PROGRAMMER TO PROGRAMMER

Beginning Linux Programming, 2nd Edition

Wrox程序员参考系列

（原书第2版）

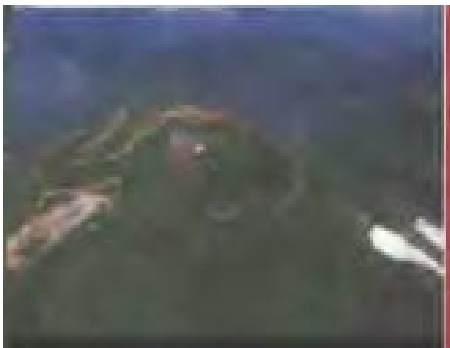
Linux 程序设计

（美）Neil Matthew Richard Stones 著

[illegible]

机械工业出版社
China Machine Press

乐思



《Linux 程序设计权威指南》

ISBN 7-111-08685-1/TP·1793
页数 710 页 定价 78.00 元

《Linux 内核源代码分析》

ISBN 7-111-08682-0/TP·1512
页数 658 页 定价 88.00 元

乐思

Beginning Linux Programming 2nd Edition

Linux 程序设计

（原书第2版）

- Linux入门知识
- Linux管理及开发工具
- 用Shell进行程序设计
- Web程序设计

系列丛书

- 《ASP3 高级编程》
- 《XML 高级编程》
- 《PHP 高级编程》
- 《Java 服务器高级编程》
- 《ASP3 程序员参考》
- 《ASP3 初级编程》
- 《Linux 程序设计》
- 《XHTML 初级编程》
- 《J2EE 服务器端高级编程》

适用水平：初、中、高级

ISBN 7-111-09322-4

9 787111 093220

www.china-pub.com

北京市西城区百万庄南街1号 100037
购书热线 (010)68995259、8006100280 (北京地区)

ISBN 7-111-09322-4/TP·2099
定价 78.00 元

Wrox 程序员参考系列

Linux 程序设计

（原书第2版）

（英） Neil Matthew
Richard Stones 著

杨晓云 王建桥 杨 涛 高文雅 等译

机械工业出版社
China Machine Press

本书介绍了Linux操作系统的基本知识，以及如何在Linux和其他UNIX风格的操作系统上进行程序开发。本书的主要内容包括：Linux的入门知识，使用Shell进行程序设计，Linux的管理及开发工具，Perl程序设计语言，Web程序设计等等。本书内容丰富、深入浅出、易于理解，还包含大量编程实例。适合Linux的初学者及希望利用Linux进行开发的程序设计人员阅读。

本书版权登记号：图字：01-2000-4096

图书在版编目(CIP)数据

Linux程序设计（原书第2版）/（英）马太（Matthew, N.）等著；杨晓云等译. -北京：机械工业出版社，2002.1
(Wrox程序员参考系列)
书名原文：Beginning Linux Programming, 2nd Edition
ISBN 7-111-09322-4

I. L… II. ①马… ②杨… 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（2001）第067258号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：宋 宏 张鸿斌
北京第二外国语学院印刷厂印刷 · 新华书店北京发行所发行
2002年1月第1版第1次印刷
787mm × 1092mm1/16 · 50印张
印数：0 001-4 000册
定价：78.00元

序　言

每一个计算机程序设计人员都有他自己的一套拿手绝活。每个人都有他自己的程序代码宝库，所有这些来源于他们自己对程序设计语言手册的学习理解，或者来源于通过Usenet学习到的方法。然而，对于那些对程序设计一无所知的人来说，不敢照搬照抄Usenet。但采用本书这样一种风格（指从小程序段开始介绍学习程序设计的风格）的著作可以说少之又少，这不能不说是一种奇怪的现象。在网络世界里存在着大量短小精悍而又切中问题核心的文档，它们直指程序设计或系统管理某些特定的领域。就拿Linux操作系统的文档项目来说吧，它的总篇幅大概只有3~10页，但它所涉及的内容却是包罗万象，从在同一台计算机上同时安装Linux和NT到把你的咖啡机接驳到Linux，可以说是应有尽有。真的，我一点也没有夸大其词。到http://sunsite.unc.edu/LDP网址上看看那里的mini-how-to（快速入门）索引就知道了。

从另一方面看，我们目前能够见到的书刊大多数或者可以说是一些学术墓园——它们既详尽又过于全面，读者很难有时间把它们一读到底，只有很少的书籍尝试着对大量实际应用领域的基本概念和基本做法进行介绍和讨论。本书就是这类书籍之一，它是对程序员个人编程宝库的补充，是通过对程序代码的“破译”（从程序员的手稿里读出正确的内容）而有机地组织起来的。

这本书目前已经是经过修改的第二版了，又扩展了许多内容，因为Linux本身也是在不断扩展中的。增加的主要内容有如何编写线程化程序（有时候我们把多线程程序设计形象地称为“一箭双雕”）和对GTK工具包的介绍，后者是GNOME版GUI程序设计的基础，并且还可能是用C语言编写X窗口界面应用程序最简单的办法。

书中还涉及到大量的Perl语言知识。有许多人认为Perl语言的春天已经到来，但也有些像我这样的人认为Perl的春天准确地说是曾经来临过，只是很久以前就过去了。当然，我个人的观点并不重要，虽说有点过时，可Perl确实是一种功能极为强大的脚本程序设计语言。所有Linux程序员（特别是那些为网络设计CGI脚本程序的人们）迟早会遇到用Perl语言编写的东西，所以这本书里出现大量的Perl程序也就不足为奇了。

本书的最后一章是你加入内核设计人员行列的一个机会。正如你将会看到的，内核程序的设计工作与为大型应用程序而进行的程序模块设计工作并没有太大的区别。戴上你的旧帽子，留出个大胡子，再喝着可乐，让我们加入到快乐的行列里来吧！

Alan Cox

参加本书翻译的人员除封面署名外还有：杨祯和、王玉敏、张玉亭、韩兰、李京山、张乔、李松、张雁东、韩东生、林红、郭明、宋建生、刘丽、郝宏志、郑志生、孙晓迪、苗苗、王大佑、董明才、许椿、赵祥生、陈广学、黄济令、姜义春、沈萍、林超、吴铭、胡平建、曹安、周友远

前　言

这是一本易于阅读使用的指南性读物，其主要内容是在Linux和其他UNIX风格的操作系统上进行程序开发。

在这本书里，我们的目标是把对读者（即那些使用UNIX的开发人员）而言非常重要的内容广泛的论题介绍给大家。我们在本书的内容组织方面确实动了不少的脑筋，不管读者现有的经验如何，我们都希望它能够最大限度地帮助你尽可能多地学习到由UNIX提供出来的东西。UNIX系统程序设计的覆盖面是非常广的，我们的目标是对广泛领域内的大量论题都进行足够深度的讨论，让读者在每一个论题方面都学习到足够的“入门”知识。

这本书是为哪些人编写的?

如果你是一个程序设计员，希望利用UNIX（和Linux）为软件开发提供的各种手段加快程序设计进度，充分利用你在程序设计方面花费的时间，并且希望自己设计出来的程序能够最大限度地利用UNIX系统本身的潜力，挑上这本书算是选对了。书中明确清晰的论述和一个步骤一个步骤的试探及检验将帮助你迅速掌握所有关键性的技巧。

我们假设读者对UNIX操作系统所涉及的基本概念都有一定的了解，如果你还有一些非UNIX环境（比如MS-DOS或者微软公司的Windows环境）中的C或C++程序设计经验就更理想了。书中出现直接比较的地方都会在内容里标出来。

> 如果你刚开始学习UNIX，请注意：这可不是一本介绍Linux安装和配置过程的书。如果你只是想多学习一些UNIX系统管理方面的知识以及一般意义上的UNIX概念和UNIX命令，那最好还是去阅读《Instant UNIX》（UNIX初学者）一书，那本书由本书两位作者和Andrew Evans合著，也是Wrox出版社出版的（国际书号是ISBN 1-874416-65-6）。

本书的目标有两个，一是作为指南性教材，向读者介绍大多数UNIX系统上都有的各种程序工具和函数/函数库系列，二是作为一本方便的速查资料。这本书的特点是方法实用，论述清晰，示例众多。

本书都讨论了哪些内容?

这本书希望能够达到以下几个目的：

- 介绍脱胎于IEEE POSIX和X/Open（SPEC 1170）等技术规范的UNIX98标准中所规定的UNIX操作系统的标准化C语言库及其他命令工具。
- 介绍如何制作编写大多数高级开发工具。
- 对流行的快速开发语言如shell、Tcl和Perl等进行简明的介绍。

- 介绍如何为X窗口系统编写图形化的用户操作界面。我们将对普通X系统上使用的Tk工具包和GNOME窗口系统中使用的GTK+工具包进行介绍。
- 在读者打下坚实的基础之后，我们将进一步介绍现实世界中的应用程序，这部分内容可能是广大读者最感兴趣的东西。

在讨论以上这些问题的时候，我们的做法将是先介绍基本的理论知识，然后用一个适当的例子和清晰的程序注释对具体问题加以阐述。在第一遍学习的时候你就能够迅速地学习到有关的知识。如有必要，你还可以回过头来仔细琢磨，进一步体会其中各种基本成分的奥妙。

书中每一个小程序示例的目的主要是演示一组函数的基本用法，或者是为了演示某些新理论概念的具体实现方法，但它们组合在一起将构成一个大型的示范性项目成果：即一个记录音乐CD唱盘详细资料的简单的数据库应用程序。随着知识面的扩展，你可以随意对该项目成果做进一步的开发、重新实现、或者扩充。尽管如此，书中每一章的内容又是相对独立的，所以读者可以根据自己的情况做跳跃性学习，但我们认为各章内容还有着一定的连贯性，前面的内容为我们将要讨论的东西提供了一些有用的示范和预备知识。在各章内容的编排方面我们尽量做到循序渐进，后面的论题肯定要比其前面的内容更高深一些。

在本书里，第一次出现的应用程序示例在shell程序设计这一章节的末尾，它向我们展示了一个比较大型的shell脚本程序是如何组织在一起的，我们将学习到如何通过shell对用户输入进行处理、如何通过shell构造菜单，以及shell是如何保存和检索数据的。

在简单回顾了编译程序、库函数链接以及在线手册等几个方面之后，我们先对shell程序设计做了一个简单的介绍。随后我们将一直使用C语言进行程序设计，讨论内容将包括利用文件进行工作、从UNIX环境里获取信息、对终端的输入输出进行处理以及curses函数库（它使交互式的输入和输出更具可扩展性）等。最终我们得以尝试使用C语言再一次实现CD唱盘管理软件。这个软件的设计思路没有什么变化，但新编写的代码使用了curses函数库向用户提供了一个基于屏幕的操作界面。

接下来，我们开始讨论数据管理方面的问题。为了学习dbm数据库函数库的使用方法，我们将再次实现这个应用软件，而这次实现中所采用的设计思路将贯穿本书其余的章节。这个应用软件的用户操作界面被单独保存在一个文件里，而CD唱盘的数据库则是另外一个程序。经过我们的改进，如今的数据库资料已经是关系化的了。

这些新应用软件的规模已经比较大了，所以我们接下来要对付一些软件维护方面的问题，比如代码调试、源代码控制、软件的发行传播以及制作文件等。

第10章是这本书的一个分水岭。在这一章里，我们将对程序在运行期间的行为特点以及如何让它们按照我们的意愿去执行等问题进行学习。程序进程可以分割和变形，同时它们也开始在彼此之间发送信号。我们还讨论了POSIX线程，并将看到如何在一个进程里创建多个执行线程。

在掌握了多进程概念之后，CD唱盘管理软件就可以被实现为一个客户端和一个服务器端，两者通过几种可靠的手段实现了相互通信。客户/服务器格局的应用软件被实现了两次，但数据库本身和用户操作界面还是相同的，只是在添加间接通信层的时候分别使用了两种办法：管道和System V的IPC。为了使这一部分内容更加完整，我们接下来又研究了套接字，并使用一个TCP/IP网络完成了进程间的通信。

再往下开始了Tcl/Tk方面的学习，我们在那里介绍了Tcl shell并通过Tk创建了各种各样的X用户操作界面。然后我们介绍了如何通过GIMP工具包（GTK+）为GNOME窗口环境开发应用软件，我们在这部分内容里开发了一个桌面时钟作为示例。

接下来，我们开始研究因特网，首先是HTML语言，然后是通用网关接口（Common Gateway Interface）；后者允许我们间断性地访问某个应用程序。这一次，我们通过远程Web浏览器来访问CD唱盘软件的用户操作界面，然后通过Web浏览器去访问由CGI程序生成的网络页面。客户和CGI程序之间隔着Web服务器软件Apach。

在这本书的最后部分，我们介绍了编写设备驱动程序的方法——这是通向理解Linux内核本身道路上的重要一步。

由于作者水平有限，在这本书里难免有所疏漏，但我们希望在我们讨论到的内容方面能够给读者一个比较清晰的概念。

学习这本书需要准备哪些东西？

在这本书里，我们向大家介绍的主要是UNIX操作系统下的程序设计。为了帮助大家更好地理解各章的内容，我们真心希望大家在阅读本书的时候要使用我们给出的程序设计示例对照学习。这些程序设计示例体现了良好的程序设计经验，指导读者在今后编写出自己的程序。

Linux发行版本是在UNIX环境下进行程序设计的理想办法，它带有一个比较完整的开发环境，其中包括GNU的C/C++编译器、开发辅助工具以及其他有用的小东西。它是自由传播的、符合POSIX规范的、健壮的、不断开发完善的，而且它的功能也十分强大。

Linux可以用在许多不同的系统上。Linux的适应性可以用这样一句话来形容：在人们的努力下，只要是其中有一个处理器芯片的东西，Linux就能以这样或那样的形式来运行！目前能够运行Linux的系统包括基于Alpha、SPARC、ARM、PowerPC和68000系列CPU的计算机，当然也包括了采用英特尔公司x86和Pentium X系列芯片（及其兼容芯片）的现代个人电脑。

我们在编写这本书的时候使用的是基于英特尔公司CPU芯片的系统，但我们讨论的内容很少是只适用于英特尔芯片的。虽然在一台只有2MB内存且没有硬盘的386机器上运行Linux也是可能的（确实可以运行），但要想成功地运行Linux并试验书中的程序示例，我们推荐至少要有以下的配置：

- 奔腾级处理器。
- 32MB内存。
- 600MB可用硬盘空间，并且最好是在同一个分区里。
- 如果想运行X窗口系统，还需要有一块它支持的显卡。

X窗口系统所支持的显卡的资料可以在http://wwwxfree86.org/网址处查到。

运行本书大部分章节中的代码所需要的硬件配置其实是相当小的，只有涉及X窗口系统的有关章节才需要比较强大的配置（或更多的耐心）！

我们使用了两台不同配置的Linux系统来编写本书的内容和开发那些程序示例，因此我们确信只要Linux能够运行在你的机器上，你就可以顺利地完成本书的学习。另外，在本书的技术校对期间，我们在另外一个Linux版本上完成了全部代码的测试工作。

在软件要求方面需要提醒读者注意的是，程序示例中有一小部分需要比较新的Linux内核版本，即2.2或更高版本才能顺利运行。Java开发工具包（Java Development Kit）要求使用最新版本的GCC和C语言库（glibc 2或更高）。至于其他工具软件，最好的办法是设法获得它们最新的版本。比如说，Tcl和Tk章节中的程序示例分别需要在7.5和8.0版本以上才能顺利运行。我们会在必要时提醒大家注意最低配置要求，如果读者在代码运行方面遇到了麻烦，使用比较新一些的工具可能会有所帮助。好在所有这些软件工具都可以很方便地下载到，而且，我们在附录C里已经准备了一份因特网资源指南以帮助读者找到它们。如果读者使用的是最近推出的Linux发行版本，就应该不会遇到什么问题。

因为Linux、GNU工具包以及其他软件都是在GPL版权规定下发行的，所以它们是有一些特点的，其中之一就是所谓的“自由性”。它们的源代码永远是公开的，任何人都不能抹杀这种自由性。因此，它们也都是一些“源代码开放”软件，“源代码开放软件”这个术语的含义要比“自由软件”的含义宽一些，因为有些专利软件在某种规定的条件下也会提供其源代码。在GNU/Linux世界里，你永远能够得到技术支持——你可以自己动手修改源代码，也可以雇佣其他人。目前，为Linux及其相关软件工具提供收费技术支持的公司正在不断增加。

实例代码

我们已经尽了最大的努力向读者提供能够最好地说明书中有关概念的示范性程序和代码段。

特别需要指出的是，我们并没有对我们调用过的每一个函数的返回值是否就是我们所预期的进行检查。在真正的应用软件成品代码里我们当然会对函数的返回值进行检查，而读者也必须这样做，因为只有这样才能为程序的错误处理提供一个有效的手段。我们在书中的第3章里介绍了一些捕获和处理程序错误的办法。

本书所有程序代码都可以从下面这个网址下载到：

http://www.wrox.com

书中的所有代码都遵守GNU的公共许可证（GNU Public License）。我们建议读者下载一份全部程序代码的拷贝，这样可以节省你不少的打字时间。

体例

为了让读者从这本书里学习到更多的知识和跟上书中论述的内容，我们在这本书里使用以下一些编写体例。

> 书中的文字框里是一些重要的，不应该被忘记的内容。它们与其周边的内容息息相关，就像电影《谍中谍》中特工们准备窃取的关键资料。

我们在程序示例开始之前会加上一个“动手试试”标记，目的是为了帮助大家把示范性的代码部分与我们准备介绍学习的代码区分开，并且让它们在书中的内容里更加明显，让读者看清楚应用程序的编写进度。在重要的场合，我们还会在代码部分的后面加上一个“操作注释”来解释与前面理论有关的代码中容易被混淆的关键之处。我们发现这两个标记能够把比较难于理解的代码清单分解为相对简单的部分。

告诉我们你的想法

我们已经努力使这本书正确而详尽，因此我们希望读者能够告诉我们本书就是你最想要和最需要的，让我们能够得到一丝满足；也希望读者对我们这本书的编排方面多提宝贵意见。

我们欢迎对我们这份努力的反馈意见，不管是批评还是赞扬，我们都将在今后的编辑工作中采纳。如果你有话要说，请按以下地址和我们联系：

Feedback@wrox.com

或者

http://www.wrox.com

现在就把这两个地址加到你的书签里去吧！

本书的英文书名为：Beginning Linux Programming, 2nd Edition

英文书书号为：ISBN 1-861002-97-1

目　录

第1章 入门知识

在这开篇的第一章里，我们来研究一下Linux是什么，它与它的领路者UNIX到底有什么联系。我们将带大家去看看UNIX开发系统都为我们准备了哪些工具，并且还将编写并运行我们的第一个程序。在本章里，我们将学习以下几方面知识：

- UNIX、Linux和GNU。
- UNIX程序和程序设计语言。
- 静态和共享库。
- UNIX程序设计的特点和原则。

1.1 什么是UNIX操作系统

UNIX操作系统最早是在贝尔试验室（Bell Laboratories）开发出来的，当时的贝尔试验室是电信巨人美国电报电话公司（AT&T）旗下的一员。它是在1970年为数字设备公司（Digital Equipment）的PDP系列计算机设计的，但随后一发不可收，逐渐发展成为一个非常流行的多用户、多任务操作系统。UNIX操作系统可以运行在大量不同种类的硬件平台上，其适用范围从PC工作站一直到采用多处理器芯片的服务器和超级计算机等。

严格来说，UNIX只是一个X/Open组织管理下的商标，它指的是一个符合X/Open技术规范XPG4.2的计算机操作系统。这个规范也叫做SPEC 1170，它定义了UNIX操作系统里所有函数的名称、接口和功能。X/Open技术规范是一系列早期的技术规范（即P1003，也叫做POSIX技术规范）的一个大的超集，它们是由IEEE（电工电子工程师学会，Institute of Electrical and Electronic Engineerings）组织开发的。

目前仍有许多种UNIX类型的操作系统在使用中，有些是商业化的，比如美国Sun公司适用于SPARC芯片和Intel芯片的Solaris操作系统；还有一些是自由免费的，比如FreeBSD和Linux操作系统等。完全符合X/Open技术规范要求的系统目前还不是很多，而只有完全符合该技术规范的要求才能挂上“UNIX98”标签。在过去，虽然POSIX在这一方面也起了相当大的作用，可不同UNIX系统之间的兼容性问题一直是个老大难。在推出X/Open技术规范之后，UNIX和其他UNIX类型的系统就有希望融为一体。

1.2 什么是Linux操作系统

正如你已经知道的那样，Linux是一个自由传播的UNIX类型内核的具体实现，而内核指的是操作系统底层的核心程序代码。因为Linux本身脱胎于UNIX系统，所以Linux程序与UNIX程序是十分相似的。事实上，几乎所有为UNIX编写的程序都可以在Linux下通过编译和运行。此外，许多为商业化UNIX操作系统版本而销售的商业化应用软件其二进制形式也几乎可以在不加

任何修改的情况下直接在Linux系统上运行。Linux是由Linus Tovalds在赫尔辛基大学开发出来的，他通过因特网得到来自UNIX程序员的大力协助。它最初只不过是一个脱胎于Andy Tanenhaum的Minix操作系统（一个小的UNIX系统）的个人爱好，但逐步发展成为一个独立完整的UNIX系统。Linux内核没有使用任何来自AT&T或其他专利源代码的代码。

1.3 发行版本

我们在前面已经提到过，Linux事实上只是一个操作系统内核。你可以设法获得该内核的源代码，编译并安装它们；然后再通过获取和编译其他自由传播的软件程序而构成一个完整的UNIX风格的系统。尽管这类安装的完整组成通常并不仅限于这个操作系统内核，但它们一般都被称为Linux系统。其中的大多数软件工具和命令程序都来源于自由软件基金会（Free Software Foundation）的GNU项目（GNU Project）。

我想你可能会赞同这个观点，即从源代码开始建立一个Linux系统是一个不小的工程。幸运的是已经有许多人开始构造“发行版本”，通常是在CD-ROM盘片上，它不仅包含了我们刚才提到的操作系统内核，还包含着许多程序设计工具和命令程序。其中通常会实现有一个X窗口系统，而X窗口系统是UNIX系统上最常见的图形化环境。发行版本一般都有一个安装程序和一些额外的文档（通常都放在那张CD盘片上），它们可以帮助你安装好你自己的Linux系统。比较有名的发行版本有Slackware、SuSE、Debian、Red Hat和Turbo Linux等，但还有许许多多不那么有名的。

1.4 GNU项目和自由软件基金会

Linux之所以能够存在并发展到今天是无数人团结合作的结果。操作系统内核本身只是一个能够实际应用的开发系统中一个很小的组成部分。商业化的UNIX系统传统上都包含着许多应用程序，是它们提供了系统服务和软件工具。对Linux系统来说，这些传统性的程序都是由不同程序员编写出来并自由传播的。

Linux社团（以及一些其他的软件开发组织）支持自由软件的概念，即软件本身不应该受到什么限制，它们应该是自由的，它们通常都遵守GNU一般公众许可证（GNU General Public License）。虽然在获取这份软件的过程中可能会有一定的费用，但除那以外可以自由使用，并且它们通常都是以源代码的形式传播的。

自由软件基金会（Free Software Foundation）是由Richard Stallman创建的，他是UNIX和其他系统上最出名的文字编辑器GNU Emacs软件的编写者。Stallman是自由软件概念的一个先锋，他开创了GNU项目（GNU Project），这个项目的宗旨是创建一个与UNIX兼容的操作系统和开发环境。最终的结果可能是它在最底层与UNIX有极大的区别，但它将支持UNIX操作系统中的应用程序。GNU这个名字代表的意思是GNU's Not Unix（GNU不是UNIX）。

GNU项目已经向软件行业提供了大量的应用程序，它们与那些我们在UNIX系统里看到的东西惟妙惟肖。这些程序被称为GNU软件，它们都是在GNU公众许可证（GNU Public License，简称GPL）条款下发行传播的。本书的附录B就是一份该许可证的拷贝。这份许可证体现了版权公开（“copyleft”，英文“copyright”（版权所有）的反话）的概念。“版权公开”的目的是阻止

其他人在自由软件的使用方面强加上某些人为的限制。

在GPL许可证条款下发行传播的GNU项目软件主要有：

- GCC　一个C语言编译器。
- G++　一个C++编译器。
- GDB　一个源代码级的调试器（debugger）。
- GNU make　UNIX系统中make命令的一个自由版本。
- Bison　一个与UNIX系统中的yacc兼容的词法分析器。
- Bash　一个命令解释器（shell）。
- GNU Emacs　一个文本编辑器和环境。

还有许多其他的软件包也已经在接受自由软件准则和GPL条款的前提下被人们开发出来了。其中包括图形图像处理工具、电子表格、源代码控制工具、编译器和解释器、因特网工具程序等，同时还有了一个完整的面向对象的环境：GNOME。我们将在后面的某个章节遇到GNOME。

在http://www.gnu.org网址处你可以找到更多与自由软件有关的内容。

1.5 为Linux系统设计程序

许多人认为UNIX程序设计就意味着使用C语言。确实，UNIX操作系统最初就是用C语言编写出来的，大部分的UNIX应用程序也是用C语言编写的；但C语言并不是UNIX程序员惟一的选择。在本书的学习过程中，我们将向读者介绍一些其他的手段，它们在某些情况下可以为程序设计难题提供一个更简洁的解决方案。

> 事实上，最初的第一个UNIX版本是于1969年用PDP 7机器的汇编语言编写出来的，差不多也在那个时候，Dennis Ritchie发明了C语言；到了1973年，他和Ken Thompson主要使用C语言重新编写了整个的UNIX操作系统内核。这在系统软件还是由汇编语言当家的那个时代是相当了不起的壮举。

对UNIX系统来说有各种各样的程序设计语言可供选用，其中有许多是免费的，它们的获取途径可以是CD-ROM光盘，也可以从因特网上的FTP档案站点下载。书后的附录C里就有一个包含着各种有用资源的清单。下面这个表格只是UNIX程序员可以使用的部分程序设计语言。

Ada	C	C++
Eiffel	Forth	Fortran
Icon	Java	JavaScript
Lisp	Modula 2	Modula 3
Oberon	Objective C	Pascal
Perl	PostScript	Prolog
Python	Scheme	Smalltalk
SQL	Tcl/Tk	UNIX Bourne Shell (sh)

在这本书里，我们将把注意力集中在很少的几种程序设计语言上。我们将在下一章里看到如何通过UNIX系统的shell来开发小到中规模的应用程序，其重点是从C语言程序员的角度去深入探讨UNIX程序接口的设计问题。在接下来的各个章节里，我们将对使用C语言进行底层程序设计的其他方面进行研究，特别是在因特网（HTML、Perl、Java等）和X窗口系统（Tcl/Tk、GNOME等）环境中进行的程序设计。

1.6 UNIX程序

UNIX下的应用程序主要由两种特殊类型的文件来代表：可执行文件和脚本程序。可执行文件是能够被计算机直接执行的程序，相当于DOS中的.exe文件。脚本程序则是一组指令，这些指令将由另外一个程序（比如说是解释器）来执行，它相当于DOS中的.bat文件或者解释型BASIC程序。

UNIX不要求可执行文件或脚本程序具备某种特定的文件名或者某种特定的扩展名。某个文件是否是一个可以被执行的程序将由第2章里介绍的文件的系统属性来决定。我们可以把UNIX系统中的脚本程序替换为经过编译的程序（反过来也行）而不影响其他程序，调用它们的人也不会感到有什么差异。事实上，从用户的角度看，这两者其实是没有什么区别的。

当你登录进入到某个UNIX系统中去的时候，和你打交道的是一个shell（命令解释器）程序（通常就是sh），由它负责为你调用执行其他的程序。它的工作原理和DOS中的COMMAND.COM是一样的。它在一组给定的子目录集合里按照你给出的文件名查找到与之同名的那个文件并把它当作你打算执行的程序。将搜索的那些子目录都被保存在一个名为PATH的shell变量里，和DOS下的情况差不多。搜索路径（你可以对这个路径进行增减）是由你的系统管理员预先配置好了的，它通常包含着用来保存系统程序的几个标准场所，其中包括：

- /bin　　二进制文件子目录，一般用来保存引导系统用的程序。
- /usr/bin　　用户级二进制文件子目录，用来保存可供一般用户使用的标准程序。
- /usr/local/bin　　本地二进制文件子目录，一般用来保存某种安装情况下的程序。

系统管理员（比如root用户）的登录可以使用一个特殊的PATH变量，它包括几个存放有系统管理员专用程序的子目录，比如/sbin和/usr/sbin等。

可选装的操作系统组件和第三方应用软件可以安装在/opt子目录中的某个下级子目录里，而安装程序可能会通过用户安装脚本在PATH变量里添加一些东西。

尽量不要删除PATH变量里出现的子目录，除非你确实知道自己这样做会产生什么样的后果。

请注意：UNIX使用冒号“:”来分隔PATH变量里的各个数据项，而DOS使用的则是分号“;”。（UNIX选用“:”在先，因此应该问为什么MS-DOS另起炉灶而不应该问为什么UNIX与之不同！）下面就是一个PATH变量的示例：

```
/usr/local/bin:/bin:/usr/bin:.:/home/neil/bin/usr/X11R6/bin
```

上面的PATH变量里包括保存程序的标准场所：当前子目录（.）、某个用户的登录子目录和X窗口系统的程序子目录。

C语言编译器

我们以下面这个程序来开始对UNIX系统的C语言开发，这是我们编写、编译和运行的第一个UNIX程序。它大概是所有程序里最有名的了——我们来看看“Hello World”程序。

> 动手试试：我们第一个UNIX的C语言程序

1) 下面是文件hello.c中的源代码。

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    exit(0);
}
```

2) 输入这个程序需要使用一个编辑器。在一个典型的Linux系统上有许许多多的编辑器可供选用。有不少用户喜欢vi编辑器。两位作者喜欢用的是emacs，这是一个功能强大的编辑器，我们建议读者花点儿时间学习一些它的使用方法。学习emacs的办法很简单，启动它以后按下“Ctrl-H”组合键，然后按下表示教学的字母“t”（英文tutorial的第一个字母）。emacs的使用手册完全是在线式的。按下“Ctrl-H”组合键后再按下“i”可以查看它的子命令信息。Emacs的某些版本可能会有菜单，你可以通过那些菜单来查阅手册和教程。

3) 在POSIX兼容的系统上，C语言编译器的名字是c89；而C语言编译器的传统名字是简单的cc。在那么多年的时间里，不同的供应商销售不同的UNIX系统，里面的C语言编译器也具有不同的功能和编译选项，但它们一般都叫做cc。

等到准备起草POSIX标准的时候，已经不可能定义出一个全体供应商都能够与之兼容的标准化cc命令了。因此，该委员会决定为C语言编译器创建一个新的标准化命令，这就是c89。只要使用的是这个命令，那么不管是在什么样的机器上，它的编译选项就永远是一致的。

在Linux系统上，你会发现任何或全部的c89、cc和gcc命令都指向系统的C语言编译器，它通常是GNU的C语言编译器。在UNIX系统上，C语言编译器几乎总是被称为cc。

在这本书里，我们将使用GNU的C语言编译器，因为它是和Linux发行版本一起提供的，同时还因为它支持C语言的ANSI标准语法。如果读者使用的UNIX系统不带GNU的C语言编译器，我们建议你设法获取这个编译器并安装上它。你可以沿着http://www.gnu.org网址上的链接找到它。在书中我们使用命令cc的地方，把它替换为你系统里相应的C语言编译器命令就可以了。

4) 对我们的程序进行编译、链接和运行。如下所示：

```
$ cc -o hello hello.c
$ ./hello
Hello World
$
```

操作注释：

我们调用系统的C语言编译器把我们的C语言源代码转换为一个名为hello的可执行文件。我

们运行这个文件，它显示出欢迎辞。这只是一个最简单的例子，但如果你在自己的系统上可以做到这一点，那就应该能够编译和运行书中其余的程序示例。如果无法完成上述操作，请检查你的系统上是否安装了C语言编译器。Red Hat公司出品的Linux发行版本有一个叫做“C Development”的安装选项，你应该选上这一项。

因为这是我们运行的第一个程序，所以有些问题最好现在就指出来。hello程序很可能就保存在你的登录子目录里。如果PATH变量没有包含指向你登录子目录的路径数据项，shell就没有办法找到hello。而且，如果PATH变量里的某个子目录中还有另外一个也叫做hello的程序，就会执行那个程序。如果在PATH变量里一个这样的子目录出现在你登录子目录的前面，也会出现这样的情况。

为了避免出现这种潜在的意外，你可以在程序名的前面加上一个“./”前缀（也就是./hello），它特别指示shell去执行在当前子目录里找到的给定名称的程序。

“-o name”选项的作用是告诉编译器把可执行代码放到哪个文件里去，如果你忘了使用这个选项，编译器会把程序放到一个名为a.out（意思是汇编器输出）的文件里去。如果你认为自己刚刚编译了一个程序但又找不到它，别忘了看看有没有一个a.out文件！在UNIX的早期，想在系统上玩游戏的人们经常把游戏做为a.out来运行，就是为了避免被系统管理员捉到；而许多大型的UNIX系统安装会在每天夜里定期删除所有名为a.out的文件。

1.7 获得帮助

从系统程序设计接口和标准软件的角度看，所有的UNIX系统都准备有充足的文档。这是因为早期的UNIX系统是鼓励程序员为他们编写的程序准备一份使用手册页的。这些使用手册页总是能够在线查阅到的，其中一些会以印刷材料的形式提供给大家。

man命令提供了访问在线使用手册页的办法。使用手册页在质量和细节上千差万别。有些只是简单地让读者去参考另外一份更详尽的文档，另一些则会给出一份该软件工具所支持的选项和命令的完整清单。不管是那种情况，使用手册页总是一个不错的出发点。

GNU套装软件和其他一些自由软件使用一种叫做“info”的在线文档系统。你可以使用一个特殊的程序info或者通过emacs编辑器的info命令来在线查看全部文档。info系统的好处是你可以通过链接和交叉引用来浏览文档并直接跳到相关的章节去。对文档的作者来说，info系统的好处是它的文件可以从同一份源文件开始自动生成为排好版的打印文稿。

动手试试：使用手册页和info系统

1) 我们来看看GNU的C语言编译器的文档。先看看使用手册页。

```
$ man gcc

GCC(1)                          GNU Tools                          GCC(1)

NAME
       gcc, g++ - GNU project C and C++ Compiler (egcs-1.1.2)

SYNOPSIS
       gcc [ option | filename ]...
       g++ [ option | filename ]...
```

```
WARNING
       The  information  in  this man page is an extract from the
       full documentation of the GNU C compiler, and  is  limited
       to the meaning of the options.

       This  man  page  is not kept up to date except when volun-
       teers want to maintain it.  If  you  find  a  discrepancy
       between  the  man  page and the software, please check the
       Info file, which is the authoritative documentation.

       If we find that the things in this man page that  are  out
       of date cause significant confusion or complaints, we will
       stop distributing the man page.  The alternative, updating
       the  man  page when we update the Info file, is impossible
       because the rest of the work of maintaining GNU CC  leaves
       us no time for that.  The GNU project regards man pages as
       obsolete and should not let them take time away from other
       things.
       For  complete and current documentation, refer to the Info
       file 'gcc' or the manual Using and  Porting  GNU  CC  (for
       version  2.0).  Both are made from the Texinfo source file
       gcc.texinfo.
...
```

如果愿意，我们可以看到编译器针对各种可供选用的目标处理器所支持的编译选项。这些目标处理器的使用手册页相当长，但仍然只是GNU全部C（和C++）语言文档的一小部分。

在阅读使用手册页的时候，可以使用空格键阅读下一页，回车键阅读下一行，“q”键退出。

2) 为了获得更多关于GNU的C语言编译器的资料，我们可以再试试info系统。

```
$ info gcc

File: gcc.info,  Node: Top,  Next: Copying,  Up: (DIR)

Introduction
************

   This manual documents how to run, install and port the GNU compiler,
as well as its new features and incompatibilities, and how to report
bugs.  It corresponds to EGCS version 1.1.2.

* Menu:

* G++ and GCC::       You can compile C or C++ programs.
* Invoking GCC::      Command options supported by 'gcc'.
* Installation::      How to configure, compile and install GNU CC.
* C Extensions::      GNU extensions to the C language family.
* C++ Extensions::    GNU extensions to the C++ language.
* Trouble::           If you have trouble installing GNU CC.
* Bugs::              How, why and where to report bugs.
* Service::           How to find suppliers of support for GNU CC.
* VMS::               Using GNU CC on VMS.

* Portability::       Goals of GNU CC's portability features.
* Interface::         Function-call interface of GNU CC output.
* Passes::            Order of passes, what they do, and what each file is for.
* RTL::               The intermediate representation that most passes work on.
* Machine Desc::      How to write machine description instruction patterns.
* Target Macros::     How to write the machine description C macros.
* Config::            Writing the 'xm-MACHINE.h' file.
-zz-Info: (gcc.info.gz)Top, 36 lines --Top-- Subfile: gcc.info-1.gz----------
Welcome to Info version 3.12f.  Type "C-h" for help, "m" for menu item.
```

我们将看到一个很长的选项菜单，通过选择其中的选项，我们就可以在该文档完整的文本内容里面四处移动。菜单项和树状结构的页面布局使我们可以在一个非常大的文档里漫游。如果是在书面上，GNU的C语言文档会有几百页之多。

当然，info系统本身还有一个info格式的帮助页面。如果你按下“Ctrl-H”组合键，就会看到一些帮助信息，其中包括一个关于info系统使用方法的教程。许多种Linux发行版本里都带有info程序，它还可以被安装到其他的UNIX系统上去。

1.8 程序开发系统的预备知识

对一个UNIX开发人员来说，多少知道一点软件工具和开发资源都放在什么地方是很重要的。我们简单地介绍几个重要的子目录和文件。这部分内容主要是针对Linux的，但同样的原理也适用于其他类型的UNIX系统。

1.8.1 程序

程序通常都被保存在专门为此保留的子目录里。系统为正常使用情况准备的程序，包括程序开发，都可以在/usr/bin子目录里找到。系统管理员为某个特定的主机系统或本地网络添加的程序可以在/usr/local/bin子目录里找到。

系统管理员一般都喜欢使用/usr/local子目录，因为它可以把供应商提供的文件和后来添加的东西和系统本身提供的程序隔离开来。/usr子目录的这种布局方法在需要对操作系统进行升级的时候非常有用，因为只有/usr/local子目录里的东西需要保留。我们建议读者编译自己的程序时让它们遵照/usr/local子目录的树状结构来运行和访问必要的文件。

某些随后安装的软件或者程序设计系统有它们自己的子目录结构，其执行程序也保存在特定的子目录里。这类情况里面最明显的例子就是X窗口系统，它通常被安装在一个名为/usr/X11的子目录里。其他的安装位置还有/usr/X11R6子目录，这是X窗口系统第6版安装的场所。XFree论坛组织发行的用于英特尔处理器芯片的各种XFree86窗口系统变体也安装在这里。大部分Linux发行版本也使用这个场所安装X窗口系统；随Solaris操作系统提供的Sun开放窗口系统（Sun Open Windows system）一般安装在/usr/openwin子目录里。

GNU的C语言编译器软件的主驱动程序gcc（我们在前面的程序设计示例中使用的就是它）通常被安装在/usr/bin或者/usr/local/bin子目录里，但通过它运行的各种编译器支持程序一般都被保存在另一个位置。这个位置是在你编译自己的编译器本身时指定的，随主机计算机类型的不同而不同。对Linux系统来说，这个位置通常是/usr/lib/gcc-lib/目录下以其版本号确定的某个下级子目录。GNU的C/C++编译器的各道编译程序以及GNU专用的头文件都保存在这里。

1.8.2 头文件

在使用C语言和其他语言进行程序设计的时候，我们需要头文件来提供对常数的定义和对系统及库函数调用的声明。对C语言来说，这些头文件几乎永远被保存在/usr/include及其下级子目录里。那些依赖于你所运行的UNIX或Linux操作系统特定版本的头文件一般可以在/usr/include/sys或/usr/include/linux子目录里找到。

其他的程序设计软件系统也可以有一些预先定义好的声明文件，它们的保存位置可以被相应的编译器自动查到。比如X窗口系统的/usr/include/X11子目录和GNU的C++编译器的/usr/include/g++-2子目录等。

在调用C语言编译器的时候可以通过给出“-I”标志来引用保存在下级子目录或者非标准位置的头文件，请看下面的命令：

```
$ gcc -I /usr/openwin/include fred.c
```

它会使编译器在/usr/openwin/include子目录和标准位置两个地方去查找fred.c程序里包括的头文件。具体情况请参考你自己的C语言编译器使用手册。

用grep命令来查找含有某些特定定义与函数声明的头文件是很方便的。假设你想知道用来返回程序退出状态的定义的名字，办法很简单：先进入/usr/include子目录，然后在grep命令里给出该名字的几个字母，如下所示：

```
$ grep EXIT_ *.h
...
stdlib.h:#define         EXIT_FAILURE    1       /* Failing exit status.  */
stdlib.h:#define         EXIT_SUCCESS    0       /* Successful exit status.  */
...
$
```

grep命令会在该子目录里所有名字以.h结尾的文件里查找字符串“EXIT_”。在上面的例子里，它（从其他文件中间）在文件stdlib.h里找到了我们需要的定义。

1.8.3 库文件

库文件是一些预先编译好的函数的集合，那些函数都是按照可再使用的原则编写的。它们通常由一组互相关联的用来完成某项常见工作的函数构成。比如用来处理屏幕显示情况的函数（curses库）和数据库访问例程（dbm库）等。我们将在后续章节遇到这些函数库文件。

标准的系统库文件一般被保存在/lib或者/usr/lib子目录里。人们必须告诉C语言编译器（更确切地说是链接程序）去查找哪些库文件；在默认的情况下，它只会查找C语言的标准库文件。这是从计算机速度还很慢、CPU周期还很昂贵的年代遗留下来的问题：在当时，把一个库文件放到标准化子目录里然后寄希望于编译器自己找到它是不实际的；库文件必须遵守一定的命名规则，还必须在命令行上明确地给出来。

库文件的名字永远以“lib”这几个字母打头，随后是说明函数库情况的部分（比如用“c”表示这是一个C语言库；而“m”表示这是一个数学运算库等）。文件名的最后部分以一个句点（.）开始，然后给出这个库文件的类型，如下所示：

.a　　传统的静态型函数库。

.so和.sa　　共享型函数库（见下面的解释）。

函数库一般分为静态和共享两种格式，用“ls /usr/lib”命令查一下就能看到。在你通知编译器查找某个库文件的时候，既可以给出其完整的路径名，也可以使用“-l”标志。如下所示：

```
$ cc -o fred fred.c  /usr/lib/libm.a
```

这条命令让编译器对fred.c文件进行编译，编译得到的程序保存到fred文件；在处理函数定义和引用时除了在C语言的标准库里进行查找外还要到数学运算库里进行查找。下面的命令也能实现同样的效果：

```
$ cc -o fred fred.c -lm
```

“-lm”（在字母“l”和“m”之间没有空格）是一种简略写法（简略写法在UNIX环境里是

很有用的），它代表的是标准库目录（本例中是/usr/lib）中的名为libm.a的函数库。“-lm”记号的额外好处是编译器会自动选用共享库（如果存在的话）。

虽然库文件在大多数情况下与头文件很相似，都是被保存在某个标准的位置，但我们仍然可以通过“-L”（大写字母）标志给编译器增加搜索子目录。如下所示：

```
$ cc -o x11fred -L/usr/openwin/lib x11fred.c -lX11
```

这条命令在编译和链接程序x11fred时将使用在子目录/usr/openwin/lib中找到的libX11函数库版本。

1.8.4 静态库

函数库最简单的形式就是一组处于可以“拿来就用”状态下的二进制目标代码文件。当有程序需要用到函数库中的某个函数时，就会通过inclue语句引用对此函数做出声明的头文件。编译器和链接程序负责把程序代码和库函数结合在一起成为一个独立的可执行程序。如果使用的不是标准的C语言运行库而是某个扩展库，就必须用“-l”选项指定它。

静态库也叫做档案（archive），它们的文件名按惯例都以“.a”结尾。比如C语言标准库/usr/lib/libc.a和X11库/usr/X11/lib/libX11.a等。

自己建立和维护静态库的工作并不困难，用ar（“建立档案”的意思）程序就可以做到，另外要注意的是应该用“cc -c”命令对函数分别进行编译。你应该尽量把函数分别保存到不同的源代码文件里去。如果函数需要存取普通数据，你可以把它们放到同一个源代码文件里并使用在其中声明为“static”类型的变量。

动手试试：静态库

1）我们自己来建立一个小函数库，库里有两个函数；然后把其中的一个用在后面的程序示例中。两个函数的名字分别叫做fred和bill，作用是显示欢迎辞。我们为这两个函数 分别建立它们各自的源代码文件（源代码文件的名字就用fred.c和bill.c好了）。

```
#include <stdio.h>

void fred(int arg)
{
    printf("fred: you passed %d\n", arg);
}
```

```
#include <stdio.h>

void bill(char *arg)
{
    printf("bill: you passed %s\n", arg);
}
```

我们对这两个函数分别进行编译以生成能够添加到一个函数库里去的二进制目标文件。这需要在调用C编译器的时候加上“-c”选项以制止编译器试图生成最终的程序。因为我们没有定义一个名为main的函数，所以直接生成程序将是失败的。

```
$ cc -c bill.c fred.c
$ ls *.o
```

```
bill.o  fred.o
```

2) 现在来编写一个程序，让它调用函数bill。首先，为我们的函数库建立一个头文件。它的作用是在我们的函数库里对这两个函数进行声明；如果有程序要使用我们的函数库，就必须用include语句引用它。

```
/*
    This is lib.h. It declares the functions fred and bill for users
*/

void bill(char *);
void fred(int);
```

实际工作中最好把这个头文件也包括在fred.c和bill.c文件里去，这样可以帮助编译器查找错误。

3) 调用者程序（program.c）可以非常简单。它用include语句引用我们的函数库并调用其中的一个函数。

```
#include "lib.h"

int main()
{
    bill("Hello World");
    exit(0);
}
```

4) 现在来编译并测试这个程序。这一次，我们向编译器明确地给出目标代码文件，让它编译我们的文件并把它与我们刚才编译的目标模块bill.o链接起来。

```
$ cc -c program.c
$ cc -o program program.o bill.o
$ ./program
bill: you passed Hello World
$
```

5) 现在来编译并使用一个库。我们用ar程序来建立档案，并把我们的目标代码文件添加到其中。这个程序之所以被叫做ar就是因为它是用来建立档案或者说把一组独立的小文件集中到一个大文件里去的。请注意，我们还可以通过ar把任何类型的文件归为档案（和许多UNIX工具一样，它是一个很基本的软件工具）。

```
$ ar crv libfoo.a bill.o fred.o
a - bill.o
a - fred.o
```

函数库建立好了，两个目标文件也添加进去了。在某些系统，尤其是从Berkley UNIX操作系统演化而来的系统上，要想成功地使用函数库，必须先为这个函数库建立一个内容表。我们用ranlib命令来完成这一工作。如果是在Linux里使用GNU的软件开发工具，这一步就不是必要的（加上也不会有什么副作用）。

```
$ ranlib libfoo.a
```

我们的函数库现在就可以用了。我们可以在编译器命令行的文件清单里加上我们的函数库来建立我们的程序了，如下所示：

```
$ cc -o program program.o libfoo.a
$ ./program
bill: you passed Hello World
$
```

我们也可以通过“-l”选项来访问我们的函数库，但是因为它并没有被保存在某个标准的地点，所以需要使用“-L”选项告诉编译器到哪儿才能找到它，如下所示：

```
$ cc -o program program.o  -L. -lfoo
```

“-L.”选项告诉编译器在当前子目录里查找函数库。“-lfoo”选项告诉编译器使用那个名为libfoo.a的函数库（或者一个名为libfoo.so的共享库——如果有的话）。

要想查看某个目标代码文件、函数库或者可执行程序里都包含有哪些函数，我们可以使用nm命令。如果我们查看一下program和libfoo.a，就会看到函数库里包含着fred和bill两个函数，而program里只包含着bill函数。在编译程序的时候，只有程序中确实用到的函数才会被包括进去。虽然一个头文件里包含着函数库中的全体函数，但在程序里使用include语句引用它并不会把整个函数库的内容都包括到最终程序中去。

如果读者熟悉MS-DOS或微软Windows的软件开发工作，就会发现两者之间有许多相似的地方。

项　目	UNIX	DOS
目标代码模块	func.o	FUNC.OBJ
静态函数库	lib.a	LIB.LIB
程序	program	PROGRAM.EXE

1.8.5 共享库

静态库的缺点是如果我们在同一时间运行多个程序而它们又都使用着来自同一个函数库里的函数时，内存里就会有许多份同一函数的拷贝，在程序文件本身里面也有许多份同样的拷贝。这会消耗大量宝贵的内存和硬盘空间。

许多UNIX系统支持共享库，它同时克服了在这两方面的无谓消耗。对共享库和它们在不同系统上实现方法的详细讨论超出了本书的范围，所以我们把自己的注意力集中在眼前Linux环境下的实现方法上。

共享库的存放位置和静态库是一样的，但有着不同的文件后缀。在一个典型的Linux系统上，C语言标准库的共享版本是/usr/lib/libc.so.N，其中的“N”是主版本号，目前是6。

在我们编写本书的时候，许多Linux发行版本都在对它们的C/C++编译器和C函数库的版本进行升级。下面示例中给出的输出来自使用GNU lib2.1的Red Hat 6.0发行版本。如果读者使用的不是这个发行版本，那么你的输出情况可能会稍有不同。

如果一个程序使用了共享库，它的链接方式是这样的：它本身不再包含函数的代码，而只保存共享代码的调用线索，共享代码是在该程序运行的时候才加入到其中的。当编译好的程序被加载到内存中准备执行的时候，函数的调用线索被解析，程序向共享库发出调用，共享库只在必要的时候才被加载到内存。

通过这种办法，在内存里系统就可以只保留一份共享库拷贝供许多程序使用，在硬盘上也只需要保存一份拷贝就行。另外一个好处是共享库的升级不再会影响到依赖于它的那些程序。我们只需修改从文件/usr/lib/libc.so.6到实际库文件的升级版本（本书写作期间是/lib/libc-2.1.1.so）

的符号链接就可以了。

对Linux系统来说，负责加载共享库并解析客户程序中函数调用线索的程序（也就是共享库的动态加载器）是ld.so或ld-linux.so.2。查找共享库的其他地点是在/etc/ld.so.conf文件里配置的；如果这个文件进行了修改（如系统里增加了X11共享库），就需要用ldconfig命令进行处理。

如果读者想了解某个程序要求使用的是哪一个共享库，可以用工具程序ldd来查看，如下所示：

```
$ ldd program
        libc.so.6 => /lib/libc.so.6 (0x40011a000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

在上面的例子里，我们看到标准的C语言函数库（libc）是共享的（.so）。我们的程序要求使用的主版本号是6，上例中是由2.1.1版的GNU libc函数库担当的。其他UNIX系统在访问共享库的时候也会有类似的安排。详细情况请参考你自己的系统文档。

共享库在许多方面与微软Windows环境下的动态链接库很相似。.so库相当于.DLL文件，是在程序运行时加入的；而.sa库类似于.LIB文件，是加入到可执行程序文件里去的。

1.9 UNIX系统中程序设计的特点和原则

我们希望在以后的章节里倡导一种良好的UNIX程序设计风格。虽然不管在哪一种计算机平台上使用C语言进行程序设计都大同小异，但应该说UNIX开发人员在程序和系统开发方面更有独到的见解。

UNIX操作系统鼓励人们采用一种独到的程序设计风格。下面就是一些典型UNIX程序和系统所共有的一些特点。

1.9.1 简单性

许多最有用的UNIX软件工具都是非常简单的，作为其结果，程序小而易于理解。KISS（英文“Keep It Small and Simple”的字头缩写，意思是小而简单）是一种值得学习的技巧。大型而又复杂的系统肯定会包含更大更复杂的程序漏洞，它的调试工作是一种我们大家都愿意回避的“痛”。

1.9.2 重点性

让一个程序把一项工作做到最好是很好的做法。一个所谓功能齐全的程序可能既不容易使用，也不容易维护。如果程序只用于一个目的，那么当更好的算法或更好的操作界面被开发出来的时候，它就更容易得到改进。在UNIX世界里，通常会在需求出现的时候把小的工具程序组合到一起去完成一项更大的任务，而不是用一个巨大的程序预测一个用户的需求。

1.9.3 可反复使用的程序组件

把应用程序的核心部分搞成一个库。带有简单而又灵活的程序设计接口并且文档齐备的函数库能够帮助其他人开发同类的项目，或者能够把这里的技巧用在新的应用领域。就拿dbm数据

库函数库来说吧，它就是一整套许多可再使用的函数的集合而不是一个单一的数据库管理系统。

1.9.4 过滤器

许多UNIX应用程序可以被用做过滤器。这句话的意思是它们可以把自己的输入转换为另外一种形式的输出。我们在后面的内容里就会看到，UNIX提供的工具程序能够从组合其他UNIX程序开始开发出相当复杂的应用软件，其组合方法既新颖又匪夷所思。当然，这类程序组合正是由我们刚才介绍的开发方法支撑着的。

1.9.5 开放的文件格式

比较成功和流行的UNIX程序所使用的配置文件和数据文件都是普通的ASCII文本。如果读者在程序开发工作中有这样的选择，这将是一个很好的做法。它使用户能够利用标准的软件工具对配置数据进行改动和搜索，从而开发出新的工具，通过新的函数对数据文件进行处理。源代码交叉引用检查软件ctags就是一个这样的好例子，它把程序中的符号位置信息以规则表达式的形式记录下来供检索程序使用。

1.9.6 灵活适应性

你根本无法预测一个不太聪明的用户会怎样使用你的程序，因此在你的程序设计里要尽可能地增加灵活适应性。尽量避免给数据域长度或者记录条数加上人为的限制。如果你能够做到，就要编写能够响应网络访问的程序，使它既能够跨网络运行又能够在本地单机上运行。永远不要认为你自己了解用户心里想的一切。

1.10 本章总结

在这个介绍性的章节里，我们讨论了Linux及其先驱UNIX系统之间的一些共同之处，还讨论了能够为我们（UNIX开发人员）所用的各种各样的程序设计系统。

我们编写了一个简单的程序和一个函数库以展示基本的C语言工具，并把它们与MS-DOS中的参照物进行了对比。最后，我们对UNIX中的程序设计进行了简单的介绍。

第2章　shell程序设计

我们刚刚在这本书的开始介绍了使用C语言进行的UNIX程序设计，现在立刻就要掉转方向进入shell程序设计。为什么?

这么说吧，shell扮演着双重角色。它与DOS中的命令处理器Command.com有不少相似之处、但本身的功能又比它强大很多，我们完全可以把它看做是一种独立的程序设计语言。你不仅能够执行命令和调用UNIX工具程序，还可以编写新的命令和程序。它是一种解释型的语言，这使调试工作比较容易进行，因为你可以逐行地执行每一条指令，而且节省了反复编译所需要花费的时间。但这也使shell不适合用来完成对执行时间比较挑剔或者处理器负荷比较大的工作。

为什么要用它来进行程序设计呢？因为shell的程序设计既迅速又简单，而且即使是最基本的UNIX安装也会有一个shell。因此，作为一个简单的框架，你可以检查自己的想法是否可行。在与容易配置、维护和可移植性相比效率不是很重要的情况下，它非常适合用来编写一些完成简单任务的小工具。你可以通过shell对进程控制进行组织，使命令按照预定计划在前一步骤顺利完成的条件下顺序执行。

在你的UNIX帐户上可能已经有不少这样的例子了，比如软件包安装程序、自由软件基金会（Free Software Foundation，简称FSF）的autoconf、.xinitrc和startx，以及/etc/rc.d子目录里用来在开机引导时对系统进行配置的各种脚本程序等。

我们再来说点儿UNIX程序设计的特色。UNIX建立在并高度依赖于代码再使用的基础上。你编写了一个小巧而又简单的工具，其他人把它作为另外一根链条上的某个环节来构成一条命令。请看下面这个简单的例子：

```
$ ls -al | more
```

它使用ls和more命令把文件清单输出经管道分次分批地传输到显示器屏幕上。每个工具都只是一个建筑构件。今后你肯定会经常性地把许多小脚本程序组织起来构成一个大型而又复杂的组合程序。

举例来说，如果你想打印出一份由man命令给出的bash使用手册，就可以使用这样的命令："man bash | col -b | lpr"。

更进一步的是，由于UNIX操作系统本身在文件处理方面的强大功能，你不必知道工具程序是用哪种语言编写的。如果需要执行得更快一些，常见的办法是这样的：把用shell搭成的UNIX工具程序框架用C或C++重新写一遍——当然先要确定这样做是值得的。反过来说，如果它们工作的够好，就别再折腾了！

人们喜欢使用的可以与C或C++语言交替使用的其他解释型语言有Perl、Tcl/Tk和Python。

是否需要重写脚本程序取决于是否需要对它进行优化、是否需要对它进行移植、是否需要让它更容易修改，以及它是否偏离了最初的设计目的（经常发生这样的情况）。

因此，如果在你的系统管理工作中必须面对某个噩梦般的shell脚本程序、如果你想把自己最新的大的（但有一种简单的美）设想勾勒出来或者你只是想加快一些重复性的工作，那这一章就是为你写的。

我们将在本章学习利用shell进行程序设计时需要掌握的语法、程序结构和命令，我们主要通过交互（基于屏幕）形式的例子进行讲解。应该把这些例子当作学习shell特性及其作用的手段。在本章结尾部分，我们将编写一个有实际用途的脚本程序，在本书的其他后续章节里，我们将用C语言对它进行重写和扩充。

在本章里，我们将学习以下内容：

- 什么是shell。
- 基本思路。
- 精巧的语法：变量、条件判断和程序控制。
- 命令表。
- 函数。
- 命令和命令的执行。
- 即时文档（here文档）。
- 调试。

2.1 什么是shell

我们先来看看shell的作用和UNIX里各种各样的shell。

shell是一个做为用户与UNIX系统之间的操作接口的程序，它允许用户向操作系统输入需要执行的命令。从这方面看它与DOS很相似，但它向用户掩盖了内核操作的细节。因此，文件的重定向只要使用“<”和“>”就行，管道用一个“|”就能代表，子进程的输出是“$(...)”，所有这些的具体细节都早已为用户准备好了。从这方面看，它本身就算得上是一种高级UNIX程序设计语言。

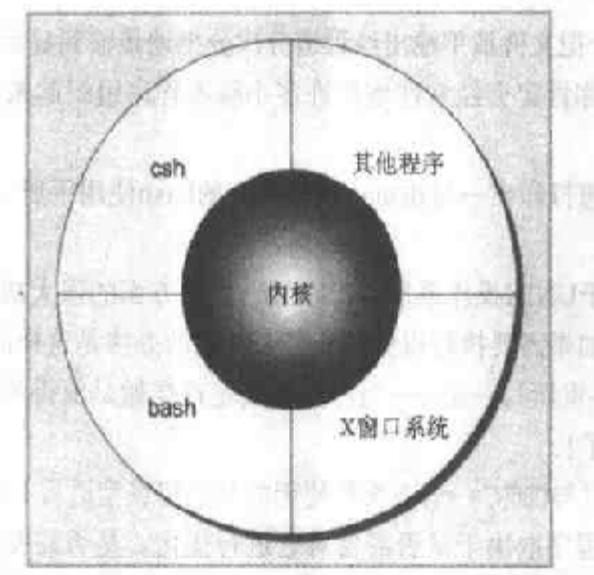


图 2-1

因为UNIX是一个模块化程度如此之高的操作系统，所以你可以从众多不同的shell里选一种来使用。它们中的大多数都是从最初的Bourne shell演变而来的（见表2-1）。

表 2-1

shell名称	掌 故
sh(Bourne)	最初的shell
csh, tcsh and zsh	Bill在Berkeley UNIX上编写的C shell。可能是bash之后最流行的shell了
ksh, pdksh	Korn shell和它的公共域兄弟，由David Korn编写
bash	Linux中的主角，来自GNU项目。bash是英文“Bourne Again Shell”的字头缩写，它的优势在于其源代码是公开的，即使你的UNIX系统上现在没有运行它，也很可能已经移植过来了
rc	比csh带有更多的C语言色彩。也来自GNU项目

除了C shell和少数变体以外，所有这些shell都很相似，并且都与X/Open 4.2和POSIX 1003.2技术规范中的规定比较吻合。POSIX 1003.2技术规范中对shell的规定是很少的，但X/Open中的扩展规定却提供了一个更加友好和强大的shell。一般说来，X/Open是一个要求比较多的技术规范，但由它而产生出来的又是一个比较友好的系统。在这里我们只列出了一些比较知名的shell变体，还存在许多其他种类的shell。

在本章里，我们使用的主要是那些与POSIX兼容的shell中常见的功能，同时我们假设默认情况下的shell被安装为/bin/sh。

> 在许多Linux系统里，/bin/sh命令通常只是一个到实际使用中的shell的链接。它在大部分Linux系统上是一个指向/bin/bash——即bash shell的链接。请在你自己的系统上用“ls -l /bin/sh”命令检查一下看是否如此。如果你还想知道自己使用的bash是什么版本的，请在bash的命令提示符处输入“/bin/bash -version”命令或“echo $BASH_VERSION”命令，它会告诉你的。

在本书后面，当我们讲到Tcl和Tk时（第14章和第15章）要分别用到tclsh和wish这两种shell。

GNU项目还提供了一组基本的shell工具，即Shellutils。与系统本身提供的shell相比，在某些安装情况下它的执行性能更好。如果你只想用shell脚本程序对文本文件进行归档的话，可以使用shar软件包。

2.2 管道和重定向

在深入shell程序设计之前，我们需要先介绍一下如何才能对UNIX程序（不仅仅是shell程序）的输入输出进行重定向。

2.2.1 对输出数据进行重定向

读者可能已经对某些类型的重定向比较熟悉了，请看：

```
$ ls -l > lsoutput.txt
```

这条命令将把ls命令的输出保存到一个名为lsoutput.txt的文件里去。

重定向可比这个简单的例子深奥多了。我们将在第3章里学习更多关于标准文件描述符的内

容，在这里我们只需知道文件描述符0代表着一个程序标准输入，文件描述符1代表着一个程序标准输出，而文件描述符2代表着一个程序标准错误输出就行了。这些标准输入输出的重定向彼此之间不会互相干扰。事实上，你还可以对其他文件描述符进行重定向，但对标准输入输出0、1和2等以外的东西进行重定向的情况是很少见的。

在上面的例子里，我们通过“>”操作符把标准输出重定向到了一个文件。在默认的情况下，如果该文件已经存在，它的内容将被覆盖。如果你想改变重定向这个默认的行为，可以使用“set -C”命令，该命令对noclobber选项进行了设置，从而防止了重定向操作对一个文件的覆盖。我们将在本章后面的内容里看到set命令更多的选项。

如果想对文件进行追加，需要使用“>>”操作符。如下所示：

```
$ ps >> lsoutput.txt
```

这条命令将把ps命令的输出追加到文件尾部。

如果想对标准错误输出进行重定向，需要把准备重定向的文件描述符编号加在“>”操作符的前面。因为标准错误输出的文件描述符是2，所以我们要使用“2>”操作符。在需要丢弃错误信息的情况下这个办法很有用，这可以防止它们堆积在屏幕上。

假设我们想从一个脚本程序里使用kill命令来终止一个进程。这样做可能会遇到的一个小麻烦是我们准备终止的进程在执行kill命令之前就已经结束了。如果出现这种情况，kill会向标准错误输出写上一条出错信息，在默认的情况下它就会出现在屏幕上。通过对标准输出和标准错误都施行重定向的办法我们就可以不让kill命令往显示器屏幕上写任何东西了。

请看下面的命令：

```
$ kill  -HUP  1234 > killout.txt  2 > killerr.txt
```

该命令将把输出和错误信息分别重定向到不同的文件里去。

如果我们想把两组输出都捕获到同一个文件里去，可以使用“&”操作符把两组输出弄到一起，请看：

```
$ kill  -1  1234 > killouterr.txt  2 > &1
```

这条命令将把输出和错误输出放到同一个文件里去。请注意操作符出现的顺序。这个命令的意思是“把标准输出重定向到文件killouterr.txt，再把标准错误重定向到与标准输出同样的地方去”。如果你把顺序弄错了，重定向就不会按照你预想的那样工作了。

因为我们可以通过返回码（以后会讲到）了解kill命令的执行结果，所以可能根本就用不着保留标准输出或标准错误。我们可以利用UNIX的万能“垃圾桶”/dev/null让这些输出信息无声无息地消失掉，就像下面这样：

```
$ kill  -1  1234 > /dev/null  2 > &1
```

2.2.2 对输入数据进行重定向

我们不仅能重定向输出信息，还可以重定向输入数据，请看下面的小例子：

```
$ more < killout.txt
```

很明显，在UNIX环境里这样做有点傻乎乎的，因为UNIX中的more命令可以接受文件名作

为自己的参数，这一点与DOS中的对应命令不一样。

2.2.3 管道

我们可以用管道操作符“|”把进程连接在一起。UNIX不同于DOS，用管道连接在一起的进程可以同时运行，并会随着数据流在它们之间的流动而自动协调。

下面是一个简单的例子，我们用sort命令对ps命令的输出进行排序。

如果我们没有使用管道，就必须把这个操作分为几个步骤，如下所示：

```
$ ps > psout.txt
$ sort psout.txt > pssort.out
```

接上一个管道就精巧得多了，如下所示：

```
$ ps | sort > pssort.out
```

如果我们还想在屏幕上看到一页一页的输出结果，可以再多接一个第三进程more，把它们都放在同一个命令行上，如下所示：

```
$ ps | sort | more
```

允许连接的进程数量是没有限制的。假设我们想看看所有运行中进程的名字，不包括shell本身，可以使用下面的命令：

```
$  ps -xo comm | sort | uniq | grep -v sh | more
```

这个命令行先用sort命令把ps命令的输出按字母表顺序排列好，再用uniq命令去掉重复的内容，然后用“grep -v sh”删去名为sh的进程，最后一页一页地把它显示在屏幕上。

在看过这些基本的shell操作之后，我们再来看看脚本程序。

2.3 可以被视为程序设计语言的shell

编写shell程序有两种办法。你可以顺序敲入一系列命令让shell交互式地执行它们；也可以把这些命令保存到一个文件里，再把这个文件当作程序一样去调用执行。

2.3.1 交互式程序

在命令行上直接敲入shell脚本程序是检验小代码块的便捷方法。

> 如果bash不是你系统上的默认shell，就需要由你转换到其他shell上。你可以直接敲入新shell的名字（比如/bin/bash）来运行它并改变命令行提示符。如果bash根本就没有安装在你的系统上，你可以从GNU的站点http://www/gnu.org上免费下载它。它的源代码可移植性非常好，极可能在你的UNIX系统上毫不费力地就通过了编译。

假设我们有许多的C语言程序文件，但我们只想编译那些包含有字符串“POSIX”的。与其先用grep命令在文件里搜索字符串然后再逐个编译那些包含着该字符串的文件，不如用一个下面这样的交互式脚本程序来完成整个操作：

```
$ for file in *
```

```
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
posix
This is a file with POSIX in it - treat it well
$
```

请注意shell提示符“$”会在你敲入shell命令的时候变为“>”。你可以一直敲下去，shell会判断出你是否已经结束了键盘输入并立刻开始执行这个脚本程序。

在这个例子里，grep命令会显示出它找到的包含有“POSIX”的文件，然后more命令会把文件的内容显示在屏幕上。最后，返回shell的提示符。还要注意我们使用了一个用来处理各个文件的shell变量file，它使我们的脚本程序能够自成文档。

shell还完成了文件通配符的扩展匹配（也叫做“globbing”）工作，你已经知道会这样了，对吗？你可能还不知道能够用“?”来匹配单个字符，而“[set]”允许匹配方括号中任何一个单个的字符。“[^set]”对集合进行取反——也就是匹配任何没有出现在你给出的字符集合中的字符。使用花括号“{ }”的通配符（只能够在部分shell上使用，其中包括bash）扩展允许你人为地把几个字符串分在一组，集中起来供shell进行扩展和匹配。请看下面的例子：

```
$ ls my_{finger, toe}s
```

这个例子将列出两个文件，而这两个文件的文件名有几个共同之处。我们用shell对当前子目录中的每一个文件进行检查。

事实上，有经验的UNIX用户会以一种更有效的办法来完成这个简单操作，也许就是使用下面这样的命令：

```
$ more `grep -l POSIX *`
```

或者功能相同的另一种命令形式：

```
$ more $(grep -l POSIX *)
```

而下面这个命令

```
$ grep -l POSIX * | more
```

将输出其内容与模版“POSIX”相匹配的文件的名字。在上面的脚本程序里，我们看到shell使用了命令grep和more来完成主要的工作。shell本身的作用却很简单，只是把几个现有的命令“粘”起来，最终构成了一个新的功能更强的命令行。

如果每次需要执行这一系列命令时都要反反复复地输入这么多的东西当然有些讨厌。我们需要把这些命令保存到一个文件里去，这就是我们常说的shell脚本程序，这样在我们需要的时候就可以随时执行它们了。

2.3.2 编写脚本程序

首先，我们必须用一个文本编辑器创建一个用来保存命令的文件。我们给这个文件取名为first.sh，它的内容如下所示：

```
#!/bin/sh

# first.sh
# This file looks through all the files in the current
# directory for the string POSIX, and then displays those
# files to the standard output.

for file in *
do
  if grep -q POSIX $file
  then
    more $file
  fi
done

exit 0
```

程序中的注释以一个“#”符号开始，一直持续到该行的结束。为了阅读上的方便，我们通常把“#”放在第一列。明确这一点之后，我们来看看第一行上的“#！/bin/sh”，它是注释语句的一个特殊形式，“#！”字符告诉系统同一行上紧跟在它后面的那个参数是用来执行本文件的程序。在这个例子里，/bin/sh就是默认情况下的shell程序。

请注意在注释中我们使用的是绝对路径。因为现实中的某些UNIX系统把解释器路径的字符长度限制在32个以内，所以最好在/bin子目录里至少保留一个你最爱用的shell的符号链接。如果你调用的是一个名字非常长的命令，或者调用的是一个位于深层嵌套子目录中的命令，它就有可能工作不正常。

因为脚本程序本原则上被看做是shell的标准输入（我们在前面介绍过这个概念），所以它可以包含任何能够通过你的PATH环境变量引用到的UNIX命令。

exit命令的作用是确保脚本程序能够返回一个合理的退出码（exit code，本章的后面还将详细介绍这个概念）。当程序是以交互方式运行时很少需要检查它的退出码，但如果你打算从另一个脚本程序里调用这个脚本程序并查看它是否执行成功，返回一个适当的退出码就很重要了。即使你从来也没想过要从另一个脚本程序里调用自己的这个脚本程序，也应该在退出时安排一个合理的退出码。这一做法体现了UNIX程序设计中的一个重要特色：重用。咱们继续学习，要相信自己的脚本程序是有用的。

在shell程序设计里，“0”表示成功。又因为脚本程序本身不检查有无错误，所以我们永远都返回一个表示成功的退出码。我们将在本章后面的内容里对exit命令做进一步的讲解，到那时再告诉大家为什么要用一个“0”来表示执行成功。

虽然在刚才的例子里我们使用“.sh”做为文件名后缀，可UNIX一般很少通过文件的扩展名来确定文件的类型。我们完全可以省略“.sh”，或者给它加上一个不同的扩展名，shell是不关心这些问题的。大多数预安装的脚本程序根本就没有什么扩展名，要想知道它们是否是脚本程序，最好的办法就是使用file命令，比如“file first.sh”或“file /bin/bash”。

2.3.3 把脚本设置为可执行程序

我们的脚本文件输入完成之后，运行它有两种办法。最简单的办法是在调用shell的时候把脚本文件的名字用做一个参数，如下所示：

```
$ /bin/sh first.sh
```

这就行了，但如果能像对待其他UNIX命令那样只敲入脚本程序的名字就调用执行它岂不是更好。没问题，我们只需用chmod命令对这个文件的状态（mode）进行修改，使这个文件能够被全体用户执行就可以了，如下所示：

```
$ chmod +x first.sh
```

> 当然，这可不是用chmod命令把文件设置为可执行程序的惟一办法，请用“man chmod”命令查看它的八进制参数和其他选项的资料。

现在我们可以用下面的命令执行它了：

```
$ first.sh
```

可这样也许不能成功，你会看到一条出错信息告诉你命令没有找到。这很可能是因为没有设置shell的环境变量PATH去查找当前子目录。这个小毛病很好解决，一种办法是在命令行上直接敲入“PATH = $PATH:.”；另一种办法是编辑你的.bash_profile文件，把刚才这条命令加到文件的末尾，退出登录后再重新登录进来。另外，在保存着你脚本程序的子目录里敲入./first.sh也行，它把该文件的相对路径通知给shell。

> 用这种办法修改根用户root的PATH变量是不允许的。这是一个系统安全方面的陷阱，因为以根用户root身份登录上机的系统管理员可能会因不明真相而调用了某个标准命令的改装版本。本书作者之一就曾经这样弄过一次，目的当然是为了向系统管理员指出这一点！即使是在普通账户上，把当前子目录包括在PATH里也多少有些危险。因此，最好的办法是养成在当前子目录中的所有命令前面都加上一个“./”的好习惯。

在确信我们的脚本程序能够正确执行之后，我们可以把它从当前子目录移到一个更合适的地方去。如果这个命令只是供你本人使用的，你可以在自己的登录子目录里创建一个bin子目录并且把它添加到你自己的PATH变量里去。如果你想让其他人也能够执行这个脚本程序，就可以把/usr/local/bin或者其他系统级的子目录用做添加新程序的适当场所。如果你在自己使用的UNIX系统上没有根权限，可以请求系统管理员替你拷贝你的文件。当然你必须先说服他们才行。为了防止其他用户对脚本程序的修改，哪怕只是意外，你应该去掉它的写权限。系统管理员用来设置文件属主权限和访问权限的系列命令如下所示：

```
# cp first.sh /usr/local/bin
# chown root /usr/local/bin/first.sh
# chgrp root /usr/local/bin/first.sh
# chmod 755 /usr/local/bin/first.sh
```

注意我们在这里不是修改特定部分的访问权限标志，而是使用了chmod命令的绝对格式，因为我们清楚地知道需要设置什么样的访问权限。

如果你愿意，还可以使用chmod命令相对长一些但可能更明确的格式，如下所示：

```
# chmod u=rwx, go=rx /usr/local/bin/first.sh
```

关于chmod命令的详细资料请参考它的使用手册。

请记住，在UNIX系统里，如果你拥有保存着某个文件的子目录的写权限，就可以删除这个文件。因此，为了安全起见，你应该确定只有根用户root才能对你想保持安全性的子目录进行写操作。

2.4 shell程序设计的语法

在看过上面那个简单的shell程序例子之后，我们来深入研究一下shell强大的程序设计能力。shell是一种很容易学习的程序设计语言，至少因为它能够在把各个小程序段组合为一个大程序之前很容易地对它们分别进行交互式的测试。利用现代UNIX操作系统的shell，我们可以编写出相当庞大的结构化程序。在接下来的几个小节里，我们将学习：

- 变量：字符串、数字、环境和参数。
- 条件：shell中的布尔值。
- 程序控制：if、elif、for、while、until、case等。
- 命令表。
- 函数。
- 内建在shell中的命令。
- 获取某个命令的执行结果。
- 即时文档（here文档）。

2.4.1 变量

在shell里，使用变量之前并不需要事先对它们做出声明。我们是在第一次用到它们的时候（比如我们给它们赋一个初始值的时候）创建它们的。在默认的情况下，所有变量都被认为是并保存为字符串，即使它们被赋值为数值时也是如此。shell和其他一些工具程序会把“数值”型字符串依次转换为正确的数值，并且按照正确的方式对它们进行操作。UNIX是一个区分字母大小写的系统，因此，shell会认为变量“foo”与“Foo”是不同的，而这两者与“FOO”又是不同的。

在shell里，在变量名前面加上一个“$”字符就可以获得它的内容，用echo命令就可以输出它的内容。只要用到变量，我们就必须在它前面加上一个“$”字符，除非我们是在对该变量进行赋值操作。请看下面的例子，我们可以在命令行上给变量salutation赋以任何值：

```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

注意，如果字符串里包含着空格，就必须用引号把它们括起来。还要注意的是等号两边不能有空格。

我们可以用read命令把用户的输入赋值给一个变量。这个命令需要有一个参数，也就是那个准备读入用户输入数据的变量的名字，它会等待用户输入字符串。当用户按下回车键时read命令继续往下执行。

1. 引号的用法

在继续学习之前，我们需要弄清shell的一个特色——引号的使用。

一般情况下，参数之间是用空白字符分隔的，比如一个空格、一个制表符或一个换行符等。如果想在一个参数里包含一个或多个这样的空白字符，就必须给参数加上引号

像$foo这样的变量在引号中的行为取决于我们使用的引号的类别。如果你把一个带有“$”字符的变量表达式放在双引号里，程序执行到这一行时就会把它替换为它的值。如果你把它放在单引号里，就不会发生替换现象。我们还可以通过在“$”字符前面加上一个“\”字符取消它的特殊意义。

字符串通常都被括在双括号里以防止它们被其中的空白字符分断开，但允许“$”字符引起的名-值替换。

动手试试：变量

我们来看看引号在变量输出中的作用：

```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

它的输出情况是这样的：

```
Hi there
Hi there
$myvar
$myvar
Enter some text
Hello World
$myvar now equals Hello World
```

操作注释：

变量myvar在创建的时候被赋值为字符串“Hi there”。我们用echo命令显示了该变量的内容，大家可以看到在变量名前面加上一个“$”符号得到的是该变量的内容。我们还看到使用双引号时不影响变量的名-值替换，而单引号和反斜线就不进行变量的名-值替换。我们还用red命令从用户那里读入了一个字符串。

2. 环境变量

在shell脚本程序开始执行的时候，某些变量会根据环境中的值进行初始化。这些变量通常

使用大写字母做名字以便把它们和用户在脚本程序里定义的（shell）变量区分开来，后者一般都使用小写字母做名字。具体会创建出哪些变量取决于你的个人配置。在系统的使用手册里列出了许多这样的环境变量，下面是一些比较重要的（见表2-2）：

表 2-2

环境变量	说 明
$HOME	当前用户的登录子目录
$PATH	以冒号分隔的用来搜索命令的子目录清单
$PS1	命令行提示符，通常是“$”字符
$PS2	辅助提示符，用来提示后续输入，通常是“>”字符
$IFS	输入区的分隔符。当shell读取输入数据的时候会把一组字符看做是单词之间的分隔字符，它们通常是空格、制表符和换行符
$0	shell脚本程序的名字
$#	传递到脚本程序的参数个数
$$	该shell脚本程序的进程ID，脚本程序一般会使用它来创建独一无二的临时文件，比如/tmp/tmpfile_$$

如果你想通过执行“env <command>”命令来查看程序在不同环境下是如何工作的，请查阅env命令的使用手册页。

我们也将在下面看到怎样使用export命令在子 Shell中设置环境变量。

3. 参数变量

如果你的脚本程序在调用时还带有参数，就会额外产生一些变量。即使根本没有传递任何参数，上表中的环境变量“$#”也依然存在，只不过它的值是0罢了。

参数变量见表2-3：

表 2-3

参数变量	说 明
$1,$2...	脚本程序的参数
$*	一个全体参数组成的清单，这是一个单独的变量，各个参数之间用环境变量IFS中的第一个字符分隔开
$@	“$*”的一种变量，它不使用IFS环境变量

“$*”和“$@”两个参数之间的区别在X/Open技术规范里有比较详细的解释。

当双括号里的字符串发生名-值替换的时候，“$*”整体扩展为一个数据域，各个参数的值之间用IFS（“internal field separator”的字头缩写，意思是数据域内部分隔符）的第一个字符分隔开；如果取消了对IFS的设置，就用一个空格符加以分隔。如果IFS被设置为一个空字符（这与取消对它的设置可不一样），参数的值就会接合在一起。请看下面的例子：

```
$ IFS=''
$ set foo bar bam
```

```
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

正如大家所看到的，双引号里面的“$@”把各个参数依然扩展为彼此分开的数据，不受IFS变量值的影响。一般来说，如果你想访问脚本程序的参数，用“$@”是明智的选择。

用echo命令可以查看到变量的内容值，而通过read命令可以读入它们。

动手试试：参数和环境变量

下面的脚本程序演示了一些基本的变量处理操作。输入脚本程序的内容并把它保存为文件try_variables，别忘了用“chmod +x try_variables”命令把它设置为可执行的。

```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"

echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"
exit 0
```

如果运行这个脚本程序，我们将得到如下所示的输出结果：

```
$ ./try_variables foo bar baz
Hello
The program ./try_variables is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

程序注释：

这个脚本程序创建了变量salutation并显示它的内容值，然后我们看到各种参数变量以及环境变量$HOME已经存在并有了适当的初始化值。

我们稍后再对参数的名-值替换做进一步介绍。

2.4.2 条件测试

程序设计语言的基础是它们具备对条件进行测试判断的能力，再根据测试结果采取不同的

行动。在讨论这个问题之前，我们先来看看我们在shell脚本程序里可以使用的条件的结构和使用这些条件的控制结构。

shell脚本程序能够对任何可以在命令行上被调用执行的命令的退出码进行测试，其中也包括用户为自己编写的脚本程序。这也是为什么要在所有你自己编写的脚本程序的末尾加上一条exit命令的重要原因。

“test”或“[ ]”命令

在实际工作中，大多数脚本程序都会大量使用“[ ]”或“test”命令——即shell的布尔判断命令。在大多数系统上，这些命令的作用都差不多。把“[ ]”符号当作一条命令多少有点奇怪，但在实际工作中，它在代码里确实会使命令的语法看起来更简单，更明确，就像其他种类的程序设计语言一样。

> 在某些UNIX系统里，这些命令将调用shell中的一个外部程序，但比较现代的shell已经把它内建在其中了。我们将在后面介绍各种命令的内容里再次讨论这个问题。
>
> 因为test命令在shell脚本程序以外用得并不是很频繁，所以许多很少编写shell脚本程序的UNIX用户往往会自己编写一个简单的程序并把它叫做test。如果一个这样的程序不能够工作，十有八九是因为它与shell中的test命令发生了冲突。要想查看你自己的系统是否有一个与此同名的外部命令，可以试试“which test”这样的命令，检查结果一般会是/bin/test或/usr/bin/test。

我们以一个最简单的条件为例来介绍test命令的用法：检查一个文件是否存在。用于实现这一操作的命令是“test -f <filename>”，所以在脚本程序里我们可以写出如下所示的代码：

```
if test -f fred.c
then
...
fi
```

我们也可以写成下面这个样子：

```
if [ -f fred.c ]
then
...
fi
```

test命令的退出码（表明条件是否被满足）决定是否需要执行后面的条件语句。

> 注意要在方括号“[ ]”和被检查的条件之间留出空格。记住“[”字符和单词“test”实际上是一样的，而单词“test”的后面当然应该有一个空格；这样你就不会忘记了。

如果你习惯于把then和if写在同一行上，就一定要用一个分号把then和前面的语句分隔开。如下所示：

```
if [ -f fred.c ]; then
...
fi
```

可以通过test命令进行测试的条件都可以归入下面这三大类别。

1. 字符串比较（见表2-4）

表 2-4

字符串比较	结 果
string1=string2	如果两个字符串相同则结果为真
string1 !=string2	如果两个字符串不同则结果为真
-n string	如果字符串不是空则结果为真
-z string	如果字符串是空（一个空白字符串）结果则为真

2. 算术比较（见表2-5）

表 2-5

算术比较	结 果
expression1 -eq expression2	如果两个表达式相等则结果为真
expression1 -ne expression2	如果两个表达式不等则结果为真
expression1 -gt expression2	如果前一个表达式大于后一个表达式则结果为真
expression1 -ge expression2	如果前一个表达式大于或等于后一个表达式则结果为真
expression1 -lt expression2	如果前一个表达式小于后一个表达式则结果为真
expression1 -le expression2	如果前一个表达式小于或等于后一个表达式则结果为真
! expression	如果表达式为假则结果为真，表达式为真则结果为假

3. 与文件有关的条件测试（见表2-6）

表 2-6

文件条件测试	结 果
-d file	如果文件是一个子目录则结果为真
-e file	如果文件存在则结果为真
-f file	如果文件是一个普通文件则结果为真
-g file	如果文件的set-group-id属性位被设置则结果为真
-r file	如果文件可读则结果为真
-s file	如果文件的长度不为0则结果为真
-u file	如果文件的set-user-id属性位被设置则结果为真
-w file	如果文件可写则结果为真
-x file	如果文件可执行则结果为真

请注意，因为历史遗留问题，“-e”选项不具备可移植性，所以“-f”用得更多一些。

读者可能对set-group-id和set-user-id（也叫做set-gid和set-uid）位有些困惑。set-uid位把程序拥有者的访问权限分配给它；而set-gid位把程序所在分组（group）的访问权限分配给它。这两个属性位都是由chmod命令设置的，对应的命令选项分别是s和g。

set-gid和set-uid标志对shell脚本程序不起作用。

各种与文件有关的条件测试其结果为真的大前提是文件必须存在。这张表格只列出了test命令最常用的选项，完整的选项清单请查阅它的使用手册。如果读者使用的是bash，那么test命令

就已经内建在其中了，你只需敲入“help test”就可以看到详细的资料。本章后面的内容将用到这里给出的部分选项。

我们已经学习了条件，现在来看一下使用它们的控制结构

2.4.3 控制结构

shell有一系列控制结构，而且它们同样与其他程序设计语言很相似。就某些结构而言（比如case语句），shell提供了更强大的功能。

> 在下面的各小节里，“statements”表示测试条件满足时将要执行的一系列命令。

1. if语句

if语句很简单。它对某个命令的执行结果进行测试，然后根据判断结果执行一组语句。如下所示：

```
if condition
then
  statements
else
  statements
fi
```

> 动手试试：使用if命令

普通的用法是问一个问题，然后根据回答作出决定，如下所示：

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
  echo "Good morning"
else
  echo "Good afternoon"
fi

exit 0
```

这将给出如下所示的输出：

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

这个脚本程序用“[ ]”命令对变量timeofday的内容进行测试，测试结果送入if命令进行判断，由它来决定执行哪部分代码。

> 请注意，我们用额外的空白符来缩进if结构内部的语句。这只是为了照顾人们的阅读习惯，shell会忽略这些多余的空白符。

2. elif语句

刚才的这个简单的脚本程序存在几个问题。它会把所有不是“yes”的回答都看做是“no”。

用elif结构就能够避免出现这样的情况，它允许我们在if结构的else部分被执行的时候增加第二个测试条件。

动手试试：用elif结构做进一步测试

我们对刚才的脚本程序做些修改，让它在用户输入“yes”和“no”以外的其他任何东西时报告一条出错信息。

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]
then
  echo "Good morning"
elif [ $timeofday = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

操作注释：

这里的脚本程序与上一个例子很相似，但增加了一个elif命令，它会在前一个if条件不满足的情况下对变量做进一步的测试。如果两次测试的结果都是不成功，就给出一条出错信息并以1为退出码结束脚本程序，调用者可以在调用程序中利用这个退出码来检查这个脚本程序是否执行成功。

3. 一个与变量有关的问题

刚才所做的修改可以弥补比较明显的缺陷，但还有一个更隐蔽的问题没有解决。让我们运行这个新的脚本程序，但这次不回答问题，直接按下回车键。我们将看到如下所示的出错信息：

```
[ : = :  unary operator expected
```

什么地方出问题了呢？原因在于第一个if子句。在对变量timeofday进行测试的时候，它遇到了一个空字符串，这使得if子句成为下面这个样子：

```
if [ = "yes " ]
```

而这不是一个合法的条件。为了避免出现这样的情况，我们必须给变量加上引号：

```
if [ "$timeofday" = "yes" ]
```

这样，一个空变量给我们的就是一个合法的条件判断子句了：

```
if [ ""= "yes" ]
```

新的脚本程序是这样的：

```
#!/bin/sh
```

```
echo "Is it morning? Please answer yes or no"
read timeofday

if [ "$timeofday" = "yes" ]
then
  echo "Good morning"
elif [ "$timeofday" = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

它对回答问题时直接按下回车键的情况也能够应付自如了。

> 如果你想让echo命令去掉每一行后面的换行符，最好的办法是使用printf命令（见后面的内容）而不是echo命令。有的shell允许使用“echo -e”的办法，但这并不是所有系统都支持的。

4. for语句

我们用for结构来循环处理一组值，这组值可以是任意字符串的集合。在程序里可以简单地把全体字符串都列出来，更常见的做法是把它与shell对文件名的通配符扩展结合在一起使用。

它的语法很简单：

```
for variable in values
do
  statements
done
```

> 动手试试：使用固定字符串的for循环

循环值通常是字符串，所以我们可以这样写程序：

```
#!/bin/sh

for foo in bar fud 43
do
  echo $foo
done
exit 0
```

我们得到的输出结果是：

```
bar
fud
43
```

> 如果把第一行由“for foo in bar fud 43”修改为“for foo in "bar fud 43"”会发生什么样事情呢？别忘了加上引号就等于告诉shell把引号之间的一切东西都看做是一个字符串。这是在字符串里保留空格的惟一办法。

操作注释：

这个例子创建了一个变量foo，然后在for循环里每次给它赋一个不同的值。因为shell认为所

有变量在默认情况下包含的都是字符串，所以字符串“43”在使用中与字符串“fud”是一样合法有效的。

动手试试：使用通配符扩展的for循环

正如我们前面提到过的，for循环更经常与shell对文件名的通配符扩展一起使用。这句话的意思是：在字符串的值里加上一个通配符，由shell在程序执行时填充出所有的值。

我们已经在最早的first.sh例子里见过这种做法了。脚本程序利用shell对文件名的通配符扩展把“*”扩展为当前子目录里全体文件的名字，然后它们依次做为for循环的变量“$i”得到处理。我们来简单地看看另外一个通配符扩展的例子：

假设你想把当前子目录里所有文件名的第一个字母是“f”的脚本程序都打印出来，并且你知道自己的脚本程序都带有“.sh”扩展名，那就可以像下面这样做：

```
#!/bin/sh

for file in $(ls f*.sh); do
  lpr $file
done
exit 0
```

操作注释：

这个例子展示了$(command)语法的用途，这一点我们将在后面的内容里做更细的讲解（请参考“在脚本程序语句里执行命令”一节）。简单地说，for命令的参数表来自括在“$()”符号中的命令的输出结果。

shell对“f*.sh”进行扩展，给出所有匹配此模版的文件的名字。

注意，在shell脚本程序里，对变量的各种通配符扩展都是在脚本程序被执行的时候而不是在编写它的时候完成的。因此，变量声明方面的语法错误只有在执行的时候才能发现，就像前面我们给“空”变量加引号的例子里看到的那样。

5. whlie语句

因为所有shell变量值在默认的情况下都被认为是字符串，所以for循环特别适用于对一系列字符串进行循环处理，但在需要执行特定次数的场合就有些力不从心了。

如果我们想让循环执行二十次，请看使用for循环的脚本程序有多么笨拙：

```
#1/bin/sh

for foo in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
do
  echo "here we go again"
done
exit 0
```

而使用通配符扩展又有可能使你陷入不知道到底会循环多少次的窘境。在这种情况下，我们可以使用一个while循环，这是它的语法：

```
while conditon do
   statements
```

```
done
```

请看下面的例子，这是一个普通的口令字检查程序：

```
#!/bin/sh

echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
  echo "Sorry, try again"
  read trythis
done
exit 0
```

这个脚本程序的一个输出示例如下所示：

```
Enter password
password
Sorry, try again
secret
$
```

这当然不是一个查问口令字的安全办法，但它恰如其分地展示了while语句的作用！do和done之间的语句将反复执行，直到条件不再为真为止。在这个例子里，我们检查的条件是变量trythis的值不等于secret，循环将一直执行到$trythis等于secret为止。随后我们将继续执行紧跟在done后面的脚本程序的其他语句。

> 动手试试：循环、循环、再循环

把while结构和数值替换结合在一起，我们就可以让某个命令执行特定的次数。这比我们前面见过的for循环要“苗条”多了。

```
#!/bin/sh

foo=1

while [ "$foo" -le 20 ]
do
  echo "Here we go again"
  foo=$(($foo+1))
done

exit 0
```

> 注意：“$(( ))”结构是最先出现在ksh中的一个用法，后来被包括在X/Open技术规范里。早期的shell要用expr来代替它，我们会在后面介绍这个命令；但这样做比较慢，并且会占用更多的资源。所以只要有可能，你就应该使用命令的“$(( ))”格式。

操作注释：

这个脚本程序使用“[ ]”命令来测试foo的值是否大于20，如果它还小于或等于20，就继续执行循环体。在while循环的内部，语法“((${foo+1}))”用来对括号内的表达式进行数值转换，所以foo会在每次循环里递增。

因为foo不可能变成空字符串，所以在我们对它的值进行测试时不需要把它放在双引号里加

以保护。这样做的原因也是为了培养一种良好的习惯。

6. until语句

until语句的语法如下所示：

```
until condition
do
  statements
done
```

它与while循环很相似，只是把条件测试倒过来了。换句话说，循环将反复执行到条件为真为止，而不是在条件为真时反复执行。

如果我们想让循环不停地执行，直到某些事件发生，就会很自然地想到应该使用until语句。请看下面的例子，我们设置一个报警程序，当某个特定的用户登录上机时它会向显示一个提示警报，该用户的登录名要通过命令行传递到脚本程序里去。如下所示：

```
#!/bin/sh

until who | grep "$1" > /dev/null
do
   sleep 60
done

# now ring the bell and announce the expected user.

echo -e \\a
echo "***** $1 has just logged in *****"

exit 0
```

7. case语句

case结构比我们前面见过的其他语句都稍微复杂些。它的语法如下所示：

```
case variable in
  pattern [ | pattern] ...) statements;;
  pattern [ | pattern] ...) statements;;
  ...
esac
```

这看上去有些庞杂，可case语句确实能够使我们通过一种比较深奥的机制把某个变量的内容与多个模版进行匹配，再根据成功匹配的模版去决定应该执行哪部分代码。请注意，每个模版行都是以双分号（";;"）结尾的。前后模版之间可以有任意多条语句，因此，这个双分号实际起到了分隔符的作用，它标志着前一个语句的结束和后一个模版的开始。

case语句结构能够匹配多个模版，执行多条语句，这使它非常适合用来对用户输入进行分析和处理。要想弄明白case的工作原理，最好的办法就是通过示例来进行说明。我们将在下面通过三个“动手试试”的例子逐步深入地对它进行介绍，每一次都对匹配模版做一点改进。

> 动手试试：case语句示例一：用户输入

我们可以用case语句结构编写一个新版的对用户输入进行检查测试的脚本程序，让它更具选择性，对非预期输入也更宽容一些。请看：

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes)  echo "Good Morning";;
    no )  echo "Good Afternoon";;
    y  )  echo "Good Morning";;
    n  )  echo "Good Afternoon";;
    *  )  echo "Sorry, answer not recognized";;
esac

exit 0
```

操作注释：

当执行到case语句的时候，它会把变量timeofday的内容与各字符串依次进行比较。一旦某个字符串与输入匹配成功，case命令就会执行紧随其后的右括号“)”后面的代码，然后就直接结束了。

case命令会对它用来做比较的字符串进行正常的通配符扩展。因此我们可以在字符串中指定几个字符的后面加上一个“*”通配符。如果“*”号单独出现，就表示匹配任意可能的字符串。所以，我们总是需要在其他匹配字符串之后再加上一个“*”号，这就能够保证即使其他字符串没有得到匹配，case语句也会完成某个默认动作。之所以能够这样做是因为case语句是顺序比较每一个字符串的。它不会去查找所谓的“最佳”匹配，而是查找第一个匹配；而默认动作条件一般都是些“最不可能出现”的条件，所以使用“*”号对脚本程序的调试很有帮助。

动手试试：case语句示例二：合并匹配模版

上面这个case语句明显比多个if语句的版本更明确更精致，但我们还可以把几个不同的匹配模版归纳在一块儿，使之更简单更容易理解。如下所示：

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes | y | Yes | YES ) echo "Good Morning";;
    n* | N* )             echo "Good Afternoon";;
    * )                   echo "Sorry, answer not recognized";;
esac

exit 0
```

操作注释：

在这个脚本程序里，case的每个子句里都使用了多重字符串；case将对各组中不同的字符串进行测试以决定是否需要执行相应的语句。这使脚本程序的长度得以缩短，在实践上也更容易阅读。我们还示范了“*”号的用法，但这样做就有可能匹配上我们意料之外的模版。比如说，如果用户输入了“never”，就会因匹配上“n*”而显示出“Good Afternoon”来，可这并不是我们希望的行为。另外一个需要提请大家注意的地方是对通配符“*”的扩展在双引号里

面不起作用。

动手试试：case语句示例三：执行多条语句

为了让这个脚本程序具备重用性，我们需要在匹配上默认模版的时候给出另外一个退出值。此外，我们还增加使用了一个字符串集合。如下所示：

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes | y | Yes | YES )
           echo "Good Morning"
           echo "Up bright and early this morning"
           ;;
    [nN]*)
           echo "Good Afternoon"
           ;;
    *)
           echo "Sorry, answer not recognized"
           echo "Please answer yes or no"
           exit 1
           ;;
esac

exit 0
```

操作注释：

为了向大家演示模版匹配的其他方法，我们改变了“no”部分的匹配办法。我们还演示了如何在case语句里为每种模版执行多条语句。请大家注意，我们很小心地把最明确的匹配放在最开始，把最一般化的匹配放在最后。这样做很重要，因为case将执行它找到的第一个匹配而不是最佳匹配。如果我们把“*)”放在头一个位置，那不管用户输入的是什么，都会匹配上这个模版。

请注意：esac前面的双分号“;;”是可以省略的。在C语言程序设计实践中，少一个break语句都算是不良的程序设计做法；但这里和C语言程序设计不同：如果默认模版是最后一个匹配情况，默认最后一个双分号“;;”是没有问题的，因为后面没有其他的匹配情况了。

为了让case的匹配功能更强大，我们可以使用下面这样的模版：

```
[yY] | [Yy][Ee][Ss]  )
```

这就限定了允许出现的字母，它既允许多种多样的答案又比“*”通配符提供了更多的控制。

8. 命令表

有时候，我们需要把几条命令按顺序接合成一个序列。比如说在执行某个语句之前需要满足好几个不同的条件，就像下面这样：

```
if [ -f this_file ]; then
    if [ -f that_file ]; then
        if [ -f the_other_file ]; then
            echo "All files present, and correct"
        fi
    fi
fi
```

或者一系列条件中至少需要其中的一个为真，像下面这样：

```
if [ -f this_file ]; then
   foo="True"
elif [ -f that_file ]; then
   foo="True"
elif [ -f the_other_file ]; then
   foo="True"
else
   foo="False"
fi
if [ "$foo" = "True" ]; then
   echo "One of the files exists"
fi
```

虽然这些情况完全可以通过复合方式的if语句来实现，但正如读者所看到的，写出来的程序可够乱的。shell为这类命令表构造特地准备了两种结构，它们分别是AND命令表和OR命令表。两者通常会结合使用，但我们先分别看看它们语法。

(1) AND命令表

AND命令表结构允许我们按这样的方式执行一连串命令：只有在前面所有的命令都执行成功的情况下才执行后一条命令。它的语法是：

```
statement1 && statement2 && statement3 && . . .
```

从左开始顺序执行每条命令，如果它返回的是true，它右边的下一条命令才能够被执行。如此循环直到有一条命令返回false，或者命令表中的全部命令都执行完毕。“&&”的作用是检查前一条命令的返回条件。

每条语句的执行都是彼此独立的，这就允许我们把许多不同的命令混合在一个命令表里，就像下面脚本程序中那样。做为一个整体，如果AND命令表中的所有命令都执行成功，就算它执行成功；否则它就是失败的。

动手试试：AND命令表

在下面的脚本程序里，我们用touch命令（检查文件是否存在，如果不存在就建立它）建立文件file_one并删除了file_two文件。然后，用AND命令表检查各个文件是否存在并通过echo命令给出相应的指示。

```
#!/bin/sh

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

执行这个脚本程序，你会看到如下所示的结果：

```
hello
in else
```

操作注释：

touch和rm命令确保当前子目录中的有关文件处于我们确知的状态。“&&”命令表接下来执行“[-f file_one]”语句，这条语句肯定会执行成功，因为我们已经确保该文件是存在的了。在前一条命令成功的前提下，echo命令得以执行；它也执行成功了（echo命令永远返回true）。再往下执行第三个测试“[-f file_two]。这条命令失败了，因为该文件并不存在。因为这条命令失败了，所以最后一条echo语句没有被执行。因为该命令表中的某条命令失败了，所以“&&”命令表的结果是false，if语句将执行它的else子句。

(2) OR命令表

OR命令表结构允许我们持续执行一系列命令直到有一条成功为止，其后的命令将不再被执行。它的语法是：

```
statement1 || statement2 || statement3 || . . .
```

从左开始顺序执行每条命令。如果它返回的是false，它右边的下一条命令才能够被执行。如此循环直到有一条命令返回true，或者命令表中的全部命令都执行完毕。

“||”命令表和“&&”命令表很相似，只是继续执行下一条命令的条件现在变为其前语句的执行结果必须是失败的（返回false）。

动手试试：OR命令表

沿用上一个例子，但要修改下面程序清单里阴影部分的语句：

```
#!/bin/sh

rm -f file_one

if [ -f file_one ] || echo "hello" || echo " there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

这个脚本程序的输出是：

```
hello
in if
```

操作注释：

头两行代码为整个脚本程序设置好相应的文件。第一个命令“[-f file_one]”失败了，因为这个文件是不存在的。接下来执行echo语句。它返回了true，因此“||”命令表中的后续命令将不会被执行到。因为“||”命令表中有一条命令（echo）返回的是true，所以if语句将执行其then

子句。

以上这两种结构的返回结果是其中执行的最后一条语句的执行结果。

这两个命令表类型的结构与C语言中对多个条件进行测试的执行情况是很相似的。只需执行最少的语句就可以确定其返回结果，而对返回结构没有影响的语句是不会被执行到的。这种情况通常被称为“短路径求值”。

这两种结构组合在一起的结果更具逻辑眼光，请看：

```
[ -f file_one] && command for true ; | command for false
```

在下面的语句里，如果测试成功就会执行前一个命令；否则就会执行后一个命令。请尝试使用更不寻常的命令表来完成你自己的工作吧。

9. 语句块

如果你想在某些只允许使用单个语句的地方（比如AND或OR命令表里）使用多条语句，可以把它们括在花括号（{ }）里来构造出一个语句块。比如说，在本章后面给出的应用程序里，读者会看到如下所示的代码：

```
get_confirm && {
    grep -v "$cdcatnum" $tracks_file > $temp_file
    cat $temp_file > $tracks_file
    echo
    add_record_tracks
}
```

2.4.4 函数

在shell里允许定义函数，如果读者需要编写比较大型的脚本程序，就会想到利用它们来构造自己的代码。

> 做为另外一种办法，你当然也可以把一个大型的脚本程序分成许多小一点的脚本程序，让每个小脚本完成一项小任务。但这样做有几个缺点：在一个脚本程序里执行另外一个脚本程序要比执行一个函数慢得多；执行结果的回传也更困难；并且可能造成小脚本过多的现象。你应该让自己脚本程序的最小构造具备有意义的功能，值得单独保存；在把一个大型脚本程序拆分为一组小脚本的时候应该有这样一个尺度。
>
> 如果你对使用shell来编写大型的程序没有把握，请记住：自由软件基金会FSF的autoconf程序和几种UNIX软件包的安装程序就是shell脚本程序。在一台UNIX系统上，你永远都可以保证有一个基本的shell。事实上，如果没有/bin/sh，大部分UNIX系统根本就不能开机引导，更不用说让用户登录上机了，所以你应该确信自己的脚本程序在绝大部分的UNIX和Linux系统上都会遇上一个能够解释并运行它的shell。

在shell里定义一个函数的办法很简单，写出它的名字，然后是一对空括号“( )”，再把有关的语句放在一对花括号“{ }”里，如下所示：

```
function_name () {
    statements
}
```

动手试试：一个简单的函数

我们从一个确实很简单的函数开始进行学习：

```
#!/bin/sh

foo() {
    echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

运行这个脚本程序会给出下面的输出结果：

```
script starting
Function foo is executing
script ending
```

操作注释：

这个脚本程序还是从自己的最顶部开始执行，这一点与其他脚本程序没有什么分别。但当它遇见“foo() {”结构的时候，会知道那是对一个名为foo的函数的定义。它会记住foo代表着一个函数并从找到“}”字符之后的位置继续执行。当shell遇到在程序行上单独出现的“foo”并准备执行它的时候，就知道应该去执行刚才定义的函数了。当这个函数执行完毕以后，执行过程会返回到调用foo函数的那条语句的后面继续前进。

在调用一个函数之前永远要先对它进行定义，这有点像Pacsal语言里函数必须先于调用而被定义的概念，只是shell里不允许出现任何形式的向前引用。这并不会成为什么问题，因为一切脚本程序都是从顶部开始执行的，所以只要把所有函数定义都放在任何一个函数的调用之前就永远都能保证所有的函数是在其调用之前被定义的了。

在调用一个函数的时候，脚本程序的位置参数“$*”、“$@”、“$#”、“$1”、“$2”等会被替换为函数的参数。这也是读取传递给函数的参数的办法。函数执行完毕之后，有关参数会被恢复为它们原来的值。

有些比较陈旧的shell在函数执行之后无法恢复位置参数的值。如果你想让自己的脚本程序具备可移植性，就最好不依赖这一行为。

我们可以通过return命令让函数返回数字值。让函数返回字符串值的常见办法是让函数把字符串保存在一个变量里，而该变量应该能够在函数结束之后被另外使用。此外，我们还可以像下面这样做：用echo命令发出一个字符串并捕获其结果，如下所示：

```
foo() { echo JAY; }
...
result = "$(foo)"
```

请注意，在shell函数里我们可以通过local关键字声明局部变量，局部变量将局限在函数的

作用范围内。函数可以对全局作用范围中的其他shell变量进行存取。如果某个局部变量和某个全局变量的名字相同，前者就会覆盖后者，但也仅限于函数的作用范围以内。比如说，我们可以对上面的脚本程序进行修改，再看看执行中会发生什么样的情况：

```
#! bin/sh

sample_text="global variable"

foo() {
    local sample_text="local variable"

    echo "Function foo is executing"
    echo $sample_text
}

echo "script starting"
echo $sample_text

foo

echo "script ended"
echo $sample_text

exit 0
```

如果在函数里没有使用return命令指定一个返回值，它返回的就是执行最后一条命令的退出状态码。

在下面的脚本程序my_name里，我们将看到函数的参数是如何被传递的以及如何让函数返回一个true或false结果。

动手试试：从函数里返回一个值

1) 在shell标头之后，我们定义了一个函数yes_or_no：

```
#!/bin/sh

yes_or_no() {
  echo "Is your name $* ?"
  while true
  do
    echo -n "Enter yes or no: "
    read x
    case "$x" in
      y | yes ) return 0;;
      n | no )  return 1;;
      * )       echo "Answer yes or no"
    esac
  done
}
```

2) 下面是它的主程序部分，如下所示：

```
echo "Original parameters are $*"

if yes_or_no "$1"
then
  echo "Hi $1, nice name"
else
  echo "Never mind"
fi
exit 0
```

这个脚本程序的典型输出如下所示：

```
$ ./my_name.sh Rick Neil
Original parameters are Rick Neil
Is your name Rick ?
Enter yes or no: yes
Hi Rick, nice name
$
```

操作注释：

随着这个脚本程序的执行，函数yes_or_no被定义，但先不会执行。在主程序部分的if语句里，脚本程序执行到函数yes_or_no，先把$1替换为最初脚本程序的第一个参数Rick，再把那一行上的剩余内容做为参数传递给这个函数。这些参数被保存在$1、$2等位置参数里，函数使用了它们并向调用者返回了一个值。if语句结构再根据这个返回值去执行相应的子句。

正如我们已经看到的，shell有着丰富的控制结构和条件语句。我们下面来学习一些内建在shell中的命令，然后就要在见不着编译器的情况下解决一个实际的程序设计问题了！

2.4.5 命令

在shell脚本程序的内部我们可以执行两大类命令——可以在命令提示符处执行的“普通”命令和我们前面提到的“内建”命令。“内建”命令是在shell内部实现的，不能做为外部程序被调用。大部分内部命令都是POSIX技术规范的组成部分，并且经常会提供有独立的对应程序。命令是内部的还是外部的一般并没有多大的重要性，只是内部命令执行起来效率更高。

> 既然提到了命令的再实现问题，我想大家可能有兴趣了解UNIX怎样把一个程序用做几个命令或者不同的文件。用“ls -l”命令查看一下mv、cp和ln命令吧。它们在许多系统上实际都是同一个文件，只不过用ln（link，链接）命令创建了几个不同的名字而已。当这个命令被调用的时候，它会先检查自己的第一个参数——在UNIX环境下这将是该命令本身的名字，然后再决定将要采取什么样的动作。

在这里，我们只对那些会在将要编写的脚本程序里用到的主要命令进行介绍，不分内部外部。做为一名UNIX使用者，读者肯定还知道许多其他能够用在命令提示符处的合法命令。永远记住：加上我们在这里介绍的内建命令，你可以在脚本程序里使用其中的任何一个。

1. break命令

我们用这条命令在控制条件尚未满足的情况下从封闭的for、while或until循环里中途退出。你可以给break命令额外加上一个数值参数，它表示准备退出的循环嵌套层数。这会增加脚本程序的阅读难度，因此我们不建议读者使用它。在默认的情况下，它只退出一个循环层。

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4
```

```
for file in fred*
do
    if [ -d "$file" ]; then
        break;
    fi
done

echo first directory starting fred was $file

rm -rf fred*
exit 0
```

2. ":" 冒号命令

冒号命令是一个空命令。它偶尔会被用来简化逻辑条件，相当于true的一个假名。因为它是内建的，所以它比true运行的要快，但它的可读性要差了不少。

读者可能会在while循环的某个条件里看到它，"while :" 表示这是一个无限循环，相当于更常见的 "while true"。

":" 还被用在对变量进行条件化设置的情况下，比如说

```
: ${var:=value}
```

如果没有 ":"，shell会尝试把$var解释为一个命令。

> 在某些旧的shell脚本程序里，你会看到冒号被用在一行的开始以引起一个注释。但现代的脚本程序总是用 "#" 来开始一个注释行，因为这样做的执行效率更高。

```
#!/bin/sh

rm -f fred
if [ -f fred ]; then
    :
else
    echo file fred did not exist
fi

exit 0
```

3. continue命令

类似C语言中的同名语句，这个命令让for\while或until循环跳到下一个循环继续执行，循环变量取循环清单里的下一个值。

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
    if [ -d "$file" ]; then
            echo "skipping directory $file"
        continue
    fi
    echo file is $file
done

rm -rf fred*
exit 0
```

continue可以有一个附加参数，它表示继续开始执行前外跳的循环嵌套层数，也就是说，你可以部分地跳出嵌套着的循环。这个参数很少被使用，因为这会使脚本程序难于理解。请看下面的例子：

```
for x in 1 2 3
do
  echo before $x
  continue 1
  echo after $x
done
```

它的输出是：

```
before 1
before 2
before 3
```

4. "." 句点命令

句点命令的作用是执行当前shell中的某个命令，如下所示：

```
. ./shell_script
```

在一般情况下，当某个脚本程序执行一个外部命令或脚本程序的时候，会创建一个新的执行环境（子shell），命令就在这个新环境里被执行。子环境会在执行完毕后丢弃，只留下退出码返回给上一级shell。但外部的source命令和句点命令（两个差不多是同义词）在执行某个脚本程序里列出的命令时使用的是调用该脚本程序的同一个shell。

这就意味着在一般情况下，新命令对环境变量做出的任何改变都会丢失。而句点命令则允许被执行命令对当前环境进行修改。当我们把脚本程序用做"打包器"为其他后续命令的执行设置环境时，这就很有用了。举例来说，如果你手里同时有几个不同的项目，就可能会遇到需要使用不同参数调用命令的情况，比如调用一个老版本编译器来维护一个老程序等。

在shell脚本程序里，句点命令的作用类似于C或C++语言里的"# include"指令。虽然它实际并不能包括上脚本程序，但它确实是在当前环境里执行命令的，所以你可以通过它在脚本程序里协调变量和函数定义。

在下面的例子里，我们的句点命令是在命令行上执行的，但我们完全可以把它用在脚本程序里。

动手试试：句点"."命令

1) 假设我们有两个保存着不同开发环境的设置情况的文件。要想为老的旧命令设置环境，比如说是classic_set，就该采取如下所示的做法：

```
#!/bin/sh

version=classic
PATH=/usr/local/old_bin:/usr/bin:/bin:.
PS1="classic> "
```

2) 为新命令我们要使用lastest-set，如下所示：

```
#!/bin/sh
```

```
version=latest
PATH=/usr/local/new_bin:/usr/bin:/bin:.
PS1=" latest version> "
```

把这两个脚本程序和句点命令结合在一起使用就可以对环境进行设置，就像下面的示范那样：

```
$ . ./classic_set
classic> echo $version
classic
classic> . latest_set
latest version> echo $version
latest
latest version>
```

5. echo命令

虽然X/Open大力宣传在现代shell里使用printf命令，我们还是依照“常识”使用echo命令来输出后面带有一个换行符的字符串。

一个常见的问题是如何去掉那个换行字符。不幸的是不同版本的UNIX采用的是不同的解决方案。普通办法是使用下面的方法：

```
echo -n "string to output"
```

但你也经常会遇到：

```
echo -e "string to output\c"
```

> 第二种办法“echo -e”确保对反斜线转义字符的解释能够起作用，比如“\t”对应于制表符，“\n”对应于回车换行等。它通常是默认设置的。详细情况请用man命令查看相应的使用手册页。如果你需要一种移植性好的去掉尾缀换行符的办法，可以利用外部的tr命令来去掉它，只是它执行得太慢了。在一般情况下，如果你需要去掉那个换行符并且你的系统上有printf，就坚持使用它好了。

6. eval命令

eval命令对参数进行求值操作。它是内建在shell里的，做为一个独立命令存在的情况并不多见。我们用X/Open技术规范中的一个小例子来说明它的用法，如下所示：

```
foo=10
x=foo
y='$'$x
echo $y
```

它的输出是“$foo”。而

```
foo=10
x=foo
eval y='$'$x
echo $y
```

的输出是“10”。因此，eval命令有点像外部的“$”命令，它给出的是某个变量的值的值。

eval命令很有用，使我们能够随时生成和运行代码。它确实给脚本程序的调试工作增加了难度，但它能够让你完成一些其他方法难于或不可能做到的事情。

7. exec命令

exec命令有两种不同的用法。它经常被用来以另一个不同的程序替换掉当前的shell。请看下面的例子：

```
exec wall "Thank for all the fish"
```

这个命令会在脚本程序里用wall命令替换掉当前的shell。exec语句后面的其他代码都不会被执行了，因为原来执行这个脚本程序的shell不复存在了。

exec的第二个用法是修改当前文件的描述符。

```
exec 3 < afile
```

这将使文件描述符“3”被打开以便从文件afile里读取数据。这种用法非常少见。

8. “exit n”命令

exit命令的作用是使脚本程序以退出码“n”结束运行。在任何一个交互式shell的命令提示符处使用这个命令都会让你退出登录。如果你允许自己的脚本程序在退出时不指定退出状态，脚本程序里最后一个被执行的命令的退出状态就会被用做返回值。自己提供一个退出码永远是一个良好的习惯。

在shell脚本程序设计实践中，退出码“0”表示成功，“1”到“125”之间的数字是留给脚本程序用的出错代码。其余数字有特定含义，请看表2-7：

表 2-7

退 出 码	说 明
126	文件不是可执行的
127	命令未找到
128及以上	引发的一个信号

对许多使用C或C++语言的程序设计人员来说，使用零表示成功有些不同寻常。在脚本程序里，这种用法的一大优点是允许我们使用多达125个用户定义的出错代码而不需要多用一个全局性的出错代码变量。

下面是个简单的例子，如果当前子目录里存在一个名为.profile的文件，就返回“0”表示操作成功。

```
#!/bin/sh

if [ -f .profile ]; then
    exit 0
fi

exit 1
```

如果你习惯精益求精，或者只是想钻研脚本程序，可以组合使用我们前面介绍的AND和OR命令表重写一下这段代码：

```
[ -f  .profile ] && exit 0 | | exit 1
```

9. export命令

export命令把做为它参数的变量名导出到子shell里，使之成为子shell的环境变量。在默认的情况下，在一个shell里创建的变量在此shell调用的下级（子）shell里是不可用的。

export命令把自己的参数创建为一个环境变量，而这个新的环境变量可以被其他脚本程序和被当前程序调用的程序看见。用技术语言讲，被导出的变量将构成从该shell衍生出来的任何子进程的环境变量。我们用下面这个由两个脚本程序export1和export2组成的示例来说明它的用法。

> 动手试试：变量的导出

1) 我们先给出export2的内容：

```
#!/bin/sh

echo "$foo"
echo "$bar"
```

2) 现在轮到export2。在这个脚本程序的末尾，我们将调用export1。

```
#!/bin/sh

foo="The first meta-syntactic variable"
export bar="The second meta-syntactic variable"

export2
```

如果我们运行它们，会得到如下所示的输出结果：

```
$ export1

The second meta-syntactic variable
$
```

输出结果中的第一个空行是因为在export2里没有foo变量，所以对$foo的求值是“空”。对“空”变量进行操作的echo命令输出的是一个空行。

当一个变量被某个shell导出以后，它就可以被该shell调用的任何脚本程序使用，也可以被后续激活的任何shell来使用。如果脚本程序export2又调用了另外一个脚本程序，新脚本程序也可以使用bar的值。

> “set -a”或“set -allexport”命令将把在它之后声明的任何变量都导出为环境变量。

10. expr命令

expr命令把它的参数当做一个表达式进行算术求值。它最常见的用法就是进行数学运算，请看下面的例子：

```
x=`expr  $x + 1`
```

> 反引号（``）使x取值为执行命令“$x + 1”后得到的结果。本章后面我们还将对命令替换做进一步的介绍。

事实上，expr是一个功能强大的命令，它可以完成许多表达式求值计算。其中比较主要的有见表2-8：

表 2-8

表达式求值	说 明
expr1 | expr2	如果expr1非零则等于expr1，否则等于expr2
expr1 & expr2	如果两个表达式都是零则等于零，否则等于expr1
expr1 = expr2	相等
expr1 > expr2	大于
expr1 >= expr2	大于或等于
expr1 < expr2	小于
expr1 <= expr2	小于或等于
expr1 != expr2	不等于
expr1 + expr2	加法
expr1 - expr2	减法
expr1 * expr2	乘法
expr1 / expr2	整数除法
expr1 % expr2	求整数除法的余数

新一代脚本程序中的expr命令通常被替换为我们将要介绍的更有效率的“$((. . .))”语法。

11. printf命令

只有比较近期的shell里才提供有printf命令。X/Open技术规范建议我们应该用它代替echo命令来产生格式化的输出。它的语法是：

```
printf "format string" parameter1  parameter2…
```

格式字符串与C和C++语言里使用的非常相似，但有一些自己的限制规定。主要是不支持浮点数，因为shell里的任何计算都是按照整数计算的原则进行的。格式字符串由各种可打印字符、转义序列和字符转换限定符组成。格式字符串里任何不是“%”和“\”的字符都将按原样出现在输出内容里。

表2-9是它支持的转义序列：

表 2-9

转义序列	说 明
\\	反斜线字符
\a	报警（响铃或蜂鸣）
\b	后退字符
\f	进纸换页字符
\n	换行符
\r	回车符
\t	制表符
\v	垂直制表符
\ooo	八进制数值ooo表示的单个字符

字符转换限定符相当复杂，所以我们只在这里列出最常见的用法。进一步的资料请参考有关的使用手册。字符转换限定符由一个“%”字符和跟在后面的一个转换限定字符组成。主要的转换字符见表2-10：

表 2-10

字符转换限定符	说 明
d	输出一个十进制数字
c	输出一个字符
s	输出一个字符串
%	输出一个“%”字符

格式字符串被用来解释printf语句后续参数的含义并按规定格式显示它们。请看下面的例子：

```
$ printf "%s\n" hello
hello
$ printf "%s %d\t%s" "Hi There" 15 people
Hi There 15   people
```

注意，我们必须用双引号括住“Hi There”字符串，使之成为一个单个的参数。

12. return命令

return命令的作用是使函数返回。我们在前面介绍函数时已经提到过它。return命令有一个数值参数，这个参数在调用该函数的脚本程序里被看做是该函数的返回值。如果没有指定参数，return命令默认返回最后一条命令的退出码。

13. set命令

set命令的作用是为shell设定参数变量。许多命令的输出是以空格分隔的各个值，如果需要使用其中的某个数据域，这个命令就十分有效。

假设我们想在一个shell脚本程序里使用当前月份的名字。系统本身有一个date命令，它的输出结果里就包含着字符串形式的月份名称，但我们需要把它与别的数据域区分开。我们可以用一个“$(...)”结构和set命令来执行date命令并返回想要的结果（我们马上就要看到具体的做法）。date命令输出结果中的月份字符串是它的第二个参数。请看：

```
#!/bin/sh

echo the date is $(date)
set $(date)
echo The month is $2

exit 0
```

这段程序把date命令的输出设置为参数表，再通过位置参数$2取得月份的名字。

请注意，我们在这里使用date命令的目的是把它做为一个简单的例子来说明提取位置参数的办法。因为date命令受语言和地域的影响比较大，所以如果目的真是取出月份的名字，就应该用“date +%B”命令来做这项工作。date命令有许多格式选项，详细资料请参考它的使用手册页。

我们还可以利用set命令和它的选项参数来控制shell的执行方式。其中最常用的是“set +x”，它让脚本程序跟踪显示它当前执行中的命令。

> 我们还会在本章后面介绍程序调试的内容里遇到set命令及其更多的选项参数。

14. shift命令

shift命令把所有参数变量向下移动一个位置，使$2成为$1、$3成为$2，依次递进。原来的

$1将被丢弃，但$0仍将保持不变。如果在调用shift命令的时候还指定了一个数字做参数，就表示参数们要如此递进指定的次数。$*、$@和$#等其他变量也会根据参数变量的新安排做相应的变动。

在扫描处理脚本程序参数时经常要用到shift命令，如果读者的脚本程序需要用到十个或更多的参数，就需要使用shift命令来存取第十个及其以后的参数。

做为示例，我们可以像下面这样依次扫描全部的位置参数：

```
#!/bin/sh

while [ "$1" != "" ]; do
    echo "$1"
    shift
done

exit 0
```

15. trap命令

trap命令用来指定在接收到操作系统信号之后将要采取的行动，信号将在本书后面的内容里详细介绍。一个常见的用途是在脚本程序被中断运行时完成清扫收尾工作。过去的shell经常使用数字代表的信号，而新的脚本程序一般都使用信号的名字，它们都被保存在用#include命令包括进来的signal.h文件里，省略了SIG字头。用“trap -l”可以查看到各种信号。

> 所谓“信号”，简单说来就是以异步方式送达到某个程序的事件。在默认的情况下，它们通常会使该程序终止运行。

trap命令有两部分参数，前一部分是一个将要采取的动作，后一部分是准备陷落的信号名字。

```
trap command signal
```

记住脚本程序通常是从“头”到“尾”解释并运行的，所以你必须把trap命令放在准备保护的脚本程序部分之前。

如果想把某个信号陷阱条件设置回其缺省的状态，在“command”处写上“-”即可。如果想忽略某个信号，把“command”设置为空字符串（''）即可。不带参数的trap命令将列出信号陷阱及其行动的当前清单。

表2-11是一些X/Open技术规范里规定的能够被捕获的比较重要的信号（括号里面的数字是对应的信号编号）。

表 2-11

信　号	说　明
HUP(1)	挂起；一般因终端掉线或用户退登录而引发
INT(2)	中断；一般因按下“Ctrl-C”组合键而引发
QUIT(3)	退出；一般因按下“Ctrl-\”组合键而引发
ABRT(6)	流产；一般因某些严重的执行错误而引发
ALRM(14)	报警；一般用来处理超时问题
TERM(15)	终止；一般由系统在它关机的时候发送出来

动手试试：信号陷阱

下面的脚本程序给出了一些简单的信号处理方法：

```
#!/bin/sh

trap 'rm -f /tmp/my_tmp_file_$$' INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$

echo "press interrupt (CTRL-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
    echo File exists
    sleep 1
done
echo The file no longer exists

trap - INT
echo creating file /tmp/my_tmp_file_$$
date >  /tmp/my_tmp_file_$$

echo "press interrupt (control-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
    echo File exists
    sleep 1
done

echo we never get here
exit 0
```

如果我们运行这个脚本程序，在各个循环里按下“CtrL-C”组合键，我们将得到如下所示的输出：

```
creating file /tmp/my_tmp_file_141
press interrupt (CTRL-C) to interrupt ....
File exists
File exists
File exists
File exists
The file no longer exists
creating file /tmp/my_tmp_file_141
press interrupt (CTRL-C) to interrupt ....
File exists
File exists
File exists
File exists
```

操作注释：

在这个脚本程序里，我们先用trap命令安排它在出现INT（中断）信号时执行“rm -f /tmp/my_tmp_file_$$”命令删除临时文件。然后让脚本程序进入一个while循环，在临时文件存在的情况下不断循环。当用户按下“Ctrl-C”组合键时，语句“rm -f /tmp/my_tmp_file_$$”就会被执行，然后继续下一个循环。但是因为临时文件已经被删除了，所以while循环在第一次执行时就会正常退出。

接着，脚本程序再次使用了trap命令，这次设定的情况是在出现INT信号的时候不执行任何命令。脚本程序重新创建临时文件并在第二个while语句里开始循环。这一次，当用户按下“Ctrl-C”组合键时，因为没有要执行的命令，所以脚本程序就会采取默认的处理行动，也就是

立即结束脚本程序的运行。又因为脚本程序立即被终止了执行，所以最后的echo和exit语句永远也不会被执行。

16. unset命令

unset命令的作用是从环境中删除变量或函数。shell本身设定的只读变量（比如IFS）不受这个命令的影响。请看下面的例子：

```
#!/bin/sh

foo="Hello World"
echo $foo

unset foo
echo $foo
```

它在第一次会显示字符串“Hello World”，但在第二次就只输出一个换行符。

> “foo=”语句和上面这个程序里的unset命令效果差不多，但把foo设置为空与从环境里删除foo是不一样的。

2.4.6 命令的执行

在我们编写脚本程序的时候，经常需要捕获某个命令的执行结果并把它用在shell脚本程序里，也就是说，我们希望执行一条命令并把这条命令的输出放到一个变量里去。通过“$(command)”语法可以实现这一目的，我们在前面介绍set命令的示例中见过这样的用法。还有一种比较老式语法形式，“`command`”，还有不少人仍在使用着。

> 请注意，在脚本程序语句里执行命令的老办法里使用的是反引号（`）而不是我们前面用来括住shell变量的单引号（'）（单引号的作用是防止变量进行扩展）。请只在你需要使自己的脚本程序具备高移植性的时候才使用这种老办法。

新脚本程序使用的都是“$(...)”形式，引入这一形式的目的是为了回避在用反引号括起来的命令中使用“$”、“`”、“\”等字符时相当复杂而又琐碎的规则。如果在“`...`”结构里需要用到反引号，就必须通过“\”字符进行转义。这些规则往往会让程序员头疼不已，有时即使是程序设计老手也会栽在这个问题上，不得不逐字逐句地改正在反引号结构里用错了的引号。

“$(command)”的结果就是其中的命令的输出。注意，那不是该命令的退出状态而是它的字符串输出结果。请看下面的例子：

```
#!/bin/sh

echo The current directory is $PWD
echo The current users are $(who)

exit 0
```

因为当前子目录是一个shell环境变量，所以第一行不涉及在脚本程序语句里执行命令的结构。可who命令就不同了，如果需要在脚本程序里使用它的执行输出，就必须用上这种结构。

把某个命令的执行结果放到一个变量里去这一概念有很大的意义，因为这使在脚本程序里

使用现有命令并捕获其输出的工作变得很容易。如果读者需要把某个命令对标准输出的输出转换为一组参数，或者需要把它们用做另一个程序的参数，就会发现xargs命令能够替你做好这一切。具体细节请参考相应的使用手册。

有时这样做也会遇到一个问题，就是我们打算调用的命令在输出我们想要的内容之前先输出了一些空白字符，或者它输出的东西比我们要求的多。在这种情况下，可以利用前面介绍的set命令加以解决。

1. 算术扩展

我们已经介绍过expr命令，它允许对简单的算术命令进行处理，但这种做法执行起来是相当慢的，因为需要调用一个新shell来处理expr命令。

一种更新颖更适用的办法是“$((. . .))”扩展。把准备求值的表达式括在“$((. . .))”里能够让我们更高效地完成简单的算术计算。如下所示：

```
#!/bin/sh

x=0
while [ "$x" -ne 10 ]; do
    echo $x
    x=$(($x+1))
done

exit 0
```

2. 参数扩展

我们已经见过最简单的参数赋值和扩展情况了，即：

```
foo=fred
echo $foo
```

当我们想在变量名里额外添加字符的时候就会遇到问题。假设我们想编写一个简短的脚本程序来处理名为1_tmp和2_tmp的两个文件。我们试试下面的办法：

```
#!/bin/sh

for i in 1 2
do
    my_secret_process $i_tmp
done
```

但在每次循环里，我们都会看到如下所示的出错信息：

```
my_secret_process: too few arguments
```

哪儿出错了呢？

问题在于shell试图替换变量“$i_tmp”的值，而这个变量其实并不存在。但shell并不会认识到这一错误，相反，它会把一个空值做为这个并不存在的变量的值，因此根本不会有参数被传递到my_secret_process。为了解决变量名里类似于“$i”这样的扩展情况，我们需要把“i”括在一对花括号（{ }）里，就像下面这样：

```
#!/bin/sh

for i in 1 2
do
    my_secret_process ${i}_tmp
done
```

以后，在每次循环里，“i”的值被替换为“$(i)”，从而给出了正确的文件名。也就是说，我们把参数的值替换为一个字符串了。

在shell里我们可以采用很多种参数替换办法。这样做通常都会让我们得到一个精巧的解决方案。

常见的参数扩展替换见表2-12：

表 2-12

参数扩展	说 明
${param:-default}	如果param是空，就把它设置为default的值
${#param}	给出param的长度
${param%word}	从param的尾部开始删除匹配word的最小部分并返回剩余部分
${param%%word}	从param的尾部开始删除匹配word的最大部分并返回剩余部分
${param#word}	从param的头部开始删除匹配word的最小部分并返回剩余部分
${param##word}	从param的头部开始删除匹配word的最大部分并返回剩余部分

在对字符串进行处理的时候，这些扩展替换经常是很有用的。特别是对字符串进行部分删除的那后四个，在对文件名和路径进行处理时作用极大。请看下面的例子。

动手试试：参数的处理

下面这个脚本程序的各个段落演示了各种参数匹配操作符的用法：

```
#!/bin/sh

unset foo
echo ${foo:-bar}

foo=fud
echo ${foo:-bar}

foo=/usr/bin/X11/startx
echo ${foo#*/}
echo ${foo##*/}

bar=/usr/local/etc/local/networks
echo ${bar%local*}
echo ${bar%%local*}

exit 0
```

它将给出如下所示的输出：

```
bar
fud
usr/bin/X11/startx
startx
/usr/local/etc
/usr
```

操作注释：

第一个语句“${foo:-bar}”给出的值是“bar”，因为在这条语句被执行的时候foo没有任何值。但变量foo并没有发生什么变化，它还停留在unset状态。

如果这条语句是“${foo:=bar}”，就会把此变量设置为$foo。这个字符串操作符的作用是检查foo是否存在且不为空。如果是这样的，它就返回变量的值；否则就把foo赋值为bar并返回它。

“${foo:?bar}”将在foo不存在或者被设置为空的情况下显示“foo: bar”并放弃这条命令。

“${foo:+bar}”会在foo存在并不为空的情况下返回bar。选择可太多了！

“{foo#*/}”语句匹配并删除最靠尾部的那个“/”字符（因为“*”匹配的是零个以上的字符）。“{foo##*/}”语句尽量匹配并删除最多的东西，所以它删除了最靠右边的“/”字符及其之前的一切内容。

“{bar%local*}”语句从字符串的末尾开始匹配，它会一直匹配到从尾部算起第一次出现的“local”（及跟在它后面的全部字符）；而“{bar%%local*}”会从尾部开始尽量匹配最多的东西，直到它找到最靠左边的“local”。

根据UNIX操作系统中的管道概念，前一个操作的结果经常需要通过人工进行重定向。假设读者需要用cjpeg程序把一个gif文件转换为一个jpeg文件。

```
$ cjpeg image.gif > image.jpg
```

但有时，读者需要对很多的文件进行这类操作。怎样才能让重定向工作自动进行呢？很简单，这样做就行：

```
#!/bin/sh
for image in  *.gif
do
  cjpeg $image > ${image%%gif}jpg
done
```

我们可以给这个脚本程序起名为giftojpeg，它会在当前子目录里为每一个gif文件创建出一个jpeg文件。

2.4.7 即时文档

从一个shell脚本程序向一条命令传送输入数据有一个特殊的办法，这就是使用一个即时文档（here文档）。允许命令在执行时就好像是在读一个文件或读键盘一样，而它实际上是从脚本程序里得到输入数据的。

即时文档的开始是两个连续的小于号“<<”，然后是一个特殊的字符序列，该序列将在文档的结尾处再次出现。“<<”是shell中起追加作用的重定向操作符，在脚本程序里表示命令的输入是一个即时文档；而特殊的字符序列则用来告诉shell即时文档在什么地方结束。特殊的字符序列不能出现在准备传递给命令的文档内容里，所以要尽量使它既容易记忆又足够不寻常。

动手试试：使用即时文档

最简单的例子就是把输入数据简单地馈入cat命令，如下所示：

```
#!/bin/sh

cat <<!FUNKY!
hello
this is a here
document
!FUNKY!
```

它将给出如下所示的输出：

```
hello
this is a here
document
```

即时文档看起来是一个相当奇怪的功能，但在实际工作中它们的作用是很大的，因为它们使我们能够调用一个交互式的程序，比如一个编辑器，并向它馈入一些事先定义好的输入。但它们更常见的用途是从一个脚本程序里做大量的输出，就像我们在刚才的示例中看到的那样。这样做可以避免每输出一行都要用上一个echo命令那样麻烦。我们加在特殊字符序列两端的惊叹号“!”的作用是保证不会引起什么误会。

如果我们想按预先安排好的做法对一个文件中的某几行进行处理，可以使用ed行编辑器并在脚本程序里通过一个即时文档向它馈入命令。如下所示：

动手试试：here文档的另一个用法

1) 下面是文件a_text_file的内容：

```
That is line 1
That is line 2
That is line 3
That is line 4
```

2) 我们可以通过混合使用一个here文档和ed编辑器来修改这个文件。

```
#!/bin/sh

ed a_text_file <<!FunkyStuff!
3
d
.,\$s/is/was/
w
q
!FunkyStuff!

exit 0
```

如果我们运行这个脚本程序，文件的新内容是：

```
That is line 1
That is line 2
That was line 4
```

操作注释：

这个脚本程序很简单，它调用ed编辑器并向它传递命令，先让它移动到第三行，然后删除该行，再把当前行（因为第三行刚刚被删掉了，所以当前行现在就是原来的最后一行，即第四行）中的“is”替换为“was”。完成这些操作的ed命令是来自脚本程序，它们构成了一个即时文

档，也就是“!FunkyStuff!”之间的那些内容。

> 注意，在这个即时文档里我们用了“\”来防止“$”字符被扩展。“\”的作用是对“$”进行转义，让shell知道不要把“$s/is/was/”替换为它的值，而它也确实没有什么值。shell把“\$”传递为“$”，再由ed编辑器对它进行解释。

2.4.8 调试脚本程序

脚本程序的调试一般都很容易进行，但应该说没有什么特定的辅助工具。我们将对常用的方法做一个快速的总结。

在出现错误的时候，shell的做法一般都很简单，它会打印出引起这个错误的语句所在的行号。如果这个错误不是立刻出现的，我们可以额外增加一些echo命令来查看变量的值，也可以交互式地逐个敲入命令来调试一段代码。

因为脚本程序是被解释运行的，所以在脚本程序的修改和重试过程中没有编译方面的开支和负担。

跟踪复杂错误的主要方法是设置各种shell选项。面这又有两种方法可供选用：一是在调用shell时加上命令行选项，二是使用set命令。表2-13是对这两种方法的总结。

表 2-13

命令行选项	set命令选项	说 明
sh -n <script>	set -o noexec	只检查语法错误，不执行命令
	set -n	
sh -v <script	set -o verbose	在执行命令之前回显它们
	set -v	
sh -x <script	set -v xtrace	在处理完命令行之后回显它们
	set -x	
	set -o nounset	如果使用了未定义变量就给出一条出错信息
	set -u	

用“-o”选项置位set命令的选项标志，用“+o”取消设置；也可以使用简化的单字符选项。

加上xtrace选项就能看到一份简单的执行跟踪报告。在刚开始调试检查工作时，我们可以先用命令行选项跟踪命令的执行情况；然后随着调试的细化和深入，逐步缩小跟踪范围，把xtrace标志（用来开、关跟踪执行命令的功能）放到我们认为有毛病的脚本程序代码的前后。执行命令跟踪功能的作用是让shell在执行脚本程序中的每行程序之前先显示已经对变量做了扩展的代码。在每行代码的前面用“+”号的个数指出变量扩展的层次数（这是它的缺省设置情况）。我们可以通过对shell配置文件里的PS4shell变量进行设置的办法把“+”修改为更有意义的文字说明。

在shell里，我们还可以通过陷落EXIT信号在脚本程序退出执行时查看到它的状态，具体做法是在脚本程序的开始放上一条如下所示的语句：

```
trap 'echo Exiting: critical variable = $critical_variable' EXIT
```

2.5 shell程序设计示例

至此，我们已经学习完shell作为程序设计语言的主要功能了。现在，我们把已经学过的知识总结在一起，编写一个有实际用途的示范程序。

我们将利用在这本书里学到的知识逐步完成一个对CD唱盘进行管理的数据库软件。我们从shell脚本程序开始，以后再用C语言重写一遍，给它逐步加上数据库等新的功能。就说到这，我们动手吧。

2.5.1 工作需求

我们来设计并编写一个管理CD唱盘的程序。假设我们收集了大量的CD唱盘。因为我们正在学习UNIX程序设计，所以以边学边练的方式实现一个电子化的唱盘目录看起来是个很不错的好主意。

在刚开始入手的时候，我们至少应该做到能够把各张CD唱盘的基本资料保存起来，比如唱盘的名字、音乐类型、歌唱家或作曲家的名字等。我们还想再增加一些简单的曲目资料。

我们希望能够以每张唱盘为单位对它们进行检索，对曲目方面的细节暂不考虑。

为了让这个小小的应用程序比较完整，我们还希望能够在这个应用程序里对唱盘资料进行输入、修改和删除。

2.5.2 设计

既然我们有对数据进行修改、检索和显示这三项操作要求，采用一个简单的菜单应该是很合适的做法。我们准备保存的数据全部都是文字性的，所以如果我们收集的CD唱盘还不是太多，就不必非使用复杂的数据库不可，有几个简单的文本文件就足可以了。把资料保存在文本文件里将使我们的应用程序比较简单；如果我们的操作要求又有了变化，文本文件总要比其他类型的文件更容易处理一些。至少我们可以用一个编辑器来人工输入或删除数据，不必非得编出一个程序来。

在数据存储方面我们需要做出一个重要的设计决定：一个文件够用吗？如果够用，它应该采用什么样的格式呢？除了曲目信息，我们想要保存的大部分资料在每张CD唱盘上只出现一次（我们暂不考虑某些CD唱盘上会有多名作曲家或歌唱家作品的情况）；而几乎所有CD唱盘都有一个以上的曲目。

需要给CD唱盘上的曲目数量加一个上限吗？这好像是个不怎么讲理也没什么必要的限制，所以我们还是抛开这个念头吧。

如果我们对曲目的个数不设置上限，就有以下几种做法可供选择：

- 只有一个文件，里面用一行来保存“标题”信息，再用n行保存该CD唱盘上的曲目信息。
- 每张CD唱盘的全部信息都保存在一行上，让这一行的长度一直延续到信息都保存完为止。
- 把标题信息和曲目信息分开，分别保存在不同的文件里。

只有第三个做法能够让我们灵活修改文件的格式，而如果今后真的想把我们的数据库转换为关系数据库形式（请参考第7章内容）的话，就肯定要对文件格式进行修改，因此我们选它作

为我们的决策。

下一个决策是要在文件里放入哪些信息。

我们决定，对每张CD唱盘，我们保存以下信息：

- CD唱盘的目录编号。
- 标题。
- 曲目风格（古典、摇滚、流行、爵士等）。
- 作曲家或歌唱家。

对每个曲目，我们保存：

- 曲目编号。
- 歌名或曲名。

为了把这两个文件“结合”起来，我们需要把曲目信息与CD唱盘的其他信息挂上钩。我们选CD唱盘的目录号来担当这一任务。因为它对每张CD唱盘来说是独一无二的，在“标题”文件里只出现一次，在“曲目”文件里也只出现一次。

我们来看看下面这个“标题”文件（见表2-14）。

表 2-14

目录编号	标 题	曲目风格	作 曲 家
CD123	Cool sax	Jazz（爵士）	Bix
CD234	Classic violin	Classical（古典）	Bach
CD345	Hit99	Pop（流行）	Various

再看看相应的“曲目”文件（见表2-15）。

表 2-15

目录编号	曲目编号	歌名或曲名
CD123	1	Some jazz
CD123	2	More jazz
CD345	1	Dizzy
CD234	1	Sonata in D minor

两个文件通过目录编号“结合”在一起。别忘了，“标题”文件中的一个数据项在“曲目”文件里一般都对应着多行数据。

最后一个决策是如何分隔这些数据项。在关系数据库里，长度固定的数据域比较常见，但有时并非最方便。另一种常见办法是用一个逗号，我们在这里就选这个办法（也就是一个变量之间用逗号分隔的文件，按英文“comma-separated variable”缩写为CSV文件）。

在后面的“动手试试”部分，为了不让你找不着北，我们把将会用到的函数列在下面。

```
get_return()
get_confirm()
set_menu_choice()
insert_title()
insert_track()
add_record_tracks()
```

```
add_records()
find_cd()
update_cd()
count_cds()
remove_records()
list_tracks()
```

动手试试：CD唱盘管理程序

1) 这个脚本程序示例的第一行，就像平时一样，是确保自己能够作为脚本程序执行的语句“#!/bin/sh”，随后是一些版权信息。如下所示：

```
#!/bin/sh

# Very simple example shell script for managing a CD collection.
# Copyright (C) 1996-99 Wrox Press.

# 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; either version 2 of the License, or (at your
# option) any later version.

# This program is distributed in the hopes that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
# Public License for more details.

# You should have received a copy of the GNU General Public License along
# with this program; if not, write to the Free Software Foundation, Inc.
# 675 Mass Ave, Cambridge, MA 02139, USA.
```

2) 首先要作的事情是设置好脚本程序将要用到的全局变量的初始值。我们设置好“标题”和“曲目”文件，还设置了一个临时文件。然后设置“Ctrl-C”键陷阱，做好脚本程序被用户中断运行时删除临时文件的准备工作。

```
menu_choice=""
current_cd=""
title_file="title.cdb"
tracks_file="tracks.cdb"
temp_file=/tmp/cdb.$$
trap 'rm -f $temp_file' EXIT
```

3) 现在开始定义函数。因为脚本程序将从文件的第一行开始执行，所以这样做可以让它在我们调用其中任何一个之前知道每个函数的定义情况。

为了避免在几个地方反复书写同样的代码，最开始的两个函数是简单的工具性函数。

```
get_return() {
  echo -e "Press return \c"
  read x
  return 0
}

get_confirm() {
  echo -e "Are you sure? \c"
  while true
  do
    read x
    case "$x" in
      y | yes | Y | Yes | YES )
        return 0;;
      n | no  | N | No  | NO )
```

```
        echo
        echo "Cancelled"
        return 1;;
      *) echo "Please enter yes or no" ;;
    esac
  done
}
```

4) 下面是主菜单函数set_menu_choice。菜单内容是动态变化的，如果用户已经选中某张CD唱盘，主菜单里会多出几个选项。

请注意，“echo-e”命令可能不能被移植到某些shell去。

```
set_menu_choice() {
  clear
  echo "Options :-"
  echo
  echo "   a) Add new CD"
  echo "   f) Find CD"
  echo "   c) Count the CDs and tracks in the catalog"
  if [ "$cdcatnum" != "" ]; then
    echo "   l) List tracks on $cdtitle"
    echo "   r) Remove $cdtitle"
    echo "   u) Update track information for $cdtitle"
  fi
  echo "   q) Quit"
  echo
  echo -e "Please enter choice then press return \c"
  read menu_choice
  return
}
```

5) 两个很短的函数insert_title和insert_track用来往数据库文件里添加数据。虽然有的人不喜欢这种长度只有一行的函数，可它们被用在其他函数里时会因为意义明确而便于他人理解。

这两个小函数之后是比较大的add_record_track函数，后者要用到它们。该函数用模版匹配确保没有输入逗号（因为我们把逗号用做数据域之间的分隔符），用算术操作在用户输入曲目时递增当前曲目的编号。

```
insert_title() {
  echo $* >> $title_file
  return
}

insert_track() {
  echo $* >> $tracks_file
  return
}

add_record_tracks() {
  echo "Enter track information for this CD"
  echo "When no more tracks enter q"
  cdtrack=1
  cdttitle=""
  while [ "$cdttitle" != "q" ]
  do
      echo -e "Track $cdtrack, track title? \c"
      read tmp
      cdttitle=${tmp%%,*}
      if [ "$tmp" != "$cdttitle" ]; then
        echo "Sorry, no commas allowed"
        continue
      fi
      if [ -n "$cdttitle" ] ; then
        if [ "$cdttitle" != "q" ]; then
          insert_track $cdcatnum,$cdtrack,$cdttitle
```

```
        fi
      else
        cdtrack=$((cdtrack-1))
      fi
    cdtrack=$((cdtrack+1))
  done
}
```

6) add_records函数用来输入新CD唱盘的标题信息。

```
add_records() {
  # Prompt for the initial information

  echo -e "Enter catalog name \c"
  read tmp
  cdcatnum=${tmp%%,*}

  echo -e "Enter title \c"
  read tmp
  cdtitle=${tmp%%,*}

  echo -e "Enter type \c"
  read tmp
  cdtype=${tmp%%,*}
  echo -e "Enter artist/composer \c"
  read tmp
  cdac=${tmp%%,*}

  # Check that they want to enter the information

  echo About to add new entry
  echo "$cdcatnum $cdtitle $cdtype $cdac"

  # If confirmed then append it to the titles file

  if get_confirm ; then
    insert_title $cdcatnum,$cdtitle,$cdtype,$cdac
    add_record_tracks
  else
    remove_records
  fi

  return
}
```

7) find_cd函数的作用是在CD唱盘“标题”文件里查找CD唱盘的有关资料，它使用grep命令来完成这一工作。我们需要知道字符串在“标题”文件里出现了多少次，但grep命令的返回值只能告诉我们是没有找到该字符串还是找到了很多次。为了解决这一问题，我们把grep的输出保存到一个临时文件里去，字符串每被找到一次就在临时文件里保存一行对应的数据，然后再统计临时文件里的行数就可以达到我们的目的了。

单词统计命令wc的输出里包含着用来分隔被统计文件中行数、单词数和字符个数的空白字符。我们利用“$(wc -l $temp_file)”记号从其输出结果里提取出第一个参数赋值给表示文件中文本行数的linefound变量。如果我们要用到wc命令输出中其他靠后的值，可以利用set命令把shell参数变量设置为wc命令的输出结果。

我们把IFS（Internal Field Separator，内部数据域分隔符）设置为一个逗号（,），这样就可以读取以逗号分隔的数据域了。另一个可替换使用的命令是cut。

```
find_cd() {
  if [ "$1" = "n" ]; then
    asklist=n
  else
```

```
    asklist=y
  fi
  cdcatnum=""
  echo -e "Enter a string to search for in the CD titles \c"
  read searchstr
  if [ "$searchstr" = "" ]; then
    return 0
  fi

  grep "$searchstr" $title_file > $temp_file

  set $(wc -l $temp_file)
  linesfound=$1

  case "$linesfound" in
  0)    echo "Sorry, nothing found"
        get_return
        return 0
        ;;
  1)    ;;
  2)    echo "Sorry, not unique."
        echo "Found the following"
        cat $temp_file
        get_return
        return 0
  esac

  IFS=","
  read cdcatnum cdtitle cdtype cdac < $temp_file
  IFS=" "

  if [ -z "$cdcatnum" ]; then
    echo "Sorry, could not extract catalog field from $temp_file"
    get_return
    return 0
  fi

  echo
  echo Catalog number: $cdcatnum
  echo Title: $cdtitle
  echo Type: $cdtype
  echo Artist/Composer: $cdac
  echo
  get_return

  if [ "$asklist" = "y" ]; then
    echo -e "View tracks for this CD? \c"
      read x
    if [ "$x" = "y" ]; then
      echo
      list_tracks
      echo
    fi
  fi
  return 1
}
```

8) update_cd函数让我们能够重新输入CD唱盘的有关资料。注意，我们想要（用grep）找的东西是以$cdcatnum打头（通过标志“^”）并且后面跟着一个逗号的文本行，因此需要把$cdcatnum变量的值扩展括在一对花括号（{ }）里再加上一个逗号，逗号和CD目录编号之间没有空白字符，这样才能找到我们想要的东西。这个函数中还有另外一对花括号，它们出现在get_confirm返回true值时，作用是把多个语句括在一起组成一个语句块。

```
update_cd() {
  if [ -z "$cdcatnum" ]; then
    echo "You must select a CD first"
```

```
    find_cd n
  fi
  if [ -n "$cdcatnum" ]; then
    echo "Current tracks are :-"
    list_tracks
    echo
    echo "This will re-enter the tracks for $cdtitle"
    get_confirm && {
      grep -v "^${cdcatnum}," $tracks_file > $temp_file
      mv $temp_file $tracks_file
      echo
      add_record_tracks
    }
  fi
  return
}
```

9) count_cds函数的作用是快速统计数据库中的CD唱盘个数和曲目总数。

```
count_cds() {
  set $(wc -l $title_file)
  num_titles=$1
  set $(wc -l $tracks_file)
  num_tracks=$1
  echo found $num_titles CDs, with a total of $num_tracks tracks
  get_return
  return
}
```

10) remove_records函数的作用是从数据库文件里删除对应的数据项，它通过“grep -v”命令删除所有匹配到的字符串。注意，我们必须使用一个临时文件来完成这一工作。

如果我们使用下面这样的命令：

```
grep -v "^$cdcatnum" > $title_file
```

$title_file文件就会在grep命令开始执行之前被“>”输出重定向操作设置为空文件，从面使grep命令实际上将从一个空文件里读取数据。

```
remove_records() {
  if [ -z "$cdcatnum" ]; then
    echo You must select a CD first
    find_cd n
  fi
  if [ -n "$cdcatnum" ]; then
    echo "You are about to delete $cdtitle"
    get_confirm && {
      grep -v "^${cdcatnum}," $title_file > $temp_file
      mv $temp_file $title_file
      grep -v "^${cdcatnum}," $tracks_file > $temp_file
      mv $temp_file $tracks_file
      cdcatnum=""
      echo Entry removed
    }
    get_return
  fi
  return
}
```

11) list_tracks函数还是使用grep命令来找出我们想要的数据行，它还通过cut命令去处理有关的数据域，再通过more命令按页提供输出。读者可以自己算一下要是用C语言来重新编写出这段大约20行左右的代码需要多少条语句，你不得不佩服shell可是多么强大的一个工具。

```
list_tracks() {
  if [ "$cdcatnum" = "" ]; then
    echo no CD selected yet
    return
  else
    grep "^${cdcatnum}," $tracks_file > $temp_file
    num_tracks=$(wc -l $temp_file)
    if [ "$num_tracks" = "0" ]; then
      echo no tracks found for $cdtitle
    else {
      echo
      echo "$cdtitle :-"
      echo
      cut -f 2- -d , $temp_file
      echo
    } | ${PAGER:-more}
    fi
  fi
  get_return
  return
}
```

12) 到这里所有的函数都已经定义好了，我们进入主程序部分。开头那几行先查明文件是否存在，然后调用主菜单函数set_menu_choice，再根据主菜单函数的输出进行相应的操作。

如果用户选择了退出（“q”或“Q”），我们先删除临时文件，再显示结束信息，最后以成功条件（退出码为0）退出这个应用程序。

```
rm -f $temp_file
if [ ! -f $title_file ]; then
  touch $title_file
fi
if [ ! -f $tracks_file ]; then
  touch $tracks_file
fi

# Now the application proper

clear
echo
echo
echo "Mini CD manager"
sleep 1

quit=n
while [ "$quit" != "y" ];
do
  set_menu_choice
  case "$menu_choice" in
    a) add_records;;
    r) remove_records;;
    f) find_cd y;;
    u) update_cd;;
    c) count_cds;;
    l) list_tracks;;
    b)
      echo
      more $title_file
      echo
      get_return;;
    q | Q ) quit=y;;
    *) echo "Sorry, choice not recognized";;
  esac
done

#Tidy up and leave

rm -f $temp_file
```

```
echo "Finished"
exit 0
```

补充说明

脚本程序开始处的trap命令起作用是陷阱用户按下“Ctrl-C”组合键的事件，根据终端设置情况的不同，它将引发EXIT或INT信号。

菜单选择操作还有其他的实现办法，特别值得一提的是bash或ksh（它没有被列在X/Open技术规范里）提供的select结构，它是一个专门用来处理菜单选择的结构。如果具备在脚本程序里使用这种结构的条件，并且不介意移植性稍差的话，可以考虑这种办法。为用户准备的多行信息还可以用即时文档来实现。

读者可能已经注意到在开始一个新的CD唱盘记录时没有对它的主关键字进行检验；新代码忽略了使用同样的代码的后续的唱盘名称，而把它们上面的曲目添加到第一个CD唱盘的曲目清单里。如下所示：

```
1 First CD Track 1
2 First CD Track 2
1 Another CD
2 With the same CD key
```

我们把这个问题和其他改进留给读者，请发挥你们自己的想象力和创造力，因为在GPL版权规则下，你们完全可以对这些代码进行修改。

2.6 本章总结

在这一章里，我们看到shell本身就是一种功能强大的程序设计语言。它能够调用其他程序并对它们的输出进行处理，这使shell成为文本和文件处理方面一个理想的工具。

当你下一次需要小工具程序的时候，请考虑是否能够用一个脚本程序把某些UNIX命令组织起来解决自己的问题。在不使用编译器的情况下，你完全可以编写出大量的工具程序，也许你自己都会吃惊的。

第3章　如何使用和处理文件

在这一章里，我们将学习关于UNIX中的文件和子目录的知识以及如何对它们进行处理。我们将学习如何建立文件、如何打开它们、如何对它们进行读写操作以及如何关闭它们。我们还将学习程序如何对子目录进行处理，比如如何建立、扫描和删除它们等。在上一章岔开去讨论shell程序设计之后，我们终于开始使用C语言进行程序设计了。

在开始讨论UNIX对文件的I/O程序之前，我们先来复习一下与文件、子目录和设备有关的概念。对网络和子目录进行的处理需要通过系统调用（UNIX中与Windows中的应用程序接口API对应的概念），但还有许多库函数和标准I/O函数库（stdio库）等使文件处理更直接高效。

本章的大部分内容都将用于讨论对文件和子目录进行处理的各种调用。因此，这一章的学习范围主要有：

- 文件和设备。
- 系统调用。
- 库函数。
- 文件访问的底层操作。
- 对文件进行管理。
- 标准I/O库。
- 格式化的输入和输出。
- 文件和子目录的维护。
- 对子目录进行扫描。
- 错误及其处理。
- 其他高级论题。

3.1　UNIX的文件结构

读者可能会问："为什么要在这里讨论文件结构？我已经知道了。"这么说吧，UNIX环境中的文件具有非常重要的意义，因为它们提供了到操作系统服务和设备的简单而又统一的接口。UNIX里的一切事物都是文件。这就是原因！

这就意味着在一般情况下，程序完全可以像对待文件那样对待磁盘文件、串行口、打印机以及其他设备。在后面的内容里我们将给出一些例外情况，比如网络方面等；但就主要用法来说，你只需要五个基本的函数就足以应付大多数问题，它们是：open、close、read、write和ioctl。

目录是特殊形式的文件。在现如今的UNIX操作系统里，甚至超级用户也不再被允许直接对目录进行写操作。在正常情况下，所有用户都必须通过高级的opendir/readdir操作接口才能读取

目录，不再需要了解系统中的目录是如何具体实现的。我们将在本章后面的内容里学习对目录本身进行操作的特殊函数。

可以这么说，UNIX里面的任何事物都可以用一个文件来代表，要不就可以通过特殊的文件来进行操作。这当然会与我们熟悉的正常文件有一定的区别，但其基本原则是一致的。下面就来看看我们刚才提到的特殊文件。

3.1.1 目录结构

一个文件，除了本身包含的内容以外，还会有一个名字和其他一些用于管理方面的“属性”信息，比如文件的建立/修改日期、它的访问权限等。这些属性都被保存在一种我们称之为inode（结点）的数据结构里，文件的长度和它在磁盘上的存放地点也保存在这里。系统使用的是文件的结点编号，而子目录结构只不过是为方便人们的使用而给文件起的名字。

目录是一种用来保存结点号和其他文件的名字的特殊文件。目录文件里一个数据项实际上是一个指向代表某个文件的inode结点的链接指针，删除该文件名就等于删除与之对应的链接指针（文件的结点号可以通过“ln -i”命令查看）。利用ln命令，我们可以在不同的子目录里创建到同一个文件的链接。如果文件的链接个数（即“ls -l”命令的输出中跟在访问权限后面的那个数字）变为零，就表示该结点以及它所指向的数据不再有人使用，磁盘上的相应位置就会被标记为可用空间。

文件排列在目录里，目录还可以有下一级的子目录。这就构成了我们熟悉的树状目录结构。用户（比如neil用户）通常会把自己的文件保存在“登录子目录”里，比如/home/neil；里面再进一步划分为电子邮件、商业信函、工具程序等各项用途的子目录。许多UNIX操作系统下的shell都允许通过一个简单（但是了不起）的记号让用户能够直接到达自己的登录子目录，这就是波浪线字符“~”。要想进入他人的登录子目录，敲入“~user”（“~”加用户名）即可。各个用户的登录子目录通常是一个高层目录的下级子目录，这个高层目录就是为这个目的而创建的，在刚才举的例子里，它就是“/home”。但要提醒读者注意的是，标准函数库里的函数不能识别和处理文件名参数中使用的波浪线符号。

“/home”又是根目录“/”的一个下级子目录，根目录位于目录树的最顶端，它的各种下级子目录里包含着系统中的一切文件。根目录一般都有存放系统程序（二进制可执行文件）用的“/bin”、系统配置文件用的“/etc”和系统函数库的“/lib”等子目录。代表物理设备和提供这些设备操作接口的文件传统上会被放在一个名为“/dev”的子目录里。Linux文件系统布局的进一步资料请参考“Linux File System Standard”（《Linux文件系统标准》），读者可以通过“man hier”命令了解一下树状目录结构的简单描述见图3-1。

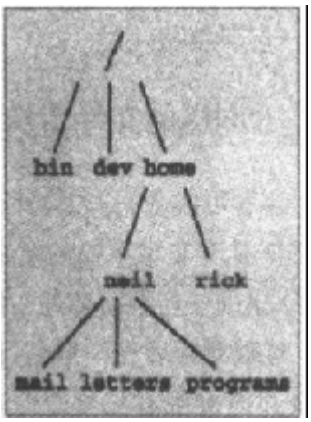


图 3-1

3.1.2 文件和设备

硬件设备在UNIX操作系统通常也被表示（映射）为文件。比方说，作为根用户，我们下面

的命令把CD-ROM驱动器映射为一个文件：

```
$ mount -t iso9660 /dev/hdc /mnt/cd_rom
$ cd /mnt/cd_rom
```

这个命令把（开机引导时被加载为hdc的）CD-ROM驱动器中的当前内容挂装为子目录/mnt/cd-rom下的文件结构。随后我们就可以象对普通文件进行操作那样在CD-ROM盘的子目录中漫游，只不过其内容是只读的罢了。

比较重要的设备文件有三个，它们是/dev/console、/dev/tty和/dev/null。

1. /dev/console设备

这个设备代表的是系统控制台。出错和诊断信息通常会被发送到这个设备。每个UNIX系统都会有一个专门的终端或显示屏用来接收控制台消息。在过去，它会是一台专用的打印终端。在现代的工作站，它通常就是“活跃”的虚拟控制台，而在X窗口系统下，它会是屏幕上一个特殊的控制台窗口。

2. /dev/tty设备

特殊文件/dev/tty是进程控制终端（键盘和显示屏，或者键盘和窗口）的一个假名（逻辑设备）——如果这个程序有控制终端的话。举例来说，通过cron运行的进程就没有控制终端，所以它们就不能打开/dev/tty设备。

在能够使用该设备的情况下，/dev/tty允许程序直接向用户输出信息，不管用户具体使用的是哪种类型的伪终端或硬件终端。在标准输出被重定向的情况下，这一功能非常有用。命令“ls -R | more”就是一个这样的例子，因为more程序需要提示用户进行键盘操作之后才能显示下一页内容。我们将在第5章看到更多使用/dev/tty的例子。

需要提醒大家注意的是，/dev/console设备只有一个，但通过/dev/tty能够访问的物理设备却可以说是数不胜数。

3. /dev/null设备

它就是我们所说的空（null）设备。所有写向这个设备的输出都将被丢弃。而读这个设备会立刻返回一个文件尾标志，所以在cp命令里可以把它用做拷贝空文件的源文件。人们经常把不想看到的输出重定向到/dev/null。

> 创建空文件的另一个办法是使用“touch <filename>”命令，它的作用是改变文件的修改时间；如果指定名字的文件不存在，就创建一个新文件。但它并不会主动把有内容的文件变成空文件。

```
$ echo do not want to see this > /dev/null
$ cp /dev/null empty_file
```

可以在/dev里找到的其他设备包括硬盘和软盘、通信端口、磁带驱动器、CD-ROM、声卡以及一些代表系统内部工作状态的设备。甚至还有一个名为/dev/zero设备，它的作用是作为内容是null字节的源文件创建零长度文件。访问其中的某些设备需要具有超级用户权限；普通用户不能通过编写程序去直接访问底层设备——比如硬盘。设备文件的名字会随系统的不同而不同。Solaris和Linux都提供了以超级用户身份运行的对设备进行管理的命令或软件，那些设备以其他

方式是无法访问的；可以由用户挂装的文件系统上提供的mount就是一个这样的命令。

在这一章里，我们将集中讨论磁盘上的文件和目录。我们将在第5章讨论另一种设备，用户的终端。

3.2 系统调用和设备驱动程序

用很少数量的函数我们就可以对文件和设备进行访问和控制。这些函数被称为系统调用，是由UNIX（和Linux）直接提供的，是通向操作系统本身的操作接口。

操作系统的核心部分，即内核，是一系列设备驱动程序。这是一些对系统硬件进行操控的底层接口，我们将在第21章对设备驱动程序做深入的研究。举例来说，磁带机就有一个与之对应的设备驱动程序，它知道如何启动磁带、如何对它前后回绕、如何对它进行读写，等等。它还知道磁带必须以固定长度的数据块为单位进行读写。因为磁带上的数据读写操作完全是线性的，所以该驱动程序并不能直接访问磁带上的数据块，必须先把它回绕到正确的位置才行。

类似地，一个底层的硬盘设备驱动程序一次必须读写一整块硬盘扇区，但能够直接存取硬盘上任意位置上的数据块，因为硬盘是一种随机存取设备。

为了向用户提供一个统一的操作界面，设备驱动程序封装了所有与硬件直接相关的功能操作。人们一般通过ioctl调用来完成典型的硬件功能。

/dev中的设备文件其用法都是一致的，它们都可以被打开、读、写和关闭。举例来说，用来访问正常文件的open调用同样可以用来访问一台用户终端、一台打印机或一台磁带机。

用来访问设备驱动程序的底层函数，即系统调用，包括：

- open　　打开一个文件或设备。
- read　　从一个打开的文件或设备里读数据。
- write　　写入一个文件或设备。
- close　　关闭一个文件或设备。
- ioctl　　把控制信息传递到设备驱动程序。

系统调用ioctl被用来提供一些与特定硬件设备有关的必要控制（与正常输入输出相对应），所以它的用法随设备的不同而不同。举例来说，一个ioctl调用对磁带机来说是回绕，对串行口来说就是设置数据流控制字符。由于这个原因，各种计算机上的ioctl倒不必具备什么可移植性。此外，每个驱动程序都有它自己的一套ioctl命令。

这些调用和其他的系统调用的文档一般被放在UNIX操作系统man命令给出的使用手册的第二小节。语法模型提供了系统调用的参数清单及其函数返回类型，而相关的“# define”定义了有关的常数，它们都由include语句中的头文件提供。每个系统调用特有的特点包括在各个调用的说明里。

3.3 库函数

在输入输出操作中直接使用底层的系统调用会产生这样一个问题，即它们的效率可能非常之低。为什么呢？

- 系统调用会降低计算机的执行性能。这是因为UNIX没有一个在运行用户程序代码和执行

它内核代码两种情况之间来回切换的开关，因此系统调用与函数调用相比开销要大一些。

- 硬件在技术和工艺上有一定的条条框框，这会给底层系统调用一次能够读写的数据块加上限制。举个例子，磁带机的写操作一般都要求达到其最小数据块长度，比如说是10k；如果你写的数据量少于这个数字，磁带机还是会以10k为单位卷绕磁带，这就在磁带上留下了空隙。

为了给设备和磁盘文件提供更高层的操作界面，UNIX操作系统提供了一系列的标准函数库。它们是一些由函数构成的集合，用户可以把它们包括在自己的程序里去处理那些与设备和文件有关的问题。带有输出缓冲功能的标准I/O库就是一个这样的例子。用户可以高效率地写任意长度的数据块，库函数则在数据满足数据块长度要求的时候安排执行底层系统调用。这就大大降低了系统调用的负面影响。

库函数的文档一般被放在UNIX操作系统man命令给出的使用手册的第三小节，并且往往会有一个与之对应的头文件，比如与标准I/O库对应的stdio.h文件。

下面这个图是对前几小节讨论的总结，图3-2的UNIX系统显示了各种文件函数与用户、设备驱动程序、内核和硬件之间的关系。

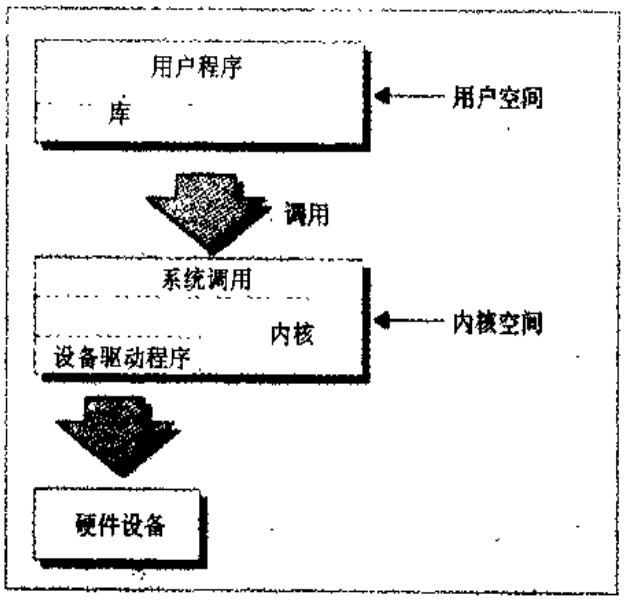


图 3-2

3.4 文件的底层访问

一个运行中的程序被称为一个进程，它有一些与之关联的文件描述符。这是一些小数值整数，用户可以通过它们访问打开的文件和设备。可用文件描述符的数量取决于UNIX系统的配置情况。在一个程序开始运行的时候，这些文件描述符里一般会有三个是已经为它打开了的。它们是：

- 0 标准输入。
- 1 标准输出。
- 2 标准错误。

用户可以通过系统调用open把其他文件描述符与文件和设备关联在一起，我们马上就要介绍到这个调用。即使只有自动打开的文件描述符，就已经足以让我们利用write编写出一些简单的程序了。

3.4.1 write系统调用

```
#include <unistd.h>

size_t write(int fildes, const void *buf, size_t nbytes);
```

系统调用write的作用是把缓冲区buf里的前nbytes个字节写入与文件描述符fildes相关联的文件中去。它的返回值是实际写出的字节数。如果文件描述符有错，或者底层设备驱动程序对数据块尺寸比较敏感，该返回值就可能会小于nbytes的值。如果这个函数的返回值是0，就表示没有写出任何数据；如果是-1，就表示在write调用中出现了错误，对应的错误代码保存在全局变量errno里面。

有了这些知识，我们来编写第一个C语言程序simple_write.c，如下所示：

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

int main()
{
    if ((write(1, "Here is some data\n", 18)) != 18)
        write(2, "A write error has occurred on file descriptor 1\n",46);

    exit(0);
}
```

这个程序很简单，只是在标准输出上显示一条消息而已。当一个程序退出运行的时候，所有已经打开的文件描述符都会自动关闭，不需要由用户来明确地关闭它们。如果我们是在处理被缓冲的输出，情况就不一样了。

```
$ simple_write
Here is some data
$
```

有一个不太重要的问题，就是write会报告说它写的字节比用户要求的少。这并不一定是个错误。用户应该在自己的程序里检查errno看看是否出现了错误，然后再次调用write写出其余的数据。

本章中的所有例子都假设当前子目录已经被放在读者的PATH变量里了，另外，读者没有在自己是超级用户时运行这些程序。如果没有把当前子目录放到PATH变量里去（超级用户绝不应该这样做），你可以象下面这样给出当前子目录来运行这些个程序：

```
$ ./simple_write
```

3.4.2 read系统调用

```
#include <unistd.h>

size_t read(int fildes, const void *buf, size_t nbytes);
```

系统调用read的作用是从与文件描述符fildes相关联的文件里读入nbytes个字节的数据并把它

们放到数据区buf里去。它的返回值是实际读入的字节数，它可能会小于nbytes的值。如果这个函数的返回值是0，就表示没有读入任何数据；它到达了文件尾。同样，如果是-1，就表示在read调用中出现了错误。

下面这个simple_read.c程序把标准输入的前128个字节拷贝到标准输出去。如果输入少于128个字节，就把它们全体拷贝过去。

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

int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1,buffer,nread)) != nread)
        write(2, "A write error has occurred\n",27);

    exit(0);
}
```

运行这个程序，我们会看到：

```
$ echo hello there | simple_read
hello there
$ simple_read < draft1.txt
Files

In this chapter we will be looking at files and directories and how to manipulate
them. We will learn how to create files, oS
```

请注意，下一个shell提示符出现在输出数据最后一行的尾部，这是因为例子里的128字节没有构成完整的两行（最后一个字符不是换行符）。

3.4.3 open系统调用

创建新的文件描述符需要使用系统调用open。

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

严格地说，在符合POSIX规范的系统上，在用到open的时候并不是非得把sys/types.h和sys/stat.h包括进来，但在某些UNIX系统上它们可能是必不可少的。

简单地说，open建立了一条到文件或设备的访问路径。如果操作成功，它将返回一个文件描述符，后续的read和write等系统调用就将使用该文件描述符对打开的那个文件进行操作。文件的文件描述符是独一无二的，并且不会与运行中的任何其他程序共享。如果两个程序同时去打开同一个文件，会导致两个彼此不一样的文件描述符。如果它们都对文件进行写操作，那么

它们会各写各的，分别接着上次离开的位置继续往下写。它们的数据不会交错出现交织在一起，而是彼此互相覆盖（后写入的内容覆盖掉前面写入的内容）。两个程序对文件的读写位置（偏移值）有各自的理解，各干各的。为了防止出现这种我们并不希望看的混乱冲突的局面，可以用上“文件加锁”功能，我们将在第7章里介绍它。

准备打开的文件或设备的名字被当作path参数传递到函数中去，oflags参数用来定义准备对打开的文件进行的操作动作。

oflags参数是通过把人们要求的文件访问模式与其他可选模式按位OR操作得到的。open调用必须给出表3-1所列的某个文件访问模式：

表 3-1

模 式	说 明
O_RDONLY	以只读方式打开
O_WRONLY	以只写方式打开
O_RDWR	以读写方式打开

open调用还可以在oflags参数中包括下列可选模式的（按位或操作OR）组合：

- O_APPEND　把写入数据追加在文件的末尾。
- O_TRUNC　把文件长度设置为零，丢弃其中现有的内容。
- O_CREAT　按mode中给出的访问模式创建文件。
- O_EXCL　与O_CREAT一起使用，确保调用者创建出文件来。open是一个最小规模的操作，也就是说，它只完成一个函数调用。而加上这个可选模式可以防止两个程序同时创建一个文件情况的出现。如果文件已经存在，open操作将失败。

其他可能出现的oflags值请参考open调用的使用手册页文档，它们出现在该文档的第二小节（用“man open 2”命令查看）。

如果open成功，它会返回一个新的文件描述符（文件描述符永远是一个非负整数）；如果失败，就返回“-1”，并对全局性的errno变量进行设置以指明失败的原因。我们将在下一小节对errno做进一步讨论。新文件描述符永远取未用描述符的最小值，这在某些情况下是非常有用的。比如说，如果一个程序关闭了自己的标准输出，然后再次调用open，就会重新使用“1”作为文件描述符，而标准输出将被重定向到另外一个文件或设备。

POSIX技术规范还标准化了一个creat调用，但它并不常用。这个调用不仅会创建文件，还会打开它——它的作用相当于以“O_CREAT | O_WRONLY | O_TRUNC”为oflags标志的open调用。

3.4.4 访问权限的初始化值

当我们用open加O_CREAT标志来创建一个文件的时候，必须使用open调用的三个参数格式。第三个参数mode是几个标志按位OR操作后得到的，这些标志是在头文件/sys/stat.h里定义的，它们是：

- S_IRUSR 读权限，文件属主。
- S_IWUSR写权限，文件属主。
- S_IXUSR 执行权限，文件属主。
- S_IRGRP 读权限，文件所在分组。
- S_IWGRP写权限，文件所在分组。
- S_IXGRP 执行权限，文件所在分组。
- S_IROTH 读权限，其他用户。
- S_IWOTH写权限，其他用户。
- S_IXOTH 执行权限，其他用户。

请看下面的例子：

```
open ("myfile", O_CREAT, S_IRUSR | S_IXOTH);
```

它的作用是创建一个名为myfile的文件，文件属主拥有它的读操作权限，其他用户拥有它的执行权限。也只有这么多权限。

```
$ ls -ls myfile
0 -r--------x   1 neil     software        0 Sep 22 08:11 myfile*
```

有几个因素会对文件的访问权限产生影响。首先，只有文件被创建出来以后才能谈及访问权限。第二，用户掩码（“user mask”，由shell的umask命令设定）会影响到被创建文件的访问权限。open调用里给出的模式值将在程序运行时与用户掩码的反值做AND操作。举例来说，如果用户掩码被设置为“001”，open调用给出了S_IXOTH模式标志，那么文件不会被创建为其他用户拥有执行权限的情况。这是因为用户掩码中规定了不允许向其他用户提供执行权限。因此，可以说open和creat调用中的标志实际上是设置权限的申请，所申请的权限是否会被设置还要取决于umask在程序运行时取的值。

3.4.5 umask变量

umask是一个系统变量，它的作用是为文件的访问权限设定一个掩码，再把这个掩码用在文件创建操作中。执行umask命令可以对这个变量进行修改，给它提供一个新值。这是一个由三个八进制数字组成的值。各数字都是八进制值1、2、4的AND操作结果。这三个数字分别对应着用户（user）、分组（group）和其他用户（other）的访问权限。请看表3-2：

表 3-2

数 字	取 值	含 义
1	0	不禁止任何属主权限
	4	禁止属主的读权限
	2	禁止属主的写权限
	1	禁止属主的执行权限
2	0	不禁止任何分组权限
	4	禁止分组的读权限
	2	禁止分组的写权限
	1	禁止分组的执行权限

（续）

数 字	取 值	含 义
3	0	不禁止任何其他用户权限
	4	禁止其他用户的读权限
	2	禁止其他用户的写权限
	1	禁止其他用户的执行权限

表3-3给出的是禁止了分组写和执行权限、禁止了其他用户写权限情况下的umask掩码：

表 3-3

位 置	取 值
1	0
2	2
	1
3	2

各位上数字的值将AND在一起；因此第2位数字的值是“2 & 1”，结果为“3”。最终的umask结果是“032”。

当我们通过一个open或creat调用创建一个文件的时候，mode参数将与umask进行比较。mode参数中被置位了的位如果在umask中也被置位了，就会被排除在访问权限的构成之外。打个比方，这样做的最终结果是用户可以设置自己的环境说“不准创建允许其他用户有写权限的文件，即使创建该文件的程序提出申请也不行。”但这样做并不会影响某个程序或用户在今后使用chmod命令（或者在程序中使用chmod系统调用）添加其他的写权限，它确实能够帮助保护用户的利益，让他们不必费心去检查和设置每一个新文件的访问权限。

3.4.6 close系统调用

```
# include <unistd.h>

int close(int fildes);
```

我们用close调用终止一个文件描述符fildes与它文件之间的关联。文件描述符被释放并能够重新使用。如果close操作成功就返回“0”，如果出错就返回“-1”。注意，检查close调用的返回结果有时十分重要。有的文件系统，特别是网络上的文件系统，可能不会在关闭文件之前报告文件写操作中出现的错误。

对一个运行中的程序来说，能够让它一次打开的文件个数是有上限的。这个限制由文件limits.h中的OPEN_MAX常数定义，会随着系统的不同而不同；但POSIX技术规范里要求它至少要等于16个。单就这个上限来说，它还会受到本地计算机上系统全局性限制的影响。

3.4.7 ioctl系统调用

```
# include <unistd.h>

int ioctl(int fildes, int cmd, . . . );
```

ioctl调用有点像是个大麻袋。它提供了一个对设备行为、设备描述符、设备底层服务的配置工作等方面进行控制的操作界面。终端、文件描述符、套接字、甚至磁带机都可以有为它们定义的ioctld，具体细节需要查阅各设备通过man命令提供使用手册页。POSIX技术规范只为流式数据定义了ioctl调用，对它的讨论超出了本书的讨论范围。

ioctl对描述符fildes指定的对象执行在cmd中给出的操作。根据特定设备所支持的函数的不同，还可能会有一个可选的第三参数。

动手试试：一个文件拷贝程序

在学习了足够多的关于open、read和write系统调用的知识以后，我们来编写一个底层程序copy_system.c，它的作用是逐个字符地把一个文件拷贝为另外一个文件。

在这一章里，我们将采用多种办法完成这一工作，目的就是比较各种方法的执行效率。为简单起见，我们将假设输入文件已经存在，输出文件不存在，并且所有的读写操作都成功了。当然，在实际程序里，我们必须对这些假设是否成立进行检验！

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main()
{
    char c;
    int in, out;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while(read(in,&c,1) == 1)
        write(out,&c,1);

    exit(0);
}
```

注意 语句“include <unistd.h>”必须最早出现，因为它根据POSIX技术规范定义的标志将影响到其他的头文件。

首先，我们需要有一个测试用的输入文件，长度大概是1MB字节，取名为file.in。

运行这个程序，大致会看到像下面这样的结果：

```
$ time copy_system
4.67user 146.90system 2:32.57elapsed 99%CPU
...
$ ls -l file.in file.out
1029 -rw-r---r-   1 neil     users     1048576 Sep 17 10:46 file.in
1029 -rw-------   1 neil     users     1048576 Sep 17 10:51 file.out
```

我们在这里用UNIX提供的time工具对这个程序花费的时间进行了测算。可以看到，1MB长的输入文件file.in被成功地拷贝到file.out，而后者是以只允许属主进行读写的权限创建出来的。这次拷贝花费了大约两分半钟，并且几乎占用了所有的CPU时间。之所以这么慢是因为它必须完成超过两百万次的系统调用。

我们采用大一些的数据块进行拷贝会改善这一点，请看下面这个改进后的程序copy_block.c，它每次拷贝长度为1K的数据块，用的还是系统调用：

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main()
{
    char block[1024];
    int in, out;
    int nread;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while((nread = read(in,block,sizeof(block))) > 0)
        write(out,block,nread);

    exit(0);
}
```

现在来运行这个程序，先删除旧的输出文件：

```
$ rm file.out
$ time copy_block
0.01user 1.09system 0:01.90elapsed 57%CPU
...
$ ls -ls file.in file.out
1029 -rw-r--r--   1 neil     users     1048576 Sep 17 10:46 file.in
1029 -rw-------   1 neil     users     1048576 Sep 17 10:57 file.out
```

改进后的程序只花费了不到两秒种的时间，因为它只需做大约2000次系统调用就足够了。当然，这些时间与系统本身的性能有很大的关系，但它们确实已经让我们看出系统调用需要巨大的开支，值得对它们的用法进行优化。

3.4.8 其他与文件管理有关的系统调用

还有许多其他的系统调用能够对这些底层文件描述符进行操作。它们允许程序对文件的使用方式和返回的状态信息进行控制。我们在这里简单介绍下，以便你可以使用它们。但读者不一定一眼就喜欢上它们。

1. lseek系统调用

```
#include <unistd.h>
#include <sys/types.h>

off_t lseek(int fildes, off_t offset, int whence);
```

lseek系统调用对文件描述符fides的读写指针进行设置，你可以用它设置文件的下一个读写位置。既可以把指针设置到文件中的某个绝对位置，也可以把它设置到相对于当前位置或文件尾的某个相对位置。offset参数用来指定位置，而whence参数定义该偏移值的用法。whence可以是下列取值之一：

- SEEK_SET　　offset是一个绝对位置。
- SEEK_CUR　　offset是从当前位置算起的一个相对位置。

• SEEK_END offset是从文件尾算起的一个相对位置。

lseek的返回值是从文件头到文件指针被设置处的字节偏移值；操作失败时返回“-1”。指针移动操作中偏移值offset的类型off_t是一个与操作系统具体实现有关的类型，它的定义在文件sys/types.h里。

2. fstat、stat和lstat系统调用

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

注意 加上sys/types.h文件的包括操作被认为是“可选但又明智的”。

fstat系统调用返回的是与一个打开的文件描述符关联着的文件的状态信息。这些信息将被写到一个buf结构里，其地址作为一个参数被传递过去。

同一系列的stat和lstat函数返回的是通过文件名查到的文件状态信息。它们的结果基本一致，但在文件是一个符号链接的情况下，lstat返回的是该符号链接本身的信息，而stat返回的是该链接指向的文件的信息。

stat结构的构成在不同UNIX系统上会有所变化，但肯定会有表3-4所示这些内容：

表 3-4

stat数据项	说 明
st_mode	文件权限和文件类型信息
st_ino	与该文件关联的inode
st_dev	文件保存在其上的设备
st_uid	文件属主的用户身份标识
st_gid	文件属主的分组身份标识
st_atime	上次被访问时间
st_ctime	文件权限、属主、分组或内容方面的上次被修改时间
st_mtime	文件内容方面的上次被修改时间
st_nlink	该文件上硬链接的个数

stat结构中返回的st_mode标志还有一些与之关联的宏定义，它们是在头文件sys/stat.h里被定义的。其中包括对访问权限、文件类型标志以及部分掩码的定义，它们能够帮助我们对特定的类型和权限进行测试。

访问权限标志与刚才在open系统调用里介绍过的内容是一致的。文件类型标志包括：

• S_IFBLK 文件是一个特殊的块设备。
• S_IFDIR 文件是一个子目录。
• S_IFCHR 文件是一个特殊的字符设备。
• S_IFIFO 文件是一个FIFO设备（有名字的管道）。
• S_IFREG 文件是一个普通文件。
• S_IFLNK 文件是一个符号链接。

其他模式标志包括：

- S_ISUID 文件在执行时置位了setUID位。
- S_ISGID 文件在执行时置位了setGID位。

用来解释st_mode标志的掩码包括：

- S_IFMT 文件类型。
- S_IRWXU 属主的读/写/执行权限。
- S_IRWXG 分组的读/写/执行权限。
- S_IRWXO 其他用户的读/写/执行权限。

还有一些用来帮助确定文件类型的宏定义。它们对经过掩码处理的模式标志和适当的设备类型标志进行比较。它们包括：

- S_ISBLK 测试是否是特殊的块设备文件。
- S_ISCHR 测试是否是特殊的字符设备文件。
- S_ISDIR 测试是否是子目录。
- S_ISFIFO 测试是否是FIFO设备。
- S_ISREG 测试是否是普通文件。
- S_ISLNK 测试是否是符号链接。

举例来说，如果想对“文件代表的不是一个子目录、设置了属主的执行权限、不再有其他权限”这种情况进行测试，我们可以使用下面的代码：

```
struct stat statbuf;
mode_t modes;

stat("filename",&statbuf);
modes = statbuf.st_mode;

if(!S_ISDIR(modes) && (modes & S_IRWXU) == S_IXUSR)
    ...
```

3. dup和dup2系统调用

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

dup系统调用提供了复制文件描述符的一个办法，使人们能够通过两个或者更多个不同的描述符来访问同一个文件，在文件的不同位置对数据进行读写。dup系统调用对作为它参数的一个文件描述符fildes进行复制，返回一个新的描述符。dup2系统调用是明确地把一个文件描述符复制为另外一个，它必须给出一个描述符作为复制品参数。

当我们通过管道在多个进程间进行通信的时候，这些调用也很有用。我们还将在第11章对dup系统调用进行研究。

3.5 标准I/O库

标准I/O库及其头文件stdio.h提供了一个全面的面向底层I/O系统调用的操作界面。这个库现在已经成为ANSI标准下的C语言组成部分，而我们前面见到的系统调用还没有被包括在ANSI标

准之内。标准I/O库提供了许多精密复杂的函数，使我们能够设置输出数据的格式和扫描处理输入数据。它还负责协调设备的数据缓冲。

在很多方面，我们可以像使用底层文件描述符那样使用这个库。用户需要先打开一个文件以建立一个访问路径。这个操作将返回一个做为其他I/O库函数参数的值。与底层的文件描述符相对应的事物叫做“流”（stream），它被实现为指向一个结构的指针，即一个“FILE *”。

注意 不要把这里的文件流与C++语言中的iostreams（输入输出流）或者AT&T UNIX System V Release 3中引入的进程间通信用的STREAMS模型混为一谈，进程间通信方面的问题已经超出本书的讨论范围了。如果读者打算进一步了解STREAMS的情况，请自行查阅X/Open技术规范和随System V版本一起提供的“AT&T STREAMS Programming Guide”（《AT&T STREAMS程序设计指南》）。

在一个程序开始运行的时候，有三个文件流是自动打开好了的。它们是stdin、stdout和stderr。它们都是在stdio.h文件里定义的，分别代表着标准输入、标准输出和标准错误输出，即分别对应着底层文件描述符0、1和2。

在接下来的内容里，我们将学习标准I/O库中的下列库函数：

- fopen和fclose。
- fread和fwrite。
- fflush。
- fseek。
- fgetc、getc和getchar。
- fputc、putc和putchar。
- fgets和gets。
- printf、fprintf和sprintf。
- scanf、fscanf和sscanf。

3.5.1 fopen函数

```
# include <stdio.h>
FILE *fopen(const char *filename, const char *moed);
```

fopen库函数的作用相当于底层的open系统调用。它主要用于文件和终端的输入输出方面。使用这个函数的前提条件是用户对准备读写的设备有明确的控制权，但这要比底层系统调用要方便多了，因为它们把用户可能不太喜欢的辅助操作（比如输入输出的缓冲安排）交给函数库去负责了。

fopen打开由filename参数给出名字的文件，并把它与一个文件流关联在一起。mode参数规定了文件的打开方式，它是下列字符串中的一个：

- "r"或"rb" 以只读方式打开。
- "w"或"wb" 以写方式打开，并把文件长度截短为零。
- "a"或"ab" 以写方式打开，新内容追加在文件尾。

- "r+"或"rb+"或"r+b" 以修改方式打开（读和写）。
- "w+"或"wb+"或"w+b" 以修改方式打开，并把文件长度截短为零。
- "a+"或"ab+"或"a+b" 以修改方式打开，新内容追加在文件尾。

字母“b”表示文件是一个二进制文件而不是一个文本文件。请注意，UNIX与DOS的一个不同之处就在于它并不明确区分文本文件和二进制文件。它对文件一视同仁，把它们都看做是二进制文件。另一个需要重点注意的地方是mode参数必须是一个字符串，不是一个字符。所以永远要使用"r"而不是'r'。

如果操作成功，fopen返回一个非空的“FILE *”指针。如果失败，它返回NULL空值，NULL空值的定义也在stdio.h文件里。

3.5.2 fread函数

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

fread库函数的作用是从一个文件流里读取数据。数据从文件流stream被读到由ptr指定的数据缓冲区里。fread和fwrite都是对数据记录进行操作的，size参数指定每个数据记录的长度，计数器nitems给出将要传输的记录个数。它的返回值是成功地读到数据缓冲区里去的记录个数（而不是字节数）。当接近文件尾的时候，它的返回值可能会小于nitems，甚至可以是零。对所有向缓冲区里写数据的标准I/O函数来说，为数据分配空间和检查有无出错的工作是由程序员负责的。参见本章对ferror和feof函数的介绍。

3.5.3 fwrite函数

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

fwrite库函数的接口与fread的接口差不多。它从指定的数据缓冲区里取出数据并把它们写到输出流去。它的返回值是写操作成功的记录个数。

> 请注意，我们不推荐把fread和fwrite用在使用了结构化数据的场合。问题的部分原因是用fwrite写的文件在不同的计算机之间可能不具备可移植性。我们将在附录A里对可移植性做进一步研究。

3.5.4 fclose函数

```
#include <stdio.h>
int fclose(FILE *stream);
```

fclose库函数关闭指定的文件流stream，使所有尚未写出的数据都写到文件里去。因为stdio函数库会对数据进行缓冲，所以使用fclose是很重要的。如果程序需要确保数据已经全部写入文件，就必须调用fclose函数。但是，当程序正常结束时，会自动对所有还打开着的文件流调用

fclose关闭它们，可这样做就没有机会检查由fclose报告出来的错误了。可用文件流的个数是有上限的，就像文件描述符的个数也有个上限一样。这个上限值叫做FOPEN_MAX，也在stdio.h文件里定义，最少要设置为8。

3.5.5 fflush函数

```
#include <stdio.h>
int fflush(FILE *stream);
```

fflush库函数的作用是把文件流里的现有数据立刻写入文件。举个例子，你可以用这个函数来保证在试图读入一个用户响应之前先送出了一个交互操作的提示符。如果想在程序继续执行之前确保重要的数据已经被写到磁盘上，也可以使用这个函数。在调试某个程序的时候，你还可以用它来核查程序是正在写数据还是被挂起了。调用fclose函数隐含着执行一次“立刻写”操作，所以我们不必在fclose的前面调用fflush函数。

3.5.6 fseek函数

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

fseek库函数是lseek系统调用的文件流对应版。它在文件流里为下一次读或写操作设置偏移位置。offset和whence参数的含义和取值与前面介绍lseek时给出的内容完全一样。但lseek返回的是一个off_t数值，而fseek返回的是一个整数：“0”表示操作成功；“-1”表示失败并设置errno指出错误的类型。这就是人们说的标准化！

3.5.7 fgetc、getc、getchar函数

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar();
```

fgetc库函数从文件流里取出下一个字节并把它当做一个字符返回。当它到达文件尾或出现错误时，它返回EOF。用户必须通过ferror或feof来区分这两种情况。

getc函数的作用相当于fgetc，但我们可以在stream参数不允许有副作用的情况下（即它既不能影响非本地的局部变量也不能影响将做为参数传递到函数里去的变量时）把getc实现为一个宏。当然，在这种情况下，用户也就不能使用getc的地址做为一个函数指针。

getchar函数相当于getc（stdin），它从标准输入里读取下一个字符。

3.5.8 fputc、putc、putchat函数

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

fputc库函数把一个字符写到一个输出文件流里去。它的返回值是它刚写的那个值，如果失败，则返回EOF。

类似于fgetc和getc之间的关系，putc函数的作用也相当于fputc，但我们可以把它实现为一个宏。

putchar函数相当于putc（c，stdout），它把一个字符写到标准输出去。需要注意的是putchar和getchar是把字符做为int值而不是char返回的。这就允许文件尾（EOF）指示器取“-1”的值，这是一个超出字符数字编码范围的值。

3.5.9　fgets、gets函数

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

fgets库函数从一个输入文件流stream里读取一个字符串。它连续不断地把读到的字符写到s指向的字符串去，直到出现下面这三种情况之一：到达换行符、已经传输了n-1个字符，或者到达文件尾。它会把换行符也传递到接收字符串里去，再加上一个表示结尾的空字节“\0”。一次调用最多传输n-1个字符，因为它必须把空字节加上以结束那个字符串，这样总数还是n个字节。

当fgets成功结束时，它返回一个指向字符串s的指针。如果文件流已经在一个文件的文件尾处，fgets会置位这个文件流的EOF指示器并返回一个空指针。如果出现读操作错误，fgets返回一个空指针并设置errno给出错误的类型。

gets函数类似于fgets，只不过前者读的是标准输入并会丢弃它所遇见的任何换行符。它在接收字符串的尾部也加上一个空字节。注意：gets对可能被传输的字符的个数并没有限制，因此它可能会溢出自己的传输缓冲区。所以应该避免使用它，尽量用fgets代替。因特网上的许多安全问题都可以追溯到出现各种缓冲区溢出现象的程序。眼前就是一个，千万要小心！

3.5.10　格式化输入和输出

如果大家用C语言编写过程序，对那些按设计格式输出数据的库函数就应该比较熟悉。这些函数包括向一个文件流输出数据的printf系列和从一个文件流读取数据的scanf系列。

1. printf、fprintf和sprintf函数

```
#include <stdio.h>

int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

printf函数家族能够对各种不同类型的参数进行格式编排和输出。参数在输出流中的表示形式是由格式参数format控制的。格式参数是一个字符串，其中包含着普通的可打印字符和我们称之为“转换控制符”的代码，转换控制符规定了其余的参数应该以何种方式被输出到何种地方。

printf函数把自己的输出送到标准输出设备中去。fprintf函数把自己的输出送到某个特定的文件流去。sprintf函数把自己的输出和一个结尾用的空字符写到做为一个参数传递过来的字符串s里去。这个字符串必须足够大，以便能够容纳所有的输出数据。printf家族还有一些其他的成员，它们以各自的方式对各自的参数进行处理。详细资料请参考printf命令的使用手册页。

普通字符被传递输出时不发生变化。转换控制符让printf取出传递过来的参数数据并对它们的格式进行编排。转换控制符永远以“%”字符打头。下面是一个简单的例子：

```
printf ("Some numbers: %d, %d, and %d\n", 1, 2, 3 );
```

它在标准输出上产生如下所示的输出：

```
Some numbers: 1, 2, and 3
```

要想输出“%”字符，就必须使用“%%”，这样就不会与格式控制符弄混了。

下面是一些常用的格式控制符：

- %d和%i　输出一个十进制整数。
- %o或%x　输出一个八进制或十六进制整数。
- %c　输出一个字符。
- %s　输出一个字符串。
- %f　输出一个（单精度）浮点数。
- %e　以科学计数法格式输出一个双精度浮点数。
- %g　以一般格式输出一个双精度浮点数。

让传递到printf函数的参数在数量和类型方面与format字符串里的转换控制符配上套是很重要的。整数参数的类型可以用一个可选的长度限定符来指定。它可以是“h”，“%hd”表示这是一个短整数（short int）；或者“l”，“%ld”表示这是一个长整数（long int）。有的编译器能够对printf语句进行检查，但这并非万无一失。如果读者使用的是GNU编译器，“gcc -Wformat”可以完成这一工作。

下面是另外一个例子：

```
char initial = 'A';
char *surname = "Matthew";
double age = 10.5;

printf("Hello Miss %c %s, aged %g\n", initial, surname, age);
```

它产生如下所示的输出：

```
Hello Miss A Matthew, aged 10.5
```

利用数据域控制符可以对数据的输出格式做进一步的控制。数据域控制符是对转换控制符的补充，能够对输出数据之间的间隔进行控制。它们的常见用法是设置浮点数的小数点位置，或者是设置字符串两端的空格个数。

数据域控制符是转换控制符里紧跟在“%”字符后面的数字。下面是一些转换控制符示例及其输出情况。为了让大家看得更清楚，我们用垂直线字符来表示输出边界（见表3-5）。

表 3-5

格 式	参 数	输 出
%10s	"Hello"	Hello
%-10s	"Hello"	Hello
%10d	1234	1234
%-10d	1234	1234
%010d	1234	0000001234
%10.4f	12.34	12.3400
%*s	10,"Hello"	Hello

上表中的示例都输出在一个10个字符宽的区域里。注意：一个负数值的数据域宽度表示数据项将在该数据域里以左对齐的格式输出。可变数据域宽度用一个星号（*）来表示。在这种情况下，下一个参数用来表示数据域的宽度。“%”字符后面的第一个“0”表示数据前面要用“0”填充。根据POSIX技术规范的要求，printf不对数据域进行截断；相反，它会扩充数据域以适应数据的宽度。因此，如果我们想打印一个长于数据域宽度的字符串，数据域会加宽。如表3-6所示：

表 3-6

格 式	参 数	输 出
%10s	"HelloTherePeeps"	HelloTherePeeps

printf函数返回一个整数，即它输出的字符个数。但sprintf的返回值没有算上最后结尾用的那个null空字符。如果发生错误，这些函数会返回一个负数值并设置errno。

2. scanf、fscanf和sscanf函数

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

scanf函数家族的工作情况与printf函数家族很相似，只是前者的作用是从一个文件流里读取数据并把数据值放到传递过来的指针参数指向的地址处的变量中去。它们也使用一个格式字符串来控制输入数据的转换操作，其工作原理和许多转换控制符都与printf系列函数方面的情况一致。

scanf函数读入的值将保存到对应的变量里去，这些变量的类型正确与否很重要，必须让它们与格式字符串精确配对。如果不是这样，内存就会发生冲突，从而使程序崩溃。编译器是不会对此做出错误提示的，但如果运气够好，用户可能会看到一个警告信息！

scanf的格式字符串里包含着普通字符和转换控制符，就像printf函数中一样。但那些普通字符用来指定在输入数据里必须出现的字符。

下面是一个简单的例子：

```
int num;
scanf("Hello %d", &num);
```

这个scanf函数的调用只有在这种情况下才能成功：标准输入上接下来的五个字符要匹配上“Hello”；随后的一些字符还必须构成一个可识别的十进制数字。数字将被读入并赋值给变量num。格式字符串中的那个空格表示忽略输入数据中转换控制符之间的各种空白字符（空格、制表符、换页符、换行符等）。这就是说，在下面两种输入情况下，这个scanf调用都会执行成功并把1234放到变量num里去：

```
Hello     1234
Hello1234
```

输入里的空白字符在进行数据转换时一般也会被忽略。也就是说，格式字符串“%d”将持续读取输入，跳过空格和换行符，直到找到一组数字为止。如果预期的字符没有在输入里出现，转换工作失败，scanf也将返回。如果不注意，就会因此导致大问题。如果用户在输入中应该出现一个整数的地方放的是一个非数字字符，就可能在自己的程序里弄出一个无限循环来。

下面是另外一些转换控制符：

- %d　　读取一个十进制整数。
- %o或%x　　读取一个八进制或十六进制整数。
- %f、%e、%g　　读取一个浮点数。
- %c　　读取一个字符。
- %s　　读取一个字符串。
- %[ ]　　读取一个字符集合（见下面的说明）。
- %%　　读取一个“%”字符。

类似于printf，在scanf的转换控制符里也可以加上对输入数据域宽度的限制。长度限定符（“h”对应于短整数，“l”对应于长整数）指明接收参数能否比缺省情况更短或更长。也就是说，“hd”表示要读入一个短整数；“%ld”表示要读入一个长整数；而“%lg”表示要读入一个双精度浮点数。

以星号（*）打头的控制符表示对应位置上的输入数据可以被忽略，不会被写到接收参数里去。

我们用“%c”控制符从输入中读取一个字符。这个控制符不跳过起始的空白字符。

我们用“%s”控制符扫描字符串，但我们必须小心从事。它会跳过起始的空白字符，但会在字符串里出现的第一个空白字符处停下来，所以我们最好还是用它来读取单词而不是一般意义上的字符串。此外，如果没有数据域宽度限定符，它能够读取的字符串的长度是没有限制的，所以接收字符串必须有足够的空间来容纳输入流里最长的字符串。最好是使用一个数据域宽度限定符，或者混合使用fgets和sscanf先读入一行输入数据再对它进行扫描。

我们使用“%[ ]”控制符读取一个由一个字符集合中的字符构成的字符串。格式字符串“%[A-Z]”将读取一个由大写字母构成的字符串。如果控制符中字符集合里的第一个字符是上箭头字符“^”，就表示将读取一个由不属于该字符集合的字符构成的字符串。因此，如果想读取一个其中带空格的字符串，但想让它停在第一个逗号处，我们就可以使用控制符“%[^,]”。

给定下面的输入行：

```
Hello, 1234, 5.768, X, string to the end of this line
```

下面的scanf函数调用能够正确地读入四个数据项：

```
char s[256];
int n;
float f;
char c;

scanf("Hello,%d,%g, %c, %[^\n]", &n,&f,&c,s);
```

scanf的返回值是它成功读取的数据项个数，如果在读第一个数据项的时候就失败了，返回值将是零。如果在读第一个数据项的时候就已经到达了输入的结尾，就会返回EOF。如果文件流上发生了读操作错误，就会置位文件流的出错标志，而错误变量errno将被设置为指明错误类型的代码。详细情况请参考后面介绍文件流错误处理的内容。

一般说来，对scanf及其伙伴的评价并不高，这主要有三方面原因：

- 从它们的历史来看，它们的具体实现都比较笨重。
- 它们的用法不够灵活。
- 从用它们写的代码上不容易看出它们要分析读取的东西是什么。

尽量使用其他函数，比方说，可以用fread或fgets读取输入行，再用字符串函数把输入分断成我们需要的数据项。

3.5.11 对数据流进行处理的其他函数

stdio函数库里还有一些其他的函数，它们或者使用文件流参数，或者使用标准的stdin、stdout和stderr流进行操作：

- fgetpos　获得文件流的当前读写位置。
- fsetpos　设置文件流的当前读写位置。
- ftell　返回文件流当前读写位置的偏移值。
- rewind　重置文件流里的文件读写位置。
- freopen　重新使用一个文件流。
- setvbuf　为文件流设置缓冲策略。
- remove　相当于unlink函数；但如果它的path参数是一个子目录的话，其作用就相当于rmdir函数。

这些都是在UNIX操作系统man使用手册第三节文档中介绍的库函数。

我们用文件流函数重写一遍前面那个文件拷贝程序，这次我们使用库函数。请看下面的copy_stdio.c程序。

动手试试：又一个文件拷贝程序

这个程序与前面的版本很相似，但逐字符的拷贝工作改为通过调用stdio.h文件里定义的函数来完成。

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

int main()
```

```
{
    int c;
    FILE *in, *out;

    in = fopen("file.in","r");
    out = fopen("file.out","w");

    while((c = fgetc(in)) != EOF)
        fputc(c,out);

    exit(0);
}
```

像前面那样运行这个程序，我们得到的结果是：

```
$ time copy_stdio
1.69user 0.78system 0:03.70elapsed 66%CPU
```

这一次，程序运行了3.7秒种，不如底层系统调用的数据块拷贝程序快，但与那个一次拷贝一个字符的版本相比可快得太多了。这是因为stdio库在FILE结构里使用了一个内部的缓冲区，只有在缓冲区填满的时候才进行底层系统调用。请读者利用stdio库函数自行编写出实现逐行拷贝和数据块拷贝两种功能的程序，看它们与我们在这一章里给出的三个程序示例相比性能到底如何。

3.5.12 文件流错误处理

为了表明有一个错误，许多stdio库函数会返回一个超范围的值，比如空指针或EOF常数等。在这些情况下，错误类型由外部变量errno指出。它的语法是：

```
# include <errno.h>
extern int errno;
```

> 注意：有许多函数可以改变errno的值。这个变量的值只有在函数操作失败的情况下才有意义。你必须在函数表明操作失败之后立刻对它进行检查。在使用检查它之前，永远应该先把它拷贝到另外一个变量里去，因为像fprintf等这样的输出函数自己就会改变errno的值。

我们也可以通过检查文件流的状态来确定是否发生了错误，是否到达了文件尾。

```
#include <stdio.h>

int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
```

ferror库函数对文件流的出错指示器进行测试，如果它被置位，就返回非零值；否则返回零。

feof库函数在一个文件流里对它的文件尾指示器进行测试，如果它被置位，就返回非零值；否则返回零。我们可以像下面这样使用它：

```
if (feof(some_stream))
    /* We're at the end */
```

clearerr库函数的作用是重置由stream指定的文件流的文件尾指示器和出错指示器。它没有返

回值，也没有定义任何错误。我们可以通过它从文件流上的错误情况中恢复过来。比如说，在磁盘数据满错误解决之后重新开始写文件流就是一个这样的例子。

3.5.13 文件流和文件描述符的关系

每个文件流都和一个底层文件描述符相互关联着。我们完全可以把底层的输入输出操作与高层的文件流操作混在一起使用，但一般说来这并不是个聪明的办法，因为它会使数据缓冲的后果难以预料。

```
#include <stdio.h>

int fileno(FILE *stream);
FILE *fdopen(int fildes, const char *mode);
```

我们可以通过调用fileno函数确定文件流使用的是哪个底层文件描述符。它返回的是给定文件流使用的文件描述符，“-1”表示调用失败。在需要对一个已经打开的文件流进行底层访问操作的时候（比如说想对它调用fstat的时候）这个函数还是很有用的。

我们还可以通过调用fdopen函数在一个已经打开的文件描述符的基础上创建一个新的文件流。这个函数的基本作用是为一个已经打开的文件描述符提供一个stdio缓冲区，这样对它的读写操作可能比较容易进行一些。

fdopen函数的执行情况与fopen函数是一样的，只是前者的参数不是一个文件名而是一个底层的文件描述符。在我们已经通过open系统调用创建了一个文件——可能是出于为了更好地控制其访问权限的目的，但又想通过文件流来对它进行读写操作，这时就用得着这个函数了。fdopen函数的mode参数与fopen函数使用的完全一样，同时还必须符合该文件在最初创建时所设定的权限配置情况。fdopen返回一个新的文件流，“NULL”表示调用失败。

3.6 文件和子目录的维护

各种标准库和系统调用为文件和子目录的创建与维护提供了全面的控制。

3.6.1 chmod系统调用

我们可以通过chmod系统调用对文件或子目录的访问权限进行修改。它构成了shell程序chmod的基础。

```
#include <sys/stat.h>

int chmod (const char *path, mode_t mode );
```

path参数指定的文件在函数调用结束后被修改为具有mode参数给出的访问权限。参数mode的设置类似于open系统调用中的做法，也是对所要求的访问权限进行按位OR操作。如果没有给这个程序以适当的优先权，就只有文件的属主或超级用户才能修改它的访问权限。

3.6.2 chown系统调用

超级用户可以使用chown系统调用改变一个文件的属主。

```
# include <unistd.h>
int chown (const char *path, uid_t owner, gid_t group );
```

这个调用要用到用户ID和分组ID（通过getuid和getgid调用获得）的数字值和一个用来限定谁可以修改文件访问权限的常数。如果已经设置了适当的优先权，文件的属主和所属分组就会发生变化。

POSIX技术规范还是给“非超级用户也能改变文件访问权限”的系统留了一个余地。虽然一切“正常的”POSIX系统都不允许这样做，但严格地说，这是它的一个补充规定（FIPS 151-2）里要求的。我们在这本书里讨论的系统其类型都符合XSI（X/Open System Interface，X/Open系统操作接口）原则，并且它们都强调遵守文件的所有权规则。

3.6.3 unlink、link、symlink系统调用

我们可以用unlink系统调用来删除一个文件。

```
#include <unistd.h>

int unlink(const char *path);
int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

unlink系统调用的作用是通过减少指定文件上的链接计数到达删除目录数据项的目的。如果对链接的递减操作执行成功，它返回“0”；如果发生错误就返回“-1”。如果你想通过调用这个函数删除子目录中某个文件所对应的目录数据项，就必须拥有这个子目录的写和执行权限。

如果某个文件上的链接计数减少到零，并且没有进程打开并使用着它，这个文件就会被删除掉。在实际操作中，目录数据项肯定是被删除掉了，但文件占用空间还要等最后一个进程（如果有的话）关闭它之后才会被系统回收。rm程序使用的就是这个调用。文件上额外的链接表示这个文件还有其他名字，这通常是由ln程序创建的。如果我们想通过编程来创建某个文件的新链接，可以使用link系统调用。

先用open创建一个文件，然后对它调用unlink是某些程序员用来创建瞬时文件的技巧。这些文件只有在被打开的时候才能被程序使用，当程序退出或关闭这些文件的时候它们就会被自动地删除掉。

link系统调用将创建一个指向由path1指定的现有文件的新链接。新目录数据项由path2给出。我们可以通过symlink系统调用以类似的方式创建新的符号链接。注意：一个文件的符号链接不会像正常（硬）链接那样能够防止该文件被删除掉。

3.6.4 mkdir和rmdir系统调用

我们可以通过mkdir和rmdir系统调用建立和删除子目录。

```
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode );
```

mkdir系统调用的作用是创建一个子目录，它相当于mkdir程序。mkdir调用将使用path做为新建子目录的名字。子目录的权限由参数mode设定，mode的内容将按open系统调用的

O_CREAT选项中的有关定义设置，当然，还要看umask的设置情况是怎样的才行。

```
#include <unistd.h>
int rmdir (const char *path );
```

rmdir系统调用的作用是删除子目录，但只有在子目录中没有文件的情况下才行。rmdir程序就是用这个系统调用来完成工作的。

3.6.5 chdir系统调用和getcwd函数

一个程序，它完全可以像一个用户在UNIX文件系统里漫游那样对子目录进行切换。我们在shell里通过cd命令切换子目录，程序则可以使用chdir系统调用。

```
# include <unistd.h>
int chdir (const char *path );
```

一个程序可以通过调用getcwd函数确定自己的当前工作子目录。

```
# include <unistd.h>
int *getcwd (char *buf, size_t size );
```

getcwd函数的作用是把当前子目录的名字写到给定的缓冲区buf里。如果子目录的名字超出了参数size给出的缓冲区长度（这是一个ERANGE错误），它就返回“null”。如果操作成功，它返回指针buf。

> 如果子目录被删除（出现EINVAL错误）或者在程序运行过程中有关权限发生了变化（出现EACCESS错误），getcwd也会返回“null”。

3.7 扫描子目录

在UNIX系统上经常遇到的一个问题就是需要对子目录进行扫描，也就是确定文件被存放在哪个子目录里。在shell程序设计中，这很容易做到——让shell做一次表达式的通配符扩展就可以搞掂。在以前，UNIX操作系统的各种变体版本都曾允许用户通过编程去访问文件系统的底层结构。我们现在依然可以把子目录当作一个普通文件那样打开并直接读取其目录数据项，可多种多样的文件系统结构及其实现方法已经使这种办法没什么可移植性了。现在，一整套标准的库函数被开发出来，把子目录的扫描工作变得简单多了。

与子目录操作有关的函数是在一个名为dirent.h的头文件里被声明的。它们使用一个名为DIR的结构做为子目录处理操作的基础。一个被称之为“子目录流”（directory stream）的指向这种结构的指针（这是一个“DIR *”类型的数据）被用来完成各种普通的子目录操作，其工作原理与文件流（“FILE *”指针）差不多。目录数据项本身被返回保存在dirent结构里，该结构也是在dirent.h文件里定义的。永远不要直接改动DIR结构里的数据域。

我们将学习下面这几个函数：

- opendir和closedir。
- readdir。

- telldir。
- seekdir。

3.7.1 opendir函数

opendir函数的作用是打开一个子目录并建立一个子目录流。如果成功，它将返回一个指向一个DIR结构的指针，目录数据项的读操作就是通过这个指针来完成的。

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
```

opendir在失败时会返回一个空指针。需要注意的是，子目录流使用了一个底层的文件描述符来访问子目录本身，所以如果打开的文件过多，opendir就可能会失败。

3.7.2 readdir函数

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

readdir函数将返回一个指针，指针指向的结构里保存着子目录流dirp中下一个目录数据项的有关资料。后续的readdir调用将返回后续的目录数据项。如果发生错误，或者到达子目录尾，readdir将返回“NULL”值。POSIX兼容系统在到达子目录尾时会返回“NULL”，但不改变errno的取值；只有在发生错误时才设置errno的取值。

注意，如果在readdir函数扫描子目录的同时还有其他进程在那个子目录里创建或删除着文件，readdir将不保证能够列出该子目录里的所有文件（和下级子目录）。

dirent结构中包含着的目录数据项内容包括以下数据：

- ino_t　　d_ino　　文件的inode
- char　　d_name[ ]　　文件的名字

要想进一步查明子目录中某个文件的详细资料，还需要再使用一个stat系统调用。

3.7.3 telldir函数

```
#include <sys/types.h>
#include <dirent.h>

long int telldir(DIR *dirp);
```

telldir函数的返回值里记录着子目录流里的当前位置。我们可以在随后的seekdir调用里利用这个值对当前位置再做一次子目录扫描。

3.7.4 seekdir函数

```
#include <sys/types.h>
#include <dirent.h>
```

```
void seekdir(DIR *dirp, long int loc);
```

seekdir函数的作用是对dirp指定的子目录流中的目录数据项的指针进行设置。loc的值用来设置指针位置，它应该通过前一个telldir调用获得。

3.7.5 closedir函数

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
```

closedir函数的作用是关闭一个子目录流并释放与之关联的资源。它在执行成功时返回“0”，如果发生错误就返回“-1”。

在下面那个printdir.c程序里，我们把许多文件处理函数集中在一起使用，目的是得到一个简单的子目录内容清单。子目录中的每个文件单独列在一行上。如果是一个下级子目录，会在它的名字后面加上一个斜线字符“/”，下级子目录中的文件在缩进四个空格后依次排列。

程序会逐个切换到每一个下级子目录里，这样使它找到的文件都有一个可用的名字，也就是说，它们都可以被直接传递到opendir函数里去。如果子目录的嵌套结构太深，程序执行时就会失败，因为允许打开的子目录流的个数是有一个上限的。

我们当然可以采取一个更通用的做法，让我们的程序能够通过一个命令行参数来指定出发点。如果读者打算编写一个更具通用性的程序，请查阅有关工具程序（如ls和find等）的Linux源代码找找灵感。

动手试试：一个子目录扫描程序

1) 程序的开始是一些必要的头文件。接下来是一个printdir函数，它的作用是给出当前子目录的内容清单。它将递归遍历各个下级子目录，depth参数用来控制输出清单中的空格缩进。

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>

void printdir(char *dir, int depth)
{
    DIR *dp;
    struct dirent *entry;
    struct stat statbuf;

    if((dp = opendir(dir)) == NULL) {
        fprintf(stderr,"cannot open directory: %s\n", dir);
        return;
    }
    chdir(dir);
    while((entry = readdir(dp)) != NULL) {
        lstat(entry->d_name,&statbuf);
        if(S_ISDIR(statbuf.st_mode)) {
            /* Found a directory, but ignore . and .. */
            if(strcmp(".",entry->d_name) == 0 ||
                strcmp("..",entry->d_name) == 0)
                continue;
            printf("%*s%s/\n",depth,"",entry->d_name);
```

```
            /* Recurse at a new indent level */
            printdir(entry->d_name,depth+4);
        }
        else printf("%*s%s\n",depth,"",entry->d_name);
    }
    chdir("..");
    closedir(dp);
}
```

2) 现在轮到我们的main函数：

```
int main()
{
    printf("Directory scan of /home/neil:\n");
    printdir("/home/neil",0);
    printf("done.\n");

    exit(0);
}
```

这个程序的输出结果如下所示（经过删节）：

```
$ printdir
Directory scan of /home/neil:
.less
.lessrc
.term/
    termrc
.elm/
    elmrc
Mail/
    received
    mbox
.bash_history
.fvwmrc
.tin/
    .mailidx/
    .index/
        563.1
        563.2
posted
attributes
active
tinrc
done.
```

操作注释：

大部分操作都是在printdir函数里完成的，所以我们重点对它进行说明。在一些初始的错误检查之后，调用opendir函数，它能够检查子目录是否存在。接下来，printdir调用chdir进入指定的子目录。在readdir函数返回的数据项不为空的前提下，程序检查该数据项是否是一个子目录。如果它不是，根据depth的值缩进并打印出文件数据项的内容。

如果该数据项确实是一个子目录，我们就需要对它进行递归遍历。在跳过“.”和“..”数据项（它们分别代表当前子目录和上一级子目录）之后，printdir函数调用自己再次进入一个同样的处理过程。那它又是如何退出这些循环的呢？到while循环完成的时候，函数调用“chdir("..")”将把它带回到目录树的上一级，从递归返回点开始继续以前的遍历，列出上级子目录的清单。函数调用closedir(dp)关闭子目录，确保打开的子目录流个数不超出它的需要。

做为第4章对UNIX环境进行讨论的引子，我们来看看一个能够使这个程序更具通用性的方法。这个程序的功能是很有限的，因为它只能对子目录/home/neil进行操作。我们按下面的办法对main进行修改，就能把它变成一个更有用的子目录浏览器：

```
int main(int argc, char* argv[])
{
    char *topdir = ".";
    if (argc >= 2)
        topdir=argv[1];

    printf("Directory scan of %s\n",topdir);
    printdir(topdir,0);
    printf("done.\n");

    exit(0);
}
```

我们修改了三条语句，增加了五条语句，可它现在是一个通用性的工具程序了。现在多了一个传递出发点子目录名的可选参数，其默认值是当前子目录。我们可以通过下面这样的命令运行它：

```
$ printdir /usr/local | more
```

输出结果将分页显示，用户可以前后翻页查看其输出。可以说，用户手里现在有了一个方便通用的目录树浏览小工具。再努把力，你还可以增加显示文件的空间占用情况、限制遍历显示的深度等其他功能。

3.8 错误处理

本章介绍了许多系统调用和函数，但正如我们已经看到的，它们会因为各种各样的原因而操作失败。在操作失败的时候，它们会设置外部变量errno的值来指明自己失败的原因。许多不同的函数库都把这个变量用做报告错误的标准办法。我们也反复告诫大家，程序必须在函数报告出错之后立刻检查errno变量，因为它可能被下一个被程序调用的函数覆盖；有时候，下一个函数即使自身并没有出错，也会覆盖这个变量。

错误代码的取值和含义都列在头文件errno.h里，其中包括：

- EPERM　　操作不允许。
- ENOENT　　文件或子目录不存在。
- EINTR　　系统调用被中断。
- EIO　　I/O错误。
- EBUSY　　设备或资源忙。
- EEXIST　　文件存在。
- EINVAL　　非法参数。
- EMFILE　　打开的文件过多。
- ENODEV　　设备不存在。
- EISDIR　　是一个子目录。
- ENOTDIR　　不是一个子目录。

有两个函数可以用来在错误出现时报告它们，它们是strerror和perror。

```
#include <string.h>
char *strerror(int errnum);
```

strerror函数把错误的编码映射为一个字符串，由该字符串对刚才发生的错误的类型及原因进行说明。

```
#include <stdio.h>
void perror( const char *s );
```

perror函数把error变量中报告的当前错误映射到一个字符串里去，再把它输出到标准错误输出流上去。错误类型的前面先加上字符串s（如果没有给出参数s，就默认使用一个null空字符串）中给出的信息，再加上一个冒号和一个空格。请看下面的例子：

```
perror("program");
```

它会在标准错误输出上给出如下所示的信息：

```
program: Too many open files
```

3.9 高级论题

读者可以跳过我们在这一小节讨论的两个问题，因为涉及它们的情况很少出现。我们还要在此介绍它们的原因是为了让大家有一个参考，因为它们能够对一些让人头痛的问题提供比较简单的解决方案。

3.9.1 fcntl系统调用

fcntl系统调用对底层文件描述符提供了更高级的操控手段。

```
#include <fcntl.h>

int fcntl(int fildes, int cmd);
int fcntl(int fildes, int cmd, long arg);
```

利用fcntl系统调用，我们可以对打开了的文件描述符完成一些杂项操作，其中包括对它们进行复制、获取和设置文件描述符标志、获取和设置文件状态标志，以及管理文件加锁功能等。

对各种操作的取舍是通过给命令参数cmd选取不同的值来实现的，这个参数的取值在头文件fcntl.h里定义。根据所选择命令的具体情况，fcntl系统调用可能还要求加上它的第三个参数arg。

请看下面这个调用：

```
fcntl(fildes, F_DUPFD, newfd);
```

它返回一个新的文件描述符，新文件描述符的数值等于或大于整数newfd。这个新文件描述符是描述符fildes的一个复制品。它的作用类似于系统调用dup(fildes)，但还要参考已经打开的文件数量和newfd的取值才能确定。

再看下面这个调用：

```
fcntl(fildes, F_GETFD)
```

它返回的是在fcntl.h文件里定义的文件描述符标志，其中包括一个FD_CLOEXEC标志，它

的作用是检查在成功地调用了某个exec类的系统调用之后该文件描述符是否被关闭了。

请看下面这个调用：

```
fcntl(fildes, F_SETFD, flags)
```

它被用来设置文件描述符标志，设置FD_CLOEXEC标志的情况最常见。

请看下面这个调用：

```
fcntl(fildes, F_GETFL)
fcntl(fildes, F_SETFL, flags)
```

这两个调用分别对文件的状态标志和权限模式代码进行设置。利用在fcntl.h文件中定义的掩码O_ACCMODE，我们可以提取出文件的权限模式代码。其他标志包括那些当open函数用O_CREAT模式打开文件时做为第三参数出现的标志。注意：用户只能对部分标志进行设置。准确地说，用户不能通过fcntl设置文件的权限。

通过fcntl还可以实现文件加锁功能。详情请参考man命令给出的使用手册页第2小节，或者等我们讲到第7章，我们将在那里讨论文件加锁问题。

3.9.2 mmap函数

UNIX里面有一个非常有用的功能，它允许程序共享内存，而Linux内核从2.0版本开始已经把这一功能包括在其中了。mmap（内存映射）函数的作用是对一段内存进行设置，使它能够被两个或更多个程序读写。一个程序做出的修改可以被其他的程序看见。

这一功能还可以用在文件的处理操作上。用户可以使某个磁盘文件的全部内容看起来就像是内存中的某个数组。如果文件由记录组成，而这些记录又能够用C语言中的结构来描述的话，用户就可以通过存取结构数组对文件内容进行修改。

这要通过使用带特殊权限设置集的虚拟内存段才能实现。对这类虚拟内存段的读写会使操作系统去读写磁盘文件与之对应的部分。

mmap函数将创建一个指向一段内存的指针，该指针将与通过一个打开的文件描述符来访问的文件的内容相关联。

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

用传递off参数的办法可以改变经共享内存段访问的文件中数据的起始偏移值。文件描述符由fildes参数给出。以这种办法被访问的数据量（即内存段的长度）由len参数设置。

我们可以通过addr参数请求使用某个特定的内存地址。如果这个参数的取值是零，结果中的指针将是自动分配的。这是推荐做法，不这样做会降低程序的可移植性，因为不同系统上的可用地址范围是不一样的。

prot参数用来设置内存段的访问权限。它是下列常数值的按位OR结果：

- PORT_READ　　允许对该内存段进行读操作。
- PORT_WRITE　　允许对该内存段进行写操作。
- PORT_EXEC　　允许该内存段被执行。

• PORT_NONE 不允许访问该内存段。

flags参数控制着程序对该段内存的访问方式，它们如表3-7所示：

表 3-7

MAP_PRIVATE	该内存段是私用的，对它的修改只在此局部范围内有效
MAP_SHARED	把对该内存段的修改保存到磁盘文件上去
MAP_FIXED	该内存段必须位于addr指定的地址处

msync函数的作用是把在该内存段的某个部分或整段中的修改写回到被映射文件里（或者从被映射文件里读出）。

```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

内存段需要修改的部分通过做为参数传递过来的起始地址addr和长度len确定。flags参数控制着修改的具体实现方式，见表3-8：

表 3-8

MS_ASYNC	采用异步写方式进行修改
MS_SYNC	采用同步写方式进行修改
MS_INVALIDATE	从文件中读回数据

munmap函数的作用是释放内存段：

```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

下面的mmap_eg.c程序演示了如何利用mmap和数组风格的存取操作对结构化数据文件进行修改。注意：2.0版本之前的Linux内核不完全支持mmap的这种用法。但这个程序在Solaris和其他系统上都能够正确运行。

动手试试：mmap函数的用法

1) 我们先定义一个RECORD数据结构，然后创建出NRECORDS个这样的记录，记录中保存着它们各自的编号。把这些记录都追加到文件records.dat里去。

```
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>

typedef struct {
    int integer;
    char string[24];
} RECORD;

#define NRECORDS (100)

int main()
```

```
{
    RECORD record, *mapped;
    int i, f;
    FILE *fp;
fp = fopen("records.dat","w+");
for(i=0; i<NRECORDS; i++) {
    record.integer = i;
    sprintf(record.string,"RECORD-%d",i);
    fwrite(&record,sizeof(record),1,fp);
}
fclose(fp);
```

2) 现在，我们把第43条记录中的整数值由43修改为143，并把它写到第43条记录中的字符串里去。

```
fp = fopen("records.dat","r+");
fseek(fp,43*sizeof(record),SEEK_SET);
fread(&record,sizeof(record),1,fp);

record.integer = 143;
sprintf(record.string,"RECORD-%d",record.integer);

fseek(fp,43*sizeof(record),SEEK_SET);
fwrite(&record,sizeof(record),1,fp);
fclose(fp);
```

3) 现在把这些记录映射到内存中去，再顺序访问到第43条记录，把它的整数值修改为243（同时还要修改该记录中的字符串），还是使用内存映射的办法。

```
f = open("records.dat",O_RDWR);
mapped = (RECORD *)mmap(0, NRECORDS*sizeof(record),
                        PROT_READ|PROT_WRITE, MAP_SHARED, f, 0);

mapped[43].integer = 243;
sprintf(mapped[43].string,"RECORD-%d",mapped[43].integer);

msync((void *)mapped, NRECORDS*sizeof(record), MS_ASYNC);
munmap((void *)mapped, NRECORDS*sizeof(record));
close(f);

exit(0);
```

在后续章节的内容里我们还将学习另外一种内存共享功能，即System V系统的共享内存技术。

3.10 本章总结

在这一章里，我们学习了UNIX所提供的直接访问文件和设备的方法。我们介绍了在那些底层函数的基础上建立库函数以达成程序设计问题灵活解决方案的方法。通过这一章的学习，我们已经能够只用几行代码就编写出功能比较强大的子目录扫描程序。

我们在这一章里对文件和子目录进行了充分的学习，并在此基础上把我们在第2章末尾编写的CD唱盘管理软件转换为一个C语言程序。我们使用的解决方案更具结构化特点，更贴近文件。在这一阶段，我们还无法给这个程序增加什么新的功能，所以对整个程序的重写工作将延续到我们学习了屏幕显示和键盘输入方面更多处理手段之后再进行，而这恰好是后两章的主题。

第4章　UNIX环境

当我们为UNIX编写程序的时候，必须想到这个程序将运行在一个多任务环境里。这句话的意思是与此同时还会有其他程序在运行，它们和我们的程序一起共享着内存、磁盘空间、CPU时间等计算机资源。甚至有可能同时运行着同一程序的多个实例。让这些程序彼此相安无事是很重要的，它们应该了解周围的情况并采取适当的措施保证这一点。

在这一章里，我们重点学习程序运行在其中的环境、程序如何通过那个环境来了解自身的操作处境，而程序的使用者又如何影响它们的行为。准确地说，我们将学习以下内容：

- 向程序传递参数。
- 环境变量。
- 查看时间。
- 临时文件。
- 获取用户和主机的有关资料。
- 生成和配置系统日志信息。
- 了解系统各项资源的限制。

4.1　程序参数

当一个用C语言编写的UNIX程序开始运行的时候，它是从函数main开始的。UNIX程序中的main函数是像下面这样被定义的：

```
int main(int argc, char *argv[ ] );
```

其中argc是程序参数的个数；而argv是一个字符串数组，里面保存着参数的值。

我们有时候会看下面这样的main函数定义：

```
main( );
```

这样也行，因为main函数的返回值类型被默认为是int，而函数中用不着的参数不需要被声明。argc和argv倒是还在那里，可如果没有对它们做出声明，就不能使用它们。

不管操作系统何时开始运行一个程序，参数argc和argv都将被设置和传递到main函数里去。这些参数通常都是由其他程序提供的——大部分情况下它总是请求操作系统运行新程序的那个shell。shell接收到用户输入给它的命令行，把它分断为一个一个的单词，再把那些单词保存到argv数组里。请记住，UNIX操作系统的shell会在设置argc和argv参数之前对文件名进行通配符扩展，而DOS则要求程序接受带通配符的参数。

举例来说，如果我们在shell里给出了如下所示的命令：

```
$ myprog left right 'and center'
```

那么，myprog程序将从main函数开始执行，并且带有如下所示的参数：

```
argc: 4
argv: ( "myprog", "left", "right", "and center" )
```

注意：参数计数里包括程序名本身，而argv数组中的第一个元素argv[0]就是程序名。又因为我们在shell命令行里使用了引号，所以第四个参数是一个包含着空格的字符串。

如果读者曾经使用ISO/ANSI标准下的C语言进行过程序设计，就应该熟悉以上这些规定。main中的参数对应于shell脚本程序中的位置参数“$0”、“$1”等。在SO/ANSI标准里只规定了main的返回值必须是一个“int”类型的整数，但在X/Open技术规范里则对此做出了明确的定义声明。

命令行参数在向程序传递信息方面是很有用的。在一个数据库应用软件里，我们可以把希望使用的数据库的名字传递给程序，这样就可以使同一个程序用于不止一个的数据库上。许多工具程序也通过命令行参数改变其行为或设置其命令选项。在对这些我们所谓的标志或开关进行设置的时候，一般都要在它们的前面加上一个短划线字符（-）。请看下面的例子，sort程序通过一个“-r”开关对文件内容进行逆排序。

```
$ sort -r file
```

命令行选项是很常见的，按统一的方法来使用它们会给程序的使用者减少不少麻烦。在过去，每一种工具程序都有它们自己对待命令行选项的办法，而这时不时地会引起一些误会和混乱。下面这些命令在使用参数上就各有各的办法：

```
$ tar cvfB /tmp/file.tar 1024
$ dd if=/dev/fd0 of=/tmp/file.dd bs=18k
$ ls -lstr
$ ls -l -s -t -r
```

另一个容易发生误会的地方是（比如说）有的程序用选项“+x”来实现“-x”的对立操作。

撇开各具特点的语法格式不说，单是把所有这些命令选项的顺序和含义都记住就已经够困难的了。而解决这一问题的办法通常只有求助于“-h”（帮助）选项或某个man命令下的使用手册页，而这还要看程序员有没有提供它们。我们很快就会看到，工具函数getopt提供了一个简洁的解决这些问题的办法。现在先来看看被传递到程序中的参数是如何得到处理的。

动手试试：程序参数

下面这个args.c程序对它自己的参数进行检查。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int arg;

    for(arg = 0; arg < argc; arg++) {
        if(argv[arg][0] == '-')
            printf("option: %s\n", argv[arg]+1);
        else
            printf("argument %d: %s\n", arg, argv[arg]);
    }
    exit(0);
}
```

当我们运行这个程序的时候，它会打印出自己的参数并对其选项（简单说来就是以连字符

getopt的返回值是在argv数组中找到的下一个选项字符（如果还有的话）。循环调用getopt就可以按顺序找出全部的选项。具体做法如下：

1) 如果选项有一个关联值，则外部变量optarg将指向这个值。

2) 如果选项处理完毕，getopt返回“-1”。特殊参数“--”将使getopt停止扫描选项的工作。

3) 如果遇到一个无法识别的选项，getopt返回一个问号“?”，并把该选项保存到外部变量optopt里去。

4) 如果选项要求有个关联值（就象我们例子里的“-f”）但没有给出这个值，getopt将返回一个冒号“:”。

外部变量optind被设置为下一个待处理选项的索引下标，getopt用它来提醒自己已经走了多远。用户很少需要在程序中对这个变量进行设置。选项都处理完毕后，optind将指向在argv数组的尾部可以找到其余参数的地方。

有些版本的getopt会在它遇到第一个非选项参数处停下来，返回“-1”并设置optind的值。而另外一些，比如Linux提供的版本，能够对付出现在程序参数任意位置上的选项。注意：在这种情况下，实际已经由getopt重写了argv数组，把所有非选项参数都集中到argv数组的后部去了，它们的其始位置是argv[optind]。GNU版本getopt的这一行为是由环境变量POSIXLY_CORRECT控制的，如果它被置位，getopt就会在第一个非选项参数处停下来。此外，还有一些版本的getopt会在遇见无法识别的选项时给出一个出错信息。根据POSIX技术规范的说法，如果opterr变量是非零值，getopt就会向stderr输出一条出错信息。上述两种行为的例子我们马上就会看到。

动手试试：getopt函数的用法

我们在例子里使用了getopt，并把这个新程序叫做argopt.c。

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

int main(int argc, char *argv[])
{
    int opt;
while((opt = getopt(argc, argv, "if:lr")) != -1) {
        switch(opt) {
        case 'i':
        case 'l':
        case 'r':
            printf("option: %c\n", opt);
            break;
        case 'f':
            printf("filename: %s\n", optarg);
            break;
        case ':':
            printf("option needs a value\n");
            break;
        case '?':
            printf("unknown option: %c\n", optopt);
            break;
        }
    }
    for(; optind < argc; optind++)
        printf("argument: %s\n", argv[optind]);
    exit(0);
}
```

现在，当我们运行这个程序的时候，就会看到全体命令行参数都被自动处理好了：

```
$ ./argopt -i -lr 'hi there' -f fred.c -q
option: i
option: l
option: r
filename: fred.c
argopt: invalid option--q
unknown option: q
argument: hi there
```

操作注释：

程序循环调用getopt对选项参数进行处理直到没有遗漏为止，此时getopt会返回“-1”。每个选项都有与之对应的处理动作，对未知选项或缺失取值的处理也包括在内。如果读者使用的getopt版本与我们的不一致，看到的结果可能也会和上面显示的不同，特别是出错信息，但大概意思应该差不多。

选项都处理完毕后，程序像以前一样把其余的参数都打印出来，但这次是从optind位置开始的。

4.2 环境变量

我们在第2章见过环境变量。这些变量可以用来控制shell脚本程序和其他程序的操作行为。我们也可以用它们来配置用户环境。举例来说，每个用户都有一个HOME环境变量，它定义了用户的登录子目录路径，即用户工作的默认出发点。正如大家已经看到的，我们可以在shell提示符上查看环境变量。如下所示：

```
$ echo $HOME
/home/neil
```

我们还可以通过shell的set命令列出所有的环境变量。

在UNIX技术规范里定义了许多标准的环境变量，它们的用途各不相同，其中包括终端类型、默认编辑器、时区等。C语言程序可以通过putenv和getenv两个函数来访问环境变量。

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv(const char *string);
```

环境是由“name=value”形式的字符串构成的。getenv函数的作用是在环境中检索一个给定名字的字符串，其返回值是与该名字关联着的取值。如果给定变量不存在，它就返回“null”。如果给定变量存在但没有关联值，结果就会是getenv成功地取回一个字符串，但字符串的第一个字节是“null”。getenv取回的字符串保存在由getenv提供的静态变量里，不允许被应用程序（比如其他后续的getenv）调用、覆盖。

putenv函数的作用是把一个“name=value”形式的字符串添加到当前环境里去。如果因为可用空间缺乏的缘故而使它无法对环境进行扩展，它就会失败并返回“-1”。如果出现这样的情况，错误代码变量errno将被设置为ENOMEM。

我们来编写一个小程序，它能够把我们选取的任何环境变量的值输出出来。如果在调用这

个程序的时候我们还给出了第二个参数，就表示将对它的值进行设置。

动手试试：getenv和putenv函数

1) main函数声明后面的那几条语句用来保证environ.c能够被正确调用：

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

int main(int argc, char *argv[])
{
    char *var, *value;

    if(argc == 1 || argc > 3) {
        fprintf(stderr,"usage: environ var [value]\n");
        exit(1);
    }
```

2) 接下来，我们用getenv从环境中取出环境变量的值：

```
var = argv[1];
value = getenv(var);
if(value)
    printf("Variable %s has value %s\n", var, value);
else
    printf("Variable %s has no value\n", var);
```

3) 然后，我们检查调用这个程序的时候是否还有第二参数。如果有，我们就通过构造一个“name=value”形式的字符串，再调用putenv把第二参数的值赋给那个环境变量：

```
if(argc == 3) {
    char *string;
    value = argv[2];
    string = malloc(strlen(var)+strlen(value)+2);
    if(!string) {
        fprintf(stderr,"out of memory\n");
        exit(1);
    }
    strcpy(string,var);
    strcat(string,"=");
    strcat(string,value);
    printf("Calling putenv with: %s\n",string);
    if(putenv(string) != 0) {
        fprintf(stderr,"putenv failed\n");
        free(string);
        exit(1);
    }
```

4) 最后，我们再次调用getenv查看环境变量的新值：

```
        value = getenv(var);
        if(value)
            printf("New value of %s is %s\n", var, value);
        else
            printf("New value of %s is null??\n", var);
    }
    exit(0);
}
```

当我们运行这个程序的时候，就可以查看和设置环境变量了：

```
$ environ HOME
Variable HOME has value /home/neil
$ environ FRED
Variable FRED has no value
$ environ FRED hello
Variable FRED has no value
Calling putenv with: FRED=hello
New value of FRED is hello
$ environ FRED
Variable FRED has no value
```

注意：一个程序对应一个环境。在程序里做出的修改不会影响到它环境以外的东西，因为变量值不会从子进程（比如我们的程序）传递回父进程（比如shell）。

4.2.1 环境变量的用途

程序经常通过环境变量改变它们工作的方式。用户可以通过下面这些办法来设置这些环境变量的值：在缺省环境中进行设置、通过自己登录shell读取的某个.profile文件来设置、通过shell专用的启动文件（rc文件）来设置，以及在命令行上对变量进行设定等。请看下面的例子：

```
$ ./environ FRED
Variable FRED has no value
$ FRED=hello environ FRED
Variable FRED has value hello
```

shell把前面那个变量赋值语句看做是对环境变量的临时改变，在上面第二个例子里，程序environ将运行在变量FRED有一个赋值的环境里。

再举个例子，在我们CD唱盘数据库管理软件的未来版本里，我们将对一个名为CDDB的环境变量进行修改，这个变量的用途是指明我们使用的是哪个数据库。用户既可以自行设定出缺省值，也可以通过一条shell命令在运行时进行设定。

```
$ CDDB=mycds; export CDDB
$ cdapp
```

或者

```
$ CDDB=mycds cdapp
```

环境变量是一个双刃剑，大家在使用它们的时候一定要多加小心。与命令行选项相比，它们更“隐蔽”，因而会给调试工作带来不便。从某种意义上来说，环境变量有点象全局变量，它们能够改变程序的行为，引起不可预料的后果。

4.2.2 environ变量

正如我们已经看到的，程序环境是由“name=value”形式的字符串构成的。这个字符串数组通过environ变量直接面向程序，下面是environ变量的定义：

```
# include <stdlib.h>
extern char **environ;
```

动手试试：environ变量

下面这个showenv.c程序将通过environ变量列出环境变量：

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

extern char **environ;

int main()
{
    char **env = environ;

    while(*env) {
        printf("%s\n",*env);
        env++;
    }
    exit(0);
}
```

在一个Linux系统上运行这个程序，我们将得到如下所示的输出，我们对它的结果稍微做了一些删节：

```
$ ./showenv
HOSTNAME=tilde.provider.com
LOGNAME=neil
MAIL=/var/spool/mail/neil
TERM=console
HOSTTYPE=i386
PATH=/usr/local/bin:/bin:/usr/bin:
HOME=/usr/neil
LS_OPTIONS=-8bit-color=tty -F -T 0
SHELL=/bin/bash
PS1=\h:\w\$
PS2=>
OSTYPE=Linux
```

操作注释：

这个程序通过对以“null”结束的字符串数组中的environ变量进行遍历输出整个环境。

4.3 时间与日期

有时候，让程序能够确定时间和日期是很有用的。它可能需要在指定的运行时间进行登录，或者在某个特定的时间改变自己的操作行为。举例来说，一个游戏可能会在上班时间拒绝运行，而一个备份程序则需要在凌晨时分自动开始一次备份工作。

各种UNIX系统上时间和日期的起点都是一致的，即格林威治时间1970年1月1日的午夜。这个时间被认为是“纪元起点”。UNIX系统上的各种时间都是从那时开始以秒计数的。这与MS-DOS处理时间的办法很相似，只不过MS-DOS纪元是从1980年开始计算的。其他系统使用其他的纪元起点时间。

对时间进行处理时要使用一个预先定义好的数据类型time_t。它是一个大到能够容纳以秒计算的日期和时间数值的整数类型。在Linux系统上，它是一个long整数，它与对时间进行处理的函数一起定义在time.h文件里。

> 在使用32位time_t类型的UNIX和Linux系统上，时间将在2038年“轮回”。我们希

望到那个时候计算机早已开始使用一个大于32位的time_t类型了。这就是所谓的Y2K38问题，它的详细资料可以在http://www.comlinks.com/mag/ddates.htm上找到。

```
# include <time.h>
time_t time(time_t *tloc );
```

调用time函数可以看到以整数计算的底层时间值，它返回的是从纪元开始至今的秒数。time会把返回的时间整数值写到由tloc指定的位置，如果这不是一个空指针的话。

动手试试：time函数

下面是一个简单的envtime.c程序，目的是演示time函数的用法：

```
#include <time.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    int i;
    time_t the_time;

    for(i = 1; i <= 10; i++) {
        the_time = time((time_t *)0);
        printf("The time is %ld\n", the_time);
        sleep(2);
    }
    exit(0);
}
```

运行这个程序，它会在20秒种的时间内每隔两秒输出一次底层时间值。如下所示：

```
$ ./envtime
The time is 928663786
The time is 928663788
The time is 928663790
The time is 928663792
The time is 928663794
The time is 928663796
The time is 928663798
The time is 928663800
The time is 928663802
The time is 928663804
```

操作注释：

这个程序用一个空指针参数调用time函数，返回以秒数计算的日期时间值。程序休眠两秒再重复调用time函数，总共调用十次。

用秒数来表示的自1970年开始的日期和时间可以用来测算某些事情持续的时间段。我们可以把它简单地视为在先后两次调用time函数得到的返回值之间的一个减法问题。但意外的是ISO/ANSI标准委员会当初没有规定用time_t类型来测算以秒计数的时间值，所以他们另外发明了一个difftime函数，这个函数对两个time_t数值之间以秒为单位的差进行计算，结果返回一个double双精度浮点数。下面是它的定义：

```
# include <time.h>
```

```
double difftime( time_t time1, time_t time2 );
```

difftime函数对两个时间值进行减法计算，返回一个浮点数“time1 - time2”。对UNIX来说，time函数的返回值是一个代表秒时间的数字，可以对它进行计算处理；但为了最大限度地增加可移植性，最好还是使用difftime函数。

为了提供（对人类）更有意义的时间表达方式，需要我们把时间整数转换为日常生活中使用的日期和时间形式。有几个标准化函数可以帮我们的忙。

gmtime函数把底层时间整数值分断为一个结构，里面的数据域更有实际使用价值。下面是对它的定义：

```
#include <time.h>
struct tm *gmtime(const time_t timeval );
```

tm结构被定义为至少包含以下几个数据元素见表4-1：

表 4-1

tm	数据元素的说明
int tm_sec	秒，0~61
int tm_min	分，0~59
int tm_hour	小时，0~23
int tm_mday	月份中的日期，1~31
int tm_mon	年份中的月份，0~11（0代表一月）
int tm_year	从1900年开始计算的年份
int tm_wday)	星期几，0~6（0代表星期日）
int tm_uday	年份中的日期，0~365
int tm_isdst	是否为夏日制

tm_sec的范围允许临时闰秒，或者叫做双闰秒。

动手试试：gmtime函数

下面这个gmtime.c程序通过tm结构和gmtime函数给出当前的日期和时间：

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

int main()
{
    struct tm *tm_ptr;
    time_t the_time;

    (void) time(&the_time);
    tm_ptr = gmtime(&the_time);

    printf("Raw time is %ld\n", the_time);
    printf("gmtime gives:\n");
    printf("date: %02d/%02d/%02d\n",
        tm_ptr->tm_year, tm_ptr->tm_mon+1, tm_ptr->tm_mday);
    printf("time: %02d:%02d:%02d\n",
        tm_ptr->tm_hour, tm_ptr->tm_min, tm_ptr->tm_sec);
    exit(0);
}
```

运行这个程序，我们将看到含义明显的日期和时间表达方式：

```
$ ./gmtime; date
Raw time is 928663946
gmtime gives:
date: 99/06/06
time: 10:12:26
Sun Jun  6 11:12:26 BST 1999
```

操作注释：

这个程序调用time函数获取底层时间值，再调用gmtime把它转换为一个包含着含义明确的时间和日期值的结构。它用printf输出结构中的数据项。严格地说，我们不应该像程序中那样打印出原始时间值，因为它并非在所有系统上都是一个long长整数。在执行完gmtime程序之后我们立刻运行了date命令以比较两者的输出。

这里还有一个小问题。如果读者是在格林威治地区以外的地理时区运行这个程序，或者读者当地已经进入夏日制时间，就会发现时间（也许还有日期）是不正确的。这是因为gmtime函数返回的是一个格林威治时间（也叫做全球协调时间）。UNIX这样做的原因是为了同步全球各地的程序和系统，使不同时区同一时刻创建的文件都具有相同的创建时间。要想看到当地时间，我们需要用localtime函数代替gmitime函数。请看对它的定义：

```
# include <time.h>

struct tm *localtime(const time_t timeval );
```

localtime函数的功能与gmtime完全一致，只是它返回的结构里所包含的数值已经根据当地的时区和夏日制情况进行了调整。把gmtime程序里的gmtime函数替换为localtime函数之后再次运行它的时候，就能看到本地正确的时间和日期了。

我们可以通过mktime函数把一个分断好的tm结构转换为一个原始的time_t数值，下面是对它的定义：

```
# include <time.h>

time_t mktime(struct tm *timeptr );
```

如果tm结构不能用一个time_t数值来表示的话，mktime函数将返回“-1”。

我们可以通过asctime和ctime函数看到比date命令的输出更“友好”的日期和时间表示形式。下面是它们的语法：

```
#include <time.h>

char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timeval);
```

asctime函数返回一个字符串，它对应于由tm结构timeptr给定的时间和日期。返回字符串的格式如下所示：

```
Sun Jun  6 12:30:34 1999\n\0
```

它永远是这种长度为26字符的固定格式。ctime函数相当于下面的函数调用：

```
asctime( localtime( timeval ) )
```

它以一个原始时间值为参数，把它转换为一个更具可读性的当地时间。

动手试试：ctime函数

我们用下面的代码来说明ctime函数的用法：

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

int main()
{
    time_t timeval;

    (void)time(&timeval);
    printf("The date is: %s", ctime(&timeval));
    exit(0);
}
```

我们给这个程序起名为ctime.c，编译并运行这个程序将看到如下所示的输出：

```
$ ./ctime
The date is: Sun Jun  6 12:50:27 1999
```

操作注释：

ctime.c程序调用time函数获取底层时间值，然后由ctime函数完成重点工作，把原始时间值转换为人们易于理解的字符串，再把它打印出来。

为了更精确地控制输出时间和日期的格式，现代UNIX系统为我们提供了一个strftime函数。它很像是一个输出时间和日期专用的sprintf函数，工作过程也差不多：

```
# include <time.h>

size_t strftime( char *s, size_t maxsize, const char *format, struct tm *timeptr );
```

strftime函数对timeptr指向的tm结构所代表的时间和日期进行格式编排，其结果放在字符串s中。该字符串的长度被设置为（最少）maxsize个字符。格式字符串format用来对写入字符串的字符进行控制。类似于printf，它包含着将被传送到字符串里去的普通字符以及编排时间和日期格式的转换控制符。转换控制符见表4-2：

表 4-2

转换控制符	说　明
%a	星期几的简写形式
%A	星期几的全称
%b	月份的简写形式
%B	月份的全称
%c	日期和时间
%d	月份中的日期，0~31
%H	小时，00~23
%I	12进制小时钟点，01~12
%j	年份中的日期，001~366
%m	年份中的月份，01~12

（续）

转换控制符	说 明
%M	分，00~59
%p	上午或下午
%s	秒，00~61
%u	星期几，1~7（1代表星期一）
%U	一年中的第几个星期，01~53（星期天是一周中的第一天）
%V	一年中的第几个星期，01~53（星期一是一周中的第一天）
%w	星期几，0~6（0代表星期天）
%x	当地格式的日期
%X	当地格式的时间
%y	年份中的最后两位数，00~99
%Y	年
%Z	地理时区名称
%%	一个“%”字符

所以，date程序给出的普通日期就相当于strftime格式字符串中的：

```
"%a %b %d %H:%M:%S %Y"
```

为了帮助人们读取日期，我们可以使用strptime函数。它以一个代表时间和日期的字符串为参数，生成一个代表同一时间和日期的tm结构：

```
#include <time.h>

char *strptime( const char *buf, const char *format, struct tm timeptr );
```

格式字符串format的构造方法与strftime函数中的format完全一样。strptime函数在字符串扫描方面类似于sscanf函数，也是对匹配数据域进行查找并把它们写入与之对应的变量。在这里，根据format字符串填出的是tm结构中的数据元素。但与strftime函数中使用的转换控制符相比，strptime函数里的要松快一些，因为strptime函数中的星期几和月份用简写或全称都行，两种写法都能够匹配strptime函数中的“%a”控制符。此外，在strftime函数里，小于10的数字前面永远要加上一个“0”，而strptime函数则把它看做是可选的。

strptime函数返回的是一个指针，它指向转换过程期间紧跟在最后一个被转换字符后面的那个字符。如果它遇到一个无法被转换的字符，转换工作就简单地停在那个位置上。调用者程序需要检查是否已经从做为参数的字符串上读入了足够的内容以保证有意义的值被写到tm结构里去。

> 动手试试：strftime和strptime函数。

请注意下面这个程序中选用的转换控制符。

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

int main()
{
    struct tm *tm_ptr, timestruct;
    time_t the_time;
```

```
    char buf[256];
    char *result;

    (void) time(&the_time);
    tm_ptr = localtime(&the_time);
    strftime(buf, 256, "%A %d %B, %I:%S %p", tm_ptr);

    printf("strftime gives: %s\n", buf);

    strcpy(buf,"Mon 26 July 1999, 17:53 will do fine");

    printf("calling strptime with: %s\n", buf);
    tm_ptr = &timestruct;

    result = strptime(buf,"%a %d %b %Y, %R", tm_ptr);
    printf("strptime consumed up to: %s\n", result);

    printf("strptime gives:\n");
    printf("date: %02d/%02d/%02d\n",
        tm_ptr->tm_year, tm_ptr->tm_mon+1, tm_ptr->tm_mday);
    printf("time: %02d:%02d\n",
        tm_ptr->tm_hour, tm_ptr->tm_min);
    exit(0);
}
```

编译并运行这个strftime.c程序，我们得到：

```
$ ./strftime
strftime gives: Sunday 06 June, 11:55 AM
calling strptime with: Mon 26 July 1999, 17:53 will do fine
strptime consumed up to:  will do fine
strptime gives:
date: 99/07/26
time: 17:53
```

操作注释：

strftime程序通过调用time和localtime函数取得本地的当前时间。然后调用strftime函数根据一个相应的格式编排参数把它转换为人们可以理解的表示形式。为了演示strptime函数的用法，程序建立了一个包含着时间和日期的字符串，然后调用strptime函数提取出时间和日期的原始值并输出它们。strptime函数里的转换控制符“%R”是控制符“%H:%M”的简写形式。

千万记住，为了成功扫描一个日期，strptime函数必须要有一个精确的格式字符串。一般来说，除非在格式编排方面的要求非常严格，否则它不会去精确扫描读自用户的日期字符串。

读者可能会在编译strftime.c程序时看到一条编译器发出的警告信息。这是因为GNU的函数库在默认情况下不会对strptime函数做出声明。这很容易解决，在包括time.h文件的语句前加上下面这条语句就行，它的作用是明确申请使用X/Open技术规范中的标准功能。

```
# define _XOPEN_SOURCE
```

4.4 临时文件

在很多情况下，程序会用到一些文件形式的临时存储手段。比如保存计算的中间结果，或者在关键操作之前制作文件的备份拷贝等。举个例子，一个数据库应用软件在删除数据记录时就会使用一个临时文件。这个文件收集需要被保留的数据库数据项，最后，在处理结束的时候，这个临时文件变成了新的数据库，而原来的数据库则被删除了。

临时文件的这种用法很常见，但也有一个隐蔽的缺点，即用户必须保证临时文件选用的文

件名是独一无二的。如果情况不是这样，因为UNIX是一个多任务系统，其他程序就可能会选上相同的文件名，两个程序就会彼此干扰影响。

用tmpnam函数可以生成一个独一无二的文件名，请看对它的声明：

```
# include <stdio.h>
char *tmpnam( char *s );
```

tmpnam函数的作用是返回一个合法的文件名，它与任何现有的文件名都不一样。如果字符串s不为空，则新文件名就写在其中。后续的tmpnam调用会覆盖用来保存返回值的静态存储单元，所以如果需要多次调用tmpnam函数，就必须使用一个字符串参数。字符串的长度至少要有L_tmpnam个字符。在一个程序里，最多可以调用TMP_MAX次tmpnam函数，而它每次都会生成一个不同的文件名。

如果遇到需要立刻用到临时文件的情况，我们可以通过tmpfile函数在获得它名字的同时打开它。这是非常重要的，因为另外一个程序可能会创建出一个与tmpnam函数的返回值同名的文件。tmpfile函数避免了这类问题的发生。下面是对它的定义：

```
# include <stdio.h>
FILE *tmpfile( void );
```

tmpfile函数返回一个文件流指针，它指向一个独一无二的临时文件。该文件已经以读写方式被打开（通过fopen函数及其“w+”选项），当该文件上的引用线索都被关闭时，它就会被自动删除。

如果操作失败，tmpfile函数将返回一个null空指针并对errno进行设置。

> 动手试试：tmpnam和tmpfile函数。

我们来看看这两个函数的用法。

```
#include <stdio.h>

int main()
{
    char tmpname[L_tmpnam];
    char *filename;
    FILE *tmpfp;

    filename = tmpnam(tmpname);

    printf("Temporary file name is: %s\n", filename);

    tmpfp = tmpfile();
    if(tmpfp)
        printf("Opened a temporary file OK\n");
    else
        perror("tmpfile");
    exit(0);
}
```

编译并运行这个tmpnam.c程序，就可以看到由tmpnam函数生成的独一无二的文件名：

```
$ ./tmpnam
Temporary file name is: /tmp/filedm9a2K
Opened a temporary file OK
```

操作注释：

程序调用tmpnam函数为临时文件生成了一个独一无二的文件名。如果我们打算用它，就必须尽快打开这个文件，把另一个程序打开一个同名文件的风险降到最小。tmpfile调用会同时完成创建并打开一个临时文件的工作，也就避免了这种危险。

老版本的UNIX有另外一种生成临时文件名的办法，就是对mktemp和mkstemp函数进行调用。它们与tmpnam函数很相似，不同之处在于我们还可以给临时文件名指定一个模版，这使我们对它们的存放地点和名字多少增加了一些控制。请看对它们的定义：

```
#include <stdlib.h>

char *mktemp(char *template);
int mkstemp(char *template);
```

mktemp函数以给定的模版为基础生成一个独一无二的文件名。参数template必须是一个末尾带有六个“X”字符的字符串。mktemp函数将把这六个“X”字符替换为合法文件名字符的一个组合。它返回一个指向生成的字符串的指针；如果它无法生成一个独一无二的名字，就返回一个空指针。

mkstemp函数类似于tmpfile，它也是同时完成一个临时文件的创建和打开工作。文件名的生成办法与mktemp函数用的一样，但返回结果则是一个打开的底层文件描述符。一般来说，我们应该尽量多用tmpnam和tmpfile函数，少用mktemp和mkstemp函数。

4.5 用户的个人资料

除init以外，一切UNIX程序都是由其他程序或用户启动的。我们将在第10章对运行中的程序或进程彼此之间的交互作用做进一步的学习。用户通常是在一个响应他们命令的shell里启动程序的。我们已经看到，通过检查环境变量和读取系统时钟，一个程序可以对自身所处的环境大部分作出确定。程序还可以查出与其使用者有关的个人资料。

一个用户，当他或她在一个UNIX系统里登录上机的时候，就会有一个用户名和一个口令字。在对这些资料进行核实之后，用户就可以进入一个shell进行操作了。从原理上来说，用户还有一个独一无二的用户身份标识符，也叫做一个UID。UNIX运行的每个程序实际上都是被某个用户运行的，因此都会有一个关联着的UID值。

我们可以对程序进行设置，让它们看上去就好像是由另外一个用户启动的。当一个程序的setUID权限被置位的时候，其运行就会像是由这个可执行文件的属主启动的。而在执行了su命令的时候，它的运行又会像是由根用户启动的。接下来，程序会核查用户的访问权限，把UID修改为目标帐户的对应值，然后执行该帐户的登录shell。通过类似手段，我们还可以让一个程序看起来就像是被另外一个用户启动的，系统管理员经常会用这种办法执行一些系统维护方面的工作。

既然说到UID是用户身份证明的关键，我们就从它开始吧。

UID有自己的数据类型——uid_t，这个类型是在sys/types.h文件里定义的。它一般会是一个小整数。有的UID是由系统预先定义的，其他一些则是由系统管理员根据新用户的上机申请创建

出来的。一般情况下，用户的UID值都大于100。

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid(void);
char *getlogin(void);
```

getuid函数返回的是与程序关联着的那个UID。它通常就是启动该程序的用户的UID。

getlogin函数返回的是与当前用户关联着的登录名。

系统文件/etc/passwd里包含着一个与用户帐户有关的数据库。它的每一行分别代表着一位用户，里面的资料包括用户名、加密口令字、用户身份标识符（即UID）、分组标识符（即GID）、用户的全名、登录子目录以及缺省shell等。下面就是一个示范行：

```
neil:zBqxfqedfpk:500:4:Neil Matthew:/home/neil:/bin/bash
```

如果我们编写一个程序，使它能够查出启动它的那个用户的UID，就可以再对它进行扩展，使它能够查看口令字文件的内容，找出用户的登录名和全名。我们并不推荐这种做法，原因是为了改善系统的安全性，现如今的UNIX系统基本上都不再使用简单的口令字文件了。许多系统有使用“shadow”（隐蔽）口令字文件的选择权，原来的文件里不再有任何加过密的口令字信息（加密信息通常会被保存到/etc/shadow文件里去，这是一个普通用户不能读取的文件）。出于这方面考虑，人们又定义并开发了几个函数，向程序员提供了一个标准而又有效的获取用户个人资料的程序设计接口。请看：

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

口令字数据库中使用的passwd结构定义在pwd.h文件里，其中包括以下数据元素见表4-3：

表 4-3

passwd元素	说 明
char *pw_name	用户的登录名
uid_t pw_uid	UID编号
gid_t pw_gid	GID编号
char *pw_dir	用户的登录子目录
char * pw_shell	用户的缺省shell

有些UNIX系统还会有一个用户全名数据域，但这并不是个标准做法：在某些UNIX系统上它是pw_gecos，而在其他系统上它又是pw_comment。这就意味着我们无法对此给出一个准确统一的用法。

getpwuid和getpwnam函数都会返回一个指针，指针又都指向一个与某个用户对应着的passwd结构。对getpwuid函数来说，用户身份是由其UID确定的；而getpwnam中使用的则是用户的登录名。如果出现错误，两个函数都会返回一个空指针并设置errno变量。

动手试试：用户的个人资料

下面这个user.c程序能够从口令字数据库里提取出用户的某些个人资料：

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    uid_t uid;
    gid_t gid;
    struct passwd *pw;

    uid = getuid();
    gid = getgid();

    pr ntf("User is %s\n", getlogin());

    printf("User IDs: uid=%d, gid=%d\n", uid, gid);

    pw = getpwuid(uid);
    printf("UID passwd entry:\n name=%s, uid=%d, gid=%d, home=%s, shell=%s\n",
        pw->pw_name, pw->pw_uid, pw->pw_gid, pw->pw_dir, pw->pw_shell);

    pw = getpwnam("root");
    printf("root passwd entry:\n");
    printf("name=%s, uid=%d, gid=%d, home=%s, shell=%s\n",
        pw->pw_name, pw->pw_uid, pw->pw_gid, pw->pw_dir, pw->pw_shell);
    exit(0);
}
```

它会给出如下所示的输出，在不同的UNIX版本里结果也会稍有差异：

```
$ ./user
User is neil
User IDs: uid=500, gid=500
UID passwd entry:
 name=neil, uid=500, gid=500, home=/usr/neil, shell=/bin/bash
root passwd entry:
name=root, uid=0, gid=0, home=/root, shell=/bin/bash
```

操作注释：

这个程序先调用getuid函数获得当前用户的UID。再把这个UID用在getpwuid函数里取得口令字文件里保存的详细资料。此外，我们还演示了通过在getpwnam函数里给出用户名“root”的办法来获取用户资料。

如果读者手里有一份Linux的源代码，就能在id命令的源代码里看到使用getuid函数的另外一个例子。

我们可以用getpwent函数扫描口令字文件中全部用户的资料。它的作用是取出连续的文件数据项，下面是它的定义：

```
#include <pwd.h>
#include <sys/types.h>

void endpwent(void);
struct passwd *getpwent(void);
void setpwent(void);
```

getpwent函数依次返回每一位用户的个人资料数据项。如果到达文件尾，它会返回一个空指

针。如果已经扫描了足够的数据项，我们可以调用endpwent函数终止这一处理过程。setpwent函数把口令字文件的读写指针重置到文件的开始，这样下一个getpwent调用将重新开始一次扫描。这些函数的操作原理与我们在第3章里遇见的子目录扫描函数opendir、readdir和closedir等大同小异。

与用户资料有关的其他函数

用户标识符和分组标识符（确实）还能被其他比较少用的函数获取：

```
#include <sys/types.h>
#include <unistd.h>

uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
int setuid(uid_t uid);
int setgid(gid_t gid);
```

分组标识符和用户标识符的详细资料请参考UNIX系统的使用手册，但读者可能会发现自己根本不需要对这些东西进行处理。

> 只有系统管理员才能调用setuid和setgid函数。

4.6 主机资料

既然一个程序可以查出与其使用者有关的资料，同样也可以查出一些与它正运行在其上的计算机有关的细节。uname(1)命令就可以提供出这类的信息。uname(2)系统调用也能在C语言程序里提供出同样的信息——用“man 2 uname”命令可以查到它的用法。

主机资料在许多方面都是很有用的。我们可以让某个程序在一个网络中根据它在其上运行的计算机的名字（比如学生机或管理员机）而实现不同的预定行为。从软件许可证的角度考虑，我们也许会限制某个程序只能运行在一台计算机上。所有这些都意味着我们需要一个能够确定程序到底运行在哪台机器上的办法。

如果UNIX系统上安装了网络组件，我们就可以通过gethostname函数轻易地获得它的网络名，这是它的定义：

```
# include <unistd.h>

int gethostname( char *name, size_t namelen );
```

gethostname函数把机器的网络名写到字符串name里去。这个字符串的长度至少要有namelen个字符。gethostname函数在成功时返回“0”，否则返回“-1”。

通过uname系统调用我们可以获取关于主机更进一步的详细资料：

```
# include <sys/utsname.h>

int uname( struct utsname*name );
```

uname函数把主机资料写到一个由name参数指向的结构里去。这个utsname结构的定义在sys/utsname.h文件里，至少包括表4-4所示几个数据元素：

表 4-4

utsname数据元素	说 明
char sysname[]	操作系统的名称
char nodename[]	主机的名字
char release[]	系统的发行级
char version[]	系统的版本号
char machine[]	硬件类型

uname函数在成功时返回一个非负整数；失败时返回“-1”并设置errno变量指出错误类型。

动手试试：主机资料

下面这个hostget.c程序能够提取出关于主机计算机的一些资料：

```
#include <sys/utsname.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    char computer[256];
    struct utsname uts;

    if(gethostname(computer, 255) != 0 || uname(&uts) < 0) {
        fprintf(stderr, "Could not get host information\n");
        exit(1);
    }

    printf("Computer host name is %s\n", computer);
    printf("System is %s on %s hardware\n", uts.sysname, uts.machine);
    printf("Nodename is %s\n", uts.nodename);
    printf("Version is %s, %s\n", uts.release, uts.version);
    exit(0);
}
```

在Linux系统上，这个程序会给出如下所示的输出。如果读者的机器已经上了网，你就会看到一个包括网络名在内的扩展主机名。

```
$ ./hostget
Computer host name is tilde
System is Linux on i686 hardware
Nodename is tilde
Version is 2.2.5-15, #2 Mon May 1016:39:40 GMT 1999
```

操作注释：

这个程序调用gethostname函数获得主机计算机的网络名。在上面的例子里它取到的名字是tilde。这台基于奔腾II处理器的Linux计算机的进一步资料是通过对uname函数的调用返回的。注意：uname函数返回的字符串与操作系统的具体实现有关，在我们的例子里，版本号字符串里还包含着内核的编译时间。

如果读者还想看看uname函数的另一个例子，请查阅uname命令的Linux源代码。

许可证问题

每台主机计算机都有一个独一无二的标识符，gethostid函数可以返回这个标识符：

```
# include <unistd.h>
long gethostid( void );
```

gethostid函数的作用是返回一个与主机计算机对应的独一无二的值。许可证管理者通过它来保证软件只能运行在拥有合法许可证的机器上。在Sun公司出品的工作站上，它返回的是在制造这台计算机时设置在非易失存储单元里的一个数字，它对系统硬件来说是独一无二的。

4.7 日志记录功能

许多应用软件需要对它们自己的活动进行记录。系统程序经常需要向控制台或者某个日志（log）文件写消息。这些消息可以是错误、警告，或者与系统状态有关的一般信息。举例来说，su程序会把某个用户尝试获取超级用户优先级但失败了的事件记录下来。

这些日志消息通常被记录在某个系统文件里，那些系统文件又被保存在一个专用的子目录里。它可能是/usr/adm子目录，也可能是/var/log子目录。对一个典型的Linux安装来说，文件/var/log/messages容纳着所有的系统级消息；/var/log/maillog文件容纳着来自电子邮件系统的其他日志消息；而/var/log/debug文件则容纳着调试信息。读者可以检查自己系统里在文件/etc/syslog.conf中给出的系统配置情况。下面是一些消息示例：

```
Nov 21 17:27:00 tilde kernel: Floppy drive(s): fd0 is 1.44M
Nov 21 17:27:00 tilde kernel: snd6 <SoundBlaster 16 4.11> at 0x220
Nov 21 17:27:00 tilde kernel: IP Protocols: ICMP, UDP, TCP
Nov 21 17:27:03 tilde sendmail[62]: starting daemon (8.6.12)
Nov 21 17:27:12 tilde login: ROOT LOGIN ON tty1
```

我们可以在这里看到系统记录下来的各种消息。前几条消息是由Linux内核在它自己开机引导和检测已安装硬件时报告的。然后，电子邮件代理sendmail报告自己已经启动。最后，login程序报告有一个超级用户登录上机。

注意 查看日志消息需要有超级用户优先权。

有些UNIX系统不提供一个这样的具有可读性的消息文件，但为系统管理员准备了用来查看系统事件数据库的工具。具体情况请读者查阅自己的系统文档。

虽然系统消息的格式和存储位置不尽相同，可产生这些消息的方法是标准的。UNIX技术规范为方便各种程序生成日志消息准备了一个操作接口，它就是syslog函数：

```
#include <syslog.h>
void syslog( int priority, const char *message, arguments . . . );
```

syslog函数的作用是把一条日志消息送到记日志工具那里去。每条消息都有一个priority参数，它是危险系数和程序标识码的按位OR结果。危险系数用来调控如何对这条日志消息作出反应，程序标识码日志着发出这条消息的是哪个程序。

定义自syslog.h文件中的程序标识码包括LOG_USER——它指出该消息来自一个用户应用程序（缺省值），以及LOG_LOCAL0、LOG_LOCAL1一直到LOG_LOCAL7--它们的含义由本地的系统管理员指定。

按危险系数递减顺序排列的优先级见表4-5：

表 4-5

优 先 级	说 明
LOG_EMERG	紧急情况
LOG_ALERT	高优先级故障，比如数据库崩溃等
LOG_CRIT	关键性错误，比如硬件操作失败等
LOG_ERR	错误
LOG_WARNING	警告
LOG_NOTICE	应该引起重视的特殊情况
LOG_INFO	通知性消息
LOG_DEBUG	调试信息

根据系统的具体配置情况，LOG_EMERG消息可能会被广播给全体用户；LOG_ALERT消息可能会邮寄给系统管理员；LOG_DEBUG消息可能会被忽略；而其他消息则记入某个消息文件里。带日志功能的程序很容易编写。需要我们做的工作很简单，只要在希望创建一条日志消息的时候调用一下syslog函数就可以了。

由syslog创建的消息分消息标题和消息内容两部分。标题部分包括产生该消息的程序标识码以及日期和时间。消息内容部分则由syslog的message参数生成，它的作用类似于一个printf格式字符串。syslog的其他参数要根据message字符串中printf风格的转换控制符来使用。转换控制符“%m”可以用来插入与错误代码变量errno的当前值相对应的出错消息字符串，在记录出错消息的时候这很有用。

动手试试：syslog函数

在这个程序里，我们尝试打开一个并不存在的文件：

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

int main()
{
    FILE *f;

    f = fopen("not_here","r");
    if(!f)
        syslog(LOG_ERR|LOG_USER,"oops - %m\n");
    exit(0);
}
```

当我们编译并运行这个syslog.c程序的时候，将看不到任何输出。但在/var/log/messages文件的尾部将会有这样一条消息：

```
Nov 21 17:56:00 tilde syslog: oops - No such file or diretory
```

操作注释：

在这个程序里，我们试图打开一个并不存在的文件。在文件打开操作失败的时候，我们调用syslog把这一事件记录到系统日志中。

注意：从日志消息里看不出调用日志功能的是哪个程序，它只记载了有人调用syslog记录一

条消息这样一个事实。转换控制符“%m”被替换为对该错误的一个说明，在我们的例子里就是“文件没有找到”。这比只给出错误编号“17”要明白多了！

日志记录功能的配置

在syslog.h文件里还定义了一些能够改变日志记录功能行为的其他函数。其中包括：

```
#include <syslog.h>

void closelog(void);
void openlog(const char *ident, int logopt, int facility);
int setlogmask(int maskpri);
```

调用openlog函数可以改变我们日志消息的内容形式。它允许我们设置一个字符串ident，它将加在我们的日志消息的前面。我们可以用它来指出哪个程序正在创建这条消息。facility参数记录着一个可以在此后的syslog调用中使用的程序标识码，它的缺省值是LOG_USER。logopt参数对此后的syslog调用的行为进行配置。它是零个或多个表4-6中的参数按位OR的结果：

表 4-6

logopt参数	说 明
LOG_PID	加上进程标识符，这是系统为每个进程在消息中分配的独一无二的数字标识
LOG_CONS	如果消息不能被记录到日志文件里，就把它们送到控制台去
LOG_ODELAY	在第一次调用syslog的时候打开日志功能
LOG_NDELAY	立刻打开日志功能，不等到第一次记日志时

openlog函数会分配并打开一个文件描述符，通过这个文件描述符来写日志。调用closelog函数将关闭那个日志文件。注意，在调用syslog之前并不需要调用openlog，因为syslog会根据需要自行打开日志功能。

通过setlogmask函数可以设置一个日志掩码，再通过这个掩码控制我们日志消息的优先级。此后，优先级没有在日志掩码里置位的syslog调用都将被丢弃，举例来说，你可以用这个办法关掉LOG_DEBUG消息而不影响程序的主体。

我们可以用LOG_MASK(priority)为日志消息创建一个掩码，它的作用是创建一个只包含一个优先级别的掩码；还可以用LOG_UPTO(priority)创建一个由指定优先级之前所有优先级（包括指定优先级）构成的掩码。

动手试试：logmask程序

我们来看看logmask程序的工作情况：

```
#include <syslog.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int logmask;

    openlog("logmask", LOG_PID|LOG_CONS, LOG_USER);
    syslog(LOG_INFO,"informative message, pid = %d", getpid());
```

```
    syslog(LOG_DEBUG,"debug message, should appear");
    logmask = setlogmask(LOG_UPTO(LOG_NOTICE));
    syslog(LOG_DEBUG,"debug message, should not appear");
    exit(0);
}
```

这个logmask.c程序不产生任何输出，但在一个典型的Linux系统上，我们可以在系统日志文件/var/log/messages的尾部看到下面这条消息：

```
Nov 21 18:19:43 tilde logmask[195]: informative message, pid = 195
```

文件/var/log/debug里将出现：

```
Nov 21 18:19:43 tilde logmask[195]: debug message, should appear
```

操作注释：

程序用自己的名字对记日志功能进行了初始化，并要求在日志消息里加上进程标识符。信息类消息将记入/var/log/messages文件，而调试信息将记入/var/log/debug文件。第二条调试信息没有出现在日志文件里，因为在它之前我们已经调用setlogmask函数对日志功能重新进行了配置，让它忽略所有优先级低于LOG_NOTICE的信息。需要注意的是，这种做法在早期的Linux内核上行不通。

如果读者在安装操作系统的时候没有激活调试信息记日志功能，或者采用的是其他配置情况，就可能看不到例子里给出的调试信息。要想记录一切调试信息，请把下面这一行加到文件/etc/syslog.conf的末尾再重启动计算机（简单点的办法是向syslogd进程发送一个挂起信号）。无论怎样做，最好先在你的系统文档里查出准确的配置细节。

```
* .debug /var/log/debug
```

logmask.c程序里使用了getpid函数，它与getppid函数的关系很密切，这两个函数的定义如下所示：

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

两个函数分别返回调用者的进程标识代码和它父进程的进程标识代码。关于PID的详细资料请参考第10章内容。

4.8 资源和限制

在UNIX系统上运行的程序在资源方面会受到一定的限制。它们可能是硬件方面的物理性限制（比如内存）、系统策略方面的限制（比如分配到的CPU时间），或者软件实现方面的限制（比如整数的长度或文件名所允许的最多字符个数）。UNIX技术规范对其中的某些限制做了定义，应用程序可以把它们检测出来。对限制及突破限制等方面的讨论请参考介绍数据管理知识的第7章内容。

头文件limits.h里有许多一目了然的常数定义，它们是一些操作系统方面的限制值。包括内

容见表4-7：

表 4-7

限制常数	它们的用途
NAME_MAX	文件名里允许使用的字符个数的最大值
CHAR_BIT	一个char类型的值里的二进制位数
CHAR_MAX	char类型的最大值
INT_MAX	int类型的最大值

能够用在应用程序里的限制常数还有很多，请读者查阅自己计算机里的头文件。需要注意的是，不同文件系统上的NAME_MAX值是不一样的。在重视可移植性的代码里应该使用pathconf函数。详细资料请参考pathconf的man命令使用手册页。

头文件sys/resource.h提供了资源操作方面的定义。其中包括对程序的最大长度、执行优先级、文件资源等方面的限制进行核查和设置的函数。请看：

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int priority);
int getrlimit(int resource, struct rlimit *r_limit);
int setrlimit(int resource, const struct rlimit *r_limit);
int getrusage(int who, struct rusage *r_usage);
```

id_t是一个用在用户标识符和分组标识符方面的整数类型。在sys/resource.h文件里定义的rusage结构用来确定当前程序占用了多少CPU时间。它至少由表4-8几个数据元素组成：

表 4-8

rusage数据元素	说 明
struct timeval ru_utime	用户占用的时间
struct timeval ru_stime	系统占用的时间

在sys/time.h文件里定义的timeval结构里的tv_sec和tv_usec数据域分别代表着以秒和微秒计算的程序占用时间。

程序消耗的CPU时间被分为用户时间（程序本身执行它自己的指令所消耗时间）和系统时间（操作系统因为这个程序的缘故而消耗的CPU时间，即完成输入输出操作的系统调用以及执行其他系统函数所花费的时间）。

getrusage函数的作用是把CPU时间信息写到由参数r_usage指向的rusage结构里去。它的who参数是表4-9中两个常数之一：

表 4-9

who常数	说 明
RUSAGE_SELF	只返回当前程序的CPU时间占用信息
RUSAGE_CHILDREN	还包括子进程的CPU时间占用信息

我们将在第10章遇到子进程和任务优先级，但为了让大家有一个比较完整的认识，我们在

这里对系统资源的实现过程稍加说明。简单地说，每个运行中的程序都有一个关联着的优先级，优先级比较高的程序分配到的CPU时间会多一点。普通用户只能降低自己程序的优先级，不能增加它们。

应用程序可以通过getpriority和setpriority函数确定和改变自己（或其他程序）的优先级。被优先级处理函数检查和修改的进程可以通过进程标识符、分组标识符或者用户标识符加以区分。which参数规定了对待who参数的方式，它们如表4-10所示：

表 4-10

which参数	说 明
PRIO_PROCESS	who是一个进程标识符
PRIO_PGRP	who是一个进程分组标识符
PRIO_USER	who是一个用户标识符

因此，要想取得当前进程的优先级，我们可以调用：

```
priority = getpriority( PRIO_PROCESS, getpid() );
```

setpriority函数允许设置一个新的优先级，如果能这样做的话。

缺省的优先级是0。正优先级用于后台任务，它们只有在没有更高优先级的任务准备运行时才会运行。负优先级会使程序运行得更频繁，从CPU的可用时间里分出更大的一块。合法优先级的范围是-20到+20。人们很容易在这方面犯糊涂，因为数值越大，执行起来其优先级反而越低。

getpriority函数在成功时返回一个合法的优先级，失败时返回“-1”并设置errno变量。因为“-1”本身是一个合法的优先级数字，所以在调用getpriority函数之前应该把errno变量设置为零，在返回时检查它是否依然是零。setpriority函数在成功时返回0，如果失败则返回“-1”。

系统资源方面的限制可以通过getrlimit和setrlimit函数读出和设置。这两个函数都要用rlimit结构来描述资源方面的限制。它在sys/resource.h文件里定义，由表4-11中元素组成：

表 4-11

rlimit数据元素	说 明
rlim_t rlim_cur	当前的软限制
rlim_t rlim_max	硬限制

定义类型rlim_t是一个用来描述资源容量的整数类型。一般来说，软限制是一个最好不要超越的限制，超越了则会引起库函数返回错误。而如果超越了硬限制，就可能会使系统终止程序的运行，办法是向程序发出一个信号：超越CPU时间限制时送出的是SIGXCPU信号，超越某个数据容量限制时送出的是SIGSEGV信号。程序可以把自己的软限制设置为小于硬限制的任意值。它可以减少自己的硬限制。只有以超级用户优先级运行的程序才能增加某项硬限制。

能够加以限制的系统资源有很多种。它们由rlimit函数的resource参数制定，有关定义都在sys/resource.h文件里。其内容见表4-12：

表 4-12

resource参数	说 明
RLIMIT_CORE	以字节计算的核心转储（core dump）文件长度限制
RLIMIT_CPU	以秒计算的CPU时间限制
RLIMIT_DATA	以字节计算的数据段（malloc和sbrk操作）长度限制
RLIMIT_FSIZE	以字节计算的文件长度限制
RLIMIT_NOFILE	文件打开数量方面的限制
RLIMIT_STACK	以字节计算的堆栈长度限制
RLIMIT_AS	以字节计算的地址空间（堆栈和数据）长度限制

下面是模仿典型应用程序的limits.c程序。它还设置并突破了一个资源限制。

动手试试：资源限制

1) 这个程序将要用到许多函数，先把对这些函数进行定义的头文件包括进来：

```
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>
#include <math.h>
```

2) work函数向一个临时文件写一个字符串10000次，然后进行一些数学运算，目的是制造一些CPU工作负荷：

```
void work()
{
    FILE *f;
    int i;
    double x = 4.5;

    f = tmpfile();
    for(i = 0; i < 10000; i++) {
        fprintf(f,"Do some output\n");
        if(ferror(f)) {
            fprintf(stderr,"Error writing to temporary file\n");
            exit(1);
        }
    }
    for(i = 0; i < 1000000; i++)
        x = log(x*x + 3.21);
}
```

3) main函数先调用work，再调用getrusage函数查看它使用了多少CPU时间。它把这些资料显示在屏幕上：

```
int main()
{
    struct rusage r_usage;
    struct rlimit r_limit;
    int priority;

    work();
    getrusage(RUSAGE_SELF, &r_usage);

    printf("CPU usage: User = %ld.%06ld, System = %ld.%06ld\n",
        r_usage.ru_utime.tv_sec, r_usage.ru_utime.tv_usec,
        r_usage.ru_stime.tv_sec, r_usage.ru_stime.tv_usec);
```

4) 接下来，它调用getpriority和getrlimit函数分别查出自己的当前优先级和文件长度限制值：

```
priority = getpriority(PRIO_PROCESS, getpid());
printf("Current priority = %d\n", priority);

getrlimit(RLIMIT_FSIZE, &r_limit);
printf("Current FSIZE limit: soft = %ld, hard = %ld\n",
    r_limit.rlim_cur, r_limit.rlim_max);
```

5) 最后，我们通过setrlimit函数设置了一个文件长度限制，然后再次调用work。work函数的执行失败了，因为它尝试创建的文件尺寸过大：

```
    r_limit.rlim_cur = 2048;
    r_limit.rlim_max = 4096;
    printf("Setting a 2K file size limit\n");
    setrlimit(RLIMIT_FSIZE, &r_limit);

    work();
    exit(0);
}
```

运行这个程序，我们将看到这个程序的CPU资源消耗情况以及这个程序运行时所处的缺省优先级。在设置了文件长度限制之后，程序对临时文件的写操作将不能超过2048个字节。

```
$ cc -o limits limits.c -lm
$ ./limits
CPU usage: User = 1.460000, System = 1.040000
Current priority = 0
Current FSIZE limit: soft = 2147483647, hard = 2147483647
Setting a 2K file size limit
File size limit exceeded
```

用nice命令启动这个程序可以改变它的优先级。我们将看到它的优先级被改为+10，结果是执行这个程序要多花些时间。

```
$ nice limits
CPU usage: User = 1.310000, System = 0.840000
Current priority = 10
Current FSIZE limit: soft = 2147483647, hard = 2147483647
Setting a 2K file size limit
File size limit exceeded
```

操作注释：

limits程序调用work函数来模仿一个典型应用程序的动作。它执行了一些计算并做了一些输出，本例是向临时文件写大约150K数据。它调用资源函数查看自身的优先级和文件长度限制。本例中的文件长度限制最初被设置为最大值，在磁盘空间允许的情况下我们可以创建出长达2GB的文件来。接下来，程序把自己的文件长度限制设置为只有2K，然后再次执行同样的工作。这一次，work函数失败了，因为它不能再创建出那么大的临时文件了。

注意 我们可以通过bash的ulimit命令给一个运行在某个特定shell下的程序加上限制。

在这个例子里，错误信息“Error writing to temporary file”可能不会象我们希望的那样显示在屏幕上。这是因为某些系统（比如Linux 2.2）会在程序超越资源限制的时候自动终止它的运行，它会发出一个SIGXFSZ信号做到这一点。我们将在第10章对信号及其用法进行学习。其他

POSIX兼容系统对超限制函数的处理可能只是让它返回一条错误信息。

4.9 本章总结

在这一章里，我们学习了UNIX环境，对程序的运行条件进行了研究。我们讨论了命令行参数和环境变量，这两样东西都能用来改变程序的默认行为，还可以为程序提供有用的操作选项。

我们还看到怎样才能让程序利用库函数处理日期和时间数据，怎样才能获取关于自身、用户以及在其上运行的计算机的信息。

因为UNIX程序通常都要共享主机上的珍贵资源，所以我们对如何确定和管理资源的问题也做了介绍。

第5章 终 端

先认真思考一下，看看对我们在第2章编写的基本应用程序都需要做哪些改进。它最明显的缺陷莫过于用户操作界面了。在功能方面它倒没有什么问题，就是不够精致。

在这一章里，我们将学习如何加强对用户终端，也就是键盘输入和屏幕输出的控制。程序需要从一个终端开始运行，而我们进一步的学习将涉及到如何确保用户向终端送去的各种输入能够被馈入程序，而程序的输出又能够到达屏幕上的正确位置。在学习过程中，我们还将体会到UNIX先驱们的良苦用心。

虽然重新编写CD唱盘管理软件的构想还要等到下一章才能见到曙光，但我们在这一章里学习到的内容却是基础性的工作。下一章内容是对curses函数库的讨论。这可不是什么古代咒语大全之类的东西（英文单词“curse”是咒语的意思），这是一个函数库，它提供了控制屏幕显示的高级代码。读者可以把本章内容看做是下一章的铺垫，它会让大家熟悉UNIX里终端输入输出的某些原理和概念。也许，我们在这一章里给出的底层访问手段恰好是读者正在寻找的东西。

在这一章里，我们将学习：

- 对终端进行读写。
- 终端驱动程序和通用终端接口（General Terminal Interface）。
- termios函数库。
- 终端的输出和terminfo功能包。
- 检测键盘击键动作。

5.1 对终端进行读写

通过第3章的学习，我们已经知道当程序从命令提示符处被调用的时候，shell会安排标准输入和标准输出流连接到我们的程序。通过getchar和printf功能可以很容易对这些缺省文件流进行读写，实现程序和用户之间的交流。

我们先用C语言把我们的菜单部分重写一遍，还使用getchar和printf这两个例程，我们给新程序起名为menu1.c。

动手试试：用C语言编写的菜单例程

1）程序开始部分的这些语句定义了一个用来显示菜单内容的字符数组和getchoice函数的框架模型：

```
#include <stdio.h>

char *menu[] = {
    "a - add new record",
    "d - delete record",
```

```
    "q - quit",
    NULL,
};

int getchoice(char *greet, char *choices[]);
```

2) main函数以样本菜单menu为参数调用getchoice函数：

```
int main()
{
    int choice = 0;

    do
    {
        choice = getchoice("Please select an action", menu);
        printf("You have chosen: %c\n", choice);
    } while(choice != 'q');
    exit(0);
}
```

3) 下面是关键性的代码：显示菜单和读取用户输入的函数getchoice：

```
int getchoice(char *greet, char *choices[])
{
    int chosen = 0;
    int selected;
    char **option;

    do {
        printf("Choice: %s\n",greet);
        option = choices;
        while(*option) {
            printf("%s\n",*option);
            option++;
        }
        selected = getchar();
        option = choices;
        while(*option) {
            if(selected == *option[0]) {
                chosen = 1;
                break;
            }
            option++;
        }
        if(!chosen) {
            printf("Incorrect choice, select again\n");
        }
    } while(!chosen);
    return selected;
}
```

操作注释：

getchoice函数显示程序的自我介绍greet和样本菜单choices，并提示用户选择一个代表某个菜单选项的字母。接下来程序进入循环，直到getchar函数返回一个与option数组中某个数据项的第一个字母匹配的字符为止。

编译并运行这个程序，我们将发现程序的行为和我们预计的并不一样。从下面这些操作对话里可以看出点迹象来：

```
$ menu1
Choice: Please select an action
a - add new record
d - delete record
```

```
q - quit
a
You have chosen: a
Choice: Please select an action
a - add new record
d - delete record
q - quit
Incorrect choice, select again
Choice. Please select an action
a - add new record
d - delete record
q - quit
q
You have chosen: q
$
```

用户需要输入“a/回车/q/回车”才能做出选择。看起来至少有两个问题，比较严重的那个是在每次正确选择的后面都会出现一个“Incorrect choice”（错误选择）提示。另外一个是只有按下回车键程序才会读取我们的输入。

为什么操作没有成功?

这两个问题是密切相关的。在系统缺省配置情况下，终端输入只有在用户按下回车键以后才会传递给程序使用。在大多数情况里这样做是有好处的，因为这可以让用户通过Backspace（后退）键和Delete（删除）键对打字错误进行修改。用户只有在对自己从屏幕上看到的东西满意的时候才会按下回车键把输入送往程序。

这种行为被称为授权模式，也叫做标准模式。所有输入都是以行为单位进行处理的。在输入行完成之前（一般就是用户按下回车键之前），对包括Backspace（后退）键在内的所有击键的处理都是由终端操作接口把持的，应用程序读不到任何字符。

它的对立面是非授权模式，此时程序对输入字符的处理有了更大的控制权。我们过一会儿再回到这两种模式上来。

UNIX终端管的事情是挺多的，其中之一是喜欢把中断字符翻译为与之对应的信号，从而自动地替用户完成对Backspace（后退）键和Delete（删除）键的处理，而用户也就不必在自己编写的每一个程序里重新实现它。我们将在第10章里看到更多的信号。

那么，我们的程序又是怎么一回事呢？是这样的：UNIX会在用户按下回车键之前先把输入字符保存起来，然后把用户选中的字符和紧随其后的回车字符一起送到程序里去。这样，用户对菜单每做一次选择，程序就会调用getchar对选择字符进行处理；然后再调用一次getchar，而这次它将会立刻返回一个回车字符。

程序实际看到的字符并不是一个ASCII回车字符CR（十进制的13，十六进制的0D），而是一个行进纸字符LF（十进制的10，十六进制的0A）。这是因为，就其内部原理来说，UNIX总是用一个行进纸字符来结束文本行。也就是说，UNIX使用一个单独的行进纸字符来表示一个换行；而其他系统——比如说DOS，要用两个字符来代表换行，一个回车加一个行进纸。如果输入或输出设备送来或请求一个回车，UNIX的终端处理部分会负责弄好它。如果读者习惯了DOS或其他环境，对这种做法多少会感到些奇怪，但UNIX自有其用意。在UNIX里，文本文件和二进制文件是没有任何实际区别的，这样做也就无可厚非。对UNIX系统来说，只有在用户对某个终端或

某些打印机和绘图仪进行输入输出时才需要对回车进行处理。

下面这些代码对多出来的回车字符进行过滤，把它加到我们的菜单程序里就可以改正其主要错误了：

```
do {
        selected = getchar();
} while(selected == '\n');
```

它解决了我们的燃眉之急。要想解决“必须按下回车键才能让程序继续执行”的问题，我们先得学点预备知识，最终我们将得到一个精巧的换行符解决方案。

5.1.1 对重定向输出进行处理

UNIX程序，甚至交互式的UNIX程序，经常会把自己的输入或输出重定向到文件或其他程序去。我们来看看当把我们的程序输出重定向到一个文件去的时候会出现什么样的情况：

```
$ menu1 > file
a
q
$
```

我们可以把这种情况看做是成功，因为输出确实被重定向到文件里去了，没有出现在屏幕上。但有时我们并不想这样做，或者我们想把准备让用户看到的提示信息与其他输出区别对待，前者仍然能够被用户看到，而后者将被安全地重定向。

检查标准输出是否被重定向的办法很简单，只要查看底层文件描述符是否关连在一个终端上就可以知道。isatty系统调用可以完成这一工作。我们只需简单地把一个合法的文件描述符传递给它，它就可以判断出该描述符是否正连接在一个终端上。

```
#include <unistd.h>
int isatty( int fd );
```

如果打开的文件描述符fd确实连接在一个终端上，isatty系统调用将返回“1”，否则返回“0”。

我们在程序里使用的是文件流，而isatty只能对文件描述符进行操作。因此，我们必须把isatty调用和我们在第3章里学到的fileno函数结合使用以完成必要的转换。

如果stdout已经被重定向了我们该怎么办呢？直接退出当然不是个好主意，因为用户没有办法知道程序为什么会运行失败。向stdout输出一条消息也无济于事，因为它肯定会从终端重定向到其他地方去。解决这个问题的办法之一是把消息写到stderr去，它不会被shell的“> file”命令重定向。

动手试试：检查是否存在输出重定向

1) 沿用我们在上一小节里编写的menu1.c程序，在其中加上一个新的include语句，把main函数按下列内容进行修改，给新文件起名为menu2.c。

```
#include <unistd.h>
...
int main()
{
```

```
    int choice = 0;

    if(!isatty(fileno(stdout))) {
        fprintf(stderr,"You are not a terminal!\n");
        exit(1);
    }
    do {
        choice = getchoice("Please select an action", menu);
        printf("You have chosen: %c\n", choice);
    } while(choice != 'q');
    exit(0);
}
```

2) 请看这个程序给出的样本输出：

```
$ menu2
Choice: Please select an action
a - add new record
d - delete record
q - quit
q
You have chosen: q
$ menu2 > file
You are not a terminal!
$
```

操作注释：

新代码段用isatty函数检查标准输出是否连接在一个终端上，如果不是，就停止运行。这也是shell用来决定是否需要提供shell提示符的检测手段。同时对stdout和stderr都进行重定向的情况也很常见。比如说，我们可以象下面这样把stderr文件流重定向到另外一个文件去：

```
$ menu2 > file  2 > file.error
$
```

或者用下面的命令把这两个输出流都重定向到同一个文件去：

```
$ menu2 > file  2 > &1
$
```

如果读者不熟悉重定向操作，请复习一下第2章内容，我们在那里对这一语法现象做了详细的说明。如果出现上面这样的情况，我们就需要把消息送往控制台。

5.1.2 与终端进行“对话”

如果我们不想让我们程序中与用户交互操作的部分被重定向，重定向的东西只限于其他的输入和输出，就必须把交互操作部分与stdout和stderr隔离开。直接对终端进行读写能够实现我们的愿望。UNIX从传统上看是一个多用户系统，往往会直接或经由网络连接多个终端，那么，我们这样才能找到正确的终端并使用它呢？

UNIX已经想到这个问题了，它通过一个特殊的/dev/tty设备消除了我们的顾虑。这个设备永远是当前终端或当前的登录会话。因为UNIX把一切事物都看做是文件，所以我们可以通过正常的文件操作对/dev/tty设备进行读写。

继续修改我们的程序，使我们能够通过向getchoice函数传递参数的办法加强对输出的控制。修改后的程序称为menu3.c。

动手试试：使用/dev/tty设备

3) 以menu2.c为蓝本对代码做下列修改，使输入和输出都被重定向到/dev/tty设备：

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

char *menu[] = {
    "a - add new record",
    "d - delete record",
    "q - quit",
    NULL,
};

int getchoice(char *greet, char *choices[], FILE *in, FILE *out);

int main()
{
    int choice = 0;
    FILE *input;
    FILE *output;

    if(!isatty(fileno(stdout))) {
        fprintf(stderr,"You are not a terminal, OK.\n");
    }

    input = fopen("/dev/tty", "r");
    output = fopen("/dev/tty", "w");
    if(!input || !output) {
        fprintf(stderr,"Unable to open /dev/tty\n");
        exit(1);
    }
    do {
        choice = getchoice("Please select an action", menu, input, output);
        printf("You have chosen: %c\n", choice);
    } while(choice != 'q');
    exit(0);
}

int getchoice(char *greet, char *choices[], FILE *in, FILE *out)
{
    int chosen = 0;
    int selected;
    char **option;

    do {
        fprintf(out,"Choice: %s\n",greet);
        option = choices;
        while(*option) {
            fprintf(out,"%s\n",*option);
            option++;
        }
        do {
            selected = fgetc(in);
        } while(selected == '\n');
        option = choices;
        while(*option) {
            if(selected == *option[0]) {
                chosen = 1;
                break;
            }
            option++;
        }
        if(!chosen) {
            fprintf(out,"Incorrect choice, select again\n");
        }
    } while(!chosen);
    return selected;
}
```

现在，虽然我们已经把输出重定向了，但在运行这个程序时依然能够看到菜单提示，而正常的程序输出则被放入单独的文件里去了：

```
$ menu3 > file
You are not a terminal, OK.
Choice: Please select an action
a - add new record
d - delete record
q - quit
d
Choice: Please select an action
a - add new record
d - delete record
q - quit
q
$ cat file
You have chosen: d
You have chosen: q
```

5.2 终端驱动程序和通用终端接口

通过简单的文件操作我们已经能够对终端进行一些控制，但我们经常需要更精细的终端控制能力。UNIX提供的一组操作接口使我们能够对终端驱动程序的操作行为加以控制，使我们对终端输入和终端输出的控制能力更上一层楼。

5.2.1 概述

请看下面的示意图，我们可以通过一组函数调用（即General Terminal Interface，通用终端接口，简称GTI）来控制终端，这些函数调用与对数据进行读写的函数调用是有区别的。这使得数据（读/写）接口非常干净，同时还允许对终端行为做细致的调控。这句话的意思并不是说终端的I/O接口上空无一物，相反，它将与千变万化的硬件设备打交道。

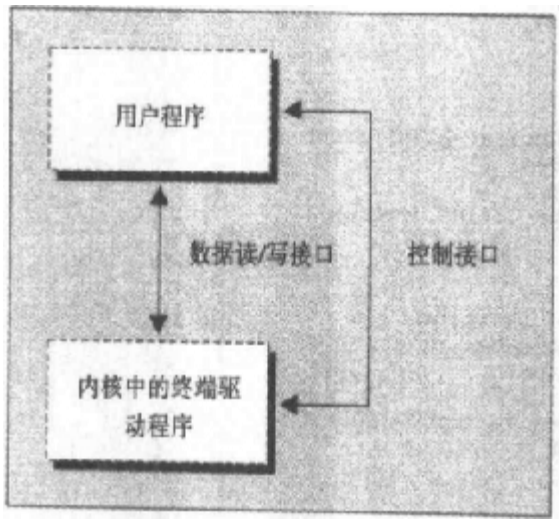


图 5-1

用UNIX的语言来说，控制接口设置了一个“行业帮规”。它使程序在定义终端驱动程序的操作行为方面拥有了极大的灵活性。

表5-1是一些我们能够控制的主要功能：

表 5-1

行编辑	是否允许用Backspace（后退）键进行编辑
缓冲	是立即读取字符，还是经过一个可配置的延迟之后再读取它们
回显	允许用户对输入内容是否回显的情况进行控制，比如读取口令字的时候
CR/LF	为输入和输出所做的映射，设置用户打印“\n”字符时应该如何处理
线速度	对使用PC电脑做为控制台来说没什么大用，但对调制解调器或连接在串行线上的终端来说就很重要了

5.2.2 硬件模型

在探讨通用终端接口的细节之前，先弄明白它将驱动的硬件模型是很重要的。

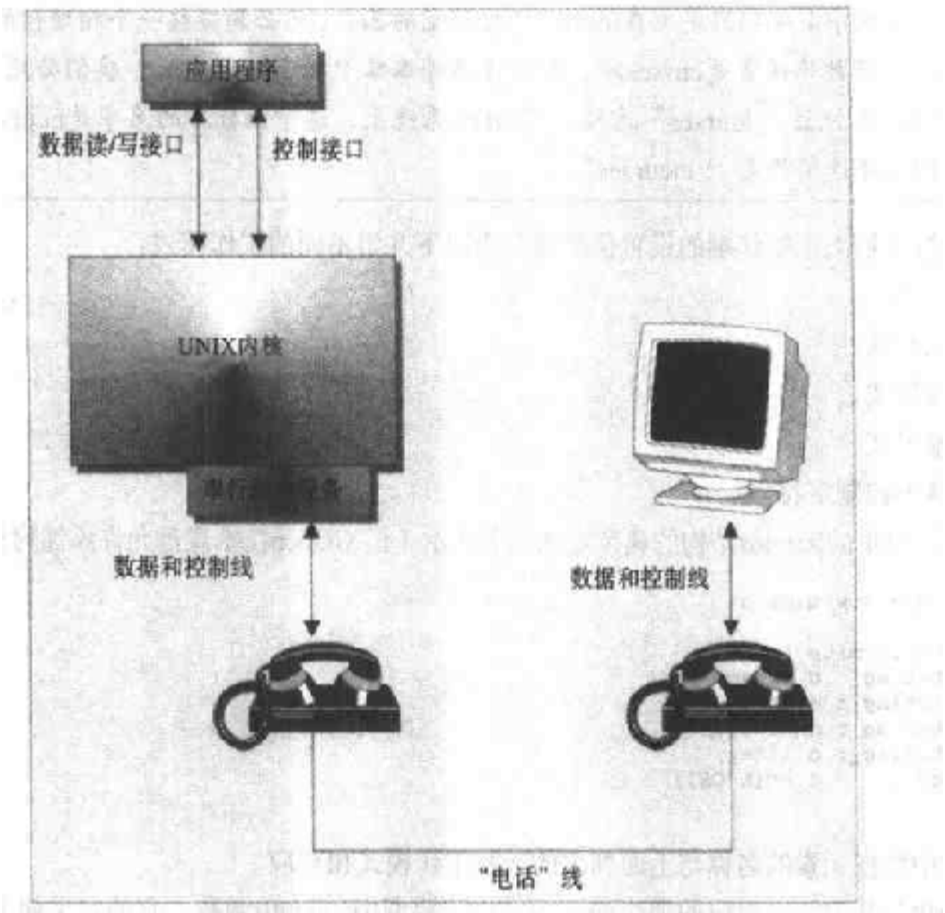


图 5-2

图中的布局结构（某些UNIX站点上的实际情况也正是如此）是让一台UNIX机器通过一个串行口连接一台调制解调器，再通过一条电话线连接到另外一台调制解调器，那个调制解调器又连接着一个终端。事实上，这正是某些小ISP（Internet service provider，因特网接入服务提供商）们使用的设置情况。我们可以把它看做是客户/服务器模型的一个“远亲”，同样都是程序运行在主机而用户工作在某个终端上。

如果读者工作在一台运行着Linux的PC电脑上，可能会认为这个模型过于复杂。但我们要说的是，因为本书的两位作者都有调制解调器，所以如果我们愿意的话，就可以按照图中的方式用两个调制解调器和一条电话线连接起来，通过一个终端仿真程序（比如minicom）在彼此的机器上运行一个远程登录作业。

使用这样一个硬件模型的好处是大多数现实世界中的情况都将只是这个最复杂的模型的一个子集。如果图中模型里少点什么，支持起来不是更容易了吗。

5.3 termios结构

termios是在POSIX技术规范里定义的标准操作接口，它与System V的termio接口很相似。终端接口由一个termois类型结构里的设置值控制，还会用到一组函数调用。这两样都定义在头文件termois.h里。

> 如果程序需要调用定义在termios.h文件里的函数，还必须链接一个辅助性的函数库。这个辅助函数库通常是curses库，所以在编译本章中的示例程序时，我们必须在编译命令行的末尾加上“-lcurses”选项。在Linux系统上，这个函数库的名字是ncurses，与之对应的编译选项将是“-lncurses”。

可以被操控来影响终端的设置值被划分为以下几组不同的工作模式：

- 输入模式。
- 输出模式。
- 控制模式。
- 本地模式。
- 特殊的控制字符。

一个最简单的termios结构的典型定义如下所示（但X/Open技术规范允许添加附加的数据域）：

```
#include <termios.h>

struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t     c_cc[NCCS];
};
```

结构中数据元素的名称与上面列出的五种工作模式相对应。

初始化与某个终端对应的那个termios结构需要调用tcgetattr函数，它的定义如下所示：

```
#include <termios.h>

int tcgetattr(int fd, struct termios *termios_p);
```

这个调用把终端接口变量的当前值写到由termios_p指向的结构里去。如果需要对这些值进行修改，可以通过tcsetattr函数来重新配置终端接口，它的定义如下所示：

```
#include <termios.h>

int tcsetattr(int fd, int actions, const struct termios *termios_p);
```

tcsetattr函数的action域控制着进行修改的方式，有三种修改方式（见表5-2），它们是：

表 5-2

TCSANOW	立即对值进行修改
TCSADRAIN	等当前输出完成后再对值进行修改
TCSAFLUSH	等当前输出完成后再对值进行修改，但还要丢弃当前可用的输入数据和尚未从read调用返回的输入数据

> 程序运行完毕后，要把终端设置恢复到程序开始运行以前的值，这一点是非常重要的。先保存这些值并在结束运行后恢复它们永远是程序的责任。

接下来我们将对模式及其相关函数调用做进一步的学习。这些模式的某些细节相当专业化，并且也很少会被用到，所以我们在此只对一些主要的功能进行介绍。如果读者需要更深入的了解，请查阅自己主机中通过man命令调出的使用手册页，也可以直接去查阅POSIX或X/Open技术规范。

在我们的第一次亲密接触里，最需要留意的是本地工作模式。我们在本章中编写的第一个应用程序出现了两个问题，授权和非授权模式就是第二个问题的解决办法。我们可以让程序等待一行输入完毕后才进行处理，也可以让它一有字符敲入就立刻行动。

5.3.1 输入模式

输入模式控制着输入数据（终端驱动程序从串行口或键盘接收到的字符）在被传递到程序之前都需要经过哪些处理。我们通过设置termios结构里c_iflag成员中的标志对它们进行控制。标志都被定义为宏，并且能够通过一个按位的OR操作组合到一起。所有的终端工作模式都采用这种办法。

可以用在c_iflag成员里的标志宏有：

- BRKINT　当在输入行上检测到一个中止条件时，产生一个中断。
- IGNBRK　忽略输入行上的中止条件。
- ICRNL　把一个接收到的回车符转换为换行符。
- IGNCR　忽略接收到的回车符。
- INLCR　把接收到的换行符转换为回车符。
- IGNPAR　忽略带奇偶校验错的字符。
- INPCK　对接收到的字符进行奇偶校验。
- PARMRK　对奇偶校验错做出标记。
- ISTRIP　把所有接收到的字符都设置为7个二进制位。
- IXOFF　激活对输入数据的软件流控制。
- IXON　激活对输出数据的软件流控制。

> 如果BRKINT和IGNBRK标志都没有被设置，输入行上的中止条件就会被读做一个NULL（0x00）字符。

用户一般不需要对输入模式进行修改，因为它的缺省值一般说来就是最合适的，所以我们也不再对它多费笔墨。

5.3.2 输出模式

输出模式控制着输出字符的处理方式，即由程序发送出来的字符在被传递到串行口或屏幕之前都需要经过哪些处理。正在如大家预料到的，其中有许多正好与输入工作模式相对立。它还有几个额外的标志，主要用于慢速终端，这些终端在处理回车等字符时要花费一定的时间。现在看来，几乎所有这些标志要不有些多余（因为现如今的终端比从前要快得多了），要不用terminfo终端性能数据库处理起来会更有效，我们在本章后面会用到terminfo终端性能数据库。

我们通过设置termios结构里c_oflag成员中的标志对输出模式进行控制。我们可以在c_iflag成员里使用的标志宏有：

- OPOST　　打开输出处理功能。
- ONLCR　　把输出中的换行符转换为回车加行进纸两个字符。
- OCRNL　　把输出中的回车符转换为换行符。
- ONOCR　　在第0列不允许对输出进行回车。
- ONLRET　　换行符等同于回车符。
- OFILL　　发送填充字符以提供延时。
- OFDEL　　把DEL而不是NULL字符用做一个填充字符。
- NLDLY　　换行符延时选择。
- CRDLY　　回车符延时选择。
- TABDLY　　TAB制表符延时选择。
- BSDLY　　Backspace后退字符延时选择。
- VTDLY　　垂直制表符延时选择。
- FFDLY　　换页符延时选择。

> 如果OPOST标志没有被设置，所有其他标志都将被忽略。

用到输出模式的情况也不是很多，所以我们也不在此做过多的讨论。

5.3.3 控制模式

控制模式控制着终端的硬件特性。我们通过设置termios结构里c_cflag成员中的标志对控制模式进行配置。其中包括以下这些标志宏：

- CLOCAL　　忽略一切调制解调器状态行。
- CREAD　　激活字符接收回执功能。
- CS5　　在发送和接收字符时使用五个二进制位。
- CS6　　在发送和接收字符时使用六个二进制位。
- CS7　　在发送和接收字符时使用七个二进制位。
- CS8　　在发送和接收字符时使用八个二进制位。

- CSTOPB　　每个字符使用两个而不是一个停止位。
- HUPCL　　关闭操作时挂断调制解调器。
- PARENB　　激活校验码的生成和检测功能。
- PARODD　　使用奇校验而不是偶校验。

> 如果HUPCL标志被置位，在终端驱动程序检测到与终端对应的最后一个文件描述符被关闭的时候它会设置调制解调器的控制线挂断电话线路。

虽然控制模式也可以用来与终端进行“对话”。但它更主要地是用在串行线连接到一台调制解调器上的情况里。一般说来，与使用termios的控制模式来改变缺省的线路连接行为相比，改变终端的配置情况更容易一些。

5.3.4 本地模式

本地模式控制着终端的许多特性。我们通过设置termios结构里c_lflag成员中的标志对本地工作模式进行配置，所用的标志宏有：

- ECHO　　激活输入字符的本地回显功能。
- ECHOE　　在接收到ERASE时执行后退、空格、后退动作组合。
- ECHOK　　在接收到KILL字符时执行行删除动作。
- ECHONL　　回显换行字符。
- ICANON　　激活授权输入处理（参见下面的说明）。
- IEXTEN　　激活实现特定的函数功能。
- ISIG　　激活信号。
- NOFLSH　　禁止清空队列的动作。
- TOSTOP　　在试图进行写操作之前给后台进程发送一个信号。

这里，最常用的两个标志是ECHO和ICANON，前者的作用是强制回显用户敲入的字符；后者的作用是在两个明显不同的接收字符处理模式之间对终端进行切换。如果ICANON标志被置位，就表示该输入行是处于授权处理模式；否则，输入行处于非授权模式。

我们将在遇到能够用在这两种工作模式里的特殊控制字符时对授权和非授权模式做详尽的解释。

5.3.5 特殊的控制字符

这是一些通过键盘上的组合键得到的字符集合，比如“Ctrl-C”等。当用户输入它们的时候，终端会采取一些特殊的动作来响应。这些由组合键输入的特殊字符通常被映射到相应的函数上去，在termios结构的c_cc数组成员里包含着映射到各个函数的特殊字符。每个字符的位置（它在数组里的下标）是由一个宏定义的，但并不要求它们必须是控制字符。

根据终端是否被设置为授权模式（即termios结构的c_lflag成员中的ICANON标志是否被置位），c_cc数组有两种差别很大的不同用法。

在这两种不同的模式里，数组下标值的使用方法有一定的重叠性，认清这一点是很重要的。既然存在这样的情况，就一定要注意别把两种模式各自的下标值弄混了。

授权模式里使用的数组下标有：

- VEOF EOF字符。
- VEOL EOL字符。
- VERASE ERASE字符。
- VINTR INTR字符。
- VKILL KILL字符。
- VQUIT QUIT字符。
- VSUSP SUSP字符。
- VSTART START字符。
- VSTOP STOP字符。

非授权模式里使用的数组下标有：

- VINTR INTR字符。
- VMIN MIN值。
- VQUIT QUIT字符。
- VSUSP SUSP字符。
- VTIME TIME值。
- VSTART START字符。
- VSTOP STOP字符。

这些特殊字符和非授权模式里的MIN与TIME值对输入字符的高级处理功能来说意义非常重要，我们对它们做一番认真研究。

1. 字符

表 5-3

字 符	说 明
INTR	这个字符将引起终端驱动程序向与终端相连接的进程发送一个SIGINT信号。我们将在第10章对信号做进一步介绍
QUIT	这个字符将引起终端驱动程序向与终端相连接的进程发送一个SIGQUIT信号
ERASE	这个字符将使终端驱动程序删除输入行中的最后一个字符
KILL	这个字符将使终端驱动程序删除整个输入行
EOF	这个字符将使终端驱动程序把输入行上的全部字符传递给读取输入的应用程序。如果输入行是空的，read调用将返回零个字符，就好象read试图读文件尾一样
EOL	这个字符就象是一个输入行结束字符，和经常使用的换行符效果相同
SUSP	这个字符将引起终端驱动程序向与终端相连接的进程发送一个SIGSUSP信号。如果读者的UNIX支持作业控制功能，当前应用程序就会被挂起
STOP	这个字符对应的动作是“截流”，即不再向终端输出任何字符。它被用来支持XON/XOFF流控制，通常被设置为ASCII的XOFF字符，“Ctrl-S”组合键
START	这个字符重新开启被STOP字符截流的输出，通常被设置为ASCII的XON字符

2. TIME和MIN值

TIME值和MIN值只能用在非授权模式里，两者结合起来对输入的读取操作进行控制。此外，如果两者一起使用，还能够控制程序试图读与某个终端相关联的一个文件描述符时将会发生怎样的事情。

这两个值能够构成四种情况，它们是：

MIN = 0 和 TIME = 0

在这种情况下，read调用将立刻返回。如果有等待处理的字符，它们就会被返回；如果没有可供处理的字符，read就会返回零，并且没有任何字符被读取。

MIN = 0 和 TIME > 0

在这种情况下，只要字符一出现，或者经过TIME个十分之一秒的时间间隔之后，read调用就会立刻返回。如果因为超时而没有读到任何字符，read将返回零。否则它将返回读入的字符个数。

MIN > 0 和 TIME = 0

在这种情况下，read将等到至少有MIN个字符可以被读取时才开始进行读操作，返回的也将是MIN个字符。到达文件尾时返回零。

MIN > 0 和 TIME > 0

这是最复杂的情况。当read被调用的时候，它会等待接收一个字符，在接收到第一个字符及后续的各个字符时，会启动一个字符时间间隔计时器（如果计时器已经启动，就重启它）。read会在MIN个字符准备好或者两个字符之间的时间间隔超过TIME个十分之一秒时返回。它可以用来区分“单独按下了Escape键”和“功能键转义序列的开始”这两种情况，是很有用的功能。但网络通讯或高处理器负荷会把这种精细的时间安排清除掉，这一点请大家多多注意。

通过设置非授权模式与使用MIN值和TIME值，程序可以逐个字符地对输入进行处理。

3. 从shell里访问终端的工作模式

如果读者在使用shell的时候想查看一下当前的termios设置值，可以用下面的命令得到一个清单：

```
$ stty -a
```

在我们的Linux系统上（它对标准的termios结构做了些扩展）这个命令的输出如下所示：

```
speed 38400 baud; rows 25; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon ixoff
-iuclc -ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon -iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
-echoctl echoke
```

从上面这个清单里，我们可以看到EOF字符对应着“Ctrl-D”组合键，并且激活回显功能。在读者做终端控制练习的时候，很容易把终端搞成一个非标准状态，再要想使用它可就费劲了。下面几个办法能够帮你摆脱这种困境。

使用下面的命令——这要求你的stty命令版本支持这种用法：

```
$ stty sane
```

如果你把回车键与换行符（用来结束命令行）之间的映射关系搞乱了，就必须先敲入“stty sane”，然后按下“Ctrl-J”组合键（这个组合键对应着换行符），而不是回车键。

第二种办法是先用“stty -g”命令把stty的当前设置值都保存起来，在必要的时候再从文件里把原始设置值读出来进行恢复。命令行上的具体命令如下所示：

```
$ stty -g > save_stty
..
<experiment with settings>
..
$ stty $(cat save_stty)
```

最后一条stty命令还是必须用“Ctrl-J”组合键来代替回车键。在shell脚本程序里也可以使用同样的技巧：

```
save_stty="$(stty -g)"
<alter stty settings>
stty $save_stty
```

如果这些招数都不管用，还有第三个办法：到另外一台终端去，用ps命令找到被弄乱了的shell，再用“kill HUP <process id>”命令强制那个shell结束运行。因为系统总是会在给出登录提示符之前重置stty参数，所以你将能够在stty的初始设置情况里重新登录上机。

4. 从命令提示符对终端的工作模式进行设置

stty命令还可以用来从命令行提示符处直接对终端的工作模式进行设置。

比如说，如果我们想在自己的shell脚本程序里把终端设置成“能够读取单字符输入”这样一种工作模式，就需要关闭授权模式，把MIN设置为“1”，把TIME设置为“0”。这个命令如下所示：

```
$ stty -icanon min 1 time 0
```

既然已经把终端设置成“立即读字符”了，我们再运行我们的第一个程序menu1来看看会是怎样的情况。你会发现它是按我们预想的路线前进的。

我们还可以对第2章里的口令字检查程序加以改进，在提示输入口令字之前把回显功能关闭掉。这个命令如下所示：

```
$ stty -echo
```

注意在实践之后用“stty echo”命令把回显功能设置回开启状态。

5.3.6 终端的速度

termios结构提供的最后一个功能是对线速度进行操控。termios结构里并没有与终端的数据传输速度对应的数据成员，它是通过函数调用对它进行设置的。输入和输出速度是分开处理的。

下面是四个有关函数的定义：

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *);
speed_t cfgetospeed(const struct termios *);
int cfsetispeed(struct termios *, speed_t speed);
int cfsetospeed(struct termios *, speed_t speed);
```

注意，这些函数是对termios结构进行操作的，不直接作用于通信端口。这就意味着要想设置一个新的速度，就必须先用tcgetattr函数读出当前设置值，用上面四个函数之一设置好新速度，再用tcsetattr函数写回termios结构。只有在tcsetattr函数结束之后，线速度才会改变。

上面四个函数中的speed参数允许使用各种各样的值，其中最重要的是：

- B0　　挂断终端。
- B1200　　1200波特。
- B2400　　2400波特。
- B9600　　9600波特。
- B19200　　19200波特。
- B38400　　38400波特。

有关标准没有定义大于38400的速度，也没有支持串行口工作在这个速度以上的标准方法。

> 包括Linux在内的许多系统为了能够选取到更高的线速度而增加定义了“B57600”、“B115200”和“B230400”。如果读者使用的是Linux的一个早期版本，里面没有对这些常数进行定义的话，可以通过setserial命令获得57600和115200这样的非标准速度。在这种情况里，必须先选择“B38400”之后才能使用这些非标准速度。这两种办法都不具备可移植性，所以在使用它们的时候要考虑清楚。

5.3.7 其他功能函数

在终端控制方面还有一些其他的函数。它们直接对文件描述符进行操作，不需要读写termios结构。下面是它们的定义：

```
#include <termios.h>

int tcdrain(int fd);
int tcflow(int fd, int flowtype);
int tcflush(int fd, int in_out_selector);
```

这些函数的功能如下所示：

- tcdrain的作用是让调用者程序等待排好队的输出数据全部发送完毕。
- tcflow用来挂起或重新开始输出。
- tcflush可以用来清空输入、输出或输入和输出。

在介绍了这么多关于termios结构的知识以后，我们来看几个实用的例子。其中最简单的大概要算读取口令字时禁止回显功能了。关闭“ECHO”标志就可以做到这一点。

动手试试：使用termios结构的口令字核查程序

1) 我们的口令字核查程序password.c以下面这些定义开始：

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

#define PASSWORD_LEN 8

int main()
{
    struct termios initialrsettings, newrsettings;
    char password[PASSWORD_LEN + 1];
```

2) 接下来，用一条语句读出标准输入设备的当前设置值，把这些值保存到我们刚才创建的termios结构里去：

```
tcgetattr( fileno(stdin), &initialrsettings );
```

3) 给这些原始设置值做一份拷贝以备对它们进行替换。在newrsettings里关闭“ECHO”标志，然后提示用户输入口令字：

```
newrsettings = initialrsettings;
newrsettings.c_lflag &= ~ECHO;

printf("Enter password: ");
```

4) 接下来，用newrsettings中的值设置终端属性并读取口令字。最后，把终端属性重新设置回它们原来的初始值并输出读到的口令字，让刚才的努力“白费”了。

```
    if(tcsetattr(fileno(stdin), TCSAFLUSH, &newrsettings) != 0) {
        fprintf(stderr,"Could not set attributes\n");
    }
    else {
        fgets(password, PASSWORD_LEN, stdin);
        tcsetattr(fileno(stdin), TCSANOW, &initialrsettings);
        fprintf(stdout, "\nYou entered %s\n", password);
    }
    exit(0);
}
```

操作注释：

```
$ password
Enter password:
You entered hello

$
```

在这个例子里，单词“hello”是用户输入的口令字，但它可不是由“Enter password:”提示符回显出来的。在用户按下回车键之前没有产生任何输出。

我们很谨慎地只修改了需要修改的标志，使用的是“X &= ~FLAG”语法结构（它的作用是清除变量X里由FLAG定义的二进制位）。如果需要，我们可以用“X |=FLAG”语法结构对FLAG定义的二进制位进行置位，当然在上面的例子里并不需要这么做。

在对终端属性进行设置的时候，我们通过TCSAFLUSH丢弃掉在此之前的所有输入。这是个培养用户好习惯的好办法，教育用户在回显功能关闭之前别开始输入自己的口令字。在这个程

序结束之前，我们还恢复了终端的原始设置。

termios结构另一个常见用法是把终端设置为这样一种状态：我们可以在每一个字符被敲入的时候立刻读取它们。具体做法是：关闭授权模式，再结合使用MIN值和TIME值进行设置。

动手试试：读取每一个字符

1) 根据新学到的知识，我们对我们的菜单程序做些修改。下面的代码与password.c有很多相似之处，把它们插入到menu3.c文件里得到一个新的程序menu4.c。在新程序的一开始，我们要加上一个新的头文件：

```
#include <stdio.h>
#include <unistd.h>
#include <termios.h>
```

2) 接下来，声明几个用在main函数中的新变量：

```
int choice = 0;
FILE *input;
FILE *output;
struct termios initial_settings, new_settings;
```

3) 在调用getchoice函数之前必须对终端的属性进行修改，我们加上如下所示的这些语句：

```
    fprintf(stderr, "Unable to open /dev/tty\n");
    exit(1);
}
tcgetattr(fileno(input),&initial_settings);
new_settings = initial_settings;
new_settings.c_lflag &= ~ICANON;
new_settings.c_lflag &= ~ECHO;
new_settings.c_cc[VMIN] = 1;
new_settings.c_cc[VTIME] = 0;
if(tcsetattr(fileno(input), TCSANOW, &new_settings) != 0) {
    fprintf(stderr,"could not set attributes\n");
}
```

4) 在退出之前还要把终端属性设置回它们的初始值：

```
    do {
        choice = getchoice("Please select an action", menu, input, output);
        printf("You have chosen: %c\n", choice);
    } while (choice != 'q');
    tcsetattr(fileno(input),TCSANOW,&initial_settings);
    exit(0);
}
```

注意，因为我们现在处于非授权模式中，“CR”和“LF”字符之间的缺省映射不复存在，所以我们必须自行对回车字符“\n”进行检查和处理了。如下所示：

```
do {
    selected = fgetc(in);
} while (selected == '\n' || selected == '\r');
```

5) 到了这一步，如果用户在我们的程序里按下“Ctrl-C”组合键，程序就会立刻退出。我们可以禁止这些特殊字符的处理功能。做法很简单，把本地模式下的ISIG标志清除掉就行了。在main函数里加上这样一条语句：

```
new_settings.c_lflag &= ~ISIG;
```

操作注释：

把这些修改都放到我们的菜单程序里之后，只要一敲入字符就能得到程序的响应，而敲入的字符并不会回显在屏幕上：

```
$ menu4
Choice: Please select an action
a - add new record
d - delete record
q - quit
You have chosen: a
Choice: Please select an action
a - add new record
d - delete record
q - quit
You have chosen: q
$
```

如果我们按下“Ctrl-C”组合键，它会被直接传递给程序，而程序会认为这是一个不正确的菜单选择。

5.4 终端的输出

通过前面的学习，我们已经知道能够通过termios结构控制来自键盘的输入，如果对程序输出到屏幕上的内容也能有同等的控制权就更好了。在这一章的开始，我们是用printf语句把字符输出到屏幕上的，但还不能把输出数据放到屏幕上的特定位置。

5.4.1 终端的类型

许多UNIX系统都要通过终端才能被使用，而现如今的许多“终端”实际上都已经是一些运行着一个终端仿真程序的PC个人电脑了。从其历史发展过程来看，曾经有大量的制造厂家生产过大量各种型号的终端。虽然它们都使用escape转义序列（一个以escape字符打头的字符串）来控制光标的位置以及终端的其他属性——比如黑体和闪烁等，但在具体实现手段方面并没有什么统一的标准。某些陈旧的终端还有一些与众不同的卷屏方式；某些终端上与Backspace键对应的动作删除光标前面的字符，而有的终端就不这样做，可以说是千差万别。

> escape转义序列有一个ANSI标准，它以数字设备公司（Digital Equipment Corpration，即人们熟知的DEC公司）的VT系列终端所使用的转义序列为基础，但并不完全一致。许多PC电脑上使用的终端程序提供了对标准终端的仿真功能，模仿的对象通常是VT100、VT200或ANSI终端，但也有模仿其他终端的。

对那些希望能够在多种终端型号上运行并对屏幕输出进行控制的程序来说，终端多样性是一个很重大的问题。举例来说吧，如果想把光标移动到一行的末尾，在ANSI终端上要使用转义序列“Escape-[-A”；而在ADM-3a终端（多年前它曾经是一种很常见的终端）上只需使用一个控制字符“Ctrl-K”。

一个UNIX系统可能连接的终端的类型实在是太多了，因此，要想编写出一个能够应付各种

不同终端的程序也实在是太困难了，这样的程序必须为每一种终端准备好相应的源代码。

所以，terminfo工具包的出现也就是顺理成章的事情了。程序不再需要为每一种终端而进行裁剪，它们可以通过一个终端型号数据库查找到正确的设置信息。在大多数现代的UNIX系统上，这个数据库已经与另外一个名为curses的函数库软件包集成在一起，我们将在下一章遇到它。

在Linux系统上，curses软件包通常被实现为ncurses数据库。在编写程序时要包括上提供了对terminfo函数进行模型化定义的头文件ncurses.h才行，而terminfo函数本身实际是在它们自己的头文件term.h里定义的。更正确地说，一般情况总是这样的。对比较新一些的Liux版本来说，terminfo和ncurses有逐渐融合的趋势，许多程序在使用terminfo函数的时候也必须包括上ncurses头文件。

5.4.2 确定终端类型的方法

UNIX的环境里有一个名为TERM的变量，它被设置为当前使用中的终端的类型。它通常是由系统在用户登录时自动设置的。系统管理员可以为与系统直接连接的终端设置一个缺省的终端类型；而远程用户和网络用户则需要在系统管理员安排的提示符处选择自己的终端类型。TERM变量的值可以通过telnet进行协调，由rlogin功能传递。

用户可以通过下面的命令对shell进行查询，看看自己正在使用的终端在系统眼里到底是哪一种类型：

```
$ echo $TERM
xterm
$
```

在这个例子里，shell是从一个名为xterm的程序里开始运行的，这是一个X窗口系统下的终端仿真器。

terminfo工具包里包含着一个由大量终端的性能指标和转义序列等信息构成的数据库，并且还为它们的使用准备了一个统一的程序设计接口。有了这个工具包，程序设计工作得到了极大的简化，并且能够随着数据库的扩展适应未来的终端，对不同类型的终端的支持不再需要由应用程序自己来提供了。

终端的terminfo性能指标通常被描述为属性。它们被保存在一系列经过编译的terminfo文件里，而这些文件一般被保存在/usr/lib/terminfo或/usr/local/terminfo子目录里。每个终端（也包括许多种打印机，它们也可以在terminfo中定义）都有一个定义其性能指标和功能被访问手段的文件。为了避免出现子目录过大的情况，实际文件都被保存在下级子目录里，下级子目录的名字就是终端类型的第一个字母。比如说，VT100型终端的定义就放在文件“.../terminfo/v/vt100”里。

每个终端类型对应一个terminfo文件，文件格式都是可读的源代码。然后再用tic命令把它们编译为一个更紧凑和高效的格式以方便应用程序的使用。但奇怪的是，X/Open技术规范提到了源代码和编译格式的定义，却没有提到从源代码转换为编译格式的tic命令。我们可以用*infocmp*程序给出terminfo编译数据项的可读版本。

下面是VT100终端的一个示例性terminfo文件：

```
$ infocmp vt100
vt100|vt100-am|dec vt100 (w/advanced video),
 am, mir, msgr, xenl, xon,
 cols#80, it#8, lines#24, vt#3,
 acsc=``aaffgghjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
 bel=^G, blink=\E[5m$<2>, bold=\E[1m$<2>,
 clear=\E[H\E[J$<50>, cr=\r, csr=\E[%i%p1%d;%p2%dr,
 cub=\E[%p1%dD, cub1=\b, cud=\E[%p1%dB, cud1=\n,
 cuf=\E[%p1%dC, cuf1=\E[C$<2>,
 cup=\E[%i%p1%d;%p2%dH$<5>, cuu=\E[%p1%dA,
 cuu1=\E[A$<2>, ed=\E[J$<50>, el=\E[K$<3>,
 el1=\E[1K$<3>, enacs=\E(B\E)0, home=\E[H, ht=\t,
 hts=\EH, ind=\n, ka1=\EOq, ka3=\EOs, kb2=\EOr, kbs=\b,
 kc1=\EOp, kc3=\EOn, kcub1=\EOD, kcud1=\EOB,
 kcuf1=\EOC, kcuu1=\EOA, kent=\EOM, kf0=\EOy, kf1=\EOP,
 kf10=\EOx, kf2=\EOQ, kf3=\EOR, kf4=\EOS, kf5=\E0t,
 kf6=\E0u, kf7=\E0v, kf8=\E0l, kf9=\E0w, rc=\E8,
 rev=\E[7m$<2>, ri=\EM$<5>, rmacs=^O, rmkx=\E[?1l\E>,
 rmso=\E[m$<2>, rmul=\E[m$<2>,
 rs2=\E>\E[?3l\E[?4l\E[?5l\E[?7h\E[?8h, sc=\E7,
 sgr=\E[0%?%p1%p6%|%t;1%;%?%p2%t;4%;%?%p1%p3%|%t;7%;%?%p4%t;5%;m%?%p9%t^N%e^O%;,
 sgr0=\E[m^O$<2>, smacs=^N, smkx=\E[?1h\E=,
 smso=\E[1;7m$<2>, smul=\E[4m$<2>, tbc=\E[3g,
```

每个terminfo定义由三种数据项组成，这三种数据项叫做“性能名”（capname），它们分别定义了终端的一种性能。

布尔性能指标比较容易理解，它告诉我们某个终端是否支持某个特定的功能。比如说，如果某个终端支持XON/XOFF流控制，就能在上面的清单里看到布尔性能指标“XON”；如果给定的“光标左置”命令能够在光标位于第0列的时候把光标移动到最右边的那一列去，就能在清单里看到布尔性能指标“cub1”。

数值性能指标定义一些有关的尺寸长度数字，比如lines定义的是屏幕显示的行数，cols定义的是屏幕显示的列数。数字和性能指标名称之间用一个井字号“#”隔开。所以，如果想定义一个有80列24行显示范围的终端，可以写为“cols#80, lines#24”。

字符串性能指标稍微复杂一些。它们用来定义两种泾渭分明的性能：用来访问终端功能的输出字符串和用户按下特定按键（通常是功能键或数字小键盘上的特殊键）时终端接收到的输入字符串。有的字符串性能指标很简单，比如“el”表示“删除到这一行的末尾”。在VT100终端上，用来完成这一操作的escape转义序列是“Esc-[-K”。在terminfo源代码格式里，这个转义序列被写为“el=\E[K”。

特殊键的定义也采用同样的办法。举例来说，VT100终端上的功能键F1对应发送的是转义序列“Esc-O-P”。它被定义为“kf1=\EOP”。

当转义序列本身还要带有参数的时候，情况会稍微复杂一些。大多数终端可以把光标移动到给定的行列位置。很明显，为光标可能停留的每个位置都定义一个不同的性能指标是不现实的，解决办法只能是一个带参数的通用性字符串，然后在使用这个字符串的时候通过它的参数值定义来给出光标的插入位置。举例来说，VT100终端会通过转义序列“Esc-[-<row>-;-<col>-H”把光标移动到指定位置。在terminfo的源代码格式里，这个转义序列被写做相当复杂的“cup=\E[%i%p1%d;%p2%dH$<5>”。

它的意思是：

- \E　　发送Escape字符。

- [发送“[”字符。
- %i 增加参数的值。
- %pl 把第一个参数放到堆栈上。
- %d 把堆栈上的数字输出为一个十进制数。
- ; 发送“;”字符。
- %p2 把第二个参数放到堆栈上。
- %d 把堆栈上的数字输出为一个十进制数。
- H 发送“H”字符。

这种写法看起来相当复杂。但它允许人们以统一固定的顺序使用这些参数，与终端希望它们最终出现在escape转义序列里时的顺序互不干扰。“%i”的作用是增加参数的值，它是必不可少的，因为光标地址的标准定位方法是把屏幕的左上角看做是(0, 0)，而VT100把这个位置定义为地址(1, 1)。最后面的“$<5>”表示需要延迟一段与输出五个字符相当的时间，终端就在这段延时里对光标移动操作进行处理。

> 我们可以对许许多多的性能指标进行定义，但幸运的是大多数UNIX系统已经把大多数终端都定义好了。如果读者需要增加一个新终端，就能在使用手册的terminfo项目下找到一个完整的性能指标清单。比较好的办法是：以与新终端型号接近的一个终端为出发点，把新终端定义为那个现有终端的变体；或者你可以逐项地对新终端的性能指标进行定义，按要求修改它们。
>
> man命令给出的使用手册页以外的标准参考资料是O'Reilly出版社出版的“Termcap and Terminfo”（《Termcap和Terminfo大全》），国际书号是ISBN 0-937175-22-6。

5.4.3 terminfo的使用方法

现在我们已经知道如何对终端的性能指标进行定义了，下面我们来看看怎样才能利用这些性能指标实现我们的想法。在使用terminfo的时候，我们要做的第一件事情是调用函数setupterm设置终端的型号。这个操作会根据当前终端的型号初始化一个TERMINAL结构。随后，我们就可以查看这个终端的性能指标和使用它的功能了。setupterm函数的调用方法如下所示：

```
# include <term.h>
int setupterm(char *term, int fd, int *errret);
```

setupterm库函数把当前终端设置为由参数term指定的型号。如果term是一个空指针，就使用环境变量TERM的值。往终端写数据需要使用一个文件描述符，它被传递为参数fd。如果函数的执行结果不是一个空指针，就会被保存到errret指向的一个整数变量里去。它的含义如下：

- -1 terminfo数据库不存在。
- 0 terminfo数据库中没有匹配数据项。
- 1 成功。

setupterm函数在成功时返回常数“OK”，失败时返回“ERR”。如果errret被设置为空指针，setupterm就会在失败时给出一条诊断信息并直接退出执行，就象下面的例子那样：

```
#include <stdio.h>
#include <term.h>
#include <ncurses.h>

int main()
{
    setupterm("unlisted",fileno(stdout),(int *)0);
    printf("Done.\n");
    exit(0);
}
```

在读者系统上运行此程序的结果可能与这里给出的不太一样，但意思应该是很明白的。你在屏幕上是看不到“Done.”的，因为setupterm在它执行失败的时候会使程序直接退出运行。

```
$ cc -o badterm badterm.c -I/usr/include/ncurses -lncurses
$ badterm
'unlisted': unknown terminal type.
$
```

请留意例子中的编译命令行：在这台Linux系统上，ncurses头文件被保存在子目录/usr/include/ncurses里了，所以我们要用“-I”选项明确地告诉编译器该到那里去查找库文件。有的Linux系统已经把curses库放到标准地点上了，在这些系统上，我们只需要在程序里包括上ncurses.h头文件，再在命令行上用“-lcurses”指定库文件就行了。

来看看我们的菜单选择函数，我们想让它能够清屏，能够在屏幕上移动光标，能够在屏幕的不同位置写数据。在成功调用setupterm函数之后，我们就可以通过下面三个函数来访问terminfo性能指标了，每个函数对应着一个性能指标类别：

```
#include <term.h>

int tigetflag(char *capname);
int tigetnum(char *capname);
char *tigetstr(char *capname);
```

函数tigetflag、tigetnum和tigetstr分别返回terminfo性能中的布尔指标、数值指标和字符串指标。在失败的时候（比如某个性能指标不存在的时候），tigetflag将返回“-1”，tigetnum将返回“-2”，而tigetstr将返回“(char *) -1”。

我们先利用terminfo性能指标数据库查出终端字符显示区的大小，具体做法是用下面这个sizeterm.c程序检索出终端的cols和lines性能指标值：

```
#include <stdio.h>
#include <term.h>
#include <ncurses.h>

int main()
{
    int nrows, ncolumns;

    setupterm(NULL, fileno(stdout), (int *)0);
    nrows = tigetnum("lines");
    ncolumns = tigetnum("cols");
    printf("This terminal has %d columns and %d rows\n", ncolumns, nrows);
    exit(0);
}
```

```
$ echo $TERM
vt100
$ sizeterm
This terminal has 80 columns and 24 rows
$
```

如果我们在工作站的一个窗口里运行这个程序，得到的答案将反应出当前窗口的字符显示区尺寸，如下所示：

```
$ echo $TERM
xterm
$ sixeterm
This terminal has 88 columns and 40 rows
$
```

如果我们用tigetstr函数来检索xterm终端的光标移动性能cup（即英文“cursor motion capability”的字头缩写）指标，就会得到这样一个参数化结果“\E[%p1%d;%p2%dH”。

这个性能指标需要两个参数，即光标将要移到那里去的行号和列号。这两个坐标都是从0开始计算的，(0,0)表示屏幕的左上角。

用tparm函数可以把性能指标中的参数替换为实际的数值。一次最多可以替换九个参数，它返回的是一个可用的escape转义序列。

```
#include <term.h>

char *tparm(char *cap, long p1, long p2, ..., long p9);
```

向终端输出控制字符串

用tparm函数构造好终端的转义序列之后，我们还必须把它送往终端。要想正确地完成这一工作，就不能简单地用printf函数向终端发送字符串。必须使用系统为我们提供的下面这几个函数才行，它们能安排好终端完成某项操作所必要的延时。这些函数是：

```
#include <term.h>

int putp(char *const str);
int tputs(char *const str, int affcnt, int (*putfunc)(int));
```

如果成功，putp将返回“OK”；失败时返回“ERR”。putp函数以终端控制字符串为参数，把它发送到stdout去。

如果想把光标移动到第五行、第30列，我们需要使用如下所示的这段代码：

```
char *cursor;
char *esc_sequence;
cursor = tigetstr("cup");
esc_sequence = tparm(cursor,5,30);
putp(esc_sequence);
```

tputs函数是为不能通过stdout访问终端的情况准备的，它允许我们指定一个用来输出控制字符串的函数。它返回的是用户指定的字符串输出函数putfunc的返回结果。affcnt参数的用途是限制这一变化所影响的输出行数，它一般被设置为“1”。用来输出控制字符串的函数其参数和返回值必须与putchar函数完全一致。事实上，“put(string)”就相当于“tputs(string, 1, putchar)”调用。我们稍后会看到tputs使用用户指定的输出函数进行工作的例子。

注意，Linux的某些老版本把tputs函数的最后一个参数定义为“int (*putfunc)(char)”，如果读者使用的是这样的版本，就要在下面的“动手试试”里修改char_to_terminal函数的定义。

读者在使用手册页里查阅tparm和终端性能指标的有关资料时，可能会看到一个名

为tgoto的函数。显而易见，用这个函数来移动光标要更容易一些，但我们并没有使用它。原因是在1997年版的X/Open技术规范（Single UNIX Specification Version 2）里没有包括这些个函数。所以我们建议大家在新程序里最好不要使用这些函数。

往我们的菜单选择函数里添加屏幕处理功能的准备工作已经基本就绪。就剩下清屏这件事还没提到，这个操作很简单，使用性能指标clear就行。clear会把光标放到屏幕的左上角，但有些终端不支持这种用法。在这种情况下，我们可以先把光标放到屏幕的左上角，再用“删除到显示内容尾”命令ed完成操作。

把上面这些内容总结在一起，我们来编写菜单程序示例的最终版本screenmenu.c，我们将通过这个程序把菜单选项“画”在屏幕上供用户选择。

动手试试：终端控制总动员

我们将重新编写menu4.c程序里的getchoice函数，由此终端将完全处在我们的掌握之中。在下面的程序清单里，我们省略了main函数，因为不需要对它进行修改。而与menu4.c不一致的地方都用灰色块标记出来了。

```
#include <stdio.h>
#include <unistd.h>
#include <termios.h>
#include <term.h>
#include <curses.h>

static FILE *output_stream = (FILE *)0;

char *menu[] = {
    "a - add new record",
    "d - delete record",
    "q - quit",
    NULL,
};

int getchoice(char *greet, char *choices[], FILE *in, FILE *out);
int char_to_terminal(int char_to_write);

int main()
{
...
}

int getchoice(char *greet, char *choices[], FILE *in, FILE *out)
{
    int chosen = 0;
    int selected;
    int screenrow, screencol = 10;

    char **option;
    char *cursor, *clear;

    output_stream = out;

    setupterm(NULL,fileno(out), (int *)0);
    cursor = tigetstr("cup");
    clear = tigetstr("clear");

    screenrow = 4;
    tputs(clear, 1, (int *) char_to_terminal);
    tputs(tparm(cursor, screenrow, screencol), 1, char_to_terminal);
    fprintf(out, "Choice: %s", greet);
```

```
    screenrow += 2;
    option = choices;
    while(*option) {
        tputs(tparm(cursor, screenrow, screencol), 1, char_to_terminal);
        fprintf(out,"%s", *option);
        screenrow++;
        option++;
    }

    do {
        selected = fgetc(in);
        option = choices;
        while(*option) {
            if(selected == *option[0]) {
                chosen = 1;
                break;
            }
            option++;
        }

        if(!chosen) {
            tputs(tparm(cursor, screenrow, screencol), 1, char_to_terminal);
            fprintf(out,"Incorrect choice, select again\n");
        }
    } while(!chosen);
    tputs(clear, 1, char_to_terminal);
    return selected;
}

int char_to_terminal(int char_to_write)
{
    if (output_stream) putc(char_to_write, output_stream);
    return 0;
}
```

操作注释：

经过重新编写的getchoice函数实现的是前面各示例程序所实现的同一个菜单，但我们对它的屏幕输出部分做了进一步加工，使它能够充分利用上terminfo性能指标。在提示用户进行下一次选择之前会有一个清屏操作，如果你想在清屏之前让“You have chosen:”信息多停留一会儿，可以在main函数里增加一个sleep调用。如下所示：

```
do {
        choice = getchoice("Please select an action", menu, input, output);
        printf("\nYou have chosen: %c\n", choice);
        sleep(1);
    } while (choice != 'q');
```

这个程序里的最后一个函数是char_to_terminal，它里面有一个putc函数。对这个函数的介绍请参考第3章内容。

为了让这一章有个圆满的结局，我们再给大家介绍一个对击键动作进行检测处理的程序示例。

5.5 检测键盘输入

曾经为MS-DOS编写过程序的人们经常会在UNIX里寻找一个与kbhit函数对等的东西，这个函数会在没有实际进行读操作的前提下检测某个键是否被按过了。寻找它的努力是徒劳的，因为UNIX里没有与它直接对应的东西。UNIX程序员对此并不在意，这是因为UNIX是一个多任务操作系统，它的程序很少忙于等待某个事件的发生。kbhit最主要的用途就是等待击键动作的发生，这在UNIX里是很少见的事。

但如果需要移植MS-DOS下的程序，有个kbhit函数的翻版用着也会挺方便的。我们可以用非授权输入模式来实现它。

动手试试：键盘击键监测程序kbhit

1) 程序的开始是标准的程序头和为终端设置值而定义的几个结构。peek_character变量用来控制对击键动作的检测。然后我们对程序里将要用到的几个函数进行了预定义。

```
#include <stdio.h>
#include <termios.h>
#include <term.h>
#include <curses.h>
#include <unistd.h>
static struct termios initial_settings, new_settings;
static int peek_character = -1;
void init_keyboard();
void close_keyboard();
int kbhit();
int readch();
```

2) main函数先调用init_keyboard配置好终端，然后每隔一秒循环调用一次kbhit检测有无击键动作。如果按下的是“q”键，就调用close_keyboard恢复正常的行为并退出程序。

```
int main()
{
    int ch = 0;

    init_keyboard();
    while(ch != 'q') {
        printf("looping\n");
        sleep(1);
        if(kbhit()) {
            ch = readch();
            printf("you hit %c\n",ch);
        }
    }
    close_keyboard();
    exit(0);
}
```

3) init_keyboard和close_keyboard分别在程序的首尾对终端进行配置。

```
void init_keyboard()
{
    tcgetattr(0,&initial_settings);
    new_settings = initial_settings;
    new_settings.c_lflag &= ~ICANON;
    new_settings.c_lflag &= ~ECHO;
    new_settings.c_lflag &= ~ISIG;
    new_settings.c_cc[VMIN] = 1;
    new_settings.c_cc[VTIME] = 0;
    tcsetattr(0, TCSANOW, &new_settings);
}

void close_keyboard()
{
    tcsetattr(0, TCSANOW, &initial_settings);
}
```

4) 检查有无击键动作的kbhit函数：

```
int kbhit()
{
```

```
    char ch;
    int nread;

    if(peek_character != -1)
        return 1;
    new_settings.c_cc[VMIN]=0;
    tcsetattr(0, TCSANOW, &new_settings);
    nread = read(0,&ch,1);
    new_settings.c_cc[VMIN]=1;
    tcsetattr(0, TCSANOW, &new_settings);

    if(nread == 1) {
        peek_character = ch;
        return 1;
    }
    return 0;
}
```

5) 按键代表的字符被下一个函数readch读取。它会把peek_character重置为“-1”以进入下一个循环。

```
int readch()
{
    char ch;

    if(peek_character != -1) {
        ch = peek_character;
        peek_character = -1;
        return ch;
    }
    read(0,&ch,1);
    return ch;
}
```

当我们运行这个程序的时候，我们将看到如下所示的输出情况：

```
$ kbhit
looping
looping
looping
you hit h
looping
looping
looping
you hit d
looping
you hit q
$
```

操作注释：

init_keyboard函数把终端配置成“等待读取一个字符才能返回”的工作模式（MIN=1，TIME=0）。kbhit把这个行为改变为立刻检查输入并返回（MIN=0，TIME=0）。在程序退出之前还要恢复终端最初的配置情况。

注意：如果有按键被按下，那么在kbhit函数里我们实际上已经把它所代表的字符读进来了，但这个字符此时只能保存在一个本地变量里，只有在下一步由readch函数“读”字符的时候才返回它。

伪终端

许多UNIX系统，包括Linux，都有一个名为伪终端的功能。这些设备的行为与我们在这一

章里使用的终端非常相似，惟一的区别就是伪终端没有实际对应的硬件。它们可以用来提供一个以终端形式访问其他程序的操作接口。

比如说，我们可以通过伪终端让两个象棋程序厮杀在一起，而它们实际上是按与人类棋手通过一个终端来过招的情况设计的。这需要有一个应用程序做为中转站，它把甲程序的棋子走法传递给乙程序，再把乙程序的走法传递回甲程序。中转站程序利用伪终端功能迷惑住那两个象棋程序，让它们在没有硬件终端在场的情况下以正常方式运行。

伪终端在过去曾经是一个系统一个样，而有的系统甚至还没有这种功能。但它们现在已经以“UNIX98 Pseudo-Terminal”（UNIX伪终端，简称PTY）的身份被纳入“Single UNIX Specification”技术规范中。

5.6 本章总结

在这一章里，我们学习了如何从三个方面对终端进行控制。本章第一部分包括对重定向的检测以及如何在标准文件描述符已经被重定向的情况下直接与终端进行对话等方面的内容。

接下来我们学习了通用终端接口和termios结构，后者提供了对UNIX终端进行处理的细节控制。

最后，我们学习了terminfo数据库及其相关函数的使用方法。通过它们，我们就能以一种与具体终端无关的形式对屏幕输出进行控制和管理。

第6章　curses函数库

我们在上一章学习了如何加强对字符输入过程的精细控制以及如何以一种与具体终端无关的形式提供字符输出等两部分内容。通用终端接口（GTI，即termios）的实际应用和tparm及其相关函数对escape转义序列的操作处理有一个共同的问题，就是必须大量使用低级代码。可在程序设计过程中最好再有一个高级的程序设计接口。如果能够简单地绘制屏幕并通过一系列函数自动处理终端控制方面的问题，我们当然求之不得。

在这一章里，我们将对一个这样的函数库进行学习，它就是curses函数库。curses函数库是一个重要的标准，它是简单的文本行程序和全图形化（一般也更难于编程）X窗口系统程序之间的一个过渡桥梁。Linux提供有svgalib函数库，但这并不是一个标准的UNIX函数库。许多全屏幕应用程序都使用了curses函数库。即使是编写基于字符的全屏幕程序，使用curses函数库的方案也更简明，而程序本身也更独立于具体的终端。在编写这类程序时，使用curses函数库要比直接使用escape转义序列容易得多。curses函数库还可以对键盘进行管理，提供一个简单易用的无阻塞字符输入模式。人们熟悉的小文本编辑器vi就是使用这个函数库编写出来的。我们将对以下几个方面进行讨论：

- curses函数库的使用方法。
- curses函数库涉及的概念。
- 基本的输入输出控制。
- 多窗口。
- 键盘上的数字小键盘。
- 彩色显示功能。

做为本章的学习成果，我们将用C语言重写CD唱盘管理软件，把我们在这几章里学到的知识都用上。

6.1　使用curses函数库进行编译

curses是一个函数库，我们必须从一个适当的系统库文件里把有关的头文件、函数和宏定义等添加到我们的程序里去，这样才能充分享受它给我们带来的便利。但在此之前，我们先来回顾一下它的历史。curses函数库有好几种不同的实现版本。最早的版本出现在BSD版的UNIX操作系统中，以后又出现在System V系列的UNIX操作系统里。在编写本章中的程序示例时，我们使用的是ncurses函数库，这是一个为Linux开发的自由软件函数库，它以System V Release 4.0中的curses函数库为基础。这个版本对其他种类的UNIX操作系统有很高的可移植性。甚至还有能够用在MS-DOS和MS-Windows下的curses函数库。如果读者使用的UNIX操作系统自带的curses函数库缺少对某些功能支持，我们建议你设法搞一份ncurses函数库代替它。

X/Open技术规范定义了两个级别的curses函数库：基本curses函数库和扩展curses函数库。在我们写作本书的时候，最新版的ncurses函数库还不能支持所有的扩展功能，但它已经把最“有用的”功能都包括在内了，比如对多窗口和彩色显示功能的支持等。这一章里有几个curses程序需要用到尚未在ncurses函数库里实现的扩展功能。

扩展curses函数库里有一些是各种附加功能的组合，其中包括一组用来处理多栏字符的函数和一组对色彩进行高级处理的函数。

在对使用了curses函数库的程序进行编译时，我们必须在程序里包括上头文件curses.h，还要在编译命令行上用“-lcurses”选项对curses函数库进行链接。根据读者系统的配置情况，它可能已经是ncurses函数库了。读者可以自行检查自己的curses配置情况，“ls -l /usr/include/*curses.h”命令用来查看头文件，“ls -l /usr/lib/*curses*”命令用来查看库文件。如果读者系统上的curses文件是对ncurses文件的链接，那么，在（用gcc命令）编译本章给出的程序文件时就可以使用下面这样的命令：

```
$ gcc program.c -o program -lcurses
```

如果读者的curses配置情况是不能自动使用ncurses函数库，就需要用下面这样的编译命令来强制使用ncurses函数库：

```
$ gcc -I/usr/include/ncurses program.c -o program -lncurses
```

“-I”选项的作用是指定检索头文件时将要使用的子目录路径。可下载代码形式的制作文件（makefile）会假设用户缺省使用的是ncurses函数库，所以读者必须对此做出修改；说不定在读者系统上还不得不手工完成编译操作。

如果读者对自己系统上的curses函数库配置情况不了解，可以参考ncurses的使用手册。

6.2 基本概念

curses例程工作在屏幕、窗口和子窗口上。“屏幕”就是我们正在写的设备（通常是一个终端屏幕）。它占据了该设备上全部的可用显示面积。当然，如果指的是X窗口系统中的某个终端窗口，那么所谓的“屏幕”就是该终端窗口内部一切可用的字符位置。无论何时，至少会有一个curses窗口，我们称之为stdscr，它与物理屏幕的尺寸完全一样。我们可以额外创建一些尺寸小于屏幕的窗口。窗口可以互相重叠，还可以有许多于窗口，但子窗口必须永远被包含在其父窗口的内部。

curses函数库用两个数据结构来映射终端屏幕，它们是stdscr和curscr。

stdscr相对来说比较重要一些，这个结构会在curses函数产生输出时被刷新。stdscr数据结构指的是“标准屏幕”。它的工作原理和stdio函数库中的标准输出stdout非常相似。在curses程序里，它是缺省的输出窗口。在程序调用refresh之前，输出数据是不会出现在屏幕上的。在需要输出时，curses函数库会把stdscr的内容（屏幕将会是什么样子）和第二个数据结构curscr（屏幕当前的样子）进行比较，然后用这两个结构之间的差异刷新屏幕。

有的curses程序需要了解curses函数库里的stdscr结构，因为有的curses函数需要有一个它这

样的参数。但实际上，stdscr结构与函数库的具体实现密切相关，决不允许直接存取。curses程序不需要用到curscr。

综上所述，在一个curses程序里，输出一个字符的过程是：

- 使用curses函数刷新一个逻辑屏幕。
- 请求curses用refresh刷新物理屏幕。

这个两级跳的办法带来的好处是curses屏幕的刷新非常有效率。这样做对一个控制台屏幕来说并没有多大的重要意义，但如果你是通过一个低速的串行口或调制解调器连接来运行你程序的话，差距就很可观了。

一个curses程序会对逻辑屏幕的输出函数进行许多次调用，比如说在屏幕把光标移动到正确的位置，再从那里开始输出字符或绘制线框。在某个阶段，用户会要求查看全部的输出。这种情况一般出现在refresh被调用期间，此时，curses会计算出让物理屏幕对应上逻辑屏幕的最佳办法。通过curses来刷新屏幕与立刻执行每一个屏幕写操作相比，因为前者使用了适当的终端性能指标并优化了光标的移动动作，所以它输出的字符一般要比后者少很多。curses函数库的名字就来源于它的光标移动优化功能。随着哑终端和低速调制解调器占主导地位的时代渐渐远去，字符输出个数已经不那么重要了，但curses函数库在经历了岁月的洗礼之后依然还是程序员工具箱里的好东西。

逻辑屏幕的布局是一个字符数组，数组下标由行号和列号组成。位置(0, 0)是屏幕的左上角。请看图6-1。

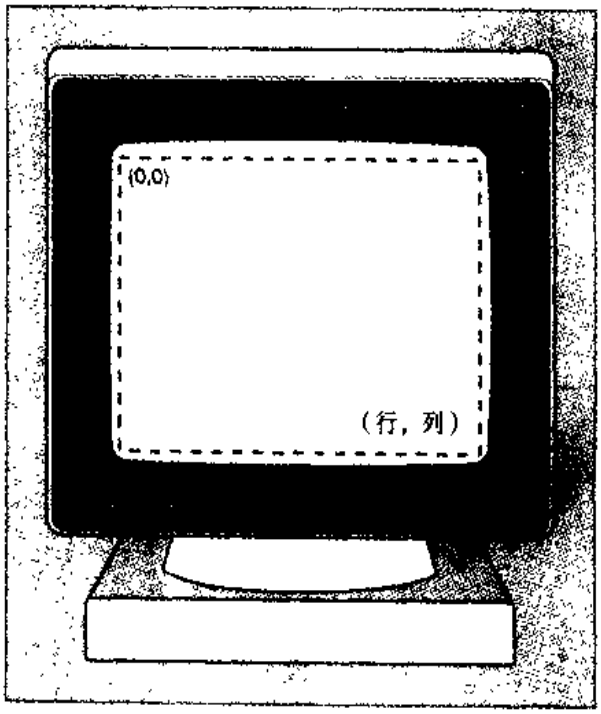


图 6-1

curses函数使用的坐标是y值（行号）在前，x值（列号）在后。每个位置不仅容纳着该屏幕地点处的字符，还保存着它的属性。属性能否被显示出来要看物理终端的性能指标，但一般至少会有黑体和下划线两种属性。

curses函数库在使用中需要建立一些临时性的数据结构，操作中又会对它们造成破坏，所以每个curses程序都必须在开始使用这个库之前对它进行初始化，在结束使用之后恢复curses的原设置。这两项工作是由initscr和endwin函数调用分别完成的。

我们来编写一个非常简单的curses程序screen1.c，让大家看看这两个函数和其他基本函数的使用情况。然后再介绍函数的定义。

动手试试：一个简单的curses程序

1) 我们在程序里加上curses.h头文件，在main函数里加上对curses函数库的初始化函数和重置函数进行调用。

```
#include <unistd.h>
#include <stdlib.h>
#include <curses.h>

int main() {
    initscr();

...

    endwin();
    exit(EXIT_SUCCESS);
}
```

2) 在初始化和重置操作之间，我们把光标移到逻辑屏幕上坐标为(5, 15)的位置，输出“Hello World”，然后刷新物理屏幕。最后，我们调用sleep(2)把程序挂起两秒钟，这样我们就能确保在程序结束之前能够看到它的输出。

```
move(5, 15);
printw("%s", "Hello World");
refresh();

sleep(2);
```

运行程序，我们将在屏幕的左上半部分看见“Hello World”字样，屏幕的其余部分是空白的。如图6-2所示。

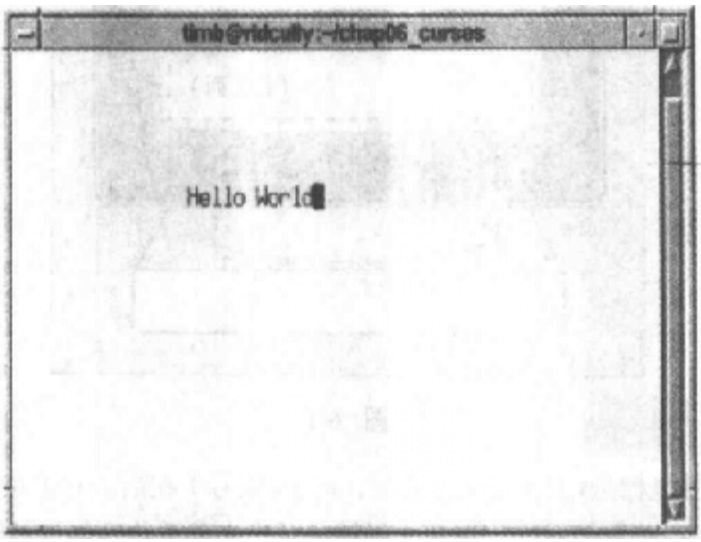


图 6-2

6.3 操作的初始化和结束

正如我们已经看到的，一切curses程序必须以initscr开始，以endwin结束。下面是它们的头文件定义：

```
#include <curses.h>

WINDOW *initscr(void);
int endwin(void);
```

initscr函数在一个程序里只能被调用一次。如果成功，initscr函数会返回一个stdscr结构的指针；如果失败，它就简单地输出一条诊断信息并使程序退出执行。

endwin函数在成功时返回“OK”，失败时返回“ERR”。我们可以先用一个endwin调用离开curses，然后通过clearok(stdscr, 1)和refresh调用重返curses操作。这等于让curses忘记物理屏幕的显示情况，从而强制它执行一次完整的原文重现操作。

WINDOW结构简单说来就是curses用来保存预期屏幕显示内容的结构。这个结构是“不透明的”，也就是说，curses的内部成员也不能直接访问它。

6.4 向屏幕输出数据

下面是几个提供屏幕刷新功能的基本函数：

```
#include <curses.h>

int addch(const chtype char_to_add);
int addchstr(chtype *const string_to_add);
int printw(char *format, ...);
int refresh(void);
int box(WINDOW *win_ptr, chtype vertical_char, chtype horizontal_char);
int insch(chtype char_to_insert);
int insertln(void);
int delch(void);
int deleteln(void);
int beep(void);
int flash(void);
```

curses有自己的字符类型chtype，它可以比标准char类型有更多的二进制位数。比如说，Linux中的ncurses所使用的chtype实际上是一个“unsigned long”整数。

add...系列函数在光标的当前位置添加给定的字符或字符串。printw函数按与printf同样的方法对一个字符串进行格式化，然后把它添加到光标的当前位置。refresh函数的作用是刷新物理屏幕，成功时返回“OK”，出错时返回“ERR”。box函数的作用是让用户围着当前窗口画一个框子。在标准的curses函数库里，用户只能使用“普通”的水平线字符和垂直线字符画框子。

在扩展curses函数里，用户可以利用ACS_VLINE和ACS_HLINE这两个定义画出更好看的框子。用户的终端需要支持画线字符，现如今它已经相当标准了。

insch函数插入一个字符，把现有字符向右移，但在行尾执行这个操作会产生什么样的结果并没有定义，具体情况取决于用户使用的终端。insertln函数的作用是插入一个空行，把现有的行依次向下移一行。那两个delete函数与两个insert函数的作用正好相反。

调用beep函数可以发出声音。有很少一部分终端不能产生任何声音，所以有些curses设置会在调用beep的时候使屏幕闪烁。如果读者是在一个比较繁忙的办公室上班的话，蜂鸣可能会从各种机器设备上产生，这时，你可能会喜欢屏幕闪烁这种方式。正如大家已经猜到的那样，flash函数的作用就是使屏幕产生闪烁，但如果无法产生闪烁效果的话，这个函数会尝试在终端上发出声音来。

6.5 从屏幕读取输入数据

虽然这一功能并不经常使用，但我们确实可以从屏幕上读取字符。下面这些函数可以做到这一点：

```
#include <curses.h>

chtype inch(void);
int instr(char *string);
int innstr(char *string, int number_of_characters);
```

inch函数永远是可用的，但instr和inndtr函数就不总是被支持的了。inch函数返回光标在当前屏幕位置处的那个字符及其属性。需要注意的是inch返回的并不是一个字符，而是一个chtype，而instr和innstr返回的是一个char字符数组。

6.6 清除屏幕

要想清除屏幕上的某个区域，主要有四种办法，它们是：

```
#include <curses.h>

int erase(void);
int clear(void);
int clrtobot(void);
int clrtoeol(void);
```

erase函数在每一个屏幕位置写上空白字符。clear函数的功能类似于erase，也是清屏；但还会通过调用clearok()强制进行一次原文重现操作。clearok强制进行清屏操作，但在下一个refresh调用使会进行原文重现操作。

clear函数的一般做法是用一条终端命令来清除整个屏幕，而不是简单地尝试清除当前屏幕

上每一个有内容的坐标点。所以clear函数的清屏操作既可靠又彻底。clear后面紧跟refresh这样的组合操作提供了一个有效的重新绘制屏幕的手段。

clrtobot清除从当前光标位置到屏幕右下角之间的所有内容；clrtoeol从当前光标位置删除到这一行的结尾。

6.7 移动光标

移动光标只需要一个函数，还有一个辅助函数用来控制刷新屏幕后curses应该把光标放在什么地方：

```
#include <curses.h>

int move(int new_y, int new_x);
int leaveok(WINDOW *window_ptr, bool leave_flag);
```

move函数的作用就是把逻辑光标的位置移动到指定的地点。记住，屏幕坐标是以左上角为(0, 0)的。curses的大多数版本里都用来确定物理屏幕尺寸的两个extern整数LINES和COLUMNS，它们可以用来确定new_y和new_x的最大可取值。调用move本身并不会使物理光标发生移动，它只改变逻辑屏幕上的光标位置，下一个输出就将出现在新位置处。如果我们想让光标位置在调用move函数之后立刻发生变化，就必须在它的后面立刻跟上一个refresh调用。

在一次屏幕刷新过后，curses需要把物理光标放在某个位置上，控制这一位置的标志就是由leaveok函数设置的。在缺省的情况下，这个标志是“false”，在刷新过后，硬件光标将停留在屏幕上逻辑光标所在的地点。如果这个标志被设置为“true”，硬件光标会被随机放置在屏幕上的任意地点。一般说来，人们更喜欢使用缺省选项。

6.8 字符的属性

每个curses字符都可以有特定的属性，属性控制着这个字符在屏幕上的显示方式，当然前提是显示设备硬件能够支持要求的属性。预定义的属性有：A_BLINK、A_BOLD、A_DIM、A_REVERSE、A_STANDOUT和A_UNDERLINE。下面这些函数可以用来设置一个属性或多个属性。

```
#include <curses.h>

int attron(chtype attribute);
int attroff(chtype attribute);
int attrset(chtype attribute);
int standout(void);
int standend(void);
```

attrset函数的作用是对curses属性进行设置，attron和attroff的作用是在不影响其他属性的前提下开启或关闭指定的属性；而standout和standend则提供了一种比较通用的“突出”模式——

在大多数终端上，它通常被映射为反显（高亮度或加黑）。

到这里我们已经把屏幕处理方面的内容介绍得差不多了。现在，我们来分析一个比较复杂的程序示例：moveadd.c。为了让大家更清楚地理解这个程序的功用，我们在程序里增加了几个refresh和sleep调用，好让大家能够看到每一阶段屏幕的显示情况。一般情况下，curses程序会尽可能少地刷新屏幕，因为这并不是一个很有效率的操作。这里给出的代码更多地考虑到演示显示效果的目的。

动手试试：移动光标，插入字符和属性

1) 我们在程序的开始列出必要的头文件，再定义了几个字符数组和一个指向这些数组的指针，然后对curses结构进行了初始化。

```
#include <unistd.h>
#include <stdlib.h>
#include <curses.h>

int main()
{
    const char witch_one[] = " First Witch  ";
    const char witch_two[] = " Second Witch ";
    const char *scan_ptr;

    initscr();
```

2) 现在是最初要显示在屏幕上几组文字。请注意文本属性标志的开关状态。

```
move(5, 15);
attron(A_BOLD);
printw("%s", "Macbeth");
attroff(A_BOLD);
refresh();
sleep(1);

move(8, 15);
attron(A_DIM);
printw("%s", "Thunder and Lightning");
attroff(A_DIM);
refresh();
sleep(1);

move(10, 10);
printw("%s", "When shall we three meet again");
move(11, 23);
printw("%s", "In thunder, lightning, or in rain ?");
move(13, 10);
printw("%s", "When the hurlyburly's done,");
move(14,23);
printw("%s", "When the battle's lost and won.");
refresh();
sleep(1);
```

3) 最后，确定演员的出场顺序，把他们的名字以一次一个字符的方式插入到指定位置。在main函数的末尾我们加上了重置函数endwin。

```
attron(A_DIM);
scan_ptr = witch_one + strlen(witch_one);
while(scan_ptr != witch_one) {
    move(10,10);
    insch(*scan_ptr--);
```

```
}
scan_ptr = witch_two + strlen(witch_two);
while (scan_ptr != witch_two) {
    move(13, 10);
    insch(*scan_ptr--);
}
attroff(A_DIM);
    refresh();
    sleep(1);

    endwin();
    exit(EXIT_SUCCESS);
}
```

当我们运行这个程序的时候，最终的屏幕如图6-3所示。

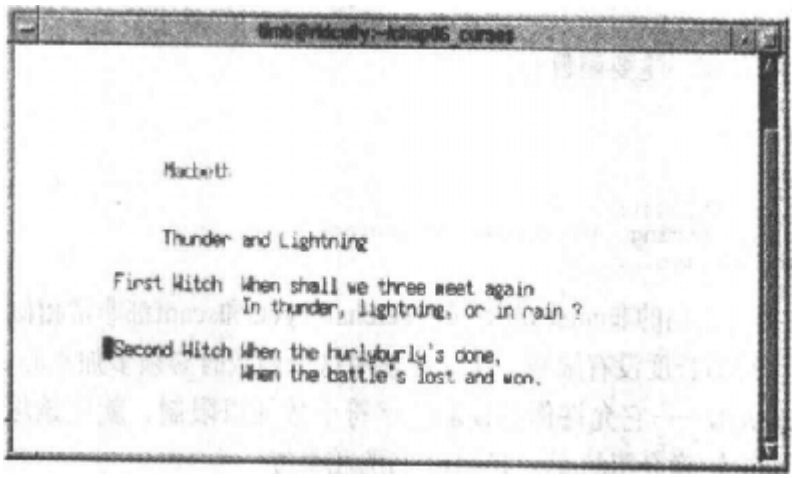


图 6-3

6.9 键盘

curses不仅提供了控制屏幕显示效果的简便手段，还为我们准备了一个控制键盘的简单办法。

6.9.1 键盘的工作模式

读键盘例程是由它的工作模式控制的。对工作模式进行设置的函数如下所示：

```
#include <curses.h>

int echo(void);
int noecho(void);
int cbreak(void);
int nocbreak(void);
int raw(void);
int noraw(void);
```

两个echo函数简单地开启或者关闭输入字符的回显功能。其余四个函数调用控制着在终端上输入的字符将如何被送入curses程序。

要想解释清楚cbreak，我们需要先弄明白缺省的输入模式。当一个curses程序以initscr调用开始运行的时候，输入模式被设置为“预处理模式”。这意味着所有处理都是以文本行为单位的，也就是说，输入数据只有在用户按下回车键之后才回被送入程序。键盘特殊字符都处于被激活

状态，所以敲击适当的按键序列就能在程序里产生一个信号。流（flow）控制也处于被激活状态。程序可以通过调用cbreak把输入模式设置为“字符中止模式”，即字符一经敲入就被立刻送往程序的模式。与预处理模式一样，键盘特殊键也处于被激活状态，但简单的按键（比如Backspace）会被直接送入程序去进行处理。所以如果读者还想让Backspace有原先的功能，就必须自己来编程。

raw调用的作用是关闭特殊键的处理功能，再想通过敲击特殊字符序列的办法来产生信号或流控制就不可能了。调用nocbreak把输入模式设置回预处理模式，但特殊字符处理功能保持不变；调用noraw等于恢复预处理模式和特殊字符处理功能。

6.9.2 键盘输入

读键盘的操作很简单。主要函数有：

```
#include <curses.h>

int getch(void);
int getstr(char *string);
int getnstr(char *string, int number_of_characters);
int scanw(char *format, ...);
```

这些函数的行为与它们的非curses对等函数getchar、gets和scanf都非常相似。需要注意的是：getstr对它返回的字符串长度没有限制，所以在使用这个函数时必须多加小心。如果你的curses函数库版本支持getnstr——它允许你对读取的字符个数加以限制，就应该尽可能用它来代替getstr。它们与我们在第3章介绍的gets和fgets的行为相类似。

下面这个ipmode.c示例程序演示了如何对键盘进行处理。

动手试试：键盘模式和输入

1) 首先，对程序的初始化curses调用进行设置。

```
#include <unistd.h>
#include <stdlib.h>
#include <curses.h>
#include <string.h>

#define PW_LEN 25
#define NAME_LEN 256

int main() {
    char name[NAME_LEN];
    char password[PW_LEN];
    char *real_password = "xyzzy";
    int i = 0;

    initscr();

    move(5, 10);
    printw("%s", "Please login:");

    move(7, 10);
    printw("%s", "User name: ");
    getstr(name);

    move(9, 10);
    printw("%s", "Password: ");
    refresh();
```

2) 用户输入口令字的时候，我们不能让口令字回显在屏幕上。然后检查口令字是否等于"xyzzy"

```
cbreak();
noecho();

memset(password, '\0', sizeof(password));
while (i < PW_LEN) {
    password[i] = getch();
    move(9, 20 + i);
    addch('*');
    refresh();
    if (password[i] == '\n') break;
    if (strcmp(password, real_password) == 0) break;
    i++;
}
```

3) 最后，重新激活键盘的回显功能并给出口令字检验是否成功的执行结果。

```
    echo();
    nocbreak();

    move(11, 10);
    if (strcmp(password, real_password) == 0) printw("%s", "Correct");
    else printw("%s", "Wrong");
    refresh();

    endwin();
    exit(EXIT_SUCCESS);
}
```

运行这个程序，看看它的运行情况是怎样的。

操作注释：

在用noecho函数关闭了键盘输入的回显功能，用cbreak函数把输入情况设置为字符中止模式之后，我们开辟一小块内存做好接收口令字的准备工作。构成口令字的每一个字符一经敲入就立刻得到处理。随后，我们在屏幕的下一个位置显示一个"*"号，注意每次都要对屏幕进行刷新。用户按下回车键后，我们使用Strcmp把刚输入的口令字和保存在程序里的口令字进行比较。

> 如果你使用的curses函数库版本非常老，就可能需要对程序做一些小修改才能保证屏幕能够被正确地刷新。即你必须在getstr调用前加上一个refresh调用。在现如今的curses函数库里，getstr被定义为调用getch完成操作，而后者会自动刷新屏幕。

6.10 窗口

到目前为止，我们一直是把终端用做一个全屏输出介质。对短小而又简单的程序来说，这样做一般已经足够了，但curses函数库的"功力"远大于此。我们可以在物理屏幕上同时显示多个不同尺寸的窗口。在这一小节里介绍的函数大部分只有X/Open技术规范里规定的扩展curses函数库才能支持。但既然curses函数库能够支持它们，在大多数平台上使用它们一般也就不会有什么问题。我们将继续学习多窗口的使用方法。到目前为止我们已经学过不少的函数了，我们将看到这些函数命令的通用化形式是如何用来处理多窗口情况的。

6.10.1 WINDOW结构

我们已经介绍过标准屏幕stdscr了，但我们一直没有用上它，因为我们前面遇见的函数都假设自己是工作在stdscr上，因此不需要把它做为一个参数进行传递。

标准屏幕stdscr只是WINDOW结构的一个特例，就象标准输出stdout是文件流中的一个特例一样。WINDOW结构通常定义在curses.h头文件里，对它的存取操作必须按规定的指令进行，程序永远不允许直接访问它，因为这个结构依赖于curses函数库的具体实现情况，它在不同版本的curses函数库里会有所变化。

我们可以用newwin和delwin调用创建和关闭窗口，下面是这两个调用的定义：

```
#include <curses.h>

WINDOW *newwin(int num_of_lines, int num_of_cols, int start_y, int start_x);
int delwin(WINDOW *window_to_delete);
```

newwin函数的作用是创建一个新窗口，窗口从屏幕位置(start_y, start_x)开始，尺寸由分别代表行数和列数的num_of_lines和num_of_cols参数指定。它返回一个指向新窗口的指针；如果窗口创建操作失败，将返回“null”。如果想让新窗口的右下角正好落在屏幕的右下角位置上，可以把它的行数或列数设置为零。窗口不论新旧大小，决不允许超越当前屏幕的范围。如果新窗口的某个部分会落在屏幕区域以外的地方，newwin就会失败。newwin创建的新窗口完全独立于任何现有的窗口。在缺省的情况下，它将被放置在一切现有窗口的最上面，遮盖（但不会改变）老窗口的内容。

delwin函数的作用是删除一个通过newwin函数创建的窗口。因为调用newwin的时候可能已经分配过内存，所以只要某个窗口不再需要被使用，最好立刻删掉它。但千万不要去尝试删除curses自己的窗口stdscr和curscr!

创建出新窗口之后，怎样才能对它进行写操作呢？答案是这样的：几乎所有我们前面见过的函数都有能够对指定窗口进行操作的通用化版本，并且，为了方便人们的使用，它们还都具备光标移动功能。

6.10.2 通用化函数

在前面的内容里，当需要把字符添加到屏幕上去的时候，我们给大家介绍了addch和printw这两个函数。这两个函数，再加上其他一些函数，都可以再添上几个前缀：前缀“w”表示对窗口进行操作；“mv”表示光标移动；而“mvw”表示对窗口进行整体移动。在curses函数库的大多数具体实现里，如果你去查看它的头文件，就会发现我们曾经使用过的许多函数都只是简单的宏定义（#define语句），宏定义的内容是一些更通用化的函数。

如果给函数添上一个“w”前缀，就必须在它参数表的最前面多加上一个WINDOW指针。如果给函数添上一个“mv”前缀，就必须多加两个参数，一个是屏幕的纵坐标y值，一个是屏幕的横坐标x值。两个坐标值设定了执行这一操作的屏幕位置。y和x是相对于窗口的坐标值而不是相对于屏幕的坐标值，(0, 0)代表窗口的左上角。

如果给函数添上一个“mvw”前缀，就必须多传递三个参数，它们是一个WINDOW指针、

一个纵坐标y值和一个横坐标x值。WINDOW指针永远出现在屏幕坐标值的前面，可有时候从函数前缀上看好象应该把y和x坐标值放在最前面，遇到这种情况千万不要犯糊涂，否则很容易出问题。

下面是addch和printw系列全体函数的框架定义：

```
#include <curses.h>

int addch(const chtype char);
int waddch(WINDOW *window_pointer, const chtype char)
int mvaddch(int y, int x, const chtype char);
int mvwaddch(WINDOW *window_pointer, int y, int x, const chtype char);
int printw(char *format, ...);
int wprintw(WINDOW *window_pointer, char *format, ...);
int mvprintw(int y, int x, char *format, ...);
int mvwprintw(WINDOW *window_pointer, int y, int x, char *format, ...);
```

其他一些函数，比如inch，也有加“mv”（移动）和“mvw”（窗口移动）前缀的通用化变体。

6.10.3 移动和刷新窗口

下面这些命令让我们能够移动和重新绘制窗口：

```
#include <curses.h>

int mvwin(WINDOW *window_to_move, int new_y, int new_x);
int wrefresh(WINDOW *window_ptr);
int wclear(WINDOW *window_ptr);
int werase(WINDOW *window_ptr);
int touchwin(WINDOW *window_ptr);
int scrollok(WINDOW *window_ptr, bool scroll_flag);
int scroll(WINDOW *window_ptr);
```

mvwin函数的作用是在屏幕上移动一个窗口。因为不允许窗口的任何部分超出屏幕区域，所以如果在移动窗口的时候它的某个部分会落到屏幕区域以外的地方，这个mvwin操作就会失败。

wrefresh、wclear和werase函数分别是refresh、clear和erase函数的通用化版本，加上那个WINDOW指针参数之后，它们就可以对指定的窗口进行操作，不再局限于stdscr了。

touchwin函数的情况比较特殊。它的作用是通知curses函数库它的参数指向的窗口中的内容已经发生了改变。这就意味着curses必须在下一次的wrefresh调用里重新绘制那个窗口，哪怕用户实际上并没有修改过那个窗口里的内容，curses也还是会照章办理。在屏幕上重叠着多个窗口时，我们可以通过这个函数来安排显示其中的一个。

scrollok和scroll两个函数控制着窗口的卷屏情况。如果传递到scrollok函数里去的是一个布尔true值（通常是非零值），就允许窗口卷屏；而在缺省的情况下，窗口是不能卷屏的。scroll函数的作用比较简单，就是把窗口的内容上卷一行。有的curses版本里还有一个wsctl函数，它多出一个指定卷行行数的参数，这个参数还可以是负数值。我们在本章后面的内容里再对卷屏问题进行进一步讨论。

现在，我们已经能够对不止一个窗口进行管理和操作了。我们来把这些新函数用在下面这个multiw1.c程序里。为简洁起见，我们的程序省略了错误检查。

动手试试：多窗口

1) 和往常一样，我们先安排好各种各样的定义：

```
#include <unistd.h>
#include <stdlib.h>
#include <curses.h>

int main()
{
    WINDOW *new_window_ptr;
    WINDOW *popup_window_ptr;
    int x_loop;
    int y_loop;
    char a_letter = 'a';

    initscr();
```

2) 然后，我们用字符填满基本窗口，填满逻辑窗口后对物理屏幕进行刷新。

```
move(5, 5);
    printw("%s", "Testing multiple windows");
    refresh();

    for (y_loop = 0; y_loop < LINES - 1; y_loop++) {
        for (x_loop = 0; x_loop < COLS - 1; x_loop++) {
            mvwaddch(stdscr, y_loop, x_loop, a_letter);
            a_letter++;
            if (a_letter > 'z') a_letter = 'a';
        }
    }

    /* Update the screen */
    refresh();
    sleep(2);
```

3) 现在，我们来创建一个10x20的窗口，并在把它绘制到屏幕上之前也给它填上一些文字。

```
new_window_ptr = newwin(10, 20, 5, 5);
mvwprintw(new_window_ptr, 2, 2, "%s", "Hello World");
mvwprintw(new_window_ptr, 5, 2, "%s",
          "Notice how very long lines wrap inside the window");
wrefresh(new_window_ptr);
sleep(2);
```

4) 接下来，我们对背景窗口里的内容做一些改动。继续执行，当我们刷新屏幕的时候，new_window_ptr指向的窗口将被遮盖住。

```
a_letter = '0';
for (y_loop = 0; y_loop < LINES -1; y_loop++) {
  for (x_loop = 0; x_loop < COLS - 1; x_loop++) {
      mvwaddch(stdscr, y_loop, x_loop, a_letter);
      a_letter++;
      if (a_letter > '9')
          a_letter = '0';
    }
}

refresh();
sleep(2);
```

5) 此时，如果我们对新窗口做一次刷新调用，什么也不会发生，因为我们还没有对新窗口做过改动。

```
wrefresh(new_window_ptr);
sleep(2);
```

6) 但如果我们先对新窗口做次touchwin调用，让curses认为新窗口里的内容已经发生了变化，下一个wrefresh调用就会再次把新窗口调到屏幕的最前面来。

```
touchwin(new_window_ptr);
wrefresh(new_window_ptr);
sleep(2);
```

7) 接着，我们再添上一个加了框的重叠窗口。

```
popup_window_ptr = newwin(10, 20, 8, 8);
box(popup_window_ptr, '|', '-');
mvwprintw(popup_window_ptr, 5, 2, "%s", "Pop Up Window!");
wrefresh(popup_window_ptr);
sleep(2);
```

8) 执行下面这些语句之后，我们就能在清屏和删除它们之前看到那个新的“弹出”窗口了。

```
    touchwin(new_window_ptr);
    wrefresh(new_window_ptr);
    sleep(2);
    wclear(new_window_ptr);
    wrefresh(new_window_ptr);
    sleep(2);
    delwin(new_window_ptr);
    touchwin(popup_window_ptr);
    wrefresh(popup_window_ptr);
    sleep(2);
    delwin(popup_window_ptr);
    touchwin(stdscr);
    refresh();
    sleep(2);
    endwin();
    exit(EXIT_SUCCESS);
}
```

我们没有办法让读者在书里看到这一切的发生过程，只能把各阶段的屏幕显示情况以快照的形式提供给大家。图6-4是绘制出第一个弹出窗口时的情景。

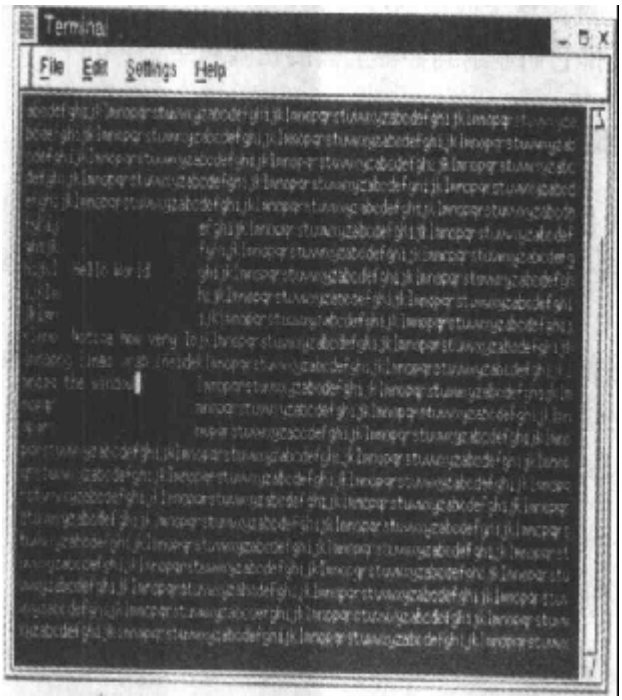


图 6-4

改变背景窗口后，又绘制出一个弹出窗口，请看图6-5的屏幕快照。

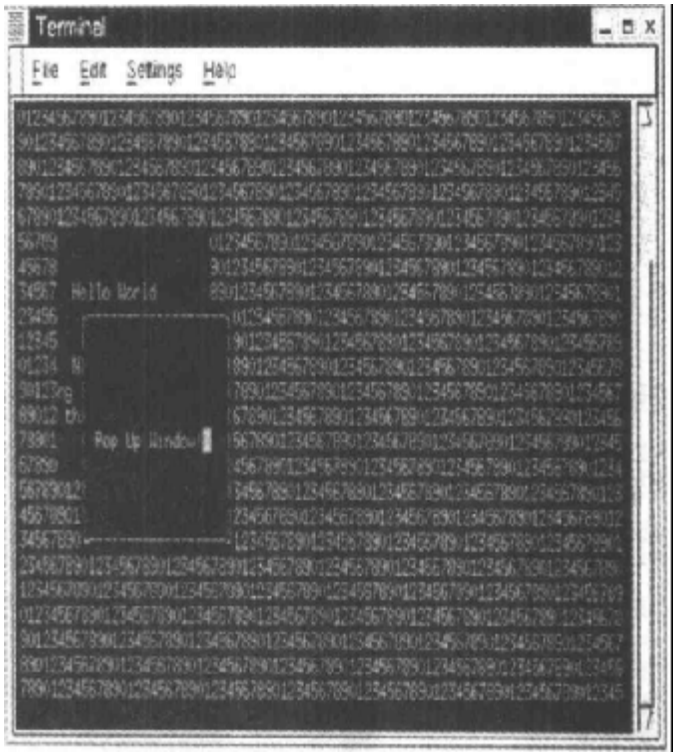


图 6-5

从示例程序的代码我们可以看出，对窗口的刷新操作必须做细致的安排才能让它们以正确的顺序显示到屏幕上。curses在刷新窗口的时候并不考虑窗口在屏幕上出现的先后顺序，所谓窗口之间的上下继承关系也无从谈起。为了确保curses能够以正确的顺序绘制窗口，就必须以正确的顺序对它们进行刷新。解决这个问题的办法之一是把全体窗口的指针都保存到一个数组或列表里，通过这个数组来安排它们显示到屏幕上去的正确顺序。

6.10.4 优化窗口的刷新操作

从上面的例子可以看出，对多个窗口进行刷新需要一定的技巧，但还不至于无章可循。尽管如此，当我们准备刷新一个慢速链路（比如一个调制解调器）上的终端时，就可能出现比较严重的问题。

在这种情况下，最主要的事情是尽量减少往屏幕上绘制的字符个数，因为慢速链路上的屏幕绘制工作可能会慢得让人难以忍受。curses为此准备了一个非常手段，这需要使用下面两个函数：wnoutrefresh和doupdate。

```
#include <curses.h>

int wnoutrefresh(WINDOW *window_ptr);
int doupdate(void);
```

wnoutrefresh函数的作用是决定需要把哪个字符发送到屏幕去，但并不真正执行字符发送操作，真正把字符发送给终端的工作由doupdate函数完成的。如果在调用wnoutrefresh函数之后立

刻跟上一个doupdate调用，就相当于调用wrefresh函数的效果。如果你想重新绘制好几个窗口，可以先为每一个窗口调用wnoutrefresh函数（当然要按正确的顺序来操作），在最后那个wnoutrefresh函数完成之后再统一调用一次doupdate函数。curses会依次完成各窗口在屏幕刷新方面的计算，仅把最终的结果刷新到屏幕上。这种做法可以最大限度地减少需要curses发送的字符个数。

6.11 子窗口

介绍完多窗口之后，我们来看看一种特殊类型的多窗口：子窗口。子窗口的创建和关闭工作是用下面这几个函数完成的：

```
#include <curses.h>

WINDOW *subwin(WINDOW *parent, int num_of_lines, int num_of_cols,
               int start_y, int start_x);
int delwin(WINDOW *window_to_delete);
```

subwin函数的参数几乎与newwin完全一样，子窗口的删除过程也与其他窗口通过delwin调用被删除的情况相同。我们可以使用带“mvw”前缀的函数来写子窗口，就象对待一个新窗口那样。事实上，在大多数时间里，子窗口的行为与一个新窗口非常相似，两者之间只有一个重大的差异：

子窗口没有独立的屏幕字符存储区，不保存自己屏幕字符的集合；它们和父窗口共享着同一块字符存储区，这个区域及其大小是创建子窗口时由父窗口设定的。这就意味着对子窗口内容所做的修改会映射到它的父窗口里去，子窗口被删除时屏幕显示不会发生变化。

这样说来，子窗口好象没什么用处。为什么不直接在父窗口里做内容修改呢？子窗口最主要的用途是有选择地卷动其他窗口里的部分内容，这个办法很简明。在编写curses程序的时候，经常会出现需要卷动屏幕某个小区域的情况。把这个小区域定义为一个子窗口，卷动这个子窗口，就能到达我们的目的。

> 使用子窗口时有一个限制规定：在应用程序刷新屏幕之前必须先对其父窗口调用touchwin函数。

动手试试：子窗口

1）首先是初始化代码部分。我们先用一些文字初始化基本窗口的显示情况。

```
#include <unistd.h>
#include <stdlib.h>
#include <curses.h>

#define NUM_NAMES 14

int main()
{
    WINDOW *sub_window_ptr;
    int x_loop;
    int y_loop;
    int counter;
    char a_letter = 'A';
```

```
    char *names[NUM_NAMES] = {"David Hudson,", "Andrew Crolla,", "James Jones,",
                              "Ciara Loughran,", "Peter Bradley,", "Nancy Innocenzi,",
                              "Charles Cooper,", "Rucha Nanavati,", "Bob Vyas,",
                              "Abdul Hussain,", "Anne Pawson,", "Alex Hopper,",
                              "Russell Thomas,", "Nazir Makandra,"};

    initscr();

for (y_loop = 0; y_loop < LINES - 1; y_loop++) {
      for (x_loop = 0; x_loop < COLS - 1; x_loop++) {
            mvwaddch(stdscr, y_loop, x_loop, a_letter);
            a_letter++;
            if (a_letter > 'Z') a_letter = 'A';
        }
    }
```

2) 现在来创建一个新的卷屏子窗口。根据前面的介绍，在刷新屏幕之前必须对父窗口调用touchwin函数。

```
sub_window_ptr = subwin(stdscr, 10, 20, 10, 10);
scrollok(sub_window_ptr, 1);
touchwin(stdscr);
refresh();
sleep(1);
```

3) 接下来，我们删掉子窗口里的内容，重新输出一些文字，刷新它。文字的卷屏动作是由一个循环来完成的。

```
werase(sub_window_ptr);
mvwprintw(sub_window_ptr, 2, 0, "%s",
          "This window will now scroll as names are added ");
wrefresh(sub_window_ptr);
sleep(1);

for (counter = 0; counter < NUM_NAMES; counter++) {
    wprintw(sub_window_ptr, "%s ", names[counter]);
    wrefresh(sub_window_ptr);
    sleep(1);
}
```

4) 循环结束后，我们删掉子窗口。然后再次刷新基本屏幕。

```
delwin(sub_window_ptr);
touchwin(stdscr);
refresh();
sleep(1);
endwin();
exit(EXIT_SUCCESS);
}
```

图6-6是我们在程序执行后看到的屏幕显示情况。

图 6-6

操作注释：

在安排sub_window_ptr指向subwin调用的结果后，我们把子窗口设置为“可卷屏”状态。在删掉子窗口并重新刷新了基本窗口（stdscr）之后，屏幕上的文字依然着保持原来的样子。这是因为子窗口实际刷新的是stdscr中的字符数据。

6.12 键盘上的数字小键盘

我们已经见过一些curses提供的键盘处理功能了。不管是哪一种键盘，上面多少会有几个光标移动键和功能键。许多键盘还带有一个数字小键盘和“Insert”、“Home”等其他按键。

对大多数终端来说，解码这些键是一个很困难的问题，因为它们往往会送出以escape字符打头的一连串字符。应用程序不仅要设法分辨出“单独按下Escape按键”和“按下功能键发送出来的一连串字符”之间的差别，还要对付因终端型号的不同而产生的“同一逻辑按键使用不同转义序列”的情况，这两个问题交织在一起，错综复杂。

幸好curses在功能键管理方面已经为我们大家准备了一个精巧的工具。对各个终端来说，它的每一个功能键发送出来的转义序列通常都被保存在一个terminfo结构里，而头文件curses.h通过一组以“KEY_”为前缀的定义把功能键都预先安排好了。

curses在启动时会关闭转义序列与逻辑按键之间的转换功能，这需要用keypad函数重新开启。如果这个函数调用成功，它将返回“OK”，否则就返回“ERR”。

```
# include <curses.h>
int keypad( WINDOW *window_ptr, bool keypad_on );
```

把keypad_on设置为true再调用keypad函数将激活Keypad模式，curses由此开始接管按键转义序列的处理工作。这样，读键盘操作不仅能返回被按下的键，还能返回与逻辑按键一一对应的“KEY_”定义。

使用Keypad模式有三条小小的限制。

第一个问题是识别escape转义序列需要一定时间，而许多网络协议要么把字符打成数据包（这会导致escape转义序列的识别不准确），要么会从某个地方开始分断它们（这将导致功能键的转义序列被识别为Escape字符和其他彼此没有联系的字符）。在WAN网络或其他繁忙的链路上这一情况将更为加剧。惟一的解决之道是设法对终端进行编程，让它针对用户使用的每一个功能键发送出单个的、独一无二的字符来——虽然这样做会限制住控制字符的数量。

第二个问题，为了让curses能够区分“单独按下Escape按键”和一个以Escape字符打头的键盘转义序列，它就必须等待很短的一小段时间。在Keypad模式被激活之后，处理Escape按键所造成的非常微小的延时也能被注意到。

第三条限制是curses不能处理二义性的escape转义序列。如果键盘上两个不同的按键会产生同一个转义序列，就会让curses不知所措，不知道该返回哪个逻辑按键。curses对这种情况的处理措施也很简单，它会放弃对这个转义序列的处理。

> 我们的看法是：从产品设计的角度看，既让某些按键发送escape转义序列又在键盘上布置一个Escape键（用这个键实现“取消操作”的软件不胜枚举）的做法是一个最不理智的决策，但我们不得不接受这个现实，而且还必须尽力搞好它。

下面这小程序keypad.c演示了Keypad模式的使用方法。在运行这个程序的时候，按下Escape键并注意观察那个微小的延时，程序将在这段延时里判断这个Escape是一个转义序列的开始还是一次单独的击键动作。

动手试试：使用Keypad模式

1) 程序和curses函数库的初始化工作完成之后，我们把Keypad模式设置为“TRUE”。

```
#include <unistd.h>
#include <stdlib.h>
#include <curses.h>

#define LOCAL_ESCAPE_KEY    27

int main()
{
    int key;

    initscr();
    crmode();
    keypad(stdscr, TRUE);
```

2) 接下来，我们必须关闭回显功能以防止光标在我们按下光标键时发生移动。清屏并显示一些文字。程序等待击键动作，如果既不是“q”也没有产生错误，就把按键对应的字符显示到屏幕上。如果击键动作匹配上终端的某个转义序列，就把这个转义序列显示到屏幕上。

```
    noecho();
    clear();
    mvprintw(5, 5, "Key pad demonstration. Press 'q' to quit");
    move(7, 5);
    refresh();
    key = getch();

    while(key != ERR && key != 'q') {
        move(7, 5);
        clrtoeol();

        if ((key >= 'A' && key <= 'Z') ||
            (key >= 'a' && key <= 'z')) {
            printw("Key was %c", (char)key);
        }
        else {
            switch(key) {
            case LOCAL_ESCAPE_KEY: printw("%s", "Escape key"); break;
            case KEY_END: printw("%s", "END key"); break;
            case KEY_BEG: printw("%s", "BEGINNING key"); break;
            case KEY_RIGHT: printw("%s", "RIGHT key"); break;
            case KEY_LEFT: printw("%s", "LEFT key"); break;
            case KEY_UP: printw("%s", "UP key"); break;
            case KEY_DOWN: printw("%s", "DOWN key"); break;
            default: printw("Unmatched - %d", key); break;
            } /* switch */
        } /* else */

        refresh();
        key = getch();
    } /* while */

    endwin();
    exit(EXIT_SUCCESS);
}
```

6.13 彩色显示功能

从前的“哑”终端很少有具备彩色显示功能的，所以早期的curses函数库也没有对这方面的支持。现如今，彩色显示功能已经是相当普遍的标准，ncurses和大多数近期的curses实现也都开

始支持它了。

屏幕的每一个字符位置都可以从多种颜色里挑一种写上去，它的背景也可以从多种颜色里挑选。比如说，我们可以在一个红色背景上写出绿色的文本。

curses中的彩色显示功能有它自己的特点，即字符颜色的定义与它的背景并不是毫无关系的。我们必须把字符的前景颜色和背景颜色定义为一组，我们称之为一个“颜色组合”。

先确定你的当前终端确实支持彩色显示功能，再对curses的彩色例程进行初始化，然后才能开始使用curses中的彩色显示功能。上面两项工作是由has_colors和start_color这两个函数完成的，下面是它们的定义：

```
#include <curses.h>

bool has_colors(void);
int start_color(void);
```

如果终端支持彩色显示功能，has_colors例程将返回“true”。接下来需要调用start_color函数，如果彩色显示功能的初始化操作成功了，它将返回“OK”。调用start_color对彩色显示功能的初始化操作成功之后，变量COLOR_PAIRS将被设置为该终端所能支持的颜色组合的最大值。最多64个颜色组合是比较常见的。变量COLORS定义了可用色彩的种类，只有八种色彩的情况比较多见。

在把颜色用做属性之前，我们必须对准备使用的颜色组合进行初始化。这项工作可以用init_pair函数完成。对颜色组合的存取要通过COLOR_PAIR函数来进行。

```
#include <curses.h>

int init_pair(short pair_number, short foreground, short background);
int COLOR_PAIR(int pair_number);
int pair_content(short pair_number, short *foreground, short *background);
```

curses.h通常会定义一些基本的颜色，其名称都以“COLOR_”打头。另外还有一个函数pair_content，它的作用是对定义好的颜色组合信息进行检索。

用下面的语句可以把绿背景红前景定义为第一号颜色组合：

```
init_pair(1, COLOR_RED, COLOR_GREEN );
```

这个颜色组合就可以被用做属性了，注意下面语句中COLOR_PAIR的用法：

```
wattron(window_ptr, COLOR_PAIR(1) );
```

这个语句的作用是把屏幕的后续内容设置为绿色背景上的红色内容。

因为一个COLOR_PAIR就是一个属性，所以我们可以把它和其他属性结合在一起使用。在PC个人电脑上，我们通常可以通过组合COLOR_PAIR属性和附加属性A_BOLD在屏幕上得到加浓的颜色，两个属性要按位OR在一起，如下所示：

```
wattron(window_ptr, COLOR_PAIR(1) | A_BOLD );
```

我们来看看这些函数在下面的color.c程序里的使用情况。

动手试试：彩色

1) 首先，检查这个程序的显示终端是否支持彩色显示功能。如果它支持，就开始彩色显示。

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

int main()
{
    int i;

    initscr();

    if (!has_colors()) {
        endwin();
        fprintf(stderr, "Error - no color support on this terminal\n");
        exit(1);
    }

    if (start_color() != OK) {
        endwin();
        fprintf(stderr, "Error - could not initialize colors\n");
        exit(2);
    }
```

2) 现在，我们可以把允许使用的颜色种类和颜色组合种类打印出来。我们将创建几个颜色组合并把它们同时显示在屏幕上。

```
    clear();
    mvprintw(5, 5, "There are %d COLORS, and %d COLOR_PAIRS available",
            COLORS, COLOR_PAIRS);
    refresh();

    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_RED, COLOR_GREEN);
    init_pair(3, COLOR_GREEN, COLOR_RED);
    init_pair(4, COLOR_YELLOW, COLOR_BLUE);
    init_pair(5, COLOR_BLACK, COLOR_WHITE);
    init_pair(6, COLOR_MAGENTA, COLOR_BLUE);
    init_pair(7, COLOR_CYAN, COLOR_WHITE);

    for (i = 1; i <= 7; i++) {
        attroff(A_BOLD);
        attrset(COLOR_PAIR(i));
        mvprintw(5 + i, 5, "Color pair %d", i);
        attrset(COLOR_PAIR(i) | A_BOLD);
        mvprintw(5 + i, 25, "Bold color pair %d", i);
        refresh();
        sleep(1);
    }

    endwin();
    exit(EXIT_SUCCESS);
}
```

这个程序示例将给出如图6-7所示的输出效果：

彩色显示功能的细化

有的终端对屏幕上能够同时显示的颜色种类有一定的限制，但允许人们对可用颜色进行细化。curses通过init_color函数提供了对这一功能的支持。

```
#include <curses.h>

int init_color(short color_number, short red, short green, short blue);
```

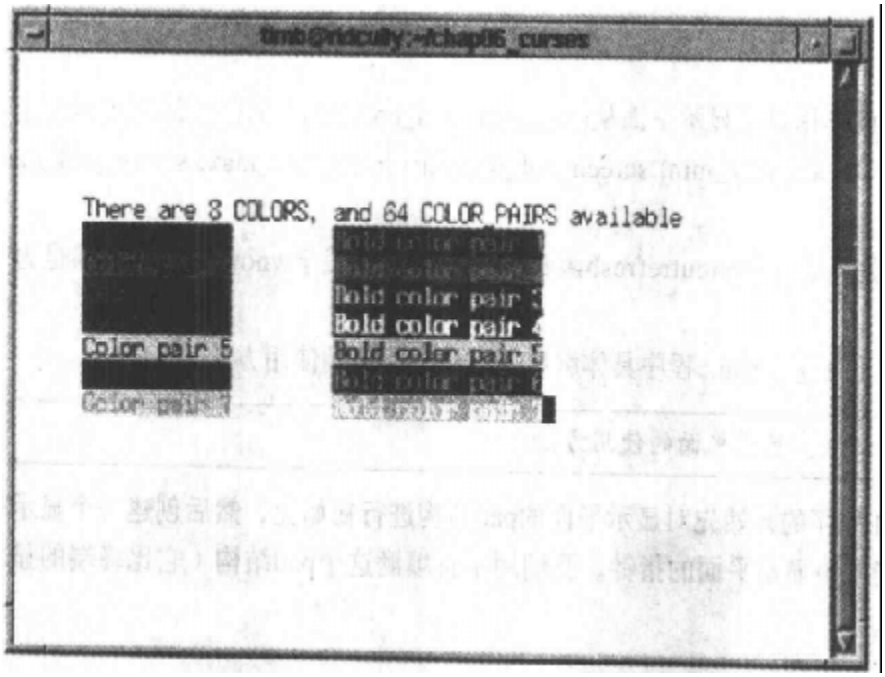


图 6-7

这个函数把一个可用色彩（0到COLORS范围之间）的深度细调为一个新的值，新值的范围是从0到1000。这有点象为PC个人电脑的屏幕定义其VGA色板的颜色值。

6.14 逻辑屏幕和显示平面

在编写比较高级的curses程序时，我们可以先创建一个逻辑屏幕，然后再把它的全部或部分内容输出到物理屏幕上，这个办法简便易行，效果也不错。如果还能有一个尺寸大于物理屏幕的逻辑屏幕，再根据需要一次只显示逻辑屏幕的某个部分，其效果往往会更好。

但就我们目前已经学过的curses函数而言，要想实现这种构思难度极大，因为任何窗口的尺寸都不能超越物理屏幕。但curses允许逻辑屏幕大于正常窗口，并且专门为尺寸大于正常窗口的逻辑屏幕上的信息处理准备了一种特殊的数据结构——显示平面（pad）。

创建显示平面和创建正常窗口在原理上是相同的：

```
#include <curses.h>

WINDOW *newpad(int number_of_lines, int number_of_columns);
```

请注意，这个函数的返回值是一个指向WINDOW结构的指针，这一点也类似于newwin函数。显示平面也要用delwin来删除，就象正常窗口一样。

显示平面有自己专用的刷新例程。显示平面不受屏幕坐标位置的限制，所以在刷新时必须指定希望把显示平面的哪个区间输出到屏幕上，还必须设定它在屏幕上占据的坐标位置。请看对显示平面进行刷新的prefresh函数的定义：

```
#include <curses.h>

int prefresh(WINDOW *pad_ptr, int pad_row, int pad_column,
```

```
            int screen_row_min, int screen_col_min,
            int screen_row_max, int screen_col_max);
```

这个函数的作用是把显示平面从(pad_row, pad_column)开始的区间写到屏幕上，屏幕显示区被定义为坐标(screen_row_min, screen_col_min)和(screen_row_max, screen_col_max)之间的那一块区域。

curses还提供了一个pnoutrefresh函数，它的作用类似于wnoutrefresh，都是为了更有效率地刷新屏幕。

我们通过下面这个pad.c程序具体解释一下这些函数的使用方法。

动手试试：显示平面的使用方法

1) 我们在程序的开始先对显示平面的pad结构进行初始化，然后创建一个显示平面，创建函数返回一个指向该显示平面的指针。我们用字符填满这个pad结构（它比终端的显示区长宽各多出50个字符）。

```
#include <unistd.h>
#include <stdlib.h>
#include <curses.h>

int main()
{
    WINDOW *pad_ptr;
    int x, y;
    int pad_lines;
    int pad_cols;
    char disp_char;

    initscr();
    pad_lines = LINES + 50;
    pad_cols = COLS + 50;
    pad_ptr = newpad(pad_lines, pad_cols);
    disp_char = 'a';

    for (x = 0; x < pad_lines; x++) {
        for (y = 0; y < pad_cols; y++) {
            mvwaddch(pad_ptr, x, y, disp_char);
            if (disp_char == 'z') disp_char = 'a';
            else disp_char++;
        }
    }
```

2) 现在，我们把显示平面的不同区间绘制到屏幕的不同区域。然后退出。

```
    prefresh(pad_ptr, 5, 7, 2, 2, 9, 9);
    sleep(1);
    prefresh(pad_ptr, LINES + 5, COLS + 7, 5, 5, 21, 19);
    sleep(1);
    delwin(pad_ptr);
    endwin();
    exit(EXIT_SUCCESS);
}
```

运行这个程序，我们将看到如图6-8所示的输出情况：

6.15 CD唱盘管理软件

curses提供的工具大家已经学得差不多了，接下来，我们趁热打铁，动手开发我们的示范软

图 6-8

件。示范软件下面这个C语言版本里使用了curses函数库。它的优点不少，比如屏幕提示信息更整齐规范，曲目清单还用上了卷屏窗口等。

整个应用程序有8页之多，所以我们先把它分成几个小节，再对各个小节里的函数做进一步的分析说明。完整的源代码可以从Wrox出版社的Web站点上获得。这个示范软件与这本书里的其他程序一样，都采用了“Gnu Public License”（Gnu公共许可证，参见附录B）。

CD数据库管理软件的这个版本是在第5章和第6章内容的基础上编写的。它脱胎于第2章里的shell脚本程序。我们沿用了原来的基本思路，没有根据C语言特点进行调整，所以还可以从这个版本里看出很多shell脚本程序的特点。

这个程序还有几个不足之处，我们将在今后的学习过程中加以解决。比如，它还不能处理唱盘名称带逗号的情况，在屏幕上显示CD唱盘中的曲目时还有数量方面的限制等。

代码本身根据功能的不同明显地可以被分为几个小节，我们就把这些功能做为“动手试试”的标题。代码中使用的编排体例与这本书的其他部分不太一样，灰影部分的内容是对应用程序里其他函数的调用。

动手试试：新编CD唱盘管理软件

1) 首先，列出所有必需的头文件并定义几个全局性常数。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <curses.h>

#define MAX_STRING 80          /* Longest allowed response       */
#define MAX_ENTRY 1024         /* Longest allowed database entry */

#define MESSAGE_LINE 6           /* Misc. messages on this line    */
#define ERROR_LINE   22          /* Line to use for errors         */
#define Q_LINE       20          /* Line for questions             */
#define PROMPT_LINE  18          /* Line for prompting on          */
```

2) 接下来，定义全局变量。变量current_cd用来保存我们正在处理的当前CD唱盘的标题名称。我们把它的第一个字符初始化为一个空字符null，表示用户还没有挑选好CD唱盘。严格地说，“\0”并不是必需的，但它能够保证这个变量确实被初始化了，而这总是一件好事情。变量current_cat用来记录当前CD唱盘的分类编号。

```
static char current_cd[MAX_STRING] = "\0" ;
static char current_cat[MAX_STRING] ;
```

3) 下面是各种文件名的声明。为了集中大家的注意力，我们在这个版本里使用的文件都是固定的，比如临时文件的文件名。如果有两个用户同时在同一个子目录里运行这个程序，就会出问题。

> 要想指定数据库文件的名字，当然还有更好的办法。我们可以把它用做程序的参数，还可以通过环境变量传递到程序中来。临时文件名的生成办法还可以再做改进，比方说，我们可以用POSIX技术规范提供的tmpnam函数来生成一个独一无二的临时文件名。我们将在今后的版本里逐步解决这些问题。

```
const char *title_file = "title.cdb";
const char *tracks_file = "tracks.cdb";
const char *temp_file = "cdb.tmp";
```

4) 接下来，我们给出所有函数的预定义。

```
void clear_all_screen(void);
void get_return(void);
int get_confirm(void);
int getchoice(char *greet, char *choices[]);
void draw_menu(char *options[], int highlight,
               int start_row, int start_col);
void insert_title(char *cdtitle);
void get_string(char *string);
void add_record(void);
void count_cds(void);
void find_cd(void);
void list_tracks(void);
void remove_tracks(void);
void remove_cd(void);
void update_cd(void);
```

5) 在看到它们的实现之前，我们需要一些菜单结构（实际上是一个菜单选项的数组），后面的工作就是分析这些菜单选项是如何实现的。第一个字符是该选项被选中时将要返回的字符，文字将被显示在屏幕上。用户挑选好一张CD唱盘后，程序将显示第二个菜单。

```
char *main_menu[] =
{
    "add new CD",
    "find CD",
    "count CDs and tracks in the catalog",
    "quit",
    0,
};

char *extended_menu[] =
{
    "add new CD",
    "find CD",
    "count CDs and tracks in the catalog",
    "list tracks on current CD",
    "remove current CD",
    "update track information",
    "quit",
    0,
};
```

以上全都是初始化方面的工作。下面是程序中的函数，我们先把这些函数的内在联系总结一下，它们一共有15个见图6-9。这些函数分成三大部分：

- 绘制菜单。
- 把CD唱盘资料添加到数据库里。
- 检索和显示CD数据。

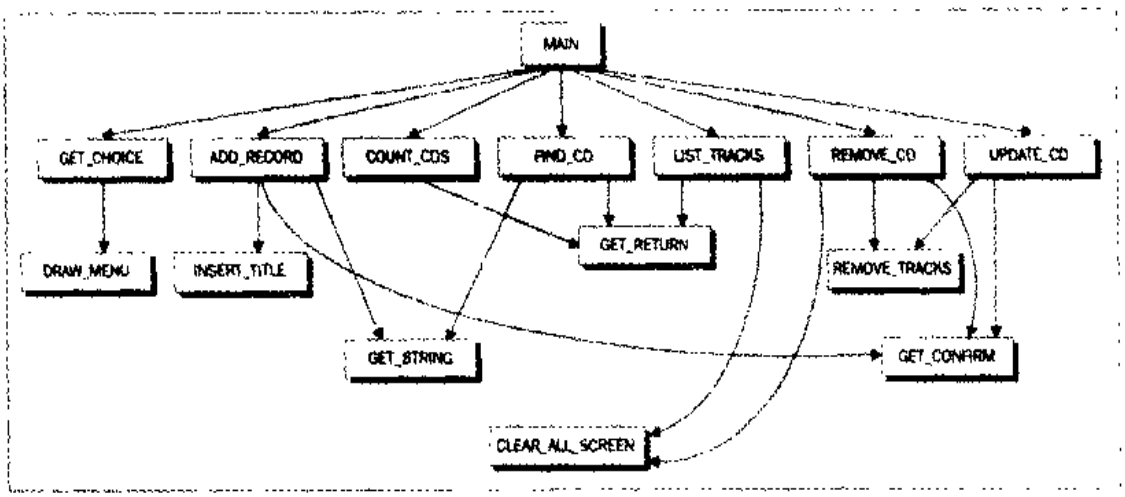


图 6-9

动手试试：CD唱盘管理软件的main函数

main函数允许用户从菜单里对操作进行选择，直到选中“quit”退出为止。下面是它的程序清单：

```
int main()
{
    int choice;
    initscr();
    do {
        choice = getchoice("Options:",
                           current_cd[0] ? extended_menu : main_menu);
```

```
            switch (choice) {
            case 'q':
                break;
            case 'a':
                add_record();
                break;
            case 'c':
                count_cds();
                break;
            case 'f':
                find_cd();
                break;
            case 'l':
                list_tracks();
                break;
            case 'r':
                remove_cd();
                break;
            case 'u':
                update_cd();
                break;
            }
        } while (choice != 'q');
        endwin();
        exit(EXIT_SUCCESS);
    }
```

下面开始对三个部分的函数进行分析。我们先来看看这个程序里与用户操作界面有关的三个函数。

动手试试：菜单

1) 被main调用的getchoice函数是这一小节里的主要函数。getchoice的参数有greet——使用方法介绍、choices——它指向主菜单或扩展菜单（这取决于用户是否挑选了一张CD唱盘）。从上面的main函数里可以看出这一点。

```
int getchoice(char *greet, char *choices[])
{
    static int selected_row = 0;
    int max_row = 0;
    int start_screenrow = MESSAGE_LINE, start_screencol = 10;
    char **option;
    int selected;
    int key = 0;

    option = choices;
    while (*option) {
        max_row++;
        option++;
    }

    /* protect against menu getting shorter when CD deleted */
    if (selected_row >= max_row)
        selected_row = 0;
    clear_all_screen();
    mvprintw(start_screenrow - 2, start_screencol, greet);
    keypad(stdscr, TRUE);
    cbreak();
    noecho();
    key = 0;
    while (key != 'q' && key != KEY_ENTER && key != '\n') {
        if (key == KEY_UP) {
            if (selected_row == 0)
                selected_row = max_row - 1;
            else
```

```
                selected_row--;
            }
            if (key == KEY_DOWN) {
                if (selected_row == (max_row - 1))
                    selected_row = 0;
                else
                    selected_row++;
            }
            selected = *choices[selected_row];
            draw_menu(choices, selected_row, start_screenrow,
                                start_screencol);
            key = getch();
        }
        keypad(stdscr, FALSE);
        nocbreak();
        echo();

        if (key == 'q')
            selected = 'q';

        return (selected);
    }
```

2) getchoice函数里还有两个局部函数，它们是clear_all_screen和draw_ menu。请注意它们的用法和用途。我们先来看看draw_ menu函数：

```
void draw_menu(char *options[], int current_highlight,
               int start_row, int start_col)
{
    int current_row = 0;
    char **option_ptr;
    char *txt_ptr;

    option_ptr = options;
    while (*option_ptr) {
        if (current_row == current_highlight) {
            mvaddch(start_row + current_row, start_col - 3, ACS_BULLET);
            mvaddch(start_row + current_row, start_col + 40, ACS_BULLET);
        } else {
            mvaddch(start_row + current_row, start_col - 3, ' ');
            mvaddch(start_row + current_row, start_col + 40, ' ');
        }

        txt_ptr = options[current_row];
        txt_ptr++;
        mvprintw(start_row + current_row, start_col, "%s", txt_ptr);
        current_row++;
        option_ptr++;
    }

    mvprintw(start_row + current_row + 3, start_col,
             "Move highlight then press Return ");
    refresh();
}
```

3) clear_all_screen的作用是清屏并重新显示软件标题。如果用户挑选了一张CD唱盘，就把它的资料显示在屏幕上。下面是它的程序清单：

```
void clear_all_screen()
{
    clear();
    mvprintw(2, 20, "%s", "CD Database Application");
    if (current_cd[0]) {
        mvprintw(ERROR_LINE, 0, "Current CD: %s, %s\n",
                 current_cat, current_cd);
    }
    refresh();
}
```

下面是对CD唱盘数据库进行添加或更新处理的函数。被main调用是函数有三个，它们是add_record、update_cd和remove_cd，它们会调用其他小节里的一些函数。我们先来看看add_record，它里面也有几个局部函数。

动手试试：对数据库文件进行处理

1) 首先，怎样把一张新CD唱盘的资料添加到数据库里去？

```
void add_record()
{
    char catalog_number[MAX_STRING];
    char cd_title[MAX_STRING];
    char cd_type[MAX_STRING];
    char cd_artist[MAX_STRING];
    char cd_entry[MAX_STRING];

    int screenrow = MESSAGE_LINE;
    int screencol = 10;

    clear_all_screen();
    mvprintw(screenrow, screencol, "Enter new CD details");
    screenrow += 2;

    mvprintw(screenrow, screencol, "Catalog Number: ");
    get_string(catalog_number);
    screenrow++;

    mvprintw(screenrow, screencol, "      CD Title: ");
    get_string(cd_title);
    screenrow++;

    mvprintw(screenrow, screencol, "       CD Type: ");
    get_string(cd_type);
    screenrow++;

    mvprintw(screenrow, screencol, "        Artist: ");
    get_string(cd_artist);
    screenrow++;

    mvprintw(PROMPT_LINE-2, 5, "About to add this new entry:");
    sprintf(cd_entry, "%s,%s,%s,%s",
            catalog_number, cd_title, cd_type, cd_artist);
    mvprintw(PROMPT_LINE, 5, "%s", cd_entry);
    refresh();
    move(PROMPT_LINE, 0);
    if (get_confirm()) {
        insert_title(cd_entry);
        strcpy(current_cd, cd_title);
        strcpy(current_cat, catalog_number);
    }
}
```

2) get_string函数的作用是从屏幕的当前位置读入一个字符串。它还会把字符串末尾的换行符去掉。下面是它的程序清单：

```
void get_string(char *string)
{
    int len;

    wgetnstr(stdscr, string, MAX_STRING);
    len = strlen(string);
    if (len > 0 && string[len - 1] == '\n')
        string[len - 1] = '\0';
}
```

3) get_confirm函数的作用是读入并判断用户的确认输入。它读入用户的输入字符串，检查它的第一个字母是否是“Y”或“y”。如果是其他字符，它就认为用户没有确认。下面是它的程序清单：

```
int get_confirm()
{
    int confirmed = 0;
    char first_char;

    mvprintw(Q_LINE, 5, "Are you sure? ");
    clrtoeol();
    refresh();

    cbreak();
    first_char = getch();
    if (first_char == 'Y' || first_char == 'y') {
        confirmed = 1;
    }
    nocbreak();

    if (!confirmed) {
        mvprintw(Q_LINE, 1, "    Cancelled");
        clrtoeol();
        refresh();
        sleep(1);
    }
    return confirmed;
}
```

4) insert_title函数的作用是在CD唱盘数据库里添加一个标题记录，具体做法是把新标题字符串追加到标题文件的末尾。下面是它的程序清单：

```
void insert_title(char *cdtitle)
{
    FILE *fp = fopen(title_file, "a");
    if (!fp) {
        mvprintw(ERROR_LINE, 0, "cannot open CD titles database");
    } else {
        fprintf(fp, "%s\n", cdtitle);
        fclose(fp);
    }
}
```

5) main调用的另外一个文件处理函数是update_cd。这个函数使用了一个带边框的卷屏子窗口，还会用到一些常数。我们把这些常数定义为全局性的，因为后面的list_tracks函数还要用到它们。这些常数如下所示：

```
#define BOXED_LINES     11
#define BOXED_ROWS      60
#define BOX_LINE_POS    8
#define BOX_ROW_POS     2
```

update_cd允许用户重新输入当前CD唱盘上的曲目。在删除从前的曲目资料后，它会提示用户输入新资料。下面是它的程序清单：

```
void update_cd()
{
    FILE *tracks_fp;
    char track_name[MAX_STRING];
    int len;
    int track = 1;
    int screen_line = 1;
    WINDOW *box_window_ptr;
```

```
    WINDOW *sub_window_ptr;

    clear_all_screen();
    mvprintw(PROMPT_LINE, 0, "Re entering tracks for CD. ");
    if (!get_confirm())
        return;
    move(PROMPT_LINE, 0);
    clrtoeol();

    remove_tracks();

    mvprintw(MESSAGE_LINE, 0, "Enter a blank line to finish");

    tracks_fp = fopen(tracks_file, "a");
```

注意 这段文字前后的程序清单是连着的。我们想用这个小停顿提醒大家注意我们是如何往一个带边框的卷屏窗口里输入数据的。这里使用的技巧是：先创建一个子窗口，围着它四周画一个边框；然后，在这个加上边框的子窗口里再创建一个能够卷屏的子窗口。最后的结果就是我们在运行程序时看到的样子了。

```
    box_window_ptr = subwin(stdscr, BOXED_LINES + 2, BOXED_ROWS + 2,
                            BOX_LINE_POS - 1  BOX_ROW_POS - 1);
    if (!box_window_ptr)
        return;
    box(box_window_ptr, ACS_VLINE, ACS_HLINE);

    sub_window_ptr = subwin(stdscr, BOXED_LINES, BOXED_ROWS,
                            BOX_LINE_POS, BOX_ROW_POS);
    if (!sub_window_ptr)
        return;
    scrollok(sub_window_ptr, TRUE);
    werase(sub_window_ptr);
    touchwin(stdscr);

    do {
        mvwprintw(sub_window_ptr, screen_line++, BOX_ROW_POS + 2,
                  "Track %d: ", track);
        clrtoeol();
        refresh();
        wgetnstr(sub_window_ptr, track_name, MAX_STRING);
        len = strlen(track_name);
        if (len > 0 && track_name[len - 1] == '\n')
            track_name[len - 1] = '\0';
        if (*track_name)
            fprintf(tracks_fp, "%s,%d,%s\n",
                    current_cat, track, track_name);
        track++;
        if (screen_line > BOXED_LINES - 1) {
            /* time to start scrolling */
            scroll(sub_window_ptr);
            screen_line--;
        }
    } while (*track_name);
    delwin(sub_window_ptr);

    fclose(tracks_fp);
}
```

6) main调用的最后一个文件处理函数是remove_cd。下面是它的程序清单：

```
void remove_cd()
{
    FILE *titles_fp, *temp_fp;
    char entry[MAX_ENTRY];
```

```
    int cat_length;

    if (current_cd[0] == '\0')
        return;

    clear_all_screen();
    mvprintw(PROMPT_LINE, 0, "About to remove CD %s: %s. ",
             current_cat, current_cd);
    if (!get_confirm())
        return;

    cat_length = strlen(current_cat);

    /* Copy the titles file to a temporary, ignoring this CD */
    titles_fp = fopen(title_file, "r");
    temp_fp = fopen(temp_file, "w");

    while (fgets(entry, MAX_ENTRY, titles_fp)) {
        /* Compare catalog number and copy entry if no match */
        if (strncmp(current_cat, entry, cat_length) != 0)
            fputs(entry, temp_fp);
    }
    fclose(titles_fp);
    fclose(temp_fp);

    /* Delete the titles file, and rename the temporary file */
    unlink(title_file);
    rename(temp_file, title_file);

    /* Now do the same for the tracks file */
    remove_tracks();

    /* Reset current CD to 'None' */
    current_cd[0] = '\0';
}
```

7) remove_tracks函数的作用是删除当前CD唱盘的曲目资料。update_cd和remove_cd两个函数都会调用它。下面是它的程序清单：

```
void remove_tracks()
{
    FILE *tracks_fp, *temp_fp;
    char entry[MAX_ENTRY];
    int cat_length;

    if (current_cd[0] == '\0')
        return;

    cat_length = strlen(current_cat);

    tracks_fp = fopen(tracks_file, "r");
if (!tracks_fp)
  return;
    temp_fp = fopen(temp_file, "w");

    while (fgets(entry, MAX_ENTRY, tracks_fp)) {
        /* Compare catalog number and copy entry if no match */
        if (strncmp(current_cat, entry, cat_length) != 0)
            fputs(entry, temp_fp);
    }
    fclose(tracks_fp);
    fclose(temp_fp);

    /* Delete the tracks file, and rename the temporary file */
    unlink(tracks_file);
    rename(temp_file, tracks_file);
}
```

动手试试：对CD数据库进行查询

1) 最基本的查询就是了解你收集和拥有的东西到底有多少。这正好是下面这个函数的功能。它会对数据库进行扫描并对统计出总的唱盘张数和曲目个数来。下面是它的程序清单：

```
void count_cds()
{
    FILE *titles_fp, *tracks_fp;
    char entry[MAX_ENTRY];
    int titles = 0;
    int tracks = 0;

    titles_fp = fopen(title_file, "r");
    if (titles_fp) {
        while (fgets(entry, MAX_ENTRY, titles_fp))
            titles++;
        fclose(titles_fp);
    }
    tracks_fp = fopen(tracks_file, "r");
    if (tracks_fp) {
        while (fgets(entry, MAX_ENTRY, tracks_fp))
            tracks++;
        fclose(tracks_fp);
    }
    mvprintw(ERROR_LINE, 0,
             "Database contains %d titles, with a total of %d tracks.",
             titles, tracks);
    get_return();
}
```

2) 把自己最喜欢的CD唱盘的袖签弄丢了？！别着急！你不是早就把资料输到电脑里了嘛。你可以用find_cd函数查出曲目清单来。它提示用户输入一个字符串，根据这个字符串在数据库里进行匹配检索，并把找到的CD唱盘标题放到全局变量current_cd里去。下面就是它的程序清单：

```
void find_cd()
{
    char match[MAX_STRING], entry[MAX_ENTRY];
    FILE *titles_fp;
    int count = 0;
    char *found, *title, *catalog;

    mvprintw(Q_LINE, 0, "Enter a string to search for in CD titles: ");
    get_string(match);

    titles_fp = fopen(title_file, "r");
    if (titles_fp) {
        while (fgets(entry, MAX_ENTRY, titles_fp)) {

            /* Skip past catalog number */
            catalog = entry;
            if (found == strstr(catalog, ",")) {
                *found = '\0';
                title = found + 1;

                /* Zap the next comma in the entry to reduce it to
                   title only */
                if (found == strstr(title, ",")) {
                    *found = '\0';

                    /* Now see if the match substring is present */
                    if (found == strstr(title, match)) {
                        count++;
                        strcpy(current_cd, title);
```

```
                        strcpy(current_cat, catalog);
                    }
                }
            }
        }
        fclose(titles_fp);
    }
    if (count != 1) {
        if (count == 0) {
            mvprintw(ERROR_LINE, 0, "Sorry, no matching CD found. ");
        }
        if (count > 1) {
            mvprintw(ERROR_LINE, 0,
                     "Sorry, match is ambiguous: %d CDs found. ", count);
        }
        current_cd[0] = '\0';
        get_return();
    }
}
```

虽然catalog指向的数组比current_cat要大并且绝对有可能会覆盖内存，但fgets中的检查弥补了这一问题。

3) 我们还需要能够把挑选到的CD唱盘里的曲目在屏幕上列出来。这里再次用到上一小节的update_cd函数里为子窗口安排的常数定义。

```
void list_tracks()
{
    FILE *tracks_fp;
    char entry[MAX_ENTRY];
    int cat_length;
    int lines_op = 0;
    WINDOW *track_pad_ptr;
    int tracks = 0;
    int key;
    int first_line = 0;

    if (current_cd[0] == '\0') {
        mvprintw(ERROR_LINE, 0, "You must select a CD first. ");
        get_return();
        return;
    }
    clear_all_screen();
    cat_length = strlen(current_cat);

    /* First count the number of tracks for the current CD */
    tracks_fp = fopen(tracks_file, "r");
    if (!tracks_fp)
        return;
    while (fgets(entry, MAX_ENTRY, tracks_fp)) {
        if (strncmp(current_cat, entry, cat_length) == 0)
            tracks++;
    }
    fclose(tracks_fp);

     /* Make a new pad, ensure that even if there is only a single
       track the PAD is large enough so the later prefresh() is always
       valid.
     */
    track_pad_ptr = newpad(tracks + 1 + BOXED_LINES, BOXED_ROWS + 1);
    if (!track_pad_ptr)
        return;

    tracks_fp = fopen(tracks_file, "r");
    if (!tracks_fp)
        return;

    mvprintw(4, 0, "CD Track Listing\n");
```

```
    /* write the track information into the pad */
    while (fgets(entry, MAX_ENTRY, tracks_fp)) {
        /* Compare catalog number and output rest of entry */
        if (strncmp(current_cat, entry, cat_length) == 0) {
            mvwprintw(track_pad_ptr, lines_op++, 0, "%s",
                      entry + cat_length + 1);
        }
    }
    fclose(tracks_fp);

    if (lines_op > BOXED_LINES) {
        mvprintw(MESSAGE_LINE, 0,
                 "Cursor keys to scroll, RETURN or q to exit");
    } else {
        mvprintw(MESSAGE_LINE, 0, "RETURN or q to exit");
    }
    wrefresh(stdscr);
    keypad(stdscr, TRUE);
    cbreak();
    noecho();
    key = 0;
    while (key != 'q' && key != KEY_ENTER && key != '\n') {
        if (key == KEY_UP) {
            if (first_line > 0)
                first_line--;
        }
        if (key == KEY_DOWN) {
            if (first_line + BOXED_LINES + 1 < tracks)
                first_line++;
        }
        /* now draw the appropriate part of the pad on the screen */
        prefresh(track_pad_ptr, first_line, 0,
                 BOX_LINE_POS, BOX_ROW_POS,
                 BOX_LINE_POS + BOXED_LINES, BOX_ROW_POS + BOXED_ROWS);
        key = getch();
    }

    delwin(track_pad_ptr);
    keypad(stdscr, FALSE);
    nocbreak();
    echo();
}
```

4) 最后这个函数叫做get_return，它的作用是提示用户按下回车键并读取它，其他字符将被忽略。下面是它的程序清单：

```
void get_return()
{
    int ch;
    mvprintw(23, 0, "%s", " Press return ");
    refresh();
    while ((ch = getchar()) != '\n' && ch != EOF);
}
```

运行这个程序，我们将看到如图6-10所示的屏幕输出情况。

6.16 本章总结

在这一章里，我们对curses函数库进行了比较全面的学习。curses为基于文本的程序提供了控制屏幕输出和读取键盘输入的好办法。与使用通用终端接口（GTI）和直接访问terminfo数据库等手段相比，虽然curses提供的控制功能没有那么多，但在简单易用方面却遥遥领先。如果你正在编写一个基于文本的全屏显示软件，就应该考虑使用curses函数库为你管理屏幕和

键盘。

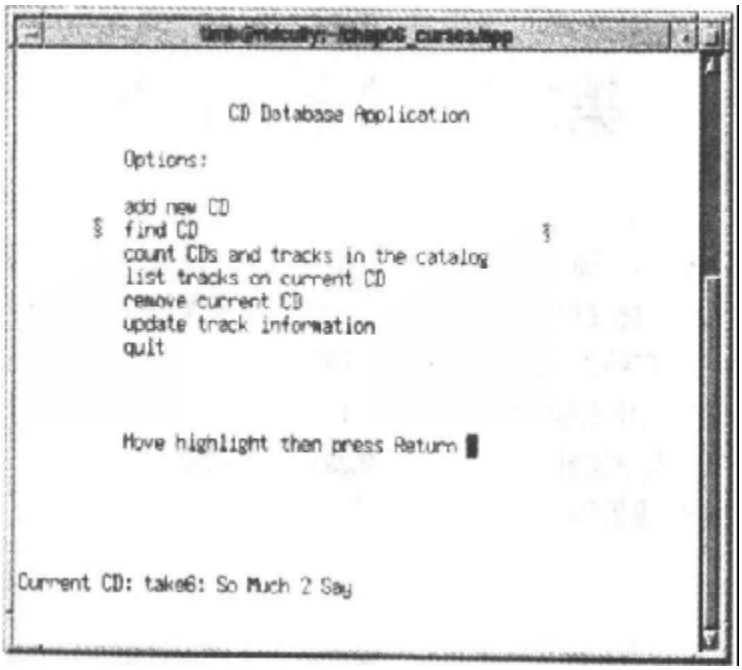


图 6-10

第7章 数据管理

我们在第3章学习了文件，在第4章接触到资源限制的问题。在这一章里，我们将从资源分配的管理方面入手；然后学习如何对可能被多个用户同时访问的文件进行处理；最后向大家介绍一个大多数UNIX系统都能提供的工具，它克服了普通数据文件的局限性。

我们可以把这些问题归纳为数据管理的三个方面：

- 动态内存管理：该做什么和UNIX不让你做什么。
- 文件锁：协调锁、共享文件的封锁区和避免造成死锁现象。
- dbm数据库：UNIX提供的一个数据库方面的函数库。

7.1 内存管理

在任何计算机系统上内存都是珍稀资源。不管插了多少内存条，它总是不够用。在并非很久远的过去，人们甚至曾经认为一兆字节的内存已经超出了任何人的实际需要，可如今，64倍于此的内存倒被看做是单用户个人电脑的最低要求了。许多系统上的内存配置已经大大超出了这个数字。

从最早期的操作系统版本开始，UNIX就以一种非常聪明的办法管理着内存。UNIX应用程序决不允许直接访问物理内存。也许应用程序看起来好象可以这样做，但应用程序看到的只是一个精心控制的假象而已。

UNIX提供给进程的永远是一个平整的连续的内存模型——这就是每个进程都能够“看见”的属于它自己的那块内存区域。几乎所有版本的UNIX操作系统都提供了内存保护机制，它保证不正确的（或者恶意的）程序无法覆盖属于其他进程或者属于操作系统的内存。在一般情况下，分配给一个进程的内存既不能被任何其他的进程读，也不能被它们写。几乎所有版本的UNIX都使用了硬件机制实施这种内存使用方面的私用性。

7.1.1 简单的内存分配机制

我们可以通过C语言标准库中的malloc调用来分配内存，它的定义如下所示：

```
# include <stdlib.h>
void *malloc( size_t size );
```

请注意，X/Open技术规范在这里与某些UNIX的具体版本有一个差异，就是它不要求有一个专用的malloc.h头文件。还要一个需要注意的问题，指定待分配内存字节数的size参数并不是一个简单的int整数，虽然它往往相当于一个不带符号的整数类型。

在大多数UNIX系统上，我们可以申请分配大量的内存，请看下面的程序：

动手试试：简单的内存分配操作

把下面这个memory1.c程序敲进计算机：

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

#define A_MEGABYTE (1024 * 1024)

int main()
{
    char *some_memory;
    int  megabyte = A_MEGABYTE;
    int exit_code = EXIT_FAILURE;

    some_memory = (char *)malloc(megabyte);
    if (some_memory != NULL) {
        sprintf(some_memory, "Hello World\n");
        printf("%s", some_memory);
        exit_code = EXIT_SUCCESS;
    }
    exit(exit_code);
}
```

当我们运行这个程序的时候，它将输出：

```
$ memory1
Hello World
```

操作注释：

这个程序请求malloc函数库给它分配一个指针，指针指向一个一兆字节的内存空间。我们对此进行了检查以确定malloc函数的确是执行成功了，随后还使用了其中的一些内存以证明它确实是存在的。当我们运行这个程序的时候，会看到程序输出的欢迎辞“Hello World”，这表示malloc确确实实地返回了一个一兆字节的可用内存。我们没有对这一兆字节进行全面检查，对malloc函数的代码总得有点信任度吧！

注意，因为malloc返回的是一个“void *”指针，所以我们把这个结果映射到我们需要的“char *”指针上去。malloc函数能够保证它所返回的内存是线性的，所以它可以被映射到任何类型的指针上去。

其实，之所以会这样的原因很简单：现如今大多数的Linux系统和UNIX系统使用的都是32位整数、指向内存的指针也是32位的，能够让用户设定多达4G的字节。系统直接使用32位的指针来寻址，不再需要段寄存器或其他技巧，这种能力用术语表达就叫做“32位平面内存模型”。Windows NT/2000和Windows 9x的32位系统使用的也是这种模型。有些UNIX操作系统被限制在16位上，但它们本来就不多，现在就更少见了。但是，我们还不能放心地认为整数永远是32位的，因为使用64位Linux和UNIX版本的系统数量正不断增加着。

7.1.2 分配大量的内存

既然我们已经看到UNIX能够轻松地超越DOS内存模型的长度限制，现在不妨给它出个难题。下面的程序将逐步申请非常多的内存，一直申请到超出机器本身拥有的物理内存容量为止。我们可以预期malloc会在接近实际物理内存容量的某个地方开始出现问题，因为内核和其他运行中

的进程也应该使用一部分内存。

动手试试：申请全部的物理内存

我们用memory2.c程序逐步申请机器上的全部内存：

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

#define A_MEGABYTE (1024 * 1024)

int main()
{
    char *some_memory;
    size_t  size_to_allocate = A_MEGABYTE;
    int  megs_obtained = 0;

    while (megs_obtained < 512) {
        some_memory = (char *)malloc(size_to_allocate);
        if (some_memory != NULL) {
            megs_obtained++;
            sprintf(some_memory, "Hello World");
            printf("%s - now allocated %d Megabytes\n", some_memory, megs_obtained);
        }
        else {
            exit(EXIT_FAILURE);
        }
    }
    exit(EXIT_SUCCESS);
}
```

这个程序的输出如下所示，我们省略了首尾之间的一大部分：

```
$ memory2
Hello World - now allocated 1 Megabytes
Hello World - now allocated 2 Megabytes
...
Hello World - now allocated 511 Megabytes
Hello World - now allocated 512 Megabytes
```

操作注释：

这个程序和前面的那个很相似。它简单地通过一个循环不断申请越来越多的内存。令人吃惊的是它自始至终没有出现错误，可我们的做法看起来就好象是已经创建了一个耗尽了作者计算机内存每一个字节的程序呀。请大家注意，我们在malloc调用里使用的是size_t类型，在现如今的Linux具体实现中，它实际上是一个“unsigned int”整数。

另外一个值得注意的事情是：至少在我的这台机器上，整个程序的运行时间也就是一眨眼的工夫。也就是说，我们不仅很明显地用尽了内存，而还是非常之快地做到了这一点。

我们用memory3.c做进一步的研究，看看这台机器上到底有多少内存可以被分配。上面的结果很明显地告诉我们UNIX在申请和分配内存方面比较精明，所以这次我们1K字节1K字节地对内存进行分配，并且要把我们分得的每一块内存都写上点东西。

动手试试：可用内存

下面就是memory3.c程序的源代码。从其本质来说，这个程序对系统极不友好，并且对一台

多用户计算机造成恶劣影响。如果读者对可能的风险有所顾虑，最好不要运行它。这应该不会妨碍读者对这部分内容的理解。

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

#define ONE_K (1024)

int main()
{
    char *some_memory;
    int  size_to_allocate = ONE_K;
    int  megs_obtained = 0;
    int  ks_obtained = 0;

    while (1) {
        for (ks_obtained = 0; ks_obtained < 1024; ks_obtained++) {
            some_memory = (char *)malloc(size_to_allocate);
            if (some_memory == NULL) exit(EXIT_FAILURE);
            sprintf(some_memory, "Hello World");
        }
        megs_obtained++;
        printf("Now allocated %d Megabytes\n", megs_obtained);
    }
    exit(EXIT_SUCCESS);
}
```

这一次，程序的简略输出是下面这样的：

```
$ memory3
Now allocated 1 Megabytes
...
Now allocated 153 Megabytes
Now allocated 154 Megabytes
```

接下去程序就结束了。它花费的时间挺不少，而且在接近机器中物理内存容量极限的时候明显地慢了下来。但这个程序还是分配了大大超出作者机器物理内存容量的内存。

操作注释：

应用程序正在分配的内存是由UNIX操作系统的内核管理着的。每当程序申请分配内存或者对已分配内存进行读写时，如何处理这些请求的决定都是由UNIX操作系统的内核做出的。

在刚开始的时候，内核还可以简单地用空闲着的物理内存满足应用程序的内存需求，但当物理内存用完之后，它就将开始使用所谓的“交换空间”（swap space）。在UNIX的大多数版本上，交换空间指的是一块独立的硬盘空间。如果你熟悉微软公司的Windows，就能看出UNIX交换空间和微软Windows的交换文件（swap file）之间的相似之处。但与微软Windows不同的是：在UNIX的交换空间里没有局部堆（local heap）、全局堆（global heap）或其他可丢弃内存段等让人操心的东西——UNIX操作系统的内核把这些管理工作都替用户包下来了。

在物理内存和交换空间之间挪动数据和程序代码的工作完全由内核来负责，因此，每当用户对内存进行读写的时候，数据总象是早已等在物理内存里了，而事实上它是在用户准备访问它之前刚刚分配或交换过来的。

用更专业一点的术语来说，UNIX实现了一个“请求页面虚拟内存系统”。用户程序看到的

所有内存都是虚拟的，即在程序使用的物理地址上并不存在真实的内存。UNIX把所有内存分成一页一页的，一页通常是4096个字节。每当程序试图访问内存的时候，就会出现一次虚拟内存和物理内存的转换，它的具体做法和所花费的时间将取决于用户使用的硬件的具体情况。如果被访问的内存没有在物理内存空间中，就会产生一个页面错误（page fault），而控制权就会上交给UNIX操作系统的内核。

UNIX对被访问内存的地址进行检查，如果这是一个允许该程序使用的合法地址，它就会确定需要向程序提供哪一个物理内存页面、然后，如果该数据从没被写过，就为它新分配一个内存页面；如果数据已经被保存到硬盘的交换空间里去了，就把包含该数据的内存页面读回物理内存（可能需要把一个现存页面转移到硬盘上去）。接着，在把虚拟内存地址映射到与之对应的物理地址之后，它再让用户程序继续执行。这些操作不需要UNIX应用程序本身去操心，因为这一切都隐藏在UNIX操作系统的内核里。

最终，如果应用程序耗尽了物理内存和交换空间，或者如果堆栈超出其最大长度，UNIX操作系统的内核就会拒绝此后的内存申请。

那么，这一切对应用程序的程序员意味着什么呢？简单地说，这是件大好事。UNIX非常擅于管理内存，能够允许应用程序使用数量非常巨大的内存，甚至一整个非常大的内存块。但我们也必须记住，分配了两块内存并不见得会肯定得到一个能够连续寻址的内存块。我们得到的就是我们想要的：两个分开的内存块。

根据这种做法，内存的供应量明显地是没有极限的，那么，这是不是意味着对malloc返回情况的检查没有意义了呢？绝不是。使用动态分配地址的C语言程序经常会出现这样一个常见的问题，即试图在某个已分配内存块以外的地方写数据。当这种情况发生的时候，程序并不会立刻终止，但很有可能已经覆盖了malloc函数库例程内部使用的某些数据。

出现这种问题之后，常见的结果是后续的malloc调用无法继续进行，不是因为没有内存可供分配，而是因为内存的结构被破坏了。追踪这类问题是相当困难的；并且，在程序里发现问题越早，找到并解决问题的机会也就越大。在讨论调试和优化的第9章内容里，我们将向大家介绍一些帮助你追踪内存错误的工具。

7.1.3 内存的滥用

我们现在对内存干点“坏事”。在程序memory4.c里，我们分配一些内存并尝试在它的首尾以外的地方写数据。

动手试试：滥用内存

```
#include <stdlib.h>

#define ONE_K (1024)

int main()
{
    char *some_memory;
    char *scan_ptr;

    some_memory = (char *)malloc(ONE_K);
```

```
    if (some_memory == NULL) exit(EXIT_FAILURE);

    scan_ptr = some_memory;
    while(1) {
        *scan_ptr = '\0';
        scan_ptr++;
    }
    exit(EXIT_SUCCESS);
}
```

程序的输出很简单，如下所示：

```
$ memory4
segmentation fault (core dumped)
```

操作注释：

UNIX的内存管理系统防止系统的其他部分受到这个内存滥用的影响。在确信一个行为恶劣的程序（这就是一个）有可能损害其他程序时，UNIX立刻停止它的运行。

运行在一个UNIX系统上的每一个程序都只能看到它自己的内存映象，这个映象与其他程序的不一样。只要操作系统知道物理内存是如何安排的，它不仅能够为用户程序管理内存，还可以对用户程序提供隔离保护。

7.1.4 空指针

现代的UNIX系统与MS-DOS不一样，它们对空指针的读写有着很强的戒备心理，但对这种情况的实际处理过程不同版本有不同的做法。

> 动手试试：访问一个空指针

我们用memory5a.c程序来看看对空指针进行读写时会出现什么情况：

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

int main()
{
    char *some_memory = (char *)0;

    printf("A read from null %s\n", some_memory);
    sprintf(some_memory, "A write to null\n");
    exit(EXIT_SUCCESS);
}
```

程序的输出如下所示：

```
$ memory5a
A read from null (null)
Segmentation fault(core dumped)
```

操作注释：

第一个printf试图输出一个取自一个空指针的字符串，接下来的sprintf试图对一个空指针进行写操作。在这种情况下，Linux容忍了读操作（这要感谢GNU的C语言库），让程序输出了一个

其实并不存在的字符串，这个字符串“神奇”地包含着“(null)\0”等字符。但它对写操作就没有那么宽容了，结果是终止了这个程序的运行。这在某些时候能够帮助我们追踪程序中的漏洞。

我们可以再试一次，但这次我们不使用GNU的C语言库。我们发现读零地址的操作也不能进行了。请看下面的memory5b.c程序：

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

int main()
{
        char z = *(const char *)0;
         printf("I read from location zero\n");

        exit(EXIT_SUCCESS);
}
```

程序的输出如下所示：

```
$ memory5b
Segmentation fault (core dumped)
```

操作注释：

这一次我们尝试直接读取零地址。这一次GNU的libc函数库没有夹在我们和操作系统内核中间，程序被终止了。读者会发现某些版本的UNIX操作系统确实允许从零地址处读取数据，但Linux不允许。

7.1.5 内存的释放

截止到目前，我们所做的事情只是简单地分配内存，并且希望到程序结束的时候我们用过的内存还没有流失。幸运的是UNIX的内存管理系统有能力在一个程序结束运行的时候把分配给它的内存收归系统。可大多数程序需要的并不是简单地申请一些内存，使用一小段时间，然后再退出。更常见的做法是在必要的时候动态地使用内存。

动态使用内存的程序必须随时注意使用free调用把当时用不着的内存块释放给malloc内存管理程序。这样做可以把彼此隔离的内存块重新合并到一起，并由malloc函数库去照顾内存，应用程序不必掺和到内存管理方面的事情里。如果由一个运行中的程序（即进程）负责自己的内存管理，在它使用并随后释放内存时，这块自由内存实际上仍将处于被分配给该程序的状态。如果这块内存当时没有被使用，UNIX的内存管理程序就可能会把它从物理内存通过页面交换放到交换空间去。内存被放到交换空间以后，它对资源的使用情况就没有什么太大的影响了。

```
# include <stdlib.h>

void free(void *ptr_to_memory );
```

调用free时使用的指针必须指向用malloc、calloc或者realloc调用分配的某段内存。calloc和realloc马上就会介绍到。

动手试试：释放内存

我们给这个程序起名为memory6.c。下面是它的程序清单：

```
#include <stdlib.h>

#define ONE_K (1024)

int main()
{
    char *some_memory;
    int exit_code = EXIT_FAILURE;

    some_memory = (char *)malloc(ONE_K);
    if (some_memory != NULL) {
        free(some_memory);
        exit_code = EXIT_SUCCESS;
    }
    exit(exit_code);
}
```

操作注释：

这个程序简单地显示了怎样调用带有指向刚分配到的内存的指针参数的free调用。

> 注意：一旦调用free释放了一块内存，它就不再属于这个进程了。它将由malloc函数库负责管理。在对一块内存做过free调用之后，就绝不要再对它进行读写了。

7.1.6 其他内存分配函数

另外还有两个可以用来分配内存的函数，但它们不象malloc和free使用的那么频繁。这就是calloc和realloc。它们的定义如下所示：

```
#include <stdlib.h>

void *calloc(size_t number_of_elements, size_t element_size);
void *realloc(void *existing_memory, size_t new_size);
```

虽然calloc分配的内存也能够通过free来释放，但它有着不同的参数。它的作用是为一个结构数组分配内存，需要把数组中的元素个数和每个元素的长度尺寸作为自己的参数，而分配到的内存都填满了零。如果calloc执行成功，就返回一个指向数组中第一个元素的指针。后续calloc调用也不保证能够返回一个连续不断的内存空间，这方面它与malloc一样。因此，如果我们想扩大一个由calloc创建的数组，简单地再次调用calloc并希望返回的内存正好接在第一次调用返回的内存后面是不现实的。

realloc函数的作用是改变以前分配的内存块的长度。它要使用一个指针做参数，指针指向通过以前的malloc、calloc或realloc调用分配到的一些内存，然后根据new_size参数做出上下调整。为了完成自己的工作，realloc函数可能需要对数据进行移动，所以用realloc调用对内存做过重新分配之后，就一定要使用新的指针，绝不要再用realoc调用之前设置的指针去访问内存。

另外一个应该引起大家注意的问题是：realloc在不能对内存块长度做出调整时会返回一个空指针。这就意味着在某些应用程序里，下面这样的代码是应该避免：

```
my_ptr = malloc(BLOCK_SIZE);
....
my_ptr = realloc(my_ptr, BLOCK_SIZE * 10);
```

如果realloc函数的执行失败了，它会返回一个空指针；也就是说，my_ptr将指向null，而原先通过malloc分配的内存将无法再通过my_ptr去访问。因此，从用户的利益考虑，在释放老内存以前最好是先用malloc申请一块新内存，再用memcpy函数把老内存块里的数据拷贝到新内存块里去。这样，即使出现错误，应用程序还是能够访问到保存在原来那块内存里的数据，我们可以用这个办法让程序退出得干净利落。

7.2 文件封锁

文件封锁在多用户多任务操作系统里扮演着一个非常重要的角色。程序经常需要共享数据，而这经常是通过文件实现的；因此，给这些程序提供一个确立文件控制权的手段就十分重要了。只有这样，对文件的修改才是安全的：当第一个程序对文件进行写入时，文件会临时进入一个“不可侵犯”的状态，第二个准备读这个文件的程序会自动停下来等待前一个程序操作的完成。

UNIX为我们准备了几个用来实现文件封锁功能的工具。最简单的办法是以原子操作的形式创建一个锁文件，这个技巧的高明之处在于创建锁文件的时候系统将不允许任何其他的事件发生。这就可以保证程序有办法创建出一个确实是独一无二的文件来，而这个文件是绝不可能被其他程序在同一时刻创建出来的。

第二个办法更高级一些，它允许程序对文件的某个部分进行封锁，让自己独享文件这部分内容的访问权。在符合X/Open技术规范的UNIX版本里，有两种不同的办法可以达到这第二种封锁的目的。我们只对其中的一种做细致的分析，因为两种办法很类似——第二种办法只不过有稍微不同的程序设计接口而已。

7.2.1 创建锁文件

许多应用程序只要能够针对某项资源创建一个锁文件就心满意足了。此后，其他程序可以检查这个文件的状态，看它们自己是否被允许访问那项资源。

锁文件一般都被集中放置在一个固定的场所，并且从它们的文件名上就可以看出它们与受控资源之间的关系。比方说，在使用调制解调器的时候，Linux就会在/usr/spool/uucp子目录里创建出一个锁文件来。在大多数UNIX系统上，这个子目录被用来表明系统上有串行口存在。请看：

```
$ ls /usr/spool/uucp
LCK . . ttyS1
```

记住，锁文件的作用就象是些信号灯，它们的使用需要程序来配合。用专业术语来说，锁文件只是建议性的，与此对立的封锁功能是强制性的。

要想创建一个用做封锁信号灯的文件，我们需要使用在fcntl.h文件（我们在第3章里介绍过它）里定义的带O_CTREAT和O_EXCL标志的open系统调用。它使我们能够完成两项工作：确定文件当时并不存在和只用一个原子操作就把它创建出来。

动手试试：创建一个锁文件

我们来看看lock1.c程序里是怎样做的。下面是它的程序清单：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

int main()
{
    int file_desc;
    int save_errno;

    file_desc = open("/tmp/LCK.test", O_RDWR | O_CREAT | O_EXCL, 0444);
    if (file_desc == -1) {
        save_errno = errno;
        printf("Open failed with error %d\n", save_errno);
    }
    else {
        printf("Open succeeded\n");
    }
    exit(EXIT_SUCCESS);
}
```

我们第一次运行这个程序的时候，它会给出如下所示的输出：

```
$ lock1
Open succeeded
```

但当我们第二次运行它的时候，我们将看到如下所示的输出：

```
$ lock1
Open failed with error 17
```

操作注释：

这个程序调用open函数来创建一个名为/tmp/LCK.test的文件，并且加上了O_CREAT和O_EXCL标志。在我们第一次运行这个程序的时候，锁文件还不存在，所以open调用成功了。再次运行这个程序时操作失败了，因为那时锁文件已经存在了。如果想让程序再次执行成功，我们就必须删掉那个锁文件。

至少是在Linux系统上，第17号错误代表着EEXIST，这个错误表示某个文件是已经存在的。错误代码都定义在头文件errno.h或它所包括的文件里。我们使用的错误代码其定义为：

```
# define  EEXIST        17              /* File exists */
```

这是一个很适合表示“open(O_CREAT | O_EXCL)”操作失败的错误。

如果一个程序在它执行的时候只需独占某项资源很短的时间——用术语来说就是这个程序有一个或几个“关键节”，它就必须在进入关键节之前先创建一个锁文件，等它退出关键节的时候再用unlink删除那个锁文件。

我们来编写一个示范程序，然后同时运行它的两份拷贝，让大家看清楚程序是如何与这种封锁机制进行协调的。我们将用到第4章里介绍过的getpid调用，它的作用是返回进程的标识代码，每个当前运行中的程序都有一个这样的独一无二的数字标识。

动手试试：协调性锁文件

1) 下面是我们的测试程序lock2.c的源代码清单：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

const char *lock_file = "/tmp/LCK.test2";

int main()
{
    int file_desc;
    int tries = 10;

    while (tries--) {
        file_desc = open(lock_file, O_RDWR | O_CREAT | O_EXCL, 0444);
        if (file_desc == -1) {
            printf("%d - Lock already present\n", getpid());
            sleep(3);
        }
        else {
```

2) 下面是关键节的开始：

```
printf("%d - I have exclusive access\n", getpid());
sleep(1);
(void)close(file_desc);
(void)unlink(lock_file);
```

3) 关键节到这里结束：

```
            sleep(2);
        }
    }
    exit(EXIT_SUCCESS);
}
```

在运行这个程序之前，我们先要用下面的命令确保锁文件不存在：

```
$ rm -f /tmp/LCK.test2
```

然后用下面这条命令同时运行这个程序的两份拷贝：

```
$ lock2 & lock2
```

下面是我们看到的输出结果：

```
1284 - I have exclusive access
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
1284 - I have exclusive access
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
1284 - I have exclusive access
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
1284   I have exclusive access
1283 - Lock already present
1283   I have exclusive access
1284 - Lock already present
1284   I have exclusive access
```

```
1283 - Lock already present
1283 - I have exclusive access
1284 - Lock already present
```

从上面的结果可以看出同一程序的这两个实例是如何进行协调的。读者在做这个实验时看到的进程标识代码可能会与上面输出中的不一样，但程序的行为应该的相同的。

操作注释：

出于演示的目的，我们让程序做了十次循环，使用的是while语句。程序通过创建一个独一无二的锁文件/tmp/LCK.test2开始访问关键性资源。如果因为文件已经存在而使创建文件的操作失败了，程序会稍等片刻后再行尝试。如果它创建文件成功，就表示它已经获得了资源的访问权，能够进驻关键节完成预定的独占性操作。

因为这只是一个演示，所以我们只需等待一小会儿。当程序完成了对资源的操作后，它将删除锁文件，解除对资源的封锁。在重新施行封锁之前，它可以先进行一些其他的处理工作（比如例子里的sleep函数）。锁文件的作用就好象是一个信号量，就“我可以使用那个资源吗？”这个问题给每个程序一个“yes”或“no”的答案。我们将在第12章和第13章对信号量做进一步学习。

这是一种协调性安排，我们必须为此编写出正确的程序来，认识到这一点是很重要的。即使程序创建锁文件的操作失败了，也不允许它简单地通过先删除再创建锁文件的办法来解决问题。否则，它倒是可以创建锁文件了，可也要创建锁文件的其他程序却不知道自己已经不再拥有对资源的独占权了，结果将是“一出悲剧在上演”。

7.2.2 文件中的封锁区

用创建锁文件的办法来控制串行口这类资源的独占权是很不错的，但对一个大型的共享文件来说，再这样做就不太合适了。这种大文件确实存在，一个程序把数据源源不断地写到这个文件里，同一时间还有许多其他的程序在对它修修补补。一个程序负责记录长期以来连续收集到的数据，其他一些程序负责对这些数据进行处理，这种情况在UNIX系统里是比较常见的。负责处理数据的程序不可能等到负责记录的程序结束之后再开始运行——因为负责记录的程序根本就不会停下来，它们只能同时对同一个文件进行访问，这就需要为它们准备一些协调措施。

我们可以用文件中的封锁区来解决这个问题：文件的某个部分被封锁了，可其他程序可以去访问这个文件的其他部分。UNIX（至少）有两个办法可以“搞掂”这件事：一是使用fcntl系统调用，二是使用lockf调用。我们的讨论将主要集中在fcntl接口上，因为它相对来说更常见一些。lockf与它大同小异。

我们在第3章里已经见过fcntl调用了，下面是它的定义：

```
# include <fcntl.h>

int fcntl(int fildes, int command,  . . .  );
```

fcntl对一个打开的文件描述符进行操作，并能根据command参数的指示完成不同的任务。它为我们感兴趣的文件封锁问题准备了三条命令，它们是：

• F_GETLK命令。

• F_SETLK命令。

• F_SETLKW命令。

在使用这些命令时，还必须加上第三个参数，这是一个指向“struct flock”的指针，最终的函数定义如下所示：

```
int fcntl(int fildes, int command, struct flock *flock_structure );
```

flock（文件封锁）结构在不同版本有不同的实现方法，但至少包含以下几个成员：

• short l_type。

• short l_whence。

• off_t l_start。

• off_t l_len。

• pid_t l_pid。

l_type成员的取值是下表中的一个，它们也是在fcntl.h文件里定义的见表7-1。它们是：

表 7-1

取 值	说 明
F_RDLCK	一个共享（读操作）锁。许多不同的进程可以拥有文件同一（或者重叠）区域上的共享锁。只要有一个进程拥有一把共享锁，那么，任何进程都不可能再拥有一把该区域上的独占锁。为了获得一把共享锁，文件必须以“读”或“读/写”权限被打开
F_UNLCK	解除封锁，用来清除各种封锁
F_WRLCK	一个独占（写操作）锁。一个文件的一个区域只能由一个进程拥有一把独占锁。只要有一个进程拥有一把这样的锁，其他任何进程都不可能再获得该区域上任何类型的锁。为了获得一把独占锁，文件必须以“写”或“读/写”权限被打开

l_whence、l_start和l_len这几个成员定义了文件中的一个区域，即一个连续的字节集合。l_whence必须是SEEK_SET、SEEK_CUR或SEEK_END（它们的定义在unistd.h文件里）这几个值中的一个，这几个值分别对应着文件头、当前位置和文件尾。它定义了相对于l_start的偏移值，而l_start是该区域的第一个字节。在实际应用中，l_whence最常见的取值是SEEK_SET，这样l_start就是从文件头开始计算的。l_len参数定义了该区域里的字节个数。

l_pid参数的作用是指出哪个进程正施行着封锁，参见下面对F_GETLK的介绍。

文件中的每个字节在任一时间只能拥有一种类型的锁，可以（不能同时）有共享访问锁、独占访问锁、和解除封锁三种状态。

fcntl调用的命令和选项组合还真不少，我们将在下面依次对它们进行介绍：

1. F_GETLK命令

先来看看F_GETLK命令。它的作用是获取fildes（fcntl调用的第一个参数）打开的那个文件的封锁信息，它不会对文件进行封锁。调用者进程把自己想创建的封锁类型信息传递给fcntl，fcntl再把这些信息和F_GETLK命令做为参数开始执行，它返回的信息告诉调用者进程有哪些因素会阻止进入封锁状态。

flock结构中使用的取值见表7-2：

表 7-2

取 值	说 明
l_type	如果是共享（只读）锁则取值为F_RDLCK；如果是独占（写操作）锁则取值为F_WRLCK
l_whence	SEEK_SET、SEEK_CUR或SEEK_END三个值中的一个
l_start	文件预定封锁区的起始字节
l_len	文件预定封锁区的字节个数
l_pid	拥有锁的进程的标识代码

进程可以通过F_GETLK调用查看文件某个区域的当前封锁状态。它必须对flock结构进行设置，给出它想申请的封锁类型并定义好它准备封锁的区域。fcntl调用在操作成功时将返回一个不是“-1”的值。如果文件目前的封锁状态不允许调用者申请的封锁成功设置，它将用有关的信息覆盖掉原来的flock结构；如果调用者申请的封锁能够被成功设置，flock结构保持不变。如果F_GETLK调用无法获得文件预定封锁区的信息，就会返回“-1”表示操作失败。

如果F_GETLK调用成功了（即它返回了一个不是“-1”的值），调用者进程必须检查flock结构的内容，看它是否发生了变化。因为l_pid值被设置为封锁进程（如果有的话）的标识代码，所以查看这个数据域就可以方便地判断flock结构是否发生了变化。

2. F_SETLK命令

这个命令的作用是对fildes指定的文件的某个区域施行或者解除封锁状态。此时，flock结构中使用的值（与F_GETLK命令下的取值不一样了）见表7-3：

表 7-3

取 值	说 明
l_type	如果是只读或共享锁则取值为F_RDLCK；如果是独占或写操作锁则取值为F_WRLCK；如果是解除文件区域的封锁状态则取值为F_UNLCK
l_pid	不使用

如果封锁成功，fcntl将返回一个不是“-1”的值；失败时将返回“-1”。fcntl的这一调用总是会立刻返回。

3. F_SETLKW命令

它与刚才介绍的F_SETLK命令作用相同，但在无法进行封锁的情况下，它会一直等待直到它能够施行封锁为止。一旦这个调用开始了等待，就只有在能够施行封锁或者收到信号时才会返回。我们将在第10章介绍信号。

程序对某个文件拥有的各种锁将在相应的文件描述符被关闭时自动清除。各种封锁在程序运行结束时也会自动清除。

7.2.3 封锁状态下的读写操作

在对文件中的区域进行了封锁之后，访问文件中的数据时最好使用底层的read和write调用而不是高级的fread和fwrite函数。这是因为fread和fwrite会把被读写的数据缓冲保存在函数库里，

所以，读取一个文件前100个字节的fread调用在执行时会（事实上是肯定如此）读取多于100个的字节，多出来的字符将被缓冲保存到函数库里去。如果程序继续用fread读取接下来的100个字符，它实际是在读取缓冲保存在函数库里的数据，不会引起一个底层的read调用从文件里取出更多的数据。

我们用希望对同一个文件进行修改的两个程序来说明这为什么会是一个问题。假设文件是由200个字节组成的，字节的取值全部是“0”。第一个程序先开始运行，并且获得了文件前100个字节的写封锁。接下来，它使用fread读取了那100个字节。可正如我们在第3章里看到的那样，fread一次会读取多达BUFSIZ个字节，所以实际上它把整个文件都读进内存里了，但它只把前100个字节传递给程序。

这时候，第二个程序开始运行了。它获得了文件后100个字节的写封锁。这肯定是成功的，因为第一个程序只封锁了文件的前100个字节。第二个程序把字节100到199全部写成“2”，关闭文件，解除封锁并退出运行。这时候第一个程序又封锁了文件的后100个字符并调用fread读取它们。因为这些数据已经被缓冲保存了，所以程序实际看到的将是100个字节的“0”，而不是文件里实际存在的100个“2”。而使用read和write就不会出现这样的问题。

这样看来文件封锁好象挺复杂的，可事实上，这件事是说起来难，做起来倒容易些。我们用下面的lock3.c程序为例来说明文件封锁是如何工作的。

解释封锁概念需要两个程序，一个用来封锁，另一个用来测试。第一个程序的任务是进行封锁。

动手试试：用fcntl封锁一个文件

1) 程序开始是各种include语句和变量声明：

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

const char *test_file = "/tmp/test_lock";

int main()
{
    int file_desc;
    int byte_count;
    char *byte_to_write = "A";
    struct flock region_1;
    struct flock region_2;
    int res;
```

2) 打开一个文件描述符：

```
file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
if (!file_desc) {
    fprintf(stderr, "Unable to open %s for read/write\n", test_file);
    exit(EXIT_FAILURE);
}
```

3) 给文件里加上一些数据：

```
for(byte_count = 0; byte_count < 100; byte_count++) {
    (void)write(file_desc, byte_to_write, 1);
}
```

4) 把文件区域1设置为共享封锁状态，从字节10到30：

```
region_1.l_type = F_RDLCK;
region_1.l_whence = SEEK_SET;
region_1.l_start = 10;
region_1.l_len = 20;
```

5) 把文件区域2设置为独占封锁状态，从字节40到50：

```
region_2.l_type = F_WRLCK;
region_2.l_whence = SEEK_SET;
region_2.l_start = 40;
region_2.l_len = 10;
```

6) 现在，封锁文件……

```
printf("Process %d locking file\n", getpid());
res = fcntl(file_desc, F_SETLK, &region_1);
if (res == -1) fprintf(stderr, "Failed to lock region 1\n");
res = fcntl(file_desc, F_SETLK, &region_2);
if (res == -1) fprintf(stderr, "Failed to lock region 2\n");
```

7) ……再等一会儿。

```
    sleep(60);

    printf("Process %d closing file\n", getpid());
    close(file_desc);
    exit(EXIT_SUCCESS);
}
```

操作注释：

程序先创建了一个文件，以读写方式打开它，然后给文件填满数据。接着把它分为两个区域，第一个区域从字节10到30，施行共享（读）封锁；第二个区域从字节40到50，施行独占（写）封锁。然后程序调用fcntl封锁这两个区域，等待一分钟后关闭文件并退出。

程序开始运行并进入等待时的文件封锁状态如图7-1所示。

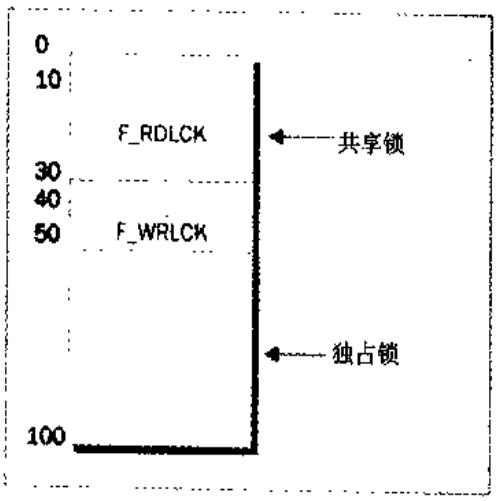


图 7-1

只有这一个程序当然没有什么用处。我们需要第二个程序lock4.c来对封锁进行测试。

动手试试：对文件的封锁状态进行测试

1) 和往常一样，程序开始是各种include语句和变量声明：

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

const char *test_file = "/tmp/test_lock";
#define SIZE_TO_TRY 5

void show_lock_info(struct flock *to_show);

int main()
{
    int file_desc;
    int res;
    struct flock region_to_test;
    int start_byte;
```

2) 打开一个文件描述符：

```
file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
if (!file_desc) {
    fprintf(stderr, "Unable to open %s for read/write", test_file);
    exit(EXIT_FAILURE);
}

for (start_byte = 0; start_byte < 99; start_byte += SIZE_TO_TRY) {
```

3) 设置准备测试的文件区域：

```
region_to_test.l_type = F_WRLCK;
region_to_test.l_whence = SEEK_SET;
region_to_test.l_start = start_byte;
region_to_test.l_len = SIZE_TO_TRY;
region_to_test.l_pid = -1;

printf("Testing F_WRLCK on region from %d to %d\n",
       start_byte, start_byte + SIZE_TO_TRY);
```

4) 现在，测试文件的封锁情况：

```
res = fcntl(file_desc, F_GETLK, &region_to_test);
if (res == -1) {
    fprintf(stderr, "F_GETLK failed\n");
    exit(EXIT_FAILURE);
}
if (region_to_test.l_pid != -1) {
    printf("Lock would fail. F_GETLK returned:\n");
    show_lock_info(&region_to_test);
}
else {
    printf("F_WRLCK - Lock would succeed\n");
}
```

5) 用共享锁再测试一次。再次设置准备测试的文件区域：

```
region_to_test.l_type = F_RDLCK;
region_to_test.l_whence = SEEK_SET;
region_to_test.l_start = start_byte;
region_to_test.l_len = SIZE_TO_TRY;
region_to_test.l_pid = -1;
```

```
printf("Testing F_RDLCK on region from %d to %d\n",
       start_byte, start_byte + SIZE_TO_TRY);
```

6) 再次测试文件的封锁情况：

```
        res = fcntl(file_desc, F_GETLK, &region_to_test);
        if (res == -1) {
            fprintf(stderr, "F_GETLK failed\n");
            exit(EXIT_FAILURE);
        }
        if (region_to_test.l_pid != -1) {
            printf("Lock would fail. F_GETLK returned:\n");
            show_lock_info(&region_to_test);
        }
        else {
            printf("F_RDLCK - Lock would succeed\n");
        }
    }
    close(file_desc);
    exit(EXIT_SUCCESS);
}

void show_lock_info(struct flock *to_show) {
    printf("\tl_type %d, ", to_show->l_type);
    printf("l_whence %d, ", to_show->l_whence);
    printf("l_start %d, ", (int)to_show->l_start);
    printf("l_len %d, ", (int)to_show->l_len);
    printf("l_pid %d\n", to_show->l_pid);
}
```

为了测试我们所做的封锁，需要先运行lock3程序，接着再运行lock4程序对被封锁文件进行测试。我们采取的办法是用下面的命令把lock3程序放到后台去运行：

```
$ lock3
$ process 1534 locking file
```

命令提示符又出现了，这是因为lock3是在后台运行的。接着，我们立刻用下面的命令运行lock4程序：

```
$ lock4
```

下面是我们得到的输出，为简洁起见，我们对它做了一定的省略：

```
Testing F_WRLOCK on region from 0 to 5
F_WRLCK - Lock would succeed
Testing F_RDLOCK on region from 0 to 5
F_RDLCK - Lock would succeed
...
Testing F_WRLOCK on region from 10 to 15
Lock would fail. F_GETLK returned:
l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 10 to 15
F_RDLCK - Lock would succeed
Testing F_WRLOCK on region from 15 to 20
Lock would fail. F_GETLK returned:
l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 15 to 20
F_RDLCK - Lock would succeed
...
Testing F_WRLOCK on region from 25 to 30
Lock would fail. F_GETLK returned:
l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 25 to 30
F_RDLCK - Lock would succeed
...
Testing F_WRLOCK on region from 40 to 45
Lock would fail. F_GETLK returned:
l_type 1, l_whence 0, l_start 40, l_len 10, l_pid 1534
Testing F_RDLOCK on region from 40 to 45
Lock would fail. F_GETLK returned:
```

```
l_type 1, l_whence 0, l_start 40, l_len 10, l_pid 1534
...
Testing F_RDLOCK on region from 95 to 100
F_RDLCK - Lock would succeed
```

操作注释：

lock4程序把数据文件每五个字节分为一组，为每组设置一个测试文件区域封锁状态的flock结构，再通过这些flock结构检查其对应区域是处于写封锁状态还是处于读封锁状态。返回信息将给出区域的字节、从字节0开始计算的偏移值、可能会使封锁申请失败的原因说明等。因为返回结构里的l_pid数据项包含着当前对文件施行封锁的程序的进程标识代码，所以我们先把它设置为“-1”（这是一个非法值），在fcntl调用返回时测试它是否被修改了。如果文件区域当时没有被封锁，l_pid将不会被改变。

为了读懂程序输出，我们需要查阅头文件fcntl.h（在Linux机器上的/usr/include/linux子目录里）。我们查到l_type参数的“1”值对应着定义F_WRLCK，而l_type参数的“0”值对应着定义F_RDLCK。因此，返回结构中l_type参数的“1”值会告诉我们不能施行封锁的原因是已经有一个写封锁了；返回结构中l_type参数的“0”值表示原因是已经有一个读封锁了。在数据文件没有被lock3程序封锁的区域上，共享封锁和独占封锁都可以成功。

从字节10到30，我们可以看到能够对它们施行共享封锁，因为lock3施行的现有封锁是共享而不是独占的。而在字节40到50之间的区域上，两种类型的封锁都将失败，因为lock3对这一区域施行的是一个独占（F_WRLCK）封锁。

7.2.4 文件封锁的竞争现象

见过如何测试一个文件上现有的封锁状态之后，我们再来看看当两个程序争夺文件同一区域的封锁权时会出现什么样的现象。我们先使用我们的lock3程序对数据文件施行封锁，然后用一个新的程序去尝试再次对它进行封锁。为了使这个程序示例更完整，我们添加了一些解除封锁的调用。

下面的lock5.c程序其作用不再是测试数据文件各部分的封锁状态了，它将试图对文件已经被封锁的区域再次施行封锁。

动手试试：文件封锁的竞争现象

1) 在各种include语句和变量声明之后，我们打开一个文件描述符：

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

const char *test_file = "/tmp/test_lock";

int main()
{
    int file_desc;
    struct flock region_to_lock;
    int res;
```

```
    file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
    if (!file_desc) {
        fprintf(stderr, "Unable to open %s for read/write\n", test_file);
        exit(EXIT_FAILURE);
    }
```

2) 程序的其余部分用来设置数据文件的不同区域，并试图对它们施行不同的封锁：

```
    region_to_lock.l_type = F_RDLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 10;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_RDLCK, region %d to %d\n", getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start +
region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
        printf("Process %d - obtained lock region\n", getpid());
    }

    region_to_lock.l_type = F_UNLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 10;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_UNLCK, region %d to %d\n", getpid(),
                                (int)region_to_lock.l_start,
(int)(region_to_lock.l_start +
        region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to unlock region\n", getpid());
    } else {
        printf("Process %d - unlocked region\n", getpid());
    }

    region_to_lock.l_type = F_UNLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 0;
    region_to_lock.l_len = 50;
    printf("Process %d, trying F_UNLCK, region %d to %d\n", getpid(),
                                (int)region_to_lock.l_start,
(int)(region_to_lock.l_start +
        region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to unlock region\n", getpid());
    } else {
        printf("Process %d - unlocked region\n", getpid());
    }

    region_to_lock.l_type = F_WRLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 16;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_WRLCK, region %d to %d\n", getpid(),
                                (int)region_to_lock.l_start,
(int)(region_to_lock.l_start +
        region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
        printf("Process %d - obtained lock on region\n", getpid());
    }

    region_to_lock.l_type = F_RDLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 40;
```

```
    region_to_lock.l_len = 10;
    printf("Process %d, trying F_RDLCK, region %d to %d\n", getpid(),
                                (int)region_to_lock.l_start,
(int)(region_to_lock.l_start +
        region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
        printf("Process %d - obtained lock on region\n", getpid());
    }

    region_to_lock.l_type = F_WRLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 16;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_WRLCK with wait, region %d to %d\n", getpid(),
                                (int)region_to_lock.l_start,
(int)(region_to_lock.l_start +
        region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLKW, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
        printf("Process %d - obtained lock on region\n", getpid());
    }

    printf("Process %d ending\n", getpid());
    close(file_desc);
    exit(EXIT_SUCCESS);
}
```

我们先在后台运行lock3程序，然后立刻运行这个新程序。下面是我们得到的输出：

```
Process 227 locking file
Process 228, trying F_RDLCK, region 10 to 15
Process 228 - obtained lock on region
Process 228, trying F_UNLCK, region 10 to 15
Process 228 - unlocked region
Process 228, trying F_UNLCK, region 0 to 50
Process 228 - unlocked region
Process 228, trying F_WRLCK, region 16 to 21
Process 228 - failed to lock on region
Process 228, trying F_RDLCK, region 40 to 50
Process 228 - failed to lock on region
Process 228, trying F_WRLCK with wait, region 16 to 21
Process 227 closing file
Process 228 - obtained lock on region
Process 228 ending
```

操作注释：

首先，这个程序试图用一个共享锁来封锁从字节10到15的一个区域。这个区域已经被一个共享锁封锁住了，但共享锁是允许同时使用的，因此封锁成功。

随后，程序解除了它自己对该区域的共享封锁，这也成功了。接下来，程序试图解除数据文件前50个字节上的封锁——虽然它本身没有对这一区域设置任何封锁。这也成功了，这是因为解锁申请的最终结果是使这个程序不再对这前50个字节拥有任何封锁——这当然不需要这个程序在事先设置什么封锁。

接着，程序试图用一个独占锁来封锁从字节16到21的一个区域。这个区域已经被一个共享锁封锁住了，因此这一次新的封锁失败了，因为无法在此区域上创建一个独占锁。

在此之后，程序尝试给从字节40到50的区域加上一个共享锁。这个区域已经被一个独占锁封锁住了，所以封锁行动再次失败。

最后，程序再次尝试给从字节16到21的区域加上一个独占锁，但这一次它使用了F_SETLKW命令等到自己能够获得一个封锁为止。程序输出在此出现了一个很长的停顿，直到对这一区域施行封锁的lock3程序因关闭文件而释放了它所拥有的全部封锁才有了转机。lock5程序开始继续执行，成功地封锁了该区域，随后它也退出了运行。

7.2.5 其他封锁命令

还有另外一种封锁文件的方法：lockf函数。它也通过文件描述符进行操作。下面是它的定义：

```
# include <unistd.h>
int lockf(int fildes, int function, off_t size_to_lock );
```

它可以选用以下funcion参数值：

- F_ULOCK　　解除封锁。
- F_LOCK　　施行独占封锁。
- F_TLOCK　　测试并施行独占封锁。
- F_TEST　　对其他进程施行的封锁进行测试。

size_to_lock参数是准备对之进行操作的字节个数，从文件的当前偏移值开始计算。

lockf的接口比fcntl的接口要简单，这主要是因为它在功能性和灵活性方面要差一些。使用这个函数的时候必须先找到准备封锁的文件区域的起始位置，然后按准备封锁的字节个数进行封锁。

与文件封锁的fcntl方法一样，lockf提供的各种封锁都只是建议性的；它们并不能真正阻止对文件的读写。对封锁状态进行检查是程序的责任。混用fcntl封锁和lockf封锁的后果没有明确的说法，所以你必须决定自己想要使用的是哪一种类型的封锁并坚持下去。

7.2.6 死锁现象

如果不说说死锁现象的危险对封锁的讨论就不能算是完整。假设有两个程序想要修改同一个数据文件。它们都需要在同一时间对字节1和字节2进行修改。程序A选择先修改字节2，再修改字节1；而程序B则决定先修改字节1，再修改字节2。

两个程序同时开始运行。程序A封锁住了字节2，程序B封锁住了字节1。接着，程序A尝试封锁字节1，但因为它已经被程序B封锁住了，所以程序A开始等待。而程序B也在尝试封锁字节2，但因为它已经被程序A封锁住了，所以程序B也开始等待。

在这种情况下，两个程序谁都无法继续前进，这一现象就叫做“死锁”或“致命拥抱”。大多数商业化的数据库软件能够检测到死锁现象并解开它们；但UNIX操作系统的内核不会这样做。要想从这团乱麻里找出头绪，必须有来自外界的斡旋，比如我们可以强行终止其中的一个程序。

程序员必须对这种局面提高警惕。当你有多个程序在等待封锁的时候，就必须非常认真地考虑这个问题，判断有无出现死锁现象的可能性。在刚才举的例子里，死锁现象是很容易避免的：两个程序应该以同样的顺序对它们准备修改的字节施行封锁，或者对一个更大的文件区域施行封锁。

我们没有足够的篇幅在这里对共发程序所面临的困难进行分析。如果读者有兴趣做进一步阅读，可以考虑购买一本“Principles of Concurrent and Distributed Programming”（《共发和分布式程序设计原理》），它的作者是M. Ben-Ari，由Prentice Hall出版社出版，国际书号是ISBN 013711821X。

7.3 数据库

我们已经学习了如何用文件来存储数据，那为什么还要使用数据库呢？其实原因很简单，在某些情况下，数据库功能可以为我们提供一个更好的解决问题的办法。与单纯用来保存数据的文件相比，数据库有两方面优点。首先，允许保存长度不固定的记录数据，这对非结构化的平面文件来说实现起来就有困难。第二，数据库通过一个索引来存储和检索记录数据。索引不必非得是记录号等简单的东西——这在平面文件里也很容易实现，它可以是任意的字符串。

7.3.1 dbm数据库

符合X/Open技术规范的UNIX版本自备了一个数据库。但这个数据库并不符合ANSI标准的SQL技术规范，只能说是一些用来存储和检索记录数据的例程而已。

有的Linux发行版本自带一个名为PostgreSQL的SQL数据库，它确实支持SQL规范，但要讨论它可就超出这本书的范围了。

dbm数据库适合存储相对比较静态的索引化数据。有些数据库人士认为dbm根本就算不上是个数据库，顶多算是个索引化的文件存储系统。但X/Open技术规范的确把dbm看做是一个数据库，所以我们在这本书里也这么称呼它。另外，不少Linux发行版本自带的数据库产品是GNU的gdbm。

dbm数据库使我们能够通过索引把长度可变的数据结构保存起来；在对这些结构进行检索的时候，既可以使用索引，也可以简单地顺序扫描整个数据库。dbm数据库最适合保存那些经常被访问但很少需要修改的数据，它创建数据项时会非常慢，但检索数据时却会很快。

讲到这我们遇上个小问题——不同Linux发行版本自带的dbm库版本是多种多样的。它们有的使用着BSD授权版本，我们可以在FTP站点ftp://ftp.cs.berkeley.edu/ucb/4bsd/和Web站点http://www.openbsd.org上找到它。其他一些使用的是GNU版本，这个版本可以在Web站点http://www.gnu.org上找到。使情况更加复杂的是：GNU系列又分为两种安装方式，一种是与X/Open的dbm技术规范不完全兼容的“普通安装方式”，另一种是与X/Open的dbm技术规范兼容的“兼容安装方式”。在我们写作这本书的时候，Sleepycat软件公司（它的网址是http://www.sleepycat.com）推出了一个名为“The Berkley Database”的开放源代码产品，这个产品也支持Dbm/Ndbm一脉相承的程序设计接口。

这一章内容将假设读者机器里安装的是一个与X/Open技术规范兼容的版本。如果读者在编译本章中的程序示例时遇到了麻烦，原因一般有两种：如果编译器提示找不到DBM类型的定义，

就可能是缺少ndbm.h头文件；如果在程序编译的链接阶段出现了问题，我们建议读者去安装一个GNU dbm库的升级版本，具体做法如下：

先建立一个临时子目录。然后到http://www.gnu.org站点上下载一份最新版的gdbm库。这个库被放在一个文件里，文件名一般采用“gdbm_?_?_tar.gz”的形式。把文件下载到你的临时子目录里，用“tar zxvf <filename>”命令展开它。先读读README文件，它会告诉你如何对它进行编译和安装。常见的做法是先执行一条“./configure”命令检查你的系统配置情况，再用“make”命令对程序进行编译。最后，用“make install”和“make install -compat”命令把基本文件和附带的兼容文件安装好。完成这些安装步骤可能需要你具备根用户的权限。最好先用“-n”选项做一次模拟性质的安装操作，看看它都会干些什么，模拟安装使用的命令是“make -n install”。

完成上面这些操作之后，你就应该拥有一个与X/Open技术规范兼容的ndbm版本了，它一般被安装在你系统的/usr/local子目录下。你编译器的默认配置可能不会去搜索这些地方，如果真是这样，就还得给gcc命令加上一个搜索头文件用的“-I/usr/local/include”选项和一个搜索库文件用的“-L/usr/local/lib”选项。

7.3.2 dbm例程

dbm数据库和我们在上一章见到的curses一样，都是由两个部分组成，一个是程序源代码文件里用的头文件，另一个是链接程序目标代码用的库文件。库文件就简单地被称为dbm，给编译命令行添上一个“-ldbm”选项（偶尔要使用“-lgdbm”选项）就可以用它完成链接。前缀字母“n”表示这是一个“新”dbm库，因为曾经出现过好多种不同版本的dbm库，加上这个字母就可以把它和老版本dbm库区分开。

dbm数据库的基本概念

在向大家介绍dbm数据库的各个函数之前，我们先来看看它到底能干些什么。这可是个很关键的问题，能够帮助我们更好地理解dbm函数的使用方法。

dbm数据库的基本元素包括两个数据块，一块是想要保存起来的数据，另一块是对其进行检索时用做关键字的数据。对每个dbm数据库而言，保存在其中的每一个数据块都必须有一个独一无二的关键字。对关键字和数据本身倒没有什么限制，对使用超长数据或超长关键字的情况也没有定义什么错误。技术规范里倒是允许在具体实现时把关键字/数据对的最大长度限制在1024个字节，但这个限制通常并没有什么意义，因为具体实现出来的东西往往比技术规范的要求更灵活。关键字的取值被用做检索存储数据的索引，就像图7-2里的标签一样。

为了对这些数据块进行操作，人们在头文件ndbm.h里定义了一个新的类型，即datum类型。这个类型的具体内容会随版本的不同而不同，但它至少会包含如下所示的成员：

```
void *dptr
size_t dsize
```

datum是一个用typedef语句定义的类型。在ndbm.h文件里还定义了一个DBM类型，这是一个用来访问数据库的结构，其作用与用来访问文件的FILE很相似。DBM类型的内部结构依赖于具体的版本，不允许程序直接访问。

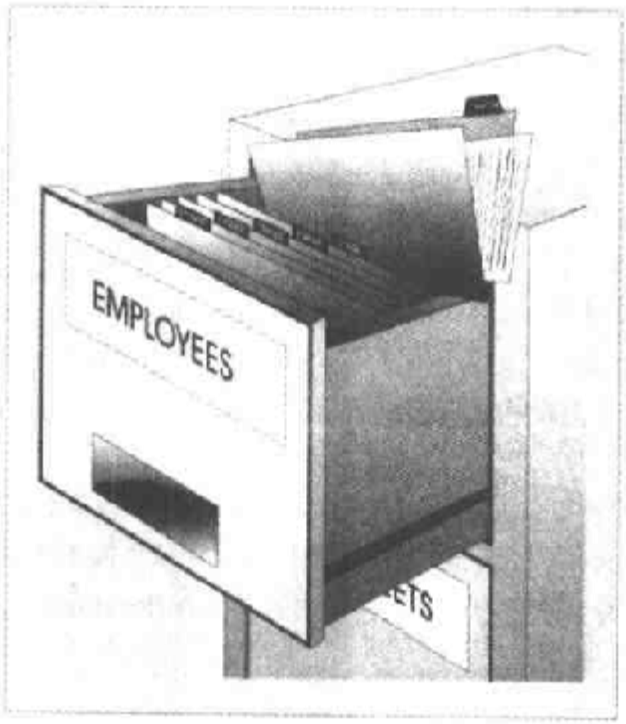


图 7-2

在使用dbm数据库的时候，要想引用一个数据块，必须先声明一个datum类型的结构，让参数dptr指向数据库记录的起始点，把数据库记录的长度放在dsize参数里。无论是保存在数据库里的记录数据，还是用来访问数据库记录的索引数据，都必须通过一个datum类型来引用。

我们可以把DBM类型看做是FILE类型的一个翻版。在打开一个dbm数据库的时候，我们将实际创建出两个物理文件来，两个文件的扩展名分别是“.pag”和“.dir”。但操作的返回值只有一个，即一个DBM类型的指针，对那两个文件的访问就要靠这个指针来完成。永远不要直接读写这两个文件，对它们的访问一定要通过dbm函数来进行。

> gdbm库把这两个文件合并到了一起，打开数据库时也只会创建出一个文件来。这一点请使用gdbm库的读者多加注意。

如果读者熟悉SQL数据库，就会注意到dbm数据库没有与之关联的数据表或数据栏结构。这是因为dbm不仅对数据库记录中的数据项是否是固定长度没有要求，对数据的内部构造也没有要求。dbm工作在非结构化二进制数据块的基础上。

7.3.3 dbm数据库的访问函数

介绍完dbm的工作基础之后，下面开始对有关函数进行说明。常用的dbm函数主要有以下几种：

```
#include <ndbm.h>

DBM *dbm_open(const char *filename, int file_open_flags, mode_t file_mode);
int dbm_store(DBM *database_descriptor, datum key, datum content, int store_mode);
datum dbm_fetch(DBM *database_descriptor, datum key);
void dbm_close(DBM *database_descriptor);
```

1. dbm_open函数

这个函数的作用是打开一个现有的数据库，也可以用来创建新数据库。filename参数是一个基本文件名，不要加上“.pag”或“.dir”扩展名。

其余参数与我们在第5章里学过的open函数的第二个和第三个参数是一样的。我们可以使用同样的“# define”定义。数据库的读、写或读/写权限由第二个参数控制。创建新数据库的时候要把这些标志与O_CREAT用二进制OR操作“或”在一起，O_CREAT标志表示将创建一个新文件。第三个参数的作用是设定被创建文件的初始权限。

dbm_open函数返回的是一个DBM类型的指针，数据库的后续操作将通过这个指针来完成。如果失败，dbm_open函数将返回“(DBM *) 0”。

2. dbm_store函数

我们用这个函数把记录数据存放到数据库里去。我们在前面曾经说过，任何被保存到数据库里去的记录数据都必须有一个独一无二的索引。而要想定义准备保存到数据库里的记录数据和用来引用记录数据的索引数据，就必须设置两个datum类型的结构：一个用于索引数据，另一个用于记录数据。store_mode参数控制用一个已经存在的关键字再次向数据库里存放数据的时候会发生什么样的事情：如果它被设置为“dbm_insert”，存放操作将失败，dbm_store返回“1”；如果它被设置为“dbm_replace”，新数据就会覆盖掉现有数据，存放成功，dbm_store返回“0”。如果出现其他错误，dbm_store将返回一个负数值。

3. dbm_fetch函数

这个函数的作用是在数据库里检索数据。它的参数包括一个从dbm_open调用返回的dbm指针和一个datum类型的结构——这个结构必须指向一个关键字。它返回的是一个datum类型的结构。如果在数据库里找到了与关键字对应的记录数据，被返回的datum结构里的dptr和dsize值将指向那条被找到的记录数据；如果没有找到与关键字对应的记录数据，dptr将被设置为“null”。

dbm_fetch返回的datum结构里只有一个指向记录数据的指针，记录数据本身依然放在dbm库内部的一个本地存储区里，在继续调用其他dbm函数之前，必须先把它拷贝到程序变量里才行。

4. dbm_close函数

这个函数的作用是关闭用dbm_open打开的数据库，它的参数是一个dbm指针，而这个指针必须是以前的某个dbm_open调用的返回值。

介绍了这么多dbm数据库的基本函数之后，我们来编写我们的第一个dbm程序，程序名是dbm1.c。在这个程序里，我们将使用一个名为test_data的结构。

动手试试：一个简单的dbm数据库

1) 程序开始部分是头文件、常数定义、main函数和对test_data结构的定义：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <ndbm.h>
#include <string.h>
```

```
#define TEST_DB_FILE "/tmp/dbm1_test"
#define ITEMS_USED 3

struct test_data {
    char misc_chars[15];
    int  any_integer;
    char more_chars[21];
};

int main()
{
```

2) 我们在main函数里安排了items_to_store和items_received这两个结构，还设置了索引数据和记录数据这两个datum类型的结构：

```
struct test_data items_to_store[ITEMS_USED];
struct test_data item_retrieved;

char key_to_use[20];
int i, result;

datum key_datum;
datum data_datum;

DBM *dbm_ptr;
```

3) dbm_ptr被定义为一个指向dbm类型的结构的指针，我们用它以读写方式打开测试用的数据库，如果文件不存在，就创建它：

```
dbm_ptr = dbm_open(TEST_DB_FILE, O_RDWR | O_CREAT, 0666);
if (!dbm_ptr) {
    fprintf(stderr, "Failed to open database\n");
    exit(EXIT_FAILURE);
}
```

4) 现在往items_to_store结构里添加一些数据：

memset(items_to_store, '\0', sizeof(items_to_store));

```
memset(items_to_store, '\0', sizeof(items_to_store));
strcpy(items_to_store[0].misc_chars, "First!");
items_to_store[0].any_integer = 47;
strcpy(items_to_store[0].more_chars, "foo");

strcpy(items_to_store[1].misc_chars, "bar");
items_to_store[1].any_integer = 13;
strcpy(items_to_store[1].more_chars, "unlucky?");

strcpy(items_to_store[2].misc_chars, "Third");
items_to_store[2].any_integer = 3;
strcpy(items_to_store[2].more_chars, "baz");
```

5) 我们必须为每个数据项建立一个供今后引用的关键字。我们把关键字设置为字符串的头一个字符加那个整数。这个关键字将由key_datum标识，而data_datum则指的是数据项items_to_store。然后，我们把数据保存到数据库里。

```
for (i = 0; i < ITEMS_USED; i++) {
    sprintf(key_to_use, "%c%c%d",
            items_to_store[i].misc_chars[0],
            items_to_store[i].more_chars[0],
            items_to_store[i].any_integer);
```

```
    key_datum.dptr = (void *)key_to_use;
    key_datum.dsize = strlen(key_to_use);
    data_datum.dptr = (void *)&items_to_store[i];
    data_datum.dsize = sizeof(struct test_data);

    result = dbm_store(dbm_ptr, key_datum, data_datum, DBM_REPLACE);
    if (result != 0) {
        fprintf(stderr, "dbm_store failed on key %s\n", key_to_use);
        exit(2);
    }
}
```

6) 现在看看能不能检索到这个新数据，最后，关闭数据库文件。

```
    sprintf(key_to_use, "bu%d", 13);
    key_datum.dptr = key_to_use;
    key_datum.dsize = strlen(key_to_use);

    data_datum = dbm_fetch(dbm_ptr, key_datum);
    if (data_datum.dptr) {
        printf("Data retrieved\n");
        memcpy(&item_retrieved, data_datum.dptr, data_datum.dsize);
        printf("Retrieved item - %s %d %s\n",
               item_retrieved.misc_chars,
               item_retrieved.any_integer,
               item_retrieved.more_chars);
    }
    else {
        printf("No data found for key %s\n", key_to_use);
    }
    dbm_close(dbm_ptr);
    exit(EXIT_SUCCESS);
}
```

当我们编译并运行这个程序的时候，它会给出下面这样简单的输出：

```
$ gcc -o dbm1 dbm1.c -ldbm
$ dbm1
Data retrieved
Retrieved item - bar 13 unlucky?
```

如果编译失败了，视具体情况可能还要再装上GNU gdbm库的兼容文件才行，或者还需要在编译时指定其他的子目录，如下所示：

```
$ gcc -I/usr/local/include -L/usr/local/lib -o dbm1 dbm1.c -ldbm
```

如果还不行，再试试用“-lgdbm”替换命令行上的“-ldbm”参数，即：

```
$ gcc -I/usr/local/include -L/usr/local/lib -o dbm1 dbm1.c -lgdbm
```

操作注释：

首先，打开数据库，如果不存在就创建它。接着填写了三个做为测试数据的item_to_store数据项。我们为这三条记录数据各创建一个索引关键字。为简单起见，我们就用两个字符串各自的头一个字母和记录中的整数来构成关键字。

接下来我们设定了两个datum结构，一个对应着关键字，另一个对应着记录数据。把三条记录保存到数据库里，然后构造一个新关键字并用一个datum结构指向它。接着，我们用这个关键字在数据库中检索记录数据。我们通过检查返回的datum结构中的dptr是否为“null”来确定我们

的检索操作是否成功。如果它不是“null”，我们就把检索到的记录数据（它可能已经被放在dbm库内部的本地存储区里了）拷贝到程序的某个结构变量里。注意：要把记录数据的长度设置为dbm_fetch返回的值；如果我们正在对长度不固定的记录数据进行处理但又没有这样做，操作结果将难以预料，想要拷贝的数据可能根本就不存在。最后，我们把检索到的记录数据输出在屏幕上，表示我们的检索操作成功了。

7.3.4 其他dbm函数

除了这些最常用的dbm函数以外，在dbm数据库里还经常用到一些其他的函数，包括下面这些：

```
int dbm_delete(DBM *database_descriptor, datum key);
int dbm_error(DBM *database_descriptor);
int dbm_clearerr(DBM *database_descriptor);
datum dbm_firstkey(DBM *database_descriptor);
datum dbm_nextkey(DBM *database_descriptor);
```

1. dbm_delete函数

dbm_delete函数的作用是从数据库里删除记录数据。其datum类型的关键字参数key用途和dbm_fetch函数中的一样，但目的是删除记录而不是检索它。它在成功时返回“0”。

2. dbm_error函数

dbm_error函数对数据库进行简单的测试，检查其中是否有错误出现。没有错误时返回“0”。

3. dbm_clearerr函数

dbm_clearerr函数的作用是清除数据库里所有已经被置位的错误条件标志。

4. dbm_firstkey和dbm_nextkey函数

这两个函数一般成对使用，它们的作用是根据关键字对数据库中全部记录进行扫描。完成这一操作需要使用下面这样的循环结构：

```
DBM *db_ptr;
datum key;

for(key = dbm_firstkey(db_ptr); key.dptr; key = dbm_nextkey(db_ptr));
```

我们用这里介绍的新函数对dbm1.c做些改进。请看dbm2.c的程序清单。

动手试试：检索和删除

1) 拷贝一份dbm1.c，打开它进行编辑。别忘了改“# define TEST_DB_FILE”语句。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <ndbm.h>
#include <string.h>

#define TEST_DB_FILE "/tmp/dbm2_test"
#define ITEMS_USED 3
```

2) 其他改动都集中在检索部分。

```
    /* now try and delete some data */
```

```
sprintf(key_to_use, "bu%d", 13);
key_datum.dptr = key_to_use;
key_datum.dsize = strlen(key_to_use);

if (dbm_delete(dbm_ptr, key_datum) == 0) {
    printf("Data with key %s deleted\n", key_to_use);
}
else {
    printf("Nothing deleted for key %s\n", key_to_use);
}

for (key_datum = dbm_firstkey(dbm_ptr);
       key_datum.dptr;
       key_datum = dbm_nextkey(dbm_ptr)) {
    data_datum = dbm_fetch(dbm_ptr, key_datum);
    if (data_datum.dptr) {
        printf("Data retrieved\n");
        memcpy(&item_retrieved, data_datum.dptr, data_datum.dsize);
        printf("Retrieved item - %s %d %s\n",
               item_retrieved.misc_chars,
               item_retrieved.any_integer,
               item_retrieved.more_chars);
    }
    else {
        printf("No data found for key %s\n", key_to_use);
    }
}
```

这个程序的输出如下所示：

```
$ dbm2
Data with key bu13 deleted
Data retrieved
Retrieved item - Third 3 baz
Data retrieved
Retrieved item - First! 47 foo
```

操作注释：

这个程序的开始部分和前面的例子是完全一样的，都是把一些数据存放到数据库里去。在这之后，我们设置了一个与第二条记录数据匹配的关键字，并把它从数据库里删掉了。

接下来，程序使用dbm_firstkey和dbm_nextkey函数依次访问数据库中的每一个关键字，检索出与之对应的记录数据来。注意，记录数据不是按顺序检索出来的。关键字不具备排序方面的功用，它只是扫描全体记录数据的一个手段而已。

7.4 CD唱盘管理软件

学习了环境和数据管理之后，现在是我们改进这个应用软件的时候了。dbm数据库看起来很适合存储我们的CD唱盘资料，所以我们以它做为新编程序的基础。因为很多东西都需要重写，所以现在正是检讨我们的设计思路并予以改进的好时机。

我们曾经在数据文件里使用逗号来分隔各个变量，这样做方便了shell脚本程序的编写，但无形中也限制了自己。许多CD唱盘在标题和曲目里都包含有逗号，既然准备使用dbm数据库，再使用逗号来分隔数据域就没有实际的意义了。我们决定改变设计思路，彻底放弃这一做法。

我们当初的一个决定是把CD唱盘资料分为“标题”和“曲目”两个部分并把它们分别保存到两个文件里，这一决定看来还不错，我们将继续沿用同样的逻辑安排。

以前的程序多少都存在着这样一个问题，即把数据访问部分和用户界面部分混在了一起，这与程序全写在一个文件里不无关系。在这个新的实现里，我们将用一个头文件来描述记录数据和访问记录数据的各个函数，把用户界面和数据处理分别放到两个文件里去。

我们当然可以继续使用由curses实现的用户界面，但我们决定重新回到基于数据行的简单界面。这会使应用程序的用户界面部分既短小又简单，让我们把精力集中到其他方面去。

虽然我们无法通过dbm代码使用SQL，但可以通过SQL术语把我们的新数据库表达得更规范。读者不必担心自己不熟悉SQL，大家可以在下面找到数据库的格式定义。我们的数据表可以用下面的代码来描述：

```
CREATE TABLE cdc_entry (
    catalog CHAR(30) PRIMARY KEY REFERENCES cdt_entry(catalog),
    title     CHAR(70),
    type      CHAR(30),
    artist    CHAR(70)
);

CREATE TABLE cdt_entry (
    catalog CHAR(30) REFERENCES cdc_entry(catalog),
    track_no  INTEGER CHECK (track_no > 0),
    track_txt CHAR(70),
    PRIMARY KEY(catalog, track_no)
);
```

通过这个非常简明的描述，我们知道了数据域的名字和长度。就cdc_entry数据表而言，它告诉我们每个记录项都有一个独一无二catalog栏；就cdt_entry数据表而言，它告诉我们曲目编号不能为零，并且，catalog和track_no两栏的组合是独一无二的。

使用dbm数据库的CD唱盘管理软件

下面开始重新编写这个软件了，我们将用数据库来保存我们需要的资料，全部程序分别保存在三个文件里，它们是cd_data.h、app_ui.c和cd_access.c。

我们还把用户界面重新编写为一个命令行程序。在本书后面的内容里，我们将探讨如何利用各种客户/服务器机制来实现我们的软件，使它最终成为一个能够通过WWW浏览器跨网络访问的软件。到那时，我们还会用到这里实现的数据库接口和一部分用户界面。把用户界面转换为一个简单的行驱动界面将节约我们在这一方面的注意力，使我们能够集中精力搞好软件里重要的部分。

数据库头文件cd_data.h和文件cd_access.c里定义的函数将在后续章节里多次出现，请大家注意它们的再使用情况。我们从头文件开始介绍，它对我们数据的结构和将要用来访问它的函数例程进行了定义。

动手试试：头文件cd_data.h

1) 这是CD唱盘管理软件的数据结构定义。我们的数据库由两个数据表构成，它们的结构和尺寸都是在这个文件里定义的。我们先定义了几个数据域长度方面的常数，然后又定义了两个结构：一个用来记录唱盘的标题资料，另一个用来记录唱盘的曲目资料：

```
/* The catalog table */
#define CAT_CAT_LEN         30
#define CAT_TITLE_LEN       70
#define CAT_TYPE_LEN        30
#define CAT_ARTIST_LEN      70

typedef struct {
    char catalog[CAT_CAT_LEN + 1];
    char title[CAT_TITLE_LEN + 1];
    char type[CAT_TYPE_LEN + 1];
    char artist[CAT_ARTIST_LEN + 1];
} cdc_entry;

/* The tracks table, one entry per track */
#define TRACK_CAT_LEN       CAT_CAT_LEN
#define TRACK_TTEXT_LEN     70

typedef struct {
    char catalog[TRACK_CAT_LEN + 1];
    int  track_no;
    char track_txt[TRACK_TTEXT_LEN + 1];
} cdt_entry;
```

2) 定义好数据结构之后，我们再定义一些用来对数据进行访问的函数例程。以“cdc_”开始的函数负责处理唱盘本身的资料，以“cdt_”开始的函数负责处理唱盘的曲目资料。

> 注意：有几个函数的返回值是一个数据结构。我们可以用把结构的内容强行设置为空的办法来表明这些函数没有执行成功的情况。

```
/* Initialization and termination functions */
int database_initialize(const int new_database);
void database_close(void);

/* two for simple data retrieval */
cdc_entry get_cdc_entry(const char *cd_catalog_ptr);
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no);

/* two for data addition */
int add_cdc_entry(const cdc_entry entry_to_add);
int add_cdt_entry(const cdt_entry entry_to_add);

/* two for data deletion */
int del_cdc_entry(const char *cd_catalog_ptr);
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no);

/* one search function */
cdc_entry search_cdc_entry(const char *cd_catalog_ptr, int *first_call_ptr);
```

> 动手试试：app_ui.c程序清单

1) 现在我们进入软件的用户界面部分。这部分程序相对来说比较简单，我们将通过它来调用我们的数据库函数，数据库函数实现在另一个文件里。我们象往常一样从头文件开始。

```
#define _XOPEN_SOURCE

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

#include "cd_data.h"
```

```
#define TMP_STRING_LEN 125 /* this number must be larger than the biggest
                              single string in any database structure */
```

2) 我们用typedef把菜单选项定义为一个类型。这种做法要比使用“# define”语句定义的常数效果要好，因为它允许编译器对菜单选项变量进行类型检查。

```
typedef enum {
    mo_invalid,
    mo_add_cat,
    mo_add_tracks,
    mo_del_cat,
    mo_find_cat,
    mo_list_cat_tracks,
    mo_del_tracks,
    mo_count_entries,
    mo_exit
} menu_options;
```

3) 下面是各种局部函数的预定义。需要提醒大家注意的是：对数据库实际进行访问的函数是通过“# include <cd_data.h>”语句包括进来的。

```
static int command_mode(int argc, char *argv[]);
static void announce(void);
static menu_options show_menu(const cdc_entry *current_cdc);
static int get_confirm(const char *question);
static int enter_new_cat_entry(cdc_entry *entry_to_update);
static void enter_new_track_entries(const cdc_entry *entry_to_add_to);
static void del_cat_entry(const cdc_entry *entry_to_delete);
static void del_track_entries(const cdc_entry *entry_to_delete);
static cdc_entry find_cat(void);
static void list_tracks(const cdc_entry *entry_to_use);
static void count_all_entries(void);
static void display_cdc(const cdc_entry *cdc_to_show);
static void display_cdt(const cdt_entry *cdt_to_show);
static void strip_return(char *string_to_strip);
```

4) 现在到达main函数。它先对current_cdc_entry结构进行初始化，我们用这个结构来保存当前中选CD唱盘的标题资料。接下来，我们分析了命令行，宣布正在运行的是哪个程序。然后，对数据库进行初始化。

```
void main(int argc, char *argv[])
{
    menu_options current_option;
    cdc_entry current_cdc_entry;
    int command_result;

    memset(&current_cdc_entry, '\0', sizeof(current_cdc_entry));

    if (argc > 1) {
        command_result = command_mode(argc, argv);
        exit(command_result);
    }

    announce();

    if (!database_initialize(0)) {
        fprintf(stderr, "Sorry, unable to initialize database\n");
        fprintf(stderr, "To create a new database use %s -i\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

5) 现在，处理用户输入的准备工作已经就绪。我们在一个循环里等待用户对菜单做出选择

并进行相应的处理，直到用户选择了退出操作的选项为止。请注意，我们把current_cdc_entry结构做为一个参数传递给了show_menu函数。这是为了让菜单选项能够根据用户是否选取了一个标题记录项做相应的变化。

```
while(current_option != mo_exit) {
        current_option = show_menu(&current_cdc_entry);

        switch(current_option) {
            case mo_add_cat:
                if (enter_new_cat_entry(&current_cdc_entry)) {
                    if (!add_cdc_entry(current_cdc_entry)) {
                        fprintf(stderr, "Failed to add new entry\n");
                        memset(&current_cdc_entry, '\0',
                                sizeof(current_cdc_entry));
                    }
                }
                break;
            case mo_add_tracks:
                enter_new_track_entries(&current_cdc_entry);
                break;
            case mo_del_cat:
                del_cat_entry(&current_cdc_entry);
                break;
            case mo_find_cat:
                current_cdc_entry = find_cat();
                break;
            case mo_list_cat_tracks:
                list_tracks(&current_cdc_entry);
                break;
            case mo_del_tracks:
                del_track_entries(&current_cdc_entry);
                break;
            case mo_count_entries:
                count_all_entries();
                break;
            case mo_exit:
                break;
            case mo_invalid:
                break;
        default:
            break;
    } /* switch */
} /* while */
```

6) 当主循环退出的时候，我们关闭数据库文件并返回到环境。欢迎辞是用announce函数输出的。

```
    database_close();
    exit(EXIT_SUCCESS);
} /* main */

static void announce(void)
{
    printf("\n\nWelcome to the demonstration CD catalog database \
            program\n");
}
```

7) 下面是show_menu函数的程序清单。它先检查用户是否已经选中一个标题记录项，用户通过输入标题记录项第一个字母的办法来选取它。如果已经选中一个标题记录项，用户将看到更多的菜单选项。

> 注意：现在要用数字来选择菜单项了。我们在前两个例子里用的都是首字母。

```
static menu_options show_menu(const cdc_entry *cdc_selected)
{
    char tmp_str[TMP_STRING_LEN + 1];
    menu_options option_chosen = mo_invalid;

    while (option_chosen == mo_invalid) {
        if (cdc_selected->catalog[0]) {
            printf("\n\nCurrent entry: ");
            printf("%s, %s, %s, %s\n", cdc_selected->catalog,
                   cdc_selected->title,
                   cdc_selected->type,
                   cdc_selected->artist);

            printf("\n");
            printf("1 - add new CD\n");
            printf("2 - search for a CD\n");
            printf("3 - count the CDs and tracks in the database\n");
            printf("4 - re-enter tracks for current CD\n");
            printf("5 - delete this CD, and all its tracks\n");
            printf("6 - list tracks for this CD\n");
            printf("q - quit\n");
            printf("\nOption: ");
            fgets(tmp_str, TMP_STRING_LEN, stdin);
            switch(tmp_str[0]) {
                case '1': option_chosen = mo_add_cat; break;
                case '2': option_chosen = mo_find_cat; break;
                case '3': option_chosen = mo_count_entries; break;
                case '4': option_chosen = mo_add_tracks; break;
                case '5': option_chosen = mo_del_cat; break;
                case '6': option_chosen = mo_list_cat_tracks; break;
                case 'q': option_chosen = mo_exit; break;
            }

        }
        else {
            printf("\n\n");
            printf("1 - add new CD\n");
            printf("2 - search for a CD\n");
            printf("3 - count the CDs and tracks in the database\n");
            printf("q - quit\n");
            printf("\nOption: ");
            fgets(tmp_str, TMP_STRING_LEN, stdin);
            switch(tmp_str[0]) {
                case '1': option_chosen = mo_add_cat; break;
                case '2': option_chosen = mo_find_cat; break;
                case '3': option_chosen = mo_count_entries; break;
                case 'q': option_chosen = mo_exit; break;
            }
        }
    } /* while */
    return(option_chosen);
}
```

8) 我们需要在几个地方稍做停顿，好让用户对他们准备进行的操作予以确认。我们不想让这段提问多次出现在代码里，我们把有关代码提取出来做为一个函数，这就是下面的get_confirm函数：

```
static int get_confirm(const char *question)
{
    char tmp_str[TMP_STRING_LEN + 1];

    printf("%s", question);
    fgets(tmp_str, TMP_STRING_LEN, stdin);
    if (tmp_str[0] == 'Y' || tmp_str[0] == 'y') {
        return(1);
    }
    return(0);
}
```

9) enter_new_cat_entey函数的作用是让用户输入一个新的标题记录。我们不想保存由fgets函数返回的换行符，所以把它去掉了。

> 注意：我们没有使用gets函数，因为它没有办法对缓冲区是否溢出进行检查。要尽量避免使用gets函数！

```
static int enter_new_cat_entry(cdc_entry *entry_to_update)
{
    cdc_entry new_entry;
    char tmp_str[TMP_STRING_LEN + 1];

    memset(&new_entry, '\0', sizeof(new_entry));

    printf("Enter catalog entry: ");
    (void)fgets(tmp_str, TMP_STRING_LEN, stdin);
    strip_return(tmp_str);
    strncpy(new_entry.catalog, tmp_str, CAT_CAT_LEN - 1);

    printf("Enter title: ");
    (void)fgets(tmp_str, TMP_STRING_LEN, stdin);
    strip_return(tmp_str);
    strncpy(new_entry.title, tmp_str, CAT_TITLE_LEN - 1);
    printf("Enter type: ");
    (void)fgets(tmp_str, TMP_STRING_LEN, stdin);
    strip_return(tmp_str);
    strncpy(new_entry.type, tmp_str, CAT_TYPE_LEN - 1);

    printf("Enter artist: ");
    (void)fgets(tmp_str, TMP_STRING_LEN, stdin);
    strip_return(tmp_str);
    strncpy(new_entry.artist, tmp_str, CAT_ARTIST_LEN - 1);

    printf("\nNew catalog entry entry is :-\n");
    display_cdc(&new_entry);
    if (get_confirm("Add this entry ?")) {
        memcpy(entry_to_update, &new_entry, sizeof(new_entry));
        return(1);
    }
    return(0);
}
```

10) 下面是输入曲目资料的enter_new_track_entries函数。它比对应的标题记录函数稍微复杂一些，因为我们允许对现存曲目记录不做修改。

```
static void enter_new_track_entries(const cdc_entry *entry_to_add_to)
{
    cdt_entry new_track, existing_track;
    char tmp_str[TMP_STRING_LEN + 1];
    int track_no = 1;

    if (entry_to_add_to->catalog[0] == '\0') return;
    printf("\nUpdating tracks for %s\n", entry_to_add_to->catalog);
    printf("Press return to leave existing description unchanged,\n");
    printf(" a single d to delete this and remaining tracks,\n");
    printf(" or new track description\n");

    while(1) {
```

11) 我们必须先查明当前曲目编号处是否有现存曲目。我们将根据查询的结果对提示做相应的修改。

```
memset(&new_track, '\0', sizeof(new_track));
existing_track = get_cdt_entry(entry_to_add_to->catalog,
                               track_no);
if (existing_track.catalog[0]) {
    printf("\tTrack %d: %s\n", track_no,
                    existing_track.track_txt);
    printf("\tNew text: ");
}
else {
    printf("\tTrack %d description: ", track_no);
}
fgets(tmp_str, TMP_STRING_LEN, stdin);
strip_return(tmp_str);
```

12) 如果曲目编号处没有现存记录，并且用户也没有添加一条记录，我们就认为曲目都已经添加完了。

```
if (strlen(tmp_str) == 0) {
    if (existing_track.catalog[0] == '\0') {
            /* no existing entry, so finished adding */
        break;
    }
    else {
        /* leave existing entry, jump to next track */
        track_no++;
        continue;
    }
}
```

13) 如果用户单独输入一个字母“d”，就会删除当前以及更高编号的曲目记录。如果del_cdt_entry函数找不到可删除的曲目，它就会返回“false”。

```
if ((strlen(tmp_str) == 1) && tmp_str[0] == 'd') {
        /* delete this and remaining tracks */
    while (del_cdt_entry(entry_to_add_to->catalog, track_no)) {
        track_no++;
    }
    break;
}
```

14) 下面这段代码的作用是添加一个曲目，或者对一个现存曲目进行修改。我们对那个cdt_entry结构的数据项new_track进行构造，然后调用数据库函数add_cdt_entry把它添加到数据库里去。

```
        strncpy(new_track.track_txt, tmp_str, TRACK_TTEXT_LEN - 1);
        strcpy(new_track.catalog, entry_to_add_to->catalog);
        new_track.track_no = track_no;
        if (!add_cdt_entry(new_track)) {
            fprintf(stderr, "Failed to add new track\n");
            break;
        }
        track_no++;
    } /* while */
}
```

15) 函数del_cat_entry的作用是删除一条标题记录。如果标题记录被删除了，原来与它对应的曲目记录也都将被删除。

```
static void del_cat_entry(const cdc_entry *entry_to_delete)
{
```

```
    int track_no = 1;
    int delete_ok;

    display_cdc(entry_to_delete);
    if (get_confirm("Delete this entry and all it's tracks? ")) {
        do {
            delete_ok = del_cdt_entry(entry_to_delete->catalog,
                                      track_no);
            track_no++;
        } while(delete_ok);

        if (!del_cdc_entry(entry_to_delete->catalog)) {
            fprintf(stderr, "Failed to delete entry\n");
        }
    }
}
```

16) 下面这个函数的作用是删除与某个标题相对应的全部曲目：

```
static void del_track_entries(const cdc_entry *entry_to_delete)
{
    int track_no = 1;
    int delete_ok;

    display_cdc(entry_to_delete);
    if (get_confirm("Delete tracks for this entry? ")) {
        do {
            delete_ok = del_cdt_entry(entry_to_delete->catalog, track_no);
            track_no++;
        } while(delete_ok);
    }
}
```

17) 接下来，我们编写了一个非常简单的标题搜索小工具。我们提示用户输入一个字符串，然后开始查找包含这个字符串的标题记录项。可能会查到多个匹配的记录，我们一次一个地把它们按顺序提供给用户：

```
static cdc_entry find_cat(void)
{
    cdc_entry item_found;
    char tmp_str[TMP_STRING_LEN + 1];
    int first_call = 1;
    int any_entry_found = 0;
    int string_ok;
    int entry_selected = 0;

    do {
        string_ok = 1;
        printf("Enter string to search for in catalog entry: ");
        fgets(tmp_str, TMP_STRING_LEN, stdin);
        strip_return(tmp_str);
        if (strlen(tmp_str) > CAT_CAT_LEN) {
            fprintf(stderr, "Sorry, string too long, maximum %d \
                             characters\n", CAT_CAT_LEN);
            string_ok = 0;
        }
    } while (!string_ok);

    while (!entry_selected) {
        item_found = search_cdc_entry(tmp_str, &first_call);
        if (item_found.catalog[0] != '\0') {
            any_entry_found = 1;
            printf("\n");
            display_cdc(&item_found);
            if (get_confirm("This entry? ")) {
                entry_selected = 1;
```

```
            }
        }
        else {
            if (any_entry_found) printf("Sorry, no more matches found\n");
            else printf("Sorry, nothing found\n");
            break;
        }
    }
    return(item_found);
}
```

18) list_tracks函数的作用是列出与给定标题记录对应的全部曲目：

```
static void list_tracks(const cdc_entry *entry_to_use)
{
    int track_no = 1;
    cdt_entry entry_found;

    display_cdc(entry_to_use);
    printf("\nTracks\n");
    do {
            entry_found = get_cdt_entry(entry_to_use->catalog,
                                        track_no);
            if (entry_found.catalog[0]) {
                display_cdt(&entry_found);
                track_no++;
            }
    } while(entry_found.catalog[0]);
    (void)get_confirm("Press return");
} /* list_tracks */
```

19) count_all_entries函数的作用是统计全部曲目：

```
static void count_all_entries(void)
{
    int cd_entries_found = 0;
    int track_entries_found = 0;
    cdc_entry cdc_found;
    cdt_entry cdt_found;
    int track_no = 1;
    int first_time = 1;
    char *search_string = "";

    do {
        cdc_found = search_cdc_entry(search_string, &first_time);
        if (cdc_found.catalog[0]) {
            cd_entries_found++;
            track_no = 1;
            do {
                cdt_found = get_cdt_entry(cdc_found.catalog, track_no);
                if (cdt_found.catalog[0]) {
                    track_entries_found++;
                    track_no++;
                }
            } while (cdt_found.catalog[0]);
        }
    } while (cdc_found.catalog[0]);

    printf("Found %d CDs, with a total of %d tracks\n", cd_entries_found,
                track_entries_found);
    (void)get_confirm("Press return");
}
```

20) 下面是display_cdc函数的代码，它的作用是显示一条标题记录项：

```
static void display_cdc(const cdc_entry *cdc_to_show)
```

```
{
    printf("Catalog: %s\n", cdc_to_show->catalog);
    printf("\ttitle: %s\n", cdc_to_show->title);
    printf("\ttype: %s\n", cdc_to_show->type);
    printf("\tartist: %s\n", cdc_to_show->artist);
}
```

下面是display_cdt函数的代码，它的作用是只显示一条曲目记录项目：

```
static void display_cdt(const cdt_entry *cdt_to_show)
{
    printf("%d: %s\n", cdt_to_show->track_no, cdt_to_show->track_txt);
}
```

21) 工具函数strip_return的作用是删除字符串后面尾缀的换行符。记住UNIX使用一个单独的换行符来结束一行。

```
static void strip_return(char *string_to_strip)
{
    int len;

    len = strlen(string_to_strip);
    if (string_to_strip[len - 1] == '\n') string_to_strip[len - 1] = '\0';
}
```

22) command_mode是一个对命令行参数进行分析的函数。getopt函数是一个确保程序能够接受符合UNIX标准体例的参数的好办法。

```
static int command_mode(int argc, char *argv[])
{
    int c;
    int result = EXIT_SUCCESS;
    char *prog_name = argv[0];

    /* these externals used by getopt */
    extern char *optarg;
    extern optind, opterr, optopt;

    while ((c = getopt(argc, argv, ":i")) != -1) {
        switch(c) {
            case 'i':
                if (!database_initialize(1)) {
                    result = EXIT_FAILURE;
                    fprintf(stderr, "Failed to initialize database\n");
                }
                break;
            case ':':
            case '?':
            default:
                fprintf(stderr, "Usage: %s [-i]\n", prog_name);
                result = EXIT_FAILURE;
                break;
        } /* switch */
    } /* while */
    return(result);
}
```

动手试试：cd_access.c程序清单

1) 现在开始介绍对dbm数据库进行访问的函数。与往常一样，我们以头文件包括语句开始。接着，我们用“#define”语句定义了一些文件，我们将把数据保存在这些文件里。

```
#define _XOPEN_SOURCE

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <ndbm.h>

#include "cd_data.h"

#define CDC_FILE_BASE "cdc_data"
#define CDT_FILE_BASE "cdt_data"
#define CDC_FILE_DIR  "cdc_data.dir"
#define CDC_FILE_PAG  "cdc_data.pag"
#define CDT_FILE_DIR "cdt_data.dir"
#define CDT_FILE_PAG "cdt_data.pag"
```

2) 我们用下面这两个文件范围变量追踪当前数据库：

```
static DBM *cdc_dbm_ptr =NULL;
static DBM *cdt_dbm_ptr =NULL;
```

3) database_initialize函数的默认操作是打开一个现有的数据库，但如果我们给它的参数new_database传递了一个非零值（即布尔真值），就可以强迫它创建一个新的（空白）数据库。如果对数据库的初始化成功了，两个数据库指针也将被初始化，以此表明打开了一个数据库。

```
int database_initialize(const int new_database)
{
    int open_mode = O_RDWR;

    /* If any existing database is open then close it */
    if (cdc_dbm_ptr) dbm_close(cdc_dbm_ptr);
    if (cdt_dbm_ptr) dbm_close(cdt_dbm_ptr);

    if (new_database) {
        /* delete the old files */
        (void) unlink(CDC_FILE_PAG);
        (void) unlink(CDC_FILE_DIR);
        (void) unlink(CDT_FILE_PAG);
        (void) unlink(CDT_FILE_DIR);

        open_mode = O_CREAT | O_RDWR;
    }

    /* Open some new files, creating them if required */
    cdc_dbm_ptr = dbm_open(CDC_FILE_BASE, open_mode, 0644);
    cdt_dbm_ptr = dbm_open(CDT_FILE_BASE, open_mode, 0644);
    if (!cdc_dbm_ptr || !cdt_dbm_ptr) {
        fprintf(stderr, "Unable to create database\n");
        cdc_dbm_ptr = cdt_dbm_ptr = NULL;
        return (0);
    }
    return (1);
}
```

4) 如果数据库是打开着的，data_close就简单地把它关上，再把两个数据库指针设置为null，以此表明当前没有打开着的数据库。

```
void database_close(void)
{
    if (cdc_dbm_ptr) dbm_close(cdc_dbm_ptr);
    if (cdt_dbm_ptr) dbm_close(cdt_dbm_ptr);
```

```
    cdc_dbm_ptr = cdt_dbm_ptr = NULL;
}
```

5) 当我们给下面这个函数传递过去一个指向一个标题文本字符串的指针时，它将检索出一个标题记录项来。如果标题记录没有找到，其返回数据中的标题域catalog将为空。

```
cdc_entry get_cdc_entry(const char *cd_catalog_ptr)
{
    cdc_entry entry_to_return;
    char entry_to_find[CAT_CAT_LEN + 1];
    datum local_data_datum;
    datum local_key_datum;

    memset(&entry_to_return, '\0', sizeof(entry_to_return));
```

6) 我们先做一些预防性检查，确定数据库已经被打开，参数也很合理——即用来进行搜索的关键字里只能包含着合法的字符串和空字符“null”：

```
if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (entry_to_return);
if (!cd_catalog_ptr) return (entry_to_return);
if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (entry_to_return);

memset(&entry_to_find, '\0', sizeof(entry_to_find));
strcpy(entry_to_find, cd_catalog_ptr);
```

7) 我们对dbm函数所要求的datum结构进行设置，然后用dbm_fetch检索数据。如果没有检索到数据，我们将返回空着的entry_to_return结构，这个结构在前面进行过初始化。

```
    local_key_datum.dptr = (void *) entry_to_find;
    local_key_datum.dsize = sizeof(entry_to_find);

    memset(&local_data_datum, '\0', sizeof(local_data_datum));
    local_data_datum = dbm_fetch(cdc_dbm_ptr, local_key_datum);
    if (local_data_datum.dptr) {
        memcpy(&entry_to_return, (char *)local_data_datum.dptr,
                    local_data_datum.dsize);
    }
    return (entry_to_return);
} /* get_cdc_entry */
```

8) 我们希望还对单个的曲目记录项进行检索，而这正是下面这个函数的功用。它的工作方式与get_cdc_entry函数一样，但需要有两参数：一个是指向一个标题字符串的指针，另一个是曲目的编号。

```
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    cdt_entry entry_to_return;
    char entry_to_find[CAT_CAT_LEN + 10];
    datum local_data_datum;
    datum local_key_datum;

    memset(&entry_to_return, '\0', sizeof(entry_to_return));

    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (entry_to_return);
    if (!cd_catalog_ptr) return (entry_to_return);
    if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (entry_to_return);
    /* set up the search key, which is a composite key of catalog entry
       and track number */
    memset(&entry_to_find, '\0', sizeof(entry_to_find));
    sprintf(entry_to_find, "%s %d", cd_catalog_ptr, track_no);

    local_key_datum.dptr = (void *) entry_to_find;
```

```
    local_key_datum.dsize = sizeof(entry_to_find);

    memset(&local_data_datum, '\0', sizeof(local_data_datum));
    local_data_datum = dbm_fetch(cdt_dbm_ptr, local_key_datum);
    if (local_data_datum.dptr) {
        memcpy(&entry_to_return, (char *) local_data_datum.dptr,
                    local_data_datum.dsize);
    }
    return (entry_to_return);
}
```

9) 下面这个add_cdc_entey函数的作用是添加一个新的标题记录项：

```
int add_cdc_entry(const cdc_entry entry_to_add)
{
    char key_to_add[CAT_CAT_LEN + 1];
    datum local_data_datum;
    datum local_key_datum;
    int result;

    /* check database initialized and parameters valid */
    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (0);
    if (strlen(entry_to_add.catalog) >= CAT_CAT_LEN) return (0);

    /* ensure the search key contains only the valid string and nulls */
    memset(&key_to_add, '\0', sizeof(key_to_add));
    strcpy(key_to_add, entry_to_add.catalog);

    local_key_datum.dptr = (void *) key_to_add;
    local_key_datum.dsize = sizeof(key_to_add);
    local_data_datum.dptr = (void *) &entry_to_add;
    local_data_datum.dsize = sizeof(entry_to_add);

    result = dbm_store(cdc_dbm_ptr, local_key_datum, local_data_datum,
                       DBM_REPLACE);

    /* dbm_store() uses 0 for success */
    if (result == 0) return (1);
    return (0);
}
```

10) add_cdt_entry函数的作用是添加一个新的曲目记录项。在访问数据库的时候，标题字符串和曲目编号组合在一起构成其关键字。

```
int add_cdt_entry(const cdt_entry entry_to_add)
{
    char key_to_add[CAT_CAT_LEN + 10];
    datum local_data_datum;
    datum local_key_datum;
    int result;

    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (0);
    if (strlen(entry_to_add.catalog) >= CAT_CAT_LEN) return (0);

    memset(&key_to_add, '\0', sizeof(key_to_add));
    sprintf(key_to_add, "%s %d", entry_to_add.catalog,
                entry_to_add.track_no);

    local_key_datum.dptr = (void *) key_to_add;
    local_key_datum.dsize = sizeof(key_to_add);
    local_data_datum.dptr = (void *) &entry_to_add;
    local_data_datum.dsize = sizeof(entry_to_add);

    result = dbm_store(cdt_dbm_ptr, local_key_datum, local_data_datum,
                       DBM_REPLACE);

    /* dbm_store() uses 0 for success and -ve numbers for errors */
    if (result == 0)
```

```
        return (1);
    return (0);
}
```

11) 既然我们可以往数据库里添东西，要是还能删除它们就更好了。下面这个函数的作用是删除标题记录：

```
int del_cdc_entry(const char *cd_catalog_ptr)
{
    char key_to_del[CAT_CAT_LEN + 1];
    datum local_key_datum;
    int result;

    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (0);
    if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (0);

    memset(&key_to_del, '\0', sizeof(key_to_del));
    strcpy(key_to_del, cd_catalog_ptr);

    local_key_datum.dptr = (void *) key_to_del;
    local_key_datum.dsize = sizeof(key_to_del);

    result = dbm_delete(cdc_dbm_ptr, local_key_datum);

    /* dbm_delete() uses 0 for success */
    if (result == 0) return (1);
    return (0);
}
```

12) 下面这个函数的作用是删除一个曲目。注意：曲目用的关键字是由标题记录项的字符串和曲目编号两者构成的一个复合索引。

```
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    char key_to_del[CAT_CAT_LEN + 10];
    datum local_key_datum;
    int result;

    if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (0);
    if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (0);

    memset(&key_to_del, '\0', sizeof(key_to_del));
    sprintf(key_to_del, "%s %d", cd_catalog_ptr, track_no);

    local_key_datum.dptr = (void *) key_to_del;
    local_key_datum.dsize = sizeof(key_to_del);

    result = dbm_delete(cdt_dbm_ptr, local_key_datum);

    /* dbm_delete() uses 0 for success */
    if (result == 0) return (1);
    return (0);
}
```

13) 现在到达简单的搜索函数。它并不是很复杂，但它展示了这样一种能力，即我们可以在事先不知道关键字的情况下扫描全部的dbm记录项。

因为我们事先并不知道会有多少记录项，所以我们安排这个函数每次调用时只返回一个记录项。如果什么也没找到，记录项将是空着的。为了扫描整个数据库，我们在开始调用这个函数的时候使用了一个指向一个整数的指针“*first_call_ptr”；在第一次调用这个函数的时候，该整数应该被设置为“1”。这样，这个函数就会知道应该从数据库的起始位置开始搜索。而在后续调用中，这个变量将被设置为“0”，而这个函数就会从上次找到记录项的位置开始

继续搜索.

当我们希望重新开始一次搜索的时候，比如要搜索另外一个标题记录项的时候，我们必须先把“*first_call_ptr”重新设置为真（非零值），然后再调用这个函数。把“*first_call_ptr”设置为真等于对搜索重新进行了初始化。

这个函数需要在自己的两次调用之间使用一些内部的状态信息。这样使用户看不到继续一次搜索的复杂性，并且维持了搜索函数在具体实现方面的“神秘感”。

如果搜索文本指向一个空字符“null”，那么所有记录项就都将被认为是匹配的。

```
cdc_entry search_cdc_entry(const char *cd_catalog_ptr, int *first_call_ptr)
{
    static int local_first_call = 1;
    cdc_entry entry_to_return;
    datum local_data_datum;
    static datum local_key_datum;    /* notice this must be static */

    memset(&entry_to_return, '\0', sizeof(entry_to_return));
```

14) 与往常一样，我们先做一些预防性检查：

```
if (!cdc_dbm_ptr || !cdt_dbm_ptr) return (entry_to_return);
if (!cd_catalog_ptr || !first_call_ptr) return (entry_to_return);
if (strlen(cd_catalog_ptr) >= CAT_CAT_LEN) return (entry_to_return);

/* protect against never passing *first_call_ptr true */
if (local_first_call) {
    local_first_call = 0;
    *first_call_ptr = 1;
}
```

15) 如果在调用这个函数的时候，“*first_call_ptr”被设置为true，就表示将从数据库的起始位置开始（或重新开始）搜索。如果“*first_call_ptr”不是ture，我们将简单地前进到数据库里的下一个关键字：

```
    if (*first_call_ptr) {
        *first_call_ptr = 0;
        local_key_datum = dbm_firstkey(cdc_dbm_ptr);
    }
    else {
        local_key_datum = dbm_nextkey(cdc_dbm_ptr);
    }

    do {
        if (local_key_datum.dptr != NULL) {
            /* an entry was found */
            local_data_datum = dbm_fetch(cdc_dbm_ptr, local_key_datum);
            if (local_data_datum.dptr) {
                memcpy(&entry_to_return, (char *) local_data_datum.dptr,
local_data_datum.dsize);
```

16) 我们的搜索工具所做的检查很简单，它只检查当前标题记录项里是否出现了搜索字符串。

```
            /* check if search string occurs in the entry */
            if (!strstr(entry_to_return.catalog, cd_catalog_ptr))
                {
                memset(&entry_to_return, '\0',
                            sizeof(entry_to_return));
                local_key_datum = dbm_nextkey(cdc_dbm_ptr);
            }
```

```
            }
        }
    } while (local_key_datum.dptr &&
            local data_datum.dptr &&
            (entry_to_return.catalog[0] == '\0'));
    return (entry_to_return);
} /* search_cdc_entry */
```

我们将通过下面这个制作文件makefile把所有程序结合在一起。现在还用不着太操心它，因为我们马上就要在下一章开始对它进行学习了。先把下面这些内容敲进计算机并保存为Makefile再说。

```
all:    application

app_ui.o: app_ui.c cd_data.h
        gcc -pedantic -Wall -ansi -g -c app_ui.c

cd_access.o: cd_access.c cd_data.h
        gcc -pedantic -Wall -ansi -g -c cd_access.c

application:    app_ui.o cd_access.o
        gcc -o application -pedantic -Wall -ansi -g app_ui.o cd_access.o -ldbm
```

记住——根据你计算机的设置情况，你可能需要把“-ldbm”替换为“-gdbm”。

要想编译这个新编写的CD唱盘管理软件，请在提示符处敲入下面的命令：

```
$ make -f  Makefile
```

如果一切顺利，可执行文件application将被编译并放在当前子目录里。

当你第一次运行这个程序的时候，别忘了加上“-i”选项好让数据库创建出来。

7.5 本章总结

在这一章里，我们学习了数据管理三个方面的知识。首先，我们学习了关于UNIX内存系统的知识，虽然请求页面虚拟内存的内在实现很高深，但用起来还是相当容易的。我们还介绍了它是如何防止操作系统和其他程序受到非法内存访问操作的干扰和损害的。

接下来，我们看到文件封锁功能是如何让多个程序在数据访问操作中彼此协调的。我们先观察了简单的二进制信号量的机制，然后是一个更复杂的情形，即为共享访问或独占访问而封锁同一个文件的不同部分。

最后，我们学习了dbm数据库，体会到它通过一种非常灵活的索引安排存储和有效检索各种数据块的能力。

第8章 开发工具

在这一章里，我们将向大家介绍一些UNIX系统中的程序开发工具。编译器和调试器肯定是少不了的；除此之外，UNIX还为我们准备了一组工具，它们每一个都能独当一面，并且允许程序设计人员把它们创造性地组合在一起。我们将在这一章介绍几个比较重要的工具，并用这些工具解决一些实际问题，其中包括：

- make命令和制作文件。
- 利用RCS和CVS系统对源代码进行控制。
- 编写一个使用手册。
- 用patch和tar命令发行软件。

8.1 多个源文件带来的问题

当在编写小程序的时候，许多人都会在编辑之后重新编译所有的文件以重建其应用程序。但对一个大型程序来说，再使用这样的办法就会带来一些明显的问题。编辑-编译-测试这一循环的周期将明显延长。如果只改动了一个文件，即使是最有耐心的程序员也不会想去重新编译所有的文件。

如果曾经创建了多个头文件并且把它们用在了不同的源文件里，情况就更复杂了。比如说，我们有三个头文件a.h、b.h、c.h以及三个C语言源文件main.c、2.c、3.c（我们希望读者在实际工作中会给这些文件起些更好的名字），也就是下面这种情况：

```
/* main.c */
#include "a.h"
...

/* 2.c */
#include "a.h"
#include "b.h"
...

/* 3.c */
#include "b.h"
#include "c.h"
...
```

那么，如果程序员只修改了c.h文件，则文件main.c和2.c是用不着重新编译的，因为它们并不依赖于这个头文件。而如果c.h文件有了改变，就必须对依赖于c.h的3.c重新进行编译。可要是b.h发生了改变而程序员又忘了重新编译2.c，就可能导致程序工作失常。

> make工具可以解决这类问题，它会在必要时重新编译受这些改动影响的所有文件。
> make命令的作用不仅仅是编译一个程序。只要你需要使用几个输入文件来产生输出文

件，就可以用它来到达目的。它的其他用法还包括对文档（比如troff文件或TeX文件等）进行处理等。

8.2 make命令和制作文件

我们将会看到make命令相当博大精深，但光凭自己它是无法知道怎样建立你的软件的。用户必须给它提供一个文件，告诉make软件应该如何构造。这个文件就叫做制作文件。

制作文件一般都会和项目的其他源文件放在同一个子目录里。用户的计算机里可以同时存在许多不同的制作文件。事实上，如果用户的项目很大，就完全可以选择使用多个不同的制作文件来管理项目的不同部分。

make命令和制作文件这样一种组合提供了在项目管理方面具有十分强大的功能。它不仅能够用来对源代码进行编译，还可以用来准备使用手册页和把应用软件安装到一个目标子目录去。

8.2.1 制作文件的语法

制作文件由一组依赖关系和规则构成。每个依赖关系由一个目标（即将要创建的文件）和它所依赖的源文件组成；而规则则描述了怎样从被依赖文件创建出目标文件来。比较常见的情况是：只有一个可执行文件被当作目标。

make命令会读取制作文件，它先确定需要创建哪些个目标文件，然后比较源文件的日期和时间以决定采用哪条规则来构造目标文件。在很多情况下，创建最终的目标文件之前必须先创建出一些过渡性目标。make命令根据制作文件来确定目标文件的创建顺序和规则的应用顺序。

8.2.2 make命令的选项和参数

make程序自己有几个选项，其中最常用的三个是：

- “-k”，它的作用是让make在遇到出错的时候继续执行，而不是在检测到第一个错误时就停下来。这样，我们就能利用这个选项通过一遍操作查出都有哪些个源文件没有通过编译。
- “-n”，它的作用是让make命令在没有实际执行的情况下列出将会执行的操作步骤。
- “-f <filebname>”，它的作用是让make命令使用指定的制作文件。如果没有使用这个选项，make命令将使用当前子目录里第一个名为makefile的文件。如果这个文件不存在，它会去查找一个名为Makefile的文件。不少UNIX程序员都喜欢使用Makefile做为制作文件的文件名。

目标文件通常都是一些可执行的程序文件，为了制作某个特定的目标文件，我们可以把它的名字做为一个参数传递给make命令。如果不指定目标文件名，make将尝试创建制作文件里的第一个目标。大多数程序员都会在自己的制作文件里把第一个目标定义为all，然后把所有其他的目标列为all的附属品。这样做可以明确地指定制作文件在没有给出任何目标的情况下将要缺省创建的目标。我们建议大家坚持使用这种办法。

1. 依赖关系

依赖关系定义了最终应用程序里的每个文件与源文件之间的关系。在上面的例子里，我们可以把依赖关系定义为最终应用程序依赖于main.o、2.o和3.o；也就是依赖于main.o的下级依赖关系（main.c和a.h）、2.o的下级依赖关系（2.c、a.h和b.h）和3.o的下级依赖关系（3.c、b.h和c.h）。也就是说，main.o受main.c和a.h文件中修改的影响，如果这两个文件发生了变化，就需要通过重新编译main.c文件来重建它。

这些规则在制作文件里的写法是：先写出目标的名字，然后紧跟着一个冒号，任意个空格或制表符，最后是用一个空格或制表符隔开的文件清单，清单里的文件将用来生成该目标文件。与我们的例子相对应的依赖关系如下所示：

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.0: 3.c b.h c.h
```

它表示myapp依赖于main.o、2.0和3.o，而main.o又依赖于main.c和a.h，等等。

这组依赖关系构成一个树状结构，源文件之间的联系一目了然。我们可以非常清楚地看到：如果b.h发生了变化，就需要改动2.o和3.o，而既然2.o和3.o变化了，整个myapp也将重建。

如果我们想一次制作多个文件，就可以利用名义上的目标all。假设我们的应用软件是由一个二进制文件myapp和一个使用手册页myapp.1组成的，我们可以用下面这行进行定义：

```
all: myapp myapp.1
```

如果我们没有使用一个all目标，make命令就将只创建它在制作文件里找到的第一个目标。

2. 规则

规则是制作文件里的第二部分内容，它们定义了目标的创建方式。那么，在我们上面的例子里，如果make确定需要重建2.o，它将使用哪条命令呢？很简单，使用“gcc -c 2.c”命令就足够了（我们将在后面的内容看到，make命令其实知道不少的缺省规则）。那如果我们想定义一个头文件子目录，或者如果我们想为今后的调试工作设置一些符号链接信息又该怎么办呢？这就需要我们在制作文件里对某些规则做明确的定义。

讲到这，我们遇到了制作文件奇怪而又糟糕的语法现象：一个空格和一个制表符是有区别的。

> 规则所在的行必须以一个制表符开始，用空格是不行的。因为几个连续的空格和一个制表符看起来很相似，又因为空格和制表符在UNIX程序设计几乎所有的其他方面都没有太大的区别，所以这经常会引起一些问题。此外，如果制作文件中的某一行是以一个空格结尾的，它就会引起make命令执行的失败。

这是一个历史遗留问题，而且因为已经有太多制作文件存在着了，所以即使想改正它也有些力不从心，自己多注意吧！如果缺少了制表符，make命令就不会工作，这多少可以让我们“死的明白”。

> 动手试试：一个简单的制作文件

大多数规则都带有一个也能在命令行上敲入执行的简单命令。就我们的例子来说，我们使用Makefile1做为制作文件的文件名。它的内容如下所示：

```
myapp: main.o 2.o 3.o
   gcc -o myapp main.o 2.o 3.o

main.o: main.c a.h
   gcc -c main.c

2.o: 2.c a.h b.h
   gcc -c 2.c

3.o: 3.c b.h c.h
   gcc -c 3.c
```

因为我们的制作文件没有使用常见的缺省文件名makefile或Makefile，所以我们将在调用make命令时给它加上一个“-f”选项。如果我们在一个没有任何源文件的子目录使用这个命令，就会出现下面这样的情况：

```
$ make -f Makefile1
make: *** No rule to make target 'main.c', needed by 'main.o'.  Stop.
$
```

make命令假定制作文件里的第一个目标myapp是我们想要创建文件。它会对其他依赖关系做进一步检查，最终确定需要有一个main.c文件。因为我们还没有创建这个文件，制作文件里也没有说怎样才能把它创建出来，所以make会报告出现一个错误。创建这个文件之后我们再来试试。因为我们只对结果感兴趣，所以不妨把这些文件弄得简单点儿。头文件可以是空的，所以我们可以用touch命令来创建它们。如下所示：

```
$ touch a.h
$ touch b.h
$ touch c.h
```

main.c文件里包含着main函数，它调用了function_two和function_three两个函数；而function_two和function_three这两个函数是在另外两个文件里定义的。这个源文件通过“#include”语句包括上相应的头文件，使它们看起来就好象是依赖于这两个头文件的内容似的。它其实算不上是个应用程序，下面是它的程序清单：

```
/* main.c */
#include "a.h"

extern void function_two();
extern void function_three();

int main()
{
    function_two();
    function_three();
    exit (EXIT_SUCCESS);
}
```

```
/* 2.c */
#include "a.h"
#include "b.h"

void function_two() {
}
```

```
/* 3.c */
#include "b.h"
#include "c.h"

void function_three() {
}
```

我们再次执行make命令，如下所示：

```
$ make -f Makefile1
gcc -c main.c
gcc -c 2.c
gcc  c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

这是一次成功的制作。

操作注释：

make命令对制作文件中定义依赖关系的部分进行了处理，确定好需要创建的文件和创建它们的顺序。虽然我们把创建myapp的规则列在最前面，make还是能够找出创建文件的正确顺序。接下来，它调用我们在规则部分给出的命令创建出相应的文件。make命令会在执行过程中把命令显示出来。现在来看看我们的制作文件能否正确地处理好对文件b.h进行了改动的情况。

```
$ touch b.h
$ make -f Makefile1
gcc -c 2.c
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

make命令读了我们的制作文件，确定了重建myapp所需要的最少命令，并以正确的顺序执行了它们。我们来看看如果删掉个目标代码文件会发生什么样的事情。

```
$ rm 2.o
$ make -f Makefile1
gcc -c 2.c
gcc -o myapp main.o 2.o 3.o
$
```

make再次正确地确定出需要采取的动作。

8.2.3 制作文件中的注释

制作文件中以井字号（#）开始的文本行就是注释行，它一直延续到这一行的结束。制作文件中的注释与C语言程序中的作用相同，可以帮助程序的编写者和其他人弄明白这个文件到底是用来干什么的。

8.2.4 制作文件中的宏

即使这是make命令和制作文件中仅有的东西，它们仍会是对多文件项目进行管理的有力武器。对由非常大量的文件构成的项目来说，其制作文件也相应的会比较庞大，比较不灵活。制

作文件允许使用宏定义，这使我们能够按普通化的格式把它们写出来。

制作文件中的宏定义语句的写法是“MACRONAME=value”，引用宏的办法是在需要使用宏定义值的地方写出“$(MACRONAME)”或“${MACRONAME}”。make的某些版本还接受“$MACRONAME”这样的写法。如果想把一个宏定义名设置为空白，可以在等号（=）后面什么都不写。

在制作文件里，宏定义通常被用做编译器的命令选项。在软件的开发过程中，一般做法是把调试信息包括在它的编译结果里，先不对它进行优化。面到了发行它的时候，往往又需要反过来做；不包含调试信息的二进制代码一般比较短小，运行得也更快。

Makefile1的另外一个问题是它假设编译器的名字是gcc。在其他UNIX系统上，我们使用的可能是cc或c89。如果我们想为另一个UNIX版本编写一个制作文件，或者只是想在现有系统上使用另外一个编译器，就需要对制作文件中的几行语句进行修改，让它能够完成我们要求的工作。而宏定义正是收集这些与系统有关的各种信息的好办法，这使得对它们的修改更容易进行。

宏通常都是在制作文件里面定义的，但也可以在调用make命令的时候在命令行上给出宏定义，比如“make CC=c89”这样。命令行上的宏定义将替换制作文件中的定义。

动手试试：带宏定义的制作文件

下面是我们制作文件的一个改进版本，它使用了宏定义，我们给它起名为Makefile2。

```
all: myapp

# Which compiler
CC = gcc

# Where are include files kept
INCLUDE = .

# Options for development
CFLAGS = -g -Wall -ansi

# Options for release
# CFLAGS = -O -Wall -ansi

myapp: main.o 2.o 3.o
   $(CC) -o myapp main.o 2.o 3.o

main.o: main.c a.h
   $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c

2.o: 2.c a.h b.h
   $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c

3.o: 3.c b.h c.h
   $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

删掉刚才的旧安装，再用这个新制作文件建立一个新安装。我们将看如下所示的输出内容：

```
$ rm *.o myapp
$ make -f Makefile2
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c 2.c
gcc -I. -g -Wall -ansi -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

操作注释：

make命令会把“$(CC)”、“$(CFLAGS)”、和“$(INCLUDE)”替换为相应的定义，这一点与C语言中的“# define”语句很相似。现在，如果我们想改变编译器命令，只需在制作文件里修改一条语句就可以了。

事实上，make内部本身就带有一些特殊的宏定义，它们可以使制作文件更简洁。我们把几个常用的列在下面的表里，读者很快就可以在操作示例中看见它们。这些宏定义是在使用前才展开的，所以宏定义的含义会随着制作文件的进展而发生变化。事实上，如果宏定义的用法不是这样，它们就没有太大的用处见表8-1。

表 8-1

$?	当前目标最近一次被修改时的改动清单
$@	当前目标的名字
$<	现在正被处理着的文件的名字
$*	现在正被处理着的文件的名字，不带任何后缀

在制作文件里我们还经常会看到另外两个特殊字符，它们出现在命令的前面。

“-”告诉make不要理会任何错误。比如说，如果你想创建一个子目录，但不想看到任何出错信息（如果子目录已经存在就会给出一条错误信息），就可以在mkdir命令的前面加上一个减号“-”。我们很快就可以在例子里看到“-”了。

“@”告诉make在执行一条命令之前不要把命令本身输出到标准输出去。当你想用echo语句给出一些提示信息的时候，这个字符正好派上用场。

8.2.5 多个制作目标

经常会出现有不止一个目标文件的情况，人们也经常把几组命令放在同一个软件里。通过扩展我们的制作文件就能达到这一目的。我们将在下面增加一个“clean”选项和一个“install”选项，前者的作用是删除不再需要的目标代码文件，而后者的作用是把应用程序移动到另一个子目录里去。

动手试试：多个制作目标

这里是制作文件下一个版本Makefile3文件的内容：

```
all: myapp

# Which compiler
CC = gcc

# Where to install
INSTDIR = /usr/local/bin

# Where are include files kept
INCLUDE = .

# Options for development
CFLAGS = -g -Wall -ansi
```

```
# Options for release
# CFLAGS = -O -Wall -ansi

myapp: main.o 2.o 3.o
   $(CC) -o myapp main.o 2.o 3.o

main.o: main.c a.h
   $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c

2.o: 2.c a.h b.h
   $(CC) -I$(INCLUDE) $(CFLAGS)  c 2.c

3.o: 3.c b.h c.h
   $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c

clean:
   -rm main.o 2.o 3.o

install: myapp
   @if [ -d $(INSTDIR) ]; \
      then \
      cp myapp $(INSTDIR);\
      chmod a+x $(INSTDIR)/myapp;\
      chmod og-w $(INSTDIR)/myapp;\
      echo "Installed in $(INSTDIR)";\
   else \
      echo "Sorry, $(INSTDIR) does not exist";\
   fi
```

这个制作文件有几个需要大家注意的地方。首先，特殊目标all还是只定义了myapp这一个目标。因此，如果我们在执行make的时候没有指定目标，其缺省行为还将是创建目标myapp。如果后续命令必须在它前面的所有命令都执行成功的前提下才会执行，我们在写出这些命令的时候就必须把它们用“&&”连接起来，如下所示：

```
@if [ -d $(INSTDIR) ]; \
      then \
      cp myapp $(INSTDIR) &&\
      chmod a+x $(INSTDIR)/myapp && \
      chmod og-w $(INSTDIR/myapp && \
      echo "Installed in $(INSTDIR)" ;\
   else \
      echo "Sorry, $(INSTDIR) does not exist" ; false ; \
   fi
```

大家应该还记得我们曾经在shell程序设计那一章里见过“&&”，它的作用是把shell命令“与”在一起，每一个后续命令只有在前面的命令都执行成功的前提下才会被执行。在这一章里，我们并不需要过分关心前面命令的执行是否成功，所以今后将坚持使用简单的写法。

还需要注意的地方是那两个新增加的目标clean和install。目标clean使用rm命令删除了目标代码文件。这个命令的前面加上了一个减号（-），告诉make不理会这条命令的执行结果，也就是说，即使rm命令因为目标代码文件不存在而返回了一个错误，“make clean”也会成功。制作目标“clean”的规则没有给“clean”定义任何依赖关系，“clean:”的后面是空着的。因此，make总会认为这个目标已经过了期，只要把clean指定为一个目标，就会执行它的规则。

目标install依赖于myapp，所以make会知道在执行制作install的其他命令之前必须先创建出myapp来。install的制作规则是由几个shell脚本程序命令组成的。make会启动一个shell来执行规则，并且每一条规则都会启动一个新的shell，所以我们必须在上面每一行代码的尾部都加上一个反斜线字符，让所有这些shell脚本程序命令在逻辑上成为一个完整的命令行，传递到一个被

启动的shell里去执行。这个命令的最前面有一个“@”符号，它告诉make在执行这些规则之前不要在标准输出上把命令本身显示出来。

要想在/usr/local/bin子目录里安装新命令，普通用户的权限就不够用了。在调用执行“make install”之前，你可以修改这个子目录的权限，或者把自己的身份修改为root根用户（用su命令）。下面是它的执行情况：

```
$ rm *.o myapp
$ make -f Makefile3
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c 2.c
gcc -I. -g -Wall -ansi -c 3.c
gcc -o myapp main.o 2.o 3.o
$ make -f Makefile3
make: Nothing to be done for 'all'.
$ rm myapp
$ make -f Makefile3 install
gcc -o myapp main.o 2.o 3.o
Installed in /usr/local/bin
$ make -f Makefile3 clean
rm main.o 2.o 3.o
$
```

操作注释：

我们先把myapp和所有目标代码文件都删除掉。make命令自己知道需要使用目标all，也就是需要创建出myapp来。然后我们再次运行make，但因为myapp已经是最新的了，所以make什么事情也没有做。接下来，我们删除掉myapp，然后执行“make install”。它会重新创建出二进制文件并把它们拷贝到安装目录里去。最后，我们执行“make clean”删除了当前子目录里全部的目标代码文件。

8.2.6 内建规则

到目前为止，我们对制作文件里的每个操作步骤都进行了精确的定义。事实上，make本身就带有大量内建的规则，它们可以极大地简化制作文件的编写工作，尤其是在我们有大批源文件的情况下更能发挥作用。我们先来编写一个foo.c文件，它的内容就是传统的“Hello World”程序。

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

int main()
{
    printf("Hello World\n");
    exit (EXIT_SUCCESS);
}
```

在不指定制作文件的情况下，我们用make来编译它：

```
$ make foo
cc     foo.c   -o foo
$
```

正如大家所看到的，虽然make选用了cc而不是gcc，但它还是知道应该如何去调用编译器的。

有时候，这些内建规则又被称为推导规则。这些缺省的规则使用的都是宏定义，因此给宏定义一个新的值就能改变其缺省行为。

```
$ rm foo
$ make CC = gcc CFLAGS = "-Wall -g" foo
gcc -Wall -g     foo.c    -o foo
$
```

我们可以通过“-p”选项让make把它的内建规则都打印出来。这些内建规则实在是太多了，根本没办法在这里把它们都列出来，所以我们只给出了GNU版make的“make -p”命令的部分输出，它只包括很少的一部分规则：

```
OUTPUT_OPTION = -o $@

COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c

.c.o:
    $(COMPILE.c) $< $(OUTPUT_OPTION)
```

既然有了这些内建的规则，我们就用不着再为目标代码文件写什么制作规则了。我们的制作文件将变得很简洁，只需写出依赖关系就可以了。下面是我们制作文件的相关内容，是不是简单多了：

```
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

我们可以在Wrox出版社站点上找到名为Makefile 4的可下载代码。

8.2.7 后缀规则

我们看到内建规则在使用中都加有后缀（有点儿类似于DOS的扩展文件名），这样做的含义是：如果向make命令给出一个带有特定后缀名的文件，它就会知道要用哪条规则去另外创建出一个后缀名不同的文件来。最常见的操作是从一个以“.c”为后缀名的文件创建出一个以“.o”为后缀名的文件来，而最常见的规则也就是干这个用的。这条规则一般是用编译器进行编译，但不对源文件进行链接。

有时候，我们需要制定一些新的规则。笔者在日常工作中经常需要用几种不同的编译器对源文件进行编译：其中两个编译器是用在MS-DOS下的，gcc是用在Linux下的。为了让两个MS-DOS编译器其中之一能够顺利工作，给源文件（它们用的是C++而不是C语言）起名字时就必须加上一个“.cpp”后缀。但在Linux中使用的make版本没有对“.cpp”文件进行编译的内建规则。（它倒是有一条针对“.cc”文件的规则，UNIX下的C++文件使用这个后缀的情况比较常见一些。）

这个问题的解决办法有两个，一是为每个源文件定义一条相应的规则；二是为了让make能够从带“.cpp”后缀名的文件开始创建出目标代码文件来而给它制定一条新的规则。因为项目中的源文件数量确实很多，所以制定一条新规则可以节省大量的打字时间，也使给项目增加新程序的工作更容易进行一些。

要想增加一个后缀规则，我们必须先在制作文件里增加一行语句，告诉make新后缀是什么；然后我们再用这个新后缀写出一条规则。make使用特殊语法“.<old suffix>.<new suffix>:”来定义一个通用性规则，采用这条规则新创建出来的文件保留了原来文件名的前半部分，但使用新后缀替换掉了老后缀。

动手试试：后缀规则

下面是我们制作文件的一个片段，这是一个新的通用性规则，它的作用是把“.cpp”文件转换为“.o”文件。这个片段需要插入到文件的顶部，紧跟在“all: myapp”语句的后面。我们给新制作文件起名为Makefile5。

```
.SUFFIXES:      .cpp

cpp.o:
   $(CC) -xc++ $(CFLAGS) -I$(INCLUDE) -c $<
```

我们来看看新规则是怎样做的：

```
$ cp foo.c bar.cpp
$ make -f Makefile5 bar
gcc -xc++ -g -Wall -ansi -I. -c bar.cpp
gcc   bar.o   -o bar
rm bar.o
$
```

操作注释：

特殊依赖关系“.cpp.o:”告诉make紧随其后的规则将用来把一个后缀名为“.cpp”的文件转换为一个后缀“.o”文件。在写出这个依赖关系的时候我们使用了特殊的宏定义名称，因为我们此时还不知道将要被转换的文件到底是什么名字。要想弄明白这条规则，大家只需记住“$<”将被扩展为（带着“老”后缀名的）起始文件的名字。注意：我们只需告诉make怎样才能从一个“.cpp”文件得到一个“.o”文件；make已经知道怎样才能从一个目标代码文件得到一个二进制可执行文件了。

当我们调用make命令的时候，它将使用我们的新规则通过那个bar.cpp文件得到一个bar.o文件，然后再使用内建规则从“.o”文件得到一个二进制可执行文件。那个“-xc++”标志的作用是告诉gcc这是一个C++源代码文件。

8.2.8 用make命令管理函数库

当我们工作在一个大型项目上的时候，一种比较常见且又方便的做法是把几个编译产品用一个函数库来管理。函数库是包含着一组目标代码文件的文件，后缀名一般是“.a”（a是英文archive的字头，表示这是一个库文件）。make命令在函数库管理方面使用了一个特殊的语法，它把函数库的管理工作变得非常简单。

这个语法是“lib(file.o)”，它的作用是把目标代码文件file.o添加到lib.a库里去。make命令有一个专门用来管理函数库的内建规则，它的常见形式如下所示：

```
.c.a:
   $(CC) -c $(CFLAGS) $<
   $(AR) $(ARFLAGS) $@ $*.o
```

宏定义“$(AR)”和“$(ARFLAGS)”的缺省取值分别是ar命令和它的rv选项。这个相当简短的语法告诉make需要使用两条规则才能从一个“.c”文件得到一个“.a”库文件。

第一条规则的作用是对源文件进行编译以生成一个目标代码；第二条规则的作用是用ar命令把新目标代码文件添加到函数库里去。假设我们已经有一个名为fud的函数库了，而库里又包含着一个名为bas.o的文件，则其处理过程应该是：第一条规则里的“$<”被替换为“bas.c”，第二条规则里的“$@”被替换为函数库的名字“fud.a”，“$*”被替换为名字“bas”。

动手试试：函数库的管理

在实践中，这个语法的使用方法其实是很简单的。先对我们的软件进行修改，把文件2.o和3.o放到一个名为mylib.a的函数库里。我们需要对制作文件做很少的几处修改，下面是Makefile6的最终结果。（我们省略了对C++程序进行处理的规则，因为我们不再需要它们了。）

```
all: myapp

# Which compiler
CC = gcc

# Where to install
INSTDIR = /usr/local/bin

# Where are include files kept
INCLUDE = .

# Options for development
CFLAGS = -g -Wall -ansi

# Options for release
# CFLAGS = -O -Wall -ansi

# Local Libraries
MYLIB = mylib.a

myapp: main.o $(MYLIB)
   $(CC) -o myapp main.o $(MYLIB)

$(MYLIB): $(MYLIB)(2.o) $(MYLIB)(3.o)
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h

clean:
   -rm main.o 2.o 3.o $(MYLIB)

install: myapp
   @if [ -d $(INSTDIR) ]; \
    then \
      cp myapp $(INSTDIR);\
      chmod a+x $(INSTDIR)/myapp;\
      chmod og-w $(INSTDIR)/myapp;\
      echo "Installed in $(INSTDIR)";\
   else \
      echo "Sorry, $(INSTDIR) does not exist";\
   fi
```

请注意我们是如何通过缺省规则来完成大部分工作的。现在对我们的新版制作文件进行测试。

```
$ rm -f myapp *.o mylib.a
$ make -f Makefile6
gcc  g -Wall -ansi    -c main.c -o main.o
gcc -g -Wall -ansi    -c 2.c -o 2.o
ar rv mylib.a 2.o
ar: creating mylib.a
a    2.o
gcc  g -Wall -ansi    -c 3.c -o 3.o
ar rv mylib.a 3.o
a    3.o
gcc -o myapp main.o mylib.a
$ touch c.h
$ make -f Makefile6
gcc  g -Wall -ansi    -c 3.c -o 3.o
ar rv mylib.a 3.o
a    3.o
gcc -o myapp main.o mylib.a
$
```

操作注释：

首先，我们删除了所有的目标代码文件和库文件，make命令会因此而重建myapp。它将先编译并创建出函数库，然后把main.o与函数库链接在一起创建出myapp。接下来，我们对3.o的依赖关系规则进行了测试，这条规则告诉make如果c.h改变了，就将对3.o重新进行编译。它正确地完成了这一工作，编译了3.o并刷新了函数库，然后重新链接，最终创建出一个新的myapp可执行文件来。

8.2.9　高级论题：制作文件和下级子目录

工作在大型项目上的时候经常会出现这样的情况，即我们需要把组成一个函数库的几个文件从那些主文件分离出来并把它们保存到一个下级子目录里去。如果使用make命令来完成这一工作，有两个办法可供选用。

第一个办法是：在下级子目录里编写出第二个制作文件，对下级子目录里的文件进行编译并把它们保存到一个函数库里，然后把那个库文件拷贝到父目录里去。在父目录的主制作文件里应该有一条制作函数库的规则，该规则的写法大致如下，其作用是调用第二个制作文件：

```
mylib.a:
    (cd mylibdirectory; $(MAKE) )
```

这条规则的含义是：当make调用建立函数库的规则时，它永远会先尝试制作mylib.a。它会切换到下级子目录mylibdirectory去，再调用一个新的make命令对函数库进行管理。为完成这一工作将启动一个新的shell，因此，使用该制作文件的程序并不执行那条“cd”命令。只有为完成建立函数库工作而启动的那个shell置身于另一个不同的子目录里。括号的作用是保证所有这些处理都是由一个shell完成的。

第二个办法是在原来的制作文件里添加一些宏定义。新添加的宏定义是通过在我们已经见过的宏定义的尾部追加一个字母得到的，字母“D”代表子目录，字母“F”代表文件名。这样我们就可以用下面的语句替代内建的“.c.o:”后缀规则：

```
.c.o:
      $(CC) $(CFLAGS) -c $(@D)/$(<F) -o $(@D)/$(@F)
```

这条规则的作用是编译一个下级子目录里的文件并把其目标代码放在该下级子目录里。然后，我们将使用下面这样的依赖关系和规则在当前子目录里制作出函数库来：

```
mylib.a:  mydir/2.o mydir/3.o
    ar -rv mylib.a $?
```

在项目里到底要选用哪一种办法需要由读者自己来决定。许多项目会采取比较简单的做法，就是避免使用下级子目录，但这样将导致出现源目录里容纳的文件数量过多的现象。通过上面这些简短的论述，读者应该看出只需稍微增加一点复杂性，我们就可以在下级子目录里使用make命令。

8.2.10 GNU的make和gcc命令

如果读者使用的是GNU的make命令和GNU的gcc编译器，那么在我们刚才介绍过的内容以外还将有两个有趣的选项。

第一个是make命令的“-jN”（j代表“job”，即计算机作业）选项。它允许make命令同时执行N个命令。如果项目的不同部分能够彼此独立地进行编译，make就会同时执行几条规则。根据你系统的配置情况，这可以大量节约重新编译所需要花费的时间。如果你有许多个源文件，这个选项就值得一试。一般做法是以一个比较小的数字（比如“-j3”）为出发点。如果读者需要和其他用户共享自己的计算机，在使用作业选项的时候就要谨慎从事。其他用户可能不喜欢你每次编译都要启动大量进程的做法。

另一个额外的收获是gcc的“-MM”选项。它的作用是产生一个依赖关系清单，清单格式适用于make命令。软件项目往往会有很多个源文件，而每个源文件又包含着头文件的多种组合。要想在一个这样的项目上理出正确的依赖关系其难度可能相当大（但又非常重要）。如果只是简单地让每一个源文件依赖于它的每一个头文件，就往往会毫无必要地对文件进行编译。而从另一方面来看，如果你漏掉了一些依赖关系，问题将会变得更糟，因为某些需要重新编译的文件被你漏掉了。

动手试试：“gcc -MM”命令

```
$ gcc -MM main.c 2.c 3.c
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
$
```

操作注释：

gcc编译器逐行列出了软件要求的依赖关系，其格式适合直接插入到一个制作文件里去。我们只需先把它的输出保存到一个临时文件，再把它插入到制作文件里就可以得到一个完美的依赖关系规则集合了。如果你拥有一份gcc编译器，还出现依赖关系错误可就说不过去了！

如果你对自己的制作文件信心十足，还可以使用makedepend工具，它的作用与“-MM”选项很相似，但它的做法是把依赖关系追加到给定制作文件的末尾。

在结束对制作文件的讨论之前，我们认为有必要让大家明白这样一件事：制作文件并不仅

仅能够用于编译代码或创建函数库。只要是可以通过一系列命令从一些输入文件得到一个输出文件的工作，都可以通过制作文件来自动地完成。典型的“非编译器”用途包括调用AWK或sed命令对某些文件进行处理，或生成使用手册页等。

8.3 源代码控制系统

随着接手的项目越来越大，源文件改动方面的管理就越来越重要，如果项目上的开发人员不止一个时就更是如此。UNIX在源文件管理方面有两个广泛使用的软件系统，它们是RCS（Revision Control System，修订控制系统）和SCCS（Source Code Control System，源代码控制系统）。

RCS工具包和它们的源代码都可以从Free Software foundation（自由软件基金会）获得，而SCCS是由AT&T公司在UNIX的System V版本上引入的，现在已经成为X/Open技术规范的一个组成部分了。除此之外，还有许多第三方的源代码控制系统，其中最有名的可能要算CVS（Concurrent Version System）了，它比SCCS或者RCS更先进，但使用面还不够广泛。

在这一章里，我们将重点介绍RCS，但也会把RCS命令与SCCS命令进行比较。大家将会看到它们提供了近似的功能，两者之间的切换也比较容易。RCS还有适用于MS-DOS的版本，其中包括一些具有商业化支持的产品。我们还将对CVS源代码控制系统略加介绍，它非常适用于多名开发人员通过网络互相合作的情况。

8.3.1 RCS系统

RCS系统提供了许多对源代码文件进行管理的命令。它能够跟踪并记录下源代码文件的每一处改动，这些改动都保存在一个文件里；改动清单记载着足够的细节，能够重建出任何一个以前的版本。它还允许我们为每个改动保存一个与之对应的注释，这在你回顾文件改动的历史时将发挥出极大的作用。

随着项目的进展，我们可以把每个大的改动或者对源文件程序漏洞的修补分别记录下来，还可以给每个改动加上一些注释并把它们也保存起来。这在需要复查文件改动情况和检查程序漏洞修补点的时候是非常有用的，我们甚至可以用它来查出程序漏洞是在什么位置引入的！

因为RCS只保存版本之间的不同之处，所以它的存储空间使用效率也是很高的。它还可以在我们误删了文件的时候检索出以前的修订情况。

1. rcs命令

我们从一个我们希望对之进行管理的文件的初始版本开始。我们将以important.c为例向大家介绍一个完整的控制流程：它最初是foo.c文件的一个拷贝，在文件的最开始有以下的注释内容：

```
/*
  This is an important file for managing this project.
  It implements the canonical "Hello World" program.
*/
```

首先要用rcs命令对这个文件的RCS控制进行初始化。命令“rcs -i”的作用是初始化RCS控制文件。如下所示：

```
$ rcs -i important.c
RCS file: important.c,v
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> This is an important demonstration file
>> .
done
$
```

我们可以使用多行注释。结束注释输入提示符需要在文本行上单独使用一个英文句号（.）或敲入文件尾字符，它通常对应于“Ctrl-D”组合键。

执行完这条命令之后，rcs将新创建出一个带“,v”后缀名的只读文件来。如下所示：

```
$ ls -l
-rw-r--r--  1 rick     users         226 Feb 11 16:25 important.c
-r--r--r--  1 rick     users         105 Feb 11 16:26 important.c,v
$
```

如果读者想把自己的RCS文件保存到另一个子目录里去，可以简单地在第一次使用rcs命令之前先建立一个名为RCS的下级子目录。所有rcs命令都将自动地把rcs文件保存到RCS子目录里去。

2. ci命令

现在给我们的文件“订个房间”，用ci命令把它的当前版本封存起来。

```
$ ci important.c
important.c,v  <-  important.c
initial revision: 1.1
done
$
```

如果我们忘记了先要使用“rcs -i”，RCS会提示输入一段对文件的描述。与SCCS相比，RCS的优点之一就是更能容忍人类的错误。如果我们现在来看看子目录中的文件，就会发现important.c已经被删除了。

```
$ ls -l
-r--r--r--  1 rick     users         442 Feb 11 16:30 important.c,v
$
```

文件内容及其控制信息都已经被保存到RCS文件important.c,v里去了。

3. co命令

如果我们想对文件进行修改，必须先“退房”，取出文件。如果只是想读一读文件，我们可以用co命令以只读权限重建出文件的当前版本。如果还想对它进行编辑，就必须用“co -l”命令来锁定文件。这样做的原因是：在一个团队开发项目中，确保任一时刻只有一个人能够修改给定文件的做法是很重要的，这也是为什么文件的某个给定版本只能有一份拷贝拥有写权限的原因。但文件以写权限被取出的时候，RCS文件将被锁定。我们来解封并锁定这个文件的一个拷贝：

```
$ co -l important.c
important.c,v  ->  important.c
revision 1.1 (locked)
done
$
```

然后查看了目录文件清单：

```
$ ls -l
- r w - r - - r - -   1 rick     users          226 Feb 11 16:35 important.c
- r - - r - - r - -   1 rick     users          452 Feb 11 16:35 important.c,v
$
```

现在多了一个供编辑使用的文件，对文件的改动将在这个文件里进行。我们做点儿编辑工作，把新版本存盘，然后再次使用ci命令把修改后的文件封存起来。important.c的输出部分现在是下面这个样子的了：

```
printf("Hello World\n" );
printp("This is an extra line added later\r" );
```

ci命令的使用方法如下所示：

```
$ ci important.c
important.c,v  <--  important.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> Added an extra line to be printed out.
>> .
done
$
```

> 如果在封存文件的时候还想保有文件的加锁状态使同一用户还能对该文件做后续调试，我们必须在调用ci的时候加上“-l”选项。这样，子目录里还将自动留有该程序的一个解封版本供同一用户继续调试。

现在，我们再次封存了文件的修订版本。如果我们查看子目录的文件清单，就会发现important.c文件又不见了：

```
$ ls -l
- r - - r - - r - -   1 rick     users          633 Feb 11 16:37 important.c,v
$
```

4. rlog命令

有时候，我们需要查看文件的改动清单。用rlog命令可以完成这一工作，如下所示：

```
$ rlog important.c

RCS file: important.c,v

Working file: important.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
comment leader: " * "
keyword substitution: kv
total revisions: 2;     selected revisions: 2
description:
This is an important demonstration file
----------------------------
revision 1.2
date: 1999/02/11 16:37:35;  author: rick;  state: Exp;  lines: +1  0
Added an extra line to be printed out.
----------------------------
revision 1.1
date: 1999/02/11 16:30:19;  author: rick;  state: Exp;
```

```
Initial revision
================================================================================
$
```

第一部分内容是对该文件的一个描述以及rcs命令正在使用的选项。接着，rlog命令列出了对文件的各项修改，同时列出的还有我们封存修订后的文件时输入的注释文本。修订版1.2中的“line: +1 -0”表明在这一修订版里我们添加了一行语句，删除了零行。

如果我们想在这时候取回该文件的第一版，可以在调用co命令的时候指定那个版本号，如下所示：

```
$ co -r1.1 important.c
important.c,v  -->  important.c
revision 1.1
done
$
```

ci命令也有一个“-r”选项，它的作用是强制主版本号取一个给定的值，请看下面这条命令：

```
ci -r2 important.c
```

它将把important.c文件封存为2.1版。RCS和SCCS都使用“1”做为辅助版本号缺省使用的第一个数字。

5. rcsdiff命令

如果我们只是想了解一下两个修订版之间有哪些改动之处，就可以使用rcsdiff命令。

```
$ rcsdiff -r1.1 -r1.2 important.c
================================================================================
RCS file: important.c,v
retrieving revision 1.1
retrieving revision 1.2
diff -r1.1 -r1.2
11a12
>     printf("This is an extra line added later\n");

$
```

它告诉我们在原来的第11行后面插入了一行语句。

6. 修订版本的确定

RCS系统可以在源文件里使用特殊的字符串（即宏定义）来帮助跟踪记录所做的改动。最常用的两个宏定义是“$RCSfile$”和“Id”。宏定义“$RCSfile$”将扩展为该文件的名字，而“Id”将扩展为一个指示修订版本的字符串。RCS支持的特殊字符串的完整清单可以在其使用手册页里找到。这些宏定义会在解封并取出文件的某个修订版本时即时扩展，并且会在封存某个修订版本时自动刷新它们的取值。

我们对自己的文件做第3次修改，加上一些刚才介绍的宏定义。

```
$ co -l important.c
important.c,v  -->  important.c
revision 1.2 (locked)
done
$
```

对解封的important.c文件进行编辑，最终成为如下所示的样子：

```
/*
```

```
  This is an important file for managing this project.
  It implements the canonical "Hello World" program.

  Filename: $RCSfile$
*/

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

static char *RCSinfo = "$Id$";

int main() {
    printf("Hello World\n");
    printf("This is an extra line added later\n");
    printf("This file is under RCS control. It's ID is\n%s\n", RCSinfo);
    exit (EXIT_SUCCESS);
}
```

封存这个修订版本，看看RCS是如何管理这些个特殊字符串的。

```
$ ci important.c
important.c,v  <--  important.c
new revision: 1.3; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>> Added $RCSfile$ and $Id$ strings
>> .
done
$
```

此时，查看子目录清单将只能看到RCS文件在里面：

```
$ ls -l
-r--r--r--   1 rick     users         907 Feb 11 16:55 important.c,v
$
```

如果我们（用co命令）解封并取出源文件的当前版本进行检查，就会发现宏定义已经被扩展了，如下所示：

```
/*
  This is an important file for managing this project.
  It implements the canonical "Hello World" program.

  Filename: $RCSfile: important.c,v $
*/

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

static char *RCSinfo = "$Id: important.c,v 1.3 1999/02/11 16:55:04 rick Exp $";

int main() {
    printf("Hello World\n");
    printf("This is an extra line added later\n");
    printf("This file is under RCS control. It's ID is\n%s\n", RCSinfo);
    exit (EXIT_SUCCESS);
}
```

动手试试：GNU的make命令和RCS系统

GNU的make命令已经内建有一些对RCS文件进行管理的规则。我们下面来看看make命令对少了一个源文件的情况是如何处理的。

```
$ rm -f important.c
```

```
$ make important
co  important.c,v important.c
important.c,v  ->  important.c
revision 1.3
done
cc    -c important.c -o important.o
cc   important.o   -o important
rm important.o important.c
$
```

操作注释：

make有这样一条缺省规则：如果当前子目录里有一个“.c”后缀名的文件，在对它进行编译的时候就不再需要从RCS里解封并取出同名的文件。而第二条缺省规则则允许make通过RCS系统从important.c,v里创建出important.c来。既然没有名为important.c的文件存在着，就需要make创建该“.c”文件，而它会用co命令解封并取出其最新版本。在编译完成后，它还会删除important.c文件，把现场清理干净。

7. ident命令

我们可以通过ident命令找出那个包含着一个“Id”字符串的文件版本。因为我们是把字符串保存到一个变量里去的，所以它也会出现在可执行的结果文件里。有时候，如果我们在程序里加上了特殊的字符串却又没有在代码里访问过它们，有的编译器就会把它们优化掉。这个问题一般这样来解决：在代码里增加一些对这些字符串的“假”访问；但随着编译器越来越好，这种做法也越来越难了！

动手试试：ident命令

```
$ important
Hello World
This is an extra line added later
This file is under RCS control. It's ID is
$Id: important.c,v 1.3 1999/02/11 16:55:04 rick Exp $
$ ident important
important:
     $Id: important.c,v 1.3 1999/02/11 16:55:04 rick Exp $
$
```

操作注释：

通过执行程序，我们看到字符串确实已经结合到可执行代码里去了。接着，我们用ident命令从可执行文件里提取出了“Id”字符串。

RCS系统和出现在可执行文件里的“Id”字符串的这种用法可以在用户报告软件出现了问题时帮助我们确定它是文件的哪一个版本。我们可以把RCS（或SCCS）系统用做项目调试工具的组成部分，让它来记录汇报上来的问题和问题的补救措施。如果读者做的是软件销售工作，或者哪怕是赠送软件，了解不同版本之间的改动情况也是很重要的。

如果读者还想进一步了解这方面的资料，那除了标准的RCS章节以外，使用手册中的rcsinfo章节还给出了更多RCS系统方面的介绍。ci、co等命令也有它们各自的使用手册页。

8.3.2 SCCS系统

SCCS系统所提供的功能与RCS很相似。SCCS系统的优势在于它已经在X/Open技术规范里

得到了认可，因此，一切正规的UNIX版本都应该支持它。但比较现实的情况是：RCS的可移植性非常好，并且可以自由发行。因此，如果读者有一个UNIX系列的系统，不管它是否符合X/Open技术规范，就总是可以设法弄到并为它安装上RCS系统。基于这个原因，我们就不在这里对SCCS系统做进一步讲解了。我们只把这两个系统各自使用的命令在下表里做一些比较，这部分内容是为那些打算切换使用这两个系统的人们准备的。

RCS系统和SCCS系统的比较

对两个系统各自使用的命令提供比较是困难的，所以下面那个表格只能做为一个快速指南性质的东西。这里列出的命令在完成同一项工作时并不一定会采用同样的选项。如果读者不得不使用SCCS系统，就必须自己去核对正确的选项是哪些，但至少你现在知道应该从哪里开始去查找了（见表8-2）。

表 8-2

RCS系统	SCCS系统
rcs	admin
ci	delta
co	get
rcsdiff	sccsdiff
ident	what

除上面列出的那些命令以外，SCCS系统中的sccs命令与RCS系统中的rcs和co命令还有一些交叉点。比如说，“sccs edit”和“sccs create”就分别相当于“co -l”和“rcs -i”。

8.3.3 CVS系统

用RCS系统来管理文件中的改动情况固然不错，但CVS（Concurrent Versions System，共发版本系统）系统也完全能够胜任这一工作。与RCS系统相比，CVS系统有一个明显的优点，也许正是这个原因使它变得越来越流行：人们能够在因特网上使用CVS，不象RCS那样只能用在本地的共享子目录里。CCVS还支持并行开发技术，即多名程序员能够同时用同一个文件开展工作，而RCS任一时间只允许一个用户使用一个文件进行工作。CVS系统的命令与RCS系统的很相似，因为CVS最初就是做为RCS的一个操作前端而开发出来的。

CVS能够以灵活的方式跨网络运作，如果软件开发人员之间的网络联系只能通过因特网的话，它就是当之无愧的候选。许多Linux和GNU项目利用CVS来帮助程序员协调他们各自的工作进度。在正常情况下，通过CVS对远端文件进行操作与用它处理本地文件并没有太大的区别。

在这一章里，我们将简单地介绍一下CVS系统的基本原理，希望大家在两方面有所收获：一是能够开始对本地文件进行开发管理；二是当CVS服务器位于因特网上的时候，知道这样才能获得项目最新源文件的拷贝。详细资料请参考CVS的使用手册，该手册由Per Cederqvist撰写，版权属于Signum Support AB公司，但它却是在GNU一般公众许可证条款的约定下发行的，在许多Web站点上都可以找到，在那些站点上还可以找到FAQ（常见问题答疑）文件和各种其他的帮助文件。

CVS系统使用入门

我们首先要创建一个文件库，CVS系统把它的控制文件和被管理文件的主拷贝保存在这个文件库里。文件库的结构呈树状，所以用户不仅能够把一个项目的子目录结构整个保存在一个文件库里，还可以在同一个文件库里保存许多个项目。用彼此独立的文件库来保存彼此没有联系的项目当然也是可以的。我们将在下面看到如何告诉CVS系统我们打算使用的文件库是哪一个。

(1) CVS的本地使用方法

我们从创建一个文件库开始入手。为了便于讲解，我们使用了一个本地的文件库；又因为我们将只使用这一个文件库，所以我们把它放在/usr/local子目录下。在大多数Linux发行版本上，一切普通用户都是users分组的成员，所以我们把文件库的分组情况也设置为users，这样所有用户就都能够访问它了。

以超级用户身份执行下面的操作，为文件库建立一个子目录：

```
# mkdir /usr/local/repository
# chgrp users /usr/local/repository
```

恢复为普通用户的身份，把它初始化为一个CVS文件库。执行这一操作需要用户有子目录/usr/local/repository的写权限。如下所示：

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

“-d”选项告诉CVS我们想在哪儿建立这个文件库。

文件库建立好以后，我们就可以把项目的初始版本保存到CVS系统里去了。在做这项工作的时候有个小技巧可以让我们少打一些字。cvs命令在查找CVS子目录的时候可以使用两个办法：一是给命令行加上“-d <path>”选项（就象我们刚才使用init命令时那样）；如果没有给出“-d”选项，它就会去查看环境变量CVSROOT。我们不想反复输入“-d”选项，所以我们采用后一个办法：对CVSROOT环境变量进行设置。如果读者使用的shell是bash，这条命令就该是如下所示的样子：

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

先把路径切换到软件项目所在的子目录，然后让cvs把所有文件都导入到这个子目录里来。

```
$ cd cvs-sp
$ cvs import -m"Initial version of Simple Project" wrox/chap8-cvs wrox start
```

这两条命令告诉CVS要导入当前子目录里的所有文件，同时还给它加上了一条记录消息。

选项“wrox/chap8-cvs”的作用是告诉CVS需要把这个新项目保存到哪里去，这是一个相对于CVS树根的路径。别忘了只要我们愿意，就可以让CVS在一个文件库里保存多个项目。选项“wrox”的作用类似于厂家的名牌，它标识着被导入文件的初始版本是由谁提供的；选项“start”的作用是充当发行号标签。发行号标签可以用来以组为单位标识多个文件——比如构成一个软件某个特定发行版的那些文件。CVS响应出以下内容：

```
N wrox/chap8-cvs/Makefile
N wrox/chap8-cvs/hello.c

No conflicts created by this import
```

告诉我们它正确地导入了两个文件。

现在来看看我们能否从CVS系统里检索出我们的文件。先建立一个名为“junk”的子目录，然后解封并取出我们的文件，看看是否一切顺利。

```
$ mkdir junk
$ cd junk
$ cvs checkout wrox/chap8-cvs
```

我们提供给CVS的路径名与我们封存文件时使用的一样。CVS在当前子目录里创建了一个wrox/chap8-cvs子目录，并把文件放到了里面。

现在可以对我们的项目做一些改动了。让我们在hello.c文件里做点儿小修改。我们在文件里添上下面这一行：

```
printf("Have a nice day\n");
```

重新编译并运行程序以保证一切顺利。

我们可以查问CVS项目里出现了哪些改动。不必告诉CVS我们想对哪个文件进行检查，它能够一次完成对整个子目录的检查。

```
$ cvs diff
```

CVS响应出以下内容：

```
cvs diff: Diffing .
Index: hello.c
===================================================================
RCS file: /usr/local/repository/wrox/chap8-cvs/hello.c,v
retrieving revision 1.1.1.1
diff -r1.1.1.1 hello.c
6c6
<
---
>   printf("Have a nice day\n");
```

我们对自己所做的改动很满意，决定把它提交给CVS。

当我们把改动提交给CVS系统的时候，它会启动一个编辑器让我们输入一个记录消息。在执行commit命令之前，我们可以把环境变量CVSEDITOR强制性地设置为自己准备使用的编辑器。

```
$ cvs commit
```

CVS的响应将告诉我们它正在检查什么东西：

```
Checking in hello.c;
/usr/local/repository/wrox/chap8-cvs/hello.c,v  <--  hello.c
new revision: 1.2; previous revision: 1.1
done
```

把新改动封存起来之后，我们就能向CVS系统查问该项目已经出现了哪些改动。我们查问的是项目wrox/chap8-cvs自修订版1.1（即初始版本）以来的所有改动。

```
$ cvs rdiff -r1.1 wrox/chap8-cvs
```

CVS给出如下所示的详细报告：

```
cvs rdiff: Diffing wrox/chap8-cvs
```

```
Index: wrox/chap8-cvs/hello.c
diff -c wrox/chap8-cvs/hello.c:1.1 wrox/chap8-cvs/hello.c:1.2
*** wrox/chap8-cvs/hello.c:1.1  Tue Aug  3 19:54:14 1999
--- wrox/chap8-cvs/hello.c      Tue Aug  3 19:57:19 1999
***************
*** 3,7 ****
--- 3,8 ----
  main(int argc, char* argv[]) {

    printf("Hello World\n");
+   printf("Have a nice day\n");

  }
```

假设用户在CVS系统以外的一个本地子目录里还有一份代码的拷贝，其中有些文件虽然用户本人没有编辑，但已经被其他人通过CVS修改过了。现在用户想刷新这个本地子目录里的文件。CVS系统的update命令能够替用户完成这一工作。移动到项目路径的上一级，在本例中就是包含着wrox的那个子目录，然后执行下面的命令：

```
$ cvs update -Pd wrox/chap8-cvs
```

CVS将开始刷新有关的文件：它把其他人通过CVS系统修改过的文件从文件库里提取出来，再放到用户的本地子目录里去。其他人做的修改当然有可能与你做的修改“撞车”，但这个问题要靠你自己去解决。CVS是好东西，可它并不会变魔术呀！

讲到这里，大家应该看出CVS的用法和RCS的用法其实是很接近的。但它们两者之间有一个我们尚未论及的重要区别，CVS具备跨网络操作的能力，并且不需要事先对文件系统进行挂装。

(2) CVS的网络使用方法

我们前面已经介绍过，在向CVS系统提供文件库的存放地点时既可以使用命令行上的“-d”选项，也可以对环境变量CVSROOT进行设置。如果想跨网络操作，就需要使用这个参数更高级的语法。给大家举个例子，在编写GNOME（英文“GNU Network Object Model Enviroment”的字头缩写，意思是GNU网络对象模型环境，它是一个比较流行的开放源代码图形化桌面系统）的时候，其开发源代码就能用CVS系统在因特网上查到。只要在CVS文件库的路径名前添上正确的网络信息，CVS系统就能找到正确的CVS文件库，需要用户做的事情只是多敲几个字符那么简单。

接着刚才的GNOME例子往下说，如果读者把自己机器中的环境变量CVSROOT设置为“pserver:anonymous@anoncvs.gnome.org:/cvs/gnoms”，就可以把自己机器上的CVS系统指向GNOME源代码的CVS文件库。这个设置告诉CVS系统：该文件库要求对口令字进行核查（pserver），它位于服务器anoncvs.gnome.org上。

在访问源代码之前，我们必须先进行登录，如下所示：

```
$ cvs login
```

在提示输入口令字的时候直接按下回车键。

现在就可以使用各种cvs命令了，命令的用法和我们对本地文件库进行操作时的情况差不多，只有一个小区别：必须给每个cvs命令加上“-z3”选项以强制执行数据压缩，这是为了节约网络的带宽。

假设我们想取回ORBit的源代码，相应的命令是：

```
$ cvs -z3 checkout ORBit
```

如果我们想把自己的文件库设置为也能通过网络来访问的情况，就必须在我们自己的机器上启动一个CVS服务器。启动服务器的工作可以通过inetd命令来完成，我们只需在/etc/inetd.conf文件里加上如下所示的一行语句，再重新启动inetd就行了。这条语句是：

```
2401 stream tcp nowait root /usr/bin/cvs cvs -b /usr/bin --allow-root =
/usr/local/repository pserver
```

它的作用是指令inetd为连接到2401号端口的客户自动启动一个CVS任务，这个端口是CVS服务器的标准端口。通过inetd启动网络服务的详细资料请参考inetd和inetd.conf的使用手册页。

这一小节简短的篇幅使我们只能肤浅地涉及到CVS系统强大功能的一点皮毛。如果读者真的需要使用CVS系统，我们强烈建议你先设置一个本地文件库多加实践，读懂CVS系统庞杂的文档，并预祝你成功！记住，CVS的源代码是开放的，所以当你实在搞不懂代码的作用和目的，或者（虽然不太可能，但确实有可能！）认为自己发现了一只“臭虫”（程序缺陷或漏洞）的时候，你总是可以弄到并亲自分析其源代码的。

CVS还有其他一些优势——cvs客户程序不仅有一个能够在微软Windows下使用的版本，还为自己没有Linux机器的人们提供了一个Java客户版本，这使它的可移植性更加广泛了。

8.4 编写使用手册

在编写一个新命令的时候，必须把为它编写使用手册页的工作当做整个开发过程的任务之一。大家可能都已经注意到了，大部分的使用手册页其排版格式都很相似，它们基本上都是由以下几部分组成的：

- Header（标题）。
- Name（名称）。
- Synopsis（语法格式）。
- Description（说明）。
- Options（选项）。
- Files（有关文件）。
- See also（其他参考）。
- “Bugs”（已知程序漏洞）。

无关部分可以不出现在使用手册里。Linux程序的使用手册里还经常会多出一个“Author”（作者）部分。

UNIX下的使用手册是用一个名为nroff的工具软件排的版，而大多数Linux系统使用的是GNU项目的对等工具groff。这两个工具软件都是在早期roff排版命令的基础上开发出来的。nroff和groff的输入都是纯文本，此外，第一眼看去，它们的语法都很晦涩难懂。别紧张！在UNIX系统里，编写新程序最简单的办法是以一个现有的程序做出发点，再加上一些变化和改进，对使用手册来说正好可以照猫画虎。

对groff（或nroff）命令所使用的各种选项、命令和宏定义细节的讨论超出了这本书的学习范围。我们在这里只提供一个简单的模版，读者可以借鉴和参考它写出自己的使用手册来。

下面是我们程序的使用手册页源代码，它算是比较简单的了：

```
.TH MYAPP 1

.SH NAME
Myapp \- A simple demonstration application that does very little.

.SH SYNOPSIS
.B myapp
[\-option ...]

.SH DESCRIPTION
.PP
\fImyapp\fP is a complete application that does nothing useful.

.PP
It was written for demonstration purposes.

.SH OPTIONS
.PP
It doesn't have any, but let's pretend, to make this template complete:

.TP
.BI \-option
If there was an option, it would not be -option.

.SH RESOURCES
.PP
myapp uses almost no resources.

.SH DIAGNOSTICS
The program shouldn't output anything, so if you find it doing so there's
probably something wrong. The return value is zero.

.SH SEE ALSO
The only other program we know with this this little functionality is the
ubiquitous hello world application.

.SH COPYRIGHT
myapp is Copyright (c) 1999 Wrox Press.

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; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA.

.SH BUGS
There probably are some, but we don't know what they are yet.

.SH AUTHORS
Neil Matthew and Rick Stones
```

正如大家所看到的，宏定义所在的文本行都要以一个英文句号（.）打头，它们的名字习惯上都采用简略的缩写形式。第一行尾部的数字“1”表示这是使用手册的第一页，而第一页一般是对命令用法的介绍。既然命令就出现在第一页上，我们对使用手册源代码的介绍也就由此展开。

读者自己的使用手册页应该以此为蓝本并根据具体情况再加以修改，必要时可以参考其他

命令的使用手册页源代码。建议大家认真学习一下“Linux Man Page mini-HowTo”（Linux使用手册页编写指南），作者是Jens Schweikhardt，收录在“Linux Documentation Project”（Linux文档项目）的压缩档案里。

写好使用手册页的源代码之后，我们就可以用groff命令对它进行处理了。groff命令可以生成ASCII文本（-Tascii）或PostSCript（-Tps）两种输出结果。我们用“-man”选项告诉groff命令这将是一个使用手册页，这会让groff加载上使用手册页专用的宏定义。

```
$ groff  -Tascii  -man  myapp.1
```

这个命令将给出如下所示的输出：

```
MYAPP(1)                                                    MYAPP(1)

NAME
       Myapp  - A simple demonstration application that does very
       little.


SYNOPSIS
       myapp [-option ...]


DESCRIPTION
       myapp is a complete application that does nothing  useful.


       It was written for demonstration purposes.


OPTIONS
       It  doesn't  have any, but let's pretend, to make this tem-
       plate complete:


       -option
              If there was an option, it would not be -option.


RESOURCES
       myapp uses almost no resources.


DIAGNOSTICS
       The program shouldn't output anything, so if you  find  it
       doing  so  there's  probably  something  wrong. The return
       value is zero.


SEE ALSO
       The only other program we know with this this little func-
       tionality is the ubiquitous hello world application.


COPYRIGHT
       myapp is Copyright (c) 1999 Wrox Press.

       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;
       either version 2 of the License, or (at your  option)  any
       later version.

       This  program  is  distributed in the hope that it will be
       useful, but WITHOUT ANY WARRANTY; without even the implied
       warranty  of  MERCHANTABILITY  or FITNESS FOR A PARTICULAR
       PURPOSE.  See the GNU  General  Public  License  for  more
```

```
        details.

                                                                    1

MYAPP(1)                                                    MYAPP(1)

        You  should have received a copy of the GNU General Public
        License along with this program; if not, write to the Free
        Software  Foundation,  Inc.,  59  Temple Place - Suite 330
        Boston, MA 02111-1307, USA

BUGS
        There probably are some, but we don't know what  they  are
        yet.

AUTHORS
        Neil Matthew and Rick Stones
```

对手册页的测试完成之后，我们还需要把它安装起来。显示使用手册页的man命令通过环境变量MANPATH来搜索使用手册页。我们可以把这个新使用手册页放到一个本地使用手册页专用的子目录里，也可以把它直接放到系统的/usr/man/man1子目录里去。

当有人第一次要求查阅使用手册页的时候，man命令会自动对它进行排版并把排版结果显示出来。有些版本的man命令可以自动生成并保存一份（还很可能经过压缩）使用手册页的预排版ASCII文本；这样，当再次出现查阅同一页的操作请求时，可以加快对它进行处理的速度。

8.5 软件的发行传播

程序在发行或传播时所面临的最主要的问题是需要保证所有必要的文件都已经包括在其中了，并且它们都属于这个版本，这两方面可出不得差错。幸好为因特网设计程序的团体早就构造出一整套健壮的解决方案了，这两方面的问题都不在话下。这些方法包括：

- 把全部组件文件都打包到一个软件包文件里，这一方法所使用的工具是各种UNIX系统上都会有的。
- 加强软件包的版本编号工作。
- 给软件包文件起名字时要加上与之对应的版本号，方便使用者辨认他们正与之打交道是哪一个版本。
- 在软件包里使用多层目录，从软件包文件里释放文件的时候确保能够把它们放到该去的子目录里，软件包里应该有什么不应该有什么将一目了然。

这些解决方案的开发意味着程序的发行传播工作能够轻松并且可靠地得到完成。简化软件安装过程的事情与发行工作并没有太大的联系，因为那要由软件本身和准备安装该软件的计算机系统来决定。但至少我们可以保证软件包里的所有组件都是正确的。

8.5.1 patch程序

在软件发行过后，用户发现了程序漏洞或作者又开发出增强版或升级版的情况几乎是不可避免的。如果作者以二进制文件的形式来发行程序，他们通常会简单地发行新的二进制文件。

有时候（还相当经常），软件开发者会简单地宣布程序又推出了一个新的版本，对具体修订情况和做了哪些改动的细节资料则一笔带过。

从另一方面看，以源代码的形式来发行软件的做法是最好的，因为这样可以让人们清楚地看到软件的目的是如何达到的，具体功能又是如何运用的。这样还能让人们得以准确地查明软件的功用，实现对源代码各部分的再利用（前提当然是他们遵守有关的许可证条款）。

但这种做法也有它不利的一面，就拿Linux内核的源代码来说吧，它在经过数据压缩之后仍然多达13MB（很粗略的估计），所以即使我们想以源代码的形式升级这个内核，包装运输方面将要消耗的资源也难以估算；而事实则是各版本之间只有很少比例的源代码会发生变化。

幸好我们已经有解决这一问题的工具了：它就是我们马上就要介绍的patch程序——给文件打补丁的好工具。它是由Larry Wall编写的，他也是程序设计语言Perl的作者。patch命令允许软件的作者只推出“不同之处”。这样，无论是谁，只要他拥有某个文件的第1版和第1版与第2版的“不同之处”，就可以通过patch命令生成完整的第2版。

下面是某个文件的第1版：

```
This is file one
line 2
line 3
there is no line 4, this is line 5
line 6
```

下面是该文件的第2版：

```
This is file two
line 2
line 3
line 4
line 5
line 6
a new line 8
```

用diff命令比较这两个版本，把它们不一致的地方列成清单：

```
$ diff file1.c file2.c > diffs
```

diffs文件里包含着如下所示的内容：

```
1c1
< This is file one
---
> This is file two
4c4,5
< there is no line 4, this is line 5
---
> line 4
> line 5
5a7
> a new line 8
```

这实际上是一组编辑器命令，它们的作用是把一个文件修改为另一个文件。假设我们已经拥有file1.c和diffs文件，就可以用patch命令升级我们的文件，如下所示：

```
$ patch file1.c diffs
Hmm...  Looks like a normal diff to me...
Patching file file1.c using Plan A...
Hunk #1 succeeded at 1.
```

```
Hunk #2 succeeded at 4.
Hunk #3 succeeded at 7.
done
$
```

现在，patch命令把file1.c修改得与file2.c一模一样。

patch命令还有另外一项“绝技”，那就是“去掉补丁”。假设我们不喜欢刚才的修改，想恢复出原来的file1.c文件（是不是挺少见的？）。没问题，再patch一回就行，只不过这一次要加上“-R”（逆向补丁）选项。如下所示：

```
$ patch -R file1.c diffs
Hmm ...  Looks like a normal diff to me...
Patching file file1.c using Plan A...
Hunk #1 succeeded at 1.
Hunk #2 succeeded at 4.
Hunk #3 succeeded at 6.
done
$
```

file1.c又恢复成它最初的样子。

patch命令还有几个其他的选项。但一般说来它是很聪明的，能够根据其自身的输入情况推测出用户想做些什么，然后简单地来个“走自己的路”。如果patch意外地失了手，打补丁失败了，它会创建出一个带“.rej”后缀名的文件，其内容是它无法打上的文件补丁。

在对软件补丁进行处理的时候，给diff命令加上一个“-c”选项是很好的办法。这个选项的作用是产生一个“上下文不同之处”——即把每处修改及其前后几行提取出来，这可以让patch命令在打补丁之前确定上下文是匹配的，而补丁本身也更容易让人读懂了。

> 如果你在某个程序里发现了一个漏洞并进行了修补，给程序的作者发送一个补丁要比只给出一个对漏洞的描述要更容易，更准确，也更礼貌。

8.5.2 软件发行方面的其他工具

UNIX程序和源代码通常都打包在一个文件里发行，这个文件的名字里包含着版本号，并且带有后缀名“.tar.gz”或“.tgz”。这是一些用gzip命令压缩过的TAR（磁带文档）文件。如果你使用的是普通的tar命令，对这些文件的处理就必须分为两个步骤。先为我们的软件创建一个gzip文件。如下所示：

```
$ tar cvf myapp-1.0.tar main.c 2.c 3.c *.h myapp.1 Makefile6
main.c
2.c
3.c
a.h
b.h
c.h
myapp.1
Makefile6
$
```

我们现在有了一个TAR文件，请看：

```
$ ls -l *.tar
```

```
-rw-r--r--   1 rick     users       10240 Feb 11 18:30 myapp-1.0.tar
$
```

我们再用gzip程序对它进行数据压缩，缩短其长度。如下所示：

```
$ gzip myapp-1.0.tar
$ ls -l *.gz
-rw-r--r--   1 rick     users        1580 Feb 11 18:30 myapp-1.0.tar.gz
$
```

文件长度的压缩幅度给人以深刻印象。然后，我们把“.tar.gz”后缀名改为一个简单点的“.tgz”后缀名。具体命令如下所示：

```
$ mv myapp-1.0.tar.gz myapp_v1.tgz
```

这种以一个句号和三个字符结尾的文件命名方式有点儿象MS-DOS和某些MS-Windows软件的做法，它们与UNIX的重大区别之一就是前者对后缀名正确与否的依赖性非常强。为了取回我们的文件，我们需要先执行解压缩操作，再从tar文件里把它们释放出来。如下所示：

```
$ mv myapp_v1.tgz myapp-1.0.tar.gz
$ gzip -d myapp-1.0.tar.gz
$ tar xvf myapp-1.0.tar
main.c
2.c
3.c
a.h
b.h
c.h
myapp.1
Makefile6
$
```

GNU版本的tar命令更方便一些，它只用一个操作就能创建出压缩文档来。如下所示：

```
$ tar zcvf myapp_v1.tgz main.c 2.c 3.c *.h myapp.1 Makefile6
main.c
2.c
3.c
a.h
b.h
c.h
myapp.1
Makefile6
$
```

解压缩操作也比较简单，如下所示：

```
$ tar zxvf myapp_v1.tgz
main.c
2.c
3.c
a.h
b.h
c.h
myapp.1
Makefile6
$
```

tar命令

在上面的例子里我们使用了tar命令，但对它选项的介绍只限于我们曾经用过的那些。在这里，我们将对这个命令和它的几个常用选项做一个简单的说明。从例子里我们可以看出，它的

基本语法是：

```
tar [ options ] [ list of files ]
```

文件清单里的第一项是该命令的操作目标，虽然我们一直是对文件进行的操作，但它也可以是一个设备（tar的意思是“tape archive”，磁带文档）。文件清单里的其他项将被添加到新文档或现有文档里去，具体操作要视命令选项的设置情况而定。文件清单可以包含子目录，此时的缺省操作是把所有的下级子目录也添加到文档里。如果执行的是文件释放操作，就没有必要给出文件的名字，因为tar命令能够在对文件进行归档时保留其完整的路径。

在这一小节里，我们使用了下面六个不同选项的组合：

c 创建一个新档案。

f 指定目标是一个文件而非一个设备。

t 列出档案的内容，但并没有把它们实际提取出来。

v 操作步骤说明状态：随程序的执行显示信息。

x 从档案中提取文件。

z 从GNU的tar命令里用gzip对档案进行过滤。

tar命令还有许多其他的选项，这些选项允许对命令的操作情况和它将要创建的文档进行细致的调控。详细资料请参考tar命令的使用手册页。

8.6 本章总结

在这一章里，我们向大家介绍了一些UNIX开发工具，它们使程序的设计开发和发行传播工作更容易管理。最先介绍的make命令和制作文件可以用来管理多个源文件，这可以说是本章最重要的内容。然后，我们学习了源代码控制方面的知识，它可以让我们在代码开发过程中对各种修改做跟踪记录。最后，我们学习了patch、tar和gzip等命令的使用方法，它们在程序的发行和升级方面用处很大。

第9章　调试与纠错

不管多么出色的软件代码也会有小缺陷，一般说来，每100行代码就会有2到3个缺陷。这些错误将导致程序或函数库无法完成预定的功能，造成实际执行情况与预期执行情况的不一致。在软件的开发过程中，查找和改正这些缺陷将耗费程序员大量的时间。

在这一章里，我们将对软件的缺陷进行研究，并介绍一些查排程序特定错误行为的工具和技巧。这与程序测试（以各种可能出现的条件检验程序操作情况的工作）是不一样的，虽然测试和纠错是密切相关的，并且许多错误正是在测试阶段被发现的。

9.1　错误的分类

造成程序缺陷的原因并不是很多，并且它们都可以通过适当的办法进行查找和纠正。

9.1.1　功能定义错误

如果程序的功能没有被正确定义，那它肯定不能完成预定的工作。在这种情况下，即使是世界上最优秀的程序员写出来的程序也是错误的。在开始程序设计（规划阶段）之前，我们必须确保自己知道和理解要用这个程序干些什么。通过认真研究程序设计要求并加强与该程序使用者之间的沟通能够纠正许多（即使不是全部）功能定义方面的漏洞。

9.1.2　设计规划错误

程序的规模有大有小。在计算机键盘前面坐下来，把源代码直接敲进去，然后程序一次通过，这种机会虽不能说没有，但确实很少很难遇到。一定要多花点时间对程序的流程和结构进行分析，看看都需要用到哪些数据结构。尽量把细节问题提前确定下来，这样就能节省许多反复改写程序的时间。

9.1.3　代码编写错误

当然，每个人都有他最容易犯的错误。根据程序规划来创建源代码的过程并不是一个不会出错的完美过程。大部分程序缺陷都是在这一阶段里产生的。根据以往的经验，当你的程序里隐藏着缺陷的时候，重新阅读程序或者与其他人进行探讨，这个办法虽然因为过于简单而往往被人忽视，但它的确很有效。你肯定会对自己通过与他人探讨程序的具体实现而能找出的漏洞之多感到惊讶。

> 像C这样带有编译器的程序设计语言有这样一个优点，那就是语法错误能够在编译阶段被检查出来，而解释型程序设计语言，比如UNIX中的shell，就只能在你运行程序

的时候才能检查出语法错误。如果问题出在程序本身的错误处理代码部分里，要想通过程序测试来找出它们可就太不容易了。

可以试着在纸上执行程序的核心部分，这个过程通常被称为“干运行”。对那些重要的例程，应该记下它们的输入值，然后一步一步地手工计算出输出结果来。对程序进行调试并不见得非用计算机不可，有时候，问题可能正是因为计算机本身的原因才出现的。即使那些编写函数库、编译器和操作系统的人们也会出错！可话又说回来，也不要一出现问题就抱怨是工具的问题，新程序里出现缺陷的概率要比编译器大的多。

9.2 常用调试技巧

对典型UNIX程序的调试和测试工作来说，现在已经有了几种固定的办法。一般的做法是先运行程序看看会发生哪些事情。如果它根本就不工作，我们再来决定需要采取哪些措施。我们可以修改程序并重新进行尝试（代码检查-试运行-出错法），也可以在程序里增加一些语句以获得更多程序内部运行情况的资料（取样法），还可以直接监测程序的操作情况（受控执行法）。程序调试可以分为五个阶段，它们是：

- 测试：找出都有哪些缺陷和漏洞。
- 固化：让缺陷反复出现。
- 定位：确定应该对此负责的代码行。
- 改正：对代码进行修改。
- 复查：确定修改解决了问题。

9.2.1 一个有漏洞的程序

我们先来看一个带有缺陷和漏洞的示例程序。在本章的学习过程中，我们将对它进行调试纠错。这个程序是在某个大型软件系统的开发过程中编写出来的。它的作用是对一个名为sort的函数进行测试，该函数准备通过冒泡排序算法对一个类型为item的结构数组进行排序。数据项将以其成员之一的key为对象进行递增排序。程序对一个测试用数组调用sort以测试它的工作情况。在现实世界里，我们可能永远也不会使用这个算法，因为它的执行效率实在是太低了。我们之所以在这里使用这个算法，原因就在于它比较短小，相对来说简单易懂，但也更容易出现错误。事实上，标准的C语言库里已经有一个这样的函数了，它的名字是qsort。

这段代码读起来稍微有点儿费劲，里面没有注释，也不知道最初的程序员是哪一位。只能靠我们自己了，我们先从最基本的例程debug1.c入手。下面是它的程序清单：

```
/*   1  */  typedef struct {
/*   2  */      char *data;
/*   3  */      int key;
/*   4  */  } item;
/*   5  */
/*   6  */  item array[] = {
/*   7  */      {"bill", 3},
/*   8  */      {"neil", 4},
/*   9  */      {"john", 2},
/*  10  */      {"rick", 5},
```

```
/*  11  */      {"alex", 1},
/*  12  */  };
/*  13  */
/*  14  */  sort(a,n)
/*  15  */  item *a;
/*  16  */  {
/*  17  */      int i = 0, j = 0;
/*  18  */      int s = 1;
/*  19  */
/*  20  */      for(; i < n && s != 0; i++) {
/*  21  */              s = 0;
/*  22  */              for(j = 0; j < n; j++) {
/*  23  */                      if(a[j].key > a[j+1].key) {
/*  24  */                              item t = a[j];
/*  25  */                              a[j] = a[j+1];
/*  26  */                              a[j+1] = t;
/*  27  */                              s++;
/*  28  */                      }
/*  29  */              }
/*  30  */              n--;
/*  31  */      }
/*  32  */  }
/*  33  */
/*  34  */  main()
/*  35  */  {
/*  36  */      sort(array,5);
/*  37  */  }
```

我们来编译这个程序。

```
$ cc -o debug1 debug1.c
```

编译进行的很顺利，既没有报告出错也没有警告信息。

在我们运行这个程序之前，先在其中添加一些代码把它的结果打印出来。要不然我们连这个程序是否在工作都不会知道。我们额外增加一些语句把排序后的数组打印出来。我们把这个新版本称为debug2.c。新增加的代码如下所示：

```
/*  34  */  main()
/*  35  */  {
/*  36  */          int i;
/*  37  */          sort(array,5);
/*  38  */          for(i = 0; i < 5; i++)
/*  39  */                  printf("array[%d] = (%s, %d)\n",
/*  40  */                          i, array[i].data, array[i].key);
/*  41  */  }
```

这些额外的代码严格说来并不是程序功能所必需的。我们加上它们完全是出于测试工作的需要。我们必须谨慎从事，不能让添加的测试用代码又引入新的缺陷。现在，再次进行编译，然后运行程序：

```
$ cc -o debug2 debug2.c
$ ./debug2
```

这样做会发生什么样的事情与你使用的UNIX（或Linux）具体版本和该版本的具体配置有关，它在作者之一的系统上运行时会给出如下所示的输出：

```
array[0] = {john, 2}
array[1] = {alex, 1}
array[2] = {(null), -1}
array[3] = {bill, 3}
array[4] = {neil, 4}
```

但它在另一位作者的系统（运行着另外一种Linux内核）上运行时给出的输出却是下面这样的：

```
Segmentation fault
```

在读者的UNIX系统上，你可能会看到这两种输出情况之一。我们希望看到的结果是：

```
array[0] = {alex, 1}
array[1] = {john, 2}
array[2] = {bill, 3}
array[3] = {neil, 4}
array[4] = {rick, 5}
```

很明显，这段代码存在着重大隐患。即使它能一气呵成地运行到底，给出的排序结果也不正确；而如果它的运行因内存段冲突被强行终止了，就说明操作系统向程序发送了一个信号，说操作系统检测到了对无效内存地址的访问操作，为防止内存崩溃，半路终止了该程序的运行。

操作系统监测无效内存访问操作的能力取决于它的硬件配置和它在实现内存管理功能时所采取的某些具体做法。在大多数系统上，操作系统分配给这个程序的内存一般都会比实际使用所需要的大一些。如果无效内存访问操作发生在内存的这一部分，硬件是检测不到这些无效访问的。这就是为什么并非所有版本的UNIX都会报告出现内存段冲突的原因。

> 有的库函数（比如printf）在某些特殊情况下（比如正使用着一个空指针的时候）也会防止无效内存访问操作的发生。

在查找数组访问方面的错误时，增加数组元素的尺寸不失为一个好办法，因为这样可以增加出现错误的概率。如果我们只是在数组的字节尾读一个字符，可能会看不到有问题发生，因为分配给程序的内存一般都会延续到由操作系统设定的内存边界上去，边界一般以8K为单位逐步扩大。

如果我们增大了数组元素的尺寸，比如在这个例子里我们把item成员data扩大为一个4096个字符组成的数组，对不存在的数组元素进行访问时内存地址就有可能落在这个程序分配到的内存边界以外的地方。数组的每个元素有4K的长度，所以我们错用了的内存将落在边界以外0~4K的范围里。

我们给如此加工过的文件起名为debug3.c，它在两位作者的Linux系统上都出现了内存段冲突错误。下面是需要加到程序里去的语句和程序的执行情况：

```
/*   2  */      char data[4096];
```

```
$ cc -o debug3 debug3.c
$ ./debug3
Segmentation fault (core dumped)
```

但还是存在着这样的可能性，即某些个系列的Linux或UNIX操作系统依然不会产生内存段冲突错误。当C语言的ANSI标准把某种行为定义为“未定义”的时候，实际上它还是允许程序做些事情的。现在就遇到了这样的情况：我们写出了一个不合规范的C语言程序，而这个不合规范的C语言程序会表现出很奇怪的行为！我们将会看到，这个错误确实属于“未定义行为”的范畴。

9.2.2 代码审查

我们刚才已经提到过，当一个程序的运行情况与我们的预期不相吻合的时候，重新阅读这

个程序将是一个很好的做法。根据这一章的学习目的，我们将假设程序代码都已经通过了复查，那些比较明显的漏洞都已经被排除了。

> 代码审查这个术语还代表着一个更正规的过程：一组开发人员逐字逐句地推敲数百行代码。但代码行的多少并不重要，它还是代码审查，并且它仍然是一直有效的技巧。

有一些工具可以帮助你完成代码复查的工作，编译器就是其中比较明显的一个。如果程序中有语法错误，它就会发现并报告出来。

> 有些编译器还有用来对可疑行为产生报警的选项，这些行为包括没有对变量进行初始化、在条件判断里使用了赋值操作等。就拿GNU编译器来说吧，它在运行时可以使用下面这些选项：
>
> ```
> gcc -Wall -pedantic -ansi
> ```
>
> 这些选项将激活许多警告性检查和一些不常用的检查，它们将对程序是否符合C语言标准进行检验。我们建议大家养成使用这些选项的习惯，特别是“-Wall”选项。因它而产生的信息在追踪程序错误时非常有用。

我们将在稍后对lint和LClint等工具进行介绍。它们的作用与编译器差不多，都是对源代码进行分析并把可能不正确的代码报告出来。

9.2.3 取样法

取样法指的是在程序里添加一些代码的行为，这样做的目的是为了进一步收集与程序运行时的行为有关的信息。增加printf语句的做法是很常见的，我们在刚才的例子里就采用了这一措施，这样做的目的是为了打印出变量在程序运行不同阶段的值。这些额外添加的printf语句往往会有好几个。需要提醒大家的是：只要对程序进行了修改，这一过程的结局肯定是多一次编辑多一次编译；而且，在错误得到补救之后需要把这些额外的代码删除掉。

有两种取样法技巧可以在这里帮到我们。第一个办法是通过C语言的预处理器有选择地添上一些取样代码，这样我们只需重新编译程序就能达到添上或去掉调试代码的目的。这个办法实行起来很简便，一个下面这样的结构就能满足我们的要求：

```
#ifdef DEBUG
  printf("variable x has value = %d\n", x);
#endif
```

如果在对程序进行编译的时候加上了编译标志“-DDEBUG”，我们就能对“DEBUG”符号进行定义，添上额外的调试代码；如果不加上这个标志，就去掉之。我们还可以使用更复杂的数值调试宏定义，就象下面这样：

```
#define BASIC_DEBUG 1
#define EXTRA_DEBUG 2
#define SUPER_DEBUG 4

#if (DEBUG & EXTRA_DEBUG)
  printf...
#endif
```

此时，我们必须对DEBUG宏进行定义，但我们可以设置它来代表一组调试信息，或者代表一个调试控制层次。比如说，在我们的这个例子里，编译标志“-DDEBUG=5”代表激活BASIC_DEBUG和SUPER_DEBUG，但不包括EXTRA_DEBUG。标志“-DDEBUG=0”将禁止一切调试信息。另外一个办法是给程序添上下面这些语句，这样，当不再需要调试工作时就不必在命令行上定义DEBUG宏了。如下所示：

```
#ifndef DEBUG
#define DEBUG 0
#endif
```

C语言预处理器已经定义了几个对调试纠错有帮助作用的宏定义。这些宏定义在扩展时会提供当前编译操作的有关信息。如表9-1所示：

表 9-1

宏 定 义	说 明
__LINE__	一个代表当前行号的十进制常数
__FILE__	一个代表当前文件名的字符串
__DATE__	一个“Mmm dd yyyy”形式的字符串，当前日期
__TIME__	一个“hh:mm:ss”形式的字符串，当前时间

注意：这些宏定义前后各有两个下划线字符。预处理器的标准记号经常采用这种形式，用户应该避免选用可能会与它们“撞车”的记号。上面说明中的“当前”指的是预处理操作正在执行的那一时刻，也就是运行编译器对文件进行处理时的时间和日期。

动手试试：调试信息

请看下面的cinfo.c程序，如果在对它进行编译时开启了调试状态，它就会打印出编译时的日期和时间。下面是它的程序清单：

```
#include <stdio.h>

int main()
{
#ifdef DEBUG
    printf("Compiled: " __DATE__ " at " __TIME__ "\n");
    printf("This is line %d of file %s\n", __LINE__, __FILE__);
#endif
    printf("hello world\n");
    exit(0);
}
```

激活调试功能（用“-DDEBUG”选项）编译这个程序，我们将看到如下所示的编译信息：

```
$ cc -o cinfo -DDEBUG cinfo.c
$ ./cinfo
Compiled: Aug  8 1999 at 11:15:21
This is line 7 of file cinfo.c
hello world
$
```

操作注释：

C预处理器是编译器的组成部分之一，它对正在编译的当前行和当前文件进行跟踪记录。只

要它遇见记号“__LINE__”和“__FILE__”，就会替换这些变量（在编译过程里）的当前值。对编译日期和时间的处理也与此相同。“__DATE__”和“__TIME__”都是字符串，所以我们可以用printf的格式字符串把它们合并起来。ANSI C允许把紧挨着的多个字符串当做一个字符串来对待。

不需要重新编译的调试技巧

在继续学习新的内容之前，我们认为值得向大家介绍一个用printf函数帮助调试工作的技巧，这个办法用不着做“# ifdef DEBUG”这样的设置，也不象后者那样需要重新编译才能开始对程序的调试。

这个方法的要旨是在程序里增加一个被用做调试标志的全局变量和增加一个调试信息记录函数，这就使得用户能够在命令行上通过“-d”选项切换调试模式的开关状态，即使程序已经发行也还是可以这样做。现在我们可以把下面这些内容加到程序代码里去：

```
if (debug) {
sprintf(msg, ...)
write_debug(msg)
}
```

调试信息需要输出到标准错误输出stderr去；或者，如果因为程序的原因不能这样做，还可以使用syslog函数提供的系统日志功能。

如果用了这种增加调试信息的办法来解决程序开发过程中的问题，开发完成时可以把这些代码留在程序里。只要你比较谨慎在意，这样做将是相当安全的。它的好处体现在程序发行之后：如果用户遇到了麻烦，他们自己就能在运行这个程序的时候打开调试功能，替你完成诊断错误的工作。除了报告程序给出“Segmentation fault”消息以外，他们还能报告出当时程序在干什么事情，而不仅仅是报告用户本人正在干什么了。两者的差距是很明显的。

当然，这样做也有一个明显的不足之处，就是程序的长度会有所增加。但在大多数情况下，这个问题对程序的使用者来说是透明的，算不上是一个实际意义上的问题。程序的长度可能会增加20%或30%，但往往并不会对程序的执行性能造成真正的影响。一般来说，只有在程序长度以几何级数增长时才会造成程序性能降低的后果，仅仅增加几条语句是不会有太大问题的。

9.2.4 程序的受控执行

现在回到我们的示例程序上，现在可以肯定地说它有一个漏洞。我们对程序进行修改，增加一些代码把程序运行时的变量值打印出来；我们还可以用一个调试器（debugger）来控制程序的执行，随时查看程序的执行状态。

UNIX系统上有许多种调试器，提供调试器的厂商也很多。比较常见的有adb、sdb和dbx等。比较复杂的调试器能够让我们在源代码级对程序的状态进行检查。sdb、dbx和GNU的调试器gdb都能够做到这一点，Linux系统经常使用的是gdb。gdb还有一些对用户更加友好的“操作前端”，xxgdb、tgdb和DDD等就是一些这样的程序。Emacs编辑器也有一个类似的功能（即“gdb-mode”gdb调试模式），它允许用户在程序上运行gdb、设置断点、查看现在执行到哪一行源代码等。

为了能够对程序进行调试，需要在对它进行编译的时候加上一个或几个编译器操作选项。

这些选项的作用是让编译器把额外的调试信息添加到程序里。这些信息包括各种记号和源代码行号：调试器将利用这些信息通知我们程序已经执行到源代码的什么位置了。

"-g" 标志是对程序进行调试性编译时常用的选项。我们需要给每一个需要调试的源文件都加上这个选项，对链接器（linker）也要这样做，它将使用特殊版本的C语言标准库完成编译和链接操作，给库函数加上程序调试方面的支持。编译器会把这些标志自动传递给链接器。有些函数库在其自身被编译的时候并没有考虑到程序调试方面的需要，如果使用了这类的函数库，调试工作虽然还能够进行，但灵活性就要差些了。

调试信息将使可执行文件的长度成倍增加（最高可以达到10倍的水平）。但尽管可执行文件增大了（并且占用了更多的磁盘空间），程序运行所需要的内存还是和原来的一样。在程序的调试工作结束之后，最好还是把调试信息从程序的发行版本里删除掉，这将节省不少磁盘空间。

> 你可以用 "strip <file>" 命令在不重新编译的前提下删除可执行程序中的调试信息。

9.3 用gdb进行调试纠错

我们将使用GNU的调试器gdb对我们的程序进行调试。这个功能强大的调试器是一个自由软件，能够用在许多UNIX平台上。它也是Linux系统上缺省使用的调试器。gdb已经被移植到许多其他的计算机平台上了，并且能够用来对嵌入式实时系统进行调试。

9.3.1 启动gdb

现在对我们的示例程序做调试性编译并启动gdb，如下所示：

```
$ cc -g -o debug3 debug3.c
$ gdb debug3
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 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-redhat-linux"...
(gdb)
```

gdb有详细的在线帮助信息，它完整的使用手册由几个文件组成，这些文件可以用info程序或者在emacs里查阅。如下所示：

```
(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)
```

gdb本身是一个基于文本的软件，但它为一些重复性操作准备了一些快捷键。它的许多版本都具备带历史记录的命令行编辑功能，用户可以（用光标移动键）卷回以前输入过的命令再次执行之。所有版本都支持空白命令，即直接按下回车键将再次执行最近执行过的那条命令。用step或next命令逐行单步执行程序时就能体会到这种设置的好处了。

9.3.2 运行一个程序

我们可以用run命令来执行一个程序。在run命令中给出的任何参数都将被传递到程序去，它们将被用做那个程序的参数。我们的示例程序不需要参数。

在这里，我们必须假设读者的系统和两位作者的一样，都出现了内存段冲突错误。如果情况并非如此，请继续往下读。如果读者在自己编写的程序里遇见过内存段冲突错误，学完这一章后就该知道怎样才能解决它。如果读者没有遇到内存段冲突错误，但还想用这个示例程序继续本章的学习，可以跳到debug4.c程序，到那时我们已经把内存访问方面的第一个问题弥补好了。请看下面的操作：

```
(gdb) run
Starting program: /home/neil/debug3

Program received signal SIGSEGV, Segmentation fault.
0x8048488 in sort (a=0x8049684, n=5) at debug3.c:23
23      /*  23  */                      if(a[j].key > a[j+1].key) {

 (gdb)
```

程序运行依然不正确。程序运行失败的时候，gdb会报告出失败的原因和程序失败时所处的位置。下面我们来调查问题的根源。

根据读者的操作系统内核、C语言库和编译器版本的具体情况，你可能会看到冲突发生在一个稍微不同的地点，比如发生在交换数组元素的第25行而不是发生在比较数组元素关键字的第23行。如果是前一种情况，你应该看到象下面这样的信息：

```
Program received signal SIGSEGV, Segmentation fault.
0x8000613 in sort (a=0x8001764, n=5) at debug3.c:25
25      /*  25  */                                  a[j] = a[j+1];
```

两个问题的根源是一样的，你还是可以沿着我们的gdb操作示例继续学习。

9.3.3 堆栈跟踪

程序停在sort函数里，地点是源文件debug3.c的第23行。如果编译程序时没有（用cc命令的“-g”选项）添加调试信息，我们就无法看到程序失败时停在了什么地点，也不能用变量名检查数据。

用backtrace命令可以查出程序是如何到达这一位置的，如下所示：

```
(gdb) backtrace
#0  0x8048488 in sort (a=0x8049684, n=5) at debug3.c:23
```

```
#1  0x804859e in main () at debug3.c:37
#2  0x40031cb3 in __libc_start_main (main=0x804858c <main>, argc=1,
    argv=0xbffffd64, init=0x8048298 <_init>, fini=0x804863c <_fini>,
    rtld_fini=0x4000a350 <_dl_fini>, stack_end=0xbffffd5c)
    at ../sysdeps/generic/libc-start.c:78
(gdb)
```

这是一个很简单的程序，因为在其他函数里调用过的函数并不多，所以跟踪信息也很短。大家可以看到：sort是由main在debug3.c文件的第37行调用的，而它自己也在这个文件里。实际工作中遇到的问题往往要复杂得多，而backtrace将帮助我们找出程序到达错误地点所经过的路线。当调试从不同地方调用的函数时，这个命令将十分有用。

backtrace命令可以被简写为bt；并且，为了与其他调试器兼容，gdb还有一个具备同样功能的where命令。

9.3.4 对变量进行检查

gdb在停止程序运行时给出的信息和从跟踪堆栈得到的资料让我们能够看到函数的参数的取值。

sort函数在被调用时有一个参数a，它的取值是0x8049684。这是那个数组的地址。不同系统上的这个值通常是不一样的，这要视用户使用的编译器和操作系统而定。

错误出现在第23行，我们在那里对数组的两个元素进行了比较，即：

```
/*  23  */          if (a[j].key > a[j+1].key) {
```

调试器能够对函数参数的内容、局部变量和全局性数据等进行检查。print命令的作用就是给出变量或其他表达式的内容，如下所示：

```
(gdb) print j
$1 = 4
```

我们看到局部变量“j”的值是4。这就是说，程序曾经尝试执行这样一条语句：

```
if (a[4].key > a[4+1].key)
```

我们传递给sort函数的数组array只有五个元素，它们的下标是从0到4。因此，这个语句实际上是在读一个并不存在的数组元素“array[5]”。循环计数器变量“j”取了一个错误的值。

如果读者用这个例子做练习时程序停在了第25行，就说明你的系统是在准备交换数组元素时才检测到“读数组时越界”错误的，当时执行的语句是：

```
/*  25  */          a[j] = a[j+1];
```

当“j”取值为4时将造成下面这样的结果：

```
a[4] = a[4+1];
```

要想查看已经处理完的数组元素，需要在print命令里使用一个表达式。gdb允许我们使用几乎一切合法的C语言表达式来查看变量的取值、数组元素和指针。

```
(gdb) print a[3]
$2 = {data = "alex", '\000' <repeats 4091 times>, key = 1}
(gdb)
```

gdb把命令的结果保存在伪变量“$<number>”里。最近一次操作的结果永远是“$”，倒数第2次操作的结果是“$$”。这使我们可以把某次操作的结果用在另一个命令里。请看下面的例子：

```
(gdb) print j
$3 = 4
(gdb) print a[$-1].key
$4 = 1
```

9.3.5 列出程序清单

在gdb里我们可以用list命令列出程序的源代码清单来。这个命令会给出当前位置前后周围的一段代码；继续使用list命令将显示得更多。我们可以给list一个行号或者一个函数名做参数，而它就会把指定地点附近的代码列出来。

```
(gdb) list
18      /*  18  */      int s = 1;
19      /*  19  */
20      /*  20  */      for(; i < n && s != 0; i++) {
21      /*  21  */              s = 0;
22      /*  22  */              for(j = 0; j < n; j++) {
23      /*  23  */                      if(a[j].key > a[j+1].key) {
24      /*  24  */                              item t = a[j];
25      /*  25  */                              a[j] = a[j+1];
26      /*  26  */                              a[j+1] = t;
27      /*  27  */                              s++;
(gdb)
```

我们可以看到，在第22行上，循环被设置为“当变量j小于n时继续执行”。在我们的例子里，n等于5，所以j的最大取值是4，再大就退出循环了。当j取值为4的时候，参加比较的数组元素分别是a[4]和a[5]，而这就引起了内存段冲突。这个问题的解决办法之一是把终止循环的条件改正为“j < n-1”。

我们做出改正，给新程序起名为debug4.c，重新编译并运行它：

```
/*  22  */          for(j = 0; j < n-1; j++) {
```

```
$ cc -g -o debug4 debug4.c
$ ./debug4
array[0] = {john, 2}
array[1] = {alex, 1}
array[2] = {bill, 3}
array[3] = {neil, 4}
array[4] = {rick, 5}
```

程序工作还是不正常，因为它给出的数组排序结果不正确。我们用gdb对程序的运行做单步调试。

9.3.6 设置断点

为了找出程序到底失败在什么地点，我们需要能够查看它在运行时都干了哪些事情。我们可以通过设置断点在任一位置停止程序的运行。这将中断程序的运行，让它把控制返回给调试器。这样，我们就能对变量进行检查，然后让程序从断点位置继续执行。

sort函数里有两个循环。外层循环的循环计数变量是i，它对每个数组元素执行一次；内层循环的作用是交换数组中靠后的元素。总的效果是让比较小的元素象“气泡”一样“冒”到数组的顶部去。外层循环每执行一次，数组中最大的元素就会“下沉”一些。我们可以通过停止程

序运行并检查数组当时状态的办法来核实这一点。

有许多命令可以用来设置断点。用gdb的“help breakpoint”命令可以列出这些命令，如下所示：

```
(gdb) help breakpoint
Making program stop at certain points.

List of commands:

awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified line or function
catch -- Set breakpoints to catch exceptions that are raised
clear -- Clear breakpoint at specified line or function
commands -- Set commands to be executed when a breakpoint is hit
condition -- Specify breakpoint number N to break only if COND is true
delete -- Delete some breakpoints or auto-display expressions
disable -- Disable some breakpoints
enable -- Enable some breakpoints
hbreak -- Set a hardware assisted  breakpoint
ignore -- Set ignore-count of breakpoint number N to COUNT
rbreak -- Set a breakpoint for all functions matching REGEXP
rwatch -- Set a read watchpoint for an expression
tbreak -- Set a temporary breakpoint
thbreak -- Set a temporary hardware assisted breakpoint
watch -- Set a watchpoint for an expression

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

我们在第20行设置一个断点，然后运行这个程序，如下所示：

```
$ gdb debug4
(gdb) break 20
Breakpoint 1 at 0x80483f1: file debug4.c, line 20.
(gdb) run
Starting program: /home/neil/debug4

Breakpoint 1, sort (a=0x8049684, n=5) at debug4.c:20
20      /*  20  */      for(; i < n && s != 0; i++) {
```

我们可以打印出数组元素的值，然后用cont命令让程序继续执行。程序会在运行到下一个断点位置时再次停下来，在我们的例子里就是它再次运行到第20行。我们可以在程序里同时设置许多个断点。

```
(gdb) print array[0]
$1 = {data = "bill",  '\000' <repeats 4091 times>, key = 3}
```

要想查看一组连续的数据项，我们可以用“@<number>”记号让gdb打印出指定个数的数组元素来。使用下面的命令可以把数组中的五个元素都打印出来：

```
(gdb) print array[0]@5
$2 = {{data = "bill", '\000' <repeats 4091 times>, key = 3}, {
    data = "neil", '\000' <repeats 4091 times>, key = 4}, {
    data = "john", '\000' <repeats 4091 times>, key = 2}, {
    data = "rick", '\000' <repeats 4091 times>, key = 5}, {
    data = "alex", '\000' <repeats 4091 times>, key = 1}}
```

我们对输出结果做了些修饰，让它们读起来容易一些。因为这是循环语句第一次执行时的情况，所以数组还没有发生变化。我们让程序继续执行，随着程序执行的进展，我们将看到数组array后续的变化：

```
(gdb) cont
```

```
Continuing.

Breakpoint 1, sort (a=0x8049684, n-4) at debug4.c:20
20      /*  20  */      for(; i < n && s != 0; i++) {

(gdb) print array[0]@5
$3 = {{data = "bill", '\000' <repeats 4091 times>, key = 3}, {
    data = "john", '\000' <repeats 4091 times>, key = 2}, {
    data = "neil", '\000' <repeats 4091 times>, key = 4}, {
    data = "alex", '\000' <repeats 4091 times>, key = 1}, {
    data = "rick", '\000' <repeats 4091 times>, key = 5}}
(gdb)
```

我们可以通过display命令对gdb进行设置，让它在程序停在每一个断点位置的时候都把数组打印出来，如下所示：

```
(gdb) display array[0]@5
1: array[0] @ 5 = {{data = "bill", '\000' <repeats 4091 times>, key = 3}, {
    data = "john", '\000' <repeats 4091 times>, key = 2}, {
    data = "neil", '\000' <repeats 4091 times>, key = 4}, {
    data = "alex", '\000' <repeats 4091 times>, key = 1}, {
    data = "rick", '\000' <repeats 4091 times>, key = 5}}
```

我们还可以对断点做进一步设置，使程序不是在断点停下来，而是把我们要求的数据显示出来后继续往下执行。我们用command命令来完成这一工作，它的作用是让我们指定在程序到达断点位置时将要执行的调试器命令。因为我们已经设置了一个displsy显示命令，所以我们只需把断点位置上的命令设置为继续执行就行了。如下所示：

```
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
> cont
> end
```

现在，当我们让程序继续执行的时候，它将一直执行到结束；外层循环每执行一次都会把数组的值打印一次。如下所示：

```
(gdb) cont
Continuing.

Breakpoint 1, sort (a=0x8049684, n=3) at debug4.c:20
20      /*  20  */      for(; i < n && s != 0; i++) {
1: array[0] @ 5 = {{data = "john", '\000' <repeats 4091 times>, key = 2}, {
    data = "bill", '\000' <repeats 4091 times>, key = 3}, {
    data = "alex", '\000' <repeats 4091 times>, key = 1}, {
    data = "neil", '\000' <repeats 4091 times>, key = 4}, {
    data = "rick", '\000' <repeats 4091 times>, key = 5}}

Breakpoint 1, sort (a=0x8049684, n=2) at debug4.c:20
20      /*  20  */      for(; i < n && s != 0; i++) {
1: array[0] @ 5 = {{data = "john", '\000' <repeats 4091 times>, key = 2}, {
    data = "alex", '\000' <repeats 4091 times>, key = 1}, {
    data = "bill", '\000' <repeats 4091 times>, key = 3}, {
    data = "neil", '\000' <repeats 4091 times>, key = 4}, {
    data = "rick", '\000' <repeats 4091 times>, key = 5}}
array[0] = {john, 2}
array[1] = {alex, 1}
array[2] = {bill, 3}
array[3] = {neil, 4}
array[4] = {rick, 5}

Program exited with code 025.
(gdb)
```

gdb报告这个程序在退出时有一个不常见的退出码。这是因为程序本身没有调用exit，没有从main函数返回一个有意义的值。这个例子里的退出码没有什么实际意义，而要想提供一个有意义的退出码，就应该调用exit。

程序执行外层循环的次数少于我们的预期值。我们看到循环终止条件中使用的参数n的值在每次到达断点的时候都有所减少。这就意味着循环没有执行到足够的次数。问题出在第30行对变量n做的减法操作上：

```
/*  30  */          n--;
```

这是一个优化程序的措施，它是这样考虑的：在每一次外层循环的末尾，数组array中最大的元素将被放到最底部，还需要排序的元素也就相应地减少了。但是，正如我们看到的，这个优化措施影响到了外层循环，并引起了问题。最简单的修补办法（当然还有其他办法）是删掉引起问题的这一行。我们用调试器打上这个“补丁”，然后再看看这个改动能否解决问题。

9.3.7 用调试器打补丁

我们已经看到我们能够通过调试器设置断点和查看变量的取值。把断点和程序动作结合起来就能检查程序漏洞的某个“补丁”是否有效，而这样做并没有对源代码做实际修改，也不必重新编译。在我们的例子里，我们需要在程序的第30行中断程序，对变量n做加法。这样，当程序执行到第30行的时候，n的值其实并没有发生变化。

我们重新开始执行这个程序。首先，我们要删除刚才设置的断点和display命令。我们可以用info命令查看自己曾经设置过哪些断点和display命令。如下所示：

```
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
1:   y  array[0] @ 5
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483f1 in sort at debug4.c:20
        breakpoint already hit 4 times
        cont
```

我们可以禁止这些设置，也可以直接删除掉它们。如果禁止它们，我们就能在今后必要的时候重新激活这些设置，如下所示：

```
(gdb) disable break 1
(gdb) disable display 1
(gdb) break 30
Breakpoint 2 at 0x8048570: file debug4.c, line 30.
(gdb) commands 2
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>set variable n = n+1
>cont
>end
(gdb) run
Starting program: /home/neil/debug4

Breakpoint 2, sort (a=0x8049684, n=5) at debug4.c:30
30      /*  30  */              n--;

Breakpoint 2, sort (a=0x8049684, n=5) at debug4.c:30
```

```
30      /*  30  */              n--;

Breakpoint 2, sort (a=0x8049684, n=5) at debug4.c:30
30      /*  30  */              n--;

Breakpoint 2, sort (a=0x8049684, n=5) at debug4.c:30
30      *  30  */               n--;

Breakpoint 2, sort (a=0x8049684, n=5) at debug4.c:30
30      *  30  */               n--;
array[0] = {alex, 1}
array[1] = {john, 2}
array[2] = {bill, 3}
array[3] = {neil, 4}
array[4] = {rick, 5}

Program exited with code 025.
(gdb)
```

程序一直执行到结束并给出了正确的结果。我们现在可以对源代码进行修改并转移到用更多数据对它进行测试的阶段去了。

9.3.8 深入学习gdb

GNU调试器的功能是非常强大的，它可以向我们提供许多与执行中的程序的内部状态有关的信息。在支持硬件断点功能的系统上，你可以通过gdb实时监控变量的变化情况。硬件断点是某些CPU提供的功能，这些处理器能够在特定事件发生时自动停止运行，这些事件通常是对某个给定内存区间的一个访问动作。gdb也有类似的表达式“监视”功能：即不管计算发生在程序的什么地方，只要表达式取了一个特定的值，gdb就会自动停止程序的运行；但这个功能会增加系统性能方面的负担。

设置断点的时候还可以给它加上计数和条件，经过预定计数之后或条件满足时才触发它们。

gdb还能把自己附着到已经运行着的程序上。这对调试客户/服务器系统很有帮助，因为你能够在异常服务器进程正在运行的时候对它进行调试，不必让它停下来，过后再启动它。编译程序时可以用（比如说）“gcc -O -g”同时获得程序优化和调试信息。但这样做的缺点是优化会对程序代码的先后顺序进行调整；因此，在对代码做单步调试的时候，原来连贯的源代码就不免要跳来转去的了。

gdb遵守GNU公共许可证的条款规定，大多数UNIX系统都能够支持它。我们强烈建议读者掌握这一工具。

9.4 其他调试工具

除了gdb等层出不穷的调试器，UNIX系统一般还提供了大量能够用来帮助完成调试工作的其他工具。其中有的提供的是关于程序的静态信息，另外一些则能够提供动态分析。

静态分析只能从源代码入手提供关于程序本身的资料。ctags、cxref和cflow等就是一些源文件的静态分析程序，它们能够提供函数调用和函数所在位置等方面的有用信息。

动态分析提供的是与程序运行过程中的行为有关的信息。prof个gprof等就是一些这样的程序，它们提供的信息包括已经执行了哪些函数，执行时间有多长等。

我们下面将对几个这类工具及其输出进行介绍。虽然这些工具都有可以免费获得的版本，但并不是在每个系统都能找到它们。

9.4.1 lint：清理程序中的“垃圾”

早期的UNIX系统提供了有关名为lint的工具。从本质上看，它只是C语言编译器的一个前端，但增加了一些“常识”性的测试功能，能够报告出一些异常的情况。它能检测出来的异常情况包括：使用了未经赋值的变量、函数的参数不够或类型不对等。

比较先进的C语言编译器也能产生类似的警告信息，这是以编译过程的性能为代价换来的。lint本身已经落后于C语言的标准化工作了。因为这个工具是在一个早期C语言编译器的基础上开发出来的，所以与ANSI语法的配合不是很理想。lint有许多种适用于UNIX的商业版本，并且在因特网上至少有一种是专为Linux开发的，它就是Larch。Larch是MIT（麻省理工学院）为早期技术规范开发工具软件这一项目的组成部分。与lint类似的工具还有lclint，它能够提供有用的代码审查注释。

> lclint可以在网址http: //www.sds.lcs.mit.edu/Larch上找到。

我们以刚才调试过的示例程序的一个前期版本为例来说明lclint的语法。下面是lclint运行时给出的输出：

```
$ lclint debug0.c
LCLint 1.4c - Fri Oct 13 10:28:08 MET 1995

debug0.c:14,22: Old style function declaration.
debug0.c:15,17: **** Processing Params ****
debug0.c:20,24: Unrecognized identifier: n
debug0.c:20,35: Variable s used before set
debug0.c:20,20: if predicate not bool, type int: i < n & s != 0
debug0.c:32,14: Path with no return in function declared to return int
debug0.c:36,14: Return value (type int) ignored: sort(array, 5)
debug0.c:37,14: Path with no return in function declared to return int

Finished LCLint checking - 8 code errors found
$
```

lclint工具报告有程序里有老式的（非ANSI标准）函数定义、函数返回类型和真正返回的值（或者没有返回值）不一致。这些其实并不会影响到程序的操作，但应该引起程序员的注意。

它还找出了两个真正的漏洞，这两个漏洞出现下面这段代码里：

```
/*  18  */      int s;
/*  19  */
/*  20  */      for(; i < n & s != 0; i++) {
/*  21  */              s = 0;
```

lclint发现第20行使用的变量s没有经过初始化，并且它认为操作符“&”应该是一个更常见的“&&”操作符。在上面这段代码里，“&”操作符改变了测试的含义，确实是这个程序存在的一个问题。

这两个错误都在调试之前的代码审查阶段就得到了纠正。虽然这两个错误都是我们为演示目的故意放在那里的，但在程序设计实践里，这样的错误可以说是屡见不鲜。

9.4.2 函数调用工具

ctags、cxref和xflow这三个工具构成了X/Open技术规范的一部分内容，因此，只要是具备软件开发能力的UNIX系统，就会有这三个工具。

> 这些工具，包括本章介绍的其他一些工具可能没有被包括在你Linux发行版本里。如果是这样，你可能会去因特网上查找它们。一个比较好的（用来查找支持RPM软件包格式的Linux发行版本）出发点是http: //rufus.w3.org/linux/RPM。

1. ctags程序

ctags程序的作用是为程序中的各种函数创建一个索引。每个函数对应一个清单，清单里列出了它在程序中的调用位置，就象书本的索引。下面是它的语法定义：

```
ctags [-a] [-f filename ] sourcefile sourcefile . . .
ctags -x sourcefile sourcefile . . .
```

在缺省的情况下，ctags将在当前子目录里创建出一个名为tags的文件来，在任一源文件里声明过的每一个函数都会出现在这个文件里，文件的格式是下面这样的文本行：

```
announce        app_ui.c            / ^static void announce(void) /
```

文件中每一行的构成是这样的：一个函数名、声明该函数的文件、一个可以用来在文件里查找该函数定义的规则表达式。Emacs等编辑器可以利用这类文件帮助程序员对源代码进行巡查。

另外一种做法是通过ctags命令的“-x”选项（如果你的版本上有的话）在标准输出上列出类似格式的清单来：

```
find_cat        403 app_ui.c        static cdc_entry find_cat
```

用“-f filename”选项可以把这些输出重定向到另一个不同的文件去；而“-a”选项可以把它追加到一个现有文件的末尾。

2. cxref程序

cxref程序对C语言源代码进行分析，最终将生成一个交叉引用表。它能够查到每个记号（变量、“# define”常数定义、函数）都在程序里的哪些地方出过。它生成的是一个经过排序的清单，记号的定义地点用一个星号（*）做标记，如下所示：

```
SYMBOL                  FILE      FUNCTION      LINE

BASENID                 prog.c      -           *12  *96  124  126  146  156  166
BINSIZE                 prog.c      -           *30  197  198  199  206
 BUFMAX                 prog.c      -           *44   45   90
 BUFSIZ  /usr/include/stdio.h       -            *4
    EOF  /usr/include/stdio.h       -           *27
   argc                 prog.c       -           36
                        prog.c     main          *37   61   81
   argv                 prog.c       -           36
                        prog.c     main          *38   61
calldata                prog.c       -            *5
                        prog.c     main           64  188
  calls                 prog.c       -           *19
                        prog.c     main           54
```

在作者的机器上，上面这些输出是用下面这条命令产生的：

```
$ cxref *.c *.h
```

这个命令的语法格式随版本的不同而不同。如果想了解系统上有没有xref或者想了解它的使用方法，请参考系统文档或用man命令查阅它的使用手册页。

3. cflow程序

cflow程序的作用是生成一个函数调用树，这个示意图按从全局到局部的顺序列出了函数彼此调用与被调用的关系。它可以让我们看清楚一个程序的框架结构，弄懂它的操作流程，了解对一个函数的改动将会产生什么样的后果等。有些版本的cflow除了能对源代码进行处理外，还可以处理目标代码文件。详细资料和使用方法请查阅使用手册页。

下面是某个cflow版本（cflow-2.0）的输出样本，该版本可以从因特网上下载到，它的维护工作是由Marty Leisner负责的。如下所示：

```
1       file_ungetc (prcc.c 997)
2       main (prcc.c 70)
3               getopt ()
4               show_all_lists (prcc.c 1070)
5                       display_list (prcc.c 1056)
6                               printf ()
7                       exit ()
8               exit ()
9               usage (prcc.c 59)
10                      fprintf ()
11                      exit ()
```

这个输出样本告诉我们：main调用了show_all_lists（以及其他一些函数），show_all_lists又调用了display_lists，而display_lists本身调用了printf。

这个版本的cflow有一个“-i”选项，它的作用是按从局部到全局的顺序生成一个函数调用树。对应着每个函数列出都有哪些函数调用了它。这说起来好象很复杂，可实际做起来很简单。下面是一个样本：

```
19      display_list (prcc.c 1056)
20              show_all_lists (prcc.c 1070)
21      exit ()
22              main (prcc.c 70)
23              show_all_lists (prcc.c 1070)
24              usage (prcc.c 59)
...
74      printf ()
75              display_list (prcc.c 1056)
76              maketag (prcc.c 487)
77      show_all_lists (prcc.c 1070)
78              main (prcc.c 70)
...
99      usage (prcc.c 59)
100             main (prcc.c 70)
```

我们可以看出（比如说）都有哪些函数调用了exit：它们是main、show_all_lists和usage。

9.4.3 执行记录

执行记录是查找程序执行性能低下这类问题原因的一种常用技巧。它需要有编译器特殊选项和辅助程序的支持，从程序的执行记录文件可以看出它的时间都用在什么操作上了。

prof和gprof程序

在编译程序的时候，给编译器加上“-p”标志（对应于prof）或“-pg”标志（对应于gprof）就能创建出这样一种程序：它在执行的时候将生成一个执行记录（类似于航空飞行记录）文件，我们把这类程序叫做“入档程序”。prof程序（和它的GNU等效工具gprof）的作用就是根据执行记录文件总结出一个程序性能报告来。编译命令如下所示：

```
$ cc -pg -o program program.c
```

程序是用一个特殊版本的C语言库链接出来的，将会增加一些监控代码。不同的系统在做法会有所差异，但一般都要靠程序的频繁中断来实现监控功能，每次中断都会把程序当前执行到的位置记录下来。这些监控数据将被写到当前子目录里的mon.out文件（gprof程序用的是gmon.out）里去。如下所示：

```
$ ./program
$ ls -ls
   2 -rw-r--r--   1 neil     users        1294 Feb  4 11:48 gmon.out
```

prof/gprof程序读取这些监控数据，生成一个报告。程序选项及其使用方面的细节请查阅它的使用手册页。下面是一些（有所删节）示例性的gprof输出：

```
cumulative   self      self     total
   time     seconds   seconds   calls  ms/call  ms/call      name
   18.5        0.10     0.10    8664     0.01     0.03    _doscan [4]
   18.5        0.20     0.10                      mcount (60)
   14.8        0.28     0.08   43320     0.00     0.00    _number [5]
    9.3        0.33     0.05    8664     0.01     0.01  _format_arg [6]
    7.4        0.37     0.04  112632     0.00     0.00    _ungetc [8]
    7.4        0.41     0.04    8757     0.00     0.00   _memccpy [9]
    7.4        0.45     0.04       1    40.00   390.02     _main [2]
    3.7        0.47     0.02      53     0.38     0.38    _read [12]
    3.7        0.49     0.02                      w4str [10]
    1.9        0.50     0.01   26034     0.00     0.00    _strlen [16]
    1.9        0.51     0.01    8664     0.00     0.00   strncmp [17]
```

9.5 假设验证

在软件的开发过程中，引入printf调用等调试性代码，进行条件性编译等做法都是很正常的；但成品软件里还留有这些信息的情况倒不多见。经常会出现这样的情况：程序操作中出现的问题与“不正确的假设”有关，程序的代码倒没有什么错误。这些“不正确的假设”往往是一些所谓“不会发生的”事件。比如说，人们在编写函数的时候会认为它的输入参数会落在一个确定的范围内；而万一给它传递了不正确的数据，它就可能毁掉整个系统。

系统的内部逻辑必须保持步调一致，X/Open为此专门提供了一个宏定义assert。它的作用是检验某个假设是否成立，如果假设不成立的话就立刻停止程序的运行。

```
# include <assert.h>

void assert(int expression )
```

assert宏对表达式expression进行求值，如果结果非零，它会先往标准错误写一些诊断信息，然后调用abort结束程序的执行。

头文件assert.h定义的宏受NDEBUG标志的影响。如果程序里包括了这个头文件，但又定义

了NDEBUG标志，assert宏就不会被定义了。这就意味着我们可以在编译期间关闭假设验证功能，只要加上“-DNDEBUG”选项或者在把下面这条语句：

```
# define NDEBUG
```

加到每一个源文件的开头，先于“# include <assert.h>”语句就可以了。

assert带来的问题

但assert的这种用法隐含着一个问题。如果你在测试阶段使用了assert，但在成品代码里把这一功能关闭了，那你的成品代码在安全检查方面就比你对它进行测试时要差一些。在成品代码里激活假设验证功能是不可取的——你愿意顾客在使用你的软件时屏幕显示一条不友好的“assert failed”错误提示，然后就退出程序吗？比较好的解决办法是编写自己的错误捕捉例程，你可以在这个例程里实现一些假设验证功能，在成品代码里也不必完全禁止这些功能了。

还必须注意不要让assert表达式带上副作用。举例来说，如果你使用了一个带副作用的函数调用，这个副作用在去掉了假设验证功能的成品代码就不会再发生了。

下面这个assert.c程序定义了一个函数，它的参数必须是一个正数值。它用一个假设验证功能来保护自己不受非法参数的影响。

动手试试：assert宏

先是必要的头文件，其中包括assert.h。然后定义了一个平方根函数，它要求自己的参数必须是正的。最后是我们编写的main函数：

```
#include <stdio.h>
#include <math.h>
#include <assert.h>
double my_sqrt(double x)
{
    assert(x >= 0.0);
    return sqrt(x);
}

int main()
{
    printf("sqrt +2 = %g\n", my_sqrt(2.0));
    printf("sqrt -2 = %g\n", my_sqrt(-2.0));
    exit(0);
}
```

现在，当我们运行这个程序的时候，如果给这个程序传递一个非法的值，就会看到一个假设验证冲突错误。假设验证冲突错误信息的格式随系统的变化而变化。

```
$ cc -o assert assert.c -lm
$ ./assert
sqrt +2 = 1.41421
assert.c:7: my_sqrt: Assertion 'x >= 0.0' failed.
Aborted (core dumped)
$
```

操作注释：

当我们试图用一个负数来调用my_sqrt的时候，假设验证失败了。assert宏给出了发生假设验

证冲突的文件名和行号，还给出了假设验证的条件。程序被一个abort中断陷阱结束了运行。这就是assert调用abort的结果。

用“-DNDEBUG”选项重新编译这个程序，假设验证功能将被排除在编译结果之外。当我们再次通过my_sqrt调用sqrt函数的时候得到的是一条算术运算错误，如下所示：

```
$ cc -o assert -DNDEBUG assert.c -lm
$ ./assert
sqrt +2 = 1.41421
Floating point exception
$
```

有些版本的数学函数库会返回一个NaN值（“Not a Number”，不是一个数字）指示一个无效的操作结果。比如：

```
sqrt -2 = nan
```

9.6 内存调试

动态内存分配是个很容易出现程序漏洞的领域，漏洞一旦出现，还很难查找。如果在程序里使用了malloc和free来分配内存，就必须对自己分配过的每一块内存都做到心中有数，并且绝不要使用已经释放了的内存块，这一切都是非常重要的。

内存块通常都是用malloc分配的，它们会被赋给一个指针变量。如果该指针变量发生了变化，又没有其他指针指向这块内存，就无法继续对它进行访问了。这是一种内存流失现象，它会使你程序的长度增大。如果你流失了大量内存，你的系统就会越来越慢，最终的结局是耗尽内存。

如果在一个已分配内存块尾部的后面（或内存块头部的前面）写数据，你就很可能损坏malloc库用来记录内存分配情况的数据结构。出现这种问题以后，经过一段时间，一个malloc调用，甚至是一个free调用，都可能会引发一次内存段冲突，而你的程序也就崩溃了。要想查出错误发生的准确位置是十分困难的，因为内存段发生冲突的现象是没有规律的，在引发程序崩溃的事件发生之前，谁也说不准要等多长的时间。

帮助解决这两类问题的工具（有商业版也有免费版）都已经被开发出来了，对此我们不应该感到吃惊。有许多不同版本的malloc和free，其中一些添加了检查内存分配和内存回收情况的代码，它们尝试并可能解决的问题包括一个内存块被释放了两次以及其他一些错误的用法。

9.6.1 ElectricFence

ElectricFence库是由Bruce Perens开发的，在RedHat等Linux发行版本上它是一个可选组件，在因特网上也很容易找到它。它尝试使用UNIX的虚拟内存功能来保护malloc和free使用的内存。它的目标是在内存被破坏之前让程序停止运行。

动手试试：ElectricFence

下面这个efence.c程序用malloc分配了一个内存块，然后在这个内存块尾部以外的地方写数据。我们来看看将会发生什么事情。

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

int main()
{
    char *ptr = (char *) malloc(1024);
    ptr[0] = 0;

    /* Now write beyond the block */
    ptr[1024] = 0;
    exit(0);
}
```

当我们编译并运行这个程序的时候，看不到任何异常的现象。但malloc的内存区却可能已经受到了一定程度的破坏，我们迟早会遇到麻烦。如下所示：

```
$ cc -o efence efence.c
$ ./efence
$
```

接下来，我们用ElectricFence库libefence.a来链接同一个程序，我们马上就收到响应了。如下所示：

```
$ cc -o efence efence.c -lefence
$ ./efence

  Electric Fence 2.0.5 Copyright (C) 1987- 1998 Bruce Perens.
Segmentation fault
$
```

在调试器下运行这个程序，找出问题根源：

```
$ cc -g -o efence efence.c -lefence
$ gdb efence
(gdb) run
Starting program: /home/neil/efence

  Electric Fence 2.0.5 Copyright (C) 1987-1998 Bruce Perens.

Program received signal SIGSEGV, Segmentation fault.
0x80008e4 in main () at efence.c:11
11          ptr[1024] = 0;
(gdb)
```

操作注释：

ElectricFence把malloc及其关联函数替换为使用计算机处理器硬件虚拟内存功能的版本，用这种办法保护系统不受无效内存访问的损害。在出现一个无效内存访问的时候，它会引发一个内存段冲突信号，而程序也就停止了。

9.6.2 Checker

Checker是由Tristan Gingold开发的，它是一个适用于Linux和UNIX操作系统的改进版编译器后端和C语言库，它能够检查出我们前面讨论过的许多问题。特别值得一提的是它能够检查出不正确的指针引用、数组访问错误和内存流失。它一般不包括在Linux的发行版本里，但可以在http: //www.gnu.org/software/checker/checker.html处找到。

要想使用Checker，程序和函数库都必须重新编译。好在Linux上绝大多数软件都找得到源代码，这也就算不上什么限制了。与Checker一起使用的预编译库也很容易在网上找到。

动手试试：Checker

下面的程序checker.c分配了一些内存，然后进行了以下几种操作：从内存块未初始化的单元里读数据、在内存块尾部以外写数据、最后把内存块的指针弄乱了。

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

int main()
{
    char *ptr = (char *) malloc(1024);
    char ch;

    /* Uninitialized read */
    ch = ptr[0];

    /* Write beyond the block */
    ptr[1024] = 0;

    /* Orphan the block */
    ptr = 0;
    exit(0);
}
```

在使用Checker的时候，我们只需简单地把我们的编译器命令替换为checkergcc就可以了。这是一个驱动性程序，负责调用正确的编译器版本和用特殊的Checkered库链接程序。

运行这个程序，我们看到它查找出许多问题：

```
$ checkergcc -o checker checker.c
$ ./checker
Checker version 0.7 Copyright (C) 1993,1994,1995 Tristan Gingold.
This program has been compiled with 'checkergcc' or 'checkerg++'.
Checker is a memory access detector.
Checker is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.
For more information, set CHECKEROPTS to '-help'
From Checker (pid:01359): 'checker' is runing (Sun Feb  4 14:54:00 1996)

From Checker (pid:01359): (ruh) read uninitialized byte(s) in a block.
When Reading 1 byte(s) at address 0x0801e12c, inside the heap (sbrk).
0 bytes into a block (start: 0x801e12c, length: 1024, mdesc: 0x0).
The block was allocated from:
        pc=0x080099db in _malloc() at ./l-malloc/malloc.c:211
        pc=0x08009a3d in malloc() at ./l-malloc/malloc.c:232
        pc=0x080001df in main() at checker.c:6
        pc=0x0800010c in _start() at :0
Stack frames are:
        pc=0x08000200 in main() at checker.c:10
        pc=0x0800010c in _start() at :0
From Checker (pid:01359): (bvh) block bounds violation in the heap.
When Writing 1 byte(s) at address 0x0801e52c, inside the heap (sbrk).
0 bytes after a block (start: 0x801e12c, length: 1024, mdesc: 0x0).
The block was allocated from:
        pc=0x080099db in _malloc() at ./l-malloc/malloc.c:211
        pc=0x08009a3d in malloc() at ./l-malloc/malloc.c:232
        pc=0x080001df in main() at checker.c:6
        pc=0x0800010c in _start() at :0
Stack frames are:
```

```
pc=0x08000225 in main() at checker.c:13
pc=0x0800010c in _start() at :0
```

这里，我们看到它查出了不良的读写操作，同时还给出了与之对应的内存块和内存块的分配位置。我们可以通过调试器给程序在错误地点设置一个断点。

Checker有许多选项，包括捕获某些特定类型错误和内存流失检查等在内。要检查我们例子里的内存流失，我们必须使用一个由CHECKEROPTS环境变量传递选项，经这个环境变量传递的选项还有其他几个。要想在程序运行结束后检查有无内存流失现象，我们需要指定“-D = end”选项，如下所示：

```
$ CHECKEROPTS=-D=end checker

...

From Checker (pid:01407): (gar) garbage detector results.
There is 1 leak and 0 potential leak(s).
Leaks consume 1024 bytes (1 KB) , 131193 KB.
( 0.00% of memory is leaked.)
Found 1 block(s) of size 1024.
Block at ptr=0x801e40c
        pc=0x08009900 in _malloc() at ./1-malloc/malloc.c:174
        pc=0x08009a3d in malloc() at ./1-malloc/malloc.c:232
        pc=0x080001df in main() at checker.c:6
        pc=0x0800010c in _start() at :0
```

操作注释：

checkergcc编译器会给我们的程序额外添上一些代码，对程序里出现的每一个指针引用进行检查。如果某个访问与某个已分配的内存块有关但又是无效的，Checker就给出一个错误提示消息。在程序即将结束的时候，回收例程开始运行，它的作用是检查有没有程序分配了但忘记释放的内存块。如果有，这类失去程序依托的内存块也会报告出来。

9.7 资源

本章介绍讨论的工具程序差不多都能从因特网上FTP站点那里找到。注意有些工具软件的作者可能保留了版权。这些工具程序有许多是来自著名的Linux软件站点ftp: //metalab.unc.edu/pub/linux，我们希望能够在它们发行的第一时间找到新的版本。

其他因特网上的资源请参考附录C。

9.8 本章总结

在这一章里，我们学习了一些调试工具和技巧。UNIX，特别是Linux里有不少工具能够帮助大家把程序漏洞找出来，修补掉。

我们用gdb查出并纠正了示例程序中的缺陷和漏洞，并向大家介绍了几个静态分析工具，其中包括cflow和lclint。

最后，我们对使用动态分配内存时可能出现的问题进行了讨论，有几个工具可以帮助我们对它们进行诊断，比如ElectricFence和Checker。

第10章　进程和信号

进程和信号是构成UNIX操作环境的一块基石。它们控制着一台UNIX计算机系统上的绝大部分活动。不管你是系统程序员、应用程序员、还是系统管理员，弄明白UNIX的进程管理将使你“一切尽在掌握”。

在这一章里，我们将看到Linuix环境中的进程是如何被操作和管理的，怎样才能查明计算机在任一给定时刻在干些什么。我们还将学习如何在自己的程序里启动和停止其他的进程、如何让进程收发消息，以及如何避免进程“僵死”等内容。总之，我们将在以下几个方面进行学习：

- 进程的结构、类型和时间安排。
- 以多种方法启动新进程。
- 父进程、子进程、僵进程。
- 什么是信号以及如何使用它们。

10.1　什么是进程

《统一UNIX技术规范》第二版（“Single UNIX Specification, Version 2”，即我们常说的UNIX98技术规范）和第一版（即UNIX95技术规范）把一个进程定义为：“一个其中运行有一个或者多个线程的地址空间和线程要求使用的系统资源。”我们将在下一章对线程进行讨论。现在，我们可以把一个进程看成是一个运行中的程序。

像UNIX这样的多任务操作系统能够让许多程序同时运行。每一个运行着的程序就构成了一个进程，这在X窗口系统（有时简单地称为X）等窗口化系统上证据就更充分了。X窗口系统和微软的Windows一样提供了一个图形化的用户操作界面，允许许多应用程序同时运行。每个应用程序可以显示一个或者多个窗口。我们将在第16章对X窗口系统做进一步学习。

UNIX又是一个多用户系统，它能够让许多个用户在同一时间访问系统。每个用户又可以同时运行许多个程序，甚至可以是同一个程序的多次运行。系统本身也运行着一些管理系统资源和控制用户权限的程序。

我们在第4章曾经学过，一个正在运行的程序（或者叫进程），是由程序代码、数据、变量（占用着系统内存）、打开的文件（文件描述符）和一个环境组成。通常，UNIX系统会让进程共享代码和系统库，所以在任何时刻内存里都只有代码的一份拷贝。

10.2　进程的结构

我们来看看操作系统是如何管理两个进程的。如果两个用户，比如说neil和rick，为了在不同的文件里查找不同的字符串而同时运行了grep程序，他们使用的进程如图10-1所示。

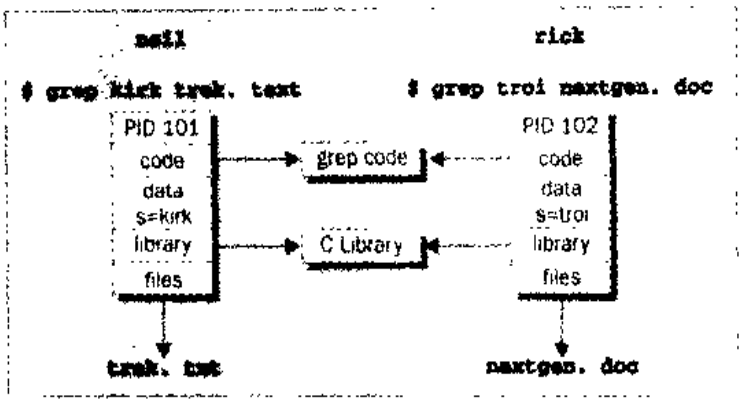


图 10-1

我们运行ps命令（马上就会介绍到），其输出情况应该和下面这些内容差不多：

```
$ ps -af
UID      PID   PPID  C  STIME  TTY   TIME      CMD
rick     101   96    0  18:24  tty2  00:00:00  grep pid_t /usr/include/sys/*.h
neil     102   92    0  18:24  tty4  00:00:00  grep XOPEN /usr/include/features.h
```

每个进程都会分配到一个独一无二的数字编号，我们称之为“进程标识码”（process identifier），或者就直接叫它PID。这是一个正整数，取值范围从2到32768。当一个进程被启动的时候，它会顺序挑选下一个未使用的编号数字做为自己的PID；如果它们已经轮过一圈了，新的编号重新从2开始。数字1一般是为特殊进程init保留的，它负责管理其他的进程。我们过一会儿再讨论init进程。现在，我们看到由neil和rick启动的两个进程分配到的标识码分别是“101”和“102”。

grep命令将要执行的程序代码保存在一个磁盘文件里。在普通正常操作情况下，一个UNIX进程是不能对用来存放程序代码的内存区域进行写操作的，也就是说，程序代码是以只读方式被加载到内存里去的。因为不允许对这个区域做写操作，所以我们可以从示意图中看到：两个进程安全地共享着一份代码拷贝。

系统库也是可以共享的。举例来说，不管有多少运行中的程序在调用printf函数，内存里只要有它的一份拷贝就足够了。与微软的Windows使用的动态链接库（dynamic link library，简称DLL）相比，UNIX的做法要复杂一些，但原理是相似的。

从示意图里还可以看出，共享带来的好处还有让容纳可执行程序grep的磁盘文件比较小，原因是它里面少了共享库代码。这对一个程序来说算不上什么，但对一个完整的操作系统来说，把常用例程提取出来放入（比如说）C语言的标准库将节省大量的空间。

当然，一个程序在运行时所需要的东西并不是都能与其他程序共享的。比如说，它使用的变量就与其他进程的有明显区别。在这个例子里，我们看到传递给grep命令的搜索字符串是以变量“s”的形式出现在每个进程的数据区里。它们各有各的去处，通常不能被其他进程读取。两个grep命令使用的文件也各不相同，进程通过各自的文件描述符访问各自的文件。

除了上面这些东西，进程还有它自己的堆栈空间，函数的局部变量与控制函数调用和返回的信息就保存在其中。它还有自己的环境空间，其环境变量设置出来的环境是供这个进程专用的，我们在第4章介绍putenv和getenv的时候已经见到过了。进程还必须有自己的程序计数器，

用来记录它自己执行到了什么位置，也就是在执行线程中的位置。我们将在下一章里看到：当我们使用了线程的时候，进程可以有不止一个执行线程。

在许多Linux系统上，也包括一些UNIX系统，在子目录/proc里有一组特殊的文件。它们是一些相当特殊的文件，因为它们允许你在进程运行的时候“透视”到进程的内部。

最后，因为UNIX有一个虚拟内存系统，能够把代码和数据以内存页面的形式交换到硬盘上去，所以它能管理的进程比物理内存所能容纳的要多得多。

10.2.1　进程表

UNIX进程表是这样一种数据结构，它把当前加载在内存里的所有进程的有关信息保存在一个表里，其中包括进程的PID、进程的状态和命令字符串、ps命令输出的各种信息等。系统通过进程的PID对它们进行管理，这个编号在进程表里被用做一个索引。这个表的长度是有限的，所以系统能够同时支持的进程个数也是有限的。“古老的”UNIX系统只能支持256个进程；在比较现代的版本上，这一限制已放松了许多，只要划分给进程表使用的内存里还有容纳数据项的地方，就允许继续启动进程。

10.2.2　查看进程

ps命令能够列出我们正在运行的进程、其他以后正在运行的进程，或者系统上正在运行的全部进程。下面是一个输出样本：

```
$ ps -af
UID        PID  PPID  C STIME TTY          TIME CMD
root       433   425  0 18:12 tty1     00:00:00 [bash]
rick       445   426  0 18:12 tty2     00:00:00 -bash
rick       456   427  0 18:12 tty3     00:00:00 [bash]
root       467   433  0 18:12 tty1     00:00:00 sh /usr/X11R6/bin/startx
root       474   467  0 18:12 tty1     00:00:00 xinit /etc/X11/xinit/xinitrc --
root       478   474  0 18:12 tty1     00:00:00 /usr/bin/gnome-session
root       487     1  0 18:12 tty1     00:00:00 gnome-smproxy --sm-client-id def
root       493     1  0 18:12 tty1     00:00:01 [enlightenment]
root       506     1  0 18:12 tty1     00:00:03 panel --sm-client-id default8
root       508     1  0 18:12 tty1     00:00:00 xscreensaver -no-splash -timeout
root       510     1  0 18:12 tty1     00:00:01 gmc --sm-client-id default10
root       512     1  0 18:12 tty1     00:00:01 gnome-help-browser --sm-client-i
root       649   445  0 18:24 tty2     00:00:00 su
root       653   649  0 18:24 tty2     00:00:00 bash
neil       655   428  0 18:24 tty4     00:00:00 -bash
root       713     1  2 18:27 tty1     00:00:00 gnome-terminal
root       715   713  0 18:28 tty1     00:00:00 gnome-pty-helper
root       717   716 13 18:28 pts/0    00:00:01 emacs
root       718   653  0 18:28 tty2     00:00:00 ps -af
```

这份输出列出了许多进程的资料，我们可以在表里找到一些与X窗口和Emacs编辑器有关的进程。“TTY”一栏表示进程是从哪个终端启动的；“TIME”是进程的CPU占用时间；“CMD”是启动进程时所用的命令。我们来仔细研究几个数据项。

```
neil       655   428  0 18:24 tty4     00:00:00 -bash
```

初始化登录操作是在第4个虚拟终端上完成的，它也是这台机器的控制台。它启动运行的程序是Linux系统的默认shell：bash。

```
root       467     433     0 18:12 tty1          00:00:00 sh /usr/X11R6/bin/startx
```

X窗口系统是由命令startx启动的。这是一个用来启动X服务器并运行一些初始化X窗口程序的shell脚本程序。

```
root              717           716 13 18:28 pts/0          00:00:01 emacs
```

这个进程代表着X窗口系统里一个运行着Emacs编辑器的窗口。它是由窗口管理器响应创建一个新窗口的请求而启动的。为了让shell能够对窗口进行读写，还分配了一个新的伪终端pts/0。

```
root              512   1   0 18:12 tty1  00:00:01 gnome-help-browser--sm-client-i
```

这是由窗口管理器启动的Gnome帮助信息浏览器。

在默认的情况下，ps程序只给出与某个终端、某个控制台、某个串行口或某个伪终端保持着连接的进程的情况。其他进程在运行时不需要通过终端与用户进行通信。这些基本上都是一些系统进程，UNIX通过系统进程来管理共享性资源。要想查看全部的进程，就要给ps加上“-a”选项；要想看到完整的信息，要加上“-f”选项。ps命令的准确语法和它输出内容的格式随系统的不同会稍有变化。有关ps命令的选项及其输出格式方面的详细资料请查阅它的使用手册页。

10.2.3 系统进程

下面是一些这台Linux系统上运行着的其他一些进程。为简洁起见，我们对输出结果做了删节。

```
$ ps -ax
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
   40  ?  S     0:01 /usr/sbin/syslogd
   46  ?  S     0:00 /usr/sbin/lpd
   51  ?  S     0:00 sendmail: accepting connections
   88 v02 S     0:00 /sbin/agetty 38400 tty2
  109  ?  R     0:41 X :0
  192 pp0 R     0:00 ps -ax
```

我们在这里看到了一个非常重要的进程。

```
1   ?       S       0:00    init
```

一般说来，每个进程都是由另一个我们称之为“父进程”的进程启动的；被父进程启动的进程叫做子进程。当UNIX开始运行的时候，它会运行一个名为init的程序，它是一切进程的“祖先”，它的进程编号是“1”。这就是操作系统的进程管理器。我们将要学习到的其他系统进程都是由init启动的，或者是由被init启动的进程启动的。

用户登录上机的处理过程就是一个这样的例子。用户登录上机需要使用串行终端或拨号调制解调器，而init会为每一个登录设备启动一次getty程序。对应的ps命令输出就是下面这个样子的：

```
88 v02     S              0:00     sbin/agetty 38400  tty2
```

getty进程等待来自终端的操作，向用户显示熟悉的登录提示符，如何把控制移交给登录程序；登录程序先设置好用户环境，最后启动一个shell。当用户shell退出时，init会再启动另外一个getty进程。

我们将会看到，启动新的进程并等待其完成的能力是整个系统的基石。我们在本章后面的内容里将会看到如何从我们的程序里通过系统调用fork、exec和wait完成同样的工作。

10.2.4 进程的调度

ps命令的输出结果里还有一个与ps命令本身对应的数据项：

```
192 pp0    R              0:00   ps -ax
```

这一行的含义是：进程192处于执行状态（R），正在执行的命令是“ps-ax”。也就是说，进程出现在它自己的输出里了！状态指示标志只表示程序随时可以运行，并不一定是正在运行。在一台单处理器计算机上，任一时刻只能有一个进程在运行，所有进程是轮流运行的。每个进程轮到的运行时间（我们称之为“时间片”）是相当短暂的，这就给人以一种多个程序在同时运行的印象。“R”只表达这样一个意思：这个程序不需要等待其他进程执行完毕，或者它不需要等待输入输出操作的完成。这就是我们的示例输出里有两个这样的进程的原因（另一个进程是X显示服务器）。

UNIX系统通过一个进程调度器（scheduler）来决定下一个时间片应该分配给哪一个进程。这就要用到进程的优先级。我们最早是在第4章里遇见优先级概念的。优先级比较高的进程运行得比较频繁，而包括后台任务在内的低优先级进程运行得就不那么频繁。在UNIX里，进程的运行时间是不可能超过分配给它们的时间片的。它们是多任务系统中的进程，它们的挂起和继续运行不需要它们的合作。但早一些的系统，比如微软的Windows 3.x，通常要求进程明确地退出时间片，然后其他进程才能继续运行。

操作系统从“优先级基数”开始参照程序的行为来确定进程的优先级，这个基数的默认值是10。长期不间断运行的程序其优先级一般会低一些。在（比如说）等待输入数据时暂停运行的程序会得到奖励。这可以使程序及时响应用户的交互操作：当程序等待来自用户的输入数据时，系统会增加它的优先级；这样，当它准备继续执行的时候，就会因为有了比较高的优先级而优先运行。进程的优先级基数可以用nice命令设置，用renice命令进行调整。nice命令的作用是把进程的优先级基数降低10个点。我们可以用ps命令的“-l”或“-f”（长格式输出）选项查看活跃进程的优先级基数。我们感兴趣的数值列在“NI”（nice）那一栏。如下所示：

```
$ ps -l
 F   UID   PID  PPID PRI  NI SIZE  RSS WCHAN      STAT TTY      TIME COMMAND
 0   501   146     1   1   0   85  756 130b85     S    v01      0:00 oclock
```

我们看到oclock程序以默认的优先级基数运行着。如果我们用下面的命令来启动它：

```
$ nice oclock &
```

它将分配到一个等于+10的优先级基数值。如果我们用下面的命令调整这个基数值：

```
$ renice 10 146
146: old priority 0,new priority 10
```

这个时钟程序就将运行得不那么频繁了。我们再用ps命令查看修改后的优先级基数值：

```
F   UID   PID  PPID PRI  NI SIZE  RSS WCHAN      STAT TTY      TIME COMMAND
0   501   146     1  20  10   85  756 130b85     S N  v01      0:00 oclock
```

状态栏“STAT”里现在多出一个“N”，这表示此进程的优先级基数已经不是它原来的默认值了。ps命令输出中的“PPID”域给出的是父进程的进程ID，它是启动此进程运行的那个进程的PID；如果原来的父进程已经不存在了，就以init为父进程（PID值为1）。

UNIX进程调度器以优先级为根据决定允许哪个进程能够运行。各种版本的具体做法当然会有差异，但高优先级进程总会运行得更频繁一些。在某些情况下，只要还有高优先级进程可以运行，低优先级进程就根本不能运行。

10.3 启动新的进程

我们可以让一个程序在另一个程序的内部运行，也就是说，我们创建了一个新的进程。这个工作可以通过库函数system来实现。下面是它的定义：

```
# include <stdlib.h>

int system (const char *string);
```

system函数的作用是执行以字符串参数的形式传递给它的命令，并等待命令的完成。这个命令的执行情况就好像shell中的下面这条命令：

```
$ sh -c string
```

如果无法启动shell运行命令，system将返回“127”；出现不能执行system调用的其他错误时返回“-1”。如果system能够顺利执行，它将返回那个命令的退出码。

动手试试：system函数

我们利用system函数编写一个程序，让它替我们运行ps命令。虽然这个程序本身的用处不是很大，但我们将在后面的例子里对这一技术做进一步开发。其实从示例程序就可以看出来，system调用能够大大简化程序的编写工作。

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

int main()
{
    printf("Running ps with system\n");
    system("ps -ax");
    printf("Done.\n");
    exit(0);
}
```

当我们编译并运行这个system.c程序的时候，将看到如下所示的执行情况：

```
$ ./system
Running ps with system
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
...
  146 v01 S N   0:00 oclock
  256 pp0 S     0:00 ./system
  257 pp0 R     0:00 ps -ax
Done.
```

因为system函数要使用一个shell来启动预定的程序，所以我们可以把它放到后台去运行，只

体做法是把system.c中的函数调用语句修改为如下所示的样子：

```
system("ps -ax &" );
```

当我们编译并运行这个新版程序的时候，我们将看到：

```
$ ./system2
Running ps with system
Done.
$   PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
  .
  146 v01 S N   0:00 oclock
  266 pp0 R     0:00 ps -ax
```

操作注释：

在第一个例子里，程序以字符串“ps -ax”为参数调用system函数，相当于执行ps程序。我们的程序在ps命令完成后从system调用中返回。system函数可以很有用，但也有局限性。因为我们的程序必须等待由system调用启动的进程结束之后才能继续，我们就不能同时执行其他工作。

在第二个例子里，对system的调用将在shell命令结束时立刻返回。因为那个shell命令只是把一个程序放到后台去运行的一个请求，所以ps程序一开始运行shell就返回了，这和我们在shell提示符处敲入下面这条命令的效果的一样的：

```
$ ps -ax &
```

shell返回后，我们的程序在打印出“Done.”之后也返回了，但此时ps命令还没来得及产生任何输出呢。我们的程序结束之后，ps命令的输出出现在shell提示符处。进程的这类行为往往会给用户带来极大的困惑。想用好进程就必须对它们的动作做细致的调控。我们下面向大家介绍一个创建进程用的底层接口：exec。

> 一般说来，system函数远非是启动其他进程的理想手段，因为它必须用一个shell来启动预定的程序。因为必须先启动一个shell，然后才能启动程序，所以这种办法的效率很低；对shell的安装情况和它所处的环境的依赖也很大。我们将在下一节看到一种先进得多的程序启动办法，与system调用相比，我们应该坚持在程序设计中把它做为首选。

替换一个进程映像

exec名下是由多个关联函数组成的一个完整系列。在新进程的启动方式和程序参数的传递办法方面，它们各有各的做法。一个exec函数可以把当前进程替换为一个新进程，新进程由path或file参数指定。

```
#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0, ...,  (char *)0);
int execlp(const char *file, const char *arg0, ...,  (char *)0);
int execle(const char *path, const char *arg0, ...,  (char *)0, const char *envp[]);
int execv(const char *path, const char *argv[]);
```

```
int execvp(const char *file, const char *argv[]);
int execve(const char *path, const char *argv[], const char *envp[]);
```

这些函数可以分为两大类：execl、execlp和execle的参数个数是可变的，参数以一个空指针结束；execv和execvp的第二个参数是一个字符串数组，新程序在启动时会把在argv数组中给定的参数传递到main。这些函数通常都是用execve实现的，这是一种约定俗成的做法，并不是非这样不可。

名字最后一个字母是“p”的函数会搜索PATH环境变量去查找新程序的可执行文件。如果可执行文件不在PATH定义的路径上，就必须把包括子目录在内的绝对文件名做为一个参数传递给这些函数。

全局变量environ可以用来把一个值传递到新的程序环境中去。execle和execve能够通过比其他函数多出来的那个参数传递一个字符串数组，这个数组将被用做新的程序环境。

如果我们想通过exec函数来启动ps程序，可以有下面几种选择：

```
#include <unistd.h>

/* Example of an argument list */
/* Note that we need a program name for argv[0] */
const char *ps_argv[] =
    {"ps", "-ax", 0};

/* Example environment, not terribly useful */
const char *ps_envp[] =
    {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* Possible calls to exec functions */
execl("/bin/ps", "ps", "-ax", 0);            /* assumes ps is in /bin */
execlp("ps", "ps", "-ax", 0);               /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "-ax", 0, ps_envp); /* passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

动手试试：execlp函数

修改程序，让它使用一个execlp调用。

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

int main()
{
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-ax", 0);
    printf("Done.\n");
    exit(0);
}
```

我们给这个程序起名为pexec.c，当运行这个程序的时候，会看到正常的ps输出，但“Done.”消息却根本没有出现。另外值得注意的地方是：ps输出里根本就没有名为pexec的进程的任何资料。

```
$ ./pexec
Running ps with execlp
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
```

```
    7  ?  S      0:00 update (bdflush)
...
  146 v01 S N    0:00 oclock
  294 pp0 R      0:00 ps -ax
```

操作注释：

程序先打印出自己的第一条消息，接着调用了execlp，这个函数会在PATH环境变量给出的子目录里搜索一个名为ps的程序。找到后执行这个程序，并用它替换掉我们的pexec程序，就好像我们给出的是如下所示的shell命令一样：

```
$ ps -ax
```

当ps结束的时候，我们看到的是一个新的shell提示符。我们没有返回到pexec去，所以第二条消息是不会被打印出来的。新进程的PID与原先的完全一样，父进程PID和优先级基数也相同。从效果上看，这里发生的一切其实就是运行中的程序转去执行来自exec调用中指定的可执行文件中的新代码了。

对一个由exec函数启动进程来说，它的参数表和环境加在一起的总长度是有限制的。这个上限由ARG_MAX给出，在Linux系统上它是128K字节。其他系统可能会设置一个大大缩减了的限度，有可能导致出现问题。POSIX技术规范规定ARG_MAX至少要有4096个字节。

如果没有出现错误，exec函数一般是不返回的。在出现错误时，exec函数将返回“-1”，并且会设置错误变量errno。

由exec启动的新进程继承了原进程的许多东西。值得注意的有：已经打开了的文件描述符在新进程里仍将是打开的——除非它们的“exce调用时关闭此文件”标志被置了位（详细说明请参考第3章中对fcntl系统调用的介绍）。但原进程中任何已经打开的子目录流将被关闭。

复制一个进程映像

如果想用进程同时执行多个函数有两种办法：一是使用我们将在下一章介绍的线程；二是从原程序里创建一个全新的进程，就像init的做法一样。不能像exec调用那样用新进程替换掉当前的执行线程了。

调用fork可以创建一个全新的进程。这个系统调用对当前进程进行复制，在进程表里创建出一个新的项目，新项目的许多属性与当前进程是相同的。新进程几乎与原进程一模一样，执行是也是相同的代码，但新进程有自己的数据空间、自己的环境和自己的文件描述符。把fork与exec函数结合起来就可以创建出新的进程了。

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

在父进程中调用fork返回的是新子进程的PID。新进程将继续执行，就像原进程一样，只不过在子进程里调用fork将返回“0”。我们可以利用这一特点把父、子进程区分开如图10-2所示。

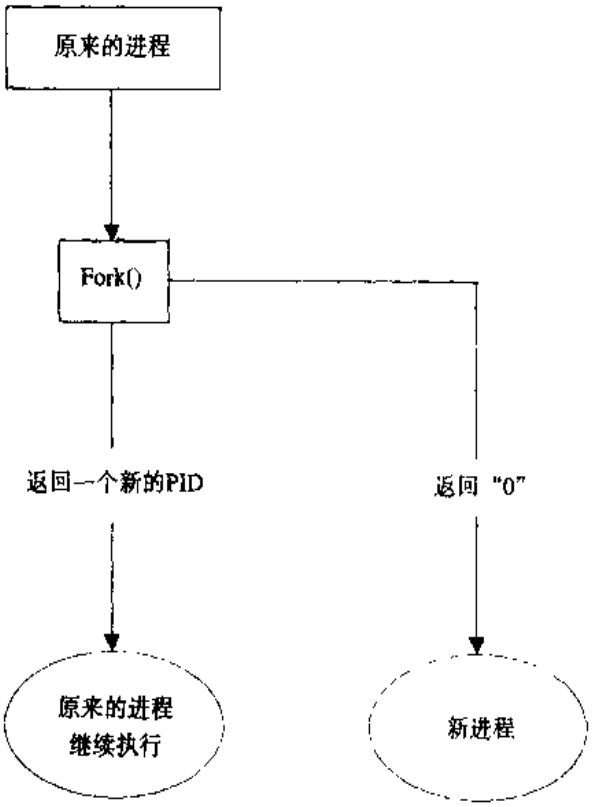


图 10-2

如果fork失败了，它将返回“-1”。失败的原因通常是父进程拥有的子进程个数超过了规定的限制（CHILD_MAX），此时errno将被设置为EAGAIN。如果失败的原因是进程表里没有足够的空间来创建一个新的项目，或者是因为虚拟内存不足，errno变量将被设置为ENOMEM。

fork的典型用法如下所示：

```
pid_t new_pid;

new_pid = fork();

switch(new_pid) {
case -1 :       /* Error */
    break;
case 0 :        /* We are child */
    break;
default :       /* We are parent */
    break;
}
```

动手试试：fork函数

请看下面这个简单的例子，fork.c程序。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid;
```

```
    char *message;
    int n;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        perror("fork failed");
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        break;
    default:
        message = "This is the parent";
        n = 3;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
    exit(0);
}
```

这个程序将产生两个进程。新创建（出生？）的子进程会输出消息五次；原进程（即父进程）只输出消息三次。父进程将在子进程打印完它的全部消息之前结束，因此我们将在输出内容里看到混杂着一个shell提示符。如下所示：

```
$ ./fork
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

操作注释：

程序在调用fork的时候被分为两个独立的进程。程序靠fork的非零返回值确定父进程，父进程设置了一个消息输出次数，两次输出之间间隔一秒。

10.3.1 等待进程

当我们用fork启动一个子进程的时候，子进程就有了自己的生命，并将独立地运行。有时候，我们需要知道某个子进程是否已经结束了。就拿刚才那个例子来说吧，父进程在子进程之前结束了，可因为子进程仍在运行，所以我们看到的程序输出不太整齐。我们可以通过调用wait安排父进程结束在子进程结束之后。

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

wait系统调用会使父进程暂停执行，直到它的一个子进程结束为止。这个调用返回的是子进程的PID，它通常是结束的子进程。状态信息允许父进程判定子进程的退出状态，即从子进程的main函数返回的值或子进程中exit语句的退出码。如果stat_loc不是一个空指针，状态信息将被写入它指向的位置。

我们可以用sys/wait.h文件里定义的宏解读状态信息。如表10-1所示：

表 10-1

宏 定 义	说 明
WIFEXITED(stat_val)	如果子进程正常结束，它就取一个非零值
WEXITSTATUS(stat_val)	如果WIFEXITED非零，它返回子进程的退出码
WIFSIGNALED(stat_val)	如果子进程因为一个未捕获的信号而终止，它就取一个非零值
WTERMSIG(stat_val)	如果WIFSIGNALED非零，它返回一个信号代码
WIFSTOPPED(stat_val)	如果子进程停止，它就取一个非零值
WSTOPSIG(stat_val)	如果WIFSTOPPED非零，它返回一个信号代码

动手试试：wait函数

我们对程序稍做修改，让它等待并检查子进程的退出状态。新程序的名字是wait.c。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
case -1:
    perror("fork failed");
    exit(1);
case 0:
    message = "This is the child";
    n = 5;
    exit_code = 37;
    break;
default:
    message = "This is the parent";
    n = 3;
    exit_code = 0;
    break;
}

for(; n > 0; n--) {
    puts(message);
    sleep(1);
}
```

程序的这一部分等待子进程的完成。

```
    if (pid != 0) {
        int stat_val;
        pid_t child_pid;

        child_pid = wait(&stat_val);

        printf("Child has finished: PID = %d\n", child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
    }
    exit(exit_code);
}
```

当我们运行这个程序的时候，我们将看到父进程等待子进程的现象：

```
$ ./wait
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 410
Child exited with code 37
$
```

操作注释：

父进程（它从fork调用得到一个非零的返回值）通过wait系统调用把自己的执行挂起，直到子进程的状态信息出现为止。这将发生在子进程调用exit的时候；我们把它的退出码设置为37。然后，父进程将继续执行，通过测试wait调用的返回值确定子进程已经正常结束，并从状态信息里提取出子进程的退出码。

10.3.2 僵进程

通过fork来创建进程确实很有用，但你必须密切注意子进程的执行情况。当一个子进程结束运行的时候，它与其父进程之间的关联还会保持到父进程也正常地结束运行或者父进程调用了wait才告终止。因此，进程表中代表子进程的数据项是不会立刻释放的，虽然不再活跃了，可子进程还停留在系统里，因为它的退出码还需要保存起来以备父进程中后续的wait调用使用。它将成为一个“僵进程”。

如果我们对fork示例程序中的消息输出次数进行修改就能看到出现僵进程的现象。如果子进程输出消息的次数比父进程少，它就会率先结束，在父进程结束之前它将成为一个僵进程。

动手试试：僵进程

fork2.c和fork.c基本一样，只是父、子进程输出消息的次数对调了一下。下面是有关的代码行。

```
switch (pid)
```

```
{
case -1:
    perror("fork failed");
    exit(1);
case 0:
    message = "This is the child";
    n = 3;
    break;
default:
    message = "This is the parent";
    n = 5;
    break;
}
```

操作注释：

如果我们用“fork2 &”命令来运行上面这个程序，然后在子进程结束之后父进程结束之前调用ps程序，就会看到下面这样一行（非Linux系统经常使用“<defunct>”而不是“<zombie>”）。

```
PID TTY STAT  TIME COMMAND
420 pp0 Z     0:00 (fork2) <zombie>
```

接下去，如果父进程非正常地结束了，子进程就会自动把PID值为1的进程（即init）当作自己的父进程。子进程现在是一个不再会运行的僵进程了，但因为它原来的父进程是非正常结束的，所以init不得不“收养”下它。这个僵进程将保留在进程表里，直到被init发现并处理掉为止，到那时它才会从进程表里被删除掉。进程表越大，这一过程就越慢。我们应该尽量避免僵进程的产生，因为在init进行清理之前，它们将一直消耗着系统的资源。

还有另外一个可以用来等待子进程的系统调用。它的名字是waitpid，你可以用它来等待某个特定进程的结束。

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

pid参数给出的是准备等待的那个子进程的PID。如果它是“-1”，waitpid将返回每一个子进程的信息。类似于wait，如果stat_loc不是一个空指针的话，waitpid也会把状态信息写到它指向的位置去。options参数允许我们改变waitpid的行为，其中最有用的一个选项是WNOHANG，它的作用是防止waitpid把调用者的执行挂起。你可以用它查出子进程是否已经结束了；如果还没有结束，程序将继续运行。其他选项与wait调用的相同。

因此，如果我们想让父进程周期性地检查某个特定的子进程是否已经结束，就可以使用下面这样的调用：

```
waitpid (child_pid,  (int *) 0, WNOHANG );
```

如果子进程尚未结束或者尚未被意外终止，它将返回零；如果子进程已经结束，就将返回child_pid。操作失败时waitpid将返回“-1”并设置errno变量。失败情况包括没有子进程（errno将被设置为ECHILD）、调用被一个信号中断了（EINTR）、选项参数不合法（EINVAL）等。

10.3.3 输入和输出重定向

已经打开的文件描述符将会在fork和exec调用中保持下来，这是一个事实。对进程这方面知识的探索使我们能够改变程序的行为。下面这个例子使用了一个过滤器程序，这种程序从它自己的标准输入读入数据，向自己的标准输出写数据，在输入和输出之间对数据要做一些必要的转换处理。

> 动手试试：重定向

下面是一个非常简单的过滤器程序filter.c，它把输入转换为大写字母。

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

int main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

编译并运行这个程序：

```
$ ./upper
hello THERE
HELLO THERE
^D
$
```

我们可以用shell的重定向在把一个文件转换为大写：

```
$ cat file.txt
this is the file, file.txt, it is all lower case.
$ upper < file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

如果我们想把这个程序用在另外一个程序里会发生什么样的事情呢？下面这个程序useupper.c，接受一个文件名做为自己的一个参数，如果对它的调用不正确，就会以一个错误信息做为响应。

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

int main(int argc, char *argv[])
{
    char *filename;

    if (argc != 2) {
        fprintf(stderr, "usage: useupper file\n");
        exit(1);
    }

    filename = argv[1];
```

我们重新打开标准输入，在这样做的同时再次检查有无错误发生，然后通过execl调用upper程序。

```
if(!freopen(filename, "r", stdin)) {
    fprintf(stderr, "could not redirect stdin from file %s\n", filename);
    exit(2);
}

execl("./upper", "upper", 0);
```

不要忘记execl会替换掉当前的进程；如果没有发生错误，下面这些语句是不会被执行到的。

```
    perror("could not exec ./upper");
    exit(3);
}
```

操作注释：

当我们运行这个程序的时候，我们可以让它把一个文件的内容全部转换为大写字母。这项工作是由upper程序完成的，但它不参与对文件名参数的处理。注意：我们不需要upper的源代码；我们可以利用这种方法运行任何可执行程序。

```
$ ./useupper file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

useupper程序通过freopen函数先关闭标准输入，再把文件流stdin与由程序参数给定的一个文件关联在一起。接下来，调用execl，用upper程序替换掉运行中的进程代码。因为打开的文件描述符会在execl调用中保持下来，所以upper程序在这里的执行情况与它在shell提示符下的执行情况是完全一样的。下面是upper做为shell命令的用法：

```
$ upper < file.txt
```

10.3.4 线程

UNIX系统里的进程能够互相合作，它们可以互相发送消息，能够互相中断，甚至能够把共享内存段的工作安排好。但从本质上来说，它们是操作系统内各自独立的实体。它们不能共享变量。

在许多UNIX和Linux系统上都有一类叫做线程（thread）的进程。涉及到线程的编程是比较困难的，但它们在某些应用软件里有着很高的价值，比如多线程数据库服务器等。Linux系统（或者更具普遍意义的UNIX系统）上的线程程序设计可不像多进程程序设计那么常见，这是因为UNIX进程相对来说还都比较简单，并且为多个合作性的进程编写程序要比为线程编写程序容易很多。线程将在下一章讨论。

10.4 信号

信号是UNIX系统响应某些状况而产生的事件，进程在接收到信号时会采取相应的行动。信号是因为某些错误条件而产生的，比如内存段冲突、浮点处理器错误或者非法指令等。它们由shell和终端管理器产生以引起中断。它们可以明确地由一个进程产生并发送给另外一个进程，用这种办法传递信息或协调操作行为。无论何种情况，程序设计的接口都是一样的。进程可以生成信号、捕捉并响应信号或屏蔽（某些不重要的）信号。信号的名称是在头文件signal.h里定

义的。它们都以“SIG”打头，见表10-2：

表 10-2

信号名称	说 明
SIGABORT	* 进程停止运行
SIGALRM	警告钟
SIGFPE	* 浮点运算例外
SIGHUP	系统挂断
SIGILL	* 非法指令
SIGINT	终端中断
SIGKILL	停止进程（此信号不能被忽略或者捕获）
SIGPIPE	向没有读者的管道写数据
SIGQUIT	终端退出
SIGSEGV	* 无效内存段访问
SIGTERM	终止
SIGUSR1	用户定义信号之一
SIGUSR2	用户定义信号之二

如果进程接收到上表中的某个信号但事先并没有安排捕捉它，进程就会立刻终止。表中带星号（*）标记的信号所对应的操作动作要视系统的具体实现而定。比较常见的办法是创建一个核心映像文件。这个文件的名字是core，放在当前子目录里，它是进程在内存中的映像，对进程的调试工作有很大意义。

• 其他信号见表10-3：

表 10-3

信号名称	说 明
SIGCHLD	子进程已经停止或退出
SIGCONT	如果被停止则继续执行
SIGSTOP	停止执行（这个信号不能被捕捉或屏蔽）
SIGTSTP	终端停止信号
SIGTTIN	后台进程请求进行读操作
SIGTTOU	后台进程请求进行写操作

SIGCHLD信号在子进程管理方面很有用。在默认的情况下，它会被屏蔽。其余信号会使接收到它们的进程停止运行，但SIGCONT信号是个例外，它的作用是让进程恢复继续执行。shell程序通过它们对作业进行管理，用户程序很少会用到它们。

我们将在稍后对前一个表里的信号做进一步的学习，现在，读者只需记住这样一件事：如果shell和终端驱动程序是按普通情况配置的话，在键盘上敲入中断字符（通常是Ctrl-C组合键）时就会向当前进程（即当前正在运行的程序）发出SIGINT信号；如果该程序事先没有安排捕捉这个信号的话就会使它结束运行。

如果我们想给非当前前台任务发送一个信号，就必须使用kill命令。这个命令有两个参数，一个是可选的信号代码或信号名称，另一个是准备向它发送信号的PID（这个PID一般需要用ps

命令查出来）。比如说，如果想向运行在另外一个终端上运行着进程ID为512的进程的shell发送一个“挂断”信号，需要使用如下所示的命令：

```
kill -HUP 512
```

kill命令还有一个有用的变体叫killall，它的作用是把一个信号发送给运行着某个特定命令的全体进程。有些UNIX版本是不支持它的，但Linux一般都没问题。如果你不知道进程的PID，或者想把一个信号同时发送给几个运行着同样命令的不同进程，这条命令就很有用了。最常见的用法是通知inetd程序重新读取它的配置选项（我们将在第14章再次遇见inetd），我们用下面这条命令完成这一工作：

```
killall -HUP inetd
```

如果想让程序能够处理信号，可以使用signal库函数，它的定义如下所示：

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

这个看起来相当复杂的函数定义说明signal是一个带sig和func两个参数的函数，准备捕捉或屏蔽的信号由参数sig给出，接收到指定信号时将要调用的函数由func给出。这个函数必须有一个int类型的参数（即接收到的信号的代码），它本身的类型是void。signal函数本身将返回一个同类型的函数——即原先用来处理这个信号的函数，或者返回下面两个特殊取值之一，见表10-4：

表 10-4

SIG_IGN	屏蔽该信号
SIG_DFL	恢复默认行为

有个例子就更容易把事情说清楚了。我们来编写一个程序ctrlc.c，它对Ctrl-C组合键的响应是输出一条消息而不是结束运行。程序将在用户第二次按下Ctrl-C组合键时结束。

动手试试：信号处理

函数ouch对通过参数sig传递来的信号作出响应。这个函数会在信号出现时被调用执行。它先输出一条消息，然后把SIGINT信号（在默认情况下，按下Ctrl-C组合键将产生这个信号）的处理设置为默认行为。

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}
```

main函数的作用是截获我们按下Ctrl-C组合键时产生的SIGINT信号。没有信号出现时，它会在一个无限循环里每隔一秒输出一个消息。

```
int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

第一次按下Ctrl-C组合键时会引起程序作出响应，然后继续执行。当我们再次按下Ctrl-C组合键时，程序将结束运行。这是因为SIGINT信号（译者注：此处原文误为SIG_ING）的处理动作已经恢复为默认的行为——让程序退出执行。

```
$ ./ctrlc
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

从这个例子可以看出，信号处理函数要用一个整数值做参数，它就是引起该函数被调用的那个信号的代码。这在同一个函数要用来处理多个信号的情况里很有用。我们打印出SIGINT的值，在这个系统上它恰好是“2”。不要过分依赖信号代码沿用下来的数字取值，在新编程序里应该永远使用信号的名称。

> 通常，当你在一个信号处理器里调用printf的时候，你需要在信号处理器里设置一个标志，并在调用结束后对这个标志进行检查。在这一章的末尾你会看到一个调用的清单，其内容是能够安全地用在信号处理器里面的各种调用。

操作注释：

我们这个程序的安排是这样的：当我们通过按下Ctrl-C组合键给出SIGINT信号时，函数ouch将被调用。程序会在中断函数ouch结束后继续执行，但信号处理动作已经被恢复为默认的行为（不同版本的UNIX，特别是从Berkley UNIX衍生出来的那些版本，不同的信号传统上都有各自明确的行为。如果你想在改变了信号处理动作并处理过信号之后把它们恢复回默认的行为，最好还是写出原来的信号处理代码）。当它接收到第二个SIGINT信号的时候，程序将采取默认的行动，也就是结束程序的运行。

如果我们想继续使用信号处理器，还让它来负责Ctrl-C组合键的处理，就必须通过再次调用signal重新建立它。这会使信号在一段时间内无法得到处理，这段时间起于中断函数的开始，止于信号处理器的重建之前。如果在这段时间里出现并接收到了第二个信号，它就会违背我们的意愿终止掉程序的运行。

我们不推荐大家使用signal接口。我们之所以会把它包括在这部分内容里是因为读者可能会在许多老程序里看到它们。我们稍后将向大家介绍一个定义更清晰，执行也更可靠的sigaction函数，在新编程序里应该坚持使用这个函数。

signal函数返回的是以前对指定信号进行处理的那个信号处理器的函数指针——如果它有的话；否则返回SIG_ERR并把errno设置为一个正数值。如果给出的是一个无效的信号，或者尝试处理的信号是不可捕捉或不可屏蔽的（比如SIGKILL），errno将被设置为EINVAL。

10.4.1 发送信号

进程可以通过调用kill向包括它本身在内的另一个进程发送信号。如果程序没有发送该信号的权限，对kill的调用就将失败，这类失败的常见原因是目标进程由另一位用户拥有。这个函数是同名shell命令的对等编程接口。请看它的定义：

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

kill函数的作用是把参数sig给定的信号发送给由参数pid给出了标识符的进程。成功时它将返回“0”。要想发送一个信号，发送者进程必须拥有相应的权限。这通常意味着两个进程必须拥有同样的用户ID，也就是说，你只能向你自己的另一个进程发送信号。但超级用户可以向任何进程发送信号。

kill调用会在失败时返回“-1”并设置errno变量。如果失败的原因是给定的信号无效，errno将被设置为EINVAL；如果是权限不足，errno将被设置为EPERM；如果指定的进程不存在，errno将被设置为ESRCH。

信号为我们提供了一个有用的闹钟功能。进程可以通过alarm函数调用安排经过预定时间后出现一个SIGALARM信号。下面是这个函数的定义：

```
# include <unistd.h>

unsigned int alarm (unsigned int seconds );
```

alarm调用的作用是在seconds秒之后安排发送一个SIGALARM信号。但由于处理上的延时和意外事件的干扰，实际闹响时间会稍微错后一点点。把参数seconds设置为0将取消全部已经设置的闹钟请求。如果在接收到SIGALARM信号之前再次调用了alarm，闹钟将重新开始计时。每个进程只能有一个可用的闹钟。alarm函数的返回值是前一个闹钟闹响之前还需经过的剩余秒数；如果调用失败将返回“-1”。

为了让大家看到alarm的工作情况，我们用fork、sleep和signal来模仿它的效果。程序可以专门启动一个新进程，让它在未来的某一时刻发送出一个信号。

动手试试：一个闹钟

1) alarm.c程序里的第一个函数ding的作用是模仿一个闹钟。

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

static int alarm_fired = 0;

void ding(int sig)
{
alarm_fired = 1;
}
```

2) 在main函数里，我们告诉子进程在等待5s后发送一个SIGALARM信号给它的父进程。

```
int main()
{
pid_t pid;

    printf("alarm application starting\n");

    pid = fork();
    switch(pid) {
    case -1:
      /* Failure */
      perror("fork failed");
      exit(1);
    case 0:
        /* child */
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }
```

3) 父进程通过一个signal调用安排好捕获SIGALARM信号的工作，然后开始等待它的到来。

```
    /* if we get here we are the parent process */
    printf("waiting for alarm to go off\n");
    (void) signal(SIGALRM, ding);

    pause();
    if (alarm_fired) printf("Ding!\n");

    printf("done\n");
    exit(0);
}
```

当我们运行这个程序的时候，它会暂停5s，等待闹钟的闹响。如下所示：

```
$ ./alarm
alarm application starting
waiting for alarm to go off
<5 second pause>
Ding!
done
$
```

这个程序里用到了一个新函数pause，它的作用很简单，就是把执行挂起直到有一个信号出现为止。当它接收到一个信号的时候，预设好的处理器都将开始运行，程序的运行也将像正常情况一样继续前进。它的定义如下所示：

```
# include <unistd.h>

int pause (void);
```

如果它自己被一个信号中断了，就会返回“-1”（如果接收到的下一个信号没有让程序结束

的话）并把errno设置为EINTR。等待信号时更常见的办法是使用我们马上就要介绍的sigsuspend函数。

操作注释：

闹钟模仿程序通过fork启动了一个新进程。这个子进程休眠5s后向自己的父进程发送一个SIGALARM信号。父进程在安排好捕捉SIGALARM信号后暂停运行，直到接收到一个信号为止。我们没有在信号处理器里直接调用printf，我们的做法是设置了一个标志，然后在父进程里检查这个标志并完成消息的输出。

使用信号和挂起进程的执行是UNIX程序设计的重要组成部分。它意味着一个程序不必总是在执行着。程序不必在一个循环里无休止地检查某个事件是否已经发生，相反，它可以等待事件的发生。这在一个多用户环境里是极其重要的，进程们共享着一个处理器，而繁忙的等待会对系统的整体性能造成极大的影响。但使用信号又会面临这样一个特殊的问题："如果一个信号出现在一个系统调用的执行过程中会发生什么事情？"答案是相当模糊的"看情况"。一般说来，你只需考虑那些比较"慢"的系统调用，比如从终端读数据等——如果在这个系统调用等待数据的时候出现了一个信号，它就会返回一个错误。如果你决定在自己的程序里使用信号，就必须警惕有些系统调用会因为接收到了一个信号而失败，而这种错误可能是你在添加信号处理之前想不到的一件事。

为信号编写程序时必须深思熟虑，因为在使用信号的程序里会出现各种各样的"竞争"现象。比如说，如果你想调用pause来等待一个信号，可信号却出现在你调用pause之前，就会使你的程序无限期地等待一个不会发生的事件。这些竞争现象都是一些对时间先后很挑剔的问题，许多初出茅庐的程序员都吃过苦头。检查信号代码时一定要认真仔细。

一个健壮的信号接口

我们已经对使用signal及其有关函数引发和捕捉信号做了比较深入的讨论，因为它们是UNIX程序里比较常见的。但X/Open和UNIX技术规范为我们推荐了一个更新更健壮的信号程序设计接口，这就是sigaction。请看它的定义：

```
# include <signal.h>

int sigaction (int sig,const struct sigaction *act,struct sigaction *oact );
```

sigaction结构是在signal.h文件里定义的，它被用来定义在接收到sig指定的信号时将要采取的操作动作，至少应该包含以下几个成员：

```
void (*) (int) sa_handler     /*  function, SIG_DFL or SIG_IGN
sigset_t sa_mask              /*  signals to block in sa_handler
int sa_flags                  /*  signal action modifiers
```

sigaction函数的作用是设置与信号sig关联着的操作动作。如果oact不是null，sigaction就将把前一个信号动作写到它指向的地方去。如果act是null，sigaction写好oact就没事了；可如果act不是null，对应于给定信号的动作就将被设置。

在对信号进行处理的时候，sigaction会在成功时返回"0"，失败时返回"-1"。如果给出的信号不合法，或者如果试图对一个不允许被捕捉或屏蔽的信号进行捕捉或屏蔽，错误变量errno

将被设置为EINVAL。

在参数act指向的sigaction结构里，函数指针sa_handler指向一个将在接收到信号sig时被调用的函数。它相当于我们前面见过的传递到signal调用中去的函数func。我们可以在sa_handler域里用特殊值SIG_IGN和SIG_DFL分别表明该信号将被屏蔽或者将把动作恢复为它的默认行为。

sa_mask域给出的是在调用由参数sa_handler指定的函数之前将被添加到该进程的信号掩码里去的一组信号。这是一组将被阻塞且不会被发送给该进程的信号。这可以防止出现信号先于它处理器的运行结束而被接收到的情况，我们在前面遇见过这样的问题。使用sa_mask域可以消除这一竞争现象。

但是，由sigaction设置的处理器所捕捉到的信号在默认的情况下是不会被重置的；如果我们希望获得类似于前面见过的用第二次signal调用对信号进行重置的效果，就必须在sa_flags域里包含上标志值SA_RESETHAND。在深入学习sigaction的细节之前，我们先把程序ctrlc.c重写一遍，用sigaction替换掉signal。

动手试试：sigaction函数

按照下面给出的代码清单修改程序，用sigaction来截获SIGINT信号。我们给新程序起名为ctrlc2.c。

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

当我们运行这个新版程序的时候，只要按下了“Ctrl-C”组合键，就会看到预定的输出信息。这是因为sigaction能够对连续到来的多个SIGINT信号做连续的处理。如果想结束这个程序，就必须按下“Ctrl-\”组合键，这个组合键的默认动作是产生SIGQUIT信号。

```
$ ./ctrlc2
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
```

```
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^\
Quit
$
```

操作注释：

这个程序用sigaction代替signal把“Ctrl-C”组合键（SIGINT信号）的信号处理器设置为函数ouch。我们需要先设置一个sigaction结构，里面包含着处理器名称、一个信号掩码和适当的标志。我们的例子不需要使用任何标志，而空白的信号掩码是用sigemptyset函数创建出来的。

运行完这个程序之后，你将发现新创建出一个名为core的文件。你可以安全地删除掉它。

10.4.2 信号集

头文件signal.h定义了sigset_t类型和用来处理信号集的函数，sigaction和其他函数将利用这些信号集来修改进程在接收到信号时的行为。

```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);
```

这些函数所完成的操作从它们的名字就能看出来。sigemptyset把一个信号集初始化为空白。sigfillset把一个信号集初始化为包含有全部的已定义信号。sigaddset和sigdelset的作用是在一个信号集里增加或删除一个给定的信号（signo）。它们在成功时都会返回“0”，失败时都会返回“-1”并设置errno。人们只为它们定义了一个错误，那就是在给定信号不合法时的EINVAL。

函数sigismember的作用是判定一个给定的信号是否是一个信号集的成员。如果该信号是该集合的一个成员，它就返回“1”；如果不是，返回“0”；如果信号不合法，返回“-1”并把errno设置为EINVAL。

```
# include <signal.h>

int sigismember (sigset_t  *set, int signo );
```

信号掩码的设置和检查工作是由sigprocmask函数完成的。信号掩码是当前被阻塞的一组信号，因此，当前进程将接收不到出现在掩码里的那些信号。

```
# include <signal.h>

int sigprocmask(int how, const sigset_t  *set,  sigset_t  *oset );
```

sigprocmask可以依据how参数指定的方法对信号掩码进行修改。信号掩码的新值（如果它不为空）由参数set传递，而以前的信号掩码将被写到信号集oset里去。

how参数的取值可以是表10-5中的一个。

表 10-5

SIG_BLOCK	把set中的信号添加到信号掩码里
SIG_SETMASK	把信号掩码设置为set中的信号
SIG_UNBLOCK	从信号掩码里去掉set中的信号

如果set参数是一个空指针，how的值就没有用处，这个调用的惟一作用就是把当前信号掩码的值取到oset里。

如果sigprocmask操作成功，它将返回“0”；如果how参数是无效的，它将返回“-1”并把errno设置为EINVAL。

如果一个信号被一个进程阻塞，就不会到达这个进程，但是会停留在待处理状态。程序可以通过调用sigpending函数查看其阻塞信号里有哪些个正停留在待处理状态。

```
# include <signal.h>
int sigpending (sigset_t *set );
```

这个函数的作用是把因阻塞而没有发送但又停留在待处理状态的一组信号写到set指向的信号集里去。成功时它将返回“0”，否则返回“-1”并设置errno指示错误的原因。如果程序需要对信号进行处理，但又需要控制处理函数的调用时间，这个函数就有用了。

进程可以通过调用sigsuspend函数挂起自己的执行，直到信号集里的一个信号到达为止。这是我们前面见过的pause函数更具普遍意义的形式。

```
# include <signal.h>
int sigsuspend (const sigset_t *sigmask );
```

sigsuspend函数的作用是先用参数sigmask给出的信号掩码替换掉进程当前的信号掩码，然后把执行挂起。它将在一个信号处理函数执行完毕之后重新开始执行。如果一个接收到的信号结束了程序，sigsuspend就不会返回。如果一个接收到的信号不结束程序，sigsuspend就返回“-1”并且把errno设置为EINTR。

1. sigaction标志

用在sigaction函数里的sigaction结构中的sa_flags域可以包含下列取值，它们的作用是改变信号的行为见表10-6：

表 10-6

SA_NOCLDSTOP	当子进程停止时不产生SIGCHLD信号
SA_RESETHAND	把信号动作设置为SIG_DFL
SA_RESTART	重新启动可中断函数而不是给出EINTR错误
SA_NODEFER	捕获时不把信号添加到信号掩码里去

当捕捉到一个信号的时候，SA_RESETHAND标志可以用来自动清除一个信号函数，我们在前面已经见过它的用法了。

用在程序中的许多函数都是可以被中断的，也就是说，当接收到一个信号的时候，它们会返回一个错误并把errno设置为EINTR以表明函数是因为一个信号而返回的。使用了信号的软件必须特别注意这一行为。如果sigaction调用里的sa_flags域中的SA_RESTART标志被置位，那么

在该信号的处理函数执行完毕之后，会立刻重新启动一个函数，新启动的函数在SA_RESTART标志没有被置位的情况下是可以被一个信号中断的。

普通的做法是：当信号处理函数正在执行的时候，又新接收到的信号将在该处理函数的执行期间被添加到该进程的信号掩码里去。这就预防了同一信号不断出现情况的发生，那样会使信号处理函数再次运行。如果这个函数不是可重入的，在它结束对第一个信号的处理之前又让第二个同样的信号再次调用它就有可能引起问题。但如果SA_NODEFER标志被置位，它接收到这个信号的时候就不会改变信号掩码。

一个信号处理函数可以在其执行期间被其他事件中断并再次被调用。当你回到第一个调用里的时候，它能否继续正确操作是很关键的。这可不是递归（调用自身），而是重入（可以安全地重新进入和执行）。操作系统内核中同时负责多个设备的中断性服务例程就需要是可重入的，因为一个优先级更高的中断可能会在同一段代码的执行期间“夹塞”进来。

下面列出的是一些能够在信号处理器内部安全使用的函数，它们或者是可重入的，或者它们本身不会再产生信号，X/Open技术规范可以保证这一点。

所有没有列在下表里的其他函数在涉及到信号问题时都被认为是不安全的。

access	alarm	cfgetispeed	cfgetospeed
cfsetispeed	cfsetospeed	chdir	chmod
chown	close	creat	dup2
dup	execle	execve	_exit
fcntl	fork	fstat	getegid
geteuid	getgid	getgroups	getpgrp
getpid	getppid	getuid	kill
link	lseek	mkdir	mkfifo
open	pathconf	pause	pipe
read	rename	rmdir	setgid
setpgid	setsid	setuid	sigaction
sigaddset	sigdelset	sigemptyset	sigfillset
sigismember	signal	sigpending	sigprocmask
sigsuspend	sleep	stat	sysconf
tcdrain	tcflow	tcflush	tcgetattr
tcgetpgrp	tcsendbreak	tcsetattr	tcsetpgrp
time	times	umask	uname
unlink	utime	wait	waitpid
write			

2. 常用信号参考

我们将在这一小节对UNIX程序常用的信号和它们的默认行为做一个总结。

表10-7里的信号其默认动作都是进程因随后的_exit调用（它类似于exit，但在返回到内核之前不做任何扫尾工作）而非正常结束。但它们的结束状态能够传递到wait和waitpid函数里去，从而指明进程是因某个特定的信号而非正常结束的。

表 10-7

信号名称	说 明
SIGALRM	由alarm函数设置的定时器产生
SIGHUP	由一个处于非连接状态的终端发送给施控进程，或者由施控进程在自身结束时发送给每一个前台进程
SIGINT	一般由从终端敲入的“Ctrl-C”组合键或预先设置好的中断字符产生
SIGKILL	因为这个信号不能被捕捉或屏蔽，所以通常是从shell用来强行结束一个异常进程
SIGPIPE	如果在向一个管道写数据时没有与之对应的读数据进程，就会产生这个信号
SIGTERM	请求其他进程结束运行。UNIX在关机时用这个信号请求系统服务停止运行。这是kill命令使用的默认信号
SIGUSR1, SIGUSR2	进程可以用这个信号进行通信，比如让进程报告状态信息等

在默认的情况下，表10-8里的信号也会引起进程的非正常结束。但可能会有一些与操作系统具体实现情况有关的额外动作，比如创建一个core文件等。

表 10-8

信号名称	说 明
SIGFPE	由浮点算术运算例外产生
SIGILL	处理器执行了一条非法指令。大多是由一个崩溃了的程序或者无效的共享内存模块引起的
SIGQUIT	一般由从终端敲入的“Ctrl-\”组合键或预先设置好的退出字符产生
SIGSEGV	内存段冲突，一般因对内存中的一个无效地址进行读写而引起，比如超越数组边界或对无效指针进行操作等。当一个函数返回到一个非法地址的时候，覆盖一个局部数组变量和引起堆栈崩溃都会引发一个SIGSEGV信号

在默认的情况下，进程接收到列在表10-9里的一个信号时会被挂起。

表 10-9

信号名称	说 明
SIGSTOP	停止执行（不能被捕捉或屏蔽）
SIGTSTP	终端停止信号，通常因按下“Ctrl-C”组合键产生
SIGTTIN, SIGTTOU	shell用这两个信号表明后台作业因需要从终端读取输入或产生输出而停止运行

SIGCONT信号的作用是让进程重新开始继续执行，如果进程没有停止，就将忽略它。SIGCHLD信号在默认情况下总是被忽略的见表10-10。

表 10-10

信号名称	说 明
SIGCONT	如果是停止的，就开始继续执行
SIGCHLD	子进程停止或退出时产生

10.5 本章总结

在这一章里，我们对进程这一UNIX操作系统的基石进行了学习。我们学习了如何启动进程，如何结束进程和如何查看进程；并且用它们解决了一些程序设计问题。

我们还对信号这种可以用来控制程序运行行为的事件进行了研究。我们看到，各种UNIX功能，包括init在内，都使用着同样的系统调用，每一个程序员都可以利用它们来开发自己的程序。

第11章 POSIX线程

在上一章里，我们学习如何在Linux（事实上是UNIX）中对进程进行处理。UNIX类别的操作系统早就具备这些多进程功能了。但有时人们认为用fork来创建一个新进程的代价还是太大。如果能让一个程序同时干两件事，或者至少看起来如此，岂不是更有用。而且，也许用户正希望两件或更多件事情以一种更密切的关系同时发生呢。

11.1 什么是线程

在一个程序里的多个执行路线就叫做线程（thread）。更准确的定义是：线程是“一个进程内部的一个控制序列”。虽然Linux和其他一些操作系统一样很擅长同时运行多个进程，但我们到目前为止见过的所有程序在执行时都被看做是一个进程。事实上，一切进程都至少有一个执行线程。我们到目前为止见过的所有进程都只有一个执行线程。

fork系统调用和创建新线程是有区别的，弄清楚这一点很重要。当一个进程执行一个fork调用的时候，会创建出进程的一个新拷贝，新进程将拥有它自己的变量和它自己的PID。这个新进程的运行时间是独立的，它在执行时（通常）几乎完全独立于创建它的进程。而当我们在进程里创建一个新线程的时候，新的执行线程会拥有自己的堆栈（因此也就有自己的局部变量），但要与它的创建者共享全局变量、文件描述符、信号处理器和当前的子目录状态。

线程的概念已经出现很长时间了，但在IEEE POSIX委员会发布有关标准之前，提供了线程功能的UNIX类操作系统并不是很多；而具备线程功能的实现版本在具体做法上也会因开发商的不同而有所差异。POSIX 1003.1c技术规范的发布把这一切都改变了；它不仅把线程的标准化工作向前推进了一大步，也使线程普遍出现在大多数Linux发行版本上。

线程的优点和缺点

在某些情况下，创建一个新线程要比创建一个新进程有更鲜明的优势。而创建一个新线程的代价要比创建一个新进程小得多。（与其他一些操作系统相比，Linux在创建新进程方面的效率是很高的。）

有时候，让一个程序看起来好象是在同时做两件事情是很有用的。经典例子是在编辑文本的同时对文档中的单词个数进行统计。一个线程负责处理用户的输入并进行着文本的编辑工作，另一个则不断地刷新着一个单词计数器变量，两个线程都能看见整个文档的内容。第一个线程（甚至可以是第三个线程）通过这个共享的计数器变量让用户能够随时了解自己工作的进展情况。另一个更具现实意义的例子是多线程的数据库服务器，这是一个明显的单进程服务多客户的情况，它会在响应某些服务请求的同时阻塞另外一些请求使之等待磁盘操作，用这种办法改善整体上的数据吞吐量。对数据库服务器来说，这个明显的多任务工作如果用多进程的办法来完成

是很难做到高效率低成本的，因为各个不同的进程必须合作得非常紧密才能满足数据封锁和数据稳定性方面的要求。可这个工作要是用多线程来完成，就会比多进程容易多了。

线程也有不足之处。因为它们相对来说还算是新生事物，所以它们不如久经考验的功能那样具有广泛的群众基础。编写多线程程序需要更全面更深入的思考。在一个多线程程序里，因时间分配上的细微偏差或者因共享了不该共享的变量而造成不良影响的可能性是很大的。调试一个多线程程序也比调试一个单线程程序困难得多。

一个把大量计算分成两个部分并把这两个部分运行为两个线程的程序在一台单处理器机器上并不见得运行得更快，因为总的计算量还是那么多，并没有因此而减少！

尽管如此，线程还是有大显身手的地方的。比如说，一个混杂着输入、计算和输出的应用程序就会因把这几个部分各自运行为三个线程而大大改善其整体性能。当输入或输出线程在等待数据连接的时候，另外一个线程可以继续执行。因此，如果一个应用程序在任一时刻最多只能做一项工作，我们就可以让它在等待数据连接之类的东西时另外做一些有用的事。

与进程之间的切换相比，线程之间的切换需要操作系统做的工作至少在理论上要少很多——这意味着线程占用的资源要比进程少很多；并且，如果程序自身的逻辑确实需要有多个执行线程，在单处理器系统上把它运行为一个多线程程序也更符合实际情况。但在实践上，操作系统为切换线程而做的工作就不一定总是比切换进程时少了。就拿Linux系统来说吧，它用一个clone系统调用来实现线程，进程里的一个新线程与一个新进程其实是很相似的。对大多数用户来说，软件只要好用就行，它具体是怎样实现的并不重要。在Linux系统里，使用线程可以节约进程方面的开销可不是一个很站得住脚的观点。请大家注意这一点。

我们将在这一章里学习以下内容：

- 检查你的计算机平台能否支持线程。
- 在进程里创建新线程。
- 在同一进程里的各个线程之间同步它们访问的数据。
- 改变一个线程的属性。
- 从一个线程控制另一个线程，而这两个线程都属于同一个进程。

11.2 检查有无线程支持

在使用线程之前，先要确定对线程的支持确实存在并且符合POSIX标准，这是很有必要的。C语言编译器在编译代码的时候会设置一些内部使用的常数，而包括在程序中的头文件还会再添加一些。通过在编译代码时检查其中都有些什么样定义，我们就可以对编译器和函数库有一个比较全面的认识，我们在这里最感兴趣的是它们对线程的支持情况。

有两种办法可以查出这一点，一种复杂点儿，另一种简单些。复杂的办法是分析几个头文件，包括limits.h、unistd.h和（Linux系统专用的）features.h等。简单的办法是编写一个很短的小程序，让它替我们查。

我们当然要用简单的办法！我们想找的定义是_POSIX_VERSION的值。如果它根本就没定义，就表示系统中根本就没有POSIX支持。如果它有定义，它的不同取值将告诉我们系统提供了什么级别的支持。一个等于或大于“199506L”的值表示系统提供了对线程的完整支持。如果

_POSIX_VERSION的值小于这个数，就表示只有部分线程支持，可能还能应付本章中的大部分示例。

动手试试：POSIX兼容性测试

我们来检查系统对POSIX支持的级别。下面是程序thread1.c的代码清单。除了在编译程序时使用的函数库里查验系统对线程的支持情况以外，它就没有其他功能了。

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

int main() {
    printf("POSIX version is set to %ld\n", _POSIX_VERSION);
    if (_POSIX_VERSION < 199506L) {
        if (_POSIX_C_SOURCE >= 199506L) {
            printf("Sorry, you system does not support POSIX1003.1c threads\n");
        }
        else {
            printf("Try again with -D_POSIX_C_SOURCE=199506L\n");
        }
    }
    else {
        printf("Your system supports POSIX1003.1c threads,\n");
        #ifdef _POSIX_THREAD_PRIORITY_SCHEDULING
            printf("including support for priority scheduling\n");
        #else
            printf("but does not support priority scheduling\n");
        #endif
    }
    exit(EXIT_SUCCESS);
}
```

我们用include语句包括了几个标准的头文件，然后检查_POSIX_VERSION的值。在Linux系统上，编译器通常会把_POSIX_C_SOURCE自动定义为一个大于或等于“199506L”的值，这样才会设置其他的定义。如果对_POSIX_VERSION的检查失败了，我们就要检查_POSIX_C_SOURCE的值，而编译器也确实设置了这个值。在两位作者的机器上，程序运行的结果都是下面这样的：

```
$ ./thread1
POSIX version is set to 199506
Your system supports POSIX1003.1c threads,
including support for priority scheduling
```

如果读者的系统没有报告出支持，或者程序根本就不能编译，就需要在命令行上设置_POSIX_C_SOURCE标志后再次编译这个程序，如下所示：

```
$ cc -D_POSIX_C_SOURCE=199506L thread1.c -o thread1
```

如果读者的系统还是没有报告出支持POSIX1003.1c线程，那你可能就无法测试本章给出的程序示例了，或者它们的工作情况将与我们预计的不一样。

操作注释：

用include语句包括上几个标准的头文件后，程序在代码编译时就会对有关定义是否存在进

行检查。

11.3 第一个线程程序

与线程有关的函数库构成了一个完整的系列，绝大多数函数的名字都是以“pthread_”打头的。为了使用这些函数库，我们必须对_REENTRANT宏进行定义，这个宏包括在头文件pthread.h文件里，链接这些线程函数库时要使用编译器命令的“-lpthread”选项。

在设计最初的库例程时，其大前提是假设每个进程里都只有一个执行线程。最明显的例子就是errno，如果对某个函数的调用失败了，人们就会用这个变量来检索错误信息。在一个多线程程序里，缺省的设置情况是只有一个errno变量供所有的线程共享。在一个线程有机会检索刚才的错误代码之前，这个变量很容易被另一个线程中的一个调用改变。类似的问题还存在于fputs等函数，这些函数通常只使用一个全局性的区域来缓冲保存输出数据。

我们需要一些“可重入”的例程。可重入代码能够被多次调用，这些调用既可以来自不同的线程，也可以是某种形式的嵌套性调用；但调用次数并不会影响它的正常功能。因此，代码的可重入部分通常只使用局部变量，这就使对这部分代码的每一个调用都有它们自己独一无二的一份数据。

我们必须在多线程程序里告诉编译器我们将要用到这些功能，这就需要在程序里的任何“#include”语句之前加上对_REENTRANT宏的定义。它会为我们做三件事，并且能够做得很完美，以致于我们一般都不需要知道它到底做了哪些事。它会对部分函数重新进行定义，让它们能够安全地运行在可重入工作方式下。这些函数的名字一般不会发生变化，只是会追加上一个“_r”记号；比如说，gethostbyname就将被改为gethostbyname_r。原来以宏定义形式实现的某些stdio.h函数会相应地变成可安全重入的函数。在errno.h里定义的errno变量现在成为一个函数调用，它能够以一种安全的多线程方式检索出真正的errno值。

包括在程序里的头文件pthread.h为我们提供了其他一些常数定义和函数声明，它们与头文件stdio.h为标准输入和标准输出所提供的东西作用相同，在编写多线程代码时一定用使用这些新的定义和声明。最后，我们要用POSIX的pthread库来链接多线程程序，与线程有关的各种函数都是在这个库里实现的。

pthread_create函数的作用是创建一个新线程，类似于创建一个新进程的fork。

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);
```

这看起来挺吓人的，但用起来其实挺简单。这个函数的第一个参数是一个指针，它指向一个pthread_t类型的数据。在创建一个线程的时候，这个指针指向的变量里会写入一个标识符，我们就用这个标识符来引用新线程。接下来的参数对程序的属性进行了设置。我们一般不需要什么特殊的属性，所以可以简单地把NULL传递给这个参数。我们将在本章后面的内容里向大家介绍这些属性的作用和用法。最后两个参数分别是线程将要启动执行的函数以及将要传递给这个函数的参数。“void *(*start_routine) (void *)”表示需要我们传递的是一个函数的地址，该函数以一个指向void的指针为参数，返回的也是一个指向void的指针。这样，我们就能传递任意类

型的单个参数并返回一个任意类型的指针了。对由fork函数创建的子进程来说，它执行的代码与父进程的完全一样，只不过将返回一个不同的值而已；但对一个新线程来说，我们必须明确地为它提供一个函数指针，新线程执行的是另外一些代码。

调用成功时返回值是“0”，如果失败则返回一个错误。使用手册页对这个函数以及将要在本章里介绍的其他函数的错误条件有详细的说明。

> pthread_create和大多数与线程有关的函数在操作失败时不返回“-1”，这在UNIX函数里是不多见的。除非你很有把握，在对错误代码进行检查之前一定要认真研究使用手册里的有关内容。

线程在结束时必须调用pthread_exit函数，这与一个进程在结束时要调用exit是同样的道理。它的作用是结束调用了这个函数的线程，返回一个指向某个对象的指针。绝不要用它返回一个指向一个局部变量的指针，因为局部变量会在线程出现严重问题时消失得无影无踪。

pthread_exit函数的定义如下所示：

```
# include <pthread.h>
void pthread_exit (void *retval );
```

pthread_join相当于进程用来等待子进程的wait函数，它的作用是在线程结束后把它们归并到一起。这个函数的定义如下所示：

```
# include <pthread.h>
int pthread_join (pthread_t th, void **thread_return );
```

第一个参数指定了将要等待的线程，它就是pthread_create返回的那个标识符。第二个参数是一个指针，它指向另外一个指针，而后者指向线程的返回值。这个函数在成功时返回“0”，失败时返回一个错误代码，与pthread_creat类似。

动手试试：一个简单的线程化程序

这个程序将创建出一个新的线程。新线程与老线程共享变量，并在结束时向老线程返回一个结果。真正的多线程程序可没有这么简单！下面是thread2.c的代码清单。

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

void *thread_function(void *arg);

char message[] = "Hello World";

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
```

```
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined, it returned %s\n", (char *)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for the CPU time");
}
```

在对这个程序进行编译的时候，我们必须先定义_REENTRANT宏；如有必要，还必须定义_POSIX_C_SOURCE。在作者的系统上，我们只需定义_REENTRANT宏，所以编译命令相对简单些，如下所示：

```
$ cc -D_REENTRANT thread2.c -o thread2 -lpthread
```

当我们运行这个程序的时候，我们将看到：

```
$ ./thread2
Waiting for thread to finish...
thread_function is running. Argument was Hello World
Thread joined, it returned Thank you for the CPU time
Message is now Bye!
```

操作注释：

我们先对创建线程时将要由它调用的函数进行预定义，如下所示：

```
void *thread_function (void *arg );
```

根据pthread_create的要求，上面这个函数只需要一个指向void的指针做参数，返回的也将是一个指向void的指针。这个函数的实际定义（和具体内容）过会儿再说。

我们在main里定义了几个变量，然后调用pthread_create让我们的新线程开始执行。如下所示：

```
pthread_t a_thread;
void *thread_result;

res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
```

我们向pthread_create传递了一个pthread_t类型对象的地址，对新线程的引用以后就要全靠它了。我们不想改变线程的缺省属性，所以我们把NULL做为第二个参数传递过去。最后两个参数是将要调用的函数和一个准备传递给这个函数的参数。

如果这个调用成功了，就会有两个线程在运行。原先的老线程（即main）将继续去执行pthread_create后面的代码，而那个新线程将去执行我们起名为thread_function的函数。

接下来，在查明新线程已经启动之后，老线程将调用pthread_join，如下所示：

```
res = pthread_join (a_thread, &thread_result );
```

我们给它传去两个参数，一个是我们正在等待其结束的线程的标识符，另一个是指向返回

值的指针。这个函数会等到其他线程结束后才会返回。接下来，老线程把新线程的返回值和全局变量message的值打印出来，然后退出。

新线程在thread_function函数里开始执行。它先打印出自己的参数，再休眠一小会儿，然后修改全局变量，最后退出并向主线程返回一个字符串。新线程对数组message进行了写操作，而这个数组老线程也能访问。如果我们调用的是fork而不是pthread_create，就不会有这样的效果。

11.4 同时执行

我们接下来要编写的程序将证明两个线程的执行是同时进行的（当然，如果是在一个单处理器系统上，这两个线程就要靠CPU的快速切换才能实现“同时执行”的效果）。因为我们还没有介绍过任何能够帮助我们完成这一工作的线程同步函数，所以这个程序的效率是很低的，它需要在两个线程之间不停地“跳跃”才能实现我们的想法。我们的程序仍然要利用这一事实：除局部函数变量以外，同一进程的多个线程将共享一切。

动手试试：两个线程的同时执行

thread3.c是在对thread2.c稍加修改的基础上编写出来的。我们额外增加了一个文件范围变量来确定哪个线程正在运行中。如下所示：

```
int run_now = 1;
```

我们将在执行main函数时把run_now设置为“1”，执行新线程时设置为“2”。

我们在main函数创建新线程的语句后面加上下面这些代码。

```
int print_count1 = 0;

while(print_count1++ < 20) {
    if (run_now == 1) {
        printf("1");
        run_now = 2;
    }
    else {
        sleep(1);
    }
}
```

如果run_now是1，我们就输出“1”并把它设置为2；否则，我们就稍做休眠然后再检查这个值。我们不断地检测下去，等待这个值被修改为1。尽管我们在两次检查之间要休眠一秒钟，这还是一种“繁忙的等待”。我们在后面会介绍一种更好的等待办法。

在新线程将要执行的thread_function函数里，我们要干的事与上面说的大同小异，只是把run_now的值颠倒了一下。如下所示：

```
int print_count2 = 0;

while(print_count2++ < 20) {
    if (run_now == 2) {
        printf("2");
        run_now = 1;
    }
    else {
        sleep(1);
```

```
    }
}
```

我们去掉了参数的传递和返回值的传递，因为我们现在不需要它们了。

当我们运行这个程序的时候，我们将看到如下所示的输出结果（读者将会发现程序要过几秒钟才会产生输出）：

```
$ cc -D_REENTRANT thread3.c -o thread3 -lpthread
$ ./thread3
12121212121212121212
Waiting for thread to finish...
Thread joined
```

操作注释：

两个线程都是用设置run_now变量的办法通知另一个线程去运行的，然后，它们会等到另一个线程修改了这个变量的值以后才会再次继续运行。我们可以清楚地看到：执行在两个线程之间自动地交替，而这两个线程共享着run_now变量。

11.5 同步

我们看到两个线程同时运行的情况了，但我们在它们之间进行切换的办法却是既笨拙又低效。幸好专门有一组函数可以帮助我们更好地控制线程的执行情况和更好地访问代码的关键部分。

我们将在这一小节学到两种基本的方法。第一种是“信号量”，它的作用就象是代码周围的门卫；第二种是“互斥量”，它的作用就象是一把多人共用的钥匙，谁拿着这把钥匙谁就可以访问受保护的代码段。

这两种方法其实是很相似的（事实上，两种办法可以互相实现），但在实践中这两种方法各有侧重，有时候这种方法效果好，有时候那种方法更简单。就拿控制共享内存这个问题来说吧，任一时刻只有一个线程能够对它进行访问，这时候用互斥量就自然得多。但在需要控制一组同等对象的访问权限时——比如从五条电话线里给某个线程分配一条的情况，计数信号量就更合适一些。具体选用哪一种取决于读者的个人偏好和最适合读者程序的机制。

11.5.1 用信号量进行同步

> 信号量有两组程序设计接口函数。一种来源于POSIX技术规范的实时扩展方案（POSIX Realtime Extension），常用于线程。另一种叫做System V信号量，常用于进程的同步（我们将在后面的章节里遇见第二类信号量）。这两组接口函数的可交换性并没有保证；而且虽然两者非常相近，但它们使用的函数调用却各不相同。

我们来看看最简单的一种信号量，一种只有“0”、“1”两种取值的二进制信号量。更具普遍意义的信号量是存在的，比如说，一个计数信号量就可以有更大的取值范围。信号量经常被用来保护一段代码，使它每次只能被一个执行线程运行；这种工作就需要一个二进制开关。有时候，我们希望有限个线程去执行一段给定的代码，这就需要用到关于计数信号量。信号量开

关是二进制信号量的一种逻辑扩展，两者实际调用的函数都是一样的。但计数信号量终归比较少见，所以我们就不在这里对它做过多的讨论了。

信号量函数的名字都以“sem_”打头，不像大多数与线程有关的函数那样以“pthread_”开始。线程里使用的基本信号量函数有四个。它们都相当简单。

信号量用sem_init函数创建的，下面是它的定义声明：

```
# include <semaphore.h>
int sem_init (sem_t *sem, int pshared, unsigned int value );
```

这个函数的作用是对由sem指定的信号量进行初始化，设置好它的共享选项（马上就要介绍到这些选项了），并给它一个整数类型的初始值。pshared参数控制着信号量的类型。如果pshared的值是0，就表示它是当前进程的局部信号量；否则，其他进程就能够共享这个信号量。我们现在只对不让进程共享的信号量感兴趣。在我们编写本书的时候，Linux还不支持这种共享，给pshared传递一个非零值将会使函数调用失败。

接下来的两个函数控制着信号量的值，它们的定义如下所示：

```
#include <semaphore.h>
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
```

这两个函数都要用一个由sem_init调用初始化的信号量对象的指针做参数。

sem_post函数的作用是给信号量的值加上一个“1”，它是一个“原子操作”——即同时对同一个信号量做加“1”操作的两个线程是不会冲突的；而同时对同一个文件进行读、加和写操作的两个程序就有可能会引起冲突。信号量的值永远会正确地加上一个“2”——因为有两个线程试图改变它。

sem_wait函数也是一个原子操作，它的作用是从信号量的值减去一个“1”，但它永远会先等待该信号量为一个非零值才开始做减法。也就是说，如果你对一个值为2的信号量调用sem_wait，线程将会继续执行，但信号量的值将减到1。如果对一个值为0的信号量调用sem_wait，这个函数就会等待直到有其他线程增加了这个值使它不再是0为止。如果有两个线程都在sem_wait中等待同一个信号量变成非零值，那么当它被第三个线程增加一个“1”时，等待线程中只有一个能够对信号量做减法并继续执行，另外一个还将处于等待状态。

信号量这种“只用一个函数就能原子化地测试和设置”的能力正是它的价值所在。还有另外一个信号量函数sem_trywait，它是sem_wait的非阻塞搭档。我们就不在这里对它做进一步介绍了，详细资料请读者去自行查阅使用手册页。

最后一个信号量函数是sem_destroy。这个函数的作用是在我们用完信号量后对它进行清理。下面是它的定义声明：

```
# include <semaphore.h>
int sem_destroy (sem_t * sem );
```

与前几个函数一样，这个函数也使用一个信号量指针做参数，归还自己占用的一切资源。在清理信号量的时候如果还有线程在等待它，用户就会收到一个错误。

和大多数Linux函数一样，这些函数在成功时都返回“0”。

动手试试：一个线程信号量

在这个thread4.c程序也是以thread2.c为基础的，因为改动的地方比较多，所以我们把它的完整清单列在下面。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = sem_init(&bin_sem, 0, 0);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Input some text. Enter 'end' to finish\n");
    while(strncmp("end", work_area, 3) != 0) {
        fgets(work_area, WORK_SIZE, stdin);
        sem_post(&bin_sem);
    }
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sem_wait(&bin_sem);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        sem_wait(&bin_sem);
    }
    pthread_exit(NULL);
}
```

第一个重要的改动是包括上了semaphore.h头文件，它使我们能够去访问信号量函数。随后我们定义了一个信号量和几个变量，并在创建新线程之前对信号量进行了初始化。如下所示：

```
sem_t bin_sem;
```

```
#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = sem_init(&bin_sem, 0, 0);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }
```

注意：我们把这个信号量的初始值设置为0。

在main函数里，当我们启动了新线程之后，我们从键盘读进一些文本并把它们放到工作区work_area数组里去，然后用sem_post函数给我们的信号量加上一个“1”。如下所示：

```
printf("Input some text. Enter 'end' to finish\n");
while(strncmp("end", work_area, 3) != 0) {
    fgets(work_area, WORK_SIZE, stdin);
    sem_post(&bin_sem);
}
```

在新线程里，我们等待信号量，然后统计来自输入的字符个数。如下所示：

```
sem_wait(&bin_sem);
while(strncmp("end", work_area, 3) != 0) {
  printf("You input %d characters\n", strlen(work_area) -1);
  sem_wait(&bin_sem);
}
```

在设置信号量的同时，我们等待着键盘的输入。当键盘输入来到的时候，我们释放信号量，在第一个线程再次读取键盘输入之前让第二个线程先统计出字符的个数。

这两个线程共享着work_area数组。为了让代码示例更简洁更容易理解，我们还省略了一些错误检查步骤，比如说，我们没有检查sem_wait的返回值。但在成品代码里，除非有特别充足的理由说可以省略这些错误检查，否则我们必须对函数调用的返回值进行错误检查。

我们来运行这个程序。

```
$ cc -D_REENTRANT thread4.c -o thread4 -lpthread
$./thread4
Input some text. Enter 'end' to finish
The Wasp Factory
You input 16 characters
Iain Banks
You input 10 characters
end

Waiting for thread to finish...
Thread joined
```

在线程化的程序里，定时方面的错误查找起来总是特别困难的，但这个程序似乎对快速的键盘输入和比较悠闲的暂停都挺适应。

操作注释：

在对信号量进行初始化的时候，我们把它的值设置为“0”。这样，当我们的线程函数启动

时，sem_wait调用就会被阻塞并等待信号量变为非零值。

在main线程里，我们会等到有文本输入，然后用sem_post给信号量加上一个“1”，这就立刻使另一个线程从sem_wait等待中返回并开始执行。在统计完字符个数之后，它又再次调用sem_wait并再次被阻塞，直到main线程再次调用sem_post增加了信号量的值为止。

这个程序没有考虑精细的定时偏差。我们对它稍加修改并另存为thread4a.c，我们想让来自键盘的输入偶尔被事先准备好的文字自动地替换掉。我们把main中的读数据循环修改为如下所示的样子：

```
printf("Input some text. Enter 'end' to finish\n");
while(strncmp("end", work_area, 3) != 0) {
  if (strncmp(work_area, "FAST", 4) == 0) {
    sem_post(&bin_sem);
    strcpy(work_area, "Wheeee...");
  } else {
    fgets(work_area, WORK_SIZE, stdin);
  }
  sem_post(&bin_sem);
}
```

现在，如果我们输入“FAST”，程序就会调用sem_post使字符统计线程开始执行，同时立刻用其他一些改掉work_area中的内容。程序运行结果如下所示：

```
$ cc -D_REENTRANT thread4a.c -o thread4a -lpthread
$ ./thread4a
Input some text. Enter 'end' to finish
Excession
You input 9 characters
FAST
You input 7 characters
You input 7 characters
You input 7 characters
end

Waiting for thread to finish...
Thread joined
```

程序正常工作的大前提是：程序接收文本输入的时间间隔要足够长，这样另一个线程才有时间在主线程还来不及再多给它一些单词去统计之前统计出工作区里字符的个数。如果我们尝试连续快速地给它两组不同的单词去统计（键盘输入的“FAST”和程序自动给出的“Weee...”），第二个线程就没有时间去执行。但信号量的加“1”操作可不止一次，所以字符统计线程就会反复统计工作区里的字符并对信号量做减法，直到它再次变成0为止。

这个例子告诉我们，对多线程程序里的时间安排应该注意、注意、再注意。刚才这个漏洞是很容易修补的，我们可以再增加一个信号量，让主线程等到统计线程完成字符个数的统计工作之后再继续前进。

11.5.2 用互斥量进行同步

另一种用在多线程程序中的同步访问手段是使用互斥量。程序员给某个对象加上一把“锁”，每次只允许一个线程去访问它。如果想对代码关键部分的访问进行控制，你必须在进入这段代码之前锁定一把互斥量，在完成操作之后再打开它。

使用互斥量要用到的基本函数与信号量需要使用的函数很相似。它们的定义声明如下所示：

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex),
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

通常，成功时将返回“0”，失败时将返回一个错误代码。但这些函数不设置errno，你必须对错误返回码进行检查。

类似于使用信号量时的做法，这些函数的参数都是一个预先声明了的对象的指针，但这次它将是pthread_mutex _t类型的。pthread_mutex _init函数中的属性参数允许我们设定互斥量的属性，这些属性控制着它的行为。缺省的属性类型是“fast”。它有一个小缺陷：如果你的程序试图对一个已经加了锁的互斥量调用pthread_mutex _lock，程序本身就会被阻塞；而因为拥有互斥量的那个线程现在也是被阻塞的线程之一，所以互斥量就永远也打不开了，程序将进入死锁状态。这可以通过改变互斥量的属性来解决，有两种做法：一是让它检测有可能发生死锁的这种现象并返回一个错误；二是让它递归地操作，允许同一个线程加上好几把锁，但前提是以后必须有同等数量的解锁钥匙。

互斥量属性的设置问题超出了本书的讨论范围，所以我们这里就简单地把NULL传递给属性指针，使用其缺省的属性。在pthread_mutex _init的使用手册页里你会找到与改变属性操作有关的详细资料。

动手试试：一个线程互斥量

这个程序也以thread2.c为蓝本，但改动的地方相当多。这次的情况是我们非常关心程序对一些关键性变量的访问，因此我们用一个互斥量来保证在任一时刻只能有一个线程去访问它们。为了让这份示例代码容易阅读，我们省略了几处应该对互斥量加、解锁调用的返回值进行的错误检查。在成品代码里，对这返回值的检查是不能少的。下面就是我们新程序thread5.c的代码清单。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex; /* protects both work_area and time_to_exit */
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_mutex_init(&work_mutex, NULL);
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
```

```
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_lock(&work_mutex);
    printf("Input some text. Enter 'end' to finish\n");
    while(!time_to_exit) {
        fgets(work_area, WORK_SIZE, stdin);
        pthread_mutex_unlock(&work_mutex);
        while(1) {
            pthread_mutex_lock(&work_mutex);
            if (work_area[0] != '\0') {
                pthread_mutex_unlock(&work_mutex);
                sleep(1);
            }
            else {
                break;
            }
        }
    }
    pthread_mutex_unlock(&work_mutex);
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

```
$ cc -D_REENTRANT thread5.c -o thread5 -lpthread
$ ./thread5
Input some text. Enter 'end' to finish
Whit
You input 4 characters
The Crow Road
You input 13 characters
end

Waiting for thread to finish...
Thread joined
```

操作注释：

我们在程序的开始对互斥量、我们的工作区、还有time_to_exit变量进行定义和声明，如下

所示：

```
pthread_mutex_t work_mutex; /* protects both work_area and time_to_exit */

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;
```

然后对互斥量进行初始化。如下所示：

```
res = pthread_mutex_init(&work_mutex, NULL);
if (res != 0) {
    perror("Mutex initialization failed");
    exit(EXIT_FAILURE);
}
```

接着启动新线程。下面是将要在线程函数里执行的代码。

```
pthread_mutex_lock(&work_mutex);
while(strncmp("end", work_area, 3) != 0) {
    printf("You input %d characters\n", strlen(work_area) -1);
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while (work_area[0] == '\0' ) {
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
    }
}
time_to_exit = 1;
work_area[0] = '\0';
pthread_mutex_unlock(&work_mutex);
```

新线程一上来先试图对互斥量进行加锁。如果它已经被锁上了，调用将被阻塞到它释放为止。一旦有了访问权，我们将先检查是否有退出程序的请求。如果有退出程序的请求，我们就简单地设置好time_to_exit变量，再把工作区里的第一个字符设置为“\0”，然后退出程序。

如果我们还不想退出，就对字符个数进行统计，然后把工作区里的第一个字符设置为一个空字符null。我们用把第一个字符设置为空字符的办法通知读输入程序，该程序表明我们已经完成了字符统计工作。接下来，我们对互斥量进行解锁并等待主线程的运行。我们将周期性地尝试给互斥量加锁，如果加锁成功，就检查主线程是否又给了我们送来另外一些字符去统计。如果还没有，我们就解开互斥量继续等待；如果有了，我们就统计那些字符并再次进入刚才的循环。

下面是主线程的代码清单：

```
pthread_mutex_lock(&work_mutex);
printf("Input some text. Enter 'end' to finish\n");
while(!time_to_exit) {
    fgets(work_area, WORK_SIZE, stdin);
    pthread_mutex_unlock(&work_mutex);
    while(1) {
        pthread_mutex_lock(&work_mutex);
        if (work_area[0] != '\0') {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
        }
        else {
            break;
        }
```

```
        }
    }
    pthread_mutex_unlock(&work_mutex);
```

这段代码很面熟。我们给工作区加上锁，把文本读到它里面，然后给它解锁使它允许被其他线程访问并对其中的字符个数进行统计。我们周期性地对互斥量再进行加锁，检查是否已经统计完了（work_area[0]是一个空字符时表示统计工作做完了），如果还需要等待就释放互斥量。正如我们前面已经说过的，好的程序设计一般不会出现这种繁忙的跳跃式检查，在现实世界里，我们应该尽可能地用一个信号量来避免出现这样的情况。这段代码只是用在这里做示范罢了。

11.6 线程的属性

我们在前面的内容里对线程的属性都是一笔带过的，而在这一小节我们将专门对它进行讨论。线程的许多属性都能够被程序员控制，但我们将只对一些你最可能用到的进行介绍。其他属性的详细资料可以在使用手册页里查到。

在我们前面的程序示例里，我们都要在允许程序退出运行之前用pthread_join把各个线程都归并到一起。如果我们想让线程向创建它的那个线程返回数据，就必须这样做。可有的时候，我们既不需要第二个线程向主线程返回信息，也不想让主线程等待它。

假设我们在主线程继续向用户提供服务的同时创建了一个新线程，新线程的作用是对用户正在编辑的一个数据文件进行备份存储。备份工作完成后，那第二个线程直接结束就行了，它没有必要再归并到主线程去。

我们可以创建出这种行为的线程来。它们叫做“脱离线程”，创建它们的办法有两种，一种是修改线程的属性，一种是调用pthread_detach。因为我们的目的是介绍属性的用法，所以我们将使用前一种办法。

我们需要用到的最重要的函数是pthread_attr_init，它的作用是对一个线程属性对象进行初始化。请看它的定义：

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

这个函数在成功时会返回“0”，在失败时会返回一个错误代码。

还有一个反操作函数pthread_attr_destroy，但在我们编写这本书的时候，它在Linux里的实现是什么也不做。它的目的是对属性对象进行清理和回收。虽然在它目前在Linux里还是什么都不做，但也许有一天它的实现会发生改变并要求对它进行调用，所以大家还是应该在程序里调用它才好。

在线程属性对象的初始化工作完成之后，我们可以调用其他一些函数来设置不同的属性行为。我们把它们都列在下面，但只对其中的两个做详细的说明。下面就是可供选用的属性函数清单。

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

```
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param)
int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param)
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inherit);
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
int pthread_attr_setstacksize(pthread_attr_t *attr, int scope);
int pthread_attr_getstacksize(const pthread_attr_t *attr, int *scope);
```

大家可以看到，线程的属性还真不少。

11.6.1 detachedstate属性

这个属性允许我们不对线程进行重新归并。它与大多数“_set”类函数一样，要用一个属性指针和一个标志来确定最终的状态。pthread_attr_setdetachstate函数用到的两个标志分别是PTHREAD_CREATE_JOIN和PTHREAD_CREATE_DETACHED。这个属性的缺省取值是PTHREAD_CREATE_JOIN，也就是允许两个线程归并。如果这个标志的状态被设置为PTHREAD_CREATE_DETACHED，我们就不能调用pthread_join来检查另一个线程的退出状态了。

11.6.2 schedpolicy属性

这个属性控制着线程的时间分配方式。它的可用取值是SCHED_OTHER、SCHED_RP和SCHED_FIFO。这个属性的缺省取值是SCHED_OTHER。另外两种定时方式只能用在以超级用户权限运行的进程里，它们都有实时定时功能，但在行为上略有区别。SCHED_RP使用轮转定时机制，而SCHED_FIFO则使用“先入先出”策略。对它们的讨论超出了本书的讨论范围。

11.6.3 schedparam属性

它是schedpolicy属性的一个搭档，它对以SCHED_OTHER定时策略运行的线程的时间分配进行控制。我们马上就会看到一个它的用法示例。

11.6.4 inheritsched属性

这个属性有PTHREAD_EXPLICT_SCHED和PTHREAD_INHERIT_SCHED两个可用取值。其中，缺省取值是PTHREAD_EXPLICT_SCHED，表示时间分配由属性来明确地设置。如果把它设置为PTHREAD_INHERIT_SCHED，新线程将继承沿用它创建者使用着的参数。

11.6.5 scope属性

这个属性控制着线程时间分配方面的计算方式。现阶段的Linux只支持它的一种取值：PTHREAD_SCOPE_SYSTEM，所以我们这里就不做进一步介绍了。

11.6.6 stacksize属性

这个属性控制着线程创建堆栈的长度，它以字节为计算单位。它属于POSIX技术规范里的“可选”部分，只有定义有_POSIX_THREAD_ATTR_STACKSIZE宏的实现版本才支持它。Linux在实现线程时缺省使用的堆栈是非常大的，所以这个功能在Linux上显得有些多余，因此也就很少有版本实现它了。

动手试试：把属性设置为脱离状态

我们以thread6.c为例向大家介绍脱离线程。我们创建一个线程属性，把它设置为脱离状态，然后用这个属性创建一个线程。子线程在完成工作时以正常方式调用pthread_exit。但这一次原先的线程不再等待要与它创建的线程重新合并了。大家将会看到，主线程通过一个简单的thread_finished标志来检查子线程是否已经结束，而线程们依然共享着变量。

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

void *thread_function(void *arg);

char message[] = "Hello World";
int thread_finished = 0;

int main() {
    int res;
    pthread_t a_thread;

    pthread_attr_t thread_attr;

    res = pthread_attr_init(&thread_attr);
    if (res != 0) {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
    if (res != 0) {
        perror("Setting detached attribute failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, &thread_attr, thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    (void)pthread_attr_destroy(&thread_attr);
    while(!thread_finished) {
        printf("Waiting for thread to say it's finished...\n");
        sleep(1);
    }
    printf("Other thread finished, bye!\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(4);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    pthread_exit(NULL);
}
```

输出结果里没有什么令人吃惊的东西。

```
$ cc -D_REENTRANT thread6.c -o thread6 -lpthread
$ ./thread6
Waiting for thread to say it's finished...
thread_function is running. Argument was Hello World
Waiting for thread to say it's finished...
Waiting for thread to say it's finished...
Waiting for thread to say it's finished...
Second thread setting finished flag, and exiting now
Other thread finished, bye!
```

操作注释：

这个程序里有两段代码比较重要，第一段代码是：

```
pthread_attr_t thread_attr;

res = pthread_attr_init(&thread_attr);
if (res != 0) {
    perror("Attribute creation failed");
    exit(EXIT_FAILURE);
}
```

这段代码用来声明一个线程属性并对它进行初始化；第二段代码是：

```
res = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
if (res != 0) {
    perror("Setting detached attribute failed");
    exit(EXIT_FAILURE);
}
```

它把属性的值设置为脱离状态。

其他不太重要的区别是创建线程和传递属性的地址：

```
res = pthread_create(&a_thread, &thread_attr, thread_function, (void *)message);
```

和为保证程序的完整性而对属性进行的清理回收：

```
pthread_attr_destroy(&thread_attr );
```

11.6.7 线程属性——调度

我们来看看另外一个我们可能会修改到的线程属性：调度。这与设置脱离状态的情况很相似，我们可以用sched_get_priority_max和sched_get_priority_min这两个函数来查明可用的优先级级别。

动手试试：调度

thread7.c与前面的例子很相似，所以我们只把它与其他示例不同的地方说明一下。

设置好脱离属性之后，我们对调度策略进行了设置，如下所示：

```
res = pthread_attr_setschedpolicy(&thread_attr, SCHED_OTHER);
if (res != 0) {
    perror("Setting scheduling policy failed");
```

```
    exit(EXIT_FAILURE);
}
```

接着我们查明允许的优先级范围，如下所示：

```
max_priority = sched_get_priority_max(SCHED_OTHER);
min_priority = sched_get_priority_min(SCHED_OTHER);
```

然后设置这一属性，如下所示：

```
scheduling_value.sched_priority = min_priority;
res = pthread_attr_setschedparam(&thread_attr, &scheduling_value);
if (res != 0) {
    perror("Setting scheduling priority failed");
    exit(EXIT_FAILURE);
}
```

操作注释：

这与设置一个脱离状态属性的过程很相似，区别仅在于我们是对调度策略进行设置的。

11.7 取消一个线程

有时候，我们想让一个线程能够请求另外一个线程结束，就像给它发送了一个信号似的。用线程是可以完成这一操作的，而与单处理相比，线程在被要求结束执行的时候还有一种改变其行为的办法。

我们先来看看要求一个线程结束执行的函数

```
# include <pthread.h>

int pthread_cancel (pthread_t thread );
```

这个定义很明白，给定一个线程标识符，我们就能要求取消它。但在取消线程请求的接收端，事情会稍微复杂一些，好在也不是太复杂。线程可以用pthread_setcancelstate设置自己的取消状态，下面是这个函数的定义：

```
# include <pthread.h>

int pthread_setcancelstate (int state, int *oldstate );
```

第一个参数可以是PTHREAD_CANCEL_ENABLE，这个值允许线程接收取消请求；还可以是PTHREAD_CANCEL_DISABLE，它的作用是屏蔽它们。线程以前的取消状态可以用oldstate指针检索出来，如果对它没有兴趣，可以简单地传递一个NULL过去。如果取消请求被接受了，线程会进入第二个控制层次——用pthread_setcanceltype设置取消类型。

```
# include <pthread.h>

int pthread_setcanceltype (int type, int *oldtype );
```

type参数可以有两种取值，一个是PTHREAD_CANCEL_ASYNCHRONOUS，接收到取消请求后立刻采取行动；另一个是PTHREAD_CANCEL_DEFERRED，在接收到取消请求之后、采取实际行动之前，先执行以下几个函数之一：pthread_join、pthread_cond_wait、pthread_cond_tomewait、pthread_testcancel、sem_wait或sigwait。

我们一般用不到这么多种函数，所以这一章也没有把它们全部介绍给大家。老办法，详细资料可以在它们的使用手册也里找到。

> 根据POSIX标准的说法，其他可以阻塞的系统调用，比如read、wait等，也可以是取消操作采取行动的取消点。在我们编写这本书的时候，Linux还不支持所有这些系统调都能用被当作取消点。但某些测试证明一些被阻塞的调用，比如sleep等，确实允许取消动作的发生。因此，为了确保安全，你可以在你估计会被取消的代码里添上一些pthread_testcancel调用。

oldtype参数的作用还是检索以前的状态，如果你不想知道以前的状态，给它传递一个NULL就行。在缺省的情况下，线程在启动时的取消状态是PTHREAD_CANCEL_ENABLE，而取消类型是PTHREAD_CANCEL_DEFERRED。

动手试试：取消一个线程

我们的程序thread8.c还是从thread2.c演变而来的。这一次，主线程向它创建的子线程发送了一个取消请求。

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

void *thread_function(void *arg);

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    sleep(3);
    printf("Cancelling thread...\n");
    res = pthread_cancel(a_thread);
    if (res != 0) {
        perror("Thread cancelation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int i, res;
    res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (res != 0) {
        perror("Thread pthread_setcancelstate failed");
        exit(EXIT_FAILURE);
    }
```

```
    res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    if (res != 0) {
        perror("Thread pthread_setcanceltype failed");
        exit(EXIT_FAILURE);
    }
    printf("thread_function is running\n");
    for(i = 0; i < 10; i++) {
        printf("Thread is still running (%d)...\n", i);
        sleep(1);
    }
    pthread_exit(0);
}
```

当我们运行这个程序的时候，我们将看到如下所示的输出情况：

```
$ cc  D_REENTRANT thread8.c -o thread8 -lpthread
$ ./thread8
thread_function is running
Thread is still running (0)...
Thread is still running (1)...
Thread is still running (2)...
Cancelling thread...
Waiting for thread to finish...
Thread is still running (3)...
$
```

操作注释：

按老法子创建出新线程之后，主线程休眠一小会儿（好让新线程有时间开始执行），然后发出一个取消请求。如下所示：

```
sleep(3);
printf("Cancelling thread...\n");
res = pthread_cancel(a_thread);
if (res != 0) {
    perror("Thread cancelation failed");
    exit(EXIT_FAILURE);
}
```

在新创建的线程里，我们先把取消状态设置为允许取消，如下所示：

```
res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
if (res != 0) {
    perror("Thread pthread_setcancelstate failed");
    exit(EXIT_FAILURE);
}
```

然后把取消类型设置为延迟取消，如下所示：

```
res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
if (res != 0) {
    perror("Thread pthread_setcanceltype failed");
    exit(EXIT_FAILURE);
}
```

最后，我们等待被取消，如下所示：

```
for(i = 0; i < 10; i++) {
    printf("Thread is still running (%d)...\n", i);
    sleep(1);
}
```

11.8 多线程

到目前为止，我们总是让程序的主执行线程另外创建出一个线程来。希望读者不要认为它只能多创建一个线程。在本章最后的示例程序thread9.c里，我们将向大家演示如何在同一个程序里创建出好几个线程，然后又是如何以不同于其启动顺序的顺序把它们归并到一起来的。

动手试试：许多线程

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

#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {
    int res;
    pthread_t a_thread[NUM_THREADS];
    void *thread_result;
    int lots_of_threads;

    for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {
        res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function,
(void *)&lots_of_threads);
        if (res != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
        sleep(1);
    }
    printf("Waiting for threads to finish...\n");
    for(lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0; lots_of_threads--) {
        res = pthread_join(a_thread[lots_of_threads], &thread_result);
        if (res == 0) {
            printf("Picked up a thread\n");
        }
        else {
            perror("pthread_join failed");
        }
    }
    printf("All done\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int my_number = *(int *)arg;
    int rand_num;

    printf("thread_function is running. Argument was %d\n", my_number);
    rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
    sleep(rand_num);
    printf("Bye from %d\n", my_number);
    pthread_exit(NULL);
}
```

当我们运行这个程序的时候，将看到如下所示的输出情况：

```
$ cc -D_REENTRANT thread9.c -o thread9 -lpthread
$ ./thread9
thread_function is running. Argument was 0
thread_function is running. Argument was 1
```

```
thread_function is running. Argument was 2
thread_function is running. Argument was 3
thread_function is running. Argument was 4
Bye from 1
thread_function is running. Argument was 5
Waiting for threads to finish...
Bye from 5
Picked up a thread
Bye from 0
Bye from 2
Bye from 3
Bye from 4
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
```

正如大家看到的，我们创建了许多线程并让它们以随意的顺序结束了执行。这个程序里面有一个漏洞，如果你把sleep调用从启动线程们的循环语句里删掉，它就会发作起来。我们借此提醒大家在编写使用线程的程序时必须多加注意。你看出来了吗？我们马上就来解释。

操作注释：

这一次，我们创建了一个线程ID的数组，如下所示：

```
pthreads_t a_thread[NUM_THREADS ];
```

然后通过循环语句创建出每一个线程，如下所示：

```
for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {
    res = pthread_create(&(a_thread[lots_of_threads]), NULL,
                         thread_function, (void *)&lots_of_threads);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    sleep(1);
}
```

线程们随机等待一段时间后退出运行，如下所示：

```
void *thread_function(void *arg) {
    int my_number = *(int *)arg;
    int rand_num;

    printf("thread_function is running. Argument was %d\n", my_number);
    rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
    sleep(rand_num);
    printf("Bye from %d\n", my_number);
    pthread_exit(NULL);
}
```

在主线程（即最一开始的那个线程）里，我们等着归并这些子线程，但顺序却不一定与创建它们时一样，如下所示：

```
for(lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0; lots_of_threads--) {
    res = pthread_join(a_thread[lots_of_threads], &thread_result);
    if (res == 0) {
        printf("Picked up a thread\n");
    }
    else {
```

```
            perror("pthread_join failed");
        }
    }
```

如果你去掉sleep调用后再运行这个程序，就可能会看到一些奇怪的现象，比如某些线程看起来好象被启动了两次或多次。你看得出为什么会发生这样的问题吗？在创建线程的时候，线程函数的参数使用的是一个局部变量，而这个变量是在循环里修改的。引起问题的代码是：

```
for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {
    res = pthread_create(&(a_thread[lots_of_threads]), NULL,
                         thread_function, (void *)&lots_of_threads);
```

如果主线程运行得足够快，就有可能改变某些线程的参数（即lots_of_threads变量）。当被共享的变量和多执行路径没有得到充分重视，程序就有可能出现这样的错误行为。改正这个错误很简单，直接传递这个参数的值就可以解决问题，如下所示：

```
res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function, (void *)lots_of_threads);
```

当然还要修改thread_function函数，如下所示：

```
void *thread_function (void *arg) {
    int my_number = (int) arg;
```

在出版社站点上的可下载代码里，修改后的程序是thread9a.c。

11.9 本章总结

在这一章里，我们对POSIX线程进行了学习。我们看到进程是如何创建出多个执行线程，而这些线程又是如何共享着文件范围变量的。

接着，我们学习了线程对关键代码和数据的访问操作进行控制的两种办法。在这之后，我们继续介绍了线程属性的控制问题。更准确地说，介绍了如何把它们与主线程隔离开，使主线程不再需要等待它创建的子线程的完成和结束。在看完一个线程如何请求另外一个线程结束执行、而接收这个请求的线程又是如何对这个请求进行处理的例子之后，我们给大家提供了一个有多个线程同时执行的程序示例。

我们没有足够的篇幅去讨论每一个函数调用和与线程有关的方方面面，但大家学到的知识已经足够你们开始编写自己的线程程序，并且，大家也应该能够通过查阅使用手册页对线程更加奇特的方面做进一步的研究了。

第12章　进程间通信：管道

我们在第10章介绍了一个进程间通信的简单方法：利用信号收发消息。我们创建通知事件，用这些事件激起一个响应，但传送的信息只限于一个单独的编号而已。

在这一章里，我们将对管道进行学习，它允许进程之间交换更多有用的数据。我们将在这一章的末尾用我们新学到的知识把CD数据库程序重新实现为一个客户/服务器软件。

我们将向大家介绍以下几个方面的知识：

- 管道的定义。
- 进程管道。
- 管道调用。
- 父进程和子进程。
- 命名管道：FIFO文件。
- 客户/服务器架构。

12.1　什么是管道

我们把从一个进程连接到另一个进程的一个数据流称为一个“管道”。我们通常是把一个进程的输出连接或“管接”（经过管道来连接）到另一个进程的输入。

UNIX用户应该对链接shell命令的概念很熟悉了，这就是把一个进程的输出直接馈入另一个的输入。对shell命令来说，它们是象下面这样输入的：

```
cmd1  |  cmd2
```

shell负责安排两个命令的标准输入和标准输出，这样：

- cmd1的标准输入来自终端键盘。
- cmd1的标准输出馈入cmd2做为它的标准输入。
- cmd2的标准输出连接到终端屏幕。

shell所做的工作从最终效果上看是这样的：重新安排标准输入和标准输出流之间的连接，使数据从键盘输入流过两个命令再输出到屏幕（见图12-1）。

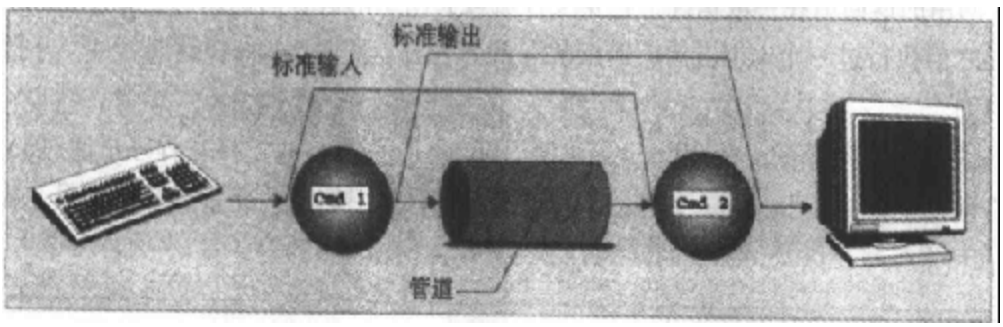


图　12-1

我们将在这一章里看到怎样才能在程序里获得这样的效果，怎样才能用管道把多个进程连接起来去实现一个简单的客户/服务器系统。

我们曾经在第5章里提到的伪终端设备和这里将要介绍的管道非常相似，但前者更加专业化，我们不准备在这里讨论它们。

12.2 进程管道

popen和pclose这两个函数大概是两个程序之间传递数据最简单的办法了。它们的框架定义如下所示：

```
#include <stdio.h>

FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

12.2.1 popen函数

popen函数允许一个程序把另外一个程序当做一个新的进程来启动，并能对它发送数据或者接收数据。command字符串是待运行程序的名字和相应的参数。open_mode必须是“r”或“w”

如果open_mode是“r”，调用者程序就可以使用来自被调用程序的输出，调用者程序利用popen返回的那个“FILE *”类型的指针用一般的stdio库函数（比如fread）就可以读这个文件流。如果open_mode是“w”，调用者程序就还能用fwrite调用向被调用命令发送数据，而被调用程序可以在自己的标准输入上读到这些数据。在一般情况下，被调用程序不会觉察到自己正在从另外一个进程读取着数据；它还象以前一样读自己的标准输入流并做出相应的动作。

每个popen调用都必须指定“r”或“w”，在popen的标准实现里不支持任何其他的选项。这就意味着我们不能调用另外一个程序同时对它进行读和写。如果操作失败，popen会返回一个空指针。如果你想通过管道实现双向通信，最普通的解决方案就是使用两个管道，每个管道负责一个方向的数据流动。

12.2.2 pclose函数

当用popen启动的进程完成了操作的时候，我们就可以用pclose关闭与之关联的文件流了。pclose调用只有在popen启动的进程结束之后才能返回。如果在调用pclose的时候它仍在运行，pclose将等待该进程的结束。

pclose调用的返回值在一般情况下将是文件流被关闭了的进程的退出码。如果调用者进程在调用pclose之前执行过一个wait语句，退出状态就会丢失，此时pclose将返回“-1”并把errno设置为ECHILD。

我们先来看一个简单的popen和pclose示例，popen1.c程序。我们将在程序里通过popen访问uname命令给出的信息。“uname -a”命令的作用是查看系统信息，变量计算机型号、操作系统名称、版本和发行号，以及计算机的网络名。

动手试试：读取外部程序的输出

完成程序的初始化工作之后，我们打开一个通往uname的管道，把管道设置为可读方式并让read_fp指向其输出。操作完成后，关闭read_fp指向的管道。

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output was:-\n%s\n", buffer);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

在作者之一的机器上运行这个程序的时候，我们将看到如下所示的输出：

```
$ popen1
Output was:-
Linux tilde 2.2.5-15 #1 Mon Apr 19 18:20:08 EDT 1999 i686 unknown
```

操作注释：

这个程序用popen调用启动带“-a”选项的uname命令。接下来，它从返回的文件流里读取最多BUFSIZ（这是stdio.h文件里的一个“# define”常数）个字符的数据，并把它打印出来显示在屏幕上。既然我们已经把uname的输出捕获到一个程序里，就可以对它做进一步处理了。

12.2.3 把输出送往popen

看过捕获外部程序输出的例子之后，我们再来看一个把输出传送到外部程序的例子。下面这个popen2.c程序把数据经管道送往另一个程序，我们这里用的是od（八进制输出）程序。

> **动手试试：把输出传送到外部程序**

读读下面的代码，愿意的话可以自己打打字……

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

int main()
{
    FILE *write_fp;
    char buffer[BUFSIZ + 1];

    sprintf(buffer, "Once upon a time, there was...\n");
```

```
    write_fp = popen("od -c", "w");
    if (write_fp != NULL) {
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        pclose(write_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

运行这个程序的时候，我们将看到如下所示的输出：

```
$ popen2
0000000   o   n   c   e       u   p   o   n       a       t   i   m   e
0000020   ,       t   h   e   r   e       w   a   s   .   .   .  \n
0000037
```

操作注释：

程序用带“w”选项的popen启动了“od -c”命令，这样就可以向它发送数据了。我们的程序发送了一个字符串，“od -c”命令接收并处理字符串，再把处理后的结果打印到自己的标准输出。

我们在命令行上可以用下面的命令得到同样的输出：

```
$ echo "once upon a time, there was . . ."lod -c
```

1. 传递更多的数据

到目前为止，我们的例子都是把全部数据放在一个fread或fwrite语句里的。可有时候我们希望能以比较小的块发送数据，或者我们不知道输出数据的长度。为了避免定义一个非常大的缓冲区，我们可以使用多个fread或fwrite调用一部分一部分地处理数据。

下面是程序popen3.c，它的全部数据都是从管道读取的。

动手试试：从管道读取大量的数据

在这个程序里，我们从被调用的“ps -alx”进程读取数据。输出数据有多少事先是不可能知道的，所以我们必须对管道进行多次读操作。

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("ps -ax", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
                buffer[chars_read - 1] = '\0';
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
```

```
    exit(EXIT_FAILURE);
}
```

为简洁起见，我们对这个程序的输出做了一些删节，如下所示：

```
$ popen3
Reading:-
   PID TTY STAT   TIME COMMAND
     1 ?   S      0:04 init
     2 ?   SW     0:00 [kflushd]
     3 ?   SW     0:00 [kpiod]
     4 ?   SW     0:00 [kswapd]
     5 ?   SW<    0:00 [mdrecoveryd]
...
   240 tty2 S     0:02 emacs draft1.txt
Reading:-
   368 tty1 S     0:00 popen3
   369 tty1 R     0:00 ps -ax
...
```

操作注释：

程序调用popen时使用了“r”选项，这与popen1.c程序的做法一样。这一次，它连续地从文件流读取数据，直到没有数据可读为止。需要提醒大家注意的是，虽然ps命令的执行要花费一些时间，但UNIX会安排好进程间的时间分配，让两个程序尽量同时运行。如果读进程popen3没有输入数据，它将被挂起直到有输入数据可读为止。如果写进程ps产生的输出超过了缓冲区的长度，它也会被挂起直到读进程消耗掉一些数据为止。

在这个例子里，你可能不会看到“Reading: -”信息的第二次出现。这是因为BUFSIZ的值超过了ps命令输出数据的长度。有些Linux系统把BUFSIZ设置为8000或更大的数字。

2. popen是如何实现的

当我们请求popen调用运行另外一个程序的时候，它先启动shell，即系统中的sh命令，然后把command字符串做为一个参数传递给它。这有两个效果，一个好，一个不太好。

在UNIX系统里，一切参数扩展都是由shell完成的，所以在程序被启动之前先启动shell来分析命令字符串就能够使各种shell扩展（比如“*.c”等）在程序开始运行之前全部完成。这个特性能够帮助我们用popen启动运行格式复杂的shell命令，非常有用。而其他一些也能够创建进程的函数（比如execl等）调用起来就比较麻烦，因为调用者进程必须自己去完成shell扩展。

可从另一方面考虑，使用shell又有一定的负面效果。popen调用不仅要启动一个被请求的程序，还要启动一个shell。每做一次popen调用，就要多启动两个进程；从系统资源的角度看，popen函数的成本偏大。

我们用popen4.c程序来演示popen的操作行为。我们将对全体popen示例程序的源代码文件的总行数进行统计，做法是用cat命令查看文件并把其结果经管道输送给“wc -l”命令，由后者统计出总的行数。如果是在命令行上实现这一操作，我们将使用下面的命令：

```
$ cat popen*.c | wc -l
```

事实上，用“wc -l popen*.c”命令就能搞掂这事，它不用打那么多字，也更有效率。但我们的例子是为了演示管道的原理，所以...

动手试试：popen启动了一个shell

这个程序照搬照抄上面给出的命令，但通过popen读取数据。

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("cat popen*.c | wc -l", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
                buffer[chars_read - 1] = '\0';
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

运行这个程序的时候，我们将看到如下所示的输出情况：

```
$ popen4
Reading:-
      101
```

操作注释：

从程序代码里可以看出，shell在启动后将把popen*.c扩展为一个文件清单，清单中的文件名字都以“popen”开始，以“.c”结束。此外，shell还要完成处理管道符号（|）和把cat命令的输出馈入wc命令的工作。我们启动了shell、cat程序和wc程序，安排了一次输出重定向，而使用这些都是在一个popen调用里完成的。调用了这些命令的程序只看到最终的结果。

12.3 pipe函数

学习完高级的popen函数之后，我们再来看看底层的pipe函数。通过它在两个程序之间传递数据时不需要启动一个shell来解释给定的命令，降低了程序的运行成本；我们对数据读、写操作的控制也加强了。

pipe函数的框架定义如下所示：

```
# include <unistd.h>

int pipe (int file_descriptor[2] );
```

pipe函数的参数是一个由两个整数类型的文件描述符组成的数组（的指针）。它在数组里填上两个新的文件描述符后返回“0”。如果操作失败，它返回“-1”并设置errno指示失败的原因。

在Linux的使用手册页里定义的错误有：

EMFILE 进程使用的文件描述符过多。

ENFILE 系统的文件表已经满了。

EFAULT 文件描述符无效。

那两个返回的文件描述符通过一种特殊的方式连接起来。写到file_descriptor[1]的所有数据都可以再从file_descriptor[0]处读回来。数据按先入先出的原则进行处理，人们把这种做法简称为FIFO（“First in，First Out”的字头缩写）。也就是说，如果你把字节“1，2，3”写到file_descriptor[1]里去，从file_descriptor[0]读回来的数据也会是“1，2，3”。这与堆栈的是不一样的，堆栈操作采用后进先出的原则，简称LIFO（“First in，First Out”的字头缩写）。

> 注意：这里必须使用的是文件描述符而不是文件流，所以访问数据时必须使用底层的read和write调用，而不是用fread和fwrite。

下面这个pipe1.c程序使用pipe创建了一个管道。

> 动手试试：pipe函数

敲入下面这些代码。注意file_pipes指针的用法，它被当作一个参数传递到pipe函数。

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        data_processed = write(file_pipes[1], some_data, strlen(some_data));
        printf("Wrote %d bytes\n", data_processed);
        data_processed = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_processed, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

运行这个程序的时候，我们将看到如下所示的输出情况：

```
$ pipe1
Wrote 3 bytes
Read 3 bytes: 123
```

操作注释：

这个程序创建的管道使用着两个文件描述符file_pipes[]。它用file_pipes[1]向管道里写数据，再从file_pipes[0]读回数据。我们给这个管道里增加了一点缓冲功能，让数据在write和read调用

的间隔里有地方保存。

请大家注意：用file_descriptor[0]读、或者用file_descriptor[1]写的后果在有关文档里没有定义，所以它们在不同版本的UNIX上的行为是会变化的。在作者的系统上，这样的调用失败了，返回值是“-1”——这多少能够使这一错误的查找工作简单些。

这个例子乍看上去好象没什么高明的地方，它做的工作用一个简单的文件也可以解决。管道真正的优越性体现在两个进程之间的数据传递上。我们在前面曾经看到过，当进程用fork调用创建出一个新的进程时，原来打开着的文件描述符还将保持在打开状态。如果我们在原来的进程里先创建一个管道再用fork函数创建一个新的进程，就可以把数据通过管道传递给其他进程。

动手试试：跨在fork调用两端的管道

1) 这是pipe2.c程序。它的开始部分和第一个例子差不多，直到我们发出fork调用。

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
```

2) 我们对fork的工作情况已经比较了解了，所以如果fork_result等于零，就说明我们是在子进程里，如下所示：

```
if (fork_result == 0) {
    data_processed = read(file_pipes[0], buffer, BUFSIZ);
    printf("Read %d bytes: %s\n", data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

3) 否则，我们就是在父进程里，如下所示：

```
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("Wrote %d bytes\n", data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

运行这个程序的时候，我们将看到如下所示的输出情况：

```
$ pipe2
```

```
Wrote 3 bytes
Read 3 bytes: 123
```

操作注释：

这个程序先用pipe调用创建了一个管道，接着用fork调用创建了一个新的进程。如果fork操作成功，父进程就把数据写到管道里去，而子进程就从管道里读出数据。父进程和子进程里分别只调用了一次write或read函数。如果父进程是在子进程前面退出的，你就在两部分输出内容之间看到一个shell提示符。

从表面上看，这个程序和第一个例子是很相似的，但这里的做法是把读写操作分别交给两个进程去完成，比过去前进了一大步。整个操作过程可以用图12-2来说明。

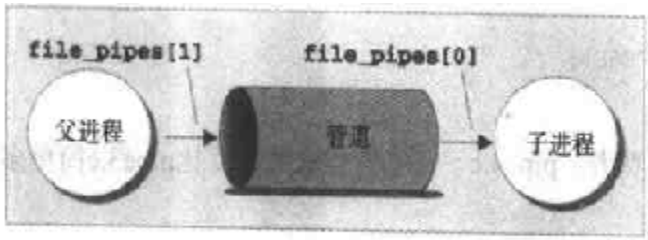


图 12-2

12.4 父进程和子进程

接下来，我们把管道的应用再推进一步：在子进程里启动并执行一个有别于它父进程的程序，而不只是一个简单的进程。我们用exec调用来完成这一工作。这就产生了一个问题：通过exec函数调用的进程需要知道应该去访问那个文件描述符。以前的例子根本用不着考虑这件事，因为子进程本身有file_pipes数据的一份拷贝，对它进行访问自然没有问题。但加上exec调用后的情况就和以前的不一样了，原先的进程已经被新的子进程替换掉了。这个问题可以这样解决：把文件描述符（它最终只是一个数字而已）当作一个参数传递到用exec启动的程序里去。

我们需要两个程序来做演示。第一个程序是“数据加工厂”，它负责创建管道和启动第二个程序，后者是一个“数据消费者”。

动手试试：管道和exec函数

1）这是“数据加工厂”pipe3.c，它是从pipe2.c修改而来的。改动之处加上了阴影。如下所示：

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

int main()
{

int data_processed;
int file_pipes[2];
const char some_data[] = "123";
char buffer[BUFSIZ + 1];
pid_t fork_result;
```

```
    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            sprintf(buffer, "%d", file_pipes[0]);
            (void)execl("pipe4", "pipe4", buffer, (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

2) 下面是“数据消费者”pipe4.c，它负责读取数据，比pipe3.c简单多了。

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

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

程序pipe3将调用程序pipe4。运行pipe3程序的时候，我们将看到如下所示的输出情况：

```
$ pipe3
980 - wrote 3 bytes
981 - read 3 bytes: 123
```

操作注释：

pipe3程序的开始部分和前面的例子一样，它先通过pipe调用创建一个管道，再通过fork调用创建一个新的进程。接下来，它用sprintf把对应于管道“读操作”的文件描述符数字保存的一个缓冲区里，该缓冲区中的内容将构成pipe4程序的一个参数。

我们通过一个excel调用来启动pipe4程序。excel调用有四个参数，它们的含义依次是：

准备启动的程序的名称。

- argv[0]，被调用程序的名称。
- argv[1]，存放着我们想让被调用程序去读取的文件描述符。

• (char *) 0，这个参数的作用是结束被调用程序的执行。

pipe4程序从参数字符串里提取出文件描述符数字，再利用文件描述符进行读操作以取得数据。

12.4.1 管道关闭后的读操作

在继续学习之前，我们先来研究研究打开着的文件描述符。我们到目前为止一直采取着一个简单的做法：让读进程读取一些数据后直接退出——认为UNIX会把清理文件当做是进程结束扫尾工作的一部分。

我们此前见过的例子都是这样做的，但大多数从标准输入读取数据的程序采用的却是另外一种做法。通常，它们并不知道有多少数据需要它去读取，所以它们往往采用循环的办法，读数据——处理数据——读更多的数据，直到没有数据可读为止。

当没有数据可读时，read调用就会阻塞，即进程暂停执行，一直等到有数据来到为止。如果管道的另一端已经被关闭了，也就是没有进程打开这个管道并向它写数据的时候，read调用就会阻塞。但阻塞并不解决问题，所以对没有为写数据而打开的管道做read调用时将返回“0”，而不是阻塞。这就使读进程能够象检测文件文件尾end-of-file标志一样对管道进行检测。注意：这与读一个无效的文件描述符不同，read把无效的文件描述符看做是一个错误，它将返回一个“-1”表示有问题发生。

如果我们在fork调用的两端使用一个管道，就会产生两个可以用来向管道写数据的文件描述符，一个在父进程里，另外一个在子进程里。只有把父进程和子进程里的两个写操作文件描述符都关闭了，管道才会被认为是关闭了，以后的read调用就会不能进行。我们还会对这一问题做进一步讨论，在学习到O_NONBLOCK标志和FIFO文件的时候，大家将看到一个这样的例子。

12.4.2 把管道用做标准输入和标准输出

知道读空管道操作失败的原因之后，我们再向大家介绍一个用管道连接两个进程时更简明的方法。我们把一个管道文件描述符设置为一个已知值，一般是标准输入“0”，或标准输出“1”。这比在父进程里进行设置要稍微复杂一些，但子程序的编写工作就大大简化了。

这样做最大的好处是我们可以调用运行操作系统中的标准程序，即那些不需要文件描述符做参数的程序。这就要用到我们在第3章里遇见的dup函数。dup有两个紧密关联的版本，它们的定义情况如下所示：

```
#include <unistd.h>

int dup(int file_descriptor);
int dup2(int file_descriptor_one, int file_descriptor_two);
```

dup调用的作用是打开一个新的文件描述符，这与open调用有点相似。dup的特点是用它新创建出来的文件描述符与做为它参数的那个现有的文件描述符指向的是同一个文件（或管道）。dup函数创建的新文件描述符永远取最小的可用值；而dup2函数创建的新文件描述符或者与file_descriptor_two相同，或者是第一个大于file_descriptor_two的可用值。

通过更具普遍意义的fcntl调用，我们能达到与调用dup和dup2相同的目的，fcntl调

用中相应的命令是F_DUPFD。虽然这么说，还是dup函数用起来更方便些，因为它们是专门来复制文件描述符用的。它们的使用面也非常广，读者在现有程序里更经常看见的是它们而不是fcntl和F_DUPFD。

那么，dup是如何帮助我们在进程间传递数据的呢？秘密就在这句话里：标准输入的文件描述符永远是“0”，而dup返回的新文件描述符永远是最小的可用数字。因此，先关闭文件描述符“0”，再调用dup，这样得到的新文件描述符就会是数字“0”。这个取值为“0”的新文件描述符是一个现有文件描述符的复制品，所以标准输入就会改为指向一个文件或管道——我们刚才传递给dup函数的参数就是它们的文件描述符。我们创建了两个文件描述符，它们指向同一个文件或管道，而且其中之一还是标准输入。

用close和dup对文件描述符进行处理

表 12-1

文件描述符	初 始 值	关 闭 后	dup调用后
0	标准输入		管道文件描述符
1	标准输出	标准输出	标准输出
2	标准错误	标准错误	标准错误
3	管道文件描述符	管道文件描述符	管道文件描述符

我们再回到上一个例子。这一次，我们将把子程序的stdin文件描述符替换为我们创建的管道的读操作端。我们还要对文件描述符做些准备工作，好让子程序能够正确地检测到管道里数据的“文件尾”。为了简洁起见，我们象往常一样省略了一些错误检查。

动手试试：管道和dup函数

把pipe3.c修改为pipec.5，程序清单如下所示：

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    (pid_t) fork_result;

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == (pid_t)0) {
            close(0);
            dup(file_pipes[0]);
            close(file_pipes[0]);
            close(file_pipes[1]);
```

```
            execlp("od", "od", "-c", (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            close(file_pipes[0]);
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            close(file_pipes[1]);
            printf("%d - wrote %d bytes\n", (int)getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

这个程序的输出情况如下所示：

```
$ pipe5
1239 - wrote 3 bytes
0000000   1   2   3
0000003
```

操作注释：

程序象以前一样先创建出一个管道，再通过fork创建出一个子进程。此时，父进程和子进程都有了能够对管道进行访问的文件描述符，一个用于读操作，另外一个用于写操作，加在一起总共有四个打开的文件描述符。

我们先来看看子进程。子进程先用“close(0)”关闭了自己的标准输入，然后调用“dup(file_pipes[0])”。这个调用把关联在管道读操作端的文件描述符复制为文件描述符“0”，即标准输入。接下来，子进程关闭了原先那个用来从管道读取数据的文件描述符file_pipes[0]。因为子进程不会向管道写数据，所以它把与管道关联着的写操作文件描述符file_pipes[1]也关闭了。现在它只有一个文件描述符是与管道关联着的了，即文件描述符0，它的标准输入。

接下来，子进程就能通过exec来启动运行各种从标准输入读取数据的程序了，例子里使用的是od命令。od命令等待数据的到来，就好像它在等待来自用户终端的输入一样。事实上，如果没有对两种情况进行判别的特殊代码，它根本就不会知道输入是来自一个管道，还是来自一个终端。

父进程一上来先关闭了管道的读操作端file_pipes[0]，因为它不会从管道读取数据。接着它向管道写入数据。当数据全部写完后，它关闭了管道的写操作端并退出了运行。现在已经没有打开着的管道写操作文件描述符了，od程序读出写到管道里的三个字节，但再往后的读操作将返回“0”字节，表示已经到达文件尾。当读操作返回“0”时，od程序退出运行。这是对在终端上运行od命令时用“Ctrl-D”组合键向od命令发送一个文件尾这种做法的模仿。

我们用下面几个示意图把执行过程表示出来。在调用了pipe函数之后见图12-3。

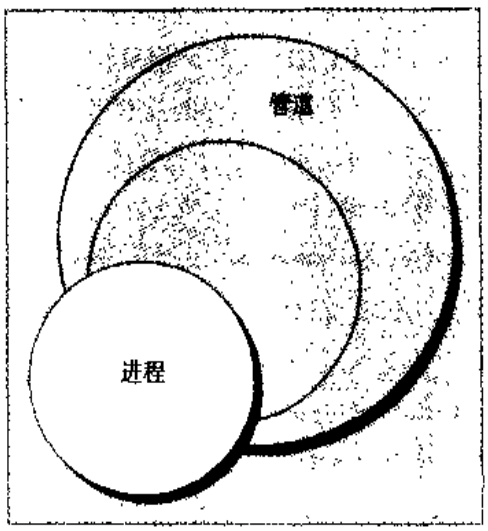


图 12-3

在调用了fork函数之后（如图12-4所示）：

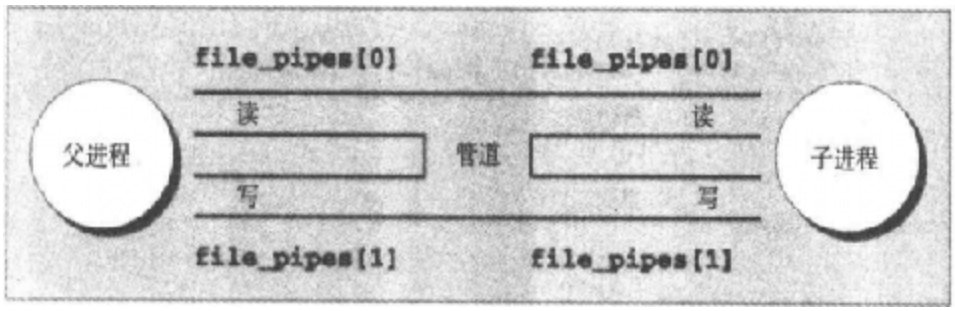


图 12-4

当程序做好数据传输的准备工作之后（如图12-5所示）：

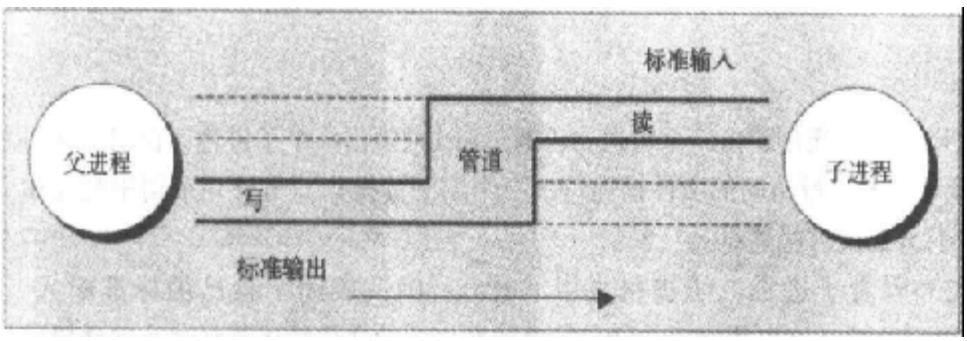


图 12-5

12.5 命名管道：FIFO文件

到目前为止，我们的数据传递工作还只能在相关程序之间进行，这些程序是由一个共同的祖先进程启动的。如果我们想在不相关的进程之间交换数据，还用以前的老办法就不是很方便了。

我们可以使用FIFO文件来做这项工作，它经常被称为命名管道。命名管道是一种特殊类型的文件（别忘了UNIX里的一切事物都是文件！），它们在文件系统里以名字的形式存在，但它们的行为却和我们刚才见过的没有名字的管道差不了多少。

命名管道可以从命令行上创建，也可以从程序里创建。命令行上用来创建命名管道的程序一直沿用的是mknod，如下所示：

```
$ mknod filename p
```

但mknod没有出现在X/Open技术规范的命令表里，所以它在某些UNIX系统上就不能用。推荐的命令行方法是使用下面这个命令：

```
$ mkfifo filename
```

有些老版本的UNIX只有mknod命令。“X/Open Issue 4 Version 2”（X/Open技术规范第4卷第2版）里面有mknod函数调用，但没有这个命令。Linux同时支持mknod和mkfifo两种命令。

在程序里，我们可以使用两种不同的函数，它们是：

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

类似于mknod命令，你可以用mknod函数建立出许多特殊类型的文件来。如果要用这个函数来创建命名管道，惟一具有可移植性的办法是把dev_t的值取为“0”，再把文件访问模式和S_IFIFO按位或（OR）在一起。我们在下面的例子里将使用相对简单一些的mkfifo函数。

动手试试：创建一个命名管道

下面是程序fifo1.c的代码清单：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
    int res = mkfifo("/tmp/my_fifo", 0777);
    if (res == 0) printf("FIFO created\n");
    exit(EXIT_SUCCESS);
}
```

我们用下面这个命令可以找到刚创建的管道：

```
$ ls -lF /tmp/my_fifo
prwxr-xr-x   1 rick     users            0 Dec 10 14:55 /tmp/my_fifo|
```

第一个字符是“p”就表示这是一个管道。尾部的“|”符号是ls命令的“-F”选项添上去的，也表示这是一个管道。

操作注释：

这个程序通过mkfifo函数创建了一个特殊的文件。虽然我们设定的模式是“0777”，但它被用户掩码（umask）设置（本例是“022”）给改变了，这与普通文件的创建操作是一样，文件最终的模式是“755”。

我们可以通过rm命令来删除FIFO文件；如果想在程序里这样做，就要使用unlink系统调用。

访问一个FIFO文件

命名管道能够出现文件系统里，也可以象平常的文件名那样用在命令里，这是一个非常有用的功能。在把我们创建的FIFO文件用到程序设计里去之前，我们先通过普通的文件命令来看看FIFO文件的表现。

动手试试：对FIFO文件的读写操作

1）首先，读一个（空白的）FIFO文件：

```
$ cat < /tmp/my_fifo
```

2) 现在，试试对FIFO文件的写操作：

```
$ echo "sdsdfasdf" > /tmp/my_fifo
```

3) 如果同时对它进行读写，我们就可以让数据通过管道：

```
$ cat < /tmp/my_fifo &
[1] 1316
$ echo "sdsdfasdf" > /tmp/my_fifo
sdsdfasdf

[1]+  Done                    cat </tmp/my_fifo
$
```

前两个命令都会被挂起直到我们按下“Ctrl-C”组合键中断它们。第三个命令可以让我们看到如上所示的屏幕输出。

操作注释：

因为FIFO文件里没有任何数据，所以cat和echo程序都阻塞了，cat要等待有数据到来，echo要等待有其他进程读走数据。

我们再来看看第三条命令，cat进程一上来就被阻塞在后台里了。当echo给它提供了一些数据时，cat命令读取这些数据并把它们打印到标准输出上去；然后cat程序立刻就退出了，不再等待其他数据。它之所以没有阻塞是因为管道已经被关闭了，cat程序中的read调用返回了“0”字节，表示已经到达了文件尾。

看过用命令行程序访问FIFO文件的情况之后，我们再来仔细看看它的程序设计接口；在访问FIFO文件的时候，它可以让我们对read和write的操作行为做更细致的控制。

> FIFO和通过pipe调用创建的管道不同，它是一个有名字的文件而表示一个打开的文件描述符，在对它进行读或写操作之前必须先打开它。FIFO文件也要用open和close函数来打开或者关闭，除了一些额外的功能外，整个操作过程与我们以前介绍的文件操作是一样的。传递给open调用的是一个FIFO文件的路径名，而不是一个正常文件的路径名。

1. 用open打开一个FIFO文件

在打开FIFO文件时需要注意一个问题：即程序不能以O_RDWR模式打开FIFO文件进行读写，有关标准里对这样做的后果没有定义。这条限制是有道理的，因为用到FIFO文件的时候基本上都是为了单方向传递一些数据，确实也用不着O_RDWR模式。如果一个管道是以读/写方式打开的话，进程就会从这个管道读回自己的输出。

如果确实需要在程序之间双向传递数据的话，我们可以同时使用一对FIFO或管道，一个方向配一个；还可以用先关闭再重新打开FIFO的办法明确地改变数据流的方向。我们将在本章稍后的内容里对使用FIFO文件做双向数据交换的问题进行讨论。

打开一个FIFO文件和打开一个普通文件的区别还在于open_flag（open函数的第二个参数）的O_NONBLOCK选项的用法上。如果在打开文件时使用了这个选项，那不仅open调用的处理方式会改变，就连read和write请求在这次open调用返回的文件描述符上被处理的方式也会改变。

O_RDONLY、O_WRONLY和O_NONBLOCK标志可以有四种合法组合。我们把它们逐个介绍给大家。

```
open (const char *path, O_RDONLY ) ;
```

如果用这个调用来打开FIFO文件，open调用将阻塞，除非有其他进程以写方式打开同一个FIFO文件，否则它将无法返回。这与前面第一个cat命令的例子一样。

```
open (const char *path, O_RDONLY | O_NONBLOCK ) ;
```

即使没有其他进程以写方式打开这个FIFO文件，这个open调用也会成功并立刻返回。

```
open (const char *path, O_WRONLY) ;
```

在这种情况下，open调用将阻塞，直到有其他进程以读方式打开同一个FIFO文件为止。

```
open (const char *path, O_WRONLY | O_NONBLOCK ) ;
```

它总是立刻就返回了，但如果没有其他进程以读方式打开这个FIFO文件，open将返回一个错误“-1”，而FIFO文件也不会真的被打开。如果真的有其他进程以读方式打开了这个FIFO文件，我们就能通过它返回的文件描述符对这个FIFO文件进行写操作。

请注意O_NONBLOCK分别搭配O_RDONLY和O_WRONLY时的不同作用：如果没有进程以读方式打开管道，非阻塞写open调用将失败；但非阻塞读open调用却总是会成功。close调用的行为不受O_NONBLOCK标志的影响。

带O_NONBLOCK标志的open调用其行为各有特点，我们来看看如何利用这些特点对两个进程进行同步。我们没有选择编写多个示例程序的做法，对FIFO文件行为的研究将只使用一个测试程序fifo2.c来完成。

动手试试：打开FIFO文件

1) 线程的开始部分是头文件，“# define”常数定义，然后检查命令行上是否给出了足够的参数。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[])
{
    int res;
    int open_mode = 0;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <some combination of\
                O_RDONLY O_WRONLY O_NONBLOCK>\n", *argv);
        exit(EXIT_FAILURE);
    }
```

2) 在假设程序会通过测试的前提下，我们根据命令行参数来设置open_mode的值，如下所示：

```
argv++;
if (strncmp(*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
if (strncmp(*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
if (strncmp(*argv, "O_NONBLOCK", 10) == 0) open_mode |= O_NONBLOCK;
argv++;
if (*argv) {
    if (strncmp(*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
    if (strncmp(*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
    if (strncmp(*argv, "O_NONBLOCK", 10) == 0) open_mode |= O_NONBLOCK;
}
```

3) 现在检查这个FIFO文件是否存在，如有必要就创建它。接下来打开这个FIFO文件。如果程序“捉住了四十大盗”——操作成功，就给出相应的输出。最后，关闭这个FIFO文件。

```
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO\n", getpid());
    res = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), res);
    sleep(5);
    if (res != -1) (void)close(res);
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

操作注释：

这个程序允许我们在命令行上设定我们准备使用的O_RDONLY、O_WRONLY和O_NONBLOCK标志的组合。它会把命令行参数与程序中的常数字符串进行比较，如果二者匹配，就（用“!=”操作符）设置上相应的标志。程序用access函数检查FIFO文件是否存在，如果不存在就创建它。

我们在程序的最后没有删除这个FIFO文件，因为我们不知道是否有其他程序正在使用它。不管怎么说，我们手里现在有自己的测试程序了，所以我们对以下几种标志组合情况进行逐个分析：

- O_RDONLY和O_WRONLY，都不带O_NONBLOCK标志。

```
$ ./fifo2 O_RDONLY &
[1] 152
Process 152 opening FIFO
$ ./fifo2 O_WRONLY
Process 153 opening FIFO
Process 152 result 3
Process 153 result 3
Process 152 finished
Process 153 finished
```

这可能是命名管道最常见的用法了。它允许读进程启动，在open调用里等待；当第二个程序打开FIFO文件的时候，两个程序都将开始继续执行。注意，读进程和写进程是在open调用处得到同步的。

> 当一个UNIX进程被阻塞的时候，它不消耗CPU的资源，所以这种进程同步方法从CPU的角度看是非常有效率的。

• 带O_NONBLOCK标志的O_RDONLY和不带该标志的O_WRONLY。

```
$ ./fifo2 O_RDONLY O_NONBLOCK &
[1] 160
Process 160 opening FIFO
$ ./fifo2 O_WRONLY
Process 161 opening FIFO
Process 160 result 3
Process 161 result 3
Process 160 finished
Process 161 finished
[1]+  Done                    fifo2 O_RDONLY O_NONBLOCK
```

这一次，读进程通过了open调用，即使写进程没有出现它也会立刻继续执行。写进程也立刻通过了open调用，因为那个FIFO文件已经以读方式打开过了。

这两个例子可能是Open模式的最常见组合。你还可以使用一些程序示例来试试其他组合情况。

2. 对FIFO文件进行读写

使用O_NONBLOCK模式会影响到FIFO文件上的readud和write调用。

空白且阻塞（即打开时使用了O_NONBLOCK标志）的FIFO文件上的read调用将等到有数据可读时才能继续执行。而没有任何数据的非阻塞FIFO文件上的read调用将返回“0”字节。

一个完全阻塞的FIFO文件上的write调用将等到有数据可写才能继续执行。如果FIFO文件不能接受写入数据的全部字节，它上面的write调用将按下面的规则执行：

• 如果请求写入的数据其长度等于或小于PIPE_BUF个字节，调用将失败，数据不能写入。

• 如果请求写入的数据其长度大于PIPE_BUF个字节，将写入部分数据，返回值是实际写入的字节数，它可以是零。

FIFO文件的长度是一个很重要的因素。系统对任一时间一个FIFO文件里所能够“保存”的数据长度是有限制的。这是一个用“# define”语句定义的常数，一般可以在头文件limits.h里查到。在Linux和许多其他的UNIX系统上，这个常数经常是4096个字节，但在某些系统上它可能会小到512个字节。系统规定：在一个以O_WRONLY方式（即阻塞模式）打开的FIFO文件上，如果写入的数据长度等于或小于PIPE_BUF个字节，那就会或者写入全部字节，或者一个字节都不写。

如果只有一个FIFO写进程和一个FIFO读进程，这个限制就没有多么重要。但只使用一个FIFO文件却允许多个不同的程序向只有一个的FIFO读进程发送请求的情况是很多见的。如果几个不同的程序同时试图写这个FIFO文件，能否保证来自不同程序的数据块不会交织在一起就非常关键了——也就是说，每一个写操作都必须是“原子化”的。怎样才能做到这一点呢？

是这样的，如果你能够保证你的写请求是对一个阻塞FIFO文件进行的操作，并且数据的长度等于或小于PIPE_BUF个字节，系统就能够确保数据不会交织在一起。在一般情况下，把通过FIFO文件传输的数据限制为一个个PIPE_BUF个字节的数据块是个很好的办法，除非你只使用着一个写进程和一个读进程。

为了演示不相关的程序如何使用命名管道进行通信，我们需要用到独立的两个程序，fifo3.c和fifo4.c。

动手试试：利用FIFO文件实现进程间通信

1) 首先是我们的“数据加工厂”程序。它会在必要时创建出管道来，并且会尽快地向管道写入数据。

注意，出于演示的目的，我们不关心那都是些什么数据，所以我们也没有对缓冲区buffer进行初始化。在这两个程序清单里，与fifo2.c不一样的地方都加上了阴影，而对命令行参数进行处理的那部分代码都省略掉了。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_WRONLY;
    int bytes_sent = 0;
    char buffer[BUFFER_SIZE + 1];

    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        while(bytes_sent < TEN_MEG) {
            res = write(pipe_fd, buffer, BUFFER_SIZE);
            if (res == -1) {
                fprintf(stderr, "Write error on pipe\n");
                exit(EXIT_FAILURE);
            }
            bytes_sent += res;
        }
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

2) 我们的第二个程序是“数据消费者”，它就简单多了。它的作用是从FIFO文件读取数据并丢弃它们。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;

    memset(buffer, '\0', sizeof(buffer));

    printf("Process %d opening FIFO O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        do {
            res = read(pipe_fd, buffer, BUFFER_SIZE);
            bytes_read += res;
        } while (res > 0);
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
    exit(EXIT_SUCCESS);
}
```

我们在运行这两个程序的同时用time命令对读进程进行计时。我们将看到如下所示的输出情况（为简洁起见做了一些删节）：

```
$ ./fifo3 &
[1] 375
Process 375 opening FIFO O_WRONLY
$ time ./fifo4
Process 377 opening FIFO O_RDONLY
Process 375 result 3
Process 377 result 3
Process 375 finished
Process 377 finished, 10485760 bytes read
0.01user 0.02system 0:00.03elapsed 85%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (80major+9minor)pagefaults 0swaps
[1]+  Done                    fifo3
```

操作注释：

两个程序使用的都是阻塞模式的FIFO文件。我们先启动fifo3（写进程/数据加工厂），它将阻塞并等待有读进程打开这个FIFO文件。当fifo4（数据消费者）启动的时候，写进程解除阻塞状态，并开始向管道写数据。同时，读进程也开始从管道读取数据。

> UNIX会为这两个进程安排好它们的时间分配情况，使它们在能够运行的时候运行，

在不能运行的时候阻塞。因此，写进程将在管道满的时候阻塞，而读进程将在管道空的时候阻塞。

time命令的输出告诉我们读进程只运行了不到0.1秒的时间，读取了10兆字节的数据。这说明管道，至少在作者使用的Linux版本里，是一种在程序间传输数据的高效办法

12.6 高级论题：以FIFO文件为基础的客户/服务器架构

作为对FIFO文件学习效果的检验，我们来看看怎样才能通过FIFO文件编写出一个非常简单的客户/服务器应用软件。我们想只用一个服务器进程来接受请求、对它们进行处理，再把处理后的结果数据返回给提出请求的另一方，客户。

我们想让多个客户进程都能够向服务器发送数据。为了使问题简单化，我们将假设将被处理的数据都可以拆分成一个个的数据块，每个数据块的长度都小于PIPE_BUF个字节。当然，要想实现这个系统可以有多种办法，但我们将只考虑能够体现出命名管道使用方法的办法。

因为服务器每次只能对一个数据块进行处理，所以只使用一个让服务器进程读、让每一个客户写的FIFO文件应该是合乎逻辑的。只要以阻塞模式打开了FIFO文件，服务器和客户就会根据具体情况自动被阻塞。

向客户发送处理后的数据稍微棘手一些。我们需要为每个客户安排一个第二管道来发送经过处理的数据。我们的做法是：在最先传递给服务器的数据里加上客户的进程标识符（即它的PID），这样双方就可以用这个PID值为用来返回数据的管道生成一个独一无二的名字了。

动手试试：一个客户/服务器软件示例

1) 首先，我们来编写一个头文件cliserv.h。它的作用是定义客户程序和服务器程序都会用到数据。为了方便其他程序的编写，它还包括上了必要的系统头文件。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define SERVER_FIFO_NAME "/tmp/serv_fifo"
#define CLIENT_FIFO_NAME "/tmp/cli_%d_fifo"

#define BUFFER_SIZE 20

struct data_to_pass_st {
    pid_t  client_pid;
    char   some_data[BUFFER_SIZE - 1];
};
```

2) 现在轮到服务器程序server.c。我们在这一部分创建并打开了服务器管道。它被设置为只读的阻塞模式。在稍做休眠（这是出于演示的目的）之后，服务器开始读取客户们发送来的数据，这些数据是按data_to_pass_st结构组织的。

```
#include "cliserv.h"
#include <ctype.h>

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int read_res;
    char client_fifo[256];
    char *tmp_char_ptr;

    mkfifo(SERVER_FIFO_NAME, 0777);
    server_fifo_fd = open(SERVER_FIFO_NAME, O_RDONLY);
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Server fifo failure\n");
        exit(EXIT_FAILURE);
    }

    sleep(10); /* lets clients queue for demo purposes */

    do {
        read_res = read(server_fifo_fd, &my_data, sizeof(my_data));
        if (read_res > 0) {
```

3) 在接下来的这一部分里，我们对刚刚从客户那里读到的数据做一些处理：我们把Some_data中的字符全部转换为大写，做把CLIENT_FIFO_NAME和接收到的client_pid结合在一起。

```
tmp_char_ptr = my_data.some_data;
while (*tmp_char_ptr) {
    *tmp_char_ptr = toupper(*tmp_char_ptr);
    tmp_char_ptr++;
}
sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
```

4) 然后，我们以只读的阻塞模式打开客户管道，把经过处理的数据发送回去。最后是关机处理：关闭服务器管道的文件描述符，用unlink调用删除服务器管道的FIFO文件，退出程序运行。

```
            client_fifo_fd = open(client_fifo, O_WRONLY);
            if (client_fifo_fd != -1) {
                write(client_fifo_fd, &my_data, sizeof(my_data));
                close(client_fifo_fd);
            }
        }
    } while (read_res > 0);
    close(server_fifo_fd);
    unlink(SERVER_FIFO_NAME);
    exit(EXIT_SUCCESS);
}
```

5) 从这里开始是客户程序client.c。程序的开始部分先检查服务器FIFO文件是否存在，如果存在就打开它。然后查出自己的进程ID，它构成了将要发送到服务器去的数据的一部分。然后创建好客户FIFO文件，进入下一部分。

```
#include "cliserv.h"
#include <ctype.h>

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int times_to_send;
    char client_fifo[256];

    server_fifo_fd = open(SERVER_FIFO_NAME, O_WRONLY);
```

```
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Sorry, no server\n");
        exit(EXIT_FAILURE);
    }

    my_data.client_pid = getpid();
    sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
    if (mkfifo(client_fifo, 0777) == -1) {
        fprintf(stderr, "Sorry, can't make %s\n", client_fifo);
        exit(EXIT_FAILURE);
    }
```

6) 这部分有五次循环。每次循环都把客户数据发送给服务器，然后打开客户FIFO文件（只读，阻塞模式），读回数据。最后，关闭客户FIFO文件并从内存里把它删掉。

```
    for (times_to_send = 0; times_to_send < 5; times_to_send++) {
        sprintf(my_data.some_data, "Hello from %d", my_data.client_pid);
        printf("%d sent %s, ", my_data.client_pid, my_data.some_data);
        write(server_fifo_fd, &my_data, sizeof(my_data));
        client_fifo_fd = open(client_fifo, O_RDONLY);
        if (client_fifo_fd != -1) {
            if (read(client_fifo_fd, &my_data, sizeof(my_data)) > 0) {
                printf("received: %s\n", my_data.some_data);
            }
            close(client_fifo_fd);
        }
    }
    close(server_fifo_fd);
    unlink(client_fifo);
    exit(EXIT_SUCCESS);
}
```

进行测试的时候，我们只需运行一次服务器程序，客户程序可以运行多次。为了让它们尽可能地在几乎同一时间启动，我们将使用如下所示的shell命令：

```
$ server &
$ for i in 1 2 3 4 5
do
client &
done
```

shell命令启动了一个服务器进程和五个客户进程。下面是客户们产生的输出，为简洁起见我们做了删节：

```
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
529 sent Hello from 529, received: HELLO FROM 529
530 sent Hello from 530, received: HELLO FROM 530
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
```

正如大家看到的，不同的客户请求交织在了一起，但每个客户提交的数据在得到服务器的处理之后都能正确地返回给对应的客户。注意，客户请求的交织现象是随机的，服务器接收到客户请求的先后顺序会随机器的不同而有所差异，即使是同一台机器，各次运行的情况也有可能发生变化。

操作注释：

在下面的注释部分里，我们将尝试对客户和服务器交互执行时各种操作的先后顺序进行说

明，这是我们以前没有涉及到的。

服务器以只读模式创建自己的FIFO文件并阻塞。它将一直等到有客户以写方式打开这个FIFO文件建立连接为止。此时，服务器进程解除阻塞状态，执行sleep语句进入休眠，这使来自客户的请求排队等候。（在实际软件里应该把sleep语句去掉。我们之所以在这里使用它，是为了演示此程序多个进程同时执行的正确操作。）

与此同时，在打开了服务器FIFO之后，客户创建自己独一无二的命名管道，为读取服务器发回的数据做好准备。这一切都完成后，客户开始向服务器写数据（如果管道是满的，或者服务器仍在休眠中就阻塞）并阻塞在自己的FIFO上的read调用处，等待服务器的答复。

接收到来自客户的数据后，服务器对数据进行处理，以写方式打开客户管道并把数据写回去，写回去的数据将解除客户的阻塞状态。当客户不再被阻塞时，它就可以从它自己的管道里读取服务器写入的数据了。

整个过程不断重复，直到最后一个客户关闭服务器管道为止，而这将使服务器的read调用失败（返回“0”），因为已经没有任何进程以写方式打开服务器管道了。如果这是一个真正的服务器进程，还需要等待以后的客户，我们就需要对它进行改进，这有两种做法：

- 对它自己的服务器管道打开一个文件描述符，阻塞最后一个read调用而不是返回“0”。
- 当read返回“0”字节时，关闭再重新打开一个服务器管道，使服务器进程阻塞在open处等待客户的到来，就象它最初启动时那样。

在重新编写CD数据库软件时，我们将向大家演示命名管道的这两种使用技巧。

12.7 CD唱盘管理软件

看过用管道实现简单的客户/服务器系统的例子之后，我们重新造访我们的CD唱盘管理软件，并准备照猫画虎地对它进行改进。我们还将添加一些信号处理内容，使我们能够在进程被中断的时候采取一些清理动作。

在我们深入到这个新版本里去之前，先来编译这个软件。如果读者已经从出版社站点上下载了源代码，请用Makefile来编译server_app和client_app这两个程序。敲入“server_app -i”将使程序初始化一个新的CD唱盘数据库。不用说，如果服务器没有启动运行，客户是不会运行的。下面是对程序进行编译用的Makefile文件。

```
all:    server_app client_app

CC=cc
CFLAGS=-I/usr/include/db1 -pedantic -Wall

# For debugging un-comment the next line
# DFLAGS=-DDEBUG_TRACE=1 -g


# Include for systems with dbm, but only as part of the BSD licensed version,
# and not in the standard locations. This is the default...
DBM_INC_PATH=/usr/include/db1
DBM_LIB_PATH=/usr/lib
DBM_LIB_FILE=db

# For systems with dbm in the standard places, comment out the previous
# definitions, and uncomment these
#
```

```
#DBM_INC_PATH=/usr/include
#DBM_LIB_PATH=/usr/lib
#DBM_LIB_FILE=ndbm

# For systems where the gdbm libraries have been fetched and installed
# separately in the default locations under /usr/local
# comment out the previous definitions, and uncomment these
#DBM_INC_PATH=/usr/local/include
#DBM_LIB_PATH=/usr/local/lib
#DBM_LIB_FILE=gdbm


.c.o:
        $(CC) $(CFLAGS) -I$(DBM_INC_PATH) $(DFLAGS) -c $<

app_ui.o: app_ui.c cd_data.h
cd_dbm.o: cd_dbm.c cd_data.h
client_if.o: client_if.c cd_data.h cliserv.h
pipe_imp.o: pipe_imp.c cd_data.h cliserv.h
server.o: server.c cd_data.h cliserv.h


client_app: app_ui.o client_if.o pipe_imp.o
        $(CC) -o client_app  $(DFLAGS) app_ui.o client_if.o pipe_imp.o

server_app:   server.o cd_dbm.o pipe_imp.o
        $(CC) -o server_app -L$(DBM_LIB_PATH) $(DFLAGS) server.o cd_dbm.o pipe_imp.o
-l$(DBM_LIB_FILE)

clean:
        rm -f server_app client_app *.o *~
```

12.7.1 目标

我们的目的是把这个软件中与数据库打交道的部分和用户程序界面部分分开。我们还希望只运行一个服务器进程，但允许许多客户进程同时运行。我们将尽量减少对现有代码的修改，只要有可能，就保留原来的代码不做改动。

为了使事情简单化，我们还希望能够在软件里面来创建（和删除）管道，这样就不需要有一个系统管理员来为我们创建我们将会用到的命名管道了。

另外一个重要的问题是我们要保证不会出现繁忙等待某个事件的现象，减少对CPU资源的消耗。我们已经了解UNIX允许我们进入阻塞状态，使用最少的资源等待事件的发生。我们将利用管道的阻塞特性来保证CPU的有效使用。总之，服务器至少在理论上可以在客户请求到来之前等上许多个小时。

12.7.2 实现

我们曾经在第7章编写过这个软件，在那个单进程版本里，我们使用了一组数据访问例程来处理数据。这些例程包括：

```
int database_initialize(const int new_database);
void database_close(void);
cdc_entry get_cdc_entry(const char *cd_catalog_ptr);
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no);
int add_cdc_entry(const cdc_entry entry_to_add);
int add_cdt_entry(const cdt_entry entry_to_add);
int del_cdc_entry(const char *cd_catalog_ptr);
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no);
```

```
cdc_entry search_cdc_entry(const char *cd_catalog_ptr,
                           int *first_call_ptr);
```

这些函数正好可以让我们把客户和服务器两部分方便地分开。

这个软件的单进程版本被编译为一个单个的程序，但我们可以把它看做是由两个部分组成的系统，如图12-6所示。

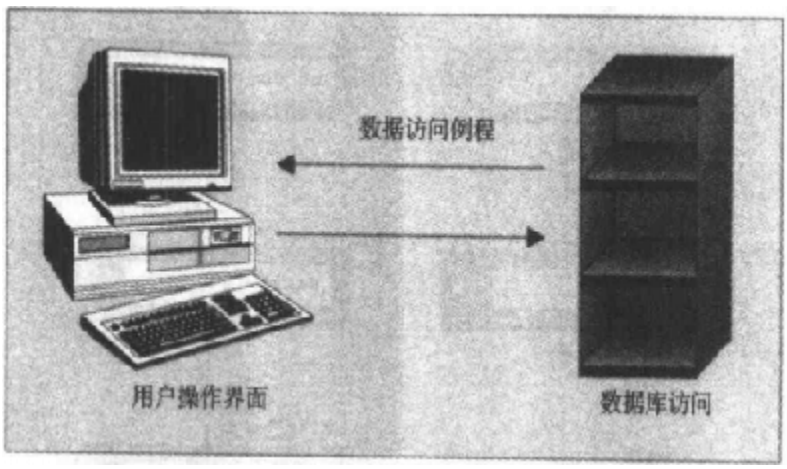


图 12-6

在客户-服务器实现里，我们想在软件的两个主要部分之间合乎逻辑地添上一些命名管道和支持性代码。

图12-7是我们需要的模型结构。

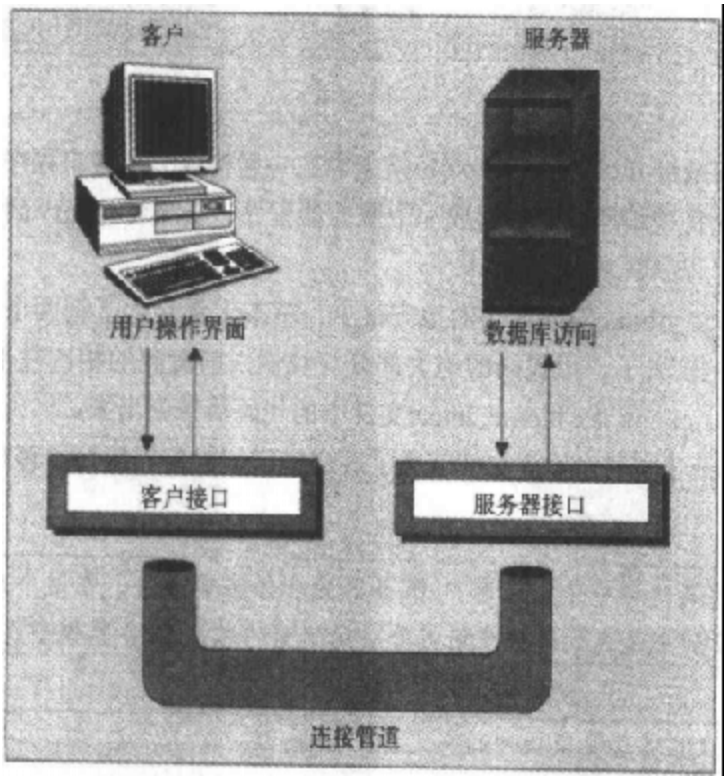


图 12-7

在我们的实现里，我们选择把客户接口例程和服务器接口例程都放在同一个文件里，即pipe_imp.c文件。这将把客户/服务器实现版本中与使用命名管道有关的代码都集中到一个文件里去，而被传输数据的格式编排和打包工作将与实现命名管道的例程隔离开来。我们的源文件更多了，但它们之间的界线也更符合逻辑了。这个软件里的调用结构如图12-8所示。

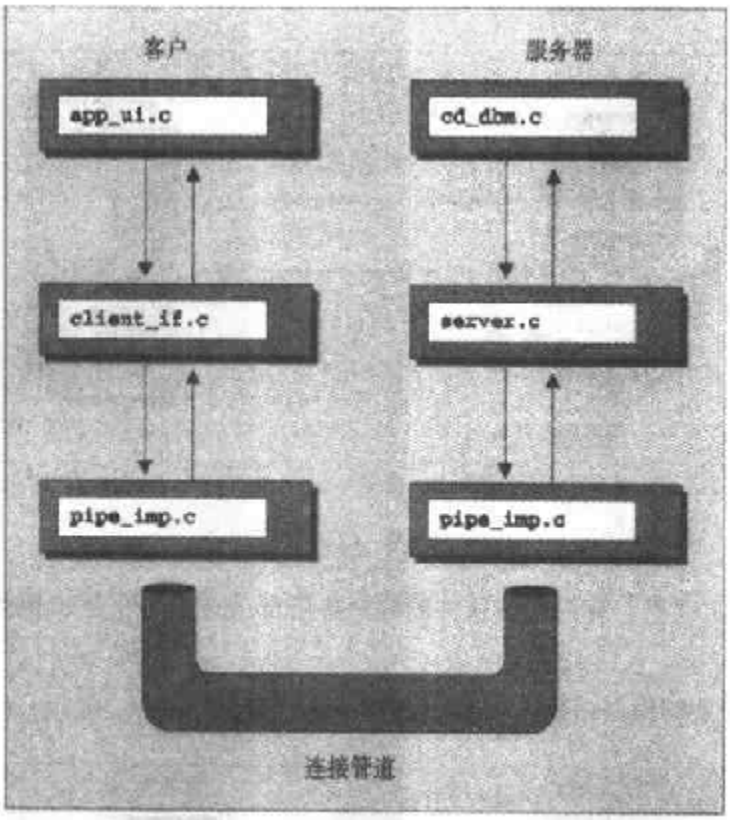


图 12-8

文件app_ui.c、client-if.c和pipe_imp.c将被编译在一起构成一个客户程序；而文件cd_dbm.c、server.c和pipe_imp.c将被编译在一起构成一个服务器程序；头文件cliserv.h将以一个公共定义头文件的形式把这两个程序联系起来。

文件app_ui.c和cd_dbm.c需要改动的地方很少，基本上都是为了把两个程序更好地分离开。这个软件现在已经比较大了，但代码的绝大部分与我们以前看到的相比并不需要改动，所以我们这里只把cliserv.h、client_if.c和pipe_imp.c文件中的代码清单列出来。

首先来看看cliserv.h文件。这个文件定义了客户接口和服务器接口。客户和服务器程序都要用到它。

这个文件的某些部分依赖于客户/服务器的具体实现办法，在这个例子里就是命名管道。在下一章的末尾我们还将改用另外一种不同的客户/服务器模型。

动手试试：CD唱盘管理软件的头文件：cliserv.h

1) 先是必要的系统头文件，如下所示：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
```

2) 接着是命名管道的定义。我们给服务器设置一个管道，给每个客户分别设置一个管道。因为可能会有多个客户，所以客户管道的名字里要加上它们各自的进程ID以保证其管道是独一无二的，如下所示：

```
#define SERVER_PIPE "/tmp/server_pipe"
#define CLIENT_PIPE "/tmp/client_%d_pipe"

#define ERR_TEXT_LEN 80
```

3) 我们把程序中的各个命令实现为枚举类型，而不是“# define”常数。

注意，这样做可以让编译器进行有关的类型检查，对软件的调试工作有帮助作用。这是因为许多调试器能够给出枚举类型常数的名字，但对“# define”指令定义的常数就不行。

第一个typedef结构定义的是向服务器发送的请求，第二个定义的是服务器返回给客户的响应，如下所示：

```
typedef enum {
    s_create_new_database = 0,
    s_get_cdc_entry,
    s_get_cdt_entry,
    s_add_cdc_entry,
    s_add_cdt_entry,
    s_del_cdc_entry,
    s_del_cdt_entry,
    s_find_cdc_entry
} client_request_e;

typedef enum {
    r_success = 0,
    r_failure,
    r_find_no_more
} server_response_e;
```

4) 接着，我们定义了一个结构，它将构成两个进程之间双向传递的数据信息。

> 因为不必在同一个响应里同时返回一个cdc_entry和一个cdt_entry，所以把它们组合在一起也是可行的。但出于简化问题的考虑，我们把它们分开来。这也使代码的维护工作比较容易进行一些。

```
typedef struct {
    pid_t               client_pid;
    client_request_e    request;
    server_response_e   response;
    cdc_entry           cdc_entry_data;
    cdt_entry           cdt_entry_data;
    char                error_text[ERR_TEXT_LEN + 1];
} message_db_t;
```

5) 最后是完成数据传输工作的各种管道接口函数，具体实现在pipe_imp.c文件里。它们又分

为服务器端和客户端两组，分别列在下面的第一部分和第二部分：

```
int server_starting(void);
void server_ending(void);
int read_request_from_client(message_db_t *rec_ptr);
int start_resp_to_client(const message_db_t mess_to_send);
int send_resp_to_client(const message_db_t mess_to_send);
void end_resp_to_client(void);

int client_starting(void);
void client_ending(void);
int send_mess_to_server(message_db_t mess_to_send);
int start_resp_from_server(void);
int read_resp_from_server(message_db_t *rec_ptr);
void end_resp_from_server(void);
```

我们把以后的讨论分为两大部分，一部分介绍客户接口函数，另一部分介绍pipe_imp.c文件中的服务器端和客户端函数的细节，我们会在必要时给出源代码。

12.7.3 客户接口函数

现在来看看client_if.c文件。它提供了“假”版本的数据库访问例程（“真”版本是由管道函数实现的）。它们对请求进行编码并把它放到一个message_db_t结构里去，然后通过pipe_imp.c中的例程把请求传输到服务器去。这样可以尽量减少对原来的app_ui.c进行修改。

动手试试：客户命令解释器

1) 这个文件实现了在cd_data.h文件里定义的9个数据库函数。它好比是一个中转站，先把请求传递给服务器，再从函数返回服务器的响应。

```
#define _POSIX_SOURCE

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cd_data.h"
#include "cliserv.h"
```

2) 静态变量mypid减少了getpig函数的调用次数，它可省了不少事。为了消除重复代码，我们还定义了一个局部函数read_one_response，如下所示：

```
static pid_t mypid;

static int read_one_response(message_db_t *rec_ptr);
```

3) 我们保留了database_initialize和database_close函数，但与以往的作用不同了。它们一个用来对管道接口的客户端进行初始化，另一个用来删除客户退出时不再有用的命名管道。

```
int database_initialize(const int new_database)
{
    if (!client_starting()) return(0);
    mypid = getpid();
```

```
    return(1);
} /* database_initialize */

void database_close(void) {
    client_ending();
}
```

4) 用一个给定的唱盘标题调用get_cdc_entry例程将从数据库里取出一个标题数据项。我们先把请求编码到一个message_db_t结构里并把它传递到服务器去，然后把服务器响应读回到另外一个message_db_t结构里来。如果能够在数据库里查到一个标题数据项，它将被存放在message_db_t结构的cdc_entry结构里，需要我们返回的就是这个东西。如下所示：

```
cdc_entry get_cdc_entry(const char *cd_catalog_ptr)
{
    cdc_entry ret_val;
    message_db_t mess_send;
    message_db_t mess_ret;

    ret_val.catalog[0] = '\0';
    mess_send.client_pid = mypid;
    mess_send.request = s_get_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                ret_val = mess_ret.cdc_entry_data;
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(ret_val);
}
```

5) 下面是read_one_response函数的程序清单，我们用它来减少重复代码，如下所示：

```
static int read_one_response(message_db_t *rec_ptr) {

    int return_code = 0;
    if (!rec_ptr) return(0);

    if (start_resp_from_server()) {
        if (read_resp_from_server(rec_ptr)) {
            return_code = 1;
        }
        end_resp_from_server();
    }
    return(return_code);
}
```

6) 其他get_xxx、del_xxx和add_xxx形式的例程实现起来与get_cdc_entry函数差不多。为保持其完整性，我们也把它们列在下面。头一个是用来检索CD曲目的那个函数：

```
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no )
```

```
{
    cdt_entry ret_val;
    message_db_t mess_send;
    message_db_t mess_ret;

    ret_val.catalog[0] = '\0';
    mess_send.client_pid = mypid;
    mess_send.request = s_get_cdt_entry;
    strcpy(mess_send.cdt_entry_data.catalog, cd_catalog_ptr);
    mess_send.cdt_entry_data.track_no = track_no;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                ret_val = mess_ret.cdt_entry_data;
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(ret_val);
}
```

7) 接下来是两个往数据库里添加数据的函数。第一个对应于标题数据库，第二个对应于曲目数据库。

```
int add_cdc_entry(const cdc_entry entry_to_add)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_add_cdc_entry;
    mess_send.cdc_entry_data = entry_to_add;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}

int add_cdt_entry(const cdt_entry entry_to_add)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_add_cdt_entry;
    mess_send.cdt_entry_data = entry_to_add;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
```

```
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

8) 最后是两个用来删除数据的函数。

```
int del_cdc_entry(const char *cd_catalog_ptr)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_del_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}


int del_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_del_cdt_entry;
    strcpy(mess_send.cdt_entry_data.catalog, cd_catalog_ptr);
    mess_send.cdt_entry_data.track_no = track_no;
    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

搜索数据库

根据CD唱盘关键字进行数据搜索的函数是比较复杂的。我们希望每开始一次新搜索的时候

只调用这个函数一次。我们在第7章里的做法是这样的：在第一次调用搜索函数的时候把“*first_call_ptr”设置为true，它将返回第一个匹配；搜索函数后续的每一次调用都要把“*first_call_ptr”设置为false，而返回的是其他匹配。

既然我们把整个软件划分为两个进程，我们在服务器里就不能再让搜索每次只前进一个数据项了，因为在前一次搜索正在进行的时候，可能会有另外一个客户请求服务器开始另外一次搜索。我们不能把来自不同客户的每一个搜索请求的上下文（即搜索操作当时到达的位置）都分别保存在服务器端，因为在搜索进行到半路的时候——比如用户已经找到了想找的CD唱盘数据、或者客户“摔倒”时，用户有可能会在客户端停止这次搜索。

我们可以改变搜索动作的执行方式，也可以象我们这里选择的那样把这些复杂的问题隐藏到接口例程里去。我们的做法是：安排服务器把一次搜索的可能匹配全部都找出来，把它们保存到客户端的一个临时文件里去，同一次查询的后续结果将在客户端处理。

> 我们原来的软件在使用中与SQL数据库的做法相类似，可以用一个光标在中间结果上移动。如果一次SQL查询产生了多个查询结果，那么在如何返回这些查询结果的问题上，这类系统也需要面对我们刚才遇见的设计抉择。

> 动手试试：搜索数据库

1) 这个函数实际上并不是很复杂，但看上去却让人眼花缭乱。它调用了三个管道函数（管道函数将在下一小节介绍），分别是send_mess_to_server、start_resp_from_server和read_resp_from_server，如下所示：

```
cdc_entry search_cdc_entry(const char *cd_catalog_ptr, int *first_call_ptr)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    static FILE *work_file = (FILE *)0;
    static int entries_matching = 0;
    cdc_entry ret_val;

    ret_val.catalog[0] = '\0';

    if (!work_file && (*first_call_ptr == 0)) return(ret_val);
```

2) 下面这个函数是开始一次搜索操作时用的第一个调用。它的“*first_call_ptr”标志是设置为true的，但一进入调用就立刻把它设置为了false——怕我们忘了。这个函数创建了一个work_file临时文件，然后对客户消息结构进行了初始化。

```
if (*first_call_ptr) {
    *first_call_ptr = 0;
    if (work_file) fclose(work_file);
    work_file = tmpfile();
    if (!work_file) return(ret_val);

    mess_send.client_pid = mypid;
    mess_send.request = s_find_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);
```

3) 接下来是三重条件判断，其作用是对刚才提到的三个管道函数的调用情况进行检查。如果向服务器发送消息成功，客户就开始等待服务器的响应。在从服务器读取查询结果的操作一直是成功的时候，这次搜索的全部匹配都会被保存到临时文件work_file里，匹配计数器entries_matching也会做相应的增加。

```
if (send_mess_to_server(mess_send)) {
    if (start_resp_from_server()) {
        while (read_resp_from_server(&mess_ret)) {
            if (mess_ret.response == r_success) {
           fwrite(&mess_ret.cdc_entry_data, sizeof(cdc_entry), 1, work_file);
                entries_matching++;
            } else {
                break;
            }
        } /* while */
    } else {
        fprintf(stderr, "Server not responding\n");
    }
} else {
    fprintf(stderr, "Server not accepting requests\n");
}
```

4) 接下来的测试检查搜索操作是否找到匹配数据。然后通过fseek调用设置work_file的下一个数据写入位置。

```
if (entries_matching == 0) {
    fclose(work_file);
    work_file = (FILE *)0;
    return(ret_val);
}
(void)fseek(work_file, 0L, SEEK_SET);
```

5) 如果这不是本次搜索操作过程中搜索函数的第一次调用，代码将检查是否还有其他匹配。最后，把下一个匹配数据项读到ret_val结构里去。此前的检查用来确定还有一个匹配数据项。

```
    } else {
            /* not *first_call_ptr */
        if (entries_matching == 0) {
            fclose(work_file);
            work_file = (FILE *)0;
            return(ret_val);
        }
    }

    fread(&ret_val, sizeof(cdc_entry), 1, work_file);
    entries_matching--;

    return(ret_val);
}
```

12.7.4 服务器接口

既然客户端有一个通往app_ui.c程序的操作界面，服务器端也需要有一个程序来控制cd_dbm.c程序（在以前的版本里叫做cd_access.c）。下面是服务器main函数的程序清单。

动手试试：server.c程序清单

1) 首先是全局变量、process_command函数的预定义和一个用来完成退出清理工作的信号捕

捉器。

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cd_data.h"
#include "cliserv.h"

int save_errno;
static int server_running = 1;

static void process_command(const message_db_t mess_command);

void catch_signals()
{
    server_running = 0;
}
```

2) 现在来到main函数面前。它先检查信号捕捉器例程能否正常工作，然后程序检查用户是否在命令行上输入了“-i”选项。如果有这个选项，它就创建一个新的数据库。如果在调用cd_dbm.c文件里的database_initilize函数时失败了，就给出一条出错信息。如果一切正常并且服务器也运转起来，来自客户的任何请求就会被送入process_command函数。我们马上就要讲到这个函数了。

```
int main(int argc, char *argv[]) {
    struct sigaction new_action, old_action;
    message_db_t mess_command;
    int database_init_type = 0;

    new_action.sa_handler = catch_signals;
    sigemptyset(&new_action.sa_mask);
    new_action.sa_flags = 0;
    if ((sigaction(SIGINT, &new_action, &old_action) != 0) ||
        (sigaction(SIGHUP, &new_action, &old_action) != 0) ||
        (sigaction(SIGTERM, &new_action, &old_action) != 0)) {
        fprintf(stderr, "Server startup error, signal catching failed\n");
        exit(EXIT_FAILURE);
    }

    if (argc > 1) {
        argv++;
        if (strncmp("-i", *argv, 2) == 0) database_init_type = 1;
    }
    if (!database_initialize(database_init_type)) {
                fprintf(stderr, "Server error:-\
                        could not initialize database\n");
                exit(EXIT_FAILURE);
    }

    if (!server_starting()) exit(EXIT_FAILURE);

    while(server_running) {
        if (read_request_from_client(&mess_command)) {
            process_command(mess_command);
        } else {
            if(server_running) fprintf(stderr, "Server ended - can not \
                                        read pipe\n");
```

```
            server_running = 0;
        }
    } /* while */
    server_ending();
    exit(EXIT_SUCCESS);
}
```

3) 任何客户消息都将被送入process_command函数，在那里它们被馈入一个case语句，进而调用cd_dbm.c文件中相应的函数。

```
static void process_command(const message_db_t comm)
{
    message_db_t resp;
    int first_time = 1;

    resp = comm; /* copy command back, then change resp as required */

    if (!start_resp_to_client(resp)) {
        fprintf(stderr, "Server Warning:-\
                start_resp_to_client %d failed\n", resp.client_pid);
        return;
    }

    resp.response = r_success;
    memset(resp.error_text, '\0', sizeof(resp.error_text));
    save_errno = 0;
    switch(resp.request) {
        case s_create_new_database:
            if (!database_initialize(1)) resp.response = r_failure;
            break;
        case s_get_cdc_entry:
            resp.cdc_entry_data =
                        get_cdc_entry(comm.cdc_entry_data.catalog);
            break;
        case s_get_cdt_entry:
            resp.cdt_entry_data =
                        get_cdt_entry(comm.cdt_entry_data.catalog,
                                      comm.cdt_entry_data.track_no);
            break;
        case s_add_cdc_entry:
            if (!add_cdc_entry(comm.cdc_entry_data)) resp.response =
                        r_failure;
            break;
        case s_add_cdt_entry:
            if (!add_cdt_entry(comm.cdt_entry_data)) resp.response =
                        r_failure;
            break;
        case s_del_cdc_entry:
            if (!del_cdc_entry(comm.cdc_entry_data.catalog)) resp.response
                        = r_failure;
            break;
        case s_del_cdt_entry:
            if (!del_cdt_entry(comm.cdt_entry_data.catalog,
                 comm.cdt_entry_data.track_no)) resp.response = r_failure;
            break;
        case s_find_cdc_entry:
            do {
                resp.cdc_entry_data =
                          search_cdc_entry(comm.cdc_entry_data.catalog,
                                           &first_time);
                if (resp.cdc_entry_data.catalog[0] != 0) {
                    resp.response = r_success;
                    if (!send_resp_to_client(resp)) {
                        fprintf(stderr, "Server Warning:-\
                            failed to respond to %d\n", resp.client_pid);
                        break;
                    }
                } else {
```

```
                    resp.response = r_find_no_more;
                }
            } while (resp.response == r_success);
        break;
        default:
            resp.response = r_failure;
            break;
    } /* switch */

    sprintf(resp.error_text, "Command failed:\n\t%s\n",
            strerror(save_errno));

    if (!send_resp_to_client(resp)) {
        fprintf(stderr, "Server Warning:-\
                failed to respond to %d\n", resp.client_pid);
    }

    end_resp_to_client();
    return;
}
```

在继续介绍管道的具体实现之前，我们先来看看在客户和服务器进程之间传递数据时各种事件发生的先后次序。图12-9给出了客户进程和服务器进程在各自启动之后，双方在处理命令和数据时的循环情况。

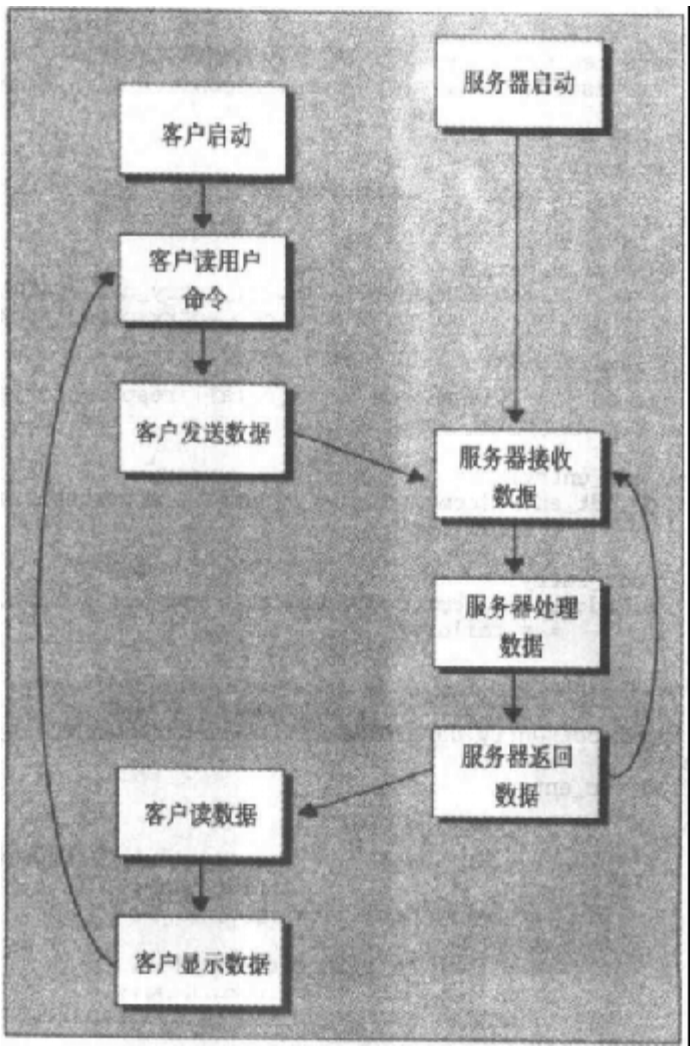


图 12-9

具体实现要比这个示意图中给出的情况稍微复杂一些，因为在查询操作中，客户只向服务器传递了一条命令，但服务器却可能发送回不止一个响应。这就增加了问题的复杂性，主要是在客户端。

12.7.5 管道

下面就是实现了管道功能的pipe_imp.c文件，它里面客户端和服务器端的函数都有。

> 我们曾经在第9章里见到过DEBUG_TRACE标志，如果在程序里定义了这个标志，就会在客户和服务器双方互相传递数据时显示出各个调用的执行顺序。

动手试试：管道函数使用的头文件

1) 首先是必要的头文件。

```
#include "cd_data.h"
#include "cliserv.h"
```

2) 我们还定义了一些值，这个文件里的那些函数会用到它们，如下所示：

```
static int server_fd = -1;
static pid_t mypid = 0;
static char client_pipe_name[PATH_MAX + 1] = {'\0'};
static int client_fd = -1;
static int client_write_fd = -1;
```

1. 服务器端函数

接下来，我们去看看服务器端的函数。我们把它分为了两个“动手试试”部分：第一部分里的函数其作用分别是打开命名管道、关闭命名管道和读取来自客户的消息；第二部分里的函数分别用来打开客户管道、向客户管道发送数据和关闭客户管道，客户管道是用客户添加在其请求消息里的进程ID值确定的。

动手试试：服务器函数

1) server_starting例程先为服务器创建出一个它将从中读取命令的命名管道，然后以只读方式打开它。这个open调用将阻塞到有客户以写方式打开这个管道时才能完成。我们使用的是一个阻塞模式，这样服务器将阻塞在这个管道的read调用上等待有命令发送过来。

```
int server_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_starting()\n",  getpid());
    #endif

        unlink(SERVER_PIPE);
    if (mkfifo(SERVER_PIPE, 0777) == -1) {
        fprintf(stderr, "Server startup error, no FIFO created\n");
        return(0);
    }

    if ((server_fd = open(SERVER_PIPE, O_RDONLY)) == -1) {
        if (errno == EINTR) return(0);
```

```
        fprintf(stderr, "Server startup error, no FIFO opened\n");
        return(0);
    }
    return(1);
}
```

2) 当服务器结束运行时，它会把自己的命名管道删除掉，这样客户就能检测出服务器不在运行中。

```
void server_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_ending()\n",  getpid());
    #endif

    (void)close(server_fd);
    (void)unlink(SERVER_PIPE);
}
```

3) 下面给出的read_request_from_client函数会阻塞服务器管道的读操作，直到有客户向其中写入一条消息为止。如下所示：

```
int read_request_from_client(message_db_t *rec_ptr)
{
    int return_code = 0;
    int read_bytes;

    #if DEBUG_TRACE
        printf("%d :- read_request_from_client()\n",  getpid());
    #endif

    if (server_fd != -1) {
        read_bytes = read(server_fd, rec_ptr, sizeof(*rec_ptr));

...

    }
    return(return_code);
}
```

4) 如果出现没有任何客户以写方式打开这个管道的特殊情况，read将返回“0”，也就是说，它会检测到一个EOF。服务器会据此关闭这个管道，然后再重新打开它。它用这个办法把自己阻塞起来，直到有客户打开它为止。这与服务器第一次启动时的情况是完全一样的；等于我们重新对服务器进行了初始化。把下面这些代码插到上面的函数里去：

```
if (read_bytes == 0) {
    (void)close(server_fd);
    if ((server_fd = open(SERVER_PIPE, O_RDONLY)) == -1) {
        if (errno != EINTR) {
            fprintf(stderr, "Server error, FIFO open failed\n");
        }
        return(0);
    }
    read_bytes = read(server_fd, rec_ptr, sizeof(*rec_ptr));
}
if (read_bytes == sizeof(*rec_ptr)) return_code = 1;
```

服务器是一个能够同时向许多客户提供服务的单进程。既然各个客户在接收自己的响应时使用的都是彼此不同的管道，服务器就必须能够向这些不同的管道写数据，这样才能把响应发送给不同的客户。又因为文件描述符是一种有限的资源，所以服务器只有在需要发送数据的时

候才打开一个客户管道。

我们把客户管道的打开、写和关闭正常分为三个独立的函数。这是为了适应数据库查询操作中有可能需要返回多个查询结果的要求，我们可以只打开管道一次，写入多个响应，然后再关闭它。

动手试试：管道操作

1) 首先，我们打开客户管道，如下所示：

```
int start_resp_to_client(const message_db_t mess_to_send)
{
    #if DEBUG_TRACE
        printf("%d :- start_resp_to_client()\n", getpid());
    #endif

    (void)sprintf(client_pipe_name, CLIENT_PIPE, mess_to_send.client_pid);
    if ((client_fd = open(client_pipe_name, O_WRONLY)) == -1) return(0);
    return(1);
}
```

2) 消息数据就是通过调用这个函数发送出去的。我们一会儿就能看到给消息数据划分数据域的客户端函数了。

```
int send_resp_to_client(const message_db_t mess_to_send)
{
    int write_bytes;

    #if DEBUG_TRACE
        printf("%d :- send_resp_to_client()\n", getpid());
    #endif

    if (client_fd == -1) return(0);
    write_bytes = write(client_fd, &mess_to_send, sizeof(mess_to_send));
    if (write_bytes != sizeof(mess_to_send)) return(0);
    return(1);
}
```

3) 最后，我们关闭客户管道，如下所示：

```
void end_resp_to_client(void)
{
    #if DEBUG_TRACE
        printf("%d :- end_resp_to_client()\n",  getpid());
    #endif

    if (client_fd != -1) {
        (void)close(client_fd);
        client_fd = -1;
    }
}
```

2. 客户端函数

pipe_imp.c文件里的客户函数与服务器是互相配合的。两部分函数很相似，请注意那个send_mess_to_server（发送消息给服务器）函数。

动手试试：客户函数

1) 在检查到有一个可以访问的服务器之后，client_starting函数对客户端管道进行了初始化，如下所示：

```
int client_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_starting\n",  getpid());
    #endif

    mypid = getpid();
    if ((server_fd = open(SERVER_PIPE, O_WRONLY)) == -1) {
        fprintf(stderr, "Server not running\n");
        return(0);
    }

    (void)sprintf(client_pipe_name, CLIENT_PIPE, mypid);
    (void)unlink(client_pipe_name);
    if (mkfifo(client_pipe_name, 0777) == -1) {
        fprintf(stderr, "Unable to create client pipe %s\n",
                client_pipe_name);
        return(0);
    }
    return(1);
}
```

2) client_ending函数的作用是关闭文件描述符并删除现在已经没有用处了的命名管道，如下所示：

```
void client_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_ending()\n",  getpid());
    #endif

    if (client_write_fd != -1) (void)close(client_write_fd);
    if (client_fd != -1) (void)close(client_fd);
    if (server_fd != -1) (void)close(server_fd);
    (void)unlink(client_pipe_name);
}
```

3) send_mess_to_server函数的作用是通过服务器管道发送请求，如下所示：

```
int send_mess_to_server(message_db_t mess_to_send)
{
    int write_bytes;

    #if DEBUG_TRACE
        printf("%d :- send_mess_to_server()\n",  getpid());
    #endif
    if (server_fd == -1) return(0);
    mess_to_send.client_pid = mypid;
    write_bytes = write(server_fd, &mess_to_send, sizeof(mess_to_send));
    if (write_bytes != sizeof(mess_to_send)) return(0);
    return(1);
}
```

与我们前面见过的服务器端函数相对应，为了处理可能会有多个搜索结果的情况，客户在从服务器取回结果的时候也用了三个函数。

动手试试：从服务器取回处理结果

1) 这个客户函数的作用是监听服务器的响应。它先以只读方式打开一个客户管道，然后又以只写方式再次打开了这个管道。请参考“操作注释”部分里的解释。

```
int start_resp_from_server(void )
```

```
{
    #if DEBUG_TRACE
        printf("%d :- start_resp_from_server()\n",  getpid());
    #endif

    if (client_pipe_name[0] == '\0') return(0);
    if (client_fd != -1) return(1);

    client_fd = open(client_pipe_name, O_RDONLY);
    if (client_fd != -1) {
        client_write_fd = open(client_pipe_name, O_WRONLY);
        if (client_write_fd != -1) return(1);
        (void)close(client_fd);
        client_fd = -1;
    }
    return(0);
}
```

2) 下面函数里的read调用是具体负责从服务器读取数据，它将取回匹配的数据库数据项。

```
int read_resp_from_server(message_db_t *rec_ptr)
{
    int read_bytes;
    int return_code = 0;

    #if DEBUG_TRACE
        printf("%d :- read_resp_from_server()\n",  getpid());
    #endif

    if (!rec_ptr) return(0);
    if (client_fd == -1) return(0);

    read_bytes = read(client_fd, rec_ptr, sizeof(*rec_ptr));
    if (read_bytes == sizeof(*rec_ptr)) return_code = 1;
    return(return_code);
}
```

3) 最后这个函数标记出服务器响应的结束。

```
void end_resp_from_server(void)
{
    #if DEBUG_TRACE
        printf("%d :- end_resp_from_server()\n",  getpid());
    #endif

    /* This function is empty in the pipe implementation */
}
```

操作注释：

start_resp_from_server函数里额外的第二个open调用是：

```
client_write_fd = open(client_pipe_name, O_WRONLY ) ;
```

它用来预防一个竞争现象的出现，这个竞争现象有可能在服务器需要响应来自同一个客户快速连续的请求时发生。

为了把这问题说清楚，我们来研究下面这个事件序列：

1) 客户把一个请求写到服务器管道。

2) 服务器读到这个请求，打开客户管道并且发回响应，但在及时关闭这个客户管道之前被挂起了。

3) 这个客户以读方式打开自己的管道，读取第一个响应并关闭了自己的管道。

4) 接着，这个客户又新发送了一个命令，再次以读方式打开自己的管道。

5) 此时，服务器恢复运行，从服务器端关闭了客户管道。

意外的事情发生了：此时客户正读着这个管道，等待着自己第二条请求的响应；但因为已经没有进程以写方式打开这个客户管道，这次的read调用将返回“0”字节。

通过允许客户以读和写两种方式同时打开自己的管道，就没有必要重复性地再次打开这个管道了，我们也就避免了这种竞争现象的发生。因为客户永远也不会真的向这个管道写数据，所以根本不会有读到不正确数据的危险。

12.7.6 对CD唱盘管理软件的总结

我们现在已经把这个CD唱盘管理软件分为一个客户和一个服务器了。这使我们能够互不干扰地对用户操作界面和内涵的数据库技术分别进行开发。我们可以看到：一个精心定义的数据库接口可以让软件的各个主要部分最好地使用计算机资源。如果我们想再前进一步，还可以把我们的管道实现方案改进为一个网络实现方案，并配上一个专用的数据库服务器计算机。我们将在第13章里对网络做进一步的学习。

12.8 本章总结

这一章的学习内容是如何利用管道在进程间传递数据。

首先，我们对通过popen或pipe调用创建的未命名管道进行了学习，并且讨论了如何利用管道和dup调用把数据从一个程序传递到另外一个程序的标准输入。

接下来，我们对命名管道进行了学习，学习了怎样才能在不相关的程序之间传递数据。

最后，我们实现了一个简单的客户/服务器架构的软件，FIFO文件给我们的不仅有进程间的同步，还有双向的数据流。

第13章　信号量、消息队列和共享内存

在这一章里，我们将对一组从AT&T System V.2版开始出现在UNIX操作系统中的进程间通信功能进行学习。因为这些功能都出现在同一个发行版本里，并且程序设计方面的接口也都很接近，所以人们又经常把它们称为IPC（InterProcess Communication，进程间通信）功能，或System V IPC。正如我们已经看到的，它们并不是进程之间进行通信的惟一方法，但人们经常把这一类别的功能统称为“System V IPC”。

13.1　信号量

如果我们编写的程序里使用了线程，那么不管它是运行在多用户系统上、多进程系统上，还是运行在多用户多进程系统上，只要我们需要保证只有一个执行进程能够拥有某项资源排他性的访问权时，就会发现自己面对着关键代码。

在第7章里，我们编写了这本书的第一个示例软件，它通过dbm函数库对一个数据库进行访问。如果有多个程序精确地在同一时间去尝试更新这个数据库，数据就会遭到破坏。不同的用户使用不同的程序向数据库输入数据，这本身并没有什么过错，问题就出在对数据库进行更新的那部分代码身上。这部分代码就是我们所说的关键代码。

为了防止出现因多个程序访问一个资源而引发的问题，我们需要有一种办法可以让我们生成并使用一个记号，使任一时刻关键代码里的执行线程只能有一个能够拥有该项资源的访问权。我们在第11章里简单地介绍了一些线程专用的办法，我们可以使用一把互斥量或者信号量来控制一个线程化程序对关键代码部分的访问权。在这一章里，我们又回到信号量的话题上，但将会对它们在进程之间的用法做更具普遍意义的讨论。注意：用在线程方面的信号量函数和我们将在这一章里介绍的根本就不是一回事，千万不要把这两种东西弄混了。

如果没有专家级硬件设备的支持，要想编写出能够达成这一目标的代码，用“难于上青天”来形容绝不算过分。虽然有一种名为“Dekker's Algorithm”（国内教材一般称之为德克算法）的纯软件算法，但这个算法必须依赖于“繁忙等待”或“轮转锁”。也就是说，进程必须无休止地运行着来等待某个内存数据的改变。在一个像UNIX这样的多任务系统上，人们并不愿意看到这种对CPU资源的浪费。

我们在以前的学习中曾经见过一个候选解决方案：创建文件时要使用open函数并且要加上O_EXCL标志，这个方案提供了原子化的文件创建操作，它可以使一个进程成功地获得一个标记物——那个新创建的文件。这个方案对简单问题来说还是不错的，但在复杂问题面前，它就比较麻烦和低效了。

Dijkstra提出的“信号量”概念是共发程序设计领域的一项重大进步。信号量是一种特殊的

变量，它只能取正整数值，对这些正整数只能进行两种操作：等待（wait）和信号（signal）。因为UNIX里的“wait”和“signal”都已经有了特殊的含义，所以我们在以后的内容里用早先的记号来表示信号量的这两种操作，它们是：（国内的教科书一般称之为“PV操作”）

- P(semaphore variable)　　代表等待
- V(semaphore variable)　　代表信号

这两个字母来自荷兰语单词passeren（等待、传递；在进入关键代码之前进行检查）和vrijgeven（信号、给与；放弃关键代码的控制权）。在提到信号量的时候你还经常会看到“开”或“关”等术语，它们沿用自开、关信号标志的说法。

13.1.1 信号量的定义

最简单的信号量是一个只能取“0”和“1”值的变量，也就是人们常说的“二进制信号量”，这也是它最常见的形式。可以取多种正整数值的信号量叫做“通用信号量”。在本章后面的内容里，我们将集中讨论二进制信号量。

PV操作的定义非常简明。假设我们有一个信号量变量sv，则这两个操作的定义如表13-1所示：

表 13-1

P(sv)	如果sv的值大于零，就给它减去1；如果sv的值等于零，就挂起该进程的执行
V(sv)	如果有其他进程因等待sv变量而被挂起，就让它恢复执行；如果没有进程因等待sv变量而被挂起，就给它加上1

我们还可以这样来看待信号量：当关键代码允许进程进入时，信号量变量sv的值是true；P(sv)操作对它做减法使它变为false，然后进程进入关键代码；进程离开关键代码时要用V(sv)操作对它做加法，使关键代码重新回到允许进程进入的状态。读者也许会认为对一个普通的变量进行类似的加减法也能够达到同样的效果，但这在C或者C++语言里不足以满足只用一个原子操作就能实现检查该变量是否为true或修改该变量使之为false的需要。而正是这一点才使得信号量的操作是如此特殊的。

13.1.2 一个理论性的例子

我们用一个简单的理论例子来说明它的工作情况。假设我们有两个进程，它们分别是proc1和proc2；这两个进程都需要在其执行过程中的某一时刻拥有对一个数据的独占性访问权。我们定义了一个二进制信号量sv，该变量的初始值是1，而这两个进程都能对它进行操作。要想对代码中的关键部分进行访问，这两个进程都需要采取同样的步骤。事实上，这两个进程可以是同一个程序的两个执行实例。

两个进程共享着sv信号量变量。如果其中之一执行了P(sv)操作，就等于它得到了信号量，也就能够进入关键代码部分了。第二个进程将无法进入关键代码，因为当它尝试执行P(sv)操作的时候，它会被挂起等待第一个进程离开关键代码并执行V(sv)操作释放这个信号量。

相应的伪代码如下所示：

```
semaphore sv = 1;

loop forever {
    P(sv);
    critical code section;
    V(sv);
    non-critical code section;
}
```

代码相当的简单，因为PV操作的定义非常明确，功能也非常有针对性。图13-1给出了用PV操作来把守进程进出关键代码部分时的情况。

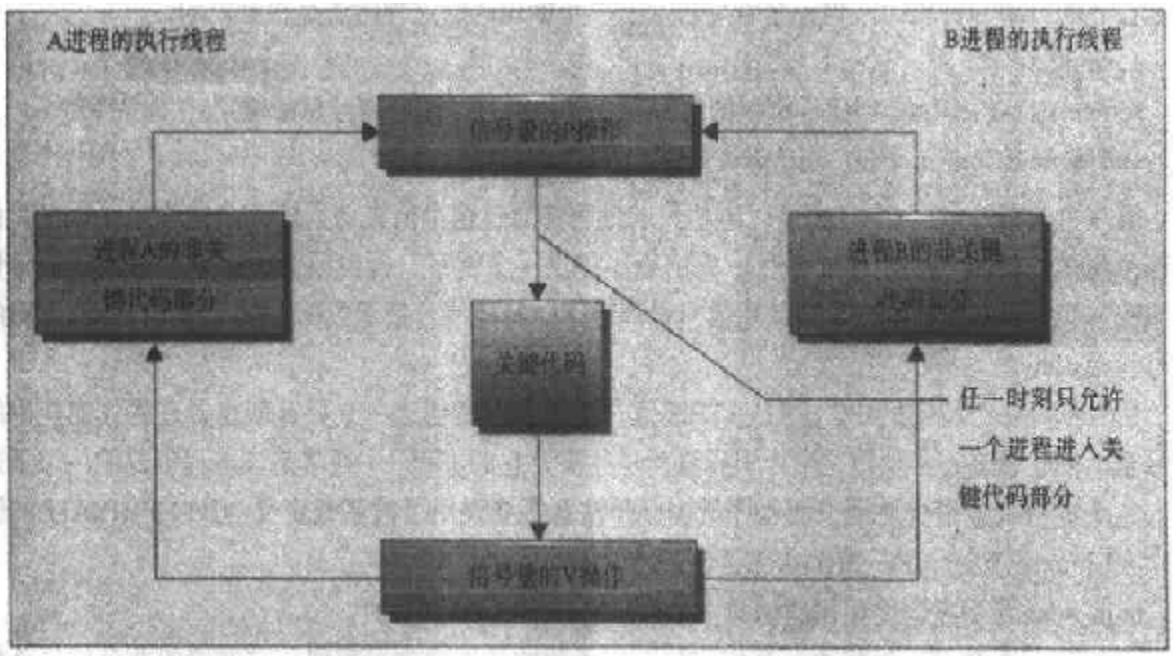


图 13-1

13.1.3 UNIX中的信号量功能

了解了信号量的含义和它们的工作原理之后，我们来看看在UNIX里是如何实现这些功能的。经过精心设计的接口提供了比一般使用要求更多的功能。UNIX中的每一个信号量函数都能对成组的通用信号量进行操作，最简单的二进制信号量对它来说自然更不在话下。这乍看起来好像会把事情弄得更复杂，但在复杂的问题里，比如一个进程需要锁定多个资源的时候，这种能够对一组信号量进行处理的能力就是求之不得的了。我们这一章的学习主要集中在一个信号量的方面，因为在绝大多数的情况下，有它也就足够大家用的了。

下面是信号量函数的定义情况：

```
#include <sys/sem.h>

int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

> 在实际工作中，还经常需要用到头文件sys/types.h和sys/ipc.h，某些特殊操作需要使用这两个头文件里的有关定义。完全不需要这两个头文件的情况还是比较少的。

在我们逐个介绍这些函数的时候，一定要记住这些函数都是能够对成组的信号量值进行操作的。与二进制信号量的情况相比，对多值信号量的操作要复杂得多的多。

键字参数key的作用很像是一个文件名，它代表着程序们使用的某个资源；如果程序们使用的信号量有着相同的key，就需要协调各自的执行情况了。同样地，由semget返回并用在其他共享内存函数中的那个标识码也与open返回的“FILE *”文件流极其相似，进程们需要通过它的值来访问共享的文件。此外，类似于文件的使用情况，哪怕使用的是同一个信号量，不同的进程也会有不同的信号量标识码。对我们将在这一章里讨论的每一个IPC功能来说，虽然它们使用的是独立的键字和标识码，但一个键字再加上一个标识码的这种用法是很常见的。

1. semget函数

semget函数的作用是创建一个新的信号量或者取得一个现有信号量的键字。

```
int semget(key_t key, int num_sems, int sem_flags );
```

第一个参数key是一个整数值，不相关的进程将通过这个值去访问同一个信号量。程序对任何信号量的访问都必须间接地进行，先由程序提供一个键字，再由系统生成一个相应的信号量标识码。只有semget函数才直接使用信号量的键字，其他信号量函数都必须使用由semget返回的信号量标识码。

信号量有一个特殊的IPC_PRIVATE键值，它的作用是创建一个只有创建者进程才能使用的信号量。创建者进程必须把这个标识码直接送往需要它的进程——通常是该进程创建的一个子进程。这个键值一般很少使用，而程序员也必须注意不要误用了被系统定义为IPC_PRIVATE的键值。在Linux系统上，IPC_PRIVATE键值通常是0。

num_sems参数是需要使用的信号量个数，它几乎总是取值为1。

sem_flags参数是一组标志，其作用与open函数的各种标志很相似。它低端的九个位是该信号量的权限，其作用相当于文件的访问权限。但它们可以与键值IPC_CREAT做按位的OR（或）操作以创建一个新的信号量。即使在设置了IPC_CREAT标志后给出的是一个现有的信号量的键字，也并不是一个错误。如果IPC_CREAT标志在函数里用不着，函数就会忽略它的作用。我们可以通过IPC_CREAT和IPC_EXCL标志的联合使用确保自己将创建出一个新的独一无二的信号量来；如果该信号量已经存在，就会返回一个错误。

semget函数在成功时将返回一个正数（非零）值，它就是其他信号量函数要用到的那个的标识码。如果失败，它将返回“-1”。

2. semop函数

semop函数的作用是改变信号量的键值，下面是它的定义情况：

```
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops );
```

第一个参数sem_id是该信号量的标识码，也就是semget函数的返回值。第二个参数sem_ops是个指向一个结构数组的指针，结构数组中的元素至少应该包含以下几个成员：

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

第一个成员sem_num是信号量的编号，如果你的工作不需要使用一组信号量，这个值一般就取为0。sem_op成员是信号量一次PV操作时加减的数值。你可以把一个信号量的计数值改变为一个非“1”的数字。一般只会用到两个值，一个是“-1”，也就是我们的P操作，等待信号量变得可用；另一个是“+1”，也就是我们的V操作，发出信号量已经变得可用的信号。

最后一个成员sem_flg通常被设置为SEM_UNDO。它将使操作系统跟踪当前进程对该信号量的修改情况。如果一个进程在没有释放信号量的情况下结束了执行，该进程掌握的信号量就将由操作系统自动释放。除非你对信号量的行为有特殊要求，否则就应该养成把sem_flg设置为SEM_UNDO的好习惯。如果你确实需要使用一个不是SEM_UNDO的值，就一定要注意保持其连贯性，否则就容易在操作系统是否需要在你的进程退出执行时替你清理信号量的问题上犯错误。

semop调用的一切动作都是一次性完成的，这是为了避免出现因使用了多个信号量而可能发生的竞争现象。semop的操作细节可以在使用手册页里找到。

3. semctl函数

semctl函数允许我们直接控制信号量的信息，下面是它的定义情况：

```
int semctl(int sem_id, int sem_num, int command, . . . );
```

第一个参数sem_id是由semget函数返回的一个信号量标识码。sem_num参数是信号量的编号，如果在工作中需要用到成组的信号量，就要用到这个编号；它一般取值为0，表示这是第一个也是惟一的信号量。command参数是将要采取的操作动作。如果还有第四个参数，那它将会是一个“union semun”复合结构，其中至少应该包含如下所示的几个成员：

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
}
```

某些Linux版本会在某个头文件里给出semun复合结构的定义，但X/Open却说这必须由程序员自己进行定义。如果读者需要定义自己的semun复合结构，就请查阅semctl的使用手册页，看看有没有已经给定的定义。如果有，我们建议你原封不动地使用其中给出的定义，即使它与我们这里给出的不一致也应该如此。下面是作者系统上使用的semun.h文件的内容，读者应该根据自己系统使用手册页中的说明行事，并在必要时对它进行修改。如下所示：

```
#ifndef _SEMUN_H

#define _SEMUN_H

union semun {
  int val;                    /* value for SETVAL */
  struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
  unsigned short int *array;  /* array for GETALL, SETALL */
  struct seminfo *__buf;      /* buffer for IPC_INFO */
};

#endif
```

semctl里的command可以有许多不同的值。我们下面介绍的是其中最常用的两个。semctl函

数的完整细节请查阅它的使用手册页。

command参数最常用的两个值是：

- SETVAL：用来把信号量初始化为一个已知的值。这个值在semun结构里是以val成员的面目传递的。它的作用是在信号量的第一次使用之前对它进行设置。
- IFC_RMID：删除一个已经没人继续使用的信号量标识码。

semctl函数会根据command参数返回好几种不同的值。就SETVAL和IFC_RMID来说，成功时它将返回“0”，失败时返回“-1”。

13.1.4 使用信号量

从上面的介绍可以看出，与信号量有关的操作可以是相当复杂的情况。这可不是什么好消息，如果多进程或者多线程里包含有关键代码，光是把它们编写好就已经是一个困难的问题，要是再加上一个复杂的程序接口，操心的事可就更多了。

但幸运的是大部分需要用信号量来解决的问题只用一个最简单的二进制信号量就能搞掂。在我们的示例里，我们将用全功能程序设计接口为二进制信号量编写一个简单很多的PV操作接口。然后，我们将通过这个比较简单的接口向大家展示信号量的工作情况。

我们将只使用一个程序sem1.c来检验我们的信号量，我们可以多次启动执行。我们将通过一个可选参数来指明信号量的创建和销毁工作是否需要由这个程序来负责。

我们用两种不同字符的输出来表示关键代码部分的进入和离开。如果程序启动时带有参数，就在进出关键代码部分时打印一个“X”；而程序的其他运行实例将在进出关键代码部分时打印一个“O”。因为在任一时刻只能有一个进程能够进入关键代码，所以字母“X”和“O”应该是成对出现的。

动手试试：信号量

1) 头文件、函数预定义和全局变量定义之后，我们到达main函数。我们用semget创建了一个信号量，它将返回一个信号量标识码。如果程序是第一个被调用的（即它在被调用的时候带着一个参数，使argc > 1），就调用set_semvalue对信号量进行初始化，并把变量op_char设置为“X”。

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

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#include "semun.h"

static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);

static int sem_id;


int main(int argc, char *argv[])
```

```
{
    int i;
    int pause_time;
    char op_char = 'O';

    srand((unsigned int)getpid());

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);

    if (argc > 1) {
        if (!set_semvalue()) {
            fprintf(stderr, "Failed to initialize semaphore\n");
            exit(EXIT_FAILURE);
        }
        op_char = 'X';
        sleep(2);
    }
```

2) 接下来是一个总共进出关键代码十次的循环语句。在每次循环的开始我们都要先做一次semaphore_p调用，因为程序将从此开始进入关键代码。

```
for(i = 0; i < 10; i++) {

    if (!semaphore_p()) exit(EXIT_FAILURE);
    printf("%c", op_char);fflush(stdout);
    pause_time = rand() % 3;
    sleep(pause_time);
    printf("%c", op_char);fflush(stdout);
```

3) 在经过一个随机等待时间之后我们准备离开关键代码，在进入下一次循环之前我们要调用semaphore_v把信号量设置为可用状态。整个循环语句执行完毕后，我们发出del_semaphore调用对代码进行清理。如下所示：

```
        if (!semaphore_v()) exit(EXIT_FAILURE);

        pause_time = rand() % 2;
        sleep(pause_time);
    }

    printf("\n%d - finished\n", getpid());

    if (argc > 1) {
        sleep(10);
        del_semvalue();
    }

    exit(EXIT_SUCCESS);
}
```

4) set_semvalue函数通过一个带SETVAL命令的semctl调用初始化信号量。在使用信号量之前我们必须这样做。如下所示：

```
static int set_semvalue(void)
{
    union semun sem_union;

    sem_union.val = 1;
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
    return(1);
}
```

5) del_semvalue函数的做法也差不多，只不过它是通过调用一个带IPC_RMID命令的semctl来删除那个信号量的标识码。如下所示：

```
static void del_semvalue(void)
{
    union semun sem_union;

    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}
```

6) semaphore_p对信号量做“-1”操作（等待）：

```
static int semaphore_p(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = -1; /* P() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}
```

7) semaphore_v把sembuf结构中的sem_op部分设置为“1”，从而使信号量变得可用。

```
static int semaphore_v(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = 1; /* V() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
    }
    return(1);
}
```

这个简单的程序只允许每个程序只有一个二进制信号量，如果需要用到更多的信号量，我们可以通过传递信号量变量的办法来扩展它。在大多数情况下，一个二进制信号量也就够用的了。

我们可以通过多次启动这个程序的办法来对它进行测试。第一次启动时要加上一个参数，表示应该由它来负责创建和删除信号量的工作。其他实例不需要参数。

下面是有两个程序实例在执行时的样本输出：

```
$ sem1 1 &
[1] 1082
$ sem1
OOXXOOXXOOXXOOXXOOXXOOOOXXOOXXOOXXOOXXXX
1083 - finished
1082 - finished
$
```

大家可以看到，字母“O”和“X”是成对出现的，这表明对关键代码的处理是正确的。如果它在读者的系统上不能运行，请在启动程序之前先试试“stty -tostop”命令，它的作用是确保会产生tty输出的后台程序不引起信号的产生。

操作注释：

在程序的开始，我们使用semget函数通过一个（随意选取的）键字得到了一个信号量标识码。IPC_CREAT标志的含义是：如果信号量不存在就创建它。

如果程序带着一个参数，信号量的初始化工作就将由它来负责。信号量的初始化工作由set_semvalue函数完成，它是更通用的semctl函数的简化接口。它还根据程序是否带有参数来判断需要它打印出哪个字符。sleep的作用是让我们有时间在程序的头一份拷贝执行太多循环次数之前调用它的其他拷贝。srand和rand的作用是给程序加上点伪随机形式的时间分配。

接下来程序循环了十次，在关键代码和非关键代码部分里会分别停留等待一段随机的时间。关键代码由我们的semaphore_p函数和semaphore_v函数前后把守着，它们是更通用的semop函数的简化接口。

在删除信号量之前，启动时带参数的程序会等待其他实例都执行完毕。如果不对信号量做删除操作，那即使没有程序在使用它，它也会继续存在于系统之中。这个问题应该引起大家高度的注意，在实际程序里，千万不要在执行之后还粗心地留下信号量没删除。它们会在你下次执行这个程序的时候引起问题；再说了，信号量也是一种需要大家节约使用的有限资源呢。

13.1.5 信号量总结

我们已经看到，信号量们有一个复杂的程序设计接口。但幸好我们还能为自己准备一个大大简化了的接口，而这个接口就已经足以解决大部分信号量的程序设计问题了。

13.2 共享内存

共享内存是三个IPC功能里面的第二个。它允许两个不相关的进程去访问同一部分逻辑内存。如果需要在两个运行中的进程之间传输数据，共享内存将是一种效率极高的解决方案。虽然X/Open标准并没有对它做出要求，但原因并不是它可有可无，而是因为大多数共享内存的具体实现都把由不同进程共享的内存安排为同一段物理内存的缘故。

13.2.1 概述

共享内存是由IPC为一个进程创建的一个特殊的地址范围，它将出现在进程的地址空间中。其他进程可以把同一段共享内存段“连接到”它们自己的地址空间里去。所有进程都可以访问共享内存中的地址，就好像它们是由malloc分配的一样。如果一个进程向这段共享内存写了数据，所做的改动会立刻被有权访问同一段共享内存的其他进程看到。

共享内存本身没有提供任何同步功能。也就是说，在第一个进程结束对共享内存的写操作之前，并没有什么自动功能能够预防第二个进程开始对它进行读操作。共享内存的访问同步问题必须由程序员负责。

图13-2中的箭头代表的是各进程的逻辑地址空间到可用物理内存的映射关系。实际情况要比这个示意图复杂的多，因为可用内存实际是由物理内存和已经交换到磁盘上的内存页面组成的。

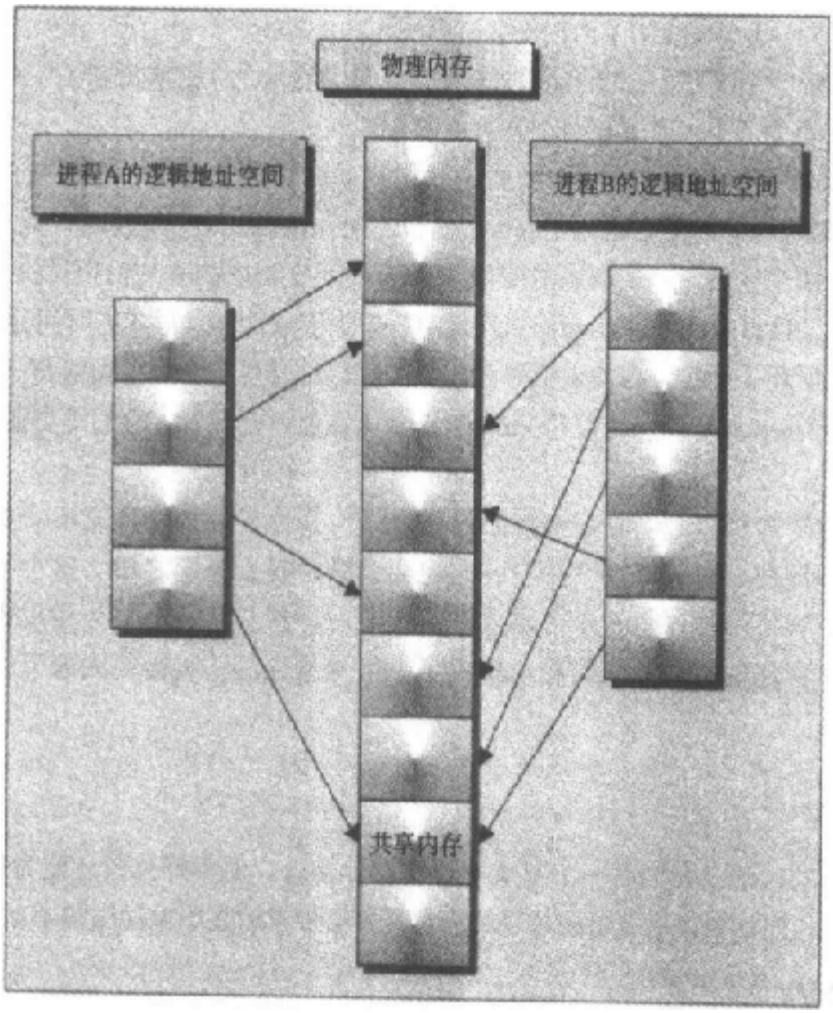


图 13-2

13.2.2 共享内存函数

共享内存使用的函数与信号量的很相似，请看它们的定义情况：

```
#include <sys/shm.h>

void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);
```

类似于信号量的情况，头文件sys/types.h和/sys/ipc.h一般也不能少。

1. shmget函数

我们通过shmget函数来创建共享内存：

```
int shmget(key_t key, size_t size, int shmflg ) ;
```

类似于信号量的情况，程序需要提供一个键字参数key，也就是这个共享内存段的名字，而shmget函数返回一个供后续共享内存函数使用的共享内存标识码。有一个特殊的键值IPC_PRIVATE，它的作用是创建本进程私用的共享内存。这个键值很少会用到，但一个进程与它

自己共享内存的事情也不是不可能。

第二个参数size以字节为单位给出了需要共享的内存量。

第三个参数shmflg由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的。由IPC_CREAT定义的特殊位必须与其他标志按位OR（或）在一起才能创建出一个新的的共享内存段来。设置TPC_CREAT标志并传递已存在的共享内存段的键字不会产生错误。如果IPC_CREAT标志用不着，就会忽略其作用。

权限标志对共享内存来说是非常有用的，因为它们允许一个进程创建出这样一种共享内存：允许共享内存的创建者用户所拥有的进程对这段共享内存进行写操作，但其他用户创建的进程却只能进行读操作。给共享内存加上相应的标志就可以既提供一种有效的数据只读访问措施，又不必冒其他用户可能会修改它的风险。

如果共享内存创建成功，shmget将返回一个非负整数，即该段共享内存的标识码；如果失败，返回“-1”。

2. shmat函数

在共享内存段刚被创建的时候，任何进程还都不能访问它。为了建立对这个共享内存段的访问渠道，必须由我们来把它连接到某个进程的地址空间。这项工作是由shmat函数完成的，下面是它的定义情况：

```
void *shmat(int shm_id, const void *shm_addr, int shmflg ) ;
```

第一个参数shm_id是shmget返回的共享内存标识码。

第二个参数shm_addr是把共享内存连接到当前进程去的时候准备放置它的那个地址。这通常是一个空指针，表示把选择共享内存出现处的地址这项工作交给系统去完成。

第三个参数shmflg是一组按位OR（或）在一起的标志。它的两个可能取值是SHM_RND（这个标志与shm_addr一起控制着共享内存连接的地址）和SHM_RDONLY（它使连接的共享内存成为一个只读区间）。很少有需要控制共享内存连接的地址的情况，一般都是由系统替你挑选一个地址；否则就会使你的软件对硬件的依赖性过高。

如果shmat调用操作成功，它将返回一个指针，指针指向共享内存的第一个字节；如果失败，它将返回“-1”。

共享内存的读写权限由它自己的属主（即共享内存的创建者）、它的访问权限和当前进程的属主情况来决定。共享内存的访问权限类似于文件的访问权限。

这里有一个例外，就是“shmflg & SHM_RDONLY”为true时的情况。此时这段共享内存将不允许写操作的执行，哪怕它的访问权限允许写操作都不行。

3. shmdt函数

shmdt函数的作用是把共享内存与当前进程脱离开。它的参数是一个由shmat返回的地址指针。如果操作成功，它将返回“0”，失败则返回“-1”。需要提醒大家的是脱离共享内存并不等于删除它，只是当前进程不能再继续访问它而已。

4. shmctl函数

与信号量那复杂的控制函数相比，共享内存的控制函数（谢天谢地）要简单得多。下面是它的定义：

```
int shmctl(int shm_id, int command, struct shmid_ds *buf ) ;
```

shmid_ds结构至少应该包含以下几个成员：

```
struct shmid_ds {
    uid_t shm_perm.uid;
    uid_t shm_perm.gid;
    mode_t shm_perm.mode;
}
```

第一个参数shm_id是由shmget函数返回的共享内存标识码。

第二个参数command是将要采取的动作。它有三个可取值（见表13-2）：

表 13-2

命 令	说 明
IPC_STAT	把shmid_ds结构中的数据设置为共享内存的当前关联值
IPC_SET	在进程有足够权限的前提下，把共享内存的当前关联值设置为shmid_ds数据结构中给出的值
IPC_RMID	删除共享内存段

第三个参数buf是一个指针，它指向一个保存着共享内存的模式状态和访问权限的数据结构。

如果操作成功，它将返回“0”，失败则返回“-1”。X/Open没有规定试图删除一个正处于连接状态的共享内存段时会发生什么事情。一般的做法是，这个已经被“删除”了的连接态共享内存段还能继续使用，直到从最后一个进程上脱离为止。但是，因为这种行为不属于技术规范里的规定，所以最好不要依赖它。

在学习了共享内存函数之后，我们来编写一个使用它们的程序。我们将编写两个程序shm1.c和shm2.c。第一个（消费者）将创建一个共享内存段，然后把写到它里面的数据都显示出来。第二个（加工厂）将连接一个现有的共享内存段，并允许我们向其中输入数据。

动手试试：共享内存

1）我们先来编写一个公共的头文件来定义我们将要用到的共享内存。我们给它起名为shm_com.h。如下所示：

```
#define TEXT_SZ 2048

struct shared_use_st {
    int written_by_you;
    char some_text[TEXT_SZ];
};
```

上面定义的结构“消费者”和“加工厂”两个程序都要用到。我们通过int（整数）类型的written_by_you标志来通知“消费者”已经有数据写入到这个结构的其他部分（即共享内存）里；而且我们决定需要传输多达2K的文本。

2）我们的第一个程序是那个“消费者”。在头文件之后，我们用一个shnget调用创建了共享内存段（其长度就是我们共享内存结构的长度），注意我们设置了IPC_CREAT标志位，如下所示：

```
#include <unistd.h>
```

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

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "shm_com.h"

int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    int shmid;

    srand((unsigned int)getpid());

    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }
```

3) 现在，让程序能够访问到这段共享内存：

```
shared_memory = shmat(shmid, (void *)0, 0);
if (shared_memory == (void *)-1) {
    fprintf(stderr, "shmat failed\n");
    exit(EXIT_FAILURE);
}

printf("Memory attached at %X\n", (int)shared_memory);
```

4) 接下来的程序把shared_memory赋值给shared_stuff，然后输出written_by_you中的文本。循环将一直执行到在written_by_you里找到“end”字样为止。sleep调用让“消费者”在关键代码里多待一会儿，让“加工厂”等在那里。

```
shared_stuff = (struct shared_use_st *)shared_memory;
shared_stuff->written_by_you = 0;
while(running) {
    if (shared_stuff->written_by_you) {
        printf("You wrote: %s", shared_stuff->some_text);
        sleep( rand() % 4 ); /* make the other process wait for us ! */
        shared_stuff->written_by_you = 0;
        if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
            running = 0;
        }
    }
}
```

5) 最后，共享内存从进程上脱离下来，随即被删除了，如下所示：

```
if (shmdt(shared_memory) == -1) {
    fprintf(stderr, "shmdt failed\n");
    exit(EXIT_FAILURE);
}

if (shmctl(shmid, IPC_RMID, 0) == -1) {
    fprintf(stderr, "shmctl(IPC_RMID) failed\n");
    exit(EXIT_FAILURE);
}
```

```
    exit(EXIT_SUCCESS);
}
```

6) 我们的第二个程序shm2.c是“加工厂”程序，我们通过它给“消费者”输入数据。它与shm1.c很相象，程序清单如下所示：

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

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "shm_com.h"

int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    char buffer[BUFSIZ];
    int shmid;

    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }

    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }

    printf("Memory attached at %X\n", (int)shared_memory);

    shared_stuff = (struct shared_use_st *)shared_memory;
    while(running) {
        while(shared_stuff->written_by_you == 1) {
            sleep(1);
            printf("waiting for client...\n");
        }
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);

        strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
        shared_stuff->written_by_you = 1;

        if (strncmp(buffer, "end", 3) == 0) {
                running = 0;
        }
    }

    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

在运行这个程序的时候，我们将看到如下所示的样本输出：

```
$ shm1 &
[1] 294
```

```
Memory attached at 50007000
$ shm2
Memory attached at 50007000
Enter some text: hello
You wrote: hello
waiting for client...
waiting for client...
Enter some text: Linux!
You wrote: Linux!
waiting for client...
waiting for client...
waiting for client...
Enter some text: end
You wrote: end
$
```

操作注释：

第一个程序shm1创建了一个共享内存段并且把它连接到自己的地址空间。我们在共享内存的开始处设置了一个shared_use_st结构，结构里有一个written_by_you标志；只要共享内存里有了数据，就设置这个标志。这个标志置位时，程序读取文本、把文本打印出来，再清除这个标志表示已经读完数据。我们用一个特殊的字符串“end”来退出循环。接下来，程序脱离共享内存段并删除了它。

第二个程序shm2通过同样的键字“1234”取得并连接上同一个共享内存段。然后提示用户输入一些文本。如果written_by_you标志处于置位状态，说明客户进程还没有把上一次的数据读完，还必须等待它。当另一个进程清除了这个标志之后，shm2写入新数据并置位该标志。它使用特别设定的“end”字符串来结束并脱离共享内存段。

这个程序里的共享内存同步机制是比较低效的“繁忙等待”（不停地循环）办法，同步标志是我们自己准备的整数变量written_by_you。我们可以在实际程序里把它改进为传递一条消息的办法（比如使用一个管道或IPC消息（马上就要讲到）、生成一个信号，或者使用一个信号量等办法）在程序读、写这两部分之间进行同步。

13.2.3 共享内存总结

共享内存为多个进程之间的数据共享和数据传递提供了一个高效率的解决方案。它本身不具备同步功能，所以我们需要用其他工具来同步对共享内存的访问和操作。典型的做法是：共享内存为一大片内存提供了高效的数据共享访问，而这片内存上的访问同步工作则由一小条消息来完成。

13.3 消息队列

我们来学习第三个也是最后一个IPC功能：消息队列。

13.3.1 概述

消息队列与命名管道有许多相似之处，但少了管道在打开和关闭方面的麻烦。但使用消息也并没有彻底解决我们在命名管道方面遇到的问题，比如管道满时的阻塞问题等。

消息队列提供了从一个进程向另外一个进程发送一块数据的方法。而且，每个数据块都被认为是有一个类型，接收者进程接收的数据块可以有不同的类型值。发送消息可以让使我们差不多完全回避命名管道上的同步和阻塞问题。更好的是，现在多少有了一些“预报”紧急消息的能力。但消息队列也有管道一样的不足，就是每个数据块的最大长度是有上限的，系统上全体队列的最大总长度也有一个上限。

X/Open技术规范规定了这些上限，但却没有提供检查发现这些上限的办法，只告诉我们超越这些限制会是某些消息队列功能失常的原因之一。Linux提供了两个常数定义MSGMAX和MSGMNB，分别代表了一条消息的最大字节长度和一个队列的最大长度。不同系统上的这些宏定义可能会不一样，甚至可能根本就没有。

13.3.2 消息队列函数

消息队列函数的定义如下所示：

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

与信号量和共享内存的情况类似，头文件sys/types.h和/sys/ipc.h一般也不能少。

1. msgget函数

我们用msgget函数来创建和访问一个消息队列：

```
int msgget (key_t key, int msgflg );
```

和其他IPC功能类似，必须由程序提供一个键字参数key，也就是某个消息队列的名字。特殊键值IPC_PRIVATE的作用是创建一个仅能由本进程访问的私用消息队列。第二个参数msgflg也由九个权限标志构成。由IPC_CREAT定义的特殊位必须与其他标志按位OR（或）在一起才能创建出一个新的消息队列。即使在设置了IPC_CREAT标志后给出的是一个现有的消息队列的键字，也并不是一个错误。如果该消息队列已经存在，就忽略IPC_CREAT标志作用。

如果操作成功，msgget将返回一个正整数，即一个消息队列标识码；如果失败，返回“-1”。

2. msgsnd函数

msgsnd函数的作用是让我们把一条消息添加到消息队列里去：

```
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

消息的结构在两方面受到制约。首先，它必须小于系统规定的上限值；其次，它必须以一个“long int”长整数开始，接收者函数将利用这个长整数确定消息的类型。在用到消息的时候，最好是把你的消息结构定义为下面这个样子：

```
struct my_message {
    long int message_type;
    /* The data you wish to transfer */
}
```

因为在接收消息时肯定要用到message_type，所以不能放着它不填。我们必须要在自己的数

据结构里加上这个长整数，最好是把它初始化为一个确定的已知值。

第一个参数msgid是由msgget函数返回的消息队列标识码。

第二个参数msg_ptr是一个指针，指针指向准备发送的消息，而消息必须像刚才说的那样以一个“long int”长整数开始。

第三个参数msg_sz是msg_ptr指向的消息的长度。这个长度不能把保存消息类型的那个“long int”长整数计算在内。

第四个参数msgflg控制着当前消息队列满或到达系统（在队列消息方面的）上限时将要发生的事情。如果msgflg中的IPC_NOWAIT标志被置位，这个函数就会立刻返回，消息不发了，返回值是“-1”；如果msgflg中的IPC_NOWAIT标志被清除，发送者进程就会挂起，等待队列中腾出空间来。

如果操作成功，这个函数将返回“0”；如果失败，返回“-1”。如果调用成功，就会对消息做一个拷贝并把它放到队列里去。

3. msgrcv函数

msgrcv函数的作用是从一个消息队列里检索消息：

```
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
```

第一个参数msqid是由msgget函数返回的消息队列标识码。

第二个参数msg_ptr是一个指针，指针指向准备接收的消息，而消息必须象前面msgsnd函数部分介绍的那样以一个“long int”长整数开始。

第三个参数msg_sz是msg_ptr指向的消息的长度，不包括保存消息类型的那个长整数。

第四个参数msgtype是一个“long int”长整数，它可以实现接收优先级的简单形式 如果msgtype的值是“0”，就提取队列中的第一个可用消息；如果它的值大于零，消息类型与之相同的第一个消息将被检索出来。如果它小于零，则消息类型值等于或小于msgtype的绝对值的第一个消息将被检索出来。

这样说着挺复杂，但用起来就好理解了。如果你只是想按照消息的发送顺序来检索它们，把msgtype设置为零就行了。如果你只是想检索某一特定类型的消息，把msgtype设置为相应的类型值就行。如果你想检索类型等于或小于“n”的消息，就把msgtype设置为“-n”。

第五个参数msgflg控制着队列中没有相应类型的消息可供接收时将要发生的事情。如果msgflg中的IPC_NOWAIT标志被置位，这个函数就会立刻返回，返回值是“-1”；如果msgflg中的IPC_NOWAIT标志被清除，接收者进程就会挂起，等待一条对应类型的消息到达。

如果操作成功，msgrcv函数将返回实际放到接收缓冲区里去的字符个数，而消息则被拷贝到msg_ptr指向的用户缓冲区里，然后删除队列里的数据。如果失败，返回“-1”。

4. msgctl函数

最后一个消息队列函数是msgctl，它的作用与共享内存的控制函数很相似。

```
int msgctl(int msqid, int command, struct msqid_ds *buf );
```

而msqid_ds结构至少应该包含以下成员：

```
struct msqid_ds {
    uid_t  msg_perm.uid;
```

```
    uid_t  msg_perm.gid
    mode_t msg_perm.mode;
}
```

第一个参数msqid是由msgget函数返回的消息队列标识码。

第二个参数command是将要采取的动作。它有三个可取值如表13-3所示：

表 13-3

命 令	说 明
IPC_STAT	把msqid_ds结构中的数据设置为消息队列的当前关联值
IPC_SET	在进程有足够权限的前提下，把消息队列的当前关联值设置为msqid_ds数据结构中给出的值
IPC_RMID	删除消息队列

如果操作成功，它将返回“0”，失败则返回“-1”。如果删除一个消息队列的时候还有进程等在msgsnd或msgrcv函数里，这两个函数将会失败。

在学习了消息队列的定义之后，我们来看它们的实际工作情况。和前面一样，我们将编写两个程序，一个是接收消息用的msg1.c，另一个是发送消息用的msg2.c。我们将允许两个程序都能创建消息队列，但最后由接收者在接收完最后一条消息后删除它。

动手试试：消息队列

1) 下面是接收者的程序清单：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>


struct my_msg_st {
    long int my_msg_type;
    char some_text[BUFSIZ];
};

int main()
{
    int running = 1;
    int msgid;
    struct my_msg_st some_data;
    long int msg_to_receive = 0;
```

2) 首先，我们来创建消息队列：

```
msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

if (msgid == -1) {
    fprintf(stderr, "msgget failed with error: %d\n", errno);
    exit(EXIT_FAILURE);
}
```

3) 接着从队列里检索消息，直到遇见“end”消息为止。最后，删除消息队列。

```
    while(running) {
        if (msgrcv(msgid, (void *)&some_data, BUFSIZ,
                   msg_to_receive, 0) == -1) {
            fprintf(stderr, "msgrcv failed with error: %d\n", errno);
            exit(EXIT_FAILURE);
        }
        printf("You wrote: %s", some_data.some_text);
        if (strncmp(some_data.some_text, "end", 3) == 0) {
            running = 0;
        }
    }

    if (msgctl(msgid, IPC_RMID, 0) == -1) {
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

4) 发送者程序与msg1.c很相似。在main函数的初始化部分，去掉对msg_to_receive的定义并把它换为buffer[BUFSIZ]，去掉删除消息队列的语句，然后在running循环里做以下修改。最后得到的是一个发送用户输入文本到队列的magsnd调用，如下所示：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

struct my_msg_st {
    long int my_msg_type;
    char some_text[MAX_TEXT];
};

int main()
{
    int running = 1;
    struct my_msg_st some_data;
    int msgid;
    char buffer[BUFSIZ];
msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

    if (msgid == -1) {
        fprintf(stderr, "msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }

    while(running) {
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);
        some_data.my_msg_type = 1;
        strcpy(some_data.some_text, buffer);

        if (msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1) {
            fprintf(stderr, "msgsnd failed\n");
            exit(EXIT_FAILURE);
        }
```

```
            if (strncmp(buffer, "end", 3) == 0) {
                running = 0;
            }
        }

        exit(EXIT_SUCCESS);
    }
```

与管道例子不同的是，不再需要由进程自己来提供同步机制了。这是消息相对于管道的一大优势。

发送者按计划创建出消息队列，在队列里放上一些数据，然后在接收者启动之前就退出了。我们先运行发送者msg2程序。下面是一些样本输出：

```
$ msg2
Enter some text: hello
Enter some text: How are you today?
Enter some text: end
$ msg1
You wrote: hello
You wrote: How are you today?
You wrote: end
$
```

操作注释：

发送者程序用msgget函数创建了一个消息队列，然后用msgsnd往队列里放上一些消息。接收者用msgget获得那个消息队列的标识码，然后开始接收消息，直到接收到特殊的“end”字样出现为止。此时它将用msgctl函数来删除消息队列，打扫战场。

13.3.3 消息队列总结

在两个不相关的进程之间传递数据有许多种办法，消息队列是其中比较简单同时也比较高效的那一种。与命名管道相比，消息队列的优势在于它们既不依赖于发送进程，也不依赖于接收进程，它们自己是可以独立存在的，这就省掉了在打开和关闭命名管道时必不可少的同步与协调环节。

13.4 应用示例

我们准备用在这一章里学到IPC功能来改造我们的CD唱盘管理软件。

我们可以通过这三种IPC功能各种不同的组合来改造我们的软件，但考虑到被传递的信息量实在很少，所以直接使用消息队列来实现请求和响应的传递工作应该是比较合情合理的。

如果需要我们传递的数据量很大，就可以考虑用共享内存来传递实际数据，再用信号量或者消息来传递一个“令牌”去通知有关进程数据已经在共享内存里准备好了。

消息队列省去了我们在第11章里遇到的问题，那时我们需要在数据传递过程中有两个进程都打开着管道。而使用消息队列就不同了，即使仅有一个进程是消息队列的当前用户，消息队列也允许它往队列里放消息。

惟一需要我们做出的重大决定是怎样向客户返回查询结果。一个简单的做法是让服务器用一个队列，每个客户用一个队列。但如果同时运行的客户数量太大，就会因申请分配的消息队

列数量太大而引起问题。要是把消息中的消息ID域用起来，我们就能对消息进行“编址”。我们的做法是：让全体消息都只使用一个队列，在每条消息里加上客户的进程ID，这样就把响应消息和某个特定的进程联系起来了。每个客户只检索那些发送给它的消息，发送给其他进程的消息就留在队列里吧。

要想把CD管理软件改造成使用IPC功能的，只需更换pipe_imp.c文件。在以下几页内容里，我们将对替换文件ipc_imp.c的主要组成部分进行说明。

动手试试：修订服务器函数

1) 首先，我们要包括上必要的头文件，定义一些消息队列的键字，还要定义一个用来保存我们消息数据的结构。如下所示：

```
#include "cd_data.h"
#include "cliserv.h"

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define SERVER_MQUEUE 1234
#define CLIENT_MQUEUE 4321
struct msg_passed {
    long int msg_key; /* used for client pid */
    message_db_t real_message;
};
```

2) 下面这两个设定了文件作用范围的变量将分别保存着msgget函数返回的两个队列标识码。如下所示：

```
static int serv_qid = -1;
static int cli_qid = -1;
```

3) 我们决定由服务器负责创建那两个消息队列，如下所示：

```
int server_starting()
{
    #if DEBUG_TRACE
        printf("%d :- server_starting()\n", getpid());
    #endif

    serv_qid = msgget((key_t)SERVER_MQUEUE, 0666 | IPC_CREAT);
    if (serv_qid == -1) return(0);

    cli_qid = msgget((key_t)CLIENT_MQUEUE, 0666 | IPC_CREAT);
    if (cli_qid == -1) return(0);

    return(1);
}
```

4) 服务器还负责在退出时打扫战场。在服务器的扫尾工作中，我们把那两个文件范围变量设置为非法的值；如果服务器在它自己已经调用过server_ending之后还试图发送消息，这种做法就可以捕捉到它们。

```
void server_ending()
{
    #if DEBUG_TRACE
        printf("%d :- server_ending()\n", getpid());
```

```
    #endif

    (void)msgctl(serv_qid, IPC_RMID, 0);
    (void)msgctl(cli_qid, IPC_RMID, 0);

    serv_qid = -1;
    cli_qid = -1;
}
```

5) 服务器读函数的作用是：从队列里读一个任意类型（意思是来自任意客户）的消息，返回消息的数据部分（忽略消息的类型）。

```
int read_request_from_client(message_db_t *rec_ptr)
{
    struct msg_passed my_msg;
    #if DEBUG_TRACE
        printf("%d :- read_request_from_client()\n", getpid());
    #endif

    if (msgrcv(serv_qid, (void *)&my_msg, sizeof(*rec_ptr), 0, 0) == -1) {
        return(0);
    }
    *rec_ptr = my_msg.real_message;
    return(1);
}
```

6) 发送响应时要用客户的ID值对消息进行“编址”，进程ID值是客户放在自己的请求里送过来的。如下所示：

```
int send_resp_to_client(const message_db_t mess_to_send)
{
    struct msg_passed my_msg;
    #if DEBUG_TRACE
        printf("%d :- send_resp_to_client()\n", getpid());
    #endif

    my_msg.real_message = mess_to_send;
    my_msg.msg_key = mess_to_send.client_pid;

    if (msgsnd(cli_qid, (void *)&my_msg, sizeof(mess_to_send), 0) == -1) {
        return(0);
    }
    return(1);
}
```

动手试试：修订客户函数

1) 当客户启动的时候，它需要找到服务器和客户的队列标识码。客户本身并不创建消息队列。如果服务器没有运行，这个函数就会因消息队列没有存在而失败。

```
int client_starting()
{
    #if DEBUG_TRACE
        printf("%d :- client_starting\n", getpid());
    #endif

    serv_qid = msgget((key_t)SERVER_MQUEUE, 0666);
    if (serv_qid == -1) return(0);

    cli_qid = msgget((key_t)CLIENT_MQUEUE, 0666);
    if (cli_qid == -1) return(0);
    return(1);
}
```

2) 与服务器中的做法一样，我们在客户结束的时候也把文件范围变量设置为非法的值。如果客户在它自己已经调用过client_ending之后还试图发送消息，这种做法就可以捕捉到它们。

```
void client_ending()
{
    #if DEBUG_TRACE
        printf("%d :- client_ending()\n",  getpid());
    #endif

    serv_qid = -1;
    cli_qid = -1;
}
```

3) 要想把消息发送给服务器，我们必须先把有关数据保存到一个结构里去。我们必须设置消息的键字。因为“0”对键字来说是一个非法的值，而不对这个键字进行定义又意味着它可以随机地取任意值，所以如果碰巧这个值是“0”的话，这个函数就会调用失败。因此，我们一定要对键字进行赋值。

```
int send_mess_to_server(message_db_t mess_to_send)
{
    struct msg_passed my_msg;
    #if DEBUG_TRACE
        printf("%d :- send_mess_to_server()\n",  getpid());
    #endif

    my_msg.real_message = mess_to_send;
    my_msg.msg_key = mess_to_send.client_pid;

    if (msgsnd(serv_qid, (void *)&my_msg, sizeof(mess_to_send), 0) == -1) {
        perror("Message send failed");
        return(0);
    }
    return(1);
}
```

4) 当客户从服务器检索一个消息的时候，它通过自己的进程ID只接收发送给它的消息，不理睬发送给其他用户的消息。

```
int read_resp_from_server(message_db_t *rec_ptr)
{
    struct msg_passed my_msg;
    #if DEBUG_TRACE
        printf("%d :- read_resp_from_server()\n",  getpid());
    #endif

    if (msgrcv(cli_qid, (void *)&my_msg, sizeof(*rec_ptr), getpid(), 0) == -1) {
        return(0);
    }
    *rec_ptr = my_msg.real_message;
    return(1);
}
```

5) 为了实现与pipe_imp.c百分之百的兼容，我们还需要额外定义四个函数。在我们的新程序里，这几个函数应该是空的。它们在使用管道时实现的操作现在已经用不着了。

```
int start_resp_to_client(const message_db_t mess_to_send)
{
    return(1);
}

void end_resp_to_client(void)
```

```
{
}

int start_resp_from_server(void)
{
    return(1);
}

void end_resp_from_server(void)
{
}
```

CD唱盘管理软件面向消息队列的改造展示了IPC消息队列的强大功能。我们需要使用的函数少了，就是还在使用的也比它们以前小得多了。

13.5 查看IPC功能状态的命令

虽然没有在X/OPen技术规范里得到规定，但大多数带有信号量功能的UNIX系统都提供了一组用来从命令行上访问IPC功能的命令。它们是ipcs和ipcrm命令，在你开发程序时肯定会发挥重要的作用。

13.5.1 信号量

如果想对系统上的信号量状态进行检查，可以使用“ipcs -s”命令。如果存在着信号量，就会给出如下格式的输出：

```
$ ipcs -s
------ Semaphore Arrays --------
semid     owner     perms     nsems     status
768       rick      666       1
```

如果因为意外原因程序遗漏了没有清除的信号量，ipcrm命令可以帮你清除它们。清除这些漏网信号量的（Linux）命令如下所示：

```
$ ipcrm sem 768
```

许多UNIX系统用的是：

```
$ ipcrm -s 768
```

13.5.2 共享内存

类似于信号量，许多系统提供了命令行程序来访问共享内存的细节情况。其中包括“ipcs -m”和“ipcrm shm <id>”（这个命令还可以写为“ipcrm -m <id>”的形式）等。

下面是几个ipcs命令的示范性输出：

```
$ ipcs -m

------ Shared Memory Segments --------
shmid     owner     perms     bytes     nattch     status
384       rick      666       4096      2
```

这里给出的情况是：只有一个长度为4KB的共享内存段，它连接了两个进程。

“ipcrm shm <id>”命令的作用是删除共享内存段。如果程序因运行失败而没有来得及清理

共享内存，你可以让这个命令显一下身手。

13.5.3 消息队列

消息队列用的命令有“ipcs -q”和“ipcrm msg <id>”（这个命令还可以写为“ipcrm -q <id>”的形式）等。

下面是ipcs命令的一些示范性输出：

```
$ ipcs -q

------ Message Queues --------
msqid      owner      perms      used-bytes   messages
384        rick       666        2048         2
```

这里给出的情况是：有两个消息，消息队列的总长度是2048个字节。

“ipcrm msg <id>”命令的作用是删除消息队列。

13.6 本章总结

在这一章里，我们学习了三种进程间的通信功能，它们最早出现在UNIX System V.2版本里，即信号量、共享内存和消息队列。我们对它们提供的复杂功能进行了学习。一旦掌握了这些函数的使用方法，就能用它们来解决许多进程间通信方面的问题。

第14章 套 接 字

在这一章里，我们将学习进程间通信的另外一种方法，它与以前学习的方法相比有一个明显的不同。以前介绍的通信功能都要靠单台计算机系统上的共享资源才能实现。这个资源多种多样，它可以是文件系统空间、可以是共享的物理内存空间、还可以是消息队列；但都是只有运行在同一机器上的进程才能使用它们。

Berkeley版的UNIX引入了一种全新的通信手段，它就是套接字接口（socket interface），它是管道概念的一个扩展。你完全可以把套接字当作管道来用，但套接字还涵盖了计算机网络中的通信。一台机器上的某个进程可以使用套接字与另一台机器上的某个进程进行通信，这就使客户/服务器系统可以分布到整个网络。同一机器上的进程也可以使用套接字进行通信。

此外，微软公司的Windows也通过公开的Windows Sockets技术标准（简称WinSock标准）实现了套接字接口，它的套接字服务是通过系统文件Winsock.dll提供的。这样，微软Windows程序和UNIX计算机就可以相互跨网络通信，实现客户/服务器系统。WinSock的程序设计接口与UNIX套接字不尽相同，但它同样是以套接字为基础的。

只用一章的篇幅是不可能把UNIX丰富的网络功能都讨论完的，而这一章的目的只是把网络方面主要的程序接口介绍给大家。掌握了这些程序设计接口，就能开始编写你们自己的网络程序了。我们将要学习的内容包括：

- 套接字连接的操作原理。
- 套接字的属性、地址和通信。
- 网络信息和因特网守护进程。
- 客户和服务器。

14.1 什么是套接字

套接字是这样一种通信过程，它使客户/服务器系统的开发工作既可以在本地单机上进行，也可以跨网络进行。UNIX函数（比如打印输出类函数）和rlogin、ftp等网络工具基本上都是通过套接字来进行通信的。

套接字的创建和使用与管道是有区别的，分处套接字两端的客户和服务器之间有明确的区别。套接字机制可以把多个客户连接到一个服务器。

14.2 套接字连接

我们可以把套接字连接想象成一座办公大楼里的电话。一个电话打到一家公司，接线员接通电话并把它转到正确部门（服务器进程），再从那转给电话要找的人（服务器套接字）。每个打进来的电话（客户）都被转到正确的分机，而总机接线员则被空出来去处理以后打进来的电

话。在我们开始研究UNIX系统里的套接字连接是如何建立起来的之前，我们先要弄明白套接字软件是如何处理一个连接的。

首先，服务器软件必须先创建出一个套接字，这是分配给该服务器进程的一个操作系统资源，因为这个套接字是由该服务器通过系统调用socket创建出来的，所以其他进程将不能对它进行访问。

接着，服务器进程会给套接字起个名字。给本地套接字起的名字是UNIX文件系统中的一个文件名，一般放在/tmp或/usr/tmp子目录里。而网络套接字的名字则是一个与客户所能连接的特定网络有关的服务标识符（也叫做端口号或访问点）。给套接字起名字（这个操作叫做“绑定”）要使用系统调用bind。然后，服务器就开始等待有客户连接到这个命名套接字上来。系统调用listen的作用是创建一个队列，来自客户的连接（接入连接）将在这个队列上排队等待服务器的处理（这个过程叫做“监听”）。服务器将通过系统调用accept来接受来自客户的接入连接。

当服务器调用accept的时候，会新创建一个套接字，这个套接字与刚才说的命名套接字不是一回事。新套接字的惟一用途就是与这个特定的客户进行通信，而命名套接字则被解放出来，准备处理来自其他客户的连接。如果服务器编写得当，就可以享受多个连接带来的好处。对一个简单的服务器来说，后来的客户需要在队列里等待服务器的重新就绪。

基于套接字系统的客户端就比较简单了。客户先通过调用socket创建出一个未命名套接字，然后调用connect利用服务器的命名套接字和一个地址来建立起一个连接。

套接字被建立起来之后，人们就可以象对待底层文件描述符那样用它来实现双向的数据通信了。

下面是一个非常简单的套接字客户程序client1.c。它创建了一个未命名套接字，然后把它连接到一个名为server_socket的服务器套接字。我们将先向大家介绍一些地址设定方面的知识，然后再讨论socket系统调用的细节。

动手试试：一个简单的本地客户

1) 必要的头文件和变量的初始化。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
    int sockfd;
    int len;
    struct sockaddr_un address;
    int result;
    char ch = 'A';
```

2) 为客户创建一个套接字。

```
sockfd = socket(AF_UNIX, SOCK_STREAM, 0 );
```

3) 根据服务器的设置情况给这个套接字起个名字：

```
address.sun_family = AF_UNIX;
strcpy(address.sun_path, "server_socket");
len = sizeof(address);
```

4) 现在，把我们的套接字连接到服务器的套接字。

```
result = connect(sockfd, (struct sockaddr *)&address, len);

if(result == -1) {
    perror("oops: client1");
    exit(1);
}
```

5) 现在就可以通过sockfd进行读写操作了。

```
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

这个程序在运行的时候会失败，这是因为我们还没有创建出服务器端的命名套接字（错误信息的正确文字会随系统的不同而不同）。如下所示：

```
$ client1
oops: client1: Connection refused
$
```

下面是一个简单的服务器程序server1.c，它的作用是接受来自我们这个客户的连接。它创建出一个服务器套接字，把它绑定到一个名字上，再创建一个监听队列，做好接受连接的准备工作。

动手试试：一个简单的本地服务器

1) 必要的头文件和变量的初始化。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_un server_address;
    struct sockaddr_un client_address;
```

2) 删掉以前的套接字，再为服务器新创建一个未命名套接字。

```
unlink("server_socket");
server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

3) 给套接字起名字。

```
server_address.sun_family = AF_UNIX;
strcpy(server_address.sun_path, "server_socket");
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

4) 创建一个连接队列，开始等待客户的到来。

```
listen(server_sockfd, 5);
while(1) {
    char ch;

    printf("server waiting\n");
```

5) 接受一个连接。

```
client_len = sizeof(client_address);
client_sockfd = accept(server_sockfd,
    (struct sockaddr *)&client_address, &client_len);
```

6) 对cilent_sockfd套接字上的客户进行读写。

```
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
    }
}
```

这个例子里的服务器程序一次只能向一个客户提供服务。它只从客户那里读取一个字符，给它加上一个“1”，再把它写回去；而在这个服务器完成处理工作之前，其他客户是无法建立起连接的。在更加复杂的系统里——即服务器需要应客户的请求完成相当多的处理工作时，这种一次只服务一个客户的做法是不能接受的。我们后面会看到几个允许建立多个连接的解决方案。

当我们运行这个程序的时候，服务器会创建出一个套接字并开始等待连接的到来。我们在后台启动它，让它独立运行起来，就可以在前台启动客户了。如下所示：

```
$ server1 &
[1] 1094
$ server waiting
```

服务器在开始等待连接的时候会打印出一条消息。在上面的例子里，服务器等待的是一个文件系统套接字，所以我们可以用普通的ls命令查看到它。记住：当你用完一个套接字的时候，就应该把它删除掉，即使是在程序因为一个信号而非正常终止的情况下也应该这么做。如下所示：

```
$ ls -lF server_socket
srwxr-xr-x   1 neil     users            0 Jan 14 08:28 server_socket=
```

访问权限前面的字母“s”和这一行末尾处的等号“=”表示该设备的类型是套接字。这个套接字就象是一个普通的文件那样被创建了出来，它的访问权限要受当前umask掩码的控制。如果使用ps命令，我们就能看到服务器正运行在后台。它目前处于休眠状态（STAT栏是S），不消耗任何CPU资源。如下所示：

```
$ ps -lx
 F   UID   PID  PPID PRI NI SIZE  RSS WCHAN      STAT TTY    TIME COMMAND
 0   501  1094   116   1  0   34  164 116976     S    pp0    0:00 server1
```

现在，当我们运行客户程序的时候，就比刚才的效果成功了。因为服务器套接字已经存在了，所以我们能够连接上它并与服务器进行通信。如下所示：

```
$ client1
server waiting
char from server = B
$
```

来自服务器的输出和来自客户的输出在我们的终端上混在了一起，但我们还是可以看出服务器确实从客户那里接受到一个字符，给它加上一个“1”后再返回了它。此后，服务器继续运行并且等待着下一个客户的到来。如果我们同时运行多个客户，就会看到它们是被依次服务的。如下所示：

```
$ client1 & client1 & client1 &
[2] 1106
[3] 1107
[4] 1108
server waiting
char from server = B
server waiting
char from server = B
server waiting
char from server = B
server waiting
[2]   Done                    client1
[3]-  Done                    client1
[4]+  Done                    client1
$
```

14.2.1 套接字属性

为了更好地理解这个例子里所使用的系统调用，我们需要先学习一些UNIX网络连接方面的知识。

套接字的特性是由三个属性确定的，即域（domain）、类型（type）和协议（protocol），另外还有一个被用做其名字的地址。地址的格式会根据域的不同而变化，也叫做协议族（protocol family）。每个协议族又可以用一个或者多个地址族来定义地址的格式。

1. 套接字的域

域定义的是套接字通信中使用的网络介质。最常用的套接字域是AF_INET，它对应着因特网网络模型，许多UNIX局域网用的都是它，当然，因特网自身用的也是它。其底层的协议是Internet Protocol（因特网协议，简称IP协议）。这个协议只有一个地址族，它使用一种独特的办法来指定网络中的计算机。这就是人们常说的IP地址。

虽然因特网上的名字指的总是联网的机器，但它们都将转换为底层的IP地址。比如说，192.1.168.1.99就是一个IP地址的例子。一切IP地址都是用四个数字表示的，每个数字都小于256，人们把它称做“四点数”。当一个客户通过套接字做跨网络连接的时候，就需要用到服务器计算机的IP地址。

服务器计算机上可能有好几个服务项目。客户可以通过一个IP端口来指定一台联网机器上的某项服务。在系统的内部，确定一个端口的办法是给它分配一个独一无二的16位整数；而在系统的外部，就要用IP地址和端口号的组合来确定它。套接字可以被看做是通信的“终点”，在开始通信之前必须被绑定到一个端口去。

服务器在特定的端口上等待连接的到来。那些常用服务所分配到的端口号在一切UNIX机器上都是一致的。它们通常（但不总是如此）是一些小于1024的数字。这些端口号包括打印机缓冲队列（515）、rlogin（513）、ftp（21）和httpd（80）等。其中最后一个就是World Wide Web（万维网，简称WWW网）用的服务器。在一般情况下，编号小于1024的端口都是为系统服务保

留的，提供相关服务的也只能是具有超级用户权限的进程。X/Open技术规范在netdb.h文件里定义了一个常数IPPORT_RESERVED，它代表着保留端口号的最大值。

因为标准服务都有标准的端口号，所以计算机就可以在不需要建立正确的端口号的前提下彼此接通。本地服务可以使用非标准的端口地址 我们第一个例子里的域就是UNIX文件系统的域AF_UNIX，即使一台没有联网的计算机上的套接字也可以使用这个域。这个域的底层协议就是文件的输入/输出，而它的地址就是绝对文件名。我们服务器套接字使用的地址是server_socket，我们在运行这个程序的时候可以看到它出现在当前子目录里。

其他可供使用的域还包括：按ISO标准中的有关协议组建的网络使用的AF_ISO域和施乐网络系统（Xerox Network System）使用的AF_NS等；但这些都不在我们这里的讨论范围之内。

2. 套接字类型

一个套接字域可以有多种不同的通信方式，而每种通信方式又有其不同的特性。但AF_UNIX域里的套接字不存在这样的问题，它们提供了一条可靠的双向通信路径。但如果一个域能够用来构建网络，我们就需要对它的底层网络的特性多加注意。

因特网协议提供了两种截然不同的服务：流（stream）和数据图（datagram）。

流式服务（有点类似于标准的输入/输出流）提供的是一个有序的可靠的双向字节流。也就是说，被发送的数据不会被丢失、复制或者先后次序被弄乱；错误会被自动纠正而不是报告给用户。大块消息将被拆分、传输、再重新组合。这很象是一个接收了大量的数据但需要以较小的数据块来把它写到底层磁盘上去的文件流。流式套接字的行为是可以预见的。

流式套接字被定义为SOCK_STREAM类型，它们是在AF_INET域里通过TCP/IP连接实现的。它们也是AF_UNIX域里最常见的套接字类型。我们在这一章里的学习将主要集中在SOCK_STREAM套接字方面，因为它们是人们在编写网络软件时最常用的。

> TCP/IP指的是“Transmission Control Protocol / Internet Protocol”（传输控制协议/因特网协议）。IP协议是数据包使用的底层协议，它提供了一台计算机穿过网络到达另一台计算机的路由安排。TCP协议提供了顺序安排、流控制和再传输功能，保证大数据传输能够到达目的地。

与此形成鲜明对照的是被定义为SOCK_DGRAM类型的数据图套接字，它不需要建立和维持一个连接。对数据图的发送长度也有限制。它在网络中传输时不会被拆分，一个数据图就是一条网络消息，它可能会被丢失、复制或者不按先后次序到达目的地。

数据图套接字是在AF_INET域里通过UDP/IP连接实现的，它提供的是一种无序的非可靠服务。但因为不需要维持网络连接，所以从资源角度看它们的开销相对要小一些。而且因为不需要花费时间去建立相关的连接，所以它们也更快。UDP代表着“User Datagram Protocol”（用户数据图协议）。

数据图适用于信息服务中的“单击”查询，主要用来提供日常状态信息或者用来完成低优先级的日志记录操作。它们的优点是服务器的“死亡”不要求客户重新启动。因为基于数据图的服务器通常不保存与连接有关的信息，所以它们可以在不打扰其客户的前提下停止并重新启动。

3. 套接字协议

只要底层的传输机制允许不止一个协议来提供要求的套接字类型，我们就可以为套接字挑选一个特定的协议。在这一章里，我们将把注意力集中在UNIX的网络套接字和文件系统套接字上，你可以直接使用其缺省值，不需要再挑选一个协议。

14.2.2 创建一个套接字

socket系统调用的作用是创建一个套接字并返回一个能够用来访问该套接字的描述符。它的定义情况如下所示：

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

创建出来的套接字是一条通信线路的一个端点。domain参数负责指定地址族，type参数负责指定与这个套接字一起使用的通信类型，而protocol参数负责指定所使用的协议。

domain参数可以使用的域如表14-1所示：

表 14-1

AF_UNIX	UNIX内部（文件系统套接字）
AF_INET	ARPA因特网协议（UNIX网络套接字）
AF_ISO	ISO标准协议
AF_NS	施乐网络系统协议
AF_IPX	Novell IPX协议
AF_APPLETALK	Appletalk DDS

最常用的套接字域是AF_UNIX和AF_INET，前者用于通过UNIX文件系统实现的本地套接字，后者用于UNIX的网络套接字。AF_INET套接字可以用在穿过包括因特网在内的各种TCP/IP网络而进行通信的程序里。微软Windows使用的WinSock接口也提供了对这个套接字域的访问功能。

套接字参数type指定了与新套接字对应的通信特性。它的可取值包括SOCK_STREAM和SOCK_DGRAM。

SOCK_STREAM是一个有序的、可靠的、基于连接的双向字节流。对一个AF_INET域的套接字来说，如果在两个流式套接字的两端之间建立的是一个TCP连接，连接时缺省提供的就是这种类型的套接字。数据可以沿着套接字连接双向传递。TCP协议里所提供的工具能够对长消息进行拆分和重新组装；并且，如果有数据丢失在网络里，它还能重新发送它。

SOCK_DGRAM是一个数据图服务。我们可以用它来发送最大长度是一个固定值（通常不太大）的消息，但消息是否会被送达或者消息的先后次序是否会在网络中被重新安排并没有保证。对AF_INET域的套接字来说，这种类型的通信是由UDP数据图提供的。

通信所用的协议通常是由套接字的类型和套接字的域来决定的，一般不再有可供挑选的余地。如果还能挑选，就需要用到protocol参数了。“0”选择缺省的协议，我们将在本章所有的例子里都这样做。

socket系统调用返回的是一个描述符，它在许多方面都类似于一个底层的文件描述符。当这个套接字和通信线路另一端的套接字连接好以后，我们就可以用read和write系统调用加上这个描述符在这个套接字上收发数据了。结束一个套接字连接要使用close系统调用。

14.2.3 套接字地址

每个套接字域都有它自己的地址格式。对一个AF_UNIX套接字来说，它的地址是由一个定义在sys/un.h头文件里的sockaddr_un结构描述的。

```
struct sockaddr_un {
    sa_family_t    sun_family;    /* AF_UNIX */
    char           sun_path[];    /* pathname */
};
```

因为不同类型的地址都需要传递到对套接字进行处理的系统调用里去，所以定义各种地址格式时使用的结构也都很相似，每个结构的开始都是一个定义地址类型（即套接字域）的数据项（即上面的sun_family）。在AF_UNIX域里，套接字的地址是用sockaddr_un结构里sun_path数据项中的文件名指定的。

sun_family_t是由X/Open技术规范定义的，在现时期的Linux系统上，它被声明为一个short短整数。此外，sun_path给出的路径名的长度也是有限制的（Linux的规定是108个字符，其他系统可能会使用灵活性好一点的常数UNIX_MAX_PATH）。因为地址结构在长度方面是不固定的，所以许多套接字调用都要用到或输出一个用来复制特定地址结构的长度值。

AF_INET域里的套接字地址是由一个定义在netinet/in.h头文件里的sockaddr_in结构确定的，它至少应该包含如下所示的几个成员：

```
struct sockaddr_in {
    short int              sin_family;  /* AF_INET */
    unsigned short int     sin_port;    /* Port number */
    struct in_addr         sin_addr;    /* Internet address */
};
```

IP地址结构in_addr被定义为：

```
struct in_addr {
    unsigned long int      s_addr;
};
```

IP地址中的四个字节组成一个32位的二进制数值。一个AF_INET套接字完全可以由它的域、IP地址和端口号确定下来。从应用程序的角度看，各种套接字的行为就象是文件描述符，用一个独一无二的整数就可以把它们表示出来。

14.2.4 给套接字起名字

要想让socket调用创建的一个套接字能够被其他进程使用，服务器程序就必须给该套接字起个名字。因此，AF_UNIX套接字就会关联到一个文件系统的路径名上去，就象我们在server1例子里看到的那样。AF_INET套接字将关联到一个IP端口号上去。

```
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address, size_t address_len);
```

bind系统调用的作用是把参数address中给出的地址赋值给与文件描述符socket相关联的未命名套接字。地址结构的长度是通过address_len参数传递的。

地址的长度和类型取决于地址族。bind调用需要用一个与之对应的地址结构指针指向真正的地址类型（即上面定义里的“struct sockaddr *”）。

bind在调用成功时将返回“0”；如果失败，就返回“-1”并把errno设置为表14-2中的某个值。

表　14-2

EBADF	该文件描述符无效
ENOTSOCK	该文件描述符代表的不是一个套接字
EINVAL	该文件描述符代表的是一个已经命名了的套接字
EADDRNOTAVAIL	该地址不可用
EADDRINUSE	该地址已经绑定有一个套接字了

AF_UNIX套接字还多出两个错误代码值见表14-3。

表　14-3

EACCESS	权限不足，不能创建文件系统中使用的名字
ENOTDIR,ENAMETOOLONG	选用的文件名不好

14.2.5　创建套接字队列

为了能够在套接字上接受接入的连接，服务器程序必须创建一个队列来保存到达的请求。这个工作是由listen系统调用完成的。

```
#include <sys/socket.h>
int listen(int socket, int backlog);
```

一台UNIX系统可能会对队列里能够容纳的排队连接的最大个数有限制。在这个最大值的范围内，listen将把队列长度设置为backlog个连接。在套接字上排队的接入连接个数最多不能超过这个数字，再往后的连接将被拒绝，客户的连接请求将会失败。这是listen提供的一个机制，在服务器程序紧张地处理着上一个客户的时候，后来的连接将被放到队列里排队等候。backlog常用的值是5。

listen函数在成功时会返回“0”，失败时返回“-1”。它的错误情况包括EBADF、EINVAL和ENOTSOCK，含义同bind系统调用的有关代码。

14.2.6　接受连接

服务器程序创建好命名套接字之后，就可以通过accept系统调用等待客户来建立对该套接字的连接了。

```
#include <sys/socket.h>
int accept(int socket, struct sockaddr *address, size_t *address_len);
```

accept系统调用会等到有客户程序试图连接到由socket参数指定的套接字时才返回。该客户

就是套接字队列里排在第一位的连接。accept函数将创建出一个新的套接字来与该客户进行通信，返回的是与之对应的文件描述符。新套接字的类型与服务器监听套接字的类型是一样的。

套接字必须事先用一个bind调用进行过命名，并且还要有一个由listen调用分配的连接队列。调用者客户的地址将被放在address指向的sockaddr结构里。如果我们不关心客户的地址，可以在这里使用一个空指针。

参数address_len给出了客户结构的长度。如果客户地址的长度超过了这个值，就会被截短。在调用accept之前，必须把address_len设置为预期的地址长度。当这个调用返回时，address_len将被设置为调用者客户的地址结构的实际长度。

如果套接字队列里没有排队等候的连接，accept将阻塞（程序就不会继续执行了）到有客户建立连接为止。这个行为可以用O_NONBLOCK标志改变，方法是对这个套接字文件描述符调用fcntl函数，如下所示：

```
int flags = fcntl(socket, F_GETFL, 0);
fcntl(socket, F_SETFL, O_NONBLOCK|flags);
```

如果有排队等候的客户连接，accept函数将返回一个新的套接字文件描述符，否则它将返回“-1”。其错误原因除类似于bind调用和listen调用中的情况之外，还多出有一个EWOULDBLOCK，如果前面指定了O_NONBLOCK标志，但队列里没有排队的连接，就会出现这个错误。如果进程阻塞在accetp调用里的时候执行被中断了，就会出现EINTR错误。

14.2.7 请求连接

当客户想要连接到服务器的时候，它会尝试在一个未命名套接字和服务器的监听套接字之间建立一个连接。它们用调用connect函数来完成这一工作。

```
#include <sys/socket.h>
int connect(int socket, const struct sockaddr *address, size_t address_len);
```

参数socket指定的套接字将连接到参数address指定的服务器套接字上去，服务器套接字的长度由参数address_len指定。套接字必须是通过socket调用获得的一个有效的文件描述符。

如果操作成功，connect将返回“0”；如果失败，就返回“-1”。这个调用可能出现的错误情况见表14-4：

表 14-4

EBADF	传递到socket里的文件件描述符无效
EALREADY	该套接字上已经有了一个正在使用的连接
ETIMEDOUT	连接操作超时
ECONNREFUSED	连接请求被服务器拒绝

如果连接不能立刻建立起来，connect会阻塞一段不确定的倒计时时间，这段倒计时时间结束后，这次连接就会流产，connect调用失败。但如果connect调用是被一个信号中断的，而这个

信号又得到了处理，connect还是会失败（errno将被设置为EINTR），可这次连接尝试却不会流产，它会以异步方式继续尝试。

类似于accept，connect的阻塞特性可以用置位该文件描述符的O_NONBLOCK标志的办法来改变。在这种情况下，如果连接不能立刻建立起来，connect会失败并把errno设置为EINPROGRESS，而连接将以异步方式继续尝试。

异步连接的处理是比较困难的，而我们可以在套接字文件描述符上用一个select调用来表明该套接字已经处于写就绪状态。select将本章后面的内容里介绍。

14.2.8 关闭一个套接字

调用close函数就可以结束服务器和客户上的套接字连接，就象对底层文件描述符进行操作一样。要想关闭套接字，就必须把服务器和客户两头都关掉才行。对服务器来说，应该在read返回“0”的时候做这件事，但如果套接字是一个面向连接的类型并且设置了SOCK_LINGER选项，close调用会在该套接字尚有未传输数据时阻塞。我们将在这一章后面的内容里学习如何设置套接字选项。

14.2.9 套接字通信

学习了与套接字有关的基本函数调用之后，我们来看几个程序示例。

我们将尽量使用网络套接字而不是使用文件系统套接字。文件系统套接字的缺点是除非程序员使用的是一个绝对路径名，否则套接字将创建在服务器程序的当前子目录里；因此，为了让它更具通用性，就需要把它创建在一个服务器及其客户都认可的能被全局性访问的子目录（比如/tmp子目录）里。而对网络套接字来说，我们只需选择一个未被使用的端口号就可以了。

我们的例子将选择使用端口号9734。这个端口号是在避开了标准服务的前提下随便挑选的。我们不能使用小于1024的端口号，因为它们都是为系统保留的。端口号和通过它们提供的服务通常都列在系统文件/etc/services里。在编写基于套接字的程序时永远要选择没有在这个配置文件里列出的端口号。

我们将在一个网络上运行我们的服务器和客户，但网络套接字的用途并不仅限于网络，只要是带因特网连接（那怕是一个调制解调器拨号连接）的机器就能使用网络套接字与其他机器进行通信。你甚至可以在一台UNIX单机上使用一个基于网络的程序，这是因为UNIX计算机通常会被配置为一个只包含着它自身的回馈（loopback）网络。出于演示的目的，我们将使用这个回馈网络，它不存在与外部网络有关的问题，对我们网络程序的调试工作是很有帮助的。

回馈网络里只包含着一台计算机，人们习惯地称它为localhost，它有一个标准的IP地址：127.0.0.1。这就是所谓的本地主机。大家可以在网络主机文件/etc/hosts列出的共享网络上各主机的名字和服务清单里找到它的地址。

与计算机进行通信的每个网络都有一个相关的硬件接口设备。网络上的每一台计算机都可以有好几个不同的网络名，当然也就有好几个不同的IP地址。就拿我的计算机tilde来说吧，它有二个网络接口，因此也就有三个地址。这些信息都保存在/etc/hosts文件里：

```
127.0.0.1       localhost               # Loopback
192.168.1.1     tilde.localnet          # Local, private Ethernet
158 152.X.X     tilde.demon.co.uk       # Modem dial-up
```

表中的第一个就是那个简单的回馈网络，第二个是通过一块以太网卡来访问的局域网，第三个是到一个因特网接入服务提供商的调制解调器链接。你编写的基于套接字的网络程序不做任何修改就可以通过这些个接口与相应的服务器进行通信。

下面是经过修改了的客户程序client2.c，它通过回馈网络与一个网络套接字进行连接。这个程序里有一个硬件依赖方面的缺陷，但我们一会儿再讨论它。

动手试试：网络客户

1) 头文件和变量的初始化。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
    int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    char ch = 'A';
```

2) 为客户创建一个套接字。

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

3) 给套接字起个名字，注意要与服务器保持一致。

```
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("127.0.0.1");
address.sin_port = 9734;
len = sizeof(address);
```

这个程序的其余部分与这一章前面出现的client1.c完全一样。当我们运行这一版本的时候，它会连接失败，因为在这台计算机的9734号端口上现在还没有服务器运行着。

```
$ client2
oops: client2: Connection refused
$
```

操作注释：

客户程序用头文件netinet/in.h里定义的sockaddr_in结构定义了一个AF_INET地址。它试图与IP地址为127.0.0.1的主机上的服务器建立连接。它使用inet_addr函数把IP地址的字符串形式转换为符合套接字编址要求的格式。inet的使用手册页里有对其他地址转换函数的详细说明。

我们还需要修改服务器程序，让它在我们挑选的端口号上等待连接的到来。下面是经过了修改的服务器server2.c

动手试试：网络服务器

1) 必要的头文件和变量的初始化。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;
```

2) 为服务器创建一个未命名套接字。

```
server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

3) 给套接字起名字。

```
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
server_address.sin_port = 9734;
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

从此往下，程序清单将与server1.c的完全一样。

操作注释：

服务器程序创建了一个AF_INET域的套接字，并且安排在它上面接受连接。这个套接字被绑定到我们选定的端口号上。程序中指定的地址决定了连接将建立在哪一台计算机上。象客户程序里一样，我们给出的是回馈网络的地址，把通信工作限制在本地主机上。

如果我们想让服务器与远程客户进行通信，就必须给出我们愿意让它们连接的一组IP地址。我们用特殊值INADDR_ANY表示我们将接受来自我们计算机任何网络接口的连接。如果愿意，也可以把不同的网络接口分开来写，比如把内部的局域网和外部的广域网分别写出来。INADDR_ANY是一个32位的整数值，它可以用在地址结构里的sin_addr.s_addr数据域里。但我们需要先来解决一个下面这样的问题。

14.2.10 主机字节顺序和网络字节顺序

我的计算机是一台使用英特尔芯片的Linux机器，当我在它上面运行新编写的服务器程序和客户程序时，可以用netstat命令查看网络连接的状况。这个命令在大多数配置了网络功能的UNIX系统上都能找到。它给出的是尚未关闭的客户/服务器之间的连接情况。连接将在一小段倒计时时间后关闭。（再说一次，它具体的输出内容会随Linux版本的不同而发生变化。）

```
$ server2 &
[4] 1225
```

```
$ server waiting
client2
server waiting
char from server = B
$ netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address  (State)       User
tcp        1      0 localhost:1574  localhost:1174   TIME_WAIT     root
```

> 在你试运行本章中的其他程序示例之前，一定要先终止正在运行的服务器示例程序，因为它们会争夺来自客户的连接，而你将看到混乱的结果。你可以用下面的命令杀掉它们：
>
> ```
> kill -1 <processid.>
> ```

大家应该能够看到分配给服务器和客户之间这条连接的端口号。“Local Address”栏给出的是服务器，而“Foreign Address”给出的是远程客户（即使是在同一台机器上，它也是通过一个网络而连接起来的）。为保证所有的套接字都是泾渭分明的，那些客户端口一般都异于服务器的监听套接字，并且在这台计算机上都是独一无二的。

可是，显示出来的本地地址（即服务器套接字）是1574，而我们选择的端口是9734呀，它们为什么会不一样呢？答案是这样的：通过套接字接口传递的端口号和地址都是二进制数字；而不同计算机上的整数所使用的字节顺序是不一样的。比如说，英特尔处理器在把32位整数保存到内存里去的时候所使用的字节顺序是1-2-3-4，1代表最大字节；而摩托罗拉处理器是以字节顺序4-3-2-1的方式来保存整数的。如果保存整数用的字节是以简单的字节对字节方式拷贝的话，在两台不同的计算机上得到的整数值就将是不一致的。

为了使不同型号的计算机在通过网络传递的多字节整数上能够取得一致的值，就必须为字节定义一个网络顺序。客户程序和服务器程序必须在传输之前把它们内部的整数表示方式转换为字节的网络顺序。这一工作是由netinet/in.h头文件里定义的函数完成的，如下所示：

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

这些函数的作用是把16位和32位整数的本地主机格式转换为标准的网络顺序。函数的名称是与之对应的转换操作的简写形式。比如说“host to network, long”（htonl，主机到网络，长整数）和“host to network, short”（htons，主机到网络，短整数）等。如果计算机本身的字节顺序与网络顺序相同的话，它们就代表空操作。

为了保证16位端口号有正确的字节顺序，我们的服务器和客户需要用这些函数来处理这个端口地址。新服务器程序server3.c里的改动是：

```
server_address.sin_addr.s_addr = htonl(INADDR_ANY );
server_address.sin_port = htons(9734 );
```

我们不需要对函数调用“inet_addr("127.0.0.1")”进行转换，因为inet_addr已经被定义为产生一个网络字节顺序的结果了。新客户程序client3.c里的改动是：

```
address.sin_port = htons(9734 );
```

服务器还需要对为允许任一IP地址连接而使用的INADDR_ANY进行改动。

现在，当我们运行server3和client3的时候，就会看到本地连接使用的是正确的端口了。

```
$ netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address   Foreign Address   (State)       User
tcp        1      0 localhost:9734  localhost:1175    TIME_WAIT     root
```

如果读者使用的计算机上的本机字节顺序与网络字节相同，就不会看到任何差异。为了让不同体系结构的计算机上的客户和服务器能够正确地实现操作，最好是把刚才介绍的转换函数加到你编写的每一个网络程序里去。

14.3 网络信息

到目前为止，我们的客户和服务器一直是把地址和端口号编译在它们自己的内部的。如果我们想让这些服务器和客户程序更具通用性，就应该使用网络信息函数来获取相应的地址和端口号。

如果有足够的权限，我们可以把我们的服务器添加到/etc/services网络里列出的已知服务清单里去，这个文件可以给端口号分配一个名字，使客户可以使用符号化的服务项目名称而不是干巴巴的数字。

类似地，如果给定一个计算机的名字，我们就可以通过调用主机数据库函数对其进行解析的方法确定它的IP地址。这些函数会通过查询网络配置文件来完成这些工作，比如查询/etc/hosts文件，或者查询网络信息服务等。比较知名的网络信息服务有NIS（Network Information Service，网络信息服务；以前叫做Yellow Pages，黄页服务）和DNS（Domain Name Service，域名解析服务）等。

主机数据库函数是在接口头文件netdb.h里定义的，如下所示：

```
#include <netdb.h>

struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
struct hostent *gethostbyname(const char *name);
```

这些函数返回的结构至少会包含以下几个成员：

```
struct hostent {
    char *h_name;           /* name of the host */
    char **h_aliases;       /* list of aliases (nicknames) */
    int h_addrtype;         /* address type */
    int h_length;           /* length in bytes of the address */
    char **h_addr_list      /* list of address (network order) */
};
```

如果没有与我们查询的主机或地址相关的数据库数据项，这些信息函数会返回一个空指针。

类似地，与服务及其关联端口号有关的信息可以通过一些服务信息函数查到，下面是它们的定义情况：

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

proto参数指定了用来连接到该项服务的协议，SOCK_STREAM类型的TCP连接对应的是“tcp”，而SOCK_DGRAM类型的UDP数据图对应的是“udp”。

servent结构至少应该包含以下几个成员：

```
struct servent {
    char *s_name;          /* name of the service */
    char **s_aliases;      /* list of aliases (alternative names) */
    int s_port;            /* The IP port number */
    char *s_proto;         /* The service type, usually "tcp" or "udp" */
};
```

如果想收集某台计算机的主机数据库信息，我们可以调用gethostbyname函数并且把结果打印出来。注意要把地址表映射到正确的地址类型上去，而且要把它们从网络字节顺序转换为一个可供打印的字符串——这个转换可以用inet_ntoa函数来完成，它的定义情况如下所示：

```
# include <arpa/inet.h>

char *inet_ntoa(struct in_addr in)
```

这个函数的作用是把一个因特网主机地址转换为一个四点数格式的可打印字符串。它在失败时将返回“-1”，但X/Open技术规范并没有定义任何错误。其他可用的新函数还有gethostname，下面是它的定义：

```
# include <unistd.h>

int gethostname(char *name, int namelength );
```

这个函数的作用是把当前主机的名字写到name指向的字符串里去。这个主机名是以空字节“0”结尾的。参数namelength限定了字符串name的长度，如果返回的主机名太长，就会被截短。gethostname在成功时返回“0”，失败时返回“-1”，但POSIX里还是没有定义任何错误。

下面这个getname.c程序的功能是查询一台主机计算机的有关信息。

动手试试：网络信息

1) 和往常一样，必要的头文件和变量定义。

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    char *host, **names, **addrs;
    struct hostent *hostinfo;
```

2) 把host变量设置为随getname调用提供的参数，如果缺省，就设置为用户正在使用的这台机器。

```
if(argc == 1) {
    char myname[256];
    gethostname(myname, 255);
    host = myname;
}
else
    host = argv[1];
```

3) 调用gethostbyname，如果信息没找到就报告一条错误。

```
hostinfo = gethostbyname(host);
if(!hostinfo) {
    fprintf(stderr, "cannot get info for host: %s\n", host);
    exit(1);
}
```

4) 显示主机名和它可能有的一切假名。

```
printf("results for host %s:\n", host);
printf("Name: %s\n", hostinfo -> h_name);
printf("Aliases:");
names = hostinfo -> h_aliases;
while(*names) {
    printf(" %s", *names);
    names++;
}
printf("\n");
```

5) 如果要查询的主机不是一个IP主机，就报告并退出。

```
if(hostinfo -> h_addrtype != AF_INET) {
    fprintf(stderr, "not an IP host!\n");
    exit(1);
}
```

6) 否则，显示它的IP地址。

```
    addrs = hostinfo -> h_addr_list;
    while(*addrs) {
        printf(" %s", inet_ntoa(*(struct in_addr *)*addrs));
        addrs++;
    }
    printf("\n");
    exit(0);
}
```

另外一种办法是使用gethostbyaddr，它可以查出给定IP地址处是哪一台主机。你可以把它用在一个服务器里去查看客户是从哪里来调用的。

操作注释：

getname程序调用gethostbyname从主机数据库里提取出主机的资料。它将给出主机名、它的假名（那台计算机的其他名字）和该主机在它的网络接口上使用的IP地址。当我在自己的机器上运行这个程序的时候，给定主机名tilde，程序给出了以太网和调制解调器两个网络接口的资料。如下所示：

```
$ getname tilde
results for host tilde:
Name: tilde.demon.co.uk
Aliases: tilde
192.9.200.4 158.152.38.110
```

当我使用主机名localhost的时候，程序只给出了回馈网络的资料。如下所示：

```
$ getname localhost
results for host localhost:
Name: localhost
Aliases:
127.0.0.1
```

我们现在来改进我们的客户程序，使它能够连接到任何有名字的主机。这一次我们不再连

接到我们示例用的服务器，而是连接到一项标准服务上去，这样可以演示端口号的提取操作。

大多数UNIX系统都有一项名为daytime的标准服务，即提供它们的系统日期和时间。客户可以连接到这项服务上去看看该服务器的当前日期和时间是什么。下面就是完成这一工作的客户程序getdate.c。

动手试试：连接到一个标准服务

1) 必要的头文件和变量定义。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    char *host;
    int sockfd;
    int len, result;
    struct sockaddr_in address;
    struct hostent *hostinfo;
    struct servent *servinfo;
    char buffer[128];

    if(argc == 1)
        host = "localhost";
    else
        host = argv[1];
```

2) 查找主机的地址，如果找不到就报告一条错误。

```
hostinfo = gethostbyname(host);
if(!hostinfo) {
    fprintf(stderr, "no host: %s\n", host);
    exit(1);
}
```

3) 检查主机上有没有daytime服务。

```
servinfo = getservbyname("daytime", "tcp");
if(!servinfo) {
    fprintf(stderr,"no daytime service\n");
    exit(1);
}
printf("daytime port is %d\n", ntohs(servinfo -> s_port));
```

4) 创建一个套接字。

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

5) 构造connect函数要用到的地址等……。

```
address.sin_family = AF_INET;
address.sin_port = servinfo -> s_port;
address.sin_addr = *(struct in_addr *)*hostinfo -> h_addr_list;
len = sizeof(address);
```

6) 建立连接并取得有关信息。

```
result = connect(sockfd, (struct sockaddr *)&address, len);
if(result == -1) {
```

```
        perror("oops: getdate");
        exit(1);
    }

    result = read(sockfd, buffer, sizeof(buffer));
    buffer[result] = '\0';
    printf("read %d bytes: %s", result, buffer);

    close(sockfd);
    exit(0);
}
```

我们可以用getdate获取任一有名字的主机上的日期和时间。如下所示：

```
$ getdate tilde
daytime port is 13
read 26 bytes: Sun Aug  1 11:29:53 1999
$
```

如果我们看到如下所示的一条错误信息：

```
oops: getdate: Connection refused
```

就说明你正在连接的计算机可能没有激活daytime服务。这是现时期Linux系统的缺省行为。我们将在下一小节里学习如何激活这项服务和其他服务项目。

操作注释：

当我们运行这个程序的时候，可以指定一个主机去建立连接。daytime服务的端口号是通过网络数据库函数getserverbyname确定的，这个函数返回的是关于网络服务方面的资料，它们和主机资料差不多。getdate程序会先去尝试连接指定主机替换地址表里列出来的地址。如果成功，它就读取daytime服务返回的信息——这是一个表示UNIX时间和日期的字符串。

14.3.1 因特网守护进程

提供多项网络服务的UNIX系统通常是以超级服务器的方式来这样做的。这个程序（因特网守护进程，即inetd）同时监听着许多端口地址上的连接。当有客户连接到某项服务时，inetd程序就会运行相应的服务器。这使服务器们不必一直运行着；它们可以在必要时由inetd启动执行。下面是inetd配置文件/etc/inetd.conf中的一个片段，这个文件的作用是决定需要运行哪些个服务器。如下所示：

```
#
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
#
# Echo, discard, daytime, and chargen are used primarily for testing.
#
daytime     stream     tcp     nowait     root     internal
daytime     dgram      udp     wait       root     internal
#
# These are standard services.
#
ftp      stream     tcp     nowait     root     /usr/sbin/tcpd     /usr/sbin/wu.ftpd
telnet   stream     tcp     nowait     root     /usr/sbin/tcpd     /usr/sbin/in.telnetd
#
# End of inetd.conf.
```

我们的getdate程序连接到的daytime服务实际上是由inetd本身负责处理的（被标记为internal，

内部），SOCK_STREAM（tcp）套接字和SOCK_DGRAM（udp）套接字都能使用这项服务。

文件传输服务ftp只能通过SOCK_STREAM套接字提供，并且是由一个外部程序提供的，例子里使用的是wu.ftpd，当有客户连接到ftp端口时，inetd就会启动它。

通过编辑/etc/inetd.conf文件（语句最前面的“#”符号表示这是一个注释行）再重新启动inetd进程的办法，我们就可以改变通过inetd提供的服务。这可以通过用kill命令向它发送一个挂起信号来实现。为了使这个操作更容易进行，有的系统会配置成让inetd把它自己的进程ID写到一个文件里（在Red Hat 6.0上，它是/var/run/inetd.pid文件）。另外一个办法是使用killall命令，如下所示：

```
#  killall -HUP inetd
```

14.3.2 套接字选项

套接字连接有许多可以用来控制其行为的选项——这些选项数量很多，这里根本没有足够的篇幅来对它们做详细的介绍。应用这些选项时要使用setsockopt函数。

```
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name,
        const void *option_value, size_t option_len);
```

你可以在协议树结构里的各种级别上对这些选项进行设置。如果想要在套接字级别上设置选项，就必须把level参数设置为SOL_SOCKET。如果想要在底层协议级别（比如TCP、UDP等）上设置选项，就必须把level参数设置为该协议的编号（这些编号从netinet/in.h头文件或者通过getprotobyname函数获得）。

option_name参数指定准备设置的选项，option_value是一个长度为option_len个字节的任意常数值参数，底层协议的处理程序需要传递来这么一个参数。

在/sys/socket.h文件里定义的套接字级别选项见表14-5：

表 14-5

SO_DEBUG	打开调试信息
SO_KEEPALIVE	为周期性传输保持连接—直接通
SO_LINGER	在关闭之前先完成传输工作

SO_DEBUG和SO_KEEPALIVE级别要用一个整数的option_value值来设置该选项的开（1）关（0）状态。SO_LINGER需要使用一个在/sys/socket.h文件里定义的linger结构来定义该选项的状态和关闭之前的延长期。

如果操作成功，setsockopt返回“0”，否则返回“-1”。它的使用手册页里介绍了更多的选项和错误。

14.4 多客户

本章到目前为止一直介绍的是如何利用套接字来实现本地或跨网络的客户/服务器系统。在建立起套接字连接之后，它的作用几乎完全类似于一个底层open调用所返回的文件描述符，并

其在行为上与双向管道有很多相似之处。

我们现在来考虑一下多个客户同时连接到一个服务器的情况。我们已经知道服务器程序在接受了一个新的连接时会创建出一个新的套接字，而原先的那个监听套接字会被空出来监听以后的连接。如果服务器不能立刻接受后来的连接，它们就会被放到一个队列里去排队等候。

既然原先的套接字被空了出来，套接字的行为又象是一个文件描述符，我们就可以利用这些事实找出一个同时服务多个客户的办法来。如果服务器调用fork为自己创建了一个第二份拷贝，已经打开的套接字就可以被那个新创建出来的子进程所继承。我们可以让它与当前连接着的客户进行通信，让主服务器继续接受后来的客户连接。这些改动对我们的服务器程序来说是很容易做到的，下面给出的就是。

既然我们正在创建子进程，但又不等待它们的完成和结束。我们就必须安排服务器不理睬SIG_CHLD信号以避免出现僵进程。

动手试试：一个可以服务多个客户的服务器

1) 这个server4.c程序的开始部分与我们前面的服务器一脉相承，只是引人注目地增加了一条包括signal.h头文件的include语句。变量的初始化和套接字的创建及命名过程都与以前一样。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <signal.h>
#include <unistd.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;

    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

2) 创建一个连接队列，屏蔽子进程的退出细节并等待客户的到来。

```
listen(server_sockfd, 5);

signal(SIGCHLD, SIG_IGN);

while(1) {
    char ch;

    printf("server waiting\n");
```

3) 接受连接。

```
client_len = sizeof(client_address);
client_sockfd = accept(server_sockfd,
    (struct sockaddr *)&client_address, &client_len);
```

4) 通过fork调用为这个客户创建一个子进程，然后测试我们是在父进程里还是在子进程里。

```
if(fork() == 0) {
```

5) 如果我们是在子进程里，就可以对client_sockfd上的客户进行读写。那个五秒钟的休眠时间纯粹出于演示的目的。

```
        read(client_sockfd, &ch, 1);
        sleep(5);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
        exit(0);
    }
```

6) 否则，我们一定是在父进程里，而在此需要我们为这个客户做的事情就是关闭它。

```
        else {
            close(client_sockfd);
        }
    }
}
```

我们在客户请求的处理过程中增加的五秒钟休眠时间是为了模仿服务器的计算工作或数据库的查询工作。如果我们在前面的服务器里这样做，client3的每次运行就都将花费五秒钟的时间。而我们的新服务器可以同时处理多个client3程序，所花费的总时间将只有五秒钟多一点。如下所示：

```
$ server4&
[7] 1571
$server waiting
client3 & client3 & client3 & ps -ax
[8] 1572
[9] 1573
[10] 1574
server waiting
server waiting
server waiting
PID TTY STAT  TIME COMMAND
 1557 pp0 S     0:00 server4
 1572 pp0 S     0:00 client3
 1573 pp0 S     0:00 client3
 1574 pp0 S     0:00 client3
 1575 pp0 R     0:00 ps -ax
 1576 pp0 S     0:00 server4
 1577 pp0 S     0:00 server4
 1578 pp0 S     0:00 server4
$ char from server = B
char from server = B
char from server = B
ps -ax
PID TTY STAT  TIME COMMAND
 1557 pp0 S     0:00 server4
 1580 pp0 R     0:00 ps -ax
[8]   Done                    client3
[9]-  Done                    client3
[10]+  Done                    client3
$
```

操作注释：

服务器程序现在会新创建一个子进程来处理每一个客户，所以我们会看到好几个服务器等待消息，而主程序将继续等待新的连接。ps命令的输出（我们进行了删节）显示PID是1557的server4主进程正等待着新客户的到来，而同时有三个client3进程正在被该服务器的三个子进程服

务着。在经过了一个五秒钟的暂停之后，所有客户都得到了它们的结果并结束了运行。子服务器进程也都退出了，只留下了主服务器还等在那里。

服务器程序利用fork函数来处理多个客户。在一个数据库软件里，这可能不是最佳的解决方案，因为服务器程序可能会相当的大，在数据库访问方面还存在着需要协调多个服务器进程的问题。事实上，我们真正需要的解决之道是怎样才能让一个服务器在不阻塞、不等待客户请求到达的前提下完成对多个客户的处理。这个问题的解决方案涉及到如何同时处理多个打开着的文件描述符，并且不仅仅局限于套接字应用程序。请继续学习select系统调用。

14.5 select系统调用

当我们在编写UNIX软件的时候，经常会出现需要检查好几个输入才能确定下一步行动的情况。就拿一个终端仿真器这样的通信程序来说吧，它至少需要能够同时读取键盘和串行口这两种设备上的状态。如果是在一个单用户系统里，运行一个“繁忙等待”循环来进行这种检查还是可以接受的，它会不停地扫描输入设备上是否有数据，数据一出现就会被读取。但这种做法用CPU时间来衡量开销就太大了。

select系统调用允许程序同时在多个底层文件描述符上等待输入的到达(或等待输出的结束)。这就意味着终端仿真器程序可以阻塞直到有事情可做为止。类似地，服务器也可以通过同时在多个打开着的套接字上等待请求到来的办法来应付多个客户。

select函数是在数据结构fd_set上完成操作的，它们是由open调用返回的文件描述符所构成的集合。为了对这些集合进行处理，人们定义了下面这些宏：

```
#include <sys/types.h>
#include <sys/time.h>

void FD_ZERO(fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
```

这些宏定义的作用从它们各自的名称上就可以看出来：FD_ZERO会把一个fd_set初始化为一个空白的集合；FD_SET和FD_CLR对与参数fd指定的文件描述符相对应的那个集合中的元素进行置位和清除；FD_ISSET是一个布尔类型的宏，如果参数fd对应的文件描述符是参数fdset指向的fd_set中的一个元素，它就返回一个非零值。一个fd_set结构里所能容纳的文件描述符的最大个数是由常数FD_SETSIZE限定的。

select函数还可以使用一个倒计时值以防止无限期阻塞现象的发生。这个倒计时时间值是用一个“struct timeval”结构给出的。这个结构的定义在sys/time.h文件里，由以下几个成员组成：

```
struct timeval {
  time_t     tv_sec;     /* seconds */
  long       tv_usec;    /* microseconds */
}
```

time_t类型在sys/time.h文件里被定义为一个整数类型。

select系统调用的定义情况如下所示：

```
#include <sys/types.h>
#include <sys/time.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct
timeval *timeout);
```

select调用的作用是检查那个文件描述符集合里是否有一个文件描述符已经处于读操作就绪状态、写操作就绪状态或有一个错误排队的状态；如果都没有，就阻塞到这些状态有一个出现为止。

nfds参数给出了需要进行测试的文件描述符个数，测试将对第0到第(nfds-1)个描述符进行。三个描述符集合都可以是一个空指针，这表示不进行相应的测试。

select函数会在以下情况返回：readfds集合里有描述符处于读操作就绪状态、writefds集合里有描述符处于写操作就绪状态、errorfds集合里有描述符遇到一个错误条件。如果这三种情况都不成立，select就会在timeout指定的倒计时时间经过之后返回。如果timeout参数是一个空指针，套接字上又没有任何活动，这个调用就会一直阻塞下去。

当select返回时，描述符集合会被修改为指示着那个描述符正处于读就绪、写就绪或有错误状态。你需要使用FD_ISSET对它们进行测试，找出需要关照的那些个描述符。某些系统还会把timeout值修改为倒计时的剩余时间，但这并不是X/Open技术规范里的规定行为。如果select是因为倒计时时间到而返回的话，所有描述符集合都将被设置为空。

select调用的返回值是这些集合里状态发生了变化的描述符的总数。如果失败，它将返回“-1”，并且会设置errno来描述那个错误。可能出现的错误情况包括表示无效描述符的EBADF、表示因中断而返回的EINTR、表示nfds或timeout取值错误的EINVAL等。

> Linux会把参数timeout指向的结构修改为指示倒计时的剩余时间，但UNIX的大多数版本都不这么做。许多现有的使用了select函数的代码的做法是初始化一个timeval结构，然后就一直用下去，不再对它的内容重新进行初始化。这类代码在Linux上的操作可能会不正确，这是因为Linux会随着倒计时的进行而对那个timeval结构做出相应的修改。如果读者正在编写或者移植使用了select函数的代码，就要注意这个差异，并且永远要初始化这个倒计时时间值。这两种行为都是正确的，只是有所不同而已！

下面这个select.c程序演示了select函数的使用方法。我们稍后还会看到和讨论一个更完整的例子。这个程序以2.5秒为倒计时时间来读键盘（即标准输入——它的文件描述符是“0”）。它只有在输入就绪的时候才会去读键盘。它很容易通过添加其他的描述符而得到扩展，比如串行线、管道、套接字等，具体做法要取决于应用软件的需要。

> 动手试试：select系统调用

1）开始部分都是老套路，必要的头文件和定义。然后，对inputs进行初始化，准备处理来自键盘的输入。

```
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>
```

```
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main()
{
    char buffer[128];
    int result, nread;

    fd_set inputs, testfds;
    struct timeval timeout;

    FD_ZERO(&inputs);
    FD_SET(0,&inputs);
```

2) 在stdin上最多等待输入2.5秒钟。

```
while(1) {
    testfds = inputs;
    timeout.tv_sec = 2;
    timeout.tv_usec = 500000;

    result = select(FD_SETSIZE, &testfds, (fd_set *)NULL, (fd_set *)NULL,
                    &timeout);
```

3) 在经过了这段时间之后，我们对result进行测试。如果还没有输入，程序将再次循环。如果出现了一个错误，程序退出运行。

```
switch(result) {
case 0:
    printf("timeout\n");
    break;
case -1:
    perror("select");
    exit(1);
```

4) 如果在倒计时等待期间里这个文件描述符上有所动作，我们就在stdin上读取输入数据并在接收到一个“<end of line>”（文本行尾）字符后把它们都回显到屏幕上去。如果输入字符是“Ctrl-D”，就退出整个程序。

```
        default:
            if(FD_ISSET(0,&testfds)) {
                ioctl(0,FIONREAD,&nread);
                if(nread == 0) {
                    printf("keyboard done\n");
                    exit(0);
                }
                nread = read(0,buffer,nread);
                buffer[nread] = 0;
                printf("read %d from keyboard: %s", nread, buffer);
            }
            break;
        }
    }
}
```

当我们运行这个程序的时候，它会每隔2.5秒显示一个“timeout”字样。如果我们敲击键盘，它就会去读标准输入并报告敲入的内容。对大多数shell来说，输入会在用户按下回车键或某个控制序列键时被发送到这个程序去，所以这个程序会在我们按下回车键的时候把输入内容显示出来。需要注意的是回车字符本身也算是一个字符，它会象其他字符一样被读取和处理（你可以不按回车键试试，看看输入几个字符再按下“Ctrl-D”组合键会是什么样）。

```
$ ./select
timeout
hello
read 6 from keyboard: hello
fred
read 5 from keyboard: fred
timeout
^D
keyboard done
$
```

操作注释：

这个程序使用select调用来检查标准输入的状态。程序通过事先安排的倒计时时间值每隔2.5秒显示一个“timeout”消息——如果select的返回值是零，就表示倒计时结束了一次。在文件的末尾，我们给标准输入描述符的输入就绪标志置位，但没有读到任何字符。

多客户

如果我们那个简单的服务器程序使用了select来实现对多个客户同时进行的处理，它就不必再依赖于子进程了。在把这个技巧实际运用到应用程序中去的时候，一定要注意不要在处理第一个连接时让其他进程等待太长的时间。

服务器可以把select同时用在监听套接字和客户连接套接字两个地方。只要select指示有动作发生，我们就可以用FD_ISSET遍历所有可能的文件描述符，查明动作到底是发生在哪一个描述符上的。

如果是监听套接字处于读操作就绪状态，就说明正有一个客户在试图建立连接，我们就可以在无阻塞风险的前提下调用accept。如果某个客户描述符被指示为处于就绪状态，就说明那里有一个客户请求需要我们读取和处理。如果读操作读到的是零字节，就表示有一个客户进程已经结束了，而我们也就可以关闭那个套接字并把它从我们的描述符集合里删除掉。

动手试试：一个改进的多客户/服务器系统

1) server5.c是本章最后一个例子，我们在这个最后的程序里要用sys/time.h和/sys/ioctl.h头文件替换掉signal.h文件，并且要为select调用额外定义一些变量。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;
    int result;
    fd_set readfds, testfds;
```

2) 为服务器创建并命名一个套接字。

```
server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
server_address.sin_port = htons(9734);
server_len = sizeof(server_address);

bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

3) 创建一个连接队列，初始化readfds集合，为处理来自server_sockfd套接字的输入做好准备。

```
listen(server_sockfd, 5);

FD_ZERO(&readfds);
FD_SET(server_sockfd, &readfds);
```

4) 现在开始等待客户和请求。我们给timeout参数传递了一个空指针，所以将不会出现倒计时时间到的问题。如果select调用的返回值小于“1”，程序将退出执行并报告出现了一个错误。

```
while(1) {
        char ch;
        int fd;
        int nread;

        testfds = readfds;

        printf("server waiting\n");
        result = select(FD_SETSIZE, &testfds, (fd_set *)0,
            (fd_set *)0, (struct timeval *) 0);

        if(result < 1) {
            perror("server5");
            exit(1);
        }
```

5) 一旦得知有动作发生，我们就将用FD_ISSET依次检查每一个描述符，看看动作是发生在哪一个套接字上面的。

```
for(fd = 0; fd < FD_SETSIZE; fd++) {
    if(FD_ISSET(fd,&testfds)) {
```

6) 如果动作发生在套接字server_sockfd上面，它肯定是一个新的连接请求，我们就把相关的client_sockfd添加到我们的描述符集合里去。

```
if(fd == server_sockfd) {
    client_len = sizeof(client_address);
    client_sockfd = accept(server_sockfd,
        (struct sockaddr *)&client_address, &client_len);
    FD_SET(client_sockfd, &readfds);
    printf("adding client on fd %d\n", client_sockfd);
}
```

7) 如果动作不是发生在服务器方面，那它肯定是客户方面的动作。如果接收到的动作是close，就说明客户已经离开，我们就要把它从描述符集合里删除掉。如果是其他动作，我们就象前面的例子里那样对该客户进行“服务”。

```
            else {
                ioctl(fd, FIONREAD, &nread);

                if(nread == 0) {
                    close(fd);
                    FD_CLR(fd, &readfds);
```

```
                    printf("removing client on fd %d\n", fd);
                }

                else {
                    read(fd, &ch, 1);
                    sleep(5);
                    printf("serving client on fd %d\n", fd);
                    ch++;
                    write(fd, &ch, 1);
                }
            }
        }
    }
}
```

> 在实际工作中，最好给程序添上一个专门用来保存已连接套接字最大编号（它可不一定是最新连接的套接字的编号）的变量。这可以避免循环检查可能多达几千个其实并没有被连接的套接字，它们根本不可能处于什么读操作就绪状态。出于简洁方面的考虑，又为了使代码比较容易阅读和理解，我们没有在此这样做。

在运行服务器的这个版本的时候，它将在一个进程里对多个客户顺序进行处理。如下所示：

```
$ server5 &
[7] 1670
$ server waiting
client3 & client3 & client3 & ps -ax
[8] 1671
[9] 1672
[10] 1673
adding client on fd 4
server waiting
adding client on fd 5
server waiting
adding client on fd 6
server waiting
PID TTY STAT  TIME COMMAND
 1670 pp0 S     0:00 ./server5
 1671 pp0 S     0:00 client3
 1672 pp0 S     0:00 client3
 1673 pp0 S     0:00 client3
 1674 pp0 R     0:00 ps -ax
$ serving client on fd 4
server waiting
char from server = B
serving client on fd 5
char from server = B
serving client on fd 6
server waiting
removing client on fd 4
removing client on fd 5
server waiting
char from server = B
removing client on fd 6
server waiting

[8]   Done                    client3
[9]-  Done                    client3
[10]+  Done                    client3
$
```

14.6 本章总结

在这一章里，我们讨论了进程间通信的又一种方法：套接字。这使我们能够开发出真正的

分布式客户/服务器软件——它们能够跨网络运行。我们简要地介绍几个主机数据库消息函数，以及UNIX如何通过因特网守护进程对标准的系统服务进行处理。我们开发了好几个客户/服务器方面的示例程序，对组网和多客户处理做了比较细致的演示。

最后，我们学习了select系统调用，它允许一个程序同时在几个打开的文件描述符和套接字上等待输入/输出活动的发生。

为了让本章开头的比喻更完整，我们表14-6里对套接字连接和电话线路进行了对比。

表 14-6

电　话	网络套接字
给公司打电话，号码是555-0828	连接到IP地址127.0.0.1
公司总机接线员拿起电话	建立起到远程主机的连接
要求转到财务部	安排使用指定的端口（如9734）
财务部总机接线员拿起电话	服务器从select调用返回
电话转到某个财务经理	服务器调用accept，在分支456号上创建新的套接字

第15章　工具命令语言Tcl

在这一章里，我们要去看看UNIX世界一个激动人心的开发成果，那是一个可扩展的和可嵌入的程序设计语言，它的影响巨大而又深远。Tcl（读做“tickle”，Tool Command Language）语言是John Ousterhout等人多年开发的成果，现在由Scriptics公司（http: //www.scriptics.com）负责着它的维护工作。从快速建模工具到扩展语言再到工业控制软件，许多不同领域的应用程序现在都把Tcl当作一种时髦的程序设计语言选择。Tcl还可以在其他平台上使用，包括微软的Windows和苹果公司的MacOS等在内。

我们将对Tcl语言的主要功能进行学习，主要内容包括Tcl语言的可扩展性和它在新软件交互式操作接口方面的用途用法，再向大家介绍几个比较成熟的Tcl语言扩展和用这种语言编写出来的应用软件。我们这一章的讨论范围包括：

- Tcl语言的命令和控制结构。
- 引号、括号和替换。
- Tcl语言中的字符串和列表。
- 输入和输出。
- Tcl语言的扩展和应用软件。

15.1　Tcl语言概述

我们先来学习Tcl语言的基本元素：如何编写和执行Tcl程序以及这种程序设计语言的功能特点。

15.1.1　第一个Tcl程序

我们来编写一个非常简单的“Hello World”程序hello.tcl，看看Tcl程序是如何被编写出来的，又是如何被执行的。下面就是这个程序的源代码：

```
set s "Hello World"
puts $s
```

Tcl是一种解释型语言，Tcl程序通常被称为脚本（script）。从这方面看，Tcl和UNIX操作系统的shell有很多的相似之处。事实上，Tcl脚本需要有一个Tcl专用的shell才能执行，这个shell也因此得名为tclsh。

我们在tclsh下运行我们的脚本程序示例。请记住，用这种办法来执行一个Tcl程序时只要求有文件的读权限就足够了。

```
$ tclsh hello.tcl
Hello World
$
```

类似于其他的UNIX命令解释器，我们可以直接运行tclsh，再由它来提示我们输入立刻执行的Tcl命令。也就是说，我们可以象下面这样直接敲入这两行语句：

```
$ tclsh
% set s "Hello World"
Hello World
% puts $s
Hello World
```

注意tclsh的命令行提示符是“%”字符，我们的命令在敲入后立刻得到了执行。如果一条命令有多个输入行，解释器就会一直读到该命令输入完成才开始执行它。我们还可以使用source命令让tclsh从一个文件里取得命令。我们再用这个办法让Tcl另外读一次我们的程序：

```
% source hello.tcl
Hello World
% exit
$
```

exit命令的作用是结束Tcl shell并返回到我们原来UNIX shell去。下面这个办法可以把我们的Tcl脚本转换为一个UNIX程序：在代码的第一行指定用来执行这个脚本的解释器，就象我们在UNIX的shell脚本程序里做的那样。我们把新程序保存在hello2.tcl文件里：

```
#!/usr/bin/tclsh

set s "Hello World - 2"
puts $s
```

现在，如果我们设置上这个脚本的可执行属性，就可以用平常的办法运行它了。Tcl语言的shell，即tclsh程序的存放位置会随系统的不同而发生变化，在我们的例子里，它被安装在/usr/bin子目录里了。

> 通常，用whereis和which命令就可以在系统的搜索路径里找出tclsh的存身之处。

```
$ chmod +x hello2.tcl
$ ./hello2.tcl
Hello World - 2
$
```

15.1.2 Tcl命令

总的说来，所有Tcl命令的格式都是一样的。它们都以一个关键字（命令的名字）开始，有的命令后面没有参数，有的命令后面会有几个参数。一条命令通常只占用一行，但如果在行尾加上一个反斜线字符（\），就可以让一条长命令占据多个文本行。

引号、括号和变量的替换（我们马上就要讲到）会首先进行，替换后得到的结果才是最终将要执行的命令。这些命令有些会产生一个结果，有些还需要多保存几条命令才能开始执行——比如遇到函数定义或循环语句等时候就会如此。

Tcl语言的语法相当简单易学，扩展起来也比较简单。但某些操作，比如赋值和计算等，乍看上去还是挺吓人的。

我们也可以把多条命令放在同一行上，但必须用分号把它们隔开。

在这一章里，当我们介绍Tcl命令的语法时，我们将使用如下所示的表达方式：

```
command argument1 argument2 ?option1? ?option2? . . . . . .
```

上面这个语法表示命令command有两个不可缺少的参数，还有几个可选用的参数，可选参数的作用一般是改变命令的操作行为。在正式的Tcl/Tk文档里使用的也是这样的语法表达方式。

15.1.3 变量和值

Tcl语言处理的基本上都是一些单个字符串和字符串组。大多数的值都可以当作字符串来处理，但Tcl会在必要时自动完成类型的转换，比如对数字进行计算的时候。给变量赋值要使用set命令，如果变量不存在，就创建它。读者可以把这一小节后面的内容当作是在Tcl shell里的一次旅行。

```
$ tclsh
% set a 123
123
% set hello there
there
%
```

变量名中的字符是分大小写的，并且可以是任意长度。变量名或变量值里面的特殊字符（比如空白字符等）需要用引号括起来，也就是要用双引号把变量名或变量值括起来。和在其他UNIX操作系统的shell里一样，在使用引号时必须小心在意。我们很快就要学习Tcl语言里引号、括号的各种使用方法了。

```
% set "hello there" "fare well"
fare well
%
```

查看一个变量的当前值也要使用set命令，但这次不必给它加上一个值参数了。

```
% set hello
there
% set "hello there"
fare well
%
```

像上面这样交互式地使用Tcl时，解释器会打印出每个命令的执行结果。拿set命令来说，在赋值时我们将会看到变量的新值被打印出来，如果没有给出新值，我们看到的就是该变量的当前值。这是因为set命令的作用就是返回变量的当前值。

如果要在另外一条命令里使用某个变量的值，就必须在它的名字前面加上一个“$”符号，就像我们在UNIX的shell里做的一样。在我们的程序示例里，我们把一个字符串赋值给变量s，然后把它用在了puts命令里，puts命令的作用是通过“$s”形式的引用把它输出到屏幕上去。如果变量的名字里包含着空格，可以使用花括号把它括起来。如下所示：

```
% puts $a
123
% puts [expr $a + $a]
246
% puts ${hello there}
fare well
%
```

如果想在一条命令里引用其他命令的结果，需要把那条命令放在方括号里，上面的第二个例子就是这样做的。这会使Tcl先执行方括号里面的命令，然后把它的值放在原来的地方供其他命令使用。这与我们在UNIX的shell里使用“$(command)”的用法非常相似。在同一个例子里，我们还使用expr命令对一个Tcl表达式进行求值并返回了结果。

我们可以用unset命令从Tcl解释器里去掉一个变量。

```
% unset a
% puts $a
can't read "a": no such variable
```

15.1.4 引用和替换

类似于UNIX的shell，Tcl对引号、括号的使用有很苛刻的限制。在Tcl里，与引号、括号和变量替换有关的规定有很多，你必须掌握它们。

1. 变量的替换

在一个Tcl命令里，只要变量名的前面加上了一个“$”符号，该变量就会被替换为它的值。这也是我们向命令传递参数的机制，在需要保存和使用普通数值或字符串值的时候用的还是这种办法。如下所示：

```
% set apple 5
5
% set orange $apple
5
%
```

2. 命令的替换

如果一条命令被放在方括号里，就会先执行这条命令，然后把它的值放在原来的位置上。如下所示：

```
% set pear [expr $orange + 1]
6
%
```

3. 反斜线字符的替换

我们用一个反斜线字符（\）来取消紧随其后的那个字符的特殊含义。我们可以用它创建出名字或值里包含有特殊字符（比如说美元符号“$”）的命令来。

一个出现在语句末尾的反斜线字符被认为是一个续行标志（也就是说，它取消了换行符的特殊含义）。如下所示：

```
% set fred \$apple
$apple
% set lemon [expr $pear \
+ 1]
7
%
```

4. 字符串引用

创建包含空白或其他特殊字符的字符串时要使用双引号。但就象UNIX的shell里一样，这种引用形式允许进行变量替换。如果变量出现在用双引号括起来的字符串里，就会被替换为它们

相应的值。如下所示：

```
% set joe "Our fruits are orange: $orange, and pear: $pear"
Our fruits are orange: 5, and pear: 6
%
```

5. 花括号引用

另外一种引用办法使用的是花括号“{ }”，它不允许进行变量替换，创建出来的字符串就是原封不动的字面形式。与UNIX的shell里使用单引号的情况相类似，花括号的作用是防止替换的发生。

我们还可以用花括号来保存函数体。我们希望变量和命令的替换发生在它们被执行的时候而不是发生在它们被定义的时候。如下所示：

```
% set moe {Our fruits are orange: $orange, and pear: $pear}
Our fruits are orange: $orange, and pear: $pear
%
```

变量joe和moe之间的区别是显而易见的。moe中的变量替换因为有花括号的存在而没有发生。

如果对花括号及其内容进行求值，实际效果就等于是去掉了最外层的花括号。如果还要做进一步的求值，其结果里将是进行了变量替换之后得到的东西。括号可以嵌套。请看下面的例子：

```
% set fred 1234
1234
% set cmd1 [puts $fred]
1234
% set cmd2 {[puts $fred]}
[puts $fred]
% set cmd1
% set cmd2
[puts $fred]
```

变量cmd1取的是一个空值，即输出变量fred这个操作的返回值。变量cmd2取的值是这个命令本身，而不是操作结果。在这里，因为有了花括号，所以puts的求值和$fred的扩展都没有发生。

这一切看上去相当容易把人给弄糊涂，所以我们来看看当Tcl的shell遇到不同类型的引号、括号时到底会做哪些事情。

6. Tcl解释器替换

Tcl对命令中的变量只进行一轮替换。在执行一个命令的时候，命令行根据其中的空白字符（不包括换行符，它被认为是命令的分隔符）被划分为一个一个的“单词”。被包括在一对双引号、方括号或花括号中的内容（即"..."、[...]或{...}）被看做是一个“单词”。解释器按下面的规则对每个“单词”进行解释：

- 如果“单词”以“$”开始，Tcl将进行变量替换。
- 如果“单词”以左方括号“[”开始，Tcl将进行命令替换，即再次调用Tcl解释器把这个“单词”当作一个新的Tcl脚本来处理。然后用其结果替换掉原来的“单词”。
- 如果“单词”以左花括号“{”开始，单词中的所有内容（除开始和结尾处的左右花括号以外）就都将保持原样，包括空格、“$”符号等。
- 如果“单词”里包含着反斜线字符“\”，Tcl将进行反斜线替换。

这些规则可以总结为表15-7：

表 15-1

单 词	替换方式	结 果
$a	变量	a的值
[commands]	命令	解释执行commands的结果
{the formatted text}	不替换	“the formatted text” 原文
\$word	反斜线	$word

如果结果包含着有进一步的变量引用或嵌套命令，它们将不会被扩展。这就意味着一个Tcl命令的执行情况是可以预见的。尽管如此，在选用引号和括号的时候，你还是要深思熟虑。我们再用下面这几个例子做进一步的说明。

```
% set age 35
35
% set str {I'm $age years old}
I'm $age years old
% set cmd {puts "$str"}
puts "$str"
%
```

这些赋值形成了两层替换。变量cmd里引用了变量str，而str又引用了变量age，变量age的值是35。如果我们想通过cmd来取得age的值，就必须再安排进一步的替换。

如果需要获得第二轮或更进一步的替换，我们可以使用eval命令或者明确地调用subst命令，只有7.4和更高版本的Tcl里有subst命令。eval命令把它的参数分别替换后集中起来，然后把它当作一个字符串传递到Tcl解释器里去执行。如下所示：

```
% eval $cmd
I'm $age years old
%
```

我们看到，最终执行的是cmd的值——“puts "$str"”命令。字符串str的值是“I'm $age years old”。如果我们想在执行puts命令之前展开这个字符串，就需要使用subst命令。subst命令的作用是对字符串进行变量、命令和反斜线替换，然后返回新得到的字符串。两者之间的区别是：eval的参数必须是一个命令，我们对这个命令进行扩展；subst进行扩展但不对其结果求值。如下所示：

```
% subst $cmd
puts "I'm $age years old"
%
```

这里，subst返回了一个有效的命令，这条命令在被求值的时候将输出一个字符串，字符串里包含着age的当前值。如下所示：

```
% eval [subst $cmd]
I'm 35 years old
%
```

替换和执行方面的控制使Tcl语言成为一个功能很强的工具。因为eval命令可以执行任何一个字符串，所以Tcl程序之间可以彼此传递可执行的命令。利用这个技巧可以开发出Tcl语言中的

Tcl调试器，用它来控制其他Tcl程序的执行情况。我们也可以把它用在因特网程序设计中，比如说让服务器发送一个程序到客户那里去执行，或者即时创建一个Tcl脚本等。

7. 注释

Tcl程序可以包含注释。注释就象是一个名字是“#”的Tcl命令，但这条命令不做任何事情。这与其他类似的程序设计语言不同，它意味着我们可以通过在语句末尾加上一个反斜线而使注释延续好几行。特别值得一提的是分号在注释里是不起作用的，所以我们可以在分号后面加上我们的注释。如下所示：

```
# This is a comment
set a 123 ; # This is a trailing comment, note the semi-colon is needed to separate it
\
but the comment continues onto this line; and this is still comment
```

15.1.5 计算

Tcl专门有一个用来完成各种算术计算的命令expr。之所以会这样是因为Tcl解释器本身没有数学函数，它必须依赖由命令提供的额外功能才能完成算术运算。虽然Tcl可以执行有限的算术运算——比如循环语句需要的计数和条件判断等，但大量的计算还是需要由expr命令来完成。

expr命令可以用一个或者更多个参数计算出一个结果来。这些参数可以是运算符、运算数和圆括号。运算数可以被指定为数字（整数或浮点数）、逻辑表达式、将被转换（包括替换操作）为数字值的字符串、变量引用和嵌套命令等。

下面是一些这样的例子：

```
% expr 5 + 5
10
% set a 5
5
% set b 6
6
% expr $a+$b
11
% expr 2**$a.$b"
11.2
% expr 3*(1+[string length "hello"])
18
%
```

能够用在expr命令里的操作符列在表15-2里，它们是按照优先级从高到低的顺序排列的。比较运算的结果用“1”代表真，用“0”代表假。

表 15-2

操作符优先级	说 明
- + ~ !	单元正负符号、按位求反、逻辑非
* / %	乘法、除法、除法求余数
+ -	加法、减法
<< >>	左移、右移（两个运算都带正负符号）

（续）

操作符优先级	说　明
< > <= >=	小于、大于、小于等于、大于等于
== !=	等于、不等于
&	按位与操作
^	按位异或操作
|	按位或操作
&&	逻辑与操作
||	逻辑或操作
cond?yes:no	双元条件判断，类似于C语言中的用法。如果cond非零，返回yes的值；否则返回no的值

下面是最后一个操作符的使用示例：

```
% set k "foo"
foo
% set result [expr {($k == "foo") ? "pass" : "fail"}]
pass
%
```

Tcl还支持在表达式里使用下列的数学函数，它们中的每一个都会调用math函数库里的同名函数。如下所示：

acos	asin	atan	atan2
ceil	cos	cosh	exp
floor	fmod	hypot	log
log10	pow	sin	sinh
sqrt	tan	tanh	

因为经常会用到如下所示的计算结构：

```
set a [expr $a + 1]
```

所以专门为此准备了一个incr命令。在缺省的情况下，它会给一个变量增加一个“1”。也可以给定一个明确的递增数。如下所示：

```
% set a 5
5
% incr a
6
% incr a 5
11
% set a
11
%
```

15.1.6 控制结构

Tcl支持多种程序流程的控制结构，其中包括条件、循环和选择等。

1. if命令

```
if expr1 ?then? body1
if expr1 ?then? body1 ?else? body2
if expr1 ?then? body1 elseif expr2 ?then? body2 ?elseif? ... ?else? ?bodyN?
```

if命令对表达式expr1进行求值的办法与expr命令是一样的。如果结果是一个布尔真值（true，即一个非负的数字，或者一个值为true或yes的字符串），就执行body1并返回它的值。否则，对下一个elseif子句重复同样的处理（如果有的话，可以没有）。如果所有elseif子句中的表达式没有一个求值为一个布尔真值，就将执行else子句（如果有的话）并返回其结果。如果一个子句也没有被执行，if命令将返回一个空字符串。布尔假值（false）是用数值零或者值为false或no的字符串表示的。

body部分只能是一个事物：即一个“单词”或一个用方括号括起来的命令。注意关键字then和else并不是必不可少的，但它们能让程序更容易阅读。编写这一结构的推荐办法是使用下面这样的形式：

```
if {expr1} {
    body1
} else {
    body2
}
```

花括号这种奇怪的布局，特别是把else和花括号写在同一行上的做法，是为了让解释器明白后面还有东西。如果不这样做，Tcl会认为前面已经是一个完整的if语句并正常地执行了它，然后认为下一个语句是以else命令起头的，从而给出一个语法错误。

2. switch命令

```
switch ?options? string pattern body ?pattern body ...?
switch ?options? string {pattern body ?pattern body ...?}
```

Tcl语言中的switch命令直接模仿UNIX的shell里的case语句和C语言中的switch（它只能在整数值上工作）。参数string将依次与各个pattern进行比较（请参考后面内容里对匹配问题的介绍）。如果找到一个匹配，与之对应的body就将被执行并返回其结果。模版“default”的作用是“匹配一切”，它能够匹配所有其他模版不能匹配一切字符串。

options参数控制着将要使用的匹配动作。如表15-3所示：

表 15-3

switch命令的选项	说 明
-exact	字符串和模版完全匹配
-glob	使用全局（即shell风格的）匹配（就象UNIX里的文件名匹配）
-regexp	使用规则表达式进行比较，就象UNIX操作系统的egrep工具程序
--	用来标记选项结束，之所以使用了两个短划线是为了避免与只使用了一个短划线的发生冲突

要想对switch命令的参数做更细致的控制，我们可以通过两种办法来安排匹配模版和程序体。把匹配模版和程序体放在花括号里有两个效果：一是防止被替换，二是不必在语句末尾加上反斜线就能够让匹配模版和程序体延续好几行。

请看下面这个程序，我们在后面的内容里还会见到它，这个程序将展示switch在处理程序的参数方面的用途：

```
foreach arg $argv {
    switch -glob -- $arg {
        -v      {set VerboseFlag true}
        -f*     {set FileToRead [string range $arg 2 end]}
        -h      {puts stderr $Usage; exit 1}
        default {error "bad argument: $arg\n$Usage"; exit 1}
    }
}
```

3. for命令

```
for start test next body
```

Tcl中的for语句与C语言中的for语句非常相似。它的start、next和body参数都必须是Tcl语言的命令字符串，而test必须构成一个布尔表达式。for命令会先执行start字符串，然后对test表达式反复求值。如果它得出的是一个布尔真值（true），就执行body字符串。如果在程序体里遇到一个break命令，循环将立刻终止。程序体中的continue命令会使for循环重新开始。每执行完一次循环，就对next表达式进行一次求值。请看下面的例子：

```
% set n 5
5
% set result 1
1
% for {set i $n} {$i} {incr i -1} {
    set result [expr $result * $i]
}
% set result
120
```

大家应该能够看出来这段程序的作用是求n的阶乘。我们稍后将看到如何创建一个Tcl过程来完成这一功能。

4. while命令

```
while test body
```

while命令的操作情况与C语言中的while语句差不多。它的作用是反复对test字符串进行求值并执行Tcl命令body，直到test产生一个布尔假值（false）为止。类似于for命令中的情况，程序体中的break或continue命令将分别终止或继续循环的执行。请看下面这个例子，它模仿UNIX的cat命令对文件foo进行了处理。文件处理方面的内容请参看后面介绍输入输出的有关小节。

```
% set fd [open "foo" "r"]
file3
% while {[gets $fd line] != -1} {
    puts "$line"
}

<The file "foo" is displayed at this point>

% close $fd
```

15.1.7 错误处理

当Tcl解释器遇到一个错误的时候，它的常见做法是打印一条错误消息然后停止执行。但我

们有时候并不想在程序里这么做，Tcl也为此准备了一个引发和捕捉错误并对之进行修补恢复的功能。

1. error命令

```
error message ?info? ?code?
```

error命令的作用是产生一个错误，如果这个错误没有被捕捉或陷落，就会终止命令的执行。消息message的作用就象是应用程序的一个指示灯，告诉我们什么东西出了问题。如果采取的是解释器缺省的错误处理动作，这条消息就会显示给用户。

如果还给出了info参数，它就会被添加到全局变量errorInfo里去。这个命令里累记着错误发生时的命令嵌套信息。随着每一条Tcl命令的执行，它会被添加到errorInfo变量里，从而形成一个堆栈跟踪记录。类似地，code参数的作用是把机器可读的信息添加到全局变量errorCode里去。详细情况请参考Tcl的有关文档。

2. catch命令

```
catch script ?varname?
```

catch命令对给定的script进行求值，陷落任何可能发生的错误。如果还给出了变量名varname，它就会被设置为该段脚本的返回值或者任何返回的错误信息。

下面这个例子取自我们后面将要看到的Tcl程序concord：

```
if [catch {set Input [ open $FileToRead r ] } res ] {
    puts stderr "$res"
```

即使试图打开输入文件的操作失败了，concord程序也会继续执行，不会因这个错误而停下来，这在某些情况下是非常有用的。如果不象这样陷落有关的错误，许多程序就不可能运行成功。

15.1.8 字符串操作

Tcl从本质上来说就是一个基于字符串的语言解释器，所以它有众多对字符串和字符串值进行操作处理的命令也就没什么奇怪的了。

对字符串进行处理的主要命令是string。根据将要执行的操作，它会用到一个选项和几个参数。

1. string命令

```
string option arg ?arg? . . .
```

string命令的选项（及其相关参数）分别介绍如下。在用到或返回字符串下标时，它永远是以零为起点计算的，也就是说，第一个字符的下标是0，第二个字符的下标是1，依次类推。

(1) first、last、compare选项

```
string first string1 string2
string last string1 string2
string compare string1 string2
```

first和last的作用是在string2里搜索string1的出现情况，如果找到匹配，就返回string2中的起始下标；如果string1没有出现在string2里，就返回“-1”。如下所示：

```
% string first "o" "foobar"
1
% string last "o" "foobar"
2
```

compare的作用是进行字符对字符的比较。当string1小于、等于、大于string2的时候分别返回“-1”、“0”或“1”，与C语言中的strcmp命令是一样的。如下所示：

```
% string compare "foo" "bar"
1
```

(2) index选项

```
string index string num
```

index从string里返回一个字符。被返回的是下标位置为num处的那个字符，注意第一个字符的下标是0。如下所示：

```
% string index "ABC" 1
B
```

(3) length选项

```
string length string
```

length返回的是字符串string的长度。如下所示：

```
% string length "ABC"
3
```

(4) match选项

```
string match pattern string
```

如果string匹配pattern，match返回“1”；否则返回“0”。这里进行的匹配在做法上与UNIX的shell所使用的文件名通配符扩展匹配很相似。如下所示：

```
% string match "*B*" "ABC"
1
% string match "*B*" "ZZZ"
0
```

(5) range选项

```
string range string first last
```

range返回的是string从下标位置first（从0开始）到last之间的那个子字符串，last可以使用关键字end表示字符串的尾。如下所示：

```
% string range "ABCDEF" 3 end
DEF
```

(6) tolower和toupper选项

```
string tolower string
string toupper string
```

这两个参数的作用是把string中的字符转换为相应的大小写形式后返回一个新的字符串。如下所示：

```
% string tolower "ABC"
abc
```

```
% string toupper "abc"
ABC
```

(7) trim、trimleft、trimright选项

```
string trim string ?chars?
string trimleft string ?chars?
string trimright string ?chars?
```

这些参数返回的都是string的子字符串，但其中去掉了chars集合中的特定字符。trim去掉首尾处的字符，而trimleft和trimright分别去掉首字符和尾字符。如果没有给出chars，被去掉的就是空白字符。如下所示：

```
% string trim "foobar" "for"
ba
% string trimleft "foobar" "for"
bar
% string trimright "foobar" "for"
fooba
```

(8) wordstart和wordend选项

```
string wordstart string index
string wordend string index
```

wordstart返回的是一个string中的下标值，它是包含着位置index的那个单词的第一个字符的下标。wordend返回的也是一个string中的下标值，它是包含着位置index的那个单词后面那个字符位置的下标。如下所示：

```
% string wordstart "Tcl is Cool" 5
4
% string wordend "Tcl is Cool" 5
6
```

2. Glob扩展匹配

在“string match”和“switch -glob”命令里要用到模版匹配的一种格式。这涉及到通配符，如果你曾经使用过UNIX的shell，对通配符就应该有所了解。这里所说的通配符与shell的case语句和文件名匹配操作所使用的是同样一些字符。

如果说一个字符串匹配了一个模版，就等于说除了模版中可能出现的特殊序列以外，它们必须是一模一样的如表15-4所示。

表 15-4

*	匹配任何字符序列
?	匹配任何一个字符
[...]	匹配集合中的任何一个字符
\<char>	匹配特殊字符<char>本身，比如“*”将匹配一个“*”字符

3. regexp和regsub匹配

regexp和regsub是我们稍后将要学习的两个命令，另外switch命令也有一个“-regexp”选项，在这些命令里，字符串的匹配要使用模版来进行。这些模版里可以有由规则表达式给出的复杂的匹配，如果读者使用过egrep工具或/ed/vi/emacs编辑器，就应该比较熟悉这些内容。

如果说一个字符串匹配了一个规则表达式，就等于说除了模版字符串里可能出现的特殊序列以外，它们必须是一模一样的。这些特殊序列见表15-5：

表 15-5

.	匹配任何一个字符
*	表示匹配前一项目的零次或多次出现
+	表示匹配前一项目的一次或多次出现
|	表示匹配两个表达式之一
[...]	匹配集合中的任何一个字符
^	匹配字符串的开始
$	匹配字符串的末尾
\<char>	匹配特殊字符<char>本身，比如“*”将匹配一个“*”字符

在规则表达式里还可以定义子匹配，方法是给它们加上一对圆括号“()”。它们可以用来从匹配中提取出子匹配，或者用在regsub的替换操作中。

4. append命令

```
append varname ?value value . . .?
```

append命令把value参数追加到变量varname当前值的末尾。如果参数的值比较长，这样做要比一个等价的赋值语句更有效率——如果你能给原来的字符串追加内容，为什么还要拷贝并丢弃它呢?

给定如下所示的命令：

```
% set a "This is: "
This is:
% set b "a long string"
a long string
```

则下面两条Tcl命令是等价的（都可以得到“This is: a long string”），但第二种做法更有效率。

```
% set a $a$b
% append a $b
```

5. regexp命令

```
regexp ?options? expression string ?match? ?submatch submatch . . . ?
```

regexp命令使用规则表达式在string里寻找匹配。如果它找到了一个匹配，就会返回“1”，否则返回“0”。expression是regexp命令用来寻找匹配的给定格式。如果该规则表达式在字符串里被找到了，变量match将被设置为字符串string里匹配上该表达式的那部分内容；而可选的submatch变量将被设置为字符串匹配了规则表达式里圆括号中的子匹配情况的那些个部分。下面的例子查找的是一个以“3”开始的数字：

```
% regexp {3[0-9]*} 3112
1
% regexp {3[0-9]*} 5112
0
```

或者

```
% regexp 3\[0-9]* 3112
```

```
1
% regexp 3\[0-9]* 5112
0
```

注意我们需要用花括号或者反斜线来阻止Tcl解释命令开头部分里的方括号。可以用来控制匹配操作情况的选项见表15-6：

表 15-6

-nocase	匹配将区分字母的大小写
-indices	使submatch变量被设置为一对切片下标，这对下标值将把匹配到的子字符串分割开

6. regsub命令

```
regsub ?options? expression string subst varname
```

regsub命令按照一个给定的规则表达式对string进行匹配，然后把一个新字符串写到变量varname里去。新字符串是sunst参数的一个拷贝，但已经完成了有关的替换。这些替换包括：

- “&”或“\0”被替换为匹配字符串。
- “\1”到“\9”被替换为第一到第九个子匹配（即圆括号中的子表达式）。

下面的例子把数字中前导的“3”替换为“5”：

```
% regsub {3([0-9]*)} 3112 {5\1} res
1
% set res
5112
```

动手试试：字符串匹配和替换

下面是一个字符串匹配和替换操作的例子。

```
% set date1 "Wednesday August 11 1999"
Wednesday August 11 1999
% set date2 "Saturday April 1 2000"
Saturday April 1 2000
% set weekend "(Saturday|Sunday)"
(Saturday|Sunday)
% regexp $weekend $date1
0
% regexp $weekend $date2 day
1
% puts $day
Saturday
% set parsedate "(\[A-Z]\[a-z]+) +(\[A-Z]\[a-z]+) +(\[0-9]+) +(\[0-9]+)"
([A-Z][a-z]+) +([A-Z][a-z]+) +([0-9]+) +([0-9]+)
% regexp $parsedate $date2 date day month dom year
1
% puts "$date breaks into $day, $month, $dom, $year"
Saturday April 1 2000 breaks into Saturday, April, 1, 2000
% regsub $parsedate $date2 {\4 \2 \3} newdate
1
% puts $newdate
2000 April 1
%
```

操作注释：

我们使用了好几个regexp调用把日期匹配为星期几，并把有关信息从日期中提取了出来。我

们用了一个子匹配来重新安排日期的格式。变量parsedate被设置为描述日期值的一个规则表达式，它使用了以下几个关键格式：

- [A-Z][a-z]+：匹配以一个大写字母开始，由两个或更多个字母组成的单词，比如日期和月份的名称。
- <space>+：匹配一个或多个连续的空格，比如日期中各个元素之间的空格等。
- [0-9]+：匹配一个十进制数字，比如日期中的年份数字等。

当你在匹配或子替换里使用反斜线或方括号时，必须注意保护它们不会因为被放在双引号里而被扩展或替换掉。另外的办法是使用花括号预防所有的扩展。

15.1.9 数组

数组是一组特殊变量组成的集合，对它们进行访问时先要给出数组的名字，后面再紧跟上一个圆括号中的下标。你不能单独使用数组的名字把它做为一个整体来处理。

Tcl支持的数组格式被称为关联数组（associative array）。数组允许使用任意形式的下标切片，这就使Tcl中的数组成为一个功能强大的概念。也就是说，数组可以使用字符串来寻址它们的元素。这就使我们能够把通用对象的有关信息保存在一个数组里，而对象本身可以被用做数组的下标。用一个例子可以说得更明白。

> 当我们讨论和处理关联数组的时候，下标切片和元素通常被称为关键字和关键值。

```
% set myarray(0) 123
123
% puts $myarray
can't read "myarray": variable is array
% puts $myarray(0)
123
%
```

数组下标并不仅仅局限于数字值，但把它们限制为只能是数字或不带空格和特殊字符的字符串是很有必要的，因为引号/括号的用法和变量的替换会把事情弄得很复杂。比较常见的技巧是用一个变量来存放下标的值，再把这个变量设置为需要的值。这就省略了与特殊字符打交道时的难度。

```
% set myarray(orange) 1
1
% set myarray(apple) jim
jim
%
```

array命令

```
array option arrayname ?arg arg . . .?
```

数组可以用array命令来处理，这个命令有好几个选项。array命令的选项（及其关联参数）有：

(1) exists选项

```
array exists arrayname
```

如果arrayname是一个数组，返回“1”；否则返回“0”。

(2) get选项

```
array get arrayname ?pattern?
```

它以下标和关联值对的形式返回一个数组元素列表。如果还给出了可选的pattern参数，就只返回数组里那些下标与模版相匹配（扩展型匹配）的元素。

(3) names选项

```
array names arrayname ?pattern?
```

它返回一个数组下标名清单，或者只列出那些与pattern匹配的下标。

(4) set选项

```
array set arrayname list
```

这个选项的作用是设置数组arrayname中的元素。list必须是一个类似于get选项返回值那样包含着成对的下标及其取值的列表。

(5) size选项

```
array size arrayname
```

它返回一个数组的长度尺寸。

下面是array命令各种选项的使用示例：

```
% array exists myarray
1
% array size myarray
3
% array names myarray
orange 0 apple
% array get myarray
orange 1 0 123 apple jim
%
```

array命令还有一些其他的选项，其中的startsearch、anymore、nextelement和donesearch选项可以对数组元素进行逐个的搜索。详细资料请参考Tcl的文档。

15.1.10 列表

除字符串和数组以外，Tcl还全面支持列表。列表是成组的Tcl对象，数字、字符串以及其他列表都可以组合到一起进行操作处理。Tcl语言中的列表允许的操作有创建、添加元素、转换为字符串和从字符串转换为列表、用来控制循环语句等。我们下面就来向大家介绍几个对列表进行操作的命令。

Tcl列表的语法是用花括号括起来的一组数据元素。事实上，我们已经在循环语句的程序体里见过列表了。

```
% set mylist { bob {1 2 3 { 4 5 } } fred }
bob {1 2 3 { 4 5 } } fred
```

这个赋值语句创建了一个有三个元素的列表。第二个元素本身又是一个四个元素的列表。注意Tcl只有在必要时才会把列表周围的花括号显示出来，以显示其正确的布局结构。类似于数组和字符串，列表中的元素可以通过位置下标进行访问，这些下标也是从零开始计算的。

如果你需要使用一个数组，可使用的又都是数字形式的切片下标，就可以考虑使用一个列表来代替数组。它们通常比更通用的数组更有效率。

1. list命令

```
list ?arg arg arg . . .?
```

在程序中我们可以通过许多办法创建出列表来，但最简单的办法莫过于使用list命令了。list命令用自己所有的参数创建出一个列表，列表中的一个元素就是list命令的一个参数的复制品。如果没有给出参数，list命令就创建一个空列表。

我们需要在必要时加上花括号或反斜线字符，使原始的命令参数可以用lindex命令（说明见下面）提取出来，而列表则可以用eval命令进行求值（如果第一个参数是某个命令的名字的话）。如下所示：

```
% set elist [list puts "Hello World"]
puts {Hello World}
% eval $elist
Hello World
```

2. split命令

```
split string ?delimiters?
```

split命令的作用是从一个字符串开始创建出一个列表来。在缺省的情况下，split创建的列表其各个元素就是string字符串中（彼此以空格隔开）的各个单词。可选的delimiters参数是一个字符列表，字符串中的单词就是根据这个字符列表中的字符拆分的。一个空的分隔符列表并不意味着没有分隔符了，而是意味着要把字符串中的全体字符一个一个地拆分为列表的元素。如下所示：

```
% split "hello there jim"
hello there jim
% split "hi there everyone" {e}
{hi th} r { } v ryon {}
```

注意，分隔符并不出现在列表里，但如果字符串里有紧挨着的分隔符，就会创建出空元素来。

在使用split命令的时候，我们必须对反斜线替换多加注意。“\$”将被转换为一个“$”而不是一个“\”和一个“$”。换句话说，如果想把“\”用做列表的一个元素，就必须使用“\\”的形式。如果想原文包括“$foo”，就必须使用“\$foo”的形式。

3. join命令

```
join list ?delimiters?
```

join命令的作用正好与split命令相反。它通过递归地把一个列表中的所有元素合并在一起的办法创建出一个字符串来。它使用可选的delimiters字符串（如果有的话）来分隔各个元素。它的缺省值是一个空格。如下所示：

```
% join {1 2 fred {bill bob} 37} ","
1,2,fred,bill bob,37
%
```

4. concat命令

```
concat ?arg arg arg . . .?
```

concat命令的可以有多个参数，它把这些参数看做是列表并把它们合并在一起。如果你没有给它提供参数，就会得到一个空结果，而不是一个错误。concat在合并列表的时候会把各个参数最外层的花括号去掉，从而使列表参数的顶层成员变成为结果列表的顶层成员。

参数的前导空格和尾缀空格都会被去掉。每个列表元素用一个空格分隔开。如下所示：

```
% concat {1 2} {3 {4 5} 6} 7 8
1 2 3 {4 5} 6 7 8
```

5. lappend命令

```
lappend listvar ?arg arg arg . . .?
```

我们可以用lappend命令往列表里添加新的元素。与使用concat命令创建一个新列表相比，它一般会更有效率。

lappend命令的每个参数都会被当作一个列表元素而添加到现有的listvar列表里去。新添加的各个元素彼此之间用一个空格分隔开。注意它与concat的做法是不一样的，它不会去掉最外层的花括号，每个参数都是按原样添加进去的。如下所示：

```
% set alist {1 2 3 {4 5}}
1 2 3 {4 5}
% lappend alist 6 {7 8}
1 2 3 {4 5} 6 {7 8}
%
```

6. lindex命令

```
lindex list index
```

我们通过lindex命令从一个列表里把下标位置index处的那个元素检索出来。列表元素的下标编号是从零开始的。如果列表里没有被检索的元素，就将返回一个空字符串。我们可以用关键字end代替一个下标值来表示列表中的最后一个元素。如下所示：

```
% set alist {1 2 3 {4 5}}
1 2 3 {4 5}
% lindex $alist 3
4 5
%
```

7. linsert命令

```
linsert list index arg ?arg arg . . .?
```

我们可以用linsert命令在列表里插入一个元素。由linsert创建出来的新列表包括list中的元素，但从index指示的位置之后要插入arg参数。如果index是零（或者是负数），新元素将被放在新列表的最开始。如果index大于或等于列表list的长度，或者是关键字end，新元素就将被插入到列表的尾部。注意原来的列表本身并不会受到linsert命令的影响。如下所示：

```
% set alist {{4 5}}
{4 5}
% linsert $alist 0 1 2 3
1 2 3 {4 5}
% linsert [linsert $alist 0 1 2 3] end {6 7 8}
1 2 3 {4 5} {6 7 8}
%
```

8. llength命令

```
llength list
```

我们可以用llength命令获得一个列表的长度，它返回的是一个字符串，其内容是一个对应于列表list长度的十进制数字。空列表返回的长度值是零。如下所示：

```
% llength hello
1
% llength "1 2 3"
3
% llength {a b {c d}}
3
```

注意Tcl会把许多不同的参数自动解释为列表。在这种情况下，加引号或不加引号的字符串以及用花括号括起来的字符串组都会被看做是列表。字符串中的每个“单词”会被解释为一个一个的元素。

9. lrange命令

```
lrange list first last
```

lrange命令的作用是提取出列表中的一个连续的子集。列表list中从first到last位置之间的元素会被提取出来做为一个新的列表，它的返回值就是新列表中的那些元素。如下所示：

```
% set alist {1 2 3 {4 5} 6 7 8}
1 2 3 {4 5} 6 7 8
% lrange $alist 2 4
3 {4 5} 6
%
```

10. lreplace命令

```
lreplace list first last ?arg arg arg . . .?
```

lreplace命令的作用是替换列表中的元素。列表list中从first到last位置之间的元素将被替换为给定的参数。原来的元素将被删除，新的元素将被插入。位置first处的元素必须存在，但将要被删除的元素个数和将要被插入的个数倒不必是一样的。我们可以用关键字end代替first或last来指定列表中的最后一个元素。如下所示：

```
% set alist {1 2 3 {4 5} 6 7 8}
1 2 3 {4 5} 6 7 8
% lreplace $alist 2 end bill {bob joe}
1 2 bill {bob joe}
%
```

11. lsearch命令

```
lsearch ?option? list pattern
```

我们可以用lsearch检索出列表里匹配上一个模板的元素来。它的选项包括-exact、-glob（缺省值）和-regexp，它们的作用是确定匹配操作的类型。Tcl匹配操作的详细情况请参考前面字符串部分的有关内容。

lsearch命令用给定的模板pattern在给定的列表list里检索与之匹配的元素。第一个匹配上的元素的下标就是它的返回值，如果没有找到这样的元素，就返回“-1”。如下所示：

```
% lsearch {apple pear banana orange} b*
2
```

12. lsort命令

```
lsort ?option? list
```

我们可以通过lsort命令按一些规则对一个列表进行排序。缺省的动作是对列表中的元素按ASCII字母序列的升序进行排序，也就是按字母表的顺序进行排序，加上标点符号。用计算机的眼光看，这就是字符的字节值按数字从小到大的顺序进行排序。如下所示：

```
% lsort {apple pear banana orange}
apple banana orange pear
```

lsort命令用来改变排序方法的选项见表15-7：

表 15-7

lsort命令的选项	说 明
-ascii	普通字符比较，缺省情况
-integer	元素被转换为整数再做比较
-real	元素被转换为浮点数字再做比较
-increasing	按升序进行排序
-decreasing	按降序进行排序
-command	把command用做排序函数。它必须是一个以两个列表元素做参数的过程，并且要根据第一个元素是否小于、等于或大于第二个元素相应地返回一个负整数、零或正整数

13. foreach命令

```
foreach varname list body
```

foreach命令使我们能够分别求值或使用列表中的每一个元素。这个命令会把列表list中的每一个元素依次赋值给变量varname并执行body。请看下面的例子：

```
% foreach i {1 2 3 bill bob joe} {
    puts "i is $i "
}
i is 1
i is 2
i is 3
i is bill
i is bob
i is joe
%
```

这是foreach命令的一个简化形式；在更具普遍意义的情况里，foreach命令可以使用一个变量名列表和好几个列表。在这些情况下，列表们会被同时遍历。每经过一次循环，变量就将被赋值为对应列表里的下一个元素。如下所示：

```
% foreach {a b} {1 2 3 4 5 6} {puts "a is $a, b is $b"}
a is 1, b is 2
a is 3, b is 4
a is 5, b is 6
% foreach a {1 2 3} b {4 5 6} {puts "a is $a, b is $b"}
a is 1, b is 4
a is 2, b is 5
a is 3, b is 6
%
```

foreach程序体里的break或continue命令与它们在for命令里的作用是一样的。如果在程序体里遇到了一个break命令，循环就将立刻终止。程序体里continue命令将跳过循环结构中当前循环的剩余语句。

foreach循环和“array names”命令合用在一起就能够有效地对一个数组进行遍历：

```
% set myarray(orange) 1
1
% set myarray(apple) 2
2
% set myarray(banana) 3
3
% foreach fruit [array names myarray] { puts "fruit is $fruit" }
fruit is orange
fruit is apple
fruit is banana
%
```

15.1.11 过程

Tcl语言中的过程与其他语言中的过程是比较相似的，但又有一些有趣的差异。过程是通过proc命令在Tcl解释器里得到定义的。在调用各个过程之前，必须先用proc命令把它们都定义好。

```
proc name args body
```

proc命令定义了一个名为name的过程。它把body（一系列可执行命令）保存起来，等到调用该过程的时候才运行它。执行时传递给过程的参数被替换为变量列表args。

过程在程序体body中的最后一条命令执行完毕时退出。过程的值就是最后执行的那个命令的值。我们可以用return命令在过程尚未全部执行完毕之前中途退出。

```
return ?option? ?string?
```

可选值string将做为这条return命令身处其中的那个过程的返回值。如果省略了string，就会返回一个空字符串。我们可以用return命令的有关选项设定一个过程在需要返回一个错误时将要采取的操作动作。详细资料请参考Tcl文档。

动手试试：过程

下面是一个简单的计算阶乘的过程。

```
% proc fact {n} {
    set result 1
    while {$n > 1} {
        set result [expr $result * $n]
        set n [expr $n - 1]
    }
    return $result
}
% fact 7
5040
```

操作注释：

过程fact通过循环来求解出阶乘值。注意这个过程使用了一个局部变量result。因为在Tcl语言里第一次使用变量之前并不需要预先声明过它们，所以在Tcl的过程里遇到的一切新变量都将被看做是局部变量，也就是说，在某个Tcl过程里创建的新变量在这个过程以外的地方是不存在的。

upvar命令

```
upvar ?level? oldname newname
```

要想获得某个全局变量的访问权，我们必须使用upvar命令。它允许访问调用堆栈里更高层的变量，就它最简单的用法来说，它允许一个过程使用在其上级过程（一般就是调用它的那个过程）中声明的某个变量或者允许它使用一个全局变量。

upvar命令将在该过程里创建出一个新的变量newname。对这个新变量的访问和赋值等价于使用存在于该过程之外的那个oldname变量。我们可以通过可选的level参数来精确地指定oldname变量是来自于哪个堆栈结构。

这个命令最常见的用途就是提供全局变量的访问权，如下所示：

```
upvar variable variable
```

下面是一个upvar命令的正确用法示例：

```
% set result 1
1
% proc badset2 {} {
    set result 2
}
% badset2
2
% set result
1
% proc goodset2 {} {
    upvar result result
    set result 2
}
% goodset2
2
% set result
2
%
```

15.1.12 输入和输出

Tcl为输入和输出操作准备了一个复杂的系统。在这一小节里，我们将对其中的一些基本功能进行讨论。进一步的详细资料请参考Tcl的文档。

1. open命令

```
open filename ?access? ?permissions?
```

open命令的作用是建立到一个文件或一个命令管道的连接。它同时具备C语言中fopen和popen函数的功能。

open命令返回的是一个通道标识符（文件流的Tcl等价物），它可以被用在gets、puts和close等其他输入输出命令的调用中。标准文件流将以stdin、stdout和stderr的形式对Tcl程序自动打开，Tcl程序可以直接使用它们。

如果还给出了access参数的话，它将决定文件的打开方式。这是一个字符串值，与C语言中fopen函数使用的文件打开模式的取值是完全一样的。见表15-8：

表 15-8

access模式	说 明
r	以读方式打开文件filename。该文件必须存在。这是缺省的打开模式
r+	以读写方式打开文件filename。该文件必须存在

（续）

access模式	说　明
w	以写方式打开文件filename。该文件将被创建或截短
w+	以读写方式打开文件filename。该文件将被创建或截短
a	以写方式打开文件filename。新数据将被追加在该文件的尾部
a+	以读写方式打开文件filename。文件的初始读写位置将被设置为文件尾位置

此外，我们还可以通过一个POSIX标志的Tcl列表来设置文件的打开模式，这些POSIX标志包括RDONLY、WRONLY、RDWR、APPEND、CREAT、EXCL、NOCTTY、NONBLOCK、TRUNC等。

如果还给出了permisions参数的话，它是一个用来对open命令所创建的文件的初始访问权限进行设置的整数值（缺省值是八进制的666）。

如果filename参数的第一个字符是一个管道符号（|），它就会被看做是一个命令管道。此时，“|”符号后面的UNIX命令将被执行，而它的输出结果将能够通过由这条open命令返回的通道标识符来读取。

2. close命令

```
close channelID
```

close命令的作用是关闭由channelID指定的通道，这个channelID必须是从以前的某个open调用（如果是在Tcl 7.5或以后的版本里，还包括socket命令）那里获得的。通道相当于C语言里的文件描述符。

3. read命令

```
read ?-nonewline? channel ?bytes?
```

read命令的作用是读取某个通道标识符上全部（或者，如果给出了byte值，就是它给定的字节数）可读的输入。

如果设置了-nonewline选项，那么如果输入中的最后一个字符是换行符，read命令就不会把它读进来。read命令会把行尾的控制字符序列转换为换行符。

4. gets命令

```
get channel ?varname?
```

gets命令的作用是读取输入的第一行数据，如果还给出了varname参数，就把读来的数据保存到这个变量里去。同一通道上后续的gets操作会依次读取下一行数据，当遇到文件尾字符时就停止操作。gets的返回值是实际读入varname里去的字符个数，出错时将返回“-1”；如果没有给出varname，就返回字符串本身。如下所示：

```
% set myfile [open "|date" r]
file3
% gets $myfile fred
28
% close $myfile
% set fred
Wed Aug 11 15:03:35 BST 1999
```

5. puts命令

```
puts ?-nonewline? ?channel? string
```

puts命令把输出写到一个通道去（缺省写往标准输出）。如果没有给出-nonewline选项，向通道写完string后还要再跟上一个换行符。换行符会自动转换为代表文本行尾的控制字符序列。

注意Tcl有内部缓冲功能，所以puts命令输出的东西可能不会立刻出现在一个输出文件里。我们可以用flush命令强制性地把数据输出到文件里去。下面是一个简单的例子，如果读者还想了解更多的东西，请参考Tcl的文档。

```
% open "fred.txt" w
file3
% puts file3 "Hello World"
% close file3
% open "fred.txt" r
file3
% gets file3
Hello World
% close file3
%
```

6. format命令

```
format formatstring ?arg arg arg . . . ?
```

Tcl语言中的format命令相当于C语言库函数sprintf，它们之间使用着许多相同的格式转换符。事实上，在format命令的实现中确实用到了sprintf函数。

format命令的作用是输出一个格式经过编排的字符串。格式转换符方面的详细资料请参考Tcl的使用手册页（用命令“man format”命令）或与printf和sprintf函数有关的C语言函数库文档。

7. scan命令

```
scan string formatstring var ?var var . . . ?
```

Tcl语言中的scan命令相当于C语言库函数sscanf。指定为参数的那些变量将被设置为根据formatstring中给出的格式转换符从字符串string里提取出来的值。类似于刚才介绍的format命令，scan命令的实现与ANSI C语言中的sscanf非常相似，格式转换符方面的详细资料也请读者参考有关的C语言函数库文档。

8. file命令

```
file option name ?arg arg . . . ?
```

Tcl语言中的文件处理操作是通过file命令实现的。改变和查看文件的属性需要用到这个命令的各种选项。file命令使用的选项包括以下这些：

(1) atime和mtime选项

```
file atime name
file mtime name
```

它们分别返回一个描述文件name的最后一次访问/修改时间的字符串。

(2) dirname选项

```
file dirname name
```

它返回文件name的子目录路径部分，即最后一个“/”字符前所有的字符。如下所示：

```
% file dirname "/bin/fred.txt"
/bin
```

(3) exists和executable选项

```
file executable name
file exists name
```

如果文件存在/可执行，就返回“1”，否则返回“0”。

```
% file exists "fred.txt"
1
% file executable "fred.txt"
0
```

(4) extension和rootname选项

```
file extension name
file rootname name
```

分别返回文件的扩展名（文件名最后一个句点后面的部分，包括该句点）和文件的根名字（文件名最后一个句点前面的部分）。

```
% file extension "fred.txt"
.txt
```

(5) isdirectory和isfile选项

```
file isdirectory name
file isfile name
```

如果name是一个子目录/普通文件，就返回“1”，否则返回“0”。

(6) owned选项

```
file owned name
```

如果文件name的属主是当前用户，就返回“1”，否则返回“0”。

(7) readable和writeable选项

```
file readable name
file writeable name
```

如果文件name可读/可写，就返回“1”，否则返回“0”。

(8) size选项

```
file size name
```

返回以字节计算的文件长度。

根据读者使用的操作系统，你还可能能够使用其他的选项。其中包括用来追踪某个Tcl数组里stat或lstat系统调用结果的选项。详细情况请参考系统方面的文档。

15.2　一个Tcl程序

我们用下面这个简单的程序来演示部分Tcl语言中的功能。它的作用是统计一个文本文件里各个单词使用的频率。它可以被用做某个单词索引软件的组成部分。

动手试试：一个单词使用频率统计程序

1) 敲入下面给出的concord.tcl程序清单。在开始部分，先给出这个脚本在其下执行的shell并

对一些变量进行初始化。

```
#! /usr/bin/tclsh

set VerboseFlag false
set FileToRead "-"
set Usage "Usage: concord \[-v] \[-f<filename>]"
```

2) 记住argv是程序的参数构成的数组，我们来分析命令行参数。设置FileToRead变量并给出使用帮助。

```
foreach arg $argv {
    switch -glob -- $arg {
        -v      {set VerboseFlag true}
        -f*     {set FileToRead [string range $arg 2 end]}
        -h      {puts stderr $Usage; exit 1}
        default {error "bad argument: $arg\n$Usage"; exit 1}
    }
}
```

3) 把缺省的输入源设置为标准输入。如果指定了一个文件，就“安全地”打开它。

```
set Input stdin

if {$FileToRead != "-"} {
    if [catch {set Input [open $FileToRead r]} res ] {
        puts stderr "$res"
        exit 1
    }
}
```

4) 对单词和文本行计数器进行初始化，然后读入输入中的每一行，根据标点符号拆分各行，增加单词使用频率统计数组中相应的计数值。

```
set NumberOfLines 0
set NumberOfWords 0

while {[gets $Input line] >= 0} {
    incr NumberOfLines
    set words [split $line " \t,.\{\}\(\)\[\];\""]
    foreach word $words {
        if {[info exists concord("$word")]} {
            incr concord("$word")
        } else {
            set concord("$word") 1
        }
        incr NumberOfWords
    }
}
```

5) 输出最后的统计结果，然后输出所有找到的单词，再往后输出每一个单词及其计数值。

```
puts stdout [format "File contained %d/%d words/lines\n" \
        $NumberOfWords $NumberOfLines]

puts stdout [array names concord]

foreach word [array names concord] {
    puts "$word: $concord($word)"
}
```

我们用一个文本文件来试试这个程序，比如这个程序的源代码文件。当我们用下面的命令执行这个程序的时候：

```
$ chmod +x concord.tcl
$ ./concord.tcl -fconcord.tcl
```

我们将得到如下所示的输出（略有删节）：

```
File contained 404/48 words/lines

{"$Input"} {"if"} {"exists"} {"%d/%d"} {"$word:"} {"--"} {"error"} {"$words"}
{"$FileToRead"} {"info"} {"set"} {"$argv"} {"contained"} {"NumberOfWords"} {"range"}
{"argument:"} {"!="} {"bad"} {"VerboseFlag"} {"Input"} {"catch"} {"cannot"}
{"$concord"} {"words"} {"File"} {""} {"0"} {"else"} {"while"} {" v"} {"word"}
{"split"} {"-h"} {"-f<filename>"} {"line"} {"concord"} {"format"} {"1"} {"switch"}
{"exit"} {"stdin"} {"stdout"} {"\"} {"stderr"} {"$NumberOfLines"} {"default"} {"$arg"}
{"\t"} {"$Usage"} {"#!/usr/bin/tclsh"\ {"2"} {"open"} {"r"} {"names"} {"-f*"}
{"string"} {"$word"} {"$arg\n$Usage"}
...

"names": 2
"-f*": 1
"string": 1
"$word": 4
"$arg\n$Usage": 1
"NumberOfLines": 2
"arg": 1
"$line": 1
"incr": 3
"puts": 5
"Usage": 1
"array": 2
"Usage:": 1
"FileToRead": 2
"end": 1
"foreach": 3
"oops": 1
">=": 1
"gets": 1
"false": 1
"--": 2
"true": 1
"$NumberOfWords": 1
"-glob": 1
"words/lines\n": 1
```

操作注释：

出于演示的目的，这个程序在显示单词频率统计清单之前会先显示它找到的单词列表。数组中的单词顺序是没有定义的，它依赖于关联数组在Tcl语言里的具体实现办法。如果我们想得到一个经过排序的输出结果，可以把数组元素们拷贝到一个列表里去，再使用lsort命令来完成这一工作；或者，我们可以把输出经管道送往UNIX的sort程序。

这个程序的工作原理是：先把输入行拆分为一个一个的单词，去掉标点符号，再把这些单词用做一个关联数组的下标。我们给这个数组起名为concord，它被用来统计单词的使用频率。我们用一个加上了双引号的字符串做为这个数组的下标，目的是防止因单词中包含着特殊字符而可能引发的问题。

网络支持

在Tcl 7.5往后的版本里，Tcl已经能够直接支持联网功能，而以前这需要有第三方增值软件才行。Tcl语言灵活而又简洁的特性加上网络程序设计范例使Tcl的用途更加广泛了。事实上，甚

至有完全用Tcl语言编写的万维网浏览器呢。其中的关键命令就是socket。

```
socket host port
```

在缺省的情况下，socket命令打开一个SOCK_STREAM连接到指定的计算机host上和服务编号port。它将返回一个能够用在普通Tcl输入/输出命令里的通道标识符。

动手试试：

为了演示Tcl中套接字的使用方法，我们来重温一下第14章里的一个小例子，并把它和下面的Tcl程序进行比较。下面就是用Tcl语言编写的socket.c程序的完整清单，它的作用是显示某个联网计算机上的日期和时间。

为了让这个程序能够正常地工作，请把主机名tilde替换为你自己内部网络上某个系统正确的名字，或者也可以使用localhost。

```
#!/usr/bin/tclsh

set sockid [socket tilde 13]
gets $sockid date
puts "Tilde says the date is: $date"
```

操作注释：

Tcl（7.5以上版本）内建有对UNIX网络功能的支持。这个程序使用socket命令创建了一个到达主机tilde第13号端口，即daytime服务的通信路径。它从套接字连接读取了一个字符串，然后把它显示出来。

请比较一下这个只有三行的解决方案和第14章里的大程序！

在这个程序里使用gets会造成一个隐患。某些个别的系统管理员会故意把他们的daytime服务配置为返回一个非常长的字符串的情况，而这可能会给你的机器造成不良影响。这类不良行为已经在因特网上发生过了，人们称之为“拒绝服务”攻击。

15.3 创建一个新Tcl语言

Tcl语言的设计目标之一就是它应该是可以嵌入的，因此我们可以把用Tcl语言编写的脚本包括在其他命令行程序里，甚至可以包括在C语言程序里。事实上，这正是Tcl的起因和人们把它称为工具命令语言的原因。从这一方面看，它与Perl非常相似。

把Tcl结合到你的程序里是相当容易的，只要编写一个简单的对Tcl解释器进行初始化的函数就可以实现这一点。你可以用Tcl函数库来链接你自己的函数，从而创建出符合你自己的特定需求的Tcl解释器来。

这个机制的具体做法超出了本章的讨论范围。事实上，许多Tcl程序员从不创建他们自己的解释器。他们认为基本的Tcl系统再加上第三方提供的Tcl扩展已经足以满足他们的需要了。

15.4 Tcl语言的扩展

经过多年的努力，现在已经有了许许多多对Tcl语言的扩展。它们大多数都是以Tcl语言附加功能版的面目出现的，专门服务于某一特殊的目的。我们将在这里对其中的几个做一个简单的介绍。

15.4.1 expect

expect程序实际上是一个经过功能扩展的Tcl解释器，其附加功能的主要目的瞄准着交互式程序的自动化控制。我们可以用它来对软件进行测试，这是因为我们可以编写expect脚本向其他程序传递命令、等待响应、再对那些响应做出反应。它的内建功能包括倒计时和错误恢复。它能够在接收到一些不同的事件时运行相应的Tcl函数。

> expect是由NIST（National Institute of Standard and Technology，国家标准和技术研究院）的Don Libes在Tcl语言在因特网上发行不久的时候编写的。它是围绕Tcl建立的第一个大型的程序。

这个扩展的得名源于它所增加的两个重要函数。它们一个是send，用来向另一个被测试程序发送交互式输入数据；另外一个是expect，用来安排在接收到不同响应（或没有接收到有关响应）时的程序动作。

15.4.2 [incr Tcl]

这个扩展的名字是C++语言的同义词，它给Tcl增加了面向对象的功能。

15.4.3 TclX

TclX是意思了“extended Tcl”（扩展了的Tcl）。因为它里面包含了许多对应于UNIX系统调用的附加函数，所以它是UNIX环境里最常用的扩展。TclX程序能够访问到操作系统的许多底层功能，但在可移植性方面付出了一定的代价。

15.4.4 图形

为了支持图形及其相关的附加操作，人们已经开发出了许多Tcl语言扩展。

Tk

一个由John Ousterhout开发的重要的Tcl扩展是Tk（“Tool Kit”，工具箱）。它使Tcl获得了创建和处理图形化用户操作界面对象的能力。利用Tk，我们可以很容易地编写出复杂的图形化程序。

Tk最早是为X窗口系统开发的，但现在它已经有了能够在微软的Windows和苹果电脑公司的MacOS上运行的版本（当然还包括Tcl本身）。因此，Tcl/Tk程序有能力在不做任何修改的前提下在多种硬件平台上运行。

我们将在下一章以Tk程序设计为主题进行讨论。

15.5 本章总结

在这一章里，我们对Tool Command Language（工具命令语言，简称Tcl语言）进行了学习。我们已经看到，它是一种影响深远的可扩展解释型程序设计语言，支持许多高级程序设计语言才具备的功能，比如关联数组和列表等。

我们简要地介绍了Tcl的几个扩展，这些新创建出来的解释器满足了某些专用软件领域的需要。

我们将在下一章里对Tk（即Tcl语言的图形程序设计工具包）进行学习，因为它的出现，UNIX环境下的许多软件被开发和推广开来。

第16章　X窗口系统的程序设计

在这一章和下一章里，我们将学习如何编写能够运行在UNIX图形化环境——X窗口系统（简称X）里的程序。现代的UNIX系统和几乎所有的Linux发行版本都带有某个版本的X。

我们将主要从程序员的角度看待X，并且假设读者对自己系统上X的配置、运行和使用都已经很熟悉了。

我们将讨论以下几个问题：

- X概念。
- X窗口管理器。
- X程序设计的模型。
- Tk；它的素材、绑定和几何尺寸管理器。

在下一章里，我们将学习GTK+工具包，它使我们能够使用C语言为GNOME系统编写用户操作界面程序。

16.1　什么是X

X诞生于MIT（麻省理工学院），开发它的目的是为了给图形化程序提供一个统一的环境。现如今，只要读者使用过计算机，基本上就可以肯定你曾经遇到过微软的Windows、X或苹果公司的MacOS这三者之一，因此你应该熟悉常用的与图形化用户操作界面（即GUI）有关的概念。需要提醒大家的是，虽然一个Windows用户能玩得转Mac的操作界面，但对程序员来说它们可是完全不同的两码事。

每种系统上的每一种窗口化环境在程序设计上都是不同的。屏幕显示的控制和程序与用户之间的通信是不同的。虽然每种系统都为程序员提供了打开和处理屏幕上的窗口的能力，但使用的具体函数一般是不同的。编写能够运行在超过一种系统上的应用程序（不使用额外的工具包）几乎是一个不可能完成的任务。

为了克服大型主机、小型机和工作站上有专利权的接口系统给我们带来的问题，X窗口系统对公众是完全开放的，并且已经在许多种系统上得到了实现。它定义了一个基于客户/服务器模型的程序设计风格，在依赖于硬件的组件和软件应用程序之间有清晰的界线。

X窗口系统主要由四个部分组成，我们将简要对它们进行介绍。它们是：

- X服务器：与用户交互操作。
- X协议：客户/服务器之间的通信。
- X库：程序设计接口。
- X客户：软件应用程序。

16.1.1　X服务器

X服务器，或者叫X显示服务器，是一个运行在软件用户计算机上的程序，它负责图形化显

示硬件设备的控制工作，完成具体的输入和输出操作。X服务器响应来自X客户程序的请求，在屏幕上“画画”或者读取键盘或鼠标的输入。它负责传递输入数据以及向客户程序报告鼠标移动与按钮动作等事件。

虽然X能够运行在许多种不同的硬件组合上，但一般说来，这些不同的硬件组合必须使用一个彼此不同的X服务器。Linux和其他基于PC的系统上最常见的X实现是XFree86（它的网址是http：//www.xfree86.org）。这个软件包里的X服务器都是为PC个人电脑上使用的各种显卡专门编写的，比如XF86_S3就是S3系列显卡专用的X服务器。Linux用户应该好好谢谢这些开发人员。

16.1.2 X协议

X客户软件和X显示服务器之间的一切交互操作都必须通过交换消息才能进行。消息的类型和用途用法就构成了X协议。X窗口系统特别有用的一个功能是：X协议不仅能够穿越网络，就是对运行在同一台机器上的客户和服务器之间也同样适用。这就意味着即使用户手里只有一台功能较低的个人电脑或一台X终端（这是一种专为运行X服务器而设计和使用的机器），他也可以在更强大的联网计算机上运行各种X客户程序，而交互式操作和输出显示都是在他自己的本地机器上进行的。

16.1.3 Xlib库

只有那些实际编写X服务器的人才会真正对X协议感兴趣。大多数X软件都要使用一个C语言函数库做为程序设计的接口。这就是Xlib库，它为X协议里的信息交换提供了一个API（应用程序设计接口）。Xlib本身并没有增加太多的东西——它只能在屏幕上画线条和对鼠标动作做出响应。如果你需要菜单、按钮、卷屏条以及所有其他的东西，就必须自己编写它们。

从另一个方面看，Xlib也没有强调任何特殊的GUI风格。它的作用只是一个中介，你可以通过它创建出自己想要的风格来。

16.1.4 X客户

X客户就是以后实际接触的软件程序，它们需要运行在某个计算机上，但是能够使用其他计算机上的显示和输入资源。它们通过向负责管理自己的X服务器提出对那些资源的访问请求而做到这一点。服务器一般都能够同时对来自许多客户的请求进行处理，它必须对键盘和鼠标在客户之间的使用情况做出裁决。客户程序使用X协议消息与服务器进行通信，而这些消息是通过Xlib函数来收发的。

16.1.5 X工具包

我们不准备在Xlib程序设计接口上多做停留，因为如果你想又快又简单地编写出程序来，它可算不上是最佳的工具。Xlib的底层接口就象是Microsoft Windows SDK开发工具包一样，它能对付相当复杂的程序，但不怎么出活。笔者书架上的一本书里有一个完全是利用Xlib库编写出来的“Hello World”程序。它在一个窗口里显示“Hello World”；窗口里还有一个“Exit”（退出）

按钮，按下这个按钮时程序就退出运行，除了这些它什么也不能做。可它的程序清单居然有五页长！

任何编写过这类Xlib程序的程序员肯定都希望找到一个更好的办法。而好办法确实有！常用的操作界面元素如按钮、卷屏条和菜单等早就被实现过很多次了。X窗口系统里的这类元素也叫做素材，把它们收集在一起就形成了人们所说的X工具包。最知名的X工具包包括随X提供的“Xt Intrinsics”开发工具包和两个商业化的产品：Sun公司的OpenLook和OSF（开放源代码基金会）的Motif。

- Xt是在X的上面编写的一个免费函数库，它给Xlib库增加了一些功能，是一个能够简化应用程序设计的跳板。
- OpenLook是Sun公司产品的一个免费的工具包，它强调了一种另类的观感。它是在一个名为Xview的函数库上面建立起来的，这个库与Xt很相似。
- Motif是OSF组织的一个标准，设计目的是为UNIX桌面提供统一的观感。Motif又分为两个主要部分：一组用来定义Xt函数中使用的各种常数的头文件和一个用来简化对话框和菜单等元素的创建工作的易于使用的函数库。Motif还定义了一种程序设计风格，不管程序员是否使用Motif工具包，都可以参照它来设计自己的程序。
- Qt是一个由Trolltech公司出品的函数库，它构成了KDE桌面环境的基础，在大多数Linux发行版本里都能找到它。
- GTK+就是GIMP工具包，它是GNOME系统的基石。我们将在下一章里介绍如何对这个高级环境进行程序设计。

> 和所有其他商业性UNIX供货商一样，Sun公司已经接受了用Motif工具包开发出来的CDE（Common Desktop Environment，通用桌面环境）。这等于是宣告了OpenLook的结束，它肯定会逐渐地消失掉的。OpenLook的某些组成部分现在已经免费对公众开放了，在某些Linux发行版本里可以看到它们。

每一种X工具包都实现有一整套素材，它们的外观和感觉一般都有明显的区别。显示在屏幕上的元素可以是扁平而又普通的样子（比如用Xt开发的软件），也可以是有雕塑感的3D效果（比如Motif）。

想知道一个工具包会造成多大的差异吗？我们来看看Linux里两个不同的文本编辑器xedit和textedit。第一个xedit是一个非常简单的编辑器，几乎没有什么复杂的用户操作界面可言。要想加载一个文件，就必须先把文件名敲到一个框子里，再点击那个标记着“Load”的按钮。

与它形成鲜明对照的是textedit编辑器，这是一个Sun公司OpenWindows软件中的一个程序，它是用OpenLook工具包编写的。在打开文件的时候，它提供了一个对话框。用户可以浏览文件系统挑出自己想要打开的文件。这个工具包还为按钮提供了我们熟悉的3D效果。

16.2 X窗口管理器

X窗口系统另外一个重要的组成部分是窗口管理器（window manager）。这是一个特殊的X客户，负责处理其他的客户。它安排客户窗口在显示器屏幕上的摆放位置，负责完成移动窗口

和调整窗口尺寸等管理性的工作。根据它使用的X工具包种类的不同，用户会看到一个它特有的外观效果。

窗口管理器的种类有很多，如表16-1所示：

表 16-1

窗口管理器	说 明
twm	全名是Tom's（或Tabbed）Window Manager，X带来的一个小而快的管理器
tvwm	Robert Nation编写的一个窗口管理器，在Linux里广受欢迎。它支持虚拟桌面，自带配置文件，并且能够通过配置文件模仿其他窗口管理器
Fvwm95	fvwm的一个模仿Windows 95操作界面的版本
Gwm	全名是Generic Window Manager，可以用一种LISP语言的变体进行编程
Olwm	全名是OpenLook Window Manager
Mwm	全名是Motif Window Manager

大多数UNIX和Linux系统上都有这些窗口管理器，但mwm需要有一份许可证。

16.3 X程序设计模型

我们已经看到X窗口系统用一个通信协议在客户应用程序和X显示服务器之间划出了清晰的功能界线。这种程序设计方法就引导出一个用来编写X应用程序的典型结构，我们在这里给大家做一个简单的介绍。

16.3.1 启动

典型的X应用程序在启动时必须对自己可能会用到的一切资源都进行初始化。它需要和X显示服务器建立起一个连接，选择使用的颜色和字体，然后在显示器上创建出一个窗口来。

客户程序连接和解除连接一个X服务器时要使用XOpenDisplay和XCloseDisplay函数。下面是它们的定义情况：

```
Display *XOpenDisplay(char *display_name );
void XCloseDisplay(Display *display );
```

display_name参数指定的是我们打算连接的显示设备。如果它是null，就使用环境变量DISPLAY的值。它的格式是“hostname:server[.display]”，一台主机可以有一个以上的X服务器，每个服务器又可以控制一个以上的显示设备。系统默认的显示设备通常就是“:0.0”，即本地计算机上第一个可用的服务器。如果想指定一个第二屏幕，比如桌面确实很大的时候，你可以使用“:0.1”。

XOpenDisplay返回的是一个Display结构，里面是刚才选择的X服务器的有关信息；如果没有X服务器可以被打开，就返回null。只有在成功地从XOpenDisplay返回之后，客户程序才能开始使用X服务器。

当客户程序用完X服务器的时候，它必须以最初由XOpenDisplay调用返回的那个Display结构为参数调用XCloseDisplay。这将清除该客户在显示设备上创建的一切窗口和其他资源；除非曾经调用XSetCloseDownMode修改过窗口的退出整理行为，但这种做法是很少见的。程序在退

出前必须调用XCloseDisplay，只有这样才能使排队中的错误得到报告。

用户可以控制启动时的大多数活动。许多X应用程序能够响应命令行参数、环境变量和配置文件中相应的设置情况，使用户能够对应用程序进行定制。后面的内容里就有几个这样的例子。

正如我们已经看到的，环境变量DISPLAY被用来把应用程序引向每个特定的显示服务器，而这个服务器可以是在另外一台联网的计算机上。下面的命令将启动xedit程序执行，但会把它的显示打开在一台名为alex的机器上。

```
$ DISPLAY=alex:0.0 xedit &
```

.Xresources文件（有时候是.Xdefaults文件）被用来配置X应用程序。每个应用程序都会使用X资源数据库里的配置数据项，这些配置项通常是在一个X系统启动时创建出来的，并且包括了用户自己的本地优先设置。用户的.Xresources文件保存在他或她的登录子目录里，其中一个典型的的配置项如下所示：

```
xedit *enableBackups: on
```

这个配置项改变了xedit的行为，让它在编辑文件的同时对文件进行备份。配置项的通用格式是：

```
Class *Resource: Value
```

请看下面这个命令行：

```
$ xedit -geometry 400x200
```

它的意思是在一个400点宽200点高的窗口里启动xedit。注意其他程序可能使用其他方式的几何尺寸设置，比如说：

```
$ xterm -geometry 80x50
```

它启动的终端仿真器有50行，每行有80列。影响X应用程序行为的各种设置情况请参考你系统的文档和应用程序的使用手册页。

16.3.2 主循环

X应用程序的主体是由一个主循环和对事件做出反应的代码构成的。在启动之后，一个典型的X程序就会开始等待它连接的X显示服务器向它发送事件。这项工作是在一个循环里通过调用XNextEvent来实现的。

一个应用程序可能需要对多达30个以上的事件做出反应。我们不准备在这里讨论它们，因为已经有许多讨论X窗口程序设计的（非常厚的）书对这一话题有过详细的论述。但我们在表16-2里把X用到的事件大致分分类：

表 16-2

键盘事件	按键的按下和释放
鼠标事件	按钮的按下和释放；鼠标移动；鼠标进入/离开一个窗口
窗口事件	窗口的创建/关闭；窗口获得/失去焦点；窗口被遮蔽/重新显示等

一个底层的X程序需要对这些甚至更多的事件做出响应。而使用了高级开发工具包或应用程

序框架的程序却不必明确地与这些底层的事件打交道，可以把注意力集中到应用程序的主要目的上，使用对话框等复杂的操作接口元素完成相应的操作任务。

16.3.3 退出整理

一个行为良好的X程序会在自己退出的时候释放它在运行时曾经分配的一切X显示资源。其实简单地断开与服务器的连接通常就已经足够了，但这样可能会导致服务器消耗过多的内存。再说了，不打招呼就走也有些不礼貌！

16.4 X程序设计概述

在本章剩下的篇幅里，我们将把X程序设计的底层问题留给那些想榨出应用程序最后一分性能并且拥有对应用程序最大控制权的人们去思考。

至于象我们这样只是急于看到成果和希望把高功能X应用程序弄得好看一些的人们，我们将把注意力集中在X程序设计领域几个最新的发明上。

随着高速个人计算机和工作站的迅速普及，用一种解释型语言来编写至少是程序的用户操作界面部分已经越来越普遍。我们已经在介绍shell和Tcl语言的有关章节里看到了几个这样的例子。我们还将在第18章里去学习掌握Perl语言的强大功能。

我们这一章主要是学习Tk（“Tool Kit”的字头缩写，意思是工具箱）的使用方法，它是Tcl语言在图形化程序设计方面的一个扩展；在下一章里将学习GTK+，它最初是做为一个控制GIMP（GNU Image Processor，GNU图像处理器软件）的工具包而开发出来的，但最终发展成为GNOME桌面里的底层图形化程序设计语言。

用Tk来进行X程序设计还带来了一个移植性方面的好处。许多非X的图形化环境（包括微软的Windows）里也有Tk，并且它不依赖于具体的硬件设备。为一种机器编写的Tk程序能够不经任何修改就运行在另外一种机器上。

如果读者对一个独立于计算机平台的程序设计系统带来的好处感兴趣，并且还希望拥有编译型语言的强大功能，Java将是一个比较不错的选择。但Java程序设计的范围实在是太广泛了，我们无法在这里详细地介绍它们。我们推荐读者从Ivor Horton的“Begining Java 2”一书开始学习，这本书也是由Wrox出版社出版的（国际书号是ISBN 1-861002-23-8）。

16.5 Tk工具包

Tk是Joho Ousterhout在Tcl语言的基础上编写出来的，它收集了大量的图形化用户操作界面（graphical user interface，简称GUI）的基本部件（人们称之为素材），目的是简化X、微软的Windows和苹果电脑的MacOS等几种图形环境下各种基本组件的程序设计工作。

Tk是一个面向动作的、组合式的、可嵌入的、可扩展的、高移植性的、基于事件的开发工具包，它的素材是用C语言编写的，并且要通过Tcl语言绑定到事件处理器（程序）上去。Tk已经被移植到许多其他的语言中，命令的绑定可以用Perl和Python等语言来完成。

最新版的Tk 8.1和Tk 8.2在Unix、Windows和Macintosh等三种平台上都保持着很好的稳

定性。

在缺省的情况下，Tk素材与它们运行在其上的平台的原装素材有相同的外观，使用中的感觉也一样，但它们具有很高的可配置性。你可以通过设置该工具包某个全局变量的办法严格按照Motif模式来使用Tk素材。Tk的程序设计接口是稳定的，为一种平台编写的大多数脚本不需要任何修改就可以运行在其他两种平台上。

> 这一小节里的所有程序示例需要至少8.0版本的Tcl和8.0版本的Tk才能正常工作。你可以从http://www.scriptics.com/resource/software/处下载这两个软件的最新版本。这一小节里的大多数程序都是用Tcl 8.0和Tk 8.0编写出来的，这是因为最新发行的Jacl和Tcl Blend只能和Tcl 8.0版一起工作。Jacl是完全用纯Java重写的Tcl解释器；而Tcl Blend是用C语言编写的一个可动态加载的Tcl扩展，它使Tcl能够更好地工作在一台Java虚拟机上。

在开始Tk程序设计之前，读者需要保证Tk的窗口化shell，即wish，已经安装在自己的系统上了，它的可执行文件都在你的搜索路径PATH里。如果Tk没有被安装到默认的地方，你需要设置环境变量TK_LIBRARY和TCL_LIBRARY指向正确的地点。如果你在自己的机器里安装了多个版本的Tcl，就一定要注意让刚才说的这两个环境变量指向正确的位置。举个例子，下面就是我为8.2b3版Tk启动wish而使用的shell脚本程序：

```
#!/bin/sh
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/tcl8.2b3/unix:/usr/local/tk8.2b3
/uni
x:
PATH=$PATH:/usr/local/bin:/usr/local/tcl8.2b3/unix:/usr/local/tk8.2b3/unix:
TCL_LIBRARY=/usr/local/tcl8.2b3/library
TK_LIBRARY=/usr/local/tk8.2b3/library
export LD_LIBRARY_PATH PATH TCL_LIBRARY TK_LIBRARY
/usr/local/tk8.2b3/unix/wish $*
```

> 当你在shell提示符处敲入wish的时候，会弹出一个小窗口，窗口颜色在缺省情况下是灰色的。如果在运行脚本程序的时候需要取消这个交互式的wish窗口，请用“wish -f”命令来启动Tk。wish可以说是一个内建了Tk函数的tclsh。

在接下来的几个小节里，我们将学习以下几个方面的内容：

- 窗口程序设计的基本概念。
- 编写我们的第一个Tk程序。
- 介绍Tk的各种素材和它们的部分配置选项。
- 绑定。
- 窗口尺寸和屏幕位置的管理。
- 应用程序资源的管理。
- 应用程序间的通信。
- 窗口的管理和应用程序的嵌入。
- 一个完全用Tcl语言编写的由现有Tk组件搭建的复合素材。
- 一个使用了Tcl事件的有日常使用价值的Tk程序示例。

虽然这些内容并不是很深入，但读者应该学习到如何开始进行Tk程序设计、如何使用它的内建素材集合来编程，以及到哪儿去进一步学习Tk，总之，你不再是一个门外汉了。

16.5.1 窗口程序设计概述

利用Tk可以快速地创建出一个图形化的操作界面，与底层窗口系统打交道的工作可以都交给Tk提供的素材去完成。接着，需要把事件处理器（程序）与这些素材结合在一起（通常是使用Tcl语言），这样它们就可以对用户的命令做出反应了。这与图形化程序设计实践的正常流程也是相吻合的。

首先，创建程序的外观，添加必要的组件以访问最终想要包括在程序中的功能。从Tk工具包里挑选一个素材，对它的外观进行初始化，然后用Tk的几何尺寸管理器安排好它在屏幕窗口里与其他素材的相对位置。接着，编写代码，让每一个GUI部件对相应的用户命令做出正确的响应。比如点击“load”按钮要做什么？点击它就加载一个图像到画布素材怎么样？等等。

这与我们在本书前面的内容里学习的过程化程序设计有很大的不同。做为一名程序员，你永远无法控制用户与程序交互操作的动作次序；而GUI的要点就是提供一个更自然更直观的用户操作界面。程序必须等待用户引发的事件并做出相应的动作。

你每次使用Tk的class命令创建出一个素材的时候，GUI就会把新创建出来的素材看做是一个新的Tcl命令，新Tcl命令的名字就是那个素材的名字。接下来，你可以在这个新创建的素材上使用有关的素材命令调用其方法（method，“方法”是面向对象的程序设计中的一个概念，Tk把它们称为配置选项）。素材命令就象是一个面向对象的系统里的对象——当素材被删除的时候，素材命令也将被删除。

请看下面这个素材命令：

```
button .b
```

它创建了一个新素材和一个名为“.b”的素材命令。你可以使用这个新命令与素材进行交流，比如说：

```
.b configure -text "Hello"
```

将把按钮“.b”上的文字设置为“Hello”。如果把“.b”想象为一个对象，这条命令就等于是在该对象上调用configure方法从而把它的text属性设置为“Hello”。

> Tk素材并不完全是面向对象的，因为它们不支持继承，变形等特性。它们与OOP原理的相似之处只有对方法进行调用这一点。

在一个Tk程序里。素材的创建和初始化部分将包括有在用户屏幕上创建和布置该素材的Tcl命令。在你创建出并布置好这些素材之后，它们就可以通过被称为事件处理器（event handler）的Tcl脚本与用户互动了。

在本章所有的程序示例中，我们都将尽量把素材的创建和素材的配置分开介绍给大家，但要想把这两部分完全分开有时候是很困难的，因为事件管理器有时就是在你创建素材的时候绑定上去的。比如说，Tk中的大多数素材都支持一个command素材命令，而它往往需要立刻进行

设置。当然，“创建出素材的时候尽快绑定事件处理器，稍后再安排该素材的屏幕位置”这一做法也是很明智的。这两种办法并没有优劣之分。在你自己的设计实践中，请选用最适当和最容易理解的做法。

为了让这个概述更有意义，我们现在就来看看下面这个hello1.tk程序，它可能是有史以来最短的多行标题签程序了！

动手试试：说“Hello”

把下面的内容敲入脚本程序文件：

```
#!/usr/bin/wish -f

pack [button .b -text "Hello\nWorld!!!" \
                -justify center \
                -width 20 \
                -command {puts "Hi"}]
```

把脚本程序设置为可执行，然后运行这个hello1.tk程序：

```
$ ./hello1.tk
```

这个程序将创建出下图所示的窗口，并且会在你每次点击按钮的时候输出字符串“Hi”。如图16-1所示。

图 16-1

操作注释：

我们来分解这个程序，看看这个“著名的”程序里都发生了哪些事情。

调用“wish -f”命令之后，我们来到完成了所有工作的那一行程序。它相当简练，我们把它扩展到好几行是为了让各种选项更清晰。先别理会pack命令，我们看到“button . . .”创建了一个名为“.b”的按钮，它的多行标题签“Hello World”居中摆放。按钮的宽度被设置为20个字符。“-command”选项给这个按钮连接了一个事件处理器，当用户点击这个按钮时，就会在父窗口里输出字符串“Hi”。注意反斜线的用法，它使你能够把这个命令分开写在好几行上。

pack命令把素材“.b”打包为该应用程序创建的默认顶级窗口，所以它占据了整个窗口。注意“pack [button .b . . .]”的工作情况和我们先初始化按钮“.b”，再调用“pack .b”的效果是一样的。

把这个素材称为“.b”并没有什么特别的原因。你可以把它叫做“.foo”或者其他什么东西，但名字必须以一个句点开始。一个应用程序里的全体素材是逐层排列的，默认的顶级“应用程序”素材及其相应的素材命令都被命名为“.”。每个素材的名字都是一个用句点隔开的层次名表，这个表描述了它在该应用程序的素材层次中的位置。比如说，路径名“.a.b.c”表示素材“.c”是“.a.b”的子、“.a”的孙、应用程序素材“.”的重孙。只要按这个办法排列好路径名，Tk中的任何一个素材都可以有任意个数的子素材。

16.5.2 配置文件

现在，我们在创建“.b”素材的语句前多加上下面这一行：

```
option add *b.activeForeground brown
```

程序再创建“.b”素材的时候会把它缺省的activeForeground（前景颜色）设置为棕色。字母b前面的星号（*）表示不管层次情况如何，任何叫做b的素材都会设置上这个选项。

我们还可以把它弄得更象一个实用的X应用程序，具体做法是把下面这一行：

```
*b.activeForeground brown
```

保存到一个名为hello.def的文件里去，然后在hello1.tk脚本里创建素材之前加上下面这一行：

```
option readfile hello.def
```

这一行的作用是在创建素材之前把应用程序的缺省配置情况从hello.def文件里读出来。

16.5.3 其他命令

你可能在想了：“除-command以外，我还能不能给素材创建更多的用户交互命令？”我们正打算创建一个这样的事件绑定（event binding）。如果用户在按下鼠标按钮的同时按下Ctrl键，该素材将输出字符串“Help !”。

下面是完成这一动作的语句：

```
bind .b <Control-Button-1> {puts "Help !" }
```

把上面这些改动都加到最终的Hello World程序hello4.tk里去，我们得到：

```
#!wish -f

option readfile hello.def
pack [button .b -text "Hello\nWorld!!!" \
                -justify center \
                -width 20 \
                -command {puts "Hi"}]
bind .b <Control-Button-1> {puts "Help!"}
```

这个只有三行的简单程序完成的工作相当于一个500行的Xlib程序或100多行的Motif代码完成的工作。它具有一个基本的X应用程序应该具有的全部特点，但却相当简单和短小。这就是Tk的价值所在。它把图形化用户操作界面程序设计中的复杂和畏惧都抛在一边了。

16.5.4 Tk素材

我们来仔细看看Tk提供的素材库。在查看Tk支持的各种素材之前，先向大家介绍一个找出某个素材所提供的全部方法和属性的简单办法。注意百分号“%”是Tk的wish命令的提示符。

动手试试：多多益善

首先，交互式地创建一个素材“.s”，如下所示：

```
$ wish
% scale .s
.s
```

调用这个素材的config方法，检查输出内容，看看这个素材都提供了些什么，如下所示：

```
% .s config
```

你将看到如下所示的输出：

```
{-activebackground activeBackground Foreground SystemButtonFace SystemButtonFace} {-
background background Background SystemButtonFace SystemButtonFace} {-bigincrement
bigIncrement BigIncrement 0 0.0} {-bd -borderwidth} {-bg -background} {-borderwidth
borderWidth BorderWidth 2 2} {-command command Command {} {}} {-cursor cursor Cursor
{} {}} {-digits digits Digits 0 0} {-fg -foreground} {-font font Font {{MS Sans Serif}
8} {{MS Sans Serif} 8}} { foreground foreground Foreground SystemButtonText
SystemButtonText} { from from From 0 0 0} {-highlightbackground highlightBackground
HighlightBackground SystemButtonFace SystemButtonFace} {-highlightcolor highlightColor
HighlightColor SystemWindowFrame SystemWindowFrame} {-highlightthickness
highlightThickness HighlightThickness 2 2} {-label label Label {} {}} {-length length
Length 100 100} {-orient orient Orient vertical vertical} {-relief relief Relief flat
flat} {-repeatdelay repeatDelay RepeatDelay 300 300} {-repeatinterval repeatInterval
RepeatInterval 100 100} {-resolution resolution Resolution 1 1.0} {-showvalue
showValue ShowValue 1 1} { sliderlength sliderLength SliderLength 30 30} {-
sliderrelief sliderRelief SliderRelief raised raised} {-state state State normal
normal} {-takefocus takeFocus TakeFocus {} {}} {-tickinterval tickInterval
TickInterval 0 0 0} {-to to To 100 100.0} { troughcolor troughColor Background
SystemScrollbar SystemScrollbar} {-variable variable Variable {} {}} {-width width
Width 15 15}
```

每个列表数据项都有下面这样的格式：

option-switch option-name option-class option-default-value option-actual-value.

你可以挨个进行实验，看看这些素材选项都是起什么作用的，它们的缺省值是什么。要想学习素材的方法和选项，研究它的使用手册页是最好的办法了。

1. 框

框（Frame）是所有Tk素材里面最简单的了。它们只能被用做容器，从下面的例子就可以看出来：

```
#!usr/bin/wish -f

. config -bg steelblue

foreach frame {sunken raised flat ridge groove} {
   frame .$frame -width 0.5i -height 0.5i -relief $frame  -bd 2
   pack .$frame -side left -padx 10 -pady 10
}
```

这个脚本以不同的3D边界效果创建了五个框，如图16-2所示。

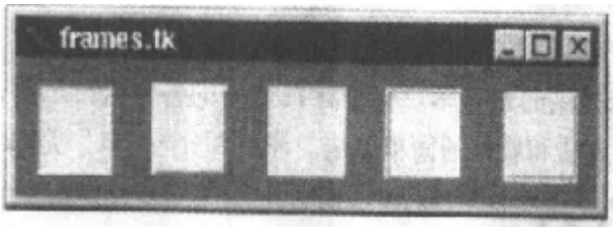


图 16-2

框一般都不可见，并且几乎总是用来创建嵌套形式的窗口布局。

操作注释：

在上面的例子里，-relief选项的作用是允许框的边界呈现凹凸效果，同时“-bd 2”选项把素材边界的宽度设置为两个点。所有Tk素材都支持这个选项，都能呈现3D的效果。

这段代码的其余部分包括用一个Tcl列表创建出五个框来。这些框的尺寸是通过把选项-

height和-width设置为0.5i（半英寸）得到调整的，它们都靠左摆放，并通过-padx和-pady选项在两边留出10个点的间隔。

2. 顶层

顶层（Toplevel）素材类似于框，但它们有自己的顶级窗口，而框则是一个顶层的内部窗口。

```
% toplevel .t -width 1.5i -height 1i  -relief ridge -bd 4
```

图 16-3

将创建出下面这样的一个顶层素材来如图16-3所示：

3. 标题签

标题签（Label）是一种能够显示多行文字的简单素材。我们可以用label命令来创建标题签，如下所示：

```
% label .l -wraplength 1i -justify right -text "Hello Tk World!"
```

这将创建出一个多行的标题签素材，每行的文本长度为一英寸。当你用下面这条命令打包这个标题签的时候：

```
% pack .l
```

图 16-4

它将创建出一个如图16-4所示这样的素材：

创建出标题签以后，就可以通过素材命令与它进行交流了。比如说，可以用下面的命令来查询该标题签素材的前景颜色：

```
% .l cget -fg
Black
```

全体Tk素材都支持cget素材命令，它的作用是检索素材的某个配置选项。我们还可以通过Tk素材的configure方法对配置选项进行交互式设置。如下所示：

```
% .l configure -fg yellow -bg blue
```

这个命令把标题签的前景设置为黄色，背景设置为蓝色。

4. 按钮

Tk提供了三种按钮（Button）：下压按钮、复选框和单选按钮。

按下下压按钮会执行一个动作。复选框用来选择或不选择一组选项。单选按钮类似于复选框，但每次只能从一组选项里惟一地选取一个选择。即使对这些名词不熟悉，这些按钮你肯定见得多了。

我们来看看下面这个例子，它演示了Tk按钮的主要用法。

动手试试：按钮的选用

1）脚本头和几个全局变量之后，我们创建了一个选择按钮来控制选择最喜欢的程序设计语言。

```
#!/usr/bin/wish -f

set lang tcl
set state 1

checkbutton .lan -text "Language" -command {changeState} -relief flat \
                 -variable state -onvalue 1 -offvalue 0
```

2) 接下来，我们创建了一个单选按钮板，每个按钮代表一种语言，如下所示：

```
radiobutton .c -text "C" -variable lang -value c -justify left
radiobutton .tcl -text "Tcl" -variable lang -value tcl -justify left
radiobutton .perl -text "Perl" -variable lang -value perl -justify left
```

3) 我们需要两个下压按钮来控制输出情况，如下所示：

```
button .show -text "Show Value" -command showVars
button .exit -text "Exit" -command {exit}
```

4) 配置完这些按钮之后，还需要把它们布置到屏幕上去。这要用到几何尺寸管理功能。

```
grid .lan  -row 1 -column 0 -sticky "w"
grid .c    -row 0 -column 1 -sticky "w"
grid .tcl  -row 1 -column 1 -sticky "w"
grid .perl -row 2 -column 1 -sticky "w"
grid .show -row 3 -column 0 -sticky "w"
grid .exit -row 3 -column 1 -sticky "w"
```

5) 选择按钮需要有一个对动作进行处理的过程，我们给它起名为changeState。它是用选择按钮的-command选项注册的。

```
proc changeState args {
    global state
    if {$state == "0"} {
        catch {
            .c config -state disabled
            .tcl config -state disabled
            .perl config -state disabled
        }
    } else {
        .c config -state normal
        .tcl config -state normal
        .perl config -state normal
    }
}
```

6) 那个下压按钮也需要有一个类似的过程来处理，我们给它起名为showVars，如下所示：

```
proc showVars args {
    global state lang
    if {$state == "0"} {
        puts "No Language is selected"
    } else {
        puts "The Language selected is $lang"
    }
}
```

运行这个程序的时候，我们将看到如图16-5所示的窗口：

图 16-5

操作注释：

程序在一开始对两个全局变量lang和state进行了初始化，这两个变量的作用是保存选择框和单选按钮的值。

我们定义了一个选择框来选择/不选择“language”选项。每次调用它的时候，它的命令将调用changeState过程。它还要根据执行命令前用户做出的选择把全局变量state相应地设置为“1”或“0”。

接着程序构造出单选按钮，它们用来从三种语言（C、Tcl、Perl）里选择一种。阅读代码就

会知道：这些个按钮共享着同一个全局变量lang，它保存着与当前选择对应的值。这就保证了用户每次只能选取一个单选按钮。

最后，我们定义了两个下压按钮，一个在被按下时将退出这个应用程序，另一个会去调用过程showVars输出当前的选择。

选择框通过命令changeState来改变三个单选按钮的整体状态，根据选择框是否被选中在单选按钮们的活跃和不活跃状态之间进行切换。下压按钮showValue通过命令showVars把当前的选择输出出来。

按钮还支持相当其他的选项，比如鼠标掠过方法等。详细资料请使用man命令去查阅button、checkbutton和option的使用手册页。标题签和按钮还支持位图（bitmap）和图像（image）做为它们的图案。我们稍后将学习图像的用法。

> 例子里以“grid ...”开始的语句其作用是对被创建素材的几何尺寸进行管理。我们将在稍后对几何尺寸管理进行介绍。

5. 消息

消息（Message）类似于标题签，也可以用来显示多行的文本。它们之间的区别是消息能够自动对文本进行换行，从而把它显示在多个行上；换行发生在单词边界处，文字会根据一定的纵横比率显示在窗口里。消息素材可以按指定的段落对齐方式来显示文本，并且还可以处理非打印字符。请看下面的例子：

```
#!/usr/bin/wish -f

message .m -aspect 400 -justify center \
           -text "This is a message widget with aspect ratio 400 and \
                  center justification. Message widgets can also to \
                  display control characters \240 \241 \242 \243 \251 \
                  \256 \257 \258  and tabs \t  etc..."

pack .m
```

这个例子将创建出一个里面包含着控制字符的简单的消息素材。

6. 输入框

输入框（Entry）是一个单行的文本素材，我们可以通过它敲入和显示一个单行的文本。输入框还支持许多编辑文本用的键盘按键绑定。请看下面这个小程序login.tk，它能够处理用户的登录操作，但我们这里没有给出核查用户口令字的代码。

> 动手试试：创建输入框

1) 首先，设置登录窗口的外观。我们还定义了一个全局变量loginName。如下所示：

```
#!/usr/bin/wish -f

set loginName "timB"

label .name -text "Login:"
entry .nameEntry -textvariable loginName
label .passwd -text "Password:"
entry .passwdEntry -textvariable passwd -show *
```

2) 然后我们从.nameEntry选取全部文字。如下所示：

```
.nameEntry selection from 1
.nameEntry selection to end
```

3) 最后，在屏幕上摆放好这些素材——我们一会儿再解释具体的做法！

```
grid .name  -row 0 -column 0 -sticky "w"
grid .passwd -row 1 -column 0 -sticky "w"
grid .nameEntry -row 0 -column 1 -columnspan 2 -sticky "W"
grid  passwdEntry -row 1  -column 1 -columnspan 2 -sticky "W"
```

运行这个程序的时候，我们将看到如图16-6所示的窗口：

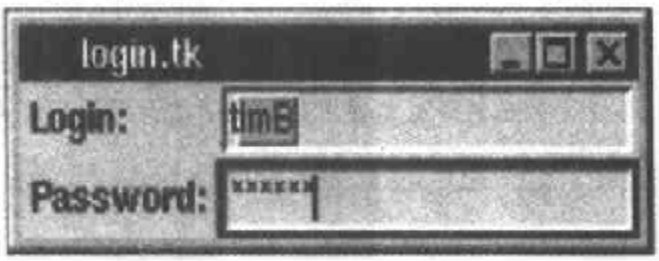


图 16-6

操作注释：

程序开始的前五行代码创建了两个标题签和两个输入框素材，接下来的两行代码对输入框里的文字进行了选取。selection命令是Tk用来在素材之间移动数据资料的方法。最后四行代码把创建好的素材安排在屏幕上排列整齐。

> X定义了一个提供和检索“选取物”的标准机制，而selection命令就是Tk用来管理客户间通信（Inter-Client-Communication）的办法　它遵守《X客户间通信符号约定规则》（X Inter-Client Communication Convention Manual，简称ICCCM）中的规定。我们在这里介绍selection的原因就是想让大家了解你可以在程序里对选取物进行设置，而其他非Tk的X客户可以使用普通的X窗口约定符号检索出那个选取物。

输入框素材通过键盘按键绑定可以实现内部的文本编辑操作。在它由man命令给出的使用手册页里，你会发现输入框素材除了能够支持“the OSF Motif style guide”（《OSF Motif风格指南》）中规定的全部Motif绑定以外，还支持大量的EMACS绑定。表16-3是最常用的几个：

表 16-3

按键绑定	说　明
Ctrl-a	把插入光标移动到输入框文字的开始处
Ctrl-e	把插入光标移动到输入框文字的结尾处
Ctrl-/	选取输入框中的全部文字

7. 列表框

列表框（List Box）素材用来显示一组字符串，并允许用户在其中选取一个或多个项目。下

面这个程序通过列表框设计出一个Motif风格的提示对话框，请注意它里面的用法。

动手试试：列表框

1) 首先，创建用户操作界面的元素。如下所示：

```
#!/usr/bin/wish -f

scrollbar .h -orient horizontal -command ".list xview"
scrollbar .v  -command ".list yview"
listbox .list -selectmode single -width 20 -height 10 \
              -setgrid 1 -xscroll ".h set" -yscroll ".v set"
label .label -text "File Selected:" -justify left
entry .e -textvariable fileSelected
```

2) 为了让素材有Motif风格的外观和动作，需要使用grid几何尺寸管理器。如下所示：

```
grid .list -row 0 -column 0  columnspan 2 -sticky "news"
grid .v -row 0 -column 2 -sticky "ns"
grid .h -row 1 -column 0 -columnspan 2 -sticky "we"
grid .label -row 2 -column 0
grid .e -row 3 -column 0  -columnspan 3 -sticky "we"

grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1
```

3) 我们用当前子目录里的内容初始化这个列表框，如下所示：

```
foreach file [glob *] {
    .list insert end $file
}
```

4) 最后，我们给这个列表框绑定一个事件处理器，让它对鼠标按键1的释放动作做出反应。如果你是一个右撇子，“鼠标按键1”一般对应于鼠标的左键；如果是左撇子，一般就对应于鼠标的右键。我们在这一部分将一直称它为“鼠标按键1”，因为这是代码中使用的惯例。

```
bind .list <ButtonRelease-1> \
    {global fileSelected;set fileSelected [%W get [%W curselection]]}
```

运行这个程序的时候，我们将看到如图16-7所示的窗口：

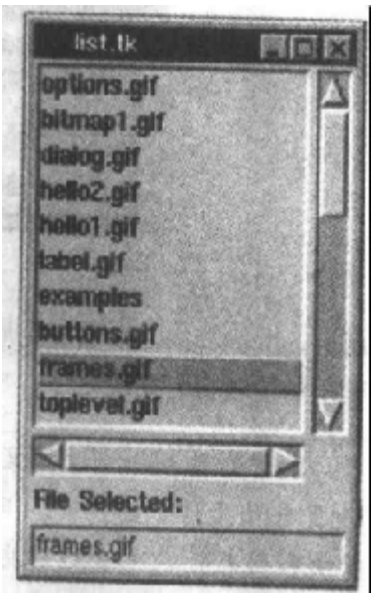


图 16-7

操作注释：

程序先创建出两个卷屏条，再把这两个卷屏条与它创建的列表框联系起来。两种素材通过卷屏条的-command选项和列表框的-xview和-yview命令建立起彼此之间的“内连接”（inter-connection）。我们通过这种办法告诉两种素材如何交流信息，如何根据对方的几何尺寸或状态做出相应的调整。我们还会在这一章里看到更多的内连接。

接着，我们通过foreach循环用当前子目录中的内容对这个列表框进行了初始化。Tcl命令glob的作用是进行模板匹配并返回这些文件名。

列表框为了能够对显示在框里的内容进行处理还提供

了其他一些配置方法，象delete、get、index、insert和scan等。

8. 卷屏条

正如我们在刚才那个例子里看到的，卷屏条（Scrollbar）经常会与其他素材联系在一起，使那个素材的显示区可以被卷动。就拿上例中的列表框来说吧，它的显示区是由两个卷屏条“.h”和“.v”控制的，如下所示：

```
scrollbar .h -orient horizontal -command ".list xview"
scrollbar .v  -command ".list yview"
```

.h通过命令“.list xview”控制着列表框的水平显示区；垂直卷屏幕条“.v”的做法与此相类似。列表框则通过下面的命令获知这一内连接情况：

```
listbox .list ... -xscroll ".h set" -yscroll ".v set"
```

也就是说，我们把这两种素材绑定在了一起并把它们一方的行为通知给对方，我们就是这样让它们彼此通信的。我们在下一小节会看到一个建立隐含内连接的做法。

9. 滑块

滑块（Scale）素材显示一个整数值，用户通过移动一个滑块对这个值进行设取。我们来看一个简单的例子：

```
#!/usr/bin/wish -f

set foo 100
label .l -text "Choose a Value:" -justify left
scale .s -orient horizontal -from 0 -to 2000 -tickinterval 500 \
        -showvalue true -length 3i  -variable foo
entry .e -width 6 -justify left -textvariable foo

pack .l -side top -anchor nw
pack .s .e -side left -padx 4m -fill x
```

运行这个程序的时候，我们将看到如图16-8所示的窗口。

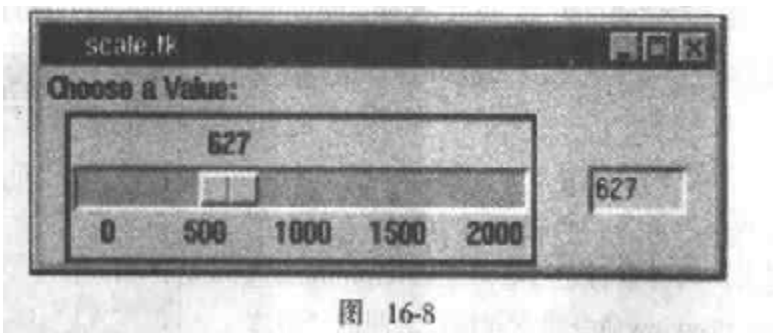


图 16-8

这里，滑块和输入框通过全局变量foo共享着一个隐含的内连接。如果你刷新滑块中的数值，输入框里的值也会自动改变。

> Tk素材的事件处理器里明确使用的一切变量都是全局范围的。如果变量不存在，Tk将自动为你创建一个。因此，在上面的例子里，变量foo就是一个全局变量，而滑块和列表框两个素材则共享着这个变量，这就为它们的行为建立了一个隐含着的联系。

10. 文本

人们用Tk的文本（Text）素材来创建多行的可编辑文本，它的用途是很广的。它们支持三种影响到文本显示效果的属性，即标签（tag）、书签（mark）和嵌入窗口（embedded window）。

- 标签允许文本的不同部分被显示为不同的字体、颜色和凹凸效果。标签还可以与Tcl命令相关联，使之能够对用户的动作产生反应。
- 书签用来记录文本中各种有趣的位置。
- 嵌入窗口用来在文本中的特定地点插入素材（即子窗口）。文本素材里可以有任意个数的嵌入窗口；而文本中所有的嵌入窗口都必须以该文本素材做为它们的父素材。

我们来看一个文本素材部分特性的演示程序。

> Tk文本素材的功能是很强大的，我们只要稍做修改就可以把它们用做HTML素材。著名的使用手册页查看器TkMan软件就使用了Tk的文本素材，它能把普通的UNIX使用手册页显示为超链接的形式。

动手试试：文本操作

1) 首先，我们创建一个垂直卷屏条并把它连接到文本素材上去。然后把它们并排打包在一起，并让文本窗口扩展到充满整个可用窗口空间。

即使调整了窗口的尺寸，我们还是要让文本窗口继续充满着整个窗口。我们是这样告诉打包器的：如果垂直方向上还有多余的空间，就扩展这两种素材，让它们占据那部分空间。但如果在水平方向上有多余的空间，就只扩展文本素材。

```
# ! /usr/bin/wish -f

scrollbar .y -command ".t yview"
text .t -wrap word -width 80 -spacing1 1m -spacing2 0.5m -spacing3 1m \
        -height 25 -yscrollcommand ".y set"

pack .t -side left -fill both -expand yes
pack .y -side left -fill y
```

2) 接着，我们来创建嵌入窗口。我们不必操心它们的管理，因为文本素材会安排好这些事情。

```
set image [image create photo -file mickey.gif -width 200 -height 200]
label .t.l -image $image
button .t.b -text "Hello World!" -command "puts Hi"
```

3) 然后我们配置出所有将要关联到文本窗口的标签，如下所示：

```
.t tag configure bold -font -*-Courier-Bold-O-Normal--*-120-*-*-*.*-*-*
.t tag configure yellowBg -background yellow
.t tag configure blueFg -foreground blue
.t tag configure yellowBgBlueFg -background yellow -foreground red
.t tag configure underline -underline 1
.t tag configure raised -relief raised -borderwidth 2
.t tag configure sunken -relief sunken -borderwidth 2
.t tag configure center -justify center
.t tag configure left  -justify left
.t tag configure right -justify right
.t tag configure super -offset 4p
```

```
.t tag configure sub -offset -2p
.t tag bind colorOnEnter <Any-Enter> ".t tag configure colorOnEnter \
                                      -background yellow"
.t tag bind colorOnEnter <Any-Leave> ".t tag configure colorOnEnter \
                                      -background {}"
```

4) 配置好标签之后，我们按它们的要求插入文本。虽然图像不是我们设计的，可文字部分确实都是我们的工作成果。

```
.t insert end "Tk text widget is so versatile that it can support many \
               display styles:\n"
.t insert end "Background: " bold
.t insert end " You can change the "
.t insert end "background"  yellowBg
.t insert end " or "
.t insert end "foreground" blueFg
.t insert end " or "
.t insert end "both" yellowBgBlueFg
.t insert end "\nUnderlining. " bold
.t insert end "You can "
.t insert end "underline" underline
.t insert end "\n3-D effects: " bold
.t insert end "You can make the text appear "
.t insert end "raised" raised
.t insert end " or "
.t insert end "sunken" sunken
.t insert end " Text"
.t insert end "\nJustification" bold
.t insert end "\nright justification" right
.t insert end "\n center justification " center
.t insert end "\nleft justification " left
.t insert end "\nSuper and Subscripts: " bold
.t insert end "Text can be "
.t insert end "super" super
.t insert end " or "
.t insert end "sub" sub
.t insert end " scripted"
.t insert end "\nBindings: " bold
.t insert end "Text can be made to react to the user interactions" colorOnEnter
.t insert end "\nEmbedded Windows: " bold
.t insert end "You can insert labels "
.t window create end -window .t.l
.t insert end " or any kind of windows "
.t window create end -window .t.b
```

运行这个程序的时候，我们将看到如图16-9所示的窗口。

图 16-9

操作注释：

在这个例子里创建出来的第一个文本窗口和那个卷屏条勾画出基本的操作界面。然后依次创建出所有的内部窗口（.t.l和.t.b），但并没有立刻安排它们显示在窗口里，因为我们将把它们插入到文本素材里。接着，我们用各种配置选项把将要用到的绑定标签配置好。请看下面这个配置命令：

```
.t tag configure bold -font -*-Courier-Bold-O-Normal--*-120-*-*-*-*-*-*
```

它将创建出一个名为bold的标签。带这个标签的字符将以字体“-*-Courier-Bold-O-Normal--*-120-*-*-*-*-*-*”的形式被插入到文本素材里去。我们一会儿再介绍Tk对字体的管理。类似地，带blueFg标签的字符在插入时将被显示为蓝色。

> 其实我们并不需要明确地创建出文本的标签。当我们插入一段带food标签的文本时，foo将被自动创建出来。我们在这里采用的做法是预先配置好这些标签，而程序也就在我们进行配置时创建出它们来。

接下来，我们按下面的命令格式插入带标签的文本：

```
text_widget insert index chars taglist chars taglist...
```

index是一个用来指明文本中某个特定位置的字符串（类似于数组的下标），比如字符的插入位置，或者某个字符范围的首尾位置等。这类文本“下标”的语法是：

```
base modifier modifier modifier . . .
```

base给出了起始点位置，各个modifier对下标进行相对于起始点的移位或调整。modifier可以从起始点开始朝两个方向移动下标。

下标的起始点base必须是表16-4中的某一项：

表 16-4

Inde x Base	说 明
line.char	代表第line行的第char个字符
@x,y	代表文本窗口里的一个字符，它正好盖住坐标值为x和y的那个像素
mark	代表紧跟在书签mark后面的那个字符
tag.first	代表文本里带tag标签的第一个字符。tag.last代表类似的最后一个字符
pathname	代表路径名为pathname的嵌入窗口的位置

modifier可以是表16-5中的格式：

表 16-5

modifier	说 明
+count chars	把下标向前调整count个字符
-count chars	把下标向后调整count个字符
+count lines	把下标向前调整count行

（续）

modifier	说 明
-count lines	把下标向后调整count行
Linestart	调整下标指向这一行的第一个字符
Lineend	调整下标指向这一行的最后一个字符
Wordstart	调整下标指向包含当前下标的那个单词的第一个字符
Wordend	调整下标指向包含当前下标的那个单词的最后一个字符

我们可以给某个特定的文本段加上一个以上的标签。比如说，文本可以同时加黑和倾斜，成为加黑的斜体字。在插入文本的时候，必须指定它们的插入位置。在我们刚才的例子里，“end”意味着“把文本插入前面显示过的最后一个字符之后”。在文本里，下标也可以加有标签和书签，所以下面这个文本命令：

```
.t insert end "right justification" right
```

将把文字“right justification”插入到文本素材里所有文字之后，并且居右摆放。

文本素材支持大量的特性，我们建议读者认真阅读由man命令给出的关于文本的使用手册页，并且仔细学习Tk发行版本里自带的文本演示程序。在结束我们对文本的介绍之前，可以想象一下用Motif或Xlib来实现刚才那个例子会是什么情况。如果用的是Motif，它可能会有好几百行；而如果用的是Xlib，可能就是几千行了。Tk的作用就是这么巨大！

11. 画布

我们用Tk的画布（Canvas）素材实现结构化的图像。画布能够显示任意个数的图形元素，包括矩形、圆形、直线、文本和嵌入窗口，这些元素可以被操作（移动或着色）和响应用户的动作。比如说，我们可以让某个特定的图形元素在用户在它上面点击鼠标按钮时改变自己的背景颜色。

在开始学习画布素材之前，我们先来介绍一些理论知识，先来看看图形元素的标识代码（identifier）和图签（tag）概念。

当我们在画布里创建出一个图形元素的时候，它会分配到一个独一无二的整数标识代码。图形元素可以有任意个数的图签关联在它自己身上。图签是一串字符，可以是整数以外的任何形式。图签被用于图形元素的归组、标识和操作目的。同一个图签可以关联许多个图形元素，把它们归组为一个类别。画布中的每个图形元素都可以通过它的ID（标识代码）或与它关联的图签被标识出来。图签all隐含地关联着画布中的每一个元素。画布里的每个图形元素都有一个绘制区，鼠标光标在画布上移动时会落在某些图形元素的绘制区里（因为图形元素可以重叠，所以鼠标光标所在位置可能是几个图形元素的绘制区），图签current指的就是位于屏幕窗口画面最顶部的那个图形元素，这个图签由Tk自动管理。

当我们在画布素材命令里指定图形元素时，如果用的是一个整数，我们就认为它指的是具

有该ID值的那个图形元素；如果用的不是一个整数，我们就认为它指的是画布里其图签匹配上这个指定符的所有图形元素。在下面的例子里，我们使用的tagOrId符号既可以是一个只选定一个图形元素的ID，也可以是一个选定零个或多个图形元素的图签。

在画布上创建任何一个图形元素都必须指定它的摆放位置。图形元素的摆放位置是由一些浮点数字指定的，这些数字的前面还可以加上可选的单个字母m、c、i和p，这几个字母的含义是：

- m 表示毫米。
- c 表示厘米。
- i 表示英寸。
- p 表示点阵数。

如果坐标数字前面没有这几个字母，程序就默认该数值是一个点阵数。现在向大家介绍几个画布命令，看它们都是干什么用的。在下面这些命令里，pathName标识符指的是画布的路径名。第一个命令：

```
pathName create arc x1 y1 x2 y2  ?option value option value ...?
```

它的作用是在画布上创建一个arc图形元素（弧线），弧线是由一个椭圆定义的，而数字x1 y1 x2 y2给出了这个椭圆外接矩形区域的坐标。上面这条命令的选项包括-extent、-fill和-outline等。请看下面例子：

```
% set k [.c create arc 10 10 50 50 -fill red -outline blue -tags redArc]
```

这条命令在画布.c里创建了一个arc图形元素，$k给出了它的ID值。它的边界线是蓝色的。这条弧线被包围在一个矩形区域里，矩形的画布坐标是10 10 50 50，并且被填充为红色。这条弧线还关联有一个redArc图签。

```
pathName itemconfigure tagOrId ?option value option value ...?
```

这个命令的作用与-configure素材命令很相似，两者的区别在于前者被用来修改只与图形元素有关的选项，而不是修改整个画布素材；被修改图形元素是由tagOrId指定的。请看下面的例子：

```
% .c itemconfigure redArc -fill yellow
```

它把所有与图签redArc关联着的图形元素的fill（填充）颜色改为黄色。

```
pathName type tagOrId
```

这条命令的作用是返回由tagOrId指定的图形元素名单里的第一个元素的类型。比如说：

```
% .c type redArc
arc
```

```
pathName bind tagOrId ?sequence? ?command?
```

这个命令的作用与bind命令很相似，但它不是对整个画布进行操作，而是只对由tagOrId指定的图形元素进行处理。如果没有给出command参数，它将返回与画布图形元素tagOrId的绑定序列sequence关联着的所有命令。如果既没有给出sequence参数也没有给出command参数，就将

返回所有与该图形元素绑定着的序列。请看：

```
% .c bind $k <Enter> ".c itemconfigure redArc -fill blue"
% .c bind redArc  <Leave> ".c itemconfigure redArc -fill red"
```

第一个绑定操作将把那个与“tagOrId $k”关联着的图形元素在鼠标进入它的时候填充为蓝色。第二个绑定操作将把那些与redArc关联着的图形元素在鼠标离开它们的时候都填充为红色。

文本和画布支持的命令实在是太多了，要想只靠这一章把它们全部论述到肯定是不行的。我们强烈建议大家仔细阅读canvas和text的man手册页，那里有大量的经典示例可以帮助大家更好地掌握这两种素材。下面这个小程序只演示了很少的几个功能。

动手试试：画布上的文字

1) 先创建画布和在它上面显示的几个对象：一把茶壶的图像、茶壶图像上的一行文字、另外一个文本对象——它提示用户可以移动图形元素、还有一个矩形。我们对这个画布进行打包，使它充满整个窗口。

```
#!/usr/bin/wish -f

set c [canvas .c -width 300 -height 300 -relief sunken -bd 2]

set image [image create photo -file teapot.ppm -width 200 -height 200]
$c create image 150 150 -anchor center -image $image -tags item

$c create text 150 150 -text " Image Object"  -fill white

$c create text 10 10 -text " Move any Item \n using Mouse " -justify center \
                    -anchor nw -tags item  -fill red
$c create rectangle 200 10  250 40 -fill yellow -outline blue -tags item

pack .c
```

2) 接着，我们绑定这个画布，这样我们就能够对显示在它上面的图形元素进行操作了。我们将在下一步定义itemDragStart和dragItem过程。

```
bind $c <1> "itemDragStart $c %x %y"
bind $c <B1-Motion> "dragItem $c %x %y"
```

3) 为了过程的正确执行，我们需要定义两个全局变量lastX和lastY。

```
global lastX lastY

# event handler for the <1> event
proc itemDragStart {c x y} {
    global lastX lastY
    set lastX [$c canvasx $x]
    set lastY [$c canvasy $y]
}
# event handler for the <B1-Motion> event
proc dragItem {c x y} {
    global lastX lastY
    set x [$c canvasx $x]
    set y [$c canvasy $y]
    $c move current [expr $x-$lastX] [expr $y-$lastY]
    set lastX $x
    set lastY $y
}
```

这个程序产生如图16-10所示的窗口输出。

图 16-10

操作注释：

这只是一个很简单的例子。我们创建了几个画布图形元素，绑定了鼠标按键，使用户能够用鼠标移动它们。比如下面这一行：

```
$c create image 150 150 -anchor center -image $image -tags item
```

在画布上面从画布坐标(150, 150)开始创建了一个图像。这个图像本身就是一个对象，所以我们可以把事件处理器绑定到与这个图形元素关联着的图签上，这样就能够移动它并让它对用户的操作动作做出反应。类似于文本，画布支持的功能也相当的多，所以在一个只有20行左右的程序示例里是很难把它们都说清楚的。我们将在最后的软件示例中看到更多的画布功能。

在我们结束对画布的学习之前，我们要向大家说明一下它们的几个特性：

- 画布元素可以连接上事件处理器。
- 一个图形元素可以关联有多个图签，但只能有一个独一无二的ID。
- 如果某个图形元素是一个素材，它必须是包含它的那个画布的子素材。
- 如果图形元素是素材，你就能够对它们进行配置，就像它们是在画布以外那样。嵌入在画布里并不影响它们原有的方法。
- 在把图形元素摆放到画布上去的时候，你可以把它们重叠起来，遮住放在下面的图形元素。用画布的raise和lower命令可以改变它们的重叠次序。

最后，为了使刚才的那个例子简单化，我们不是通过图签而是直接通过画布把过程绑定到

对象上的。请仔细看看dragItem过程，语句：

```
set x [$c canvasx $x]
```

把x的值从实际屏幕坐标x设置为canvasx坐标x。而语句：

```
$c move current [expr $x-$lastX] [expr $y-$lastY]
```

把鼠标光标下的当前对象（用下标current表示）移动到一个新的地点，从lastX移动到x。当用户事件处理器itemDragStart被调用时，先要保存lastX的值；这个过程的绑定操作是由下面这条语句完成的：

```
bind $c <1> "itemDragStart $c %x  %y "
```

这个绑定操作的含义是当用户在画布上点击鼠标按钮1时，就以canvasx、%x（鼠标点击位置的x值）和%y（鼠标点击位置的y值）为参数调用itemDragStart事件处理器。我们在本章后面的内容里还要对绑定操作进行论述。

12. 图像

Tk能够显示两种内建类型的图像（Image）：照片（photo）和位图（bitmap）。照片类型可以用来显示gif和ppm/pgm文件，而位图格式能够显示xbm文件。image命令可以用来创建图像。image命令的通用格式如下所示：

```
image option ?arg arg . . . ?
```

其中option参数可以用来创建、删除、设置各种选项，比如高度、名称、图像类型等等。为了探究image命令，我们将以Tk发行版本自带的滑块示例程序为基础写个程序，用image命令让老程序焕发出新面貌。

动手试试：图像处理

1) 先要创建图像素材。然后对准备容纳拼图块的框素材进行配置。再把框素材打包，安排好基本窗口。

```
#!/usr/bin/wish -f

set image [image create photo -file mickey.gif -width 160 -height 160]

frame .frame -width 120 -height 120 -borderwidth 2 -relief sunken \
-bg grey
pack .frame -side top -pady 1c -padx 1c
```

2) 现在来创建各拼图块。这需要代码循环执行15次，代码的作用是把原始图像分割为适合按钮尺寸大小的图块。

```
set order {3 1 6 2 5 7 15 13 4 11 8 9 14 10 12}
for {set i 0} {$i < 15} {set i [expr $i+1]} {
    set num [lindex $order $i]
    set xpos($num) [expr ($i%4)*.25]
    set ypos($num) [expr ($i/4)*.25]

    set x [expr $i%4]
    set y [expr $i/4]

    set butImage [image create photo image-$(num) -width 40 -height 40]
    $butImage copy  $image  -from [expr round($x*40)] \
                                  [expr round($y*40)] \
```

```
                                    [expr round($x*40+40)] \
                                    [expr round($y*40+40)]
        button .frame.$num -relief raised -image $butImage \
                           -command "puzzleSwitch  $num" \
                           -highlightthickness 0
        place .frame.$num -relx $xpos($num) -rely $ypos($num) \
                          -relwidth .25 -relheight .25
    }
```

3) 最后，我们让事件处理器去处理用户的输入。两个全局变量被设置为使拼图最初的空地位于右下角。

```
set xpos(space) .75
set ypos(space) .75

proc puzzleSwitch { num} {
    global xpos ypos
    if {(($ypos($num) >= ($ypos(space) - .01))
           && ($ypos($num) <= ($ypos(space) + .01))
           && ($xpos($num) >= ($xpos(space) - .26))
           && ($xpos($num) <= ($xpos(space) + .26)))
           || (($xpos($num) >= ($xpos(space) - .01))
           && ($xpos($num) <= ($xpos(space) + .01))
           && ($ypos($num) >= ($ypos(space) - .26))
           && ($ypos($num) <= ($ypos(space) + .26)))) {
        set tmp $xpos(space)
        set xpos(space) $xpos($num)
        set xpos($num) $tmp
        set tmp $ypos(space)
        set ypos(space) $ypos($num)
        set ypos($num) $tmp
        place .frame.$num -relx $xpos($num) -rely $ypos($num)
    }
}
```

当你运行这个程序的时候，你会看到一个由15块图像组成的拼图，如图16-11所示。

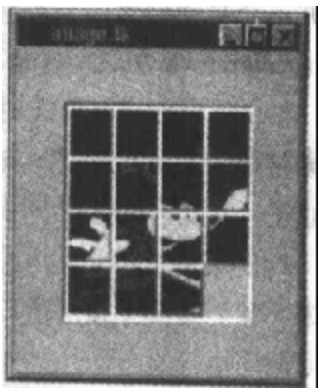


图 16-11

操作注释：

这个程序的第一行创建了一个照片图像，图像的原文件是mickey.gif，它被赋值给变量image。这个图像被分割为十五个小图块，这些图块被for循环中的“set butImage”和其他语句拷贝到十五个按钮上去。最终的结果是十五个按钮与十五个关联图像一一对应，十五个关联图像是从原来的$image大图像切割下来的。程序的其余部分主要是一个事件处理器，它负责在用户点击小图块时安排它们的动作。我们将在学习几何尺寸管理时再回过头来讨论这一部分。

puzzleSwitch算法逻辑的要点基于这样一个的事实：当用户点击一个按钮的时候，如果按钮旁边有一个空地，按钮和空地就会互相交换位置。如果你玩过这个15块的拼图游戏，就会注意到能够占据空地的图块需要遵守以下规则之一：

- 它与空地的x位置相同，而y位置与空地相差（上或下）0.25个单位（即该图块必须与空地在同一列上，并且或上或下地紧挨着空地）。
- 它与空地的y位置相同，而x位置与空地相差（左或右）0.25个单位（即该图块必须与空地在同一行上，并且或左或右地紧挨着空地）。

算法利用这些规则判断决定是否交换图块和空地。

按钮和标题签支持图像做为它们上面的图案。当然这些图案也可以嵌入到画布和文本素材里去。Tk对image命令的详细支持情况请参考bitmap、photo和image等的man使用手册页。

13. 菜单

从传统上讲，应用程序通过菜单（Menu）向用户提供一组选择，并且这样做不会使应用程序的外观形象产生很大的改变。用户不必离开主窗口就可以通过菜单方便地访问到应用程序的各种功能。Tk的menu命令将创建出这样一个素材：它在一个独立的顶层（toplevel）窗口里显示一组数据项。菜单不是一个容器型素材，它只是一个嵌入了多个对象的单个素材而已。

菜单可以嵌入三种数据项，它们是：

- 命令项，可以执行命令。
- 单选项，从多个选择里选取一个。
- 选择项，从一组选项里选取一个或多个选择。

菜单还可以通过层叠项递归地包含其他菜单。

菜单项可以显示为多达三个独立的区域。其中之一是一个标题签（以文本的形式出现）、一个位图或一个图像，与这三种东西对应的分别是-label、-bitmap和-image选项。第二个区域是使用-accelerator选项设置的快捷键序列，它应该出现在标题签的旁边。accelerator描述了一个用来调用与某个菜单项关联着的特定数据项的键盘按键序列。第三个选项是一个指示标志，告诉用户这个菜单项是单选项还是选择项，它出现在标题签的左边。注意Tk在指定了-accelerator选项时并不会自动创建一个按键绑定，该绑定必须用bind命令明确地设置之后才能起作用——设置了-accelerator选项只能在菜单里显示那个按键组合而已。

菜单项可以用多种选项进行配置，比如前景和背景颜色、字体等，这些配置工作是通过菜单素材命令的entryconfigure选项完成的。菜单项还可以用-state选项禁止掉，如果一个菜单项被禁止了，它就不再会响应用户的操作动作。

Tk的菜单是非常灵活的。如果你使用了菜单的-tearoff选项，就可以把该菜单从菜单条上拖下来，象对待一个顶层窗口那样使用它。你还可以设定菜单在被张贴和从菜单条上被拖下来时将要调用执行的命令。

菜单项可以根据它们在菜单里的位置来进行索引，也可以通过它们的标题签或“last”和“end”标签来指定。菜单可以在程序里通过post和unpost命令被张贴或关闭。Tk文档用“张贴”这个词来描述菜单被显示在窗口里的动作，因为它认为“下拉”或“弹出”只限于描述特定机器上特定菜单的动作，而张贴则是一个更具普遍意义的术语。

Tk中的每一个顶层素材只能有一个菜单素材做为该窗口的缺省菜单条。菜单条是并排摆放在一个框素材里的一组菜单。菜单条可以通过与顶层素材相关联的-menu素材选项连接到那个顶层素材里去。

Tk库里有一个<<MenuSelect>>虚拟事件，当一个菜单或它的某个菜单项变为活跃状态的时候，就会触发这个虚拟事件。菜单命令有许多的选项，要想看到它完整的选项清单，请查阅menu命令的使用手册页。

菜单系统在Tk 4.0版里进行了彻底的改进，到了Tk 8.0版又增加了许多改进。在Tk 8.0之前，用户必须使用下面这样的函数来创建菜单条：

```
tk_menuBar frame ?menu menu . . .
tk_bindFortTraversal arg arg . . .
```

人们不喜欢这些函数，在4.0以上版本的Tk里它们已经不起作用了。

动手试试：菜单

下面这个程序演示了menu命令的大部分功能。这个示例需要用到文本素材的功能。我们将通过菜单来改变背景颜色和文本显示字体的特性。我们还要创建一个把位图图像插入到文本素材里去的菜单。

1) 首先，我们来创建主窗口里的部件，包括一个文本素材、一个与文本素材关联着的卷屏条、一个显示菜单选择情况及错误信息等状态的标题签素材。我们通过grid命令把这些素材在屏幕上安排好。我们还创建了一个名为myfont的新字体。我们将使用菜单来设置字体的属性，使文本素材里的文字改变它们的面貌。

```
wm title . "Menu demonstration"
wm iconname . "Menu demo"

# create the basic UI
scrollbar .yscroll -orient vertical -command ".text yview"
font create myfont -family Courier -size 10 -weight bold -slant italic \
    -underline 1
text .text -height 10 -width 40 -bg white -yscrollcommand ".yscroll set" -font myfont
label .msg -relief sunken -bd 2 -textvariable message -anchor w -font "Helvetica 10"
.text insert end "Menu Demonstration!"

# manage the widgets using the grid geometry manager.
grid .text -row 0  column 0  -sticky "news"
grid .yscroll -row 0 -column 1 -sticky "ns"
grid .msg -row 1 -columnspan 2 -sticky "ew"

grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1
```

2) 接着，我们来编写调用函数，它们将关联到相应的菜单项上去。SetBg过程将改变文本的背景颜色。ConfigureFont将改变myFont的属性。InsertImage将把一个有名字的位图插入到文本缓冲区里去。InsertImage过程有一个副作用：如果那个有名字的位图已经存在于文本缓冲区里，它就会被删除，然后再插入一个新的位图。OpenFile过程提示用户选择一个文件，如果用户选择了一个文件，它的内容就会显示在文本素材里。

```
# procedure to set text background color
proc SetBg {} {
    global background
    .text configure -bg $background
}

# procedure to configure the previously created font.

proc ConfigureFont {} {
    global bold italic underline
    expr {$bold ? [set weight bold]: [set weight normal]}
```

```
    expr {$italic? [set slant italic]: [set slant roman]}
    expr {$underline? [set underline 1]: [set underline 0]}
    font configure myfont -weight $weight -slant $slant -underline $underline
}

# Procedure to insert images in the text widget

proc InsertImage {image} {
    catch {destroy .text.$image}
    label .text.$image -bitmap $image
    .text window create end -window .text.$image
}

# Callback for open menubutton

proc OpenFile {} {
    global message
    set file [tk_getOpenFile]
    if {$file == ""}  {
        set message "No file selected..."
        return;
    }
    .text delete 0.0 end
    set fd [open $file "r"]
    while {![eof $fd] != 1} {
        gets $fd line
        .text insert end $line
        puts $line
        update idletasks
    }
    close $fd
}
```

3) 接下来，我们把注意力集中到菜单素材和它的组成部件上。我们先创建一个将用做顶层窗口菜单条的菜单。

```
# create toplevel menu

menu .menu -tearoff 0  -type menubar

# Create File menu

set m .menu.file
```

4) 我们将增加一个File子菜单，里面的菜单项有“open”和“exit”。“open”项用一个打开文件对话框提示用户选择文件。如果用户选择了一个文件，就将通过OpenFile过程把那个文件显示在文本素材里。“exit”菜单项用来退出这个应用程序。正如大家看到的，Tk不会因为某个菜单项使用了-accelerator配置选项而为这个菜单项创建一个缺省的全局绑定。要想让快捷键起作用，我们必须明确地创建出那个绑定来。

```
menu $m -tearoff 0
.menu add cascade -label "File" -menu $m -underline 0
set modifier Meta
$m add command -label "Open..." -accelerator $modifier+o -command "OpenFile" -
underline 0 -command OpenFile
bind . <$modifier-o> "OpenFile"
$m add separator
$m add command -label "Exit..." -accelerator $modifier+x -command "exit" -underline 0
bind . <$modifier-x> "exit"
```

5) 我们接着再给主菜单增加一个Option子菜单。这个子菜单又有Background和Font两个子菜单。Background菜单里是一组单选按钮，用来改变文本素材的背景颜色。Font菜单里是一组复

选按钮，用来设置myfont字体的属性。

```
#
# Create options menu
#
set m .menu.options
menu $m -tearoff 1
.menu add cascade -label "Options" -menu $m -underline 0
$m add cascade -label "Background" -menu .menu.options.bg -underline 0
$m add cascade -label "Font" -menu  .menu.options.font -underline 0

#
# create Radio button cascade menu
#

set m .menu.options.bg
menu $m -tearoff 0
$m add radio  -label "Red" -background red -variable background -value red \
    -command SetBg
$m add radio  -label "Yellow" -background yellow -variable background \
    -value yellow -command SetBg
$m add radio  -label "Blue" -background blue -variable background -value blue \
    -command SetBg
$m add radio  -label "White" -background white -variable background -value white \
    -command SetBg
$m invoke 3

#
# Insert option button cascade Menu
#

set m .menu.options.font
menu $m -tearoff 0
$m add check -label "Bold" -variable bold -command ConfigureFont
$m add check -label "Italic" -variable italic -command ConfigureFont
$m add check -label "underline" -variable underline -command ConfigureFont
$m invoke 3
```

从上面的代码可以看出，在标准的素材选项之外，菜单里的菜单项还可以配置为有不同的背景和前景。

6) 接下来，我们在主菜单里再增加一个把位图插入到文本素材里去的子菜单项。我们已经说过，这些位图菜单项命令有一个副作用——但只有这样才能保证每次选择这个菜单项都能把位图成功地插入到文本素材里。

```
#
# Create insert menu option
#
set m .menu.insert
menu $m -tearoff 0
.menu add cascade -label "Insert" -menu $m -underline 0
foreach i {info questhead error} {
    $m add command -bitmap $i -command "puts {You invoked the $i bitmap}"\
        -hidemargin 1 -command "InsertImage $i"
}
$m entryconfigure 2 -columnbreak 1
```

观察上面的代码段可以发现：菜单里的菜单项可以通过“entryconfigure”命令和它的“-columnbreak”选项被摆放为一个表格的样子。

7) 最后，我们把菜单连接到顶层素材上去，使它成为缺省的菜单条。我们还用到了<<MenuSelect>>虚拟事件，它会在任何菜单或它的菜单项被选取时触发。<<MenuSelect>>虚拟

事件会在那个消息标题签里显示一条消息，告诉用户哪个菜单项被选中了。

```
#
# Attach the menu to the toplevel menu
#
. configure -menu .menu


#
#  Bind  global tags
#

bind Menu <<MenuSelect>> {
    global message
    if {[catch {%W entrycget active -label} label]} {
        set label "     "
    }
    set message "You have selected $label..."
}
```

8) 当你用下面的命令运行这个程序示例的时候，

```
$ wish menu.tcl
```

你将看到如图16-12所示的窗口。

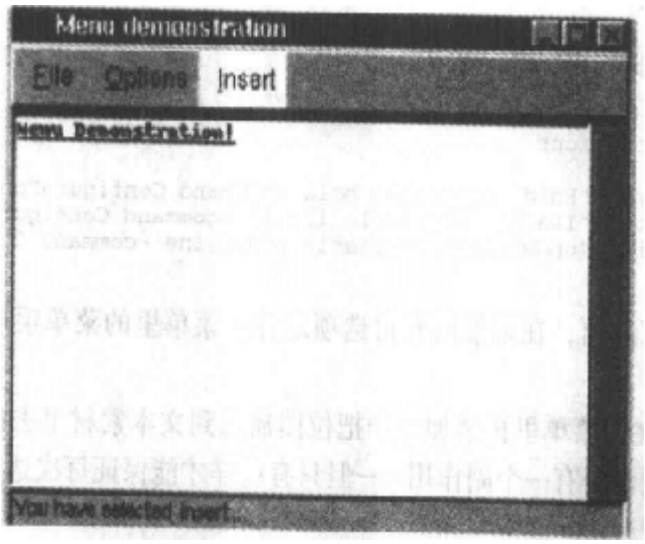


图 16-12

14. 菜单按钮

菜单按钮（MenuButton）和带关联菜单的下压按钮其作用效果是一样的。它们能够通过一个按钮提供一组选择项，而这在图形化用户操作界面里是很常见也很有用的。但菜单按钮和菜单条又是有区别的，后者通常关联着不止一个的下级菜单，而前者只关联有一个菜单。一个顶层窗口只能关联有一个菜单条，可菜单按钮却可以嵌入在用户操作界面的任何位置。

菜单按钮也可以组织起来构成菜单条，这个菜单条可以放在用户操作界面里的任何位置（与顶层菜单不同）。你可以用它们在自己的用户操作界面里创建出工具条或类似的东西来。通常，如果设置了-indicator选项，菜单按钮的行为也可以配置得象是一个选项菜单。菜单按钮的命令语法如下所示：

```
menubutton pathName ?options?
```

动手试试：菜单按钮

为了演示菜单按钮的功能，我们将编写一个简单的应用程序，它在画布上以指定的填充色绘制圆形和正方形。我们先来编写一个能够在画布上绘制圆形和正方形的功能过程。它使用了三个全局变量：x、y、sqsize。坐标(x, y)用来指定需要在画布上的什么地方绘制出对象来；sqsize被用做圆形的直径或正方形的边长。

1) 让我们先从画圆形和方形开始。

```
wm title . "Menubutton demonstration"
wm iconname . "Menubutton demo"
#
# Initial parameters to draw circles and squares
#
set x 50
set y 50
set sqsize 30
#
# procedure to draw canvas objects
#
proc AddObject {type} {
    global x y sqsize sqsize  fillc
    if {$type == "circle"} {
        .c create oval $x $y [expr $x+$sqsize] [expr $y+$sqsize] \
            -tags item -fill $fillc
    } elseif {$type == "square"} {
        .c create rectangle $x $y [expr $x+$sqsize] [expr $y+$sqsize] \
            -tags item -fill $fillc
    }
    incr x 10
    incr y 10
}
```

2) 接下来，我们创建一个带有一个框、一个菜单按钮和一个退出按钮的画布。框素材用来容纳菜单和退出按钮。我们打包所有这些元素，创建出用户操作主界面来。

```
# create the basic User Interface canvas, 2 menu buttons and a dismiss button
set c [canvas .c -width 200 -height 200 -bd 2 -relief ridge]

frame .f -bd 2

menubutton .f.m1 -menu .f.m1.menu -text "Draw" -relief raised -underline 0 \
    -direction left
button .f.exit -text "Dismiss" -command "exit"

# manage the widgets using the grid geometry manager.
pack .c -side top -fill both -expand yes
pack .f -side top -fill x -expand yes

pack .f.m1 .f.exit -side left  -expand 1
```

3) 最后，我们给菜单按钮加上一个菜单。我们创建一个菜单再给它加上三个菜单项：两个命令菜单项，分别绘制圆形和正方形；一个子菜单素材，用来选择填充颜色。

```
set m .f.m1.menu
menu $m -tearoff 0

$m add command -label "Circle" -command "AddObject circle" -accelerator "Meta-c"
bind . <Meta-c> "AddObject circle"
$m add command -label "Square" -command "AddObject square" -accelerator "Meta-s"
bind . <Meta-s> "AddObject square"
```

```
$m add separator
$m add cascade -label "Fill Color.." -menu .f.m1.cascade

set m  .f.m1.cascade
menu $m -tearoff 0
$m add radio  -label "Red" -background red -variable background -value red \
    -command "set fillc red"
$m add radio  -label "Yellow" -background yellow -variable background \
    -value yellow -command "set fillc yellow"
$m add radio  -label "Blue" -background blue -variable background -value blue \
    -command "set fillc blue"
$m add radio  -label "White" -background white -variable background -value white \
    -command "set fillc white"
$m invoke 1
```

从上面的代码可以看出，我们不仅可以给菜单项加上快捷键，还可以通过bind命令明确地创建出绑定来。

15. 弹出菜单

Tk还支持弹出菜单（Popup Menu）。菜单按钮和菜单条提供的都是静态菜单，弹出菜单与它们不同，它被用来提供上下文敏感的菜单系统。举个例子，假设你正在设计一个文本编辑器，当你的用户选取了一块文字并点击了鼠标右键的时候，你就可以动态地由程序生成一个菜单，菜单里是“Spell . . .”、“Format . . .”、“Copy . . .”、“Delete . . .”等菜单项。弹出菜单能够帮助完成这类的任务。弹出菜单能够把菜单关联到任何类型的素材上去，比如文本素材和画布素材等。弹出菜单上没有任何关联的菜单按钮，它们只是普通的菜单。把一个绑定关联到某个特定的素材，然后在该绑定的事件处理器里调用tk_popup命令，这样就可以动态地张贴出“普通的”弹出菜单了。因为这些菜单是动态地张贴出来的，所以弹出菜单里的菜单项可以动态地生成，只显示那些用得着的菜单项。

创建弹出菜单的常用命令格式如下所示：

```
tk_popup menu x y ?entry?
```

其中menu是将要张贴的菜单，x和y给出的是坐标，entry给出了菜单里某个菜单项的下标。摆放菜单就等于是把菜单项摆放到指定地点。我们来建立几个弹出菜单。

动手试试：弹出菜单

1) 首先，创建一个菜单，再给与该菜单关联的顶层窗口增加一个绑定。这样，当用户在这个窗口上按下鼠标的第三键时，菜单就会被张贴出来。

```
set w .menu
catch {destroy $w}
menu $w
bind . <Button-3> {
    tk_popup .menu %X %Y
}
```

2) 创建菜单项。大家可以看出，弹出菜单在功能方面与普通菜单是完全一样的。

```
# Add menu entries


$w add command -label "Print hello" \
        -command {puts stdout "Hello"} -underline 6
```

```
$w add command -label "Red" -background red

# Add a  Cascade menu

set m $w.cascade

$w add cascade -label "Cascades" -menu $m -underline 0
menu $m -tearoff 0
$m add cascade -label "Check buttons" \
        -menu $w.cascade.check -underline 0
set m $w.cascade.check
menu $m -tearoff 0
$m add check -label "Oil checked" -variable oil
$m add check -label "Transmission checked" -variable trans
$m add check -label "Brakes checked" -variable brakes
$m add check -label "Lights checked" -variable lights
$m add separator
$m invoke 1
$m invoke 3

$m add cascade -label "Radio buttons" \
        -menu $w.cascade.radio -underline 0
set m $w.cascade.radio
menu $m -tearoff 0
$m add radio -label "10 point" -variable pointSize -value 10
$m add radio -label "14 point" -variable pointSize -value 14
$m add radio -label "18 point" -variable pointSize -value 18
$m add radio -label "24 point" -variable pointSize -value 24
$m add radio -label "32 point" -variable pointSize -value 32
$m add sep
$m add radio -label "Roman" -variable style -value roman
$m add radio -label "Bold" -variable style -value bold
$m add radio -label "Italic" -variable style -value italic
$m invoke 1
$m invoke 7
```

当你通过“wish popup.tk”命令运行这个程序示例的时候，将看到如图16-13所示的窗口。

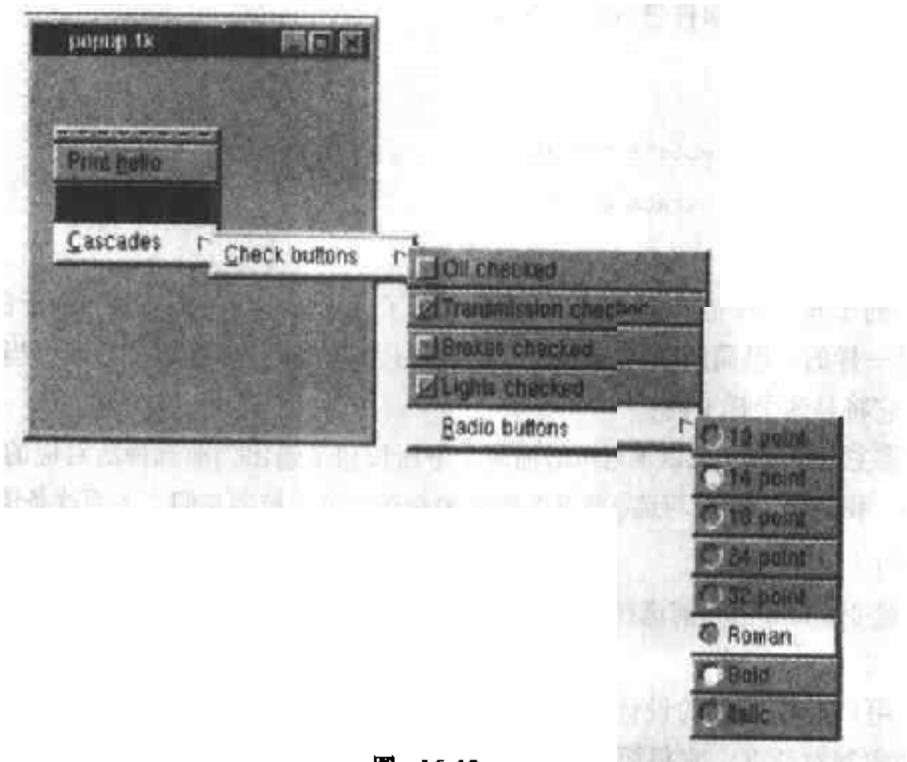


图 16-13

16. 选项菜单

Tk的选项菜单（Option Menu）完全是用Tk编写的，目的是为了模仿Motif的选项按钮。它的命令语法如下所示：

```
tk optionMenu w varName value ?value value . . . ?
```

tk_optionMenu命令创建出一个选项菜单按钮w，再给它关联上一个菜单。菜单按钮及其关联菜单加在一起使用户能够选取由value参数给定的值中的一个。当前值将被保存在全局变量varName里，用户可以通过这个变量对选项按钮进行操作和处理。调用tk_optionMenu返回的是与该选项按钮关联着的菜单。

让我们用选项按钮来重新实现前面的按钮示例程序：

动手试试：菜单选项

1) 我们把全局变量state设置为等于1，创建一个单选按钮和一个选项菜单。打包，把它们并排摆放在一起。如下所示：

```
#!/usr/bin/wish -f

set state 1

checkbutton .lan -text "Language" -command {changeState} -relief flat \
                 -variable state  -onvalue 1 -offvalue 0
set optMenu [tk_optionMenu .opt lang Tcl C Lisp C++]

pack .lan .opt -side left

# now make C As the default using lang variable
set lang C++
```

2) 下面是对应用程序事件进行处理的过程：

```
proc changeState {} {
    global state
    if $state {
        .opt config -state normal
    } else {
        .opt config -state disabled
    }
}
```

在这个例子里，我们用一个选项菜单代替了三个单选按钮。这个例子的工作情况和buttons.tk是一样的，但简洁得多了。它也减少了显示选项所需要的屏幕空间。当你运行这个程序的时候，它将是这个样子的：

如果你愿意，还可以象以前那样增加两个下压按钮来输出当前选择所对应的值。这个例子还告诉我们：我们可以通过与选项菜单关联着的全局变量来控制它们。下面这条语句：

```
set lang C++
```

通过设置变量lang把当前选择设置为“C++”。

17. 对话框

在一个用户操作界面的设计周期里会使用大量的对话框（Dialog）。Tk提供了一个名为tk_dialog的定制对话框。它很简单，但如果用得聪明，就可以用它完成大部分这方面的工作。

tk_dialog的语法如下所示：

```
tk_dialog window title text bitmap default string string . . .
```

它就创建出一个窗口标题为title的模型对话框，对话框里的消息文本是text，并且带有指定的位图bitmap。它还会创建出几个按钮，按钮的标题分别由各个string给出。当用户按下其中任何一个按钮时，tk_dialog会返回这个按钮的编号，关闭它自己。

我们先来看一个简单的例子：

```
#!/usr/bin/wish -f

wm withdraw .

set i [tk_dialog .info  "Info" "Simple Info Dialog" info  0 Ok Cancel]
if {$i==0} {
    puts "Ok Button Pressed"
} else {
    puts "Cancel Button Pressed"
}
exit
```

第一行语句的作用是不显示由wish创建的缺省顶层窗口。这是很有必要的，因为tk_dialog自己会创建出一个顶层窗口，而我们不想在这个简单的例子里弹出两个窗口来。接下来的那一行语句创建出一个模型对话框.info，它的窗口标题是“Info”，其中的消息文本是“Simple Info Dialog”，窗口内还有一个名为info的位图。另外，它还创建了两个按钮，分别是“OK”和"Cancel"，并且把编号为0的按钮（即“OK”按钮）设定为缺省的。

大家可能注意到了，我们在提到对话框的时候使用了这样一个词：模型。这是什么意思呢？它的意思是说用户的选择范围是有限的。在用户点击“OK”或“Cancel”按钮对这个对话框做出反应之前，不能在应用程序里做其他任何事情。只有在用户按下了这两个按钮之一以后，控制才会返回到应用程序，此时变量i被设置为用户按下的那个按钮的编号。

我们可以通过grab和tkwait命令实现模型交互操作的效果。关于这方面的详细资料请参考它们的使用手册页或介绍Tk的书籍。

16.5.5 Tk内建的对话框

除了提供有tk_dialog命令以外，Tk还准备了许多工具性的对话框过程。大多数基于GUI的应用程序都有许多共同的功能，比如提示用户选取输入或输出文件、选择颜色等等。Tk为这些操作都准备了相应的工具性对话框。此外，这些内建的对话框的都被编写为具有Tcl脚本运行在其上的操作系统所特有的外观和操作感觉。我们将在这一小节里学习这些工具性的对话框。这些工具性对话框不是Tk的内建命令，它们是提供了特定对话框功能的工具性脚本。

1. 颜色选择器（Color Chooser）

```
tk_chooseColor ?option value . . . ?
```

大多数基于GUI的应用程序都可以让它们的使用者通过选择颜色和字体来定制应用程序的外观和操作手感。人们在提到颜色时习惯于使用描述性的名称，可大多数图形系统在处理颜色时使用的都是RGB或CYMK等不同的彩色方案。RGB是红、绿、蓝三原色方案，系统中任何其他的颜色都是用这三种颜色的组合来代表的。

Tk对颜色的描述与HTML很相似——如果读者曾经为Web网页设计过程序，就不应该对这个词感到陌生。有许多合法有效的颜色名称与特定的颜色值相对应。但如果你想更精确地控制Tk的颜色，就必须进一步了解它内部对颜色的表示方法。

TK中的一切颜色都是用一个十六进制整数来表示的。你可以使用各种长度的十六进制数字，可以是三位数、六位数、九位数或十二位数。比如说，你可以使用#000000到#ffffff之间的六位十六进制数字。从左至右，红、绿、蓝三色的程度依次各用两位数代表。因此，#ff9900就表示红色值为#ff，绿色值为#99，而蓝色值为#00，最终得到的颜色是一种橘黄色。在三原色系统里，#000000是黑色，#ffffff是白色。象#a5a5a5这样三组数字都相同的情况将得到一种灰色。其他组合将是另外的色彩。

Tk能够接受的其他十六进制数的长度当然也要分为相同的三个部分，三组十六进制数分别代表三原色的值，与六位十六进制数的原理完全一样。

tk_chooseColor对话框为采用三原色方案选择某种颜色提供了一种简单的办法。这个对话框里还能这样使用：给出一种颜色的名称，查出与之对应的RGB值。过程tk_chooseColor会弹出一个对话框供用户选择一种颜色。表16-6是它的几个可选的命令行参数：

表 16-6

-initialcolor color	指定对话框在弹出时显示的颜色。color参数必须是一个能够被Tk_GetColor函数接受的格式，比如red或#ff0000（#ff0000是red的RGB对应值）
-parent window	让窗口window作为颜色对话框逻辑上的父窗口。颜色对话框将显示在它父窗口的上面
-title titleString	指定显示在颜色对话框里的窗口标题。如果没有给出这个选项，就会显示一个缺省的窗口标题

如果用户选定了一种颜色，tk_chooseColor将返回该颜色的名称，其格式可以直接用在Tk的素材命令里。如果用户取消了这次操作，命令就会返回一个空字符串。

下面的脚本程序演示了tk_chooseColor对话框的用法：

```
#
# tk_chooseColor demo
#

label .l -text "Set my background color:"
button .b -text "Choose Color..." -command ".l config -bg \[tk_chooseColor\]"
pack .l .b -side left -padx 10
```

运行上面这个脚本程序，当你点击“Choose Color . . .”按钮时，你将会看到如图16-14所示。

2. 打开/保存文件对话框

tk_getOpenFile和tk_getSaveFile是两个很方便的函数，它们的功能分别是提示用户选择输入或输出文件。在大多数基于GUI的操作系统里，所有的应用程序在选择输入和输出文件时提供的都是一样的对话框。Tk通过这两个方便的函数向所有基于Tk的应用程序提供了打开输入/输出文件的功能。这些对话框的外观、操作手感和操作行为保持着与应用程序同样的风格。大多数错误情况都由这些对话框自己来负责处理，程序员除了创建和初始化这些对话框以外并不需要做

太多的事情。它们提供的程序接口使开发人员能够设定过滤器，在文件清单里只列出匹配上特定模板的文件。tk_getOpenFile命令通常与File（文件）菜单里的“Open . . .”命令相关联，而tk_getSaveFile通常与“Close . . .”命令相关联。

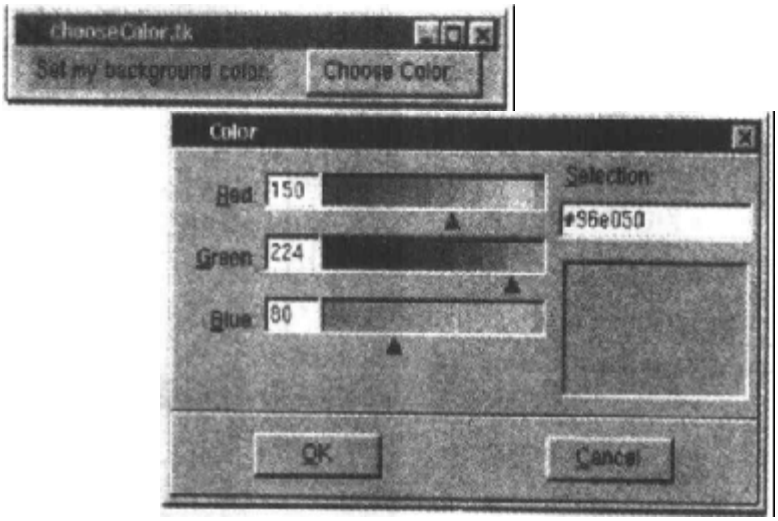


图 16-14

如果用户选择了一个已经存在的文件，对话框会要求用户确认是否需要覆盖那个现有的文件。下面就是这两个命令的语法：

```
tk_getOpenFile ?option value . . . ?
tk_getSaveFile ?option value . . . ?
```

这些命令完整的选项清单请参考tk_getOpenFile的使用手册页。

下面这个例子演示了这两个命令的用法：

```
# tk_getOpenFile demo
# tk_getSaveFile demo

label .o -text "File to open:"
entry .oe -textvariable open
set types {
    {{Text Files}       {.txt}    }
    {{TCL Scripts}      {.tcl}    }
    {{C Source Files}   {.c}      TEXT}
    {{GIF Files}        {.gif}        }
    {{GIF Files}        {}        GIFF}
    {{All Files}        *             }}

button .ob -text "Open..." -command "set open \[tk_getOpenFile -filetypes \$types \]"

label .s -text "File to save:"
entry .se -textvariable save
button .sb -text "Save..." -command "set save \[tk_getSaveFile\]"

# Create a dismiss button

button .b -text "Dismiss" -command "exit"

# Manage the widgets

grid .o -row 0 -column 0 -sticky e -padx 10
grid .oe -row 0 -column 1 -padx 10
grid .ob -row 0 -column 2 -padx 10
```

```
grid .s -row 1 -column 0 -sticky e -padx 10
grid .se -row 1 -column 1 -padx 10
grid .sb -row 1 -column 2 -padx 10
grid .b -row 2 -pady 10
```

从代码里可以看出，对这些对话框例程可以设定文件过滤器模板，从而只列出符合条件的文件清单。运行上面这个示例程序，当你点击“Open . . .”按钮的时候，你将会看到如图16-15所示。

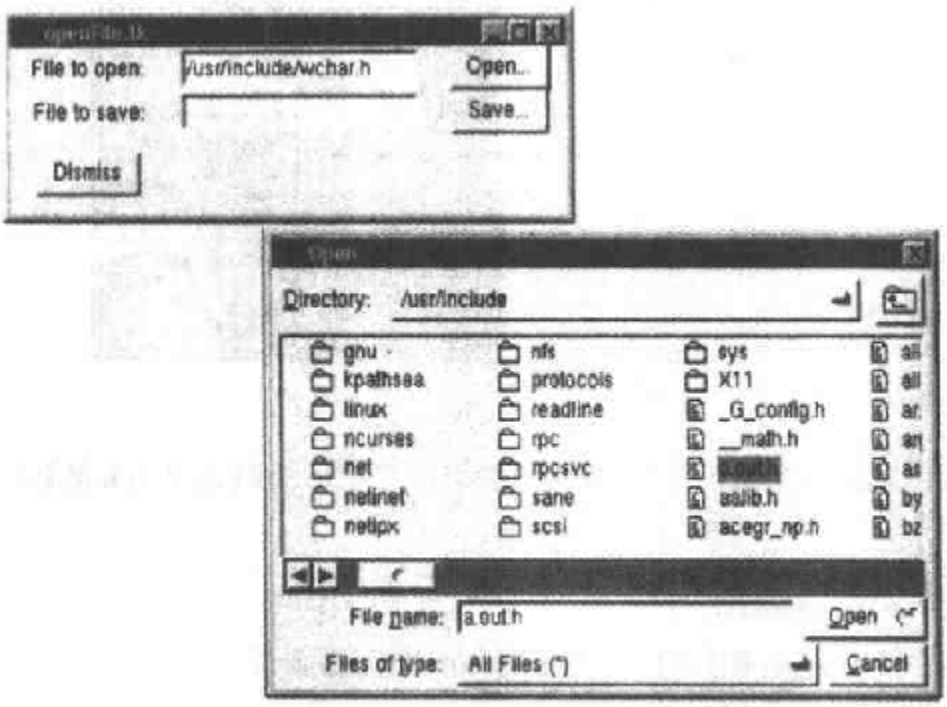


图 16-15

16.5.6 颜色方案

当你在Tk 4.0及以后的版本里使用Tk素材命令创建新素材的时候，所有素材都会有一个黑颜色的前景和一个灰颜色的背景。因此，当你编写出一个完整的应用程序时（比如前面那些例子），应用程序就将有一个黑颜色的前景和一个灰颜色的背景。那如果你想编写一个有浅蓝色背景的应用程序时应该怎么办呢？要想实现这一目的，一种办法是把创建出来的所有素材的背景都配置为浅蓝色。但因为代码大多数中的命令都是配置命令，所以这样做会使应用程序的代码大大膨胀，可读性降低，从而掩盖了应用程序的逻辑。为了解决这个问题，Tk提供了一个方便的全局性改变应用程序颜色方案的办法。下面这些个命令可以用来设置任何应用程序的全局性颜色方案：

```
tk_setPalette background
tk_setPalette name value ?name value ...?
tk_bisque
```

如果在调用tk_setPalette的时候给出了一个参数，这个参数就会被用做所有素材的缺省背景颜色，而tk_setPalette也会计算出使用这种颜色的调色板来。请看下面这些命令：

```
tk_setPalette steelblue
button .b -text "Linux is cool!"
pack .b
```

这将创建出一个金属蓝色背景的按钮。而这个应用程序里未来素材的背景色也都会被设置为金属蓝。

此外，tk_setPalette的参数还可以由任意个数的“name-value”对组成，每个名值对中的第一个参数是Tk选项数据库里某个选项的名字，而第二个参数是该选项将要使用的新设置值。现时期能够被设置的选项数据库名字如下表所示：

activeBackground	disabledForeground	foreground
highlightColor	highlightBackground	insertBackground
selectBackground background	selectColor activeForeground	selectForeground
troughColor		

请参考options(n)使用手册页中对选项数据库的介绍。tk_setPalette将为没有指定的选项计算出它们合理的缺省值来。你也可以指定没有在上表里列出来的其他选项，Tk会根据这部分选项对素材做相应的改变。

tk_bisque是为了向后兼容性而准备的一个过程：它会把应用程序的颜色恢复为Tk 3.6以前的版本里所使用的浅棕色（“冰激凌”）颜色方案。

16.5.7 字体

如果你曾经在X窗口系统上使用Xlib或Motif编写过程序，就会知道字体是X的比较模糊难缠的领域之一。字体的名字必须在“X逻辑字体描述结构”（X Logic Font Description，简称XLFD结构）里进行定义。举个例子，在使用8.0版以前的Tk编写出来的应用程序里，如果想用指定字体创建出一个按钮，就要使用下面这样的命令：

```
Button .b -text  "Hello" -font -font -*-Courier-Bold-O-Normal--*-120-*-*-*-*-*-*
```

我们在前面也曾经见过这样的写法，但现在我们将向大家介绍怎样才能绕过这种“丑陋”的格式。X之所以会被设计成这个样子，是为了保证X的客户应用程序能够适应各种版本的服务器，而这些服务器上的文件系统、命名规则、字体库等等都是千差万别的。X客户还必须有能力去动态地确定任一给定的服务器上有哪些字体是可用的，而只有这样，才能提供出用户能够看懂的信息，才能选择最佳运行路线。XLFD提供了一组适当的字体属性框架定义，比如FOUNDRY、FAMILY_NAME、WEIGH_NAME、SLANT等等。如果读者想对XLFD做进一步的学习，请在X窗口系统的文档里查阅它的技术规范，或者在你自己的Linux机器上钻研钻研xfontsel命令。

虽然XLFD的功能很强，灵活性也很好，但使用起来却既不简单也不直观。类似子颜色的情况，人们习惯于给字体起一个简单的名字，比如“Helvetica 12 point italic”等。而Tk向其他非X平台上的移植又引出另外一个复杂性问题：因为那些窗口化系统没有使用XLFD，所以它们的用户就必须强迫自己去学习XLFD。除上述问题以外，XLFD还不支持通过名称来创建字体的

做法。

Tk 8.0在字体处理方面引入了一个新机制：使用font命令。可以给字体起名字，创建新字体时可以使用人们习惯的计量单位，新字体到系统有关接口的转换工作在Tk的内部完成。新的font命令给程序员带来的好处之一是使字体的定义工作独立于计算机软硬件平台。它还提供了一个把新建字体与某个名字关联起来的办法。font命令的语法如下所示：

```
font create ?fontname? ?option value ...?
font configure fontname ?option? ?value option value ...?
```

font命令的选项有-family、-size、-slant等。font命令完整的选项清单可以在它的使用手册页里查到。

请看下面的例子，我们可以用这条命令创建出一个名为myfont的字体来：

```
font create myfont -family Courier -size 20 -weight bold -slant italic\
        -underline 1 -overstrike 1
```

在字体创建出来以后，就可以用名字代表它对应的设置值用在-font素材选项里了。在创建出字体之后，如果我们用wish来运行下面这些代码：

```
button .b -text "Hello World!" -font myfont
pack .b
```

就会看到如图16-16所示的效果。

图 16-16

16.5.8 绑定

我们曾经说过：在素材被创建出来之后，我们就可以把事件处理器与它们关联起来，使素材能够响应用户的操作动作。比如说，在前面最后一个“Hello World”程序里，我们用下面的命令把一个事件处理器关联到那个按钮上：

```
bind .b <Control-Button-1> {puts "Help !" }
```

事件处理器与素材之间所有这些联系都必须通过绑定命令bind来建立。bind命令的作用就是把Tcl脚本和X事件关联起来，它的功能是非常强大的。它的通用语法如下所示：

```
bind tag
bind tag sequence
bind tag sequence script
bind tag sequence +script
```

tag参数用来确定要把绑定施加在哪个窗口上。如果tag以一个句点开始，比如“.a.b.c”这样，就肯定是某个窗口的路径名；否则，它可以是一个任意的字符串。每个窗口都会有一组关联标签，而一个绑定将作用于一个特定的窗口上。如果tag是为窗口定义的那些标签中的一个，那么，

缺省的绑定标签将采取以下行动：

- 如果tag是某个内部窗口的名字，绑定将施加到该窗口上。
- 如果tag是某个顶层窗口的名字，绑定将施加到该顶层窗口及其全体内部窗口上。
- 如果tag是某种素材的类型名称，比如button（按钮），绑定将施加到该类型的所有素材上。
- 如果tag的值是all，绑定将施加到该应用程序的全体窗口上。

举例来说，我们来看看对按钮类型调用绑定命令会发生什么事情：

```
% bind Button
<Key-space> <ButtonRelease-1> <Button-1> <Leave> <Enter>
```

命令执行的结果告诉我们：按钮类型有与这五个事件序列对应的绑定。继续研究，看看对其中的一个绑定调用bind命令的第二种形式会发生什么事情：

```
% bind Button <Key-space>
    tkButtonInvoke %W
```

这个结果告诉我们：button（按钮）类素材上的<Key-space>绑定（一切下压按钮都属于这个button类）将以该按钮的路径名为参数调用Tcl脚本程序tkButtonInvoke。

我们还可以在绑定命令里直接使用素材的类型名称，如下所示：

```
% bind Button <Control-Button-1> { puts "Help! " }
```

这将把<Control-Button-1>绑定设置到这个应用程序里所有的按钮类素材上。允许使用多个绑定都来匹配一个X事件。如果几个绑定被关联到不同的标签，这些绑定将逐个依次执行；其缺省的执行顺序是：某个素材类上的绑定（如果有的话）最先执行，其次是该素材自身上的绑定，接着是该素材的顶层窗口上的绑定，最后是一个all绑定（如果该事件有一个与之对应的all绑定的话）。我们可以用bindtags命令改变特定窗口的绑定执行顺序，还可以再给这个窗口额外加上一些绑定标签。

当某个绑定匹配上特定的用户动作序列时，与这个绑定关联着的脚本程序就将被调用执行。在调用脚本程序的时候，我们可以让bind命令从激活该绑定的X事件那里给对应的脚本程序传递一些参数过来。这需要使用特殊的修饰符来实现。举例来说，在前面的画布例子里，我们有这样一个绑定：

```
bind $c <1> "itemDragStart $c %x %y"
```

它告诉bind要在调用itemDragStart命令时传递$c、%x和%y。bind会在调用实际发生时把%x和%y替换为那个X事件结构的x坐标和y坐标。bind命令支持大量的替换参数，它们完整的清单可以在bind命令的使用手册页里查到。

16.5.9 bindtags命令

前面已经讲过，bind命令的作用就是把一个动作和一个绑定关联起来。用bind命令创建关联关系时必须指定一个标签。tag参数用来确定要把绑定施加在哪个窗口上。正常情况下，tag可以是某个素材的名字、某个素材类型的名字、关键字all或者任何其他的文本字符串。每个窗口都

有一个标签关联表，如果某个绑定的标签是为某个窗口定义的标签中的一个，这个绑定就将作用于这个特定的窗口。当窗口里发生一个事件时，它将按一定次序顺序作用于这个窗口的每一个标签。对各个标签来说，与给定标签和发生事件最匹配的绑定将最先执行。请看下面这个代码段：

```
bind . <F1> "puts Toplevel"
entry .e
pack .e
bind .e <F1> "puts Entry"
```

在wish shell里执行了这个代码段以后，如果你在输入框素材里按下F1键，就会先看到字符串“Entry”、再看到“Toplevel”被打印出来。在素材.e上调用bindtags命令给出的结果是“.e Entry . all”；它的意思是这样的：如果在素材.e上触发了一个事件，先要检查.e素材自己的标签，然后再检查包含该输入框的那个顶层素材的标签。如果想让顶层素材上的绑定先于输入框上的被执行又该怎么办呢？这时，你就可以利用bindtags命令改变Tk默认的绑定执行顺序，如下所示：

```
Bindtags .e {. .e Entry all}
```

在缺省的情况下，每个窗口都有四个绑定标签，按默认执行顺序依次为：

- 该窗口的名字。
- 该窗口的类型名。
- 该窗口最近的顶层祖先的名字。
- all。

顶层窗口只有三个缺省的标签，这是因为它祖先的名字和窗口自己的名字是同一个。

bindtags命令可以用来给窗口任意添加绑定标签。事实上，bindtags命令的这个功能可以帮助我们完成许多事情。有了它，我们可以只创建一次绑定，却把它关联给任意多个素材；具体做法很简单：只要把这次创建的绑定标签插入到素材的绑定标签表里就行了。第二点是它使素材能够在四个标准绑定标签以外再拥有其他标签。标签使我们能够用名字而不是用一个键序列来标识一个动作。请注意下面例子里bindtags命令的实际用法。

```
set count 0
button .b -text "Tick(ms)"
label .ticker -textvariable count
pack .b .ticker

bind timer <ButtonPress-1> {
    set count 0
    StartTimer %W
}

bind timer <ButtonRelease-1> {
    StopTimer %W
}

proc StartTimer { widget } {
    global pending count
    incr count 200
    set pending [after 200 [list StartTimer $widget]]
}

proc StopTimer { widget } {
```

```
    global pending
    after cancel $pending
}

bindtags .b [linsert [bindtags .b] 0  timer]
```

在上面的例子里，我们先创建了一个简单的用户操作界面，里面有一个按钮和一个用来显示计时数字的标题签。然后用timer做标签名创建了一个绑定。我们先后给timer标签增加了两个键序列，第一个序列是<ButtonPress-1>，启动计时器；第二个是<ButtonRelease-1>，停止计时器。代码相当简单，很容易看懂。这里值得注意的就是bindtags命令的用法。我们只用一行语句就把这些键序列添加到按钮.b上了。如果没有bindtags命令，我们就不得不象下面那样做：

```
bind .b  <ButtonPress-1> "+{set count 0; StartTimer %W}"
```

对<ButtonRelease-1>绑定也得这样做。如果只是单看这些绑定的代码，外人是很难知道我们到底想做些什么。而使用了bindtags命令，我们就能够用名字timer来标识这些序列了。如果我们还要再创建一个按钮做另一个计时器，只需在那个按钮上调用binftags命令插入这个timer绑定就可以了。

16.5.10 几何尺寸管理

在创建了素材并使用bind命令绑定好事件处理器之后，我们需要把各种素材摆放到屏幕上去，使GUI既有意义又有用处。几何尺寸管理器负责完成这项工作。Tk当前支持三个几何尺寸管理器，它们是

- 打包器Packer，使用pack命令。
- 摆放器Placer，使用palce命令。
- 表格或表格线管理器，使用grid命令。

1. 打包器Packer

pack命令的作用是在主窗口或主素材里安排它所包含的附属素材们相对于主窗口边界的位置。pack命令的语法可以是下面几种形式之一：

```
pack option arg ?arg ...?
pack slave ?slave ...? ?options?
pack configure slave ?slave ...? ?options?
```

我们来看一个例子，学习几个pack选项的用法：

```
#!/usr/bin/wish -f

foreach i  {1 2 3 4 } {
    button .b$i -text "Btn $i"
    pack .b$i -side left -padx 2m -pady 1m
}
```

这段代码将产生如图16-17所示的输出。

这里，全体按钮都靠左摆放，两个按钮之间有一个2毫米的水平间距，各个按钮与主窗口边界的垂直间距是1毫米。

再看下面这个打包操作序列：

```
#!/usr/bin/wish -f

foreach i  {1 2 3 4 } {
    button .b$i -text "Btn $i"
    pack .b$i -side left -ipadx 2m -ipady 1m
}
```

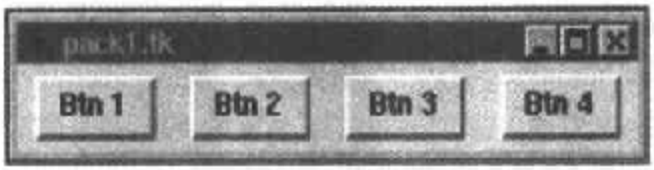


图 16-17

在这个例子里，-ipadx指示附属素材（即那些按钮）以内部排列方式摆放，彼此水平间距还是2毫米。内部排列方式的意思是：如果窗口里有空余的地方，附属素材（这里是按钮）就会自动伸展占据那些空间。

> pack命令有许许多多的组合形式。进一步的学习请参考pack命令的使用手册页和由Tk的作者Ousterhout本人编写的“Tcl and the Tk Toolkit”（《Tcl和Tk工具包》），Addison-Wesley出版社出版（国际书号ISBN-0-201-63337-X）。

2. 摆放器Placer

当你需要指定一个窗口（附属窗口）在另外一个窗口（主窗口）里固定的准确尺寸和位置时，就需要使用负责窗口固定位置摆放的Placer几何尺寸管理器。我们以前面的拼图程序为例来说明Placer几何尺寸管理器的使用方法。从下面这个代码段可以看出拼图中的按钮是怎样创建出来的。

```
set order {3 1 6 2 5 7 15 13 4 11 8 9 14 10 12}
for {set i 0} {$i < 15} {set i [expr $i+1]} {
    set num [lindex $order $i]
    set xpos($num) [expr ($i%4)*.25]
    set ypos($num) [expr ($i/4)*.25]

    set x [expr $i%4]
    set y [expr $i/4]

    set butImage [image create photo image-${num} -width 40 -height 40]
    $butImage copy  $image  -from [expr round($x*40)] \
                                  [expr round($y*40)] \
                                  [expr round($x*40+40)] \
                                  [expr round($y*40+40)]
    button .frame.$num -relief raised -image $butImage \
                       -command "puzzleSwitch  $num" \
                       -highlightthickness 0
    place .frame.$num -relx $xpos($num) -rely $ypos($num) \
                      -relwidth .25 -relheight .25
}
```

这个循环创建按钮并把它们摆放到主窗口.frame里。代码中的“-relx 0”代表的是主窗口的左边界，“- relx 1”是主窗口的右边界；“-rely 0”和“-rely 1”也与此类似。仔细研究那个循环的代码就会发现按钮们是怎样摆放才构成拼图的。

3. 表格管理器grid

grid几何尺寸管理器的作用是把（附属）素材以行列对齐的方式摆放到另外一个窗口里，后

者被称为几何主窗口。grid是一个功能非常强大的几何尺寸管理器；有了它，我们就可以轻松的创建出复杂的窗口布局。请看这个只有10行的TK代码段，创建一个用来输入个人资料的输入框还真是够简单的：

```
#!/usr/bin/wish -f

set row 0
foreach item {name email address phone} {
  label .$item-label -text "${item}:"
  entry .$item-entry  -width 20
  grid .$item-label -row $row -column 0 -sticky e
  grid .$item-entry -row $row -column 1 -columnspan 2 -sticky "ew"
  incr row
}

grid columnconfigure . 1 -weight 1
```

如果运行这个程序，我们将看到如图16-18所示的输出效果。

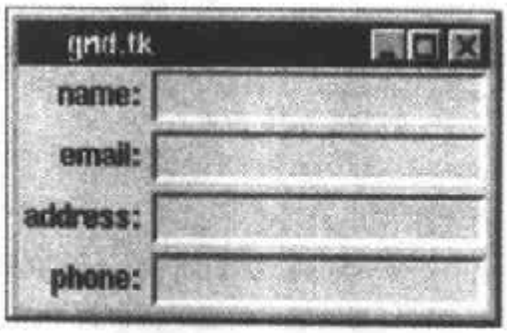


图 16-18

这段代码把附属素材“.$item-label”和“.$item-entry”按照“-row $row”和-column选项的设定摆放到了主窗口里。你还可以用-rowspan和-columnspan选项来指定附属素材的行、列的延伸选项，这些选项的作用是让附属素材延伸span行或span列，或者同时在行、列两个方向上延伸。请看下面这条命令：

```
grid .$item-entry -row $row -column 1 -columnspan 2 -sticky "ew"
```

它设定entry素材占据两列，“-sticky "ew"”表示附属素材将在分配给它的空间里从右向左展开。如果在-sticky选项里只给出了一个字母，就表示素材将固定靠左（w）或固定靠右（e）来摆放。再看下面这行代码：

```
grid columnconfigure . 1 -weight 1
```

这是代码段的最后一行代码，它设定：如果主窗口（.）在水平方向上调整了尺寸，第1列将获得尺寸调整后的部分。

> 在我们边学习边开发的程序示例里已经多次用到了grid几何尺寸管理器。它使窗口布局既容易设计又易于理解。它是从4.1版本开始出现在Tk里的。

16.5.11 焦点及其切换

如果在你的计算机屏幕上有好几个顶层窗口，那么，当你按下一个键的时候，哪个窗口会接收到这个KeyPress事件呢？答案是拥有焦点的那个窗口，即焦点决定了键盘输入的目的地。顶层窗口的焦点管理是由窗口管理器自动完成的。有的窗口管理器的做法是：只要鼠标进入了某个顶层窗口，就把输入焦点设置给它；而有的做法则是当用户点击了某个窗口之后才改变输入焦点的设置。但窗口管理器通常只负责顶层窗口的焦点设置工作，顶层窗口的子窗口中间的

焦点设置工作一般都要由应用程序自己负责。

Tk在应用程序级准备了两个焦点模型。一个是implicit暗焦模型，它把焦点设置给鼠标当前位置上的素材；另一个是explicit明焦模型，用户必须明确地点击了素材或明确地通过键盘切换到素材才能使之得到焦点。在缺省的情况下，在明焦模型里的素材之间切换焦点需要使用Tab键。

Tk会记住每个顶层窗口的焦点窗口是哪一个（该顶层窗口最直接的后代将接收到焦点）；当窗口管理器把焦点分配给某个顶层窗口时，Tk会自动把它设置到自己记住的窗口去。在缺省的情况下，Tk在一个顶层窗口的内部使用的是明焦模型：在一个顶层窗口内部移动鼠标一般不会改变焦点；只有在某个素材明确地申请了焦点（因为有一个按钮被按下）或者用户敲了一个键（比如Tab键）来移动焦点时，焦点才会发生变化。

focus焦点命令的语法如下所示：

```
focus
focus window
focus option ?arg arg ...?
```

focus命令使用方法的详细资料请参考它的使用手册页。

当应用程序或者它的某个顶层窗口得到焦点之后，我们就可以通过调用一个Tcl过程tk_focusFollowsMouse创建出一个暗焦模型来。它会重新配置Tk，然后只要鼠标进入了某个窗口，焦点就会被设置给它。请看下面的例子，它会在应用程序顶层窗口得到了焦点的大前提下指示窗口管理器把焦点设置给鼠标位于其上的任何成分。如果你运行这个例子，就会看到：鼠标只要一掠过按钮，就会获得焦点，根本不需要点击鼠标按键了。

```
tk_focusFollowsMouse

button .b1  -text "Button 1"
button .b2  -text "button 2"
button .b3  -text "button 3"

pack .b1 .b2 .b3 -side left -padx 10
```

Tcl过程tk_focusNext和tk_focusPrev实现了同一顶层窗口的子窗口之间焦点顺序切换的操作；前者的缺省绑定是Tab键，后者的缺省绑定是“Shift-Tab”组合键。

tk_focusNext和tk_focusPrev命令的语法如下所示：

```
tk_focusNext window
tk_focusPrev window
```

tk_focusNext是一个用在键盘操作方面的工具性过程，它返回一个窗口，这个窗口在焦点先后顺序里是排在window窗口后面的“下一个”窗口。焦点顺序是由窗口重叠的先后次序和窗口的层次结构决定的。如果是在同一层次结构上，焦点顺序和窗口的重叠顺序就是一致的，即最下面的窗口将第一个得到焦点。如果一个窗口有子窗口，焦点会先到达这个窗口，再依次到达它的各子窗口（递归性的），然后前往与它同层次的下一个窗口。焦点不会进入window自己的顶层窗口以外的其他顶层窗口里——它们都会被隔过去，也就是说：过程tk_focusNext永远不会返回它自己的顶层窗口以外的另一个顶层窗口里的窗口。

在计算出下一个窗口之后，tk_focusNext会检查该窗口的-takefocus选项，看它是否应该被隔

过去。如果是要隔过去，tk_focusNext就会检查焦点顺序里的下一个窗口，这样一直下去，直到它最终找到一个能够接受焦点的窗口或转一圈回到自己为止。

tk_focusPrev命令和tk_focusNext基本一样，差别只是前者返回的是焦点顺序里本窗口前面的那个窗口而已。

请看下面的例子，注意按钮素材.b2是如何通过设定“-takefocus 0”选项躲开焦点的。当我们运行这个示例程序的时候，“skip focus”按钮完全不接受焦点，即使鼠标光标放在它上面也是如此。

```
tk_focusFollowsMouse

button .b1  -text "Button 1"
button .b2  -text "skip focus" -takefocus 0
button .b3  -text "button 3"

pack .b1 .b2 .b3 -side left -padx 10
```

16.5.12 选项数据库

和Motif和Xt的情况一样，Tk中的每个素材都可以归入一个类别（calss），这个类别可以通过下面这个命令检索出来：

```
winfo class widget-path-name
```

这些类别名被用来设定应用程序的缺省设置和对一整个素材类别的绑定。Tk使用一个特殊的数据库——它的名字是选项数据库（option database）——来保存和检索应用程序的资源。比如说，在本章最开始的第一个程序里我们用了这样一条语句：

```
option add *b.activeForeground brown
```

这就等于是在通知选项数据库：名字是.b的按钮（它可以有任意的父素材）必须有一个棕色的活跃前景颜色。我们可以对整个按钮类别进行设置来获得同样的效果：

```
option add *Button.activeForeground brown
```

当Tk创建一个素材的时候，在设置好命令行之后，它会去检索选项数据库，根据数据库中的设置值对相应的资源进行设置。如果它找到了该资源的一个匹配项，就使用匹配到的选项；否则，它将使用一个缺省值。

option命令和X资源文件的作用是一样的。事实上，我们可以把应用程序将要用到的全部资源都保存到一个文件里，再用option命令读这个文件，就象hello4.tk程序示例中那样。我们还可以使用option命令来查询保存着的选项，这个操作的语法如下所示：

```
option get window name class
```

选项数据库可以说是一专多能，比简单的.Xdefaults文件要能干的多。我们可以利用它来模仿同一个应用程序在Xt环境支持下缺省的加载机制。比如说，在加载这个应用程序之前，可以在X窗口环境变量指定的子目录里找到缺省的设置文件，这些环境变量包括XFILESEARCHPATH、XAPPLRESDIR、XUSERFILESEARCHPATH和XENVIRONMENT等。

我们可以给应用程序的各个缺省配置文件确定一个优先级，根据优先级把这些配置文件放

到环境变量指示的各个子目录里去；而当我们用“option readfile ...”命令把它们读到应用程序里来的时候，就可以按它们的优先级确定先后顺序。完成这一模仿的代码应该和下面这段程序差不多：

```
global env

if [info exists env(XFILESEARCHPATH)] {
   look for the app-defaults file in XFILESEARCHPATH dir
   load the file with priority 1
   } else {
   look in one of {/usr/lib/X11/app-defaults, /usr/openwin/lib/app-defaults,
/usr/lib/app-defaults..} directories and load the file with priority 1
   }
   if [info exists env(XUSERFILESEARCHPATH)] {
      look for the app-defaults file in XUSERFILESEARCHPATH dir
      load the file with priority 2 over riding XFILESEARCHPATH priority
       } elseif [info exists env(XAPPLRESDIR)] {
            look for the app-defaults file in XAPPLRESDIR dir
      load the file with priority 2 over riding XFILESEARCHPATH priority
       } elseif
          load app defaults file if exists from current dir with priority 2
      }
 if [ the defaults exist in .Xdefaults ] {
    load them with priority 3  }
      if [info exists env(XENVIRONMENT)] {
load the file XENVIRONMENT as the app defaults file with second-highest priority
 }
 finally load command-line options with the highest priority
```

16.5.13 应用程序间的通信

Tk为共享着同一个显示服务器（但有可能分别显示在不同屏幕上）的两个应用程序之间的通信准备了一个有力的武器：send命令。举个例子，应用程序A可以向应用程序B（比如另一个名字是B的Tcl解释器）发送一条命令，让后者输出字符串“hello”，如下所示：

```
send B [list puts "hello" ]
```

应用程序B接收到这个命令后会执行“puts "hello"”命令。这一方面的进一步情况请参考send命令的使用手册页。

16.5.14 selection命令

我们来设想这样一个场面：一位用户正在一个运行着多个终端仿真程序的桌面上操作着，他用鼠标左键在某个终端仿真程序里选取了一块文本，然后用鼠标中键把那块文本粘贴到了另一个终端仿真程序里去。在这个看起来并不复杂的操作过程里，实际发生了许多的事情。

当用户决定在一个终端仿真程序里选取些东西的时候，这个终端仿真程序首先必须弄清楚用户到底都选取了哪些东西，而它也就成为了这个选取物的属主。成为选取物的属主意味着要做这样的事情：当有其他应用程序请求获得这个选取物时，属主要把这个选取物转换为请求方应用程序指定的类型。一个希望以某种特定格式获得这个选取物的客户向这个选取物的属主提出请求。X客户之间所有这些交互操作都在“X客户间通信符号约定规则”（X Inter-Client Communication Convention Manual，简称ICCCM）中有明确的规定。

选取物可以是PRIMARY、SECONDARY和CLIPBOARD等多种类型。PRIMARY类型选取

物的名字是“XA_PRIMARY”，它是各种客户使用的缺省值。SECONDARY类型选取物的名字是“XA_SECONDARY”，当应用程序需要使用一个以上的选取物时，才会用到它。CLIPBOARD（剪贴板）选取物经常用来保存被删除的数据。

selection命令为我们准备了一个通往X选取机制的Tcl编程接口，它完全实现了“X客户间通信符号约定规则”中描述的选取功能。下面是一个演示选取操作的例子。它创建了一个新的从属Tk解释器和一个文本素材。它通过文本的sel标签对文本素材里的文本进行选取。使用text命令的sel标签将把选取物缺省地设为PRIMARY类型。这样，当那个从属解释器调用“selection own”时，它将拥有这个选取物，成为它的属主。接着，主控解释器检索这个选取物，并把它输出到标准输出stdout去。

```
#!wish
interp create foo

foo eval {
    load {} Tk
    text .t
    pack .t
    .t insert end "Hello World!"
    .t tag add sel  0.0 end
    selection own
    .t insert end "\n"
#   .t insert end "[selection get -selection SECONDARY]"
    .t insert end "[selection get ]"

}
puts "[selection get]"
```

如果把上例中注释行前面的“#”号去掉，应用程序将报告出现一个错误，因为并没有第二份选取物。选取物的属主还可以拒绝任何其他应用程序或其他组件对选取物的检索。

16.5.15 Clipboard命令

CLIPBOARD（剪贴板）是X中的另外一种选取机制。CLIPBOARD选取物可以用来容纳被删除的数据。举例来说，甲客户可以把删掉的数据放到剪贴板上并退出。而乙客户却依然能够从剪贴板上检索到被删除掉的选取物——尽管此时原来的属主客户早已经离开了。这对PRIMARY和SECONDARY类型的选取物来说是不可能的，因为当有客户请求PRIMARY或SECONDARY选取物的时候，它会向属主发送一个请求。如果属主已经不在了，选取物检索请求就会失败。

clipboard命令提供了一个到Tk剪贴板的Tcl编程接口，这个Tk剪贴板通过selection机制把数据保存起来供后续操作检索。Tk_clipboard与大家在其他操作系统上见过的系统剪贴板不是同一个概念。使用Tk工具包开发的应用程序会产生删除数据，为了让这些被删除的数据能够在这些应用程序之间得到利用，人们才设计了Tk_clipboard，把删除数据放在它里面；也就是说，Tk_clipboard只是一个容纳删除数据的容器而已。要想把数据拷贝到剪贴板去，必须先调用“clipboard clear”，然后是一连串的一个或多个“clipboard append”调用。下面的例子演示了clipboard命令的用法。

```
#!wish
interp create foo
```

```
foo eval {
    load {} Tk
    clipboard clear
    clipboard append -type STRING "Clipboard Data"

}
       interp delete foo
puts "[Selection get -selection CLIPBOARD]"
```

上例创建了一个名为foo的新Tk解释器。foo把数据追加到剪贴板上去。主控解释器删掉从属解释器，然后从剪贴板检索出数据来。正如大家看的那样，即使从属解释器已经不在了，选取物还是能够被检索到。

16.5.16 窗口管理器

Tk为窗口和窗口管理器之间的通信准备了一个wm命令。窗口管理器的典型功能包括：管理应用程序窗口之间的焦点切换、设置顶层窗口的属性、给窗口分配颜色图（colormap）、在显示器屏幕上摆放顶层窗口，等等。应用程序内部的各种窗口都是由应用程序自己负责管理的，窗口管理器只与各种应用程序的顶层窗口打交道，所以wm命令的参数必须是顶层窗口。客户可以向窗口管理器请求的功能包括以下这些：

- 把顶层窗口缩小为图标，或把图标恢复为顶层窗口。
- 把顶层窗口摆放到屏幕上的特定位置。
- 把顶层窗口恢复为初始化时的尺寸位置。
- 设置顶层窗口的窗口标题。
- 设置应用程序的焦点模型。
- 调整一个顶层窗口的长度、宽度。
- 覆盖窗口管理器提供的缺省修饰，等等。

我们用几个例子来进行说明。在程序里，我们可以象下面这样在应用程序里用wm命令查询某个顶层窗口的状态：

```
% wm iconify .
% puts "[wm state .]"
iconic
```

用户还可以对顶层窗口上的窗口管理器协议进行设置，改变窗口管理器操作的缺省行为。比如说，我们可以在顶层窗口里建立一个自己接收到焦点或自己被删除时调用的事件处理器，如下所示：

```
% wm protocol . WM_TAKE_FOCUS {puts "window . got foucs"}
% wm protocol . WM_DELETE_WINDOW {puts "window . is being deleted"; exit}
```

运行上面的代码，当顶层窗口“.”获得焦点时，程序会输出字符串“window . got focus”；当我们用窗口管理器的delete（删除）按钮删除顶层窗口“.”时，程序会输出字符串“window . is being deleted”。

wm命令还可以用来设置或者查询顶层窗口的窗口标题。如下所示：

```
wm title ?newtitle? .
```

客户还可以要求窗口管理器不对顶层窗口进行修饰。当客户做出这样的请求时，窗口管理器就不会在该顶层窗口的窗口管理器框里提供完成最小化、还原或尺寸调整等操作的按钮。下面这段代码请求窗口管理器不要给顶层窗口.t加上任何修饰。

```
toplevel .t
wm withdraw .t
wm transient .t .
wm deiconify .t
```

运行这个程序，我们将看到如图16-19所示的窗口。

图 16-19

这段代码演示了窗口管理器命令的好几个功能。应用程序先请求窗口管理器从屏幕上撤掉自己的顶层窗口；然后请求窗口管理器去掉该顶层窗口的最小化/最大化修饰；最后，它请求窗口管理器把自己的窗口重新在屏幕上显示出来。

16.5.17 动态/静态加载

在上一小节里，我们创建了新解释器并静态地加载了Tk。解释器可以用两种办法加载Tk工具包——静态或者动态，两种办法都要使用load命令。就静态加载来说，可执行程序必须与Tk预加载在一起。请看下面例子里的代码：

```
interp create debugInterp

debugInterp eval {
    load {} Tk
    text .t
    pack .t
    update
}

proc debug {interp args} {
    debugInterp eval [list  .t insert end "$args"]
}

debug . "hello world!"
```

这段代码静态地把Tk可执行程序加载到一个新创建的解释器里。从这个例子还可以看出解释器之间是如何进行通信的。主控解释器创建了一个debug例程，它通过与debugInterp进行的通信把调试信息显示出来。

那么，我们怎样才能把一个Tk解释器动态地加载到一个tclsh应用程序里去呢？下面的例子里演示了具体的做法：

```
interp create debugInterp

debugInterp eval {
    load /usr/local/tk8.2b3/unix/libtk8.2.so Tk
    text .t
    pack .t
    update
}

proc debug {interp args} {
```

```
    debugInterp eval [list  .t insert end "$args"]
}

debug . "hello world!"
vwait foob
```

这段代码执行的前提是你必须已经对Tcl/Tk的发行版本进行了编译，使它们成为可动态加载的，并且，Tk的动态库libtk8.2.so已经放到了/usr/local/tk8.2b3/unix/子目录里。从上面代码里可以看出：tclsh创建了一个新的解释器，动态地把Tk加载到这个新创建出来的解释器里去，然后主控解释器用vwait命令进入事件循环。如果主控解释器没有进入事件循环，这个应用程序会默默地退出，没有任何警告信息。如果你用下面这个命令运行上面这段代码：

```
Tclsh8.0 dynamicLoad.tk
```

Tk解释器就会动态地被加载上！

16.5.18 Safe Tk

假设你从网络上下载了一个Tcl脚本，执行它。如果这个脚本不怀好意，它就会对你的系统造成严重的损害。它可能会删除你所有的文件，也可能会把你的文件传输到其他计算机去。如何才能保证这些无法信任的脚本不造成损害呢？在1994年，Marshall和Rose两人提出了Safe-Tcl，它最初被认为是这样一种机制：它允许电子邮件包含将要在接收者的计算机上运行的Tcl脚本。Safe-Tcl被添加到7.5版本的Tcl核心代码里。

Safe-Tcl的目的是建立一个实验箱环境，让用户可以在这个环境里安全地执行无法信任的Tcl脚本，不必担心产生什么副作用。

安全解释器有一个有限制的命令集合。下表里的命令将被隐藏起来：

cd	exec	exit	fconfigure
glob	load	open	socket
source	vwait	pwd	file

Tcl的safe base使用手册页里对如何建立安全的解释器有详细的介绍。有时候，给新创建的安全解释器加上一些访问限制是很有必要的；比如说，允许它打开某个特定子目录里的文件等。Safe-Tcl里已经提供了这样的访问限制。利用“::safe::interpCreate”命令创建的解释器利用假名原理通过无法信任的脚本对那些有潜在危险的功能提供了适度的访问权限。因此，我们可以认为Safe-Tcl是一种安全地执行无法信任的Tcl脚本的机制，并且通过这类脚本对那些有潜在危险的功能提供的访问权限是适度的。

类似于Safe-Tcl，在执行无法信任的Tk脚本也需要建立实验箱环境。比如说，你肯定不想让Tk插件删掉你所有的顶层窗口，也不想让它偷走你的X剪贴内容。Safe Tk增加了把解释器配置为能够安全地完成Tk操作和把Tk加载到安全解释器中去的能力。在缺省的情况下，你是不能把Tk加载到安全解释器里去的，因为在安全解释器里根本就不允许使用load命令。Safe Tk把无法信任的Tk脚本单独放到一个实验箱环境里去执行，这就保证它不会造成任何损害。

“::safe::loadTk”命令对安全解释器里要求的数据结构进行必要的初始化，然后把Tk加载到其中。这个命令返回的是那个安全解释器的名字。“::safe::loadTk”把取自主控解释器的tk_library的值添加到安全解释器的虚拟路径里，这样就可以在安全解释器里实现Tk的自动加载了。

下面的例子演示的操作是：除非你使用了“::safe::loadTk”命令，否则你是不能把Tk加载到一个安全解释器里去的。

```
::safe::interpCreate  safeInterp

::safe::interpAddToAccessPath safeInterp /tmp

::safe::loadTk safeInterp

interp create -safe safeInterp2

puts " 1 -> [interp hidden safeInterp]"
puts " 2 -> [interp hidden safeInterp2]"

puts " 1 -> [interp aliases safeInterp]"
puts " 2 -> [interp aliases safeInterp2]"

safeInterp eval {
    text .t
    pack .t
    update
}

safeInterp2 eval {
     load {} Tk
 }
```

运行上面这个脚本，你将看到如下所示的错误信息：

```
~/wrox/tk>wish safeInterp.tk
 1 -> file socket send open pwd glob exec encoding clipboard load fconfigure source
exit toplevel wm grab menu selection tk bell cd
 2 -> file socket open pwd glob exec encoding load fconfigure source exit cd
 1 -> file load source exit encoding
 2 ->
Error in startup script: invalid command name "safeInterp2"
    while executing
"safeInterp2 eval {
     load {} Tk
 }"
    (file "safeInterp.tk" line 26)
```

这表示你不能把Tk加载到一个安全解释器里去，因为它不是用“::safe::createInterp”命令创建的。如果你想进一步了解安全解释器的使用方法，请自行研究Tk发行版本里自带的safeDebug.tk示例。

16.6 一个复合素材

我们已经学习了如何利用各种各样的素材命令编写应用程序的方法。但有时候人们需要全新类型的素材来显示他们的信息，比如格子、电子表、记事本、轮转列表等。虽然Scriptics公司的Tk研发队伍正在计划把这些素材添加到核心里去，但在我们编写本书的时候它们还没有添加进去。所以我们在这一小节的问题就是：人们怎样才能创建出定制的素材？

有两种办法可以做的这一点。第一种办法是使用Tcl和Tk的C语言扩展API，它也叫做TEA（Tcl and Tk's C Extension API的字头缩写）。另外一种办法是使用纯粹的Tcl和现有的Tk素材来实现这一目的。因为我们还没有讨论过Tcl和Tk的C语言API，而且就是我们想在这里介绍它们也多少有些力不从心，所以我们将要采取的办法是完全利用Tcl和现有的Tk素材来创建一个复合素材（MegaWidget）。我们将要创建的复合素材是一个“树”素材。就目前来说，Tk还没有内建的树素材。这个素材示例还说不上完整，但它还是相当有用，并且给大家指出了一个正确的方向。

> 这个例子里采用的算法和某些接口取自GPL许可证下的comp.lang.tcl新闻组。我见过的其他版本都没有这个基于包概念/名字空间（package/namespace-based）的实现版本的灵活适应性。

我们将使用Tcl的包（package）和名字空间（namespace）机制来把我们的树素材封装在一个抽象的数据结构里。接下来的问题是我们需要给这个树素材提供什么样的素材命令和配置命令？出于配置方面的考虑，我们将为它准备上大部分标准的Tk配置选项，比如-font、-backgroundcolor等，另外再加上一些树素材特有的配置选项。因为树素材支持层次结构，所以我们应该提供诸如additem、delitem、config、setseletion、getseletion等方法。好了，我们这就开始定义这个树素材包。

```
#

#       Sample tree mega-widget.Can be used to display hierachies. The clients
#       who use this package need to specify parent and tail procedures for any
#       element of the tree hierarchy. All the nodes that get stored inside the
#       tree are complete path names separated by '/'. The toplevel node is
#       always /
#

package provide tree 1.0

namespace eval tree {
   variable tree


   #
   # default font setup
   #

   switch $tcl_platform(platform) {
      unix {
    set tree(font) \
        -adobe-helvetica-medium-r-normal-*-11-80-100-100-p-56-iso8859-1
      }
      windows {
    set tree(font) \
        -adobe-helvetica-medium-r-normal-*-14-100-100-100-p-76-iso8859-1
      }
   }

   #
   # Bitmaps used to show which parts of the tree can be opened/closed
   #

   set maskdata "#define solid_width 9\n#define solid_height 9"
   append maskdata {
     static unsigned char solid_bits[] = {
0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01,
```

```
    0xff, 0x01, 0xff, 0x01, 0xff, 0x01
      };
  }
  set data "#define open_width 9\n#define open_height 9"
  append data {
     static unsigned char open_bits[] = {
   0xff, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x7d, 0x01, 0x01, 0x01,
   0x01, 0x01, 0x01, 0x01, 0xff, 0x01
      };
  }

  set tree(openbm) [image create bitmap openbm -data $data \
         -maskdata $maskdata \
         -foreground black -background white]

  set data "#define closed_width 9\n#define closed_height 9"
  append data {
     static unsigned char closed_bits[] = {
   0xff, 0x01, 0x01, 0x01, 0x11, 0x01, 0x11, 0x01, 0x7d, 0x01, 0x11, 0x01,
   0x11, 0x01, 0x01, 0x01, 0xff, 0x01
      };
  }
  set tree(closedbm) [image create bitmap closedbm -data $data \
           -maskdata $maskdata \
           -foreground black -background white]


   namespace export create additem delitem config setselection getselection
   namespace export openbranch closebranch labelat
}
```

如上所示，从树素材的外面看去，它将有一个名为tree的包变量和additem、delitem、config、setselection等方法。一个tree包变量将用来保存创建出来的所有树素材的必要信息。从上面的代码里可以看出，tree还将创建出一些图像，这些图像将用来完成打开、关闭树枝以及把树枝保存在tree数据结构里等操作。

我们下一步要定义素材命令。我们先来定义tree::create命令。这个命令的执行过程是这样的：对配置选项进行分析，用在create命令里给出的路径创建一个画布。create命令还要检查-parent和-tail素材创建选项。这些选项是由客户程序指定的过程，这样，tree命令就能确定任何一个结点的前部和后部。后部大概说来就是结点名字的最后一部分。请看这个例子：假设结点的名字是“a/b/c”，则tail命令将返回“c”（假设字符“/”被用做是一个路径分隔符）；而parent命令将返回“/a/b”。树复合素材要求使用“/”做为路径分隔符，并且所有结点都用绝对路径来表示。

-parent和-tail选项命令身后的逻辑考虑是：要允许一棵“树”显示各种层次信息，而不仅仅是子目录结构的架子。create命令还将对变量进行初始化，比如要把selection和selidx分别初始化为当前中选的结点和它的标签。create命令会安排稍后重新绘制这棵树。

```
# tree::create --
#
#   Create a new tree widget. Canvas is used to emulate a tree
#   widget. Initialized all the tree specific data structures. $args become
#   the configuration arguments to the canvas widget from which the tree is
#   constructed.  #

# Arguments:
#    -paren   proc
#
#      sets the parent procedure provided by the application. tree
#      widget will use this procedure to determine the parent of an
#      element. This procedure will be called with the node as an
```

```
#       argument
#
#    -tail     proc   |Given a complete path this proc will give the end-element
#                        name|
#
# Results:    A tree widget with the path $w is created.
#

proc tree::create {w args} {
   variable tree
   set newArgs {}

   for {set i 0} {$i < [llength $args]} {incr i} {
      set arg [lindex $args $i]
      switch -glob -- $arg {
    -paren* {set tree($w:parenproc) [lindex $args [expr $i +1]]; incr i}
     tail* {set tree($w:tailproc) [lindex $args [expr $i +1]]; incr i}
    default {lappend newArgs $arg}
    }
      }

   if ![info exists tree($w:parenproc)] {
      set tree($w:parenproc) parent
   }

   if ![info exists tree($w:tailproc)] {
      set tree($w:tailproc) tail
   }

  eval canvas $w -bg white $newArgs
  bind $w <Destroy> "tree::delitem $w /"
  tree::DfltConfig $w /
  tree::BuildWhenIdle $w
  set tree($w:selection) {}
  set tree($w:selidx) {}
}
```

创建树素材的时候，一个名为“/”的根结点将自动被创建出来。每次添加一个新结点的时候，所有结点将按有关的缺省配置被初始化，其中包括与该结点关联着的子结点、是否需要把该结点显示为打开状态、与该结点关联着的图标和标签等等。tree::DfltConfig就是用来对结点进行初始化的。

```
# tree::DfltConfig  --
#
#   Internal fuction used to initial the attributes associated with an item/node.
Usually called when an item is added into the tree
#
#  Arguments:
#   wid   tree widget
#   node  complete path of the new node
#
#  Results:
#   Initializes the attributes associated with a node.


proc tree::DfltConfig {wid node} {
  variable tree
  set tree($wid:$node:children) {}
  set tree($wid:$node:open) 0
  set tree($wid:$node:icon) {}
  set tree($wid:$node:tags) {}

}
```

就象其他Tk素材一样，树素材应该支持-config素材命令。这是通过树素材的tree::config类方

法（class method）来实现的。

```
# tree::config --
#
#   Fuction to set tree widget configuration options.
#
#  Arguments:
#   args  any valid configuration option a canvas widget takes
#
#  Results:
#   Configures the underlying canvas widget with the options
#

proc tree::config {wid args} {
variable tree
    set newArgs {}
    for {set i 0} {$i < [llength $args]} {incr i} {
      set arg [lindex $args $i]
      switch -glob -- $arg {
    -paren* {set tree($w:parenproc) [lindex $args [expr $i +1]]; incr i}
    -tail* {set tree($w:tailproc) [lindex $args [expr $i +1]]; incr i}
    default {lappend newArgs $arg}
    }
      }
  eval $wid config $newArgs
}
```

到了这里，树素材的创建和配置工作就完成了。下一步是在树上增加一个结点项。这个例程必须保证不会给这个树增加一个重复的结点项。例程查出新项目的父结点，把前者添加到后者的子结点中去。例程还要分析该项目的标签，比如-image和-tag等；还要对该项目的属性进行设置。-tag选项把标签挂到新添加的项目上去。additem例程还要安排在应用程序不忙的时候重新绘制这个树素材。如下所示：

```
#  tree::additem --
#
#       Called to add a new node to the tree.
#
#  Arguments:
#       wid   tree widget
#       node  complete path name of the node (path is separated by /)
#       args  can be -image val, -tags (taglist) to identify the item
#
#  Results:
#       Adds the new item and configures the new item
#

proc tree::additem {wid node args} {
  variable tree
   set parent [$tree($wid:parenproc) $node]
   set n [eval $tree($wid:tailproc) $node]
  if {![info exists tree($wid:$parent:open)]} {
    error "parent item \"$parent\" is missing"
  }
  set i [lsearch -exact $tree($wid:$parent:children) $n]
  if {$i>=0} {
      return
  }
  lappend tree($wid:$parent:children) $n
  set tree($wid:$parent:children) [lsort $tree($wid:$parent:children)]
  tree::DfltConfig $wid $node
  foreach {op arg} $args {
    switch -exact -- $op {
      -image {set tree($wid:$node:icon) $arg}
      -tags {set tree($wid:$node:tags) $arg}
```

```
    }
  }
  tree::BuildWhenIdle $wid
}
```

删除一个项目正好与additem的功能相反，它的作用是从整个tree数据结构里和它的父结点项的子结点名单里删除这个项目。它也要安排稍后重新绘制这棵树。

```
#  tree::delitem  --
#
#   Deletes the specified item from the widget
#
#  Arguments:
#   wid    tree widget
#   node   complete path of the node
#
#  Results:
#   If the node exists, it will be deleted.
#

proc tree::delitem {wid node} {
  variable tree
  if {![info exists tree($wid:$node:open)]} return
  if {[string compare $node /]==0} {
    # delete the whole widget
    catch {destroy $wid}
    foreach t [array names tree $wid:*] {
      unset tree($t)
    }
  }
  foreach c $tree($wid:$node:children) {
    catch {tree::delitem $wid $node/$c}
  }
  unset tree($wid:$node:open)
  unset tree($wid:$node:children)
  unset tree($wid:$node:icon)
  set parent [$tree($wid:parenproc) $node]
   set n [eval $tree($wid:tailproc) $node]
  set i [lsearch -exact $tree($wid:$parent:children) $n]
  if {$i>=0} {
    set tree($wid:$parent:children) [lreplace $tree($wid:$parent:children) $i $i]
  }
  tree::BuildWhenIdle $wid
}
```

用户能够控制树素材里的哪个结点将被设置为一个选取项。setselection和getselection例程就是用来完成这一工作的。选取项对象将用高亮度背景色绘制出来。

```
 tree::setselection  --
#
#   Makes the given node as the currently selected node.
#
#  Arguments:
#   wid - tree widget
#   node - complete path of the one of nodes
#
#  Results:
#   The given node will be selected
#

proc tree::setselection {wid node} {
  variable tree
  set tree($wid:selection) $node
  tree::DrawSelection $wid
}
```

```
#  tree::getselection  --
#
#   Get the currently selected tree node
#
#  Arguments:
#   wid - tree widget
#
#  Results:
#   If a node is currently selected it will be returned otherwise NULL
#

proc tree::getselection wid {
  variable tree
  return $tree($wid:selection)
}
```

下一个任务是建立/构造这个树。算法递归地调用每一个结点绘制出它自己的子结点。在绘制出这个树之后，它将设立卷屏区。这样，当把这个树与卷屏条相关联的时候，它就能够正确地动作。它还会绘制出当前的选取项。如下所示：

```
#  tree::Build --
#
#   Internal function to rebuild the tree
#
#  Arguments:
#   wid -  tree widgets
#
#  Results:
#
#     This routine has no complex logic in it. Deletes all the current items
#    on the canvas associated with the tree and re-builds the tree.  #
#
#

proc tree::Build wid {
  variable tree
  $wid delete all
  catch {unset tree($wid:buildpending)}
  set tree($wid:y) 30
  tree::BuildNode $wid / 10
  $wid config -scrollregion [$wid bbox all]
  tree::DrawSelection $wid
}
```

树素材绘制算法的精华是BuildNode。它是一个基本的树绘制算法，如果父结点的打开属性被设置上了的话，这个父结点和它的子结点就都将被绘制出来。如果某个子结点的打开属性也被设置上了，就将递归调用BuildNode把它的子结点也绘制出来。这个绘制算法本身就是对自己功能的一个很好的说明。

```
#  tree::BuildNode --
#
#   Function called by tree::build to incrementally build each node
#
#  Arguments:
#   wid - tree widget
#       node - complete path of the node
#   in - the starting x-cordinate
#
#  Results:
#   The node gets drawn
#

proc tree::BuildNode {wid node in} {
```

```
  variable tree
  if {$node=="/"} {
    set vx {}
  } else {
    set vx $node
  }
  set start [expr $tree($wid:y)-10]
  foreach c $tree($wid:$node:children) {
    set y $tree($wid:y)
    incr tree($wid:y) 17
    $wid create line $in $y [expr $in+10] $y -fill gray50
    set icon $tree($wid:$vx/$c:icon)
    set taglist x
    foreach tag $tree($wid:$vx/$c:tags) {
      lappend taglist $tag
    }
    set x [expr $in+12]
    if {[string length $icon]>0} {
      set k [$wid create image $x $y -image $icon -anchor w -tags $taglist]
      incr x 20
      set tree($wid:tag:$k) $vx/$c
    }
    set j [$wid create text $x $y -text $c -font $tree(font) \
                                  -anchor w -tags $taglist]
    set tree($wid:tag:$j) $vx/$c
    set tree($wid:$vx/$c:tag) $j
    if {[string length $tree($wid:$vx/$c:children)]} {
      if {$tree($wid:$vx/$c:open)} {
         set j [$wid create image $in $y -image $tree(openbm)]
         $wid bind $j <1> "set tree::tree($wid:$vx/$c:open) 0; tree::Build $wid"
         tree::BuildLayer $wid $vx/$c [expr $in+18]
      } else {
         set j [$wid create image $in $y -image $tree(closedbm)]
         $wid bind $j <1> "set tree::tree($wid:$vx/$c:open) 1; tree::Build $wid"
      }
    }
  }
  set j [$wid create line $in $start $in [expr $y+1] -fill gray50 ]
  $wid lower $j
}
```

现在，当这个树已经被显示在屏幕上之后，如果用户又点击了某个结点旁边的加号“+”图案打开了这个树枝，下面给出的openbranch例程就会安排重新绘制这个树，把该结点的子结点也显示出来。如下所示：

```
#  tree::openbranch --
#
#       A callback that gets called to open a node to show its children
#
#  Arguments:
#       wid - tree widget
#       node - the node whose children should be shown
#
#  Results:
#       The children of the node will be drawn
#

proc tree::openbranch {wid node} {
  variable tree
  if {[info exists tree($wid:$node:open)] && $tree($wid:$node:open)==0
      && [info exists tree($wid:$node:children)]
      && [string length $tree($wid:$node:children)]>0} {
    set tree($wid:$node:open) 1
    tree::Build $wid
  }
}
```

类似地，当用户点击某个结点旁边的减号“-”图案时，closebranch例程将安排重新绘制这个树，关闭那个结点，不再显示该结点的子结点。如下所示：

```
# tree::closebranch   --
#
#       The opposite of open branch, see above
#
#  Arguments:
#
#
#  Results:
#

proc tree::closebranch {wid node} {
  variable tree
  if {[info exists tree($wid:$node:open)] && $tree($wid:$node:open)==1} {
    set tree($wid:$node:open) 0
    tree::Build $wid
  }
}
```

DrawSelection例程将用高亮度背景色绘制出当前被选取的结点。

```
#  tree::DrawSelection --
#
#       Highlights the current selection
#
#  Arguments:
#       wid - tree widget
#
#  Results:
#       The current selection will be high-lighted with skyblue
#

proc tree::DrawSelection wid {
  variable tree
  if {[string length $tree($wid:selidx)]} {
    $wid delete $tree($wid:selidx)
  }
  set node $tree($wid:selection)
  if {[string length $node]==0} return
  if {![info exists tree($wid:$node:tag)]} return
  set bbox [$wid bbox $tree($wid:$node:tag)]
  if {[llength $bbox]==4} {
    set i [eval $wid create rectangle $bbox -fill skyblue -outline {{}}]
    set tree($wid:selidx) $i
    $wid lower $i
  } else {
    set tree($wid:selidx) {}
  }
}
```

有这么多的例程需要重新绘制这个树，但一一做来的效率是很低的。BuildWhenIdle例程的任务就是对这些重新绘制操作进行收集整理，然后安排一个事件处理器统一绘制出这个树来。

```
#  tree::BuildWhenIdle --
#
#       Function to reduce the number redraws of the tree. When a redraw is not
#       immediately warranted this function gets called
#
#  Arguments:
#       wid  - tree wiget
#
#  Results:
#       Set the tree widget to be redrawn in future.
```

```
#

proc tree::BuildWhenIdle wid {
  variable tree
  if {![info exists tree($wid:buildpending)]} {
    set tree($wid:buildpending) 1
    after idle "tree::Build $wid"
  }
}
```

最后的tree::labelat例程将返回给定x和y素材坐标处的结点。关键在于使用了画布素材的内建命令。如下所示：

```
#  tree::labelat --
#
#       Returns the tree node closest to  x,y co-ordinates
#
#  Arguments:
#       wid       tree widget
#       x,y       co-ordinates
#
#  Results:
#       The node closest to x,y will be returned.


proc tree::labelat {wid x y} {
  set x [$wid canvasx $x]
  set y [$wid canvasy $y]
  variable tree
  foreach m [$wid find overlapping $x $y $x $y] {
    if {[info exists tree($wid:tag:$m)]} {
      return $tree($wid:tag:$m)
    }
  }
  return ""
}
```

因为树素材的基石是画布，所以树素材将支持画布所有的绑定命令，命令的语法也完全一样。

包文件的生成

现在，我们已经定义好复合型的树素材了，我们该怎样使用它呢？在我们开始使用这个树素材开发一个新的应用程序之前，需要为这个树素材生成一个pkgIndex文件。具体做法是：把tree.tcl文件拷贝到/usr/local/lib/tcl子目录里，然后在一个wish shell里运行下面这些命令：

```
$ wish
% cd /usr/local/lib/tcl
% pkg_mkIndex . tree.tcl
```

这会在/usr/local/lib/tcl子目录里生成一个pkgIndex.tcl文件。在创建这个文件之前请先查明没有旧的pkgIndex.tcl文件，因为tcl将覆盖掉它。

生成pkgIndex.tcl文件之后，还需要通知你的应用程序你打算使用树素材，这需要把/usr/local/lib/tcl子目录追加到auto_loadpath路径上，具体做法是增加下面这两行：

```
Lappend auto_path /usr/local/lib/tcl
Package require tree
```

16.7 使用复合型树素材的应用程序

我们用树素材包来开发一个简单的应用程序。我们将要开发的应用程序将把一个根文件系统的树状层次结构显示出来。

我们先要通知应用程序两件事：素材包文件的存放地点和要用它来加载树素材包。如下所示：

```
#!/usr/local/bin/wish
#
# Simple application showing the use of tree mega widget
#
lappend auto_path /usr/local/lib/tk
package require tree
```

现在，我们要通知在parent和tail例程上面的树素材。在缺省的情况下，它们就是普通的文件操作例程dirname和tail，因为我们将要显示的是一个根文件系统。如下所示：

```
#
# Create utility procs that tree wiget uses to query parent
# and tail components of a node
#
proc parent {item} {
   return [file dirname $item]
}
proc tail {item} {
   return [file tail $item]
}
```

创建用来显示子目录和文件图像的图像，如下所示：

```
# Create imanges that we use to display directory and a normal file
#

image create photo idir -data {
    R0lGODdhEAAQAPIAAAAAAHh4eLi4uPj4APj4+P///wAAAAAAACwAAAAAEAAQAAADPVi63P4w
    LkKCtTTnUsXwQqBtAfh910UU4ugGAEucpgnLNY3Gop7folwNOBOeiEYQ0acDpp6pGAFArVqt
    hQQAO///
}
image create photo ifile -data {
    R0lGODdhEAAQAPIAAAAAAHh4eLi4uPj4+P///wAAAAAAAAAAACwAAAAAEAAQAAADPkixzPOD
    yADrWE8qC8WN0+BZAmBq1GMOqwigXFXCrGk/cxjjr27fLtout6n9eMIYMTXsFZsogXRKJf6u
    P0kCADv/
}
```

我们编写的下面这个例程的功能是这样的：如果用户双击一个项目，而这个项目又恰好是一个子目录，就把该结点的子结点们动态地增加显示出来。

```
#
# Dynamically add entries to the tree widget
#
proc AddDir {wid dir} {
    if ![file isdirectory $dir ] {
        return;
    }
    foreach file [exec ls $dir] {
        set file [file join $dir $file]
        if [file isdirectory $file] {
            tree::additem $wid [file join $dir $file] -image idir
        } else {
            tree::additem $wid [file join $dir $file] -image ifile
```

```
        }
    }
}
```

主程序部分创建出这个树，并为树素材设置好双击动作的绑定。它还负责给这个树素材加上顶层结点。

```
#
# main proc
#
#
# Create tree wiget and set up bindings
#
tree::create .t -width 150 -height 400

#
# open a node when gets double clicked.
#
.t bind x <Double-1> {
    puts "Callled"
    set child [tree::labelat %W %x %y]
    AddDir %W $child
    tree::openbranch %W $child
}


AddDir .t /

# mange the wiget

pack .t -fill both -expand 1
update
```

图 16-20

当你运行这个程序的时候，应该看到类似于图16-20这样的窗口。

16.8 Tk进程记录查看器

在学习如何创建一个复合素材的方法之后，有没有兴趣用Tk 8.0里的新功能（比如事件机制）编写一个应用程序？Linux用户经常会发现自己每天都使用着“tail -f”或“find / -print”命令。那我们为什么不编写一个应用程序把这些命令的输出显示在一个文本窗口里呢？我们还将为他们运行的命令创建快捷键，这样，用户就可以通过点击这些快捷键再次运行那些命令了。

这个应用程序支持两种形式的记录——文件记录和命令记录。文件记录记的是“tail -f filename”类型的命令，而命令记录记的是“find / -print”命令。用户可以在文件记录的输出里指定一个文件名和一个快捷键。一个“tail -f filename”类型的命令将被构造和关联到指定的快捷键上去。但对命令记录来说，用户就必须给出完整的命令名和快捷键名。

我们给这个应用程序起名为“Tk Process Log Viewer”（Tk进程记录查看器）。那么，打造这个应用程序都需要哪些材料呢？从用户操作界面方面考虑，我们需要一个用来处理命令的菜单条、一个用来显示输出的文本素材、一个用来显示错误信息的状态条、几个用来设定命令及其快捷键名的输入框、还要有一个用来显示当前可用快捷键名的选项按钮。我们还需要有一个用来终止当前查看操作的“stop”按钮，在菜单条里还应该有一个用来删除不想要的快捷键名的“delete”按钮。

我们从定义全局变量开始动手，这些变量将用来建造我们的应用程序。我们把所有这些全局变量保存在一个名为tailOpts的数组里。

```
#!/usr/local/bin/wish8.0
# logView.tk  --
#
#       Essentially a general purpose GUI wrapper to tail, gui and any commands
#       that will output data continuosly. This GUI has the ability to record
#       the commands as smart buttons, so that you can re-run the same commands
#       again and again without having to retype


set tailRc "~/.tailrc"
wm title .  "Process Log Viewer"
wm iconname . "Log Viewer"

global tailSize textw fileName tailFd curNick tailOpts statusImgWin

# tailSize --> size in lines of tail output to display
# fileName --> File name: variable
# tailFd   --> proc fd or file fd of current tail process
# curNick  --> current nick being shown; nick essentially a shortcut to a cmd.
# tailOpts --> saved options
# statusImgWin --> window showing what kind of error it is!

set tailSize 20
set fileName "/usr/local/processlog/logView.tk";     #include your own path here

#
# file types for the file selection dialog box.
#

set tailOpts(types) {
   {"All files"                *}
   {"Text files"               {.txt .doc}       }
   {"Text files"               {}            TEXT}

   {"Tcl Scripts"              {.tcl}        TEXT}
   {"C Source Files"           {.c .h}           }
   {"All Source Files"         {.tcl .c .h}      }
   {"Image Files"              {.gif}            }
   {"Image Files"              {.jpeg .jpg}      }
   {"Image Files"              ""      {GIFF JPEG}}
}
set tailOpts(wins) {}
```

接着，我们来建造用户操作界面。首先是带“File”和“Edit”命令的菜单条。“File”菜单通过一个“Add New . . .”命令按钮支持新命令快捷键的添加操作。“File”菜单上还有一个用来退出这个应用程序的“exit”按钮。“Edit”菜单里有一个“Delete Nicks”按钮，它被用来编辑当前快捷键。

```
proc BuildTailGui {w} {
    global tailSize textw fileName tailFd curNick tailOpts statusImgWin
    global viewOptMenu

    if {$w == "."} {
       set w "";
    }

    #
    # Build Menu for file
    #

    menu $w.menu -tearoff 0
```

```
# File menu
set m $w.menu.file
menu $m -tearoff 0
$w.menu add cascade -label "File" -menu $m -underline 0
$m add command -label "Add New ..." -command {AddNew} -underline 0
$m add command -label "Exit" -command {exit} -underline 0

# Edit Menu

set m $w.menu.edit
menu $m -tearoff 0
$w.menu add cascade -label "Edit" -menu $m -underline 0
$m add command -label "Delete Nicks.." -command {DeleteNicks} -underline 0

# Help Menu
set m $w.menu.help
menu $m -tearoff 0
$w.menu add cascade -label "Help" -menu $m -underline 0
$m add command -label "About..." -underline 0 -command {
    tk_messageBox -parent . -title "Process Log Viewer" -type \
    ok -message    "Tk Tail Tool \nby Krishna Vedati(kvedati@yahoo.com)"
}
```

接着这个例程添加一个用来显示两个记录进程的输出的文本素材和一个用来显示错误和辅助信息的状态条。

然后，例程创建了几行素材。第一行使用户能够在应用程序里增加文件类型快捷键。第二行使用户能够增加命令类的快捷键。最后一行上有两个按钮，一个是用来终止当前记录进程执行的“stop”按钮，另一个是快速选择快捷键用的一个选项按钮。

```
#
# Create status/error message window
#

frame $w.status  -relief sunken -bd 2
set statusImgWin [label $w.status.flag -bitmap info]
label $w.status.lab -textvariable statusText -anchor w -bg "wheat"
pack $w.status.flag -side left
pack $w.status.lab -side left -fill both -expand 1

#
# File name: entry panel
#

frame $w.file
label $w.file.label -text "File name:" -width 13 -anchor w
entry $w.file.entry -width 30 -textvariable fileName
button $w.file.choose -text "..." -command \
    "set fileName \[tk_getOpenFile -filetypes \$tailOpts(types) \
     -parent \[winfo toplevel $w.file\]\];"

button $w.file.button -text "Tail File" \
   -command "AddToView  file \$fileName"
pack $w.file.label $w.file.entry -side left
pack $w.file.choose -side left -pady 5 -padx 10
pack $w.file.button -side left -pady 5 -padx 10
bind $w.file.entry <Return> " AddToView  file \$fileName"
focus $w.file.entry

#
# Command entry panel
#

     frame $w.fileC
     label $w.fileC.cLabel -text "Command:" -width 13 -anchor w
     entry $w.fileC.cEntry -width 40 -textvariable command
     label $w.fileC.nLabel -text "Nick:" -anchor w
```

```
    entry $w.fileC.nEntry -width 15 -textvariable nick
    button $w.fileC.add -text "Add" -command "AddToView \"command\"\
        \$command \$nick;"
    pack $w.fileC.cLabel $w.fileC.cEntry -side left
    pack $w.fileC.nLabel -side left -pady 5 -padx 10
    pack $w.fileC.nEntry -side left -pady 5 -padx 5
    pack $w.fileC.add -side left -pady 5 -padx 5

    # Option Menu command panel

    frame $w.optF
    label $w.optF.label -text "View:" -width 12 -anchor w
    set viewOptMenu [tk_optionMenu $w.optF.optB curNick " "]
    button $w.optF.stop -text "Stop" -command Stop

    pack $w.optF.label -side left
    pack $w.optF.optB -side left -pady 5
    pack $w.optF.stop -side left -pady 5

    # create text widget
    frame $w.textf -bg red
    text $w.textf.text -height [expr $tailSize] -xscrollcommand \
       "$w.textf.scrollh set" -yscrollcommand "$w.textf.scrollv set" -bg lightblue
    set textw $w.textf.text
    scrollbar $w.textf.scrollh -orient horizontal -command "$w.textf.text xview"
    scrollbar $w.textf.scrollv -orient vertical -command "$w.textf.text yview"

    pack $w.textf.scrollv -side right -fill y -expand 1
    pack $w.textf.scrollh -side bottom -fill x -expand 1
    pack $w.textf.text -fill x -fill y -expand 1


    # pack all the frames
    [winfo toplevel $w.textf] configure -menu $w.menu
    pack $w.status -side bottom -fill x -pady 2m
    pack $w.file -side top -fill x -expand 1
    pack $w.fileC -side top -fill x -expand 1
    pack $w.optF -side top -fill x -expand 1
    pack $w.textf -side top -fill x -fill y -expand 1

}
```

用户给某个命令或某个文件设置了与之对应的快捷键之后，TailFile方法将被调用。这个方法会先检查指定的文件是否存在。它构造出一条命令并把它打开为一个进程。然后，它把一个读事件绑定到文件描述符上并返回。此后，每当与文件描述符关联着的进程有数据可读时，就将由读事件负责启动TailUpdate例程读取数据。

```
#  TailFile --
#
#        Show the tail of the request file.
#
#  Arguments:
#        file name to be tailed.
#
#  Results:
#        The tail of the file is shown in the window.
#

proc TailFile { type file {nick ""}} {
    global tailSize tailFd  textw curNick

    set w $textw
    catch {
        fileevent $tailFd readable {}
        close $tailFd
        update
```

```
    }
    $w delete 1.0 end

    if {$type == ""} {
        $w insert end "Illegal type...";
        return
    }
    if {$type == "file"} {
        if {$file == ""} {
            $w insert end "please specify a valid filename..."
            return
        }
        if ![file exists $file] {
            DeleteFromView $file
            $w insert end "file $file does not exist..."
            return
        }
        set nick $file
    } elseif {$type == "command"} {
        if {$file == ""} {
            $w insert end "please specify a command..."
            return
        }
    }

    if {$type == "file"} {
        set tailFd [open "|tail -f $file" r]
        wm title [winfo toplevel $w] "Tail tool \[tail -f $file\]"
    } elseif {$type == "command"} {
        if [catch {set tailFd [open "|$file" r]}] {
            SetStatus error  "can't execute command $file..."
            DeleteFromView $nick
            set curNick ""
            return
        }

        wm title [winfo toplevel $w] "Tail tool \[tail |$file\]"
    }
    fconfigure $tailFd -blocking 0
    set lines 0
    fileevent $tailFd readable "TailUpdate \$tailFd"
}
```

TailUpdate过程做为当前记录进程读状态上事件处理器的一部分被调用。在这个过程被调用的时候，它从进程那里收集输出并把它插入到文本素材里去。它还负责保证任何时刻显示在文本窗口里的文本不超过$tailSize行。

```
proc TailUpdate {fileFd} {
   global textw curNick
   global tailSize tailFd

   set w $textw
   if [eof $tailFd] {
      fileevent $tailFd readable {}
      $w insert end "Tailing \"$curNick\" done..."
      return
   }

   set line [gets $tailFd]

   $w insert end $line
   $w insert end "\n"

   set lines [lindex [split [$w index end] .] 0]
   if {$lines == [expr $tailSize+1]} {
      $w delete 1.0 2.0
   }
```

```
    $w yview moveto 1.0
}
```

Stop过程用来终止当前的logProcess操作。

```
#
# Stop the current tailing process
#

proc Stop {} {
    global tailFd
    set pid [pid $tailFd]
    catch {exec kill -9 $pid}
}
```

每当用户通过“File”菜单的“Add New . . .”菜单命令增加一个新的快捷键时，AddNew过程就会被调用。它会创建一个简单的GUI供用户创建一个新的命令快捷键。如图16-21所示。

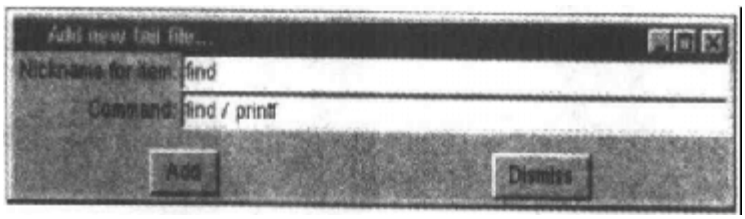


图 16-21

```
#  AddNew --
#       Add a new tail file to the system
#
#  Arguments:
#       none.
#
#  Results:
#

proc AddNew {args} {
    toplevel .addnew
    set w .addnew
    wm title $w "Add new tail file..."
    frame  $w.top
    frame $w.sep -bd 2  -relief ridge
    frame $w.bot


    set k $w.top

    label $k.name -text "Nickname for item:"
    label $k.command -text "Command:"

    grid $k.name -row 0 -column 0 -sticky e
    grid $k.command -row 1 -column 0 -sticky e

    entry $k.nameE -textvariable nameE -width 40
    entry $k.commandE -textvariable commandE -width 50

    grid $k.nameE -row 0 -column 1 -sticky ew
    grid $k.commandE -row 1 -column 1 -sticky ew

    grid columnconfigure $k 1 -weight 1
    grid propagate $k 1

    pack $w.top  -side top -fill both -expand 1
```

```
    pack $w.sep -side top -fill x -expand 1 -pady 5
    pack $w.bot -side top -fill x -expand 1


    button $w.bot.apply -text "Add" -command "AddToView \"command\"  \"\$commandE\"
\"\$nameE\""
    button $w.bot.dismiss -text "Dismiss" -command {destroy .addnew}
    pack $w.bot.apply $w.bot.dismiss -side left -expand  1
    PositionWindow $w
}
```

SetStatus过程用来设置状态窗口里的GUI状态消息。它是一个通用性的例程，错误信息和提示性信息都要用它来显示。如果出现一个类型错误，就会在状态窗口里显示一个表示出现错误的位图。

```
proc SetStatus {type text {timer 5000}} {
    global statusText  statusImgWin
    set statusText $text
    after $timer "set statusText \"\""
    $statusImgWin config -bitmap $type
    after $timer "$statusImgWin config -bitmap \"\""
}
```

AddToView命令的作用是把一个快捷键添加到选项按钮里去。在它把新快捷键添加到选项按钮里之前，它会检查用户是否已经提供了必要的资料。如下所示：

```
proc AddToView  {type command {nick ""}}  {
    global tailOpts  viewOptMenu


     catch {Stop}
    if {$type == "file"} {
        set nick $command
        if {$command == ""} {
            SetStatus error "Please supply File name..."
        }
    } elseif  {$type == "command"} {
        if {($nick == "") || ($command == "") } {
            SetStatus error "Please supply both nick and command names..."
            return
        }
    }

    set l [list  "$type" "$nick" "$command"]
    if ![info exists tailOpts(wins)] {
        set tailOpts $l
        return
    } else {
        foreach item $tailOpts(wins) {
            if {$nick == [lindex $item 1]} {
                if {$type == "file"} {
                    SetStatus info "File $nick is all ready in the tail list...."
                } else {
                    SetStatus info "Nick $nick is all ready in the tail list...."
                }
                return;
            }
        }
        lappend tailOpts(wins) $l
    }

    UpdateOptionMenu
    $viewOptMenu invoke end
}
```

DeleteNicks例程将创建出一个简单的基于列表框的用户操作界面如图16-22所示，用户通过它来删除不再想要的快捷键。

图 16-22

```
proc DeleteNicks {} {
   global   tailOpts

   if ![info exists tailOpts(wins)] {
      SetStatus info "No entries to delete..."
      return;
   }
   if {$tailOpts(wins) == {}} {
      SetStatus info "No entries to delete..."
      return;
   }
    catch {destroy .delent}
   toplevel .delent
   set w .delent
   wm title $w "Delete Entry"

   scrollbar $w.h -orient horizontal -command "$w.list xview"
   scrollbar $w.v -orient vertical -command "$w.list yview"
   listbox $w.l -selectmode single -width 20 -height 10 \
        -setgrid 1 -xscroll "$w.h set" -yscroll "$w.v set"

    frame $w.buts
    button $w.buts.d -text "Delete" -command {
       set index [.delent.l curselection ];
       if {$index == ""} {return}
       set sel [.delent.l get $index ];
       puts "sel $sel ; index $index"
       DeleteFromView $sel;
       .delent.l  delete $index
   }

   button $w.buts.dismiss -text "Dismiss" -command "destroy $w"

   grid $w.l -row 0 -column 0 -columnspan 2 -sticky "news"
   grid $w.v -row 0 -column 2 -sticky "ns"
   grid $w.h -row 1 -column 0 -columnspan 2 -sticky "we"
   grid $w.buts -row 2 -column 0 -columnspan 3

   pack $w.buts.d $w.buts.dismiss -side left -padx 10

  foreach ent $tailOpts(wins) {
    $w.l insert end [lindex $ent 1]
```

```
    }

    PositionWindow $w
}
```

DeleteFromView例程是一个内部例程，作用是从数据结构里删除指定的快捷键并刷新选项按钮optionbutton。如下所示：

```
proc DeleteFromView {nick} {
   global tailOpts

    if {$nick == ""} {
        return
    }
   set newList {}
   if ![info exists tailOpts(wins)] {
      return
   }
   foreach item $tailOpts(wins) {
      if {$nick != [lindex $item 1]} {
         lappend newList $item
      }
   }
   set tailOpts(wins) $newList
   UpdateOptionMenu
}
```

PositionWindow例程负责把一个顶层对话框居中摆放在它的父窗口里。它负责把对话框摆放在主窗口里，不让它跑到显示器屏幕的其他角落去。

```
#
# PositionWindow  --
#
#       Position the toplevel window centered to its parent.
#
#  Arguments:
#       toplevel window.
#
#  Results:
#       Positions the window
#
proc PositionWindow {w} {
   set paren [winfo parent $w]
   wm iconify $w
   set parenConf [wm geometry $paren]
   set parenConf [split $parenConf {+ - x}]
   set winConf [split [wm geometry $w] {+ - x}]
   set X [expr [lindex $parenConf 2] + [lindex $parenConf 0]/2 - \
              [winfo reqwidth $w]/2]

   set Y [expr [lindex $parenConf 3] + [lindex $parenConf 1]/2 - \
              [winfo reqheigh $w]/2]
   wm geometry $w +$X+$Y
   wm deiconify $w
}
```

UpdateOptionMenu命令用当前活跃的快捷键名单刷新那个快捷键选项菜单素材。如下所示：

```
#  UpdateOptionMenu --
#
#
#
#  Arguments:
#
```

```
#
#  Results:
#

proc UpdateOptionMenu {} {
   global tailOpts curNick viewOptMenu

   $viewOptMenu delete 0 end
   if ![info exists tailOpts(wins)] {
      return
   }
   if {$tailOpts(wins) == {}} {
      set curNick ""
      return
   }
   foreach item $tailOpts(wins) {
       $viewOptMenu add command -label [lindex $item 1] -command "catch Stop; TailFi
\"[lindex $item 0]\" \"[lindex $item 2]\" \"[lindex $item 1]\" "
      set curNick [lindex $item 1]
   }
}
```

最后，我们建立起主窗口。如下所示：

```
wm withdraw .

toplevel .t

BuildTailGui .t
```

当你运行这个程序的时候，应该看到类似于图16-23这样的窗口。

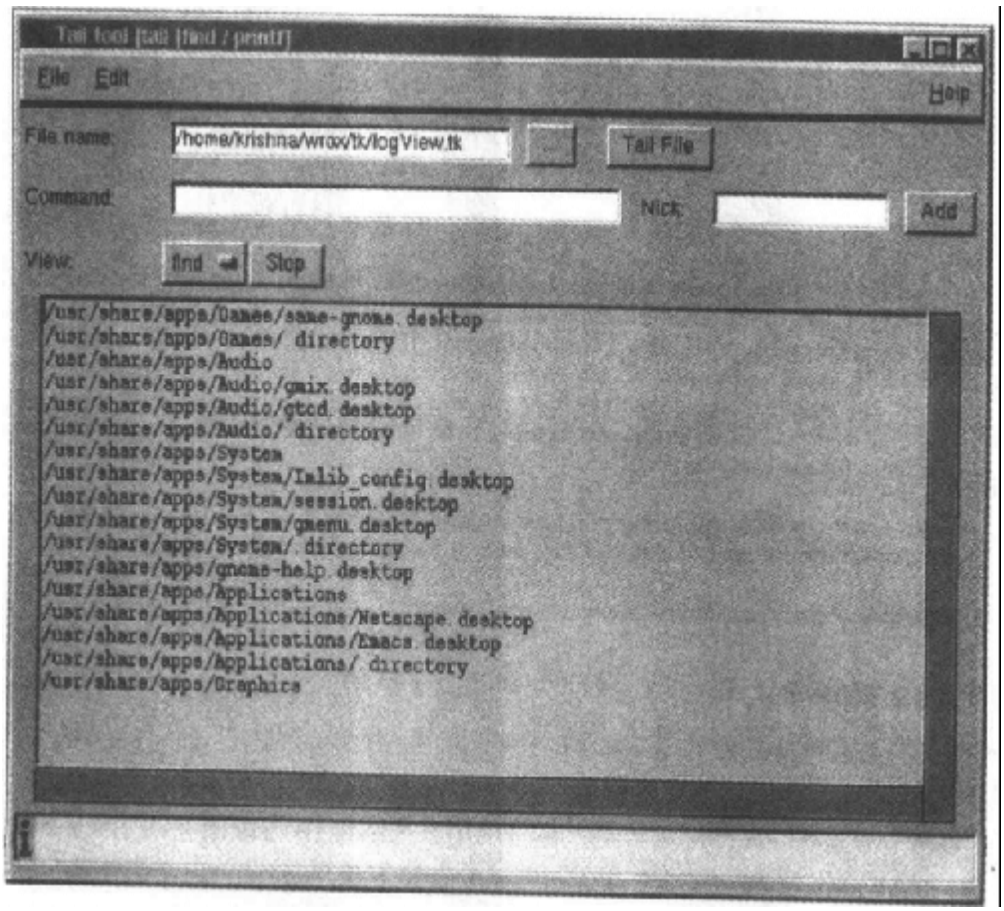


图 16-23

16.8.1 国际化

对Tk素材我们已经讲得不少了，但还是有许多论题没有涉及到。其中之一就是所谓的国际化问题。Tcl 8.1增加了许多新的功能，比如Unicode支持、创建和访问消息目录的函数（这样你就可以把你所有对话框的文字保存为多种语言）、对不同语言编码方案和通用性字符串处理的支持等。从Tcl 8.1开始，Tcl对字符串进行处理的所有函数都能接受和返回按照UTF-8格式来编码的Unicode字符串了。但因为Unicode/UTF-8编码工作是在Tcl内部实现的，所以从脚本里是看不出Tcl 8.0和Tcl 8.1在字符串处理方面有什么不同。事实上，Tcl 8.1里的所有命令都能够无缝地处理Unicode了。举个例子，你可以把一个按shiftjis编码方案编码的文件读到Tcl里，Tcl的读命令会把shiftjis编码方案自动地转换为它自己的UTF-8编码方案。请看下面这段代码：

```
set fd [open $file r]
fconfigure $fd -encoding shiftjis
set jstring [read $fd] close $fd
close $fd
```

更进一步，Tcl 8.1里新实现的规则表达式解决方案能够全面处理Unicode字符。

因为Tcl里的所有字符串都是按Unicode编码的了，Tk素材在以特定字体显示字符时也就能自动地做好必要的编码方案转换工作了。请看下面这个小代码段：

```
set str "\u592a\u9177"
??
% button .b -text $str
.b
% pack .b
```

它将把按钮标题签上的“Tcl”显示为中文字样——当然你必须安装有正确的X字体才能看到对应的中文字。如果你为素材设定的主字体里没有你想显示的某个特定Unicode字符的字模，Tk会尽量找出一个能够显示它的字体来。只要有可能，Tk就会尽量找出一个与素材的主字体各方面特性（比如宽窄、斜度等等）匹配最多的字体来。一旦Tk找到了这样的字体，它就会用新字体把那个字符显示出来。换句话说，素材使用主字体来显示所有它能够显示的字符，只有在必要是才使用后备字体。

国际化是一个很引人入胜的话题，但我们这一章里没有足够的篇幅了。下面这个站点是了解Tcl/Tk的国际化问题的好地方：

```
http://www.scriptics.com/services/support/howto/i18n.html
```

16.8.2 业界动态

只要你在使用Tk（包括man使用手册页）时遇到了麻烦，就总是可以去“Tk Widget Tour”（Tk素材之旅）看看，那里面有各种Tk素材用法的示例。敲入“widget”就可以运行这个程序。

用Tk编写的比较著名的软件程序包括Xadmin、Exmh、ical、TkMan、TkElm、TkWWW和SurfIT等。TkWWW是一个HTML编辑器，你可以用它来制作3W万维网上的主页；SurfIT是一个用Tcl编写出来的Web浏览器，它具备从Web主页上直接下载和运行Tcl程序的能力。当然，如果对方是一个你不熟悉的主机，这就会是一个有危险性的功能！ical是一个基于X的日历程序。

在我们编写本章的时候，在Tk世界里还有许多事情正在发生，所以我们下面就对几个比较值得注意的项目进行简单的介绍。

1. Tix

Tix用超过40个专业的Motif风格的素材对Tk进行了扩展。Tix素材的功能十分强大，甚至能够直接支持Motif 2.0的运行。Tix的网址是http：//www.xpi.com/tix/。

2. [incr Tk]

[incr Tcl]和[incr Tk]构成了一个面向对象的Tcl/Tk扩展。它的3.0版本最近刚刚发布，你可以在Web主页http：//www.tcltk.com/itcl/上找到它们。

[incr Tcl]提供了建立大型Tcl/Tk应用程序所额外需要的语言支持。它引入了对象的记号，把对象做为一个应用软件的建筑基石。每个对象都是一个封装起来的数据包，它带有一系列过程——或者叫做“方法”，对象的处理将由这些方法来实现。对象被组织为“类”（class），同一个类里的对象都有相同的特性；而类又可以彼此继承各自的功能。这种面向对象的做法在基本的变量/过程等元素上增加了新的组织功能，使最终得到的代码既易于阅读和理解又易于管理和维护。

[incr Tk]是一个使用[incr Tcl]对象系统构建复合素材的框架结构。复合素材是高端的素材，比如一个文件浏览器或一个表格记事本等，但在使用中可以把它们当作简单的Tk素材一样来对待。复合素材本身是用基本Tk素材构建出来的复合物，构建时不再需要使用C语言代码。从使用效果上看，复合素材与Tk素材的外观和行为都是一样的，但用在程序设计当中就简单的多了。

3. BLT

BLT-2.1工具包用许多新素材扩展了TK，如可以用来绘制线图和直方图的blt_graph素材、blt_htext超文本素材，以及用于后台执行情况的各种素材等。BLT工具包的主页地址是：http：//www.tcltk.com/blt。

我们最后要说的是，comp.lang.tcl和comp.lang.tcl.announce是提出Tk问题的最佳场所。那里的人们都是很友好的，肯定会有人愿意解答你的问题。但在把你的问题张贴到comp.lang.tcl上之前，最好先查查它的常见问题答疑（FAQ）目录，这是新闻组里的定期稿件。

在Tcl的新家，Scriptics公司的Web主页http: //www.scriptics.com/resource上你可以找到大量的Tk资源。

我们对Tk的学习到这里差不多就结束了，但对Tk来说，这还只是个简单的开头而已，因为John Outsterhout把Tcl/Tk设计为一种可扩展和可嵌入的工具。Tcl命令和Tk素材都是用C语言编写的，你可以编写出自己的东西并把它们添加到其中，也可以坐享因特网上的素材扩展。这是一个高级论题，因为篇幅的原因我们就不再这里讨论它了；但这将是你从自己的图形化Tcl/Tk前端访问C语言程序的手段之一。具体内容请参考John Outsterhout的论著。最后，祝大家学习和使用Tcl/Tk愉快！

16.9 本章总结

我们在这一章里对X窗口程序设计进行走马观花式的简单学习。

在介绍了X窗口的基础知识以及它不同的实现方法之后，我们对Tk进行了学习，它是对第15章学习的Tcl语言的补充。利用Tk丰富的素材库，我们可以为我们的应用程序快速开发出GUI前端来。

我们将在下一章里学习一种用C语言设计图形化软件程序的新方法——GNOME开发工具包GTK+。

第17章　使用GTK+进行GNOME程序设计

我们将在这一章里向大家介绍Linux世界令人激动的软件开发成果：GNOME。GNOME的意思是“GNU Network Object Model Environment”（GNU网络对象模型环境），虽然它的名字比较长，但这个软件项目的目的却很简单——那就是实现一个用户友好的、功能强大的用户和开发桌面环境，而这一环境又完全是以免费的开放源代码软件为基础的。因为篇幅上的限制，大家会发现这一章介绍的内容都不是很深入，但你们可以把下面这些论题做为GNOME程序设计的出发点，然后利用在线文档做进一步的学习。我们将在这一章里讨论的问题有：

- GNOME的简单介绍。
- GNOME的体系结构。
- GNOME桌面。
- GTK+素材库。
- GNOME素材。
- 一个GNOME软件。

GNOME项目的主页是www.gnome.org，大家可以在那里找到最新的新闻，查阅完整的在线文档，并且能够找到大量的GNOME程序设计参考资料。

Red Hat和Debian两家的新版Linux发行版本目前安装的缺省桌面系统就是GNOME，但你可以从www.gnome.org上下载与大多数Linux和UNIX系统兼容的GNOME桌面环境的二进制代码和源代码。

17.1　GNOME简介

在提到GNOME的时候，最困难的可能就是说清楚它到底是什么和它能干些什么。对普通用户来说，把GNOME看做是各种桌面工具的一个集合可能更好理解一些。但正像它名字说的那样，GNOME既是一个桌面，也是一个完整的程序设计环境，面它的内涵远远超过一个吸引人的用户操作界面。

GNOME是底层X窗口系统和高层窗口管理器软件中间的一个程序设计界面，它向GUI程序员提供了丰富的功能强大的开发工具，面这些工具是传统意义上的Linux所缺乏的。因为GNOME只是增加了资源，所以即使程序不是为GNOME开发的，一般也不存在兼容性方面的问题。特别值得一提的是，GNOME对UNIX程序员在以下四个领域遇到的问题提出了很好的解决方案：

- 在编写应用程序时，缺乏用来保持用户操作界面统一性的集成性框架。
- 缺乏应用程序间的通信手段。
- 缺乏适用于打印操作的标准。

• 缺乏适用于会话管理的标准。

在开始学习如何利用GNOME解决这些问题之前，我们先来看看这个项目的发展历史。

GNOME项目开始于1997年的秋季，主要发起人员有：Peter Mattis和Spencer Kimball（他们两人是“GNU Image Manipulation Program”即GIMP软件的开发人员）、自由软件基金会（Free Software Foundation 简称FSF）的Richard Stallman以及Red Hat公司的Erik Troan和Mark Ewing。

GNOME的原动力最初起源于对另一个Linux桌面环境项目KDE许可证问题的不满。KDE（K Desktop Environment，K桌面环境）是以TrollTech公司开发的一个名为Qt的素材开发工具包为基础的。Qt不采用GPL许可证制度，许多人认为使用一个不遵守GPL游戏规则的工具包来开发一个Linux桌面对Linux来说是一种倒退，因为其传统是要在非专利的、自由的（意思是“你可以按自己的意愿来使用软件”）开放源代码软件基础上进行程序开发。

这队程序员摈弃了KDE，开始在GTK+的基础上编写GNOME。GTK+是一个遵守GPL制度的开发工具包，没有Qt上面的那些限制。经过了十八个月，GNOME的1.0版终于面世了。

KDE也有众多的追随者，在与GNOME的竞争中同样继续发展壮大着。具体选用哪一种纯粹是个人口味方面的问题。如果你是个忠实的C++拥护者，那么，因为它是Qt的“母语”，你可能会更喜欢KDE一些。你可以在一个系统上同时安装GNOME和KDE两套软件——它们完全可以做到和平共处；而你只需编辑一下自己的X启动脚本，就可以选择加载执行它们中的一个。KDE的详细情况可以从该项目的主页www.kde.org上查到。

17.1.1 GNOME的体系结构

GNOME中的基本工具包是GTK+（即GIMP工具包），它最初是为了简化GIMP的开发工作而编写出来的。GTK+是一个性能优异的、面向对象的、跨平台的、不依赖于具体语言的开发工具包，在GNOME出现之前就已经大量地用来开发各种软件程序了。选择GTK+做为GNOME开发工具包的原因之一是它能够支持许多种程序设计语言，其中包括C、C++、TOM、PERL、Python、GUILE、ADA等等。我们将在这一章里坚持使用C语言，它也是最流行的语言。GTK+和GNOME都是用C语言编写出来的，好的理由有很多，但要求成品必须做到面向对象多少算是个强制性因素。选用C语言也没什么不好，但与用C++编写的程序比起来，我们的代码的整齐程度要稍差一点。

GTK+、Qt或Tk等工具包都是由各种素材（按钮、菜单、对话框等GUI对象）以及各种通用支持函数构成的集合。

GTK+使用GLIB（GIMP库）和GDK（GIMP Drawing Kit，GIMP绘图工具包）系列的开发库。GLIB定义了数据类型，提供了错误处理和内存管理方面的函数；而GDK则是本地图形化API和GTK+中间的一个过渡层，它需要依赖具体的计算机平台。因此，向其他计算机平台上移植GTK+只需要重新编写GDK。

GNOME对GTK+的扩展在于增加了一个由独立的GNOME专用库和GNOME素材构成的层面，它对某些GTK+库和GTK+素材做了替换和改进，目的是提供更方便的程序设计手段和鼓励应用程序开发人员编写出稳定一致的用户操作界面。

举个例子，GNOME提供了现成的对话框，我们只需加上自己的文字并指定相应的反馈按钮

就完成了设计。所以对话框代码编写起来更迅速更简单了，而它们在所有GNOME应用程序里也都有了统一的设计。

GTK+和GNOME都向我们提供了创建某些素材（比如菜单和工具条）的例程，但在遇到这种情况时我们将只讨论GNOME方法，因为它毫无疑问地是一种功能更强的选择。

我们先把GNOME体系结构的特性总结一下：

- 它具有功能强大、品种丰富的素材库。
- GNOME桌面的前端提供了控制板、控制中心、文件管理器等友好而又易于使用的应用程序。
- 它定制实现了一个名为ORBit的CORBA系统，使软件对象之间能够有效地相互通信。
- 它具有一个名为Bonobo的框架结构允许在应用程序里嵌入文档，它类似于微软公司提出的“对象链接和嵌入”（Object Linking and Embedding，简称OLE）方案。举个例子，它允许把一份电子表嵌入到一个文本文档里去。
- 它是一个完整的国际化系统，带有对Unicode编码方案和Complex Text Processing（复杂文本处理）系统的支持，而这些是显示中东和南亚地区文字时所必需的。
- 具有输出打印框架结构。

GNOME确实很棒！我们现在先把复杂的东西抛在脑后，来看看我们将在屏幕上看到的东西——桌面（如图17-1所示）。

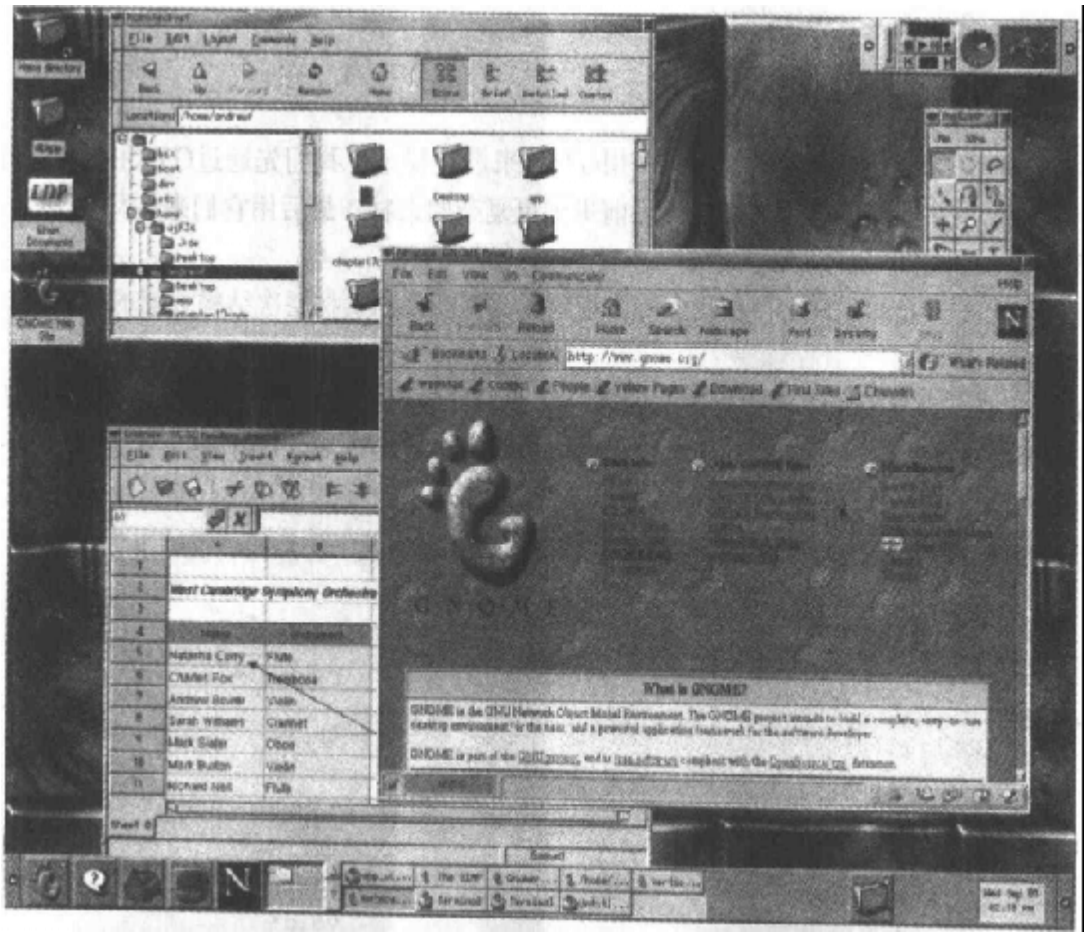


图 17-1

17.1.2 GNOME桌面

当我们把GNOME和所有必要的库文件正确地安装好以后（X启动脚本的最后一行必须是“exec gnome-session”。根据你使用的计算机平台和它具体的配置情况，X启动脚本可能是~/.xinitrc、.xsession或.Xcliebts这几个脚本之一），我们就可以用下面这个熟悉的命令启动桌面了：

```
$ startx
```

熟悉桌面最好的办法就是使用它！你很快就会发现它的设计是多么的优秀，它对用户又是多么的友好，而且，它又是多么巨大地改善了Linux的使用性。

GNOME桌面的核心是它的控制板，这是一个配置性非常好的任务条。各个独立的控制板可以被放置在屏幕的任意边角，程序的快捷键、菜单的快捷键和小的应用程序（applet，人们经常称之为“插件程序”）都可以放置在控制板里。点击控制板两端向外的箭头可以把它隐藏起来，它们滑出屏幕的姿态是极其顺畅，就像动画一样——你肯定会百看不厌！

你当然可以随心所欲地选择与GNOME一起使用的窗口管理器，它们的作用是控制窗口在屏幕上的摆放位置和外观，但其中有几个比其他的更符合GNOME的要求。最流行的选择是“Enlightenment”和“IceWM”。这两个都是百分之百符合GNOME要求的，你可以通过它们提供的数不胜数的选项对自己桌面的外观进行定制。

GNOME的文件管理器是GNOME版的“mc”（midnight commander）程序。MS Windows用户对文件目录树和完全的拖放功能肯定会觉得如鱼得水。我们可以方便地完成文件的拷贝、移动、换名、改变文件属主和优先级等操作。

17.1.3 在GNOME里利用GTK+设计程序

我们现在开始最激动人心的部分——用GTK+来设计程序！我们先通过GTK+基本素材的使用示例来学习一些必要的基本知识，然后前进到更复杂的素材，最后用它们来编写一个GNOME应用程序。

在上路之前，我们先来看一些我们将会用到的数据类型和素材层次结构方面的概念。

1. 数据类型

GLIB定义有自己的基本数据类型（datatype）集合，它们中的大多数都有直接对应的C语言标准数据类型。这使得不同计算机平台之间的代码移植工作更容易实现；并且在某些情况下——比如用gpointer代替void*的情况下，它可以改进程序的可读性，使之更容易理解。坚持使用这些新的数据类型可以保证我们的代码即使在其底层实现发生了变化的情况下仍然能够继续工作（见表17-1）。

表 17-1

GLIB类型	C语言类型
gchar	char
gshort	short
glong	long
gint	int
gboolean	char
gpointer	void*

2. 素材的层次结构

GNOME和GTK+中的素材都是一个层次结构上的成员，这样对一组素材都适用的函数（比如gtk_widget_show函数）就只需要被实现一次。为了尽量减少重复性的代码编写工作，新的素材是从现有素材推导出来的，程序员只需要编写出新素材的新功能就大功告成。我们来看看GnomeApp素材的素材层次结构，它是GNOME中的顶层窗口素材，下面是我们节选自“Gnome User Interface Library Reference Manual”（《GNOME参考手册——用户操作接口库》）中的内容：

```
GtkObject
      +----- GtkWidget
                +------GtkContainer
                            +-----GtkBin
                                      +---  -GtkWindow
                                                  +-----GnomeApp
```

它告诉我们GnomeApp素材是从GtkWindow素材推导出来的，而GtkWindow素材又是从GtkBin素材里推导出来的，依次类推。根据这个层次结构，我们就可以把GtkWindow、GtkBin、GtkContainer、GtkWidget和GtkObject的函数用在GnomeApp素材上。

用来创建素材的一切函数，比如gnome_app_new函数等，返回的都是一个GtkWidget指针，GtkWidget是最基本的素材。这就意味着如果我们想调用GnomeApp专用的函数，比如调用gnome_app_set_menus(GnomeApp *app, GtkMenuBar *menubar)的时候，我们就必须用一个宏定义（这个例子对应的是GNOME_APP）把GtkWidget类型映射为一个GnomeApp类型。我们之所以能够这样做，正是因为GnomeApp是从GtkWidget推导出来的。如果类型映射不正确，编译器是不会发现的，但当程序运行时GTK+库会向控制台报告出现了这样的错误。GTK+在报告错误方面做得很好，对我们的调试工作有很大的帮助。我们将看到许多使用宏定义进行类型映射的例子。

现在，我们要暂时离开素材这个话题来编写我们的第一个GNOME程序，它将在屏幕上显示一个窗口。

动手试试：创建窗口

```
#include <gnome.h>

int main (int argc, char *argv[])
{
        GtkWidget *app;
        gnome_init ("example", "0.1", argc, argv);
        app = gnome_app_new ("example", "Window Title");
        gtk_widget_show (app);
        gtk_main ();
        return 0;
}
```

在开始编译示例程序的代码之前必须先做好一项重要的工作：把全部GNOME库和GTK+库都安装好，并且把这些库文件的路径都设置好。

为了简化编译工作，GNOME自己带有一个名为gnome-config的shell脚本程序，它的作用是向编译器提供编译操作所需要的正确标志，我们要用反引号（`）把它的输出追加在编译命令行的尾部。

请输入下面的命令来编译程序gnome1.c（注意反引号和双短划线！）：

```
$ gcc gnome1.c -o example1 `gnome-config --cflags --libs gnomeui`
```

编译完成后请在一个GNOME桌面的终端窗口里执行它。

如果你在对程序进行编译时遇到了麻烦，可以在www.gnome.org站点查到常见问题的详细原因和解决办法。

这个示例程序在运行时会弹出一个长宽为200×200点阵的空白窗口，如图17-2所示。

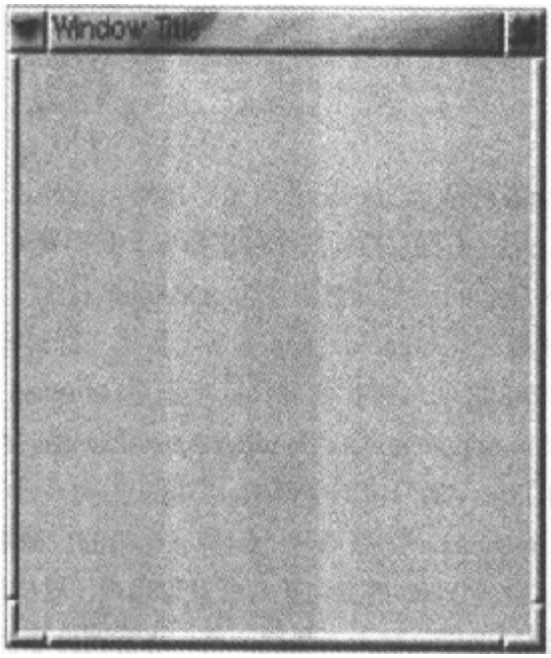


图 17-2

你可以对这个窗口进行移动、尺寸调整和关闭等操作，但它并不会让我们返回到shell提示符去，这是因为我们还没有给它设置一个退出动作的处理器。

操作注释：

我们来一行一行地解释这段代码。首先，我们用“# include <gnome.h>”语句给程序添加上必要的头文件，它将负责处理好一切与GNOME库和GTK+库有关的定义。在main函数里，我们声明了一个指向我们窗口对象的GtkWidget指针app，然后调用了gnome_init函数，它的作用是对各个库进行了初始化、设置对会话的管理和加载用户的首选设置。我们给gnome_init传去这个应用程序的id（本例中是“example”）、它的版本号“0.1”以及main函数接收到的命令行参数；这都是供GNOME内部使用的。

接下来我们调用gnome_app_new创建出我们的窗口。这个函数需要的参数有：应用程序的名字，它可以与在gnome_init函数里给出的字符串不同（它也是内部使用的）；窗口的标题，我们这个例子里用的是“window title”，但它可以是NULL。千万不要被这个函数的名字给弄糊涂了，它的作用是创建一个顶层窗口而不是一个“新应用程序”，我们每次想创建一个新窗口时都要调用gnome_app_new。为了能够在屏幕上看见我们创建的窗口，我们又调用了gtk_widget_show函

数。最后，我们通过gtk_main把控制移交给GNOME，这样才能对各种事件、鼠标点击、按钮动作等进行响应和处理。

一旦调用了gtk_main就再也回不来了——我们将只能通过回调函数（callback）把用户事件和我们的程序中的函数连接起来以完成进一步的动作。我们将在下一个例子里看到几个回调函数。

动手试试：无处不在的“Hello World”

在第一个例子里加上下面带阴影的内容或做相应的修改：

```
#include <gnome.h>

static void button_clicked(GtkWidget *button, gpointer data)
 {
         char *string = data;
         g_print(string);
 }

int main (int argc, char *argv[])
{
         GtkWidget *app;
         GtkWidget *button;

         gnome_init ("example", "0.1", argc, argv);
         app = gnome_app_new ("example", "Window Title");

         button = gtk_button_new_with_label ("Hello,\n GNOME world!");

         gtk_signal_connect (GTK_OBJECT (app), "delete_event",
                                    GTK_SIGNAL_FUNC (gtk_main_quit),
                                    NULL);

         gtk_signal_connect (GTK_OBJECT (button), "clicked",
                                    GTK_SIGNAL_FUNC (button_clicked),
                                    "Ouch!\n");

         gnome_app_set_contents (GNOME_APP (app), button);
         gtk_widget_show_all (app);
         gtk_main ();
         return 0;
}
```

运行这个程序的时候，我们将看到一个如图17-3所示的窗口。

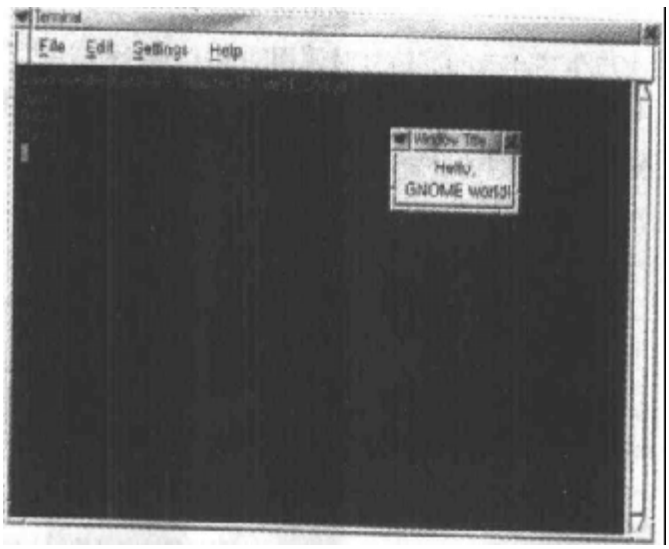


图 17-3

操作注释：

我们使用gtk_button_new_with_label函数创建了一个带标题签的按钮并通过gnome_app_set_conents把它放在了窗口里，我们用GNOME_APP宏定义把GtkWidget指针映射为GnomeApp类型。

我们还注册了两个回调函数，其中一个通过调用GLIB的g_print函数（这个库函数通常用于调试目的，它与printf的不同之处是它的输出可以被覆盖）在按钮每次被点击时输出字符串“Ouch!”；第二个会在窗口被关闭时调用gtk_main_quit函数使我们的程序干净利落地退出运行。

在我们的例子里省略了对GNOME函数返回结果的检查，但在成熟的窗口环境里，那些“重量级”应用软件必须对它们进行检查！

下面来仔细研究一下程序里的信号处理器函数。

3. 信号和回调函数的处理

每当鼠标移动、鼠标进出素材、按钮被按下、开关按钮切换了状态、菜单项被选取等事件发生的时候，就会有一个信号被送到应用程序，然后信号再被传递到一个回调函数去（GNOME信号与我们在第10章里介绍的UNIX信号可不是一回事）。应用程序对大多数信号都没有什么兴趣，但一般都要对其中的几个进行处理并采取一定的行动。一个绘图程序可能需要在以后按下每个鼠标按钮时绘制一条直线，而一个音响播放程序可能需要在一个滑块素材被调节时增减输出音量。在GNOME和GTK+里，我们通过调用gtk_signal_connect函数把信号和处理器函数联系在一起。下面是它的语法定义：

```
guint gtk_signal_connect (GtkObject *object , const gchar *name, GtkSignalFunc  func,
gpointer data)
```

它有四个参数，这四个参数告诉GNOME和GTK+这个回调函数是与哪一个素材关联着的、要对哪一个信号进行处理、信号出现时将要被调用的函数是哪一个，以及将要传递给信号处理函数的任何其他数据。如果我们现在回过头来再看看第一个例子，就可以明白其中的下面这个调用是什么意思了：

```
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                                    GTK_SIGNAL_FUNC (button_clicked),
                                    "Ouch!\n");
```

它会在按钮素材每次产生“clicked”信号时调用button_clicked，把“Ouch!”做为参数发送出来。注意GTK+把信号看做是字符串，不像普通的信号处理过程那样把它看做是常数，因此GTK+里的信号拼写错误将在运行时才会被捕捉到。

不同的素材会发送出不同的信号来，与按钮有关的信号见表17-2：

表 17-2

信　号	动　作
clicked	按钮被点击（即按下和释放）
pressed	按钮被鼠标按下
released	按钮被释放
enter	鼠标移动进入按钮上空
leave	鼠标移动离开按钮上空

这些全都是为GNOME用户操作界面编写程序的基本要素。在本章其余的例子里，我们将对一些更复杂的素材进行学习。

不要惊讶于GTK+和Tk有那么多的相似之处。这可不是巧合！

在“Hello World”例子里，窗口会收缩为只包容着那个按钮素材，这也许正是你的预期效果。事实上，窗口只能有一个子素材，如果你试图创建并在窗口里增加一个按钮，它就会取代第一个按钮。大家可能会问了：“我怎样才能在窗口里放置不止一个的素材呢？”这个问题把我们带到包容器概念上来了。

4. 包容器

GTK+是一个基于包容器的工具包，即只有给素材指定一个父包容器才能把它们摆放到窗口里去。窗口是一个单素材包容器，因此GTK+需要使用看不见的“包装箱”（packing box）来容纳多个素材以创建窗口的布局。包装箱就是装素材的“箱子”，分水平摆放和竖直摆放两种，分别由gtk_hbox_new和gtk_vbox_new创建。如果想把素材放到这些包装箱里去，我们只需为每个素材调用gtk_box_pack_start函数并指定几个格式选项就行了（如图17-4所示）。

下面的代码段演示了创建一个竖直包装箱并在其中放置两个标题签素材的操作过程：

```
GtkWidget *vbox, *label;
gboolean expand = FALSE;
gboolean fill = TRUE;
gint padding = 2;

vbox = gtk_vbox_new (homogeneous, spacing);

label = gtk_label_new ("This is the top label widget");
gtk_box_pack_start (GTK_BOX (vbox), label, expand, fill, padding);

label = gtk_label_new ("This is the bottom label widget");
gtk_box_pack_start  (GTK_BOX (vbox), label, FALSE, TRUE, 2);
```

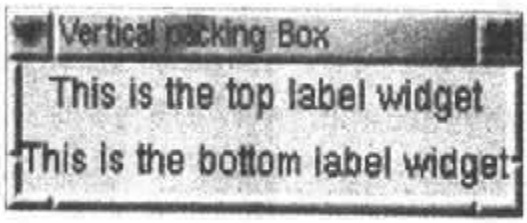


图 17-4

我们对第二个标题签使用了expand、fill和padding属性的典型值。它们的含义见表17-3中的解释。

表 17-3

参 数	类 型	说 明
homogeneous	gboolean	使包装箱里的所有素材都占据与箱里最大的那个素材同样的空间
spacing	gint	确定邻接素材之间的间隔
expand	gboolean	允许包装箱延展到填满剩余的空间。如果设置了homogenous标志，这个标志的设置情况将被忽略
Fill	gboolean	允许每个特定的素材延展到填满剩余的空间
padding	gint	确定素材框的宽度

从现在起，我们将在所有的例子里使用包装箱，就从下面这个介绍更多按钮素材的程序开始吧。

5. 按钮

GTK+提供了四种按钮，它们是：简单的下压按钮、开关按钮、复选框和单选按钮。

GNOME里的按钮与其他GUI里的按钮在基本功能方面都是一样的。下压按钮用来实现一个点击动作；开关按钮和复选框是带有一个关联状态的按钮——开或关，选中或没有被选中；而单选按钮构成了一组选项，每次只能有其中的一个按钮被选中。

我们用下面的例子介绍一下复选框和单选按钮的用法。

动手试试：更多选择

1) 先定义素材，并用GSList创建一个空着的单选按钮组。如下所示：

```
#include <gnome.h>

int main (int argc, char *argv[])
{
GtkWidget *app;
        GtkWidget *button1, *button2;
        GtkWidget *radio1, *radio2, *radio3, *radio4;
        GtkWidget *vbox1, *vbox2;
        GtkWidget *hbox;
        GSList *group = NULL;

gnome_init ("example", "0.1", argc, argv);
        app = gnome_app_new ("example", "Music choices");
```

2) 我们给窗口设置一个边界，并创建出我们的包装箱。如下所示：

```
gtk_container_border_width (GTK_CONTAINER (app), 20);

        vbox1 = gtk_vbox_new (FALSE, 0);
        vbox2 = gtk_vbox_new (FALSE, 0);
        hbox =  gtk_hbox_new (FALSE, 0);
```

3) 我们在第一个竖直包装箱里放上两个复选框，如下所示：

```
button1 = gtk_check_button_new_with_label( "Orchestra");
gtk_box_pack_start (GTK_BOX (vbox1), button1, FALSE, FALSE, 0);

button2 = gtk_check_button_new_with_label ("Conductor");
gtk_box_pack_start (GTK_BOX (vbox1), button2, FALSE, FALSE, 0);
```

4) 然后在第二个包装箱里放上四个单选按钮，把按钮逐个地添加到组里去。如下所示：

```
radio1 = gtk_radio_button_new_with_label (group, "Strings");
gtk_box_pack_start (GTK_BOX (vbox2), radio1, FALSE, FALSE, 0);
group = gtk_radio_button_group (GTK_RADIO_BUTTON (radio1));

radio2 = gtk_radio_button_new_with_label (group, "Wind");
gtk_box_pack_start (GTK_BOX (vbox2), radio2, FALSE, FALSE, 0);
group = gtk_radio_button_group (GTK_RADIO_BUTTON (radio2));

radio3 = gtk_radio_button_new_with_label (group, "Brass");
gtk_box_pack_start (GTK_BOX (vbox2), radio3, FALSE, FALSE, 0);
group = gtk_radio_button_group (GTK_RADIO_BUTTON (radio3));

radio4 = gtk_radio_button_new_with_label (group, "Percussion");
```

```
gtk_box_pack_start (GTK_BOX (vbox2), radio4, FALSE, FALSE, 0);
group = gtk_radio_button_group (GTK_RADIO_BUTTON (radio4));
```

5) 最后，我们增加一个退出事件的处理器，并把包装箱放到一起。如下所示：

```
        gtk_signal_connect (GTK_OBJECT (app), "delete_event",
                            GTK_SIGNAL_FUNC (gtk_main_quit),
                            NULL);

gtk_container_add (GTK_CONTAINER (hbox), vbox1);
        gtk_container_add (GTK_CONTAINER (hbox), vbox2);

        gnome_app_set_contents (GNOME_APP (app), hbox);

        gtk_widget_show_all(app);
        gtk_main();
        return 0;
}
```

包装箱的布局结构如图17-5所示。

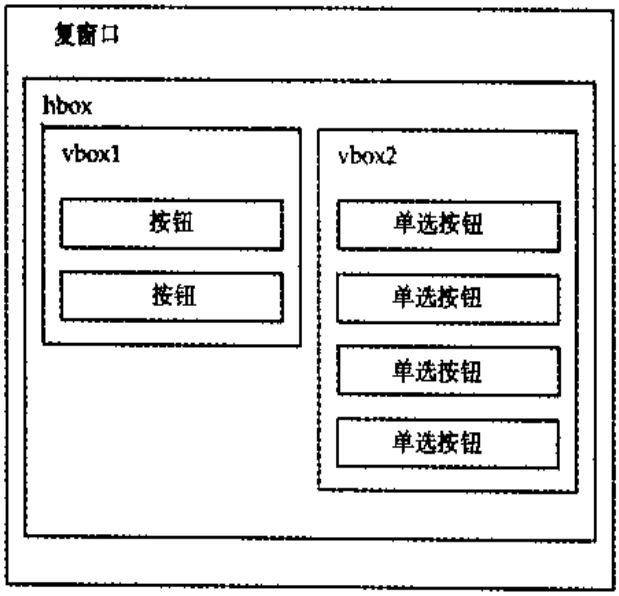


图 17-5

运行这个程序的时候，我们将看到一个如图17-6所示的窗口。

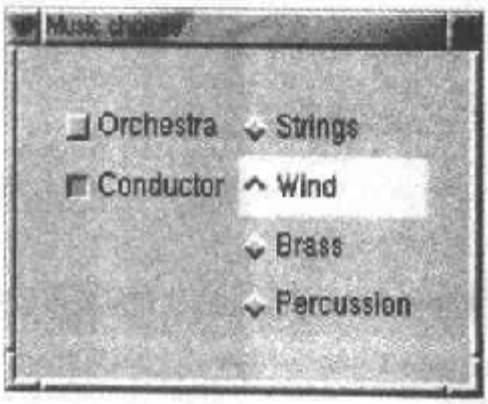


图 17-6

操作注释：

我们可以毫不脸红地说这是一个创造性的程序示例，它演示了新素材的用法和嵌套式包装箱结构，通过它们我们创建了一个窗口布局。我们创建了两个复选框并把它们添加到第一个竖直包装箱vbox1里。它们的当前状态可以通过调用gtk_toggle_get_state(button1)函数读取，该函数返回一个gboolean值，TRUE表示按钮被按下。

单选按钮们的创建工作与复选框的情况很相似。我们把单选按钮归到一个组里，在每创建一个新按钮之后就调用gtk_radio_button_group函数刷新这个组。在第一个按钮被添加到这个组里之前，它应该是空的（即它的值必须是NULL），因为这个组当时还不存在。

单选按钮是从复选框推导出来的，而复选框又是从开关按钮推导出来的，所以我们可以使用同一组函数来读取和修改它们的状态，还可以使用同样的事件。与这些按钮有关的函数的完整定义可以在www.gtk.org站点上的GTK+参考文档里查到。

(1) 输入框素材

输入框是一个用gtk_entry_new函数创建的单行文本素材，一般用在需要输入少量信息的情况。下面这个简单的程序创建了一个登录窗口，把输入框的可见性标志设置为不可见（FALSE），然后在按下回车键产生activate信号的时候输出口令字域里的内容。

动手试试：GNOME风格的登录窗口

1) 我们先定义enter_pressed回调函数，它会在每次按下回车键时被调用。如下所示：

```
#include <gnome.h>

static void enter_pressed(GtkWidget *button, gpointer data)
 {
         GtkWidget *text_entry = data;
         char *string = gtk_entry_get_text(GTK_ENTRY (text_entry));
         g_print(string);
 }
```

2) 接下来我们定义变量、初始化GNOME、再创建一个水平包装箱。如下所示：

```
int main (int argc, char *argv[])
{
GtkWidget *app;
        GtkWidget *text_entry;
        GtkWidget *label;
        GtkWidget *hbox;
        gchar *text;

gnome_init ("example", "0.1", argc, argv);
        app = gnome_app_new ("example", "entry widget");

        gtk_container_border_width (GTK_CONTAINER (app), 5);

        hbox = gtk_hbox_new (FALSE, 0);
```

3) 现在创建一个标题签，调整好它的位置并把它放到包装箱里去。如下所示：

```
label = gtk_label_new("Password: ");
        gtk_misc_set_alignment (GTK_MISC (label), 0, 1.0);
        gtk_box_pack_start (GTK_BOX (hbox), label, FALSE, FALSE, 0);
```

4) 接下来，创建输入框并把它的可见性设置为不可见，这会使其中的内容显示为星号。如下所示：

```
text_entry = gtk_entry_new();
gtk_entry_set_visibility (GTK_ENTRY (text_entry), FALSE);
gtk_box_pack_start (GTK_BOX (hbox), text_entry, FALSE, FALSE, 0);
```

5) 最后，我们设置好信号处理器，再把包装箱放置到窗口里去。如下所示：

```
        gtk_signal_connect (GTK_OBJECT (app), "delete_event",
        GTK_SIGNAL_FUNC (gtk_main_quit),
                                        NULL);

        gtk_signal_connect (GTK_OBJECT (text_entry), "activate",
                                      GTK_SIGNAL_FUNC (enter_pressed),
                                        text_entry);
gnome_app_set_contents( GNOME_APP (app), hbox);
        gtk_widget_show_all (app);
        gtk_main ();
        return 0;
}
```

编译并运行这个程序，我们将看到一个如图17-7所示的窗口。

图 17-7

(2) 列表框和列表输入框

列表框素材可以容纳一个字符串列表，根据该列表框的配置情况，用户可以在其中选择一个或者多个字符串。列表输入框是列表框的一种，它增加了一个下拉菜单，用户可以在这个菜单里进行选择；同样地，根据该素材的配置情况，用户能够进行的选择可能会局限于列表中的某几项（如图17-8所示）。

下面这个代码段演示了这两种素材的用法。完整的代码清单包括在Wrox出版社Web站点上的源代码包里。

```
listbox = gtk_list_new ();
gtk_list_set_selection_mode (GTK_LIST (listbox),
                                    GTK_SELECTION_MULTIPLE);

item = gtk_list_item_new_with_label ("Beethoven");
gtk_container_add (GTK_CONTAINER (listbox),item);

item = gtk_list_item_new_with_label ("Brahms");
gtk_container_add (GTK_CONTAINER (listbox),item);

item = gtk_list_item_new_with_label ("Bach");
gtk_container_add (GTK_CONTAINER (listbox),item);
gtk_box_pack_start (GTK_BOX (vbox), listbox, FALSE, FALSE, 0);

/* add items ad infintum */

label = gtk_label_new("Choose an era:");
gtk_misc_set_alignment (GTK_MISC (label), 0, 1.0);
gtk_box_pack_start (GTK_BOX (vbox), label, FALSE, FALSE, 0);
```

```
combolist = NULL;

combolist = g_list_append (combolist, "Renaissance");
combolist = g_list_append (combolist, "Baroque");
combolist = g_list_append (combolist, "Classical");
combolist = g_list_append (combolist, "Romantic");
combolist = g_list_append (combolist, "Impressionism");

combo = gtk_combo_new ();
gtk_combo_set_popdown_strings (GTK_COMBO(combo), combolist);
gtk_box_pack_start (GTK_BOX (vbox), combo, FALSE, FALSE, 0);
```

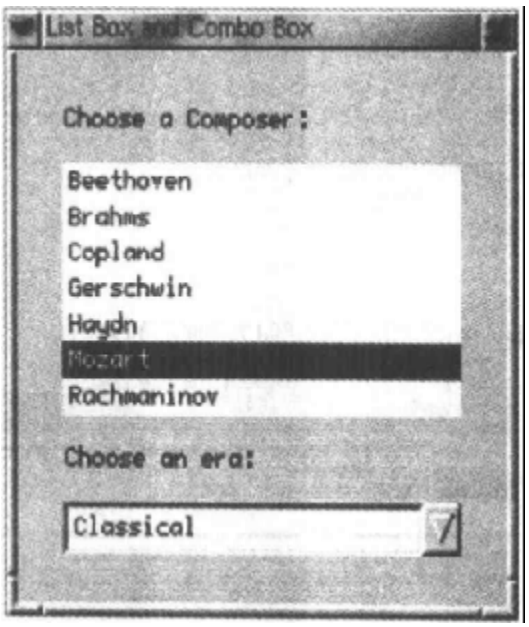


图 17-8

下面开始对GNOME的专用库进行介绍，它们包含着更为复杂的素材，这些素材会使程序员的日子好过许多。如果单纯使用GTK+，通常会有大量的重复性代码；同时，即使是创建菜单、工具条和对话框等普通的工作，GTK+里也提供了各种不同的方法。但这并不意味着GTK+的效率低下；正是这种灵活性保证了它对计算机平台的不依赖性。我们将会看到，如果我们使用了GNOME库，它将给我们的应用程序增添大量的附加功能，但因此而增加的编码工作却是很小的。

(3) 菜单和工具条

GNOME使我们能够为GnomeApp素材创建出菜单和工具条来，而它们又都可以在窗口里被最小化或从最小化状态还原。需要我们做的只是把必要的信息填写到一个数组里，然后再调用gnome_app_create_menu或gnome_app_create_toolbar函数就可以了。

菜单或工具条里通常都会有不止一个的数据项，每一个这样的数据项其属性（类型、字符串、回调函数指针、快捷键等等）都要在相应的菜单数组或工具条数组里用一个结构来进行定义。这个结构的细节请查阅libgnomeui的API参考手册。在大多数情况下，菜单项都是非常简单的，因此我们通常可以通过GNOME提供的一组宏定义来简化这个结构的创建工作。每一种常见的菜单和工具条选项都有一个对应的宏定义。下面就是它们的简单介绍。

首先是一些用来创建顶层菜单的顶层宏定义，传递给它们的数组可以包含一个或者全部表17-4中的GnomeUIInfo结构。

表 17-4

菜 单	宏 定 义
File	GNOMEUIINFO_MENU_FILE_TREE (tree)
Edit	GNOMEUIINFO_MENU_EDIT_TREE (tree)
View	GNOMEUIINFO_MENU_VIEW_TREE (tree)
Settings	GNOMEUIINFO_MENU_SETTINGS_TREE (tree)
Windows	GNOMEUIINFO_MENU_WINDOWS_TREE (tree)
Help	GNOMEUIINFO_MENU_GAME_TREE (tree)
Game	GNOMEUIINFO_MENU_GAME_TREE (tree)

在顶层菜单里面又定义了超过三十个用来创建常用菜单项的宏定义。这些宏定义可以给每个菜单项加上小图标（点图）和快捷键。我们只需要定义一个将会在该菜单项被选中时调用执行的回调函数和一个传递给那个函数的指针（格式为“(cb, data)”）就行了。我们把一些常用的宏定义列在表17-5里，一会儿就会用到它们。

表 17-5

顶层菜单	菜 单 项	宏 定 义
File	New	GNOMEUIINFO_MENU_NEW_ITEM (label,hint,cb,data)
	Open	GNOMEUIINFO_MENU_OPEN_ITEM (cb,data)
	Save	GNOMEUIINFO_MENU_SAVE_ITEM (cb,data)
	Print	GNOMEUIINFO_MENU_PRINT_ITEM (cb,data)
	Exit	GNOMEUIINFO_MENU_EXIT_ITEM (cb,data)
Edit	Cut	GNOMEUIINFO_MENU_CUT_ITEM (cb,data)
	Copy	GNOMEUIINFO_MENU_COPY_ITEM (cb,data)
	Paste	GNOMEUIINFO_MENU_PASTE_ITEM (cb,data)
Settings	Preferences	GNOMEUIINFO_MENU_PREFERENCES_ITEM (cb,data)
Help	About	GNOMEUIINFO_MENU_ABOUT_ITEM (cb,data)

工具条的情况与菜单是很相似的，我们要用GNOMEUIINFO_ITEM_STOCK(label, tooltip, callback, stock_id)宏定义来创建一个相应的数组，其中，stock_id是我们打算用为该项目图标的一个预定义图标的id值。这些预定义图标的完整清单也列在libgnomeui的参考手册里。

另外还有几个特殊的宏定义，包括GNOMEUIINFO_SEPERATOR——它的作用是创建一条用来物理性地分隔菜单项或工具条项目的线条；还有GNOMEUIINFO_END——它的作用是表示数组到此结束，等等。

我们来看看这些数组和宏定义的实际工作情况。

动手试试：菜单和工具条

1) 先来创建一个回调函数，当有项目被选中时它会输出相应的文字。如下所示：

```
#include <gnome.h>

static void callback (GtkWidget *button, gpointer data)
{
        g_print ("Item Selected");
}
```

2) 现在来创建一个有两个元素的数组，它们将被放在File菜单里。这两个元素一个是用来调用回调函数的选项，另外一个是退出选项。如下所示：

```
GnomeUIInfo file_menu[] = {
        GNOMEUIINFO_ITEM_NONE ("A Menu Item","This is the statusbar info",
                               callback),
        GNOMEUIINFO_MENU_EXIT_ITEM(gtk_main_quit, NULL),
        GNOMEUIINFO_END
};
```

3) 接下来创建菜单的结构，它只有一个顶层的文件菜单，它指向我们刚才创建的数组。如下所示：

```
GnomeUIInfo menubar[] = {
        GNOMEUIINFO_MENU_FILE_TREE(file_menu),
        GNOMEUIINFO_END
};
```

4) 工具条的情况与菜单类似。我们创建一个有两个元素的数组，它们一个是打印按钮，一个是退出按钮。如下所示：

```
GnomeUIInfo toolbar[] = {
        GNOMEUIINFO_ITEM_STOCK("Print", "This is another tooltip",
                               callback,
                               GNOME_STOCK_PIXMAP_PRINT),
        GNOMEUIINFO_ITEM_STOCK("Exit", "Exit the application",
                               gtk_main_quit,
                               GNOME_STOCK_PIXMAP_EXIT),
        GNOMEUIINFO_END
};
```

5) 最后，我们创建出菜单和工具条并把它们摆放到窗口里去。如下所示：

```
int main(int argc, char *argv[])
{
        GtkWidget *app;

        gnome_init ("example", "0.1", argc, argv);
        app = gnome_app_new ("example", "simple toolbar and menu");

        gnome_app_create_menus (GNOME_APP (app), menubar);
        gnome_app_create_toolbar (GNOME_APP (app), toolbar);

        gtk_widget_show_all (app);
        gtk_main ();
        return 0;
}
```

我们将看到一个小窗口，里面有一个菜单和一个工具条。我们可以对它们进行点击和拖放。如果我们退出后再重新进入，GNOME还会记得它们原先的位置（见图17-9）！

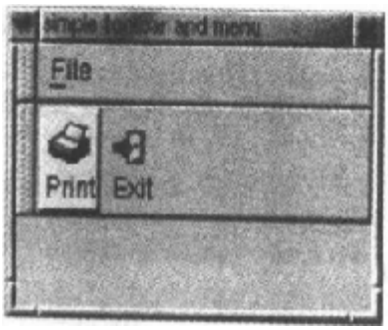


图 17-9

(4) 对话框

当我们想要在GNOME桌面环境里创建能够向用户显示文字信息的对话框时，就需要调用gnome_message_box_new函数并给它传去消息的内容文本、我们需要的对话框类型和我们想安排在对话框里的按钮，所有这些都将被放在一

个以NULL结束的列表里。接下来要把我们刚才创建的那个对话框素材的“clicked”信号和一个信号处理器绑定在一起，用户按下的按钮将以一个整数参数的形式传递给信号处理器。然后我们只需调用gtk_widget_show就可以把这个非模型化的对话框显示在窗口里了。请看下面的例子：

```
static void messagebox_clicked(GnomeDialog *dlg, gint button, gpointer data)
{
        switch (button)
        {
              case 1: /* user pressed apply*/
              return;
              case 0: /*user presser ok*/
              case 2: /*user pressed close */
              gnome_dialog_close(dlg);
        }
}

GtkWidget *dlg;

dlg = gnome_message_box_new ("This is the message text that appears in the dialog
box",
                                  GNOME_MESSAGE_BOX_QUESTION,
                                  GNOME_STOCK_BUTTON_OK,
                                  GNOME_STOCK_BUTTON_APPLY,
                                  GNOME_STOCK_BUTTON_CLOSE,
                                  NULL);

gtk_signal_connect (GTK_OBJECT(dlg), "clicked",
     GTK_SIGNAL_FUNC(messagebox_clicked),
      NULL);

gtk_widget_show (dlg);
```

这段代码创建出一个简单的提问消息框，里面有三个按钮，能够对用户的点击动作做出响应（见图17-10）。

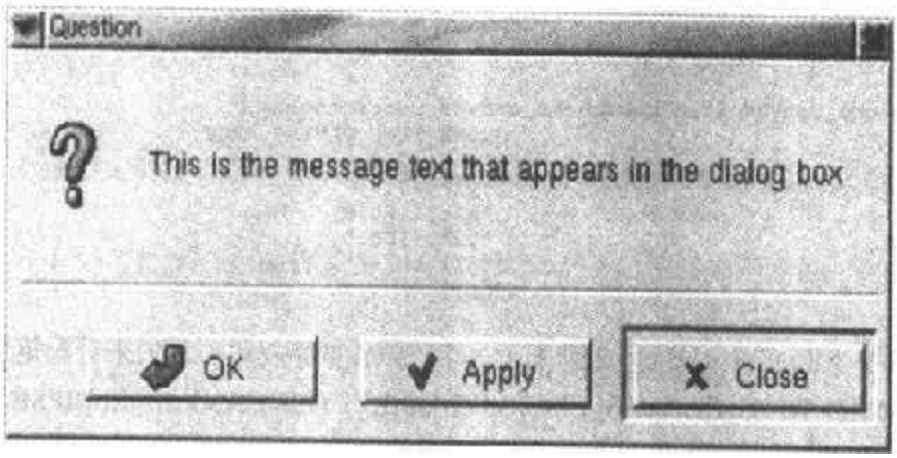


图 17-10

(5) 画布素材

画布素材是GNOME素材里最复杂又最多能的了，仅它自己就值得用一章的篇幅来介绍！GNOME画布是一种高性能的绘图素材，它被设计为应用程序的一种通用性显示引擎。它支持直线和矩形等简单的图形对象，而应用程序还可以定义自己的画布项目以提供更为复杂的显示

效果。

画布素材支持两种着色后端，一个出于速度和效率方面的考虑使用的是Xlib，另一个是基于Libart的高画面质量的抗失真引擎。

画布素材将负责其中所有项目的绘制和屏幕刷新工作，所以我们不必操心为防止出现图像闪烁而必须解决的数据缓冲问题。更让人满意的是画布中的项目本身都是素材，所以我们可以动态地重新定义它们的属性。

如果想创建基于Xlib引擎的画布，我们需要以下几个函数：

```
GtkWidget *canvas;

gtk_widget_push_visual(gdk_imlib_get_visual());
gtk_widget_push_colormap(gdk_imlib_get_colormap());
canvas = gnome_canvas_new();
gtk_widget_pop_visual();
gtk_widget_pop_colormap();
```

如果我们想在画布上放置图像，就必须调用visual和colormap函数。gnome_canvas_new用来创建我们的画布。

接下来我们需要调用gnome_canvas_pixels_per_unit来设置画布坐标单位和屏幕显示点阵的转换比例。

在创建画布项目时，我们需要给它指定一个组，让它成为这个组里的一个成员。这样就能够以组为单位进行全局性的改动，比如对一个组进行整体移动或隐藏等。

创建画布项目需要调用gnome_canvas_item函数，它的参数是一个以NULL结束的参数列表。列表里的第一项是它的父组名称，然后是一个宏定义——它定义了我们将要创建的对象的类型（矩形、直线、椭圆，等等），接下来是该对象特有的各种属性。

我们来看看怎样才能创建出一个尺寸为100×100画布单位的红色矩形：

```
GnomeCanvasItem *item;

Item = gnome_canvas_item_new(gnome_canvas_root(canvas),
                             GNOME_TYPE_CANVAS_RECT,
                              "x1", 0.0,
                              "y1", 0.0,
                              "x2", 100.0,
                              "y2", 100.0,
                             "fill_color", "red",
                            NULL);
```

我们在这个矩形每一个属性值之前加上一个该属性的字符串。我们来看看矩形和椭圆（类型分别为GNOME_TYPE_CANVAS_ERCT和GNOME_TYPE_CANVAS_ELLIPSE）两者共有的一些参数如表17-6。

表 17-6

参 数	类 型	说 明
x1	double	矩形或椭圆最左端的坐标值
y1	double	矩形或椭圆最顶端的坐标值

（续）

参 数	类 型	说 明
x2	double	矩形或椭圆最右端的坐标值
y2	double	矩形或椭圆最底端的坐标值
fill_color	string	填充色的X颜色值
outline_color	string	边框线的X颜色值
width_pixels	uint	以屏幕像素为单位的边框宽度（当画布的缩放比例变化时边框线不会按比例缩放）
width_units	double	以画布坐标单位计算的边框宽度（当画布的缩放比例变化时边框线也会按比例缩放）

对矩形和椭圆可以使用同样的结构，这是因为椭圆是被放在一个由给定坐标确定的不可见矩形框里的。

17.1.4 GNOME应用程序

我们已经简要地介绍了GNOME应用程序主要的建筑材料，下面将通过组合这些素材编写一个小应用程序来强化我们学到的知识。

GNOME里最灵活和最吸引人的素材就要数画布了，因此我们将编写一个图形化的时钟来展示它的威力。

1. 软件的设计要求

我们将设计一个传统的模拟时钟，所以我们需要读取计算机的本地时间并把它转换为时钟表针的坐标。表针是一些画布项目，我们将每隔一秒就用新坐标值刷新表针的位置。我们还要画出一个圆形的表盘，它的周围要有一些用来表示小时和分钟的圆点。

当然还要适当地加上几个菜单和工具条。还要加上一个个人偏好对话框，我们将通过这个对话框隐藏秒针和改变这块画布的缩放比例。

2. 我们需要编写哪些代码

我们需要编写几个能够完成下面这些工作的函数：

- 创建用户操作界面，包括一个窗口、几个菜单和一个工具条。
- 一个重画表针的例程。
- 个人偏好对话框的创建和处理。

我们开始写程序了。

动手试试：一个GNOME时钟

1）和往常一样，我们通过“# include”语句包括上必要的头文件。我们需要使用一些来自time库和math库的函数来确定时钟表针的坐标。如下所示：

```
#include <gnome.h>
#include <time.h>
#include <math.h>
```

2）现在用“# define”语句为我们的时钟定义几个常数。如下所示：

```
#define CANVAS_SIZE 100.0
#define MIDDLE (CANVAS_SIZE/2.0)
#define SECOND_HAND_LENGTH 40.0
#define MINUTE_HAND_LENGTH 45.0
#define HOUR_HAND_LENGTH 20.0
#define DOT_RADIUS 45.0    /* distance from center of the clock to dots */
```

3) 接下来初始化我们的全局变量。如下所示：

```
GtkWidget *canvas = NULL;
GnomeCanvasItem *second_hand = NULL;
GnomeCanvasItem *hand = NULL;
gboolean secondhand_is_visible = TRUE;
GnomeCanvasPoints *points, *second_hand_points;  /* the arrays that hold the hand
coordinates */
GtkWidget *clock_app;
```

4) 下面是函数们的预定义：

```
static void create_dots (int dots, GtkWidget *canvas);
static void change_canvas_scale (GtkAdjustment *adj, gfloat *value);
static void show_preference_dlg (void);
static void show_about_dlg (void);
static void apply_preferences (GnomePropertyBox *property_box, gint page_num,
                               GtkWidget *sh_checkbox);
static gint redraw (gpointer data);
```

5) 我们在这里定义三个顶层菜单：file、settings和help，它们分别包含着“exit”、“preferences”和“about”选项。我们把“preferences”和“about”菜单项与程序中的函数链接起来以便弹出与之对应的对话框。如下所示：

```
GnomeUIInfo file_menu[] = {
  GNOMEUIINFO_MENU_EXIT_ITEM(gtk_main_quit, NULL),
  GNOMEUIINFO_END
};

GnomeUIInfo help_menu[] = {
  GNOMEUIINFO_MENU_ABOUT_ITEM(show_about_dlg, NULL),
  GNOMEUIINFO_END
};

GnomeUIInfo settings_menu[] = {
  GNOMEUIINFO_MENU_PREFERENCES_ITEM(show_preference_dlg, NULL),
  GNOMEUIINFO_END
};

GnomeUIInfo menubar[] = {
  GNOMEUIINFO_MENU_FILE_TREE(file_menu),
  GNOMEUIINFO_MENU_SETTINGS_TREE(settings_menu),
  GNOMEUIINFO_MENU_HELP_TREE(help_menu),
  GNOMEUIINFO_END
};
```

6) 接下来定义工具条，它只容纳着一个项目：“Exit”按钮。如下所示：

```
GnomeUIInfo toolbar[] = {
  GNOMEUIINFO_ITEM_STOCK("Exit", "Exit the application",
                         gtk_main_quit,
                         GNOME_STOCK_PIXMAP_EXIT),
GNOMEUIINFO_END
};
```

7) 现在定义的函数将计算并画出表盘四围的圆点。我们将调用它60次，每次代表一分钟。

这个函数将计算出表盘四围所有圆点的位置并为它们创建一个CanvasItem。如下所示：

```
static void create_dot( int dots, GtkWidget* canvas)
{
 double angle = dots * M_PI / (360.0/12);
 double x1, y1, x2, y2;
 double size;
 GnomeCanvasItem *dot;
```

8) 我们需要根据圆点出现在表盘上的位置选择它们的大小。我们让出现在12、3、6、9点钟位置的圆点最大，然后每隔五分钟稍大，其他的圆点最小。如下所示：

```
 if ((dots % 15) == 0) {
     size = 2.0;
 }
  else if ((dots % 5) == 0) {
    size = 1.0;
 }
  else size = 0.5;

 x1 = MIDDLE - size + (DOT_RADIUS * sin (angle));
 y1 = MIDDLE - size + (DOT_RADIUS * cos (angle));
 x2 = MIDDLE + size + (DOT_RADIUS * sin (angle));
 y2 = MIDDLE + size + (DOT_RADIUS * cos (angle));

 dot = gnome_canvas_item_new(gnome_canvas_root (GNOME_CANVAS(canvas)),
                                          GNOME_TYPE_CANVAS_ELLIPSE,
                                          "x1", x1,
                                          "y1", y1,
                                          "x2", x2,
                                          "y2", y2,
                                          "fill_color", "red",
                                        NULL);

}
```

9) 接下来是这个应用程序的核心部分，表针的重画函数，或者更准确地说是表针的坐标修改函数，表针的坐标是根据计算机的本地当前时间变化的。对时针和分针来说，我们实际使用的是一个用三个点决定的直线对象，所以我们每秒钟只需重画两个画布项目，即秒针和时/分针。如下所示：

```
static gint redraw (gpointer data)
{
  struct tm *tm_ptr;
  time_t the_time;
  float second_angle;
  float minute_angle;
  float hour_angle;

  time(&the_time);
  tm_ptr = localtime (&the_time); /* See chapter 4 for an explanation of the time
function */
  second_angle = tm_ptr->tm_sec * M_PI / 30.0; /* The angle the second hand makes wit
the vertical */
```

10) 设置秒针坐标数组，我们用这四个坐标值画出从时钟中心点到表盘上圆点的直线来。如下所示：

```
second_hand_points->coords[0] = MIDDLE;
```

```
second_hand_points->coords[1] = MIDDLE;
second_hand_points->coords[2] = MIDDLE + (SECOND_HAND_LENGTH*sin(second_angle));
second_hand_points->coords[3] = MIDDLE -  (SECOND_HAND_LENGTH * cos (second_angle));
```

11) 接下来计算时针和分针之间的夹角，并把计算结果填写到坐标数组里去。如下所示：

```
minute_angle = tm_ptr->tm_min * M_PI / 30.0;
hour_angle = (tm_ptr->tm_hour % 12) * M_PI / 6.0 +  (M_PI * tm_ptr->tm_min / 360.0);


points->coords[0] = MIDDLE + (HOUR_HAND_LENGTH * sin (hour_angle));
points->coords[1] = MIDDLE - (HOUR_HAND_LENGTH * cos (hour_angle));
points->coords[2] = MIDDLE;
points->coords[3] = MIDDLE;
points->coords[4] = MIDDLE + (MINUTE_HAND_LENGTH * sin (minute_angle));
points->coords[5] = MIDDLE - (MINUTE_HAND_LENGTH * cos (minute_angle));
```

12) 检查表针是否已经被创建出来了。如果没有，就创建它们；否则就把新坐标设置给它们。如下所示：

```
if (hand == NULL) {

   hand = (gnome_canvas_item_new(gnome_canvas_root (GNOME_CANVAS (canvas)),
                                 GNOME_TYPE_CANVAS_LINE,
                                 "points",points,
                                 "fill_color", "blue",
                                 "width_units", 3.5,
                                 "cap_style", GDK_CAP_ROUND,
                                 "join_style", GDK_JOIN_ROUND,
                                 NULL));

  second_hand = (gnome_canvas_item_new(gnome_canvas_root (GNOME_CANVAS (canvas)),
                                 GNOME_TYPE_CANVAS_LINE,
                                 "points",second_hand_points,
                                 "fill_color", "white",
                                 "width_pixels", 2,
                                 NULL));
  }
 else
  gnome_canvas_item_set (hand, "points", points, NULL);
  gnome_canvas_item_set (second_hand, "points", second_hand_points, NULL);
 }
```

13) 下面这个函数的作用是创建一个about对话框。当我们点击“Help”菜单里的“about”菜单项时，这个对话框就会出现在屏幕上。如下所示：

```
static void show_about_dlg(void)
{
 GtkWidget *about;
  const gchar *authors[] = { "Andrew Froggatt", NULL };

  about = gnome_about_new("Gnome Clock", "0.1",
                          "Released under the GNU Public License",
                          authors,
                          ("A simple graphical clock for GNOME"),
                          NULL);
  gtk_widget_show (about);
}
```

14) 我们在这里创建个人偏好对话框，里面放上一个选择框和一个滑块素材。GnomeProperty-Box素材是从GnomeDialog素材推导出来的，它的增加功能是一个嵌入式笔记本素材。笔记本素材提供了彼此分开的“页”，每页有一个标签，我们可以通过这些分开的页把素材分成不同的组。

GnomePropertyBox还定义了两个新的信号：apply和help。只要用户点击了ok或apply按钮，它就会随时发送出apply信号；而当用户点击help按钮时，将发送出help信号，而我们要用这些信号来提供一个非模型化的对话框。如下所示：

```
static void show_preference_dlg(void)
{
  GtkWidget *preferencebox;
  GtkWidget *vbox, *hbox;
  GtkWidget *sh_checkbox;
  GtkObject *adj;
  GtkWidget *hscale;
  GtkWidget *label;

  preferencebox = gnome_property_box_new();

  vbox = gtk_vbox_new(FALSE, 2);
  hbox = gtk_hbox_new(FALSE, 0);

  sh_checkbox = gtk_check_button_new_with_label( "Second hand visible");
  gtk_toggle_button_set_state(GTK_TOGGLE_BUTTON (sh_checkbox),
                              secondhand_is_visible);
  gtk_box_pack_start (GTK_BOX (vbox), sh_checkbox, FALSE, FALSE, 0);
```

15) 我们为那个单选框设置一个信号处理器。当用户选择了这个单选框的“开”状态时，ok和apply按钮将变为“敏感的”（从显示效果来看是阴影去掉了，从使用效果看是允许点击了）。如下所示：

```
gtk_signal_connect_object (GTK_OBJECT(sh_checkbox), "toggled",
                           GTK_SIGNAL_FUNC(gnome_property_box_changed),
                           GTK_OBJECT(preferencebox));


label = gtk_label_new("Clock zoom factor: ");
gtk_misc_set_alignment (GTK_MISC (label), 1.0, 1.0);
gtk_box_pack_start (GTK_BOX (hbox), label, FALSE, FALSE, 0);
```

16) 接下来我们创建一个用来缩放我们画布的滑块素材。当用户移动这个滑块素材时将产生value_changed信号，我们把这个信号与两个信号处理器链接在一起。如下所示：

```
  adj = gtk_adjustment_new (2.0, 0.1, 7.0, 1.0, 1.0, 1.0);  /* (default, min, max,
step, page, page size) */
  hscale = gtk_hscale_new (GTK_ADJUSTMENT (adj));

  gtk_signal_connect_object (GTK_OBJECT (adj), "value_changed",
                             GTK_SIGNAL_FUNC(gnome_property_box_changed),
                             GTK_OBJECT(preferencebox));

  gtk_signal_connect_object (GTK_OBJECT (adj), "value_changed",
                             GTK_SIGNAL_FUNC (change_canvas_scale),
                             &(GTK_ADJUSTMENT (adj)->value));

  gtk_signal_connect (GTK_OBJECT (preferencebox), "apply",
                      GTK_SIGNAL_FUNC(apply_preferences), sh_checkbox);

  gtk_box_pack_start (GTK_BOX (hbox), hscale, TRUE, TRUE, 0);
  gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, FALSE, 0);

  gnome_dialog_set_parent(GNOME_DIALOG(preferencebox), GTK_WINDOW(clock_app));

  gnome_property_box_append_page(GNOME_PROPERTY_BOX(preferencebox),
                                 vbox,
                                 gtk_label_new("General"));
```

```
  gtk_widget_show_all(preferencebox);
}
```

17) 下面的函数负责处理个人偏好对话框里ok或apply按钮被点击的事件。page_num是当前的笔记本页编号，它对笔记本各页上的apply按钮提供了未来性支持。它的原理是：每个笔记本页会发送一个带该页编号的apply信号和一个page_num值为“-1”的apply信号。我们在这里将不理会page_num值不是“-1”的apply信号。而当我们接收到“-1”页的时候，将根据个人偏好对话框里选择框的状态来显示或隐藏时钟的秒针。我们需要刷新全局变量secondhand_is_visible，这样当我们关闭属性对话框后再重新打开它时，选择框就能显示为正确的状态。当ok或close按钮被点击的时候，GnomePropertyBox素材将负责替我们关闭属性窗口。如下所示：

```
static void apply_preferences (GnomePropertyBox *property_box, gint page_num,
GtkWidget *sh_checkbox)
{
  if (page_num != -1)
       return;
  if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON (sh_checkbox) ) == FALSE) {
   gnome_canvas_item_hide(second_hand);
   secondhand_is_visible = FALSE;
  }
   else {
   gnome_canvas_item_show(second_hand);
   secondhand_is_visible = TRUE;
  }
```

18) 当滑块素材被移动的时候，我们将更新画布的缩放比例并把窗口调整为适合时钟的大小。如下所示：

```
static void change_canvas_scale (GtkAdjustment *adj, gfloat *value)
{
   gnome_canvas_set_pixels_per_unit (GNOME_CANVAS (canvas),
                                     (double) *value);
   gtk_widget_set_usize(GTK_WIDGET (clock_app),
                                110 * (double) *value,
                                125 * (double) *value);
}
```

19) 最后，我们在main函数里与往常一样进行初始化和素材的创建工作。如下所示：

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

  GnomeCanvasItem *clock_outline;
  gint dots;

  gnome_init ("clock", "0.1", argc, argv);
  clock_app = gnome_app_new ("clock", "Gnome Clock");
gtk_widget_set_usize (clock_app, CANVAS_SIZE, CANVAS_SIZE);
gtk_widget_push_visual(gdk_imlib_get_visual());
gtk_widget_push_colormap(gdk_imlib_get_colormap());
canvas = gnome_canvas_new();
gtk_widget_pop_visual();
gtk_widget_pop_colormap();

gnome_canvas_set_pixels_per_unit (GNOME_CANVAS (canvas), 2);
second_hand_points = gnome_canvas_points_new(2);
points = gnome_canvas_points_new(3);
```

```
gtk_signal_connect (GTK_OBJECT (clock_app), "delete_event",
                    GTK_SIGNAL_FUNC (gtk_main_quit),
                    NULL);

gtk_widget_set_usize(clock_app, 220, 300);


gnome_app_set_contents (GNOME_APP (clock_app), canvas);
gnome_app_create_menus (GNOME_APP (clock_app), menubar);
gnome_app_create_toolbar (GNOME_APP (clock_app), toolbar);

clock_outline = gnome_canvas_item_new (gnome_canvas_root(GNOME_CANVAS (canvas)),
                                       GNOME_TYPE_CANVAS_ELLIPSE,
                                       "x1", 0.0,
                                       "y1", 0.0,
                                       "x2", CANVAS_SIZE,
                                       "y2", CANVAS_SIZE,
                                       "outline_color", "yellow",
                                       "width_units", 4.0,
                                       NULL);
```

20) 我们在这里创建出60个圆点来标记我们时钟上的分钟位置。如下所示：

```
for (dots = 0; dots < 60; dots++) {
  create_dot ( dots, canvas);
}
```

21) 下面这个函数很关键，它每隔1000毫秒调用一次redraw函数来刷新我们的时钟。

```
  gtk_timeout_add(1000, redraw, canvas);

  redraw(canvas); /* make sure our hands are created before we display the clock */
  gtk_widget_show_all(clock_app);
  gtk_main();
 return 0;
}
```

以上是这个时钟程序的完整代码。当我们编译并运行这个程序的时候，将会在屏幕上看到我们的时钟滴滴答答地走着。图17-11是这几个窗口的显示画面。

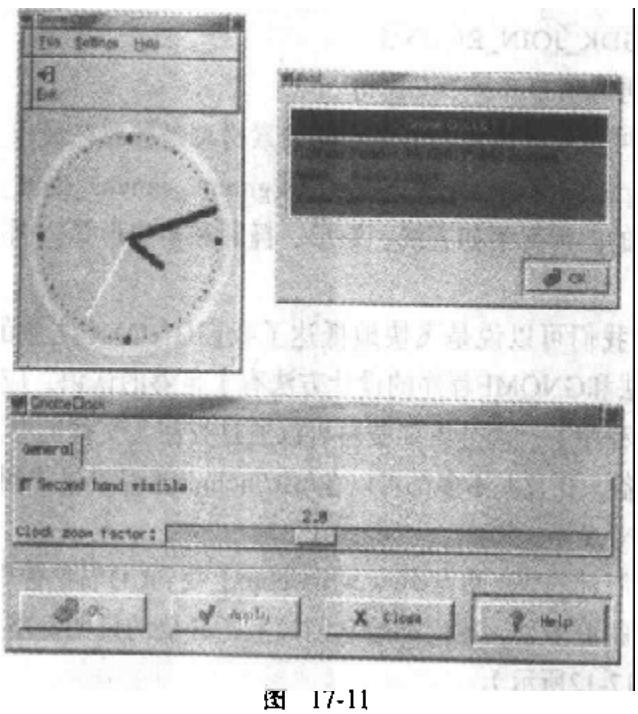


图 17-11

操作注释：

这个程序好像挺长，可它从本质上讲还只是一个相当简单的程序。这个时钟程序的关键是gtk_timeout_add函数，它会每过一秒调用redraw函数一次。在redraw函数里，我们计算出表针都应该指向什么地方，设置好坐标数组，然后调用gnome_canvas_item_set刷新画布上的项目。

对属性对话框函数我们得多说几句。我们创建了一个用来切换秒针可见状态的选择框素材和一个控制画布缩放比例的滑块素材。我们在处理复选框和滑块素材的状态改变情况时采取的做法是不同的。我们会等到ok或者apply按钮被按下之后才会去根据选择框的状态改变秒针的可见性；但画布的缩放比例会通过滑块素材产生的value_changed信号随着滑块素材的移动立刻反应出来。这就产生了随滑块素材的调整即时缩放时钟的效果。

也许你愿意创建一个抗失真的画布来给出一个高画面质量的时钟，但千万要谨慎从事！它可能会占用大量的资源。你只须把画布的创建语句修改为下面这样就可以达到目的：

```
gtk_widget_push_visual(gdk_rgb_get_visual());
gtk_widget_push_colormap(gdk_rgb_get_cmap());
canvas = gnome_canvas_new_aa();   /* create an anti-aliased canvas */
```

画布直线是通过指定一个直线上的点的数组画出来的，你必须用下面这条语句创建和初始化：

```
GnomeCanvasPoints* points = gnome_canvas_points_new (gint num_of_points)
```

其中的num_of_points是你想在画布上串起来的点的个数，所以这个数组的长度将是这个数字的两倍，x坐标将被放在偶数编号的元素里，y坐标将被放在奇数元素里。

我们没有使用分开的时针和分针，我们用一条直线把时针位置和中心点连起来，然后再连到分钟位置，所以要给gnome_canvas_points传递一个“3”。这样做与时针分针分开创建相比并没有什么特别的好处，我们只是通过它来演示两条直线的join-style相交属性的用法而已，这个属性在例子里被设置为GDK_JOIN_ROUND。

在我们结束之前，再对画布素材多说几句。

画布上的项目和组可以重叠摆放。最后创建的素材总是被放在最顶上；如果你想改变这个重叠次序，就需要调用gnome_canvas_item_lower或gnome_canvas_item_raise函数。画布支持文本和图像，还支持简单的图形元素如直线、矩形、椭圆和多边形等。画布上的项目可以连接信号处理器。

下面该是什么了？我们可以说是飞快地抵达了我们GNOME之旅的终点，但大家应该对GNOME素材的使用方法和GNOME程序的设计方法有了足够的认识，应该能参考着Web网上的资料来编写自己的应用程序了。部分重要资料可以通过查阅头文件的办法获得；这类头文件的注释文档一般都比较齐备，并且差不多都可以在/usr/include子目录里找到。

3. GNOME中的CD唱盘管理软件

读者可以在Wrox出版社的Web站点www.wrox.com上找到CD唱盘管理软件的GNOME前端的源代码，它用到的素材就更多了，我们都给它们加上了必要的说明。本章的程序示例全都可以在该站点上找到（如图17-12所示）。

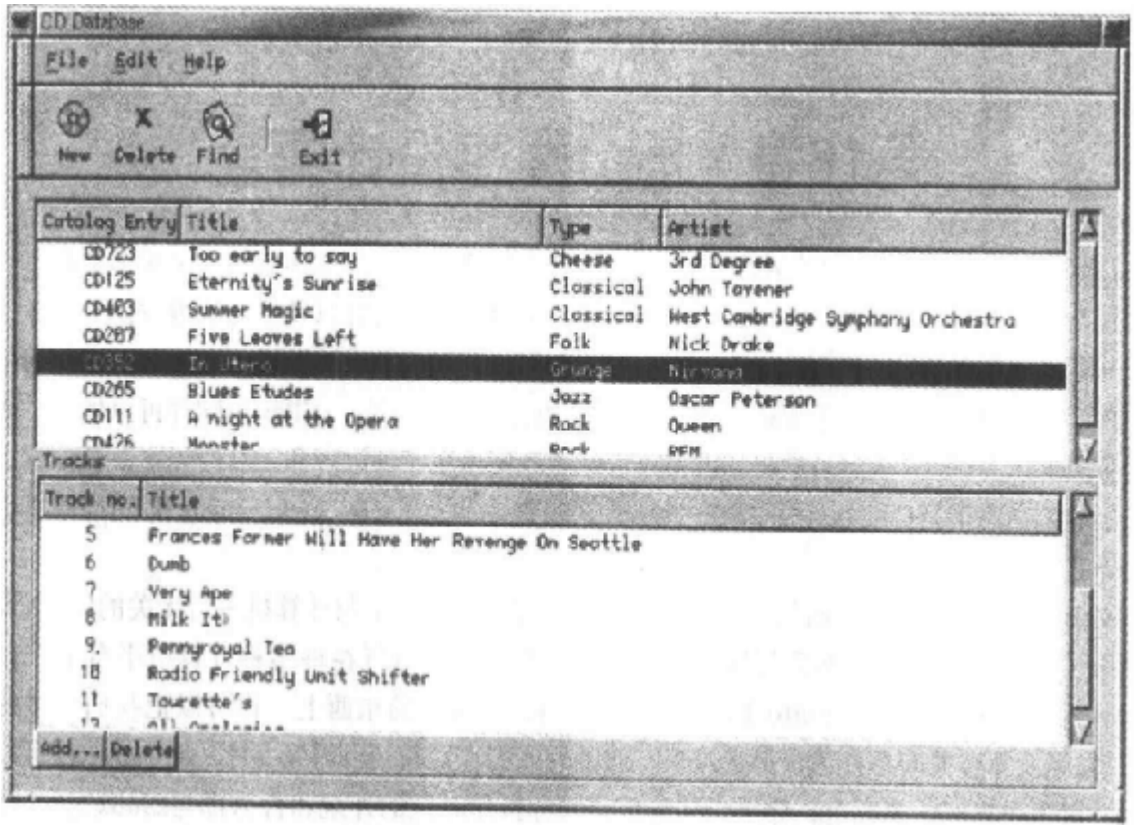


图 17-12

很明显没有足够的篇幅来讨论所有的事，而且毫无疑问我们只是刚接触到GNOME的皮毛。从某种意义上讲，一本书还是只介绍基本概念最好，因为任何一本详细介绍GNOME最新开发成果的书面材料很快就都过时了。

GNOME是一个繁忙而又人才济济的大家庭。现时期，世界范围大约有300名程序员定期对这个项目进行着研究和贡献。也许这一章的学习激发了你足够的创造力，使你成为其中的一员也说不定。祝你好运，GNOME愉快！

17.2 本章总结

我们在这一章里学习了GNOME的基本概念和利用GNOME提供的材料来编写应用程序的基本方法。

我们向大家介绍了基础性的开发工具包GTK+，学习了它几个素材的使用方法。另外，我们还学习了GNOME画布素材的使用方法。

第18章　Perl程序设计语言

Larry Wall的Perl程序设计语言身处shell和C语言之间，并从许多标准的UNIX工具那里吸取了大量营养，这使它非常适合于完成文本处理、CGI脚本程序设计以及系统管理方面的任务。同时，我们还将看到Perl的扩展性也非常好（你甚至可以通过PerlTk扩展用它来编写图形化的用户操作界面）。我们完全可以这样说：只要是能用C语言做到的事情，用Perl也同样可以做到，而且可能会更简单。Perl非常容易学习，因为它借鉴了各种程序设计语言和工具性程序，其中肯定有你已经非常熟悉的。C和shell程序员会特别感到如鱼得水，而sed、awk、Basic和Tcl程序员也不会觉得它陌生。

Perl最优秀的特色之一就是它在操作系统的顶部提供了一个与计算机平台无关的UNIX风格的抽象层面。这句话是什么意思呢？是这样的：Perl可以运行在许多种计算机平台上，比如Windows、苹果公司的Macintosh以及任何看起来像UNIX的东西上。你可以把我们学过的与Linux程序设计有关的各种想法都带过来，而不管我们正在使用的系统是哪一种，Perl都可以尽可能好地实现它们。事实上，你可以假定任何目标操作系统的外观和行为都与Linux完全一致，而这种对待事物的方法很让人高兴——我想你肯定会同意这一观点的。

在这一章里，我们将学习如何编写基本的Perl脚本程序，并且会把我们前面学过的知识运用到Perl里来。我们不打算在这一章里涉及到Perl语言的所有方面，只是想把最有用和最常用的内容介绍给大家；另外，我们还将用Perl再次实现大家都很熟悉了的CD唱盘管理软件。

18.1　Perl语言简介

我们从学习Perl语言的基本概念入手，它们是变量、操作符和函数、规则表达式以及文件的输入输出。后面内容里的信息量是很大的，而我们要到本章结束时才会把它们串在一起。但马上就要出现的那些概念对读者来说都应该是比较熟悉的，所以阅读量应该不会太大。

首先，我个人认为，我们应该检查大家的系统上是不是已经安装了Perl。大多数Linux现在都自带有Perl，所以大家只需敲入“perl -v”看看屏幕上会不会出现以下的内容：

```
This is perl, version 5.005_03 built for i386-linux

Copyright 1987-1999, Larry Wall
Perl may be copied only under the terms of either the Artistic License or the GNU
General Public License, which may be found in the Perl 5.0 source kit.

Complete documentation for Perl, including FAQ lists, should be found on this system
using 'man perl' or 'perldoc perl'.  If you have access to the Internet, point your
browser at http://www.perl.com/, the Perl Home Page.
```

如果没有上述内容，请检查你的路径设置和Linux的软件包管理器（如果你有的话），看看自己到底把Perl安装到哪儿去了。如果根本就没有安装，你可以从这几个来源获得一份最新的Perl版本：Linux发行版本的CD盘、Linux发行商FTP站点的contrib或main目录、或者从CPAN那

里弄一份源代码来（CPAN是“Comprehensive Perl Archive Network”的字头缩写，意思是“智能化Perl档案网”，我们稍后有专题介绍它）。在Perl的主页http：//www.perl.com/上有大量关于Perl的文档、新闻、通用性资料。

18.1.1 “Hello” Perl程序

首先，我们来看看简单的Perl程序是什么样子的，怎样才能运行它。这是我们的脚本程序，我们给它起名为hello.pl。如下所示：

```
print "Hello World\n ";
```

现在，我们在命令行上通过Perl解释器来运行它：

```
$ perl hello.pl
Hello, World
$
```

一切按计划进行，Perl打了个招呼并把我们返回到shell提示符处。我们可以用“#！”记号把Perl解释器的安装位置告诉Linux，这与当初编写shell脚本程序时的做法是完全一样的。我们把hello.pl修改为下面的样子：

```
#!/usr/bin/perl -w
# hello.pl, version 2
print "Hello, World\n";
```

（读者可能需要把“#！/usr/bin/perl”修改为自己Perl二进制可执行文件的实际存放位置。你可以用“which perl”或你shell里的等价命令查找它。）

接下来，我们需要给这个文件设置上可执行权限才能直接运行它。如下所示：

```
$ chmod 755 hello.pl
```

现在，我们就可以像对待shell脚本程序那样运行这个文件了，如下所示：

```
$ ./hello.pl
```

从这个例子里我们可以看出：

1) Perl语句和C语言语句一样都是以分号结尾的。字符串里的换行符也像C语言里那样用“\n”来表示。所有其他的“巨字符”（即控制字符序列）都完全借鉴自C语言。

2) 类似于shell和Tcl语言中的用法，注释行是以“#”字符开始的（注意“#!”的特殊含义，这也与shell一样）。

3) 第一行里的“-w”是perl解释器的一个选项，它的作用是打开全部警告功能，我们强烈推荐这种做法。虽然Perl对代码的书写格式并没有太严格的要求，但如果我们的脚本程序能够通过“-w”测试的话，它们将运行得更可靠。

18.1.2 Perl语言中的变量

Perl语言有三种变量类型，它们是：标量（scalar）、数组（array）和哈希表（hash）。

1. 标量

标量是普通的字符串和数字；它们的表达方式和操作方式与shell变量有很多共同之处。可

以把我们的脚本程序重新写成下面这个样子：

```
$message = "Hello World\n ";
print $message;
```

C语言程序员可能更愿意写成“print ($message);”的样子，没问题。Perl语言对print这类函数是否加有括号并不是太在意；但如果加上括号有助于提高可读性，那就不要去掉它们。另外，赋值语句的等号两头可以有空格，这与C语言类似、与shell相反。值得注意的是我们不必象C语言那样为变量分配和回收内存空间，也不需要事先声明它们；Perl会负责所有这类事情，让程序员专心编写出好程序来。

C语言程序员也许更愿意用printf来替代print；Perl也确实准备了一个printf函数，它的功能与C和shell里的同名函数是一样的。但print的执行效率要更高一些，所以除非需要用到函数printf的格式编排功能，最好是坚持使用print函数。

2. 数组

数组是列表（list）的变量形式，正如Tcl语言中一样。但数组变量不是以“$”而是以“@”字符打头的。下面是一个简单的列表：

```
(1, 2, 3, 4)
```

我们用下面这条语句把它放到一个变量里去：

```
@mylist = (1, 2, 3, 4 );
```

如果想把数据再从数组里取出来应该怎么办？我们先来看看下面这段列表操作，然后再解释发生了什么事情。

```
@message = ("\n", " ", "World", "Hello,");
print $message[3], $message[1], $message[2], $message[0];
```

好了，你肯定知道它会输出什么样的结果，但这是怎么做到的呢？引起我们注意的第一件事情是print可以有好几个用逗号隔开的参数，它们实际上构成了一个列表，但这并不是个重要问题（它稍后会变得重要起来）。

接下来，大家应该注意到虽然那个数组叫做@message，但引用其中的元素时我们使用的却是$message[element]。如果从“你想要什么”而不是“你得到了什么”的角度来看待这个问题就容易理解了：当我们一个一个地引用着数组的元素时，我们实际“想要”的是它的标量值而不是数组本身。（不要被误导地认为@message和$message指的是同一个变量——它们没有丝毫联系，光是$message自个代表不了数组中的任何元素。）

最后，数组中的第一个元素下标是0，如果读者习惯于C语言会觉得很舒服，可如果读者习惯于使用BASIC、Pascal或sed/awk编程就会有点不适应。如果愿意，你完全可以把数组中第一个元素的编号改为1，但这样做的后果往往弊大于利，所以最好是不要去管它。

也许你想取出的不是单个的元素而是一个范围，或者叫做一个数组切片。指定元素的范围区间就能达到这一目的，但这次要注意，我们“想要”的是一个列表而不是一个标量。如下所示：

```
@a = ("zero", "one", "two", "three", "four");
@b = @a[0,2..3]; # @b is ("zero", "two", three");

# Reducing it to one statement:
@b = ("zero", "one", "two", "three", "four")[0,2..3];
```

Perl会对列表自动进行平面化，这句话的意思是说列表里不能再有其他的列表，数组也不能是多维的（你可以通过引用来实现多维列表或数组，但这超出了本章的讨论范围。具体操作办法请参考perlref和perllol的使用手册页）。这就意味着下面这些语句功能是相同的：

```
@a    ("zero", "one", "two", "three", "four");

@a    ("zero", "one", ("two", "three"), "four");

@half = ("zero", "one", "two"); @half2 = ("three", "four");
@a = (@half, @half2);
```

列表允许出现在赋值语句等号的左边，而且这是Perl语言一个很有用的操作。如下所示：

```
($first, $last) = ("alpha", "omega");

@a = ("alpha", "omega") ; ($first, $last) = @a; # Same thing.
```

能够被放在赋值语句等号的左边的事物叫做“左值”，即列表可以是左值。它最值得炫耀的用法是：在交换两个变量的值时不需要使用一个临时变量。下面是C语言程序员完成这一操作的做法：

```
$temp = $last;
$last = $first;
$first = $temp;
```

而Perl语言老手会写出：

```
($first, $last ) = ($last, $first );
```

3. 哈希表

最后一个变量类型是哈希表（hash）变量，它们也叫做关联数组。它允许你通过键字来保存和检索数据，哈希表变量的打头字符是“%”。哈希表非常适用于表示数据之间的关系。下面是一个简单的用来保存电话号码的哈希表：

```
%phonebook = ( "Bob" => "247305",
               "Phil" => "205832",
               "Sara" => "226010" );
```

哈希表的创建过程和列表相同，但要使用“=>”操作符（它在许多情况里与一个逗号的作用差不多）来分隔键字-键值对。我们现在来取出其中的一项，如下所示：

```
print "Sara's phone number is ", $phonebook{"Sara"}, "\n";
print "Bob's phone number is ", $phonebook{Bob}, "\n";
```

因为这次想要的还是一个标量，所以我们使用了“$”符号。为了更清楚地表示我们想要找的东西，要使用花括号而不是方括号；另外，从第二行可以看出并不是必须给键字加上引号。但需要提醒大家注意的是：哈希表的键字就像变量名一样是分大小写的，也就是说：$phonebook{Bob}等同于$phonebook{"Bob"}，但与$phonebook{bob}不是一回事。讲到这里，对哈希表中的某个数据项进行修改的操作就不难理解了：

```
$phonebook{Bob} = "293028";
print "Bob's new number is ", $phonebook{Bob}, "\n";
```

4. 引号和替换

类似于shell脚本程序中的情况，放在双引号字符串里的变量如果没有用反斜线字符进行转义，就会被替换。我们现在把电话号码本例子重新改写为：

```
%phonebook = ( "Bob" => "247305",
               "Phil" => "205832",
               "Sara" => "226010" );
print "Sara's phone number is $phonebook{Sara}\n";
print "Bob's phone number is $phonebook{Bob}\n";
```

（很明显，我们不能在双引号字符串里放入双引号字符串。）

和大家想的一样，单引号不会引起变量替换，也不会让“\n”等特殊字符起作用；这与shell程序设计中的规定是相同的。请看下面的例子：

```
$myvar = "quoting";
print 'Take care when $myvar strings\n';
print "Take care when $myvar strings\n";
```

它将给出如下所示的输出：

```
Take care when $myvar strings\nTake care when quoting strings
```

类似于Tcl语言，Perl会自动完成数值和字符串间的类型转换。如下所示：

```
print "4 bananas"+1; # Gives "5"
print "123" + "456"; # Gives "579"
```

5. 特殊变量

Perl语言有很多的特殊变量，我们在继续学习的过程中会遇到其中的一些。这些特殊变量里最重要的有三个。第一个“$_”是许多操作和函数缺省的标量。举例来说，不带参数的print语句就会把“$_”的值打印出来。这个变量很有用，但有时候也会导致出现下面这样的代码：

```
# Strip comment lines.
while (<>) { print unless /^#/ }
```

这里使用了三次“$_”变量，一次用来从标准输入读入一行文本（“<>”读行操作符处，我们稍后再做介绍——现在只要知道它是什么就行了）、一次用来作为print函数的参数，还有一次用来测试文本行是否以井字符（#）开始（我们将在规则表达式部分对“/^#/”的含义做出解释）。对一个老练的Perl程序员来说，这类代码用起来很顺手，但新手可能就不知所云了。除非语句的意思很明显，否则不要在你自己的代码里轻易省略“$_”变量。

另外一个重要的特殊变量是@ARGV数组，它包含着用户应用程序的参数。但它不像C语言那样能够通过argv[0]获得应用程序的名字（脚本程序名在Perl里被保存在“$0”变量里）；数组元素$ARGV[0]将是程序的第一个参数。

最后，哈希表变量%ENV允许你查看和改变环境变量，就像你在shell程序设计中那样。如下所示：

```
print $ENV{PATH}; # /usr/local/bin:/usr/bin...
print $ENV{EDITOR}; # vi
$ENV{EDITOR} = "emacs"; # change to emacs for the rest of the program.
```

18.1.3 操作符和函数

在Perl语言里，操作符和函数的区别并不是很明显，有些你认为是函数的东西其实是操作符，

而有些你认为是操作符的东西其实是函数。因为这个原因，我们在这里也不对这两者做细致的区分。我们不加区别地使用“函数”和“操作符”来进行讨论。

1. 数值运算符

我们对数字都可以进行哪些操作？先猜四个：+（加）、-（减）、*（乘）、\（除），没错，它们的工作情况和你想象的一样，并且可以按常见情况与括号联合使用。它们的优先规则与C语言里也完全一样。

```
$a = (4*5)*3; # 23
$b = 1/(4+4); # 0.125
$c = 1/4+4; # 4.25
```

我们还可以使用一个余数操作符（也叫做求余操作符）“%”。但使用这个操作符时一定要保证是在对正数进行操作；如果你使用的是“$a % -$b”，结果将是“($a % $b) - $a”，而这可能就不是你想要的结果了。另外还有一个乘方操作符“**”。请看下面的例子：

```
$a = 17 % 5; # 17 into 5 goes 3 times remainder 2, so $a = 2
$a = 22 % -7; # 22 into 7 goes 3 times remainder 1, so $a = 1-7 = -6
$a = 2 ** 8; # 2 raised to the 8 is 256
```

对变量来说，可以进行前缀或后缀方式的递增递减运算（但这些前后缀操作符对字符串的意义是不同的），具体做法和效果类似于C语言。如下所示：

```
$a = 5; $b = 7;
$c = ++$a + $b; # $c is 6+7 = 13
```

然后是一些科学计算函数，比如三角学方面的sin和cos、平方根sqrt和自然对数log等函数。科学计算函数可以在perlop和perlfunc使用手册页里查到。

2. 字符串操作符

我们对字符串最经常进行的操作是合并它们，这个工作是用句点操作符（.）实现的。数字和字符串之间的自动转换会照常进行。请看下面的例子：

```
$a = "foo" . "bar" . "baz" ; # Gives us "foobarbaz"
$a = "number " . 1 ;         # Gives "number 1"
$a = "1" . "2" ;             # Gives "12"
```

有时候需要某个字符串重复出现。这是用操作符“x”实现的，请看：

```
$a = "ba".("na"x4) ; # "bananananana"
$a = 1 x 3 ; # 111
```

第二个例子里让我们能够看清楚“x”和“*”操作符之间的区别。

另外一个常见的字符串操作是删除最后一个字符。chop函数可以用来删除最后一个字符，不管它是什么；chomp更精细一些，它的作用是从字符串的尾部删除输入记录之间的分隔符（通常是一个换行符）。当你需要从文本文件里读取输入但又不想去猜测最后一个字符是否是一个换行符时，这些函数将大派用场。chop和chomp这两个函数都会返回一个新字符串并且改变你分配给它们的变量。请看：

```
$a="bite me\n";
chomp($a); # $a is now "bite me"
chomp($a); # $a is still "bite me" since there's no newline
chop($a)  # $a is now "bite m"
```

如果我们的字符串完全是由字母组成的，对它们进行递增操作将返回可能正是你预期的结果——最后一个字符的ASCII值将被增加；如果它原来是“z”或“Z”，就会折返为“a”。如果字符串是以一个数字打头的，Perl将把它转换为一个数值，数字后面的内容都将丢失。如下所示：

```
$a = "abc"; print ++$a; # Returns "abd"
$a = "azz"; print ++$a; # Returns "baa"
$a = "0 Goodbye Cruel World"; print ++$a; # Returns 1
```

除了对字符串进行合并以外，有时也需要把它们拆分开，而这是由split函数完成的。你可以对一个字符串或一个规则表达式进行split操作；如果没有给出参数，它就会把“$_”在空白字符处拆分开。split函数可以有两个可选的参数，第一个是一个按照它来进行拆分的模板，第二个是将要被拆分的字符串。注意：如果你给出了一个准备被拆分的字符串，就必须给出一个模板做为第一个参数；一个经常出现的错误就是split函数里没有模板却只有文本。请看下面的例子：

```
# Split $_ on whitespace
$_ = "one two three";
@a = split; # ("one", "two", "three")

$passwd="simon:x:500:500:Simon Cozens,,,:/home/simon:/bin/zsh";

@passwd= split ":", $passwd; # Split on a string.
# Can also write it like this:
@passwd= split /:/ $passwd; # Split on a regular expression.(Same thing)
($uid, $gid) = @passwd[2,3];

# More idiomatically:

($uid, $gid) = (split ":", $passwd)[2,3];
```

逆操作（把一个列表转换为一个字符串）可以用join来完成。你可以把列表元素和分隔符一起合并为一个字符串。如下所示：

```
@mylist = ("one", "two", "three", "four");
$mystring = join "?", @mylist; # one?two?three?four
```

熟悉BASIC语言的人都知道substr是干什么用的；它返回的是字符串的一个子字符串。你必须给出两个参数，一个是字符串本身，另一个是偏移值。这两个参数的作用是把从偏移值开始一直到字符串结尾的所有东西都取下来（如果给出的偏移值是一个负数，就表示将“从字符串的尾部开始计数”，-1表示倒数第一个字符，-2表示倒数第二个字符，依次类推）。如果还给出了一个长度参数，就表示最多取下那么多个的字符来。请看下面的例子：

```
$string="the glistening trophies";
print substr($string, 15); # trophies
print substr($string, -3); # ies
print substr($string, -18, 9); # listening
print substr($string, 4, 4); # glis
```

如果再加上一个参数，你就可以substr对字符串进行替换性修改，如下所示：

```
substr($string, 7, 4, "tter"); # Returns "sten"
print $string;                 # the glittering trophies
```

（注意这将改变原来的字符串，substr这种用法的返回值是字符串指定位置上原来的内容。）

这个替换性操作还可以用一个更正规的赋值语句来完成，如下所示：

```
substr($string, 7, 4) ="tter"; # Functions as lvalues.
```

我们对剩下的几个函数做一个快速的介绍，因为它们都比较简单。length就像它的字面含义那样返回的是字符串的长度（但你不能通过修改字符串长度的办法来截短字符串）。reverse返回一个首尾倒置的字符串。但reverse更经常被用来对列表进行操作：它把它的参数看做是一个列表，然后按首尾倒置的顺序返回这个列表。为了得到想要的结果，你必须给它加上scalar关键字，强制它对标量类型的数据进行操作。如下所示：

```
$a="Just Another Perl Hacker";
print length $a; # 24
print reverse $a; # list context   "Just Another Perl Hacker"
print scalar reverse $a; # "rekcaH lreP rehtonA tsuJ"
```

最后，uc和lc分别用来把字符串中的全部字母转换为大写或小写字符。而ucfirst和lcfirst用来把字符串的第一个字母转换为大写或小写字符。如下所示：

```
$zippy="YOW!! I am having FUN!!";
print uc($zippy); # YOW!! I AM HAVING FUN!!
print lc($zippy); # yow!! i am having fun!!
print ucfirst(lc($zippy)); # Yow!! i am having fun!!
print lcfirst(uc($zippy)); # yOW!! I AM HAVING FUN!!
```

3. 逻辑操作符与二进制按位操作符

Perl中的二进制按位操作符“&”（与）、“|”（或）、“^”（异或）、~（非）和移位操作符“>>”与“<<”都是对整数进行操作的，与C语言中的用法完全一样。你甚至可以通过“0x”和“0”前缀来表示十六进制数或八进制数。如下所示：

```
0xF0 | 0x0F = 255 (0xFF)
0xAA ^ 0x10 = 186 (0xBA)
```

逻辑运算符“&&”、“||”和“!”与C语言中的用法也完全一样，但在Perl里你还可以使用英文“and”、“or”和“not”。因为Perl语言对逻辑表达式采用的是短路计算方法（如果后续的逻辑计算影响不了整个表达式的结果，就不再对它们进行计算了），所以这几个操作符经常被用在程序的控制流程里。如下所示：

```
risky_function()
  and print "Worked fine\n"
or print "Function didn't succeed\n";

# Also written as:
risky_function() && print "Worked fine\n"
|| print "Function didn't succeed\n";
```

你也可以按同样的办法使用“if”和“unless”关键字：

```
print "Worked fine\n" if risky_function();
$a="Default value" unless $a;
```

你可以从标量值里获得你的真值或假值（“0”和未定义是假值，其他一切都是真值），也可以从各种比较操作里获得真值或假值。对数字进行比较时可以使用标准的“<”、“>”、“==”（只有一个等号“=”时是赋值操作，不要把这两者弄混了）和“!=”。但对字符串进行比较时要使用另外一套比较操作符：“lt”表示字符串小于、“gt”表示大于、“eq”表示等于、“ne”表示不

等于。

4. 数组操作

字符串和数字这两种标量类型上的操作基本上就介绍完了。数组和列表上的情况又是如何呢?

数组上最重要的操作之一是查出它里面有多少个元素，你可能会认为length能够奏效，但不行。我们必须把数组放到标量上下文里进行求值，就像我们刚才的倒置字符串操作那样。注意：Perl不支持“稀疏的数组”，这样的数组会被认为是填满了未定义元素。

```
@array = ("zero","one","two","three");
print scalar @array; # 4 elements in the array

$array[200] = "two hundred";
print scalar @array; # 201 elements; some of them are empty, though.
```

Perl还允许我们查看最高编号元素的下标。这个值一般比数组元素的个数少一个，因为数组元素是从零开始编号的。如下所示：

```
@array = ("zero","one","two","three");
print $#array; # 3
```

接下来，我们可以从数组里取出其中的元素。我们既可以把它想象为shell程序设计中的一个数组并通过shift从它的首端取出元素，也可以把它想象为一个堆栈，通过pop从它的尾端取出元素。如下所示：

```
@array = ("zero","one","two","three");

print shift @array; # zero - array is now ("one","two","three")
print pop @array; # three - array is now ("one","two")
print shift @array; # one - array is now ("two")
print pop @array; # two - array is now ()
```

类似地，我们能够再把东西放进去，这可以通过unshift或push来完成。我们可以一次插入多个元素。如下所示：

```
@array = ();
push @array, "two"; # array is now ("two")
unshift @array, "one"; # array is now ("one","two")
push @array, "three", "four"; # array is now ("one","two","three","four")
unshift @array, "minus one","zero";
print join ",", @array;
# minus one,zero,one,two,three,four
```

现在让我们来看看reverse对列表的使用效果：

```
print join ",", reverse @array;
# four,three,two,one,zero,minus one
```

我们可以用sort把列表排序为ASCII顺序。如下所示：

```
@a = ("delta","alpha","charlie","bravo");
@b = sort @a; # ("alpha","bravo","charlie","delta");
```

我们还可以通过特殊的“块”格式指定自己的排序顺序；Perl把$a和$b设置为进行比较的两个值。对这种情况下的数值来说，我们必须使用特殊的比较操作符“<=>”——如果左边的数大

于右边的数，它返回“-1”；两数相等返回“0”；右数大于左数则返回“1”。如下所示：

```
@a = (5, 8, 3, 0, 1);
@b = sort {$a <=> $b} @a; #(0, 1, 3, 5, 8)
```

5. 哈希表操作

最后，我们介绍哈希表上的各种操作。最神奇的是reverse操作，它把哈希表的查找方向倒了过来，如下所示：

```
%phonebook = ( "Bob" => "247305",
               "Phil" => "205832",
               "Sara" => "226010" );
%index = reverse %phonebook;
print $index{"226010"}; # Sara
```

注意不要让两个键字有相同的键值——否则其中的一个就会丢失。我们还可以用keys和values函数把键字和键值分别提取到不同的列表里。如下所示：

```
@names = keys %phonebook; # ("Bob", "Phil", "Sara")
@numbers = values %phonebook; # ("247305", "205832", "226010")
```

用each可以提取出键字-键值对；每调用一次each就会返回一个由两个元素组成的列表，这两个元素分别是一个新的键字和与之对应的键值；如此循环直到全部处理完为止。如下所示：

```
while ( ($key, $value) = each %phonebook ) {
        print "$key\'s phone number is $value\n";
}
```

不要错误地认为自己把数据项放到哈希表里的顺序就是从中提取它们的顺序。Perl在返回哈希表数据项时采用的是一种随机的顺序，事实上，甚至两遍提取操作分别得到的顺序都不能保证是一样的。

18.1.4 规则表达式

规则表达式（圈里人称之为“regexps”）是Perl是强项之一。它们可以实现功能极其强大的模板匹配和替换，它们大概称得上是Perl程序员武器库里火力最猛的武器了。熟悉sed和awk或egrep的人们对规则表达式的基本规定都应该是比较了解的；Perl的规则表达式引擎是在sed模型的基础上扩展而来的。

我们先来看看如何使用规则表达式来匹配字符串，然后再研究如何替换它们。

1. 匹配

最基本的规则表达式就是我们想在一个字符串里查找的一小段文字。传统上习惯把规则表达式放在两个斜线字符之间，即“/regexp/”的样子；把一个规则表达式运用到一个变量或标量数据时要使用语法“$scalar =~ /regexp/”。如果匹配是成功的，这个做为操作符的函数将返回真值（还有一个“!~”操作符，它类似于“=~”，但两者的结果正好相反——只有匹配失败时“!~”才返回真值）。我们来看一个例子：检查我们的字符串里是否包含着“jaws”，如下所示：

```
$sea = "water sand Jaws swimmers";
print "Shark alert!" if $sea =~ /jaws/;
```

不过，这两条语句打印不出什么东西，这是因为规则表达式在缺省情况下是区分大小写的；我们可以通过限制符“i”关闭这个功能。如下所示：

```
print "Shark alert!" if $sea =~ /jaws/i;
```

我们还可以在规则表达式里使用一些特殊字符：“^”代表“字符串的开始”——也就是说，我们准备查找的表达式必须出现在字符串的开始部分。类似地，“$”代表“字符串的结尾”——即这个表达式必须出现在字符串的结尾部分。但如果只给出了“/regexp/”，它就将对“$_”变量进行测试。我们现在就可以说清楚几页前介绍“$_”的隐含使用方法时给大家看的代码了，那条语句是这样的：

```
# Strip lines starting with a hash
while (<>) { print unless /^#/ }
```

把它写得明白些，就是：

```
# Strip lines starting with a hash.
while ($_ = <>) { print $_ unless $_ =~ /^#/ }
```

换句话说，当我们把“$_”设置为标准输入的下一行使它为真时（即包含有内容时），如果这一行不是以“#”字符开始的，就把它打印出来。

接下来向大家介绍通配符：“?”匹配某个字符的0次或1次出现——也就是说它前面的字符在匹配里是可有可无的；“*”匹配0次或多次，就像shell里的情况一样（但要注意这样的情况：“/q*/”永远都会得到匹配，即使字符串里根本没有“q”也是如此，因为它匹配了0次。）；“+”匹配至少一次。表18-1是这些通配符的几个用法示例：

表 18-1

/shoo?t/	匹配“shot”或“shoot”
/sho+t?/	匹配“shot”或“shoot”或“shooot”等等，但不匹配“sht”
/sho*t/	匹配“sht”、“shot”、“shoot”等等

下一组特殊字符用来匹配不同类型的字符，包括常见的巨字符，即“\t”匹配制表符；“\n”匹配换行符等等；“\s”匹配任何看起来像空格的字符；“\w”匹配一个“单词”（英文字母数字或“_”）字符；而“\d”匹配一个数字。这些特殊字符的大写形式等于否定形式的匹配，即“\S”是一个非空格；“\W”是一个非单词字符；“\D”是一个非数字等等。最后，“.”匹配任何字符和字符串。请看表18-2里的用法示例：

表 18-2

/push\s*chair/	匹配“push”后面有零个或多个空格或制表符，然后是“chair”
/number\s*\d+\s/	匹配“number”后面有零个或多个空格，然后是一个或多个数字，最后是一个空格
/^e.*d$/	匹配以字母“e”为第一个字符，字母“d”为最后一个字符的任意一行文本
/\s/	匹配任意一行不光有空格的文本行
/\sperl\s/	匹配两头各有一个空格的字符串“Perl”

最后一个用法示例最好写成“/\bPerl\b/”，其中的“\b”是“单词边界”巨字符，它将匹配字符串前后的非单词字符（包括空格、标点符号等）。因此，“/\b\w+\b/”将匹配一个“单词”。

如果我们想知道匹配到了什么东西该怎么办？这么办：如果我们给规则表达式里我们感兴趣的部分加上括号，Perl就会替我们把它匹配到的东西保存起来。类似于sed里的做法，Perl将把第一个括号中的表达式匹配到的东西保存在一个名为“$1”的变量里，下一个保存到“$2”里，依次类推。我们就可以从中看出匹配操作是如何进行的了：

```
$test = "he said she said";
$test =~ /\b(\w+)\b/; # $1 is set to "he"
$test =~ /(sa.*d)/; # $1 is set to "said she said"
```

先看第一个例子。Perl永远都是从最左边开始进行匹配的，这次它返回从左边开始找到的第一个匹配（这叫做“热心”）。再看第二个例子，Perl会尽量把第一次匹配操作中的找到的匹配保存得时间长一些；它返回自己找到的第一个匹配并尝试使这个匹配尽可能地大（这叫做“贪婪”）。你可以关闭“贪婪”功能，办法是在通配符后面加上一个问号“?”——比如“/(sa.*?d)/”将只匹配到“said”。如果我们在规则表达式里使用了多个括号，Perl会把所有匹配到的东西都放在一个列表里返回给我们，如下所示：

```
$test = "he said she said";
@matches = $test =~ /\s*(\w+)\s*(\w+)\s*(\w+)\s*(\w+)/;
# @matches is ("he", "said", "she", "said");
# (Now you can see that split() actually splits on /\b/)
```

（当然，如果操作的规模比较大，这种办法会很罗嗦；我们稍后会向大家介绍另一种做法。）我们还可以用括号给出选项：“/(boy|girl)/”将匹配字符串里的“boy”或“girl”。最后，我们还可以在规则表达式里定义字符集合——它们要用方括号进行定义。比如说，“[a-z]”将匹配全体小写字母，而“[aeiou]”将匹配小写的元音字母。巨字符也可以放在字符集合里；如果在字符集合的最前面放上一个“^”，就表示对字符集合的补集进行匹配；比如说，“[^a-zA-Z]”将匹配任何不是字母的字符。

如果你想匹配这些特殊字符本身，包括“?”、“.”、“(”、“/”等，就必须用一个反斜线进行转义。“/\.\s*\((.*?)\)/”的执行情况是：先匹配一个句号，然后是零个或多个空格，然后一个左括号，再把文本复制到一个变量里（我们把这个变量称为“后引用”，原因稍后再解释）直到遇见右括号为止。

2. 替换

对字符串进行了匹配操作之后，我们可能想把匹配替换为其他的内容。这是通过语法“s/regexp/replacement/”实现的。如下所示：

```
$test = "he said she said";
$test =~ s/said/did/; # Gives "he did she said"
```

我们可以看到，它找到了第一个匹配，替换它，然后就结束了。要想完成全部的查找和替换，需要使用另外一个限制符“g”（sed里也这样用，甚至vi和ed也是如此）。

```
$test = "he said she said";
$test =~ s/said/did/g; # Gives "he did she did"
```

在匹配操作中也可以使用“g”限制符，如下所示：

```
$test= "123 456 7 890";
@array = $test =~ /\b(\d+)\b/g; # (123, 456, 7, 890)
```

当然，在替换操作的文本替换部分里，匹配巨字符不再有特殊的意义了。这很明显，你不可能把一个匹配替换为“任意数字”或“0或多个字母t”。替换文本就像是双引号中的字符串，会进行变量替换等操作（事实上，规则表达式也会这样做。你可以把“子规则表达式”放在变量里，使整个表达式更整洁）。这一部分最有价值的是“后引用”变量$1和$2等。请看下面的例子：

```
$test = "Swap this and that.";
$test =~ s/Swap (.*) and (.*)\./Swap $2 and $1./;
print $test; # "Swap that and this."
```

做为规则表达式的最后一个例子，我们对那个用来去掉注释行的表达式做点扩展：

```
# Strip comments, version two.
while (<>) { s/#.*// ; print if /\S/ }
```

它能完成什么工作？它的工作情况是这样的：先读入一个文本行并把它保存到$_变量里。然后把井字号（#）及其后面的文字替换为空字符串。然后，如果还有东西剩下——也就是如果这一行上还有不是空格的字符的话，就把这一行打印出来。（千万不要真的用这个表达式来清除你Perl程序里的注释，它会把使用“$#array”语法的语句也删掉一半，还会把字符串或规则表达式里的“#”字符串及其后面的内容都删掉。再说了，留着注释总是会好一些。）

利用规则表达式匹配可以做许多许多的事情，大部分常见的用途在我们这个简单的介绍里也都提到过了。在shell提示符处输入“perldoc perlre”可以看到对匹配进行介绍的完整文档。

18.1.5 控制结构和子例程

我们一路走来，已经见过一些Perl语言的控制结构了，比如while循环、if和unless的语句内用法等，其他的控制结构都是比较容易掌握的了。

1. 测试

Perl有一个与C语言类似的if-elsif-else语句，两者之间的惟一区别是在Perl语言里必须把语句块放在花括号里。也就是说，你不能使用下面的语句结构：

```
if (/y(?:es)?/i)
        $answer = 1;
else
        $answer = 0;
```

而是必须使用下面这样的语句结构：

```
if (/y(?:es)?/i) {
        $answer = 1;
} else {
        $answer = 0
}
```

（当然，在实际编程的时候你是不会写出这种语句来的，但你肯定会写出“$answer = (/y(?:es)?/i);”这样的东西来。）Perl还允许你使用unless做为if的否定——如果这样做能够使你的代码更清晰的话。

2. 循环

我们已经见过while循环了，Perl语言还准备了一个until循环做为与while的对立面。此外，

还有两种for循环——与C语言相互对应。如下所示：

```
for ($i=0;$i<10;$i++) { print "Counter: $i\n";}
```

还有foreach循环，它用来对一个列表进行遍历，依次把每个元素赋值给$_变量或另外一个给定的变量。foreach或for语句都可以用来实现foreach循环。如下所示：

```
for $i (0..9) { print "Counter: $i\n"; } # As above.

foreach (@array) { print $_."\n"; } # Print each element of an array.
```

我们感兴趣的是对循环流的控制方法：一个next语句，类似于C语言中的continue语句，会在while类或for类循环结构里立刻跳到下一次循环去。请看下面这另外一个能够去掉程序注释（和空白行）的语句：

```
while (<>) { next if /^#/; next unless /\S/; print }
```

而last的作用就像C语言的break语句，它的作用是整个结束一个循环。如下所示：

```
# Get the Subject of an email, then give up:
while (<>) {
        last if ($subj)=/^Subject: (.*)/;
}
print "Subject was $subj\n";
```

很少使用的redo语句会再次跳转到循环的开始，但不对循环条件进行测试。你可以把它用在下面这种情况里：

```
while (<>) {
        if (/\\\s*$/) { # Line ends with a backslash - continuation
                $_.=<>;
                redo;
        }
        ...
}
```

Perl语言缺少C语言里的switch（或者其他语言里的case）语句。这是一个大家普遍关心的问题，有许多办法可以解决它；perlsyn和perlfaq7文档里介绍了几个解决方案。

3. 语句限定符

除了C语言语法风格的控制结构外，我们还可以在单独一条语句里通过“语句限定符”给它施加一个控制结构，我把这种情况称为“语句内”控制结构。我们已经见过if和unless的这种用法了：

```
print "Operation successful" if all_ok();

$logfile = "output.log" unless $nologging;
```

你也可以把while、until和for用做语句限定符。如下所示：

```
$input .= $_ while <>; # One (bad) way of concatenating a file

$cowscomehome=0;
main_loop() until $cowscomehome;

        print chr for (74, 117, 115, 116, 32, 65, 110, 111, 116, 104, 101,
        114, 32, 80, 101, 114, 108,32, 72, 97, 99, 107, 101, 114,);
```

但没有多少人会这样使用for结构。

4. 子例程

在Perl语言里我们并不明确区分函数和子例程；如果愿意，你可以让它们返回一个返回值。简单的子例程是像下面这样定义的：

```
sub greeting {
        print "Hello, world\n";
}
```

以后你就可以用“greeting();”（或者只用“greeting;”——如果你知道Perl不会把它解释为一个字符串的话）来调用这个子例程。参数的传递是通过“@_”数组完成的。请看下面的例子：

```
sub action {
        ($one, $two, $three) = @_;
        warn "Not enough parameters!"
            unless $one and $two and $three;
        print "The $one $two on the $three\n";
}

action("cat", "sat", "mat");
```

warn函数打印出一条错误信息但允许你的程序继续执行。另外有一个die函数，它的作用是打印出一条错误信息并立刻停止程序的执行。

“@_”是数组操作的缺省变量，就像“$_”是标量类型的缺省变量一样。传统上是使用shift命令来获取参数的。如下所示：

```
sub greeting2 {
        $name = shift;
        print "Hello, $name\n";
}

greeting2("Robert");
greeting2("world");
```

我们可以用return函数返回一个返回值，既可以返回一个标量数据，也可以返回一个列表——wantarray函数会告诉你调用者希望得到的是什么。请看下面的例子：

```
Sub mysub {
        # Do some stuff
        return unless defined wantarray; # void context, go home early
        if (wantarray) { # Want an array
             return @results;
        } else { # Just want a scalar
             return $summary
        }
        # Can't get here
}
```

Perl一般会把变量的作用范围设置为全局性的，这就意味着变量通常都是全局变量——子例程里进行的修改会影响到调用者。如果你真的需要局部变量，就要使用my函数。my函数声明的变量叫做字典范围变量（在Perl文档里经常简称为“字典变量”），它们甚至对当前例程的子例程都是不可见的；它们确实够“局部”的。如果你想让什么东西的作用范围是当前例程及其子例程，就要使用动态全局变量。请看下面的例子：

```
$myvar = "outside";
sub a { $myvar = "changed by a"; }
```

```
a();
print $myvar; # "changed by a";
$myvar = "outside";
sub b { my $myvar = "changed by b";
        print "In b: $myvar";
}
b(); #  "In b: changed by b"
print $myvar; # "outside"
```

18.1.6 文件的输入和输出

学习到现在，我们只见过一个文件输入输出操作：从标准输入读取一行数据。这可能比较适合编写过滤器类型的程序，但实际软件还需要具备读写其他文件的能力。

Perl里的文件访问操作通常都是通过文件句柄完成的。我们前面曾经说过Perl有三种变量类型，这其实并不准确；文件句柄就是一种非常特殊的变量类型。当程序启动运行的时候，会自动打开三个文件句柄，即标准输入STDIN、标准输出STDOUT和标准错误STDERR。我们可以通过一个特殊格式的print语句对这几个文件句柄进行写操作；事实上，“print list”就是“print STDOUT list”的简写形式。

```
if ($statusok) {
        print STDOUT "Processed successfully.\n";
} else {
        print STDERR "An error occurred...\n";
}
```

请仔细观察这段代码，在文件句柄和将要输出的文本之间是没有逗号的；文件句柄不是列表的组成部分。print有两种语法格式，一种带文件句柄，另一种不带。千万千万不要把这两种格式给弄混了。

我们曾经见过从标准输入读取一行数据的语句，它使用了“<>”——而这正是<STDIN>的简写形式。那文件系统里的文件又该怎样办呢？我们可以通过open函数创建出一个文件句柄；类似于C语言中的做法，我们需要给出一个文件句柄和一个文件名，还要说明我们是为输入还是为输出而打开这个文件的。但不再分配模式编号了，我们使用shell风格的语法"filename"（或"<filename"）表示以读方式打开文件；">filename"表示截短或创建文件并向它写数据；">>filename"向文件追加数据；等等（我们甚至可以打开管道进行读或写！）。下面这段程序从一个文件读入数据，把错误记录到一个日志记录文件里去：

```
open LOG, ">error.log"
        or die "Can't write on error.log: $!";
# $! tells you why not.
print LOG "Error logging started.\n";
open INPUT, $inputfile;
while (<INPUT>) {
        next unless /\S/; # Skip blank lines.
        next if /^#/; # You know what this does.
        chomp;
        if (fictitious_error_generator($_)) {
              print LOG "Error processing $_\n";
        }
        do_something_with($_);
}
close LOG;
close INPUT;
```

Perl会在程序结束时自动关闭文件句柄，但我们在这里用close函数关闭了它们。这是一个好习惯。

system()和``

在结束我们Perl旅行之前，我们再学习一下与系统互动——我们的意思是运行外部命令。有两个办法：一是system()——它的工作情况与C语言模型完全一样；二是反引号（``）——它的工作情况与shell模型差不多，但换行符的处理方式有所差异。这使两大阵营的程序员们都很高兴。

这两种方法有很大的区别：sytem()挂起运行中的程序，允许用户的互动，但并不归还程序的STDOUT；反引号归还程序的STDOUT，却不把它显示在屏幕上，因此，在交互程序里，用户可能看不到任何提示符 sytem()允许你取得被执行程序的返回值（把sytem()的返回值除以256，或者检查$?变量）；反引号则取不到。

基本原则是：如果你只是想运行一个程序，就使用sytem()；如果想知道它产生了什么样的输出，就使用反引号。两种办法都使用shell来处理命令行参数、管道、重定向等问题。但在使用sytem()时可以把这些参数安排到一个列表里而不是把它们看做是一整个参数，这样就可以省掉shell的那些预处理功能。

某些程序员，特别是出身自shell脚本程序的那些程序员，总是习惯于用shell来对付一切。而又有那么一些人，不到万不得以时决不使用shell。不管怎么说，两种方法都能使你的代码既整齐又有效率。

```
system("clear"); # Easiest way to clear screen.

$status = system("sendmail -q"); # Flush mail queue
# To avoid shell processing, use
# $status = system("sendmail","-q");
print STDERR "Something funny happened to sendmail: $?"
        unless $status==0;

$mail= `frm`;
if ($mail !~ /^You have no mail\./) {
        print "You have new mail:\n";
        print $mail;
        # $mail will contain newlines, so we don't need them.
}
```

18.2 一个完整的例子

实际编写一个实用Perl程序所必须的东西我们都已经学得差不多了，现在来看看如何编写我们的Perl版CD唱盘数据库软件。这其实只是增加了一两个Perl特色的shell脚本程序版的简单转换而已。事实上，我们并没有去掉shell脚本程序版中我们不想要的几个小缺陷——比如说，你还是不能在曲目名称里使用逗号。我们将在这章的结尾看到这个数据库软件一个完整的重写版本。就现在来说，你可能更愿意把这个程序和第2章里的脚本程序做逐行的比较。

我们也不是一点改动也没做，以前我们是在文件里对数据进行处理的，但现在我们将在程序的一开始就把文件读到数组里来，然后在程序的末尾把它们写回到文件去。这就避免了与临时文件打交道的麻烦（也避免了我们在Perl里不容易做到同时对一个文件进行读写的问题）。

1) 首先，像shell脚本程序一样，我们需要告诉内核这是一个Perl脚本，然后是我们的版权声明。如下所示：

```
#! /usr/bin/perl -w

# Perl translation of chapter 2's shell CD database
# Copyright (C) 1999 Wrox Press.

# 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; either version 2 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-#1307  USA
```

2) 下面是全局变量的定义。请注意最后一行：我们为键盘中断（键盘上的“Ctrl-C”组合键）设置了一个信号处理器子例程。我们把想做的工作全部放在一个“sub {}”结构里并直接取得它的返回值，这就创建出一个所谓的“匿名子例程引用”来。这个子例程引用调用子例程tidy_up把曲目和标题数组输出到文件，然后退出运行。tidy_up子例程写数据到文件的方式和我们把它们读到数组里来的方式差不多。惟一有点难办的是换行符的处理：文件要用换行符来分隔数据记录，但数组里面不需要有换行符。所以我们在读入数据的同时把它们用chomp函数“砍”掉了，在写出数据时又把它们加了上去。

```
$menu_choice="";
$title_file="title.cdb";
$tracks_file="tracks.cdb";
$temp_file="/tmp/cdb.$$";
$SIG{INT} = sub { tidy_up(); exit; } ;

sub read_in {
open TITLES, $title_file or die "Couldn't open $title_file : $!\n";
while (<TITLES>) { chomp; push @titles, $_ };
close TITLES;

open TRACKS, $tracks_file or die "Couldn't open $tracks_file : $!\n";
while (<TRACKS>) { chomp; push @tracks, $_ };
close TRACKS;
}

sub tidy_up {
# Die aborts with an error, and $! is the error message from open()
open TITLES, ">".$title_file or die "Couldn't write to $title_file : $!\n";
foreach (@titles) { print TITLES "$_\n"; }
close TITLES;

open TRACKS, ">".$tracks_file or die "Couldn't open $tracks_file : $!\n";
foreach (@tracks) { print TRACKS "$_\n"; }
close TRACKS;
}
```

3) 两个读取键盘输入的小函数。如下所示：

```
sub get_return {
        print "Press return ";
        <> # Get a line from STDIN, and ignore it.
}

sub get_confirm {
        print "Are you sure? ";
```

```
        while (1) {
                $_ = <>; # Get a reply into $_
                return 1 if (/^y(?:es)?$/i); # 1 is true, not 0
                if (/^no?$/i) {
                        print "Cancelled!\n";
                        return 0;
                }
                print "Please enter yes or no.\n";
        }
}
```

4) 现在，显示主菜单，然后从用户那里得到一个选择。“<<EOF”语法是一个here文档用法，它会一直输出到遇见单词“EOF”（或者你设定的任何休止字符串）为止。

```
sub set_menu_choice {
        print `clear`; # Shelling out to clear screen. Yuck.

        print <<EOF;
   a) Add new CD
   f) Find CD
   c) Count the CDs and tracks in the catalog
EOF

        if ($cdcatnum) {
                print "   l) List tracks on $cdtitle\n";
                print "   r) Remove $cdtitle\n";
                print "   u) Update track information for $cdtitle\n";
        }

        print "   q) Quit\n\n";
        print "Please enter choice then press return\n";
        chomp($menu_choice=<>);
        return
}
```

5) 先是往数组里添加新唱盘的标题记录项的子例程，它只有一行语句。然后是添加曲目资料的子例程。

```
sub insert_title {
        push @titles, (join "," , @_);
}

sub insert_track {
        push @tracks, (join "," , @_);
}

sub add_record_tracks {
        print "Enter track information for this CD\n";
        print "When no more tracks enter q\n";
        $cdtrack=1;
        $cdttitle="";
        while ($cdttitle ne "q") {
                print "Track $cdtrack, track title? ";
                chomp($cdttitle=<>);
                if ($cdttitle =~ /,/) {
                        print "Sorry, no commas allowed.\n";
                        redo;
                }
                if ($cdttitle and $cdttitle ne "q") {
                        insert_track($cdcatnum,$cdtrack,$cdttitle);
                        $cdtrack++;
                }
        }
}
```

6) 现在来编写add_records子例程，它的作用是把一张新CD唱盘的记录添加到数据库里去。

```
sub add_records {
        print "Enter catalog name ";
        chomp($cdcatnum=<>);
        $cdcatnum=~ s/,.*//; # Drop everything after a comma.

        print "Enter title ";
        chomp($cdtitle=<>);
        $cdtitle =~ s/,.*//;

        print "Enter type ";
        chomp($cdtype=<>);
        $cdtype =~ s/,.*/,;

        print "Enter artist/composer ";
        chomp($cdac=<>);
        $cdac =~ s/,.*//;

        print "About to add a new entry\n";
        print "$cdcatnum $cdtitle $cdtype $cdac\n";

        if (get_confirm()) {
                insert_title($cdcatnum,$cdtitle,$cdtype,$cdac);
                add_record_tracks();
        } else {
                remove_records();
        }
}
```

7) 因为我们已经有了一个记录行数组，查找CD唱盘的工作就非常简单了。我们只需遍历数组找到匹配就行了。用Perl的grep函数更容易实现这一功能，因为这个函数就是为检索目的设计的。

```
sub find_cd {
        # $asklist is true if the first member of @_
        # (That is, the first parameter) is not "n"
        $asklist = ($_[0] ne "n");

        $cdcatnum="";
        print "Enter a string to search for in the CD titles ";
        chomp($searchstr=<>);
        return 0 unless $searchstr;

        # The \Q and \E metacharacters stop other metacharacters
        # from working, so question marks, asterisks and so on
        # in titles aren't dangerous.

        @matches = grep /\Q$searchstr\E/, @titles;
   if (scalar @matches == 0) {
        print "Sorry, nothing found.\n";
        get_return();
        return 0;
   } elsif   (scalar @matches != 1 ) {
   print "Sorry, not unique.\n";
   print "Found the following:\n";
   foreach (@matches) {
         print "$_\n";
   }
   get_return();
   return 0;
}
($cdcatnum,$cdtitle,$cdtype,$cdac) =
    split "," , $matches[0];
unless ($cdcatnum) {
    print "Sorry, could not extract catalog field\n";
    get_return();
    return 0;
}
```

```
print "\nCatalog number: $cdcatnum\n";
print "Title: $cdtitle\n";
print "Type: $cdtype\n";
print "Artist/Composer: $cdac\n\n";
get_return();
if ($asklist) {
   print "View tracks for this CD? ";
   $_ = <>;
    if (/^y(?:es)?$/i) {
        print "\n";
        list_tracks();
        print "\n";
        }
}
return 1;
}
```

8) 在剥离了从数组里删除旧曲目的功能之后，update_cd很容易实现。我们用另一个grep函数来做这项工作，但这一次我们要用“!/regexp/”语法否定那个规则表达式。

```
sub update_cd {
        unless ($cdcatnum) {
                print "You must select a CD first\n";
                find_cd("n");
        }
        if ($cdcatnum) {
                print "Current tracks are :-\n";
                list_tracks();
                print "\nThis will re-enter the tracks for $cdtitle\n";
                if (get_confirm()) {
                        @tracks = grep !/^$cdcatnum,/, @tracks;
                        add_record_tracks();
                }
        }
}
```

9) 因为数据都已经被保存到数组里去了，所以统计数据库内容的工作是轻而易举的。

```
sub count_cds {
        print "Found ".(scalar @titles)." CDs, ";
        print "with a total of ".(scalar @tracks)." tracks.\n";
        get_return();
}
```

10) 我们已经见过怎样才能使用grep和一个规则否定表达式从一个数组里删除数据项了；现在再做一次：

```
sub remove_records {
        unless ($cdcatnum) {
                print "You must select a CD first\n";
                find_cd("n");
        }

        if ($cdcatnum) {
                print "You are about to delete $cdtitle\n";
                if (get_confirm()) {
                        @titles = grep !/^$cdcatnum,/, @titles;
                        @tracks = grep !/^$cdcatnum,/, @tracks;
                        @cdcatnum="";
                        print "Entry removed";
                }
                get_return();
        }
}
```

11) list_tracks需要分页显示，所以我们需要把数据写到一个临时文件，再用shell来对它进行

操作。

```
sub list_tracks {
        unless ($cdcatnum) {
                print "No CD selected yet.\n";
                return
        }
        open(TEMP, ">$temp_file")
             or die "Can't write to $temp_file: $!\n";
        @temp = grep /^$cdcatnum,/ , @tracks;
        if (scalar @temp == 0) {
                print "No tracks found for $cdtitle\n";
        } else {
                print TEMP "\n$cdtitle :-\n\n";
                foreach (@temp) {
                        s'^.*?,//; # Remove the first field
                        print TEMP $_."\n";
                }
                close TEMP;
                system("more $temp_file");
                unlink($temp_file); # Delete it.
        }
        get_return();
}
```

12) 下面是主程序部分。千万不要忘记在退出之前要把数组里的数据写回到文件去。在开始读取文件之前，我们还必须保证它们确实存在，如果不存在就建立它。当然，我们也不必非得这样做——如果文件不存在，我们可以不必报告出错，数组都是空的，文件可以等到我们退出程序时再创建不迟。

```
# File tests work like shell
system("touch $title_file") unless ( -f $title_file );
system("touch $tracks_file") unless ( -f $tracks_file );

read_in();

system("clear");
print "\n\nMini CD manager\n";
sleep(3);
while (1) {
        set_menu_choice();
        if ($menu_choice =~ /a/i) { add_records(); }
        elsif ($menu_choice =~ /r/i) { remove_records(); }
        elsif ($menu_choice =~ /f/i) { find_cd("y"); }
        elsif ($menu_choice =~ /u/i) { update_cd(); }
        elsif ($menu_choice =~ /c/i) { count_cds(); }
        elsif ($menu_choice =~ /l/i) { list_tracks(); }
        elsif ($menu_choice =~ /b/i) {
                print "\n";
                foreach (@titles) {
                        print "$_\n";
                }
                print "\n";
                get_return();
        }
        elsif ($menu_choice =~ /q/i) { last; }
        else { print "Sorry, choice not recognized.\n"; }
}

tidy_up();
exit;
```

18.3 命令行上的Perl

现在，我们已经见过一个完整的Perl程序了，再来点Perl的日常使用怎么样？这么说吧，

Perl就像sed和awk一样非常适合用做过滤器来完成日常的系统维护工作。事实上，Perl提供了一组相当有用的命令行选项来帮助我们完成这些工作。就像我们在这一章的开始见到的“-w”一样，完全可以把这些选项放在我们脚本的“#!”语句行上。

首先要介绍的是“-e”。类似于sed和awk中的用法，它允许我们执行只有一行的Perl脚本，如下所示：

```
$ perl -e 'print "Hello, world\n";'
Hello, world
$
```

类似地，我们可以给Perl提供一个文件名让它用做标准输入，可以把我们熟悉的注释清理语句写成下面这个样子：

```
$ perl -e 'while(<>) { print unless /^#/ }' myfile
```

它会去掉myfile里使用的注释行后再打印出来。因为循环处理文件的每一行是一个经常要用到的操作，所以Perl专门为此准备了一个特殊的语法：“-n”选项。如下所示：

```
$ perl -n -e 'print unless /^#/' myfile
```

我们甚至可以不使用print命令——下面代码中的“-p”标志就可以完成这一工作：

```
while (<>) {
        ...
        print or die "-p destination: $!\n";
}
```

这在查找与替换操作使用的规则表达式里很有用，比如说，“perl -p -e 's/foo/bar/g' file”会把文件打印出来，并且把每一处出现“foo”的地方都替换为“bar”。现在，我们再前进一步，假设我们确实想把文件里的每一个“August”都替换为“September”。普通的做法是先把替换后的输出结果保存到一个临时文件里，然后通过mv命令用临时文件替换掉旧文件。但在Perl里不必这样麻烦。Perl通过“-i”选项支持对文件进行“当时当地”的修改。请看，“perl -p -i -e 's/August/September/g' myfile”相当于shell里的下列命令：

```
$ sed 's/August/September/g' myfile > tmpfile;
$ mv tmpfile myfile;
```

想给原始文件做一个备份吗？没问题！给“-i”项加上一个后缀就行，命令“perl -p -i.bak -e 's/August/September/g' myfile”既完成了对文件的修改，又保存了一个名为myfile.bak的备份文件。这个办法可以轻易地建立起功能非常强大的过滤器和文件编辑器。

那么，命令行上还能变出其他魔术吗？我们已经见过“-w”可以为你的脚本打开附加的错误报警功能。现在再向大家介绍“-c”选项——它可以在不运行脚本的前提下对程序进行语法检查；和“-d”调试器选项——它是一个功能强大的追踪脚本程序错误问题的工具。最后，再看看这个：是不是已经厌烦了要在print语句里写上“\n”却又要在输入里用chomp“砍”掉它们？用“-l”选项打开行自动处理功能试试。它将自动去掉通过readline操作符读入脚本的任何“垃圾”，而通过print语句输出的任何东西又都会自动地加上“\n”。把“-p”和“-e”选项用在一起真管用！

18.4 模块

如果你是在认认真真地编写Perl程序，就会逐渐地发现自己正在编写的代码有许多是以前曾经编写过的，比如网络程序设计、文本和HTML的处理、命令行参数的处理、保存数据到文件，等等。也许还会出现这样的情况：你认为有些事情在Perl里是很难完成的，因此需要使用一个C语言扩展等。

Perl模块就是用来对付这两种情况的：它们提供了代码的再使用性，就像C语言里的函数库；同时也允许与其他程序设计语言进行通信和交流。我们不准备在这里深入探讨各种模块的使用方法和如何建立你自己的模块，但将尽量让大家对可以利用模块来完成的工作有一个初步的认识。

18.4.1 CPAN

CPAN是一个重要的Perl模块仓库站点，全称是“Comprehensive Perl Archive Network”（智能化Perl档案网）。正如它名字里说的那样，它是一组几乎囊括了所有Perl模块的镜像档案；模块之多超出你的想象。CPAN模块仓库的入口点是http://www.cpan.org/或者是http://www.perl.com/CPAN/；它们会把你引导到离你最近的镜像站点。从CPAN上你可以下载文档、教程以及最新版本的Perl源代码，但你最想从它那里得到的应该是模块了。在本地镜像站点的/modules/bycategory子目录里可以查到其中存放着的模块的清单，而各种CPAN模块的介绍说明则可以在主目录中的CPAN.html文件里查出来。

18.4.2 安装一个模块

从CPAN下载到一个模块之后，我们就可以按照下面的步骤把它安装到自己的系统里。我们以Net::Telnet模块为例进行说明。假设我们已经从CPAN下载了Net-Telnet-3.01.tar.gz文件，现在就需要执行下面这些命令：

```
$ tar zxf Net-Telnet-3.01.tar.gz
$ cd Net-Telnet-3.01
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Net::Telnet
$ make install
```

最后一个步骤可能需要具备特定的权限才能把模块文件安装到Perl子目录里去。

安装模块还有另外一个办法。在你的Perl发行版本里带有一个由Andreas Konig编写的名为CPAN的模块，它可以引导你完成整个安装过程。你只需敲入下面的命令：

```
$ perl -MCPAN -e shell
```

再根据提示进行操作就行了。这个办法可以用来安装成批的模块，比如libwww这样拥有大量依赖关系的模块等。

18.4.3 perldoc命令

一切模块（包括Perl自带的模块，以及Perl语言本身）都应该带有完整的文档。这些文档是

用POD（Plain Old Document）语法格式写出来的，我们可以用perldoc命令来阅读它们。举例来说，要想阅读刚才安装的Net::Telnet模块的文档，我们要使用命令：

```
$ perldoc Net::Telnet
```

如果想进一步了解Perl语言，可以从“perldoc perl”开始；然后认真阅读它指向的所有页面——当然这需要你有足够耐心。perldoc有两个很有用的选项，一个是“-q keyword”，它的作用是在浩瀚的Perl FAQ里查找指定的关键字。举例来说，“perldoc -q Y2K”会把千年虫方面的资料找出来。另一个选项是“-f function”，它会把perlfunc文档里与function函数有关的章节找出来。比如说，你可以试试“perldoc -f unshift”。

18.4.4 网络功能

这些模块对我们的帮助可以说是雪中送炭。你需要从CPAN站点下载一组相关的模块。

1. LWP

LWP（libwww-perl）是涵盖了Web服务器操作和客户操作的一组模块。让我们假设整个模块组都已经安装好了。下面先通过模块LWP::Simple来完成一些简单的操作。首先下载HTML格式的当日新闻，如下所示：

```
use LWP::Simple;
$news = get "http://news.bbc.co.uk/text_only.htm";
```

现在用HTML分析器（HTML-Parser）库里的HTML::LinkExtor模块把其中的链接全部提取出来。如下所示：

```
use HTML::LinkExtor;
$p = HTML::LinkExtor->new();
$p->parse($news);
@links = $p->links; # Array of all the links in the file
```

好了，大家可能还不太看得懂这段代码，但这只是因为我们还没有向大家介绍面向对象的程序设计。不管怎么说，就像C语言的函数库一样，Perl模块可以极大地简化程序设计工作。

2. IO::Socket

现在，学习点套接字网络功能怎么样？还记得那个连接到一个时间服务器并获取了当地时间的C语言程序吗？下面是我们在Perl里的做法。套接字库IO::Socket是一个应该随你的Perl发行版本一起提供的标准模块。

```
use IO::Socket;
$host = "localhost" unless ($host = shift);
$socket = IO::Socket::INET->new(
        PeerAddr => $host,
        PeerPort => "daytime")
or die "Couldn't connect to $host: $@";
$time = <$socket>; # Sockets act like filehandles.
print $time;
```

我们当然可以用Perl内建的各种套接字函数（socket、connect、gethostbyname等）来完成这项工作，但现在的办法简洁得多。让模块来干这些事情吧。

3. 网络模块

如果你打算用Perl语言来实现系统任务的自动化，就会发现Net::系列模块的用处。我们刚才安装的Net::Telnet模块提供了对telnet会话的访问和控制手段，包括自动化地连接、登录和执行命令等。类似地，Net::FTP模块（属于libnet模块组）能够帮助我们自动化地完成FTP任务。下面是我们从CPAN下载MD5模块的操作过程：

```
use Net::FTP;
$ftp = Net::FTP->new("ftp.cpan.org") or die "Couldn't connect: $@\n";
$ftp->login("anonymous");
$ftp->cwd("/pub/modules/by-name/MD5/");
$ftp->get("MD5-1.7.tar.gz");
$ftp->quit();
```

这个模块系列中还包括：用于新闻阅读和发布的Net::NNTP模块（属于libnet模块组），用于DNS查询的Net::DNS模块（属于Net-DNS模块组），用来收取电子邮件的Net::POP3模块（属于libnet模块组），以及Net::Ping模块、Net::Whois模块（属于Net-Whois模块组）和Net::IRC模块（属于Net-IRC模块组），等等。

18.4.5 数据库

在Perl里，保存和检索数据的手段是很多的。我们已经见过扁平文件型数据库的处理办法，但更常用的方法是我们在第7章里学习的DBM（数据库管理）系统。我们可以通过把DBM系统与哈希表相结合的办法来访问它们。这就意味着哈希表里的数据将与磁盘上的数据联系起来。我们使用标准的AnyDBM_File模块做为我们与GDBM库系列的接口（事实上，AnyDBM_File可以访问许多种DBM软件包）。请看下面的例子：

```
use AnyDBM_File;
tie %database, "AnyDBM_File", "data.db";
# We can now use %database as a normal hash; any adding, deleting or
# modifying of keys will be reflected in the data file.

untie %database;
```

还有许多类似的模块可以把数据结构和文件联系起来。举例来说，DB_FILE模块可以通过Berkeley版本的DB库把一个数组和一个文本文件联系起来——在我们CD唱盘数据库例子里完全可以用它来取代程序开始和结束时对数据文件进行的读写操作。MLDBM（Mutil-level DBM，多层次数据库）是一个用来在DBM里保存复杂数据结构的模块，我们将用它来实现CD唱盘数据库的最终版本。

最后，我们可以通过DBI接口把数据保存到MySQL、PostgreSQL、Oracle、Informix等类型的关系数据库里去，并且还能够执行SQL查询命令和语句。相关模块包括做为程序接口的DBI模块以及做为特定数据库驱动程序的DBD::[Oracle, mysql . . .]等模块。

18.5 改进版CD唱盘数据库

对Perl的能力大家应该已经很了解了，我们准备把CD唱盘数据库程序用Perl语言完整地编写出来。这里用到了几个我们没有介绍过的概念（引用下标、嵌套数据结构等），但这不会有什么影响。即使对这里发生的一切都不明白也没有关系，我们的目的并不在此——我们的目的是想让

大家看看一个“真正的”Perl程序到底是什么样子的。

1) 我们还是像前面那样以注释开始我们的程序。如下所示：

```
#! /usr/bin/perl -w

# Perl translation of chapter 2's shell CD database
# Copyright (C) 1999 Wrox Press.

# 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; either version 2 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-#1307 USA
```

2) 我们先把主程序列在这里，函数的改进随后进行。如下所示：

```
use MLDBM qw(AnyDBM_File);
my $record;
tie(%tmp, "MLDBM", "cddb.db")
or die "Couldn't tie DB.\n"; # Scary complex hash contains the whole DB.
%database = %tmp; # Overcome a limitation in MLDBM. *sigh*

# Tidy up nicely
$SIG{INT} = sub { %tmp = %database; untie %tmp } ;

system("clear");
print "\n\nCD Database Manager\n\n";

while (1) {
        my $menu_choice = main_menu($record);
        if ($menu_choice eq "a") { $record = add_cd(); }
        elsif ($menu_choice eq "r") { remove_cd($record); undef $record; }
        elsif ($menu_choice eq "f") { $record = find_cd("y"); }
        elsif ($menu_choice eq "u") { update_cd($record); }
        elsif ($menu_choice eq "c") { count_cds(); }
        elsif ($menu_choice eq "l") { list_tracks($record); }
        elsif ($menu_choice =~ /q/i) { last; }
        else {
                print "Can't get here.\n";
        }
}

%tmp=%database;
untie %tmp;
```

3) 现在显示主菜单并检查用户的选择是否有效。如下所示：

```
sub main_menu {
    my $record = shift;
    my $choice;
    my $title = $database{$record}->{title} if $record;
    print <<EOF;

    Options :

       a) Add new CD
       f) Find CD
       c) Count CDs and tracks in the catalogue
    EOF
    if ($record) {
```

```
            print "    l) List tracks on $title\n";
            print "    r) Remove $title\n";
            print "    u) Update entry for $title\n";
    }
    print "    q) Quit\n";
    print "Your choice: ";
    while (1) {
            $choice=lc(<>);
            substr($choice,1)="";
            # Now, we see if the choice is contained in the string of
            # acceptable options. (Which includes 1, r and u if we've
            # selecetd a record.) by using it as a regexp. Looks weird?
            return $choice if ("afcq".($record?"lru":"") =~ /$choice/);

            # If not, that's invalid
            print "Invalid choice.\nTry again: ";
    }
}
```

4) 下面来解决向数据库添加记录的问题。数据库实际上是一个相当复杂的哈希表：键字是分类编号，而每个键值本身又是一些哈希表。这些哈希表的键字分别是“title”、“type”、“artist”和“tracks”。这就是我们在上面使用“$database{$record} -> {title}”这种古怪写法的原因——“$database{$record}”是一个哈希表（它实际上只是某个哈希表的一个引用下标，C语言程序员可能会把它们想象为指针。引用下标的详细介绍请参考perlref文档）。“-> {title}”语法在这个哈希表引用下标里查出“title”键字的对应值。“tracks”的键值当然应该是一个曲目数组的引用下标，它指向一个哈希表里的哈希表里的数组——够拗口的了吧，没点时间还真习惯不过来呢。

```
sub add_cd {
        while(1) {
                print "Enter catalog number: ";
                chomp($record=<>);
                if (exists $database{$record}) {
                        print "Already exists. ";
                        print "Please enter a different number.\n";
                } else {
                        last;
                }
        }

        print "Enter title: ";
        chomp($title=<>);

        print "Enter type: ";
        chomp($type=<>);
        print "Enter artist/composer: ";
        chomp($artist=<>);

        $database{$record}= {
                "title" => $title,
                "type" => $type,
                "artist" => $artist
        };

        add_tracks($record);
        return $record; # Tell the main menu the new record number.
}
```

5) 这里是添加曲目的子例程。我们就是在这里引出数组引用概念的。

```
sub add_tracks {
        my $record = shift;
        print "Enter track information for this CD\n";
        print "Enter a blank line to finish.\n\n";
```

```
        my $counter=0; my @tracks;
        while (1) {
                print ++$counter.": ";
                chomp($track=<>);
                if ($track) {
                        # @{...} means "interpret as an array"
                        push @{$database{$record}->{tracks}}, $track;
                } else {
                        last;
                }
        }
}
```

6) 这段查找CD唱盘用的代码有点复杂，这是因为我们需要为哈希表里的每个$record值去遍历“$database{record} -> {title}”的每一个值。还用grep来救火……

```
sub find_cd {
        $view = ($_[0] eq "y");

        print "Enter a string to search for: ";
        chomp($search=<>);

        # For each key, (record) add the key to the @found array if the
        # title field of that record contains the search string.
        @matches =  grep {$database{$_}->{title} =~/\Q$search\E/ }
                        keys %database;
        if (scalar @matches == 0) {
                print "Sorry, nothing found.\n";
                return;
        } elsif (scalar @matches != 1 ) {
                print "Sorry, not unique.\n";
                print "Found the following:\n";
                foreach (@matches) {
                        print $database{$_}->{title}."\n";
                }
                return;
        }
        $record=$matches[0];
        print "\n\nCatalog number: ".$record."\n";
        print "Title: ".$database{$record}->{title}."\n";
        print "Type: ".$database{$record}->{type}."\n";
        print "Artist/Composer: ".$database{$record}->{artist}."\n\n";

        if ($view) {
                print "Do you want to view tracks? ";
                $_ = <>;
                if (/^y(?:es)?$/i) {
                        print "\n";
                        list_tracks($record);
                        print "\n";
                }
        }
        return $record;
}
```

7) 现在，列出曲目清单的工作已经很简单了！

```
sub list_tracks {
        my $record = shift;
        foreach (@{$database{$record}->{tracks}}) {
                print $_."\n";
        }
}
```

8) 更新一张CD唱盘意味着删除一个旧曲目，再增加一个新曲目。如下所示：

```
sub update_cd {
        my $record = shift;
        print "Current tracks are: \n";
        list_tracks($record);
        print "\nDo you want to reenter them?\n";
        if (($_ = <>) =~ /^y(?:es)?$/i) {
                # Remove the old entry from the hash
                delete $database{$record}->{tracks};
                add_tracks($record);
        } else {
                print "OK, canceling.\n"
        }
}
```

9) 类似地，删除一张CD唱盘就意味着删除与之对应的哈希表数据项。如下所示：

```
sub remove_cd {
        my $record = shift;
        print "\nDo you want to delete this CD?\n";
        if (($_ = <>) =~ /^y(?:es)?$/i) {
                delete $database{$record};
        } else {
                print "OK, cancelling.\n"
        }
}
```

10) 最后，CD唱盘的统计工作已经很简单了——只要计算一下哈希表里的键字个数就行了。但曲目的统计工作需要有点小技巧：我们把数据库里每一个键字对应的曲目数组按标量类型的上下文进行统计，再把这些值加在一起。（你也可以用map()函数完成统计工作，但那样不如现在的办法来得清晰。）

```
sub count_cds {
        my $totaltracks=0;
        print "Found ".(scalar keys %database)." CDs and ";
        foreach (keys %database) {
                $totaltracks+= scalar @{$database{$_}->{tracks}};
        }
        print $totaltracks." tracks.\n";
}
```

18.6 本章总结

在这一章里，我们学习了Perl程序设计语言提供的部分功能的使用方法，接触了它的一些模块，并使用Perl语言实现了我们的CD唱盘数据库软件。

第19章　因特网程序设计：HTML

我们将在这一章里学习一个图形信息程序设计方法。我们不需要编写绘制图形和文本的程序，我们只要指定想看的东西，再使用另外一个程序进行浏览就可以达到目的。

我们将学习使用World Wide Web语言HTML编写和建立一个服务器（程序）的方法，这样我们就能使用一个客户浏览器（程序）通过网络来查看我们的文档。我们将学习以下几个方面的内容：

- World Wide Web的历史和本质。
- HTML文档的结构。
- 标签、表格、图像和超文本链接锚点。
- 客户端和服务器端的可点击图片。
- 在Web上建立服务器和页面的技巧。

19.1　什么是World Wide Web

World Wide Web经常被简称为WWW或Web，它于1989年起源于日内瓦的CERN实验室，Tim Berners-Lee当时正在研究传播信息的手段。到了1992年，CERN把一个实验性质的接口和协议放到了公共域里。它迅速被因特网团体所接受，而Web也因此而诞生了。

World Wide Web由三个部分组成：容纳着文档和图像的服务器、用来传输信息的网络和显示这些信息的客户——浏览器。

在这一章里，我们将简要地介绍一下服务器（不管是UNIX版本还是其他计算机平台，都有许多可以免费获得的服务器软件）的建立过程，但主要学习内容将是如何对准备向外提供的信息进行定义。

虽然提到World Wide Web时说的几乎总是因特网（Internet），但你完全可以使用同样的软件在内部网络（即所谓的intranet）上提供信息。公司完全能够在自己的内部建立起提供内部信息的WWW服务器，并且许多公司已经这样做了。正如我们将会看到的那样，这其实并不困难，并且它能够以一种非常易于使用的格式提供信息。

WWW上的文档几乎都是超文本文档，也就是说，大多数文档就其本身来说都是不完整的，它们需要链接上其他的文档，而它们自己也会成为其他文档的链接目标。

比如说，一家公司通过一个WWW站点发布当地的旅游信息。它可能只有一个顶层主页，但在这个主页上却提供了许多链接，通过这些链接可以查看到其他页面上的详细资料。因为WWW文档都是保存在一台计算机上的，所以这些链接可以是“活着的”，当它们被点击选中的时候，就会把用户带到一个不同的页面去。这个假想中的旅游信息服务还可以提供到邻近地区旅游信息提供商那里的链接，而邻近地区的商家又可能会提供一个指向本地区的链接做为回报。

World Wide Web的影响和吸引力很难用语言来表达。体会它的惟一办法就是亲自上阵！

19.2 术语

在继续前进之前，我们先来学习几个术语，把它们的意思弄明白。有好几个不同的标准是与Web密切相关的。

这些标准中有许多是在称为RFC文件的文档里定义的。它们散布在因特网上的各个地方，如果你在自己附近找不到提供RFC文件的站点，就可以去ds.internic.net机器上看看。如果你有FTP权限，就可以在子目录rfc中找到它们；如果你只有收发电子邮件的权限，可以向mailserv@ds.internic.net发一封邮件，把主题（即Subject）部分空着，在信体里写上“help”；该邮件服务器会回复你怎样才能得到RFC文件。

沿着http: //www.w3.org/上的链接，你还可以找到与World Wide Web有关的其他文档；这是World Wide Web论坛组织的站点。而下面是一些你可能会遇到的术语的简单解释。

19.2.1 超文本传输协议

超文本传输协议（The HyperText Transfer Protocol，HTTP）是用来在客户计算机和服务器计算机之间传输信息的协议。

19.2.2 因特网邮件多媒体扩展

因特网邮件多媒体扩展（Mutilmedia Internet Mail Extensions，MIME）最初是因特网电子邮件协议一个允许传输非纯文本的信息的扩展规定。但它的巨大发展已经超越了它最初的目标，现在成为了对HTTP协议上传输的信息格式进行定义的事实标准。为了让客户和服务器彼此了解被传输数据的格式，人们定义了许多种MIME类型和子类型。你可以在RFC1521号文件里查到更多资料。

19.2.3 标准通用置标语言

标准通用置标语言（Standard Generalized Markup Language，SGML）是一个用来定义文档格式的国际性标准，编号为ISO 8879:1986。它提供了一个标准化的文档结构定义方法。

19.2.4 文档类型定义

文档类型定义（Document Type Definition，DTD）是定义如何把SGML应用到一个具体的文档类型上的一组规定。

19.2.5 超文本置标语言

超文本置标语言（HyperText Markup Language，HTML）是SGML的一个具体实施方案，它定义了在WWW上使用的置标语言。它允许你对一个文档的布局和结构进行定义。在我们编写本书的时候，最新的标准是HTML 4.0（定义在W3C论坛的推荐标准REC-html40文件里，你可以在

http://www.w3.org/TR/REC-html40/处找到它）。但如果XHTML被接受为一个推荐标准的话，它就会被XHTML取代。

HTML4.0定义了三个DTD类型，每个DTD类型支持的标签个数都不一样。这是因为这一版HTML标准的评估工作又去掉了许多在该标准的前两个版本（即2.0版和3.2版）里引进的改动。HTML标准早先在创建新标签方面受两大浏览器供应商Netscape（网景公司）和Microsoft（微软公司）的影响很大，除有限的修改外，基本上是照搬照抄。但这偏离了HTML标准只对文档结构进行定义的初衷。这个标准的前两个版本引进了一些纯样式性质的元素，而它们本不应该被纳入这个标准。在HTML 4.0标准里，许多对样式进行定义的标签都已经被标记为删除，人们把这些标签称为“贬值标签”。在DTD的三个版本里还有一些数目不等的被标记为删除的贬值标签。这三个DTD版本是：

- Strict（严格）——只包括那些没有贬值和不出现在frameset（结构设置）文档里的标签。
- Transitional（过渡）——包括除出现在frameset文档里的标签以外的所有标签，包括那些带贬值标记的标签。
- Frameset（结构设置）——包括允许在DTD过渡版本里使用的和允许在frameset文档里使用的所有标签。

把标准分为几个不同的版本的理由是为了让人们既可以继续写出能够在老浏览器上浏览的文档，又能够享受到HTML 4.0带来的新功能。

我们在这一章里使用的是过渡版本的HTML标准，你编写的HTML文档几乎可以在所有的浏览器上被浏览。

19.2.6 可扩展置标语言

可扩展置标语言（Extensible Markup Language，XML）根据SGML标准产生的另外一种语言。如果说HTML是SGML的一个具体实施方案，那就应该说XML是SGML一个更简化的产物。人们认为它既有SGML的功能和灵活性，又避免了SGML的复杂性。XML具备SGML所有的常用功能，同时，它也象SGML那样是一个元语言——即一个可以用来描述其他语言的语言，它描述的是类似于HTML这样的根据用户需要而定制出来的置标语言。XML的实力在于它能够以一种人们能够阅读的形式对结构化数据进行灵活定义。如果不仔细看，XML和HTML的外在形式非常相似，而它的设计原则之一就是要保证有与HTML互换操作的能力。

19.2.7 层叠样式表

层叠样式表（Cascading Style Sheets，CSS）是一个国际性标准，它允许你把与样式有关的内容从HTML文档里分离出去。这个标准目前有两个版本：一个是1996年12月17日宣布的CSS1--它可以在http://www.w3.org/ TR/REC-CSS1处找到；该标准最新的版本是1998年5月宣布的CSS2--它可以在http:// www.w3.org/TR/REC-CSS2处找到。

19.2.8 可扩展超文本置标语言

可扩展超文本置标语言（Extensible HyperText Markup Language，XHTML）目前还是一个

待推荐标准（请参考http://www.w3.org/TR/1999/PR-xhtml1-19990824），它的目标是把HTML定义为XML的一个实施方案。从实际使用角度看，你的主页可能会与HTML 4.0里的情况有细微的差异，但它保证了用户写出来的HTML页面的确符合HTML 4.0里的某个DTD类型。

19.2.9 统一资源定位器

统一资源定位器（Uniform Resource Locator，URL）是指定一项资源的方法。它通常由以下几个部分组成：一个协议名称、一个冒号（:）、两个斜线字符（/）、一个机器名和一个到达资源的路径。如果省略了文件的路径名，一般做法是返回一个缺省文件。

我们来举几个例子。文档名是rfc1866.txt，它保存在名为ds.internic.net的计算机里，如果想对它进行FTP访问，正确的URL定义就是ftp: //ds.internic.net/rfc/rfc1866.txt。Wrox出版社的WWW主页可以在http: //www.wrox.com/处找到。

19.2.10 统一资源标识符

URI是网络资源一种更具普遍意义的命名机制，它可以说是比较针对具体协议的URL命名机制的超集。详细资料请参考RFC1734和RFC1630。

19.3 一个HTML文档

在具体学习HTML之前，先来看看下面的HTML小文档，下面是它的内容和它在浏览器里的显示情况。

动手试试：一个简单的HTML文档

下面是这个非常简单的HTML文档的源代码。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">

<HTML>
<HEAD>
<TITLE>A Simple HTML Document, html1.html</TITLE>
</HEAD>
<BODY>
<H1>This is a title</H1>
<P>
And here is some ordinary text.
</P>
</BODY>
</HTML>
```

如果用一个WWW客户程序来浏览这个文件，其显示效果会根据我们具体使用的客户程序的不同而稍有差异。要想打开这个文件，可以通过“file://directory/html1.html”的形式给出一个URL，也可以通过菜单进行选择。下面是网景公司的Navigator浏览器在X窗口系统里加载这个文件后显示出来的画面如图19-1所示。

操作注释：

大家可能已经看出来了，HTML的标记部分要放在尖括号（< >）里，并且是不显示的。它

们的作用是控制其他文本的显示效果。

文档最开始处的“<! DOCTYPE . . .”部分表明该文档符合HTML 4.0标准过渡版本的要求。如果在编写HTML文档时没有考虑让它符合每个特定的标准，可以省略DOCTYPE定义。许多HTML文档里都没有这个定义。用上浏览器最新最强的功能当然很不错，但在大多数情况下，尽量同时兼顾HTML标准新老版本的要求会使你的HTML页面有更多的观众。

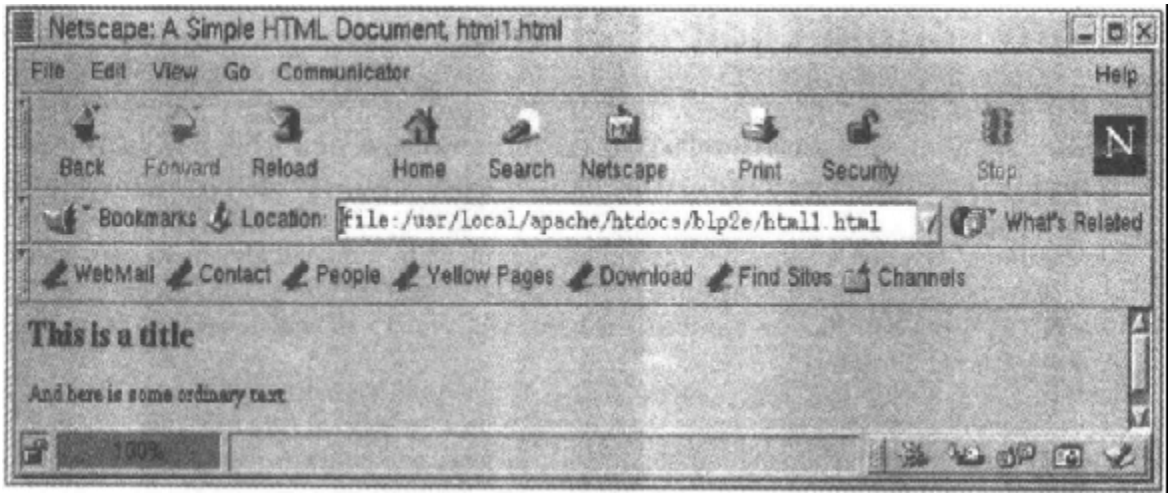


图 19-1

文档的其余部分都放在<HTML>和</HTML>标签之间，这两个标签的作用是把文档定义为HTML。接下来里面分为两个段落：一个文档头（head）和一个文档体（body）。在文档头里我们给出了一个文档标题（title），它与其他内容是分开显示的。在大多数浏览器上，它会被显示为整个窗口的窗口标题。

文档体部分有一个括在<H1>和</H1>标签之间的一级段落标题和一个括在<P>和</P>标签之间的文本段。你会发现许多HTML文档省略了</P>标签。这在以前是可以接受的，但现在随着HTML被定义得更加严格，总是应该加上</P>标签以明确地划分出各个段落来。纯文本内容按原样显示，并会根据浏览器窗口的宽度自动调整显示情况。如果你改变了窗口的尺寸，文本会随之做出相应的显示安排。

注意几乎使用的HTML标签都是成对出现的，并且标签要按照“最后打开最先关闭”的原则依次关闭。

置标标签是不分大小写的，所以<TITLE>、<Title>和<title>的作用都是一样的。但我个人认为把标签写为大写字母有助于把它们和文档的其余部分区别开。

19.4　深入学习HTML

见过HTML文档的一般格式之后，我们现在来学习HTML定义的常用标签。HTML第一个正式的格式标准是2.0，它相当初级，但得到了几乎所有的浏览器的广泛支持。在我们编写本书的时候，该标准的最新版本是4.0，与2.0版本相比它增加了许多许多的东西。但即使声称支持全部的4.0版本功能的浏览器也还是会漏下几个不太重要的小功能，好在HTML 4.0的核心功能在大多

数浏览器上都得到了很不错的支持。你尝试使用的功能越不同寻常，浏览器上缺少对它的支持或者支持不好的概率也就越大，使最终的显示效果偏离了你的预期。

许多浏览器接受对这个标准的扩展。如果给浏览器提供了一个文档，文档里面有浏览器不认识的标签，浏览器的一般做法是忽略它们。在一般情况下，如果你想让自己的文档在大多数浏览器上都有一个统一的显示效果，最好的办法就是坚持使用有关标准里定义的标签而不使用特定浏览器专用的扩展。

HTML是一个很重要的标准，因为它是与计算机硬件无关的。我们完全能够实现一个适用于一切图形显示器的HTML基本浏览器。甚至还有只显示文本的HTML浏览器，但很明显，它们所能显示的东西也会打一定的折扣。非图形化浏览器对视力不良的人士来说是很重要的，对那些上网速度很慢的人来说也有它的好处。

我们可以用HTML写出一个排好版的文档再把它发送给使用其他计算机平台的人们，而他们看到的文档和你写出来的文档是没有太大差异的。在过去，人们曾经为排版文档的跨平台浏览做过许多努力，但HTML无疑是到目前为止最成功的，而且你还不必为了书写这种文档去购买专门的软件。

HTML文档是由ISO 10646标准定义的字符构成的。而HTML文档中的字符或者构成将要被显示的文本，或者构成控制信息显示效果的标签。标签全都是以一个小于号字符（<）开始，以一个大于号字符（>）结束的；两者之间就是用来定义标签的字符，不分大小写。有的标签在“<”和“>”字符之间还带有其他属性（属性可能要分大小写字母），其一般格式为“attribute = "value"”。属性控制着标签的解释细节。

大多数标签都是封闭性的，即以一个起始标签开始，中间是一段文本，然后以一个对应的结束标签把文本段括起来。结束标签和起始标签的内容相同，但在标签名称前要加上一个“/”字符。就拿<H1>来说吧，它是一级段落标题的起始标签，与之对应的结束标签就是</H1>。标签允许嵌套，但结束标签必须按起始标签顺序的逆序出现。因此，“<H1> Heading <B> one </B> </H1>”是一个有效序列，而“<H1> Heading <B> one </H1> </B>”就不是了。

HTML文档以一个<HTML>标签开始，以一个</HTML>标签结束，中间是由文档头和文档体两部分组成的整个文档，所以这两个标签又被称为HTML包装符。规矩的HTML作者会在HTML文档的最开始加上一个文档类型定义，象下面这样：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
```

如果采纳了XHTML，就意味着一切HTML文档都必须强制性地加上一个DTD定义。因此，即使现在还没有严格要求必须加上它，最好还是从现在做起、从我做起。

HTML文档必须放在<HTML>和</HTML>两个标签之间。HTML标签有好几个属性，但只有那个按ISO-639标准规定的语言代码指定语言的属性比较常见一些。如果使用的是英语，这个属性就可以省略；如果使用的是其他语言，就最好加上它。属性必须放在标签里面，它们的写法是“attribute = "value"”。所以德语HTML文档应该以下面这一行开始：

```
<HTML lang = "de">
```

两个HTML包装标签之间，我们必须定义的第一个段落是文档头（或叫表头）。虽然在文档

头部分里有不少可干的事情，但人们一般只在这里用<TITLE>标签给出一个文档标题，其他什么事情也不做。

HTML页面的其他部分就是文档体了，它包含着你想在自己页面里显示的信息。它可以是完全空白的！

总之，一个最短而又合法的HTML文档就是下面这个样子的：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
</BODY>
</HTML>
```

可是这个文档既没有文档标题，也没有可显示的文字，根本没什么用处！

我们前面曾经提到过，几乎所有的HTML浏览器都接受不带DOCTYPE定义的HTML文档；而Web上省略了这个定义的文档实在太多了，所以这应该是一桩好事。但这桩好事也有让人悲哀的一面，那就是只有很少一部分Web文档能够达到HTML正式标准的要求。但在XHTML里这一切都要转变过来，因为它要求加上一个DTD部分使文档的格式符合有关规定。我们建议大家尽量严格地让自己的HTML符合有关规定，尽量加上DOCTYPE定义部分。对HTML文档的全貌有了大概的了解之后，我们来仔细研究一下它的置标标签。

19.4.1 HTML标签

以下是HTML里的一些常用主要标签。

1. 文档标题

所有HTML文档都必须有一个总的文档标题，文档标题必须在文档头部的<TITLE>和</TITLE>标签之间进行定义。文档标题通常会被显示在段落内容以外，面标签中间的文字往往会被存到书签或收藏夹里去，所以应该仔细斟酌用词。象“Home Page”这样没有特点的总标题在书签簿里很容易被人忽略掉。

2. 注释

和程序代码一样，HTML文档也可以从注释中获益。HTML注释行是下面这个样子的：

```
<!-- this is a comment -->
```

其中的“!”和“--”字符必须是这个样子和顺序。尽量不使用扩展多行的注释，这样可以增加点安全系数，因为有些早期的浏览器不能正确地处理跨行的注释。

3. 段落标题

段落标题一共有6级，从H1到H6；最顶层的段落标题是H1。标题文字要放在开始和结束标签之间。下面是一个二级段落标题的写法：

```
<H2> This is a   heading2 </H2>
```

HTML老标准里曾经增加了一个ALIGN属性，它已经在HTML 4.0严格版本里禁止使用了。

但它可以说是标段落题最具直观效果的属性了，所以我们还是把它介绍给大家吧。它可以带left（居左）、right（居右）、center（居中）或justify（对齐）等几种参数。请看一个居中的三级段落标题的写法：

```
<H3 ALIGN = "center"> Centered level 3 heading </H3>
```

4. 文本格式

HTML文档体部分的文字通常会被格式化为一个不间断的文本流，不考虑原始文件里的断行断句。这很有必要，因为文档作者不可能知道读者将会用多宽的窗口去浏览它。表19-1里的标签提供了额外的格式控制。

表 19-1

标 签	说 明
 	这个标签没有对应的结束标签，它的作用是使文本流从下一行开始
<P>	这个标签的作用是开始一个新的段落，并且经常会先插入一个空行。漏掉它结束标签的做法是错误的。注意<P><P>通常并不会插入两个空行。如果你想在一个HTML文档里留出几个空行，可以使用<PRE>标签（后面有说明）
<HR>	这个标签的作用是在页面上画出一条水平直线

5. 文本样式

浏览时看到的文本外观是由一组样式标签控制的。比较常用的有：

<ADDRESS> This is my address </ADDRESS>把标签间文字排版为普通信件地址的写法。

<B>This text is bold </B>把标签间文字显示为一种黑体。

<BLOCKQUOTE> This text is a quote </BLOCKQUOTE>把标签间文字显示为与文档其他内容不一样的形式，多采用缩进显示。一般用在引经据典的时候。

<CITE> A citation </CITE>把引用性文字标识出来。

<CODE> Some source code </CODE>把标签间文字显示为源代码程序清单的格式。

<EM> Emphasized text </EM>设置强调性文字，但还是明确地指明黑体或斜体比较好一些。

<I>This text is in italics </I>把标签间文字设置为一种斜体。

<KBD> text type by a user </KBD>表示这是用户敲入的文字，通常显示为一种等宽字体。

<PRE> Pre_formatted text </PRE>不让浏览器对标签间文字进行排版，使它们的显示效果就是HTML源文件里的样子。但因为事先无法知道浏览器窗口的高度和宽度，所以往往会在浏览时给某些用户带来相当大的困难，尽量少用这个标签。

<STRONG> This text is strong </STRONG>把标签间文字设置为加强强调效果。

_{This is subscript}把标签间文字设置为下标。

^{and this is superscript}把标签间文字设置为上标。

<TT> Typewriter spacing </TT>把标签间文字设置为一种等宽字体，就像是用打字机打出来的那样。

大多数比较普通的排版要求一般用<B>、<I>、<PRE>、<CODE>等就足够了，而大多数浏览器都能保证这几个标签显示效果的一致性。

动手试试：文本排版

学了就得用。看了这么多的标签之后，我们来看看它们的实际使用情况。其他浏览器显示出来的画面效果可能会和我们这里的不一样。

下面这个HTML文档演示了我们刚才介绍的几个文本排版标签的使用方法：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html2.html</TITLE>
</HEAD>
<BODY>
<H2>This is file html2.html</H2>
<P>
And here is some ordinary text.
<!- And here we have inserted a comment ->
<!- which takes two lines to type in ->
</P>
<H4>Heading level four</H4>
looks like the above.
<H5 ALIGN="center">This is a centered level five</H5>
heading, showing the extra ALIGN tag.
<P>
we can do quite a lot of typing, then put a break in here,<BR>so the text flow is
broken up.
</P><P>
If we didn't use the BR tag then text would just keep on flowing across the page. If
the user changes the width of the browser window then the text adjusts accordingly.
<HR>
Used sparingly a horizontal rule is a good separator.
</P><P>
Let us try some other changes. Here is some <B>bold</B> text, some <I>Italicized</I>
text and some <EM>emphasized</EM> text. A fixed width font is selected with TT like
<TT>this section</TT> of text.
</P><P>
If we would like to
include some code we can make it look like this:<BR>
<CODE>
#include &lt;stdio.h&gt;<BR>
int main() {<BR>

printf("Hello World\n");<BR>
}<BR>
</CODE>
</P><P>
Perhaps an easier way is to use the PRE
tag, like this:<BR>
<PRE>
#include &lt;stdio.h&gt;

int main() {
printf("Hello World\n");
}
</PRE>
</P><P>
Here is an example of an address:<BR>
<ADDRESS>
Mr. Postman Pat<BR>
7 Posty Lane,<BR>
Greendale<BR>
Lancashire<BR>
Great Britain<BR>
</ADDRESS>
</P><P>
This is a block quotation from<BR>
```

```
<CITE>Macbeth, by William Shakespeare</CITE>
<BLOCKQUOTE>
Is this a dagger which I see before me, The handle toward my hand? Come,
let me clutch thee:<BR>
I have thee not, and yet I see thee still.
</BLOCKQUOTE>
<P><P>
Which I consider an outstandingly good play.
</P>
</BODY>
</HTML>
```

图19-2是用网景公司的Navigator浏览器浏览它时的窗口画面。

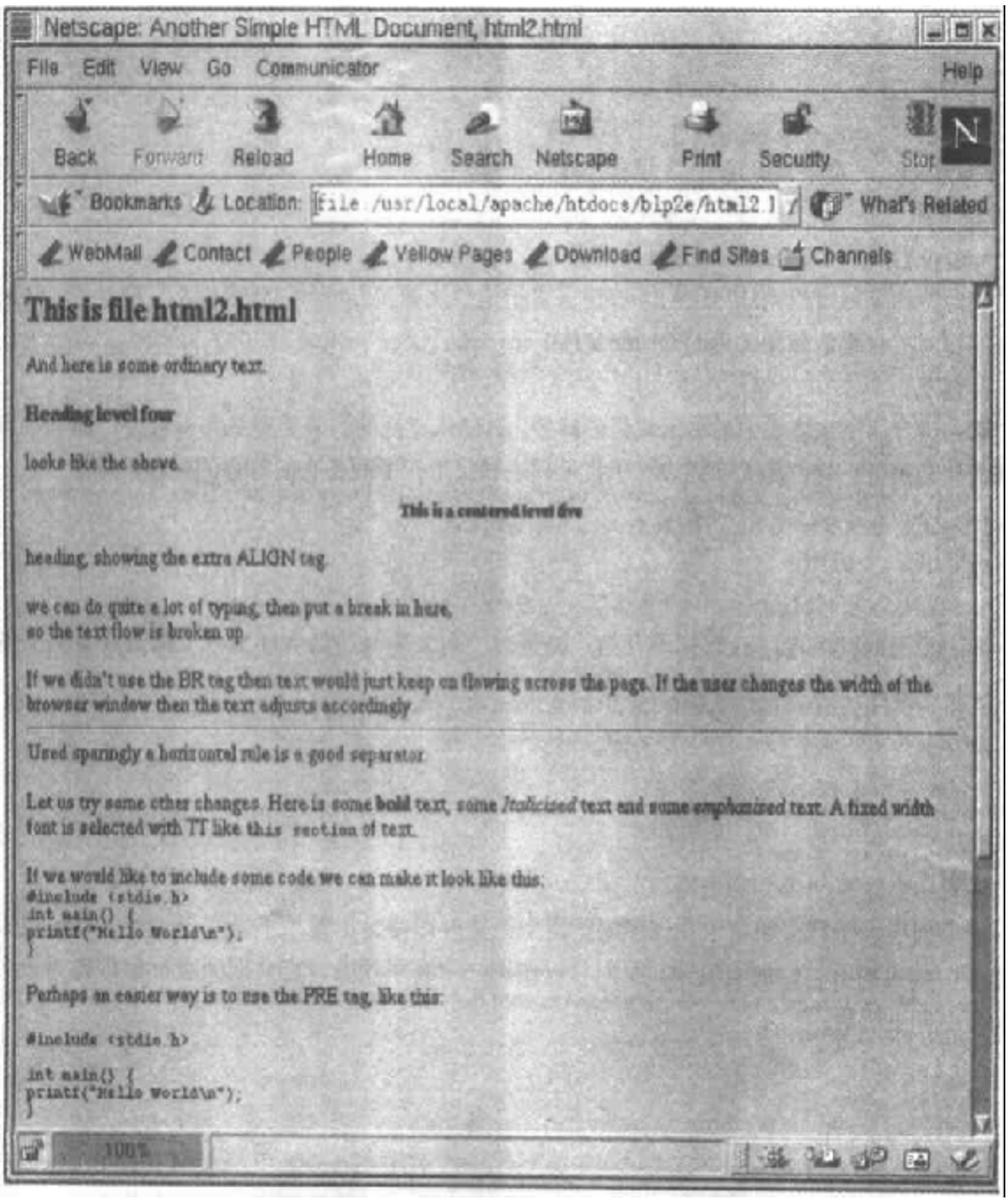


图 19-2

操作注释：

这只是前一个例子的扩展而已。注意我们不能在代码段里使用“<”或“>”字符，因为它们是为突出标签而保留的。我们要用一种特殊语法来定义这些字符。

6. 特殊字符

HTML标准里定义的特殊字符有很多，下面是一些比较常用的：

&lt	<	大于号
&gt	>	小于号
&amp	&	“and”符号
&quot	"	双引号
&copy	©	版权符号
&aaccute	á	a字母加重音
&aelig	æ	字母a、e靠在一起
&ccedil	ç	带尾c字母。
&uuml	ü	两点u字母

其他特殊字符可以用HTML的字符编码来指定，比如“&#064”代表“&”符号等。HTML编码字符集的完整清单请参考RFC1866文件。

7. 列表

有一组专门用来提供列表显示效果的标签。HTML支持的列表类型总共有五种，但其中有三种彼此比较接近，它们是无序列表、子目录列表和菜单，下面是它们各自的标签：

- <UL> </UL>
- <DIR> </DIR>
- <MENU> </MENU>

列表里面的列表项以<LI>标签开始。浏览时，列表项开始处会加上一个圆点或者类似的符号。下面就是一个简单的无序列表示例：

```
<UL>
<LI>The first item
<LI>The second item
<LI>The last item
</UL>
```

注意结束标签</LI>可以省略。

无序列表、子目录和菜单在许多浏览器上浏览时即使显示效果不是完全一样，肯定也差不了多少。如果想给它们加上编号，我们可以使用一个有序列表，它会把圆点标记替换为一个顺序编号，如下所示：

```
<OL>
<LI>The first item is 1
<LI>The second item is 2
<LI>The last item is 3
</OL>
```

列表的第五种类型是定义列表。它最适合显示字典形式的数据项，带一个键字和一些解释

性文字。因为一个列表项现在是由两个元素组成的了，所以需要使用两组标签：术语定义标签<DT>和术语定义解释标签<DD>，与之对应的结束标签</DT>和</DD>可以省略。但整个定义列表的结束标签</DL>是少不了的。定义列表的用法如下所示：

```
<DL>
<DT>Bush
<DD>A normal growing pattern for most Fuchsias<DT>Standard<DD>The main part of the
plant is raised on a single stem<DT>Basket
<DD>A group of fuchsias that look good in hanging baskets.
</DL>
```

动手试试：列表

下面是一个包含有列表的例子。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html3.html</TITLE>
</HEAD>
<BODY>
<UL>
<LI>This is entry number 1
<LI>This is entry number 2
<LI>This is entry number 3
<LI>This is entry number 4
</UL>
<OL>
<LI>This is entry number 1
<LI>This is entry number 2
<LI>This is entry number 3
<LI>This is entry number 4
</OL>
<H4>Fuchsia types</H4>
<DL>
<DT>Bush</DT>
<DD>A normal growing pattern for most Fuchsias.</DD>
<DT>Standard</DT>
<DD>The main part of the plant is raised on a single stem.</DD>
<DT>Basket</DT>
<DD>A group of fuchsias that look good in hanging baskets.</DD>
</DL>
</BODY>
</HTML>
```

它的浏览器显示画面如图19-3所示。

19.4.2 图像

文本的格式编排已经很不错了，但要是能在浏览画面里加上一些图片就更好了。这还真不难做到：使用IMG标签就行了。这个标签有几个控制图像显示效果的属性。

带属性的完整标签如下所示：

```
<IMG SRC="image URL" ALT="alternate text" ALIGN="top"|"middle"|"bottom"|"right"|"left"
WIDTH=n HEIGHT=n ISMAP>
```

用“|”字符隔开的属性是多选一的，即ALIGN只能取其中的一个值。SRC和ALT属性是必须给出的。

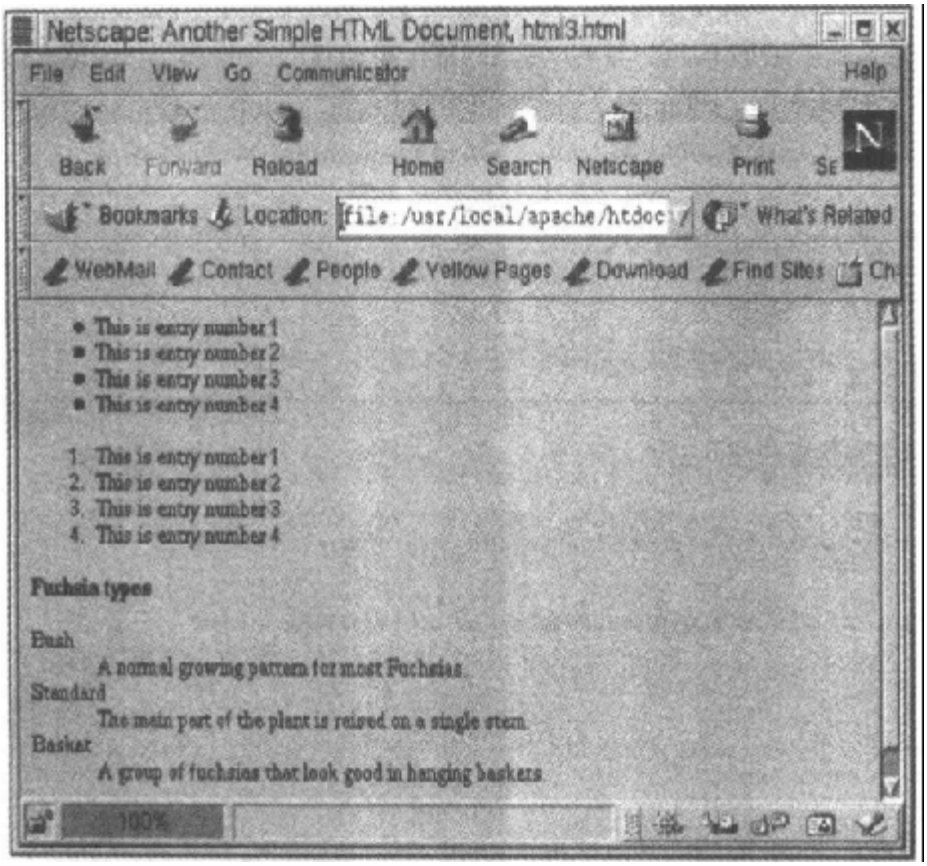


图 19-3

SRC给出了图像源文件的名称，它通常是GIF或JPEG格式，但有时PNG（Portable Network Graphic，可移植网络图形）格式的文件也是可以接受的。我们一般应该使用这些编码方法给出最小长度的文件来，具体操作要取决于图像的类型。哪种格式能够在最小尺寸时给出最佳的结果图像需要多做实验才能确定。图像文件的名字可以是一个简单的文件名，也可以是一个指向其他地点的URL。一般来说，JPEG适合显示照片，GIF适合显示其他图像。

ALT给出的是一段替换性文字，如果浏览器不能显示图像（比如用户因自己上网速度比较慢，等待图像下载的时间比较长而关闭图像加载功能时）就会把这段文字显示出来。ALT部分的文字最好弄得吸引人点。

现在已经贬值了的ALIGN属性取值为“|”字符隔开的字符串中的一个，它的作用是设定后续文本与这幅图像的摆放方式。

WIDTH和HEIGHT告诉浏览器要给图像留多大的地方，这两个值以像素为单位。如果浏览器要在文本加载完成之后才开始加载图像（这是上网速度比较慢的用户最经常使用的设置），这两个值就可以让浏览器在取回图像之前先把文本摆放好。

ISMAP属性表示这是一个可点击图片，我们马上就要介绍这个概念了。

动手试试：添加图像

下面的HTML文档在浏览时会把两幅啤酒图像插入到其中，这是我最爱喝的两个牌子。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">

<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html4.html</TITLE>
</HEAD>
<BODY>
<H3>Black Sheep Ale</H3>
<IMG SRC="blac_s.jpg" ALT="Black Sheep Ale" ALIGN="left" HEIGHT=150
WIDTH=100>
<P>
Here is a picture of a bottle of Black Sheep Ale.
</P><P>
The culmination of five generations of brewing expertise.<P>
Brewed at Paul Theakston's Black Sheep Brewery in Masham, North
Yorkshire.<P>
And nowhere else.
</P>
<PRE>
</PRE>
<HR>
<H3>Spitfire</H3>
<IMG SRC="spit_s.jpg" ALT="Spitfire Bitter" ALIGN="right" HEIGHT=150
WIDTH=100>
<P>
Here is a picture of Spitfire Bitter.
</P><P>
This is a bottle conditioned bitter brewed at the Sheperd Neame brewery in
Kent.
</P><P>
Established in 1698 the bottle proclaims that it is Britain's oldest brewery.
</P>

</BODY>
</HTML>
```

我们将看到如图19-4所示的浏览器画面。

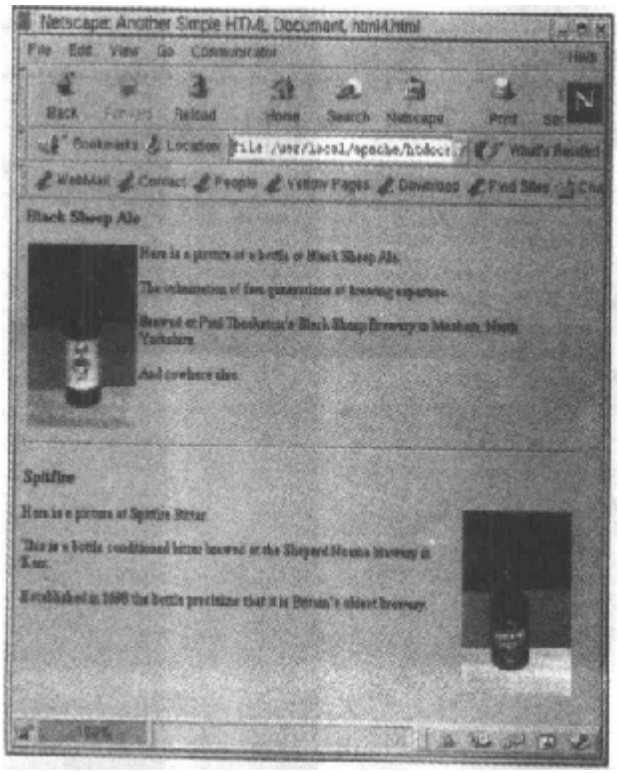


图 19-4

操作注释：

浏览器分别加载文本和图像，然后把它们一起显示到屏幕上。我们例子里用的是JPEG格式的图像。请注意ALIGN属性是如何影响图像后面的文本摆放方式的。

19.4.3 表格

HTML定义里原来是没有表格的，但它们现在已经成为数据排版的一个重要手段。除了可以用它们显示常规的表格以外，我们还经常会用不带表格线的表格控制图片和文本的摆放方式。

我们只介绍几个基本的表格属性。如果读者需要使用复杂的或嵌套式表格，请自行研究HTML 4.0标准或Wrox出版社出版的“Instant HTML：Programmer's reference”（《HTML速成：程序员参考手册》）一书，国际书号是ISBN 1-861001-56-8。表格是用行和列来描述的。表格中的数据项可以占用不止一行或不止一列的地方。

表格需要放在<TABLE>和</TABLE>标签之间。另外还有一些只能用在这两个标签中间的标签，它们可以帮助我们更好地设置表格的显示效果。这些标签包括：

- <CAPTION>和</CAPTION> 定义表格的标题。
- <TR>和</TR> 标签之间的文本构成表格中的一行。
- <TH>和</TH> 标签之间的文本构成表格的第一行或第一列。
- <TD>和</TD> 标签之间的文本构成表格中的一个表格项。

上述标签都只在表格行内有效。

TABLE标签还有几个属性，其中最重要的是CELLPADDING和CELLSPACING，它们分别控制着表格项的宽度和表格文字与表格线的间距。

动手试试：一个表格

下面的HTML文档会在浏览器里显示为一个简单的表格。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html5.html</TITLE>
</HEAD>
<BODY>
<TABLE CELLPADDING=5>
<CAPTION>A simple table</CAPTION>
<TR>
        <TD>flats</TD>
        <TD>houses</TD>
        <TD>shops</TD>
        <TD>factories</TD>
</TR>
<TR>
        <TD>small</TD>
        <TD>larger</TD>
        <TD>may be very big</TD>
        <TD>usually very large</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

我们将看到如图19-5所示的浏览器画面。

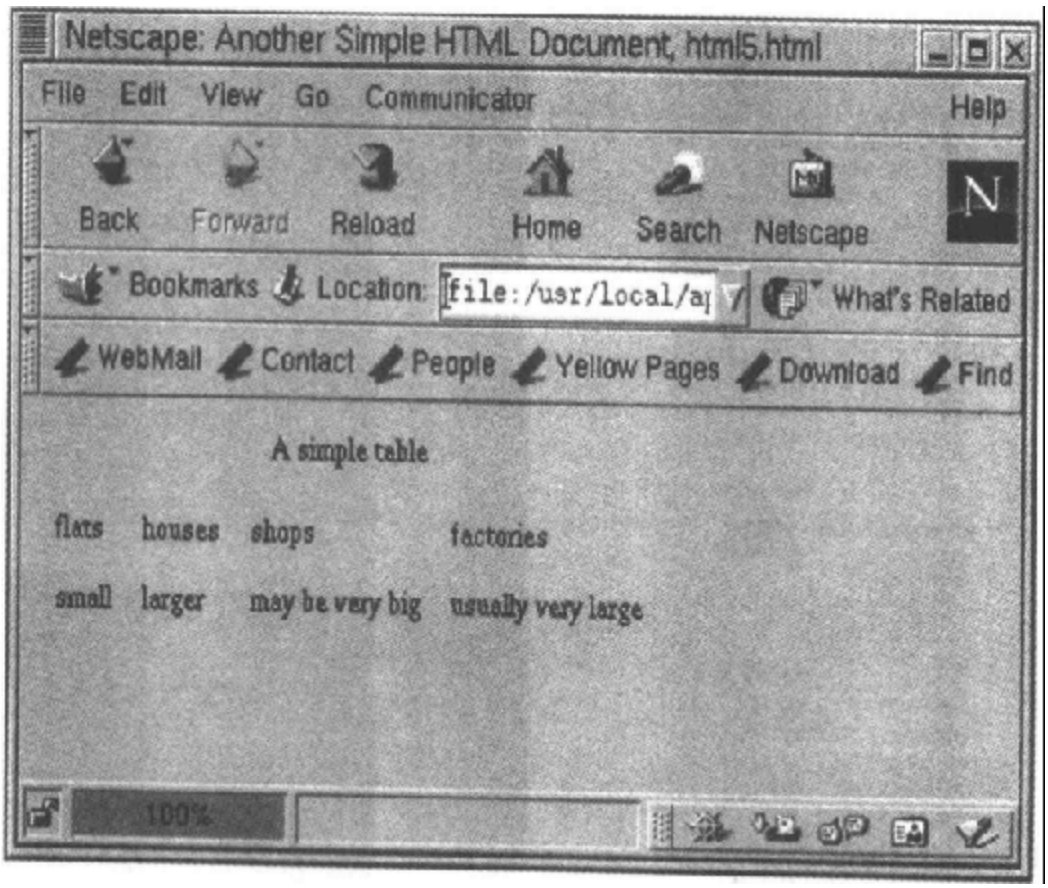


图 19-5

操作注释：

表格里的每一行都要以<TR>标签开始，以</TR>标签结束。表格里的列是用<TD>和与之对应的</TD>结束标签隔开的。许多文档省略了</TR>和</TD>标签，由浏览器根据上下文来决定表格行或表格列的结束位置。我们希望读者不要少加了这两个结束标签。

浏览器会按照表格中的定义生成一个行、列数正确的表格。它还会按要求调整表格项的宽度，使之适合显示其中的内容。

动手试试：另外一个表格

下面的HTML文档将生成一个比较复杂的表格。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html6.html</TITLE>
</HEAD>
<BODY>
<TABLE BORDER=2 CELLPADDING=15>
```

```
<CAPTION>HP printers in use</CAPTION>
<TR>
        <TH></TH>
        <TH COLSPAN=3>Inkjet</TH>
        <TH COLSPAN=2>Laser</TH>
</TR>
<TR>
<TH></TH>
        <TH>Original</TH>
        <TH>500</TH>
        <TH>600</TH>
        <TH>2p</TH>
        <TH>4m</TH>
</TR>
<TR>
        <TH>Software</TH>
        <TD>0</TD>
        <TD>2</TD>
        <TD>1</TD>
        <TD>0</TD>
        <TD>2</TD>
</TR>
<TR>
        <TH>Hardware</TH>
        <TD>1</TD>
        <TD>1</TD>
        <TD>0</TD>
        <TD>1</TD>
        <TD>0</TD>
</TR>
<TR>
        <TH>Sales</TH>
        <TD>0</TD>
        <TD>0</TD>
        <TD>0</TD>
        <TD>1</TD>
        <TD>1</TD>
</TR> <TR>
        <TH>Drawing</TH>
        <BR>Office
        <TD>0</TD>
        <TD>2</TD>
        <TD>0</TD>
        <TD>0</TD>
        <TD>0</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

我们将看到如图19-6所示的浏览器画面。

操作注释：

我们给<TABLE>标签加上了设定表格边框宽度的BORDER属性和设定表格文字与表格线间距的CELLPADDING属性。

我们用空白的<TH>标签跳过我们不想在其中输入内容的表格列。COLSPAN属性的作用是让“Inkjet”和“Laser”标题能够占据表格几个列的地盘。另外请注意
标签让“Drawing Office”占据了两行地盘。

浏览器在显示表格时缺省使用固定宽度的列，但这个例子里的浏览器使用的是可变宽度的列。

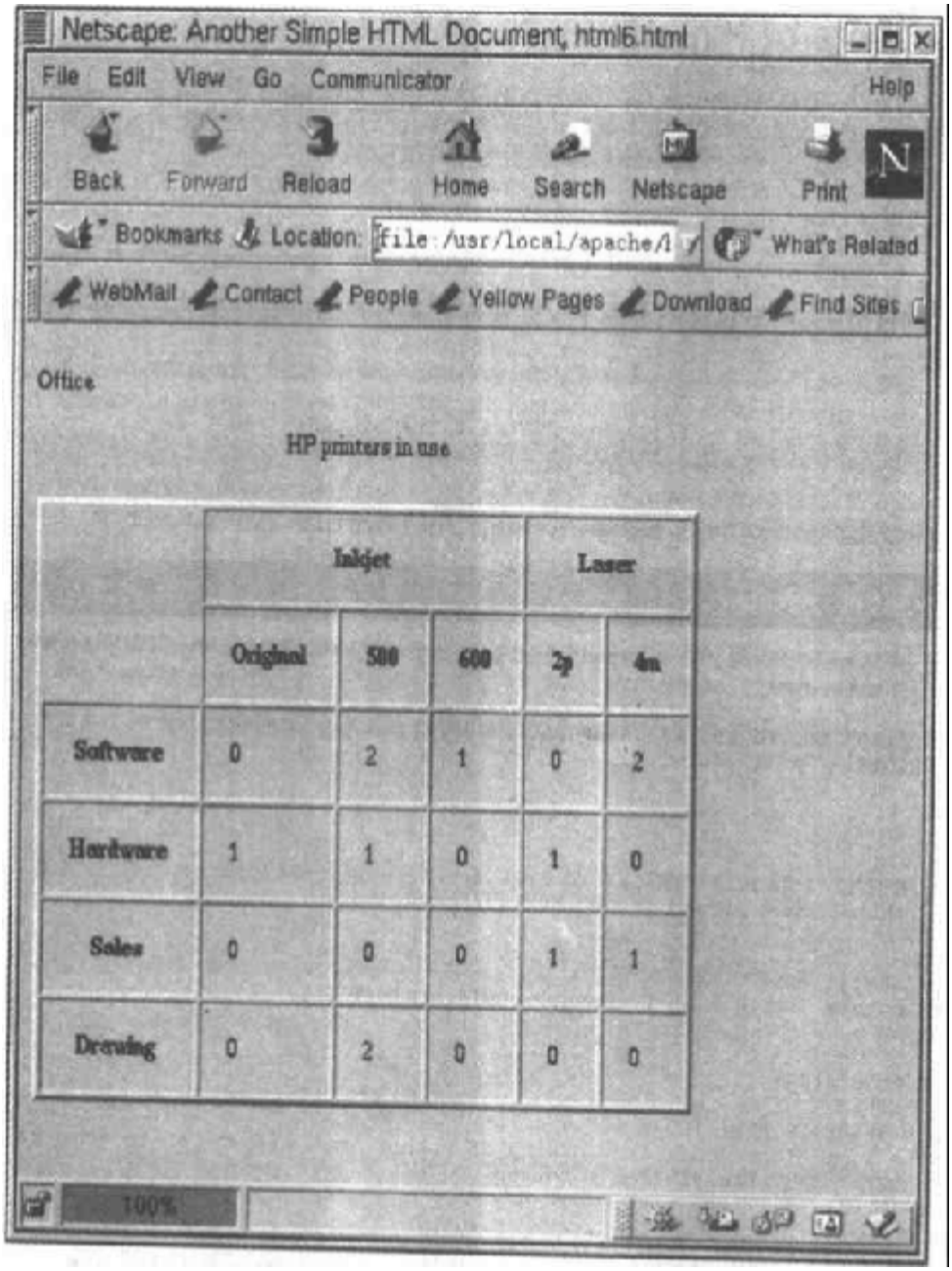


图 19-6

19.4.4 锚点或超链接

World Wide Web文档的一个重要特性就是它们之间的链接，而HTML文档之所以能够创造出World Wide Web这样引人入胜的事物也正是因为有了这个功能。

锚点标签其实是很简单的，如下所示：

```
<A NAME = "where I am now" HREF = "URL to go to" > some text </A>
```

NAME属性提供一个在文档里命名当前位置的办法。它是可以省略的。

HREF是一个超文本链接，它提供了该链接被激活时将要转去的URL网址。如果HREF里出

现有“#”字符，则它后面的所有字符指的就是某个文档中某个锚点的名字，而它前面的所有字符将构成指向该文档的URL。

起始锚点<A . . . >和结束锚点</A>之间的所有文字都会被突出显示出来。当用户选中这个突出显示的文字时，浏览器就会转到HREF属性指定的URL处。

动手试试：锚点

下面两个HTML文档里包含的锚点不仅能使你在其各自的内部转来转去，还可以使你在它们之间转来转去。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html7.html</TITLE>
</HEAD>
<BODY>
<A NAME="top"></A>
<P>
Here is a simple document. It contains an anchor that allows you to jump to
<A HREF="html8.html"> html8.html</A> if you click on the highlighted text.
</P><P>
You can also jump to the <A NAME="bottom" HREF="html8.html#bottom"> bottom
of html8.html</A> if you wish.
</P>
</BODY>
</HTML>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">

<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html8.html</TITLE>
</HEAD>
<BODY>
<A NAME="top"></A>
<P>
<H2>This is html8.html</H2>
Here is a simple document. It contains an anchor that allows you to jump to
the <A HREF="html7.html#bottom"> bottom
of html7.html</A> if you wish.
<P> We need to insert some text.</P>
<P> Quite a lot of text.</P>
<P> If we didn't, how could you tell which was <B>top</B> and which was
<B>bottom</B> of this document?</P>
<P> You can also <A HREF="#top"> jump to the top</A> of this document if you
wish</P>
<P> It might well all appear all on the same page.</P>
<P> Soon we will have enough text.</P>
<P> To prevent this fitting </P>
<P> on</P>
<P> a</P>
<P> single</P>
<P> page!</P>
<P> This is the bottom, but you can jump to the <A NAME="bottom"
HREF="#top">top</A> if you want, or back to <A HREF="html7.html">
html7.html</A> if you prefer!</P>
</BODY>
</HTML>
```

图19-7是html8.html的浏览器画面的一部分，链接都呈突出显示效果。

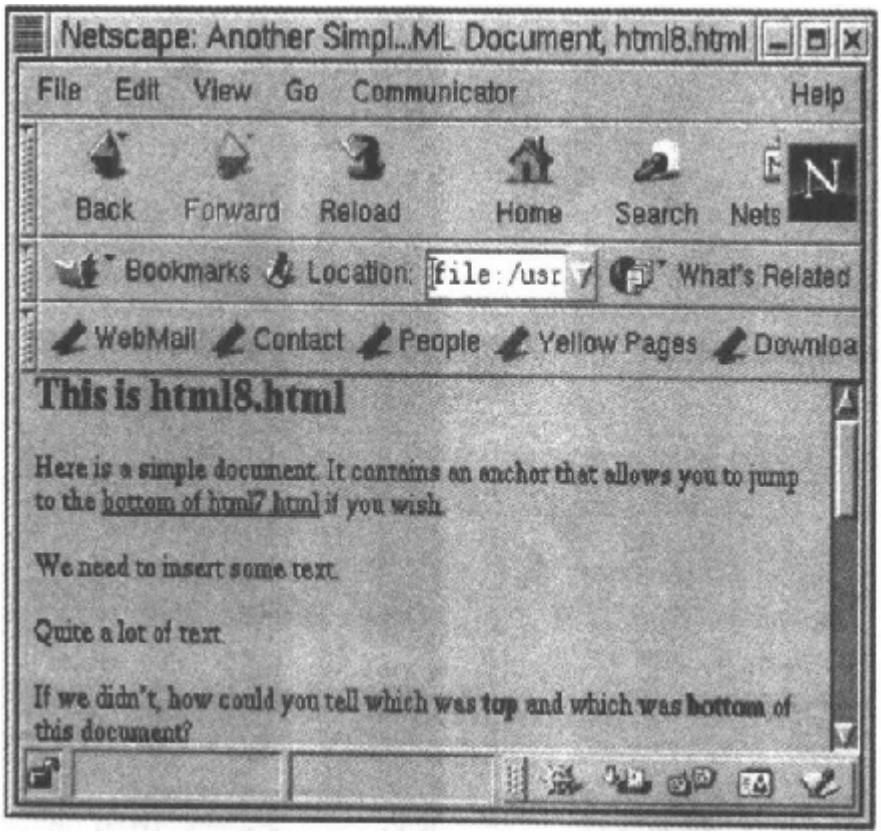


图 19-7

操作注释：

这两个HTML文档演示了锚点标签的用法，我们可以学习到如何通过<A NAME = . . .>标签来标记文档中的某个地点，而其他文档又是如何通过<A HREF = . . .>标签引用它的。注意这里使用的所有标签使用的都是相对URL，没有给出计算机的机器名。

我们还可以看到如何运用这两个标签设定一个能够到达另一个文档中某个特定位置的跳转链接。

19.4.5 给图像加上锚点

给图像加上锚点的做法是很常见的，因为点击一个图像要比点击于巴巴的文字有意思的多。把锚点标签<A>和图像标签<IMG>联合使用就可以实现这一效果，如下所示：

动手试试：图像和锚点

请看下面这个HTML文档：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html9 html</TITLE>
</HEAD>
<BODY>
```

```
<A NAME="top"></A>
<H2>This is html9.html</H2>
Here is a simple document. It contains an anchor that allows you to jump by
clicking on a picture. Actually the HREF file "beer.html" doesn't exist,
but the browser does not discover this unless you click to take the hyper link.
<P> You can also <A HREF="beer.html"><IMG SRC "blue_s.jpg" ALIGN="MIDDLE"
ALT="jump to beer.html"></A> use images as
clickable items if you wish.</P>

<P> The end.</P>
</BODY>
</HTML>
```

我们将看到如图19-8所示的浏览器画面

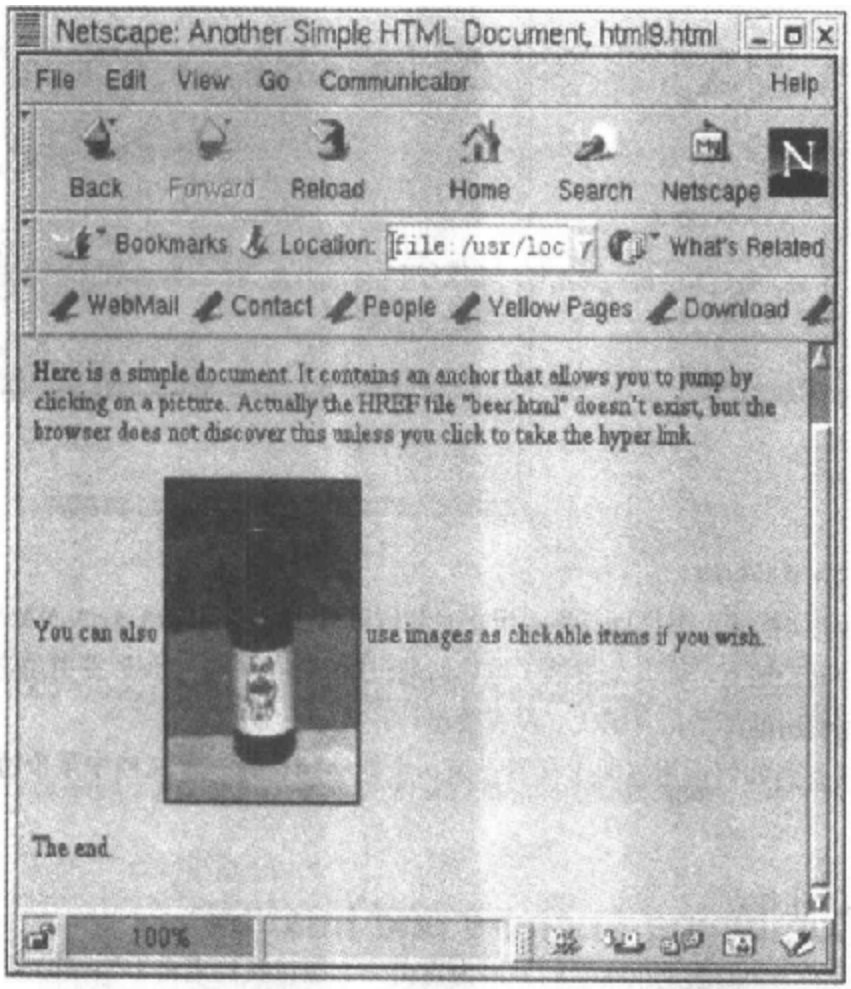


图 19-8

操作注释：

<IMG>标签的作用是在锚点里插入一幅图像。注意ALT属性的用法，即使浏览器因为某些原因不能显示图像，通过这个办法也可以让别人知道链接是转往什么地方去的，并且也能够转到那里去。如果图像被选中，浏览器就会尝试加载该链接设定的URL——beer.html文件。

这里使用的图像相对于文字来说尺寸有点大了。在正常情况下，小一点的图像看起来更舒

服一些，我们这里是为了举例子才这样做的。

19.4.6 非HTML的URL地址

到目前为止，我们的所有链接指向的都是同一机器上的其他HTML页面，但我们实际上是可以使用其他形式的URL地址的。最常见的用法是象下面这样让一个锚点指向一个图像文件：

```
<A HREF = "picture.jpg" >
```

点击这个链接时将返回一个JPEG格式的图像。

我们可以把响应WWW浏览器请求的服务器配置成这样的状态：让它能够根据文件的扩展名把文件映射到将在HTTP协议下传输的正确的MIME文件类型。

具体机制取决于文档将从什么地方被加载过来。如果文件是从一个服务器那里取来的，其MIME类型及文件的子类型就将由服务器来负责决定，服务器会在实际开始传输数据之前先使用HTTP协议把这些类型信息发送给浏览器。而浏览器将负责决定如何对该MIME类型及其子类型进行处理。

如果加载的是一个本地文件，就必须由浏览器来决定MIME类型及其子类型，如何处理这些类型的工作也是由浏览器决定的。

如何把文件扩展名映射到MIME类型上的完整讨论超出了本章的讨论范围；文件扩展名与MIME类型的映射细节请读者参考自己服务器和浏览器的使用手册。这些内容一般可以在名为mime-types和mailcap的文件里查到。做为一个例子，我们下面给出一个Netscape浏览器的.mailcap文件，它告诉我们声音文件将用“play”命令来处理。如下所示：

```
audio/* ; play %s
```

HTML页面里经常会用到的其他URL包括：

ftp://ftp.site.name/pub/filename：如果被选中，浏览器将被请求开始一个FTP会话去检索那个文件。

mailto:name@machine.com：如果被选中，浏览器将被请求开始一个e-mail会话，这里给出的就是收信地址。

news:comp.os.linux.announce：如果被选中，浏览器将被请求开始一个新闻会话，这里给出的就是新闻组地址。

19.4.7 链接到其他站点

如果所有的计算机都只能提供本地HTML页面，WWW网也就不会是现在这个样子的资源宝库了。WWW网的强大之处和诱人之处就在于页面可以链接到位于不同机器上的其他资源。这是用URL地址的绝对形式来实现的，其格式为http：//machine.name/file.html。

动手试试：链接到其他站点

下面这个HTML文档使用了一些不常用的URL地址和一些到其他站点的链接：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
```

```
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Another Simple HTML Document, html10.html</TITLE>
</HEAD>
<BODY>
<H2>This is html10.html</H2>
This shows some less common URLs and links to remote machines.
<P> If you want to find out about some other WROX books we suggest you visit
their <A HREF="http://www.wrox.com/">home page</A>.</P>

<P> To send feedback on Wrox books you can email them at
their <A HREF="mailto:feedback@wrox.demon.co.uk">feedback address.</A></P>

<P> For more information about the World Wide Web, visit the <A
HREF="http://www.w3.org/">World Wide Web Consortium</A>. You will find many
specifications, draft specifications, lists of http servers, and lots of other helpful
files.</P>

<P> If you are just getting started with Linux, then you should subscribe to
the newsgroups</P>

<A HREF="news:comp.os.linux.announce"> comp.os.linux.announce</A>, and
<A HREF="news:comp.os.linux.answers"> comp.os.linux.answers</A>.
<P> Some people are not sure how Linux is</P>

<A HREF="english.au">pronounced</A>. Well now you know!
<P>
<HR>
</P>
<P>
<IMG SRC="pblinux.gif" ALT="Powered by Linux" ALIGN="middle">
This server is running on a Linux box. Be proud, be powered by Linux!
<HR>
<CITE>Many thanks to <A HREF="mailto:Alan.Cox@linux.org">Alan Cox</A> for
permission to use this graphic </CITE>
</P> <P>
This page is &#169 Copyright Wrox Press.
</P>
</BODY>
</HTML>
```

我们将看到如图19-9所示的浏览器画面。

操作注释：

这个例子里显示的页面里有一些绝对URL地址和一些较少使用的地址。注意我们把图像的背景颜色设置为透明的（这是GIF文件功能之一，大多数浏览器都支持），这样我们看到的图像就没有边框。Linux图像来自http：//www.linux.org.uk/；感谢Alan.Cox@linux.org允许我们使用它。

与图像有关的几个小问题

在对图像进行处理时还有一些其他的小技巧。一个比较流行的传统做法是给页面加上一个背景。在<BODY>标签里加上BACKGROUND属性可以做到这一点。你应该使自己的背景图像尽可能的小，并且千万不要让它喧宾夺主，把你的主要内容给比下去。设置背景的标签如下所示：

```
<BODY BACKGROUND = "background.gif">
```

你可以把GIF文件中的某个颜色设置为透明的。这使背景在屏幕上看起来更透亮，比起页面上一个方形的图像区来效果要好得多了。

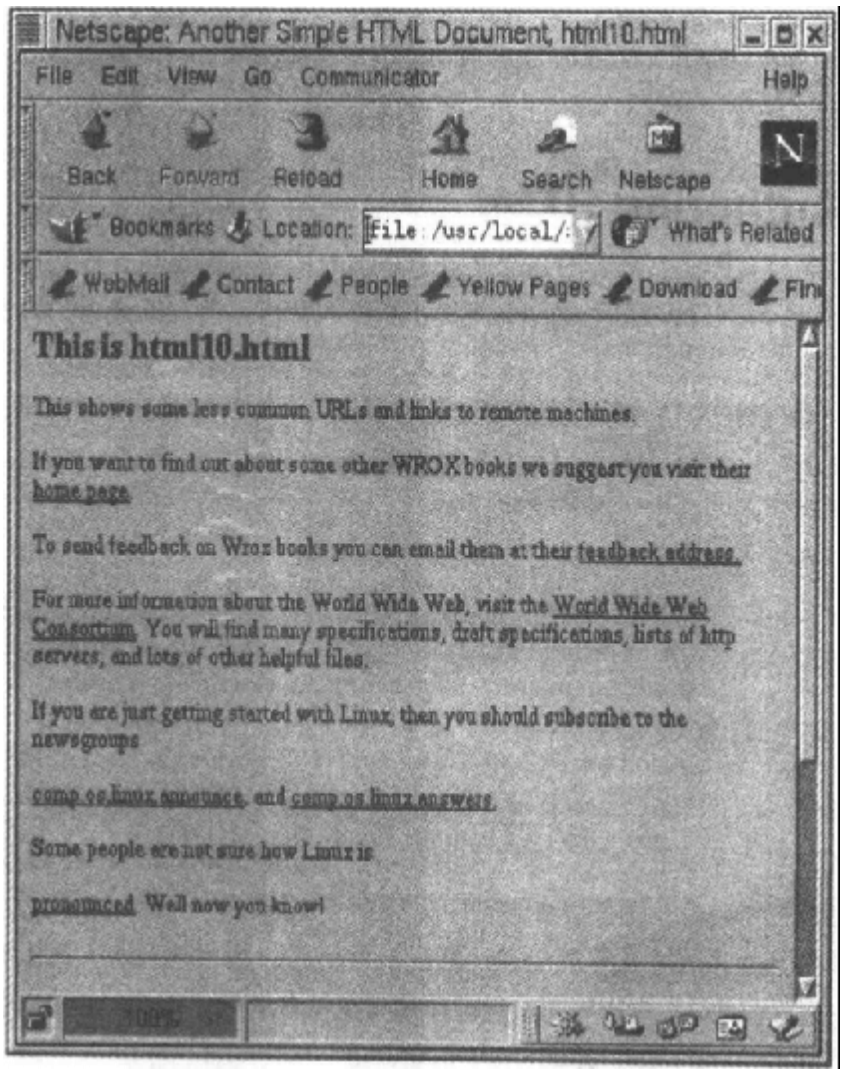


图 19-9

GIF文件的另外一个功能是你可以把几幅图像保存在同一个GIF文件里。部分浏览器能够处理这样的情况，按顺序依次显示GIF文件里的每一幅图像，形成一个简单的动画效果。

19.5 编写HTML文件

我们一直以来都是采用简单的键盘输入方法来编写HTML文档的。当然，当掌握了足够的HTML知识以后，直接输入它们也确实没有多费事。本章的HTML文档都是我们用emacs编辑器直接输入的。

如果你开始编写大量的HTML，就可能想了解一些其他的书写方法。主要的方法有：

- 用一个文本编辑器直接输入它。
- 用一个转换器程序把它从一个现有的格式转换过来。有许多种转换器，有些还能把HTML转换为另外一种格式，比如PostScript等。我们特别感兴趣的格式是Linuxdoc-sgml。类似于HTML，它也是一种符合SGML定义的结构化文档，但它还可以被转换为包括HTML在内的许多其他种格式。如果你只想写一次文档，但又需要把它转换为许多不同的格式，就

可以考虑使用Linuxdoc-sgml。

- 使用上下文敏感的编辑器。有些编辑器“知道”HTML的规定，能够帮助你选择正确的标签来安排HTML文档的结构。你也可以使用emacs编辑器的“HTML helper”帮助模式。
- 直接使用一个能够生成HTML的WYSIWYG编辑器。

使用WYSIWYG生成器可能会带来一些问题。提出HTML的目的是为了适应不同计算机平台上各种各样的浏览器，因此它定义的是文档普遍意义上的格式而不是字体字号之类的细节。这样，虽然你在自己的WYSIWYG生成器上确实是所见即所得，但在其他人的机器上可能就是所得非所见了。

编写好自己的HTML文档之后，还需要做两件事情。首先，通过一个HTML检查器程序来运行它。这方面确实有几个很不错的程序，其中包括“html -check”和“weblint”。第二，尽可能多地在你能收集到的浏览器里（和计算机平台上）对它进行测试，好让最终用户看到的每一个页面都是合理的画面。

除表单（form）和可点击图片以外，HTML的主要内容基本上都在这里了。我们将在下一章里遇到表单，而可点击图片方面的内容我们很快就会在本章后面的部分里开始讨论。

19.6 HTML页面服务

在前面的学习过程里，我们一直是编写HTML页面，再把它们做为文件加载到一个浏览器里去查看。如果你只是一个人，用这种办法来浏览一些文档倒也未尝不可；但如果你想让许许多多的人看到自己的工作成果，就必须给他们每人发送一份文档的拷贝。

更好的办法是把文档放在计算机里，计算机再联上网，然后等到有人请求查阅文档的时候再通过网络把它发送过去——WWW网的起因和基础也正在于此。用户不必事先提出请求就能开始对文档进行浏览（“冲浪”这个词是不是更形象？）。

我们可以这样做：在一台计算机上安装运行一个服务器程序，所有的文档也都保存在那里，使客户程序能够穿过网络来检索这些文档。

不要认为只有在因特网上这才是有用的。公司等机构组织完全可以利用它来传播企业的内部文件，这比记录、保存、修改、发送一大堆“废纸”要简单而且有效的多了。只要把文档的一份拷贝放到了内部LAN局域网中某个服务器上，就可以从连接在这个LAN里的任何一台工作站上查看到它。再说了，Linux和许多HTML服务器程序都是自由免费的，所以这绝对是一个低成本的解决方案。

19.6.1 网络中的HTML文档

图19-10给出了客户请求浏览HTML文档（及图像等其他相关文件）的流程，这些文档都存放在一台服务器上。

客户计算机上浏览器程序提出的请求经网络到达服务器。服务器的HTTP守护进程（UNIX系统上的服务器程序经常被称为“守护进程”）对这个请求进行处理，从自己的文档库里读出文档并把它们发送回客户那里，文档在发送前要经过MIME编码处理。所有数据都使用HTTP协议来传递，这个协议运行在底层的网络协议之上。

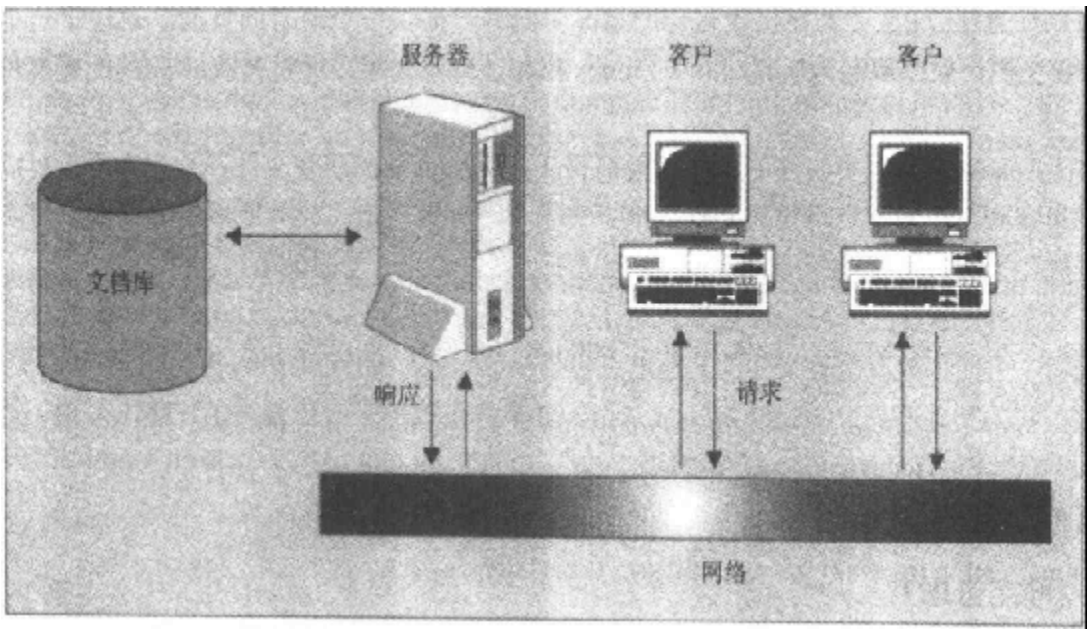


图 19-10

整个系统按“请求和响应”的方式运转着，客户请求一个文档，服务器发送一个文档或一个错误（如果文档不存在的话）做为响应。如果文档是由HTML文本和图像组成的，客户在请求了那份文档之后可能还要继续请求该文档要求的其他文件，比如<IMG>标签中给出的图像文件等等。

客户和服务器之间没有多少“交谈”。服务器的做法很简单：对客户请求做出响应，记下发送了哪些文件，但它并不记录客户曾经提出过的请求

HTTP守护进程可以为许多客户提供服务，它按自己收到请求的顺序对它们依次做出处理。事实上，许多HTTP守护进程的实现版本都采用多进程的办法增强请求和响应的处理能力。

19.6.2 设置一个服务器

许多Linux发行版本都已经带有一个HTTP服务器了，它一般是非常流行的（同时也是自由免费的）“Apache Server”服务器。虽然还有许多其他的服务器，而且其中有许多也是免费的，但Apache服务器应该能够满足你的一切要求；如果没有很特殊的理由需要使用其他的Web服务器，就用它好了。

大多数Linux发行版本都自带有一个Web服务器，一般来说就是Apache。我们只需在自己的缺省配置里选择安装并运行它就行了，没有什么复杂的东西。如果读者的发行版本不带Web服务器，或者愿意一点一滴地来安装它，我们建议你使用来自http：//www.apache.org的Apache。它是一个自由免费的服务器软件，有着出色的质量，它的网站文档齐备，而且安装起来也很容易。

安装时需要对缺省主页进行配置，缺省主页是这样一个页面：如果客户没有指定任何页面，服务器就会把它提供给客户。如果我们想在因特网上去Wrox出版社看看，那么，我们有充分的理由认为http：//www.wrox.com是一个可以考虑的计算机地址，但应该请求哪个页面来看呢？

答案是把它空着不填，就拿http：//www.wrox.com/试试，也就是不指定准备浏览的页面。

我们可以配置HTTP服务器在出现这样的情况时向客户提供一个缺省的页面。对应于这个页面的HTML文档一般都取名为Index.html或index.html，但我们可以通过修改httpd的配置文件来改变它。

查阅Apache自身档案文件的工作一般都要从查找Apache的配置文件开始。你不仅可以在配置文件里查到自己的文档都被放到什么地方去了，还可以在追踪故障原因的时候查到日志记录文件被放在了什么地方。研究Apache的配置选项也是一件很有意思的事情。

如果你不知道自己的Apache被安装到什么地方去了，可以从/etc/rc.d/init.d.httpd文件里找到一些线索。可能存放有配置文件的其他地点包括/etc/httpd/conf或/usr/local/Apache/conf子目录。你应该查找的文件叫做httpd.conf或srm.conf。如果你想找的是自己编写的HTML文档，请检查srm.conf文件里的DocumentRoot配置选项。

总而言之，在Linux上建立一个Web服务器是非常轻而易举的！

19.7 可点击图片

HTML文档里的有这样一种图像，点击它的不同区域会引起不同的操作动作，这样的图像就叫做可点击图片。可点击图片又分为两大类，分别是服务器端可点击图片和客户端可点击图片。下面就对它们进行简单的介绍。

19.7.1 服务器端可点击图片

如果在HTML页面里的<IMG>标签里加上了ISMAP属性，浏览器客户就会在这个图片被点击的时候把这个图片的名字和鼠标点击位置的坐标(x, y)送回到服务器。不同的服务器在对这个请求的处理方面可能稍微有些差别，但最终的结果都是一样的：图片上的(x, y)位置被转换为一个将返回给浏览器的URL地址，而它就是浏览器将要加载的下一个页面。

我们用服务器上的一个“映射图”文件来实现这一做法。映射图文件把图像的各个区域定义为到不同URL地址的链接。在大多数情况下，图形区域可以是圆形、矩形和多边形；此外，你还可以指定一个缺省的URL。

映射图文件有两种生成方法。第一种方法需要使用xv等图形处理程序：它读入一个图形文件，把你鼠标光标位置的坐标显示出来；你边划分准备映射到URL的图形区域，边把该区域的坐标记下来。这个方法比较适合处理少量的简单图形，如果东西多了，就有点力不从心。

第二种方法要先找到这样一个程序：在你划分图形区域的同时，它会自动生成一个映射图文件。UNIX平台上就有几个这样的程序。下面是一个NCSA格式的映射图文件示例，它把图像的不同区域映射到不同的URL地址上去：

```
rect ovary.html 290,107 324,166
circle stigma.html 289,516 298,529
rect style.html 275,383 291,501
poly filament.html 183,366 109,491 120,496 186,375
poly filament.html 311,392 339,564 349,561 319,383
poly filament.html 264,396 261,541 268,536 271,397
circle anther.html 348,581 355,593
circle anther.html 266,550 277,556
circle anther.html 105,505 114,521
poly tube.html 287,158 328,168 328,208 314,183 280,176
```

```
poly sepals.html 325,230 315,186 200,160 232,204 325,237
poly sepals.html 332,233 371,264 403,334 406,371 422,372 424,323 375,246 329,227
poly corolla.html 211,205 348,249 403,366 343,399 159,344
```

可惜的是不同的UNIX服务器在映射图的实现上有细微的差异，这不仅表现在文件的格式上，还表现在服务器对它们的存取方式上。你必须参考自己服务器使用手册中的有关内容才能处理好这个问题。

要想在Apache服务器上实现服务器端映射图是比较容易的。我们只需把映射图文件放到HTTP文件的主目录的一个名为maps的下级子目录里，再象下面这样给HTML文档里的图像文件加上一个锚点就行了：

```
<A HREF = "http://maps/fuchsia.map"> <IMG ISMAP SRC = "fuchsia.gif"> </A>
```

有的服务器需要增加一个配置文件来指明映射图文件的保存位置，还需要把一个程序（通常是imagemap）保存到服务器的cgi-bin子目录里去。

19.7.2 客户端可点击图片

客户端可点击图片是可点击图片一种比较新的形式，计算和点击动作都将由客户程序来完成和处理，不再产生网络操作；与服务器端可点击图片相比，它在这些方面的优势是非常明显的。

我们先要用<IMG>标签及其USEMAP属性来定义客户端可点击图片，如下所示：

```
<A HREF="http:/fuchsia.map"><IMG SRC="fuchsia.gif" USEMAP="#fuchsiamap"></A>
```

然后在同一个HTML文档的某个地方用<MAP>和</MAP>标签把真正的映射图文件包括进来，如下所示：

```
<MAP NAME="fuchsiamap">
<AREA SHAPE="rect" COORDS="290,107 324,166"  HREF="ovary.html">
...
<AREA SHAPE="poly" COORDS="325,230 315,186 200,160 232,204 325,237"
HREF="sepals.html">
</MAP>
```

注意客户端映射图里数据域的次序与服务器端映射图是不一样的。

19.8 服务器端的预处理功能

Web服务器程序还有一个页面预处理功能，即在把页面发送给客户之前可以对它进行一定的处理。这就是服务器端的预处理功能，它使我们不仅能够在书写页面的时候访问服务器上的信息，甚至在取得页面时也能够这样做。大多数服务器端的预处理功能要使用序列“<!-#”开始，使用序列“->”结束。

每个服务器端预处理功能都有一条对应的指令，也许还有几个属性。这些指令是：

- echo，允许访问环境变量。常用的环境变量有：
  - DATE_GMT：当前格林威治日期和时间。现在叫做UTC时间。
  - DATE_LOCAL：当地日期和时间。
  - LAST_MODIFIED：当前文档最近一次被修改的日期和时间。
- include，在把HTML文档发送给客户之前先在其中额外插入一个文件。

• exec，在把HTML文档发送给客户之前先在其中插入一个命令的执行结果。

下面是在HTML文档里插入最近修改日期时间的服务器端预处理命令：

```
<!-# echo var = "LAST(MODIFIED" -->
```

并非所有的服务器都能够处理这些服务器端的预处理命令；即使能够处理，也可能会禁止掉这些功能或者只允许执行少数几个指令。服务器端的预处理功能增加了额外的处理工作，因而会给服务器带来许多麻烦。如果对所有文件都激活服务器端的预处理功能，httpd程序就必须对它发送的每一个文件进行处理以检查有无服务器端预处理指令，这就大大增加了服务器的处理负担。有些服务器只对某些带特殊扩展名的文件进行预处理，这个扩展名一般是.shtml。

另外一个问题是服务器端预处理功能等于是允许别人在你的计算机上运行程序，特别是在使用了exec指令的时候。把一台计算机设置为提供HTML页面服务就已经是对别人开放你的机器，也就等于降低了安全性；但因为这只是发送一些数据流，所以还是有一定的安全保证的。可如果你允许别人通过exec来执行程序问题就不一样了，你的安全性将急剧下降。因为这个原因，许多商业性的服务提供商都不允许服务器端的exec预处理功能。

动手试试：客户端映射图和服务器端预处理功能

下面这个HTML文档演示了客户端映射图和服务器端预处理两方面功能。在选择用做映射图的图像时一定要谨慎从事，如果它们不能显示在客户程序的窗口里，就不会有什么实际价值。大尺寸图像只有在本地加载时才是真正可以接受的做法。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Demonstration Page</TITLE>
</HEAD>
<BODY>
<H1><CENTER>This document is html11.html</CENTER></H1>
<P> This document was last modified on: <!-#echo var="LAST_MODIFIED"-></P>

<P> It demonstrates server side includes, a client side map, and other links.</P>

<P>
<HR>
</P> <P> Here is a fortune, different just about every time this page is loaded!</P>

<PRE>

<!-#exec cgi="/cgi-bin/fortune"->

</PRE>
<P>
<HR>
This is a clickable client side map.
</P>
<P> Try clicking on different parts of the fuchsia:</P>


<P>
<A><IMG SRC="fuchsia.gif" ALT="Fuchia" USEMAP="#fuschiamap">
</A>
</P>
<P>
<HR>
</P>
```

```
<P> If you want to find out about some other WROX books we suggest you visit
their <A HREF="http://www.wrox.com/">home page</A>.</P>

<P> For more information about the World Wide Web, visit the <A
HREF="http://www.w3.org/">World Wide Web Consortium</A>. You will find many
specifications, draft specifications, lists of httpd servers, and lots of other
helpful files. </P>

<P> If you are just getting started with Linux, then you should subscribe to
the newsgroups
<A HREF="news:comp.os.linux.announce"> comp.os.linux.announce</A>, and
<A HREF="news:comp.os.linux.answers"> comp.os.linux.answers</A>.</P>

<P> Some people are not sure how Linux is
<A HREF="english.au">pronounced</A>. Well now you know!</P>

<P>
<HR>
</P>
<P>
<IMG SRC="pblinux.gif" ALT="Powered by Linux" ALIGN="middle">
This server is powered by Linux!
<HR>
<CITE>Many thanks to <A HREF="mailto:Alan.Cox@linux.org">Alan Cox</A> for
permission to use this graphic.</CITE>
</P>
<P>
This page is &copy; Copyright Wrox Press.
</P>

<MAP NAME="fuschiamap">
<AREA SHAPE="rect" COORDS="290,107 324,166" HREF="ovary.html">
<AREA SHAPE="rect" COORDS="279,504 300,530" HREF="stigma.html">
<AREA SHAPE="rect" COORDS="275,383 291,501" HREF="style.html">
<AREA SHAPE="poly" COORDS="183,366 109,491 120,496 186,375" HREF="filament.html">
<AREA SHAPE="poly" COORDS="311,392 339,564 349,561 319,383" HREF="filament.html">
<AREA SHAPE="poly" COORDS="264,396 261,541 268,536 271,397" HREF="filament.html">
<AREA SHAPE="rect" COORDS="333,561 363,581" HREF="anther.html">
<AREA SHAPE="rect" COORDS="255,538 280,563" HREF="anther.html">
<AREA SHAPE="rect" COORDS="95,496 120,521" HREF="anther.html">
<AREA SHAPE="poly" COORDS="287,158 328,168 328,208 314,183 280,176" HREF="tube.html">
<AREA SHAPE="poly" COORDS="325,230 315,186 200,160 232,204 325,237"
HREF="sepals.html">
<AREA SHAPE="poly" COORDS="332,233 371,264 403,334 406,371 422,372 424,323 375,246
329,227" HREF="sepals.html">
<AREA SHAPE="poly" COORDS="211,205 348,249 403,366 343,399 159,344"
HREF="corolla.html">
</MAP>

</BODY>

</HTML>
```

The resulting page is:

我们将看到如图19-11所示的浏览器画面。

操作注释：

服务器端预处理功能#echo把当前日期和时间插入到页面里。

服务器端预处理功能#exec执行了fortune程序（fortune程序保存在cgi-bin子目录里，而这个子目录是由Apache服务器设定的），把执行后得到的结果文本插入到页面里。我们把这个结果放在<PRE>和</PRE>标签中间以保留它的格式，这对许多fortune程序的“cookie”来说是很重要的。因为每次获取这个页面的时候都会调用fortune，所以它会返回一个伪随机结果，使每次请

求这个页面时得到的结果都不相同。

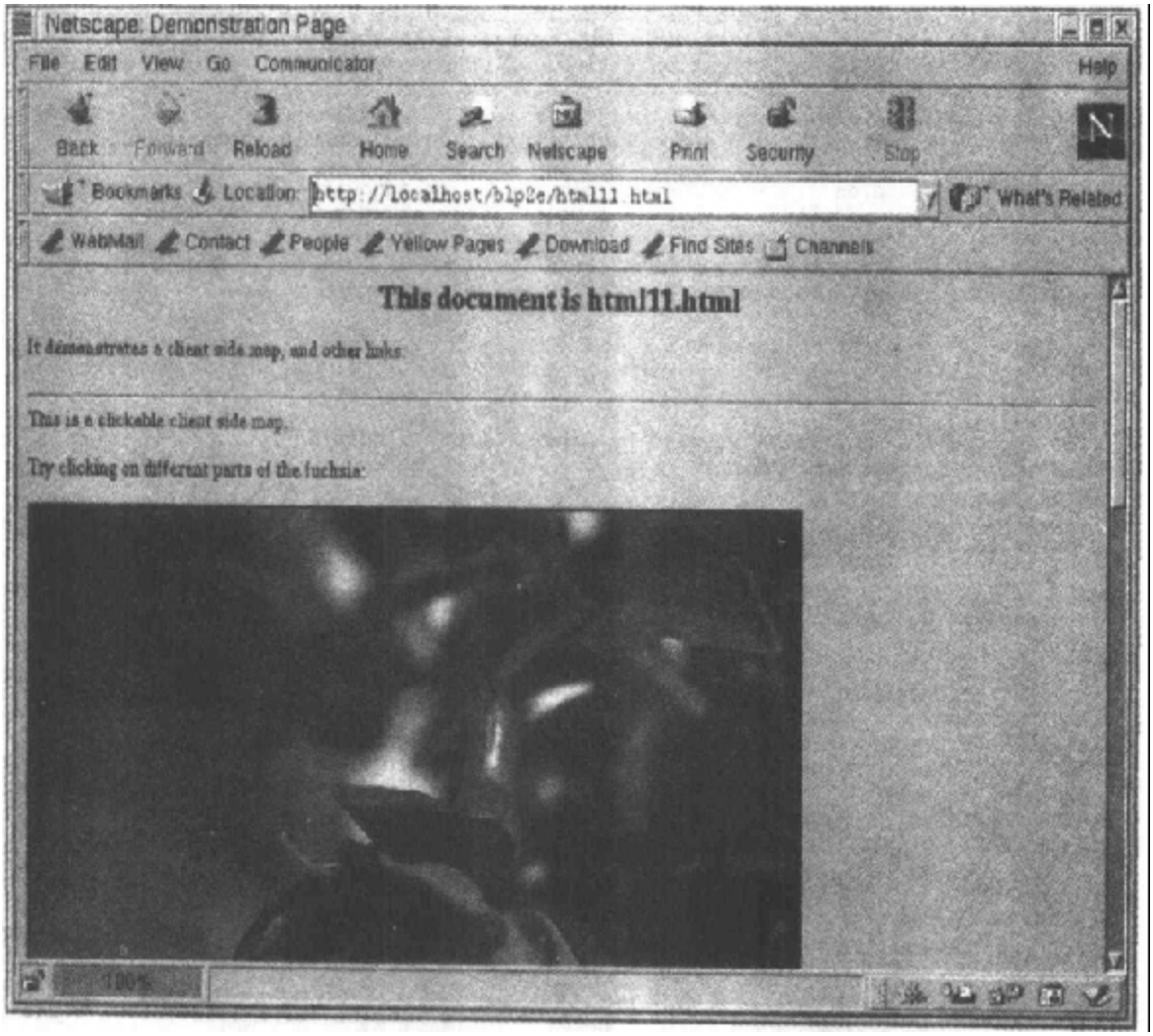


图 19-11

最后，客户端映射图使用户能够通过点击图像的不同部分转向不同的链接。

19.9 编写WWW主页的技巧

下面是一些编写Web主页的小技巧：

- 记住文档是可以被链接的。你不需要把全部内容都塞进一个文档里。有些内容可以用超文本链接来隐藏起来。
- 不要把自己的文档分成太多的“碎片”。如果你把自己的文档分得太碎，每一页上的信息就会很少，这些只言片语很难领会。
- 遵守HTML标准。即使你最喜欢的浏览器里实现了一些对标准的扩展，也并不意味着你必须使用它们。并不是每个人都使用着和你一样的浏览器。对浏览器所能支持的功能尽量做保守的估计。
- 别忘了检查自己的HTML文档。有几个非常好的HTML检查器是可以免费获得的。要用它

们来保证你HTML文档的正确性。

- 谨慎对待图像。如果你在自己的页面里放上了尺寸很大的图像，就会让使用慢速调制解调器连接的人们等图像等得不耐烦。请记住，调制解调器连接要比在服务器机器上的浏览慢很多。
- 在<IMG>标签里使用ALT属性。这将使非图形化浏览器用户（以及那些禁止了图像下载功能以提高浏览速度的人们）能够浏览你的页面。请记住，并不是每个人都能看到图像的，请在你自己的机器上测试不带图像的文档，看它们能不能正确显示和动作。
- 文档的标题通常会被人们用做指向这个文档的书签。要让它尽量简洁和准确。

在因特网上各种各样的HTML文档编写指南里你还可以找到许多窍门、技巧和规定。

19.10 本章总结

我们在这一章里学习了HTML语言的基本知识，利用这种置标语言编写的页面将出现在WWW网上。

我们学习了如何在页面里加上图像，如何把页面链接在一起——不仅是本地站点上的页面，还包括因特网上的页面呢。

我们还告诉大家：检查你的HTML文档将保证它能够适应广泛的计算机平台。

我们还向大家介绍了一些可以用在HTML页面里的高级功能，它们会在服务器把这个页面提供给客户程序时发挥作用；如果只是读取本地文件（不使用浏览器），那它们不过是一些普通的文本而已。

最后，我们给大家介绍了一些编写HTML页面时的小技巧。

第20章　因特网程序设计II：CGI

我们在上一章里学习了如何把信息编写到HTML文档里，而这些信息既可以从本地计算机上查看，也可以通过网络来浏览。这确实是一个传播信息的好办法，但美中不足的是它还是一些静态的东西。为了能够提供动态的信息，我们需要让用户能够通过Web页面与服务器上的程序进行动态的交流。

在这一章里，我们将学习怎样才能让浏览器把信息发送回服务器，服务器又是怎样把这个信息传递给程序，程序又如何以一种动态的方式对客户程序做出响应。我们将只对服务器端的处理过程进行讨论，代码就是在这个远离客户的地方执行的。我们将集中讨论服务器端的独立程序，而不是讨论类似于PHP使用的嵌入在Web页面里的脚本。

编写客户端程序也是可以做到的，它们通常被称为动态HTML（Dynamic HTML，简称DHTML）。这是一些在客户端计算机上执行的脚本，但我们不准备在这里讨论它们。

浏览器向服务器回传信息需要有一个程序接口，定义这个接口的技术规范叫做“通用网关接口”（Common Gateway Interface），人们一般把它简称为CGI接口。接受那些来自浏览器的信息的服务器端程序也就被称为CGI程序，它们对这些信息进行处理，然后通过HTTP协议把命令或动态文档发送回浏览器（如图20-1所示）。

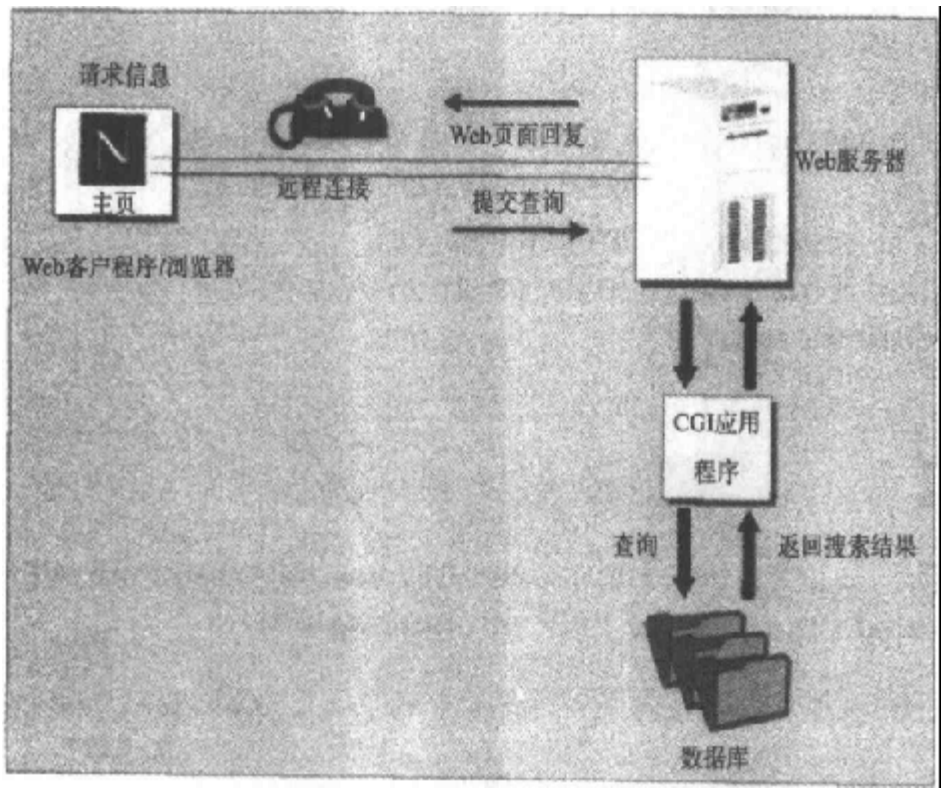


图　20-1

在开始学习CGI之前，我们先来看一个以前没有遇到过的HTML结构：表单。

20.1 表单元素

在我们前面看过的HTML文档里，所有标签都只与一件事情有关——为信息在客户端的显示而控制其布局结构；除超文本链接部分以外，对来自浏览器的输入可以说是无能为力。但这并不是事情的结束。

我们可以通过<FORM>标签在HTML文档里安排一些允许用户进行输入的域。这是一个复合型标签。它里面还可以再加上其他一些只能用在表单里的标签。<FORM>标签和它嵌入标签（人们称之为“表单（FORM）元素”）的语法定义如下所示：

```
<FORM ACTION= METHOD= ENCTYPE= >
  <INPUT NAME= TYPE= VALUE= SIZE= MAXLENGTH= CHECKED>
  <SELECT NAME= SIZE= MULTIPLE>
    <OPTION SELECTED VALUE= >
  </ SELECT>
  <TEXTAREA NAME= ROWS= COLS= > </ TEXTAREA>
</FORM>
```

除此之外，我们还可以把上一章掌握的“普通”标签用在<FORM>标签里。注意几乎所有的标签都有一个NAME属性，这个属性是为服务器准备的。我们将在本章后面介绍服务器对表单的一般性处理时再详细讨论如何对NAME属性进行处理。

我们可以看出，<FORM>是一个很复杂的标签。下面来依次学习它的各个元素。

20.1.1 FORM标签

<FORM>标签标志着一个HTML表单的开始，它有三个属性：

- ACTION属性：对表单进行处理需要有一个程序，这个属性给出的就这个程序的URL地址。
- METHOD属性：这个属性的取值或者是GET，或者是POST。
- ENCTYPE属性：如果不想在发送这个表单时包括上一个文件，就可以省略这个属性。如果确实想加上一个文件，就要把这个属性设置为"multipart/form-data"，但这的确是很少见的。如果你只是想提交这个表单，那就用它的缺省值"x-www-form-urlencoded"好了。

总而言之，这几个属性控制着信息回传给服务器的方式。ACTION的值必须指向一个能够在服务器上被启动调用的程序。这些程序一般都保存在HTTP服务器上某个与文档页面不相干的子目录里，几乎所有系统都使用cgi-bin做为这个子目录的名字。METHOD属性控制着信息是如何送到服务器上的某个程序去的。我们过一会儿再回到cgi-bin子目录里的程序和METHOD属性上来。

20.1.2 INPUT标签

<INPUT>标签定义了客户输入的类型，输入数据的格式和行为由TYPE属性控制。TYPE属性支持的取值包括以下8种：

1. TEXT

当我们把TYPE的值设置为TEXT的时候，浏览器将显示一个单行的输入框，用户可以在这

个框子里输入文本。NAME属性给这个输入框起了个名字，当这个表单在服务器上被处理的时候就会用到这个名字。SIZE属性给出了这个输入框在Web页面上显示出来的宽度；而MAXLENGTH设定了这个输入框最大的输入长度；如果这个值大于SIZE，输入框会随着用户的输入而卷动。VALUE属性给出了这个输入框被显示时出现在其中的缺省字符串。

下面这段HTML演示了TEXT类用户输入框的用法：

```
Please enter your
<BR>
salutation: <INPUT TYPE=TEXT NAME=sal SIZE=5 MAXLENGTH=10 VALUE="Mr. ">
<BR>
first name: <INPUT TYPE=TEXT NAME=fname SIZE=20 MAXLENGTH=30>
<BR>
second name: <INPUT TYPE=TEXT NAME=sname SIZE=20 MAXLENGTH=30>
```

2. PASSWORD

PASSWORD输入框的用法与TEXT输入框的用法是一样的，只是它的内容不会显示在浏览器里。但从真正口令字的角度看这并不是个安全的办法，因为这个域在被传递到服务器去的时候是纯文本形式；如果有人截获了传递给服务器的表单信息，就可以直接看到这里输入的口令字。但不管怎么说，它可以防止别人看到你输入在这个框子里的东西。

下面这段HTML演示了PASSWORD类输入框的用法：

```
Password: <INPUT TYPE=PASSWORD NAME=passwd SIZE=8>
```

3. HIDDEN

把TYPE设置为HIDDEN得到的输入框是不会出现在浏览器显示画面里的，当然用户也就没办法往里面输入东西。这类输入框是供服务器程序使用的。与PASSWORD类型的输入框相类似，它也称不上安全。

下面这段HTML演示了HIDDEN类输入框的用法：

```
<INPUT TYPE=HIDDEN NAME=camefrom VALUE="foo/bar/baz.html">
```

4. CHECKBOX

CHECKBOX类型允许用户从一组选项里选择几项。几个NAME属性相同的选择框将被划分为一组，而浏览器将允许用户从一组同名的选择框里进行选择。

CHECKBOX比其他输入框多出一个CHECKED属性，其作用是提供缺省的选择，缺省选择可以有多个。VALUE属性用来把信息返回给服务器。为了让用户能够了解自己选择的是什么，你应该给每个选择框加上一些提示性的文本。

下面这段HTML演示了CHECKBOX类输入框的用法。用户将看到一个显示着世界各地的选择区，这个选择区允许进行多项选择。如下所示：

```
<FORM ACTION="program" METHOD=POST>
<BR>
Please indicate which areas of the world you would like
to visit:<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="1">Asia<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="2" CHECKED>Africa<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="3">North America<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="4">South America<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="5">Antarctica<BR>
```

```
<INPUT TYPE=CHECKBOX NAME=cb VALUE="6">Europe<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="7" CHECKED>Australasia<BR>
</FORM>
```

5. RADIO

RADIO类型的<INPUT>标签与CHECKBOX很相似，但它每次只允许选择一项，标记为CHECKED的缺省选择也只能有一个。如果没有设定缺省选择，浏览器就在第一次显示这些单选按钮的时候把第一个按钮设置为选中状态。

下面这段HTML演示了RADIO类输入框的用法。用户还看到一个显示着世界各地的选择区，但这个选择区只允许进行一项选择。如下所示：

```
<FORM ACTION="program" METHOD=POST>
<BR>
Please indicate in which area of the world you
live:<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="1">Asia<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="2">Africa<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="3" CHECKED>North America<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="4">South America<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="5">Antarctica<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="6">Europe<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="7">Australasia<BR>
</FORM>
```

6. IMAGE

IMAGE类型的<INPUT>标签允许用户从一幅图像上选择一个(x, y)坐标，返回给服务器的输入数据与映射图的形式差不多。现在已经很少有人会使用它了，因为映射图办法的效果更好，所以我们也就不多说什么了。

7. SUBMIT

SUBMIT类型会使浏览器在浏览窗口里显示一个按钮。当这个按钮被选中的时候，表单的内容就将被传递到服务器去进行处理。几乎使用的表单都会有一个SUBMIT提交按钮，这个按钮的具体文字内容由VALUE属性给出。

下面是一个提交按钮的例子：

```
<INPUT TYPE=SUBMIT NAME=processnow VALUE="Submit Form">
```

8. RESET

“TYPE=RESET”将使浏览器在浏览窗口里显示一个按钮。当这个按钮被选中的时候，表单元素会被重新设置为表单第一次被加载时候的初始值。这个操作完全是由浏览器负责的，不会引起与服务器的互动。RESET类型也可以有一个VALUE属性。请看下面的例子：

```
Clear the form: <INPUT TYPE=RESET VALUE="Reset Form">
```

20.1.3 SELECT标签

<SELECT>标签允许用户从一系列值里进行选择。它的属性包括NAME、MULTIPLE和SIZE。

NAME属性和往常一样用来给将返回到服务器的信息起名字。MULTIPLE属性允许一个以上的选项被选中，但用户每次见到的只有一个。使用这个属性的人并不多，因为选择框也能实现

这一效果，而且做得更好。SIZE属性定义了能够被选取的项目的个数，它的缺省值是1，即给出的选择清单每次只能选择一个选项。

<SELECT>标签中的每个项目都要用一个<OPTION>标签来设定，它的VALUE属性给出了该项被选中时将要返回给服务器的值。可选属性SELECTED用来指定缺省的选择值。

下面是一个<SELECT>标签的用法示例：

```
<SELECT NAME="Speed">
<OPTION VALUE="vslow">9600 or slower
<OPTION VALUE="slow">14400
<OPTION SELECTED VALUE="OK">28800
<OPTION VALUE="quick">better than 28000
</SELECT>
```

20.1.4 TEXTAREA标签

<TEXTAREA>标签允许用户输入并向服务器返回多行文字。除常见的NAME属性以外，它还有用来设定输入区大小的ROWS（行）和COLS（列）属性。浏览器会根据ROWS和COLS的值来显示一个文本区，为了方便用户在文本输入超出这个区域时的查看之用，还会给文本区再加上一个卷动条。

下面是一个<TEXTAREA>标签的用法示例：

```
Enter your address:<BR>
<TEXTAREA NAME="address" ROWS=5 COLS=50>
</TEXTAREA>
```

如果文本内容被输入到起始<TEXTAREA>标签和结束<TEXTAREA>标签之间，那么当这一页被显示时它就作为文本区的缺省文本。

20.2 一个主页示例

为了给下面的学习做好准备，也为了对前面的学习做一个总结，我们在这里给出一个比较全面的“动手试试”。你准备出门旅行，于是去了一家虚拟的旅行社，你向这家旅行社查询你旅行目的地的情况，并给出自己调制解调器的速度（为了让他们对你旅行手册里的图像内容做出适当的安排）。你的居住地将用来确定这家旅行社的哪个分社将与你具体接洽。

动手试试：一个简单的的查询表单

1) 我们从下面这个基本的HTML文档模板开始，先给它起名为cgi1.html。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>A Simple HTML Form Document</TITLE>
</HEAD>
<BODY>
<H1>A demonstration of an HTML form</H1>
<P>

</P>
</BODY>
</HTML>
```

2) 从下面开始是构成整个页面的表单元素。先是TEXT和PASSWORD类型的元素：

```
<FORM ACTION="program" METHOD=GET>

Enter your name: <INPUT NAME=name TYPE=TEXT SIZE=20 MAXLENGTH=40>
and password: <INPUT NAME=passwd TYPE=PASSWORD SIZE=8 MAXLENGTH=8><BR>
<BR><BR>
```

3) 我们向顾客提供一系列CHECKBOX选择框，让他从中选择自己的出访目的地：

```
Please indicate which areas of the world you would like
to visit:<BR>
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="1">Asia
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="2">Africa
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="3">North America
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="4">South America
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="5">Antarctica
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="6">Europe
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="7">Australasia
<BR><BR>
```

4) 为了节约时间，我们把上一部分的内容抄过来，然后用单选按钮输入框确定顾客的居住地。如下所示：

```
Please indicate in which area of the world you
live:<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="1">Asia<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="2">Africa<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="3" CHECKED>North America<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="4">South America<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="5">Antarctica<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="6">Europe<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="7">Australasia<BR>
<BR><BR>
```

5) 接下来我们用<SELECT>标签了解顾客的调制解调器速度，如下所示：

```
Please indicate your modem speed:
<SELECT NAME="Speed">
<OPTION value="none">No modem
<OPTION value="vslow">9600 or slower
<OPTION value="slow">14400
<OPTION SELECTED value="OK">28800
<OPTION value="quick">better than 28000
</SELECT>
<BR><BR>
```

6) 让顾客在一个TEXTAREA类型的输入框里输入自己的地址：

```
Please enter your address:<BR>
<TEXTAREA NAME="address" ROWS=5 COLS=50>
</TEXTAREA>
<BR><BR>
```

7) 最后，给表单加上提交按钮和重置按钮，然后结束<FORM>标签。

```
<INPUT TYPE=RESET VALUE="Clear fields">
<CENTER>
<INPUT TYPE=SUBMIT VALUE="Send Information">
</CENTER>
</FORM>
```

把这些内容都敲完之后，如果你在自己的Web浏览器里查看这个表单，就应该看到如图20-2

所示的画面。

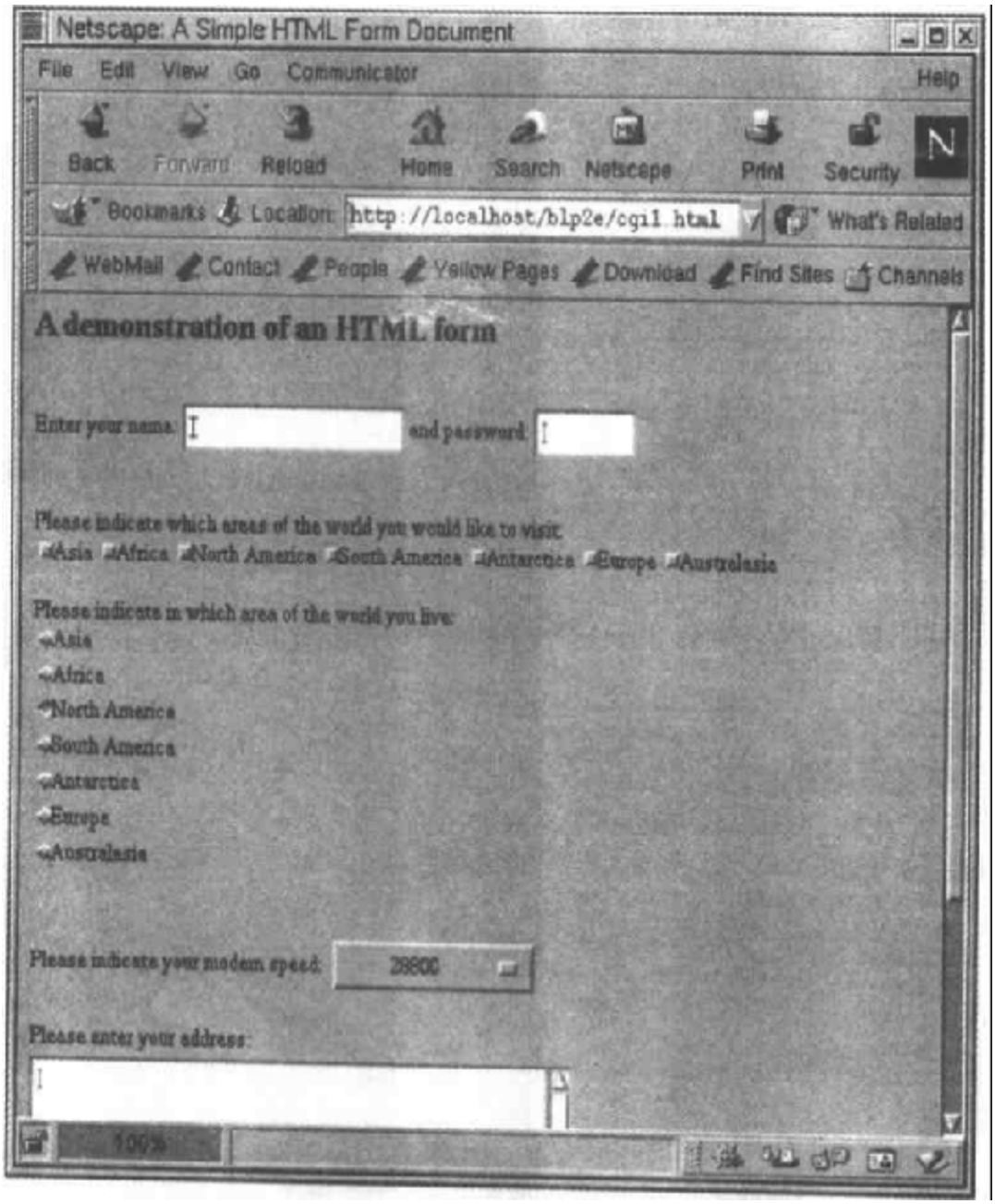


图 20-2

操作注释：

我们以一个基本的HTML文档为基础开始编写我们的页面。然后我们用<FORM>标签开始构造一个表单。我们现在暂时不要理会ACTION和METHOD属性。

接下来，我们通过TEXT和PASSWORD类型的<INPUT>标签让用户输入自己的姓名和口令字。因为我们没有在它们之间加上
标签，所以它们将出现在同一行上。

在这段文本后面我们用了一个
标签重新开始一个新段落。我们先显示一行提示信息，

然后用CHECKBOX类型的<INPUT>标签让用户做多项选择。这部分<INPUT>标签里都有一个相同的“NAME=cb”域，这将使浏览器把这一组选择项安排在一起。

再另起一段，我们用一组RADIO类型的<INPUT>标签让用户只能做出一项选择。我们给这组单选按钮都加上一个相同的“NAME=rb”域，也就是把它们分为一组。我们还用CHECKED属性设置了一个缺省选项。

调制解调器速度部分演示了另一种类型的选择办法——SELECT类型。这类型的元素在不同计算机平台和不同浏览器上的显示效果有着很大的差异。我们窗口画面里给出的是OSF /Motif风格的SELECT显示效果。

我们接着用<TEXTAREA>标签提示用户输入一些格式随意的信息，这里是用来输入他的邮政地址。

最后，我们还准备了一个用来以缺省值重置表单的按钮和一个用来向服务器发送信息的提交按钮。注意这里使用的<CENTER>标签虽然能够满足我们的想法，但在HTML 4里已经贬值了——因为我们的目的是为了演示HTML的基本用法而不是为了向大家介绍样式表或XHTML等复杂的东西。

20.3 向WWW服务器发送信息

在学习了如何编写HTML文档以使浏览器用户能够输入信息并把它们发送回服务器之后，我们要到系统的另一端去看看信息是如何传递到服务器以及它们又是如何在那里被处理的。

当用户选中提交按钮的时候，浏览器将开始对表单里的信息进行编码并会使用HTTP协议把它们发送给服务器，服务器软件将负责启动一个存放在服务器机器上的对应程序并把这些信息传递给它。服务器上的那个程序传统上被称为是一个CGI（或cgi-bin）程序，这是因为WWW服务器软件与这个程序之间的接口是由CGI技术规范定义的。其他的办法还有另外启动一个独立的程序，这通常被称为“内进程处理”。IIS（微软公司出品的一个服务器软件）的ISAPI接口和Netscape服务器的NSAPI接口用的都是内进程处理模型，Java服务器插件（servlet，即服务器小程序）也采用这种执行方式。我们将在这一章的末尾看到我们自己的内进程处理——我们把一个Perl解释器嵌入到Apache的Web服务器里去，这样就能用Perl语言直接编写CGI程序，不再需要每执行一个CGI程序就启动一次Perl解释器了。

20.3.1 对信息进行编码

我们已经见过<FORM>标签的ENCTYPE属性了，它的作用就是设定一个用来向服务器回传信息的编码方案。HTML文档一般都会省略它，这是因为它的缺省值applicatin/x-www- form-urlencoded通常就是人们所需要的。

在对表单数据进行处理以便回传给服务器的时候，浏览器将对它们做一系列转换以达到对数据进行编码的目的。这些转换主要包括：

表单域的名字和值将被“转义”。具体做法是这样的：每一对域和值里的空格都会被替换为一个加号（+）字符。不是字母或数字的字符将被替换为它们的十六进制数字形式，格式为%HH，HH是该字符的ASCII十六进制值。
标签将被替换为“%0D%0A”。

信息是按它们在表单里出现的顺序排列的，数据域的名字和数据域的值通过等号（=）字符连接在一起。各对名/值再通过“&”字符连接在一起。“;”字符在很多时候可以用来替代“&”字符，但这还没有得到有关标准的认可。

经过这些编码处理之后，表单信息就整个成为一个连续的字符流（中间没有空白字符），里面包含着将被送往服务器的全部信息。

20.3.2 服务器程序

我们现在来看看服务器上的程序。它们通常都被保存在HTTP服务器主目录的一个下级子目录里，最常见的名字是cgi-bin（因为它们是一些实现了CGI接口的二进制可执行程序）。将被启动的程序是由<FORM>标签里的ACTION属性指定的，如下所示：

```
<FORM ACTION = ",cgi-bin/myprogram" METHOD - POST >
```

安全性

现在正是给大家在安全性方面敲敲警钟的时候。当你允许他人向你的服务器发送表单信息的时候，就等于允许他们任意构造数据并把它传递到某个程序去，而这个程序将要在你的服务器上运行。这就是一个潜在的安全性方面的问题。根据CGI程序对数据的处理方式和你是否允许它再调用其他的程序，就可能会有人找到启动一些你不愿意让他们启动的程序的办法。请密切关注最新的WWW安全问题FAQ（常见问题答疑）报告，并且尽量不让或少让cgi-bin子目录里的程序具备调用其他程序的能力。

对初学者来说，下面是一些极有用的建议：

- 不要相信客户。你根本不能控制他们使用的是什么浏览器，也根本控制不了浏览器都会把什么东西发送到你的程序来。
- 在CGI程序里不要使用shell或Perl的eval语句，因为它的潜台词是允许使用一个任意的字符串来调用程序。
- 尽量不要或少调用其他程序，特别是其中带有popen和system函数调用的程序。如果客户足够聪明，就有可能通过精心构造的输入数据调用“错误”的程序。

但说归说，许多ISP提供的WWW服务项目里还是有允许顾客自行编写CGI程序这一项，因为非授权访问而出现的惊险故事也算不上多。对安全问题应该引起足够的重视，可也别因为这个就不建立Web服务器了！

20.3.3 编写服务器端的CGI程序

在可以用哪种语言来编写CGI程序方面没有什么硬性的规定。在这一章里，我们基本上是坚持使用C语言；但只要你愿意并做得到，用shell脚本程序、Tcl或者现在最时髦的Perl都没问题。

服务器程序可以通过三种途径接收信息：通过环境变量、从命令行和从标准输入。具体使用哪一种办法要由<FORM>标签的METHOD属性来决定。

在“METHOD = GET”的时候，向CGI程序传递表单编码信息的正常做法是通过命令行来进行的。但如果系统对允许以这种方式传递的信息的总量有限制，就有可能造成一些问题。所

以许多服务器采用的是另外一种做法：除那些最简单的请求是通过命令行来传递的以外，大多数表单编码信息都是通过一个名为QUERY(STRING的环境变量来传递的。

如果“METHOD = POST”，表单信息将通过标准输入来读取。

> 从传统上讲，GET方法主要用来处理那些“没有副作用”的表单，即那些只向服务器提出一个简单查询的表单；而POST方法则用来处理那些可能会在服务器上引起某些变化的复杂表单。

还有一种根本不使用表单就可以向CGI程序传递信息的办法——这就是把信息直接追加在URL地址后面，信息和URL之间用问号（?）来分隔。具体做法会在后面介绍。

1. 环境变量

不管你使用哪种方法来传递表单信息，有几个重要信息都是以环境变量的形式传递到CGI程序去的。有关标准给出了如表20-1中一些变量：

表 20-1

变 量 名	说 明
SERVER_SOFTWARE	与接收请求并调用程序的那个服务器软件有关的资料
SERVER_NAME	服务器的主机名或IP地址
GATEWAY_INTERFACE	服务器上实现的CGI标准的版本号
SERVER_PROTOCOL	接收来自客户的请求时使用的协议的版本号
SERVER_PORT	接收到请求的端口号。WWW服务器一般使用80号端口
REQUEST_METHOD	提出请求的方法，即GET或POST
PATH_INFO	关于CGI程序路径的附加信息
PATH_TRANSLATED	CGI程序的物理路径
SCRIPT_NAME	正在执行的脚本程序的名字
REMOTE_HOST	提出这一请求的计算机主机的名字
REMOTE_ADDR	提出这一请求的计算机主机的IP地址
AUTH_TYPE,	如果服务器支持用户身份验证，就要用到这两个变量
REMOTE_USER	
REMOTE_IDENT	远程用户的用户名。这很不可靠，也很少使用
CONTENT_TYPE	正被传输的信息的内容类型。它通常是application/x- www-form-urlencoded
CONTENT_LENGTH	传给程序的数据的字节数。当以“METHOD = POST”方式读取输入的时候，最好利用这个变量来识别结束字符流的null字符或文件尾标志

当“METHOD = GET”时，不允许设置CONTENT(TYPE和CONTENT(LENGTH变量。

除了这些，有的服务器还会再添加一些其他的变量。这些变量一般都以“HTTP_”打头，这是为了避免与HTTP协议或CGI技术规范的未来版本将增加的变量名发生冲突。

Apache服务器和许多其他种类的服务器还提供了：

QUERY(STRING 它包含着“METHOD = GET”或信息作为URL的一部分时传递给CGI程序的信息。

2. 学以致用：我们的第一个CGI程序

经过前面的学习，现在已经可以开始编写我们的第一个CGI程序了。我们用这个程序来显示

客户提出请求时的环境变量。其实已经有现成的库函数可以替我们完成这一工作了，但自己动手可以加深我们对这一问题的理解，它会帮助我们更好地消化有关的基本原理。

我们现在还不太明白在对来自客户的信息进行了处理之后，我们的CGI程序如何才能把处理结果返回给客户。事实上，CGI程序的标准输出将直接送往浏览器，但在向客户发送HTML数据之前，需要先发送一些HTTP表头信息。就目前而言，只要知道照下面这样做让浏览器把简单的文本显示到屏幕上就已经足够了：写下“Content-type: text/plain”，加上一个空白行，再加上你打算显示的文字。

我们这个例子将使用一个shell程序，因为它是能够完成我们这个信息处理工作的最简单的办法。

动手试试：我们的第一个CGI程序

1) 我们将要编写的程序叫做cgi1.sh，所以我们把它注释在程序头部。如下所示：

```
#!/bin/sh
# cgi1.sh
# A simple script for showing environment variable information passed to a CGI
program.
```

2) 我们用刚才介绍的两行开始向浏览器进行输出：

```
echo Content-type: text/plain
echo
```

3) 然后我们把调用这个CGI程序时的命令行参数也显示出来：

```
echo argv is "$*"
echo
```

4) 接下来是CGI请求出现时的环境变量，这是我们这个程序的精华所在。如下所示：

```
echo SERVER_SOFTWARE=$SERVER_SOFTWARE
echo SERVER_NAME=$SERVER_NAME
echo GATEWAY_INTERFACE=$GATEWAY_INTERFACE
echo SERVER_PROTOCOL=$SERVER_PROTOCOL
echo SERVER_PORT=$SERVER_PORT
echo REQUEST_METHOD=$REQUEST_METHOD
echo PATH_INFO=$PATH_INFO
echo PATH_TRANSLATED=$PATH_TRANSLATED
echo SCRIPT_NAME=$SCRIPT_NAME
echo REMOTE_HOST=$REMOTE_HOST
echo REMOTE_ADDR=$REMOTE_ADDR
echo REMOTE_IDENT=$REMOTE_IDENT
echo QUERY_STRING=$QUERY_STRING
echo CONTENT_TYPE=$CONTENT_TYPE
echo CONTENT_LENGTH=$CONTENT_LENGTH
    exit 0
```

5) 我们需要对我们那个HTML示例文档进行修改（就叫它cgi2.html好了），使表单提交操作指向我们这个shell脚本程序。具体修改办法是把紧跟在HTML标题后面的ACTION属性改为如下所示的样子：

```
<FORM ACTION="/cgi-bin/cgi1.sh" METHOD=POST>
```

6) 最后，因为只有通过WWW服务器来访问CGI程序才能使它们被正确地启动调用，所以我

们必须把程序文件拷贝到相应的子目录里去。我们的CGI程序cgi1.sh应该被拷贝到服务器主目录下的cgi-bin子目录里去；而我们的HTML文档cgi2.html还是和其他HTML文档放在一起好了。我们还必须确保cgi1.sh是一个可执行文件。

现在可以通过服务器来访问我们的表单了。当我们按下表单上的提交按钮后，将看到如图20-3所示的页面。

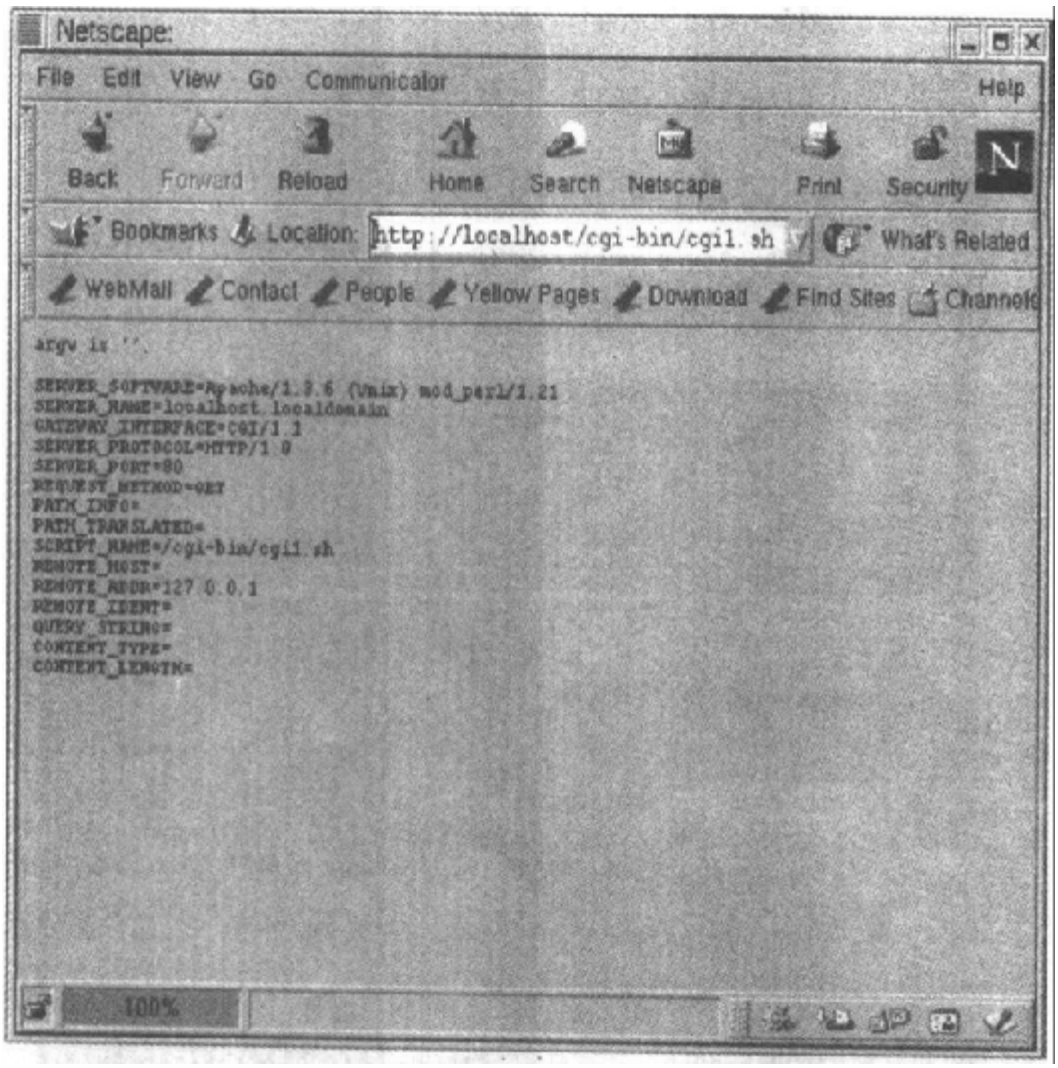


图 20-3

操作注释：

当用户按下提交按钮的时候，表单数据将被送往服务器，再由服务器来启动cgi1.sh程序。cgi1.sh程序通过自己的标准输出把数据返回给浏览器，再由浏览器把这些返回数据显示到屏幕上。注意：虽然ACTION属性的值看起来象是一个绝对路径，但服务器会在调用这个程序之前把服务器文件的主目录路径名自动加上去。

再请大家注意：到目前为止我们还没有见到来自表单元素的实际数据，我们看到的只是CGI程序被调用时所处的环境

现在对这个CGI脚本程序做一点谨慎的扩展，让它能够读取传递给它的数据。因为我们使用的是“METHOD = POST”，所以这个数据将出现在标准输入上。

动手试试：读取来自表单的数据

拷贝cgi1.sh，把新文件保存为cgi2.sh。我们给自己的脚本程序做点小修改，让它能够读取和返回出现在标准输入上的数据。把下面这些代码添加到脚本程序的末尾：

```
echo The data was:
read x
while [ "$x" != "" ]; do
    echo $x
    read x
done
```

别忘了把HTML表单（即cgi2.html）里的ACTION属性改为指向我们新的CGI程序cgi2.sh。我们提交表单时的屏幕画面是图20-4这样的。

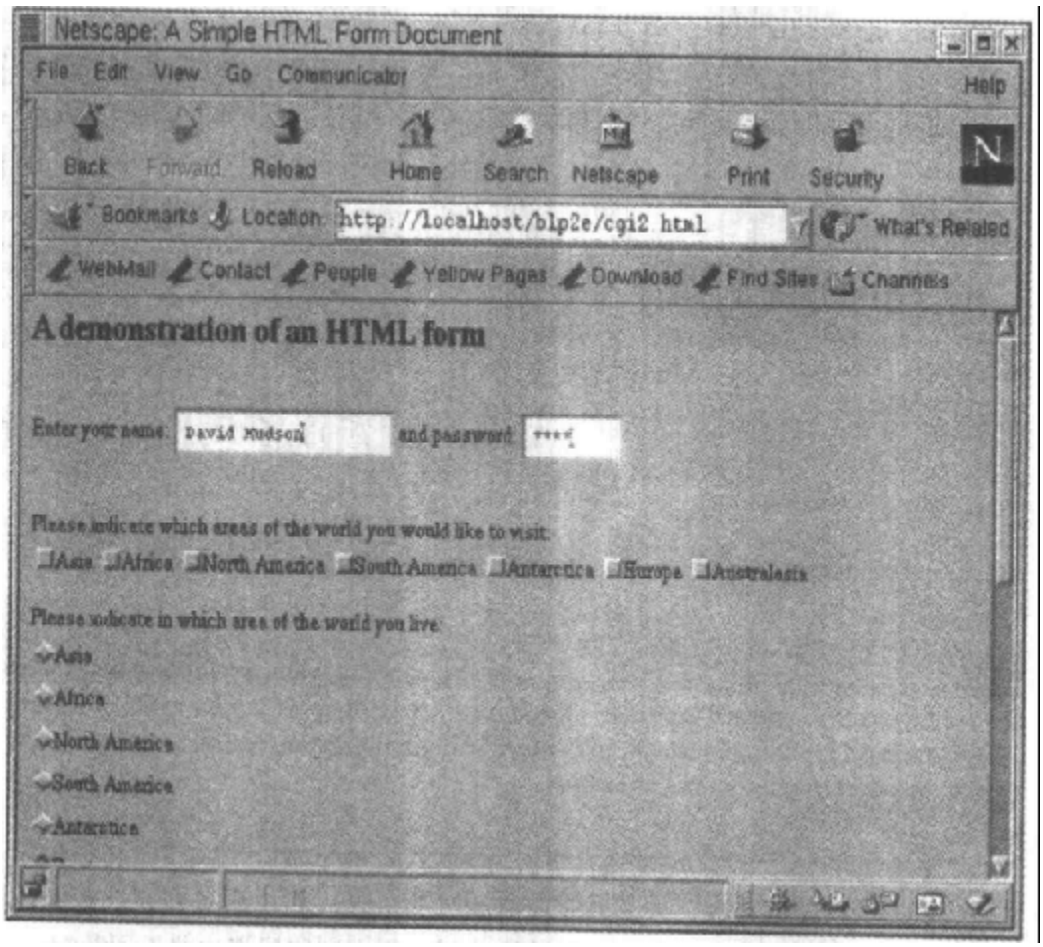


图 20-4

我们提交表单后，浏览器的结果画面是图20-5这样的。

操作注释：

我们现在能够把标准输入流整个地读取下来并把它返回给浏览器了。需要提醒大家注意的

是“简单地读取标准输入直到它停止”并不是一种合乎标准的表单数据访问办法；只不过它确实能够很好的体现我们给出这个例子的用意。

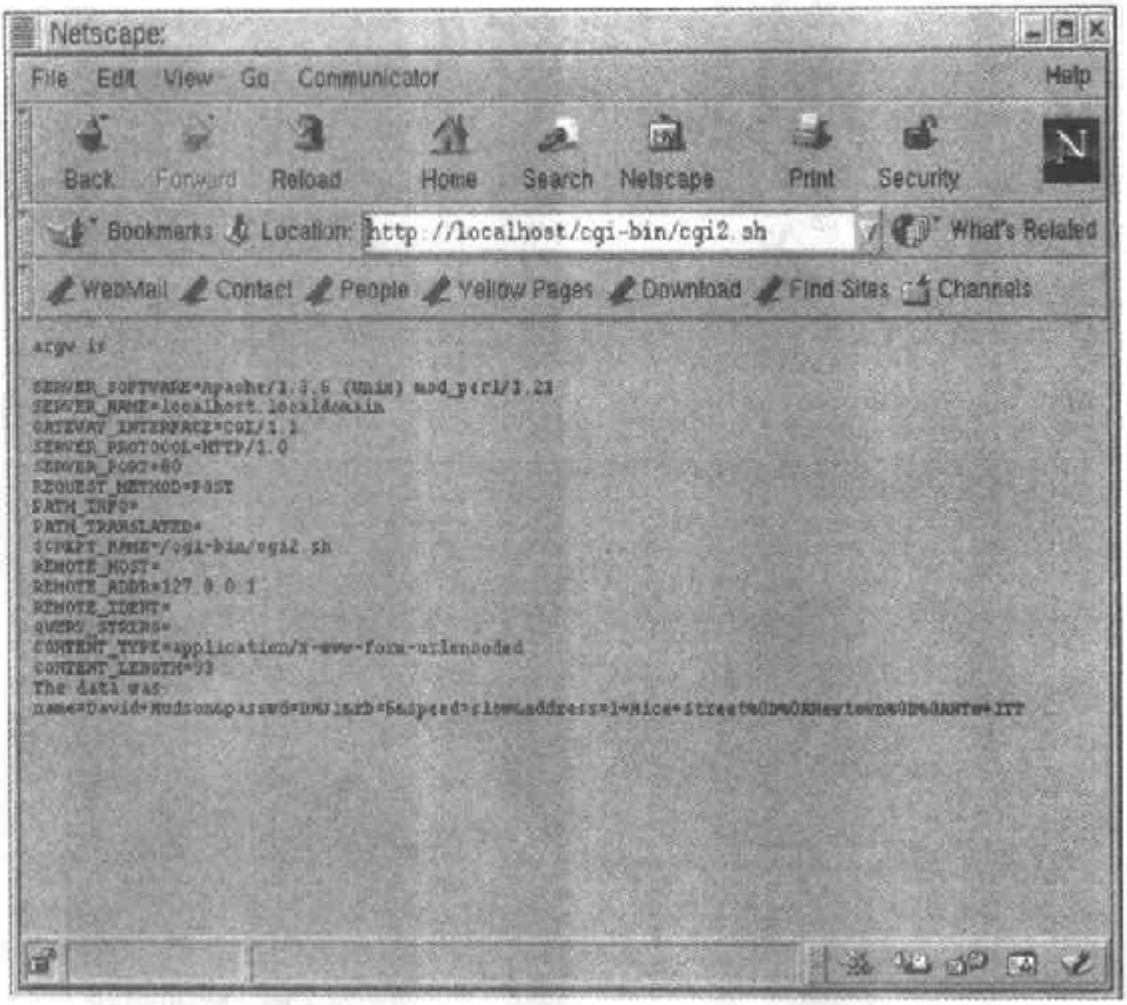


图 20-5

> 在实际程序里，一定要使用CONTENT_LENGTH环境变量来确定将要被处理的数据的长度。CGI技术规范要求必须用这个办法来确定表单数据的长度，这是因为环境变量本身一般很难准确地检测到标准输入上的文件尾标志。

就CGI程序接收到的信息而言，有两件很重要的事情需要大家注意：

- 数据已经经过了编码，所以它现在是一个不间断的不带空白字符的字符串。
- 里面没有与HTML表单上的选择框部分对应的信息。这是因为该部分没有被选中的框，所以浏览器也就省略了与这部分对应的数据。因为这种做法能够减少需要在网上传递的数据的总量，所以它在很多方面都是有积极意义的；关键是表单解码程序必须考虑空白输入域的省略问题。浏览器对没有使用的域是一律省略的。

> 动手试试：使用GET方法

为了让大家对表单有一个完整的认识，我们把HTML示例中的METHOD类型改为GET，它就是我们的cgi3.html文件。如果现在再提交我们的表单，数据传递给cgi2.sh的方式就与刚才稍有不同了，我们现在看到的画面如图20-6所示。

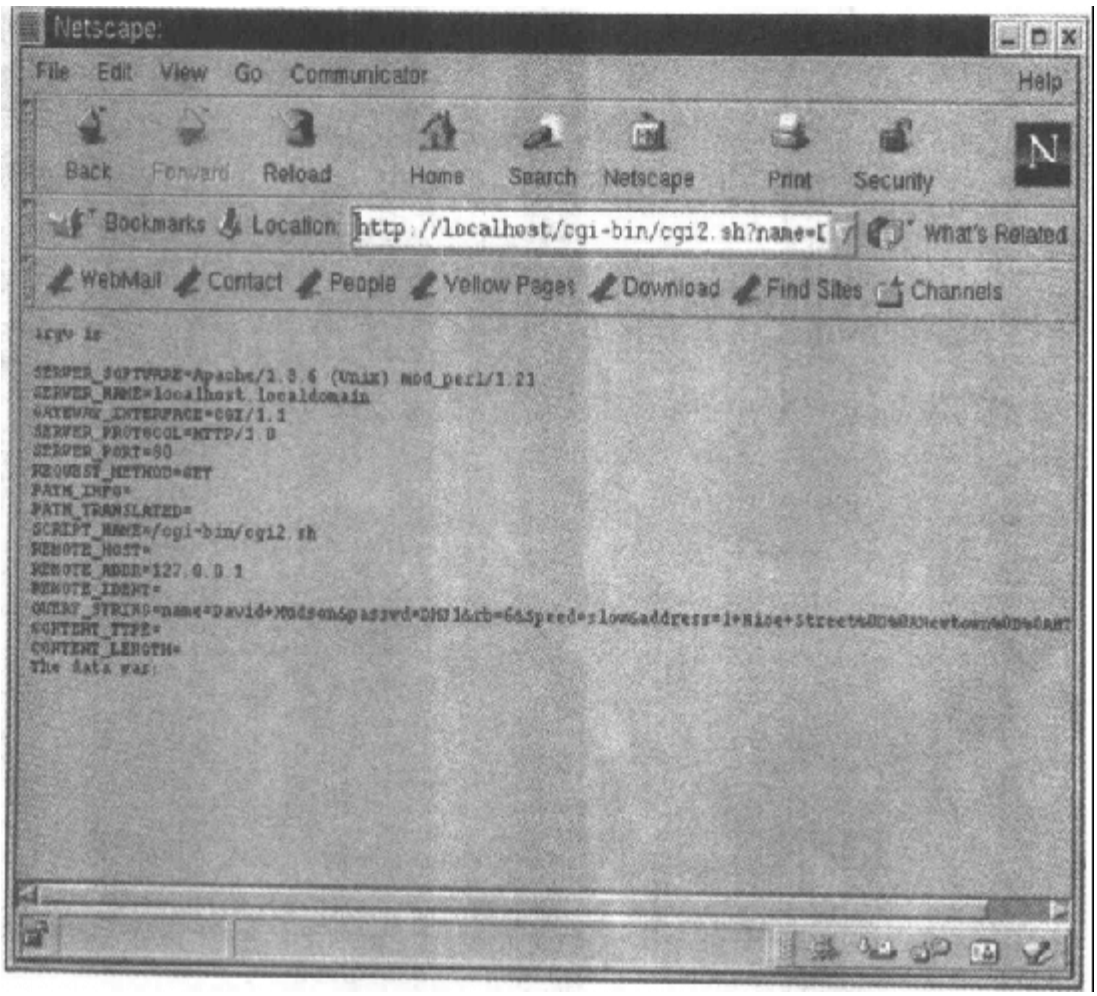


图 20-6

表单信息的编码工作还是照常进行，但CONTENT_LENGTH环境变量不再出现了。这个信息现在出现在QUERY_STRING变量里，不再出现在标准输入上。“METHOD = GET”处理起来比较容易，但“METHOD = POST”适用于除简单表单以外的所有情况。

20.3.4 使用扩展URL的CGI程序

在转向对传递给CGI程序的数据进行解码操作之前，我们先来看看向CGI程序发送数据的其他办法——把信息追加在URL地址的后面。服务器程序分离出URL地址的前半部分（即问号“?”前面的部分），把它用做将被调用的CGI程序的名字；而URL地址的后半部分将整个地用做该程序的一个参数，就好象它是从表单提交来的一样。

动手试试：一个查询字符串

再次调用cgi2.sh，但这次要使用如下所示的URL地址：

```
http://localhost/cgi-bin/cgi2.sh?Andrew+Stones=10
```

CGI程序这次看到的表单数据如图20-7所示。

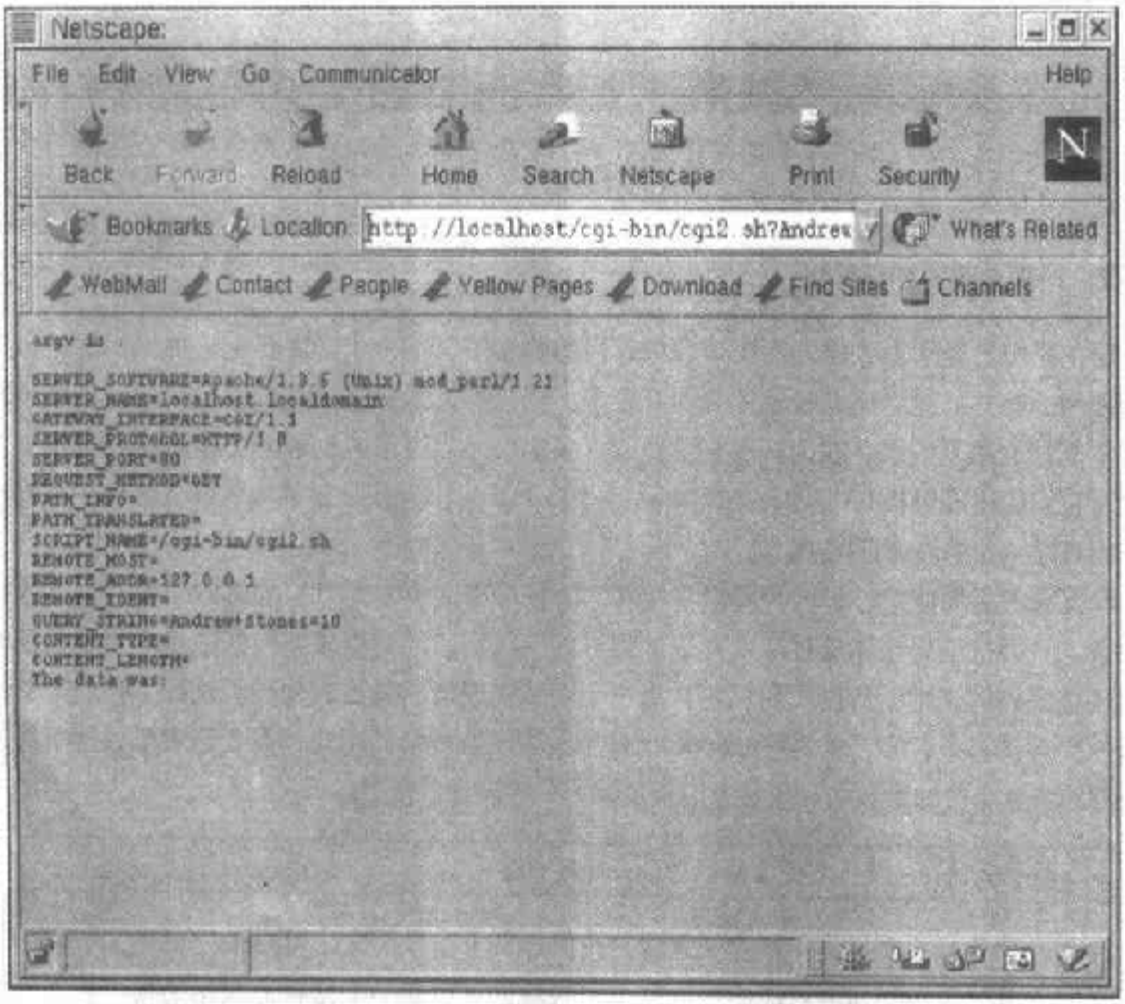


图 20-7

这种向CGI程序传递信息的办法经常用在这样的情况里：服务器已经生成了客户正在浏览的HTML页面，但又想在页面里额外添加一些编码数据。需要用户身份验证的Web主页就是一个这样的例子。因为对某个页面的每次请求在服务器看来都是没有前后联系的，所以，在理论上，用户每访问一个页面就需要输入一次他们的口令字。而实际上可以让用户去访问一个特殊的URL地址，这个URL地址里已经把用户名/口令字或者一个身份验证密钥编码在其中了。

假设有这样一位用户，他的名字是Jenny Stone，口令字是secret（从她选的口令字上看，她可不够聪明...）。她想去访问一个在线报纸的天气预报主页，而主页要求她先注册后访问。这个主页不是向她显示一个需要填写姓名和口令字的表单，而是让她输入一个特殊的URL地址，地址的格式是下面这个样子的：

```
http://www.paper.com/cgi-bin/access?user=Jenny+Stones&passwd=secret&page=weather
```

虽然这看起来有些麻烦，但她可以把它保存为一个浏览器书签或快捷键，再想访问这个

URL地址时只要选一下就行了。以后再访问这个主页时就不必每次都要重新输入用户名和口令字了。而Jenny Stone也不必记住或写下自己以前访问各种在线服务时给出的用户名和口令字了。

这个办法的缺点是口令字是以纯文本形式保存的，而这并不是一个保存或在网上传递用户名和口令字的好方法。在实际应用中，人们经常使用一种名为“cookie”的功能来把某个用户的访问权利保存在一个站点里，这就使用户不必记忆自己的口令字。可这个方法也有缺点，那就是cookie是由浏览器保存在本地机器上的。因此，即使你在自己家里上网注册并在自家的PC上保存了一个cookie，这对你上班时使用另外一台PC去访问那个站点也不会有什么帮助。我们不会在这一章里继续对cookie进行讨论了。

20.3.5 对表单数据进行解码

现在我们已经知道怎样才能获得表单返回的数据了，下一步就需要把它解码为更适合实际处理使用的格式。这个问题涉及的方面有很多，但只要你解决过一次，就可以把这个解码软件一直重复性地用下去了。在因特网上已经有许多能够完成表单数据解码工作的现成程序了，用来编写它们的语言包括Tcl、Perl、C和C++等许多种。它们有些是公共域软件，另外一些则有限制性相对强一些的许可证问题。

但大家不必为此担心，因为我们将用自己的程序来完成表单数据的解码工作。其实这也并不复杂，只要把编码操作反过来就可以达到目的。我们已经知道表单数据的编码规则，把它反过来就是解码的规则，所以我们只要写出一个能够实现解码规则的软件不就行了吗。先给我们这第一次尝试起个名字，就叫它decode1.c好了。为了使事情简单化，需要给表单数据里我们能够处理的参数个数和参数名、值各自的长度加上一些硬性的限制。

动手试试：一个用C语言编写的CGI解码程序

1) 在套路化的头文件和常数定义之后，我们定义了一个数据结构name_value_st，它的作用是保存输入域的名字和与之对应的值，每个输入域都对应着一个这样的数据结构。接下来是我们将要用到的各种函数的框架定义。

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

#define FIELD_LEN 250 /* how long each name or value can be */
#define NV_PAIRS 200  /* how many name=value pairs we can process */

typedef struct name_value_st {
    char name[FIELD_LEN + 1];
    char value[FIELD_LEN + 1];
} name_value;

name_value name_val_pairs[NV_PAIRS];

static int get_input(void);
static void send_error(char *error_text);
static void load_nv_pair(char *tmp_buffer, int nv_entry_number_to_load);
static char x2c(char *what);
static void unescape_url(char *url);
```

2) 主函数main本身做的事情并不多。它先调用get_input函数对表单的编码数据进行处理，

把处理结果保存到name_val_pairs结构里去。然后把以“Content-type”和一个空白行开始的解码数据发送回客户。如下所示：

```
int main(int argc, char *argv[])
{
    int nv_entry_number = 0;

    if (!get_input()) {
        exit(EXIT_FAILURE);
    }

    printf("Content-type: text/plain\r\n");
    printf("\r\n");

    printf("Information decoded was:-\r\n\r\n");
    while(name_val_pairs[nv_entry_number].name[0] != '\0') {
        printf("Name=%s, Value=%s\r\n",
               name_val_pairs[nv_entry_number].name,
               name_val_pairs[nv_entry_number].value);
        nv_entry_number++;
    }
    printf("\r\n");
    exit(EXIT_SUCCESS);
}
```

3) 下面是get_input函数的定义，这个函数的功能就像它名字说的那样是“取得输入”。我们先要把请求类型是POST还是GET确定下来，然后再把数据拷贝到一个名为ip(data的数据块里去。如下所示：

```
static int get_input(void)
{
    int nv_entry_number = 0;
    int got_data = 0;
    char *ip_data = 0;
    int ip_length = 0;
    char tmp_buffer[(FIELD_LEN * 2) + 2];
    int tmp_offset = 0;
    char *tmp_char_ptr;
    int chars_processed = 0;

    tmp_char_ptr = getenv("REQUEST_METHOD");
    if (tmp_char_ptr) {
      if (strcmp(tmp_char_ptr, "POST") == 0) {
                tmp_char_ptr = getenv("CONTENT_LENGTH");
                if (tmp_char_ptr) {
                    ip_length = atoi(tmp_char_ptr);
                    ip_data = malloc(ip_length + 1); /* allow for NULL character */
                    if (fread(ip_data, 1, ip_length, stdin) != ip_length) {
            send_error("Bad read from stdin");
            return(0);
            }
                    ip_data[ip_length] = '\0';
                    got_data = 1;
                }
      }

}

tmp_char_ptr = getenv("REQUEST_METHOD");
if (tmp_char_ptr) {
  if (strcmp(getenv("REQUEST_METHOD"), "GET") == 0) {
    tmp_char_ptr = getenv("QUERY_STRING");
            if (tmp_char_ptr) {
                ip_length = strlen(tmp_char_ptr);
                ip_data = malloc(ip_length + 1); /* allow for NULL character */
                strcpy(ip_data, getenv("QUERY_STRING"));
                ip_data[ip_length] = '\0';
```

```
                got_data = 1;
            }
        }
    }

    if (!got_data) {
        send_error("No data received");
        return(0);
    }

    if (ip_length <= 0) {
        send_error("Input length not > 0");
        return(0);
    }
```

4) 进行到这，我们就已经把编了码的数据保存在ip_data里了。接下来要把HTML提交数据里的NAME和VALUE成双成双地提取出来，对每对数据项分别进行解码。我们知道“&”字符是用来分隔一对一对的“NAME=VALUE”组合的，所以我们就要从出现“&”字符的位置把它们分断，再把结果依次传递到load_nv_pair函数去分别进行解码。

```
    memset(name_val_pairs, '\0', sizeof(name_val_pairs));
    tmp_char_ptr = ip_data;
    while (chars_processed <= ip_length && nv_entry_number < NV_PAIRS) {

        /* copy a single name=value pair to a tmp buffer */
        tmp_offset = 0;
        while (*tmp_char_ptr &&
               *tmp_char_ptr != '&' &&
               tmp_offset < FIELD_LEN) {
            tmp_buffer[tmp_offset] = *tmp_char_ptr;
            tmp_offset++;
            tmp_char_ptr++;
            chars_processed++;
        }
        tmp_buffer[tmp_offset] = '\0';

        /* decode and load the pair */
        load_nv_pair(tmp_buffer, nv_entry_number);

        /* move on to the next name=value pair */
        tmp_char_ptr++;
        nv_entry_number++;
    }
    return(1);
}
```

5) 大家应该注意到我们把所有错误都传递到那个名为send_error的函数去了，它将向客户发回一个错误信息字符串。这个函数没有多复杂，下面就是它的定义：

```
static void send_error(char *error_text)
{
    printf("Content-type: text/plain\r\n");
    printf("\r\n");
    printf("Woops:- %s\r\n", error_text);
}
```

6) 实际解码工作是由函数load_nv_pair负责的，我们在get_input函数里调用的就是它。它先把每对“NAME=VALUE”组合进一步分断为NAME和VALUE两个部分，把它们分别放到我们数据结构的不同单元里去；然后调用另一个名为unescape_url的函数继续解码。

```
/* Assumes name_val_pairs array is currently full of NULL characters */
static void load_nv_pair(char *tmp_buffer, int nv_entry)
```

```
{
    int chars_processed = 0;
    char *src_char_ptr;
    char *dest_char_ptr;

    /* get the part before the '=' sign */
    src_char_ptr = tmp_buffer;
    dest_char_ptr = name_val_pairs[nv_entry].name;
    while(*src_char_ptr &&
          *src_char_ptr != '=' &&
          chars_processed < FIELD_LEN) {

        /* Change a '+' to a ' ' */
        if (*src_char_ptr == '+') *dest_char_ptr = ' ';
        else *dest_char_ptr = *src_char_ptr;
        dest_char_ptr++;
        src_char_ptr++;
        chars_processed++;
    }

    /* skip the '=' character */
    if (*src_char_ptr == '=') {

        /* get the part after the '=' sign */
        src_char_ptr++;
        dest_char_ptr = name_val_pairs[nv_entry].value;
        chars_processed = 0;
        while(*src_char_ptr &&
              *src_char_ptr != '=' &&
              chars_processed < FIELD_LEN) {
            /* Change a '+' to a ' ' */
            if (*src_char_ptr == '+') *dest_char_ptr = ' ';
            else *dest_char_ptr = *src_char_ptr;
            dest_char_ptr++;
            src_char_ptr++;
            chars_processed++;
        }
    }

    /* Now need to decode %XX characters from the two fields */
    unescape_url(name_val_pairs[nv_entry].name);
    unescape_url(name_val_pairs[nv_entry].value);
}
```

7) unescape_url函数再调用函数x2c把（不是字母或数字的）特殊字符从其%HH表示方式解码为文本字符。如下所示：

```
/* this routine borrowed from the examples that come with the NCSA server */
static void unescape_url(char *url)
{
    int x,y;
    for (x=0,y=0; url[y]; ++x,++y) {
        if ((url[x] = url[y]) == '%') {
            url[x] = x2c(&url[y+1]);
            y += 2;
        }
    }
    url[x] = '\0';
}

/* this routine borrowed from the examples that come with the NCSA server */
static char x2c(char *what)
{
    register char digit;
    digit = (what[0] >= 'A' ? ((what[0] & 0xdf) - 'A')+10 : (what[0] - '0'));
    digit *= 16;
    digit += (what[1] >= 'A' ? ((what[1] & 0xdf) - 'A')+10 : (what[1] - '0'));
    return(digit);
}
```

我们对这个程序进行编译，然后把它拷贝到cgi-bin子目录。接下来修改我们的HTML表单，使表单的ACTION属性指向/cgi-bin/decode1，METHOD属性为POST。新HTML文档的名字是cgi5.html。

图20-8是我们准备提交的表单信息

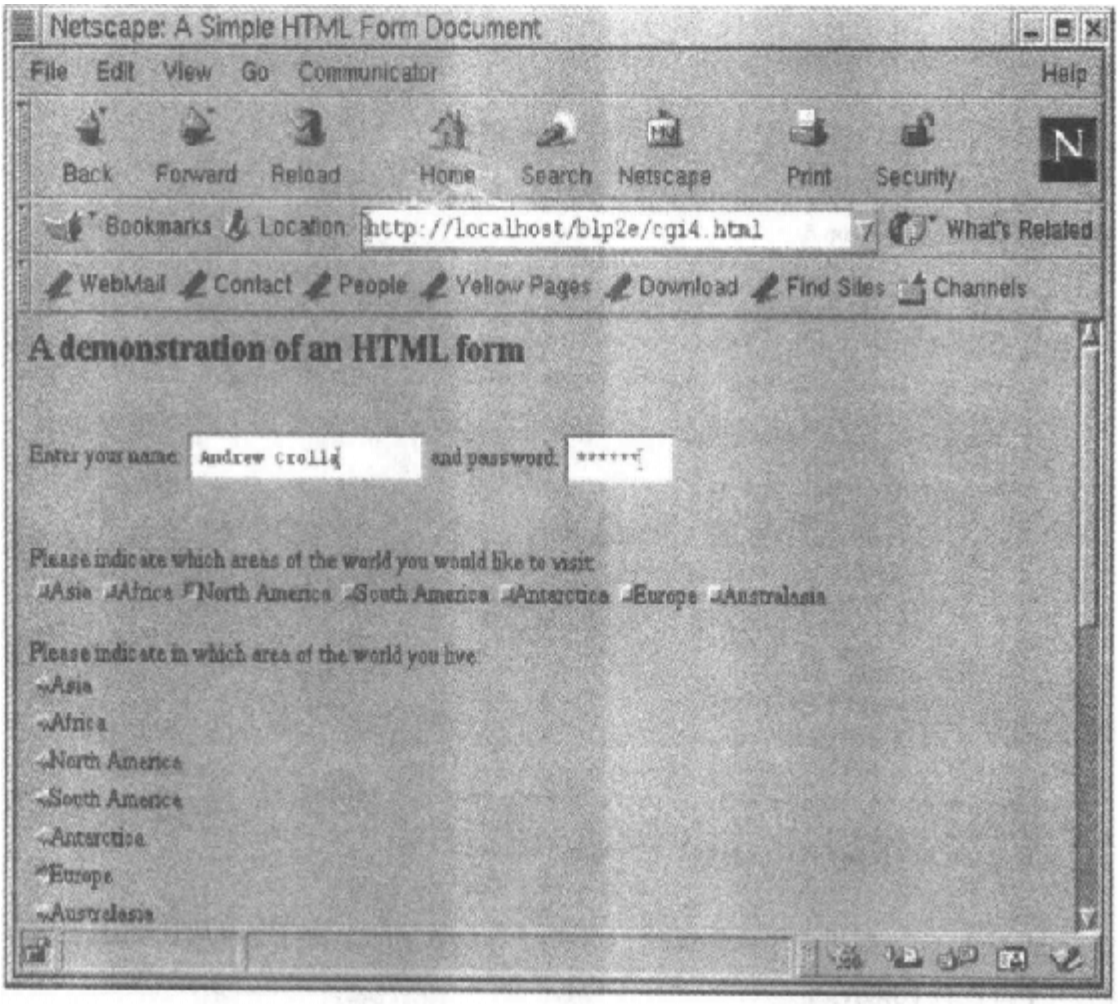


图 20-8

提交表单之后，我们将看到如图20-9所示的浏览器画面

大家可以看到，表单数据中的换行符都被保留下来了。

操作注释：

这些C语言代码可够多的，我们把它分为几个小段落进行说明。我们在程序的一开始先定义了一个用来成对儿保存一个名/值组合的结构，然后定义了一个数组来保存这些结构。我们对数据长度做了武断地限制，这可以使这个程序示例实现起来比较简单，也比较容易理解。一个实用性程序可能会利用一个结点链接列表来做这件事，每个结点保存一个指针，指针指向用malloc函数分配到的一个内存区域，而名/值对就保存在这些个内存区域里——这就可以取消数据长度方面的限制了。

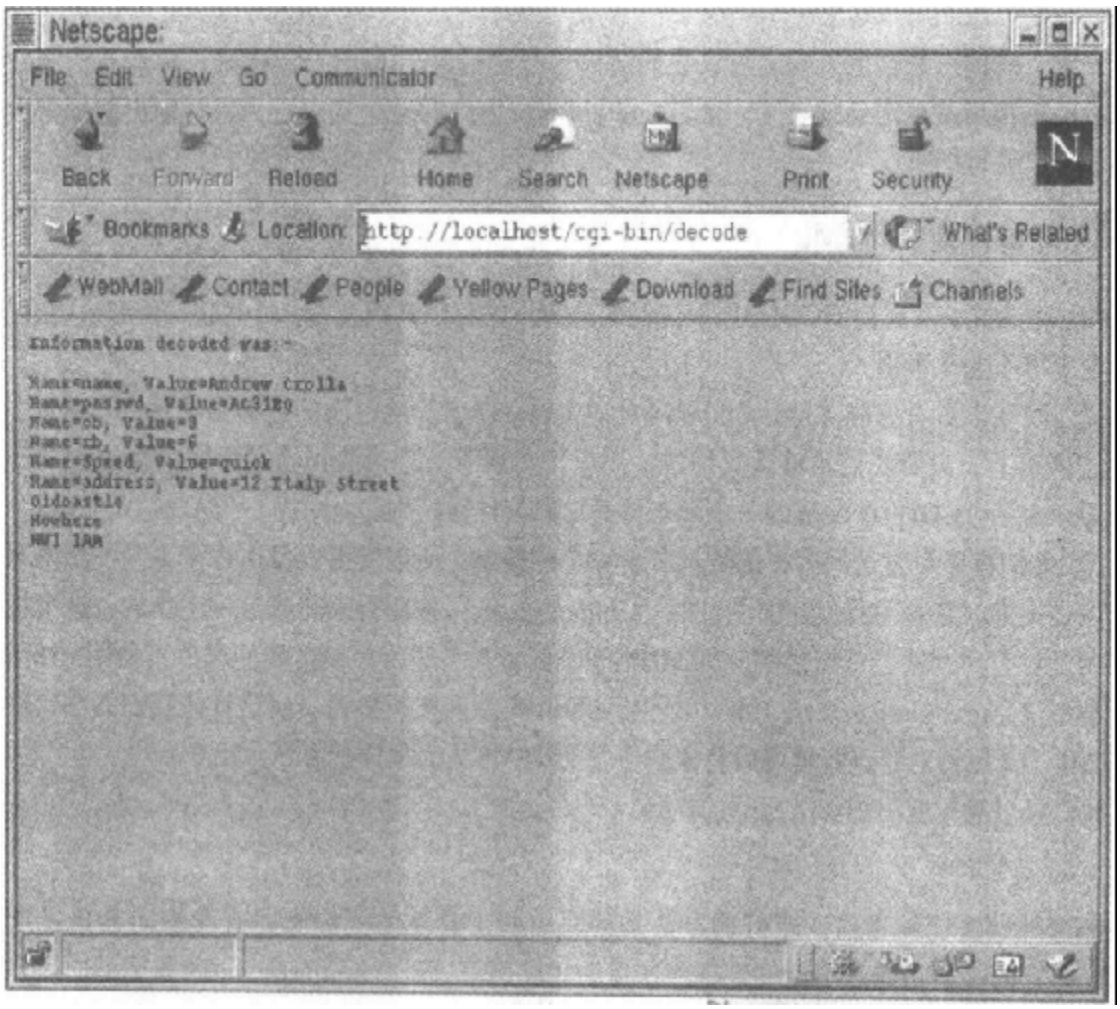


图 20-9

接下来我们调用get_input例程来加载name_val_pairs数组，整个操作需要好几个步骤才能完成。首先，我们要检查数据是采用哪种方法（POST或GET）传递过来的，再把这些数据传到由ip(data指向的字符串里去。此时，不管是用POST方法还是用GET方法传递来的数据都将被保存在同一个地方，而我们也就可以用同样的代码来处理这两种情况了。

第一阶段是对这一整个字符串进行分断，使我们今后能够对“name=value”形式的各个子字符串逐个进行处理。注意这些字符串都是以“&”字符隔开的。接下来我们对每对名/值组合调用一次load_nv_pair，它的作用是把成对的数据项从等号“=”字符处分断开。它还会把加号“+”字符替换为空格字符。

把名字和值分为两个字符串之后，我们再替换其中的特殊字符，这些特殊字符在数据里是以两个十六进制字节“%XX”的形式出现的。我们用x2c和unescape_url两个函数来完成这一工作。

x2c和unescape_url这两个例程在所有以C和C++编写的解码软件里几乎都会出现（甚至连内容都一模一样）。就我们知道的情况来说，它们最早出现在随NCSA WWW服务器软件提供的程序示例中。我们既没见过它们的版权资料，也没听说过有什么人声称拥有它。因为它们的传播范围是那么的广，并且确实能解决问题，所以我们信得过NCSA服务器软件里的这个小东西的准

确性，把它照搬照用在这里。

完成了对输入数据的处理之后，我们简单地把得到的名/值对打印出来。注意我们象往常一样在自己的输出数据之前加上了一个“Content-type”行和一个空白行，每行输出数据都以一个回车和一个换行结束。

20.4 向客户返回HTML

到目前为止，从我们的CGI程序发送到客户那里的都是些纯文本。虽然它工作起来没有问题，但看上去可不怎么吸引人。

我们现在来看看怎样才能让CGI程序自动生成HTML，使自己的输出更象是一个普通的Web页面。事实上，只要我们足够认真细致，用户根本就没有理由看出由某个服务器返回的一个静态页面和由某个CGI程序生成的一个动态页面之间有什么区别。

因为CGI程序对发送回客户的数据有着完全的控制，所以它可以发送许多不同类型的数据而不仅仅是文本。它是用我们前面见过的“Content-type:”控制行来完成这一工作的，这个控制行的作用是对客户浏览器预期接收的MIME类型及其子类型进行控制。我们可以不使用MIME类型及子类型信息text/plain来发送文本，利用text/html类型同样行得通。而这将使我们能够向客户发送HTML，浏览器对它进行解码后将象对待普通HTML页面那样把它显示出来。

我们可以简单地写出如下所示的代码：

```
printf("<H1>A heading</H1>\r\n");
```

但这种做法比较粗糙。更好的办法是编写一些工具性的函数来完成我们向客户发送各种HTML标签的底层功能。下面就是一些这样的底层例程。函数html(content的作用是向客户发送一个字符串，通知它我们将向它发送HTML，如下所示：

```
static void html_content(void)
{
    printf("Content-type: text/html\r\n\r\n");
}
```

html_start函数的作用是开始发送HTML的文档标题部分，它会给客户发送去一个标题字符串并开始HTML的文档体部分，如下所示：

```
static void html_start(const char *title)
{
    printf("<HTML>\r\n");
    printf("<HEAD>\r\n");
    printf("<TITLE>%s</TITLE>\r\n", title);
    printf("</HEAD>\r\n");
    printf("<BODY>\r\n");
}
```

我们还可以再编写两个便利的工具性函数，它们一个是负责按指定级别输出段落标题文字的html(header，另一个是负责输出普通段落文本的html_text，下面是这两个函数的定义情况：

```
static void html_header(int level, const char *header_text)
{
    if (level < 1 || level > 6) level = 6; /* force the level to a valid number */
    if (!header_text) return;
    printf("<H%d>%s</H%d>\r\n", level, header_text, level);
```

```
}

static void html_text(const char *text)
{
    printf("%s\r\n", text);
}
```

最后是用来结束HTML页面的html_end函数。如下所示：

```
static void html_end(void)
{
    printf("</BODY>\r\n");
    printf("</HTML>\r\n");
}
```

注意我们输出的每一行都必须以一个回车和一个换行这两个字符结尾，这与UNIX常用的以一个换行符来结束文本行的做法是不一样的。一个最小化的CGI程序可以只由这些例程组成，简单到只有下面这几行语句：

```
html_content();
html_start("Example");
html_header(1, "Hello World");
html_text("Pleased to be here!");
html_end();
```

只要在我们的decode1.c程序后面加上刚才定义的这五个函数和一些其他的修改，我们就可以让它向客户发送HTML信息而不是纯文本了。我们把新程序叫做decode2.c。

动手试试：向客户返回HTML

1) 先在程序的开始部分把我们将在后面使用的那几个函数的预定义添上。它们可以参照我们刚才定义的那五个例程来编写。如下所示：

```
static int get_input(void);
static void send_error(char *error_text);
static void load_nv_pair(char *tmp_buffer, int nv_entry_number_to_load);
static char x2c(char *what);
static void unescape_url(char *url);
static void html_content(void);
static void html_start(const char *title);
static void html_end(void);
static void html_header(int level, const char *header_text);
static void html_text(const char *text);
```

2) 大多数改动都发生在main函数里，因为绝大部分的输出都是由它来承担的。从本质上看，这只是个定义文本缓冲区的活儿——在把数据以HTML格式送往客户去显示之前，它们都将被保存在这个缓冲区里。再就是修改一下输出命令，让它通过我们的新函数来把数据送往客户。如下所示：

```
int main(int argc, char *argv[])
{
    char tmp_buffer[4096];
    int nv_entry_number = 0;

    if (!get_input()) {
            exit(EXIT_FAILURE);
    }

    html_content();
```

20.5 技巧与窍门

下面是一些对大家编写CGI程序有帮助的提示和技巧

20.5.1 确保CGI程序能够退出

在CGI程序执行的过程中或者在它向客户发送响应数据的时候，可能出现客户已经下网或网络出现故障的情况。如果真的发生了这类的事情，程序就可能陷入等待向标准输出发送输出数据但又永远无法完成的困境。为了避免这种问题的发生，确保程序有个万无一失的退出机制是十分明智和非常必要的。一个比较好的解决办法是在程序启动时预先设置一个SIGALRM信号陷阱；然后，在开始进行输出之前，调用alarm在比如说30秒之后产生一个SIGALRM信号。CGI程序本身运行的时间一般是比较短的，通常不超过一秒。因此，如果这个信号的处理器被调用了的话，就说明出现了比较严重的情况，这很有可能就是因为输出被阻塞而造成的，所以这个时候最好是让CGI程序退出执行而不是尝试去向客户发送错误信息。如果系统允许，我们也可以采用另外一个解决方案，即使用非阻塞I/O以避免空等现象的发生。

20.5.2 对客户进行重定向

另外一个经常用在CGI程序里的技巧是根据被传递信息把请求重定向到另外一个不同的Web页面去。我们可以利用向客户发送状态信息的办法做到这一点。状态信息有许多种，但对CGI程序有用的却并不很多。我们在此只考虑状态302，它的含义是通知客户页面暂时转移了。

假设我们有一个页面向客户提问他们是喜欢爵士音乐还是喜欢古典音乐，然后，根据客户对这一问题的回答把他们引导到不同的主页去。如果我们想在CGI程序里即时生成结果页面，就会在是否需要在程序里生成HTML页面的问题上面进退两难，这就不如设法去直接利用HTML文件来得好。

让客户转移到一个不同的主页去是一个简单许多的解决方案。假设我们在某个CGI程序里对表单信息进行解码，结果是得到了一个取值为jazz（爵士）或classical（古典）的变量。我们就可以用下面这样的代码把客户引导到相应的主页：

```
printf("Status: 302\r\n");
if (strcmp(result, "jazz") == 0) {
    printf("Location: /jazz.html\r\n");
}
else {
    printf("Location: /class.html\r\n");
}
exit(EXIT_SUCCESS);
```

注意这是在这种情况下我们将要发送的第一条信息，所以不需要在它前面再加上“Content-type”之类的东西。当客户程序看到“Status 302”时，它就会认为自己请求的URL已经临时转移到另外一个新地点去了。它会检查随后的“Location:”行以确定主页转移到什么地方去了。当“Location:”行到达时，客户就会简单地转向该行指定的URL。

20.5.3 动态图形

许多WWW主页都有一个访问或点击计数器。这些计数器一般都显示在由普通HTML元素构

成的页面上，可这些会随时刷新的图形化访问计数器是如何跑到页面上去的呢?

答案通常是这样的：使用一个<IMG>标签，但把它的SRC（正常情况下多是.gif或.jpg文件）属性设置为一个cgi-bin程序。当客户访问基础主页的时候，它就会因看到了<IMG>标签而尝试取回一个指定的图像。如果cgi-bin程序能够即时生成图像，就可以返回一个动态刷新的图像。

对这一问题的详细讨论超出了本章的范围，但如果读者想在这一方面继续钻研的话，请设法找到一个名为gd的的图形库，它可以帮助你从代码里生成图像。

20.5.4 隐藏上下文信息

在很多情况下，服务器需要在前后表单之间传递一些信息。这可能是一个用户名、一个顾客编号，或者是一些其他的信息。

可表单们完全是分开处理的，因此，我们需要在表单上“暗藏”一些信息。我们可以用在这一章的前面遇到的隐藏域来做这件事情，或者可以把信息添加到URL地址上的问号“?”字符后面。

当一个利用CGI程序生成的表单被发送给客户的时候，我们可以在其上定义一个或多个隐藏域并给它们设置上缺省的值。通过这个办法，我们就可以把信息隐藏起来，让用户暂时看不到它们。当用户填写好表单并把它发送回服务器的时候，CGI程序就可以使用隐藏域中的数据值提取出上下文信息来。

注意这些域只是隐藏起来了，并不是秘不可见的。用户还是可以通过查看HTML文档源文件的方法看到这些东西。隐藏域绝不是隐藏秘密口令字的好地方!

追加在URL地址上问号“?”后面的数据被当做是用户利用“METHOD = GET”方法提交的表单信息。CGI程序可以对这些附加的信息进行解码并把它用做上下文信息。我们将在下面的应用程序示例里演示这种做法。

20.6 一个应用程序

这是大家一直期盼的东西：我们将看到怎样才能把我们的数据库放到网上去访问。在这个例子里，我们将只实现对数据库的读访问。这使我们既能够向大家展示通过一个CGI程序来访问数据库的基本原理，也又能够使我们的应用程序保持一定的简单性。如果愿意，大家可以自行扩展这个示例，使数据库也能被修改。

我们将以我们在第7章里开发的使用了dbm数据库的应用程序为基础，用一个全新的操作前端app(html.c来代替app(ui.c文件。我们将用到在这一章前面写的许多函数。

动手试试：一个HTML数据库接口

1) 下面是app_html.c程序的完整清单。main函数启动了数据库，然后根据URL命令行上是否给出了一个CD唱盘分类编号调用另外两个其他的函数。如下所示：

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

#include "cd data.h"
```

```
#include "html.h"

const char *title = "HTML CD Database";
const char *req_one_entry = "CAT";
void process_no_entry(void);
void process_cat(const char *option, const char *title);
void space_to_plus(char *str);

int main(int argc, char *argv[])
{
    if (!database_initialise(0)) {
        html_content();
        html_start(title);
        html_text("Sorry, database could not initialize");
        html_text("<BR><BR>");
        html_text("Please mail <AHREF=\"mailto:webmaster@anyhost.com\">
                   webmaster</A> for assistance");
        html_end();
        exit(EXIT_SUCCESS);
    }

    if (!get_input()) {
        database_close();
        html_content();
        html_start(title);
        html_text("Sorry, METHOD not POST or GET");
        html_text("Please mail <AHREF=\"mailto:webmaster@anyhost.com\">
                   webmaster</A> for assistance");
        html_end();
        exit(EXIT_SUCCESS);
    }

    html_content();
    html_start(title);
```

2) 如果是一个合法的查询字符串，就显示该CD唱盘上的曲目。否则就列出数据库里的全部CD唱盘来。如下所示：

```
    if (strcmp(name_val_pairs[0].name, req_one_entry) == 0) {
        process_cat(name_val_pairs[0].name, name_val_pairs[0].value);
    }
    else {
        process_no_entry();
    }
    html_end();
    database_close();
    exit(EXIT_SUCCESS);
}
```

3) 如果我们到达这里，就说明用户一个项目都没有选。所以将显示一个标准的画面，把数据库里的CD唱盘全都列出来。如下所示：

```
void process_no_entry(void)
{
    char tmp_buffer[120];
    char tmp2_buffer[120];
    cdc_entry item_found;
    int first_call = 1;
    int items_found = 1;

    html_header(1, "CD database listing");
```

```
        html_text("Select a title to show the tracks");
        html_text("<BR><BR><HR><BR><BR>");
        while(items_found) {
            item_found = search_cdc_entry("", &first_call);
            if (item_found.catalog[0] == '\0') {
                items_found = 0;
            }
          else {
              sprintf(tmp_buffer, "Catalog: %s", item_found.catalog);
              html_text(tmp_buffer);
              html_text("<BR><BR>");
              strcpy(tmp2_buffer, item_found.catalog);
              space_to_plus(tmp2_buffer);
              sprintf(tmp_buffer, "Title: <A HREF=\"/cgi-bin/cddb/cdhtml?CAT=%s\">
                                   %s</A>", tmp2_buffer, item_found.title);
              html text(tmp buffer);
              html_text("<BR><BR>");
              sprintf(tmp_buffer, "Type: %s", item_found.type);
              html_text(tmp_buffer);
              html_text("<BR><BR>");
              sprintf(tmp_buffer, "Artist: %s", item_found.artist);
              html_text(tmp_buffer);
              html_text("<BR><BR><HR><BR><BR>");
          }
      }
}
```

4) 如果我们到达这里，就说明附加参数都已经被分析完了。如下所示：

```
void process_cat(const char *name_type, const char *cat_title)
{
    char tmp_buffer[120];
    cdc_entry cdc_item_found;
    cdt_entry cdt_item_found;
    int first = 1;
    int track_no = 1;

    if (strcmp(name_type, req_one_entry) == 0) {
```

5) 下面这些代码将显示一个里面只有一个项目的窗口，该项目是由cat_title设定的。如下所示：

```
html_header(1, "CD catalog entry");
html_text("<BR><BR>");
html_text("Return to: <A HREF=\"/cgi-bin/cddb/cdhtml\">list</A>");
html_text("<BR><BR>");
html_text("<HR>");

cdc_item_found = search_cdc_entry(cat_title, &first);
if (cdc_item_found.catalog[0] == '\0') {
    html_text("Sorry, couldn't find item ");
    html_text(cat_title);
}
else {
    sprintf(tmp_buffer, "Catalog: %s", cdc_item_found.catalog);
    html_text(tmp_buffer);
    html_text("<BR><BR>");
```

```
                sprintf(tmp_buffer, "Title: %s", cdc_item_found.title);
                html_text(tmp_buffer);
                html_text("<BR><BR>");
                sprintf(tmp_buffer, "Type: %s", cdc_item_found.type );
                html_text(tmp_buffer);
                html_text("<BR><BR>");
                sprintf(tmp_buffer, "Artist: %s", cdc_item_found.artist);
                html_text(tmp_buffer);
                html_text("<BR><BR>");
                html_text("<OL>");
                cdt_item_found = get_cdt_entry(cdc_item_found.catalog, track_no);
                while(cdt_item_found.catalog[0] != '\0') {
                    sprintf(tmp_buffer, "<LI> %s", cdt_item_found.track_txt);
                    html_text(tmp_buffer);
                    track_no++;
                    cdt_item_found = get_cdt_entry(cdc_item_found.catalog,track_no);
                }
                html_text("</OL>");
                html_text("<HR>");
            }
        }
    }
```

6) 下面这个简单的函数就像它名字里说的那样接受一个字符串并把其中的所有空格字符替换为加号“+”字符。如下所示：

```
void space_to_plus(char *str)
{
    while (*str) {
        if (*str == ' ') *str = '+';
        str++;
    }
}
```

操作注释：

第一部分处理是检查能否对数据库进行初始化。如果它失败了，我们将输出一些简单的HTML让用户能够向网络管理员发邮件报告问题。接下来我们将检查是否有一个查询字符串，它第一个成分的名字是否是req_one_entry（它应该被设置为字符串CAT）。如果有，我们就调用process_cat打印出该项目曲目来。否则，我们将调用process_no_entry列出数据库里全部现有的CD唱盘来。

一张指定CD唱盘上的曲目资料是由process_cat函数给出来的，它稍微简单一点，所以我们先来看看它。首先，我们送出一些HTML增加一个锚点，用户点击这个锚点就可以取回CD唱盘的目录清单。接下来我们调用在第7章里开发的例程search_cdc_entry检索出那张CD唱盘的资料并再发送一些用来显示它们的HTML给用户。在文档头信息之后，我们调用函数search_cdt_entry扫描出它上面的曲目来。我们通过HTML生成一个有序列表把曲目显示出来，有序列表会自动把数据项排好序。

如果用户没有选定CD唱盘，就将调用process_no_entry函数，它将在一个标准屏幕里把数据库里的全部CD唱盘列出来。这个函数在搜索数据库时使用的是一个空白字符串，这就可以把全部的CD唱盘都找出来。然后，我们把它们用HTML显示出来。

我们比较得意的代码是下面这几行：

```
strcpy(tmp2_buffer, item_found.catalog);
space_to_plus(tmp2_buffer);
sprintf(tmp_buffer, "Title: <A HREF=\"/cgi-bin/cddb/cdhtml?CAT=%s\">%s</A>",
tmp2_buffer, item_found.title);
```

这几行代码建立了CD唱盘标题目录项的一个拷贝，但已经把所有的空格都转换为加号“+”了——这是对表单数据进行x-www-form-encoded方式编码的标准规则所要求的。我们接着生成了一个同时包含CD唱盘名称（它将被显示在屏幕上）和编码CD唱盘名称的锚点行；这样选择这个链接就将调用程序cgi-bin/cddb/cdhtml?CAT=CDA66374了。这等于是在调用程序cdhtml的时候在环境变量QUERY_STRING里传递了CAT=CDA66374。程序可以检测出这件事，对它进行解码，再用它来选择proecss_cat函数。此时表单数据项的名字被设置为CAT，它的值被设置为CDA66374，它将被用来查找和显示曲目资料。

为了使这个应用程序更加完整，我们还需要多做一些工作。首先，我们需要对所有出现在app_html.c文件里的函数进行定义。我们建立一个名为html.c的文件，文件的开始部分是下面这样的：

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

#include "html.h"

name_value name_val_pairs[NV_PAIRS];
```

然后我们把在decode3.c程序里定义的全部函数（当然不包括main在内）都添加到这个头文件里。接下来我们需要编写出在源代码文件里包括上的两个头文件html.h和cd_data.h，以及另外一个源代码文件cd_access.c。

html.c文件的内容是下面这样的：

```
#define FIELD_LEN 250 /* how long can each name or value be */
#define NV_PAIRS 200 /* how many name=value pairs can we process */

/* This structure can hold one field name and one value of that field  */
typedef struct name_value_st {
    char name[FIELD_LEN + 1];
    char value[FIELD_LEN + 1];
} name_value;

extern name_value name_val_pairs[NV_PAIRS];


int get_input(void);
void send_error(char *error_text);
void load_nv_pair(char *tmp_buffer, int nv_entry_number_to_load);
char x2c(char *what);
void unescape_url(char *url);

void html_content(void);
void html_start(const char *title);
void html_end(void);
void html_header(int level, const char *header_text);
void html_text(const char *text);
```

另外剩下的两个文件就不必这么费事了，它们是代码再使用的光辉典范——它们和我们在第7章里使用的同名文件中的程序内容是一模一样的。我们使用了一个短小的制作文件makefile

把所有这些东西组织在一起，如下所示：

```
all:    cdhtml


.c.o:
        gcc -g -c $?

html.o: html.c html.h
        gcc -g -c html.c

app_html.o: app_html.c cd_data.h html.h
        gcc -g -c app_html.c

cd_access.o: cd_access.c cd_data.h
        gcc -I/usr/include/db1 -g -c cd_access.c

cdhtml: app_html.o cd_access.o html.o
        gcc -o cdhtml -pedantic -g app_html.o cd_access.o \
html.o -ldb

install: cdhtml
        -echo Depending on your setup, you need to do something like...
        -echo cp cdhtml /usr/local/apache/cgi-bin/cddb
        -echo cp cdc_data.db /usr/local/apache/cgi-bin/cddb
        -echo cp cdt_data.db /usr/local/apache/cgi-bin/cddb
```

在用make命令对程序进行了编译之后，我们需要把它移到cgi-bin子目录里去（注意别忘了沿用自这个应用程序以前版本的数据库文件cdc_data！）。我们可以通过URL地址http：local/cgi-bin/cddb/cdhtml来访问它。

这将给我们一个像图20-11这样的CD唱盘清单。

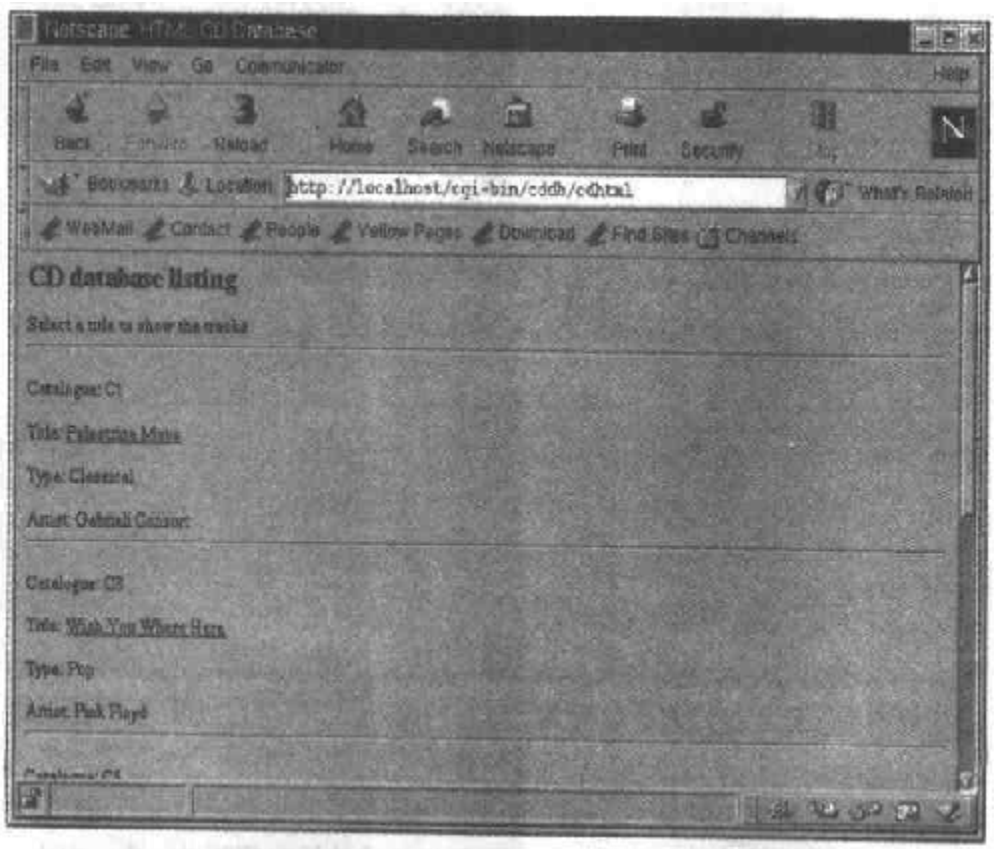


图 20-11

如果这时点击一个高亮度的链接，我们将要访问的还是同一个可执行文件，但这次是在URL的尾部增加了一个查询字符串。我们将会看到中选CD唱盘上的曲目清单。如图20-12所示。

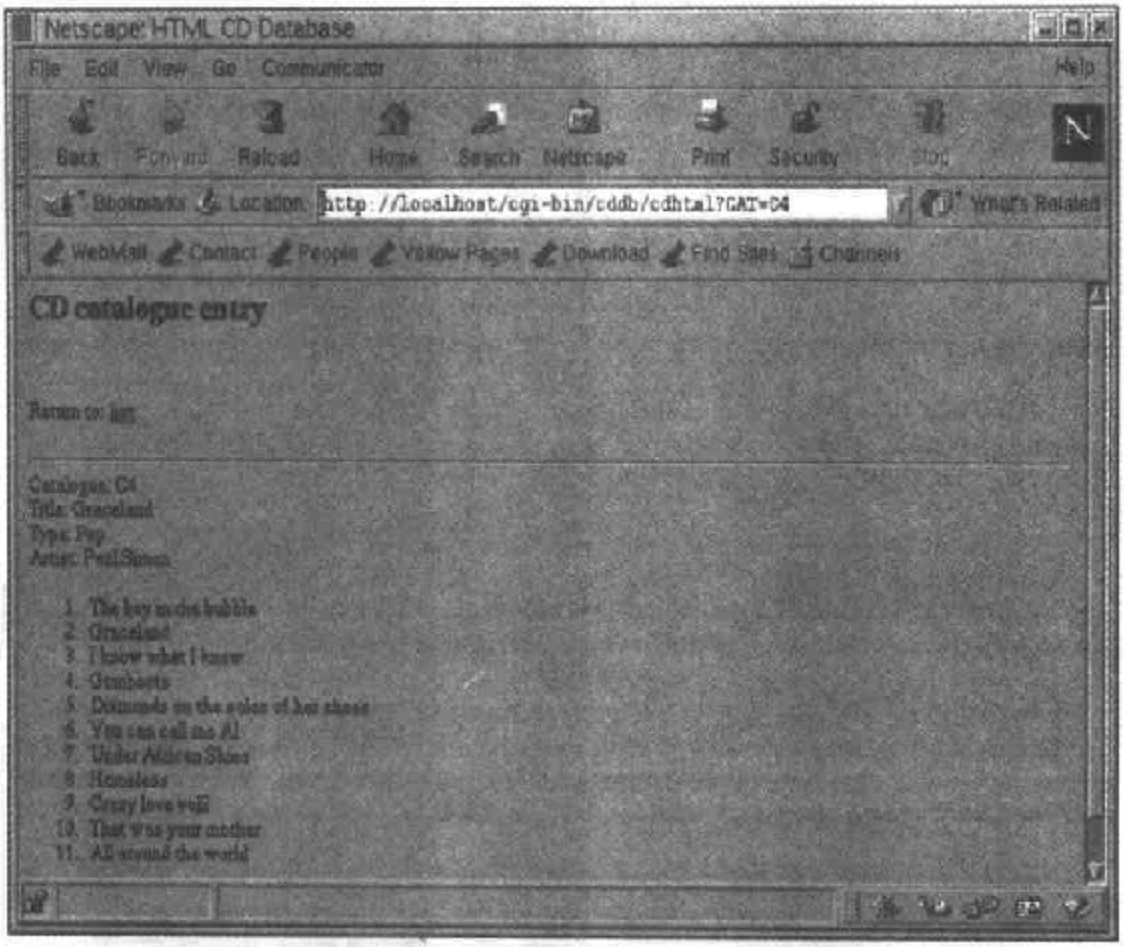


图 20-12

> 你需要把刚才拷贝到cgi-bin/cddb子目录里去的cdc_和cdt_系列文件的读/写权限分配给系统中的每一个人。虽然cdhtml是只读性的，但打开数据库的cd_access例程需要以读写方式打开这文件。

如果读者不想自己把字符都一个一个地敲进去，请记住我们的提醒：这个程序和本书使用的其他所有程序示例的源代码定义都可以从Wrox出版社的Web站点上找到。

20.7 应用Perl语言

一本介绍Linux操作系统上Web服务器Apache软件程序设计方面内容的书如果没有提到Perl语言，那它就算不上完整。虽然为了保持整书格调的统一我们基本上一直使用着C语言，但Perl却可能是人们在Liux或UNIX系统上编写CGI程序时最经常使用的程序设计语言。

Perl语言里提供了许多与编写CGI程序有关的Perl模块，其中最醒目的要说CGI.pm模块了，它能够为你完成大部分表单信息的分析工作。但我们还将继续向前跃进一步，向大家介绍一个

更高级的论题：强大的mod_perl功能模块。它可以有效地把Perl解释器嵌入到Web服务器Apache软件里，从而允许Apache在处理需要执行Perl脚本的请求时不再需要为CGI程序的每次调用都启动一个Perl解释器。

除此之外，我们还将能够访问到某些Apache的内部状态信息，这可以帮助我们开发出生成机制更为复杂的Web主页来；而且，我们以后就不必在每次对Perl脚本进行了修改之后都必须重建Web服务器了。有了这个功能模块，你可以随时对用Perl语言编写的CGI程序进行修改，然后在Apache服务器空闲下来时重新启动它并重新加载你自己的Perl脚本就行了。

但你也必须为此付出一点代价。首先，利用mod_perl模块执行的Perl脚本必须编写得很小心——因为它们将运行在服务器的内部，所以在变量的使用和初始化方面就必须非常非常的谨慎。第二点，因为进程的长度可能会很大，所以你只有在大量增加了服务器机器上的内存之后才能更多更好地使用mod_perl模块。尽管如此，许多站点仍然认为mod_perl模块的功能和灵活性使这一问题的利远大于弊，因此因特网上广泛使用着mod_perl模块。在大多数情况下，你甚至可以把mod_perl脚本未经修改地从Linux或UNIX服务器机器上的Apache里直接移植到微软Windows系统上的Apache去。

这本书的篇幅只允许我们向大家简单地介绍一下怎样才能把mod(perl模块支持添加到Apache里以及如何开始为它编写你们的第一个Perl脚本。我们希望对这一高级论题的简单介绍能够鼓励大家继续对此做深入的研究，而在大家掌握了基本的原理之后，网上还有无数的在线文档可以帮助你不断前进。

你首先需要有Apache的源代码。如果你的发行版本只有预装的Apache而没有源代码，就必须按我们曾经走过的老路做一遍——卸下预装的Apache，下载它最新的稳定版源代码，自行编译并安装它们。别担心，Apache的编译和安装工作是很容易进行的，这只要花几分钟的时间。

我们假设你将会在一台测试用机上做这件事情。并且在把与现有的Apache服务器有关的一切东西都保存起来以后，已经把它卸安装完毕并重新开始一次“清洁的”安装。如果你的Linux发行版本已经自带了Apache和mod_perl模块，那你可就省大事了；你可以把这些步骤都跳过去。如果不是这样，最安全的办法莫过于把发行版本自带的所有Web服务器软件程序都卸下来，然后从源代码开始重新建立Apache，这样就可以把mod_perl模块链接到其中去。别担心，这真的很容易。

先去http：//www.apache.org站点下载最新版的源代码，它们一般都保存在一个经过gzip压缩的tar文件里，文件名带有版本号。把这个文件下载到一个方便的子目录里，对它进行解压和解档操作。注意阅读INSTALL文件里的安装指南。如果你打算把Apache安装到子目录/usr/local/apache里去（这是最常用的安装位置），那么第一步要执行“./configure --prefix = /usr/local/apache”命令，然后再执行make命令。如果一切正常，用su命令把自己变为根用户，再执行“make install”命令。最后，你用“/usr/local/apache/bin/apachectr start”命令启动自己的Web服务器就大功告成了。我们说什么来着，确实不难嘛！

让自己从源代码开始建立起来的Apache运转起来之后，你就可以开始安装mod_perl模块了。这一过程在我们写这本书的时候还需要有点技巧，但并不是特别复杂。

假设你已经安装好Perl，但手里还没有什么附加模块，那么要想让mod_perl模块工作还必须

先到CPAN站点上取一些附加模块回来，CPAN的全称是“Comperhensive Perl Archive Network”（智能Perl档案网络），站点网址是http：//www.cpan.org。mod_perl模块需要依赖大量的其他模块，几乎每次编译它都会报告你需要先安装另外一个模块，这是它不太方便的地方。但这也可以看出Perl模块的再使用性是多么的高，所以这并不是什么坏消息！但令人头痛的是那些模块本身还需要其他的模块。我们这里借用Douglas Adams在“The Hitch Hiker's Guide to the Galaxy”（《黑客的天堂之路》）一书里的话来安慰大家一下：“别着急”。其实有一个比较容易的解决方案，那就是CPAN的Perl模块。

第一步是下载FTP模块。在我们编写本书的时候，它还是libnet模块的一部分。登录站点http：//www.cpan.org的FTP服务，把该模块下载到一个空目录里去，把文件释放出来。然后，你可以只用下面这四条命令就把它安装好：

```
perl Makefile.PL
make
make test
make install
```

严格说来你可以省略“make test”步骤，但安全总比后悔要好。现在你就有了一个能够替你完成FTP工作的Perl模块了。其实它还可以用于SMTP、NNTP等几种其他的协议，但我们现在最感兴趣的就是FTP 。

第二步是安装Andreas Konig编写的确实奇妙的CPAN模块。这个模块需要你配置一次，你必须告诉它对你自己的站点来说哪个CPAN镜象站点是最好的下载地点，然后它就能够从CPAN那里把你需要的模块取回来，找出它们的依赖模块，把那些也取回来，然后一次性把模块们都安装好。

对CPAN模块重复执行刚才对libnet模块进行的操作。CPAN模块会问你几个问题，但不必担心，这都是些“我在哪儿可以创建一个工作目录”或“你居住在哪个大陆”之类的小儿科问题，没有更难回答的了。

把CPAN模块安装好以后，就进入第三阶段：下载安装Apache模块包。

注意，因为下载CPAN模块需要连接因特网，但在两次下载操作之间可能需要完成一些本地的处理工作，所以如果你使用的是一个有自动倒计时功能的拨号上网链接，就可能需要采取一些步骤来保持拨号链接的活跃状态。告诉大家一个简单的花招，就是执行一条“ping -i15 www.perl.com”命令，但千万别忘了在CPAN变完魔术后把ping进程杀掉，要不然拨号链接可就下不了线了。

用下面的命令以交互方式启动CPAN模块：

```
perl  _MCPAN  _e shell
```

Perl将启动，你会看到一个“cpan>”交互操作提示符。

输入下面这条命令：

```
install Bundle: :Apache
```

然后就坐在电脑前看魔术表演吧。它会把所有的苦差事都替你搞掂。

如果最后一步安装mod(perl模块的操作失败了，请不必担心。我们过一会儿还得遇见它——我们还得给它设定一些附加选项呢。

CPAN完成操作之后，输入“exit”命令返回到shell提示符下。

第四步是重新从CPAN站点手动下载mod_perl模块（看到这儿你可别害怕），因为我们需要一些与CPAN模块给我们的缺省值不一样的配置选项。象往常一样解开压缩文件，但在对它的这次配置里我们要额外加上一些命令行选项。如下所示：

```
perl Makefile.PL EVERYTHING=1 APACHE_PREFIX=/usr/local/apache
```

这条命令表示我们需要全部的mod_perl选项，而我们的Apache网络服务器软件安装在子目录/usr/local/apache里。如果你的Apache安装在另外一个地方，请对这条命令做相应的修改。命令几乎是立刻提示你输入在其中建立Apache的源文件目录，输入它，注意可能需要你在路径名的最后加上“/src”部分。我们是在一个本地子目录上建立Apache的，在作者之一的机器上，我们回答的是/home/rick/apache/src/apache_1.3.9/src。

在大家读到这本书的时候，Apache的版本号可能已经增加了。

接下来要提问一些与重建Apache有关的问题，如果没有什么其他的想法，直接yes到底就是了。接下来要执行make命令重新建立我们的httpd二进制代码，它将建立在Apache服务器的src子目录里。

编译步骤结束后，你就可以执行“make test”命令来检查自己是否已经建立起一个激活了mod_perl模块的Apache服务器。这条命令会用一个不同的配置文件在另外一个端口运行一个特殊的httpd进程，不会与机器上正在运行的现有Web服务器发生冲突。但这个操作可能会失败，在我们的机器上就是如此！好在这看起来是测试工作本身的一个问题而不是新建立的httpd的过错，所以如果没有其他的什么毛病，继续前进好了。

现在我们已经得到了一个带mod_perl支持的httpd程序，最后一个步骤是安装和配置它，然后我们就可以开始编写一个测试用的Perl模块了。

把路径切换到/usr/local/appche子目录，用su命令变为根用户。

如果你的Web服务器正在运行中，请用下面这条命令停止它：

```
/usr/local/apache/bin/apachectr stop
```

备份现有的/bin/httpd和conf/httpd.conf文件以防万一。

现在用你刚才在Apache的/src子目录里新建立的httpd文件覆盖/bin/httpd文件。你会注意到新httpd文件要比原来的老文件大许多。

在我们开始编写自己那简单的Perl模块之前还剩下一件事情要做了，那就是在Apache的配置文件里加上与Perl有关的设置项。

有好几种办法可以完成这件工作。而我们将在这里介绍给大家的办法从执行角度看并不是最有效率的，但它可能是最简单的办法了，而且对学习mod_perl模块也很有帮助。

编辑httpd.conf文件（记得先做个备份）。在文件里找个合适的地方把下面这几行加进去：

```
PerlFreshRestart On
<Location /blp-hello-perl>
        SetHandler perl-script
        PerlHandler Apache::Hello
</Location>
```

“PerlFreshRestart On”告诉Apache：当它被重新启动的时候，要重新加载所有的Perl脚本。

如果你正在调试Perl脚本，就必须加上这个选项。

接下来的内容有点像HTML文档里的一个段落，它告诉Apache在有人请求blp-hello-perl文档的时候它需要使用一个Perl脚本处理器来处理，而它将调用的Perl脚本是Apache模块里的"Hello"。

现在停止再重新启动Apache服务器，让它重新读入配置文件。如下所示：

```
/usr/local/apache/bin/apachectr stop
/usr/local/apache/bin apachectr start
```

到这里，我们已经有了自己的嵌入了Perl解释器的Apache服务器了，另外我们还配置它在我们请求一个名为blp-hello-perl的文档时去启动调用一个Perl脚本。现在是动手试试的时间了。

动手试试：一个mod_perl模块

我们将要编写的文件是/usr/local/apache/lib/perl/Apache/Hello.pm。

Apache安装目录下的lib/perl子目录是Apache中的mod_perl将自动在其中搜索Perl脚本的地点之一。如果你想把它放到另外一个地方去，就需要在httpd.conf配置文件里另外增加一行"PerlSetEnv PERL5LIB <comma separted list of places to look>"语句，新路径名用逗号隔开放在命令中的尖括号里。之所以把Hello.pm文件放在lib/perl/Apache子目录里是因为我们打算把我们的文件最终放入一个名为Apache的软件包里去。下面就是我们的第一个Perl模块。

```
package Apache::Hello;

use strict;
use Apache::Constants ':common';

sub handler {
    my $r = shift;
    my $user_agent = $r->header_in('User-Agent');

    $r->content_type('text/html');
    $r->send_http_header;
    my $host = $r->get_remote_host;
    $r->print(<<END);
<HTML>
<HEAD>
<TITLE>HELLO</TITLE>
</HEAD>
<BODY>
<BR>
END
    $r->print("Hello $host with browser $user_agent \n\r<BR><BR>Welcome to Apache
running mod_perl\r\n");
    $r->print("</BODY>\r\n</HTML>\r\n");

    return OK;
}

1;
```

操作注释：

这个文件声明自己是模块Apache里的那个Hello文件。然后我们打开"strict"（严格）选项（应该永远这样做），声明自己希望使用Apache模块里"common"类别的各种常数。

我们定义了一个子例程handler，这是一个特殊名字，Apache知道要调用它来为一个Web主

页生成输出。

“my $r = shift;”语句使我们取得了一个包含着请求对象的变量r。这个对象带有从Perl生成我们的主页所必须调用的全部常用例程

接下来，我们取得了已经从HTTP请求里传递来的用户方代理，发送出content_type（内容类型）、发送出一个预先定义好的HTTP表头、再取得远程主机的名字。

最后那几行都是比较简单的Perl语句，用来生成一个非常基本的HTML主页，这个主页将简单地通过调用请求对象自带的print方法而被发送给浏览器。

这就是我们要做的全部工作，就差一件事——测试，也许还需要做点调试工作。

启动我们的浏览器，把它指向http：//localhost/blp-hello-perl。

如果一切顺利，我们就将看到如图20-13所示的画面。

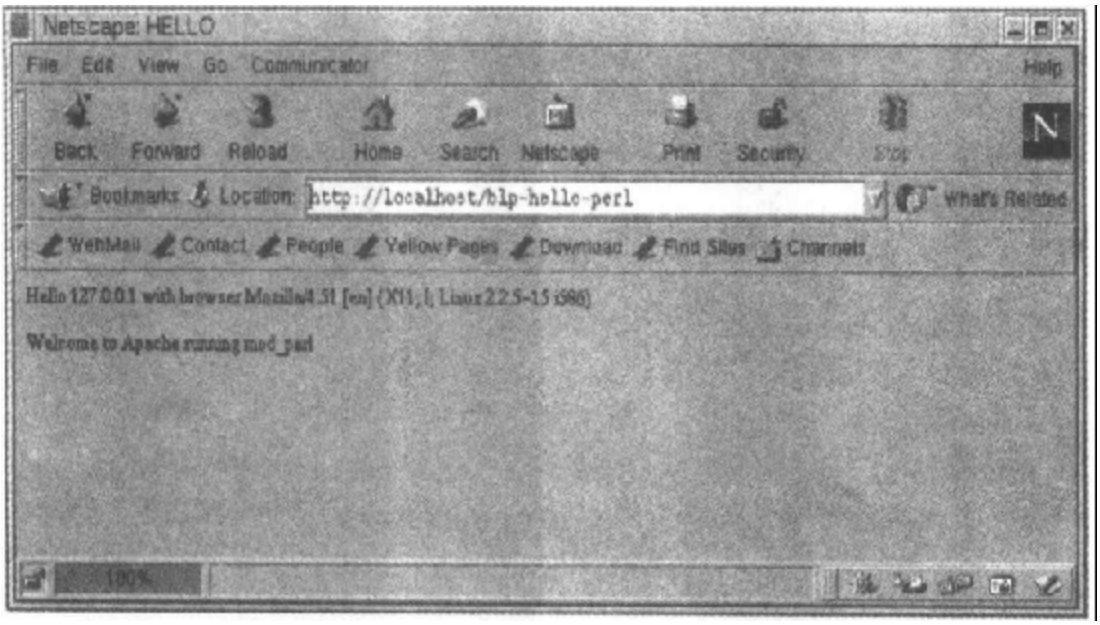


图 20-13

如果第一次尝试没有成功，不要灰心。先去试试请求一下http：//localhost/index.html。你应该看到标准的安装画面，告诉你Apache运行得一切正常。如果能够看到这个画面，你可以到/usr/local/apache/logs子目录里的日志文件里查查，看能不能找出一些哪里出了问题的好线索。

可惜的是因为篇幅有限，我们只能给大家准备这样一个极其简单的mod_perl示例了。但好在网上有数不胜数的资料，沿着http：//www.apache.org站点上的链接找去吧；如果你真的在Apache模块上遇到了很严重的问题，可以再找几本专门介绍Apache的书来看看。

20.8 本章总结

这一章向大家介绍了创建动态交互式Web主页的方法。这是一个发展快得让人跟不上节奏的领域，在网上显示信息的方法随时都在更新。我们希望这一章内容能够在理论和实践两方面帮助大家掌握用C语言自行编写简单CGI应用程序的基本原理。

下面是我们这一章学习内容的一个总结：

- 用来编写表单的HTML语言元素，表单用来输入客户端信息。
- 对表单进行传输编码。
- 用Perl或C语言对编码表单信息进行解码。
- 利用CGI程序的响应动态创建HTML主页。
- CGI程序设计的技巧和窍门。
- 使用CGI程序访问我们的服务器数据库。
- 简单介绍了mod_perl模块这个高级论题，它可以把一个Perl解释器嵌入到Web服务器Apache软件里去。

第21章　设备驱动程序

到现在为止，我们一直是把注意力集中在应用程序的编写方面。我们接触到许多面向应用程序设计人员的函数库、工具开发包、应用程序开发系统等。它们可以帮助程序员访问与文件系统、内存、网络等事物连接着的计算机硬件和设备。这些东西有一个共性，那就是允许我们的应用程序向内核——操作系统的核心部分提出操作要求。

为了使这本书的内容更加完整，我们将从另外一个角度对内核进行一番研究。为了满足来自应用程序的操作要求，内核需要依靠不同的设备驱动程序（device driver）与它可能遇到的各种类型的设备打交道。尽管硬件设备的底层技术千差万别，但通过设备驱动程序，内核却能够向应用程序提供一个形式统一的程序接口。文件系统就是一个明显的例子——文件可以保存在IDE硬盘、SCSI硬盘、CD-ROM光盘、软盘甚至RAM盘上，但我们对文件进行操作时使用的命令却都是一样的。

编写设备代码并不见得比编写应用程序代码更困难，但它需要我们掌握一些新函数库，考虑一些新问题，这些问题都是我们在应用程序空间里不曾遇到的。在编写应用程序的时候，内核能够为我们的错误提供一张安全网；但我们将要编写的代码将构成内核自身的一部分，而这就意味着在内核空间里原来的安全网已不复存在。内核代码对计算机有绝对的控制权，能够阻止其他任何进程的执行；而我们编写的代码绝不能滥用这种权利，它们应该成为其他进程的友好邻居。

在为Linux编写内核模块的时候，我们拥有一个无可比拟的优势，那就是全部的源代码都可以让我们随时拿来研究、修改和再使用到我们的代码里去。在这一章里，我们将随时指出各种头文件和模块的源代码出处供大家做进一步参考。做为信息、忠告和灵感的源泉，源代码的重要性无论怎么强调都是不过分的。

这一章的学习重点集中在英特尔公司的x86体系结构上，但我们也在一台Compaq公司的Alpha机器上对其中的各种模块进行了测试，所以它们的可移植性还是有一定保证的。当然，在这两种平台以及其他平台之间肯定存在着差异，我们将在后面专门讨论可移植性问题的内容里尽可能地把它们都勾画出来。可以这样说，绝大部分代码在其他体系结构上都是能够正常工作的，而我们将随时把学习过程中遇到的依赖于特定计算机平台的部分（它们基本上都是因为硬件体系结构方面的差异而造成的）尽可能明确地指出来。在最近几年的内核开发工作中，可移植性已经成为一个非常重要的焦点，几乎所有的内核驱动程序API的可移植性都非常高。当然，部分代码的可移植性永远也到不达百分之百的程度，但如果我们没有在讨论内容里特别提到可移植性方面的问题，它就应该能够工作在读者可能使用的各种计算机上。

21.1　设备

设备到底是什么呢？它可以是一台计算机上任何一个部分，既可以直接制造在计算机的核

心，也可以是一个外围设备——比如SCSI控制器、扩充的串行口、网卡等等。在说起编写新的设备驱动程序时我们一般都指的是后一种情况——即为你弄到手的最新的接口卡添加必要的软件方面的支持。

设备驱动程序是这样一类软件：它们控制着设备的操作动作，并且提供了一个可用的程序接口使其他程序能够与这个设备互动。设备驱动程序并不一定控制着某个物理性的硬件外部设备，比如/dev/random（用来产生随机数据，后面还有对它的讨论）和/dev/vcs0（当前虚拟终端）就是这样的例子。这些设备与真实的硬件并没有什么联系，它们只是从内核获取数据再送往应用程序的一种手段，它们通常被称为软件设备。

各种设备驱动程序构成了它们所控制的硬件和操作系统内核之间的一个过渡层次。这个层次扎根于硬件，服务于内核，极大地简化了内核的设计和应用——它向外界提供了一个精心定义的接口，具体的工作将由各个设备驱动程序去完成，而内核就不必亲自去与每一个设备打交道了。这也意味着操作系统的内核部分只要还能适应有关模型的框架，就可以（而且确实是）在不了解各种不同设备具体情况的前提下被编写出来并做进一步的开发。比如说，如果想给Linux内核新增加一种文件系统，我们并不需要全部重写内核代码，并且这一工作能够通过现有代码的再使用相对容易地得到完成。就拿文件系统来说吧，它被分为一个普遍意义上的虚拟文件系统VFS和各种注册在VFS上的特定文件系统。内核的一切主要部分都是按这种方式设计的——比如CD-ROM子系统、SCSI控制层，等等。这就提高了整体上的模块化水平，新设备驱动程序的设计工作也因此而更容易进行（如图21-1所示）。

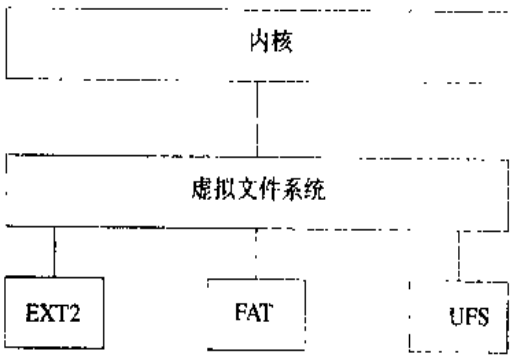


图 21-1

21.1.1 设备的分类

设备驱动程序可以根据它们的行为分为不同的类型。两种最基本的类型是字符设备和块设备。字符设备以字节为单位进行读写，数据缓冲系统对它们的访问不提供缓存。请大家想象一下磁带上的数据或歌曲是如何以一种不间断的信息流的形式被读出的吧——字符设备正是这样提供数据的。而块设备则是另外一种情况，它们允许随机访问，每次读写的数据量都是数据块长度的整数倍数，并且访问还会经过缓冲区缓存系统才能实现。与其他UNIX版本形成对照的是：Linux内核允许不是数据块长度整数倍数的数据量被读取，但这种区别纯粹是学术方面的问题。

大多数设备驱动程序都要通过文件系统进行访问，但网络设备是一个例外——它们在Linux

上没有与之对应的文件数据项，就象其他一些UNIX变体一样。/dev子目录里充斥着各种设备特殊文件，其内容看起来与其他子目录没什么两样，如下所示：

```
$ ls -l /dev
...
brw-rw----   1 root     disk      22,   1 May  5  1998 hdc1
crw-rw----   1 root     daemon     6,   0 May  5  1998 lp0
...
$
```

使这些“文件”与普通文件看起来有所不同的一个地方是那些c和b“标志”，它们分别把对应的设备划分为一个字符设备或一个块设备。从这个例子里可以看出这两种设备的明显区别：例如，当从磁盘上拷贝文件时不必重新从最开始处读取整个磁盘时，允许对打印机进行随机访问也就没有丝毫意义了。设备特殊文件并不一定非得放在/dev子目录里，但这是一个传统上的做法，同时还有利于保持系统的秩序。部分UNIX系统在/dev下还细分有磁盘子系统专用的子目录，但Linux通常都只有一个庞大而又拥挤的子目录。

跟在特殊文件属主和分组数据项后面的数字分别是设备的主编号和辅编号。主编号表明这是哪一种设备，而辅编号表明这是哪一个具体的设备。在上面的/dev子目录清单里，hdc1的主编号是22，辅编号是1。主编号22是第二块IDE控制器的正式编号，而IDE子系统将根据辅编号的值标识出主设备和从设备上的分区。事实上，你可以给特殊文件起任意的名字，内核只关心它的类型和主编号。

新特殊文件是用mknod系统调用创建的，而这个系统调用是由同名的工具命令调用的。mknod命令/的语法定义如下所示：

```
mknod name type major minor
```

因此如果想创建出上面的lp0项，你必须以根用户身份执行下面的命令：

```
# mknod /dev/lp0 c 6 0
#
```

你可以把它们创建在任何地方，但前提是主机文件系统必须支持你的做法。大多数参数的含义从字面上就能看出来，我们只对与设备分类有关的type做一下说明——b代表块设备，c代表字符设备。进一步资料可以从mknod命令的使用手册页里查到。

21.1.2 用户空间与内核空间

我们首先要弄清编写设备驱动程序和编写用户空间应用程序之间的区别。Linux运转在两种模式下，一种是用户模式，另一种是超级用户模式。我们将从现在起把后一种称为内核模式，因为它能更真实地体现出该模式中的事态发展。这反映出处理器实际处理指令方式的改变——对这两种执行模式的支持都是在其内部完成并转换的。英特尔的x86（x >= 3）把自己的执行模式命名为Ring（环）0、1、2、3，第0环的优先级最高。在Linux里，第0环代表内核执行模式，第3环代表用户执行模式——其余两环没有使用。其他体系结构有类似的规定，但在执行模式的命名方式上采取了不同的做法。我们不打算在硬件细节方面浪费大家的精力，只想告诉大家一个进程所处的操作模式将对允许它做的事情施加特定的限制。有些事是运行在用户空间的进程绝不允许做的，其中就包括直接访问硬件和执行某些指令在内。

当你编写一个普通程序的时候，你会想当然地用到一些函数，比如printf和strcpy等。它们都是C语言标准库libc的组成部分，在相应的头文件里都有预先定义好的架构。用户空间应用程序通常都要与libc库进行链接，而这些符号将在运行时得到解析。但内核模块就不同了，它们将与内核进行链接，在使用它们自己向外提供的函数方面是有限制的。一个做为内核模块编写出来的设备驱动程序的运行并不是普通意义上的运行，模块中的符号是在它被加载到内核里去的时候得到解析的。

在编写内核级代码的时候大家应该遵循一些与此有关的程序设计准则，注意培养自己良好的程序设计习惯。下面就是一些这样的原则——有的一目了然，有的需要点明。

- 不要使用浮点运算。内核在切换处理器的执行模式时不保存它的FP状态，所以如果你确实使用了浮点运算，就必须自行保存其状态。但通常并没有什么好的理由需要在内核代码里使用浮点数。
- 在你的驱动程序里不要进行繁忙的等待。用户空间里的一个应用程序永远也不可能完全独占CPU；但内核里的一个用时1秒的循环看上去就象是把整个系统挂起了这么长的时间，并且在这段时间里，其他什么工作也不能做。
- 不要自以为是。我们将在后面看到，内核空间里的调试工作会遇到很多困难，而驱动程序在与硬件打交道的时候往往需要精确的定时——甚至增加一条打印语句都有可能搞乱定时从而把一个明显的漏洞掩盖起来。要尽可能地保持代码的简洁性和易理解性。

1. 功能取舍

你还必须决定哪些功能将被实现为模块，还有哪些功能要留在用户空间。一般原则是：只要是能够在用户空间里编程实现的东西，就绝不要把它放到内核里去！这是一个理由非常充分的忠告；在代码有错的情况下，用户空间里的错误会输出内存映象，而内核空间里的错误则可能会完全挂起。让内核空间充斥着毫无用处的代码可不是什么好习惯。如果执行速度和操作定时方面没有那么严格和关键，在内核模块里准备一些供应用程序使用的调用入口通常会达到最佳的效果；而且这样做还可以让你继续享受有用户空间标准函数库可以链接的好处。

2. 建立模块

对内核模块进行编译并不比你已经习惯的对普通应用程序进行编译来得困难。本书中的例子里都附带有makefile供读者查阅。对内核模块进行编译的基本过程如下所示：

```
gcc -D__KERNEL__ -D__SMP__ -DMODULE -DMODVERSIONS -I/usr/src/linux/include -Wall
                    -O2 -o module.o -c module.c
```

如果读者从没有注意过内核的建立过程，可能会觉得这个命令看起来有点唬人。上面大部分定义都可以在模块里通过读取内核中相应的设置值而得到直接的处理。我们来看看它们都是干什么用的（见表21-1）。

表 21-1

__KERNEL__	并非所有的内核头文件都是只能由内核本身来使用的。有些用户空间应用程序也会把它们包括上，只是其中有些内容是属于内核专用的，必须把它们对用户空间隐藏起来。将要插入到内核里去的代码必须定义编译标志“__KERNEL__”以看到头文件的全部内容

（续）

__SMP__	内核可以被编译为供SMP（Symmetric Multi Processor，对称多处理器系统，即有一个以上处理器的系统）或UP（Uni Processor，单处理器系统）机器使用。如果一个模块将被插入到一个SMP内核，就必须定义“__SMP__”编译标志
MODULE	如果代码将被编译为一个内核模块，就必须定义这个编译标志
MODVERSIONS	这个标志的作用是检查内核与模块之间的不兼容性。详细说明见下面的内容

剩下的gcc命令行选项都不是内核专用的，而大家肯定也已经在前面见过它们的用法了。“-O”开关告诉gcc在编译期间需要进行几遍优化。为了使用内核级的函数——比如outb等，内核模块在编译时至少要达到“O2”或更高的优化级别才行。高于“O2”的级别也管用，但我们并不推荐这样做。把所有的警告机制全都打开，这对编写包括内核模块在内的任意类型的程序来说都是个好习惯。

/usr/src/linux是安排Linux内核树的好地方，但这并不是一个硬性的要求，你可以把它存放到任何地点。gcc命令的“-I”选项（头文件路径）告诉它把这个路径也添加到头文件的搜索路径上去。内核专用的头文件一般都保存在内核树的include/子目录里，所以，当我们写出如下所示的代码时：

```
# include <linux/module.h>
```

我们实际指的是/usr/src/linux/include/linux/module.h文件。在include/子目录里有两个子目录比较重要，它们一个是linux/，另一个是asm/。前一个子目录里包含着与计算机平台无关的文件；后一个子目录实际上是一个指向asm-arch/子目录的符号链接，arch可以是“i386”或“alpha”等。这个链接是在配置和创建内核时创建的，而且指向头文件子目录的两个符号链接分别指向/usr/include/linux和/usr/include/asm。

MODVERSIONS定义用在带函数版本检查功能的内核上。它可以防止模块被加载到一个与它不兼容的内核里去，要不然轻则引起功能的紊乱，重则引起内核的崩溃。在对模块进行编译的时候，模块们都将针对当前运行中的内核版本而建立起来，以后一般也只能加载到与此精确对应的内核版本上。带版本检查功能的内核会给向外界提供的导出函数加上校验和（checksum）信息做为其后缀，这些校验和信息与它导出的程序接口有密切关系。

这就使我们能够安全地了解到底层API是否在模块建立好以后又出现了变化。你可以利用查看/proc/ksyms文件的方法来检查自己是否运行在一个带版本检查功能的内核上，这个文件包含着运行中的内核导出的各种符号。如果导出函数的名字后面有一个“_Rxxxxxx”形式的后缀，就说明它激活了版本检查功能。如下所示：

```
$ cat /proc/ksyms
...
c0115728 printk_Rsmp_1b7d4074
c01d3ed0 sprintf_Rsmp_3c2c5af5
...
$
```

上例中，函数真正的名字是printk和sprintf。从上面的输出结果可以看出我们正运行着一个激活了版本检查功能SMP内核。要想对这些名字进行解析，在编写模块时就必须包括上头文件linux/modversions.h文件。

另一个问题与名字空间有关，除非你曾经参加过大型项目的开发工作，大多数人是很少会特别注意到这一方面的。在做内核开发的时候，一定要特别注意不要把全局性的内核名字空间弄乱了。在导出函数的名字之前加上驱动程序的名字是一个避免出现名字冲突现象的好办法，绝大多数内核开发人员也正是这样做的。另一个好习惯是只导出将会被其他驱动程序用到的函数和变量，两种办法同时使用效果就更好了。把全局变量和函数声明为静态变量和静态函数也可以达到这一目的，但有一些其他的副作用。其他变量和函数的导出要明确地使用EXPORT_SYMBOL宏命令来进行，它会把它们添加到内核的全局符号表里去。一般说来，只有在准备把驱动程序分为几个模块或者准备暴露驱动程序的内部细节以做它用的情况下才需要考虑这一问题。但不管怎么说，把全局名字空间的“污染”降低到最小程度永远是一个好主意。有关命令的语法定义都是很容易理解的，如表21-2所示：

表 21-2

EXPORT_SYMBOL(name)	导出代表变量或函数的符号name
EXPORT_SYMBOL_NOVERS(name)	导出代表变量或函数的符号name，但不加上模块版本检查后缀——即使定义了也不加
EXPORT_NO_SYMBOLS	不导出任何符号

有关定义和声明都必须出现在函数之外，比如头文件里、模块的底部等位置。这几个宏命令的定义在linux/module.h头文件里。在使用这些宏命令中的任何一个之前，必须先定义EXPORT_SYMTAB标志。

3. 数据类型

Linux可以运行在许多不同类型的体系结构上，有些是32位的，有些是64位的。甚至曾经有人尝试让Linux运行在16位的硬件上，今后会发展成什么样谁也说不好。因此，千万不要认为一个指定类型的变量其尺寸长度就是固定不变的，这一点很重要。Linux定义了一些标准类型，它们在各种平台上的尺寸长度都是一致的，见表21-3。

表 21-3

__u8, . . . __u64	字符到64位长度之间带正负号和不带正负号的变量
__s8, . . . __s64	

如果你需要一个特殊长度的变量，就一定要使用上表里定义的类型。当你几个月后回过头来查阅驱动程序的源代码时，这些定义既让人看着舒服，又使代码更容易理解。还有一些特殊类型是用在驱动程序的入口点处的，请看下面这个定义：

```
ssize_t schar_read(…, size_t count, loff_t *offset)
```

ssize_t和size_t就是在内核里有特殊作用的类型定义——我们会在后面的内容里对它们做进一步的介绍。这些定义的作用都很直接，因此程序员就不需要为数据的长度问题操太多的心了。对类型的定义都在linux/types.h和asm/posix_types.h头文件里，如果你想知道它们具体的typedef定义长度是多少，可以自行查阅这两个文件。当你需要把它们赋值给编译器的整数类型时，记住一定要使用一个正确的投射关系。gcc对数据尺寸的不匹配问题相当挑剔，会把它认为有问题

的报告给你。

动手试试：一个内核模块

我们现在开始编写我们的第一个内核模块。根据以往的传统做法，我们决定编写一个“hello kernel”模块——别不耐烦，这是我们这本书里最后一次编写这样的程序了。

1) 我们从下面的hello.c文件开始动手。

```
#include <linux/module.h>

#if defined(CONFIG_SMP)
#define __SMP__
#endif

#if defined(CONFIG_MODVERSIONS)
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/kernel.h>

int init_module(void)
{
        printk(KERN_DEBUG "Hello, kernel!\n");
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_DEBUG "Good-bye, kernel!\n");
}
```

printf在内核中的对应函数是printk。它们在使用方法上差不多，但后者不支持浮点数的打印输出。printk语句中的KERN_DEBUG定义的作用是设置被打印消息的优先级。优先级的可用值都定义在linux/kernel.h文件里，如下所示：

```
#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 */
```

因此下面这个写法和上面那条语句的作用是一样的：

```
printk("<7>Hello, kernel!\n" );
```

但使用预定义的优先级水平可以使代码更容易阅读，大家想必都会同意这一点吧。消息的记录级别控制着哪些消息会被输出到控制台，哪些消息会被追加到系统记录的尾部。这是由syslogd控制着的，其具体设置一般是在/etc/syslog.conf文件完成的。因为这个原因，你可能不会直接在屏幕上看到这条消息，它们会被保存到专门为这类消息分配的内核缓冲区里去。dmesg程序可以把保存在那里的东西显示给你看（从你上次开机算起，应该有不少东西了）。

除了printk，这段代码里应该没有什么不熟悉的东西了——新东西我们一会儿再说。就目前而言，我们先在你创建hello.c文件的子目录里用下面这条命令来编译这个模块：

```
$ gcc -D__KERNEL__ -I/usr/src/linux/include -DMODULE -Wall -O2 -c hello.c -o hello.o
$
```

2) 现在把这个模块插入到内核里去。模块是用insmod（插入模块）命令加载的，它一般存放在/sbin/子目录里。只有根用户才能插入和移出一个模块，如果不是这样就将是一个严重的安全漏洞——要是什么人都能鼓捣内核不就全乱了嘛！请把自己变为根用户，输入下面的命令：

```
# insmod hello.o
#
```

3) 没有出错信息，那么我们怎样才能肯定模块已经成功地插入到内核里去了呢？我们可以用下面这条命令查看dmesg从内核缓冲区里读出来的最后一条消息：

```
# dmesg | tail -n1
Hello, kernel!
#
```

大家可以看到来自我们模块的消息出现在内核缓冲区里了。为了进一步证明这个模块确实被加载了，我们可以再用lsmod（列模块清单）命令检查一下。它会把当前插入的所有模块都列出来。如下所示：

```
# lsmod
Module                  Size  Used by
...
hello                    176   0  (unused)
...
#
```

4) hello和其他模块一起出现了，这可没有什么好奇怪的。lsmod命令还列出了每个已加载模块的长度和使用计数。使用计数我们一会儿就要讲到。现在，我们先把hello模块从内核里取下来。rmmod（移除模块）命令将把一个模块从内核里取出来。如下所示：

```
# rmmod hello
# dmesg | tail -n1
Good-bye, kernel!
#
```

操作注释：

这里出现了几个新生事物。linux/module.h还包括着另外一个名为linux/config.h的头文件（以及其他头文件），它包含着以常数定义“# define”形式定义的内核编译选项。这对我们来说是非常的方便，因为我们可以肯定只要内核是为SMP系统编译的，CONFIG_SMP标志就会被定义，而我们的__SMP__定义也就能够发挥作用。CONFIG_MODVERSIONS标志也是如此：如果配置上了这个标志，我们就能够定义MODVERSIONS，进面包括上头文件linux/modversions.h以获得版本检查信息的访问权。

正如我们看到的，当模块被加载和卸下的时候，一条小消息被记录到内核缓冲区里去了。我们定义了两个函数来完成这两项工作，它们是真正需要在模块里实现的东西。在加载模块时需要调用init_module函数，它负责设置模块内部的数据结构、初始化硬件，以及做好第一次使用设备之前应该完成的所有其他工作。上面这个例子只是一个简单的框架，所以我们没有真正做什么事情。事情总是两方面的，所以cleanup_module负责关闭设备和释放设备可能占用过的各

种资源。

总之，我们建立了一个模块，并且成功地把它插入到内核里去了。我们现在已经克服了对Linux内核的恐惧，接下来我们去接触一些更实际的东西。

21.2 字符设备

上面的hello模块可能是大家能够见到的最简单的内核模块了，它并没有做什么真正有意义的事。我们不能在它加载之后与它互动，它也没有导出什么有用的函数给用户空间的应用程序。做为前进的第一步，我们来看一个字符驱动程序的框架结构示例。

字符设备必须向内核注册它们自己，向内核提供一些必要的信息使它能够在有应用程序希望与这个设备互动的时候调用正确的函数进行处理。register_chrdev就是负责这一工作的，它的调用语法如下所示：

```
int register_chrdev(unsigned int major, const char *name, struct file_operations
*fops)
```

它在失败时的返回值是一个负数；如果成功则返回一个非负数（正整数或0）——这是给出了内核函数主编号的情况。如果在调用这个函数的时候把major设置为0，内核将给这个设备动态地分配一个主编号。在这种情况下，设备将以下一个可用主编号进行注册，而函数的返回值就是这个主编号。主编号的动态分配使用起来并不困难，但如果在你上次加载过这个模块后主编号又发生了变化，你这次就不得不创建一个正确的特殊文件才能对设备进行访问。因为这个原因，我们将在书中使用系统保留的主编号以降低模块的编写难度。主编号42和120-127是系统为本地设备预留的，成品模块是不使用这几个主编号的。详细资料请参考Documentation/device.txt文件，那里面还有如何申请一个正式的主编号的办法。

第二个参数name只有一个用途——就是向/proc/devices进行注册：这个名字将出现在那里，仅此而已。最后一个参数文件操作结构是最有意义的。它定义了设备与外界交流的方式方法；特别是规定了哪些功能由它自己来负责完成，又有哪些功能需要由内核中的缺省函数来完成。这个结构的定义出现在linux/fs.h文件里，是由一系列函数指针组成的。这个概念被用在内核中的许多地方，它是不同设备层次的一个抽象格式，规定了它们的互动方式。

文件操作

对设备的访问需要经过文件系统中的特殊文件，这我们在第一小节就已经知道了。不管设备本身是与文件系统直接有关的（比如一个硬盘驱动程序）还是完全没有关系的（比如一个并行口驱动程序），它们的特殊文件都独立于设备的具体类型。因此，设备驱动程序需要注册一组文件操作，这些文件操作定义了设备提供的特定功能。下面是文件操作结构及其最新的框架定义。我们把所有可能的函数都列在这里了，但很少有一个模块需要全部定义它们的情况。你会发现这些函数的名字有许多和我们在第三章里见过的底层设备访问函数是一模一样的。事实上，一旦内核确定下来需要联系哪个设备以满足操作请求，就会用设备的文件操作函数取代那些底层的调用。

```
struct file_operations {
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct entry *);
int (*fasync) (int, struct file *, int);
int (*check_media_change) (kdev_t dev);
int (*revalidate) (kdev_t dev);
int (*lock) (struct file *, int, struct file_lock *);
}
```

驱动程序需要向内核报告自己代表的设备都提供了哪些功能，为此驱动程序可以随意选择这些文件操作函数的组合。我们来看看这些函数都是干什么用的见表21-4：

表 21-4

llseek	llseek对应着用户空间里的lseek，它的作用是改变文件结构中的操作位置。说的明确一些就是修改file->f_pos；我们将在讨论文件结构的时候再详细介绍它。在成功时，它返回一个新位置，失败时返回一个负数值
read	read（读）是从应用程序的角度看的说法，所以read实际上就是把数据写到用户空间里去。如果返回值是一个正数，就是实际“读”到的字节数。负数返回值表示出现一个错误
write	write的作用是向设备馈送数据，返回值方面与read相同
readdir	readdir只有文件系统才能使用，它的作用是读取某个子目录里的内容
poll	poll允许应用程序响应来自设备的给定事件。它在BSD UNIX里的对应函数是select，但Linux不推荐使用select，所以我们应该用poll代替它
ioctl	ioctl的含义是I/O控制，它允许应用程序通过ioctl系统调用控制设备的行为或者从设备取得数据
mmap	mmap实现了设备地址空间到用户空间的地址映射。它可以用来提供对设备驱动程序的内部缓冲区或外设内存空间的进行直接访问的功能
open	open是应用程序打开设备时将要调用的文件操作。它是惟一一个对字符设备和块设备都有缺省实现的函数。因此，如果你不需要或不想知道设备会在何时被打开，就可以不对这个文件操作进行定义
flush	flush的作用是把缓冲区数据“冲”出去。由于字符设备不使用缓冲区，所以这个条目只对块设备有意义
release	release是在设备关闭时将被调用的文件操作
fsync	fsync的作用是同步内存中与磁盘上的数据状态，把输出缓冲区里尚未写到磁盘上去的数据写出去。它在结束操作之前是不应该返回的。这个条目也只与块设备有关
fasync	fasync将在应用程序通过fcntl改变设备行为时调用
check_media_change	check_media_change检查自从上次访问之后介质是否经过了更换。因此它只对处理可更换介质（比如CD-ROM和软盘）的块设备有意义
revalidate	revalidate和check_media_change是密切相关的。如果检测出更换了盘片，就需要调用revalidate来更新设备内部的信息。revalidate也是只对处理可更换介质的块设备有意义
lock	lock使用户可以锁定一个文件。它也是只对文件系统有意义

任何一个设备都不太可能需要定义所有上述这些方法。你的设备需要哪些操作，你就定义哪些操作，而那些用不着的要设置为NULL。

当字符设备向内核进行注册的时候，设备的file_operations结构和设备的名字将添加到一个全局性的chrdevs数组里去，这个数组是由一些device_struct结构组成的，数组的下标就是设备的主编号。这个数组被称为字符设备切换表。device_struct结构的定义如下所示：

```
struct device_struct {
   const char *name;
   struct file_operations *fops;
};
```

这样，通过查看chrdevs[YOUR_MAJOR]->fops，内核就会知道如何与设备进行交谈以及设备都支持那些入口点见图21-2。

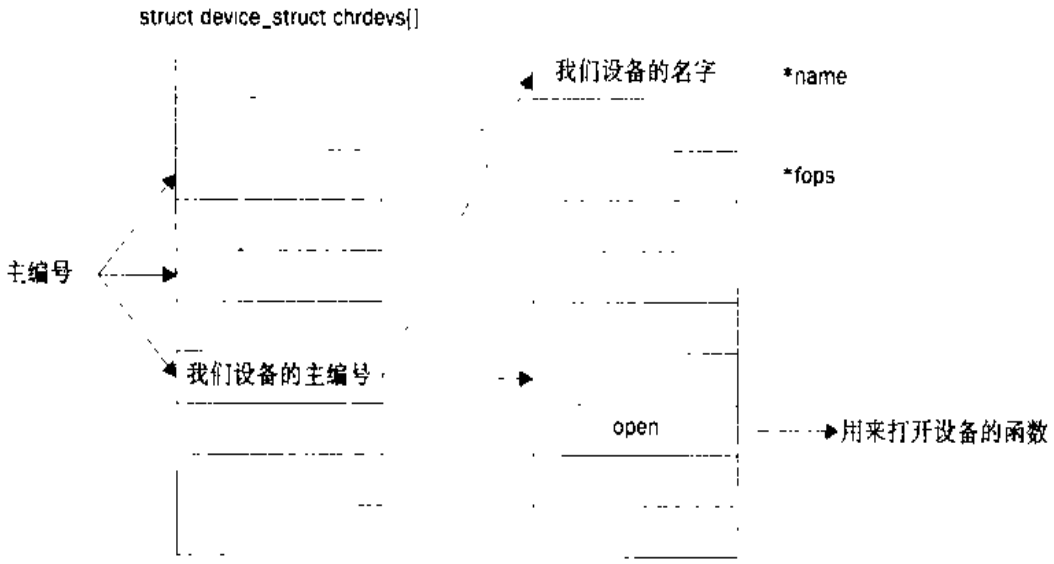


图 21-2

21.3 字符设备驱动程序示例：Schar

我们来看一个字符设备驱动程序示例，我们给它起名为Schar。这个设备以它自己杜撰的方式实现了读、写功能，它假装从一个以固定时间间隔准备好数据的字符设备那里取得数据再提供给应用程序。应用程序对设备的响应机制是由内核提供的定时器来管理的，我们将在下一小节对此做出详细的解释。

Schar.c以函数的预声明部分开始，这些提前声明了的函数组成了我们将要实现的该设备的file_operations结构。我们将分阶段编写出这些文件操作函数来。

```
/* forward declarations for _fops */
static ssize_t schar_read(struct file *file, char *buf, size_t count, loff_t *offset);
static ssize_t schar_write(struct file *file, const char *buf, size_t count, loff_t
*offset);
static unsigned int schar_poll(struct file *file, poll_table *wait);
static int schar_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
unsigned long arg);
static int schar_mmap(struct file *file, struct vm_area_struct *vma);
static int schar_open(struct inode *inode, struct file *file);
```

```
static int schar_release(struct inode *inode, struct file *file);

static struct file_operations schar_fops = {
        NULL,                   /* llseek */
        schar_read,
        schar_write,
        NULL,                   /* readdir */
        schar_poll,
        schar_ioctl,
        NULL,                   /* mmap */
        schar_open,
        NULL,                   /* flush */
        schar_release,
        NULL,                   /* fsync */
        NULL,                   /* fasync */
        NULL                    /* lock */
};
```

有些人喜欢把file_operations结构放在模块的最开始，有些人喜欢把它们放在最后面。如果采用后一种办法，我们就不需要非得在文件的开始部分对它们进行预定义。但这都是些个人习惯方面的问题。现在不要去管那些头文件，我们会随着内容的展开把它们介绍给大家。

21.3.1 MSG宏命令

在深入Schar.c的源代码之前，我们先要把我们将在这一章反复用到的MSG宏命令介绍给大家。它替代打印语句来输出调试信息，但用起来更方便。为了不让代码里面到处都是“丑陋”的“# ifdef”语句，我们把MSG的定义放在schar.h文件里去，如下所示：

```
#define DEBUG

#ifdef DEBUG
#define MSG(string, args...) printk(KERN_DEBUG "schar:" string, ##args)
#else
#define MSG(string, args...)
#endif
```

这会使代码看起来很整洁，并且会在每一条调试信息之前加上模块的名字。为各种事件定义一个掩码也很容易做到；如果愿意的话，你可以用这种办法对调试信息的记录功能做更精细的控制。Schar很简单，所以把一些都记录下来还是可以做到的。如果编译时没有定义DEBUG标志，MSG就将是一个空操作。有时候，在调试语句里加上更多的信息是有好处的，特别是大型设备驱动程序项目更是如此。我们也可以在调试信息里加上行号和文件名信息，这样我们就能在源代码里快速地找到与某条信息对应的位置。这应该是下面这样的一些语句：

```
#define MSG(string, args...) printk(KERN_DEBUG \
"schar:__FILE__-__LINE__:" string, ##args)
```

具体选择哪个宏定义完全在于你个人的决定。书中的模块都使用前一种定义，因为这个模块比较简单，不需要额外的信息。

21.3.2 字符设备的注册

类似于hello模块，Schar自然也需要用一个init_module入口点。它还将完成许多工作，比如分配内存、初始化定时器和变量，等等。但现在我们先来看看设备的注册问题。

```
int init_module(void)
{
   int res;

   if (schar_name == NULL)
      schar_name = "schar";

   /* register device with kernel */
   res = register_chrdev(SCHAR_MAJOR, schar_name, &schar_fops);
   if (res) {
      MSG("can't register device with kernel\n");
      return res;
   }
}
```

schar_name可以做为一个参数传递到模块里去——我们稍后再讨论它。代码先检查有没有一个给定的名字，如果没有，就把它设置为缺省值。

SCHAR_MAJOR在schar.h文件里被定义为42，我们前面已经说过这是一个系统保留给本地使用的设备主编号。其实，只要没有被其他活跃的设备使用，你选哪个编号都没关系。但既然是自己做实验，还是选一个保留的主编号更安全。

```
# define SCHAR_MAJOR      42
```

动态分配主编号也可以考虑。这个办法的好处是你不必操心主编号方面的事情，但坏处是（至少在我们这个例子里）我们每次加载这个模块的时候都要在创建了新的设备特殊文件后才能使用这个模块（如果模块每次加载时返回的主编号都不同的话）。因为这个原因，我们还是使用静态主编号吧。动态注册实现起来并不难，它应该和下面这段代码差不多：

```
int major = 0, res;

res = register_chrdev(major, schar, &schar_fops);
if (res < 0) {
   MSG("can't register device with kernel\n");
   return res;
} else {
   major = res;
   MSG("device registered with major %d\n" , major);
}
```

这一章的所有设备示例使用的都是静态主编号方法，我们把向动态分配方法的转换留给读者做练习。

在成功调用register_chrdev之后，设备注册到了内核，设好的文件操作结构添加到了字符切换表里。在/proc/devices的字符设备部分也应该能够看到我们的设备名了。

21.3.3 模块的使用计数

内核需要记录加载到系统里的每一个模块的使用情况。如果不是这样，它就无法知道什么时候移掉一个模块是安全的。如果在你正在向硬盘拷贝文件的时候一个硬盘驱动程序从内核里被去掉了，会有什么样的后果？它会给硬盘带来灾难并使文件系统处于一个不可靠的状态。大家肯定不希望发生这样的问题。

修改模块使用计数器需要用到两个宏命令，它们一个是MOD_INC_USE_COUNT，另一个是MOD_DEC_USE_COUNT。前者给计数器加上“1”，而后者给它减去“1”。维护模块使用计数器的责任完全落在编写驱动程序的程序员身上，他既要保证模块不会被意外地加载，还要保

证当模块不再被使用时能够安全地从内核里去掉。MOD_IN_USE宏命令能够求出模块当前的使用计数，但这几乎没有什么必要，因为内核自己在尝试去掉一个模块之前是会对这个数字进行检查的。

21.3.4 open和release：设备的打开和关闭

现在模块已经被加载，它会在系统上等待有人打开与之关联的设备。当设备被一个进程打开的时候，schar_open将被调用。模块的使用计数就是在这里得到增加的，这可以防止设备在忙于工作的时候被意外地从内核里删除掉。请看下面这段代码：

```
static int schar_open(struct inode *inode, struct file *file)
{
   /* increment usage count */
   MOD_INC_USE_COUNT;
```

这告诉内核现在至少已经有一个进程在使用着这个模块了。如果此时有用户执行了一个“rmmod module_name”命令，内核就会对所有使用计数大于零的模块（即它们正在使用中）返回一个“-EBUSY”错误。保持使用计数处于最新状态的责任完全在于模块自己。我们在schar_open里增加使用计数以保证有进程打开模块的时候它不会被意外删除。当有进程关闭设备的时候，我们再减少使用计数以保持其平衡。

传递给schar_open的file参数是内核对返回给应用程序的文件描述符的一个内部描述。file结构里有关于设备打开模式的信息。下面这个代码段可以用来测试设备是不是以读方式打开的，如下所示：

```
if (file->f_mode & FMODE_READ) {
   MSG("opened for reading\n");
...
```

file结构里还有下一个读操作发生位置等其他信息。下面给出的是该结构与我们这个模块有关的东西（当然还有许多其他的数据项）：

```
struct file {
...
mode_t          f_mode;
loff_t          f_pos;
unsigned int    f_flags;
...
};
```

f_mode是表示的打开模式，可以是读、写或者读写。f_flags提供了更多的信息，其中有一些是与设备模块有关的，比如O_NONBLOCK或O_NDELAY等。这些标志是从用户空间应用程序传递给open调用的，它们控制着设备在读和写方面的操作行为。所有有关资料都可以在asm/fcntl.h文件里查到。f_pos是将要进行读写操作的下一个位置。

我们再回到schar_open函数上来，下一步是设置定时器——我们将在下一小节对它进行详细的讨论。schar_open最后会把被打开设备的主、辅编号打印出来，这两个编号是从传递给它的inode那里提取出来的。我们现在还用不着它们，但在后面对模块进行检查的部分将利用这个信息区分这个驱动程序的不同实例。我们在第3章里介绍过inode的概念，它标识出磁盘或内存中的

文件，里面有属主、访问时间、长度、文件系统类型等许多有用的信息。stat系统调用可以把inode里的信息提取出来。如果对一个文件执行stat命令，就可以查到诸如这个inode提供了什么数据之类的许多线索（stat命令是stat系统调用的内核外对应事物）。详细资料请查阅linux/fs.h文件——注意文件系统的类型是如何在一个union类型的数据结构u里被定义的。Linux支持许多种文件系统！

schar_release除了将使用计数减1之外没有其他事情可做。也没有什么事情需要它做，这是因为Schar设备在每次被打开时既不保有通过malloc分配到的内存，也没有需要刷新的其他状态信息。

```
static int schar_release(struct inode *inode, struct file *file)
{
        MOD_DEC_USE_COUNT;
        MSG("schar_release\n");
        return 0;
}
```

21.3.5 文件操作read：从设备读出数据

schar_read函数假装从设备读取了数据（实际上它是从一个内部数据队列里读到的），然后把数据传递到用户空间去。Schar用一个全局变量对数据队列进行记录：当有数据可用的时候，变量schar_pool里保存的是设备可以供应的数据字节数。这就意味着如果在调用函数schar_read的时候变量schar_pool是零或是一个负数，读进程就必须转入非活跃状态直到有数据可读为止。

```
static ssize_t schar_read(struct file *file, char *buf, size_t count,
          loff_t *offset)
{

   /* if less data than requested is here, put process to sleep */
   while (count > schar_pool) {
      MSG("putting process with pid %u to sleep\n", current->pid);
      /* go to sleep, but wake up on signals */
      interruptible_sleep_on(&schar_wq);
      if (signal_pending(current)) {
         MSG("pid %u got signal\n", (unsigned)current->pid);
         /* tell vfs about the signal */
         return -EINTR;
      }
   }

   /* copy the data from our buffer */
   if (copy_to_user(buf, schar_buffer, count))
      return -EFAULT;

   schar_pool -= count;
   schar_data_read += count;

   MSG("want to read %u bytes, %ld bytes in queue\n", (unsigned)count,
                        schar_pool);

   /* return data written */
   file->f_pos += count;
   return count;
}
```

这看起来挺唬人，我们来一步一步地讲解它。count变量是读进程请求的数据总量。如果count大于schar_pool，这个读请求就无法满足，我们就必须把读进程挂起来直到它请求的数据充

足为止。interrupible_sleep_on的作用是改变进程的状态并把它添加到schar_wq等待队列里去。我们很快就会讲到等待队列了——它向我们提供了这样一种手段：把对设备进行读操作的进程设置为一个非执行状态，再通过调度器程序选择一个新进程去执行。来自我们内部缓冲区的数据在copy_to_user的帮助下拷贝回应用程序，这个概念我们也要过一会儿才能详细讨论。

在schar_read函数的结束部分，我们从缓冲池里（以字节计算的长度保存在schar_pool变量里）减去count字节，然后返回已经读取的字节数。在这个例子里返回的将是count，这是因为我们假定此时已经精确地写了应用程序所请求的那么多字节。它与dd命令合作的很好，因为我们可以指令dd准确地请求读取我们需要的字节数。我们一会还会在Schar设备上试试cp命令，到那时大家就会明白我们的意思了——在我们的系统上，cp以4KB的数据块为单位请求数据直到读操作返回0字节为止。也就是说，你可以向Schar设备提供尽可能多的数据，而cp命令则永不满足地请求更多的东西。

如果你现在对schar_read函数的许多地方还弄不明白，请不要着急。我们将在后面这几个小节里把它的各个部分分开来讲解。

21.3.6 current任务

在schar_read函数里的好几处地方出现了current宏定义。这个宏定义代表着当前运行中的进程，是一个任务结构的数据项。也就是说，我们在schar_read里处理的那个叫做current的东西代表的就是正在做读设备操作的当前进程。任务结构有许多元素——它完整的清单可以在linux/sched.h头文件里查到。下面是我们直接或间接地用到了的几个数据域：

```
struct task_struct {
   volatile long state;
   ...
   int sigpending;
   ...
   pid_t pid;
   ...
   sigset_t signal, blocked;
   ...
};
```

正如大家看到的，系统里的所有任务其实是链接在一个双向链接列表里的。state给出的是进程的当前状态，即它是正在运行、被停止、还是正在被切换。我们将在讨论等待队列的那一小节对这一问题组深入探讨。pid是该进程的程序标识码。signal保存着与发送给该进程的一切信号有关的信息，而blocked是进程本身决定屏蔽的信号掩码。最后，sigpending保存着与是否有一个非阻塞信号发送给了该进程有关的信息，signal_pending函数检查的就是这个变量。因此，如果signal_pending返回的是真值，我们就通知VFS并让它重新开始数据传输过程。

21.3.7 等待队列

当没有数据可读的时候，我们利用等待队列把current任务设置为休眠状态；当有新的数据拷贝到Schar设备来的时候，我们再把它唤醒。这就解放了系统，使它可以去运行其他的进程。操作系统的进程调度器在我们唤醒任务之前是不会去考虑它的，而我们只有在使它休眠的条件得

到了满足的前提下才会去唤醒它。这使内核代码对那些访问自己的用户空间应用程序拥有了极大的控制权。

schar_read就使用了这个技术，我们过一会儿还会再看到它。现在先来看看它的具体实现过程。下面是等待队列的数据结构定义：

```
struct wait_queue {
   struct task_struct *task;
   struct wait_queue *next;
};
```

这个定义没有什么好讲的，它的构造和作用是很明显的。task元素把被“催眠”进程的有关信息保存在一个任务结构里，而next是一个指向等待队列中下一个条目的指针。很明显，在了解了等待队列的构造之后，你就可以让任意多个进程休眠在某个等待队列上。那么，怎样才能让进程休眠呢？

```
void interruptible_sleep_on(struct wait_queue **p)
long interruptible_sleep_on_timeout(struct wait_queue **p, long timeout)
```

这两个宏定义的作用是把进程“催眠”为某个状态，但允许它们被信号唤醒。_timeout变体在内部调用了schedule_timeout，使作为其调用参数的等待队列能够让进程在时间到了的时候自动苏醒。我们将在后面讨论定时器的部分介绍如何设定倒计时时间。

```
void sleep_on(struct wait_queue **p)
long sleep_on_timeout(struct wait_queue **p, long timeout)
```

这两个宏定义的语法和上面两个函数是完全一样的，只不过进程的休眠状态被设置为了TASK_ININTERRUTIBLE。

如果读者有兴趣做深入研究，可以在kernel/sched.c里查到它们。

一个休眠中的进程迟早需要被唤醒，唤醒的办法也不外两种：

```
wake_up_interruptible(struct wait_queue **p)
wake_up(struct wait_queue **p)
```

这是由__wake_up延伸而来的两个宏定义。前一个只唤醒可中断休眠进程，而后一个可以唤醒两种状态的休眠进程。但明确地被停止了执行（比如发送一个SIGSTOP信号）的进程将不会被唤醒。

当Schar没有数据可提供时，它会把读数据进程催眠在自己的等待队列上。而当有一个写数据进程提供了足够多的数据，从而能够满足进程的请求时，休眠进程将被一个定时器唤醒。

```
interruptible_sleep_on(&schar_wq);
if (signal_pending(current))
            return -EINTR;
```

这个结构在内核的许多地方都可以看到。我们把current进程催眠了，但允许它在信号的触发下苏醒过来。在interruptible_sleep_on成功之后，进程或者因为wake_up_interruptible调用而被唤醒，或者因为接收到一个信号而苏醒。如果是后一种情况，signal_pending先要返回“1”，我们就会利用返回一个“-EINTR”错误的办法来激发一个中断调用，VFS会根据这个中断重新启动读数据进程。如果我们使用sleep_on函数简单地催眠了进程，进程就会进入不可中断的休眠以等

待数据；在这种情况下，即使是一个SIGKILL信号也不能消除它。

就象我们前面介绍的那样，我们在定时器处理器和schsr_write两个地方分别唤醒相应的读数据进程。

```
wake_up_interruptible(&schar_wq );
```

这个调用会把休眠在队列里的所有读数据进程都唤醒。这样做是否合理要视具体情况而定——因为我们只能满足一个读进程，所以我们是否应该唤醒所有的读数据进程让它们竞争数据呢？Schar可以通过为设备的每次打开分别建立一个等待队列的办法来解决这个问题。这样做并不涉及什么新概念，所以我们把它做为练习留给有兴趣的读者去完成。我们可以让各个设备拥有自己的数据，这是个既有趣又有用的概念，我们将在后面通过lomap模块演示这个问题的解决方案。

21.3.8 文件操作write：向设备写入数据

相对而言，schar_write就简单的多了，它给schar_pool增加count个字节，再修改file->f_pos的值以反映出有多少数据被写到了设备（实际上是读到设备的内部缓冲区里）。之所以说它比读操作简单的多是因为Schar不需要对写到设备里来的这些数据进行处理，它只要张开大口把我们扔给它的数据照单全收就行了。除此之外，我们只有一件事可做，那就是唤醒读数据进程，因为这个时候可能已经有足够的数据能够满足它们的读数据请求了。下面是这个函数的实现代码：

```
static ssize_t schar_write(struct file *file, const char *buf,
   size_t count, off_t *offset)
{
   schar_pool += count;
   schar_data_written += count;

   /* if we were really writing - modify the file position to
      reflect the amount of data written */
   file->f_pos += count;
   if (copy_from_user(schar_buffer, buf, count))
      return -EFAULT;

   /* wake up reading processes, if any */
   if (schar_pool > 0) {
      wake_up_interruptible(&schar_wq);
      wake_up_interruptible(&schar_poll_read);
   }

   MSG("trying to write %u bytes, %ld bytes in queue now\n",
               (unsigned)count, schar_pool);

   /* return data written */
   return count;
}
```

21.3.9 非阻塞性读操作

提供数据服务的驱动程序必须区分阻塞和非阻塞两种打开方式。如果没有足够的数据满足一个请求，我们惯常的做法是使进程进入休眠状态；而一旦有了足够的数据，就再把它唤醒。但如果设备是以非阻塞方式打开的，我们就不能象刚才那样做了：我们必须尽可能多地供应数据，而进程在没有数据可读时会立刻返回而不是进入休眠状态。给schar_read函数加上下面阴影

部分里的代码就可以实现非阻塞读操作了：

```
static ssize_t schar_read(struct file *file, char *buf, size_t count,
          loff_t *offset)
...
   while (count > schar_pool) {
      /* if the device is opened non blocking satisfy what we
         can of the request and don't put the process to sleep. */
      if (file->f_flags & O_NONBLOCK) {
         if (schar_pool > 0) {
            file->f_pos += schar_pool;
         if (copy_to_user(buf, schar_buffer, schar_pool))
            return -EFAULT;
         count = schar_length;
         schar_pool = 0;
         return count;
      } else {
         return -EAGAIN;
      }
      MSG("putting process with pid %u to sleep\n", current->pid);
      /* go to sleep, but wake up on signals */
      interruptible_sleep_on(&schar_wq);
      if (signal_pending(current)) {
         MSG("pid %u got signal\n", (unsigned)current->pid);
         /* tell vfs about the signal */
         return -EINTR;
      }
   }
}
...
}
```

schar_read检查f_flags以确定设备当前是以什么方法打开的。如果应用程序请求的数据比我们数据池里现有的要多，我们就先把里面有的返回给应用程序，等数据池空了时再返回一个“-EAGAIN”错误。这等于暗示读数据进程应该过一会儿再来试试自己的请求。

如果一个设备驱动程序只实现了read和write未免缺少点活力。文件操作结构可还提供了不少其他的入口点呢——我们决定在Schar设备上实现其中的一些，并且要向大家说明为什么有的文件操作即使我们没有在驱动程序里写出一个明确的处理函数也能够执行一定的动作。

21.3.10 查找操作

Schar没有实现自己的查找函数，因此它要依赖于内核里实现的缺省查找功能。如果在传递给register_chrdev函数的文件操作结构里把llseek注册为NULL，就表明查找操作将调用内核里的缺省实现来完成。内核版本的查找操作可以在fs/read_write.c文件里找到，它的名字是default_llseek，这个调用提供了SEEK_SET、SEEK_CUR和SEEK_END三项功能。这些与llseek用在一起的宏定义的作用是修改file->f_pos的值，即当前文件流里将被读取的下一个字节的位置。如果你想自己处理查找操作（比如某个设备不支持相对于文件尾的查找操作时）就必须自行编写代码实现llseek调用。下面是一个典型的用法示范：

```
loff_t own_llseek(struct file *file, loff_t offset, int mode)
{
   switch (mode) {
      case 0:                                   /* SEEK_SET */
         file->f_pos = offset;
         return file->f_pos;
      case 1:                                   /* SEEK_CUR */
```

```
            file->f_pos += offset;
            return file->f_pos;
        case 2:                                    /* SEEK_END */
            return -EINVAL;
        default:                                   /* cannot happen */
            return -EINVAL;
    }
}
```

如果该设备上的查找操作根本没有意义，就必须定义一个llseek函数以阻止查找。此时我们只需简单地返回一个“-ESPIPE”，它的意思是“查找操作非法”，即这个设备没有查找操作。

```
loff_t own_llseek(struct file *file, loff_t offset, int mode)
{
        /* illegal seek */
        return -ESPIPE;
}
```

21.3.11 文件操作ioctl：I/O控制

有时候，能够改变或者读取一个运行中驱动程序的参数是很有用的；要不然我们就得重新对它进行配置、编译和运行。但对某些驱动程序来说，如果它们在使用中没有间歇，我们也就不可能把它们从系统里拿下来，重新配置、编译和运行它也就无从谈起。而ioctl就是驱动程序里让我们能够在它运行时设置或检索其有关设置情况的入口点。

内核里的每个设备都对应地有一个独一无二（基本如此，但也有两块硬盘这样的例外）的ioctl基地址和一组设备命令。举个例子，SCSI主控制器的ioctl基地址是0x22，并且整个子范围0x00 - 0xff也都分配给了它。因为大多数设备并不需要支持多达256个命令，所以只用到了子范围的一小部分。16位的基地址构成ioctl命令的上半部分，而16位的设备命令构成了ioctl命令的下半部分——因此，SCSI主控制器的第一个命令就是0x2200。ioctl基地址都写在Documentation/ioctl-number.txt文档里，但我们写这本书的时候它已经很久没有更新了。这个文件还介绍了怎样才能为你自己的设备申请一个适当的设备命令范围。我们为Schar挑选了一个没人使用的基地址0xbb。底层实现可就要比我们这么简单的一选要复杂多了，我们还是不要去管它了。但如果读者确实对它感兴趣的话，它的定义可以在asm/ioctl.h文件里查到。

Linux能够区分四种类型的ioctl调用，它们是：直接命令、读、写、读和写。这是由模块里标识符的写法定义的。这几种类型的定义见表21-5：

表 21-5

_IO(base,command)	定义了中选命令。在发出ioctl调用时没有需要传递进出应用程序的数据。一个_IO类的ioctl调用将返回一个正整数（也就是说它不会被解释为一个错误）
_IOR(base,command,size)	一个从应用程序角度看的读操作ioctl命令。size是需要传回给应用程序的参数的长度
_IOW(base,command,size)	一个从应用程序角度看的写操作ioctl命令。size是从应用程序传来的参数的长度
_IOWR(base,command,size)	一个读写操作的ioctl命令。size是来回传递的参数的长度

此外，还有几个用来对待发送命令的合法性进行检查的宏定义。内核方面的编码机制把这个数据域划分为方向、长度和命令几个部分。这些信息可以用表21-6中的宏定义提取出来：

表 21-6

_IOC_DIR(command)	command命令的方向。根据上表中介绍的命令类型，这个方向可以是_IOC_NONE、_IOC_WRITE或_IOC_READ。对_IOWR类的命令，这个方向值是_IOC_WRITE与_IOC_READ的逻辑或结果
_IOC_TYPE(command)	这个数据项的ioctl基地址部分，Schar的是0xbb
_IOC_NR(command)	这个数据项的设备命令部分
_IOC_SIZE(command)	参数的长度——如果有参数的话

ioctl函数本身是在模块注册时提供给内核的file_operations结构里定义的。我们把Schar设备支持的ioctl命令定义在schar.h文件里。

```
#define SCHAR_IOCTL_BASE            0xbb
#define SCHAR_TOGGLE_DEBUG          _IO(SCHAR_IOCTL_BASE, 0)
#define SCHAR_GET_POOL              _IOR(SCHAR_IOCTL_BASE, 1, unsigned long)
#define SCHAR_EX_TIMER_DELAY        _IOWR(SCHAR_IOCTL_BASE, 5, unsigned long)
```

我们选择的基地址远离一切有用的设备，这样就不会引起任何冲突。在Schar里我们使用了三种类型的ioctl命令。SCHAR_TOGGLE_DEBUG切换打印或不打印调试信息，因为它是一个_IO类型的ioctl命令，所以不带参数。SCHAR_GET_POOL返回数据池里现有数据的字节长度，SCHAR_EX_TIMER_DELAY设置定时器的每次延时的“jiffies”（内核时基）个数并返回原来的设置值。“jiffies”是内核用来测量时间的一个基本单位，我们很快就会讲到它。

schar_ioctl函数几乎完全由一个switch语句构成，设备支持的ioctl命令就在这个语句里得到处理。如下所示：

```
static int schar_ioctl(struct inode *inode, struct file *file,
                       unsigned int cmd, unsigned long arg)
{
   /* make sure that the command is really one of schar's */
   if (_IOC_TYPE(cmd) != SCHAR_IOCTL_BASE)
      return -ENOTTY;

   switch (cmd) {
      case SCHAR_TOGGLE_DEBUG: {
         schar_debug = !schar_debug;
         return 0;
      }
      case SCHAR_GET_POOL: {
         if (put_user(schar_pool, (unsigned long *)arg))
            return -EFAULT;
         break;
      }
      case SCHAR_EX_TIMER_DELAY: {
         unsigned long tmp = schar_timer_delay;
         if (!capable(CAP_SYS_ADMIN))
            return -EACCES;
         if (get_user(schar_timer_delay,
               (unsigned long *)arg))
            return -EFAULT;
         if (put_user(tmp, (unsigned long *)arg))
            return -EFAULT;
         break;
      }
      default: {
```

```
            MSG("ioctl: no such command\n");
            return -ENOTTY;
        }
    }
    /* to keep gcc happy */
    return 0;
}
```

如果读者曾经使用过ioctl系统调用，就能轻松地看懂这段代码。这里出现了两个我们以前没有见过的调用——get_user和put_user，我们稍后再向大家说明它们的细节，现在只要知道它们的功能是在用户空间和内核空间之间来回拷贝数据就可以了。default子句负责处理一切Schar设备不支持的ioctl命令。而“-ENOTTY”对设备不支持的ioctl命令来说是个很合适的返回值。这个错误的本意是“不是一台打字机”——那个年代算的上是设备的只有电传打字机；如今用在ioctl调用的返回值里时，它的含义是“此ioctl命令不适用于这个设备”。

21.3.12 检查用户权限

只要用户拥有打开某个设备的权限，他就可以对该设备进行ioctl调用。但并不是每个用户都能使用全部的命令，这要根据将要执行的具体操作来定。SCHAR_EX_TIMER_DELAY命令需要检查打开设备的人是不是超级用户，因为如果倒计时的时间值设置得过低会使定时器很快就会结束计时，因而使机器没有足够的时间来完成任何有意义的工作。Linux定义了许多与用户个人权限有关的设置项目，最适合本例的是CAP_SYS_ADMIN。各种CAP_类定义都可以在linux/capability.h文件里查到，里面还有对各种允许执行的操作的解释。请看下面这个函数定义：

```
int capable(int cap)
```

capable的作用是检查用户的权限，如果用户拥有与参数cap对应的能力，它将返回“1”，否则返回“0”。这个用法并不仅仅局限于ioctl，它们在内核里使用的相当频繁。

21.3.13 文件操作poll：设备对进程的调度

文件操作poll指的是设备对应用程序进程的调度功能，它提供了一种让应用程序进程休眠在设备上等待特定事件发生的机制。请不要把它与对设备状态的反复检查混为一谈，这两者完全是两回事。使用poll系统调用是避免繁忙循环却又能够等待事件发生的有效手段。Schar设备在一定程度上实现了进程调度功能，正好适用于我们的示例。poll功能的实现是很简单的。因为对Schar设备的写操作总是成功的，所以我们只需要检查读操作时的情况。这里引入了一个schar_poll_read等待队列。如果我们的数据池是空的，读数据进程就将休眠直到有足够的数据可供读取为止。

```
static unsigned int schar_poll(struct file *file,
                               poll_table *wait)
{
    unsigned int mask = 0;

    poll_wait(file, &schar_poll_read, wait);

    /* if the pool contains data, a read will succeed */
    if (schar_pool > 0)
        mask |= POLLIN | POLLRDNORM;
```

```
    /* a write always succeeds */
    mask |= POLLOUT | POLLWRNORM;

    return mask;
}
```

这就是Schar设备里进程调度的全部工作！如果Schar设备的数据池长度还有一个上限，也很容易再给它加上一个schar_poll_write等待队列——只要对POLLOUT做类似的检查就可以做到。头文件asm/poll.h里包含着各种可能的poll掩码，我们把其中一些标准的掩码列在表21-7里。

表 21-7

POLLIN	设备可以非阻塞地向后续的读操作提供数据
POLLRDNORM	
POLLOUT	设备可以非阻塞地从后续的写操作接受数据
POLLWRNORM	
POLLERR	出现一个错误

如果读者曾经在用户空间应用程序里使用过poll功能，就会比较熟悉这些掩码的功用，因为从内核看去它们是很相似的。

动手试试：对Schar设备进行读写

好了，Schar设备的基本组成都介绍完了——我们马上要开始对定时器和内存方面的问题进行议论了。但在继续学习之前，我们最好先动手试试，看它在使用中有什么样的表现。大家可以从Wrox出版社的Web站点www.wrox.com下载到Schar设备驱动程序的源代码。你可以在自己系统里的任何位置释放下载到的档案文件，Schar文件将释放到它的下级子目录modules/schar里。

1) 我们首先要为Schar设备正确创建一个特殊文件。我们在开始时介绍的mknod命令就是干这个用的。我们需要创建一个字符特殊文件，它的设备主编号是42，辅编号是0。

```
# mknod /dev/schar c 42 0
#
```

2) 进入保存着Schar模块源代码的子目录，输入执行make命令。编译结束时应该没有错误或警告。结果模块也在同一个子目录里，名字是schar.o。用下面的命令把模块插入到内核：

```
# insmod schar.o
#
```

设备现在已经注册到内核，可以提供服务了。查看/proc/devices文件证实一下，这个方法我们前面解释过。或者输入dmesg命令，看看Schar是怎样欢迎我们的。

```
$ dmesg | tail -n1
schar: module loaded
$
```

3) 现在向Schar设备拷贝一个文件，看看会出现什么情况。你可以任意选择拷贝文件，但如果方便的话，最好先用一个小文本文件试试，要不就为此专门创建一个好了。最好另外打开一个终端窗口，在里面重复执行dmesg命令查看Schar的输出。我们将把dmesg命令的输出列在每一个命令的下面。

如果cp命令询问是否覆盖/dev/schar文件，直接回答yes。

```
$ cp small_file /dev/schar
schar: opened for writing
schar: major: 42 minor: 0
schar: trying to write 4096 bytes, 4096 bytes in queue now
schar: trying to write 4096 bytes, 8192 bytes in queue now
schar: trying to write 3241 bytes, 11433 bytes in queue now
schar: release
```

我们先进入的是schar_open，头两行信息就是它输出的。接下来进入schar_write。我们以每次拷贝4096个字节的方式拷贝了总共11433个字节。写操作结束后，进入schar_release，文件的写操作就全部完成了。

4) 现在Schar设备里已经有数据可读了，我们再把它拷贝出来。

```
$ cp /dev/schar out_file
schar: opened for reading
schar: major: 42 minor: 0
schar: want to read 4096 bytes, 7337 bytes in queue
schar: want to read 4096 bytes, 3241 bytes in queue
schar: putting process with pid 757 to sleep
schar: pid 757 got signal
```

这一次的入口点还是schar_open，但接下来进入的将是schar_read。数据以4096个字节为单位又被读了出来——在数据池中的数据少于cp命令所请求的之前已经成功地执行了两次每次4KB字节的读操作。然后，我们让读数据进程（即cp命令）进入休眠状态，直到有更多数据可读为止。我们后来失去了耐心，用“Ctrl-C”结束了读进程。“Ctrl-C”向进程发送了一个信号，结果是产生了最后一行输出。

我们在上面例子里提供数据时每次是4KB字节，但要注意这并不是Schar设备规定，而是cp命令的设计安排。我们完全能够以每次1字节或16KB字节的方式提供数据。

现在，我们既可以向Schar写入数据，又可以把数据读回来了。下面可以开始研究这个设备驱动程序里的其他函数了。

动手试试：ioctl

1) 确定Schar模块已经建立并加载。我们来在保存着Schar源代码的子目录里创建一个名为schar_ioctl.c的文件。

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#include "schar.h"

int main(int argc, char *argv[])
{
        int fd = open("/dev/schar", O_RDWR);

        /* complain if the open failed */
        if (fd == -1) {
                perror("open");
                return 1;
        }
```

```
        /* complain if the ioctl call failed */
        if (ioctl(fd, SCHAR_TOGGLE_DEBUG) == -1) {
                perror("ioctl");
                return 2;
        }

        printf("Schar debug toggled\n");

        return 0;
}
```

2) 用下面两个命令分别编译和执行这个程序：

```
$ cc -o schar_ioctl schar_ioctl.c
$ ./schar_ioctl
Schar debug toggled
$
```

3) 如果在加载Schar时激活了调试功能（这是它的缺省设置），它现在会停止输出打开、读数据等操作时的通知性信息了。用刚才的cp命令再试试，然后检查dmsg命令输出。你将看到调试信息将不再输出到内核缓冲区里去了。

21.3.14 模块的参数

Linux提供了一个简单的办法使我们能够给模块添加上一些参数，这些参数可以在模块被modprobe或insmod命令加载的时候给出来。Schar的大多数用户定义选项是用下面的办法安排设置的：

```
MODULE_PARM(variable, type)
```

variable是准备设置的变量参数，而type是由一个长度值和该变量的类型组成的字符串。它们可以放在模块里各函数以外的任何地方，但通常都会被放在文件开始全局变量的定义和声明部分的后面。我们来看一个Schar模块里的例子：

```
MODULE_PARM(schar_debug, "1b" );
```

类型字符串“1b”表示这是一个长度为1的字节值。我们还可以设置长度值的最小和最大尺寸。比如说，“5-10b”表示这是一个字节尺寸的数组，它最少有5个元素，最多有10个元素。如果没有给出长度数字，就认为其长度是1。在字节尺寸之外还定义了四种其他的类型，如表21-8所示：

表 21-8

h	short，短整数
i	int，整数
l	long，长整数
s	string，字符串

还可以再给模块的各种参数加上一些描述性的文字。这个信息可以被modinfo命令提取出来，让用户不需要研究模块的源代码就能够有机会了解模块都支持了哪些参数。

```
MODULE_PARM_DESC(schar_debug, "Enable debugging messages");
```

上面这一行描述了schar_debug变量的含义，这个描述会跟在MODULE_PARM声明内容的后面显示给用户。最后，填写一些整体性的介绍文字和程序作者，如下所示：

```
MODULE_DESCRIPTION("Sample character driver");
MODULE_AUTHOR("Jens Axboe");
```

modinfo命令有几个命令行选项，你可以利用它们提取出自己需要的信息来。需要提醒大家的是modutils-2.1.121里有一个缺陷，它会使信息一打印起来就循环个没完没了。所以我们建议大家弄个新一点的版本来。

动手试试：modinfo命令

我们来动手试试，把保存在Schar编译结果模块里的模块参数信息列出来。切换到保存着Schar模块的子目录，然后输入如下所示的命令：

```
$ modinfo -p schar.o
schar_name string, description "Name of device"
schar_debug byte, description "Enable debugging messages"
schar_inc int, description "Byte increase in pool per timer fire"
schar_timer_delay long, description "Timer ticks between timer fire"
schar_pool_min long, description "Timer fill pool minimum in bytes"
```

“-p”选项的作用是让modinfo把可以在模块加载时传递给它的参数的资料打印出来——这包括它的名字、类型以及各个选项的描述文字等。

21.3.15 proc文件系统接口

proc文件系统工作起来和一个真正的文件系统非常相似，它也需要挂装，也要用标准的文件操作命令来读写。从proc文件系统的某个文件里读出来的数据是由模块或内核即时生成的，我们可以通过它们了解到系统运行时的统计资料和其他有关信息；而可写数据项可以用来改变驱动程序的配置或行为。因此，为Schar设备在proc文件系统里添加一个条目将使我们能够直接从运行着的模块那里检索信息，不再需要编写程序来发出ioctl命令。

1. sysctl系统调用

注册一个条目的最佳地点是/proc/sys子目录下。这样该条目就可以用cat等文件类命令或者sysctl系统调用来检索了。设备需要把自己注册到/dev子目录里去，我们的Schar就是这样做的——它创建了一个名为“Schar”的下级子目录，在这个子目录里又创建了一个名字是“0”的条目。注册条目是在一个由许多ctl_table结构组成的数组里定义的，这些结构依次叠加，直到整条注册链结束为止。这个注册链从根开始（即/proc/sys/dev），依次是下级子目录注册条目/proc/sys/dev/schar和最终的注册项/proc/sys/dev/schar/0。下面是Schar.c文件里与此有关的一段代码：

```
/* sysctl entries */
static char schar_proc_string[SCHAR_MAX_SYSCTL];
static struct ctl_table_header *schar_root_header = NULL;
static int schar_read_proc(ctl_table *ctl, int write, struct file *file,
           void *buffer, size_t *lenp);

static ctl_table schar_sysctl_table[] = {
```

```
    { DEV_SCHAR_ENTRY,                  /* binary id */
      "0",                              /* name */
      &schar_proc_string,               /* data */
      SCHAR_MAX_SYSCTL,                 /* max size of output */
      0644,                             /* mode */
      0,                                /* child - none */
      &schar_read_proc },               /* set up text */
    { 0 }
};

static ctl_table schar_dir[] = {
    { DEV_SCHAR,                        /* /proc/dev/schar */
      "schar",                          /* name */
      NULL,
      0,
      0555,
      schar_sysctl_table },             /* the child */
    { 0 }
};

static ctl_table schar_root_dir[] = {
    { CTL_DEV,                          /* /proc/dev */
      "dev",                            /* name */
      NULL,
      0,
      0555,
      schar_dir },                      /* the child */
    { 0 }
};
```

DEV_SCHAR_ENTRY是发出sysctl系统调用时使用的二进制ID编号，它必须是这个设备所独有的。schar_proc_string是返回给读这个注册项的进程的数据，它的字节长度不能大于定义常数SCHAR_MAX_SYSCTL的取值。具体实现过程是很简单的，把即时生成的统计信息拷贝到一个专用的缓冲区（即schar_proc_string字符串）就完事了，这个用做缓冲区的字符串是由schar_read_proc在这次读注册项操作发生时生成的。

把这些表都设置好以后，这个条目就可以注册到proc文件系统上去了。注册操作将返回一个ctl_table_header结构；如果内存不够，它会返回“0”——但这种情况极其少见。

```
schar_root_header = register_sysctl_table(schar_root_dir, 0);
schar_root_dir->child->de->fill_inode = &schar_fill_inode;
```

最后一行设置了fill_inode。在进入和离开/proc/sys/dev/schar子目录的时候，赋值语句右边的函数schar_fill_inode将对模块的使用计数做相应的增减。一个fill参数会传递到这个函数指明与这个子目录关联着的inode结点是否正有人使用。增加使用计数的目的是使模块在这个子目录被访问期间不会被移出内核，要不然就会制造出一个内核错误（也叫做“Oops”，我们将在调试部分讨论这些问题）。

proc_dostring为我们完成其余的工作，把数据拷贝到用户空间并对文件的位置做相应的修改。它不会对参数本身做什么修改，只是简单地把它们传递过去而已。

```
return proc_dostring(ctl,write,file,buffer,lenp);
```

从对内核中的proc和sysctl的支持方面看，Schar并不是很细心。如果这是一个成品模块，就必须给这段代码实现加上一些必要的定义检查以确保安全。也就是说，我们需要用下面这个框架把有关代码括起来：

```
#ifdef CONFIG_SYSCTL
```

```
...
#endif
```

这就起到了定义检查的保护作用。否则，如果我们使用的内核在建立时没有带上sysctl支持——但这种情况是极其少见的，这个模块就会编译失败。

2. 可写数据项

Schar注册的条目只向用户提供信息，也就是说，它们是只读性的。但我们设置的模式其实是允许执行写操作的。在下面的代码里，用户写入的字符串将被简单地输出到系统日志记录里去。如下所示：

```
if (write) {
   char *tmp = (char *) get_free_page(GFP_KERNEL);
   MSG("proc: someone wrote %u bytes\n", *lenp);
   if (tmp) {
      if (!copy_from_user(tmp, buffer, *lenp))
         MSG("proc: %s", tmp);
      free_page((unsigned long)tmp);
      file->f_pos += *lenp;
   }
   return 0;
}
```

一个write参数被传递到注册了的处理器，它指明了这次访问进行的是读操作还是写操作。如果是后一种情况，写入数据就会拷贝给我们并打印出来。

要增加一个函数来分析输入并对配置情况做相应的调整并不困难。另一个办法是创建许多个条目，每个条目对应一个Schar的配置选项（就象/proc/sys条目一样），这就使用户能够简单地使用echo命令直接修改各个“文件”；甚至还可以用proc_dostring来同时完成输入和输出。sysctl调用的信息提取功能在这一方面走得更远，但对这两种方法的学习将为你今后对proc文件系统进一步的研究准备好必要的工具。这其实是个合二为一的买卖，因为注册sysctl表会同时照顾到设置相应的proc文件系统条目。文件kernel/sysctl.h是查找进一步资料的好地方。

21.3.16 Schar的执行情况

Schar把schar_pool变量用做一个内部的计数器，它的值对应着设备供应或者请求的字节个数。一个负数值表示读数据进程需要schar_pool个字节的数据，而一个正数值则表示有schar_pool个字节等待读数据进程读取。对设备进行的写操作将给数据池增加一定数量的字节以备随后的耗用。Schar里有一个持续运行的内核定时器，在这个定时器的控制下，每经过SCHAR_TIMER_DELAY的时间间隔就给schar_pool增加SCHR_INC个字节；而在这期间，一个读数据进程将等待有数据可读。对设备进行的写操作永远会成功；可如果某个读数据进程请求的数据比数据池里的现有数据还要多的话，它就将被添加到Schar的等待队列里，等待队列里的进程会在定时器处理器每次运行的时候被唤醒一次。它把相应的信息输出到系统的日志记录里。大家不妨把这个模块插入到内核里试试，亲身体验一下它的工作情况。

Schar用一个空闲的内存页面做为自己的内部缓冲区，读出和写入的数据都是从那里拷贝进出的。这个页面除了演示数据在内核空间与用户空间之间的移动情况外没有什么其他的特殊目的，内存方面的问题我们一会儿还会做进一步讨论。

Schar本身并没有什么真正的用处，它只是一个字符驱动设备的驱动程序的基本框架。我们利用一个内核计时器向数据池里馈送数据，通过这个办法模仿一个字符设备的调度情况。如果设备不能以中断的形式向驱动程序提供反馈，就必须要使用调度。中断概念我们将在中断驱动的字符设备部分进行讨论。

21.3.17 小结

在这一节里，我们学习了字符设备是如何通过register_chrdev函数把自己注册到内核的设备切换表里去的。它们给这个函数传递去一个file_operations结构，而这个结构定义了我们在驱动程序里都提供了哪些访问方法。模块的使用计数确保驱动程序在有人使用时不会被卸载，它的取值要用MOD_INC_USE_COUNT和MOD_DEC_USE_COUNT宏命令来改变。

我们学习了Schar实现的读文件操作和写文件操作，学习了如何利用等待队列对当前进程进行“催眠”和“唤醒”。这是一个非常重要的概念，在每一个字符设备驱动程序里都可以看到它的应用。

接下来，我们学习了驱动程序在读、写功能以外提供的其他文件操作入口点。第一个是通过llseek实现的设置读写位置功能；事实上，如果我们没有在模块里自己编写出能够实现这一功能的代码，llseek系统调用就会成为驱动程序在这方面的缺省调用功能。我们还介绍了如何禁止读写位置设置功能——编写一个没有什么实质内容且只返回“-ESPIPE”的实现。

下一个是ioctl命令，即I/O控制命令。我们知道怎样把调用分为两个部分：一个基地址和一个命令。Schar驱动程序里的ioctl命令处理差不多完全是用一个switch语句来实现的，如果用户应用程序调用了一个未知的ioctl命令，驱动程序仅返回“-ENOTTY”。并不是任何用户都能使用所有的ioctl调用。特定权限可以用capable函数来核查。

poll提供了一个使读数据进程在等待数据准备好的同时进入休眠状态的办法。Schar里的实现只对读数据进程进行检查，因为它随时能够接受写数据进程写来的数据。虽然在Schar里只测试读数据进程，但我们把其他调度掩码也介绍给了大家。

再往后，我们简单地介绍了设定模块参数方面的知识，然后学习了如何在Schar里设置proc文件系统的注册项目。我们介绍了完整的设置过程，并且给出了一个sysctl设置项——它既可以通过读取/proc/sys/dev/schar里的项目来查看，也可以通过sysctl系统调用来查看。

最后，我们简单地介绍了数据拷贝进出Schar设备时它的执行情况。我们建议大家自行编译这个模块并运行它试试，这样才能全面掌握它的作用和原理。

21.4 定时和时基：jiffies变量

内核把时间计数保存在全局性的jiffies变量里。你可以把它想象为内核的心跳，时间每流逝一个单位时间（即时基），jiffies就增加一个值。常数定义“HZ”控制着时间计数的频率，它是在头文件asm/param.h里定义的，你可以按自己的意愿修改它。但修改这个常数需要重新编译所有现有的模块（当然也包括内核），并且会严重影响依赖于“HZ”缺省值的应用程序。除了采用Alpha体系结构的计算机以外，其他各种平台都把这个值设置为100，也就是说，每个时间计数代表着10毫秒。增大这个值将会以在进程调度器身上花费更多时间为代价提高系统的交互性能，

但让它保持为100可能是更安全的做法。许多人认为应该修改“HZ”常数，因为它们自Linux出现之日起就一直是这个值，从来没有改变过；这样的争论时不时地就会出现。但直到处理器芯片速度已经大幅度提高的今天，增加这个值能否改善计算机的性能仍然没有定论。

在编写设备驱动程序的时候，你可以会遇到需要检测某个操作是否已经在给定的时间里完成了的情况。有些设备不支持在操作完成时产生中断，它们必须通过针对其状态而采取的某种形式的调度方案来处理。这通常只涉及到一些老式或设计不良的硬件产品；现代化的设备都应该支持某种形式的事件通知功能。繁忙循环在设备驱动程序的设计实践里从没有被认为是好的做法。如果你不得不使用这些“损招”，就一定只能用在设备检测或类似情况等老套路里。在这种情况下，最经常使用的程序框架都是下面这段代码的变体：

```
unsigned long timeout = jiffies + PROBE_TIMEOUT;
do {
   do_command(PROBE);
   stat = get_status();
} while (stat == BUSY && time_before(jiffies, timeout));

if (time_after_eq(jiffies, timeout))
   printk("operation timed out.\n");
```

你可能会对time_before和tiem_after_eq这两个函数的作用产生质疑，为什么不直接比较timeout和jiffies这两个值呢？是这样的，Linux是一个非常稳定的系统，在两次重启动之间计算机运行了相当长的时间并不希奇；可jiffies变量是被定义为一个不带正负号的长整数的，它终将会溢出归零，然后又从零重新开始计时。如果在对jiffies和PROBE_TIMEOUT做加法时出现了时间计数溢出归零的现象，那么下面这个简单的测试：

```
do {
   do_command(PROBE);
   stat = get_status();
while (stat == BUSY && (jiffies < timeout));
```

就会正确地退出循环，但这并不是正确的结果！这很明显是我们不希望看到的，而自行处理时间的溢出归零问题虽然并不困难，但终归是一件困扰驱动程序设计人员的事情。因为这个原因，人们设计了四个宏命令来为我们安全地做好这方面的工作见表21-9：

表 21-9

time_before(jiffies,x)	考虑了溢出归零问题的“jiffies < x”比较
time_after(jiffies,x)	考虑了溢出归零问题的“jiffies > x”比较
time_before_eq(jiffies,x)	考虑了溢出归零问题的“jiffies <= x”比较
time_after_ed(jiffies,x)	考虑了溢出归零问题的“jiffies >= x”比较

这几个宏命令不仅能够在上表这样的直接比较操作中（此时出现溢出归零现象的可能性是很小的）一显身手，在对驱动程序里比较大的时间计数值进行比较时（比如数据包被安排到一个队列里，在经过了很长的时间之后才进行“时间到”测试的情况）更是大放异彩。在SCSI卡或网卡驱动程序的代码里就可能出现这样的情况。这些宏命令是在linux/timer.h文件里定义的。

在考虑可移植性和使用jiffies进行计时的时候，永远不要想当然地认为常数“HZ”就是等于100。在Alpha体系结构上这就已经是一个错误的假设了，它把这个数字定义为1024；而rtLinux

（Real Time Linux，实时Linux）使用的则是10,000。用户可以自由选用自己喜欢的数值，改变“HZ”不过是修改asm/param.h文件再重新编译系统而已。因此，在上面这样的结构里定义PROBE_TIMEOUT的时候一定考虑“HZ”的设置情况，不能仅使用一个数值来表示时间。

21.4.1 短暂延时

有时候，你需要用到比“HZ”提供的精确度更高的时间控制手段。要记住，在大多数平台上“HZ”的取值是100，它只能给你大约0.01秒的分辨率。这段时间看起来并不长，但对大多数硬件来说可就长得很了。如果你确实需要更精确的时间分辨率，udelay可以提供最高到微秒级（10^{-6}）的精度。

```
/* delay for 0.5 milliseconds */
udelay(500);
```

udelay在计算时间时要用到系统开机引导时计算出来的“BogoMIPS rating”值，虽然你不可能得到百分之百精确的微秒级分辨率，但它确实是相当接近的。大家对开机引导时屏幕上出现的“BogoMIPS rating”值的含义和用途可能都不太了解——在开机引导时会执行一个基准性的循环，这个循环的每秒执行次数就是“BogoMIPS rating”值，这个值将被用来评估其他循环的执行速度。这个值与硬件速度并没有直接的关系（所以才叫它BogoMIPS，即为MIPS），但它确实是一个能够使udelay及其同类可以比较精确地提供延时操作的办法。千万不要用udelay进行超过1或2毫秒的延时，因为在“BogoMIPS rating”值非常高的机器上这样做可能会引起溢出。事实上，在使用这个办法之前一定要确定自己确确实实需要使用这种延时方法！如果你需要的延时比这个要长，可以采取一些变通办法。比如说，象下面这样循环一定次数，每次用udelay延时1毫秒：

```
/* delay for 10 milliseconds */
int i;
for (i = 0; i < 10; i++)
   udelay(1000);
```

或者使用mdelay，这是一个功能为延时1毫秒的宏定义。如下所示：

```
/* delay for 10 milliseconds */
mdelay(10);
```

21.4.2 定时器

内核定时器是驱动程序里需要使用时间值时的一个最佳选择。我们可以不考虑循环和繁忙等待等办法，利用现成的定时器函数就能隔过一段定量的时间，大家在上面也都看到过了，定时器还可以用来以固定时间间隔对设备进行调度。定时器被实现为一个双向链接列表，其格式如下所示：

```
struct timer_list {
   struct timer_list *next;
   struct timer_list *prev;
   unsigned long expires;
   unsigned long data;
   void (*function)(unsigned long);
};
```

这个定义可以在linux/timer.h里查到。为了避免不必要的复杂性，最好不要去改动*next和*prev指针；它们也用不着你改。两个指针根本就不是为程序员准备的，它们由内核在内部使用。在kernel/sched.c文件里可以查到它们一些有趣的用法。add_timer负责申请必要的操作锁，然后再调用internal_add_timer。我们不必操心internal_add_timer内部的具体工作原理，这使我们面对的程序设计接口非常简洁。我们稍后会在讨论SMP和可重入代码方面的问题时向大家介绍各种各样的加锁机制。

init_timer为我们对定时器进行初始化。它实际完成的操作只是把next和prev初始化为空指针，这样同一定时器的两次添加就会被注意到。我们在Schar的“open”过程里就使用这个技巧来检查它内部的定时器是否已经被初始化过了。我们对Schar的有关代码做个细致的分析，看看定时器是如何用在程序上下文里面的。下面是定时器的定义声明：

```
static struct timer_list schar_timer ;
```

init_module函数里的下面这条语句对定时器进行初始化，做好使用准备：

```
init_timer(&schar_timer );
```

schar_open函数设置定时器开始工作——前提是它没有被其他读数据进程设置过：

```
if (!timer_pending(&schar_timer)) {
   schar_timer.function = &schar_timer_handler;
   mod_timer(&schar_timer, SCHAR_TIMER_DELAY);
}
```

最重要的是定时器处理器schar_timer_handler函数，schar_timer定时器的function指针将指向这个函数。下面是它的代码清单：

```
static void schar_timer_handler(unsigned long data)
{
   /* setup timer again */
   if (schar_pool < schar_pool_min) {
      schar_pool += SCHAR_INC;
      schar_timer.expires = jiffies + schar_timer_delay;
      schar_timer.function = &schar_timer_handler;
      add_timer(&schar_timer);
      MSG("setting timer up again, %ld data now\n", schar_pool);
   }

   /* if the module is in use, we most likely have a reader waiting for
      data. wake up any processes that are waiting for data. */
   if (MOD_IN_USE && schar_pool > 0) {
      wake_up_interruptible(&schar_wq);
      wake_up_interruptible(&schar_poll_read);
   }

   return;
}
```

这段代码涉及到的语法现象并不复杂，我们就不对它做过多解释了。我们只用这个处理器函数负责管理了一个定时器，所以Schar里没有用到定时器的data参数。但我们可以让读数据进程和写数据进程同时使用一个定时器，然后通过data的值来表示出它正被用做读数据进程的定时器（SCHAR_READER）还是写数据进程的定时器（SCHAR_WRITER）。data的用法还有许多——它是做为一个参数传递到定时器的处理函数里去的，因此你可以按照自己的想法随意使用它。此外，如果不把expires与当前时间结合起来，定时器也就没有什么价值——你必须明确地给它和

jiffies变量做加法。

定时器会在它们失效时被自动检测出来，但如果模块被卸载后还把它们留在内核里处于挂起状态，就会造成内核的挂起——这是个挂起内核的好办法。这也意味着你可以在定时器处理函数里再次添加同一个定时器。cleanup_module调用timer_pending来检查定时器是否已经失效。如果定时器尚未失效，它返回true；否则返回false。仍在运行中的定时器可以用del_timer明确地删除掉，但很少有必要这样做；对已经失效的定时器调用del_timer不会产生什么副作用。很少会出现需要调用del_timer的情况，我们在这里介绍它的目的是为了把与定时器有关的各种函数都告诉大家。

如果你需要修改定时器的设置值以加大计时时间，就要使用mod_timer。不必使用下面这样的代码：

```
del_timer(&timer1);
timer1.expires = jiffies + new_value;
add_timer(&timer1);
```

你可以用下面这一条语句就完成了操作：

```
mod_timer(&timer1, new_value );
```

两种做法效果相同，可后一种更简单也更易于阅读理解。如果在调用mod_timer的时候定时器已经失效了，它的作用就象add_timer一样会让这个定时器再投入使用。

动手试试：在Schar里实现的定时器功能

Schar利用一个内核定时器周期性地把数据添加到自己的数据池里。其时间间隔既可以通过一个ioctl调用来完成，也可以通过修改schar.h文件里的SCHAR_TIMER_DELAY来实现。

定时器是在有读数据进程打开了这个设备时投入使用的，并且会在读数据过程中在定时器处理器里被反复添加，直到读数据的操作结束为止。

```
$ cp /dev/schar out_file
schar: opened for reading
schar: major: 42 minor: 0
schar: putting process with pid 889 to sleep

schar: setting timer up again, 1024 data now
schar: putting process with pid 889 to sleep

schar: setting timer up again, 2048 data now
schar: putting process with pid 889 to sleep

schar: setting timer up again, 3072 data now
schar: putting process with pid 889 to sleep

schar: setting timer up again, 4096 data now
schar: want to read 4096 bytes, 0 bytes in queue
schar: putting process with pid 889 to sleep
```

定时器处理器的每次运行都会给数据池增加SCHAR_INC个字节的数据——我们的例子里是1024个字节。定时器处理器在每次运行时都会唤醒那个读数据进程（如果你运行不止一个进程，就会把它们都唤醒），告诉它们可供读取的数据已经增加了。当数据池里的数据量达到4096个字节时，cp命令就会接收它的第一段数据，然后再重新开始这一过程。

21.4.3 让出处理器

与其循环等待一段时间的流逝，不如采用“把当前进程的执行挂起来，等这段时间流逝过以后再恢复执行”这种更好的办法。改变进程的状态并调用进程调度器（scheduler）就可以实现这一目的。我们在前面的内容里解释过Linux的任务结构，这个结构里的state项给出的就是任务的状态，它可以取表21-10中任何一个值：

表 21-10

TASK_RUNNING	进程调度器上次运行时选择了这个任务，它目前正处于运行状态。当前进程进入驱动程序时就是这个状态
TASK_INTERRUPTIBLE	任务处于一种休眠状态，但可以被信号唤醒
TASK_UNINTERRUPTIBLE	任务处于一种休眠状态，并且不能被信号唤醒。这种进程可能正在等待某个事件的发生，而在事件发生之前唤醒它是没有意义的

还有一些其他的状态，但与我们这里的情况没有什么关系。在linux/sched.h文件里可以查到全部的任务状态。

在调用进程调度器之前，必须把current->state的值设置为TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE。具体使用哪个状态要视进程等待的条件而定。对有的进程来说，在它等待的事件发生之前唤醒它是没有意义的，并且把它放到一种不可中断的休眠状态是最符合逻辑的选择——只有对这样的情况才应该使用后一种设置。大多数问题都不需要这样来解决——如果你用grep命令检索Linux的源代码，就会发现只有非常少的几个地方使用了TASK_UNINTERRUPTIBLE。其中的一个例子是：当需要把属于某个进程的内存页面从硬盘上的交换分区加载到内存里来的时候，如果这项工作还没有完成，我们就不能让那个进程被唤醒。

下面是设置任务状态的原理性代码：

```
current->state = TASK_INTERRUPTIBLE ;
schedule() ;
```

进程被挂了起来，调度器开始运行。可我们怎样才能知道进程何时重返回来呢？如果接下来没有合适的进程来运行，就会再次选中并执行同一任务，而它几乎是立刻又停止在刚才的位置上了。这就是schedule_timeout出马的时候了——我们不再调用schedule而是调用schedule_timeout，它返回的是进程还将休眠的时基数字。调度器将保证在设定的时间流逝过去之前不选择该进程去执行。如下所示：

```
/* put the process away for a second */
current->state = TASK_INTERRUPTIBLE;
schedule_timeout(HZ);
```

参数HZ是进程再次执行之前将要休眠的时基数。schedule_timeout的内部工作原理是很有意思的——它使用了一个内核定时器，定时器的data参数就是当前进程的地址。在安排好定时器之后，schedule_timeout再调用schedule来调度自己；它的定时器处理函数的工作很简单——根据保存在data参数里的地址把与之对应的进程唤醒就完事了。如果读者有兴趣，可以自行研究kernel/sched.c文件。

21.4.4 任务队列

我们已经知道如何利用定时器在未来的某个时刻去唤醒进程了。定时器确实很方便，而且我们还可以利用它把工作推迟到未来某个时刻去执行。等到我们和中断处理例程及中断的后处理（它们与任务队列密切相关）打交道的时候，这就更有意思了。我们现在先来看看任务队列的情况，中断处理方面的内容留到下一小节再讨论。

```
struct tq_struct {
   struct tq_struct *next;          /* linked list of active bh's */
   unsigned long sync;              /* must be initialized to zero*/
   void (*routine)(void *);         /* function to call */
   void *data;                      /* argument to function */
};
```

这个结构定义出现在linux/tqueue.h头文件里。和往常一样，大家最好不要自己去改动这个结构里的数据域。内部使用的*next指针把队列中的任务串在一个链接列表里，参数sync将保证同一任务不会在队列里出现两次。这会造成链接列表的数据崩溃，有了sync之后，如果你还想让一个任务排两次队，你的第二次排队请求就会被屏蔽。

内核里有一些预先定义好的任务队列。我们先介绍如何定义自己的队列：

```
DECLARE_TASK_QUEUE(q_task);
struct tq_struct q_run = { NULL, 0, (void *)q_task_handler,
            &q_task };
```

这就是队列中结构的定义。这基本上就是从头文件里直接抄下来内容，惟一不同的是我们定义了自己的任务队列处理函数，并且在定义之后立刻把它投入了使用。在上面的代码里，DECLARE_TASK_QUEUE是把队列初始化为NULL的宏定义；q_task就是我们的任务队列；任务队列q_run调用q_task_handler函数对队列里的任务进行处理。给这个任务队列添加一个新进程要使用：

```
queue_task(&q_run, &q_task );
```

queue_task可以被调用任意次数。准备好执行队列里的任务时，调用run_task_queue，排在q_task队列里的任务会依次得到处理。如下所示：

```
run_task_queue(&q_task );
```

21.4.5 预定义任务队列

我们不准备对用户定义的任务队列做过多的讨论，因为内核已经预备下几个任务队列供我们使用，而大部分情况下有它们就足够了。你几乎不需要创建自己的任务队列见表21-11。

表 21-11

tq_immediate	这个队列里的任务会尽可能迅速地得到运行。我们将在中断处理部分对它做详细讨论
tq_timer	定时器每跳动一个数字，tq_timer就得到一次运行机会。这个队列是内核提供给内核定时器内部使用的
tq_scheduler	调度器每调用一次，这个队列就得到一次运行机会——如果这个队列里有任务在排队的话
tq_disk	VFS和数据块处理函数大多使用这个队列来处理操作请求

很明显，前三个队列是最有意思的；第四个tq_disk队列有非常特殊的用途，所以你最好离它远点。排除tq_disk队列不算，前三个队列是按照任务进入其中后得到运行机会的快慢顺序排列的（第一个最快）。

在使用内建的任务队列的时候，根本用不着你自己去发出run_task_queue调用。根据你使用的是哪一个队列，你只要在任务添加到其中之后给那个队列加上一个标记，它就会自动运行了。在介绍如何给队列加上运行标记之前，我们先来快速了解一下如何在linux里实现中断的后处理。

在讲述中断处理内容的小节里我们将对中断的后处理做更深入的讨论，但它们与任务队列有密切的关系，所以我们先在这里用它们做个例子。简单地说，中断处理可以被分为两个部分——前处理和后处理。前处理部分负责接收中断并完成紧急、基本的处理，比如把它通知给设备等；而后处理部分会稍后再运行，它负责对前处理部分接收到的数据进行具体的处理。这就使Linux里的中断处理器能够处理的中断个数要多于它能管理的中断种类数。

事实上，任务队列在Linux里被实现为后处理。内核从调度器里通过do_bottom_half来执行被标记为活跃的后处理部分。内核在linux/interrupt.h文件里维护着一个各种可能的后处理名单。下面是其中的一段：

```
enum {
        TIMER_BH = 0,
        CONSOLE_BH,
        TQUEUE_BH,
        ...
        IMMEDIATE_BH,
        ...
};
```

这是一些老式的嵌入式后处理，它们都有特殊的用途，模块不能直接访问它们。在这个头文件里你找不到与tq_scheduler对应的项，这是因为它不需要加上执行标记。调度器做的第一件事就是查看tq_scheduler里有没有排队的任务，如果有就执行它们。这也就意味着每次调用调度器的时候，这个队列都会被运行；这取决于许多因素。

如果tq_timer队列里有排着队的任务，就会在定时器每次跳动的时候被运行，因此它也不需要加上执行标记。这也意味着等候在这个队列里的任务是在中断时间里运行的，这就给它们能够完成的工作加上了一定的限制，比如说，在中断时间里运行的函数不允许阻塞或休眠。这听起来好象并不难做到，但你不要忘记这需要把这样一个函数的整个调用路径都考虑进来——也就是说，你不能调用任何可能会阻塞的函数。

最后一个可以考虑用做一般用途的队列是tq_immediate。正如这个名字表示的那样（单词immediate的意思是“立刻”），这个队列得到运行的机会是相当快的。它通常被用来推迟中断处理器里的部分工作，我们会在中断处理部分用到它见表21-12。

表 21-12

void init_bh(int nr,void(*routine)(void))	把函数routine设置为nr的后处理部分
void mark_bh(int nr)	把nr的对应位标记为bh_active；这样，当调度器被调用时就会运行这个任务队列

tq_immediate是惟一需要明确标记的队列，这是因为它的预定用途是做为后处理部分。我们

将在中断处理部分给出几个任务队列和后处理的例子。

21.4.6 小结

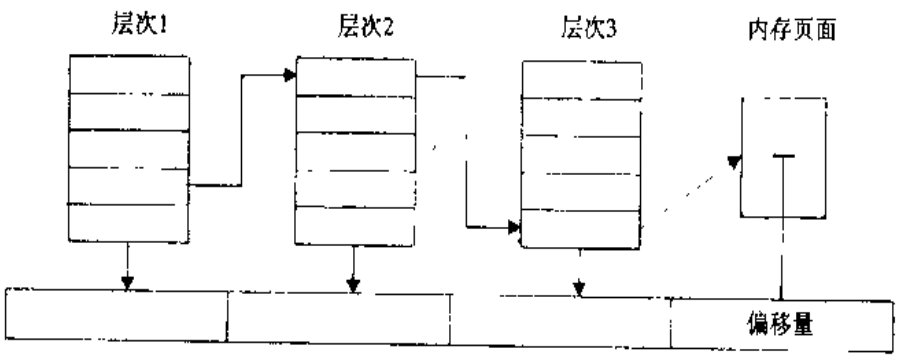
我们从时基开始了这一小节的学习，它可以被看做是内核的心跳。标准内核的时基频率是每秒100，每经过一个时基就给jiffies变量增加一个值。jiffies迟早会溢出归零，然后再从零重新开始计时。因此，我们向驱动程序的编写人员介绍了四个宏命令来帮助他们判断一段预定的时间是否已经流逝过去了。我们介绍了进程的再调度，它既可以通过一个倒计时数值来实现，也可以通过强制调度器运行来选择一个新的进程去执行（此时它可能会再次选中同一个进程）。

我们还向大家介绍了内核定时器，并且演示了如何在Schar里通过它在今后某个给定的时刻来调用一个特定的定时器处理函数。任务队列是推迟工作稍后再执行的另一种方法，但它们不能准确地在未来某个给定的时刻被调用；并且，从整体上看，任务队列更适合需要快速响应的处理工作或者运行成批的计算机作业。

21.5 内存管理

在开始学习如何在驱动程序里正确地分配内存之前，我们先来看看内存管理方面的基本知识以及Linux在这方面的处理办法。大家可能已经知道有“虚拟内存”这么一个术语，但对它说的到底是什么可能还不太清楚。当你在用户空间应用程序里分配内存的时候，得到的并不是计算机RAM里的一个完全归你个人使用的区段。当然，也许有一种操作系统在内存分配方面使用的就是这种办法。如果真是这样，那么，它允许你运行的应用程序也就只有其物理内存能够容纳的那么多。Linux可不是这样——它假定还有更多的内存可供使用（这就是“虚拟内存”这个术语的出处，意思是“不是真正存在的内存”），面分配给应用程序的都是些虚拟地址。

虚拟地址和物理地址之间的映射保存在好几个结构里，它们构成了内存页面表。内存页面表由三个层次组成——内存页面目录、内存页面中间目录和内存页面表项目，页面表项目指向的内存页提供了一个虚拟地址的偏移量。这是内存管理的最底层，驱动程序的设计人员通常并不需要直接与内存页面表打交道见图21-3。



虚拟地址到物理地址的转换

图 21-3

21.5.1 虚拟内存区

上面说的内存页面表驻留在虚拟内存区里。它们是一大片连续的虚拟内存地址，就向提供给应用程序时那样。

```
struct vm_area_struct {
   …
   unsigned long vm_start;
   unsigned long vm_end;
   …
   pgprot_t vm_page_prot;
   …
   struct vm_operations_struct *vm_ops;
   unsigned long vm_offset;
   struct file *vm_file;
   …
};
```

这只是该结构一个高度浓缩了的版本，大家可以在linux/mm.h里查到它。我们对这个结构里我们一会儿将会用到的几个成员做个介绍：vm_start和vm_end分别代表虚拟内存区的开始和结尾；vm_page_prot是分配给这个虚拟内存区的保护属性——它可以是共享、私用、可执行，等等；vm_ops类似于字符设备和块设备使用的file_operations文件操作结构，它总结虚拟内存区上的操作并把它们抽象为一个结构；vm_offset是这个区域里的偏移量；而vm_file则用来把文件的内存映射再进一步映射到虚拟内存区里。我们将在分析Schar的mmap函数时再对这个结构做深入的介绍。

某个特定进程所做的映射可以在/proc/<pid>/maps处查到，它们每个都对应于一个独立的vm_area_struct结构。与某个映射关联的虚拟内存区长度、总长度、保护属性等许多信息都可以从proc注册项里读出来。

21.5.2 地址空间

整个可编址内存区（在32位平台上是4GB）被分为两个主要部分——内核空间和用户空间

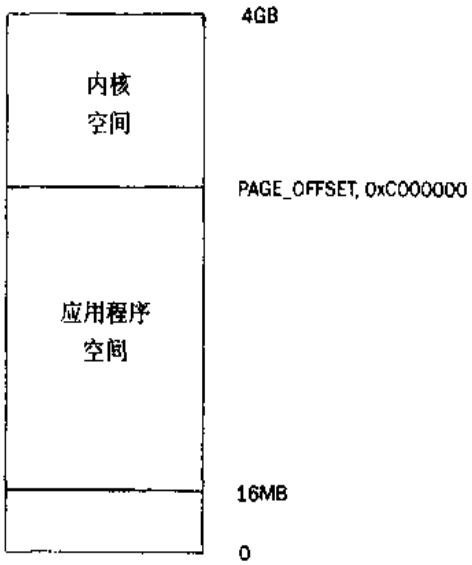


图 21-4

（或者叫应用程序空间）。PAGE_OFFSET定义了这两个部分的分界线，这个值是可以修改的，它的定义在asm/page.h文件里。内核空间被安排在分界线的上面，用户空间则位于分界线的下面。PAGE_OFFSET在Intel平台上的缺省值是0xc0000000，也就是说，它给内核提供了大约1GB的内存空间，剩下3GB给用户空间使用。因此，在Intel平台上，从内核方面看到的虚拟地址直接就是物理地址的一个偏移量。在其他平台上就未必如此了，那就必须对这两种地址进行转换（图21-4）。

21.5.3 内存地址的类型

做为一名设备驱动程序的编写者，你必须弄清楚有三个类型的地址：

- **物理地址**：这是“真正的”地址，它被用来描述计算机主板上的内存总线。
- **虚拟地址**：只有CPU和内核（通过它的内存页面表和TLB）知道虚拟地址。
- **总线地址**：CPU以外的所有设备。在某些平台上它与物理地址相同。

可见，如果你打算和一个外设卡进行交谈，你就不能给它一个虚拟地址并告诉它给你传递多少多少字节来。因为外设卡访问不到内存页面表，所以它完全不知道内核和CPU采用的是什么样的编址方案，也就不会懂得内存地址代表的是什么地方。类似地，内核对一切事物都使用虚拟地址，而总线内存的访问则会随平台的不同而变化。Linux为这三种地址之间的转换准备了一些方便的宏定义和函数。

```
unsigned long virt_to_phys(void *address)
void *phys_to_virt(unsigned long address)

unsigned long virt_to_bus(void *address)
void *bus_to_virt(unsigned long address)
```

与外部设备交谈需要在虚拟地址（内核知道哪对哪）和总线地址（设备知道哪对哪）来回进行转换。这与外设安装在哪种类型的总线上没有关系，它可以是PCI、ISA或者其他任何类型。注意：只有在确实需要把一个指向内存区的指针明确地直接传递给设备的时候，才有必要跳过不使用那些地址转换操作；DMA传输就是一个这样的例子。在其他情况里，我们一般都是从设备的I/O内存或I/O端口那里来读取数据的。

21.5.4 在设备驱动程序里申请内存

在目标机器上，内存都是以PAGE_SIZE长的内存块为单位而分配的。Intel平台上的页面长度是4KB，而Alpha平台使用的是8KB的页面长度，而且这不是一个允许用户配置的选项。千万记住页面长度在不同的平台上是不一样的。为驱动程序分配内存的办法有很多，最底层的办法是使用一个如下所示的调用：

```
unsigned long __get_free_page(int gfp_mask)  精确地分配一个页面的内存。
```

gtp_mask给出的是我们打算获得的内存页面的优先级和各种属性。在驱动程序里经常会用到的如表21-13所示：

表 21-13

GFP_ATOMIC	只要还有可用内存，就必须返回调用申请的内存；调用不允许阻塞，也不允许让硬盘交换分区中的页面进入内存
GFP_KERNEL	只要还有可用内存，就必须返回调用申请的内存；但如果需要把页面交换出去才能有可用内存，这个调用可能会被阻塞
GFP_DMA	为了适合被用做DMA缓冲区，返回的必须是前16MB里的内存。这个标志只适用于ISA外设，因为它们不能对16MB以上的地址进行寻址

如果你想在中断时间里分配内存，就必须指定GFP_ATOMIC——如果没有合适的页面，这个标志将确保当前进程不会被调度出去。ISA板卡最多只能看到16MB的内存，如果你是在为某个ISA外设分配一个DMA传输缓冲区的话，就必须指定GFP_DMA。根据计算机里插的内存容量和内存段的划分策略，带GFP_DMA标志的内存分配不见得会成功。PCI设备没有这个限制，可以使用__get_free_page返回的任何内存进行DMA传输。

```
__get_free_page is actually just a special case of __get_free_pages.

unsigned long __get_free_pages(int gfp_mask, unsigned long order)
```

gtp_mask的作用和含义与刚才说的相同，order（幂序）却是个新概念。页面只能以2的幂为单位进行分配，也就是说，返回页面的个数将是2^{order} PAGE_SIZE常数定义了软件内存页面的长度，在Intel平台上这个值是12（2^{12}个字节就是4KB） 如果order等于0，就将返回一个长度为PAGE_SIZE字节的内存页面，依此类推。内核在内部为不同的幂序建立五个换算表，这就把order的最大值限制为5。因此，在Intel平台上，一次内存分配最多可以让你得到128KB（$2^5 \times 4 = 128$）。

读者可能奇怪为什么这个函数的名字是以双下划线“__”打头的，这里有一个非常好的解释：它们实际上是get_free_page和get_free_pages这两个函数的变体，但两个变体的执行速度更快。这里的奥妙在于双下划线“__”版本在返回内存页面之前不对它们进行清理。如果你还需要把内存拷回到用户空间应用程序去，最好对页面以前的内容进行清理——因为它们可能会在不经意间存留着一些不应该传递到其他进程去的敏感信息。__get_free_page和__get_free_pages执行起来比较快，如果分配到的内存只用于内部，清理页面的工作一般是能省就省。

在用完内存后一定要记得再释放它们，这是非常非常重要的。内核不会在模块被卸载时替模块回收由它们分配的内存页面，所以模块必须完全承担起自己的内存管理责任如表21-14所示。

表 21-14

void free_page(unsigned long addr)	释放内存地址addr处的一个或多个内存页面
void free_pages(unsigned long addr, unsigned long order)	你必须记住自己分配的内存页面们的长度，这是因为free_pages需要你按分配内存时的顺序把它们提供出来

1. kmalloc

对设备驱动程序来说，通过get_free_page系列函数来分配内存多少有些麻烦，需要在内存管理方面做大量的工作。根据你想通过使用内存而达到的目的，单页面内存分配机制可能不是最合适的方式。此外，单页面机制分配的内存其总长度以2的幂次递增，而你需要的内存长度一般

不会那么正好。这就可能导致出现大量不能被利用的内存。Linux为此准备了一个kmalloc函数，它使你能够按自己的意愿分配任意长度的内存

```
void *kmalloc(size_t size, int flags)
```

size是申请分配的内存总量，它会自动向上取整为最接近的页面长度。flags是一个优先级掩码，作用和含义与get_free_page系列的相同。长度限制依然存在——你还不能用它分配超过128KB的内存。尝试分配更多的内存会在日志记录里留下一个错误，例如告诉你“kmalloc: Size (135168) too large”。

```
void kfree (const void *addr)
```

kfree将释放以前用kmalloc分配的内存。如果你习惯于在应用程序里使用malloc动态地分配内存，也就会习惯使用kmalloc。

2. vmalloc

获取内存的第三种也是最后一种办法是使用vmalloc。get_free_page和kmalloc返回的内存都是物理连续的，但vmalloc提供的是虚拟地址空间里的连续内存，所以可以满足一些特殊的要求。它通过分配独立页面和处理内存页面表来实现其功能

```
void *vmalloc(unsigned long size)
void vfree(void *addr)
```

vmalloc允许分配的页面长度远大于kmalloc，但分配到的内存只能用在内核里面，传递给外部设备的区域不能用vmalloc来分配，就是因为它们在物理地址空间是不连续的。虚拟内存只能用在内核/CPU上下文环境里，因为这里可以查看内存页面表。

vmalloc不能用在中断时间，这是因为它可能会被挂起。原因很简单，vmalloc在调用时没有设置GFP_ATOMIC标志。这一般不会成为一个严重的问题，因为一个中断处理器需要的内存比__get_free_pages能够分配的还要多的情况是极其少见的，甚至可以说是不正常的。

总而言之，vmalloc最适合内部存储使用。在块设备部分实现的RAM盘模块Radimo里就使用了vmalloc来分配内存。

21.5.5 在用户空间和内核空间之间传递数据

运行在系统上的应用程序只能访问PAGE_OFFSET分界线以下的内存。这就保证了任何进程都不能覆盖由内核管理着的内存，不会影响到系统的稳定性和功能性。但是这也带来了一个问题，那就是如何才能把数据回传到用户空间去。运行在内核环境里的进程允许访问两个内存空间，但同时也必须验证进程给出的内存位置确实在它的虚拟内存区以内。

```
int access_ok(int type, const void *addr,
              unsigned long size)
```

如果能够以type方式对从地址addr开始字节长度为size的内存位置进行访问，上面这个宏命令将返回“1”，否则返回“0”。type的取值是VERIFY_READ或VERIFY_WRITE之一，这要根据数据传输的方向而定。用户空间来、回方向上的每一次数据传输都必须先确认给定的地址是合法的。完成这一确认工作的代码与机器的体系结构有关，大家可以在头文件asm/uaccess.h里查到。

数据传输实际上是由各种各样的函数来具体完成的，这要取决于被传输数据的长度（见表21-15）：

表 21-15

get_user(void*x,const void *addr)	从用户空间地址addr开始把size(addr)个字节拷贝到变量x里来
put_user(void*x,const void *addr)	从地址addr开始把size(addr)个字节拷贝到用户空间的变量x里去

addr指针的类型必须是已知的，必要时要进行指针投射转换——正因为如此，这两个函数才不需要有数据长度方面的参数。它们的实现相当难解，在刚才提到的头文件里可以查到它们的代码清单。它们经常被用来实现ioctl调用，因为那些调用经常需要来回传输一些单值变量。

你可能奇怪为什么在Schar的代码里没有看到access_ok调用的身影。是这样的，因为经常会错误地漏掉这个检查，所以人们把这个检查加到x_user系列函数里去了。如果数据拷贝成功，函数返回值将是“0”；如果访问非法，返回值就将是“-EFAULT”。

```
__get_user(x, addr)
```

加上了双下划线“__”前缀的版本不进行检查。它们通常用于连续多个单值数据的拷贝操作上，而此时重复进行权限检查是多余的。请看下面这段代码：

```
char foo[2];
...
if (access_ok(VERIFY_WRITE, arg, 2*sizeof(*arg)) {
   __put_user(foo[0], arg);
   __put_user(foo[1], arg+1);
} else {
   return -EFAULT;
}
```

这只是个很简单的例子，但在原理上是很清晰的。x_user系列函数还有第三个版本。因为总是要对返回值进行检查，访问非法时又总是返回“-EFAULT”，所以人们又编写了最后一种变体。如下所示：

```
void get_user_ret(x, addr, ret)
void put_user_ret(x, addr, ret)
```

_ret版本会在出现错误时返回ret里的值——它们不返回一个错误代码。这就简化了ioctl命令的程序设计，因而出现了下面这样简单的代码：

```
get_user_ret(tmp, (long *)arg, -EFAULT);
```

移动更多的数据

经常会出现需要拷贝的数据比几个单值变量要多的情况。如果还使用上面介绍的方法来编写程序不行了，效率又很低，代码也难看。Linux提供了一些能够一次传递较多数据的函数。我们在Schar的读、写函数里使用了其中的几个。

```
copy_to_user(void *to, void *from, unsigned long size)
copy_from_user(void *to, void *from, unsigned long size)
```

它们把总数为size个的字节拷贝到指针指定地点或从指定地点拷贝走。如果操作成功，函数的返回值就将是“0”；如果访问不被允许，就将返回一个非零值（实际就是没有传输的字

节数），这是因为copy_xx_user系列函数内部也调用了access_ok。在Schar里可以找到一个下面这样的例子：

```
if (copy_to_user(buf, schar_buffer, count))
   return -EFAULT;
```

类似于get_user的情况，它们也有不进行权限检查的版本，而且同样是加上一个双下划线“__”前缀。

```
__copy_to_user(void *to, void *from, unsigned long size)
__copy_from_user(void *to, void *from, unsigned long size)
```

最后，同样有在访问非法时返回ret的_ret变体，如下所示：

```
copy_to_user_ret(void *to, void *from, unsigned long size, int ret)
copy_from_user_ret(void *to, void *from, unsigned long size, int ret)
```

请大家注意，以上介绍的这些函数都必须运行在一个进程上下文环境里，这就意味着它们在中断处理器和定时器函数等情况里的使用是严格禁止的。在这些情况里，内核函数不属于某个特定的进程，因而无法知道current任务是否与你有任何关系。因此，在这些情况里，比较好的办法是先把数据拷贝到一个由驱动程序负责管理的缓冲区里，过后再把数据转移到用户空间去。我们马上就要开始学习的设备驱动程序缓冲区的内存映射技术可以解决这一问题，而且不需要额外的拷贝操作。

21.5.6 简单的内存映射

有时候，与其在内核空间和用户空间不停地来回拷贝数据，不如给应用程序提供一个办法，让它能够连续地查看到设备里面的内存。这个概念就叫做内存映射，大家在编写应用程序的时候可能用过这个技术来映射整个文件——即不采用而向文件的普通读写调用而是通过指针来对文件进行读写。如果不是这样，请读者再复习一下第3章对mmap的功能及其在用户空间使用情况的介绍。事实上，许多概念在那里都解释过了，它们同样适用于解释我们在这里的操作。

直接向用户空间拷贝数据并不总是安全的。正在拷贝数据的进程可能会被调度器调度出去而不能继续执行，而这对中断处理器等函数来说是致命的。解决方案之一是使用一个内部是缓冲区，让这类函数对缓冲区进行读写，过后再把数据拷贝到适当的地方去。可这样做就需要对同样的数据拷贝两次，一次拷贝到内部缓冲区，另一次拷贝到应用程序的内存区域，从而额外增加了系统的开销。如果驱动程序本身实现有mmap文件操作入口点，它就能够向应用程序提供一个直接查看自己驱动程序缓冲区的手段，也就不必读数据再做第二次拷贝操作了。

我们在Schar的file_operations结构里加上schar_mmap文件操作，以此来表示我们支持这一操作。下面就是Schar里的具体实现情况：

```
static int schar_mmap(struct file *file,
                      struct vm_area_struct *vma)
{
   unsigned long size;

   /* mmap flags - could be read and write, also */
   MSG("mmap: %s\n", vma->vm_flags & VM_WRITE ? "write" :
       "read");
```

```
    /* we will not accept an offset into the page */
    if(vma->vm_offset != 0) {
       MSG("mmap: offset must be 0\n");
       return -EINVAL;
    }

    /* schar_buffer is only one page */
    size = vma->vm_end - vma->vm_start;
    if (size != PAGE_SIZE) {
       MSG("mmap: wanted %lu, but PAGE_SIZE is %lu\n",
           size, PAGE_SIZE);
       return -EINVAL;
    }

    /* remap user buffer */
    if (remap_page_range(vma->vm_start,
        virt_to_phys(schar_buffer),
        size, vma->vm_page_prot))
       return -EAGAIN;

    return 0;
}
```

我们的schar_mmap函数要用到两个参数，一个file文件结构和一个与我们的内存映射相关联的虚拟内存区。我们已经在前面介绍过：vm_start和vm_end分别代表映射的开始和结束位置，它们的差就是要求的长度。Schar的缓冲区只有一个页面长，所以尺寸更大的映射要求将会被拒绝。vm_offset是缓冲区里面的偏移量。在我们的例子里，如果在区区一个页面里还使用偏移量未免有点小题大做，所以我们的schar_mmap会拒绝执行指定了偏移量的内存映射操作。

最后一步最为重要。remap_page_range把内存页面表里从vma->wvm_start开始的内存区域的总长度设置为size个字节。这样就把物理地址有效地映射到虚拟地址空间里来了。

```
remap_page_range(unsigned long from, unsigned long phys_addr,
                 unsigned long size, pgprot_t prot)
```

如果操作成功，这个函数将返回“0”；如果失败则返回“-ENOMEM”。prot参数设定了这个区域的保护属性：比如MAP_SHARED表示这是一个共享区域，而MAP_PRIVATE则表示这是一个私用区域，等等。Schar把它直接从应用程序的mmap调用里传递给我们的函数。

被映射了的页面必须加上适当的操作锁以免被内核考虑用做其他用途。系统里的每一个内存页面在mem_map[]数组里都有一个与之对应的项目，我们可以在那里查看到页面的分配情况并对其属性做必要的设置。

`unsigned long MAP_NR(unsigned long page_addr)`	返回页面在mem_map[]数组里的下标。

Schar分配了一个页面的内存，然后根据MAP_NR返回的下标调用mem_map_reserve，这是为了保证Schar能够独占性地拥有这个页面。当我们的驱动程序被卸载时，这个页面上的操作锁将被函数mem_map_unreserve解除，并在clean_module里得到释放。这一连串操作的次序是很重要的，因为free_page不能释放被设置为独占性拥有的内存页面。内存页面的结构及其各种标志和属性都可以在linux/mm.h头文件里查到。

我们给出这个例子的目的是为了向大家演示如何利用remap_page_range来实现从用户空间对内核中的虚拟内存的访问。但在许多实际情况里，人们更经常地利用驱动程序的内存映射功能

来访问外部设备本身上面的缓冲区。接下来的内容是I/O内存，在这部分学习过程中，我们会简单地介绍一下怎样才能实现对外部设备自己的缓冲区进行访问。

21.5.7 I/O内存

I/O内存是我们将要学习的最后一种地址空间。它包括1MB限制之下的ISA内存和高端的PCI内存，但这两种内存的访问方法在概念上都是相同的。I/O内存不是普通意义上的内存，它们是一些映射到相关内存区域里的端口或缓冲区。外部设备可能会有一个状态端口或板上缓冲区，我们就是要对它们进行访问。示例模块Iomap演示了有关操作的原理，可以用来实现对I/O内存的读、写或内存映射操作。

I/O内存将被映射到什么地方在很大程度上要取决于具体使用的计算机平台。在Intel平台上，只要对指针进行退化（即使指针不再具有索引功能）就可以用它来访问低端的内存。但退化指针并不见得总能够落在物理内存的范围里，所以我们必须使用二次映射（remap，一般称之为“二次映射”或“再映射”）来确保万无一失。

```
void *ioremap(unsigned long offset, unsigned long size)
```

ioremap的作用是把一个物理内存地点映射为一个内核指针，被映射数据的长度由size参数设定。Iomap模块的功能是把一块图形卡上的帧缓冲区（这类卡的主要用途就是提供帧缓冲区）二次映射到一个我们可以从驱动程序里访问的虚拟地址上去。Iomap设备被定义为下面这个结构：

```
struct Iomap {
   unsigned long base;
   unsigned long size;
   char *ptr;
}
```

结构定义里的base是帧缓冲区的起始地点（基地址），size是缓冲区的长度，ptr将用来保存ioremap函数的返回值。基地址可以从/proc/pci注册项中查到——当然你得先有一块PCI或AGP适配器才行。这个值就是下面列出来的prefetchable（预读）地址值：

```
$ cat /proc/pci
PCI devices found:
  Bus  1, device   0, function  0:
    VGA compatible controller: NVidia Unknown device (rev 17).
      Vendor id=10de. Device id=29.
      Medium devsel.  Fast back-to-back capable.  IRQ 16.  Master Capable.
Latency=64.  Min Gnt=5.Max Lat=1.
      Non-prefetchable 32 bit memory at 0xdf000000 [0xdf000000].
      Prefetchable 32 bit memory at 0xe2000000 [0xe2000008].
```

/proc/pci里面有你系统里所有PCI设备的资料，从中找出你的图形适配器，记下它的预读地址（即清单里前面加有“Prefetchable”字样标记的数字）——大家可以看到，它在我们这台系统上是“0xe2000000”。Iomap最多可以管理16个不同的映射，这些映射都是通过ioctl命令设置的。我们在稍后对Iomap进行测试时将用到这个地址值。

二次映射完成之后，就可以直接对其中的数据进行读写了。Iomap使用的是数据长度以字节为单位的函数。如下所示：

```
unsigned char *readb(void *addr)
unsigned char *writeb(unsigned char data, void *addr)
```

readb返回从地址addr处读到的字节，而writeb把数据写到指定地址去。writeb还可以返回它刚才写的数据——如果你需要这样做的话。还有以doubleword（双字）或long（长整数）为数据长度单位来进行读写的函数，它们是：

```
unsigned short *readw(void *addr)
unsigned short *writew(unsigned short data, void *addr)
unsigned long *readl(void *addr)
unsigned long *writel(unsigned long data, void *addr)
```

如果我们在Iomap模块里定义了IOMAP_BYTE_WISE标志，它就会使用数据长度以字节为单位的函数进行读写，它也正是这样做的。大家必须清楚，如果是拷贝兆字节长度的数据，这些函数的速度就不能令人满意了，而这也并不是使用它们的目的所在。如果没有定义IOMAP_BYTE_WISE标志，Iomap就会利用其他函数来拷贝数据，如下所示：

```
void *memcpy_fromio(void *to, const void *from, unsigned long size)
void *memcpy_toio(void *to, const void *from, unsigned long size)
```

这几个函数的工作情况类似于memcpy，区别就在于它们的操作对象是I/O内存。还有一个与memset函数相对应的版本，作用是把整个区域设置为一个指定的值。如下所示：

```
void *memset_io(void *addr, int value, unsigned long size)
```

Iomap模块里实现读功能和写功能的两个函数其工作原理与Schar模块里的差不多，我们这里就不再列出它们的代码清单了。数据将通过一个内核缓冲区在用户空间和经过二次映射的I/O内存之间移来移去，记得要对文件的读写位置做相应的修改。

二次映射的区域必须在卸载模块时解除映射关系和释放。把ioremap函数返回的那个指针传递到iounmap函数可以解除映射关系，如下所示：

```
void iounmap(void *addr)
```

可移植性

由读函数和写函数返回的数据以字节的降序存储格式（即低位字节保存在低地址里）排列，这与它是否是目标机器的本地字节顺序没有关系。这是PCI外设本身固定使用的字节存储顺序，上面介绍的这些函数都能在必要时交换数据的字节存储顺序。如果数据需要在两种数据类型字之间进行转换，Linux也备有完成相关转换的指令。这在专门讨论可移植性问题的附录里有进一步的说明。

21.5.8 IOmap里的设备分配

Iomap用一个全局性的数组来保存可能创建的设备，数组的下标就是设备的辅编号。这是需要管理多台设备时广泛使用的方法，它的具体应用并不复杂。全局数组iomap_dev里为每个可能会被访问的设备保存着一个指针。在设备每一个功能函数的入口点位置上，将要被操作的具体设备会从数组里提取出来，如下所示：

```
Iomap *idev = iomap_dev[MINOR(inode->i_rdev)];
```

如果函数不直接使用一个inode结点做为参数，它也会从file文件结构里被间接地提取出来。文件结构里有一个指向与文件关联着的dentry项目（“directoey entry”，目录项）的指针，而inode可以从这个结构里查到。如下所示：

```
Iomap *idev = iomap_dev[MINOR(file->f_dentry->d_inode->i_rdev)];
```

21.5.9 对I/O内存实现mmap文件操作

除实现有read和write两种文件操作以外，Iomap模块还实现了mmap文件操作，用户空间应用程序通过内存映射功能将能够直接访问二次映射下的I/O内存。对内存页面实际进行的二次映射与Schar里的做法很相似，但因为映射并不涉及真正的物理页面，所以后者不需要进行阻塞。I/O内存不是真正的RAM，因此mem_map里没有与它对应项目。

```
remap_page_range(vma->vm_start, idev->base, size,
                                vma->vm_page_prot)
```

类似于Schar中的情况，remap_page_range函数是iomap_mmap文件操作的核心，它为我们完成了内存页面表的设置工作。最终编写出来的函数并不需要太多的代码。

```
static int iomap_mmap(struct file *file, struct vm_area_struct *vma)
{
   Iomap *idev = iomap_dev[MINOR(file->f_dentry->d_inode->i_rdev)];
   unsigned long size;

   /* no such device */
   if (!idev->base)
      return -ENXIO;

   /* size must be a multiple of PAGE_SIZE */
   size = vma->vm_end - vma->vm_start;
   if (size % PAGE_SIZE)
      return -EINVAL;

   /* remap the range */
   if (remap_page_range(vma->vm_start, idev->base, size,
                vma->vm_page_prot))
      return -EAGAIN;

   MSG("region mmapped\n");
   return 0;
}
```

我们先从iomap_dev数组里查出具体是那一个设备，然后检查这个设备是否已经被初始化过了（即是否存在）。如果还没有，我们就相应地返回一条错误信息。我们还要求将要被二次映射的范围必须是页面长度的整数倍；如果不是，也相应地返回一条错误信息。如果这些要求都满足，我们就调用remap_page_range完成内存映射操作。

动手试试：Iomap模块

Iomap的源代码可以从Wrox出版社的Web站点下载到。进入modules/iomap子目录，你应该在那里看到下面这几个文件：

```
$ ls
iomap.c    iomap.h    iomap_setup.c    Makefile
```

1) 以根用户身份执行make命令建立iomap模块。然后创建两个特殊文件，它们的辅编号一个是0，另一个是1。再把模块插入到内核里去。

```
# make
# mknod /dev/iomap0 c 42 0
# mknod /dev/iomap1 c 42 1
# insmod iomap.o
iomap: module loaded
```

2) 现在一切都准备就绪了。单有Iomap是做不成什么事情的，所以我们先要建立两个设备才能开始测试。接下来你必须查出自己显示适配器帧缓冲区的基地址——请按照在这一小节开始部分介绍的方法从/proc/pci中把这个数字查出来。我们系统上的这个地址是0xe2000000。编写一个小程序，通过ioctl调用建立两个设备，这时要用到帧缓冲区的基地址。在存放着Iomap模块源代码的子目录里建立一个名为iomap_setup.c的文件，修改现有代码也行，它的内容如下所示：

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#include "iomap.h"

#define BASE    0xe2000000

int main(int argc, char *argv[])
{
int fd1 = open("/dev/iomap0", O_RDWR);
int fd2 = open("/dev/iomap1", O_RDWR);
Iomap dev1, dev2;

if (fd1 == -1 || fd2 == -1) {
   perror("open");
   return 1;
}

/* setup first device */
dev1.base = BASE;
dev1.size = 512 * 1024;
if (ioctl(fd1, IOMAP_SET, &dev1)) {
   perror("ioctl");
   return 2;
}

/* setup second device, offset the size of the first device */
dev2.base = BASE + dev1.size;
dev2.size = 512 * 1024;
if (ioctl(fd2, IOMAP_SET, &dev2)) {
   perror("ioctl");
   return 3;
}

return 0;
}
```

别忘了把BASE修改为你自己帧缓冲区的基地址！要不然结局可能就会是对你系统里的另外一个设备进行写操作了，而这可能是致命的。编译并运行iomap_setup程序，我们的测试设备就将建立出来。操作流程如下所示：

```
$ cc -Wall -o iomap_setup iomap_setup.c
$ ./iomap_setup
iomap: setting up minor 0
iomap: setup: 0xe2000000 extending 0x80000 bytes
```

```
iomap: setting up minor 1
iomap: setup: 0xe2080000 extending 0x80000 bytes
```

3) 现在我们有了两个测试用的设备，第一个从帧缓冲区的开始映射了0.5MB字节，第二个紧接着第一个又映射了0.5MB字节。它们直接映射到显示适配器的显存里去，对它们的写操作将引起屏幕图像的抖动。在继续尝试之前，先要加载上X窗口系统，然后从一个终端窗口里执行下面的命令：

```
$ cp /dev/iomap1 /dev/iomap0
```

4) 现在就可以看到上面这条命令的效果了！现在你显示器上与第二个设备的映射区对应的显示区也将出现在屏幕的顶部——这次的X任务看着可够怪的。继续进行，给屏幕画面顶部拷贝一些随机数据过去：

```
$ dd if=/dev/random of=/dev/iomap0 bs=512 count=1024
```

21.6 I/O端口

I/O端口是采用x86体系结构的某些计算机平台独有的事物。它们可以是某个外部设备上的状态端口，也可以是连接着你的鼠标的串行端口。对端口进行的读写操作必须按它规定的数据宽度来进行。其他平台（比如采用Alpha体系结构的计算机）没有真正的端口，只有I/O内存，它们通过对I/O地址的读写来实现对I/O数据的访问。

Linux里有许多能够对I/O地址进行读写的函数。它们在原理上大同小异，区别主要在端口的数据宽度方面。注意这一小节讨论的是I/O端口而不是I/O内存，I/O内存的内容已经在上一小节介绍过了。与这一小节内容有关的头文件主要是asm/io.h——这可是个乱糟糟的大文件，自己多注意吧。

一个给定的端口是否可用需要由驱动程序自己来检测。其他驱动程序可能已经占用了我们正在检测的那个端口，我们可不想输出一些会引起设备误操作的数据而造成灾难性的后果。

```
int check_region(unsigned int from, unsigned long extent)
```

from是我们准备测试的端口，extent是它的字节宽度。成功时函数返回“0”；如果端口已经被其他设备占用，就会返回一些其他的值。在找到适当的端口后，我们用下面的函数申请占用它：

```
void request_region(unsigned int from, unsigned long extent, const char *name)
void release_region(unsigned int from, unsigned long extent)
```

这几个函数的参数基本上都是一样的——name是出现在/proc/ioports里的名字，它们可以被看做是/proc/devices注册链上的设备标签。端口的数据宽度可以是8字节、16字节或32字节。

```
__u8  inb(unsigned int port)
__u16 inw(unsigned int port)
__u32 inl(unsigned int port)
```

这几个调用的含义很明显——从端口按指定宽度读入数据。返回值是读到的数据，这些数据会根据具体使用的计算机平台自动取为对应数据宽度的数据类型。对端口进行写操作的函数也都差不多：

```
void outb(__u8 data, unsigned int port)
void outw(__u16 data, unsigned int port)
void outl(__u32 data, unsigned int port)
```

对数据类型的要求是比较松散的，在不同的平台上会有一定的变化。其实确定数据类型并不是I/O端口仅有的一个问题，有的平台没有固定的端口，需要通过读写某些个内存地址来模仿它们。我们不准备对此做深入的探讨——大家可以去研究研究内核里的某些驱动程序。

Linux还提供了一些字符串版本的函数，使我们一次能够传输更多的数据，提高效率。

```
void insb(unsigned int port, void *addr, unsigned long count)
void outsb(unsigned int port, void *addr, unsigned long count)
```

addr是内存里的一个地址，数据将传输到这里或从这里传输出去；count是以端口数据宽度为单位的数据项个数。以字或双字为长度单位的函数版本语法定义与此类似，只是函数名字里的字母“b”要相应地改变为“w”或“l”——这些函数执行得很快，比使用inb加循环的办法效率要高很多。

21.6.1 可移植性

并非所有的平台都象Intel体系结构那样有固定的I/O端口，很多计算机采用映射到内存固定地址的办法来实现它们。不同的平台有不同的要求，所以上面介绍的这些I/O端口读写函数也不是惟一的手段。它们返回的数据都排列为字节的降序存储格式（即低位字节保存在低地址里），这可能不适用于某些平台，而某些采用字节升序存储格式的平台所提供的I/O读写函数没有对字节的存储做升降序转换处理。与各种平台有关的具体规定请参考asm/io.h文件。

21.6.2 中断处理

大多数真的硬件并不依赖于poll调度功能来控制数据流，它们通过中断向驱动程序给出表示数据可用的信号或者其他的硬件状态，驱动程序再根据这些信号采取相应的动作。编写ISR（Interrupt Service Routine，中断服务例程）一直笼罩着神秘的色彩，但这只是因为人们还不了解在Linux里编写这样的例程其实有多么的简单。它并不需要什么特殊的操作，这是因为Linux的中断处理器注册接口和（中断到来时最终的）中断处理接口都非常精致，而且并不复杂。

那么，中断到底是个什么东西呢？这是设备唤起设备驱动程序注意的一个办法，设备通过中断告诉驱动程序自己需要某种方式的服务了。中断实际上就是一些信号，信号可以表示“数据已经准备好，可以开始传输”，也可以表示“以前排在队中的命令已经执行完毕，设备可以接受新的命令”。

中断在Linux里的内部处理与计算机的体系结构密切相关，完全依赖于平台硬件安装的中断控制器的具体型号。如果读者对这一问题有兴趣，可以在arch/<your arch>/kernel/irq.c文件里查到有关的资料——“<your arch>”代表你计算机的体系结构，例如对采用Intel处理器芯片的计算机来说，这个文件可以是arch/i386/kernel/irq.c。

没有分配到专用处理器的中断将由Linux负责处理，或者采取缺省的响应动作，或者简单地屏蔽掉。通过列出/proc/interrupts子目录文件清单的办法可以查出你的系统上安装了哪些中断处

理器，请看下面的片段：

```
           CPU0       CPU1
0:     1368447    1341817    IO-APIC-edge  timer
1:       47684      47510    IO-APIC-edge  keyboard
2:           0          0          XT-PIC  cascade
4:      181793     182240    IO-APIC-edge  serial
5:      130943     130053    IO-APIC-edge  soundblaster
...
```

这是我们系统上中断处理器清单的一个片段。最左边的那一列是中断的编号，然后是每个CPU处理过的这种中断的次数。最后两列分别是中断的类型和注册这个中断的设备。我们看到，CPU0已经处理了130943次来自soundblaster设备的中断，而CPU1处理了130053次。0是一个特例——定时器中断（对x86而言，其他平台可能会与此不同）——它表示系统开机引导以来定时器的跳动次数；它的第四列表明该中断是如何被处理的。中断的处理方式与我们将要学习的内容没有太大的关系——你只要知道在SMP环境里IO-APIC表示中断是在CPU之间分布处理的，而XT-PIC是标准的中断处理器就足够了。

另外一个值得看看的文件是/proc/stat。这个文件的内容有一部分是已经发生过的中断的总次数。我们要查的那一行是intr，它的格式是“intr total irq0 irq1 irq2 ...”，其中的total是所有中断的总次数，irq0是0号中断的总次数，以此类推。这个文件还会列出没有注册处理器的中断的触发次数——而/proc/interrupts里就没有这个数字，当你打算测试自己的第一个中断驱动的设备驱动程序时，这个文件就很有方便了。

1. 分配一个中断

知道了怎样查看/proc里收集的中断统计情况之后，我们再来看看如何申请注册你自己的IRQ。下面是IRQ申请函数的语法定义，参数的解释在下面的表格里：

```
int request_irq(unsigned int irq,
        void (*handler)(int, void *, struct pt_regs *),
        unsigned long irqflags,
        const char *devname,
        void *dev_id)
```

request_irq成功时返回“0”；失败时返回一个相应的负错误值——其中最值得注意的是：如果申请的IRQ超出允许范围，返回“-EINVAL”；如果申请的是一个共享处理器而irqflags标志又不能匹配上某个已安装处理器，返回“-EBUSY”。表21-16是对参数的解释。

表 21-16

irq	你希望实际处理的IRQ编号
handler(int irq,void *dev_id, struct pt_regs *regs)	中断发生时将被调用的函数。这就是我们所说的IRQ中断处理器
irqflags	它控制着中断的行为。我们稍后会做深入的讲解
devname	将出现在/proc/interrupts文件清单里的名字
dev_id	用于支持中断的共享。它将被传递到我们自己编写的中断处理器里去，所以你可以利用它来传递一些必要的信息。比如说，IDE子系统在中断发生时利用它来区分自己控制的主设备和从设备（通常是第一块硬盘和第二块硬盘）

irqflags参数是以下几个标志的组合见表21-17：

表 21-17

SA_INTERRUPT	如果中断处理器在注册时带有这个标志，它在运行时就将禁止所有的IRQ。如果没有设置这个标志，就只禁止由它自己提供服务的IRQ
SA_SHIRQ	允许这个IRQ链被一个以上的设备共享。驱动程序必须符合irqflags标志里其他掩码设置的要求并提供正确的dev_id，否则共享将不能成功
SA_SAMPLE_RANDOM	Linux内核里有一个由random设备管理的随机数发生池。如果某个中断处理器负责管理的设备不按固定频率产生中断，它就能对这个池的"随机"概念做点贡献，也就应该设置上这个标志。很明显，这对被驱动的实际硬件有很强的依赖性

注册之后的中断处理器在被调用时需要有三个参数。它们是：中断号irq——它只有在中断处理器负责管理一个以上的IRQ时才有点用处，否则你就已经知道发生的是哪个中断了；中断发生时各个CPU寄存器的映象regs——它的用处很少，定义见asm/ptrace.h文件；设备号dev_id——这是中断处理器的第二个参数，我们刚才介绍过了。

取消IRQ处理器的注册要使用free_irq来完成。它的参数类似于request_irq，我们就不再解释了。

```
void free_irq(unsigned int irq, void *dev_id)
```

2. 获得一个适当的IRQ

在开始为驱动程序注册一个中断处理器之前，需要先查清楚哪些个IRQ是可以让我们来使用的。这对硬件的依赖程度很高，主要涉及外部设备和设备总线两方面问题。设备总线分为ISA、PCI或SBUS（SPARC机器上使用的一种设备总线）等。在外部设备方面，有些设备允许你从一个状态端口读取它的配置信息；其他的则需要进行试测。如果你真的需要为实际硬件编写一个驱动程序，就需要从它的供应商那里了解它的程序设计指标，他们会告诉你怎样才能正确地查出硬件本身必要的信息资料。

ISA和PCI是最主流的总线类型（至少在Intel平台上是如此）。人们曾经努力尝试过ISA部分地加上即插即用的功能，可它毕竟是在即插即用概念出现以前发明的，它既没有真正的标准，速度也让人无法恭维，所以如今新出的设备已经很少有使用ISA总线的了。PCI设备在其配置信息的检索方面准备了一个简洁而又标准化的手段，我们不再需要对它进行繁杂的试测和推断。如何管理控制PCI设备超出了我们这本书的讨论范围，如果你想对PCI做进一步研究，linux/pci.h是一个很不错的出发点，在Linux源代码里也有大量的实际例子。这一小节的其他内容将主要讨论传统意义上的设备。

如果硬件允许直接检索其配置信息，我们就不需要自己去做任何试测工作。我们前面也提到过，这个信息一般都列在硬件设备的使用手册里，对它们我们没有什么可讲的。如果没有对设备更有效的中断检测办法，我们就只能使用Linux提供的手段了：

```
unsigned long probe_irq_on(void)
int probe_irq_off(unsigned long unused)
```

probe_irq_on表示IRQ试测工作由此展开，而probe_irq_off表示试测结束。你要把触发设备产生一个IRQ的代码放在它们中间，probe_irq_off的返回值就将是我们想要确定的IRQ编号。如

果设备产生了一个以上的IRQ，probe_irq_off会返回一个负数值（这个值与找到并被触发的第一个IRQ有关，我们可以从中找出点线索来）。试测代码的典型写法如下所示：

```
int irq;
unsigned long foo;

/* clear dangling interrupts */
probe_irq_off(probe_irq_on());

foo = probe_irq_on();

/* this should provoke an interrupt from the device */
outb(TRIGGER_IRQ, PORT);

irq = probe_irq_off(uu);

if (irq > 0)
   printk("irq %d detected\n", irq);
```

这是一个纯理论意义上的例子，具体的IRQ试测操作需要在它的基础上增加针对具体情况的代码。probe_irq_on并没有实际的用途，因此probe_irq_off把它标记为未使用。我们感兴趣的是试测工作结束后的返回值——它就是我们想查的中断编号。

21.6.3 IRQ处理器

查到可用的IRQ之后，就需要由你来编写一段代码来处理设备产生的中断，这段代码就是人们说的中断处理器。中断处理器的作用是响应中断并提供相应的服务。设备在产生中断时通常还会产生一些需要传递到处理器里去数据，硬件改变工作状态时也会把这个改变通知给我们。根据SA_INTERRUPT标志的设置情况，中断处理器在运行时或者只禁止它自己负责处理的中断（使能其他所有中断），或者禁止所有的中断；也就是说，在中断处理器结束运行前，来自同一个设备的后续中断都将丢失。我们后面会将到如何解决这一问题。

中断发生时正常的执行流将被停止——内核停下它正在做的工作去启动相应的已注册处理器。中断处理器与普通的驱动程序入口点是有区别的，它们运行在中断期间，不代表任何特定的进程。这就意味着current当前进程与驱动程序并没有任何联系，也不应该去管驱动程序的事情——这也包括对用户空间的任何访问，比如数据的来回拷贝等。

中断处理器应该尽快完成执行，要不然就有可能丢失来自设备的另一个中断。如果这个中断是与其他设备共享的，来自其他设备的中断也将无法得到服务。我们前面已经说过，但这里再强调一下：在中断期间你不应该阻塞。如果你坚持这样做，就可能会启动调度器，而这是决不允许的。你会在控制台上看到一条Oop信息，告诉你发生“Scheduling in interrupt”（中断期间调度）错误。在中断处理器里也不允许休眠。作为一个基本原则，你必须在中断期间的运行里充分考虑如何与系统的其他部分搞好协调。

除了上面提到的这些事项，中断处理器再也没有什么特殊之处了，所以我们也就不准备给出一个它详细的编写示例。下面是中断处理器方面类似于中断号试测情况的一个理论性示例：

```
void our_intr(int irq, void *dev_id, struct pt_regs *regs)
{
   int status;

   printk("received interrupt %d\n", irq);
```

```
    /* reading status from board */
    inb(STATUS_PORT, status);

    /* we are sharing irq, check if it was our board */
    if (status & MY_IRQ_STAT)
       return;

    /* acknowledge IRQ */
    outb(STATUS_PORT, MY_ACK_IRQ);

    >transfer data from the device, if needed<

    /* schedule bottom half for execution */
    our_taskqueue.routine = (void *)(void *)our_bh;
    our_taskqueue.data = (void *)dev_id;
    queue_task(&our_taskqueue, &tq_immediate);
    mark_bh(IMMEDIATE_BH);

    return;
}
```

我们做的第一件事情是检查IRQ是否产生自这个驱动程序负责管理的设备，具体做法是从硬件上的预定端口读取一个状态信息。这是处理中断共享的一个办法。如果这个处理器控制着同一硬件设备的多个实例，就需要使用dev_id来区分它们。

21.6.4 中断的后处理

上面的处理器引入了一个新概念：中断的后处理。我们在任务队列部分对这个概念做过简单的介绍，但它们值得我们多加关注。在程序设计实践中，我们很少把全部工作都放在中断处理器里完成。相反，我们会定义一个任务队列并把它添加到immediate（立刻执行）队列里去——这样可以保证我们的队列能够很快地得到执行。our_bh就是这个设备上中断的后处理部分，当我们从中断的前处理部分（即真正的中断处理器）返回时，排在immediate队列里的our_bh将尽快得到执行。前处理部分的工作基本上只是把数据从设备拷贝到一个内部的缓冲区里，而对这些数据的必要处理则交给后处理部分来完成。

是否值得另外设置一个后处理部分取决于你需要在前处理部分花费多长的时间以及这个IRQ是否是共享的。只要中断处理器从执行中返回，设备的IRQ就将重新被使能。这就意味着中断的后处理部分是在中断处于激活状态的情况下运行的——与不采用这种做法相比，中断处理器将能够为更多的中断提供服务。后处理部分彼此都是原子化的，因此你不必担心会出现重入方面的问题。前处理部分可以在后处理部分正在执行的时候被再次调用。如果一个后处理部分在它运行的时候又被加上了执行标记，它会尽快地再次得到运行机会；但如果它还排在队列里的时候给加上了两次执行标记，它就会只运行一次。

因为后处理部分是对来自前处理部分的数据进行处理，所以它们之间通常都需要共享一些东西。这需要我们认真对待。我们将在专门的章节里对原子性操作和可重入问题进行讨论。

我们并不是非得使用tq_immediate队列，但这个队列用得最多；原因很简单——它能够最快地得到运行。因为常用的后处理部分都是在内核里预先定义好的，所以你可以在需要的时候用它们来完成自己的工作。

21.6.5 可重入性

可重入性是设备驱动程序必须要注意解决的重要问题之一。我们已经在学习过程中零零散散地介绍过一些这方面的内容，但基本上都是一笔带过；现在，这个问题正式地摆在了我们的面前。我们来看一下你的驱动程序被几个进程同时打开时的情况。通常，在一个驱动程序的内部需要用到多个保存在不同地方的数据结构。问题是这些结构的完整性并不是不可侵犯的，那么，你怎样才能保证那几个进程不会同时对同一个结构进行修改呢？随着SMP系统的日益流行，机器里有两个CPU已经不是什么少见的事情，这就更增加了问题的严重性。Linux 2.0对这个问题的解决方案是用一把大锁把整个内核空间都监控起来，确保任一时刻只有一个CPU在内核里活动。这个解决方案倒是很有效，但随着CPU数量的增加，它就会越来越力不从心。

如果我们想让Linux有能力控制有两个以上CPU的机器并控制得好，就必须采用更精细的加锁机制，这在Linux 2.1版的开发周期里得到了共识。最终结果是舍弃了采用一把大锁并让进程们在进入内核空间之前申请这把大锁的做法，引入了新的加锁指令。内核里重要的数据结构现在是用一把单独的操作锁监控着的，让无数进程运行在内核里的想法如今成为了现实。如果某段代码准备修改的数据结构可能会被其他进程同时修改，这段代码就被称为关键节，而它就是我们需要在重入时加以保护的代码段落。

我们刚才已经提到，要想运行一个内核空间里的进程必须先提出申请，因此你可以放心current不会在你的眼皮子底下被偷梁换柱——它们必须明确地放弃执行。这条规定基本上是正确的——因为还有中断可以在任何时间进入并打断当前的执行流。当然还存在让进程休眠和在驱动程序里明确地进行进程调度等方面的问题——我们必须在这些情况发生时把可重入性问题的处理安排好。那么，这是不是意味着我们必须把所有变量都监控起来呢？当然不是，因为只有全局性的数据结构共享着同样的地址空间。函数的本地变量都存放在各进程专用的内核堆栈里，每个访问进程都有它自己的变量存放地点。

```
int global;

int device_open(struct inode *inode, struct file *file)
{
   int local;

   printk("pid = %d : ", current->pid);
   printk("global = 0x%p, local = 0x%p\n", &global, &local);
   ...
}
```

这是某个模块里的一段代码。它的输出证实了全局变量是共享的，而本地变量则是些彼此独立的拷贝：

```
pid = 909 : global = 0xc18005fc, local = 0xc08d3f2c
pid = 910 : global = 0xc18005fc, local = 0xc098df2c
```

把本地变量放到内核堆栈里的做法是一种解脱，但对能够在其中存放多少东西又不可避免地会有一个限度。Linux 2.2为每个进程保留了大约7KB的内核堆栈，这对大多数应用来说都足够了。这里面有一部分是为中断处理或类似情况保留的——注意不要超越了这个限度。如果你的需要超过了大约6KB，就应该对它进行动态分配。

预防可重入问题的老式做法是：全面禁止所有的中断，做自己的工作，然后重新使能中断。中断处理器们以及运行在中断期间的其他一切事物都必须工作在与你的驱动程序保持着异步的状态下，可能会被这些处理器改变的数据结构必须在你使用它们的时候受到保护。如下所示：

```
unsigned long flags;

/* save processor flags and disable interrupts */
save_flags(flags);
cli();

_read/modify structures, critical section_

restore_flags(flags);
```

这个办法是有效的，但它禁止了所有CPU上的所有中断，因而是一种低效率且又非合作性质的保护措施。如果目标机器确定是UP系统（单CPU系统），因为它只有一个可以执行内核代码的CPU，所以你这样做当然无可厚非。在这种情况下，指令将依次进入CPU执行，彼此之间保持为原子化状态。

21.6.6 单独禁止一个中断

如果你知道只有你自己的中断处理器会修改某个内部数据结构，把系统上的所有中断都禁止掉未免有些小题大做。其实你只需保证当你在修改那些数据结构的时候你自己的中断处理器不会再次运行就足可以了。针对这样的情况，Linux提供了用来禁止单条IRQ链的函数，如下所示：

```
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
void enable_irq(unsigned int irq);
```

代码的关键部分可以放在中断的disable和enable之间，中断的前处理部分在这条中断链动作的同时将不会被再次调用。正常的disable_irq函数和变体的disable_irq_nosync版本之间的区别在于：前者能够保证在返回之前指定的IRQ在任何CPU上的都不会被执行；而后者在禁止了指定的中断之后会立刻返回，不理会前处理器是否正在运行。

21.6.7 原子化操作

如果指令的执行是一次性完成，中间没有间断——即你在完成之前不会被中断打扰，我们就说它是原子化的。我们已经知道：只要禁止了中断，就没有人能够在关键节里中断我们的操作，因而达到原子化操作的目的。Linux还提供了一些对变量进行操作的原子化指令，它们在执行时不把其他人锁在外面。这些指令都是在asm/atomic.h文件里定义的。

```
void atomic_add(int i, volatile atomic_t *v)
void atomic_sub(int i, volatile atomic_t *v)
void atomic_inc(volatile atomic_t *v)
void atomic_dec(volatile atomic_t *v)
int atomic_dec_and_test(volatile atomic_t *v)
```

这些指令的操作对象都是atomic_t类型的数据。这是一个只包含着一个计数器成员的结构，具体包含的是什么东西并不重要，因为你只会通过atomic_x系列函数来访问它。只有这样才能保

证操作的原子性。它们的主要用途是对信号量进行加减，但你可以把它们用在任何地方。

原子化操作通常用来防止出现竞争现象。假设某个进程决定需要休眠在一个事件上，休眠与否要根据某个表达式的求值结果来决定，但如果对这个表达式进行的求值操作是非原子化的，就可能出现竞争现象。Schar用不着这样的构造，但因为这个问题很常见，所以我们在下面给出一个使用示例：

```
/* if device is busy, sleep */
if (device->stat & BUSY)
   sleep_on(&queue);
```

如果判断设备是否忙的测试操作是非原子化的，那么在测试之后但在sleep之前，这个判断条件就可能会变为false，这就有可能使进程永远休眠在队列上。Linux里有一些非常方便的二进制位测试操作是保证以原子化方式进行的。如表21-18所示：

表 21-18

set_bit(int nr,volatile void * addr) clear_bit(int nr, volatile void *addr) test_bit(int nr, volatile void *addr)	用addr地址处的位掩码设置、清除或测试nr里给出的二进制位

把上面那个设备忙测试改为下面这样：

```
/* if device is busy, sleep */
if (test_bit(BUSY, &device->stat)
   sleep_on(queue);
```

就可以完全避免出现竞争现象了。在asm/bitops.h文件里还定义了一些其他的操作，比如测试加设置等。

21.6.8 对关键节进行保护

很明显，认为自己的模块将只运行在UP系统上是非常不智的想法。Linux提供了两种轮转锁机制，它们可以用来向多CPU环境中的数据结构提供保护。在UP系统上，它们将缺省地等同于前面介绍过的调用cli来禁止中断的办法；但在SMP系统上，它们将只禁止本地CPU上的中断——如果驱动程序里的全体函数在修改共享数据结构之前申请使用的都是同一把轮转锁，这就已经足够了。

1. 基本轮转锁

轮转锁是最基本的加锁指令之一。如果一个进程试图进入已经被另外一个进程用轮转锁保护起来的关键区域，就会在原地“兜圈子”，即循环等待锁的释放和可申请。

在asm/spinlock.h文件里可以查到各种类型的轮转锁。如果你对它们在单CPU和多CPU配置情况下的具体实现方法有兴趣，这个文件也是必读材料之一。Linux里的轮转锁有两种基本类型。下面是它们的语法定义：

```
spinlock_t our_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;

spin_lock(&our_lock);
spin_lock_irqsave(&our_lock, flags);
```

如果在中断期间不打算对数据进行修改，spin_lock就足够用了。它能预防来自其他CPU的修改操作，但甚至连本地CPU上的中断都没有禁止。这使它的速度比spin_lock_irqsave要快很多，而后者还提供了针对中断处理器的预防措施。

```
spin_unlock(&our_lock);
spin_unlock_irqrestore(&our_lock, flags);
```

这两个宏定义的作用是在完成数据修改工作后解除相应的轮转锁。

在asm/spinlock.h文件里还有许多其他的函数，比如用来在尝试获取一把锁之前测试能否成功获取这把锁的宏定义等。如果你需要使用进一步的功能，就可以在那里找到需要的资料。

2. 读、写操作锁

刚才介绍的轮转锁提供的是全面的加锁机制，能够在各种目的的重入情况下给代码提供保护。但有时候需要为数据提供更精细的访问控制（比如提供数据的只读、只写等操作）也是很有必要的。Linux为这些目的也准备了加锁机制，允许我们申请获得只读或者只写等操作权限）即允许多个读操作或一个写操作同时进入关键区域。

```
rwlock_t our_lock = RW_LOCK_UNLOCKED;
unsigned long flags;

read_lock(&our_lock);
read_lock_irqsave(&our_lock, flags);
write_lock(&our_lock);
write_lock_irqsave(&our_lock, flags);
```

与基本轮转锁相比，这里的语法现象完全相同，所以我们也用不着多加解释了。类似的宏定义提供了对该区域进行解锁的功能。

```
read_unlock(&our_lock);
read_unlock_irqrestore(&our_lock, flags);
write_unlock(&our_lock);
write_unlock_irqrestore(&our_lock, flags);
```

现在，大家应该知道如何利用基本的轮转锁和读/写操作锁来保护自己免受重入问题的干扰影响了。如果是在一个UP（单CPU）环境里，所有非IRQ版本的函数就都会扩展为空操作（IRQ版本还会禁止中断，这是因为它们是在中断一发生就得到处理的，因此还必须异步地运行），对系统和你代码的性能都不会产生不良的影响；但它们在SMP（多CPU）系统上的效果与基本的cli构造相比就灵活有效的多了。

3. 自动加锁机制

供设备驱动程序使用的大多数函数在内部都已经受到轮转锁的保护了——这要归功于内核，因此不需要额外再加上锁了。在定时器部分的内容里我们已经给出了一个这样的例子：add_timer在对给定的定时器进行处理之前会申请使用内部的timer_list锁。如果那是一个仅在该函数内使用的局部定时器，就不需要进行加锁，直接调用internal_add_timer就行。但我们建议大家要坚持使用“更安全的”函数，加上这几句话的目的是希望大家明白我们为什么没有在Schar里给等待队列加上锁来保护它。

21.7 块设备

我们在这本书里还要学习另外一种类型的设备。它们是与字符设备完全不同的东西，这种

设备吞吐的不是一个一个的字节，而是整块的数据。应用程序访问字符设备时一般都采用直接读写的办法，但访问块设备时就要利用系统里的缓冲区了。

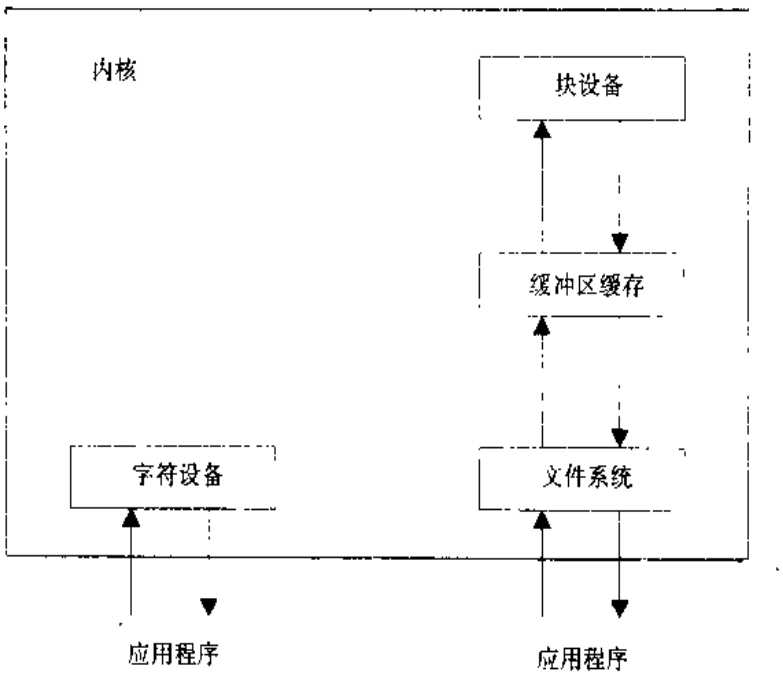


图 21-5

图21-5只说对了一半，这是因为只有数据的读、写操作是需要利用缓冲区缓存的，而open、close和ioctl等操作一半都有专用的入口点。

块设备上一般都容纳着文件系统，只要给出准备对哪些数据块进行读写，就可以对它们进行随机性访问。这与字符设备形成明显的对比，对字符设备只允许进行顺序的非随机性访问，因此很难用来提供对文件系统的存储。

Linux对这两种设备划分并不严格，甚至连程序接口都是相同的。我们在设计我们的第一个字符设备Schar的时候就曾发现：file_operations结构里的部分元素并非仅能用于面向字符的访问机制，两种类型的设备都可以使用它们。

21.7.1 一个简单的RAM盘模块：Radimo

掌握块设备及其底层系统内部工作原理的最佳方法就是扎扎实实地编写一个活生生的例子。Radimo是一个RAM盘驱动程序，它可以容纳一个尺寸变化着的文件系统，具体容量要取决于系统中有多少可用的内存。

请求函数是每一个块设备的核心。它负责接收读写请求并把它们转换为设备能够理解的命令。如果我们是在编写一个IDE驱动程序，请求函数就将生成一系列命令并把它们传递到控制器去，控制器执行这些命令对数据的双向传输进行初始化。模块的开始部分需要对几件事情（包括请求函数在内）按规定的次序做出定义。各种头文件的先后次序就不必多说了，但在包括上linux/blk.h头文件之前，必须先把下面几样东西定义好见表21-19：

表 21-19

#define MAJOR_NR	RADIMO_MAJOR	设备的主编号，这条定义必不可少
#define DEVICE_NAME	"radimo"	设备的名字——可以省略，省略时将被标识为“unknown”（未知）设备。它是出现数据请求错误时将被打印出来的设备名，除此之外没有其他的用处
#define DEVICE_REQUEST	radimo_request	设备的请求函数。它也是必不可少的
#define DEVICE_NR(device)	(MINOR(device))	用在可分区设备上（比如硬盘），设备将利用这个辅编号来选择分区
#define DEVICE_ON(device) #define DEVICE_OFF(device)		电动设备可以通过设置这两个函数来控制该设备的启动和停止。块设备系统会在完成对数据请求的服务之后调用停止（_OFF）函数。DEVICE_OFF必须得到定义，哪怕是一个空定义；DEVICE_ON可以省略
#define DEVICE_NO_RANDOM		如果定义了这个标志，就不允许把这个设备添加到随机数字发生池队列里去。它与中断处理器的SA_SAMPLE_RANDOM标志作用类似

在定义好这几样东西之后，就可以加上linux/blk.h文件了。

1. 数据块的长度

块设备有两种扇区长度，一个是硬件扇区长度，一个是软件扇区长度。前者描述的是数据在设备控制下的硬件介质上的存储格式，而后者描述的是数据在设备里的排放形式。现时期大多数设备的硬件扇区长度是512个字节，部分设备如MO驱动器等通常使用2048个字节的扇区长度。

这些长度值是由设备的初始化过程在以设备主编号为下标的全局性数组里设置的：

```
#define RADIMO_HARDS_SIZE            512
#define RADIMO_BLOCK_SIZE            1024
static int radimo_hard = RADIMO_HARDS_SIZE;
static int radimo_soft = RADIMO_BLOCK_SIZE;

...

hardsect_size[RADIMO_MAJOR] = &radimo_hard;
blksize_size[RADIMO_MAJOR] = &radimo_soft;
```

我们按传统选用512个字节作为硬件扇区长度。这其实并不重要，因为Radimo的数据是保存在一个内部缓冲区里的。软件方面的数据块长度可以取符合下列规则的任何值：它必须等于或大于硬件扇区长度，它必须是硬件扇区长度的整数倍，它还必须小于PAGE_SIZE。如果你没有给出自己的设置，硬件扇区长度的缺省值将取为512个字节，而块长度将取为1024个字节。

全局数组里除扇区长度以外，还保存着设备的总长度。这个长度参数的单位是KB，让内核自动返回“-ENOSPC”（设备上没有空闲空间）。

```
blk_size[RADIMO_MAJOR] = &radimo_size ;
```

有趣的是，如果我们建立了好几个虚拟设备（采用以设备的辅编号为下标的办法），它们的

radimo_size将组成一个数组，数组下标的形式为“[MAJOR][MINOR]”。与块设备有关的各种全局性数据结构的定义都放在drivers/block/ll_rw_blk.c文件里，并且以程序注释的形式进行了说明。

2. 块设备的注册

在完成各项定义之后，设置一个file_operations文件操作结构，如下所示：

```
static struct file_operations radimo_fops = {
    NULL,                           /* llseek */
    block_read,                     /* generic block read */
    block_write,                    /* generic block write */
    NULL,                           /* readdir */
    NULL,                           /* poll */
    radimo_ioctl,
    NULL,                           /* mmap */
    radimo_open,
    NULL,                           /* flush */
    radimo_release,
    NULL,                           /* fsync */
    NULL,                           /* fasync */
    radimo_media_change,
    radimo_revalidate,
    NULL                            /* lock */
};
```

文件操作结构搭建起我们的通信系统。我们一般不需要定义自己的读、写函数，这是因为该结构的缺省配置能够在数据请求到来时调用ll_rw_block把请求排到队列里供我们执行。这些缺省操作可以在fs/block_dev.c和drivers/block/ll_rw_block.c文件里查到。事实上，使用块设备的时候并不需要知道它这些内部的工作原理。我们定义了自己的ioctl函数，为了让块设备能够与标准的文件系统工具命令正确地配合工作，有几个ioctl命令是我们必须支持的，我们一会儿再介绍它们。open和release与它们在字符设备里的作用是一样的，剩下的就是文件操作radimo_media_change了。

块设备的注册工作与字符设备也基本相同，如下所示：

```
res = register_blkdev(RADIMO_MAJOR, "radimo", &radimo_fops);
if (res) {
    MSG(RADIMO_ERROR, "couldn't register block device\n");
    return res;
}
```

21.7.2 介质的更换

块设备在挂装的时候会调用定义了的check_media_change函数来检查介质是否被更换过了。RAM盘驱动程序永远不会遇到需要更换介质的问题，但我们假装这种事情也可以发生。Radimo设置了一个内部的定时器，每隔60秒，如果设备不忙，就清除我们的存储区。同时，它还把系统缓冲区里的缓冲数据设置为已失效，强制实现对它们的重新读取而不是从缓存里提供给应用程序。只要我们返回一个表示介质已经被更换过了的“1”，VFS自己就会把缓冲区设置为已失效；但要等到挂装设备时才会发生这样的事情。如果用户只是简单地用dd或cp命令来拷贝设备的话，缓冲区里的数据不会发生变化。

revalidate与介质的更换是密切相关的。如果介质被更换了，radimo_media_change就将返回

“1”，把这次更换通知给VFS。然后调用revalidate函数让设备刷新它内部保存的设备信息。如果必要的操作动作都可以在介质更换函数里完成的话，就没有必要再专门实现一个revalidate函数；而且，它返回的是什么都没有关系——我们保留这个函数的目的是为了让大家更好地理解块设备的文件操作。

21.7.3 块设备的ioctl文件操作

既然块设备是用来容纳文件系统的，那么让所有的块设备都能接受一些标准的ioctl命令也就再正常不过了。我们在字符设备部分介绍过ioctl的实现方法，如果你还有什么不明白的地方，请复习一下以前的内容。Radimo实现了最常见的ioctl标准命令。如表21-20所示：

表 21-20

BLKFLSBUF	块设备的缓冲区强制写操作。把缓冲区里的当前数据强制性地写到磁盘上。Radimo调用fsync把缓冲区里的当前数据强制性地写到磁盘上，然后把它们标记为已失效
BLKGETSIZE	查看块设备的容量。以1024个字节为单位返回块设备的容量。与文件系统有关的各种工具性命令（比如fsck等）都要通过这个ioctl命令来确定设备的总容量。如果设备不支持这个命令，它们就得猜了
BLKSSZGET	查看块设备的扇区长度。返回块设备的软件扇区长度
BLKRAGET	块设备的预读功能。返回该设备当前读写位置前面的那个数据值
BLKRASET	块设备的预写功能。设置该设备当前读写位置前面的那个数据值
BLKRRPART	重新读取块设备的分区表。fdisk程序会在重写分区表的时候调用这个ioctl命令。Radimo没有分区，因而不支持这个命令

这些函数的实现都比较直白，我们就不在这里把它们列出来了。块设备还有一些其他的标准命令，如果读者有兴趣，可以到linux/fs.h文件里查看一下。Radimo没有实现任何设备专用的命令，可要是真的需要，在那个switch开关语句里加上它们就行了。

21.7.4 请求函数：request

request函数可以说是块设备的脊梁。字符设备接收的是一个数据流，而块设备处理的是数据传输请求。这个请求要么是一个读操作，要么是一个写操作，而request函数的对应工作要么是根据发送来的数据在设备控制的介质上进行检索，要么是把发送来的数据保存到介质上。根据外设的具体情况，request函数执行的动作会有很大的不同。

请求保存在结构列表里的，做为列表元素的结构其类型是“struct request”。这个函数自己是不会去遍历这个列表的，它通过CURRENT宏定义（千万不要把它和任务结构的current进程弄混了）对各个请求进行访问和处理。

```
#define CURRENT (blk_dev[MAJOR_NR].current_request)
```

有关定义都在linux/blk.h头文件里。request结构的具体构成如表21-21所示，我们把与本节讨论的块设备系统无关的部分都省略了——Radimo设备使用的各种数据结构将在我们下面学习它的request函数时做进一步介绍。

表 21-21

CURRENT->	
volatile int rq_status	请求本身的状态，它的取值是RQ_ACTIVE或RQ_INACTIVE二者之一（SCSI子系统有更多的状态取值）。当内核在请求列表里查找一个未使用项目时要用到这个状态信息
kdev_t rq_dev	请求针对的设备。如果驱动程序正管理着几台设备（以它们的辅编号区别），可以从这里用MINOR宏命令提取出有关信息来
int cmd	请求的类型，取值是READ或WRITE二者之一
int errors	可以为每个请求分别建立一个错误状态计数
unsigned long sector, current_nr_sectors	将要被操作的起始扇区和扇区个数
char *buffer	我们读/写数据的地方
struct buffer_head *bh;	与这个请求相关联的缓冲区表头。我们将在缓冲区部分对缓冲区表头做进一步说明

这就是一个请求的详细资料。Radimo在初始化过程中会使用vmalloc分配一个数组，这些数据就保存在这个数组里。当需要为请求提供服务时，它会按照具体的操作指令把数据拷贝出、入CURRENT->buffer——各个扇区都是这个存储区数组里的一个偏移量。request请求函数有一个比较特殊的格式——我们先来看看它的Radimo版本，代码清单的后面有一些具体的说明。

```
void radimo_request(void)
{
    unsigned long offset, total;

radimo_begin:

    INIT_REQUEST;

    MSG(RADIMO_REQUEST, "%s sector %lu of %lu\n",
             CURRENT->cmd == READ ? "read" : "write",
             CURRENT->sector,
             CURRENT->current_nr_sectors);

    offset = CURRENT->sector * radimo_hard;
    total = CURRENT->current_nr_sectors * radimo_hard;

    /* access beyond end of the device */
    if (total+offset > radimo_size*1024) {
       /* error in request  */
       end_request(0);
       goto radimo_begin;
    }

    MSG(RADIMO_REQUEST, "offset = %lu, total = %lu\n", offset, total);

  if (CURRENT->cmd == READ) {
     memcpy(CURRENT->buffer, radimo_storage+offset, total);
  } else if (CURRENT->cmd == WRITE) {
     memcpy(radimo_storage+offset, CURRENT->buffer, total);
  } else {
     /* can't happen */
     MSG(RADIMO_ERROR, "cmd == %d is invalid\n", CURRENT->cmd);
     end_request(0);
  }

  /* successful */
  end_request(1);

  /* let INIT_REQUEST return when we are done */
```

```
    goto radimo_begin;
}
```

宏定义INIT_REQUEST负责好几个动作。它首先检查CURRENT是否包含着一个请求，如果没有就返回。接着它对请求进行合法性检验（请求是针对这个设备的吗？），再检查CURRENT->bh（后处理队列）上的操作锁是否已经安排好了。就是因为要检查CURRENT里是否包含着一个请求，我们才会反复跳转到INIT_REQUEST处开始循环——如果请求都处理完了，它就会让我们返回到用户空间的调用者那里。存储区数组里的偏移量通过对给定的起始扇区CURRENT->sector和我们的硬件扇区长度做乘法得到。需要传输的数据总量也采用类似的方法根据CURRENT->current_nr_sector给定的扇区总数计算出来。接下来，根据CURRENT->cmd指明的方向用一个简单的memcpy命令完成对数据的拷贝。

```
void end_request(int uptodate) 结束CURRENT请求
```

如果uptodate的值是“1”，就表示成功地满足了这个请求。CURRENT将被设置为下一个请求。如果下一个请求还是针对Radimo的，控制还将回到我们手中并由radimo_request继续进行处理；如果不是针对Radimo的，就由其他request函数出场表演。

如果请求无法满足（比如出现给定扇区位置超出设备边界等情况时），uptodate的值将是“0”，end_request将被调用。这将在系统日志里记下一个I/O错误，记载内容包括出错设备的名字和引起错误的扇区。在我们的例子里，请求是通过阻塞方式的读写函数接收的，收到的请求都是经检查没有越界的。如果把blk_size[RADIMO_MAJOR]设置为“NULL”，在创建请求时就会把这个简单的检查绕过去，再出现设备访问越界时就会产生一个读操作错误。

```
radimo: read sector 4096 of 2
end_request: I/O error, dev 2a:00 (radimo), sector 4096
```

错误信息里包括十六进制的设备主、辅编号和我们在DEVICE_NAME里定义的设备名。

21.7.5 缓冲区缓存

从块设备上读写到的数据块会在缓冲区里得到缓存。这改善了系统的整体性能，如果一个进程想读一个刚读过或刚写过的数据块，它就可以直接从缓冲区里被提取出来而不必再向介质发出一新的读操作命令。从内部原理上看，整个缓冲区是由一系列缓冲区表头结构组成的双向列表，这些表头的下标组成了一个哈希表。大家可能没注意我们已经在请求函数radimo_request里与缓冲区表头结构擦肩而过，可CURRENT->buffer指针指向的确实是一个缓冲区表头里面的数据区。

如果我们把RADIMO_REQUEST加到MSG宏定义里再运行Radimo，就可以清楚地看到请求函数对请求进行处理的全过程。举个例子，即使我们发出一个读命令对相同的10个数据块进行两次读操作，Radimo也只会读它们一次——第二次请求的数据是从缓冲区里得到的。写操作方面的情况也是如此——你可以试着把同样的数据库写到设备上去，但它们并不是立刻得到处理的。在真正被写到设备上去之前，数据块会在缓冲区里停留一段时间。即使Radimo不把发送给它的数据拷贝到自己的内部存储区里去，只要被缓存的数据还停留在缓冲区域里，设备就能够正常

工作。随内核而来的RAM盘模块rd（请参考/drivers/block/rd.c文件）使用的就是这个原理。rd模块没有自己的内部存储区，它把自己用到的缓冲区标记为锁定状态，以此保证它们将停留在缓冲区域里而不会被添加到空闲缓冲区表上去。如果我们定义了Radimo模块的RADIMO_CHEAT_REQUEST标志，它也会模仿出这种行为。

缓冲区表头结构可以在linux/fs.h头文件里查到。对其细节的分析超出了本书的讨论范围，这里只介绍它的几个状态标志——Radimo模块中有几个函数会间接地用到它们（见表21-22）。

表 21-22

BH_Uptodate	缓冲区里的数据与介质磁盘上的数据完全一致
BH_Dirty	缓冲区里的数据已经发生了变化，需要写到介质磁盘上
BH_Lock	缓冲区被锁定，不能添加到空闲缓冲区表里去
BH_Protected	
BH_Req	如果缓冲区失效，就会清除这个标志位

Radimo模块会用到缓冲区域里的许多个缓冲区，而调用invalidate_buffers函数将解除这些缓冲区与Radimo之间的关联关系。它将对自己使用的所有没有设置上BH_Lock标志的缓冲区进行清理，释放它们，使它们能够重新投入使用。

动手试试：Radimo设备

Radimo是Wrox站点源代码下载包里包括的最后一个模块。我们将把它顺理成章地放在modules/radimo子目录里。和往常一样，在对这个设备进行测试之前需要先对源代码进行编译并把模块插入到内核，还要创建一个与之对应的特殊文件。如下所示：

```
# make
# mknod /dev/radimo b 42 0
# insmod radimo.o
radimo: loaded
radimo: sector size of 512, block size of 1024, total size = 2048Kb
```

我们在书中给出的所有选项都可以在加载时进行指定，只要给insmod命令加上适当的参数就行。这些选项可以在Radimo内容的开始部分查到，也可以通过modinfo命令把它们找出来。如果只进行我们这里介绍的测试，使用这些选项的缺省值效果就不错。

加载上这个模块之后，我们就可以在设备上创建文件系统了。具体创建哪一种文件系统都无所谓，我们使用的是ext2类型的文件系统。如下所示：

```
# mke2fs /dev/radimo
# dmesg | tail -n1
radimo: ioctl: BLKGETSIZE
```

因为我们自己在Radimo里实现了ioctl文件操作的BLKGETSIZE调用，mke2fs命令用不着进行猜测就能查出并利用上这个设备的全部容量。接下来，我们可以挂装上文件系统并对它来回拷贝文件——就象你对普通硬盘进行操作那样。

```
# mount -t ext2 /dev/radimo /mnt/radimo
# cp /vmlinuz /mnt/radimo
```

这些命令都不是什么新概念了，我们也用不着多说。用umount命令卸下设备，等待60秒（这个数字在radimo.h文件里被定义为常数RADIMO_TIMER_DELAY），测试它的介质更换机制。然后，再挂装它：

```
# umount /dev/radimo; sleep 60
# mount -t ext2 /dev/radimo /mnt/radimo
mount: wrong fs type, bad option, bad superblock on 'dev/radimo,
       or too many mounted file systems
# dmesg
radimo: media has changed
VFS: Disk change detected on device radimo(42,0)
radimo: revalidate
```

我们的定时器已经在上一次挂装时运行过了，它把media_changed标志设置为“1”。这将使radimo_media_change向VFS返回一个“1”，表示介质已经被更换了。VFS相应地输出一条信息报告这一情况，然后调用radimo_revalidate让设备完成它应该在磁盘更换时完成的必要处理。我们没有专门编写这方面的代码，所以最终的结果是挂装操作失败了。

21.7.6 小结

虽然我们讲了这么多内容，可这也就能算是对块设备的一个简单介绍而已，我们重点介绍了request请求函数、与之相关的数据结构和块设备的ioctl文件操作调用。Radimo是一个非常简单的驱动程序，无法全面体现块设备驱动程序与真实硬件外设之间的互动关系。典型的真实硬件都会利用中断来控制数据的流动，在这种情况下，请求函数无法立刻判断出操作请求是否成功地被完成了。这些问题一般都要由中断处理器来解决，设备发出操作结束的信号，中断处理器判断操作是否成功完成。由中断来驱动的块设备在内核里有各种各样的实际例子；如有必要，你可以自己挑一个来研究。

块设备经常用来存储文件系统，所以它们一般都支持基于分区的访问操作。Linux通过头文件linux/genhd.h里定义的partition和gendisk结构提供了基础性的分区支持。它的具体实现可以在子目录drivers/block里的gendisk.c文件里查到，这个子目录里还有各种辅助性驱动程序。给Radimo加上分区支持并不困难，从进一步了解基本的磁盘子系统方面考虑，这将是一个很好的练习。这就留给你们大家自己去完成吧！

内核里的块设备大都可以归入某个特定的类，一般被称为SCSI驱动程序、IDE驱动程序或者CD-ROM驱动程序等。这些都可以被看做是普通的块设备，而Linux还为它们准备了一些特殊的程序设计接口，大家可以在工作中加以利用。内核中现成的例子、可以在子目录Documentation/里或者在Linux Documentation Project（Linux文档项目）站点上查到的文档都是我们研究这方面课题的无价之宝。

21.8 调试

设备驱动程序和普通应用程序有一点是毫无二致的，那就是它们几乎都不可避免地存在着漏洞和程序错误——即我们常说的“bug”。

内核级的代码不存在普通意义上的内存段冲突错误，出现错误时也不会产生“良好的”核

心映象（core dump——类UNIX操作系统下的应用程序在出现致命错误时会把当时的内存映象保存到一个文件里，这个文件就叫做核心映象文件），所以设备驱动程序的调试工作与普通的调试差别很大。有这么一种说法，即人脑是最好的调试器，说得实在是有道理。当设备驱动程序出现问题的时候，我们能够采取的最佳措施就是回到源代码上逐字逐句地进行分析推敲。Linux的源代码全部是公开的，这对我们来说是一个天大的好事——好好利用它吧。能够使用调试器当然是好事情，但必须保证最后的结局是捉住了真正的“虫子”而不是用创可贴之类的东西把真正的问题掩盖起来。

21.8.1 Oops追查法

Oops追查法是我们必须掌握的一项基本调试技能。内核里的大多数bug大都以NULL退化指针（即不再具有索引功能的NULL指针，也叫做失效指针）的面目出现。根据它们出现的位置，内核往往还能够继续运行。这与应用程序里的内存段冲突错误极其相似，但此时不会生成一个核心映象文件。Oops信息在不同的CPU处理器上有不同的格式，以下内容的讨论对象是x86体系结构下的Oops追查法。但核心映象的解码过程对各种计算机平台来说基本上都是一致的，因此本小节里的内容也同样适用于非Intel体系结构。我们来看看下面这条Oops信息：

```
Unable to handle kernel paging request at virtual address 01380083
current >tss.cr3 = 06704000, %cr3 = 06704000
*pde = 00000000
Oops: 0000
CPU:    1
EIP:    0010:[<c0144040>]
EFLAGS: 00010202
eax: c0144000   ebx: 01380083   ecx: 00000005   edx: c64e8550
esi: c64e9c20   edi: c5f17f84   ebp: 00000000   esp: c5f17f3c
ds: 0018   es: 0018   ss: 0018
Process bash (pid: 390, process nr: 32, stackpage=c5f17000)
Stack: c5f17f84 c64e859c c64e859c ffffffe c012dfaf c64e8550 c64e9c20
       c5f17f84 00000000 c60dd004 00000001 c012e17a c64e9620 c5f17f84
       c60dd000 c60dd000 c5f16000 bffff6b0 c60dd002 00000002 000006b3
       c012e26c c60dd000 c64e9620
Call Trace: [<c012dfaf>] [<c012e17a>] [<c012e26c>] [<c012c232>] [<c0108be4>]
Code: 66 83 3b 00 74 4e 31 c9 8b 74 24 20 66 8b 4b 02 3b 4e 44 75
```

如果这是你第一次看到Oops信息，上面这一大堆东西看起来可够唬人的。它分为以下几个部分：错误发生时各个CPU寄存器的映象、一个堆栈追溯、一个函数调用追溯以及一段引发这个Oops的代码清单。如果我们不能把看到的地址与实际的函数名称联系起来，这一堆东西也就没有什么用处。ksymoops工具可以为我们完成这一工作——它还有许多其他用途。它一般存放在你内核源代码目录下的scripts/ksymoops子目录里。在ksymoops里运行上面这个Oops，我们将看到一些如下所示的内容：

```
Unable to handle kernel paging request at virtual address 01380083
current >tss.cr3 = 06704000, %cr3 = 06704000
*pde = 00000000
Oops: 0000
CPU:    1
EIP:    0010:[<c0144040>]
EFLAGS: 00010202
eax: c0144000   ebx: 01380083   ecx: 00000005   edx: c64e8550
esi: c64e9c20   edi: c5f17f84   ebp: 00000000   esp: c5f17f3c
ds: 0018   es: 0018   ss: 0018
```

```
Process bash (pid: 390, process nr: 32, stackpage=c5f17000)
Stack: c5f17f84 c64e859c c64e859c ffffffe c012dfaf c64e8550 c64e9c20
       c5f17f84 00000000 c60dd004 00000001 c012e17a c64e9620 c5f17f84
       c60dd000 c60dd000 c5f16000 bffff6b0 c60dd002 00000002 000006b3
       c012e26c c60dd000 c64e9620
Call Trace: [<c012dfaf>] [<c012e17a>] [<c012e26c>] [<c012c232>] [<c0108be4>]
Code: 66 83 3b 00 74 4e 31 c9 8b 74 24 20 66 8b 4b 02 3b 4e 44 75

>>EIP: c0144040 <proc_lookup+4c/e0>
Trace: c012dfaf <real_lookup+4b/74>
Trace: c012e17a <lookup_dentry+126/150>
Trace: c012e26c <__namei+28/58>
Trace: c012c232 <sys_newstat+2a/8c>
Trace: c0108be4 <system_call+34/38>
Code:  c0144040 <proc_lookup+4c/e0>            00000000 <_EIP>:
Code:  c0144040 <proc_lookup+4c/e0>               0:    66 83 3b 00     cmpw
$0x0,(%ebx)
Code:  c0144044 <proc_lookup+50/e0>               4:    74 4e           je     54
<_EIP+0x54> c0144094 <proc_lookup+a0/e0>
Code:  c0144046 <proc_lookup+52/e0>               6:    31 c9           xorl
%ecx,%ecx
Code:  c0144048 <proc_lookup+54/e0>               8:    8b 74 24 20     movl
0x20(%esp,1),%esi
Code:  c014404c <proc_lookup+58/e0>               c:    66 8b 4b 02     movw
0x2(%ebx),%cx
Code:  c0144050 <proc_lookup+5c/e0>              10:    3b 4e 44        cmpl
0x44(%esi),%ecx
Code:  c0144053 <proc_lookup+5f/e0>              13:    75 00           jne    15
<_EIP+0x15> c0144055 <proc_lookup+61/e0>
```

在把地址解析为函数名的时候，ksymoops需要用到当前运行内核的系统地图。内核在建立过程中会在内核源代码子目录的根位置产生或刷新一个系统地图文件，一般情况下就是/usr/src/linux/System.map文件。但这个文件里只能告诉我们内核在编译时都包括有哪些文件，因为它不可能知道某个特定的模块最终会被加载到什么地方。要想知道模块化代码的具体加载位置必须在使用insmod命令加载模块时给它加上“-m”选项。模块向外界提供的导出符号也会出现在/proc/ksyms里。

这个Oops的函数调用追溯从system_call开始，沿途经过Oops里列出来的各个函数，最后到达出现了错误的proc_lookup。追查法按下面的格式把沿途的函数列出来：

```
<function_name+offset/length>
```

offset偏移量是function函数里的跳转出发点，length是这个函数的总长度。引起错误的代码经反汇编以后显示给我们。出现在proc_lookup函数0x4c偏移量位置上的是一个对ebx寄存器进行的字比较操作，通过查看寄存器映象我们得知出现在其中的是一个非法地址。接下来，我们需要找出包含着这个函数的文件，看看到底是什么引起的这个问题。在我们的这个例子里，proc_lookup函数看起来象是与proc文件系统有关，而我们也确实在fs/proc/root.c文件里找到了这个函数。内核里的Makefile制作文件允许我们通过执行“make fs/proc/root.s”命令的办法得到一个包含着调试信息的汇编语言清单。用一个编辑器打开root.s文件，找到那个proc_lookup函数和引起问题的操作，root.c文件里的行号也会出现在root.s文件里。检查结果表明，传递给proc_lookup的dentry指针完全是失效的。

事情才刚刚开始——你还得继续查明为什么会出现这样的事情。这也许是一个必不可少的操作，要是这样的话，就必须加上一个指针失效与否的检查操作。但问题更有可能是出在我们正在编写的驱动程序身上。这个Oops出现在我们对Schar里实现的proc文件系统接口进行测试的时

候。问题的根源终于被找到了：当有人在访问proc注册项的时候，这个模块从内核里被去掉了，与它关联着的子目录注册项也就被释放了。因此，下一次查看这个注册项的时候，传递过来的dentry就不再是有效的了。对Schar的修补措施是：在有人在访问它的proc注册项时增加它的模块使用计数。问题就这样解决了。

21.8.2 对模块进行调试

我们不可能象对待普通应用程序那样对内核代码采用单步执行方式进行调试，至少不能一上来就这么做。调试你自己模块的最佳手段是在问题区域里有策略地增加一些printk语句，然后再从头开始执行到那个地方。但要注意的是有些难以查出来的bug却会因为增加了一个简单的printk语句而消失无踪，而原因只不过是printk语句对操作定时或数据对齐方面的细微影响。但这仍不失为一个好办法，来看看接下去该怎么办。如果模块挂起来了，就必须查出是在哪里挂起的，把当时的关键变量打印出来。Oops信息可以用刚才演示的技巧进行解码。此外，kymoops会把/proc/ksyms缺省地包括进来，这就意味着它还能够对被加载模块提供给外界的各个导出函数进行解码。我们建议大家在开发过程中把所有的函数和变量都导出来，确保ksymoops能把它们都抓住。

21.8.3 “魔术键”

最不幸事情就是遇上能够使整个系统彻底崩溃的bug们了。如果真的遇到这样的情况，人们称之为“魔术键”的SysRq按键——也叫做“System Attention Key”（SAK键，系统呼唤键）——就可能会帮上你很大的忙。这个便利的功能是在2.1版的开发期里添加进来的，你可以在配置内核的时候激活这个选项。它只是解除系统完全死机现象的一个办法，不会给正常的系统操作增加任何开销。因此，不管你是否正从事着内核的开发工作，永远要激活这项功能。同时按下Alt、SysRq和一个命令键就可以激活不同的命令。在Documentation/sysrq.txt文件里可以查出各种可用的命令——我们将在这里看一下“p”命令的用法。我们来按下“Alt SysRq p”组合键：

```
SysRq: Show Regs

EIP: 0010:[<c0107cd0>] EFLAGS: 00003246
EAX: 0000001f EBX: c022a000 ECX: c022a000 EDX: c0255378
ESI: c0255300 EDI: c0106000 EBP: 000000a0 DS: 0018 ES: 0018
CR0: 8005003b CR2: 4000b000 CR3: 00101000
```

这是一个处理器状态的明细表，包括各个标志和寄存器。EIP是指令寄存器，它给出的是内核当前执行指令的位置，需要查看内核的符号图才能确定这个值所对应的地点。我们的内核与这里给出的EIP值最接近的匹配是：

```
c0107c8c T cpu_idle
c0107ce4 T sys_idle
```

这表明我们的内核现在正在cpu_idle里执行着。如果有个模块陷在一个无限循环里了，“SAK p”就能告诉你它到底发生在什么地方——前提是进程调度器还在运行中。系统也可能是完全死机，真要是这样，SAK也帮不上你了。

21.8.4 内核调试器——KDB

使用内核调试器既可以安全地对一个运行中的内核进行调试，又能够把对正常的系统操作的影响降低到最小程度。但要想在Intel平台上使用这个办法还必须先给内核打补丁，这是因为该功能目前还没有被集成到内核里去——也许将来会吧。其他一些平台也提供有类似的功能，而且还不需要对内核打补丁。请读者自行检查自己的内核配置情况，如有必要，可以再到因特网上搜索一番。

这个调试器可以在不同地点被启动。你可以在开机引导时向lilo传递一个“kdb”参数使调试器尽快被启动。然后，在系统操作过程中，按下键盘上的Pause键就可以手动进入调试器；而如果出现了一个Oops错误，调试器将自动被启动。你可以在调试器里做的事情包括：查看和修改CPU寄存器或变量的值、单步执行、设置断点，等等。

这个补丁自带的使用手册内容充实，对它的用法做了详细的介绍，我们就不在这里多浪费笔墨了。这个调试器很容易使用，但它与gdb的配合并不是很好，而且在功能方面与gdb相比也要弱一些。虽然与我们期望的内建调试器相比还有一段差距，但它仍不失为一个非常方便的工具。进入调试器，设上一个断点，然后在到达断点时让调试器自动启动——这使我们能够在不过多影响系统整体性能的前提下单步调试自己的模块。

21.8.5 远程调试

我们在第9章里学习了gdb的使用方法，对它在普通应用程序方面的应用已经有了一定的认识。给内核打上kdb补丁之后，运行中内核的调试工作就完全可以利用一条串行线通过gdb来进行，就象对待其他任何程序一样。这个方法需要使用两台计算机：一个是用来对gdb的运行情况加以调控的主控机，另一个是被调试的受控机。两台计算机不使用调制解调器直接用电缆连接起来，最好用Minicom或其他类似的终端程序测试一下这个连接以保证链路通畅而又可靠。

给受控机打上内核补丁之后，重新编译，再重新启动。如果我们不执行该补丁自带的一个调试脚本，机器的运行情况与往常并没有什么不同。在内核里设上一个断点，然后启动调试脚本，受控机将停止运行，把控制移交给主控机。gdb在主控机上的使用方法和平时一样——用“continue”命令恢复受控机的运行，如果用户按下“Ctrl-C”组合键或者它执行到达一个断点时受控机的执行将再次停止。

KDB补丁还提供了串行线调试功能，可如果你真的有两台机器，还是打上kgdb补丁更好一些。kgdb提供了通向功能更强大的gdb调试器的操作接口。是否需要选用那些高级功能就全在你个人了。

21.8.6 调试工作中的注意事项

在内核进行调试有各种各样的手段，虽然它们手法各异，但都还有一些共性。一般来说，对断点必须特别注意。如果你使用的是一个集成性的调试器，就应该尽量避免某些键盘处理过程里设置断点。如果是通过网络连接而进行的远程调试，在驱动程序或网络堆栈里设置断点时也要考虑这个建议。它有可能达到目的，也有可能根本就不管用——可要是还执行了fsck命令的

话，你就准备好关电源喝咖啡吧——它几乎百分之百地会出问题！有些驱动程序对时间关系的要求很严格，对它们进行单步调试也有可能是白费工夫。中断处理器和定时器处理器也存在这样的问题。有些断点是不能设的，可在你明白这个道理之前也许已经在调试内核时遇上好几次死机现象了——要有应付系统崩溃的思想准备。

以我们的经验看，专门用一台机器来进行测试和调试是最灵活的解决方案。而且这个解决方案的花费也用不着很大——我们使用的就是一台老式的486机器，我们把它连接到我们的主工作站上，通过网络来启动它，再使用NFS来挂装它的根文件系统。这台测试用机里只有一块带少量RAM内存的主板、一块网卡、一个软盘驱动器和一块廉价的显卡。开发工作在工作站上完成，而测试和调试工作则在这台测试用机上独立完成。如果它真的崩溃了，因为不需要检查恢复文件系统，它只要不到半分钟的时间就能够完成重启动，而我们对源代码的编辑修改工作可以不受影响地在工作站上进行。再加上一根串行线，对测试用机的远程调试就更有把握了。

21.9 可移植性

既然编写的是设备驱动程序，自然应该能够运行在尽可能多的计算机平台上。发展到了今天，各种计算机平台上的API其可移植性都非常好，容易引起问题的主要就是平台本身之间的差异了。大多数可移植性方面的问题我们都在容易产生这类问题的有关章节讨论过了。这小节再对其中的一些做进一步的探讨，同时再介绍几个新问题。

21.9.1 数据类型

在本章开始的“数据类型”小节里，我们向大家介绍了_uXX和_sXX数据类型。在需要使用特定长度尺寸的变量时，使用这些数据类型永远是个好主意，这是因为（比如说）我们不能保证long长整数在所有的计算机平台上都有相同的长度尺寸。

21.9.2 字节的存储顺序

Intel和Alpha等体系结构的计算机平台采用的是字节的降序存储方式，也就是说，最高位和最低位的字节值在内存里是颠倒过来存放的。而同样能够运行Linux的Power PC和SPARC体系结构的CPU则是一些采用字节升序存储方式的平台，它们按数据原来的字节顺序存放它们。这也是C语言看待事物的方式，人们也更熟悉这样的写法。大部分时候用不着你去关心目标机器上的字节存储顺序，可如果你真的需要建立或者检索某个特定字节存储顺序的数据，就需要在这两者之间进行转换。升序存储需要定义_BIG_ENDIAN_BITFIELD标志，而对降序平台则要定义_LITTLE_ENDIAN_BITFIELD标志。有关代码要放在定义检查的中间，如下所示：

```
#if defined( __LITTLE_ENDIAN_BITFIELD)
   byteval = x >> 8;
#else
   byteval = x & 0xff;
#endif
```

Linux还准备了一些用来对变量的字节存储顺序进行转换的指令。这些指令涉及面很广，它们都定义在头文件linux/byteorder/generic.h里。其中最常用的（至少在Intel平台上）见表21-23。

表 21-23

unsigned long cpu_to_be32(unsigned long x)	把变量x从CPU使用的字节存储顺序转换为升序存储方式
unsigned short cpu_to_be16(unsigned short x)	
unsigned long be32_to_cpu(unsigned long x)	把以字节的升序存储方式保存的变量x转换为CPU使用的字节存储顺序
unsigned short be16_bo_cpu(unsigned short x)	

还有许多其他的函数是用来完成对16位、32位、64位变量的字节存储顺序进行转换的；它们都可以在刚才提到的头文件里查到。

21.9.3 数据的对齐

如果一个数据在内存里的存放地址能够使CPU处理器以最有效的方式对它进行访问，我们就说这个数据是正确对齐了的。非对齐数据的访问和访问非对齐数据的后果取决于CPU处理器的具体类型。如果计算机的体系结构允许非对齐访问，后果会是执行速度的降低；如果计算机的体系结构不支持非对齐访问，后果将是操作失败。

```
get_unaligned(ptr)          访问非对齐数据。
put_unaligned(val,ptr)
```

如果你确实需要访问某个肯定是非对齐的数据，就请使用表21-24里的宏定义。它们的定义在asm/unaligned.h头文件里。如果计算机的体系结构直接支持非对齐访问，这几个宏定义将展开为普通的退化指针。

其他可移植性方面的问题已经在各有关章节里介绍过了，你应该不会再遇到我们没有在这里或那里讲到的问题。可移植代码是优美的代码，大家要在程序设计工作中多考虑这方面的问题！

21.10 本章总结

我们希望大家能够体会到内核开发工作中的乐趣。内核源代码的长度实在是太大了，所以它经常给人以一种不可企及的感觉，但我们希望这一章内容能够给大家提供足够的装备和各自为战的信心。对这本书的学习无论如何也比不上你自己对源代码的研究。毫无疑义，源代码是惟一能够完全跟上潮流变化的文档！这确实是开放源代码项目赋予程序员的强大武器，整个源代码都毫无保留地呈现在你的面前任你研究。Linux内核方面的文档能够称得上好的实在是不多，所以阅读他人编写的驱动程序和内核改进就成为我们大家能够拥有的最佳资源。

完全掌握Linux内核及其设备驱动程序的道路到处是崎岖险阻。我们希望这个简单的介绍能够帮助大家开始起步去追求自己的目标。为了一个深藏不露的“小虫子”，我们可能会经历许多个头悬梁锥刺骨的不眠之夜，但我们最终捉住了它；也许正是这些才激励着你去孜孜不倦地研究源代码和不断地学习。有些内核开发人员极具天份，他们的工作绝对值得我们花费时间和精力去学习。

要想成为内核程序员需要克服很多的困难，而最大的障碍也许就是不知道某个特定的代码段被存放在内核源代码的什么地方。我们希望这一章能够给大家一个好的开始，对内核里的来龙去脉有

一个大致的认识。如果你在某个具体问题上遇到了困难，grep命令会一如既往地帮你查到所有相关示例。但正如我们在这一章开始时说的那样，内核永远在发展当中，一本印刷出来的书已经肯定不是最新最全面的了。如果大家发现这里写的内容与内核里的实际情况不一致，请以内核为准。

21.11 内核源代码解剖图

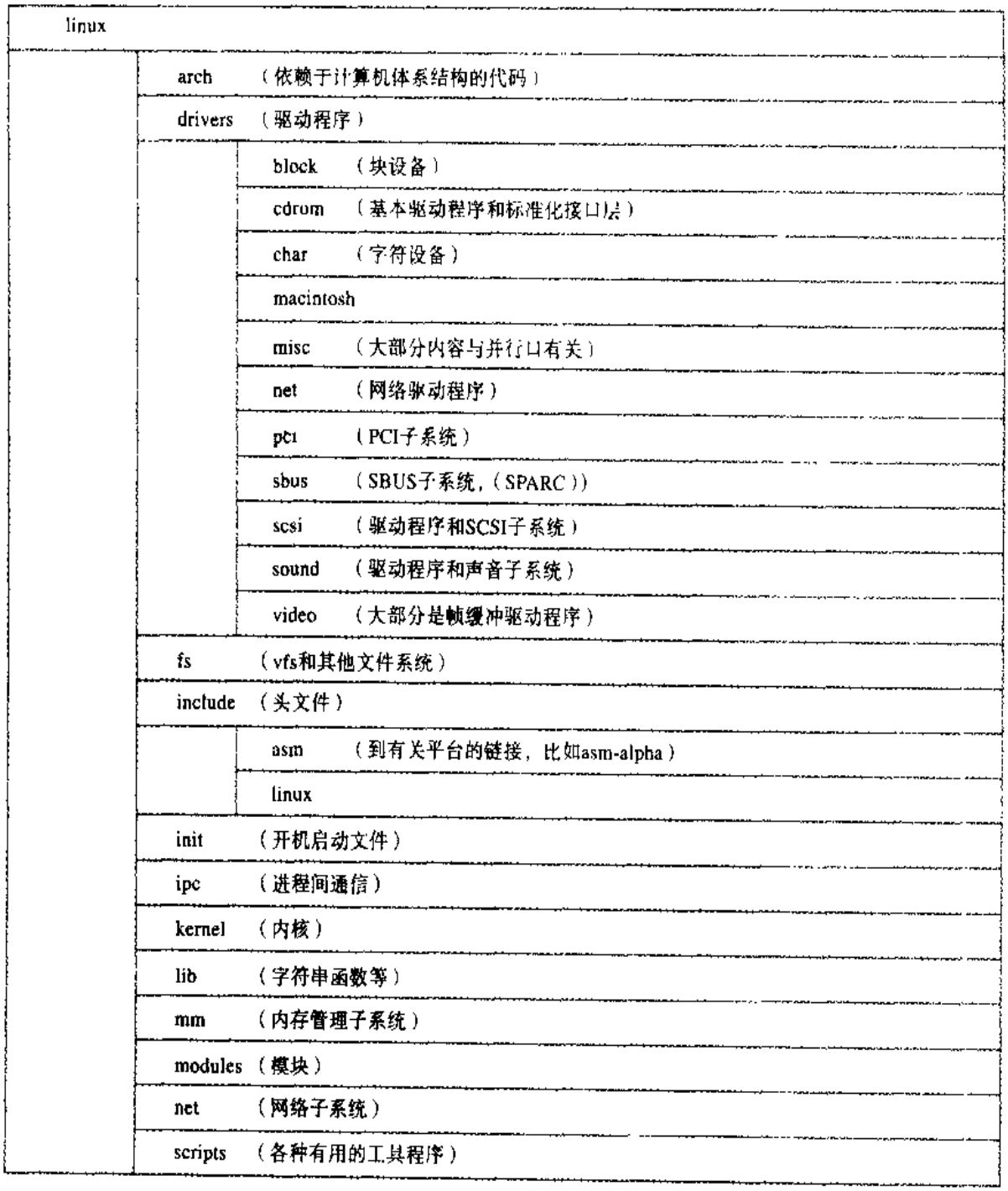


图 21-6

图21-6就是内核的基本结构，但我们省略了许多内容，希望大家能够把它补充上。

附录A 可移植性

可移植性涉及到的问题主要有三个方面：

- 不同操作系统之间的可移植性，它们可以是UNIX操作系统的其他实现形式，也可以是完全不同的操作系统，比如OS-9、VMS或MS-DOS等。
- 不同供应商的编译器之间的可移植性，包括使用不同的C语言编译器和做为C++代码进行编译等方面。
- 不同硬件之间的可移植性，比如字长度、字节存储顺序以及其他可能导致问题的差异等。

我们在这本书里断断续续地提到过不同操作系统之间的可移植性问题，而这个附录的主要论题是其他方面的可移植性。

UNIX世界里的可移植性永远是非常重要的，这是因为各种计算机平台上曾经有过许多版本的UNIX操作系统。截至到不久前，Linux还只能用在80x86系列的机器上；可现在Linux已经能够用在其他平台上了，其中最值得注意的是Motorola公司出品的68k系列和DEC公司出品的Alpha处理器。

我们在这个附录里准备了一些与可移植性有关的技巧、窍门和一般性论述，目的是为了帮助大家保证自己的软件能够在不同的UNIX平台和不同的编译器版本之间进行移植。

A.1 程序语言方面的可移植性

首先，我们来看看你需要采取哪些措施才能保证自己的代码能够在不同版本的C语言编译器和不同版本的UNIX操作系统上进行移植。

A.1.1 预处理器符号

有几个预处理器符号是用来指示编译器类型和编译器与有关标准之间符合程度的。这些符号的定义没有采用头文件这种普通的方法，它们是由预处理器或编译器系统提供的。这些符号对可移植性的影响很大。

1. __STDC__

也许最重要的预处理器符号就是“__STDC__”了。只要编译环境能够满足ISO（国际标准化组织）的C语言标准，就必须定义这个符号。从传统上讲，它经常被用来在只能编译“Kernighan & Ritchie”原始格式的老式系统和紧随C语言标准发展动向的新式系统之间移植代码。

现如今，尽管很多编译器还没有通过认证——也就是说多多少少地与C语言标准还有细微的差异，但明显不同于C语言标准的编译器可以说是少之又少了。使用这个符号将使你能够把需要按ISO的C语言标准而不是按老式的“K & R”惯例来对待处理的代码部分隔离出来。它们之间

的大部分区别在于函数的定义格式方面。

使用这个符号需要在程序里加上如下所示的代码：

```
#ifdef __STDC__
    int my_function(const char *mystring, const int a_value);
#else
    int my_function();
#endif
```

如果你使用的是GNU编译器gcc，请使用-ansi开关来激活标准的C编译处理。

有些编译器把“__STDC__”符号定义为“0”，这很明显地表明它们与ISO的C语言标准是不相符合的！如果你真的遇见了这样的情况，就不得不使用“#if __STDC__”做为替代办法。

2. _XOPEN_SOURCE

如果你的代码符合（或者估计符合）X/Open和POSIX标准，就请在包括任何头文件之前先行定义“_XOPEN_SOURCE”符号。

这个定义会改变头文件的行为，使它们与X/Open标准保持一致。如果你已经定义了“_XOPEN_SOURCE”符号，就不必再定义老式的“_POSIX_SOURCE”符号了。

它的使用方法如下所示：

```
# define _XOPEN_SOURCE
# include <unistd.h>
. . .
```

A.1.2 系统保留字

有一些名字是系统保留使用的。ISO的标准C语言里就有这样一个保留字清单，但要注意某些非标准化的编译器还可能保留了一些其他的名字。遇到这类情况时就请以编译器的参考手册为准。

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

此外，如果你希望在今后把自己的程序由C语言转换到C++，那么最好也要避免使用任何C++的关键字。在我们编写这本书的时候，ANSI（美国国家标准化协会）的C++标准仍在制定当中，所以现在就给出一个完整的清单是不可能的。但不管怎样，你至少要避免使用下面这些保留字：

asm	catch	class	delete
friend	inline	new	operator
private	protected	public	template
this	throw	try	virtual

还有一些其他的名字也需要保留。需要特别注意的有：以一个下划线字符“_”打头所有名字都是保留给编译器使用的；以“_t”结尾的名字又都是保留给POSIX头文件里使用的。也就是说，永远不要用“_”做名字的开始字符，也不要用“_t”来结束它。

只要你开始在自己的程序文件里包括上头文件，就会使用到额外的名字，而这就可能有不良的后果。我们来想象一下这样的情况：你已经有一个能够正确编译的文件了，你又在这个文件里编写了一个新函数，这个新函数要求新包括上一个头文件；可这个新头文件使用的名字与你现有的名字发生了冲突，你将不得不重新编写程序代码的某些部分。要想解决这个问题也并不困难，只要注意避免使用各种头文件所使用的一切名字就相安无事了。

我们把各种头文件里使用的各种名字总结在表A-1里：

表 A-1

头 文 件	使用的名字
ctype.h	所有名字都是以“is”或“to”开始的
dirent.h	所有名字都是以“d_”开始的
errno.h	所有名字都是以大写字母“E”开始，后面或者跟着一个数字，或者跟着其他的大写字母
fnctl.h	所有名字都是以“l_”、“F_”、“O_”或“S_”开始的
grp.h	所有名字都是以“gr_”开始的
limits.h	所有名字都是以“_MAX”结尾的
locale.h	所有名字都是以“LC_”开始的
pwd.h	所有名字都是以“pw_”开始的
signal.h	所有名字都是以“sa_”、“SIG”或“SA_”开始的
string.h	所有名字都是以“mem”、“str”或“wcs”开始的
sys/stat.h	所有名字都是以“st_”或“S_”开始的
sys/times.h	所有名字都是以“tms_”开始的
termios.h	所有名字都是以“c_”、“V”、“I”、“O”或“TC”开始的

A.1.3 资源极限

头文件limits.h里定义了许多最大值。一定要坚持使用头文件里定义的值，这里千万要不得你自己的发明创造。如果你确实需要定义一个自己的极限值，同时还希望代码能够被用在许多不同的系统上，就应该加上些“一次性”代码去检查当前系统上的极限值与你程序要求的是否一样或者更好。

当有人试图运行一个程序的时候，如果它不能运行，先说明哪些个极限值与程序要求的不兼容然后再“体面地退出”是比较好的做法；强行运行却又行为失常是不足取的。

A.2 硬件方面的可移植性

UNIX，以及如今的Linux，能够在范围很广的硬件上使用。你必须采取必要的步骤以保证自己不会失手写出只能限制使用在某个或某几个平台上的代码来。

C语言被广泛地认为是一种可移植性很强的语言。它适合完成各种截然不同的任务——从底

层的硬件访问操作一直到数据库等高层的应用程序。它还是一种效率极高的语言，它可以直接被编译器翻译为机器代码，以很小的运行开销高效率地完成预定的任务。

但事情并非总是如此。C语言有许多不能准确定义的成分——当我们在不同系统之间移植C语言源代码的时候，它们就容易引起问题。整数的长度大概是其中最经典的例子了。有许多年，UNIX程序员使用着能够容纳极大数值的32位整数——而同时代的MS-DOS程序员使用的还都是16位整数。UNIX环境下的代码认为整数可以容纳大于65 535的数值，把这样的代码移植到MS-DOS环境里去就会造成很严重的后果。

我们认为编写可移植代码有三种做法：

- 在一个单平台上编写代码，可移植性问题以后再说。这当然能够比较快地把最初的代码编写好，但在向其他平台移植的时候极有可能还需要大面积重写，而新的代码里不可避免地会出现许多新的bug。
- 在一个系统上编写它，注意避免使用不易移植的功能。如果确实需要使用肯定会在可移植性方面引起问题的功能，尽量把它分离出来，单独保存到一个文件或函数调用里。这个办法意味着在移植代码的时候肯定需要做一些工作，但工作量并不大，而且主要的问题都局限在我们已经知道需要针对可移植性进行处理的小代码段里。
- 在尽可能多的平台上切换着进行开发，避免使用任何不可移植的功能。这个办法很可能会使你软件的在第一版就需要花费非常非常多的时间才能有所结果，而且还有可能会运行得很慢。而当需要把它移植到一个它从未运行过的系统上时，还需要做一些工作。没有人（包括专业级高手在内）能够在每一个可能出现的问题真正出现之前就把它们都解决掉。

大家可能都已经猜到笔者倾向于第二种办法。我们把可移植性方面一些常见的陷阱分列在下面，并将就如何避免它们提出一些看法和建议。

A.2.1 数据长度

在C语言里，整数被定义为CPU处理器实际的字长度。在8086上这就将是16位的，而在68030上就将是32位的。int类型这种随机器而变化的长度是可移植性方面问题的一个常见原因。你会发现自己的整数在另外一台机器上不仅取值范围发生了变化，就是在对整数的某个位进行设置或测试时也容易引起问题。

我们来看看下面这个例子。假设我们想对整数变量“i”的第2位（从0算起）进行置位，下面这个表达式可以让我们达到目的：

```
i |= 0x04 ;
```

不管整数“i”的宽度是多少，这个表达式都能够对预定的位进行置位。

清除这个位可就不那么简单了。比较明显的做法是写出这样的代码：

```
i &= 0xfffb ;  /* Do not write this */
```

如果“i”是16位宽的整数，这个操作确实能清除制定的位。可如果“i”是32位宽的整数，问题就来了——我们不仅清除了第2位，连高端的16个位也都被清除了！正确的写法应该是这样的：

```
i &= ~0x04 ;
```

它告诉编译器（它知道整数的宽度应该是多少）要先使用“~”操作符构造出补集值等于0x04的二进制序列，然后在把这个序列施加到整数“i”上去。这个办法能够保证掩码里的位个数永远是正确的。

A.2.2 字节的存储顺序

不同的CPU处理器在把字节存放到内存里去时使用的字节存储顺序是不同的。少数处理器甚至还有改变内存里位存储顺序的指令。在这一方面，处理器可以被划分为两大类。

1. 字节的降序存储

这一大类里的处理器会把变量的最低位（least significant bits，简称LSBs）保存在最低位的地址里。VAX和80x86系列是使用这种内存访问方式的典型处理器。

2. 字节的升序存储

这一大类里的处理器会把变量的最高位（most significant bits，简称MSBs）保存在最低位的地址里。SPARC和68k系列处理器使用的就是这种内存访问方式。

> 极少数处理器允许你在运行时选择使用哪一种字节存储顺序，但它们不是我们这里关心的事情。

如果你是在这两大类处理器之间移植程序，就需要面对许多问题，特别是在向函数传递参数的时候。要尽量避免取函数参数地址的操作。请看下面这段代码：

```
char z;
func(z);

func(char zz) {
    char *p, tmp;
    p = &zz; /* Not a good idea */
    tmp = zz;
    p = &tmp; /* much safer */
```

更严重的问题出现在当你在这两种不同类型的处理器之间使用磁盘或跨网络来传递数据的时候。对此其他的全面讨论超出了本书的范围，但在必要的情况下，你应该考虑使用htonl、htons、ntohl和ntohs等函数，它们的作用是在主机的字节存储顺序和网络的字节存储顺序之间进行转换。在XDR（external data representation，外部数据表示方法）标准里可以查到一组适用范围更广的例程。

A.2.3 字符

字符的表示方法分带符号（取值范围是-127到+128）和不带符号（取值范围是0到256）两种，有许多程序能否正确工作完全要依赖于字符所使用的表示方法，而且这类程序的数量还相当惊人。不要妄下定论。如果确实需要字符带符号或不带符号，就一定要明确地把它们定义为“signed char”或“unsigned char”。短整数和整数也可能出现类似的问题，但一是非常少见，二是不象字符的影响这么大。

A.2.4 C语言的类型打包结构：union

编译器对union里类型域的安排有关标准里没有定义。假设你声明了一个下面这样的类型打

包结构：

```
union baz {
    short bar_short;
    char bar_char;
} my_baz;
```

如果你的代码把一个值保存在这个union结构的一个“部分”里，然后又通过另外一个“部分”把数据读出来，将会出现怎样的情况在有关标准里是没有定义的。使用一个union结构来改变变量的类型是不符合可移植性要求的。如下所示：

```
my_baz.bar_short = 7;
if (my_baz.bar_char == 7) { /* No ! */
```

A.2.5 结构的边界对齐

struct结构也会出现类似的问题，如下所示：

```
struct foo {
    short bar_short;
    char bar_char;
} my_foo;
```

foo的尺寸在不同的CPU处理器上是会变化的，具体情况取决于编译器在这个结构的各个成员之间到底都添加了多少字节、是怎样添加的。

struct结构方面的另外一个陷阱是试图用memcmp函数对它们进行比较。还是使用我们的foo结构，你可能会写出下面这样的代码：

```
struct foo foo1;
struct foo foo2;

/* some code that manipulates the structures */

if (memcmp(foo1, foo2, sizeof(struct foo))) { /* No !*/
```

千万不要这样做。你是在要求比较foo1结构里的每一个字节是否与foo2里面的相同，但因为存在调整结构成员间距的问题，结构里可能会存在一些与结构里的各个成员都没有必然联系的字节。如果你确实需要对两个结构进行比较，就必须明确的比较它们的各个成员。但使用普通的赋值语句或memcpy对结构进行拷贝复制是允许的，因为这是一个传递所有结构成员的操作。

A.2.6 指针长度

有许多老式的UNIX代码会假设表达式“sizeof(int) = sizeof(void *);”是成立的。

永远不要想当然地认为自己可以把指针看做是整数或者把整数看做是指针。如果你需要使用一个匿名指针，就一定要使用“void *”这样的表示方式。另外要特别注意qsort函数，下面是这个函数的语法定义：

```
void qsort(void *base, size_t num_el, size_t size_el,
        int (*compare)(const void *, const void *));
```

假设你想比较我们foo结构里的两个成员。在编写比较函数的时候千万不要使用分别指向两

个foo结构的两个指针。这个函数的语法定义要求使用的是“void *”。正确的做法是：编写一个接受两个void指针的比较函数，再通过局部变量投射这两个指针。如下所示：

```
int compare(void *struct_one_ptr, void *struct_two_ptr)
{
    struct foo *foo1_ptr;
    struct foo *foo2_ptr;

    foo1_ptr = (struct foo *)struct_one_ptr;
    foo2_ptr = (struct foo *)struct_two_ptr;
}
```

A.2.7 函数参数的求值

函数参数的求值顺序是有关标准没有定义的。如果你写出下面这样的代码：

```
func(a++, b=a+c );
```

就可能在可移植性方面遇到问题。这是因为有的编译器先对“a++”进行求值，有的则先对“b=a+c”求值。此外，对函数参数或在某些表达式的特定部分里使用宏定义的问题也要引起高度的重视，有些宏定义会产生副作用。下面就是一个这样的例子：

```
isdigit(*ptr++ ); /* No! */
```

最好不要使用这样的用法，因为isdigit经常被实现为一个宏。

A.3 向C++移植

在今天用ISO的C语言编写出来的代码可能在明天就需要增加一些新功能，而同时还需要把代码编译为C++程序。随着C++标准的发展，它与C语言之间的差异越来越大，把C语言代码移植到C++编译器的工作也相应地越来越难。我们已经向大家介绍了一些需要避免使用的C++关键字。除此之外，为了使今后向C++的移植工作相对简单一些，在C语言程序里还需要避免其他一些事情：

- 永远要给出函数的预定义。C++要求预定义；而C语言仅仅是鼓励这样做。
- 一个C语言函数可以有任意多个参数；而C++则不允许函数有参数。
- C++里的全局变量只能定义一次；C语言就没有这么严格。
- C语言里的全局性const有外部链接；而C++则使用文件链接。
- C语言里的struct变量在定义时需要在结构名之前加上关键字struct；C++里则不做要求。
- C++对枚举类型的类型检查比C语言里的更严格，而后者把它们都看做是整数。

A.4 编译器的使用

大多数现代的编译器都非常擅于针对可疑代码产生警告。用编译器选项把警告功能都打开。如果你使用的是gcc，就请加上“-Wall”选项——它的作用就是打开所有警告。如果你打算把自己的代码移植到另外一个不同的环境里去，还要再考虑加上gcc的“-pedantic”选项——它确实非常的“引经据典”（英文“pedantic”的意思就是“引经据典的”）！它能够根据今后程序移植的需要帮助你找出需要推敲的代码结构。与移植之后千辛万苦地查找和纠正有缺陷的程序相比，

在代码移植之前就改正掉编译器报告出来的所有警告要容易得多。

如果你是在编写成品代码，就需要考虑使用“-Wall”选项并且保证你的编译器没有产生任何警告。这看起来好象是一个过于理想化的目标，因为你几乎会不可避免地会收到一些警告而相应的程序结构却完全没有问题。

请记住两件事。一是编译器开发团队对程序设计语言精妙之处的了解比一般程序员要丰富得多；二是如果你从无警告编译开始并保持编译无警告，就不容易漏掉重要的警告信息。如果某个文件总是产生一大堆的警告信息，人们就会因习惯而成自然，一个新的重要的警告信息就很容易被忽略过去。

A.5 程序是给人看的

最后一个窍门是：记住代码是给人看的，而几乎总是看的次数比写的次数要多得多。许多代码的寿命要比它们始作者的估计要长。今日的“替补队员”（应急措施）可能就是明年的“最佳射手”（软件里的关键性应用程序）。

要注意使用正确的、有意义的变量名。把“i”和“j”用做简单的整数循环变量多少还可以接受，但其他的变量就应该有更有意义的名字。

给函数起名字时也要遵循同样的原则。这么说吧，如果你有一组与某个特定选项的状态有关的函数，它们就应该有一个整齐统一的命名方式——比如象get_option_baud()和get_option_typehead()这样。这将帮助阅读你程序的人们更快更容易地找到相关的函数。

附录B　自由软件基金会和GNU项目

下面这句话摘自GNU公告板："自由软件基金会（FSF）致力于消除在计算机软件的使用、复制、修改和再发行方面强加给人们的限制"。

"free"这个词的含义与金钱无关。花钱购买一张包含有GNU软件的CD盘决没有什么不对的地方。我们这里说的"free"指的是"自由"，即拥有源代码、学习它的操作原理、改写它以及与别人分享它等方面的自由。

自由软件基金会（Free Software Foundation，简称FSF）正在努力完成一个完整的软件系统。这个软件系统被称为GNU，意思是"GNU's Not UNIX"（GNU不是UNIX）。它的目的是向上兼容UNIX，同时又允许人们获得完整的源代码。这个项目至今仍未全部完成，但许多部分都已经很成熟了。大多数Linux发行版本里都包括有大量来自GNU项目的软件。

B.1　GNU项目

这个项目的主要工作包括：

- The Hurd。这是运行在名为Mach的内核上的一系列服务器进程，开发工作由CMU大学负责。The Hurd目前还没有完成，但那些服务器进程在Linux内核上都运行得好极了！
- The GNU C Library（GNU的C语言函数库）。大多数Linux发行版本都使用着它。它几乎已经完成了，但仍有工作要做。
- GNU EMACS。这大概是最具可配置性的程序设计用编辑器了，至今仍在编写中。它很难掌握，但为掌握它而付出的努力终将得到回报。说实在的，EMACS上的工作好象会永远继续下去！
- GCC。GNU的C语言编译器。它实际上由许多部分组成，各种不同的前端一直在增加中。目前进行的工作是Pascal语言和CHILL语言（http：//www.kvatro.no/telecom/chipsy/）前端，并且差不多算是完成了。在我们编写这本书的时候，这个编译器还没有通过ISO（国际标准化组织）的C语言认证——通过这个认证需要一笔数额相当巨大的资金，而FSF这个基金会的支出能力却总是捉襟见肘。不管怎么说，GCC与有关标准非常吻合，就是有区别，也相当细微。GCC还是一个C++编译器。
- Ghostscript。这是一个用于文档查看和输出方面的PostScript处理软件。
- 还有许许多多的工具性程序，比如名为bash的shell、bison和flex（YACC和LEX的替代软件）、make和无数其他的好东西。

除了极少数的例外，GNU版本的标准工具都要比原来的好。你可以通过捐赠或者购买他们软件（直接购买或者从向该基金会进行捐赠的供应商那里购买）来帮助自由软件基金会。

B.2 GNU公共许可证

这个许可证经常被人们称为“copyleft”或GPL。它处处体现着“自由”这个概念，与其他版权方案的限制性本质形成了鲜明的对比。

下面是GNU公共许可证的完整原文：

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such

program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".
Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.
You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 2 above, provided that you also meet all of these conditions:
   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.
   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.
   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 3) in object code or executable form under the terms of Sections 2 and 3 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 2 and 3 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 2 and 3 above on a medium customarily used for software interchange; or,
c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.
If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.
If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.
It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims;

this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.
This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A ARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the ``copyright'' line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright ©19yy <name of author>

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; either version 2 of the License, or (at your option) any later version.
This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

    Gnomovision version 69, Copyright © 19yy name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'
    This is free software, and you are welcome to redsitribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items-whatever suits your program.
You should also get your employer (if you work as a programmer) or your school, if any, to sign a ``copyright disclaimer'' for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License

附录C　因特网资源

因特网上有许多许多关于Linux、UNIX和其他相关软件的资源。我们这里给出的只是其中很少的一部分，你可以从它们开始起步。因特网是个每天都有新变化的资源，所以当你读到这本书的时候，有的站点可能已经不复存在了。别放弃！用你喜欢的搜索引擎再找处一些新的来，也许还有连我们都不知道的呢。

C.1　WWW站点

Web和Web上的站点个数总是在增长。请大家从这些出发点开始，沿着链接走下去，看看能找到些什么。

C.1.1　Linux专题

1. Linux新闻

Linux的最新新闻可以在下面这些站点上找到：

http://www.slashdot.org/
http://www.freshmeat.net/
http://linuxtoday.com/
http://www.linuxworld.com/
http://www.linuxgazette.com/

2. Linux内核资源

Alan Cox's Diary
http://www.linux.org.uk/diary

Linux Kernel Archives
http://www.kernel.org/

Linux-kernel mailing archive
http://www.uwsg.indiana.edu/hypermail/linux/kernel/index.html

KDB patches providing integrated debugging
http://reality.sgi.com/slurn_engr/

3. Linux发行商

Debian Project
http://www.debian.org/

Red Hat
http://www.redhat.com/

SuSE
http://www.suse.com/

Caldera Inc.
http://www.caldera.com/

4. 其他Linux资源

Linux Software Map
http://www.execpc.com/lsm

Linux Documentation Project
http://www.linuxdoc.org/

The Linux Home Page
http://www.linux.org/

Linuxberg
http://www.linuxberg.com/

Linux v2 Information HQ
http://www.linuxhq.com/

The Linux Threads Library
http://pauillac.inria.fr/~xleroy/linuxthreads/

LINUX.ORG.UK
http://www.uk.linux.org/

Linux 2.0 Penguins
http://www.isc.tamu.edu/~lewing/linux/

GNOME Project homepage
http://www.gnome.org/

Linux packages in RPM format
http://rufus.w3.org/

Pacific HiTech Home Page
http://www.pht.com/

The Linux Midi + Sound Main Page
http://www.bright.net/~dlphilp/linuxsound/

Trusted Information Systems: Firewall Toolkit on Linux
http://www.pauck.de/marco/misc/misc.html

Linux International: Promoting Linux
http://www.li.org/

InfoMagic Main Index Page
http://www.infomagic.com/

The Cathedral and the Bazaar
http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html/

StarOffice office productivity suite
http://www.sun.com/products/staroffice/

Applixware
http://www.applix.com/applixware/

C.1.2 UNIX操作系统和程序设计

The GNU Project Homepage
http://www.gnu.org/

Cygnus
http://www.cygnus.com/

The Electronic Frontier Foundation
http://www.eff.org/

UK UNIX User Group Home Page
http://www.ukuug.org/

Troll Tech - Qt Multi-platform C++ GUI Framework
http://www.troll.no

UnixWorld Online Magazine Home Page
http://www.wcmh.com:80/uworld/

Walnut Creek CDROM Electronic Catalog
http://www.cdrom.com/

Association of C & C++ Users WWW
http://www.accu.org/ - mail membership@accu.org for more info

Tcl WWW Info
http://www.sco.com/Technology/tcl/Tcl.html

The X Consortium Homepage
http://www.x.org/

Andrew S. Tanenbaum (provides a reference to Minix, a tiny UNIX-like OS)
http://www.cs.vu.nl/~ast

Tcl/Tk Project at the Scriptics Corporation
http://www.scriptics.com/

Tcl Contributed Software Archive
http://www.neosoft.com/tcl/contributed_sources/

Perlmongers homepage

http://www.perl.org/

V: a Freeware Portable C++ GUI Framework for Windows and X
http://www.objectcentral.com/

The Hungry Programmers (writing a free Motif clone)
http://www.hungry.com

Sun User Group
http://www.sunukug.org/
http://www.european.org/

Unix freeware sources
http://linux.robinson.cam.ac.uk/linapps/linapps.html

The NetBSD Project
http://www.netbsd.org

The Jargon File: a Collection of Slang Terms Used by Computer Hackers
http://www.fwi.uva.nl/~mes/jargon

The Official fvwm Homepage
http://www.fvwm.org/

SAMBA Web Pages: Allows SMB Access to UNIX Files and Printers
http://samba.anu.edu.au/samba/

The Xfree Home Page
http://www.XFree86.org/

Xforms C++ GUI library
ftp://ftp.cs.ruu.nl/pub/XFORMS/

C++ Information
http://www.maths.warwick.ac.uk/c++
http://www.ocsltd.com/c++

C.1.3 HTML语言和HTTP资料

1. 通用资料

The World Wide Web Consortium: Information on W3C
http://www.w3.org/

The World Wide Web Consortium: Protocol Information
http://www.w3.org/Protocols/Overview.html

Beginner's Guide to HTML
http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html

Guides to Writing Style for HTML Documents
http://union.ncsa.uiuc.edu/HyperNews/get/www/html/guides.html

Tools for Aspiring Web Authors
http://www.nas.nasa.gov/NAS/WebWeavers/

Location of the **weblint** Program
http://www.khoral.com/staff/neilb/weblint.html

Internet Engineering Task Force: HTML Standards
http://www.ics.uci.edu/pub/ietf/html/

Style Guide for Online Hypertext
http://www.w3.org/hypertext/WWW/Provider/Style/Overview.html

Tools for WWW Providers
http://www.w3.org/Tools/Overview.html

Introduction to HTML
http://www.cwru.edu/help/introHTML/toc.html

Webmaster Reference Library™
http://www.webreference.com/

2. CGI

The Common Gateway Interface
http://hoohoo.ncsa.uiuc.edu/cgi/

Matt's Script Archive
http://worldwidemart.com/scripts/

GIFs on the FLY
http://www.unimelb.edu.au/fly/fly.html

Graphics Library for GIF Creation
http://www.starship.python.net/~richard/gdmodule/

Beginner's Guide to HTML and CGI scripting
http://www.demon.co.uk/dita/new2html.html

Web Developer's Virtual Library: CGI
http://www.charm.net/Vlib/Providers/CGI.html

CGI security: Pitfalls to Avoid for CGI Programmers
http://www.cerf.net/customer_service/unix-web/secure_s.htm

`cgihtml`: Library of CGI and HTML Routines Written in C
http://cgi.resourceindex.com/Programs_and_Scripts/C_and_C++/

3. Web资料

The World Wide Web Consortium: Information on W℃
http://www.w3.org/

World Wide Web FAQ

http://www.boutell.com/openfaq/browsers/

4. HTTP服务器

World Wide Web Server Software - Lots of Servers Listed Here
http://www.w3.org.Servers.html

Apache HTTP Server Project
http://www.apache.org/

Status of the W3C httpd
http://www.w3.org/hypertext/WWW/Daemon/Status.html

The NCSA httpd Home Page
http://hoohoo.ncsa.uiuc.edu/

An Overview of the WN Server
http://hopf.math.nwu.edu/docs/overview.html

C.2 新闻组

Usenet永远是一个好出发点。

C.2.1 普通UNIX新闻组

alt.unix.wizards
comp.sources.unix
comp.std.unix
comp.unix <various>
gnu.announce

C.2.2 Linux专题新闻组

comp.os.linux <various>

如果你正在使用Linux并且能够上因特网，那么你至少应该订阅：

comp.os.linux.announce和comp.os.linux.answers

C.3 FTP下载站点

FTP下载站点有很多，其中几个比较主要的有：

ftp://sunsite.unc.edu
ftp://tsx-11.mit.edu
ftp://prep.ai.mit.edu/pub/gnu

记住，好地方总是很忙的，因特网上也不例外。请尽可能选择一个离你比较近的镜象站点。

第9章介绍的开发工具的URL地址

FTP主站点是：ftp://sunsite.unc.edu/pub/linux/devel/lang/c/

- Checker-0.7.2.lsm
- Checker-0.7.2.tgz
- Checker-libs-0.7.2.tgz
- ElectricFence-2.0.5.lsm
- ElectricFence-2.0.5.tar.gz
- cflow-2.0.tar.gz
- cflow.lsm
- cxref-1.0.lsm
- cxref-1.0.tgz
- cxref-1.0.txt
- lclint-2.1a.common-mini.tar.gz
- lclint-2.1a.common.tar.gz
- lclint-2.1a.linux-a.out.tar.gz
- lclint-2.1a.linux-elf.tar.gz
- lclint-2.1a.lsm
- lclint-2.1a.src.tar.gz

注意版本号可能会有所变化。在我们编写这本书的时候，lclint的最新版本是lclint-2.4b。Larch项目（lclint）的主站点是：http://larch-www.lcs.mit.edu:8001/larch/lclint。

附录D 参考书目

1. 有关标准

X/Open Single UNIX Specification
(Often referred to as Spec. 1170)
ISBN 1-85912-085-7
X/Open publications, +44 (0) 1993 708731
http://www.xopen.org/

IEEE Portable Operating System Interface for Computer Environments, POSIX
IEEE Std 1003.1-1988 C Language interface
ISBN 1-55937-003-3
IEEE, The Standards Office, 445 Hoes Lane, PO Box 1331,
Piscataway, NJ 08855-1331, USA
http://www.ieee.org

ISO C
ISO/IEC 9899:1990, Programming Languages, C
(Technically identical to ANSI standard ANS X3.159-1989, Programming Languages, C)

2. 其他文档和资源

Linux Manual Pages.
Linux HOWTO guides.
The many and various Linux CD-ROM distributions.

3. 推荐阅读书目

Advanced Programming in the UNIX Environment
W. Richard Stevens
Addison-Wesley
ISBN 0-201-56317-7

The UNIX Programming Environment
Brian W. Kernighan & Rob Pike
Prentice-Hall
ISBN 0-13-937681-X

Linux Application Development
Michael K. Johnson & Erik W. Troan
Addison-Wesley
ISBN 0-201-30821-5

Writing Apache Modules with Perl and C
Lincoln Stein & Doug MacEachern
O'Reilly & Associates

ISBN 1-56592-567-X

Tcl and the Tk Toolkit
John K. Ousterhout
Addison-Wesley
ISBN 0-201-63337-X

Advanced UNIX Programming
Marc J. Rochkind
Prentice-Hall
ISBN 0-13-011800-1

X Window Systems Programming and Applications with Xt
Douglas A. Young
Prentice-Hall
ISBN 0-13-972167-3

Instant UNIX
Evans, Matthew & Stones
Wrox Press
ISBN 1-874416-65-6

The Design of the UNIX Operating System
Maurice J. Bach
Prentice-Hall
ISBN 0-13-201757-1

The UNIX system
S.R. Bourne
Addison-Wesley
ISBN 0-201-13791-7

Programming the UNIX system
M. R. M. Dunsmuir & G. J. Davies
Macmillan
ISBN 0-333-37156-9

Programming Perl
Larry Wall & Randal L. Schwartz
O'Reilly & Associates
ISBN 0-937175-64-1

Learning Perl
Randal L. Schwartz & Tom Christiansen
O'Reilly & Associates
ISBN 1-56592-284-0

Practical Programming in Tcl and Tk
Brent B. Welch
Prentice-Hall
ISBN 0-13-182007-9

POSIX Programmer's Guide

Donald Lewine
O'Reilly & Associates
ISBN 0-937175-73-0

The Korn shell
Morris I. Bolasky & David G. Korn
Prentice-Hall
ISBN 0-13-516972-0

Principles of Concurrent and Distributed Programming
M. Ben-Ari
Prentice-Hall
ISBN 013711821X

HTML The Definitive Guide
Chuck Muciano & Bill Kennedy
O'Reilly & Associates
ISBN

Spinning the Web
Andrew Ford
International Thomson Publishing
ISBN 1-850-32141-8

Managing Projects with make
Andrew Oram & Steve Talbott
O'Reilly & Associates
ISBN 0-937175-90-0

Managing Internet Information Services
Cricket Liu, Jerry Peek, Russ Jones, Bryan Buus & Adrian Nye
O'Reilly & Associates
ISBN 1-56592-062-7

Linux in a Nutshell
Jessica Perry Hekman
O'Reilly & Associates
1-56592-167-4

Exploring Expect
Don Libes
O'Reilly & Associates
ISBN 1-56592-090-2.

UNIX Network Programming
W Richard Stevens
Prentice-Hall
ISBN 0-13-949876-1

Operating System Concepts
Abraham Silberschatz & Peter Galvin
Addison-Wesley
ISBN 0-20-159113-8

The Magic Garden Explained
Berny Goodheart & James Cox
Prentice-Hall
ISBN 0-13-098138-9

Linux Device Drivers
Alessandro Rubini
O'Reilly & Associates
ISBN 1-56-592292-1

Open Sources
Chris DiBona, Sam Ockman & Mark Stone (eds)
O'Reilly and Associates
ISBN 1 56592 582 3

4. 三本脱机阅读书目

Gödel, Escher, Bach: An Eternal Golden Braid
Douglas R. Hofstadter
Penguin Books
ISBN 0140055797

The Illuminatus! Trilogy
Robert Shea & Robert Anton Wilson
Dell
ISBN 0440539811

The Wasp Factory
Iain Banks
Abacus Fiction
ISBN 0-349-10177-9

The Magic Garden Explained
Berny Goodheart & James Cox
Prentice-Hall
ISBN 0-13-098138-9

Linux Device Drivers
Alessandro Rubini
O'Reilly & Associates
ISBN 1-56-592292-1

Open Sources
Chris DiBona, Sam Ockman & Mark Stone (eds)
O'Reilly and Associates
ISBN 1 56592 582 3

4. 三本脱机阅读书目

Gödel, Escher, Bach: An Eternal Golden Braid
Douglas R. Hofstadter
Penguin Books
ISBN 0140055797

The Illuminatus! Trilogy
Robert Shea & Robert Anton Wilson
Dell
ISBN 0440539811

The Wasp Factory
Iain Banks
Abacus Fiction
ISBN 0-349-10177-9

The Magic Garden Explained
Berny Goodheart & James Cox
Prentice-Hall
ISBN 0-13-098138-9

Linux Device Drivers
Alessandro Rubini
O'Reilly & Associates
ISBN 1-56-592292-1

Open Sources
Chris DiBona, Sam Ockman & Mark Stone (eds)
O'Reilly and Associates
ISBN 1 56592 582 3

4. 三本脱机阅读书目

Gödel, Escher, Bach: An Eternal Golden Braid
Douglas R. Hofstadter
Penguin Books
ISBN 0140055797

The Illuminatus! Trilogy
Robert Shea & Robert Anton Wilson
Dell
ISBN 0440539811

The Wasp Factory
Iain Banks
Abacus Fiction
ISBN 0-349-10177-9

The Magic Garden Explained
Berny Goodheart & James Cox
Prentice-Hall
ISBN 0-13-098138-9

Linux Device Drivers
Alessandro Rubini
O'Reilly & Associates
ISBN 1-56-592292-1

Open Sources
Chris DiBona, Sam Ockman & Mark Stone (eds)
O'Reilly and Associates
ISBN 1 56592 582 3

4. 三本脱机阅读书目

Gödel, Escher, Bach: An Eternal Golden Braid
Douglas R. Hofstadter
Penguin Books
ISBN 0140055797

The Illuminatus! Trilogy
Robert Shea & Robert Anton Wilson
Dell
ISBN 0440539811

The Wasp Factory
Iain Banks
Abacus Fiction
ISBN 0-349-10177-9

The Magic Garden Explained
Berny Goodheart & James Cox
Prentice-Hall
ISBN 0-13-098138-9

Linux Device Drivers
Alessandro Rubini
O'Reilly & Associates
ISBN 1-56-592292-1

Open Sources
Chris DiBona, Sam Ockman & Mark Stone (eds)
O'Reilly and Associates
ISBN 1 56592 582 3

4. 三本脱机阅读书目

Gödel, Escher, Bach: An Eternal Golden Braid
Douglas R. Hofstadter
Penguin Books
ISBN 0140055797

The Illuminatus! Trilogy
Robert Shea & Robert Anton Wilson
Dell
ISBN 0440539811

The Wasp Factory
Iain Banks
Abacus Fiction
ISBN 0-349-10177-9

The Magic Garden Explained
Berny Goodheart & James Cox
Prentice-Hall
ISBN 0-13-098138-9

Linux Device Drivers
Alessandro Rubini
O'Reilly & Associates
ISBN 1-56-592292-1

Open Sources
Chris DiBona, Sam Ockman & Mark Stone (eds)
O'Reilly and Associates
ISBN 1 56592 582 3

4. 三本脱机阅读书目

Gödel, Escher, Bach: An Eternal Golden Braid
Douglas R. Hofstadter
Penguin Books
ISBN 0140055797

The Illuminatus! Trilogy
Robert Shea & Robert Anton Wilson
Dell
ISBN 0440539811

The Wasp Factory
Iain Banks
Abacus Fiction
ISBN 0-349-10177-9