键盘快捷键、提示和回调函数来分别表示菜单项。标签中，一个字母前的下划线表示在显示标签时将在这个字母下加上下划线，当使用键盘浏览菜单时，可以使用该字母来激活这个菜单项。

6. 键盘快捷键

键盘快捷键是使用GTK+的快捷键语法指定的。在本例中，所有的快捷键都是在按住Ctrl键的情况下再按下一个简单的字母。提示是一个简单的字符串，而回调函数和你之前已看到的回调函数没有什么不同。

请注意，这些回调说明符中引用的函数目前还不存在。我们将很快在程序中添加它们，但首先我们需要在build_gui()函数中添加两个变量：

```

GtkActionGroup *actiongroup;
GtkUIManager *ui_manager;

```

GtkActionGroup对象用于维护由UIManager使用的一组GtkActions。构造ActionGroup和UIManager的代码应位于build_gui()中任何其他构件的构造代码之前，因为菜单栏位于窗口的顶部，而且将构件及其相关对象的构造顺序安排得和它们在窗口中出现的顺序大致相同也有助于代码的维护。

```

actiongroup = gtk_action_group_new("MainwindowActiongroup");
gtk_action_group_add_actions(actiongroup,
    mainwindow_action_entries,
    G_N_ELEMENTS(mainwindow_action_entries),
    NULL);

ui_manager = gtk_ui_manager_new();
gtk_ui_manager_insert_action_group(ui_manager, actiongroup, 0);
gtk_ui_manager_add_ui_from_string(ui_manager,
    "<ui>\n"
    "  <menubar name='MainMenu'>\n"

```

```

    "      <menu action='FileMenu'>
    "        <menuitem action='OpenFile' />
    "        <separator name='fsep1' />
    "        <menuitem action='QuitPlayer' />
    "      </menu>
    "      <menu action='HelpMenu'>
    "        <menuitem action='HelpAbout' />
    "      </menu>
    "    </menubar>
  "</ui>",
  -1,
  NULL);

```

这应该是相当简单的，虽然它看起来与你看到的所有构件构造代码非常不同。GtkActionGroup使用我们之前创建的数组进行初始化，然后它被关联到UI管理器。G_N_ELEMENTS()是一个Glib宏，它用于确定一个数组中的元素数。用户界面的XML描述只是一个结构的规范，它说明了每个动作所对应的菜单项类型。Gtk.UIManager还提供了直接从文件中装载这类结构的函数。

当创建main_vbox之后，给它添加菜单栏：

```

gtk_box_pack_start(
  GTK_BOX(main_vbox),
  gtk_ui_manager_get_widget(ui_manager, "/ui/MainMenu"),
  FALSE, FALSE, 0);

```

GTK+对菜单栏的处理方式与处理任何其他构件一样，所以你可以简单地将它封装到VBox的顶部。用户通过指定一个特定的路径来要求UI管理器返回菜单栏构件。通过使用类似的路径，我们可以获取由UI管理器建立的任何构件。如果你想获取指向某个菜单项的指针以对该菜单项进行手工修改，那么这种方法是非常方便的。

此时，你可能已注意到，我们在这里看到的在XML中定义菜单结构和本章开始处对libglade功能的描述之间的相似性。Gtk.UIManager可以被看作是将libglade的功能整合到GTK+中的一个起点，而后者正是目前GTK+ 2.10要做的事情。

7. 回调函数

显然，如果我们不将UI构件连接到一些回调函数，这个程序就不能做太多的事情。此时，这个程序还不能执行其全部功能，因为程序还没有提供音频播放代码，但我们可以添加所需的回调函数的骨架并将它们连接起来。我们现在完全可以实现Help→About菜单项的完整功能，它的代码如下所示。

菜单结构中的动作所需的回调函数返回类型为void，并以一个GtkAction指针作为参数。当回调函数被调用时，这个参数就调用回调函数的动作。通过使用该参数，程序员可以将同一个回调函数关联到多个动作并根据是哪个动作触发了该函数而有不同的表现。

```

static void file_open(GtkAction *action) {
  /* We will write this function later */
}

static void file_quit(GtkAction *action) {
  gtk_main_quit();
}

```

```

static void help_about(GtkAction *action) {
    GtkWidget *about_dialog = gtk_about_dialog_new();

    gtk_about_dialog_set_name(GTK_ABOUT_DIALOG(about_dialog), "Music Player");
    gtk_about_dialog_set_version(GTK_ABOUT_DIALOG(about_dialog), "0.1");
    gtk_about_dialog_set_copyright(GTK_ABOUT_DIALOG(about_dialog), "Copyright 2006,
The Author");

    gtk_dialog_run(GTK_DIALOG(about_dialog));
}

```

在help_about函数中，你第一次见到了GtkDialog的使用。GtkAboutDialog是一个简单的对象，它提供了一个非常适用于“关于这个应用程序”功能的完整对话框。应用程序的名称、版本和版权都通过一个简单的方法调用进行设置，然后我们调用gtk_dialog_run()来显示它。你还会在后面实现file_open()回调函数时遇到gtk_dialog_run()。

```

static void play_clicked(GtkWidget *widget, gpointer data) {

static void pause_clicked(GtkWidget *widget, gpointer data) {

static void stop_clicked(GtkWidget *widget, gpointer data) {
}

```

这三个回调函数提供播放、暂停和停止按钮的功能，所以我们需要回到build_gui()上，并将它们连接到按钮的clicked信号。

```

g_signal_connect(G_OBJECT(play_button), "clicked",
    G_CALLBACK(play_clicked), NULL);
g_signal_connect(G_OBJECT(pause_button), "clicked",
    G_CALLBACK(pause_clicked), NULL);
g_signal_connect(G_OBJECT(stop_button), "clicked",
    G_CALLBACK(stop_clicked), NULL);

```

现在编译并运行这个程序，Help→About菜单项应该可以正常工作了，File→Quit菜单项同样如此。

8. GStreamer的使用

现在GUI的大部分内容都已做好，我们可以向程序中添加播放音乐的功能了。这个功能将由GStreamer多媒体框架提供。

GStreamer的操作需要应用程序的开发者创建管道。每个管道由一组元素组成，每个元素都执行一个特定功能。通常情况下，一个管道以某种类型的源元素开始，这可能是被称为filesrc的元素，它从磁盘上读取文件并提供该文件的内容，也可能是通过一个网络连接提供缓冲数据的元素，甚至可能是一个视频捕捉设备获取数据的元素。然后，管道中是一些其他类型的元素，如解码器（用于将声音文件转换为处理所需的标准格式）、分离器（用于从一个声音文件中分解出多个声道）或其他类似的处理器。管道以一个输出元素结束，它可以是从一个文件写入器到一个高级Linux音频体系结构(ALSA)音频输出元素或一个基于Open GL的视频播放元素的任何元素。这些输出元素被称为“sink”(接收器)。

因此，一个使用ALSA播放磁盘上的Ogg Vorbis音频文件的典型管道将依次包含一个filesrc元

素、一个vorbisdecoder元素和一个alsasink元素。

元素是通过“pad”(衬垫)连接在一起的。每个衬垫都对应一组能力，它表明了衬垫的用途。有些衬垫提供某一特定类型的数据——例如，来自一个文件中的原始数据或来自一个解码器元素中的解码音频。有些衬垫接受某一特定类型或种类的数据，而还有一些衬垫提供元数据或同步信息。

为了简化我们必须编写的代码，我们将利用由GStreamer 0.10提供的一个被称为playbin的便利元素。这是一个高级元素，它实际上是一个预建立的管道。通过使用GStreamer的文件类型检测功能，它可以从任何指定的URI读取数据，并确定合适的解码器和输出接收器来正确地播放它。在本例中，这意味着它可以识别和正确地解码在GStreamer中有相应插件的任何音频文件(你可以通过在终端上运行命令gst-inspect-0.10来列出GStreamer 0.10中的所有插件)。

所有的GStreamer代码将放在另一个文件中。首先创建playback.h头文件，如下所示：

```
#ifndef PLAYBACK_H
#define PLAYBACK_H

#include <gst/gst.h>

gboolean load_file(const gchar *uri);
gboolean play_file();
void stop_playback(void);
void seek_to(gdouble percentage);

#endif
```

这个头文件提供了main.c调用它所需功能的接口。修改main.c以包括playback.h，然后创建playback.c，该源文件应该包括playback.h和main.h。还需要更新Makefile，从而将playback.o连接到最终可执行文件。下面是playback.c文件的主要内容。

```
#include "main.h"
#include "playback.h"

static GstElement *play = NULL;
static guint timeout_source = 0;

gboolean load_file(const gchar *uri) {
    if (build_gst_pipeline(uri)) return TRUE;
    return FALSE;
}
```

函数build_gst_pipeline()以一个URI为参数，并构建playbin元素，指向该元素的指针被保存在变量play中，以备后用。

```
static gboolean build_gst_pipeline(const gchar *uri) {
    /* If there is already a pipeline, destroy it */
    if (play) {
        gst_element_set_state(play, GST_STATE_NULL);
        gst_object_unref(GST_OBJECT(play));
        play = NULL;
    }
```

```

/* Create a playbin element */
play = gst_element_factory_make("playbin", "play");
if (!play) return FALSE;
g_object_set(G_OBJECT(play), "uri", uri, NULL);

/* Connect the message bus callback */
gst_bus_add_watch(gst_pipeline_get_bus(GST_PIPELINE(play)), bus_callback, NULL);

return TRUE;
}

```

请注意，这个代码现在还不能编译，因为还缺少一个bus_callback()函数。我们很快就会来补充这个函数。

build_gstreamer_pipeline()是一个相当简单的函数。它首先检查play变量是否为NULL，如果它不是，则表明已有一个playbin元素。如果是这样，就调用gst_object_unref()以减少playbin的引用计数。因为在这个代码中playbin只有一个引用，所以减少它的引用计数将导致playbin被销毁。然后我们将play设置为NULL以表明现在没有可用的playbin。

我们通过调用gst_element_factory_make()来构建playbin元素，该函数是一个可以构建任何GStreamer元素的通用构造函数。它的第一个参数指定要构建的元素名。GStreamer使用字符串名称来确定元素类型，从而方便添加新元素。如果需要，一个程序可以从配置文件或用户那里接受元素名称并使用新的元素而不需要重新编译程序来包括定义这些元素名的头文件。只要指定的元素有正确的能力（这可以在程序运行时进行检查），它们就可以完美地操作而不需要改变任何代码。在本例中，构建了一个playbin元素并将它命名为play，后者就是gst_element_factory_make()的第二个参数。元素名称在程序的其余部分不再使用，但它对识别一个复杂管道中的元素确实有其用处。

然后，代码将检查gst_element_factory_make()返回的指针是否有效以确定元素是否被正确构建。如果是，就调用g_object_set()将playbin元素的标准GObject特性uri设置为要播放文件的URI。GStreamer元素广泛使用特性来配置它们的行为，不同元素可用的特性也有所不同。

最后，gst_bus_add_watch()连接一个用于侦听管道消息总线的回调函数。GStreamer为管道和应用程序之间的通信使用了一个消息总线。通过提供这个机制，运行在不同线程中的管道（如playbin）可以传递消息给应用程序而不需要该程序的作者担心跨线程的数据同步问题。消息和命令使用类似的封装通过另一个途径进行传递。

为了使用这个回调函数，当然需要定义它。当它被调用时，GStreamer为它提供一个触发该回调函数的GstBus对象、一个包含被发送消息的GstMessage对象和一个用户提供的指针，在本例中，我们没有使用该指针。

```

static gboolean bus_callback (GstBus *bus,
    GstMessage *message, gpointer data)
{
    switch (GST_MESSAGE_TYPE (message)) {
        case GST_MESSAGE_ERROR: {
            GError *err;
            gchar *debug;

            gst_message_parse_error(message, &err, &debug);
            g_print("Error: %s\n", err->message);
        }
    }
}

```

```

g_error_free(err);
g_free(debug);

gtk_main_quit();
break;
}
}

```

错误处理代码非常简单，它打印错误信息并终止程序。在一个更成熟的应用程序中，我们应采用更智能的错误处理技术，即根据所遇错误的确切性质采取不同的处理方法。错误消息本身是GError对象的一个使用示例，该对象是由Glib提供的一个通用错误描述对象。

```

case GST_MESSAGE_EOS:
    stop_playback();
    break;
}

```

EOS消息表明管道已到达当前流的结尾。在本例中，它将调用stop_playback()函数，该函数将在后面进行定义。

```

case GST_MESSAGE_TAG: {
    /* The stream discovered new tags. */
    break;
}

```

TAG消息表明GStreamer在数据流中遇到了元数据，如标题或艺术家信息。这种情况的处理我们也在后面实现，虽然对于实际播放文件这个任务来说它是微不足道的。

```

default:
    /* Another message occurred which we are not interested in
     * handling. */
    break;
}

```

默认情况下，将简单地忽略没有进行明确处理的任何消息。GStreamer会生成大量的消息，但对于像本例这样简单的音频播放程序来说，只有极少数消息需要我们处理。

```

    return TRUE;
}

```

最后，这个函数返回TRUE以表明它已对消息进行了处理，不需要再采取进一步的行动了。

为了完成这个函数的功能，你还需要定义stop_playback()函数，它将设置GStreamer管道的状态并进行适当的清理。要理解该函数，你首先需要定义函数play_file()，它做的事情可能与你期望的差不多。

```

gboolean play_file() {

    if (play) {
        gst_element_set_state(play, GST_STATE_PLAYING);
    }
}

```

元素状态GST_STATE_PLAYING表明一个正在播放数据流的管道。将元素的状态改变为该状态将启

动管道的播放，如果播放已经开始，则它是一个空操作。元素的状态将控制管道对数据流的处理，所以你还可能会遇到诸如GST_STATE_PAUSED这样的状态，它的功能应该是不言自明的。

```
timeout_source = g_timeout_add(200,
    (GSourceFunc)update_time_callback, play);
```

`g_timeout_add()`是一个Glib函数，它在Glib主循环中添加一个超时处理函数。回调函数`update_time_callback`将每200毫秒被调用一次，其参数为指针`play`。这个函数用于获取播放的进度并对GUI进行相应的更新。`g_timeout_add()`返回超时函数的一个数字ID，它可以在今后被用于对该函数进行删除或修改。

```
    return TRUE;
}

return FALSE;
}
```

如果开始播放了，这个函数就返回TRUE，否则返回FALSE。

现在，除了缺少`update_time_callback()`的定义以外，我们可以开始定义`stop_playback()`函数了，它给予程序启动和停止文件播放的能力——虽然GUI现在还不能提供文件URI给播放代码。

```
void stop_playback() {
    if (timeout_source) g_source_remove(timeout_source);
    timeout_source = 0;
```

这个函数使用保存的超时ID从主循环中删除超时函数，因为我们没有必要在不播放文件时每秒钟调用更新函数5次，因此也不需要使用这个超时函数了。

```
if (play) {
    gst_element_set_state(play, GST_STATE_NULL);
    gst_object_unref(GST_OBJECT(play));
    play = NULL;
}
}
```

管道被停用并销毁。GST_STATE_NULL导致管道停止播放并自行重置，释放它可能持有的任何资源，如播放缓冲或音频设备上的文件句柄。

回调函数使用`gst_element_query_position()`和`gst_element_query_duration()`来更新GUI的时间。这两个方法以一种指定的格式获取一个元素的位置和数据流的持续时间。这里使用的是标准的GStreamer时间格式，它以高精度的整数显示数据流中的精确位置。

这两个方法在成功时将返回并把获取的值放入提供的地址中。为了将时间格式化为一个字符串以显示给用户，我们使用了`g_snprintf()`。它是Glib版本的`snprintf()`，提供它是为了确保即便在没有`snprintf()`的系统中也具备可移植性。`GST_TIME_ARGS()`是一个宏，它将位置转换为适用于`printf()`风格函数的参数。

```
static gboolean update_time_callback(GstElement *pipeline) {
    GstFormat fmt = GST_FORMAT_TIME;
    gint64 position;
```

```

gint64 length;
gchar time_buffer[25];

if (gst_element_query_position(pipeline, &fmt, &position)
    && gst_element_query_duration(pipeline, &fmt, &length))
{
    g_snprintf(time_buffer, 24,
               "%u.%02u", GST_TIME_ARGS(position));
    gui_update_time(time_buffer, position, length);
}

return TRUE;
}

```

这个函数还调用了一个新函数gui_update_time()。我们将这个新函数添加到main.c的GUI代码中，并在main.h中放入合适的声明以允许playback.c中的代码调用它。gui_update_time()以格式化时间字符串、位置和长度作为参数，并更新GUI中的构件。

```

void gui_update_time(const gchar *time,
                     const gint64 position, const gint64 length)
{
    gtk_label_set_text(GTK_LABEL(time_label), time);
    if (length > 0) {
        gtk_range_set_value(GTK_RANGE(seek_scale),
                           ((gdouble)position / (gdouble)length) * 100.0);
    }
}

```

不用说，当播放引擎中有事件发生时，GUI中还有其他构件需要更新。为了确保能把所有这些更新代码放在一起，我们编写了函数gui_status_update()：

```

void gui_status_update(PlayerState state) {
    switch (state) {
        case STATE_STOP:
            gtk_widget_set_sensitive(GTK_WIDGET(stop_button), FALSE);
            gtk_widget_set_sensitive(GTK_WIDGET(pause_button), FALSE);
            gtk_label_set_markup(GTK_LABEL(status_label), "<b>Stopped</b>");
            gtk_range_set_value(GTK_RANGE(seek_scale), 0.0);
            gtk_label_set_text(GTK_LABEL(time_label), "--:----");
            break;
        case STATE_PLAY:
            gtk_widget_set_sensitive(GTK_WIDGET(stop_button), TRUE);
            gtk_widget_set_sensitive(GTK_WIDGET(pause_button), TRUE);
            gtk_label_set_markup(GTK_LABEL(status_label), "<b>Playing</b>");
            break;
        case STATE_PAUSE:
            gtk_label_set_markup(GTK_LABEL(status_label), "<b>Paused</b>");
            break;
        default:
            break;
    }
}

```

针对STOP、PAUSE和PLAY这三个状态中的每一个，该函数相应地设置GUI不同部分的状态。这里使用的唯一一个新方法是gtk_label_set_markup()，它改变标签的文本并确保标签的use_markup特性被启用。我们在main.h中添加一个gui_status_update()的声明，并在playback.c中的播放开始和停止的位置添加对它的调用。

9. 打开文件

现在这个段码基本上已准备好开始播放文件了，剩下来的工作就是让用户能够自己选择他（她）希望播放的文件。为了完成这项工作，一个显而易见的选择就是使用GtkFileChooserDialog构件，它是一个用来打开和保存文件的完整对话框。它还有一个模式可以用来打开目录，但在本例中，你只需要打开文件。

```
static void file_open(GtkAction *action) {
    GtkWidget *file_chooser = gtk_file_chooser_dialog_new(
        "Open File", GTK_WINDOW(main_window),
        GTK_FILE_CHOOSER_ACTION_OPEN,
        GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
        GTK_STOCK_OPEN, GTK_RESPONSE_ACCEPT,
        NULL);
```

需要对这个构造函数进行解释。它的第一个参数指定要显示给用户的窗口的标题。第二个参数指定这个对话框的父窗口，这个参数有助于窗口管理器正确地布局和连接窗口。在本例中，其父窗口显然为main_window——它是应用程序中的唯一的一个其他窗口，而且显示FileChooserDialog的命令也是在该窗口中调用的。GTK_FILE_CHOOSER_ACTION_OPEN表明FileChooser应该允许用户选择要打开的文件。如果在这里指定一个不同的动作将极大地改变对话框的外观和功能，比如GNOME的保存对话框（GTK_FILE_CHOOSER_ACTION_SAVE）与其对应的打开文件的对话框之间的差别是相当大的。

接下来的四个参数指定要在对话框中使用的按钮以及它们的响应ID，如果这个程序运行在一个从左向右书写的语言（如英语）系统中，这些按钮将以从左向右的顺序排列（如果是在一个从右向左的本地环境中，GTK+将自动使一些窗口布局反转）。这种排序方法与GNOME人性化界面指南的要求是一致的，代码首先指定一个固化的“cancel”按钮，然后是一个固化的“open”按钮。最后一个参数NULL表明对话框中没有更多的按钮了。

响应ID非常重要，因为它们都是按钮被按下时返回的值。由于我们使用gtk_dialog_run()来调用该对话框，所以程序将阻塞直到该对话框返回——即直到用户选择一个按钮或按下一个执行相同功能的键盘快捷键以关闭对话框为止。

如果你想实现非模态（nonmodal）对话框，请记住GtkDialog是我们熟悉的GtkWindow的一个子类，通过手工处理一些事件（特别是点击按钮）就可以实现非模态对话框。gtk_dialog_run()的返回值是被点击按钮的响应ID（GTK_RESPONSE_ACCEPT被GTK+看作为默认按钮的响应ID，所以带有该响应ID的按钮就成为用户按下回车键时触发的按钮）。因此，打开文件的代码只需要在对话框返回GTK_RESPONSE_ACCEPT时运行：

```
if (gtk_dialog_run(GTK_DIALOG(file_chooser)) == GTK_RESPONSE_ACCEPT)
{
    char *filename;
    filename = gtk_file_chooser_get_uri(GTK_FILE_CHOOSER(file_chooser));
```

我们知道用户将选择一个文件，该文件的URI可以通过包含在FileChooserDialog中的FileChooser构件获取。虽然我们可以只获取它的UNIX文件路径，但由于playbin期望使用一个URI，所以我们坚持使用同一种格式会使得文件的处理更加方便。请注意，这个URI的格式可能并不是file://，当系统中运行着GNOME时，GTK+的FileChooser将使用GNOME的函数库来增强其能力，其中包括gnome-vfs（虚拟文件系统层）。因此，在某些情况下，GtkFileChooser可能会提供位于网络中或其他文件中的文档的URI。一个真正的gnome-vfs兼容应用程序可以处理这类URI而不会有任何问题——事实上，在这个应用程序中使用playbin意味着一些网络URI也许可以被正确地处理，但这取决于你的系统配置。

```
g_signal_emit_by_name(G_OBJECT(stop_button), "clicked");
```

因为要打开一个新的文件，代码需要确保所有的现有文件不再继续播放。完成这一工作的最简单方法就是假装用户点击了停止按钮，所以使用上面的代码让停止按钮发送它的clicked信号。

然后，当前URI的本地拷贝将被更新，接着调用load_file()以准备要播放的文件：

```
if (current_filename) g_free(current_filename);
current_filename = filename;
if (load_file(filename))
    gtk_widget_set_sensitive(GTK_WIDGET(play_button), TRUE);
}

gtk_widget_destroy(file_chooser);
}
```

最后，我们使用gtk_widget_destroy()销毁FileChooser。

这个程序现在可以编译、运行和播放一个音乐文件了。但还有两件事没做：元数据和搜索。

10. 元数据

元数据很简单。首先，应该提供一个接口方法使播放代码可以改变GUI中的元数据标签。这需要在main.h中添加一个声明并在main.c中添加它的定义，如下所示：

```
void gui_update_metadata(const gchar *title, const gchar *artist) {
    gtk_label_set_text(GTK_LABEL(title_label), title);
    gtk_label_set_text(GTK_LABEL(artist_label), artist);
}
```

这段代码本身非常简单。正如你可能已经意识到的那样，如果消息类型是GST_MESSAGE_TAG，该消息应该从播放器的消息处理中被调用。在本例中，GstMessage对象包含一个标签消息，可以使用该消息具备的几个方法来提取用户感兴趣的信息。代码如下所示：

```
case GST_MESSAGE_TAG: {
    GstTagList *tags;
    gchar *title = "";
    gchar *artist = "";
    gst_message_parse_tag(message, &tags);
    if (gst_tag_list_get_string(tags, GST_TAG_TITLE, &title)
        && gst_tag_list_get_string(tags, GST_TAG_ARTIST, &artist))
        gui_update_metadata(title, artist);
    gst_tag_list_free(tags);
    break;
}
```

标签到达GstMessage并封装在一个GstTagList对象中，我们可以通过gst_message_parse_tag()来提取该对象。这将生成GstTagList的一个新拷贝，所以千万不要忘记在不需要它时使用gst_tag_list_free()释放它。如果不这样做，可能会导致相当严重的内存泄漏。

一旦我们从message中提取出来了标签列表，使用gst_tag_list_get_string()从tags中提取标题和艺术家标签就是一件相当简单的事情了。GStreamer提供了预定义的常量来提取标准的元数据域，当然你也可以提供任意的字符串来提取媒体中可能包含的其他域。gst_tag_list_get_string()在成功找到请求的标签值时返回true，否则返回false。如果两个调用都成功了，gui_update_metadata将使用新值来更新GUI。

11. 搜索

要允许在文件中进行搜索，最理想的情况是允许用户点击seek_scale的滑块并将它拖动到一个新位置，从而让数据流立刻改变其播放位置。幸运的是，这正是GStreamer允许你实现的功能。当用户改变GtkScale构件的值时，它将发送一个value-changed信号。我们将一个回调函数连接到这个信号：

```
g_signal_connect(G_OBJECT(seek_scale),
    "value-changed", G_CALLBACK(seek_value_changed), NULL);
```

接着我们在main.c中定义这个回调函数：

```
static void seek_value_changed(GtkRange *range, gpointer data) {
    gdouble val = gtk_range_get_value(range);

    seek_to(val);
}
```

seek_to()使用一个百分比数字作为其参数，它表示用户想要搜索的位置离数据流的开始有多远。这个函数在playback.h中声明并在playback.c中定义，如下所示：

```
void seek_to(gdouble percentage) {
    GstFormat fmt = GST_FORMAT_TIME;
    gint64 length;

    if (play && gst_element_query_duration(play, &fmt, &length)) {
```

首先，该函数将检查是否有一个有效的管道。如果有而且可以成功获取当前数据流的持续时间，它将根据这个持续时间和用户提供的百分比来计算用户想要搜索的位置的GStreamer时间值。

```
    gint64 target = ((gdouble)length * (percentage / 100.0));
```

实际的搜索是通过gst_element_seek()调用完成的。

```
if (!gst_element_seek(play, 1.0, GST_FORMAT_TIME,
    GST_SEEK_FLAG_FLUSH, GST_SEEK_TYPE_SET,
    target, GST_SEEK_TYPE_NONE, GST_CLOCK_TIME_NONE))
    g_warning("Failed to seek to desired position\n");
}
```

gst_element_seek()函数使用几个参数来定义搜索。幸运的是，对于默认行为来说，大多数参数可以使用预定义的函数库常量来设置。这些参数设置了元素的格式和类型，以及搜索的终止时间和

类型。我们唯一需要提供的参数是接收事件的元素（变量play）和搜索的时间值（变量target）。

因为gst_element_seek()在成功时返回true，所以上面的代码检查它是否返回一个false值。如果是，就打印一个消息表示搜索失败。对用户来说，这虽然没有什么实际用途，但你可以很容易想到提供一个更有帮助的信息，尤其当你要查询管道以检查其实际状态时更是如此。

增加了搜索功能之后，这个音乐播放器声明的功能基本上完成了。剩下的暂停功能和我们在下面一节中提出的一些改进建议就作为留给读者的一些练习吧。

不幸的是，这段代码在执行搜索时有重大的缺陷：如果当用户在拖动滑块时seek_scale的位置被播放引擎更新了，滑块的位置就将产生跳跃。为了避免这种情况的发生，你需要阻止播放代码在用户进行拖动时更新滑动条。因为播放代码是通过调用gui_update_time()来完成这一工作的，所以该限制可以完全放在GUI代码中。我们首先在main.c的顶部增加一个新的标记变量：

```
gboolean can_update_seek_scale = TRUE;
```

修改gui_update_time()函数，使得它只有在can_update_seek_scale为true时才更新seek_scale的位置。而时间标签的更新应该保持不变，因为这不会引起任何问题，而且当用户在音轨中拖动滑块进行搜索时，通过一些显示以表明音乐正在继续播放也是有用的。

为了确保这个变量被正确设置，它需要在用户开始和停止拖动滑块时被更新。这可以通过使用由GtkWidget类所提供的事件来完成，该类是seek_scale构件所属类的祖先。当用户在鼠标指针经过构件时按下鼠标按钮就将触发button-press-event。当在button-press-event中按下的按钮被释放时就会触发button-release-event——即使用户已移动鼠标指针从而离开该构件时也是如此。这确保不会遗漏按钮释放事件。你已遇到过的clicked信号是这两个事件的结合，它是在构件观察到鼠标主按钮的按下和释放后触发的。

我们针对seek_scale的按钮按下和释放事件编写一些信号处理函数。它们都很简明易懂：

```
gboolean seek_scale_button_pressed(
    GtkWidget *widget, GdkEventButton *event, gpointer user_data) {
    can_update_seek_scale = FALSE;
    return FALSE;
}

gboolean seek_scale_button_released(
    GtkWidget *widget, GdkEventButton *event, gpointer user_data) {
    can_update_seek_scale = TRUE;
    return FALSE;
}
```

每个函数都相应地更新标记变量，然后返回FALSE。如果一个信号的回调函数原型返回gboolean值，那么该回调函数通常使用这个返回值来表明它是否已完全处理了这个信号。返回TRUE告诉GTK+这个信号已完全处理了，而不需要针对该信号再执行更多的信号处理函数。返回FALSE则允许该信号被继续传播给其他信号处理函数。

在本例中，返回FALSE将允许构件的默认信号处理函数也处理这个信号，从而保留构件的行为。返回TRUE将阻止用户调整滑块的位置。

以这种方式工作的信号通常针对的都是与鼠标按钮和移动相关的事件，而不像clicked信号那样，后者是在构件已接收到鼠标事件并对它做出解释之后发送的。

11.3 本章总结

本章中编写的这个程序只涵盖了少数几个领域，但通过使用本章提供的信息以及GTK+和GStreamer的文档，有兴趣的读者应该能够实现如下的改进：

- FileChooserDialog在每次使用后都被销毁，这使得程序忘记用户上次浏览过的位置。请修改程序以建立一个FileChooserDialog，它在用户每次打开另一个文件时都被重复使用。
- 暂停功能已被刻意留下来作为读者的一个练习。使用GStreamer管道状态GST_STATE_PAUSED实现它，并对GUI的状态做出相应的改变。
- 文件的元数据直到管道开始播放时才被读取，因为GStreamer只有在开始播放文件后才能读取它。请修改load_file以从底层元素建立另一个管道，它可以获取这些元数据而不会导致任何声音的输出。它的消息处理函数在接收到包含所需信息的元数据消息之后将停止管道。你需要研究GStreamer的元素filesrc、decodebin和fakesink来实现该功能。

现在，你应该熟悉了使用C语言编写GNOME应用程序的基本概念。本章并没有涉及编写一个成熟的GNOME应用程序所需的所有领域，但有了对GObject和GTK+的理解，你在开始使用其他一些函数库（如Gconf）时应该不会有太多的困难。GNOME为应用程序开发者提供了很多帮助，尽管情况正在向有利于如C#等语言的方向改变，但C仍然是核心桌面应用程序开发的首选语言。

自由桌面项目

虽然GTK+提供了非常好的用于建立应用程序的工具包，而且GNOME本身也提供了一组很好的应用程序，但还有许多与桌面相关的重要功能由于各种原因既不属于一个构件工具包函数库也不属于一个单独的应用程序集。某些功能可能比应用程序的构建者需要考虑的层次略低一些，它们的实现应该与桌面保持中立（这样，即使使用一个不同的桌面环境（如KDE），它们也可以正常工作）。幸运的是，对于程序员来说存在一个中间解决方案。自由桌面（FreeDesktop）提供了一些可以在任何Linux桌面环境中工作的项目。它提供了针对许多常用桌面功能的通用应用程序编程接口（API）。

桌面总线、硬件抽象层和网络管理器是三个流行的自由桌面项目，它们为常见桌面问题提供了有用的接口。本章将演示如何编写代码以利用这些自由桌面项目来与实现了自由桌面的任何Linux桌面环境交互。在本章的最后，我们将介绍另外几个自由桌面项目，它们可能会在Linux编程中派上用场。

12.1 D-BUS：桌面总线

D-Bus是进程间通信（IPC）的一个机制，它属于FreeDesktop.org项目的一部分。它被设计用来分别取代GNOME和KDE先前使用的IPC机制：公共对象请求代理体系结构（CORBA）和桌面通信协议（DCOP）。D-Bus专门针对Linux桌面进行了订制，正因为如此，在很多情况下它的使用都非常简单。D-Bus的强大功能的关键就隐藏在它的名字中：bus守护进程。D-Bus没有使用其他IPC机制常用的二进制字节流，而使用了二进制消息的概念，消息由消息头和相应的数据组成。

本节首先讨论D-Bus的工作原理，然后演示如何编写一个使用D-Bus在客户机/服务器环境中传递消息的应用程序。

12.1.1 什么是 D-Bus

必须引起注意的是，虽然D-Bus是一个IPC机制，但它并不是一个通用目的的IPC机制。它是专用于桌面的，可能并不适合于其他方面的应用。具体来说，它的设计目的主要有两个：方便同一个桌面会话中的多个应用程序之间的通信；有助于桌面应用程序和操作系统之间的通信。这里所说的操作系统包括内核、系统守护进程和进程。

将这些牢记心中之后，你就会觉得D-Bus同时使用一个系统总线和一个会话总线是很自然的事情。在系统中的第一个总线实例是系统总线，它是一个系统范围内的特权守护进程，类似于inetd或httpd。系统总线可以接收的消息类型有严格的限制。此外，D-Bus还针对SELinux上下文进行了整合，这进一步提高了它的安全性。当一个用户登录到桌面时，另一个使用该用户上下文的总线会被创建，这是一个会话总线，它允许用户的应用程序相互之间进行通信。当应用程序同时使用这两个总线时，

它们可以实现相当强大的功能，例如获取系统事件如网络断线的通知，或基于网络状态执行任务。在本章的后面，你将学到强大的网络管理器应用程序是如何充分利用系统总线和会话总线的。

12.1.2 D-Bus 基础

D-Bus API并没有绑定到一个特定的语言或框架。例如，有Glib的D-Bus绑定（用C语言编写）、Qt的D-Bus绑定（用C++编写）、Python和C#的D-Bus绑定。因为我们刚刚在上一章介绍了有关GNOME的知识，它用C语言编写并且使用的是Glib，所以本章将重点介绍D-Bus的C语言绑定和Glib绑定。但这里介绍的所有原则都适用于其他各种绑定。

为了在应用程序中使用D-Bus，首先需要在程序中包括如下的头文件。

```
#include <dbus/dbus.h>
#include <dbus/dbus-glib.h>
```

然后每个应用程序必须连接到消息总线守护进程。请记住，有两个消息总线守护进程可以连接，现在是你决定连接哪个总线的时候了。如果你要连接会话总线，请使用如下的代码段：

```
DBusError error;
DBusConnection *conn;

dbus_error_init (&error);
conn = dbus_bus_get (DBUS_BUS_SESSION, &error);

if (!conn) {
    fprintf (stderr, "%s: %s\n", error.name, error.message);
    /* Other error Handling */
}
```

如果你要连接到系统总线，只需将DBUS_BUS_SESSION替换为DBUS_BUS_SYSTEM即可。但连接到系统总线可能会有点麻烦，因为系统总线对可以连接到它的用户有限制。你可能需要提供一个.service文件以详细说明谁能够或不能够连接到一个指定的服务器。本章重点讨论对会话总线的连接，但其原理对连接到系统总线同样适用。

无论应用程序是连接到系统总线还是会话总线，D-Bus都会分配一个唯一的连接名称给它。这个名字以冒号（:）开始，例如：80-218，而且它也是唯一一个可以以冒号开头的名称类型。对于每个守护进程实例来说，这些名称都是唯一的，而且在总线的生命周期中是决不会被复用的。但应用程序可以要求连接以一个更容易理解的名称引用。例如，org.foo.bar。

D-Bus是与应用程序中的对象而不是应用程序自身进行通信（尽管与应用程序的顶层对象通信将具有与直接与应用程序通信相同的效果）。在你的Glib应用程序中，这意味着是与GObject及其派生对象通信。但在应用程序中，这些对象是以应用程序地址空间中内存地址的形式在传递，将这些内存地址传递给其他应用程序是没有意义的。D-Bus采用的方法是传递对象路径。这些对象路径看上去类似于文件系统路径，例如，/org/foo/bar表示Foo Bar应用程序的顶层对象路径。虽然为对象路径划分命名空间并不是必需的，但我们强烈建议这样做以避免产生冲突。

D-Bus有4种不同类型的消息可以发送给一个对象。

- 信号消息：**它通知一个对象一个指定的信号已被发送或一个事件已发生。
- 方法调用消息：**要求在一个特定对象上调用一个方法。
- 方法返回消息：**返回在一个对象上调用一个方法的执行结果。

□ 错误消息：返回在一个对象上调用一个方法所产生的任何异常。

下面的代码段可用于发送一个信号：

```
#include <dbus/dbus.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
    DBusError dberr;
    DBusConnection *dbconn;
    DBusMessage *dbmsg;
    char *text;

    dbus_error_init (&dberr);
    dbconn = dbus_bus_get (DBUS_BUS_SESSION, &dberr);
    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "getting session bus failed: %s\n", dberr.message);
        dbus_error_free (&dberr);
        return EXIT_FAILURE;
    }

    dbmsg = dbus_message_new_signal ("/com/wiley/test",
                                    "com.wiley.test",
                                    "TestSignal");
    if (dbmsg == NULL) {
        fprintf (stderr, "Could not create a new signal\n");
        return EXIT_FAILURE;
    }

    text = "Hello World";
    dbus_message_append_args (dbmsg, DBUS_TYPE_STRING, &text, DBUS_TYPE_INVALID);

    dbus_connection_send (dbconn, dbmsg, NULL);
    printf ("Sending signal to D-Bus\n");

    dbus_message_unref (dbmsg);

    dbus_connection_close (dbconn);
    dbus_connection_unref (dbconn);

    return EXIT_SUCCESS;
}
```

当然，能够发送信号是很好，但为了使发送的信号有用，你必须知道如何处理D-Bus信号。这可能更复杂一些，但也不是太麻烦。它的开始代码和发送信号的开始代码非常相似：

```
int main (int argc, char *argv[])
{
    DBusError dberr;
    DBusConnection *dbconn;

    dbus_error_init (&dberr);
    dbconn = dbus_bus_get (DBUS_BUS_SESSION, &dberr);
```

```

if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "getting session bus failed: %s\n", dberr.message);
    dbus_error_free (&dberr);
    return EXIT_FAILURE;
}

接下来,为了能够正确地接收到信号,我们需要确保“拥有”com.wiley.test命名空间。
dbus_bus_request_name (dbconn, "com.wiley.test",
                       DBUS_NAME_FLAG_REPLACE_EXISTING, &dberr);
if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "requesting name failed: %s\n", dberr.message);
    dbus_error_free (&dberr);
    return EXIT_FAILURE;
}

```

增加一个过滤函数是一个好方法,它可以用来接收信号、过滤出你想要处理的信号并定义你想如何处理它们。现在我们只是简单地注册该函数,它的详细定义将在后面列出。

```

if (!dbus_connection_add_filter (dbconn,
                               filter_func, NULL, NULL))
    return EXIT_FAILURE;

```

既然我们想接收一个信号,我们还需要确保该信号不会被忽略。因为通过总线发送的信号有很多,而非所有的应用程序都对它们感兴趣,所以在默认情况下,信号将被忽略以阻止“信号泛滥”,即便它们处于同一个接口上也是如此。

```

dbus_bus_add_match (dbconn,
                    "type='signal',interface='com.wiley.test'",
                    &dberr);

if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "Could not match: %s", dberr.message);
    dbus_error_free (&dberr);
    return EXIT_FAILURE;
}

```

最后,你需要确保程序不会立刻退出,代码中必须存在某种类型的信号持续轮询。D-Bus提供了[一个底层API函数](#)用于完成这一工作。

```

while (dbus_connection_read_write_dispatch (dbconn, -1))
    /* empty loop body */

return EXIT_SUCCESS;
}

```

实现了程序的基本控制代码之后,现在我们可以开始实现过滤函数了。

```

static DBusHandlerResult
filter_func (DBusConnection *connection,
            DBusMessage *message,
            void *user_data)
{
    dbus_bool_t handled = FALSE;
    char *signal_text = NULL;

```

```

if (dbus_message_is_signal (message, "com.wiley.test", "TestSignal")) {
    DBusError dberr;

    dbus_error_init (&dberr);
    dbus_message_get_args (message, &dberr, DBUS_TYPE_STRING, &signal_text,
    DBUS_TYPE_INVALID);
    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "Error getting message args: %s", dberr.message);
        dbus_error_free (&dberr);
    } else {
        DBusConnection *dbconn = (DBusConnection*) user_data;

        printf ("Received TestSignal with value of: '%s'\n", signal_text);

        handled = TRUE;
    }
}

return (handled ? DBUS_HANDLER_RESULT_HANDLED :
DBUS_HANDLER_RESULT_NOT_YET_HANDLED);
}

```

这段代码非常简单。它首先检查从而确认信号是否是一个使用了com.wiley.test接口的TestSignal。然后，它获取期望的参数（这是从被发送的信号中得知的）并将它们打印到主控台上。先运行dbus-get-hello，然后在另一个主控台上运行dbus-send-hello，它们的输出如下所示：

```

% ./dbus-get-hello
Received TestSignal with value of: 'Hello World'

% ./dbus-send-hello
Sending signal to D-Bus

```

祝贺你！你现在有了一个可以正常工作的D-Bus客户机和服务器！

12.1.3 D-Bus方法

有时候，仅仅来回地发送信号并不能满足一个程序的需求。例如，一个程序可能想要“问一个问题”或调用一些函数并返回执行的结果，比如网络管理器可能想告知它的系统守护进程让它连接到一个特定网络并返回该网络的特性。正如我们前面提到的那样，D-Bus允许使用方法调用消息和方法返回消息。

D-Bus信号消息是由dbus_message_new_signal()创建的。如果你想调用一个D-Bus方法，你只需使用dbus_message_new_method_call()创建一个消息。下面我们将尝试调用一个方法，它将三个整数加在一起并返回结果。我们从当前的小时、分钟和秒中获取整数。

首先，包括所需的头文件并准备好D-Bus。这些步骤和发送信号所需的步骤非常相似。

```

#include <dbus/dbus.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

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

```

```

DBusError dberr;
DBusConnection *dbconn;
DBusMessage *dbmsg, *dbreply;
DBusMessageIter dbiter;
int arg1, arg2, arg3, result;
struct tm *cur_time;
time_t cur_time_t;

dbus_error_init (&dberr);
dbconn = dbus_bus_get (DBUS_BUS_SESSION, &dberr);
if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "getting session bus failed: %s\n", dberr.message);
    dbus_error_free (&dberr);
    return EXIT_FAILURE;
}

```

与发送一个信号不同，这里我们要调用一个方法，所以准备方法调用：

```

dbmsg = dbus_message_new_method_call ("com.ailлон.test",
                                      "/com/ailлон/test",
                                      "com.ailлон.test",
                                      "add_three_ints");

if (dbmsg == NULL) {
    fprintf (stderr, "Couldn't create a DBusMessage");
    return EXIT_FAILURE;
}

```

现在，方法调用已准备好。它只需知道使用什么参数调用该方法即可。我们获取当前时间并通过它向方法提供一些整数。

```

cur_time_t = time (NULL);
cur_time = localtime (&cur_time_t);
arg1 = cur_time->tm_hour;
arg2 = cur_time->tm_min;
arg3 = cur_time->tm_sec;

dbus_message_iter_init_append (dbmsg, &dbiter);
dbus_message_iter_append_basic (&dbiter, DBUS_TYPE_INT32, &arg1);
dbus_message_iter_append_basic (&dbiter, DBUS_TYPE_INT32, &arg2);
dbus_message_iter_append_basic (&dbiter, DBUS_TYPE_INT32, &arg3);

```

你会注意到上面的代码使用了一个DBusMessageIter来添加参数。以前，参数是使用dbus_message_append_args()来添加的。两者都是正确的，它们不过是执行同一个任务的两种不同方法而已。

接下来，是时候调用该方法了：

```

printf ("Calling add_three_ints method\n");
dbus_error_init (&dberr);
dbreply = dbus_connection_send_with_reply_and_block (dbconn, dbmsg, 5000,
&dberr);
if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "Error getting a reply: %s!", dberr.message);
    dbus_message_unref (dbmsg);
    dbus_error_free (&dberr);
    return EXIT_FAILURE;
}

```

```

}

/* Don't need this anymore */
dbus_message_unref (dbmsg);

```

最后，从响应中提取结果并将它打印到主控台上，然后完成清理工作：

```

dbus_error_init (&dberr);
dbus_message_get_args (dbreply, &dberr, DBUS_TYPE_INT32, &result);
if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "Error getting the result: %s!", dberr.message);
    dbus_message_unref (dbmsg);
    dbus_error_free (&dberr);
    return EXIT_FAILURE;
}

printf ("Result of : add_three_ints (%d, %d, %d) is %d\n", arg1, arg2, arg3,
result);

dbus_message_unref (dbreply);

dbus_connection_close (dbconn);
dbus_connection_unref (dbconn);

return EXIT_SUCCESS;
}

```

当然，现在需要实现该方法了。首先连接会话总线：

```

int main (int argc, char *argv[])
{
    DBusError dberr;
    DBusConnection *dbconn;

    dbus_error_init (&dberr);
    dbconn = dbus_bus_get (DBUS_BUS_SESSION, &dberr);

    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "getting session bus failed: %s\n", dberr.message);
        dbus_error_free (&dberr);
        return EXIT_FAILURE;
    }

```

接下来，请求名称的拥有权以使得D-Bus知道我们拥有该命名空间并将方法调用转发给我们。我们没有必要明确地为该方法调用添加一个匹配，因为D-Bus将自动地转发这些方法调用，但我们还是像前面做的那样添加了一个过滤函数，然后开始循环以轮询D-Bus消息。

```

    dbus_bus_request_name (dbconn, "com.ailion.test",
                           DBUS_NAME_FLAG_REPLACE_EXISTING, &dberr);
    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "requesting name failed: %s\n", dberr.message);
        dbus_error_free (&dberr);
        return EXIT_FAILURE;
    }

    if (!dbus_connection_add_filter (dbconn, filter_func, dbconn, NULL))

```

```

        return EXIT_FAILURE;

    while (dbus_connection_read_write_dispatch (dbconn, -1))
        ; /* empty loop body */

    return EXIT_SUCCESS;
}

```

过滤函数只需要进行检查以确保我们正在接收的方法调用是正确的，然后转发该调用：

```

static DBusHandlerResult
filter_func (DBusConnection *connection,
            DBusMessage *message,
            void *user_data)
{
    dbus_bool_t handled = FALSE;

    if (dbus_message_is_method_call (message, "com.ailлон.test",
"add_three_ints")) {
        printf ("Handling method call\n");
        add_three_ints (message, connection);
        handled = TRUE;
    }

    return (handled ? DBUS_HANDLER_RESULT_HANDLED :
DBUS_HANDLER_RESULT_NOT_YET_HANDLED);
}

```

整个代码的核心是实际的函数调用，在本例中，就是add_three_ints()。它首先从传递进来的DBusMessage中提取出需要的参数。

```

static void
add_three_ints (DBusMessage *message, DBusConnection *connection)
{
    DBusError dberr;
    DBusMessage *reply;
    dbus_uint32_t arg1, arg2, arg3, result;

    dbus_error_init (&dberr);
    dbus_message_get_args (message, &dberr, DBUS_TYPE_INT32, &arg1,
DBUS_TYPE_INT32, &arg2, DBUS_TYPE_INT32, &arg3, DBUS_TYPE_INVALID);

```

如果参数不符合要求，我们就创建一个错误响应消息以发送回调用者。否则，创建一个方法返回消息并添加参数。

```

    if (dbus_error_is_set (&dberr)) {
        reply = dbus_message_new_error_printf (message, "WrongArguments", "%s",
dberr.message);
        dbus_error_free (&dberr);
        if (reply == NULL) {
            fprintf (stderr, "Could not create reply for message!");
            exit (1);
        }
    } else {
        result = arg1 + arg2 + arg3;
    }
}

```

```

        reply = dbus_message_new_method_return (message);
        if (reply == NULL) {
            fprintf (stderr, "Could not create reply for message!");
            exit (1);
        }

        dbus_message_append_args (reply, DBUS_TYPE_INT32, &result,
DBUS_TYPE_INVALID);
    }
}

```

最后，发送消息并执行清理：

```

dbus_connection_send (connection, reply, NULL);

dbus_message_unref (reply);
}

```

对该程序的调用将产生如下的输出：

```

% ./dbus-call-method
Calling add_three_ints method
Result of : add_three_ints (16, 19, 7) is 42
% ./dbus-call-method
Calling add_three_ints method
Result of : add_three_ints (16, 19, 8) is 43
% ./dbus-call-method
Calling add_three_ints method
Result of : add_three_ints (16, 19, 9) is 44

```

很好，方法被成功调用并返回了结果。通过使用这两个基本概念，你可以创建自己的应用程序以利用D-Bus的功能了。

12.2 硬件抽象层

你已在上一节中了解了D-Bus的强大功能并学习了如何使用它。但是，应用程序当然希望最好能对发送的信号和消息进行标准化。本节将介绍什么是硬件抽象层（HAL）、它和D-Bus的关系以及如何在应用程序中使用HAL。

12.2.1 使硬件可以即插即用

Linux的老用户肯定还记得以前在获得一个新硬件时所遇到的麻烦。他们需要花费时间找出可以让这个设备正常工作所需的内核模块。如果这个设备支持热拔插，例如一个USB设备，你可能不希望在没有使用这个设备时仍然装载对应的模块。但当你在许多设备之间不停地进行切换时，你很容易混淆到底应该装载哪个模块。

当然，即使设备能够正常的工作，这一过程也显得过于冗长乏味，当一些临时用户发现他们不能使插入计算机中的照相机即插即用，或不能立刻访问他们的USB硬盘或一台打印机时，他们可能会感到沮丧。应用程序不可能关注用户具体使用的是什么硬件，否则，程序的作者就必须针对他想要支持的每一种硬件来更新他的软件。

最终，Red Hat公司的Havoc Pennington撰写了一篇论文，题为“使硬件可以即插即用”(*Making Hardware Just Work*, <http://ometer.com/hardware.html>)，在文中他概述了一些他认为存在的问题，并提

出了使用硬件抽象层（HAL）解决这些问题的想法。几个月之后，第一段代码被提交到CVS版本库。如今，HAL已能够发现各种各样的设备，如存储设备、网络设备、打印机、媒体播放器如iPod和数码相机等。

HAL是一个系统守护进程，它监控当前系统中的硬件。它通过在/etc/dev.d目录中安装一个脚本来监听热拔插事件如在系统中插入和删除硬件设备，每当udev修改/dev中的条目时，该脚本就会运行。HAL还会在其启动时通过udev获取系统中所有设备的列表。除了提供设备的信息以外，HAL还可以执行诸如锁定一个硬件以使得应用程序可以获得对该设备的独占式访问等特定任务。

HAL API完全在D-Bus中，但因为它是一个相当大的API，而且许多程序都是用C语言编写的，所以存在一个名为libhal的C辅助函数库。HAL的使用相当简单，下面我们来创建一个应用程序，它监听下一个将要添加到系统中或从系统中删除的设备，并打印一条简短消息，然后退出。

```
#include <libhal.h>
#include <dbus/dbus.h>
#include <dbus/dbus-glib.h>
#include <glib.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
    GMainLoop *mainloop;

    DBusConnection *dbconn;
    DBusError dberr;

    LibHalContext *hal_ctx;

    mainloop = g_main_loop_new (NULL, FALSE);
    if (mainloop == NULL) {
        fprintf (stderr, "Can't get a mainloop");
        return EXIT_FAILURE;
    }

    dbus_error_init (&dberr);
    dbconn = dbus_bus_get (DBUS_BUS_SYSTEM, &dberr);
    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "Can't get D-Bus system bus!");
        return EXIT_FAILURE;
    }

    dbus_connection_setup_with_g_main (dbconn, NULL);
}
```

一切就绪之后，现在我们可以创建一个LibHalContext并告诉它哪些事件应该让回调函数处理了。请注意，这里的语句顺序非常重要。首先，我们必须使用libhal_ctx_new()创建一个LibHalContext，然后，在调用libhal_ctx_init()之前，所有的回调函数、D-Bus连接等都必须被注册。

```
hal_ctx = libhal_ctx_new ();
if (hal_ctx == NULL) {
```

```

        fprintf (stderr, "Can't create a LibHalContext!");
        return EXIT_FAILURE;
    }

    /* Associate HAL with the D-Bus connection we established */
    libhal_ctx_set_dbus_connection (hal_ctx, dbconn);
    /* Register callbacks */
    libhal_ctx_set_device_added (hal_ctx, my_device_added_callback);
    libhal_ctx_set_device_removed (hal_ctx, my_device_removed_callback);
    /* We will be breaking out of the mainloop in a callback */
    libhal_ctx_set_user_data (hal_ctx, mainloop);

    dbus_error_init (&dberr);
    libhal_device_property_watch_all (hal_ctx, &dberr);
    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "libhal_device_property_watch_all() failed: '%s',
dberr.message);
        dbus_error_free (&dberr);
        libhal_ctx_free (hal_ctx);
        return EXIT_FAILURE;
    }

    dbus_error_init (&dberr);
    libhal_ctx_init (hal_ctx, &dberr);
    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "libhal_ctx_init() failed: '%s'. Is hald running?",
                dberr.message);
        dbus_error_free (&dberr);
        libhal_ctx_free (hal_ctx);
        return EXIT_FAILURE;
    }
}

```

HAL已做好最后的准备了。但为了接收事件，我们必须启动主循环：

```

g_main_loop_run (mainloop);

return EXIT_SUCCESS;
}

```

现在，需要实现的还有当一个设备被添加或删除时HAL期待调用的回调函数。这个测试程序只关心一个设备是否被添加或删除，然后就退出（当然是在完成了清理工作并关闭HAL连接之后）。

```

static void all_done (LibHalContext *hal_ctx)
{
    DBusError dberr;
    GMainLoop *mainloop;

    mainloop = (GMainLoop*) libhal_ctx_get_user_data (hal_ctx);

    dbus_error_init (&dberr);
    libhal_ctx_shutdown (hal_ctx, &dberr);
    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "libhal shutdown failed: '%s'", dberr.message);
        dbus_error_free (&dberr);
    }
}

```

```

    }
    libhal_ctx_free (hal_ctx);

    g_main_loop_quit (mainloop);
}

static void my_device_added_callback (LibHalContext *hal_ctx, const char *udi)
{
    printf ("Device added: '%s'\n", udi);

    all_done (hal_ctx);
}

static void my_device_removed_callback (LibHalContext *hal_ctx, const char *udi)
{
    printf ("Device removed: '%s'\n", udi);

    all_done (hal_ctx);
}

```

编译并启动程序，然后在系统中插入一个设备，如一个USB鼠标或数码相机。程序的输出如下所示：

```

% ./helloworld
Device added: '/org/freedesktop/Hal/devices/usb_device_46d_c016_noserial'
% ./helloworld
Device removed:
'/org/freedesktop/Hal/devices/usb_device_46d_c016_noserial_if0_logicaldev_input'

```

12.2.2 HAL 设备对象

上一节我们介绍了HAL的基础知识，包括如何连接HAL以及当设备被添加或删除时如何让它通知你。现在，是时候来探讨HAL设备对象的细节了。

一个HAL设备对象由几部分组成。每个HAL设备都有一个唯一设备标识符（UDI），在同一时间，设备之间不能共享UDI。UDI是通过使用总线上的信息进行计算的，这意味着设备每次插入时所获得的UDI应该都是不变的。但对于那些不提供序列号的设备来说，这未必能完全做到。此时，HAL将使用产品和厂商信息，有时候还使用设备的物理位置（如在总线42上的PCI插槽2）来生成UDI，这就可能造成设备每次插入时所获得的UDI略有不同。

HAL设备对象还有与其关联的键/值属性。键是一个ASCII字符串，值可以是下面几种类型之一：

- bool, int (32位有符号整数);
- uint64, string (UTF-8字符串);
- strlist (UTF-8字符串的有序列表);
- double。

属性键使用以.字符分隔的命名空间，例如info.linux.driver和net.80211.mac_address。

HAL属性键可以被认为属于如下四类。

- 元数据属性：由HAL设置，用于描述设备之间互连的关系和设备类型。
- 物理属性：由HAL通过设备自身或设备信息文件来确定，它包含产品和厂商ID等信息。
- 功能属性：给出设备的当前状态信息。例如：网络连接状态、文件系统装载位置、笔记本电池充电，等等。

□ 策略属性：定义设备应如何被用户使用，它通常是由系统管理员在设备信息文件中设置的。

HAL还提供了与设备的功能相关的信息。这是在info.capabilities属性中给出的，该属性提供了一个设备所具备的功能的信息。例如，一个便携式音乐播放器（如iPod）还可以被用作为一个存储设备。在这种情况下，info.capabilities将是一个包含portable_music_player和storage.disk的strlist。info.category键指明这个设备的主要类型，它将被设置为portable_music_player。

最后，值得指出的是HAL设备对象并不一定要与物理设备之间存在一对一映射的关系。例如，一个多功能的打印机可能有三个分别用于打印、扫描和存储的HAL设备对象。而其他用于多种用途的设备可能只有一个HAL设备接口。主设备将有一个列出其所有功能的单一列表，而在HAL树中该设备的下面将有三个设备分别对应每个功能。

现在你已对HAL设备对象的工作原理有了一个基本的了解，下面我们来编写一个程序用于找到系统中的光驱并打印关于它们的一些细节信息。光驱具备storage.cdrom能力，所以这应该是我们要求HAL提供的信息。为了确定光驱是否还支持读取DVD，我们还需要询问HAL storage.cdrom.dvd属性的值。当然，光驱的最大读取速度对我们也是有用处的，它是由storage.cdrom.read_speed属性确定的。

HAL的初始化过程与上一个例子类似。但是，因为这次我们所需要的只是获取当前设备的列表，所以我们不需要监听事件或启动主循环，因此也不需要使用Glib了。

```
#include <libhal.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
    DBusConnection *dbconn;
    DBusError dberr;

    LibHalContext *hal_ctxt;
    LibHal.PropertyType property_type;
    char *property_key;
    dbus_bool_t bool_value;
    int int_value;
    char **udis;
    int num_udis;
    int i;

    dbus_error_init (&dberr);
    dbconn = dbus_bus_get (DBUS_BUS_SYSTEM, &dberr);
    if (dbus_error_is_set (&dberr)) {
        fprintf (stderr, "Can't get D-Bus system bus!");
        return EXIT_FAILURE;
    }

    hal_ctxt = libhal_ctxt_new ();
    if (hal_ctxt == NULL) {
        fprintf (stderr, "Can't create a LibHalContext!");
        return EXIT_FAILURE;
    }
```

```

/* Associate HAL with the D-Bus connection we established */
libhal_ctx_set_dbus_connection (hal_ctx, dbconn);

dbus_error_init (&dberr);
libhal_ctx_init (hal_ctx, &dberr);
if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "libhal_ctx_init() failed: '%s'. Is hald running?",
            dberr.message);
    dbus_error_free (&dberr);
    libhal_ctx_free (hal_ctx);
    return EXIT_FAILURE;
}

```

初始化HAL之后，现在是时候来查找具备storage.cdrom功能的设备了。

```

/* Looking for optical drives: storage.cdrom is the capability */
udis = libhal_find_device_by_capability (hal_ctx, "storage.cdrom", &num_udis,
                                         &dberr);
if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "libhal_find_device_by_capability error: '%s'\n",
            dberr.message);
    dbus_error_free (&dberr);
    libhal_ctx_free (hal_ctx);
    return EXIT_FAILURE;
}

```

变量udis现在包含了HAL找到的匹配storage.cdrom功能的一个udis数组。接下来需要做的是遍历该数组以便获取每个设备的属性。我们将首先使用libhal_device_get_property_type来确保每个属性值的类型都是我们所期望的。

```

printf ("Found %d Optical Device(s)\n", num_udis);
for (i = 0; i < num_udis; i++) {
    /* Ensure our properties are the expected type */
    property_type = libhal_device_get_property_type (hal_ctx,
                                                      udis[i],
                                                      "storage.cdrom.dvd",
                                                      &dberr);
    if (dbus_error_is_set (&dberr) ||
        property_type != LIBHAL_PROPERTY_TYPE_BOOLEAN)
    {
        fprintf (stderr, "error checking storage.cdrom.dvd type");
        dbus_error_free (&dberr);
        libhal_ctx_free (hal_ctx);
        return EXIT_FAILURE;
    }

    property_type = libhal_device_get_property_type (hal_ctx,
                                                      udis[i],
                                                      "storage.cdrom.read_speed",
                                                      &dberr);
    if (dbus_error_is_set (&dberr) || property_type !=
LIBHAL_PROPERTY_TYPE_INT32) {
        fprintf (stderr, "error checking storage.cdrom.read_speed type");
        dbus_error_free (&dberr);
    }
}

```

```

    libhal_ctx_free (hal_ctx);
    return EXIT_FAILURE;
}

```

现在我们知道每个属性值的类型都是正确的，我们可以安全地获取它们的值并将值打印到标准输出了。因为属性值的类型是bool和int，所以我们需要使用libhal_device_get_property_bool和libhal_device_get_property_int。还有其他方法对应它们各自的类型。这些方法都相当简单，请查看libhal.h以了解它们的更多细节。

```

/* Okay, now simply get property values */
bool_value = libhal_device_get_property_bool (hal_ctx, udis[i],
                                              "storage.cdrom.dvd",
                                              &dberr);
if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "error getting storage.cdrom.dvd");
    dbus_error_free (&dberr);
    libhal_ctx_free (hal_ctx);
    return EXIT_FAILURE;
}
int_value = libhal_device_get_property_int (hal_ctx, udis[i],
                                             "storage.cdrom.read_speed",
                                             &dberr);
if (dbus_error_is_set (&dberr)) {
    fprintf (stderr, "error getting storage.cdrom.dvd");
    dbus_error_free (&dberr);
    libhal_ctx_free (hal_ctx);
    return EXIT_FAILURE;
}

/* Display the info we just got */
printf ("Device %s has a maximum read speed of %d kb/s and %s read
DVDs.\n",
       udis[i], int_value, bool_value ? "can" : "cannot");
}

return EXIT_SUCCESS;
}

```

编译并运行这个程序。其输出如下所示：

```

% ./hal-optical-test
Found 2 Optical Device(s)
Device /org/freedesktop/Hal/devices/storage_model_HL_DT_STDVD_ROM_GDR8162B has a
maximum read speed of 8467 kb/s and can read DVDs.
Device /org/freedesktop/Hal/devices/storage_model_HL_DT_ST_GCE_8483B has a maximum
read speed of 8467 kb/s and cannot read DVDs.

```

为了更好地了解HAL所关心的属性列表，请查看程序lshal的输出。它将输出HAL知道的所有设备以及它知道的该设备上的所有属性。其输出列表如下所示：

```

% lshal
Dumping 99 device(s) from the Global Device List:
...
udi = '/org/freedesktop/Hal/devices/storage_model_HL_DT_STDVD_ROM_GDR8162B'

```

```

info.addons = {'hald-addon-storage'} (string list)
block.storage_device =
'/org/freedesktop/Hal/devices/storage_model_HL_DT_STDVD_ROM_GDR8162B' (string)
info.udi = '/org/freedesktop/Hal/devices/storage_model_HL_DT_STDVD_ROM_GDR8162B'
(string)
storage.cdrom.write_speed = 0 (0x0) (int)
storage.cdrom.read_speed = 8467 (0x2113) (int)
storage.cdrom.support_media_changed = true (bool)
storage.cdrom.hddvdrw = false (bool)
storage.cdrom.hddvdr = false (bool)
storage.cdrom.hddvd = false (bool)
storage.cdrom.bdre = false (bool)
storage.cdrom.bdr = false (bool)
storage.cdrom.bd = false (bool)
storage.cdrom.dvdplusrdl = false (bool)
storage.cdrom.dvdplusrw = false (bool)
storage.cdrom.dvdplusr = false (bool)
storage.cdrom.dvdram = false (bool)
storage.cdrom.dvdwr = false (bool)
storage.cdrom.dvdr = false (bool)
storage.cdrom.dvd = true (bool)
storage.cdrom.cdrw = false (bool)
storage.cdrom.cdr = false (bool)
storage.requires_eject = true (bool)
storage.hotpluggable = false (bool)
info.capabilities = {'storage', 'block', 'storage.cdrom'} (string list)
info.category = 'storage' (string)
info.product = 'HL-DT-STDVD-ROM GDR8162B' (string)
storage.removable = true (bool)
storage.physical_device = '/org/freedesktop/Hal/devices/pci_8086_24db_ide_1_0'
(string)
storage.firmware_version = '0015' (string)
storage.vendor = '' (string)
storage.model = 'HL-DT-STDVD-ROM GDR8162B' (string)
storage.drive_type = 'cdrom' (string)
storage.autounmount_enabled_hint = true (bool)
storage.media_check_enabled = true (bool)
storage.no_partitions_hint = true (bool)
storage.bus = 'ide' (string)
block.is_volume = false (bool)
block.minor = 0 (0x0) (int)
block.major = 22 (0x16) (int)
block.device = '/dev/hdc' (string)
linux.hotplug_type = 3 (0x3) (int)
info.parent = '/org/freedesktop/Hal/devices/pci_8086_24db_ide_1_0' (string)
linux.sysfs_path_device = '/sys/block/hdc' (string)
linux.sysfs_path = '/sys/block/hdc' (string)
...
Dumped 99 device(s) from the Global Device List.
-----
```

就像标准的ls命令对文件所做的那样，lshal产生系统中所有的被HAL识别的设备列表。这是一个方便、迅速而又随性的方法，可以使用它来确定Linux系统识别了哪些硬件设备以及这些设备是如

何被配置的。

本节介绍了使用HAL的基础以及如何从HAL中提取硬件信息。下一节将通过介绍自由桌面项目网络管理器来带你进入网络世界。

12.3 网络管理器

网络管理器作为Red Hat的无状态(stateless) Linux项目的一部分被引入。其基本前提是Linux中的网络应该是即插即用的。其目标是确保用户不需要通过输入各种iwconfig命令来手工配置无线网卡以获取一个IP地址。

正如你已学习过的，当系统进行网络连接时，网络管理器利用HAL和D-Bus的强大功能为用户提供了非常好的无线体验。但为了获得真正的无状态体验，其他应用程序也需要关注网络。例如，Web浏览器可能需要知道机器是否连接到网络以便确定是通过网络装载Web页面还是通过其磁盘缓存读取页面。音乐播放器可能希望在用户连接到网络时显示因特网广播电台，否则它将关闭该功能。电子邮件程序也可能会有类似的需求。

桌面应用程序可以通过监听网络管理器所使用的同一个D-Bus接口来确定网络状态。网络管理器还公开一个接口供使用Glib的应用程序用来快速地确定网络状态而不需要它们找出其D-Bus信号。被称为libnm(用于网络管理器)的接口很容易使用。

首先，包括头文件并初始化libnm-glib:

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

int main(int argc, char* argv[])
{
    GMainLoop      *loop;
    libnm_glib_ctx *ctx;
    guint id;

    ctx = libnm_glib_init ();
    if (ctx == NULL)
    {
        fprintf (stderr, "Could not initialize libnm.\n");
        exit (1);
    }
}
```

然后，注册回调函数(回调函数本身将在后面定义)：

```
id = libnm_glib_register_callback (ctx, status_printer, ctx, NULL);
```

最后，请确认启动了Glib的主循环：

```
loop = g_main_loop_new (NULL, FALSE);
g_main_loop_run (loop);

exit (0);
}
```

当然，我们还需要定义回调函数。因为这只是一个测试程序，所以打印状态就足够了：

```
static void status_printer (libnm_glib_ctx *ctx, gpointer user_data)
{
    libnm_glib_state      state;
```

```

g_return_if_fail (ctx != NULL);

state = libnm_glib_get_network_state (ctx);
switch (state)
{
    case LIBNM_NO_DBUS:
        fprintf (stderr, "Status: No DBUS\n");
        break;
    case LIBNM_NO_NETWORKMANAGER:
        fprintf (stderr, "Status: No NetworkManager\n");
        break;
    case LIBNM_NO_NETWORK_CONNECTION:
        fprintf (stderr, "Status: No Connection\n");
        break;
    case LIBNM_ACTIVE_NETWORK_CONNECTION:
        fprintf (stderr, "Status: Active Connection\n");
        break;
    case LIBNM_INVALID_CONTEXT:
        fprintf (stderr, "Status: Error\n");
        break;
    default:
        fprintf (stderr, "Status: unknown\n");
        break;
}
}

```

通过使用网络管理器，我们很容易检查网络连接的状态。与不得不搜寻硬件网卡并查询其特征相比，我们在这里只需使用几行代码就可以和Linux系统上的底层网络配置进行交互并获取其状态。

12.4 其他自由桌面项目

自由桌面项目还有其他几个比较重要的软件。虽然本书不会详细讨论它们，但对其中的几个软件有一个基本的认识还是很有必要的。

使用开放源码操作系统（如Linux）的好处之一就是不会缺乏应用程序。Linux系统上有丰富的文本编辑器、图像查看器、Web浏览器和媒体播放器供用户安装。但要想搞清楚哪个应用程序可以打开哪个类型的文件以及它们是否已安装到了系统上是很困难的。此外，因为现在文件类型的数目太多，期望用户知道他们拥有哪些类型的文件也是不合理的。当然，用户可以对文件运行file命令，但一般用户不应该必须这么做。从文件浏览器（如Nautilus）中打开文件时应该启动对应的查看器。此外，用户应该能够配置他或她个人所偏爱的应用程序。

FreeDesktop.org针对这个问题提出了它的解决方案：shared-mime-info项目（<http://freedesktop.org/wiki/Standards/shared-mime-info-spec>）。该项目有效地编译了一个静态数据库以表明文件如何映射到相应的MIME类型。这个数据库提供规则以允许应用程序确定某些文件名（如*.txt文件）应该被视为某些类型（如text/plain）。通过使用这些信息，当用户请求打开某个文件时，OS可以对启动哪个应用程序做出更好的选择。

为了做出上述决定，系统需要知道每个应用程序可以处理的文件类型。桌面文件规范（<http://freedesktop.org/wiki/Standards/desktop-entry-spec>）提供了该信息以及可以由菜单系统和应用程序

启动器使用的应用程序的其他细节。桌面文件是一个键/值文本文件，它提供了应用程序显示在UI中的名称、用来表示应用程序的图标、要启动的可执行文件、当然还有该应用程序可以处理的MIME类型。在Red Hat/Fedora系统中，桌面文件可以在/usr/share/applications/中找到。这些文件非常简单，通过查看它们将有助于你理解如何编写它们。

你必须确保已正确安装了桌面文件。请使用在<http://www.freedesktop.org/software/desktop-file-utils>上提供的最新版本的desktop-file-utils软件包。desktop-fileinstall程序将确保所有缓冲都被正确地更新。

正如你已看到的，自由桌面项目为Linux程序员提供了广泛的应用程序，这里显示的只是冰山一角。FreeDesktop.org网站上还有更多的应用程序可以供你在自己的程序中使用，请仔细阅读该网站并尽情地在你的程序中利用它们。

12.5 本章总结

在本章中，你了解了D-Bus是由Linux桌面使用的一种IPC机制，知道了它有一个系统总线和一个会话总线。你还了解了D-Bus可以发送的消息有四种不同的类型以及如何发送和接收它们。然后，你实现了一个D-Bus客户机和服务器。你学习了什么是HAL项目以及它是如何利用D-Bus的。你学习了如何监听HAL事件、什么是HAL设备对象以及如何获取有关它的信息。你还学习了网络管理器和如何监听网络状态事件。最后，你对桌面文件规范和shared-mime-info项目有了一个基本的了解。最重要的是，通过学习本章，你知道了如何将所有这一切结合起来以改善用户的Linux桌面体验。

图形和音频

Linux高级编程的许多方面都需要用到高级的图形和音频支持。从建立数学模型到编写最新的动作游戏等一切都需要系统具备绘制复杂几何对象或制作复杂几何对象动画的能力。同样地，伴随如今的多媒体热潮，许多应用程序需要处理音频文件，包括数字文件和直接来自音频音乐CD的数据。但不幸的是，大多数Linux系统所包含的标准编程工具并不支持开发这些高级图形和音频程序的方法。本章将介绍两个可以添加到你的编程工具包中的工具，它们可以帮助你创建具备高级图形和音频功能的Linux程序。

13.1 Linux 和图形

如今的大多数Linux发行版都是基于图形的。当启动系统时，你将看到一个欢迎你登录的图形画面，它通常还包含一个有趣的背景图像和一个图形化的用户和选项列表。一旦你登录进Linux系统，你将身处一个图形化的桌面环境中，它包含各种程序和设备的图标以及一个用于选择程序的菜单图标。启动程序只需点击相应的桌面图标或从菜单中选择相应的选项即可。

但为了做到这一点，Linux系统不得不在幕后做大量的工作。Linux系统中有许多程序代码专用于显示图形。图形编程也是Linux中最复杂的编程环境之一。操纵图形对象需要用户了解显卡和显示器是如何进行交互以产生屏幕图像的。

幸运的是，有几个工具可以帮助Linux图形程序员。本节将介绍4个用于在Linux系统中建立图形的工具：X Windows、OpenGL、GLUT和SDL。

13.1.1 X 视窗

如今操作系统所提供的最常见的图形形式就是windows视窗系统。除了同名的著名的微软操作系统以外，还有其他几个基于视窗的操作系统也为程序员提供了图形化视窗环境。Linux通过支持多种类型的图形化视窗环境加入了图形化革命。

所有的操作系统都依赖于两个基本元素来控制视频环境：安装在系统中的显卡和用于显示图像的显示器。这两个元素都必须被检测和控制以正确显示程序中的图形。我们所需要的是能够轻松地访问到这些元素。

Linux发行版使用X视窗标准来管理图形。该标准定义了作为图形化应用程序与系统的显卡、显示器组合之间接口的协议。

因为X视窗标准扮演了一个中间人的角色，所以它提供了一个针对各种显卡和显示器的通用接口。与X视窗标准交互的应用程序可以运行在使用任何显卡和显示器组合的系统中，只要这个显卡和显示器提供了一个X视窗接口。它们之间的关系如图13-1所示。

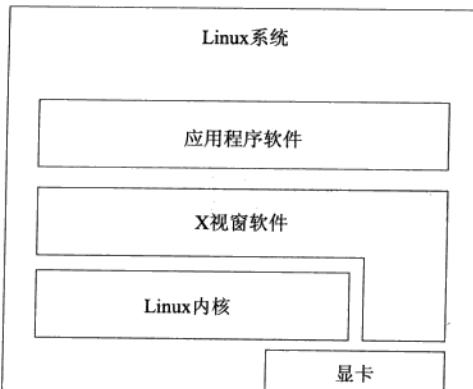


图13-1 Linux系统中的显卡和显示器与X视窗接口之间的关系

X视窗标准本身只是一个规范，而不是一个产品。每个操作系统都使用不同的X视窗软件来实现X视窗标准。在Linux世界中，目前有两个实现X视窗的软件包：

- XFree86
- X.org

XFree86软件包是最早的X视窗标准实现。在相当长的时间内，它也是Linux唯一可用的X视窗软件包。正如其名字所暗示的那样，它是一个自由、开放源码的运行在Intel x86系统上的X视窗软件。

最近，一个名为X.org的新软件包进入Linux的视野。X.org软件一开始只是作为XFree86软件的一个开发分支，但由于其添加了一些更高级的功能，它迅速地在各种Linux发行版中普及。许多Linux发行版现在都使用X.org软件来替代老的XFree86系统。

这两个软件包有相同的工作方式，它们都控制Linux如何与显卡交互以在显示器上显示图形内容。为了做到这一点，X视窗软件必须针对它所运行的特定硬件系统进行配置。在过去，你不得不通过浏览一个很大的配置文件来针对所使用的特定类型的显卡和显示器对XFree86软件进行手工配置。错误的配置可能会烧毁一台昂贵的显示器。如今，大多数Linux发行版在系统被安装时就将自动检测显卡和显示器并创建适当的配置文件而不需要来自用户的任何干预了。

在安装过程中，Linux通过探测来获得显卡和显示器的信息，然后创建包含所发现信息的合适的X视窗配置文件。当你在安装一个Linux发行版时，可能会注意到它扫描显示器以获得所支持视频模式的时刻。有时候，这会导致显示器白屏几秒钟。由于显卡和显示器的类型非常多，所以这一过程可能需要一些时间来完成。

但不幸的是，有时候安装过程不能自动检测到要使用的视频设置，尤其是对于那些较新的、更高级的显卡。更糟的是，有些Linux发行版在不能发现系统的特定显卡设置时将安装失败。而其他发行版则会放弃自动检测并在安装过程中询问几个问题来搜集所需的信息。还有一些发行版会在自动检测失败时进行默认设置，这将产生一个并非针对某个特定视频环境定制的屏幕图像。

许多PC用户拥有高档显卡，如3-D加速卡，使他们能够运行高解析度的游戏。3-D加速卡提供了可以提高在屏幕上绘制三维对象的速度的硬件和软件。在过去，当你尝试安装Linux时，特殊的视频硬件可能会导致很多问题。但最近，显卡厂商正在通过提供他们的产品的Linux驱动程序来帮助解决这

个问题。而且现在许多定制的Linux发行版甚至会包括用于专业显卡的驱动程序。

一旦提供了X视窗接口，应用程序就可以使用它来和安装在Linux系统上的视频环境进行交互。流行的KDE和GNOME桌面就运行在X视窗环境之上，它们使用X视窗应用程序编程接口（API）调用来执行所有的绘图功能。

虽然X视窗系统对在视窗环境中创建程序是非常有用的，但它可以做的事情也有限。由于它是一个旧标准，X视窗并不能提供程序员在编写高级图形程序时所需的很多功能，尤其是处理三维对象的功能。对于这一类型的编程，你必须求助于其他软件包。

13.1.2 开放式图形库

开放式图形库（OpenGL）用于提供访问包含在高档3-D视频加速卡中的高级图形功能的通用API。X视窗标准致力于为程序员提供一个视窗环境，而OpenGL项目则着重于提供专用于绘制高级二维（2-D）和三维（3-D）对象的API。

与X视窗一样，OpenGL项目也不是一个特定的软件函数库。相反，它是一个通用规范，它定义了一个图形工作站上执行绘制对象的任务需要哪些函数。每一个获得OpenGL规范许可的显卡厂商都会为它们的特定显卡编写API。程序员可以使用同一个API与不同的OpenGL许可的显卡进行交互。

与使用针对特定显卡的特定OpenGL库相比，也可以使用一个完全由软件实现的通用OpenGL库。软件库将所有的2-D和3-D对象转换为标准的X视窗函数调用，这些函数调用可以由标准显卡进行处理。它们之间的关系如图13-2所示。

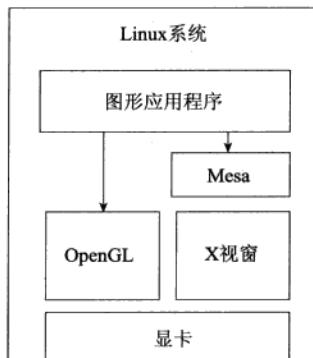


图13-2 标准的X视窗函数调用

Mesa程序是一个提供OpenGL函数库软件实现的流行软件包。对于那些没有安装高档3-D显卡但又想实现高级3-D图形的Linux系统来说，Mesa软件通常是默认安装的。Mesa软件还包括开发者函数库，可以利用它来创建自己的2-D和3-D应用程序而不需要购买高档的3-D加速卡。可以在Mesa中运行的任何软件程序也可以在高档显卡上使用OpenGL正常运行。

13.1.3 OpenGL 应用工具包

OpenGL库提供了用于高级2-D和3-D图形的函数，但不幸的是，它没有提供用于标准窗口编程的函数。为了帮助弥补OpenGL的这一缺点，Mark Kilgard开发了OpenGL应用工具包（GLUT）。GLUT

采用OpenGL规范并将它们与通用窗口函数集成以创建一个可以建立窗口、与用户交互、绘制2-D和3-D对象的完整函数库。

GLUT库成为OpenGL和X视窗系统之间的一个中间人。与OpenGL一样，许多显卡制造商为他们的显卡提供特定的GLUT库，Mesa软件也为普通显卡系统提供了完全由软件实现的GLUT。如果你正在使用OpenGL来创建图形对象，你很可能希望包括GLUT库以使你的工作更加轻松。

13.1.4 简单直接媒介层

如果你需要绘制多边形并让它们围绕一个轴进行旋转，那么OpenGL和GLUT库非常适合你。但在一些图形环境中，你需要做更多的事情。许多程序（尤其是游戏程序）除了绘制对象以外，还需要能够快速地在屏幕上显示完整的图像。此外，大多数游戏程序需要访问音频系统以播放声音效果和音乐，它们还需要能够访问键盘、鼠标和操纵杆以和玩家进行交互。使用GLUT来完成这些事情并非易事。

此时，就是更高层的游戏库发挥作用的时候了。用于Linux的最流行的全功能图形和音频库是简单直接媒介层（SDL）库。SDL库提供了许多高级函数用于操纵屏幕上的图像文件、通过图像文件创建动画、访问音频和处理键盘以及鼠标甚至操纵杆事件。

SDL库是一个跨平台的函数库。它有针对大多数操作系统的实现。你可以将自己在Linux系统上使用SDL开发的应用程序轻松地移植到Microsoft Windows或Apple Macintosh环境中。这一特征使得SDL库成为专业游戏程序员所喜爱的工具。许多商业游戏开发者都通过切换到SDL将Windows游戏移植到Linux环境中。

13.2 编写OpenGL应用程序

为了创建OpenGL应用程序，必须在Linux系统中同时安装OpenGL运行库和开发库，但要满足这一需求可能不容易。

OpenGL库并不是一个开放源码软件，它是一个受到硅谷图形公司（SGI）控制的授权规范。你必须拥有该软件的一个授权拷贝才能在自己的Linux系统上运行它。许多商业显卡厂商都提供了针对他们的特定显卡的授权OpenGL库。但是，如果你有一块不包含专用OpenGL驱动程序的显卡而你又想要进行OpenGL编程，那么你可以使用一个OpenGL的软件实现。

迄今为止，最流行的OpenGL软件实现是Mesa项目。作为对OpenGL库的简单实现，Mesa是由Brian Paul创建的一个开放源码项目。但Mark^①并没有宣称Mesa是一个OpenGL的兼容替代，因此Mesa不具备OpenGL许可。

Mesa提供了一个类似于OpenGL命令语法的环境，并且得到SGI的授权以开放源码的形式发布。使用Mesa不会违反任何许可证限制。可以使用Mesa编程环境开发可以在任何OpenGL实现中运行的OpenGL应用程序，而且可以在你的Linux系统上运行Mesa运行库而不需要来自SGI的许可。

正如我们在13.1.3节中所述的那样，OpenGL库并没有提供必需的窗口函数来编写有用代码。但幸运的是，Mesa项目同时也提供了GLUT库，这使得你仅使用一个开发环境就可以编写全功能的图形窗口应用程序了。

本章使用Mesa库来演示如何开发OpenGL应用程序。下面几节将描述如何下载和安装Mesa库以及如何使用它来创建可以运行在任何OpenGL实现之上的OpenGL应用程序。

① 这里的Mark指的是开发GLUT的Mark Kilgard。——译者注

13.2.1 下载和安装

许多Linux发行版包括Mesa的安装软件包，大多数Linux发行版默认都会安装Mesa运行库。这允许你在没有一块高档3-D显卡的情况下也可以运行OpenGL应用程序。

为了开发OpenGL应用程序，你很可能不得不自行安装Mesa开发头文件和库文件。这些文件通常被捆绑在名称为mesa-devel的安装软件包中。如果你找不到针对你的特定Linux发行版的Mesa开发安装软件包，或者你喜欢使用最新版本的软件包，那么你可以从Mesa网站上下载该软件包。

Mesa项目的网页位于www.mesa3d.org，在主页面中点击Downloading/Unpacking链接以找到下载页面。在该页面中，你可以下载最新版本的Mesa库。

有三个软件包可以下载。

- MesaLib-x.y.z:** 主Mesa OpenGL库文件。
- MesaDemos-x.y.z:** OpenGL示例程序集，用于演示OpenGL编程。
- MesaGLUT-x.y.z:** Mesa的OpenGL应用工具包实现。

x.y.z命名法标识Mesa库软件包的版本，其中x是主版本号，y是次版本号，z是补丁号。Mesa遵循如下的发布原则：次版本号为偶数的发行版是稳定版本，而次版本号是奇数的发行版是开发版本。在撰写本书时，Mesa库的最新稳定版本是6.4.2，最新开发版本是6.5.1。

为了在Linux系统上创建OpenGL应用程序，你至少应该下载MesaLib软件包和MesaGLUT软件包。它们为你提供了一个完整的OpenGL和GLUT开发环境以让你创建OpenGL应用程序。

你可以根据哪种压缩格式文件更容易在你的Linux发行版中使用来下载.tar.gz或.zip格式的文件。将软件包下载到一个工作目录中，并使用tar或unzip命令提取其中的文件。三个软件包都将文件释放到同一个目录Mesa-x.y.z中。

在下载和解压缩软件包之后，你可以开始编译函数库和示例程序（如果你还下载了Demo软件包）了。Mesa软件没有使用configure程序来确定编译环境，它使用的是多个make目标，每一个针对一种类型的支持环境。

要想看到所有可用的目标，可以在命令行中输入make并回车，它将输出所有可用的make目标列表。对于Linux系统来说，可以使用通用的linux目标或一个特定于你的环境的目标，如linux-x86。你只需在make命令之后输入适当的目标名即可：

```
$ make linux-x86
```

在编译好库文件之后，你可以选择将它们安装到标准的库文件和头文件目录中。在默认情况下，make install命令将把头文件安装到/usr/local/include/GL目录中，而库文件将被安装到/usr/local/lib目录中。

OpenGL所做的一件很棒的事情是可以在系统不同的位置安装多种版本的OpenGL库文件（例如一个专用于你的显卡的版本，再加上Mesa软件库）。当运行程序时，你可以选择使用哪个库来实现OpenGL函数并可以比较不同库的性能。

13.2.2 编程环境

在安装了Mesa OpenGL和GLUT的头文件和库文件之后，你已准备好开始编写程序了。你必须使用标准的#include语句将OpenGL和GLUT头文件包括进你的应用程序：

```
#include <GL/gl.h>
#include <GL/glut.h>
```

当你编译OpenGL应用程序时，必须包括Mesa OpenGL和GLUT头文件的路径并使用-I参数连接Mesa OpenGL和GLUT库：

```
$ cc -I/usr/local/include -o test test.c -L/usr/local/lib -lGL -lglut
```

请注意，Mesa OpenGL库（GL）以大写方式指定，而GLUT库则为小写。你还应该将/usr/local/lib目录包括进/etc/ld.so.conf配置文件以使得Linux可以找到OpenGL和GLUT动态库文件。不要忘记在修改过该文件后运行ldconfig命令。你现在已准备好开始创建OpenGL应用程序了。

13.2.3 使用 GLUT 库

正如你期望的那样，GLUT库包含用于创建窗口、绘制对象和处理用户输入的许多函数。本节将通过使用GLUT库创建下面一些简单的图形应用程序来演示其背后的一些基本原则：

- 创建一个窗口；
- 在窗口中绘制对象；
- 处理用户输入；
- 使你绘制的对象动起来。

在你可以绘制任何对象之前，必须先有一个工作空间。GLUT中使用的工作空间就是窗口。所以我们的第一个主题就是介绍如何使用GLUT库在标准X视窗环境中创建一个窗口。在创建了一个窗口之后，你就可以使用基本的GLUT库命令来绘制对象，如线、三角形、多边形和其他复杂的数学模型了。

当在一个视窗环境中工作时，如何捕获用户的输入是关键性问题。应用程序必须能够侦测到键盘事件以及任何鼠标移动。最后，因为每个游戏程序都需要动画，所以GLUT库提供了简单的动画命令来赋予对象生命。

1. 创建一个窗口

任何图形程序最基本的要素就是创建放置图形元素的窗口。我们通常使用5个函数来初始化GLUT环境并建立一个窗口：

```
void glutInit(int argc, char **argv);
void glutInitDisplayMode(unsigned int mode);
void glutInitWindowSize(int width, int height);
void glutInitWindowPosition(int x, int y);
int glutCreateWindow(char *name);
```

这一系列的函数调用几乎出现在每一个OpenGL程序中。glutInit()函数初始化GLUT库并使用传递给程序的命令行参数作为其参数来定义GLUT如何与X视窗系统进行交互。可以使用的命令行参数如下所示。

- display:** 指定要连接的X服务器。
- geometry:** 指定在X服务器上放置窗口的位置。
- iconic:** 要求所有的顶层窗口以图标状态创建。
- indirect:** 强制使用间接OpenGL渲染上下文。

- direct:** 强制使用直接OpenGL渲染上下文。
- gldebug:** 在每个OpenGL函数调用之后调用glGetError。
- sync:** 启用同步X视窗协议事务。

glutInitDisplayMode() 函数指定用于OpenGL环境的视频模式。显示模式通过对一系列标记值进行或操作来定义，如表13-1所示：

表13-1 定义显示模式的一系列标记值

标记	说明
GLUT_SINGLE	提供一个单缓冲窗口
GLUT_DOUBLE	提供可以交换的双缓冲窗口
GLUT_RGBA	基于红-绿-蓝(RGB)值控制窗口颜色
GLUT_INDEX	基于颜色索引控制窗口颜色
GLUT_ACCUM	为窗口提供一个积聚缓冲
GLUT_ALPHA	为窗口颜色提供一个alpha透明度分量
GLUT_DEPTH	提供一个带有颜色深度缓冲的窗口
GLUT_STENCIL	提供一个带有模具(stencil)缓冲的窗口
GLUT_MULTISAMPLE	提供一个带有多级采样支持的窗口
GLUT_STEREO	提供一个立体窗口缓冲
GLUT_LUMINANCE	为窗口颜色提供一个亮度分量

glutInitWindowSize() 函数和 glutInitWindowPosition() 函数所做的事情正如其名字所说的那样，它们允许设置初始窗口大小（以像素为单位）和位置（根据窗口坐标系统）。窗口坐标系统的原点(0,0)位于窗口显示的左下角。X轴向窗口的下方伸展，而Y轴则位于窗口的左侧。

glutCreateWindow() 函数在屏幕上创建实际的窗口。其使用的参数指定窗口的名称，该名称将显示在窗口顶部的标题栏中。

在窗口创建之后，真正的乐趣才能显现。GLUT和所有其他基于窗口的编程语言一样都使用事件驱动编程。与在程序代码中线性步进以执行命令不同，GLUT在创建的窗口中等待事件的发生。当一个事件发生时，它将执行绑定到该事件上的函数（称为回调函数），然后等待下一次事件发生。

一个视窗程序中的所有工作都发生在回调函数中。你必须提供在你的应用程序之外所发生事件的处理代码（如用户点击鼠标按钮或按下键盘上的一个按键）。GLUT提供了几个函数调用使得你可以注册自己的回调函数，当某个特定事件发生时，它们将被GLUT调用，如表13-2所示：

表13-2 GLUT提供的可以让用户注册回调函数的函数调用

函数	说明
GlutDisplayFunc(void (*func)(void))	指定当窗口中的内容需要重新绘制时要调用的函数
GlutReshapeFunc(void (*func) (int width, int height))	指定当窗口被调整大小或移动时要调用的函数
GlutKeyboardFunc(void (*func) (unsigned int key, int x, int y))	指定当用户按下键盘上的一个ASCII键时要调用的函数
GlutSpecialFunc(void (*func) (int key, int x, int y))	指定当用户按下键盘上的一个特殊键（如F1~F12）或一个方向键时要调用的函数

(续)

函数	说明
GlutMouseFunc(void (*func) (int button, int state, int x, int y))	指定当鼠标按钮被按下或释放时要调用的函数
GlutMotionFunc(void (*func) (int x, int y))	指定当鼠标指针移动到窗口中时要调用的函数
GlutTimerFunc(int msec, void (*func)(int value), value)	指定在msec毫秒之后要调用的函数，并传递value值作为其参数
GlutPostRedisplay(void)	将当前窗口标记为需要重新绘制

如果不注册与事件相关的函数，GLUT就不会将这类事件的发生传递给程序，并将忽略这些事件。注册事件函数后，必须使用glutMainLoop()函数启动GLUT以等待事件的发生。

代码清单13-1中的testwin.c程序演示了如何在GLUT环境中创建一个窗口并在窗口中绘制一个简单的对象。

代码清单13-1 一个示例OpenGL窗口程序

```

/*
 *
 * Professional Linux Programming - OpenGL window test
 *
 */
#include <GL/gl.h>
#include <GL/glut.h>

void display(void)
{
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glRotatef(30, 1.0, 0.0, 0.0);
    glRotatef(30, 0.0, 1.0, 0.0);
    glutWireTetrahedron();
    glFlush();
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Test Window");

    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```

testwin.c程序的main()部分使用标准的GLUT初始化函数创建显示OpenGL绘图的窗口。一个回调函数被注册来处理窗口显示，然后启动GLUT的主事件循环。

回调函数display用于控制窗口中放置些什么，这是程序的真正功能之所在。在窗口中放置任何东西之前先清理已有的东西总是一个好主意。glClear()函数用于清理窗口，可以使用glClearColor()函数设置用于清理窗口的背景色。

glClearColor()函数使用4个参数：

```
glClearColor(float red, float green, float blue, float alpha)
```

red、green和blue参数指定添加到背景色中的红、绿、蓝颜色的数量。0.0表示没有，而1.0表示全色。如果你和我一样不知道如何使用红、绿、蓝组合来创建颜色，请参考表13-3，该表列出了一些基本的颜色组合：

表 13-3

红	绿	蓝	颜色
0.0	0.0	0.0	黑
1.0	0.0	0.0	红
0.0	1.0	0.0	绿
1.0	1.0	0.0	黄
0.0	0.0	1.0	蓝
1.0	0.0	1.0	洋红
0.0	1.0	1.0	青
1.0	1.0	1.0	白

请记住，这些都是浮点值，所以在0.0和1.0之间有很多颜色，但这里列出的信息应该给你提供了一个如何创建一些基本颜色的思路。alpha参数控制应用到颜色上的透明度。0.0表示全透明，而1.0表示完全不透明。

在设置背景色和清除背景之后，可以开始你的绘制了。绘制对象的颜色由glColor3f()函数设置。这个函数使用3个RGB值（红、绿、蓝）来定义颜色，它们的浮点数取值范围从0.0到1.0。上面显示的glClearColor()函数的颜色组合也同样适合于这个函数。

在设置颜色之后，现在是时候绘制一个对象了。这个简单的例子使用了一个来自GLUT库的内置几何对象。有多种类型的内置形状可以使用，其范围从简单对象（如立方体、球体、圆锥体）一直到更高级的对象（如四面体、八面体和十二面体）。glutWireTetrahedron()函数用于创建一个四面体的轮廓。在默认情况下，对象将以轴原点为中心显示。为了增添一点趣味，我们使用glRotatef()函数旋转图像的视图。glRotatef()函数使用4个参数：

```
glRotatef(float angle, float x, float y, float z)
```

第一个参数angle指定旋转对象的度数（我们选择旋转30度）。接下来三个参数指定从原点开始的一个坐标点方向作为对象的旋转轴方向。两个glRotatef()函数调用先绕着X轴旋转对象30度，然后再绕着Y轴旋转对象30度。这允许你获得对象的更真实的3-D感觉。glFlush()函数用于确保在回调函数调用的最后将绘图发送到显示器。

当你输入testwin.c的代码之后，你必须针对你的系统对它进行编译：

```
$ cc -I/usr/local/include -o testwin testwin.c -L/usr/local/lib -lGL -lglut
```

为了运行这个程序，必须在系统上打开一个X视窗会话。如果你正在使用一个桌面管理系统（如KDE或GNOME），可以打开一个命令行窗口以启动testwin程序。程序窗口以及其中的对象应该出现在你的桌面上，如图13-3所示。

因为这个测试程序没有注册处理鼠标或键盘事件的回调函数，所以你按键或点击鼠标都不会影响这个程序的窗口。你可以通过关闭窗口来停止程序。你需要注意的一件事情是窗口重新绘制图像的方式。每当窗口需要被重新绘制时就会调用display回调函数，例如当窗口被移动或调整大小时。每当这样的事情发生时，glRotatef()函数将再将图像旋转30度。通过调整窗口的大小，你可以看到图像绕着轴进行旋转。在本章的后面你将学习如何使图像自动移动。

2. 绘制对象

你已看到如何创建一个窗口并在窗口中绘制对象了，现在是时候学习你可以绘制些什么了。除了内置的3-D对象以外，GLUT使用OpenGL库来绘制2-D对象和3-D对象。

在OpenGL中，所有的对象都通过1个或多个顶点来定义。1个顶点是由 glVertex() 函数创建的。该函数可以以几种不同的格式定义：

```
void glVertex{234}{sifd}[v](TYPEcoords)
```

正如你看到的那样，这个函数可以使用的函数名称有好几个版本。函数名本身说明了 glVertex() 对象定义的方式：

- 使用多少个坐标（2, 3或4）。
- 指定坐标的数据类型（短整型、整型、双精度或浮点数）。
- 指定坐标是一组顶点还是一个向量。

在默认情况下，OpenGL将所有顶点指定为一个3-D对象的顶点(x,y,z)。如果你只绘制一个2-D对象，可以使用二维坐标选项，Z轴值假设为0。例如，要使用两个整数坐标来定义一个2-D顶点，你可以使用：

```
glVertex2i(3, 6);
```

如果你想使用浮点数坐标来定义一个顶点，可以使用：

```
glVertex3f(1.0, 0.0, 3.4);
```

你还可以使用一个向量来定义顶点：

```
Gldouble vec[3] = {1.0, 0.0, 3.4};  
glVertex3fv(vec);
```

一组glVertex()顶点定义一个对象。它们需要使用glBegin()和glEnd()组合进行定义：

```
glBegin(mode);  
glVertex2i(0,0);  
glVertex2i(1,0);  
glVertex2i(0,1);  
glEnd();
```

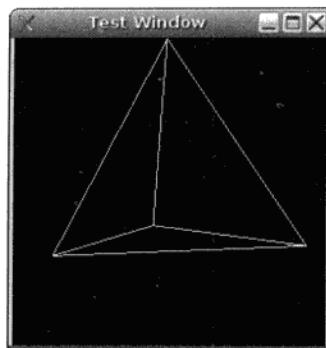


图13-3 testwin程序的执行结果

mode参数定义顶点之间的连接方式。表13-4列出了可以使用的几种基本类型：

表 13-4

基本类型	说 明
GL_POINTS	单个、未连接顶点集
GL_LINES	两个顶点解释为一个线段
GL_LINE_STRIP	一连串两个或多个连接线段
GL_LINE_LOOP	一连串两个或多个连接线段，第一个和最后一个线段也连接在一起，即闭合折线
GL_TRIANGLES	三个顶点解释为一个填充三角形
GL_TRIANGLE_STRIP	一连串两个或多个连续填充三角形
GL_TRIANGLE_FAN	一连串两个或多个连续填充三角形，第一个和最后一个三角形也连接在一起
GL_QUADS	四个顶点解释为一个填充四边形
GL_QUAD_STRIP	一连串两个或多个连续填充四边形
GL_POLYGON	五个或更多个顶点解释为一个简单、凸填充多边形的边界

请注意，GL_LINE_STRIP类型和GL_LINE_LOOP类型只创建一个对象的轮廓，而其他类型创建的是一个填充对象。填充对象的内部将使用与每个顶点所选择颜色相匹配的颜色进行填充。

还需注意，同一个顶点集可以用几种不同的方式进行解释，这取决于定义的基本类型。三个顶点可以被解释为三个单独的顶点、两个线段、由三个线段组成的闭合折线或一个填充三角形对象。

当创建一个填充对象时，需要小心glVertex()顶点的定义顺序。对象将以顶点被指定的顺序进行绘制。在绘制对象时，如果出现交叉线将导致一个不匹配的对象。

glVertex()函数使用一个坐标系统来定义顶点的位置。这一过程中比较棘手的部分是知道当你绘制对象时所使用的坐标系统是什么。这是一个稍微复杂的话题，尤其是当工作在窗口上时更是如此。

每当调整窗口大小时，窗口中使用的坐标就会改变。为了使事情变得简单一些，OpenGL允许你在窗口中设置一个视口。视口允许你将一个通用坐标系统映射到物理窗口中分配的可用空间中。

为了在每次调整窗口大小时设置视口，必须使用glutReshapeFunc()注册一个回调函数。请记住，每当窗口大小改变时，glutReshapeFunc()回调函数就将被调用。包含在回调函数中的代码如下所示：

```
void resize(int width, int height)
{
    glViewport(0, 0, (GLsizei)width, (GLsizei)height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) width, 0.0, (GLdouble)height);
}
```

glViewport()函数调整坐标系统到窗口的新大小 (width和height值由glReshapeFunc()函数传递)。glMatrixMode()函数用于定义3-D对象是如何显示在2-D显示器上的。GL_PROJECTION矩阵模式将3-D图像投影到2-D屏幕上，就如同照相机将一个3-D图像投影到2-D照片上一样。你也可以尝试使用GL_MODELVIEW矩阵模式以获得一个略微不同的效果。glLoadIdentity()函数用于建立用在窗口中的矩阵模式。gluOrtho2D()函数用于设置视口中原点 (0,0) 的位置。上面的格式将原点设置在窗口的左下角。该函数并没有用在所有的绘图程序中，因为它与GLUT坐标系统冲突。如果必须使用GLUT处理鼠标事件，请不要使用gluOrtho2D()来重新映射OpenGL坐标，因为这很容易把你弄糊涂。

你已看到了绘制一个对象所需的代码，现在是时候看一个例子了。代码清单13-2中的testdraw.c程序演示了如何使用OpenGL绘制一个简单的2-D对象。

代码清单13-2 使用OpenGL绘制对象

```
/*
 * Professional Linux Programming - OpenGL drawing test
 */
#include <GL/gl.h>
#include <GL/glut.h>

void display(void)
{
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_LINE_LOOP);
    glColor3f(1.0, 1.0, 1.0);
    glVertex2i(20,30);
    glVertex2i(60,30);
    glVertex2i(20,60);
    glEnd();

    glBegin(GL_TRIANGLES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex2i(180, 30);
    glVertex2i(220, 30);
    glVertex2i(180, 60);

    glEnd();

    glBegin(GL_QUADS);
    glColor3f(1.0, 1.0, 0.0);
    glVertex2i(20, 130);
    glVertex2i(20, 160);
    glVertex2i(60, 160);
    glVertex2i(60, 130);
    glEnd();

    glBegin(GL_LINE_LOOP);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2i(180, 130);
    glVertex2i(180, 160);
    glVertex2i(220, 160);
    glVertex2i(220, 130);
    glEnd();

    glFlush();
}

void resize(int width, int height)
```

```

{
    glViewport(0, 0, (GLsizei)width, (GLsizei)height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble)width, 0.0, (GLdouble)height);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Test Drawing");

    glutDisplayFunc(display);
    glutReshapeFunc(resize);
    glutMainLoop();

    return 0;
}

```

testdraw.c程序中的第一件值得注意的事情是它的main()部分与testwin.c程序几乎完全一样。唯一的不同是添加了glutReshapeFunc()函数。由于所有的实际工作都是在display()回调函数中完成的，所以你通常可以为所有OpenGL程序的main()部分使用一个标准模板。

此外，你还可以注意到resize()回调函数的内容和本节前面展示的完全一样。用于调整视口大小的回调函数在所有的OpenGL程序中通常也是一样的。

display()回调函数将首先清理窗口，然后定义要在窗口中绘制的四个不同类型的对象。在对象定义的结尾，glFlush()函数用于确保新的对象显示在窗口中。

在编译程序之后，尝试在你的系统中运行它。你应该看到一个如图13-4所示的输出窗口。

你可以尝试使用 glVertex() 函数的顶点定义和 glBegin() 函数的基本类型来创建自己的形状和对象。更复杂的对象可以通过重叠几个基本形状来创建。

3. 处理用户输入

下一步是处理用户输入。正如13.2.3节所述，必须提供一个回调函数让GLUT在一个键盘或鼠标事件被侦测到时调用。

glutKeyboardFunc()函数和glutSpecialFunc()函数定义当一个键盘事件被处理时要调用的回调函数。glutKeyboardFunc()函数监听键盘上标准的ASCII键被按下或释放，而glutSpecialFunc()函数只监听特殊键，如F1-F12功能键、Escape键(ESC)或任意一个方向键、Home、End、PgDn或PgUp键。glutKeyboardFunc()函数的格式为：

```
glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))
```

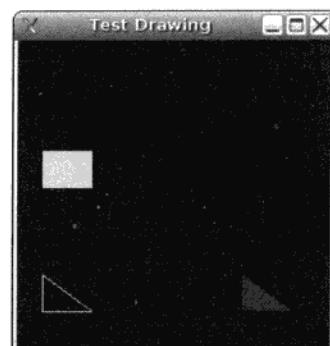


图13-4 testdraw.c程序的输出结果

回调函数被定义为func。glutKeyboardFunc()函数传递按下的键key以及当键被按下时窗口中鼠标指针的x和y坐标。

当一个使用Shift、Ctrl或Alt的组合键被选择时，它的ASCII键或特殊键的值被用于触发回调函数。glutGetModifiers()函数用于在回调函数中确定哪个组合键被按下以及它的ASCII或特殊键值。该函数返回一个整数值，我们可以用这个返回值与GLUT_ACTIVE_SHIFT、GLUT_ACTIVE_CTRL或GLUT_ACTIVE_ALT值进行比较以确定用户选择了哪个组合键。

同样地，我们可以使用glutMouseFunc()和glutMotionFunc()函数来跟踪窗口中的鼠标按键和移动。glutMouseFunc()函数的格式为：

```
glutMouseFunc(void (*func)(int button, int state, int x, int y))
```

这个函数向你的回调函数func发送一个整数值button以确定用户按下的鼠标按钮(GLUT_LEFT_BUTTON、GLUT_MIDDLE_BUTTON或GLUT_RIGHT_BUTTON)、一个整数值state以显示状态(GLUT_UP或GLUT_DOWN)、x和y坐标值以显示当鼠标按钮被按下时鼠标指针的位置。鼠标指针位置使用的坐标系统以屏幕的左上角为原点。这通常会让人感到困惑，所以，如果你必须处理鼠标位置，请小心。

代码清单13-3中的testkey.c程序演示了如何捕获键盘按键以控制一个绘制对象的移动。

代码清单13-3 在OpenGL中处理键盘事件

```
/*
 *
 * Professional Linux Programming - OpenGL keyboard test
 *
 */
#include <GL/gl.h>
#include <GL/glut.h>

int xangle = 0;
int yangle = 0;

void display(void)
{
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glRotatef(xangle, 1.0, 0.0, 0.0);
    glRotatef(yangle, 0.0, 1.0, 0.0);
    glutWireCube(1);
    glFlush();
}

void keyboard(unsigned char key, int x, int y)
{
    if (key == 'q' || key == 'Q')
        exit(0);
}

void special(int key, int x, int y)
```

```

{
    switch(key)
    {
        case(GLUT_KEY_UP):
            xangle += 15;
            break;
        case(GLUT_KEY_DOWN):
            xangle -= 15;
            break;
        case(GLUT_KEY_RIGHT):
            yangle += 15;
            break;
        case(GLUT_KEY_LEFT):
            yangle -= 15;
            break;
    }
    glutPostRedisplay();
}

void resize(int width, int height)
{
    glViewport(0, 0, (GLsizei)width, (GLsizei)height);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Test Keyboard");

    glutDisplayFunc(display);
    glutReshapeFunc(resize);
    glutKeyboardFunc(keyboard);
    glutSpecialFunc(special);
    glutMainLoop();

    return 0;
}

```

`testkey.c`程序使用两个全局变量`xangle`和`yangle`来控制绘制对象的视角。它们的初始值被设置为0。`display()`回调函数控制窗口中对象的绘制（使用`glutWireCube()`函数创建一个立方体）。`resize()`回调函数的内容和其他示例中的内容几乎完全一样，但本例中没有使用`gluOrtho2D()`函数，这是为了保持轴的原点位于屏幕的中间以使得旋转可以正常工作。

`keyboard()`回调函数和`special()`回调函数用于控制当一个键被按下时做什么。`keyboard()`回调函数只检查按键是否为`q`或`Q`，所有其他的键都被忽略。如果按键是`q`或`Q`，那么程序终止。

`special()`回调函数使用预定义的GLUT值来检查是否有一个方向键被按下了。它将针对每一个方向键使用相应的全局变量来改变针对一个特定轴的旋转角度。（注意，这并不是一个用于改变旋转

的正确数学模型，因为前一个旋转值并没有被清除。)

在旋转角度被改变之后，glutPostRedisplay()函数被调用。这个函数强制OpenGL重新绘制窗口的内容，这将再次使用display()回调函数。因为旋转角度改变了，所以当对象被重新绘制时，图像的视图也将旋转。

在编译程序之后，运行它并观察窗口。窗口中将出现一个立方体，你可以通过按下方向键来控制立方体的视角。按下q键将关闭窗口并彻底终止程序。

4. 动画

迄今为止，你已看到了如何在窗口大小改变或用户按下键盘上某个键或鼠标按钮时改变一个绘制的图像。但对于许多应用程序（如游戏）来说，即使什么也没有发生，也需要移动对象。OpenGL库提供了自动让你绘制的对象动起来所需的工具。

glutTimerFunc()函数允许你指定一个回调函数并以毫秒为单位指定在多长时间之后调用该回调函数。程序将等待指定的时间，然后触发回调函数。这允许你在程序中以预定的时间间隔执行特定的函数。

定时器回调函数的内容通常会包含一个glutPostRedisplay()函数以强制使用新的图像重新绘制窗口。定时器函数glutTimerFunc()还可以被添加到display()回调函数中以触发另一个定时器来再次更新窗口显示。

代码清单13-4中的testtimer.c程序演示了如何使用glutTimerFunc()函数来让窗口中绘制的对象动起来。

代码清单13-4 为OpenGL中的对象赋予生命

```
/*
 *
 * Professional Linux Programming - OpenGL timer test
 *
 */
#include <GL/gl.h>
#include <GL/glut.h>

int millisec = 100;

void timer(int value)
{
    glutPostRedisplay();
}

void display(void)
{
    glutTimerFunc(millisec, timer, 1);
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glRotatef(1, 0.0, 1.0, 1.0);
    glutWireCube(1);
    glutSwapBuffers();
}
```

```

void resize(int width, int height)
{
    glViewport(0, 0, (GLsizei)width, (GLsizei)height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
    if (key == 'q' || 'Q')
        exit(0);
}

void special(int key, int x, int y)
{
    switch(key)
    {
        case(GLUT_KEY_UP):
            if (millisec >= 10)
                millisec -= 10;
            break;
        case(GLUT_KEY_DOWN):
            millisec += 10;
            break;
        case(GLUT_KEY_HOME):
            millisec = 100;
    }
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Test Window");

    glutDisplayFunc(display);
    glutReshapeFunc(resize);
    glutKeyboardFunc(keyboard);
    glutSpecialFunc(special);
    glutMainLoop();

    return 0;
}

```

你应该能看懂testtimer.c程序中的大部分功能。display()回调函数使用glutWireCube()函数显示一个简单的立方体对象并使用glRotatef()函数将它绕X轴旋转1度。请注意，这里没有使用标准的glFlush()函数将图像显示到窗口中，而是使用了另一个函数。

glutSwapBuffer()函数是一个由GLUT提供的便利函数，它允许你操纵两个显示缓冲区。当在main()函数中调用glutInitDisplayMode()函数时，它指定了GLUT_DOUBLE参数，这将创建两个显示缓冲区。其中一个缓冲区总是活跃的，它的内容显示在窗口中；另一个缓冲区在当新对象被绘制时

使用。任何OpenGL绘图函数的执行都是在闲置缓冲区中完成的。`glutSwapBuffer()`立刻将闲置缓冲区中的内容交换到活跃缓冲区。与试图直接更新活跃屏幕缓冲区相比，这有助于更平滑地实现屏幕上的动画。

此外，还请注意`display()`回调函数中的第一个语句是对`glutTimerFunc()`函数的调用。它将以一个预设的间隔（100毫秒）启动一个定时器并将在定时器到期时调用`timer()`回调函数。`timer()`回调函数的唯一工作就是触发`glutPostRedisplay()`函数，该函数将强制再次执行`display()`回调函数，而`display()`反过来又启动另一个`glutTimerFunc()`。正是这个循环使得窗口图像每隔100毫秒就被重新绘制。

`special()`回调函数用于处理向上、向下方向键。如果向上方向键被按下，`timer()`延迟值将减少以加快动画的刷新率。如果向下方向键被按下，`timer()`延迟值将增加以降低动画刷新率。如果Home键被按下，则延迟值将恢复到100毫秒。

编译并在你的系统中运行这个程序，你将看到出现一个窗口，在窗口中有一个不断旋转的立方体。请注意，这个立方体的旋转非常平滑。按下向上的方向键将增加它的旋转速率直至最后因没有延迟而造成一片模糊。为应用程序确定最优的刷新率通常是一件很棘手的事情。大多数商业品质的动画都使用每秒24帧的刷新率。

13.3 编写SDL应用程序

虽然OpenGL和GLUT库在绘制2-D和3-D对象方面非常出色，但它们有其局限性。对于更高级的图形编程和在你的应用程序中集成音频来说，SDL库提供的函数将使你的工作变得更轻松。

本节将介绍如何下载并在你的Linux系统上安装SDL库，本节还将演示SDL库的一些功能。

13.3.1 下载和安装

SDL库可以从SDL网站www.libsdl.org上下载。网站主页面包含一个到最新版本下载页面的链接（当前版本是SDL-1.2.11^①）。在下载页面中，你既可以下载SDL的二进制发行版也可以下载完整的源代码软件包并进行编译。源代码文件的格式有两种：`.tar.gz`压缩打包格式和`zip`格式。

如果你选择自己从源代码开始编译SDL库，请下载源代码发行版文件到一个工作目录中，然后使用适当的方法解压缩文件：

```
$ tar -zxvf SDL-1.2.11.tar.gz
```

这将在当前工作目录下创建一个名称为SDL-1.2.11的目录。SDL发行版使用标准的`configure`程序来检测平台相关的编译配置。首先使用`./configure`命令，然后使用`make`命令来编译SDL库：

```
$ ./configure
$ make
$ su
Password:
# make install
#
```

SDL头文件被安装到`/usr/local/include/SDL`目录中，而库文件则被安装到`/usr/local/lib`

^① 译者翻译本书时SDL库的最新版本是1.2.12。

目录中。

13.3.2 编程环境

为了使用SDL库创建程序，必须在编译程序时包括SDL头文件和库文件的路径以及连接SDL库：

```
$ cc -I/usr/local/include -o test test.c -L/usr/local/lib -lSDL
```

另外，在应用程序中，必须使用`#include`语句包括SDL/SDL.h头文件。

请确保你的库环境使用`/usr/local/lib`目录引用动态库，它可以在`/etc/ld.so.conf`文件中找到。

13.3.3 使用SDL库

SDL库包含了用于控制视频、音频、键盘、鼠标、操纵杆和CD-ROM（如果系统中有一个光驱）的函数。在你可以使用这些函数之前，必须在你的应用程序中初始化SDL库。而且，必须在程序的结尾关闭SDL库。

`SDL_Init()`函数用于初始化SDL库，而`SDL_Quit()`函数用于关闭它。当你初始化SDL库时，你必须指明将使用这个库的哪些部分。

- `SDL_INIT_VIDEO`: 初始化视频函数。
- `SDL_INIT_AUDIO`: 初始化音频函数。
- `SDL_INIT_CDROM`: 初始化CD-ROM函数。
- `SDL_INIT_TIMER`: 初始化定时器函数。

你可以使用OR操作来初始化一个以上的部分，如下所示：

```
#include "SDL.h"

int main(int argc, char**argv)
{
    if(SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO) < 0
    {
        perror("SDLInit");
        exit(1);
    }
    ...
    SDL_Quit();
    Return 0;
}
```

这个代码初始化SDL库的视频和音频部分。以下各节将演示使用SDL库可以实现的一些功能。

1. 显示一个图像

虽然可以在窗口上手工绘制图像非常好，但对于大图像来说，这种绘制方法显得比较麻烦。如果能够直接在窗口中装载和操纵一个预制作的图像通常会显得更加方便。SDL库就可以让你做到这一点。

SDL在一个`SDL_Surface`对象上显示图像。`SDL_Surface`对象显示在一个可视屏幕上，这个可视屏幕可以是一个窗口中或是整个屏幕。一个图像装载进一个`SDL_Surface`对象，多个图像就装载进多个`SDL_Surface`对象。然后，`SDL_Surface`对象可以快速地换进和换出可视屏幕。

在显示图像之前，必须设置用于图像的视频模式。SDL_SetVideoMode()函数就用于这个目的。它的格式为：

```
SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags)
```

SDL_SetVideoMode()函数返回的SDL_Surface指针成为一个活跃的可视表面（surface）。对象必须被装载进这个表面才能在屏幕上看到。这个函数调用请求一个特定的视频宽度、高度和每像素位数（bpp）。flags参数定义请求的额外视频模式特征。flags可以定义为表13-5中标记的一个或多个：

表13-5 flags参数的标记

标记	说明
SDL_SWSURFACE	在系统内存中创建视频表面
SDL_HWSURFACE	在显卡的内存中创建视频表面
SDL_ASYNCBLIT	启用异步位块传送 ^① ，用于多处理器机器
SDL_ANYFORMAT	如果请求的bpp不可用，SDL将使用可用的视频bpp
SDL_HWPALETTE	给予SDL独占式的显卡调色板访问
SDL_DOUBLEBUF	启用双缓冲。只有和SDL_HWSURFACE一起使用才有效
SDL_FULLSCREEN	尝试使用全屏模式
SDL_OPENGL	创建一个OpenGL渲染上下文
SDL_OPENGLBLIT	创建一个OpenGL渲染上下文，但允许图像位块传送
SDL_RESIZABLE	创建一个可调整大小的窗口。在默认情况下，窗口是不可调整大小的
SDL_NOFRAME	如果可能，SDL将创建一个没有标题栏或无框架的窗口

函数调用中定义的视频模式并不一定要是视频系统所允许的模式。“求你所需，用你所得”这一格言经常被用在图形编程中。我们可以检查由SDL_SetVideoMode()函数返回的SDL_Surface对象以确定什么是视频系统真正允许的模式。

在创建了一个SDL_Surface之后，你可以将一个图像装载进一个闲置的SDL_Surface对象，然后再将该对象转移到一个活跃的SDL_Surface对象中。这一过程被称为blitting。第一步，将一个图像装载进一个SDL_Surface对象是由SDL_LoadBMP()函数完成的：

```
SDL_Surface *SDL_LoadBMP(const char *file)
```

图像必须是.bmp格式的，它由file参数指定。如果在将图像装载进表面时出现错误，这个函数将返回一个NULL值。

一旦新图像被装载进一个SDL_Surface对象，你就可以使用SDL_BlitSurface()函数将该图像传送到一个活跃的SDL_Surface对象中：

```
int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstrect)
```

正如预期的那样，src和dst参数分别代表源SDL_Surface对象和目标（活跃）SDL_Surface对象。两个SDL_Rect参数定义一个以像素为单位的矩形区域作为传送区域。SDL_Rect对象是一个结构，它的定义为：

^① “异步位块传送”的原文为“asynchronous blitting”，blitting的含义是指将一个平面的一部分或全部图像整块从这个平面复制到另一个平面，这里将blitting译为位块传送。

```
typedef struct {
    Sint16 x, y;
    Uint16 w, h;
} SDL_Rect;
```

w和h成员定义了对象的宽度和高度，而x和y成员定义了矩形左上角的坐标。

如果你只装载新图像的一部分到屏幕上，可以使用SDL_Rect来定义矩形区域。如果你装载整个图像到屏幕上，可以为srcrect参数指定一个NULL值。dstrect参数则稍微有点麻烦。它指定新图像将被装载到屏幕的那个位置。我们通常将宽度(w)值和高度(h)值设置为图像的宽度和高度，而x和y值则取决于你想要把图像放到屏幕的那个位置。

开始时，这一切看起来似乎让人困惑，但它并非想象的那么糟糕。要创建一个屏幕、载入一个图像，然后将它置于屏幕的中央，可以使用如下的代码：

```
SDL_Surface *screen, *image;
SDL_Rect location;
screen = SDL_SetVideoMode(800, 600, 0, 0);
image = SDL_LoadBMP("test.bmp");
location.x = (screen->w - image->w) / 2;
location.y = (screen.h - image.h) / 2;
location.w = image->w;
location.h = image->h;
SDL_BlitSurface(image, NULL, screen, &location);
SDL_UpdateRects(screen, 1, &location);
```

location对象定义新图像传送到活跃屏幕上的方式。location的x和y值被设置为将新图像置于屏幕的中央。请注意，在调用了SDL_BlitSurface()函数之后，SDL_UpdateRects()函数用于对活跃的SDL_Surface对象执行实际的更新。正如我们所提到的那样，不需要在没有必要的情况下更新整个屏幕图像。

代码清单13-5中的imagetest.c程序演示了如何使用SDL将图像装载到一个屏幕上。

代码清单13-5 使用SDL显示图像文件

```
/*
 *
 * Professional Linux Programming - SDL image test
 *
 */
#include <stdio.h>
#include <SDL/SDL.h>

int main(int argc, char **argv)
{
    SDL_Surface *screen, *image1, *image2;
    SDL_Rect location1, location2;
    int delay = 4;

    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        perror("SDLInit");
        return 1;
    }
    printf("SDL initialized.\n");
```

```
screen = SDL_SetVideoMode(800, 600, 0, 0);
if (screen == NULL)
{
    printf("Problem: %s\n", SDL_GetError());
    return 1;
}
printf("Video mode set.\n");

SDL_WM_SetCaption("SDL test image", "testing");
image1 = SDL_LoadBMP("image1.bmp");
if (image1 == NULL)
{
    printf("Problem: %s\n", SDL_GetError());
    return 1;
}
printf("image1 loaded.\n");

image2 = SDL_LoadBMP("image2.bmp");
if (image2 == NULL)
{
    printf("Problem: %s\n", SDL_GetError());
    return 1;
}
printf("image2 loaded.\n");

location1.x = (screen->w - image1->w) / 2;
location1.y = (screen->h - image1->h) / 2;
location1.w = image1->w;
location1.h = image1->h;

location2.x = (screen->w - image2->w) / 2;
location2.y = (screen->h - image2->h) / 2;
location2.w = image2->w;
location2.h = image2->h;

if (SDL_BlitSurface(image1, NULL, screen, &location1) < 0)
{
    printf("Problem: %s\n", SDL_GetError());
    return 1;
}
SDL_UpdateRects(screen, 1, &location1);
SDL_Delay(delay * 1000);

if (SDL_BlitSurface(image2, NULL, screen, &location2) < 0)
{
    printf("Problem: %s\n", SDL_GetError());
    return 1;
}
SDL_UpdateRects(screen, 1, &location2);
SDL_Delay(delay * 1000);
SDL_Quit();
return 0;
}
```

`imagetest.c`程序使用`SDL_SetVideoMode()`函数请求一个 800×600 大小的屏幕区域，使用系统默认的`bpp`和额外视频模式。对于大多数系统来说，默认值将产生一个在目前正在运行的X视窗系统中的窗口。如果你想让程序使用全屏模式，则需要将该函数的最后一个参数0替换为`SDL_FULLSCREEN`。

两个`SDL_LoadBMP()`函数将两张图像(`image1.bmp`和`image2.bmp`)分别装载进两个`SDL_Surface`对象。为了完成这一任务，你需要准备好两张图像(命名为`image1.bmp`和`image2.bmp`)。在每张图像都被载入之后，图像的`SDL_Rect`位置被计算以将图像置于可视屏幕的中央。然后，`SDL_BlitSurface()`和`SDL_UpdateRects()`函数用于装载每张图像。`SDL_Delay()`函数提供了一个容易使用的定时器。程序使用它在显示每个图像之间停顿4秒钟。

编译这个应用程序并使用两张测试图像来运行它。请注意，这两张图像之间的切换非常快速(尤其当你使用的是一块高级显卡时更是如此)。通过这个程序，你不难想象如何在屏幕上实现数张图像的快速切换。

2. 播放一个音频文件

每一个成功的游戏都需要有好的声音效果。在Linux系统中，将音频文件发送给音频设备有时是一件比较棘手的事情。SDL库提供了一个出色的接口用于在任何支持音频的操作系统上方便地播放音频文件。

SDL使用一个`SDL_AudioSpec`结构处理音频文件。这个结构的格式为：

```
typedef struct {
    int freq;
    Uint16 format;
    Uint8 channels;
    Uint8 silence;
    Uint16 samples;
    Uint32 size;
    void (*callback)(void *userdata, Uint8 *stream, int len);
    void *userdata;
} SDL_AudioSpec;
```

`SDL_AudioSpec`结构包含处理一个音频文件所需的所有信息。其结构成员见表13-6：

表13-6 `SDL_AudioSpec`的结构成员列表

成 员	说 明
<code>freq</code>	每秒音频频率取样
<code>format</code>	音频数据格式
<code>channels</code>	声道数(1为单声道，2为立体声)
<code>silence</code>	音频缓冲静音值
<code>samples</code>	以取样值为单位的音频缓冲区大小
<code>size</code>	以字节为单位的音频缓冲区大小
<code>callback</code>	用于填充音频缓冲区的回调函数
<code>userdata</code>	传递给回调函数的音频数据的指针

大多数`SDL_AudioSpec`的值都是在音频文件被装载时自动填充的。`SDL_LoadWAV()`函数用于将一个.wav音频文件装载进`SDL_AudioSpec`结构：

```
SDL_AudioSpec *SDL_LoadWAV(const char *file, SDL_AudioSpec *spec,
                           Uint8 **audio_buf, Uint8 *audio_len)
```

`SDL_LoadWAV()`函数将音频文件`file`装载进`SDL_AudioSpec`对象`spec`。它还设置`audio_buf`指针指向音频数据被装载进的缓冲区，`audio_len`参数为音频数据的长度。

音频数据的长度和指针是很重要的元素，它们需要和`SDL_AudioSpec`信息一起被跟踪。处理这些信息的最常见方式是使用另一个结构：

```
struct {
    SDL_AudioSpec spec;
    Uint8 *sound;
    Uint32 soundlen;
    int soundpos;
} wave;
SDL_LoadWAV("test.wav", &wave.spec, &wave.sound, &wave.soundlen);
```

这个结构在一个位置提供了处理音频文件所需的所有信息。它使用`sound`指针指向音频数据缓冲区位置，而音频数据缓冲区的长度则存放在`soundlen`变量中。当我们开始播放文件时就将用到`soundpos`变量。一旦音频文件被装载进内存，你就已为播放它做好准备了。

但不幸的是，此时事情开始变得有点复杂了。你还记得`SDL_AudioSpec`对象指定了一个回调函数用于播放音频数据吗？正是这个回调函数使得事情开始变得复杂。

`SDL`将音频数据作为一个流来处理。音频流可能会在播放音频数据的过程中多次被停止和重启。每次当音频流被重启时，回调函数就会被调用。它必须知道它在音频数据文件中所处的位置，否则它只能每次都从文件的开始处开始播放。

为了解决这个问题，回调函数必须使用一个指针来记录音频文件在音频流中的什么位置停止，并使用该指针来获取继续播放的位置。这里就是`soundpos`变量发挥作用的时候了。下面这个简短的回调函数使用`SDL_MixAudio()`函数来播放一个音频文件流：

```
void SDLCALL playit(void *unused, Uint8 *stream, int len)
{
    Uint32 amount = wave.soundlen - wave.soundpos;
    if (amount > len)
        amount = len;
    SDL_MixAudio(stream, &wave.sound[wave.soundpos], amount, SDL_MIX_MAXVOLUME);
    wave.soundpos += amount;
}
```

当`SDL_AudioSpec`对象被创建时，它传递音频流及其长度给回调函数`playit`。局部变量`amount`根据`soundpos`变量的当前值来确定还有多少音频数据需要播放。`SDL_MixAudio()`函数用于将音频流发送给音频系统。该函数的格式为：

```
void SDL_MixAudio(Uint8 *dst, Uint8 *src, Uint32 length, int volume)
```

`SDL_MixAudio()`调用将当前音频文件的位置添加给音频系统以试图播放剩下的音频数据。其最后一个参数`volume`指定播放音频的相对音量。它的取值范围从0到`SDL_MIX_MAXVOLUME`。每次当音频文件的一部分被播放时，`soundpos`变量就被更新为音频文件中下一个还没有播放的位置。

一旦`SDL_MixAudio()`函数被调用，它就试图将音频文件发送给音频系统。`SDL_PauseAudio()`函数控制是否应该播放音频。使用参数1将暂停播放，使用参数0将开始播放。在默认情况下，它的参数将被设置为1以暂停播放直至下一个`SDL_PauseAudio()`函数调用。

代码清单13-6中的audiotest.c程序演示了如何使用SDL库播放一个.wav文件。

代码清单13-6 使用SDL播放音频文件

```
/*
 *
 * Professional Linux Programming - SDL audio test
 *
 */
#include <stdio.h>
#include <SDL/SDL.h>
#include <SDL/SDL_audio.h>

struct {
    SDL_AudioSpec spec;
    Uint8 *sound;
    Uint32 soundlen;
    int soundpos;
} wave;

void SDLCALL playit(void *unused, Uint8 *stream, int len)
{
    Uint32 amount = wave.soundlen - wave.soundpos;
    if (amount > len)
        amount = len;
    SDL_MixAudio(stream, &wave.sound[wave.soundpos], amount, SDL_MIX_MAXVOLUME);
    wave.soundpos += amount;
}

int main(int argc, char **argv)
{
    char name[32];

    if (SDL_Init(SDL_INIT_AUDIO) < 0)
    {
        perror("Initializing");
        return 1;
    }
    if (SDL_LoadWAV(argv[1], &wave.spec, &wave.sound, &wave.soundlen) == NULL)
    {
        printf("Problem: %s\n", SDL_GetError());
        return 1;
    }
    wave.spec.callback = playit;
    if (SDL_OpenAudio(&wave.spec, NULL) < 0)
    {
        printf("Problem: %s\n", SDL_GetError());
        SDL_FreeWAV(wave.sound);
        return 1;
    }
    SDL_PauseAudio(0);
    printf("Using audio driver: %s\n", SDL_AudioDriverName(name, 32));
    SDL_Delay(1000);
    SDL_CloseAudio();
```

```

    SDL_FreeWAV(wave.sound);
    SDL_Quit();
    return 0;
}

```

`audiotest.c`程序使用命令行参数传递要装载的音频文件的文件名。`SDL_OpenAudio()`函数使用已装载.wav文件所需的`SDL_AudioSpec`参数来打开一个音频设备。如果系统中没有一个音频设备可以播放该文件，`SDL_OpenAudio()`函数将返回-1。在打开音频设备之后，我们使用参数0调用`SDL_PauseAudio()`函数以开始音频文件的播放。

只有在程序是活跃状态时音频才会播放。正因为如此，我们添加了一个`SDL_Delay()`函数为音频复制文件的播放结束等待1秒。对于较长的音频复制文件，你将不得不改变延迟值。在音频文件播放结束之后，`SDL_CloseAudio()`函数用于释放音频设备，而`SDL_FreeWAV()`函数用于释放分配给音频文件的缓冲区空间。

3. 访问CD-ROM

在如今的多媒体世界中，许多Linux系统都包含了CD-ROM驱动器。SDL库允许你使用CD-ROM驱动器在应用程序中播放音乐CD。

`SDL_CDOpen()`函数用于初始化CD-ROM设备并检查放入驱动器中的CD。该函数只使用一个参数：要访问的CD-ROM驱动器的编号（CD-ROM驱动器从0开始编号，0为检测到的第一个驱动器）。

`SDL_CDOpen()`函数返回一个`SDL_CD`对象。这个对象是一个结构，它包含与CD相关的信息：

```

typedef struct {
    int id;
    CDstatus status;
    int numtracks;
    int cur_track;
    int cur_frame;
    SDL_CDTTrack track[SDL_MAX_TRACKS+1];
} SDL_CD;

```

`CDstatus`成员返回CD-ROM驱动器的当前状态：

- `CD_TRAYEMPTY`;
- `CD_STOPPED`;
- `CD_PLAYING`;
- `CD_PAUSED`;
- `CD_ERROR`。

`numtracks`和`cur_track`成员的含义比较明显，它们分别表示CD-ROM驱动器中载入CD的音轨数和当前选择的音轨编号。`track`成员包含CD中每个音轨的描述。`cur_frame`成员的含义并不那么明显，它包含当前音轨的帧位置。帧是CD音轨的基本计算单位，它等于音频中的1/75秒。可以使用`FRAMES_TO_MSF()`函数将帧值转换为分钟、秒和帧。

一旦你知道CD已装载进CD-ROM驱动器，只需使用`SDL_CDPlayTracks()`函数就可以播放音轨了：

```

int SDL_CDPlayTracks(SDL_CD *cdrom, int start_track, int start_frame,
                     int ntracks, int nframes)

```

其中`cdrom`是由`SDL_CDOpen()`函数打开的一个`SDL_CDROM`对象，`start_track`是开始播放的音轨，`start_frame`是开始播放的帧偏移，`ntracks`是要播放的音轨数，而`nframes`是从上一次停止播放的音轨开

始的帧数。

在默认CD-ROM驱动器中打开一个CD并播放第一个音轨的代码如下所示：

```
SDL_CD *cdrom;
cdrom = SDL_CDOpen(0);
SDL_CDPlaytrack(cdrom, 0, 0, 1, 0);
```

这就是你所需要的一切！从CD中播放音乐就是这么简单。SDL还集成了其他一些函数用于帮助搜集CD相关的信息。

代码清单13-7中的cdtest.c程序演示了如何使用SDL中的一些CD函数来有选择性地播放CD中的音乐。

代码清单13-7 使用SDL播放音乐CD

```
/*
 *
 * Professional Linux Programming - SDL CD test
 *
 */
#include <stdio.h>
#include <SDL/SDL.h>

int main()
{
    SDL_CD *cdrom;
    int track, i, m, s, f;

    if (SDL_Init(SDL_INIT_CDROM) < 0)
    {
        perror("cdrom");
        return 1;
    }

    cdrom = SDL_CDOpen(0);
    if (cdrom == NULL)
    {
        perror("open");
        return 1;
    }

    if (CD_INDRIVE(SDL_CDStatus(cdrom)))
    {
        printf("CD tracks: %d\n", cdrom->numtracks);
        for( i = 0; i < cdrom->numtracks; i++)
        {
            FRAMES_TO_MSF(cdrom->track[i].length, &m, &s, &f);
            if (f > 0)
                s++;
            printf("\tTrack %d: %d.%2.2d\n", cdrom->track[i].id, m, s);
        }
    } else
    {
        printf("Sorry, no CD detected in default drive\n");
    }
}
```

```

        return 1;
    }

    while(1)
    {
        printf("Please select track to play: ");
        scanf("%d", &track);
        if(track == -1)
            break;
        printf("Playing track %d\n", track);
        track--;

        if (SDL_CDPlayTracks(cdrom, track, 0, 1, 0) < 0)
        {
            printf("Problem: %s\n", SDL_GetError());
            return 1;
        }
        SDL_Delay((m * 60 + s) * 1000);
    }

    SDL_CDClose(cdrom);
    SDL_Quit();
    return 0;
}

```

cdtest.c程序的开始阶段如我们所预期的那样，它使用SDL_CDOpen()函数打开默认的CD-ROM驱动器。CD_INDRIVE()函数使用SDL_CDStatus()函数的结果来确定是否有一个CD可用。这提供了一个简单的接口来解释SDL_CDStatus()的结果。

如果CD存在，就启动一个循环以遍历CD上的每一首音轨，使用SDL_CD对象中的信息来获取每一首音轨的ID和帧数。FRAME_TO_MSF()函数用于将帧数转换为标准的分钟：秒格式。

在显示了音轨信息之后，程序要求用户输入想播放的音轨。输入-1将终止程序。因为SDL将音轨从0开始编号，所以在处理音轨编号时需要小心。SDL_CDPlayTracks()函数用于播放用户选择的音轨。

SDL_CDPlayTracks()函数是非阻塞的。也就是说，当音轨播放时它不会停止程序的执行。程序在启动CD播放后，将继续它的快乐旅程。为了暂停程序以等待CD音轨播放结束，cdtest.c程序使用SDL_Delay()函数来等待用户选择的CD音轨播放完毕。

编译程序，将音乐CD放入CD-ROM托盘，并测试它：

```

$ ./cdtest
CD tracks: 11
    Track 1: 3:56
    Track 2: 3:33
    Track 3: 4:28
    Track 4: 3:12
    Track 5: 4:23
    Track 6: 5:16
    Track 7: 3:30
    Track 8: 4:07
    Track 9: 4:08
    Track 10: 3:32
    Track 11: 4:08
Please select track to play:

```

程序成功地读取了音乐CD的信息，并将播放我所选择的音轨。

13.4 本章总结

在今天的多媒体世界中，能够创建包含图形和音频的程序是非常重要的。有几种方法可以在你的程序中集成图形和音频。OpenGL库用于为应用程序提供高级的2-D和3-D图形编程功能。OpenGL应用工具包（GLUT）可以和OpenGL一起使用来提供一个基于窗口的图形编程环境。许多3-D显卡提供了一个OpenGL库接口以使得高级3-D对象可以使用显卡上的板载硬件来绘制。对于普通显卡来说，你可以使用软件模拟来产生3-D对象。Mesa软件库是一个开放源码项目，它通过使用软件模拟为标准显卡提供OpenGL兼容的能力。

另一个流行的图形和音频库是简单直接媒介层（SDL）库。SDL库是一个跨平台库，它允许你使用可以运行在任何操作系统平台上的高级图形和音频功能来创建程序。SDL库还包括用于处理播放音乐CD以及标准.wav格式文件的函数。

第 14 章

LAMP

在当今世界里，几乎一切都围绕着因特网，因此使你的Linux应用程序可通过网络访问几乎是必需的。所幸的是，在开放源码社区中，有大量的软件可以帮助你达到这一目的。Linux-Apache-MySQL-PHP（LAMP）软件组合提供了一个完整的Web应用程序环境所需的所有元素。虽然LAMP本身并不是一个单独的软件包，但将这些开放源码软件包组合在一起非常容易，它们将带你进入Web的世界。

本章旨在为程序员提供一个如何使用开放源码LAMP平台开发Web应用程序的快速入门指南。本章涵盖了LAMP环境的基础，展示了如何在Linux服务器上集成Apache Web服务器、MySQL数据库服务器和PHP编程语言。在这之后，本章列出了一个完整的Web应用程序的例子，它表明使用LAMP环境实现Web应用程序是多么的容易。

14.1 什么是 LAMP

许多人都不知道，其实一些大型的知名网站都是基于开放源码软件建立的。如Google、BBC、Yahoo!、Slashdot和Wikipedia这样的网站都是建立在任何人都可以免费获取的组件之上并针对它们各自的需求对组件进行了修改。据Netcraft调查(http://news.netcraft.com/archives/web_server_survey.html)，全世界约有70%的网站是建立在LAMP平台上的。

LAMP是建立在开放源码技术（有时也包括一些专有技术）之上的Web应用程序的一个全方位的标签。传统上，一个LAMP应用程序被视为由下列开放源码组件构成：

- Linux
- Apache
- MySQL
- PHP（或Perl）

然而，上面所有的开放源码项目都是由不同的开发小组开发的，每一个软件都有其适用领域。鉴于此，这些组件之间并没有很强的依赖关系，将其中一个或多个组件用其他软件来替换也是很常见的，例如将MySQL替换为PostgreSQL、PHP替换为Perl、Apache替换为IIS、Linux替换为Microsoft Windows、Apache和PHP替换为Java/Tomcat，等等。

由于这种灵活性，对LAMP提供一个严格的定义不容易，下面这个定义可能相对贴切一些：

一个开放的Web应用程序平台，对所用的组件具备可选性和灵活性。

具备其他Web应用程序开发经验的读者可以将LAMP与其他平台或语言（如Microsoft的.NET、

ColdFusion或WebLogic) 进行比较, 这些商业产品针对从Web服务器到编程语言和数据访问层的所有组件提供了一个完整的实现栈。但是, 这些平台的封闭性限制了开发人员对应用程序中可使用组件的选择。不过, 也有人认为这提供了更好的组件整合并降低了复杂性, 它可能会提高开发的效率。

因为本书是关于Linux编程的, 所以其他操作系统中的方案将不在这里进行讨论——但值得指出的是, 这里介绍的许多组件至少都有针对Microsoft Windows和其他UNIX变体 (Solaris、FreeBSD、NetBSD、OS X等) 的版本。

14.1.1 Apache

Apache在1995年2月作为NSCA httpd Web服务器的一个分支诞生。它的名字反映了其早期的开发特点——通过邮件列表交流补丁集, 因此被称为“一个尽是补丁的”(A Patchy) Web服务器。Netcraft的调查显示, 自1996年以来, Apache就在Web服务器市场占据了垄断地位。它的流行是因为其开放源码的特性、安全性、可扩展性和灵活性——很难找到一个没有使用Apache的LAMP应用程序部署。有一些可供选择的应用程序服务器 (如Tomcat[Java]或Zope[Python]) 也可用来作为Web服务器——虽然它们也倾向于使用Apache作为其前端以解决安全性和配置问题。目前Apache有两个主要的版本——1.3 (V1分支中的最终版本) 版和版本2。后者提供了更好的性能, 所以在可能的情况下应该使用后者, 为了利用一些第三方的模块 (如mod_jk) 或老版本的PHP, 你可能需要使用较早的版本。

14.1.2 MySQL

MySQL是LAMP应用程序事实上的标准数据库选项。其中部分原因是由于PHP的早期版本对MySQL的出色支持, 但也包括一些其他因素, 如针对简单应用程序的高速MyISAM表格类型、良好的图形化开发工具、Windows支持和其商业支持公司MySQLAB的良好营销。MySQL数据库有许多可选的替代产品, 其中最常见的是PostgreSQL。有趣的是, 数据库的选择总是会挑起开发者和管理者之间的许多充满激情的争论。MySQL具有双许可证——GPL或商业许可证——这取决于应用程序, 它允许MySQL作为闭源应用程序的一部分被再次分发而不用继承GPL的限制。

14.1.3 PHP

PHP是Web应用程序开发中使用最广泛的语言——据Netcraft在2006年2月的调查, 接近1/3的网站报告说它们运行在PHP之上。PHP是一个灵活的面向Web的脚本语言, 它针对各种数据库存储引擎和各种工具 (如zip、PDF、Flash) 以及图形绘制等提供了广泛的支持。

由于当时缺乏建立简单网站的工具, Rasmus Lerdorf于1995年开始了PHP的开发。它一开始只是一些Perl脚本的集合, 但随着这些工具被越来越多的人使用, 更多的开发者开始加入它的开发队伍, 它被使用C语言重新实现并加入了一些额外的功能。到1997年, 这个项目最初的名字为“个人主页工具”(Personal Home Page Tools) ——被改为递归形式的缩写“PHP超文本预处理器”(PHP Hypertext Preprocessor), 并被完全重写, 从而形成PHP 3。2000年PHP 4发布——它支持了有限的面向对象功能, 同时商业公司Zend成立——它是PHP背后主要的推动力量之一。2005年PHP 5的发布被许多人看作是一个巨大的飞跃——它具备完整的面向对象功能 (包括异常)。

PHP有很多替代产品 (巧合的是, 其中一些也以字母P开头): Python, 一个面向对象的脚本语言; Perl, 它有着悠久的历史 (常被称为“因特网的传送带”); Java (与J2EE); 相对较新的语言如Ruby。每一种语言都有其自己的框架选择, 它可以使应用程序的创建变得更加容易 (如J2EE、Tapestry、

Hibernate、Ruby on Rails、PHPNuke、Drupal、Zope和Plone)。

14.1.4 反叛平台

一些业内专家将LAMP归为“反叛平台”(而非.NET或J2EE这样的“行业标准”平台)。这一定义来自于对LAMP(特别是PHP)的洞察，它颠覆了这个行业在历史上(尽管历史不长)由单一厂商主导平台(如ColdFusion或.NET)支配的局面。LAMP使得建立应用程序环境的财务费用显著降低，从而也降低了进入这一行业的门槛，带动了许多较小的和单人公司的增长——它正在改变着Web开发行业的面貌。

在这里，术语“行业标准”有点误导读者，因为LAMP所占据的市场份额，它已成为事实上的行业标准。然而，由于缺乏来自大型IT供应商(如IBM、微软、Sun)的重要支持，导致它不具备足够的资源(或中心组织)来与大型公司玩一场平等的营销游戏，从而使得它更容易在大型的项目中失败(这往往是因为这些项目背后的组织决策架构认为品牌决定一切)。但从另一方面来看，缺乏一个单独的大型企业的支持也有其优势——它导致只有较少的手续、透明和开放的开发(例如，公开的bug跟踪和讨论)。

14.1.5 评价LAMP平台

在决定是否使用基于LAMP的应用程序时有各种问题需要考虑，但其中许多问题都不是技术上的，而是取决于你所工作或参与的组织的决策文化。

- 费用与许可证；
- 认证；
- 支持和服务；
- 厂商/开发者和社区的关系；
- 集成和支持其他技术。

1. 费用与许可证

关于开放源码项目的一个常见误解是认为它们必须是免费的——从费用上来说。其实许多许可证都允许为产品收费，虽然LAMP栈中的绝大多数组件实际上都可以在大多数环境中自由下载与使用，但也有例外，如MySQL(针对商业应用程序)或第三方的模块与插件。对于建立一个LAMP应用程序所需的费用可做的一般假设是：大部分费用来自于开发者和管理者，而不是来自于软件使用许可。如果使用的是其他平台，许可证的费用可能成为整个项目费用的一个重要部分——尤其当为某些许可证付费时需要根据每个软件销售“单位”进行付费时更是如此。

LAMP平台包含组件的开放许可证，以及许多使用它建立的应用程序的开放许可证——为软件复用提供了非常好的机会，而这将进一步降低开发成本。如果你有这个技术，还可以自由调整或扩展组件以适合你的需要。

2. 认证

认证在过去一直是软件产业的一个重要组成部分，所有大公司都为它们自己的产品用户提供认证计划。认证在雇主和软件委托者之间一直都很流行——它提供了评估潜在雇员或开发公司技能的标准，这将节省雇主的宝贵时间。因此，开发人员和管理人员总是非常热衷于获取这样的认证证书。

对LAMP平台组件的认证是一个相对较新的领域。由于缺乏企业对这些产品的支持，这意味着没有公认的机构来验证这类认证。然而，随着支持各种LAMP组件的商业公司的不断发展，业界认可的

认证如Zend的PHP认证、来自MySQLAB和LPI的MySQL认证、RedHat和Suse的Linux管理认证已出现并迅速流行开来。这类认证理论上有助于雇主雇用那些满足特定要求的雇员。

3. 支持和服务

多年来，获得软件（包括编程语言和开发平台）的“公认”途径是从供应商那里购买软件包，其中还包括一定数量的支持。人们期望（在一些部门的人现在仍然期望如此）软件和服务都来自同一地方。

但就开放源码软件而言，这将导致一个问题。因为人们并没有从一个供应商那里购买软件，所以有意购买对该产品支持的公司往往不知道应该到哪里去购买。这一问题有几种解决方案，第一种方案是遵循传统路线。有时候，应用程序由一个公司开发并以开放源码的形式发布以利用开发者社区（或纯粹是为了支持该产品而组成公司）。在这种情况下，你可以找到一个单一的实体来提供支持。此外，为了对开放源码应用程序提供支持，有一个产业在不断增长——这往往采取的是在某一特定场合提供产品支持、知识和专门技术的个人承包商或小型公司的形式。第三，产品的大型用户可能会发现雇用或培训内部支持人员会更加节约成本。

4. 厂商/开发者和社区的关系

就其本质来说，开放源码往往会导致一个更加透明和活跃的开发过程，在这个过程中，错误能够得到迅速解决，新的功能也能更快速地到达用户那里。许多开放源码项目提供邮件列表以允许有兴趣的人士参与对软件功能开发的讨论，而这可能会影响项目的未来发展方向。可以立刻获得源代码并具备自己编译应用程序的能力将导致一个永无休止的“beta”发布——虽然应当注意不要在一个现实的或陌生的环境中使用未经测试的软件。

一些开放源码项目允许有兴趣的团体资助某一特定功能的开发——这可能是通过代码或开发时间的捐赠或费用的捐赠——也被称为奖金。奖金一般不集中管理，而是作为一个开放的命题提供，开发者可以选择进行开发。但是，奖金的提供并不能保证一个软件功能将被实现。

在一个商业环境中，开发者和社区之间的关系趋向于更正式和缺乏灵活性。虽然许可证和软件支持协议确保软件供应商应对客户担负一定程度的责任，但几乎没有什么协议会让客户有机会影响产品的发展方向。

对于非常重要的新功能，开放源码产品的用户通常可以要求雇用关键开发人员来实现他们所需的功能。但对于一个闭源系统的用户来说，这就比较困难了。如果你对某个特定产品抱有疑问，而它又是开放源码的，那么你并不需要通过多么复杂的途径就可以使用各种方式（如邮件列表、即时聊天（IRC）和在线论坛）与开发人员（或社区中的其他人）直接联系。

5. 集成和支持其他技术

一般情况下，开放源码产品之间的集成比较容易，而开放源码产品和闭源组件之间的集成问题可能就比较多，但它总比集成两个闭源应用程序要容易的多。比如PHP已支持了Oracle、Sybase、Microsoft SQL服务器和Java，但你能想象Microsoft会在ASP中支持PHP吗？

现在你已对LAMP有一个总体了解，是时候介绍LAMP的每个组件并了解如何安装和使用它们了。下面几节将分别介绍每个LAMP组件并说明如何在你的Linux系统中安装和运行它们。

14.2 Apache

本节将举例说明与LAMP应用程序相关的一些有用的、特定的Apache元素。我们将介绍如何为

Apache建立一个虚拟主机、整合Apache与PHP、实现HTTP认证。最后，我们将展示如何使用SSL加密。

几乎所有的Linux发行版都会预装Apache。下面的例子使用的是Apache 2.x版本，它可以在任何派生自Debian的Linux系统中找到。

如果你没有在你的Debian系统中安装Apache，只需运行下面的安装命令就足够了：

```
apt-get install apache2
```

对于其他Linux发行版来说，请参考具体的软件安装说明以了解如何安装Apache软件包。对于派生自Red Hat的系统，这通常需要安装一个或两个.rpm文件。

14.2.1 虚拟主机

大多数Apache的安装都会创建一个通用的配置文件以启动Apache服务器。但是，如果你想通过Apache做一些实际的工作，你必须针对你的环境来定制配置文件。在现实世界的Web服务器中，一个常用的功能就是虚拟主机。

虚拟主机是一种让多个Web网站监听同一个IP地址的方法。这非常有用，因为它允许在有限数目的IPv4地址上实现无限多的Web网站。

请回忆本书前面对HTTP的介绍，在使用TCP协议与服务器的80端口建立连接时，客户端发送一个“Host:”消息给服务器，服务器使用它来将用户访问的网站与服务器配置的虚拟主机进行匹配。

要定义一个虚拟主机，我们需要以root用户身份在/etc/apache2/sites-available目录下创建一个类似下面这样的文件：

```
<VirtualHost *>
    ServerName test.example.com
    ServerAlias test www.test.example.com
    ErrorLog /var/log/apache2/test-error_log
    CustomLog /var/log/apache2/test-access_log common
    DocumentRoot /www/test.example.com
    DirectoryIndex index.php
    <Directory "/www/test.example.com/">
        Options Indexes MultiViews FollowSymLinks
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

ServerName指令指定这个网站的名字，而ServerAlias允许同一个站点匹配多个其他的URL（例如，在一个内部网络中，你可能会直接输入网站的主机名而跳过其域名）。ErrorLog和CustomLog指令指定Apache在哪个文件中记录与这个站点相关的信息（例如客户浏览网站内容的请求等）。DocumentRoot指令指定在文件系统的哪个位置放置这个站点的文件。DirectoryIndex指定当用户只在URL中提供一个目录名时，站点应该使用哪个页面。

<Directory>允许任何人访问指定目录中的内容（因此指定Allow from all）。

现在，你需要使用如下命令告诉Apache加载你的配置：

```
a2ensite the_file_name
/etc/init.d/apache2 reload
```

只要你的网站在DNS服务器或你的本地/etc/hosts文件中有合适的条目，你现在就可以通过Web浏览器访问该网站了。

14.2.2 安装和配置 PHP 5

虽然本节的标题看起来好像我们有些超前了，但安装PHP 5软件实际上是Apache安装过程的一个组成部分。Apache发行版将PHP 5软件作为一个模块并包含在其中。Apache中的模块是一个自包含的功能体，它可以被添加到Apache服务器中。

与基本的Apache服务器安装一样，安装和配置PHP 5 Apache模块的方法将根据你所使用的Linux发行版的不同而不同。所有主流的Linux发行版都包含预包装的PHP软件包，可以使用正常方法来安装它。请查找标记为php-mod或类似形式的安装文件。

在将PHP 5模块安装到Apache服务器中之后，你需要检查它是否能正常工作。为了测试你的设置，在Web网站的文档根目录下创建一个包含如下内容的HTML文件：

```
<?php  
phpinfo();  
?>
```

通过Web浏览器访问这个文件应该显示一个包含PHP相关信息的页面。如果确实如此，你已成功地安装了PHP模块。如果没有，你可能需要查询你的Linux发行版的特定软件安装指导以确定必须装载哪些模块。

1. PHP配置

通常情况下，你并不需要改变默认的PHP配置，但为了供你参考，下面的信息可能对你有用。

你可以通过修改php.ini文件来调整PHP中的很多设置。但PHP的CLI版本（命令行方式的PHP）和Apache模块通常使用不同的配置文件。

2. 安装PHP模块

PHP本身包含了几个功能模块，它们可以在Linux系统上单独安装。PHP一般以一组软件包的形式针对Linux发布，这允许你选择在服务器上安装哪些软件包。默认安装的软件包通常不带数据库相关的扩展，所以它无法立刻和PostgreSQL或MySQL数据库通信。你必须安装PHP MySQL模块以使得PHP可以与MySQL数据库（LAMP环境中另一个组件）通信。

对于派生自Debian的系统来说，你可以安装php-mysql模块以包含PHP中与MySQL数据库通信的连接组件。安装了附加的软件包之后，可以使用phpinfo()页面再次测试你的设置。这次，它应该显示PHP可以与MySQL数据库连接和通信了。

说明 一般不要在一个现实的网站上保留phpinfo()风格的页面并允许别人访问，因为它潜在地允许恶意用户访问超过你的站点配置中实际允许的信息。

14.2.3 Apache Basic 认证

如果你要限制你的网站（或仅仅是网站的一部分内容）只给特定的客户访问，你应该实现basic认证。Apache可以基于每个目录（或有时候，基于每个文件）提供简单的资源认证。这既可以在站点的Apache全局配置文件（在本例的情况下，是在我们前面定义的VirtualHost容器中）中指定，也可以在需要保护的目录下的.htaccess文件中定义。

Apache提供的认证机制并不安全（例如，密码以明文方式传输）。如果安全性是你要考虑的一个问题，请阅读使用SSL认证的相关内容。

需要用户登录才能访问某个特定资源（在本例的情况下，是目录/admin）的Apache配置可能像下面这样定义，它应该被放入前面显示的VirtualHost容器中。

```
<VirtualHost *>
    ...
    <Location "/admin">
        AuthType Basic
        AuthName "Administration Site"
        AuthUserFile /etc/apache2/htpasswd
        AuthGroupFile /etc/apache2/htgroup
        Require group dvd-admin
    </Location>
    ...
</VirtualHost>
```

你需要以如下方式创建文件/etc/apache2/htpasswd和/etc/apache2/htgroup（它们的名字是随意指定的——定义在上面的配置文件中——但确实是一个有意义的名字），你创建了两个用户和一个指向它们的组文件：

```
$> htpasswd2 -c -m /etc/apache2/htpasswd my_user_name
$> htpasswd2 -m /etc/apache2/htpasswd an_other_user
$> echo "dvd-admin: my_user_name an_other_user" > /etc/apache2/htgroup
```

要注意的是，-c标记不应当被用在一个已有的文件上，否则，你将丢失该文件的内容！

现在，如果你试图访问网站的/admin部分，你将看到一个弹出的认证窗口。

14.2.4 Apache与SSL

除了basic认证以外，你可能还想实现Web站点的加密。对于处理个人信息的站点来说，这应该是一个必要条件。SSL用于加密HTTP协议之下的传输层。因此，不管从开发者还是从用户的角度来看，除了使用SSL可以确保内容传输的安全性以外，使用或没有使用HTTP+SSL（HTTPS）的站点是没有区别的。设置SSL的方法取决于你所使用的Linux发行版，在基于Debian的系统中，你可以使用如下命令：

```
$> apache2-ssl-certificate
```

如果你已有了一个SSL证书，你需要为这个脚本提供-force选项。如果你希望定义这个证书的有效期，需要使用-days参数。

这个命令将引导你回答一系列的问题，并要求你提供需要使用SSL证书保护的站点信息。在它成功运行之后，你需要重新配置Web服务器以包含类似下面这样的信息：

```
NameVirtualHost *:443
<VirtualHost *:443>
    SSLEngine On
    SSLCertificateFile /etc/apache2/ssl/apache.pem
    ServerName mysecureserver.example.com
    ServerAlias mysecureserver
    .....other directives as before.....
</VirtualHost>
```

注意VirtualHost定义的修改——这是为了将它绑定到正确的SSL端口（443）。

14.2.5 SSL与HTTP认证的整合

你当然可以让每个人都改用HTTPS风格的链接，但这样做对你的用户来说可能有些不方便，因为他们必须更新他们的书签，所以这种改变最好可以自动发生。另外，如果你没有注册过的证书，你将不得不要求你的用户信任你提供的SSL证书。值得庆幸的是，关于书签的问题，我们可以使用Apache的mod_rewrite模块重定向用户对http://myserver/admin的访问到https://myserver/admin，并使用我们前面讨论过的htpasswd风格的认证方法提示用户登录。

在先前的非SSL站点中，将如下内容添加到VirtualHost容器中：

```
RewriteEngine on
RewriteCond %{REQUEST_URI} ^/admin(.*)
RewriteRule ^/(.*) https://securesite/$1
```

现在，如果有用户访问URL <http://mysite/admin/anything>，将发生如下情况：

1) URL将匹配Apache中的RewriteCond的规则^/admin。

2) Apache将用户重定向到<https://securesite/admin/anything>。

3) 用户将被提示接受https://securesite的SSL证书（如果它没有得到诸如Thwate这样的授权机构的信任）。

4) SSL会话将启动，浏览器将要求用户认证。

5) 用户输入认证的细节，传输的数据都经过SSL加密。

6) 服务器返回用户请求的内容，该数据同样经过SSL加密。

现在，你有了一个正常运行的Apache Web服务器，它支持PHP 5程序，而该程序又可以访问MySQL数据库。下一步，当然是要安装MySQL数据库以保存你的Web数据了。

14.3 MySQL

在LAMP模型中，你需要的下一个组件是MySQL数据库。本节讨论如何安装MySQL数据库并针对你的环境对它进行配置。一旦MySQL服务器可以正常运行了，你就需要为数据库配置一些表。本节通过向你展示一些基本的数据库编程命令以使得你可以建立自己的数据库。

14.3.1 安装 MySQL

与安装Apache和PHP 5类似，对大多数Linux发行版来说，MySQL的安装也不过只是找到正确的安装文件，然后使用你的安装软件来安装它而已。MySQL在Linux世界中非常受欢迎，因此，现在一些Linux发行版默认就会安装它。你可以通过检查你的Linux发行版的已安装软件列表来查看系统是否已安装了MySQL（或者你也可以在命令行中输入mysql命令来查看是否有输出）。

如果你的系统没有安装MySQL，对于派生自Debian的系统来说，你可以使用apt-get命令来安装基本软件包：

```
apt-get install mysql
```

有些Linux发行版将MySQL服务器和客户程序放入不同的安装软件包中。如果你的发行版是这么做的，那么你还需要安装客户端软件包。它提供了通过命令提示符与MySQL服务器交互的应用程序。

这使得创建和管理数据库更加容易。

在大多数Linux发行版中，安装包将安装MySQL数据库软件，但它只提供一个基本的配置或根本不进行配置。所以下一步的工作就是配置和启动你的MySQL数据库。

14.3.2 配置和启动数据库

在启动MySQL服务器之前，必须创建系统数据库，MySQL将使用它们来跟踪表、用户和其他数据库信息。要完成这个工作，需要以root用户身份运行由MySQL提供的脚本mysql_install_db。

当运行这个脚本时，应该可以看到显示脚本运行进度的信息。这个脚本将建立如下几个表：

```
Creating db table
Creating host table
Creating user table
Creating func table
Creating tables_priv table
Creating columns_priv table
```

当这个脚本执行完毕后，你就可以启动MySQL服务器了。具体的启动方法还是与发行版有关。有些发行版将MySQL数据库的启动脚本放到自动启动的区域中，当Linux系统启动时，MySQL服务器将会被自动启动。但是，如果你的发行版没有这么做，你就必须手工启动它了：

```
/usr/local/bin/safe_mysqld --user=mysql &
```

safe_mysqld脚本使用提供的用户账号启动MySQL数据库（如以root用户身份启动服务器被认为有安全风险）。大多数发行版都提供mysql账号用于这一目的。命令结尾的&符号强制服务器在后台运行。

14.3.3 修改默认密码

MySQL服务器维护它自己的用户列表，这与Linux系统的用户是分开的。服务器的管理员账号是root。但在默认情况下，root用户账号没有密码。对于可以通过因特网访问的数据库来说，这很不安全。

mysqladmin程序允许你在服务器上执行管理功能，如停止服务器、显示正在运行服务器的状态，当然还包括修改用户的密码。修改root用户密码的命令格式如下：

```
/usr/local/bin/mysqladmin -u root -p password 'newpassword'
```

当然，要使用mysqladmin程序，你必须是root用户，所以这个命令要做的第一件事情就是获取root用户的密码。由于默认密码为空，所以你只需按下回车键即可。然后，root用户的密码就被修改为你在命令行中提供的密码。

14.3.4 MySQL 客户端接口

当MySQL服务器正常运行之后，你可以尝试使用一个交互提示符来登录服务器。mysql命令启动一个命令行程序以允许你连接到数据库并执行SQL命令。要使用root用户账号连接到默认的MySQL数据库，请输入：

```
mysql -u root -p
```

在输入密码之后，你应该可以看到MySQL提示符：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 250 to server version: 3.23.36

Type 'help;' or '\h' for help. Type '\c' to clear the buffer

mysql>
```

现在，你可以开始输入SQL命令建立自己的数据库了。

14.3.5 关系数据库

在如今的软件开发中，关系数据库（RDB）被广泛使用。传统的UNIX平面文件数据库已在前面的章节中讨论过。但这些传统数据格式将为应用程序建立良好数据模型等诸多的日常任务留给程序员来完成，而且当每次开发新的应用程序时，程序员都需要重新实现它们。来自现实世界中的数据往往都有一些物理限制，而且这些限制需要在应用程序的某处反映出来。许多程序员选择将它们放在应用程序的逻辑代码中，但它们的更恰当的位置应该是数据层本身（特别是在应用程序中，当你有一个以上的数据前端时更是如此）。关系数据库及其相关软件——关系数据库管理系统（RDBMS）可以在数据定义中反映这些限制，并在数据被输入或修改时进行相应的限制，这样可以确保开发人员不会突然忘记实现某个已在另一个用户接口中存在的限制——导致创建数据的不一致。

14.3.6 SQL

RDB中数据的定义、创建、访问和修改都使用结构化查询语言（SQL）。这是大多数开发者都很熟悉的一个RDB元素，因为即使对它只有一个基本的了解（通常都是对来自其他资源的代码进行剪切和粘贴）也足以让你的应用程序正常工作。我们之所以将本节放在对关系模型的解释之前（虽然这样做是次序颠倒了）是因为许多开发者已对SQL比较熟悉，即便他们可能并不了解关系模型，而且使用SQL有助于我们对数学模型的解释。如果你对SQL已经非常熟悉了，你可以跳过本节，否则，本节介绍的概念将有助于你理解下一节中对关系模型和SQL的比较。

本节并不打算成为关于SQL语言的全面而完整的参考资料，而只是作为一个简单的“入门”指南。我们解释了一些最基本的语句并提供信息以帮助那些之前没有SQL使用经验的读者。

特定类型的数据被存储在表中（例如一个人的所有数据都被收集到名为Person的表中，等等），数据的属性作为表的列（或字段）。记录（或行）是该类型的实例，所以“Jane Smith”及她的所有相关数据都被存储在Person表的一个记录中。

本节介绍的所有SQL语句都是标准语句。这意味着它们都符合大多数关系数据库管理系统实现的SQL 92标准。这些语句可以在广泛的RDBMS系统中运行，这将允许你创建厂商无关的应用程序。

本节示例中的一些数据类型是PostgreSQL专用的。如果你选择的RDBMS不接受这些数据类型，应该可以根据DBMS文档中的类型定义对表的定义进行简单修改。

下面列出了一些SQL语句的注意事项：

- SQL语句使用分号（;）结束，大多数关系数据库的编程接口会自动将分号添加到SQL语句后。
- 所有SQL语句都有基本语法，通过给它辅以可选的参数可以使语句的含义更加明确。
- 按照惯例，所有SQL关键字都以大写字母表示——这使得区分关键字与数据和函数变得相当简单。

1. 创建表

创建一个表（关系）的基本语法是：

```
CREATE TABLE tablename(fieldname type, fieldname type, ... fieldname type);
```

我们的例子：

```
CREATE TABLE dvd(title character varying(150), director character varying(150),
release_date int4);
```

将创建一个名为dvd的表，它有三个字段——两个有长度限制的文本字段title和director，一个针对发行年代的整数字段。

2. 插入记录

当你定义了一个表后，你会想插入一些数据到表中。有两个基本的语法格式用于完成这个工作：

```
INSERT INTO tablename VALUES (value, value, ... value)
INSERT INTO dvd VALUES ('How to Lose a Guy in 10 Days', '2003', 'Donald Petrie',
'12', 'Comedy');
```

这将依次提取每个值并试图将它们按照创建表时定义的字段顺序存储到表中。

但人们经常只需要插入一些特定字段或它们需要以不同的顺序插入。在这种情况下，我们可以使用下面的语法来定义适当的字段和相应的插入值的次序。

```
INSERT INTO tablename (fieldname, fieldname, ... fieldname) VALUES (value, value,
... value)
```

要在DVD表中插入值，根据前面的声明，在不知道乐队指挥（director）的情况下，可以使用如下的语句：

```
INSERT INTO dvd (title, release_year, rating, genre) VALUES ('How to Lose a Guy in
10 Days', '2002', '12', 'Comedy');
```

3. 查询记录

根据各种条件定位记录可能是大多数系统中最常见的数据库操作了。其语法很简单，但它有一些选项。最简单的语法版本是：

```
SELECT * FROM tablename WHERE field = value;
```

这将返回表中给定字段完全匹配指定值的所有记录。一个查询语句中执行搜索的两个主要组成部分是SELECT子句和WHERE子句。

- SELECT子句

SELECT部分涉及对需要在查询结果中显示的字段的定义，这缩小了查询的输出结果。多个字段通过以逗号分隔的列表来提供：

```
SELECT title FROM dvd;
SELECT title, director FROM dvd;
```

上面的例子将列出DVD表中所有记录的标题。

- WHERE子句

如果没有WHERE子句，查询语句返回的将是表中所有的记录。WHERE子句根据给定的条件限制了返回的记录数。该子句还用于其他SQL语句。

要找出所有在2000年以前发行的DVD，我们可以使用如下的语句：

```
SELECT * FROM dvd WHERE release_year < 2000;
```

为了进一步缩小返回的记录数，我们可以在WHERE子句中使用AND（或OR）来找出所有由Terry Gilliam当指挥的DVD：

```
SELECT * FROM dvd WHERE release_date < 2000 AND director = 'Terry Gilliam';
```

在WHERE子句中，=运算符匹配完全相等而且它是区分大小写的。此外，还有许多其他的比较运算符可以使用，例如：

- > (大于);
- < (小于);
- AND / OR / NOT;
- != (或<>);
- >= 或<= (大于等于， 小于等于);
- IS NULL / IS NOT NULL。

4. 编辑记录

当数据库表包含一些数据后，你很可能需要更新它。UPDATE语句用于完成这一工作：

```
UPDATE tablename SET fieldname = value, fieldname = value, ... fieldname = value  
WHERE fieldname = value;
```

这里的WHERE子句用于识别要更新的记录。如果WHERE子句匹配多个记录，它将更新所有匹配的记录，所以你需要确保指定的条件仅选择你想要更新的记录。在将WHERE子句用在UPDATE语句（或DELETE语句）中之前，先在SELECT语句中对它进行测试往往是一个好主意。

要更新DVD表中一个字段的值——例如，如果输入的是电影的发行日期而不是DVD版本的发行日期，可以使用如下语句进行更新：

```
UPDATE dvd SET release_year = 2003 WHERE title = 'How to Lose a Guy in 10 Days';
```

5. 删除记录

除了必须删除整个记录，删除记录和更新记录的语法很像，所以前者没有必要指定需要记录的哪些部分。它的语法是：

```
DELETE FROM tablename WHERE field = value;
```

例如：

```
DELETE FROM tablename WHERE title = 'How to Lose a Guy in 10 Days';
```

上面的语句将只删除How to Lose a Guy in 10 Days记录，因为没有其他记录匹配这个标题。

14.3.7 关系模型

许多临时的（和一些非临时的）关系数据库用户对用于定义和约束RDBMS行为的数学原理并不熟悉。要想真正理解如何构造和搜索数据，你最好能对RDB背后的理论有一个深刻的理解。但本节并不打算提供这种程度的知识，而只是提供一些名词和原理的概述。本节的内容是经过高度简化的，你需要在此基础上进行进一步的学习。

如果你有兴趣建立含义明确的、健壮的RDB应用程序，你应该在阅读本书的基础上继续阅读介绍数据库理论的优秀图书，如C.J.Date的*Database in Depth: Relational Theory for Practitioners*。

1. 关系、元组和属性

关系模型的基本构建单位是关系，它大致等同于SQL中的表的概念，它是RDB中的主要数据单元。关系代表了某一特定类型的数据集。在本章最后的PHP示例应用程序中实现了一个非常简单的DVD库，它里面的基本关系有dvd、library_user和genre。元组大致相当于行，所以包含Serenity, 2005, Joss Whedon, 15, Sci Fi值集的行就是一个元组。属性大致等同于列的概念，所以Director就是关系dvd的一个属性。

2. 关键字和关联

为了能对数据执行某些操作，如检索数据，每个元组需要有一种方法能被唯一标识。能够被用来唯一标识一个元组的属性或属性组被称为关键字。关键字可以是各种类型。候选键是可以唯一标识一个记录的任何关键字。但是，由于一个关系可能有多个可能的唯一属性组，所以人们从候选键中选定一个唯一标识记录的键作为主键——这更多的是一个RDBMS的实现问题，因为基于关系模型的操作可以在任何一个候选键上执行。在实践中，关键字用来准确地标识关系之间的联系，如果一个表中的主键出现在另一个表中——用于将两个表关联起来——而且该字段不是另一个表的候选键，则它是该表的外键。

关系、元组和属性本身并不能反映数据的真实结构。单独来看每个关系只能准确地反映数据的一个小子集，而数据项之间的关联以及这些关联的限制和约束才可以帮助你得到数据的更加准确的表示。在本章的示例应用程序中的关联并不是说明关联的最好例子，因为它在genre关系中表示dvd关系中的数据分类——这是强加给数据的关联。一个更好的例子可能是学校结构的表示，因为学生和班级有明确的关联。关联是根据关联两端参与者的数量定义的。它的值可以是0、1或许多，你可以对每个关联设置一个最小值和最大值。这方面的例子将在介绍数据模型时解释。

3. 关系运算符

当你有了一组数据后，你当然希望能够操纵它——要做到这一点，你就需要一些运算符。关系赋值运算符用于给一个关系赋值（对于SQL而言，就是INSERT、UPDATE或DELETE语句）。除此之外，还有关系运算符，分为限制（restrict）、投影（project）、积（product）、交（intersect）、并（union）、差（difference）、连接（join）和除（divide）。在本节介绍所有这些运算符会显得过于详细，但我们要指出的是，其中许多运算符其实都是不同类型的连接（例如，并是连接的一个特例）。理解这些概念以及它们是如何影响一个数据集的将有助于你为自己的任务选择合适的和最有效的SQL语句。

4. 数据完整性

在现实世界中的数据集，如放在家里架子上的DVD，或你的孩子去学校——都有这个世界（无论是政府、他们的父母或物理规则）制定的一些实际限制。你通常可以定义一些在你的系统允许范围之内的限制，它们永远不应被打破。你的限制可以应用在不同的层次——从一个属性的大小（英国国民保险号始终是9个字符）、一个元组的内容（来自dvd关系中的元组必须有一个非空的title字段）到关系之间的关联。

5. 数据模型和规范化

确定系统界限以及其限制所在，可能是设计任何系统时最至关重要的部分。确保数据存储系统准确地对要表示的数据进行建模将增加应用程序作为一个整体来满足用户实际需求的机会，但这是一件

非常困难的事情。当对系统进行建模时，应该确保你完全理解了数据及其关联，或至少从对数据有最佳理解的人（不幸的是，这样的一个人可能根本就不存在或你没有办法联系他）那里收集了信息并小心地解决任何疑问。

当对一个系统中的数据进行建模时，首先需要尽可能多地收集关于系统及其工作方式的信息。以下类型的信息有助于完成这一工作：

- 需求规格说明书；
- 现有记录（来自以前的电子系统或纸质记录）；
- 对用户如何执行他们任务的描述（由用户提出或来自访谈）；
- 观察用户的工作。

一旦有了一套文档，就可以开始审查它们并挑选出相关信息。通过这些信息，你可以找出与系统相关的规则——被称为完整性规则——它们可能是直接从文档中提取出来的句子，或者是通过各种信息来源得出的结论。这些规则可能和数据直接相关也可能无关，但将它们挑选出来将有助于你做出决定。在一个学校系统中，一些完整性规则可能如下所示（来自一个假想的针对于学校记录系统的需求规格说明书）：

...
...
每个教师只教一个“班级”。
每个学生同时只能属于一个班级，而且每个学生必须属于一个班级。
必须保留每个学生的地址和其家长的详细联系方式以备急用。
班级必须由属于同一年级的学生组成。
教师必须在每次注册时为每个学生记录一个值（出席或缺席）。
在可能的情况下，在注册记录中的缺席值应附带批准理由。
...
...

这些完整性规则一开始可以被用于确定实体（具体或抽象的概念，如人、图书、学校等）和属性（用于描述实体）。在这个例子中，可以确定几个可能的实体，包括“教师”、“学生”和“班级”。学生的可能属性包括“地址”和“家长详细联系方式”，当然，这很明显还需要进行进一步定义。这些实体和属性与数据库设计中的关系和属性的原型相当。

当有了一组构成系统的实体和属性之后，我们就可以通过检查数据及其关系来产生一个有效并且高效的设计。这是通过一个被称为规范化的过程完成的。规范化过程使用一个数据集——它最初大致被分为各种关系——然后对这些关系进行组织以使得应用于数据的关系模型是有效并且高效的——例如，通过删除重复的信息。规范化过程有不同的阶段或程度，每一个都比其上一个要更“严格”——常用于数据库设计的有：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）和鲍依斯-科得范式（BCNF）。第四和第五范式用于处理一些涉及数据的非常罕见的情况，它们并不被经常采用。

许多开发者会发现，如果他们已使用了一段时间关系数据库——即使他们不了解范式，他们实际上已使用了规范化过程中执行的一些步骤，因为对于数据库的设计和使用来说这些步骤是显而易见的。如果在这里介绍规范化的例子将使得我们陷入过多的细节，在这方面有许多优秀的参考图书可以指导你通过使用样本数据逐步地完成这一过程。通过对一些现实世界中的数据应用这些原则（可能来自一个真正的应用程序）将有助于你更快速和容易的理解这一过程。

14.4 PHP

PHP是一种专门用于开发动态Web应用程序的编程语言。与Perl和Python语言不同，PHP最初的设计只是创建用于帮助生成动态Web网页的一组工具，其后，它在牢记这一目的的前提下经过了不断的演变（经历了数次重写）。

14.4.1 PHP语言

LAMP环境的最后一个组成部分是PHP编程语言。PHP是一个简单的脚本语言，但它却允许你在一个HTML Web页面中执行复杂的功能。本节假定读者已有一定的PHP编程的经验。我们将重点介绍PHP 5，因为它具有增强的面向对象实现（接口和抽象类等）。

本节将涵盖一些PHP的基本原则并重点说明PHP与其他主流语言的主要不同之处。然后我们将介绍PHP的一些基本优化技巧。在掌握了这些内容之后，本节将向你显示如何创建一个简单的Web表单并使用适当的脚本来处理结果，然后以它为基础建立一个更实际的带有多个页面和数据库后端的例子。在本节的最后，你将学习Smarty模板语言的使用、用于事件记录的PEAR Log模块、用于代码剖析和调试的xdebug模块、一些面向对象例子、错误和异常处理以及PEAR的数据库模块。

本章不会花时间解释PHP语言的结构等基本的概念，其部分原因是因为我们假设读者已有了一些编程经验。但我们会试图指出这个语言的一些独特之处，正是这些特点使得它能脱颖而出并吸引大众的眼光。

如果你想更详细地了解某个特定的PHP函数，在线的PHP API是非常好的资源，它包含丰富的由用户贡献的说明和例子。你可以访问www.php.net并在其后跟上特定的函数名，例如[www.php.net /microtime](http://www.php.net/microtime)。

PHP语言的语法和结构类似于许多其他语言（Perl、C、Java），任何人只要熟悉这些语言中的一种就将发现PHP很容易使用。

1. 语言基础

按惯例，PHP代码块保存在以.php为后缀的文件中。PHP代码可能会和HTML代码混合在一起，它以<?php开头，并以?>结束。在某些环境中，也可能使用<% ... %>或<? ... ?>。

PHP语言支持所有常见的条件语句（for、while、do/while、if）。变量总是以\$开头（和Perl语言一样）。PHP不需要定义变量的类型，类型转换将根据上下文自动发生。

例如，一个文件包含如下内容：

```
<?php
$variable1 = "Bob";
$variable2 = Array("This", "Is", "an", "Array");
echo "Variable 1 is : $variable1\n";
echo "The second item in the array is : " . $variable2[1] . "\n";
?>
```

执行该文件将产生如下输出：

```
Variable 1 is : Bob
The second item in the array is : Is
```

当编写PHP代码时，程序员可以选择在一个双引号标记的字符串中包含变量（如上面的Variable 1 is : Bob中的变量）或使用.符号将变量附加到一个字符串的结尾。每种方法都是正确的，虽然在

大多数情况下，将变量包含进字符串的方法好像看起来更整洁。但它确实对性能有影响，我们将在后面的关于优化章节中对它进行讨论。

下面的例子显示了PHP和普通HTML代码是如何混合在一个页面中的。当读取Web页面时，PHP解析器理解PHP的开始标记（<?、<?php或<%，这取决于具体的配置）并执行适当的代码：

```
<html>
<head><title>My first PHP page</title></head>
<body>
<?php
$count = 0;
while($count<50) {
    echo "<br/>Hello World\n";
    $count++;
}
?>
</body>
</html>
```

这个例子将在一个Web页面上输出Hello World 50次。

PHP还可以作为一个独立脚本编写并使用/usr/bin/php命令执行。

例如：

```
#!/path/to/php
<?php
for($i=0; $i<50; $i++) {
    print "Hello world\n";
}
?>
```

2. Require/Include文件

为了减少重复代码，PHP可以通过使用require或include函数从外部文本文件引入代码。下面是一个使用该函数的例子。这种做法值得鼓励，因为它允许你在另一个文件（如config.php）中定义某些全局常量或将相关函数组合在一起放入其他文件中。

config.php包含如下内容：

```
<?php
$debug=true;
?>
```

下面是index.php的内容：

```
<?php
require("config.php");

if($debug) {
    echo "We are set to debug";
}
?>
```

require和include之间的区别很小（如果require失败，这是一个致命错误，而include只会导致一个警告）。在较大的项目中，我们往往很难跟踪在何处引入了哪个文件，而引入一个文件两次将

产生问题（例如，产生函数已被定义的错误）。为了解决这个问题，我们可以使用这两个函数的姊妹函数require_once和include_once，它们的功能正如它们的名字所暗示的那样。

3. 神奇的类型转换

PHP在默认情况下会根据上下文对数据类型进行动态转换，例如：

```
<?php
// string + int = int
echo "5" + 5; // 10
// string + string = number
echo "5" + "5"; // 10
// string . int = string
echo "5" . 5; // 55
// string * string = int
echo "5" * "5abcdef"; // 25
// string / string = .666...
echo "2a" / "3b"; // division by zero error ( 0 / 0 )
// hex + int = int
echo 0x05 + 10; // 15
?>
```

一个空字符串的值为零，与一个未定义的变量一样。正因为如此，才会有==和!=运算符存在，它们会检查变量的类型及其内容。

```
<?php
$tmp1="";
$tmp2=False;

if ($tmp1 == $tmp2)
    echo "Equal with ==\n";
else
    echo "Not equal with ==\n";

if ($tmp1 === $tmp2)
    echo "Equal with ===\n";
else
    echo "Not equal with ===\n";
?>
```

这个例子将产生如下输出结果：

```
Equal with ==
Not equal with ==

Equal with ===
Not equal with ===
```

未定义变量可以使用isset函数或if语句来发现，如下所示：

```
<?php
if (!$tmp) { echo 'tmp not defined'; }
?>
or :
<?php
if(!isset($tmp)) { echo 'tmp is not set'; }
?>
```

但请注意，如果\$tmp包含“0”或“”，它将通过`isset`的测试，但`if (!$tmp)`的测试将失败。基于这个原因，你最好使用`isset`而不是一个简单的`if`检测。

4. 数组/列表/字典

PHP不区分一个哈希（或关联列表）与一个数组。例如：

```
<?php
$array = Array(5, 4, 3, 2, 1);
echo var_dump($array);
?>
```

将产生如下的输出：

```
array(5) {
    [0]=>
    int(5)
    [1]=>
    int(4)
    [2]=>
    int(3)
    [3]=>
    int(2)
    [4]=>
    int(1)
}
```

请注意PHP是如何基于变量被插入的顺序给它们分配适当的数组索引的。

```
<?php
$foo = Array(55 => "Bob", 4 => 9, "Roger" => "Rover");
echo var_dump($foo);
?>
```

产生如下输出：

```
array(2) {
    [55]=>
    string(3) "Bob"
    [4]=>
    int(9)
    ["Roger"]=>
    "Rover"
}
```

你可以使用`array_push`和`array_pop`函数将数组作为栈/队列来使用，也可以使用`$array[]`语法将变量添加到数组的结尾。正如下面显示的那样：

```
<?php
$foo = Array();
array_push($foo, "Bar");
$foo[] = "Foo"; // Array("Bar", "Foo")
$last = array_pop($foo); // returns "Foo", Array("Bar")
?>
```

从计算机科学的角度来看，有趣的是，一个数组中的元素顺序是保留的，即使它被严格作为一个哈希来使用。这是PHP的强大之处，它使得建立复杂、动态的数据结构变得非常容易。

5. 函数

下面显示的是在PHP中一个函数的例子：

```
<?php
function increment($arg1) {
    return $arg1 + 1;
}
echo "10 increments to : " . increment(10);
?>
```

函数允许你编写的代码块可以被复用。请务必在使用函数之前阅读下一节关于作用域的内容。

6. 作用域

在PHP中，变量的作用域一般是全局的，它将跨越可能由include或require函数引入的文件。但用户定义的函数有它们自己的本地作用域，如果想在函数内访问在函数之外分配的变量，必须使用global关键字（或\$GLOBALS数组），正如下面的例子说明的那样：

```
<zphp
$foo = "Something";
function do_something() {
    $tmp = $GLOBALS['foo'];
    echo "We should do " . $tmp . "\n";
}
function do_something_else() {
    global $foo;
    echo "We should do " . $foo . " else\n";
}
function destructive_call() {
    global $foo;
    $foo = "nothing";
}

do_something();
do_something_else();
destructive_call();
echo "Finally \$foo is : $foo\n";
?>
```

它将给出如下的输出：

```
We should do Something
We should do Something else
Finally $foo is nothing
```

如果希望一个变量在函数被调用之后还能保留其值，可以使用关键字static。

7. 循环和迭代子

如果想遍历一个数组，有几种方法可以完成这个工作：

```
foreach($array as $value) {
    echo "value is $value";
}
foreach($array as $key => $value){
    echo "key is $key and value is $value";
}
```

如果想遍历一个随机的对象，只需要定义一个next()方法，例如：

```
<?php
class Shelf {
    $objects = Array("a", "b", "c");
    function next() {
        // delegate the next call to the array
        return next($objects);
    }
}
$a = new Shelf();
foreach($a as $string) {
    echo "Shelf Entry : $string\n";
}
?>
```

8. 对象

当在PHP中调用一个对象的方法时，我们使用语法：object->method(parameters,...)。如果你想访问一个对象中的public变量，你可以使用：object->。

PHP 4的面向对象实现并不是特别完善。PHP 5对它进行了完全重写，现在PHP 5支持了多重继承、抽象、异常和接口。

PHP的大部分面向对象语法与其他面向对象语言类似，但也有不同之处，如下面这个例子：

```
<?php
class Foo {
    protected $name;
    function __construct($name) {
        $this->name = $name;
    }
    public function getName() {
        return $this->name;
    }
}
class Bar extends Foo {
    protected $password;
    function __construct($name, $password) {
        $this->password = $password;
        parent::__construct($name);
    }
    public function getPassword() {
        return $this->password;
    }
}
$foo = new Foo("Joe");
$bar = new Bar("Joe", "secret");
echo "Name: " . $bar->getName() . "\n";
echo "Password: " . $bar->getPassword() . "\n";
?>
```

请注意在上面例子中是如何使用关键字\$this来引用类变量的。否则，它将假设只返回一个存在于函数作用域范围内的变量。

与其他语言一样，PHP支持构造器和多态。但它不能使用多个定义（不同的参数）来重载一个方法。如果想这么做，需要使用可选参数，如下所示：

```
<?php
class Foo {
    function do_something($arg1, $arg2, $arg3 = FALSE) {
        if($arg3 === FALSE) {
            // do one thing
        }
        else {
            // do something else...
        }
    }
}
$foo = new Foo();
$foo->do_something("a", "b");
$foo->do_something("a", "b", Array(1,2,3,"four"));
?>
```

这个参数允许我们“欺骗”PHP以创建不同的定义。如果调用时没有提供第三个参数，则它的值假设为构造函数中提供的值。

9. 引用传递/值传递

传统上，PHP一直是一个值传递语言并使用了按需复制技术。例如，如果我们传递一个大数组给一个函数，该函数遍历数组并修改接近数组结尾的一个元素，则该函数将首先使用原数组进行遍历，只有当数组被修改时才会发生额外的复制操作。

如果你想强制使用引用传递，则需要在变量前面加上一个&符号，如下面的例子所示。

一般来说，所有参数都是通过值传递的，除非你使用对象，在这种情况下：

```
<?php
class MyObject {
    public $foo = 0;
}

function do_it($foo) {
    $foo++;
}

function doItObj(&$object) {
    $object->foo++;
}

$bar = 0;
$myobject = new MyObject();

do_it($bar);
echo "Bar is $bar\n";
do_it(&$bar);
echo "Bar is $bar\n";

do_it($myobject->foo);
echo "Object has ($myobject->foo)\n";
```

```
doItObj($myobject);
echo "Object has {$myobject->foo}\n";
?>
```

输出：

```
Bar is 0
Bar is 1
Object has 0
Object has 1
```

当在一个过程上下文中使用PHP时，所有的变量都是值传递而不是引用传递。如果想使用引用传递，可以使用`&$variable`符号或使用`=&赋值`。

(补充说明，我们一般不赞成像上面例子那样在使用调用时引用传递，因为它将使得代码复杂化(而且函数未必可以安全地使用引用传递)。通常更好的做法是：

```
function do_it(&$foo) {
    $foo++;
}
$bar=0;
do_it($bar)
echo "Bar is now : $bar\n";
```

这样做使得无论函数在哪里被调用，它总是以引用传递的方式操作。

14.4.2 错误处理

在本节中，我们将介绍在PHP脚本执行过程中检测和处理错误的一些方法。

PHP错误处理的传统做法是让程序员检查函数执行后的返回值，因此，我们经常会看到类似下面这样的代码：

```
<?php
$db_conn = mysql_connect(".....");
if(!$db_conn) {
    // handle error condition here. Call mysql_error() to get error message.
}
?>
```

不幸的是，这依赖于程序员不会忘记检查各种函数的返回值，而且它还使得代码更加冗长——因为需要使用许多额外的if语句来检查返回值。解决这个问题的一种方法是使用trigger_error函数(从PHP 4.0.1开始提供)，如下所示：

```
<?php
$foo = bar("54");
function bar($arg1) {
    if($arg1 > 50) {
        trigger_error("argument to function bar cannot be greater than 50",
E_USER_ERROR);
    }
}
```

有关错误处理的更多信息请访问<http://uk2.php.net/manual/en/function.error-reporting.php>。

类似这样的警告信息通常会直接显示在HTML文档中。这种做法一般不可取而且它将使得应用程序看起来不是非常专业。如果你认为错误并不是很重要，可以使用error_reporting函数控制错误的输出或使用ini_set()，如下所示：

```
<?php
// display all warnings and notices - ideal for use during development.
error_reporting(E_STRICT | E_ALL);
// alternatively :
// ini_set("error_reporting", E_ALL);
ini_set("display_errors", true);
// Note: won't be of any use for fatal startup errors.
// Useful for debugging only.
ini_set("log_errors", true);
ini_set("error_log", "/tmp/php.error.log");
?>
```

有时候我们需要抑制一个错误信息的报告，这可以通过在一个函数名前加上@符号来完成，如下所示：

```
<?php
$file = @file("http://www.example.com");
if(!$file) {
    // handle error condition.
}
?>
```

但在大多数情况下，最好是定义一个全局错误处理函数，如下所示：

```
<?php
function my_error_handler($errno, $errstr, $errfile, $errline){
    // generate pretty html page
}
set_error_handler("my_error_handler");
.... code that may generate errors ....
?>
```

这个错误处理函数可以处理传递过来的任何错误并以通用的格式显示适当的错误信息。现在，你可以在PHP程序的任何位置通过这个错误处理函数来显示发生的错误事件。

14.4.3 异常错误处理

从PHP 5开始，PHP已支持了包括异常和多重继承的完整OO模型。PHP的异常处理代码看起来与Java的异常代码非常类似。例如：

```
<?php
try {
    // something that can cause problems here
}
catch(DbException $ex) {
    echo "Database error : " . $ex->getMessage();
}
catch(Exception $ex) {
```

```

        echo "Unknown Error : " . $ex->getMessage();
    }
?>

```

PHP提供了一个全局异常处理函数，它可以向最终用户提供“友好”的错误信息，例如：

```

<?php
class DbException extends Exception {
}
class ValidationException extends Exception {
}

function my_exception_handler($exception) {
    if($exception instanceof DbException) {
        echo "Database error occurred. Please contact support\n";
        // don't print db exceptions to users; they're meaningless.
    }
    if($exception instanceof ValidationException) {
        echo "The page was unable to validate the data you supplied to it. Please
try again\n";
        echo "Error: " . $exception->getMessage();
    }
    // etc.
}
set_exception_handler('my_exception_handler');

// randomly throw one or the other.
$rand = rand(0,1);
if($rand == 0) {
    throw new DbException('sql error at line 44');
}
else {
    throw new ValidationException("invalid input, can't contain '-'");
}
?>

```

运行这段代码将产生一个用户友好的错误，而不是一个丑陋的PHP错误信息。

```

$ php php-04.php
The page was unable to validate the data you supplied to it. Please try again
Error: invalid input, can't contain '-'
$ php php-04.php
Database error occurred. Please contact support

```

正如前面提到的，PHP在使用对象或函数时一般采用的是值传递，除非将参数指定为&\$foo或在赋值时使用=&或给函数传递一个对象，在这些情况下使用的是引用传递。

14.4.4 优化技巧

本节将举例说明一些优化PHP代码的简单方式，这些方式都是普遍适用的。本节还介绍了xdebug PHP模块，它可用于进行代码剖析。

几乎与所有其他语言一样，开发时最好以一种相对未优化的方式进行。对代码进行显著的优化往往需要花费程序员大量的时间，而且如果没有正确地判断，所进行的优化可能对应

用程序的性能只有略微的提高。但一般来说，在开发时采用一些简单的技巧（例如下面介绍的这些）并不会对代码的执行速度或代码的清晰度有多大的影响，所以我们应将程序的优化放到开发过程的最后进行。在大多数情况下，程序员在开发过程中花费在优化上的时间可能是成本大于效益。

1. 字符串

如果使用大量的循环，PHP需要扫描所有双引号标记的字符串以查找其中是否有变量，如下例所示：

```
<?php
$number = 500000;

echo "It is now : " . microtime(true) . "\n";
$start = microtime(true);
for($i=0;$i<$number;$i++) {
    $temp = "This is a plain double quoted string";
}
$end = microtime(true);
echo "$number double quotes (variable-less) took : " . ($end - $start) . "\n";

$start = microtime(true);
for($i=0;$i<$number;$i++) {
    $temp = 'This is a plain single quoted string';
}
$end = microtime(true);
echo "$number single quotes (variable-less) took : " . ($end - $start) . "\n";

$start = microtime(true);
for($i=0;$i<$number;$i++) {
    $temp = "This is a string $i";
}
$end=microtime(true);
echo "$number double quotes took : " . ($end - $start) . "\n";

$start = microtime(true);
for($i=0;$i<$number;$i++) {
    $temp = 'This is a string' . $i;
}
$end = microtime(true);
echo "$number single quotes took : " . ($end - $start) . "\n";
?>
```

它的输出是：

```
It is now : 1141551380.13
500000 double quotes (variable-less) took : 1.12232685089
500000 single quotes (variable-less) took : 1.11122107506
500000 double quotes took : 3.25105404854
500000 single quotes took : 1.71430587769
```

请注意上面四个例子输出时间的不同。很明显，在字符串中包含变量将显著降低循环的性能。

2. 数组

当对数组进行遍历时，与使用其他方法相比，使用foreach更方便并且它提供了更好的性能，如

下所示：

```
<?php
$number=50000;

$array1 = Array();
for($i=0;$i<$number;$i++) {
    $array1[] = $i;
}

$start = microtime(true);
reset($array1);
foreach($array1 as $key => $value) {
    $foo = $value;
}
echo "It took : " . (microtime(true) - $start) . " using a foreach over $number
\n";

$start = microtime(true);
$array_len = sizeof($array1);
for($i=0;$i<$array_len;$i++) {
    $foo = $array1[i];
}
echo "It took : " . (microtime(true) - $start) . " using a for loop.\n";

$start = microtime(true);
reset($array1);
while(list($key, $value) = each($array1)) {
    $foo = $value;
}
echo "It took : " . (microtime(true) - $start) . " using each/list etc.\n";
?>
```

它的输出是：

```
It took : 0.161890029907 using a foreach over 50000
It took : 0.309633970261 using a for loop.
It took : 2.0244410038 using each/list etc.
```

`foreach`循环所花费的时间几乎比`for`循环减少了一半，而`while`循环所花费的时间显著增加。在创建循环代码时，这是需要记住的一个重要特点。

3. Xdebug：剖析和调试PHP代码

`xdebug`是一个开放源码的第三方模块，它可以安装到你的PHP环境中。它提供了一些对开发有用的功能，主要包括：代码剖析、脚本执行分析和调试支持。本节重点介绍`xdebug 2.0`版本的使用。

`xdebug`可以从www.xdebug.org上下载，该网站还包含它的安装说明。

当安装了`xdebug`并且PHP经过配置装载它之后，你将立即受益于生成错误信息时它所产生的漂亮的栈跟踪。但是，它其实可以做更多的事情。

由于其本质，`xdebug`需要在PHP自身的初始化过程中被装载和配置。因此，我们不可能通过使用PHP的`ini_set`功能来改变`xdebug`的设置。我们在本章前面曾经简要地介绍过`php.ini`文件——这里我们将对该文件进行有意义的修改：

```

zend_extensions=xdebug.so
xdebug.profiler_enable=1
xdebug.profiler_output_dir = "/tmp"
xdebug.auto_profile = 1

```

上面的设置告诉xdebug自动剖析代码并将输出保存到/tmp目录下的文件中（文件名类似/tmp/cachegrind.out.\$number）。

因此，如果我们执行下面的PHP文件，xdebug将“神奇地”生成一个cachegrind.out文件，我们可以使用诸如kcachegrind这样的应用程序来查看它。

```

<?php
if(!extension_loaded('xdebug')) {
    die("<a href='http://xdebug.org'>Xdebug needed</a>");
}
function foo($string) {
    $val = preg_replace("/foo/", "bar", $string);
    $tmp = explode(" ", $val);
    $val = implode(" ", $tmp);
    echo "Val: $val\n";
}
function bar($string) {
    return $string . "BAR";
}

for($i=0;$i<100;$i++) {
    foo("kettle of fish near a foo bar");
}
?>

```

当上面文件的输出被装载进kcachegrind之后，你将看到如图14-1所示的内容。

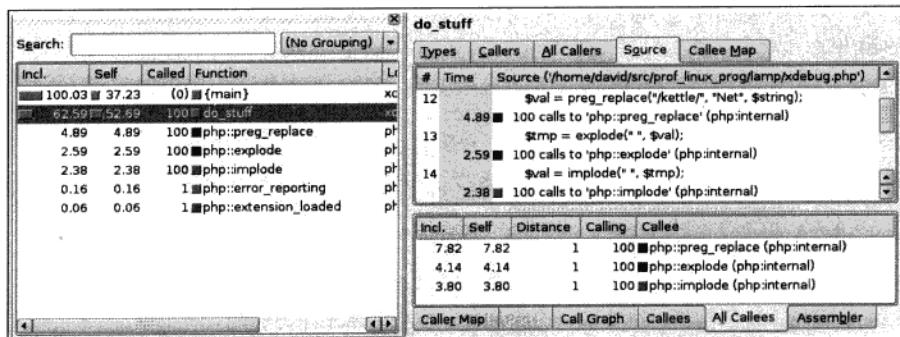


图14-1 当cachegrind.out文件的输出被装载到kcachegrind中后显示的结果

虽然这里使用的页面比较简单，但它确实提供了适当的时序（此外，这里没有显示的内存使用情况）。这允许你对常用函数进行优化并采用前后分析法进行测试。

kcachegrind给出一个完整的图形化输出以显示函数名（包括它们是否是PHP的内置函数或来自一个包含文件）、内存和CPU使用率、百分比和调用计数（一个函数被调用的次数），如果提供了源代

码（即它不是一个内置函数），它还可以被查看。

需要指出的是，使用xdebug可能会导致过多的内存或磁盘活动——但对于诊断一个脚本的性能问题来说，它的价值是无法衡量的。

xdebug还可被配置为在发生错误时连接到一个远程调试器，而且它可以很好地与ActiveState的Komodo集成开发环境整合，如图14-2所示。

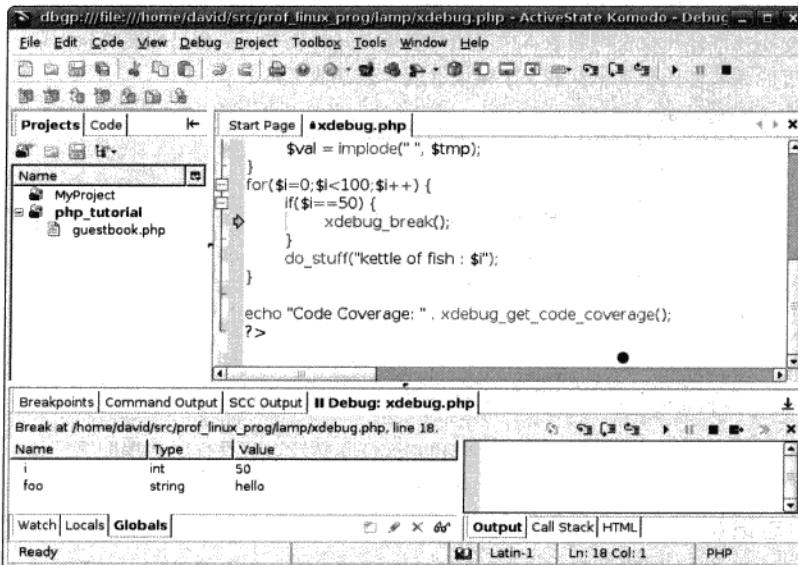


图14-2 xdebug在发生错误时连接到远程调试器

Komodo允许用户设置断点、修改变量值，以及做一个调试器允许你做的所有其他事情。

要配置xdebug和komodo，需要对php.ini文件做出如下的修改：

```
xdebug.remote_host=IP_Address_of_Host_Running_Komodo
xdebug.remote_mode=req
xdebug.remote_autostart=On
xdebug.remote_enable=On
xdebug.remote_port = 56973
```

remote_port设置用于匹配在Komodo的Debug→Listener Status中显示的端口号。看起来Komodo在每次启动时都会选择一个随机的端口号。

ActiveState的Komodo可以从www.activestate.com/Products/Komodo上下载，它有90天的免费试用期。

14.4.5 安装额外的PHP软件

PEAR项目(<http://pear.php.net>)包含大量针对PHP的有用的附加软件包。它应该是默认安装的，如果你的系统没有安装它，你有必要使用如apt-get install pear这样的命令来安装它。

当安装了pear之后，你可以在/usr/share/php目录中看到许多有用的类。所有类的完整文档可以在PEAR项目的网站上找到。如果你想安装额外的软件，你可以使用apt-get命令或使用如下所示的pear内置命令来安装它：

```
pear list-all | grep -i xml
pear install XML_Util
```

上面的命令演示了如何安装XML_Util PHP软件包。

pear命令还允许你在线使用新版本（当然希望是更好的版本）更新已安装的软件包。根据软件包的不同，这可能需要你在系统中安装一个编译器或一些其他工具。

```
$> pear list-upgrades
Available Upgrades (stable):
=====
Package          Local      Remote      Size
Archive_Tar      1.1 (stable) 1.3.1 (stable) 14.8kB
Date             1.4.5 (stable) 1.4.6 (stable) 53kB
HTML_Template_IT 1.1 (stable) 1.1.3 (stable) 18.9kB
HTTP_Request     1.2.4 (stable) 1.3.0 (stable) 13.6kB
Log               1.9.0 (stable) 1.9.3 (stable) 34kB
Net_UserAgent_Detect 2.0.1 (stable) 2.1.0 (stable) 9.5kB
PEAR              1.3.6 (stable) 1.4.7 (stable) 274kB
XML_RPC          1.4.0 (stable) 1.4.5 (stable) 29kB
$> pear upgrade-all
```

请注意，不要使用pear升级一个由你的Linux发行版软件包管理程序管理的软件包，因为这两种机制可能使用不同的版本更新同一个文件，这可能会在今后给你带来麻烦。

14.4.6 日志记录

对错误、信息和一般调试消息的记录是任何项目必备的一个组成部分——不管这个项目是大是小。在某些情况下，只使用echo并将调试信息显示在如HTML这样的页面中也是可以接受的，但这并不是一种理想的做法，在可能的情况下将日志记录到外部日志文件中始终是我们首选的方案。日志记录除了是软件开发的一个基本组成部分以外，它对于部署后的应用程序也至关重要。但应注意，不要过分记录日志，因为这可能会影响性能或导致产生巨大的日志文件，它将消耗一个活跃站点上的所有的磁盘空间！软件部署后的日志记录为软件的支持和维护提供了便利——例如，检测安全问题、跟踪应用程序的使用情况，或当用户遇到一个错误时，查看两天前发生了什么故障。

我们在这里重点介绍PEAR::Log软件包。

如果该软件包还没有被安装，请使用pear install Log安装必需的文件。

Log软件包的使用比较简单。但为了方便你对日志记录进行控制，你最好将它的配置和设置放在一个地方以便更容易地启用或关闭（或调整）它。在示例应用程序中，我们把它放在config.php文件中，每一个页面都包含该文件。

日志记录用法的一个例子如下所示：

```
require_once 'Log.php';

$logger = Log::factory('file', '/path/to/log.file', 'ANameForThisLogger');
$mask = Log::UPTO(PEAR_LOG_DEBUG);
```

```
$logger->setMask($mask);

$logger->debug("this is a debug message");
$logger->error("this is an error message");
```

关于Log软件包的更多信息请查看它的主页<http://pear.php.net/package/Log>和www.indelible.org/pear/Log。

通过使用不同的日志记录方法（`debug`、`error`、`info`、`emerg[ency]`），我们可以将信息分类并进行有选择的记录（例如，调试信息可能过于冗长并消耗硬盘资源，它并不适合在一个活跃的应用站点上记录）——这是通过`setMask`功能完成的。

日志信息可以被很容易地定向到主控台、电子邮件或文件（如上所示）。

14.4.7 参数处理

当从用户的请求（GET或POST）中接收值时，最好进行某种形式的验证。这既可能是如长度检查这样简单的验证，也可能是一个复杂的正则表达式验证。不幸的是，为用户提供的输入添加这样的检查既乏味又重复。一些框架（例如，Symfony，www.symfony-project.com）可以在这方面提供协助。

下面是一个检查用户在HTML表单中所提供的输入的例子：

```
<html>
<head></head><body>
<form action='$PHPSELF' method='post'>
Name <input type='text' name='the_name' value=''%gt;
<input type='submit'>
</form>

<?php
if(isset($_POST['the_name'])) {
    $name = $_POST['the_name'];
    if(preg_match("/[^A-Za-z0-9 ]/", $name) ) {
        // name contains a character that isn't ABCDE....Z, abc....z,
        // 0123456789 or a space.
        // Display error message, or send user back to entry form
        // for resubmission.
    }
    // Do something with $name.
    echo "<p>You entered : $name</p>";
}
?>
</body></html>
```

此外，当在Web页面上显示用户提供的变量或将用户提供的变量保存到数据库中时需要小心。对数据库来说，最好的做法之一是使用prepared语句，因为参数将由所用的函数库进行正确的处理。我们必须注意要确保用户不能在表单域中嵌入HTML的<或>括号，因为这些数据随后将未经改动地重新显示给用户——这将允许用户嵌入JavaScript或其他跨站脚本（XSS）攻击。在为内部网络开发应用程序时，你可以决定是否有必要使用这类的检查。

为了对付XSS攻击，PHP包含了`htmlentities`函数。它执行字符的转换，例如它可以将<转换为`<`，浏览器将把它显示为<，但它不会成为页面结构的一部分。

一般参数处理采取的是读取\$_array（\$_POST、\$_GET、\$_SESSION或\$_COOKIE）中参数的形式（如上面的例子所示）。PHP本身处理这个数组的产生，作为程序员，我们需要做的是检查它所提供的值的内容并以适当的方式使用它。

一个执行如下两个任务的PHP页面是相当普遍的——一个是在默认情况下显示数据的任务，而第二个任务是更新数据。当显示页面时，我们将检查一个特定的POST/GET/COOKIE/SESSION参数是否被设置，如果是，数据就将被更新。

注册为全局变量

`register_globals`是php.ini里的一个设置，有关php.ini的更多信息请查看PHP配置的相关内容。

PHP以前的版本包含一个功能，从一个表单提交的（或包含在一个cookie中的）变量将自动以它们的名字（即表单元素名`my_name`成为`$my_name`）在脚本的范围内可用。不幸的是，这造成了PHP应用程序中的许多漏洞，因为它允许恶意用户通过设置一些本应是未设置的变量来改变代码的正常执行。为了解决这个问题，所有最新版本的PHP都默认关闭了这个功能，而且几乎所有的PHP应用程序都已被重写不再使用这个功能——但还是有一些程序仍然在用它。如果你正在安装一个需要开启`register_globals`的应用程序，请小心。

14.4.8 会话处理

PHP在幕后处理一些会话特征——例如向客户端发送cookie头。通过启用php.ini中的适当设置(`session.auto_start`)，PHP可以被设置为自动启用会话。但在大多数情况下，我们并不推荐这样做，因为如果你在一个会话中保存对象而会话又在对象的定义被装载之前启动，你就可能会遇到问题。

`php.ini`文件设置的更多信息请见PHP安装和配置的相关内容。

在默认情况下，你需要调用`session_start()`函数来使用HTTP会话。该函数必须在任何文本输出给用户之前执行，否则文本的输出将导致HTTP头发送给客户并阻止用于会话跟踪的cookie的发送。

```
<?php
session_start();
$_SESSION['key'] = "Something";
$_SESSION['key'] = Array("some", $values);
?>
```

历史上还有两个和会话有关的函数`session_register`和`session_unregister`，但实际上它们是多余的，而且容易引起混淆。直接读取`$_SESSION`数组会更容易（而且打字也较少）。

14.4.9 单元测试

单元测试实践——早已在Java社区中被接受的做法——最近已在PHP世界中越来越流行，并且这么做有着充分的理由。修改脚本语言中的代码会引入错误。虽然脚本仍然可以在99%的时间内完美工作，但这些错误肯定会在某个时候伤害到所有的脚本开发人员。大多数开发人员都认为测试无趣而且重复——它确实如此。使用一个测试框架同样需要做大量的工作以实现初始测试，这仍然是相当枯燥和重复的。但利用框架可以使得重复测试变得更容易和自动化，而且它还减少了用户在执行测试时犯错误的机会。

PHP有许多单元测试框架，它们通常看来都是基于junit方法的。

有关junit的更多信息请访问<http://junit.sf.net>。

PHPUnit2是一个PEAR项目，它提供了一个完整的单元测试框架，而且它还可以很好地与xdebug整合以提供测试代码覆盖率分析。但不幸的是，当传递全局变量给函数时，它会引起问题——由于这个原因，我们在例子中使用的是SimpleTest。

SimpleTest是另一个PHP测试框架，它和PHPUnit非常类似，而且两者用于产生测试的代码也几乎完全相同（这至少使得改变测试框架变得非常容易）。

SimpleTest可以在www.lastcraft.com/simple_test.php上找到。

与所有派生自junit的测试用具一样，setUp和tearDown例程用于创建和清理测试环境。它们应在每次测试之前运行以确保数据被初始化为一个已知状态。

我们通过使用以test开头的方法名来定义一个测试。测试用具使用反射/自省来检查类并调用所有以test为前缀的方法。

下面显示的是一个非常简单的例子：

```
<?php
require_once 'simpletest/unit_tester.php';
require_once 'simpletest/reporter.php';

class VerySimpleTest extends UnitTestCase {

    public function setUp() {
        $this->foo = "Bar";
    }
    public function tearDown() {
        unset($this->foo);
    }
    public function testSimple1() {
        $this->assertTrue(isset($this->foo), "this->foo not defined!");
    }
    public function testSimple2() {
        $this->assertTrue($this->foo === "Bar", "Bar != Bar ??");
    }
}
$test = new VerySimpleTest();
$test->run(new TextReporter());
?>
```

希望看到一个更复杂例子的读者可以查看CD-ROM上的dvd-lib-smarty-test代码（特别是DBTest.php和LogTest.php源文件）。

测试用具本身提供了断言方法（assertTrue、assertFalse和assertEqual）。在检测失败时，提供一个消息通常是帮助调试的一个好主意。

运行上面的测试将产生如下的输出：

```
VerySimpleTest
OK
Test cases run: 1/1, Passes: 2, Failures: 0, Exceptions: 0
```

SimpleTest可以被配置为“伪装”向某个文件提交表单值，这允许对表单及其相关逻辑进行自动

化测试。这类功能应该是你对所使用的任何测试套件的最起码的要求了。在更复杂的情况下，我们可以将多种测试放在一个较大的批处理文件中运行。

如果你正在一个团队项目中工作，针对中央服务器安排一个基于每小时的任务是一个好主意，任务的内容从subversion签出源代码、运行测试并向整个团队报告发现的任何错误。这样的自动化做法有助于确保subversion中（即它管理的所有项目）代码的质量，并使得你不必等待有人来解决那些他们在昨晚9点不慎引入的错误。最终经过几次窘迫之后，大多数开发人员都会在编写完代码之后首先自行对代码运行测试，然后才将他们的改动签入subversion（或其他版本控制系统）。这类做法确实需要测试被不断更新、补充和维护，但它带来的好处是巨大的。

但不管你采取什么样的做法，它都比手工花费数小时来测试一个应用程序要强，而这样做的目的仅仅是为了发现并修复一个错误，然后还不得不手工地再次重复所有的测试！

14.4.10 数据库和PHP

在默认情况下，PHP可能支持MySQL或PostgreSQL，或两者都支持。厂商通常会提供一个php-(mysql|pgsql)软件包，你应该安装它（它通常作为一个Apache模块，正如我们在前面14.2节中讨论的那样）。

PHP内置的所有数据库函数都是唯一的，它们都有自己独立的函数名（例如，mysql_connect、pgsql_query，等等）。这种命名方式导致难以创建跨数据库的应用程序——历史上，应用程序一直倾向于支持MySQL，对其他数据库的支持要么不存在要么就是滞后于主开发分支。有些项目实现了它们自己的数据库无关层，但这实际上为应用程序的开发增加了一个多余的负担。

为解决上述问题，PEAR（PHP扩展与应用资源库，<http://pear.php.net>）项目提供了一个DB透明模块，它支持各种数据库后端，并且在改变数据库时，除了连接字符串以外，不需要修改其他代码。但我们仍需要小心，要坚持使用标准SQL，因为我们很容易就放入一些某个RDBMS专用的命令或将来自你可能需要迁移到的RDBMS没有实现的命令。你将在本章最后的DVD库实例中看到使用PearDB的代码示例。

14.4.11 PHP框架

当你开始编写一个更高一级的神奇应用程序时，将它建立在他人所做工作的基础之上通常是可取的。这样做有利于代码复用、减少维护需求，并且希望能改善性能/安全和加快产品上市/开发。

现在有很多PHP框架可用。在低端，有类似PEAR函数库这样的项目，它们旨在为PHP应用程序提供一些最好的实用构建块——例如非数据库特定的数据库方法、日志记录方法等。

在高端，有类似Drupal或Joomla（或Mambo）这样的CMS应用程序。它们提供了几乎所有需要的基础架构（登录、用户跟踪、搜索功能、购物篮等），并且可以在很大程度上被修改和定制，从而几乎可以为所有类型的Web应用程序提供构建基础。

PHP应用程序开发的一个问题是开发出一个可以正常工作的页面太容易了，但你关注的主要内容（模型，视图和控制器）却混合在一个页面中。其中部分原因是因为PHP自身是一个功能强大的语言，而且因此它也可以用于实现简单的模板。

Smarty是PHP项目之一，它为PHP提供了一个简单而功能又相当强大的模板框架。它所具备的有用构件有助于表单的生成。如有必要，它还可以在一个Smarty模板中嵌入PHP。

Smarty模板文件以.tpl为后缀名（按照惯例）。使用Smarty的一个例子见本章后面的DVD库示例。

Smarty还提供了一些有用的（尽管并不是特别优雅）例程，它们可以帮助处理日期以及其他一些难于处理的数据（如时间）的输入。

14.5 DVD 库

DVD库示例集将本章介绍的大部分内容汇集到一个连贯的例子中。我们并没有采用只提供一个最好的实践示例的方式，而是提供了一个版本不断更新的应用程序，它显示了代码是如何根据可读性和可维护性不断改进的。

DVD库是一个简单的DVD租借应用程序的开端，其目的是让用户可以找到并租借DVD。这个示例涵盖了这一行为的一个子集——针对“公共”用户的一个基本的搜索和浏览接口，以及一个管理接口——允许用户维护库中DVD的信息（添加、修改和删除）。

用户界面和外观在整个例子中都没有改变，因为v1之后的所有版本都使用相同的页头和页脚文件。软件功能在整个例子中也没有变化——除了出于方便性的考虑，其中一个版本只实现了功能的一个子集。

14.5.1 版本 1：开发者的噩梦

任何曾经被要求在已有代码上进行工作的开发者都会告诉你他们最糟糕的梦魇是发现与第一个例子中的代码有点相似的代码。它以最坏的方式实现DVD库中的一些功能，这个示例集显示了一个难于阅读、难于维护的应用程序，如果你不得不使用它，你将真正体会到烦心的感觉。当这个应用程序不断变大并且你无权从头开始重建它（或至少将它重构为一个完全不同的状态）时，它甚至将成为一个更大的梦魇。通过这个只实现了几个功能的例子我们可以看出，要想将你的代码搞得一团糟是多么容易。下面显示的代码摘自版本1目录中的list-all.php，它没有显示HTML代码（不幸的是，这些代码也和所有其他代码一样包含在同一个文件中，并且在所有其他文件中也重复包含）。

```
<?php
echo "<h2>All DVDs in the catalogue</h2>";
$db_host = 'localhost';
$db_user = 'prof';
$db_password = 'linuxprog';
$db_name = 'dvd';

$connection = mysql_connect($db_host, $db_user, $db_password);
if (!$connection) {
    print "Failed to connect:\n" . mysql_error() . "\n";
    die;
}

$sel = mysql_select_db($db_name, $connection);

if (!$sel) {
    print "Failed to connect to database $db_name.\n" . mysql_error();
    die;
}

$sql = "SELECT title from dvd";
```

```

$res = mysql_query($sql, $connection);

if(!$res) {
    trigger_error("Error executing query :" . mysql_error());
    die;
}

$all_dvds = array();
while($row = mysql_fetch_row($res)){
    array_push($all_dvds, $row);
}
if(!$all_dvds) {
    trigger_error("Error retrieving all_dvds :" . mysql_error());
    die;
}

mysql_close($connection);

if (sizeof($all_dvds)>0){
    echo "<ul>\n";
    foreach($all_dvds as $dvd){
        echo "<li><a href='view_dvd.php?part_title={$dvd[0]}'>".$dvd[0]."</a></li>\n";
    }
    echo "</ul>\n";
}
?>

```

当然，这里显示的代码并不多，它看上去可读性还不错。但请查看它的源文件，这些代码处于HTML代码的上下文中，然后查看实现第二个功能的源文件view_dvd.php，它更大一些，而且显得更凌乱。请注意这两个例子实现的功能都是非常简单的。

然后，你决定要改变站点的外观。对于没有使用任何模板机制的开发人员来说，这是一个相当普遍而且令人恐惧的事情，能够完成这个任务的人一定是一位勇敢的（而且相当愚蠢）开发者，即便是对于一个很小的系统，他也必须浏览每一个文件的代码并对HTML进行修改。但是，如果你想切换另一个数据库怎么办？当一个总是使用PostgreSQL的客户要求你的产品也使用该数据库怎么办？因为他们会付给你，所以你会对软件进行移植，但基于你目前使用的架构，这将是相当困难且易于出错的。一个良好的软件重构是必须的。

14.5.2 版本2：使用DB数据层的基本应用程序

版本2将应用程序从版本1的单一、充满复制的系统向前迈进了一步。要让你的应用程序更易于维护，你需要执行几个基本步骤。第1步（也可能是最重要的）是将程序基本表现从逻辑中分离（这是大部分复制会出现的地方，而且即使你有一个在功能方面非常稳定的应用程序，这也将是你所做的大部分修改所处的位置）。在实践中，你很难将它们完全分离，但一个好的开端是将它们分离到简单的页头和页脚文件中，这些文件包含基本的HTML和诸如菜单生成或认证校验这样的PHP代码。后面所有例子中使用的示例页头和页脚文件都可以在例子所在的目录中找到。

下一步是将数据库相关的代码提取到一个单独文件中。你将注意到第一个例子中的两个文件有大量重复的代码用于连接数据库、与数据库断开连接以及完成简单的查询结果获取操作。这些重复代码

可以被放入通用函数，从而可以在任何地方复用。将这两个元素从代码中分离将把上面的代码段转变为如下形式：

```
<?php

session_start();
require_once("dvd-db.php");
require_once("dvd-util.php");
include "../dvd-lib-common/head.php";

echo "<h2>All DVDs in the catalogue</h2>";

$all_dvds = get_dvd_titles("dvd");
if (sizeof($all_dvds)>0){
    echo "<ul>\n";
    foreach($all_dvds as $title){
        echo "<li><a href='view_dvd.php?part_title=$title'>$title</a></li>\n";
    }
    echo "</ul>\n";
}

include "../dvd-lib-common/foot.html"; ?>
```

它依赖于来自其他文件中的代码，包括来自数据库文件中的片断（在本例中是dvd-db_mysql.php，你也可以在dvd-db.php中将它替换为Postgres版本的php文件），这个代码已经比较小，而且非常清晰，也更易阅读了。这个文件只执行显示所需信息所需要的动作（其他例子还将包含处理表单输入数据所需的代码）。

数据库文件包含简单、通用的connect、disconnect、execute_query（针对有数据返回的查询）和execute_update（针对没有结果返回的insert、update和delete语句）函数，它们可以被任何特定函数调用。

```
<?php

function connect(){
    $db_host = 'localhost';
    $db_user = 'prof';
    $db_password = 'linuxprog';
    $db_name = 'dvd';

    $connection = pg_connect("host=$db_host user=$db_user password=$db_password
dbname=$db_name");
    if(!$connection) {
        print "Failed to connect:\n" . pg_last_error($connection) . "\n";
        die;
    }
    return $connection;
}

function disconnect($db){
    pg_close($db);
```

```

}

function execute_query($sql) {
    $db = connect();
    $res = pg_query($db, $sql);
    if (!$res) {
        trigger_error("Error executing query : " . pg_last_error($db));
        die;
    }

    $results = array();
    while ($row = pg_fetch_row($res)) {
        array_push($results, $row);
    }
    disconnect($db);
    return $results;
}

function execute_update($sql) {
    $db = connect();
    $res = pg_query($db, $sql);
    if (!$res) {
        trigger_error("Error executing update : " . pg_last_error($res));
        die;
    }
    disconnect($db);
}
?>

```

完成整个功能的是函数get_dvd_titles()，它用于从数据库中获取DVD标题并将一切组合在一起：

```

<?php
/**
 * Retrieve all titles for a DVD
 * @return an array containing the titles
 */
function get_dvd_titles(){
    $querystring = "SELECT title from dvd";
    $results = execute_query($querystring);
    $names = array();
    foreach($results as $row) {
        array_push($names, $row[0]);
    }
    return $names;
}
?>

```

将代码分离到不同的函数中并放入合适的文件使得每个段落更易于阅读和理解，这样我们就不需要查看所有的代码页以找到合适的段落。当然，如果主php文件（直接呈现给用户的文件，通常是你开始浏览的页面）包含了许多不同的文件，而你又不熟悉这个应用程序的结构，你可能不得不查看一些文件（或使用你喜爱的搜索工具）以找到正确的地点，但总的来说，这种做法是对可维护性的一种改进。

你可能也注意到有两个版本的数据库文件——分别针对PostgreSQL和MySQL。虽然它们在语法结构上非常相似，但它们使用不同的函数名，因此不能通用。唯一的解决方法是创建两个版本的函数，它们有相同的函数定义和行为。但值得欣慰的是，因为所有的数据库代码已从每个文件中移除，所以我们只需要在一个文件中进行替换并将它包含在需要的文件中即可。因为在显示代码和数据库之间的接口中提供了一个抽象层，所以我们可以根据特定的MySQL或PostgreSQL访问函数的差异包括适当的代码——同样的做法也适用于其他有PHP访问函数的数据库。

在进行上述修改之后，你将明显发现，除了代码的可读性以外，还有更多的原因会给程序带来问题，所以需要对代码进行调试。很明显，应用程序也迫切需要一些日志记录工具来为调试（和其他）目的记录信息。如果没有这些工具，可以使用一个专用的调试器（见前面的xdebug一节）或回过头来再使用老式的很容易出错的方法——在代码中包括print（或echo）语句。这通常需要进行细致的工作来添加代码（并在完成调试后删除它们）并将改变应用程序的行为。

需要注意的最后一件事情是这个版本的代码的错误处理（或在许多情况下缺少它）。当需要检测时，它使用传统的方式检查一个函数的返回值，如下面的从edit_dvd.php脚本中摘录的一段代码：

```
$success = edit_dvd($data);
if($success) {
    echo "\nDVD updated!\n";
}
else {
    echo "Problems were encountered while trying to update the DVD";
}
```

虽然从这个程度来说它并不是太凌乱，但整个函数调用栈都涉及对返回值的检测，这将在应用程序中添加很多非常类似的代码——尤其是对于有多个抽象层的大型应用程序更是如此。

14.5.3 版本 3：重写数据层，添加日志记录和异常

这个应用程序的下一个版本解决了在前一个版本的代码中反映出来的许多问题。

1. 增加数据库抽象

使用数据库特定的mysql_和pg_函数意味着你不得不编写两个版本的代码。但是，通过使用我们在前面介绍的数据库抽象层可以允许你生成“一次编写，随处使用”风格的代码。在本例中，我们使用PEAR DB模块来为数据库提供一个接口。从理论上讲，现在如果要改变数据库，唯一需要做的改动就是连接字符串（但在本例中，由于MySQL没有ILIKE运算符——它在不区分大小写的情况下对普通文本字符串使用LIKE。所以不幸的是，搜索函数将不得不做出改动）。

所以，我们现在有了一套新的可以被其他函数使用的通用函数集（注意我们现在已将连接字符串转移到配置文件中）：

```
<?php
require_once 'config.php';
require_once 'DB.php';

class DbException extends Exception{
    function DbException($msg){
        $this->message = "Error in DB Communications: " . $msg;
    }
}
```

```
function connect(){
    global $connection_string;
    global $logger;
    $db = DB::connect($connection_string/*, $options*/);
    if (DB::isError($db)) {
        $err = "Failed to Connect: " . $db->getMessage();
        $logger->err($err);
        throw new DbException($err);
    }
    $logger->debug("Connected to DB successfully!");
    return $db;
}

function disconnect($db){
    $db->disconnect();
}

function execute_query($sql, $values = false) {
    global $logger;
    $db = connect();
    $prep = $db->prepare($sql);
    if($values === false) {
        $values = Array();
    }
    $res = $db->execute($prep, $values);
    if(DB::isError($res)){
        $err = "Could not execute $sql in query: " . $res->getMessage();
        $logger->err($err);
        throw new DbException($err);
    }
    while($row =& $res->fetchRow()) {
        $results[] = $row;
    }
    return $results;
}
/***
 * Retrieve all titles for a DVD
 * @return an array containing the titles
 */
function get_dvd_titles(){
    global $logger;
    $querystring = "SELECT title from dvd";
    $results = execute_query($querystring, Array());
    $names = Array();
    foreach($results as $row) {
        array_push($names, $row['title']);
    }
    $logger->debug("Found : " . sizeof($names) . " dvd titles");
    return $names;
}
```

2. 异常

上面的代码还添加了非常精妙的异常情况处理。PEAR DB代码自身并没有为我们抛出合适的异常以传递给用户。当发生错误时，它并不返回错误代码或一个空的对象或数组，而是返回一个DB_Error对象，该对象包含（通常情况下）与发生错误有关的有用信息。我们可以通过检查返回值来判断是否发生了一个错误，然后我们可以抛出我们定制的异常，它包含错误发生的位置以及来自DB_Error对象的信息。

请注意，get_dvd_titles函数并没有做任何错误检查。它的前一个版本也没有这么做，但对于前一个版本来说，因为代码在每一个层次都需要进行检查，所以这确实是一种疏漏。在前者的情况下，如果在底层发生了一个致命错误，用户将看到一个并不是十分友好的错误画面。如果发生了一个非致命错误，它可能就被默默地忽略了——这不是一种理想的做法！而本例的实现正是我们想要的，异常将被向上传递直到它到达一个catch代码块或根本没有被捕获（在这种情况下，它将像任何其他错误一样显示给用户）。通过捕获异常，我们可以指定我们要处理异常的位置并指定捕获异常时要做的事情——因此，我们可以从捕获的异常中提取出信息并按照自己的想法来使用它。下面的编辑页面代码就是一个很好的例子，它显示了如何使用异常来保持你的代码清晰。它使得你很容易将“成功”代码（当一切都按期望的那样工作时依序执行的代码）和“失败”代码分开。在这个例子中，所有针对页面的正常操作都在顶部（在必要的检查和try代码块开始之后），而错误处理都隐藏在底部。

```
<?php
session_start();
require_once "dvd-db.php";
require_once "dvd-util.php";
ensure_authenticated();

if(array_key_exists('title', $_POST)){
    try{
        edit_dvd($_POST);
        add_message("\nDVD updated!\n");
        $_SESSION['last_edit'] = $title;
        header("Location: index.php");
    }catch(DbException $dbe){
        echo "<h3 class='error'>An error occurred while trying to update the DVD:
{$dbe->getMessage()}</h3>";
    }
}
else {
    echo "<h3 class='error'>No DVD was selected for editing. Please <a
href='edit-select.php'>Try Again</a></h3>";
}
?>
```

3. 日志记录

为了使应用程序的调试更简单并提高系统的可维护性，我们还使用前面介绍过的PEAR日志记录软件包增加了一个用于记录各种类型信息的系统。它的设置非常简单，只需在配置文件(config.php)中增加几行内容，然后它就可以被包括在任何需要记录日志的地方：

```
require_once 'Log.php';
```

```
$logger = Log::factory('file', '/tmp/php.log', 'AdvancedDVDLibrary');
$mask = Log::UPTO(PEAR_LOG_DEBUG);
$logger->setMask($mask);
```

下面是在数据库处理文件（dvd-db.php）中的另一个函数，它包括了各种级别的日志记录：

```
function check_password($username, $password) {
    global $logger;

    $querystring = "SELECT password from library_user WHERE username = ?";
    $results = execute_query($querystring, Array($username));
    $numRows = sizeof($results);

    if ($numRows > 1) {
        $err = "Error fetching password for $username for checking: 1 row expected,
$numRows found";
        $logger->err($err);
        throw new DbException($err);
    }
    elseif ($numRows == 0) {
        $logger->info("No record matching $username");
        return FALSE;
    }
    $row = $results[0];
    $enc_password = md5($password);
    if ($enc_password == $row['password']){
        $logger->info("Password authenticated for $username");
        return TRUE;
    }
    else return FALSE;
}
```

在这个例子中，你可以看到logger被用来提供info和error信息：info用来记录用户成功登录或用户试图登录的信息（登录的用户名不存在，当有人使用一个脚本试图以随机的用户名反复进行登录尝试以找出可以进行登录的用户名时，这将非常有用）；error用来记录超出正常范围的情况（在本例中，当一个本应返回一个记录的匹配语句返回了多个记录——这是绝对不能发生的事情）。注意logger没有给出任何敏感的信息（如密码），因为如果你的系统被侵入了，日志文件往往可以很容易地访问，而且在一些系统中，它记录着许多非常有趣的信息！

该示例输出到日志文件中的内容如下所示：

```
Mar 06 20:58:16 AdvancedDVDLibrary [info] Password authenticated for david
Mar 06 20:58:16 AdvancedDVDLibrary [debug] Connected to DB successfully!
Mar 06 20:58:16 AdvancedDVDLibrary [debug] Found : 11 dvd titles
Mar 06 20:58:16 AdvancedDVDLibrary [debug] Connected to DB successfully!
```

如果你不打算再记录调试信息，比如应用程序已正式开始使用，那么你只需在配置文件中改变日志记录级别即可。

14.5.4 版本4：应用模板框架

此时，我们已有了一个非常干净而整洁的应用程序，但我们还可以在将页面显示从逻辑代码中分离这方面做更多的事情。我们前面介绍过的Smarty模板系统提供了一个清晰而简单的机制来完成这一

工作。为了减少代码量，Smarty模板只处理页面的显示部分。正如先前解释的那样，Smarty通过自定义的模板与PHP整合，然后这些模板可以组合在一起并显示页面。这个示例最初的噩梦代码版本从数据库提取DVD列表的任务现在是通过使用一个普通的PHP文件和一个模板来生成和显示的。PHP文件除了包含正常的逻辑以外，还包含（阴影显示部分）：

```
<?php

session_start();
require_once("config.php");

$smarty = new Smarty;
$smarty->compile_check=true;

require_once("dvd-db.php");
require_once("dvd-util.php");

try {
    $all_dvds = get_all("dvd");
    $dvd_list = array();
    foreach($all_dvds as $dvd){
        array_push($dvd_list, $dvd['title']);
    }

    $smarty->assign("dvd_list", $dvd_list);
    $smarty->assign("header", "All DVDs in the Library");
} catch(DbException $dbe){

    display_error_page($smarty, "An error occurred while fetching the list of DVDs:
    ($dbe->getMessage())");
}
$smarty->display("list_dvds.tpl");
```

请注意display_error_page函数，它看起来如下所示（摘自dvd-util.php）。它用于显示一个简单的错误信息：

```
function display_error_page($smarty, $error){
    $smarty->assign("message", $error);
    $smarty->display("error.tpl");
    return;
}
```

与它关联的模板非常简单：

```
{include file="header.tpl"}
<h2>An Error has occurred</h2>
<ul>
    <p class='error'>{$message}</p>
</ul>
{include file="footer.tpl"}
```

模板文件只是简单地显示分配给Smarty对象的变量。特殊情况是对列表的处理，Smarty使用一个特殊的标记来遍历提供的列表。然后，name属性被用来确定循环中的位置（和数组）。

```
{include file="header.tpl"}  
<h2>{$heading}</h2>  
<ul>  
 {section name=dvd loop=$dvd_list}  
 <li><a href='view_dvd.php?part_title={$dvd_list[dvd]}'>{$dvd_list[dvd]}</a></li>  
{/section}  
</ul>  
{include file="footer.tpl"}
```

虽然这两个文件可能在整体上给软件带来更多的代码，但我认为将这样的文件（尤其是使用不带循环的模板）递交给非编程的网页设计师来实现与递交给他一些PHP代码相比，前者犯错的几率可能会大大减少。

14.6 本章总结

本章讨论了流行的Linux-Apache-MySQL-PHP（LAMP）Web编程环境。这些组件的结合可以产生一个完整的开放源码Web服务器、数据库和网页编程解决方案。虽然每个组件都是一个独立的开放源码项目，但它们可以很容易地集成到一个单一的服务器环境中。

Apache和PHP软件程序通常使用针对特定Linux发行版而安装的软件包，并且它们将一起被装载。MySQL软件则是一个单独的安装软件包，Linux发行版也提供了该软件包。在Linux服务器上安装这三个软件包非常容易，你只需运行你的Linux发行版的软件安装程序即可。

最后，本章演示了如何使用PHP程序通过标准HTML代码访问数据库中的数据。这一技术被世界各地的商业Web应用程序所使用。现在你已具备了这样的能力。

封面
书名
版权
前言
目录

第1章 L i n u x 简介

- 1 . 1 L i n u x 发展简史
 - 1 . 1 . 1 G N U 项目
 - 1 . 1 . 2 L i n u x 内核
 - 1 . 1 . 3 L i n u x 发行版
 - 1 . 1 . 4 自由软件与开放源码
- 1 . 2 开发起步
 - 1 . 2 . 1 选择一个 L i n u x 发行版
 - 1 . 2 . 2 安装 L i n u x 发行版
 - 1 . 2 . 3 沙盒和虚拟化技术
- 1 . 3 L i n u x 社区
 - 1 . 3 . 1 L i n u x 用户组
 - 1 . 3 . 2 邮件列表
 - 1 . 3 . 3 I R C
 - 1 . 3 . 4 私有社区
- 1 . 4 关键差别
 - 1 . 4 . 1 L i n u x 是模块化的
 - 1 . 4 . 2 L i n u x 是可移植的
 - 1 . 4 . 3 L i n u x 是通用的
- 1 . 5 本章总结

第2章 工具链

- 2 . 1 L i n u x 开发过程
 - 2 . 1 . 1 使用源代码
 - 2 . 1 . 2 配置本地环境
 - 2 . 1 . 3 编译源代码
- 2 . 2 G N U 工具链的组成
- 2 . 3 G N U 二进制工具集
 - 2 . 3 . 1 G N U 汇编器
 - 2 . 3 . 2 G N U 连接器
 - 2 . 3 . 3 G N U o b j c o p y 和 o b j d u m p
- 2 . 4 G N U M a k e
- 2 . 5 G N U 调试器
- 2 . 6 L i n u x 内核和 G N U 工具链
 - 2 . 6 . 1 内联汇编
 - 2 . 6 . 2 属性标记
 - 2 . 6 . 3 定制连接器脚本
- 2 . 7 交叉编译
- 2 . 8 建立 G N U 工具链
- 2 . 9 本章总结

第3章 可移植性

- 3 . 1 可移植性的需要
- 3 . 2 L i n u x 的可移植性
 - 3 . 2 . 1 抽象层
 - 3 . 2 . 2 L i n u x 发行版
 - 3 . 2 . 3 建立软件包
 - 3 . 2 . 4 可移植的源代码
- 3 . 3 硬件可移植性
 - 3 . 3 . 1 6 4 位兼容
 - 3 . 3 . 2 字节序中立
 - 3 . 3 . 3 字节序的门派之争
- 3 . 4 本章总结

第4章 软件配置管理

- 4 . 1 S C M 的必要性
- 4 . 2 集中式开发与分散式开发
- 4 . 3 集中式工具
 - 4 . 3 . 1 C V S
 - 4 . 3 . 2 S u b v e r s i o n
- 4 . 4 分散式工具
 - 4 . 4 . 1 B a z a a r - N G

	4 . 4 . 2 L i n u x 内核 S C M
4 . 5	集成化 S C M 工具
4 . 6	本章总结
第 5 章	网络编程
5 . 1	L i n u x 套接字编程
5 . 1 . 1	套接字
5 . 1 . 2	网络地址
5 . 1 . 3	使用面向连接的套接字
5 . 1 . 4	使用无连接套接字
5 . 2	传输数据
5 . 2 . 1	数据报与字节流
5 . 2 . 2	标记消息边界
5 . 3	使用网络编程函数库
5 . 3 . 1	l i b C u r l 函数库
5 . 3 . 2	使用 l i b C u r l 库
5 . 4	本章总结
第 6 章	数据库
6 . 1	持久性数据存储
6 . 1 . 1	使用标准文件
6 . 1 . 2	使用数据库
6 . 2	B e r k e l e y D B 软件包
6 . 2 . 1	下载和安装
6 . 2 . 2	编译程序
6 . 2 . 3	基本数据处理
6 . 3	P o s t g r e S Q L 数据库服务器
6 . 3 . 1	下载和安装
6 . 3 . 2	编译程序
6 . 3 . 3	创建一个应用程序数据库
6 . 3 . 4	连接服务器
6 . 3 . 5	执行 S Q L 命令
6 . 3 . 6	使用参数
6 . 4	本章总结
第 7 章	内核开发
7 . 1	基础知识
7 . 1 . 1	背景先决条件
7 . 1 . 2	内核源代码
7 . 1 . 3	配置内核
7 . 1 . 4	编译内核
7 . 1 . 5	已编译好的内核
7 . 1 . 6	测试内核
7 . 1 . 7	包装和安装内核
7 . 2	内核概念
7 . 2 . 1	一句警告
7 . 2 . 2	任务抽象
7 . 2 . 3	虚拟内存
7 . 2 . 4	不要恐慌
7 . 3	内核编程
7 . 4	内核开发过程
7 . 4 . 1	g i t : 傻瓜内容跟踪器
7 . 4 . 2	L i n u x 内核邮件列表
7 . 4 . 3	“ m m ” 开发树
7 . 4 . 4	稳定内核小组
7 . 4 . 5	L W N : L i n u x 每周新闻
7 . 5	本章总结
第 8 章	内核接口
8 . 1	什么是接口
8 . 2	外部内核接口
8 . 2 . 1	系统调用
8 . 2 . 2	设备文件抽象
8 . 2 . 3	内核事件
8 . 2 . 4	忽略内核保护
8 . 3	内部内核接口
8 . 3 . 1	内核 A P I
8 . 3 . 2	内核 A B I

第9章	8 . 4 本章总结
	L i n u x 内核模块
9 . 1	模块工作原理
9 . 1 . 1	扩展内核命名空间
9 . 1 . 2	没有对模块兼容性的保证
9 . 2	找到好的文档
9 . 3	编写L i n u x 内核模块
9 . 3 . 1	开始之前
9 . 3 . 2	基本模块需求
9 . 3 . 3	日志记录
9 . 3 . 4	输出的符号
9 . 3 . 5	分配内存
9 . 3 . 6	锁的考虑
9 . 3 . 7	推迟工作
9 . 3 . 8	进一步阅读
9 . 4	分发L i n u x 内核模块
9 . 4 . 1	进入上游L i n u x 内核
9 . 4 . 2	发行源代码
9 . 4 . 3	发行预编译模块
9 . 5	本章总结
第10章	调试
10 . 1	调试概述
10 . 2	基本调试工具
10 . 2 . 1	GNU调试器
10 . 2 . 2	Valgrind
10 . 3	图形化调试工具
10 . 3 . 1	DDD
10 . 3 . 2	E c l i p s e
10 . 4	内核调试
10 . 4 . 1	不要惊慌！
10 . 4 . 2	理解oops
10 . 4 . 3	使用UML进行调试
10 . 4 . 4	一件轶事
10 . 4 . 5	关于内核调试器的注记
10 . 5	本章总结
第11章	GNOME开发者平台
11 . 1	GNOME函数库
11 . 1 . 1	G l i b
11 . 1 . 2	G O b j e c t
11 . 1 . 3	C a i r o
11 . 1 . 4	G D K
11 . 1 . 5	P a n g o
11 . 1 . 6	G T K +
11 . 1 . 7	l i b g l a d e
11 . 1 . 8	G C o n f
11 . 1 . 9	G S t r e a m e r
11 . 2	建立一个音乐播放器
11 . 2 . 1	需求
11 . 2 . 2	开始：主窗口
11 . 2 . 3	建立G U I
11 . 3	本章总结
第12章	自由桌面项目
12 . 1	D - B U S : 桌面总线
12 . 1 . 1	什么是D - B u s
12 . 1 . 2	D - B u s基础
12 . 1 . 3	D - B u s方法
12 . 2	硬件抽象层
12 . 2 . 1	使硬件可以即插即用
12 . 2 . 2	H A L设备对象
12 . 3	网络管理器
12 . 4	其他自由桌面项目
12 . 5	本章总结
第13章	图形和音频
13 . 1	L i n u x 和图形

1 3 . 1 . 1	X视窗
1 3 . 1 . 2	开放式图形库
1 3 . 1 . 3	O p e n G L 应用工具包
1 3 . 1 . 4	简单直接媒介层
1 3 . 2	编写O p e n G L 应用程序
1 3 . 2 . 1	下载和安装
1 3 . 2 . 2	编程环境
1 3 . 2 . 3	使用G L U T库
1 3 . 3	编写S D L 应用程序
1 3 . 3 . 1	下载和安装
1 3 . 3 . 2	编程环境
1 3 . 3 . 3	使用S D L 库
1 3 . 4	本章总结

第14章 L A M P

1 4 . 1	什么是L A M P
1 4 . 1 . 1	A p a c h e
1 4 . 1 . 2	M y S Q L
1 4 . 1 . 3	P H P
1 4 . 1 . 4	反叛平台
1 4 . 1 . 5	评价L A M P平台
1 4 . 2	A p a c h e
1 4 . 2 . 1	虚拟主机
1 4 . 2 . 2	安装和配置P H P 5
1 4 . 2 . 3	A p a c h e _ B a s i c 认证
1 4 . 2 . 4	A p a c h e与S S L
1 4 . 2 . 5	S S L与H T T P认证的整合
1 4 . 3	M y S Q L
1 4 . 3 . 1	安装M y S Q L
1 4 . 3 . 2	配置和启动数据库
1 4 . 3 . 3	修改默认密码
1 4 . 3 . 4	M y S Q L客户端接口
1 4 . 3 . 5	关系数据库
1 4 . 3 . 6	S Q L
1 4 . 3 . 7	关系模型
1 4 . 4	P H P
1 4 . 4 . 1	P H P语言
1 4 . 4 . 2	错误处理
1 4 . 4 . 3	异常错误处理
1 4 . 4 . 4	优化技巧
1 4 . 4 . 5	安装额外的P H P软件
1 4 . 4 . 6	日志记录
1 4 . 4 . 7	参数处理
1 4 . 4 . 8	会话处理
1 4 . 4 . 9	单元测试
1 4 . 4 . 1 0	数据库和P H P
1 4 . 4 . 1 1	P H P框架
1 4 . 5	D V D库
1 4 . 5 . 1	版本1：开发者的噩梦
1 4 . 5 . 2	版本2：使用D B数据层的基本应用程序
1 4 . 5 . 3	版本3：重写数据层，添加日志记录和异常
1 4 . 5 . 4	版本4：应用模板框架
1 4 . 6	本章总结