

Linux命令行 与shell脚本 编程大全 (第2版)

Linux Command Line and
Shell Scripting Bible **Second Edition**

- 亚马逊书店五星推荐
- 轻松全面掌握命令行和shell
- 黑客进阶必读

[美] Richard Blum
Christine Bresnahan 著

武海峰 译

人民邮电出版社
POSTS & TELECOM PRESS

www.linuxidc.com

亚马逊读者评论

“Blum的知识和经验非常有用。这既是一本优秀的初学者入门指南，又会是你的Linux书库中非常不错的参考书。”

“本书讲解透彻、代码示例丰富，并详细说明了不同shell之间的差异。花点时间学会编写shell脚本，你将从中长期受益。”

“如果你想从整体上了解Linux并开始学写脚本，那么就从本书开始吧。”

掌握Linux命令和shell脚本编程，尽在本书中。

本书是关于Linux命令行和shell命令的全面参考资料，秉承“大全”系列书籍的一贯优良品质，涵盖详尽的动手教程和实际应用中的实用信息，并提供相关参考信息和背景资料。书中内容共分为四部分27章，引领读者从Linux命令行基础入手，直到能写出自己的shell。

本书主要内容包括：

- ◆ 在命令行上工作并学习基本的shell命令；
- ◆ 编写shell脚本来实现日常工作和报告的自动化；
- ◆ 控制如何以及何时在系统上运行shell脚本；
- ◆ 学习shell脚本中操作数据的高级方法；
- ◆ 修改脚本适应图形化桌面和其他Linux shell；
- ◆ 从网站提取数据并在系统间发送数据；
- ◆ 创建有专业水准的shell脚本，适应现实环境的挑战。



www.wiley.com

Copies of this book sold without a Wiley sticker
on the cover are unauthorized and illegal.

图灵社区：www.ituring.com.cn

新浪微博：[@图灵教育](#) [@图灵社区](#)

反馈 投稿 推荐信箱：contact@turingbook.com

热线：(010)51095186转604

分类建议

计算机 / 操作系统 / Linux

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-28889-9



ISBN 978-7-115-28889-9

定价：99.00元

Linux命令行 与shell脚本 编程大全

(第2版)

Linux Command Line and
Shell Scripting Bible Second Edition

[美] Richard Blum
Christine Bresnahan 著

武海峰 译

人民邮电出版社
北京

www.linuxidc.com

图书在版编目（C I P）数据

Linux命令行与shell脚本编程大全：第2版 / (美)布卢姆 (Blum, R.) , (美)布雷斯纳汉 (Bresnahan, C.) 著 ; 武海峰译。— 北京 : 人民邮电出版社, 2012. 9
(图灵程序设计丛书)
书名原文: Linux Command Line and Shell Scripting Bible, Second Edition
ISBN 978-7-115-28889-9

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

中国版本图书馆CIP数据核字(2012)第173331号

内 容 提 要

本书是一本关于 Linux 命令行与 shell 脚本编程的全面教程。全书分为四部分：第一部分介绍 Linux shell 命令行；第二部分介绍 shell 脚本编程基础；第三部分深入探讨 shell 脚本编程的高级内容；第四部分介绍如何在现实环境中使用 shell 脚本。本书不仅涵盖了详尽的动手教程和现实世界中的实用信息，还提供了与所学内容相关的参考信息和背景资料。

本书内容全面，语言简练，示例丰富，适合于 Linux 系统管理员及 Linux 爱好者阅读参考。

图灵程序设计丛书 Linux命令行与shell脚本编程大全（第2版）

◆ 著 [美] Richard Blum Christine Bresnahan
译 武海峰
责任编辑 朱巍
执行编辑 罗词亮
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷
◆ 开本: 800×1000 1/16
印张: 39.75
字数: 939千字 2012年9月第1版
印数: 1~4 000册 2012年9月北京第1次印刷
著作权合同登记号 图字: 01-2011-3262号
ISBN 978-7-115-28889-9

定价: 99.00元

读者服务热线: (010)51095186转604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

译者序

欢迎来到命令行和shell脚本编程的世界！看到这篇序言，说明你已经翻开这本超过600页的书，愿意花时间去了解命令行和shell脚本编程的细节。在Linux世界中，这种精神非常重要，我们将这种不满足于现状、愿意潜心了解其中细节而作出进一步改进的精神称为“黑客精神”。

正是这种黑客精神激励着成百上千人一起协作、不断改进着Linux内核^①，使其支持更多的硬件、具备更多有趣的功能。越来越多的硬件厂商成立了Linux开发团队，向Linux内核贡献代码，使其能够运行在他们的硬件平台上或顺利地使用他们的硬件。也正是这种黑客精神和GNU的开放性，才使得Linux在服务器操作系统领域和移动设备操作系统领域获得了迅捷而长足的发展。

今天，移动互联网应用的流行使得每周都有成百上千的应用部署在云平台上，而背后支撑这些云的正是Linux。这些应用大多分成两部分：作为移动应用的一部分会部署在用户的持设备上，实现用户交互、数据的采集和传送；作为Web应用的另一部分部署在远端Linux服务器上，存储数据并进行一些定期的数据处理。

这时，要进行Linux服务器上大量的数据处理和管理工作，以及Web应用的部署和监测，你就需要命令行和shell脚本的帮助。在Linux系统中，大部分的系统配置和管理都是通过shell脚本完成，而Linux对shell脚本的良好支持又使其成为系统配置和管理自动化的首选。加之几乎所有Linux主机都支持通过SSH远程访问来进行系统配置，大多数时候你都是在命令行下完成这些配置和管理工作。

当然，伴随着便利性的还有巨大的破坏性。稍不留神，你可能就会将整个根目录全部干掉，或者错误处理重要的配置文件。这时，了解命令行和shell脚本编程细节、遵循Linux使用规范就显得格外重要了。

本书面向系统管理员、配置管理人员、系统开发人员，以及所有想有效使用Linux系统的黑客。除了Linux外，本书内容还适用于提供了命令行界面和shell脚本编程支持的其他类Unix系统（我们称之为“*nix”系统），其中包括BSD和Mac OS X。

读过本书后，如果想继续了解shell脚本编程，你可以接着看*Wicked Cool Shell Scripts*。这本书提供了大量的例子和解释供你学习。如果有意扩展Linux系统管理方面的知识，可以参考IBM的*Linux Performance and Tuning Guidelines*和*Linux Administration Handbook (Second Edition)*。如果对shell如何工作感到好奇，Use the source, Luke! 最简单的shell——yash^②只有一百来行。

① 参见Linux基金会提供的内核开发报告：<http://go.linuxfoundation.org/who-writes-linux-2012>。

② 参考页面：<http://samiam.org/software/yash.html>。

这里要特别感谢图灵公司的傅志红和罗词亮两位老师。他们的细致工作使得这本译作中的诸多术语都得以一一推敲，他们的耐心等待使我得以在旅行中完成本书的翻译工作。

尽管我在翻译过程中小心翼翼，尽力排查原书中的错误、尽量避免中文表达上的问题，但囿于经验和疏忽，译作中难免有疏漏和问题。还请读者朋友给予反馈，我们会在图灵社区维护一份勘误表（<http://www.ituring.com.cn/book/980>），并尽量为您在阅读本书中遇到的问题提供力所能及的帮助。



引言

欢迎打开《Linux命令行和shell脚本编程大全（第2版）》。和所有“大全”系列书籍一样，本书内容涵盖了详尽的动手教程和实际应用中的实用信息，还提供了与所学内容相关的参考信息和背景资料。本书是关于Linux命令行和shell命令的相当全面的资源。读完本书，你将可以轻松写出自己的shell脚本来自动化处理Linux系统上的任何任务。

读者对象

如果你是Linux环境下的系统管理员，那么学会编写shell脚本将让你受益匪浅。本书并未细述安装Linux系统的每个步骤，但只要系统已安装好Linux并能运行起来，你就可以开始考虑如何将一些日常的系统管理任务实现自动化。这时shell脚本编程就能发挥作用了，而这也正是书的作用所在。本书将演示如何使用shell脚本来自动处理系统管理任务，包括从监测系统统计数据和数据文件到为你的老板生成报告。

如果你是Linux爱好者，那你也能从本书中受益。现今，用户很容易在诸多部件(widget)堆积而成的图形环境中迷失。大多数桌面Linux发行版都尽量向一般用户隐藏系统内核。然而有时你确实需要知道内部发生了什么。本书将告诉你如何启动Linux命令行以及启动后下一步做什么。通常，如果是执行一些简单任务(比如文件管理)，那么在命令行下操作要比在花哨的图形界面上方便得多。在命令行下有大量的命令可供使用，本书将会展示如何使用它们。

本书结构

本书将会引领你从Linux命令行基础一直学到写出自己的shell脚本。全书分成四部分，每部分都基于前面的内容。

第一部分假定你已经有个能运行的Linux系统，或者正在设法获取Linux系统。第1章“初识Linux shell”，描述了构成整个Linux系统的各个部分，并且说明了shell是如何融入Linux的。在介绍了Linux系统的基础知识之后，这部分继续探讨了：

- 使用终端模拟包来访问shell（第2章）；
- 介绍基本的shell命令（第3章）；
- 使用更高级的shell命令来窥探系统信息（第4章）；

- 使用shell变量来操作数据（第5章）；
- 理解Linux文件系统和安全（第6章）；
- 在命令行上操作Linux文件系统（第7章）；
- 在命令行上安装和更新软件（第8章）；
- 使用Linux编辑器开始编写shell脚本（第9章）。

第二部分将从编写shell脚本开始。在你阅读各章内容时，你会：

- 学习如何创建和运行shell脚本（第10章）；
- 改变shell脚本中程序的流程（第11章）；
- 迭代代码片段（第12章）；
- 在脚本中处理用户输入的数据（第13章）；
- 了解在脚本中存储和显示数据的不同方法（第14章）；
- 控制脚本如何以及何时在系统中运行（第15章）。

第三部分深入探讨shell脚本编程的更高级领域，包括：

- 在所有脚本中创建自己的函数（第16章）；
- 利用Linux图形化桌面来和脚本用户交互（第17章）；
- 使用高级Linux命令过滤和解析数据文件（第18章）；
- 使用正则表达式来定义数据（第19章）；
- 学习在脚本中操作数据的高级方法（第20章）；
- 从原始数据生成报告（第21章）；
- 修改shell脚本，使其能在其他Linux shell中运行（第22章）。

本书的最后一部分——第四部分演示了如何在现实环境中使用shell脚本。在这部分，你将：

- 学习如何在shell脚本中使用流行的开源数据库（第23章）；
- 学习如何从网站上提取数据并在系统间发送数据（第24章）；
- 使用E-mail向外部用户发送通知和报告（第25章）；
- 编写shell脚本来自动化你的日常系统管理工作（第26章）；
- 利用你在本书中学到的所有功能来创建专业水平的shell脚本（第27章）。

约定和排版

为帮助读者更好地理解本书内容，全书作了很多不同的组织和排版上的处理。

说明和注意

当有重要的内容想让读者注意时，这部分信息会出现在注意中。

注意 这部分信息很重要，所以放在单独的段落里并采用特殊排版。注意提供了要特别注意的信息，不管是小小的不便还是对数据和系统潜在的危害。

对于与正文有关的其他有意思的内容，我们会用说明给出。

说明 说明提供了有用的补充或辅助信息，但有些偏离当前讲述的主题。

最低需求

本书并不局限于某个特定Linux发行版，你可以使用任何可用的Linux系统来跟着书中内容学习。书中大部分内容都采用了bash shell。在大多数Linux系统中，bash shell是默认shell。

下一步做什么

看完了本书，你就已经可以在日常工作中使用Linux命令了。在不断变化的Linux世界，最好能不断了解Linux的最新发展。通常Linux发行版会发生一些变化，增加新的功能同时移除过时的功能。经常关注Linux方面的资讯，能保证你的Linux知识也在不断更新。找一个不错的Linux论坛，关注一下Linux世界的最新动态。有很多流行的Linux新闻站点，比如Slashdot和Distrowatch，都能提供哪怕是几分钟前发生的Linux新进展。



致 谢

首先，所有的荣誉和赞美都献给上帝。是他通过他的儿子耶稣，让这一切成为可能，并赐予我们永生。

非常感谢John Wiley & Sons出版团队的诸位为本书作出的突出贡献。感谢组稿编辑Mary James为我们提供写作本书的机会。感谢策划编辑Brian Herrmann保证本书的写作顺利进行，并将内容更好地呈现给读者。感谢Brian的努力和勤勉。本书的技术编辑Jack Cox为保证本书的内容正确作出了卓越贡献，并对本书内容提出了若干改进建议。感谢本书的文字编辑Nancy Rapoport，她的耐心和努力使得本书的可读性更好。还要感谢Waterside Productions公司的Carole McClendon为我们安排本书的写作事务，并在我们的写作道路上给予了我们很大的帮助。

在此，Christine还想感谢她的先生Timothy，感谢他的鼓励、耐心和倾听，即使在他并不理解她说的是什么时。

目 录

第一部分 Linux 命令行	
第 1 章 初识 Linux shell2	
1.1 什么是 Linux	2
1.1.1 深入探究 Linux 内核.....3	
1.1.2 GNU 工具链.....10	
1.1.3 Linux 桌面环境.....11	
1.2 Linux 发行版.....16	
1.2.1 核心 Linux 发行版.....16	
1.2.2 专业 Linux 发行版.....17	
1.2.3 Linux LiveCD.....17	
1.3 小结	19
第 2 章 走进 shell20	
2.1 终端模拟.....20	
2.1.1 图形功能.....21	
2.1.2 键盘.....24	
2.2 terminfo 数据库.....25	
2.3 Linux 控制台.....28	
2.4 xterm 终端.....29	
2.4.1 命令行参数.....30	
2.4.2 xterm 主菜单.....31	
2.4.3 VT 选项菜单.....32	
2.4.4 VT 字体菜单.....34	
2.5 Konsole 终端.....36	
2.5.1 命令行参数	36
2.5.2 标签式窗口会话.....37	
2.5.3 配置文件	38
2.5.4 菜单栏	39
2.6 GNOME Terminal.....43	
2.6.1 命令行参数	43
2.6.2 标签.....43	
2.6.3 菜单栏.....44	
2.7 小结	47
第 3 章 基本的 bash shell 命令48	
3.1 启动 shell.....48	
3.2 shell 提示符	49
3.3 bash 手册	51
3.4 浏览文件系统	52
3.4.1 Linux 文件系统.....52	
3.4.2 遍历目录	54
3.5 文件和目录列表	56
3.5.1 基本列表功能	56
3.5.2 修改输出信息	57
3.5.3 完整的参数列表	58
3.5.4 过滤输出列表	60
3.6 处理文件	61
3.6.1 创建文件	61
3.6.2 复制文件	61
3.6.3 链接文件	63
3.6.4 重命名文件	65
3.6.5 删除文件	65
3.7 处理目录	67
3.7.1 创建目录	67
3.7.2 删除目录	67
3.8 查看文件内容	68
3.8.1 查看文件统计信息	68
3.8.2 查看文件类型	69
3.8.3 查看整个文件	69
3.8.4 查看部分文件	72
3.9 小结	73

第 4 章 更多的 bash shell 命令	75	6.1.5 修改用户	128
4.1 监测程序	75	6.2 使用 Linux 组	130
4.1.1 探查进程	75	6.2.1 /etc/group 文件	131
4.1.2 实时监测进程	82	6.2.2 创建新组	131
4.1.3 结束进程	84	6.2.3 修改组	132
4.2 监测磁盘空间	85	6.3 理解文件权限	133
4.2.1 挂载存储媒体	86	6.3.1 使用文件权限符	133
4.2.2 使用 df 命令	89	6.3.2 默认文件权限	134
4.2.3 使用 du 命令	89	6.4 改变安全性设置	136
4.3 处理数据文件	90	6.4.1 改变权限	136
4.3.1 排序数据	91	6.4.2 改变所属关系	137
4.3.2 搜索数据	94	6.5 共享文件	138
4.3.3 压缩数据	96	6.6 小结	139
4.3.4 归档数据	99		
4.4 小结	100		
第 5 章 使用 Linux 环境变量	101		
5.1 什么是环境变量	101	第 7 章 管理文件系统	141
5.1.1 全局环境变量	102	7.1 探索 Linux 文件系统	141
5.1.2 局部环境变量	103	7.1.1 基本的 Linux 文件系统	141
5.2 设置环境变量	106	7.1.2 日志文件系统	142
5.2.1 设置局部环境变量	106	7.1.3 扩展的 Linux 日志文件系统	143
5.2.2 设置全局环境变量	107	7.2 操作文件系统	145
5.3 删除环境变量	107	7.2.1 创建分区	145
5.4 默认 shell 环境变量	108	7.2.2 创建文件系统	147
5.5 设置 PATH 环境变量	111	7.2.3 如果出错了	149
5.6 定位系统环境变量	112	7.3 逻辑卷管理器	150
5.6.1 登录 shell	112	7.3.1 逻辑卷管理布局	150
5.6.2 交互式 shell	116	7.3.2 Linux 中的 LVM	151
5.6.3 非交互式 shell	118	7.3.3 使用 Linux LVM	153
5.7 可变数组	118	7.4 小结	157
5.8 使用命令别名	119		
5.9 小结	120		
第 6 章 理解 Linux 文件权限	122	第 8 章 安装软件程序	158
6.1 Linux 的安全性	122	8.1 包管理基础	158
6.1.1 /etc/passwd 文件	122	8.2 基于 Debian 的系统	159
6.1.2 /etc/shadow 文件	124	8.2.1 用 aptitude 管理软件包	159
6.1.3 添加新用户	125	8.2.2 用 aptitude 安装软件包	161
6.1.4 删除用户	127	8.2.3 用 aptitude 更新软件	163
		8.2.4 用 aptitude 卸载软件	164
		8.2.5 aptitude 库	164
		8.3 基于 Red Hat 的系统	166
		8.3.1 列出已安装包	166
		8.3.2 用 yum 安装软件	167

8.3.3 用 yum 更新软件	168	10.7.1 expr 命令	212
8.3.4 用 yum 卸载软件	169	10.7.2 使用方括号	214
8.3.5 处理损坏的包依赖关系	169	10.7.3 浮点解决方案	215
8.3.6 yum 软件库	171	10.8 退出脚本	218
8.4 从源码安装	172	10.8.1 查看退出状态码	218
8.5 小结	174	10.8.2 exit 命令	219
第 9 章 使用编辑器	176	10.9 小结	221
9.1 Vim 编辑器	176	第 11 章 使用结构化命令	222
9.1.1 Vim 基础	176	11.1 使用 if-then 语句	222
9.1.2 编辑数据	178	11.2 if-then-else 语句	224
9.1.3 复制和粘贴	179	11.3 嵌套 if	225
9.1.4 查找和替换	180	11.4 test 命令	226
9.2 Emacs 编辑器	180	11.4.1 数值比较	227
9.2.1 在控制台上使用 Emacs	180	11.4.2 字符串比较	228
9.2.2 在 X Window 中使用 Emacs	185	11.4.3 文件比较	232
9.3 KDE 系编辑器	186	11.5 复合条件测试	239
9.3.1 KWrite 编辑器	186	11.6 if-then 的高级特性	240
9.3.2 Kate 编辑器	190	11.6.1 使用双尖括号	240
9.4 GNOME 编辑器	192	11.6.2 使用双方括号	241
9.4.1 启动 gedit	192	11.7 case 命令	242
9.4.2 基本的 gedit 功能	193	11.8 小结	243
9.4.3 设定偏好设置	194	第 12 章 更多的结构化命令	245
9.5 小结	196	12.1 for 命令	245
第二部分 shell 脚本编程基础		12.1.1 读取列表中的值	246
第 10 章 构建基本脚本	200	12.1.2 读取列表中的复杂值	247
10.1 使用多个命令	200	12.1.3 从变量读取列表	248
10.2 创建 shell 脚本文件	201	12.1.4 从命令读取值	249
10.3 显示消息	202	12.1.5 更改字段分隔符	250
10.4 使用变量	203	12.1.6 用通配符读取目录	251
10.4.1 环境变量	204	12.2 C 语言风格的 for 命令	253
10.4.2 用户变量	205	12.2.1 C 语言的 for 命令	253
10.4.3 反引号	206	12.2.2 使用多个变量	255
10.5 重定向输入和输出	207	12.3 while 命令	255
10.5.1 输出重定向	208	12.3.1 while 的基本格式	255
10.5.2 输入重定向	208	12.3.2 使用多个测试命令	256
10.6 管道	209	12.4 until 命令	258
10.7 执行数学运算	212	12.5 嵌套循环	259
		12.6 循环处理文件数据	261

12.7 控制循环.....	262	14.4.5 关闭文件描述符.....	303
12.7.1 break 命令.....	262	14.5 列出打开的文件描述符.....	304
12.7.2 continue 命令.....	265	14.6 阻止命令输出.....	305
12.8 处理循环的输出.....	267	14.7 创建临时文件.....	306
12.9 小结.....	269	14.7.1 创建本地临时文件.....	306
第 13 章 处理用户输入.....	270	14.7.2 在/tmp 目录创建临时文件.....	308
13.1 命令行参数.....	270	14.7.3 创建临时目录.....	308
13.1.1 读取参数.....	270	14.8 记录消息.....	309
13.1.2 读取程序名.....	272	14.9 小结.....	310
13.1.3 测试参数.....	274		
13.2 特殊参数变量.....	274	第 15 章 控制脚本.....	312
13.2.1 参数计数.....	274	15.1 处理信号.....	312
13.2.2 抓取所有的数据.....	276	15.1.1 重设 Linux 信号.....	312
13.3 移动变量.....	277	15.1.2 产生信号.....	313
13.4 处理选项.....	278	15.1.3 捕捉信号.....	314
13.4.1 查找选项.....	279	15.1.4 捕捉脚本的退出.....	315
13.4.2 使用 getopt 命令.....	282	15.1.5 移除捕捉.....	316
13.4.3 使用更高级的 getopts.....	284	15.2 以后台模式运行脚本.....	317
13.5 将选项标准化.....	286	15.2.1 后台运行脚本.....	317
13.6 获得用户输入.....	287	15.2.2 运行多个后台作业.....	318
13.6.1 基本的读取.....	287	15.2.3 退出终端.....	319
13.6.2 超时.....	289	15.3 在非控制台下运行脚本.....	319
13.6.3 隐藏方式读取.....	290	15.4 作业控制.....	320
13.6.4 从文件中读取.....	290	15.4.1 查看作业.....	320
13.7 小结.....	291	15.4.2 重启停止的作业.....	322
第 14 章 呈现数据.....	293	15.5 调整谦让度.....	323
14.1 理解输入和输出.....	293	15.5.1 nice 命令.....	323
14.1.1 标准文件描述符.....	293	15.5.2 renice 命令.....	324
14.1.2 重定向错误.....	295	15.6 定时运行作业.....	324
14.2 在脚本中重定向输出.....	297	15.6.1 用 at 命令来计划执行作业.....	325
14.2.1 临时重定向.....	297	15.6.2 计划定期执行脚本.....	328
14.2.2 永久重定向.....	298	15.7 启动时运行.....	330
14.3 在脚本中重定向输入.....	299	15.7.1 开机时运行脚本.....	330
14.4 创建自己的重定向.....	299	15.7.2 在新 shell 中启动.....	332
14.4.1 创建输出文件描述符.....	300	15.8 小结.....	333
14.4.2 重定向文件描述符.....	300		
14.4.3 创建输入文件描述符.....	301		
14.4.4 创建读写文件描述符.....	302		

第三部分 高级 shell 脚本编程

第 16 章 创建函数.....	336
16.1 基本的脚本函数.....	336

16.1.1 创建函数	337	18.2.1 更多的替换选项	385
16.1.2 使用函数	337	18.2.2 使用地址	387
16.2 返回值	339	18.2.3 删除行	389
16.2.1 默认退出状态码	339	18.2.4 插入和附加文本	391
16.2.2 使用 return 命令	340	18.2.5 修改行	392
16.2.3 使用函数输出	341	18.2.6 转换命令	393
16.3 在函数中使用变量	342	18.2.7 回顾打印	394
16.3.1 向函数传递参数	342	18.2.8 用 sed 和文件一起工作	396
16.3.2 在函数中处理变量	344	18.3 小结	398
16.4 数组变量和函数	346	第 19 章 正则表达式	399
16.4.1 向函数传递数组参数	346	19.1 什么是正则表达式	399
16.4.2 从函数返回数组	348	19.1.1 定义	399
16.5 函数递归	349	19.1.2 正则表达式的类型	400
16.6 创建库	350	19.2 定义 BRE 模式	401
16.7 在命令行上使用函数	351	19.2.1 纯文本	401
16.7.1 在命令行上创建函数	352	19.2.2 特殊字符	402
16.7.2 在.bashrc 文件中定义函数	352	19.2.3 锚字符	403
16.8 小结	354	19.2.4 点字符	405
第 17 章 图形化桌面上的脚本编程	355	19.2.5 字符组	405
17.1 创建文本菜单	355	19.2.6 排除字符组	407
17.1.1 创建菜单布局	356	19.2.7 使用区间	408
17.1.2 创建菜单函数	356	19.2.8 特殊字符组	409
17.1.3 添加菜单逻辑	357	19.2.9 星号	409
17.1.4 整合 shell 脚本菜单	358	19.3 扩展正则表达式	411
17.1.5 使用 select 命令	359	19.3.1 问号	411
17.2 使用窗口	360	19.3.2 加号	412
17.2.1 dialog 包	361	19.3.3 使用花括号	412
17.2.2 dialog 选项	366	19.3.4 管道符号	413
17.2.3 在脚本中使用 dialog 命令	368	19.3.5 聚合表达式	414
17.3 使用图形	369	19.4 实用中的正则表达式	414
17.3.1 KDE 环境	369	19.4.1 目录文件计数	415
17.3.2 GNOME 环境	372	19.4.2 验证电话号码	416
17.4 小结	376	19.4.3 解析邮件地址	417
第 18 章 初识 sed 和 gawk	377	19.5 小结	419
18.1 文本处理	377	第 20 章 sed 进阶	420
18.1.1 sed 编辑器	377	20.1 多行命令	420
18.1.2 gawk 程序	380	20.1.1 next 命令	421
18.2 sed 编辑器基础	385	20.1.2 多行删除命令	424

20.1.3 多行打印命令	424	21.6 内建函数	460
20.2 保持空间	425	21.6.1 数学函数	460
20.3 排除命令	426	21.6.2 字符串函数	461
20.4 改变流	428	21.6.3 时间函数	463
20.4.1 跳转	429	21.7 自定义函数	463
20.4.2 测试	430	21.7.1 定义函数	463
20.5 模式替代	431	21.7.2 使用自定义函数	464
20.5.1 and 符号	431	21.7.3 创建函数库	464
20.5.2 替换单独的单词	432	21.8 小结	465
20.6 在脚本中使用 sed	433		
20.6.1 使用包装脚本	433		
20.6.2 重定向 sed 的输出	434		
20.7 创建 sed 实用工具	434		
20.7.1 加倍行间距	434		
20.7.2 对可能含有空白行的文件			
加倍行间距	435		
20.7.3 给文件中的行编号	436		
20.7.4 打印末尾行	437		
20.7.5 删除行	437		
20.7.6 删除 HTML 标签	439		
20.8 小结	441		
第 21 章 gawk 进阶	442		
21.1 使用变量	442		
21.1.1 内建变量	442		
21.1.2 自定义变量	447		
21.2 处理数组	449		
21.2.1 定义数组变量	449		
21.2.2 遍历数组变量	450		
21.2.3 删除数组变量	451		
21.3 使用模式	451		
21.3.1 正则表达式	451		
21.3.2 匹配操作符	452		
21.3.3 数学表达式	452		
21.4 结构化命令	453		
21.4.1 if 语句	453		
21.4.2 while 语句	455		
21.4.3 do-while 语句	456		
21.4.4 for 语句	457		
21.5 格式化打印	457		
		第 22 章 使用其他 shell	467
		22.1 什么是 dash shell	467
		22.2 dash shell 的特性	468
		22.2.1 dash 命令行参数	468
		22.2.2 dash 环境变量	469
		22.2.3 dash 内建命令	471
		22.3 dash 脚本编程	472
		22.3.1 创建 dash 脚本	473
		22.3.2 不能使用的功能	473
		22.4 zsh shell	477
		22.5 zsh shell 的组成	478
		22.5.1 shell 选项	478
		22.5.2 内建命令	480
		22.6 zsh 脚本编程	485
		22.6.1 数学运算	485
		22.6.2 结构化命令	487
		22.6.3 函数	487
		22.7 小结	489
		第四部分 高级 shell 脚本编程主题	
		第 23 章 使用数据库	492
		23.1 MySQL 数据库	492
		23.1.1 安装 MySQL	492
		23.1.2 MySQL 客户端界面	494
		23.1.3 创建 MySQL 数据库对象	498
		23.2 PostgreSQL 数据库	500
		23.2.1 安装 PostgreSQL	501
		23.2.2 PostgreSQL 命令行界面	501
		23.2.3 创建 PostgreSQL 数据库对象	503

23.3 使用数据表	505	25.2.1 sendmail	541
23.3.1 创建数据表	505	25.2.2 Postfix	543
23.3.2 插入和删除数据	507	25.3 使用 Mailx 发送消息	545
23.3.3 查询数据	508	25.4 Mutt 程序	548
23.4 在脚本中使用数据库	509	25.4.1 安装 Mutt	548
23.4.1 连接到数据库	509	25.4.2 Mutt 命令行	548
23.4.2 向服务器发送命令	511	25.4.3 使用 Mutt	549
23.4.3 格式化数据	514	25.5 小结	551
23.5 小结	516		
第 24 章 使用 Web	517	第 26 章 编写脚本实用工具	552
24.1 Lynx 程序	517	26.1 监测磁盘空间	552
24.1.1 安装 Lynx	518	26.1.1 需要的功能	552
24.1.2 Lynx 命令行	518	26.1.2 创建脚本	555
24.1.3 Lynx 配置文件	523	26.1.3 运行脚本	556
24.1.4 Lynx 环境变量	524	26.2 进行备份	557
24.1.5 从 Lynx 中抓取数据	524	26.3 管理用户账户	563
24.2 cURL 程序	527	26.3.1 需要的功能	563
24.2.1 安装 cURL	527	26.3.2 创建脚本	569
24.2.2 探索 cURL	527	26.4 小结	575
24.3 使用 zsh 处理网络	528		
24.3.1 TCP 模块	528	第 27 章 shell 脚本编程进阶	576
24.3.2 客户端/服务器模式	529	27.1 监测系统统计数据	576
24.3.3 使用 zsh 进行 C/S 编程	530	27.1.1 系统快照报告	576
24.4 小结	533	27.1.2 系统统计数据报告	582
第 25 章 使用 E-mail	534	27.2 问题跟踪数据库	589
25.1 Linux E-mail 基础	534	27.2.1 创建数据库	589
25.1.1 Linux 中的 E-mail	534	27.2.2 记录问题	591
25.1.2 邮件传送代理	535	27.2.3 更新问题	594
25.1.3 邮件投递代理	536	27.2.4 查找问题	599
25.1.4 邮件用户代理	537	27.3 小结	602
25.2 建立服务器	540		
		附录 A bash 命令快速指南	604
		附录 B sed 和 gawk 快速指南	611

Part 1

第一部分

Linux 命令行

本部分内容

- 第 1 章 初识 Linux shell
- 第 2 章 走进 shell
- 第 3 章 基本的 bash shell 命令
- 第 4 章 更多的 bash shell 命令
- 第 5 章 使用 Linux 环境变量
- 第 6 章 理解 Linux 文件权限
- 第 7 章 管理文件系统
- 第 8 章 安装软件程序
- 第 9 章 使用编辑器

本章内容

- 什么是Linux
- Linux内核的组成部分
- 探索Linux桌面
- 了解Linux发行版

在深入研究如何使用Linux命令行和shell之前，最好先了解一下什么是Linux、它的历史及运作方式。本章将带你逐步了解什么是Linux，并介绍命令行和shell在Linux整体架构中的位置。

1.1 什么是Linux

如果以前从未接触过Linux，你可能会对为什么会有这么多不同的Linux发行版有些困惑。在看Linux软件包时，你肯定听过发行版、LiveCD和GNU之类的术语，也肯定被搞晕过。第一次接触Linux的人理解这些会有些困难。本章将在你了解命令和脚本之前，揭示Linux系统内部结构的一些信息。

首先，Linux可划分为以下四部分：

- Linux内核；
- GNU工具组件；
- 图形化桌面环境；
- 应用软件。

在Linux系统里，这四部分中的每一部分都扮演着一个特别的角色。如果将它们分开来，每一部分都没太大的作用。图1-1是Linux系统的基本结构框图，说明了各部分是如何协作起来构成整个Linux系统的。

本节将会详细介绍这四部分，然后将概述它们是如何一起协作构成一个完整的Linux系统的。

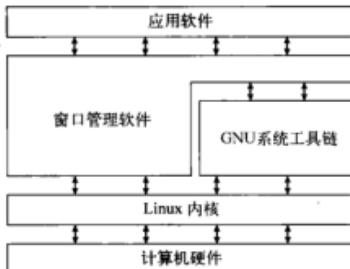


图1-1 Linux系统

1.1.1 深入探究Linux内核

Linux系统的核心是内核。内核控制着计算机系统上的所有硬件和软件；必要时分配硬件，有需要时执行软件。

如果你一直都在关注Linux世界，那么毫无疑问，你肯定听说过Linus Torvalds。Linus还在赫尔辛基大学上学时就开发了第一版Linux内核。起初他只是希望Linux成为Unix操作系统的一份副本，因为当时Unix操作系统在很多大学都很流行。

当Linus完成了开发工作后，他将Linux内核发布到了互联网社区并征求改进意见。这个简单的动作引发了计算机操作系统领域内的一场革命。很快，Linus就收到了来自世界各地的学生和专业程序员的各种建议。

如果Linux内核允许任何人修改内核程序代码，那么随之而来的将是完全的混乱。简单起见，Linus担当起了所有改进建议的把关员。能否将建议代码放进内核完全取决于Linus的决定。时至今日，这种概念依然在Linux内核代码开发过程中继续使用，所不同的是，现在是由一组开发人员来做这件事，而不再是Linus一个人了。

内核基本负责以下四项主要功能：

- 系统内存管理；
- 软件程序管理；
- 硬件设备管理；
- 文件系统管理。

后面几节将会进一步探究其中的每一项功能。

1. 系统内存管理

操作系统内核的基本功能之一是内存管理。内核不仅管理服务器上的可用物理内存，还可以创建和管理虚拟内存（即并不实际存在的内存）。

内核通过硬盘上的存储空间来实现虚拟内存，这块区域称为交换空间（swap space）。内核不

不断地在交换空间和实际的物理内存之间反复交换虚拟内存存储单元中的内容。这使得系统以为它拥有比物理内存更多的可用内存（如图1-2所示）。

内存存储单元会被按组分成很多块，这些块称作页面（page）。内核会将每个内存页面放在物理内存或交换空间。然后，内核会维护一个内存页面表，来指明哪些页面位于物理内存内，哪些页面被换到磁盘上。

内核会记录哪些内存页面正在使用中，并自动把一段时间未访问的内存页面复制到交换空间区域（称之为换出，swapping out）——即使还有可用内存。当程序要访问一个已被换出的内存页面时，内核必须从物理内存换出另外一个内存页面来给它让出空间，然后从交换空间换入（swapping in）请求的内存页面。显然，这个过程要花费时间，并使得运行中的进程变慢。只要Linux系统在运行，为运行中的程序换出内存页面的过程就不会停歇。

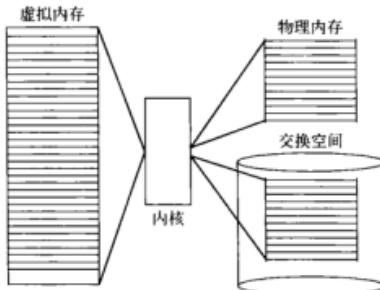


图1-2 Linux系统内存映射

你可以查看专门的/proc/meminfo文件来观察Linux系统上虚拟内存的当前状态。下面是/proc/meminfo文件的一个样例。

```
rich@rich-desktop:~$ cat /proc/meminfo
MemTotal:      1026084 kB
MemFree:       666356 kB
Buffers:        49900 kB
Cached:        152272 kB
SwapCached:      0 kB
Active:        171468 kB
Inactive:      154196 kB
Active(anon):   131056 kB
Inactive(anon):    32 kB
Active(file):   40412 kB
Inactive(file): 154164 kB
Unevictable:     12 kB
Mlocked:        12 kB
HighTotal:     139208 kB
HighFree:       252 kB
```

```

LowTotal:          886876 kB
LowFree:           666104 kB
SwapTotal:         2781176 kB
SwapFree:          2781176 kB
Dirty:              588 kB
Writeback:          0 kB
AnonPages:         123500 kB
Mapped:             52232 kB
Shmem:              7600 kB
Slab:               17676 kB
SReclaimable:      9788 kB
SUnreclaim:        7888 kB
KernelStack:       2656 kB
PageTables:        5072 kB
NFS_Unstable:      0 kB
Bounce:              0 kB
WritebackTmp:       0 kB
CommitLimit:       3294216 kB
Committed_AS:     1234480 kB
VmallocTotal:      122880 kB
VmallocUsed:       7520 kB
VmallocChunk:      110672 kB
HardwareCorrupted: 0 kB
HugePages_Total:    0
HugePages_Free:    0
HugePages_Rsvd:    0
HugePages_Surp:    0
Hugepagesize:      4096 kB
DirectMap4k:        12280 kB
DirectMap4M:       897024 kB
rich@rich-desktop:~$
```

MemTotal: 行表明这个Linux服务器有1 GB 的内存，该文件还表明大约有660 MB 的空闲空间 (MemFree)。输出表明这个系统上大约有2.5 GB的交换空间 (SwapTotal)。

默认情况下，运行在Linux系统上的每个进程都有各自的内存页面。进程不能访问其他进程正在使用的内存页面。内核维护着它自己的内存区域。出于安全考虑，用户进程不能访问内核进程使用的内存。

为了方便共享数据，你可以创建一些共享内存页面。多个进程可在同一块共用内存区域进行读取和写入操作。内核负责维护和管理这块共用内存区域并控制每个进程访问这块共享区域。

ipcs命令专门用来查看系统上的当前共享内存页面。以下是ipcs命令示例的输出：

```

# ipcs -m
----- Shared Memory Segments -----
key      shmid   owner    perms      bytes  nattch  status
0x00000000 0        rich    600      52228      6      dest
0x395ec51c 1        oracle   640     5787648      6
```

#

每个共享内存段都有个所有者，也就是创建它的用户。每个段也都有标准的Linux权限设置来设定其他用户是否可以访问该段。这个键值用来限定其他用户是否可以访问共享内存段。

2. 软件程序管理

Linux操作系统称运行中的程序为进程。进程可以在前台运行，将输出显示在屏幕上；也可以在后台运行，隐藏到幕后。内核控制着Linux系统如何管理运行在系统上的所有进程。

内核创建了第一个进程（称为init进程）来启动系统上所有其他进程。当内核启动时，它会将init进程加载到虚拟内存中。内核在启动任何其他进程时，都会在虚拟内存中给新进程分配一块专有区域来存储该进程用到的数据和代码。

一些Linux发行版使用一个表来管理在系统开机时要自动启动的进程。在Linux系统上，这个表通常位于专门文件/etc/inittab中。

另外一些系统（比如现在流行的Ubuntu Linux发行版）则采用/etc/init.d目录，将开机时启动或停止某个应用的脚本放在这个目录下。这些脚本通过/etc/rcX.d目录下的人口^①启动，这里X代表运行级（run level）。

Linux操作系统的init系统采用了运行级。运行级决定了init进程运行/etc/inittab文件或/etc/rcX.d目录中定义好的某些特定类型的进程。Linux操作系统有5个启动运行级。

运行级为1时，只有基本的系统进程会启动，同时会启动唯一一个控制台终端进程。我们称之为单用户模式。单用户模式通常用来在系统有问题时进行紧急的文件系统维护。显然，在这种模式下仅有一个人（通常是系统管理员）能登录到系统上操作数据。

标准的启动运行级是3。在这个运行级上，大多数应用软件，比如网络支持程序，都会启动。另一个Linux中常见的运行级是5。在这个运行级上系统会启动图形化的X Window系统，同时允许用户通过图形化桌面窗口登录系统。

Linux系统可以通过调整运行级来控制整个系统的功能。通过将运行级从3调整成5，系统就可以从基于控制台的系统变成更先进的图形化X Window系统。

在第4章，你将会学习如何使用ps命令查看当前运行在Linux系统上的进程。下面示例演示了使用ps命令时获得的输出：

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:03 init
    2 ?        SW     0:00 [kflushd]
    3 ?        SW     0:00 [kupdate]
    4 ?        SW     0:00 [kpiod]
    5 ?        SW     0:00 [kswappd]
 243 ?        SW     0:00 [portmap]
 295 ?        S      0:00 syslogd
 305 ?        S      0:00 klogd
 320 ?        S      0:00 /usr/sbin/atd
 335 ?        S      0:00 crond
 350 ?        S      0:00 inetd
```

^① 这些入口实际上是到/etc/init.d目录中启动脚本的符号链接。——译者注

```

365 ?      SW    0:00 [lpd]
403 tty50   S     0:00 gpm -t ms
418 ?      S     0:00 httpd
423 ?      S     0:00 httpd
424 ?      SW    0:00 [httpd]
425 ?      SW    0:00 [httpd]
426 ?      SW    0:00 [httpd]
427 ?      SW    0:00 [httpd]
428 ?      SW    0:00 [httpd]
429 ?      SW    0:00 [httpd]
430 ?      SW    0:00 [httpd]
436 ?      SW    0:00 [httpd]
437 ?      SW    0:00 [httpd]
438 ?      SW    0:00 [httpd]
470 ?      S     0:02 xfs -port -1
485 ?      SW    0:00 [smbd]
495 ?      S     0:00 nmrd -D
533 ?      SW    0:00 [postmaster]
538 tty1   SW    0:00 [mingetty]
539 tty2   SW    0:00 [mingetty]
540 tty3   SW    0:00 [mingetty]
541 tty4   SW    0:00 [mingetty]
542 tty5   SW    0:00 [mingetty]
543 tty6   SW    0:00 [mingetty]
544 ?      SW    0:00 [prefdm]
549 ?      SW    0:00 [prefdm]
559 ?      S     0:02 [kwm]
585 ?      S     0:06 kikbd
594 ?      S     0:00 kwmsound
595 ?      S     0:03 kpanel
596 ?      S     0:02 kfm
597 ?      S     0:00 krootwm
598 ?      S     0:01 kbgrndwm
611 ?      S     0:00 kcmlaptop -daemon
666 ?      S     0:00 /usr/libexec/postfix/master
668 ?      S     0:00 qmgr -l -t fifo -u
787 ?      S     0:00 pickup -l -t fifo
790 ?      S     0:00 telnetd: 192.168.1.2 [vt100]
791 pts/0   S     0:00 login -- rich
792 pts/0   S     0:00 -bash
805 pts/0   R     0:00 ps ax
$
```

第一列输出显示了进程的进程号（Process ID，即PID）。注意，第一个进程就是我们的好朋友init进程，Linux系统分配给它的PID值是1。之后所有其他进程的PID都是按序分配的。没有两个进程拥有相同的PID，虽然旧的PID数值在原进程结束后会被系统重新使用。

第三列显示了进程的当前状态（S代表在睡眠，SW代表在睡眠和等待，R代表在运行中）。进程名字显示在最后一列。方括号中的进程是由于不活动而被从内存中换出到磁盘交换空间的进程。你能发现有些进程被换出了，但大部分运行中的进程未被换出。

3. 硬件设备管理

内核的另一职责是管理硬件设备。任何Linux系统需要与之通信的设备，都需要在内核代码中加入其驱动程序代码（driver code）。驱动程序代码相当于应用程序和硬件设备的中间人，允许内核同设备之间交换数据。在Linux内核中有两种方法用来插入设备驱动代码：

- 编译进内核的设备驱动代码；
- 可插入内核的设备驱动模块。

以前，插入设备驱动代码的唯一途径是重新编译内核。每次给系统添加新设备时，你都要重新编译一遍内核代码。随着Linux内核支持越来越多的硬件设备，这个过程也变得越来越低效。不过好在Linux开发人员设计出了一种更好的将驱动代码插入运行中的内核的方法。

开发人员提出了内核模块的概念。它允许将驱动代码插入到运行中的内核而无需重新编译内核。同时，当设备不再使用时也可将内核模块从内核中移走。这种方式极大地简化和推动了硬件设备在Linux上的使用。

Linux系统将硬件设备当成特殊的文件，称为设备文件。设备文件有3种不同的分类：

- 字符型设备文件；
- 块设备文件；
- 网络设备文件。

字符型设备文件是指处理数据时每次只能处理一个字符的设备。大多数类型的调制解调器和终端都是作为字符型设备文件创建的。块设备文件是指处理数据时每次能处理大块数据的设备，比如硬盘。

网络设备文件是指采用数据包发送和接收数据的设备，包括各种网卡和一个特殊的回环设备。这个回环设备允许Linux系统使用通用的网络编程协议同自己通信。

Linux为系统上的每个设备都创建一种特殊的文件，称为“节点”。与设备的所有通信都是通过设备节点完成的。每个节点都有一个唯一的数值对，供Linux内核标识它。数值对包括一个主设备号和一个次设备号。类似的设备被划分到同样的主设备号下。次设备号用于标识同一主设备号下的某个特殊设备。以下是在Linux服务器上的一些设备文件的例子：

```
rich@rich-desktop: ~$ cd /dev
rich@rich-desktop:~/dev$ ls -al sda* ttyS*
brw-rw---- 1 root disk 8. 0 2010-09-18 17:25 sda
brw-rw---- 1 root disk 8. 1 2010-09-18 17:25 sda1
brw-rw---- 1 root disk 8. 2 2010-09-18 17:25 sda2
brw-rw---- 1 root disk 8. 5 2010-09-18 17:25 sda5
crw-rw---- 1 root dialout 4. 64 2010-09-18 17:25 ttyS0
crw-rw---- 1 root dialout 4. 65 2010-09-18 17:25 ttyS1
crw-rw---- 1 root dialout 4. 66 2010-09-18 17:25 ttyS2
crw-rw---- 1 root dialout 4. 67 2010-09-18 17:25 ttyS3
rich@rich-desktop:/dev$
```

不同的Linux发行版在处理设备时采用不同的设备名。在这个发行版上，sda设备是第一个ATA硬盘，ttyS设备是标准的IBM PC COM端口。这个清单显示了示例Linux系统上创建的所有sda设备。虽然并不都在实际中使用，但它们都被创建出来了以备系统管理员不时之需。类似地，这

个列表显示了已创建的所有ttyS设备。

第5列是主设备节点号。注意，所有sda设备都拥有同一主设备号8；而所有的ttyS设备都使用4。第6列是次设备节点号。同一个主设备节点号下的每个设备都拥有自己唯一的次设备节点号。

第1列显示了该设备文件的权限。权限的第一个字符表示的是设备文件的类型。注意，ATA硬盘文件都被标记为b (block device, 块设备)，而COM端口设备文件则被标记为c (character device, 字符型设备)。

4. 文件系统管理

不同于其他一些操作系统，Linux内核支持多种不同类型的文件系统来从硬盘中读取或写入数据。除了自有的诸多文件系统外，Linux还支持从其他操作系统（比如Microsoft Windows）所采用的文件系统中读取或写入数据。内核必须在编译时就加入对所有可能用到的文件系统的支持。表1-1列出了Linux系统用来读写数据的标准文件系统。

表1-1 Linux文件系统

文件系统	描述
ext	Linux扩展文件系统，最早的Linux文件系统
ext2	第二扩展文件系统，在ext的基础上提供了更多的功能
ext3	第三扩展文件系统，支持日志功能
ext4	第四扩展文件系统，支持高级日志功能
hpfs	OS/2高性能文件系统
jfs	IBM日志文件系统
iso9660	ISO 9660文件系统（CD-ROM）
minix	MINIX文件系统
msdos	微软的FAT16
ncp	Netware文件系统
nfs	网络文件系统
ntfs	支持Microsoft NT文件系统
proc	访问系统信息
ReiserFS	高级Linux文件系统，能提供更好的性能和硬盘恢复功能
smb	支持网络访问的Samba SMB文件系统
sysv	较早期的Unix文件系统
ufs	BSD文件系统
umsdos	贮存在msdos上的类Unix文件系统
vfat	Windows 95文件系统（FAT32）
XFS	高性能64位日志文件系统

任何供Linux服务器访问的硬盘都必须格式化成表1-1所列文件系统类型中的一种。

Linux内核采用虚拟文件系统（Virtual File System, VFS）作为和每个文件系统交互的接口。

这为Linux内核同任何类型文件系统通信提供了一个标准接口。当每个文件系统被挂载和使用时，VFS将信息都缓存在内存中。

1.1.2 GNU工具链

除了有内核来控制硬件设备外，操作系统还需要工具链来执行一些标准功能，比如控制文件和程序。当Linus创建Linux系统内核时，是没有系统工具链运行其上的。然而他很幸运，就在他开发Linux内核的同时，有一组人正在互联网上共同努力，模仿Unix操作系统开发一系列标准的计算机系统工具。

GNU组织（GNU代表GNU's Not Unix）开发了一套完整的Unix工具链，但没有可以运行它们的内核系统。这些工具链是在开源软件（Open Source Software, OSS）的软件开发理念下开发的。

开源软件理念允许程序员开发软件并将其免费发布。任何人都可以使用、修改该软件，或将该软件集成进自己的系统，而无需支付任何授权费用。将Linus的Linux内核和GNU操作系统工具整合起来，就可以创造一个完整的、功能丰富的免费操作系统。

尽管通常我们将Linux内核和GNU工具链的结合体称为Linux，你也会在互联网上看到一些Linux纯粹主义者将其称为GNU/Linux系统来表彰GNU组织为此所作的贡献。

1. 核心GNU工具链

GNU项目一开始主要是为Unix系统管理员设计的，用以提供一个类Unix环境。这个目标导致这个项目移植了很多Unix系统通用的命令行工具。为Linux系统提供的一组核心工具被称为coreutils（core utilities）软件包。

GNU coreutils软件包由3部分构成：

- 用以处理文件的工具；
- 用以操作文本的工具；
- 用以管理进程的工具。

这三组主要工具中的每一组都包含一些对Linux系统管理员和程序员至关重要的工具。本书将详细介绍GNU coreutils软件包中包含的所有工具。

2. shell

GNU/Linux shell是个交互式工具。它为用户提供了启动程序、管理文件系统上的文件以及管理运行在Linux系统上的进程的途径。shell的核心是命令行提示符。命令行提示符是shell的交互部分。它允许你输入文本命令，之后将解释命令并在内核中执行。

shell包含了一组内置命令，你可以用这些命令来完成一些操作，例如复制文件、移动文件、重命名文件以及显示和终止系统上正运行的程序。shell也允许你在命令行提示符中输入程序的名称，它会将程序的名称传递给内核以启动它。

你也可以将shell命令放入文件中作为程序执行。这些文件被称作shell脚本。你在命令行上执行的任何命令都可放进一个shell脚本中作为一组命令执行。这为创建那种需要把几个命令放在一

起来工作的工具提供了便利。

在Linux系统上，通常有好几种Linux shell可用。不同的shell有不同的特性，有些更利于创建脚本，有些更利于管理进程。所有Linux发行版默认的shell基本上都是bash shell。bash shell作为标准Unix shell——Bourne shell（沿用创建者的名字）的替代，由GNU项目开发的。bash shell的名称就是针对这个Bourne shell的文字游戏，称为Bourne again shell。

除了bash shell外，我们在本书中还将介绍其他几种常见的shell。表1-2列出了Linux中常见的几种不同shell。

表1-2 Linux shell

shell	描述
ash	运行在内存受限环境中简单的、轻量级shell，但与bash shell完全兼容
korn	与Bourne shell兼容的编程shell，但支持一些高级的编程特性，比如关联数组和浮点运算
tcsh	将C语言中的一些元素引入到shell脚本中的shell
zsh	将bash、tcsh和korn的特性引入，同时提供高级编程特性、共享历史文件和主题化提示符的高级shell

大多数Linux发行版包含多个shell，虽然它们通常会采用其中一个作为默认shell。如果你的Linux发行版包含多个shell，尽情尝试不同的shell，看看哪个能满足你的需要。

1.1.3 Linux桌面环境

在Linux的早期（20世纪90年代早期），系统上可用的仅是一个简单的与Linux操作系统交互的文本界面。这个文本界面允许系统管理员运行程序、控制程序的执行以及在系统中移动文件。

随着Microsoft Windows的普及，电脑用户期望的就不仅仅是对着老式的文本界面工作了。这点推动了OSS社区的更多开发活动，Linux图形化桌面环境出现了。

Linux一直都以可用多种方式来完成工作而声名在外。在图形化桌面上更是如此。Linux有各种图形化桌面可供选择。后面几节中将会介绍其中比较流行的桌面。

1. X Window系统

有两项基本组件能决定你的视频环境：显卡和显示器。要在电脑上显示绚丽的画面，Linux软件就得知道如何来连接它们。X Window软件是图形显示的核心元素。

X Window软件是直接和PC上的显卡以及显示器一起工作的底层软件。它控制着Linux程序如何在电脑上显示出绚丽的窗口和画面。

Linux并非唯一使用X Window的操作系统，X Window有针对多种不同操作系统的版本。在Linux世界里，仅有两个软件包能实现X Window。

XFree86软件包是二者中较早的那个，并且在很长一段时间里都是仅有的为Linux开发的X Window包。顾名思义，它是X Window的免费开源版本。

二者中较新的X.org在Linux世界里占据优势，现在是二者中较为普遍的。它同样提供了X Window系统的开源实现，但支持的更多是现今在用的较新显卡。

两个软件包都以同样的方式工作，控制着Linux如何使用显卡来在显示器上显示内容。为此，你需要针对你的系统专门配置它们。一般地，这个配置过程会在安装Linux时自动完成。

在首次安装Linux发行版时，它会检测显卡和显示器，然后创建一个含有必要信息的X Window配置文件。在安装过程中，你可能会注意到安装程序会检测一次显示器，以此来确定它所支持的视频模式。有时显示器会黑屏几秒。由于现在有多种不同类型的显卡和显示器，这个过程可能会需要一段时间来完成。

核心的X Window软件可以产生图形化显示环境，但仅此而已。虽然对于运行独立应用这已经足够，但在日常PC使用中却并不是那么有用。它没有桌面环境供用户操作文件或是开启程序。为此，你需要一个在X Window系统软件之上的桌面环境。

2. KDE桌面

KDE（K Desktop Environment，K桌面环境）最初于1996年作为开源项目发布。它会生成一个类似于Microsoft Windows的图形化桌面环境。如果你是Windows用户，KDE集成了所有你熟悉的功能。图1-3显示了运行在openSuSE Linux发行版上的KDE 4桌面。

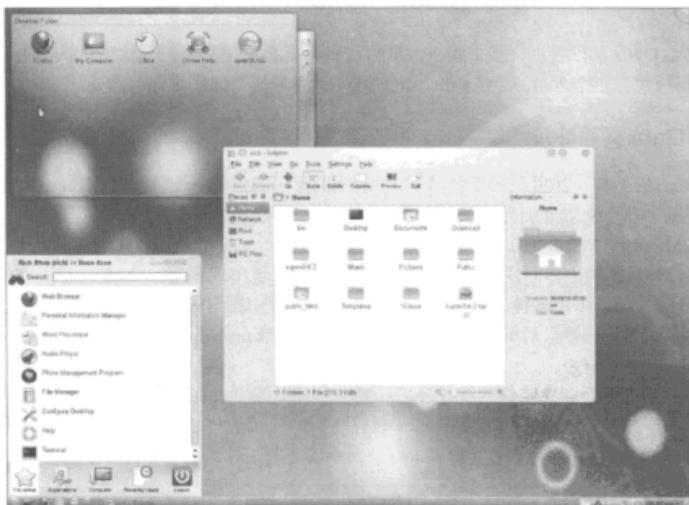


图1-3 openSuSE Linux系统上的KDE 4桌面

KDE桌面允许你把应用程序图标和文件图标放置在桌面的特定位置上。单击应用程序图标，Linux系统就会运行该应用程序。单击文件图标，KDE桌面就会确定使用哪种应用程序来处理该文件。

桌面底部的横条称为面板，它由以下四部分构成。

- **KDE菜单：**和Windows的开始菜单非常类似，KDE菜单包含了启动已安装程序的链接。
- **程序快捷方式：**在面板上有直接从面板启动程序的快速链接。
- **任务栏：**任务栏显示着当前桌面正运行的程序的图标。
- **小应用程序：**面板上还有一些特殊小应用程序的图标，这些图标常常会根据小应用程序的状态发生变化。

所有的面板功能都和你在Windows上看到的类似。除了桌面功能，KDE项目还开发了大量的可运行在KDE环境中的应用程序。表1-3列出了这些程序。（注意，作为惯例，KDE应用命名时经常有个大写的K。）

表1-3 KDE应用程序

应用程序	描述
amaroK	音频播放器
digiKam	数码相机软件
dolphin	文件管理器
K3b	CD烧录软件
Kaffeine	视频播放器
Kmail	E-mail客户端
Koffice	Office应用套件
Konqueror	文件和Web浏览器
Kontact	个人信息管理器
Kopete	即时消息客户端

这里仅列出了KDE项目开发的部分应用。在KDE桌面中还包含着更多的应用。

3. GNOME桌面

GNOME（The GNU Network Object Model Environment，GNU网络对象模型环境）是另一个流行的Linux桌面环境。GNOME于1999年首次发布，现已成为许多Linux发行版（最普遍的是Red Hat Linux）默认的桌面环境。

尽管GNOME决定不再沿用Microsoft Windows的标准外观，但它还是集成了许多Windows用户习惯的功能：

- 一块放置图标的桌面区域；
- 两个面板区域；
- 拖放功能。

图1-4显示了Ubuntu Linux发行版采用的标准GNOME桌面。

不甘示弱于KDE，GNOME开发人员也开发了一组集成进GNOME桌面的图形化程序，如表1-4所示。

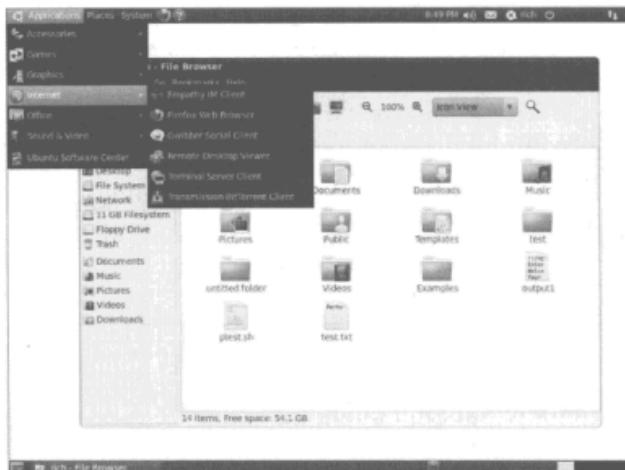


图1-4 Ubuntu Linux系统上的GNOME桌面

表1-4 GNOME应用程序

应用程序	描述
epiphany	Web浏览器
evince	文档查看器
gcalc-tool	计算器
gedit	GNOME文本编辑器
gnome-panel	桌面面板，用以启动程序
gnome-nettool	网络诊断工具
gnome-terminal	终端模拟器
nautilus	图形化文件管理器
nautilus-cd-burner	CD烧录工具
sound juicer	音频CD抓轨工具
tomboy	笔记程序
totem	多媒体播放器

如你所见，对于GNOME桌面，同样有不少可用的应用程序。除了这些程序，大多数采用GNOME桌面的Linux发行版还集成了KDE库，允许用户在GNOME桌面上运行KDE应用程序。

4. 其他桌面

图形化桌面环境的弊端在于它们要占用相当一部分系统资源。在Linux发展之初，Linux的特

征和卖点之一就是，它可以运行在功能较弱的早期PC上运行，而这些PC无法运行较新的微软桌面。然而随着KDE和GNOME桌面环境的普及，情况发生了变化。运行KDE或GNOME桌面要占用和微软的最新桌面环境一样多的内存资源。

如果你的PC比较老旧，也不要泄气。Linux开发人员已经联手让Linux返璞归真。他们开发了一些低内存开销的图形化桌面环境，仅提供能在早期PC上完美运行的基本功能。

尽管这些图形化桌面环境并没有专为其设计的大量应用，但它们仍然能运行许多基本的图形化程序，支持如文字处理、电子表格、数据库、绘图以及多媒体等功能。

表1-5列出了一些可在配置较低的PC和笔记本电脑上运行的轻量级Linux图形化桌面环境。

表1-5 其他Linux图形化桌面

桌面	描述
fluxbox	一个没有面板的轻型桌面，仅有一个弹出式菜单来启动程序
xfce	和KDE很像的一个桌面，但少了很多图形以适应低内存环境
JWM	Joe的窗口管理器（Joe's Window Manager），适用于低内存低硬盘空间环境的理想超微型桌面
fwm	支持诸如虚拟桌面和面板等高级桌面功能，但在低内存环境中运行
fwm95	从fwm衍生而来，但看起来更像是Windows 95桌面

这些图形化桌面环境并不如KDE或GNOME桌面一样绚丽，但它们提供了基本的图形化功能。图1-5显示了Puppy Linux antiX发行版所采用的JWM桌面的外观。



图1-5 Puppy Linux发行版所采用的JWM桌面

如果你用的是早期PC，尝试一下基于上述某个桌面环境的Linux发行版，看看会怎么样。你可能会大吃一惊。

1.2 Linux发行版

到此为止，你已经了解了构成完整Linux系统所需要的4个关键部件，你可能在考虑要怎样才能把它们放在一起构建一个Linux系统。幸运的是，已经有人为你做好这些了。

我们将完整的Linux系统包称为发行版。有各种不同的Linux发行版来满足可能存在的各种运算需求。大多数发行版是为某个特定用户群定制的，比如商业用户、多媒体爱好者、软件开发人员或者普通家庭用户。每个定制的发行版都支持特定功能所需的各种软件包，比如为多媒体爱好者准备的音频和视频编辑软件，为软件开发人员准备的编译器和集成开发环境。

不同的Linux发行版通常归类为三种：

- 完整的核心Linux发行版；
- 专业发行版；
- LiveCD测试发行版。

后面几节将会探讨这些不同类型的Linux发行版，然后会展示每种类型中的一些Linux发行版示例。

1.2.1 核心Linux发行版

核心Linux发行版含有内核、一个或多个图形化桌面环境以及预编译好的几乎所有能见到的Linux应用。它提供了一站式的完整Linux安装。表1-6列出了较流行的核心Linux发行版。

表1-6 核心Linux发行版

发 行 版	描 述
Slackware	最早的Linux发行版中的一员，在Linux极客中比较流行
Red Hat	一个主要用于Internet服务器的商业发行版
Fedora	从Red Hat分离出的家用发行版
Gentoo	为高级Linux用户设计的发行版，仅包含Linux源代码
Mandriva	主要是家用（之前叫Mandrake）
openSuSE	用于商用和家用的发行版
Debian	在Linux专家和商用Linux产品中流行的发行版

在Linux的早期，发行版是作为一叠软盘发布的。你必须下载多组文件然后将其复制到软盘上。通常要用20张或更多的软盘来创建一个完整的发行版。毋庸多言，这是个痛苦的过程。

现今，家用电脑基本都有内置的CD和DVD光驱，Linux发行版也就用几张CD光盘或单张DVD光盘来发布。这大大简化了Linux的安装过程。

然而当新手在安装核心Linux发行版时，他们仍经常遇到各种各样的问题。为了照顾到想用

Linux的人可能碰到的各种情形，单个发行版也必须包含很多应用软件。它们包含了从高端Internet数据库服务器到通用游戏的所有软件。鉴于Linux上可用的大量应用程序，一个完整的发行版通常至少要4张CD。

发行版中的大量可选配置对于Linux极客来说是好事，但对于新手来说就是一场噩梦。多数发行版会在安装过程中问一系列问题以决定哪些应用要默认加载、哪些硬件是连接到PC的以及怎样配置硬件设备。新手经常会被这些问题困扰，因此他们经常在机器上加载了过多的程序或没有加载足够的程序以至于最后发现计算机没有如他们期望那样的工作。

对新手来说幸运的是，还有更简便的安装Linux的方法。

1.2.2 专业Linux发行版

Linux发行版的一个子群开始出现了。它们通常基于某个主流发行版，但仅包含主流发行版中一小部分用于某种特定用途的程序。

除了提供专业软件外（比如仅为商业用户提供的办公应用），专业发行版还尝试通过自动检测和自动配置来帮助新手安装Linux。这让安装Linux变得更容易。

表1-7列出了一些专业Linux发行版以及它们的专长。

表1-7 专业Linux发行版

发 行 版	描 述
Xandros	一个为新手配置的商业Linux发行版
SimplyMEPIS	一个免费的家用Linux发行版
Ubuntu	一个免费的学校和家庭用的Linux发行版
PCLinuxOS	一个免费的家庭和公用的Linux发行版
Mint	一个免费的家庭娱乐用的Linux发行版
dyne:bolic	一个免费的包含音频和MIDI应用程序的Linux发行版
Puppy Linux	一个免费的适用于早期PC的小型Linux发行版

这只是专业Linux发行版中的一小部分而已。大约有数百个专业Linux发行版，而在互联网上还不断有新的出现。不管你的专长是什么，你都能找到一个为你量身定做的Linux发行版。

许多专业Linux发行版都是基于Debian Linux的。它们使用和Debian一样的安装文件，但仅打包了完整Debian系统中的一部分软件。

1.2.3 Linux LiveCD

Linux世界中一个相对比较新的现象是可引导启动的Linux CD发行版的出现。它允许不安装Linux就可以看看Linux系统是什么样的。多数现代PC都能从CD启动，而不是必须从标准硬盘启动。基于这点，一些Linux发行版创建了包含有示例Linux系统（称为Linux LiveCD）的可引导启动CD。由于单张CD容量的限制，示例系统无法包含一个完整的Linux系统，但你会为它们能加入的各种程序感到欣喜。最终你可以通过CD来启动PC并且无需在硬盘安装任何东西就能运行Linux。

发行版了。

这是一个不弄乱PC而体验各种Linux发行版的绝妙方法。只需插入CD就能引导了！所有的Linux软件都将直接从CD上运行。有很多Linux LiveCD可供你从互联网上下载、刻录到CD上体验。

表1-8列出了可用的一些流行Linux LiveCD。

表1-8 Linux LiveCD发行版

发 行 版	描 述
Knoppix	一个德语Linux发行版，最早的LiveCD Linux
SimplyMEPIS	为家庭用户设计的Linux发行版
PCLinuxOS	成熟的LiveCD上的Linux发行版
Ubuntu	为多种语言设计的世界级项目
Slax	基于Slackware Linux的LiveCD Linux
Puppy Linux	为早期PC设计的全功能Linux

你能在这张表中看到熟悉的面孔。许多专业Linux发行版都有Linux LiveCD版本。一些Linux LiveCD发行版，比如Ubuntu，允许直接从LiveCD安装整个发行版。这使你能从CD引导启动Linux体验此Linux发行版；如果喜欢，就安装到硬盘上。这个功能极其方便易用。

就像所有美好的事物一样，Linux LiveCD也有一些美中不足的地方。由于要从CD上访问所有东西，应用程序会运行得更慢，尤其当你把缓慢的早期机器和CD光驱放在一起用时。还有，由于无法向CD写入数据，对Linux系统作的任何修改在重启后都会失效。

不过，有一些Linux LiveCD的改进帮助解决了上述一些问题。这些改进包括：

- 能将CD上的Linux系统文件复制到内存中；
- 能将系统文件复制到硬盘上；
- 能在U盘上存储系统设置；
- 能在U盘上存储用户设置。

一些Linux LiveCD，如Puppy Linux，被设计成有极少量的Linux系统文件。当CD引导启动时，LiveCD的启动脚本直接把它们复制到内存中。这允许在Linux启动后立即把CD从光驱中取走。这不仅使得程序运行得更快（因为程序从内存中运行时更快），而且空出CD光驱供你用Puppy Linux自带的软件转录音频CD或播放视频DVD。

其他Linux LiveCD用另外的方法，同样允许你在启动后将CD从光驱中拿走。其中一种就是把核心Linux文件作为一个文件复制到Windows硬盘上。待CD启动后，系统会寻找那个文件，并从那个文件中读取系统文件。dyne:bolic Linux LiveCD采用的就是这种技术，我们称之为对接（docking）。当然，你必须在从CD引导启动之前把系统文件复制到硬盘里。

使用通用U盘（也称为闪存或闪盘）是一项非常流行的存储Linux LiveCD会话数据的技术。几乎每个Linux LiveCD都能识别插入的U盘（即使是在Windows下格式化的）并从U盘上读取或写入文件。这允许你启动Linux LiveCD，使用Linux应用来创建文件，将这些文件存储在U盘上，然后用Windows应用（或者在另外一台电脑上）访问这些文件。这该有多酷！

1.3 小结

本章探讨了Linux系统以及它是如何工作的这个基本概念。Linux内核是系统的核心，控制着内存、程序和硬件是如何与对方交互的。GNU工具链也是Linux系统中的一个重要部分。本书关注的焦点Linux shell是GNU核心工具集中的一部分。本章还讨论了Linux系统中的最后一个组件Linux桌面环境。随着时间推移，一切都发生了改变。现今Linux支持几个图形化桌面环境。

本章还探讨了各种Linux发行版。Linux发行版就是把Linux系统的各个不同部分汇集起来组成的单个易于安装的包。Linux发行版有囊括各种软件的成熟的核心Linux发行版，也有只包含针对某种特定功能软件包的专业发行版。Linux LiveCD则是一种无需将Linux安装到硬盘就能体验Linux的发行版。

下一章你将开始了解开启命令行和shell脚本编程体验所需的基本知识。你将了解如何从绚丽的图形化桌面环境获得Linux shell工具。那绝非易事。



本章内容

- 终端模拟
- terminfo数据库
- Linux控制台
- xterm终端
- Konsole终端
- GNOME Terminal

在Linux早期，可以用来工作的只有shell。那时，系统管理员、程序员和系统用户都坐在Linux命令行终端前，输入文本命令，查看文本输出。而现在，因为有了绚丽的图形化桌面环境，在系统上找到shell提示符都变得困难起来。本章将讨论提供命令行环境需要什么，然后带你逐步了解可能会在各种Linux发行版中碰到的终端模拟软件包。

2.1 终端模拟

在图形化桌面出现之前，和Unix系统交互的唯一方式就是通过shell提供的文本命令行界面（CLI，Command Line Interface）。CLI只允许输入文本，而且只能显示文本和低级图形输出。

由于这个限制，输出设备不必非常好。通常一个简单的哑终端就是和Unix系统交互所需要的所有设备了。哑终端（dumb terminal）通常是由通信电缆（通常是多线串行电缆，也叫带状电缆）连接到Unix系统上的显示器和键盘（尽管后来鼠标的出现多多少少改善了这种状况）。这个简单的组合提供了向Unix系统输入文本数据和显示文本结果的一条捷径。

如你所熟知的，今天的Linux环境已经完全不同了。几乎所有Linux发行版都采用了某种类型的图形化桌面环境。但要访问shell，你仍然需要一个文本显示来和CLI交互。于是现在的问题归结为一点：有了所有图形化桌面的新功能，有时在Linux发行版上找个进入CLI的途径还真不是件容易的事。

进入CLI的一个途径是让Linux系统退出图形化桌面模式，进入文本模式。这样在显示器上只提供了一个简单的shell CLI，就跟图形化桌面出现以前一样。这种模式称作Linux控制台，因为它

模拟了早期的硬接线（hard-wired）控制台终端，而且是跟Linux系统交互的直接接口。

进入Linux控制台的另一种办法是使用图形化Linux桌面环境里的终端模拟包，终端模拟包会模拟在哑终端上工作，所有的都在桌面上的一个图形化窗口中。图2-1显示了一个图形化Linux桌面环境上运行的终端模拟器的例子。

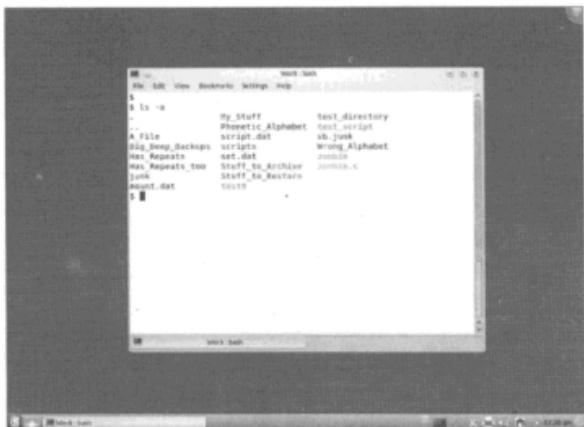


图2-1 Linux桌面上运行的简单终端模拟器

每个终端模拟包都可以模拟一种或多种特定类型的哑终端。如果你要使用Linux中的shell，很不幸你需要知道一点关于终端模拟的知识。

了解过去哑终端的核心功能，可以帮你在使用图形化终端模拟器时决定选用哪种模拟类型，并将所有可用的功能最大限度地发挥起来。哑终端中用到的主要功能可以分成两块：图形功能和键盘。本节将会介绍这些功能并讨论它们和不同类型的终端模拟器有多大的关系。

2.1.1 图形功能

终端模拟的最重要部分是它如何在显示器上显示信息。当你听到短语“文本模式”时，可能你最不会想到的就是图形。但即使是最低级的哑终端也支持某些屏幕操作方法（例如清空屏幕和在屏幕上特定位置显示文本）。

本节将会细述使每种不同的终端类型有别于其他类型的图形功能，以及在终端模拟包中都有什么。

1. 字符集

所有的终端都必须在屏幕上显示字符（否则，文本模式就没有意义了）。关键在于要显示什么样的字符以及Linux系统需要发送什么样的代码来显示它们。字符集是一组二进制命令，Linux

系统可以将它们发给显示器来显示字符。各种终端模拟包支持以下几种字符集。

- ASCII——美国信息交换标准码（American Standard Code for Information Interchange）。这个字符集中含有用7位码存储的英文字符，由128个英文字母（包括大小写）、数字和特殊符号组成。这个字符集由美国国家标准协会（ANSI）批准为US-ASCII。你会经常在终端模拟器中看到它被引用为ANSI字符集。
- ISO-8859-1（通常称为Latin-1）——ASCII字符集的一个扩展，由ISO（International Organization for Standardization，国际标准化组织）制定。它采用8位码来支持标准ASCII字符以及大多数西欧语言中的特殊外语字符。Latin-1字符集在多国终端模拟包中很流行。
- ISO-8859-2——ISO字符集，支持东欧语言字符。
- ISO-8859-6——ISO字符集，支持阿拉伯语字符。
- ISO-8859-7——ISO字符集，支持希腊语字符。
- ISO-8859-8——ISO字符集，支持希伯来语字符。
- ISO-10646（通常称为Unicode）——ISO双字节字符集，包含大部分英语和非英语语言的代码。这个字符集包含了所有ISO-8869-x系列字符集中定义的所有字符。Unicode字符集很快会在开源应用中流行起来。

到目前为止，英语国家中在用的最常见字符集是Latin-1字符集。Unicode字符集越来越流行，很有可能有一天成为字符集中的新标准。大部分流行的终端模拟器都允许你在终端模拟中选择要用哪个字符集。

2. 控制码

除了能显示字符外，终端还必须能控制显示器和键盘上的特殊功能，比如屏幕上光标的位置。终端用控制码系统来实现这个。控制码是未在字符集中使用的特殊代码，它会发信号给终端来执行特殊的非打印操作。

常见的控制码功能有回车（将光标返回到行首）、换行（将光标放到下一水平行）、水平制表（将光标移动指定数目的空格）、方向键（上、下、左、右）和翻页键（上翻和下翻）。这些代码主要模拟一些控制将光标放在显示器上什么位置的功能，但还有一些其他的代码，比如清空整个屏幕，甚至还有个铃声（模拟早期打字机的换行铃声）。

控制码也可以用来控制哑终端的通信功能。哑终端会通过某种类型的通信信道（通常是串行通信电缆）来连到计算机系统上。有时需要在通信信道上控制数据，所以开发人员就设计出只用于数据通信目的的特殊控制码。虽然这些代码在现代终端模拟器上并不是必需的，但大多数终端模拟器都支持这些代码以保持兼容性。这类中最常用的代码是XON和XOFF代码，它们分别开启和停止到终端的数据传输。

3. 块模式图形

由于哑终端逐渐地流行起来，制造商开始试验基本的图形功能。到目前为止，最流行的“图形化”哑终端类型是DEC（Digital Equipment Corporation，美国数字设备公司）的VT系列终端。1978年，伴随着DEC VT100的发布哑终端发生了转变。DEC VT100终端是第一个支持完整ANSI字符集（包括块模式图形字符）的终端。

ANSI字符集包含的代码不但允许显示器显示文本，而且允许显示基本的图形符号，比如框、线和块。到目前为止，20世纪80年代中Unix运行中使用的最流行的哑终端之一是VT100的升级版DEC VT102。大多数现代终端模拟程序仍然会模拟VT102显示的运行，支持所有的ANSI代码来创建块模式图形。

4. 矢量图形

Tektronix公司生产了一系列流行的终端，它们采用了一种叫做矢量图形的显示方法。矢量图形是基于DEC的块模式图形方法设计的，它将所有的屏幕图像（包括字符）变成一系列的线段（矢量）。Tektronix 4010终端是生产的最流行的图形化哑终端。许多终端模拟包仍然会模拟它的功能。

4010终端通过使用电子束绘制一系列的矢量来显示图像，非常像用铅笔绘制。由于矢量图形不用点来创建线，它能用相比其他基于点的图形终端更高的精度来绘制几何形状。这是一个在数学家和科学家中流行的功能。

现代终端模拟器使用软件来模拟Tektronix 4010终端的矢量图形绘制功能。对于那些需要绘制高精度图形，或仍在运行使用矢量图形函数来绘制复杂图表的应用的人来说，这仍然是一个受欢迎的功能。

5. 显示缓冲

图形显示的一个关键要素是终端缓冲数据的能力。缓冲数据需要终端内部有额外的内存来存储当前未在显示器上显示的字符。

DEC VT系列终端使用了两种类型的数据缓冲：

- 在主显示窗口中翻屏时缓冲数据（该数据被称为历史）；
- 缓冲一个完全独立的显示窗口（称为备用屏幕）。

第一种类型的缓冲被称为滚动区域（scroll region）。滚动区域是终端拥有的内存数量，它使得终端能在翻屏时“记住”数据。标准的DEC VT102终端含有一个25行字符的观察区域。在终端显示一行新字符时，前面一行就滚上去了。当终端达到了显示的底行时，下一行会让顶行滚出显示。

VT102终端中的内存允许它保存最后滚出显示的64行。用户能够锁定当前显示器显示，用方向键向后翻出前面已经“滚出”显示的行。终端模拟包允许你用边上的滚动条或鼠标的滚动键来翻出保存的数据，而不用锁定显示。当然，为了保持模拟的完整兼容性，大多数终端模拟包也允许你锁定显示，用方向键或翻页键来翻出保存的数据。

第二种类型的缓冲被称为替代屏幕（alternative screen）。通常，终端会直接向显示器上的普通显示区域写入数据。人们开发了一种方法使用两个屏幕区域存储数据来粗糙地实现动画。控制码用来发信号给终端，向而不是当前显示屏幕写数据替代屏幕。那部分数据会保留在内存中。另一个控制码会发信号给终端，几乎立即在普通屏幕数据和替代屏幕数据之间切换显示器显示。通过将后续数据页存储在替代屏幕区域，然后再显示，你可以粗糙地模拟移动的图像。

模拟VT系列终端的终端模拟程序能够同时支持滚动区域和替代屏幕缓冲方法。

6. 色彩

即使是在黑白（或绿）世界的哑终端时代，程序员也在试验用不同的方法来呈现数据。大多

数终端支持特殊的控制码来生成下列类型的特殊文本：

- 加粗字符；
- 下划线字符；
- 图像反转（白底黑字）；
- 闪烁；
- 上面这些功能的组合。

在过去，如果你想引起别人的注意，你会用加粗、闪烁和图像反转文本。现在有些东西可能会刺激到你的眼睛了！

在彩色终端到来时，程序员们添加了特殊的控制码来显示各种颜色和形状的文本。ANSI字符集包括了一些控制码，它们用来指定显示器上显示的前端文本和背景色的颜色。大部分终端模拟器支持ANSI色彩控制码。

2.1.2 键盘

对终端来说，除了显示器如何操作之外，还有很多内容。如果你曾经用过不同类型的哑终端，你应该会发现键盘上通常含有与现在不同的键。对于终端模拟包来说，模拟特定哑终端上的特定键已经被证明不是件容易的事。

PC键盘的发明者不可能将哑终端上每种可能的特殊键类型都包含进来。一些PC制造商曾尝试过包含一些带特定功能的特殊键，但最终PC键盘的按键在某种程度上已被标准化了。

对于要完全模拟特定类型哑终端的终端模拟包来说，它必须重新映射PC键盘上没有的所有哑终端键。这种再映射的功能经常会叫人困惑，尤其是当不同系统对同一个键使用不同的控制码时。

以下是一些你会在终端模拟包中看到的常见特殊键。

- 中断（Break）——给主机发送一串0。它通常用来中断shell中当前正在执行的程序。
- 滚动锁定（Scroll Lock）——也叫“禁止滚动”（No Scroll），它会停止显示上的输出。有些终端含有存储显示内容的内存，从而用户可以在滚动锁定打开时翻出前面看过的信息。
- 重复（Repeat）——当按下这个键和其他键时，它会让终端反复地向主机发送另一个键的键值。
- 返回（Return）——通常用来向主机发送一个回车字符。大部分时候用来表明一个命令的结束，以便主机处理（现在在PC键盘上称为Enter键）。
- 删除（Delete）——虽然基本上是个简单功能，但删除键给终端模拟包造成了混乱。有些终端删除当前光标位置的字符，而其他的则会删除光标前面的字符。为了解决这个窘境，PC键盘上设置了两个删除键，退格键（Backspace）和删除键（Delete）。
- 方向键（Arrow Key）——通常用来将光标放到特定位置——例如，在滚动一个列表输出时。
- 功能键（Function Key）——专用键的组合，可以在程序中赋给特殊值，类似于PC的F1~F12键。DEC VT系列终端实际上有两组功能键，F1~F20和PF1~PF4。

键盘模拟是终端模拟包中极为关键的部分。很遗憾，应用经常写成需要用户点击特定键来执

行特定功能。我见过很多个通信包，它们使用过去DEC的 PF1 ~ PF4键，这些键通常很难在终端模拟键盘上找到。

2.2 terminfo 数据库

既然你知道了终端模拟包可以模拟不同类型的终端，你需要一个途径来让Linux系统知道你模拟的是具体哪个终端。Linux系统需要知道在和终端模拟器通信时使用哪些控制码。这是通过使用一个环境变量（参见第5章）和一组共称为terminfo数据库的特殊文件来实现的。

terminfo数据库是一组文件，这些文件标识了各种可以用在Linux系统上的终端的特性。Linux系统将每种终端类型的terminfo数据作为一个单独的文件存储在terminfo数据库目录。这个目录的位置经常随发行版的不同而不同。常见的位置有/usr/share/terminfo、/etc/terminfo和/lib/terminfo。

为了便于组织（通常有大量不同的terminfo文件），你会看到terminfo数据库目录含有针对不同字母的目录。特定终端的单独文件被存储在它们的终端名称对应的字母目录下。举个例子，在/usr/share/terminfo/v里是VT终端模拟器。

terminfo文件是个二进制文件，它是编译文本文件的结果。这个文本文件含有定义了屏幕功能的代码字，以及在终端上实现这个功能所需的控制码。

由于terminfo数据库文件是二进制的，你无法看到这些文件中的代码。不过，你可以用infocmp命令来将二进制条目转换成文本。使用这条命令的例子如下：

```
$ infocmp vt100
# Reconstructed via infocmp from file: /lib/terminfo/v/vt100
vt100|vt100|dec vt100 (w/advanced video).
am, msgr, xenl, xon,
cols#80, it#8, lines#24, vt#3,
acsc=``aaffggjjkkllmmnnoopprrssttuuvvwwxxyyzz{{||}}--,
bel=~G, blink=\E[5m<2>, bold=\E[1m<2>,
clear=\E[H\E[J<2>, cr=~M, csr=\E[1i%p1%d;%p2%dr,
cub=\E[%p1%dD, cubl=~H, cud=\E[%p1%D, cudl=~J,
cuf=\E[%p1%Dc, cufl=\E[0<2>,
cup=\E[%p1%D:p2%h<5>, cuu=\E[%p1%Da,
cuul=\E[A<2>, ed=\E[J<50>, el=\E[K<3>, et1=\E[1K<3>,
enacs=\E(B\E)D, home=\E[H, ht=~I, hts=\EH, ind=~J, kai=\EOq,
ka3=\EOs, kb2=\EOr, kbs=~H, kc1=\EOp, kc3=\EOn, kcub1=\EOd,
kcud1=\EOB, kcuf1=\EOC, kcui1=\EOA, kent=\EOM, kf0=\EOy,
kf1=\EOP, kf10=\EOx, kf2=\EOQ, kf3=\EOR, kf4=\EOS, kf5=\EOt,
kf6=\EOu, kf7=\EOv, kf8=\EOI, kf9=\EOw, rc=\E8,
rev=\E[7m<2>, ri=\EM<2>, rmacs=~0, rman=\E[?7?],
rmkx=\E[?21\ES, rmso=\E[m<2>, rmul=\E[m<2>,
rs2=\E[?31\ES, rs4=\E[?41\ES, rs5=\E[?51\ES, rs7=\E[?7h\EC[?8h,
sc=\E7,
sgr0=\E[m\017<2>, smacs=~N, sman=\E[?7h, smkx=\E[?1h\EC[?8h,
smso=\E[7m<2>, smul=\E[4m<2>, tbc=\E[3g.
```

terminfo条目定义了终端名（本例中是vt100），以及可以跟终端名关联起来的所有别名。注意第一行说明了提取这些值的terminfo文件的位置。

之后，infocmp命令列出了终端定义的功能，以及用来模拟每个功能的控制码。一些功能可以打开，也可以关闭。如果功能出现在这个列表中了，它就会由终端定义打开（比如am，即自动右侧边缘调整功能）。其他功能必须定义一个特定的控制码序列来执行该任务（比如清空显示器屏幕）。表2-1列出了你在vt100 terminfo定义文件中看到的一些功能。

表2-1 terminfo功能代码

代 码	描 述
am	设置右侧自动边缘调整
msgr	以突出模式安全移动光标
xenl	忽略80列后的换行符
xon	终端使用XON/XOFF字符进行流控制
cols#80	每行80列
it#8	制表符设为8个空格
lines#24	每屏显示24行
vt#3	虚拟终端数为3
bel	用来模拟响铃的控制码
blink	用来生成闪烁文本的控制码
bold	用来生成加粗文本的控制码
clear	用来清空屏幕的控制码
cr	用来输入回车符的控制码
csr	用来修改滚动区域的控制码
cub	向左移动一个字符，但不擦除其他字符
cubl	将光标同退一格
cud	将光标下移一行
cudl	将光标下移一行的控制码
cuf	向右移动一个字符，但不擦除其他字符
cuf1	向右移动一个字符但不擦除其他字符的控制码
cup	移动到显示器的第1行第2列位置的控制码
cuu	将光标向上移动一行
cuul	将光标向上移动一行的控制码
ed	清除到屏幕底部的内容
el	清除到行尾的内容
ell	清除到行首的内容
enacs	启用备用字符集
home	用来将光标移到起始位置——第1行第2列（同cup）的控制码
ht	制表符
hts	在每一行的当前位置设置制表符
ind	向上滚动文本
kal	小键盘的左上键
ka3	小键盘的右上键

(续)

代 码	描 述
kb2	小键盘的中心键
kbs	退格键
kcl	小键盘的左下键
kc3	小键盘的右下键
kcub1	左方向键
kcud1	向下键
kcuf1	右方向键
kcuu1	向上键
kent	Enter键
kf0	F0功能键
kf1	F1功能键
kf10	F10功能键
rc	将光标放回上次保存的位置
rev	反转图像模式
r1	向下滚动文本
rmacs	停用备用字符集
rmam	关闭自动边缘调整
rmkx	退出键盘发送模式
rms0	退出突出模式
rmul	退出下划线模式
rs2	重置
sc	保存当前光标位置
sgr	定义图像属性
sgr0	关闭所有属性
smacs	启用备用字符集
smam	打开自动边缘调整
smkx	启用键盘发送模式
sms0	启用突出模式
smul	启用下划线模式
tbc	清除所有制表位

Linux shell 使用TERM环境变量来定义对特定会话使用terminfo数据库中的哪个终端模拟设置。当TERM环境变量设为vt100时，shell就知道使用跟vt100 terminfo数据库条目关联的控制码来向终端模拟器发送控制码。要查看TERM环境变量，你可以在CLI中显示它：

```
$ echo $TERM
xterm
$
```

这个例子说明当前终端类型设成了terinfo数据库中的xterm条目。

2.3 Linux控制台

在Linux的早期，在启动系统时你只会在显示器上看到一个登录提示符，没别的了。如前面提到的，这就是Linux控制台。它是你可以为系统输入命令的唯一地方。

在现代Linux系统上，当Linux系统启动时它会自动创建几个虚拟控制台。虚拟控制台是运行在Linux系统内存中的一个终端会话。不用将几个哑终端连到PC上，大部分Linux发行版会启动7个（有时会更多）虚拟控制台。你可以只用一个PC键盘和显示器来访问它们。

在大多数Linux发行版中，你可以使用简单的按键组合来访问这些虚拟控制台。通常你必须按下Ctrl+Alt组合键，然后按一个功能键（F1 ~ F8）来进入你要使用的虚拟控制台。功能键F1生成虚拟控制台1，F2键生成虚拟控制台2，依次类推。

虚拟控制台中的6个都使用全屏文本终端模拟器来显示文本登录界面，如图2-2所示。



图2-2 Linux控制台登录界面

在用用户ID和密码登录后，你会被带到Linux bash shell CLI。在Linux控制台中，你不能运行任何图形化程序。你只能使用文本程序来在Linux文本控制台上进行显示。

登录到虚拟控制台上后，你可以让它保持活动并切换到另外一个虚拟控制台上而不会丢失活动的会话。你可以在所有的虚拟控制台之间切换，运行多个活动会话。

前两个或最后两个虚拟控制台通常为X Window图形化桌面保留。有的发行版只会分配一个，

所以你可能需要测试所有三个Ctrl+Alt+F1、Ctrl+Alt+F7和Ctrl+Alt+F8，来看看你的发行版使用的是哪个。大部分发行版会在开机顺序完成后自动切换到一个图形化虚拟控制台，提供了完整的图形化登录和桌面体验。

先登录到文本虚拟终端会话、然后再切换到一个图形化的会话会比较麻烦。幸运的是，在Linux系统上有更好的办法来在图形化模式和文本模式之间切换：终端模拟包是从图形化桌面会话访问shell CLI的一个流行方法。下面几节将会介绍在图形化窗口中提供终端模拟的最常用的软件包。

2.4 xterm 终端

最早的也是最基本的X Window终端模拟包是xterm。xterm包自从有了X Window之时起就有了，默认包含在大多数X Window包中。

xterm包提供了一个基本的VT102/220终端模拟CLI和一个图形化Tektronix 4014环境（类似于4010环境）。虽然xterm是一个完整的终端模拟包，但它并不需要额外的资源（比如内存）来运行。鉴于这点，在设计来在较早硬件平台上运行的Linux发行版中，xterm包仍然流行。一些图形化桌面环境，比如fluxbox，将它用作默认的终端模拟包。

虽然并未提供太多好用的功能，但xterm包把一件事做到了极致，那就是模拟VT220终端。新版本的xterm甚至可以模拟支持色彩控制码的VT系列，允许你在脚本中使用色彩。

图2-3展示了运行在图形化Linux桌面上的基本xterm显示。

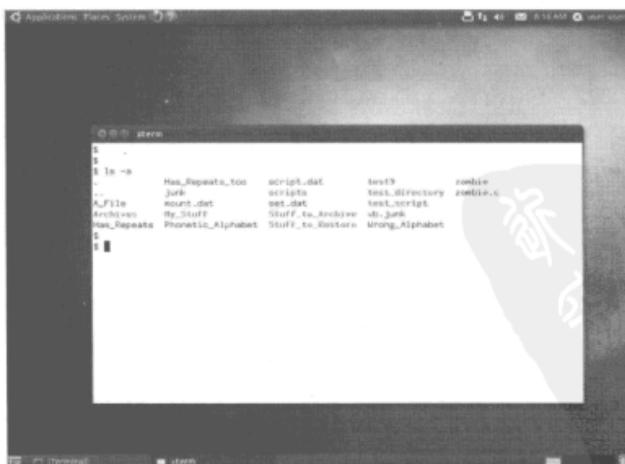


图2-3 基本xterm显示

xterm包允许你用命令行参数和4个简单的图形化菜单来设置每个功能。下面几节将会讨论这些功能以及如何修改它们。

2.4.1 命令行参数

xterm的命令行参数列表真是包罗万象。你可以控制很多功能来定制终端模拟的功能，比如启用或禁用某个VT模拟功能。

xterm的命令行参数使用加号和减号来说明功能是怎样设置的。加号表明这个功能将会恢复为默认设置，减号表明你正在将该功能设为非默认值。表2-2列出了一些你可以用命令行参数设置的较常见功能。

表2-2 xterm命令行参数

参 数	描 述
132	默认情况下，xterm不支持一行显示132个字符
ah	总是高亮显示文本光标
aw	启用自动换行
bc	启用文本光标闪烁
bg color	指定用作背景的颜色
cm	禁止识别ANSI色彩改变控制码
fb font	指定给加粗文本用的字体
fg color	指定给前端文本用的颜色
fn font	指定给文本用的字体
fw font	指定给宽文本用的字体
hc color	指定给高亮文本用的颜色
j	使用跳跃式滚动，一次滚动多行
l	启用记录屏幕数据功能，将数据记录到一个日志文件
lf filename	指定用来记录屏幕数据的文件名
mb	当光标到达行尾时，响一下边缘响铃
ms color	指定给文本光标用的颜色
name name	指定出现在标题栏上的应用名称
rv	通过交换背景和前端的颜色启用图像反转
sb	使用边滚动条来允许滚动已保存的滚动数据
t	以Tektronix模式启动xterm
tb	指定xterm应该在顶部显示一个工具栏

重要的是要注意，并非所有的xterm实现都支持以上这些命令行参数。你可以在系统上启动xterm时用-help参数来确定你的xterm实现了哪些参数。

2.4.2 xterm主菜单

xterm主菜单包含作用在VT102和Tektronix窗口上的配置选项。你可以在xterm会话窗口中通过按下Ctrl键并单击鼠标键（右手鼠标上的左键，左手鼠标的右键）来打开主菜单。图2-4为xterm主菜单。

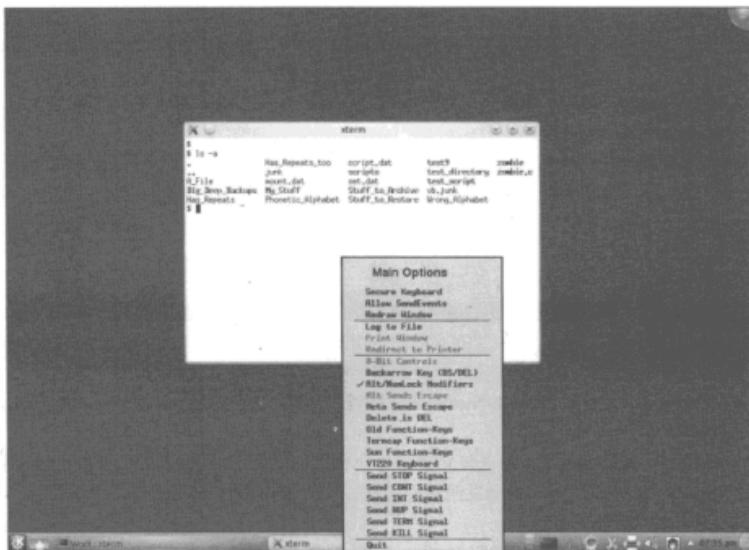


图2-4 xterm主菜单

在xterm主菜单中有4个部分，我们将在后面几节中进一步介绍。

1. X事件命令

X事件命令部分包含允许你管理xterm如何与X Window显示交互的功能。

- ❑ **Toolbar** (工具条) ——如果xterm安装支持工具条，那么这个条目会使能或禁止在xterm窗口中显示工具条(同tb命令行参数)。
- ❑ **Secure Keyboard** (安全键盘) ——将键盘按键限定到特定xterm窗口。这在键入密码时很有用，保证它们不会被另外一个窗口“劫走”。
- ❑ **Allow SendEvents** (允许发送事件) ——允许这个xterm窗口接受其他X Window应用产生的X Window事件。
- ❑ **Redraw Window** (重绘窗口) ——指示X Window刷新xterm窗口。

再次说明，你的特定xterm实现未必支持所有这些功能。如果它们不支持，它们会在菜单中变灰显示。

2. 输出捕捉

xterm包允许你捕捉显示在窗口中的数据，要么将它记录到一个文件中，要么将它发给X Window定义的默认打印机。在这部分出现的功能有以下几项。

- Log to File（记录到文件）——将xterm窗口中显示的所有数据都发给一个文本文件。
- Print Window（打印窗口）——将当前窗口中显示的所有数据都发给默认X Window打印机。
- Redirect to Printer（重定向到打印机）——也是将xterm窗口中显示的所有数据发给默认X Window打印机。要停止打印数据，必须关闭这个功能。

如果你正在显示区域使用图形字符或控制字符（比如彩色文本），那么捕捉功能可能会乱成一团。所有发送给屏幕的字符（包括控制字符）都会保存在日志文件中或发送给打印机。

xterm打印功能会假定你在X Window系统上定义了一个默认打印机，如果你没有，那么这个功能将会在菜单中变灰显示。

3. 键盘设置

键盘设置部分含有允许你定制xterm如何向主机系统发送键盘字符的功能。

- 8-bit Controls（8位控制）：发送VT220终端中使用的8位控制码，而不是7位ASCII控制码。
- Back Arrow Key（后退方向键）：在退格字符和删除字符之间切换后退方向键。
- Alt/Numlock Modifiers（Alt/Numlock修饰符）：控制Alt键或Numlock键是否改变PC数字键盘的行为。
- Alt Sends Escape（Alt发送转义字符）：Alt键发送转义字符控制码以及按下的其他键。
- Meta Sends Escape（Meta发送转义字符）：控制功能键是否发送一个双字符控制码，包括转义控制码。
- Delete is DEL（删除键是DEL功能）：PC的删除键发送一个删除字符而不是退格字符。
- Old Function Keys（旧功能键）：PC的功能键会模拟DEC VT100的功能键。
- Termcap Function Keys（Termcap功能键）：PC的功能键会模拟Berkley Unix 的Termcap功能键。
- Sun Function Keys（Sun功能键）：PC的功能键会模拟Sun工作站的功能键。
- VT220 Keyboard（VT220键盘）：PC的功能键会模拟DEC VT220的功能键。

如你所知，设置键盘偏好通常取决于特定应用和你的工作环境。它还包括相当数量的个人偏好。通常，这就是一个采用什么样的设置能让它为你最好地工作的问题。

2.4.3 VT选项菜单

VT选项菜单会设置xterm在VT102模拟中使用的功能。你可以通过按下控制键（Ctrl）并单击第二个鼠标键来打开VT选项菜单。通常，第二个鼠标键是鼠标中键。如果你使用的是双键鼠标，

大多数Linux X Window配置会在你同时单击鼠标左键和右键时模拟鼠标中键。图2-5展示了VT选项菜单的样子。



图2-5 xterm VT选项菜单

如你在图2-5中所能看到的，许多你从命令行参数中设置的VT功能也可以从VT选项菜单中设置。这会生成一个很大的可用选项列表。VT选项被分成三组命令，下面将一一介绍。

1. VT功能

VT功能部分会命令修改xterm如何实现VT102/220模拟的功能。它们包括：

- Enable Scrollbar (显示滚动条);
- Enable Jump Scrollbar (显示跳跃式滚动条);
- Enable Reverse Video (开启图像反转);
- Enable Auto Wraparound (开启自动回绕);
- Enable Reverse Wraparound (开启逆向自动回绕);
- Enable Auto Linefeed (开启自动换行);
- Enable Application Cursor Keys (使能应用程序的光标键);
- Enable Application Keypad (使能应用程序的小键盘);
- Scroll to Bottom on Keypress (按键时滚动到底部);

- Scroll to Bottom on TTY Output (有TTY输出时滚动到底部);
- Allow 80/132 Column Switching (允许80/132列切换);
- Select to Clipboard (选择到剪贴板);
- Enable Visual Bell (开启可视响铃);
- Enable Bell Urgency (开启紧急窗口提示);
- Enable Pop on Bell (开启前置窗口铃声);
- Enable Blinking Cursor (开启闪烁光标);
- Enable Alternate Screen Switching (开启备用屏幕切换);
- Enable Active Icon (开启活动图标)。

你可以通过单击菜单中的选项来开启或关闭这些功能。开启的功能前面会有一个对号。

2. VT命令

VT命令部分会向xterm的模拟窗口发送特定的重置命令。它们包括：

- Do Soft Reset (执行软重置);
- Do Full Reset (执行全重置);
- Reset and Clear Saved Lines (重置并清除保存的行)。

软重置会发送一个控制码来重置屏幕区域。在程序未能正确设置滚动区域时，这个会很方便。全重置会清空屏幕，重置全部的设置制表符，并将在会话中设置的全部终端模式功能重置为初始状态。“重置并清除保存的行”命令会执行一个全重置，并且会清空滚动区域的历史文件。

3. 当前屏幕命令

当前屏幕命令部分会向xterm模拟器发送命令，影响当前活动屏幕。

- Show Tek Window (显示Tek窗口): 显示Tektronix终端窗口以及VT100终端窗口。
- Switch to Tek Window (切换到Tek窗口): 隐藏VT100终端窗口并显示Tektronix终端窗口。
- Hide VT Window (隐藏VT窗口): 显示Tektronix终端窗口的同时隐藏VT100终端窗口。
- Show Alternate Screen (显示备用屏幕): 显示当前存储在VT100备用屏幕区域中的数据。

xterm终端模拟器允许以VT100终端模式（默认情况下）或Tektronix终端模式（用t命令行参数）启动。在你用任意一种模式启动后，你可以用这个菜单区域在会话中切换到另外一种模式。

2.4.4 VT字体菜单

VT字体菜单会设置在VT100/220模拟窗口中用到的字体风格。你可以通过按下控制键（Ctrl）并单击第三个鼠标键（右手鼠标的右键，左手鼠标的左键）来打开这个菜单。图2-6给出了VT字体菜单。

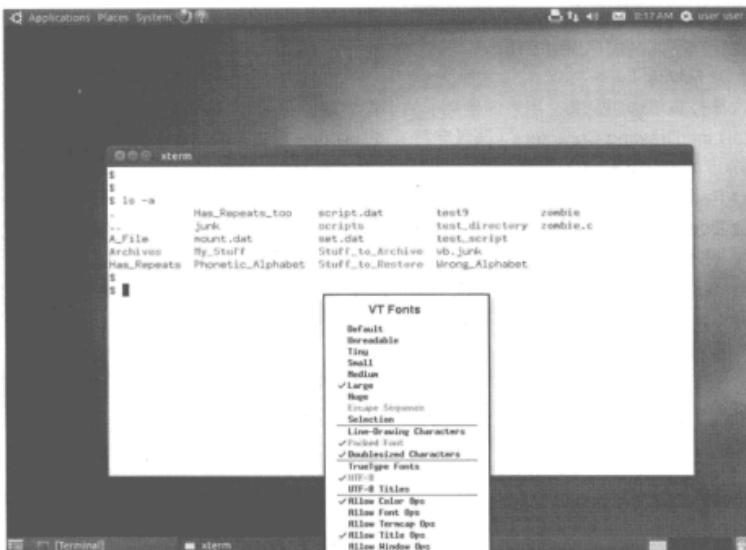


图2-6 xterm的VT字体菜单

VT字体菜单含有三部分选项，下面就来详细介绍。

1. 设置字体

这些菜单选项会设置在xterm窗口中使用的字体大小。可选的值有：

- Default (默认值);
- Unreadable (不可读);
- Tiny (非常小);
- Small (小);
- Medium (中等);
- Large (大);
- Huge (巨大);
- Escape Sequence (转义序列);
- Selection (选择)。

默认字体是在当前X Window框中显示文本采用的标准大小。不可读字体名副其实，它会将xterm窗口缩小到一个实际上没法用的大小。但当你想最小化窗口到桌面上而不想将它最小化到系统上时，这很方便。大字体和巨大字体选项会为视力弱的用户生成非常大的字体。

转义序列选项会将字体设成VT100设置字体控制码设置的最后一个字体。选择选项则允许你用一个特殊字体名保存当前字体。

2. 显示字体

这部分菜单选项定义了用来创建文本的字符类型。有以下3个可用选项。

- Line Drawing Characters** (画线字符): 告诉Linux系统生成ANSI图形化线而不是使用所选字体的线字符。
- Packed Font** (压缩字体): 告诉Linux系统使用压缩字体。
- Doublesized Characters** (双倍大小字符): 告诉Linux系统将设定的字体放大到正常大小的两倍。

画线字符允许你确定在文本模式下绘制图形时使用那种类型的图形化功能。你可以使用选定字体源中提供的字符，或DEC VT100控制码提供的字符。

3. 指定字体

这部分菜单提供了选项来选择用什么类型的字体来创建字符：

- TrueType Fonts** (TrueType字体);
- UTF-8 Fonts** (UTF-8字体);
- UTF-8 Titles** (UTF-8标题)。

TrueType字体在图形化环境中很流行。每个字符在行中不是占用等量的空间，而是跟它们的自然大小成比例。因此，字母i比字母m在行中占用更少的空间。UTF-8字体允许你为不支持外文字符^①的应用临时切换到使用Unicode字符集。标题选项允许xterm窗口的标题用UTF-8编码。

2.5 Konsole 终端

KDE桌面项目创建了它自己的终端模拟包，称为Konsole。Konsole包不仅集成了基本的xterm功能，而且还有一些我们所期望的在Windows应用中才有的高级功能。本节将会介绍Konsole终端的功能，并演示如何使用它们。

2.5.1 命令行参数

通常Linux发行版会提供一个方法来直接从图形化桌面的菜单系统启动应用。如果你的发行版不支持这个功能，你可以按如下格式来手动启动Konsole：

```
konsole parameters
```

跟xterm一样，Konsole包使用命令行参数来设置新会话中的功能。表2-3列出了可用的Konsole命令行参数。

^① 显然，这里说的外文字符是指非英文字符，比如希伯来文、汉字等。——译者注

表2-3 Konsole命令行参数

参数	描述
-e command	执行命令而不是shell
--keytab file	使用指定的密钥文件来定义密钥映射
--keytabs	列出所有可用的密钥表
--ls	用登录屏幕来启动Konsole会话
--name name	设置出现在Konsole标题栏上的名字
--noclose	在最后一个会话已经关闭时，禁止Konsole窗口关闭
--noframe	启动不带边框的Konsole
--nohist	禁止Konsole在会话中保存滚动历史
--nomenubar	启动不带标准菜单栏选项的Konsole
--noresize	禁止调整Konsole窗口区域的大小
--notabbar	启动不带标准会话标签区域的Konsole
--noxft	启动不支持较小字体锯齿化的Konsole
--profile file	使用保存在指定文件中的设置启动Konsole
profiles	列出所有可用的Konsole配置文件
--schema name	使用指定的模式名或文件启动Konsole
--schemata	列出Konsole中可用的模式
-T title	设置Konsole窗口标题
--type type	用指定的类型启动Konsole会话
--types	列出所有可用的Konsole会话类型
--vt_sz CxL	指定终端的列 (C) 和行 (L)
--workdir dir	指定Konsole存放临时文件的工作目录

2.5.2 标签式窗口会话

启动Konsole时，你会注意到它有一个标签式窗口，其中一个标签在终端模拟会话中打开。这个是默认的标签式窗口会话，它通常是标准bash shell CLI。Konsole允许同时有几个标签处于活动状态。标签位于窗口区域的顶部或底部，允许你方便地切换会话。对于需要在标签中编辑代码，同时在另一个标签式窗口中测试代码来说，这个功能非常有用。你可以在Konsole中不同活动标签间来回切换。图2-7展示了有3个活动标签的Konsole窗口。

类似于xterm的终端模拟器，如果你右键单击活动标签区域，Konsole会弹出一个简单菜单，这个菜单有如下选项。

- Copy (复制)**: 将选定文本复制到剪贴板。
- Paste (粘贴)**: 将剪贴板中的内容粘贴到选定区域。
- Clear Scrollback & Reset (清除回滚并重置)**: 清空当前标签中的所有文本并重置终端。
- Open File Manager (打开文件管理器)**: 在当前工作目录中，打开KDE的默认文件管理器Dolphin。

- Change Profile (修改配置文件): 修改当前标签的配置文件。
- Edit Current Profile (编辑当前配置文件): 编辑当前标签的配置文件。
- Show Menu Bar (显示菜单栏): 打开/关闭菜单栏显示。
- Character Encoding (字符编码): 选择用来发送和显示字符的字符集。
- Close Tab (关闭标签): 终止标签化窗口会话。如果关闭Konsole窗口里的最后一个标签, 则Konsole会关闭。

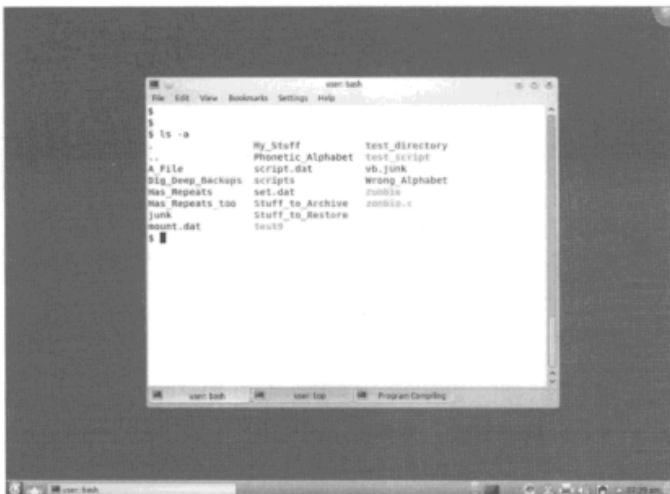


图2-7 有3个活动会话的Konsole终端模拟器

Konsole还提供了另外一个访问新标签菜单的快捷方法——按下Ctrl键并右键单击标签区域。在修改标签化窗口后, 你可以使用配置文件来保存修改以便将来使用。

2.5.3 配置文件

Konsole提供了一个称为配置文件 (Profile) 的强大方法来保存和重用标签会话的设置。当你第一次启动Konsole时, 这个标签会话设置会从默认的配置文件Shell中提取。这些设置包括使用什么shell、什么配色方案等选项。修改了当前标签会话后, 你可以将这些修改保存为一个新的配置文件。这个功能允许设置多个标签, 比如标签会话使用非bash shell。

配置文件还能用来自动化普通任务, 比如登录到另外一个系统。你可以定义许多配置文件并在每个打开的标签会话中使用不同的配置文件。要创建新的配置文件, 可以使用前面在简单的

Konsole菜单中介绍的Edit Current Profile设置。要从当前配置文件切换到另外一个配置文件，可以使用简单的菜单选项Change Profile。这些选项都在菜单栏上提供了。

默认情况下，Konsole使用菜单栏来提供额外的功能，这样你就能修改并保存你的Konsole标签和配置文件了。

2.5.4 菜单栏

默认的Konsole安装使用了菜单栏来方便你查看和修改标签中的选项和功能。菜单栏由6个选项组成，我们将在后面几小节中介绍。

1. File

File（文件）菜单栏选项提供了一个地方来选择在当前窗口还是新窗口中启动新标签。它含有以下选项。

- New Tab（新标签）：使用默认配置文件Shell来在当前终端窗口中启动新Konsole标签。
- New Window（新窗口）：启动一个新终端窗口来容纳新Konsole标签。
- List of Defined Profiles（已定义配置列表）：在当前标签会话中切换到一个新配置文件。
- Open File Manager（打开文件管理器）：在当前工作目录中打开文件管理器。
- Close Tab（关闭标签）：关闭当前标签。
- Quit（退出）：退出Konsole应用程序。

当你第一次运行Konsole时，List of Defined Profiles中唯一列出的配置文件会是Shell。随着越来越多的配置文件被创建和保存，它们的名字也会出现在列表中。

2. Edit

Edit（编辑）菜单栏提供了处理会话中文本的选项以及其他一些选项。

- Copy（复制）：将选定的文本复制到剪贴板。
- Paste（粘贴）：将当前系统剪贴板中的文本粘贴到当前光标位置。如果文本含有换行符，它们会被shell处理。
- Rename Tab（重命名标签）：修改当前标签名。除了文本，还可以使用下面的代号：
 - %#——会话号；
 - %D——当前目录（绝对路径名）；
 - %d——当前目录（相对路径名）；
 - %n——程序名；
 - %u——用户名；
 - %w——Shell来设置窗口标题。
- Copy Input To（复制输入至）：将当前标签中键入的文本发送给当前终端窗口中的一个或多个标签。
- ZModem Upload（ZModem上载）：使用ZModem协议将一个文件上载到系统。
- Clear and Reset（清除并重置）：发送控制码来重置终端模拟器，并清空当前会话窗口。

Konsole提供了一个绝妙的方法来追踪标签的功能。使用Rename Tab菜单选项，你就能命名一个标签来匹配它的配置文件。这有助于追踪哪个打开的标签正在执行什么功能。

3. View

View（视图）菜单栏选项含有控制Konsole窗口中单个标签会话的选项。

Split View（分割视图）。控制当前终端模拟窗口中的显示。视图可以被以下选项修改。

- Split View Left/Right（左右分割视图）：将当前显示并排分割为两个对等屏幕。
- Split View Top/Bottom（上下分割视图）：将当前显示上下分割为两个对等屏幕。
- Close Active（关闭活动屏幕）：将当前分割终端窗口合并为单个窗口。
- Close Others（关闭其他屏幕）：将非当前分割终端窗口合并为单个窗口。
- Expand View（扩大视图）：调整终端窗口活动那部分，以占据更多显示窗口。
- Shrink View（缩减视图）：调整终端窗口活动那部分，以占据更少显示窗口。

Detach View（分离视图）。从Konsole窗口中删除当前标签，并用当前标签启动一个新的Konsole窗口。只有在打开多个活动标签时才有。

Show Menu Bar（显示菜单栏）。打开/关闭菜单栏的显示。

Full Screen Mode（全屏模式）。打开/关闭终端窗口填满整个显示器显示区域。

Monitor for Silence（监测静默期）。当有10秒没有新文本出现在标签中时，打开/关闭特殊的图标显示。这允许在等待某个应用程序的输出停止时切换到其他标签，比如在编译大型应用程序时。

Monitor for Activity（监测活动期）。当有新文本出现在标签中时，打开/关闭特殊的图标显示。这允许在等待一个应用程序的输出时切换到其他标签。

Character Encoding（字符编码）。选择用来发送和显示字符的字符集。

Increase Text Size（增加文本大小）。增加文本字体的大小。

Decrease Text Size（减小文本大小）。减小文本字体的大小。

Konsole中的Split View选项会保持分割视图中当前打开的标签数。举个例子，如果你在终端窗口中有3个标签并分割了视图，那么每个视图都会有3个标签。

4. Scrollback

Konsole为标签保留了一个历史区域，正式称为回滚缓冲（scrollback buffer）。历史区域含有输出的文本中已经滚出终端模拟器可视区域的行。默认情况下，回滚缓冲保留最近的1000行输出。Scrollback菜单提供了回看缓冲的各种选项。

Search Output（查找输出）。在当前标签的底部打开一个对话框。Find对话框使得Konsole可以在回滚缓冲中查找特定文本。它还有针对大小写、正则表达式以及查找方向的选项。

Find Next（查找下一个）。在最近的回滚缓冲历史中查找下一个匹配的文本。

Find Previous（查找上一个）。在较早的回滚缓冲历史中查找下一个匹配的文本。

Save Output（保存输出）。将回滚缓冲的内容保存到一个文本文件或HTML文件。

Scrollback Options（回滚选项）。控制回滚缓冲的活动。有以下几个可用的修改选项。

- No Scrollback（禁止回滚）。禁用回滚缓冲。

- Fixed Scrollback (固定长度回滚)。设置回滚缓冲大小 (行数)。默认为1000行。
- Unlimited Scrollback (无限长回滚)。允许无数行存储到回滚缓冲中。
- Save to Current Profile (保存到当前配置文件)。将回滚缓冲选项保存到当前配置文件设置中。

Clear Scrollback & Reset (清除回滚并重置)。删除回滚缓冲的内容并重置终端窗口。

可以用可视区域中的滚动条滚动回滚缓冲或按下Shift键和向上方向键来一行一行回滚、或向上翻页键一页一页 (24行) 回滚。

5. Bookmarks

Bookmarks (书签) 菜单选项提供了在Konsole窗口中管理书签的一个途径。你可以使用书签保存活动会话中的目录位置，然后方便地在同一个会话或新会话中返回那里。你是否经历过顺次打开几层目录来查找Linux系统上的一些东西，退出，然后却忘了是怎样到达那里的？书签可以解决这个问题。当你到了需要的目录位置，添加一个新书签。当你要返回时，在Bookmarks中找到你的新书签，然后它会自动将目录切换到你要的位置。书签选项包括以下几项。

- Add Bookmark** (添加书签)：在当前目录位置创建新书签。
- Bookmark Tabs as Folder** (标记标签为文件夹)：为当前终端窗口所有标签创建一个书签。
- New Bookmark Folder** (新建书签文件夹)：为书签创建一个新的存储文件夹。
- Edit Bookmarks** (编辑书签)：编辑已有的书签。
- 你的书签列表**：所有你创建的书签。

你可以在Konsole中保存任意多个书签，但书签太多可能会容易让人引起混淆。默认情况下，它们都出现在Bookmarks区域的同一级中。你可以创建新的书签文件夹，使用Edit Bookmarks选项将单个书签移动到新文件夹中，来管理书签。

6. Settings

Settings菜单栏区域允许你定制和管理你的配置文件以及给当前标签会话添加一些功能。这个区域包括以下几项。

- Change Profile** (修改配置文件)：将一个选定的配置文件应用到当前标签。
- Edit Current Profile** (编辑当前配置文件)：打开一个对话框，其中有大量配置文件设置可以修改。
- Manage Profiles** (管理配置文件)：允许特定配置文件作为默认配置文件，并使得你可以创建和删除配置文件。还可以管理配置文件出现在File菜单中的顺序。
- Configure Shortcuts** (配置快捷键)：创建Konsole命令的键盘快捷方式。
- Configure Notifications** (配置提醒)：为特定会话事件设置动作。
- Configure Konsole** (配置Konsole)：创建特定Konsole模式和会话。

Configure Notifications区域非常好用。它允许你将会话中可能出现的5种特定事件关联到6个不同的动作上。当其中某个事件发生时，定义好的动作就可以被执行了。

Edit Current Profile设置是一个强大的工具，它提供了对配置文件功能的一些高级控制。这个

对话框为创建和保存各种配置文件以便以后使用提供了一个途径。图2-8展示了Edit Current Profile的主对话框。

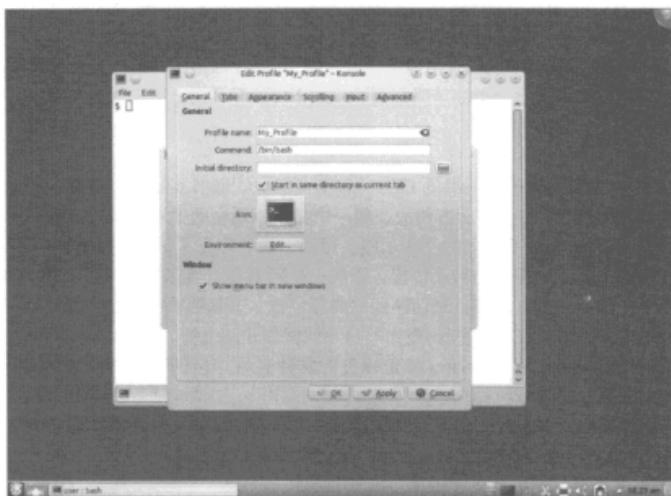


图2-8 Konsole的Edit Current Profile对话框

在Edit Current Profile对话框中有6个标签项。

- General (常规): 允许你设置配置文件的名称、图标、初始文件系统目录等。你也可以指定在打开标签时执行的命令。它通常指向bash shell、/bin/bash，但也可以是经常使用的shell命令，比如top。
- Tabs (标签): 定义标签的标题格式和标签栏的位置。
- Appearance (外观): 如标签的配色方案和字体设置一类的选项都在这个窗口中。
- Scrolling (回滚): 包括的设置有回滚缓冲大小和回滚条在窗口中的位置。
- Input (输入): 可以在这里设置键绑定、当特定键盘组合按下时将哪些字符发送到终端模拟。
- Advanced (高级): 允许你在这个窗口中配置一些设置，包括终端功能、字符编码、鼠标交互和光标功能。

7. Help

Help(帮助)菜单选项提供了完整的Konsole手册(如果Linux发行版中安装了KDE手册的话)，“今日提示”功能可以在每次启动Konsole时显示一些有意思的鲜为人知的快捷方式和技巧，此外还有标准的About Konsole对话框。

2.6 GNOME Terminal

如你所料, GNOME桌面项目也有它自己的终端模拟程序。GNOME Terminal软件包有许多跟Konsole和xterm相同的功能。本节将会带你逐步了解配置和使用GNOME Terminal的各个部分。

2.6.1 命令行参数

GNOME Terminal应用程序也提供了大量的命令行参数来允许你在启动它时控制GNOME的行为。表2-4列出了可用的参数。

表2-4 GNOME Terminal命令行参数

参 数	描 述
-e command	在默认终端窗口中执行参数
-x	在默认终端窗口中执行这个参数之后命令行的整个内容
--window	用默认终端窗口打开一个新窗口, 你可以加多个--window参数来启动多个窗口
--window-with-profile=	用指定配置文件打开新窗口。你还可以在命令行多次添加这个参数
--tab	在最近打开的终端窗口中打开一个新的标签化终端
--tab-with-profile=	用指定配置文件在最近打开的终端窗口中打开一个新的标签化终端
--role=	设置最后指定的窗口的角色
--show-menubar	在终端窗口的顶部启用菜单栏
--hide-menubar	在终端窗口的顶部禁用菜单栏
--full-screen	完全最大化显示这个终端窗口
--geometry=	指定X Window的几何形状参数
--disable-factory	不要用激活名称服务器来注册
--use-factory	用激活名称服务器来注册
--startup-id=	设置Linux启动通知协议的ID
-t, --title=	为终端窗口设置窗口标题
--working-directory=	为终端窗口设置默认工作目录
--zoom=	设置终端的放大倍数
--active	将最后指定的终端标签设为活动标签

GNOME Terminal的命令行参数允许你在GNOME Terminal启动时自动设置大量的功能。但你也可以在它启动后在GNOME Terminal窗口中设置其中大部分功能。

2.6.2 标签

类似于Konsole, GNOME Terminal将每个会话称为标签, 它还用标签来跟窗口中运行的多个会话保持联系。图2-9展示了一个有3个活动会话的GNOME Terminal窗口。

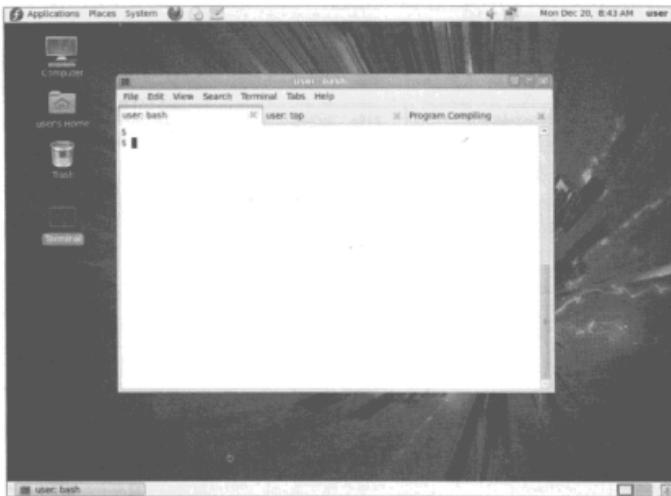


图2-9 有3个活动会话的GNOME Terminal

你可以在标签窗口中单击右键来查看快速菜单。这个快速菜单为你提供了一些可在标签会话中使用的动作。

- Open Terminal**（打开终端）：用默认标签会话打开一个新的GNOME Terminal窗口。
- Open Tab**（打开标签）：在已有GNOME Terminal窗口中打开一个新的会话标签。
- Close Tab or Close Window**（关闭标签或关闭窗口）：如果打开了多个标签，则会显示菜单选项Close Tab并会关闭当前会话标签。如果只打开了一个标签，它会显示菜单选项Close Window并且会关闭GNOME Terminal窗口。
- Copy**（复制）：将当前会话标签中的高亮文本复制到剪贴板。
- Paste**（粘贴）：将剪贴板中的数据粘贴到当前会话标签的当前光标位置。
- Profiles**（配置文件）：修改当前会话标签的配置文件或编辑当前标签的配置文件。
- Show Menubar**（显示菜单栏）：打开/关闭菜单栏显示。
- Input Methods**（输入方法）：允许你将当前输入方法改成另外一个字符转换程序或将它彻底关掉。

快速菜单提供了轻松访问终端窗口中标准菜单栏上含有的常用动作的方法。

2.6.3 菜单栏

GNOME Terminal的主要操作是在菜单栏中完成的。菜单栏含有让GNOME Terminal按你想要

的方式工作的所有配置和定制选项。下面几节将会介绍菜单栏中的不同选项。

1. File

File菜单栏选项含有创建和管理终端标签的选项。

Open Terminal (打开终端): 在新GNOME Terminal窗口中启动一个新shell会话。

Open Tab (打开标签): 在已有的GNOME Terminal窗口的新标签中启动一个新shell会话。

New Profile (新建配置文件): 允许你定制标签会话并将它保存为配置文件, 供你在后面调用。

Save Contents (保存内容): 将回滚缓冲的内容保存到一个文本文件中。

Close Tab (关闭标签): 关闭窗口中的当前标签。

Close Window (关闭窗口): 关闭当前GNOME Terminal会话, 关闭所有活动的会话。

File菜单中的大多数选项也能通过在标签区域中单击右键来访问。New Profile选项允许你定制会话选项设置并将它们保存, 供以后使用。

New Profile首先会要求你指定新配置文件的名称, 然后它会产生一个Editing Profile对话框, 如图2-10所示。

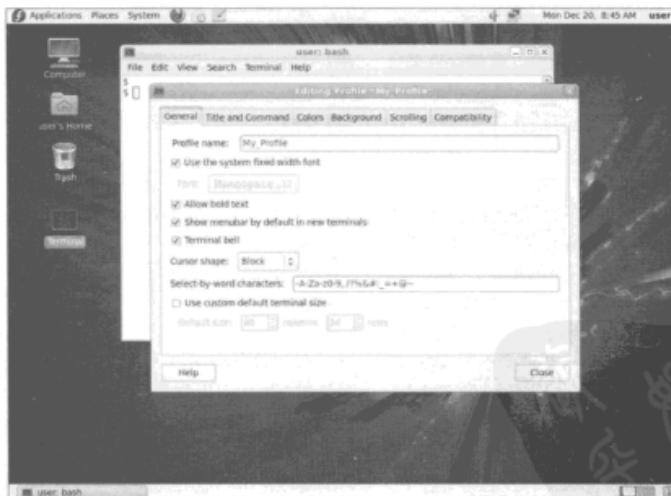


图2-10 GNOME Terminal的Editing Profile对话框

这里你可以为会话设置终端模拟功能。它包含6个区域。

General (常规): 提供常规设置, 比如字体、铃声和菜单栏。

Title and Command (标题和命令): 允许你为会话标签设置标题(显示在标签上)并确

定会话是以一个特殊命令而不是shell启动。

- Colors (色彩)**: 设置会话标签中使用的前景和背景颜色。
- Background (背景)**: 允许你为会话标签设置一张背景图片，或者将它设为透明的，这样你就能透过会话标签看到桌面。
- Scrolling (回滚)**: 控制是否创建回滚区域以及多大。
- Compatibility (兼容性)**: 允许你设置退格键和删除键向系统发送哪个控制码。

设置好配置文件后，你就能在打开新会话标签时指定它了。

2. Edit

Edit菜单选项含有处理标签中文本的选项。你可以用鼠标在标签窗口中任意地方复制和粘贴文本。它允许你方便地将命令行输出中的文本复制到剪贴板，然后再将它导入编辑器。你也可以将其他GNOME应用程序的文本粘贴到这个标签会话中。

- Copy (复制)**: 将选定文本复制到GNOME剪贴板。
- Paste (粘贴)**: 将GNOME剪贴板中的文本粘贴到这个标签会话中。
- Select All (全选)**: 选择整个回滚缓冲中的输出。
- Profiles (配置)**: 添加、删除或修改GNOME Terminal中的配置文件。
- Keyboard Shortcuts (键盘快捷键)**: 创建键组合来快速访问GNOME Terminal的功能。
- Profile Preference (配置文件偏好)**: 提供一个快捷方式来编辑当前会话标签使用的配置文件。

配置文件编辑功能是极其强大的工具，它可以用来定制若干配置文件并在切换会话时切换配置文件。

3. View

View菜单选项含有控制会话选项卡窗口如何显示的选项。它们包括以下选项。

- Show Menubar (显示菜单栏)**: 打开/关闭菜单栏显示。
- Full Screen (全屏显示)**: 将GNOME Terminal窗口放大到整个桌面大小。
- Zoom In (放大)**: 放大选项卡窗口中的字体。
- Zoom Out (缩小)**: 缩小选项卡窗口中的字体。
- Normal Size (普通大小)**: 将选项卡字体恢复到默认大小。

如果你隐藏了菜单栏，你可以通过右键单击任何会话选项卡打开Show Menubar选项来轻松地恢复它。

4. Terminal

Terminal菜单选项含有控制这个选项卡会话的终端模拟功能的选项。它们包括以下选项。

- Change Profile (改变配置文件)**: 允许你在会话选项卡中切换到另一个设置好的配置文件。
- Set Title (设置标题)**: 设置会话选项卡上的标题来方便识别它。
- Set Character Encoding (设置字符编码)**: 选择用于发送和显示字符的字符集。
- Reset (重置)**: 向Linux系统发送重置控制码。
- Reset and Clear (重启和清除)**: 向Linux系统发送重置控制码并清除当前在选项卡区域

显示的所有文本。

- **Window Size List (窗口尺寸列表)**: 列出当前GNOME Terminal窗口可以调整的不同尺寸。选择一个尺寸，窗口会自动调整它的大小。

字符编码提供了许多可供选择的字符集的列表。在必须处理非英语语言时，这会尤其方便。

5. Tabs

Tabs菜单选项提供了控制选项卡的位置和选择哪个选项卡处于活动状态的选项。这个菜单只有在你有多个选项卡会话打开的情况下才会显示。

- **Next Tab (下一个选项卡)**: 让列表中的下一个选项卡处于活动状态。
- **Previous Tab (前一个选项卡)**: 让列表中的前一个选项卡处于活动状态。
- **Move Tab to the Left (将选项卡向左移动)**: 将当前选项卡移动到前一个选项卡的前边。
- **Move Tab to the Right (将选项卡向右移动)**: 将当前选项卡移动到下一个选项卡的前边。
- **Detach Tab (分离窗口)**: 删除选项卡并用该选项卡会话启动一个新的GNOME Terminal窗口。
- **The Tab List (选项卡列表)**: 列出终端窗口中当前正在运行的会话选项卡。选择一个选项卡来直接跳到那个会话。

这部分允许你管理选项卡，一次打开多个选项卡时，这会特别方便。

6. Help

Help菜单选项提供了完整的GNOME Terminal用户手册，这样你就能仔细研究GNOME Terminal中使用的各个选项和功能了。

2.7 小结

要开始学习Linux命令行命令，你需要先能访问命令行。在图形化界面的世界里，有时这会费点小周折。本章讨论了从图形化桌面环境访问Linux命令行时，你应该考虑的一些事情。本章首先讨论了终端模拟，并介绍了你应该知道哪些功能，来保证Linux系统能够正确同终端模拟包通信以及正确显示文本和图形。

本章特别介绍了3个不同类型的终端模拟器。xterm终端模拟包是Linux中第一个可用的终端。它能模拟VT102和Tektronix 4014终端。KDE桌面项目创建了Konsole终端模拟包。它提供了一些很好的功能，比如在同一个窗口中包含多个会话，使用控制台和xterm会话，从而完全控制终端模拟参数。

最后，本章讨论了GNOME桌面项目的GNOME Terminal模拟包。GNOME Terminal也允许在单个窗口中打开多个终端会话，另外，它还提供了配置许多终端功能的便捷方法。

下一章将开始介绍Linux命令行命令。将会逐一介绍导航Linux文件系统，以及创建、删除和操作文件所需的各种文件。



本章内容

- 启动shell
- shell提示符
- bash手册
- 浏览文件系统
- 文件和目录列表
- 处理文件
- 处理目录
- 查看文件内容

大多数Linux发行版的默认shell都是GNU bash shell^①。本章将介绍bash shell的一些基本特性，然后带你逐步了解怎样用bash shell提供的基本命令来操作Linux文件和目录。如果你已经在Linux环境中熟练操作文件和目录，则可以直接跳过本章从第4章开始了解更多高级命令。

3.1 启动 shell

GNU bash shell能提供对Linux系统的交互式访问。它是作为常规程序运行的，通常是在用户登录终端时启动。登录时系统启动的shell依赖于用户账户的配置。

/etc/passwd文件包含了所有系统用户账户列表以及每个用户的基本配置信息。如下是从/etc/passwd文件中取出的样例条目：

```
rich:x:501:501:Rich Blum:/home/rich:/bin/bash
```

每个条目有七个字段，字段之间用冒号分隔。系统使用字段中的数据来赋予用户账户某些特定特性。这些字段包括：

- 用户名；
- 用户密码（如果密码存储在其他文件中，则是个占位符）；

① 在6.10之后的大部分Ubuntu版本上，默认的shell是dash。——译者注

- 用户的系统UID（用户ID）；
- 用户的系统GID（组ID）；
- 用户的全名；
- 用户的默认主目录；
- 用户的默认shell程序。

这些字段中的大部分将在第6章深入讨论。现在只需关心指定的shell程序就可以了。

多数Linux系统在为用户启动命令行界面（Command Line Interface，CLI）时采用默认的bash shell程序。bash程序同样使用命令行参数来修改所启动shell的类型。表3-1列出了bash支持的可定义启动shell类型的命令行参数。

表3-1 bash命令行参数

参 数	描 述
-c string	从string中读取命令并处理它们
-r	启动限制性shell，限制用户在默认目录下活动
-i	启动交互性shell，允许用户输入
-s	从标准输入读取命令

默认情况下，bash shell启动时会自动处理用户主目录下.bashrc文件中的命令。许多Linux发行版在此文件中加载特殊的共用文件，在共用文件中保存着针对所有系统用户的命令和设置。通常该文件位于/etc/bashrc，它经常设置各种应用程序中用到的环境变量（参见第5章）。

3.2 shell 提示符

一旦启动了终端模拟包或者从Linux控制台登录，你就会看到shell命令行界面。界面上的提示符就是到shell世界的大门，通常在这里输入shell命令。

默认的bash shell提示符是美元符号（\$），这个符号表明shell在等待用户输入。但你也可以更改shell提示符的格式。不同的Linux发行版采用不同格式的提示符。在Ubuntu Linux系统上，bash shell提示符看起来是这样的：

```
rich@user-desktop:~$
```

在Fedora Linux系统上，是这样的：

```
[rich@testbox~]$
```

你也可以配置提示符来让它显示环境的基本信息。第一个例子在提示符中显示了3条信息：

- 启动shell的用户名；
- 当前虚拟控制台编号；
- 当前目录（波浪线是主目录的缩略表示）。

第二个例子提供了类似的信息，除了它使用主机名而不是虚拟控制台编号。有两个环境变量

是用来控制命令行提示符的格式的：

- PS1：控制默认命令行提示符的格式。
- PS2：控制后续命令行提示符的格式。

shell在首次输入数据条目时使用默认的PS1提示符。输入一个需要其他信息的命令时，shell会显示由PS2环境变量指定的后续命令行提示符。

可以用echo命令来显示当前提示符设置：

```
rich@user-desktop:~$ echo $PS1
${debian_chroot:+($debian_chroot)}\u@\h:\w\$
rich@user-desktop:~$ echo $PS2
>
rich@user-desktop:~$
```

提示符环境变量的格式看起来非常奇特。shell使用特殊字符在命令行提示符中标记元素。表3-2列出了可以在提示符字符串中使用的特殊字符。

表3-2 bash shell提示符字符

字 符	描 述
\a	报警字符
\d	“日 月 年” 格式显示的日期
\e	ASCII转义字符
\h	本地主机名
\H	完全限定域名 (FQDN)
\j	shell当前管理的任务数
\l	shell的终端设备名中的基名
\n	ASCII换行符
\r	ASCII回车符
\s	shell的名称
\t	24时制HH:MM:SS格式的当前时间
\T	12时制HH:MM:SS格式的当前时间
\@	12时制am/pm格式的当前时间
\u	当前用户的用户名
\v	bash shell的版本
\V	bash shell的发行版本
\w	当前工作目录
\W	当前工作目录的基名
\!	这个命令在bash shell历史记录中的位置
\#	这个命令在当前命令行的位置
\\$	普通用户下的美元符 (\$), root用户下的井号 (#)
\nnn	与八进制数nnn对应的字符
\`	反斜线 (\`)
\[开始一个控制字符序列
\]	结束一个控制字符序列

注意，所有提示符特殊字符都从反斜线 (\) 开始。该字符将提示符中的特殊字符和普通文本分开来。在之前的例子中，提示符既有特殊字符，也有普通字符 (@符和方括号[])。你可以在提示符中创建任何提示符字符的组合。要创建新的提示符，只需给PS1变量赋一个新的字符串就行：

```
[rich@testbox-]$ PS1="\t[\u]\$ "
[14:40:32][rich]$
```

新shell提示符显示了当前时间和用户名。这个新的PS1定义只在这个shell会话中有效。启动新shell时，默认的shell提示符定义会重载。第5章将介绍如何改变所有shell会话的默认shell提示符。

3.3 bash 手册

大多数Linux发行版自带了用以查找shell命令以及其他GNU工具信息的在线手册。熟悉手册的使用对使用各种Linux工具大有裨益，尤其当你要弄清各种命令行参数时。

`man`命令用来访问存储在Linux系统上的手册页面。在你想要查找的工具的名称后面输入`man`命令，就可以找到那个工具相应的手册条目。图3-1展示了查找`date`命令的手册页面的例子。

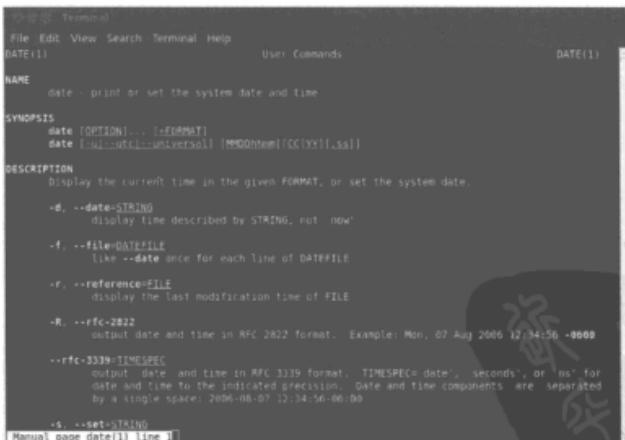


图3-1 Linux `date`命令的手册页面

用户手册将与该命令相关的信息分成几节，如表3-3所示。

表3-3 Linux man 页面格式

节	描述
Name	显示命令的名称和简介
Synopsis	显示命令的格式
Description	描述每个命令的选项
Author	提供该命令开发人员的信息
Reporting Bugs	提供提交bug报告的途径
Copyright	提供该命令源代码的版权情况
See Also	推荐查看相似的命令

在阅读手册页面时，可以按下空格键来翻页，也可以用方向键来上下翻动页面文本（假定你的终端模拟包支持方向键的相关功能）。当你看完手册页面了，直接按q键退出。

为查看bash shell的相关信息，你可以用下面的命令来打开man手册页面：

```
$ man bash
```

这样你就能翻阅bash shell的所有man手册页面了。在写脚本时，这种方法特别方便，不用再去翻书或者到网站上去搜索某个命令的特定格式了。man手册就在那里，你可以随时翻阅。

3.4 浏览文件系统

在启动shell会话时，如你从提示符中看到的，所在的位置一般都是主目录。通常你要离开主目录，转到系统的其他目录。本节将告诉你如何使用shell命令来完成目录切换。在开始前，先让我们来了解一下Linux文件系统，为下一步作铺垫。

3.4.1 Linux文件系统

如果你刚接触Linux系统，那么你可能很难弄清楚Linux标识文件和目录的方式，对于已经习惯于Microsoft Windows操作系统方式的人来说更是如此。在继续探索Linux系统之前，首先了解一下它的构建方式是有好处的。

你将注意到的第一点不同是，Linux在路径名中不使用驱动器盘符。在Windows中，PC上安装的物理驱动器决定了文件的路径名。Windows会为每个物理磁盘驱动器分配一个盘符，每个驱动器都会有自己的目录结构，以便访问存储在其上的文件。

举个例子，在Windows中，你经常看到这样的文件路径：

```
c:\Users\Rich\Documents\test.doc.
```

这说明文件test.doc位于Documents目录中，Documents又位于Rich目录中，Rich则位于Users目录中，而Users则放在盘符是C的硬盘分区中（通常C盘是PC上的第一块硬盘）。

Windows文件路径会告诉你究竟哪块物理硬盘分区上有文件test.doc。如果你想把文件保存在闪存中，比如，J盘标识的那个闪存，单击J盘的图标。文件将会使用路径J:\test.doc。这个路径说

明文件位于J盘的根目录下。

Linux则采用一种不同的方式。Linux将文件存储在单个目录结构中，这个目录我们称之为虚拟目录（virtual directory）。虚拟目录包含了安装在PC上的所有存储设备的文件路径，并将其并入到一个目录结构中。

Linux虚拟目录结构包含一个称为根（root）目录的基础目录。根目录下的目录和文件会按照访问它们的目录路径一一列出。这点跟Windows类似。

提示 你会发现，Linux使用正斜线（/）而不是反斜线（\）来在文件路径中划分目录。在Linux中反斜线用来标识转义字符，所以如果还在文件路径中使用会导致各种各样的问题。如果你是从Windows环境转过来的，那需要一点时间来适应。

举个例子，Linux文件路径/home/rich/Documents/test.doc说明，文件test.doc位于Documents目录，Documents又位于rich目录中，rich则在home目录中。但它并没有说明文件存放在哪个物理磁盘上。

Linux虚拟目录中比较复杂的部分是它如何来协调管理各个存储设备。我们称在Linux PC上安装的第一块硬盘为根驱动器。根驱动器包含了虚拟目录的核心，其他目录都是从那里开始构建的。

Linux会在根驱动器上创建一些特别的目录，我们称之为挂载点（mount point）。挂载点是虚拟目录中用于分配额外存储设备的目录。

虚拟目录会让文件和目录出现在这些挂载点目录中，然而实际上它们却存储在另外一个驱动器中。

通常系统文件会存储在根驱动器中，而用户文件则存储在另一驱动器中，如图3-2所示。

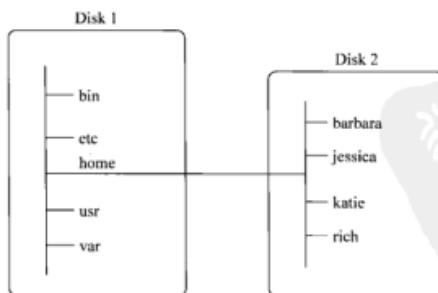


图3-2 Linux文件结构

在图3-2中，PC有两块硬盘。一块硬盘和虚拟目录的根目录（即一条正斜线“/”）关联起来。剩下的硬盘就可以挂载到虚拟目录结构中的任何地方。在这个例子中，第二块硬盘被挂载到了

/home位置，用户目录都位于这个位置。

Linux文件系统结构是从Unix文件结构演进过来的。遗憾的是，由于多年来不同Unix流派的推进，Unix文件结构已经变得很复杂。现今，貌似没有两个Unix或Linux系统遵从同样的文件系统结构。然而，一些具有相同功能的共用目录名称依然在沿用。表3-4列出了一些比较常见的Linux虚拟目录名称。

表3-4 常见Linux目录名称

目 录	用 途
/	虚拟目录的根目录。通常不会在这里存储文件
/bin	二进制目录，存放许多GNU用户级的工具
/boot	启动目录，存放启动文件
/dev	设备目录，Linux在这里创建设备节点
/etc	系统配置文件目录
/home	主目录，Linux在这里创建用户目录
/lib	库目录，存放系统和应用程序的库文件
/media	媒体目录，存放可移动媒体设备挂载点的地方
/mnt	挂载目录，另一个存放可移动媒体设备挂载点的地方
/opt	可选目录，通常用于存放可选的软件包
/root	根主目录
/sbin	系统二进制目录，存放许多GNU管理员级工具
/tmp	临时目录，可以在该目录中创建和删除临时工作文件
/usr	用户安装软件的目录
/var	可变目录，用以存放经常变化的文件，比如日志文件

在启动一个新的shell提示符后，会话通常从用户的主目录开始。主目录是分配给用户账户的一个特有目录。在创建用户账户时，系统通常会为其分配一个特有的目录（参见第6章）。

在Windows中，用户通常习惯于在图形化界面上在目录结构间切换。要想从命令行界面下在Linux的虚拟目录之间切换，需要使用cd命令。

3.4.2 遍历目录

在Linux文件系统上，你可以使用切换目录命令cd来将shell会话切换到另一个目录。cd命令的格式非常简单：

```
cd destination
```

cd命令可带单个参数destination，用以指定你想切换到的目录名。如果用户没有为cd命令指定目标路径，它将切换到你的用户主目录。

然而目标路径参数可以用两种方式表达：

- 绝对文件路径；

□ 相对文件路径。

后续几节将会进一步阐述二者之间的差别。

1. 绝对文件路径

用户可在虚拟目录中采用绝对文件路径来引用目录名。绝对文件路径定义了在虚拟目录结构中该目录的确切位置，以虚拟目录的根目录开始，相当于目录的全名。

因此，要引用usr目录中包含的lib目录中的Network Manager目录，用户可以使用绝对文件路径：

```
/usr/lib/Network Manager
```

采用绝对文件路径，毫无疑问直接说明了用户想切换到的确切目录。要用绝对文件路径来切换到文件系统中的某个特定位置，用户只需在cd命令后指定全路径名：

```
rich@testbox[-]$ cd /etc  
rich@testbox[etc]$
```

提示符说明，在使用cd命令后shell的新工作目录已是/etc。用户可以通过绝对文件路径切换到整个Linux虚拟目录结构的任何一级：

```
rich@testbox[-]$ cd /usr/lib/NetworkManager  
rich@testbox[NetworkManager]$
```

但当用户已经在其主目录下工作了，使用绝对路径往往就会过于冗长。比如说，如果用户已经在目录/home/rich中，再像下面这样敲命令切换到Documents目录就烦琐了：

```
cd /home/rich/Documents
```

幸好，我们还有简单的方法。

2. 相对文件路径

相对文件路径允许用户指定一个基于当前位置的目标文件路径，而无需再从根目录开始。相对文件路径不以代表根目录的正斜线 (/) 开头，而以目录名（如果用户准备切换到当前工作目录下的一个目录）或是一个表示基于用户当前目录的相对位置的特殊字符开始。有两个特殊字符：

□ 单点符 (.)，表示当前目录；

□ 双点符 (..)，表示当前目录的父目录。

双点符在导航目录层级时非常便利。比如说用户在主目录下的Documents目录而需要切换到Desktop目录，可以这么做：

```
rich@testbox[Documents]$ cd ../../Desktop  
rich@testbox[Desktop]$
```

双点符先将用户带到上一级目录，也就是用户的主目录，然后/Desktop这部分再将用户带到下一级目录，即Desktop目录。必要时用户也可用多个双点符来向上切换目录。比如用户现在在主目录 (/home/rich)，想切换到/etc目录，可以输入如下命令：

```
rich@testbox[-]$ cd ../../etc  
rich@testbox[etc]$
```

当然，在上面这种情况下，采用相对路径其实比采用绝对路径输入的字符更多，用绝对路径的话，用户只需输入/etc。

3.5 文件和目录列表

shell的最基本功能就是显示系统上有哪些文件。列表命令ls用于完成这个任务。本节将描述ls命令和所有可用来格式化其输出信息的选项。

3.5.1 基本列表功能

ls命令最基本的格式会显示当前目录下的文件和目录：

```
$ ls
4rich Desktop Download Music Pictures store store.zip test
backup Documents Drivers myprog Public store.sql Templates Videos
```

注意，ls命令输出的列表是按字母排序的（按列排序而不是按行排序）。如果用户用的是支持彩色的终端模拟器，ls命令还可以用不同的颜色来区分不同类型的文件。LS_COLORS环境变量控制着这个功能。不同的Linux发行版都根据各自终端模拟器的能力来设置这个环境变量。

如果没安装支持彩色的终端模拟器，可用带-F参数的ls命令来轻松的区分文件和目录。使用-F参数可以得到如下输出：

```
$ ls -F
4rich/ Documents/ Music/ Public/ store.zip Videos/
backup.zip Download/ myprog* store/ Templates/
Desktop/ Drivers/ Pictures/ store.sql test
```

-F参数在目录名后加了正斜线 (/)，以方便用户在输出中分辨它们。类似地，它会在可执行文件（比如上面的myprog文件）的后面加个星号，以便用户找出可在系统上运行的文件。

基本的ls命令在某种意义上有点容易让人误解。它显示了当前目录下的文件和目录，但并没有将全部都显示出来。Linux经常采用隐藏文件来保存配置信息。在Linux上，隐藏文件通常是文件名以句点开始的文件。这些文件并没有在默认的ls命令输出中显示出来（因此，我们称它们是隐藏文件）。

要把隐藏文件和普通文件和目录一起显示出来，就得用到-a参数。图3-3显示了给ls命令加-a参数的例子。

哇！现在很不一样了。在从图形化登录的用户的主目录里，我们可以看到很多隐藏的配置文件。上面这个例子是从一个登录进GNOME桌面会话的用户那里看到的。同样，请注意文件名以.bash开头的3个文件。它们是bash shell环境使用的3个隐藏文件。这些功能的细节我们将会在第5章中探讨。

-R参数是ls命令可用的另一个参数。它列出了当前目录下包含的目录中的文件。如果有很多个目录，这个输出会很长。这里有个简单的采用-R参数获得的输出：

```
$ ls -F -R
.:
file1 test1/ test2/
```

```
./test1:  
myprog1* myprog2*  
./test2:  
$
```

```
Terminal  
File Edit View Search Terminal Help  
$ ls -a  
.. .dircache .gtk-bookmarks .openoffice.org .thumbnails  
Documents .kde4.2.0-kde4 Pictures .update-manager-core  
Downloads .xfs Plugins postponed .update-notifier  
optitude .xrd auth .KCaauthority .printer-groups.xml .wininfo  
archive .evolution .httrack .profile .virtualBox  
avast .funconfig .httrackviewer .recently-used .winplayer  
.bash_history .gconf .kde .pulse .vmware  
.bash_logout .gconf8 .local .pulse-cookie .vnc  
.bashrc .gnome-2.0 .macromedia .recently-used.xbel .vnc  
.sogoufilter .gnome-2.0 Mail .selected-editor .xsession-errors  
.cache .gnome-lock .mission-control .recently-used.xbel .xsession-errors.old  
.compiz .gnome .mozilla .xsession-errors.old  
.config .gnome2 .Music .xterm .xsession-errors.old  
.xsession .gnome2_private .xawt .xterm .xsession-errors.old  
Desktop .gstreamer-0.10 .nautilus .Templates
```

图3-3 ls命令加-a参数

注意，首先-R参数显示了当前目录下的内容：一个文件（file1）和两个目录（test1和test2）。然后-R遍历了这两个目录，列出了每个目录下的文件。test1目录下有两个文件（myprog1和myprog2），而test2目录下没有文件。如果在test1或test2目录中还有更深的子目录，-R参数将会继续遍历这些目录。如你所见，对于大型目录结构来说，这个输出可能会很长很长。

3.5.2 修改输出信息

如你在前面看到的，ls命令并未输出每个文件的太多相关信息。要显示更多信息，另一个常用的参数是-1。-1参数会产生长列表格式的输出，包含了目录中每个文件的更多相关信息：

```
$ ls -1  
total 2064  
drwxrwxr-x 2 rich rich 4096 2010-08-24 22:04 4rich  
-rw-r--r-- 1 rich rich 1766205 2010-08-24 15:34 backup.zip  
drwxr-xr-x 3 rich rich 4096 2010-08-31 22:24 Desktop  
drwxr-xr-x 2 rich rich 4096 2009-11-01 04:06 Documents  
drwxr-xr-x 2 rich rich 4096 2009-11-01 04:06 Download  
drwxrwxr-x 2 rich rich 4096 2010-07-26 18:25 Drivers  
drwxr-xr-x 2 rich rich 4096 2009-11-01 04:06 Music  
-rwxr--r-- 1 rich rich 30 2010-08-23 21:42 myprog  
drwxr-xr-x 2 rich rich 4096 2009-11-01 04:06 Pictures
```

```
drwxr-xr-x 2 rich rich 4096 2009-11-01 04:06 Public
drwxrwxr-x 5 rich rich 4096 2010-08-24 22:04 store
-rw-rw-r-- 1 rich rich 98772 2010-08-24 15:30 store.sql
-rw-r--r-- 1 rich rich 107507 2010-08-13 15:45 store.zip
drwxr-xr-x 2 rich rich 4096 2009-11-01 04:06 Templates
drwxr-xr-x 2 rich rich 4096 2009-11-01 04:06 Videos
[rich@testbox ~]$
```

这种长列表格式的输出在每一行中列出了单个文件或目录。除了文件名，输出中还有其他有用信息。输出的第一行显示了在目录中包含的块的总数。之后，每一行都包含了关于文件（或目录）的下述信息：

- 文件类型，比如目录（d）、文件（-）、字符型文件（c）或块文件（b）；
- 文件的权限（参见第6章）；
- 文件的硬链接总数（参见3.6.3节）；
- 文件属主的用户名；
- 文件属组的组名；
- 文件的大小（以字节为单位）；
- 文件的上次修改时间；
- 文件名或目录名。

-l参数是一个强大的工具。有了它，你几乎可以看到系统上任何文件或目录的所有信息。

3.5.3 完整的参数列表

在进行文件管理时，ls命令的很多参数可能会派上用场。针对ls运行一下man命令，你就能看到可用来修改ls命令输出的参数就有好几页。

ls命令采用两种格式的命令行参数：

- 单字母参数；
- 全字参数。

单字母参数通常由英文破折号开始。全字参数则更易于看懂，通常以双英文破折号开始。许多参数都有单字母和全字两种版本，而有些则只有一种。表3-5列出了有助于使用ls命令的一些较常用的参数。

表3-5 一些常用的ls命令参数

单字母	全字	描述
-a	--all	输出包括以“.”打头的隐藏文件
-A	--almost-all	不要输出“.”和“..”文件
	--author	输出每个文件的作者
-b	--escape	输出不可打印字符的八进制值
	--block-size=size	按size字节大小的块来计算块大小（块数）
-B	--ignore-backups	不要列出名称中包含波浪线（~）的条目（波浪线用来表示备份的副本）
-c		按最后一次修改时间排序

(续)

单字母	全字	描述
-C	--color=when	按列输出条目 何时使用彩色 (always、never或者auto)
-d	--directory	列出目录条目而非内容，并且不要跟踪符号链接
-F	--classify	给条目追加文件类型标识符
	--file-type	只在部分文件类型（非可执行文件）后追加文件类型标识符
	--format=word	将输出格式化成 across (交叉)、commas (逗号)、horizontal (水平)、long (长)、single-column (单列)、verbose (详细) 或 vertical (垂直)
-g		输出除文件属主之外的所有信息
	--group-directories-first	在文件之前列出所有目录
-G	--no-group	在长列表输出格式下，不要显示组名
-h	--human-readable	打印大小，K表示千字节，M表示兆字节，G表示吉字节 和-h相同，但进率为1000，而非1024
	--si	
-i	--inode	显示每个文件的索引值 (inode)
-l		按长列表输出格式显示
-L	--dereference	对于链接文件，显示原文件信息
-n	--numeric-uid-gid	显示数字类型的userid和groupid以替代名字
-o		在长列表格式下不要显示组名
-r	--reverse	在输出文件和目录时，反转排序的顺序
-R	--recursive	递归地列出子目录内容
-s	--size	输出每个文件的块大小
-S	--sort=size	按文件大小排序输出
-t	--sort=time	按文件的修改时间排序输出
-u		输出文件的最后访问时间而非最后修改时间
-U	--sort=none	不要将输出排序
-v	--sort=version	按文件版本排序输出
-x		按行而非列输出条目
-X	--sort=extension	按文件扩展名排序输出

如果需要，也可一次使用多个参数。多个双破折线参数必须分开输入，而多个单破折线可以组合成一个字符串跟在一个单破折线后面。常见的组合是，用-a参数去列出所有文件，用-i参数列出每个文件的索引节点 (inode)，用-l参数产生一个长列表，再用-s参数列出文件的块大小。文件或目录的索引节点是内核分配给文件系统中每个对象的唯一标识数字。组合起所有这些参数就生成了很容易记住的-sail参数：

```
$ ls -sail
total 2360
301860  8 drwx----- 36 rich rich    4096 2010-09-03 15:12 .
65473   8 drwxr-xr-x  6 root root    4096 2010-07-29 14:20 ..
360621  8 drwxrwxr-x  2 rich rich    4096 2010-08-24 22:04 4rich
301862  8 -rw-r--r--  1 rich rich     124 2010-02-12 10:18 .bashrc
```

```

361443  8 drwxrwxr-x  4 rich rich   4096 2010-07-26 20:31 .ccache
301879  8 drwxr-xr-x  3 rich rich   4096 2010-07-26 18:25 .config
301871  8 drwxr-xr-x  3 rich rich   4096 2010-08-31 22:24 Desktop
301870  8 -rw-----  1 rich rich    26 2009-11-01 04:06 .dmrc
301872  8 drwxr-xr-x  2 rich rich   4096 2009-11-01 04:06 Download
360207  8 drwxrwxr-x  2 rich rich   4096 2010-07-26 18:25 Drivers
301882  8 drwx----- 5 rich rich   4096 2010-09-02 23:40 .gconf
301883  8 drwx----- 2 rich rich   4096 2010-09-02 23:43 .gconfd
360338  8 drwx----- 3 rich rich   4096 2010-08-06 23:06 .gftp

```

除了常用的-**l**参数的输出信息，还能看到每行加入了两个额外的数字。第一个数字是文件或目录的索引节点号，第二个数字是文件的块大小。第3列是文件的类型以及权限。我们将会在第6章中进一步了解文件的类型和权限。

尾随其后的，下一列数字是指向文件的硬链接数（将在3.6.3节中进一步探讨），紧接着是文件的属主、文件的属组、文件的大小（以字节为单位）、文件的最后修改时间（默认情况下），以及真实文件名。

3.5.4 过滤输出列表

由前面的例子可知，默认情况下，**ls**命令会输出目录下的所有文件。有时这个输出显得过多，而当你只需要查看单个文件的详细信息时更是如此。

幸而**ls**命令还支持在命令行下定义过滤器。它会用过滤器来决定应该在输出中显示哪些文件或目录。

这个过滤器就是个进行简单文本匹配的字符串。你可以在要用的命令行参数之后添加这个过滤器：

```
$ ls -l myprog
-rwxr--r-- 1 rich rich 30 2007-08-23 21:42 myprog
$
```

当用户指定特定文件的名称作为过滤器时，**ls**命令只会显示那个文件的信息。有时你可能不知道要找的那个文件的确切名称。**ls**命令能够识别标准通配符，并在过滤器中用它们来进行模式匹配。

- 句号代表一个字符。
- 星号代表零个或多个字符。

句号可用在过滤器字符串中替代任意位置的单个字符。例如：

```
$ ls -l mypro?
-rw-rw-r-- 1 rich rich 0 2010-09-03 16:38 myprob
-rw-rw-r-- 1 rich rich 30 2010-08-23 21:42 myprog
$
```

其中，过滤器mypro?与目录中的两个文件匹配。类似地，星号可用来匹配零个或多个字符：

```
$ ls -l myprob*
-rw-rw-r-- 1 rich rich 0 2010-09-03 16:38 myprob
-rw-rw-r-- 1 rich rich 0 2010-09-03 16:40 myproblem
$
```

其中，星号在匹配文件myprob时匹配了零个字符，而在匹配文件myproblem时匹配了3个字符。在不确定要寻找的文件的名称时，这是一个非常强大的功能。

3.6 处理文件

bash shell提供了很多在Linux文件系统上操作文件的命令。本节将带你逐步了解命令行下满足你所有操作文件需要的基本命令。

3.6.1 创建文件

你总会时不时地遇到要创建空文件的情况。有时应用程序希望在它们写入数据之前，某个日志文件已经存在。这时，你可用touch命令来轻松创建空文件：

```
$ touch test1
$ ls -il test1
1954793 -rw-r--r-- 1 rich rich 0 Sep 1 09:35 test1
$
```

touch命令创建了你指定的新文件，并将你的用户名作为文件的属主。因为在ls命令中我们采用了-l参数，所以输出结果的第一列显示了分配给该文件的索引节点号。在Linux文件系统中，每个文件都有唯一的索引节点号。

注意，文件的大小是零，这是因为touch命令只创建了一个空文件。touch命令还可用来自改变已有文件的访问时间和修改时间，而不改变文件的内容：

```
$ touch test1
$ ls -l test1
-rw-r--r-- 1 rich rich 0 Sep 1 09:37 test1
$
```

test1文件的修改时间现在已经从原来的时间更新了。如果只改变访问时间，可用-a参数。如果只改变修改时间，可用-m参数。默认情况下，touch使用当前时间。你可以通过-t参数加上特定的时间戳来指定时间：

```
$ touch -t 201112251200 test1
$ ls -l test1
-rw-r--r-- 1 rich rich 0 Dec 25 2011 test1
$
```

现在文件的修改时间已经被设定为另外一个时间了。

3.6.2 复制文件

对系统管理员来说，在文件系统中将文件和目录从一个位置复制到另一个位置是家常便饭。cp命令可完成这个任务。

在最基本的用法里，cp命令需要两个参数，源对象和目标对象：

```
cp source destination
```

当source和destination参数都是文件名时，cp命令将源文件复制至一个新文件，并且以destination命名。新文件就像个全新的文件一样，有新的创建时间和修改时间：

```
$ cp test1 test2
$ ls -il
total 0
1954793 -rw-r--r-- 1 rich rich 0 Dec 25 2011 test1
1954794 -rw-r--r-- 1 rich rich 0 Sep 1 09:39 test2
$
```

新文件test2显示了一个不同的索引节点号，说明它是一个崭新的文件。同时你也会发现test2的修改时间是它被创建的时间。如果目标文件已经存在了，那么cp命令将会提示你是否要覆盖已有文件：

```
$ cp test1 test2
cp: overwrite 'test2'? y
$
```

如果不回答y，文件复制将不会继续。也可以将文件复制到现有目录中：

```
$ cp test1 dir1
$ ls -il dir1
total 0
1954887 -rw-r--r-- 1 rich rich 0 Sep 6 09:42 test1
$
```

新文件现在在目录dir1中了，和源文件同名。在上述例子中都采用了相对路径，也可以采用绝对路径来表示源对象和目标对象。

可以用点标来把文件复制到当前所在的目录中：

```
$ cp /home/rich/dir1/test1 .
cp: overwrite './test1'?
```

同大多数命令一样，cp命令也有一些可以提供帮助的命令行参数，如表3-6所示。

表3-6 cp命令参数

参数	描述
-a	归档文件，并保留它们现有的属性
-b	创建已存在目标文件的备份，而非覆盖它
-d	保留
-f	强制覆盖已存在的目标文件，不提示
-i	在覆盖目标文件之前提示
-l	创建文件链接而非复制文件
-p	如果可能，保留文件属性
-r	递归地复制文件
-R	递归地复制目录
-s	创建一个符号链接而非复制文件
-S	覆盖默认的备份文件的后缀（默认是-）

(续)

参数	描述
-u	仅在源文件比目标文件新的情况下复制（相当于更新）
-v	详细模式，解释到底发生了什么
-x	仅限于当前文件系统的复制 ^①

下例将使用-p参数来为目标文件保留源文件的访问时间和修改时间：

```
$ cp -p test1 test3
$ ls -l
total 4
1954886 drwxr-xr-x  2 rich    rich        4096 Sep  1 09:42 dir1/
1954793 -rw-r--r--  1 rich    rich         0 Dec 25 2011 test1
1954794 -rw-r--r--  1 rich    rich         0 Sep  1 09:39 test2
1954888 -rw-r--r--  1 rich    rich         0 Dec 25 2011 test3
$
```

现在，即使文件test3是个全新的文件，它也和源文件test1有同样的时间戳。

-R参数极其强大。它允许你通过一个命令递归地复制整个目录的内容：

```
$ cp -R dir1 dir2
$ ls -l
total 8
drwxr-xr-x  2 rich    rich        4096 Sep  6 09:42 dir1/
drwxr-xr-x  2 rich    rich        4096 Sep  6 09:45 dir2/
-rw-r--r--  1 rich    rich         0 Dec 25 2011 test1
-rw-r--r--  1 rich    rich         0 Sep  6 09:39 test2
-rw-r--r--  1 rich    rich         0 Dec 25 2011 test3
$
```

现在dir2是dir1的完整副本。在cp命令中还可使用通配符：

```
$ cp -f test* dir2
$ ls -al dir2
total 12
drwxr-xr-x  2 rich    rich        4096 Sep  6 10:55 .
drwxr-xr-x  4 rich    rich        4096 Sep  6 10:46 ..
-rw-r--r--  1 rich    rich         0 Dec 25 2011 test1
-rw-r--r--  1 rich    rich         0 Sep  6 10:55 test2
-rw-r--r--  1 rich    rich         0 Dec 25 2011 test3
$
```

这个命令将所有文件名以test开头的文件复制到dir2。-f参数用来强制覆盖dir2目录中已有的test1文件，而不会提示用户。

3.6.3 链接文件

你可能已经注意到，cp命令的许多参数都是针对链接文件的。这是Linux文件系统的一个优

^① 这句话的背景是，Linux可同时挂载多个不同的文件系统类型的存储设备。——译者注

势。如需要在系统上维护同一文件的两份或多份副本，除了保存多份单独的物理文件副本之外，还可以采用保存一份物理文件副本和多个虚拟副本的方法。这种虚拟的副本就称为链接。链接是目录中指向文件真实位置的占位符。在Linux中有两种不同类型的文件链接：

- 符号链接，即软链接；
- 硬链接。

硬链接会创建一个独立文件，其中包含了源文件的信息以及位置。引用硬链接文件等同于引用了源文件：

```
$ cp -l test1 test4
$ ls -il
total 16
1954866 drwxr-xr-x  2 rich    rich    4096 Sep  1 09:42 dir1/
1954893 drwxr-xr-x  2 rich    rich    4096 Sep  1 09:45 dir2/
1954793 -rw-r--r--  2 rich    rich     0 Sep  1 09:51 test1
1954794 -rw-r--r--  1 rich    rich     0 Sep  1 09:39 test2
1954888 -rw-r--r--  1 rich    rich     0 Dec 25 2011 test3
1954793 -rw-r--r--  2 rich    rich     0 Sep  1 09:51 test4
$
```

-l参数创建了一个指向文件test1的硬链接test4。在文件列表中可以看出，文件test1和test4的索引节点号是相同的，这表明，实际上它们是同一个文件。还要注意，链接计数（列表输出的第3列）表明现在两个文件都有两个链接了。

说明 你只能在同种存储媒体上的文件之间创建硬链接，不能在不同挂载点下的文件间创建硬链接。在后一种情况下，你可以使用软链接。

而-s参数会创建一个符号链接，或者称为软链接：

```
$ cp -s test1 test5
$ ls -il test*
total 16
1954793 -rw-r--r--  2 rich    rich    6 Sep  1 09:51 test1
1954794 -rw-r--r--  1 rich    rich     0 Sep  1 09:39 test2
1954888 -rw-r--r--  1 rich    rich     0 Dec 25 2011 test3
1954793 -rw-r--r--  2 rich    rich     6 Sep  1 09:51 test4
1954891 lrwxrwxrwx  1 rich    rich     5 Sep  1 09:56 test5 -> test1
$
```

在文件的列表输出中要注意一些事项。首先，你会发现新建的test5文件有一个不同于test1文件的索引节点号，这说明Linux系统把它当做一个单独的文件。其次，文件变小了。链接文件只需要存储源文件的信息，并不需要存储源文件中的数据。列表的文件名区域说明了二者之间的关系。

提示 如果你想链接文件，还可用ln命令来替代cp命令。默认情况下，ln命令会创建硬链接。如想创建软链接，仍然要加-s参数。

在复制链接文件时要注意，如果用cp命令来复制一个链接到另一个源文件的文件，那么你复制的其实是源文件的另一份副本，而不是链接文件的。这点很容易混淆。可以创建一个指向源文件的新链接，而不用复制链接文件。可以创建指向同一文件的多个链接，但不要创建指向其他符号链接文件的多个符号链接。这样会生成一个链接文件链，不但容易混淆，还容易断掉，造成各种各样的问题。

3.6.4 重命名文件

在Linux中，重命名文件称为移动（moving）。mv命令就是用来将文件和目录移动到另外一个位置的：

```
$ mv test2 test6
$ ls -il test*
1954793 -rw-r--r--  2 rich    rich   6 Sep  1 09:51 test1
1954888 -rw-r--r--  1 rich    rich   0 Dec 25  2011 test3
1954793 -rw-r--r--  2 rich    rich   6 Sep  1 09:51 test4
1954891 lrwxrwxrwx  1 rich    rich   5 Sep  1 09:56 test5 -> test1
1954794 -rw-r--r--  1 rich    rich   0 Sep  1 09:39 test6
$
```

注意，移动文件会改变文件名，但保持同样的索引节点号和时间戳。移动一个有软链接指向它的文件会带来麻烦：

```
$ mv test1 test8
$ ls -il test*
total 16
1954888 -rw-r--r--  1 rich    rich   0 Dec 25  2011 test3
1954793 -rw-r--r--  2 rich    rich   6 Sep  1 09:51 test4
1954891 lrwxrwxrwx  1 rich    rich   5 Sep  1 09:56 test5 -> test1
1954794 -rw-r--r--  1 rich    rich   0 Sep  1 09:39 test6
1954793 -rw-r--r--  2 rich    rich   6 Sep  1 09:51 test8
[rich@test2 clsc]$ mv test8 test1
```

采用硬链接的test4文件依然有同样的索引节点号，这样完全没有问题。但test5文件指向了一个无效文件，它已不再是一个有效链接。

也可以用mv命令来移动目录：

```
$ mv dir2 dir4
```

整个目录里的内容都没有变化。唯一改变的是目录的名字。因此，mv命令运行起来比cp命令快很多。

3.6.5 删除文件

在你的Linux生涯里，迟早会遇到要删除已有文件的情况。不管是清理某个文件系统还是删除某个软件包，基本上你都需要删除文件。

在Linux中，删除（deleting）叫做移除（removing）^①。bash shell中删除文件的命令是rm。rm命令的基本格式非常简单：

```
$ rm -i test2
rm: remove 'test2'? y
$ ls -l
total 16
drwxr-xr-x  2 rich    rich    4096 Sep  1 09:42 dir1/
drwxr-xr-x  2 rich    rich    4096 Sep  1 09:45 dir2/
-rw-r--r--  2 rich    rich     6 Sep  1 09:51 test1
-rw-r--r--  1 rich    rich     0 Dec 25 2011 test3
-rw-r--r--  2 rich    rich     6 Sep  1 09:51 test4
lrwxrwxrwx  1 rich    rich      5 Sep  1 09:56 test5 -> test1
$
```

注意，命令提示你是不是要真的删除该文件。bash shell中没有回收站或者垃圾箱，所以文件一旦删除，就无法再找回了。

这里有一段关于删除带链接的文件的小故事：

```
$ rm test1
$ ls -l
total 12
drwxr-xr-x  2 rich    rich    4096 Sep  1 09:42 dir1/
drwxr-xr-x  2 rich    rich    4096 Sep  1 09:45 dir2/
-rw-r--r--  1 rich    rich     0 Dec 25 2011 test3
-rw-r--r--  1 rich    rich     6 Sep  1 09:51 test4
lrwxrwxrwx  1 rich    rich      5 Sep  1 09:56 test5 -> test1
$ cat test4
hello
$ cat test5
cat: test5: No such file or directory
$
```

文件test1含有一个指向它的硬链接test4和一个指向它的软链接test5，而它被删除了。注意观察发生了什么。两个链接文件依然都在，尽管test1文件已经消失了（在支持彩色的终端上能看到test5文件这时已经变成红色了）。当你查看硬链接test4文件的内容时，它依然显示了文件的内容。当你查看软链接test5文件的内容时，bash表明它已经不复存在了。

记住，硬链接文件采用和源文件相同的索引节点号。硬链接会一直维持这个索引节点号来保留数据，直到你删除了最后一个硬链接它的文件。所有的软链接文件都知道它所指向的文件不在了，所以指向的也就是一个无效文件。这是处理链接文件时要记住的一个重要特性。

rm命令的另外一个特性是，如果你要删除很多文件而不想被提示符烦到，可以用-f参数来强制删除。小心为妙！

^① 这里原文可理解为删除的功能实际上是移除（remove）命令rm完成的，在本书中我们依然用“删除”这个大家已经习惯了的叫法。——译者注

提示 和复制文件时一样，可在rm命令中采用通配符。同样的，做这件事时要格外小心。你删除的任何文件，哪怕是不小心删除的，都无法找回。

3.7 处理目录

在Linux中，有些命令（比如cp命令）对文件和目录都有效，而有些只对目录有效。创建新目录需要使用本节中讲到的一个特殊命令。删除目录也很有意思，所以本节也会讲到。

3.7.1 创建目录

在Linux中创建目录很简单——只要用mkdir命令就行：

```
$ mkdir dir3
$ ls -l
total 16
1954886 drwxr-xr-x  2 rich    rich    4096 Sep  1 09:42 dir1/
1954889 drwxr-xr-x  2 rich    rich    4096 Sep  1 10:55 dir2/
1954893 drwxr-xr-x  2 rich    rich    4096 Sep  1 11:01 dir3/
1954888 -rw-r--r--  1 rich    rich     0 Dec 25 2011 test3
1954793 -rw-r--r--  1 rich    rich     6 Sep  1 09:51 test4
$
```

系统创建了一个新目录，并给它分配了一个新的索引节点号。

3.7.2 删除目录

删除目录是件很棘手的事情，这是有原因的。删除目录时，很有可能会发生一些不好的事情。bash shell会尽可能地防止用户误删目录。删除目录的基本命令是rmdir：

```
$ rmdir dir3
$ rmdir dir1
rmdir: dir1: Directory not empty
$
```

默认情况下，rmdir命令只删除空目录。在dir1目录中有一个文件，所以rmdir没有删除它。你可以使用--ignore-fail-on-non-empty参数来删除非空目录。

老朋友rm命令也可以在处理目录上帮我们一把。如果你在用它时不带参数，你可能会有点失望：

```
$ rm dir1
rm: dir1: is a directory
$
```

但如果你真想删除一个目录，可以用-r参数来递归地删除目录中的文件，最后删除目录自身：

```
$ rm -r dir2
rm: descend into directory 'dir2'? y
```

```
rm: remove 'dir2/test1'? y
rm: remove 'dir2/test3'? y
rm: remove 'dir2/test4'? y
rm: remove directory 'dir2'? y
$
```

这种方法虽然可行，但很难用。你依然要确认每个文件是否要被删除。如果该目录有很多个文件和子目录，将会非常琐碎。省心而惯用的终极办法是使用rm命令，并加上参数-r和-f：

```
$ rm -rf dir2
$
```

就这么简单。没有警告，也没有一大堆问题，仅需另一个shell提示符。当然，这个方法是非常危险的，尤其是当你作为root用户登录时；尽量保守地采用这种做法，在使用之前再三检查，确认你真的要这样做。

说明 你应该已经注意到了，在上个例子中，两个命令行参数合并在了一起，用了一条单破折线。这是bash shell的一个特性，这样就可以把几个命令行参数合并，减少键入。

3.8 查看文件内容

到目前为止，我们几乎讲了关于文件的所有内容，只剩下怎样查看文件的内容了。Linux中有几个命令可以查看文件的内容而不需要调用其他文本编辑器（见第11章）。本节将介绍一些可以帮助你查看文件内容的命令。

3.8.1 查看文件统计信息

前面讲了ls命令可以提供很多关于文件的有用信息。然而依然有很多信息是你通过ls命令看不到的（或者至少是无法一次看到的）。

stat命令可以提供文件系统上某个文件的所有状态信息：

```
$ stat test10
  File: "test10"
  Size: 6          Blocks: 8          Regular File
Device: 306h/774d      Inode: 1954891      Links: 2
Access: (0644/-rw-r--r--)
  Access: Sat Sep  1 12:10:25 2010
  Modify: Sat Sep  1 12:11:17 2010
  Change: Sat Sep  1 12:16:42 2010
$
```

stat命令的结果显示了几乎所有你想知道的关于被检查文件的信息，甚至连存储该文件的设备的主设备和次设备编号都有。

3.8.2 查看文件类型

尽管stat命令提供了几乎所有的信息，但依然有一项重要的信息它没提供——文件类型。在尝试打开一个1 kB大小的文件之前，了解一下这个文件是什么类型的也会大有裨益。如果打开了一个二进制文件，你会在屏幕上看到各种乱码，甚至会叫你的终端模拟器挂起。

file命令就是一个用来查看文件类型的方便工具。它能够探测文件的内部并决定文件是什么类型的：

```
$ file test1  
test1: ASCII text  
$ file myscript  
myscript: Bourne shell script text executable  
$ file myprog  
myprog: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),  
dynamically linked (uses shared libs), not stripped  
$
```

file命令将文件分成3类。

- **文本文件**：包含可打印字符的文件。
- **可执行文件**：可在系统上运行的文件。

- **数据文件**：包含不可打印二进制字符、又不能在系统上运行的文件。

第一个例子显示了一个文本文件。file命令不光判断出该文件包含文本，还判断出了文本的字符编码。第二个例子显示了一个文本脚本文件。虽然文件是文本的，却是个脚本文件，可以在系统上运行。最后一个例子是一个二进制的可执行程序。file命令判断出了文件在编译时面向的平台，以及它需要哪种类型的库。如果你有从未知源获得的二进制文件，这会是个非常有用的特性。

3.8.3 查看整个文件

如果手头有一个很大的文本文件，你可能会想看看里面是什么内容。在Linux上有3个不同的命令可以完成这个任务。

1. cat命令

cat命令是显示文本文件中所有数据的得力工具：

```
$ cat test1  
hello  
  
This is a test file.  
  
That we'll use to      test the cat command.  
$
```

没什么特别的，只是文本文件的内容。这里还有一些可以和cat命令一起用的参数，可能会对你有所帮助。

-n参数会给所有的行加上行号：

```
$ cat -n test1
1 hello
2
3 This is a test file.
4
5
6 That we'll use to      test the cat command.
$
```

这个功能在检查脚本时会很有用。如果你只想给有文本的行加上行号，可以用**-b**参数：

```
$ cat -b test1
1 hello

2 This is a test file.

3 That we'll use to      test the cat command.
$
```

如果需要将多个空白行压缩到单个空白行，可用**-s**参数：

```
$ cat -s test1
hello

This is a test file.

That we'll use to      test the cat command.
$
```

最后，如果不想让制表符出现，可用**-T**参数：

```
$ cat -T test1
hello

This is a test file.

That we'll use to^Itest the cat command.
$
```

-T参数会用^I字符组合去替换文中的所有制表符。

对于大型文件来说，**cat**命令会有点烦琐。文件的文本会在显示器上一晃而过。好在这里有一个解决这个问题的简单办法。

2. more命令

cat命令的主要缺陷是，一旦运行了**cat**命令，你就无法控制后面的操作了。为了解决这个问题，开发人员写了**more**命令。**more**命令会显示文本文件的内容，但会在显示每页数据之后停下来，**more**命令在屏幕上的输出可以在图3-4中看到。

注意图3-4中屏幕的底部，**more**命令显示了一个标签，说明你仍然在**more**程序中，以及现在在这个文本文件中的位置。这是**more**命令的提示符。这时你可以输入表3-7中选项里的其中一个。

```

Terminal
File Edit View Search Terminal Help
root:x:0:0:root:/root:/bin/bash
daemon:x:1:daemon:/usr/sbin:/bin/sh
bin:x:2:bin:/bin:/bin/sh
sys:x:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/sync
games:x:5:66:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:mail:/var/mail:/bin/sh
news:x:9:news:/var/spool/news:/bin/sh
uucp:x:10:16:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101:/var/lib/libuuid:/bin/sh
syslog:x:101:103::/home/syslog:/bin/false
messagebus:x:102:105::/var/run/dbus:/bin/false
avahi-autoipd:x:103:108:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
avahi:x:104:109:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
...Mo: e - (591)

```

图3-4 用more命令来显示文本文件

表3-7 more命令选项

选 项	描 述
H	显示帮助菜单
spacebar	显示文件文本的下一屏
z	显示文件文本的下一屏
ENTER	显示文件文本的下一行
d	显示文件文本的后面半屏（会更新11行）
q	退出程序
s	显示文件文本的下一行
f	显示下一屏文件文本
b	显示上一屏文件文本
/expression	在文件中查找匹配文本表达式的内容
n	在文件中查找下一处匹配已指定表达式的内容
,	跳到指定表达式匹配到的第一处内容
!cmd	执行shell命令
v	在当前行启动vi编辑器
CTRL-L	重绘当前屏
=	显示当前行在文件中的行号
.	重复执行前一个命令

more命令只支持了文本文件中基本的移动。如果要更多高级功能，可以试试less命令。

3. less命令

尽管从名字上看上去，它不会像more命令一样高级，但less命令的命名实际上是个文字游戏（从俗语“less is more”得来），它实为more命令的升级版本。它提供了一些极为实用的在文本文件中前后翻动的功能，还有一些极先进的搜索功能。

less命令也可显示文件的内容，而不用读取整个文件。这点是cat和more命令在读取大文件时的明显缺陷。

less命令和more命令的功能基本上差不多，每次显示一屏文件文本。图3-5显示了实际使用中的less命令。

```

Terminal
File Edit View Search Terminal Help
daemon:x:1:daemon:/usr/sbin:/bin/sh
bin:x:2:bin:/bin:/bin/sh
sys:x:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:68:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:mail:/var/mail:/bin/sh
news:x:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/noneexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
sysLog:x:101:103::/home/syslog:/bin/false
messagebus:x:102:105::/var/run/dbus:/bin/false
avahi-autopid:x:103:108:Avahi autopid daemon,...:/var/lib/avahi-autopid:/bin/false
avahi:x:104:109:Avahi mDNS daemon,...:/var/run/avahi-daemon:/bin/false
couchdb:x:105:113:CouchDB Administrator,...:/var/lib/couchdb:/bin/bash

```

图3-5 使用less命令查看文件内容

注意less命令在提示符中提供了额外的信息——显示了文件的总行数以及现在显示的行号区间。less命令支持了more命令支持的所有参数，同时还多了一些选项。想了解全部选项，可以查看less命令的手册页面。其中一组特性就是less命令能够识别上下键以及上下翻页键（如果用的是正确配置的终端的话）。这样在查看文件内容时你就有完全控制权了。

3.8.4 查看部分文件

通常用户要查看的数据要么在文本文件的开头，要么在文本文件的末尾。如果这些数据是在一个大型文件中，那你就得等cat或more加载完整个文件之后才能看到内容。如果数据是在文件的末尾（比如日志文件），你可能需要翻成千上万行的文本才能到最后的内容。好在Linux有解决这两个问题的专用命令。

1. tail命令

tail命令会显示文件末尾部分的内容。默认情况下它会显示文件的末尾10行，你也可以通过

参数来指定显示的行数，如表3-8所示。

表3-8 tail命令行参数

参 数	描 述
-c bytes	显示文件最后的bytes个字节的字符
-n lines	显示文件最后的lines行
-f	让tail程序一直保持活动状态，如果有新的内容加到文件的末尾就显示出来
--pid=PID	和-f参数一起，跟踪一个文件直到进程ID为PID的进程结束
-s sec	和-f参数一起，在每次循环输出之间休眠sec秒
-v	总是显示带文件名的头
-q	从不显示带文件名的头

-f参数是tail命令的一个突出的特性，它允许你在其他进程使用该文件时查看文件的内容。tail命令会保持活动状态并不断地显示添加到文件中的内容。这是实时监测系统日志的绝妙方式。

2. head命令

尽管不及tail命令那么知名，head命令依然可以如你所愿：它会显示文件开头那些行的内容。默认情况下，它会显示文件前十行的文本。类似于tail命令，它也支持-c和-n参数，这样就可以指定想要显示的内容了。

通常文件的开头不会改变，head命令并不支持-f参数特性。head命令是不知道文件内容而想知道大致内容时的利器，而无需加载全部文件。

3.9 小结

本章涵盖了在shell提示符下操作Linux文件系统的基础知识。一开始我们讨论了bash shell，之后介绍了怎样和shell交互。命令行采用提示符来表明你可以输入命令了。你可以定制提示符来显示系统信息、登录ID，甚至是日期和时间。

bash shell提供了很多可用以创建和操作文件的工具。在开始操作文件之前，先了解一下Linux怎么存储文件很有必要。本章讨论了Linux虚拟目录的基础知识，然后展示了Linux如何挂载存储设备。在描述了Linux文件系统之后，本章还带你逐步了解了如何使用cd命令在虚拟目录里切换目录。

展示如何进入指定目录后，我们又演示了怎样用ls命令来列出目录中的文件和子目录。ls命令有很多参数可用来定制输出内容。用户可以通过ls命令获得有关文件和目录的信息。

touch命令非常有用，可创建空文件和修改已有文件的访问时间和修改时间。本章还介绍了如何使用cp命令将已有文件复制到其他位置，之后逐步介绍了如何链接文件以快速在两个位置访问同一文件。cp命令可完成这样的操作，ln命令也可以。

再后面，我们讲了怎样用mv命令重命名文件（在Linux中称为移动文件），以及如何用rm命令

删除文件（Linux中称为移除文件），还介绍了怎样用mkdir和rmdir命令完成对目录的类似操作。

最后，本章以如何查看文件的内容结尾。cat、more和less命令可以提供查看文件全部内容的简便方法，而tail和head命令则可查看文件中的一小部分内容。

下章将继续讨论bash shell的命令。我们会了解一下管理Linux系统时经常用到的高级系统管理命令。

本章内容

- 管理进程
- 获取磁盘统计信息
- 挂载新磁盘
- 排序数据
- 归档数据

第3章介绍了Linux文件系统上切换目录以及处理文件和目录的基本知识。文件管理和目录管理是Linux shell的主要功能之一；在开始脚本编程之前我们还需要了解一下其他方面的知识。本章将详细介绍Linux系统管理命令，演示如何通过命令行命令来探查Linux系统的内部信息，最后将会介绍一些可以用来操作系统上数据文件的命令。

4.1 监测程序

Linux系统管理员面临的最复杂的任务之一是，查看哪些程序正在系统上运行——尤其是现在，图形化桌面集成了大量的应用，整合为一个独立的完整桌面。系统上通常运行着大量的程序。

好在有一些命令行工具可以使你的生活轻松一些。本节将会介绍一些能帮你在Linux系统上管理程序的基本工具，以及如何使用。

4.1.1 探查进程

当程序运行在系统上时，我们称之为进程（process）。想监测这些进程，需要熟悉ps命令的用法。ps命令好比工具的瑞士军刀，它能输出运行在系统上的所有程序的许多信息。

遗憾的是，随它的稳健而来的还有复杂性——有数不清的参数，这或许让ps命令成了最难掌握的命令。大多数系统管理员在掌握了能提供他们需要信息的一组参数之后，就一直坚持只使用这组参数。

默认情况下，ps命令并不会提供那么多的信息：

```
$ ps
 PID TTY      TIME CMD
 3081 pts/0    00:00:00 bash
 3209 pts/0    00:00:00 ps
$
```

没什么特别的吧？默认情况下，ps命令只会显示运行在当前控制台下的属于当前用户的进程。在此例中，我们只运行了bash shell（注意，shell也只是运行在系统上的另一个程序而已），以及ps命令自己。

上例中的基本输出显示了程序的进程号（PID，Process ID）、它们运行在哪个终端（TTY）以及进程已用的CPU时间。

说明 ps命令叫人头疼的地方（也正是它如此复杂的原因）在于，曾经它有两个版本。每个版本都有自己的命令行参数集，这些参数控制着输出什么信息以及如何显示。最近，Linux开发人员已经将这两种ps命令格式合并到了单个ps命令中。当然，也加入了他们自己的风格。

Linux系统中使用的GNU ps命令支持3个不同类型的命令行参数：

- Unix风格的参数，前面加单破折线；
- BSD风格的参数，前面不加破折线；
- GNU风格的长参数，前面加双破折线。

下面将进一步解析这3种不同的参数类型，并举例演示它们如何工作。

1. Unix风格的参数

Unix风格的参数是从贝尔实验室开发的AT&T Unix系统上原有的ps命令继承下来的。这些参数如表4-1所示。

表4-1 Unix风格的ps命令参数

参 数	描 述
-A	显示所有进程
-N	显示与指定参数不符的所有进程
-a	显示除控制进程（session leader ^① ）和无终端的进程外的所有进程
-d	显示除控制进程外的所有进程
-e	显示所有进程
-C cmdlist	显示包含在cmdlist列表中的进程
-G grpplist	显示组ID在grpplist列表中的进程
-U userlist	显示属主的用户ID在userlist列表中的进程
-g grpplist	显示会话或组ID在grpplist列表中的进程 ^②

① 关于session leader的概念，可参考《Unix环境高级编程：第2版》第9章内容。——译者注

② 这个在不同的Linux发行版中可能不尽相同，有的发行版中grpplist代表会话ID，有的发行版中grpplist代表有效组ID。——译者注

(续)

参数	描述
-p pidlist	显示PID在pidlist列表中的进程
-s sesslist	显示会话ID在sesslist列表中的进程
-t ttylist	显示终端ID在ttylist列表中的进程
-u userlist	显示有效用户ID在userlist列表中的进程
-F	显示更多额外输出（相对-f参数而言）
-O format	显示默认的输出列以及format列表指定的特定列
-M	显示进程的安全信息
-C	显示进程的额外调度器信息
-f	显示完整格式的输出
-J	显示任务信息
-l	显示长列表
-o format	仅显示由format指定的列
-y	不要显示进程标记（process flag，表明进程状态的标记）
-Z	显示安全标签（security context）信息 ^①
-H	用层级格式来显示进程（树状，用来显示父进程）
-n namelist	定义了WCHAN列显示的值
-w	采用宽输出模式，不限宽度显示
-L	显示进程中的线程
-V	显示ps命令的版本号

上面给出的参数已经很多了，还有很多。使用ps命令的关键不在于记住所有可用的参数，而在于记住对你最有用的那些参数。大多数Linux系统管理员都有自己的一组参数，他们会牢牢记住这些用来提取有用的进程信息的参数。举个例子，如果你想查看系统上运行的所有进程，可用-e f参数组合（ps命令允许你像这样把参数组合在一起）：

```
$ ps -ef
UID      PID  PPID  C STIME TTY      TIME CMD
root      1  0 11:29 ?        00:00:01 init [5]
root      2  0 11:29 ?        00:00:00 [kthread]
root      3  2  0 11:29 ?        00:00:00 [migration/0]
root      4  2  0 11:29 ?        00:00:00 [ksoftirqd/0]
root      5  2  0 11:29 ?        00:00:00 [watchdog/0]
root      6  2  0 11:29 ?        00:00:00 [events/0]
root      7  2  0 11:29 ?        00:00:00 [khelper]
root     47  2  0 11:29 ?        00:00:00 [kblockd/0]
root     48  2  0 11:29 ?        00:00:00 [kacpid]
68    2349  1  0 11:30 ?        00:00:00 hald
root   2489  1  0 11:30 tty1    00:00:00 /sbin/mingetty tty1
root   2490  1  0 11:30 tty2    00:00:00 /sbin/mingetty tty2
root   2491  1  0 11:30 tty3    00:00:00 /sbin/mingetty tty3
root   2492  1  0 11:30 tty4    00:00:00 /sbin/mingetty tty4
```

① security context也叫security label，是SELinux采用的声明资源的一种机制。——译者注

```

root      2493  1  0 11:30 tty5    00:00:00 /sbin/mingetty tty5
root      2494  1  0 11:30 tty6    00:00:00 /sbin/mingetty tty6
root      2956  1  0 11:42 ?
apache   2958  2956 0 11:42 ?
apache   2959  2956 0 11:42 ?
root      2995  1  0 11:43 ?
root      2997  2995 0 11:43 ?
root      3078  1981 0 12:00 ?
rich     3080  3078 0 12:00 ?
rich     3081  3080 0 12:00 pts/0  00:00:00 -bash
rich     4445  3081 3 13:48 pts/0  00:00:00 ps -ef
$
```

上例中，我们略去了输出中的不少行，以节约空间。但如你所看到的，Linux系统上运行着很多的进程。这个例子用了两个参数：-e参数指定显示所有运行在系统上的进程；-f参数则扩展了输出，这些扩展的列包含了有用的信息。

- UID**: 启动这些进程的用户。
- PID**: 进程的进程号(PID)。
- PPID**: 父进程的进程号(如果该进程是由另一个进程启动的)。
- C**: 进程生命周期中的CPU利用率。
- STIME**: 进程启动时的系统时间。
- TTY**: 进程启动时的终端设备。
- TIME**: 运行进程需要的累计CPU时间。
- CMD**: 启动的程序名称。

上例中输出了合理数量的信息，这也正是大多数系统管理员希望看到的。如果想要获得更多的信息，可采用-1参数，它会产生一个长格式输出：

```

$ ps -1
F S  UID PID PPID C PRI NI ADDR SZ WCHAN TTY      TIME CMD
D S  500 3081 3080 0 80 0 - 1173 wait pts/0  00:00:00 bash
D R  500 4463 3081 1 80 0 - 1116 -  pts/0  00:00:00 ps
$
```

注意使用了-1参数之后出现的那些额外的列。

- F**: 内核分配给进程的系统标记。
- S**: 进程的状态(0代表正在运行；S代表在休眠；R代表可运行，正等待运行；Z代表僵化，进程已结束但父进程已不存在；T代表停止)。
- PRI**: 进程的优先级(越大的数字代表越低的优先级)。
- NI**: 谦让度(nice)值用来参与决定优先级。
- ADDR**: 进程的内存地址。
- SZ**: 假如进程被换出，所需交换空间的大致大小。
- WCHAN**: 进程休眠的内核函数的地址。

在继续讲下一个内容之前，要特别提一个非常好用的参数——-H。-H参数能把输出的进程组织成一个层级的格式，简单地说就是树状。你可以很轻松地看懂哪些进程启动了哪些进程。以下

是从加参数-eFH的输出中截取下来的片段：

```
$ ps -efH
UID  PID  PPID  C STIME TTY      TIME      CMD
root  3078  1981  0 12:00 ?        00:00:00  sshd: rich [priv]
rich  3080  3078  0 12:00 ?        00:00:00  sshd: rich@pts/0
rich  3081  3080  0 12:00 pts/0    00:00:00  -bash
rich  4803  3081  1 14:31 pts/0    00:00:00  ps -efH
```

注意CMD这列输出内容的平移，它表明运行中程序的层级：首先是root用户启动的sshd进程（这是Secure Shell，也就是SSH的服务器端的会话，负责监听远程SSH连接）。然后，由于这个会话是从远程的终端连接到系统上的，SSH主进程启动了一个终端进程（pts/0），之后终端又启动了bash shell。

那之后，ps命令作为bash进程的子进程开始运行了。在多用户系统上，在定位失控的进程或跟踪这些进程属于哪个userid或终端时，这个工具非常有用。

2. BSD风格的参数

了解了Unix风格的参数之后，我们来一起看一下BSD风格的参数。BSD（伯克利软件发行版）是加州大学伯克利分校开发的一个Unix版本。它和AT&T Unix系统有着许多细小的不同，这也导致多年的Unix争论。BSD版的ps命令参数如表4-2所示。

表4-2 BSD风格的ps命令参数

参数	描述
T	显示跟当前终端关联的所有进程
a	显示跟任意终端关联的所有进程
g	显示所有的进程，包括控制进程
r	仅显示运行中的进程
x	显示所有的进程，甚至包括未分配任何终端的进程
U userlist	显示归userlist列表中某用户ID所有的进程
p pidlist	显示PID在pidlist列表中的进程
t ttylist	显示所关联的终端在ttylist列表中的进程
D format	除了默认输出的列之外，还输出由format指定的列
X	按过去的Linux i386寄存器格式显示
Z	将安全信息添加到输出中
J	显示任务信息
I	采用长模式
O format	仅显示由format指定的列
S	采用信号格式显示
U	采用基于用户的格式显示
V	采用虚拟内存格式显示
N namelist	定义在WCHAN列中使用的值
O order	定义显示信息列的顺序
S	将数值信息从子进程加到父进程上，比如CPU和内存的使用情况
c	显示真实的命令名称（用以启动进程的程序名称）

(续)

参数	描述
e	显示命令使用的环境变量
f	用分层格式来显示进程，表明哪些进程启动了哪些进程
h	不显示头信息
k sort	指定用以将输出排序的列
n	和WCHAN信息一起显示出来，用数值来表示用户ID和组ID
w	为较宽屏幕显示宽输出
H	将线程按进程来显示
m	在进程后显示线程
L	列出所有格式指定符
V	显示ps命令的版本号

如你所见，Unix和BSD类型的参数有很多重叠的地方。你使用其中某种类型参数得到的信息也同样可以使用另一种获得。大多数情况下，你只要选择自己所喜欢格式的参数类型就行了（比如你在使用Linux之前就已经习惯BSD环境了）。

在使用BSD参数时，ps命令会自动改变输出以模仿BSD格式。下例是使用1参数的输出：

```
$ ps 1
F  UID PID PPID PRI NI VSZ RSS WCHAN STAT TTY      TIME COMMAND
0 500 3081 3080 20  0 4692 1432 wait   Ss  pts/0    0:00 -bash
0 500 5104 3081 20  0 4468 844 -       R+  pts/0    0:00 ps 1
$
```

注意，其中大部分的输出列跟我们使用Unix风格参数时的输出是一样的，只有一小部分不同。

□ VSZ：进程在内存中的大小，以千字节（kB）为单位。

□ RSS：进程在未换出时占用的物理内存。

□ STAT：代表当前进程状态的双字符状态码。

许多系统管理员都喜欢BSD风格的1参数。它能输出更详细的进程状态码（STAT列）。双字符状态码能比Unix风格输出的单字符状态码更清楚地表示进程的当前状态。

第一个字符采用了和Unix风格S列的输出相同的值，说明进程是在休眠、运行还是等待。第二个参数进一步说明进程的状态。

□ <：该进程运行在高优先级上。

□ N：该进程运行在低优先级上。

□ L：该进程有页面锁定在内存中。

□ S：该进程是个控制进程。

□ T：该进程是多线程的。

□ +：该进程运行在前端。

从前面的例子可以看出，bash命令处于休眠状态，但同时它也是一个控制进程（在我的会话中，它是主要的进程）。而ps命令则运行在系统的前端。

3. GNU全字参数

最后，GNU开发人员在这个新的整合过的ps命令中加入了另外一些参数。一些GNU全字参数复制了现有的Unix或BSD类型的参数，而一些提供了新功能。表4-3列出了现有的GNU全字参数。

表4-3 GNU风格的ps命令参数

参数	描述
--deselect	显示所有进程，命令行中列出的进程
--Group grplist	显示组ID在grplist列表中的进程
--User userlist	显示用户ID在userlist列表中的进程
--group grplist	显示有效组ID在grplist列表中的进程
--user userlist	显示有效用户ID在userlist列表中的进程
--pid pidlist	显示pid在pidlist列表中的进程
--ppid pidlist	显示父pid在pidlist列表中的进程
--sid sidlist	显示会话ID在sidlist列表中的进程
--tty ttylist	显示终端设备号在ttylist列表中的进程
--format format	仅显示由format指定的列
--context	显示额外的安全信息
--cols n	将屏幕宽度设置为n列
--columns n	将屏幕宽度设置为n列
--cumulative	包含已停止的子进程的信息
--forest	用层级结构显示出进程和父进程之间的关系
--headers	在每页输出中都显示列的头
--no-headers	不显示列的头
--lines n	将屏幕高度设为n行
--rows n	将屏幕高度设为n排
--sort order	指定将输出按哪列排序
--width n	将屏幕宽度设为n列
--help	显示帮助信息
--info	显示调试信息
--version	显示ps命令的版本号

你可以将GNU全字参数和Unix风格的或BSD风格的参数混用来定制输出。GNU全字参数中最好用的功能就是--forest参数。它会显示进程的层级信息，并用ASCII字符绘出可爱的图表：

```
1981 ?    00:00:00 sshd
3078 ?    00:00:00  \_ sshd
3080 ?    00:00:00      \_ sshd
3081 pts/0  00:00:00          \_ bash
16676 pts/0  00:00:00            \_ ps
```

这种格式使得跟踪子进程和父进程变得十分容易。

4.1.2 实时监测进程

ps命令虽然在收集运行在系统上的进程信息时非常有用，但它也有不足之处：ps命令只能显示某个特定时间点的信息。如果你想观察频繁换进换出内存的进程的趋势，用ps命令就不方便了。

而top命令刚好适用这种场景。top命令跟ps命令相似，能够显示进程信息，但它是实时显示的。图4-1是top命令运行时输出的截图。

输出的第一部分显示的是系统的概况：第一行显示了当前时间、系统的运行时间、登入的用户数以及系统的平均负载。

平均负载有3个值：最近1分钟的、最近5分钟的和最近15分钟的平均负载。值越大说明系统的负载越高。最近1分钟的负载值很高也很常见，因为有时会有进程突然开始活动；但如果15分钟平均负载都很高，说明系统可能有问题了。

The screenshot shows the terminal window of a Linux desktop environment. The title bar says "rich@rich-desktop: ~". The main area displays the output of the "top" command.

top - 16:04:38 up 1 min, 2 users, load average: 0.82, 0.52, 0.28														
Tasks:	179 total	1 running	178 sleeping	0 stopped	0 zombie									
Cpu(s):	0.5%us, 1.3%sy, 0.0%ni, 97.0%id, 0.3%wa, 0.0%hi, 0.0%si, 0.0%st													
Mem:	1826684K total	433076k used	593088k free	58448k buffers										
Swap:	2781176k total	0k used	2781176k free	191088k cached										
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND			
952	root	20	0	35924	22m	7576	S	1	2.3	0:03.99	xorg			
1432	root	20	0	15656	1868	1516	S	0	0.2	0:00.25	prl_mmouse_d			
1527	rich	20	0	78512	17m	13m	S	0	1.7	0:09.43	nautilus			
1668	rich	20	0	64568	15m	11m	S	0	1.5	0:01.25	gnome-terminal			
1	root	20	0	28684	1656	1208	S	0	0.2	0:00.61	init			
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd			
3	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0			
4	root	20	0	0	0	0	S	0	0.0	0:00.01	ksftirqd/0			
5	root	RT	0	0	0	0	S	0	0.0	0:00.00	watchdog/0			
6	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/l			
7	root	20	0	0	0	0	S	0	0.0	0:00.00	ksftirqd/l			
8	root	RT	0	0	0	0	S	0	0.0	0:00.00	watchdog/l			
9	root	20	0	0	0	0	S	0	0.0	0:00.00	events/0			
10	root	20	0	0	0	0	S	0	0.0	0:00.04	events/1			
11	root	20	0	0	0	0	S	0	0.0	0:00.00	cpuset			
12	root	20	0	0	0	0	S	0	0.0	0:00.00	khelper			
13	root	20	0	0	0	0	S	0	0.0	0:00.00	netns			
14	root	20	0	0	0	0	S	0	0.0	0:00.00	async/mgr			
15	root	20	0	0	0	0	S	0	0.0	0:00.00	pmd			
17	root	20	0	0	0	0	S	0	0.0	0:00.00	sync_supers			

图4-1 top命令运行时的输出

说明 Linux系统管理的要点在于如何定义系统的高负载。这个值取决于系统的硬件配置以及系统上通常运行的程序。对某个系统来说是高负载的值可能对另一系统来说就是个正常值。通常，如果系统的负载值超过了2，就说明系统比较繁忙了。

第二行显示了概要的进程信息——top命令的输出中进程叫做任务（task）：多少进程处在运行、休眠、停止或是僵化状态（僵化状态是指进程完成了，但父进程没有响应）。

下一行显示了CPU信息。top根据进程的属主（用户还是系统）和进程的状态（运行、空闲

还是等待)将CPU利用率分成几类输出。

紧跟其后的两行说明了系统内存的状态。前行说的是系统的物理内存:总共有多少内存,当前用了多少,还有多少空闲。后一行说的同样的信息,不过是针对系统交换空间(如果分配了的话)的状态来说的。

最后一部分显示了当前运行中的进程的详细列表——有些列跟ps命令的输出类似。

- ❑ PID: 进程的进程号。
- ❑ USER: 进程属主的名字。
- ❑ PR: 进程的优先级。
- ❑ NI: 进程的谦让度值。
- ❑ VIRT: 进程占用的虚拟内存总量。
- ❑ RES: 进程占用的物理内存总量。
- ❑ SHR: 进程和其他进程共享的内存总量。
- ❑ S: 进程的状态(D代表可中断的休眠状态, R代表在运行状态, S代表休眠状态, T代表跟踪状态或停止状态, Z代表僵化状态)。
- ❑ %CPU: 进程使用的CPU时间比例。
- ❑ %MEM: 进程使用的内存占可用内存的比例。
- ❑ TIME+: 自进程启动到目前为止的CPU时间总量。
- ❑ COMMAND: 进程的命令行名称,也就是启动的程序名。

默认情况下,top命令在启动时会按照%CPU值来排序,你可以在top运行时用下面的交互式命令之一来重新排序。每个交互式命令都是单字符,在top命令运行时键入可改变top的行为。这些命令都列在了表4-4中。

表4-4 top的交互式命令

命 令	描 述
1	切换单CPU状态模式和对称处理器模式
B	打开/关闭表中重要数字的加粗显示
I	切换Irix/Solaris模式
Z	设置表的颜色
l	显示/关闭平均负载信息行
t	显示/关闭CPU信息行
m	显示/关闭MEM和SWAP行
f	添加/移除输出中的不同信息列
o	更改信息行的显示顺序
F或O	选择一列来将进程排序(默认为%CPU)
< 或 >	将排序的行左移或右移一列
R	切换正常排序还是倒序排序
H	显示/关闭显示线程情况
C	切换显示进程的命令名还是完整的命令行输入(包括参数)

(续)

命 令	描 述
i	切换是否显示空闲进程
S	切换显示累计CPU时间还是相对CPU时间
x	打开/关闭高亮显示排序序列
y	打开/关闭高亮显示运行中的任务
z	切换彩色模式还是单色模式
b	打开/关闭x和y模式的高亮模式
u	显示某个用户的进程
n或#	设置要显示的进程数
k	结束指定的进程（必须是进程属主或root用户）
r	改变指定进程的优先级（必须是进程属主或root用户）
d或S	改变更新的间隔（默认值为3s）
w	把当前设置写到一个配置文件中
q	退出top命令

用户在top命令的输出上有很大的控制权。用这个工具，你就能经常找出占用系统大部分资源的罪魁祸首了。当然了，一旦找到了，下一步就是结束这些进程。下个话题就是这方面的。

4.1.3 结束进程

做系统管理员很重要的一个技能就是知道何时以及如何结束一个进程。有时进程挂起了，只要有办法对付一下让进程重新运行或结束就行了。但有时，有的进程会耗尽CPU而且不释放资源。这两种情景下，你就需要能控制进程的命令。

Linux沿用了Unix进行进程间通信的方法。在Linux上，进程之间通过信号来通信。进程的信号就是预定义好的一个消息，进程能识别它并决定忽略还是作出反应。开发人员编程实现程序如何处理信号。大多数写得好的程序都能接收和处理标准Unix进程信号。这些信号都列出在表4-5中。

表4-5 Linux进程信号

信 号	名 称	描 述
1	HUP	挂起
2	INT	中断
3	QUIT	结束运行
9	KILL	无条件终止
11	SEGV	段错误
15	TERM	尽可能终止
17	STOP	无条件停止运行，但不终止
18	TSTP	停止或暂停，但继续在后台运行
19	CONT	在STOP或TSTP之后恢复执行

在Linux上有两个命令可以向运行中的进程发出进程信号：kill命令和killall命令。

1. kill命令

kill命令可通过PID（进程号）给进程发信号。默认情况下，kill命令会向命令行中列出的全部PID发送一个TERM信号。遗憾的是，你只能用进程的PID而不能用命令名，所以kill命令有时并不好用。

要发送进程信号，你必须是进程的属主或登录为root用户。

```
$ kill 3940  
-bash: kill: (3940) - Operation not permitted  
$
```

TERM信号告诉进程可能的话停止运行。不过，如果有跑飞了的进程，它通常会忽略这个请求。如果要强制终止，-s参数支持指定其他信号（用信号名或信号值）。

通常可接受的处理方式是先试试TERM信号。如果进程忽略它，可用INT或HUP信号。程序收到了这些信号，会在关掉进程前有序地停止它正在做的事。KILL信号的强制性最强。当进程接收到这个信号时，它会立即停止运行。这可能会导致文件损坏。

你能从下例中看到，kill命令不会有任何输出：

```
# kill -s HUP 3940  
#
```

要检查kill命令是不是起作用了，可以再运行个ps或top命令看那些进程是否已经停止运行。

2. killall命令

killall命令非常强大，它支持通过进程名而不是进程号来结束进程。killall命令也支持通配符，这在系统因负载过大而变得很慢时很有用：

```
# killall http*  
#
```

上例的命令结束了所有以http开头的进程，比如Apache Web服务器的httpd服务。

警告 以root用户身份登录系统时使用killall命令要特别小心，很容易就误用通配符而结束了重要的系统进程。这可能会破坏文件系统。

4.2 监测磁盘空间

系统管理员的另一个重要任务就是监测系统磁盘的使用情况。不管运行的是个简单的Linux台式机还是大型的Linux服务器，你都要知道还有多少空间给你的应用程序用。

在Linux系统上有几个命令行命令可以用来帮助管理存储媒体。本节将介绍在日常系统管理中经常用到的核心命令。

4.2.1 挂载存储媒体

如第3章中讨论的，Linux系统将所有的磁盘都挂载到一个虚拟目录下。在使用新的存储媒体之前，你需要把它放到虚拟目录下。这项工作称为挂载（mounting）。

在今天的图形化桌面环境里，大多数Linux发行版都能自动挂载指定的可移动存储媒体。明显的，可移动存储媒体是可从PC上轻易移除的媒体，比如CD-ROM、软盘和U盘。

如果你用的发行版不支持自动挂载和卸载可移动存储媒体，那就必须手动完成。本节将介绍一些Linux命令行命令，可以帮你来管理可移动存储设备。

1. mount命令

Linux上用来挂载媒体的命令叫做mount。默认情况下，mount命令会输出当前系统上挂载的设备列表：

```
$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sdal on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/sdb1 on /media/disk type vfat
(rw,nosuid,nodev,uhelper=hal,shortname=lower,uid=503)
$
```

mount命令提供如下四部分信息：

- 媒体的设备文件名；
- 媒体挂载到虚拟目录的挂载点；
- 文件系统类型；
- 已挂载媒体的访问状态。

上面例子的最后一行输出中，U盘被GNOME桌面自动挂载到了挂载点/media/disk。vfat文件系统类型说明它是在Windows机器上被格式化的。

要手动在虚拟目录中挂载设备，需要以root用户身份登录。下面是手动挂载媒体设备的基本命令：

```
mount -t type device directory
```

type参数指定了磁盘被格式化的文件系统类型。Linux可以识别非常多的文件系统类型。如果是和Windows PC共用这些存储设备，那通常会是如下文件系统类型。

□ vfat：Windows长文件系统。

□ ntfs：Windows NT、XP、Vista以及Windows 7中广泛使用的高级文件系统。

□ iso9660：标准CD-ROM文件系统。

大多数U盘和软盘会被格式化成vfat文件系统。而数据CD则必须使用iso9660文件系统类型。

后面两个参数定义了该存储设备的设备文件的位置以及挂载点在虚拟目录中的位置。比如说，手动将U盘/dev/sdb1挂载到/media/disk，可用下面命令：

```
mount -t vfat /dev/sdb1 /media/disk
```

媒体设备挂载到了虚拟目录后，root用户就有了对该设备的所有访问权限，而其他用户的访问则会被限制。你可以通过目录权限（将在第6章中介绍）指定用户对设备的访问权限。

如果你要用到mount命令的一些高级功能，这些参数已在表4-6中列出。

表4-6 mount命令的参数

参 数	描 述
-a	挂载/etc/fstab文件中指定的所有文件系统
-f	使mount命令模拟挂载设备，但并不真的挂载
-F	和-a参数一起使用，将会并行地挂载所有文件系统
-v	详细模式，将会说明挂载设备的每一步
-l	不启用任何/sbin/mount.filesystem下的文件系统帮助文件
-t	给ext2、ext3或XFS文件系统自动添加文件系统标签
-n	挂载设备，但不注册到/etc/mtab已挂载设备文件中
-p num	对加密文件进行挂载时，从文件描述符num中获得密码短语
-s	忽略该文件系统不支持的挂载选项
-r	将设备挂载为只读的
-w	将设备挂载为可读可写的（默认参数）
-L label	将设备按指定的label挂载
-U uuid	将设备按指定的uuid挂载
-o	和-a参数一起使用，限制命令只作用到特定的一组文件系统上
-o	给文件系统添加特定的选项

-o参数允许在挂载文件系统时添加一些以逗号分隔的额外选项。以下为常用的选项。

- ro：按只读的挂载（read-only）。
- rw：按读写允许的挂载（read-write）。
- user：允许普通用户挂载文件系统。
- check=none：挂载文件系统时不进行完整性校验。
- loop：挂载一个文件。

现在Linux中流行的一个做法是按.iso文件来发行CD。.iso文件是将一个完整的CD镜像文件。大多数烧录CD的软件都能基于这个.iso文件来烧录一张CD。mount命令的特性之一就是允许你将一个.iso文件直接挂载到Linux虚拟目录里，而不用先将它烧录到CD里。可用-o参数加上loop选项来完成这个操作：

```
$ mkdir mnt
$ su
Password:
# mount -t iso9660 -o loop MEPIS-KDE4-LIVE-DVD_32.iso mnt
```

```
# ls -l mnt
total 16
-r--r--r-- 1 root root 702 2007-08-03 08:49 about
dr-xr-xr-x 3 root root 2048 2007-07-29 14:30 boot
-r--r--r-- 1 root root 2048 2007-08-09 22:36 boot.catalog
-r--r--r-- 1 root root 894 2004-01-23 13:22 cdrom.ico
-r--r--r-- 1 root root 5229 2006-07-07 18:07 MCWL
dr-xr-xr-x 2 root root 2048 2007-08-09 22:32 mepis
dr-xr-xr-x 2 root root 2048 2007-04-03 16:44 OSX
-r--r--r-- 1 root root 107 2007-08-09 22:36 version
# cd mnt/boot
# ls -l
total 4399
dr-xr-xr-x 2 root root 2048 2007-06-29 09:00 grub
-r--r--r-- 1 root root 2392512 2007-07-29 12:53 initrd.gz
-r--r--r-- 1 root root 94760 2007-06-14 14:56 memtest
-r--r--r-- 1 root root 2014704 2007-07-29 14:26 vmlinuz
#

```

mount命令会将CD镜像.iso文件当做真实CD来挂载，用户就能访问它的文件系统了。

2. umount命令

从Linux系统上移除一个可移动设备时，不能直接从系统上移除，而应该先卸载它。

提示 Linux上不能直接弹出已挂载的CD。如果你在从光驱中移除CD时遇到麻烦，通常是该CD还挂载在虚拟目录里。先卸载它，然后再去尝试弹出。

卸载设备的命令叫umount(注意，命令名中并没有字母n，不是卸载对应的英文单词unmount)。umount命令的格式非常简单：

```
umount [directory | device]
```

umount命令支持通过设备文件或者是挂载点来指定要卸载的设备。如果有任何程序正在使用设备上的文件，系统就不会允许你卸载它。

```
[root@testbox mnt]# umount /home/rich/mnt
umount: /home/rich/mnt: device is busy
umount: /home/rich/mnt: device is busy
[root@testbox mnt]# cd /home/rich
[root@testbox rich]# umount /home/rich/mnt
[root@testbox rich]# ls -l mnt
total 0
[root@testbox rich]#
```

在上例中，命令行提示符仍然在挂载设备的文件系统结构中，所以umount命令不会卸载该镜像文件。一旦命令提示符移动到了该镜像文件的文件系统，umount命令就能成功卸载该镜像文件了。^①

① 如果在卸载设备时，系统提示设备繁忙，无法卸载设备，通常是有进程还在访问该设备或使用该设备上的文件。这时可用lsof命令获得使用它的进程信息，然后在应用中停止使用该设备或停止该进程。lsof命令的用法很简单：lsof /path/to/device/node，或者lsof /path/to/mount/point。——译者注

4.2.2 使用df命令

有时你需要知道在某个设备上还有多少磁盘空间。df命令就是用来轻松查看所有已挂载磁盘的使用情况的：

```
$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/sda2        18251068   7703964   9605024  45% /
/dev/sdal        101086     18680     77187  20% /boot
tmpfs           119536         0    119536  0% /dev/shm
/dev/sdb1        127462    113892    13570  90% /media/disk
$
```

df命令会显示每个有数据的已挂载文件系统。如你在前例中看到的，有些已挂载设备仅限系统内部使用。命令输出如下：

- 设备的设备文件位置；
- 能容纳多少个1024字节大小的块；
- 已用了多少个1024字节大小的块；
- 还有多少个1024字节大小的块可用；
- 已用空间所占的比例；
- 设备挂载到了哪个挂载点上。

df命令有一些命令行参数可用，但基本上你不会用到。一个常用的参数是-h。它会把输出中的磁盘空间按人类可读的形式显示，通常用M来替代兆字节，用G代替吉字节：

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sdb2        18G  7.4G  9.2G  45% /
/dev/sdal        99M  19M   76M  20% /boot
tmpfs           117M     0  117M  0% /dev/shm
/dev/sdb1        125M 112M   14M  90% /media/disk
$
```

说明 Linux系统后台一直有进程来处理文件或使用文件。df命令的输出值显示的是Linux系统认为的当前值。有可能系统上有运行的进程已经创建或删除了某个文件，但尚未释放文件。这个值是不会算进闲置空间的。

4.2.3 使用du命令

通过df命令，你很容易发现哪个磁盘就快要没存储空间了。系统管理员面临的下一个问题是发生这种情况时怎么办。

另一个有用的命令是du命令。du命令可以显示某个特定目录（默认情况下是当前目录）的磁盘使用情况。这是用来判断你系统上某个目录下是不是有超大文件的快速方法。

默认情况下，du命令会显示当前目录下所有的文件、目录和子目录的磁盘使用情况，它会以磁盘的块为单位来显示每个文件或目录占用了多大存储。在标准的主目录中，这个输出会是一个比较长的列表。下面是du命令的部分输出：

```
$ du
484 ./gstreamer-0.10
8 ./Templates
8 ./Download
8 ./ccache/7/0
24 ./ccache/7
368 ./ccache/a/d
384 ./ccache/a
424 ./ccache
8 ./Public
8 ./gphpedit/plugins
32 ./gphpedit
72 ./gconfd
128 ./nautilus/metafiles
384 ./nautilus
72 ./bit torrent/data/metainfo
20 ./bit torrent/data/resume
144 ./bit torrent/data
152 ./bit torrent
8 ./Videos
8 ./Music
16 ./config/gtk-2.0
40 ./config
8 ./Documents
```

每行输出开始地方的数值，是每个文件或目录占用的磁盘块数。注意这个列表是从一个目录层级的最底部开始的，然后按文件、子目录、目录逐级向上。

这么用du命令（不加参数，用默认参数）作用并不大。我们更想知道每个单独的文件和目录占用了多大的磁盘空间，而在找想要的信息时翻这么多页的输出并无意义。

下面是能让du命令用起来更方便的几个命令行参数。

- -c：显示所有已列出文件总的大小。
- -h：按人类可读的格式输出大小，即用K替代千字节，用M替代兆字节，用G替代吉字节。
- -s：显示每个输出参数的总计。

系统管理员下一步要用的就是操作大量数据时用的文件处理命令。下节将主要讲这个。

4.3 处理数据文件

当你有大量数据时，通常很难处理这些信息及提取有用信息。正如在上节中学习的关于du命令的使用，系统命令很容易就输出过量的信息。

Linux系统提供了一些命令行工具来处理大量数据。本节将会介绍每个系统管理员以及每个日常Linux用户都应该知道的、能叫生活轻松些的基本命令。

4.3.1 排序数据

处理大量数据时的一个常用命令是sort命令。从名字就可以知道，sort命令是用来对数据进行排序的。

默认情况下，sort命令按你为这个会话指定的默认语言的排序规则来对文本文件中的数据行排序。

```
$ cat file1
one
two
three
four
five
$ sort file1
five
four
one
three
two
$
```

相当简单。但事情并非总是像看起来那么简单。看下面的例子：

```
$ cat file2
1
2
100
45
3
10
145
75
$ sort file2
1
10
100
145
2
3
45
75
$
```

你期望这些数字能按值排序，但叫你失望了。默认情况下，sort命令会把数字当做字符而执行标准的字符排序，产生的输出可能根本就不是你要的。解决这个问题可用-n参数，它会告诉sort命令把数字识别成数字而不是字符，并且将它们按值排序：

```
$ sort -n file2
1
2
3
10
```

```
45
75
100
145
$
```

现在结果是不是好多了？另一个常用的参数是-M，按月排序。Linux的日志文件经常会在每行的起始位置有一个时间戳，用来表明事件是什么时候发生的：

```
Sep 13 07:10:09 testbox smartd[2718]: Device: /dev/sda, opened
```

如果将含有时间戳日期的文件按默认的排序方法来排序，会得到类似于下面的结果：

```
$ sort file3
Apr
Aug
Dec
Feb
Jan
Jul
Jun
Mar
May
Nov
Oct
Sep
$
```

这并不是想要的结果。如果用了-M参数，sort命令就能识别三字符的月份命名，并相应的排序：

```
$ sort -M file3
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
$
```

还有其他一些方便的sort参数可用，如表4-7所示。

表4-7 sort命令参数

单破折线	双破折线	描述
-b	--ignore-leading-blanks	排序时忽略起始的空白
-C	--check=quiet	不排序，如果数据无序也不要报告
-c	--check	不排序，但检查输入数据是不是已排序；未排序的话，报告

(续)

单破折线	双破折线	描述
-d	--dictionary-order	仅考虑空白和字母，不考虑特殊字符
-f	--ignore-case	默认情况下，会将大写字母排在前面；这个参数会忽略大小写
-g	--general-number-sort	按通用数值来排序（跟-n不同，把值当浮点数来排序，支持科学计数法表示的值）
-i	--ignore-nonprinting	在排序时忽略不可打印字符
-k	--key=POS1[,POS2]	排序从POS1位置开始；如果指定了POS2的话，到POS2位置结束
-M	--month-sort	用三字符月份名按月份排序
-m	--merge	将两个已排序数据文件合并
-n	--numeric-sort	按字符串数值来排序（并不转换为浮点数）
-o	--output=FILE	将排序结果写出到指定的文件中
-R	--random-sort	按随机生成的哈希表的键值排序
	--random-source=FILE	指定-R参数用到的随机字节的源文件
-r	--reverse	反序排序（升序变成降序）
-S	--buffer-size=SIZE	指定使用的内存大小
-s	--stable	禁用最后重排序比较
-T	--temporary-direction=DIR	指定一个位置来存储临时工作文件
-t	--field-separator=SEP	指定一个用来区分键位置的字符
-u	--unique	和-c参数一起使用时，检查严格排序；不和-c参数一起用时，仅输出第一例相似的两行
-z	--zero-terminated	用NULL字符来为每一行结尾而不是用换行符

4

-k和-t参数在对按字段分隔的数据进行排序时非常有用，例如/etc/passwd文件。可以用-t参数来指定字段分隔符，然后用-k参数来指定排序的字段。举个例子，要对前面提到的密码文件/etc/passwd根据用户ID进行数值排序，可以这么做：

```
$ sort -t: -k 3 -n /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin/nologin
daemon:x:2:2:daemon:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

现在数据已经按第3个字段——用户ID的数值排序。

-n参数在排序数值时非常有用，比如du命令的输出：

```
$ du -sh * | sort -nr
1008k  mrtg-2.9.29.tar.gz
972k  bldgl
888k  fbs2.pdf
760k  Printtest
680k  rsync-2.6.6.tar.gz
660k  code
516k  fig1001.tiff
496k  test
496k  php-common-4.0.4pl1-6mdk.i586.rpm
448k  MesaGLUT-6.5.1.tar.gz
400k  plp
```

注意，-r参数让结果按降序输出，这样就更容易看到目录下的哪些文件占用空间最多。

说明 本例中用到的管道命令（|）将du命令的输出重定向到sort命令。我们将在第10章中进一步讨论。

4.3.2 搜索数据

你会经常需要在大文件中找一行数据，而这行数据埋藏在文件的中间。这时，你并不需要人工看完整个文件，你只需要grep命令来帮查找。grep命令的命令行格式如下：

```
grep [options] pattern [file]
```

grep命令会到输入中或你指定的文件中查找包含匹配指定模式的字符的行。grep的输出就是包含了匹配模式的行。

下面两个简单的例子演示了使用grep命令来对4.3.1节中用到的文件file1进行搜索：

```
$ grep three file1
three
$ grep t file1
two
three
$
```

第一个例子在文件file1中搜索能匹配模式three的文本。grep命令输出了匹配了该模式的行。第二个例子在文件file1中搜索能匹配模式t的文本。这个例子里，file1中有两行匹配了指定的模式，两行都输出了。

由于grep命令非常流行，目前它还在经历着大量的更新。有很多功能加进了grep命令。如果看一下它的手册页面，你会发现它是多么通用。

如果要进行反向搜索（输出不匹配该模式的行），可加-v参数：

```
$ grep -v t file1
one
four
five
$
```

如果要显示匹配模式的行所在的行号，可加-n参数：

```
$ grep -n t file1
2:two
3:three
$
```

如果只要知道多少行含有匹配的模式，可用-c参数：

```
$ grep -c t file1
2
$
```

如果要指定多于一个匹配模式，得到满足两个模式中任意一个的所有结果，用-e参数来指定每个模式：

```
$ grep -e t -e f file1
two
three
four
five
$
```

这个例子输出了含有字符t或字符f的所有行。

默认情况下，grep命令用基本的Unix风格正则表达式来匹配模式。Unix风格正则表达式采用特殊字符来定义怎样查找匹配的模式。

要想进一步了解正则表达式的细节，可以参考第19章的内容。

这里有个简单在grep搜索中使用正则表达式的例子：

```
$ grep [tf] file1
two
three
four
five
$
```

正则表达式中的方括号表明grep应该搜索包含t或者f字符的匹配。如果不使用正则表达式，grep就会搜索匹配字符串tf的文本。

egrep命令是grep的一个衍生，支持POSIX扩展正则表达式。POSIX扩展正则表达式含有更多的可以用来指定匹配模式的字符（参见第19章）。fgrep则是另外一个重要版本，支持将匹配模式指定为用换行符分隔的一列固定长度的字符串。这样就可以把一列字符串放到一个文件中，然后在fgrep命令中用那列字符串来在一个大型文件中搜索字符串了。

4.3.3 压缩数据

如果你接触过Microsoft Windows，你必然用过zip文件。它如此流行以至于微软已经将其集成进了Windows XP操作系统。zip工具可以轻松地将大型文件（文本文件和可执行文件）压缩成占用更少空间的小文件。

Linux含有几个文件压缩工具。虽然听上去不错，但实际上这经常会在用户下载文件时给用户造成混淆。表4-8列出了Linux上的文件压缩工具。

表4-8 Linux文件压缩工具

工 具	文件扩展名	描 述
bzip2	.bz2	采用Burrows-Wheeler块排序文本压缩算法和霍夫曼编码
compress	.Z	原始的Unix文件压缩工具，逐渐消失中
gzip	.gz	GNU压缩工具，用Lempel-Ziv编码
zip	.zip	Windows上PKZIP工具的Unix实现

compress文件压缩工具已经很少在Linux系统上看到了。如果下载了带.Z扩展名的文件，你通常可以用第8章中介绍的软件包安装方法来安装compress包（在很多Linux发行版上叫ncompress），然后再用uncompress命令来解压文件。

1. bzip2工具

bzip2工具是个正在逐渐普及的相对来说较新的压缩包，在压缩大型二进制文件领域尤其流行。bzip2软件包有以下几个工具。

- ❑ bzip2：用来压缩文件。
- ❑ bzcat：用来显示压缩的文本文件的内容。
- ❑ bunzip2：用来解压压缩后的.bz2文件。
- ❑ bzip2recover：用来尝试恢复损毁的压缩文件。

默认情况下，bzip2命令尝试压缩原始文件，并用压缩后的文件（同样的文件名后加.bz2扩展名）替换它：

```
$ ls -l myprog
-rwxrwxr-x 1 rich rich 4882 2007-09-13 11:29 myprog
$ bzip2 myprog
$ ls -l my*
-rwxrwxr-x 1 rich rich 2378 2007-09-13 11:29 myprog.bz2
$
```

myprog程序的原始大小是4882 B，用bzip2压缩后的大小是2378 B。同时，bzip2命令自动用压缩后的bzip2文件替换了原文件，.bz2扩展名说明我们是采用什么技术进行压缩的。

解压文件用bunzip2命令：

```
$ bunzip2 myprog.bz2
$ ls -l myprog
$ -rwxrwxr-x 1 rich rich 4882 2007-09-13 11:29 myprog
$
```

如你所见，解压后的文件又回到了原文件大小。如果压缩了文本文件，你无法再用cat、more或less命令来查看文件的内容。这时，你可以用bzcat命令：

```
$ bzcat test.bz2
This is a test text file.
The quick brown fox jumps over the lazy dog.
This is the end of the test text file.
$
```

bzcat命令显示了压缩文件里的文本内容，而并不需要解压文件。

2. gzip工具

到目前为止，Linux上最流行的文件压缩工具就是gzip工具了。gzip包是GNU项目发起的试图替代原来Unix的compress工具的压缩工具。这个软件包含有下面的工具。

- gzip**: 用来压缩文件。
- gzcat**: 用来查看压缩过的文本文件的内容。
- gunzip**: 用来解压文件。

这些工具基本上跟bzip2工具的用法一样：

```
$ gzip myprog
$ ls -l my*
-rwxrwxr-x 1 rich rich 2197 2007-09-13 11:29 myprog.gz
$
```

gzip命令会压缩你在命令行指定的文件。你也可以在命令行指定几个文件名甚至用通配符来一次压缩几个文件：

```
$ gzip my*
$ ls -l my*
-rwxr--r-- 1 rich rich 103 Sep 6 13:43 myprog.c.gz
-rwxr--r-- 1 rich rich 5178 Sep 6 13:43 myprog.gz
-rwxr--r-- 1 rich rich 59 Sep 6 13:46 myscript.gz
-rwxr--r-- 1 rich rich 60 Sep 6 13:44 myscript.gz
$
```

gzip命令会压缩该目录中匹配通配符的每个文件。

3. zip工具

zip工具和流行的Phil Katz为MS-DOS和Windows开发的PKZIP软件是兼容的。Linux的zip软件包有5个。

- zip**: 创建一个压缩文件，包含指定的文件和目录。
- zipcloak**: 创建一个加密的压缩文件，包含指定的文件和目录。
- zipnote**: 从zip文件中提取批注。
- zipsplit**: 将一个现有zip文件分割成多个更小的固定大小的文件（最开始是用来把大的zip文件复制进软盘的）。
- unzip**: 从压缩过的zip文件中提取文件和目录。

要查看zip工具的所有可选参数，在命令行下输入命令自身：

```
$ zip
Copyright (C) 1990-2005 Info-ZIP
Type 'zip "-L"' for software license.
Zip 2.31 (March 8th 2005). Usage:
zip [-options] [-b path] [-t mmddyyyy] [-n suffixes] [zipfile list]
[-xi list]

The default action is to add or replace zipfile entries from list,
which can include the special name - to compress standard input.

If zipfile and list are omitted, zip compresses stdin to stdout.

-f freshen: only changed files -u update: only changed or new files
-d delete entries in zipfile -m move into zipfile (delete files)
-r recurse into directories -j junk directory names
-o store only -l convert LF to CR LF
-i compress faster -g compress better
-q quiet operation -v verbose operation
-c add one-line comments -z add zipfile comment
-@ read names from stdin -o make file as old as latest entry
-x exclude the following names -i include only the following names
-F fix zipfile (-FF try harder) -D do not add directory entries
-A adjust self-extracting exe -J Junk zipfile prefix (unzipsfx)
-T test zipfile integrity -X exclude extra file attributes
-y store symbolic links as the link instead of the referenced file
-R PKZIP recursion (see manual)
-e encrypt -n don't compress these suffixes
$
```

zip工具的强大之处在于，它能够将整个目录下的文件都压缩进单个文件。这让它成为归档整个目录结构的理想工具。

```
$ zip -r testzip test
adding: test/ (stored 0%)
adding: test/test1/ (stored 0%)
adding: test/test1/myprog2 (stored 0%)
adding: test/test1/myprog1 (stored 0%)
adding: test/myprog.c (deflated 39%)
adding: test/file3 (deflated 2%)
adding: test/file4 (stored 0%)
adding: test/test2/ (stored 0%)
adding: test/file1.gz (stored 0%)
adding: test/file2 (deflated 4%)
adding: test/myprog.gz (stored 0%)
$
```

这个例子创建了一个叫testzip.zip的zip文件，并递归目录test把找到的每个文件和目录都加进该zip文件。从输出可以看到，不是所有存进该zip文件的文件都能够被压缩。Zip工具自动决定针对每个单独文件的压缩类型。

警告 当使用zip命令的递归特性时，在压缩过的zip文件中文件依然存储在相同的目录结构中。在压缩过的zip文件中子目录中的文件依然在同样的子目录下。所以在解压时要分外小心，unzip命令会在新的位置重新构建整个目录结构。在当前目录下有很多文件和子目录时，这个会有点叫人头大。

4.3.4 归档数据

虽然zip命令能够很好地将数据压缩和归档进单个文件，但它不是Unix和Linux中的标准归档工具。目前，Unix和Linux上最广泛使用的归档工具是tar命令。

tar命令最开始是用来将文件写到磁带设备上归档的，然而它也能把输出写到文件里，这种用法在Linux上已经普遍用来归档数据了。

下面是tar命令的格式：

```
tar function [options] object1 object2 ...
```

function参数定义了tar命令应该做什么，如表4-9所示。

表4-9 tar命令的功能

功 能	全字名称	描 述
-A	--concatenate	将一个已有tar归档文件追加到另一个已有tar归档文件
-c	--create	创建一个新的tar归档文件
-d	--diff	检查归档文件和文件系统的不同之处
	--delete	从已有tar归档文件中删除
-r	--append	追加文件到已有tar归档文件末尾
-t	--list	列出已有tar归档文件的内容
-u	--update	将比tar归档文件中已有的同名文件新的文件追加到该tar归档文件中
-x	--extract	从已有tar归档文件中提取文件

每个功能可用选项来针对tar归档文件定义一个特定行为。表4-10列出了这些选项中能和tar命令一起使用的最常用的选项。

表4-10 tar命令选项

选 项	描 述
-C dir	切换到指定目录
-f file	输出结果到文件或设备file
-j	将输出重定向给bzip2命令来压缩内容
-p	保留所有文件权限
-v	在处理文件时显示文件
-Z	将输出重定向给gzip命令来压缩内容

这些选项经常合并到一起使用。首先，你可以用这个命令来创建一个归档文件：

```
tar -cvf test.tar test/ test2/
```

上面的命令创建了一个含有test和test2目录内容的叫test.tar的归档文件。接着，用这个命令：

```
tar -tf test.tar
```

列出tar文件test.tar的内容，但并不解压文件。最后，用命令

```
tar -xvf test.tar
```

来从tar文件test.tar中提取内容。如果tar文件是从一个目录结构创建的，那整个目录结构都会在当前目录下重新创建。

如你所见，tar命令是给整个目录结构创建归档文件的简便方法。这是Linux中分发开源程序源码文件采用的普遍方法。

提示 下载了开源软件之后，你会经常看到文件名以.tgz结尾。这些是gzip压缩过的tar文件，可以用命令tar -zxvf filename.tgz来解压。

4.4 小结

本章讨论了Linux系统管理员和程序员用到的一些高级bash命令。ps和top命令在判断系统的状态时特别重要，能看到哪些应用在运行以及它们消耗了多少资源。

在可移动存储普遍的今天，另外一个系统管理员的普遍话题是挂载存储设备。mount命令可以将一个物理存储设备挂载到Linux虚拟目录结构上。umount命令用来移除设备。

最后，本章讨论了各种处理数据的工具。sort工具能轻松地对大数据文件进行排序，方便你组织数据；grep实用程序能快速检索大数据文件来查找特定信息。Linux上有一些不同的文件压缩工具，包括bzip2、gzip和zip。每种工具都允许在文件系统上压缩大型文件来节省空间。tar工具能将整个目录都归档到单个文件中，方便把数据迁移到另外一个系统上。

后面一章将讨论Linux环境变量。环境变量允许你在脚本中访问系统信息，同时提供一个在脚本中存储数据的简便方法。



本章内容

- 什么是环境变量
- 设置环境变量
- 删除环境变量
- 默认shell环境变量
- 设置PATH环境变量
- 定位系统环境变量
- 可变数组
- 使用命令别名

Linux环境变量能帮你提升Linux shell体验。但对新手来说，这是一个容易感到困惑的话题。很多程序和脚本都通过环境变量来获取系统信息、存储临时数据和配置信息。在Linux系统上有很多地方可以设置环境变量，了解去哪里设置相应的环境变量很重要。本章将带你逐步了解Linux环境变量：它们存储在哪里，怎样使用，以及怎样创建自己的环境变量。本章将以一个相关的话题——在shell会话中定义和使用别名（alias）收尾。

5.1 什么是环境变量

bash shell用一个称作环境变量（environment variables）的特性来存储有关shell会话和工作环境的信息。这也是它们为什么被称作环境变量的原因。它允许你在内存中存储数据，以便运行在shell上的程序和脚本访问。这也是存储永久数据的一种简便方法，这些数据可以是用来识别用户账户、系统、shell的特性以及任何其他你需要存储的数据。

在bash shell中，环境变量分为两类：

- 全局变量；
- 局部变量。

本节将描述环境变量的每一种类型，并演示怎么查看和使用它们。

说明 尽管bash shell使用一致的专有环境变量值，但不同的Linux发行版经常会添加它们自有的环境变量。你在本章中看到的环境变量的例子可能会跟你安装的发行版中看到的结果略微不同。如果遇到有关本书未讲到的环境变量的问题，你可以查看你的Linux发行版上的文档。

5.1.1 全局环境变量

全局环境变量不仅对shell会话可见，对所有shell创建的子进程也可见。局部变量则只对创建它们的shell可见。这让全局环境变量对那些子进程中需要获得父进程信息的程序来说非常有用。

Linux系统在你开始bash会话之前就设置了一些全局环境变量（如想了解此时哪些变量已被设置了，参见5.6节）。系统环境变量一律使用全大写字母以区别于普通用户的环境变量。

查看全局变量，可用printenv命令：

```
$ printenv
ORBIT_SOCKETDIR=/tmp/orbit-user
HOSTNAME=localhost.localdomain
IMSETTINGS_INTEGRATE_DESKTOP=yes
TERM=xterm
SHELL=/bin/bash
_XDG_SESSION_COOKIE=787b3cf537971ef8462260960000006b-1284670942.440386-1012435051
HISTSIZE=1000
GTK_RC_FILES=/etc/gtk gtkrc:/home/user/.gtkrc-1.2-gnome2
WINDOWID=29360131
GNOME_KEYRING_CONTROL=/tmp/keyring-Egjauk
IMSETTINGS_MODULE=none
USER=user
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:
bd=40;33:01:cd=40;33:01:or=40;31:01:mi=01;05:37;41:su=37;41:sg=30;43
...
SSH_AUTH_SOCK=/tmp/keyring-Egjauk/ssh
SESSION_MANAGER=local:unix:@/tmp/.ICE-unix/1331,unix/unix:/tmp/.ICE-unix/1331
USERNAME=user
DESKTOP_SESSION=gnome
MAIL=/var/spool/mail/user
PATH=/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin
QT_IM_MODULE=xim
PWD=/home/user/Documents
XMODIFIERS=@im=none
GDM_KEYBOARD_LAYOUT=us
LANG=en_US.UTF8
GNOME_KEYRING_PID=1324
GDM_LANG=en_US.UTF8
GDMSESSION=gnome
SSH_ASKPASS=/usr/libexec.openssh/gnome-ssh-askpass
HISTCONTROL=ignoredups
HOME=/home/user
```

```

SHLVL=2
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=user
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
ifKtHnDKnU.guid=6f061c07be822c134f12956b00000081
LESSOPEN=| /usr/bin/lesspipe.sh %
WINDOWPATH=1
DISPLAY=:0.0
G_BROKEN_FILERAMES=1
XAUTHORITY=/var/run/gdm/auth-for-user-GIU7sE/database
COLORTERM=gnome-terminal
_= /usr/bin/printenv
OLDPWD=/home/user
$
```

如你所见，系统为bash shell设置了很多全局环境变量。它们中的大部分都是系统在用户登录系统时设置的。

要显示单个环境变量的值，可用echo命令。当引用环境变量时，必须在环境变量的名称前放置一个\$符：

```

$ echo $HOME
/home/user
$
```

正如前面提到的，全局环境变量在当前shell会话的子进程中也是可见的：

```

$ bash
$ echo $HOME
/home/user
$
```

在这个例子中，用bash命令启动一个新的shell后，显示了HOME环境变量的当前值，这个值是在你登录进主shell时设定的。当然，这个值在子shell进程中依然是存在的。

5.1.2 局部环境变量

顾名思义，局部环境变量只能在定义它们的进程中可见。尽管它们是局部的，却和全局环境变量一样重要。事实上，Linux系统也默认定义了标准局部环境变量。

查看局部环境变量的列表有点复杂。遗憾的是，在Linux系统并没有这样一个命令只显示局部环境变量。set命令会显示为某个特定进程设置的所有环境变量。当然，这也包括全局环境变量。

下面是用来示例的set命令的输出：

```

$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:
force_fignore:hostcomplete:interactive_comments:
progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
```

```

BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="1" [2]="7" [3]="1"
[4]=""release" [5]=""1386-redhat-linux-gnu")
BASH_VERSION='4.1.7(1)-release'
COLORS=/etc/DIR_COLORS
COLORTERM=gnome-terminal
COLUMNS=80
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
1fkHnDKnU,guid=6f061c07be822c134f12956b00000081
DESKTOP_SESSION=gnome
DIRSTACK=()
DISPLAY=:0.0
EUID=500
GOMSESSION=gnome
GDM_KEYBOARD_LAYOUT=us
GDM_LANG=en_US.UTF8
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_KEYRING_CONTROL=/tmp/keyring-Egjauk
GNOME_KEYRING_PID=1324
GROUPS=()
GTK_RC_FILES=/etc/gtk gtkrc:/home/user/.gtkrc-1.2-gnome2
G_BROKEN_FILERAMES=1
HISTCONTROL=ignore_dups
HISTFILE=/home/user/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/home/user
HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS=$' \t\n\r'
IMSETTINGS_INTEGRATE_DESKTOP=yes
IMSETTINGS_MODULE=none
LANG=en_US.UTF8
LESSOPEN='| /usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=user
LS_COLORS="rs=0;di=01:34:ln=01:36:mh=00:pi=40:33:so=01:35:do=01:35:
bd=40:33:01:cd=40:33:01:or=40:31:01:mn=01:05:37:41:su=37:41:sg=30:43
...
MACHETYPE=i386-redhat-linux-gnu
MAIL=/var/spool/mail/user
MAILCHECK=60
OLDPWD=/home/user
OPTERR=1
OPTIND=1
ORBIT_SOCKETDIR=/tmp/orbit-user
OSTYPE=linux-gnu
PATH=/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin
PIPESTATUS=(0)=""0"
PPID=1674
PROMPT_COMMAND='echo -ne .\033]0;${USER}@${HOSTNAME}%%.*':
```

```

${PWD/#$HOME/-}"; echo -ne '\007'
PS1=[\u@\h \W]\$ .
PS2=> '
PS4='+ '
PWD=/home/user/Documents
QT_IM_MODULE=xim
SESSION_MANAGER=local/unix:@/tmp/.ICE-unix/1331,unix/unix:/tmp/.ICE-unix/1331
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor
SHLVL=2
SSH_ASKPASS=/usr/libexec.openssh/gnome-ssh-askpass
SSH_AUTH_SOCK=/tmp/keyring-Egjauk/ssh
TERM=xterm
UID=500
USER=user
USERNAME=user
WINDOWID=29360131
WINDOWPATH=1
XAUTHORITY=/var/run/gdm/auth-for-user-GIU7sE/database
XDG_SESSION_COOKIE=787b3cf537971ef8462260960000006b-1284670942.440386-1012435051
XMODIFIERS=@im=none
_=printenv
colors=/etc/DIR_COLORS
_udisks ()
{
    local IFS=
    :
    local cur="${COMP_WORDS[COMP_CWORD]}";
    if [ "${COMP_WORDS[$((COMP_CWORD - 1))]}" = "--show-info" ]; then
        COMPREPLY=(${compgen -W "$(_udisks --enumerate-device-files)" -- $cur});
    else
        if [ "${COMP_WORDS[$((COMP_CWORD - 1))]}" = "--inhibit-polling" ]; then
            :
            fi;
            fi;
            fi;
            fi;
    }
command_not_found_handle ()
{
    runcnf=1;
    retval=127;
    [ ! -S /var/run/dbus/system_bus_socket ] && runcnf=0;
    [ ! -x /usr/sbin/packagekitd ] && runcnf=0;
    if [ $runcnf -eq 1 ]; then
        /usr/libexec/pk-command-not-found $1;
        retval=$?;
    else
        echo "bash: $1: command not found";
        fi;
    return $retval
}
$
```

可以看到，所有通过printenv命令能看到的全局环境变量都出现在了set命令的输出中。但在set命令的输出中还有一些其他的环境变量，这些就是局部环境变量。

5.2 设置环境变量

你可以在bash shell中直接设置自己的环境变量。本节将介绍怎样在交互式shell或shell脚本程序中创建自己的环境变量并引用它们。

5.2.1 设置局部环境变量

一旦启动了bash shell（或者执行一个shell脚本），你就能创建在这个shell进程中可见的局部变量了。你可以通过等号来给环境变量赋值，值可以是数值或字符串：

```
$ test=testing  
$ echo $test  
testing  
$
```

非常简单！现在每次引用test环境变量的值，只要用\$test引用即可。

如果要给变量赋一个含有空格的字符串值，必须用单引号来界定字符串的开始和末尾：

```
$ test='testing a long string'  
-bash: a: command not found  
$ test='testing a long string'  
$ echo $test  
testing a long string  
$
```

没有单引号的话，bash shell会以为下个字符串是另一个要执行的命令。注意，你定义的局部环境变量用的是小写字母，而到目前为止你所看到的系统环境变量都是用大写字母。

这是bash shell的标准惯例。创建新的环境变量时，推荐你用小写字母。它能帮助你区分用户个人环境变量和系统环境变量。

警告 记住，在环境变量名称、等号和值之间没有空格，这一点非常重要。如果你在赋值表达式中放了空格，bash shell就会把值当成一个单独的命令：

```
$ test2 = test  
-bash: test2: command not found  
$
```

设置了局部环境变量后，就能在shell进程的任何地方使用它了。但是，如果创建了另外一個shell，它在子shell中就不可用了：

```
$ bash  
$ echo $test  
$ exit
```

```
exit  
$ echo $test  
testing a long string  
$
```

在这个例子中，你启动了一个子shell。如你所见，test环境变量在子shell中并不可见（它含的是一个空值）。当你退出子shell回到原来的shell时，局部环境变量依然在。

类似地，如果你在子进程中设置了一个局部环境变量，一旦你退出了子进程，那个局部环境变量就不能用了：

```
$ bash  
$ test=testing  
$ echo $test  
testing  
$ exit  
exit  
$ echo $test  
  
$
```

当我们回到父shell时，子shell中设置的test环境变量就不再存在了。

5.2.2 设置全局环境变量

全局环境变量在设定该全局环境变量的进程创建的所有子进程中都是可见的。创建全局环境变量的方法是先创建一个局部环境变量，然后再把它导出到全局环境中。

这个过程通过export命令来完成：

```
$ echo $test  
testing a long string  
$ export test  
$ bash  
$ echo $test  
testing a long string  
$
```

导出局部环境变量test后，我们启动了子shell进程并在子shell中查看了test环境变量的值。这次，因为export命令让它变成了全局的，环境变量保持了它的值。

警告 注意在导出局部环境变量时，不能用美元符 (\$) 去引用变量名。

5.3 删除环境变量

当然，既然可以创建一个新的环境变量，删除一个已经存在的环境变量也就有了用。你可以用unset命令来删除环境变量：

```
$ echo $test
testing
$ unset test
$ echo $test

$
```

在unset命令中引用环境变量时，记住不要用美元符（\$）。

在处理全局环境环境变量时，事情就有点复杂了。如果你是在子进程中删除了一个全局环境变量，它只对子进程有效。该全局环境变量在父进程中依然有效：

```
$ test=testing
$ export test
$ bash
$ echo $test
testing
$ unset test
$ echo $test

$ exit
exit
$ echo $test
testing
$
```

在这个例子中，你设置了一个局部环境变量叫test，然后把它导出，使它成了一个全局变量。紧接着你又启动了一个子shell并且查看了全局环境变量test确认它在子shell中是存在的。之后，仍是在子shell中，你用unset命令删除了全局环境变量test并退出了子shell。回到原来的父shell时，检查test环境变量的值，它依然是有效的。

5.4 默认 shell 环境变量

默认情况下bash shell会用一些特定的环境变量来定义系统环境。你可以一直使用Linux系统上默认定义的这些变量。bash shell是从最开始的Unix Bourne shell衍生出来的，所以它保留了Unix Bourne shell里定义的那些环境变量。

表5-1列出了bash shell提供的跟Unix Bourne shell兼容的环境变量。

表5-1 bash shell支持的Bourne变量

变量	描述
CDPATH	冒号分隔的目录列表，作为cd命令的搜索路径
HOME	当前用户的主目录
IFS	shell用来分隔文本字符串的一列字符
MAIL	当前用户收件箱的文件名；bash shell会检查这个文件来看有没有新邮件
MAILPATH	冒号分隔的当前用户收件箱的文件名列表；bash shell会检查列表中的每个文件来看有没有新邮件
OPTARG	getopts命令处理的最后一个选项参数值
OPTIND	getopts命令处理的最后一个选项参数的索引号

(续)

变 量	描 述
PATH	冒号分隔的shell查找命令的目录列表
PS1	shell命令行界面的主要提示符
PS2	shell命令行界面的次提示符

到目前位置这个列表中最重要的环境变量就是PATH环境变量。在shell命令行界面 (Command Line Interface, CLI) 输入命令时, shell必须在系统中查找程序。在我的Linux系统上, PATH环境变量是这样的:

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin
```

\$

这说明shell将在6个目录中查找命令。PATH中的每个目录都由冒号分隔。在PATH变量的末尾没什么特殊符号来说明这是目录列表的结尾。你可以通过在PATH变量的末尾加个冒号再加上新的目录来添加其他目录。PATH变量同时显示了shell查找命令的顺序。

除了默认的Bourne的环境变量, bash shell还提供一些自有的变量, 如表5-2所示。

表5-2 bash shell环境变量

变 量	描 述
BASH	运行当前shell实例的全路径名
BASH_ALIASES	当前已设置别名的关联数组
BASH_ARGC	含有传给子函数或shell脚本的参数总数的可变数组
BASH_ARGV	含有传给子函数或shell脚本的参数的可变数组
BASH_CMDS	shell执行过的命令的所在位置的关联数组
BASH_COMMAND	shell正在执行的命令或马上就执行的命令
BASH_ENV	设置了的话, 每个bash脚本会在运行前先尝试运行一下这个变量定义的启动文件
BASH_EXECUTION_STRING	通过bash -c选项传递过来的命令
BASH_LINENO	含有当前执行的shell函数在源代码中行号的可变数组
BASH_REMATCH	含有模式和它们通过正则表达式比较运算符“=~”匹配到的子模式的只读可变数组
BASH_SOURCE	含有当前正在执行的shell函数的源码文件名的可变数组
BASH_SUBSHELL	当前子shell环境的嵌套级别, 初始值是0
BASH_VERSION	当前运行的bash shell的版本号
BASH_VERSINFO	含有当前运行的bash shell的主版本号和次版本号的可变数组
BASH_XTRACEFD	若设置成了有效的文件描述符(0,1,2), 则“set -x”调试选项声称的跟踪输出可被重定向; 通常用来将跟踪输出分出到一个文件中
BASHOPTS	当前使用的bash shell选项的列表
BASHPID	当前bash进程的PID
COLUMNS	当前bash shell实例所用终端的宽度

(续)

变 量	描 述
COMP_CWORD	含有当前光标位置的COMP_WORDS变量的索引值
COMP_LINE	当前命令行
COMP_POINT	当前光标位置相对于当前命令起始位置的索引
COMP_KEY	用来调用shell函数补全功能的最后一个键值
COMP_TYPE	代表尝试调用补全shell函数的补全类型的整数值
COMP_WORDBREAKS	Readline库里做单词补全的词分隔字符
COMP_WORDS	含有当前命令行所有词的可变数组
COMPREPLY	含有由shell函数生成可能的填充字的可变数组
DIRSTACK	含有目录栈当前内容的可变数组
EMACS	设置为't'时, 表明emacs shell缓冲区正在工作而行编辑不能工作
EUID	当前用户的有效用户ID
FCEDIT	供fc命令用的默认编辑器
FIGNAME	冒号分隔的做文件名补全时要忽略的后缀名列表
FUNCNAME	当前执行的shell函数的名称
GLOBIGNORE	定义了文件名展开时忽略的文件名集合的冒号分隔的模式列表
GROUPS	含有当前用户属组列表的可变数组
histchars	控制历史记录展开的字符, 最多可有3个字符
HISTCMD	当面命令在历史记录中的位置
HISTCONTROL	控制哪些命令留在历史记录列表中
HISTFILE	保存shell历史记录列表的文件名 (默认是.bash_history)
HISTFILESIZE	最多在历史文件中存多少行
HISTIGNORE	冒号分隔的用来决定哪些命令不存进历史文件的模式列表
HISTSIZE	最多在历史文件中存多少条命令
HOSTFILE	shell在补全主机名时读取的文件的名称
HOSTNAME	当前主机的名称
HOSTTYPE	当前运行bash shell的机器
IGNOREEOF	shell在退出前必须收到连续的EOF字符的数量。如果这个值不存在, 默认是1
INPUTRC	Readline初始化文件名 (默认是.inputrc)
LANG	shell的语言环境分类
LC_ALL	定义一个语言环境, 覆盖LANG变量
LC_COLLATE	设置对字符串排序时用的对照表顺序
LC_CTYPE	决定着在文件名展开和模式匹配时用字符如何解释
LC_MESSAGES	决定解释前置美元符 (\$) 的双引号字符串的语言环境设置
LC_NUMERIC	决定着格式化数字时的语言环境设置
LINENO	当前执行的脚本的行号
LINES	定义了终端上可见的行数
MACHTYPE	用“cpu-公司-系统”格式定义的系统类型
MAILCHECK	shell查看新邮件的频率 (以秒为单位, 默认值是60)

(续)

变 量	描 述
OLDPWD	shell之前的工作目录
OPTERR	设置为1时，bash shell会显示getopts命令产生的错误
OSTYPE	定义了shell运行的操作系统
PIPESTATUS	含有前端进程的退出状态列表的可变数组
POSIXLY_CORRECT	设置了的话，bash会以POSIX模式启动
PPID	bash shell父进程的PID
PROMPT_COMMAND	设置了的话，在命令行主提示符显示之前会执行这条命令
PROMPT_DIRTRIM	用来定义当启用了\w或\W提示符字符串转义时显示的尾部目录名数。删除的目录名会用一组英文句点替换
PS3	select命令的提示符
PS4	如果使用了bash的-x参数，在命令行显示之前显示的提示符
PWD	当前工作目录
RANDOM	返回一个0-32767的随机数；对其赋值可作为随机数生成器的种子
REPLY	read命令的默认变量
SECONDS	自从shell启动到现在的秒数；对其赋值将会重置计数器
SHELL	bash shell的全路径名
SHLOPTS	冒号分隔的打开的bash shell选项列表
SHLVL	shell的级别：每次启动一个新bash shell，值增加1
TIMEFORMAT	指定了shell显示时间值的格式
TMOUT	select和read命令在没输入的情况下等待多久（以秒为单位）。默认值为零，表示无限长
TMPDIR	bash shell创建临时文件的目录名
UID	当前用户的真实用户ID

5

你可能已经注意到，不是所有的默认环境变量都会在运行set命令时列出。尽管这些是默认环境变量，但不是它们所有都必须有一个值。

5.5 设置 PATH 环境变量

PATH环境变量是Linux系统上造成最多问题的变量。它定义了命令行输入命令的搜索路径。如果找不到命令，它会产生一个错误：

```
$ myprog
-bash: myprog: command not found
$
```

问题是通常应用会把可执行程序放到不在PATH环境变量中的目录。解决的办法是保证PATH环境变量包含了所有存放应用的目录。

你可以添加新的搜索目录到现有的PATH环境变量，无需从头定义。PATH中的目录之间是用冒号分隔的，所以你只需引用原来的PATH值，然后再给字符串添加新目录就行了。可以参考下面的

例子：

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin

$ PATH=$PATH:/home/user/test
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin:/home/user/test
$ myprog
The factorial of 5 is 120.
$
```

将目录加到PATH环境变量之后，现在可以在虚拟目录结构的任何位置执行程序了：

```
[user@localhost ~]$ cd /etc
[user@localhost etc]$ myprog
The factorial of 5 is 120
[user@localhost etc]$
```

程序员通常用的办法是将单点符也加到PATH环境变量里。这个单点符代表当前目录（参见第3章）：

```
[user@localhost ~]$ PATH=$PATH:.
[user@localhost ~]$ cd test2
[user@localhost test2]$ myprog2
The factorial of 6 is 720
[user@localhost test2]$
```

下节，你会了解到如何修改环境变量使其能一直在你的系统上，这样你就能一直执行你的程序了。

5.6 定位系统环境变量

Linux系统用环境变量来在程序和脚本中标识它自己。这为你的程序提供了获得系统信息的一个简便办法。问题是如何设置这些变量。

在你登录Linux系统启动一个bash shell时，默认情况下bash在几个文件中查找命令。这些文件称作启动文件。bash检查的启动文件取决于你启动bash shell的方式。启动bash shell有3种方式：

- 登录时当做默认登录shell；
- 作为非登录shell的交互式shell；
- 作为运行脚本的非交互shell。

下面几节介绍了bash shell在不同的启动方式下检查的启动文件。

5.6.1 登录shell

当你登录Linux系统时，bash shell会作为登录shell启动。登录shell会从4个不同的启动文件里读取命令。下面是bash shell处理这些文件的次序：

- /etc/profile;
- \$HOME/.bash_profile;
- \$HOME/.bash_login;
- \$HOME/.profile。

/etc/profile文件是系统上默认的bash shell的主启动文件。系统上的每个用户登录时都会执行这个启动文件。另外3个启动文件是用户专有的，所以可根据每个用户的具体需求定制。我们来仔细看一下各个文件。

1. /etc/profile文件

/etc/profile文件是bash shell的主启动文件。只要你登录了Linux系统，bash就会执行/etc/profile文件中的命令。不同的Linux发行版在这个文件里放了不同的命令。在这个Linux系统上，它看起来是这样的：

```
$ cat /etc/profile
# /etc/profile

# System wide environment and startup programs. for login setup
# Functions and aliases go in /etc/bashrc

# It's NOT good idea to change this file unless you know what you
# are doing. Much better way is to create custom.sh shell script in
# /etc/profile.d/ to make custom changes to environment. This will
# prevent need for merging in future updates.

pathmunge () {
    case ":${PATH}:" in
        *:"$1":*)
            ;;
        *)
            if [ "$2" = "after" ] ; then
                PATH=$PATH:$1
            else
                PATH=$1:$PATH
            fi
    esac
}

if [ -x /usr/bin/id ]; then
    if [ -z "$EUID" ]; then
        # ksh workaround
        EUID=`id -u`
        UID=`id -ru`
    fi
    USER=`id -un`
    LOGNAME=$USER
    MAIL="/var/spool/mail/$USER"
fi

# Path manipulation
```

```

if [ "$EUID" = "0" ]; then
    pathmunge /sbin
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
else
    pathmunge /usr/local/sbin after
    pathmunge /usr/sbin after
    pathmunge /sbin after
fi

HOSTNAME='/bin/hostname 2>/dev/null'
HISTSIZE=1000
if [ "$HISTCONTROL" = "ignorespace" ] ; then
    export HISTCONTROL=ignoreboth
else
    export HISTCONTROL=ignoredups
fi

export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL

for i in /etc/profile.d/*.sh ; do
    if [ -r "$i" ]; then
        if [ "$PS1" ]; then
            . $i
        else
            . $i >/dev/null 2>&1
        fi
    fi
done

unset i
unset pathmunge
$
```

这个文件中你能看到的大部分命令和脚本都会在第10章具体讲到。现在重要的是留意一下这个文件中设置的环境变量。看一下文件底部的导出行：

```
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL
```

这保证了这些环境变量对这个登录shell创建的所有子进程都有效。

`profile`文件还有一个复杂的特性。它有个`for`语句，会逐一访问位于`/etc/profile.d`目录的每个文件（`for`语句我们会在第12章中讨论）。它为Linux系统提供了一个集中存放用户登录时要执行的应用专属的启动文件的地方。在这个Linux系统上，`profile.d`目录下有如下文件：

```
$ ls -l /etc/profile.d
total 72
-rw-r--r--. 1 root root 1133 Apr 28 10:43 colorls.csh
-rw-r--r--. 1 root root 1143 Apr 28 10:43 colorls.sh
-rw-r--r--. 1 root root 192 Sep  9  2004 glib2.csh
-rw-r--r--. 1 root root 192 Dec 12  2005 glib2.sh
-rw-r--r--. 1 root root  58 May 31 06:23 gnome-ssh-askpass.csh
-rw-r--r--. 1 root root  70 May 31 06:23 gnome-ssh-askpass.sh
```

```
-rw-r--r--. 1 root root 184 Aug 25 11:36 krb5-workstation.csh
-rw-r--r--. 1 root root 57 Aug 25 11:36 krb5-workstation.sh
-rw-r--r--. 1 root root 1741 Jun 24 15:20 lang.csh
-rw-r--r--. 1 root root 2706 Jun 24 15:20 lang.sh
-rw-r--r--. 1 root root 122 Feb 7 2007 less.csh
-rw-r--r--. 1 root root 108 Feb 7 2007 less.sh
-rw-r--r--. 1 root root 837 Sep 2 05:24 PackageKit.sh
-rw-r--r--. 1 root root 2142 Aug 10 16:41 udisks-bash-completion.sh
-rw-r--r--. 1 root root 74 Mar 25 19:24 vim.csh
-rw-r--r--. 1 root root 248 Mar 25 19:24 vim.sh
-rw-r--r--. 1 root root 161 Nov 27 2007 which2.csh
-rw-r--r--. 1 root root 169 Nov 27 2007 which2.sh
$
```

不难发现，这些基本都跟系统上的特定应用有关。大部分应用会创建两个启动文件：一个给 bash shell（使用.sh扩展名），一个给c shell（使用.csh扩展名）。

lang.csh和lang.sh文件会尝试去判定系统上所采用的默认语言文字集，然后正确地设置LANG环境变量。

2. \$HOME目录下的启动文件

剩下的3个启动文件都起着同一个作用：提供一个用户专属的启动文件来定义用户专有的环境变量。大多数Linux发行版只用这3个启动文件中的一个：

- \$HOME/.bash_profile;
- \$HOME/.bash_login;
- \$HOME/.profile。

注意这3个文件都以点开头，这说明它们是隐藏文件（不会在通常的ls命令输出列表中出现）。它们在用户的HOME目录下，所以每个用户可以编辑这些文件并添加自己的环境变量来给他们启动的每个bash shell会话用。

这个Linux系统的.bash_profile文件的内容如下：

```
$ cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
```

.bash_profile启动文件会先去检查HOME目录中是不是还有另一个叫.bashrc的启动文件（我们将会在5.6.2节中介绍）。如果有的话，启动文件会先去执行它里面的命令。下一步，启动文件将一个目录加到了PATH环境变量，在HOME目录下提供了一个放置可执行文件的通用位置。

5.6.2 交互式shell

如果你的bash shell不是登录系统时启动的（比如你在命令行提示符下敲入bash启动），你启动的shell称作交互式shell。交互式shell不会像登录shell一样运行，但它依然提供了命令行提示符来输入命令。

如果bash是作为交互式shell启动的，它不会去访问/etc/profile文件，而会去用户的HOME目录检查.bashrc是否存在。

在这个Linux系统上，这个文件看起来如下：

```
$ cat .bashrc
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
$
```

.bashrc文件有两个作用：一是查看/etc目录下的共用bashrc文件，二是为用户提供了一个定制自己的命令别名（将在5.8节中讨论）和私有脚本函数（将在第16章中讲到）的地方。

通用的/etc/bashrc启动文件会被系统上每个启动交互式shell会话的用户执行。在这个Linux系统上它看上去是这样的：

```
$ cat /etc/bashrc
# /etc/bashrc

# System wide functions and aliases
# Environment stuff goes in /etc/profile

# It's NOT good idea to change this file unless you know what you
# are doing. Much better way is to create custom.sh shell script in
# /etc/profile.d/ to make custom changes to environment. This will
# prevent need for merging in future updates.

# By default, we want this to get set.
# Even for non-interactive, non-login shells.
# Current threshold for system reserved uid/gids is 200
# You could check uidgid reservation validity in
# /usr/share/doc/setup/*uidgid file
if [ $UID -gt 199 ] && [ `id -gn` = `id -un` ]; then
    umask 002
else
    umask 022
fi

# are we an interactive shell?
if [ "$PS1" ]; then
    case $TERM in
```

```
xterm*)
    if [ -e /etc/sysconfig/bash-prompt-xterm ]; then
        PROMPT_COMMAND=/etc/sysconfig/bash-prompt-xterm
    else
        PROMPT_COMMAND='echo -ne
"\033[0;${USER}@${HOSTNAME%.*}:
${PWD/#$HOME/-}"; echo -ne "\007"'
        fi
    ;;
screen)
    if [ -e /etc/sysconfig/bash-prompt-screen ]; then
        PROMPT_COMMAND=/etc/sysconfig/bash-prompt-screen
    else
        PROMPT_COMMAND='echo -ne
"\033[${USER}@${HOSTNAME%.*}:
${PWD/#$HOME/-}"; echo -ne "\033\\"
        fi
    ;;
*)
    [ -e /etc/sysconfig/bash-prompt-default ] &&
PROMPT_COMMAND=/etc/sysconfig/bash-prompt-default
    ;;
esac
# Turn on checkwinsize
shopt -s checkwinsize
[ "$PS1" = "\s-\v\$ " ] && PS1="[\u@\h \W]\$ "
# You might want to have e.g. tty in prompt (e.g. more virtual machines)
# and console windows
# If you want to do so, just add e.g.
# if [ "$PS1" ]; then
#   PS1="[\u@\h:\t \W]\$ "
# fi
# to your custom modification shell script in /etc/profile.d/ directory
fi

if ! shopt -q login_shell ; then # We're not a login shell
    # Need to redefine pathmunge, it get's undefined at the end of /etc/profile
    pathmunge () {
        case ":${PATH}:" in
            *:$1:*)
                ;;
            *)
                if [ "$2" = "after" ] ; then
                    PATH=$PATH:$1
                else
                    PATH=$1:$PATH
                fi
        esac
    }
    # Only display echos from profile.d scripts if we are no login shell
    # and interactive - otherwise just process them to set envvars
    for i in /etc/profile.d/*.*; do
        if [ -r "$i" ]; then
```

```

if [ "$PS1" ]; then
    $1
else
    $1 >/dev/null 2>&1
fi
fi
done

unset i
unset pathmunge
fi
# vim:ts=4:sw=4
$
```

默认的文件会设置一些环境变量，但注意它并没有执行export命令让它们成为全局的。记住，交互式shell的启动文件只会在每次有新的交互式shell启动时才运行，因此任何子shell都会自动执行这个交互式shell的启动文件。

还能看出，/etc/bashrc文件也会执行位于/etc/profile.d目录下的那些应用专属的启动文件。

5.6.3 非交互式shell

最后，最后一种类型的shell是非交互式shell。系统执行shell脚本时用的就是这种shell。你不用担心它没有命令行提示符，但当你每次在系统上运行脚本时仍要运行特定的启动命令。

为了处理这种情况，bash shell提供了BASH_ENV环境变量。当shell启动一个非交互式shell进程时，它会检查这个环境变量来查看要执行的启动文件。如果有指定的，shell会执行文件里的命令。在这个Linux发行版里，默认情况下这个环境变量并未设置。

5.7 可变数组

环境变量很好的一个功能是它们可作为数组使用，数组是能够存储多个值的变量。值可按单个值或整个数组来引用。

要给某个环境变量设置多个值，可以把值放在括号里，值与值之间用空格分隔：

```
$ mytest=(one two three four five)
$
```

没什么特别的地方。如果你想把数组当做普通的环境变量显示，你可能要失望了：

```
$ echo $mytest
one
$
```

只有数组的第一个值显示出来了。要引用一个单独的数组元素，你必须要用代表它在数组中位置的数值索引值。数值要用方括号括起来：

```
$ echo ${mytest[2]}
three
$
```

警告 环境变量数组的索引值都是从零开始。这通常会带来一些困惑。

要显示整个数组变量，可用星号作为通配符放在索引值的位置：

```
$ echo ${mytest[*]}\none two three four five\n$
```

你也可以改变某个索引值位置的值：

```
$ mytest[2]=seven\n$ echo ${mytest[*]}\none two seven four five\n$
```

你甚至能用unset命令来删除数组中的某个值，但是要小心，这可能会有点复杂。看下面的例子：

```
$ unset mytest[2]\n$ echo ${mytest[*]}\none two four five\n$ \n$ echo ${mytest[2]}\n$ echo ${mytest[3]}\nfour\n$
```

这个例子用unset命令来删除在索引值为2位置的值。显示整个数组时，看起来像是索引里面已经没这个索引了。但当专门显示索引值为2的位置的值时，能看到这个位置是空的。

最后，可以在unset命令后跟上数组名来删除整个数组：

```
$ unset mytest\n$ echo ${mytest[*]}\n$
```

有时可变数组会比让事情很麻烦，所以在shell脚本编程时不经常用。它们不跟其他shell环境通用，这在需要多种shell环境下编写大量脚本时会带来很多不便。bash系统环境变量中有很多都用数组（比如BASH_VERSINFO），但总体上说，你不会太经常用到。

5.8 使用命令别名

尽管不是标准的环境变量，shell命令别名的用法跟环境变量的用法差不多。命令别名允许为通用命令（和它们的参数一起）创建一个别名，这样就能通过最少的键入调用想要的命令了。

极有可能，你的Linux发行版已经设置了一些通用的命令别名。查看已有的别名列表，可以用alias命令加-p参数：

```
$ alias -p
alias l='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot
--show-tilde'
$
```

注意，在这个Linux发行版中，有个别名会覆盖已有的ls命令。它会自动提供--color参数，指定终端支持彩色模式列出。

你也可以用alias命令来创建自己的命令别名：

```
$ alias li='ls -ll'
$ li
total 32
75 drwxr-xr-x. 2 user user 4096 Sep 16 13:11 Desktop
79 drwxr-xr-x. 2 user user 4096 Sep 20 15:40 Documents
76 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Downloads
80 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Music
81 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Pictures
78 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Public
77 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Templates
82 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Videos
$
```

一旦定义了命令别名，你就能在任何时候在shell中使用了，包括shell脚本中。

命令别名的行为和局部环境变量差不多，它们通常只在定义它们的shell进程中有效：

```
$ alias li='ls -ll'
$ bash
$ li
bash: li: command not found
$
```

当然，现在你知道了解决这个的办法。bash shell在启动交互式shell时总会读取位于\$HOME/.bashrc的启动文件。那里是创建命令别名的好地方（如先前指出的.bashrc文件的注释里提到的）。

5.9 小结

本章介绍了Linux的环境变量。全局环境变量可以在定义它们的进程创建的子进程中使用。局部环境变量只能在定义它们的进程中使用。

Linux系统使用全局环境变量和局部环境变量来存储系统环境信息。你可以从shell的命令界面上使用这些信息，也可以在shell脚本中使用。bash shell支持最初Unix Bourne shell定义的系统环境变量，也支持很多新的环境变量。PATH环境变量定义了bash shell在查找可执行命令时的搜索目录。你可以修改PATH环境变量来添加自己的搜索目录，甚至是当前目录符号（“.”），来方便程序的运行。

你也可以创建自用的全局和局部环境变量。一旦创建了环境变量，在shell会话的整个过程中

它都是可访问的。

bash shell会在启动时执行几个启动文件。这些启动文件里有一些环境变量定义，它们会为每个bash会话定义标准环境变量。每次登录Linux系统时，bash shell都会访问/etc/profile启动文件，然后访问3个针对每个用户的本地启动文件：\$HOME/.bash_profile、\$HOME/.bash_login以及\$HOME/.profile。用户在这些文件中定制自己想要的环境变量和启动脚本。

bash shell还提供了环境变量数组。这些环境变量可在单个变量中包含多个值。你可以通过指定索引值来访问其中的单个值，或是通过环境变量数组名来引用所有的值。

最后，本章讨论了命令别名的使用。虽然不是环境变量，命令别名却和环境变量很像。它们允许你为一个命令（和它们的参数一起）定义一个别名。这样，你就能给一个很长的命令及其参数起一个简单的别名，方便在shell会话中使用。

下章将会介绍Linux文件的权限。对Linux新手来说，这可能是最难懂的内容。然而要写出好的脚本，你必须懂得文件权限是如何作用的以及如何在Linux系统中使用它们。



本章内容

- 理解Linux的安全性
- 解码文件权限
- 操作Linux组

缺乏安全性的系统不是完整的系统。系统上必须要有一套保护文件不被非授权用户访问或修改的机制。Linux沿用了Unix文件权限的办法，即允许用户和组基于每个文件和目录的一组安全性设置来访问文件。本章将介绍如何用Linux文件安全系统来在需要时共享数据和保护数据。

6.1 Linux 的安全性

Linux安全系统的核心是用户账户。每个能进入Linux系统的用户都会被分配一个唯一的用户账户。用户对系统上对象的访问权限取决于他们登录系统时用的账户。

用户权限是通过创建用户时分配的用户ID（User ID，通常缩写为UID）来跟踪的。UID是个数值，每个用户都有个唯一的UID。但在登录系统时不是用UID来登录，而是用登录名（login name）。登录名是用户用来登录系统的最长8字符的字符串（字符可以是数字或字母），同时会关联一个对应的密码。

Linux系统使用特定的文件和工具来跟踪和管理系统上的用户账户。在我们讨论文件权限之前，先来看一下Linux是怎样处理用户账户的。本节会介绍管理用户账户需要的文件和工具，这样在处理文件权限问题时，你就知道如何使用它们了。

6.1.1 /etc/passwd文件

Linux系统使用一个专门的文件来将用户的登录名匹配到对应的UID值。这个文件就是/etc/passwd文件，它包含了一些与用户有关的信息。下面是Linux系统上典型的/etc/passwd文件的一个例子：

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:./sbin/nologin
rpm:x:37:37:/var/lib/rpm:/sbin/nologin
vcspa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
mailnull:x:47:47::/var/spool/mqueue:/sbin/nologin
smmsp:x:51:51::/var/spool/mqueue:/sbin/nologin
apache:x:48:48:Apache:/var/www:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/sbin/nologin
ntp:x:38:38::/etc/ntp:/sbin/nologin
nscd:x:28:28:NSCD Daemon:/:/sbin/nologin
tcpdump:x:72:72::/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
avahi:x:70:70:Avahi daemon:/:/sbin/nologin
hsqldb:x:96:96::/var/lib/hsqldb:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
haldaemon:x:68:68:HAL daemon:/:/sbin/nologin
xfs:x:43:43:X Font Server:/etc/X11/fs:/sbin/nologin
gdm:x:42:42::/var/gdm:/sbin/nologin
rich:x:500:500:Rich Blum:/home/rich:/bin/bash
mama:x:501:501:Mama:/home/mama:/bin/bash
katie:x:502:502:katie:/home/katie:/bin/bash
jessica:x:503:503:Jessica:/home/jessica:/bin/bash
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
$
```

root用户账户是Linux系统的管理员，通常分配给它的UID是0。就像上例中显示的，Linux系统会为各种各样的功能创建不同的用户账户，而这些账户并不是真的用户。这些账户称作系统账户，是系统上运行的各种服务进程访问资源用的特殊账户。所有运行在后台的服务都需要用一个系统用户账户登录到Linux系统上。

在安全成为一个大问题之前，这些服务经常会用根账户登录。遗憾的是，如果有非授权的用户攻入了这些服务中的一个，他就能作为root用户进入整个系统了。为了防止这种情况发生，现在几乎每个Linux服务器上后台运行的服务都有自己的用户账户。这样，即使有人攻入了某个服务，他也无法访问整个系统。

Linux为系统账户预留了500以下的UID值。有些服务甚至要用特定的UID才能正常工作。为普通用户创建账户时，大多数Linux系统会将500起始的第一个可用UID分配给这个账户（这未必

适用于所有的Linux发行版)。

你可能已经注意到/etc/passwd文件中还有很多用户登录名和UID之外的信息。/etc/passwd文件的字段包含了如下信息：

- 登录用户名；
- 用户密码；
- 用户账户的UID；
- 用户账户的GID；
- 用户账户的文本描述(称为备注字段)；
- 用户HOME目录的位置；
- 用户的默认shell。

/etc/passwd文件中的密码字段都被设置成了x，这并不是说所有的用户账户都用相同的密码。在早期的Linux上，/etc/passwd文件里有加密后的用户密码。但鉴于很多程序都需要访问/etc/passwd文件获取用户信息，这就成了一个安全隐患。随着用来破解加密过的密码的工具的不断演进，用心不良的人开始忙于破解存储在/etc/passwd文件中的密码。Linux开发人员需要重新构思这个策略。

现在，Linux系统将用户密码保存在另一个单独的文件中(称为shadow文件，位置在/etc/shadow)。只有特定的程序才能访问这个文件，比如登录程序。

你已经看到了，/etc/passwd是标准的文本文件。你可以用任何文本编辑器来直接手动地在/etc/password文件里进行用户管理，比如添加、修改或删除用户账户。但这样做极其危险，如果/etc/passwd文件损坏了，系统就无法读取它的内容了，这样用户就无法正常登录了，甚至连root用户也会无法登录。用标准的Linux用户管理工具去执行这些用户管理功能就会安全许多。

6.1.2 /etc/shadow文件

/etc/shadow文件能对Linux系统如何管理密码有更多的控制。只有root用户才能访问/etc/shadow文件，这让它比起/etc/passwd来安全许多。

/etc/shadow文件为系统上的每个用户账户保存了一条记录。记录就像下面这样：

rich:\$1\$FfcK0ns\$f1UgiyHQ25wrB/hykCn020:11627:0:99999:7:::

在/etc/shadow文件的每条记录中有9个字段：

- 与/etc/passwd文件中的登录名对应的登录名；
- 加密后的密码；
- 自1970年1月1日(上次修改密码的日期)到当天的天数；
- 多少天后才能更改密码；
- 多少天后必须更改密码；
- 密码过期前多少天提醒用户更改密码；
- 密码过期后多少天禁用用户账户；

- 用户账户被禁用的日期，用自1970年1月1日到当天的天数表示；
- 预留字段，给将来使用。

使用shadow密码系统后，Linux系统可以更好地控制用户密码了。它可以控制用户多久更改一次密码，以及密码未更新的话多久后禁用该用户账户。

6.1.3 添加新用户

添加新用户到Linux系统的工具是useradd。这个命令提供了一次性创建新用户账户及设置用户HOME目录结构的简便方法。useradd命令使用系统的默认值以及命令行参数来设置用户账户。可以用useradd命令加-D参数来查看你的Linux系统的系统默认值：

```
# /usr/sbin/useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
#
```

6

说明 一些Linux发行版会把Linux用户和组工具放在/usr/sbin目录下，有可能不在PATH环境变量里。如果你的Linux系统是这样的话，可以将这个目录添加进PATH环境变量，或者用绝对文件路径名来运行这个程序。

-D参数显示了在创建新用户时如果你不在命令行指定的话useradd命令将使用的默认值。这个例子列出了这些默认值：

- 新用户会被添加到GID为100的公共组；
- 新用户的HOME目录将会位于/home/loginname；
- 新用户账户密码在过期后不会被禁用；
- 新用户账户未被设置为某个日期后就过期；
- 新用户账户将bash shell作为默认shell；
- 系统会将/etc/skel目录下的内容复制到用户的HOME目录下；
- 系统为该用户账户在mail目录下创建一个用于接收邮件的文件。

倒数第二个值很有意思。useradd命令允许管理员创建一份默认的HOME目录配置，然后把它作为创建新用户HOME目录的模板。这样，就能自动在每个新用户的HOME目录里放置默认的系统文件。在Ubuntu Linux系统上，/etc/skel目录有下列文件：

```
$ ls -al /etc/skel
total 32
drwxr-xr-x  2 root root  4096 2010-04-29 08:26 .
drwxr-xr-x 135 root root 12288 2010-09-23 18:49 ..
-rw-r--r--  1 root root   220 2010-04-18 21:51 .bash_logout
```

```
-rw-r--r-- 1 root root 3103 2010-04-18 21:51 .bashrc
-rw-r--r-- 1 root root 179 2010-03-26 08:31 examples.desktop
-rw-r--r-- 1 root root 675 2010-04-18 21:51 .profile
$
```

根据第5章的知识，可能你已经看出了这些文件是做什么的。它们是bash shell环境的标准启动文件。系统会自动将这些默认文件复制到你创建的每个用户的HOME目录。

你可以用默认系统参数创建一个新用户账户来试一下，并检查一下新用户的HOME目录：

```
# useradd -m test
# ls -al /home/test
total 24
drwxr-xr-x 2 test test 4096 2010-09-23 19:01 .
drwxr-xr-x 4 root root 4096 2010-09-23 19:01 ..
-rw-r--r-- 1 test test 220 2010-04-18 21:51 .bash_logout
-rw-r--r-- 1 test test 3103 2010-04-18 21:51 .bashrc
-rw-r--r-- 1 test test 179 2010-03-26 08:31 examples.desktop
-rw-r--r-- 1 test test 675 2010-04-18 21:51 .profile
#
```

默认情况下，useradd命令不会创建HOME目录，但是-m命令行选项会叫它创建HOME目录。你能在例子中看到，useradd命令创建了新的HOME目录，并将/etc/skel目录中的文件复制了过来。

说明 运行本章中提到的用户账户管理命令，需要以root用户账户登录或者通过sudo命令以root用户账户身份运行这些命令。

要想在创建用户时改变默认值或默认行为，可以使用命令行参数。表6-1列出了这些参数。

表6-1 useradd命令行参数

参数	描述
-c comment	给新用户添加备注
-d home_dir	为主目录指定一个名字（如果不想用登录名作为主目录名的话）
-e expire_date	用YYYY-MM-DD格式指定一个账户过期的日期
-f inactive_days	指定这个账户密码过期后多少天这个账户被禁用：0表示密码一过期就立即禁用，-1表示禁用这个功能
-g initial_group	指定用户登录组的GID或组名
-G group ...	指定用户除登录组之外所属的一个或多个附加组
-k	必须和-m一起使用，将/etc/skel目录的内容复制到用户的HOME目录
-m	创建用户的HOME目录
-M	不创建用户的HOME目录（当默认设置里指定创建时，才用到）
-n	创建一个同用户名同名的新组
-r	创建系统账户
-p passwd	为用户账户指定默认密码
-s shell	指定默认的登录shell
-u uid	为账户指定一个唯一的UID

你会发现，在创建新用户账户时使用命令行参数，可以更改系统指定的默认值。但如果发现你一直需要修改一个值时，最好修改系统的默认值。

你可以用-D参数后跟一个代表要修改的值的参数，来修改系统默认的新用户值。这些参数如表6-2所示。

表6-2 useradd更改默认值的参数

参数	描述
-b default_home	更改默认的创建用户HOME目录的位置
-e expiration_date	更改默认的新账户的过期日期
-f inactive	更改默认的新用户从密码过期到账户被禁用的天数
-g group	更改默认的组名称或GID
-s shell	更改默认的登录shell

更改默认值非常简单：

```
# useradd -D -s /bin/tsch
# useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/tsch
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
#
```

现在useradd命令会将tsch作为所有新建用户的默认登录shell。

6.1.4 删除用户

如果你想从系统中删除用户，userdel可以满足这个需求。默认情况下，userdel命令会只删除/etc/passwd文件中的用户信息，而不会删除系统中属于该账户的任何文件。

如果加上-r参数，userdel会删除用户的HOME目录以及mail目录。然而，系统上仍可能存有归已删除用户所有的其他文件。这在有些环境中会造成问题。

下面是用userdel命令删除已有用户账户的一个例子：

```
# /usr/sbin/userdel -r test
# ls -al /home/test
ls: cannot access /home/test: No such file or directory
#
```

加了-r参数后，用户之前的那个/home/test目录已经不存在了。

警告 在有大量用户的环境中使用-r参数时要特别小心。你永远不知道用户是否在他的HOME目录下存放了其他用户或其他程序要使用的重要文件。记住在删除用户的HOME目录之前一定要检查清楚！

6.1.5 修改用户

Linux提供了一些不同的工具来修改已有用户账户的信息。表6-3列出了这些工具。

表6-3 用户账户修改工具

命 令	描 述
usermod	修改用户账户的字段，并可以指定主要组以及附加组的所属关系
passwd	修改已有用户的密码
chpasswd	从文件中读取登录名密码对，并更新密码
chage	修改密码的过期日期
chfn	修改用户账户的备注信息
chsh	修改用户账户的默认登录shell

这些工具中的每个都会提供修改用户账户信息的一个专门的功能。下面的几节将详细介绍每一个工具。

1. usermod

usermod命令是用户账户修改工具中最强大的一个。它能用来修改/etc/passwd文件中的大部分字段，只需用与想修改的字段对应的命令行参数就可以了。参数大部分跟useradd命令的参数一样，比如-c用来修改备注字段，-e用来修改过期日期，-g用来修改默认的登录组。但还有一些实用的额外参数：

- -l用来修改用户账户的登录名；
- -L用来锁定账户，这样用户就无法登录了；
- -p用来修改账户的密码；
- -U用来解除锁定，解除后用户就能登录了。

-L参数尤其实用。用这个参数就把账户锁定，用户就无法登录了，而不用删除账户和用户的数据。要让账户恢复正常，只要加-U参数就行了。

2. passwd和chpasswd

改变用户密码的一个简便方法就是用passwd命令：

```
# passwd test
Changing password for user test.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
#
```

如果只用passwd命令，它会改变你自己的密码。系统上的任何用户都能改变他自己的密码，但只有root用户才有权限改变别人的密码。

-e选项能强制用户下次登录时修改密码。你可以先给用户设置一个简单的密码，之后再强制他们在下次登录时改成他们能记住的更复杂的密码。

如果需要为系统中的大量用户来修改密码，chpasswd命令能让事情简单许多。chpasswd命令能从标准输入自动读取登录名和密码对（由冒号分割）列表，给密码加密，然后为用户账户设置。你也可以用重定向命令来将含有*userid:passwd*对的文件重定向给命令：

```
# chpasswd < users.txt
#
```

3. chsh、chfn和chage

chsh、chfn和chage工具专门用来修改特定的账户信息。chsh命令用来快速修改默认的用户登录shell。使用时必须用shell的全路径名作为参数，不能只用shell名：

```
# chsh -s /bin/csh test
Changing shell for test.
Shell changed.
#
```

chfn命令提供了在/etc/passwd文件的备注字段中存储信息的标准方法。chfn命令会将Unix的finger命令用到的信息存进备注字段，而不是简单地存入一些随机文本（比如昵称之类的），或是将备注字段留空。finger命令可以用来简单地查看Linux系统上的用户信息：

```
# finger rich
Login: rich                                Name: Rich Blum
Directory: /home/rich                         Shell: /bin/bash
On since Thu Sep 20 18:03 (EDT) on pts/0 from 192.168.1.2
No mail.
No Plan.
#
```

说明 出于安全性的考虑，很多Linux系统管理员会在系统上禁用finger命令。

如在使用chfn命令时不加参数，它会向你询问要存进备注字段的恰当值：

```
# chfn test
Changing finger information for test.
Name []: Ima Test
Office []: Director of Technology
Office Phone []: (123)555-1234
Home Phone []: (123)555-9876

Finger information changed.
# finger test
Login: test                                  Name: Ima Test
Directory: /home/test                         Shell: /bin/csh
Office: Director of Technology
Office Phone: (123)555-1234
Home Phone: (123)555-9876
Never logged in.
No mail.
No Plan.
#
```

查看/etc/passwd文件中的记录，你会看到下面这样的结果：

```
# grep test /etc/passwd
test:x:504:504:Ima Test,Director of Technology.(123)555-
1234,(123)555-9876:/home/test:/bin/csh
#
```

所有的人物信息现在都存在了/etc/passwd文件中了。

最后，chage命令用来帮助管理用户账户的有效期。它有一些参数可以用来设置每个值，如表6-4所示。

表6-4 chage命令参数

参数	描述
-d	设置上次修改密码到现在的天数
-E	设置密码过期的日期
-I	设置密码过期到锁定账户的天数
-m	设置修改密码之间最少要多少天
-W	设置密码过期前多久开始出现提醒信息

chage命令的日期值可以用下面两种方式中的任意一种：

- YYYY-MM-DD格式的日期；
- 代表从1970年1月1日起到该日期天数的数值。

chage命令中有个好用的功能是设置账户的过期日期。通过它，你就能创建临时用户了。设定的日期一过，临时账户就会自动过期，而不需要记住在那天去删除这些账户。过期的账户跟锁定的账户很相似：账户仍然存在，但用户无法用它登录。

6.2 使用Linux组

用户账户在控制单个用户安全性方面很惯用，但他们在允许一组用户共享资源时就捉襟见肘了。为了达到这个目的，Linux系统用了另外一个概念——组（group）。

组权限允许多个用户共享一组共用的权限来访问系统上的对象，比如文件、目录或设备之类的（会在6.3节中细述）。

Linux发行版在处理默认组的归属关系时略有差异。有些Linux发行版会创建一个组，把所有用户都当做这个组的成员；遇到这种情况要特别小心，因为文件很有可能对其他用户也是可读的。有些发行版会为每个用户创建一个单独的组，这样可以更安全一些。^①

每个组都有一个唯一的GID——跟UID类似，在系统上这是个唯一的数值。和GID一起的，每个组还有一个唯一的组名。Linux系统上有一些组工具可以用来创建和管理你自己的组。本节

^① 例如，Ubuntu就会为每个用户创建一个单独的与用户账户同名的组。在添加用户前后可用grep命令或tail命令查看/etc/group文件的内容比较（grep USERNAME /etc/group或tail /etc/group）。——译者注

将细述组信息是如何保存的，以及如何用组工具来创建新组、修改已有的组。

6.2.1 /etc/group文件

类似于用户账户，组信息也保存在系统的一个文件中。/etc/group文件包含系统上用到的每个组的信息。下面是一些来自Linux系统上/etc/group文件中的典型例子：

```
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon
rich:x:500:
mama:x:501:
katie:x:502:
jessica:x:503:
mysql:x:27:
test:x:504:
```

类似于UID，GID也是用特有的格式分配的。系统账户用的组通常会分配低于500的GID值，而用户组的GID则会从500开始分配。/etc/group文件有4个字段：

- 组名；
- 组密码；
- GID；
- 属于该组的用户列表。

组密码允许非组内成员通过它临时性地成为该组成员。这个功能并不是非常通用，但确实存在。

千万不能直接修改/etc/group文件来添加用户到一个组，而要用usermod命令（在6.1节中介绍过）来添加。在添加用户到不同的组之前，首先得创建组。

说明 用户账户列表某种意义上有些误导人。你会发现在列表中，有些组并没有列出用户。这并不是说，这些组没有成员。当一个用户在/etc/passwd文件中指定某个组作为默认组时，用户账户不会作为该组成员再出现在/etc/group文件中。多年来被这个问题难倒过的系统管理员可不是一个两个。

6.2.2 创建新组

groupadd命令用来在系统上创建新组：

```
# /usr/sbin/groupadd shared
# tail /etc/group
haldaemon:x:68:
xfs:x:43:
```

```
gdm:x:42:
rich:x:500:
mama:x:501:
katie:x:502:
jessica:x:503:
mysql:x:27:
test:x:504:
shared:x:505:
#
```

在创建新组时，默认没有用户属于该组成员。groupadd命令没有提供将用户添加到组的选项，但可以用usermod命令来添加用户到该组：

```
# /usr/sbin/usermod -G shared rich
# /usr/sbin/usermod -G shared test
# tail /etc/group
haldaemon:x:68:
xfs:x:43:
gdm:x:42:
rich:x:500:
mama:x:501:
katie:x:502:
jessica:x:503:
mysql:x:27:
test:x:504:
shared:x:505:rich, test
#
```

shared组现在有两个成员，test和rich。usermod命令的-G参数会把这个新组添加到该用户账户的组列表里。

说明 如果更改了已登录系统账户所属的用户组，该用户必须登出系统后再登录，组关系的更改才能生效。

警告 在将组添加到用户账户时要格外小心。如果加了-g参数，指定的组名会替换掉该账户的默认组。-G参数则将该组添加到用户的属组的列表里，而不会影响默认组。

6.2.3 修改组

我们在/etc/group文件中看到，组信息并不多，也没什么可以修改的。groupmod命令可以修改已有组的GID（加-g参数）或组名（加-n参数）：

```
# /usr/sbin/groupmod -n sharing shared
# tail /etc/group
haldaemon:x:68:
xfs:x:43:
```

```
gdm:x:42:
rich:x:500:
mama:x:501:
katie:x:502:
jessica:x:503:
mysql:x:27:
test:x:504:
sharing:x:505:test.rich
#
```

修改组名时，GID和组成员不会变，只有组名会改变。由于所有的安全权限都是基于GID的，你可以随意改变组名而不会影响文件的安全性。

6.3 理解文件权限

现在你已经了解了用户和组，可以进一步了解用ls命令时输出的神秘文件权限了。本节将会介绍如何对权限码进行解码以及它们的来历。

6.3.1 使用文件权限符

6

如果你还记得第3章，那应该知道ls命令可以用来查看Linux系统上的文件、目录和设备的权限：

```
$ ls -l
total 68
-rw-rw-r-- 1 rich rich 50 2010-09-13 07:49 file1.gz
-rw-rw-r-- 1 rich rich 23 2010-09-13 07:50 file2
-rw-rw-r-- 1 rich rich 48 2010-09-13 07:56 file3
-rw-rw-r-- 1 rich rich 34 2010-09-13 08:59 file4
-rwxrwxr-x 1 rich rich 4882 2010-09-18 13:58 myprog
-rw-rw-r-- 1 rich rich 237 2010-09-18 13:58 myprog.c
drwxrwxr-x 2 rich rich 4096 2010-09-03 15:12 test1
drwxrwxr-x 2 rich rich 4096 2010-09-03 15:12 test2
$
```

输出结果的第一个字段是描述文件和目录权限的码。这个字段的第一个字符代表了对象的类型：

- 代表文件；
- d代表目录；
- l代表链接；
- c代表字符型设备；
- b代表块设备；
- n代表网络设备。

之后有3组三字符的码。每一组三字符码表示三重访问权限：

- r代表对象是可读的；
- w代表对象是可写的；
- x代表对象是可执行的。

如果没有某种权限，在该权限位会出现单破折线。这3组三字码分别对应对象的3个安全级别：

对象的属主；

对象的属组；

系统其他用户。

这个概念可以拆分到图6-1中。

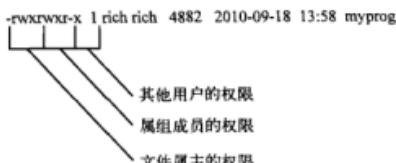


图6-1 Linux文件权限

讨论这个问题的最简单的办法是，找个例子然后逐个分析文件权限：

```
-rwxrwxr-x 1 rich rich 4882 2010-09-18 13:58 myprog
```

文件myprog有下面3组权限。

rwx：文件的属主（设为登录名rich）。

rwx：文件的属组（设为组名rich）。

r-x：系统上其他人。

这些权限说明登录名为rich的用户可以读取、写入以及执行这个文件（可以看做有全部权限）。类似地，属组rich的成员也可以读取、写入和执行这个文件。然而不属于rich组的其他用户只能读取和执行这个文件：w被单破折线取代了，说明这个安全级别没有写入权限。

6.3.2 默认文件权限

你可能会问这些文件权限从何而来，答案是umask。umask命令用来设置用户创建文件和目录的默认权限：

```
$ touch newfile
$ ls -al newfile
-rw-r--r-- 1 rich rich 0 Sep 20 19:16 newfile
$
```

touch命令用分配给我的用户账户的默认权限创建了这个文件。umask命令可以显示和设置这个默认权限：

```
$ umask
0022
$
```

遗憾的是，umask命令的设置不是那么简单明了；想弄明白它如何工作，更是让人一头雾水。第一位代表了一项特别的安全特性，叫作粘着位（sticky bit）。关于这部分内容，我们会在6.5节再讲。

后面的3位表示文件或目录的umask的八进制值。要理解umask是怎么工作的，先得理解八进制模式的安全性设置。

八进制模式的安全性设置先获取这3组rwx权限的值，然后将其转换成3位二进制值来表示一个八进制值。在这个二进制表示中，每个位置代表一个二进制位。因此，如果读权限是唯一置位的权限，权限值就是r--，转换成二进制值就是100，代表的八进制值是4。表6-5列出了可能会遇到的组合。

表6-5 Linux文件权限码

权 限	二进制值	八进制值	描 述
---	000	0	没有任何权限
--X	001	1	只有执行权限
-W-	010	2	只有写入权限
-WX	011	3	有写入和执行权限
r--	100	4	只有读取权限
r-X	101	5	有读取和执行权限
rW-	110	6	有读取和写入权限
rWX	111	7	有全部权限

八进制模式先取得权限的八进制值，然后再把这3组安全级别（属主、属组和其他用户）的八进制值顺序列出。因此，八进制模式的值664代表属主和属组成员都有读取和写入的权限，而其他用户都只有读取权限。

现在你了解了八进制模式权限是怎么工作的，umask值反而更叫人困惑了。我的Linux系统上默认的八进制的umask值是0022，而我所创建的文件的八进制权限却是644，这是如何得来的呢？

umask值只是个掩码。它会屏蔽掉不想授予该安全级别的权限。接下来我们稍微进行一点八进制运算来把这个原因讲完。

umask值会从对象的全权限值中减掉，对文件来说，全权限的值是666（所有用户都有读和写的权限）；而对目录来说，是777（所有用户都有读、写、执行权限）。所以，在上面的例子中，文件一开始的权限是666，然后umask值022作用后，剩下的文件权限就成了644。

umask值通常会在/etc/profile启动文件中设置（参见第5章）。你可以用umask命令为默认umask设置指定一个新值：

```
$ umask 026
$ touch newfile2
$ ls -l newfile2
-rw-r----- 1 rich    rich          0 Sep 20 19:46 newfile2
$
```

在把umask值设成026后，默认的文件权限变成了640，因此新文件现在对属组成员来说是只读的，而系统里的其他成员则没有任何权限。

umask值同样会作用在创建目录上：

```
$ mkdir newdir
$ ls -l
drwxr-x--x  2 rich    rich        4096 Sep 20 20:11 newdir/
$
```

由于目录的默认权限是777，umask作用后生成的目录权限不同于生成的文件权限。umask值026会从777中减去，留下来751作为目录权限设置。

6.4 改变安全性设置

如果你已经创建了一个目录或文件而需要改变它的安全性设置，在Linux系统上有一些不同的工具能完成这个功能。本节将告诉你如何更改文件和目录的已有权限、默认文件属主以及默认属组。

6.4.1 改变权限

chmod命令用来改变文件和目录的安全性设置。chmod命令的格式如下：

```
chmod options mode file
```

mode参数后可跟八进制模式或符号模式来设置安全性设置。八进制模式设置非常直接，直接用期望赋予文件的标准的3位八进制权限码：

```
$ chmod 760 newfile
$ ls -l newfile
-rwxrw----  1 rich    rich        0 Sep 20 19:16 newfile
$
```

八进制文件权限会自动应用到指定的文件上。符号模式的权限就沒这么简单了。

与通常用到的3组三字符权限字符不同，chmod命令用了另外一种实现。下面是在符号模式下指定权限的格式：

```
[ugoa...][[+-=][rwxXstugo...]]
```

非常有意义，不是吗？第一组字符定义了权限作用的对象：

- u代表用户；
- g代表组；
- o代表其他；
- a代表上述所有。

下一步，后面跟着的符号表示你是想在现有权限基础上增加权限（+），还是在现有权限基础上移除权限（-），还是将权限设置成后面的值（=）。

最后，第3个符号代表作用到设置上的权限。你会发现，这个值要比rwx多。额外的设置有以下几项。

- X: 如果对象是目录或者它已有执行权限，赋予执行权限。
- s: 运行时重新设置UID或GID。
- t: 保留文件或目录。
- u: 将权限设置为跟属主一样。
- g: 将权限设置为跟属组一样。
- o: 将权限设置为跟其他用户一样。

这么使用这些权限：

```
$ chmod o+r newfile
$ ls -l newfile
-rwxrwxr-- 1 rich    rich          0 Sep 20 19:16 newfile
$
```

不管其他用户在这一安全级别之前都有什么权限，o+r给这一级别添加了读取权限。

```
$ chmod u-x newfile
$ ls -l newfile
-rw-rw-r-- 1 rich    rich          0 Sep 20 19:16 newfile
$
```

6

u-x移除了属主已有的执行权限。回顾一下之前讲过的ls命令的那个参数-F，能够在具有执行权限的文件名后加一个星号。可用在这里来检查前后的变化。

options参数为chmod命令提供了另外一些功能。-R参数可以让权限的改变递归地作用到文件和子目录。可以在指定文件名时用通配符将权限的更改通过一个命令作用到多个文件上。

6.4.2 改变所属关系

有时你需要改变文件的属主，比如有人离职或开发人员创建了一个在产品环境中运行时需要归属在系统账户下的应用。Linux提供了两个命令来完成这个功能：chown命令用来改变文件的属主，chgrp命令用来改变文件的默认属组。

chown命令的格式如下：

```
chown options owner[,group] file
```

可用登录名或UID来指定文件的新属主：

```
# chown dan newfile
# ls -l newfile
-rw-rw-r-- 1 dan    rich          0 Sep 20 19:16 newfile
#
```

非常简单。chown命令也支持同时改变文件的属主和属组：

```
# chown dan:shared newfile
# ls -l newfile
-rw-rw-r-- 1 dan    shared        0 Sep 20 19:16 newfile
#
```

如果你不嫌麻烦，那么你可以这么改变一个目录的默认属组：

```
# chown .rich newfile
# ls -l newfile
-rw-rw-r-- 1 dan      rich          0 Sep 20 19:16 newfile
#
```

最后，如果你的Linux系统采用和用户登录名匹配的组名，你可以只用一个条目就改变二者：

```
# chown test. newfile
# ls -l newfile
-rw-rw-r-- 1 test      test         0 Sep 20 19:16 newfile
#
```

chown命令采用一些不同的选项参数。-R参数加通配符可以递归地改变子目录和文件的所属关系。-h参数可以改变该文件的所有符号链接文件的所属关系。

说明 只有root用户能够改变文件的属主。任何属主都可以改变文件的属组，但前提是属主必须是源和目标属组的成员。

chgrp命令可以很方便地更改文件或目录的默认属组：

```
$ chgrp shared newfile
$ ls -l newfile
-rw-rw-r-- 1 rich      shared        0 Sep 20 19:16 newfile
$
```

现在shared组的任意一个成员都可以写这个文件了。这是Linux系统共享文件的一个途径。然而，在系统上给一组人共享一个文件很复杂。下一节会介绍如何做。

6.5 共享文件

可能你已经猜到了，创建组是Linux系统上共享文件访问权限的方法。但在一个完整的共享文件的环境中，事情会复杂得多。

在6.3节中你已经看到，创建新文件时，Linux会用默认UID和GID来给文件分配权限。想让其他人也能访问文件，你要么改变其他用户所在安全组的访问权限，要么就给文件分配一个新的包含其他用户的默认属组。

如果你想在大的环境中创建文档并将文档与人共享，这会很烦琐。幸好有解决这个问题的简单方法。

Linux还为每个文件和目录存储了3个额外的信息位。

□ **设置用户ID（SUID）**：当文件被用户使用时，程序会以文件属主的权限运行。

□ **设置组ID（SGID）**：对文件来说，程序会以文件属组的权限运行；对目录来说，目录中创建的新文件会以目录的默认属组作为默认属组。

□ **粘着位**：进程结束后文件还会在内存中。

SGID位对文件共享非常重要。使能了SGID位，你能让在一个共享目录下创建的新文件都属

于该目录的属组，也就是每个用户的组。

SGID可通过chmod命令设置。它会加到标准3位八进制值之前（组成4位八进制值），或者在符号模式下用符号s。

如果你用的是八进制模式，你需要知道这些位的位置，如表6-6所示。

表6-6 chmod SUID、SGID和粘着位的八进制值

二进制值	八进制值	描述
000	0	所有位都清零
001	1	粘着位置位
010	2	SGID位置位
011	3	SGID位和粘着位都置位
100	4	SUID位置位
101	5	SUID位和粘着位都置位
110	6	SUID位和SGID位都置位
111	7	所有位都置位

因此，要创建一个共享目录，使目录里的新文件都能沿用目录的属组，你只需将该目录的SGID位置位：

```
$ mkdir testdir
$ ls -l
drwxrwxr-x  2 rich    rich        4096 Sep 20 23:12 testdir/
$ chgrp shared testdir
$ chmod g+s testdir
$ ls -l
drwxrwsr-x  2 rich    shared      4096 Sep 20 23:12 testdir/
$ umask 002
$ cd testdir
$ touch testfile
$ ls -l
total 0
-rw-rw-r--  1 rich    shared      0 Sep 20 23:13 testfile
$
```

首先，用mkdir命令来创建希望共享的目录。其次，通过chgrp命令将目录的默认属组改为含有所有需要共享文件用户的组。最后，将目录的SGID位置位，以保证目录中新建文件都用shared作为默认属组。

为了让这个环境能正常工作，所有组成员都需要把他们的umask值设置成文件对属组成员可写。在前面的例子中，umask改成了002，所以文件是对属组是可写的。

做完了这些，组成员就能到共享目录下创建新文件了。跟期望的一样，新文件会沿用目录的属组，而不是用户的默认属组。现在shared组的所有用户都能访问这个文件了。

6.6 小结

本章讨论了管理Linux系统安全性需要知道的一些命令行命令。Linux通过用户ID和组ID来限

制对文件、目录以及设备的访问。Linux将用户账户的信息存储在/etc/passwd文件中，将组信息存储在/etc/group文件中。每个用户都会被分配一个唯一的用户ID，以及在系统中识别用户的文本登录名。组也会被分配一个唯一的组ID以及组名。组可以包含一个或多个用户以支持对系统资源的共享访问。

有若干命令可以用来管理用户账户和组。useradd命令用来创建新的用户账户，groupadd命令用来创建新的组账户。修改已有用户账户，我们用usermod命令。类似的groupmod命令用来修改组账户信息。

Linux采用复杂的位系统来判定文件和目录的访问权限。每个文件都有3个安全等级：文件的属主、有访问文件权限的默认属组，以及系统上的其他用户。每个安全等级通过3个访问权限位来定义：读取、写入以及执行。如果权限被否定了，权限对应的符号会用单破折线代替（比如r--代表只读权限）。

符号权限通常以八进制值被引用。每个八进制值都是由3个二进制位组合而成，而3个八进制值代表着3个安全等级。umask命令用来设置系统上创建的文件和目录的默认安全设置。系统管理员通常会在/etc/profile文件中设置一个默认的umask值，但你可以随时通过umask命令来修改自己的umask值。

chmod命令用来修改文件和目录的安全设置。只有文件的属主才能改变文件或目录的权限。但root用户也可以改变系统上任意文件或目录的安全设置。chown和chgrp命令可以用来改变文件默认的属主和属组。

最后，本章以如何使用设置组ID位来创建共享目录结尾。SGID位会强制某个目录下创建的新文件或目录都沿用该父目录的属组，而不是创建这些文件的用户的属组。这可以为系统的用户之间共享文件提供一个简便的途径。

现在你已经了解了文件权限，下面就可以进一步了解如何在Linux上处理真实的文件系统。下一章将会介绍如何从命令行在Linux上创建新的分区，以及如何格式化新分区以使其在Linux虚拟目录中可用。



本章内容

- 什么是文件系统
- Linux文件系统
- 文件系统命令

使用Linux系统时，你需要作的决策之一是存储设备用什么文件系统。安装时大多数Linux发行版会为系统提供一个默认的文件系统，大多数入门级用户想都不想就用默认的那个了。

使用默认文件系统也不算糟糕，但了解一下可用的选择有时也会有所帮助。本章将探讨Linux世界里可选用的不同文件系统，并向你演示如何在命令行上创建和管理它们。

7

7.1 探索Linux文件系统

第3章讨论了Linux如何通过文件系统来在存储设备上存储文件和目录。文件系统为Linux提供了从硬盘中存写的0和1到你在应用中用到的文件和目录之间的桥梁。

Linux支持多种类型的文件系统来管理文件和目录。每种文件系统都在存储设备上实现了虚拟目录结构，只是特性略有不同。本章将带你逐步了解Linux环境中较常用的文件系统的优点和缺陷。

7.1.1 基本的Linux文件系统

最早，Linux系统用的是模仿Unix文件系统功能的一个简单文件系统。本节将讨论那个文件系统的演进过程。

1. ext文件系统

Linux操作系统中引入的最早的文件系统叫做扩展文件系统(extended filesystem, 简记为ext)。它为Linux提供了一个基本的类Unix文件系统：使用虚拟目录来操作硬件设备，在物理设备上按定长的块来存储数据。

ext文件系统采用称作索引节点的系统来存放虚拟目录中所存储文件的信息。索引节点系统在每个物理设备中创建一个单独的表（称为索引节点表）来存储这些文件的信息。存储在虚拟目

录中的每一个文件在索引节点表中都有一个条目。条目名称的扩展部分来自其跟踪每个文件的额外数据，包括：

- 文件名；
- 文件大小；
- 文件的属主；
- 文件的属组；
- 文件的访问权限；
- 指向存有文件数据的每个硬盘块的指针。

Linux通过唯一的数值（称作索引节点号）来引用索引节点表中的每个索引节点，这个值是创建文件时由文件系统分配的。文件系统通过索引节点号而不是文件全名及路径来标识文件。

2. ext2文件系统

最早的ext文件系统有不少限制，比如文件大小不得超过2 GB。在Linux出现后不久，ext文件系统就升级到了第二扩展文件系统，称作ext2。

如你所猜的，ext2文件系统是ext文件系统基本功能的一个扩展，但维护着同样的结构。ext2文件系统扩展了索引节点表的格式来保存系统上每个文件的更多信息。

ext2的索引节点表为文件添加了创建时间值、修改时间值和最后访问时间值来帮助系统管理员追踪文件的访问情况。ext2文件系统还将允许的最大文件大小增加到了2 TB（在ext2的后期版本中，增加到了32 TB），以容纳数据库服务器中常见的大文件。

除了扩展索引节点表外，ext2文件系统还改变了文件在数据块中存储的方式。ext文件系统常见的问题是在文件写入到物理设备时，存储数据用的块很容易就分散在整个设备上（称作碎片化，fragmentation）。数据块的碎片化会降低文件系统的性能，因为需要更长的时间查找存储设备来访问特定文件的所有块。

保存文件时，ext2文件系统通过按组分配磁盘块来减轻碎片化。通过将数据块分组，文件系统不需要为了数据块查找整个物理设备来读取文件。

多年里，ext文件系统都是Linux发行版采用的默认文件系统。但它也有一些限制。索引节点表虽然支持文件系统保存有关文件的更多信息，但会造成对系统来说致命的问题。文件系统每次存储或更新文件，它都要用新信息来更新索引节点表。但它并不是连成一气的。

如果在存储文件和更新索引节点表的过程中，计算机系统发生了什么事情，这二者就不同步了。ext2文件系统由于容易在系统崩溃或断电时损坏而臭名昭著。这样即使文件数据正常地保存到了物理设备上，如果索引节点表记录没完成更新，ext2文件系统甚至都不知道那个文件存在。

很快开发人员就开始尝试开发不同的Linux文件系统了。

7.1.2 日志文件系统

日志文件系统给Linux系统增加了一层安全性。取代了之前先将数据直接写入存储设备再更新索引节点表的做法，日志文件系统会先将文件的更改写入到临时文件（称作日志，journal）中，然后在数据成功写到存储设备和索引节点表之后，再删除对应的日志条目。

如果系统在数据被写入到存储设备之前崩溃了或断电了，日志文件系统下次会读取日志文件并处理上次留下的未写入的数据。

Linux中有3种不同的广泛使用的日志方法，每个都有不同等级的保护，如表7-1所示。

表7-1 文件系统日志方法

方 法	描 述
数据模式	索引节点和文件都会被写入日志；丢失数据风险低，但性能差
排序模式	只有索引节点数据会被写入日志，但只有数据成功写入后才删除；性能和安全之间的良好折中
回写模式	只有索引节点数据会被写入日志，但不管文件数据何时写入；丢失数据风险高，但仍比不用日志好

数据模式日志方法是目前为止最安全的保护数据的方法，但它同时也是最慢的。所有写到存储设备上的数据都必须写两次：第一次写到日志，第二次写到真正的存储设备上。这样会导致性能很差，尤其是对要做大量数据写入的系统。

多年来，在Linux上还出现了一些其他日志文件系统。后面几节将会详述常见的Linux日志文件系统。

7.1.3 扩展的Linux日志文件系统

开发ext和ext2文件系统的同一组人，作为Linux项目的一部分，也开发了这两个文件系统的支持日志版本。这些日志文件系统同ext2文件系统兼容，并且很容易在它们之间转换。现在有基于ext2文件系统的两个独立的日志文件系统。

1. ext3文件系统

2001年，ext3文件系统加到了Linux内核中，直到最近都是几乎所有Linux发行版默认的文件系统。它采用和ext2文件系统相同的索引节点表结构，但给每个存储设备增加了一个日志文件，来将准备写入存储设备的数据先写进日志文件。

默认情况下，ext3文件系统用排序模式的日志功能——只将索引节点信息写入日志文件，直到数据块都被成功写入存储设备才删除。你可以在创建文件系统时用简单的一个命令行选项将ext3文件系统的日志方法改成数据模式或回写模式。

虽然ext3文件系统为Linux文件系统添加了基本的日志功能，但它仍然缺一些东西。例如ext3文件系统无法恢复误删的文件，它没有任何内建的数据压缩功能（虽然有个需单独安装的补丁支持这个功能），ext3文件系统也不支持加密文件。鉴于这些原因，Linux项目的开发人员选择继续做些工作来提高ext3文件系统。

2. ext4文件系统

扩展ext3文件系统功能的结果就是ext4文件系统。ext4文件系统在2008年时被Linux内核官方支持，现在已是大多数流行的Linux发行版采用的默认文件系统，比如Fedora和Ubuntu。

除了支持数据压缩和加密，ext4文件系统还支持一个称作区段(extent)的特性。区段在存储设备上按块分配空间，但在索引节点表中只保存起始块的位置。由于无需列出所有用来存储文件中数据的数据块，它可以在索引节点表中节省一些空间。

ext4还整合了块预分配 (block preallocation)。如果你想在存储设备上给一个你知道要变大的文件预留空间，通过ext4文件系统你可以为文件分配所有期望的块，不只是物理上存在的块。ext4文件系统用0填满预留的数据块，并知道不要将它们分配给其他文件。

3. Reiser文件系统

2001年，Hans Reiser为Linux创建了第一个日志文件，称为ReiserFS。ReiserFS文件系统只支持回写日志模式——只把索引节点表数据写到日志文件。ReiserFS文件系统是Linux上最快的日志文件系统之一。

整合进ReiserFS文件系统的两个有意思的特性是：你可以在线调整已有文件系统的大小；它还采用了一项称作尾部压缩 (tail packing) 的技术，该技术能将一个文件的数据填进另一个文件的数据块中的空白空间。如果你必须为已有文件系统扩容来容纳更多的数据，在线调整文件系统大小功能非常好用。

注意 由于Hans Reiser的法律问题，ReiserFS文件系统的状态还是个未知数。虽然ReiserFS是个开源项目，Hans是这个项目的首席开发人员，但他现在仍在监狱里。ReiserFS文件系统最近都没开发活动。也不太清楚会不会有其他人接手这个项目的开发工作。考虑到现在已有其他可用的替代日志文件系统，大多数新的Linux发行版不会用ReiserFS文件系统。但你仍可能会遇到还在使用它的现有Linux系统。

4. JFS文件系统

作为可能依然在用的最老的日志文件系统之一，JFS (Journalized File System, 日志文件系统^①) 是IBM在1990年为它的Unix衍生版——AIX开发的。然而，直到第2版它才被移植到Linux环境中。

说明 IBM官方称JFS文件系统的第2版为JFS2，但大多数Linux系统提到它时都用JFS。

JFS文件系统采用顺序日志方法，即只在日志中保存索引节点表数据，直到真正的文件数据被写进存储设备时才删除它。这个方法是ReiserFS的速度和数据模式日志方法的完整性之间的一个折中。

JFS文件系统采用基于区段的文件分配，即为每个写入存储设备的分配一组块。这个方法可以减少存储设备上的碎片。

除了IBM Linux版本外，JFS文件系统很少使用，但你有可能在Linux旅程中碰到。

5. XFS文件系统

XFS日志文件系统是另一个文件系统，最初是为一个商业Unix系统开发的，现在正在逐渐走进Linux世界。硅图公司 (SGI) 最早在1994年为它的IRIX Unix系统开发了XFS。在2002年，为了

^① 此处“日志文件系统”是指Journalized File System这一Journal File System概念的具体实现。为防止读者混淆，后文中都将用JFS缩写代替。——译者注

共用，它被发布到了Linux环境。

XFS文件系统采用回写模式的日志，它提供了很好的性能但也引入了相当大的风险，因为真实数据依然未存进日志文件。XFS文件系统还允许在线调整文件系统的大小，类似于ReiserFS文件系统，除了XFS文件系统只可扩大不能缩小。

7.2 操作文件系统

Linux提供了一些不同的工具，可以在命令行下操作文件系统更为方便。你可以通过自己的键盘舒服地创建新的文件系统或者修改已有的文件系统。本节将会带你逐步了解在命令行下同文件系统交互的命令。

7.2.1 创建分区

一开始，你必须在存储设备上创建分区来容纳文件系统。分区可以是整个硬盘，也可以是硬盘的一部分，来容纳虚拟目录的一部分。

fdisk工具用来帮助管理安装在系统上的任何存储设备上的分区。fdisk命令是个交互式程序，允许你输入命令来逐步走完给硬盘分区的步骤。

要启动fdisk命令，你必须指定要分区的存储设备的设备名：

```
$ sudo fdisk /dev/sdc
[sudo] password for rich:
Device contains neither a valid DOS partition table, nor Sun, SGI or
OSF disklabel
Building a new DOS disklabel with disk identifier 0x4beedc66.
Changes will remain in memory only, until you decide to write them.
After that, of course, the previous content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected
by w(write)

WARNING: DOS-compatible mode is deprecated. It's strongly recommended
To switch off the mode (command 'c') and change display
units to sectors (command 'u').
```

Command (m for help):

如果这是你第一次给该存储设备分区，fdisk会警告你设备上没有分区表。

fdisk交互式命令提示符使用单字母命令来告诉fdisk做什么。表7-2显示了fdisk命令提示符下的可用命令。

表7-2 fdisk命令

命 令	描 述
a	设置一个标识，说明这个分区是可启动的
b	编辑BSD Unix系统用的磁盘标签

(续)

命 令	描 述
c	设置DOS兼容标识
d	删除分区
l	显示可用的分区类型
m	显示命令选项
n	添加一个新分区
o	创建DOS分区表
p	显示当前分区表
q	退出, 不保存更改
s	为Sun Unix系统创建一个新磁盘标签
t	修改分区的系统ID
u	改变使用的存储单位
v	验证分区表
w	将分区表写入磁盘
x	高级功能

尽管看上去很恐怖, 但实际上你在日常工作中用到的只有几个基本命令。

对于初学者, 你可以用p命令将一个存储设备的详细信息显示出来:

```
Command (m for help): p
Disk /dev/sdc: 5368 MB, 5368946688 bytes
255 heads, 63 sectors/track, 652 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x4beedc66

Device Boot      Start         End      Blocks   Id  System

```

Command (m for help):

输出说明这个存储设备有5368 MB (5 GB) 的空间。存储设备明细后的列表说明这个设备上是否已有分区。这个例子中的输出中没有显示任何分区, 所以设备还未分区。

下一步, 你要在该存储设备上创建新的分区。用n命令来创建:

```
Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-652, default 1): 1
Last cylinder, +cylinders or +size{K,M,G} (1-652, default 652): +2G

Command (m for help):
```

分区可以按主分区（primary partition）或扩展分区（extended partition）创建。主分区可以被文件系统直接格式化，而扩展分区只能容纳其他主分区。扩展分区出现的原因是每个存储设备上只能有4个分区。你可以通过创建多个扩展分区然后在扩展分区中创建主分区来进行扩展。上例中创建了一个主分区，在存储设备上给它分配了分区号1，然后给它分配了2 GB的存储设备空间。你可以再用p命令来查看结果：

```
Command (m for help): p
Disk /dev/sdc: 5368 MB, 5368946688 bytes
255 heads, 63 sectors/track, 652 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x4beedc66

Device Boot      Start        End      Blocks   Id  System
/dev/sdc1            1       262     2104483+  83  Linux
```

Command (m for help):

现在输出中该存储设备上有了一个分区（叫做/dev/sdc1）。Id那列定义了Linux怎么对待该分区。fdisk允许创建多种分区类型。使用l命令列出可用的不同类型。默认类型是83，定义了一个Linux文件系统。如果你想为其他文件系统创建一个分区（比如Windows的NTFS分区），只要选择一个不同的分区类型。

你可以重复上面的过程，将存储设备上剩下的空间分配给另一个Linux分区，在你创建了想要的分区之后，用w命令来将更改保存到存储设备上：

```
Command (m for help): w
The partition table has been altered!
Calling ioctl() to re-read partition table.
Syncing disks.
```

有了存储设备的一个分区后，你可以在Linux系统上对其进行格式化了。

说明 有时创建新硬盘分区的最难之处在于找到Linux系统上的物理磁盘。Linux在给硬盘分配设备名时采用标准格式，但你需要熟悉这种格式。对于较早的IDE硬盘，Linux用/dev/hdx，其中x代表基于硬盘发现顺序的字母（a代表第一块硬盘，b代表第二块，以此类推）。对于较新的SATA硬盘和SCSI硬盘，Linux采用/dev/sdx，其中x代表硬盘发现的顺序（同样，a代表第一块硬盘，b代表第二块，以此类推）。在格式化分区以前，再三检查，保证你用的是正确的盘，你会一直受益。

7.2.2 创建文件系统

在将数据存储到这个分区之前，你必须用某种文件系统格式化它，这样Linux才能用它。每

种文件系统类型都用自己的命令行程序来格式化分区。表7-3列出了本章中讨论的不同文件系统所对应的工具。

表7-3 创建文件系统的命令行程序

工 具	用 途
mkfs	创建一个ext文件系统
mke2fs	创建一个ext2文件系统
mkfs.ext3	创建一个ext3文件系统
mkfs.ext4	创建一个ext4文件系统
mkreiserfs	创建一个ReiserFS文件系统
jfs_mkfs	创建一个JFS文件系统
mkfs.xfs	创建一个XFS文件系统

每个文件系统命令都有很多命令行选项，允许你定制创建在那个分区上的文件系统。要查看所有可用的命令行选项，可用man命令来显示该文件系统命令的手册页面（参见第2章）。所有的文件系统命令都允许通过不带选项的简单命令来创建默认文件系统：

```
$ sudo mkfs.ext4 /dev/sdc1
[sudo] password for rich:
mke2fs 1.41.11 (14-Mar-2010)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
131648 inodes, 526120 blocks
26306 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=541065216
17 block groups
32768 blocks per group, 32768 fragments per group
7744 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
$
```

这个新的文件系统采用ext4文件系统类型，是Linux上的日志文件系统。注意，创建过程中有一部分是创建新的日志。

为分区创建了文件系统之后，下一步是将它挂载到虚拟目录下的某个挂载点，这样就可以将数据存储在新文件系统中了。你可以将新文件系统挂载到虚拟目录中需要额外空间的任何位置。

```
$ sudo mkdir /mnt/testing
$ sudo mount -t ext4 /dev/sdb1 /mnt/testing
$ ls -al /mnt/testing
total 24
drwxr-xr-x 3 root root 4096 2010-09-25 19:25 .
drwxr-xr-x 3 root root 4096 2010-09-25 19:38 ..
drwx----- 2 root root 16384 2010-09-25 19:25 lost+found
$
```

`mkdir`命令会在虚拟目录中创建挂载点，`mount`命令会将新的硬盘分区添加到挂载点。现在你可以在新分区中保存新文件和目录了。

注意 这种挂载文件系统的方法只会临时挂载该文件系统。当重启Linux系统时，文件系统不会自动挂载。要强制Linux在启动时自动挂载这个新文件系统，可以将文件系统添加到`/etc/fstab`文件中。

本章演示了如何处理物理存储设备中的文件系统。Linux还提供了一些不同方法来为文件系统创建逻辑存储设备。下一节将会演示如何使用逻辑存储设备。

7.2.3 如果出错了

即使是最好的日志文件系统，突然断电或正在访问文件时有程序锁定了系统，可能也会出错。幸而有一些可用的命令行工具可以帮你尝试将文件系统恢复正常状态。

每个文件系统都有自己的和文件系统交互的恢复命令。这可能会让系统变得很难用，因为在Linux环境中有很多的文件系统可用也会造成越来越多的要知道的命令。好在有个共用的前端，它可以决定存储设备上的文件系统并根据要恢复的文件系统调用对应的文件系统恢复命令。

`fsck`命令用来检查和修复任意类型的Linux文件系统，包括本章早些时候讨论过的所有文件系统——ext、ext2、ext3、ext4、ReiserFS、JFS和XFS。该命令的格式是：

```
fsck options filesystem
```

你可以在命令行上列出多个要检查的文件系统条目。文件系统可以通过设备名、在虚拟目录中的挂载点以及分配给文件系统的唯一UUID值来引用。

`fsck`命令使用`/etc/fstab`文件来自动决定挂载到系统上的存储设备的文件系统。如果存储设备通常不挂载（比如你刚刚在新的存储设备上创建了个文件系统），你需要用`-t`命令行选项来指定文件系统类型。表7-4列出了其他可用的命令行选项。

表7-4 fsck的命令行选项

选 项	描 述
-a	如果检测到错误，自动修复文件系统
-A	检查 <code>/etc/fstab</code> 文件中列出的所有文件系统
-C	给支持进度条功能的文件系统显示一个进度条（只有ext2和ext3）

(续)

选 项	描 述
-N	不进行检查，只显示哪些检查会执行
-r	出现错误时提示
-R	使用-A选项时跳过根文件系统
-s	检查多个文件系统时，依次进行检查
-t	指定要检查的文件系统类型
-T	启动时不显示头信息
-V	在检查时产生详细输出
-y	检测到错误时自动修复文件系统

你可能注意到了，有些命令行选项是重复的。这是试图为多个命令实现一个共用的前端带来的部分问题。有些文件系统修复命令有一些额外的可用选项。如果需要做更高级的错误检查，你需要查看这个文件系统修复工具的手册页面来确定是不是有那个文件系统专用的扩展选项。

注意 你只能在未挂载的文件系统上运行fsck命令。对大多数文件系统来说，你只需卸载文件系统来进行检查，检查完成了重新挂载就好了。但因为根文件系统含有所有核心的Linux命令和日志文件，你不能在运行的系统上卸载它。

这正是亲手体验Linux LiveCD的好时机。只需用LiveCD启动系统就可以了，然后在根文件系统上运行fsck命令。

7.3 逻辑卷管理器

如果用标准分区在硬盘上创建了文件系统，往已有文件系统添加额外的空间在某种意义上会是个痛苦的过程。你只能将分区的大小扩展到同一个物理硬盘上的可用空间那么大。如果硬盘上没有额外的可用空间，你就必须弄一个更大的硬盘然后手动将已有的文件系统移动到新的硬盘上。

这时能派上用场的就是，通过将另外一个硬盘上的分区加到已有文件系统，来动态地向已有文件系统添加空间的方法。Linux逻辑卷管理器（Logical Volume Manager, LVM）软件包正好可以用来做这个。它为你提供了在Linux系统上无需重新构建整个文件系统而操作硬盘空间的简便办法。

7.3.1 逻辑卷管理布局

逻辑卷管理的核心是它如何处理安装在系统上的硬盘分区。在逻辑卷管理的世界里，硬盘称作物理卷（Physical Volume, PV）。每个物理卷都会映射到硬盘上创建的某一物理分区。

多个物理卷元素集中在一起可以组成一个卷组 (Volume Group, VG)。逻辑卷管理系统会把卷组当做物理硬盘一样对待，但事实上卷组可能是由分布在多个物理硬盘上的多个物理分区组成的。卷组提供了一个创建逻辑分区的平台，而这些逻辑分区事实上包含了文件系统。

整个结构中的最后一层是逻辑卷 (Logical Volume, LV)。逻辑卷为Linux提供了创建文件系统的分区环境，作用类似于到目前为止我们一直在探讨的Linux中的物理硬盘。Linux系统将逻辑卷当做物理分区对待。你可以使用任意一种标准Linux文件系统来格式化逻辑卷，然后再将它在某个挂载点添加进Linux虚拟目录中。

图7-1显示了典型Linux逻辑卷管理环境的布局图。



图7-1 逻辑卷管理环境

图7-1中的卷组横跨了3个单独的物理硬盘，覆盖了5个独立的物理分区。在卷组内部有两个独立的逻辑卷。Linux系统将每个逻辑卷当做物理分区。每个逻辑卷可以被格式化成ext4文件系统，然后挂载到虚拟目录中某个特定位置。

注意图7-1中第3个物理硬盘有一个未使用的分区。通过逻辑卷管理，你可以在后面轻松地将这个未使用分区分配到已有卷组，然后在你需要更多空间时用它创建一个新的逻辑卷或将它添加到已有逻辑卷。

类似地，如果你给系统添加了一块硬盘，逻辑卷管理系统允许你将它添加到已有卷组，然后为已有卷组创建更多空间，或者启动一个新的用来挂载的新逻辑卷。这是用来扩展文件系统的更为好用的方法。

7.3.2 Linux中的LVM

Linux LVM是由Heinz Mauelshagen开发的，于1998年发布到Linux社区。它允许你在Linux上用简单的命令行命令管理一个完整的逻辑卷管理环境。

Linux LVM有两个可用的版本。

- **LVM1：**最初的LVM包于1998年发布，只在Linux内核2.4版本上可用。它仅提供了基本的逻辑卷管理功能。

□ **LVM2**: LVM的更新版本，在Linux内核2.6中才可用。它在标准的LVM1功能外提供了额外的功能。

大部分采用2.6内核版本的现代Linux发行版都提供对LVM2的支持。除了标准的逻辑卷管理功能外，LVM2为你提供了一些其他的在Linux系统上好用的功能。

1. 快照

最早的Linux LVM允许你将一个已有的逻辑卷在逻辑卷在线的状态下复制到另一个设备。这个功能称作快照（snapshot）。快照功能对备份由于高可靠性需求而无法锁定的重要数据来说非常好。传统的备份方法在将文件复制到备份媒体上时通常要将文件锁定。快照允许你在复制的同时继续运行关键任务的Web服务器或数据库服务器。遗憾的是，LVM1只允许你创建只读快照。一旦创建了快照，你就不能再写入东西了。

LVM2允许你创建在线逻辑卷的可读写快照。有了可读写的快照，你就可以删除原先的逻辑卷然后将快照作为替代挂载上。这个功能对快速故障转移或要修改数据的程序试验（一旦失败，就要重启系统）非常有用。

2. 条带化

LVM2提供的另一个有意思的功能是条带化（striping）。有了条带化，可跨多个物理硬盘创建一个逻辑卷。当Linux LVM将文件写入逻辑卷时，文件中的数据块会被分散到多个硬盘上。每个后继数据块会被写到下一个硬盘上。

条带化有助于提高硬盘的性能，因为Linux可以将一个文件的多个数据块同时写入多个硬盘，而不是必须等待单个硬盘移动读写磁头到多个不同位置。这个改进同样适用于读取顺序访问的文件，因为LVM可同时从多个硬盘读取数据。

注意 LVM条带化不同于RAID条带化。LVM条带化不提供用来创建容错环境的校验信息。事实上，LVM条带化会增加文件因硬盘失误丢失的概率。单个硬盘失误可能会造成多个逻辑卷无法访问。

3. 镜像

通过LVM安装文件系统并不意味着文件系统就不会出问题。和物理分区一样，LVM逻辑卷也容易受到断电和硬盘崩溃的影响。一旦文件系统损坏了，就总有可能无法恢复它。

LVM快照过程提供了一些便利，知道你可以在任何时间创建一个逻辑卷的备份副本，但对有些环境来说可能不够。有很多数据改变的系统，比如数据库服务器，自上次快照可能要存储数百或数千条记录。

这个问题的一个解决办法就是LVM镜像。镜像是一个实时更新的逻辑卷的一份完整副本。当你创建镜像逻辑卷时，LVM会将原始逻辑卷同步到镜像副本中。根据原始逻辑卷的大小，这可能需要一些时间才能完成。

一旦原始同步完成了，LVM会为文件系统的每次写过程进行两次写过程——一个写到主逻辑

卷，一个写到镜像副本。如你能猜到的，这个过程会降低系统的写入性能。然而，如果原始逻辑卷因为某些原因损坏了，你就可以在手头有一个完整的最新副本了。

7.3.3 使用Linux LVM

现在你已经知道了Linux LVM可以做什么了，本节将讨论如何实现它来帮助组织你系统上的硬盘空间。Linux LVM包只提供了创建和管理逻辑卷管理系统中所有组建的命令行程序。有些Linux发行版包含了命令行命令对应的图形化前端，但为了完全控制你的LVM环境，最好习惯直接使用这些命令。

1. 定义物理卷

这个过程的第一步就是将硬盘上的物理分区转换成Linux LVM使用的物理卷区段。我们的朋友fdisk命令在这里可以帮我们。在创建了基本的Linux分区之后，你需要通过t命令改变分区类型：

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 8e
Changed system type of partition 1 to 8e (Linux LVM)
```

```
Command (m for help): p
```

```
Disk /dev/sdc: 2147 MB, 2147992064 bytes
255 heads, 63 sectors/track, 261 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x4bc26133
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sdcl		1	261	2096451	8e	Linux LVM

```
Command (m for help):
```

8e分区类型表示这个分区将会被用作Linux LVM系统的一部分，而不是一个直接的文件系统，如你在前面看到的83分区。

下一步是用分区来创建真实的物理卷，这可以通过pvcreate命令来完成：

```
$ sudo pvcreate /dev/sdcl
Physical volume "/dev/sdcl" successfully created
$
```

pvcreate命令为PV定义了使用的物理卷。它简单地将分区标记成Linux LVM系统中的物理卷。你可以用pvdisplay命令来显示已创建的物理卷列表，如果你想看看你在这个过程中的进度的话。

2. 创建卷组

这个过程的下一步是从物理卷中创建一个或多个卷组。没有固定的规则说一定要为你的系统创建多少卷组，你可以将所有的可用物理卷加到一个卷组，或者你组合不同的物理卷创建多个卷组。

要从命令行创建卷组，你需要使用vgcreate命令。vgcreate命令要求一些命令行参数来定义卷组名以及你用来创建卷组的物理卷名：

```
$ sudo vgcreate Vol1 /dev/sdcl
Volume group "Vol1" successfully created
$
```

输出并没都叫人很兴奋。如果你想看看新创建的卷组的细节，可用vgdisplay命令：

```
$ sudo vgdisplay ---
--- Volume group ---
VG Name           Vol1
System ID
Format            lvm2
Metadata Areas   1
Metadata Sequence No 1
VG Access         read/write
VG Status         resizable
MAX LV
Cur LV
Open LV
Max PV
Cur PV
Act PV
VG Size          2.00 GB
PE Size          4.00 MB
Total PE         511
Alloc PE / Size  0 / 0
Free  PE / Size  511 / 2.00 GB
VG UUID          Cyi1HZ-YB40-BTUn-Wvti-4S6Q-bHHT-C113I0
```

\$

这个例子使用/dev/sdcl分区上创建的物理卷，创建了一个名为Vol1的卷组。

创建一个或多个卷组后，你就已经准备好创建逻辑卷了。

3. 创建逻辑卷

逻辑卷是Linux系统用来模拟物理分区以及保存文件系统的。Linux系统会像处理物理分区一样处理逻辑卷，允许你定义逻辑卷中的文件系统，然后将文件系统挂载到虚拟目录上。

要创建逻辑卷，使用lvcreate命令。虽然你通常不需要在其他Linux LVM命令中使用命令行选项，但lvcreate命令要求至少输入一些选项。表7-5显示了可用的命令行选项。

表7-5 lvcreate的选项

选 项	长选项名	描 述
-C	--chunksize	指定快照逻辑卷的单位大小
-C	--contiguous	设置或重置连续分配策略
-i	--stripes	指定条带数
-I	--stripesize	指定每个条带的大小

(续)

选 项	长选项名	描 述
-l	--extents	指定分配给新逻辑卷的逻辑块数，或者要用的逻辑块的百分比
-L	--size	指定分配给新逻辑卷的硬盘大小
-m	--minor	指定设备的次设备号
-M	--persistent	创建设备的镜像数
-n	--name	让次设备号一直有效
-p	--permission	指定新逻辑卷的名称
-r	--readahead	为逻辑卷设置读/写权限
-R	--regionsize	设置预读扇区数
-s	snapshot	指定将镜像分成多大的区
-z	--zero	创建镜像逻辑卷
		设置在新逻辑卷的前1 KB数据为零

虽然命令行选项看起来可能有点吓人，但大多数情况下你可以只用少数几个选项：

```
$ sudo lvcreate -l 100%FREE -n lvtest Vol1
Logical volume "lvtest" created
$
```

如果想查看你所创建的逻辑卷的详细情况，可用lvdisplay命令：

```
$ sudo lvdisplay
--- Logical volume ---
LV Name           /dev/Vol1/lvtest
VG Name           Vol1
LV UUID           usDxti-pAEj-fEIz-3kWV-LNAu-PFNx-2LGqNv
LV Write Access   read/write
LV Status         available
# open            0
LV Size           2.00 GB
Current LE        511
Segments          1
Allocation        inherit
Read ahead sectors auto
- currently set to 256
Block device      253:2
$
```

现在你可以看到你刚刚创建的逻辑卷。注意卷组名（Vol1）用来标识创建新逻辑卷是要使用的卷组。

-l参数定义了要为逻辑卷指定多少卷组的空闲空间。注意，你按这个卷组中空闲空间的百分比来指定这个值。本例中为新逻辑卷使用了所有的空闲空间。

你可以用-l参数来按可用空间的百分比来指定这个大小，或者用-L参数以字节、千字节（KB）、兆字节（MB）或吉字节（GB）为单位来指定实际的大小。-n参数允许你为逻辑卷提供一个名称（在本例中称作lvtest）。

4. 创建文件系统

在运行了lvcreate命令之后，逻辑卷已存在但没有文件系统。要做文件系统，你必须用要创建的文件系统对应的命令行程序：

```
$ sudo mkfs.ext4 /dev/Vol1/lvtest
mke2fs 1.41.9 (22-Aug-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
130816 inodes. 523264 blocks
26163 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=536870912
16 block groups
32768 blocks per group. 32768 fragments per group
8176 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 21 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
$
```

在创建了新的文件系统之后，你可以用标准Linux mount命令将这个卷挂载到虚拟目录中，就跟它是物理分区一样。唯一的不同是你需要用特殊的路径来标识逻辑卷：

```
$ sudo mount /dev/Vol1/lvtest test
$ cd test
$ ls -al
total 24
drwxr-xr-x. 3 root root 4096 2010-09-17 17:36 .
drwx-----.. 33 rich rich 4096 2010-09-17 17:37 ..
drwx-----.. 2 root root 16384 2010-09-17 17:36 lost+found
$
```

注意，mkfs.ext4和mount命令中用到的路径都有点奇怪。路径用卷组名和逻辑卷名一起来标识，而不是物理分区路径。一旦文件系统被挂载上了，你就可以访问虚拟目录的这块新区域了。

5. 修改LVM

鉴于使用Linux LVM来动态修改文件系统的这些好处，你会期望有些工具来完成这些。在Linux有一些可用的工具允许你修改现有的逻辑卷管理配置。

如果你无法通过一个很炫的图形化界面来管理你的Linux LVM环境，你并不是什么都干不了。你已经在本章学了一些Linux LVM命令行的可上手的命令。同样，有很多在安装了LVM之后用来管理LVM设置的其他命令行程序。表7-6列出了在Linux LVM包中能找到的共用命令。

表7-6 Linux LVM命令

命 令	功 能
vgchange	激活和禁用卷组
vgremove	删除卷组
vgextent	将物理卷加到卷组中
vgreduce	从卷组中删除物理卷
lvextend	增加逻辑卷的大小
lvreduce	减小逻辑卷的大小

通过使用这些命令行程序，你就能完全控制你的Linux LVM环境了。

注意 在手动增加或减小逻辑卷的大小时，要特别小心。存储在逻辑卷中的文件系统需要手动修复来处理大小上的改变。大多数文件系统包含重新调整文件系统格式的命令行程序，比如给ext2和ext3文件系统用的resize2fs程序。

7.4 小结

在Linux上处理存储设备需要懂一点文件系统。懂得如何在命令行下创建和处理文件系统在你工作在Linux系统上是能派得上用场。本章讨论了如何从Linux命令行处理文件系统。

Linux系统和Windows的不同之处在于它支持大量不同的存储文件和目录的方法。每个文件系统方法都有不同的特性，使得它适用于不同的情况。同时，每个文件系统方法都采用与存储设备交互的不同命令。

在将文件系统安装到存储设备之前，必须首先准备这个设备。fdisk命令用来将存储设备分区以使它们为文件系统准备好。在分区存储设备时，必须定义在上面使用什么类型的文件系统。

在完成存储设备分区后，你可以为该分区使用几种不同文件系统中的一种。最流行的Linux文件系统是ext3和ext4。这两个文件系统都提供了日志文件系统功能，使得它们在Linux系统崩溃时较少地遇到错误或问题。

在存储设备分区上直接创建文件系统的一个限制因素是，如果硬盘空间用完了，你不能轻易地改变文件系统的大小。但Linux支持逻辑卷管理，一种跨多个存储设备创建虚拟分区的方法。这种方法允许你轻松地扩展一个已有文件系统，而不用完全重新构建。Linux LVM包提供了跨多个存储设备创建逻辑卷的命令行命令。

现在你已经了解了核心的Linux命令行命令，差不多是时候开始编写一些shell脚本程序了。但在你开始编码前，还有一个我们需要讨论的元素：编辑器。如果你打算写shell脚本，你需要一个环境来完成你的杰作。下一章将讨论如何在不同的Linux环境中从命令行下安装和管理软件包。

本章内容

- 安装软件
- 使用Debian包
- 操作Red Hat包

在 Linux的早期，安装软件是一件痛苦的事。幸好，Linux开发人员已经通过把软件打包成更易于安装的预编译好的包，让我们的生活简单了一些。但在我们这边，仍需要一点工作来安装软件包，尤其是如果你准备从命令行下安装。本章将介绍Linux上能见到的各种包管理系统（Package Management System, PMS），以及用来安装、管理和删除软件用的命令行工具。

8.1 包管理基础

在深入了解Linux软件包管理之前，本章将先介绍一些基础知识。每个主要的Linux发行版都利用包管理系统的某些形式来控制安装软件应用和库。PMS利用一个数据库来记录：

- Linux系统上已安装了什么软件包；
- 每个包安装了什么文件；
- 每个已安装软件包的版本。

软件包存储在服务器上，并通过运行在本地Linux系统上的PMS工具通过互联网访问。这些服务器称为库（repository）。你可以用PMS工具来搜索新的软件包，或者是系统上已安装软件包的更新。

软件包通常都有使其能正确运行的必须提前安装的依赖关系或其他包。PMS工具将会检测这些依赖关系并在安装要求的包之前提供安装所有额外需要的软件包。

PMS不好的一面是现在并没有一个标准工具。不管你用的是哪个Linux发行版，到目前为止，本书中讨论的所有bash shell命令都能工作；而这却并不适用于软件包管理。

PMS工具和他们关联的命令在不同的Linux发行版上有很大的不同。Linux中广泛使用的两个主要PMS基础工具是dpkg和rpm。

基于Debian的发行版，比如Ubuntu和Linux Mint，在它们PMS工具的底层用的是dpkg命令。

这个命令会直接和Linux系统上的PMS交互，用来安装、管理和删除软件包。

基于Red Hat的发行版，比如Fedora、openSUSE以及Mandriva，在它们PMS的底层用的是rpm命令。类似于dpkg命令，rpm命令能够列出已安装包、安装新包和删除已有软件。

注意，这两个命令是它们各自PMS的核心，而不是整个PMS自身。许多使用dpkg或rpm方法的Linux发行版在这些基础命令之上构建了其他专有PMS工具来帮助生活简单点。后面几节将带你逐步了解你可能碰到的主流Linux发行版上的各种PMS工具命令。

8.2 基于 Debian 的系统

dpkg命令是基于Debian系PMS工具的核心。包含在这个PMS中的其他工具有：

- apt-get;
- apt-cache;
- aptitude。

到目前为止，最常用的命令行工具是aptitude，而这是有原因的。aptitude工具本质上是apt工具和dpkg的前端。dpkg是一个软件包管理系统工具，而aptitude则是一个完整的软件包管理系统。

命令行下使用aptitude命令会帮助你避免常见的软件安装问题，比如软件依赖关系缺失，系统环境不稳定以及其他一些不必要的麻烦。本节将会介绍如何在命令行下使用aptitude命令工具。

8.2.1 用aptitude管理软件包

Linux系统管理员面对的一个常见任务是判断系统上已经安装了什么软件包。幸运的是，aptitude有个很方便的交互式界面让这个任务变得很简单。

在shell提示符键入aptitude并按下回车键。你会被切到aptitude的全屏模式，如你在图8-1中看到的。

你可以用方向键来在菜单上移动。选择菜单选项Installed Packages来查看已安装了什么软件包。你可以看到几组软件包，比如编辑器等。每组后面的括号里有个数字，表示这个组包含多少个软件包。

使用方向键高亮显示一个组，按回车键来查看每个软件包分组。你会看到每个单独的软件包名称以及它们的版本号。在软件包上按回车键可以获得更详细的信息，比如软件包的描述、主页、大小和维护人员等。

当你看完了已安装软件包，按q键来退出显示。你可以继续用方向键和回车键来打开或关闭软件包和它们所在的分组。当你想退出了，按几次q键直到你收到弹出的屏幕“Really quit Aptitude?”。

如果你已经知道了系统上的那些软件包，只想快速显示某个特定包的详细信息，就没必要到aptitude的交互式界面。你可以在命令行下以单个命令的方式用aptitude：

```
aptitude show package_name
```

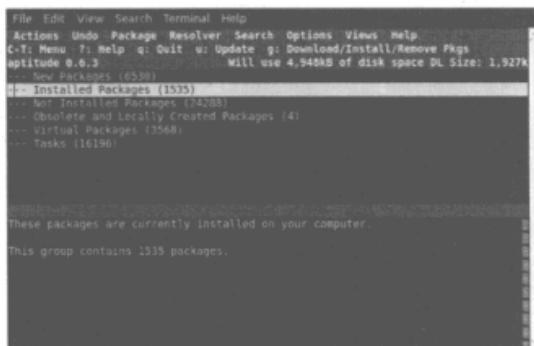


图8-1 aptitude主窗口

下面是显示包grub2-theme-mint的例子：

```
$ aptitude show grub2-theme-mint
Package: grub2-theme-mint
New: yes
State: installed
Automatically installed: no
Version: 1.0.3
Priority: optional
Section: misc
Maintainer: Clement Lefebvre <root@linuxmint.com>
Uncompressed Size: 442k
Description: Grub2 theme for Linux Mint
Grub2 theme for Linux Mint
```

\$

说明 aptitude show命令并不表明那个软件包已经在系统上安装了。它只显示从软件库中得到的详细的软件包信息。

你无法通过aptitude看到的一个细节是所有跟某个特定软件包关联的所有文件的列表。要得到这个列表，你需要用dpkg命令本身：

```
dpkg -L package_name
```

这里有个用dpkg列出作为grub2-theme-mint软件包一部分安装的所有文件的例子：

```
$
$ dpkg -L grub2-theme-mint
/
/boot
```

```

/boot/boot
/boot/boot/grub
/boot/boot/grub/linuxmint.png
/boot/grub
/boot/grub/linuxmint.png
/usr
/usr/share
/usr/share/doc
/usr/share/doc/grub2-theme-mint
/usr/share/doc/grub2-theme-mint/changelog.gz
/usr/share/doc/grub2-theme-mint/copyright
/etc
/etc/grub.d
/etc/grub.d/06_mint_theme
$
```

你同样可以进行反向操作——查找某个特定文件属于哪个软件包：

```
dpkg --search absolute_file_name
```

注意，必须用绝对文件路径来和它一起使用：

```

$ dpkg --search /boot/grub/linuxmint.png
grub2-theme-mint: /boot/grub/linuxmint.png
$
```

输出说明linuxmint.png文件是作为grub2-theme-mint包的一部分安装的。

8.2.2 用aptitude安装软件包

既然你已经了解了怎样在系统中列出软件包信息，本节将带你逐步了解怎样安装软件包。首先，你要确定准备安装的软件包名称。你怎么找一个特定的软件包呢？用aptitude命令加search选项：

```
aptitude search package_name
```

search选项之美在于，你无需在package_name边上加通配符。通配符会隐式添加。下面是用aptitude来查找wine软件包的例子：

```

$ aptitude search wine
p gnome-wine-icon-theme
p libkwinewin32l-api
p libkwinewin32l
p qdwine
p shiki-wine-theme
p wine
p wine-dev
p wine-gecko
p winel_0
p winel_0-dev
p winel_0-gecko
p winel_2
- red variation of the GNOME- ...
- library used by effects...
- Qt4 GUI for wine (W.I.N.E)
- red variation of the Shiki- ...
- Microsoft Windows Compatibility ...
```

```
p  wine1.2-dbg           - Microsoft Windows Compatibility ...
p  wine1.2-dev          - Microsoft Windows Compatibility ...
p  wine1.2-gecko         - Microsoft Windows Compatibility ...
p  winefish              - LaTeX Editor based on Bluefish
$
```

注意，在每个包名字之前都有一个p或一个i。如果你看到一个i，说明这个包现在已经安装到了你的系统上了。如果你看到一个p，说明有这个包但还没安装。如你在前一个列表中看到的，这个系统现在没有wine安装在上面，但这个包在软件库中有。

在系统上用aptitude从软件库中安装软件包可以像下例这样简单：

```
aptitude install package_name
```

一旦你通过search选项找到了软件包名称，只要将它通过install选项插入aptitude命令：

```
$ sudo aptitude install wine
The following NEW packages will be installed:
  cabextract{a} esound-clients{a} esound-common{a} gnome-exe-thumbnailer
{a}
  icoutils{a} imagemagick{a} libaudio2{a} libaudiofile0{a} libcdt4{a}
  libesd0{a} libgraph4{a} libgvc5{a} liblibase6{a} libmagickcore3-extra
{a}
  libmpg123-0{a} libnetpbm10{a} libopenall{a} libopenexr6{a}
  libpathplan4{a} libxdot4{a} netpbm{a} ttf-mscorefonts-installer{a}
  ttf-symbol-replacement{a} winbind{a} wine wine1.2{a} wine1.2-gecko{a}
0 packages upgraded, 27 newly installed, 0 to remove and 0 not upgraded.
Need to get 0B/27.6MB of archives. After unpacking 121MB will be used.
Do you want to continue? [Y/n/?] Y
Preconfiguring packages ...
...
All done, no errors.
All fonts downloaded and installed.
Updating fontconfig cache for /usr/share/fonts/truetype/mscorefonts
Setting up winbind (2:3.5.4-dfsg-lubuntu7) ...
 * Starting the Winbind daemon winbind
[ OK ]
Setting up wine (1.2-0ubuntu5) ...
Setting up gnome-exe-thumbnailer (0.6-0ubuntu1) ...
Processing triggers for libc-bin ...
ldconfig deferred processing now taking place
$
```

说明 在前面的列表中，aptitude命令之前用了sudo命令。sudo命令允许你以root用户身份运行一个命令。你可以用sudo命令来运行管理任务，比如安装软件。

要检查安装过程是否正确地执行，只要再次使用search选项就可以了。这次你可以看到在wine软件包之前有一个i，这说明它已经安装了。

你可能还注意到这里有另外一些包前面也有一个*i*。这是因为aptitude为我们自动解析了必要的包依赖关系并安装了需要的额外库和软件包。这是许多包管理系统都有的特别棒的功能。

8.2.3 用aptitude更新软件

aptitude可以帮忙解决安装软件时遇到的问题，而要协调更新有依赖关系的多个包就比较繁琐了。要安全地用软件库中的新版本更新系统上所有的软件包，可用safe-upgrade选项：

```
aptitude safe-upgrade
```

注意，这个命令不需要跟一个软件包名称作为参数。这是因为safe-upgrade选项会将所有已安装的包更新到软件库中的最新版本，更有利于系统稳定。

这里有个运行aptitude safe-upgrade命令的示例输出：

```
$ sudo aptitude safe-upgrade
The following packages will be upgraded:
 evolution evolution-common evolution-plugins gsfonts libevolution
 xserver-xorg-video-geode
6 packages upgraded. 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 9,312kB of archives. After unpacking 0B will be used.
Do you want to continue? [Y/n/?] Y
Get:1 http://us.archive.ubuntu.com/ubuntu/ maverick/main
 libevolution 1386 2.30.3-1ubuntu4 [2,096kB]
...
Preparing to replace xserver-xorg-video-geode 2.11.9-2
(using .../xserver-xorg-video-geode_2.11.9-3_1386.deb) ...
Unpacking replacement xserver-xorg-video-geode ...
Processing triggers for man-db ...
Processing triggers for desktop-file-utils ...
Processing triggers for python-gmenu ...
...
Current status: 0 updates [-6].
$
```

还有一些不那么保守的可用的软件升级选项。

- aptitude full-upgrade。
- aptitude dist-upgrade。

这些选项执行相同任务，将所有的软件包升级到最新版本。它们同safe-upgrade的区别在于它们不会检查包与包之间的依赖关系。整个包依赖关系问题非常麻烦。如果你不是很确定各种包的依赖关系，那还是坚持用safe-upgrade选项吧。

说明 显然，你应该定期地运行aptitude的safe-upgrade选项来保持系统到最新，但在一个全新的发行版安装之后运行尤其重要。通常在一个发行版的最近一次完整发布之后，会有更多的安全补丁和更新发布。

8.2.4 用aptitude卸载软件

用aptitude来卸载软件包与安装及更新它们一样容易。你要做的唯一选择是，是否要在之后保留软件数据和配置文件。

只删除软件包但不删除数据和配置文件，可用aptitude的remove选项。要删除软件包和相关的数据和配置文件，可用purge选项：

```
$ sudo aptitude purge wine
[sudo] password for user:
The following packages will be REMOVED:
cabextract[{\u} esound-clients[{\u} esound-common[{\u} gnome-exe-thumbnailer
{\u}]
icoutils[{\u} imagemagick[{\u} libaudio2[{\u} libaudiofile0[{\u} libcdt4[{\u}
libesd0[{\u} libgraph4[{\u} libgvc5[{\u} libilmbase6[{\u} libmagickcore3-extra
{\u}]
libmpg123-0[{\u} libnetpbm10[{\u} libopenal1[{\u} libopenexr6[{\u}
libpathplan4[{\u} libxdot4[{\u} netpbm[{\u} ttf-mscorefonts-installer[{\u}
ttf-symbol-replacement[{\u} winbind[{\u} wine[{\u} wine1.2[{\u} wine1.2-gecko
{\u}]
0 packages upgraded, 0 newly installed, 27 to remove and 6 not upgraded.
Need to get 0B of archives. After unpacking 121MB will be freed.
Do you want to continue? [Y/n/?] Y
(Reading database ... 120968 files and directories currently installed.)
Removing ttf-mscorefonts-installer ...
...
Processing triggers for fontconfig ...
Processing triggers for ureadahead ...
Processing triggers for python-support ...
$
```

要看软件包是否已经被删除，你可以再用aptitude search选项。如果你在软件包名称的前面看到一个c，那意味着软件已被删除，但配置文件尚未从系统中清除。前面是个p的话说明配置文件也被删除。

8.2.5 aptitude库

aptitude默认的软件库位置是在安装Linux发行版时设置的。库位置存储在文件/etc/apt/sources.list中。

很多情况下，你根本不需要添加或删除软件库，因此你不需要碰这个文件。但aptitude只会从这些库中下载文件。还有，在搜索要安装或更新的软件时，aptitude只会检查这些库。如果你需要为你的PMS添加一些额外的软件库，这里正是做这个的地方。

提示 Linux发行版开发人员在努力工作以保证添加到软件库的包版本不会互相冲突。通常通过库来升级或安装软件包是最安全的。即使在其他地方有更新的版本，你也尽可能不要安装，直到该版本在你的Linux发行版库中可用时。

下面是Ubuntu系统中sources.list文件的例子：

```
$ cat /etc/apt/sources.list
#deb cdrom:[Ubuntu 10.10 _Maverick Meerkat_ - Alpha i386
(20100921.1)]/ maverick main restricted
# See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade
# to
# newer versions of the distribution.

deb http://us.archive.ubuntu.com/ubuntu/ maverick main restricted
deb-src http://us.archive.ubuntu.com/ubuntu/ maverick main restricted

## Major bug fix updates produced after the final release of the
## distribution.
deb http://us.archive.ubuntu.com/ubuntu/ maverick-updates main
restricted
deb-src http://us.archive.ubuntu.com/ubuntu/ maverick-updates main
restricted
...
## This software is not part of Ubuntu, but is offered by third-party
## developers who want to ship their latest software.
deb http://extras.ubuntu.com/ubuntu maverick main
deb-src http://extras.ubuntu.com/ubuntu maverick main

deb http://security.ubuntu.com/ubuntu maverick-security main restricted
deb-src http://security.ubuntu.com/ubuntu maverick-security main
restricted
deb http://security.ubuntu.com/ubuntu maverick-security universe
deb-src http://security.ubuntu.com/ubuntu maverick-security universe
deb http://security.ubuntu.com/ubuntu maverick-security multiverse
deb-src http://security.ubuntu.com/ubuntu maverick-security multiverse
$
```

首先，注意文件里满是帮助性的注释和警告。指定库的源用下面的结构：

`deb (or deb-src) address distribution_name package_type_list`

deb或deb-src的值表明了软件包的类型。deb值说明这是一个编译后程序的源，而deb-src值说明这是一个源代码的源。

address条目是软件库的Web地址。distribution_name条目是这个特定软件库的发行版版本的名称。在这个例子中，发行版名称是maverick。这并不能说明你运行的发行版就是Ubuntu Maverick Meercat，它只是说明这个Linux发行版正在用Ubuntu Maverick Meercat软件库。举个例子，在Linux Mint的sources.list文件中，你能看到混用的Linux Mint和Ubuntu的软件库。

最后，package_type_list条目可能不止一个单词，它表明库里面有什么类型的包。举个例子，你可以看到的值有main、restricted、universe和partner。

当你需要给你的源文件添加软件库时，你可以自己发挥，但通常会带来问题。通常软件库网站或各种包开发人员网站会有那么一两行文本，你可以直接从他们的网站上复制，粘贴到你的sources.list文件中。最好选择安全的途径并且只复制/粘贴。

aptitude前端界面提供了智能命令行选项来和基于Debian的dpkg工具一起工作。现在是时候了解一下基于Red Hat的发行版的rpm工具和它的各种前端界面。

8.3 基于 Red Hat 的系统

和基于Debian的发行版类似，基于Red Hat的系统也有几种不同的可用前端工具。常见的有以下3种。

- yum：在Red Hat和Fedora中使用。
- urpm：在Mandriva中使用。
- zypper：在openSUSE中使用。

这些前端都是基于rpm命令行工具的。下一节会讨论如何用这些基于rpm的各种工具来管理软件包。重点是在yum上，但同样会包含zypper和urpm的信息。

8.3.1 列出已安装包

要找出系统上已安装的包，可在shell提示符下输入如下命令：

```
 yum list installed
```

输出的信息可能会在屏幕上一闪而过，所以最好是将已安装包的列表重定向到一个文件中。你可以用more或less命令（或一个GUI编辑器）来可控地查看这个列表。

```
 yum list installed > installed_software
```

要列出你的openSUSE或Mandriva发行版上的已安装包，可参考表8-1中的命令。遗憾的是，Mandriva中采用的urpm工具无法产生一个当前已安装软件列表。因此，你需要转向底层的rpm工具。

表8-1 如何用zypper和urpm列出已安装软件

描述	前端工具	命令
Mandriva	urpm	rpm -qa > installed_software
openSUSE	zypper	zypper search -I > installed_software

要找到某个特定软件包的详细信息，yum就非常突出了。不仅它能给出关于包的非常详尽的描述，而且你可以通过一条简单命令就能查看包是否已安装：

```
# yum list xterm
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Available Packages
xterm.i686 261-2.fc14 fedora
#
# yum list installed xterm
Loaded plugins: refresh-packagekit
Error: No matching Packages to list
#
```

用urpm和zypper列出详细软件包信息的命令见表8-2。你还可使用zypper命令的info选项从库中获得一份更详细的包信息。

表8-2 如何用zypper和urpm查看各种包详细信息

信息类型	前端工具	命 令
包信息	urpm	urpmq -i package_name
是否安装	urpm	rpm -q package_name
包信息	zypper	zypper search -s package_name
是否安装	zypper	同样命令，在Status列查找一个i

最后，如果你需要找出什么软件包提供了系统上的某个特定文件，万能的yum也可以做到。只要输入命令：

```
yum provides file_name
```

这里有个查找什么软件提供了配置文件/etc/yum.conf的例子：

```
#  
# yum provides /etc/yum.conf  
Loaded plugins: langpacks, presto, refresh-packagekit  
Adding en_US to language list  
yum-3.2.28-5.fc14.noarch : RPM installer/updater  
Repo : fedora  
Matched from:  
Filename : /etc/yum.conf
```

```
yum-3.2.28-5.fc14.noarch : RPM installer/updater  
Repo : installed  
Matched from:  
Other : Provides-match: /etc/yum.conf
```

```
#
```

yum会查看两个分开的库：fedora和installed。从两个库中答案都是：该文件是yum软件包提供的。

8.3.2 用yum安装软件

用yum安装软件包简单得令人难以置信。下面的简单命令会从库安装某个软件包、所有它需要的库以及包依赖关系：

```
yum install package_name
```

下面是安装xterm包的例子：

```
$ su -
Password:
# yum install xterm
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
fedora/metalink | 20 kB 00:00
fedora | 4.3 kB 00:00
fedora/primary_db | 11 MB 01:57
updates/metalink | 16 kB 00:00
updates | 4.7 kB 00:00
updates/primary_db | 3.1 MB 00:30
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package xterm.i686 0:261-2.fc14 set to be installed
...
Installed:
xterm.i686 0:261-2.fc14

Complete!
#
#
```

说明 在以上代码清单中，在运行yum命令之前，我们用了su-命令。这个命令允许你切换到root用户。在Linux系统上，#表明你是以root用户身份登录的。你应该在运行管理性的任务时临时切换到root用户，比如安装和更新软件。sudo命令也是一个选择。

你也可以手动下载rpm安装文件并用yum安装。这称为本地安装（local installation）。基本的命令是：

```
yum localinstall package_name.rpm
```

你可以发现yum的优点之一是，它使用非常有逻辑的、用户友好的命令。

表8-3显示了如何用urpmi和zypper执行包安装。注意，如果不是以root用户身份登录，你会在使用urpmi时得到一个“command not found”的错误消息。

表8-3 如何用zypper和urpmi安装软件

前端工具	命 令
urpmi	urpmi package_name
zypper	zypper install package_name

8.3.3 用yum更新软件

在大多数Linux发行版上，如果你是在GUI上工作，你会看到一些好看的小通知图标，告诉你需要更新了。在命令行下，它需要一点额外的工作。

要列出所有针对已安装包的可用更新，输入如下命令：

```
yum list updates
```

如果这个命令没有输出那就太好了，因为它说明你没有任何需要更新的！但如果你发现某个特定软件包需要更新，输入如下命令：

```
yum update package_name
```

如果你想更新所有列在更新列表中的包，只要输入如下命令：

```
yum update
```

Mandriva和openSUSE上用来更新软件包的命令列在了表8-4中。在使用urpm时，软件库数据库会自动更新，软件包也会更新。

表8-4 如何用zypper和urpm更新软件

前端工具	命 令
urpm	urpmi --auto-update --update
zypper	zypper update

8.3.4 用yum卸载软件

yum工具还提供了简单地卸载在你系统上不再想要的应用。和aptitude一样，你需要决定是否保留软件包的数据和配置文件。

只删除软件包而保留配置文件和数据文件，用如下命令：

```
yum remove package_name
```

要删除软件和它所有的文件，用erase选项：

```
yum erase package_name
```

你能在表8-5中发现，用urpm和zypper删除软件同样简单。这两个工具起的作用类似于yum的erase选项。

表8-5 如何用zypper和urpm卸载软件

前端工具	命 令
urpm	urpme package_name
zypper	zypper remove package_name

虽然有了PMS包生活变得相当简单，但并非就一直不会遇到问题了。偶尔事情也会出错，幸运的是，有解决的办法。

8.3.5 处理损坏的包依赖关系

有时在安装多个软件包时，某个包的软件依赖关系可能会被另一个包的安装覆盖掉。这称为

损坏的包依赖关系 (broken dependency)。

如果你的系统出现了这个问题，首先试试下面的命令：

```
 yum clean all
```

然后试着用yum命令的update选项。有时，只要清理了放错位置的文件就可以了。

如果这还解决不了问题，试试下面的命令：

```
 yum deplist package_name
```

这个命令显示了所有包的库依赖关系以及什么软件可以提供这些库依赖关系。一旦你知道了某个包需要的库，你就能安装它们了。下面是一个判断xterm包依赖关系的例子：

```
# yum deplist xterm
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Finding dependencies:
package: xterm.i686 261-2.fc14
dependency: libutempter.so.0
provider: libutempter.i686 1.1.5-4.fc12
dependency: rtld(GNU_HASH)
provider: glibc.i686 2.12.90-17
provider: glibc.i686 2.12.90-21
dependency: libc.so.6(GLIBC_2.4)
provider: glibc.i686 2.12.90-17
provider: glibc.i686 2.12.90-21
...
dependency: /bin/sh
provider: bash.i686 4.1.7-3.fc14
dependency: libICE.so.6
provider: libICE.i686 1.0.6-2.fc13
dependency: libXmu.so.6
provider: libXmu.i686 1.0.5-2.fc13
dependency: libc.so.6(GLIBC_2.3)
provider: glibc.i686 2.12.90-17
provider: glibc.i686 2.12.90-21
dependency: libXaw.so.7
provider: libXaw.i686 1.0.6-4.fc12
dependency: libX11.so.6
provider: libX11.i686 1.3.4-3.fc14
dependency: libc.so.6(GLIBC_2.2)
provider: glibc.i686 2.12.90-17
provider: glibc.i686 2.12.90-21
#
```

如果这样还没解决问题，你还有最后一个工具：

```
 yum update --skip-broken
```

--skip-broken选项允许你忽略依赖关系损坏的那个包而更新其他软件包。这可能没法帮助损坏的包，但至少你可以更新系统上的其他包了。

在表8-6中列出了用urpm和zypper来尝试修复损坏的依赖关系的命令。用zypper时，这里只有唯一的命令来验证和修复损坏的依赖关系。用urpm时，如果clean选项不工作，你可以跳过那

些烦人的有问题包的更新。要这么做的话，你必须将有问题包的名字添加到文件/etc/urpmi/skip.list中。

表8-6 用zypper和urpm修复损坏的依赖关系

前端工具	命 令
urpm	urpmi -clean
Zypper	zypper verify

8.3.6 yum软件库

类似于aptitude系统，在安装时yum就会建立它的软件仓库。对于大多数目的，这些预装的库就能很好地满足你的需要了。但如果或当你需要从另一个库安装程序，你还需要知道一些东西。

提示 聪明的系统管理员会坚持使用通过审核的库。通过审核的库是指该发行版官方网站上指定的库。如果你添加了未通过审核的库，你就失去了稳定性方面的保证，可能陷入损坏的依赖关系惨剧中。

要知道你现在正从什么库中获取软件，输入如下命令：

```
yum repolist
```

如果你没找到可以下载你需要的软件的库，你可以编辑一下配置文件。yum库定义文件位于/etc/yum.repos.d。你需要添加正确的URL并获得必要的加密密钥。

好的库网站，比如rpmfusion.org，会列出使用它们的所有必要步骤。有时这些库网站会提供一个可下载并用yum localinstall命令安装的rpm文件。rpm文件的安装过程会做所有的库设置工作。现在方便多了。

urpm称它的库为媒体（media）。查看urpm媒体和zypper的库的命令列在了表8-7中。注意，用这两个前端工具时不需要编辑配置文件。相反，要添加一个媒体或库，只要在命令中输入就可以了。

表8-7 zypper和urpm的库

动 作	前端工具	命 令
显示库	urpm	urpmq --list-media
添加库	urpm	urpmi.addmedia path_name
显示库	zypper	zypper repos
添加库	zypper	zypper addrepo path_name

基于Debian的和基于Red Hat的系统都使用包管理系统来简化管理软件的过程。现在我们就要迈出包管理系统的步伐，看看稍微麻烦一点的，直接从源码安装。

8.4 从源码安装

第4章中讨论了tarball包——如何通过tar命令行命令来创建它们和解压它们。在好用的rpm和dpkg工具出现之前，管理员需要知道如何从tarball来解压和安装软件。

如果你经常在开源软件环境中工作，你就很有可能会遇到软件压缩成一个tarball的情形。本节就带你逐步了解解压和安装打包成tarball的软件包的过程。

在这个例子中，我们会用软件包sysstat。Sysstat工具是个非常好用的提供了一系列系统监测工具的软件包。

首先你必须下载sysstat的tarball到你的Linux系统上。你经常能在各种不同的Linux网站上找到sysstat包，但通常最好是直接到程序的源网站下载。在本例中，就是网站<http://sebastien.godard.pagesperso-orange.fr/>。

单击Download（下载）链接，你就会到有下载文件的页面。写本书时的当前版本是9.1.5，发行文件名是sysstat-9.1.5.tar.gz。

单击链接下载文件到你的Linux系统上。下载完这个文件，就可以解压它了。

要解压一个软件tarball，用标准的tar命令：

```
# tar -zvxf sysstat-9.1.5.tar.gz
sysstat-9.1.5/
sysstat-9.1.5/sar.c
sysstat-9.1.5/iostat.c
sysstat-9.1.5/sadc.c
sysstat-9.1.5/sa.h
sysstat-9.1.5/iconfig
sysstat-9.1.5/CHANGES
sysstat-9.1.5/COPYING
sysstat-9.1.5/CREDITS
sysstat-9.1.5/sa2.in
sysstat-9.1.5/README
sysstat-9.1.5/crontab.sample
sysstat-9.1.5/nls/
...
sysstat-9.1.5/nfsiostat.c
sysstat-9.1.5/sysstat-9.1.5.lsm
sysstat-9.1.5/cifsiostat.c
sysstat-9.1.5/nfsiostat.h
sysstat-9.1.5/cifsiostat.h
sysstat-9.1.5/sysstat-9.1.5.spec
#
```

既然这个tarball已经解压了，而且文件已经顺利地放到了一个叫sysstat-9.1.5的目录中，你可以跳到那个目录下继续了。

首先，用cd命令跳到这个新目录中，然后列出这个目录的内容：

```
$ cd sysstat-9.1.5
$ ls
activity.c      INSTALL      prf_stats.h  sysconfig.in
build           ioconf.c     pr_stats.c   sysstat-9.1.5.lsm
```

```

CHANGES      ioconf.h    pr_stats.h   sysstat-9.1.5.spec
cifsiostat.c iostat.c    rd_stats.c  sysstat.cron.daily.in
cifsiostat.h iostat.h    rd_stats.h  sysstat.crond.in
common.c     Makefile.in README      sysstat.cron.hourly.in
common.h     man          sal.in     sysstat.in
configure     mpstat.c    sa2.in     sysstat.ioconf
configure.in  mpstat.h    sa_common.c sysstat.sysconfig.in
contrib       nfsiostat.c sadc.c     TODO
COPYING      nfsiostat.h sadf.c     version.in
CREDITS      nls          sadf.h    xml
crontab.sample pidstat.c sa.h      -
FAQ          pidstat.h  sar.c     -
iconfig      prf_stats.c sa_wrap.c
$
```

在这个目录的列表中，你应该能看到典型的README或AAAREADME文件。读这个文件非常重要的。这个文件中的就是你完成软件安装所需要的实际指令。

按照README文件中的建议，下一步是为你的系统配置sysstat。它会检查你的Linux系统来保证除了用来编译源代码的合适的编译器外，它还有正确的包依赖关系：

```

# ./configure
Check programs:
.
.
.
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
.
.
.
checking for ANSI C header files... (cached) yes
checking for dirent.h that defines DIR... yes
checking for library containing opendir... none required
checking ctype.h usability... yes
checking ctype.h presence... yes
checking for ctype.h... yes
checking errno.h usability... yes
.
.
.
Check library functions:
.
.
.
checking for strchr... yes
checking for strcspn... yes
checking for strstr... yes
checking for strstr... yes
checking for sensors support... yes
.
.
.
Check configuration:
.
.
.
config.status: creating Makefile

Sysstat version:          9.1.5
Installation prefix:      /usr/local
rc directory:             /etc/rc.d
Init directory:            /etc/rc.d/init.d
Configuration directory:  /etc/sysconfig
Man pages directory:      /usr/local/man
Compiler:                 gcc
Compiler flags:           -g -O2
```

#

如果哪里有错了，configure步骤会显示一条错误消息说明缺什么。

下一步就是用make命令来构建各种二进制文件。make命令会编译源码，然后链接器会为这个包创建最终的可执行文件。和configure命令一样，make命令会在编译和链接所有的源码文件时产生大量的输出：

```
# make
...
gcc -o nfsiostat -g -O2 -Wall -Wstrict-prototypes -pipe
-O2 nfsiostat.o librdrstats.o libsyscom.a -s
gcc -o cfsiostat.o -c -g -O2 -Wall -Wstrict-prototypes -pipe
-O2 -DSA_DIR=\"/var/log/sa\"
-DSADC_PATH=\"/usr/local/lib/sa/sadc\" cfsiostat.c
gcc -o cfsiostat -g -O2 -Wall -Wstrict-prototypes -pipe
-O2 cfsiostat.o librdrstats.a libsyscom.a -s
#
```

make步骤结束时，你就在目录下有了实际可用的sysstat软件程序。然而，从那个目录下运行这个程序有点不方便。和现在不同的是，你想将它安装到你Linux系统上一个常见的位置。要做到这样，你必须以root用户身份登录（或者用sudo命令，如果你的Linux发行版偏好这个的话），然后用make命令的install选项：

```
# make install
mkdir -p /usr/local/man/man1
mkdir -p /usr/local/man/man8
rm -f /usr/local/man/man8/sa1.8*
install -m 644 -g man/man/sa1.8 /usr/local/man/man8
rm -f /usr/local/man/man8/sa2.8*
install -m 644 -g man/man/sa2.8 /usr/local/man/man8
rm -f /usr/local/man/man8/sadc.8*
install -m 644 -g man/man/sadc.8 /usr/local/man/man8
rm -f /usr/local/man/man1/sar.1*
...
install -m 644 sysstat.sysconfig /etc/sysconfig/sysstat
install -m 644 CHANGES /usr/local/share/doc/sysstat-9.1.5
install -m 644 COPYING /usr/local/share/doc/sysstat-9.1.5
install -m 644 CREDITS /usr/local/share/doc/sysstat-9.1.5
install -m 644 README /usr/local/share/doc/sysstat-9.1.5
install -m 644 FAQ /usr/local/share/doc/sysstat-9.1.5
install -m 644 *.lsm /usr/local/share/doc/sysstat-9.1.5
#
```

现在sysstat包已经安装在系统上了。虽然不像通过PMS安装软件包那样简单，通过tarball安装软件也没那么难。

8.5 小结

本章讨论了如何用软件包管理系统（PMS）来在命令行下安装、更新或删除软件。虽然大部分Linux发行版都使用GUI工具来进行软件包管理，你也可以在命令行下执行包管理。

基于Debian的Linux发行版使用dpkg工具来在命令行下和PMS交互。dpkg工具的一个前端是aptitude，它提供了处理dpkg格式软件包的简单命令行选项。

基于Red Hat的Linux发行版都基于rpm工具但在命令行下使用不同的前端工具。Red Hat和Fedora用yum来安装和管理软件包。openSUSE发行版采用zypper来管理软件，而Mandriva发行版采用urpm。

本章最后以关于如何安装仅以源代码tarball形式发布的软件包的讨论结束。tar命令可以从tarball中解压出源代码文件，然后configure和make命令从源代码中构建出最终的可执行程序。

下章将会讲述Linux发行版中可用的不同编辑器。在你开始编写shell脚本时，知道使用什么可用的编辑器将派得上用场。

本章内容

- Vim编辑器
- Emacs编辑器
- KDE系编辑器
- GNOME编辑器

在 开始shell脚本编程之前，你必须知道如何在Linux上使用至少一个文本编辑器。对如何使用查找、剪切和粘贴等好用的功能了解得越多，你就能越快地编写shell脚本。本章将讨论在Linux中能见到的主要文本编辑器。

9.1 Vim 编辑器

如果你工作在命令行模式下，可能会想了解至少一个在Linux控制台上操作的文本编辑器。Vi编辑器是Unix系统上早先的编辑器。它使用控制台图形模式来模拟文本编辑窗口，允许查看文件中的行，在文件中移动，以及插入、编辑和替换文本。

尽管它可能是世界上最复杂的编辑器（至少讨厌它的人是这么认为的），但它提供了许多功能，反使其成为多年来Unix管理员的支柱性工具。

在GNU项目将Vi编辑器移植到开源世界时，他们决定对其作一些改进。由于它不再是以前Unix中的那个原始的Vi编辑器了，开发人员也就将它重命名为Vi improved，或Vim。

为了方便使用，几乎所有Linux发行版都创建了一个名为vi的别名，指向vim程序：

```
$ alias vi  
alias vi='vim'  
$
```

本节将会带你逐步了解使用Vim编辑器编辑文本shell脚本文件的基础知识。

9.1.1 Vim基础

Vim编辑器在内存缓冲区中处理数据。启动Vim编辑器，只要键入vim命令（或vi，如果这个

别名存在的话)和你要编辑的文件的名字:

```
$ vim myprog.c
```

如在启动Vim时未指定文件名,或者这个文件不存在,Vim会新开一段缓冲区域来编辑。如果你在命令行下指定了一个已有文件的名字,Vim会将文件的整个内容都读到一块缓冲区域来准备编辑,如图9-1所示。

```

rich@rich-desktop: ~
File Edit View Terminal Help
#include <stdio.h>
int main()
{
    int i;
    int factorial = 1;
    int number = 5;

    for(i = 1; i <= number; i++)
    {
        factorial = factorial * i;
    }

    printf("The factorial of %d is %d\n", number, factorial);
    return 0;
}

```

"myprog.c" 36 lines, 237 characters

图9-1 Vim主窗口

Vim编辑器为这个会话检测了终端的类型(参见第2章),并用全屏模式来将整个控制台窗口作为编辑器区域。

最初的Vim编辑窗口显示了文件的内容,并在窗口的底部显示了一条消息行。如果文件内容并未占据整个屏幕,Vim会在非文件内容行放置一个波浪线(如图9-1所示)。

底部的消息行显示了在编辑文件的信息,根据文件的状态以及Vim安装时的默认设置。如果文件是新建的,会出现消息[New File]。

Vim编辑器有两种操作模式:

- 普通模式;
- 插入模式。

当你刚打开要编辑的文件时(或新建一个文件时),Vim编辑器会进入普通模式。在普通模式中,Vim编辑器会将按键解释成命令(本章后面会讨论更多)。

在插入模式下,Vim会将你在当前光标位置输入的每个键都插入到缓冲区。要进入插入模式,按下*i*键。要退出插入模式回到普通模式,按下键盘上的退出键(ESC键,也就是Escape键)就可以了。

在普通模式中,你可以用方向键来在文本区域移动光标(只要Vim能正确识别你的终端类型)。如果你恰巧在一个古怪的没有定义方向键的终端连接上,也不是完全没有希望。Vim命令

中有用来移动光标的命令。

- h: 左移一个字符。
- j: 下移一行 (文本中的下一行)。
- k: 上移一行 (文本中的上一行)。
- l: 右移一个字符。

在大的文本文件中一行一行地移动会特别麻烦。幸而Vim提供了一些命令来帮助提高速度。

- PageDown (或Ctrl+F): 下翻一屏数据。
- PageUp (或Ctrl+B): 上翻一屏数据。
- G: 移到缓冲区的最后一行。
- num G: 移动到缓冲区中的第num行。
- gg: 移到缓冲区的第一行。

Vim编辑器在普通模式下有个特别的功能叫命令行模式。命令行模式提供了一个可供输入额外命令来控制Vim中行为的交互式命令行。要进入命令行模式，在普通模式下按下冒号键。光标会移动到消息行，冒号出现了，等待输入命令。

在命令行模式下有几个命令来将缓冲区的数据保存到文件中并退出Vim。

- q: 如果未修改缓冲区数据，退出。
- q!: 取消所有对缓冲区数据的修改并退出。
- w filename: 将文件保存到另一个文件名下。
- wq: 将缓冲区数据保存到文件中并退出。

了解了这些基本的Vim命令后，你可能就理解为什么有人会痛恨Vim编辑器了。要用到Vim的全部功能，你必须知道大量鲜为人知的命令。然而一旦了解了一些基本的Vim命令，无论是什么环境，你都能快速在命令行下直接修改文件了。另外，一旦适应了敲入命令，在命令行下将数据和编辑命令一起输入就跟第二天性一样，再使用鼠标反倒觉得很奇怪。

9.1.2 编辑数据

在插入模式下，你可以向缓冲区插入数据。然而有时你需要在将数据输入到缓冲区中后添加或删除它。在普通模式下，Vim编辑器提供了一些命令来编辑缓冲区中的数据。表9-1列出了一些常用的Vim编辑命令。

表9-1 Vim编辑命令

命 令	描 述
x	删除当前光标所在位置的字符
dd	删除当前光标所在行
dw	删除当前光标所在位置的单词
d\$	删除当前光标所在位置至行尾的内容
J	删除当前光标所在行行尾的换行符 (拼接行)

(续)

命 令	描 述
u	撤销前一编辑命令
a	在当前光标后追加数据
A	在当前光标所在行行尾追加数据
r char	用char替换当前光标所在位置的单个字符
R text	用text覆盖当前光标所在位置的数据，直到按下ESC键

有些编辑命令允许使用数字修饰符来指定重复该命令多少次。比如，命令2x会删除从光标当前位置开始的两个字符，命令5dd会删除从光标当前所在行开始的5行。

警告 在Vim编辑器中使用PC键盘上的退格键（Backspace键）和删除键（Delete键）时要注意。

Vim编辑器通常会将删除键识别成x命令的功能，删除当前光标所在位置的字符。通常，Vim编辑器不会识别退格键。

9.1.3 复制和粘贴

现代编辑器的标准功能之一是剪切或复制数据，然后粘贴在文本中的其他地方。Vim编辑器也可以这么做。

剪切和粘贴相对容易一些。你已经看到表9-1中用来从缓冲区中删除数据的命令。但在Vim删除数据时，实际上它会将数据保存在单独的一个寄存器中。你可以用p命令来取回数据。

举例来说，可以用dd命令来删除一行文本，然后把光标移动到缓冲区的某个要放置该行文本的位置，之后用p命令。p命令会将文本插入到当前光标所在行之后。你可以将它和任何删除文本的命令一起搭配使用。

复制文本则要稍微复杂点。Vim中复制命令是y（代表yank）。你可以与y使用和d命令相同的第二字符（yw表示复制单词，y\$表示复制到行尾）。在复制文本后，把光标移动到你想放置文本的地方，输入p命令。复制的文本就会出现在该位置。

复制的复杂之处在于，由于不会影响到你复制的文本，你没法看见到底是怎么操作的。你很难确定你到底复制了什么，直到将它粘贴到其他地方才能明白。但Vim还有另外一个功能来解决这个问题。

可视模式会在你移动光标时高亮显示文本。你可以用可视模式来选取要复制的文本。要进入可视模式，移动光标到要开始复制的位置，并按下v键。你会注意到光标所在位置的文本已经被高亮显示了。下一步，移动光标来覆盖你想要复制的文本（甚至可以向下移动几行来复制更多行的文本）。在移动光标时，Vim会高亮显示复制区域的文本。在覆盖了要复制的文本后，按y键来激活复制命令。现在寄存器中已经有了要复制的文本，移动光标到你要放置的位置，使用p命令来粘贴。

9.1.4 查找和替换

你可以使用Vim查找命令来轻松查找缓冲区中的数据。要输入一个查找字符串，按下斜线(/)键。光标会跑到消息行，然后Vim会显示斜线。在输入你要查找的文本后，按回车键。Vim编辑器会有3种回应。

- 如果要查找的文本出现在光标当前位置之后，则光标会跳到该文本出现的第一个位置。
- 如果要查找的文本未在光标当前位置之后出现，则光标会绕过文件末尾，显示在该文本出现的第一个位置（并用一条消息显示）。
- 输出一条错误消息，说明在文件中没有找到要找的文本。

要继续查找同一个单词，按下斜线键，然后按回车键。或者使用n键，表示下一个(next)。

替换命令允许你快速用另一个单词来替换文本中的某个单词。要进入到替换命令，你必须是在命令行模式下。替换命令的格式是：

```
:s/old/new/
```

Vim编辑器会跳到old第一次出现的地方并用new来替换。可以对替换命令作一些修改来替换多处要替换的文本。

- :s/old/new/g：一行命令替换所有old。
- :n..ms/old/new/g：替换行号n和m之间所有old。
- :%s/old/new/g：替换整个文件中的所有old。
- :%s/old/new/gc：替换整个文件中的所有old，但在每次出现时提示。

如你所看到的，对于一个命令行文本编辑器而言，Vim包含了不少高级功能。由于每个Linux发行版都会包含它，所以应该至少了解一下Vim编辑器的一些基本用法，这样一来，不管你所处的环境如何，你就总能编辑脚本了。

9.2 Emacs 编辑器

Emacs编辑器是一个极其流行的编辑器，它在Unix出现之前就已存在。开发人员非常喜欢它，于是就将其移植到了Unix环境中，现在它也被移植到了Linux环境中。跟Vi很像，Emacs编辑器一开始也被作为控制台编辑器，但已经迁移到了图形化世界。

Emacs编辑器仍然提供最早的命令行模式编辑器，但现在它也能使用图形化的X Window窗口，从而允许在图形化环境中编辑文本。当你从命令行启动Emacs编辑器时，编辑器一般会判断是否有X Window会话，以便启动图形模式。如果没有，它会以控制台模式启动。

本节将介绍控制台模式和图形模式的Emacs编辑器，这样你就知道如何使用任意一个了。

9.2.1 在控制台上使用Emacs

控制台模式版本的Emacs同样也要使用大量按键命令来执行编辑功能。Emacs编辑器使用包括控制键（PC键盘上的Ctrl键）和Meta键的按键组合。在大多数PC终端模拟器中，Meta键被映射

到了PC的Alt键。Emacs官方文档将Ctrl键缩写为C-，而Meta键缩写为M-。所以，如果你要输入Ctrl-x组合键，文档会显示成C-x。为了避免冲突，本章将会沿用这种用法。

1. Emacs基础

要在命令行用Emacs编辑文件，输入：

```
$ emacs myprog.c
```

随Emacs控制台模式窗口一起出现的是一个简短的介绍以及帮助界面。不要紧张，只要按下任意键，Emacs会将文件加载到工作缓冲区并显示文本，如图9-2所示。

```
int factorial(int number)
{
    int i;
    int factorial = 1;
    int number = 5;

    for(i = 1; i <= number; i++)
    {
        factorial *= i;
    }

    printf("The factorial of %d is %d\n", number, factorial);
    return 0;
}
```

图9-2 用控制台模式的Emacs编辑器编辑文件

你会注意到在控制台模式窗口的顶部显示典型的菜单栏。遗憾的是，你无法在控制台模式中使用菜单栏，只能在图形模式中使用。

说明 假如虽有图形化桌面，但你却更倾向于在控制台模式而不是X Window模式下使用Emacs，则可在命令行使用 -nw选项。

不同于Vim编辑器的是，你必须在输入命令和插入文本之间切换插入模式，因为Emacs编辑器只有一个模式。如果你输入可打印字符，Emacs就将它插入到光标当前位置，如果你输入一个命令，Emacs就执行命令。

要在缓冲区域移动光标，你可以用方向键和PageUp以及PageDown键，如果Emacs正确地检测到了你的终端模拟器。如果未能正确监测，有一些命令可用来移动光标。

- C-p：上移一行（文本中的前一行）。
- C-b：左移一字符。
- C-f：右移一字符。
- C-n：下移一行（文本中的下一行）。

还有一些在文本中让光标进行较大跳跃的命令。

- M-f: 右移到下个单词。
- M-b: 左移到上个单词。
- C-a: 移至行首。
- C-e: 移至行尾。
- M-a: 移至当前句首。
- M-e: 移至当前句尾。
- M-v: 上翻一屏。
- C-v: 下翻一屏。
- M-: 移至文本的首行。
- M->: 移至文本的尾行。

还有几个你应该知道的将编辑器缓冲区保存至文件并退出Emacs的命令。

- C-x C-s: 保存当前缓冲区到文件。
- C-z: 退出Emacs并保持在这个会话中继续运行，以便你切回。
- C-x C-c: 退出Emacs并停止该程序。

你会注意到这些功能中有两条需要两个键命令。C-x命令称作扩展命令（extend command）。这为我们提供了另外一组命令。

2. 编辑数据

Emacs编辑器在插入和删除缓冲区中的文本时非常强大。要插入文本，只需将光标移动到你想插入文本的位置就可以开始输入了。要想删除（Delete）文本，Emacs使用退格键来删除光标当前所在位置之前的字符，使用删除键来删除光标当前位置之后的字符。

Emacs编辑器还有剪切^①文本的命令。删除文本和剪切文本的差别在于，当你剪切文本时，Emacs会将其放在一个临时区域，你可以取回（参见下一小节）；而删除的文本则会永远消失。

有几个命令可用来剪切缓冲区中的文本。

- M-Backspace: 剪切光标当前所在位置之前的单词。
- M-d: 剪切光标当前所在位置之后的单词。
- C-k: 剪切光标当前所在位置至行尾的文本。
- M-k: 剪切光标当前所在位置至句尾的文本。

Emacs编辑器还包括了一种独特的块剪切的方法。移动光标到你要剪切的区域的开始位置并按下C-@或C-Spacebar键，然后移动光标到你要剪切的区域的结束位置并按下C-w命令键。这两个位置之间的文本都将被剪切。

如果你恰好在剪切文本时弄错了，使用C-u命令就能撤销剪切命令，返回到剪切前的状态。

3. 复制和粘贴

你已经看到了如何从Emacs缓冲区域剪切数据，现在是时候看看如何将它粘贴到其他地方了。

^① 英文为kill，Emacs专有的说法。——译者注

遗憾的是，如果你用过Vim编辑器，那么使用Emacs编辑器可能会叫你有点困惑。

不幸的巧合是，在Emacs中粘贴数据也叫yanking。在Vim编辑器中，复制叫做yanking。如果你恰好要用两种编辑器，这可能会比较难记。

当你用剪切命令剪切了某个数据后，将光标移动到你要粘贴数据的位置，用C-y来粘贴。这会将文本从临时区域取出并将其粘贴在光标的当前位置。C-y命令会取出最后一个剪切命令存下的文本。如果你执行了多个剪切命令，你可以用M-y命令来循环选择它们。

要复制文本，只需将它粘贴到剪切它的地方然后移动到新的位置并再使用一次C-y命令即可。如果需要，你可以粘贴文本任意多次。

4. 查找和替换

在Emacs编辑器中查找文本可用C-s和C-r命令。C-s命令会在缓冲区域中从光标当前位置到缓冲区尾部执行前向查找，而C-r命令会在缓冲区域从光标当前所在位置到缓冲区起始执行后向查找。

当输入C-s或C-r任一命令时，底行会出现一个提示，询问要查找的文本。Emacs可以执行两种类型的查找。

在渐进式（incremental）查找中，Emacs编辑器在你键入单词时实时地执行文本查找。当你键入第一个字母时，它会高亮显示缓冲区中所有该字母出现的地方。当你键入第二个字母时，它会高亮显示文本中所有出现这两个字母组合的地方。如此往复，直到输完了要查找的文本。

在非渐进式（non-incremental）查找中，在C-s或C-r命令后按下回车键。这会将查找问询锁定在底行区域，并允许你在查找前输入完整的要查找的文本。

要用一个新字符串来替换一个已有文本字符串，你必须用M-x命令。这个命令要求一个文本命令和参数。

该文本命令是replace-string。输入该命令后，按下次回车键，Emacs会询问要替换的已有字符串。输入之后，再按一次回车键，Emacs会询问用来替换的新字符串。

5. 在Emacs中使用缓冲区

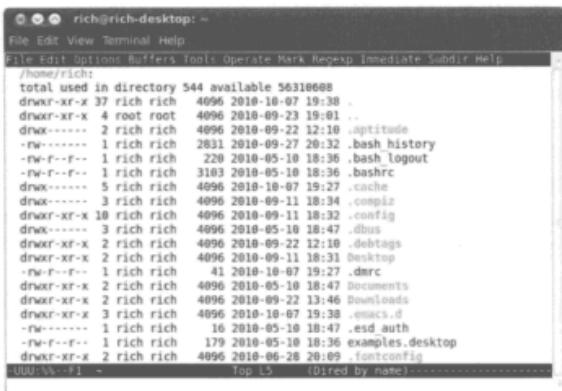
通过多个缓冲区域，Emacs编辑器允许同时编辑多个文件。你可以把文件加载到一个缓冲区中，编辑时在多个缓冲区中切换。

当你在Emacs中时，要加载一个新的文件到缓冲区，可用C-x C-f组合键。这是Emacs的查找文件（Find a File）模式。它会把你带到窗口的底行，允许你输入要开始编辑的文件的名称。如果你不知道文件的名称或位置，可以按下次回车键。它会在编辑窗口启动一个文件浏览器，如图9-3所示。

从这里，你可以浏览到要编辑的文件。要向上遍历一个目录级时，移动到双点条目并按下次回车键。要向下遍历一个目录级时，移动到该目录条目并按下次回车键。当你找到你要编辑的文件时，按下次回车键。Emacs会自动将它加载到新的缓冲区域。

你可以按C-x C-b扩展命令组合来列出工作缓冲区。Emacs编辑器会拆分编辑器窗口，在底部窗口显示一个缓冲区列表。除了主要的编辑缓冲区，Emacs还提供了两个缓冲区：

- 草稿区域，称为*scratch*；
- 消息区域，称为*Messages*。



```
rich@rich-desktop: ~
File Edit Options Buffers Tools Operate Mark Regexp Immediate Subdir Help
/home/rich:
total used in directory 544 available 56310068
drwxr-xr-x 37 rich rich 4096 2018-10-07 19:38 .
drwxr-xr-x 4 root root 4096 2018-09-23 19:01 ..
drwx----- 2 rich rich 4096 2018-09-22 12:18 .aptitude
-rw----- 1 rich rich 2831 2018-09-27 20:32 .bash_history
-rw-f--r-- 1 rich rich 220 2018-05-18 18:36 .bash_logout
-rw-f--r-- 1 rich rich 3103 2018-05-18 18:36 .bashrc
drwx----- 5 rich rich 4096 2018-10-07 19:27 .cache
drwx----- 3 rich rich 4096 2018-09-11 18:34 .compiz
drwxr-xr-x 10 rich rich 4096 2018-09-11 18:32 .config
drwx----- 3 rich rich 4096 2018-05-18 18:47 .dbus
drwxr-xr-x 2 rich rich 4096 2018-09-22 12:16 .debtags
drwxr-xr-x 2 rich rich 4096 2018-09-11 18:31 Desktop
-rw-f--r-- 1 rich rich 81 2018-10-07 19:27 .dmrc
drwxr-xr-x 2 rich rich 4096 2018-05-18 18:47 Documents
drwxr-xr-x 2 rich rich 4096 2018-09-22 13:46 Downloads
drwxr-xr-x 3 rich rich 4096 2018-10-07 19:38 .emacs.d
-rw----- 1 rich rich 16 2018-05-18 18:47 .esd_auth
-rw-f--r-- 1 rich rich 179 2018-05-18 18:36 examples.desktop
drwxr-xr-x 2 rich rich 4096 2018-06-26 20:09 fontconfig
FUDU@FUDU:~/ - Top 15 (Dired by name)
```

图9-3 Emacs查找文件模式浏览器

草稿区域允许你输入LISP编程命令以及留给你自己的笔记。消息区域则显示在操作中由Emacs生成的消息。如果在使用Emacs时出现了任何错误，它们会显示在消息区域中。

有两种方式来在窗口中切换到不同的缓冲区域。

- C-x o: 切换到缓冲区列表窗口。用方向键来移动你想要的缓冲区域并按下回车键。
- C-x b: 输入你要切换到的缓冲区域的名字。

当你选择切换到缓冲区列表窗口的选项时，Emacs会在新的窗口区域打开缓冲区域。Emacs编辑器允许你在单个会话中打开多个窗口。下一节将讨论如何在Emacs中管理多个窗口。

6. 在控制台模式的Emacs中使用窗口

控制台模式的Emacs编辑器是在图形化窗口出现的多年前开发的。然而，在当时它也是先进的，因为它可以支持在主Emacs窗口中打开多个编辑窗口。

你可以用下面两个命令之一来将Emacs编辑窗口拆分成多个窗口。

- C-x 2: 将窗口水平拆分成两个窗口。
- C-x 3: 将窗口竖向拆分成两个窗口。

要从一个窗口移动到另一个，可用C-x o命令。注意，当你创建一个新窗口时，Emacs会在新窗口中使用原始窗口的缓冲区域。一旦移动到了新窗口，你可以在新窗口中用C-x C-f命令来加载一个新文件，或者用命令之一来切换到一个不同的缓冲区域。

要关闭窗口，移动到该窗口并用C-x 0（数字零）命令，如果你想关掉除了你所在窗口之外的所有窗口，用C-x 1（数字一）命令。

9.2.2 在X Window中使用Emacs

如果你在X Window环境中使用Emacs（比如KDE或GNOME桌面上），它会以图形模式启动，如图9-4所示。



图9-4 Emacs图形化窗口

如果你已经在控制台模式下用过Emacs，你应该熟悉X Window模式。所有的键命令都以菜单栏条目的形式存在。Emacs菜单栏包括下列条目。

- ❑ **File:** 允许你在窗口中打开文件、创建新窗口、关闭窗口、保存缓冲区和打印缓冲区。
 - ❑ **Edit:** 允许你将选择的文本剪切和复制到剪贴板，将剪贴板的内容粘贴到光标当前所在位置，查找文本和替换文本。
 - ❑ **Options:** 提供对许多Emacs功能的设定，比如高亮显示、自动换行、光标类型和字体设置。
 - ❑ **Buffers:** 列出当前可用的缓冲区并允许你简便地在缓冲区域间切换。
 - ❑ **Tools:** 提供对Emacs中高级功能的访问，比如命令行界面访问、拼写检查、文件间文本比较（称为diff）、发送E-mail消息、日历以及计算器。
 - ❑ **Help:** 提供Emacs的在线手册，来获取针对特定的Emacs功能的帮助。
- 除了普通的Emacs图形化菜单栏条目外，通常在编辑器缓冲区还有一个单独的专属于文件类

型的条目。图9-4显示了打开一个C程序，所以Emacs提供了一个C菜单条目，允许在命令提示符针对C语法高亮显示、编译、运行以及调试代码进行高级设置。

图形化的Emacs窗口是老的控制台程序迁移到图形化世界的一个例子。现在许多Linux发行版提供了图形化桌面（甚至在不需要它们的服务器上），图形化编辑器也越来越司空见惯了。流行的两种Linux桌面环境（KDE和GNOME）都支持针对各自环境的图形化文本编辑器，本章后面将会介绍。

9.3 KDE系编辑器

如果你正在用采用KDE桌面的Linux发行版（参见第1章），那么在使用文本编辑器时你就可能有几种选择。KDE项目官方支持两个不同的文本编辑器。

- KWrite：**单屏幕文本编辑程序。
- Kate：**全功能、多窗口文本编辑程序。

这两个编辑器是包含许多高级功能的图形化文本编辑器。Kate编辑器提供了更高级的功能，以及标准文本编辑器中不常见的细致之处。本节将介绍每个编辑器，并演示一些可用来帮你编写shell脚本的功能。

9.3.1 KWrite编辑器

KDE环境的基本编辑器是KWrite。它提供了简单的文字处理类型的文本编辑功能，以及对代码语法的高亮显示和编辑的支持。默认的KWrite编辑窗口如图9-5所示。

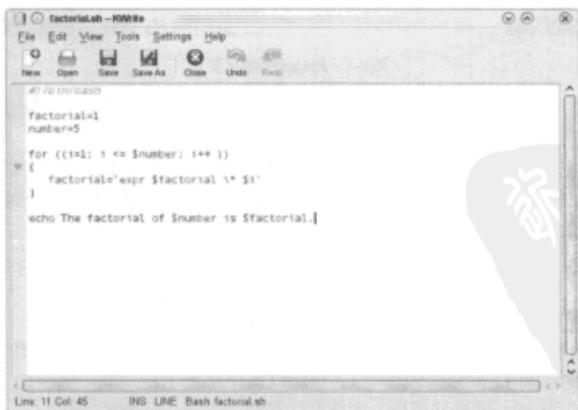


图9-5 编辑shell脚本程序时默认的KWrite窗口

尽管你可能没法在图9-5中分辨出来，但KWrite编辑器确实可以识别几种类型的编程语言，并采用彩色代码来标识常量、函数和注释。同样，注意在for循环处有个图标连接起开始和结束的花括号。这是处理大型应用时一个非常好的功能。

KWrite编辑窗口用鼠标和方向键提供了完整的剪切和粘贴功能。跟在文字处理器中一样，你可以高亮显示并剪切文本区域中任意位置的文本，并将它粘贴到其他任意位置。

要用KWrite编辑文件，你可以从桌面上的KDE菜单系统中选择KWrite（一些Linux发行版甚至为其创建了一个面板按钮）或从命令行下启动：

```
$ kwrite factorial.sh
```

kwrite命令有以下几个命令行参数可用来定制它如何启动。

- stdin**: 让KWrite从标准输入设备中而非文件中读取数据。
- encoding**: 为文件指定一个采用的字符编码类型。
- line**: 指定编辑器窗口中开始的文件的行号。
- column**: 指定编辑器窗口中开始的文件的列号。

KWrite编辑器在编辑器窗口的顶部提供了菜单栏和工具栏，允许你选择KWrite编辑器的功能以及修改其配置设置。

菜单栏含有下面的条目。

- File**: 加载、保存、打印以及导出文件中的文本。
- Edit**: 操作缓冲区中的文本。
- View**: 管理如何在编辑器窗口中显示文本。
- Bookmarks**: 处理返回文本中特定位置的指针（这个选项可能要在配置中启用）。
- Tools**: 包含操作文本的特定功能。
- Settings**: 配置编辑器处理文本的方式。
- Help**: 获取编辑器和命令的有关信息。

Edit菜单栏提供了你所有文本编辑需要的命令。不需要记住密码一般的键命令（顺便提一下，KWrite也支持），你只用在Edit菜单栏中选取条目即可，如表9-2所示。

表9-2 KWrite Edit菜单条目

条 目	描 述
Undo	取消最后一个动作或操作
Redo	取消最后一个撤销动作
Cut	删除选择的文本并将其放入剪贴板
Copy	将选择的文本复制到剪贴板
Copy as HTML	将选择的文本作为HTML码复制到剪贴板
Paste	在光标当前所在位置插入剪贴板的当前内容
Select All	选择编辑器中的所有文本
Deselect	取消选择当前选定的文本
Overwrite Mode	切换插入模式到改写模式；在改写模式中，文本会被新输入的文本覆盖，而不是仅插入新文本

(续)

条 目	描 述
Find	产生一个查找文本对话框，允许你定制文本查找
Find Next	在缓冲区中向前重复上一个查找操作
Find Previous	在缓冲区中向后重复上一个查找操作
Replace	产生一个替换文本对话框，允许你定制文本查找和替换
Find Selected	查找选定文本下一个出现的地方
Find Selected Backwards	查找选定文本上一个出现的地方
Go to Line	产生一个Go to（跳到）对话框，允许你输入一个行号。光标会移到指定行

Find功能有两种模式：普通模式，执行简单的文本搜索；高级查找和替换模式，可以进行必要的高级查找和替换。你可以用Find部分的绿箭头来切换这两种模式，如图9-6所示。

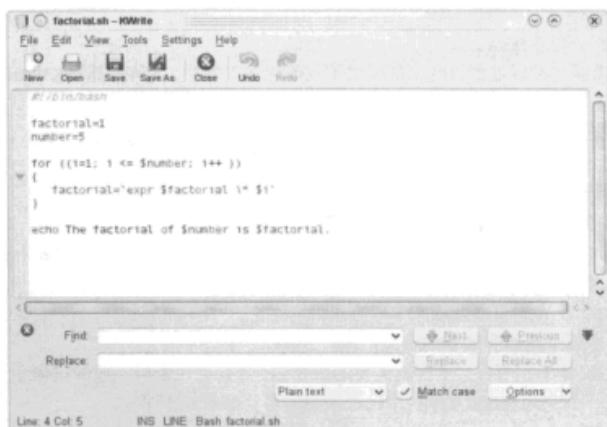


图9-6 KWrite Find部分

Find的高级模式不仅允许用词查找，还允许用正则表达式查找（参见第19章）。还有一些其他选项也可以用来定制查找，比如，是否在执行查找时忽略大小写，是全词匹配还是部分文本匹配。

Tools工具栏条目提供了一些在处理缓冲区文本时非常有用的功能。表9-3列出了KWrite中可用的工具。

表9-3 KWrite工具

工 具	描 述
Read Only Mode	锁定文本，这样在编辑器中就无法作任何修改
Encoding	设定文本采用的字符集编码

(续)

工具	描述
Spelling	从文本的开始进行拼写检查
Spelling (from cursor)	从光标当前所在位置开始进行拼写检查
Spellcheck Selection	仅在选定的文本区域中进行拼写检查
Indent	增加一级段落缩进
Unindent	减少一级段落缩进
Clean Indentation	将所有段落缩进重置
Align	强制当前行或选定行回到默认的缩进设置
Uppercase	将选定的文本或光标当前所在位置的字符设为大写
Lowercase	将选定的文本或光标当前所在位置的字符设为小写
Capitalize	大写选定文本的首字母或当前光标所在位置的单词的首字母
Join Lines	合并选定的行，或合并光标当前所在行及下一行
Word Wrap Document	使能文本自动换行。如果一行超过了编辑器窗口边界，该行在下一行继续

对一个简单的文本剪辑器来说有很多的工具。

Settings菜单包括了配置编辑器对话框，如图9-7所示。

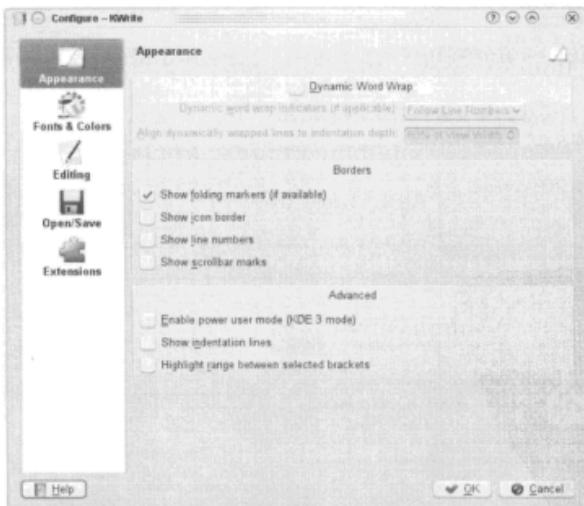


图9-7 KWrite配置编辑器对话框

配置编辑器对话框在左侧用图标来让你选择要配置的KWrite中的功能。当你选择一个图标时，对话框右侧显示了该功能的配置设置。

Appearance功能允许你设定几种功能，它们用来控制文本如何在文本编辑器窗口中显示。你可以在此使用自动换行、行号（对程序员非常有用）以及折叠标记。在Font & Colors功能中，你可以为编辑器定制完整的色彩方案，决定在某类程序代码中的文本上使用什么颜色。

9.3.2 Kate编辑器

Kate编辑器是KDE项目的旗舰编辑器。它采用和KWrite应用同样的文本编辑器（所以大部分功能相同），但它将很多其他功能集成到了单个程序上。

在启动Kate编辑器时，你首先要注意的是编辑器并未启动。相反，你会看到一个对话框，如图9-8所示。

Kate编辑器会按会话来处理文件。你可以在同一个会话中打开多个文件，也可以将多个会话保存。在启动Kate时，它会提供选项确定恢复到哪个会话。当关闭Kate会话时，它会记住你打开的文档并在下次启动Kate时显示它们。这允许你通过分开每个项目的工作空间来轻松管理多个项目的文件。

在选择一个会话后，你会看到Kate主编辑器窗口，如图9-9所示。

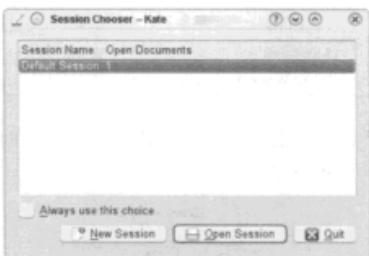


图9-8 Kate会话对话框

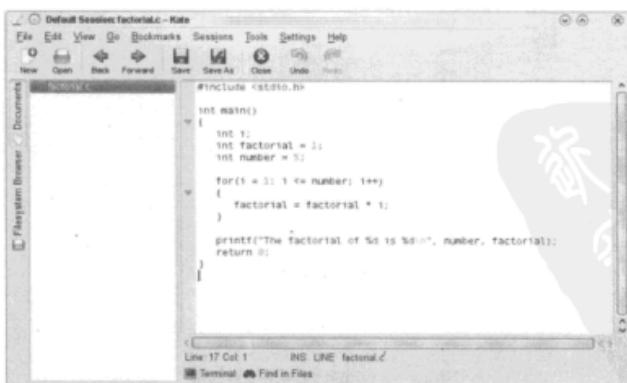


图9-9 Kate主编辑器窗口

左侧的框中显示了当前在这个会话中打开的文档。你可以通过点击文档名来在文档间切换。要编辑一个新文件，单击左侧的Filesystem Browser选项卡。左侧的框现在是一个完整的图形化文件系统浏览器，允许你在图形界面中浏览定位文件。

Kate编辑器的一个很好的功能是内建终端窗口，如图9-10所示。

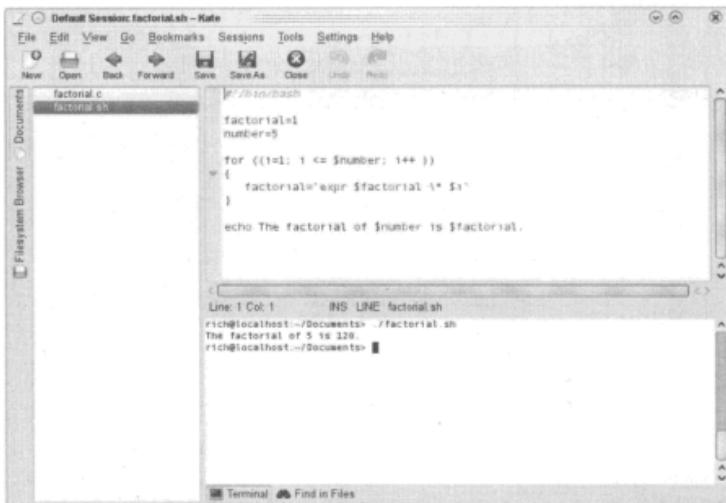


图9-10 Kate内建的终端窗口

9

文本编辑器窗口底部的Terminal选项卡启动了Kate内建的终端模拟器(采用KDE Konsole终端模拟器)。这个功能水平地划分了当前编辑窗口，创建了一个新窗口供Konsole运行。现在你无需离开编辑器就能输入命令行命令、启动程序或是检查系统设置。而要想关掉终端窗口，在命令行提示下输入exit即可。

如你从终端功能中看到的，Kate支持多窗口。Window菜单栏条目提供选项：

- 用当前会话创建新的Kate窗口；
- 竖直划分当前窗口来创建新窗口；
- 水平划分当前窗口来创建新窗口；
- 关闭当前窗口。

要设置Kate中的配置选项，在Settings菜单栏条目下选择Configure Kate，会出现配置对话框，如图9-11所示。

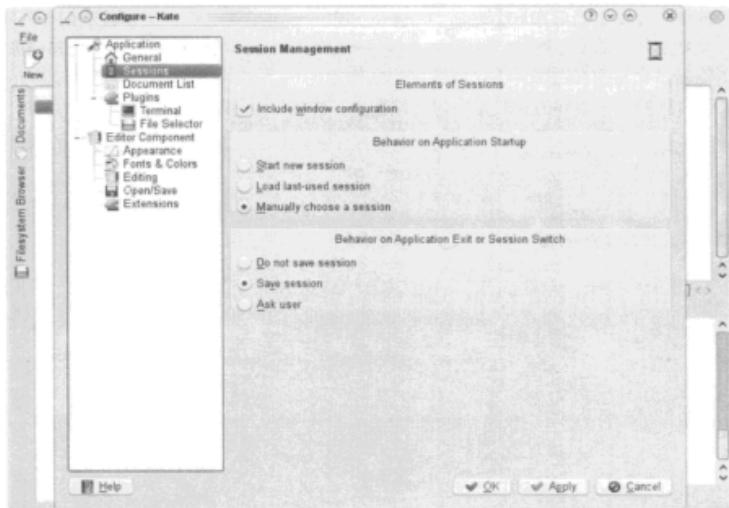


图9-11 Kate配置设置对话框

你会注意到Editor设置区域和KWrite的完全一样。这是因为这两个编辑器共享同一文本编辑器引擎。Application设置区域允许你配置Kate功能的设置，比如控制会话（如图9-11所示）、文档列表以及文件系统浏览器。Kate还支持外部插件应用，可以在这里激活。

9.4 GNOME 编辑器

如果你在采用GNOME作为桌面环境的Linux系统上工作，你也会用到一个图形化文本编辑器。gedit文本编辑器是一个基本的文本编辑器，有一些出于兴趣加进去的高级功能。本节将带你逐步了解gedit的功能并演示如何使用它来进行shell脚本编程。

9.4.1 启动gedit

大多数GNOME桌面环境将gedit放在Accessories面板条目中。如果你在那里找不到gedit，你可以从命令行下启动：

```
$ gedit factorial.sh myprog.c
```

当你启动gedit外带多个文件时，它会将所有的文件都加载到不同的缓冲区并在主编辑器窗口中按标签化的窗口来显示每个文件，如图9-12所示。



图9-12 gedit主编辑器窗口

gedit主编辑器窗口中左侧框显示了你当前在编辑的文档。右侧显示了含有缓冲区文本的标签窗口。如果你将鼠标在每个标签上晃动几下，会出现一个对话框，显示文件的全路径名、MIME类型以及它所采用的字符集编码。

9.4.2 基本的gedit功能

除了编辑器窗口，gedit采用菜单栏和工具栏来设置功能和配置设置。工具栏提供了到菜单栏条目的快捷方式。以下是可用的菜单栏条目。

- File:** 处理新文件、保存已有文件以及打印文件。
- Edit:** 在工作缓冲区域操作文本并设定编辑器偏好设置。
- View:** 设定显示在窗口中的编辑器功能以及设定文本的高亮显示模式。
- Search:** 在工作缓冲区域查找和替换文本。
- Tools:** 访问安装在gedit上的插件工具。
- Documents:** 管理缓冲区域打开的文件。
- Help:** 访问完整的gedit手册。

这里没什么特别的地方。File菜单提供了选项Open Location，允许你通过互联网世界中流行的标准统一资源标示符（Uniform Resource Identifier, URI）格式打开网络中的文件。这个格式标识了用来访问文件的协议（比如HTTP或FTP）、文件所在的服务器以及到服务器上访问该文件的完整路径。

Edit菜单含有标准的剪切、复制和粘贴功能，还有一个非常贴心的允许在文本中简单地用几

种不同格式输入时间日期的功能。Search菜单提供了标准的查找功能，它会生成一个供你输入要查找的文本的对话框，还能选择查找如何工作的方式（匹配大小写、匹配全字和查找方向）。它还提供了渐进式查找功能（工作在实时模式），用来在你输入单词的字母时查找文本。

9.4.3 设定偏好设置

Edit菜单包含了一个Preferences条目，它会产生gedit Preferences对话框，如图9-13所示。

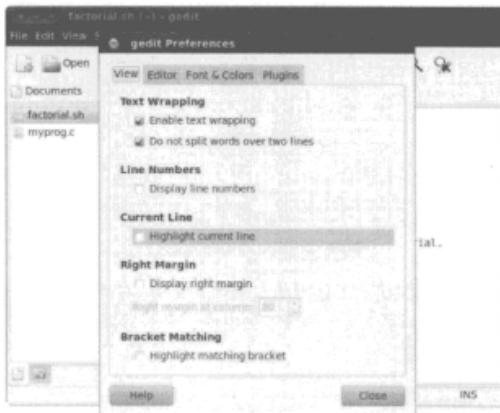


图9-13 gedit Preferences对话框

这里是定制gedit编辑器操作的地方。偏好设置对话框包含5个标签化区域来设定编辑器的功能和行为。

1. View

View选项卡提供了gedit如何在编辑器窗口中显示文本的选项。

- Text Wrapping:** 决定如何处理编辑器中的长行。Enable text wrapping选项会将长行自动换到编辑器中的下行。Do not split words over two lines选项阻止在长单词中自动插入连字符，进而阻止它们被分割到两行。
 - Line Numbers:** 在编辑器窗口的左边显示行号。
 - Current Line:** 高亮显示光标所在当前行，使得你能轻松找到光标位置。
 - Right Margin:** 可使右边留白并允许你设置在编辑器窗口中可有多少列。默认值是80列。
 - Bracket Matching:** 开启了的话，高亮显示代码中的括号对，允许轻松地匹配在if-then语句中、for和while循环中和其他使用括号的代码元素中的括号。
- 行号和括号匹配功能为程序员提供了文本编辑器中不常见的排除故障环境。

2. Editor

Editor选项卡提供了gedit编辑器如何处理标签和缩进以及如何保存文件的选项。

- Tab Stops:** 设定按下制表符时跳过的空白数，默认值是8。这个功能还包括一个复选框，允许在选定时插入空格来填充制表符跳过的空白。
- Automatic Indentation:** 开启了的话，让gedit在文本中自动为段落和代码元素（比如if-then语句和循环）缩进。
- File Saving:** 提供保存文件的两个功能：在编辑窗口中打开时是否创建文件的备份副本，以及是否在预设的间隔自动保存文件。

自动保存功能是保证你对文件的更改被规律性地保存从而避免系统崩溃或断电时发生灾难的有效途径。

3. Font & Colors

Font & Colors选项卡允许你配置两个条目。

- Font:** 允许你用默认的Monospace 10字体，或从对话框中选用定制的字体和字体大小。
- Color Scheme:** 允许你选用默认的用来指定文本、背景、选择的文本以及选定内容的色彩的方案，或为每个类型选用一个定制的颜色。

gedit默认的色彩通常和桌面选定的标准GNOME桌面主题匹配。这些色彩可以更改来匹配你为桌面选定的方案。

4. Plug-ins

Plugins选项卡提供了对gedit中用到的插件的控制。插件是独立的可以和gedit交互以提供额外功能的程序。Plugins选项卡如图9-14所示。

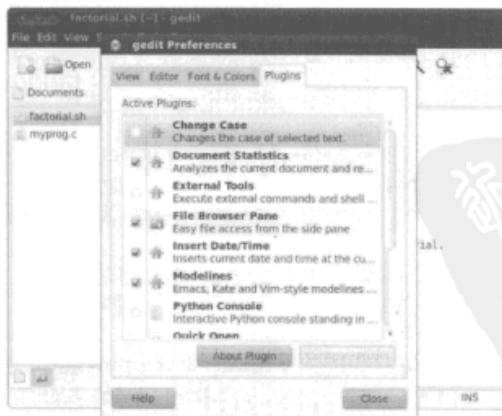


图9-14 gedit插件偏好设置标签

有些插件可在gedit中使用，但并非所有插件都默认安装。表9-4介绍了当前安装在gedit上的插件。

表9-4 gedit插件

插件名	描述
Change Case	改变选定文本的大小写
Document Statistics	报告单词、行、字符和非空字符的数量
External Tools	在编辑器中提供一个shell环境来执行命令和脚本
File Browser Pane	提供了一个简易的文件浏览器，让选择要编辑的文件简单些
Insert Date/Time	在光标当前位置以几种格式插入当前日期和时间
Modelines	在编辑器窗口底部显示类Emacs的消息行
Python Console	在编辑器窗口底部提供一个用来输入Python语言命令的交互式控制台
Quick Open	直接在gedit编辑窗口中打开文件
Snippets	允许你存储常用的文本段以方便在文本中取回使用
Sort	快速排序整个文件或选定文本
Spell Checker	为文本文件提供词典式拼写检查
Tag List	提供一个可轻松输入到文本中的常用字符串列表

使能了的插件会在它们名字边上的复选框里显示一个对号。一些插件，比如External Tool插件，也在选用它们后提供了额外的配置功能。它允许你设定快捷键来启动终端、指定gedit的输出显示在哪里以及启动shell会话。

遗憾的是，并非所有插件都安装在gedit菜单栏的同一个地方。一些插件会出现在Tools菜单栏中（比如Spell Checker和External Tools插件），而另一些则出现在Edit菜单栏（比如Change Case和Insert Date/Time插件）。

9.5 小结

在创建shell脚本时，你需要某种类型的文本编辑器。在Linux环境下，有一些流行的可用文本编辑器。Unix世界最流行的编辑器Vi已作为Vim编辑器被移植到了Linux中。Vim编辑器提供了简单的从控制台编辑文本的功能，采用了基本的全屏图形模式。Vim编辑器提供了很多高级编辑器功能，比如文本查找和替换。

另一个流行的Unix编辑器Emacs也已步入了Linux世界。Linux版本的Emacs有控制台和X Window图形两种模式，使其成为连接新旧世界的桥梁。Emacs编辑器提供了多个缓冲区域，允许你同时编辑多个文件。

KDE项目创建了两个在KDE桌面中使用的编辑器。KWrite编辑器是一个简单的提供了基本文本编辑功能的编辑器，还提供了一些高级功能，比如程序代码的高亮显示、行编号和代码折叠。Kate编辑器为程序员提供了更多的功能。Kate中一个伟大的功能是内建终端窗口。你可以在Kate编辑器中直接打开一个命令行接口会话，而不用打开一个单独的终端模拟器窗口。Kate编辑器还

允许你打开多个文件，为每个打开的文件提供不同的窗口。

GNOME项目也为程序员提供了一个简单的文本编辑器。gedit编辑器是一个基本的文本编辑器，同时还提供了一些高级功能，例如代码语法高亮显示和行编号，但它被设计成一个干练的编辑器。为了丰富gedit编辑器的功能，开发人员创建了可扩展gedit中可用功能的插件。当前插件包括一个拼写检查器、一个终端模拟器和一个文件浏览器。

本章结束了在Linux同命令行一起工作的背景章节。本书的下一部分将会深入介绍shell编程的世界。下章将从演示如何创建shell脚本文件和如何在Linux系统上运行开始。它还会向你展示shell脚本的基础知识，允许你通过创建将多个命令串起来放到可运行脚本中的简单程序。



Part 2

第二部分

shell 脚本编程基础

本部分内容

- 第 10 章 构建基本脚本
- 第 11 章 使用结构化命令
- 第 12 章 更多的结构化命令
- 第 13 章 处理用户输入
- 第 14 章 呈现数据
- 第 15 章 控制脚本

本章内容

- 构建基本脚本
- 使用多个命令
- 创建shell脚本文件

现 在我们已经介绍了Linux系统和命令行的基础知识，可以开始编程了。本章讨论编写shell脚本的基础知识。在开始编写自己的shell脚本大作前，你必须知道这些基本概念。

10.1 使用多个命令

到目前为止，你已经了解了如何使用shell的命令行界面提示符来输入命令和查看命令的结果了。shell脚本的关键在于输入多个命令并处理每个命令的结果，即使有可能将一个命令的结果传给另一个命令。shell允许你只用一步就将多个命令串连起来使用。

如果要两个命令一起运行，可在同一提示行输入它们，用分号分隔开：

```
$ date ; who
Mon Feb 21 15:36:09 EST 2011
Christine  tty2          2011-02-21 15:26
Samantha  tty3          2011-02-21 15:26
Timothy    tty1          2011-02-21 15:26
user       tty7          2011-02-19 14:03 (:0)
user       pts/0          2011-02-21 15:21 (:0.0)
```

\$

恭喜你，你刚刚已经写了一个脚本了。这个简单的脚本只用到了两个bash shell命令。date命令先运行，显示了当前日期和时间；后面紧跟着who命令的输入，显示当前是谁登录到了系统上。使用这种办法，你就能将任意多个命令串连在一起使用了，只要不超过最大命令行字符数255就行。

然而这种技术仅适用于小的脚本，它有一个致命的缺陷，即每次运行之前你都必须在命令提示符下输入整个命令。但不需要手动将这些命令都输入命令行中，你可以将命令合并成一个简单的文本文件。在需要运行这些命令时，你可以简单地在运行这个文本文件。

10.2 创建 shell 脚本文件

要将shell命令放到一个文本文件中，首先需要用一个文本编辑器（参见第9章）来创建一个文件，然后将命令输入到文件中。

在创建shell脚本文件时，必须在文件的第一行指定要使用的shell。其格式为：

```
#!/bin/bash
```

在通常的shell脚本的行里，井号（#）用作注释行。shell脚本中的注释行是不被shell执行的。然而，shell脚本文件的第一行是个特例，井号后接感叹号告诉shell用哪个shell来运行脚本（是的，你可以用bash shell来运用你的脚本程序，也可以用其他shell）。

在指定了shell之后，可在文件的每行输入命令，后加一个回车符。之前提到过，注释可用井号添加。例如：

```
#!/bin/bash
# This script displays the date and who's logged on
date
who
```

这就是所有脚本的内容了。如有需要，可用分号来在一一行输入你要用的两个命令。但在shell脚本中，你可以在不同行来列出命令。shell会按根据命令在文件中出现的顺序来处理命令。

还有，要注意另有一行也以井号（#）开头，并添加了一个注释。以井号开头的行都不会被shell处理（除了以井号开头的第一行）。在脚本中留下注释来说明脚本做了什么，这种方法非常好，所以两年后回过来再看这个脚本时，你还可以很容易记起来你做了什么。

将这个脚本保存在名为test1的文件中，就基本好了。在运行新脚本前，还要做其他一些事。

现在运行脚本，如下结果可能会叫你有点失望：

```
$ test1
bash: test1: command not found
$
```

你要跨过的第一个障碍是让bash shell能找到脚本文件。如第5章所述，shell会通过PATH环境变量来查找命令。快速地查看一下PATH环境变量就可以指出我们的问题了：

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin
:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin
$
```

PATH环境变量被设成只在一组目录中查找命令。要让shell找到test1脚本，我们只需采取下述做法之一：

- 将shell脚本文件所处的目录添加到PATH环境变量中；
- 在提示符中用绝对或相对文件路径来引用shell脚本文件。

提示 有些Linux发行版将\$HOME/bin目录添加进了PATH环境变量。它在每个用户的HOME目录下提供了一个存放脚本文件的地方，shell可以在那里查找要执行的命令。

在这个例子中，我们将用第二种方式来告诉shell脚本文件所处的确切位置。记住要引用当前目录下的文件，你要在shell中使用单点操作符：

```
$ ./test1
bash: ./test1: Permission denied
$
```

现在shell已经可以找到脚本文件了，但还有一个问题。shell表明你还没有执行文件的权限。快速查看文件权限就能找到问题所在：

```
$ ls -l test1
-rw-r--r-- 1 user user 73 Sep 24 19:56 test1
$
```

在创建test1文件时，umask的值决定了新文件的默认权限设置。由于umask变量设成了022（参见第6章），系统创建的文件只有文件属主才有读写权限。

下一步是通过chmod命令（参见第6章）赋予文件属主执行文件的权限：

```
$ chmod u+x test1
$ ./test1
Mon Feb 21 15:38:19 EST 2011
Christine tty2 2011-02-21 15:26
Samantha tty3 2011-02-21 15:26
Timothy tty1 2011-02-21 15:26
user tty7 2011-02-19 14:03 (:0)
user pts/0 2011-02-21 15:21 (:0.0) $
```

成功了！现在万事俱备，能够执行新的shell脚本文件了。

10.3 显示消息

许多shell命令会产生自己的输出，这些输出会显示在脚本所运行的控制台显示器上。然而许多情况下，你可能想要添加自己的文本消息来告诉脚本用户脚本正在做什么。你可以通过echo命令来做这个。echo命令能显示一个简单的文本字符串，如果你通过如下命令添加了字符串：

```
$ echo This is a test
This is a test
$
```

注意，默认情况下，你不需要使用引号将要显示的文本字符串圈起来。但有时在字符串中出现引号的话可能就比较麻烦：

```
$ echo Let's see if this'll work
Lets see if thisll work
$
```

echo命令可用单引号或双引号来将文本字符串圈起来。如果你在字符串中用到了它们，你需要在文本中使用其中一种引号，而用另外一种来将字符串圈起来：

```
$ echo "This is a test to see if you're paying attention"
This is a test to see if you're paying attention
```

```
$ echo 'Rich says "scripting is easy".'
Rich says "scripting is easy".
$
```

现在所有的引号都正确地在输出中显示了。

你可以将echo语句添加到shell脚本中任何需要显示额外信息的地方：

```
$ cat test1
#!/bin/bash
# This script displays the date and who's logged on
echo The time and date are:
date
echo "Let's see who's logged into the system: "
who
$
```

当运行这个脚本时，它会产生如下输出：

```
$ ./test1
The time and date are:
Mon Feb 21 15:41:13 EST 2011
Let's see who's logged into the system:
Christine tty2      2011-02-21 15:26
Samantha tty3     2011-02-21 15:26
Timothy tty1       2011-02-21 15:26
user    tty7       2011-02-19 14:03 (:0)
user    pts/0       2011-02-21 15:21 (:0.0)
$
```

很好，但如果你想在同一行显示一个文本字符串作为命令输出，应该怎么办呢？你可以用echo语句的-n参数。只要将第一个echo语句改成这样就行：

```
echo -n "The time and date are: "
```

你需要在字符串的两侧使用引号来保证在显示的字符串尾部有一个空格。命令输出将会紧接着字符串结束的地方开始。现在输出会是这个样子：

```
$ ./test1
The time and date are: Mon Feb 21 15:42:23 EST 2011
Let's see who's logged into the system:
Christine tty2      2011-02-21 15:26
Samantha tty3     2011-02-21 15:26
Timothy tty1       2011-02-21 15:26
user    tty7       2011-02-19 14:03 (:0)
user    pts/0       2011-02-21 15:21 (:0.0)
$
```

完美！echo命令是shell脚本中同用户交互的重要工具。你会在很多情况下用到它，尤其是当你想要显示脚本中变量的值时。我们下面继续了解这个。

10.4 使用变量

运行shell脚本中的单个命令很有用，但它有自身的限制。通常你可能会要用shell命令中的其

他数据来处理信息。这点可以通过变量来完成。变量允许你临时性地将信息存储在shell脚本中，以便和脚本中的其他命令一起使用。本节将介绍如何在shell脚本中使用变量。

10.4.1 环境变量

你已经亲自了解了一种Linux变量。第5章介绍了Linux系统中存在的环境变量。你也可以在脚本中访问这些值。

shell维护着一组环境变量，用来记录特定的系统信息。比如系统的名称，登录到系统上的用户的名称，用户的系统ID（也称为UID），用户的默认主目录以及shell查找程序的搜索路径。你可以用set命令来显示一份完整的活动的环境变量列表：

```
$ set
BASH=/bin/bash
...
HOME=/home/Samantha
HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS=$' \t\n'
IMSETTINGS_INTEGRATE_DESKTOP=yes
IMSETTINGS_MODULE=none
LANG=en_US.utf8
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=Samantha
...
```

你可以在环境变量名称之前加个美元符（\$）来在脚本中使用这些环境变量。下面的脚本中将会演示：

```
$ cat test2
#!/bin/bash
# display user information from the system.
echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
$
```

\$USER、\$UID和\$HOME环境变量用来显示已登录用户的有关信息。输出看起来应该是这样的：

```
$ chmod u+x test2
$ ./test2
User info for userid: Samantha
UID: 1001
HOME: /home/Samantha
$
```

注意，echo命令中的环境变量会在脚本运行时替换成当前值。还有在第一个字符串中我们可以将\$USER系统变量放置到双引号中，而shell依然能够知道我们的意图。但采用这种方法也有一个问题。看看下面这个例子会怎么样：

```
$ echo "The cost of the item is $15"
The cost of the item is 5
```

显然这不是我们想要的。只要脚本在引号中看到美元符，它就会以为你在引用一个变量。在这个例子中，脚本会尝试显示变量\$1（但并未定义），再显示数字5。要显示美元符，你必须在它前面放置一个反斜线：

```
$ echo "The cost of the item is \$15"
The cost of the item is $15
```

看起来好多了。反斜线允许shell脚本将美元符解释成为实际的美元符，而不是变量。下一节将会介绍如何在脚本中创建自己的变量。

说明 你可能还见过通过\${variable}形式引用的变量。变量名两侧额外的花括号通常用来帮助识别美元符后的变量名。

10.4.2 用户变量

除了环境变量，shell脚本还允许在脚本中定义和使用自己的变量。定义变量允许临时存储数据并在整个脚本中使用，从而使shell脚本看起来更像计算机程序。

用户变量可以是任何不超过20个字母、数字或下划线的文本字符串。用户变量区分大小写，所以变量Var1和变量var1是不同的。这个小规矩经常让脚本编程初学者感到头疼。

值通过等号赋给用户变量。在变量、等号和值之间不能出现空格（另一个困扰初学者的用法）。这里有一些给用户变量赋值的例子：

```
var1=10
var2=-57
var3=testing
var4="still more testing"
```

shell脚本会自动决定变量值的数据类型。在脚本的整个生命周期里，shell脚本中定义的变量会一直保持着它们的值，但在shell脚本完成时删除掉。

类似于系统变量，用户变量可通过美元符引用：

```
$ cat test3
#!/bin/bash
# testing variables
days=10
guest="Katie"
echo "$guest checked in $days days ago"
days=5
guest="Jessica"
echo "$guest checked in $days days ago"
$
```

运行脚本会有如下输出：

```
$ chmod u+x test3
$ ./test3
Katie checked in 10 days ago
Jessica checked in 5 days ago
$
```

变量每次被引用时，都会输出当前赋给它的值。重要的是记住，引用一个变量值时需要使用美元符，而引用变量来对其进行赋值时则不要使用美元符。通过一个例子你就能明白我的意思：

```
$ cat test4
#!/bin/bash
# assigning a variable value to another variable

value1=10
value2=$value1
echo The resulting value is $value2
$
```

当你在赋值语句中使用value1变量的值时，你仍然必须用美元符。这段代码产生如下输出：

```
$ chmod u+x test4
$ ./test4
The resulting value is 10
$
```

要是你忘了用美元符，使得value2的赋值行弄成这样：

```
value2=value1
```

你会得到如下输出：

```
$ ./test4
The resulting value is value1
$
```

没有美元符，shell会将变量名解释成普通的文本字符串，通常这并不是你想要的结果。

10.4.3 反引号

shell脚本中的最有用的特性之一就是反引号 (`)。注意，这并非是那个你所习惯的用来圈起字符串的普通单引号字符。由于在shell脚本之外很少用到它，你可能甚至都不知道在键盘什么地方能找到它。但你必须慢慢熟悉它，因为这是许多shell脚本中的重要组件。提示：在美式键盘上，它通常和波浪线 (~) 位于同一键位。

反引号允许你将shell命令的输出赋给变量。尽管这看起来并不那么重要，但它却是脚本编程中的一个主要构件。

你必须用反引号把整个命令行命令圈起来：

```
testing=`date`
```

shell会运行反引号中的命令，并将其输出赋给变量testing。这里有个通过普通的shell命令输出创建变量的例子：

```
$ cat test5
#!/bin/bash
# using the backtick character
testing=`date`
echo "The date and time are: " $testing
$
```

变量testing收到了date命令的输出，并在echo语句中用来显示它。运行这个shell脚本生成如下输出：

```
$ chmod u+x test5
$ ./test5
The date and time are: Mon Jan 31 20:23:25 EDT 2011
$
```

这个例子并没什么特别吸引人的地方（你也可以很轻松地将该命令放在echo语句中），但只要将命令的输出放到了变量里，你就能用它来干任何事情了。

下面这个例子很常见，它在脚本中通过反引号获得当前日期并用它来生成唯一文件名：

```
#!/bin/bash
# copy the /usr/bin directory listing to a log file
today=`date +%y%m%d`
ls /usr/bin -al > log.$today
```

today变量被赋予格式化后的date命令的输出。这是用来为日志文件名抓取日期信息常用的一种技术。`+%y%m%d`格式告诉date命令将日期显示为两位数的年、月、日的组合：

```
$ date +%y%m%d
110131
$
```

这个脚本将值赋给一个变量，之后再将其作为文件名的一部分。文件自身含有目录列表的重定向输出（将在10.5节中详细讨论）。运行脚本之后，你应该能在目录中看到一个新文件：

```
-rw-r--r-- 1 user user 769 Jan 31 10:15 log.110131
```

目录中出现的日志文件采用\$today变量的值作为文件名的一部分。日志文件的内容是/usr/bin目录的列表输出。如果脚本在后一天运行，日志文件名会是log.110201，因此每天创建一个新文件。

10.5 重定向输入和输出

有些时候你想要保存某个命令的输出而非在显示器上显示它。bash shell提供了一些不同的操作符来将某个命令的输出重定向到另一个位置（比如文件）。重定向可以通过将某个文件重定向到某个命令上来用在输入上，也可以用在输出上。本节介绍了如何在shell脚本中使用重定向。

10.5.1 输出重定向

重定向最基本的类型是将命令的输出发到一个文件中。bash shell采用大于号（>）来完成这项功能：

```
command > outputfile
```

之前显示器上出现的命令的输出会被保存到指定的输出文件中：

```
$ date > test6
$ ls -l test6
-rw-r--r-- 1 user      user        29 Feb 10 17:56 test6
$ cat test6
Thu Feb 10 17:56:58 EDT 2011
$
```

重定向操作符创建了一个文件test6（通过默认的umask设置）并将date命令的输出重定向到test6文件中。如果输出文件已经存在了，则这个重定向操作符会用新的文件数据覆盖已经存在的文件：

```
$ who > test6
$ cat test6
user    pts/0    Feb 10 17:55
$
```

现在test6文件的内容保存的是who命令的输出。

有时，取代覆盖文件的内容，你可能想要将命令的输出追加到已有文件上，比如你正在创建一个记录系统上某个操作的日志文件。在这种情况下，你可以用双大于号（>>）来追加数据：

```
$ date >> test6
$ cat test6
user    pts/0    Feb 10 17:55
Thu Feb 10 18:02:14 EDT 2011
$
```

test6文件仍然包含早些时候who命令处理的数据，加上现在新从date命令获得的输出。

10.5.2 输入重定向

输入重定向和输出重定向正好相反。输入重定向将文件的内容重定向到命令，而非将命令的输出重定向到文件。

输入重定向符号是小于号（<）：

```
command < inputfile
```

记住它的简易办法是在命令行上，命令总是在左侧，而重定向符号“指向”数据流动的方向。小于号说明数据正在从输入文件流向命令。

这里有个和wc命令一起使用输入重定向的例子：

```
$ wc < test6
      2      11      60
$
```

wc命令提供了对数据中文本的计数。默认情况下，它会输出3个值：

- 文本的行数；
- 文本的词数；
- 文本的字节数。

通过将文本文件重定向到wc命令，你可以得到对文件中的行、词和字节的快速计数。这个例子说明test6文件有2行、11个单词以及60字节。

还有另外一种输入重定向的方法，称为内联输入重定向（inline input redirection）。这种方法允许你在命令行而不是在文件指定输入重定向的数据。乍看一眼，这可能有点奇怪，但有些应用会用到这个过程（比如将在10.7节中提到的那些）。

内联输入重定向符号是双小于号(<<)。除了这个符号，你必须指定一个文本标记来划分要输入数据的开始和结尾。你可以用任何字符串的值来作为文本标记，但在数据的开始和结尾必须一致：

```
command << marker
data
marker
```

在命令行上使用内联输入重定向时，shell会用PS2环境变量中定义的次提示符（参见第5章）来提示输入数据。下面是使用它的情况：

```
$ wc << EOF
> test string 1
> test string 2
> test string 3
> EOF
      3      9     42
$
```

次提示符会一直提示输入更多数据，直到你输入了作为文本标记的那个字符串值。wc命令会对内联输入重定向提供的数据执行行、词和字节计数。

10.6 管道

有时你需要发送某个命令的输出作为另一个命令的输入。可以用重定向，只是有些笨拙：

```
$ rpm -qa > rpm.list
$ sort < rpm.list
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
```

```
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
```

...
rpm命令管理着通过Red Hat包管理系统（RPM）安装到系统上的软件包，比如上面列出的Fedora系统。在和`-qa`参数一起使用时，它会生成已安装包的列表，但并不会遵循某种特定的顺序。如果你在查找某个特定的包或一组包，可能会比较难在rpm命令的输出中找到它。

通过标准的输出重定向，rpm命令的输出被重定向到了文件`rpm.list`。命令完成后，`rpm.list`文件保存着我系统上安装的所有软件包列表。下一步，输入重定向用来将`rpm.list`文件的内容发送给`sort`命令来将包名按字母顺序排序。

这很有用，但仍然是一种比较烦琐的生成信息的方式。取代将命令的输出重定向到文件，你可以重定向输出到另一个命令。这个过程称为管道连接（piping）。

类似于反引号(`)，管道的符号在shell编程之外也很少用到。该符号由两个竖线构成，一个在另一个上面。然而，pipe符号印刷体通常看起来更像是单个竖线(|)。在美式键盘上，它通常和反斜线(\)位于同一个键。管道放在命令键，将一个命令的输出重定向到另一个上：

```
command1 | command2
```

不要以为管道链接会一个一个地运行。Linux系统实际上会同时运行这两个命令，在系统内部将它们连接起来。在第一个命令产生输出的同时，输出会被立即送给第二个命令。传输数据不会用到任何中间文件或缓冲区域。

现在，通过管道你可以轻松的将rpm命令的输出管道连接到sort命令来产生结果：

```
$ rpm -qa | sort
abrt-1.1.14-1.fc14.i686
abrt-addon-cpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
```

...
除非你眼神特别好，否则该命令的输出会一闪而过，你很有可能来不及看清就没了。由于管道功能是实时运行的，所以只要rpm命令一输出数据，sort命令就会立即对其进行排序。等到rpm命令输出完数据，sort命令就已经将数据排好序并显示在显示器上了。

可以在一条命令中使用任意多条管道。可以继续将命令的输出通过管道传给其他命令来简化操作。

在这个例子中，sort命令的输出会一闪而过，所以你可以用一条文本分页命令（例如less或more）来强行将输出按屏显示。

```
$ rpm -qa | sort | more
```

这行命令序列会先执行rpm命令，将它的输出通过管道传给sort命令，然后再将sort的输出通过管道传给more命令来显示，在显示完每屏信息后停下来。这样你就可以在继续处理前停下来，阅读显示器上显示的信息，如图10-1所示。

如果想要更别致点，你也可以搭配使用重定向和管道来将输出保存到文件中：

```
$ rpm -qa > rpm.list
$ more rpm.list
abrt-1.1.14-1.fc14.1686
abrt-addon-ccpp-1.1.14-1.fc14.1686
abrt-addon-kerneloops-1.1.14-1.fc14.1686
abrt-addon-python-1.1.14-1.fc14.1686
abrt-desktop-1.1.14-1.fc14.1686
abrt-gui-1.1.14-1.fc14.1686
abrt-libs-1.1.14-1.fc14.1686
abrt-plugin-bugzilla-1.1.14-1.fc14.1686
abrt-plugin-logger-1.1.14-1.fc14.1686
abrt-plugin-runapp-1.1.14-1.fc14.1686
acl-2.2.49-8.fc14.1686
...
...
```

如我们所期望的，rpm.list文件中的数据现在被排序了。



图10-1 通过管道将数据发送给more命令

到目前为止，管道最流行的用法之一是将命令产生的长输出结果通过管道传送给more命令。这对ls命令来说尤其普遍，如图10-2所示。

ls -l命令产生了一个目录中所有文件的长列表。对于有大量文件的目录来说，这个列表会相当长。通过将输出管道连接到more命令，你可以强制输出在每屏数据的末尾停下来。

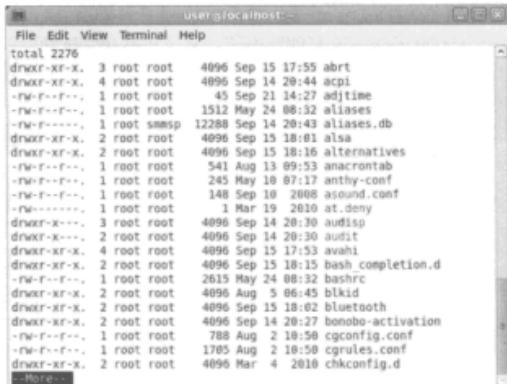


图10-2 和ls命令一起使用more命令

10.7 执行数学运算

另一个对任何编程语言都很重要的特性是操作数字的能力。遗憾的是，对shell脚本来说，这个过程会比较麻烦。在shell脚本中有两种途径来进行数学运算操作。

10.7.1 expr命令

最开始，Bourne shell提供了一个特别的命令用来处理数学表达式。expr命令允许在命令行上处理数学表达式，但是特别笨拙：

```
$ expr 1 + 5
6
```

expr命令能够识别一些不同的数学和字符串操作符，见表10-1。

表10-1 expr命令操作符

操作符	描述
ARG1 ARG2	如果没有参数是null或零值，返回ARG1；否则返回ARG2
ARG1 & ARG2	如果没有参数是null或零值，返回ARG1；否则返回0
ARG1 < ARG2	如果ARG1小于ARG2，返回1；否则返回0
ARG1 <= ARG2	如果ARG1小于或等于ARG2，返回1；否则返回0
ARG1 = ARG2	如果ARG1等于ARG2，返回1；否则返回0
ARG1 != ARG2	如果ARG1不等于ARG2，返回1；否则返回0
ARG1 >= ARG2	如果ARG1大于或等于ARG2，返回1；否则返回0

(续)

操作符	描述
ARG1 > ARG2	如果ARG1大于ARG2，返回1；否则返回0
ARG1 + ARG2	返回ARG1和ARG2的算术运算和
ARG1 - ARG2	返回ARG1和ARG2的算术运算差
ARG1 * ARG2	返回ARG1和ARG2的算术乘积
ARG1 / ARG2	返回ARG1被ARG2除的算术商
ARG1 % ARG2	返回ARG1被ARG2除的算术余数
STRING : REGEXP	如果REGEXP匹配到了STRING中的某个模式，返回该模式匹配
match STRING REGEXP	如果REGEXP匹配到了STRING中的某个模式，返回该模式匹配
substr STRING POS LENGTH	返回起始位置为POS（从1开始计数）、长度为LENGTH个字符的子字符串
index STRING CHARS	返回在STRING中找到CHARS字符串的位置；否则，返回0
length STRING	返回字符串STRING的数值长度
+ TOKEN	将TOKEN解释成字符串，即使是个关键字
(EXPRESSION)	返回EXPRESSION的值

尽管标准操作符在expr命令中工作得很好，但在脚本或命令行上使用它们时仍有问题出现。许多expr命令操作符在shell中有其他意思（比如星号）。在expr命令中使用它们会得到一些诡异的结果：

```
$ expr 5 * 2
expr: syntax error
$
```

要解决这个问题，在传给expr命令前，你需要使用shell的转义字符（反斜线）来识别容易被shell错误解释的任意字符：

```
$ expr 5 \* 2
10
$
```

现在麻烦才刚刚开始！在shell脚本中使用expr命令也同样麻烦：

```
$ cat test6
#!/bin/bash
# An example of using the expr command
var1=10
var2=20
var3='expr $var2 / $var1'
echo The result is $var3
```

要将一个数学算式的结果赋给一个变量，你需要使用反引号来获取expr命令的输出：

```
$ chmod u+x test6
$ ./test6
The result is 2
$
```

幸好, bash shell有一个针对处理数学运算符的改进, 你将会在下一节中看到。

10.7.2 使用方括号

bash shell为了保持跟Bourne shell的兼容而包含了expr命令, 但它同样也提供了一个执行数学表达式的更简单的方法。在bash中, 在将一个数学运算结果赋给某个变量时, 你可以用美元符和方括号 (`$[operation]`) 将数学表达式圈起来:

```
$ var1=$[1 + 5]
$ echo $var1
6
$ var2 = ${$var1 * 2}
$ echo $var2
12
$
```

用方括号来执行shell数学运算比用expr命令方便很多。这种技术在shell脚本中也能工作起来:

```
$ cat test7
#!/bin/bash
var1=100
var2=50
var3=45
var4=$[$var1 * ($var2 - $var3)]
echo The final result is $var4
$
```

运行这个脚本会得到如下输出:

```
$ chmod u+x test7
$ ./test7
The final result is 500
$
```

同样, 注意在使用方括号来计算公式时, 不用担心shell会误解乘号或其他符号。shell知道它不是通配符, 因为它在方括号内。

在bash shell脚本中进行算术运算会有一个主要的限制。看看下面这个例子:

```
$ cat test8
#!/bin/bash
var1=100
var2=45
var3=$[$var1 / $var2]
echo The final result is $var3
$
```

现在, 运行一下, 看看会发生什么:

```
$ chmod u+x test8
$ ./test8
The final result is 2
$
```

bash shell数学运算符只支持整数运算。如果你要进行任何实际的数学计算，这是一个巨大的限制。

说明 z shell (zsh) 提供了完整的浮点数算术操作。如果你需要在shell脚本中进行浮点数运算，你可以考虑看看z shell (将在第22章中讨论)。

10.7.3 浮点解决方案

有几种解决方案能够解决bash中数学运算的整数限制。最常见的解决方法是用内建的bash计算器，称作bc。

1. bc的基本用法

bash计算器其实是允许你在命令行输入浮点表达式、解释表达式、计算并返回结果的一种编程语言。bash计算器能够识别：

- 数字 (整数和浮点数);
- 变量 (简单变量和数组);
- 注释 (以井号开始的行或C语言中的/* */对);
- 表达式;
- 编程语句 (例如if-then语句);
- 函数。

你可以在shell提示符下通过bc命令访问bash计算器：

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
12 * 5.4
64.8
3.156 * (3 + 5)
25.248
quit
$
```

这个例子由输入表达式 $12 * 5.4$ 开始。bash计算器会返回答案。每个输入到计算器的后续表达式都会被执行，结果会显示出来。要退出bash计算器，你必须输入quit。

浮点运算是由一个内建的称为scale的变量控制的。你必须将这个值设置为结果里你想要的小数后的位数，否则你不会得到想要的结果的：

```
$ bc -q
3.44 / 5
0
scale=4
```

```
3.44 / 5
.6880
quit
$
```

scale变量的默认值是0。在scale值被设置前，bash计算器提供的答案没有小数点后的位置。在其值设置成4后，bash计算器显示的答案有四位小数。-q命令行参数会将bash计算器输出的很长的欢迎条屏蔽掉。

除了普通数字，bash计算器还能支持变量：

```
$ bc -q
var1=10
var1 * 4
40
var2 = var1 / 5
print var2
2
quit
$
```

一旦变量的值被定义了，你就可以在整个bash计算器会话中使用变量了。print语句允许你打印变量和数字。

2. 在脚本中使用bc

现在你可能想问bash计算器是如何在shell脚本中帮助你处理浮点运算的。你还记得老朋友反引号吗？是的，你可以用反引号来运行bc命令并将输出赋给一个变量。基本格式是这样的：

```
variable=`echo "options; expression" | bc`
```

第一部分options允许你来设置变量。如果你需要设置不止一个变量，可以用分号来分开它们。expression参数定义了通过bc执行的数学表达式。这里有个在脚本中使用它的例子：

```
$ cat test9
#!/bin/bash
var1=`echo " scale=4; 3.44 / 5" | bc`
echo The answer is $var1
$
```

这个例子将scale变量设置成了四位小数，并为表达式指定了特定的运算。运行这个脚本会产生如下输出：

```
$ chmod u+x test9
$ ./test9
The answer is .6880
$
```

太好了！现在你不会被限制只能用数字作为表达式值了。你也可以用shell脚本中定义好的变量：

```
$ cat test10
#!/bin/bash
var1=100
```

```
var2=45
var3=`echo "scale=4; $var1 / $var2" | bc`
echo The answer for this is $var3
$
```

脚本定义了两个变量，它们都可以用在表达式中发送给bc命令。记住用美元符来表示变量的值而不是变量自身。这个脚本的输出如下：

```
$ ./test10
The answer for this is 2.2222
$
```

当然，一旦一个值被赋给了变量，那个变量就能在其他运算中使用了：

```
$ cat test11
#!/bin/bash
var1=20
var2=3.14159
var3=`echo "scale=4; $var1 * $var1" | bc`
var4=`echo "scale=4; $var3 * $var2" | bc`
echo The final result is $var4
$
```

这个方法适用于较短的运算，但有时你会更多地和你自己的数字打交道。如果你有很多运算，在同一个命令行中列出多个表达式就会有点麻烦。

针对这个问题有个解决办法。bc命令能识别输入重定向，允许你将一个文件重定向到bc命令来处理。然而，这同样会叫人困惑，因为你必须将表达式存储到文件中。

最好的办法是使用内联输入重定向，允许你直接在控制台重定向数据。在shell脚本中，你可以将输出赋给一个变量：

```
variable=`bc << EOF
options
statements
expressions
EOF`
```

EOF文本字符串标识了内联重定向数据的开始和结尾。记住仍然需要反引号来将bc命令的输出赋给变量。

现在你可以将所有独立的bash计算器元素放到同一个脚本文件中的不同行。下面是在脚本中使用这种技术的例子：

```
$ cat test12
#!/bin/bash

var1=10.46
var2=43.67
var3=33.2
var4=71

var5=`bc << EOF
scale = 4
```

```
a1 = ($var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
echo The final answer for this mess is $var5
$
```

将每个选项和表达式放在脚本中不同的行中，可以让事情变得更清晰，也更容易阅读和跟进。EOF字符串标识了重定向给bc命令的数据的起始和结尾。当然，你需要用反引号来标识输出要赋给变量的命令。

你还会注意到这个例子中，你可以在bash计算器中赋值给变量。重要的是记住，在bash计算器中创建的变量只在bash计算其中有效，不能在shell脚本中使用。

10.8 退出脚本

到目前为止，在我们的示例脚本中，我们总是匆忙结尾。在我们运行完最后一条命令时，就结束了脚本。这里还有一个更好的结束事情的办法。

shell中运行的每个命令都使用退出状态码（exit status）来告诉shell它完成了处理。退出状态码是一个0~255之间的整数值，在命令结束运行时由命令传给shell。你可以捕获这个值并在脚本中使用。

10.8.1 查看退出状态码

Linux提供了\$?专属变量来保存上个执行的命令的退出状态码。你必须在你要查看的命令之后马上查看或使用\$?变量。它的值会变成shell中执行的最后一条命令的退出状态码：

```
$ date
Sat Jan 15 10:01:30 EDT 2011
$ echo $?
0
$
```

按照惯例，一个成功结束的命令的退出状态码是0。如果一个命令结束时有错误，退出状态码中就会有一个正数值：

```
$ asdfg
-bash: asdfg: command not found
$ echo $?
127
$
```

无效命令会返回一个退出状态码127。Linux错误退出状态码没有什么标准惯例。但有一些可用的参考，如表10-2所示。

表10-2 Linux退出状态码

状态码	描述
0	命令成功结束
1	通用未知错误
2	误用shell命令
126	命令不可执行
127	没找到命令
128	无效退出参数
128+x	Linux信号x的严重错误
130	命令通过Ctrl+C终止
255	退出状态码越界

退出状态码126表明用户没有执行命令的正确权限：

```
$ ./myprog.c
-bash: ./myprog.c: Permission denied
$ echo $?
126
$
```

另一个你会碰到的常见错误是给某个命令提供了无效参数：

```
$ date xt
date: invalid date `xt'
$ echo $?
1
$
```

这会生成通用的退出状态码1，表明在命令中发生了未知错误。

10.8.2 exit命令

默认情况下，shell脚本会以脚本中的最后一个命令的退出状态码退出：

```
$ ./test6
The result is 2
$ echo $?
0
$
```

你可以改变这个来返回你自己的退出状态码。exit命令允许你在脚本结束时指定一个退出状态码：

```
$ cat test13
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$((var1 + var2))
```

```
echo The answer is $var3
exit 5
$
```

当你查看脚本的退出码时，你会得到作为参数传给exit命令的值：

```
$ chmod u+x test13
$ ./test13
The answer is 40
$ echo $?
5
$
```

你也可以在exit命令的参数中使用变量：

```
$ cat test14
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$(( var1 + var2 ))
exit $var3
$
```

当你运行这个命令时，它会产生如下退出状态：

```
$ chmod u+x test14
$ ./test14
$ echo $?
40
$
```

然而，你要注意这个功能，退出状态码最大只能是255。看下面例子中会怎样：

```
$ cat test14b
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$(( var1 * var2 ))
echo The value is $var3
exit $var3
$
```

现在运行它，你会得到如下输出：

```
$ ./test14b
The value is 300
$ echo $?
44
$
```

退出状态码减到了0~255的区间。shell通过模运算得到这个结果。一个值的模就是被除后的余数。结果中的数是指定的数被256除的余数。在这个例子中的300（返回的值），余数是44，它

就成了最后的状态退出码。

在第11章中，你会了解如何用if-then语句来检查某个命令返回的错误状态，以便知道命令是否成功。

10.9 小结

bash shell脚本允许你将多个命令串起来放进脚本中。创建脚本的最基本的方式是将命令行中的多个命令通过分号分开来。shell会按顺序执行每个命令，在显示器上显示每个命令的输出。

你也可以创建一个shell脚本文件，将多个命令放进同一个文件以便shell按顺序执行。shell脚本文件必须定义运行脚本的shell。这个可以通过#!符号在脚本文件的第一行指定，后面跟上shell的全路径。

在shell脚本内，你可以通过在变量前使用美元符来引用环境变量。你也可以定义自己的变量以便在脚本内使用，并对其赋值甚至是通过反引号捕获的某个命令的输出。变量值可以在脚本中使用，在变量名前放置一个美元符。

bash shell允许你将命令的输入和输出从标准行为重定向。你可以通过大于号重定向任意命令在显示器屏幕上的输出到一个文件，后面紧跟着用来存储输出的文件的名称。你也可以通过双大于号将输出数据追加到已有文件。小于号用来重定向输入到命令。你可以将文件中的输入重定向到命令。

Linux管道命令（断条符号）允许你将命令的输出直接重定向到另一个命令的输入。Linux系统会同时运行这两条命令，将第一个命令的输出发送给第二个命令的输入，而不用任何重定向文件。

bash shell提供了许多方式来在shell脚本中执行数学操作。expr命令是一个进行整数运算的简便方法。在bash shell中，你也可以通过方括号包围的表达式来执行基本的数学运算，前加一个美元符。要执行浮点运算，你需要利用bc计算器命令，并从内联数据重定向输入、将输出存储到用户变量中。

最后，本章讨论了如何在shell脚本中使用退出状态码。shell中运行的每个命令都会产生一个退出状态码。退出状态码是一个0~255间的整数值，表明命令是否成功执行，以及如果不是，可能的原因是什么。退出状态码0表明命令成功执行了。你可以在shell脚本中用exit命令来在脚本完成时声明特定的退出状态码。

到目前为止，在你的脚本中，所有一切都会按有序的方式来处理，从一个命令到下一个命令。在下章中，你会了解如何用一些逻辑流控制来交替脚本中要执行的命令。



本章内容

- 使用if-then语句
- if-then-else语句
- 嵌套if
- test命令
- 复合条件测试
- if-then的高级特性
- case命令
- 管理用户账户

在 第10章中给出的那些shell脚本中，shell按照出现的次序来处理shell脚本中的每个单独命令。对于顺序操作来说这已经足够了，如果你只想所有的命令都能按正确的顺序执行。然而，并非所有程序都如此操作。

许多程序要求在shell脚本中的命令间有一些逻辑流控制。这意味着shell在某些环境下执行特定的命令，但在其他某些环境下执行另外一些命令。有一类命令会基于变量值或其他命令的结果等条件使脚本跳过或循环执行命令。这样的命令通常称为结构化命令（structured command）。

结构化命令允许你改变程序执行的顺序，在某些条件下执行一些命令而在其他条件下跳过另一些命令。在bash shell中有不少结构化命令，我们会逐个研究它们。在本章，我们来看一下if-then语句。

11.1 使用 if-then 语句

结构化命令中，最基本的类型就是if-then语句。if-then语句有如下格式：

```
if command  
then  
    commands  
fi
```

如果你在用其他编程语言的if-then语句，这种形式可能会让你有点困惑。在其他编程语言

中，if语句之后的对象是一个等式来测试是TRUE还是FALSE值。bash shell的if语句并非如此工作。bash shell的if语句会运行if行定义的那个命令。如果该命令的退出状态码（参见第10章）是0（该命令成功运行），位于then部分的命令就会被执行。如果该命令的退出状态码是其他什么值，那then部分的命令就不会被执行，bash shell会继续执行脚本中的下一个命令。

这里有个简单的例子来解释这个概念：

```
$ cat test1
#!/bin/bash
# testing the if statement
if date
then
    echo "it worked"
fi
```

这个脚本在if行采用了date命令。如果命令成功结束了，echo语句就会显示该文本字符串。当你在命令行运行该脚本时，你会得到如下结果：

```
$ ./test1
Sat Jan 23 14:09:24 EDT 2011
it worked
$
```

shell执行了if行的date命令。由于退出状态码是0，它就又执行了then部分的echo语句。

下面是另外一个例子：

```
$ cat test2
#!/bin/bash
# testing a bad command
if asdfg
then
    echo "it did not work"
fi
echo "we are outside of the if statement"
$
$ ./test2
./test2: line 3: asdfg: command not found
we are outside of the if statement
$
```

在这个例子中，我们在if语句行故意放了一个不能工作的命令。由于这个命令没法工作，它会输出一个非零的退出状态码，bash shell会跳过then部分的echo语句。还要注意，运行if语句中的那个命令所生成的错误消息依然会显示在脚本的输出中。有时你不想这种情况发生。第14章将会讨论如何避免这种情况。

在then部分，你可以用多个命令。你可以像脚本中其他地方一样列出多条命令。bash shell会将这部分命令当成一个块，在if语句行的那个命令返回退出状态码0时执行这块命令，在该命令返回非零退出状态码时跳过这块命令：

```
$ cat test3
#!/bin/bash
# testing multiple commands in the then section
```

```
testuser=rich
if grep $testuser /etc/passwd
then
    echo The bash files for user $testuser are:
    ls -a /home/$testuser/.b*
fi
```

if语句行使用了grep命令来在/etc/passwd文件中查找某个特定用户名是否在当前系统上使用。如果有用户使用了那个登录名，脚本会显示一些文本并列出该用户主（HOME）目录的bash文件：

```
$ ./test3
rich:x:500:500:Rich Blum:/home/rich:/bin/bash
The files for user rich are:
/home/rich/.bash_history  /home/rich/.bash_profile
/home/rich/.bash_logout  /home/rich/.bashrc
$
```

但是，如果你将testuser变量设置成一个系统上不存在的用户，则什么都不会显示：

```
$ ./test3
$
```

看起来也没什么新鲜的。可能我们在这里显示一些消息说明这个用户名在系统中未找到会稍微友好一些。是的，我们可以做到，用if-then语句的另外一个特性。

说明 你可能会在一些脚本中看到if-then语句的另一种格式：

```
if command; then
    commands
fi
```

在要执行的命令结尾加个分号，你就能在同一行使用then语句了，这样看起来就更像其他编程语言中的if-then语句是如何处理的了。

11.2 if-then-else语句

在if-then语句中，不管命令是否成功执行，你都只有一种选择。如果命令返回一个非零退出状态码，bash shell仅会继续执行脚本中的下一条命令。在这种情况下，如果能够执行另一组命令就好了。这正是if-then-else语句的作用。

if-then-else语句在语句中提供了另外一组命令：

```
if command
then
    commands
else
    commands
fi
```

当if语句中的命令返回退出状态码0时，then部分中的命令会被执行，跟普通的if-then语句一样。当if语句中的命令返回非零退出状态码时，bash shell会执行else部分中的命令。

现在你可以参考如下修改测试脚本：

```
$ cat test4
#!/bin/bash
# testing the else section
testuser=badtest
if grep $testuser /etc/passwd
then
    echo The files for user $testuser are:
    ls -a /home/$testuser/.b*
else
    echo "The user name $testuser does not exist on this system"
fi
$
$ ./test4
The user name badtest does not exist on this system
$
```

这样就更友好了。跟then部分一样，else部分可以包含多条命令。fi语句说明else部分结束了。

11.3 嵌套 if

有时你需要检查脚本代码中的多种条件。不用写多个分立的if-then语句，你可以用else部分的替代版本，称作elif。

elif会通过另一个if-then语句来延续else部分：

```
if command1
then
    commands
elif command2
then
    more commands
fi
```

elif语句行提供了另一个要测试的命令，类似于原始的if语句。如果elif后命令的退出状态码是0，则bash会执行第二个then语句部分的命令。

你可以继续将多个elif语句串起来，形成一个大的if-then-elif嵌套组合：

```
if command1
then
    command set 1
elif command2
then
    command set 2
elif command3
then
    command set 3
```

```
elif command4
then
    command set 4
fi
```

每块命令都会根据哪个命令会返回退出状态码0来执行。记住，**bash shell**会依次执行if语句，只有第一个返回退出状态码0的语句中的then部分会被执行。在11.7节，你将了解如何用case命令而不用嵌套多个if-then语句。

11.4 test 命令

到目前为止，你所了解到的if语句中的命令都是普通shell命令。你可能想问，if-then语句是否能测试跟命令的退出状态码无关的条件。

答案是不能。但是，在**bash shell**中有个好用的工具可以帮你通过if-then语句测试其他条件。

test命令提供了在if-then语句中测试不同条件的途径。如果test命令中列出的条件成立，test命令就会退出并返回退出状态码0，这样if-then语句就与其他编程语言中的if-then语句以类似的方式工作了。如果条件不成立，test命令就会退出并返回退出状态码1，这样if-then语句就会失效。

test命令的格式非常简单：

```
test condition
```

condition是test命令要测试的一系列参数和值。当用在if-then语句中时，test命令看起来是这样的：

```
if test condition
then
    commands
fi
```

bash shell提供了另一种在if-then语句中声明test命令的方法：

```
if [ condition ]
then
    commands
fi
```

方括号定义了test命令中用到的条件。注意，你必须在左括号右侧和右括号左侧各加一个空格，否则会报错。

test命令可以判断3类条件：

- 数值比较；
- 字符串比较；
- 文件比较。

后续章节将会介绍如何在if-then语句中使用这3类条件测试。

11.4.1 数值比较

使用test命令最常见的情形是对两个数值进行比较。表11-1列出了测试两个值时可用的条件参数。

表11-1 test命令的数值比较功能

比 较	描 述
n1-eq n2	检查n1是否与n2相等
n1-ge n2	检查n1是否大于或等于n2
n1-gt n2	检查n1是否大于n2
n1-le n2	检查n1是否小于或等于n2
n1-lt n2	检查n1是否小于n2
n1-ne n2	检查n1是否不等于n2

数值条件测试可以用在数字和变量上。这里有个例子：

```
$ cat test5
#!/bin/bash
# using numeric test comparisons
val1=10
val2=11

if [ $val1 -gt 5 ]
then
    echo "The test value $val1 is greater than 5"
fi

if [ $val1 -eq $val2 ]
then
    echo "The values are equal"
else
    echo "The values are different"
fi
```

第一个条件测试：

```
if [ $val1 -gt 5 ]
```

会测试变量val1的值是否大于5。第二个条件测试：

```
if [ $val1 -eq $val2 ]
```

会测试变量val1的值是否和变量val2的值相等。运行脚本并观察结果：

```
$ ./test5
The test value 10 is greater than 5
The values are different
$
```

两个数值条件测试都跟预期的一样执行了。

但是测试数值条件也有个限制。看下面的脚本：

```
$ cat test6
#!/bin/bash
# testing floating point numbers
vall=`echo "scale=4; 10 / 3 " | bc`
echo "The test value is $vall"
if [ $vall -gt 3 ]
then
    echo "The result is larger than 3"
fi
$
$ ./test6
The test value is 3.3333
./test6: line 5: [: 3.3333: integer expression expected
$
```

这个例子使用了bash计算器来生成一个浮点值并存储在变量vall中。下一步，它使用了test命令来判断这个值。显然这里出错了。

在第10章中，你已经了解了如何在bash shell中处理浮点数值；在脚本中仍然有一个问题。test命令无法处理vall变量中存储的浮点值。

记住，bash shell能处理的数仅有整数。当你使用bash计算器时，你可以让shell将浮点值作为字符串值存储进一个变量。如果你只是要通过echo语句来显示这个结果，那它可以很好地工作；但它无法在基于数字的函数中工作，例如我们的数值测试条件。尾行恰好说明了你不能在test命令中使用浮点值。

11.4.2 字符串比较

test命令还允许比较字符串值。比较对字符串比较烦琐，你马上就会看到。表11-2列出了可用来比较两个字符串值的函数。

表11-2 test命令的字符串比较功能

比 较	描 述
str1 = str2	检查str1是否和str2相同
str1 != str2	检查str1是否和str2不同
str1 < str2	检查str1是否比str2小
str1 > str2	检查str1是否比str2大
-n str1	检查str1的长度是否非0
-z str1	检查str1的长度是否为0

下面几节将会详细介绍不同的字符串比较功能。

1. 字符串相等性

字符串的相等和不等条件不言自明，很容易看出两个字符串值相同还是不同：

```
$ cat test7
#!/bin/bash
# testing string equality
testuser=rich

if [ $USER = $testuser ]
then
    echo "Welcome $testuser"
fi
$
$ ./test7
Welcome rich
$
```

不等字符串条件也允许你判断两个字符串是否具有相同的值：

```
$ cat test8
#!/bin/bash
# testing string equality
testuser=baduser

if [ $USER != $testuser ]
then
    echo "This is not $testuser"
else
    echo "Welcome $testuser"
fi
$
$ ./test8
This is not baduser
$
```

比较字符串的相等性时，test的比较会将所有的标点和大写也考虑在内。

2. 字符串顺序

要测试一个字符串是否比另一个字符串大就开始变得烦琐了。有两个问题会经常困扰正要开始使用test命令的大于小于特性的shell程序员：

- 大于小于符号必须转义，否则shell会把它们当做重定向符号而把字符串值当做文件名；
- 大于小于顺序和sort命令所采用的不同。

在编写脚本时，第一条可能会导致一个不易察觉的严重问题。这里有个shell脚本编程初学者时常遇到的例子：

```
$ cat badtest
#!/bin/bash
# mis-using string comparisons

val1=baseball
val2=hockey

if [ $val1 > $val2 ]
then
```

```

echo "$val1 is greater than $val2"
else
  echo "$val1 is less than $val2"
fi
$
$ ./badtest
baseball is greater than hockey
$ ls -l hockey
-rw-r--r-- 1 rich    rich          0 Sep 30 19:08 hockey
$
```

在这个脚本中只用了大于号，没有错误出现，但结果是错的。脚本把大于号解释成了输出重定向。因此，它创建了一个名为hockey的文件。由于重定向顺利完成了，test命令返回了退出状态码0，而if语句则以为所有命令都成功结束了。

要解决这个问题，你需要正确地转义大于号：

```

$ cat test9
#!/bin/bash
# mis-using string comparisons

val1=baseball
val2=hockey

if [ $val1 \> $val2 ]
then
  echo "$val1 is greater than $val2"
else
  echo "$val1 is less than $val2"
fi
$
$ ./test9
baseball is less than hockey
$
```

现在那个答案已经是我们希望通过字符串比较得到的了。

第二个问题更细微，除非你经常处理大小写字母，否则几乎遇不到。sort命令处理大写字母的方法刚好跟test命令的相反。让我们在脚本中测试一下这个特性：

```

$ cat test9b
#!/bin/bash
# testing string sort order
val1=Testing
val2=testing

if [ $val1 \> $val2 ]
then
  echo "$val1 is greater than $val2"
else
  echo "$val1 is less than $val2"
fi
$
$ ./test9b
Testing is less than testing
```

```
$ sort testfile
testing
Testing
$
```

在test命令中大写字母会被当成小于小写字母的。但当你将同样的字符串放进文件中并用sort命令排序时，小写字母会先出现。这是由各个命令使用的排序技术不同造成的。test命令会使用标准的ASCII顺序，根据每个字符的ASCII数值来决定排序顺序。sort命令使用的是系统的本地化语言设置中定义的排序顺序。对于英语，本地化设置指定了在排序顺序中小写字母出现在大写字母前。

警告 注意，test命令使用标准的数学比较符号来表示字符串比较，而用文本代码来表示数值比较。这个细微的特性被很多程序员理解反了。如果你为数值使用了数学运算符号，shell会将它们当成字符串，可能无法产生正确结果。

3. 字符串大小

-n和-z参数用来检查一个变量是否含有数据：

```
$ cat test10
#!/bin/bash
# testing string length
val1=testing
val2=''

if [ -n "$val1" ]
then
    echo "The string '$val1' is not empty"
else
    echo "The string '$val1' is empty"
fi

if [ -z "$val2" ]
then
    echo "The string '$val2' is empty"
else
    echo "The string '$val2' is not empty"
fi

if [ -z "$val3" ]
then
    echo "The string '$val3' is empty"
else
    echo "The string '$val3' is not empty"
fi
$

$ ./test10
The string 'testing' is not empty
The string '' is empty
The string '' is empty
$
```

这个例子创建了两个字符串变量。val1变量包含了一个字符串，val2变量则以空字符串创建。后续的比较如下：

```
if [ -n "$val1" ]
```

判断val1变量是否长度非零；而它的长度正好非零，所以then部分被执行了。

```
if [ -z "$val2" ]
```

判断val2变量是否长度为零；而它正好长度为零，所以then部分被执行了。

```
if [ -z "$val3" ]
```

判断val3变量是否长度为零。这个变量并未在shell脚本中定义过，所以它说明字符串长度仍然为零，尽管它并未被定义过。

警告 空的和未初始化的变量对shell脚本测试来说可能有灾难性的影响。如果你不是很确定一个变量的内容，最好在数值或字符串比较中使用它之前先通过-n或-z来测试一下变量是否含有值。

11.4.3 文件比较

最后一类测试比较可能是shell编程中最强大的也是最经常用到的比较。test命令允许你测试Linux文件系统上文件和目录的状态。表11-3列出了这些比较。

表11-3 test命令的文件比较功能

比 较	描 述
-d file	检查file是否存在并是一个目录
-e file	检查file是否存在
-f file	检查file是否存在并是一个文件
-r file	检查file是否存在并可读
-s file	检查file是否存在并非空
-w file	检查file是否存在并可写
-x file	检查file是否存在并可执行
-0 file	检查file是否存在并属当前用户所有
-G file	检查file是否存在并且默认组与当前用户相同
file1 -nt file2	检查file1是否比file2新
file1 -ot file2	检查file1是否比file2旧

这些条件使你能够在shell脚本中检查文件系统中的文件，并且经常用在要访问文件的脚本中。鉴于它们被如此广泛地使用，我们来逐个看看。

1. 检查目录

-d测试会检查指定的文件名是否在系统上以目录形式存在。当写文件到某个目录之前，或者是将文件放置到某个目录位置之前时，这会非常有用：

```
$ cat test11
#!/bin/bash
# look before you leap
if [ -d $HOME ]
then
    echo "Your HOME directory exists"
    cd $HOME
    ls -a
else
    echo "There is a problem with your HOME directory"
fi
$
$ ./test11
Your HOME directory exists
.. Documents .gvfs .pulse-cookie
.. Downloads .ICEauthority .recently-used.xbel
.aptitude .esd_auth .local
.sudo_as_admin_successful
.bash_history examples .desktop .mozilla Templates
.bash_logout .fontconfig Music test11
.bashrc .gconf .nautilus Videos
.cache .gconfd .openoffice.org
.xsession-errors
.config .gksu.lock Pictures
.xsession-errors.old
.dbus .gnome2 .profile
.Desktop .gnome2_private Public
.dmrc .gtk-bookmarks .pulse
$
```

示例代码中使用了-d测试条件来检查用户的\$HOME目录是否存在。如果它存在的话，它将继续使用cd命令来切到\$HOME目录并进行目录列表。

2. 检查对象是否存在

-e比较允许你在脚本中使用对象前检查文件或目录对象是否存在：

```
$ cat test12
#!/bin/bash
# checking if a directory exists
if [ -e $HOME ]
then
    echo "OK on the directory, now to check the file"
    # checking if a file exists
    if [ -e $HOME/testing ]
    then
        # the file exists, append data to it
        echo "Appending date to existing file"
        date >> $HOME/testing
    else
```

```

# the file does not exist, create a new file
echo "Creating new file"
date > $HOME/testing
fi
else
    echo "Sorry, you do not have a HOME directory"
fi
$
$ ./test12
OK on the directory, now to check the file
Creating new file
$ ./test12
OK on the directory, now to check the file
Appending date to existing file
$ 
```

第一个检查用-e比较来判断用户是否有\$HOME目录。如果有，下一个-e比较会检查并判断testing文件是否存在于\$HOME目录中。如果文件不存在，shell脚本会用单个大于号(输出重定向符号)来用date命令的输出创建一个新文件。你第二次运行这个shell脚本时，它会使用双大于号，这样它就能将date的输出追加到已经存在的文件后面。

3. 检查文件

-e比较适用于文件和目录。要确定指定的对象是个文件，你必须用-f比较：

```

$ cat test13
#!/bin/bash
# check if a file
if [ -e $HOME ]
then
    echo "The object exists, is it a file?"
    if [ -f $HOME ]
    then
        echo "Yes, it is a file!"
    else
        echo "No, it is not a file!"
        if [ -f $HOME/.bash_history ]
        then
            echo "But this is a file!"
        fi
    fi
else
    echo "Sorry, the object does not exist"
fi
$
$ ./test13
The object exists, is it a file?
No, it is not a file!
But this is a file! 
```

这一小段脚本作了很多检查。首先，它用-e比较测试\$HOME是否存在。如果存在，继续用-f来测试它是不是一个文件。如果它不是文件(当然不会是文件了)，我们用-f比较来测试

\$HOME/.bash_history是不是一个文件（当然，是个文件）。

4. 检查是否可读

在尝试从文件中读取数据之前，最好先测试一下是否能读文件。你可以通过-r比较测试：

```
$ cat test14
#!/bin/bash
# testing if you can read a file
pwfile=/etc/shadow

# first, test if the file exists, and is a file
if [ -f $pwfile ]
then
    # now test if you can read it
    if [ -r $pwfile ]
    then
        tail $pwfile
    else
        echo "Sorry, I am unable to read the $pwfile file"
    fi
else
    echo "Sorry, the file $file does not exist"
fi
$
$ ./test14
Sorry, I am unable to read the /etc/shadow file
$
```

/etc/shadow文件含有系统用户加密后的密码，所以它对系统上的普通用户是不可读的。-r比较判断出我没有这个文件的读权限，所以test命令失败了，而且bash shell执行了if-then语句的else部分。

5. 检查空文件

你应该用-s比较来检查文件是否为空，尤其是在你要删除文件时。当-s比较成功时要特别小心，它说明文件中有数据：

```
$ cat test15
#!/bin/bash
# testing if a file is empty
file=t15test
touch $file

if [ -s $file ]
then
    echo "The $file file exists and has data in it"
else
    echo "The $file exists and is empty"
fi
date > $file
if [ -s $file ]
then
    echo "The $file file has data in it"
else
```

```

echo "The $file is still empty"
fi
$
$ ./test15
The t15test exists and is empty
The t15test file has data in it
$
```

touch命令创建了这个文件但不会写入任何数据。在我们使用date命令并将其输出重定向到文件中后，-w比较说明文件中有数据。

6. 检查是否可写

-w比较会判断你是否对文件有可写权限：

```

$ cat test16
#!/bin/bash
# checking if a file is writeable

logfile=$HOME/t16test
touch $logfile
chmod u-w $logfile
now=`date +%Y%m%d-%H%M`

if [ -w $logfile ]
then
    echo "The program ran at: $now" > $logfile
    echo "The first attempt succeeded"
else
    echo "The first attempt failed"
fi

chmod u+w $logfile
if [ -w $logfile ]
then
    echo "The program ran at: $now" > $logfile
    echo "The second attempt succeeded"
else
    echo "The second attempt failed"
fi
$
$ ./test16
The first attempt failed
The second attempt succeeded
$ cat $HOME/t16test
The program ran at: 20110124-1602
```

这个脚本内容很多。首先，它在你的\$HOME目录定义了一个日志文件，将文件名存进变量logfile中，创建文件并通过chmod命令移除该用户对文件的写权限。下一步，它创建了变量now并通过date命令保存了一个时间戳。在这之后，它会检查你是否对新日志文件有写权限（你刚刚移除了写权限）。由于现在你没有了写权限，你应该会看到那条未成功消息。

之后，脚本又通过chmod命令重新赋予该用户写权限，并再次尝试写文件。这次写入成功了。

7. 检查是否可执行

-x比较是一个简便的判断你对某个特定文件是否有执行权限的方法。虽然可能大多数命令用不到它，但如果要在shell脚本中运行大量脚本，它可能很方便：

```
$ cat test17
#!/bin/bash
# testing file execution
if [ -x test16 ]
then
    echo "You can run the script: "
    ./test16
else
    echo "Sorry, you are unable to execute the script"
fi
$ 
$ ./test17
You can run the script:
The first attempt failed
The second attempt succeeded
$
$ chmod u-x test16
$
$ ./test17
Sorry, you are unable to execute the script
$
```

这段示例shell脚本用-x比较来测试是否有权限执行test16脚本。如果有权限，它会运行这个脚本（注意，即使是在shell脚本中，你必须用正确的路径来执行不在你的\$PATH路径中的脚本）。在首次成功运行test16脚本后，更改文件的权限并再试。这次，-x比较失败了，因为你没有test16脚本的执行权限。

8. 检查所属关系

-0比较允许你轻松地测试你是否是文件的属主：

```
$ cat test18
#!/bin/bash
# check file ownership

if [ -O /etc/passwd ]
then
    echo "You are the owner of the /etc/passwd file"
else
    echo "Sorry, you are not the owner of the /etc/passwd file"
fi
$ 
$ ./test18
Sorry, you are not the owner of the /etc/passwd file
$
$ su
Password:
$
# ./test18
You are the owner of the /etc/passwd file
#
```

这段脚本用-G比较来测试运行该脚本的用户是否是/etc/passwd文件的属主。第一次，这个脚本运行在普通用户账户下，所以测试失败了。第二次，我们用su命令切换到root用户，测试通过了。

9. 检查默认属组关系

-G比较会检查文件的默认组，如果它匹配了用户的默认组，那就通过了。由于-G比较只会检查默认组而非用户所属的所有组，这会叫人有点困惑。这里有个例子：

```
$ cat test19
#!/bin/bash# check file group test

if [ -G $HOME/testing ]
then
    echo "You are in the same group as the file"
else
    echo "The file is not owned by your group"
fi
$
$ ls -l $HOME/testing
-rw-rw-r-- 1 rich rich 58 2011-01-30 15:51 /home/rich/testing
$
$ ./test19
You are in the same group as the file
$
$ chgrp sharing $HOME/testing
$
$ ./test19
The file is not owned by your group
$
```

第一次运行脚本时，\$HOME/testing文件是在rich组的，所以-G比较通过了。下一次，组被改成了sharing组，用户也是其中的一员，但-G比较失败了，因为它只比较默认组，不会去比较其他额外的组。

10. 检查文件日期

最后一组方法用来进行两个文件创建日期相关的比较。这在编写安装软件的脚本时非常有用。有时你不会想安装一个比系统上已安装文件还要早的文件。

-nt比较会判定某个文件是否比另一个文件更新。如果文件更新，那它会有一个比较近的文件创建日期。-ot比较会判定某个文件是否比另一个文件更老。如果文件更老，它会有一个更早的创建日期：

```
$ cat test20
#!/bin/bash
# testing file dates

if [ ./test19 -nt ./test18 ]
then
    echo "The test19 file is newer than test18"
```

```

else
    echo "The test18 file is newer than test19"
fi
if [ ./test17 -ot ./test19 ]
then
    echo "The test17 file is older than the test19 file"
fi
$
$ ./test20
The test19 file is newer than test18
The test17 file is older than the test19 file
$
$ ls -l test17 test18 test19
-rwxr--r-- 1 rich rich 167 2011-01-30 16:31 test17
-rwxr--r-- 1 rich rich 185 2011-01-30 17:46 test18
-rwxr--r-- 1 rich rich 167 2011-01-30 17:50 test19
$
```

比较中用到的文件路径是相对于你运行该脚本的目录来说的。如果你要检查的文件是可以被移来移去的，这可能会造成一些问题。另一个问题是这些比较中没有哪个会先检查文件是否存在。试试这个测试：

```

$ cat test21
#!/bin/bash
# testing file dates

if [ ./badfile1 -nt ./badfile2 ]
then
    echo "The badfile1 file is newer than badfile2"
else
    echo "The badfile2 file is newer than badfile1"
fi
$
$ ./test21
The badfile2 file is newer than badfile1
$
```

这个小例子演示了如果文件不存在，-nt比较会返回一个无效的条件。在你尝试在-nt或-ot比较中使用文件之前，必须先确认文件存在。

11.5 复合条件测试

if-then语句允许你使用布尔逻辑来组合测试。有两种布尔运算符可用：

- [condition1] && [condition2];
- [condition1] || [condition2]。

第一个布尔运算使用AND布尔运算符来组合两个条件。要让then部分的命令执行，两个条件都必须满足。

第二个布尔运算使用OR布尔运算符来组合两个条件。如果任何一个条件最后都能得到一个真值, then部分的命令就会执行。

```
$ cat test22
#!/bin/bash
# testing compound comparisons

if [ -d $HOME ] && [ -w $HOME/testing ]
then
    echo "The file exists and you can write to it"
else
    echo "I cannot write to the file"
fi
$
$ ./test22
I cannot write to the file
$ touch $HOME/testing
$
$ ./test22
The file exists and you can write to it
$
```

使用AND布尔运算符时, 两个比较都必须满足。第一个比较会检查用户的\$HOME目录是否存在。第二个比较会检查在用户的\$HOME目录是否有叫testing的文件, 以及用户是否有该文件的写权限。如果两个比较中的任意一个失败了, if语句就会失败, shell就会执行else部分的命令。如果两个比较都通过了, if语句就通过了, shell会执行then部分的命令。

11.6 if-then 的高级特性

bash shell有两项较新的扩展, 提供了可在if-then语句中使用的高级特性:

- 用于数学表达式的双尖括号;
- 用于高级字符串处理功能的双方括号。

后面几节将会进一步描述这些特性中的每一项。

11.6.1 使用双尖括号

双尖括号命令允许你将高级数学表达式放入比较中。test命令只允许在比较中进行简单的算术操作。双尖括号命令提供了更多的为用过其他编程语言的程序员所熟悉的数学符号。双尖括号命令的格式如下:

```
(( expression ))
```

术语expression可以是任意的数学赋值或比较表达式。除了test命令使用的标准数学运算符, 表11-4列出了双尖括号命令中会用到的其他运算符。

表11-4 双尖括号命令符号

符 号	描 述
val++	后增
val--	后减
++val	先增
--val	先减
!	逻辑求反
-	位求反
**	幂运算
<<	左位移
>>	右位移
&	位布尔和
	位布尔或
&&	逻辑和
	逻辑或

你可以在if语句中用双尖括号命令，也可以在脚本中的普通命令里使用来赋值：

```
$ cat test23
#!/bin/bash
# using double parenthesis

val1=10

if (( $val1 ** 2 > 90 ))
then
    (( val2 = $val1 ** 2 ))
    echo "The square of $val1 is $val2"
fi
$
$ ./test23
The square of 10 is 100
$
```

注意，你不需要将双尖括号中表达式里的大于号转义。这是双尖括号命令提供的另一个高级特性。

11.6.2 使用双方括号

双方括号命令提供了针对字符串比较的高级特性。双方括号命令的格式如下：

[[expression]]

双方括号里的expression使用了test命令中采用的标准字符串进行比较。但它提供了test命令未提供的另一个特性——模式匹配 (pattern matching)。

在模式匹配中，你可以定义一个正则表达式（将在第19章中详细讨论）来匹配字符串值：

```
$ cat test24
#!/bin/bash
# using pattern matching

if [[ $USER == r* ]]
then
    echo "Hello $USER"
else
    echo "Sorry, I do not know you"
fi
$
$ ./test24
Hello rich
$
```

双方括号命令匹配了\$USER环境变量来看它是否以字母r开头。如果是的话，比较就会通过，shell会执行then部分的命令。

11.7 case 命令

你会经常发现自己在尝试计算一个变量的值或在一组可能的值中寻找特定值。在这种情形下，你最终必须写出很长的if-then-else语句，像这样：

```
$ cat test25
#!/bin/bash
# looking for a possible value

if [ $USER = "rich" ]
then
    echo "Welcome $USER"
    echo "Please enjoy your visit"
elif [ $USER = barbara ]
then
    echo "Welcome $USER"
    echo "Please enjoy your visit"
elif [ $USER = testing ]
then
    echo "Special testing account"
elif [ $USER = jessica ]
then
    echo "Do not forget to logout when you're done"
else
    echo "Sorry, you are not allowed here"
fi
$
$ ./test25
Welcome rich
Please enjoy your visit
$
```

elif语句继续进行if-then检查，为单个比较变量寻找特定值。

你可以使用case命令，而不用写出那么多elif语句来不断检查相同变量值。case命令会检查单个变量列表格式的多个值：

```
case variable in
pattern1 | pattern2) commands1;;
pattern3) commands2;;
*) default commands;;
esac
```

case命令会将指定的变量同不同模式进行比较。如果变量和模式是匹配的，那么shell会执行为该模式指定的命令。你可以通过竖线操作符来分割模式，在一行列出多个模式。星号会捕获所有跟所有列出的模式都不匹配的值。这里有个将if-then-else程序转换成用case命令的例子：

```
$ cat test26
#!/bin/bash
# using the case command

case $USER in
rich | barbara)
    echo "Welcome, $USER"
    echo "Please enjoy your visit";;
testing)
    echo "Special testing account";;
jessica)
    echo "Do not forget to log off when you're done";;
*)
    echo "Sorry, you are not allowed here";;
esac
$
$ ./test26
Welcome, rich
Please enjoy your visit
$
```

case命令提供了一个更清晰的方法来为变量每个可能的值指定不同的选项。

11.8 小结

结构化命令允许你改变shell脚本的正常执行流。最基本的结构化命令是if-then语句。该语句允许你执行一个命令并根据该命令的输出来执行其他命令。

你也可以扩展if-then语句来在指定命令失败时让bash shell执行另一组命令。仅在测试命令返回非零退出状态码时，if-then-else语句才允许执行命令。

你也可以将if-then-else语句通过elif语句连接起来。elif等同于使用else if语句，会在测试命令失败时提供额外的检查。

在很多脚本中，你可能希望测试一种条件而不是一个命令，比如数值、字符串内容或者文件或目录的状态。test命令为你提供了测试所有这些条件的简单方法。如果条件计算出的是真值情况，test命令会为if-then语句产生退出状态码0。如果条件计算出一个假值情况，test命令会为

if-then语句产生一个非零的退出状态码。

方括号是与test命令同义的特殊bash命令。你可以在if-then语句中将测试条件放在方括号中来测试数值、字符串和文件条件。

双尖括号允许你使用额外的操作符来进行高级数学运算，双方括号命令允许你进行高级字符串模式匹配运算。

最后，本章讨论了case命令。它是执行多个if-then-else命令的简便方式，会参照一个值列表来检查单个变量的值。

下一章会继续讨论结构化命令，介绍shell的循环命令。for和while命令允许你创建循环来在给定时间内重复执行一些命令。

本章内容

- 用for语句来循环
- 用until语句来迭代
- 使用while语句
- 循环
- 重定向循环的输出

在

上一章里，你了解了如何通过检查命令的输出和变量的值来改变shell脚本程序的流程。在本章中，我们会继续介绍能够控制shell脚本流程的结构化命令。你会了解如何重复一些过程和命令，也就是循环执行一组命令直至达到了某个特定条件。本章将会讨论和演示bash shell的循环命令for、while和until。

12.1 for 命令

重复执行一系列命令在编程中很常见。通常你需要重复一组命令直至达到某个特定条件，比如处理某个目录下的所有文件、系统上的所有用户或是某个文本文件中的所有行。

bash shell提供了for命令，允许你创建一个遍历一系列值的循环。每个迭代都通过一个该系列中的值执行一组预定义的命令。下面是bash shell中for命令的基本格式：

```
for var in list  
do  
    commands  
done
```

在list参数中，你提供了迭代中要用的一系列值。你可以通过几种不同的途径来指定列表中的值。

在每个迭代中，变量var会包含列表中的当前值。第一个迭代会使用列表中的第一个值，第二个迭代使用第二个值，以此类推，直到列表中的所有值都过一遍。

在do和done语句之间输入的命令可以是一条或多条标准的bash shell命令。在这些命令中，\$var变量包含着这次迭代对应的当前那个列表中的值。

说明 只要你愿意，也可以将do语句和for语句放在同一行，但必须用分号将其同列表中的值分开：for var in list; do。

我们前面提过有几种不同的方式来指定列表中的值。下面几节将会介绍各种方式。

12.1.1 读取列表中的值

for命令最基本的用法就是遍历for命令自身中定义的一系列值：

```
$ cat test1
#!/bin/bash
# basic for command

for test in Alabama Alaska Arizona Arkansas California Colorado
do
    echo The next state is $test
done
$ ./test1
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
$
```

每次for命令遍历提供的值列表时，它会将列表中的下个值赋给\$test变量。\$test变量像for命令语句中的任何其他脚本变量一样使用。在最后一次迭代后，\$test变量的值会在shell脚本的剩余部分一直保持有效。它会一直保持最后一次迭代的值（除非你修改了它）：

```
$ cat test1b
#!/bin/bash
# testing the for variable after the looping

for test in Alabama Alaska Arizona Arkansas California Colorado
do
    echo "The next state is $test"
done
echo "The last state we visited was $test"
test=Connecticut
echo "Wait, now we're visiting $test"
$ ./test1b
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
The last state we visited was Colorado
Wait, now we're visiting Connecticut
$
```

\$test变量保持了它的值，也允许我们修改它的值并在for命令循环之外跟其他变量一样使用。

12.1.2 读取列表中的复杂值

事情并不像for循环看上去那么简单。有时你会遇到难处理的数据。下面是个给shell脚本程序员带来麻烦的经典例子：

```
$ cat badtest1
#!/bin/bash
# another example of how not to use the for command

for test in I don't know if this'll work
do
    echo "word:$test"
done
$ ./badtest1
word:I
word:don't know if this'll
word:work
$
```

真麻烦。shell看到了列表值中的单引号并尝试使用它们来定义一个单独的数据值，这个过程一团混乱。

有两种办法解决这个问题：

- 使用转义字符（反斜线）来将单引号转义；
- 使用双引号来定义用到单引号的值。

每个解决方法都并非那么神奇，但每个都能解决这个问题：

```
$ cat test2
#!/bin/bash
# another example of how not to use the for command

for test in I don't know if "this'll" work
do
    echo "word:$test"
done
$ ./test2
word:I
word:don't
word:know
word:if
word:this'll
word:work
$
```

在第一个有问题的值上，添加了反斜线字符来转义don't值中的单引号。在第二个有问题的值上，将this'll值用双引号圈起来。两种方法都能正常工作，辨别出这个值。

你可能遇到的另一个问题是多个词的值。记住，for循环假定每个值都是用空格分割的。如果有包含空格的数据值，你可能会遇到另一个问题：

```
$ cat badtest2
#!/bin/bash
# another example of how not to use the for command

for test in Nevada New Hampshire New Mexico New York North Carolina
do
    echo "Now going to $test"
done
$ ./badtest1
Now going to Nevada
Now going to New
Now going to Hampshire
Now going to New
Now going to Mexico
Now going to New
Now going to York
Now going to North
Now going to Carolina
$
```

这不是我们想要的结果。for命令用空格来划分列表中的每个值。如果在单独的数据值中有空格，那么你必须用双引号来将这些值圈起来：

```
$ cat test3
#!/bin/bash
# an example of how to properly define values

for test in Nevada "New Hampshire" "New Mexico" "New York"
do
    echo "Now going to $test"
done
$ ./test3
Now going to Nevada
Now going to New Hampshire
Now going to New Mexico
Now going to New York
$
```

现在for命令可以正确的区分不同值了。还有，注意当你在某个值两边使用双引号时，shell不会将双引号当成值的一部分。

12.1.3 从变量读取列表

通常shell脚本遇到的情况是，你将一系列值都集中存储在了一个变量中，然后需要遍历整个列表。你也可以通过for命令完成这个：

```
$ cat test4
#!/bin/bash
# using a variable to hold the list

list="Alabama Alaska Arizona Arkansas Colorado"
list=$list" Connecticut"
```

```

for state in $list
do
    echo "Have you ever visited $state?"
done
$ ./test4
Have you ever visited Alabama?
Have you ever visited Alaska?
Have you ever visited Arizona?
Have you ever visited Arkansas?
Have you ever visited Colorado?
Have you ever visited Connecticut?
$ 
```

\$list变量包含了给迭代用的标准文本值列表。注意，代码还是用了另一个赋值语句来向\$list变量包含的已有列表添加了一个值。这是向变量中存储的已有文本字符串尾部添加文本的一个常用方法。

12.1.4 从命令读取值

生成列表中要用的值的另外一个途径就是使用命令的输出。你可以用反引号来执行任何能产生输出的命令，然后在for命令中使用该命令的输出：

```

$ cat test5
#!/bin/bash
# reading values from a file

file="states"

for state in `cat $file`
do
    echo "Visit beautiful $state"
done
$ cat states
Alabama
Alaska
Arizona
Arkansas
Colorado
Connecticut
Delaware
Florida
Georgia
$ ./test5
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
$ 
```

这个例子在反引号中使用了cat命令来输出文件states的内容。你会注意到states文件每行都有一个州，而不是通过空格分割的。for命令仍然以每次一行的方式遍历了cat命令的输出，假定每个州都是在单独的一行上。但这并没有解决数据中有空格的问题。如果你列出了一个名字中有空格的州，for命令仍然会将每个单词当作单独的值。这是有原因的，下一节我们将会了解。

说明 test5的代码范例将文件名赋给变量，只用了文件名而不是路径。这要求文件和脚本位于同一个目录中。如果不是的话，你需要使用全路径名（不管是绝对路径还是相对路径）来引用文件位置。

12.1.5 更改字段分隔符

这个问题的原因是特殊的环境变量IFS，称为内部字段分隔符（internal field separator）。IFS环境变量定义了bash shell用作字段分隔符的一系列字符。默认情况下，bash shell会将下列字符当作字段分隔符：

- 空格；
- 制表符；
- 换行符。

如果bash shell在数据中看到了这些字符中的任意一个，它就会假定你在列表中开始了一个新的数据段。在处理可能含有空格的数据（比如文件名）时，这会非常麻烦，如你在上一个脚本示例中看到的。

要解决这个问题，你可以在shell脚本中临时更改IFS环境变量的值来限制一下被bash shell当作字段分隔符的字符。但这种方式有点奇怪。比如，如果你修改IFS的值使其只能识别换行符，你必须这么做：

```
IFS=$'\n'
```

将这个语句加入到脚本中，告诉bash shell在数据值中忽略空格和制表符。对前一个脚本使用这种方法，将获得如下输出：

```
$ cat test5b
#!/bin/bash
# reading values from a file

file="states"

IFS=$'\n'
for state in `cat $file`
do
    echo "Visit beautiful $state"
done
$ ./test5b
```

```

Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
Visit beautiful New York
Visit beautiful New Hampshire
Visit beautiful North Carolina
$
```

现在shell脚本能够使用列表中含有空格的值了。

警告 在处理长脚本时，可能在一个地方需要修改IFS的值，然后忘掉它了并在脚本中其他地方以为还是默认的值。一个可参考的简单实践是在改变IFS之前保存原来的IFS值，之后再恢复它。

这种技术可以这样编程：

```

IFS_OLD=$IFS
IFS=$'\n'
<use the new IFS value in code>
IFS=$IFS_OLD
```

这会为脚本中后面的操作保证IFS的值恢复到了默认值。

还有其他一些IFS环境变量的绝妙用法。假定你要遍历一个文件中用冒号分割的值（比如在/etc/passwd文件中）。你要做的就是将IFS的值设为冒号：

```
IFS=:
```

如果你要指定多个IFS字符，只要将它们在赋值行串起来就行：

```
IFS=$'\n:;'"
```

这个赋值会将换行符、冒号、分号和双引号作为字段分隔符。如何使用IFS字符解析数据没有任何限制。

12.1.6 用通配符读取目录

最后，你可以用for命令来自动遍历满是文件的目录。进行此操作时，你必须在文件名或路径名中使用通配符。它会强制shell使用文件扩展匹配（file globbing）。文件扩展匹配是生成匹配指定的通配符的文件名或路径名的过程。

这个特性在处理目录中的文件而你不知道所有的文件名时非常好用：

```
$ cat test6
#!/bin/bash
# iterate through all the files in a directory

for file in /home/rich/test/*
do

    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
$ ./test6
/home/rich/test/dirl is a directory
/home/rich/test/myprog.c is a file
/home/rich/test/myprog is a file
/home/rich/test/myscript is a file
/home/rich/test/newdir is a directory
/home/rich/test/newfile is a file
/home/rich/test/newfile2 is a file
/home/rich/test/testdir is a directory
/home/rich/test/testing is a file
/home/rich/test/testprog is a file
/home/rich/test/testprog.c is a file
$
```

for命令会遍历`/home/rich/test/*`输出的结果。该代码用test命令测试了每个条目（使用方括号方法），以查看它是个目录（通过-d参数）还是个文件（通过-f参数）。（参见第11章。）

注意，在这个例子中，我们和if语句里的测试处理得有些不同：

```
if [ -d "$file" ]
```

在Linux中，目录名和文件名中包含空格当然是合法的。要容纳这种值，你应该将\$file变量用双引号圈起来。如果不这么做，遇到含有空格的目录名或文件名时会有错误产生：

```
./test6: line 6: [: too many arguments
./test6: line 9: [: too many arguments
```

在test命令中，bash shell会将额外的单词当作参数，造成错误。

你也可以在for命令中通过列出一系列的目录通配符来将目录查找方法和列表方法合并进同一个for语句：

```
$ cat test7
#!/bin/bash
# iterating through multiple directories

for file in /home/rich/.b* /home/rich/badtest
do
    if [ -d "$file" ]
    then
```

```

echo "$file is a directory"
elif [ -f "$file" ]
then
    echo "$file is a file"
else
    echo "$file doesn't exist"
fi
done
$ ./test7
/home/rich/.backup.timestamp is a file
/home/rich/.bash_history is a file
/home/rich/.bash_logout is a file
/home/rich/.bash_profile is a file
/home/rich/.bashrc is a file
/home/rich/badtest doesn't exist
$
```

for语句首先使用了文件扩展匹配来遍历通配符生成的文件列表，然后它会遍历列表中的下一个文件。你可以将任意多的通配符放进列表中。

警告 注意，你可以在列表数据中放入任何东西。即使文件或目录不存在，for语句也会尝试处理你放到列表中的任何东西。在处理文件或目录时，这可能会是个问题。你也不知道你正在尝试遍历一个并不存在的目录：在处理之前测试一下文件或目录总是好的。

12.2 C语言风格的for命令

如果你用C语言编程序，你可能会对bash shell使用for命令的方式有点惊奇。在C语言中，for循环通常定义一个变量，然后这个变量会在每次迭代时自动改变。通常，程序员会将这个变量用作计数器，并在每次迭代中让计数器增一或减一。bash的for命令也提供了这个功能。本节将会告诉你如何在bash shell脚本中使用C语言风格的for命令。

12.2.1 C语言的for命令

C语言的for命令有一个用来指明变量的特殊方法、一个必须保持成立才能继续迭代的条件，以及另一个为每个迭代改变变量的方法。当指定的条件不成立时，for循环就会停止。条件等式通过标准的数学符号定义。比如，考虑下面的C语言代码：

```

for (i = 0; i < 10; i++)
{
    printf("The next number is %d\n", i);
}
```

这段代码产生了一个简单的迭代循环，其中变量*i*作为计数器。第一部分将一个默认值赋给该变量。中间的部分定义了循环重复的条件。当定义的条件不成立时，for循环就停止迭代了。

最后一部分定义了迭代的过程。在每次迭代之后，最后一部分中定义的表达式会被执行。在本例中，*i*变量会在每次迭代后增一。

bash shell也支持一个版本的for循环，看起来跟C语言风格的for循环类似，虽然它也有一些细微的不同，包括一些会叫shell脚本程序员困惑的东西。这是bash中C语言风格的for循环的基本格式：

```
for (( variable assignment : condition ; iteration process ))
```

C语言风格的for循环的格式可能对bash shell脚本程序员来说有点困惑，由于它使用了C语言风格的变量引用方式而不是shell风格的变量引用方式。C语言风格的for命令看起来如下：

```
for (( a = 1; a < 10; a++ ))
```

注意，有一些事情没有遵循标准的bash shell for命令：

- 给变量赋值可以有空格；
- 条件中的变量不以美元符开头；
- 迭代过程的算式未用expr命令格式。

shell开发人员创建了这种格式以更贴切地模仿C语言风格的for命令。这对C语言程序员来说很好，但也可能将即使是专家级的shell程序员弄得一头雾水。在脚本中使用C语言风格的for循环时要小心。

这里有个在bash shell程序中使用C语言风格的for命令的例子：

```
$ cat test8
#!/bin/bash
# testing the C-style for loop

for (( i=1; i <= 10; i++ ))
do
    echo "The next number is $i"
done
$ ./test8
The next number is 1
The next number is 2
The next number is 3
The next number is 4
The next number is 5
The next number is 6
The next number is 7
The next number is 8
The next number is 9
The next number is 10
$
```

for循环通过定义好的变量（本例中是字母*i*）遍历了这些命令。在每个迭代中，\$*i*变量包含了for循环中赋给的值。在每次迭代后，循环的迭代过程会作用在变量上，在本例中，变量增一。

12.2.2 使用多个变量

C语言风格的for命令也允许你为迭代使用多个变量。循环会单独处理每个变量，允许你为每个变量定义不同的迭代过程。当你有多个变量时，你只能在for循环中定义一种条件：

```
$ cat test9
#!/bin/bash
# multiple variables

for (( a=1, b=10; a <= 10; a++, b-- ))
do
    echo "$a - $b"
done
$ ./test9
1 - 10
2 - 9
3 - 8
4 - 7
5 - 6
6 - 5
7 - 4
8 - 3
9 - 2
10 - 1
$
```

变量a和b每个都用不同的值来初始化并且定义了不同的迭代过程。循环每个迭代中增加变量a的同时，减小了变量b。

12.3 while 命令

while命令某种意义上是if-then语句和for循环的混杂体。while命令允许你定义一个要测试的命令，然后循环执行一组命令，只要定义的测试命令返回的是退出状态码0。它会在每个迭代的一开始测试test命令。在test命令返回非0退出状态码时，while命令会停止执行那组命令。

12.3.1 while的基本格式

while命令的格式是：

```
while test command
do
    other commands
done
```

while命令中定义的test命令和if-then语句(参见第11章)中定义的是一样的格式。和if-then语句中一样，你可以使用任何普通的bash shell命令，或者用test命令作为条件，比如变量值。

while命令的关键是，指定的test命令的退出状态码必须随着循环中运行的命令改变。如果退出状态码从不变，那while循环将会一直不停地循环。

最常见的test命令的用法是，用方括号来查看循环命令中用到的shell变量的值：

```
$ cat test10
#!/bin/bash
# while command test

var1=10
while [ $var1 -gt 0 ]
do
    echo $var1
    var1=$(( $var1 - 1 ))
done
$ ./test10
10
9
8
7
6
5
4
3
2
1
$
```

while命令定义了每次迭代时检查的测试条件：

```
while [ $var1 -gt 0 ]
```

只有测试条件成立，while命令才会继续遍历执行定义好的命令。在这些命令中，测试条件中用到的变量必须被修改，否则你就进入了一个无限循环。在本例中，我们用shell算术来将变量值减一：

```
var1=$(( $var1 - 1 ))
```

while循环会在测试条件不再成立时停止。

12.3.2 使用多个测试命令

在极少数情况下，while命令允许你在while语句行定义多个测试命令。只有最后一个测试命令的退出状态码会被用来决定什么时候结束循环。如果你不够小心，这可能会导致一些有意思的结果。下面的例子将会说明：

```
$ cat test11
#!/bin/bash
# testing a multicommmand while loop

var1=10

while echo $var1
[ $var1 -ge 0 ]
```

```

do
    echo "This is inside the loop"
    var1=$((var1 - 1))
done
$ ./test11
10
This is inside the loop
9
This is inside the loop
8
This is inside the loop
7
This is inside the loop
6
This is inside the loop
5
This is inside the loop
4
This is inside the loop
3
This is inside the loop
2
This is inside the loop
1
This is inside the loop
0
This is inside the loop
-1
$
```

注意本例中做了什么。在while语句中定义了两个测试命令：

```

while echo $var1
    [ $var1 -ge 0 ]
```

第一个测试会简单地显示var1变量的当前值。第二个测试用方括号来决定var1变量的值。在循环内部，echo语句会显示一条简单的消息，说明循环被执行了。注意当你运行本例时输出是如何结尾的：

```

This is inside the loop
-1
$
```

while循环会在var1变量等于零时执行echo语句，然后将var1变量的值减一。下一步，测试命令会为下个迭代执行。echo测试命令被执行了，显示了var1变量的值（现在比0小）。shell直到执行test测试命令了while循环才会停止。

这说明在含有多个命令的while语句中，在每次迭代中所有的测试命令都会被执行，包括最后一个测试命令不成立的最后那次循环。要小心这种用法。另一个要小心的是你如何指定多个测试命令。注意每个测试命令都是在单独的一行上。

12.4 until 命令

until命令和while命令工作的方式完全相反。until命令要求你指定一个通常输出非零退出状态码的测试命令。只有测试命令的退出状态码非零，bash shell才会指定循环中列出的那些命令。一旦测试命令返回了退出状态码0，循环就结束了。

如你所期望的，until命令的格式如下：

```
until test commands
do
    other commands
done
```

类似于while命令，你可以在until命令语句中有多个测试命令。只有最后一个命令的退出状态码决定bash shell是否执行定义好的其他命令。

下面是使用until命令的一个例子：

```
$ cat test12
#!/bin/bash
# using the until command

var1=100

until [ $var1 -eq 0 ]
do
    echo $var1
    var1=$(( $var1 - 25 ))
done
$ ./test12
100
75
50
25
$
```

本例中会测试var1变量来决定until循环何时停止。只要变量的值等于0了，until命令就会停止循环了。同while命令一样，在和until命令一起使用多个测试命令时要注意：

```
$ cat test13
#!/bin/bash
# using the until command

var1=100

until echo $var1
    [ $var1 -eq 0 ]
do
    echo Inside the loop: $var1
    var1=$(( $var1 - 25 ))
done
$ ./test13
100
```

```
Inside the loop: 100
75
Inside the loop: 75
50
Inside the loop: 50
25
Inside the loop: 25
0
$
```

shell会执行指定的测试命令，只有在最后一个命令成立时停止。

12.5 嵌套循环

循环语句可以在循环内使用任意类型的命令，包括其他循环命令。这种称为嵌套循环（nested loop）。注意，在使用嵌套循环时，你是在迭代中使用迭代，命令运行的次数是乘积关系。不注意这点有可能会在脚本中造成问题。

这里有个在for循环中嵌套for循环的简单例子：

```
$ cat test14
#!/bin/bash
# nesting for loops

for (( a = 1; a <= 3; a++ ))
do
    echo "Starting loop $a:"
    for (( b = 1; b <= 3; b++ ))
    do
        echo "    Inside loop: $b"
    done
done
$ ./test14
Starting loop 1:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
Starting loop 2:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
Starting loop 3:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
$
```

这个被嵌套的循环（也称为内部循环）会在外部循环的每次迭代中遍历一遍它所有的值。注意，两个循环的do和done命令没有任何差别。bash shell知道当第一个done命令执行时是指内部循环而非外部循环。

在混用循环命令时也一样，比如在while循环内部放置一个for循环：

```
$ cat test15
#!/bin/bash
# placing a for loop inside a while loop

var1=5

while [ $var1 -ge 0 ]
do
    echo "Outer loop: $var1"
    for (( var2 = 1; $var2 < 3; var2++ ))
    do
        var3=$(( $var1 * $var2 ))
        echo "Inner loop: $var1 * $var2 = $var3"
    done
    var1=$(( $var1 - 1 ))
done
$ ./test15
Outer loop: 5
    Inner loop: 5 * 1 = 5
    Inner loop: 5 * 2 = 10
Outer loop: 4
    Inner loop: 4 * 1 = 4
    Inner loop: 4 * 2 = 8
Outer loop: 3
    Inner loop: 3 * 1 = 3
    Inner loop: 3 * 2 = 6
Outer loop: 2
    Inner loop: 2 * 1 = 2
    Inner loop: 2 * 2 = 4
Outer loop: 1
    Inner loop: 1 * 1 = 1
    Inner loop: 1 * 2 = 2
Outer loop: 0
    Inner loop: 0 * 1 = 0
    Inner loop: 0 * 2 = 0
$
```

同样，shell能够区分开内部for循环的do和done命令和外部while循环的do和done命令。
如果你真的想锻炼一下大脑，甚至可以混用until和while循环：

```
$ cat test16
#!/bin/bash
# using until and while loops

var1=3

until [ $var1 -eq 0 ]
do
    echo "Outer loop: $var1"
    var2=1
    while [ $var2 -lt 5 ]
    do
        var3=$(echo "scale=4; $var1 / $var2" | bc)
        echo "    Inner loop: $var1 / $var2 = $var3"
    done
done
```

```

var2=$[ $var2 + 1 ]
done
var1=$[ $var1 - 1 ]
done
$ ./test16
Outer loop: 3
    Inner loop: 3 / 1 = 3.0000
    Inner loop: 3 / 2 = 1.5000
    Inner loop: 3 / 3 = 1.0000
    Inner loop: 3 / 4 = .7500
Outer loop: 2
    Inner loop: 2 / 1 = 2.0000
    Inner loop: 2 / 2 = 1.0000
    Inner loop: 2 / 3 = .6666
    Inner loop: 2 / 4 = .5000
Outer loop: 1
    Inner loop: 1 / 1 = 1.0000
    Inner loop: 1 / 2 = .5000
    Inner loop: 1 / 3 = .3333
    Inner loop: 1 / 4 = .2500
$
```

外部的until循环以值3开始并继续执行直到值等于0。内部while循环会以值1开始并一直执行，只要值小于5。每个循环都必须改变在测试条件中用到的值，否则循环就会无止尽进行下去。

12.6 循环处理文件数据

通常，你必须遍历存储在文件中的数据。这要求结合已经讲过的两种技术：

- 使用嵌套循环；
- 修改IFS环境变量。

通过修改IFS环境变量，你就能强制for命令将文件中的每行都当成单独的一个条目来处理，即便数据中有空格也是如此。一旦你从文件中提取出了单独的行，你可能需要再次循环来提取其中的数据。

经典的例子是处理/etc/passwd文件中的数据。这要求你逐行遍历/etc/passwd文件并将IFS变量的值改成冒号，这样你就能分隔开每行中的各个单独的数据段了：

```

#!/bin/bash
# changing the IFS value

IFS_OLD=$IFS
IFS=$'\n'
for entry in `cat /etc/passwd`
do
    echo "Values in $entry -"
    IFS=:
    for value in $entry
    do
        echo "    $value"
    done
done
```

```
done
$
```

这个脚本使用了两个不同的IFS值来解析数据。第一个IFS值解析出/etc/passwd文件中的单独的行。内部for循环进一步将IFS的值修改为冒号，允许你从/etc/passwd的行中解析出单独的值。

在运行这个脚本时，你会得到如下输出：

```
Values in rich:x:501:501:Rich Blum:/home/rich:/bin/bash -
rich
x
501
501
Rich Blum
/home/rich
/bin/bash
Values in katie:x:502:502:Katie Blum:/home/katie:/bin/bash -
katie
x
506
509
Katie Blum
/home/katie
/bin/bash
```

内部循环会解析出/etc/passwd每行中的每个单独的值。这用来处理通常导入电子表格采用的逗号分隔的数据也很方便。

12.7 控制循环

你可能会想，一旦启动了循环，就必须等到循环完成所有的迭代。不是这样的。有几个命令能帮我们控制循环内部的情况：

- break命令；
- continue命令。

每个命令在如何控制循环的执行上有不同的用法。下面几节将会介绍如何使用这些命令来控制循环的执行。

12.7.1 break命令

break命令是退出进行中的循环的一个简单方法。你可以用break命令来退出任意类型的循环，包括while和until循环。

有几种情况你可以使用break命令。本节将会介绍这些方法中的每一种。

1. 跳出单个循环

在shell执行break命令时，它会尝试跳出正在处理的循环：

```
$ cat test17
#!/bin/bash
```

```
# breaking out of a for loop

for var1 in 1 2 3 4 5 6 7 8 9 10
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Iteration number: $var1"
done
echo "The for loop is completed"
$ ./test17
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
The for loop is completed
$
```

for循环通常都会遍历一遍列表中指定的所有值。但当满足if-then的条件时，shell会执行break命令，for循环会停止。

这种方法同样适用于while和until循环：

```
$ cat test18
#!/bin/bash
# breaking out of a while loop

var1=1

while [ $var1 -lt 10 ]
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Iteration: $var1"
    var1=$(( $var1 + 1 ))
done
echo "The while loop is completed"
$ ./test18
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
The while loop is completed
$
```

while循环会在if-then的条件满足时执行break命令，终止。

2. 跳出内部循环

在处理多个循环时，break命令会自动终止你所在最里面的循环：

```
$ cat test19
#!/bin/bash
```

```
# breaking out of an inner loop

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
        if [ $b -eq 5 ]
        then
            break
        fi
        echo "  Inner loop: $b"
    done
done
$ ./test19
Outer loop: 1
  Inner loop: 1
  Inner loop: 2
  Inner loop: 3
  Inner loop: 4
Outer loop: 2
  Inner loop: 1
  Inner loop: 2
  Inner loop: 3
  Inner loop: 4
Outer loop: 3
  Inner loop: 1
  Inner loop: 2
  Inner loop: 3
  Inner loop: 4
$
```

内部循环里的`for`语句指明一直重复直到变量`b`等于100。但内部循环的`if-then`语句指明当变量`b`的值等于5时执行`break`命令。注意，即使内部循环通过`break`命令终止了，外部循环依然按指定的继续执行。

3. 跳出外部循环

有时你在内部循环，但需要停止外部循环。`break`命令接受单个命令行参数值：

```
break n
```

其中`n`说明了要跳出的循环层级。默认情况下，`n`为1，表明跳出的是当前的循环。如果你将`n`设为2，`break`命令就会停止下一级的外部循环：

```
$ cat test20
#!/bin/bash
# breaking out of an outer loop

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
```

```

if [ $b -gt 4 ]
then
    break 2
fi
echo "  Inner loop: $b"
done
$ ./test20
Outer loop: 1
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
$
```

注意当shell执行了break命令时，外部循环停止了。

12.7.2 continue命令

continue命令是提早结束执行循环内部的命令但并不完全终止整个循环的一个途径。它允许你在循环内部设置shell不执行命令的条件。这里有个在for循环中使用continue命令的简单例子：

```

$ cat test21
#!/bin/bash
# using the continue command

for (( var1 = 1; var1 < 15; var1++ ))
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "Iteration number: $var1"
done
$ ./test21
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Iteration number: 10
Iteration number: 11
Iteration number: 12
Iteration number: 13
Iteration number: 14
$
```

当if-then语句的条件被满足时（值大于5而小于10），shell会执行continue命令，跳过循环中的其他命令，但循环会继续。当if-then的条件不再被满足时，一切又回到正轨了。

也可以在while和until循环中使用continue命令，但要特别小心。记住，当shell执行continue命令时，它会跳过剩余的命令。如果你在这些条件中的某条中增加测试条件变量，问题就出现了：

```
$ cat badtest3
#!/bin/bash
# improperly using the continue command in a while loop

var1=0

while echo "while iteration: $var1"
    [ $var1 -lt 15 ]
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "    Inside iteration number: $var1"
    var1=$((var1 + 1))
done
$ ./badtest3 | more
while iteration: 0
    Inside iteration number: 0
while iteration: 1
    Inside iteration number: 1
while iteration: 2
    Inside iteration number: 2
while iteration: 3
    Inside iteration number: 3
while iteration: 4
    Inside iteration number: 4
while iteration: 5
    Inside iteration number: 5
while iteration: 6
$
```

你可能要确保你将脚本的输出重定向到了more命令，这样才能停止这些。所有一切看起来都很正常，直到满足了if-then的条件，shell执行了continue命令。当shell执行continue命令时，它会跳过while循环中的其他命令。遗憾的是，这正是while测试命令中被测的\$var1计数变量增加的地方。这意味着这个变量不会再增长了，正如你从前面连续的输出显示中看到的。

和break命令一样，continue命令也允许通过命令行参数指定要继续哪级循环：

continue n

其中n定义了要继续的循环层级。下面是继续外部for循环的一个例子：

```
$ cat test22
#!/bin/bash
# continuing an outer loop

for (( a = 1; a <= 5; a++ ))
do
    echo "Iteration $a:"
    for (( b = 1; b < 3; b++ ))
    do
        if [ $a -gt 2 ] && [ $a -lt 4 ]
        then
            continue 2
        fi
        var3=$(( $a * $b ))
        echo "    The result of $a * $b is $var3"
    done
done
$ ./test22
Iteration 1:
    The result of 1 * 1 is 1
    The result of 1 * 2 is 2
Iteration 2:
    The result of 2 * 1 is 2
    The result of 2 * 2 is 4
Iteration 3:
Iteration 4:
    The result of 4 * 1 is 4
    The result of 4 * 2 is 8
Iteration 5:
    The result of 5 * 1 is 5
    The result of 5 * 2 is 10
$
```

其中的if-then语句：

```
if [ $a -gt 2 ] && [ $a -lt 4 ]
then
    continue 2
fi
```

用continue命令来停止处理循环内的命令但继续处理外部循环。注意值为3的迭代的脚本输出未再处理任何内部循环语句，因为continue命令停止了处理过程，但外部循环依然会继续。

12.8 处理循环的输出

最后，在shell脚本中，你要么管接要么重定向循环的输出。你可以在done命令之后添加一个处理命令：

```
for file in /home/rich*
do
    if [ -d "$file" ]
    then
```

```

echo "$file is a directory"
elif
echo "$file is a file"
fi
done > output.txt

```

shell会将for命令的结果重定向到文件output.txt中，而不是显示在屏幕上。

考虑下面重定向for命令的输出到文件的例子：

```

$ cat test23
#!/bin/bash
# redirecting the for output to a file

for (( a = 1; a < 10; a++ ))
do
    echo "The number is $a"
done > test23.txt
echo "The command is finished."
$ ./test23
The command is finished.
$ cat test23.txt
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
$ 

```

shell创建了文件test23.txt并将for命令的输出重定向到这个文件。shell在for命令之后如常显示了echo语句。

这种方法同样适用于将循环的结果管接给另一个命令：

```

$ cat test24
#!/bin/bash
# piping a loop to another command

for state in "North Dakota" Connecticut Illinois Alabama Tennessee
do
    echo "$state is the next place to go"
done | sort
echo "This completes our travels"
$ ./test24
Alabama is the next place to go
Connecticut is the next place to go
Illinois is the next place to go
North Dakota is the next place to go
Tennessee is the next place to go
This completes our travels
$ 

```

state值并没有在for命令列表中以特定次序列出。for命令的输出传给了sort命令，它会改变for命令输出的结果的顺序。运行这个脚本实际上说明了结果已经在脚本内部排好序了。

12.9 小结

循环是编程中的框架部分。**bash shell**提供了3种不同的可在脚本中使用的循环命令。for命令允许你遍历一系列的值，不管是在命令行里提供好的、变量里包含的，还是通过文件扩展匹配获得的、通过通配符提取的文件名和目录名。

while命令提供了基于命令条件的循环，使用普通命令或test命令来测试变量的条件。只有在命令（或条件）产生退出状态码0时，while循环才会继续遍历指定的一组命令。

until命令也提供了遍历命令的一种方法，但它的迭代是建立在有命令（或条件）产生非零退出状态码的基础上的。这个特性允许你设置一个迭代结束前都必须满足的条件。

你可以在**shell**脚本中组合循环，生成多个层级的循环。**bash shell**提供了continue和break命令，来基于循环内的不同值改变通常的循环过程的流程。

bash shell还允许使用标准的命令重定向和管道来改变循环的输出。你可以使用重定向来将循环的结果重定向到一个文件，或管道来将循环的结果重定向给另一个命令。这提供了控制**shell**脚本执行的有用的特性。

下一章将会讨论如何和**shell**脚本用户交互。通常，**shell**脚本不是完全自成一体的。它们需要一些运行时提供的外部数据。下一章将讨论可用来向**shell**脚本提供实时数据的不同的方法。

本章内容

- 命令行参数
- 特殊参数变量
- 移动变量
- 处理选项
- 将选项标准化
- 获得用户输入

到目前为止你看到的都是如何编写脚本处理数据、变量和Linux系统上的文件。有时，你需要写个和运行脚本的人交互的脚本。`bash shell`提供了一些不同的方法来从用户处获得数据，包括命令行参数（添加在命令后的数据值）、命令行选项（可修改命令行为的单字母值）以及直接从键盘读取输入的能力。本章将会讨论如何将这些不同的方法放进你的`bash shell`脚本来从运行脚本的用户处获得数据。

13.1 命令行参数

向`shell`脚本传数据的最基本方法是使用命令行参数。命令行参数允许在运行脚本时向命令行添加数据值：

```
$ ./addem 10 30
```

本例向脚本`addem`传递了两个命令行参数（10和30）。脚本会通过特殊的变量来处理命令行参数。后面几节将会介绍如何在`bash shell`脚本中使用命令行参数。

13.1.1 读取参数

`bash shell`会将一些称为位置参数（positional parameter）的特殊变量分配给命令行输入的所有参数。这甚至包括`shell`执行的程序的名字。位置参数变量是标准的数字：`$0`是程序名，`$1`是第一个参数，`$2`是第二个参数，依次类推，直到第9个参数`$9`。

下面是在shell脚本中使用单个命令行参数的简单例子：

```
$ cat test1
#!/bin/bash
# using one command line parameter

factorial=1
for (( number = 1; number <= $1 ; number++ ))
do
    factorial=$(( $factorial * $number ))
done
echo The factorial of $1 is $factorial
$
$ ./test1 5
The factorial of 5 is 120
$
```

可以在shell脚本中像使用其他变量一样使用\$1变量。shell脚本会自动将命令行参数的值分配给变量。不需要作任何处理。

如果需要输入更多的命令行选项，则在命令行上每个参数都必须用空格分开：

```
$ cat test2
#!/bin/bash
# testing two command line parameters

total=$(( $1 * $2 ))
echo The first parameter is $1.
echo The second parameter is $2.
echo The total value is $total.
$
$ ./test2 2 5
The first parameter is 2.
The second parameter is 5.
The total value is 10.
$
```

shell会将每个参数分配给对应的变量。

在本例中，用到的命令行参数都是数值。也可以在命令行上用文本字符串：

```
$ cat test3
#!/bin/bash
# testing string parameters

echo Hello $1, glad to meet you.
$
$ ./test3 Rich
Hello Rich, glad to meet you.
$
```

shell将输入到命令行的字符串值传给了脚本。但在用含有空格的文本字符串时，会遇到问题：

```
$ ./test3 Rich Blum
Hello Rich, glad to meet you.
$
```

记住，每个参数都是用空格分割的，所以shell会将空格当成分割两个值的分隔符。要在参数值中包含空格，必须要用引号（单引号双引号均可）：

```
$ ./test3 'Rich Blum'
Hello Rich Blum, glad to meet you.
$
$ ./test3 "Rich Blum"
Hello Rich Blum, glad to meet you.
$
```

说明 将文本字符串作为参数传递时，引号并不是数据的一部分。它们只是将数据的开始和结尾和其他内容分开。

如果脚本需要多于9个命令行参数，你仍然可以处理，但是需要稍微修改一下变量名。在第9个变量之后，你必须在变量数字周围加花括号，比如\${10}。下面是个实现的例子：

```
$ cat test4
#!/bin/bash
# handling lots of parameters

total=${[ ${10} * ${11} ]}
echo The tenth parameter is ${10}
echo The eleventh parameter is ${11}
echo The total is $total
$
$ ./test4 1 2 3 4 5 6 7 8 9 10 11 12
The tenth parameter is 10
The eleventh parameter is 11
The total is 110
$
```

这个技术将允许你向脚本添加任意多的要用的命令行参数。

13.1.2 读取程序名

你可以用\${0}参数来获取shell在命令行启动的程序的名字。这在写有多个功能的工具时很方便。但有个小问题需要处理一下。看下面这个简单例子中会怎样：

```
$ cat test5
#!/bin/bash
# testing the $0 parameter

echo The command entered is: $0
$
$ ./test5
The command entered is: ./test5
$
$ /home/rich/test5
The command entered is: /home/rich/test5
$
```

当传给\$0变量的真实字符串是整个脚本的路径时，程序中就会使用整个路径，而不仅仅是程序名。

如果你要写个基于命令行下运行的脚本名来执行不同功能的脚本，这需要一点工夫。你需要能去掉命令行下运行脚本的任何路径。

幸而有个方便的小命令正好可以做这个。basename命令会只返回程序名而不包括路径。让我们修改一下示例脚本，看看它如何工作：

```
$ cat test5b
#!/bin/bash
# using basename with the $0 parameter

name=`basename $0`
echo The command entered is: $name
$ ./test5b
The command entered is: test5b
$ ./home/rich/test5b
The command entered is: test5b
$
```

现在好多了。你可以用这种方法来编写基于所用的脚本名而执行不同功能的脚本。这里有个简单的例子来说明：

```
$ cat test6
#!/bin/bash
# testing a multi-function script

name=`basename $0`

if [ $name = "addem" ]
then
    total=$[ $1 + $2 ]
elif [ $name = "multem" ]
then
    total=$[ $1 * $2 ]
fi
echo The calculated value is $total
$ chmod u+x test6
$ cp test6 addem
$ ln -s test6 multem
$ ls -l
-rwxr--r-- 1 rich      rich   211 Oct 15 18:00 addem
lrwxrwxrwx  1 rich      rich      5 Oct 15 18:01 multem -> test6
-rwxr--r-- 1 rich      rich   211 Oct 15 18:00 test6
$
$ ./addem 2 5
The calculated value is 7
$
$ ./multem 2 5
```

```
The calculated value is 10
$
```

本例从test6的代码创建了两个不同的文件名，一个通过复制文件创建，另一个通过链接创建。在两种情况下，脚本都会先获得脚本的基名，然后根据该值执行相应的功能。

13.1.3 测试参数

在shell脚本中使用命令行参数时要小心些。如果脚本不加参数运行，可能会出问题：

```
$ ./addem 2
./addem: line 8: 2 + : syntax error: operand expected (error
token is ")
The calculated value is
$
```

当脚本认为参数变量中会有数据而实际上并没有时，你很可能会从脚本得到一个错误消息。这种写脚本的方法并不可取。在使用数据前检查数据确实已经存在于变量里很有必要：

```
$ cat test7
#!/bin/bash
# testing parameters before use

if [ -n "$1" ]
then
    echo Hello $1, glad to meet you.
else
    echo "Sorry, you did not identify yourself."
fi
$
$ ./test7 Rich
Hello Rich, glad to meet you.
$
$ ./test7
Sorry, you did not identify yourself.
$
```

在本例中，在test命令里使用了-n参数来检查命令行参数中是否有数据。在下一节中，你会看到还有另一种检查命令行参数的方法。

13.2 特殊参数变量

在bash shell中有些特殊变量，它们会记录命令行参数。本节将会介绍它们都是哪些变量以及如何使用它们。

13.2.1 参数计数

如你在上一节中看到的，通常在脚本中使用命令行参数之前应该检查一下命令行参数。对于使用多个命令行参数的脚本来说，这有点麻烦。

你可以只数一下命令行中输入了多少个参数，而不同测试每个参数。`bash shell`为此提供了一个特殊变量。

`$#`特殊变量含有脚本运行时就有的命令行参数的个数。你可以在脚本中任何地方使用这个特殊变量，就跟普通变量一样：

```
$ cat test8
#!/bin/bash
# getting the number of parameters

echo There were $# parameters supplied.
$ ./test8
There were 0 parameters supplied.
$ ./test8 1 2 3 4 5
There were 5 parameters supplied.
$ ./test8 1 2 3 4 5 6 7 8 9 10
There were 10 parameters supplied.
$ ./test8 "Rich Blum"
There were 1 parameters supplied.
$
```

现在你就能在使用参数前测试参数的总数了：

```
$ cat test9
#!/bin/bash
# testing parameters

if [ $# -ne 2 ]
then
    echo Usage: test9 a b
else
    total=$[ $1 + $2 ]
    echo The total is $total
fi
$ ./test9
Usage: test9 a b
$ ./test9 10
Usage: test9 a b
$ ./test9 10 15
The total is 25
$ ./test9 10 15 20
Usage: test9 a b
$
```

`if-then`语句用`test`命令来对命令行提供的参数总数执行数值测试。如果参数总数不对，你可以打印一条错误消息说明脚本的正确用法。

这个变量还提供了一个简便方法来在命令行上抓取最后一个参数，而不用知道到底用了多少个参数。不过你需要花点工夫。

如果你仔细考虑过，你可能会觉得既然\$#变量含有参数的总数值，那变量\${\$#}就代表了最后一个命令行参数变量。试试看会发生什么：

```
$ cat badtest1
#!/bin/bash
# testing grabbing last parameter

echo The last parameter was ${$#}
$
$ ./badtest1 10
The last parameter was 15354
$
```

啊，怎么了？显然，出了点问题。它表明你不能在花括号内使用美元符。你必须将美元符换成感叹号。很奇怪，它竟然能工作：

```
$ cat test10
#!/bin/bash
# grabbing the last parameter

params=${#}
echo The last parameter is $params
echo The last parameter is ${!#}
$
$ ./test10 1 2 3 4 5
The last parameter is 5
The last parameter is 5
$
$ ./test10
The last parameter is 0
The last parameter is ./test10
$
```

太好了。这个测试将\$#变量的值赋给了变量params，然后也按特殊命令行参数变量的格式使用了该变量。两个版本都能工作。重要的是要注意，当命令行上没有任何参数时，\$#的值为零，在params变量中也为零，但\${!#}变量会返回命令行用到的脚本名。

13.2.2 抓取所有的数据

有些情况下，你只想抓取命令行上提供的所有参数，然后遍历它们。你可以使用一组其他的特殊变量，而不用先用\$#变量来判断命令行上有多少参数然后再遍历它们。

\$*和\$@变量提供了对所有参数的快速访问。这两个都能够在单个变量中存储所有的命令行参数。

\$*变量会将命令行上提供的所有参数当做单个单词保存。每个词是指命令行上出现的每个值。基本上，\$*变量会将这些都做一个参数，而不是多个对象。

反过来说，\$@变量会将命令行上提供的所有参数当做同一字符串中的多个独立的单词。它允

许你遍历所有的值，将提供的每个参数分割开来。这通常通过for命令完成。

理解这两个变量如何工作可能容易叫人困惑。让我们看个例子，你就能理解二者之间的区别了：

```
$ cat test11
#!/bin/bash
# testing $* and $@

echo "Using the \$* method: $*"
echo "Using the \$@ method: $@"
$

$ ./test11 rich barbara katie jessica
Using the $* method: rich barbara katie jessica
Using the $@ method: rich barbara katie jessica
$
```

表面上看，两个变量产生的是同样的输出，都立即显示了提供的所有命令行参数。

下面的例子给出了二者的差异：

```
$ cat test12
#!/bin/bash
# testing $* and $@

count=1
for param in "$*"
do
    echo "\$* Parameter #$count = $param"
    count=$(( $count + 1 ))
done

count=1
for param in "$@"
do
    echo "\$@ Parameter #$count = $param"
    count=$(( $count + 1 ))
done
$

$ ./test12 rich barbara katie jessica
$* Parameter #1 = rich barbara katie jessica
$@ Parameter #1 = rich
$@ Parameter #2 = barbara
$@ Parameter #3 = katie
$@ Parameter #4 = jessica
$
```

现在清楚多了。通过使用for命令遍历这两个特殊变量，你能看到它们是如何不同地处理命令行参数的。`$*`变量会将所有参数当成单个参数，而`$@`变量会单独处理每个参数。这是变量命令行参数的绝妙方法。

13.3 移动变量

`bash shell`工具链中另一个工具是`shift`命令。`bash shell`提供了`shift`命令来帮助操作命令行参数。跟字面上的意思一样，`shift`命令会根据它们的相对位置来移动命令行参数。

在使用shift命令时，默认情况下它会将每个参数变量减一。所以，变量\$3的值会移到\$2，变量\$2的值会移到\$1，而变量\$1的值则会被删除（注意，变量\$0的值，也就是程序名不会改变）。

这是遍历命令行参数的另一个绝妙方法，尤其是在你不知道到底有多少参数时。你可以只操作第一个参数，移动参数，然后继续操作第一个参数。

这里有个例子来解释它如何工作：

```
$ cat test13
#!/bin/bash
# demonstrating the shift command

count=1
while [ -n "$1" ]
do
    echo "Parameter #$count = $1"
    count=$(( $count + 1 ))
    shift
done
$
$ ./test13 rich barbara katie jessica
Parameter #1 = rich
Parameter #2 = barbara
Parameter #3 = katie
Parameter #4 = jessica
$
```

这个脚本通过测试第一个参数值的长度执行了一个while循环。当第一个参数的长度为零时，循环结束。

测试第一个参数后，shift命令会将所有参数的位置移动一位。

另外，你也可以给shift命令提供一个参数来执行多位移动。只要提供你想移动的位数就行：

```
$ cat test14
#!/bin/bash
# demonstrating a multi-position shift

echo "The original parameters: $*"
shift 2
echo "Here's the new first parameter: $1"
$
$ ./test14 1 2 3 4 5
The original parameters: 1 2 3 4 5
Here's the new first parameter: 3
$
```

警告 使用shift命令时要小心。当一个参数被移除后，它的值会被丢掉并且无法恢复。

13.4 处理选项

如果你认真读过本书前面的所有内容，你应该见过了几个同时提供了参数和选项的bash命

令。选项（Options）是跟在单破折线后面的单个字母，能改变命令的行为。本节将会介绍3种不同的处理shell脚本中选项的方法。

13.4.1 查找选项

表面上看，命令行选项也没什么特殊的。在命令行上，它们紧跟在脚本名之后，就跟命令行参数一样。实际上，如果愿意，你可以像处理命令行参数一样处理命令行选项。

1. 处理简单选项

在前面的test15脚本中，你看到了如何使用shift命令来向下移动提供给脚本程序的命令行参数。你也可以用同样的方法来处理命令行选项。

在提取每个单独参数时，用case语句来判断参数是否被格式化成了选项：

```
$ cat test15
#!/bin/bash
# extracting command line options as parameters

while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done
$ ./test15 -a -b -c -d
Found the -a option
Found the -b option
Found the -c option
-d is not an option
$
```

case语句会检查每个参数是不是有效选项。是的话，相应的命令就会在case语句中运行。

不管选项按什么顺序出现在命令行上，这种方法都适用：

```
$ ./test15 -d -c -a
-d is not an option
Found the -c option
Found the -a option
$
```

case语句会在命令行参数中找到选项时就处理每个选项。如果有命令行上还提供了其他参数，你可以在case语句的通用情况处理部分中处理。

2. 分离参数和选项

你会经常遇到想在shell脚本中同时使用选项和参数的情况。Linux中处理这个问题的标准方式是用特殊字符来将二者分开，该字符会告诉脚本选项何时结束以及普通参数何时开始。

对于Linux来说，这个特殊字符是双破折线（--）。shell会用双破折线来表明选项结束了。看到双破折线之后，脚本会安全地将剩下的命令行参数当做参数来处理，而不是选项。

要检查双破折线，只要在case语句中加一项就行了：

```
$ cat test16
#!/bin/bash
# extracting options and parameters

while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break ;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done

count=1
for param in $0
do
    echo "Parameter #${count}: $param"
    count=$(( $count + 1 ))
done
$
```

这段脚本用break命令来在它遇到双破折线时跳出while循环。由于过早地跳出了循环，我们需要再加一条shift命令来将双破折线移出参数变量。

对于第一个测试，试试用一组普通的选项和参数来运行这个脚本：

```
$ ./test16 -c -a -b test1 test2 test3
Found the -c option
Found the -a option
Found the -b option
test1 is not an option
test2 is not an option
test3 is not an option
$
```

结果说明，在处理时脚本以为所有的命令行参数都是选项。下一步，试试同样的操作，只是这次会用双破折线来将命令行上的选项和参数划分开来：

```
$ ./test16 -c -a -b -- test1 test2 test3
Found the -c option
Found the -a option
Found the -b option
Parameter #1: test1
Parameter #2: test2
Parameter #3: test3
$
```

当脚本遇到双破折线时，它就停止处理选项了，并将剩下的参数都当做命令行参数。

3. 处理带值的选项

有些选项会带上一个额外的参数值。在这种情况下，命令行看起来像下面这样：

```
$ ./testing -a test1 -b -c -d test2
```

当命令行选项要求额外的参数时，脚本必须能检测并能正确地处理。下面有个如何处理的例子：

```
$ cat test17
#!/bin/bash
# extracting command line options and values

while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option";;
        -b) param="$2"
            echo "Found the -b option, with parameter value $param"
            shift 2;;
        -c) echo "Found the -c option";;
        --) shift
            break;;
        *) echo "$1 is not an option";;
    esac
    shift
done

count=1
for param in "$@"
do
    echo "Parameter #${count}: $param"
    count=~$[ ${count} + 1 ]
done
$ ./test17 -a -b test1 -d
Found the -a option
Found the -b option, with parameter value test1
-d is not an option
$
```

在这个例子中，case语句定义了3个它要处理的选项。-b选项也要求一个额外的参数值。由于要处理的参数是\$1，额外的参数值就应该位于\$2（因为所有的参数在处理完之后都会被移出去）。只要将参数值从\$2变量中提取出来就可以了。当然，因为我们要为这个选项使用两个参数位，所以你还需要设定shift命令移动两个位置。

只用基本的特性，这个过程就能工作，不管按什么顺序放置选项（但要记住包含每个选项相应的选项参数）：

```
$ ./test17 -b test1 -a -d
Found the -b option, with parameter value test1
Found the -a option
```

```
$ ./test17 -d
-d is not an option
$
```

现在shell脚本中已经有了处理命令行选项的基本能力，但还有一些限制。比如，如果你想将多个选项放进一个参数中时，它就不能工作了：

```
$ ./test17 -ac
-ac is not an option
$
```

在Linux中，合并选项是一个很常见的用法，而且如果脚本要更用户友好一些，那么也要给用户提供这种特性。幸好，有另外一种处理选项的方法能够帮忙。

13.4.2 使用getopt命令

getopt命令是一个在处理命令行选项和参数时非常方便的工具。它能够识别命令行参数，从而在脚本中解析它们时更方便。

1. 命令的格式

getopt命令可以接受一系列任意形式的命令行选项和参数，并自动将它们转换成适当的形式。它的命令格式如下：

```
getopt options optstring parameters
```

optstring是这个过程的关键所在。它定义了命令行有效的选项字母，还定义了哪些选项字母需要参数值。

首先，在optstring中列出你要在脚本中用到的每个命令行选项字母。然后，在每个需要参数值的选项字母后加一个冒号。getopt命令会基于你定义的optstring解析提供的参数。

下面是个getopt如何工作的简单例子：

```
$ getopt ab:cd -a -b test1 -cd test2 test3
-a -b test1 -c -d -- test2 test3
$
```

optstring定义了4个有效选项字母，a、b、c和d。它还定义了选项字母b需要一个参数值。当getopt命令运行时，它会检查提供的参数列表，并基于提供的optstring解析。注意它会自动将-cd选项分成两个单独的选项，并插入双破折线来分开行中的额外参数。

如果你指定了一个不在optstring中的选项，默认情况下，getopt命令会产生一条错误消息：

```
$ getopt ab:cd -a -b test1 -cde test2 test3
getopt: invalid option -- e
-a -b test1 -c -d -- test2 test3
$
```

如果想忽略这条错误消息，可以在命令后加-q选项：

```
$ getopt -q ab:cd -a -b test1 -cde test2 test3
-a -b 'test1'-c -d -- 'test2''test3'
$
```

注意，getopt命令必须列在optstring之前。现在你可以在脚本中使用此命令处理命令行选项了。

2. 在脚本中使用getopt

你可以在脚本中使用getopt来格式化输入给脚本的任何命令行选项。但用起来略微复杂。

方法是用getopt命令生成的格式化后的版本来替换已有的命令行选项和参数。用set命令能够做到。

在第5章中，你就已经见过set命令了。set命令能够和shell中的不同变量一起工作。第5章介绍了如何使用set命令来显示所有的系统环境变量。

set命令的选项之一是双破折线，它会将命令行参数替换成set命令的命令行的值。

然后，该方法会将原始的脚本的命令行参数传给getopt命令，之后再将getopt命令的输出传给set命令，用getopt格式化后的命令行参数来替换原始的命令行参数。看起来如下：

```
set -- `getopts -q ab:c "$@"`
```

现在原始的命令行参数变量的值会被getopt命令的输出替换，而getopt已经为我们格式化好了命令行参数。

利用该方法，现在我们就可以写出能帮我们处理命令行参数的脚本了：

```
$ cat test18
#!/bin/bash
# extracting command line options and values with getopt

set -- `getopt -q ab:c "$@"`
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) param="$2"
            echo "Found the -b option, with parameter value $param"
            shift ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done

count=1
for param in "$@"
do
    echo "Parameter #$count: $param"
    count=$(( $count + 1 ))
done
```

你会注意到它跟test17脚本一样。唯一不同的是加入了getopt命令来帮助格式化命令行参数。现在运行脚本并加上复杂选项，可以看出它工作得更好了：

```
$ ./test18 -ac
Found the -a option
Found the -c option
$
```

当然，所有的原始功能还能顺利工作：

```
$ ./test18 -a -b test1 -cd test2 test3 test4
Found the -a option
Found the -b option, with parameter value 'test1'
Found the -c option
Parameter #1: 'test2'
Parameter #2: 'test3'
Parameter #3: 'test4'
$
```

现在事情看起来好多了。但是，仍然有一个小问题潜伏在getopt命令中。看看这个例子：

```
$ ./test18 -a -b test1 -cd "test2 test3" test4
Found the -a option
Found the -b option, with parameter value 'test1'
Found the -c option
Parameter #1: 'test2'
Parameter #2: 'test3'
Parameter #3: 'test4'
$
```

getopt命令并不擅长处理带空格的参数值。它会将空格当做参数分隔符，而不是根据双引号将二者当做一个参数。幸而，还有另外一个办法能解决这个问题。

13.4.3 使用更高级的getopts

bash shell包含了getopts命令（注意是复数）。它跟近亲getopt看起来很像，但有一些扩展功能。

与getopt将命令行上找到的选项和参数处理后只生成一个输出不同，getopts命令能够和已有的shell参数变量对应地顺序工作。

每次调用它时，它只处理一个命令行上检测到的参数。处理完所有的参数后，它会退出并返回一个大于零的退出状态码。这让它非常适合用在解析命令行所有参数的循环中。

getopts命令的格式如下：

```
getopts optstring variable
```

optstring值类似于getopt命令中的那个。有效的选项字母都会列在optstring中，如果选项字母要求有个参数值，就加一个冒号。要去掉错误消息的话，可以在optstring之前加一个冒号。getopts命令将当前参数保存在命令行中定义的variable中。

getopts命令会用到两个环境变量。如果选项需要跟一个参数值，OPTARG环境变量就会保存这个值。OPTIND环境变量保存了参数列表中getopts正在处理的参数位置。这样你就能在处理完选项之后继续处理其他命令行参数了。

让我们看个使用getopts命令的简单例子：

```
$ cat test19
#!/bin/bash
# simple demonstration of the getopts command

while getopts :ab:c opt
do
    case "$opt" in
        a) echo "Found the -a option" ;;
        b) echo "Found the -b option, with value $OPTARG" ;;
        c) echo "Found the -c option" ;;
        *) echo "Unknown option: $opt" ;;
    esac
done
$ ./test19 -ab test1 -c
Found the -a option
Found the -b option, with value test1
Found the -c option
$
```

while语句定义了getopts命令，指明了要查找哪些命令行选项，以及每次迭代中存储它们的变量名。

你会注意到在本例中case语句的用法有些不同。getopts命令解析命令行选项时，它会移除开头的单破折线，所以在case定义中不用单破折线。

getopts命令有几个好用的功能。对新手来说，你可以在参数值中包含空格：

```
$ ./test19 -b "test1 test2" -a
Found the -b option, with value test1 test2
Found the -a option
$
```

另一个好用的功能是将选项字母和参数值放在一起使用，而不用加空格：

```
$ ./test19 -abtest1
Found the -a option
Found the -b option, with value test1
$
```

getopts命令能够从-b选项中正确解析出test1值。getopts命令的另一个好用的功能是，它能够将命令行上找到的所有未定义的选项统一输出成问号：

```
$ ./test19 -d
Unknown option: ?
$
$ ./test19 -acde
Found the -a option
Found the -c option
Unknown option: ?
Unknown option: ?
$
```

optstring中未定义的选项字母会以问号形式发送给代码。

getopts命令知道何时停止处理选项，并将参数留给你处理。在getopts处理每个选项时，它会将OPTIND环境变量值增一。在getopts完成处理时，你可以将OPTIND值和shift命令一起使用来移动参数：

```
$ cat test20
#!/bin/bash
# processing options and parameters with getopts

while getopts :ab:cd opt
do
    case "$opt" in
        a) echo "Found the -a option" ;;
        b) echo "Found the -b option, with value $OPTARG";;
        c) echo "Found the -c option";;
        d) echo "Found the -d option";;
        *) echo "Unknown option: $opt";;
    esac
done
shift ${OPTIND}-1

count=1
for param in "$@"
do
    echo "Parameter $count: $param"
    count=$((count + 1))
done
$
$ ./test20 -a -b test1 -d test2 test3 test4
Found the -a option
Found the -b option, with value test1
Found the -d option
Parameter 1: test2
Parameter 2: test3
Parameter 3: test4
$
```

现在你就有了一个能在所有shell脚本中使用的全功能命令行选项和参数处理工具。

13.5 将选项标准化

在创建shell脚本时，显然你可以控制具体怎么做。你完全可以决定用哪些字母选项和如何使用。

但有些字母选项在Linux世界里已经演变成某种标准的含义。如果你能在shell脚本中支持这些选项，脚本看起来能更友好一些。

表13-1显示了Linux中用到的一些命令行选项的通用含义。

表13-1 通用的Linux命令选项

选 项	描 述
-a	显示所有对象
-c	生成一个计数
-d	指定一个目录
-e	扩展一个对象
-f	指定读入数据的文件
-h	显示命令的帮助信息
-i	忽略文本大小写
-l	产生输出的长格式版本
-n	使用非交互模式（批量）
-o	指定将所有输出重定向到的输出文件
-q	以安静模式运行
-r	递归地处理目录和文件
-s	以安静模式运行
-v	生成详细输出
-x	排除某个对象
-y	对所有问题回答yes

通过学习本书时遇到的各种bash命令，你大概已经知道这些选项中大部分的含义了。你的选项也采用同样的含义，会让用户在和你的脚本交互时不用总去查手册。

13.6 获得用户输入

尽管命令行选项和参数是从脚本用户获得输入的一种重要方式，但有时脚本的交互性还可以更强一些。有时你想要在脚本运行时问个问题，并等待运行脚本的人来回答。**bash shell**为此提供了**read**命令。

13.6.1 基本的读取

read命令接受从标准输入（键盘）或另一个文件描述符（参见第14章）的输入。在收到输入后，**read**命令会将数据放进一个标准变量。下面是**read**命令的最简单用法：

```
$ cat test21
#!/bin/bash
# testing the read command

echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my program. "
```

```
$ ./test21
Enter your name: Rich Blum
Hello Rich Blum, welcome to my program.
$
```

相当简单。注意生成提示的echo命令使用了-n选项。它会移掉字符串末尾的换行符，允许脚本用户紧跟其后输入数据，而不是下一行。这让脚本看起来更像表单。

实际上，read命令包含了-p选项，允许你直接在read命令行指定提示符：

```
$ cat test22
#!/bin/bash
# testing the read -p option
read -p "Please enter your age: " age
days=$(( $age * 365 ))
echo "That makes you over $days days old!"
$ ./test22
Please enter your age:10
That makes you over 3650 days old!
$
```

你会注意到，在第一个例子中当有名字输入时，read命令会将姓和名保存在同一个变量中。read命令会为提示符输入的所有数据分配一个变量，或者你也可以指定多个变量。输入的每个数据值都会分配给表中的下一个变量。如果变量表在数据之前用完了，剩下的数据就都会分配给最后一个变量：

```
$ cat test23
#!/bin/bash
# entering multiple variables

read -p "Enter your name: " first last
echo "Checking data for $last, $first..."
$ ./test23
Enter your name: Rich Blum
Checking data for Blum, Rich...
$
```

你可以在read命令行中不指定变量。如果那么做了，read命令会将它收到的任何数据都放进特殊环境变量REPLY中：

```
$ cat test24
#!/bin/bash
# testing the REPLY environment variable

read -p "Enter a number: "
factorial=1
for (( count=1; count <= $REPLY; count++ ))
do
    factorial=$(( $factorial * $count ))
done
```

```

echo "The factorial of $REPLY is $factorial"
$ ./test24
Enter a number: 5
The factorial of 5 is 120
$ 
```

REPLY环境变量会保存输入的所有数据，它可以在shell脚本中像其他变量一样使用。

13.6.2 超时

使用read命令时有个危险，就是脚本很可能会等脚本用户的输入一直等下去。如果脚本必须继续执行下去，不管是否有数据输入，你可以用-t选项来指定一个计时器。-t选项指定了read命令等待输入的秒数。当计时器过期后，read命令会返回一个非零退出状态码：

```

$ cat test25
#!/bin/bash
# timing the data entry

if read -t 5 -p "Please enter your name: " name
then
    echo "Hello $name, welcome to my script"
else
    echo
    echo "Sorry, too slow!"
fi
$ ./test25
Please enter your name: Rich
Hello Rich, welcome to my script
$ ./test25
Please enter your name:
Sorry, too slow!
$ 
```

由于计时器过期的话read命令会以非零退出状态码退出，我们可以使用标准的结构化语句，如if-then语句或while循环来轻松地记录发生的情况。在本例中，计时器过期时，if语句不成立，shell会执行else部分的命令。

可以让read命令来对输入的字符计数，而非对输入过程计时。当输入的字符达到预设的字符数时，它会自动退出，将输入的数据赋给变量：

```

$ cat test26
#!/bin/bash
# getting just one character of input

read -n1 -p "Do you want to continue [Y/N]? " answer
case $answer in
Y | y) echo
        echo "fine, continue on...":;
N | n) echo 
```

```

echo OK, goodbye
exit::

esac
echo "This is the end of the script"
$
$ ./test26
Do you want to continue [Y/N]? Y
fine, continue on...
This is the end of the script
$
$ ./test26
Do you want to continue [Y/N]? n
OK, goodbye
$
```

本例中将-n选项和值1一起使用，告诉read命令在接受单个字符后退出。只要你按下单个字符回答后，read命令就会接受输入并将它传给变量，而不必按回车键。

13.6.3 隐藏方式读取

有时你需要读取脚本用户的输入，但不想输入出现在屏幕上。典型的例子是输入的密码，但还有很多其他需要隐藏的数据类型。

-s选项会阻止将传给read命令的数据显示在显示器上（实际上，数据会被显示，只是read命令会将文本颜色设成跟背景色一样）。这里有个在脚本中使用-s选项的例子：

```

$ cat test27
#!/bin/bash
# hiding input data from the monitor

read -s -p "Enter your password: " pass
echo
echo "Is your password really $pass? "
$
$ ./test27
Enter your password:
Is your password really T3sting?
$
```

输入提示符输入的数据不会出现在屏幕上，但会赋给变量，以便在脚本中使用。

13.6.4 从文件中读取

最后，你也可以用read命令来读取Linux系统上文件里保存的数据。每次调用read命令会从文件中读取一行文本。当文件中再没有内容时，read命令会退出并返回非零退出状态码。

其中最难的部分是将文件中的数据传给read命令。最常见的方法是将文件运行cat命令后的输出通过管道直接传给含有read命令的while命令。下面的例子说明怎么处理：

```
$ cat test28
#!/bin/bash
```

```
# reading data from a file

count=1
cat test | while read line
do
    echo "Line $count: $line"
    count=$(( $count + 1 ))
done
echo "Finished processing the file"
$

$ cat test
The quick brown dog jumps over the lazy fox.
This is a test, this is only a test.
O Romeo, Romeo! Wherefore art thou Romeo?
$

$ ./test2B
Line 1: The quick brown dog jumps over the lazy fox.
Line 2: This is a test, this is only a test.
Line 3: O Romeo, Romeo! Wherefore art thou Romeo?
Finished processing the file
$
```

while循环会继续通过read命令处理文件中的行，直到read命令以非零退出状态码退出。

13.7 小结

本章描述了3种不同的方法来从脚本用户获得数据。命令行参数允许用户运行脚本时直接从命令行输入数据。脚本通过位置参数来取回命令行参数并将它们赋给变量。

shift命令可以借助位置参数来轮转命令行参数，从而进行操作。这个命令允许你轻松遍历参数而无需知道到底有多少个参数。

有3个特殊变量可以用来处理命令行参数。shell会将 \$# 变量设为命令行输入的参数总数。\$* 变量会将所有参数都保存为单个字符串。\$@ 变量将所有变量保存为单独的词。这些变量在处理长参数列表时非常有用。

除了参数外，脚本用户可能还会用命令行选项来给脚本传递信息。命令行选项是单个字母，前面加个单破折线。可以给不同的选项赋值，从而改变脚本的行为。bash shell提供了3种方式来处理命令行选项。

第一种方式是将它们像命令行参数一样处理。可以利用位置参数变量来遍历选项，在每个选项出现在命令行上时处理它。

另一种处理命令行选项的方式是用getopt命令。该命令会将命令行选项和参数转换成可以在脚本中处理的标准格式。getopt命令允许你指定哪些字母识别成选项以及哪些选项需要额外的参数值。getopt命令会处理标准的命令行参数并按正确顺序输出选项和参数。

处理命令行选项的最后一一种方法是通过getopts命令（注意是复数）。getopts命令提供了处理命令行参数的高级功能。它支持多个值的参数，以及识别脚本未定义选项的处理。

从脚本用户处获得数据的一种交互方法是read命令。read命令支持脚本向用户提问并等待。

read命令会将脚本用户输入的数据赋给一个或多个变量，你在脚本中可以使用它们。

read命令有一些选项支持定制输入给脚本的数据，比如使用隐藏数据选项、实行超时选项以及要求特定数目输入字符的选项。

下一章，我们会进一步看一下bash shell脚本如何输出数据。到目前为止，你已经学习了如何在屏幕上显示数据，并将它重定向给文件。下一步，我们会探索一些其他选项，不但可以将数据定向到特定位置，还可以将特定类型的数据定向到特定位置。它会让脚本看起来更专业。



本章内容

- 重新考虑重定向
- 标准输入和输出
- 报告错误
- 舍弃数据
- 创建日志文件

到目前为止，本书中出现的脚本都是通过将数据打印在屏幕上或将数据重定向到文件中来显示信息。第10章中演示了如何将命令的输出重定向到文件中。本章将会展开那个主题，演示如何将脚本的输出重定向到Linux系统的不同位置。

14.1 理解输入和输出

到目前为止，你已经知道了两种显示脚本输出的方法：

- 在显示器屏幕上显示输出；
- 将输出重定向到文件中。

这两种方法都会将数据输出全部显示或完全不显示。但有时可能希望在显示器上显示一些数据，其他数据保存到文件中。对于这些情况，了解Linux如何处理输入输出会派上一点用场，这样你就能将脚本输出放到正确位置。

下面几节会介绍如何用标准的Linux输入和输出系统来帮助将脚本输出重定向到特定位置。

14.1.1 标准文件描述符

Linux系统将每个对象当做文件来处理。这包括输入和输出的过程。Linux用文件描述符来标识每个文件对象。文件描述符是一个非负整数，可以唯一地标识会话中打开的文件。每个进程一次最多可以有9个文件描述符。出于特殊目的，bash shell保留了最早的3个文件描述符（0、1和2）。这些在表14-1中显示了。

表14-1 Linux的标准文件描述符

文件描述符	缩写	描述
0	STDIN	标准输入
1	STDOUT	标准输出
2	STDERR	标准错误

这3个特殊文件描述符会处理脚本的输入和输出。shell用它们来将shell默认的输入和输出（默认情况下通常是显示器）定向到相应的位置。下面几节将会进一步介绍这些标准文件描述符。

1. STDIN

STDIN文件描述符代表shell的标准输入。对终端界面来说，标准输入是键盘。shell从STDIN文件描述符对应的键盘获得输入，在用户输入时处理每个字符。

在使用输入重定向符号(<)时，Linux会用重定向指定的文件来替换标准输入文件描述符。它会读取文件并提取数据，就如同它是键盘上键入的。

许多bash命令能接受STDIN的输入，尤其是没有在命令行上指定文件的话。下面是个用cat命令处理STDIN输入的数据的例子：

```
$ cat
this is a test
this is a test
this is a second test.
this is a second test.
```

当你在命令行上只输入cat命令自身时，它会接受STDIN的输入。当你在每行输入时，cat命令会将每行显示在输出中。

但你也可以通过STDIN重定向符号强制cat命令接受来自另一个非STDIN文件的输入：

```
$ cat < testfile
This is the first line.
This is the second line.
This is the third line.
$
```

现在cat命令会用testfile文件中的行作为输入。你可以使用这种技术来向任何能从STDIN接受数据的shell命令输入数据。

2. STDOUT

STDOUT文件描述符代表标准的shell输出。在终端界面上，标准输出就是终端显示器。shell的所有输出（包括shell中运行的程序和脚本）会被定向到标准输出中，也就是显示器。

默认情况下，很多bash命令会定向输出到STDOUT文件描述符。如第10章中所述，你可以用输出重定向来改变：

```
$ ls -l > test2
$ cat test2
total 20
-rw-rw-r-- 1 rich rich 53 2010-10-16 11:30 test
```

```
-rw-rw-r-- 1 rich rich 0 2010-10-16 11:32 test2
-rw-rw-r-- 1 rich rich 73 2010-10-16 11:23 testfile
$
```

通过输出重定向符号，通常会显示到显示器的所有输出会被shell重定向到指定的重定向文件。你也可以将数据追加到某个文件。可以用>>符号来完成：

```
$ who >> test2
$ cat test2
total 20
-rw-rw-r-- 1 rich rich 53 2010-10-16 11:30 test
-rw-rw-r-- 1 rich rich 0 2010-10-16 11:32 test2
-rw-rw-r-- 1 rich rich 73 2010-10-16 11:23 testfile
rich      pts/0        2010-10-17 15:34 (192.168.1.2)
$
```

who命令生成的输出会被追加到test2文件中以后的数据后面。

但是，如果你对文件用了标准的输出重定向，你可能会遇到一个问题。下面的例子说明了可能会上出现什么情况：

```
$ ls -al badfile > test3
ls: cannot access badfile: No such file or directory
$ cat test3
$
```

当命令生成错误消息时，shell并未将错误消息重定向到输出重定向文件。shell创建了输出重定向文件，但错误消息却显示在了显示器屏幕上。注意，在显示test3文件的内容时并没有任何错误。test3文件创建成功了，只是里面是空的。

shell处理错误消息是跟处理普通输出分开的。如果你创建了在后台模式下运行的shell脚本，通常你必须依赖发送到日志文件的输出消息。用这种方法，它们就没法保存在日志文件中。你需要换种方法来处理。

3. STDERR

shell通过特殊的STDERR文件描述符来处理错误消息。STDERR文件描述符代表shell的标准错误输出。shell或shell中运行的程序和脚本出错时生成的错误消息都会发送到这个位置。

默认情况下，STDOUT文件描述符会和STDOUT文件描述符指向同样的地方（尽管分配给它们的文件描述符值不同）。也就是说，默认情况下，错误消息也会输出到显示器输出中。

但从上面的例子可以看出，重定向STDOUT并不会自动重定向STDERR。处理脚本时，你常常会想改变这种行为，如果你想将错误消息保存到日志文件中时尤其如此。

14.1.2 重定向错误

你已经知道如何用重定向符号来重定向STDOUT数据。成功定向STDERR数据也没太大差别，你只要是在使用重定向符号时定义STDERR文件描述符就可以了。有几种办法来处理。

1. 只重定向错误

如你在表14-1中看到的，STDERR文件描述符被设成2。你可以选择只重定向错误消息，将该

文件描述符值放在重定向符号前。该值必须紧紧地放在重定向符号前，否则不会工作：

```
$ ls -al badfile 2> test4
$ cat test4
ls: cannot access badfile: No such file or directory
$
```

现在运行该命令，错误消息不会出现在屏幕上。而是，该命令生成的任何错误消息都会保存在输出文件中。用这种方法，shell会只重定向错误消息，而非普通数据。这里有个在同一输出中混用STDOUT和STDERR的例子：

```
$ ls -al test badtest test2 2> test5
-rw-rw-r-- 1 rich rich 158 2010-10-16 11:32 test2
$ cat test5
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
$
```

`ls`命令的正常STDOUT输出仍然会发送到默认的STDOUT文件描述符，也就是显示器。由于该命令重定向了文件描述符2的输出（STDERR）到了一个输出文件，shell会将生成的任何错误消息直接发送到指定的重定向文件中。

2. 重定向错误和数据

如果你想重定向错误和正常输出，你必须用两个重定向符号。你需要在想要重定向的每个数据前添加对应的文件描述符，并将它们指向对应的保存数据的输出文件：

```
$ ls -al test test2 test3 badtest 2> test6 1> test7
$ cat test6
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
$ cat test7
-rw-rw-r-- 1 rich rich 158 2010-10-16 11:32 test2
-rw-rw-r-- 1 rich rich    0 2010-10-16 11:33 test3
$
```

shell利用`1>`符号将`ls`命令的本该输出到STDOUT的正常输出重定向到了`test7`文件。任何本该输出到STDERR的错误消息通过`2>`符号被重定向到了`test6`文件。

你可以用这种方法来将脚本的正常输出和脚本生成的错误消息分离开来。它允许你轻松地识别错误，而不用在成千上万行正常输出数据中艰难查找。

另外，如果愿意，你也可以将STDERR和STDOUT的输出重定向到同一个输出文件。为此bash shell提供了特殊的重定向符号`&>`：

```
$ ls -al test test2 test3 badtest &> test7
$ cat test7
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
-rw-rw-r-- 1 rich rich 158 2010-10-16 11:32 test2
-rw-rw-r-- 1 rich rich    0 2010-10-16 11:33 test3
$
```

当使用`&>`符时，命令生成的所有输出都会发送到同一位置，包括数据和错误。你会注意到错误消息中有一条跟你期望的顺序不同。这条针对`badtest`文件的错误消息（列出的最后一个文件）出现在输出文件中的第二行。`bash shell`会自动给错误消息分配较标准输出来说更高的优先级。这样你就能在一处地方查看错误消息了，不用翻遍整个输出文件。

14.2 在脚本中重定向输出

你可以在脚本中用`STDOUT`和`STDERR`文件描述符来在多个位置生成输出，只要简单地重定向相应的文件描述符。有两种方法来在脚本中重定向输出：

- 临时重定向每行输出；
- 永久重定向脚本中的所有命令。

14.2.1 临时重定向

如果你要故意在脚本中生成错误消息，可以将单独的一行输出重定向到`STDERR`。你所需要做的是使用输出重定向符来将输出重定向到`STDERR`文件描述符。在重定向到文件描述符时，你必须在文件描述符数字之前加一个`and`符（`&`）：

```
echo "This is an error message" >&2
```

这行会将文本显示在脚本的`STDERR`文件描述符所指向的任何位置，而不是通常的`STDOUT`。下面是个用此特性的脚本例子：

```
$ cat test8
#!/bin/bash
# testing STDERR messages

echo "This is an error" >&2
echo "This is normal output"
$
```

正常运行这个脚本，你可能看不出什么区别：

```
$ ./test8
This is an error
This is normal output
$
```

记住，默认情况下Linux会将`STDERR`定向到`STDOUT`。但是，如果你在运行脚本时重定向了`STDERR`，脚本中所有定向到`STDERR`的文本都会被重定向：

```
$ ./test8 2> test9
This is normal output
$ cat test9
This is an error
$
```

太好了！通过`STDOUT`显示的文本显示在了屏幕上，而发送给`STDERR`的`echo`语句的文本则被重

定向到了输出文件。

这个方法非常适合在脚本中生成错误消息。如果有人用了你的脚本，他们可以轻松地通过STDERR文件描述符重定向错误消息，如上所示。

14.2.2 永久重定向

如果脚本中有大量数据需要重定向，那重定向每个echo语句就会很烦琐。取而代之，你可以用exec命令告诉shell在脚本执行期间重定向某个特定文件描述符：

```
$ cat test10
#!/bin/bash
# redirecting all output to a file
exec 1>testout

echo "This is a test of redirecting all output"
echo "from a script to another file."
echo "without having to redirect every individual line"
$ ./test10
$ cat testout
This is a test of redirecting all output
from a script to another file.
without having to redirect every individual line
$
```

exec命令会启动一个新shell并将STDOUT文件描述符重定向到文件。脚本中发给STDOUT的所有输出会被重定向到文件。

你可以在脚本中间重定向STDOUT：

```
$ cat test11
#!/bin/bash
# redirecting output to different locations

exec 2>testerror

echo "This is the start of the script"
echo "now redirecting all output to another location"

exec 1>testout

echo "This output should go to the testout file"
echo "but this should go to the testerror file" >&2
$

$ ./test11
This is the start of the script
now redirecting all output to another location
$ cat testout
This output should go to the testout file
$ cat testerror
but this should go to the testerror file
$
```

这个脚本用exec命令来将发给STDERR的输出重定向到文件testerror。下一步，脚本用echo

语句显示了一些行到STDOUT。那之后，再次使用exec命令来将STDOUT重定向到testout文件。注意，尽管STDOUT被重定向了，你仍然可以指定将echo语句的输出发给STDERR，在本例中仍然是重定向到testerror文件。

这个特性在你要将脚本的部分输出重定向到另一个位置时能派上用场，比如错误日志。在使用该特性时你会碰到个小问题。

一旦你重定向了STDOUT或STDERR，你就无法轻易将它们重定向回原来的位置。你需要在重定向中来回切换的话，有个办法可以用。14.4节将会讨论该方法以及如何在脚本中使用。

14.3 在脚本中重定向输入

你可以在脚本中重定向STDOUT和STDERR的同样方法来将STDIN从键盘重定向到其他位置。exec命令允许你将STDIN重定向到Linux系统上的文件中：

```
exec 0< testfile
```

这个命令会告诉shell它应该从文件testfile中获得输入，而不是STDIN。这个重定向只要在脚本需要输入时就会作用。下面是该用法的实例：

```
$ cat test12
#!/bin/bash
# redirecting file input

exec 0< testfile
count=1

while read line
do
    echo "Line #$count: $line"
    count=$((count + 1))
done
$ ./test12
Line #1: This is the first line.
Line #2: This is the second line.
Line #3: This is the third line.
$
```

第13章介绍了如何使用read命令来读取用户在键盘上输入的数据。将STDIN重定向到文件后，当read命令试图从STDIN读入数据时，它会到文件去取数据，而不是键盘。

这是在脚本中从要处理的文件中读取数据的绝妙办法。Linux系统管理员的一项日常任务就是从要处理的日志文件中读取数据。这是完成该任务最简单的办法。

14.4 创建自己的重定向

在脚本中重定向输入和输出时，并不局限于这3个默认的文件描述符。我曾提到过，在shell中最多可以有9个打开的文件描述符。其他6个文件描述符会从3排到8，并且当做输入或输出重定

向都行。你可以将这些文件描述符中的任意一个分配给文件，然后在脚本中使用它们。本节将介绍如何在脚本中使用其他文件描述符。

14.4.1 创建输出文件描述符

你可以用exec命令来给输出分配文件描述符。和标准的文件描述符一样，一旦你给一个文件位置分配了另外一个文件描述符，那个重定向就会一直有效，直到你重新分配。这里有个在脚本中使用其他文件描述符的简单例子：

```
$ cat test13
#!/bin/bash
# using an alternative file descriptor

exec 3>test13out

echo "This should display on the monitor"
echo "and this should be stored in the file" >&3
echo "Then this should be back on the monitor"
$ ./test13
This should display on the monitor
Then this should be back on the monitor
$ cat test13out
and this should be stored in the file
$
```

这个脚本用exec命令来将文件描述符3重定向到另一个文件位置。当脚本执行echo语句时，它们如你所期望的那样，显示在STDOUT上。但你重定向到文件描述符3的那行echo语句输出到了另外那个文件。它允许你在显示器上保持正常的输出，而将特定信息重定向到文件中，比如日志文件。

你也可以使用exec命令来将输出追加到现有文件中，而不是创建一个新文件：

```
exec 3>>test13out
```

现在输出会被追加到test13out文件，而不是创建一个新文件。

14.4.2 重定向文件描述符

现在介绍怎么从已重定向的文件描述符中恢复。你可以分配另外一个文件描述符给标准文件描述符，反之亦然。这意味着你可以重定向STDOUT的原来位置到另一个文件描述符，然后将该文件描述符重定向回STDOUT。这听起来可能有点复杂，但实际上相当直接。这个简单的例子能帮你理清楚：

```
$ cat test14
#!/bin/bash
# storing STDOUT, then coming back to it

exec 3>&1
exec 1>test14out
```

```

echo "This should store in the output file"
echo "along with this line."
exec 1>&3

echo "Now things should be back to normal"
$ ./test14
Now things should be back to normal
$ cat test14out
This should store in the output file
along with this line.
$
```

这个例子有点叫人抓狂，我们来一段一段地看。首先，脚本将文件描述符3重定向到文件描述符1的当前位置，也就是STDOUT。这意味着任何发送给文件描述符3的输出都将出现在显示器上。

第二个exec命令将STDOUT重定向到文件，shell现在会将发送给STDOUT的输出直接重定向到输出文件中。但是，文件描述符3仍然指向STDOUT原来的位置，也就是显示器。如果此时将输出数据发送给文件描述符3，它仍然会出现在显示器上，尽管STDOUT已经被重定向过。

在向STDOUT（现在指向一个文件）发送一些输出之后，脚本会将STDOUT重定向到文件描述符3的当前位置（仍然是设置到显示器）。这意味着现在STDOUT指向了它原来的位置，也就是显示器。

这个方法可能有点叫人困惑，但它是在脚本中临时将输出重定向然后再将输出恢复到通常设置的通用办法。

14.4.3 创建输入文件描述符

你可以用和重定向输出文件描述符同样的办法来重定向输入文件描述符。在重定向到文件之前，先将STDIN文件描述符保存到另外一个文件描述符，然后在读取完文件之后再将STDIN恢复到它原来的位置：

```

$ cat test15
#!/bin/bash
# redirecting input file descriptors

exec 6<&0

exec 0< testfile

count=1
while read line
do
    echo "Line #$count: $line"
    count=$(( $count + 1 ))
done
exec 0<&6
read -p "Are you done now? " answer
```

```

case $answer in
Y|y) echo "Goodbye":;
N|n) echo "Sorry, this is the end.";;
esac
$ ./test15
Line #1: This is the first line.
Line #2: This is the second line.
Line #3: This is the third line.
Are you done now? y
Goodbye
$
```

在这个例子中，文件描述符6用来保存STDIN的位置。然后脚本将STDIN重定向到一个文件。read命令的所有输入都是从重定向后的STDIN中来的，也就是输入文件。

在读取了所有行之后，脚本会将STDIN重定向到文件描述符6，从而将STDIN恢复到原先的位置。该脚本用了另外一个read命令来测试STDIN是否恢复正常了。这次它会等待键盘的输入。

14.4.4 创建读写文件描述符

尽管看起来可能会很奇怪，你也可以打开单个文件描述符来作为输入和输出。你可以用同一个文件描述符来从文件中读取数据，并将数据写到同一个文件中。

但用这种方法时，你要特别小心。由于你在向同一个文件进行读取数据、写入数据操作，shell会维护一个内部指针，指明现在在文件中什么位置。任何读或写都会从文件指针上次保存的位置开始。如果你不够小心，它会产生一些有意思的结果。看看下面这个例子：

```

$ cat test16
#!/bin/bash
# testing input/output file descriptor

exec 3<> testfile
read line <&3
echo "Read: $line"
echo "This is a test line" >&3
$ cat testfile
This is the first line.
This is the second line.
This is the third line.
$ ./test16
Read: This is the first line.
$ cat testfile
This is the first line.
This is a test line
ine.
This is the third line.
$
```

这个例子用了exec命令来将用作读取输入和写入输出的文件描述符3分配给文件testfile。下一步，它通过分配好的文件描述符来用read命令读取文件中的第一行，然后将读取的那行输入

显示在STDOUT上。之后，它用echo语句来向通过文件描述符打开的文件写入一行数据。

在运行脚本时，最开始事情看起来都还正常。输出说明脚本读了testfile文件中的第一行。但在运行脚本后，显示testfile文件的内容，你会看到写到文件中的数据覆盖了已有的数据。

当脚本向文件中写入数据时，它会从文件指针所处的位置开始。read命令读取了第一行数据，所以它会让文件指针指向第二行数据的第一个字符。在echo语句将数据输出到文件时，它会将数据放在文件指针的当前位置，覆盖该位置的任何数据。

14.4.5 关闭文件描述符

如果你创建了新的输入或输出文件描述符，shell会在脚本退出时自动关闭它们。然而还有一些情况，你需要在脚本结束前手动关闭文件描述符。

要关闭文件描述符，将它重定向到特殊符号&-。脚本中看起来如下：

```
exec 3>&-
```

该语句会关闭文件描述符3，从而阻止在脚本中再使用它。这里有个例子说明当你尝试使用已关闭的文件描述符时会怎样：

```
$ cat badtest
#!/bin/bash
# testing closing file descriptors

exec 3>test17file

echo "This is a test line of data" >&3

exec 3>&-

echo "This won't work" >&3
$ ./badtest
./badtest: 3: Bad file descriptor
$
```

一旦关闭了文件描述符，你就不能在脚本中向它写入任何数据了，否则shell会生成错误消息。

在关闭文件描述符时还要注意另一件事。如果后面你在脚本中打开了同一个输出文件，shell会用一个新文件来替换已有文件。这意味着如果你要输出任何数据，它将会覆盖已有文件。考虑下面这个问题的例子：

```
$ cat test17
#!/bin/bash
# testing closing file descriptors

exec 3> test17file
echo "This is a test line of data" >&3
exec 3>&-

cat test17file
```

```

exec 3> test17file
echo "This'll be bad" >&3
$ ./test17
This is a test line of data
$ cat test17file
This'll be bad
$ 
```

在向test17file文件发送一个数据字符串并关闭该文件描述符之后，脚本用了cat命令来显示文件的内容。到目前为止，一切都还好。下一步，脚本重新打开了该输出文件并向它发送了另一个数据字符串。当你显示该输出文件的内容时，你所能看到的只有第二个数据字符串。shell覆盖了原来的输出文件。

14.5 列出打开的文件描述符

对你来说，只有9个文件描述符可用，你可能会觉得要让事情简单直接并不难。但有时要记住哪个文件描述符被重定向到了哪里很难。为了便于理清楚，bash shell提供了lsof命令。

lsof命令会列出整个Linux系统打开的所有文件描述符。这是个有争议的功能，因为它会向非系统管理员用户提供Linux系统的信息。鉴于此，许多Linux系统隐藏了该命令，这样用户就不会一不小心就发现了。

在我的Fedora Linux系统上，lsof命令位于/usr/sbin目录。要用普通用户账户来运行它，我必须通过全路径名来引用它：

```
$ /usr/sbin/lsof
```

它会产生大量的输出。它会显示当前Linux系统上打开的每个文件的有关信息。这包括后台运行的所有进程以及登录到系统的任何用户。

有足够的命令行选项和参数帮助过滤lsof的输出。最常用的有-p和-d，前者允许指定进程ID（PID），后者允许指定要显示的文件描述符个数。

要知道该进程的当前PID，你可以用特殊环境变量\$\$（shell会将它设为当前PID）。-a选项用来对其他两个选项的结果执行布尔AND运算，产生如下输出：

```

$ /usr/sbin/lsof -a -p $$ -d 0,1,2
COMMAND PID USER FD TYPE DEVICE SIZE NODE NAME
bash 3344 rich 0u CHR 136,0 2 /dev/pts/0
bash 3344 rich 1u CHR 136,0 2 /dev/pts/0
bash 3344 rich 2u CHR 136,0 2 /dev/pts/0
$ 
```

上例显示了当前进程（bash shell）的默认文件描述符。lsof的默认输出中有7列信息，见表14-2。

表14-2 lsof的默认输出

列	描述
COMMAND	正在运行的命令名的前9个字符
PID	进程的PID
USER	进程属主的登录名
FD	文件描述符数目以及访问类型 (r代表读, w代表写, u代表读写)
TYPE	文件的类型 (CHR代表字符型, BLK代表块型, DIR代表目录, REG代表常规文件)
DEVICE	设备的设备号 (主设备号和从设备号)
SIZE	如果有的话, 文件的大小
NODE	本地文件的节点数
NAME	文件名

与STDIN、STDOUT和STDERR关联的文件类型是字符型。因为STDIN、STDOUT和STDERR文件描述符都指向终端, 所以输出文件的名称就是终端的设备名。所有3种标准文件都支持读和写(尽管向STDIN写数据以及从STDOUT读数据看起来有点奇怪)。

现在, 我们看一下打开了多个替代性文件描述符的脚本中lsof命令的结果:

```
$ cat test18
#!/bin/bash
# testing lsof with file descriptors

exec 3> test18file1
exec 6> test18file2
exec 7< testfile

/usr/sbin/lsof -a -p $$ -d0,1,2,3,6,7
$ ./test18
COMMAND PID USER FD   TYPE DEVICE SIZE    NODE NAME
test18 3594 rich  0u    CHR  136,0          2 /dev/pts/0
test18 3594 rich  1u    CHR  136,0          2 /dev/pts/0
est18 3594 rich  2u    CHR  136,0          2 /dev/pts/0
18 3594 rich   3w    REG  253,0      0 360712 /home/rich/test18file1
18 3594 rich   6w    REG  253,0      0 360715 /home/rich/test18file2
18 3594 rich   7r    REG  253,0      73 360717 /home/rich/testfile
$
```

该脚本创建了3个替代性文件描述符, 两个作为输出(3和6), 一个作为输入(7)。在脚本运行lsof命令时, 你可以在输出中看到新的文件描述符。我们去掉了输出中的第一部分, 这样你就能看到文件名的结果了。文件名显示了文件描述符中采用的文件的完整路径名。它将每个文件都显示成REG类型的, 说明它们是文件系统中的常规文件。

14.6 阻止命令输出

有时候你不想显示脚本的输出。这在将脚本作为后台进程执行时很常见(参见第15章)。如

果在脚本后台运行时有任何错误消息出现，shell会通过电子邮件将它们发给进程的属主。这会很麻烦，尤其是在你运行会生成很多烦琐的小错误的脚本时。

要解决这个问题，你可以将STDERR重定向到一个称作null文件的特殊文件。null文件跟它的名字很像，文件里什么都没有。shell输出到null文件的任何数据都不会保存，这样它们就都被丢掉了。

在Linux系统上null文件的标准位置是/dev/null。你重定向到该位置的任何数据都会被丢掉，不会显示：

```
$ ls -al > /dev/null  
$ cat /dev/null  
$
```

这是阻止任何错误消息而不保存它们的一个通用方法：

```
$ ls -al badfile test16 2> /dev/null  
-rwxr--r-- 1 rich rich 135 Oct 29 19:57 test16*
```

你也可以在输入重定向将/dev/null作为输入文件。由于/dev/null文件不含有任何内容，程序员通常用它来快速移除现有文件中的数据而不用先删除文件再创建：

```
$ cat testfile  
This is the first line.  
This is the second line.  
This is the third line.  
$ cat /dev/null > testfile  
$ cat testfile  
$
```

文件testfile仍然存在系统上，但现在它是空文件。这是清除日志文件的一个通用方法，它们必须留在原来位置等待应用写入。

14.7 创建临时文件

Linux系统有一个特殊的留给临时文件用的目录位置。Linux使用/tmp目录来存放不需要一直保留的文件。大多数Linux发行版配置了系统在启动时自动删除/tmp目录的所有文件。

系统上的任何用户账户都有权限在/tmp目录中读和写。这个特性为你提供了简单地创建临时文件的途径，而不用管清理工作。

有个特殊命令可以用来创建临时文件。mktemp命令可以轻松地在/tmp目录中创建一个唯一的临时文件。shell会创建这个文件，但不用默认的umask值（参见第6章）。它会将文件的读和写权限分配给文件的属主，并将你设成文件的属主。一旦创建了文件，你就在脚本中有了完整的读写权限，但其他人没法访问它（当然，root用户除外）。

14.7.1 创建本地临时文件

默认情况下，mktemp会在本地目录中创建一个文件。要用mktemp命令在本地目录中创建一个临时文件，你只要指定一个文件名模板就行了。模板可以包含任意文本文件名，在文件名末尾加

上6个X就行了：

```
$ mktemp testing.XXXXXX
$ ls -al testing*
-rw----- 1 rich      rich      0 Oct 17 21:30 testing.UfIi13
$
```

`mktemp`命令会用6个字符码替换这6个X，从而保证文件名在目录中是唯一的。你可以创建多个临时文件，它可以保证每个文件都是唯一的：

```
$ mktemp testing.XXXXXX
testing.1DRLuV
$ mktemp testing.XXXXXX
testing.TVBtkW
$ mktemp testing.XXXXXX
testing.PggNKG
$ ls -l testing*
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.1DRLuV
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.PggNKG
-rw----- 1 rich      rich      0 Oct 17 21:30 testing.UfIi13
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.TVBtkW
$
```

如你所看到的，`mktemp`命令的输出正是它所创建的文件的名字。在脚本中使用`mktemp`命令时，你可能要将文件名保存到变量中，这样你就能在后面的脚本中引用了：

```
$ cat test19
#!/bin/bash
# creating and using a temp file

 tempfile=`mktemp test19.XXXXXX`

 exec 3>$tempfile

 echo "This script writes to temp file $tempfile"

 echo "This is the first line" >&3
 echo "This is the second line." >&3
 echo "This is the last line." >&3
 exec 3>-

 echo "Done creating temp file. The contents are:"
 cat $tempfile
 rm -f $tempfile 2> /dev/null
 $ ./test19
 This script writes to temp file test19.vChoya
 Done creating temp file. The contents are:
 This is the first line
 This is the second line.
 This is the last line.
 $ ls -al test19*
 -rwxr--r-- 1 rich      rich      356 Oct 29 22:03 test19*
 $
```

这个脚本用mktemp命令来创建临时文件并将文件名赋给\$tempfile变量。然后它将这个临时文件作为文件描述符3的输出重定向文件。在将临时文件名显示在STDOUT之后，它会将一些行写到临时文件，然后它会显示临时文件的内容，并用rm命令删除。

14.7.2 在/tmp目录创建临时文件

-t选项会强制mktemp命令来在系统的临时目录来创建该文件。在用这个特性时，mktemp命令会返回用来创建临时文件的全路径，而不是只有文件名：

```
$ mktemp -t test.XXXXXX
/tmp/test.xG3374
$ ls -al /tmp/test*
-rw----- 1 rich rich 0 2010-10-29 18:41 /tmp/test.xG3374
$
```

由于mktemp命令返回了全路径名，你就可以在Linux系统上的任何目录下引用该临时文件，而不用管它将临时文件放在了什么位置：

```
$ cat test20
#!/bin/bash
# creating a temp file in /tmp

$tempfile=`mktemp -t tmp.XXXXXX` 

echo "This is a test file." > $tempfile
echo "This is the second line of the test." >> $tempfile

echo "The temp file is located at: $tempfile"
cat $tempfile
rm -f $tempfile
$ ./test20
The temp file is located at: /tmp/tmp.Ma3390
This is a test file.
This is the second line of the test.
$
```

在mktemp创建临时文件时，它会将全路径名返回给变量。这样你就能在任何命令中使用该值来引用该临时文件了。

14.7.3 创建临时目录

-d选项告诉mktemp命令来创建一个临时目录而不是临时文件。这样你就能用该目录做想做任何需要的操作了，比如创建额外的临时文件：

```
$ cat test21
#!/bin/bash
# using a temporary directory

$tempdir=`mktemp -d dir.XXXXXX`
cd $tempdir
```

```

tempfile1=`mktemp temp.XXXXXX`
tempfile2=`mktemp temp.XXXXXX`
exec 7> $tempfile1
exec 8> $tempfile2

echo "Sending data to directory $tempdir"
echo "This is a test line of data for $tempfile1" >&7
echo "This is a test line of data for $tempfile2" >&8
$ ./test21
Sending data to directory dir.outT8S8
$ ls -al
total 72
drwxr-xr-x  3 rich   rich        4096 Oct 17 22:20 .
drwxr-xr-x  9 rich   rich        4096 Oct 17 09:44 ..
drwxr-xr-x  2 rich   rich        4096 Oct 17 22:20 dir.outT8S8/
-rw-r--r--  1 rich   rich       338 Oct 17 22:20 test21*
$ cd dir.outT8S8
[dir.outT8S8]$ ls -al
total 16
drwx-----  2 rich   rich        4096 Oct 17 22:20 .
drwxr-xr-x  3 rich   rich        4096 Oct 17 22:20 ..
-rw-----  1 rich   rich        44 Oct 17 22:20 temp.N5F306
-rw-----  1 rich   rich        44 Oct 17 22:20 temp.SQs1b7
[dir.outT8S8]$ cat temp.N5F306
This is a test line of data for temp.N5F306
[dir.outT8S8]$ cat temp.SQs1b7
This is a test line of data for temp.SQs1b7
[dir.outT8S8]$

```

这段脚本在当前目录创建了一个目录，然后它用cd命令跳进该目录，并创建了两个临时文件。之后这两个临时文件被分配给文件描述符用来存储脚本的输出。

14.8 记录消息

有时将输出一边发送到显示器一边发送到日志文件，这会有一些好处。你不用将输出重定向两次，只要用特殊的tee命令就行。

tee命令相当于管道的一个T型接头。它将从STDIN过来的数据同时发给两个目的地。一个目的地是STDOUT，另一个目的地是tee命令行所指定的文件名：

```
tee filename
```

由于tee会将从STDIN过来的数据重定向，你可以用它和管道命令来一起将任何命令的输出重定向：

```

$ date | tee testfile
Sun Oct 17 18:56:21 EDT 2010
$ cat testfile
Sun Oct 17 18:56:21 EDT 2010
$
```

输出出现在了STDOUT中，并且也写入了指定的文件中。注意，默认情况下，tee命令会在每

次使用时覆盖输出文件内容：

```
$ who | tee testfile
rich pts/0 2010-10-17 18:41 (192.168.1.2)
$ cat testfile
rich pts/0 2010-10-17 18:41 (192.168.1.2)
$
```

如果你想将数据追加到文件中，必须用-a选项：

```
$ date | tee -a testfile
Sun Oct 17 18:58:05 EDT 2010
$ cat testfile
rich pts/0 2010-10-17 18:41 (192.168.1.2)
Sun Oct 17 18:58:05 EDT 2010
$
```

利用这个方法，你就既能将数据保存在文件中，也能将数据显示在屏幕上上了：

```
$ cat test22
#!/bin/bash
# using the tee command for logging

tempfile=test22file

echo "This is the start of the test" | tee $tempfile
echo "This is the second line of the test" | tee -a $tempfile
echo "This is the end of the test" | tee -a $tempfile
$ ./test22
This is the start of the test
This is the second line of the test
This is the end of the test
$ cat test22file
This is the start of the test
This is the second line of the test
This is the end of the test
$
```

现在你就能在将输出显示给用户的同时也永久保留一份了。

14.9 小结

理解bash shell如何处理输入和输出在创建脚本时很有用。你可以改变脚本获取数据以及显示数据的方式，从而在任何环境中定制脚本。脚本的输入可以从标准输入（STDIN）重定向到系统上的任意文件上。脚本的输出也可以从标准输出（STDOUT）重定向到系统上的任意文件中。

除了STDOUT，你可以通过重定向STDERR输出来重定向由脚本产生的错误消息。这可以通过重定向与STDERR输出关联的文件描述符（也就是文件描述符2）来实现。你可以将STDERR输出和STDOUT输出到同一个文件中，也可以输出到完全不同的文件中。这使得你可以将正常的脚本消息同脚本产生的错误消息分离开。

bash shell允许在脚本中创建自己的文件描述符。你可以创建文件描述符3~9，并将它们分配

给要用到的任何输出文件。一旦创建了文件描述符，你就可以利用标准的重定向符号将任意命令的输出重定向过去。

`bash shell`也允许将输入重定向到一个文件描述符，这提供了将文件的数据读入到脚本中的一个简便途径。你可以用`lsof`命令来显示`shell`中在用的文件描述符。

`Linux`系统提供了一个特殊的文件（称作`/dev/null`）来重定向不要的输出。`Linux`系统会删掉任何重定向到`/dev/null`文件的东西。你也可以用此文件来产生一个空文件，通过将`/dev/null`文件的内容重定向到该文件。

`mktemp`命令是`bash shell`很有用的一个特性，可以轻松地创建临时文件和目录。简单地给`mktemp`命令指定一个模板，它就能在每次调用时基于文件模板的格式创建一个唯一的文件。你也可以在`Linux`系统的`/tmp`目录创建临时文件和目录，系统启动时会清空该特殊位置的内容。

`tee`命令便于将输出同时发送给标准输出和日志文件。这使得你可以既在显示器上显示脚本的消息，又能同时在日志文件中保存。

在第15章中，你将了解如何控制和运行脚本。`Linux`提供了几种不同的方法来运行脚本，除了直接在命令行提示符里运行之外。你还将了解如何在特定时间运行脚本，以及在脚本运行时如何暂停。

本章内容

- 处理信号
- 以后台模式运行脚本
- 在非控制台下运行脚本
- 作业控制
- 调整谦让度
- 定时运行作业
- 启动时运行
- 文件系统命令

当 你开始构建高级脚本时，你大概会问如何在Linux系统上运行和控制它们。在本书中，到目前为止，我们运行脚本的唯一方式就是以实时模式在命令行界面上直接运行。这并不是Linux上运行脚本的唯一方式，还有其他一些运行shell脚本的选择。本章将会介绍运行脚本的不同方式。还有，有时脚本会陷入循环中，这时你需要找到令它停止而不用关掉整个Linux系统的方法。本章会介绍控制何时以及如何在系统上运行shell脚本的不同方法。

15.1 处理信号

Linux通过信号来在运行在系统上的进程之间通信。第4章介绍了不同的Linux信号以及Linux如何用这些信号来停止、启动以及无条件终止进程。你也可以用这些信号来控制shell脚本的运行，通过对脚本进行编程使其在收到Linux系统的特定信号时执行特定的命令。

15.1.1 重温Linux信号

Linux系统和应用程序可以生成超过30个信号。表15-1列出了在Linux编程时可能会遇到的最常见的Linux系统信号。

表15-1 Linux信号

信 号	值	描 述
1	SIGHUP	挂起进程
2	SIGINT	终止进程
3	SIGQUIT	停止进程
9	SIGKILL	无条件终止进程
15	SIGTERM	可能的话终止进程
17	SIGSTOP	无条件停止进程，但不是终止进程
18	SIGTSTP	停止或暂停进程，但不终止进程
19	SIGCONT	继续运行停止的进程

默认情况下，bash shell会忽略收到的任何SIGQUIT（3）和SIGTERM（5）信号（正因为这样，交互式shell才不会被意外终止）。但是bash shell会处理收到的SIGHUP（1）和SIGINT（2）信号。

如果bash shell收到了SIGHUP信号，它会退出。但在退出之前，它会将SIGHUP信号传给shell启动的所有进程（比如shell脚本）。通过SIGINT信号，可以中断shell。Linux内核会停止将CPU的处理时间分配给shell。当这种情况发生时，shell会将SIGINT信号传给shell启动的所有进程，来通知它们。

shell会将这些信号传给shell脚本程序来处理。而shell脚本的默认行为是忽略这些信号。它们可能会不利于脚本的运行。要避免这种情况，你可以脚本中加入识别信号和处理信号结果的命令。

15.1.2 产生信号

bash shell允许用键盘上的键组合生成两种基本的Linux信号。这个特性在需要停止或暂停跑飞了的程序时就能派上用场。

1. 终止进程

Ctrl+C组合键会生成SIGINT信号，并将其发送给shell中当前运行的所有进程。可以运行一条需要很长时间才能完成的命令，然后按下Ctrl+C组合键来测试它：

```
$ sleep 100
^C
$
```

Ctrl+C组合键会停止shell中当前运行的进程。sleep命令会暂停指定的秒数。通常，命令提示符直到计时器超时才会返回。在计时器超时前按下Ctrl+C组合键，你就会提前终止sleep命令。

2. 暂停进程

你可以在进程运行中间暂停它，而不用终止它。尽管有时这可能会比较危险（比如，脚本打开了一个关键的系统文件的文件锁），但通常它允许你一窥究竟，而不用实际上终止进程。

Ctrl+Z组合键会生成一个SIGTSTP信号，停止shell中运行的任何进程。停止一个进程跟终止进

程不同，停止进程会让程序继续保留在内存中，并能从上次停止的位置继续运行。在15.4节，你会了解如何重启一个已经停止的进程。

当你用Ctrl+Z组合键时，shell会通知你进程已经被停止了：

```
$ sleep 100
^Z
[1]+  Stopped                  sleep 100
$
```

方括号中的数字是shell分配的作业号。shell将shell中运行的每个进程称为作业，并为每个作业分配一个唯一的作业号。它会给第一个作业分配作业号1，第二个作业号2，以此类推。

如果你有个分配到shell会话的停止了的作业，在退出shell时，bash会提醒你：

```
$ exit
logout
There are stopped jobs.
$
```

你可以用ps命令来查看已停止的作业：

```
$ ps au
USER PID  %CPU %MEM   VSZ RSS TTY STAT START   TIME COMMAND
rich 20560  0.0  1.2  2688 1624 pts/0   S    05:15  0:00 -bash
rich 20605  0.2  0.4  1564  552 pts/0   T    05:22  0:00 sleep 100
rich 20606  0.0  0.5  2584  740 pts/0   R    05:22  0:00 ps au
$
```

在STAT一列中，ps命令显示了已停止作业的状态为T。这说明命令正在被跟踪或被停止了。

如果你真的想退出shell，尽管已停止的作业仍然在活动，只要再输入一遍exit命令就行了。shell会退出，终止已停止作业。另外，既然你已经知道了已停止作业的PID，你可以用kill命令来发送一个SIGKILL信号来终止它：

```
$ kill -9 20605
$
[1]+  Killed                  sleep 100
$
```

在你无条件终止作业时，最开始你不会得到任何反应。但下次有shell提示符时，你会看到一条消息说明作业已经被无条件终止了。每当shell产生一个提示符时，它就会显示shell中已经改变状态的作业的状态。在你无条件终止一个作业后，下次强制shell生成一个提示符时，shell会显示一条消息说明作业在运行时被无条件终止了。

15.1.3 捕捉信号

你也可以不忽略信号，在信号出现时捕捉它们并执行其他命令。trap命令允许你来指定shell脚本要观察哪些Linux信号并从shell中拦截。如果脚本收到了trap命令中列出的信号，它会阻止它被shell处理，而在本地处理它。

trap命令的格式是：

```
trap commands signals
```

非常简单！在trap命令行上，你只要列出你想要shell执行的命令，以及一组用空格分开的你想要捕捉的信号。你可以用数值或Linux信号名来指定信号。

这里有个简单的用trap命令来忽略SIGINT和SIGTERM信号的例子：

```
$ cat test1
#!/bin/bash
# testing signal trapping
#
trap "echo ' Sorry! I have trapped Ctrl-C'" SIGINT SIGTERM
echo This is a test program
count=1
while [ $count -le 10 ]
do
    echo "Loop #$count"
    sleep 5
    count=$[ $count + 1 ]
done
echo This is the end of the test program
$
```

本例中用到的trap命令会在每次检测到SIGINT或SIGTERM信号时显示一行简单的文本消息。捕捉这些信号会阻止用户用bash shell键盘的Ctrl+C命令来停止程序：

```
$ ./test1
This is a test program
Loop #1
Loop #2
Loop #3
^C Sorry! I have trapped Ctrl-C
Loop #4
Loop #5
Loop #6
Loop #7
^C Sorry! I have trapped Ctrl-C
Loop #8
Loop #9
Loop #10
This is the end of the test program
$
```

每次使用Ctrl+C组合键，脚本都会执行trap命令中指定的echo语句，而不是忽略此信号并允许shell停止该脚本。

15.1.4 捕捉脚本的退出

除了在shell脚本中捕捉信号，你也可以在shell脚本退出时捕捉它们。这是在shell完成任务时执行命令的简便方法。

要捕捉shell脚本的退出，只要在trap命令后加上EXIT信号就行：

```
$ cat test2
#!/bin/bash
# trapping the script exit

trap "echo byebye" EXIT

count=1
while [ $count -le 5 ]
do
    echo "Loop #$count"
    sleep 3
    count=$(( $count + 1 ))
done
$
$ ./test2
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
byebye
$
```

当脚本运行到常规的退出点时，捕捉就被触发了。shell会执行你在trap命令行指定的命令。EXIT捕捉即使是在提前退出脚本时也会工作：

```
$ ./test2
Loop #1
Loop #2
^Cbyebye
$
```

Ctrl+C组合键用来发送SIGINT信号，脚本退出（由于该信号并未列在捕捉列表中），但在脚本退出前，shell执行了trap命令。

15.1.5 移除捕捉

你可以用单破折线作为命令，后跟要移除的信号来移除一组信号捕捉，将其恢复到正常状态：

```
$ cat test3
#!/bin/bash
# removing a set trap

trap "echo byebye" EXIT

count=1
while [ $count -le 5 ]
do
    echo "Loop #$count"
    sleep 3
```

```
count=$(( $count + 1 )  
done  
trap - EXIT  
echo "I just removed the trap"  
$  
$ ./test3  
Loop #1  
Loop #2  
Loop #3  
Loop #4  
Loop #5  
I just removed the trap  
$
```

一旦信号捕捉被移除了，脚本会忽略该信号。但是，如果在捕捉被移除前收到信号，脚本就会在trap命令中处理它：

```
$ ./test3  
Loop #1  
Loop #2  
^Cbyebye  
  
$
```

在本例中，Ctrl+C组合键用来提前终止脚本。由于该脚本是在捕捉移除前被终止了，脚本就会执行trap中指定的命令。

15.2 以后台模式运行脚本

直接在命令行界面运行shell脚本有时不怎么方便。一些脚本可能要执行很长一段时间，而你可能不想在命令行界面一直等。当脚本在运行时，你没法在终端会话里做别的事情。幸好有个简单的方法可以解决。

在用ps命令时，你会看到运行在Linux系统上的一系列不同进程。显然，所有这些进程都不是运行在你的终端显示器上的。这样的现象我们称为在后台（background）运行进程。在后台模式中，进程运行时不会和终端会话上的STDIN、STDOUT以及STDERR关联（参见第14章）。

你也可以在shell脚本中试试这个特性，允许它们在后台运行而不用占用终端会话。下面几节将会介绍如何在Linux系统上以后台模式运行脚本。

15.2.1 后台运行脚本

以后台模式运行shell脚本是很容易的事。要在命令行界面以后台模式运行shell脚本，只要在命令后加个&符就行了：

```
$ ./test1 &  
[1] 1976  
$ This is a test program  
Loop #1
```

```

Loop #2
ls /home/user/Desktop
gnome-screenshot.desktop konsole.desktop virtualbox.desktop
gnome-terminal.desktop ksnapshot.desktop
$ Loop #3
Loop #4
...
$
```

当&符放到命令后时，它会将命令和bash shell分离开来，并将它作为系统上的独立后台进程运行。显示的第一行是：

```
[1] 1976
```

方括号中的数字是shell分配给后台进程的作业号。下一个数是Linux系统分配给进程的进程ID (PID)。Linux系统上运行的每个进程都必须有一个唯一的PID。

一旦系统显示了这些内容，新的命令行界面提示符就出现了。你可以回到shell，而你所执行的命令正在以后台模式安全的运行。

这时，你可以在提示符输入新的命令（如本例所示）。但是，当后台进程仍在运行时，它仍会使用终端显示器来显示STDOUT和STDERR消息。你会从本例中注意到test1脚本的输出会和shell中运行的其他命令的输出混在一起。

当后台进程结束时，它会在终端上显示一条消息：

```
[1]+ Done
```

```
./test1
```

输出中这些是作业号和作业的状态（完成了），以及开始作业所用的命令。

15.2.2 运行多个后台作业

可以在命令行提示符下同时启动任意多个后台作业：

```

$ ./test1 &
[3] 2174
$ This is Test Program 1

$ ./test2 &
[4] 2176
$ I am Test Program 2

$ ./test3 &
[5] 2178
$ Well this is Test Program

$ ./test4 &
[6] 2180
$ This is Test Program 4

$
```

每次启动新作业时，Linux系统都会为其分配一个新的作业号和PID。通过ps命令，你可以看到所有脚本都在运行中：

```
$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START TIME COMMAND
...
user     1826  0.0  0.3  6704  3408 pts/0  Ss  14:07 0:00 bash
user     2174  0.0  0.1  4860  1076 pts/0  S  15:23 0:00 /bin/bash ./test1
user     2175  0.0  0.0  3884  504 pts/0  S  15:23 0:00 sleep 300
user     2176  0.0  0.1  4860  1068 pts/0  S  15:23 0:00 /bin/bash ./test2
user     2177  0.0  0.0  3884  508 pts/0  S  15:23 0:00 sleep 300
user     2178  0.0  0.1  4860  1068 pts/0  S  15:23 0:00 /bin/bash ./test3
user     2179  0.0  0.0  3884  504 pts/0  S  15:23 0:00 sleep 300
user     2180  0.0  0.1  4860  1068 pts/0  S  15:23 0:00 /bin/bash ./test4
user     2181  0.0  0.0  3884  504 pts/0  S  15:23 0:00 sleep 300
user     2182  0.0  0.1  4592  1100 pts/0  R+ 15:24 0:00 ps au
$
```

你启动的每个后台进程都出现在了ps命令输出的运行中的进程列表里。如果所有的进程都在终端会话中显示输出，那很快就会乱成一团。好在有个简单的解决办法，我们将会在下一节中讨论。

15.2.3 退出终端

在终端会话中使用后台进程时，必须特别小心。注意，ps命令的输出中，每个后台进程都绑定到了该终端会话的终端上了（pts/0）。如果进程会话退出了，后台进程也会退出。

如果你有关联到该终端的还在运行的后台进程，有的终端模拟器会提醒你，其他的就不会提醒。如果你想退出控制台后，脚本继续以后台形式运行，你需要其他的方法。下一节将会讨论这个过程。

15.3 在非控制台下运行脚本

有时你会想在终端会话中启动shell脚本，然后让脚本一直以后台模式运行，直到其完成，即使你退出了终端会话。可以用nohup命令来实现。

nohup命令运行了另外一个命令来阻断所有发送给该进程的SIGHUP信号。这会在退出终端会话时阻止进程退出。

nohup命令的格式如下：

```
$ nohup ./test1 &
[1] 19831
$ nohup: ignoring input and appending output to 'nohup.out'
$
```

和普通后台进程一样，shell会给命令分配一个作业号，Linux系统会分配一个PID号。区别在于，当你使用nohup命令时，如果关闭该会话，脚本会忽略任何终端会话发过来的SIGHUP信号。

由于nohup命令会从终端解除进程的关联，进程会丢掉到STDOUT和STDERR的链接。为了保存该

命令产生的输出，nohup命令会自动将STDOUT和STDERR的消息重定向到一个名为nohup.out的文件中。

nohup.out文件包含了通常发到终端显示器上的所有输出。在进程完成运行后，你可以查看nohup.out文件中的输出结果：

```
$ cat nohup.out
This is a test program
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
Loop #6
Loop #7
Loop #8
Loop #9
Loop #10
This is the end of the test program
$
```

输出会出现在nohup.out文件中，就跟进程在命令行下运行时一样。

警告 如果你用nohup运行了另一个命令，输出会追加到现有的nohup.out文件中。在运行同目录中的多个命令时，要小心了，因为所有的输出都会发送到同一个nohup.out文件，可能会有点叫人困惑。

15.4 作业控制

在本章前面，你了解了如何用组合键来停止shell中正在运行的作业。在作业停止后，Linux系统会让你无条件终止它或重启它。你可以用kill命令终止该进程。要重启停止的进程需要向其发送一个SIGCONT信号。

启动、停止、无条件终止以及恢复作业的这些功能统称为作业控制。通过作业控制，你就能完全控制shell中正在运行的所有作业了。

本节就来介绍用于查看和控制在shell中运行的作业的命令。

15.4.1 查看作业

作业控制中的关键命令是jobs命令。jobs命令允许查看shell当前正在处理的作业：

```
$ cat test4
#!/bin/bash
# testing job control

echo "This is a test program $$"
count=1
while [ $count -le 10 ]
```

```

do
    echo "Loop #$count"
    sleep 10
    count=$(( $count + 1 ))
done
echo "This is the end of the test program"
$
$ ./test4
This is a test program 29011
Loop #1
^Z
[1]+  Stopped                  ./test4
$
$ ./test4 > test4out &
[2] 28861
$
$ jobs
[1]+  Stopped                  ./test4
[2]-  Running                  ./test4 >test4out &
$
```

此脚本用\$\$变量来显示Linux系统分配给该脚本的PID，然后进入循环，每次迭代都休眠10s。在例子中，第一个脚本是从命令行界面启动的，然后用Ctrl+Z组合键停止。下一步，另一个作业以后台进程的方式启动，使用了&符。要让事情简单一些的话，将脚本的输出重定向到文件中，这样它就不会出现在屏幕上。

在两个作业开始后，我们用jobs命令来查看分配给shell的作业。jobs命令会显示这两个已停止/运行中的作业，以及它们的作业号和作业中使用的命令。

jobs命令使用一些不同的命令行参数，见表15-2。

表15-2 jobs命令参数

参数	描述
-l	列出进程的PID以及作业号
-n	只列出上次shell发出的通知后改变了状态的作业
-p	只列出作业的PID
-r	只列出运行中的作业
-s	只列出已停止的作业

你会注意到jobs命令输出中的加号和减号。带加号的作业会被当做默认的作业。在使用作业控制命令时，如果未在命令行指定任何作业号，该作业会被当成操作对象。带减号的作业则会在当前默认作业完成处理时成为下一个默认作业。任何时候都只有一个带加号的作业和一个带减号的作业，不管shell中有多少个正在运行的作业。

下面例子说明了队列中的下一个作业在默认作业移除时如何成为默认的作业：

```

$ ./test4
This is a test program 29075
Loop #1
^Z
```

```
[1]+ Stopped                  ./test4
$ 
$ ./test4
This is a test program 29090
Loop #1
^Z
[2]+ Stopped                  ./test4
$ 
$ ./test4
This is a test program 29105
Loop #1
^Z
[3]+ Stopped                  ./test4
$ 
$ jobs -1
[1] 29075 Stopped            ./test4
[2]- 29090 Stopped            ./test4
[3]+ 29105 Stopped            ./test4
$ 
$ kill -9 29105
$ 
$ jobs -1
[1]- 29075 Stopped            ./test4
[2]+ 29090 Stopped            ./test4
$ 
```

在本例中，3个单独的进程被启动，然后停止。jobs命令列出了3个进程和它们的状态。注意，默认进程（带有加号的那个）是最后启动的那个进程。

然后我们调用了kill命令来向默认进程发送了一个SIGHUP信号。在下一个jobs命令输出中，先前带有减号的作业成了现在的默认作业。

15.4.2 重启停止的作业

在bash作业控制中，你可以将已停止的作业作为后台进程或前台进程重启。前台进程会接管你当前工作的终端，所以在使用该功能时要小心了。

要以后台模式重启一个作业，可用bg命令加上作业号：

```
$ bg 2
[2]+ ./test4 &
Loop #2
$ Loop #3
Loop #4

$ jobs
[1]+ Stopped                  ./test4
[2]- Running                  ./test4 &
$ Loop #6
Loop #7
Loop #8
Loop #9
Loop #10
This is the end of the test program
```

```
[2]- Done          ./test4
$
```

由于作业是以后台模式重启的，命令行界面的提示符会出现，允许输入其他命令。jobs命令的输出现在说明该作业实际上是在运行的（如你从显示器上的输出中看到的）。

要以前台模式重启作业，可用fg命令，加上作业号：

```
$ jobs
[1]+  Stopped          ./test4
$ fg 1
./test4
Loop #2
Loop #3
```

由于作业是以前台模式运行的，命令行界面的提示符不会出现，直到该作业完成。

15.5 调整谦让度

在多任务操作系统中（Linux就是），内核负责将CPU时间分配给系统上运行的每个进程。实际上每次只有一个进程可以运行在CPU上，所以内核将CPU时间轮流分配给每个进程。

默认情况下，从shell启动的所有进程在Linux系统上都有同样的调度优先级（scheduling priority）。调度优先级是内核分配给该进程的相对于分配给其他进程的CPU时间总量。

调度优先级是个整数值，从-20（最高优先级）到+20（最低优先级）。默认情况下，bash shell以优先级0来启动所有进程。

提示 最低值-20是最优优先级，而最高值19是最低优先级，这太容易记混了。只要记住那句俗语“好人难做”就好。越是“好”或高的值，获得CPU时间的机会越低。

这意味着，只需要一点处理时间的简单脚本会获得和需要耗费数小时运行的复杂数学算法相同的CPU时间片。

有时你想要改变特定命令的优先级，不管是通过降低它的优先级（这样它就不会从CPU获得那么多的处理能力），还是给它更高的优先级（这样它就能获得更多的处理时间）。你可以通过nice命令做到。

15.5.1 nice命令

nice命令允许你在启动时调整一个命令的调度优先级。要让命令以更低的优先级运行，只要用nice的-n命令行来指定新的优先级级别：

```
$ nice -n 10 ./test4 > test4out &
[1] 29476
$ ps al
F  UID  PID  PPID PRI  NI WCHAN  STAT TTY      TIME COMMAND
100 501 29459 29458  12  0 wait4 S    pts/0      0:00 -bash
```

```

000 501 29476 29459 15 10 wait4 SN pts/0      0:00 /bin/bash
000 501 29490 29476 15 10 nanosl SN pts/0      0:00 sleep 10
000 501 29491 29459 14 0 -      R pts/0      0:00 ps a1
$
```

nice命令会让脚本以更低的优先级运行。但如果你想增加命令中某个的优先级，你可能会大吃一惊：

```

$ nice -n -10 ./test4 > test4out &
[1] 29501
$ nice: cannot set priority: Permission denied
[1]+ Exit 1           nice -n -10 ./test4 >test4out
```

nice命令阻止普通系统用户来增加命令的优先级。这个安全特性会阻止普通用户以高优先级运行他的命令。

15.5.2 renice命令

有时你想改变系统上已运行命令的优先级。这正是renice命令所干的事。它允许你指定运行进程的PID来改变它的优先级：

```

$ ./test4 > test4out &
[1] 29504
$ ps a1
  F  UID  PID  PPID PRI  NI WCHAN  STAT TTY          TIME COMMAND
100 501 29459 29458 12  0 wait4 S  pts/0      0:00 -bash
000 501 29504 29459  9  0 wait4 S  pts/0      0:00 /bin/bash .
000 501 29518 29504  9  0 nanosl S  pts/0      0:00 sleep 10
000 501 29519 29459 14  0 -      R  pts/0      0:00 ps a1
$ renice 10 -p 29504
29504: old priority 0, new priority 10
$ ps a1
  F  UID  PID  PPID PRI  NI WCHAN  STAT TTY          TIME COMMAND
100 501 29459 29458 16  0 wait4 S  pts/0      0:00 -bash
000 501 29504 29459 14 10 wait4 SN  pts/0      0:00 /bin/bash .
000 501 29535 29504  9  0 nanosl S  pts/0      0:00 sleep 10
000 501 29537 29459 14  0 -      R  pts/0      0:00 ps a1
$
```

renice命令会自动更新当前运行进程的调度优先级。和nice命令一样，renice命令也有一些限制：

- 你只能对属于你的进程执行renice；
- 你只能通过renice降低进程的优先级；
- root用户可以通过renice来调整任何进程的优先级到任何级别。

如果想完全控制运行进程，你必须以根账户身份登录或使用sudo命令。

15.6 定时运行作业

我敢肯定，当你开始使用脚本时，肯定会有你想要在某个预设时间运行脚本的情况，通常

在你不在场的情况下。Linux系统提供了多个在预选时间运行脚本的方法：at命令和cron表。每个方法使用一种不同的技术来计划何时运行脚本以及多久运行一次。下面几节将会介绍其中的每一种方法。

15.6.1 用at命令来计划执行作业

at命令允许指定Linux系统何时运行脚本。at命令会将作业提交到队列中，指定shell何时运行该作业。at的守护进程atd会以后台模式运行，并检查作业队列来运行作业。大多数Linux发行版会在启动时运行此守护进程。

atd守护进程会检查系统上的一个特殊目录（通常位于/var/spool/at）来获取用at命令提交的作业。默认情况下，atd守护进程会每60s检查一下这个目录。有作业时，atd守护进程会检查作业设置运行的时间。如果时间跟当前时间匹配，atd守护进程会运行此作业。

后面几节会介绍如何用at命令提交要运行的作业以及如何管理这些作业。

1. at命令的格式

at命令的基本格式非常简单：

```
at [-f filename] time
```

默认情况下，at命令会将STDIN的输入放到队列中。你可以用-f参数来指定读取命令（脚本文件）的文件名。

time参数指定了Linux系统何时运行该作业。在如何指定时间这个问题上，你可以非常灵活。at命令能识别多种不同的时间格式：

- 标准的小时和分钟格式，比如10:15；
 - ~AM/~PM指示符，比如10:15~PM；
 - 特定可命名的时间，比如now、noon、midnight或者teatime（4~PM）。
- 如果指定了一个已经过去的时间，at命令会在第二天该时间运行该作业。
- 除了指定运行作业的时间，也可以指定特定的日期，通过不同的日期格式：
- 标准日期格式，比如MMDDYY、MM/DD/YY或DD.MM.YY；
 - 文本日期，比如Jul 4或Dec 25，加不加年份均可。

你也可以指定时间增量：

- 当前时间+25 min；
- 明天10:15~PM；
- 10:15 + 7 天。

在你使用at命令时，该作业会被提交到作业队列（job queue）中。作业队列会保存通过at命令提交的待处理的作业。针对不同优先级，存在26中不同的作业队列。作业队列通常用小写字母a~z来引用。

说明 几年前，batch命令是允许脚本在后面某个时间运行的另一种方法。batch命令很特别，你可以计划在系统低负载时运行脚本。但现在，batch命令只是一个简单的脚本，/usr/bin/batch，它会调用at命令并将作业提交到b队列中。

作业队列的字母排序越高，作业运行的优先级就越低。默认情况下，at的作业会被提交到at作业队列。如果你想以更高优先级运行作业，你可以用-q参数指定不同的队列字母。

2. 获取作业的输出

当作业在Linux系统上运行时，没有屏幕会关联到该作业。取而代之的是，Linux系统会将提交该作业的用户的E-mail地址作为STDOUT和STDERR。任何发到STDOUT或STDERR的输出都会通过邮件系统发送给该用户。

这里有个使用at命令计划执行作业的例子：

```
$  
$ cat test5  
#!/bin/bash  
#  
# testing the at command  
#  
  
echo This script ran at `date`  
echo This is the end of the script >&2  
$  
$ date  
Mon Oct 18 14:38:17 EDT 2010  
$  
$ at -f test5 14:39  
warning: commands will be executed using /bin/sh  
job 57 at Mon Oct 18 14:39:00 2010  
$  
$ mail  
"/var/mail/user": 1 message 1 new  
>N 1 user           Mon Oct 18 14:39  15/538  Output from your job  
& 1  
Date: Mon, 18 Oct 2010 14:39:00 -0400  
Subject: Output from your job      57  
To: user@user-desktop  
From: user <user@user-desktop>  
  
This script ran at Mon Oct 18 14:39:00 EDT 2010  
This is the end of the script  
  
& exit  
$
```

at命令会生成一条警告消息，说明系统用哪个shell来运行脚本/bin/sh，以及关联到该作业的作业号和该作业计划运行的时间。

警告 在大多数Linux发行版中，赋给/bin/sh的默认shell是bash shell。但Ubuntu将dash shell作为其默认shell。第22章将会介绍更多关于dash shell的信息。

当作业完成时，屏幕上不会有任何输出，但系统生成了一封邮件消息。该邮件消息会显示脚本产生的输出。如果脚本没有产生任何输出，默认情况下它就不会生成邮件消息。你可以在at命令中用-m选项来改变这种情况。它会生成一封E-mail消息，说明作业完成了，即使脚本并未产生任何输出。

3. 列出等待的作业

atq命令可以查看系统中有哪些作业在等待：

```
$ at -f test5 15:05
warning: commands will be executed using /bin/sh
job 58 at Mon Oct 18 15:05:00 2010
$

$ at -f test5 15:10
warning: commands will be executed using /bin/sh
job 59 at Mon Oct 18 15:10:00 2010
$

$ at -f test5 15:15
warning: commands will be executed using /bin/sh
job 60 at Mon Oct 18 15:15:00 2010
$

$ at -f test5 15:20
warning: commands will be executed using /bin/sh
job 61 at Mon Oct 18 15:20:00 2010
$$ atq
61    Mon Oct 18 15:20:00 2010 a user
58    Mon Oct 18 15:05:00 2010 a user
59    Mon Oct 18 15:10:00 2010 a user
60    Mon Oct 18 15:15:00 2010 a user
$
```

作业列表中显示了作业号、系统运行该作业的日期和时间及其所在作业队列。

4. 删除作业

一旦你知道了哪些作业在作业队列中等待，你就能用atrm命令来删除等待中的作业：

```
$ atq
59    Mon Oct 18 15:10:00 2010 a user
60    Mon Oct 18 15:15:00 2010 a user
$
$ atrm 59
$
$ atq
60    Mon Oct 18 15:15:00 2010 a user
$
```

只要指定你想要删除的作业号就行了。你只能删除你提交的要执行的作业，不能删除其他人

提交的作业。

15.6.2 计划定期执行脚本

用at命令来计划在预设时间执行脚本非常好用，但如果你需要脚本在每天的同一时间运行或是每周一次甚至每月一次呢？不用继续使用at来提交作业，你可以用Linux系统的另一个功能。

Linux系统使用cron程序来计划要定期执行的作业。cron程序会在后台运行并检查特殊的称做cron时间表（cron table）的表，来获得计划执行的作业。

1. cron时间表

cron时间表采用一种特别的格式来指定作业何时运行。cron时间表的格式如下：

```
min hour dayofmonth month dayofweek command
```

cron时间表允许你用特定值、值范围（比如1~5）或者是通配符（星号）来指定条目。例如，如果想在每天的10:15运行一个命令，你可以用cron时间表条目：

```
15 10 * * * command
```

在dayofmonth、month以及dayofweek字段中使用的通配符说明，cron会在每个月每天的10:15执行该命令。要指定一个每周一4:15 PM运行的命令，你可以用下面的条目：

```
15 16 * * 1 command
```

你可以用三字符的文本值（mon、tue、wed、thu、fri、sat、sun）或数值（0为周日，6为周六）来指定dayofweek条目。

这里有另外一个例子，在每个月的第一天中午12点执行命令，你可以用下面的格式：

```
00 12 1 * * command
```

dayofmonth条目会为月份指定个日期值（1~31）。

说明 聪明的读者可能会问如何设置一个命令在每个月的最后一天执行，因为你无法设置dayofmonth的值来涵盖所有的月份。这个问题困扰着Linux和Unix程序员，也激发了不同的解决办法。常用的方法是加一条使用date命令的if-then语句来检查明天的日期是不是01：

```
00 12 * * * if [ `date +%d -d tomorrow` = 01 ] ; then : command
```

它会在每天中午12点来检查是不是当月的最后一天，如果是，cron将会运行该命令。

命令列表必须指定要运行的命令或脚本的全路径名。你可以添加任何想要的命令行参数和重定向符号，作为定期命令行：

```
15 10 * * * /home/rich/test4 > test4out
```

cron程序会用提交作业的用户账户运行该脚本。因此，你必须有访问该命令和命令中指定的输出文件的权限。

2. 构建cron时间表

每个系统用户都可以用他自己的cron时间表（包括root用户）来运行计划好的任务。Linux提供了crontab命令来处理cron时间表。要列出已有的cron时间表，可以用-1参数：

```
$ crontab -l
no crontab for rich
$
```

默认情况下，每个用户的cron时间表文件不存在。要为cron时间表添加条目，可以用-e参数。在你操作时，crontab命令会用已有的cron时间表（或者一个空文件，如果时间表不存在的话）启动一个文本编辑器（参见第9章）。

3. cron目录

当你创建的脚本不要求有精确的执行时间时，用预配置的cron脚本目录会更方便。有4个基本目录：hourly、daily、monthly和weekly。

```
$ ls /etc/cron.*ly
/etc/cron.daily:
 0anacron    aptitude      exim4-base      man-db      sysstat
  apt        bsdmainutils  logrotate      popularity-contest
  apport     dpkg         mlocate      standard
/etc/cron.hourly:
/etc/cron.monthly:
 0anacron
/etc/cron.weekly:
 0anacron  apt-xapian-index  man-db
$
```

因此，如果你有脚本需要每天运行一次，只要将脚本复制到daily目录，cron就会每天执行它。

4. anacron程序

cron程序的唯一问题是它假定Linux系统是 7×24 小时运行的。除非你将Linux当成服务器环境来运行，否则此假设并不成立。

如果在cron时间表中的作业计划运行时间内Linux系统被关闭，那作业就不会运行了。当系统开机时，cron程序不会回去再运行哪些错过的作业。要解决这个问题，许多Linux发行版还包含了anacron程序。

如果anacron知道作业错过了计划好的运行，它会尽快运行该作业。这意味着如果Linux系统关机了几天，当它再次开机时，关机时错过执行的计划好的作业会自动运行。

这个功能通常用于进行常规日志维护的脚本。如果系统在脚本应该运行的时间刚好关机，日志文件就不会被整理，可能会变得很大。通过anacron，你至少可以保证系统每次启动时日志文件会被整理。

anacron程序只会处理位于cron目录的程序，比如/etc/cron.monthly。它用时间戳来决定作业是否在适当的计划间隔内运行了。每个cron目录都有个时间戳文件，位于/var/spool/anacron：

```
$  
$ sudo cat /var/spool/anacron.monthly  
20101123  
$
```

anacron程序有自己的用来检查作业目录的表（通常位于/etc/anacrontab）：

```
$  
$ cat /etc/anacrontab  
# /etc/anacrontab: configuration file for anacron  
  
# See anacron(8) and anacrontab(5) for details.  
  
SHELL=/bin/sh  
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin  
  
# These replace cron's entries  
1 5 cron.daily nice run-parts --report /etc/cron.daily  
7 10 cron.weekly nice run-parts --report /etc/cron.weekly  
@monthly 15 cron.monthly nice run-parts --report /etc/cron.monthly  
$
```

anacron时间表的基本格式会和cron时间表略有不同：

```
period delay identifier command
```

period条目定义了作业多久运行一次，以天为单位。anacron程序用此条目来检查作业的时间截文件。**delay**条目会指定系统启动后多少分钟后anacron程序开始运行错过的脚本。**command**条目包含了run-parts程序和一个cron脚本目录名。run-parts程序负责运行目录中传给它的任何脚本。

注意，anacron不会运行位于/etc/cron.hourly的脚本。这是因为anacron程序不会处理执行时间需求小于一天的脚本。

identifier条目是一种特别的非空白字符字符串，如cron-weekly。它用于唯一识别日志消息和错误E-mail中的作业。

15.7 启动时运行

启动shell脚本的最后一一种方法是，在Linux系统刚开机时或用户刚启动一个新的bash shell会话时自动运行。开机时运行的脚本通常是执行系统功能的特殊脚本，比如配置网口或启动服务器进程。但如果你是Linux系统管理员，你也可能需要在Linux系统每次开机时执行某个功能，比如重置定制的日志文件或启动特定应用。

在每次用户启动一个新的bash shell时（即使只是特定用户启动bash shell时），运行脚本也可能会派上用场。有时你想设置shell会话的shell功能或确保某个特定文件设置了。

本节将会介绍如何配置Linux系统来在开机时或新bash shell启动时运行脚本。

15.7.1 开机时运行脚本

在你让shell脚本在开机时运行前，你需要知道一点关于Linux开机过程如何工作的知识。Linux

采用特定的顺序来在开机时运行脚本，了解这个过程能帮你让脚本正确工作。

1. 开机过程

在你开始运行Linux系统时，Linux内核会加载到内存中并运行。它做的第一件事是开始UNIX System V init过程或Upstart init过程，具体取决于发行版和版本。然后这个过程将负责启动Linux系统上的所有其他进程。

● System V init过程

作为开机过程的一部分，System V init过程会读取/etc/inittab文件。inittab文件会列出系统的运行级（run level）。不同的Linux运行级会启动不同的程序和脚本。表15-3列出了基于Red Hat的发行版的Linux运行级。

表15-3 Linux运行级——基于Red Hat的发行版

运 行 级	描 述
0	关机
1	单用户模式
2	多用户模式，通常不支持网络
3	全功能的多用户模式，支持网络
4	可定义用户
5	多用户模式，支持网络和图形化X Window会话
6	重启

Linux发行版普遍采用表15-3中列出的运行级来更好地控制启动系统上的那些服务。但基于Debian的发行版，比如Ubuntu和Linux Mint，并不区分运行级2~5，如表15-4所示。

表15-4 Linux运行级——基于Debian的发行版

运 行 级	描 述
0	关机
1	单用户模式
2~5	多用户模式，支持网络和图形化X Window会话
6	重启

Ubuntu发行版甚至都没有/etc/inittab文件来设定运行级。默认情况下，Ubuntu会以运行级2运行。如果你想修改默认的运行级，你必须在Ubuntu发行版上创建一个/etc/inittab文件。

每个运行级将会定义System V init过程启动或停止哪些脚本。这些开机脚本（startup script）都是shell脚本，通过提供必要的环境变量来启动应用程序。

这是基于System V init的开机过程的一部分，这个过程本身有点含混不清，主要是因为不同的Linux发行版存放开机脚本的位置略有不同。有些发行版将开机脚本放在/etc/rc#.d目录，其中#代表运行级。其他的则采用/etc/init.d目录，还有其他一些则采用/etc/init.d/rc.d目录。通常瞥一眼/etc目录结构就可以很容易判定此发行版所采用的格式。

- Upstart init过程

Upstart init过程是许多Linux发行版正在迁移过去的管理服务进程的一个新标准。Upstart并不关注系统的运行级，而关注时间，比如系统开机。在Upstart中，系统开机称为开机事件（startup event）。

Upstart使用位于/etc/event.d或/etc/init目录下的文件来启动进程，具体取决于发行版和版本。为了维护向后兼容性，许多Upstart实现仍会调用较早的位于/etc/init.d以及/etc/rc#.d目录中的System V init脚本。

Upstart init过程标准目前还在快速演进。对于你创建的想在系统开机事件中启动的脚本，可能还得保守些，用早些的System V init过程方法。此方法将会在下一节介绍。

2. 定义自己的开机脚本

最好不要和Linux发行版上已有的独立开机脚本文件混起来。通常发行版会提供一些工具来在你添加服务器程序时自动构建这些脚本，而手动修改这些脚本可能会造成各种问题。

与之不同的是，大多数Linux发行版提供了一个本地开机文件专门让系统管理员添加开机时运行的脚本。当然，该文件的名字和位置在不同Linux发型版中也不相同。表15-5定义了五大主流Linux发行版上该开机文件的位置。

表15-5 Linux本地开机文件位置

发 行 版	文件位置
debian	/etc/init.d/rc.local
Fedora	/etc/rc.d/rc.local
Mandriva	/etc/rc.local
openSuse	/etc/init.d/boot.local
Ubuntu	/etc/rc.local

在本地开机文件中，你可以指定特定命令或语句，或者输入任何你想在开机时启动的脚本。记住，如果你要用脚本，你必须指定该脚本的全路径名，这样系统才能在开机时找到。

警告 不同Linux发行版也会在开机过程的不同时间点执行该本地开机脚本。有时该脚本会在网络支持等启动前运行。在运行你的发行版中的本地开机脚本时，参考一下你的Linux发行版的文档。

15.7.2 在新shell中启动

每个用户的主目录下都有两个文件，bash shell会用它们来自动启动脚本并设置环境变量：

- .bash_profile文件；
- .bashrc文件。

当新shell是新的登录生成的话，bash shell会运行.bash_profile文件。可以把任何登录时要运行的脚本放到该文件中。

当新shell启动时，包括有新的登录的情况，bash shell会运行.bashrc文件。你可以通过向你的主目录中的.bashrc文件加一条echo语句并启动一个新shell来测试：

```
$ bash  
This is a new shell!!  
$
```

如果你想为系统中的所有用户运行一个脚本，大多数Linux发行版提供了/etc/bashrc文件（注意，在bashrc文件名之前没有句点）。每次有用户在系统上启动一个新的bash shell时，bash shell就会执行该文件中的语句。

15.8 小结

Linux系统允许利用信号来控制shell脚本。bash shell接受信号并将它们传给该shell进程中运行的所有进程。Linux信号允许你来方便地无条件终止一个失控进程或暂停一个长时间运行的进程。

你可以在脚本中用trap语句来捕捉信号并执行特定命令。这个功能提供了简单的控制用户是否可以在脚本运行时中断脚本的方法。

默认情况下，当你在终端会话shell中运行脚本时，交互式shell会暂停直到脚本运行完。你可以在命令名后加一个&符来让脚本或命令以后台模式运行。当你以后台模式运行命令或脚本时，交互式shell会返回，允许你继续输入其他命令。任何通过这种方法运行的后台进程仍会绑定到该终端会话。如果你退出了终端会话，后台进程也会退出。

要阻止这种情况发生，可以用nohup命令。该命令会拦截任何发给某个命令来停止其运行的信号，比如，当你退出终端会话时。这会允许脚本继续以后台模式运行，尽管你已经退出了终端会话。

当你将进程移动到后台时，你仍然可以控制它的运行。jobs命令允许查看该shell会话启动的进程。只要你知道后台进程的作业号，你就可以用kill命令向该进程发送Linux信号，或者用fg命令将该进程带到该shell会话的前端来。你可以用Ctrl+Z组合键来暂停运行中的前端进程，然后用bg命令将其放回后台模式。

nice命令和renice命令允许你调整进程的优先级。通过降低进程的优先级，你可以让CPU给该进程分配更少的时间。这在运行可能会占用大量CPU时间的长时间进程时可以派上用场。

除了在进程运行时控制进程外，你还可以决定进程在系统上的启动时间。不用直接在命令行界面的提示符上直接运行脚本，你可以计划在另一个时间运行该进程。有几种不同的实现途径。at命令允许你在预设时间运行脚本。cron程序提供了定期运行脚本的接口。

最后，Linux系统提供了脚本文件来在系统开机时或用户启动新bash shell时运行脚本。各个发行版提供了不同的文件来让你列出系统每次开机时要启动的脚本。类似地，.bash_profile和.bashrc文件位于每个用户的主目录，从而可以提供一个地方来存放新shell启动时要运行的脚本和命令。.bash_profile文件会在用户每次登录到系统时运行脚本，.bashrc文件会在启动每个shell实例时运行脚本。

下一章中，我们将会了解如何编写脚本函数。你只要在脚本函数编写一次代码块，就能在整个脚本中多次使用这些代码块了。



Part 3

第三部分

高级 shell 脚本编程

本部分内容

- 第 16 章 创建函数
- 第 17 章 图形化桌面上的脚本编程
- 第 18 章 初识 sed 和 gawk
- 第 19 章 正则表达式
- 第 20 章 sed 进阶
- 第 21 章 gawk 进阶
- 第 22 章 使用其他 shell

本章内容

- 基本的脚本函数
- 返回值
- 在函数中使用变量
- 数组变量和函数
- 函数递归
- 创建库
- 在命令行上使用函数

通常在编写shell脚本时，你会发现在多个地方使用了同一段代码。如果只是一小段代码，一般也无关紧要。但要在shell脚本中多次重写大块代码段就会比较辛苦了。`bash shell` 支持用户定义的函数，这样就解决了这个难题。你可以将shell脚本代码放进函数中封装起来，这样就能在脚本中的任何地方多次使用它了。本章将会带你逐步了解创建自己的shell脚本函数的过程，并演示如何在shell脚本应用程序中使用它们。

16.1 基本的脚本函数

在开始编写较复杂的shell脚本时，你会发现自己在重用执行特定任务的部分代码。有时这部分代码很简单，比如显示一条文本消息，或者从脚本用户那里获得一个答案；有时则会比较复杂，作为较大进程中的一部分多次使用。

在每种情况下，在脚本中一遍又一遍地写同样的代码块都会是件烦人的事。如果能只写一次代码块，而在脚本中多次引用该代码块就太好了。

`bash shell` 有个特性允许你这么做。函数（function）是可以起个名字并在代码中任何位置重用的代码块。你要在脚本中使用该代码块时，只要使用分配的函数名就行了（这个过程称为调用函数）。本节将会介绍如何在shell脚本中创建和使用函数。

16.1.1 创建函数

有两种格式可以用来在bash shell脚本中创建函数。第一种格式采用关键字function，后跟分配给该代码块的函数名：

```
function name {  
    commands  
}
```

name属性定义了赋予函数的唯一名称。你必须给脚本中定义的每个函数赋个唯一的名称。

commands是构成函数的一条或多条bash shell命令。在调用该函数时，bash shell会按命令在函数中出现的顺序执行命令，跟在普通脚本中一样。

在bash shell脚本中定义函数的第二种格式跟在其他编程语言中定义函数很像：

```
name() {  
    commands  
}
```

函数名后的圆括号为空，表明正在定义的是一个函数。这种格式的命名规则和上个定义shell脚本函数的格式一样。

16.1.2 使用函数

要在脚本中使用函数，在行上指定函数名就行了，跟使用其他shell命令一样：

```
$ cat test1  
#!/bin/bash  
# using a function in a script  
  
function func1 {  
    echo "This is an example of a function"  
}  
  
count=1  
while [ $count -le 5 ]  
do  
    func1  
    count=$((count + 1))  
done  
  
echo "This is the end of the loop"  
func1  
echo "Now this is the end of the script"  
$  
$ ./test1  
This is an example of a function  
This is the end of the loop
```

```
This is an example of a function
Now this is the end of the script
$
```

每次引用func1函数名时，bash shell会回到func1函数的定义并执行你在那里定义的命令。函数定义不必是shell脚本中最前面的事，但要小心。如果在函数被定义前使用函数，你会收到一条错误消息：

```
$ cat test2
#!/bin/bash
# using a function located in the middle of a script

count=1
echo "This line comes before the function definition"

function func1 {
    echo "This is an example of a function"
}

while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func2
echo "Now this is the end of the script"

function func2 {
    echo "This is an example of a function"
}
$ ./test2
This line comes before the function definition
This is an example of a function
This is the end of the loop
./test2: func2: command not found
Now this is the end of the script
$
```

第一个函数func1在脚本中是在几条语句之后才定义的，这当然没任何问题。当func1函数在脚本中被使用时，shell知道去哪里找它。

然而，脚本却试图在func2函数被定义之前使用它。由于func2函数还没有定义，脚本运行到使用它的地方时，产生了一条错误消息。

你也必须注意函数名。记住，函数名必须是唯一的，否则也会有问题。如果你重定义了函数，新定义会覆盖原来函数的定义，而不会产生任何错误消息：

```
$ cat test3
#!/bin/bash
# testing using a duplicate function name

function func1 {
echo "This is the first definition of the function name"
}

func1

function func1 {
    echo "This is a repeat of the same function name"
}

func1
echo "This is the end of the script"
$

$ ./test3
This is the first definition of the function name
This is a repeat of the same function name
This is the end of the script
$
```

func1函数的原始定义工作正常，但在func1函数的第二次定义后，后续该函数的出现都会用第二个定义。

16.2 返回值

bash shell会把函数当做小型脚本，运行结束时会返回一个退出状态码（参见第10章）。有3种不同的方法来为函数生成退出状态码。

16.2.1 默认退出状态码

默认情况下，函数的退出状态码是函数中最后一条命令返回的退出状态码。在函数执行结束后，你可以用标准的\$?变量来决定函数的退出状态码：

```
$ cat test4
#!/bin/bash
# testing the exit status of a function

func1() {
    echo "trying to display a non-existent file"
    ls -l badfile
}

echo "testing the function: "
func1
echo "The exit status is: $?"
$

$ ./test4
```

```
testing the function:
trying to display a non-existent file
ls: badfile: No such file or directory
The exit status is: 1
$
```

函数的退出状态码是1，这是因为函数中的最后一条命令没有成功运行。但无法知道函数中其他命令中是否成功运行。看下面的例子：

```
$ cat test4b
#!/bin/bash
# testing the exit status of a function

func1() {
    ls -l badfile
    echo "This was a test of a bad command"
}

echo "testing the function:"
func1
echo "The exit status is: $?"
$ 
$ ./test4b
testing the function:
ls: badfile: No such file or directory
This was a test of a bad command
The exit status is: 0
$
```

这次，由于函数以成功运行的echo语句结尾，函数的退出状态码就是0，尽管函数中有一条命令没有成功运行。使用函数的默认退出状态码是很危险的。幸运的是，有几种办法可以解决这个问题。

16.2.2 使用 return 命令

bash shell使用return命令来退出函数并返回特定的退出状态码。return命令允许指定一个整数值来定义函数的退出状态码，从而提供了编程设定函数退出状态码的简便途径：

```
$ cat test5
#!/bin/bash
# using the return command in a function

function dbl {
    read -p "Enter a value: " value
    echo "doubling the value"
    return ${value * 2}
}

dbl
echo "The new value is $?"
$
```

dbl函数会将\$value变量中用户输入的值翻倍，然后用return命令返回结果。脚本用\$?变量显

示了该值。

但当用这种方法从函数中返回值时，要小心了。记住下面两条技巧来避免问题：

- 记住，函数一结束就取返回值；
- 记住，退出状态码必须在0~255之间。

如果你在用\$?变量提取函数返回值之前执行了其他命令，函数的返回值可能会丢失。记住，\$?变量会返回执行的最后一条命令的退出状态码。

第二个问题定义了使用这种返回值方法的限制。由于退出状态码必须小于256，函数的结果必须生成一个小于256的整数值。任何大于256的值都会返回一个错误值：

```
$ ./test5
Enter a value: 200
doubling the value
The new value is 1
$
```

要返回较大的整数值或者字符串值的话，你就不能用这种返回值的方法了。取而代之，你必须使用另外一种方法，下节中将会介绍。

16.2.3 使用函数输出

正如同可以将命令的输出保存到shell变量中一样，也可以将函数的输出保存到shell变量中。可以用这种技术来获得任何类型的函数输出，并将其保存到变量中：

```
result=`dbl`
```

这个命令会将dbl函数的输出赋给\$result变量。下面是在脚本中使用这种方法的例子：

```
$ cat test5b
#!/bin/bash
# using the echo to return a value

function dbl {
    read -p "Enter a value: " value
    echo ${value}*2
}

result=`dbl`
echo "The new value is $result"
$

$ ./test5b
Enter a value: 200
The new value is 400
$

$ ./test5b
Enter a value: 1000
The new value is 2000
$
```

新函数会用echo语句来显示计算的结果。该脚本会捕捉dbl函数的输出，而不是看答案的退

出状态码。

本例演示了一个小技巧。你会注意到db1函数输出了两条消息。read命令输出了一条简短的消息来向用户询问输入值。**bash shell**脚本会聪明地不将它作为STDOUT输出的一部分，并且忽略掉它。如果你用echo语句生成这条消息来向用户查询，则shell命令会将其与输出值一起读进变量中。

说明 通过这种方法，你还可以返回浮点值和字符串值。这让它非常适合返回函数值。

16.3 在函数中使用变量

你可能已经注意到，上一节的test5例子中我们在函数里用了一个叫做\$value的变量来保存处理的值。在函数中使用变量时，你需要注意一下如何定义和处理它们。这是**shell**脚本中常见的产生问题的原因。本节将会复习一下在**shell**脚本函数内外处理变量的一些方法。

16.3.1 向函数传递参数

如我们在16.2节中提到的，**bash shell**会将函数当做小型脚本来对待。这意味着你可以向函数传递参数，就跟普通脚本一样（参见第13章）。

函数可以使用标准的参数环境变量来代表命令行上传给函数的参数。例如，函数名会在\$0变量中定义，函数命令行上的任何参数都会通过\$1、\$2等定义。也可以用特殊变量\$#来判断传给函数的参数数目。

在脚本中指定函数时，必须将参数和函数放在同一行，像这样：

```
func1 $value1 10
```

然后函数可以用参数环境变量来获得参数值。这里有个使用此方法向函数传值的例子：

```
$ cat test6
#!/bin/bash
# passing parameters to a function

function addem {
    if [ $# -eq 0 ] || [ $# -gt 2 ]
    then
        echo -1
    elif [ $# -eq 1 ]
    then
        echo ${!1} + ${!1}
    else
        echo ${!1} + ${!2}
    fi
}
echo -n "Adding 10 and 15: "
```

```

value='addem 10 15'
echo $value
echo -n "Let's try adding just one number: "
value='addem 10'
echo $value
echo -n "Now trying adding no numbers: "
value='addem'
echo $value
echo -n "Finally, try adding three numbers: "
value='addem 10 15 20'
echo $value
$ ./test6
Adding 10 and 15: 25
Let's try adding just one number: 20
Now trying adding no numbers: -1
Finally, try adding three numbers: -1
$
```

text6脚本中的addem函数首先会检查脚本传给它的参数数目。如果没有任何参数，或者如果有多个参数，addem会返回值-1。如果只有一个参数，addem会将参数自己加到自己上面生成结果。如果有两个参数，addem会将它们加起来生成结果。

由于函数使用特殊参数环境变量作为自己的参数值，它不能直接从脚本的命令行获取脚本的参数值。下面的例子将会运行失败：

```

$ cat badtest1
#!/bin/bash
# trying to access script parameters inside a function

function badfunc1 {
    echo ${$1 * $2}
}

if [ $# -eq 2 ]
then
    value='badfunc1'
    echo "The result is $value"
else
    echo "Usage: badtest1 a b"
fi
$ ./badtest1
Usage: badtest1 a b
$ ./badtest1 10 15
./badtest1: * : syntax error: operand expected (error token is "*")
$ The result is
$
```

尽管函数使用了\$1和\$2变量，但它们却和脚本主体中的\$1和\$2变量不尽相同。要在函数中使用这些值，必须在调用函数时手动将它们传过去：

```
$ cat test7
#!/bin/bash
# trying to access script parameters inside a function

function func7 {
    echo ${1}*${2}
}

if [ $# -eq 2 ]
then
    value=`func7 $1 $2`
    echo "The result is $value"
else
    echo "Usage: badtest1 a b"
fi
#
$ ./test7
Usage: badtest1 a b
$ ./test7 10 15
The result is 150
$
```

通过将\$1和\$2变量传给函数，它们就能跟其他变量一样，供函数使用了。

16.3.2 在函数中处理变量

给shell脚本程序员经常带来麻烦的事是变量的作用域。作用域是什么情况下变量可见。函数中定义的变量可以跟普通变量的作用域不同，也就是说，它们可以在脚本的其他部分隐藏起来。

函数会用两种类型的变量：

- 全局变量；
- 局部变量。

下面几节将会介绍如何在函数中使用这两种类型的变量。

1. 全局变量

全局变量是在shell脚本中任何地方都有效的变量。如果你在脚本的主体部分定义了一个全局变量，那么你可以在函数内读取它的值。类似地，如果你在函数内定义了一个全局变量，你可以在脚本的主体部分读取它的值。

默认情况下，你在脚本中定义的任何变量都是全局变量。在函数外定义的变量可在函数内正常访问：

```
$ cat test8
#!/bin/bash
# using a global variable to pass a value

function dbl {
    value=${value}*2
}

read -p "Enter a value: " value
```

```
db1
echo "The new value is: $value"
$
$ ./test8
Enter a value: 450
The new value is: 900
$
```

\$value变量是在函数外定义的，并在函数外被赋值了。当db1函数被调用时，变量及其值在函数中都依然有效。如果变量在函数内被赋予了新值，那么在脚本中引用该变量时，新值依然有效。

但这其实是很危险的事情，尤其是如果你想在不同的shell脚本中使用函数的话。它要求你知道函数中具体使用了哪些变量，包括那些用来计算并没有返回到脚本的值的变量。这里有个例子来说明事情是如何搞砸的：

```
$ cat badtest2
#!/bin/bash
# demonstrating a bad use of variables

function func1 {
    temp=$[ $value + 5 ]
    result=$[ $temp * 2 ]
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
$
$ ./badtest2
The result is 22
temp is larger
$
```

由于\$temp变量在函数中用到了，它的值在脚本中使用时，产生了一个意想不到的结果。有个简单的办法可以在函数中解决这个问题，下节将会介绍。

2. 局部变量

不用在函数中使用全局变量，函数内部使用的任何变量都可以被声明成局部变量。要那么做时，只要在变量声明的前面加上local关键字就可以了：

```
local temp
```

也可以在给变量赋值时在赋值语句中使用local关键字：

```
local temp=$(( $value + 5 ))
```

`local`关键字保证了变量只局限在该函数中。如果脚本中在该函数之外有同样名字的变量，那么shell将会保持这两个变量的值是分离的。现在你就能很轻松地让函数变量和脚本变量分离开来，只共用想共用的：

```
$ cat test9
#!/bin/bash
# demonstrating the local keyword

function func1 {
    local temp=${value}+5
    result=$((temp * 2))
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
$ ./test9
The result is 22
temp is smaller
$
```

现在在`func1`函数中使用`$temp`变量时，并不会影响在主体脚本中赋给`$temp`变量的值。

16.4 数组变量和函数

第5章讨论了使用数组来在单个变量中保持多个值的高级方法。在函数中使用数组变量值稍微麻烦一些，而且还需要作一些特殊考虑。本节将会介绍一种方法来解决这个问题。

16.4.1 向函数传数组参数

向脚本函数传递数组变量的方法会有点不好理解。将数组变量当做单个参数传递的话，它不会起作用：

```
$ cat badtest3
#!/bin/bash
# trying to pass an array variable

function testit {
    echo "The parameters are: $@"
    thisarray=$@
    echo "The received array is ${thisarray[@]}"
}
```

```
myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
testit ${myarray}
$ 
$ ./badtest3
The original array is: 1 2 3 4 5
The parameters are: 1
./badtest3: thisarray[*]: bad array subscript
The received array is
$
```

如果你试图将该数组变量当成一个函数参数，函数只会取数组变量的第一个值。

要解决这个问题，你必须将该数组变量的值分解成单个值然后将这些值作为函数参数使用。在函数内部，你可以将所有的参数重组到新的数组变量中。下面是个具体的例子：

```
$ cat test10
#!/bin/bash
# array variable to function test

function testit {
    local newarray
    newarray=(${!#})
    echo "The new array value is: ${newarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
testit ${myarray[*]}
$ 
$ ./test10
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
$
```

该脚本用\$myarray变量来保存所有的单个数组值，然后将它们都放在该函数的命令行上。之后该函数从命令行参数重建该数组变量。在函数内部，数组仍然可以像其他数组一样使用：

```
$ cat test11
#!/bin/bash
# adding values in an array

function addarray {
    local sum=0
    local newarray
    newarray=(${!#})
    for value in ${newarray[*]}
    do
        sum=$(( $sum + $value ))
    done
    echo $sum
}

myarray=(1 2 3 4 5)
```

```

echo "The original array is: ${myarray[*]}"
arg1=`echo ${myarray[*]}`
result=`addarray $arg1`
echo "The result is $result"
$
$ ./test11
The original array is: 1 2 3 4 5
The result is 15
$
```

addarray函数会遍历所有的数组值，将它们加在一起。你可以在myarray数组变量中放置任意多的值，addarry函数会将它们都加起来。

16.4.2 从函数返回数组

从函数里向shell脚本回传数组变量也用类似的方法。函数用echo语句来按正确顺序输出单个数组值，然后脚本再将它们重新放进一个新的数组变量中：

```

$ cat test12
#!/bin/bash
# returning an array value

function arraydbl {
    local origarray
    local newarray
    local elements
    local i
    origarray=(`echo "$@"`)
    newarray=(`echo "$@"`)
    elements=${# - 1}
    for (( i = 0; i <= $elements; i++ ))
    {
        newarray[$i]=${origarray[$i]} * 2
    }
    echo ${newarray[*]}
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=`echo ${myarray[*]}`
result=`arraydbl $arg1`
echo "The new array is: ${result[*]}"
$
$ ./test12
The original array is: 1 2 3 4 5
The new array is: 2 4 6 8 10
```

该脚本用\$arg1变量将数组值传给arraydbl函数。arraydbl函数将该数组重组到新的数组变量中，生成该输出数组变量的一个副本。之后，它遍历了数组变量中的单个值，将每个值翻倍，并将结果放到函数中该数组变量的副本中。

之后，arraydbl函数使用echo语句来输出该数组变量值中的每个值。脚本用arraydbl函数

的输出来用这些值重新生成一个新的数组变量。

16.5 函数递归

局部函数变量提供的一个特性是自成体系（self-containment）。自成体系的函数不需要使用任何外部资源，除了脚本在命令行上传给它的变量。

这个特性使得函数可以递归地调用，也就是说函数可以调用自己来得到结果。通常递归函数都有一个最终可以迭代到的基准值。许多高级数学算法用递归来一级一级地降解复杂的方程，直到到基准值定义的那级。

递归算法的经典例子是计算阶乘。一个数的阶乘是该数之前的所有数乘以该数的值。因此，要计算5的阶乘，你可以执行如下方程：

$$5! = 1 * 2 * 3 * 4 * 5 = 120$$

使用递归，方程可以简化成以下形式：

$$x! = x * (x-1)!$$

也就是说， x 的阶乘等于 x 乘以 $x-1$ 的阶乘。这可以用简单的递归脚本表达为：

```
function factorial {
    if [ $1 -eq 1 ]
    then
        echo 1
    else
        local temp=${$1 - 1}
        local result=`factorial $temp`
        echo ${result}*${$1}
    fi
}
```

阶乘函数用它自己来计算阶乘的值：

```
$ cat test13
#!/bin/bash
# using recursion

function factorial {
    if [ $1 -eq 1 ]
    then
        echo 1
    else
        local temp=${$1 - 1}
        local result=`factorial $temp`
        echo ${result}*${$1}
    fi
}

read -p "Enter value: " value
result=`factorial $value`
echo "The factorial of $value is: $result"
```

```
$  
$ ./test13  
Enter value: 5  
The factorial of 5 is: 120  
$
```

使用阶乘函数很容易。创建了这样的函数后，你可能想在其他脚本中使用它。下一步，我们将会了解如何有效地使用它。

16.6 创建库

很容易发现，在单个脚本中，使用函数可以省去一些键入的工作，但如果你碰巧要在脚本间使用同一个代码块呢？显然，在每个脚本中都定义同样的函数而只使用一次会比较麻烦。

有个方法能解决这个问题。`bash shell`允许创建函数库文件，然后在需要时在多个脚本中引用该文件。

这个过程的第一步是创建一个包含脚本中所需函数的公用库文件。这里有个简单的称为`myfuncs`的库文件，它定义了3个简单的函数：

```
$ cat myfuncs  
# my script functions  
  
function addem {  
    echo $[ $1 + $2 ]  
}  
  
function multem {  
    echo $[ $1 * $2 ]  
}  
  
function divem [  
    if [ $2 -ne 0 ]  
    then  
        echo $[ $1 / $2 ]  
    else  
        echo -1  
    fi  
]  
$
```

下一步是在要用这些函数的脚本文件中包含`myfuncs`库文件。从这里开始，事情变得复杂起来。

问题出在`shell`函数的作用域上。和环境变量一样，`shell`函数只对定义它的`shell`会话有效。如果你在`shell`命令行界面的提示符下运行`myfuncs`脚本，`shell`会创建一个新的`shell`并在新`shell`中运行这个脚本。它会为那个新`shell`定义这3个函数，但当你运行另外一个用到这些函数的脚本时，它们却不可用。

这同样适用于脚本。如果你尝试将该库文件当做普通脚本文件运行，函数不会出现在脚本中：

```
$ cat badtest4
#!/bin/bash
# using a library file the wrong way
./myfuncs

result=`addem 10 15`
echo "The result is $result"
$
$ ./badtest4
./badtest4: addem: command not found
The result is
$
```

使用函数库的关键在于source命令。source命令会在当前的shell上下文中执行命令，而不是创建一个新的shell来执行命令。可以用source命令来在shell脚本中运行库文件脚本。这样函数就对脚本可用。

source命令有个快捷别名，称作点操作符（dot operator）。要在shell脚本中运行myfuncs库文件，你只需添加下面这行：

```
./myfuncs
```

这个例子假定myfuncs库文件和shell脚本位于同一目录。如果不是，你需要使用相应路径访问该文件。这里有个用myfuncs库文件创建脚本的例子：

```
$ cat test14
#!/bin/bash
# using functions defined in a library file
./myfuncs

value1=10
value2=5
result1=`addem $value1 $value2`
result2=`multem $value1 $value2`
result3=`divem $value1 $value2`
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
$
$ ./test14
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
```

该脚本成功地使用了myfuncs库文件中定义的函数。

16.7 在命令行上使用函数

可以用脚本函数来创建一些十分复杂的操作。有时，在命令行界面的提示符下直接使用这些命令也很有必要。

和在shell脚本中将脚本函数当命令使用一样，你也可以在命令行界面上将脚本函数当命令用。这是一个很好的功能，因为一旦你在shell中定义了函数，你就可以在整个系统上直接用它了，而无需担心脚本是不是在PATH环境变量里。重点在于让shell找到这些函数。有几种方法可以实现。

16.7.1 在命令行上创建函数

由于在键入命令时shell就会解释命令，你可以在命令行上直接定义一个函数。有两种方法：一种方法是在一行内定义整个函数：

```
$ function divem { echo ${ $1 / $2 }: }
$ divem 100 5
20
$
```

当你在命令行上定义函数时，你必须记住在每个命令后面加个分号，这样shell就能知道在那里分开命令了：

```
$ function doubleit { read -p "Enter value: " value; echo $[
$ value * 2 ]: }
$ doubleit
Enter value: 20
40
$
```

另一种方法是用多行来定义函数。在定义时，bash shell会使用次提示符来提示输入更多命令。用这种方法，你不用在每条命令的末尾放一个分号，只要按下回车键就行：

```
$ function multem {
> echo ${ $1 * $2 }
>
$ multem 2 5
10
$
```

在函数的尾部使用花括号，shell就会知道你已经完成了函数的定义。

警告 在命令行上创建函数时要特别小心。如果你给函数起了个跟内建命令或另一个命令相同的名字，函数将会覆盖原来的命令。

16.7.2 在.bashrc 文件中定义函数

在命令行上直接定义shell函数的明显缺点是当退出shell时，函数就消失了。对于复杂的函数来说，这可能会是个问题。

一个简单得多的方法是将在一个每次启动新shell时都会加载的地方定义函数。

最好的地方就是.bashrc文件。bash shell会在每次启动时在主目录查找这个文件，不管是交互

式的还是从现有shell中启动一个新的shell。

1. 直接定义函数

可以直接在主目录的.bashrc文件中定义函数。许多Linux发行版已经在.bashrc文件中定义了一些东西了，所以注意别删掉这些内容，只要在已有文件的末尾加上你写的函数就行了。这里有个例子：

```
$ cat .bashrc
# .bashrc

# Source global definitions
if [ -r /etc/bashrc ]; then
    . /etc/bashrc
fi

function addem {
    echo $[ $1 + $2 ]
}
$
```

直到下次启动新bash shell时该函数才会生效。添加并重启bash shell后，你就能在系统上任意地方使用该函数了。

2. 读取函数文件

只要是在shell脚本中，你都可以用source命令（或者它的别名点操作符）来将已有库文件中的函数添加到你的.bashrc脚本中：

```
$ cat .bashrc
# .bashrc

# Source global definitions
if [ -r /etc/bashrc ]; then
    . /etc/bashrc
fi

. /home/rich/libraries/myfuncs
$
```

确保包含了引用库文件的正确路径名，以便于bash shell来查找该文件。下次启动shell时，库中的所有函数都可在命令行界面下使用了：

```
$ addem 10 5
15
$ multem 10 5
50
$ divem 10 5
2
$
```

更好的一点是，shell还会将定义好的函数传给子shell进程，这样这些函数在该shell会话中的任何shell脚本中也都可用。你可以写个脚本来测试，直接使用这些函数而不用单独定义或读取它们：

```
$ cat test15
#!/bin/bash
# using a function defined in the .bashrc file

value1=10
value2=5
result1=`addem $value1 $value2`
result2=`multem $value1 $value2`
result3=`divem $value1 $value2`
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
$
$ ./test15
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
$
```

甚至不用读取库文件，这些函数也能在shell脚本中很好地工作。

16.8 小结

shell脚本函数允许你将脚本中多处用到的脚本代码放到一个地方。你可以创建一个包含该代码块的函数然后在脚本中通过函数名来引用这块代码，而不用一次次地重复那段代码。`bash shell`只要看到脚本中用的那个函数名，就会自动跳到函数代码块处。

你甚至可以创建能返回值的函数。这样你就能创建和脚本交互的函数，返回数字和字符串数据。脚本函数可以用函数中最后一条命令的退出状态码或`return`命令来返回数值。`return`命令可以基于函数的结果，通过编程将函数的退出状态码设为特定值。

函数也可以用标准的`echo`语句来返回值。你可以跟其他`shell`命令一样用反引号来获取输出的数据。这样你就能从函数中返回任意类型的数据了，包括字符串和浮点数。

你可以在函数中使用`shell`变量，对其赋值以及从中取值。这样你就能将任何类型的数据从主体脚本程序的脚本函数中传入传出。函数也支持定义只能在函数代码块内访问的局部变量。局部变量使得用户可以创建自成体系的函数，这样就不会影响主体`shell`脚本中用到的变量和进程了。

函数也可以调用其他函数，包括它自身。函数的自调用行为称为递归。递归函数通常有个作为函数终结条件的基准值。函数会一直用降低的参数值调用自身，直到达到基准值。

如果在`shell`脚本中用到了大量函数的话，你可以创建脚本函数库文件。库文件可以用`source`命令在任何`shell`脚本文件中引用，这也称为读取库文件。`shell`不会运行该脚本文件，但会使这些函数在运行该脚本的`shell`中一直有效。可以用同样的方法创建在普通`shell`命令行上使用的函数。可以直接在命令行上定义函数，或者将它们加到`.bashrc`文件中，每次启动新`shell`时它们就可用。这样你就不用管`PATH`环境变量设置成什么，而可以直接创建要用的工具了。

下一章将会介绍脚本中文本图形的使用。在现代化图形界面普及的今天，有时只有普通的文本界面是不够的。`bash shell`提供了一些简便的方法来将简单的图形功能加入到脚本中。

本章内容

- 创建文本菜单
- 创建文本窗口部件
- 添加X Window图形

多 年来，shell脚本一直都被认为是枯燥乏味的。但如果你准备在图形化环境中运行脚本时，情况就不同了。有很多并不依赖read和echo语句的和脚本用户交互的方式。本章将会介绍一些可以让交互式脚本更友好一些的不同方法，这样它们看起来就不那么古板了。

17.1 创建文本菜单

创建交互式shell脚本最常用的方法是使用菜单。提供各种选项可以帮助脚本用户了解脚本能做什么，不能做什么。

通常菜单脚本会清空显示区域，然后显示可用的选项列表。用户可以按下与每个选项关联的字母或数字来选择选项。图17-1显示了示例菜单的布局。



图17-1 在shell脚本中显示菜单

shell脚本菜单的核心是case命令（参见第11章）。case命令会根据用户在菜单上的选择来执行特定命令。

后面几节将会带你逐步了解创建基于菜单的shell脚本的步骤。

17.1.1 创建菜单布局

创建菜单的第一步显然是决定在菜单上显示哪些元素以及将它们按你想要显示的方式布局。

在创建菜单前，通常要先清理一下显示器上的输出。这样就能在干净的、没有干扰文本的环境中显示菜单了。

clear命令用当前终端会话的terminfo数据（参见第2章）来清理出现在屏幕上的文本。运行clear命令之后，可以用echo命令来显示菜单元素。

默认情况下，echo命令只显示可打印文本字符。在创建菜单项时，通常非可打印字符也很有用，比如制表符和换行符。要在echo命令中包含这些字符时，必须用-e选项。因此，命令：

```
echo -e "1.\tDisplay disk space"
```

会生成如下输出行：

```
1.      Display disk space
```

这极大地方便了菜单项布局的格式化。用一些echo命令，你就能创建一个看上去还行的菜单了：

```
clear
echo
echo -e "\t\t\tSys Admin Menu\n"
echo -e "\t1. Display disk space"
echo -e "\t2. Display logged on users"
echo -e "\t3. Display memory usage"
echo -e "\t0. Exit menu\n\n"
echo -en "\tEnter option: "
```

最后一行的-en选项会让显示的最后一行末尾不加换行符。这让菜单看上去更专业一些，而光标会一直在行尾等待用户的输入。

创建菜单的最后一步是从用户那里得到输入。这步我们用read命令（参见第13章）。因为我们期望只有单字符输入，所以在read命令中用了-n选项来限定只取一个字符。这样用户输入一个数字就行，不用按回车键了：

```
read -n 1 option
```

下一步，你需要创建自己的菜单函数了。

17.1.2 创建菜单函数

shell脚本菜单选项作为一组独立的函数创建起来更为容易。这样你就能创建简洁方便、容易理解的case命令了。

要那么做，你需要针对每个菜单选项创建独立的shell函数。创建shell菜单脚本的第一步是决定你想在脚本中执行哪些函数，然后将它们当做独立函数在代码中分布开来。

通常我们会为还没有实现的函数先创建一个桩函数（stub function）。桩函数是还没有任何命令的函数，或者只有echo语句来说明最终那里需要什么命令：

```
function diskspace {
    clear
    echo "This is where the diskspace commands will go"
}
```

这样，在你还在编写单个函数时，菜单仍然能正常操作。你不需要写出所有函数之后才能让菜单工作。你会注意到函数从clear命令开始。这样，你就能在一个干净的屏幕上开始该函数，而不会再显示菜单。

shell脚本菜单中还有一件有用的事是将菜单布局本身作为一个函数来创建：

```
function menu {
    clear
    echo
    echo -e "\t\t\tSys Admin Menu\n"
    echo -e "\t1. Display disk space"
    echo -e "\t2. Display logged on users"
    echo -e "\t3. Display memory usage"
    echo -e "\t0. Exit program\n\n"
    echo -en "\t\tEnter option: "
    read -n 1 option
}
```

这样，任何时候你都能调用menu函数来重新显示菜单。

17.1.3 添加菜单逻辑

现在你已经建好了菜单布局和函数，下面你需要做的就是创建程序逻辑来将二者联系在一起。前面提到过，这需要case命令。

case命令应该根据菜单中输入的字符选择来调用相应的函数。通常，用默认的case命令字符（星号）来捕捉所有不正确的菜单项是个很好的习惯。

下面的代码展示了典型菜单中case命令的使用：

```
menu
case $option in
0)
    break ;;
1)
    diskspace ;;
2)
    whoseon ;;
3)
    memusage ;;
*)
    clear
```

```
echo "Sorry, wrong selection":;
esac
```

这段代码首先用menu函数清空屏幕并显示菜单。menu函数中的read命令会一直等待，直到用户在键盘上键入了字符。一旦这个过程完成，case命令就会接管了。case命令会基于返回的字符调用相应的函数。在函数运行结束后，case命令退出。

17.1.4 整合shell脚本菜单

现在你已经看到了构成shell脚本菜单的所有内容，让我们将它们放在一起，看看如何整合它们。这里有个完整的菜单脚本的例子：

```
$ cat menu1
#!/bin/bash
# simple script menu

function diskspace {
    clear
    df -k
}

function whoson {
    clear
    who
}

function memusage {
    clear
    cat /proc/meminfo
}

function menu {
    clear
    echo
    echo -e "\t\t\tSys Admin Menu\n"
    echo -e "\t1. Display disk space"
    echo -e "\t2. Display logged on users"
    echo -e "\t3. Display memory usage"
    echo -e "\t0. Exit program\n\n"
    echo -en "\t\tEnter option: "
    read -n 1 option
}

while [ 1 ]
do
    menu
    case $option in
        0)
            break ;;
        1)
            diskspace ;;
        2)
```

```

whoseon ::

3)
memusage ::

*)
    clear
    echo "Sorry, wrong selection":"
esac
echo -en "\n\n\t\tHit any key to continue"
read -n 1 line
done
clear
$
```

这个菜单创建了3个函数来用常见命令提取Linux系统的管理信息。它会用while循环来不停地循环这个菜单，直到用户选择了选项0，这时，它会用break命令来跳出这个while循环。

你可以用同样的模板来创建shell脚本菜单界面。它提供了一个简单的跟用户交互的途径。

17.1.5 使用select命令

你可能已经注意到，创建文本菜单的一半工夫都花在了创建菜单布局和获取输入的字符上。bash shell提供了一个很容易上手的小工具来自动完成这些工作。

select命令允许从单个命令行创建菜单，然后再提取输入的答案并自动处理。select命令的格式如下：

```

select variable in list
do
    commands
done
```

list参数是构成菜单的空格分割的文本选项列表。select命令会在列表中将每个选项作为一个编号号的选项显示，然后为选项显示一个特殊的由PS3环境变量定义的提示符。

这里有个实用中select命令的简单例子：

```

$ cat smenu1
#!/bin/bash
# using select in the menu

function diskspace {
    clear
    df -k
}

function whoseon {
    clear
    who
}

function memusage {
    clear
    cat /proc/meminfo
```

```

}

PS3="Enter option: "
select option in "Display disk space" "Display logged on users"
"Display memory usage" "Exit program"
do
    case $option in
        "Exit program")
            break ;;
        "Display disk space")
            diskspace ;;
        "Display logged on users")
            whoson ;;
        "Display memory usage")
            memusage ;;
    *)
        clear
        echo "Sorry, wrong selection";;
esac
done
clear
$
```

运行这个程序时，它会自动生成如下菜单：

```

$ ./smenu1
1) Display disk space      3) Display memory usage
2) Display logged on users 4) Exit program
Enter option:
```

在使用select命令时，记住，存储在变量中的结果值是整个文本字符串而不是跟菜单选项相關的数字。文本字符串值是你要在case语句中比较的。

17.2 使用窗口

使用文本菜单是方向正确的一步，但在交互脚本中仍然欠缺很多东西，尤其是相比图形化窗口而言。幸运的是，开源界有些很聪明的人帮我们都已经做好了。

dialog包是最早由Savio Lam创建的一个小巧的工具，现在由Thomas E. Dickey维护。该包能够在文本环境中用ANSI转义控制字符来创建标准的窗口对话框。本节将会介绍dialog包并演示如何在shell脚本中使用它。

说明 并非在所有的Linux发行版中都会默认安装dialog包。即使默认情况下未安装，鉴于它的流行程度，你也几乎总能在软件库中找到它。查看Linux发行版的文档来了解如何下载dialog包。在Ubuntu Linux发行版中，下面的命令行命令用来安装它：

```
sudo apt-get install dialog
```

这条命令将会为你的系统安装dialog包以及需要的库。

17.2.1 dialog包

dialog命令使用命令行参数来决定生成哪种窗口部件（widget）。部件是dialog包中窗口元素类型的术语。dialog包现在支持表17-1中的部件类型。

表17-1 dialog部件

部 件	描 述
calendar	提供选择日期的日历
checkbox	显示多个选项（其中每个选项都能打开或关闭）
form	构建一个表单（用标签和文本字段来填充）
fselect	提供一个文件选择窗口来浏览选择文件
gauge	显示完成的百分比进度条
infobox	显示一条消息，但不用等待回应
inputbox	提供一个输入文本用的文本表单
inputmenu	提供一个可编辑的菜单
menu	显示可选择的一系列选项
msgbox	显示一条消息，并要求用户选择OK按钮
pause	显示一个进度条来显示特定暂定时间的状态
passwordbox	显示一个文本框来输入文本，但会隐藏输入的文本
passwordform	显示一个带标签和隐藏文本输入的表单
radiolist	提供一组菜单选项，但只能选择其中一个
tailbox	用tail命令在滚动窗口中显示文件的内容
tailboxbg	跟tailbox一样，但是在后台模式中运行
textbox	在滚动窗口中显示文件的内容
timebox	提供一个选择小时、分钟和秒数的窗口
yesno	提供一条简单的带Yes和No按钮的消息

如你在表17-1中看到的，我们可以选择很多不同的部件。这可以让你只花一点工夫，但脚本看起来更专业。

要在命令行上指定某个特定的部件，你需要使用双破折线格式：

```
dialog --widget parameters
```

其中*widget*是表17-1中的部件名，*parameters*定义了部件窗口的大小以及部件需要的文本。

每个dialog部件都提供了两种格式的输出：

- 使用STDERR；
- 使用退出状态码。

dialog命令的退出状态码决定了用户选择的按钮。如果选择了Yes或OK按钮，dialog命令会返回退出状态码0。如果选择了Cancel或No按钮，dialog命令会返回退出状态码1。可以用标准的

\$?变量来查看dialog部件中具体选择了哪个按钮。

如果部件返回了任何数据，比如菜单选择，那么dialog命令会将数据发送到STDERR。可以用标准的bash shell方法来将STDERR输出重定向到另一个文件或文件描述符中：

```
dialog --inputbox "Enter your age:" 10 20 2>age.txt
```

这个命令会将文本框中输入的文本重定向到age.txt文件中。

后面几节将会看一下shell脚本中更经常用到的一些dialog部件。

1. msgbox部件

msgbox部件是对话框中最常见的类型。它会在窗口中显示一条简单的消息，并等待用户单击OK按钮后才消失。使用msgbox部件时要用下面的格式：

```
dialog --msgbox text height width
```

text参数是你想在窗口中显示的字符串。dialog命令会根据由height和width参数创建的窗口的大小来自动换行。如果想在窗口顶部放一个标题，你也可以用--title参数，后跟作为标题的文本。这里有个使用msgbox部件的例子：

```
$ dialog --title Testing --msgbox "This is a test" 10 20
```

在输入这条命令后，消息框会显示在你所用的终端模拟器会话的屏幕上。图17-2就是它看上去的样子。



图17-2 在dialog命令中使用msgbox

如果你的终端模拟器支持鼠标，你可以单击OK按钮来关闭对话框。你也可以用键盘命令来模拟单击动作——按下回车键。

2. yesno部件

yesno部件将msgbox部件延伸一步，允许用户回答窗口中显示的yes/no问题。它会在窗口底部生成两个按钮，一个是Yes，一个是No。用户可以用鼠标、制表符键或者键盘方向键来切换按钮。要选择按钮的话，用户可以按下空格键或者回车键。

这里有个使用yesno部件的例子：

```
$ dialog --title "Please answer" --yesno "Is this thing on?" 10 20
$ echo $?
1
$
```

这会产生如图17-3中所示的部件。

dialog命令的退出状态码会根据用户选择的按钮来设置。如果用户选择了No按钮，退出状态码是1；如果选择了Yes按钮，退出状态码就是0。



图17-3 在dialog命令中使用yesno部件

3. inputbox部件

inputbox部件为用户提供了一个简单的文本框区域来输入文本字符串。dialog命令会将文本字符串的值发给STDERR。你必须重定向STDERR来获取用户输入。图17-4显示了inputbox部件看起来的样子。

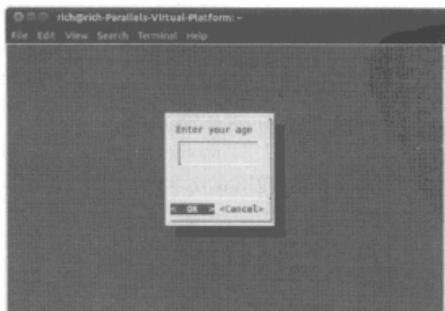


图17-4 inputbox部件

如图17-4所示，inputbox提供了两个按钮，OK和Cancel。如果选择了OK按钮，命令的退出状态码就是1；反之，退出状态码就会是0：

```
$ dialog --inputbox "Enter your age:" 10 20 >age.txt
$ echo $?
0
$ cat age.txt
12$
```

你会注意到，在使用cat命令显示文本文件的内容时，在该值后并没有换行符。这让你能够轻松地将文件内容重定向到shell脚本中的变量里，以提取用户输入的字符串。

4. textbox部件

textbox部件是在窗口中显示大量信息的极佳办法。它会生成一个滚动窗口来显示参数中指定文件中的文本：

```
$ dialog --textbox /etc/passwd 15 45
/etc/passwd文件的内容会显示在可滚动的文本窗口中，如图17-5所示。
```

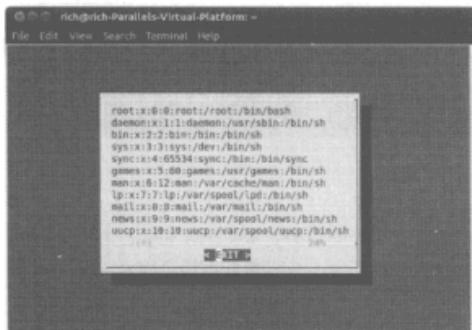


图17-5 textbox部件

可以用方向键来左右或上下滚动文件的内容。窗口底部的行会显示文本在你所查看文件中位置的百分比。文本框只包含一个用来选择退出部件的Exit按钮。

5. menu部件

menu部件允许你来创建我们在本章前面创建的文本菜单的窗口版本。你只要为每个选项提供一个选择标号和文本就行了：

```
$ dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space"
2 "Display users" 3 "Display memory usage" 4 "Exit" 2> test.txt
```

第一个参数定义了菜单的标题，之后的两个参数定义了菜单窗口的高和宽，而第4个参数定义了一次在窗口中显示的菜单项总数。如果有更多的选项，可以用方向键来滚动显示它们。

在这些参数后面，你必须添加菜单项对。第一个元素是用来选择菜单项的标号。每个标号对每个菜单项都应该是唯一的，可以通过在键盘上按下对应的键来选择。第二个元素是菜单中使用的文本。图17-6展示了由示例命令生成的菜单。

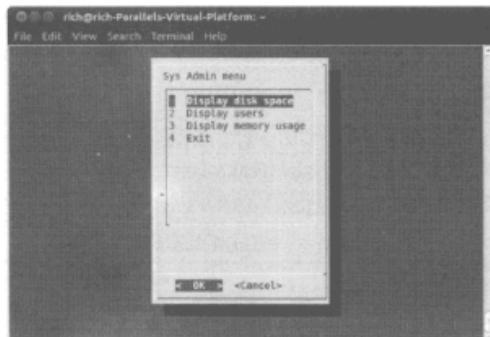


图17-6 带有菜单项的menu部件

如果用户通过按下标号对应的键选择了某个菜单项，则那个菜单项会高亮显示但不会直接选定。直到用户用鼠标或回车键选择了OK按钮时，选项才会最终选定。dialog命令会将选定的菜单项文本发送到STDERR。你可以根据需要重定向STDERR。

6. fselect部件

dialog命令提供了几个非常炫的内置部件。fselect部件在处理文件名时非常方便。不用强制用户键入文件名，你可以用fselect部件来浏览文件的位置并选择文件，如图17-7所示。



图17-7 fselect部件

fselect部件的格式看起来如下：

```
$ dialog --title "Select a file" --fselect $HOME/ 10 50 2>file.txt
```

fselect选项后的第一个参数是窗口中使用的起始目录位置。fselect部件窗口由左侧的目录列表、右侧的显示了左侧选定目录的所有文件列表和含有当前选定的文件或目录的简单文本框组成。可以手动在文本框键入文件名，或者用目录和文件列表来选定一个。

17.2.2 dialog选项

除了标准部件，你还可以在dialog命令中定制很多不同的选项。你已经在实例中看了--title参数。它允许你给部件设置一个标题，在窗口顶部显示。

还有许多允许完全定制窗口外观和操作的其他选项。表17-2显示了dialog命令中可用的选项。

表17-2 dialog命令选项

选 项	描 述
--add-widget	继续下个对话框，直到按下Esc或Cancel按钮
--aspect ratio	指定窗口宽度和高度的宽高比
--backtitle title	指定显示在屏幕顶部背景上的标题
--begin x y	指定窗口左上角的起始位置
--cancel-label label	指定Cancel按钮的替代标签
--clear	用默认对话背景色来清空显示
--colors	在对话文本中嵌入ANSI色彩编码
--cr-wrap	在对话文本中允许使用换行符并强制换行
--create-rc file	将示例配置文件的内容复制到指定的file文件中 ^①
--defaultno	将yes/no对话的默认答案设为No
--default-item string	设定复选列表、表单或菜单对话中的默认项
--exit-label label	指定Exit按钮的替代标签
--extra-button	在OK按钮和Cancel按钮中显示一个额外按钮
--extra-label label	指定额外按钮的替代标签
--help	显示dialog命令的帮助信息
--help-button	在OK按钮和Cancel按钮后显示一个Help按钮
--help-label label	指定Help按钮的替代标签
--help-status	当选定Help按钮时，会在帮助信息后写入多选列表、单选列表或表单信息
--ignore	忽略dialog不能识别的选项

① dialog命令支持运行时配置，该命令会根据配置文件模板创建一份配置文件。dialog启动时会先去检查是否设置了DIALOGRC环境变量，该变量会保存配置文件名信息。如果未设置该变量或未找到该文件，它会将\$HOME/.dialogrc作为配置文件。如果这个文件还不存在的话，就尝试查找编译时指定的GLOBALRC文件，也就是/etc/dialogrc。如果这个文件也不存在的话，就用编译时的默认值。——译者注

(续)

选 项	描 述
--input-fd <i>fd</i>	指定另一个文件描述符，而不是STDIN
--insecure	在password部件中键入时显示星号
--item-help	为多选列表、单选列表或菜单中的每个标号在屏幕的底部添加一个帮助栏
--keep-window	不要清除屏幕上显示过的部件
--max-input-size	指定输入的最大字符串长度。默认为2048
--nocancel	隐藏Cancel按钮
--no-collapse	在对话文本中不要将制表符转换成空格
--no-kill	将tailboxbg对话放到后台，并禁止该进程的SIGHUP信号
--no-label <i>label</i>	为No按钮指定替代标签
--no-shadow	不要显示对话窗口的投影效果
--ok-label <i>label</i>	指定OK按钮的替代标签
--output-fd <i>fd</i>	指定另一个输出文件描述符，而不是STDERR
--print-maxsize	将对话窗口的最大尺寸打印到输出中
--print-size	将每个对话窗口的大小打印到输出中
--print-version	将dialog的版本打印到输出中
--separate-output	一次一行地输出checkbox部件的结果，而不用引号
--separator <i>string</i>	为每个部件指定分割输出的字符串
--separate-widget <i>string</i>	为每个部件指定分割输出的字符串
--shadow	在每个窗口的右下角绘制阴影
--single-quoted	需要时对多选列表的输出采用单引号
--sleep <i>sec</i>	在处理完对话窗口之后延迟指定的秒数
--stderr	将输出发送到STDERR（默认就是这样的）
--stdout	将输出发送到STDOUT
--tab-correct	将制表符转换成空格
--tab-len <i>n</i>	指定一个制表符占用的空格数（默认为8）
--timeout <i>sec</i>	指定无用户输入的话 <i>sec</i> 秒后退出并返回错误代码
--title <i>title</i>	指定对话窗口的标题
--trim	从对话文本中删除前面的空格和换行符
--visit-items	修改对话窗口中制表符的停留位置，使其包括选项列表
--yes-label <i>label</i>	为Yes按钮指定替代标签

--backtitle选项是为脚本中的菜单创建公共标题的简便办法。如果你为每个对话窗口都指定了该选项，那么它在你的应用中就会保持一致，让脚本看起来更专业。

由表17-2可知，可以覆盖对话窗口中的任意按钮标签。这个特性允许你创建几乎任何需要的窗口。

17.2.3 在脚本中使用dialog命令

在脚本中使用dialog命令非常迅捷。你必须记住两件事：

- 如果有Cancel或No按钮，检查dialog命令的退出状态码；
- 重定向STDERR来获得输出值。

如果你遵循了这两个规则，那么你的交互脚本就马上看起来很专业了。这里有个使用dialog部件来生成本章前面我们创建的系统管理菜单的例子：

```
$ cat menu3
#!/bin/bash
# using dialog to create a menu

temp=`mktemp -t test.XXXXXX`
temp2=`mktemp -t test2.XXXXXX`

function diskspace {
    df -k > $temp
    dialog --textbox $temp 20 60
}

function whoseon {
    who > $temp
    dialog --textbox $temp 20 50
}

function memusage {
    cat /proc/meminfo > $temp
    dialog --textbox $temp 20 50
}

while [ 1 ]
do
    dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space" 2
    "Display users" 3 "Display memory usage" 0 "Exit" 2> $temp2
    if [ $? -eq 1 ]
    then
        break
    fi

    selection=`cat $temp2` 

    case $selection in
    1)
        diskspace ;;
    2)
        whoseon ;;
    3)
        memusage ;;
    0)
        break ;;
    *)
```

```

dialog --msgbox "Sorry, invalid selection" 10 30
esac
done
rm -f $temp 2> /dev/null
rm -f $temp2 2> /dev/null
$
```

这段脚本用while循环和一个常量真值创建了个无限循环来显示菜单对话。这意味着，在每个函数之后，脚本会返回继续显示菜单。

由于menu对话包含了一个Cancel按钮，脚本会检查dialog命令的退出状态码，以防用户按下Cancel按钮退出。因为它是而在while循环中，所以退出该菜单就跟用break命令跳出while循环一样简单。

脚本用mktemp命令创建两个临时文件来保存dialog命令的数据。第一个临时文件\$temp用来保存df和meminfo命令的输出，这样就能在textbox对话中显示它们了（如图17-8所示）。第二个临时文件\$temp2用来保存从主菜单对话中选定的值。

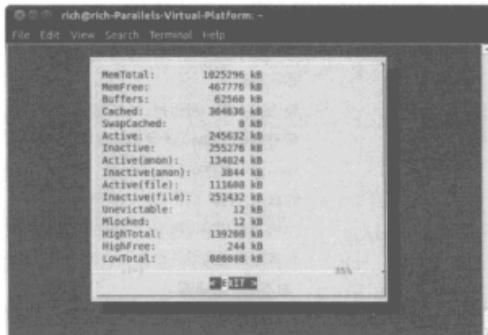


图17-8 用textbox对话选项显示的meminfo命令输出

现在这看起来像个可以给别人展示的真实程序了。

17.3 使用图形

如果想给交互脚本加入更多的图形元素，可以再进一步。KDE和GNOME桌面环境（参见第1章）都扩展了dialog命令的想法，包含用来在各自对应的环境中生成X Window图形化部件的命令。

本节将描述kdialog和zenity包，它们各自为KDE和GNOME桌面提供了图形化窗口部件。

17.3.1 KDE环境

KDE图形化环境默认包含kdialog包。kdialog包使用kdialog命令来在KDE桌面上生成类似于

dialog式部件的标准窗口。但这些窗口能跟你的其他KDE应用窗口很好地融合，不会产生笨重的感觉。这允许你直接在shell脚本中创建可以和Windows相媲美的用户界面。

说明 你的Linux发行版使用KDE桌面并不代表它就默认安装了kdialog包。你可能需要从发行版的软件库中手动安装。

1. kdialog部件

和dialog命令完全类似，kdialog命令使用命令行选项来指定具体使用哪种类型的窗口部件。下面是kdialog命令的格式：

```
kdialog display-options window-options arguments
```

window-options选项允许指定使用哪种类型的窗口部件。可用的选项如表17-3所示。

表17-3 kdialog窗口选项

选 项	描 述
--checklist title [tag item status]	多选列表菜单，状态会说明该选项是否被选定
--error text	错误消息框
--inputbox text [init]	输入文本框。可以用init值来指定默认值
--menu title [tag item]	带有标题（title）的菜单选择框，以及用tag标识的选项列表
--msgbox text	显示指定文本的简单消息框
--password text	隐藏用户输入的密码输入文本框
--radiolist title [tag item status]	单选列表菜单，状态会说明该选项是否被选定
--separate-output	为多选列表和单选列表菜单返回按行分开的选项
--sorry text	“对不起”消息框
--textbox file [width] [height]	显示file的内容的文本框，另外指定了width和height
--title title	为对话窗口的TitleBar区域指定一个标题
--warningyesno text	带有Yes和No按钮的警告消息框
--warningcontinuecancel text	带有Continue和Cancel按钮的警告消息框
--warningyesnocancel text	带有Yes、No和Cancel按钮的警告消息框
--yesno text	带有Yes和No按钮的提问框
--yesnocancel text	带有Yes、No和Cancel按钮的提问框

如你在表17-3中所见，所有的标准窗口对话框类型都在那里。但在使用kdialog窗口部件时，它会看起来更像是KDE桌面上的独立窗口，而不是终端模拟器会话中的。

checkbox和radiolist部件允许你在列表中定义单独的选项以及它们默认是否选定：

```
$kdialog --checkbox "Items I need" 1 "Toothbrush" on 2 "Toothpaste"
off 3 "Hair brush" on 4 "Deodorant" off 5 "Slippers" off
```

最终的多选列表窗口如图17-9所示。



图17-9 kdialog多选列表对话窗口

指定为“on”的选项会在多选列表中高亮显示。要选择或取消选择多选列表中的某个选项，只要单击它就行了。如果你选择了OK按钮，kdialog就会将标号值发到STDOUT上：

```
"1" "3" "5"
$
```

当你按下次回车键时，kdialog窗口框就和选定选项一起出现了。当你单击OK或Cancel按钮时，kdialog命令会将每个标号作为一个字符串值返回到STDOUT（这些就是你在输出中看到的“1”、“3”和“5”）。脚本必须能解析结果值并将它们和原始值匹配起来。

2. 使用kdialog

你可以在shell脚本中使用kdialog窗口部件，方法类似于你如何使用dialog部件。最大的不同是kdialog窗口部件用STDOUT来输出值，而不是STDERR。

以下是一个将前面创建的系统管理菜单转换成KDE应用的脚本：

```
$ cat menu4
#!/bin/bash
# using kdialog to create a menu

temp=`mktemp -t temp.XXXXXX`
temp2=`mktemp -t temp2.XXXXXX`

function diskspace {
    df -k > $temp
    kdialo --textbox $temp 1000 10
}

function whoseon {
    who > $temp
    kdialo --textbox $temp 500 10
}

function menusage {
```

```

cat /proc/meminfo > $temp
kdialog --textbox $temp 300 500
}

while [ 1 ]
do
kdialog --menu "Sys Admin Menu" "1" "Display diskspace" "2" "Display
users" "3" "Display memory usage" "0" "Exit" > $temp2
if [ $? -eq 1 ]
then
break
fi

selection=`cat $temp2` 

case $selection in
1)
diskspace ::;
2)
whoseon ::;
3)
memusage ::;
0)
break ::;
*)
kdialog --msgbox "Sorry, invalid selection"
esac
done
$
```

使用kdialog命令和dialog命令在脚本中并无太大区别。生成的主菜单如图17-10所示。

现在你的那个简单shell脚本看起来就像个真的KDE应用，你在交互式脚本中的操作已经没有任何局限了。

17.3.2 GNOME环境

GNOME图形化环境支持两个流行的可生成标准窗口的包：

- gdialog;
- zenity。

到目前为止，zenity是大多数GNOME桌面Linux发行版上最经常安装的包（在Ubuntu和Fedora上默认安装）。本节将会介绍zenity的功能并演示如何在脚本中使用它。

1. zenity部件

如你所期望的，zenity允许用命令行选项创建不同的窗口部件。表17-4列出了zenity能够生成的不同部件。



图17-10 采用kdialog的系统管理菜单脚本

表17-4 zenity窗口部件

选 项	描 述
--calendar	显示整月日历
--entry	显示文本输入对话窗口
--error	显示错误消息对话窗口
--file-selection	显示完整的路径名和文件名对话窗口
--info	显示信息对话窗口
--list	显示多选列表或单选列表对话窗口
--notification	显示通知图标
--progress	显示进度条对话窗口
--question	显示yes/no对话窗口
--scale	显示可调整大小的窗口
--text-info	显示含有文本的文本框
--warning	显示警告对话窗口

zenity命令行程序与kdialog和dialog程序的工作方式有些不同。许多部件类型都用命令行上的其他选项定义，而不是将它们包括在选项的参数里。

zenity命令能够提供一些非常炫的对话窗口。calendar选项会产生一个整月日历，如图17-11所示。

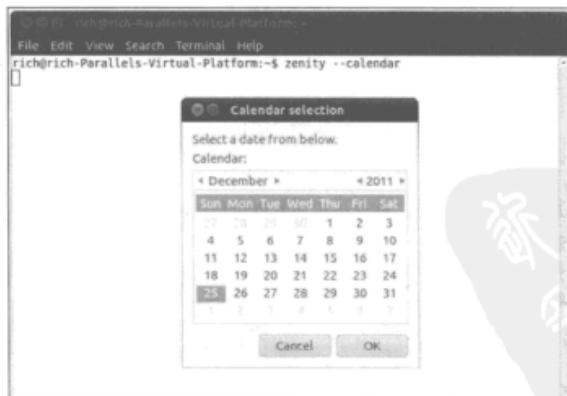


图17-11 zenity日历对话窗口

当你在日历中选择了日期时，zenity命令会将值返回到STDOUT中，就跟kdialog一样：

```
$ zenity --calendar
12/25/2011
$
```

zenity中另一个很酷的窗口是文件选择选项，如图17-12所示。

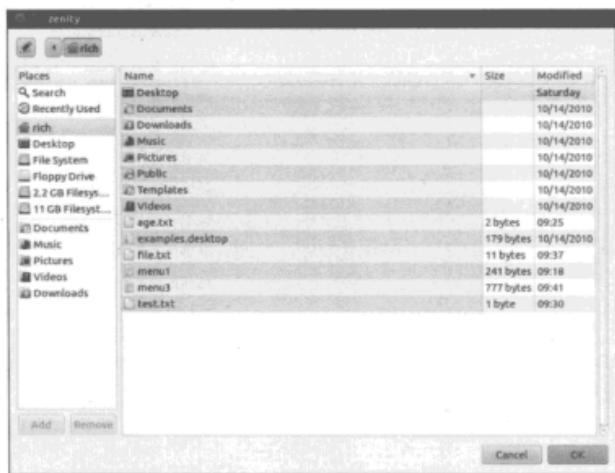


图17-12 zenity文件选择对话窗口

你可以用对话窗口来浏览系统上任意一个目录位置（只要你有查看该目录的权限）并选择文件。当你选定文件时，zenity命令会返回完整的文件路径名：

```
$ zenity --file-selection
/home/ubuntu/menu5
$
```

有了这样可以任意发挥的工具，创建shell脚本就没什么限制了。

2. 在脚本中使用zenity

如你所期望的，zenity在shell脚本中运行得非常顺利。但是，zenity没有沿袭dialog和kdialog中所采用的选项惯例，因此要将已有的交互式脚本迁移到zenity上要花点工夫。

在将系统管理菜单从kdialog迁移到zenity的过程中，我们发现需要对部件定义做相当一部分工作：

```
$cat menu5
#!/bin/bash
# using zenity to create a menu

temp=`mktemp -t temp.XXXXXX`
```

```

temp2=`mktemp -t temp2.XXXXXX` 

function diskspace {
    df -k > $temp
    zenity --text-info --title "Disk space" --filename=$temp
    --width 750 --height 10
}

function whoseon {
    who > $temp
    zenity --text-info --title "Logged in users" --filename=$temp
    --width 500 --height 10
}

function memusage {
    cat /proc/meminfo > $temp
    zenity --text-info --title "Memory usage" --filename=$temp
    --width 300 --height 500
}

while [ 1 ]
do
zenity --list --radiolist --title "Sys Admin Menu" --column "Select"
--column "Menu Item" FALSE "Display diskspace" FALSE "Display users"
FALSE "Display memory usage" FALSE "Exit" > $temp2
if [ $? -eq 1 ]
then
    break
fi

selection=`cat $temp2`
case $selection in
"Display disk space")
    diskspace ;;
"Display users")
    whoseon ;;
"Display memory usage")
    memusage ;;
Exit)
    break ;;
*)
    zenity --info "Sorry, invalid selection"
esac
done
$ 

```

由于zenity并不支持菜单对话窗口，我们改用单选列表窗口来作为主菜单，如图17-13所示。该单选列表用了两列，每列都有一个头。第一列含有选定用的单选按钮。第二列则是选项文本。单选列表也不用选项里的标号。当你选定一个选项时，该选项的所有文本都会返回到STDOUT。这会让case命令的内容丰富一些。必须在case选项中使用选项中的全文本。如果文本中有任何空格，你需要给文本加上引号。

使用zenity包，你可以给GNOME桌面上的交互式shell脚本带来一种Windows式的体验。

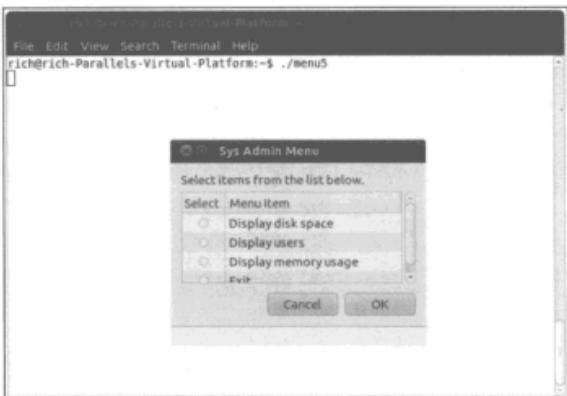


图17-13 采用zenity的系统管理菜单

17.4 小结

交互式shell脚本因为枯燥乏味而声名狼藉。你可以在多数Linux系统上使用一些不同技术和工具来改变这个状况。首先，你可以用case命令和shell脚本函数来为你的交互式脚本创建菜单系统。

case命令允许你用标准的echo命令来绘制菜单，然后用read命令来读取用户输入。之后case命令会选择根据输入值来选择对应的shell脚本函数。

dialog程序提供了一些预建的文本部件来在基于文本的终端模拟器上创建类Windows对象。你可以用dialog程序创建对话框来显示文本、输入文本以及选择文件和日期。这会让你的脚本生动许多。

如果是在图形化X Window环境中运行shell脚本，你可以在交互脚本中用更多的工具。对KDE桌面来说，有kdialog程序。该程序提供了简单命令来为所有基本窗口功能创建窗口部件。对GNOME桌面来说，有gdialog和zenity程序。每个程序都提供了能像真实Windows应用一样融入GNOME桌面的窗口部件。

下一章将深入介绍编辑和处理文本数据文件方面的内容。通常shell脚本最大的用途就在分析和显示文本文件中的数据，比如日志文件和错误文件。Linux环境包含了两个非常有用的工具，sed和gawk，来在shell脚本中处理文本数据。下一章将介绍这些工具并演示如何使用它们。

本章内容

 文本处理 sed编辑器基础

到目前为止，shell脚本最常见一个用途是处理文本文件。可以检查日志文件、可以读取配置文件以及处理数据元素，shell脚本可以帮助自动化处理文本文件中各种类型数据的日常任务。但只用shell脚本命令来处理文本文件的内容有点勉为其难。如果想在shell脚本中处理任何类型的数据，你应该了解一下Linux中的sed和gawk工具。这两个工具能够极大地简化需要进行的数据处理任务。

18.1 文本处理

第9章演示了如何用Linux环境中常带的不同编辑器程序来编辑文本文件。这些编辑器使你能够用简单命令或鼠标单击来轻松地处理文本文件中的文本。

但总有一些时候，你会发现需要自动处理文本文件中的文本，而不需要拉出全副武装的交互式文本编辑器。在这种情况下，有个能够自动地格式化、插入、修改或删除文本元素的简单命令行编辑器就方便多了。

Linux系统提供了两个常见的具备上述功能的编辑器。本节将会介绍Linux世界中最广泛使用的两个命令行编辑器：sed和gawk。

18.1.1 sed编辑器

sed编辑器被称作流编辑器（stream editor），跟普通交互式文本编辑器恰好相反。在交互式文本编辑器中（比如Vim），你可以用键盘命令来交互式地插入、删除或替换数据中的文本。流编辑器则会在编辑器处理数据之前基于预先提供的一组规则来编辑数据流。

sed编辑器可以基于输入到命令行的或是存储在命令文本文件中的命令来处理数据流中的数据。它每次从输入中读取一行，用提供的编辑器命令匹配数据、按命令中指定的方式修改流中的数据，然后将生成的数据输出到STDOUT。在流编辑器将所有命令与一行数据进行匹配后，它会读

取下一行数据并重复这个过程。在流编辑器处理完流中的所有数据行后，它就会终止。

由于命令都是一行一行顺序处理的，sed编辑器必须一次就完成对文本的修改。这使得sed编辑器比交互式编辑器快很多，这样你就能很快地完成对数据的自动修改了。

使用sed命令的格式如下：

```
sed options script file
```

选项参数允许你修改sed命令的行为，同时还包含表18-1中列出的选项。

表18-1 sed命令选项

选 项	描 述
-e script	在处理输入时，将script中指定的命令添加到运行的命令中
-f file	在处理输入时，将file中指定的命令添加到运行的命令中
-n	不要为每个命令生成输出，等待print命令来输出

script参数指定了将作用在流数据上的单个命令。如果需要用多个命令，你必须用-e选项来在命令行上指定它们，或用-f选项来在单独的文件中指定。有大量的命令可用来处理数据。我们将会在本章后面介绍一些sed编辑器中用到的基本命令，然后在第20章中看另外一些高级命令。

1. 在命令行定义编辑器命令

默认情况下，sed编辑器会将指定的命令应用到STDIN输入流上。这样你可以直接将数据管道输出到sed编辑器上处理。这里有个演示如何做的简短例子：

```
$ echo "This is a test" | sed 's/test/big test/'  
This is a big test  
$
```

这个例子在sed编辑器中使用了s命令。s命令会用斜线间指定的第二个文本字符串来替换第一个文本字符串。在本例中，big test替换了test。

在运行这个例子时，它应该立即就显示了结果。这就是使用sed编辑器的强大之处。可以在几乎和交互式编辑器的启动时间一样的时间内就对数据作多处修改。

当然，这个简单的测试只修改了一行数据。在修改多处文件数据时，你应该能得到差不多快的结果：

```
$ cat data1  
The quick brown fox jumps over the lazy dog.  
The quick brown fox jumps over the lazy dog.  
The quick brown fox jumps over the lazy dog.  
The quick brown fox jumps over the lazy dog.  
$  
$ sed 's/dog/cat/' data1  
The quick brown fox jumps over the lazy cat.  
The quick brown fox jumps over the lazy cat.  
The quick brown fox jumps over the lazy cat.  
The quick brown fox jumps over the lazy cat.  
$
```

sed命令几乎瞬间就执行完并返回数据。在它处理每行数据的同时，结果也显示出来了。可以在sed编辑器处理完整个文件之前就开始观察结果。

重要的是要记住，sed编辑器自身不会修改文本文件的数据。它只会将修改后的数据发送到STDOUT。如果你查看原来的文本文件，它仍然保留着原始数据：

```
$ cat data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
```

2. 在命令行使用多个编辑器命令

要在sed命令行上执行多个命令时，只要用-e选项就可以了：

```
$ sed -e 's/brown/green/; s/dog/cat/' data1
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
$
```

两个命令都可以作用到文件中的每行数据上。命令之间必须用分号分隔，并且在命令末尾和分号之间不能有空格。

也可以用bash shell中的次提示符来分隔命令，而不用分号。只要输入第一个单引号来开始编写，bash会继续提示你输入更多命令，直到你输入了封尾的单引号：

```
$ sed -e '
> s/brown/green/
> s/fox/elephant/
> s/dog/cat/' data1
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

必须记住，要在封尾单引号所在行结束命令。bash shell一旦发现了封尾的单引号，就会执行命令。开始后，sed命令就会将你指定的每条命令应用到文本文件中的每一行上。

3. 从文件中读取编辑器命令

最后，如果有大量要处理的sed命令，将它们放进一个文件中通常会更方便一些。可以在sed命令中用-f选项来指定文件：

```
$ cat script1
s/brown/green/
s/fox/elephant/
s/dog/cat/
$
$ sed -f script1 data1
The quick green elephant jumps over the lazy cat.
```

```
The quick green elephant jumps over the lazy cat.  
The quick green elephant jumps over the lazy cat.  
The quick green elephant jumps over the lazy cat.  
$
```

在这种情况下，不用在每条命令后面放一个分号。sed编辑器知道每行都有一条单独的命令。跟在命令行输入命令一样，sed编辑器会从指定文件中读取命令，并将它们应用到数据文件中的每一行上。

我们会在18.2节中继续介绍另外一些便于处理数据的sed编辑器命令。在那之前，我们先快速了解一下其他的Linux数据编辑器。

18.1.2 gawk程序

虽然sed编辑器是自动修改文本文件的非常方便的工具，但它也有自身的限制。通常你需要一个用来处理文件中的数据的更高级工具，它能提供一个类编程环境，允许修改和重新组织文件中的数据。这正是gawk的特点。

gawk程序是Unix中的原始awk程序的GNU版本。gawk程序让流编辑迈上了一个新的台阶，它提供了一种编程语言而不只是编辑器命令。在gawk编程语言中，你可以做下面的事：

- 定义变量来保存数据；
- 使用算术和字符串操作符来处理数据；
- 使用结构化编程概念，比如if-then语句和循环，来为数据处理增加逻辑；
- 提取数据文件中的数据元素并将它们按另一顺序或格式重新放置，从而生成格式化报告。

gawk程序的报告生成能力通常用来从大文本文件中提取数据元素并将它们格式化成可读的报告。最完美的例子是格式化日志文件。在日志文件中找出错误行会很难，gawk程序允许从日志文件中只过滤出你要看的数据元素，并以某种更容易读取重要数据的方式将它们格式化。

1. gawk命令格式

gawk程序的基本格式如下：

```
gawk options program file
```

表18-2显示了gawk程序的可用选项。

表18-2 gawk选项

选 项	描 述
-F fs	指定行中分隔数据字段的字段分隔符
-f file	指定读取程序的文件名
-v var=value	定义gawk程序中的一个变量及其默认值
-mf N	指定要处理的数据文件中的最大字段数
-mr N	指定数据文件中的最大数据行数
-W keyword	指定gawk的兼容模式或警告等级

命令行选项提供了一个简单的途径来定制gawk程序中的功能。我们会在探索gawk时进一步了解这些选项。

gawk的强大之处在于程序脚本。你可以写脚本来读取文本行的数据，然后处理并显示数据，创建任何类型的输出报告。

2. 从命令行读取程序脚本

gawk程序脚本用一对花括号来定义。你必须将脚本命令放到两个括号中。由于gawk命令行假定脚本是单个文本字符串，你必须将脚本放到单引号中。下面是个在命令行上指定简单的gawk程序脚本的例子：

```
$ gawk '{print "Hello John!"}'
```

这个程序脚本会定义一个命令，即print命令。print命令名副其实：它会将文本打印到STDOUT。如果你尝试运行这个命令，你可能会有些失望，因为不会有立即发生。因为并没有在命令行上指定文件名，gawk程序会从STDIN接收数据。在运行这个程序时，它会一直等待从STDIN输入的文本。

如果你输入一行文本并按下次车键，gawk会对这行文本运行一遍所有的程序脚本：

```
$ gawk '{print "Hello John!"}'
This is a test
Hello John!
hello
Hello John!
This is another test
Hello John!
```

\$

跟sed编辑器一样，gawk程序会针对数据流中的每行文本执行一遍程序脚本。由于程序脚本被设为显示固定的文本字符串，因而不管你在数据流中输入什么文本，你都会得到同样的文本输出。

要终止gawk程序，你必须发出信号说明数据流已经结束了。bash shell提供了一对组合键来生成EOF（End-of-File）字符。Ctrl+D组合键会在bash中产生一个EOF字符。使用这对组合键就能终止gawk程序并返回到命令行界面提示符下。

3. 使用数据字段变量

gawk的基本特性之一是它处理文本文件中数据的能力。它会自动给每行中的每个数据元素分配一个变量。默认情况下，gawk会将如下变量分配给它在文本行中发现的每个数据字段：

- \$0代表整个文本行；
- \$1代表文本行中的第1个数据字段；
- \$2代表文本行中的第2个数据字段；
- \$n代表文本行中的第n个数据字段。

每个数据字段在文本行中都是通过字段分隔符来划分的。gawk读取一行文本时，它会用预定的字段分隔符划分每个数据字段。gawk中默认的字段分隔符是任意的空白字符（例如空格或制

表符)。

下面是个gawk程序读取文本文件并显示第1数据字段值的例子：

```
$ cat data3
One line of test text.
Two lines of test text.
Three lines of test text.
$
$ gawk '{print $1}' data3
One
Two
Three
$
```

该程序用\$1字段变量来仅显示每行文本的第1个数据字段。

如果你要读取用其他字段分隔符的文件，可以用-F选项指定：

```
$ gawk -F: '{print $1}' /etc/passwd
root
daemon
bin
sys
sync
games
man
lp
mail
...
```

这个简短的程序显示了系统上密码文件的第1个数据字段。由于/etc/passwd文件用冒号来分隔数字字段，因而如果要分隔开每个数据元素，则必须在gawk选项中将它指定为字段分隔符。

4. 在程序脚本中使用多个命令

如果某编程语言只能执行一条命令，那么它不会有太大用处。gawk编程语言允许你将多条命令组合成一个正常的程序。要在命令行上的程序脚本中使用多条命令，只要在每条命令之间放个分号即可：

```
$ echo "My name is Rich" | gawk '{$4="Christine"; print $0}'
My name is Christine
$
```

第一条命令会将一个值赋给\$4字段变量。第二条命令会打印整个数据字段。注意，输出中gawk程序已经将原文本中的第4个数据字段替换成新值。

也可以用次提示符来一次一行地输入程序脚本命令：

```
$ gawk '{
> $4="testing"
> print $0 }'
This is not a good test.
This is not testing good test.
$
```

在你用了开始的单引号后，bash shell会出现次提示符来提示你输入更多数据。你可以每次在

每行加一条命令，直到你输入了结尾的单引号。要退出程序，只要按下Ctrl+D组合键来表明数据结束。

5. 从文件中读取程序

跟sed编辑器一样，gawk编辑器允许将程序存储到文件中，然后再在命令行中引用：

```
$ cat script2
{ print $1 "'s home directory is " $6 }
$
$ gawk -F: -f script2 /etc/passwd
root's home directory is /root
daemon's home directory is /usr/sbin
...
Samantha's home directory is /home/Samantha
Timothy's home directory is /home/Timothy
Christine's home directory is /home/Christine
$
```

script2程序脚本会再次使用print命令来打印/etc/passwd文件的主目录数据字段（字段变量\$6），以及userid数据字段（字段变量\$1）。

可以在程序文件中指定多条命令。要这么做的话，只要将每条命令放到一个新的行就好了，不需要用分号：

```
$
$ cat script3
{
text = "'s home directory is "
print $1 text $6
}
$
$ gawk -F: -f script3 /etc/passwd
root's home directory is /root
daemon's home directory is /usr/sbin
...
Samantha's home directory is /home/Samantha
Timothy's home directory is /home/Timothy
Christine's home directory is /home/Christine
$
```

script3程序脚本定义了一个变量来保存print命令中用到的文本字符串。你会注意到，gawk程序在引用变量值时并未像shell脚本一样使用美元符。

6. 在处理数据前运行脚本

gawk还允许指定程序脚本何时运行。默认情况下，gawk会从输入中读取一行文本，然后针对文本行的数据执行程序脚本。有时可能需要在处理数据前运行脚本，比如为报告创建开头部分。BEGIN关键字就是用来做这个的。它会强制gawk在读取数据前执行BEGIN关键字后指定的程序脚本：

```
$ gawk 'BEGIN {print "Hello World!"}'
Hello World!
$
```

这次print命令会在读取任何数据前显示文本。但在它显示了文本后，它会快速退出而不用

等待任何数据。这样的原因是在gawk处理任何数据前，BEGIN关键字只执行指定的脚本。要想在正常的程序脚本中处理数据，必须用另一个脚本段来定义程序：

```
$  
$ cat data4  
Line 1  
Line 2  
Line 3  
$  
$ gawk 'BEGIN { print "The data4 File Contents:" } { print $0 }' data4  
The data4 File Contents:  
Line 1  
Line 2  
Line 3  
$
```

现在在gawk执行了BEGIN脚本后，它会用第二段脚本来处理任何文件数据。这么做时要小心，注意两段脚本在gawk命令行上会被当成一个文本字符串。你需要相应地加上单引号。

7. 在处理数据后运行脚本

跟BEGIN关键字类似，END关键字允许你指定一个程序脚本，gawk会在读完数据后执行它：

```
$  
$ gawk 'BEGIN { print "The data4 File Contents:" } { print $0 }  
END { print "End of File" }' data4  
The data4 File Contents:  
Line 1  
Line 2  
Line 3  
End of File  
$
```

当gawk程序完成打印文件内容后，它会执行END脚本中的命令。这是在处理完所有正常数据后给报告添加结尾部分的最佳方法。

你可以将所有这些内容放到一起，组成一个很小的程序脚本文件来从简单的数据文件创建一份完整的报告：

```
$ cat script4  
BEGIN {  
print "The latest list of users and shells"  
print " Userid      Shell"  
print "-----      -----"  
FS=":";  
}  
  
{  
print $1 "      $"7"  
}  
  
END {  
print "This concludes the listing"  
}  
$
```

这个脚本用BEGIN脚本来为报告创建开头部分。它还定义了一个称作FS的特殊变量。这是定义字段分隔符的另一种方法。这样你就不用依赖脚本用户来在命令行选项中定义字段分隔符了。

这里有段运行gawk程序脚本时截取的输出：

```
$ gawk -f script4 /etc/passwd
The latest list of users and shells
Userid      Shell
-----
root        /bin/bash
daemon      /bin/sh
bin         /bin/sh
...
Samantha   /bin/bash
Timothy     /bin/sh
Christine   /bin/sh
This concludes the listing
$
```

与预想的一样，BEGIN脚本创建了开头的文本，程序脚本处理了指定数据文件（/etc/passwd）中的信息，END脚本生成了结尾的文本。

这让你体验到了使用简单gawk脚本的威力。第21章介绍了另外一些编写gawk脚本时的简单原则，以及一些更高级的可用在gawk程序脚本中的编程概念，你可以利用这些概念来从哪怕是最神秘的数据文件中生成看起来很专业的报告。

18.2 sed 编辑器基础

成功使用sed编辑器的关键在于掌握它各式各样的能帮你定制文本编辑行为的命令和格式。本节将介绍一些能集成到脚本中方便使用sed编辑器的基本命令和功能。

18.2.1 更多的替换选项

你已经懂得了如何用s命令来用新文本替换一行内的文本。但还有一些其他的substitute命令选项能让事情变得更为简单。

1. 替换标记

关于substitute命令如何替换字符串中匹配的模式需要注意一点。看看下面这个例子中会出现什么情况：

```
$ cat data5
This is a test of the test script.
This is the second test of the test script.
$
$ sed 's/test/trial/' data5
This is a trial of the test script.
This is the second trial of the test script.
```

substitute命令在替换多行中的文本时能正常工作，但默认情况下它只替换每行中出现的第一

一处。要让替换命令对一行中不同地方出现的文本都起作用，必须使用替换标记（substitution flag）。替换标记会在替换命令字符串之后设置：

s/pattern/replacement/flags

有4种可用的替换标记：

- 数字，表明新文本将替换第几处模式匹配的地方；
- g，表明新文本将会替换所有已有文本出现的地方；
- p，表明原来行的内容要打印出来；
- w file，将替换的结果写到文件中。

在第一类替换中，你可以指定sed编辑器用新文本替换第几处模式匹配的地方：

```
$ sed 's/test/trial/2' data5
This is a test of the trial script.
This is the second test of the trial script.
$
```

将替换标记指定为2的结果是，sed编辑器只替换每行中第二次出现的匹配模式。g替换标记使你能替换文本中每处匹配模式出现的地方：

```
$ sed 's/test/trial/g' data5
This is a trial of the trial script.
This is the second trial of the trial script.
$
```

p替换标记会打印包含与subsitute命令中指定的模式匹配的行。这通常会和sed的-n选项一起使用：

```
$ cat data6
This is a test line.
This is a different line.
$
$ sed -n 's/test/trial/p' data6
This is a trial line.
$
```

-n选项将禁止sed编辑器输出。但p替换标记会输出修改过的行。将二者配合使用则会只输出被substitute命令修改过的行。

w替换标记会产生同样的输出，不过会将输出保存到指定文件中：

```
$ sed 's/test/trial/w test' data6
This is a trial line.
This is a different line.
$
$ cat test
This is a trial line.
$
```

sed编辑器的正常输出是在STDOUT中，而只有那些包含匹配模式的行才会保存在指定的输出文件中。

2. 替换字符

有时你会遇到一些文本字符串中的字符不方便在替换模式中使用的情况。Linux中一个流行的例子是正斜线。

替换文件中的路径名会比较麻烦。比如，如果想用C shell替换/etc/passwd文件中的bash shell，你必须这么做：

```
$ sed 's/\bin\/bash\bin\csh/' /etc/passwd
```

由于正斜线通常用作字符串分隔符，因而如果它出现在了模式文本中的话，你必须用反斜线来转义。这通常会带来一些困惑和错误。

要解决这个问题，sed编辑器允许选择其他字符来作为substitute命令中的字符串分隔符：

```
$ sed '!/bin/bash!/bin/csh!' /etc/passwd
```

在这个例子中，感叹号被用作字符串分隔符，使得路径名很容易被读取和理解。

18.2.2 使用地址

默认情况下，在sed编辑器中使用的命令会作用于文本数据的所有行。如果只想将命令作用于特定某行或某些行，你必须用行寻址（line addressing）。

在sed编辑器中有两种形式的行寻址：

- 行的数字范围；
- 用文本模式来过滤出某行。

两种形式都使用相同的格式来指定地址：

[address]command

也可以为特定地址将多个命令放在一起：

```
address {
    command1
    command2
    command3
}
```

sed编辑器会将指定的每条命令只作用到匹配指定地址的行上。

本节将会演示如何在sed编辑器脚本中使用两种寻址方法。

1. 数字方式的行寻址

当使用数字方式的行寻址时，你可以用它们在文本流中的行位置来引用行。sed编辑器会将文本流中的第一行分配为第一行，然后继续按顺序为新行分配行号。

在命令中指定的地址可以是单个行号，或是用起始行号、逗号以及结尾行号指定的一定范围内的行。这里有个sed命令作用到指定某行的例子：

```
$ sed '2s/dog/cat/' data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy dog
```

```
The quick brown fox jumps over the lazy dog
$
```

sed编辑器会只修改地址指定的第二行的文本。这里有另一个例子，这次使用了行地址范围：

```
$ sed '2,3s/dog/cat/' data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy dog
$
```

如果想将一条命令作用到文本中某行开始到结尾的所有行，你可以用特殊地址——美元符：

```
$ sed '$,2s/dog/cat/' data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
$
```

你可能不知道文本中到底有多少行数据，所以美元符用起来通常很方便。

2. 使用文本模式过滤器

另一种限制命令作用到哪些行上的方法会稍稍复杂一些。sed编辑器允许指定文本模式来过滤出命令要作用的行。格式如下：

```
/pattern/command
```

必须用正斜线将要指定的pattern封起来。sed编辑器会将该命令只作用到包含指定文本模式的行上。

举个例子，如果你想只修改用户Samantha的默认shell，你会这么用sed命令：

```
$
$ grep Samantha /etc/passwd
Samantha:x:1001:1002:Samantha,4...:/home/Samantha:/bin/bash
$
$ sed '/Samantha/s/bash/csh/' /etc/passwd
root:x:0:0:root:/root:/bin/bash
...
Samantha:x:1001:1002:Samantha,4...:/home/Samantha:/bin/csh
Timothy:x:1002:1005::/home/Timothy:/bin/sh
Christine:x:1003:1006::/home/Christine:/bin/sh
$
```

该命令只作用到匹配文本模式的行上。虽然使用固定文本模式能帮你过滤出特定的值，就跟上面这个用户名的例子一样，但你能做的通常有些局限。sed编辑器在文本模式中会采用一种称为正则表达式（regular expression）的特性来帮助创建能很好地匹配的模式。

正则表达式允许创建高级文本模式——匹配表达式来匹配各种数据。这些表达式会将一系列通配符、特殊字符以及固定文本字符组合到一起，生成一个几乎能匹配任何文本情形的简练模式。正则表达式是shell脚本编程中比较可怕的部分之一。第19章将会详细介绍。

3. 组合命令

如果需要在单行上执行多条命令，可以用花括号将多条命令组合在一起。sed编辑器会处理列在地址行的每条命令：

```
$ sed '2{
> s/fox/elephant/
> s/dog/cat/
> }' data1
The quick brown fox jumps over the lazy dog.
The quick brown elephant jumps over the lazy cat.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
```

两条命令都会作用到该地址上。当然，也可以在一组命令前指定一个地址范围：

```
$ sed '3|{
> s/brown/green/
> s/lazy/active/
> }' data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick green fox jumps over the active dog.
The quick green fox jumps over the active dog.
$
```

sed编辑器会将所有命令作用到该地址范围内的所有行上。

18.2.3 删 除 行

文本替换命令不是sed编辑器中的唯一命令。如果需要删除文本流中的特定行，可以用删除(delete)命令。

删除命令d名副其实，它会删除匹配指定寻址模式的所有行。使用删除命令时要特别小心，因为你忘记了加一个寻址模式的话，流中的所有文本行都会被删除：

```
$ cat data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
$ sed 'd' data1
$
```

很明显，删除命令在和指定地址一起使用时最有用。这允许你从数据流中删除特定的文本行，不管是通过行号指定：

```
$ sed '3d' data1
This is line number 1.
This is line number 2.
This is line number 4.
$
```

还是通过特定行范围指定：

```
$ sed '2,3d' data7
This is line number 1.
This is line number 4.
$
```

或者通过文件尾特殊字符：

```
$ sed '3,$d' data7
This is line number 1.
This is line number 2.
$
```

sed编辑器的模式匹配特性也适用于删除命令：

```
$ sed '/number 1/d' data7
This is line number 2.
This is line number 3.
This is line number 4.
$
```

sed编辑器会删掉包含匹配指定模式的文本的行。

说明 记住，sed编辑器不会修改原始文件。你删除的行只是从sed编辑器的输出中消失了。原始文件仍然包含那些“删掉的”行。

你可以删除用两个文本模式来删除某个范围内的行，但这么做时要小心。你指定的第一个模式会“打开”行删除功能，第二个模式会“关闭”行删除功能。sed编辑器会删除两个指定行之间所有的行（包括指定的行）：

```
$ sed '/1/,/3/d' data6
This is line number 4.
$
```

除此之外，你要特别小心，因为只要sed编辑器在数据流中匹配到了开始模式，删除功能就会打开。这可能会导致意外的结果：

```
$ cat data8
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is line number 1 again.
This is text you want to keep.
This is the last line in the file.
$
```

```
$ sed '/1/,/3/d' data7
This is line number 4.
$
```

第二个出现数字“1”的行再次触发了删除命令，删除了数据流中的剩余行，因为停止模式

再没找到。当然，如果你指定了一个从未在文本中出现的停止模式，会出现另一个明显的问题：

```
$ sed '/1/,/5/d' data8
$
```

因为删除功能在匹配到第一个模式的时候打开了，但一直没匹配到结束模式，所以整个数据流都被删掉了。

18.2.4 插入和附加文本

如你所期望的，跟其他编辑器类似，sed编辑器允许你向数据流插入和附加文本行。两个操作的区别可能比较费解：

- 插入（insert）命令*i*会在指定行前增加一个新行；
- 追加（append）命令*a*会在指定行后增加一个新行。

这两条命令的费解之处在于它们的格式。不能在单个命令行上使用这两条命令。你必须指定是要将行插入还是附加到另一行。格式如下：

```
sed '[address]command
new_line'
```

*new_line*中的文本将会出现在sed编辑器输出中你指定的位置。记住，当使用插入命令时，文本会出现在数据流文本的前面：

```
$ echo "Test Line 2" | sed 'i\Test Line 1'
Test Line 1
Test Line 2
$
```

当使用附加命令时，文本会出现在数据流文本的后面：

```
$ echo "Test Line 2" | sed 'a\Test Line 1'
Test Line 2
Test Line 1
$
```

在命令行界面提示符上使用sed编辑器时，你会看到次提示符来提醒输入新的行数据。你必须在该行完成sed编辑器命令。一旦你输入了结尾的单引号，bash shell就会执行该命令了：

```
$ echo "Test Line 2" | sed 'i\
> Test Line 1'
Test Line 1
Test Line 2
$
```

这样能够给数据流中的文本前面或后面添加文本，但如果要添加到数据流里面呢？

要给数据流中插入或附加数据，你必须用寻址来告诉sed编辑器你想让数据出现在什么位置。你可以在用这些命令时只指定一个行地址。可以匹配一个数字行号或文本模式，但不能用地址区间。这符合逻辑，因为你只能将文本插入或附加到单个行的前面或后面，而不能是行区间的前面或后面。

下面是将一个新行插入到数据流第3行前的例子：

```
$ sed '3i\
> This is an inserted line.' data7
This is line number 1.
This is line number 2.
This is an inserted line.
This is line number 3.
This is line number 4.
$
```

下面是将一个新行附加到数据流中第3行后面的例子：

```
$ sed '3a\
>This is an appended line.' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is an appended line.
This is line number 4.
$
```

它使用跟插入命令相同的过程，只是将新文本行放到了指定的行号后面。如果你有一个多行数据流，你要将新行附加到数据流的末尾，只要用代表数据最后一行的美元符就可以了：

```
$ sed '$a\
> This is a new line of text.' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is a new line of text.
$
```

同样的方法也适用于你要在数据流开始的地方增加一个新行。只要在第一行之前插入新行就可以了。

要插入或附加多行文本，你必须对新文本中的每一行使用反斜线，直到你要插入或附加的文本的最后一行：

```
$ sed '1i\
> This is one line of new text.\\
> This is another line of new text.' data7
This is one line of new text.
This is another line of new text.
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
```

指定的两行都会添加到数据流中。

18.2.5 修改行

修改 (change) 命令允许修改数据流中整行文本的内容。它跟插入和附加命令的工作机制一

样，你必须在sed命令中单独指定新行：

```
$ sed '3c\  
> This is a changed line of text.' data7  
This is line number 1.  
This is line number 2.  
This is a changed line of text.  
This is line number 4.  
$
```

在这个例子中，sed编辑器会修改第3行中的文本。你也可以用文本模式来寻址：

```
$ sed '/number 3/c\  
> This is a changed line of text.' data7  
This is line number 1.  
This is line number 2.  
This is a changed line of text.  
This is line number 4.  
$
```

文本模式修改命令会修改它匹配的数据流中的任意文本行。

```
$ sed '/number 1/c\  
> This is a changed line of text.' data8  
This is a changed line of text.  
This is line number 2.  
This is line number 3.  
This is line number 4.  
This is a changed line of text.  
This is yet another line.  
This is the last line in the file.  
$
```

你可以在修改命令中使用地址区间，但结果可能并不是你想要的：

```
$ sed '2,3c\  
> This is a new line of text.' data7  
This is line number 1.  
This is a new line of text.  
This is line number 4.  
$
```

sed编辑器会用这一行文本来替换数据流中的两行文本，而不是逐一修改那两行文本。

18.2.6 转换命令

转换（transform, y）命令是唯一可以处理单个字符的sed编辑器命令。转换命令格式如下：

[address]y/inchars/outchars/

转换命令会进行inchars和outchars值的一对一映射。inchars中的第一个字符会被转换为outchars中的第一个字符，第二个字符会被转换成outchars中的第二个字符。这个映射会一直持续到处理完指定字符。如果inchars和outchars的长度不同，则sed编辑器会产生一条错误消息。

这里有个使用转换命令的简单例子：

```
$ sed 'y/123/789/' data8
This is line number 7.
This is line number 8.
This is line number 9.
This is line number 4.
This is line number 7 again.
This is yet another line.
This is the last line in the file.
$
```

如你在输出中看到的，inchars模式中指定字符的每个实例都会被替换成outchars模式中相同位置的那个字符。

转换命令是一个全局命令，也就是说，它会自动替换文本行中找到的指定字符的所有实例，而不会考虑它们出现的位置：

```
$ echo "This 1 is a test of 1 try." | sed 'y/123/456/'
This 4 is a test of 4 try.
$
```

sed编辑器替换了在文本行中匹配到的字符“1”的两个实例。你无法限定只更改该字符出现的特定某个地方。

18.2.7 回顾打印

18.2.1节介绍了如何使用p标记和替换命令显示sed编辑器修改过的行。这里有3条也能用来打印数据流中的信息的命令：

- 小写p命令用来打印文本行；
- 等号(=)命令用来打印行号；
- l(L的小写)命令用来列出行。

1. 打印行

跟替换命令中的p标记类似，p命令可以打印sed编辑器输出中的一行。如果只用这个命令，也没什么特别的：

```
$ echo "this is a test" | sed 'p'
this is a test
this is a test
$
```

它打印出的数据文本是你已经知道的。打印命令最常见的用法是打印包含匹配文本模式的行：

```
$ sed -n '/number 3/p' data7
This is line number 3.
$
```

在命令行上用-n选项，你就能禁止其他行，只打印包含匹配文本模式的行。你也可以用它来快速打印数据流中的一些行：

```
$ sed -n '2,3p' data7
This is line number 2.
This is line number 3.
$
```

如果需要在修改之前查看行，那么你可以使用打印命令，比如与替换或修改命令一起使用。你可以创建一个脚本来在修改行之前显示该行：

```
$ sed -n '/3/{p
s/line/test/p
}' data7
This is line number 3.
This is test number 3.
$
```

sed编辑器命令会查找包含数字“3”的行，然后执行两条命令。首先，脚本用p命令来打印出该行原来的版本；然后它用s命令替换文本，并用p标记打印出了最终文本。输出同时显示了原来的行文本和新的行文本。

2. 打印行号

等号命令会打印行在数据流中的当前行号。行号由数据流中的换行符决定。每次数据流中出现一个换行符，sed编辑器会认为它结束了一行文本：

```
$ sed '=' data1
1
The quick brown fox jumps over the lazy dog.
2
The quick brown fox jumps over the lazy dog.
3
The quick brown fox jumps over the lazy dog.
4
The quick brown fox jumps over the lazy dog.
$
```

sed编辑器在实际的文本行出现前打印了行号。如果你要在数据流中查找特定文本模式的话，等号命令能派得上用场：

```
$ sed -n '/number 4/{=
=
p
}' data7
4
This is line number 4.
$
```

使用了-n选项，你就能让sed编辑器只显示包含匹配文本模式的行的行号和文本。

3. 列出行

列出命令(1)允许打印数据流中的文本和不可打印的ASCII字符。任何不可打印字符都用它们的八进制值前加一个反斜线或标准C风格的命名法(用于常见的不可打印字符)，比如\t来代表制表符：

```
$ cat data9
This    line    contains      tabs.
$ 
$ sed -n '1' data9
This\line\tcontains\ttabs.$
$
```

制表符的位置显示的是\t。行尾的美元符表明这里是换行符。如果数据流包含了转义字符，则list命令会用八进制码来显示它：

```
$ cat data10
This line contains an escape character
$ 
$ sed -n '1' data10
This line contains an escape character \033[44m$
```

data10文本文件包含了转义控制码（参见第17章）来改变显示的颜色。当你用cat命令来显示文本文件时，你不会看到转义控制码，它只会改变显示的颜色。

但用列出命令，你就能显示转义控制码。\\033是Esc键对应的八进制ASCII码。

18.2.8 用sed和文件一起工作

替换命令包含允许你和文件一起工作的标记。还有一些sed编辑器命令也允许你那么做，而不用替换文本。

1. 向文件写入

w命令用来向文件写入行。w命令的格式如下：

```
[address]w filename
```

filename可以指定为相对或绝对路径名，但不管是哪种，运行sed编辑器的人都必须有文件的写权限。地址可以是sed中支持的任意类型的寻址方式，例如单个行号、文本模式或者一系列行号或文本模式。

下面是将数据流中的前两行打印到一个文本文件中的例子：

```
$ sed '1,2w test' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
$ cat test
This is line number 1.
This is line number 2.
```

当然，如果你不想让行显示到STDOUT上，你可以用-n选项。

如果你要基于一些公共文本值从主文件创建一份数据文件，比如下面的邮件列表中的，那么它会非常好用：

```
$ cat data11
Blum, Katie      Chicago, IL
Mullen, Riley   West Lafayette, IN
Snell, Haley     Ft. Wayne, IN
Woenker, Matthew    Springfield, IL
Wisecarver, Emma   Grant Park, IL
$
$ sed -n '/IN/w INcustomers' data11
$
$ cat INcustomers
Mullen, Riley   West Lafayette, IN
Snell, Haley     Ft. Wayne, IN
$
```

sed编辑器会只将包含文本模式的数据行写入目标文件。

2. 从文件读取数据

你已经了解了如何在sed命令行上向数据流中插入或附加文本。读取命令(r)允许你将一个独立文件中的数据插入到数据流中。

读取命令的格式如下：

```
[address]r filename
```

filename参数指定了数据文件的绝对或相对路径名。你不能对读取命令使用地址区间，而只能指定单独一个行号或文本模式地址。sed编辑器会将文件中的文本插入到地址后。

```
$ cat data12
This is an added line.
This is the second added line.
$
$ sed '3r data12' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is an added line.
This is the second added line.
This is line number 4.
```

sed编辑器将数据文件中的所有文本行插入到数据流中了。同样的方法在使用文本模式地址时也适用：

```
$ sed '/number 2/r data12' data7
This is line number 1.
This is line number 2.
This is an added line.
This is the second added line.
This is line number 3.
This is line number 4.
$
```

如果你要在数据流的末尾添加文本，只要用美元符地址符就行了：

```
$ sed '$r data12' data7
This is line number 1.
```

```
This is line number 2.  
This is line number 3.  
This is line number 4.  
This is an added line.  
This is the second added line.  
$
```

读取命令的另一个很好的用途是，将它和删除命令一起使用来用另一文件中的数据替换文件中的占位文本。例如，假如你有个这样的保存在文本文件中的套用信件：

```
$ cat letter  
Would the following people:  
LIST  
please report to the office.  
$
```

套用信件将通用占位文本LIST放在名单的位置。要在占位文本后插入名单，你只要用读取命令就行了。但这样的话，占位文本仍然会留在输出中。要删除占位文本，你可以用删除命令。结果看起来如下：

```
$ sed '/LIST/{  
> r dataall  
> d  
> }' letter  
Would the following people:  
Blum, Katie Chicago, IL  
Mullen, Riley West Lafayette, IN  
Snell, Haley Ft. Wayne, IN  
Woenker, Matthew Springfield, IL  
Wisecarver, Emma Grant Park, IL  
please report to the office.  
$
```

现在占位文本已经被替换成数据文件中的名单。

18.3 小结

虽然shell脚本能独自完成很多事情，但通常只用shell脚本很难处理数据。Linux提供了两个方便的工具来帮助处理文本数据。sed编辑器是个流编辑器，它能在读取数据时快速地自动处理数据。你必须给sed编辑器提供一些作用到数据上的编辑命令。

gawk程序是一个来自GNU组织的工具，它模仿并扩展了Unix中awk程序的功能。gawk程序包括可用来编写处理数据的脚本的一种内建编程语言。你可以用gawk程序来从大数据文件中提取数据元素，并将它们按你要的格式输出。这让处理大型日志文件并从数据文件中生成报告变得非常简单。

使用sed和gawk程序的关键在于了解如何用正则表达式。正则表达式是为提取和处理文本文件中数据创建定制过滤器的关键。下一章将会深入介绍经常被人们误解的正则表达式世界，并演示如何构建正则表达式来操作各种类型的数据。

本章内容

- 定义正则表达式
- 了解基本正则表达式
- 扩展正则表达式
- 创建正则表达式

在 shell脚本中成功使用sed编辑器和gawk程序的关键在于熟练使用正则表达式。这可不总是简单的事，从大量数据中过滤出特定数据可能会（而且经常会）很复杂。本章将介绍如何在sed编辑器和gawk程序中创建正则表达式来过滤出需要的数据。

19.1 什么是正则表达式

理解正则表达式的第一步在于弄清它们到底是什么。本节将会解释什么是正则表达式并介绍Linux如何使用正则表达式。

19.1.1 定义

正则表达式是你定义的、Linux工具用来过滤文本的模式模板。Linux工具（比如sed编辑器或gawk程序）能够在数据流向工具时对数据进行正则表达式模式匹配。如果数据匹配模式，它就会被接受并进一步处理。如果数据不匹配模式，它就会被滤掉。图19-1中描述了这个过程。

正则表达式模式利用通配符来代表数据流中的一个或多个字符。Linux中有很多例子都用通配符来代表不确定的数据。你已经看过了一个用通配符和Linux的ls命令一起列出文件和目录的例子（参见第3章）。

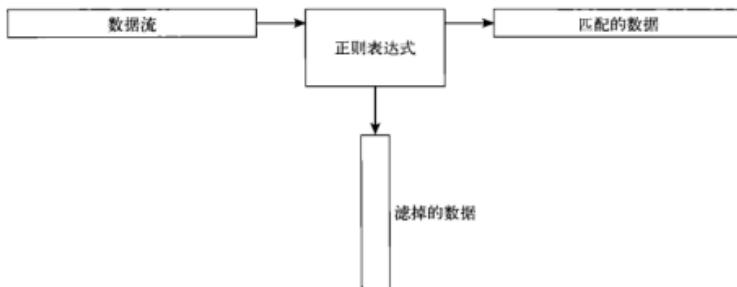


图19-1 对正则表达式模式进行数据匹配

星号通配符允许你只列出满足特定条件的文件。例如：

```
$ ls -al da*
-rw-r--r--  1 rich   rich        45 Nov 26 12:42 data
-rw-r--r--  1 rich   rich       25 Dec  4 12:40 data.tst
-rw-r--r--  1 rich   rich      180 Nov 26 12:42 data1
-rw-r--r--  1 rich   rich      45 Nov 26 12:44 data2
-rw-r--r--  1 rich   rich      73 Nov 27 12:31 data3
-rw-r--r--  1 rich   rich      79 Nov 28 14:01 data4
-rw-r--r--  1 rich   rich     187 Dec  4 09:45 datatest
$
```

da*参数会让ls命令只列出文件名以da开头的文件。文件名中da之后可以有任意多个字符（甚至可以为零）。ls命令会读取目录中所有文件的信息，但只显示跟通配符匹配的文件信息。

正则表达式通配符模式的工作原理与之类似。正则表达式模式含有文本或特殊字符，为sed编辑器和gawk程序定义了一个模板来在匹配数据时参考。你可以在正则表达式中使用不同的特殊字符来定义特殊的过滤数据的模式。

19.1.2 正则表达式的类型

使用正则表达式最大的问题在于有不止一种类型的正则表达式。Linux中的不同应用程序可能会用不同类型的正则表达式。这包括各种应用程序，比如编程语言（Java、Perl和Python）、Linux工具（比如sed编辑器、gawk程序和grep工具），以及主流应用（比如MySQL和PostgreSQL数据库服务器）。

正则表达式是用正则表达式引擎（regular expression engine）实现的。正则表达式引擎是解释正则表达式模式并使用这些模式进行文本匹配的底层软件。

在Linux中，有两种流行的正则表达式引擎：

- POSIX基本正则表达式（BRE）引擎；
- POSIX扩展正则表达式（ERE）引擎。

大多数Linux工具都至少符合POSIX BRE引擎规范，能够识别它定义的所有模式符号。遗憾

的是，有些工具（比如sed编辑器）只实施了BRE引擎规范的子集。这是由于速度限制导致的，sed编辑器希望能尽可能快地处理数据流中的文本。

POSIX BRE引擎通常能在依赖正则表达式做文本过滤的编程语言中找到。它为常见模式提供了高级模式符号和特殊符号，比如匹配数字、单词以及按字母排序的字符。gawk程序用ERE引擎来处理它的正则表达式模式。

由于实现正则表达式的方法太多，很难用一个简洁的描述来涵盖所有可能的正则表达式。后续几节将会讨论最常见的正则表达式并演示如何在sed编辑器和gawk程序中使用它们。

19.2 定义 BRE 模式

最基本的BRE模式是匹配数据流中的文本字符。本节将会演示如何在正则表达式中定义文本以及会得到什么样的结果。

19.2.1 纯文本

第18章演示了如何在sed编辑器和gawk程序中用标准文本字符串来过滤数据。通过下面的例子来复习一下：

```
$ echo "This is a test" | sed -n '/test/p'
This is a test
$ echo "This is a test" | sed -n '/trial/p'
$
$ echo "This is a test" | gawk '/test/{print $0}'
This is a test
$ echo "This is a test" | gawk '/trial/{print $0}'
$
```

第一个模式定义了一个单词，test。sed编辑器和gawk程序脚本用它们各自的print命令打印了匹配该正则表达式模式的所有行。由于echo语句在文本字符串中包含了test，数据流文本匹配定义的正则表达式模式，sed编辑器显示了该行。

第二个模式也定义了一个单词，这次是trial。因为echo语句文本字符串没包含该单词，所以正则表达式模式没有匹配，从而sed编辑器和gawk程序都没打印该行。

你可能注意到正则表达式不管模式在数据流中的位置。它也不管模式出现了多少次。一旦正则表达式匹配了文本字符串中的模式，它就会将该字符串传回Linux工具。

关键在于将正则表达式模式匹配到数据流文本上。重要的是记住正则表达式对匹配的模式非常挑剔。要记住的第一条原则是正则表达式模式都区分大小写，这意味着它们只会匹配大小写也相符的模式：

```
$ echo "This is a test" | sed -n '/this/p'
$
$ echo "This is a test" | sed -n '/This/p'
This is a test
$
```

第一次尝试没能匹配成功，因为this在字符串中并不都是小写，而第二次尝试在模式中使用大写字母，所以能正常工作。

在正则表达式中，你不用写出整个单词。只要定义的文本在数据流中出现了，正则表达式就会匹配剩下的：

```
$ echo "The books are expensive" | sed -n '/book/p'
The books are expensive
$
```

尽管数据流中的文本是books，但数据中含有正则表达式book，因此正则表达式模式跟数据匹配。当然，反之正则表达式就不成立了：

```
$ echo "The book is expensive" | sed -n '/books/p'
$
```

完整的正则表达式文本并未在数据流中出现，因此匹配失败，sed编辑器不会显示任何文本。你也不用局限于在正则表达式中只用单个文本单词，可以在正则表达式中使用空格和数字：

```
$ echo "This is line number 1" | sed -n '/ber 1/p'
This is line number 1
$
```

在正则表达式中，空格会得到跟其他字符一样的对待：

```
$ echo "This is line number1" | sed -n '/ber 1/p'
$
```

如果你在正则表达式中定义了空格，那么它必须出现在数据流中。你甚至可以创建匹配多个连续空格的正则表达式模式：

```
$ cat data1
This is a normal line of text.
This is a line with too many spaces.
$ sed -n '/^ /p' data1
This is a line with too many spaces.
$
```

单词间有两个空格的行匹配正则表达式模式。这是用来查看文本文件中空格问题的好办法。

19.2.2 特殊字符

由于你在正则表达式模式中使用了文本字符，你就必须注意一些事情。在正则表达式中定义文本字符时有一些特例。正则表达式会赋予一些字符特别的意思。如果你要在文本模式中使用这些字符，你不会得到期望的结果。

正则表达式识别的特殊字符包括：

```
.*[^${}]+?|()
```

继续阅读本章，你就会了解这些特殊字符在正则表达式中有何用处。不过现在只要记住不能在文本模式中单独使用这些字符就行了。

如果要用某个特殊字符作为文本字符，你必须转义。在转义特殊字符时，可以在它前面加一个特殊字符来告诉正则表达式引擎，它应该将下一个字符当做普通文本字符来解释。这个特殊字符就是反斜线 (\)。

举个例子，如果你要查找文本中的美元符，只要在它前面加个反斜线：

```
$ cat data2
The cost is $4.00
$ sed -n '/\$/{p}' data2
The cost is $4.00
$
```

由于反斜线是特殊字符，如果要在正则表达式模式中使用它，你必须转义，这样就产生了两个反斜线：

```
$ echo "\ is a special character" | sed -n '/\\/{p}'
\ is a special character
$
```

最终，尽管斜线不是正则表达式的特殊字符，但如果你在sed编辑器或gawk程序的正则表达式中用了，会得到一个错误：

```
$ echo "3 / 2" | sed -n '///{p}'
sed: -e expression #1. char 2: No previous regular expression
$
```

要使用斜线，你也要转义：

```
$ echo "3 / 2" | sed -n '/\//{p'
3 / 2
$
```

现在sed编辑器能正确解释正则表达式模式了，一切都很顺利。

19.2.3 锚字符

如19.2.1节所述，默认情况下，当你指定一个正则表达式模式时，只要模式出现在数据流中的任何地方，它都会匹配。有两个特殊字符可以用来将模式锁定在数据流中的行首或行尾。

1. 锁定在行首

脱字符（caret character, ^）定义从数据流中文本行的行首开始的模式。如果模式位于其他位置而不是文本行的行首，正则表达式模式会不成立。

要用脱字符，你必须将它放在正则表达式中指定的模式前面：

```
$ echo "The book store" | sed -n '/^book/p'
$ echo "Books are great" | sed -n '/^Book/p'
Books are great
$
```

脱字符会在每个由换行符决定的新数据行的行首检查模式：

```
$ cat data3
```

```
This is a test line.
this is another test line.
A line that tests this feature.
Yet more testing of this
$ sed -n '/this/p' data3
this is another test line.
$
```

只要模式出现在新行的行首，脱字符就会捕捉到。

如果你将脱字符放到模式中的其他位置而不是开头，它就会跟普通字符一样，而不是特殊字符：

```
$ echo "This ^ is a test" | sed -n '/s ^/p'
This ^ is a test
$
```

由于脱字符列在了正则表达式模式的最后，sed编辑器会将它当做普通字符来匹配。

说明 如果指定正则表达式模式时只用了脱字符，你不需要用反斜线来转义。但如果你在模式中先指定了脱字符，后跟一些额外的文本，那么你必须在脱字符前用转义字符。

2. 锁定在行尾

跟在行首查找模式相对的就是在行尾查找。美元符 (\$) 特殊字符定义了行尾锚点。将这个特殊字符加在文本模式之后来指明数据行必须以该文本模式结尾：

```
$ echo "This is a good book" | sed -n '/book$/p'
This is a good book
$ echo "This book is good" | sed -n '/book$/p'
$
```

结尾的文本模式的问题在于你必须留意要查找的模式：

```
$ echo "There are a lot of good books" | sed -n '/book$/p'
$
```

将行尾的单词book改成复数形式，就意味着它不再匹配正则表达式模式了，尽管book仍然在数据流中。文本模式必须是模式要匹配的行最后的部分。

3. 组合锚点

在一些常见情况下，你会在同一行中将行首锚点和行尾锚点组合在一起使用。在第一种情况中，假定你要查找含有特定文本模式的数据行：

```
$ cat data4
this is a test of using both anchors
I said this is a test
this is a test
I'm sure this is a test.
$ sed -n '/this is a test$/p' data4
this is a test
$
```

sed编辑器忽略了除了指定的文本还包含其他文本的行。

第二种情况乍一看可能有些怪异，但极其有用。将两个锚点直接组合在一起，不加任何文本，你可以过滤出数据流中的空白行。考虑下面这个例子：

```
$ cat data5
This is one test line.

This is another test line.
$ sed '/$/d' data5
This is one test line.
This is another test line.
$
```

定义的正则表达式模式会查找行首和行尾之间什么都没有的那些行。由于空白行在两个换行符之间没有文本，他们刚好匹配了正则表达式模式。sed编辑器用删除命令d来删除匹配该正则表达式模式的行，因此删除了文本中的所有空白行。这是从文档中删除空白行的有效方法。

19.2.4 点字符

点特殊字符用来匹配任意的单字符，除了换行符。但点字符必须匹配一个字符，如果在点字符的位置没有字符，那么模式就不成立。

让我们看一些在正则表达式模式中使用点字符的例子：

```
$ cat data6
This is a test of a line.
The cat is sleeping.
That is a very nice hat.
This test is at line four.
at ten o'clock we'll go home.
$ sed -n '/.at/p' data6
The cat is sleeping.
That is a very nice hat.
This test is at line four.
$
```

你应该能够明白为什么第1行不成立而第2行和第3行通过了。第4行有点复杂。注意我们匹配了at，但在at前面并没有任何字符来匹配点字符。不对，有的！在正则表达式中，空格会被当做字符，因此at前面的空格刚好匹配了该模式。第5行证明了这点，将at放在行首就不会匹配该模式了。

19.2.5 字符组

点特殊字符在用任意字符来匹配某个字符位置时很有用。但如果要你限定匹配哪些字符呢？在正则表达式中，这称为字符组（character class）。

你可以定义用来匹配文本模式中某个位置的一组字符。如果字符组中的某个字符出现在了数据流中，那它就匹配了该模式。

要定义一个字符组，你要用方括号。方括号中应该含有你要在该组中包含的任何字符。然后

你可以在模式中使用整个组，就跟其他通配符一样。这需要一点时间来适应，但一旦你适应了，它就能生成许多非常好的结果。

下面是个创建字符组的例子：

```
$ sed -n '/[ch]at/p' data6
Thé cat is sleeping.
That is a very nice hat.
$
```

使用在点特殊字符例子中同样的数据文件，我们得到了一个不同的结果。这次我们成功滤掉了含有单词at的行。匹配这个模式的单词只有cat和hat。还要注意以at开头的行也没有匹配。字符组中必须有个字符来匹配相应的位置。

字符组在你不太确定某个字符的大小写情况时非常有用：

```
$ echo "Yes" | sed -n '/[Yy]es/p'
Yes
$ echo "yes" | sed -n '/[Yy]es/p'
yes
$
```

你可以在单个表达式中用多个字符组：

```
$ echo "Yes" | sed -n '/[Yy][Ee][Ss]/p'
Yes
$ echo "yEs" | sed -n '/[Yy][Ee][Ss]/p'
yEs
$ echo "yeS" | sed -n '/[Yy][Ee][Ss]/p'
yeS
$
```

正则表达式使用了3个字符组来覆盖3个字符位置含有大小写的情况。

字符组不必只含有字母，你也可以在它们中使用数字：

```
$ cat data7
This line doesn't contain a number.
This line has 1 number on it.
This line a number 2 on it.
This line has a number 4 on it.
$ sed -n '/[0123]/p' data7
This line has 1 number on it.
This line a number 2 on it.
$
```

这个正则表达式模式匹配了任意含有数字0、1、2或3的行。含有其他数字以及不含有数字的行都会被忽略掉。

你可以将字符组合在一起检查数字是否正确格式化了，比如电话号码和邮编。但当你尝试匹配某个特定格式时，你必须小心。这里有个匹配邮编出错的例子：

```
$ cat data8
60633
46201
223001
```

```

4353
22203
$ sed -n
>/[0123456789][0123456789][0123456789][0123456789]/p
>' data8
60633
46201
223001
22203
$
```

这可能生成了意料之外的结果。它成功过滤掉了过短而不可能是邮编的数字，因为最后一个字符组没有字符可匹配。但它也通过了那个六位数，尽管我们只定义了5个字符组。

记住，正则表达式模式可见于数据流中文本的任何位置。经常有匹配模式的字符之外的其他字符。如果你要确保只匹配五位数，你必须将匹配的字符和其他字符分开，要么用空格，要么像这个例子中这样，指明它们就在行首和行尾：

```

$ sed -n
> /[0123456789][0123456789][0123456789][0123456789][0123456789]/p
>' data8
60633
46201
22203
$
```

现在好多了！本章后面，我们将会看如何进一步简化它。

字符组的一个极其常见的用法是解析拼错的单词，比如用户表单输入的数据。你可以轻松地创建正则表达式来接受数据中常见的拼写错误：

```

$ cat data9
I need to have some maintenence done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
$ sed -n
/maint[ea]n[ae]nce/p
/sep[ea]r[ea]te/p
` data9
I need to have some maintenence done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
$
```

本例中的两个sed打印命令利用正则表达式字符组来帮助找到文本中拼错的单词maintenance和separate。同样的正则表达式模式也能匹配正确拼写的maintenance。

19.2.6 排除字符组

在正则表达式模式中，你也可以反转字符组的作用。你可以寻找组中没有的任意字符，而不是去寻找组中含有的字符。要这么做的话，只要在字符组的开头加个脱字符：

```
$ sed -n '/[^ch]at/p' data6
This test is at line two.
$
```

通过排除字符组，正则表达式模式会匹配c或h之外的任何字符，以及文本模式。由于空格字符也在这类字符中，它通过了模式匹配。但即使是排除，字符组仍然必须匹配一个字符，所以以at开头的行仍然未能匹配模式。

19.2.7 使用区间

你可能注意到我前面演示邮编的例子的时候，必须在每个字符组中列出所有可能的数字，这有点麻烦。好在你可以用快捷方式，所以不必那么做了。

你可以用单破折线符号来在字符组中使用字符区间。指定区间的第一个字符、单破折线，然后是区间的最后一个字符。根据Linux系统采用的字符集（参见第2章），正则表达式包含在这个指定区间内的任意字符。

现在你可以通过指定数字区间来简化邮编的例子：

```
$ sed -n '/^0-9]0-9]0-9]0-9]0-9$/p' data8
60633
46201
45902
$
```

这样节省了很多键入。每个字符组都会匹配0-9的任意数字。如果字母出现在数据中的任何位置，这个模式都将不成立：

```
$ echo "a8392" | sed -n '/^0-9]0-9]0-9]0-9]0-9$/p'
$
$ echo "1839a" | sed -n '/^0-9]0-9]0-9]0-9]0-9$/p'
$
$ echo "18a92" | sed -n '/^0-9]0-9]0-9]0-9]0-9$/p'
$
```

同样的方法也适用于字母：

```
$ sed -n '/[c-h]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

新的模式[c-h]at匹配了首字母在字母c和字母h之间的单词。这种情况下，只含有单词at的行将无法匹配该模式。

你还可以在单个字符组指定多个不连续的区间：

```
$ sed -n '/[a-ch-m]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

该字符组允许区间a~c、h~m中的字母出现在at文本前。这个区间不允许d~g之间的字母：

```
$ echo "I'm getting too fat." | sed -n '/[a-ch-m]at/p'
$
```

该模式放弃了fat文本，因为它没在指定的区间。

19.2.8 特殊字符组

除了定义自己的字符组外，BRE还包含可用来匹配特殊字符类型的特殊字符组。表19-1介绍了可用的BRE特殊字符。

表19-1 BRE特殊字符组

组	描述
[:alpha:]	匹配任意字母字符，不管是大写还是小写
[:alnum:]	匹配任意字母数字字符0-9、A-Z或a-z
[:blank:]	匹配空格或制表符
[:digit:]	匹配0-9之间的数字
[:lower:]	匹配小写字母字符a-z
[:print:]	匹配任意可打印字符
[:punct:]	匹配标点符号
[:space:]	匹配任意空白字符：空格、制表符、NL、FF、VT和CR
[:upper:]	匹配任意大写字母字符A-Z

可以在正则表达式模式中将特殊字符组像普通字符组一样使用：

```
$ echo "abc" | sed -n '/[:digit:]/p'
$
$ echo "abc" | sed -n '/[:alpha:]/p'
abc
$ echo "abc123" | sed -n '/[:digit:]/p'
abc123
$ echo "This is, a test" | sed -n '/[:punct:]/p'
This is, a test
$ echo "This is a test" | sed -n '/[:punct:]/p'
$
```

使用特殊字符组可以很方便地定义区间。你可以用[:digit:]，而不必使用区间[0-9]。

19.2.9 星号

在字符后面放置星号说明该字符将会在匹配模式的文本中出现0次或多次：

```
$ echo "ik" | sed -n '/ie*k/p'
ik
$ echo "iek" | sed -n '/ie*k/p'
iek
$ echo "ieek" | sed -n '/ie*k/p'
```

```
ieek
$ echo "ieek" | sed -n '/ie*k/p'
ieek
$ echo "eeeeek" | sed -n '/ie*k/p'
eeeeek
$
```

这个模式符号广泛用于处理有常见拼写错误或在不同语言中有拼写变种的单词。举个例子，如果需要写个可能用在美式或英式英语中的脚本，你可以这么写：

```
$ echo "I'm getting a color TV" | sed -n '/colou*r/p'
I'm getting a color TV
$ echo "I'm getting a colour TV" | sed -n '/colou*r/p'
I'm getting a colour TV
$
```

模式中的`u*`表明字母`u`可能出现或不出现在匹配模式的文本中。类似地，如果你知道一个单词经常被拼错，你可以用星号来允许这种错误：

```
$ echo "I ate a potatoe with my lunch." | sed -n '/potatoe*/p'
I ate a potatoe with my lunch.
$ echo "I ate a potato with my lunch." | sed -n '/potatoe*/p'
I ate a potato with my lunch.
$
```

在可能的额外字母后面放个星号将会接受拼错的单词。

另一个方便的特性是将点特殊字符和星号特殊字符组合起来。这个组合提供了匹配任意多个任意字符的模式。它通常用在数据流中两个可能相邻或不相邻的文本字符串之间：

```
$ echo "this is a regular pattern expression" | sed -n '
> /regular.*expression/p'
this is a regular pattern expression
$
```

使用这个模式，你就能轻松地查找数据流中可能出现在文本行中任意位置的多个单词。

星号还能用在字符组上。它允许指定可能在文本中出现多次的一组字符或一个字符区间：

```
$ echo "bt" | sed -n '/b[ae]*t/p'
bt
$ echo "bat" | sed -n '/b[ae]*t/p'
bat
$ echo "bet" | sed -n '/b[ae]*t/p'
bet
$ echo "btt" | sed -n '/b[ae]*t/p'
btt
$
$ echo "baat" | sed -n '/b[ae]*t/p'
baat
$ echo "baaaeet" | sed -n '/b[ae]*t/p'
baaaeet
$ echo "baeeeateat" | sed -n '/b[ae]*t/p'
baeeeateat
$ echo "baakeeet" | sed -n '/b[ae]*t/p'
$
```

只要a和e字符出现在b和t字符中的任意组合中(包括根本不出现的情况),模式就匹配了。如果有任何不在定义的字符组内的其他字符出现,该模式匹配就会不成立。

19.3 扩展正则表达式

POSIX ERE模式包括供一些Linux应用和工具使用的若干额外符号。gawk程序能够识别ERE模式,但sed编辑器不能。

警告 记住sed编辑器和gawk程序的正则表达式引擎之间是有区别的。gawk程序可以使用大多数扩展正则表达式模式符号,并且能提供一些额外的sed编辑器没有的额外过滤功能。但正因为如此,它通常在处理数据流时更慢。

本节将介绍可用在gawk程序脚本中的较常见的ERE模式符号。

19.3.1 问号

问号类似于星号,不过有点细微的不同。问号表明前面的字符可以出现0次或1次,但只限于此。它不会匹配多次出现的该字符:

```
$ echo "bt" | gawk '/be?t/{print $0}'
bt
$ echo "bet" | gawk '/be?t/{print $0}'
bet
$ echo "beet" | gawk '/be?t/{print $0}'
$
$ echo "beeet" | gawk '/be?t/{print $0}'
$
```

如果e字符并未在文本中出现,或者它只在文本中出现了一次,那么模式会匹配。

跟星号一样,你可以将问号符号和字符组一起使用:

```
$ echo "bt" | gawk '/b[ae]?t/{print $0}'
bt
$ echo "bat" | gawk '/b[ae]?t/{print $0}'
bat
$ echo "bot" | gawk '/b[ae]?t/{print $0}'
$
$ echo "bet" | gawk '/b[ae]?t/{print $0}'
bet
$ echo "baet" | gawk '/b[ae]?t/{print $0}'
$
$ echo "beat" | gawk '/b[ae]?t/{print $0}'
$
$ echo "beet" | gawk '/b[ae]?t/{print $0}'
$
```

如果字符组中的字符出现了0次或1次,模式匹配就成立。但如果两个字符都出现了,或者如

果其中一个字符出现了两次，模式匹配都不成立。

19.3.2 加号

加号是类似于星号的另一个模式符号，但跟问号也有不同。加号表明前面的字符可以出现1次或多次，但必须至少出现1次。如果该字符没有出现，那么模式就不会匹配：

```
$ echo "beet" | gawk '/be+t/{print $0}'
beet
$ echo "beet" | gawk '/be+t/{print $0}'
beet
$ echo "bet" | gawk '/be+t/{print $0}'
bet
$ echo "bt" | gawk '/be+t/{print $0}'
$
```

如果e字符没有出现，模式匹配就不成立。加号同样适用于字符组，跟星号和问号的使用方式相同：

```
$ echo "bt" | gawk '/b[ae]+t/{print $0}'
$
$ echo "bat" | gawk '/b[ae]+t/{print $0}'
bat
$ echo "bet" | gawk '/b[ae]+t/{print $0}'
bet
$ echo "beat" | gawk '/b[ae]+t/{print $0}'
beat
$ echo "beet" | gawk '/b[ae]+t/{print $0}'
beet
$ echo "beeat" | gawk '/b[ae]+t/{print $0}'
beeat
$
```

这次如果字符组中定义的任一字符出现了，文本就会匹配指定的模式。

19.3.3 使用花括号

ERE中的花括号允许你为可重复的正则表达式指定一个上限。这通常称为区间（interval）。你可以用两个格式来指定区间：

- m——正则表达式准确出现m次；
- m, n——正则表达式至少出现m次，至多n次；

这个特性可以微调允许字符或字符集在模式中具体出现多少次。

警告 默认情况下，gawk程序不会识别正则表达式区间。你必须为gawk程序指定`--re-interval`命令行选项来识别正则表达式区间。

这里有个使用简单的单值区间的例子：

```
$ echo "bt" | gawk --re-interval '/be{1}t/{print $0}'
$
$ echo "bet" | gawk --re-interval '/be{1}t/{print $0}'
bet
$ echo "beet" | gawk --re-interval '/be{1}t/{print $0}'
$
```

通过指定区间为1，你限定了该字符在匹配模式的字符串中出现的次数。如果该字符出现多次，模式匹配就不成立。

很多时候指定下限和上限也能派上用场：

```
$ echo "bt" | gawk --re-interval '/be{1,2}t/{print $0}'
$
$ echo "bet" | gawk --re-interval '/be{1,2}t/{print $0}'
bet
$ echo "beet" | gawk --re-interval '/be{1,2}t/{print $0}'
beet
$ echo "beeet" | gawk --re-interval '/be{1,2}t/{print $0}'
$
```

在这个例子中，字符e可以出现一次或两次，这样模式匹配就能成立了；否则，模式匹配不成立。

区间模式匹配同样适用于字符串：

```
$ echo "bt" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "bat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
bat
$ echo "bet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
bet
$ echo "beat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
beat
$ echo "beet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
beet
$ echo "beeat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "baeet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
```

如果在文本模式中有字母a或e的一到两个实例，这个正则表达式模式就会匹配，但如果在任何组合中有多于两个出现，它就会不匹配了。

19.3.4 管道符号

管道符号允许你在检查数据流时，用逻辑OR方式指定正则表达式引擎要用的两个或多个模式。如果任何一个模式匹配了数据流文本，文本就通过了。如果没有模式匹配，数据流文本匹配就不成立。

使用管道符号的格式如下：

`expr1|expr2|...`

这里有个例子：

```
$ echo "The cat is asleep" | gawk '/cat|dog/{print $0}'
The cat is asleep
$ echo "The dog is asleep" | gawk '/cat|dog/{print $0}'
The dog is asleep
$ echo "The sheep is asleep" | gawk '/cat|dog/{print $0}'
$
```

这个例子会在数据流中查找正则表达式cat或dog。正则表达式和管道符号之间不能有空格，否则它们也会加到正则表达式模式中。

管道符号两侧的正则表达式都可用任何正则表达式模式（包括字符组）来定义模式：

```
$ echo "He has a hat." | gawk '/[ch]at|dog/{print $0}'
He has a hat.
$
```

这个例子会匹配数据流文本中的cat、hat或dog。

19.3.5 聚合表达式

正则表达式模式也可以用圆括号聚合起来。当你聚合正则表达式模式时，该组就会被当成标准字符。你可以给该组使用特殊字符，就跟你给正常字符使用一样。举个例子：

```
$ echo "Sat" | gawk '/Sat(urday)?/{print $0}'
Sat
$ echo "Saturday" | gawk '/Sat(urday)?/{print $0}'
Saturday
$
```

这组结尾的urday加一个问号，就会允许模式匹配完整名Saturday或缩写名Sat。

将聚合和管道符号一起使用来创建可能的模式匹配组是很常见的：

```
$ echo "cat" | gawk '/(c|b)a(b|t)/{print $0}'
cat
$ echo "cab" | gawk '/(c|b)a(b|t)/{print $0}'
cab
$ echo "bat" | gawk '/(c|b)a(b|t)/{print $0}'
bat
$ echo "bab" | gawk '/(c|b)a(b|t)/{print $0}'
bab
$ echo "tab" | gawk '/(c|b)a(b|t)/{print $0}'
$
$ echo "tac" | gawk '/(c|b)a(b|t)/{print $0}'
$
```

模式`(c|b)a(b|t)`会匹配第一组中字母的任意组合以及第二组中字母的任意组合。

19.4 实用中的正则表达式

现在你已经了解了使用正则表达式模式的规则和一些简单的例子，是时候实践一下了。后面

几节将会演示shell脚本中常见的一些正则表达式例子。

19.4.1 目录文件计数

一开始，让我们看一个shell脚本，它会对PATH环境变量中定义的目录里的可执行文件进行计数。要这么做，首先你得将PATH变量解析成独立的目录名。第5章介绍了如何显示PATH环境变量：

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/usr/games:/usr/java/
j2sdk1.4_1_01/bin
$
```

根据Linux系统上应用程序所放的位置，PATH环境变量会有所不同。关键是要意识到PATH中的每个路径由冒号分隔。要获取可在脚本中使用的目录列表，你必须用空格来替换冒号。现在你意识到sed编辑器用一条简单表达式就能完成替换工作：

```
$ echo $PATH | sed 's/:/ /g'
/usr/local/bin /bin /usr/bin /usr/X11R6/bin /usr/games /usr/java/
j2sdk1.4_1_01/bin
$
```

一旦得到分离了的目录，你就能在标准for语句中（参见第12章）用它们来遍历每个目录：

```
mypath=`echo $PATH | sed 's/:/ /g'`
for directory in $mypath
do
...
done
```

你有了每个目录后，你就可以用ls命令来列出每个目录中的文件，并用另一个for语句来遍历每个文件，为每个文件将计数器增一。

这个脚本的最终版本如下：

```
$ cat countfiles
#!/bin/bash
# count number of files in your PATH
mypath=`echo $PATH | sed 's/:/ /g'`
count=0
for directory in $mypath
do
    check=`ls $directory`
    for item in $check
    do
        count=$((count + 1))
    done
    echo "$directory - $count"
    count=0
done
$ ./countfiles
/usr/local/bin - 79
/bin - 86
/usr/bin - 1502
/usr/X11R6/bin - 175
```

```
/usr/games - 2
/usr/java/j2sdk1.4.1_01/bin - 27
$
```

现在我们开始看到正则表达式背后的强大之处了。

19.4.2 验证电话号码

前面的例子演示了如何用简单的正则表达式和sed一起替换数据流中的字符来处理数据。通常用正则表达式来验证数据，为脚本确保数据格式正确。

这里有个常见的数据验证应用程序来检查电话号码。通常，数据输入表单会要求填入电话号码，而通常用户不会输入格式正确的电话号码。在美国，有几种常见的方式来显示电话号码：

```
(123)456-7890
(123) 456-7890
123-456-7890
123.456.7890
```

这样用户在表单中输入的电话号码就有四种可能。正则表达式必须足够强大才能处理每一种情况。

在构建正则表达式时，最好从左手边开始，然后构建用来匹配可能遇到的字符的模式。在这个例子中，电话号码中可能有一个或没有左圆括号。这可以用如下模式来匹配：

```
^\(?
```

脱字符用来表明数据的开始。由于左圆括号是个特殊字符，把它当做普通字符使用时必须将它转义。问号表明左圆括号可能出现，也可能不出现。

紧接着就是三位区号。在美国，区号以数字2开始（没有区号以数字0或1开始），最大可到9。要匹配区号，你会用如下模式：

```
[2-9][0-9]{2}
```

这要求第一个字符是2~9的数字，后跟任意两位数字。在区号后面，收尾的右圆括号可能存在，也可能不存在：

```
\)?
```

在区号后，可能有一个空格或没有空格、一条单破折线或一个点。你可以将这些用管道符号和圆括号聚合起来：

```
( | - | \.)
```

第一个管道符号紧跟在左圆括号后，用来匹配没有空格的情形。你必须将点字符转义，否则它会被解释成可匹配任意字符。

紧接着是三位电话交换机号码，没什么特殊的：

```
[0-9]{3}
```

在电话交换机号码之后，你必须匹配一个空格、一条单破折线或一个点（这次你不用考虑匹配没有空格的情况，因为在电话交换机号码和其余号码间至少有一个空格）：

(|-|.)

然后，要完成所有事情，你必须在字符串尾部匹配四位本地电话分机号：

[0-9]{4}\$

将整个模式放在一起就生成了这个：

^\(?[2-9][0-9]{2}\)\)?(| |-|.)[0-9]{3}(| |-|.)[0-9]{4}\$

你可以在gawk程序中用这个正则表达式模式来过滤掉不符合格式的电话号码。现在你要做的就是创建一个简单的脚本在gawk程序中使用这个正则表达式，然后用脚本过滤你的电话列表。记住，在gawk程序中使用正则表达式区间时，必须使用--re-interval命令行选项，否则你就没法得到正确的结果。

脚本如下：

```
$ cat isphone
#!/bin/bash
# script to filter out bad phone numbers
gawk --re-interval '/^\(?[2-9][0-9]{2}\)\)?(| |-|.)[0-9]{3}(| |-|.)[0-9]{4}/ {print $0}'
$
```

虽然从上面的清单中看不出来，但是在shell脚本中gawk命令是在一行上的。然后你可以将电话号码重定向到脚本来处理：

```
$ echo "317-555-1234" | ./isphone
317-555-1234
$ echo "000-555-1234" | ./isphone
$
```

或者你也可以将含有电话号码的整个文件重定向到脚本来过滤掉无效的号码：

```
$ cat phonelist
000-000-0000
123-456-7890
212-555-1234
(317)555-1234
(202) 555-9876
33523
1234567890
234.123.4567
$ cat phonelist | ./isphone
212-555-1234
(317)555-1234
(202) 555-9876
234.123.4567
$
```

只有匹配该正则表达式模式的有效电话号码才会出现。

19.4.3 解析邮件地址

在今天这个时代，E-mail地址已经成为一种重要的通信方式。验证E-mail地址已经成为脚本

程序员的一个挑战，因为创建E-mail地址的方式太多了。E-mail地址的基本格式为：

`username@hostname`

`username`值可用字母数字字符以及以下特殊字符：

- 点号；
- 单破折线；
- 加号；
- 下划线。

在有效的E-mail用户名中，这些字符可能以任意组合形式出现。E-mail地址的`hostname`部分由一个或多个域名和一个服务器名组成。服务器名和域名也必须遵照严格的命名规则，只允许字母数字字符以及以下特殊字符：

- 点号；
- 下划线。

服务器名和域名都用点分隔，先指定服务器名，紧接着指定子域名，最后是后面不带点号的顶级域名。

过去只有十分有限数目的顶级域名，正则表达式模式编写者会尝试将它们都加在验证的模式中。遗憾的是，随着互联网的发展，可用的顶级域名也增多了。这种方法已经不再可行。

让我们从左侧开始构建这个正则表达式模式。我们知道，用户名中可以有多个有效字符。这个相当容易：

`^([a-zA-Z0-9_\.-\._]+)+@`

这个聚合指定用户名中允许的字符，加号表明必须有至少一个字符。下一个字符很明显是@符，没什么意外的。

`hostname`模式使用同样的方法来匹配服务器名和子域名：

`([a-zA-Z0-9_\.-\._]+)+`

这个模式可以匹配文本：

```
server
server.subdomain
server.subdomain.subdomain
```

对于顶级域名，有一些特殊的规则。顶级域名只能是字母字符，必须不少于两个字符（国家或地区代码中使用），并且长度上不得超过5个字符。下面就是顶级域名用的正则表达式模式：

`\.([a-zA-Z]{2,5})$`

将整个模式放在一起会生成如下模式：

`^([a-zA-Z0-9_\.-\._]+)+@[a-zA-Z0-9_\.-\._]+\.([a-zA-Z]{2,5})$`

这个模式会从数据列表中过滤掉那些没有严格格式化的E-mail地址。现在你可以创建脚本来实施这个正则表达式了：

```
$ echo "rich@here.now" | ./isE-mail
rich@here.now
$ echo "rich@here.now." | ./isE-mail
$
$ echo "rich@here.n" | ./isE-mail
$
$ echo "rich@here-now" | ./isE-mail
$
$ echo "rich.blum@here.now" | ./isE-mail
rich.blum@here.now
$ echo "rich_blum@here.now" | ./isE-mail
rich_blum@here.now
$ echo "rich/blum@here.now" | ./isE-mail
$
$ echo "rich#blum@here.now" | ./isE-mail
$
$ echo "rich*blum@here.now" | ./isE-mail
$
```

19.5 小结

如果你在shell脚本中处理数据文件，你必须熟悉正则表达式。正则表达式用在使用正则表达式引擎的Linux工具、编程语言和应用中。在Linux中有一些不同的正则表达式引擎。最流行的两种是POSIX基本正则表达式（BRE）引擎和POSIX扩展正则表达式（ERE）引擎。sed编辑器基本符合BRE引擎，而gawk程序则使用了ERE引擎中的大多数特性。

正则表达式定义了用来过滤数据流中文本的模式模板。模式由标准文本字符和特殊字符的组合组成。正则表达式引擎用特殊字符来匹配一系列一个或多个字符，类似于其他应用程序中通配符的工作方式。

将字符和特殊字符组合起来，你就能定义匹配大多数类型数据的模式了。然后你可以用sed编辑器或gawk程序来从大型数据流中过滤特定数据了，或者验证从其他数据输入应用程序收到的数据。

下一章将会更深入地使用sed编辑器来进行高级文本处理。sed编辑器中的许多高级功能让它在处理大型数据流和过滤数据时非常有用。

本章内容

- 多行命令
- 保持空间
- 排除命令
- 改变流
- 模式替代
- 在脚本中使用sed
- 创建sed实用程序

第

18章介绍了如何用sed编辑器的基本功能来处理数据流中的文本。sed编辑器基本命令能满足大多数日常文本编辑需求。本章将会介绍一下sed编辑器提供的更多高级特性。这些功能你可能不经常用，但需要时，知道有哪些可用以及如何使用是很好的。

20.1 多行命令

在使用sed编辑器的基本命令时，你可能注意到了一个局限。所有的sed编辑器命令都是对单行数据执行操作的。在sed编辑器读取数据流时，它会基于换行符的位置将数据分成行。sed编辑器一次一行地处理数据行——在一个文本行上运行定义好的脚本命令，然后移到下一行重复这个过程。

有时你需要在跨多行的数据上执行特定的操作。如果你要查找或替换一个短语，就更是如此了。

举个例子，如果你正在数据中查找短语Linux System Administrators Group，很有可能短语在其中任意两个词之间被分成两行。如果你用普通的sed编辑器命令来处理文本，几乎不可能发现短语是怎么被分开的。

幸运的是，sed编辑器背后的设计人员已经考虑到了这种情况并且设计了一个解决方案。sed编辑器包含了3个可用来处理多行文本的特殊命令。

- N：将数据流中的下一行加进来创建一个多行组来处理。

D: 删除多行组中的一行。

P: 打印多行组中的一行。

后面几节将会进一步讲解这些多行命令并向你演示如何在脚本中使用它们。

20.1.1 next命令

在了解多行next命令之前，你首先需要看一下单行版本的next命令是如何工作的，然后，理解多行版本的next命令如何操作就容易多了。

1. 单行的next命令

小写的n命令会告诉sed编辑器移动到数据流中的下一本行，而不用重新回到命令的最开始再执行一遍。记住，通常sed编辑器会在移动到数据流中的下一本行前在这行上执行所有定义好的命令。单行next命令改变了这个流程。

这听起来可能有些复杂，有时确实是。在这个例子中，你有个数据文件，共有5行内容，其中两行是空的。目标是删除头行之后的空白行，而留下最后一行之前的空白行。如果你写一个删掉空白行的sed脚本，你会删掉两个空白行：

```
$ cat data1
This is the header line.

This is a data line.

This is the last line.
$

$ sed '/$/d' data1
This is the header line.
This is a data line.
This is the last line.
$
```

由于要删除的行是空白的，你没有任何可以查找的文本来唯一识别该行。解决办法是用n命令。在这个next例子中，脚本要查找含有单词header的唯一行。一旦脚本找到了该行，n命令会将sed编辑器移动到文本的下一行，那是个空行：

```
$ sed '/header/{n ; d}' data1
This is the header line.

This is a data line.

This is the last line.
$
```

这时，sed编辑器会继续执行命令列表，该命令列表使用d命令来删除空白行。sed编辑器执行完命令脚本后，它会从数据流中读取下一行文本并开始从命令脚本顶部执行命令。sed编辑器不会再找到另一带有单词header的行了。因此，没有其他行会被删掉。

2. 合并文本行

了解了单行版本的next命令，现在来看看多行版本的。单行next命令会将数据流中的下一本行移动到sed编辑器的工作空间（称为模式空间）。多行版本的next命令（用大写N）会将下一

文本行加到已经在模式空间中的文本上。

这样的作用是将数据流中的两个文本行合并到同一个模式空间。文本行仍然用换行符分隔，但sed编辑器现在会将两行文本当成一行来处理。

这里有个说明N命令如何工作的例子：

```
$ cat data2
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed '/first/{ N ; s/\n/ / }' data2
This is the header line.
This is the first data line. This is the second data line.
This is the last line.
$
```

sed编辑器脚本查找含有单词first的那行文本。当它找到了该行，它会用N命令将下一行合并到那行，然后用替换命令s将换行符替换成空格。结果是文本文件中的两行在sed编辑器的输出中成了一行。

你要在数据文件中查找一个可能被分成两行的文本短语的话，这是个很实用的应用程序。这里有的例子：

```
$ cat data3
The first meeting of the Linux System
Administrator's group will be held on Tuesday.
All System Administrators should attend this meeting.
Thank you for your attendance.
$
$ sed 's/System Administrator/Desktop User/' data3
The first meeting of the Linux System
Administrator's group will be held on Tuesday.
All Desktop Users should attend this meeting.
Thank you for your attendance.
$
```

替换命令会在文本文件中查找特定的双词短语System Administrator。短语在一行中的话，事情很好处理，替换命令可以直接替换文本。但如果短语被分成两行了，替换命令就没法识别匹配的模式了。

N命令帮忙解决了这个问题：

```
$ sed 'N ; s/System Administrator/Desktop User/' data3
The first meeting of the Linux Desktop User's group will be held on Tuesday.
All Desktop Users should attend this meeting.
Thank you for your attendance.
$
```

用N命令将下一行和发现第一个单词的那行合并后，即使短语内出现了换行，你仍然可以找到它。

注意，替换命令在System和Administrator之间用了通配符模式(.)来匹配空格和换行符这两

种情况。但当它匹配了换行符时，它就从字符串中删掉了换行符，导致两行合并成一行。这可能不是你想要的。

要解决这个问题，你可以在sed编辑器脚本中用两个替换命令，一个用来匹配多行出现的情况，一个用来匹配单行出现的情况：

```
$ sed '
> N
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data4
The first meeting of the Linux Desktop
User's group will be held on Tuesday.
All Desktop Users should attend this meeting.
Thank you for your attendance.
$'
```

第一个替换命令专门查找这两个要查找的单词间的换行符，并将它放在了替换字符串中。这样你就能在新文本的同样位置添加换行符了。

但这个脚本中仍有个小问题。这个脚本总是在执行sed编辑器命令前将下一行文本读入到模式空间。当它到了最后一行文本时，没有下一行文本可读了，所以N命令会叫sed编辑器停止。如果要匹配的文本正好在数据流的最后一行上，命令就不会发现要匹配的数据：

```
$ cat data4
The first meeting of the Linux System
Administrator's group will be held on Tuesday.
All System Administrators should attend this meeting.
$
$ sed '
> N
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data4
The first meeting of the Linux Desktop
User's group will be held on Tuesday.
All System Administrators should attend this meeting.
$'
```

由于System Administrator文本出现在了数据流中的最后一行，N命令会错过它，因为没有其他行可读入到模式空间跟这行合并了。你可以轻松地解决这个问题——将单行命令放到N命令前面，并将多行命令放到N命令后面，像这样：

```
$ sed '
> s/System Administrator/Desktop User/
> N
> s/System\nAdministrator/Desktop\nUser/
> ' data4
The first meeting of the Linux Desktop
User's group will be held on Tuesday.
All Desktop Users should attend this meeting.
$'
```

现在，查找单行中短语的替换命令在数据流的最后一行也能正常工作，多行替换命令则会负责短语出现在数据流中间的情况。

20.1.2 多行删除命令

第18章介绍了单行删除（*delete*）命令d。sed编辑器用它来删除模式空间中的当前行。但和N命令一起使用时，使用单行删除命令就要小心了：

```
$ sed 'N : /System\nAdministrator/d' data4
All System Administrators should attend this meeting.
$
```

删除命令会在不同的行中查找单词System和Administrator，然后在模式空间中将两行都删掉。这可能是也可能不是你想要的结果。

sed编辑器提供了多行删除命令D，它只删除模式空间中的第一行。它会删除到换行符（含换行符）的所有字符：

```
$ sed 'N : /System\nAdministrator/D' data4
Administrator's group will be held on Tuesday.
All System Administrators should attend this meeting.
$
```

文本的第二行被N命令加到模式空间了，但仍然完好。如果需要删掉从中找到数据字符串的那行的前一文本行时，它能派得上用场。

这里有个例子，它会删除数据流中出现在第一行前的空白行：

```
$ cat data5
This is the header line.
This is a data line.

This is the last line.
$
$ sed '/^$/{N : /header/D}' data5
This is the header line.
This is a data line.

This is the last line.
$
```

sed编辑器脚本会查找空白行，然后用N命令来将下一文本行添加到模式空间。如果新的模式空间内容含有单词header，则D命令会删除模式空间中的第一行。如果不组合使用N命令和D命令，几乎不可能只删除第一个空白行而不删除其他空白行。

20.1.3 多行打印命令

到目前为止，你可能已经了解了单行和多行版本命令间的差异了。多行打印（print）命令P沿用了同样的方法。它只打印多行模式空间中的第一行。这包括模式空间中直到换行符的所有字符。当你用-n选项来阻止脚本输出时，它和显示文本的单行p命令的用法大同小异：

```
$ sed -n 'N ; /System\nAdministrator/P' data3
The first meeting of the Linux System
$
```

当多行匹配出现时，P命令只会打印模式空间中的第一行。多行P命令的强大之处在于和N命令及D命令组合使用时才能显现出来。

D命令有个特性是会强制sed编辑器返回到脚本的起始处，并在同一模式空间重复执行这些命令（它不会从数据流中读取新的文本行）。在命令脚本中加入N命令，你就能单步扫过整个模式空间，将多行一起匹配。

下一步，用P命令，你能打印出第一行，然后用D命令你能删除第一行并回环到脚本的起始处。一旦你回到了脚本的起始处，N命令会读取下一行文本并重新开始这个过程。这个循环会一直继续下去直到数据流的结尾。

20.2 保持空间

模式空间（pattern space）是一块活动缓冲区，在sed编辑器执行命令时它会保存sed编辑器要检验的文本。但它并不是sed编辑器保存文本的唯一空间。

sed编辑器还利用了另一块缓冲区域，称做保持空间（hold space）。你可以在处理模式空间中其他行时用保持空间来临时保存一些行。有5条命令用来操作保持空间，见表20-1。

表20-1 sed编辑器的保持空间命令

命 令	描 述
h	将模式空间复制到保持空间
H	将模式空间附加到保持空间
g	将保持空间复制到模式空间
G	将保持空间附加到模式空间
x	交换模式空间和保持空间的内容

这些命令用来将文本从模式空间复制到保持空间。它可以清空模式空间来加载其他要处理的字符串。

通常，在使用h或H命令将字符串移动到保持空间后，最终你要用g、G或x命令来将保存的字符串移回模式空间（否则，你就不用在一开始考虑保存它们了）。

由于有两个缓冲区域，决定哪行文本在哪个缓冲区域有时会比较麻烦。这里有个简短的例子演示如何用h和g命令来将数据在sed编辑器缓冲空间之间移动：

```
$ cat data2
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
```

```
$ sed -n '/first/{  
    > h  
    > p  
    > n  
    > p  
    > g  
    > p  
}> }' data2  
This is the first data line.  
This is the second data line.  
This is the first data line.  
$
```

我们来一步一步地看前面这个代码例子：

- (1) sed脚本在地址中用正则表达式来过滤出含有单词first的行；
- (2) 当含有单词first的行出现时，h命令将该行放到保持空间；
- (3) p命令打印模式空间也就是第一个数据行的内容；
- (4) n命令提取数据流中的下一行（This is the second data line），并将它放到模式空间；
- (5) p命令打印模式空间的内容，现在是第二个数据行；
- (6) g命令将保持空间的内容（This is the first data line）放回模式空间，替换当前文本；
- (7) p命令打印模式空间的当前内容，现在变回第一个数据行了。

通过使用保持空间来回移动文本行，你可以强制输出中第一个数据行出现在第二个数据行后面。如果你丢了第一个p命令，你可以以相反的顺序输出这两行：

```
$ sed -n '/first/{  
    > h  
    > n  
    > p  
    > g  
    > p  
}> }' data2  
This is the second data line.  
This is the first data line.  
$
```

这是个有用的开端。你可以用这种方法来创建一个sed脚本将整个文件的文本行反转。但要那么做，你需要了解sed编辑器的排除特性，也就是下节的内容。

20.3 排除命令

第18章演示了sed编辑器如何将命令应用到数据流中的每一个文本行，或者由单个地址或地址区间特别指定的多行。你也可以配置命令使其不要作用到数据流中的特定地址或地址区间。

感叹号命令(!)用来排除(negate)命令，也就是让原本会起作用的命令不起作用。这里有 个演示这一特性的例子：

```
$ sed -n '/header/!p' data2  
This is the first data line.
```

```
This is the second data line.  
This is the last line.  
$
```

普通p命令只打印data2文件中包含单词header的那行。加了感叹号之后，除了含有地址中指定文本的行外，文件中所有的行都打印了。

使用感叹号在有些应用中能源上用场。回忆一下，20.1.1节演示了一种情况：sed编辑器不处理数据流中最后一行文本，因为它后面再没有其他行了。可以用感叹号来解决那个问题：

```
$ sed '!; s/System Administrator/Desktop User/' data4  
The first meeting of the Linux Desktop User's group will be held on Tuesday  
All System Administrators should attend this meeting.  
$  
$ sed '$!; s/System Administrator/Desktop User/' data4  
The first meeting of the Linux Desktop User's group will be held on Tuesday  
All Desktop Users should attend this meeting.  
$
```

这个例子演示了和N命令一起使用的感叹号，还用到了美元符特殊地址。美元符代表数据流中的最后一行文本，所以当sed编辑器到了最后一行时，它没有执行N命令。但它对所有其他行都执行了这个命令。

使用这种方法，你可以反转数据流中文本行的顺序。要反转它们在文本流中出现的顺序（先显示最后一行，最后显示第一行），你需要用保持空间做一些漂亮的操作步骤。

你要用的模式会像这样：

- (1) 在模式空间中放置一行；
- (2) 将模式空间中的行放到保持空间中；
- (3) 在模式空间中放入下一行；
- (4) 将保持空间附加到模式空间后；
- (5) 将模式空间中的所有内容都放到保持空间中；
- (6) ~ (9)重复执行第(3)~(5)步，直到所有行都反序放到了保持空间中；
- (10) 提取并打印行。

图20-1详细描述了这个过程。

在使用这种方法时，你不想在处理时打印行。这意味着要使用sed的-n命令行选项。下一步是决定如何将保持空间文本附加到模式空间文本后面。这可以用G命令完成。唯一的问题是你不想将保持空间附加到要处理的第一行文本后面。这可以用感叹号命令轻松解决：

```
!G
```

下一步就是将新的模式空间（带附加反转行的行）放到保持空间。这也非常简单，只要用h命令就行。

将模式空间中的整个数据流都反转了之后，你要做的就是打印结果。当到达数据流中的最后一行时，你就知道已经得到了模式空间的整个数据流。要打印结果，用下面的命令：

```
$p
```

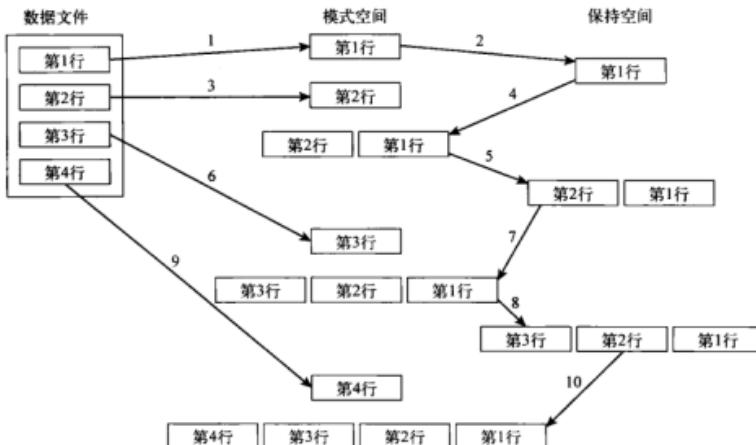


图20-1 使用保持空间来反转文本文件中行的顺序

这些都是你创建反转行sed编辑器脚本的片段。现在你可以试运行一下：

```
$ cat data2
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed -n '{!G : h : $p }' data2
This is the last line.
This is the second data line.
This is the first data line.
This is the header line.
$
```

sed编辑器脚本按期望执行了。脚本的输出反转了文本文件中原来的行。这演示了在sed脚本中使用保持空间的强大之处。它提供了在脚本输出中操作行顺序的简单办法。

说明 可能你想说，有个Linux命令已经有反转文本文件的功能了。tac命令将会倒序显示一个文本文件。你可能已经注意到这个命令的名字很巧妙，它执行的正好是cat命令的反向功能。

20.4 改变流

通常，sed编辑器会从脚本的顶部开始执行命令并一直处理到脚本的结尾（D命令是个例外，

它会强制sed编辑器返回到脚本的顶部，而不读取新的行）。sed编辑器提供了一个方法来改变命令脚本的流，生成的结果类似于结构化编程环境的结果。

20.4.1 跳转

在前面一节中，你了解了如何用感叹号命令来排除某个命令作用在某行上。sed编辑器提供了基于地址、地址模式或地址区间排除一整块命令的方法。这允许你只对数据流中的特定行执行一组命令。

跳转（branch）命令b的格式如下：

[address]b [label]

address参数决定了哪行或哪些行的数据会触发跳转命令。label参数定义了要跳转到的位置。如果没有加label参数，跳转命令会跳转到脚本的结尾：

```
$ sed '{2,3b :s/This is/Is this./;s/line./test?/}' data2
Is this the header test?
This is the first data line.
This is the second data line.
Is this the last test?
$
```

跳转命令为数据流中的第2行和第3行跳过了那两个替换命令。

你可以为跳转命令定义一个跳转到的标签，而不用直接跳到脚本的结尾。标签以冒号开始，最多可以有7个字符：

:label1

要指定标签，将它加到b命令后就行。使用标签允许你在匹配的跳转地址跳过一些命令，但仍执行脚本中的其他命令：

```
$ sed '/first/b jump1 ; s/This is the/No jump on/
> :jump1
> s/This is the/Jump here on/' data2
No jump on header line
Jump here on first data line
No jump on second data line
No jump on last line
$
```

跳转命令指定如果匹配文本first出现在该行了，程序应该跳到标为jump1的脚本行。如果跳转命令的模式没有匹配，sed编辑器会继续执行脚本中的命令，包括跳转标签后的命令（所以，3个替换命令都会在不匹配跳转模式的行上执行）。

如果某行匹配了跳转模式，则sed编辑器会跳转到跳转标签那行。因此，只有最后一个替换命令会执行。

这个例子演示了跳转到sed脚本后面的标签上。你也可以跳到脚本中前面的标签上，这样就达到了循环的效果：

```
$ echo "This, is, a, test, to, remove, commas." | sed -n '{
```

```
> :start
> s/.//lp
> b start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
```

脚本的每次迭代都会删除文本中的第一个逗号，并打印字符串。这个脚本有个问题：它永远不会结束。这就创建了一个无穷循环，会一直查找逗号，直到你最后用Ctrl+C组合键来给它发送一个信号，手动停止它。

要防止这个问题，可以为跳转命令指定一个地址模式来查找。如果没有模式，跳转就应该结束：

```
$ echo "This, is, a, test, to, remove, commas." | sed -n '{
> :start
> s/.//lp
> ./b start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
$
```

现在跳转命令只会在行中有逗号的情况下跳转。在最后一个逗号被删除后，跳转命令不会再执行，脚本也就能正常停止了。

20.4.2 测试

类似于跳转命令，测试（test）命令也用来改变sed编辑器脚本的流。测试命令会基于替换命令的输出跳转到一个标签，而不是基于地址跳转到一个标签。

如果替换命令成功匹配并替换了模式，测试命令就会跳转到指定的标签。如果替换命令未能匹配指定的模式，测试命令就不会跳转。

测试命令使用跟跳转命令相同的格式：

`[address]t [label]`

跟跳转命令一样，在没有指定标签的情况下，如果测试成功，sed会跳转到脚本的结尾。

测试命令提供了对数据流中的文本执行基本的if-then语句的一个低成本办法。举个例子，如果已经做了一个替换，不需要再做另一个替换，那么测试命令能帮上忙：

```
$ sed '{
> s/first/matched/
```

```
> t
> s/This is the/No match on/
> ]' data2
No match on header line
This is the matched data line
No match on second data line
No match on last line
$
```

第一个替换命令会查找模式文本first。如果它匹配了行中的模式，它会替换文本，而且测试命令会跳过后面的替换命令。如果第一个替换命令未能匹配模式，第二个命令就会被执行。

使用测试命令，你就能结束之前用跳转命令未能结束的循环：

```
$ echo "This, is, a, test, to, remove, commas." | sed -n '{
> :start
> s//.1p
> t start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
$
```

当没有替换要做时，测试命令不会跳转而是继续执行剩下的脚本。

20.5 模式替代

你已经了解了如何在sed命令中使用模式来替代数据流中的文本。然而在使用通配符时，很难知道哪些文本会匹配模式。

举个例子，假如你想在行中匹配的单词边上放两个问号。如果你只是要匹配模式中的一个单词，那非常简单：

```
$ echo "The cat sleeps in his hat." | sed 's/cat/"cat"/'
The "cat" sleeps in his hat.
$
```

但如果你在模式中用通配符(.)来匹配多个单词呢？

```
$ echo "The cat sleeps in his hat." | sed 's/.at/.at"/g'
The ".at" sleeps in his ".at".
$
```

替换字符串用点通配符来匹配at前面的一个字母。遗憾的是，替代字符串未能匹配要匹配单词的通配符。

20.5.1 and符号

sed编辑器提供了一个解决办法。and符号（&）用来代表替换命令中的匹配模式。不管匹配

预定义模式的是什么文本，你都能用and符号来在替代模式中调用它。这让你可以操作匹配预定义模式的任何单词：

```
$ echo "The cat sleeps in his hat." | sed 's/.at/"/g'
The "cat" sleeps in his "hat".
$
```

当模式匹配了单词cat，“cat”出现在了替换后的单词里。当它匹配了单词hat，“hat”就出现在了替换后的单词中。

20.5.2 替换单独的单词

and符号会提取匹配替换命令中指定模式的整个字符串。有时你只想提取这个字符串的一部分。可以那么做，只是要稍微花点心思。

sed编辑器用圆括号来定义替换模式的子字符串。然后你可以用替代模式中的特殊字符来引用每个子字符串。替代字符由反斜线和数字组成。数字表明子字符串模块的位置。sed编辑器会给第一个模块分配字符\1，给第二个模块分配字符\2，依此类推。

警告 当你在替换命令中使用圆括号时，你必须用转义字符来将它们识别为聚合字符而不是普通的圆括号。这跟转义其他特殊字符正好相反。

我们来看一个在sed编辑器脚本中使用这个特性的例子：

```
$ echo "The System Administrator manual" | sed '
> s/(\System\) Administrator/\1 User/'
The System User manual
$
```

这个替换命令用一对圆括号将单词System括起来，从而将它识别为子字符串模块。然后它在替代模式中使用\1来调用第一个识别的模块。这没什么特别的，但在处理通配符模式时特别有用。

如果需要用一个单词来替换一个短语，而这个单词刚好是该短语的子字符串，但那个子字符串碰巧使用了通配符，这时使用子字符串模块会方便很多：

```
$ echo "That furry cat is pretty" | sed 's/furry \(.at\)/\1/'
That cat is pretty
$ echo "That furry hat is pretty" | sed 's/furry \(.at\)/\1/'
That hat is pretty
$
```

在这种情况下，你不能用and符号，因为它会替换整个匹配的模式。子字符串模块提供了答案，允许你选择将模式中的哪部分作为替代模式。

当你需要在两个或多个子字符串模式中插入文本时，这个特性尤其有用。这里有个脚本，它使用子字符串模块来在长数字中插入逗号：

```
$ echo "1234567" | sed '{
> :start
```

```
> s/\(.*[0-9]\)\([0-9]{3}\)/\1.\2/
> t start
> ]
1,234,567
$
```

这个脚本将匹配的模式分成了两部分：

```
.*[0-9]
[0-9]{3}
```

这个模式会查找两个子字符串。第一个子字符串是以数字结尾的任意长度的字符。第二个字符串是一系列三位数字组成的块（关于如何在正则表达式中使用花括号的内容可参考第19章）。如果这个模式在文本中找到了，替代文本会在两个模块之间加一个逗号，每个模块都会通过其模块位置来识别。这个脚本使用测试命令来遍历这个数字，直到所有的逗号都放置好了。

20.6 在脚本中使用 sed

现在你已经了解了sed编辑器的各部分，可以将它们放在一起在shell脚本中使用了。本节将演示一些你在bash shell脚本中使用sed编辑器时应该知道的特性。

20.6.1 使用包装脚本

你可能已经注意到，实施sed编辑器脚本可能会很烦琐，尤其是脚本很长的话。你可以将sed编辑器命令放到shell包装脚本(wrapper)中，而不用每次使用时都重新键入整个脚本。包装脚本充当着sed编辑器脚本和命令行之间的中间人角色。

在shell脚本中，你可以将普通的shell变量及参数和sed编辑器脚本一起使用。这里有个将命令行参数变量作为sed脚本输入的例子：

```
$ cat reverse
#!/bin/bash
# shell wrapper for sed editor script to reverse lines

sed -n '{
1!G
h
$p
}' $1
$
```

名为reverse的shell脚本用sed编辑器脚本来反转数据流中的文本行。它使用\$1 shell参数来从第一个命令行提取第一个参数，这也应该是要反转的文件的名称：

```
$ ./reverse data2
This is the last line.
This is the second data line.
This is the first data line.
This is the header line.
$
```

现在你能在任何文件上轻松使用这个sed编辑器脚本，而不用每次都在命令行上重新输入。

20.6.2 重定向sed的输出

默认情况下，sed编辑器会将脚本的结果输出到STDOUT上。你可以在shell脚本中使用所有重定向sed编辑器输出的标准方法。

你可以在脚本中用反引号来将sed编辑器命令的输出重定向到一个变量中供后面使用。下面是个使用sed脚本来向数值计算结果添加逗号的例子：

```
$ cat fact
#!/bin/bash
#add commas to numbers in factorial answer

factorial=1
counter=1
number=$1

while [ $counter -le $number ]
do
    factorial=$(( $factorial * $counter ))
    counter=$(( $counter + 1 ))
done

result=`echo $factorial | sed '{
:start
s/[0-9]\([0-9]{3}\)/\1,\2/
t start
}'` 

echo "The result is $result"
$ ./fact 20
The result is 2,432,902,008,176,640,000
$
```

在你使用普通的阶乘计算脚本后，脚本的结果会被作为sed编辑器脚本的输入，它会给结果加上逗号。然后这个值会用在echo语句中产生最终结果。

20.7 创建 sed 实用工具

如你在本章前面的那些简短例子中看到的，用sed编辑器可以做很多很好用的数据格式化工作。本节将会介绍一些众所周知的容易上手的sed编辑器脚本来进行常见的数据处理工作。

20.7.1 加倍行间距

首先，让我们看一个向文本文件的行间插入空白行的简单sed脚本：

```
$ sed 'G' data2
```

```
This is the header line.  
This is the first data line.  
This is the second data line.  
This is the last line.
```

\$

看起来相当简单！这个技巧的关键在于保持空间的默认值。记住，G命令会简单地将保持空间内容附加到模式空间内容后。当你启动sed编辑器时，保持空间只有一个空行。将它附加到已有行后面，你就在已有行后面创建了一个空白行。

你可能已经注意到，这个脚本在数据流的最后一行后面也加了一个空白行，使得文件的末尾也产生了一个空白行。如果你不想要这个空白行，你可以用排除符号（!）和尾行符号 (\$) 来确保脚本不会将空白行加到数据流的最后一行后：

```
$ sed '$!G' data2  
This is the header line.  
  
This is the first data line.  
  
This is the second data line.  
  
This is the last line.  
$
```

现在看起来好一些了。只要该行不是最后一行，G命令就会附加保持空间内容。当sed编辑器到了最后一行时，它会跳过G命令。

20.7.2 对可能含有空白行的文件加倍行间距

将加倍行间距推进一步，如果文本文件已经有一些空白行，但你想给所有行加倍行间距怎么办呢？如果用前面的脚本，有些区域会有太多的空白行，因为每个已有的空白行也会被加倍：

```
$ cat data6  
This is line one.  
This is line two.  
  
This is line three.  
This is line four.  
$  
$ sed '$!G' data6  
This is line one.  
  
This is line two.  
  
This is line three.  
This is line four.  
$
```

现在在原来空白行的位置有3个空白行了。这个问题的解决办法是，首先删除数据流中的所有空白行，然后用G命令在所有行后插入新的空白行。要删除已有的空白行，你需要将d命令和一个匹配空白行的模式一起使用：

```
/$/d
```

这个模式使用了行首符号 (^) 和行尾符号 (\$)。将这个模式加到脚本中会生成想要的结果：

```
$ sed '/^$/d;:$!G' data6
This is line one.
```

```
This is line two.
```

```
This is line three.
```

```
This is line four.
```

```
$
```

完美！

20.7.3 给文件中的行编号

第18章演示了如何用等号来显示数据流中行的行号：

```
$ sed '=' data2
1
This is the header line.
2
This is the first data line.
3
This is the second data line.
4
This is the last line.
$
```

这可能有点难看，因为行号是在数据流中实际行的上面一行。比较好的解决办法是将行号和文本放在同一行。

你已经了解了如何用N命令合并行，在sed脚本中使用这个命令应该不难。这个工具的技巧在于，你不能将两个命令放到同一个脚本中。

一旦你有了等号命令的输出，你可以将输出管道输出给另一个sed编辑器脚本，它会使用N命令来合并这两行。你还需要用替换命令将换行符替换成空格或制表符。最终的解决办法看起来如下：

```
$ sed '=' data2 | sed 'N; s/\n/ /'
1 This is the header line.
2 This is the first data line.
3 This is the second data line.
4 This is the last line.
$
```

现在看起来好多了。在查看错误消息的行号时，这是一个很好用的小工具。

20.7.4 打印末尾行

到目前为止，你已经了解了如何用p命令来打印数据流中所有的行或匹配某个特定模式的行。如果你只要处理一个长输出（比如日志文件）中的末尾几行，怎么办？

美元符代表数据流中最后一行，所以只显示最后一行很容易：

```
$ sed -n '$p' data2
This is the last line.
$
```

那么，如何用美元符来显示数据流末尾的若干行呢？答案是创建滚动窗口（rolling window）。

滚动窗口是检验模式空间中文本行组成的块的常用方法，它会用N命令将它们合并起来。N命令将下一行文本附加到已在模式空间中的文本行后面。一旦你在模式空间有了一块10行的文本，你可以用美元符来检查你是否在数据流的尾部。如果不在结尾，就继续向模式空间增加行，并删除原来的行（记住D命令，它会删除模式空间的第一行）。

通过循环N命令和D命令，你向模式空间的文本行块增加了新行，同时也删除了旧行。跳转命令完美适用于这个循环。要结束循环，只要识别最后一些并用q命令退出就可以了。

最终的sed编辑器脚本看起来如下：

```
$ sed '{
> :start
> $q
> N
> 11,$D
> b start
> }' /etc/passwd
user:x:1000:1000:user...:/home/user:/bin/bash
polkituser:x:113:121:PolicyKit...:/var/run/PolicyKit:/bin/false
sshd:x:114:65534::/var/run/sshd:/usr/sbin/nologin
Samantha:x:1001:1002:Samantha_4...:/home/Samantha:/bin/bash
Debian-exim:x:115:124::/var/spool/exim4:/bin/false
usbmux:x:116:46:usbmux daemon...:/home/usbmux:/bin/false
rtkit:x:117:125:RealtimeKit...:/proc:/bin/false
Timothy:x:1002:1005::/home/Timothy:/bin/sh
Christine:x:1003:1006::/home/Christine:/bin/sh
kdm:x:118:65534::/home/kdm:/bin/false
$'
```

这个脚本会首先检查这行是不是数据流中最后一行。如果是，退出（quit）命令会停止循环。N命令会将下一行附加到模式空间的当前行后。如果当前行在第10行后面，11,\$D命令会删除模式空间中的第一行。这在模式空间中创建了滑动窗口效果。

20.7.5 删除行

另一个有用的sed编辑器工具是删除数据流中不需要的空白行。删除数据流中的所有空白行很容易，但要选择性地删除空白行则需要一点创造力了。本节将会给出一些简短的sed编辑器脚本，它们可以用来帮助删除数据中不需要的空白行。

1. 删除连续的空白行

多余的空白行出现在数据文件中会非常令人生厌。通常数据文件中会有空白行，但有时数据行缺失会产生太多的空白行（如你在前面的加倍行间距例子中所见）。

删除连续空白行的最简单办法是用地址区间来检查数据流。第18章介绍了如何在地址中使用区间，包括如何在地址区间中加入模式。sed编辑器会对所有匹配指定地址区间的行执行该命令。

删除连续空白行的关键在于，创建一个包含一个非空白行和一个空白行的地址区间。如果sed编辑器遇到了这个区间，它不会删除行。但对于不匹配这个区间的行（两个或更多的空白行），它会删除这些行。

下面是完成这个操作的脚本：

```
/. /!$/!d
```

区间是/. /到/^\$/。区间的开始地址会匹配任何含有至少一个字符的行。区间的结束地址会匹配一个空行。在这个区间内的行不会被删除。

下面是实际的脚本：

```
$ cat data6
This is the first line.
```

```
This is the second line.
```

```
This is the third line.
```

```
This is the fourth line.
```

```
$
$ sed '/. /!$/!d' data6
This is the first line.
```

```
This is the second line.
```

```
This is the third line.
```

```
This is the fourth line.
```

```
$
```

不管文件中数据行之间出现了多少空白行，输出只会在行间放一个空白行。

2. 删除开头的空白行

数据文件开头有多个空白行时也会很令人生厌。通常将数据从文本文件中导入到数据库时，空白行会创建一些空项，让使用这些数据的计算都没法进行。

删除数据流顶部的空白行不难。下面是完成这个功能的脚本：

```
/. /,$!d
```

这个脚本用地址区间来决定哪些行要删掉。这个区间从含有字符的行开始，一直到数据流结束。在这个区间内的任何行都不会从输出中删除。这意味着含有字符的第一行之前的任何行都会删除。

看看实用中的这个简单脚本：

```
$ cat data7
This is the first line.
This is the second line.
$
$ sed '/./,$!d' data7
This is the first line.
This is the second line.
$
```

测试文件在数据行之前有两个空白行。这个脚本成功地删除了开头的两个空白行，而没有动数据中的空白行。

3. 删除结尾的空白行

很遗憾，删除结尾的空白行并不像删除开头的空白行那么容易。就跟打印数据流的结尾一样，删除数据流结尾的空白行也需要一点创造力，以及循环。

在开始讨论前，让我们先看看脚本是什么样的：

```
sed '{
:start
/\n*/{$d; N; b start }
}'
```

这可能乍一看有点奇怪。注意，在正常脚本的花括号里还有花括号。这允许你在整个命令脚本中将一些命令聚合在一起。该命令组将会作用到指定的地址模式上。地址模式匹配只含有一个换行符的任意行。当找到了一行，如果是最后一行，删除命令会删掉它。如果不是最后一行，N命令会将下一行附加到它后面，跳转命令会跳到起始点重新开始。

下面是实际的脚本：

```
$ cat data8
This is the first line.
This is the second line.
```

```
$
$ sed '{
:start
/\n */{$d; N; b start }
}' data8
This is the first line.
This is the second line.
$
```

这个脚本成功删除了文本文件结尾的空白行。

20.7.6 删 除 HTML 标签

现如今，从网站下载文本并将其保存或用作应用程序的数据并不罕见。但有时，当你从网站

下载文本时，你也下载了格式化数据用的HTML标签。当你想要看的只是数据时，这会是个问题。

标准的HTML Web页面包含几种不同类型的HTML标签，标明需要的正确显示页面信息的格式化功能。这里有个HTML文件的例子：

```
$ cat data9
<html>
<head>
<title>This is the page title</title>
</head>
<body>
<p>
This is the <b>first</b> line in the Web page. This should provide
some <i>useful</i> information for us to use in our shell script.
</body>
</html>
$
```

HTML标签由大于号和小于号来识别。大多数HTML标签都是成对出现的：一个标签开始格式化过程（比如****用来加粗），另一个标签结束格式化过程（比如****用来结束加粗）。

但如果你不小心的话，删除HTML标签可能会带来问题。乍一看，你可能认为删除HTML标签的办法就是查找以小于号（<）开头大于号（>）结尾、符号间有数据的文本字符串：

```
s/<.*>/ /g
很遗憾，这个命令会出现一些意料之外的结果：
```

```
$ sed 's/<.*>/ /g' data9
```

```
This is the line in the Web page. This should provide
some information for us to use in our shell script.
```

```
$
```

注意，标题文本以及加粗和倾斜的文本都不见了。sed编辑器将这个脚本字面理解为大于号和小于号之间的任何文本，包括其他的大于号和小于号。每次文本被HTML标签（比如**first**）包围，这个sed脚本会删掉整个文本。

这个问题的解决办法是让sed编辑器忽略掉任何嵌入到原始标签中的大于号。要这么做，你可以创建一个字符组来排除大于号。脚本改为：

```
s/<[^>]*>/ /g
这个脚本现在能够正常工作了，它会显示你要在Web页面HTML代码里看的数据：
```

```
$ sed 's/<[^>]*>/ /g' data9
```

```
This is the page title
```

This is the first line in the Web page. This should provide some useful information for us to use in our shell script.

\$

好一些了。要清理一下，你可以加一条删除命令来删除多余的空白行：

```
$ sed '/<[^>]*>/g;/$d' data9
This is the page title
This is the first line in the Web page. This should provide
some useful information for us to use in our shell script.
$
```

现在清爽多了，只有你要看的数据。

20.8 小结

sed编辑器提供了一些高级特性，允许你处理跨多行的文本模式。本章介绍了如何使用next命令来提取数据流中的下一行并将它放到模式空间中。只要在模式空间中了，你就可以执行复杂的替换命令来替换跨行的短语了。

多行删除命令允许在模式空间含有两行或更多行时删除第一行。这是遍历数据流中多行的简便办法。类似地，多行打印命令允许在模式空间含有两行或更多行时只打印第一行。多行命令的组合允许遍历数据流并创建多行替换系统。

紧接着，本章讨论了保持空间。保持空间允许在处理多行文本时先将某些文本行搁置在一边。你可以在任何时间取回保持空间的内容来替换模式空间的文本或附加到模式空间文本后。使用保持空间允许你对数据流排序，反转文本行在数据中出现的顺序。

本章还讨论了sed编辑器的流控制命令。跳转命令为你提供了改变脚本中sed编辑器命令常规流的途径，在特定条件下创建循环或跳过命令。测试命令为sed编辑器命令脚本提供了if-then类型的语句。测试命令只有在前面的替换命令成功替换行中文本的情况下才会跳转。

本章以如何在shell脚本中使用sed脚本的讨论结尾。对于大型sed脚本来说，常用的方法是将脚本放到shell包装脚本中。你可以在sed脚本中使用命令行参数变量来传递shell命令行的值。这为在命令行上甚至在其他脚本中直接使用sed编辑器脚本提供了一个简便的途径。

下一章将会深入介绍gawk世界。gawk程序支持许多高级编程语言特性。你可以只用gawk就创建一些相当复杂的数据操作及报告程序。下一章将会介绍各种编程语言特性，并演示如何用它们来从简单数据生成自己的漂亮的报告。

本章内容

- 重新审视gawk
- 在gawk程序中使用变量
- 使用结构化命令
- 格式化打印
- 操作函数

第21章 18章介绍了gawk程序，并演示了用它从原始数据文件生成格式化报告的基本方法。本章将进一步深入了解如何定制gawk来生成报告。gawk是一门功能丰富的编程语言，允许你通过编写高级程序来处理数据。如果你在接触Shell脚本前用过其他编程语言，那么gawk会让你感到十分亲切。在本章中，你将会了解如何使用gawk编程语言来完成可能遇到的各种数据格式化任务。

21.1 使用变量

所有编程语言共有的一个重要特性是使用变量来存取值。gawk编程语言支持两种不同类型的变量：

- 内建变量；
- 自定义变量。

gawk有一些内建变量。这些变量存放用来处理数据文件中的数据字段和数据行的信息。你也可以在gawk程序里创建你自己的变量。下面几节将带你逐步了解如何在gawk程序里使用变量。

21.1.1 内建变量

gawk程序使用内建变量来引用程序数据里的一些特殊功能。本节将介绍gawk程序中可用的内建变量并演示如何使用它们。

1. 字段和数据行分隔符变量

第18章演示了gawk中的一种内建变量类型——数据字段变量。数据字段变量允许你使用美元

符号 (\$) 和数据字段在数据行中位置对应的数值来引用该数据行中的字段。因此，要引用数据行中的第一个数据字段，就用变量 \$1；要引用第二个字段，就用 \$2；依次类推。

字段是由字段分隔符来划定的。默认情况下，字段分隔符是一个空白字符，也就是空格符或者制表符 (tab)。第18章讲了如何在命令行下使用命令行参数 -F 或者在 gawk 程序中使用特殊的内建变量 FS 来更改字段分隔符。

内建变量 FS 是控制 gawk 如何处理输入输出数据中的字段和数据行的一组变量中的一个。表 21-1 列出了该组内建变量。

表 21-1 gawk 数据字段和数据行变量

变 量	描 述
FIELDWIDTHS	由空格分隔开的定义了每个数据字段确切宽度的一列数字
FS	输入字段分隔符
RS	输入数据行分隔符
OFS	输出字段分隔符
ORS	输出数据行分隔符

变量 FS 和 OFS 定义了 gawk 如何处理数据流中的数据字段。你已经了解了如何使用变量 FS 来定义什么字符分割数据行中的字段。变量 OFS 具备相同的功能，不过是用在 print 命令的输出上。

默认情况下，gawk 将 OFS 设成一个空格，所以如果你用命令

```
print $1,$2,$3
```

你会看到如下输出：

```
field1 field2 field3
```

在下面的例子中，你能看到这点：

```
$ cat data1
data11,data12,data13,data14,data15
data21,data22,data23,data24,data25
data31,data32,data33,data34,data35
$ gawk 'BEGIN{FS=".":} {print $1,$2,$3}' data1
data11 data12 data13
data21 data22 data23
data31 data32 data33
$
```

print 命令会自动将 OFS 变量的值放置在输出的每个字段间。通过设置 OFS 变量，你可以在输出中使用任意字符（串）来分割字段：

```
$ gawk 'BEGIN{FS="-"; OFS="-"} {print $1,$2,$3}' data1
data11-data12-data13
data21-data22-data23
data31-data32-data33
$ gawk 'BEGIN{FS="--"; OFS="--"} {print $1,$2,$3}' data1
data11--data12--data13
data21--data22--data23
```

```
data31--data32--data33
$ gawk 'BEGIN{FS=""; OFS="<-->"} {print $1.$2.$3}' data1
data1<-->data12<-->data13
data21<-->data22<-->data23
data31<-->data32<-->data33
$
```

FIELDWIDTHS变量允许你读取数据行，而不用字段分隔符来划分字段。在一些应用程序中，不用字段分隔符，数据是被放置在数据行的某些列中的。这种情况下，你必须设定FIELDWIDTHS变量来匹配数据在数据行中的位置。

一旦设置了FIELDWIDTHS变量，gawk就会忽略FS变量，而根据提供的字段宽度大小来计算字段。下面是个采用字段宽度而非字段分隔符的例子：

```
$ cat data1a
1005.3247596.37
115-2.349194.00
05810.1298100.1
$ gawk 'BEGIN{FIELDWIDTHS="3 5 2 5"}{print $1.$2.$3.$4}' data1b
100 5.324 75 96.37
115 -2.34 91 94.00
058 10.12 98 100.1
$
```

FIELDWIDTHS变量定义了4个字段，gawk依此来解析数据行。每个数据行中用以表示数字的字符串根据定义好的字段宽度值来分割。

警告 一定要记住，一旦设定了FIELDWIDTHS变量的值，就不能改变了。这种方法并不适用于变长的字段。

变量RS和ORS定义了gawk程序如何处理数据流中的数据行。默认情况下，gawk将RS和ORS设为换行符。默认的RS值表明，输入数据流中的每行新文本就是一个新数据行。

有时，你会碰到在数据流中字段占了多行的情况。经典的例子是包含地址和电话号码的数据，其中地址和电话号码各占一行：

```
Riley Mullen
123 Main Street
Chicago, IL 60601
(312)555-1234
```

如果你用默认的FS和RS变量值来读取这组数据，gawk就会把每行误读为一个单独的数据行，并把数据行中的每个空格当做字段分隔符。这绝非你想要的。

要解决这个问题，只需把FS变量设置成换行符。这就表明数据流中的每行都是一个单独的字段，每行上的所有数据都属于同一个字段。但现在令你头疼的是无从判断一个新的数据行从何开始。

要解决这个问题，只需把RS变量设置成空字符串，然后在数据行间留一个空白行。gawk会把每个空白行当作一个数据行分隔符。

下面就是个使用这种方法的例子：

```
$ cat data2
Riley Mullen
123 Main Street
Chicago, IL 60601
(312)555-1234

Frank Williams
456 Oak Street
Indianapolis, IN 46201
(317)555-9876

Haley Snell
423 Elm Street
Detroit, MI 48201
(313)555-4938
$ gawk 'BEGIN{FS="\n"; RS=""} {print $1,$4}' data2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

太好了，现在gawk把文件中的每行都当成一个字段，把空白行当做数据行分隔符。

2. 数据变量

除了字段和数据行分隔符变量外，gawk还提供了一些其他的内建变量来帮助你了解数据发生了什么变化并提取shell环境的信息。表21-2列出了gawk中的其他内建变量。

表21-2 更多的gawk内建变量

变 量	描 述
ARGC	当前命令行参数个数
ARGVIND	当前文件在ARGV中的位置
ARGV	包含命令行参数的数组
CONVFMT	数字的转换格式（参见printf语句）；默认值为%.6 g
ENVIRON	当前shell环境变量及其值组成的关联数组
ERRNO	当读取或关闭输入文件发生错误时的系统错误号
FILENAME	用作gawk输入数据的数据文件的文件名
FNR	当前数据文件中的数据行数
IGNORECASE	设成非零值时，忽略gawk命令中出现的字符串的字符大小写
NF	数据文件中的字段总数
NR	已处理的输入数据行数目
OFMT	数字的输出格式；默认值为%.6 g
RLENGTH	由match函数所匹配的子字符串的长度
RSTART	由match函数所匹配的子字符串的起始位置

你应该能从 shell 脚本编程中认识上面的一些变量。ARGC 和 ARGV 变量允许从 shell 中获得命令行参数的总数以及它们的值。但这可能有点麻烦，因为 gawk 并不会将程序脚本当成命令行参数的一部分：

```
$ gawk 'BEGIN{print ARGC,ARGV[1]}' data1
2 data1
$
```

ARGC 变量表明命令行上有两个参数。这包括 gawk 命令和 data1 参数（记住，程序脚本并不算参数）。ARGV 数组从代表该命令的索引 0 开始。第一个数组值是 gawk 命令后的第一个命令行参数。

说明 注意，跟 shell 变量不同，在脚本中引用 gawk 变量时，变量名前不加美元符。

ENVIRON 变量看起来可能有点陌生。它使用关联数组来提取 shell 环境变量。关联数组用文本作为数组的索引值，而不用数值。

数组索引中的文本是 shell 环境变量，而数组的值则是 shell 环境变量的值。下面有个例子：

```
$ gawk '
> BEGIN{
>   print ENVIRON["HOME"]
>   print ENVIRON["PATH"]
> }
> /home/rich
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
$
```

ENVIRON["HOME"] 变量从 shell 中提取了 HOME 环境变量的值。类似地，ENVIRON["PATH"] 提取了 PATH 环境变量的值。你可以用这种方法来从 shell 中提取任何环境变量的值来在 gawk 程序中使用。

当你要在 gawk 程序中记录数据字段和数据行时，FNR、NF 和 NR 变量就能派上用场。有时你不知道数据行中到底有多少个数据字段。NF 变量允许你指定数据行中的最后一个数据字段，而不用知道它的具体位置：

```
$ gawk 'BEGIN{FS=":"; OFS=":"} {print $1,$NF}' /etc/passwd
rich:/bin/bash
testy:/bin/csh
mark:/bin/bash
dan:/bin/bash
mike:/bin/bash
test:/bin/bash
$
```

NF 变量含有数据文件中最后一个数据字段的数字值。你可以在它前面加个美元符将它用作字段变量。

FNR 和 NR 变量彼此类似，但略有不同。FNR 变量含有处理过的当前数据文件中的数据行总数，NF 变量则含有处理过的所有数据行总数。让我们看几个例子来了解一下这个差别：

```
$ gawk 'BEGIN{FS=":"}{print $1,"FNR=$FNR"}' data1 data1
data1 FNR=1
```

```
data21 FNR=2
data31 FNR=3
data11 FNR=1
data21 FNR=2
data31 FNR=3
$
```

在这个例子中，gawk程序的命令行定义了两个输入文件（它两次指定了同样的输入文件）。这个脚本会打印第一个数据字段的值和FNR变量的当前值。注意，当gawk程序处理第二个数据文件时，FNR值被设回1了。

现在，让我们加上NR变量看看会输出什么：

```
$ gawk '
> BEGIN {FS=".}
> {print $1,"FNR="FNR,"NR="NR}
> END{print "There were",NR,"records processed"}' data1 data1
data11 FNR=1 NR=1
data21 FNR=2 NR=2
data31 FNR=3 NR=3
data11 FNR=1 NR=4
data21 FNR=2 NR=5
data31 FNR=3 NR=6
There were 6 records processed
$
```

FNR变量的值在gawk处理第二个数据文件时被重置了，而NR变量则在进入第二个数据文件后继续计数。结果是，如果只使用一个数据文件作为输入，那么FNR和NR的值将会相同。如果使用多个数据文件作为输入，那么FNR的值会在处理每个数据文件时被重置，而NR的值则会继续计数直到处理完所有的数据文件。

说明 在使用gawk时你可能会注意到，gawk脚本通常会比shell脚本中其他部分还要大一些。在本章的例子中，简单起见，我们利用shell的多行功能直接在命令行上运行了gawk脚本。当你在shell脚本中使用gawk时，你应该将不同的gawk命令放到不同的行，这样会比较容易阅读和理解，而不要在shell脚本中将所有的命令都塞到同一行。还有，如果你发现在不同的shell脚本中用到了同样的gawk脚本，记得将这段gawk脚本放到一个单独的文件中，并用-f参数来在shell脚本中引用它（参见第18章）。

21.1.2 自定义变量

跟任何其他经典编程语言一样，gawk允许你定义自己的变量来在程序代码中使用。gawk自定义变量名可以是任意数目的字母、数字和下划线，但不能以数字开头。还有，要记住gawk变量名区分大小写。

1. 在脚本中给变量赋值

在gawk程序中给变量赋值跟在shell脚本中赋值类似，都用赋值语句：

```
$ gawk '
> BEGIN{
> testing="This is a test"
> print testing
> }'
This is a test
$
```

print语句的输出是testing变量的当前值。跟shell脚本变量一样，gawk变量可以保存数值和文本值：

```
$ gawk '
> BEGIN{
> testing="This is a test"
> print testing
> testing=45
> print testing
> }'
This is a test
45
$
```

在这个例子中，testing变量的值会从文本值变成数值。

赋值语句还可以包含数学算式来处理数字值：

```
$ gawk 'BEGIN{x=4; x= x * 2 + 3; print x}'
11
$
```

如你在这个例子中看到的，gawk编程语言包含了用来处理数字值的标准数学操作符。其中包括求余符号（%）和幂运算符号（^或**）。

2. 在命令行上给变量赋值

你也可以用gawk命令行来给程序中的变量赋值。这允许你在普通代码的外面赋值，即时改变变量的值。这里有个使用命令行变量来显示文件中特定数据字段的例子：

```
$ cat script1
BEGIN{FS=","}
{print $n}
$ gawk -f script1 n=2 data1
data12
data22
data32
$ gawk -f script1 n=3 data1
data13
data23
data33
$
```

这个特性允许你改变脚本的行为而不需要修改实际的脚本代码。第一个例子显示了文件的第二个数据字段，而第二个例子显示了第三个数据字段，只要在命令行上设置了n变量的值就行。

使用命令行参数来定义变量值会有个问题。在你设置了变量后，这个值在代码的BEGIN部分不可用：

```
$ cat script2
BEGIN{print "The starting value is",n; FS=".,"}
{print $n}
$ gawk -f script2 n=3 data1
The starting value is
data13
data23
data33
$
```

你可以用-v命令行参数来解决这个问题。它允许你指定在BEGIN代码部分之前设定的变量。在命令行上，-v命令行参数必须放在脚本代码之前。

```
$ gawk -v n=3 -f script2 data1
The starting value is 3
data13
data23
data33
$
```

现在n变量在BEGIN代码部分中已经含有命令行上设的值了。

21.2 处理数组

许多编程语言都提供数组来在单个变量中存储多个值。gawk编程语言使用关联数组来提供数组功能。

关联数组跟数字数组不同之处在于它的索引值可以是任意文本字符串。你不需要用连续的数字来标识数组中的数据元素。相反，关联数组用各种字符串来引用值。每个索引字符串都必须是唯一的，并唯一地标识赋给它的数据元素。如果你熟悉其他编程语言的话，这跟哈希表和字典是同一个概念。

后面几节将会带你逐步熟悉在gawk程序中使用关联数组。

21.2.1 定义数组变量

可以用标准赋值语句来定义数组变量。数组变量赋值的格式如下：

```
var[index] = element
```

其中var是变量名，index是关联数组的索引值，element是数据元素值。这里有些gawk中数组变量的例子：

```
capital["Illinois"] = "Springfield"
capital["Indiana"] = "Indianapolis"
capital["Ohio"] = "Columbus"
```

在引用数组变量时，必须包含索引值来提取相应的数据元素值：

```
$ gawk 'BEGIN{
> capital["Illinois"] = "Springfield"
> print capital["Illinois"]'
```

```
> }'
Springfield
$
```

在引用数组变量时，数据元素的值会出现。数据元素值是数字值时也一样：

```
$ gawk 'BEGIN{
> var[1] = 34
> var[2] = 3
> total = var[1] + var[2]
> print total
> }'
37
$
```

如你在这个例子中看到的，可以像使用gawk程序中的其他变量一样使用数组变量。

21.2.2 遍历数组变量

关联数组变量的问题在于你可能无法知晓索引值是什么。跟使用连续数字作为索引值的数字数组不同，关联数组的索引可以是任何东西。

如果要在gawk中遍历一个关联数组，你可以用for语句的一种特殊形式：

```
for (var in array)
{
    statements
}
```

这个for语句会在每次将关联数组array的下一个索引值赋给变量var时，执行一遍statements。重要的是记住这个变量是索引值而不是数组元素值。你可以将这个变量用作数组的索引，轻松地取出数据元素值：

```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> var["m"] = 3
> var["u"] = 4
> for (test in var)
> {
>     print "Index:", test, " - Value:", var[test]
> }
> }'
Index: u - Value: 4
Index: m - Value: 3
Index: a - Value: 1
Index: g - Value: 2
$
```

注意，索引值不会按任何特定顺序返回，但它们每个都会有个对应的数据元素值。明白这点很重要，因为你不能指望返回的值都是按顺序的，你只能确定索引值和数据值是对应的。

21.2.3 删除数组变量

从关联数组中删除数组索引要用一个特别的命令：

```
delete array[index]
```

删除命令会从数组中删除关联索引值和相关的数据元素值。

```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> for (test in var)
> {
>   print "Index:" . test . " - Value:" . var[test]
> }
> delete var["g"]
> print "---"
> for (test in var)
>   print "Index:" . test . " - Value:" . var[test]
> }'
Index: a - Value: 1
Index: g - Value: 2
---
Index: a - Value: 1
$
```

一旦从关联数组中删除了索引值，你就没法再提取它了。

21.3 使用模式

gawk程序支持几种类型的匹配模式来过滤数据行，跟sed编辑器大同小异。第18章已经介绍了使用中的两种特别模式。BEGIN和END关键字是用来在读取数据流之前或之后执行命令的特殊模式。类似地，你可以创建其他模式来在数据流中出现匹配数据时执行一些命令。

本节将会演示如何在gawk脚本中用匹配模式来限定程序脚本作用在哪些数据行上。

21.3.1 正则表达式

第19章介绍了如何将正则表达式用作匹配模式。你可以用基本正则表达式（BRE）或扩展正则表达式（ERE）来过滤程序脚本作用在数据流中哪些行上。

在使用正则表达式时，正则表达式必须出现在它要控制的程序脚本的左花括号前：

```
$ gawk 'BEGIN{FS=".") /11/{print $1}' data11
data11
$
```

正则表达式/11/匹配了数据字段中含有字符串11的数据行。gawk程序会用正则表达式对数据行中所有的数据字段进行匹配，包括字段分隔符：

```
$ gawk 'BEGIN{FS=".") /.d/{print $1}' data11
data11
```

```
data21
data31
$
```

这个例子在正则表达式中匹配了用作字段分隔符的逗号。这也并不总是好的。它可能会造成试图匹配某个数据字段中的特定数据，而这些数据也可能出现在其他数据字段中。如果需要用一个正则表达式来对一个特定数据实例进行匹配，你应该使用匹配操作符。

21.3.2 匹配操作符

匹配操作符（matching operator）允许将正则表达式限定在数据行中的特定数据字段。匹配操作符是波浪线（~）。你要一起指定匹配操作符、数据字段变量以及要匹配的正则表达式：

```
$1 ~ /data/
```

\$1变量代表数据行中的第一个数据字段。这个表达式会过滤出第一个字段以文本data开头的所有数据行。下面是在gawk程序脚本中使用匹配操作符的例子：

```
$ gawk 'BEGIN{FS=".":} $2 ~ /data2/{print $0}' data1
data21,data22,data23,data24,data25
$
```

匹配操作符会用正则表达式/^data2/来匹配第二个数据字段，该正则表达式指明字符串要以文本data2开头。

这是个gawk程序脚本中常用的在数据文件中查找特定数据元素的强大工具：

```
$ gawk -F: '$1 ~ /rich/{print $1,$NF}' /etc/passwd
rich /bin/bash
$
```

这个例子会在第一个数据字段中查找文本rich。当它在数据行中找到了这个模式时，它会打印数据行的第一个和最后一个数据字段值。

你也可以用!符号来排除正则表达式的匹配：

```
$1 !~ /expression/
```

如果数据行中没有找到匹配正则表达式的文本，那程序脚本就会作用到数据行数据：

```
$ gawk '!F: '$1 !~ /rich/{print $1,$NF}' /etc/passwd
root /bin/bash
daemon /bin/sh
bin /bin/sh
sys /bin/sh
... output truncated ...
$
```

在这个例子中，gawk程序脚本会打印/etc/passwd文件中所有不匹配用户ID rich的数据行的用户名ID和登录shell。

21.3.3 数学表达式

除了正则表达式，你也可以在匹配模式中用数学表达式。这个功能在匹配数据字段中的数字

值时非常有用。举个例子，如果你想显示所有属于root用户组（组ID为0）的系统用户，你可以用这个脚本：

```
$ gawk -F: '$4 == 0{print $1}' /etc/passwd
root
sync
shutdown
halt
operator
$
```

这段脚本会查看第4个数据字段含有值0的数据行。在这个Linux系统中，有5个用户账户属于root用户组。

你可以使用任意的普通数学比较表达式。

- $x == y$: 值 x 等于 y 。
- $x <= y$: 值 x 小于等于 y 。
- $x < y$: 值 x 小于 y 。
- $x >= y$: 值 x 大于等于 y 。
- $x > y$: 值 x 大于 y 。

也可以对文本数据使用表达式，但必须小心。跟正则表达式不同，表达式必须完全匹配。数据必须跟模式正好匹配：

```
$ gawk -F: '$1 == "data"{print $1}' data1
$
$ gawk -F: '$1 == "data11"{print $1}' data1
data11
$
```

第一个测试没有匹配任何数据行，因为第一个数据字段的值不是任何数据行中的数据。第二个测试用值data11匹配了一个数据行。

21.4 结构化命令

gawk编程语言支持常见的结构化编程命令。本节将会介绍每个命令并演示如何在gawk编程环境中使用它们。

21.4.1 if语句

gawk编程语言支持标准的if-then-else格式的if语句。你必须为if语句定义一个评估的条件，并将其用圆括号括起来。如果条件评估为TRUE，紧跟在if语句后的语句会执行。如果条件评估为FALSE，那这条语句就会被跳过。可以用这种格式：

```
if (condition)
    statement1
```

或者你可以将它放在一行上，像这样：

```
if (condition) statement1
```

这里有个演示这种格式的简单的例子：

```
$ cat data4
10
5
13
50
34
$ gawk '{if ($1 > 20) print $1}' data4
50
34
$
```

并不复杂。如果需要在if语句中执行多条语句，你必须用花括号将它们括起来：

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> }
> }' data4
100
68
$
```

注意，不能弄混if语句的花括号和用来开始和结束程序脚本的花括号。如果弄混了，gawk程序能发现缺花括号并产生一条错误消息：

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> }' data4
gawk: cmd. line:7: (END OF FILE)
gawk: cmd. line:7: parse error
$
```

gawk的if语句也支持else子句，允许在if语句条件不成立的情况下执行一条或多条语句。这里有个使用else子句的例子：

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> } else
> {
>   x = $1 / 2
>   print x
> }}' data4
5
```

```
2.5
6.5
100
68
$
```

可以在单行上使用else子句，但必须在if语句部分之后使用分号：

```
if (condition) statement1; else statement2
```

这里是上一个例子的单行格式版本：

```
$ gawk '{if ($1 > 20) print $1 * 2; else print $1 / 2}' data4
5
2.5
6.5
100
68
$
```

这个格式更紧凑，但也更难理解。

21.4.2 while语句

while语句为gawk程序提供了一个基本的循环功能。下面是while语句的格式：

```
while (condition)
{
    statements
}
```

while循环允许遍历一组数据，并检查结束迭代的条件。在计算中必须使用每个数据行中的多个数据值时，它能帮得上忙：

```
$ cat data5
130 120 135
160 113 140
145 170 215
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
>     total += $i
>     i++
> }
> avg = total / 3
> print "Average:".avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
$
```

while语句会遍历数据行中的数据字段，将每个值都加到total变量上，然后将计数器变量i

增一。当计数器值等于4时，while的条件变成了FALSE，循环结束，然后会执行脚本中的下条命令。那条语句会计算平均值，然后平均值会打印出来。这个过程会为数据文件中的每个数据行不断重复。

gawk编程语言支持在while循环中使用break和continue语句，允许从循环中跳出：

```
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
>     total += $1
>     if (i == 2)
>         break
>     i++
> }
> avg = total / 2
> print "The average of the first two data elements is:",avg
> }' data5
The average of the first two data elements is: 125
The average of the first two data elements is: 136.5
The average of the first two data elements is: 157.5
$
```

break语句用来在i变量的值为2时从while循环中跳出。

21.4.3 do-while语句

do-while语句类似于while语句，但会在检查条件语句之前执行命令。下面是do-while语句的格式：

```
do
{
    statements
} while (condition)
```

这种格式保证了语句会在条件被评估之前至少执行一次。当你需要在条件被评估之前执行一些语句时这非常有用：

```
$ gawk '{
> total = 0
> i = 1
> do
> {
>     total += $1
>     i++
> } while (total < 150)
> print total }' data5
250
160
315
$
```

这个脚本会从每个数据行读取数据字段并将它们加在一起，直到累加结果达到150。如果第一个数据字段大于150（如在第二个数据行中看到的），则脚本会保证在条件被评估前至少读取第一个数据字段。

21.4.4 for语句

for语句是许多编程语言用来做循环的常见方法。gawk编程语言支持C风格的for循环：

```
for( variable assignment; condition; iteration process)
```

它帮助将几个功能合并到一个语句中来简化循环：

```
$ gawk '{  
> total = 0  
> for (i = 1; i < 4; i++)  
> {  
>     total += $i  
> }  
> avg = total / 3  
> print "Average:", avg  
> }' data5  
Average: 128.333  
Average: 137.667  
Average: 176.667  
$
```

定义了for循环中的迭代计数器，你就不用担心要像使用while语句一样自己负责给计数器增一。

21.5 格式化打印

你可能已经注意到了print语句在gawk如何显示数据上并未提供多少控制。你能做的大概只是控制输出字段分隔符（OFS）。如果你正在创建详细的报告，通常你需要将数据按特定的格式放到特定的位置。

解决办法是使用格式化打印命令，称为printf。如果你熟悉C语言编程的话，gawk中的printf命令用法一样，允许指定具体的如何显示数据的指令。

下面是printf命令的格式：

```
printf "format string", var1, var2 . . .
```

format string是格式化输出的关键。它会用文本元素和格式化指定符来具体指定如何呈现格式化输出。格式化指定符是一种特殊的代码，它会指明什么类型的变量可以显示以及如何显示。gawk程序会将每个格式化指定符作为命令中列出的每个变量的占位符使用。第一个格式化指定符会匹配列出的第一个变量，第二个会匹配第二个变量，依此类推。

格式化指定符采用如下格式：

```
%[modifier]control-letter
```

其中control-letter是指明显示什么类型数据值的单字符码，而modifier定义了另一个可选的格式化特性。表21-3列出了可用在格式化指定符中的控制字母。

表21-3 格式化指定符的控制字母

控制字母	描述
c	将一个数作为ASCII字符显示
d	显示一个整数值
i	显示一个整数值（跟d一样）
e	用科学计数法显示一个数
f	显示一个浮点值
g	用科学计数法或浮点数中较短的显示
o	显示一个八进制值
s	显示一个文本字符串
x	显示一个十六进制值
X	显示一个十六进制值，但用大写字母A-F

因此，如果你需要显示一个字符串变量，你可以用格式化指定符%s。如果你需要显示一个整数值，你可以用%d或%i (%d是C风格中用来显示十进制数的)。如果你要用科学计数法显示很大的值，你会用%e格式化指定符：

```
$ gawk 'BEGIN{
> x = 10 * 100
> printf "The answer is: %e\n", x
> }'
The answer is: 1.000000e+03
$
```

除了控制字母外，还有3种修饰符可以用来进一步控制输出。

- width：指定了输出字段最小宽度的数字值。如果输出短于这个值，printf会向右对齐，并用空格来填充这段空间。如果输出比指定的宽度还要长，它就会覆盖width值。
- prec：指定了浮点数中小数点后面位数的数字值，或者文本字符串中显示的最大字符数。
- -（减号）：减号指明在向格式化空间中放入数据时采用左对齐而不是右对齐。

在使用printf语句时，你对输出如何呈现有着完全的控制权。举个例子，在21.1.1节，我们用print命令来显示数据行中的数据字段：

```
$ gawk 'BEGIN{FS="\n"; RS=""} {print $1,$4}' data2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

你可以用printf命令来帮助格式化输出，使得输出看起来好一些。首先，让我们将print命令转换成printf命令并看看那么做会怎样：

```
$ gawk 'BEGIN{FS="\n"; RS=""} {printf "%s %s\n", $1, $4}' data 2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

它会产生跟print命令相同的输出。printf命令用%s格式化指定符来作为这两个字符串值的占位符。

注意你需要在printf命令的末尾手动添加换行符来生成新行。没加的话，printf命令会继续用同一行来打印后续输出。

如果你需要用几个单独的printf命令来在同一行上打印多个输出，它会非常有用：

```
$ gawk 'BEGIN{FS=".") {printf "%s . $1} END{printf "\n"}' data1
data11 data21 data31
$
```

每个printf的输出都会出现在同一行上。为了终止该行，END部分打印了一个换行符。

下一步，让我们用修饰符来格式化第一个字符串值：

```
$ gawk 'BEGIN{FS="\n"; RS=""} {printf "%16s %s\n", $1, $4}' data2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

通过添加一个值为16的修饰符，我们强制第一个字符串的输出采用16位字符。默认情况下，printf命令使用右对齐来将数据放到格式化空间中。要改成左对齐，只要给修饰符加一个减号就行了：

```
$ gawk 'BEGIN{FS="\n"; RS=""} {printf "%-16s %s\n", $1, $4}' data2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

现在看起来专业多了。

printf命令在处理浮点值时也非常有用。通过为变量指定一个格式，你可以让输出看起来更统一：

```
$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>   total += $i
> }
> avg = total / 3
> printf "Average: %.1f\n", avg
> }' data5
Average: 128.3
Average: 137.7
Average: 176.7
$
```

使用%5.1f格式指定符，你可以强制printf命令将浮点值近似到小数点后一位。

21.6 内建函数

gawk编程语言提供了一些内置函数，可进行一些常见的数学、字符串以及时间函数运算。你可以在gawk程序中利用这些函数来减少脚本中的编码工作。本节将会带你逐步熟悉gawk中这些不同的内建函数。

21.6.1 数学函数

如果你用任意类型的语言编过程，那么你可能会很熟悉在代码中使用内建函数来进行一些常见的数学函数运算。gawk编程语言不会让这些想借助高级数学功能降低编码量的人失望。

表21-4列出了gawk中内建的数学函数。

表21-4 gawk数学函数

函 数	描 述
atan2(x, y)	x/y的反正切，x和y以弧度为单位
cos(x)	x的余弦，x以弧度为单位
exp(x)	x的指数函数
int(x)	x的整数部分，取靠近零一侧的值
log(x)	x的自然对数
rand()	比0大比1小的随机浮点值
sin(x)	x的正弦，x以弧度为单位
sqrt(x)	x的平方根
srand(x)	为计算随机数指定一个种子值

虽然并未提供很多数学函数，但gawk提供了标准数学运算中要用到的一些基本元素。int()函数会生成一个值的整数部分，但它并不会四舍五入取近似值。它的做法更像其他编程语言中的floor函数。它会生成该值和0之间最接近该值的整数。

这意味着int()函数在值为5.6时返回5，而在值为-5.6时则返回-5。

rand()函数非常适合于创建随机数，但你需要用点技巧才能得到有意义的值。rand()函数会返回一个随机数，但这个随机数只在0和1之间（不包括0或1）。要得到更大的数，你就需要放大返回值。

产生较大整数随机数的常见方法是用rand()函数和int()函数创建一个算法：

```
x = int(10 * rand())
```

这会返回一个0~9（包括0和9）的随机整数值。只要为你的程序用上限值替换掉等式中的10就可以了。

在使用一些数学函数时要小心，因为gawk语言有个它能处理的数值的限定区间。如果超出了这个区间，就会得到一条错误消息：

```
$ gawk 'BEGIN{x=exp(100); print x}'  
26881171418161356094253400435962903554686976  
$ gawk 'BEGIN{x=exp(1000); print x}'  
gawk: warning: exp argument 1000 is out of range  
inf  
$
```

第一个例子会计算e的100次幂，虽然很大但尚在系统的区间内。第二个例子尝试计算e的1000次幂，它已经超出了系统的数值区间，所以生成了一条错误消息。

除了标准数学函数外，gawk还支持一些按位操作数据的函数。

- **and(v1, v2)**: 执行值v1和v2的按位与运算。
- **compl(val)**: 执行val的补运算。
- **lshift(val, count)**: 将值val左移count位。
- **or(v1, v2)**: 执行值v1和v2的按位或运算。
- **rshift(val, count)**: 将值val右移count位。
- **xor(v1, v2)**: 执行值v1和v2的按位异或运算。

位操作函数在处理数据中的二进制值时非常有用。

21.6.2 字符串函数

gawk编程语言还提供了一些可用来处理字符串值的函数，如表21-5所示。

表21-5 gawk字符串函数

函数	描述
asort(s [, d])	将数组s按数据元素值排序。索引值会被替换成表示新的排序顺序的连续数字。另外，如果指定了d，则排序后的数组会存储在数组d中
asorti(s [, d])	将数组s按索引值排序。生成的数组会将索引值作为数据元素值，用连续数字索引来表明排序顺序。另外如果指定了d，排序后的数组会存储在数组d中
gensub(r, s, h [, t])	查找变量r或目标字符串t（如果提供了的话）来匹配正则表达式r。如果h是一个以g或G开头的字符串，就用s替换掉匹配的文本。如果h是一个数字，它表示要替换掉第九处r匹配的地方
gsub(r, s [, t])	查找变量r或目标字符串t（如果提供了的话）来匹配正则表达式r。如果找到了，就全部替换掉字符串s
index(s, t)	返回字符串t在字符串s中的索引值；如果没有找到的话返回0
length([s])	返回字符串s的长度；如果没有指定的话，返回\$0的长度
match(s, r [, a])	返回字符串s中正则表达式r出现位置的索引。如果指定了数组a，它会存储s中匹配正则表达式的那部分
split(s, a [, r])	将s用FS字符或正则表达式r（如果指定了的话）分开放到数组a中。返回字段的总数

(续)

函 数	描 述
sprintf(format, variables)	用提供的format和variables返回一个类似于printf输出的字符串
sub(r, s [, t])	在变量\$0或目标字符串t中查找正则表达式r的匹配。如果找到了，就用字符串s替换掉第一处匹配
substr(s, i [, n])	返回s中从索引值i开始的n个字符组成的子字符串。如果未提供n，则返回s剩下的部分
tolower(s)	将s中的所有字符转换成小写
toupper(s)	将s中的所有字符转换成大写

一些字符串函数相对来说显而易见：

```
$ gawk 'BEGIN{x = "testing"; print toupper(x); print length(x) }'
TESTING
7
$
```

但一些字符串函数会相当复杂。asort和asorti函数是新加的gawk函数，允许你基于数据元素值(asort)或索引值(asorti)对数组变量进行排序。这里有个使用asort的例子：

```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> var["m"] = 3
> var["u"] = 4
> asort(var, test)
> for (i in test)
>     print "Index:"i," - value:",test[i]
> }'
Index: 4 - value: 4
Index: 1 - value: 1
Index: 2 - value: 2
Index: 3 - value: 3
$
```

新数组test含有排序后的原数组中的数据元素，但索引值现在变为表明正确顺序的数字值了。

split函数是将数据字段放到数组中以进一步处理的好办法：

```
$ gawk 'BEGIN{ FS=". "}{
> split($0, var)
> print var[1], var[5]
> }' data1
data11 data15
data21 data25
data31 data35
$
```

新数组使用连续数字作为数组索引，从含有第一个数据字段的索引值1开始。

21.6.3 时间函数

gawk编程语言包含一些函数来帮助处理时间值，如表21-6所示。

表21-6 gawk的时间函数

函 数	描 述
mktime(<i>datespec</i>)	将一个按YYYY MM DD HH MM SS [DST]格式指定的日期转换成时间戳 ^①
strftime(<i>format</i> [. <i>timestamp</i>])	将当前时间的时间戳或 <i>timestamp</i> （如果提供了的话）转化成用shell函数格式 <i>date()</i> 的格式化日期
systime()	返回当前时间的时间戳

时间函数通常用来处理日志文件，日志文件通常含有需要进行比较的日期。通过将日期的文本表示转换成epoch时间（自1970-01-01 00:00:00 UTC到现在的秒数），你可以轻松地比较日期。

下面是个在gawk程序中使用时间函数的例子：

```
$ gawk 'BEGIN{
> date = systime()
> day = strftime("%A, %B %d, %Y", date)
> print day
> }'
Friday, December 28, 2010
$
```

这个例子用systime函数来从系统获取当前的epoch时间戳，然后用strftime函数来将它转换成人类可读的格式，转换过程中使用了shell的date命令的日期格式化字符。

21.7 自定义函数

你并未被限定只能用gawk中的内建函数。你可以在gawk程序中创建自定义函数。本节将会介绍如何在gawk程序中定义和使用自定义函数。

21.7.1 定义函数

要定义自己的函数，你必须用function关键字：

```
function name([variables])
{
    statements
}
```

函数名必须能够唯一标识函数。你可以在调用的gawk程序中传给这个函数一个或多个变量：

```
function printthird()
```

^① 这里时间戳是指自1970-01-01 00:00:00 UTC到现在，以秒为单位的计数，通常称为epoch time。systime()函数的返回值也是这种形式。——译者注

```
{
    print $3
}
```

这个函数会打印数据行中的第3个数据字段。

函数还能用return语句返回值：

```
return value
```

值可以是变量，或者最终能计算出值的算式：

```
function myrand(limit)
{
    return int(limit * rand())
}
```

你可以将函数的返回值赋给gawk程序中的一个变量：

```
x = myrand(100)
```

这个变量最终会含有函数的返回值。

21.7.2 使用自定义函数

在定义函数时，它必须出现在所有代码块之前（包括BEGIN代码块）。乍一看这可能有点怪异，但它有助于将函数代码和gawk程序的其他部分分开：

```
$ gawk '
> function myprint()
> {
>     printf "%-16s - %s\n", $1, $4
> }
> BEGIN{FS="\n"; RS=""}
> {
>     myprint()
> }' data2
Riley Mullen      - (312)555-1234
Frank Williams   - (317)555-9876
Haley Snell       - (313)555-4938
$
```

这个函数定义了myprint()函数，它会格式化数据行中的第1个和第4个数据字段供打印用。然后，gawk程序用该函数显示了数据文件中的数据。

一旦定义了函数，你就能在程序的代码中随便使用了。在使用很长的算法时，这会节省许多工作。

21.7.3 创建函数库

显而易见地，每次使用时都重写一遍函数并不美妙。不过，gawk提供了一种途径来将函数放到一个库文件中，这样你就能在所有的gawk编程中使用了。

首先，你需要创建一个存储所有gawk函数的文件：

```
$ cat funclib
function myprint()
{
    printf "%-16s - %s\n", $1, $4
}
function myrand(limit)
{
    return int(limit * rand())
}
function printthird()
{
    print $3
}
$
```

funclib文件含有3个函数定义。要使用它们，你需要使用-f命令行参数。很遗憾，你不能将-f命令行参数和内联gawk脚本放到一起使用，不过你可以在同一个命令行中使用多个-f参数。

因此，要使用库，只要创建一个含有你的gawk程序的文件，然后在命令行上同时指定库文件和程序文件：

```
$ cat script4
BEGIN{ FS="\n"; RS="" }
{
    myprint()
}
$ gawk -f funclib -f script4 data2
Riley Mullen      - (312)555-1234
Frank Williams   - (317)555-9876
Haley Snell       - (313)555-4938
$
```

你要做的是当需要使用库中定义的函数时，将funclib文件加到你的gawk命令行上就可以了。

21.8 小结

本章带你逐步了解了gawk编程语言的高级特性。每种编程语言都要使用变量，gawk也不例外。gawk编程语言包含了一些内建变量，可以用来引用特定的数据字段值和获取数据文件中处理过的数据字段和数据行数信息。你也可以自定义一些变量来在脚本中使用。

gawk编程语言还提供了许多你期望编程语言有的标准结构化命令。你可以用if-then逻辑、while和do-while以及for循环轻松地创建强大的程序。每个命令都允许你改变gawk程序脚本的流来遍历数据字段的值、创建详细的数据报告。

如果要定制报告的输出，printf命令会是一个强大的工具。它允许指定具体的格式来显示gawk程序脚本的数据。你可以轻松地创建格式化报告，将数据元素放到正确的位置。

最后，本章讨论了gawk编程语言的许多内建函数并介绍了如何创建自定义函数。gawk程序含有许多有用的函数来处理数学功能（比如标准的平方根运算和对数运算，以及三角函数）。还有若干字符串相关的函数，它们使得从较大字符串中提取子字符串成为轻而易举的事。

你并不局限于gawk程序的内建函数。如果你正在写一个要用到大量特定算法的应用程序，你可以创建自定义函数来处理这些算法，然后在代码中使用这些函数。你也可以创建一个含有所有你要在gawk程序中用到的函数的库文件，以节省时间和精力。

下一章会稍微换个方向。它会介绍你可能会遇到的其他一些shell环境。虽然bash shell是Linux中最常用的shell，但它不是唯一的shell。了解一点其他shell和它们跟bash shell的区别是有好处的。



本章内容

- 理解dash shell
- dash shell脚本编程
- zsh shell介绍
- zsh脚本编程

虽然bash shell是Linux发行版中最广泛使用的shell，但它不是唯一的。现在你已经了解了标准的Linux bash shell，以及你能用它做什么，是时候了解一下Linux世界中的其他一些shell了。本章将会介绍另外两个你可能会碰到的shell，以及它们跟bash shell有什么区别。

22.1 什么是 dash shell

Debian的dash shell的历史很有趣。它是ash shell的直系后代，而ash shell是Unix系统上原来的Bourne shell的简化版本（见第1章）。Kenneth Almquist为Unix系统开发了一个Bourne shell的简化版本，并将它命名为Almquist shell，缩写为ash。ash shell最早的版本极其小和快，但没有许多高级功能，比如命令行编辑或命令使用记录功能，使它很难用作交互式shell。

NetBSD Unix操作系统移植了ash shell，而且今天依然将它用作默认shell。NetBSD开发人员对ash shell进行了定制，增加了一些新的功能，使它更接近Bourne shell。新功能包括使用emacs和vi编辑器命令进行命令行编辑，以及历史命令来查看前面输入的命令。ash shell的这个版本也被FreeBSD操作系统用作默认登录shell。

Debian Linux发行版创建了它自己的ash shell版本（称作Debian ash，或dash）来将其放进Debian Linux中。多数时候，dash复制了ash shell的NetBSD版本的功能，提供了一些高级命令行编辑能力。

但令人不解的是，实际上dash shell在许多基于Debian的Linux发行版中并不是默认的shell。由于bash shell在Linux中的流行，大多数基于Debian的Linux发行版将bash shell用作普通登录shell，而只将dash shell用作安装脚本的快速启动shell来安装发行版文件。

流行的Ubuntu发行版是个例外。这通常会让shell脚本程序员很困惑，并给Linux环境中运行shell脚本带来了很多问题。Ubuntu Linux发行版将bash shell用作默认的交互shell，但将dash shell

用作默认的/bin/sh shell。这个“特性”叫shell脚本程序员一头雾水。

如你在第10章中看到的，每个shell脚本都必须以声明所用的shell的行开头。在bash shell脚本中，我们一直用下面的行：

```
#!/bin/bash
```

它会告诉shell使用位于/bin/bash的shell程序来执行脚本。在Unix世界中，默认shell一直是/bin/sh。许多熟悉Unix环境的shell脚本程序员会将这种用法带到他们的Linux shell脚本中：

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

在大多数Linux发行版上，/bin/sh文件是链接到/bin/bash shell程序的一个符号链接（参见第3章）。它允许你简单地将为Unix Bourne shell设计的shell脚本移植到Linux环境中，而不用修改。

很遗憾，Ubuntu Linux发行版将/bin/sh文件链接到了/bin/dash shell程序。由于dash shell只含有原来Bourne shell中的一部分命令，这可能会（而且经常会）让有些shell脚本无法正确工作。

下一节将带你逐步了解dash shell的基础知识以及它跟bash shell的区别。如果你编写了可能要在Ubuntu环境中运行的bash shell脚本，了解这些尤其重要。

22.2 dash shell 的特性

尽管bash shell和dash shell都以Bourne shell为模型，但它们还是有一些差别的。在我们深入了解shell脚本编程特性之前，本节将会带你了解Debian dash shell的一些特性来让你熟悉dash shell如何是工作的。

22.2.1 dash命令行参数

dash shell使用命令行参数来控制它的行为。表22-1列出了命令行参数并介绍了每个参数是做什么的。

表22-1 dash命令行参数

参数	描述
-a	导出分配给shell的所有变量
-c	从特定命令字符串读取命令
-e	如果是非交互式shell的话，在有未经测试的命令失败时立即退出
-f	显示路径名通配符
-n	如果是非交互式shell的话，读取命令但不执行它们
-u	在尝试展开一个未设置的变量时，将错误消息写出到STDERR
-v	在读取输入时将输入写出到STDERR
-x	在执行命令时将每个命令写出到STDERR
-I	在交互式模式下，忽略输入中的EOF字符
-i	强制shell运行在交互式模式下

(续)

参数	描述
-m	打开作业控制（在交互式模式下默认开启）
-s	从STDIN读取命令（在没有指定文件参数时的默认行为）
-E	打开emacs命令行编辑器
-V	打开vi命令行编辑器

有一些额外的命令行参数是Debian加到原来ash shell的命令行参数列表中的。-E和-V命令行参数会打开dash shell特有的命令行编辑功能。

-E命令行参数允许使用emacs编辑器命令来编辑命令行的文本（参见第9章）。你可以使用所有的emacs命令来处理一行中的文本，其中会用到Ctrl和Meta组合键。

-V命令行参数允许使用vi编辑器命令来编辑命令行文本（参见第9章）。这个功能允许用Esc键在普通模式和vi编辑器模式之间切换。当你在vi模式中时，你可以用标准的vi编辑器命令（例如x删除一个字符，i插入文本）。完成命令行编辑后，必须再次按下Esc键退出vi编辑器模式。

22.2.2 dash环境变量

dash shell用相当多的默认环境变量来记录信息，你也可能创建自己的环境变量。本节将会介绍环境变量以及dash如何处理它们。

1. 默认环境变量

表22-2列出了默认的dash环境变量并描述了它们的用途。

表22-2 dash shell环境变量

变量	描述
CDPATH	cd命令的搜索路径
HISTSIZE	历史记录文件中保存的行数
HOME	用户的默认登录目录
IFS	输入字段分隔符。默认值是空格、制表符和换行符
MAIL	用户收件箱文件的名称
MAILCHECK	在收件箱文件中检查新邮件的频率
MAILPATH	冒号分割的多个收件箱文件名称。设置了的话，这个值会覆盖MAIL环境变量
OLDPWD	上一个工作目录的值
PATH	可执行文件的默认查找路径
PPIO	当前shell的父进程的进程ID
PS1	shell的主命令行交互提示符

(续)

变 量	描 述
PS2	shell的次命令行交互提示符
PS4	当使能了执行追踪时，在每行前面打印的一个字符
PWD	当前工作目录的值
TERM	shell的默认终端设置

要注意，dash环境变量跟bash环境变量很像（参见第5章）。这绝非偶然。记住dash和bash shell都是Bourne shell的延伸，所以它们都吸收了很多Bourne shell的特性。不过，由于dash的目标是简洁，dash shell比bash shell的环境变量少多了。在dash shell环境中编写脚本时，你要记住这点。

dash shell用set命令来显示环境变量：

```
$set
COLORTERM=''
DESKTOP_SESSION='default'
DISPLAY=':0.0'
DM_CONTROL='/var/run/xdmctl'
GS_LIB='/home/atest/.fonts'
HOME='/home/atest'
IFS=''
.
KDEROOTHOME='/root/.kde'
KDE_FULL_SESSION='true'
KDE_MULTIHEAD='false'
KONSOLE_DCOP='DCOPRef(konsole-5293,konsole)'
KONSOLE_DCOP_SESSION='DCOPRef(konsole-5293.session-1)'
LANG='en_US'
LANGUAGE='en'
LC_ALL='en_US'
LOGNAME='atest'
OPTIND='1'
PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
PPID='5293'
PS1='${ '
PS2='-> '
PS4='+' '
PWD='/home/atest'
SESSION_MANAGER='local/testbox:/tmp/.ICE-unix/5051'
SHELL='/bin/dash'
SHLVL='1'
TERM='xterm'
USER='atest'
XCURSOR_THEME='default'
_=ash'
$
```

你的默认dash shell环境很可能与之略有不同，因为不同的Linux发行版在登录时分配了不同的默认环境变量。

2. 位置参数

除了默认环境变量，dash shell还给命令行上定义的参数分配了特殊变量。下面是dash shell中用的位置参数变量。

- \$0: shell的名称。
- \$n: 第n个位置参数。
- \$*: 含有所有参数内容的单个值，由IFS环境变量中的第一个字符分隔；没定义IFS的话由空格分隔。
- \$@: 将所有的命令行参数展开为多个参数。
- \$#: 位置参数的总数。
- \$?：最近一个命令的退出状态码。
- \$-：当前选项标记。
- \$\$: 当前shell的进程ID (PID)。
- \$\$!: 最近一个后台命令的进程ID (PID)。

所有dash位置参数都类似于bash shell中的位置参数。你可以在shell脚本中使用位置参数，就像bash shell中的用法一样。

3. 用户自定义的环境变量

dash shell还允许你定义自己的环境变量。跟bash一样，你可以在命令行上用赋值语句来定义新的环境变量：

```
$ testing=10
$ echo $testing
10
$
```

默认情况下，环境变量只在定义它们的shell会话中可见。要让环境变量在子shell或子进程中可见，必须用export命令：

```
$ testing=10 ; export testing
$ dash
$ echo $testing
10
$
```

如果不用export命令，用户自定义的环境变量就只在当前shell或进程中可见。

警告 dash变量和bash变量之间有个巨大的差异。dash shell不支持可变数组。这个小特性给高级shell脚本开发人员带来了各种问题。

22.2.3 dash内建命令

跟bash shell一样，dash shell含有一组它能识别的内建命令。你可以在命令行界面上直接使用这些命令，或者将它们放到shell脚本中。表22-3列出了dash shell的内建命令。

表22-3 dash shell内建命令

命 令	描 述
alias	创建代表文本字符串的别名字符串
bg	以后台模式继续指定的作业
cd	切换到指定的目录
echo	显示文本字符串和环境变量
eval	将所有参数用空格连起来 ^①
exec	用指定命令代替shell进程
exit	终止shell进程
export	导出指定的环境变量，供子shell使用
fc	列出、编辑或重新执行之前在命令行中输入的命令
fg	以前台模式继续指定的作业
getopts	从一列参数中提取选项和参数
hash	维护并提取最近执行的命令和它们位置的哈希表
pwd	显示当前工作目录的值
read	从STDIN读取一行并将其赋给一个变量
readonly	从STDIN读取一行并赋给一个不能修改的变量
printf	用格式化过的字符串显示文本和变量
set	列出或设置选项标记和环境变量
shift	按指定的次数移动位置参数
test	测试一个表达式，成立的话返回0，不成立的话返回1
times	显示当前shell和所有shell进程的累计用户和系统时间
trap	在shell收到某个指定信号时解析并执行命令
type	解释指定的名称并显示解析结果（别名、内建、命令或关键字）
ulimit	查询或设置进程限制
umask	设置默认文件值和目录权限
unalias	删除指定的别名
unset	从导出的变量中删除指定的变量或选项标记
wait	等待指定的命令完成然后返回退出状态码

你可能在bash shell中认识了上面的所有内建命令。dash shell支持许多和bash shell一样的内建命令。你会注意到其中没有操作命令历史记录或目录栈的命令。dash shell不支持这些特性。

22.3 dash 脚本编程

很遗憾，dash shell不能识别bash shell的所有脚本编程功能。为bash环境编写的脚本通常在dash

① 这条命令的重点在于将所有参数用空格连接起来后，它会重新解析并执行这条命令。——译者注

shell中会运行失败，这给shell脚本程序员带来了各种痛苦。本节将介绍你应该了解的一些差别，这样你才能让shell脚本在dash shell环境中正常运行。

22.3.1 创建dash脚本

到目前为止，你可能已经猜到，为dash shell编写脚本和为bash shell编写脚本非常类似。你要在脚本中一直指定要用哪个shell，保证脚本是用正确的shell运行的。

可以在shell脚本的第一行指定：

```
#!/bin/dash
```

还可以在这行指定shell命令行参数，22.2.1节中介绍了这些参数。

22.3.2 不能使用的功能

很遗憾，由于dash shell只是Bourne shell功能的一个子集，bash shell脚本中有些功能没法在dash shell中工作。这些通常称作bash主义（bashism）。本节是对你在bash shell脚本中习惯使用、但在dash shell环境中没法工作的bash shell功能的一个简单总结。

1. 使用算术运算

第10章介绍了3种在bash shell脚本中表达数学运算的方法。

- 使用expr命令：expr operation。
- 使用方括号：\$[operation]。
- 使用双圆括号：\$((operation))。

dash shell支持expr命令和双圆括号方法，但不支持方括号方法。如果你有许多数学运算采用方括号的话，这可能会是个问题：

```
$ cat test5
#!/bin/dash
# testing mathematical operations

value1=10
value2=15

value3=$[ $value1 * $value2 ]
echo "The answer is $value3"
$ ./test5
./test5: 7: value1: not found
The answer is
$
```

在dash shell脚本中执行算术运算的正确格式是用双圆括号方法：

```
$ cat test5b
#!/bin/dash
# testing mathematical operations

value1=10
```

```

value2=15

value3=$(( $value1 * $value2 ))
echo "The answer is $value3"
$ ./test5b
The answer is 150
$
```

现在shell可以正确运行这个计算了。

2. test命令

虽然dash shell支持test命令，但你必须注意如何使用。bash shell版本的test命令跟dash shell版本的略有不同。

bash shell的test命令允许你使用双等号(==)来测试两个字符串是否相等。这是为了照顾习惯在其他编程语言中使用这种格式的程序员而加上去的：

```

$ cat test6
#!/bin/bash
# testing the == comparison

test1=abcdef
test2=abcdef

if [ $test1 == $test2 ]
then
    echo "They're the same!"
else
    echo "They're different"
fi
$ ./test6
They're the same!
$
```

非常简单。但你要在dash shell环境中运行这个脚本的话，你会得到一个不想要的输出：

```

$ ./test6
[: ==: unexpected operator
They're different
$
```

dash shell中的test命令不能识别用作文本比较的==符号，它只能识别=符号。如果你将文本比较符号改成单等号，那么一切就能同时在bash shell环境和dash shell环境中工作起来了：

```

$ cat test7
#!/bin/dash
# testing the = comparison

test1=abcdef
test2=abcdef

if [ $test1 = $test2 ]
then
    echo "They're the same!"
```

```

else
    echo "They're different"
fi
$ ./test7
They're the same!
$
```

这么一点点bash主义就足以让shell程序员折腾几个小时了。

3. echo语句选项

对dash shell程序员来说，简单的echo语句也会带来很多麻烦。在dash和bash shell中，它的行为并不一样。

在bash shell中，如果要在输出中显示一个特殊字符，你必须用-e命令行参数：

```
echo -e "This line contains\t a special character"
```

echo语句含有代表制表符的\t特殊字符。不用-e命令行参数的话，bash版本的echo语句会忽略这个特殊字符。下面是bash shell中的一个测试：

```

$ cat test8
#!/bin/bash
# testing echo commands

echo "This is a normal test"
echo "This test uses \t special character"
echo -e "This test uses \t special character"
echo -n "Does this work: "
read test
echo "This is the end of the test"
$ ./test8
This is a normal test
This test uses \t special character
This test uses a special character
Does this work: N
This is the end of the test
$
```

没用-e命令行参数的echo语句只将\t字符作为普通字符显示了，所以要输出制表符，必须用-e命令行参数。

在dash shell中，情况有点不同。dash shell中的echo语句会自动识别并显示特殊字符。正因为如此，dash shell没有-e命令行参数。如果在dash环境中运行同一个脚本，你会得到下面的输出：

```

$ ./test8
This is a simple test
This line uses a      special character
-e This line uses a      special character
Does this work: N
This is the end of the test
$
```

如你在输出中所看到的，dash shell版本的echo命令识别了该行中的特殊字符。没用-e命令行参数的那行刚好能正常工作，但对于用了-e命令行参数的那行，echo命令却将-e当成普通文本显

示出来了。

很遗憾，这个问题没有简单的解决办法。如果你写的脚本必须同时能在bash和dash shell环境中工作，最好的解决办法是用printf命令来显示文本。这个命令在两种shell环境中完全一样，它也能正常显示特殊字符。

4. function命令

第16章演示了如何在shell脚本中定义自己的函数。bash shell支持两种定义函数的方法。第一种方法是用function语句：

```
function name {
    commands
}
```

name属性定义了分配给函数的唯一名称。必须为在脚本中定义的每个函数都分配一个唯一的名称。

commands是构成函数的一条或多条bash shell命令。当你调用这个函数时，bash shell会按它们在函数中出现的顺序执行每一条命令，就跟在普通脚本中一样。

在bash shell脚本中定义函数的第二种格式更接近于其他编程语言中定义函数的方式：

```
name() {
    commands
}
```

函数名后空的圆括号表明你定义了一个函数。这种形式采用跟原来shell脚本的函数格式一样的命名规则。

dash shell不支持第一种定义函数的方法（它不支持function语句）。在dash shell中你必须用函数名和圆括号定义函数。如果你要在dash shell中运行一个为bash shell定义的函数，你会得到一条错误消息：

```
$ cat test9
#!/bin/dash
# testing functions

function func1() {
    echo "This is an example of a function"
}
count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func1
echo "This is the end of the script"
$ ./test9
./test9: 4: Syntax error: "(" unexpected
$
```

dash shell不会将函数代码赋给函数，而是执行函数定义中的代码，然后提示shell脚本的格式

有错。

如果你在编写能在dash环境中运行的shell脚本，你必须用第二种定义函数的方法：

```
$ cat test10
#!/bin/dash
# testing functions

func1() {
    echo "This is an example of a function"
}

count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func1
echo "This is the end of the script"
$ ./test10
This is an example of a function
This is the end of the loop
This is an example of a function
This is the end of the script
$
```

现在dash shell能够识别脚本中定义的函数并能在脚本中使用它了。

22.4 zsh shell

你可能会碰到的另一个流行的shell是Z shell（称作zsh）。zsh shell是由Paul Falstad开发的一个开源Unix shell。它集成了所有现有shell的思想并增加了许多独到的功能，为程序员创建了一个全功能的高级shell。

下面是zsh shell的一些独特的功能：

- 改进的shell选项处理；
- shell兼容性模式；
- 可加载模块。

在所有这些功能中，可加载模块是shell设计中最先进的功能。如你在bash和dash shell中看到的，每种shell都包含一组可用的内建命令，而不用求助于外部工具程序。内建命令的好处在于执行速度快。shell不必在运行命令前先加载一个工具程序。内建命令已经在shell内存中了，随时可用。

zsh shell提供了一组核心内建命令，并提供了添加额外命令模块（command module）的能力。

每个命令模块都为特定环境提供了一组额外的内建命令，比如网络支持和高级数学功能。你可以只添加你觉得有用的模块。

这个功能提供了一个极佳的方式，在需要较小shell体积和较少命令时限制zsh shell的体积，或者在需要更快执行速度时增加可用的内建命令数。

22.5 zsh shell的组成

本节将带你逐步了解zsh shell的基础知识，介绍可用的内建命令（或可以通过安装模块添加的命令）以及命令行参数和环境变量。

22.5.1 shell选项

大多数shell采用命令行参数来定义shell的行为。zsh shell使用一些命令行参数来定义shell的操作，但大多数情况下它用选项来定制shell的行为。你可以在命令行上或在shell中用set命令设置shell选项。表22-4列出了zsh shell可用的命令行参数。

表22-4 zsh shell命令行参数

参 数	描 述
-c	只执行指定的命令，然后退出
-i	作为交互式shell启动，会提供一个命令行交互提示符
-s	强制shell从STDIN读取命令
-o	指定命令行选项

虽然这看起来像是一小组命令行参数，但-o参数有些容易让人产生误解。它允许你设置shell选项来定义shell的功能。到目前为止，zsh shell是所有shell中可定制性最强的。有很多功能可以用来在shell环境中替换。不同的选项可以分成以下几大类。

- **更改目录：**控制cd和dirs命令如何处理目录更改的选项。
- **补全：**控制命令补全功能的选项。
- **扩展和扩展匹配：**控制命令中文件扩展的选项。
- **历史记录：**控制命令历史记录的选项。
- **初始化：**控制shell在启动时如何处理变量和启动文件的选项。
- **输入输出：**控制命令处理的选项。
- **作业控制：**控制shell如何处理作业和启动作业的选项。
- **提示：**控制shell如何处理命令行提示符的选项。
- **脚本和函数：**控制shell如何处理shell脚本和定义函数的选项。
- **shell模拟：**允许设置zsh shell来模拟其他类型shell行为的选项。
- **shell状态：**定义启动哪种shell的选项。
- **zle：**控制zsh行编辑器功能的选项。

- 选项别名：可以用作其他选项别名的特殊选项。

有这么多种不同的shell选项，你可以想象一下zsh shell支持多少实际的选项。后面几节会抽样介绍定制zsh shell环境时可用的不同选项。

1. shell状态选项

有6种不同的zsh shell选项来定义shell启动的类型。

- 交互模式 (-i, interactive)：提供了命令行界面提示符来输入内建命令和程序名。
- 登录模式 (-l, login)：默认的zsh shell类型，处理zsh shell的启动文件并提供命令行界面提示符。
- 特权模式 (-p, privileged)：有效的用户ID (EUID) 跟实际用户ID不一致 (用户成为了root用户) 时的默认类型。它会禁止用户启动文件。
- 限制模式 (-r, restricted)：在shell中将用户限定在特定目录结构中。
- shin_stdin模式 (-s)：从STDIN读取命令。
- single_command模式 (-t)：执行一条从STDIN读取的命令，然后退出。

shell状态定义了shell是否在启动时提供命令行界面提示符，以及用户在shell中有什么访问权限。

2. shell模拟选项

shell模拟选项允许定制zsh shell以提供类似于程序员中流行的C shell (csh) 或Korn shell (ksh) 的运行。这些选项如下。

- bsd_echo：让echo语句跟C shell的echo命令兼容。
- csh_junkie_history：用不带指定符的history命令来引用前面的命令。
- csh_junkie_loops：允许while和for循环使用类似于C shell的end，而不是do和done。
- csh_junkie_quotes：修改使用单引号和双引号的规则来跟C shell保持一致。
- csh_nulcmd：在执行没有命令的重定向时，不使用NULLCMD和READNULLMD变量的值。
- ksh_array：使用Korn风格的数组，采用从0开始的数字索引值，并在引用数组元素时使用方括号。
- ksh_autoload：模拟Korn shell的自动加载函数功能。
- ksh_option_print：模拟Korn shell打印选项的方法。
- ksh_typeset：替换处理typeset命令参数的方式。
- posix_builtins：使用builtin命令来执行内建命令。
- sh_file_expansion：在执行其他展开之前先进行文件名展开。
- sh_nulcmd：在进行重定向时不使用NULLCMD和READNULLCMD变量。
- sh_option_letters：用类似于Korn shell的方式解释单字母命令行选项。
- sh_word_split：在未加引号的参数展开中执行字段分割。
- traps_async：在等待程序退出时，处理信号并立即运行捕捉。

有了这么多选项，你可以选择使用你要在zsh shell中模拟的csh或ksh shell功能，而不用模拟整个shell。

3. 初始化选项

有一些选项可以用来处理shell启动功能。

- all_export**: 所有的参数和变量会自动导出到子shell进程中。
- global_export**: 导出到环境中的参数不会在函数中本地化。
- global_rcs**: 如果没有设置, zsh shell不会运行全局启动文件, 但仍然会运行本地启动文件。
- rcts**: 如果没有设置, zsh shell会运行/etc/zshenv启动文件, 但不会运行其他文件。

初始化选项允许指定在shell环境中运行哪些zsh shell启动文件(如果有的话)。你也可以在启动文件中设置这些值来限定shell执行哪些选项。

4. 脚本和函数选项

脚本和函数选项允许你在zsh shell中定制shell脚本环境。这是设置函数在shell中运行方式的简便途径。

- c_bases**: 用C格式(0xdddd)显示十六进制数而不是用shell格式(16#ddd)。
- err_exit**: 如果命令以非零退出状态码退出, 执行ZERR捕捉中的命令并退出。
- err_return**: 如果命令以非零退出状态码退出, 立即从其所在函数返回。
- eval_lineno**: 如果设置了, 用eval内建命令评估的表达式的行号会和shell环境中的其余部分分开记录。
- exec**: 执行命令。如果未设置这个选项, 会读取命令并报告错误, 但不会执行命令。
- function_argzero**: 将\$0设置成函数名或脚本名。
- local_options**: 设置了的话, 当shell函数返回时, 恢复所有在该函数之前设置的选项。
- local_traps**: 设置了的话, 当在函数内设置了信号捕捉, 函数退出时恢复前一个捕捉的状态。
- multios**: 在尝试执行多个重定向时, 执行隐式tee或cat命令。
- octal_zeros**: 将任何以0开头的整数字符串都解释成八进制数。
- typeset_silent**: 未设置的话, 使用typeset和参数名来显示参数的当前值。
- verbose**: 在shell读取输入行时显示它们。
- xtrace**: 在shell执行命令时显示命令和命令的参数。

zsh shell允许定制退出shell中定义的函数时的许多功能。

22.5.2 内建命令

zsh shell的独到之处在于它允许扩展shell中的内建命令。它为许多不同的应用程序提供了大量的快速工具。

本节将会介绍核心内建命令以及写作本书时已有的各种模块。

1. 核心内建命令

zsh shell的核心包括一些基本的内建命令, 跟你在其他shell中常见的差不多。表22-5列出

了可用的内建命令。

表22-5 zsh核心内建命令

命 令	描 述
alias	为命令和参数定义一个替代性名称
autoload	将shell函数预加载到内存中以便快速访问
bg	以后台模式执行一个作业
bindkey	将组合键和命令绑定到一起
builtin	执行指定的内建命令而不是同样名称的可执行文件
bye	跟exit相同
cd	切换当前工作目录
chdir	切换当前工作目录
command	将指定命令当做外部文件执行而不是函数或内建命令
declare	设置变量的数据类型(同typeset)
dirs	显示目录栈的内容
disable	临时禁用指定的哈希表元素
disown	从作业表中移除指定的作业
echo	显示变量和文本
emulate	用zsh来模拟另一个shell,比如Bourne、Korn或C shell
enable	使能指定的哈希表元素
eval	在当前shell进程中执行指定的命令和参数
exec	执行指定的命令和参数来替换当前shell进程
exit	退出shell并返回指定的退出状态码。如果没有指定,使用最后一条命令的退出状态码
export	允许在子shell进程中使用指定的环境变量名及其值
false	返回退出状态码1
fc	从历史记录中选择某范围内的命令
fg	以前台模式执行指定的作业
float	将指定变量设为保存浮点值的变量
functions	将指定名称设为函数
getln	从缓冲栈中读取下一个值并将其放到指定变量中
getopts	提取命令行参数中的下一个有效选项并将它放到指定变量中
hash	直接修改命令哈希表的内容
history	列出历史记录文件中的命令
integer	将指定变量设为整数类型
jobs	列出指定作业的信息,或分配给shell进程的所有作业
kill	向指定进程或作业发送信号(默认为SIGTERM)

(续)

命 令	描 述
let	执行算术运算并将结果赋给一个变量
limit	设置或显示资源限制
local	为指定变量设置数据属性
log	显示受watch参数 ^① 影响的当前登录到系统上的所有用户
logout	同exit，但只在shell是登录shell时有效
popd	从目录栈中删除下一项
print	显示变量和文本
printf	用C风格的格式字符串来显示变量和文本
pushd	改变当前工作目录，并将上一个目录放到目录栈中
pushln	将指定参数放到编辑缓冲栈中
pwd	显示当前工作目录的完整路径名
read	读取一行并用IFS变量将数据字段赋给指定变量
readonly	将值赋给不能修改的变量
rehash	重建命令哈希表
set	为shell设置选项或位置参数
setopt	为shell设置选项
shift	读取并删除第一个位置参数，然后将剩余的参数向前移动一个位置
source	找到指定文件并将其内容复制到当前位置
suspend	挂起shell的执行，直到它收到SIGCONT信号
test	如果指定条件为TRUE的话，返回退出状态码0
times	显示当前shell以及shell中所有运行进程的累计用户时间和系统时间
trap	阻断指定信号从而让shell无法处理，如果收到信号则执行指定命令
true	返回退出状态码0
ttyctl	锁定和解锁显示
type	显示shell会如何解释指定的命令
typeset	设置或显示变量的特性
ulimit	设置或显示shell或shell中运行进程的资源限制
umask	设置或显示创建文件和目录的默认权限
unalias	删除指定的命令别名
unfunction	删除指定的已定义函数
unhash	删除哈希表中的指定命令

① zsh提供了一种途径来监测和报告指定用户的登录情况，通过设置watch参数来指定要监测的用户、远程登录系统的主机和虚拟终端。——译者注

(续)

命 令	描 述
unlimit	删除指定的资源限制
unset	删除指定的变量特性
unsetopt	删除指定的shell选项
wait	等待指定的作业或进程完成
whence	显示指定命令会如何被shell解释
where	显示指定命令的路径名, 如果shell找到的话
which	用csh风格的输出显示指定命令的路径名
zcompile	编辑指定的函数或脚本从而能更快地自动加载
zmodload	对可加载zsh模块执行特定操作

zsh shell在提供内建命令方面太强大了。你可以从bash中对应的命令来识别其中的大多数命令。zsh shell内建命令最重要的功能是模块。

2. 附加模块

有一长列模块可以为zsh shell提供额外的内建命令, 而且这个列表会随着程序员不断增加新模块而继续增长。表22-6列出了在写作本书时已有的模块。

表22-6 zsh模块

模 块	描 述
zsh/cap	POSIX兼容性命令
zsh/clone	将运行中的shell克隆到另一个终端的命令
zsh/comctl	控制命令补全的命令
zsh/complete	命令行补全命令
zsh/complist	命令行补全列表扩展命令
zsh/comutil	命令行补全的实用工具命令
zsh/datetime	额外的日期和时间命令及变量
zsh/deltochar	重现了emacs功能的行编辑函数
zsh/files	基本的文件处理命令
zsh/mapfile	通过关联数组来访问外部文件
zsh/mathfunc	额外的科学函数
zsh/parameter	通过关联数组来访问命令哈希表
zsh/pcre	扩展的正则表达式库
zsh/sched	按设定时间执行命令的计划命令
zsh/net/socket	Unix域套接字支持
zsh/stat	访问stat系统调用来提供系统的统计状况

(续)

模 块	描 述
zsh/system	访问各种底层系统功能的接口
zsh/net/tcp	访问TCP套接字
zsh/termcap	termcap数据库的接口
zsh/terminfo	terminfo数据库的接口
zsh/zftp	专用FTP客户端命令
zsh/zle	zsh行编辑器
zsh/zleparameter	用变量访问并修改zle
zsh/zprof	允许对shell函数进行性能参数统计
zsh/zpty	在虚拟终端中执行一条命令
zsh/zselect	阻断，直到文件描述符就绪才返回
zsh/zutil	各种shell实用工具

zsh shell模块涵盖了很多方面的功能，从简单的命令行编辑功能到高级网络功能。zsh shell的思想是提供基本最小的shell环境，让你在编程时再添加需要的模块。

3. 查看、添加和删除模块

zmodload命令是zsh模块的管理接口。你可以用这个命令来从zsh shell会话中查看、添加或删除模块。

zmodload命令不加任何参数会显示zsh shell中当前已安装的模块：

```
% zmodload
zsh/zutil
zsh/complete
zsh/main
zsh/terminfo
zsh/zle
zsh/parameter
%
```

不同的zsh shell实现在默认情况下包含了不同的模块。要添加新模块，在zmodload命令行上指定模块名称就行了：

```
* zmodload zsh/zftp
*
```

不会有信息表明模块已经加载成功了。你可以再运行一下zmodload命令，新添加的模块会出现在已安装模块的列表中。

一旦你加载了模块，该模块中的命令就成为可用的内建命令了：

```
% zftp open myhost.com rich testing1
Welcome to the myhost FTP server.
% zftp cd test
% zftp dir
```

```
01-21-11 11:21PM  120823 test1
01-21-11 11:23PM  118432 test2
% zftp get test1 > test1.txt
% zftp close
%
```

zftp命令允许你在zsh shell命令行操作完整的FTP会话。你可以在zsh shell脚本中使用这些命令，来直接在脚本中进行文件传输。

要删除已安装的模块，用-u参数和模块名：

```
% zmodload -u zsh/zftp
% zftp
zsh: command not found: zftp
%
```

说明 通常会将zmodload命令放进\$HOME/.zshrc启动文件中，这样在zsh启动时常用的函数就会自动加载。

22.6 zsh 脚本编程

zsh shell的主要目的是为shell程序员提供一个高级编程环境。认识到这点，你就能理解为什么zsh shell会提供那么多方便脚本编程的功能了。

22.6.1 数学运算

如你所期望的，zsh shell允许你方便地执行数学函数。过去，Korn shell因为支持浮点数一直在数学运算支持方面领先。zsh shell在所有数学运算中都提供了对浮点数的全面支持。

1. 进行计算

zsh shell提供了执行数学运算的两种方法：

- let命令；
- 双圆括号。

在使用let命令时，你应该在算式前后加上双引号，这样才能支持空格：

```
% let value1=" 4 * 5.1 / 3.2 "
% echo $value1
6.37499999999991
%
```

注意，使用浮点数会带来精度问题。要解决这个问题，通常使用printf命令，并指定能正确显示结果的小数点精度：

```
% printf "%6.3f\n" $value1
6.375
%
```

现在好多了！

第二种方法是使用双圆括号。这个方法合并了两种定义数学运算的方法：

```
% value1=$(( 4 * 5.1 ))
% (( value2 = 4 * 5.1 ))
% printf "%6.3f\n" $value1 $value2
20.400
20.400
%
```

注意，你可以将双圆括号放在算式两边（前面加个美元符）或整个赋值表达式两边。两种方法输出了同样的结果。

如果一开始你没用typeset命令来声明变量的数据类型，那么zsh shell会尝试自动分配数据类型。这在处理整数和浮点数时很危险。看看下面这个例子：

```
% value=10
% value2=$(( $value / 3 ))
% echo $value2
3
%
```

现在这个结果可能并不是你所期望的。在指定数字时没指定小数点后的位数的话，zsh shell会将它们都当成整数值并进行整数运算。要保证结果是浮点数，你必须指定该数小数点后的位数：

```
% value=10.
% value2=$(( $value / 3. ))
% echo $value2
3.333333333333335
%
```

现在结果是浮点数形式了。

2. 数学函数

zsh shell中，内建数学函数可多可少。默认的zsh并不含任何特殊的数学函数。但如果安装了zsh/mathfunc模块，你会拥有远远超出你可能用到的数学函数：

```
% value1=$(( sqrt(9) ))
zsh: unknown function: sqrt
% zmodload zsh/mathfunc
% value1=$(( sqrt(9) ))
% echo $value1
3.
%
```

非常简单。现在你就有了整个数学函数库了。

说明 zsh中支持很多数学函数。要查看zsh/mathfunc模块提供的所有数学函数的清单，可以参看zshmodules手册页面。

22.6.2 结构化命令

zsh shell为shell脚本提供了常用的结构化命令：

- if-then-else语句；
- for循环（包括C语言风格的）；
- while循环；
- until循环；
- select语句；
- case语句。

zsh中的每个结构化命令采用的语法跟你熟悉的bash shell中的一样。zsh shell还包含了另外一个叫做repeat的结构化命令。repeat命令使用如下格式：

```
repeat param
do
    commands
done
```

param参数必须是一个数字或能算出一个值的数学算式。然后repeat命令会执行指定命令这么多次数：

```
% cat test1
#!/bin/zsh
# using the repeat command

value=$(( 10 / 2 ))
repeat $value
do
    echo "This is a test"
done
$ ./test1
This is a test
%
```

这条命令还允许你基于计算结果执行指定的代码块若干次。

22.6.3 函数

zsh shell支持使用function命令或用圆括号定义函数名来创建自定义函数：

```
% function functest1 {
> echo "This is the test1 function"
}
% functest2() {
> echo "This is the test2 function"
}
% functest1
```

```
This is the test1 function
% functest2
This is the test2 function
%
```

跟bash shell函数一样（参见第16章），你可以在shell脚本中定义函数并使用全局变量或传值给该函数。这里有个使用全局变量的例子：

```
% cat test3
#!/bin/zsh
# testing functions in zsh

dbl() {
    value=$(( $value * 2 ))
    return $value
}

value=10
dbl
echo The answer is $?
% ./test3
The answer is 20
%
```

你不需要将函数放在shell脚本中。zsh shell允许在解析函数名时能访问的单独文件中定义函数。zsh shell通过`fpath`环境变量查找函数。你可以将函数文件储存在这个路径中的任何目录下。

下面是典型Linux工作站上`fpath`的值：

```
% echo $fpath
/usr/local/share/zsh/site-functions
/usr/share/zsh/4.2.5/functions/Completion
/usr/share/zsh/4.2.5/functions/Completion/AIX
/usr/share/zsh/4.2.5/functions/Completion/BSD
/usr/share/zsh/4.2.5/functions/Completion/Base
/usr/share/zsh/4.2.5/functions/Completion/Cygwin
/usr/share/zsh/4.2.5/functions/Completion/Darwin
/usr/share/zsh/4.2.5/functions/Completion/Debian
/usr/share/zsh/4.2.5/functions/Completion/Linux
/usr/share/zsh/4.2.5/functions/Completion/Mandrake
/usr/share/zsh/4.2.5/functions/Completion/Redhat
/usr/share/zsh/4.2.5/functions/Completion/Unix
/usr/share/zsh/4.2.5/functions/Completion/X
/usr/share/zsh/4.2.5/functions/Completion/zsh
/usr/share/zsh/4.2.5/functions/MIME
/usr/share/zsh/4.2.5/functions/Misc
/usr/share/zsh/4.2.5/functions/Prompts
/usr/share/zsh/4.2.5/functions/TCP
/usr/share/zsh/4.2.5/functions/Zftp
/usr/share/zsh/4.2.5/functions/Zle
%
```

如你所看到的，zsh shell会到很多地方去解析函数名。在这个系统上，你可以将函数放在`/usr/local/share/zsh/site-functions`目录中，zsh shell是能够解析它们的。

但在zsh解析任何函数前，你必须用autoload命令。这条命令会将函数加载到内存供shell访问。这里有个独立函数的例子：

```
% cat dbl
#!/bin/zsh
# a function to double a value
dbl() {
    value=$(( $1 * 2 ))
    return $value
}
% cp dbl /usr/local/share/zsh/site-functions
%
```

根据你的Linux系统是如何搭起来的，你可能需要作为root用户（或使用sudo命令）才能将文件复制到zsh库目录中。好了，现在函数已经在文件中创建并放到fpath中的目录下了。但在尝试用它时，你会得到一条错误消息，直到将它加载到了内存中：

```
% dbl 5
zsh: command not found: dbl
% autoload dbl
% dbl 5
% echo $?
10
%
```

这也适用于shell脚本。如果你有个要用的函数，你需要使用autoload命令来保证它是可用的：

```
% cat test4
#!/bin/zsh
# testing an external function

autoload dbl

dbl $1
echo The answer is $?
% ./test4 5
The answer is 10
%
```

zsh shell的另一个有趣的功能是zcompile命令。这条命令会处理函数文件并为shell创建一个编译后的版本。这跟你熟悉的其他编程语言中的编译不是一回事。但它会将函数编译成二进制格式的，从而zsh shell能够更快地加载。

当你运行zcompile命令时，它会创建这个函数文件的.zwc版本。在autoload命令在fpath中查找命令时，它会查看这个.zwc版本，并加载它而不是加载文本函数文件。

22.7 小结

本章讨论了你可能遇到的两种流行的替代Linux shell。dash shell是作为Debian Linux发行版的一部分开发的，主要出现在Ubuntu Linux发行版中。它是Bourne shell的精简版，所以它并不像bash

shell一样支持那么多功能，这可能会给脚本编程带来一些问题。

zsh shell通常会用在编程环境中，因为它为shell脚本程序员提供了许多好用的功能。它使用可加载的模块来加载单独的代码库，这让使用高级函数跟使用命令行命令一样简单。可加载模块支持很多功能，从复杂数学算法到如FTP和HTTP的网络应用。

本书的下一部分将会深入探讨Linux环境中可能会用到的一些特定脚本编程应用。下一章将介绍如何在shell脚本中使用Linux世界中最流行的两种数据库包——MySQL和PostgreSQL来处理数据。

Part 4

第四部分

高级 shell 脚本编程主题

本部分内容

- 第 23 章 使用数据库
- 第 24 章 使用 Web
- 第 25 章 使用 E-mail
- 第 26 章 编写脚本实用工具
- 第 27 章 shell 脚本编程进阶

本章内容

- MySQL数据库介绍
- PostgreSQL数据库介绍
- 创建数据库对象
- 编写数据库shell脚本

Shell脚本的一个问题是永久数据。你可以就将所有信息都保存在shell脚本变量中，但脚本运行结束后，这些变量就不存在了。有时你会想将数据保存下来备用。过去，存储和提取shell脚本中的数据需要创建一个文件，从该文件中读取数据、解析数据，然后将数据存回到该文件中。在文件中查找数据意味着要去文件中的每一个数据行来查找。现在数据库这么流行，将shell脚本和有专业水准的开源数据库对接起来非常容易。Linux中最流行的两个开源数据库是MySQL和PostgreSQL。本章将会介绍如何让这两个数据库在Linux系统上运行起来，然后花一些时间习惯在命令行上操作它们。最后介绍如何用普通bash shell脚本来与其中每一个进行交互。

23.1 MySQL 数据库

到目前为止，Linux环境中最流行的数据库是MySQL。它作为LAMP（Linux-Apache-MySQL-PHP）服务器环境的一部分而逐渐流行起来。许多因特网Web服务器都采用LAMP来搭建在线商店、博客和其他Web应用。

本节将会介绍如何在Linux环境中安装MySQL数据库，以及如何创建必要的数据库对象来在shell脚本中使用。

23.1.1 安装MySQL

过去在Linux系统上安装MySQL会有些麻烦，但现在Linux发行版通常使用自动化软件安装程序来安装（参见第8章）。这些程序不仅允许方便地从远程软件库下载和安装新软件，而且还会为已安装的软件包自动检查更新。

图23-1演示了Ubuntu Linux发行版上的添加软件功能。

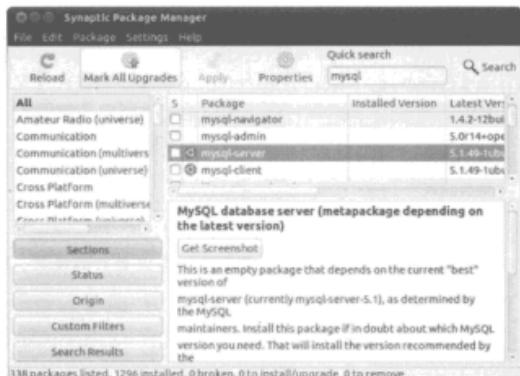


图23-1 在Ubuntu Linux上安装MySQL-

你只要选择mysql-server选项，Synaptic包管理器就会下载并安装整个MySQL服务器（以及客户端）软件了。应该没有比这更简单的办法了。

openSUSE Linux发行版也使用了一个高级软件管理系统。图23-2演示了软件管理窗口，你可以基于软件分类选择包，或者查找特定包。



图23-2 在openSUSE Linux系统上安装MySQL

还是一样，选择MySQL软件那个选项来下载和安装需要的软件包。

如果你使用的Linux发行版不支持自动化软件安装程序，或者你想用最新版本的MySQL服务器，你可以直接从MySQL官方网站（www.mysql.com）下载安装文件自行安装。

MySQL网站同时提供了为特定Linux发行版预编译的二进制包（使用Red HatRPM包格式或Debian DEB包格式）和源码包。预编译包更容易安装，所以如果有的话，最好使用它。如果需要安装二进制包或源码包的帮助信息，可参考第8章。

说明 如果选择手动下载安装MySQL（用预编译包或源码包），你必须做一些额外的工作来配置MySQL服务器，而Linux发行版安装包会安装一个配置好的设置，你几乎不用做什么工作。

23.1.2 MySQL客户端界面

到MySQL数据库的门户是mysql命令行界面程序。本节将会介绍mysql客户端程序以及如何使用它和数据库交互。

1. 连接到服务器

mysql客户端程序允许你通过用户账户和密码，连到网络中任何地方的任意MySQL数据库服务器上。默认情况下，如果你在命令行上输入mysql，不加任何参数，它会试图用Linux登录用户名连接运行在同一Linux系统上的MySQL服务器。

大多数情况下，这不是你连接数据库的方式。有很多命令行参数可用，通过使用它们，你不仅可以控制连接到哪台MySQL服务器上，还可以控制mysql界面的行为。表23-1列出了你可以和mysql程序一起使用的命令行参数。

表23-1 mysql命令行参数

参 数	描 述
-A	禁用自动重新生成哈希表
-b	禁用出错后的beep声
-B	不使用历史文件
-C	压缩客户端和服务器之间发送的所有信息
-D	指定要用的数据库
-e	执行指定语句并退出
-E	竖直方向显示查询输出，每行一个数据字段
-f	如果有SQL错误产生，继续执行
-G	使能命名命令的使用
-h	指定MySQL服务器主机名（默认为localhost）
-H	用HTML代码显示查询输出

(续)

参数	描述
-i	忽略函数名后的空格
-N	结果中不显示列名称
-o	忽略语句，除了在命令行上命名的默认数据库的语句
-p	为用户账户提示输入命令
-P	指定网络连接用的TCP端口号
-q	不缓存每条查询结果
-r	显示列输出，不转义
-s	使用安静模式
-S	为本地（localhost）连接指定一个套接字
-t	以表的形式显示输出
-T	在程序退出时显示调试信息、内存以及CPU统计信息
-u	指定登录用户名
-U	只允许指定了键值的UPDATE和DELETE语句
-v	使用详细模式
-w	如果连接没有完成，等待并重试
-X	用XHTML代码显示查询输出

如你所看到的，有一些命令行选项可以用来改变登录MySQL服务器的方式。

默认情况下，mysql客户端会尝试用你的Linux登录名来登录MySQL服务器。如果名字没有配置为MySQL的用户账户，你需要用-u参数来指定登录用户名：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 39
Server version: 5.1.49-1ubuntu8.1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

mysql>

-p参数告诉mysql程序提示输入跟登录用户名对应的密码。一旦登录上服务器，你就可以输入命令了。

2. mysql命令

mysql程序使用两种不同类型的命令：

□ 特殊的mysql命令；

□ 标准SQL语句。

mysql程序使用它自有的一组命令，方便你控制环境和提取关于MySQL服务器的信息。表23-2列出了这些命令。

表23-2 mysql命令

命 令	简写命令	描 述
?	\?	帮助信息
clear	\c	清空命令
connect	\r	连接到数据库和服务器
delimiter	\d	设置SQL语句分隔符
edit	\e	用命令行编辑器编辑命令
ego	\G	将命令发送到MySQL服务器并垂直显示结果
exit	\q	退出mysql程序
go	\g	将命令发送到MySQL服务器
help	\h	显示帮助信息
nopager	\n	禁用输出分页并将输出发送到STDOUT
note	\t	不要将输出发送到输出文件
pager	\p	将分页命令设为指定的程序（默认是more）
print	\p	打印当前命令
prompt	\R	修改mysql命令提示符
quit	\q	退出mysql程序（同exit）
rehash	\#	重新构建命令补全哈希表
source	\.	执行指定文件中的SQL脚本
status	\s	从MySQL服务器提取状态信息
system	\!	在系统上执行shell命令
tee	\T	将所有输出附加到指定文件中
use	\u	使用另外一个数据库
charset	\C	切换到另一个字符集
warnings	\W	在每条语句之后显示警告消息
nowarning	\w	不要在每条语句之后显示警告消息

你可以直接在mysql命令提示符上使用完整命令或简写命令：

```
mysql> \s
-----
mysql Ver 14.12 Distrib 5.0.45, for redhat-linux-gnu (i386) using
readline 5.0
```

```

Connection id:          10
Current database:      root@localhost
Current user:          Not in use
SSL:                  stdout
Current pager:         less
Using outfile:          ''
Using delimiter:        ;
Server version:        5.1.49-lubuntu8.1 (Ubuntu)
Protocol version:      10
Connection:            Localhost via UNIX socket
Server characterset:   latin1
Db     characterset:   latin1
Client characterset:   latin1
Conn. characterset:    latin1
UNIX socket:           /var/lib/mysql/mysql.sock
Uptime:                4 hours 15 min 24 sec

Threads: 1 Questions: 53 Slow queries: 0 Opens: 23 Flush tables:
1 Open tables: 17 Queries per second avg: 0.003
-----
```

mysql>

mysql程序实现了MySQL服务器支持的所有标准SQL（Structured Query Language，结构化查询语言）命令。23.1.3节将会进一步详细讨论。

mysql程序实现的一条常用SQL命令是SHOW命令。使用这条命令，你可以提取关于MySQL服务器的信息，比如创建的数据库和表：

```

mysql> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
+-----+
2 rows in set (0.04 sec)
```

```

mysql> USE mysql;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_mysql |
+-----+
| columns_priv   |
| db             |
| func           |
| help_category  |
| help_keyword   |
| help_relation  |
| help_topic     |
| host           |
| proc           |
| procs_priv    |
+-----+
```

```
| tables_priv      |
| time_zone       |
| time_zone_leap_second |
| time_zone_name  |
| time_zone_transition |
| time_zone_transition_type |
| user            |
+-----+
17 rows in set (0.00 sec)
mysql>
```

在这个例子中，我们用SHOW SQL命令来显示当前在MySQL服务器上配置过的数据库，然后用USE SQL命令来连接到单个数据库。mysql会话一次只能连一个数据库。

你会注意到在每个命令后面我们加了一个分号。在mysql程序中，分号表明命令的结束。如果你不用分号，它会提示输入更多数据：

```
mysql> SHOW
-> DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
+-----+
2 rows in set (0.00 sec)

mysql>
```

在处理长命令时，这个功能很有用。你可以在一行输入命令的一部分、按下回车键，然后在下一行继续输入。以这种方式，一条命令可以占任意多行，直到你用分号表明命令结束了。

注意 本章中，我们用大写字母来表示SQL命令，这已经成了编写SQL命令的通用方式，但mysql程序支持用大写或小写字母来指定SQL命令。

23.1.3 创建MySQL数据库对象

在你开始编写shell脚本来和数据库交互之前，你需要一些数据库对象。至少，你需要用到下面这些：

- 存储应用数据的唯一数据库；
- 从脚本中访问数据库的唯一用户账户；
- 组织数据的一个或多个数据表。

你可以用mysql程序来构建所有这些对象。mysql程序直接跟MySQL服务器对接，使用SQL命令来创建和修改每种对象。

你可以用mysql程序向MySQL服务器发送任何类型的SQL命令。本节将会带你逐步了解为

shell脚本构建基本数据库对象时要用的不同SQL语句。

1. 创建数据库

MySQL服务器将数据组成数据库。数据库通常保存着单个应用的数据，将它同使用这个数据库服务器的其他应用分离开。为每个shell脚本应用创建一个单独的数据库有助于消除混淆和数据混用。

创建一个新的数据库要用如下SQL语句：

```
CREATE DATABASE name
```

非常简单。当然，你必须拥有在MySQL服务器上创建新数据库的权限。最简单的办法是作为root用户登录MySQL服务器：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 39
Server version: 5.1.49-1ubuntu8.1 (Ubuntu)
```

```
Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.02 sec)
```

```
mysql>
```

你可以使用SHOW命令来查看新数据库是否创建成功：

```
mysql> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| test           |
+-----+
3 rows in set (0.01 sec)
```

```
mysql>
```

是的，它已经成功创建了。现在你应该可以连接到新数据库了：

```
mysql> USE test;
Database changed;
mysql> SHOW TABLES;
Empty set (0.00 sec)
mysql>
```

SHOW TABLES命令允许查看是否有表创建了。Empty set结果表明现在还没有可以用的表。但在我们创建表之前，还有一件事要做。

2. 创建用户账户

到目前为止，你已经了解了如何用根管理员账户连接到MySQL服务器。这个账户可以完全控制所有的MySQL服务器对象（就跟Linux的根账户可以完全控制Linux系统一样）。

为普通应用使用MySQL的根账户是极其危险的。如果有安全漏洞或有人弄到了root用户账户的密码，各种糟糕事情都可能发生在你的系统上（以及数据上）。

为了阻止这种情况的发生，在MySQL上创建只有应用中所使用数据库的权限的单独用户账户是明智的。你可以用GRANT SQL语句来完成：

```
mysql> GRANT SELECT,INSERT,DELETE,UPDATE ON test.* TO test IDENTIFIED  
by 'test';  
Query OK, 0 rows affected (0.35 sec)  
  
mysql>
```

它是一条很长的命令。让我们进一步看看每一部分，了解它具体是做什么的。

第一部分定义了用户账户对哪些数据库有哪些权限。这条语句允许用户查询数据库数据（select权限）、插入新数据行、删除已有数据行以及更新已有数据行。

test.*项定义了权限作用的数据库和表。这通过下面的格式指定：

database.table

如你在这个例子中看到的，在指定数据库和表时可以使用通配符。这种格式会将指定的权限作用在名为test的数据库中的所有表上。

最后，你可以指定权限作用的用户账户。grant命令的便利之处在于，如果用户账户不存在，它会创建它。identified by部分允许你为新用户账户设定默认密码。

你可以直接在mysql程序中测试新用户账户：

```
$ mysql test -u test -p  
Enter password:  
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 40  
Server version: 5.1.49-lubuntu8.1 (Ubuntu)  
  
Type ' help;' or '\h' for help. Type '\c' to clear the buffer.  
  
mysql>
```

第一个参数指定使用的默认数据库（test）。如你所看见的，-u参数定义了登录的用户，-p用来提示输入密码。输入test用户账户的密码后，你就连到服务器了。

现在已经有了数据库和用户账户，可以为数据创建一些表了。但首先，我们看一下可以用的另一个数据库。

23.2 PostgreSQL 数据库

PostgreSQL数据库起初是作为一个学术项目发起的，用来演示如何将高级数据库技术集成到

函数式数据库服务器中。多年来，PostgreSQL已经演变为Linux环境中可用的最先进的开源数据库服务器。

本节将带你逐步学习安装、运行PostgreSQL数据库服务器，然后建立用户账户和数据库供shell脚本使用。

23.2.1 安装PostgreSQL

跟MySQL类似，你可以使用系统自动化软件安装系统安装PostgreSQL数据库服务器包，或手动从PostgreSQL官方网站（www.postgresql.org）下载安装。

为使用Linux发行版的自动化软件安装系统，可以沿用23.1.1节中列出的步骤。类似于MySQL，PostgreSQL下载页面提供了为若干Linux发行版预编译的二进制包和源码包。仍然一样，参考第8章来判断你的Linux发行版是否使用了PostgreSQL二进制包支持的包管理系统。如果不是，你需要下载源码包，在Linux系统上手动编译。这是一个有趣的工作。

说明 编译源码包需要在Linux系统上安装C软件开发包。现今在服务器上这已经很普遍了，但如果你用的是桌面Linux发行版的话，可能还没有安装。关于如何编译软件以及编译C源码项目需要哪些软件包，请参考你的Linux发行版的文档。

安装好PostgreSQL数据库服务器后，登录PostgreSQL服务器会和登录MySQL服务器稍稍不同。可能你还记得，MySQL数据库维护着自己的内部用户数据库。虽然PostgreSQL也具备这种能力，但大多数PostgreSQL实现（包括默认源码安装）都会利用现有的Linux系统用户账户来认证PostgreSQL用户。

虽然这有时会叫人困惑，但它提供了一个友好而简洁的方式来在PostgreSQL中控制用户账户。你只需要保证每个PostgreSQL用户都在Linux系统上有一个有效账户就行了，而不用担心一组完整而独立的用户账户。

PostgreSQL的另外一大区别在于，PostgreSQL的管理员账户称为postgres，而不是root。鉴于这个要求，在安装PostgreSQL时，系统上必须有一个叫做postgres的用户账户。

下一节中我们看一看如何用postgres账户访问PostgreSQL服务器。

23.2.2 PostgreSQL命令行界面

PostgreSQL命令行客户端程序称作psql。这个程序提供了对PostgreSQL服务器中配置的数据对象的完整访问。本节将会介绍psql命令并演示如何用它来和PostgreSQL服务器交互。

1. 连接服务器

psql客户端程序提供了访问PostgreSQL服务器的命令行界面。如你所期望的，它使用命令行参数来控制哪些功能在客户端界面中使能了。每个选项都采用完整格式或简写格式。表23-3列出了可用的命令行参数。

表23-3 psql命令行参数

简写名称	完整名称	描述
-a	--echo-all	在输出中显示脚本文件中执行的所有SQL行
-A	--no-align	将输出格式设为非对齐模式。数据不会显示成格式化的表
-c	--command	执行指定的SQL语句并退出
-d	--dbname	指定要连接的数据库
-e	--echo-queries	将所有的查询输出到屏幕上
-E	--echo-hidden	将隐藏的psql元命令输出到屏幕上
-f	--file	执行指定文件中的SQL命令并退出
-F	--field-separator	指定在非对齐模式中分开列数据的字符。默认是逗号
-h	--host	指定远程PostgreSQL服务器的IP地址或主机名
-l	--list	显示服务器上已有的数据库列表并退出
-o	--output	将查询输出重定向到指定文件中
-p	--port	指定要连接的PostgreSQL服务器的TCP端口
-P	--pset	将表打印选项设为指定的值
-q	--quiet	安静模式，不会显示输出消息
-R	--record-separator	将指定字符作为数据行分隔符。默认为换行符
-s	--single-step	在每个SQL查询后提示继续还是退出
-S	--single-line	指定回车键而不是分号为一个SQL查询的结束
-t	--tuples-only	在表输出中禁用列的头部和尾部
-T	--table-attr	在HTML模式时使用指定的HTML表标签
-U	--username	使用指定的用户名连接PostgreSQL服务器
-v	--variable	将指定变量设成指定值
-V	--version	显示psql版本号并退出
-W	--password	强制命令提示符
-X	--expanded	使能扩展表输出以显示数据行的额外信息
-X	--nopsqlrc	不要运行psql启动文件
-?	--help	显示psql命令行帮助信息并退出

如上一节中提到的，PostgreSQL的管理员账户称作postgres。因为PostgreSQL使用Linux用户账户来验证用户，你必须以Linux账户postgres登录才能以postgres用户身份访问PostgreSQL服务器。

要绕过这个问题，可以用sudo命令以postgres用户账户来运行psql命令程序：

```
$ sudo -u postgres psql
[sudo password for rich]:
psql (8.4.5)
Type " help" for help.

postgres#
```

默认psql提示符指明了你连接的数据库。提示符中的井号（#）表明你已经作为管理员用户登录了。现在你可以开始输入一些命令来和PostgreSQL服务器交互了。

2. psql命令

类似于mysql程序，psql程序使用两种不同类型的命令：

- 标准SQL语句；
- PostgreSQL元命令。

PostgreSQL元命令允许你方便地获取有关数据库环境的确切信息，此外还具有psql会话的set功能。元命令用反斜线来标识。针对许多不同的设置和功能有大量PostgreSQL元命令，但现在还不必要考虑它们。最常用的有：

- \l来列出已有数据库；
- \c来连接到数据库；
- \dt来列出数据库中的表；
- \du来列出PostgreSQL用户；
- \z来列出表的权限；
- \?来列出所有可用的元命令；
- \h来列出所有可用的SQL命令；
- \q来退出数据库。

如果你要找一个元命令，只要输入\?元命令就行了。你会看到一个所有可用元命令的列表以及附带说明。

要测试元命令，使用\l元命令来列出已有数据库：

```
postgres=# \l
List of databases
   Name    | Owner     | Encoding | Collation | Ctype    | Access
-----+-----+-----+-----+-----+-----+
 postgres | postgres | UTF8    | en_US.utf8 | en_US.utf8 |
 template0 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =c/postgres
 template1 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =c/postgres
(3 rows)
postgres#
```

列表给出了服务器上已有的数据库，以及它们的功能（我们去掉了Access列的内容，这样才能刚好匹配页面宽度）。PostgreSQL服务器提供了一些默认数据库。postgres数据库维护着服务器的所有系统数据。template0和template1数据库为你提供了创建新数据库时可复制的默认数据库模板。

现在你可以开始在PostgreSQL中处理你自己的数据了。

23.2.3 创建PostgreSQL数据库对象

本节将会带你逐步学习创建数据库和访问它的用户账户的过程。你会了解到，虽然PostgreSQL中的有些工作跟MySQL中的完全一样，但有些则完全不同。

1. 创建数据库对象

创建数据库的方法跟MySQL的差不多。记住要以postgres管理员账户登录来创建新数据库：

```
$ sudo -u postgres psql
psql (8.4.5)
Type "help" for help.

postgres=# CREATE DATABASE test;
CREATE DATABASE
postgres#
```

创建数据库之后，用\l元命令来看看它是否出现在了数据库列表中，然后用\c元命令连接：

```
postgres=# \l
List of databases
   Name    | Owner     | Encoding | Collation | Ctype      | Access
-----+-----+-----+-----+-----+-----+
postgres | postgres | UTF8    | en_US.utf8 | en_US.utf8 | 
template0 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =c/postgres
template1 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =c/postgres
test     | postgres | UTF8    | en_US.utf8 | en_US.utf8 | 
(4 rows)

postgres=# \c test
psql (8.4.5)
You are now connected to database " test".
test#
```

在连接到test数据库时，psql提示符改变了，显示的是新数据库名。它会提醒你已经可以创建数据库对象了，这样你就能方便地知道你在系统中的什么位置。

说明 PostgreSQL向数据库增加了一个控制层，称为模式（schema）。数据库可以有多个模式，每个模式包含多个表。这样允许你根据特定应用或用户将数据库进一步细分。

默认情况下，每个数据库都有一个模式，称为public。如果你只想让一个应用使用数据库，用这个public模式工作就可以了。如果你要做得更好一些，可以创建新模式。在这个例子中，我们使用了public模式来创建表。

2. 创建用户账户

在创建了新数据库之后，下一步就是创建用户账户来在shell脚本中访问新建数据库。你已经看到了，PostgreSQL中的用户账户跟MySQL中的有很大不同。

PostgreSQL中的用户账户称为登录角色（Login Role）。PostgreSQL服务器会将登录角色和Linux系统用户账户匹配。正因如此，有两种常用方法来创建登录角色来运行访问PostgreSQL数据库的shell脚本：

- 创建一个跟PostgreSQL登录角色对应的特殊Linux账户来运行所有的shell脚本；
- 为每个需要运行shell脚本来访问数据库的Linux用户账户创建PostgreSQL账户。

举个例子，让我们选择第二种方法来创建跟Linux用户账户对应的PostgreSQL账户。这样，你可以直接在Linux用户账户中运行访问PostgreSQL数据库的shell脚本。

首先，你必须创建登录角色：

```
test=# CREATE ROLE rich login;
CREATE ROLE
test#
```

非常简单。不用`login`参数的话，这个角色是不允许登录到PostgreSQL服务器的，但可以被授予一些权限。这种角色类型称为组角色（group role）。当你在有大量用户和表的大型环境中工作时，组角色非常有用。你不用记录哪个用户对哪些表有哪些权限，而只要为对表的特定访问类型创建一些组角色，然后将登录角色分配给适当的组角色就可以了。

对于简单的shell脚本编程，你基本不用考虑创建组角色，直接将权限赋给登录角色就行。这正是我们在这个例子中的做法。

不过，PostgreSQL处理权限的方式跟MySQL有些不同。它不允许你将所有权限赋给匹配到表一级的所有数据库对象。相反，你需要为每一个新建的表授予权限。虽然这有些痛苦，但它确实有助于增强严格的安全策略。直到创建表后，你才能授予权限。这正是这个过程中的下一步。

23.3 使用数据表

现在你已经让你的MySQL或PostgreSQL服务器运行起来了，并且新建了一个数据库和一个访问它的用户账户，可以开始处理数据了。幸好mysql和psql程序都用标准SQL来创建和管理数据表。本节将带你逐步学习一些SQL来在两种环境中创建表、插入和删除数据以及查询已有数据。

23.3.1 创建数据表

MySQL和PostgreSQL服务器都被当做关系数据库（relational database）。在关系数据库中，数据由数据字段、数据行和表组成。数据字段是单独一条信息，比如员工的姓或工资。数据行是相关数据字段的集合，比如员工ID号、姓、名、地址和工资。每个数据行都代表一组数据字段。

表含有保存相关数据的所有数据行。因此，你会有一个叫做Employees的表来保存每个员工的数据行。

要在数据库中新建一张表，你需要用SQL命令`CREATE TABLE`：

```
$ mysql test -u root -p
Enter password:
mysql> CREATE TABLE employees (
    -> empid int not null,
    -> lastname varchar(30),
    -> firstname varchar(30),
    -> salary float,
    -> primary key (empid));
Query OK, 0 rows affected (0.14 sec)
```

```
mysql>
```

首先，注意要新建一张表，我们需要用root用户账户登录到MySQL上，因为test用户没有新建表的权限。下一步，注意我们在mysql程序命令行上指定了test数据库。不那么做的话，我们需要用SQL命令`USE`来连接到test数据库。

警告 在创建新表前确保你在正确的数据库中极其重要。还要确保你用管理员用户账户（MySQL中的root用户，PostgreSQL中的postgres用户）登录来创建表。

表中的每个数据字段都用数据类型来定义。MySQL和PostgreSQL数据库支持许多不同的数据类型。表23-4列出了其中较流行的一些数据类型。

表23-4 MySQL和PostgreSQL的数据类型

数据类型	描述
char	定长字符串
Varchar	变长字符串
int	整数值
float	浮点值
Boolean	布尔类型true/false值
Date	YYYY-MM-DD格式的日期值
Time	HH:mm:ss格式的时间值
Timestamp	日期和时间值的组合
Text	长字符串值
BLOB	大的二进制值，比如图片或视频剪辑

empid数据字段还指定了一个数据约束（data constraint）。数据约束会限制输入的什么类型数据可以创建一个有效的数据行。not null数据约束指明每个数据行都必须有一个指定的empid值。

最后，primary key定义了可以唯一标识每个数据行的数据字段。这意味着在表中每个数据行都必须有一个唯一的empid值。

新建了表之后，你可以用对应的命令来确保它创建成功了。在mysql中，用show tables命令：

```
mysql> show tables;
+-----+
| Tables_in_test |
+-----+
| employees      |
+-----+
1 row in set (0.00 sec)
```

```
mysql>
```

在psql中，用\dt元命令：

```
test=# \dt
          List of relations
 Schema |   Name    | Type  | Owner
-----+-----+-----+-----+
 public | employees | table | postgres
(1 row)

test=#

```

你可能还记得在23.2.3节中，在PostgreSQL里你必须在表一级分配权限。现在你有了一张表，你需要将你的登录角色给它：

```
$ sudo -u postgres psql
psql (8.4.5)
Type " help" for help.

postgres=# \c test
psql (8.4.5)
You are now connected to database " test".
test=# GRANT SELECT, INSERT, DELETE, UPDATE ON public.employees TO rich;
GRANT
test=#

```

用来指定表的格式必须包含模式名，默认为public。还有，记住以postgres登录角色来执行这条命令，并连接到test数据库。

有了新建的表，现在你可以开始保存一些数据了。下一节将会介绍怎么做。

23.3.2 插入和删除数据

一点也不新奇，用SQL命令INSERT来向表插入一些新数据行。每条INSERT命令都必须指定数据字段值来供MySQL或PostgreSQL服务器接受该数据行。

INSERT SQL命令的格式如下：

```
INSERT INTO table VALUES (...)
```

每个数据字段的值都会用逗号分开：

```
$ mysql test -u test -p
Enter password:
mysql> INSERT INTO employees VALUES (1, ' Blum' , ' Rich' , 25000.00);
Query OK, 1 row affected (0.35 sec)
```

或者，在PostgreSQL中：

```
$ psql test
psql (8.4.5)
Type " help" for help.

test=> INSERT INTO employees VALUES (1, ' Blum' , ' Rich' , 25000.00);
INSERT 0 1
test=>
```

MySQL的例子用-u命令行参数来以test用户账户登录。PostgreSQL的例子用当前Linux用户账户登录，所以它用了前面创建的rich用户账户。

INSERT命令会将指定的数据推送到表中的数据字段里。如果你试图添加另外一条重复了empid数据字段值的数据行，你会得到一条错误消息：

```
mysql> INSERT INTO employees VALUES (1, ' Blum' , ' Barbara' , 45000.00);
ERROR 1062 (23000): Duplicate entry ' 1' for key 1
```

但如果你将empid的值改成唯一的值，一切就都可以了：

```
mysql> INSERT INTO employees VALUES (2, 'Blum', 'Barbara', 45000.00);
Query OK, 1 row affected (0.00 sec)
```

现在在表中，你应该有两个数据行了。

如果你需要从表中删除数据，可以用DELETE SQL命令，但要非常小心。

DELETE命令的基本格式如下：

```
DELETE FROM table;
```

其中table指定了要从中删除数据行的表。这个命令有个小问题，它会删除该表中所有数据行。

要指定只删除一条或多条数据行，必须用WHERE子句。WHERE子句允许创建一个过滤器来指定删除哪些数据行。可以像下面这样使用WHERE子句：

```
DELETE FROM employees WHERE empid = 2;
```

这会将删除限定在empid值为2的所有数据行。当你执行这条命令时，mysql程序会返回一条消息来说明有多少个数据行符合条件：

```
mysql> DELETE FROM employees WHERE empid = 2;
Query OK, 1 row affected (0.29 sec)
```

跟期望的一样，只有一个数据行符合条件并被删除。

23.3.3 查询数据

一旦将所有数据都弄到了数据库中，就可以开始执行报告功能来提取信息了。

所有查询都是用SELECT SQL命令来完成。SELECT命令非常强大，但随之而来的是复杂性。

SELECT语句的基本格式如下：

```
SELECT datafields FROM table
```

datafields参数是用逗号分开的你希望查询返回的数据字段名称。如果你要提取所有的数据字段值，可以用星号来做通配符。

你还必须指定查询要查找的特定表。要得到有意义的结果，你必须将查询数据字段和正确的表对应起来。

默认情况下，SELECT命令会返回指定表中的所有数据行：

```
mysql> SELECT * FROM employees ;
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
| 1 | Blum | Rich | 25000 |
| 2 | Blum | Barbara | 45000 |
| 3 | Blum | Katie Jane | 34500 |
| 4 | Blum | Jessica | 52340 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

```
mysql>
```

你可以用一个或多个修饰符来定义数据库服务器如何返回查询要找的数据。下面列出了常用的修饰符。

□ WHERE：显示符合特定条件的数据行子集。

□ ORDER BY：以指定顺序显示数据行。

□ LIMIT：只显示数据行的一个子集。

WHERE子句是最常用的SELECT命令修饰符。它允许你指定条件来过滤结果中的数据。这里有个使用WHERE子句的例子：

```
mysql> SELECT * FROM employees WHERE salary > 40000;
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
|     2 | Blum    | Barbara   | 45000 |
|     4 | Blum    | Jessica  | 52340 |
+-----+-----+-----+-----+
2 rows in set (0.01 sec)
```

mysql>

现在你可以看到将数据库访问添加到shell脚本的强大之处了。你可以轻松地控制你的数据管理需要，只要一些SQL命令以及mysql或psql程序就行了。

下一节将会介绍如何将这些功能放到shell脚本中。

23.4 在脚本中使用数据库

现在你已经有了一个可以工作的数据库在运行，终于可以将精力放回shell脚本编程了。本节将会介绍如何用shell脚本同数据库交互。

23.4.1 连接到数据库

显然，要连接到数据库上，你必须在shell脚本中利用mysql或psql程序。这并不是一个特别复杂的过程，但你要注意一些事情。

1. 查找客户端程序

你需要跨过的第一个障碍是在Linux系统上找出mysql和psql命令行客户端程序在什么位置。Linux软件安装不好的一点是通常不同的Linux发行版会将软件包安装到不同位置。

幸运的是，有which命令可以帮忙。which命令会告诉你shell在命令行上试图运行一个命令时会去哪里查找这个命令：

```
$ which mysql
/usr/bin/mysql
$ which psql
/usr/bin/psql
$
```

处理这个信息最简单的办法是将它赋给一个环境变量，然后当你要引用对应的程序时在shell脚本中使用这个变量：

```
MYSQL=' which mysql'
PSQL=' which psql'
```

现在\$MYSQL变量会指向mysql程序的可执行文件，而\$PSQL变量则会指向psql程序的可执行文件。

2. 登录到服务器

在找到客户端程序的位置后，你可以在脚本中用它们来访问数据库服务器。对于PostgreSQL服务器来说，这非常简单：

```
$ cat ptest1
#!/bin/bash
# test connecting to the PostgreSQL server

PSQL=' which psql'

$PSQL test
$ ./ptest1
psql (8.4.5)
Type " help" for help.

test=>
```

由于脚本是在某个Linux用户账户中运行而PostgreSQL账户以同样的名字存在，你要在psql命令行上指定的只是要连接的数据库名称。ptest1脚本连接到test数据库，然后会将你留在数据库中的psql提示符上。

如果已经在MySQL中为shell脚本创建了一个特殊用户账户，那么你需要在mysql命令行上指定它：

```
$ cat mtest1
#!/bin/bash
# test connecting to the MySQL server

MYSQL=' which mysql'

$MYSQL test -u test -p
$ ./mtest1
Enter password:
Reading table .information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 43
Server version: 5.1.49-lubuntu8.1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type ' help;' or ' \h' for help. Type ' \c' to clear the current input statement.

mysql>
```

它能工作起来，但对于非交互式脚本来说并不够好。-p命令行参数导致mysql暂停下来并要用户输入密码。你可以将密码也放在命令行上来解决这个问题：

```
$ MySQL test -u test -ptest
```

但这并不是一个好办法。任何能访问脚本的人都会看到数据库的用户账户和密码。

要解决这个问题，你可以用mysql程序的一个特殊配置文件。mysql程序使用\$HOME/.my.cnf文件来读取特殊的启动命令和设置。其中一项设置是由该用户账户发起的mysql会话的默认密码。

要在这个文件中设置默认密码，可以加入下面内容：

```
$ cat .my.cnf
[client]
password = test
$ chmod 400 .my.cnf
$
```

chmod命令用来限制.my.cnf文件访问，这样就只有你能查看它。现在你可以在命令行上测试一下：

```
$ mysql test -u test
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 44
Server version: 5.1.49-1ubuntu8.1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

mysql>

太好了！现在你不用在shell脚本的命令行中包含密码了。

23.4.2 向服务器发送命令

在建立起到服务器的连接后，你会想发送命令来和数据库交互。有两种方法来交互：

- 发送单个命令并退出；
- 发送多个命令。

要发送单个命令，你必须将命令作为mysql或psql命令行的一部分包含进去。

对于mysql命令，你可以用-e参数：

```
$ cat mtest2
#!/bin/bash
# send a command to the MySQL server

MySQL=`which mysql`
```

```
$ MYSQL test -u test -e 'select * from employees'
$ ./mtest2
+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+
|    1 | Blum     | Rich      | 25000   |
|    2 | Blum     | Barbara   | 45000   |
|    3 | Blum     | Katie Jane | 34500   |
|    4 | Blum     | Jessica  | 52340   |
+-----+-----+-----+
$
```

对于psql命令，你可以用-c参数：

```
$ cat ptest2
#!/bin/bash
# send a command to the PostgreSQL server

PSQL=`which psql`:

$PSQL test -c 'select * from employees'
$ ./ptest2
empid | lastname | firstname | salary
-----+-----+-----+
1 | Blum     | Rich      | 25000
2 | Blum     | Barbara   | 45000
3 | Blum     | Katie Jane | 34500
4 | Blum     | Jessica  | 52340
(4 rows)
$
```

数据库服务器会将SQL命令的结果返回给shell脚本，会将它们显示在STDOUT中。

如果你需要发送多条SQL命令，你可以使用文件重定向（参见第14章）。要在shell脚本中重定向行，你必须定义一个结束字符串（end of file string）。结束字符串指明了重定向数据的开始和结尾。

下面有个定义了结束字符串及其中数据的例子：

```
$ cat mtest3
#!/bin/bash
# sending multiple commands to MySQL

MYSQL=`which mysql`:
$MYSQL test -u test <<EOF
show tables;
select * from employees where salary > 40000;
EOF
$ ./mtest3
Tables_in_test
employees
empid      lastname    firstname    salary
2          Blum        Barbara      45000
4          Blum        Jessica    52340
$
```

shell会将EOF分隔符之间的所有内容都重定向给mysql命令。mysql命令会执行这些行，就像你在提示符下亲自输入的一样。用这种方法，只要需要你可以向MySQL服务器发送任意多的命令。但你会注意到，在输出和每条命令之间没有没有任何分隔。在23.4.3节中，你会了解如何解决这个问题。

说明 你应该还注意到了，当你使用重定向输入方法时，mysql程序改变了默认的输出风格。mysql程序检测到了输入是重定向过来的，所以它只返回了原始数据而不是在数据两边加上ASCII符号框。这非常便于提取单独数据元素。

同样的方法也适用于psql命令：

```
$ cat ptest3
#!/bin/bash
# sending multiple commands to PostgreSQL.

PSQL='`which psql`'

$PSQL test <<EOF
\dt
select * from employees where salary > 40000;
EOF
$ ./ptest3
      List of relations
 Schema |   Name    | Type  | Owner
-----+-----+-----+
 public | employees | table | postgres
(1 row)

empid | lastname | firstname | salary
-----+-----+-----+
      2 | Blum     | Barbara   | 45000
      4 | Blum     | Jessica   | 52340
(2 rows)
$
```

psql程序直接在STDOUT上按指定顺序显示了每条命令的输出。

当然，你并未被限制只能从数据表中提取数据。你可以在脚本中使用任何类型的SQL命令，比如INSERT语句：

```
$ cat mtest4
#!/bin/bash
# send data to the table in the MySQL database

MYSQL='`which mysql`'

if [ $# -ne 4 ]
then
  echo "Usage: mtest4 empid lastname firstname salary"
else
```

```

statement=" INSERT INTO employees VALUES ($1, '$2', '$3', '$4')"
$MYSQL test -u test << EOF
$statement
EOF
if [ $? -eq 0 ]
then
    echo Data successfully added
else
    echo Problem adding data
fi
fi
$ ./mtest4
Usage: mtest4 empid lastname firstname salary
$ ./mtest4 5 Blum Jasper 100000
Data added successfully
$ 
$ ./mtest4 5 Blum Jasper 100000
ERROR 1062 (23000) at line 1: Duplicate entry '5' for key 1
Problem adding data
$ 

```

这个例子演示了使用这种方法的一些注意事项。在指定结束字符串时，它必须是该行的唯一内容，并且该行必须以这个字符串开头。如果我们将EOF文本缩进以和其余的if-then缩进对齐，它就不会起作用了。

在INSERT语句里，注意在文本值周围用了单引号，在整个INSERT语句周围用了双引号。一定不要弄混了引用字符串值的引号和定义脚本变量文本的引号。

还有，注意我们怎样使用\$?特殊变量来测试mysql程序的退出状态码。它会方便你判断命令是否成功执行。

只将这些命令的结果发送到STDOUT并不是管理和操作数据的最简单方法。下一节将会演示一些技巧来帮助脚本抓取从数据库中提取出来的数据。

23.4.3 格式化数据

mysql和psql命令的标准输出并未让它特别适合提取数据。如果要用提取的数据做点实际的事情，你需要做一些特别的操作。本节将会介绍一些技巧来帮你从数据库报告中提取数据。

1. 将输出赋给变量

提取数据库数据的第一步是将mysql和psql命令的输出重定向到一个环境变量中。这允许你在其他命令中使用输出信息。这里有个例子：

```

$ cat mtest5
#!/bin/bash
# redirecting SQL output to a variable

MYSQL=`which mysql` 

$db=' $MYSQL test -u test -Bse " show databases" '
for db in $db

```

```
do
    echo $db
done
$ ./mtest5
information_schema
test
$
```

这个例子在mysql程序的命令行上用了两个额外参数。-B参数指定mysql程序工作在批处理模式下，还用了-s (silent) 参数，列标题和格式化符号都被禁掉了。

通过将mysql命令的输出重定向到一个变量，这个例子可以逐步输出每个返回的数据行里的每个值。

2. 使用格式化标签

在前面的例子中，你了解了给mysql程序命令行加上-B和-s参数是如何禁掉输出的标题信息的，所以你得到的只有数据。你可以用一些其他参数来方便使用。

为Web页面生成数据是现今常见的任务。mysql和psql程序都提供了一个以HTML格式显示结果的选项。两个程序都用-H命令行参数来使能这个功能：

```
$ psql test -H -c 'select * from employees where empid = 1'
<table border="1" >
<tr>
    <th align="center" >empid</th>
    <th align="center" >lastname</th>
    <th align="center" >firstname</th>
    <th align="center" >salary</th>
</tr>
<tr valign=" top" >
    <td align=" right" >l</td>
    <td align=" left" >Blum</td>
    <td align=" left" >Rich</td>
    <td align=" right" >25000</td>
</tr>
</table>
<p>(1 row)<br />
</p>
$
```

mysql程序还支持另外一种流行格式，称做可扩展标记语言（eXtensive Markup Language, XML）。这种语言使用类HTML标签来标识数据名和值。

对于mysql程序，你可以用-X命令行参数来输出：

```
$ mysql test -u test -e 'select * from employees where empid = 1'
<?xml version=" 1.0" ?>

<resultset statement=" select * from employees" >
<row>
    <field name=" empid" >l</field>
    <field name=" lastname" >Blum</field>
```

```
<field name="firstname" >Rich</field>
<field name="salary" >25000</field>
</row>
</resultset>
$
```

使用XML，你可以轻松地标识每个数据行中的单独数据行和每行中的数据值。

23.5 小结

本章讨论了如何在shell脚本中保存、修改和提取数据库中的数据。你可以在shell脚本中轻松地访问MySQL和PostgreSQL数据库服务器。

在安装了MySQL和PostgreSQL服务器后，你可以用它们相应的客户端程序来在命令行上或shell脚本中访问服务器。mysql客户端程序提供了到MySQL服务器的命令行界面。你可以在脚本中将SQL命令以及定制的MySQL命令发送到服务器，然后提取结果。

psql客户端程序访问PostgreSQL服务器的方式也一样。有很多命令行参数可以用来帮助将数据格式化为正确的格式。

这两个客户端程序都允许你发送单个命令到服务器或使用输入重定向来发送一批命令。程序通常会将数据库返回的数据结果发送到STDOUT，但你可以将输出重定向到一个变量，然后在shell脚本中使用这些信息。

下一章将会介绍万维网。让shell脚本和互联网上的网站对接起来有点麻烦，但一旦掌握了，从哪里提取数据就取决于你了。



本章内容

- 用Lynx程序冲浪
- 探索cURL程序
- 客户端/服务器zsh编程

通 常在考虑shell脚本编程时，最不可能考虑到的就是互联网了。命令行世界看起来往往跟丰富多彩的互联网世界有些格格不入。但你可以用一些不同的实用工具来在shell脚本中访问Web中以及其他网络设备上的数据内容。本章将带你逐步了解3种流行的方法来让shell脚本和网络世界交互。

24.1 Lynx 程序

几乎和互联网同时出现，Lynx程序作为基于文本的浏览器，在1992年由堪萨斯大学的学生创建。由于是基于文本的，Lynx程序允许你直接从终端会话中浏览网站，将那些Web页面上的美丽图片替换成了HTML文本标签。这允许你在基于任何类型的Linux终端上都能浏览互联网。图24-1是一个Lynx界面的例子。



图24-1 使用Lynx查看Web页面

Lynx使用标准键盘按键来浏览网页。链接会在Web页面上以高亮文本的形式出现。使用向右方向键可以跟随一个链接到下一个Web页面。

你可能想问如何在shell脚本中使用图形化文本程序。Lynx程序还提供了一个功能，允许你将Web页面的文本内容转存到STDOUT中。这个功能非常适合用来挖掘Web页面中包含的数据。本节将会介绍如何在shell脚本中用Lynx程序来从互联网网站上提取数据。

24.1.1 安装Lynx

尽管Lynx程序有点古老，但它仍在积极开发中。在本书写作时，Lynx的最新版本是2010年6月发布的2.8.8，且新版本正在开发中。鉴于它在shell脚本程序员中十分流行，许多Linux发行版都将它作为默认安装。

如果你正在用一个不带Lynx程序的Linux系统，检查一下该发行版的安装包。大多数情况下你都能在那里找到Lynx包并轻松安装。

如果发行版没有提供Lynx包，或者你想用最新版的，你可以从lynx.isc.org网站上下载源码并编译（假定你已经在Linux系统上安装了C开发库）。参考第8章获取有关如何编译并安装源码包的信息。

说明 Lynx程序使用了Linux中的curses文本图形库。大多数发行版默认会安装这个库。如果你的发行版没有安装，在尝试编译Lynx前先参考你的发行版的安装指南来安装curses库。

下一节将会介绍如何在命令行上使用lynx命令。

24.1.2 lynx命令行

lynx命令行命令在从远程网站上提取信息方面极其强大。当你用浏览器查看Web页面时，你只是看到了传送到浏览器的信息中的一部分。Web页面由3种类型的数据元素组成：

- HTTP头；
- Cookie；
- HTML内容。

HTTP头提供了有关连接中传送的数据类型、发送数据的服务器以及连接中采用的安全类型的信息。如果你正在发送特殊类型的数据，比如视频或音频剪辑，服务器会在HTTP头中识别。Lynx程序允许你查看Web页面会话中所有发出去的HTTP头。

如果你浏览过Web页面，那么你一定了解Web页面cookie。网站用cookie来存储有关网站访问的数据供将来使用。每个网站都能存储信息，但只能访问它自己设置的信息。lynx命令提供了一些选项来查看Web服务器发送的cookie，以及接受或拒绝服务器发过来的特定cookie。

Lynx程序支持3种不同的查看Web页面真实HTML内容的格式：

- 在终端会话中采用curses图形库的文本图形显示；

- 文本文件，从Web页面中转储的原始数据；
- 文本文件，从Web页面中转储的原始HTML源码。

对于shell脚本，查看原始数据或HTML源码是万能的。一旦你抓到了从网站上提取的信息，你就能轻松地从中提取每一条信息。

如你所看到的，Lynx程序在它所能处理的方面非常强大。但随之而来的是复杂性，尤其是对命令行参数来说。Lynx程序是你在Linux世界中遇到的较复杂的程序之一。

lynx命令的基本格式如下：

```
lynx options URL
```

其中URL是你要连接的HTTP或HTTPS目的地，options则是一个或多个选项，这些选项可以在Lynx与远程网站交互时改变它的行为。表24-1列出了你可以跟lynx命令一起使用的所有命令行参数。

表24-1 Lynx命令行参数

参数	描述
-	接受来自STDIN的选项和参数
-accept_all_cookies	使能Set-Cookie处理的话，接受cookie但不提示。默认设为off
-anonymous	对匿名账户实行限制
-assume_charset=name	未指定字符集的文档的默认字符集
-assume_local_charset=name	本地文件的默认字符集
-assume_unrec_charset=name	有不能识别的字符集时采用的默认字符集
-auth=id:pw	访问受保护文档的认证信息
-base	在-source的文本/html输出前加一条请求URL注释和BASE标签
-bibhost=URL	本地bib服务器URL（默认为http://bibhost/）
-book	将标签页作为起始页。默认设为off
-buried_news	开启对埋藏的引用扫描新闻文章。默认设为on
-cache=n	内存中缓存的文档总数
-case	开启用户搜索的大小写区分。默认设为off
-center	开启HTML <table>标签中内容的居中对齐。默认设为on
-cfg=filename	指定一个配置文件而不用默认的lynx.cfg文件
-child	在开始文件中用向左方向键退出，并停止保存到硬盘上
-cmd_log=filename	将按键命令记录到指定文件中
-cmd_script=filename	从指定文件中读取按键命令
-connect_timeout=n	设置连接超时（以秒为单位）。默认为18 000 s
-cookie_file=filename	指定用来读取cookie的文件
-cookie_save_file=filename	指定用来存储cookie的文件
-cookies	处理Set-Cookies头。默认设为on
-core	出现严重错误时强制内核转储（core dump）。默认设为off

(续)

参数	描述
-crawl	和-traversal一起使用，将每个页面输出到一个文件中；和-dump一起使用时，跟-traversal一样格式化输出，不过输出到STDOUT
-curses_pads	使用curses的pad功能来支持左右移动。默认设为on
-debug_partial	以MessageSecs延迟来显示增量显示步骤。默认设为off
-debug=n	设置状态行消息上的延迟（以秒为单位）。默认设为0.000
-display=display	为X窗口程序设置显示变量
-display_charset=name	终端输出的字符集
-dont_wrap_pre	当-dump和crawl开启式不要在 <pre>部分自动换行。在交互式会话中标记自动换行。默认设为on</pre>
-dump	将第一个URL转储到STDOUT并退出
-editor=editor	用指定编辑器开启编辑模式
-emacskeys	开启类emacs的按键移动。默认设为off
-enable_scrollback	开启对回滚键的兼容。默认设为off
-error_file=filename	将HTTP状态码写到指定文件中
-exec	开启执行本地命令
-force_empty_hrefless_a	强制不带href属性的 <a>元素为空。默认设为off
-force_html	强制将第一个文档解释为HTML。默认设为off
-force_secure	对SSL cookie要求安全标记。默认设为off
-forms_options	使用基于表单的选项菜单。默认设为on
-from	使能From头的传送。默认设为on
-ftp	禁止FTP访问。默认设为off
-get_data	为get表单从STDIN读取数据，以---结尾
-head	发送HEAD请求。默认设为off
-help	打印用法信息
-hiddenlinks=option	指定如何处理隐藏链接。option可以是merge、listonly或ignore
-historical	使用>而不是-->作为注释的结尾。默认设为off
-homepage=URL	将主页和起始页分开设置
-image_links	为所有图片开启包含链接功能。默认设为off
-index=URL	设置默认的索引文件名
-ismap	当支持客户端MAP时，包含ISMAP链接。默认设为off
-link=n	为-crawl生成的lnk#.dat文件设置计数起点。默认为0
-localhost	禁止指向远程主机的URL。默认设为off
-locexec	只使能本地文件中的本地程序执行。默认设为off
-mime_header	包含MIME头并强制源码转储

(续)

参数	描述
-minimal	使用最小注释解析而不是验证注释解析。默认设为off
-nested_tables	使用嵌套表逻辑。默认设为off
-newschunksize=n	设置分块新闻列表中的文章数目
-newsmaxchunk=n	设置分块之前列表中新闻文章数目的上限
-nobold	禁止加粗视频属性
-nobrowse	禁止目录浏览
-nocc	禁止Cc: 在给自己发邮件时提示。默认设为off
-nocolor	禁止彩色输出
-noexec	禁止本地程序执行。默认设为on
-nofilereferrer	对于文件URL, 禁止发送Referer头。默认设为on
-nolist	在转储中禁止链接列表功能。默认设为off
-nolog	禁止将错误消息发送给文档所有者。默认设为on
-nonrestarting_sigwinch	让窗口大小调整处理程序不再重启。默认设为off
-nopause	禁止强制暂停状态行消息
-noprint	禁止一些打印功能, 比如-restrictions=print。默认设为off
-noredir	禁止跟随Location:重定向。默认设为off
-noreferer	禁止发送Referer头。默认设为off
-noreverse	禁止反转视频属性
-nostatus	禁止各种信息消息。默认设为off
-nounderline	禁止给视频加下划线属性
-number_fields	强制对链接和表单输入区域进行标号。默认设为off
-number_links	强制对链接标号。默认设为off
-partial	在下载时显示部分页面。默认设为on
-partial_thres=n	设定在用部分显示逻辑重绘显示前渲染的行数。默认为-1, 表示禁用这个功能
-pauth=id:pw	为受保护的代理服务器设置认证信息
-popup	在弹出窗口而不是单选框中处理单选SELECT选项。默认设为off
-post_data	为post表单从STDIN读取数据, 以---结尾
-preparsed	和-source一起使用, 显示解析后的文本/html MIME类型, 并在源码视图中可视化显示Lynx遇到无效HTML时如何处理。默认设为off
-prettysrc	在源码视图中使用语法高亮和超链接处理。默认设为off
-print	开启打印功能, 跟-noprint相反。默认设为on
-pseudo_inlines	对没有ALT字符串的内嵌图片使用伪ALT字符串。默认设为on
-raw	为启动字符集使用默认8 bit字符转换设置或CJK模式。默认设为off

(续)

参数	描述
-realm	在开始范围内限制对URL的访问。默认设为off
-reload	清空代理服务器的缓存（只有第一个文档会受影响）。默认设为off
-restrictions=options	设置限制选项。使用不带参数的-restrictions来看参数列表
-resubmit_posts	当文档是使用PREV_DOC命令或历史列表返回时，强制用POST方法重新提交（非缓存）表单。默认设为off
-rlogin	禁止rlogin功能。默认设为off
-selective	请求www_browsable文件来浏览目录
-short_url	开启检查状态行中长URL的开头和结尾。默认设为off
-show_cursor	设为off时，将光标隐藏到右下角；否则显示光标。默认设为on
-show_rate	显示传输速率。默认设为on
-soft_dquotes	使用过去Netscape和Mosaic问题的模拟，它会把>当做双引号和标签的共同结束符。默认设为off
-source	将第一个URL的源码转储到STDOUT中并退出
-stack_dump	禁止SIGINT清理程序。默认设为off
-startfile_ok	和-validate一起允许非HTTP的起始页和主页。默认设为off
-stdin	从STDIN读取起始文件。默认设为off
-tagsoup	使用TagSoup而不是SortaSGML解析器。默认设为off
-telnet	禁止telnet会话。默认设为off
-term=term	指定要模拟的终端类型
-tlog	对当前会话使用Lynx追踪日志。默认设为on
-tna	使用“文本字段需激活”（Textfields Need Activation）模式。默认设为off
-trace	使用Lynx追踪模式。默认设为off
-trace_mask	定制Lynx追踪模式。默认设为0
-traversal	遍历起始文件中的所有HTTP链接
-trim_input_fields	缩小表单中的文本输入段。默认设为off
-underline_links	对链接使用加下划线和加粗属性。默认设为off
-underscore	对转储使用下划线格式。默认设为off
-use_mouse	开启鼠标支持。默认设为off
-useragent=Name	设置备用Lynx User-Agent头
-validate	只接受http URL（意为验证）。暗中包含比-anonymous更多的限制，但允许http和https重定向。默认设为off
-verbose	使用[LINK]、[IMAGE]和[INLINE]注释和这些图片的文件名。默认设为on
-version	显示Lynx版本信息
-vikeys	使能类vi的按键移动。默认设为off
-width=n	为格式化转储信息设置屏幕宽度。默认为80列
-with_backspaces	使用-dump或-crawl参数的话，排除输出中的退格。默认设为off

如你所见，可以直接从命令行完全控制任何类型的HTTP或HTML设置。举个例子，要想使用HTTP POST方法向Web表单发送数据，只需将数据放在-post-data参数中。要将网站接收到的cookie存储到某个特定位置，可以使用-cookie_save_file参数。

许多命令行参数定义了在全屏模式中使用Lynx时控制它的行为，允许在浏览Web页面时改变Lynx的行为。

通常在正常的浏览环境中，你会发现有几组命令行参数非常有用。你不用每次使用Lynx时都在命令行上将这些参数输入一遍，Lynx提供了一个通用配置文件来定义使用Lynx时它的基本行为。我们将在下一节中讨论这个配置文件。

24.1.3 Lynx配置文件

lynx命令会从一个配置文件中读取它的许多参数设置。默认情况下，这个文件位于/usr/local/lib/lynx.cfg，不过有许多Linux发行版将其改放到了/etc目录下（/etc/lynx.cfg；Ubuntu发行版将lynx.cfg改放到了/etc/lynx-curl目录中）。

lynx.cfg配置文件将相关的参数分组到段中，这样更容易找到参数。配置文件中条目的格式为：

PARAMETER:*value*

其中*PARAMETER*是参数的全名（通常但不总是用大写字母），*value*是跟参数关联的值。

浏览一下这个文件，你会发现许多参数都跟命令行参数类似，比如ACCEPT_ALL_COOKIES参数就等同于设置了-accept_all_cookies命令行参数。

还有一些配置参数功能类似，但名称不同。FORCE_SSL_COOKIES_SECURE配置文件参数设置可以用-force_secure命令行参数给覆盖掉。

但你还会发现有些配置参数跟命令行参数并不匹配。这些值只能在配置文件中设定。

你不能在命令行上设置的最常见的配置参数是设定代理服务器的。有些网络（尤其是公司网络）使用代理服务器作为客户端浏览器和目标网站的桥梁。不能直接向远程Web服务器发送HTTP请求，客户端浏览器必须将它们的请求发到代理服务器上。然后代理服务器会将请求发给远程Web服务器，提取结果，将结果转发回客户端浏览器。

虽然这看起来像在浪费时间，但它是保护客户端不受互联网上危险侵害的重要功能。代理服务器可以过滤不良内容和恶意代码，甚至可以发现用于互联网钓鱼计划的网站（为了获得用户数据，流氓服务器会装成别人）。代理服务器还可以帮助减少网络带宽的使用，因为它们缓存了共同浏览的Web页面并将它们返回给客户端而不用再下载一回原始页面。

用来定义代理服务器的配置参数有：

```
http_proxy:http://some.server.dom:port/
https_proxy:http://some.server.dom:port/
ftp_proxy:http://some.server.dom:port/
gopher_proxy:http://some.server.dom:port/
news_proxy:http://some.server.dom:port/
newspost_proxy:http://some.server.dom:port/
```

```

newsreply_proxy:http://some.server.dom:port/
snews_proxy:http://some.server.dom:port/
snewspost_proxy:http://some.server.dom:port/
newsreply_proxy:http://some.server.dom:port/
nntp_proxy:http://some.server.dom:port/
wais_proxy:http://some.server.dom:port/
finger_proxy:http://some.server.dom:port/
cso_proxy:http://some.server.dom:port/
no_proxy:host.domain.dom

```

你可以为任何Lynx支持的网络协议定义一个不同的代理服务器。NO_PROXY参数是逗号分隔的网站列表，你更倾向于直接访问这些网站而不用代理服务器。这些通常是不需要过滤的内部网站。

24.1.4 Lynx环境变量

从前面大量的命令行选项和配置文件参数中你可以看到，Lynx程序的可定制性极强。但定制并不局限于此。你可以用环境变量覆盖许多配置文件参数。如果你在一个不能访问lynx.cfg配置文件的环境中使用它，你可以通过设置局部环境变量来覆盖一些默认参数。表24-2列出了你在受限环境中可能会用到的较常用的Lynx环境变量。

表24-2 Lynx环境变量

变量	描述
LYNX_CFG	指定备用配置文件的位置
LYNX_LSS	指定默认Lynx字符集样式表单的位置
LYNX_SAVE_SPACE	指定用来将文件保存到硬盘的位置
NNTPSERVER	指定用来获取和发布USENET新闻的服务器
PROTOCOL_PROXY	覆盖指定协议的代理服务器
SSL_CERT_DIR	为访问受信任站点指定包含受信任证书的目录
SSL_CRET_FILE	指定含有受信任证书的文件
WWW_HOME	指定Lynx在启动时使用的默认URL

你可以在使用Lynx程序前像设置其他环境变量一样设置这些环境变量：

```
$ http_proxy=http://myproxy.com:8080
$ lynx
```

要指定代理服务器，你必须提供协议、服务器名称以及用来和服务器通信的端口。如果你需要这个变量设置，通常放在shell的常用启动文件中更好（比如bash shell的.bashrc文件），这样你就不用每次都输入了。

24.1.5 从Lynx中抓取数据

当你在shell脚本中使用Lynx时，大多数情况下你只是要提取Web页面中的某条（或某几条）特定信息。完成这个任务的方法称作屏幕抓取（screen scraping）。在屏幕抓取中，你要尝试通过

编程寻找图形化屏幕上某个特定位置的数据，这样你就可以抓取它并在脚本中使用了。

用lynx进行屏幕抓取的最简单办法是用-dump选项。这个选项不会影响在终端屏幕上显示Web页面。相反，它会直接将Web页面文本数据显示在STDOUT上：

```
$ lynx -dump http://localhost/RecipeCenter/
The Recipe Center
" Just like mom used to make"
Welcome
[1]Home
[2]Login to post
[3]Register for free login
```

```
[4]Post a new recipe
```

每个链接都由一个标号标定，Lynx在Web页面数据后显示了所有的标签选项。

一旦你获得了Web页面上的所有文本数据，你很可能已经知道我们会从工具箱中取出什么工具来提取数据。对的，我们的老朋友sed编辑器和gawk程序（参见第18章）。

首先，让我们找一些有意思的数据来收集。Yahoo!天气页面是找到全世界范围内当前天气状况的很好的源。每个位置都用一个单独的URL来显示该城市的天气信息（你可以在一般浏览器中打开该站点并输入你的城市信息来获取你所在城市的特定URL）。查看伊利诺伊州芝加哥市的天气情况的lynx命令如下：

```
lynx -dump http://weather.yahoo.com/united-states/illinois/chicago-2379574/
```

这条命令会从页面中显示出很多很多的数据。第一步是找到你要的精确信息。要做到这点，将lynx命令的输出重定向到一个文件中，然后在文件中查找数据。执行了前面的命令后，我们在输出文件中找到了这段文本：

```
Current conditions as of 1:54 pm EDT
Mostly Cloudy
```

```
Feels Like:
32 °F
```

```
Barometer:
30.13 in and rising
```

```
Humidity:
50%
```

```
Visibility:
10 mi
```

```
Dewpoint:
15 °F
```

```
Wind:
W 10 mph
```

这都是你需要的关于当前天气的所有信息。但这段输出中有个小问题。你会注意到数字都是在标题下面一行。只提取单独的数字有些困难。第18章讨论了如何处理这样的问题。

解决这个问题的关键是写一个sed脚本先查找数据标题。当你找到它后，你就可以到正确的行提取数据。在这个例子中我们比较幸运，我们需要的数据那行只有它们自己。我们应该能用sed脚本来解决这个问题。如果在同一行还有其他文本，我们需要使用gawk工具来过滤出我们需要的数据。

首先，你需要创建一个sed脚本来查找Current conditions文本，然后跳到下一行来获取描述当前天气状况的文本，最后打印出来。脚本看起来如下：

```
$ cat sedcond
/Current conditions/{
```

```
n
p
}
$
```

地址指明要先查找含有指定文本的行。如果sed命令找到它了，n命令会跳到下一行，然后p命令会打印当前行的内容，也就是描述该城市当前天气状况的文本。

下一步，你需要一段sed脚本来查找Feels Like:文本并到下一行打印出温度：

```
$ cat sedtemp
/Feels Like:/{
```

```
n
p
}
$
```

太好了。现在你可以在shell脚本中用这两个sed脚本，首先将lynx的Web页面输出抓到一个临时文件中，然后将这两个sed脚本作用到Web页面数据上来只提取你要找的数据。这里有个如何处理的例子：

```
$ cat weather
#!/bin/bash
# extract the current weather for Chicago, IL

URL="http://weather.yahoo.com/united-states/illinois/chicago-2379574/"
LYNX=`which lynx`
TMPFILE=`mktemp tmpXXXXXX`
$LYNX -dump $URL > $TMPFILE
conditions=`cat $TMPFILE | sed -n -f sedcond`
temp=`cat $TMPFILE | sed -n -f sedtemp`
rm -f $TMPFILE
echo " Current conditions: $conditions"
echo The current temp outside is: $temp
$ ./weather
Current conditions: Mostly Cloudy
The current temp outside is: 32 *F
$
```

天气脚本会连接到指定城市的Yahoo!天气Web页面，将Web页面保存到一个文件中，提取相应的文本，删除临时文件，然后显示天气信息。这么做的好处在于，一旦你从网站上提取到了数据，你可以随便怎么处理它，例如创建一个温度表。然后你可以创建一个每天运行的cron任务（参见第15章）来跟踪每天温度。

警告 互联网是动态地域。如果你花费了几个小时找到了Web页面上数据的精确位置，而几个星期后就发现它已经转移了，脚本没法工作了，不必感到惊讶。事实上，很有可能上面这个例子在你阅读本书时已经无法工作。重要的是要知道从Web页面提取数据的过程。然后你可以将这一原则应用到任何情形中。

24.2 cURL 程序

Lynx的流行催生了另一个类似的产品，称作cURL。cURL程序允许使用特定URL来自动从命令行传送文件。现在它支持FTP、FTPS、HTTP、HTTPS、SCP、SFTP、TFTP、telnet、DICT、LDAP、LDAPS以及FILE协议作为URL中指定的协议。

虽然cURL本身并不用作Web页面浏览器，但它允许直接从命令行或shell脚本方便地发送或提取数据，不需要人值守，只用一条简单命令就行。这又为你的shell脚本编程工具箱提供了一个很好的工具。

本节将会带你逐步了解在shell脚本中安装和使用cURL的过程。

警告 有一门编程语言也叫curl，由Sumisho计算机系统公司拥有并销售。不要将cURL和curl编程语言混为一谈。

24.2.1 安装cURL

24

由于越来越流行，cURL已经在许多Linux发行版中默认安装。在Ubuntu中，你必须手动从远程软件仓库中安装。使用下面的命令：

```
$ sudo apt-get install curl
```

如果你的Linux发行版中没有cURL，或者你要使用最新版，你可以从curl.haxx.se网站下载源代码并在Linux系统上编译。

还有，标准的免责声明又来了，你必须在Linux系统上安装了C开发库。

下一节将会介绍如何在命令行上使用cURL。

24.2.2 探索cURL

默认情况下，cURL会将完整的Web页面HTML代码返回到STDOUT上：

```
$ curl http://www.google.com
<!doctype html><html><head><meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" ><title>Google</title>
[ listing truncated ]
```

跟Lynx程序一样，你可以用标准的shell脚本技术来从转储的Web页面中提取单个数据元素。

我们喜欢用cURL来批量下载文件。现在，看起来我们总是在下载最新的Linux发行版ISO文件。由于ISO文件太大了，我们需要开始下载，然后在它下载时离开一会儿。一旦我们知道了ISO文件的URL，我们就可以创建一个简单的shell脚本，用cURL来自动化这个过程：

```
$ cat downmid
#!/bin/bash
# download latest cURL file automatically
curl -s -o /home/rich(curl-7.18.0.tar.gz
http://curl.haxx.se/download/curl-7.18.0.tar.gz
$
```

-s命令行选项让cURL工作在安静模式下，不会发送任何数据到STDOUT。-o命令行选项会将输出重定向到一个文件名。虽然从这段代码中很难看出来，但curl命令在脚本中是在一行上的。这个简单的脚本运行后会直接从cURL官方网站下载文件。现在你可以用at或cron命令（参见第15章）来将这个下载任务安排在晚上不用电脑或网络的时候进行。

24.3 使用zsh处理网络

第22章介绍了zsh shell中的所有功能。zsh shell是Linux和Unix环境中较新的一个shell。zsh shell的一个功能是插件模块。不用将大量功能都放到核心zsh shell，zsh shell使用专门的模块来提供这些功能，所以你可以选择加载你需要的命令。其中一个模块是TCP模块。

zsh shell中的TCP模块提供了一些非常强大的功能，可以直接在命令行上进行网络连接。你可以直接在命令行上（或脚本中）创建和另一个网络设备的完整TCP会话。本节将会讨论zsh TCP模块的功能，并演示一个可以用zsh shell脚本构建的简单客户端/服务器（C/S）应用。

24.3.1 TCP模块

zsh shell使用模块来向核心zsh shell提供额外的功能。每个模块都包含特定领域的内建命令。TCP模块为很多网络功能提供了内建命令。

在zsh shell中安装TCP模块，可以做如下操作：

```
% zmodload zsh/net/tcp
%
```

在shell中安装模块库就是这么简单。如果你在shell脚本中使用TCP模块，记住将这行加到脚本中。这个模块只会在当前shell上有效。

一旦加载了TCP模块，你就可以使用ztcp命令了。ztcp命令的格式如下：

```
ztcp [-acfIltv] [-d fd] [args]
```

有以下几个可用的命令行选项。

- a: 接受一个新连接。
- c: 关闭一个已有连接。
- d: 对连接使用指定的文件描述符。
- f: 强制关闭连接。
- l: 打开新的监听套接字。
- L: 列出当前已连接的套接字。
- t: 如果没有连接在等待, 退出。
- v: 显示连接的详细信息。

ztcp命令使用文件描述符来和一个打开的TCP连接交互。默认情况下, zsh会用环境变量\$REPLY来引用该文件描述符。你要做的只是将数据发送给\$REPLY变量中指定的文件描述符, 而TCP模块会将它转发给远程主机。类似地, 如果远程主机给你发送了一些数据, 你要做的只是从\$REPLY变量指定的文件描述符中读取数据。用它进行网络编程太简单了。

24.3.2 客户端/服务器模式

在进一步了解如何用zsh shell创建客户端/服务器(C/S)应用之前, 最好先了解一下客户端和服务器端程序是如何运行的。显然, 它们在网络连接和数据传送方面各自有不同的职能。

服务器程序会监听网络, 等待来自客户端的请求。客户端则会向服务器发起一个连接请求。一旦服务器接受了连接请求, 双方就建立起了一条双向信道来发送和接收数据。这个过程如图24-2所示。

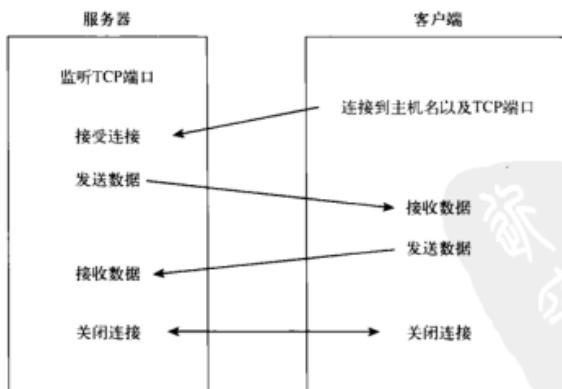


图24-2 C/S通信模式

如你在图24-2中看到的, 服务器必须做两个工作才能和客户端通信。首先, 它必须建立一个特定的TCP端口来监听发过来的请求。当连接请求过来时, 它必须接受连接。

客户端的职能就简单多了，它需要做的只是在服务器监听的特定TCP端口尝试连接。如果服务器接受了连接，双向通信就建立起来，可以发送数据了。

一旦服务器和客户端的连接建立了，两个设备之间就必须有一些通信过程（或规则）。如果两个设备试图同时监听一条消息，它们就会形成死锁，什么也做不了。类似地，如果它们试图同时发送消息，还是什么也做不了。

作为网络程序员，你的任务就是决定客户端程序和服务器程序必须遵循的协议规则。

24.3.3 使用zsh进行C/S编程

为了演示用ztcp创建C/S程序，让我们建立一个简单的网络应用。我们要创建的服务器程序会在TCP端口5150监听连接请求。当连接请求过来时，服务器会接受请求然后向客户端发送一条欢迎消息。

然后服务器程序会等待接收客户端消息。如果收到消息，服务器会显示这条消息，然后向客户端发送同一条消息。在发送这条消息后，服务器会继续监听等待另一条消息。这个循环会一直延续下去，直到服务器收到一条文本消息exit。这时，服务器会终止这个会话。

我们要创建的客户端程序会向服务器TCP端口5150发送一个连接请求。当连接建立时，客户端需要收到服务器的欢迎消息。

在收到这条消息后，客户端会显示它，然后会向用户询问要发送给服务器的数据。在得到用户的消息后，客户端程序会将它发送给服务器，并等待接收回过来的消息。如果消息回过来了，客户端会显示这条消息，然后返回来向用户请求下一条消息。这个循环会一直延续下去，直到用户输入了文本exit。这时，客户端会将exit文本发送给服务器然后终止会话。

后面几节将会介绍服务器和客户端程序。

1. 服务器程序

下面是服务器程序的代码：

```
% cat server
#!/bin/zsh
# zsh TCP server script
zmodload zsh/net/tcp
ztcp -l 5150
fd=$REPLY

echo "Waiting for a client..."
ztcp -a $fd
clientfd=$REPLY
echo "client connected"

echo "Welcome to my server" >& $clientfd

while [ 1 ]
do
    read line <& $clientfd
    if [[ $line = "exit" ]]

```

```

then
    break
else
    echo Received: $line
    echo $line >& $clientfd
fi
done
echo " Client disconnected session"
ztcp -c $fd
ztcp -c $clientfd
%
```

服务器程序遵循图24-2中演示的C/S范式。它会先用-t参数来在指定的端口(5150)上监听。\$REPLY变量含有Linux系统返回的用来标识该连接的文件描述符。服务器用-a参数来接受新的网络请求。这个命令会一直等待直到有新的连接请求进来(称为阻塞,blocking)。而这个脚本不会进展直到有新的连接请求被接受。

每个客户端连接都用一个单独的文件描述符以和监听端口文件描述符区分开来。这允许你同时维护多个客户端连接,如果你有这种需要的话(在这个简单练习中没这个需求)。

在接受了连接后,服务器会将欢迎消息重定向到客户端文件描述符上:

```
echo "Welcome to my server" >& $clientfd
```

zsh shell的TCP模块会处理保证数据发送到远程客户端的所有技术细节。

下一步,服务器程序进入了一个无穷循环中,它用read命令来等待数据从客户端回来:

```
read line <& $clientfd
```

这个命令会阻断脚本的执行,直到它收到了客户端过来的数据。如果客户端断了网络连接的话,这会是个麻烦。为了阻止这种情况发生,你可以在read一行使用-t选项来指定一个超时值(以秒为单位)。如果服务器没有在限定时间内收到从客户端过来的数据,它会继续执行。

如果服务器收到了从客户端过来的数据,它会将数据显示在STDOUT上,然后将它发送回客户端。如果数据等于文本字符串exit,服务器会退出循环,并用ztcp中的-c参数来关闭客户端文件描述符和监听端口的文件描述符。如果你想让服务器监听另一个连接,你可以在关闭客户端文件描述符后返回循环等待接受新的连接。

2. 客户端程序

下面是客户端shell脚本程序的代码:

```

% cat client
#!/bin/zsh
# zsh TCP client program
zmodload zsh/net/tcp

ztcp localhost 5150
hostfd=$REPLY

read line <& $hostfd
echo $line
```

```

while [ 1 ]
do
    echo -n " Enter text: "
    read phrase
    echo Sending $phrase to remote host...
    echo $phrase >& $hostfd
    if [[ $phrase = " exit" ]]
    then
        break
    fi
    read line <& $hostfd
    echo "     Received: $line"
done
ztcp -c $hostfd
%
```

客户端程序必须指定运行服务器程序的系统的IP地址（或主机名）以及服务器所监听的正确的TCP端口号。服务器接受连接后，ztcp程序设置该连接的文件描述符，并保存\$REPLY变量中的值。客户端程序会读取服务器的欢迎消息然后显示它：

```

read line <& $hostfd
echo $line
```

下一步，客户端程序会进入一个while循环中，向用户询问要发送给服务器的文本，读取输入文本并将文本发给服务器。在发出文本后，它会查看输入的文本是不是exit。如果是，它会跳出循环并关闭文件描述符，也就是关掉TCP连接，如果不是，它会继续等待服务器的回应并显示它。

3. 运行程序

可以在网络中的两个不同Linux系统上或同一个系统中的两个不同终端会话中运行这两个程序。你必须先启动服务器程序，这样它就可以监听客户端启动时发出的连接请求：

```

% ./server
Waiting for a client...
```

然后你启动客户端程序：

```

% ./client
Welcome to my server
Enter text: test
Sending test to remote host...
Received: test
```

当客户端连接时，你会在服务器上看到下面的内容：

```

client connected
Received: test
```

这个过程会一直继续，直到用户在客户端上输入了文本exit：

```

Enter text: exit
Sending exit to remote host...
%
```

然后你会看到服务器程序自动退出了：

```
Client disconnected session  
%
```

现在你已经有了一个能全面工作的网络程序的开头了。有了zsh shell和TCP模块，在脚本之间发送数据就容易多了，这些脚本运行在同一网络的不同系统上。

24.4 小结

本章带你逐步了解了用shell脚本同互联网交互的过程。最流行的工具之一就是Lynx程序。Lynx是一个可以在终端会话中用文本图形显示网站信息的命令行程序。除了这个功能外，Lynx还提供了从网站上提取原始数据并将其输出到STDOUT的方法。你可以用Lynx来从网站上提取信息，然后用标准的Linux文本处理工具（如sed和gawk）来解析数据，查找特定信息。

cURL程序是和互联交互时的另一个方便的工具。cURL程序允许转储网站上的数据，并且提供了一种方法来编程从许多不同类型的服务器上下载文件。

最后，本章介绍了如何用zsh shell的TCP模块来编写网络程序。zsh shell提供了一个在网络上不同系统中的shell脚本间进行通信的简单途径。

下一章，我们将会学习如何在shell脚本中使用E-mail。通常在你使用shell脚本自动化一些过程时，最好能收到一条消息说明这个过程失败了还是完成了。了解了如何使用安装在你系统上的E-mail软件后，你可以轻松地向世界上的任何人发送自动消息。

本章内容

- E-mail和Linux
- 建立E-mail服务器
- 发送简单的消息
- 在Mutt程序中使用附件

随着E-mail的普及，现在几乎每一个人都有一个E-mail地址。正因如此，人们通常更期望通过邮件接收数据而不是看文件或打印出的资料。在shell脚本编程中也是如此。如果你从shell脚本生成了任何类型的报告，大多数情况下你会被要求用E-mail将结果发送给某个人。本章将介绍如何设置Linux系统来支持直接从shell脚本中发送E-mail。还介绍了如何保证Linux系统可以发送邮件消息，以及如何保证你有一个可以从命令行上发送邮件的邮件客户端。但首先，本章会先概述一下Linux处理E-mail的方式。

25.1 Linux E-mail 基础

有时在shell脚本中使用E-mail最难的部分是理解E-mail系统在Linux上如何工作。了解什么软件包能完成某个特定任务对于在shell脚本中将E-mail发到收件箱尤其重要。本节将会带你逐步了解Linux系统如何使用E-mail的基础知识以及在使用前的准备工作。

25.1.1 Linux中的E-mail

Unix操作系统的主要目标之一就是将软件模块化。Unix开发人员开发一些较小的程序，其中每个程序负责完整整个系统功能中的一小部分，而不是开发一个大程序来处理完成该功能需要的所有任务。

这种思想在实现Unix上的E-mail系统时被采用，并被带到了Linux环境中。在Linux中，E-mail功能被分成几部分，每部分又被分给不同的程序。图25-1演示了Linux环境中的大多数开源E-mail软件如何模块化E-mail功能。

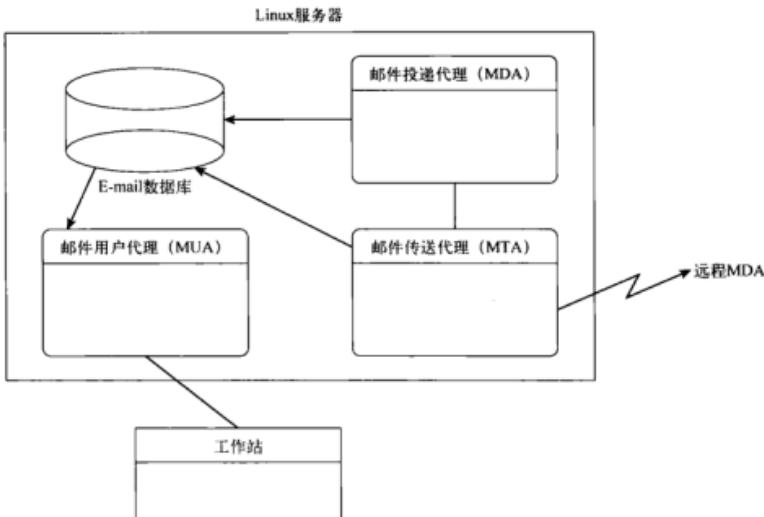


图25-1 Linux模块化E-mail环境

如你在图25-1中看到的，在Linux环境中E-mail过程通常分为3个功能：

- 邮件传送代理（Mail Transfer Agent, MTA）；
- 邮件投递代理（Mail Delivery Agent, MDA）；
- 邮件用户代理（Mail User Agent, MUA）。

这3种功能之间的界限非常模糊。有些E-mail软件包合并了MDA和MTA的功能，而有些则合并了MDA和MUA的功能。下面几节将会详细介绍这些基本的E-mail模块以及它们在Linux系统中如何实现。

25.1.2 邮件传送代理

MTA软件是Linux E-mail系统的核心。它负责处理系统中邮件的收发。对于每个发出的邮件消息，MTA必须确定收件人地址的目的地。如果目的地主机是本地系统，MTA会将它直接发送到本地邮箱或将消息传给本地MDA来投递。

但如果目的地主机是远程邮件服务器，MTA必须和远程主机上的MTA软件建立起一个通信连接来传送该消息。MTA软件常用两种方法来将邮件递送到远程主机上：

- 直接递送（direct delivery）；
- 代理递送（proxy delivery）。

如果你的Linux系统直接连到了互联网上，通常它可以直接将发往远程主机上收件人的消息直接投递到远程主机上。MTA软件用域名系统（Domain Name System, DNS）解析正确的网络IP地址来投递邮件消息，然后用简单邮件传输协议（Simple Mail Transfer Protocol, SMTP）来建立起网络连接。

很多时候主机并不直接连到互联网上，或者它不想直接跟其他远程主机通信。这种情况下，它通常会使用一个前端主机（smart host）。前端主机是一个代理服务器，它会接收来自Linux系统的邮件消息，然后尝试直接将它们投递到目标收件人。

说明 由于垃圾邮件，前端主机变得越来越难在互联网上工作了。流氓服务器可以通过前端主机转发数千封不请自来的商业邮件（Unsolicited Commercial E-mail, UCE）消息来发送垃圾邮件，还可以隐藏身份。现在大多数前端主机在向其他主机转发消息前都会要求某种形式的认证。

对于收到的邮件，MTA必须能够接受来自远程邮件服务器的连接请求，接收发往本地用户的消息。这个过程最常用的协议依然是SMTP。

Linux环境有许多不同类型的开源MTA程序。每个程序都能提供不同的特性来和其他程序区分开来。到目前为止，你可能遇到的最流行的两个是：

- sendmail；
- Postfix。

我们会在25.2节中详细介绍这两个E-mail MTA软件。

25.1.3 邮件投递代理

MDA程序的职能是投递发往本地用户的消息。它会接收来自MTA程序的消息，并且必须确定这些邮件如何投递以及投递到哪里。图25-2演示了MDA程序如何与MTA程序交互来投递邮件。

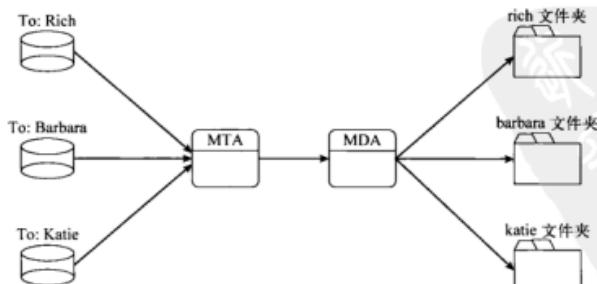


图25-2 在邮件服务器上使用MDA程序

虽然有时MDA的功能会由MTA程序自己执行，但通常Linux E-mail的实现会依赖一个独立的MDA程序来将消息投递到本地用户。由于这些MDA程序只关注将邮件投递到本地用户，它们可以添加一些额外的铃声或哨声，这些在附带MDA功能的MTA程序中是没有的。这使得邮件管理员可以提供额外的邮件功能给邮件用户，比如垃圾邮件过滤、休假转发以及邮件自动排序等。

当MDA程序收到了一条消息后，它必须确保消息投递到了正确的位置，要么在本地用户的收件箱中，要么在本地用户定义的备用位置。

目前在Linux系统上有3种常用的不同类型用户邮箱：

- /var/spool/mail或/var/mail文件；
- \$HOME/mail文件；
- Maildir风格的邮箱目录。

每种邮箱类型都有自己的具有吸引力的特性。大多数Linux发行版都使用/var/spool/mail或/var/mail目录来包含单独的邮箱文件，每个文件对应系统上的一个用户账户。这是所有邮箱文件的集中位置，所以MUA程序知道到哪里去查找用户的邮箱文件。

一些Linux发行版允许你将单独的邮箱文件移动到每个用户的\$HOME目录。由于每个邮箱文件所处的位置已经设置了正确的访问权限，这样安全性就更强了。

Maildir风格邮箱是相对较新的一个功能，由一些高级MTA、MDA和MUA应用支持。邮箱是一个目录，每条消息是该目录中的一个单独文件，而不用将每个消息作为邮箱文件的一部分。它有助于降低邮箱崩溃的风险，因为单个消息不会破坏整个邮箱。

虽然Maildir风格的邮箱目录提供了更好的性能、安全性以及容错性，有许多流行的MDA和MUA程序却不能使用它们。几乎所有的MDA和MUA程序都能使用/var/spool/mail邮箱文件。

说明 原始的Unix邮箱位置是/var/spool/mail。大多数Linux发行版沿用了这个文件命名习惯，但有一些Linux发行版使用的是/var/mail。

如果你的系统使用特殊的MDA程序来处理收到的邮件消息，大多数情况下它会是流行的Procmail程序。Procmail允许每个单独用户创建定制的配置文件来定义邮件过滤器、休假转发回复和单独的邮箱。

25.1.4 邮件用户代理

到目前为止，我们已经跟着E-mail的传送从远程主机到了本地主机，现在到了每个用户的邮箱。下一步就是允许用户查看他们的E-mail消息。

Linux E-mail模型使用本地邮箱文件或目录来为每个用户保存消息。MUA程序的工作是为用户提供一种方法来和他们的邮箱交互、读取他们的消息。

重要的是要记住MUA不接收消息，它们只显示已经在邮箱中的消息。许多MUA程序还提供了创建单独邮件文件夹的能力，这样用户可以将邮件从默认邮箱（通常称为收件箱，Inbox）移

动到单独的文件夹。

许多MUA程序还提供了发送邮件的能力。这部分有点叫人困惑，因为如你在前面看到的，发送E-mail消息是MTA程序的工作。要执行这个功能，大多数MUA程序利用SMTP的前端主机功能。要么MUA程序自动将消息递送到本地MTA程序来进一步递送，要么你在MUA配置中定义一个远程前端主机来发送消息以进一步递送。

这些年来，Linux平台已经有许多不同的开源MUA程序可用。下面几节将会介绍你可能会碰到的其中一些较流行的MUA程序。

1. Mailx

Mailx程序是Linux环境中在用的最流行的命令行MUA程序。在所有的安装中，Mailx程序都会安装可执行文件mail，表明它是mail程序的一个替代，而不是另一个程序。

说明 基于图形化桌面的Linux发行版，比如Ubuntu或openSUSE，默认不会安装命令行Mailx程序。你必须手动将它作为邮件客户端包的一部分安装（参见第8章）。但你必须小心，因为不是所有的Linux邮件包都一样。对于Ubuntu，这个包叫做mailutils，它使用GNU Mailutils包而不是Mailx。它提供了同样的功能，但是命令行参数略有不同。

Mailx程序允许用户完全在命令行上访问他们的邮箱，来读取已存储的消息以及将消息发送给其他邮件用户。这里有个Mailx会话的例子：

```
$ mail
"/var/mail/rich" : 2 messages 2 new
>N 1 Rich Blum      Thu Dec  9 10:07 13/579  Test message
N 2 Rich Blum      Thu Dec  9 10:08 13/593  This is another test
? 1
Return-Path: <rich@rich-Parallels-Virtual-Platform>
X-Original-To: rich@rich-Parallels-Virtual-Platform
Delivered-To: rich@rich-Parallels-Virtual-Platform
Received: by rich-Parallels-Virtual-Platform (Postfix, from userid 1000)
          id 5C03F2606CF; Thu,  9 Dec 2010 10:07:48 -0500 (EST)
To: <rich@rich-Parallels-Virtual-Platform>
Subject: Test message
X-Mailer: mail (GNU Mailutils 2.1)
Message-Id: <20101209150748.5C03F2606CF@rich-Parallels-Virtual-Platform>
Date: Thu,  9 Dec 2010 10:07:48 -0500 (EST)
From: rich@rich-Parallels-Virtual-Platform (Rich Blum)

This is a test message
? d
? q
Held 1 message in /var/mail/rich
$
```

第一行说明Mailx程序执行时没加命令行选项。默认情况下，这允许用户在邮箱中检查邮件。在输入mail命令后，显示了用户邮箱中所有消息的摘要。Mailx程序只能读取/var/mail格式或

\$HOME/mail格式的消息。它不能处理采用Maildir邮件目录格式的邮件。

每个用户都有一个含有他所有消息的单独文件。对于一些Linux发行版，邮箱文件一开始并不存在，直到用户接收到消息才生成。文件名通常是用户的系统登录名，位于系统邮箱目录。因此，用户名为rich的所有消息都存储在Linux系统上的文件/var/mail/rich中。当接收到该用户的新邮件时，它们会被附加在文件的末尾。

你还可以用mail命令行程序来发送邮件消息：

```
$ mail barbara
Cc:
Subject: This is a test sent to Barbara
Hello Barbara -
This is a test message I'm sending from the command line.
$
```

收件人的名称和程序名称一起包含在了命令行上。Mailx程序会询问发送副本的地址（“CC:”提示符）和消息标题。然后它允许你键入消息文本。要结束消息，按下Ctrl+D组合键。然后Mailx程序会尝试将消息传给MTA程序来递送。

2. Mutt

随着Unix环境的进步，MUA程序变得越来越漂亮。Unix系统上图形界面的第一次尝试就是ncurses图形库。使用ncurses，程序可以在终端屏幕中操作光标位置，在终端上几乎任何地方放置字符。

有个MUA程序利用了ncurses库的优点，它就是Mutt程序。当你开启Mutt时，它会在终端显示上绘制一个用户友好的菜单，像Mailx程序的输出一样列出消息。你可以选择一条消息并在显示中查看它，如图25-3所示。



图25-3 Mutt程序

Mutt程序是用组合键来执行特定功能，比如读取消息和创建新消息。对shell脚本程序员来说可能最有用的功能就是直接在命令行上发送消息，而不用进入文本图形模式。我们会在25.4节中进一步介绍Mutt程序。

3. 图形化E-mail客户端端

几乎所有的Linux系统都支持图形化X Window环境。有许多E-mail MUA程序利用X Window系统来显示消息信息。现有的最流行的两个图形化MUA程序是：

- KDE窗口环境中的KMail；
- GNOME窗口环境中的Evolution。

其中每个程序都允许你和本地的Linux邮箱交互，以及连接到远程邮件服务器来读取邮件消息。图25-4演示了一个Evolution会话屏幕的例子。

为了和远程服务器连接，KMail和Evolution都支持POP协议（Post Office Protocol，邮局协议）和更先进的IMAP协议（Internet Message Access Protocol，互联网消息访问协议）。虽然KMail和Evolution MUA程序在桌面Linux中很有用，但它们在shell脚本编程中用处不大。

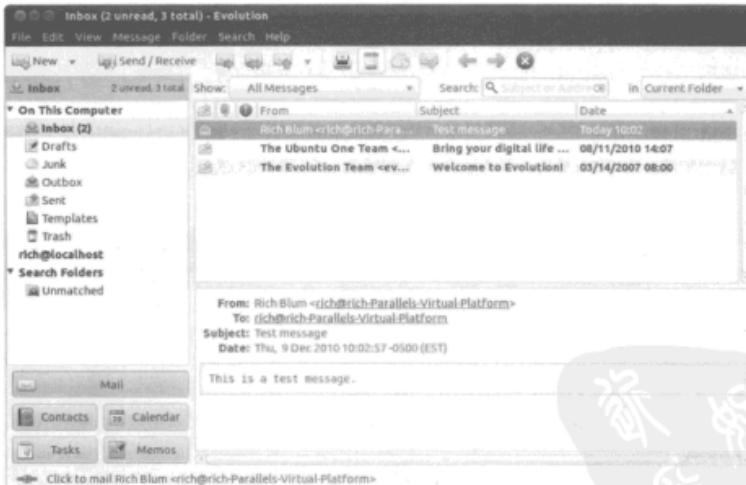


图25-4 Evolution MUA程序主界面

25.2 建立服务器

在开始向全世界发送自动化E-mail消息之前，你需要保证你的Linux系统有个正确配置的MTA

程序在运行。这本身绝非小事，不过幸运的是，有些Linux发行版提供了若干基本工具来帮你。

本节将会带你逐渐了解Linux中最流行的两个E-mail MTA程序——sendmail和Postfix的基本知识。虽然已经有完整的书介绍怎样正确配置其中每一个程序，但我们会了解一下基本知识，看看如何在Linux系统上获取邮件并将其放到收件箱中。

25.2.1 sendmail

sendmail MTA程序是Internet邮件服务器使用的最流行的开源MTA程序之一。过去，它备受后门和安全漏洞事件的困扰；但它已经被重写，不仅修复了安全漏洞，而且还集成了很多新的MTA功能，比如垃圾邮件控制。新版本的sendmail程序已经被证明很安全而且很强大。

1. sendmail程序的组件

主要的可执行程序称为sendmail。它通常运行在后台模式，监听来自远程邮件服务器的SMTP连接并转发来自本地用户的邮件。

除了主要的sendmail程序，它还有一个配置文件和几个表，用来保存一些信息以在处理接收和发出的邮件消息时使用。表25-1列出了正常的sendmail安装中用到的所有组件。

表25-1 sendmail配置文件

文 件	描 述
sendmail.cf	控制sendmail程序行为的文本文件
sendmail.cw	含有sendmail程序接收消息用的域名列表的文本文件
sendmail.ct	含有可以控制sendmail运行的受信任用户列表的文本文件
aliases	含有可以将邮件重定向到另一个用户、文件或程序的有效本地邮件地址列表
newaliases	从文本文件创建新aliases数据库文件的可执行程序
mailq	检查邮件队列并打印任何消息的可执行程序
mqueue	用来存储待投递消息的目录
mailertable	用来指定特定域的路由路径的文本文件
domainable	用来将旧域名映射到新域名的文本文件
virtusertable	用来将用户和域映射到备用地址的文本文件
relay-domains	用来列出可以通过sendmail程序转发消息的特定主机的文本文件
access	列出了特定域的文本文件，来自这些域的消息被允许或禁止

除非你正在为公司或互联网服务提供商运行主邮件服务器，你只用关心sendmail.cf配置文件。事实上，许多带sendmail的Linux发行版会自动为你创建和配置一份核心sendmail.cf配置文件，该配置文件应该能在大多简单应用中工作起来。

2. sendmail.cf文件

你需要告知sendmail程序在服务器收到消息时如何处理它们。作为MTA，sendmail会处理收到的邮件并将它转给另外一个邮件程序，不管是远程系统上的还是本地系统上的。这个配置文件用来告诉sendmail如何处理目标邮件地址来确定如何转发这些消息以及转发到哪里。这个配置文

件的默认位置是/etc/mail/sendmail.cf。

sendmail.cf文件由规则组构成，这些规则组解析收到的邮件消息并决定执行什么操作。每个规则组都用来识别特定的邮件格式并告诉sendmail如何处理那条消息。

在sendmail程序收到消息时，它会解析消息的头并将消息传给各种规则组来决定怎么处理该消息。sendmail配置文件包含了允许sendmail以多种方式处理邮件的规则。从SMTP主机收到的邮件和来自本地的邮件有不同的头字段（header field）。sendmail程序必须知道如何处理任何邮件形式。

规则还可以有在配置文件中定义的帮助程序。你可以定义3种不同类型的帮助程序。

类： 定义一些公用的短语来帮助规则组识别特定类型的消息。

宏： 设定一些值来简化在配置文件中输入长字符串。

选项： 设置参数来控制sendmail程序的运行。

配置文件由一系列的类、宏、选项和规则组构成。每个功能都在配置文件中以单个文本行的形式定义。

配置文件中的每行都以定义该行命令的单个字母开头。以空格或制表符开头的行是前一命令行的延续。以井号（#）开头的行表明是注释，不会被sendmail处理。

文本行开头的命令定义了该行是用来做什么的。表25-2列出了标准sendmail命令以及它们代表的意义。

表25-2 sendmail配置文件命令

配置命令	描述
C	定义文本的类
D	定义宏
F	定义含有文本的类的文件
H	定义头字段和命令
K	定义含有要查找的文本的数据库
M	定义邮件传送代理
O	定义sendmail选项
P	定义sendmail的优先级值
R	定义解析地址的规则组
S	定义规则组的集

如之前提到的，你很可能不需要从头开始编写sendmail.cf配置文件。Linux发行版会为你创建一份标准模板文件。大多数情况下，你需要担心的只有你是否必须使用前端主机来为你转发邮件。DS配置行用来控制这个功能：

DSmyisp.com

只要将前端主机的主机名加在DS标签的后面即可（没有空格）。

25.2.2 Postfix

Postfix软件包很快就会成为Unix和Linux系统上比较流行的E-mail程序之一。Postfix由Wietse Venema开发，为Unix类型的服务器提供一个备用MTA。Postfix软件能够让Unix或Linux系统变成一个全功能的E-mail服务器。

MTA的职能是管理进入或离开邮件服务器的消息。Postfix使用几种不同的模块化程序和邮件队列目录系统来完成邮件跟踪。每个程序通过各种消息队列来处理消息，直到它们都被递送到了最终目的地。如果任何时间邮件服务器在消息传送过程中崩溃了，Postfix可以确定该消息上次成功放置在哪个队列，并尝试继续处理该消息。

1. Postfix系统的组件

Postfix系统由若干邮件队列目录和可执行程序组成，它们会彼此交互以提供邮件服务。

Postfix程序使用了一个一直作为后台进程运行的主程序。主程序允许Postfix启动一些程序来扫描邮件队列以查看新消息并将它们发送到正确的目的地。

主程序会使用其他帮助程序，这些帮助程序会根据它们的功能按需启动。可以将它们配置成在使用后仍运行一定时间。这允许主程序在必要时再次利用运行中的帮助程序，从而节省处理时间。在设定时间限制过后，帮助程序会默默地停止运行。

主程序用来控制Postfix的整体运行。表25-3列出了Postfix用来传送邮件消息的帮助程序。

表25-3 Postfix帮助程序

程 序	描 述
bounce	为退回的消息在退回消息队列发一条日志，并将退回的消息发送回发送者
cleanup	处理收到的邮件头并将消息放到收件队列中
error	处理来自qmqr的消息递送请求，强制消息退回
flush	处理等待被远程邮件服务器提取的消息
local	投递发往本地用户的消息
pickup	等待maildrop队列中的消息，并将它们发送到清理程序以开始处理
pipe	将来自队列管理器程序的消息转发到外部程序
postdrop	当普通用户对此队列没有写权限时，将接收的消息移动到maildrop队列
qmqr	处理接收队列中的消息，确定它们应该递送到哪里以及何时递送，在递送时启动程序
sendmail	为程序提供一个同sendmail兼容的接口来将消息发送到maildrop队列
showq	报告Postfix邮件队列的状态
smtp	使用SMTP协议将消息转发到外部邮件主机
smtpd	使用SMTP协议接收来自外部邮件主机的消息
trivial-rewrite	接收来自清理程序的消息，为qmqr程序保证头地址符合标准格式，被qmqr程序用来解析远程主机地址

在处理消息时，Postfix使用几种不同的消息队列来管理E-mail消息。每个消息队列都包含一些处于Postfix系统中不同消息状态的消息。表25-4列出了Postfix使用的消息队列。

表25-4 Postfix消息队列

队列	描述
maildrop	接收自本地用户的待处理的新消息
incoming	接收自远程主机的待处理的新消息以及来自本地用户的处理过的消息
active	准备好被Postfix递送的消息
deferred	首次递送失败、等待第二次递送的消息
flush	发往远程主机的消息，远程主机将连接邮件服务器来获取它们
mail	保存的已递送消息，供本地用户阅读

如果Postfix系统在任何时间要被关闭，消息会仍然保存在最后一次放置它们的队列中。在Postfix重启后，它会自动开始处理来自这些队列的消息。这是Postfix的一个非常好的功能，使它成为最健壮的E-mail服务器系统之一。

2. Postfix配置文件

Postfix系统的一个重要组件是配置文件。Postfix使用3个独立的配置文件来允许你设定用来指导Postfix如何处理消息的参数。跟其他一些MTA程序不同，它可以在Postfix服务器运行时修改配置信息，然后运行一个命令来让Postfix加载新的配置而不用完全停掉邮件服务器。

这3个配置文件通常存储在公共Postfix目录。该目录的默认位置一般位于/etc/postfix。通常，所有用户都有查看这些配置文件的权限，而只有root用户才有修改文件中值的权限。当然，你可以根据自己的安全情况修改权限。表25-5列出了Postfix的配置文件。

表25-5 Postfix配置文件

文件	描述
install.cf	含有安装Postfix时使用的安装参数信息
main.cf	含有Postfix程序在处理消息时使用的参数
master.cf	含有Postfix主程序在运行核心程序时使用的参数

install.cf配置文件允许你提取Postfix软件首次安装到系统上时使用的安装参数。这为确定哪些功能在安装软件时有还是没有提供了一个简单途径。

master.cf配置文件会控制核心Postfix程序的行为。每个程序以及控制它运行的参数会列在一个单独的行上。这里有个使用默认设置的master.cf文件的例子：

```
# -----
#service type private unpriv chroot wakeup maxproc command + args
#           (yes)   (yes)   (yes)   (never) (50)
#
smtp    inet  n  -  n  -  -  smtpd
pickup  fifo  n  -  n  60  1  pickup
cleanup  unix  -  -  n  -  0  cleanup
qmgr    fifo  n  -  n  300  1  qmgr
rewrite  unix  -  -  n  -  -  trivial-rewrite
bounce   unix  -  -  n  -  0  bounce
defer    unix  -  -  n  -  0  bounce
```

```
trac      unix  -   n  -  0  bounce
verify    unix  -   n  -  1  verify
flush     unix  n   n  1000 0  flush
proxymap  unix  -   n  -  -  proxymap
smtp      unix  -   n  -  -  smtp
relay     unix  -   n  -  -  smtp -o fallback_relay=
showq    unix  n   n  -  -  showq
error     unix  -   n  -  -  error
local     unix  -   n  n  -  -  local
virtual   unix  -   n  n  -  -  virtual
lmtmp    unix  -   n  -  -  ltmp
anvil    unix  -   n  -  1  anvil
scache    unix  -   n  -  1  scache
```

master.cf配置文件还含有一些用来指导Postfix怎样同外部MDA软件(比如Procmail)相接的行。Postfix运行参数是在main.cf配置文件中设置的。所有的Postfix运行参数都有一个Postfix系统中隐含的默认值。如果有参数值没有出现在main.cf文件中，它的值会由Postfix预设。如果参数值在main.cf文件中，那它的内容会覆盖掉默认值。

在配置文件中每个Postfix参数和它的值都列在单独的一行中，用下面的格式：

```
parameter = value
```

parameter和value两个都是纯文本字符串，这样在需要时可以方便地读取和修改。在Postfix首次启动以及每次调用命令重新加载配置文件时，Postfix主程序都会读取main.cf文件中的参数值。

Postfix参数的两个例子是myhostname和mydomain参数。如果未在main.cf配置文件中指定它们，那么myhostname参数会用Linux系统上gethostname()命令的结果，而mydomain则会用默认的myhostname参数的域部分。通常单个邮件服务器会处理整个域的邮件。在Postfix配置文件中，这是一个很简单的设置：

```
myhostname = mailserver.smallorg.org
mydomain = smallorg.org
```

在Postfix启动时，它会将本地邮件服务器识别为mailserver.smallorg.org，将本地域识别为smallorg.org，并忽略任何系统设定的值。

如果你需要指定一个前端主机，可以用relayhost参数：

```
relayhost = myisp.com
```

你还可以在这里指定一个IP地址，但必须用方括号将其括起来。

25.3 使用 Mailx 发送消息

现有的可用来从shell脚本中发送E-mail消息的主要工具是Mailx程序。你不仅可以用它来交互地读取和发送消息，还可以用命令行参数来指定如何发送消息。

Mailx程序发送消息的命令行的格式为：

```
mail [-einv] [-a header] [-b addr] [-c addr] [-s subj] to-addr
```

mail命令使用表25-6中列出的命令行参数。

表25-6 Mailx命令行参数

参 数	描 述
-a	指定额外的SMTP头中的行
-b	给消息增加一个BCC收件人
-c	给消息增加一个CC收件人
-e	如果消息为空，不要发送消息
-i	忽略TTY中断信号
-I	强制Mailx以交互模式运行
-n	不要读取/etc/mail.rc起始文件
-s	指定一个标题行
-v	在终端上显示递送的细节

如你在表25-6中看到的，你完全可以使用命令行参数来创建整个E-mail消息。你唯一需要添加的就是消息正文。

要这么做，你需要将文本重定向给mail命令。这里有个如何直接在命令行上创建和发送E-mail消息的简单例子：

```
$ echo " This is a test message" | mail -s " Test message" rich
```

Mailx程序将来自echo命令的文本作为消息正文发送。这提供了一个从shell脚本发送消息的简单途径。下面是个简单的例子：

```
$ cat factmail
#!/bin/bash
# mailing the answer to a factorial

MAIL=`which mail` 

factorial=1
counter=1

read -p " Enter the number: " value
while [ $counter -le $value ]
do
    factorial=$((factorial * $counter))
    counter=$((counter + 1))
done

echo The factorial of $value is $factorial | mail -s " Factorial
answer" $USER
echo " The result has been mailed to you."
```

这段脚本不会假定Mailx程序位于标准位置。它使用which命令来确定mail程序的位置。

在计算出阶乘函数的结果后，shell脚本使用mail命令来将这个消息发送到用户自定义的\$USER环境变量，他应该是运行这个脚本的人。

```
$ ./factmail
Enter the number: 5
The result has been mailed to you.
$
```

你要做的就是查看邮件，确定是否收到回信：

```
$ mail
"/var/mail/rich" : 1 message 1 new
>N 1 Rich Blum      Thu Dec 9 10:32 13/586 Factorial answer
?
Return-Path: <rich@rich-Parallels-Virtual-Platform>
X-Original-To: rich@rich-Parallels-Virtual-Platform
Delivered-To: rich@rich-Parallels-Virtual-Platform
Received: by rich-Parallels-Virtual-Platform (Postfix, from userid 1000)
          id B4AA2A260081; Thu, 9 Dec 2010 10:32:24 -0500 (EST)
Subject: Factorial answer
To: <rich@rich-Parallels-Virtual-Platform>
X-Mailer: mail (GNU Mailutils 2.1)
Message-Id: <20101209153224.B4AA2A260081@rich-Parallels-Virtual-Platform>
Date: Thu, 9 Dec 2010 10:32:24 -0500 (EST)
From: rich@rich-Parallels-Virtual-Platform (Rich Blum)

The factorial of 5 is 120
```

在消息正文中只发送一行文本并不总是很方便。通常，你需要将整个输出作为E-mail消息发送。在这种情况下，你总是可以将文本重定向到临时文件中，然后用cat命令将输出重定向给mail程序：

```
$ cat diskmail
#!/bin/bash
# sending the current disk statistics in an e-mail message

date=`date +\%m/\%d/\%Y`
MAIL=`which mail`
TEMP=`mktemp tmp.XXXXXX` 

df -k > $TEMP
cat $TEMP | $MAIL -s "Disk stats for $date" $1
rm -f $TEMP
```

diskmail程序用date命令（以及一些特殊格式）得到了当前日期，找到了Mailx程序的位置，然后创建了一个临时文件。在此之后，它用df命令显示了当前磁盘空间的统计信息（参见第4章），并将输出重定向到了那个临时文件。

然后它会将临时文件重定向到mail命令，使用第一个命令行参数作为目的地地址，在标题头中使用当前日期。在运行这个脚本时，你不会看到任何命令行输出：

```
$ ./diskmail rich
```

但如果你检查邮件，你就会看到发出的消息：

```
$ mail
"/var/mail/rich" : 1 message 1 new
>N 1 Rich Blum      Thu Dec 9 10:35 19/1020 Disk stats for 12/09/2010
?
Return-Path: <rich@rich-Parallels-Virtual-Platform>
X-Original-To: rich@rich-Parallels-Virtual-Platform
Delivered-To: rich@rich-Parallels-Virtual-Platform
```

```

Received: by rich-Parallels-Virtual-Platform (Postfix, from userid 1000)
        id 3671B260081; Thu, 9 Dec 2010 10:35:39 -0500 (EST)
Subject: Disk stats for 12/09/2010
To: <rich@rich-Parallels-Virtual-Platform>
X-Mailer: mail (GNU Mailutils 2.1)
Message-Id: <20101209153539.3671B260081@rich-Parallels-Virtual-Platform>
Date: Thu, 9 Dec 2010 10:35:39 -0500 (EST)
From: rich@rich-Parallels-Virtual-Platform (Rich Blum)

Filesystem      1K-blocks    Used   Available Use% Mounted on
/dev/sdal       63315876  2595552  57504044  5% /
none            507052     228    506824  1% /dev
none            512648     192    512456  1% /dev/shm
none            512648     100    512548  1% /var/run
none            512648      0    512648  0% /var/lock
none            4294967296    0  4294967296  0% /media/psf
?

```

现在你要做的是用cron功能计划每日运行该脚本，你就能让磁盘空间报告自动发送到你的收件箱。系统管理不能比这个更简单了！

25.4 Mutt程序

Mutt程序是另外一个流行的命令行E-mail客户端程序，在1995年由Michael Elkins开发。它有一个Mailx程序中没有的功能，使得它成为shell脚本的很好的手头工具。

Mutt程序可以在E-mail消息中将文件作为附件发送。你不用像在使用Mailx时那样将长文本文件放进E-mail消息正文，而可以使用Mutt程序，将该文本文件作为消息正文的单独附件加进来。这个特性对发送长文本文件（比如日志文件）来说太好用了。

本节将会带你逐步了解在Linux系统上安装Mutt并用它来在shell脚本中将文件附加到E-mail消息上。

25.4.1 安装Mutt

在图形化E-mail客户端（比如Kmail和Evolution）流行的今天，Mutt程序并不算一个流行的程序，因此你的Linux发行版非常可能没有默认安装它。不过大多数Linux发行版都使用标准软件安装方法（参见第8章），将它包含在了安装用的普通发行版文件中。对于Ubuntu环境，你可以用下面的命令行命令来安装Mutt：

```
sudo apt-get install mutt
```

如果你的Linux发行版没有包含Mutt程序，或者你想安装最新版本，你可以到Mutt网站（www.mutt.org）上下载最新源码包，然后用第8章中讨论的方法来从源码包编译和安装这个包。

25.4.2 Mutt命令行

mutt命令为你提供了一些参数来控制Mutt如何运行。表25-7列出了可用的命令行参数。

表25-7 Mutt命令行参数

参 数	描 述
-A alias	将指定别名的展开版本传给STDOUT
-a file	用MIME协议将指定的文件附加到消息上
-b address	指定一个BCC (Blind Carbon Copy, 秘密抄送) 收件人
-c address	指定一个CC (Carbon Copy, 抄送) 收件人
-D	将所有配置选项值打印到STDOUT
-e command	指定在处理完初始化文件后运行的配置命令
-f mailbox	指定要加载的邮箱文件
-F muttrc	指定要读取的初始化文件, 而不是\$HOME/.muttrc
-h	显示帮助文本
-H draft	指定一个含有标题和正文的草稿文件来发送消息
-i include	指定一个文件来包含在消息的正文中
-m type	指定默认邮箱类型
-n	忽略系统配置文件
-p	恢复一个过期的消息
-Q query	查询一个配置变量值。查询会在所有配置文件都被解析以及任何在命令行上指定的命令都被执行后执行
-R	以只读模式打开邮箱
-s subject	指定消息的标题
-v	显示Mutt版本号和编译时定义
-x	模拟Mailx编辑模式
-y	以由邮箱命令指定的所有邮箱列表启动
-z	和-f一起使用时, 如果邮箱内没消息就不启动
-Z	打开由邮箱命令指定的含有新邮件的第一个邮箱

有了这么多命令行参数, 你可以直接从命令行定制E-mail消息, 而这也正是你想在shell脚本中做的。

跟Mailx程序一样, 有一个东西你无法在Mutt程序命令行指定, 那就是消息正文文本。如果你没有向Mutt重定向任何文本的话, 它会以文本图形模式为你启动一个编辑器窗口来输入消息正文。

这对shell脚本来说不是好事, 所以你一直要重定向一些文本作为消息正文, 即使你使用了附件选项来指定一个附加的文件。下一节将会介绍怎么操作。

25.4.3 使用Mutt

现在你可以开始在shell脚本中使用Mutt程序了。要在shell脚本中创建基本的mutt命令, 你需要包含一些命令行选项来指定消息的标题、附件和消息的所有收件人:

```
mutt -s Subject -a file -- recipients
```

recipients列表是以空格分隔的E-mail地址列表, 消息会发送到这些地址。如果你要附加多

个文件，你可以在-a选项后面将它们用空格分开；--符号用来将文件名和收件人地址列表分开。file参数必须是一个绝对路径名，或基于运行mutt命令的当前工作目录的相对路径名。

mutt命令有另外一个问题。如果你没有重定向文本作为消息正文，Mutt会自动进入全屏模式来让你在编辑器窗口输入文本。这很有可能并不是你想要做的，所以必须保证重定向一些文本作为消息正文，即使它是个空文件：

```
# echo "Here's the log file" | mutt -s "Log file" -a
/var/log/messages -- rich
```

这个命令会将系统的日志文件作为附件发送给本地系统上的E-mail地址rich。注意，你还必须有你要附加的文件的正确访问权限。图25-5演示了在Evolution邮件客户端中收到的E-mail消息。

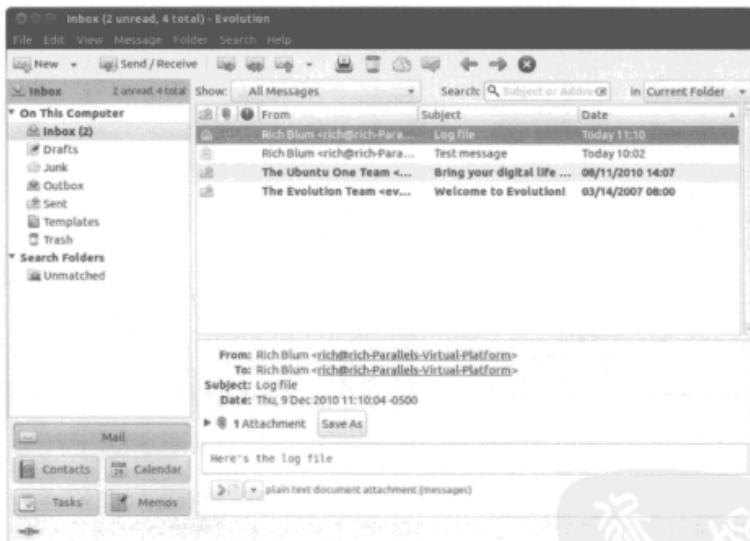


图25-5 使用Evolution来查看带附件的消息

注意，这条消息包括echo语句输出的正文文本以及那个单独的图标代表的附件。你可以直接从KMail客户端中保存附件。

警告 如果看了附件的名称，你会注意到Mutt用附加文件的基名作为附件的文件名。在使用临时文件时要小心，因为Mutt会使用临时文件名作为附件文件名。最好用更具描述性的名字来保存临时文件，而不是使用临时文件名。

25.5 小结

本章讨论了如何将E-mail功能纳入shell脚本中。定期将报告用E-mail发送给用户的能力是shell脚本中提供的一个非常好的功能。

在使用命令行发送E-mail消息前，你需要知道在Linux环境中E-mail是如何工作的，以及你需要安装和配置哪些应用。Linux E-mail环境由3个元素组成：邮件传送代理程序（MTA）、邮件投递代理程序（MDA）和邮件用户代理程序（MUA）。

MTA程序负责为Linux系统发送和接收邮件消息。它必须知道如何将收到的邮件消息传递给正确的用户邮箱，以及如何发送发往远程邮件服务器上的用户的外发邮件消息。通常，MTA程序会使用代理服务器（也叫前端主机）来执行具体的邮件递送。它会将任何发往远程邮件服务器上的用户的消息转发到前端主机来进一步递送。它依赖前端主机来独自完成递送。

MDA程序负责确保发往本地用户的邮件出现在正确的本地邮箱中。有时这个功能会直接由MTA程序执行。但如果你需要一些高级邮件投递功能，比如休假提醒或垃圾邮件过滤，你可以配置MTA程序来将消息传给MDA程序（MDA程序通常会自带这些功能）。

MUA程序允许独立的系统用户访问他们的邮箱并将给其他用户的外发消息传到MTA来进一步递送。这些程序的范围从简单的命令行程序（比如Mailx和Mutt）到绚丽的图形化程序（比如KMail和Evolution）。

从shell脚本中发送E-mail的最简单办法是使用Mailx程序。这个程序允许你在命令行上指定标题头和一个或多个收件人。你可以通过将文本重定向到Mailx程序来创建消息正文。你可以使用echo命令来输出单行文本，或使用cat命令来将整个文件的内容重定向到邮件消息。

Mutt程序是一个更高级的命令行MUA程序，它提供了向邮件消息附加文件的功能，而不用将文件中的文本放到消息正文中。这允许你附加大的文本文件，收件人可以方便地将它们保存到硬盘上，用其他程序来打开，比如电子表格或文字处理程序。

本书的最后两章会介绍shell脚本的一个重要部分，编写管理功能的脚本。如果你是Linux系统管理员，那你极有可能会遇到要监测某个系统功能的状态的情形。通过创建shell脚本并将它放到cron任务中，你可以轻松地监测Linux系统的当前状态。

本章内容

- 监测磁盘空间
- 进行备份
- 管理用户账户

没有什么情况，会比为Linux系统管理员编写脚本实用工具更让shell脚本编程有意义了。一般的Linux系统管理员每天会有大量的任务要做，从监测磁盘空间到备份重要文件再到管理用户账户。shell脚本工具可以让系统管理员的工作轻松许多。本章将会演示一些可以通过在bash shell中编写脚本工具获得的能力。

26.1 监测磁盘空间

对于多用户Linux系统来说，最大的问题之一是可用磁盘空间的总量。在有些情况下，比如在文件共享服务器上，磁盘空间很可能会因为一个粗心的用户而立刻用完。

这个shell脚本工具会帮你找出指定目录的前十名磁盘空间用户。它会生成一个以日期命名的报告，使得磁盘空间使用量可以被监测。

26.1.1 需要的功能

你需要用到的第一个工具是du命令（参见第4章）。该命令可以为每个文件和目录显示磁盘的使用情况。`-s`选项用来在目录一级总结整体使用状况。在计算单个用户使用的总体磁盘空间时，它会非常有用。下面的例子使用du命令来为`/home`目录内容总结每个用户的`$HOME`目录的情况：

```
$ du -s /home/*
6174428      /home/consultant
4740         /home/Development
4740         /home/Production
3860         /home/Samantha
7916         /home/Timothy
140376116    /home/user
$
```

-s选项能够很好地处理用户的\$HOME目录，但如果我们要查看系统目录比如/var/log的磁盘使用情况呢？

```
$ du -s /var/log/*
4      /var/log/alternatives.log
44     /var/log/alternatives.log.1
4      /var/log/apparmor
176    /var/log/apt
0      /var/log/aptitude
4      /var/log/aptitude.1.gz
4      /var/log/aptitude.2.gz
4      /var/log/aptitude.3.gz
4      /var/log/aptitude.4.gz
4      /var/log/aptitude.5.gz
160    /var/log/auth.log
...
$
```

这个列表很快就变得过于琐碎。这里，-S选项能更好地达到我们的目的，它为每个目录和子目录分别提供了一个总计。这允许你快速查明问题区域：

```
$ du -S /var/log
176    /var/log/apt
52     /var/log/exim4
1048   /var/log/dist-upgrade/20101011-1337
6148   /var/log/dist-upgrade
1248   /var/log/installer
228    /var/log/gdm
4      /var/log/news
4      /var/log/samba/cores/winbindd
4      /var/log/samba/cores
16     /var/log/samba
4      /var/log/unattended-upgrades
4      /var/log/sysstat
4      /var/log/speech-dispatcher
108    /var/log/ConsoleKit
64     /var/log/cups
4      /var/log/apparmor
12     /var/log/fsck
4844   /var/log
$
```

由于我们感兴趣的是占用了大块磁盘空间的目录，你需要对du产生的输出使用sort命令（参见第4章）：

```
$ du -S /var/log | sort -rn
6148   /var/log/dist-upgrade
4864   /var/log
1248   /var/log/installer
1048   /var/log/dist-upgrade/20101011-1337
228    /var/log/gdm
176    /var/log/apt
```

```

108   /var/log/ConsoleKit
64    /var/log/cups
52    /var/log/exim4
16    /var/log/samba
12    /var/log/fsck
4     /var/log/unattended-upgrades
4     /var/log/sysstat
4     /var/log/speech-dispatcher
4     /var/log/samba/cores/winbindd
4     /var/log/samba/cores
4     /var/log/news
4     /var/log/apparmor
$
```

-n选项允许按数字排序。-r选项会先列出最大数字。这对于找出占用磁盘空间最多的用户很有用。

sed编辑器（参见第18章和第20章）可以让这个列表更容易读懂。我们要关注的是前十名磁盘空间用户，所以当到了第11行，**sed**会删除列表的剩余部分。下一步是给列表中的每行加一个行号。第18章演示了如何给**sed**命令加个等号来实现这一目的。要让行号和磁盘空间文本位于同一行，可以用**N**命令将文本行合并在一起，跟我们在第20章中的处理一样。我们要用的**sed**命令看起来如下：

```

sed '{11,$0: =}' |
sed 'N; s/\n/ /'
```

现在输出可以用**gawk**命令清理了（参见第21章）。**sed**编辑器的输出会通过管道输出到**gawk**命令，然后用**printf**函数打印出来。

```
gawk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
```

在行号后，我们加了一个冒号(:)，还给每行文本的输出行中的每个字段间放了一个制表符。这会生成一个格式精致的前十名磁盘空间用户的列表：

```

$ du -S /var/log |
> sort -rn |
> sed '{11,$0: =}' |
> sed 'N; s/\n/ /' |
> gawk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
1:      6148    /var/log/dist-upgrade
2:      4864    /var/log
3:      1248    /var/log/installer
4:      1048    /var/log/dist-upgrade/20101011-1337
5:      228     /var/log/gdm
6:      176     /var/log/apt
7:      108     /var/log/ConsoleKit
8:       64     /var/log/cups
9:       52     /var/log/exim4
10:      16     /var/log/samba
$
```

现在你已经做好了。下一步就是用这些信息创建脚本。

26.1.2 创建脚本

为了节省时间和精力,这个脚本会为多个指定目录创建报告。我们用一个叫做CHECK_DIRECTORIES的变量来完成这一任务。这里为了说明我们的目的,该变量被设为两个目录:

```
CHECK_DIRECTORIES="/var/log /home"
```

脚本包含一个for循环来为这个变量中列出的每个目录执行du命令。这个方法用来读取和处理列表中的值(参见第12章)。每次for循环都会遍历变量CHECK_DIRECTORIES中的值列表,它会将列表中的下一个值赋给DIR_CHECK变量:

```
for DIR_CHECK in $CHECK_DIRECTORIES
do
...
du -S $DIR_CHECK
...
done
```

为了方便识别,我们用date命令给报告的文件名加个日期戳。脚本用exec命令(参见第14章)来将它的输出重定向到加了日期戳的报告文件中:

```
DATE=$(date '+%m%d%y')
exec > disk_space_$DATE.rpt
```

为了生成格式精致的报告,这个脚本会用echo命令来输出一些报告标题:

```
echo "Top Ten Disk Space Usage"
echo "for $CHECK_DIRECTORIES Directories"
```

现在让我们看一下将这个脚本的各部分全部放在一起会是什么样子:

```
#!/bin/bash
#
# Big_Users - find big disk space users in various directories
#####
# Parameters for Script
#
#CHECK_DIRECTORIES="/var/log /home"      #directories to check
#####
# Main Script #####
#
DATE=$(date '+%m%d%y')          #Date for report file
#
exec > disk_space_$DATE.rpt      #Make report file Std Output
#
echo "Top Ten Disk Space Usage"    #Report header for whole report
echo "for $CHECK_DIRECTORIES Directories"
#
for DIR_CHECK in $CHECK_DIRECTORIES      #loop to du directories
do
  echo ""
  echo "The $DIR_CHECK Directory:"      #Title header for each directory
#
# Create a listing of top ten disk space users
  du -S $DIR_CHECK 2>/dev/null |
```

```

sort -rn |
sed '{11,$D-}' |
sed 'N; s/\n/ /' |
gawk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
#
done                                #end of for loop for du directories
#

```

现在你已经得到完整的脚本了。这个简单的shell脚本会为你选择的每个目录创建一个加了日期截的前十名磁盘空间用户报告。

26.1.3 运行脚本

在让Big_Users脚本自动运行之前，你会想手动测试几次，保证它如你期望的那样运行：

```

$ ./Big_Users
$
$ cat disk_space_012311.rpt
Top Ten Disk Space Usage
for /var/log /home Directories

The /var/log Directory:
1:      6148    /var/log/dist-upgrade
2:      4892    /var/log
3:      1248    /var/log/installer
4:      1048    /var/log/dist-upgrade/20101011-1337
5:      176     /var/log/apt
6:      108     /var/log/ConsoleKit
7:       64     /var/log/cups
8:       52     /var/log/exim4
9:       16     /var/log/samba
10:      12     /var/log/fsck

The /home Directory:
1:  92365332  /home/user/.VirtualBox/HardDisks
2:  18659720  /home/user/Downloads
3:  17626092  /home/user/archive
4:  6174408   /home/Timothy/Junk/More_Junk
5:  6174408   /home/Timothy/Junk
6:  6174408   /home/consultant/Work
7:  6174408   /home/consultant/Downloads
8:  3227768   /home/user/vmware/Mandriva
9:  3212464   /home/user/vmware/Fedora
10: 104632    /home/user/vmplayer
$
```

它能正常运行！现在你可以让这个脚本在需要时自动运行了。你可以用cron表来实现（参见第15章）。

说明 在运行含有需要root权限的bash shell命令的脚本时，需要用su或sudo命令，否则脚本会输出意料之外的结果。

运行这个脚本的频率取决于文件服务器的活跃程度。要一个星期运行一次这个脚本，可以将下面的内容加到cron表中：

```
15 7 * * 1 /home/user/Big_Users
```

该条目会在每周一早晨7:15运行这个脚本。现在，每周一早晨的第一件事就是喝着咖啡，查看每周的磁盘使用情况报告。

26.2 进行备份

不管你负责的Linux系统是商用的还是家用的，丢失数据的后果都很严重。为了防止这种事情发生，执行定时备份会比较好。

但好想法和可行性经常是两回事。要安排一个备份计划来存储重要文件绝非易事。而shell脚本经常能在这方面帮上忙。

本节将会演示两种使用shell备份Linux系统上的数据的不同方法。

归档数据文件

如果你正在用Linux系统来做一个重要项目，你可以创建一个shell脚本来自动捕获特定目录的快照。在一个配置文件中指定这些目录将允许你在特定项目有变化时修改它们。这会帮忙避免耗时的主要归档文件恢复过程。

本节将会介绍如何创建自动化shell脚本来捕获指定目录的快照并保留一份过去的数据归档。

1. 需要的功能

Linux中归档数据的主要工具是tar命令（参见第4章）。tar命令用来将整个目录归档到单个文件中。这里有个用tar命令来创建工作目录归档文件的例子：

```
$ tar -cf archive.tar /home/user/backup_test
tar: Removing leading '/' from member names
$
```

tar命令会显示一条警告消息，该消息说明它要删除路径名开头的斜线来将它从绝对路径名变成相对路径名（参见第3章）。这允许将tar归档文件解压到文件系统中的任何地方。这可能想在脚本中去掉这条消息。这可以将STDERR重定向到/dev/null文件实现（参见第14章）：

```
$ tar -cf archive.tar /home/user/backup_test 2>/dev/null
$
```

由于tar归档文件会消耗大量的磁盘空间，通常应该压缩一下这个文件。可以简单地加一个-z选项就好了。这会将tar归档文件压缩成gzip格式的tar文件，也叫tarball。确保使用正确的文件权限来表示这个文件是个tarball。用.tar.gz或.tgz都行。这里有个创建工作目录的tarball的例子：

```
$ tar -zcf archive.tar.gz /home/user/backup_test 2>/dev/null
$
```

现在你已经完成了归档脚本的主要部分。

不用为你要备份的每个新目录或文件修改或增加新的归档脚本，你可以使用配置文件。配置文件应该包含你要在归档中包含的每个目录或文件：

```
$ cat Files_To_Backup
/home/user/Downloads
/home/user/Documents/CLandSS_V2
/home/Samantha/Documents
/home/Does_not_exist
/home/Timothy/Junk
/home/consultant/Work
$
```

警告 如果你使用的是带图形化桌面的Linux发行版，那么归档整个\$HOME目录时要注意。虽然桌面很炫，但\$HOME目录含有很多跟图形化桌面有关的配置文件和临时文件。它会生成一个比你期望的大很多的归档文件。选择一个用来存储工作文件的子目录，然后在归档配置文件中使用那个子目录。

你可以让脚本读取配置文件，然后将每个目录或文件的名字加到归档列表中。为了做到这个，可以使用简单的read命令（参见第13章）来读取该文件中的每一条记录。但不要像我们在第13章中那样用cat命令通过管道传给while循环，这个脚本会将用exec命令（参见第14章）来重定向STDIN。下面就是这个用法：

```
exec < $CONFIG_FILE
read FILE_NAME
```

注意，我们为归档配置文件以及从该文件中读取的每条记录使用了变量。只要read命令在配置文件中发现还有记录要读，它就会在?变量中（参见第10章）返回一个退出状态码0来表示成功。你可以将它作为while循环的测试条件来读取配置文件中的所有记录：

```
while [ $? -eq 0 ]
do
...
read FILE_NAME
done
```

一旦read命令到了配置文件的末尾，它就会返回一个非零状态码。那时，脚本会退出while循环。

在while循环中，我们需要做两件事。首先，你必须将文件名或路径名加到归档列表中。而更重要的是，检查和判断那个文件和目录是不是存在。很可能你从文件系统中删除了一个目录而忘了更新归档配置文件。你可以用简单的if语句和test命令的文件比较（参见第11章）来检查文件或目录是否存在。如果文件或目录存在，它会被加到要归档的文件列表FILE_LIST中。否则，就显示一条警告消息。下面是它看起来的样子：

```

if [ -f $FILE_NAME -o -d $FILE_NAME ]
then
    # If file exists, add it's name to the list.
    FILE_LIST="$FILE_LIST $FILE_NAME"
else
    # If file doesn't exists, issue warning
    echo
    echo "$FILE_NAME. does not exist."
    echo "Obviously, I will not include it in this archive."
    echo "It is listed on line $FILE_NO of the config file."
    echo "Continuing to build archive list..."
    echo
fi
#
FILE_NO=$[FILE_NO + 1]      # Increase Line/File number by one.

```

由于归档配置文件中的记录可以是文件名或目录，`if`语句会用`-f`和`-d`选项测试它们的存在。`or`选项`-o`允许文件或目录的存在测试返回一个真值来让整个`if`语句都成立。

为了在跟踪不存在的目录和文件上提供一点额外帮助，我们添加了变量`FILE_NO`。这样，这个脚本可以告诉你在归档配置文件中哪行含有不正确或缺失的文件或目录。

现在你应该有足够的信息来开始构建这个脚本了。下一节将会带你逐步创建这个归档脚本。

2. 创建按日归档的脚本

`Daily_Archive`脚本会自动在指定位置创建一个归档，使用当前日期来唯一标识该文件。下面是脚本中的那部分代码：

```

DATE='date +%y%m%d'
#
# Set Archive File Name
#
FILE=archive$DATE.tar.gz
#
# Set Configuration and Destination File
#
CONFIG_FILE=/home/user/archive/Files_To_Backup
DESTINATION=/home/user/archive/$FILE
#

```

`DESTINATION`变量会将归档文件的全路径名加上去。`CONFIG_FILE`变量指向含有要归档文件的归档配置文件。如果需要，二者都可以很方便地改成备用目录和文件。

将所有的内容放在一起，`Daily_Archive`脚本看起来如下：

```

#!/bin/bash
#
# Daily Archive - Archive designated files & directories
#####
#
# Gather Current Date
#
DATE='date +%y%m%d'
#
# Set Archive File Name
#

```

```

FILE=archive$DATE.tar.gz
#
# Set Configuration and Destination File
#
CONFIG_FILE=/home/user/archive/Files_To_Backup
DESTINATION=/home/user/archive/$FILE
#
##### Main Script #####
#
# Check Backup Config file exists
#
if [ -f $CONFIG_FILE ] # Make sure the config file still exists.
then                      # If it exists, do nothing but continue on.
    echo
else                      # If it doesn't exist, issue error & exit script.
    echo
    echo "$CONFIG_FILE does not exist."
    echo "Backup not completed due to missing Configuration File"
    echo
    exit
fi
#
# Build the names of all the files to backup
#
FILE_NO=1      # Start on Line 1 of Config File.
exec < $CONFIG_FILE      # Redirect Std Input to name of Config File
#
read FILE_NAME      # Read 1st record
#
while [ $? -eq 0 ]      # Create list of files to backup.
do
    # Make sure the file or directory exists.
    if [ -f $FILE_NAME -o -d $FILE_NAME ]
    then
        # If file exists, add its name to the list.
        FILE_LIST="$FILE_LIST $FILE_NAME"
    else
        # If file doesn't exist, issue warning
        echo
        echo "$FILE_NAME, does not exist."
        echo "Obviously, I will not include it in this archive."
        echo "It is listed on line $FILE_NO of the config file."
        echo "Continuing to build archive list..."
        echo
    fi
    #
    FILE_NO=$[$FILE_NO + 1] # Increase Line/File number by one.
    read FILE_NAME          # Read next record.
done
#
#####
#
# Backup the files and Compress Archive
#
tar -czf $DESTINATION $FILE_LIST 2> /dev/null
#

```

3. 运行按日归档的脚本

测试Daily_Archive脚本非常简单：

```
$ ./Daily_Archive
```

```
/home/Does_not_exist. does not exist.  
Obviously, I will not include it in this archive.  
It is listed on line 4 of the config file.  
Continuing to build archive list...
```

你会看到这个脚本捕捉到了一个不存在的目录，/home/Does_not_exist。它会让你知道这个错误的行在配置文件中的行号，然后会继续创建列表和归档数据。现在数据稳妥地归档到了tarball文件中。

4. 创建按小时归档的脚本

如果你是在文件更改很频繁的高容量生产环境中，按日归档可能不够用。如果你要将归档频率提高到每小时更新一次，你还要考虑另一个因素。

在按小时备份文件而依然使用date命令来在每个文件名中包含时间戳时，事情很快就会变得很乱。筛选一个含有文件而文件名看起来如下的目录会很乏味：

```
archive01021110233.tar.gz
```

不必将所有的归档文件放到同一个目录中，你可以为归档文件创建一个目录层级。图26-1说明了这个原则。

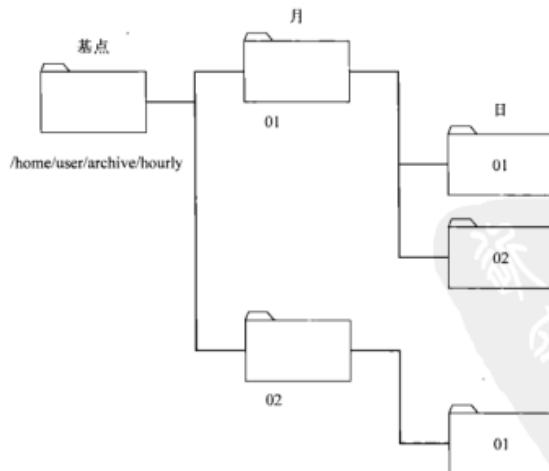


图26-1 创建归档目录层级结构

这个归档目录包含了跟一年中的各个月份对应的目录，将月的序号作为目录名。而每月的目录中又包含跟一个月中的各天对应的目录（用天的序号来作为目录名）。这样允许你只用给每个归档文件加时间戳然后将它们放到跟日和月份对应的目录中就行。

现在还有个新的问题要解决。这个脚本必须自动创建每个月和天目录，并且知道如果它们已经存在的话，就不必创建了。

如果仔细查看mkdir命令的命令行选项的话（参见第3章），你会找到-p命令行选项。这个选项允许在单个命令中创建目录和子目录；而且，额外的好处是，如果目录已经存在，则它不会产生错误消息。太好了。

现在我们可以创建Hourly_Archive脚本了。这里是前半部分脚本：

```
#!/bin/bash
#
# Hourly_Archive - Every hour create an archive
#####
#
# Set Configuration File
#
CONFIG_FILE=/home/user/archive/hourly/Files_To_Backup
#
# Set Base Archive Destination Location
#
BASEDEST=/home/user/archive/hourly
#
# Gather Current Day, Month & Time
#
DAY='date +%d'
MONTH='date +%m'
TIME='date +%k%M'
#
# Create Archive Destination Directory
#
mkdir -p $BASEDEST/$MONTH/$DAY
#
# Build Archive Destination File Name
#
DESTINATION=$BASEDEST/$MONTH/$DAY/archive$TIME.tar.gz
#
##### Main Script #####
...
```

一旦脚本到了Hourly_Archive的Main Script部分，脚本就跟Daily_Archive脚本完全一样了。大部分工作已经完成了。

Hourly_Archive会从date命令提取天的序号和月份值，以及用来唯一识别这个归档文件的时间戳。然后它用这个信息创建跟天对应的目录（如果已经存在的话，就安静地退出）。最终，这个脚本用tar命令创建了归档文件并将它压缩成一个tarball。

5. 运行按小时归档的脚本

跟Daily_Archive脚本一样，最好在将Hourly_Archive脚本放到cron表中之前测试一下：

```
$ cat /home/user/archive/hourly/Files_To_Backup
/home/Development/Simulation_Logs
/home/Production/Machine_Errors
$
$ ./Hourly_Archive
$
$ ls /home/user/archive/hourly/01/02
archive1601.tar.gz
$
$ ./Hourly_Archive

$ ls /home/user/archive/hourly/01/02
archive1601.tar.gz      archive1602.tar.gz
$
```

这个脚本第一次运行很正常，创建了相应的月和天目录，然后创建了这个归档文件。为了进行充分测试，我们又运行了一次看它遇到已有目录时会不会有问题。这个脚本依然运行正常并创建了第二个归档文件。现在可以放到cron表中了。

26.3 管理用户账户

管理用户账户不只是添加、修改和删除账户，你还得考虑安全问题、保留工作的需求以及对账户的精确管理。这是一个耗时的工作。这又是一个证明写脚本工具可以节约时间的实例。

26.3.1 需要的功能

删除账户是管理账户工作中比较复杂的。在删除账户时，至少需要4个步骤：

- (1) 获得要删除用户账户的正确账户名；
- (2) 强制终止正在系统上运行的属于该账户的进程；
- (3) 确认系统上属于该账户的所有文件；
- (4) 删除用户账户。

很容易遗漏某个步骤。本节的shell脚本工具会帮你避免犯类似的错误。

1. 获取正确用户名

账户删除过程中的第一步最重要，获取要删除的用户账户的正确名称。由于这是个交互式脚本，你可以用read命令（参见第13章）来获取账户名称。如果脚本用户走开了，留下了悬而未决的问题，你可以在read命令中用-t选项，给用户60秒的时间回答问题后超时退出：

```
echo "Please enter the username of the user "
echo -e "account you wish to delete from system: \c"
read -t 60 ANSWER
```

由于打扰不可避免，最好给用户三次机会来回答问题。为了做到这个，你可以用一个while循环（参见第12章）加一个-z选项来测试ANSWER变量是否为空。在脚本第一次进入while循环时，ANSWER变量会是空的，因为问题会放到循环的最后：

```

while [ -z "$ANSWER" ]
do
...
echo "Please enter the username of the user "
echo -e "account you wish to delete from system: \c"
read -t 60 ANSWER
done

```

现在你需要一个途径来跟脚本用户交互，在第一次提问超时发生时，在只剩一次回答问题的机会时，等等。case语句（参见第11章）是最适合这里的结构化命令。通过增长的ASK_COUNT变量，你可以设定不同的消息来和脚本用户交互。这部分的代码看起来如下：

```

case $ASK_COUNT in
2)
echo
echo "Please answer the question."
echo
;;
3)
echo
echo "One last try...please answer the question."
echo
;;
4)
echo
echo "Since you refuse to answer the question..."
echo "exiting program."
echo
#
exit
;;
esac
#

```

现在这个脚本已经有了为询问用户要删除哪个账户它所需要的结构。在这个脚本中，你还需要另外几个问题来询问用户，而只提那么一个问题就已经是一大堆代码了！因此，让我们将这段代码放到一个函数中（参见第16章）来在Delete_User脚本中多处使用。

你要做的第一件事是声明函数名，get_answer。下一步，用unset命令（参见第5章）清除脚本用户前面给出的答案。做这两件事的代码如下：

```

function get_answer {
#
unset ANSWER

```

在原来代码中你要改变的另一件事是对用户脚本的实际提问。这个脚本不会每次都问同一个问题，所以让我们创建两个新的变量LINE1和LINE2来处理提问行：

```

echo $LINE1
echo -e $LINE2" \c"

```

但不是每个问题都有两行要显示，有的只要一行。你可以用if结构（参见第11章）来解决这个问题。这个函数会测试LINE2是否为空，如果为空，则只用LINE1：

```

if [ -n "$LINE2" ]
then
    echo $LINE1
    echo -e $LINE2"\c"
else
    echo -e $LINE1"\c"
fi

```

最终，我们的函数需要在后面通过清空LINE1和LINE2变量来清除一下自己。因此，现在这个函数看起来如下：

```

function get_answer {
#
unset ANSWER
ASK_COUNT=0
#
while [ -z "$ANSWER" ]
do
    ASK_COUNT=${ASK_COUNT}+1
#
    case $ASK_COUNT in
    2)
        echo
        ...
    esac
#
    echo
    if [ -n "$LINE2" ]           #Print 2 lines
    then
        echo $LINE1
        echo -e $LINE2"\c"
    else                         #Print 1 line
        echo -e $LINE1"\c"
    fi
#
    read -t 60 ANSWER
done
#
unset LINE1
unset LINE2
#
} #End of get_answer function

```

要问脚本用户删除哪个账户，你需要设置一些变量，然后调用get_answer函数。使用新函数让脚本代码清爽了许多：

```

LINE1="Please enter the username of the user "
LINE2="account you wish to delete from system:"
get_answer
USER_ACCOUNT=$ANSWER

```

鉴于可能存在输入错误，你会想要验证一下输入的用户账户。这很容易，因为我们已经有了提问的代码：

```
LINE1="Is $USER_ACCOUNT the user account."
LINE2="you wish to delete from the system? [y/n]"
get_answer
```

只要问题得到回答，脚本就需要处理回答了。变量ANSWER再次将脚本用户的回答带回问题中。如果用户回答了“yes”，你就得到了要删除的正确用户账户，可以在脚本中继续执行下去了。你可以用case语句（参见第11章）来处理这个回答。一定要用case语句，这样它才会检查输入“yes”的多种方法：

```
case $ANSWER in
y|Y|YES|yes|Yes|yEs|yeS|YES|yES )
#
;;
*)
    echo
    echo "Because the account, $USER_ACCOUNT, is not "
    echo "the one you wish to delete, we are leaving the script..."
    echo
    exit
;;
esac
```

这个脚本需要处理很多次用户的yes/no回答。因此，创建一个函数来处理这个任务是有意义的。只要对前面的代码作很少的改动就可以了。简单地声明一下函数名，给case语句中加两个变量EXIT_LINE1和EXIT_LINE2就可以了。这些修改以及最后的一些变量清理工作会生成process_answer函数：

```
function process_answer {
#
case $ANSWER in
y|Y|YES|yes|Yes|yEs|yeS|YES|yES )
#
*)
    echo
    echo $EXIT_LINE1
    echo $EXIT_LINE2
    echo
    exit
;;
esac
#
unset EXIT_LINE1
unset EXIT_LINE2
#
} #End of process_answer function
```

现在一个简单的函数调用就会处理回答了：

```
EXIT_LINE1="Because the account, $USER_ACCOUNT, is not "
EXIT_LINE2="the one you wish to delete, we are leaving the script..."
process_answer
```

用户已经给了我们要删除的账户名并验证过了。现在最好核对一下这个用户账户是否在系统

上真实存在。还有，最好将完整的账户记录发给脚本用户，核对这是不是真的要删除的那个账户。要完成这些工作，使用变量USER_ACCOUNT_RECORD，将它设成grep（参见第4章）在/etc/passwd文件中查找该用户账户的输出。-w选项允许你在用户账户记录中对这个特定用户账户做一个精确匹配：

```
USER_ACCOUNT_RECORD=$(cat /etc/passwd | grep -w $USER_ACCOUNT)
```

如果在/etc/passwd中没找到用户账户记录，那意味着这个账户已经被删除了或者从未存在过。在任何一种情况中，你都需要让脚本用户知道这个，然后退出脚本。grep命令的退出状态码可以在这里帮到我们。如果没找到这个账户记录，?变量会被设成1：

```
if [ $? -eq 1 ]
then
    echo
    echo "Account, $USER_ACCOUNT, not found."
    echo "Leaving the script..."
    echo
    exit
fi
```

如果找到了这个记录，你仍然需要验证这个脚本用户是不是正确的账户。这里我们建立函数所花的工夫完全值了。你要做的只是设置正确的变量并调用函数：

```
echo "I found this record:"
echo $USER_ACCOUNT_RECORD
echo
#
LINE1="Is this the correct User Account? [y/n]"
get_answer
#
EXIT_LINE1="Because the account, $USER_ACCOUNT, is not"
EXIT_LINE2="the one you wish to delete, we are leaving the script..."
process_answer
```

2. 删除属于账户的进程

到目前为止，你已经得到并验证了要删除的用户账户的正确名称。为了从系统上删除该用户账户，这个账户不能拥有任何当前运行中的进程。因此，下一步就是查找并终止这些进程。这会稍微麻烦一些。

查找用户进程较为简单。这里脚本可以用ps命令（参见第4章）和-u选项来定位属于该账户的任何运行中的进程。如果有任何进程存在，它们会显示给脚本用户：

```
ps -u $USER_ACCOUNT
```

你可以用ps命令的退出状态码和case结构来决定下一步做什么：

```
case $? in
1)
    echo "There are no processes for this account currently running."
    echo
;;
0)
```

```

unset ANSWER
LINE1="Would you like me to kill the process(es)? [y/n]"
get_answer
...
esac

```

如果退出状态码返回了1，那么系统上没有属于该用户账户的进程在运行。但如果退出状态码返回了0，那么系统上有属于该账户的进程在运行。在这种情况下，脚本需要问脚本用户他们是否要终止这些进程。可以用get_answer函数来完成这个任务。

你可能会认为这个脚本的下一步动作是调用process_answer函数。很遗憾，下一个任务对于process_answer来说太复杂了。你需要嵌入另一个case语句来处理脚本用户的回答。case语句的第一部分看起来和process_answer函数很像：

```

case $ANSWER in
  y|Y|YES|yes|Yes|yeS|yEs|yES ) # If user answers "yes".
    #kill User Account processes.
  ;;
*) # If user answers anything but "yes", do not kill.
  echo
  echo "Will not kill the process(es)"
  echo
;;
esac

```

你可以看出，case语句本身没什么有意思的。真正有意思的是case语句的“yes”部分。这里，该用户账户的进程要被终止。要这么做，再调用一次ps命令。这次将输出发送到临时报告文件中，而不是屏幕上：

```
ps -u $USER_ACCOUNT > $USER_ACCOUNT_Running_Process.rpt
```

你可以用exec命令和while循环来读取报告文件，类似于前面我们在Daily_Archive脚本中读取备份配置文件的方法：

```

exec < $USER_ACCOUNT_Running_Process.rpt
read USER_PROCESS_REC
while [ $? -eq 0 ]
do
  ...
  read USER_PROCESS_REC
done

```

在while循环内，用cut命令来从每个运行中的进程的状态记录中提取进程ID（PID）。一旦你有了PID，使用kill命令（参见第4章）和-9选项来无条件终止进程：

```

USER_PID=$(echo $USER_PROCESS_REC | cut -d " " -f1)
kill -9 $USER_PID
echo "Killed process $USER_PID"

```

现在所有属于该用户账户的进程都已经被终止，脚本可以进行下一步，查找该用户账户的所有文件了。

3. 查找属于账户的文件

在从系统上删除用户账户时，最好将属于该用户的所有文件归档。除此之外，最好删除这些文件或将他们的所属关系赋给另一个账户。如果你要删除的账户的UID是1003，而你没有删除或修改它们的所属关系，那么下一个创建的UID为1003的账户会拥有这些文件。你能想到在这种情况下会出现安全隐患。

`Delete_User`脚本不会替你做所有的事，但它会创建一个在`Daily_Archive`脚本中可以用作备份配置文件的报告。你可以用这个报告来帮助你删除文件或重新分配文件的所属关系。

要找到用户文件，你可以用`find`命令。`find`命令用`-u`选项查找整个文件系统，它会精确定位属于该用户的所有文件。这个命令看起来如下：

```
find / -user $USER_ACCOUNT > $REPORT_FILE
```

跟处理用户账户的进程相比，它非常简单。`Delete_User`脚本的下一步——删除用户账户会更简单。

4. 删除账户

对从系统上删除用户账户心存畏惧总是好事。因此，你应该再问一次脚本用户是否真的想删除该账户：

```
LINE1="Do you wish to remove $User_Account's account from system? [y/n]"
get_answer
#
EXIT_LINE1="Since you do not wish to remove the user account."
EXIT_LINE2="$USER_ACCOUNT at this time, exiting the script..."
process_answer
```

最后，我们就到了整个脚本的主要目的，真正从系统上删除该用户账户。这里你可以用`userdel`命令（参见第6章）：

```
userdel $USER_ACCOUNT
```

现在，我们已经有了所有部分，可以将它们一起拼成一个完整的、有用的脚本工具。

26.3.2 创建脚本

记住，`Delete_User`脚本跟脚本用户的互动很多。因此，包含一些提示来在脚本执行时告诉用户正在做什么很重要。

在脚本的顶部声明了两个函数，`get_answer`和`process_answer`。然后脚本就进入了删除用户的4个步骤：获得并确认用户账户名，查找和终止用户的进程，创建一份属于该用户账户的所有文件的报告，以及最终删除用户账户。

下面是整个`Delete_User`脚本：

```
#!/bin/bash
#
#Delete_User - Automates the 4 steps to remove an account
#
```

```
#####
# Define Functions
#
#####
function get_answer {
#
unset ANSWER
ASK_COUNT=0
#
while [ -z "$ANSWER" ]    #While no answer is given, keep asking.
do
  ASK_COUNT=${ASK_COUNT}+1
#
  case $ASK_COUNT in
    2)
      echo
      echo "Please answer the question."
      echo
      ;;
    3)
      echo
      echo "One last try...please answer the question."
      echo
      ;;
    4)
      echo
      echo "Since you refuse to answer the question...""
      echo "exiting program."
      echo
      #
      exit
      ;;
  esac
#
echo
#
if [ -n "$LINE2" ]
then
  echo $LINE1          #Print 2 lines
  echo -e $LINE2 '\c'
else
  echo -e $LINE1 '\c'
fi
#
# Allow 60 seconds to answer before time-out
read -t 60 ANSWER
done
# Do a little variable clean-up
unset LINE1
unset LINE2
#
} #End of get_answer function
#####

```

```

function process_answer {
#
case $ANSWER in
y|Y|YES|yes|Yes|yEs|YeS|yEs )
# If user answers "yes", do nothing.
;;
*)
# If user answers anything but "yes", exit script
    echo
    echo $EXIT_LINE1
    echo $EXIT_LINE2
    echo
    exit
;;
esac
#
# Do a little variable clean-up
#
unset EXIT_LINE1
unset EXIT_LINE2
#
} #End of process_answer function
#
#####
# End of Function Definitions
#
#####
# Main Script #####
# Get name of User Account to check
#
echo "Step #1 - Determine User Account name to Delete"
echo
LINE1="Please enter the username of the user "
LINE2="account you wish to delete from system:"
get_answer
USER_ACCOUNT=$ANSWER
#
# Double check with script user that this is the correct User Account
#
LINE1="Is $USER_ACCOUNT the user account "
LINE2="you wish to delete from the system? [y/n]"
get_answer
#
# Call process_answer function:
#     if user answers anything but "yes", exit script
#
EXIT_LINE1="Because the account, $USER_ACCOUNT, is not "
EXIT_LINE2="the one you wish to delete, we are leaving the script..."
process_answer
#
#####
# Check that USER_ACCOUNT is really an account on the system
#
USER_ACCOUNT_RECORD=$(cat /etc/passwd | grep -w $USER_ACCOUNT)
#

```

```

if [ $? -eq 1 ]          # If the account is not found, exit script
then
    echo
    echo "Account, $USER_ACCOUNT, not found."
    echo "Leaving the script..."
    echo
    exit
fi
#
echo
echo "I found this record:"
echo $USER_ACCOUNT_RECORD
echo
#
LINE1="Is this the correct User Account? [y/n]"
get_answer
#
#
# Call process_answer function:
#     if user answers anything but "yes", exit script
#
EXIT_LINE1="Because the account, $USER_ACCOUNT, is not "
EXIT_LINE2="the one you wish to delete, we are leaving the script...""
process_answer
#
#####
# Search for any running processes that belong to the User Account
#
echo
echo "Step #2 - Find process on system belonging to user account"
echo
echo "$USER_ACCOUNT has the following processes running: "
echo
#
ps -u $USER_ACCOUNT      #List user processes running.

case $? in
1)   # No processes running for this User Account
    #
    echo "There are no processes for this account currently running."
    echo
    ;;
0)   # Processes running for this User Account.
    # Ask Script User if wants us to kill the processes.
    #
    unset ANSWER
    LINE1="Would you like me to kill the process(es)? [y/n]"
    get_answer
    #
    case $ANSWER in
        y|Y|YES|yes|Yes|yeS|yEs|yES )      # If user answers "yes",
                                                # kill User Account processes.
        #
        echo
    esac
esac

```

```

#
# Clean-up temp file upon signals
trap "rm $USER_ACCOUNT_Running_Process.rpt" SIGTERM SIGINT SIGQUIT
#
# List user processes running
ps -u $USER_ACCOUNT > $USER_ACCOUNT_Running_Process.rpt
#
exec < $USER_ACCOUNT_Running_Process.rpt      # Make report Std Input
#
read USER_PROCESS_REC          # First record will be blank
read USER_PROCESS_REC
#
while [ $? -eq 0 ]
do
  # obtain PID
  USER_PID=$(echo $USER_PROCESS_REC | cut -d " " -f1)
  kill -9 $USER_PID
  echo "Killed process $USER_PID"
  read USER_PROCESS_REC
done
#
echo
rm $USER_ACCOUNT_Running_Process.rpt      # Remove temp report.
;;
*) # If user answers anything but "yes", do not kill.
echo
echo "Will not kill the process(es)"
echo

;;
esac
;;
esac
#####
# Create a report of all files owned by User Account
#
echo
echo "Step #3 - Find files on system belonging to user account"
echo
echo "Creating a report of all files owned by $USER_ACCOUNT."
echo
echo "It is recommended that you backup/archive these files."
echo "and then do one of two things:"
echo " 1) Delete the files"
echo " 2) Change the files' ownership to a current user account."
echo
echo "Please wait. This may take a while..."
#
REPORT_DATE=`date +%y%m%d`
REPORT_FILE=$USER_ACCOUNT_Files_"$REPORT_DATE".rpt
#
find / -user $USER_ACCOUNT > $REPORT_FILE 2>/dev/null
#
echo

```

```

echo "Report is complete."
echo "Name of report:      $REPORT_FILE"
echo "Location of report:   `pwd`"
echo
#####
# Remove User Account
echo
echo "Step #4 - Remove user account"
echo
#
LINE1="Do you wish to remove $USER_ACCOUNT's account from system? [y/n]"
get_answer
#
# Call process_answer function:
#     if user answers anything but "yes", exit script
#
EXIT_LINE1="Since you do not wish to remove the user account."
EXIT_LINE2="$USER_ACCOUNT at this time, exiting the script..."
process_answer
#
userdel $USER_ACCOUNT          #delete user account
echo
echo "User account, $USER_ACCOUNT, has been removed"
echo
#

```

工作量很大！但Delete_User脚本会是个很好的省时工具，会帮你避免很多删除用户账户时出现的琐碎问题。

运行脚本

由于被设计成了一个交互式脚本，Delete_User脚本不能放在cron表中。但保证它能按期望工作仍然很重要。我们会通过删除一个系统上临时设置的consultant账户来测试这个脚本：

```

$ ./Delete_User
Step #1 - Determine User Account name to Delete

Please enter the username of the user
account you wish to delete from system: consultant

Is consultant the user account
you wish to delete from the system? [y/n] y

I found this record:
consultant:x:1004:1001:Consultant....:/home/consultant:/bin/bash

Is this the correct User Account? [y/n] y

Step #2 - Find process on system belonging to user account

consultant has the following processes running:

```

```
PID TTY TIME CMD
There are no processes for this account currently running.
```

Step #3 - Find files on system belonging to user account

Creating a report of all files owned by consultant.

It is recommended that you backup/archive these files,
and then do one of two things:

- 1) Delete the files
- 2) Change the files' ownership to a current user account.

Please wait. This may take a while...

Report is complete.

Name of report: consultant_Files_110123.rpt
Location of report: /home/Christine

Step #4 - Remove user account

Do you wish to remove consultant's account from system? [y/n] y

User account, consultant, has been removed
\$

它能正常工作了。现在你已经有了一个脚本工具来在你需要删除用户账户时帮助你。更好的一点是你可以修改它来满足组织的需要。

26.4 小结

本章很好地利用了本书中介绍的一些shell脚本编程信息来创建Linux实用工具。在你负责Linux系统时，不管它是大型多用户系统，还是你自己的系统，你都要考虑很多事情。不用手动运行命令，你可以创建shell脚本工具来替你完成工作。

本章首先演示了如何用du命令来确定磁盘空间使用情况。然后我们用sed和gawk命令来提取数据中的特定信息。将命令的输出传给sed和gawk来分析数据是shell脚本中的一个常见功能，所以最好知道怎么做。

接下来的一节带你逐步了解使用shell脚本来归档和备份Linux系统上的数据文件。tar命令是归档数据的常用命令。本章演示了如何在shell脚本中用它来创建归档文件，以及如何在归档目录中管理归档文件。

我们以介绍使用shell脚本来删除用户账户的4个步骤结束本章。为脚本中重复的shell代码创建函数会让代码阅读和修改起来更清爽。这个脚本由多个不同的结构化命令组成，例如case和while命令。本章还介绍了给cron表用的脚本和交互式脚本在脚本结构上的差异。

现在，我们可以继续编写更高级的shell脚本，来帮助解决更复杂的Linux系统问题。

本章内容

- 监测系统统计数据
- 问题跟踪数据库

在 本章中，我们会探讨一下shell脚本中用到的高级方法。这些方法使你能用多种不同方法来处理系统上的脚本。

27.1 监测系统统计数据

在bash shell中有许多工具可以帮你监测Linux系统的性能和资源使用情况。遗憾的是，很难找到空闲使用它们，尤其是当你要管理几个系统时。编写一些脚本来用这些工具的输出生成报告，这是最节省时间的方法。

本节将会带你逐步编写一些高级shell脚本来帮助你监测各种系统性能的统计数据。

27.1.1 系统快照报告

了解系统性能和资源使用情况的一条好途径是查看快照报告。快照报告是特定时间点由系统的统计数据构成的图。这种报告是系统健康状况的一个“执行摘要”。

生成这个快照报告的脚本可以在一天中运行很多次，只要你需要图。甚至，这个脚本还会将报告通过E-mail发送给你。

1. 需要的功能

你会需要使用4条不同的bash shell命令来生成快照报告：uptime、df、free和ps。这些不同的命令会给你需要的统计数据。

● 运行时间

第一个命令uptime是最基本的系统统计命令：

```
$ uptime  
14:15:23 up 1 day, 5:10, 3 users, load average: 0.66, 0.54, 0.33
```

uptime命令会提供一些我们要用到的不同基本信息：

- 当前时间；
- 系统运行的天数、小时数和分钟数；
- 当前登录到系统上的用户数；
- 一分钟、五分钟、十五分钟的平均负载。

系统的运行时间是快照报告中的重要统计数据。但抓取这个统计数据时可能会有个问题。你的系统可能只开启了几分钟，也可能已经运行几天了。（传说有些Linux服务器甚至运行了几年！）下面是uptime命令在运行了几个小时的系统上的输出：

```
$ uptime
18:00:20 up 8:55, 3 users, load average: 0.62, 0.45, 0.37
```

下面是uptime命令在运行了几天的系统上的输出：

```
$ uptime
13:29:32 up 3 days, 4:24, 4 users, load average: 1.44, 0.83, 0.46
```

如你能看到的，从这个输出中获得正确的统计数据还是比较烦琐的。gawk命令（参见第16章）可以为运行了几天的系统完成这个任务，没有任何问题：

```
$ uptime | gawk '{print $2,$3,$4,$5}'
up 3 days, 4:27.
```

而下面是当同样的gawk命令用在只运行了几个小时的系统上时产生的输出：

```
$ uptime | gawk '{print $2,$3,$4}'
up 8:59, 3 users.
```

要解决这个难题，你可以用gawk的一个更高级的功能——if语句（参见第19章）。如果系统运行了几天，gawk变量\$4会含有单词days。要测试\$4是否为字符串days，gawk命令应该看起来如下：

```
$ uptime |
> gawk '{if ($4 == "days") {print $2,$3,$4,$5} else {print $2,$3}}'
up 9:23.
```

你可能会遇到系统只运行了一天的情况。因此，你还需要测试字符串day。可用布尔操作符OR (||) 达到这个目的。这样，现在gawk命令看起来如下：

```
gawk '{if ($4 == "days" || $4 == "day") {print $2,$3,$4,$5} else {print $2,$3}}'
```

怎么处理输出中这些没处理掉的逗号呢？使用sed命令，你可以简单地将它们替换成空格：

```
$ uptime | sed -n '/,/s// /gp' |
> gawk '{if ($4 == "days" || $4 == "day") {print $2,$3,$4,$5}
> else {print $2,$3}}'
up 9:32
```

这样看起来好多了。为了确认一下，让我们在运行多天的系统上测试这条命令：

```
$ uptime | sed -n '/,/s// /gp' |
> gawk '{if ($4 == "days" || $4 == "day") {print $2,$3,$4,$5}
> else {print $2,$3}}'
up 3 days 4:34
```

这样看起来也不错。现在我们来看看另一个创建系统快照报告要用到的性能工具命令。

● 磁盘使用情况

创建快照报告的下一条命令是df。df命令会说明磁盘空间使用情况的统计数据：

```
$ df
Filesystem 1K-blocks Used Available Use% Mounted on
/dev/sda2 307465076 185930336 105916348 64% /
none 503116 240 502876 1% /dev
none 508716 252 508464 1% /dev/shm
none 508716 100 508616 1% /var/run
...
```

df命令给出了系统上所有物理磁盘和虚拟磁盘的当前磁盘空间统计数据。你可以将单个磁盘的名称传给df命令来提取只跟那个磁盘有关的信息。为了让信息更加易读，可以给这个命令加一个-h选项（人类可读形式）：

```
$ df -h /dev/sda2
Filesystem Size Used Avail Use% Mounted on
/dev/sda2 294G 178G 102G 64% /
$
```

sed和gawk命令再次派上用场，为报告分析和打印需要的信息。由于我们不需要包含df命令输出的标题行，所以可以用sed来查找数据行中独有的东西。字符串% /只能在数据行中找到。但sed会将斜线当成它的命令结构中的一部分。要解决这个问题，可以在斜线的前面放置转义字符\。这会允许sed将斜线当成字符串的一部分而正确进行查找。最后，使用gawk来打印磁盘使用的百分比，在gawk变量中是\$5。整个命令行看起来如下：

```
$ df -h /dev/sda2 | sed -n '/% \//p' | gawk '{print $5}'
64%
```

这能非常好地工作了。现在我们去获取系统内存的快照信息。

● 内存使用情况

你可以用几个不同的bash shell命令来获取内存的统计数据。为了达到我们的目的，这里我们会专注于free命令。

free命令会显示物理内存的总量，以及其中多少是空闲的，多少是分配了的。它还会为交换内存显示同样的数据，包括内核缓冲区。这里有个free命令输出的例子：

```
$ free
total used free shared buffers cached
Mem: 1017436 890148 127288 0 31536 484100
-/+ buffers/cache: 374512 642924
Swap: 200776 0 200776
$
```

由于我们提取了磁盘使用情况统计数据中的百分比，最好也为内存使用情况做同样的事情。但你会注意到free命令并没有提供任何使用情况的百分比。这不是问题，用gawk命令，你可以计算出一个百分比。

free命令的输出已经用空格划分好了，所以gawk可以方便地用它们的变量名来引用这个计算

要用到的两个字段。`$2`变量是总的可用内存，`$3`变量是总的已使用内存。用`$3`除以`$2`，gawk就能为你提供使用内存的百分比了：

```
$ free | sed -n '2p' | gawk 'x = ($3 /$2) {print x}'  
0.87165
```

注意，命令中用`sed`来从`free`命令的输出中只提取数据的第二行，它会允许你丢掉标题行。生成的数字仍然不是报告需要的格式。你可以通过将这个数乘以100然后用`gawk`的整数函数`int`，来清理一下这个百分比。最后，用`sed`加上一个百分号：

```
$ free | sed -n '2p' |  
> gawk 'x = int(($3 /$2)* 100) (print x)' |  
> sed '$s/$%/'  
87%
```

现在内存的统计数据已经符合你要的格式了。让我们接着看最后一个要包含的东西——僵尸！

● 僵尸进程

Linux系统上的僵尸是指处于未知状态的进程。这些进程已经完成了它的工作，但因为种种原因还处于未完成的状态。因此，跟它的名字一样，僵尸进程既没死掉，也没在运行。

虽然有一两个僵尸进程在Linux系统上不是什么大问题，但多了就可能会带来麻烦。僵尸进程会占用着进程ID，这个进程ID不会释放到进程ID表中，直到僵尸进程被终止。因此，另一个要观察的系统统计数据是当前的僵尸进程数。

这可以用`ps`命令来轻松实现（参见第4章）。带`-al`选项的`ps`命令会返回系统上所有的进程以及它们的进程状态。状态`Z`表明进程是个僵尸：

```
$ ps -al  
F S  UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  
0 T 1000 2174 2031 0 80 0 - 2840 signal pts/0 00:00:00 mail  
0 T 1000 2175 2031 0 80 0 - 2839 signal pts/0 00:00:00 mail  
1 Z 1000 8779 8797 0 80 0 - 0 exit pts/0 00:00:00 zomb <defunct>  
0 R 1000 8781 8051 0 80 0 - 1094 - pts/5 00:00:00 ps  
$
```

这里，`gawk`命令会再次帮到你。将`ps`命令的输出通过管道传给`gawk`，你只要打印出字段`$2`和`$4`来抓取进程ID和它的进程状态。但你只想显示僵尸进程，所以用`grep`命令搜索输出就可以了。需要的命令如下：

```
$ ps -al | gawk '{print $2,$4}' | grep Z  
Z 8779
```

这会为快照报告很好地工作。现在，你已经有了所有的片段，让我们将它们拼成一个可以工作的完整脚本。

2. 创建快照脚本

你要建立的第一项是脚本主体中要用的各种变量。这些变量应该放在脚本的顶部，以便于将来修改。

我们的快照报告会需要一个日期截，所以会包含一个日期变量`DATE`。对于性能统计数据，你可能要收集不止一个磁盘的统计数据。因此，数组变量`DISKS_TO_MONITOR`会含有系统上所有磁盘

的清单。由于不同的管理员会设置不同的邮件工具，让我们加一个邮件变量MAIL来指定这里用哪个工具（参见第25章）。还有，因为某些系统的mail命令可能会位于罕见的位置，你应该使用which命令（参见第23章）找出mail命令的位置。将报告发给谁会在变量MAIL_TO中指定。最后，报告的名称在变量REPORT中设定。

快照脚本的变量部分看起来如下：

```
# Set Script Variables
#
DATE=`date +%m%d%Y`
DISKS_TO_MONITOR="/dev/sda1 /dev/sda2"
MAIL=`which mutt`
MAIL_TO=user
REPORT=/home/user/Documents/$DATE.rpt
```

由于你正在同一个脚本来生成报告并发送出去，你需要保存文件描述符，并在后面恢复（参见第14章）。保存文件描述符和将输出重定向到STDOUT的shell命令看起来如下：

```
# Create Report File
#
exec 3>&1 #Save file descriptor
#
exec 1> $REPORT #direct output to rpt file.
```

下面的代码，放在mail命令之前，会将STDOUT恢复为已保存的文件描述符：

```
# Restore File Descriptor & Mail Report
#
exec 1>&3 #Restore output to STDOUT
```

将所有的组件放到一起，现在脚本看起来如下：

```
#!/bin/bash
#
# Snapshot_Stats - produces a report for system stats
#####
# Set Script Variables
#
DATE=`date +%m%d%Y`
DISKS_TO_MONITOR="/dev/sda1 /dev/sda2"
MAIL=`which mutt`
MAIL_TO=user
REPORT=/home/user/Documents/Snapshot_Stats_$DATE.rpt
#
#####
# Create Report File
#
exec 3>&1 #Save file descriptor
#
exec 1> $REPORT #direct output to rpt file.
#
#####
#
```

echo

```

echo -e "\t\tDaily System Report"
echo
#
#####
# Date Stamp the Report
#
echo -e _Today is _ `date +%m/%d/%Y`
echo
#
#####
# 1) Gather System Uptime Statistics
#
echo -e _System has been \c"
uptime | sed -n ./././ /gp' |
    gawk .{if ($4 == „days“ || $4 == „day“)
        {print $2,$3,$4,$5}
        else {print $2,$3}}'
#
#####
# 2) Gather Disk Usage Statistics
#
echo
for DISK in $DISKS_TO_MONITOR      #loop to check disk space
do
    echo -e "$DISK usage: \c"
    df -h $DISK | sed -n '/% //p' | gawk '{print $5}'
done
#
#####
# 3) Gather Memory Usage Statistics
#
echo
echo -e _Memory Usage: \c"
#
free | sed -n .2p' |
    gawk .x = int(($3 / $2) *100) {print x}' |
    sed .s/$%/'
#
#####
# 4) Gather Number of Zombie Processes
#
echo
ZOMBIE_CHECK='ps -al | gawk .{print $2,$4}' | grep Z'
#
if [ „$ZOMBIE_CHECK“ = „“ ]
then
    echo „No Zombie Process on System at this Time“
else
    echo „Current System Zombie Processes“
    ps -al | gawk .{print $2,$4}' | grep Z
fi
echo
#####
# Restore File Descriptor & Mail Report

```

```

#
exec 1>&3      #Restore output to STDOUT
#
$MAIL -a $REPORT -s _System Statistics Report for $DATE"
-- $MAIL_TO < /dev/null
#
#####
# Clean up
#
rm -f $REPORT
#

```

注意，在脚本运行到步骤#4时，它首先用放到ZOMBIE_CHECK变量中的ps命令的结果检查了僵尸进程。如果变量为空，脚本会报告不存在僵尸进程；否则它会列出当前状态为Z的所有进程。

现在，测试一下这个新的快照报告脚本，看看它运行得怎样：

```

$ ./Snapshot_Stats

```

这里没看到太多的东西。但如果你打开了E-mail客户端（本例中是Mutt），你会看到如图27-1所示的快照报告输出。

```

File Edit View Search Terminal Help
[...]
[Attachment #2: Snapshot_Stats_12122010.rpt --]
[Attachment #2: Snapshot_Stats_12122010.rpt --]

Daily System Report
Today is 12/12/2010
System has been up 3 days 4:44
/dev/sda1 usage: 1%
/dev/sde2 usage: 64%
Memory Usage: 94%
Current System Zombie Processes
Z 8894
[...]
Bottom of message is shown.

```

图27-1 在Mutt中显示快照报告

这个快照报告为你提供了系统的整体健康状况，非常有用。但这类报告没法提供随着时间推移性能和资源使用情况的趋势。因此，让我们继续写一组脚本来提供那类信息。

27.1.2 系统统计数据报告

所有Linux系统的核心统计数据都是CPU和内存的使用情况。如果这些值开始失去控制，事情很快就会变得很糟。本节将介绍如何编写脚本来帮你监测和跟踪一段时间内Linux系统上的CPU和内存的使用情况。

1. 需要的功能

第一个脚本的目的是将性能统计数据收集到一个数据文件中供后面使用。因此，你需要确定你到底要用脚本收集哪些数据以及用什么命令。

对于这个脚本，我们将会再次使用uptime命令。不过这次不是提取系统运行的时间，而是提取系统上用户的数量。用户数据会直接影响CPU和内存的使用。通过sed，我们会在uptime命令的输出文本中查找单词users，在当前行中定位到这个单词后，用sed删掉该文本行中的剩余部分，包括单词users。这样会将你要的统计数据，系统上的用户数，作为该行最后一部分内容留下。要获得最后一项数据内容，你可以用gawk变量NF。整个命令行及其输出看起来如下：

```
$ uptime | sed 's/users.*$/ /' | gawk '{print $NF}'
4
```

系统负载是另一个不错的统计数据，你可以从uptime命令的输出中解析出来。系统负载说明，平均起来，系统上的CPU有多忙。平均负载为1说明单个CPU一直在忙。但如果系统的系统上有两个CPU，这意味着每个都只有一半那么忙。来自uptime命令的系统负载是按最后1分钟、最后5分钟和最后15分钟的平均值来说的。我们会用最后15分钟的系统平均负载数据，它位于输出文本行的末尾：

```
$ uptime | gawk '{print $NF}'
0.19
```

另一个很好的提取系统信息的命令是vmstat。这里有个vmstat命令输出的例子：

```
$ vmstat
procs      memory          swap      io      system      cpu
r b   swpd   free   buff   cache   si   so   bi   bo   in   cs   us   sy   id   wa
0 0   11328  42608 165444 502500    0   0     1   2   68   10   1   0 99   0
```

你第一次运行vmstat命令时，它会显示自上次重启以来的平均负载值。要得到当前统计数据，你必须加命令行参数来运行vmstat命令：

```
$ vmstat 1 2
procs      memory          swap      io      system      cpu
r b   swpd   free   buff   cache   si   so   bi   bo   in   cs   us   sy   id   wa
0 0   11328  43112 165452 502444    0   0     1   2   68   10   1   0 99   0
0 0   11328  40540 165452 505064    0   0     0   0   58  177   1   1 98   0
```

第二行含有Linux系统的当前统计信息。如你所能看到的，vmstat命令的输出看起来有点神秘。表27-1解释了每个符号代表什么意思。

表27-1 vmstat的输出符号

符 号	描 述
r	等待CPU时间的进程数
b	处于不可中断休眠中的进程数
swpd	使用的虚拟内存总量（单位：MB）
free	空闲的物理内存总量（单位：MB）

(续)

符 号	描 述
buff	用作缓冲区的内存总量(单位: MB)
cache	用作高速缓存的内存总量(单位: MB)
si	从磁盘交换进来的内存总量(单位: MB)
so	交换到磁盘的内存总量(单位: MB)
bi	从块设备收到的块数
bo	发送给块设备的块数
in	每秒的CPU中断次数
cs	每秒的CPU上下文切换次数
us	用于执行非内核代码的CPU时间所占的百分比
sy	用于执行内核代码的CPU时间所占的百分比
id	处于空闲状态的CPU时间所占的百分比
wa	用于等待I/O的CPU时间所占的百分比

这里有很多信息。可用(空闲)内存和处于空闲状态的CPU时间所占的百分比是我们想要的统计数据。

你可能已经注意到了vmstat命令的输出含有表头信息，显然你不想在数据中包含这个。要解决这个问题，使用sed命令来只显示含有数字值的行：

```
$ vmstat 1 2 | sed -n '/[0-9]/p'
1 0 11328 38524 165476 506548 0 0 1 2 68 10 1 0 99 0
0 0 11328 35820 165476 509528 0 0 0 0 82 160 1 1 98 0
```

好多了，但现在你需要的只是第二行数据。再调用一次sed可以解决这个问题：

```
$ vmstat 1 2 | sed -n '/[0-9]/p' | sed -n '2p'
0 0 11328 36060 165484 509560 0 0 0 0 58 175 1 1 99 0
```

现在你可以用gawk轻松地提取你要的数据值了。

最后，你会想用日期和时间戳来标记每个性能数据记录，以说明这些统计数据是什么时候抓取的。简单地用date命令，指定数据记录的格式：

```
$ date +"%m/%d/%Y %k:%M:%S"
12/12/2010 13:55:31
```

现在考虑一下你想如何记录性能数据记录。对于定期采样的数据，通常最好是直接将数据输出到日志文件中。你可以在\$HOME目录创建这个文件，将每次运行shell脚本得到的数据附加上去。当你要看结果时，查看这个日志文件就可以了。

你会想保证日志文件中的数据可以被方便地读取。你可以用许多不同方法来格式化日志文件中的数据。常见的格式是逗号分割文件(CSV)。这种格式会将每个数据记录放到单独一行，并用逗号来分隔记录中的数据字段。这是一种流行的格式，因为很容易将它导入到电子表格、数据库和报表工具中。

这个脚本会将数据保存在一个CSV格式的文件中。我们在本章后面创建的报告脚本会使用这个文件。

2. 创建捕捉脚本

一旦所有需要的功能就位，用来抓取数据的这个脚本就非常简单了。它看起来如下：

```
#!/bin/bash
#
# Capture_Stats - Gather System Performance Statistics
#####
# Set Script Variables
#
REPORT_FILE=/home/user/Documents/capstats.csv
DATE=`date +%m/%d/%Y`
TIME=`date +%k:%M:%S`
#
#####
# Gather Performance Statistics
#
USERS=`uptime | sed 's/users.*$/ /' | gawk '{print $NF}'``
LOAD=`uptime | gawk '{print $NF}'``
#
FREE=`vmstat 1 2 | sed -n '/[0-9]/p' | sed -n '2p' |
gawk '{print $4}'``
IDLE=`vmstat 1 2 | sed -n '/[0-9]/p' | sed -n '2p' |
gawk '{print $15}'``
#
#####
# Send Statistics to Report File
#
echo "$DATE,$TIME,$USERS,$LOAD,$FREE,$IDLE" >> $REPORT_FILE
#
```

这个脚本会解析来自`uptime`和`vmstat`命令的统计数据，将它们放进变量。然后这些变量被写到了日志文件`REPORT_FILE`中，用逗号分隔，还加了日期和时间戳。注意数据是用`>>`重定向符号来附加到这个日志文件的。这会允许你继续向日志文件添加任意长的数据，只要你觉得有必要。

在创建了这个脚本后，你可能应该在命令行上测试一下，然后再通过`cron`表来让它定期运行：

```
$ cat capstats.csv
12/09/2010, 9:06:50,2.0.29,645988.99
12/09/2010, 9:07:55,2.0.28,620252.100
12/09/2010, 9:40:51,3.0.37,474740.100
12/10/2010,14:36:46,3.0.30,46640.98
12/12/2010, 7:16:26,4.0.25,27308.98
12/12/2010,13:28:53,4.0.42,58832.100
```

下一个脚本将会从CSV数据创建一份报告并将它通过E-mail发送给相应的人。由于报告脚本会是一个单独的脚本，你可以让`cron`在跟`Capture_Stats`脚本不同的时间运行它。这样你就可以灵活安排抓取多少数据后再发报告。

3. 生成报告脚本

有了一份全是原始数据的文件，可以开始创建脚本来生成一份好看的报告了。用来生成报告

脚本的最好工具是gawk命令。

gawk命令允许从CSV文件中提取原始数据并将它呈现为想要的形式。首先，用Capture_Stats脚本新建的capstats.csv文件从命令行上测试这个命令：

```
$ cat capstats.csv |
> gawk -F, '{printf "%s %s - %s\n", $1, $2, $4}'
12/09/2010 9:06:50 - 0.29
12/09/2010 9:07:55 - 0.28
12/09/2010 9:40:51 - 0.37
12/10/2010 14:36:46 - 0.30
12/12/2010 7:16:26 - 0.25
12/12/2010 13:28:53 - 0.42
```

你需要给gawk命令使用-F选项来将逗号定义为数据的字段分隔符。之后，你可以提取每个单独的数据字段，并根据需要用printf功能显示它。

对于报告，我们会使用HTML格式。HTML成为格式化Web页面的标准方法很多年了。它使用简单的标签来描述Web页面中的数据类型。不过，HTML不只用于Web页面。你也会经常发现HTML用邮件消息中。你能或不能查看内嵌HTML的E-mail文档，取决于你的邮件工具（参见第25章）。较好的解决办法是创建HTML报告并将它作为E-mail消息的附件发送。

使用HTML创建报告允许你用很小的工作量生成精美的格式化报告。显示报告的程序会完成格式化和显示报告的苦差。你所要做的就是插入适当的HTML标签来格式化数据。

在HTML中显示电子表格一类的数据的最简单方法是用<table>标签。表格标签允许用行和单元格来创建表格，在HTML中称为“分区”（division）。你可以用<tr>标签来定义行的起始，用</tr>标签来定义行的结尾。类似地，你可以用<td>和</td>标签对来定义单元格。

完整表格的HTML如下：

```
<html>
<body>
<h2>Report title</h2>
<table border="1">
<tr>
  <td>Date</td><td>Time</td><td>Users</td>
  <td>Load</td><td>Free Memory</td><td>%CPU Idle</td>
</tr>
<tr>
  <td>12/09/2010</td><td>11:00:00</td><td>4</td>
  <td>0.26</td><td>57076</td><td>87</td>
</tr>
</table>
</body>
</html>
```

每个数据行都是<tr>...</tr>标签对的一部分。每个数据字段都在自己的<td>...</td>标签对中。

当你在浏览器中显示HTML报告时，浏览器会为你自动创建这个表格，如图27-2所示。

对脚本来说，你要做的就是用echo命令来生成HTML标题代码，用gawk命令来生成数据的HTML代码，然后再用echo命令给表格收尾。

The screenshot shows a web browser window with the URL `file:///tmp/capstats.html`. The page title is "Report for 12/12/2010". Below the title is a table with the following data:

Date	Time	Users	Load	Free Memory	%CPU Idle
12/09/2010	9:06:50	2	0.29	645988	99
12/09/2010	9:07:55	2	0.28	620252	100
12/09/2010	9:40:51	3	0.37	474740	100
12/10/2010	14:36:46	3	0.30	46640	98
12/12/2010	7:16:26	4	0.25	27308	98
12/12/2010	13:28:53	4	0.42	58832	100

图27-2 在HTML表格中显示数据

下面是Report_Stats脚本，它会生成性能报告并将其用邮件发出：

```
#!/bin/bash
#
# Report_Stats - Generates Rpt from Captured Perf Stats
#####
# Set Script Variables
#
REPORT_FILE=/home/user/Documents/capstats.csv
TEMP_FILE=/home/user/Documents/capstats.html
#
DATE=`date +%m/%d/%Y`
#
MAIL='`which mutt`'
MAIL_TO=user
#
#####
# Create Report Header
#
echo "<html><body><h2>Report for $DATE</h2>" > $TEMP_FILE
echo "<table border='1'>" >> $TEMP_FILE
echo "<tr><td>Date</td><td>Time</td><td>Users</td>" >> $TEMP_FILE
echo "<td>Load</td><td>Free Memory</td><td>%CPU Idle</td></tr>" >>
$TEMP_FILE
#
#####
# Place Performance Stats in Report
#
cat $REPORT_FILE | gawk -F, '{
printf "<tr><td>%s</td><td>%s</td><td>%s</td><td>%s</td>", $1, $2, $3;
printf "<td>%s</td><td>%s</td><td>%s</td></tr>\n", $4, $5, $6;
}' >> $TEMP_FILE
#
```

```

echo "</table></body></html>" >> $TEMP_FILE
#
#####
# Mail Performance Report & Clean up.
#
$MAIL -a $TEMP_FILE -s "Performance Report $DATE"
-- $MAIL_TO < /dev/null
#
rm -f $TEMP_FILE
#

```

我们用mutt来通过邮件发出性能报告，而你可以将它改为本地的E-mail客户端。临时HTML报告文件的名称也可以修改以更好地符合需要。

警告 大部分E-mail客户端能通过文件扩展名自动检测附件的类型。出于这个原因，你应该报告文件名的末尾加上.html。

4. 运行脚本

在创建了Report_Stats脚本后，尝试从命令行上运行它，看看会发生什么：

```

$ ./Report_Stats

```

好吧，没什么特别的。真正的测试是查看你的邮件消息，最好是在图形化E-mail客户端中，例如KMail或Evolution。图27-3演示了如何在Evolution E-mail客户端上查看消息。注意，数据用HTML表格格式化得多好看，就跟在浏览器中查看时一样。

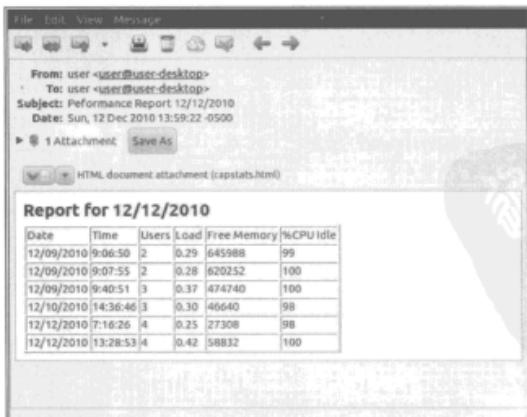


图27-3 在Evolution中查看报告附件

现在你已经有脚本来生成两份不同的报告，帮你追踪系统性能并避免由资源紧张等造成的问题。但即使是优秀的系统监测也无法阻止意料之外的问题发生。快速解决这些意料之外的问题、从中吸取教训会有助于以后最小化它们的影响。因此，在下一节我们会带你逐步创建支持跟踪问题的高级数据库脚本。

27.2 问题跟踪数据库

每个系统都注定要经历一些意外情况。但对问题的分析允许你作一些改变，以最小化以后未知的困难。它还可以帮忙记录你是如何解决问题的，以防再次发生。当你和一个团队一起工作时，好处会更大：大家可以记录下每个人经历的问题和解决办法。问题/解决办法等信息会更容易在团队成员之间分享。

在本节中，我们会带你经历建立一个基本的问题跟踪数据库的过程。我们还会帮你创建一些高级脚本来记录、更新和回顾这些问题及其解决办法。

27.2.1 创建数据库

规划数据库的结构是创建问题跟踪数据库中最重要的部分。你需要确定你要跟踪什么样的信息，以及谁有修改和使用存储在数据库中的数据的权限。

这个数据库的最基本目的是记录问题以及它是如何被解决的，这样你就可以用这些信息来更快地解决以后类似的问题。为了达到这个目的，你应该跟踪如下信息：

- 报告问题的日期；
- 解决问题的日期；
- 问题的描述及症状；
- 问题解决办法的描述。

我们会使用mysql（参加第23章）作为问题跟踪数据库的数据库系统。要开始，你得先创建一个空数据库：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.

mysql> CREATE DATABASE Problem_Trek;
Query OK, 1 row affected (0.02 sec)

mysql>
```

有了创建的空数据库，现在可以加入表了。我们会创建一个有4个字段来存储问题信息的表，problem_logger。另外，你还要加个字段id_number来作为数据库的主键（primary key）。表27-2列出了需要的5个字段。

表27-2 problem_logger表的字段

字段名	数据类型
id_number	整数
report_date	日期
fixed_date	日期
prob_symptoms	文本
prob_solutions	文本

在mysql中创建这个表相当简单：

```
mysql> USE Problem_Trek;
Database changed
mysql>
mysql> CREATE TABLE problem_logger (
-> id_number int not null,
-> report_date Date,
-> fixed_date Date,
-> prob_symptoms text,
-> prob_solutions text,
-> primary key (id_number));
Query OK. 0 rows affected (0.04 sec)
```

```
mysql>
```

注意，我们使用了USE命令来保证我们是在正确的数据库中创建新表。让我们再用DESCRIBE命令来看一下新的problem_logger表。以下是显示加到表中的字段的一个简单明了的方法：

```
mysql> DESCRIBE problem_logger;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id_number | int(11) | NO | PRI | NULL | 
| report_date | date | YES | | NULL | 
| fixed_date | date | YES | | NULL | 
| prob_symptoms | text | YES | | NULL | 
| prob_solutions | text | YES | | NULL | 
+-----+-----+-----+-----+-----+
5 rows in set (0.01 sec)
```

```
mysql>
```

到目前为止，我们需要用根账户来创建数据库和表。现在这些步骤完成了，对于这里的工作来说，根账户不再是必要的了。如你在第23章中了解到的，在数据库脚本中使用根账户并不是一个好的实践。因此，你需要给数据库加一个有适当权限的新账户。这个账户会是脚本用来访问Problem_Trek数据库的那个：

```
mysql> GRANT SELECT,INSERT,DELETE,UPDATE ON Problem_Trek.* TO
-> cbres IDENTIFIED BY 'test_password';
Query OK. 0 rows affected (0.03 sec)
```

```
mysql>
```

给数据库加了新的用户账户后，一定要测试一下：

```
$ mysql Problem_Trek -u cbres -p
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor. Commands end with ; or \g.
...
mysql>
```

新账户可以顺利工作。现在，只要一步，你就作好准备，可以开始整理这个数据库脚本需要的功能了。用你最喜欢的文本编辑器，创建特殊配置文件\$HOME/.my.cnf，它会包含你之前为新用户账户设置的密码。这会允许脚本访问mysql程序而不用在脚本中包含密码。

```
$ cat $HOME/.my.cnf
[client]
password=test_password
$
```

不要忘了限定它只能由你访问：

```
$ chmod 400 $HOME/.my.cnf
```

现在已经建好Problem_Trek数据库，并为它设置了适当的权限。你可以开始准备数据库脚本了。

27.2.2 记录问题

跟踪问题的第一步是记录它的症状。由于用mysql命令来记录问题会很耗时，我们用一个易用的脚本来封装这个过程，从而简化它。

1. 需要的功能

要将问题的信息放到你的problem_logger表中，可以用mysql的INSERT命令（参见第23章）。为了将新记录插入到problem_logger表中，调用这些命令：

```
$
$ mysql Problem_Trek -u cbres -p
Enter password:
...
mysql> INSERT INTO problem_logger VALUES (
-> 1012111322,
-> 20101211,
-> 0,
-> "When trying to run script Capture_Stats, getting message: bash:
./Capture_Stats: Permission denied",
-> "");
Query OK, 1 row affected (0.02 sec)

mysql>
```

非常简单。你可以用SELECT命令看一下你刚放到表中的记录。通过将新记录的ID号指定给WHERE，只有一个输入的记录会被显示：

```
mysql> SELECT * FROM problem_logger WHERE id_number=1012111322;
+-----+-----+-----+-----+
| id_number | report_date | fixed_date | prob_symptoms | prob_solutions |
+-----+-----+-----+-----+
| 1012111322 | 2010-12-11 | 0000-00-00 | When trying to run script Capture_Stats, getting message: bash: ./Capture_Stats: Permission denied |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
mysql>
```

虽然你可以看到输入的记录，但用这种格式很难看懂数据。为了让记录更容易看懂，可以在命令的末尾加上\G而不是分号(;)：

```
mysql> SELECT * FROM problem_logger WHERE id_number=1012111322\G
***** 1. row *****
id_number: 1012111322
report_date: 2010-12-11
fixed_date: 0000-00-00
prob_symptoms: When trying to run script Capture_Stats,
               getting message: bash: ./Capture_Stats: Permission denied
prob_solutions:
1 row in set (0.00 sec)

mysql>
```

这看起来好多了，也能在脚本中很好地工作。这两个mysql命令是这个脚本需要的主要功能。这样，我们可以开始构建Record_Problem脚本了。

2. 创建脚本

在第23章中，你知道了使用which命令来在系统上查找mysql命令并将它赋给环境变量是明智之举。你可以再进一步将mysql要用的数据库名和用户账户也加上：

```
#
MYSQL=`which mysql` Problem_Trek -u cbres
#
```

Record_Problem是个交互式脚本。为了让它保持简单，你可以提问来获取需要的信息。在脚本得到回答时，它会将它们赋给各种变量。一旦所有的数据都收集齐了，你可以用变量名和INSERT命令一起将信息输入到数据库中。这样，脚本用户就没必要懂得需要的mysql命令了：

```
INSERT INTO problem_logger VALUES (
    $ID_NUMBER,
    $REPORT_DATE,
    $FIXED_DATE,
    "$PROB_SYMPTOMS",
    "$PROB_SOLUTIONS");
```

ID_NUMBER变量会用作表中的主键字段。由于它是主键字段，每个记录过的问题的ID号必须是唯一的。为了让它唯一，你可以用当前日期和时间的组合来生成需要的10位键值：

```
#
ID_NUMBER=`date +%y%m%d%H%M`
#
```

由于这个脚本只是用来记录问题的，`FIXED_DATE`和`PROB_SOLUTIONS`变量不必含有数据。因此，在将记录插到表中之前，将这两个变量设为相应的空值：

```
#  
# Set Fixed Date & Problem Solution to null for now  
FIXED_DATE=0  
PROB_SOLUTIONS=""  
#
```

为了显示成功添加了的问题记录，给脚本加一条`SELECT`语句。为了避免显示整个数据库的内容，`WHERE`命令用`ID_NUMBER`变量来只选择刚添加的记录：

```
SELECT * FROM problem_logger WHERE id_number=$ID_NUMBER\G
```

将所有这些部分放到一起，就生成了如下的脚本：

```
#!/bin/bash  
#  
# Record_Problem - records system problems in database  
#####
# Determine mysql location & put into variable  
#  
MYSQL=`which mysql`" Problem_Trek -u cbres"  
#  
#####  
# Create Record Id & Report_Date  
#  
ID_NUMBER=`date +%Y%m%d%H%M`  
#  
REPORT_DATE=`date +%Y%m%d`  
#  
#####  
# Acquire information to put into table  
#  
echo  
echo -e "Briefly describe the problem & its symptoms: \c"  
#  
read ANSWER  
PROB_SYMPTOMS=$ANSWER  
#  
# Set Fixed Date & Problem Solution to null for now  
FIXED_DATE=0  
PROB_SOLUTIONS=""  
#  
#####  
# Insert acquired information into table  
#  
#  
echo  
echo "Problem recorded as follows:"  
echo  
$MYSQL <<EOF  
INSERT INTO problem_logger VALUES (  
 $ID_NUMBER,
```

```
$REPORT_DATE,
$FIXED_DATE,
"$PROB_SYMPOMS",
"$PROB_SOLUTIONS");
SELECT * FROM problem_logger WHERE id_number=$ID_NUMBER\G
EOF
#
#####

```

现在，测试一下Record_Problem脚本，看它是否已经可以使用了：

```
$ ./Record_Problem
Briefly describe the problem & its symptoms:
Running yum to install software and
getting 'not found' message.

Problem recorded as follows:
*****
1. row *****
id_number: 1012111510
report_date: 2010-12-11
fixed_date: 0000-00-00
prob_symptoms:
Running yum to install software and getting 'not found' message.
prob_solutions:
$
```

它能够正常工作了。你可以看出，相比用单独的mysql命令来将问题插入到数据库中，用这样的脚本来记录问题简单太多了。

既然你已经有了记录问题的方法，你需要一个更新记录的方法。让我们开始创建Update_Problem脚本。

27.2.3 更新问题

跟踪问题的下一步是记录问题的解决办法。本节中创建的脚本会允许向问题记录添加这样的信息。

1. 需要的功能

使用mysql来更新记录非常简单。一旦获得了所有数据，你只要使用mysql的UPDATE命令就可以了：

```
UPDATE problem_logger SET
prob_solutions="$PROB_SOLUTIONS",
fixed_date=$FIXED_DATE
WHERE id_number=$ID_NUMBER;
```

非常简单。不过，将Update_Problem脚本的各部分放到一起相比使用UPDATE函数要多花一点工夫。

2. 创建脚本

跟前面的脚本一样，我们想让事情对脚本用户来说相当简单。要达到这个目标，还需要加一

些东西。

我们需要用Problem_Trek数据库的主键id_number来标识要更新的记录。如果知道ID号，那我们就能将它作为参数传给脚本了（参见第13章）。你需要做的就是检查参数，然后将它赋给变量ID_NUMBER。如果ID号没有作为参数传进来，脚本会向用户要它：

```
if [ $# -eq 0 ] #Check if id number was passed
then
    #if not passed ask for it.
#
...
echo
echo "What is the ID number for the"
echo -e "problem you want to update?: \c"
read ANSWER
ID_NUMBER=$ANSWER
else
    ID_NUMBER=$1
fi
```

但向用户要求问题ID号有点不公平，没几个人会记一个标识他们正在解决的某个问题的10位数字。因此，你应该提供一个数据库中各种待解决问题的列表，方便脚本用户查看。

首先，查看是否有待解决的记录。为空的字段是fixed_date和prob_solutions。如果这两个字段中任意一个为空，那么记录就需要更新。你可以在WHERE语句中用OR命令和SELECT一起查找这两个字段。完成这个任务的代码看起来如下：

```
RECORDS_EXIST=`$MYSQL -Bse "SELECT id_number FROM problem_logger
WHERE fixed_date='0000-00-00' OR prob_solutions=''"`
```

注意，这个命令的输出会赋给变量RECORDS_EXIST。为了做到这个，-Bse选项必须用在mysql命令上。它们允许命令批量（一次执行完）执行，然后退出。如果RECORDS_EXIST变量中含有数据，那么就有记录需要更新。用如下代码将它们显示在屏幕上：

```
#
if [ "$RECORDS_EXIST" != "" ]
then
echo
echo "The following record(s) need updating..."
$MYSQL <<EOF
SELECT id_number, report_date, prob_symptoms
FROM problem_logger
WHERE fixed_date='0000-00-00' OR
prob_solutions=""\G
EOF
fi
#
```

这会提供一个好看的未解决问题记录及其ID号列表。从列表中选择一个ID号比记住这些数据方便多了。

现在你可以将这些部分组成更新记录的整个脚本，方便使用。它看起来应该跟下面的类似：

```
#!/bin/bash
```

```

#
# Update Problem - updates problem record in database
#####
# Determine sql location & set variable
#
MYSQL='which mysql' " Problem_Trek -u cbres"
#
#####
# Obtain Record Id
#
if [ $# -eq 0 ] #Check if id number was passed
then           #if not passed ask for it.
#
#      Check if any unfinished records exist.
RECORDS_EXIST=~$MYSQL -Bse 'SELECT id_number FROM problem_logger
WHERE fixed_date="0000-00-00" OR prob_solutions=""'
#
if [ "$RECORDS_EXIST" != "" ]
then
echo
echo "The following record(s) need updating...""
$MYSQL <<EOF
SELECT id_number, report_date, prob_symptoms
    FROM problem_logger
    WHERE fixed_date="0000-00-00" OR
        prob_solutions=""\G
EOF
fi
#
echo
echo "What is the ID number for the"
echo -e "problem you want to update?: \c"
read ANSWER
ID_NUMBER=$ANSWER
else
    ID_NUMBER=$1
fi
#
#####
# Obtain Solution (aka Fixed) Date
#
echo
echo -e "Was problem solved today? (y/n) \c"
read ANSWER
#
case $ANSWER in
y|Y|YES|yes|Yes|yeS|yEs|yES )
#
    FIXED_DATE=`date +%Y%m%d`"
;;
*)
    # If answer is anything but yes, ask for date

```

```

echo
echo -e "What was the date of resolution? [YYYYMMDD] \c"
read ANSWER
#
# FIXED_DATE=$ANSWER
;;
esac
#
#####
# Acquire problem solution
#
echo
echo -e "Briefly describe the problem solution: \c"
#
read ANSWER
PROB_SOLUTIONS=$ANSWER
#
#####
# Update problem record
#
#
echo
echo "Problem record updated as follows:"
echo
$MYSQL <<EOF
UPDATE problem_logger SET
    prob_solutions="$PROB_SOLUTIONS",
    fixed_date=$FIXED_DATE
    WHERE id_number=$ID_NUMBER;
SELECT * FROM problem_logger WHERE id_number=$ID_NUMBER\G
EOF
#

```

测试一下这个脚本看看它运行得怎样：

```

$ ./Update_Problem
The following record(s) need updating...
***** 1. row *****
id_number: 1012111624
report_date: 2010-12-11
prob_symptoms: Running yum to install software and getting
'not found' message.

What is the ID number for the
problem you want to update?: 1012111624

Was problem solved today? (y/n) y

Briefly describe the problem solution: Network service was down.
Issued the command: service network restart

Problem record updated as follows:
***** 1. row *****
id_number: 1012111624

```

```

report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms: Running yum to install software and getting
'not found' message.
prob_solutions: Network service was down.
Issued the command: service network restart
$
```

它能很好地工作而且用起来相当容易。

3. 附加的修改

现在为了让这个过程对脚本用户来说更简单，让我们修改Record_Problem脚本来调用Update_Problem（如果需要的话）。有时，如果很忙的话，“批量”记录问题会更容易一些。在Record_Problem脚本的末尾，你可以加入如下代码：

```

# Check if want to enter a solution now
#
echo
echo -e "Do you have a solution yet? (y/n) \c"
read ANSWER
#
case $ANSWER in
y|Y|YES|yes|Yes|yeS|yEs|yES )
    ./${HOME}/scripts/Update_Problem $ID_NUMBER
#
;;
*)
# If answer is anything but yes, just exit script
;;
esac
#####
#####
```

现在让我们测试一下这个改动，看看它能否工作：

```

$ ./Record_Problem

Briefly describe the problem & its symptoms:
Moved script from Fedora to Ubuntu and was not working properly.

Problem recorded as follows:

*****
1. row *****
id_number: 1012111631
report_date: 2010-12-11
fixed_date: 0000-00-00
prob_symptoms:
Moved script from Fedora to Ubuntu and was not working properly.
prob_solutions:
Do you have a solution yet? (y/n) y

Was problem solved today? (y/n) y

Briefly describe the problem solution:
Added #!/bin/bash to the top of the script.
```

This is needed because Ubuntu's default shell is dash.

Problem record updated as follows:

```
***** 1. row *****
id_number: 1012111631
report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms:
Moved script from Fedora to Ubuntu and was not working properly.
prob_solutions: Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.
$
```

这能很好地工作。现在你已经让在Problem_Trek数据库中添加和更新记录变得很简单了。

27.2.4 查找问题

使用像Problem_Trek这样的数据库的真正好处在于可以回顾这些记录。回顾过去的问题以及解决办法允许你更快地解决重复出现的问题并发现趋势。

1. 需要的功能

要在回顾的表中查找记录，你可以再次使用SELECT命令。在本章前面我们用它查看过problem_logger表中新近增加的记录：

```
SELECT * FROM problem_logger WHERE id_number=$ID_NUMBER\G
```

对于Find_Problem脚本，需要对我们前面用过的SELECT命令做一点轻微的改动。这样过去的问题才能更容易找到，脚本会要求用户输入一个关键字。因此，如果你记得上个月你遇到过一个yum的问题，你可能会输入关键字yum。要在数据库中查找一条含有这个关键字的记录，可以在脚本的SELECT语句中加个LIKE命令。代码看起来如下：

```
mysql> SELECT * FROM problem_logger
-> WHERE prob_symptoms LIKE 'yum'\G
Empty set (0.00 sec)
```

等等，不行啊！这是因为这里需要通配符。要查找埋在prob_symptoms字段中的单词yum，需要在关键字的两边加上通配符。事实上，你是在让mysql查找一条记录，它的prob_symptoms字段含有单词yum。要用的通配符是百分号（%）。因此，代码看起来如下：

```
mysql> SELECT * FROM problem_logger
-> WHERE prob_symptoms LIKE '%yum%\G
***** 1. row *****
id_number: 1012111624
report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms: Running yum to install software and getting
'not found' message.
prob_solutions: Network service was down.
Issued the command: service network restart
1 row in set (0.00 sec)
```

你会想让SELECT命令来查找prob_solutions字段以及prob_symptoms字段，它允许对要找的记录进行更全面的查找：

```
mysql> SELECT * FROM problem_logger WHERE
-> prob_symptoms LIKE '%yum%'
-> OR
-> prob_solutions LIKE '%yum%\G
***** 1. row *****
id_number: 1012111624
...
```

说明 如果你担心大小写问题，那完全没必要。默认情况下，mysql会忽略大小写，所以你可以随便输入关键字yum或关键字YUM。

要对Problem_Trek数据库进行一次有效的查找，应该允许每次查找不止一个关键字。mysql命令LIKE不支持多个关键字。但你可以使用正则表达式命令REGEXP。REGEXP的一个好处是它会在整个指定字段中查找你指定的关键字，而不需要通配符。要查找多个关键字，你可以在它们中间加一个逻辑或，也就是一个管道符。因此，你要的代码看起来如下：

```
mysql> SELECT * FROM problem_logger WHERE
-> prob_symptoms REGEXP 'yum|dash'
-> OR
-> prob_solutions REGEXP 'yum|dash'\G
***** 1. row *****
id_number: 1012111624
report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms: Running yum to install software and getting
'not found' message.
prob_solutions: Network service was down.
Issued the command: service network restart
***** 2. row *****
id_number: 1012111631
report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms:
Moved script from Fedora to Ubuntu and was not working properly.
prob_solutions: Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.
2 rows in set (0.00 sec)
```

现在你已经有了Find_Problem脚本需要的关键功能了。让我们将脚本中的其余部分放到一起。

2. 创建脚本

在调用脚本时，最好允许关键字作为参数传入。但你并不知道有多少关键字会传进来。可能是5个，也可能是1个，甚至可能没有。事实上这是个很好解决的问题。使用if测试条件来看看是否有关键字传入。如果有，你可以用\$#参数（参见第13章）将它们一次提取；如果没有，脚本需要向用户要它们。完成这个任务的代码看起来如下：

```

# Obtain Keyword(s)
#
if [ -n "$1" ]      #Check if a keyword was passed
then                 #Grab all the passed keywords
#
KEYWORDS=$@          #Grab all the params as separate words, same string
#
else                #Keyword(s) not passed. Ask for them.
echo
echo "What keywords would you like to search for?"
echo -e "Please separate words by a space: \c"
read ANSWER
KEYWORDS=$ANSWER
fi
#

```

有了这些关键字后，下一步就是构建查找语句了。记住，需要在每对关键字之间放一个逻辑或(|)来让REGEXP正确查找它们。这可以用sed简单地实现。KEYWORDS数组变量中的每个空格都可以用|来替代：

```
KEYWORDS='echo $KEYWORDS | sed 's/ /|/g''
```

现在脚本看起来如下：

```

#!/bin/bash
#
# Find_Problem - finds problem records using keywords
#####
# Determine sql location & set variable
#
MYSQL=`which mysql`` Problem_Trek -u cbres`
#
#####
# Obtain Keyword(s)
#
if [ -n "$1" ]      #Check if a keyword was passed
then                 #Grab all the passed keywords
#
KEYWORDS=$@          #Grab all the params as separate words, same string
#
else                #Keyword(s) not passed. Ask for them.
echo
echo "What keywords would you like to search for?"
echo -e "Please separate words by a space: \c"
read ANSWER
KEYWORDS=$ANSWER
fi
#
#####
# Find problem record
#
echo
echo "The following was found using keywords: $KEYWORDS"
echo

```

```

#
KEYWORDS=`echo $KEYWORDS | sed 's/ /|/g'`
#
$MYSQL <<EOF
SELECT * FROM problem_logger WHERE
    prob_symptoms REGEXP '($KEYWORDS)'
    OR
    prob_solutions REGEXP '($KEYWORDS)'\G
EOF
#

```

当然，下一步是充分测试新脚本。由于多关键字问题是主要关心的，让我们先在命令行上尝试给脚本传多个关键字：

```
$ ./Find_Problem yum dash

The following was found using keywords: yum dash
```

```
***** 1. row *****
id_number: 1012111624
...
prob_solutions: Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.
```

现在，让我们不发送任何作为参数的关键字来测试一下脚本。相反，我们会让脚本来向我们要关键字：

```
$ ./Find_Problem

What keywords would you like to search for?
Please separate words by a space: Ubuntu

The following was found using keywords: Ubuntu
```

```
***** 1. row *****
id_number: 1012111631
...
prob_solutions: Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.
```

两个测试都可以运行，没有任何问题。现在在Problem_Trek数据库中查找问题记录会很容易。

有了这3个脚本，Record_Problem、Update_Problem和Find_Problem，你可以开始创建自己的问题数据库了。我们希望你花在创建这些脚本上的时间抵得上你在解决和避免问题上节省的时间。

27.3 小结

在本章中，我们回顾了一些你可以放到shell脚本中的高级功能。用户经常忽视了shell脚本在特定系统管理任务上的作用，因为这些脚本需要一些高级功能来正确运行。

本章首先演示了如何用shell性能工具（如uptime、df和free）来收集系统性能和资源使用情况的快照。需要的信息会用sed和gawk从命令的输出中收集，包括它们的一些高级功能。在从信

息生成报告后，它会通过E-mail客户端被发送到指定的用户账户。

接下来的一节介绍了用uptime和vmstat来创建更高级的性能监测脚本。数据是从这些工具中收集的，然后被存储在一个CSV文件中供其他脚本和工具使用。我们还创建了一个脚本来读取CSV文件中的性能数据并创建HTML格式的报告。由于是HTML格式的，报告可以在E-mail工具或喜欢的浏览器中阅读。

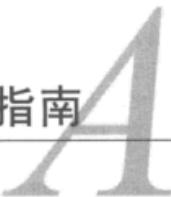
在本章的最后，我们逐步创建了一个问题跟踪数据库和用来在数据库上保存、更新以及报告的高级shell脚本。我们使用了mysql数据库软件，在脚本中包含了一些命令，例如INSERT、UPDATE和SELECT。我们还涉及了一些复杂的问题，比如查找记录，用LIKE命令和正则表达式命令REGEXP作比较。

本章中编写的所有这些脚本都可以方便地修改来在你自己的特定环境中使用。通过指出一些独特的高级方法来将shell脚本用在工作中，我们希望让你想到一些使用shell脚本的新方法以及你可以创建的新脚本。

感谢加入这段Linux命令行和shell脚本编程之旅。我们希望你能够享受这个过程并学会如何在命令行上自由游走，以及如何创建shell脚本来节省时间。但请不要就此停下命令行学习的脚步。在开源世界里总有一些新开发的东西，不管它是一个新的命令行工具还是一个全功能的shell。多关注Linux社区，紧紧跟随新的进步和功能。

附录 A

bash命令快速指南



本附录内容

- bash内建命令
- GNU的其他shell命令
- bash环境变量

如 你在全书中看到的，bash shell含有许多功能，因此有许多可用的命令。本附录提供了一个简明指南，你可以从中快速查找可以在bash命令行或bash shell脚本中使用的功能或命令。

A.1 内建命令

bash shell含有许多流行的内建在shell中的命令。这样在使用这些命令时，就能有更快的处理速度。表A-1列出了bash shell中直接可用的内建命令。

相比外部命令，内建命令提供了更高的性能，但越多的内建命令加到shell中，shell就会消耗越多的内存，而有些命令你几乎永远也不会用到。bash shell还含有为shell提供扩展功能的外部命令。这些都会在A.2节中讨论。

表A-1 bash内建命令

命 令	描 述
alias	为指定命令定义一个别名
bg	后台模式恢复作业的运行
bind	将键盘序列绑定到一个readline函数或宏
break	退出for、while、select或until循环
builtin	执行指定的shell内建命令
cd	将当前目录切换为指定的目录
caller	返回所有活动子函数调用的上下文
command	执行指定的命令，而不用通常的shell查找
compgen	为指定单词生成可能的补全匹配

(续)

命 令	描 述
complete	显示指定的单词是如何补全的
continue	继续执行for、while、select或until循环的下一次迭代
declare	声明一个变量或变量类型。
dirs	显示当前存储目录的列表
disown	为该进程将指定的作业从作业表中删除
echo	将指定字符串输出到STDOUT
enable	启用或禁用指定的内建shell命令
eval	将指定的参数拼接成一个命令，然后执行该命令
exec	用指定命令替换shell进程
exit	强制shell以指定的退出状态码退出
export	设置指定变量使其对子shell进程可用
fc	从历史记录中选择一列命令
fg	以前端模式恢复作业的运行
getopts	分析指定的位置参数
hash	查找并记住指定命令的全路径名
help	显示帮助文件
history	显示命令历史记录
jobs	列出活动的作业
kill	向指定的进程ID (PID) 发送一个系统信号
let	计算一个数学表达式中的每个参数
local	在函数中创建一个作用域受限的变量
logout	退出shell登录
popd	从目录栈中删除记录
printf	使用格式化字符串显示文本
pushd	向目录栈添加一个目录
pwd	显示当前工作目录的路径名
read	从STDIN读取一行数据并将其赋给一个变量
readonly	从STDIN读取一行数据并将其赋给一个不可修改的变量
return	强制函数以某个值退出，这个值可能会被调用的脚本提取
set	设置并显示环境变量的值和shell特性
shift	将位置参数依次向下降一个位置
shopt	打开/关闭控制shell可选行为的变量值
suspend	暂停shell的执行，直到收到一个SIGCONT信号
test	基于指定条件返回退出状态码0或1
times	显示累计的用户和系统时间
trap	如果收到了指定的系统信号，执行指定的命令
type	显示指定的单词如果作为命令将会如何被解释
ulimit	替系统用户给指定的资源设置一个上限

(续)

命 令	描 述
umask	为新建的文件和目录设置默认权限
unalias	删除指定的别名
unset	删除指定的环境变量或shell特性
wait	等待指定的进程完成，并返回退出状态码

A.2 bash命令

除了内建命令外，bash shell还使用外部命令来允许你操控文件系统以及操作文件和目录。表A-2列出了在使用bash shell时会用到的常见外部命令。

表A-2 bash shell外部命令

命 令	描 述
bzip2	采用Burrows-Wheeler块排序文本压缩算法和霍夫曼编码的压缩方法
cat	列出指定文件的内容
chage	修改指定系统用户账户的密码过期日期
chfn	修改指定用户账户的备注信息
chgrp	修改指定文件或目录的默认属组
chmod	为指定文件或目录修改系统安全权限
chown	修改指定文件或目录的默认属主
chpasswd	读取一个登录名/密码对文件并更新密码
chsh	修改指定用户账户的默认shell
compress	原始的Unix文件压缩工具
cp	将指定文件复制到另一个位置
date	以各种格式显示日期
df	为所有挂载的设备显示当前磁盘空间的统计数据
du	为指定文件路径显示磁盘使用情况的统计数据
file	查看指定文件的文件类型
find	对文件进行递归查找
finger	在Linux或远程系统上显示有关用户账户的信息
free	查看系统上可用的和已用的内存
grep	在文件中查找指定的文本字符串
groupadd	创建新的系统组
groupmod	修改已有的系统组
gzip	采用Lempel-Ziv编码的GNU项目压缩工具
head	显示指定文件内容的开头部分
killall	基于进程名向运行中的进程发送一个系统信号

(续)

命 令	描 述
less	查看文件内容的高级方法
link	用别名创建一个指向文件的链接
ls	列出目录内容
mkdir	在当前目录下创建指定目录
more	列出指定文件的内容，在每屏数据后暂停下来
mount	显示虚拟文件系统上挂载的磁盘设备或将磁盘设备挂载到虚拟文件系统上
mv	重命名文件
nice	在系统上使用不同优先级来运行命令
passwd	修改某个系统用户账户的密码
ps	显示系统上运行中进程的信息
pwd	显示当前目录
renice	修改系统上运行中应用的优先级
rm	删除指定文件
rmdir	删除指定目录
sort	基于指定的顺序组织数据文件中的数据
stat	显示指定文件的文件统计数据
sudo	作为root用户账户运行应用
tail	显示指定文件内容的末尾部分
tar	将数据和目录归档到单个文件中
touch	新建一个空文件，或更新一个已有文件的时间戳
top	显示活动进程，并说明重要的系统统计数据
umount	从虚拟文件系统上删除一个已挂载的磁盘设备
uptime	显示关于这个系统已经运行了多久的信息
useradd	新建一个系统用户账户
userdel	删除已有系统用户账户
usermod	修改已有系统用户账户
vmstat	生成一个详尽的系统上内存和CPU使用情况报告
which	查找可执行文件的位置
zip	Windows PKZIP程序的Unix版本

你能用这些命令在命令行上完成几乎所有你要做的任务。

A.3 环境变量

bash shell还使用了许多环境变量。虽然环境变量不是命令，但它们通常会影响shell命令的执行，所以知道这些shell环境变量很重要。表A-3列出了bash shell中可用的默认环境变量。

表A-3 bash shell环境变量

变 量	描 述
\$*	将所有命令行参数当做单个文本值包含
\$@	将所有命令行参数当做独立的文本值包含
\$#	命令行参数数目
\$?	最近使用的前端进程的退出状态码
\$-	当前命令行选项标记
\$\$	当前shell的进程ID (PID)
\$!	最近执行的后台进程的PID
\$0	来自命令行的命令名称
\$_	shell的绝对路径名
BASH	用来调用shell的全文件名
BASH_ARGC	当前子函数中的参数数目
BASH_ARGV	含有所有指定命令行参数的数组
BASH_COMMAND	当前正在被执行的命令的名称
BASH_ENV	如果设置了的话，每个bash脚本都会尝试在运行前执行由这个变量定义的起始文件
BASH_EXECUTION_STRING	在-c命令行选项中用到的命令
BASH_LINENO	含有脚本中每个命令的行号的数组
BASH_REMATCH	含有与指定的正则表达式匹配的文本元素的数组
BASH_SOURCE	含有shell中已声明函数所在源文件的名称的数组
BASH_SUBSHELL	当前shell产生的子shell数目
BASH_VERSION	当前bash shell实例的版本号
BASH_VERSINFO	含有当前bash shell实例的主版本号和次版本号的可变数组
COLUMNS	含有当前bash shell实例使用的终端的终端宽度
COMP_CWORD	含有当前光标位置的变量COMP_WORDS的索引值
COMP_LINE	当前命令行
COMP_POINT	当前光标位置相对于当前命令起始位置的索引
COMP_WORDBREAKS	在进行单词补全时用作单词分隔符的一组字符
COMP_WORDS	含有当前命令行上所有单词的可变数组
COMPREPLY	含有由shell函数生成的可能填充字的可变数组
DIRSTACK	含有目录栈当前内容的可变数组
EUID	当前用户的数字有效用户ID
FCEDIT	fc命令使用的默认编辑器
IGNORE	冒号分隔的文件名补全时要忽略的后缀名列表
FUNCNAME	当前执行的shell函数的名称
GLOBIGNORE	以冒号分隔的模式列表，定义了文件名展开时要忽略的文件名集合
GROUPS	含有当前用户属组列表的可变数组
histchars	控制历史记录展开的字符，最多可有3个字符
HISTCMD	当前命令在历史记录中的位置
HISTCONTROL	控制哪些命令留在历史记录列表中

(续)

变 量	描 述
HISTFILE	保存shell历史记录列表的文件名 (默认是.bash_history)
HISTFILESIZE	在历史文件中保存行数的上限
HISTIGNORE	冒号分隔的用来决定哪些命令不存进历史文件的模式列表
HISTSIZE	最多在历史文件中存多少条命令
HOSTFILE	含有shell在补全主机名时读取的文件的名称
HOSTNAME	当前主机的名称
HOSTTYPE	当前运行bash shell的机器
IGNOREEOF	shell在退出前必须收到连续的EOF字符的数量。如果这个值不存在, 默认是1
INPUTRC	readline初始化文件名 (默认是.inputrc)
LANG	Shell的语言环境分类
LC_ALL	定义一个语言环境分类, 覆盖LANG变量
LC_COLLATE	设置对字符串值排序时用的对照表顺序
LC_CTYPE	决定在文件名展开和模式匹配时用字符如何解释
LC_MESSAGES	决定解释前置美元符 (\$) 的双引号字符串的语言环境设置
LC_NUMERIC	决定格式化数字时的语言环境设置
LINENO	脚本中当前执行的行号
LINES	定义了终端上可见的行数
MACHTYPE	用“cpu-公司-系统”格式定义的系统类型
MAILCHECK	Shell多久查看一次新邮件 (以秒为单位, 默认值是60)
OLDPWD	shell之前的工作目录
OPTERR	设置为1时, bash shell会显示getopts命令产生的错误
OSTYPE	定义了shell运行的操作系统
PIPESTATUS	含有前端进程的退出状态码列表的可变数组
POSIXLY_CORRECT	设置了的话, bash会以POSIX模式启动
PPID	bash shell父进程的PID
PROMPT_COMMAND	设置了的话, 在命令行主提示符显示之前会执行这条命令
PS1	主命令行提示符字符串
PS2	次命令行提示符字符串
PS3	select命令的提示符
PS4	如果使用了bash的-x参数, 在命令行显示之前显示的提示符
PWD	当前工作目录
RANDOM	返回一个0~32 767的随机数; 对其赋值可作为随机数生成器的种子
REPLY	read命令的默认变量
SECONDS	自从shell启动到现在的秒数; 对其赋值将会重置计数器

(续)

变 量	描 述
SHELLOPTS	冒号分割的打开的bash shell选项列表
SHLVL	表明shell级别，每次有新bash shell（启动自增）
TIMEFORMAT	指定了shell显示时间值的格式
TMOUT	select和read命令在没输入的情况下等待多久（以秒为单位）。默认值为零，表示无限长
UID	当前用户的真实用户ID

你可以用set内建命令来显示这些环境变量。在不同的Linux发行版之间，开机时设置的默认shell环境变量可能而且经常会不一样。

附录 B

sed和gawk快速指南

B

本附录内容

- sed编辑器
- gawk程序

如果你要在shell脚本中做任何类型的数据处理，很可能你需要使用sed或gawk程序（有时二者都要用到）。本附录提供了一份sed和gawk的快速参考。当你在shell脚本中处理数据时，它们应该能派上用场。

B.1 sed编辑器

sed编辑器可以基于输入到命令行上的命令或存储在命令文本文件中的命令来操作数据流中的数据。它每次从输入中读取一行数据，并用提供的编辑器命令匹配该数据、按命令中指定的操作修改数据，然后将生成的新数据输出到STDOUT。

B.1.1 启动sed编辑器

使用sed命令的格式如下：

```
sed options script file
```

options参数允许你定制sed命令的行为，可包含表B-1中所列的选项。

script参数指定了作用在数据流上的单条命令。如果需要不止一条命令，你必须要用-e选项在命令行上指定它们，要么用-f选项在一个单独文件中指定它们。

表B-1 sed命令选项

选 项	描 述
-e script	将script中指定的命令添加到处理输入时运行的命令中
-f file	将file文件中指定的命令添加到处理输入时运行的命令中
-n	不要为每条命令产生输出，但会等待打印命令

B.1.2 sed命令

sed编辑器脚本含有sed针对输入流中的每行数据执行的命令。本节将会介绍一些较常见的sed命令。

1. 替换

s命令会替换输入流中的文本。s命令的格式如下：

s/pattern/replacement/flags

其中*pattern*是要被替换的文本，*replacement*是sed要插到数据流中的新文本。

*flags*参数控制着替换如何进行。有4种类型的替换标记可用。

- 一个数字，表明该模式出现的第几处应该被替换。
- g：表明所有该文本出现的地方都应该被替换。
- p：表明原来行中的内容应该被打印出来。
- w *file*：表明替换的结果应该写入到文件*file*中。

在第一类替换中，你可以指定sed编辑器应该替换第几个匹配模式的地方。举个例子，你用2来只替换该模式第二次出现的地方。

2. 寻址

默认情况下，你在sed编辑器中使用的命令会作用在文本数据的所有行上。如果你想让命令只作用在指定行上，或一组行上，你必须使用行寻址 (line addressing)。

在sed编辑器中，有两种形式的行寻址：

- 行的数字范围；
- 可以过滤出一行的文本模式。

两种形式都使用相同的格式来指定地址：

[address]command

当使用数字行寻址时，你通过行在文本流中的位置来引用行。sed编辑器会给数据流中的第一行分配行号1，然后对每个新行依次顺序增加。

\$ sed '2.3s/dog/cat/' data1

另一个限制命令作用在哪些行上的方法有点复杂。sed编辑器用允许你指定一个文本模式，它会用这个文本模式来为命令过滤出行。格式如下：

/pattern/command

你必须用斜线来将你指定的*pattern*包围起来。sed编辑器会将*command*只作用在包含你指定的文本模式的行上：

\$ sed '/rich/s/bash/csh/' /etc/passwd

这个过滤器能够找到含有文本rich的行，然后用csh来替换文本bash。

你也可以为某个特定地址将多条命令放在一起：

```
address {
    command1
    command2
    command3 }
```

sed编辑器会将你指定的每条命令只作用在匹配指定地址的行上。sed编辑器会执行出现在地址行上的每条命令：

```
$ sed '2{
> s/fox/elephant/
> s/dog/cat/
> }' data1
```

sed编辑器将每一条替换都作用在数据文件的第二行上。

3. 删除行

删除（delete）命令d跟它的名字十分相配。它会删除所有与提供的地址模式匹配的文本行。使用删除命令时要小心，因为如果你忘记加地址模式，所有的行都会被从数据流中删掉：

```
$ sed 'd' data1
```

显然删除命令跟指定的地址一起使用时才最有用。它允许你从数据流中删除特定的文本行，通过行数指定：

```
$ sed '3d' data6
```

或通过特定的行区间指定：

```
$ sed '2,3d' data6
```

sed编辑器的模式匹配功能也适用于删除命令：

```
$ sed '/number 1/d' data6
```

只有匹配指定文本的行才会被从流中删掉。

4. 插入和附加文本

如你所预期的，跟任何其他编辑器一样，sed编辑器允许你将文本行插入和附加到数据流。这两个命令的区别有点不那么明显：

- 插入命令（insert, i）在指定行前面添加一个新行；
- 附加命令（append, a）在指定行后面添加一个新行。

这两条命令的格式叫人很困惑：你不能在单个命令行上同时使用这两条命令。你必须在单独的一行中指定要插入或要附加的行。格式如下：

```
sed '[address]command\nnew_line'
```

new_line中的文本按你指定的位置出现在sed编辑器的输出中。记住，当你使用插入命令时，文本会出现在该行数据流文本的前面：

```
$ echo "testing" | sed 'i\
> This is a test'
This is a test
testing
$
```

当你使用追加命令时，文本会出现在该行数据流文本的后面：

```
$ echo "testing" | sed 'a\
> This is a test'
testing
This is a test
$
```

这允许你在普通文本的末尾插入文本。

5. 修改行

修改 (change) 命令允许你修改数据流中的整行文本。其格式跟插入和附加命令一样，你必须将新行跟sed命令的其余部分分开：

```
$ sed '3c\
> This is a changed line of text.' data6
```

反斜线字符用来表明脚本中的新数据行。

6. 转换命令

转换命令 (transform, y) 是唯一一个作用在单个字符上的sed编辑器命令。转换命令使用如下格式：

```
[address]y/inchars/outchars/
```

转换命令执行inchars值和outchars值之间的一一映射。inchars中的第一个字符会转换为outchars中的第一个字符，inchars中的第二个字符会转换为outchars中的第二个字符，依此类推，直到过了指定字符的长度。如果inchars和outchars长度不同，sed编辑器会报错。

7. 打印行

类似于替换命令中的p标记，p命令会在sed编辑器的输出中打印一行。打印 (print) 命令最常见的用法是打印与指定文本模式匹配的文本所在的行：

```
$ sed -n '/number 3/p' data6
This is line number 3.
$
```

打印命令允许你从输入流中只过滤出特定的数据行。

8. 写入到文件

w命令用来将行写入到文件中。w命令的格式为：

```
[address]w filename
```

filename可以用相对路径名或绝对路径名指定，但在任何一种情况下，运行sed编辑器的人都必须对这个文件有写权限。address可以是任意类型的寻址方法，比如单个行号、文本模式、行号区间或多个文本模式。

这里有个例子，它只打印数据流的前两行到文本文件：

```
$ sed '1,2w test' data6
```

输出文件test只含有输入流的前两行。

9. 从文件中读取

你已经了解了如何从命令行向数据流插入和附加文本。读取命令 (r) 允许你将单独文件中的数据插入。读取命令的格式为：

[address]r filename

其中filename参数用相对路径名或绝对路径名来指定含有数据的文件。不能给读取命令用地址区间，而只能指定单个行号或文本模式地址。sed编辑器会将文件中的文本插入到地址后面：

\$ sed '3r data' data2

sed编辑器将data文件中的全部文本都插入到了data2文件中，从data2文件的第3行开始。

B.2 gawk程序

gawk程序是原来Unix上的awk程序的GNU版本。awk程序通过提供了一门编程语言而不仅仅是编辑器命令，在sed编辑器的基础上将流编辑推进了一步。本节将会介绍gawk程序的基础知识，作为对它的功能的快速参考。

B.2.1 gawk命令格式

gawk程序的基本格式如下：

gawk options program file

表B-2列出了gawk程序支持的选项。

表B-2 gawk选项

选 项	描 述
-F fs	指定分隔行中数据字段的文件分隔符
-f file	指定读取程序的文件名
-v var=value	定义gawk程序中的一个变量及其默认值
-mf N	指定要处理的数据文件中的最大字段数
-mr N	指定数据文件中的最大数据行数
-W keyword	指定gawk的兼容模式或警告等级。用help选项来列出所有可用的关键字

命令行选项提供了一个简单办法来定制gawk程序的功能。

B.2.2 使用gawk

你可以从命令行上或shell脚本中直接使用gawk。本节将会介绍如何使用gawk程序以及如何输入脚本来让gawk处理。

1. 从命令行上读取程序脚本

gawk程序脚本是由一对花括号定义的。你必须将脚本命令放在两个括号之间。由于gawk命令

行假定脚本是单个文本字符串，你还必须用单引号来将脚本圈起来。这里有个简单的在命令行上指定gawk程序脚本的例子：

```
$ gawk '{print $1}'
```

这个脚本会显示输入流中每行的第一个数据字段。

2. 在程序脚本中使用多个命令

如果只能执行一条命令的话，那么这门编程语言不会太有用处。gawk编程语言允许你将多条命令组合成一个普通程序。要在命令行上指定的程序脚本中使用多条命令，只要在每个命令之间放一个分号就可以了：

```
$ echo "My name is Rich" | gawk '{$4="Dave"; print $0}'
My name is Dave
$
```

这个脚本执行了两条命令：首先用一个不同的值来替换第4个数据字段，然后显示流中的整个数据行。

3. 从文件中读取程序

跟sed编辑器一样，gawk编辑器允许你将程序存储在文件中，然后在命令行上引用它们：

```
$ cat script2
{ print $5 "'s userid is " $1 }
$ gawk -F: -f script2 /etc/passwd
```

gawk程序在输入数据流上执行了文件中指定的所有命令。

4. 在处理数据前运行脚本

gawk程序还允许你指定程序脚本何时运行。默认情况下，gawk从输入中读取一行文本，然后执行针对文本行中的数据执行程序脚本。有时，你可能需要在处理数据之前（比如创建报告的标题）运行脚本。为了做到这点，你可以使用BEGIN关键字。它会强制gawk先执行BEGIN关键字后面指定的程序脚本，然后再读取数据：

```
$ gawk 'BEGIN {print "This is a test report"}'
This is a test report
$
```

你可以在BEGIN块中放置任何类型的gawk命令，比如给变量赋默认值的命令。

5. 在处理数据后运行脚本

类似于BEGIN关键字，END关键字允许你指定一个程序脚本，在gawk读取数据后执行：

```
$ gawk 'BEGIN {print "Hello World!"} {print $0} END {print
"byebye"}'
Hello World!
This is a test
This is a test
This is another test.
This is another test.
byebye
$
```

gawk程序会先执行BEGIN块中的代码，然后处理输入流中的数据，最后执行END块中的代码。

B.2.3 gawk变量

gawk程序不只是一个编辑器，它是一个完整的编程环境。正因为如此，有许多命令和功能和gawk关联着。本节将会介绍你在用gawk编程时需要知道的主要功能。

1. 内建变量

gawk程序使用内建变量来在程序数据中引用特定功能。本节将会为你介绍gawk的内建变量来在gawk程序中使用，并演示如何使用它们。

gawk程序将数据定义成数据行和数据字段。数据行是一行数据（默认用换行符分隔），而数据字段是行中一个单独的数据元素（默认用空白字符比如空格或制表符分隔）。

gawk程序使用数据字段来引用每个数据行中的数据元素。表B-3介绍了这些变量。

表B-3 gawk数据字段和数据行变量

变 量	描 述
\$0	整个数据行
\$1	数据行中的第一个数据字段
\$2	数据行中的第二个数据字段
\$n	数据行中的第n个数据字段
FIELDWIDTHS	由空格分隔开的定义了每个字段确切宽度的一列数字
FS	输入字段分隔符
RS	输入数据行分隔符
OFS	输出字段分隔符
ORS	输出数据行分隔符

除了字段和数据行分隔符变量，gawk还提供了一些其他的内建变量来帮你了解数据中正在做什么以及从shell环境中提取信息。表B-4介绍了gawk中其他的内建变量。

表B-4 更多的gawk内建变量

变 量	描 述
ARGC	当前命令行参数个数
ARGVIND	当前文件在ARGV中的位置
ARGV	包含命令行参数的数组
CONVFMT	数字的转换格式（参见printf语句），默认值为%.6g
ENVIRON	当前shell环境变量及其值组成的关联数组
ERRNO	当读取或关闭输入文件发生错误时的系统错误号
FILENAME	用作gawk输入数据的数据文件的文件名
FNR	当前数据文件中的数据行数
IGNORECASE	设成非零时，忽略gawk命令中出现的字符串的字符大小写

(续)

变 量	描 述
NF	数据文件中的字段总数
NR	已处理的输入数据行数目
OFMT	数字的输出格式，默认值为%.6g
RLENGTH	由match函数所匹配的子字符串的长度
RSTART	由match函数所匹配的子字符串的起始位置

你可以在gawk程序脚本中的任何地方使用内建变量，包括BEGIN和END代码块中。

2. 在脚本中给变量赋值

在gawk程序中给变量赋值类似于在shell脚本中给变量赋值——使用赋值语句：

```
$ gawk '
> BEGIN{
> testing="This is a test"
> print testing
> }'
This is a test
$
```

给变量赋值后，你就可以在gawk脚本中任何地方使用该变量了。

3. 在命令行上给变量赋值

你也可以用gawk命令行来为gawk程序给变量赋值。这允许你在普通代码外设置值，即时修改值。这里有个使用命令行变量来显示文件中特定数据字段的例子：

```
$ cat script1
BEGIN{FS=". "}
{print $n}
$ gawk -f script1 n=2 data1
$ gawk -f script1 n=3 data1
```

这个功能是在gawk脚本中处理来自shell脚本的数据的好办法。

B.2.4 gawk程序的功能

gawk程序有一些功能使它非常便于操作数据，允许你创建gawk脚本来解析几乎任何类型的文本文件，包括日志文件。

1. 正则表达式

你可以使用基本正则表达式（BRE）或扩展正则表达式（ERE）来过滤程序脚本要作用在数据流中的哪些行上。

在使用正则表达式时，正则表达式必须出现在它控制的程序代码的左花括号之前：

```
$ gawk 'BEGIN{FS=". "}
/test/{print $1}' data1
This is a test
$
```

2. 匹配操作符

匹配操作符（matching operator）允许你将正则表达式限定在数据行中的特定数据字段上。

匹配操作符是波浪线（~）。你可以指定匹配操作符、数据字段变量以及要匹配的正则表达式：

```
$1 ~ /^data/
```

这个表达式会过滤出第一个数据字段以文本data开头的数据行。

3. 数学表达式

除了正则表达式外，你还可以在匹配模式中使用数学表达式。这个功能在匹配数据字段中的数字值时非常有用。举个例子，如果你要显示所有属于root用户组（组ID为0）的系统用户，你可以使用如下脚本：

```
$ gawk -F: '$4 == 0{print $1}' /etc/passwd
```

这个脚本显示了第4个数据字段含有值0的所有行的第一个数据字段。

4. 结构化命令

gawk程序支持如下结构化命令。

if-then-else语句：

```
if (condition) statement1; else statement2
```

while语句：

```
while (condition)
{
    statements
}
```

do-while语句：

```
do {
    statements
} while (condition)
```

for语句：

```
for(variable assignment; condition; iteration process)
```

这为gawk脚本程序员提供了大量的编程机会。你能写出几乎可以跟用任何高级语言编写的程序功能相媲美的gawk程序。

版 权 声 明

Original edition, entitled *Linux Command Line and Shell Scripting Bible, Second Edition*, by Richard Blum, Christine Bresnahan, ISBN 978-1-1180-0442-5, published by John Wiley & Sons, Inc.

Copyright © 2011 by John Wiley & Sons, Inc. All rights reserved. This translation published under License.

Simplified Chinese translation edition published by POSTS & TELECOM PRESS Copyright ©2012.

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书简体中文版由John Wiley & Sons, Inc.授权人民邮电出版社独家出版。

本书封底贴有John Wiley & Sons, Inc.激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

[General Information]

书名=LINUX命令行与SHELL脚本编程大全 第2版

作者=(美)布卢姆,(美)布雷斯纳汉著

页数=620

出版社=北京市：人民邮电出版社

出版日期=2012.08

SS号=13057851

DX号=000008334722

URL=<http://book2.duxiu.com/bookDetail.jsp?>

dxNumber=000008334722&d=48B6BBBBFBAADE2576BD60B

15E0815BF9

封面

书名

版权

前言

目录

第一部分Linux命令行

第1章 初识Linux shell

1.1什么是Linux

 1.1.1深入探究Linux内核

 1.1.2 GNU工具链

 1.1.3 Linux桌面环境

1.2 Linux发行版

 1.2.1核心Linux发行版

 1.2.2专业Linux发行版

 1.2.3 Linux LiveCD

1.3小结

第2章 走进shell

2.1终端模拟

 2.1.1图形功能

 2.1.2键盘

2.2 terminfo数据库

2.3 Linux控制台

2.4 xterm终端

 2.4.1命令行参数

- 2.4.2 xterm主菜单
- 2.4.3 VT选项菜单
- 2.4.4 VT字体菜单
- 2.5 Konsole终端
 - 2.5.1命令行参数
 - 2.5.2标签式窗口会话
 - 2.5.3配置文件
 - 2.5.4菜单栏
- 2.6 GNOME Terminal
 - 2.6.1命令行参数
 - 2.6.2标签
 - 2.6.3菜单栏
- 2.7小结

- 第3章 基本的bash shell命令
 - 3.1启动shell
 - 3.2 shell提示符
 - 3.3 bash手册
 - 3.4浏览文件系统
 - 3.4.1 Linux文件系统
 - 3.4.2遍历目录
 - 3.5文件和目录列表
 - 3.5.1基本列表功能
 - 3.5.2修改输出信息
 - 3.5.3完整的参数列表

3.5.4过滤输出列表

3.6处理文件

3.6.1创建文件

3.6.2复制文件

3.6.3链接文件

3.6.4重命名文件

3.6.5删除文件

3.7处理目录

3.7.1创建目录

3.7.2删除目录

3.8查看文件内容

3.8.1查看文件统计信息

3.8.2查看文件类型

3.8.3查看整个文件

3.8.4查看部分文件

3.9小结

第4章 更多的bash shell命令

4.1监测程序

4.1.1探查进程

4.1.2实时监测进程

4.1.3结束进程

4.2监测磁盘空间

4.2.1挂载存储媒体

4.2.2使用df命令

4.2.3使用du命令

4.3处理数据文件

4.3.1排序数据

4.3.2搜索数据

4.3.3压缩数据

4.3.4归档数据

4.4小结

第5章 使用Linux环境变量

5.1什么是环境变量

5.1.1全局环境变量

5.1.2局部环境变量

5.2设置环境变量

5.2.1设置局部环境变量

5.2.2设置全局环境变量

5.3删除环境变量

5.4默认shell环境变量

5.5设置PATH环境变量

5.6定位系统环境变量

5.6.1登录shell

5.6.2交互式shell

5.6.3非交互式shell

5.7可变数组

5.8使用命令别名

5.9小结

第6章理解Linux文件权限

6.1 Linux的安全性

- 6.1.1 /etc/passwd文件
- 6.1.2 /etc/shadow文件
- 6.1.3添加新用户
- 6.1.4删除用户
- 6.1.5修改用户

6.2使用Linux组

- 6.2.1 /etc/group文件
- 6.2.2创建新组
- 6.2.3修改组

6.3理解文件权限

- 6.3.1使用文件权限符
- 6.3.2默认文件权限

6.4改变安全性设置

- 6.4.1改变权限
- 6.4.2改变所属关系

6.5共享文件

6.6小结

第7章 管理文件系统

7.1探索Linux文件系统

- 7.1.1基本的Linux文件系统
- 7.1.2日志文件系统
- 7.1.3扩展的Linux日志文件系统

7.2操作文件系统

7.2.1创建分区

7.2.2创建文件系统

7.2.3如果出错了

7.3逻辑卷管理器

7.3.1逻辑卷管理布局

7.3.2 Linux中的LVM

7.3.3使用Linux LVM

7.4小结

第8章 安装软件程序

8.1包管理基础

8.2基于Debian的系统

8.2.1用aptitude管理软件包

8.2.2用aptitude安装软件包

8.2.3用aptitude更新软件

8.2.4用aptitude卸载软件

8.2.5 aptitude库

8.3基于Red Hat的系统

8.3.1列出已安装包

8.3.2用yum安装软件

8.3.3用yum更新软件

8.3.4用yum卸载软件

8.3.5处理损坏的包依赖关系

8.3.6 yum软件库

8.4从源码安装

8.5小结

第9章 使用编辑器

9.1 Vim编辑器

9.1.1 Vim基础

9.1.2编辑数据

9.1.3复制和粘贴

9.1.4查找和替换

9.2 Emacs编辑器

9.2.1在控制台上使用Emacs

9.2.2在X Window中使用Emacs

9.3 KDE系编辑器

9.3.1 KWrite编辑器

9.3.2 Kate编辑器

9.4 GNOME编辑器

9.4.1启动gedit

9.4.2基本的gedit功能

9.4.3设定偏好设置

9.5小结

第二部分shell脚本编程基础

第10章 构建基本脚本

10.1使用多个命令

10.2创建shell脚本文件

10.3显示消息

10.4 使用变量

10.4.1 环境变量

10.4.2 用户变量

10.4.3 反引号

10.5 重定向输入和输出

10.5.1 输出重定向

10.5.2 输入重定向

10.6 管道

10.7 执行数学运算

10.7.1 expr命令

10.7.2 使用方括号

10.7.3 浮点解决方案

10.8 退出脚本

10.8.1 查看退出状态码

10.8.2 exit命令

10.9 小结

第11章 使用结构化命令

11.1 使用if-then语句

11.2 if-then-else语句

11.3 嵌套if

11.4 test命令

11.4.1 数值比较

11.4.2 字符串比较

11.4.3 文件比较

11.5复合条件测试

11.6if-then的高级特性

 11.6.1使用双尖括号

 11.6.2使用双方括号

11.7 case命令

11.8小结

第12章 更多的结构化命令

12.1 for命令

 12.1.1读取列表中的值

 12.1.2读取列表中的复杂值

 12.1.3从变量读取列表

 12.1.4从命令读取值

 12.1.5更改字段分隔符

 12.1.6用通配符读取目录

12.2 C语言风格的for命令

 12.2.1C语言的for命令

 12.2.2使用多个变量

12.3 while命令

 12.3.1 while的基本格式

 12.3.2使用多个测试命令

12.4 until命令

12.5嵌套循环

12.6循环处理文件数据

12.7控制循环

12.7.1 break命令

12.7.2 continue命令

12.8处理循环的输出

12.9小结

第13章 处理用户输入

13.1命令行参数

13.1.1读取参数

13.1.2读取程序名

13.1.3测试参数

13.2特殊参数变量

13.2.1参数计数

13.2.2抓取所有的数据

13.3移动变量

13.4处理选项

13.4.1查找选项

13.4.2使用getopt命令

13.4.3使用更高级的getopts

13.5将选项标准化

13.6获得用户输入

13.6.1基本的读取

13.6.2超时

13.6.3隐藏方式读取

13.6.4从文件中读取

13.7小结

第14章 呈现数据

14.1理解输入和输出

 14.1.1标准文件描述符

 14.1.2重定向错误

14.2在脚本中重定向输出

 14.2.1临时重定向

 14.2.2永久重定向

14.3在脚本中重定向输入

14.4创建自己的重定向

 14.4.1创建输出文件描述符

 14.4.2重定向文件描述符

 14.4.3创建输入文件描述符

 14.4.4创建读写文件描述符

 14.4.5关闭文件描述符

14.5列出打开的文件描述符

14.6阻止命令输出

14.7创建临时文件

 14.7.1创建本地临时文件

 14.7.2在/tmp目录创建临时文件

 14.7.3创建临时目录

14.8记录消息

14.9小结

第15章 控制脚本

15.1处理信号

- 15.1.1重温 Linux信号
 - 15.1.2产生信号
 - 15.1.3捕捉信号
 - 15.1.4捕捉脚本的退出
 - 15.1.5移除捕捉
 - 15.2以后台模式运行脚本
 - 15.2.1后台运行脚本
 - 15.2.2运行多个后台作业
 - 15.2.3退出终端
 - 15.3在非控制台下运行脚本
 - 15.4作业控制
 - 15.4.1查看作业
 - 15.4.2重启停止的作业
 - 15.5调整谦让度
 - 15.5.1 nice命令
 - 15.5.2 renice命令
 - 15.6定时运行作业
 - 15.6.1用at命令来计划执行作业
 - 15.6.2计划定期执行脚本
 - 15.7启动时运行
 - 15.7.1开机时运行脚本
 - 15.7.2在新shell中启动
 - 15.8小结
- 第三部分 高级shell脚本编程

第16章 创建函数

16.1 基本的脚本函数

16.1.1 创建函数

16.1.2 使用函数

16.2 返回值

16.2.1 默认退出状态码

16.2.2 使用 return 命令

16.2.3 使用函数输出

16.3 在函数中使用变量

16.3.1 向函数传递参数

16.3.2 在函数中处理变量

16.4 数组变量和函数

16.4.1 向函数传数组参数

16.4.2 从函数返回数组

16.5 函数递归

16.6 创建库

16.7 在命令行上使用函数

16.7.1 在命令行上创建函数

16.7.2 在 .bashrc 文件中定义函数

16.8 小结

第17章 图形化桌面上的脚本编程

17.1 创建文本菜单

17.1.1 创建菜单布局

17.1.2 创建菜单函数

- 17.1.3添加菜单逻辑
 - 17.1.4整合shell脚本菜单
 - 17.1.5使用select命令
 - 17.2 使用窗口
 - 17.2.1 dialog包
 - 17.2.2 dialog选项
 - 17.2.3 在脚本中使用dialog命令
 - 17.3 使用图形
 - 17.3.1 KDE环境
 - 17.3.2 GNOME环境
 - 17.4 小结
- 第18章 初识sed和gawk
- 18.1 文本处理
 - 18.1.1 sed编辑器
 - 18.1.2 gawk程序
 - 18.2 sed编辑器基础
 - 18.2.1 更多的替换选项
 - 18.2.2 使用地址
 - 18.2.3 删除行
 - 18.2.4 插入和附加文本
 - 18.2.5 修改行
 - 18.2.6 转换命令
 - 18.2.7 回顾打印
 - 18.2.8 用sed和文件一起工作

18.3小结

第19章 正则表达式

19.1什么是正则表达式

19.1.1定义

19.1.2正则表达式的类型

19.2定义BRE模式

19.2.1纯文本

19.2.2特殊字符

19.2.3锚字符

19.2.4点字符

19.2.5字符组

19.2.6排除字符组

19.2.7使用区间

19.2.8特殊字符组

19.2.9星号

19.3扩展正则表达式

19.3.1问号

19.3.2加号

19.3.3使用花括号

19.3.4管道符号

19.3.5聚合表达式

19.4实用中的正则表达式

19.4.1目录文件计数

19.4.2验证电话号码

- 19.4.3解析邮件地址
- 19.5小结
- 第20章sed进阶
 - 20.1多行命令
 - 20.1.1 next命令
 - 20.1.2多行删除命令
 - 20.1.3多行打印命令
 - 20.2保持空间
 - 20.3排除命令
 - 20.4改变流
 - 20.4.1跳转
 - 20.4.2测试
 - 20.5模式替代
 - 20.5.1 and符号
 - 20.5.2替换单独的单词
 - 20.6在脚本中使用sed
 - 20.6.1使用包装脚本
 - 20.6.2重定向sed的输出
 - 20.7创建sed实用工具
 - 20.7.1加倍行间距
 - 20.7.2对可能含有空白行的文件加倍行间距
 - 20.7.3给文件中的行编号
 - 20.7.4打印末尾行

20.7.5删除行

20.7.6删除HTML标签

20.8小结

第21章gawk进阶

21.1使用变量

21.1.1内建变量

21.1.2自定义变量

21.2处理数组

21.2.1定义数组变量

21.2.2遍历数组变量

21.2.3删除数组变量

21.3使用模式

21.3.1正则表达式

21.3.2 匹配操作符

21.3.3数学表达式

21.4结构化命令

21.4.1 if语句

21.4.2 while语句

21.4.3 do-while语句

21.4.4 for语句

21.5格式化打印

21.6内建函数

21.6.1数学函数

21.6.2字符串函数

21.6.3时间函数

21.7自定义函数

21.7.1定义函数

21.7.2使用自定义函数

21.7.3创建函数库

21.8小结

第22章 使用其他shell

22.1什么是dash shell

22.2 dash shell的特性

22.2.1 dash命令行参数

22.2.2 dash环境变量

22.2.3 dash内建命令

22.3 dash脚本编程

22.3.1创建dash脚本

22.3.2不能使用的功能

22.4 zsh shell

22.5 zsh shell的组成

22.5.1 shell选项

22.5.2内建命令

22.6 zsh脚本编程

22.6.1数学运算

22.6.2结构化命令

22.6.3函数

22.7小结

第四部分 高级shell脚本编程主题

第23章 使用数据库

23.1 MySQL数据库

23.1.1 安装MySQL

23.1.2 MySQL客户端界面

23.1.3 创建MySQL数据库对象

23.2 PostgreSQL数据库

23.2.1 安装PostgreSQL

23.2.2 PostgreSQL命令行界面

23.2.3 创建PostgreSQL数据库 对象

23.3 使用数据表

23.3.1 创建数据表

23.3.2 插入和删除数据

23.3.3 查询数据

23.4 在脚本中使用数据库

23.4.1 连接到数据库

23.4.2 向服务器发送命令

23.4.3 格式化数据

23.5 小结

第24章 使用Web

24.1 Lynx程序

24.1.1 安装Lynx

24.1.2 lynx命令行

24.1.3 Lynx配置文件

- 24.1.4 Lynx环境变量
- 24.1.5从Lynx中抓取数据
- 24.2 cURL程序
 - 24.2.1安装cURL
 - 24.2.2探索cURL
- 24.3使用zsh处理网络
 - 24.3.1 TCP模块
 - 24.3.2客户端 / 服务器模式
 - 24.3.3使用zsh进行C/S编程

24.4小结

第25章 使用E-mail

- 25.1 Linux E-mail基础
 - 25.1.1Linux中的E-mail
 - 25.1.2邮件传送代理
 - 25.1.3邮件投递代理
 - 25.1.4邮件用户代理
- 25.2建立服务器
 - 25.2.1 sendmail
 - 25.2.2 Postfix
- 25.3使用Mailx发送消息
- 25.4 Mutt程序
 - 25.4.1安装Mutt
 - 25.4.2 Mutt命令行
 - 25.4.3使用Mutt

25.5小结

第26章 编写脚本实用工具

26.1监测磁盘空间

26.1.1需要的功能

26.1.2创建脚本

26.1.3运行脚本

26.2进行备份

26.3管理用户账户

26.3.1需要的功能

26.3.2创建脚本

26.4小结

第27章shell脚本编程进阶

27.1监测系统统计数据

27.1.1系统快照报告

27.1.2系统统计数据报告

27.2问题跟踪数据库

27.2.1创建数据库

27.2.2记录问题

27.2.3更新问题

27.2.4查找问题

27.3小结

附录A bash命令快速指南

附录B sed和gawk快速指南