

Linux之父 Linus Torvalds
O'Reilly创始人 Tim O'Reilly 倾力推荐



20周年纪念版

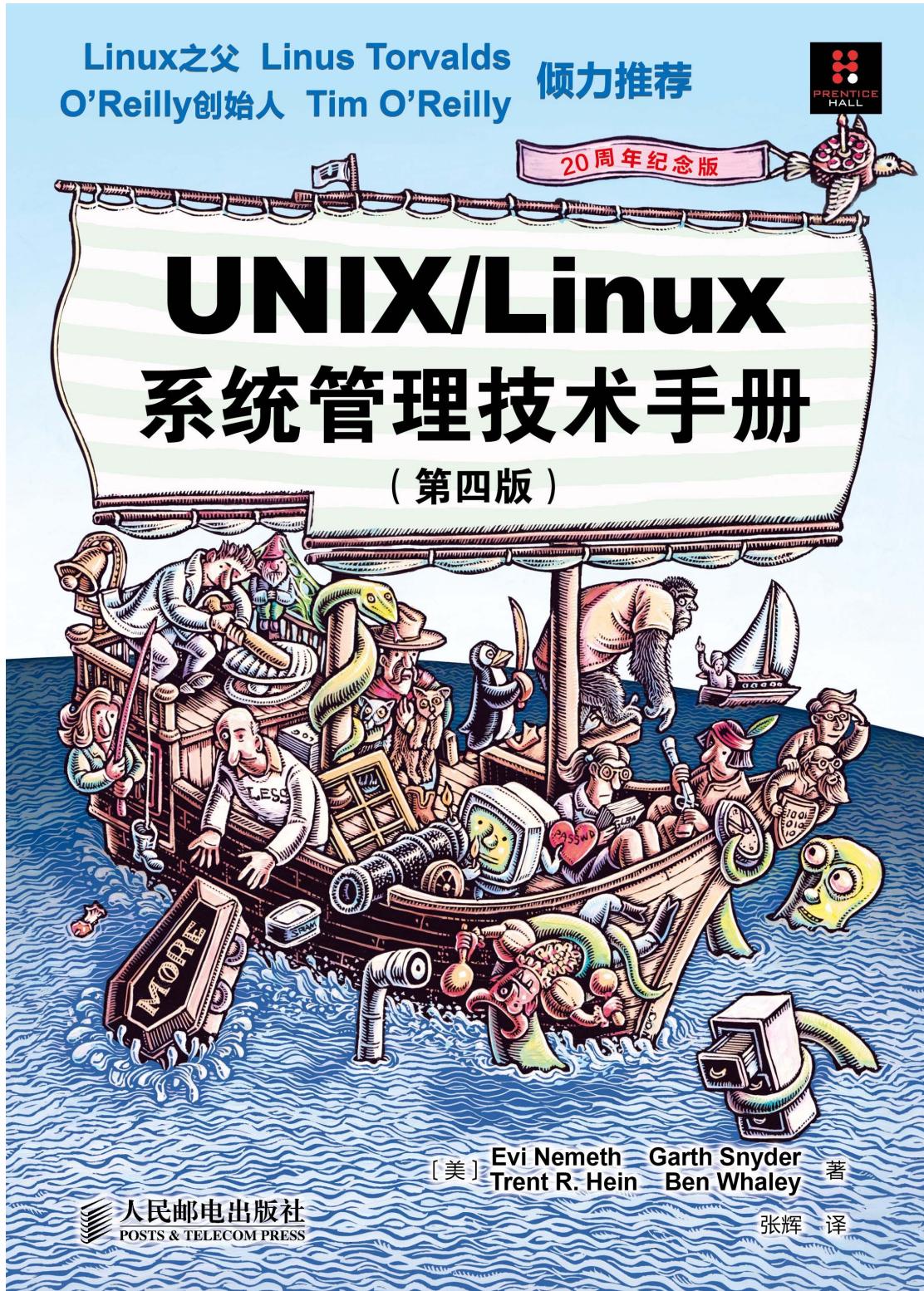
UNIX/Linux 系统管理技术手册

(第四版)

[美] Evi Nemeth Garth Snyder
Trent R. Hein Ben Whaley 著

张辉 译

人民邮电出版社
POSTS & TELECOM PRESS



UNIX/Linux 系统管理技术手册（第四版）

Evi Nemeth

[美] Garth Snyder 著

Trent R. Hein

Ben Whaley

张辉 译

人民邮电出版社

北京

版 权 声 明

Authorized translation from the English language edition, entitled UNIX and Linux System Administration Handbook, 4nd Edition, 9780131480056 by Evi Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley, published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2011 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and POSTS & TELECOMMUNICATIONS PRESS Copyright © 2011.

本书封面贴有 Pearson Education (培生教育出版集团) 激光防伪标签。无标签者不得销售。

UNIX/Linux 系统管理技术手册（第四版）

◆ 著 [美]Evi Nemeth, Garth Snyder, Trent R. Hein, Ben White

译 张 辉

责任编辑 俞 彬

◆ 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 印刷

◆ 开本：787×1092 1/16

印张：63

字数：1830 千字 2012 年 5 月第 4 版

印数：20001~30000 册 2012 年 5 月北京第 1 次印刷

著作权合同登记号 图字：01-2011-1387 号

ISBN 978-7-115-27900-2/TP

定价：.00 元

读者服务热线：(010)67132692 印装质量热线：(010)6712921

S 前 言

当我们在 20 世纪 80 年代撰写本书的第一版时，我们就急于把我们的手稿与其他讲述系统管理技术的书籍做个比较。令我们高兴的是，我们当时只找到了三本同类的书。现如今，读者可以有数以百计的不同选择。下面是我们这本书有别于其他书的特色。

□ 我们采取言传身教的讲授方式。读者已经有丰富的手册可供参考；我们的目标是总结我们在系统管理工作中积累起来的经验，并且推荐值得

读者花时间尝试的方法。这本书包含了许多在现实中向困难宣战的故事，也给出了大量注重实践的建议。

- 这本书不是讲如何在家里、在车库里或者在 PDA 上运行 UNIX 或者 Linux。我们介绍的是如何管理生产环境，比如商业公司、政府机关以及大学。
- 我们详细地讲解了连网技术。这是系统管理工作中最为困难的方面，也是我们认为自己可以向读者提供最多帮助的领域。
- 我们介绍了主流的 UNIX 和 Linux 版本。

本书的组织

本书分为 3 大部分：基本管理技术、网络管理技术和其他管理技术。

基本管理技术部分从系统管理员的角度全面介绍 UNIX 和 Linux。其中的章节涉及运行单机系统所需要的大部分知识和技术。

网络管理技术部分描述了 UNIX 系统使用的各种协议，介绍了用来配置、扩展和维护网络以及面向因特网的服务器所使用的各种技术。在这个部分里还介绍了上层的网络软件。各章的专题内容包括域名系统、网络文件系统、路由技术、电子邮件和网络管理。

其他管理技术部分包含各种各样的补充信息。其中有些章节讨论了一些可供选用的功能，例如支持服务器虚拟化的功能。其他一些章节就各种主题——从环保型计算技术到驱遣系统管理团队的控制手腕——提供了若干建议。

每一章的后面都有一组练习题。用星号标出了我们估计完成这道题的工作量，“工作量”是一项通过任务难度和耗时两方面来衡量的指标。题目分为 4 个等级：

没有星号 简单题目，应该很容易就能做出来；

★ 比较难或者要花比较长时间的题目，可能要求做实验；

★★ 最困难或者最花时间的题目，要求做实验，并进行深入分析；

★★★★ 作为整个学期的项目来做（只在个别章节里出现）。

有些习题需要有系统上的 root 或者 sudo 访问权限，有些题目要求得到本地系

统管理小组的许可。有这两种要求的时候，习题会予以说明。

我们的供稿人

我们很高兴 Ned McClain、David Schweikert 和 Tobi Oetiker 能以供稿人的身份再度参与本书的编写工作。我们还欢迎 Terry Morreale 和 Ron Jachim 在这一版成为我们新的供稿人。他们在各个领域的深厚知识业已极大地丰富了本书的内容。

联络信息

请将意见、建议和 bug 报告给 ulsah@book.admin.com。我们会回邮件，但请稍有耐心一些；有时候要等几天时间，我们中间的某一个人才能回复邮件。由于这个使用别名的邮件地址接收到的邮件数量巨大，所以我们很遗憾不能回答技术问题。

访问我们的 Web 网站 admin.com 可以查到一份当前的 bug 列表，还可以看到其他的最新信息。

我们希望您会喜欢本书，并祝您的系统管理工作好运连连！

Evi Nemeth
Garth Snyder
Trent R. Hein
Ben Whaley

目 录

第一部分 基本管理技术..... 1

第 1 章 从何处入手

1.1 系统管理员的基本任务

1.1.1 账号管理

1.1.2 增删硬件

1.1.3 执行备份

1.1.4 安装和更新软件

1.1.5 监视系统

1.1.6 故障诊断

1.1.7 维护本地文档

1.1.8 时刻警惕系统安全

1.1.9 救火

1.2 读者的知识背景

1.3 UNIX 和 Linux 之间的摩擦

1.4 Linux 的发行版本

1.5 本书使用的示例系统

1.5.1 用作示例的 Linux 发行版本

1.5.2 用作示例的 UNIX 发行版本

1.6 特定于系统的管理工具

1.7 表示法和印刷约定

1.8 单位

1.9 手册页和其他联机文档

1.9.1 手册页的组织

1.9.2 man : 读取手册页

1.9.3 手册页的保存

1.9.4 GNUTexinfo

1.10 其他的权威文档

1.10.1 针对系统的专门指南

1.10.2 针对软件包的专门文档

1.10.3 书籍

1.10.4 RFC 和其他 Internet 文档

1.10.5 LDP

1.11 其他的信息资源

1.12 查找和安装软件的途径

1.12.1 判断软件是否已经安装

1.12.2 增加新软件

1.12.3 从源代码编译软件

1.13 重压下的系统管理员

1.14 推荐读物

1.15 习题

第 2 章 脚本和 shell

2.1 shell 的基础知识

2.1.1 编辑命令

2.1.2 管道和重定向

2.1.3 变量和引用

2.1.4 常见的过滤命令

2.2 bash 脚本编程

2.2.1 从命令到脚本

2.2.2 输入和输出

2.2.3 命令行参数和函数

2.2.4 变量的作用域

2.2.5 控制流程

2.2.6 循环

2.2.7 数组和算术运算

2.3 正则表达式

2.3.1 匹配过程

2.3.2 普通字符

2.3.3 特殊字符

2.3.4 正则表达式的例子

2.3.5 捕获

2.3.6 贪心、懒惰和灾难性的回溯

2.4 Perl 编程

2.4.1 变量和数组

2.4.2 数组和字符串文字

2.4.3 函数调用

2.4.4 表达式里的类型转换

2.4.5 字符串表达式和变量

2.4.6 哈希

2.4.7 引用和自动生成

2.4.8 Perl 语言里的正则表达式

2.4.9 输入和输出

2.4.10 控制流程

2.4.11 接受和确认输入

2.4.12 Perl 用作过滤器

2.4.13 Perl 的附加模块

2.5 Python 脚本编程

2.5.1 Python 快速入门

2.5.2 对象、字符串、数、列表、字典、元组和文件

2.5.3 确认输入的例子

2.5.4 循环

2.6 脚本编程的最佳实践

2.7 推荐读物

Shell 基础知识和 bash 脚本编程

正则表达式

Perl 脚本编程

Python 脚本编程

2.8 习题

第3章 引导和关机

3.1 引导

3.1.1 恢复模式下引导进入 shell

3.1.2 引导过程的步骤

3.1.3 初始化内核

3.1.4 配置硬件

3.1.5 创建内核进程

3.1.6 操作员干预（仅限恢复模式）

3.1.7 执行启动脚本

3.1.8 引导进程完成

3.2 引导 PC

3.3 GRUB：全面统一的引导加载程序

3.3.1 内核选项

3.3.2 多重引导

3.4 引导进入单用户模式

3.4.1 用 GRUB 引导单用户模式

3.4.2 SPARC 上的单用户模式

3.4.3 HP-UX 的单用户模式

3.4.4 AIX 的单用户模式

3.5 启动脚本

3.5.1 init 及其运行级

3.5.2 启动脚本概述

3.5.3 Red Hat 启动脚本

3.5.4 SUSE 的启动脚本

3.5.5 Ubuntu 的启动脚本和 Upstart 守护进程

3.5.6 HP-UX 的启动脚本

3.5.7 AIX 的启动

3.6 引导 Solaris

3.6.1 Solaris 的 SMF

3.6.2 崭新的世界：用 SMF 引导系统

3.7 重新引导和关机

3.7.1 shutdown：停止系统的妥善方式

3.7.2 halt 和 reboot：关闭系统的更简单方式

3.8 习题

第 4 章 访问控制和超级权限

4.1 传统的 UNIX 访问控制

4.1.1 文件系统的访问控制

4.1.2 进程的所有权

4.1.3 root 账号

4.1.4 setuid 和 setgid 执行方式

4.2 现代的访问控制

4.2.1 基于角色的访问控制

4.2.2 SELinux：增强安全性的 Linux

4.2.3 POSIX 能力 (Linux)

4.2.4 PAM：可插入式身份验证模块

4.2.5 Kerberos：第三方的加密验证

4.2.6 访问控制列表

4.3 实际中的访问控制

4.3.1 选择 root 的口令

4.3.2 登录进入 root 账号

4.3.3 su : 替换用户身份

4.3.4 sudo : 受限的 su

4.3.5 口令保险柜和口令代管

4.4 root 之外的其他伪用户

4.5 习题

第 5 章 进程控制

5.1 进程的组成部分

5.1.1 PID : 进程的 ID 号

5.1.2 PPID : 父 PID

5.1.3 UID 和 EUID : 真实的和有效的用户 ID

5.1.4 GID 和 EGID : 真实的和有效的组 ID

5.1.5 谦让度

5.1.6 控制终端

5.2 进程的生命周期

5.3 信号

5.4 kill: 发送信号

5.5 进程的状态

5.6 nice 和 renice : 影响调度优先级

5.7 ps : 监视进程

5.8 用 top、prstat 和 topas 动态监视进程

5.9 /proc 文件系统

5.10 strace、truss 和 tusc：追踪信号和系统调用

5.11 失控进程

5.12 推荐读物

5.13 习题

第6章 文件系统

6.1 路径名称

6.1.1 绝对路径和相对路径

6.1.2 文件名中的空白

6.2 挂载和卸载文件系统

6.3 文件树的组织

6.4 文件类型

6.4.1 普通文件

6.4.2 目录

6.4.3 字符设备文件和块设备文件

6.4.4 本地域套接口

6.4.5 有名管道

6.4.6 符号链接

6.5 文件属性

6.5.1 权限位

6.5.2 setuid 和 setgid 位

6.5.3 粘附位

6.5.4 ls : 列出和查看文件

6.5.5 chmod : 改变权限

6.5.6 chown 和 chgrp : 改变归属关系和组

6.5.7 umask : 分配默认的权限

6.5.8 Linux 上的额外标志

6.6 访问控制列表

6.6.1 UNIX ACL 简史

6.6.2 ACL 的实现

6.6.3 系统支持的 ACL

6.6.4 POSIX 的 ACL

6.6.5 NFSv4 的 ACL

6.7 习题

第 7 章 添加新用户

7.1 /etc/passwd 文件

7.1.1 登录名

7.1.2 加密的口令

7.1.3 UID 号

7.1.4 默认的 GID 号

7.1.5 GECOS 字段

7.1.6 主目录

7.1.7 登录 shell

7.2 /etc/shadow 和 /etc/security/passwd 文件

7.3 /etc/group 文件

7.4 添加用户：基本步骤

7.4.1 编辑 passwd 和 group 文件

7.4.2 设置口令

7.4.3 创建主目录并安装启动文件

7.4.4 设置权限和所属关系

7.4.5 设置邮件主目录

7.4.6 配置角色和管理特权

7.4.7 收尾步骤

7.5 用 useradd 添加用户

7.5.1 Ubuntu 上的 useradd

7.5.2 SUSE 上的 useradd

7.5.3 Red Hat 上的 useradd

7.5.4 Solaris 上的 useradd

7.5.5 HP-UX 上的 useradd

7.5.6 AIX 的 useradd

7.5.7 useradd 举例

7.6 用 newusers 成批添加用户 (Linux)

7.7 删 除 用户

7.8 禁止登录

7.9 用系统的专门工具管理用户

7.10 用 PAM 降低风险

7.11 集中管理账号

7.11.1 LDAP 和 Active Directory

7.11.2 单一登录系统

7.11.3 身份管理系统

7.12 推荐读物

7.13 习题

第8章 存储

8.1 只想加一块硬盘！

8.1.1 Linux 的做法

8.1.2 Solaris 的做法

8.1.3 HP-UX 的做法

8.1.4 AIX 的做法

8.2 存储硬件

8.2.1 硬盘

8.2.2 固态盘

8.3 存储硬件接口

8.3.1 PATA 接口

8.3.2 SATA 接口

8.3.3 并行 SCSI

8.3.4 串行 SCSI

8.3.5 SCSI 和 SATA 的比较

8.4 层层剖析：存储上的软件

8.5 硬盘的安装和底层管理

8.5.1 在硬件层面上的安装核实

8.5.2 磁盘设备文件

8.5.3 格式化和坏块管理

8.5.4 ATA 安全擦除

8.5.5 hdparm : 设置磁盘和接口参数 (Linux)

8.5.6 使用 SMART 监视磁盘

8.6 磁盘分区

8.6.1 传统的分区方式

8.6.2 Windows 的分区

8.6.3 GPT : GUID 分区表

8.6.4 Linux 的分区

8.6.5 Solaris 的分区

8.6.6 HP-UX 的分区

8.7 RAID : 廉价磁盘冗余阵列

8.7.1 软硬 RAID 对比

8.7.2 RAID 的级别

8.7.3 硬盘故障恢复

8.7.4 RAID 5 的缺点

8.7.5 mdadm : Linux 上的软 RAID

8.8 逻辑卷管理

8.8.1 LVM 的实现

8.8.2 Linux 的逻辑卷管理

8.8.3 HP-UX 的逻辑卷管理

8.8.4 AIX 的逻辑卷管理

8.10 文件系统

8.9.1 Linux 文件系统 : ext 家族的文件系统

8.9.2 HP-UX 文件系统

8.9.3 AIX 的 JFS2 文件系统

8.9.4 文件系统的术语

8.9.5 文件系统的多态性

8.9.6 mkfs : 格式化文件系统

8.9.7 fsck : 检查和修复文件系统

8.9.8 挂载文件系统

8.9.9 设置自动挂载

8.9.10 挂载 USB 设备

8.9.11 启用交换分区

8.10 ZFS : 解决所有存储问题

8.10.1 ZFS 体系结构

8.10.2 举例 : Solaris 磁盘分区

8.10.3 文件系统和属性

8.10.4 属性继承

8.10.5 每个用户一个文件系统

8.10.6 快照和克隆

8.10.7 原始卷

8.10.8 通过 NFS、CIFS 和 iSCSI 共享文件系统

8.10.9 存储池管理

8.11 存储区域网络

8.11.1 SAN 网络

8.11.2 iSCSI : SCSI over IP

8.11.3 从 iSCSI 卷引导

8.11.4 iSCSI 的厂商特性

8.12 习题

第 9 章 周期性进程

9.1 cron:按时间表执行命令

9.2 crontab 文件的格式

9.3 crontab 管理

9.4 Linux 及其 Vixie-CRON 的扩展

9.5 cron 的常见用途

9.5.1 简单的提醒功能

9.5.2 清理文件系统

9.5.3 配置文件的网络分布

9.5.4 循环日志文件

9.6 习题

第 10 章 备份

10.1 备份基本原理

10.1.1 从中心位置执行所有的备份

10.1.2 给备份介质加卷标

10.1.3 选择合理的备份间隔

10.1.4 仔细选择文件系统

10.1.5 在单一介质上做日常转储

10.1.6 异地保存介质

10.1.7 保护备份

10.1.8 备份期间限制活动

10.1.9 查验介质

10.1.10 发掘介质的寿命

10.1.11 为备份而设计数据

10.1.12 做最坏的准备

10.2 备份设备和介质

10.2.1 光盘：CD-R/RW、DVD±R/RW、DVD-RAM 和蓝光

10.2.2 便携和移动硬盘

10.2.3 磁带概述

10.2.4 小型磁带机：8mm 磁带和 DDS/DAT

10.2.5 DLT 和 S-DLT

10.2.6 AIT 和 SAIT

10.2.7 VXA 和 VXA-X

10.2.8 LTO

10.2.9 自动选带机、自动换带机以及磁带库

10.2.10 硬盘

10.2.11 因特网和云备份服务

10.2.12 介质类型小结

10.2.13 设备选型

10.3 节省空间和时间与增量备份

10.3.1 简单的计划

10.3.2 适中的计划

10.4 用 dump 建立备份机制

10.4.1 转储文件系统

10.4.2 用 restore 从转储中恢复

10.4.3 恢复整个文件系统

10.4.4 恢复到新硬盘上

10.5 为系统升级而执行转储和恢复

10.6 使用其他存档程序

10.6.1 tar: 给文件打包

10.6.2 dd : 处理位流

10.6.3 ZFS 的备份

10.7 使用同一卷磁带上的多个文件

10.8 Bacula

10.8.1 Bacula 的模型

10.8.2 设置 Bacula

10.8.3 安装数据库和 Bacula 的守护进程

10.8.4 配置 Bacula 的守护进程

10.8.5 公共的配置段

10.8.6 bacula-dir.conf : 配置控制文件

10.8.7 bacula-sd.conf : 配置存储守护进程

10.8.8 bconsole.conf : 配置控制台

10.8.9 安装和配置客户端的文件守护进程

10.8.10 启动 Bacula 的守护进程

10.8.11 向存储池添加介质

10.8.12 执行一次手工备份

10.8.13 执行一次恢复工作

10.8.14 给 Windows 客户机做备份

10.8.15 监视和调试 Bacula 的配置

10.8.16 Bacula 的技巧和窍门

10.8.17 Bacula 的替代工具

10.9 商用备份产品

10.9.1 ADSM/TSM

10.9.2 VeritasNetBackup

10.9.3 EMCNetWorker

10.9.4 其他选择

10.10 推荐读物

10.11 习题

第 11 章 系统日志与日志文件

11.1 日志文件的位置

11.1.1 不用管理的文件

11.1.2 厂商特有的文件

11.2 syslog : 系统事件的日志程序

11.2.1 syslog 的体系结构

11.2.2 配置 syslogd

11.2.3 配置文件举例

11.2.4 调试 syslog

11.2.5 syslog 的其他替代方案

11.2.6 Linux 内核和引导时刻日志

11.3 AIX : 日志记录和出错处理

11.3.1 AIX 的 syslog 配置

11.4 logrotate: 管理日志文件

11.5 分析日志文件

11.6 日志记录的策略

11.7 习题

第 12 章 软件安装和管理

12.1 安装 Linux 和 OpenSolaris

12.1.1 从网络引导 PC

12.1.2 为 Linux 设置 PXE

12.1.3 非 PC 的网络引导

12.1.4 Kickstart : RHEL 的自动安装程序

12.1.5 AutoYaST : SUSE 的自动安装工具

12.1.6 用 Ubuntu 的安装程序自动安装

12.2 安装 Solaris

12.2.1 使用 JumpStart 网络安装

12.2.2 使用自动安装程序进行网络安装

12.3 安装 HP-UX

12.3.1 用 Ignite-UX 自动安装

12.4 使用 NIM 安装 AIX

12.5 软件包管理

12.6 Linux 的高级软件包管理系统

12.6.1 rpm : 管理 RPM 软件包

12.6.2 dpkg : 管理 Debian 的软件包

12.7 Linux 的高级软件包管理系统

12.7.1 软件包的库

12.7.2 RHN : Red Hat 网络

12.7.3 APT : 高级软件包工具

12.7.4 配置 apt-get

12.7.5 /etc/apt/sources.list 文件的例子

12.7.6 创建本地的库镜像

12.7.7 自动执行 apt-get

12.7.8 yum : 管理 RPM 的发布

12.7.9 Zypper : SUSE 的软件包管理

12.8 UNIX 的软件包管理

12.8.1 Solaris 软件包

12.8.2 HP-UX 软件包

12.8.3 AIX 的软件管理

12.9 版本控制

12.9.1 创建备份文件

12.9.2 正规的版本控制系统

12.9.3 Subversion

12.9.4 Git

12.10 软件的本地化和配置

12.10.1 本地化的组织

12.10.2 测试

12.10.3 本地编译软件

12.10.4 发布本地软件

12.11 配置管理工具

12.11.1 cfengine : 计算机免疫系统

12.11.2 LCFG : 大规模配置系统

12.11.3 Template Tree 2 : cfengine 的帮手

12.11.4 DMTF/CIM : 公共信息模型

12.12 通过 NFS 共享软件

12.12.1 软件包的名字空间

12.12.2 依赖关系的管理

12.12.3 封装脚本

12.13 推荐读物

12.14 习题

第 13 章 驱动程序和内核

13.1 内核的适应性

13.2 驱动程序和设备文件

13.2.1 设备文件和设备号

13.2.2 创建设备文件

13.2.3 设备的命名约定

13.2.4 自定义内核和可加载模块

13.3 配置 Linux 内核

13.3.1 调整 Linux 内核参数

13.3.2 构造 Linux 内核

13.3.3 内核没问题就不要改它

13.3.4 配置内核选项

13.3.5 构建 Linux 内核的二进制文件

13.3.6 添加 Linux 设备驱动程序

13.4 配置 Solaris 内核

13.4.1 Solaris 内核区

13.4.2 用/etc/system 配置内核

13.4.3 添加一个 Solaris 设备驱动程序

13.4.4 调试 Solaris 的配置

13.5 配置 HP-UX 内核

13.6 管理 AIX 内核

13.6.1 ODM

13.6.2 内核调配

13.7 可加载内核模块

13.7.1 Linux 的可加载内核模块

13.7.2 Solaris 的可加载内核模块

13.8 Linux udev 的意义和作用

13.8.1 Linux sysfs : 设备对外的窗口

13.8.2 用 udevadm 浏览设备

13.8.3 构造规则和固定不变的名字

13.9 推荐读物

13.10 习题

第二部分 网络管理技术

第 14 章 TCP/IP 网络

14.1 TCP/IP 和 Internet

14.1.1 Internet 的运行管理

14.1.2 网络的标准和文献

14.2 连网技术概述

14.2.1 IPv4 和 IPv6

14.2.2 分组和封装

14.2.3 以太网组帧

14.2.4 最大传输单位 (MTU)

14.3 分组地址

14.3.1 硬件 (MAC) 地址

14.3.2 IP 地址

14.3.3 主机名“地址”

14.3.4 端口

14.3.5 地址类型

14.4 IP 地址详解

14.4.1 IPv4 地址分类

14.4.2 子网

14.4.3 计算子网的技巧和工具

14.4.4 CIDR : 无类域间路由

14.4.5 地址分配

14.4.6 私用地址和 NAT

14.4.7 IPv6 地址

14.5 路由选择

14.5.1 路由表

14.5.2 ICMP 重定向

14.6 ARP : 地址解析协议

14.7 DHCP : 动态主机配置协议

14.7.1 DHCP 软件

14.7.2 DHCP 的工作方式

14.7.3 ISC 的 DHCP 服务器

14.8 安全问题

14.8.1 IP 转发

14.8.2 ICMP 重定向

14.8.3 源路由

14.8.4 广播 ping 和其他形式的定向广播

14.8.5 IP 欺骗

14.8.6 基于主机的防火墙

14.8.7 虚拟私用网络

14.9 PPP：点对点协议

14.10 基本的网络配置

14.10.1 分配主机名和 IP 地址

14.10.2 ifconfig：配置网络接口

14.10.3 网络硬件参数

14.10.4 route：配置静态路由

14.10.5 配置 DNS

14.11 特定于系统的网络配置

14.12 Linux 连网

14.12.1 NetworkManager

14.12.2 Debian 和 Ubuntu 的网络配置

14.12.3 SUSE 的网络配置

14.12.4 Red Hat 的网络配置

14.12.5 Linux 的网络硬件配置选项

14.12.6 Linux 的 TCP/IP 配置选项

14.12.7 有关安全的内核变量

14.12.8 Linux 的 NAT 和包过滤

14.13 Solaris 连网

14.13.1 基本网络配置

14.13.2 网络配置举例

14.13.3 DHCP 的配置

14.13.4 ndd : 调整 TCP/IP 和接口

14.13.5 安全

14.13.6 防火墙和过滤机制

14.13.7 NAT

14.13.8 Solaris 连网的特别之处

14.14 HP-UX 连网

14.14.1 基本网络配置

14.14.2 网络配置举例

14.14.3 DHCP 的配置

14.14.4 动态的重新配置和调整

14.14.5 安全、防火墙、过滤和 NAT

14.15 AIX 连网

14.15.1 no : 管理 AIX 的网络可配参数

14.16 推荐读物

14.17 习题

第 15 章 路由选择

15.1 近观包转发

15.2 路由守护进程和路由协议

15.2.1 距离向量协议

15.2.2 链路状态协议

15.2.3 代价度量

15.2.4 内部协议和外部协议

15.3 路由协议巡礼

15.3.1 RIP 和 RIPng : 路由信息协议

15.3.2 OSPF : 开放最短路径优先

15.3.4 EIGRP : 增强内部网关路由协议

15.3.5 IS-IS : ISO 的“标准”

15.3.6 RDP 和 NDP

15.3.7 BGP : 边界网关协议

15.4 路由策略的选择标准

15.5 路由守护进程

15.5.1 routed : 过时的 RIP 实现

15.5.2 gated : 第一代的多协议路由守护进程

15.5.3 Quagga : 主流的路由守护进程

15.5.4 ramd : HP-UX 的多协议路由系统

15.5.5 XORP : 计算机里的路由器

15.5.6 各操作系统的特性

15.6 思科路由器

15.7 推荐读物

15.8 习题

第 16 章 网络硬件

16.1 以太网：连网技术中的瑞士军刀

16.1.1 以太网的工作方式

16.1.2 以太网拓扑结构

16.1.3 无屏蔽双绞线

16.1.4 光纤

16.1.5 连接和扩展以太网

16.1.6 自动协商

16.1.7 以太网供电

16.1.8 巨大帧

16.2 无线：流动人士的 LAN

16.2.1 无线网络的安全

16.2.2 无线交换机和轻量级 AP

16.3 DSL 和 CM：最后一英里

16.4 网络测试和调试

16.5 建筑物布线

16.5.1 UTP 电缆的选择

16.5.2 到办公室的连接

16.5.3 布线标准

16.6 网络设计问题

16.6.1 网络结构与建筑物结构

16.6.2 扩展

16.6.3 拥塞

16.6.4 维护和建档

16.7 管理问题

16.8 推荐的厂商

16.8.1 电缆和连接器

16.8.2 测试仪器

16.8.3 路由器/交换机

16.9 推荐读物

16.10 习题

第 17 章 DNS：域名系统

17.1 谁需要 DNS

17.1.1 管理 DNS

17.2 DNS 的工作原理

17.2.1 资源记录

17.2.2 授权

17.2.3 缓存和效率

17.2.4 多重响应

17.3 DNS 速成

17.3.1 向 DNS 添加新机器

17.3.2 配置 DNS 客户机

17.4 域名服务器

17.4.1 权威与仅缓存服务器

17.4.2 递归和非递归服务器

17.5 DNS 名字空间

17.5.1 注册二级域名

17.5.2 创建子域

17.6 设计 DNS 环境

17.6.1 名字空间管理

17.6.2 权威服务器

17.6.3 缓存服务器

17.6.4 硬件要求

17.6.5 安全

17.6.6 总结

17.7 DNS 的新特性

17.8 DNS 数据库

17.8.1 区文件中的命令

17.8.2 资源记录

17.8.3 SOA 记录

17.8.4 NS 记录

17.8.5 A 记录

17.8.6 PTR 记录

17.8.7 MX 记录

17.8.8 CNAME 记录

17.8.9 巧用 CNAME

17.8.10 SRV 记录

17.8.11 TXT 记录

17.8.12 IPv6 资源记录

17.8.13 SPF 记录

17.8.14 DKIM 和 ADSP 记录

17.8.15 SSHFP 资源记录

17.8.16 粘合记录：区之间的链接

17.9 BIND 软件

17.9.1 判定版本

17.9.2 BIND 的组成

17.9.3 配置文件

17.9.4 include 语句

17.9.5 options 语句

17.9.6 acl 语句

17.9.7 (TSIG) key 语句

17.9.8 trusted-keys 语句

17.9.9 server 语句

17.9.10 masters 语句

17.9.11 logging 语句

17.9.12 statistics-channels 语句

17.9.13 zone 语句

17.9.14 rdc 的 controls 语句

17.9.15 分离式 DNS 和 view 语句

17.10 BIND 配置举例

17.10.1 localhost ~~区~~

17.10.2 一家小型的安全公司

17.10.3 ISC

17.11 NSD/Unbound 软件

17.11.1 安装和配置 NSD

17.11.2 运行 nsd

17.11.3 安装和配置 Unbound

17.12 更新区文件

17.12.1 区传送

17.12.2 BIND 的动态更新

17.13 安全问题

17.13.1 再谈 BIND 访问控制列表

17.13.2 开放的解析器

17.13.3 在监管环境下运行

17.13.4 使用 TSIG 和 TKEY 保障服务器与服务器之间通信的安全

17.13.5 为 BIND 设置 TSIG

17.13.6 NSD 里的 TSIG

17.13.7 DNSSEC

17.13.8 DNSSEC 策略

17.13.9 DNSSEC 资源记录

17.13.10 启用 DNSSEC

17.13.11 生成密钥对

17.13.12 区签名

17.13.13 DNSSEC 信任链

17.13.14 DLV : 域旁路认证

17.13.15 DNSSEC 密钥延期

17.13.16 DNSSEC 工具

17.13.17 调试 DNSSEC

17.14 微软和 DNS

17.15 测试和调试

17.15.1 BIND 的日志功能

17.15.2 NSD/Unbound 的日志功能

17.15.3 域名服务器的控制程序

17.15.4 域名服务器统计

17.15.5 用 dig 进行调试

17.15.6 残缺授权

17.15.7 其他 DNS 检查工具

17.15.8 性能问题

17.16 各操作系统的特定信息

17.16.1 Linux

17.16.2 Solaris

17.16.3 HP-UX

17.16.4 AIX

17.17 推荐读物

17.17.1 邮递列表和新闻组

17.17.2 书籍和其他文档

17.17.3 网上资源

17.17.4 RFC

17.18 习题

第 18 章 网络文件系统

18.1 NFS 概述

18.1.1 状态问题

18.1.2 性能问题

18.1.3 安全

18.2 NFS 的方法

18.2.1 NFS 协议的版本和历史

18.2.2 传输协议

18.2.3 状态

18.2.4 文件系统导出

18.2.5 文件上锁机制

18.2.6 安全问题

18.2.7 NFSv4 的标识映射

18.2.8 root 访问与 nobody 账号

18.2.9 NFSv4 的性能考虑

18.2.10 磁盘配额

18.3 服务器端 NFS

18.3.1 share 命令和 dfstab 文件 (Solaris/HP-UX)

18.3.2 exportfs 命令和 exports 文件 (Linux/AIX)

18.3.3 在 AIX 上导出文件系统

18.3.4 在 Linux 上导出文件系统

18.3.5 nfsd : 提供文件服务

18.4 客户端 NFS

18.4.1 在启动时挂载远程文件系统

18.4.2 端口安全限制

18.5 NFSv4 的标识映射

18.6 nfsstat : 转储 NFS 统计信息

18.7 专用 NFS 文件服务器

18.8 自动挂载

18.8.1 间接映射文件

18.8.2 直接映射文件

18.8.3 主控映射文件

18.8.4 可执行的映射文件

18.8.5 自动挂载的可见性

18.8.6 重复的文件系统和自动挂载

18.8.7 自动的 automount (除 Linux 之外其他系统上的 NFSv3)

18.8.8 Linux 的特定信息

18.9 推荐读物

18.10 习题

第 19 章 共享系统文件

19.1 共享什么

19.2 把文件复制到各处

19.2.1 NFS 的选项

19.2.2 “推”系统和“拉”系统

19.2.3 rdist : 推文件

19.2.4 rsync : 更安全地传输文件

19.2.5 拉文件

19.3 LDAP : 轻量级目录访问协议

19.3.1 LDAP 数据的结构

19.3.2 LDAP 的特点

19.3.3 LDAP 的文档和规范

19.3.4 OpenLDAP : 传统的开源 LDAP

19.3.5 389 Directory Server : 另一种开源 LDAP 服务器

19.3.6 用 LDAP 代替/etc/passwd 和/etc/group

19.3.7 LDAP 查询

19.3.8 LDAP 和安全

19.4 NIS : 网络信息服务

19.4.1 NIS 模型

19.4.2 理解 NIS 的工作方式

19.4.3 NIS 的安全

19.5 确定管理信息源的优先级

19.5.1 nscd : 缓存查找的结果

19.6 推荐读物

19.7 习题

第 20 章 电子邮件

20.1 邮件系统

20.1.1 用户代理

20.1.2 提交代理

20.1.3 传输代理

20.1.4 本地投递代理

20.1.5 消息库

20.1.6 访问代理

20.1.7 内容太多 , 时间太少

20.2 剖析邮件消息

20.2.1 阅读邮件信头

20.3 SMTP 协议

20.3.1 EHLO

20.3.2 SMTP 出错代码

20.3.3 SMTP 身份验证

20.4 邮件系统的设计

20.4.1 使用邮件服务器

20.5 邮件别名

20.5.1 从文件中获取别名

20.5.2 发邮件给文件

20.5.3 发邮件给程序

20.5.4 别名举例

20.5.5 散列的别名数据库

20.5.6 邮递列表和实现清单的软件

20.5.7 维护邮递列表的软件包

20.6 内容扫描：垃圾邮件和恶意软件

20.6.1 垃圾邮件

20.6.2 伪造邮件

20.6.3 消息隐私

20.6.4 垃圾邮件过滤

20.6.5 何时过滤

20.6.6 灰名单技术/DCC

20.6.7 SpamAssassin

20.6.8 黑名单

20.6.9 白名单

20.6.10 邮件过滤库

20.6.11 SPF 和 Sender ID

20.6.12 DomainKeys、DKIM 和 ADSP

20.6.13 MTA 特有的反垃圾邮件功能

20.6.14 MailScanner

20.6.15 amavisd-new

20.6.16 测试 MTA 的扫描效力

20.7 电子邮件配置

20.8 sendmail

20.8.1 开关文件

20.8.2 运行模式

20.8.3 邮件队列

20.9 配置 sendmail

20.9.1 m4 预处理器

20.9.2 sendmail 的配置

20.9.3 从.mc 样板文件构建配置文件

20.10 sendmail 基本配置原语

20.10.1 表和数据库

20.10.2 通用宏和功能

20.10.3 客户端选项

20.10.4 配置选项

20.10.5 sendmail 中处理垃圾邮件的功能

20.10.6 sendmail 中的 milter 配置

20.10.7 amavisd 和 sendmail 的连接

20.11 安全与 sendmail

20.11.1 所有权

20.11.2 权限

20.11.3 向文件和程序更安全地发邮件

20.11.4 隐私选项

20.11.5 运行一个 chroot 过的 sendmail (真正严格的要求)

20.11.6 拒绝服务攻击

20.11.7 SASL : 简单的身份验证和安全层

20.11.8 TLS : 传输层安全

20.12 sendmail 的性能

20.12.1 投递方式

20.12.2 队列分组和信封分割

20.12.3 队列运行器

20.12.4 控制平均负载

20.12.5 队列中无法投递的消息

20.12.6 内核调优

20.13 sendmail 测试和调试

20.13.1 队列监视

20.13.2 日志机制

20.14 Exim

20.14.1 安装 Exim

20.14.2 Exim 的启动脚本

20.14.3 Exim 的工具

20.14.4 Exim 的配置语言

20.14.5 Exim 的配置文件

20.14.6 全局的配置选项

20.14.7 ACL

20.14.8 ACL 内容扫描

20.14.9 身份验证器

20.14.10 路由

20.14.11 传输

20.14.12 重试配置

20.14.13 重写配置

20.14.14 本地扫描功能

20.14.15 amavisd 和 Exim 的连接

20.14.16 日志机制

20.14.17 调试机制

20.15 Postfix 邮件系统

20.15.1 Postfix 的体系结构

20.15.2 安全

20.15.3 Postfix 命令和文档

20.15.4 配置 Postfix

20.15.5 虚拟域

20.15.6 访问控制

20.15.7 反垃圾邮件和病毒

20.15.8 用 amavisd 做内容过滤

20.15.9 调试

20.16 DKIM 配置

20.16.1 DKIM：域密钥身份识别邮件

20.16.2 DKIM 邮件过滤

20.16.3 在 amavisd-new 中配置 DKIM

20.16.4 sendmail 中的 DKIM

20.16.5 Exim 中的 DKIM

20.16.6 Postfix 中的 DKIM

20.17 综合的电子邮件解决方案

20.18 推荐读物

20.19 习题

第 21 章 网络管理和调试

21.1 网络故障的检测

21.2 ping : 检查主机是否正常

21.3 SmokePing : ping 的累计统计

21.4 traceroute : 跟踪 IP 包

21.5 netstat : 获得网络统计信息

21.5.1 检查接口的配置信息

21.5.2 监视网络连接的状态

21.5.3 标识正在监听的网络服务

21.5.4 检查路由表

21.5.5 查看各种网络协议运行的统计信息

21.6 检查工作接口的活动

21.7 包嗅探器

21.7.1 tcpdump : 业界标准的包嗅探器

21.7.2 Wireshark 和 TShark : 增强型的 tcpdump

21.8 ICSI Netslyzr

21.9 网络管理协议

21.10 SNMP : 简单网络管理协议

21.10.1 SNMP 的组织结构

21.10.2 SNMP 协议的操作

21.10.3 RMON : 远程监视 MIB

21.11 NET-SNMP 代理程序

21.12 网络管理应用程序

21.12.1 NET-SNMP 工具

21.12.2 SNMP 数据的采集和绘图

21.12.3 Nagios : 基于事件的 SNMP 和服务监视工具

21.12.4 终极网络监测软件包 : 仍在寻觅

21.12.5 商业管理平台

21.13 NetFlow : 面向连接的监视

21.13.1 用 nfdump 和 Nfsen 监测 NetFlow 数据

21.13.2 在思科路由器上配置 NetFlow

21.14 推荐读物

21.15 习题

第 22 章 安全

22.1 UNIX 安全吗 ?

22.2 安全性是如何受损害的

22.2.1 社交工程

22.2.2 软件漏洞

22.2.3 配置错误

22.3 安全的技巧和思想

22.3.1 补丁

22.3.2 不必要的服务

22.3.3 远程的事件日志

22.3.4 备份

22.3.5 病毒和蠕虫

22.3.6 特洛伊木马

22.3.7 隐匿木马

22.3.8 包过滤

22.3.9 口令

22.3.10 警惕性

22.3.11 普遍原则

22.4 口令和用户账号

22.4.1 口令时限

22.4.2 组登录名和共享登录名

22.4.3 用户的 shell

22.4.4 获得 root 权限的办法

22.5 PAM：验证奇才

22.5.1 系统对 PAM 的支持

22.5.2 配置 PAM

22.5.3 Linux 上详细配置举例

22.6 setuid 程序

22.7 有效使用 chroot

22.8 加强安全的工具

22.8.1 nmap : 网络端口扫描程序

22.8.2 Nessus : 下一代的网络扫描程序

22.8.3 John the Ripper : 找出不安全的口令

22.8.4 hosts_acces : 主机访问控制

22.8.5 Bro : 可编程的网络入侵检测系统

22.8.6 Snort : 流行的网络入侵检测系统

22.8.7 OSSEC : 基于主机的入侵检测

22.9 强制访问控制 (MAC)

22.9.1 SELinux

22.10 加密的安全工具

22.10.1 Kerberos : 用于网络安全的统一方法

22.10.2 PGP : 很好的私密性

22.10.3 SSH : 安全的 shell

22.10.4 Stunnel

22.11 防火墙

22.11.1 包过滤防火墙

22.11.2 如何过滤服务

22.11.3 状态检查防火墙

22.11.4 防火墙保险吗

22.12 Linux 的防火墙功能

22.12.1 规则、链和表

22.12.2 规则目标

22.12.3 设置 iptables 防火墙

22.12.4 一个完整的例子

22.13 UNIX 系统的 IPFilter

22.14 VPN

22.14.1 IPSec 隧道

22.14.2 VPN 就够了吗

22.15 认证和标准

22.15.1 认证

22.15.2 安全标准

22.16 安全信息的来源

22.16.1 CERT : 卡耐基梅隆大学的注册服务商标

22.16.2 SecurityFocus.com 网站和 BugTraq 邮递列表

22.16.3 施耐德的安全博客

22.16.4 SANS : 系统管理、网络和安全协会

22.16.5 厂商特有的安全资源

22.16.6 其他邮递列表和网站

22.17 如何对付站点攻击

22.18 推荐读物

22.19 习题

第23章 Web 主机托管

23.1 Web 主机托管的基本知识

23.1.1 Web 上资源的位置

23.1.2 统一资源定位符

23.1.3 HTTP 工作原理

23.1.4 即时生成内容

23.1.5 应用服务器

23.1.6 负载均衡

23.2 HTTP 服务程序的安装

23.2.1 选择服务器软件

23.2.2 安装 Apache

23.2.3 配置 Apache

23.2.4 运行 Apache

23.2.5 分析日志文件

23.2.6 高性能主机托管的静态内容优化

23.3 虚拟接口

23.3.1 使用基于名字的虚拟主机

23.3.2 配置虚拟接口

23.3.3 告诉 Apache 有关虚拟接口的信息

23.4 SSL

23.4.1 产生签发证书的请求

23.4.2 配置 Apache 使用 SSL

23.5 缓存和代理服务程序

23.5.1 Squid 缓存和代理服务器

23.5.2 设置 Squid

23.5.3 Apache 的反向代理

23.6 超越上限

23.6.1 云计算

23.6.2 主机代管

23.6.3 内容分发网络

23.7 习题

第三部分 其他管理技术

第 24 章 虚拟化技术

24.1 虚拟技术的种类

24.1.1 全虚拟化

24.1.2 半虚拟化

24.1.3 操作系统级虚拟化

24.1.4 原生虚拟化

24.1.5 云计算

24.1.6 动态迁移

24.1.7 虚拟化技术比较

24.2 虚拟化技术的好处

24.3 实施方案

24.4 Linux 虚拟化

24.4.1 Xen 简介

24.4.2 Xen 基础知识

24.4.3 用 virt-install 安装 Xen 的 guest 系统

24.4.4 Xen 动态迁移

24.4.5 KVM

24.4.6 KVM 的安装和使用

24.5 Solaris 的 zone 和 container

24.6 AIX 的 WPAR

24.7 HP-UX 的 IVM

24.7.1 创建和安装虚拟机

24.8 VMWARE

24.9 亚马逊的 AWS

24.10 推荐读物

24.11 习题

第 25 章 X 窗口系统

25.1 X 显示管理器

25.2 运行一个 X 应用程序

25.2.1 环境变量 DISPLAY

25.2.2 客户机身份验证

25.2.3 用 SSH 转发 X 连接

25.3 配置 X 服务器

25.3.1 Device 段

25.3.2 Monitor 段

25.3.3 Screen 段

25.3.4 InputDevice 段

25.3.5 ServerLayout 段

25.3.6 xrandr : X 服务器的配置工具

25.3.7 内核模式设定

25.4 故障排查和调试

25.4.1 X 的特殊键盘组合

25.4.2 X 服务器出问题

25.5 桌面环境简述

25.5.1 KDE

25.5.2 GNOME

25.5.3 KDE 和 GNOME 谁更好

25.6 推荐读物

25.7 习题

第 26 章 打印

26.1 打印系统的体系结构

26.1.1 主要的打印系统

26.1.2 打印

26.2 CUPS 的打印

26.2.1 打印系统的界面

26.2.2 打印队列

26.2.3 多台打印机和打印队列

26.2.4 打印机实例

26.2.5 网络打印

26.2.6 过滤器

26.2.7 CUPS 服务器的管理

26.2.8 设置网络打印服务器

26.2.9 自动配置打印机

26.2.10 配置网络打印机

26.2.11 打印机的配置举例

26.2.12 设置打印机的类

26.2.13 关闭服务

26.2.14 其他配置工作

26.3 桌面打印环境

26.3.1 kprinter : 打印文档

26.3.2 Konqueror 和打印

26.4 SystemV 的打印

26.4.1 概述

26.4.2 打印目的地及打印类

26.4.3 lp 简述

26.4.4 lpsched 与 lpshut : 启动和停止打印

26.4.5 lpadmin : 配置打印环境

26.4.6 lpadmin 举例

26.4.7 lpstat : 获取状态信息

26.4.8 cancel : 删除打印作业

26.4.9 accept 和 reject : 控制假脱机处理

26.4.10 enable 和 disable : 控制打印

26.4.11 lpmove : 转移作业

26.4.12 接口程序

26.4.13 lp 系统混乱状况的处理方法

26.5 BSD 和 AIX 的打印

26.5.1 BSD 打印系统的体系结构概述

26.5.2 控制打印环境

26.5.3 lpd : 假脱机打印程序

26.5.4 lpr : 提交打印作业

26.5.5 lpq : 查看打印队列

26.5.6 lprm : 删除打印作业

26.5.7 lpc : 管理性修改

26.5.8 /etc/printcap 文件

26.5.9 printcap 变量

26.6 漫长和奇特的历程

26.6.1 打印的历史和打印系统的出现

26.6.2 打印机的多样性

26.7 常用的打印软件

26.8 打印机的语言

26.8.1 PostScript

26.8.2 PCL

26.8.3 PDF

26.8.4 XPS

26.8.5 PJL

26.8.6 打印机驱动程序及其对 PDL 的处理

26.9 PPD 文件

26.10 纸型

26.11 实际使用打印机的问题

26.11.1 打印机的选择

26.11.2 GDI 打印机

26.11.3 双面打印

26.11.4 其他打印机配件

26.11.5 串口和并口打印机

26.11.6 网络打印机

26.11.7 给打印机的其他建议

26.12 故障排查的技巧

26.12.1 重启打印守护进程

26.12.2 日志

26.12.3 直接打印的问题

26.12.4 网络打印的问题

26.12.5 发行版本特有的问题

26.13 推荐读物

26.14 习题

第 27 章 数据中心基础

27.1 数据中心的可靠性级别

27.2 冷却

27.2.1 电子设备

27.2.2 照明设备

27.2.3 操作人员

27.2.4 总的热负荷

27.2.5 冷热通道

27.2.6 湿度

27.2.7 环境监视

27.3 供电

27.3.1 机架的供电要求

27.3.2 kVA 和 kW

27.3.3 远程控制

27.4 机架

27.5 工具

27.6 推荐读物

27.7 习题

第 28 章 绿色 IT

28.1 绿色 IT 的兴起

28.2 绿色 IT 的生态金字塔

28.3 绿色 IT 策略：数据中心

28.3.1 应用合并

28.3.2 服务器合并

28.3.3 SAN 存储

28.3.4 服务器虚拟化

28.3.5 随用随开的服务器

28.3.6 细粒度使用和容量规划

28.3.7 优化能源的服务器配置

28.3.8 云计算

28.3.9 免费冷却

28.3.10 数据中心的高效冷却

28.3.11 停运时的降级模式

28.3.12 延长设备寿命

28.3.13 数据中心的较高温度

28.3.14 低功率设备

28.4 绿色 IT 策略：用户空间

28.5 绿色 IT 的朋友

28.6 习题

第 29 章 性能分析

29.1 做什么可以提高性能

29.2 影响性能的因素

29.3 如何分析性能问题

29.4 系统性能检查

29.4.1 盘点硬件

29.4.2 收集性能数据

29.4.3 CPU 使用情况分析

29.4.4 系统如何管理内存

29.4.5 内存使用情况分析

29.4.6 磁盘 I/O 分析

29.4.7 xdd : 分析磁盘子系统的性能

29.4.8 sar : 连续采集和报告统计信息

29.4.9 nmon 和 nmon_analyser : AIX 上的监视工具

29.4.10 选择 Linux 的 I/O 调度器

29.4.11 oprofile : 详细剖析 Linux 系统

29.5 求助！系统为何越来越慢

29.6 推荐读物

29.7 习题

第 30 章 同 Windows 协作

30.1 从 Windows 登录到 UNIX 系统

30.2 远程桌面访问

30.2.1 在 Windows 计算机上运行 X 服务器

30.2.2 VNC : 虚拟网络计算

30.2.3 Windows RDP : 远程桌面协议

30.3 运行 Windows 和类似 Windows 的应用

30.3.1 双重引导 , 为何不该用

30.3.2 微软 Office 的替代软件

30.4 在 Windows 上使用命令行工具

30.5 Windows 遵守电子邮件和 Web 标准

30.6 通过 Samba 和 CIFS 共享文件

30.6.1 Samba : UNIX 的 CIFS 服务器

30.6.2 Samba 的安装

30.6.3 文件名编码

30.6.4 用户身份验证

30.6.5 基本的文件共享

30.6.6 用户组共享

30.6.7 用微软的 DFS 做透明重定向

30.6.8 smbclient : 简单的 CIFS 客户端

30.6.9 Linux 的客户端对 CIFS 的支持

30.7 用 Samba 共享打印机

30.7.1 从 Windows 安装打印机驱动程序

30.7.2 从命令行安装打印机驱动程序

30.8 调试 Samba

30.9 Active Directory 身份验证

30.9.1 准备好集成 AD

30.9.2 配置 Kerberos

30.9.3 Samba 作为 Active Directory 的域成员

30.9.4 配置 PAM

30.9.5 winbind 的备选方案

30.10 推荐读物

30.11 习题

第 31 章 串行设备和串行终端

31.1 RS-232C 标准

31.2 备选连接器

31.2.1 DB-9 连接器

31.2.2 RJ-45 连接器

31.3 硬载波和软载波

31.4 硬流控

31.5 串行设备文件

31.6 setserial：把串口参数通知给驱动程序

31.7 伪终端

31.8 硬件终端的配置

31.8.1 登录过程

31.8.2 /etc/ttypype 文件

31.8.3 /etc/gettytab 文件

31.8.4 /etc/gettydefs 文件

31.8.5 /etc/inittab 文件

31.8.6 Linux 上的 getty 配置

31.8.7 Ubuntu 的 Upstart

31.8.8 Solaris 和 sacadm

31.9 特殊字符和终端驱动程序

31.10 stty : 设置终端的选项

31.11 tset : 自动设置选项

31.12 僵住的终端

31.13 调试串行线

31.14 连接到串行设备的控制台

31.15 习题

第 32 章 管理、政策与政治

32.1 IT 的目标

32.1.1 预算和支出

32.1.2 IT 政策

32.1.3 SLA

32.2 IT 职能机构的组成

32.2.1 基础：工单和任务管理系统

32.2.2 工单系统的常见功能

32.2.3 工单的所有权

32.2.4 用户对工单系统的接受程度

32.2.5 工单系统举例

32.2.6 工单分派

32.2.7 IT 内部的技能培养

32.2.8 时间管理

32.3 咨询组

32.3.1 服务范围

32.3.2 咨询可用性

32.3.3 咨询上瘾

32.4 企业构架师

32.4.1 过程可再现

32.4.2 留下记录

32.4.3 认可文档的重要性

32.4.4 定制和编程

32.4.5 保持系统干净整洁

32.5 运行组

32.5.1 瞄准最短停机时间

32.5.2 依靠文档

32.5.3 重用或淘汰老硬件

32.5.4 维护本地文档

32.5.5 保持环境独立

32.5.6 自动化

32.6 管理的职能

32.6.1 领导

32.6.2 人事管理

32.6.3 聘用

32.6.4 解聘

32.6.5 人事管理的机制

32.6.6 质量控制

32.6.7 管理但别管闲事

32.6.8 社区关系

32.6.9 管理上级

32.6.10 采购

32.6.11 化解矛盾

32.7 政策和规程

32.7.1 政策和规程之间的区别

32.7.2 政策的最佳实践

32.7.3 规程

32.8 灾难恢复

32.8.1 风险评估

32.8.2 灾难管理

32.8.3 处理灾难的人员准备

32.8.4 电源和 HVAC

32.8.5 互联网连接的冗余性

32.8.6 安全事件

32.9 合规：规章与标准

32.9.1 ITIL：信息技术基础设施库

32.9.2 NIST：国家标准和技术研究所

32.10 法律问题（美国）

32.10.1 隐私

32.10.2 落实政策

32.10.3 控制=义务

32.10.4 软件许可证

32.11 组织、会议及其他资源

32.12 推荐读物

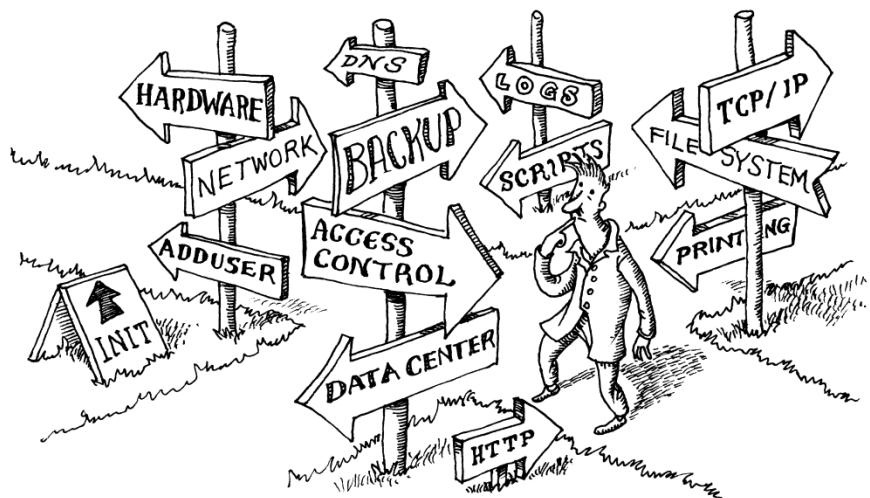
32.13 习题

附录 A 系统管理简史

附录 B 为 AIX 辩护

跋

第1章 从何处入手



现如今，有关 UNIX 和 Linux 的知识太多太多，由手册、博客、期刊、图书以及其他参考资料构成了浩如烟海的知识库，因此，我们在策划本书内容的时候，就从上述的知识库中遴选出沧海一粟，力求专门满足系统管理员的需求。

首先，本书是一本入门培训指南。书中探讨了主要的管理性系统，先区分它们彼此间的不同之处，然后又阐述它们如何协同工作。在很多情况

下，同一概念有若干种不同的实现，系统管理员必须要在其中做出自己的选择，我们则介绍每种实现的优缺点。

其次，本书是一本快速参考手册。书中总结了很多知识，在各种常见的 UNIX 和 Linux 系统上，如果系统管理员要履行日常职责，就需要掌握它们。例如，**ps** 命令，它能够给出正在运行的进程的状态。在 Linux 系统上，这条命令支持 80 多个命令行参数。但只要用极少的命令行参数组合，就能满足系统管理员 99% 的需求；具体请参见本书 5.7 节的内容。

最后，本书集中介绍对企业级服务器和网络的系统管理。换言之，就是对重大系统的管理。配置一个单机桌面系统很简单；但要保持一个虚拟化网络平稳运行，就会困难得多，因为要面临负载高峰、硬盘故障和恶意攻击等问题。我们介绍了技术和经验方法，它们能帮助网络从灾难中恢复过来。我们还帮助读者选择解决方案，方案会随着系统规模、复杂度以及异构性的增大而扩展。

我们不是说要绝对客观地实现这些目标，但是我们认为，透过字里行间的意思，我们已经相当清楚地讲出了我们自己的偏好。在系统管理领域里，存在这样一个有趣的现象：“该由什么构成最合适的策略和步骤？”对于这一问题，理性的人们可能会有相当不同的观点。我们把我们的主观意见作为原始数据提供给读者。对于我们的建议，读者必须自己决定能接受多少，并在多大程度上用于自己的环境。

1.1 系统管理员的基本任务

在维基百科上，对“系统管理员”一词的解释中，有一段很不错的讨论，探讨了通常认为的系统管理工作应该包括的任务。对于系统管理和软件开发之间的界限，这个名词解释网页现在划分得很清楚，但是根据我们的经验，在编写脚本程序上，专业的系统管理员花了很多自己的时间。读者可参考第 2 章来了解更多有关脚本编程的知识。虽说这并不会让系统管理员本身变成开发人员，但确实意味着系统管理员同样需要具备分析和结构设计的能力。

下面各小节简要介绍了要求系统管理员履行的一些主要任务。这些职责未必是由一个人来执行的，在许多地方，这项工作会分配给几个人。不过，确实至少要有一个人懂得全部工作，并确保每项任务都能正确完成。

1.1.1 账号管理

系统管理员负责为新用户增设账号，将不再活动的用户账号删除，还要处理在账号存在期间所有与该账号有关的事务（例如，忘记口令）。增删用户的处理过程可以自动进行，但在增设新用户之前，仍然必须做出某些管理上的决定（把用户的主目录放在什么位置、在哪些机器上创建账号等）。参考第 7 章了解有关添加新用户的更多知识。

当某个用户不应该再访问系统时，必须禁用该用户的账号。该账号拥有的所有文件必须在备份后予以清除，以使系统不会随着时间的增长而积累不想要的信息。

1.1.2 增删硬件

购买了新硬件或把硬件从一台机器移到另外一台机器上时，必须配置系统，使之识别并使用该硬件。硬件支持的任务可能很简单，例如添加打印机，也可能是更复杂的任务，例如添加磁盘阵列。参考第 8 章、第 13 章和第 26 章了解有关这些专题的更多知识。

现如今，虚拟化技术已经运用到企业级计算环境中来了，硬件的配置也比以前更复杂。设备可能要安装在虚拟化层次结构中的多个层面，系统管理员需要制定规范的策略，让硬件能够安全和公平地被共享。

1.1.3 执行备份

执行备份可能是系统管理员最重要的工作了，这也是经常被忽视，或者不经心去做的工作。

备份工作耗时且单调，但又绝对必要。备份可以自动进行，且能授权给下属去做，但系统管理员仍然要确保备份工作在按计划正确进行（备份介质确实可以恢复出原文件来）。参考第 10 章来了解有关备份的更多知识。

1.1.4 安装和更新软件

在获得新软件以后，必须安装并测试它，而且经常要在几种类型的操作系统和硬件下进行安装和测试。一旦软件能够正常工作，就必须通知用户可以用这个软件，还要告诉用户这个软件的位置。在补丁和安全升级发布出来的同时，也必须平稳地将它们融入到本地环境

里。参考第 12 章来了解有关软件管理的更多知识。

应该把本地软件和管理脚本正确地打包和管理，使之适用于本地系统自身采取的升级机制

随着该软件向前发展，新的发布版本应该先安排测试，然后再部署到整个站点上去。

1.1.5 监视系统

大型的部署环境需要时刻进行监视。除非事态严重，否则不要指望用户向系统管理员报告问题。比起花时间写说明报告问题，绕过问题通常要快得多，所以用户往往怎么省事怎么来。

定期做检查，确保电子邮件和 Web 服务正确运行；查看日志文件，掌握故障出现的早期迹象；确保本地网络都连接正确；监视系统资源（例如磁盘空间）的可用性。所有这些工作都很有可能自动执行，各种各样现成的监视系统能够帮助系统管理员完成这项任务。

1.1.6 故障诊断

系统发生故障在所难免。充当技工来诊断故障，在必要时请来专家排除故障等，都成了管理员的工作职责。找出问题往往比修复难上十倍。

1.1.7 维护本地文档

系统为适合单位的需要而逐渐改造，随着改造的进行，该系统便开始变得和文档当初所描述的那个基本系统不一样。既然系统管理员负责做这些定制工作，那么写文档讲清楚变化

也是管理员的职责。这项职责包括把电缆的走线位置、电缆连接方式等信息写入文档，保留所有硬件的维护记录，记录备份的状态，编写本地处理过程和策略的文档等。参考 32.5 节来了解有关文档的建议。

1.1.8 时刻警惕系统安全

系统管理员必须实施某种安全策略，并且做定期检查，确保系统安全没有受到侵害。在安全要求低的系统上，这项工作任务可能仅仅包括对非授权访问进行粗略的检查。在安全要求高的系统上，这可能包括仔细布置由陷阱和审计程序构建的监视网。有关安全主题的更多知识，请参考第 22 章。

1.1.9 救火

尽管在系统管理员工作的规定范围之内，很少会包括帮助用户解决各种问题，但对于大多数系统管理员而言，这通常占据了他们日常工作的很大一部分。系统管理员往往遭到形形色色问题的狂轰滥炸，问题从“我的程序昨天正常工作了，今天却不干活了！您改变了什么？”，到“我把咖啡洒在键盘上了！我应该把水倒到键盘上洗掉咖啡么？”，什么都有。在大多数情况下，系统管理员对这些事情的响应，要比他可能具备的任何实际技术技能，更能够让别人感觉到他作为一个系统管理员的价值所在。要么对此愤愤不平，要么就欣然接受这样的事实：深更半夜花 5 个小时调试问题，和妥善处理一张故障工单相比，在绩效评定上都一样。

1.2 读者的知识背景

本书假定读者已经具备一定程度的 Linux 或者 UNIX 经验。尤其要从用户的角度对系统的风格有个一般性的概念，因为我们不再复述这方面的材料。有一些优秀的书籍可以帮助读者迅速掌握这些概念，参见本章 1.14 节。

即便在现今有三维图形显示功能的桌面计算机上，用于 UNIX 和 Linux 系统管理的 GUI 工具与其下层丰富的软件相比，依然显得相当简单。在现实世界里，我们仍然要通过编辑配置文件和编写脚本来完成管理工作，因此读者需要习惯于使用某种命令行的 shell 和某种文本编辑器。

编辑器可以是像 **gedit** 那样的 GUI 工具，也可以是像 **vi** 或者 **emacs** 那样的命令行工具。像 Microsoft Word 和 OpenOffice Writer 这样的字处理程序和文本编辑器差别很大，对于系统管理工作几乎没有什么用处。命令行工具有个优势，因为它们可以通过一条 SSH 连接运行，还可以在出问题不能启动的系统上使用；而且不需要用图形窗口系统。对于系统管理员常做的微小编辑工作来说，它们的运行速度也快得多。

我们推荐读者学会使用 **vi**（现在最常见的是它的一种经过重写的形式，即 **vim**），它是所有 UNIX 和 Linux 系统上的标准软件。尽管同 **emacs** 这样的华丽软件相比，**vi** 可能显得有些平淡，但 **vi** 的功能还是非常强大和完善的。GNU 的 **nano** 是一种简单、易上手的“入门编辑器”，它有屏幕提示功能。选用非标准的编辑器则要小心，如果读者已经对这样一种编辑器“上了瘾”，那么很快就会对它感到厌烦，因为需要把它带在身边，以便在每个新系统上都安装一次。

系统管理工作的主要支柱之一（并且也是贯穿本书的一个主题），是使用脚本来自动完成管理任务。要成为一名高效率的系统管理员，必须能够阅读并修改 **Perl** 和 **bash/sh** 脚本

参考第 2 章，了解更多有关脚本编程的知识。

对于编写新脚本，我们推荐使用 **Perl** 或者 **Python**。作为一种编程语言，**Perl** 有些奇怪。

不过，它确实包含了许多对于管理员来说是必不可少的特性。由 O'Reilly 出版、Larry Wall 等编写的 *Programming Perl* 一书是 Perl 的标准教程；该书也是技术图书的典范。

本章 1.14 节给出了该书的完整信息。

许多系统管理员更愿意用 Python 而不是 Perl，而且我们也知道有些地方正在一起努力从 Perl 转向 Python。Python 是一种比 Perl 更优美的语言，而且 Python 脚本往往可读性更好，也更容易维护（正如亚马逊公司的 Steve Yegge 所说，“长久以来，Python 社区都是从 Perl 阵营逃出来的人的庇护所，这些人吞下红色的药丸，从 Perl 的 Matrix 里苏醒过来”）。从 python.org/doc/Comparisons.html 可以找到一组有用的链接，把 Python 同其他脚本语言（包括 Perl）进行了一番比较。

Ruby 是一款正处在上升期的语言，它保留了 Perl 的许多强大功能，又避免了 Perl 的一些语法缺陷，还增加了现代的面向对象特性。它作为系统管理员的一款脚本语言，其根基还不深，但在未来几年内，这种情况可能会有所改变。

我们还推荐读者学会 **expect**，它与其说是一款编程语言，倒不如说是用于驱动交互式程序执行的前端语言。它是一款高效率的“胶水”语言，能够替代一些复杂的脚本编程。

expect 脚本学起来很容易。

本书第 2 章总结了有关 **bash**、Perl 和 Python 脚本编程最重要的知识。这一章还复习了正则表达式（文本匹配模式）和一些 shell 的惯用法，它们对于系统管理工作来说很有用。

1.3 UNIX 和 Linux 之间的摩擦

因为 UNIX 和 Linux 系统颇为相似，所以本书会同时讲解这两种系统的管理。遗憾的是，在一句话里同时用 UNIX 和 Linux 这两个称谓，就仿佛一脚踏入了政治漩涡，或许也像是陷入了一大片泥潭。不过，既然 UNIX 和 Linux 之间的关系似乎呈现出某些混淆和冲突，所以要避开不谈我们的立场也比较困难。下面是我们的观点，还有我们对事实的简要说明有关 UNIX 和 Linux 历史的更多资料，可参考本书后面的介绍。

Linux 重新实现并优化了 UNIX 内核。Linux 遵循 POSIX 标准，能够在几种硬件平台上运行，兼容现有的大多数 UNIX 软件。它同大多数（但不是全部）别的 UNIX 变体不一样，区别之处在于，它是自由的，开放源代码的，而且是由成百上千不同的个人和组织无私奉献协同开发出来的。与此同时，传统的 UNIX 厂商则继续改善自己的系统，所以肯定存在一些领域，在这些领域内，商业 UNIX 系统比 Linux 强。

无论两种系统有什么相对而言的优缺点，Linux 从法律上、从开发上以及从历史上看，都和 UNIX 有着明显的区别，因此不能把 Linux 称为 UNIX，或者 UNIX 的一种版本。这样做会忽视 Linux 社区所做的工作和创新。同时，坚持认为 Linux 不是 UNIX 的话，又会有点儿误导性。如果做出来的东西走路像鸭子，叫起来像鸭子，那或许可以说，造的就是鸭子
〔译者注：源自一句西方谚语。If it looks like a duck, walks like a duck, and quacks like a duck, it must be a duck! 看起来像鸭子，走起路来像鸭子，叫起来也像鸭子，那一定是鸭子！这里的意思是，不能因为 Linux 像 UNIX，就把 Linux 当成 UNIX。〕

即便在 Linux 阵营里也存在分歧。有人举证说，把 Linux 发行版本简称为“Linux”，那么开发人员在内核之外运行的那些软件（在一般系统上实际是很大一部分软件）上投入的工作，就没有得到承认。偏偏不巧，最常推荐的替代称谓“GNU/Linux”也有其自己的政治包

袱，所以只有 Debian 这个发行版本才正式采用它。维基百科上有关“GNU/Linux 命名争议”的条目列举出了争执双方的论点¹。有意思的是，现如今，甚至在大多数 UNIX 系统上，也是开源软件的使用占据主导，但还没有人提 GNU/UNIX 这样的说法²。

Linux 软件就是 UNIX 软件。多亏有了 GNU 计划，使 UNIX 系统更富价值的重要软件大多都以某种开放源代码的形式被开发出来³。在 Linux 和非 Linux 系统上运行的代码是相同的。比如说，Web 服务器 Apache 全然不在意它是在 Linux 上还是在 HP-UX 上运行。从应用软件的角度来看，Linux 只不过是得到最好支持的 UNIX 变体之一罢了。

另外还值得一提的是，Linux 不是现今世界唯一的自由 UNIX 版本。OpenSolaris 系统也是自由和开源的，虽然在某些讲求纯开源的人眼里来看，它的许可证条款存有疑点 FreeBSD、NetBSD 和 OpenBSD——都是从 UC Berkeley（加州大学伯克利分校）的 BSD（伯克利软件发布，即 Berkeley Software Distribution）发展而来的分支——都有其各自的热诚追随者。虽然这些操作系统从第三方软件厂商得到的支持有点儿少，但它们在功能和可靠性上却往往可以同 Linux 相媲美。

UNIX 和 Linux 系统用于生产环境已经有许多年了，而且它们都表现不错⁴。现在要在它们之间做出选择，更多考虑的是软件打包机制、能获得的支持以及使用习惯的惰性，而不是品质或者成熟度实际有什么样的差别。

本书里对“Linux”的评论一般也都适用于各种 Linux 发行版本，但不适用于传统的 UNIX 变体。“UNIX”的含义稍有点儿多变，因为我们偶尔也会用在包括 Linux 在内的所有 UNIX 变

1. 因为维基百科包含 Linux 的信息，必然会经常提到 Linux，这场称谓上的争议对维基百科自身而言也有特殊关系。针对维基百科文章的讨论也值得一读。

2. 毕竟 GNU 不是 UNIX！(GNU's Not UNIX!)

3. 对于我们似乎把世界上大多数自由软件的创作功劳都算在 GNU 头上的做法，我们的技术审稿人中有几位持有异议。我们的态度并不是这样的！但是，推动自由软件作为社会事业的思想，对许可证制度发出质疑，促进自由和非自由软件之间的相互交流，GNU 肯定比其他任何组织做的工作都要多。

4. 我们说的“工作环境”是一个单位赖以完成实际工作的环境（与之对照的是测试、研究或者开发环境）。

体所共有的属性上（例如，“UNIX 文件权限”）。为了避免混淆，在想表达二者都有的情况下，我们通常说“UNIX 和 Linux”。

1.4 Linux 的发行版本

所有的 Linux 发行版本共享相同的内核源，但是和内核一起的辅助软件则随着发行版本的不同而有很大不同。各发行版本在其侧重点、支持和流行程度上有所不同。独立的 Linux 发行版本在数量上仍然有数百种之多，但是我们认为，未来 5 年内，基于 Debian、Red Hat 和 SUSE 源的那些发行版本将在生产环境中占据优势地位。

各种 Linux 发行版本之间存在差别，但是差别并没有大到远隔千山万水的地步。实际上，为什么会有这么多不同的发行版本，而每种发行版本都把“易于安装”和“海量软件库”作为其特色来宣传，这就挺难解释的。于是难免会得出这样的结论，人们只是喜欢“做”出新的 Linux 发行版本而已。

许多较小的发行版本在成熟度和完善性方面极具竞争力。所有的主流发行版本，包括二线的发行版本，都有相当简便的安装步骤、调配良好的桌面环境，以及某种形式的软件包管理机制。大多数发行版本还可以让用户从 DVD 光盘上直接启动，这对于调试来说很方便，而且还是一个对关注的新发行版本很快了解的好办法。

既然我们在本书中所关注的是大规模安装的管理问题，那么我们就会侧重考虑像 RHEL (Red Hat Enterprise Linux) 这样用于管理计算机网络的发行版本。有些发行版本的设计考虑到了生产环境，其他的发行版本则没有。面向生产环境的系统所提供的额外支持功能，对于方便系统管理工作来说，带来了巨大的不同。

当用户采用了一种发行版本之后，就是在某个特定发行商的做事方式上进行了投资。不要只看到软件安装以后的功能，而是要明智地考虑到，自己的单位和那家发行商要在未来的数年里共事。

因此要取得一些重要问题的答案。

- 该发行版本能够在今后 5 年内继续存在吗？
- 该发行版本会持续有最新的安全补丁吗？
- 该发行版本会迅速发布更新软件吗？
- 如果我有问题，发行商会搭理我吗？

有了这样的认识再来看，有些更有意思、规模更小的发行版本就不太有吸引力了。但是也别把它们排除在外：例如 E*Trade 就运行在 Gentoo Linux 上。

最有生命力的发行版本并不一定是最商业化的发行版本。例如，我们认为 Debian (对，就是 Debian GNU/Linux！) 在相当长的一段时间里是会存在的，尽管 Debian 并不是一家公司，既不销售任何东西，也不正式提供需要的支持。Debian 本身不是一种广泛使用的发行版本，但是它得益于有一个坚定的贡献群体，也得益于 Ubuntu 这个基于 Debian 的发行版本的盛行。

表 1.1 列举了最流行的主流发行版本。

表 1.1 最流行的通用 Linux 发行版本		
发行版本	Web 站点	说 明
CentOS	centos.org	模仿 Red Hat Enterprise Linux 的免费发行版本
Debian	debian.org	一种流行的非商业性质的发行版本
Fedora	fedoraproject.com	从 Red Hat Linux 分出的非商业运作的发行版本
Gentoo	gentoo.org	用户用源代码自行编译和优化的发行版本
Linux Mint	linuxmint.com	基于 Ubuntu，有优美的应用
Mandriva	mandriva.com	历史悠久，“容易上手尝试”

openSUSE	opensuse.org	模仿 SUSE Linux Enterprise 的免费发行版本
----------	--------------	----------------------------------

续表

发行版本	Web 站点	说 明
Oracle Enterprise Linux	oracle.com	Oracle 公司提供支持的 RHEL 版本
PCLinuxOS	pclinuxos.com	Mandriva 的分支，面向 KDE 的发行版本
Red Flag	redflag-linux.com	中文的发行版本，类似于 Red Hat
Red Hat Enterprise	redhat.com	Red Hat Linux 的公司商业化运作的发行版本
Slackware	slackware.com	古老而长寿的发行版本
SUSE Linux Enterprise	novell.com/linux	在欧洲流行的多语言发行版本
Ubuntu	ubuntu.com	Debian 的精炼版本

从 linux.org/dist、lwn.net/Distributions 或者 distrowatch.com 可以找到很全的发行版本清单，其中包括许多非英语的发行版本。

1.5 本书使用的示例系统

我们已经选择了 3 款 Linux 发行版本和 3 款 UNIX 变体，把它们作为我们全书讨论所采用的示例系统：Ubuntu Linux、openSUSE、Red Hat Enterprise Linux、Solaris、HP-UX 和 AIX。这些系统是整个市场的代表，它们加起来占据了现如今投入使用的系统的绝大多数。

除非指定一种特定发行版本，否则本书中的内容一般都适用于我们用作示例的所有发行版

本。只适用于某种特定发行版本的细节将采用发行商的徽标来标记：

Ubuntu® 9.10 “Karmic Koala”

openSUSE® 11.2

Red Hat® Enterprise Linux® 5.5

Solaris™ 11 and OpenSolaris™ 2009.06

HP-UX® 11i v3

AIX® 6.1

这些徽标的使用已经分别得到其拥有者的善意许可。不过，这些厂商没有复查或者提供本书的内容。下面的段落对每种示例系统稍加说明。

1.5.1 用作示例的 Linux 发行版本



针对 Linux 而不是任何特定发行版本的知识，左边用 Tux 企鹅徽标表示。

Ubuntu 的发行版本都保持着支持 Linux 社群开发和开放访问的思想意识，所以它不存在任何有关自身哪部分自由、哪部分可以重新发布的问题。Ubuntu 目前还在享受着南非企业家 Mark Shuttleworth 的慈善资助。Ubuntu 基于 Debian 这个发行版本，使用 Debian 的软件打包系统。它发展成两种形式：桌面版本（Desktop Edition）；服务器版本（Server Edition）。两个版本实质上是类似的，但是服务器版本的内核针对服务器的使用预先做了调配，它不能安装 GUI，或者像 OpenOffice 这样的 GUI

应用软件。



SUSE 现在属于 Novell 公司，已经走上了和 Red Hat 一样的路子，分成两种相关的发行版本：一种叫做 openSUSE，只包含自由软件；另一种叫做 SUSE Linux Enterprise，要花钱购买，包含正规的支持渠道，并且提供了一些额外的东西。本书的内容不专门针对哪一种 SUSE 的发行版本，所以我们就直接把它们都统称为“SUSE”。

在过去将近 10 年的时间里，Red Hat 公司一直是 Linux 业界的主导力量，它的发行版本在北美地区得到广泛使用。2003 年，Red Hat Linux 发行版本分成了两种，一种是以生产应用为中心的产品线，叫做 Red Hat Enterprise Linux（我们在本书里称之为 RHEL 或者 Red Hat），另一种是依托 Linux 社群的开发项目，叫做 Fedora。在技术、经济、后勤和法律等众多因素的综合作用下，才促成这次分裂。

这两种发行版本一开始曾经比较相似，但是 Fedora 在过去 5 年里做出了一些重大改变，这两种系统现在不再以任何有意义的方式保持同步了。RHEL 提供大量支持，而且稳定性好，但是如果向 Red Hat 公司支付许可证费，则无法有效地使用它。

CentOS 项目（centos.org）收集了 Red Hat 为遵守各种许可证协议（最知名的就是 GNU 的 GPL 许可证）而必须公布的源代码，把这些源代码整理成一个类似于 RHEL 但却免费的完整发行版本。这个发行版本没有 Red Hat 商标，也没有个别的

一些专有工具，但是其他各方面都和

1.5.2 用作示例的 UNIX 发行版本

Solaris 是 System V 的一种变体，它从 Sun 公司获得了许多扩展。Sun 公司以前很有名，现在则是 Oracle 公司的一部分⁵。Sun UNIX（在 20 世纪 80 年代中期 Solaris 就曾叫这个名字）最早源于 Berkeley UNIX，但是 Sun 和 AT&T 之间的合作伙伴关系（现在这种关系已经成为历史）导致其代码基础发生了变化。Solaris 可以在很多不同的硬件平台上运行，其中最著名的要数 Intel x86 和 SPARC。Solaris 在 Sun 公司的手中可以自由下载和使用。但是 Oracle 改变了这项政策，现在下载的 Solaris 被标为 90 天免费试用版。OpenSolaris 作为一种明确说明是 Solaris 的开源版本，它的出现又让情况复杂化了。此刻（2010 年中期），Oracle 对 Solaris 和 OpenSolaris 的确切规划尚不明朗。

预计在今年的某个时刻发布 Solaris 11，迄今为止的种种迹象表明，它会和 OpenSolaris 很相近。本书中我们称为“Solaris”的是一种混合系统，它基于产品级的 Solaris 10 和 OpenSolaris 的发布版，并根据我们在 Oracle 内的“卧底”提供的指导做了调整。在不多的几个地方，我们会指出是针对 Solaris 10 还是 OpenSolaris。

HP-UX 基于 System V，并且只用于 HP 公司的硬件平台。在 UNIX 族谱上它比 Solaris 和 AIX 离源头更近，但是 HP 一直紧跟操作系统界的发展脚步，给 HP-UX 加入了各种各样的增强功能。现在 HP 也开始支持 Linux 了，HP-UX 的未来有点儿不太清晰。

5. 参考书后有关 BSD、System V 和 UNIX 历史的大体介绍。

IBM 的 AIX 一开始是伯克利 4.2BSD 的一种变体，但是到 1994 年的第 4 版，这个操作系统的大部分都迁移到了 System V 上面。现在 AIX 和前面两种源头系统的距离都已经相当遥远了。

一般而言，我们对 AIX 的印象是，比起大多数 UNIX 变体来说，AIX 和其他系统的交流很少。AIX 也似乎受到了 IBM 的大型机和 AS/400 操作系统某些不好的影响，变得有点儿强人所难，它从后者那里继承了像 ODM (ObjectDataManager，对象数据管理器)、使用配置命令而不是配置文件，以及 SMIT 管理界面等这样的传统。随着时间的推移，人们或许会厚道地说，AIX 已经变得越来越像它自己了[译者注：作者这里的意思是说，AIX 缺乏和其他系统的相互交流，变得很自我，和别的系统越来越不一样]。

在过去 10 年中的大多数时间里，IBM 为营销自己的硬件设备，对操作系统的选择一直力求保持一种不明朗的态度，这点显得很有意思。IBM 继续开发和推广 AIX，但是它也同 Red Hat 和 Novell 形成伙伴关系，确保这两家的 Linux 发行版本能够顺利地运行在 IBM 的硬件上。静观这种做法在未来几年会带来怎样的效果也挺有意思的。

1.6 特定于系统的管理工具

在现代的操作系统内，包含有各种各样可视化的工具和控制面板（比如 SUSE 的 YaST2 和 IBM 的 SMIT），帮助用户配置或者管理选定的系统功能。这些工具非常有用，特别是对于管理员新手来说更是如此，但是它们也倾向于不能完整地体现下层软件的实际操作。它

们让管理工作变得容易，但大多数又变得不那么权威。

在本书中，出于下面几个原因，我们要介绍可视化工具调用的底层机制，而不是工具本身。第一，可视化工具趋于专有化（或者至少是趋于为系统所特有）。它们给配置过程带来了变数，而在各个系统之间，这些过程在较低层面实际上可能是相当一致的。第二，我们相信，对于系统管理员来说，准确了解他们的系统是如何工作，这点非常重要。当系统发生故障的时候，可视化工具通常对于确定并修正问题来说没有什么帮助。最后，手工配置往往更好、更快、更灵活、更可靠，而且更易于用脚本实现。

1.7 表示法和印刷约定

在本书中，文件名、命令和命令的参数用粗体（黑体）来表示；占位符（例如，不应该按字面直接照搬的命令参数）用斜体来表示。例如，在下面的命令中：

```
cp file directory
```

用户应该用实际的文件名和实际目录来分别替换 *file* 和 *directory*。

配置文件和终端会话的节录用等宽字体来显示⁶。有时候，我们用斜体字来给交互式会话做注释，例如：

```
$ grep Bob /pub/phonelist          /* 查找Bob 的电话号码 */  
Bob Knowles 555-2834  
Bob Smith 555-2311
```

在这些特定情况以外，我们试图把特殊字体和格式约定减到最少，只要我们这样做不会影

⁶. 实际上，并不是真正的等宽字体，但看起来是的。比起我们所尝试过的真正的等宽字体，我们更喜欢这种安排。这就是为什么在有些示例中，列没有正好对齐的原因。

响理解即可。例如，我们经常会提到的一些词条，比如叫做 daemon 的 Linux 组和打印机 anchor-lw 等，根本不采用特殊格式。

一般说来，我们使用的约定与联机用户手册中用来表示命令的语法相同：

- 中括号（“[”和“]”）之间的任何内容都是可选的；
- 英文省略号（“...”）后面的任何内容都是可以重复的；
- 大括号（“{”和“}”）表示应该选择由竖线（“|”）隔开的各项中的一个。

例如，规则：

```
bork [-x] {on | off } filename ...
```

可以匹配下面任何一条命令：

```
bork on /etc/passwd  
bork -x off /etc/passwd /etc/termcap  
bork off /usr/lib/tmac
```

我们将 shell 风格的特殊字符用于模式匹配：

- 星号（*）匹配零个或多个字符；
- 问号（?）匹配一个字符；
- 中波浪线（~）表示当前用户的主目录（home）⁷；
- ~user 表示 user 用户的主目录。

例如，我们会采用速记模式/etc/rc*.d 来指代启动脚本目录/etc/rc0.d、/etc/rc1.d 等。

引号中的文本经常具有精确的技术含义。在这些情况下，我们忽略美式英语的一般规则，把标点符号放在引号外边，以使读者不会对在引号中包括什么不包括什么产生混淆。

⁷. Solaris 10 给 root 的默认 shell 是最初的 Bourne shell，它居然都不能识别~或者~user（真令人惊异）。

1.8 单位

像 kilo- (千)、 mega- (兆) 和 giga- (吉) 这样的公制前缀都定义为 10 的幂：一百万是 1,000,000。但是，计算机的类型定义长期以来一直借用这些前缀，但却用它们代表 2 的幂。例如，一兆内存实际是 2^{20} ，即 1,048,576 字节。这种借用的单位甚至混入了正式标准，像 JEDEC 固态技术协会的标准 100B.01，该标准认定这些前缀表示 2 的幂（虽然有些质疑）。

为了恢复清楚的含义，国际电工委员会（International Electrotechnical Commission，IEC）规定了一组数字前缀（分别是 kibi-、mebi-、gibi- 等，缩写为 Ki、Mi 和 Gi），明确基于 2 的幂。这类单位含义总是清楚的，但它们才开始得到广泛使用。原来的 kilo- 系列前缀则两种含义都在用。

通过上下文关系才能帮助判断到底按哪一种含义算。RAM 总是按 2 的幂来算，而网络带宽一定按 10 的幂来算。存储空间一般按 10 的幂为单位算，但是块和页的大小则用 2 的幂计算。

我们在书中以 2 的幂计算 IEC 的单位，用 10 的幂计算公制单位，对粗略值以及确切的底数不清楚、没有文档或者不可能确定的情况用公制。在命令输出里，以及配置文件节选中我们都保留原本的值和单位记法。我们把 bit (位) 缩写为 b，而把 byte (字节) 缩写为 B。表 1.2 给出了一些例子。

表 1.2

单位释义举例

示例	含义
56kb/s 串行线	能够每秒传输 56,000 位的串行线
1kB 文件	包含 1,000 字节的文件

<u>4KiB SSD 页</u>	包含 4,096 字节的 SSD 页
<u>8KB 内存</u>	本书不采用这种说法；参见下面的注释
<u>100MB 文件大小限制</u>	名义上指 10^8 字节；放在上下文里看，会有二义性
<u>100MB 磁盘分区</u>	名义上指 10^8 字节；放在上下文里看，可能是指 99,999,744 字节 ^a
<u>1GiB 内存</u>	准确地说是 1,073,741,824 字节内存 ^b
<u>1Gb/s 以太网</u>	能够每秒传输 1,000,000,000 位数据的网络
<u>1TB 硬盘</u>	存储 1,000,000,000,000 字节的硬盘

^a. 也就是说，硬盘块大小为 512 字节，数倍之后最接近 10^8 的数值

^b. 但是根据微软的说法，仍然是没有足够的内存运行 64 位版的 Windows 7

在“8KB 内存！”中，缩写 K 不属于任何标准。它是一个计算机行话，指公制缩写 k（代表 kilo-，千），后者起初表示 1,024 而不是 1,000。但是即便更大一些公制前缀的缩写都已经变成大写[译者注：M、G、T]，对于 k 却不能以此类推也用 K。后来，人们开始混淆这种区别，开始也用 K 代表 1,000。

Ubuntu 的 Linux 发行版本做了大胆尝试，以求在这个问题上保持合理性和一致性；参考 wiki.ubuntu.com/UnitsPolicy 了解更多的细节。

1.9 手册页和其他联机文档

手册页通常称为“man 手册页”，因为要用 man 命令来阅读，它构成了传统的“联机”文档（当然，现如今所有的文档都以这样或者那样的形式在线存在）。man 手册页一般伴随系统一起安装。针对特定程序的 man 手册页通常随着安装新软件包一同安装。

man 手册页对单条命令、驱动程序、文件格式或者库例程给出简洁的说明。它们不会解释

诸如“我该怎样安装一个新设备？”或者“为什么我们的系统这么慢？”这样更普通的话题。对于这些问题来说，用户可以参考厂商提供的系统管理指南（参见 1.10 节），对于 Linux 系统来说，也可以从“Linux 建档项目（Linux Documentation Project）”中获得文档。

1.9.1 手册页的组织

所有的系统都把 **man** 手册页分成若干节，但是在各个节的规定上面，不同系统间略有区别。我们的示例系统所采用的基本结构如表 1.3 所示。

表 1.3 **man** 手册页的各节及其内容

Linux	Solaris	HP-UX	AIX	内 容
1	1	1	1	用户级命令和应用程序
2	2	2	2	系统调用和内核出错代码
3	3	3	3	库调用
4	7	7	4	设备驱动程序和网络协议
5	4	4	5	标准文件格式
6	6	-	6	游戏和演示
7	5	5	7	各种文件和文档
8	1m	1m	8	系统管理命令
9	9	-	-	含糊的内核规范和接口
-	-	9	-	HP-UX 的一般信息

手册页中有些节会做进一步细分。例如，Solaris 的第 3c 节包含了有关系统 C 标准库的手

册页；有些系统的第 8 节没有内容，而把系统管理命令放到第 1 节。许多系统撤销了游戏和演示，第 6 节什么内容都没有。很多系统在手册中都有一个称之为 l (L 的小写) 的节，用于本地手册页。

对于大多数主题而言，各节的确切结构并不重要，因为只要有匹配的手册页，**man** 命令就会找到它。用户只需要对出现在不同节内同名的主题，知道各节的规定就行了。例如，**passwd** 既是一条命令，也是一个配置文件，所以在第 1 节和第 4 或第 5 节都有关于它的手册项。

1.9.2 man : 读取手册页

man *title* 命令格式化特定的手册页并通过 **more**、**less** 命令，或者在 PAGER 环境变量中指定的任何程序把手册页发送到用户终端。*title* 通常是一个命令、设备、文件名或者库例程名。手册中的各节大致是按照数字顺序进行搜索的，不过通常首先搜索描述命令的那些节（第 1、8 和 6 节）。

命令格式 **man** *section title* 可让用户从某个特定的节获取手册页。于是在大多数系统上，**man sync** 命令可得到 **sync** 命令的手册页，**man 2sync** 命令可得到系统调用 **sync** 的手册页。

在 Solaris 上，必须在节号前面加上 -s 标志；例如，**man -s 2 sync**。

man -k *keyword* 或者 **apropos** *keyword* 输出一份手册页清单，在其单行概要中有

keyword。例如：

```
$ man -k translate
objcopy (1)      - copy and translate object files
dcgettext (3)    - translate message
tr (1)           - translate or delete characters
snmptranslate (1) - translate SNMP OID values into more useful information
tr (1p)          - translate characters
gettext (1)      - translate message
ngettext (1)     - translate message and choose plural form
...
```

keyword 指代的库可能已经过期了。如果要向系统添加 **man** 手册页，需重构这个库文件，在 Ubuntu、SUSE 上使用 **mandb** 命令，在 Red Hat 上使用 **makewhatis** 命令，在 Solaris、HP-UX 和 AIX 上使用 **catman -w** 命令。

1.9.3 手册页的保存

手册页的 **nroff** 输入通常保存在 **/usr/share/man** 下的多个目录中。Linux 系统会把它们用 **gzip** 压缩以节省空间（命令 **man** 知道如何当场将它们解压缩）。如果在 **/var/cache/man** 或者 **/var/share/man** 下的适当目录有写入权限，那么命令 **man** 会在那里维护一个有格式的手册页的缓存，但是这存在安全风险。大多数系统会在安装的时候预处理一次手册页的格式（参考 **catman** 命令），或者根本不做预处理。

除了传统的 **nroff** 格式之外，Solaris 还支持 SGML 格式的手册页 SGML 格式手册页相关各节的目录都在 **/usr/share/man** 下。

命令 **man** 会在若干不同的目录中寻找用户需要的手册页。在 Linux 系统上，用户可以用 **manpath** 命令来确定搜索路径。这个搜索路径（在 Ubuntu 中）一般为：

```
ubuntu$ manpath  
/usr/local/man:/usr/local/share/man:/usr/share/man
```

如果有必要，用户可以设置自己的环境变量 **MANPATH** 来覆盖默认路径：

```
export MANPATH=/home/share/localman:/usr/share/man
```

有些系统能让用户为手册页设置一个自定义的系统级默认搜索路径，如果用户想要维护平行的一套 **man** 手册页目录树（比如 OpenPKG 创建的目录树），就能够用上这种功能。不过，如果用户想要用手册页的形式发布本地文档，更简单的做法是，用系统的标准打包机制打包，并把这些手册页放在标准的 **man** 目录下。参考第 12 章了解有关软件安装和管

理的更多知识。

1.9.4 GNUTexinfo

Linux 系统包括一种补充的联机手册页系统，叫做 Texinfo。很久以前，GNU 人士针对设定 man 页面格式的 **nroff** 为 AT&T 专有命令这一现实情况，发明了 Texinfo 文档。现在我们已经有了 GNU 的 **groff** 来为我们完成这项工作，**nroff** 的问题不再重要了，但它仍然在人们脑海里阴魂不散。

虽然 Texinfo 逐渐不用了，但仍有很多几个 GNU 软件包坚持用 Texinfo 文件而不是 man 页面来提供自身的文档。用户可以把读取 Texinfo 的命令 **info** 的输出通过管道送给 **less** 命令，避开 **info** 命令内建的浏览体系。幸运的是，采用 Texinfo 提供文档的软件包通常会安装指示性的 man 页面，告诉用户使用 **info** 命令来阅读了解那些特殊的软件包。用户可以坚持使用 **man** 命令来查找手册，只有在被告知要采用 **info** 时再这么做，这样的做法很保险。命令 **info info** 将会把用户带入 Texinfo 的黑暗神秘世界。

1.10 其他的权威文档

手册页仅仅是官方文档中的很小一部分。遗憾的是，其余更大一部分的文档都散布在 Web 上。

1.10.1 针对系统的专门指南

大多数发行商都有自己专门的文档项目，许多发行商还出整本书那样的手册。现在，一般都能找到联机形式的手册，而不是纸质的书。文档的规模和质量则大有不同，但是大多数发行商都至少提供一份系统管理指南和一份安装指南。表 1.4 给出了在哪儿可以找到我们示例系统的文档。

在这其间最出众的是 IBM，IBM 针对系统管理的各方面出版了大量篇幅完整的图书。读者可以买书，也可以直接免费下载。IBM 的文档完整也有不好的一面，就是许多文档似乎都要比当前 AIX 的发布落后一两个版本号。

表 1.4 到哪儿找操作系统发行商的专门文档

系 红	URL	说 明
Ubuntu	help.ubuntu.com	绝大多数都面向用户；参考“sever guide（服务器指南）”
SUSE	novell.com/documentation	系统管理的材料都在“reference guide（参考指南）”
RHEL	redhat.com/docs	绝大多数文档都是介绍 Red Hat 的扩展
Solaris	docs.sun.com	材料的编目很广
HP-UX	docs.hp.com	书、白皮书和技术指南
AIX	www.redbooks.ibm.com ibm.com/support	说明、FAQ 等支持材料的入口

在文档方面的竞赛中，Red Hat 遗憾地落后了。Red Hat 的大多数文

档都和其专有的有附加值的系统有关系，而不是介绍通用的 Linux 管理技术。

1.10.2 针对软件包的专门文档

UNIX 和 Linux 世界里重要的软件包大多数是由个人或者第三方来维护的，比如 Internet 软件联盟（Internet Software Consortium）和 Apache 软件基金会（Apache Software Foundation）。这些组织一般为它们所发布的软件包编写文档。文档质量则从令人尴尬到叹为观止都有，但因为有了 svnbook.red-bean.com 的“*Version Control with Subversion*”这样的珍品存在，所以还是值得花时间去找的。

UNIX 厂商和 Linux 发行商总会在其软件包里带上 man 手册页。遗憾的是，他们对其他文档则显得不够用心，这大多因为确实没有一个标准位置保存文档（查看 **/usr/share/doc**）。查看软件的原出处，看看有没有更多的材料，常常比较有用。

补充材料包括白皮书（技术报告）、设计规范和针对专题的书或者小册子。这些补充材料不仅限于介绍一条命令，所以它们可以采用教程或者规程的形式。许多软件既有 man 手册页，也有介绍文章。例如，**vi** 的手册只告诉用户有关 **vi** 能够支持的命令行参数，而用户要深入学习才能掌握如何真正编辑一个文件。

1.10.3 书籍

在打印材料中，给系统管理员的最好资料来源就是 O'Reilly 的系列图书（除了本书之外）。这个系列从 20 年前的“*UNIX in a Nutshell*（UNIX 简明教程）”开始，到如今针对每个 UNIX 和 Linux 子系统和命令都已经有一本独立成册的书。这个系列的图书也包括介绍

Internet、Windows 和其他 UNIX 之外的专题。所有这些书都定价合理、内容及时而且针对性强。

Tim O'Reilly 已经开始表现出对开源运动具有很大兴趣，他为开源举办了一个名为 OSCON 的大会，而且也为其他流行的技术主题举办大会。OSCON 每年举办两次，一次在美国，另一次在欧洲。参考 oreilly.com 了解更多情况。

1.10.4 RFC 和其他 Internet 文档

RFC (Request for Comments , 请求注释) 的系列文档描述了 Internet 上使用的协议和规程。这些文档大多相当详尽而且技术性强，但是有些 RFC 只是作为概述来写。它们都绝对有权威性，许多 RFC 对系统管理员相当有用。参考 14.1.2 节了解这些文档的更完整说明。

1.10.5 LDP

Linux 系统还有另一种主要的参考资源：Linux 建档项目 (Linux Documentation Project , LDP)，地址在 tldp.org。这个网站保存了大量用户贡献的文档，内容从 FAQ 到完整篇幅的指南不一而足。LDP 还集中力量把与 Linux 相关的文档都翻译成其他语言。遗憾的是，许多 LDP 文档现在并没有得到很好维护。因为 Linux 的时间概念相对于真实的时间来说要短得多，没人管的文档容易很快过时。一定要查看 HOWTO 或者指南上的时间戳，并据此估计它的可信度。

1.11 其他的信息资源

前面几小节讨论的资源一般都最可靠，但它们极少成为 UNIX 和 Linux 世界里仅有的文档。Internet 充满了不计其数的博客、论坛以及新种子。但是，毋庸置疑，谷歌是系统管理员的最好朋友。除非正在查的是一条特殊命令或者文件

格式的细节，否则谷歌应该是咨询任何系统管理问题的首选资源。如果还没有的话，就养成查谷歌的习惯，这样就可以避免在线论坛上提问题，却被人报以一个到谷歌的链接做回答，结果又耽误时间又丢面子的情况发生⁸。万事不决问谷歌吧。

我们无法一一列举 Internet 上的每种 UNIX 和 Linux 信息源，但是表 1.5 里给出了其中最重要的一些。

另一个既有意思又有用的资源是 Bruce Hamilton 在 bhami.com/rosetta.html 上提供的“Rosetta Stone (罗塞塔石碑)”。其中包含了很多链接，指向在许多不同的操作系统上执行各种系统管理任务所用到的命令和工具。

如果是 Linux 站点，不要羞于访问普通的 UNIX 资源。大多数信息都能直接用于 Linux。

表 1.5 Web 上的系统管理资源

Web 站点	说 明
blogs.sun.com	技术文章的宝库；许多和 Solaris 有关
cpan.org	权威的 Perl 模块库
freshmeat.net	Linux 和 UNIX 软件的海量索引库
kernel.org	Linux 内核的官方网站
linux.com	Linux 论坛，适合新用户 ^a
linux.org	Linux 一般信息交换地
linux.slashdot.org	技术新闻巨头 Slashdot 针对 Linux 的网站

续表

Web 站点	说 明
linuxhq.com	有关内核的信息和补丁的汇编

⁸ 或者更糟的话，用一条通过 lmgtfy.com 的链接连谷歌。

lwn.net	Linux 和开放源代码方面的通讯社
lxer.com	Linux 新闻集散地
rootvg.net	面向 AIX 的站点，有许多链接和好论坛
securityfocus.com	计算机安全方面的一般信息
serverfault.com	针对系统管理问题而分工协作编辑的数据库，
ServerFiles.com	网络管理软件和硬件的目录
slashdot.org	各类技术新闻
solariscentral.org	有 Solaris 相关新闻和文章的开放博客
sun.com/bigadmin	专门针对 Sun 系统管理知识的集散地
sunhelp.org	有关 Sun 的很不错的材料集中库
ugu.com	UNIX Guru Universe (UNIX 高手大世界) ——所有内容都和系统管理有关

a. 这个站点现在由 Linux Foundation (Linux 基金会) 负责运行

1.12 查找和安装软件的途径

第 12 章详细讲述了软件方面的知识。但是对于没有耐性的读者来说，这一节是个初级速成教材，教给读者怎样知道自己系统上已经安装好的软件有哪些，以及怎样获得和安装新软件。

现代操作系统把自己分成多个软件包，软件包可以逐个独立安装。默认安装的只包括一定范围的起步软件包，用户可以根据需要再扩充。

附加软件常常也以预先编译好的软件包形式提供，这是一种主流的方式，各个系统间的区

别仅仅在于程度的不同。大多数软件都是由独立的小组开发，且以源代码的形式发布出来。接下来，软件库收集这些源代码，然后根据软件库为之服务的系统上的使用习惯，对其进行编译，再把编译结果打成软件包。安装针对特定系统的二进制软件包通常要比取得并编译原来的源代码更容易。不过，打包软件有时候要比当前版本落后一两个版本号。

两种系统使用相同的软件打包系统不一定意味着两个系统的软件包能够互换使用。例如，Red Hat 和 SUSE 都使用 RPM，但是它们的文件系统布局却稍有不同。如果能找到专为自己系统做的软件包，那就用它们，这是最好的做法。

主流的 Linux 发行版本都提供优异的软件包管理体系，里面包括能够访问和搜索 Internet 上软件库的工具。发行商替社区积极维护这些软件库，所以 Linux 系统管理员很少需要跳出自己系统上的默认软件包管理器。生活真美好。

UNIX 系统在软件包管理方面显得畏手畏尾。Solaris、HP-UX 和 AIX 都提供打包软件，在单机的层次上使用没问题。但是，这些系统的厂商却没有维护开源软件库，而让用户群大多进行自我维护⁹。遗憾的是，维系一个打包软件环境所依赖的纽带之一，是有一条途径可以让软件包可靠地检索其他软件包，从而掌握依赖性和兼容性方面的信息。没有某种中心作为协调，整个体系就会很快分崩离析。

在现实世界里，情形则各不相同。Solaris 有一套附加系统（blastwave.org 的 **pkgutil**），可以方便地从 Internet 软件库安装软件，这和 Linux 发行版本上自带的系统很像。HP-UX 有一个不错的 Internet 软件库，它采用了 HP-UX 移植和存档中心（HP-UX Porting and Archiving Centre）的形式，地址为 hpxx.connect.org.uk，但是必须手工逐个下载软件包。在这个领域内做得较差的典型，就是 AIX，给它预先打包好的软件找起来很分散。

⁹ . OpenSolaris 确实提供了质量达到 Linux 水平的软件包管理系统和 Internet 软件库。Solaris 10 没有这项功能，但是可能在 Solaris 11 上会有。

系统管理员没找到打好包的二进制软件，就必须采用老办法安装软件：下载一个 tar 源代码包，手工配置、编译和安装它。这个过程可长可短，取决于具体软件和操作系统。

在本书里，我们一般认为可选软件都已经装好了，并不折磨读者按生搬硬套的步骤去安装每个软件包。如果有可能发生混淆，我们有时候也会给出完成某个特定项目所需软件包的确切名称。但是，对于大部分内容来说，我们都不会重复讲述安装步骤，因为前后软件包的安装都很类似。

1.12.1 判断软件是否已经安装

出于各种原因，要判断实际需要的东西在哪个软件包里，需要有点儿小技巧。除了在软件包的级别上查找之外，更简单的做法是使用 shell 的 **which** 命令找出相关的二进制程序是否已经在搜索路径中。例如，下面的命令就揭示出已经在机器上安装了 GNU 的 C 编译器。

```
aix$ which gcc  
/opt/pware/bin/gcc
```

如果 **which** 没有找到要找的命令，那么试试 **whereis**；该命令搜索更大范围的系统目录，与 shell 的搜索路径无关。

另一种方法是采取作用非凡的 **locate** 命令，它参照预先编译好的一个文件系统索引，以此确定与特定模式相匹配的文件名。**locate** 命令是 GNU 的 findutils 软件包里的一个程序，这个软件包在大多数 Linux 系统上都是默认包含的，但在 UNIX 系统上必须手工安装。**locate** 的搜索并不只针对命令或者软件包，而是能够找到任何类型的文件。例如，如果读者不能确定在哪儿能找到头文件 **signal.h**，可以试试：

```
ubuntu$ locate signal.h
```

```
/usr/include/signal.h  
/usr/include/asm/signal.h  
/usr/include/asm-generic/signal.h  
/usr/include/linux/signal.h
```

locate 的数据库通常由 **updatedb** 命令定期更新，这条命令由 **cron** 来运行。因此，执行一次 **locate** 的结果不是总能够反映出文件系统新近的变化。

如果用户知道正在查找的软件包的名字，那么也可以使用系统上的软件包工具来直接检查是否有该软件包出现（以及已装软件的版本）。例如，在 Red Hat 或者 SUSE 系统上，下面的命令可以检查是否有 Python 脚本语言：

```
redhat$ rpm -q python  
python-2.4.3-21.el5
```

参考第 12 章，了解有关软件包管理的更多知识。

1.12.2 增加新软件

如果需要安装额外的软件，首先要确定相关软件包的标准名称。例如，需要把“我想装 **locate**”转换为“我需要安装 **findutils** 软件包”，或者把“我要 **named**”转换为“我必须安装 **BIND**”。在网上各种针对特定系统的索引能够帮助做转换，但是 Google 通常更有效。例如，搜索“**locate** 命令”，就能直接找到若干与之相关的讨论。如果是在 UNIX 上，那么还需要输入操作系统的名字。

一旦知道了相关软件的确切名称，就可以下载并安装它。在 Linux 和安装了 **pkgutil** 的 Solaris 系统上，整个安装过程通常用一条命令就够了。对于 HP-UX 和 AIX 而言，则要么下载预编译好的二进制软件包，要么下载项目的源代码。如果是后者，需要通过 Google

定位该项目的正式网站，然后从项目的镜像站点之一下载源代码。

下面的例子展示了在我们的每一种示例系统上安装 **wget** 命令。它是一个很棒的 GNU 工具，能够把 HTTP 和 FTP 下载变成单条命令——对于脚本编程来说非常有用。我们所举的每种 Linux 系统默认都安装了 **wget**，但是下面给出的命令用于初始化安装和后续升级。

Ubuntu 使用 APT (Debian Advanced Package Tool , Debian 高级软件包工具) :

```
ubuntu# apt-get install wget
Reading package lists... Done
Building dependency tree
Reading state information... Done
wget is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 0 not
upgraded.
```



SUSE 版的操作：

```
suse# yast --install wget
```

<在一个基于终端的 UI 里运行>

Red Hat 版的操作：

```
redhat# yum install wget
Loaded plugins: fastestmirror
...
Parsing package install arguments
Package wget-1.10.2-7.el5.i386 is already installed and
latest version
Nothing to do
```

在已经装有 **pkgutil** 的 Solaris 系统上 (参考 blastwave.org 来了解配置

说明) :

```
solaris# /opt/csw/bin/pkgutil --install wget
```

<安装了 7 个软件包产生的多页输出信息>

对于 HP-UX , 我们在 hpxx.connect.org.uk 上找到了一个适用的二进制软

件包 , 把它下载到/tmp 目录下。解压缩和安装的命令如下 :

```
hpxx# gunzip /tmp/wget-1.11.4-hppa-11.31.depot.gz
hpxx# swinstall -s /tmp/wget-1.11.4-hppa-11.31.depot wget
=====
      05/27/09 13:01:31 EDT    BEGIN swinstall SESSION
(non-interactive) (jobid=hpxx11-0030)

* Session started for user "root@hpxx11".

* Beginning Selection
* Target connection succeeded for "hpxx11:/".
* Source: /tmp/wget-1.11.4-hppa-11.31.depot
* Targets: hpxx11:/
* Software selections:
wget.wget-RUN,r=1.11.4,a=HP-UX_B./800
* Selection succeeded.
* Beginning Analysis and Execution
...
* Analysis and Execution succeeded.
```

在 **swinstall** 命令行里出现的 depot 软件包必须用/开头的完整路径 ; 否则 ,

swinstall 就会到网络上找文件。最后的 **wget** 告诉 **swinstall** 从 depot

文件里安装哪个软件包。

遗憾的是 , 这个安装过程没有乍看上去那么简单。装好的 **wget** 版本实际

上无法运行 , 因为没有装它所依赖的几个库。

```
hpxx$ wget http://samba.org/samba/docs/Samba3-HOWTO.pdf
/usr/lib/dld.sl:      Can't open shared library:
/usr/local/lib/ libcrypto.sl
/usr/lib/dld.sl: No such file or directory
```

```
[HP ARIES32]: Core file for 32 bit PA-RISC application  
[HP ARIES32]: /usr/local/bin/wget saved to /tmp/core.wget.
```

swinstall 内置有依赖关系的管理机制，但是遗憾的是，它的这种功能不能延伸到 Internet 上的软件库里。用户不得不根据提示安装所有要求预装的软件包（本例中有 6 个之多）以达到最终目标。

1.12.3 从源代码编译软件

实际上，**wget** 至少存在一种可用于 AIX 的二进制软件包，它是 RPM 格式的。用 Google 搜索“aix wget rpm”应该就能找到一些不错的线索。下载下来之后，安装命令很简单

```
aix# rpm --install wget-1.11.4-1.aix5.1.ppc.rpm
```

但为了演示，我们从源代码开始编译 AIX 版本的 **wget**。

我们的第一项任务是找到代码，这很容易：用 Google 搜索“wget”得到的第一个结果就正好把我们指向 GNU 的 **wget** 项目，再点一下就能找到源代码的 tar 包。把当前版本的代码下载到/tmp 目录里，再解开、配置和安装：

```
aix# cd /tmp; gunzip wget-1.11.4.tar.gz  
aix# tar xfp wget-1.11.4.tar  
aix# cd wget-1.11.4  
  
aix# ./configure --disable-ssl --disable-nls # 参考下面的注释  
  
configure: configuring for GNU Wget 1.11.4  
checking build system type... rs6000-ibm-aix  
...  
config.status: creating src/config.h config.status: executing default  
commands generating po/POTFILES from ./po/POTFILES.in creating po/Makefile  
aix# make
```

<几页的编译输出>

```
aix# make install
```

<大约有一页的输出>

对于大多数 UNIX 和 Linux 软件来说，都可以依次执行 **configure/make/make**

install 三条命令，只要已经安装了开发环境以及预先要求有的软件包，就会得到结果。

不过，最好查看软件包的 **INSTALL** 或者 **README** 文件，了解该软件的特别之处。

在这个例子里，**configure** 命令的两个选项 **--disable-ssl** 和 **--disable-nls** 省略了

wget 的一些特性，这些特性要依赖其他一些还没有安装的库。在实际中，用户可能想还

是安装要求预装的那些库。使用 **configure--help** 查看所有的配置选项。另一个有用的

配置选项是 **--prefix=directory**，这个选项可以把软件安装到 **/usr/local** 之外的其他目

录里。

1.13 重压下的系统管理员

系统管理员通常有多种身份。在现实生活中，系统管理员经常从事其他工作，但同时又被请来管理身边的一些计算机。如果您是处于这种情形之下，那么也许想要考虑一下长期兼职最终可能出现的情况。

您的系统管理经验越丰富，那么用户群体对您的依赖也就越大。网络规模总是在增长，随着管理系统变得越发复杂，管理工作花费的时间也就会越多。很快，您就会发现在您所在的机构中，您成了唯一懂得怎样去执行各种重要任务的人。

一旦同事们开始把您看作是本地的系统管理员，那时要把自己从这种角色中解救出来就很困难了。那虽说不一定是件坏事，但我们就知道有几个人为了逃避这种状况已经换了工作。由于许多管理任务是无形中的，您还可能发现自己所在的单位希望您既成为一名全职的系

统管理员，同时也成为全职的工程师、作家或分析员。

不乐意这样做的系统管理员，常常倾向于通过态度不好和服务差劲儿来摆脱这类要求¹⁰。

这种方法往往事与愿违，不但有损自己的形象，并且可能产生更多问题。

相反，可以考虑把花费在系统管理上的时间详细记录下来。您的目标应该是保持其工作处于可以管理的水平，而且应该搜集让您在申请减轻管理任务时用得上的证据。在大多数单位里，您需要游说管理层半年到一年的时间以后，才可能让自己的管理员位置由别人来取代，因此应该提前计划好。另一方面，您可能发现自己喜欢系统管理员的工作，而且喜欢程度超过了本职工作。好处是职业前景保持良好。不好之处是，您的政治问题可能会激化参考第 32 章，事先做好准备。

1.14 推荐读物

ROBBINS, ARNOLD. UNIX in a Nutshell (4th Edition). Sebastopol, CA: O'Reilly Media, 2008.

SIEVER, ELLEN, AARON WEBER, AND STEPHEN FIGGINS. Linux in a Nutshell (5th Edition). Sebastopol, CA: O'Reilly Media, 2006.

GANCARZ, MIKE. Linux and the Unix Philosophy. Boston: Digital Press, 2003.

SALUS, PETER H. The Daemon, the GNU & the Penguin: How Free and Open Software is Changing the World. Reed Media Services, 2008.

在 groklaw.com 上，可以找到由最知名的 UNIX 历史学家写的开源运动的多彩多姿的发展史，这本书采用 Creative Commons 许可证。它自己的 URL 链接相当长；到 groklaw.com 上找当前的链接，或者试试这个等价的压缩链接：tinyurl.com/d6u7j。

RAYMOND, ERIC S. The Cathedral & The Bazaar: Musings on Linux and OpenSource by an Accidental Revolutionary. Sebastopol, CA: O'Reilly Media, 2001.

10 . 在 Simon Travaglia 的“Bastard Operator from Hell”故事里写出了这种既可笑又可气的态度；到 bofh.ntk.net 上找来读读这些档案材料吧（在 BOFH 下面找）。

系统管理

LIMONCELLI, THOMAS A., CHRISTINA J. HOGAN, AND STRATA R. CHALUP. *The Practice of System and Network Administration* (Second Edition). Reading, MA: Addison-Wesley, 2008.

这是一本好书，它的强项在于讲述系统管理的政策和规程方面的内容。作者们在 everythingsysadmin.com 上维护着一个系统管理方面的博客。

FRISCH, ÆLEEN. *Essential System Administration* (3rd Edition). Sebastopol, CA: O'Reilly Media, 2002.

这是一本 UNIX 系统管理的经典指南，但遗憾的是内容有点儿过时。我们希望有出新版的规划！

必备工具

ROBBINS, ARNOLD, ELBERT HANNAH, AND LINDA LAMB. *Learning the vi and Vim Editors*. Sebastopol, CA: O'Reilly Media, 2008.

POWERS, SHELLY, JERRY PEEK, TIM O'REILLY, AND MIKE LOUKIDES. *UNIX PowerTools* (3rd Edition). Sebastopol, CA: O'Reilly Media, 2003.

MICHAEL, RANDAL K. *Mastering UNIX Shell Scripting: BASH, Bourne, and Korn Shell Scripting for Programmers, System Administrators, and UNIX Gurus*. Indianapolis, IN: Wiley Publishing, Inc., 2008.

ROBBINS, ARNOLD AND NELSON H. F. BEEBE. *Classic Shell Scripting*. Sebastopol, CA: O'Reilly Media, 2005.

WALL, LARRY, TOM CHRISTIANSEN, AND JON ORWANT. *Programming Perl* (3rd Edition). Cambridge, MA: O'Reilly Media, 2000.

CHRISTIANSEN, TOM, AND NATHAN TORKINGTON. *Perl Cookbook* (2nd Edition). Sebastopol, CA: O'Reilly Media, 2003.

BLANK-EDELMAN, DAVID N. *Automating System Administration with Perl* (2nd Edition). Sebastopol, CA: O'Reilly Media, 2009.

PILGRIM, MARK. *Dive Into Python*. Berkeley, CA: Apress, 2004.

这本书也可以从 diveintopython.org 这个网站获得。

FLANAGAN, DAVID, AND YUKIHIRO MATSUMOTO. *The Ruby Programming Language*. Sebastopol, CA: O'Reilly Media, 2008.

这本书虽然乐观地冠以副标题“*Everything You Need to Know*”，但遗憾的是，这方面的内容不多。不过，该书涵盖了 Ruby 1.9 版的内容，包括了很多只有该语言的设计者才有资格知道的丰富细节。最适合已经有 Perl 或者 Python 工作经验的人。

1.15 习题

E1.1 应该使用什么命令读终端驱动程序 **tty** (不是 **tty** 命令) ? 应该怎样阅读保

存在 /usr/local/share/man 中的 **tty** 本地手册页 ?

E1.2 在您的站点上 , 有一个系统级的配置文件来控制 **man** 的行为吗 ? 如果想要

在 /doc/man 中保存本地的文档 , 应该在这个文件中加入哪些行 ? 要

在 /doc/man 中使用怎样的目录结构才能让它成为 man 手册页层次结构

中的一部分 ?

★E1.3 Linux 内核的当前发展状态如何 ? 热点问题是什么 ? 关键人物都有谁 ? 内核

开发的项目是如何管理的 ?

★E1.4 研究几种 UNIX 和 Linux 系统 , 为下列每个应用推荐一种系统 , 解释做出选

择的理由。

a) 在家办公的单个用户。

b) 大学的计算机科学实验室。

c) 企业的 Web 服务器。

d) 为一家航运公司运行数据库的服务器集群。

★E1.5 假定您发现了在 Ubuntu 提供的文档中说明的 Apache httpd 的某种功能并不起

作用。

a) 在报告这个 bug 之前 , 您应该做什么 ?

b) 如果认定这确实是一个 bug , 那么您应该通知谁 ? 怎样通知 ?

c) 要让 bug 报告有用处的话 , 它必须包含什么信息 ?

★★E1.6 Linux 已经在生产环境中取得广泛应用了吗？UNIX 要不行了吗？为什么会有这样的结论？

第2章 脚本和 shell



好的系统管理员都要写脚本。脚本以标准和自动的方式履行系统管理员的

繁杂事务，藉此把管理员的时间节省出来，以花在更重要和更有意思的任务上。

从某种意义上讲，脚本也是一种低质量的文档，因为它们充当了一种权威提纲，提纲里列出完成特殊任务所需的步骤。

从复杂性来看，系统管理脚本的范围很广，小到一个脚本，简单得只封装几条静态命令，大到一个重要的软件项目，为整个站点管理主机配置和管理性数据。在本书里，我们所感兴趣主要是系统管理员通常会碰到的较小的日常脚本项目。因此，对于较大项目才需要的支持功能（例如，bug 追踪和设计评审），我们不会讲得太多。

系统管理脚本应该注重两点，即编程人员的开发效率和代码的清晰可读性。计算效率不应该成为关注重点，但这不应成为草率行事的借口，而是要认识到，很少需要在意一个脚本是在半秒还是两秒内运行完。优化脚本获得的回报都非常低，甚至对通过 **cron** 定期运行的脚本来说也不例外。

长期以来，编写系统管理脚本的标准语言都是 shell 所定义的语言。在大多数系统上，默认的 shell 都是 **bash**（即“Bourne again”shell），但是在几种不多的 UNIX 系统上，也用 **sh**（最初的 Bourne shell）和 **ksh**（Korn shell）。shell 脚本一般用于轻量级的任务，如自动执行一系列命令，或者把几个过滤器组合起来处理数据。

各种操作系统上都有 shell，所以 shell 脚本的可移植性相当好，除了它们调用的命令之外，要依赖的东西也不多。无论是否选择 shell 来编写脚本，都会碰到 shell：大多数环境都包括对已有 **sh** 脚本的强大补充，系统管理员会频频阅读、理解和调整这些脚本。

对于更为复杂高端的脚本来说，建议转而采用一种真正的编程语言来写，像 Perl 或者 Python 这样的语言，它们两者都很适合于系统管理工作。这两种语言融入的设计理念比 shell 领先 20 年，它们的字处理功能（对于系统管理员来说，价值难以估量）如此强大，**sh** 在它们面前黯然失色。Perl 和 Python 的主要缺点在于，建立它们的环境要麻烦一点儿，尤其是要用到的第三方库，而库里又包含已经编译好的部件的时候。shell 没有模块结构，也没有第三方的库，因此避开了这个特殊的问题。

本章简要介绍了 **bash**、Perl 和 Python 作为脚本编程语言的用法，以及正则表达式这种通用的技术。

2.1 shell 的基础知识

在我们开始介绍 shell 的脚本编程之前，让我们先看看 shell 的一些基本特性和语法。不管读者正在使用的是何种平台，本节都适用于 **sh** 大家庭里的所有主流 shell（包括 **bash** 和 **ksh**，但不包括 **csh** 或者 **tcsh**）。尝试一下自己不熟悉的 **sh** 形式，做做实验吧！

2.1.1 编辑命令

我们已经注意到一点，太多人都用方向键来编辑命令行。但读者朋友不会在文字编辑器里这么做，对吗？

如果喜欢 **emacs**，那么在编辑命令历史的时候，所有的 **emacs** 基本命令都用得上。用 <Control-E> 到行尾，用 <Control-A> 到命令行的开头。用 <Control-P> 一条条回退到

最近执行过的命令，重新把它们调出来进行编辑。用 <Control-R> 增量搜索命令历史找出老命令。

如果喜欢 **vi**，那么用下面的命令就能让 shell 的命令行编辑进入 **vi** 模式：

```
$ set -o vi
```

和在 **vi** 里一样，编辑操作是有模式的；不过，一开始会进入输入模式。按 <Esc> 键离开输入模式，按 “i” 键重新进入输入模式。在编辑模式下，“w” 键向前进一个单词，“fX” 在本行里找到下一个 X，等等。用 <Esc> k [译者注：即同时按下 <Esc> 键和 k 键] 可以遍历过去输入的命令。想要再次回到 **emacs** 编辑模式吗，可使用下面的命令实现

```
$ set -o emacs
```

2.1.2 管道和重定向

每个进程都至少有 3 个信道：“标准输入” (STDIN)、“标准输出” (STDOUT) 和“标准出错” (STDERR)。内核给每个进程都设置了这 3 个信道，所以进程本身不必知道这三个信道通到哪里。举例来说，它们可能连接到一个终端窗口、一条网络连接，或者属于另一个进程的信道。

UNIX 有一个统一的 I/O 模型，在这个模型中，每个信道都以一个整数来命名，它叫做文件描述符。分配给一个信道整数值到底是哪个，通常而言并没有意义，但要保证 STDIN、STDOUT 和 STDERR 对应文件描述符 0、1 和 2，所以保险的做法是，用数字来引用这三个信道。在交互式的终端窗口里，STDIN 一般读取键盘的输入，而 STDOUT 和 STDERR 把它们的输出写到屏幕上。

大多数命令都接受从 STDIN 来的输入，并且把自己的输出写到 STDOUT，而把出错消息写到 STDERR。有了这样的约定，用户就能把命令像积木一样串起来，创建出混合管道。shell 将 <、> 和 >> 解释成指令，用来把一条命令的输入或者输出重新定向到一个文件。< 这个符号把这条命令的 STDIN 和已有的某个文件的内容联系起来。符号 > 和 >> 则重定向 STDOUT；> 会替换文件的现有内容，而 >> 则给文件追加内容。例如，下面的命令

```
$ echo "This is a test message." > /tmp/mymessage
```

在 /tmp/mymessage 这个文件里存入一行内容，如果必要，还会创建这个文件。下面的命令把该文件的内容用电子邮件发给用户 johndoe。

```
$ mail -s "Mail test" johndoe < /tmp/mymessage
```

为了把 STDOUT 和 STDERR 都重定向到同一个地方，可以用 >& 这个符号。仅仅重定向 STDERR 的话，则用 2>。

命令 find 演示了想要分开处理 STDOUT 和 STDERR 的原因，因为它会在两个信道提供输出，特别是以非特权用户身份运行的时候。例如，像下面这条命令

```
$ find / -name core
```

通常会导致很多“permission denied”这样的出错消息，从而把真正的结果给淹没在混乱的输出里了。要消除所有出错消息，可以用这条命令：

```
$ find / -name core 2> /dev/null
```

在这个版本的命令里，只有真正匹配的结果（该用户对父目录有读权的地方）才会出现在终端窗口中。要把匹配路径的清单保存在一个文件里，可以试试下面的命令：

```
$ find / -name core > /tmp/corefiles 2> /dev/null
```

这一行命令把匹配的路径发到/**tmp/corefile** 这个文件，丢弃出错消息，向终端窗口什么都不发。

要把一条命令的 STDOUT 连接到另一条命令的 STDIN 上，可以用|这个符号，它常叫做管道。下面是一些管道的例子：

```
$ ps -ef | grep httpd  
$ cut -d: -f7 < /etc/passwd | sort -u
```

第一个例子运行 **ps** 产生一份进程清单，由管道送给 **grep** 命令选出包含 **httpd** 这个词的若干行。**grep** 命令的输出没有重定向，所以匹配的结果都出现在终端窗口里。

第二个例子用 **cut** 命令从/**etc/passwd** 文件里把每个用户的 shell 的路径选出来。接着，列出的 shell 的路径都通过 **sort-u** 进行处理，产生的清单中，路径名不但依次排序，且路径名只出现一次。

要让第二条命令只有在第一条命令成功完成之后才执行，可以用一个&&符号把两条命令隔开。例如：

```
$ lpr /tmp/t2 && rm /tmp/t2
```

这条命令当且仅当/**tmp/t2** 成功送入打印队列之后，才会删除/**tmp/t2**。在这里，**lpr** 命令产生的退出码为 0 的话，就算它执行成功，所以，如果读者已经习惯了其他编程语言中的“短路”计算，而这里用一个表示“逻辑与”的符号，那么就可能造成混乱。不要想得太多；仅仅把它当做一个 shell 的习惯用法就行了。

相反，||这个符号表明，只有前一条命令执行不成功（产生了一个非零的退出码）时，才执行后面的命令。

在一个脚本里，可以用反斜线把一条命令分成多行来写，从而把出错处理代码和命令管道

的其他部分区分开来：

```
cp --preserve --recursive /etc/* /spare/backup \  
|| echo "Did NOT make backup"
```

要实现相反的效果——将多条命令整合在一行里——可以用分号作为语句分隔符。

2.1.3 变量和引用

变量名在赋值的时候没有标记，但在访问它们的值的时候要在变量名之前加一个\$符。例如：

```
$ etcdir='/etc'  
$ echo $etcdir  
/etc
```

不要在等号两边留空白，否则 shell 会把变量名误认为是命令名。

引用一个变量的时候，可以用花括号把这个变量的名字括起来，让分析程序和阅读代码的人能清楚地知道变量名的起止位置；例如，用 **\${etcdir}** 代替 **\$etcdir** 。正常情况下不要求有这对花括号，但是，如果想要在双引号引起的字符串里扩展变量，它们就会派上用场了。因为人们经常想要在一个变量的内容之后跟着字母或者标点符号。例如：

```
$ echo "Saved ${rev}th version of mdadm.conf."  
Saved 8th version of mdadm.conf.
```

给 shell 变量起名字没有标准的命名规范，但如果变量名的所有字母都大写，一般表明该变量是环境变量，或者是从全局配置文件里读取的变量。本地变量则多半是所有字母都小写，而且在变量名的各个部分之间用下划线隔开。变量名区分大小写。

环境变量会被自动导入 **bash** 的变量名空间，所以它们可以用标准的语法来设置和读取。

命令 **export varname** 将一个 **shell** 变量提升为一个环境变量。用来在用户登录时设置环境变量的那些命令，都应该放在该用户的 **~/.profile** 或者 **~/.bash_profile** 这两个文件里。而其他像 **PWD**（代表当前工作目录）这样的环境变量都由 **shell** 自动维护。

对于用单引号和双引号括起来的字符串而言，**shell** 以相似的方式处理它们，例外之处在于双引号括起来的变量可以进行替换（用 * 和 ? 这样的文件名匹配元字符做扩展）和变量扩展。

例如：

```
$ mylang="Pennsylvania Dutch"
$ echo "I speak ${mylang}."
I speak Pennsylvania Dutch.
$ echo 'I speak ${mylang}.' 
I speak ${mylang}.
```

左引号也叫做撇号，对它的处理和双引号类似，但是它们还有其他作用，即能够把字符串的内容按一条 **shell** 命令来执行，并且用该命令的输出来替换这个字符串。例如：

```
$ echo "There are `wc -l /etc/passwd` lines in the passwd file."
There are 28 lines in the passwd file.
```

2.1.4 常见的过滤命令

任何“从 **STDIN** 读入数据，向 **STDOUT** 输出结果”这样循规蹈矩的命令，都可以当作一个过滤器（也就是说，管道的一个环节）来处理数据。在这一小节，我们简要回顾一些使用较为广泛的过滤器命令（包括上面已经用到过的一些命令），但是这样的过滤器命令实际上 是无穷无尽的。过滤器命令多面向“集团作战”，所以有时候它们各自的用处很难单独体现出来。

大多数过滤器命令都接受在命令行上提供的一个或者多个文件名作为输入。只有在一个文件都未指定的时候，它们才从自己的标准输入读取数据。

cut : 把行分成域

cut 命令从它的输入行中选出若干部分，再打印出来。该命令最常见的用法是提取被限定的若干域，如 2.1.2 节里的例子所示，但是它也能返回由列边界所限定的若干区段。默认的限定符是<Tab>，但是可以用**-d** 选项改变这个限定符。**-f** 选项指定输出里包括哪些域。参考下面介绍 **uniq** 命令一节的内容，了解 **cut** 用法的例子。

sort : 将行排序

sort 命令对输入行进行排序。简单吧，不是吗？或许并不简单——到底按每行的哪些部分（即“关键字”）进行排序，以及进行排序的顺序，都可以做精细的调整。表 2.1 给出了一些比较常见的选项，但要查看手册页才能了解到其他选项。

表 2.1 sort 命令的选项

选 项	含 义
-b	忽略开头的空白
-f	排序不区分大小写
-k	指定构成排序关键字的列
-n	按整数值比较域[译者注：即按数值排序]
-r	颠倒排序的顺序[译者注：即逆序]
-t	设定域分隔符（默认的分隔符是空白）
-u	只输出唯一记录[译者注：重复的记录只输出一次]

下面的命令展示出了数值排序和字典排序的不同之处，默认按字典排序。这两条命令都用

了-**t:**和-**k3,3**两个选项，对/etc/group文件的内容按照由冒号分隔的第三个域（即组ID）进行排序。第一条命令按照数值排序，而第二条命令则按照字母排序¹¹。

```
$ sort -t: -k3,3 -n /etc/group1
root:x:0:
bin:x:1:daemon
daemon:x:2:
...
$ sort -t: -k3,3 /etc/group
root:x:0:
bin:x:1:daemon
users:x:100:
...
```

uniq : 重复行只打印一次

uniq命令在思想上和**sort -u**类似，但它有一些**sort**不能模拟的选项：**-c** 累计每行出现的次数，**-d** 只显示重复行，而**-u** 只显示不重复的行。**uniq**命令的输入必须先排好序，因此通常把它放在**sort**命令之后运行。

例如，下面的命令显示出：有 20 个用户把/bin/bash 作为自己的登录 shell，12 个用户把/bin/false 作为登录的 shell（后者要么是伪用户，要么就是账号被禁用的用户）。

```
$ cut -d: -f7 /etc/passwd | sort | uniq -c
20 /bin/bash
12 /bin/false
```

WC : 统计行数、字数和字符数

统计一个文件里的行数、字数和字符数是另一项常用的操作，**wc**（表示 word count，即字数统计）命令是完成这项操作的一条方便途径。如果不带任何参数运行 **wc**，它会显示全部 3 种统计结果：

```
$ wc /etc/passwd
32 77 2003 /etc/passwd
```

¹¹. **sort**命令能够接受-**k3**（而不是-**k3,3**）这样的关键字说明，但它的执行效果可能不是用户期望的那样。如果没有指出结束域号，那么排序关键字就一直延续到行尾。

在脚本编程的应用场合里，常给 **wc** 命令加上**-l**、**-w** 或者**-c** 选项，让它只输出一个数。在撇号里最常出现这种形式的命令，这样一来，命令的执行结果就可以被保存起来，或者根据执行结果确定下一步的操作。

tee：把输入复制到两个地方

命令的管道一般都是线性的，但是从中间插入管道里的数据流，然后把一份副本发送到一个文件里，或者送到终端窗口上，也往往会很有帮助。用 **tee** 命令就能做到这一点，该命令把自己的标准输入既发送到标准输出，又发送到在命令行上指定的一个文件里。可以把它想成是水管上接的一个三通。

设备/**dev/tty** 是当前终端的同义语。例如：

```
$ find / -name core | tee /dev/tty | wc -l
```

该命令把名叫 **core** 的文件的路径名，以及找到的 **core** 文件的数量都打印出来了。

把 **tee** 命令作为一条执行时间很长的命令管道的最后一条命令，这是一种常见的习惯用法，这样一来，管道的输出既可以送到一个文件，又可以送到终端窗口供用户查看。用户可以预先看到一开始的输出结果，从而确保一切按预期执行，然后用户就可以在命令运行的同时不去管它，因为他们知道结果会被保存下来。

head 和 **tail**：读取文件的开头或者结尾

管理员会经常碰到一项操作，即查看一个文件开头或者结尾的几行内容。这两条命令默认显示前 10 行内容，但用户可以带一个命令行参数，指定到底要看多少行内容。

对于交互式的应用场合，**head** 命令已经或多或少被 **less** 命令所取代，后者能够给被显示的文件标出页数，但是 **head** 命令仍然在脚本里大量使用。

tail 也有一个不错的**-f** 选项，对于系统管理员来说，这个选项特别有用。**tail -f** 命令在按

要求的行数打印完之后，不是立即退出，而是等着有新行被追加到文件末尾，再随着新行的出现打印新行——对于监视日志文件来说很有用。不过要注意，写文件的那个程序可能会缓冲它的输出。即使从逻辑上讲，新行是按有规律的时间间隔追加的，但它们可能只按 1KiB 或者 4KiB 的块来显示¹²。

键入<Control-C>即可停止监视。

grep : 搜索文本

grep 命令搜索给它输入的文本，打印出匹配某种模式的行。它的名字源于 **g/regular-expression/p** 这条命令，该命令是 **ed** 来的（UNIX 系统仍然还有这种编辑器），**ed** 是最早版本的 UNIX 上所带的一种老编辑器。

“正则表达式”是匹配文本的模式，它用一种标准的、能准确刻画的模式匹配语言来编写。

尽管在不同的实现上，正则表达式存在轻微的变化，但它们是大多数做模式匹配的程序都要用到的一种通用标准。它之所以叫这么个奇怪的名字，是因为它起源于理论计算研究。我们将在 2.3 节更详细地讨论正则表达式。

和大多数过滤器一样，**grep** 命令有许多选项，这其中包括：打印匹配行数的**-c**、匹配时忽略大小写的**-i**，以及打印不匹配行（而不是匹配行）的**-v**。另一个有用的选项是**-l**（L 的小写），它让 **grep** 只打印匹配文件的名字，而不是匹配的每一行。例如，下面的命令

```
$ sudo grep -l mdadm /var/log/*
/var/log/auth.log
/var/log/syslog.0
```

表明 **mdadm** 的日志项出现在两个不同的日志文件里。

从传统上看，**grep** 命令是一个相当基础的正则表达式引擎，但是有些版本的 **grep** 能够选择其他正则表达式的变体语法。例如，Linux 上的 **grep -p** 命令选择采用 Perl 风格的

12 . 参见 1.8 节对这些单位的介绍。

表达式，虽然手册页警告说，它们还处在“实验初级阶段”。如果需要用这类正则表达式的完整功能，那么只能用 Perl 或者 Python 语言。

2.2 bash 脚本编程

bash 特别适合编写简单的脚本，用来自动执行那些以往在命令行输入的操作。在命令行用的技巧也能用在 **bash** 的脚本里，反之亦然，这让用户在 **bash** 上投入的学习时间获得了最大的回报。不过，一旦 **bash** 脚本超过了 100 行，或者需要的特性 **bash** 没有，那么就要换到 Perl 或者 Python 上了。

bash 脚本的注释以一个井号 (#) 开头，并且注释一直延续到行尾。和命令行中一样，可以把逻辑上的一行分成多个物理上的多行来写，每行末尾用反斜线消除换行符 (newline)。还可以用分号分隔语句的办法，在一行里书写多条语句。

bash 脚本可以只包含一系列的命令行，此外其他什么都没有。例如，下面的 **helloworld** 脚本就只有一条 **echo** 命令。

```
#!/bin/bash
echo "Hello, world!"
```

第一行叫做“#!”语句，它声明这个文本文件是一个脚本，要由**/bin/bash** 来解释。内核在决定如何执行这个文件的时候，要先找这个语句。从派生出来执行这个脚本的 shell 的角度来看，“#!”行只是一个注释行。如果 **bash** 不在这行指定的位置那里，那么就需要调整这行的内容。

要让这个文件做好能运行的准备，只要设置它的可执行位即可（参考 6.5.5 节）¹³。

13. 如果 **helloworld** 命令前面没有`./`，shell 也能理解，那么就意味着当前目录`(.)`也在搜索路径中。这个做法不好，因为它给其他用户提供了设下陷阱的机会，这样的用户希望您 `cd` 进入到他们有写权限的

```
$ chmod +x helloworld  
$ ./helloworld3  
Hello, world!
```

还可以把 shell 当做解释程序直接调用：

```
$ bash helloworld  
Hello, world!  
$ source helloworld  
Hello, world!
```

第一条命令在一个 shell 的新实例中运行 **helloworld** 脚本，第二条命令让当前的登录 shell 读取并执行这个文件的内容。当这个脚本用来设置环境变量，或者只对当前的 shell 做定制的时候，就采用后一种选择。在脚本编程中，这种形式常用来加入一个配置文件的内容，该文件里面写的是对一系列 **bash** 变量进行赋值¹⁴。

如果是 Windows 用户，那么可能已经习惯于这样的做法，即由文件的扩展名标明该文件的类型，以及是否可以执行。但在 UNIX 和 Linux 上，要由文件的权限位来指定一个文件是否可以执行，如果可执行，那么由谁可以执行。如果愿意，可以给自己的 **bash** 脚本加.**sh** 后缀，提醒用户它们是什么文件，但在运行该命令的时候，就必须得输入.**sh**，因为 UNIX 不会对扩展名做特殊处理。参考 6.5.1 节了解有关权限位的更多知识。

2.2.1 从命令到脚本

在我们开始介绍 **bash** 的脚本编程特性之前，先讲一下方法。大多数人写 **bash** 脚本的时候，都按照和他们写 Perl 或者 Python 脚本一样的方式：用一个文本编辑器来写。不过，把常规的 shell 命令行当做一种交互式的脚本开发环境，考虑这样用的话，效果会更高。

目录里，然后尝试执行某些命令。

14 .“点”命令和 **source** 命令同义，例如**helloworld**。

例如，假定在一个目录层次结构中，散布着很多日志文件，它们的名字后缀为.log 和.LOG，现在想把它们都改为大写的形式。首先，让我们看看是否能找到所有这样的文件。

```
$ find . -name '*log'  
.do-not-touch/important.log  
admin.com-log/  
foo.log  
genius/spew.log  
leather_flog  
...
```

哦，看起来我们要在搜索模式中包括点号（·），而且还要排除目录。键入<Control-P>重新找回这条**find**命令，然后对它进行修改。

```
$ find . -type f -name '*.log'  
.do-not-touch/important.log  
foo.log  
genius/spew.log  
...
```

好了，这次看上去结果更好了。不过，.do-not-touch 目录看上去挺危险的；我们或许不应该让它出来捣乱。

```
$ find . -type f -name '*.log' | grep -v .do-not-touch  
foo.log  
genius/spew.log  
...
```

好了，正好剩下需要的文件清单。让我们生成一些新的名字。

```
$ find . -type f -name '*.log' | grep -v .do-not-touch | while read fname  
> do  
> echo mv $fname ${fname/.log/.LOG/}  
> done  
mv foo.log foo.LOG  
mv genius/spew.log genius/spew.LOG  
...
```

好，那几条命令就是我们想要的命令，把它们运行起来就可以执行改名操作。那么在现实中，我们该怎么做呢？我们可以把这条命令重新找回来，编辑一下把**echo**去掉，让**bash**执行**mv**命令，而不仅仅是打印**mv**命令。不过，用管道把这些命令都送到另一个

`shell` 的实例，这样更不容易出错，而且需要对前面命令做的编辑也更少。

当键入`<Control-P>`的时候，我们会发现 **bash** 考虑得很精心，它把这个小小的脚本变成了一行。对于这个紧凑的命令行，我们只要加一个管道，把输出送给 **bash -x** 就行了。

```
$ find . -type f -name '*.log' | grep -v .do-not-touch | while read fname; do
    echo mv $fname ${fname/.log/.LOG}; done | bash -x
+ mv foo.log foo.LOG
+ mv genius/spew.log genius/spew.LOG
...
```

给 **bash** 加了`-x` 选项后，它在执行每条命令之前，会先打印这条命令。

我们现在已经完成了实际的改名工作，但是仍然想把整个脚本保存下来，以便可以再次使用它。**bash** 的内置命令 **fc** 和`<Control-P>`非常像，但它不是让上次的命令重新出现在命令行，而是把该命令送到用户选择的编辑器里。再加一个“#！”行和用法说明之后，把这个文件写到一个可以执行的地方（或许是`~/bin`，或者`/usr/local/bin`），让这个文件可执行，于是就得到了最终的脚本。

上述方法总结如下：

- 按一个管道的方式开发脚本（或者脚本的组成部分），一次开发一步，完全都在命令行上做；
- 把输出送到标准输出，检查并确保结果正确；
- 每开发一步，用 `shell` 的 **history** 命令重新找回命令管道，用 `shell` 的编辑功能调整它们；
- 在得到正确输出之前，都不实际执行任何操作，所以如果命令不正确，也不需要撤销什么操作；
- 一旦得到正确的输出，就真正执行命令，并核对命令能按预期要求工作；
- 用 **fc** 命令捕获工作结果，整理后保存下来。

在上面的例子里，我们打印出数行命令，然后用管道把它们送入一个子 `shell` 去执行。这

一技术并不一定行得通，但它经常还是有帮助的。另一种做法是，可以把输出重定向到一个文件，得到这个结果。无论怎样，都要预先看到正确的结果，才做任何可能有破坏性的操作。

2.2.2 输入和输出

echo 命令虽然原始，但易于使用。要想对输出做更多的控制，就需要使用 **printf** 命令。

因为采用 **printf** 的话，必须显式地在必要的地方加换行符（用“\n”），所以它用起来稍有不便，不过它也能让用户使用制表符，而且能让输出里的数字有更好的格式。比较下面两条命令的输出。

```
$ echo "\taa\tbb\tcc\n"
\taa\tbb\tcc\n
$ printf "%taa\tbb\tcc\n"
      aa    bb    cc
```

有些系统带有操作系统级的 **echo** 和 **printf** 命令，通常分别位于 **/bin** 和 **/usr/bin** 目录下。虽然这两条命令和 shell 的内置命令都很相似，但是它们的细节还是稍有不同，特别是 **printf**，差别更大一些。对此，要么坚持采用 **bash** 的语法，要么用完整路径名调用外部的 **printf** 命令。

用 **read** 命令可以提示输入。下面是一个例子：

```
#!/bin/bash
echo -n "Enter your name: "
read user_name
if [ -n "$user_name" ]; then
    echo "Hello $user_name!"
    exit 0
else
    echo "You did not tell me your name!"
    exit 1
fi
```

echo 命令的**-n** 选项消除了通常的换行符，但也可以在这里用 **printf** 命令。我们简要介绍一下 **if** 语句的语法，它的作用在这里很明显。**if** 语句里的**-n** 判断其字符串参数是否为空，不为空的话则返回真（**true**）。下面是这个脚本运行后的结果：

```
$ sh readexample
Enter your name: Ron
Hello Ron!
```

2.2.3 命令行参数和函数

给一个脚本的命令行参数可以成为变量，这些变量的名字是数字。**\$1** 是第一个命令行参数，**\$2** 是第二个，以此类推。**\$0** 是调用该脚本所采用的名字。这个名字可以是像`../bin/example.sh` 这样的奇怪名字，所以它的取值并不固定。

变量`$#`是提供给脚本的命令行参数的个数，变量`$*`里保存有全部参数。这两个变量都不包括或者算上`$0`。

如果调用的脚本不带参数，或者参数不正确，那么该脚本应该打印一段用法说明，提醒用户怎样使用它。下面这个脚本的例子接受两个参数，验证这两个参数都是目录，然后显示它们。如果参数无效，那么这个脚本会打印一则用法说明，并且用一个非零的返回码退出。如果调用这个脚本的程序检查该返回码，那么它就会知道这个脚本没有正确执行。

```

#!/bin/bash

function show_usage {
    echo "Usage: $0 source_dir dest_dir"
    exit 1
}

# Main program starts here

if [ $# -ne 2 ]; then
    show_usage
else # There are two arguments
    if [ -d $1 ]; then
        source_dir=$1
    else
        echo 'Invalid source directory'
        show_usage
    fi
    if [ -d $2 ]; then
        dest_dir=$2
    else
        echo 'Invalid destination directory'
        show_usage
    fi
fi

printf "Source directory is ${source_dir}\n"
printf "Destination directory is ${dest_dir}\n"

```

我们创建了一个单独的 `show_usage` 函数，用它打印用法说明。如果这个脚本以后又做了更新，能够接受更多的参数，那么只要在一个地方修改用法说明就行了¹⁵。

```

$ mkdir aaa bbb
$ sh showusage aaa bbb
Source directory is aaa
Destination directory is bbb
$ sh showusage foo bar
Invalid source directory
Usage: showusage source_dir dest_dir

```

bash 函数的参数就按命令行参数那样处理。第一个参数变成 `$1`，以此类推。正如上面的例子所示，`$0` 是这个脚本的名字。

要让上面的例子更健壮一点儿，我们可以编写 `show_usage` 函数，让它接受一个出错码作为参数。对于执行不成功的每一种不同类型，返回一个定义好的出错码。下面的代码片段给出了该函数的样子。

```

function show_usage {
    echo "Usage: $0 source_dir dest_dir"
    if [ $# -eq 0 ]; then
        exit 99 # Exit with arbitrary nonzero return code
    else
        exit $1
    fi
}

```

15. 注意，出错信息和用法说明都会送到标准输出。难道它们不应该送到标准出错信道吗？那样做实际上更正确，但是因为这个脚本并不是用作过滤器，所以这种差别并不重要。

下面这个版本的函数，其参数可有可无。在一个函数内部，\$#表明传入了多少个参数。如果没有提供更确定的出错码，那么这个脚本就返回代码 99。但是如果给这个函数一个确定的出错码值，就会让脚本在打印用法说明之后以那个出错码退出，例如：

```
show_usage 5
```

(shell 变量 \$? 是上次执行的命令退出的状态，而且无论该命令是在一个脚本内部使用，还是在命令行上使用。)

在 **bash** 里，函数和命令之间很类似。用户可以在自己的`~/.bash_profile` 文件里定义自己的函数，然后在命令行上使用它们，就好像它们是命令一样。例如，如果站点里统一将网络端口 7988 用于 SSH 协议（“不公开，即安全”的一种形式），就可以在`~/.bash_profile` 文件里定义

```
function ssh {  
    /usr/bin/ssh -p 7988 $*  
}
```

以保证 **ssh** 总是带选项`-p7988` 来运行。和许多 shell 一样，**bash** 也有一种别名机制，能更加简洁地再现上面这个限制端口的例子，不过采用函数的方法更通用，功能也更强。忘掉别名，采用函数吧。

2.2.4 变量的作用域

在脚本里的变量是全局变量，但是函数可以用 `local` 声明语句，创建自己的局部变量。考虑下面的代码：

```

function localizer {
    echo "==> In function localizer, a starts as '$a"
    local a
    echo "==> After local declaration, a is '$a"
    a="localizer version"
    echo "==> Leaving localizer, a is '$a"
}

a="test"
echo "Before calling localizer, a is '$a"
localizer
echo "After calling localizer, a is '$a"

```

下面的日志显示在 `localizer` 函数内，局部变量 `$a` 屏蔽了全局变量 `$a`。在 `localizer` 内，在碰到 `local` 声明了局部变量 `$a` 之前，全局变量 `$a` 都可见；`local` 实际上是一条命令，它从执行的那个地方开始，创建局部变量。

```

$ sh scopetest.sh
Before calling localizer, a is 'test'
==> In function localizer, a starts as 'test'
==> After local declaration, a is ''
==> Leaving localizer, a is 'localizer version'
After calling localizer, a is 'test'

```

2.2.5 控制流程

我们在本章里已经见过几种 `if-then` 和 `if-then-else` 语句的形式；它们的功能在其名字中得以体现。一条 `if` 语句的结束标识是 `fi`。要把几条 `if` 语句串起来，可以用 `elif` 这个关键字，它的意思是“`else if`”。例如：

```

if [ $base -eq 1 ] && [ $dm -eq 1 ]; then
    installDMBase
elif [ $base -ne 1 ] && [ $dm -eq 1 ]; then
    installBase
elif [ $base -eq 1 ] && [ $dm -ne 1 ]; then
    installDM
else
    echo '==> Installing nothing'
fi

```

用 `[]` 做比较的奇特语法，以及整数比较运算符的名字（例如，`-eq`），都看上去像是命令行选项，它们二者都是从原来 Bourne shell 的 `/bin/test` 命令延续下来的。方括号实际上是

调用 **test** 的一种快捷方式，而不是 **if** 语句的语法要求¹⁶。

表 2.2 给出了 **bash** 的数值和字符串比较运算符。**bash** 比较数值采用文字运算符，而比较字符串采用符号运算符，这正好和 Perl 相反。

表 2.2

bash 的基本比较运算符

字符串	数 值	为真，如果
$x = y$	$x -eq y$	x 等于 y
$x != y$	$x -ne y$	x 不等于 y
$x < y$	$x -lt y$	x 小于 y
$x \leq y$	$x -le y$	x 小于等于 y
$x > y$	$x -gt y$	x 大于 y

续表

字符串	数 值	为真，如果
$x \geq y$	$x -ge y$	x 大于等于 y
$-n x$	-	x 不为空
$-z x$	-	x 为空

bash 对文件属性取值的那些选项是其出彩之处（还是其**/bin/test** 遗留下来的特性）。

bash 有大量的测试文件和比较文件的运算符，表 2.3 列出了其中几个。

表 2.3

bash 的文件取值运算符

运 算 符	为真，如果
$-d file$	$file$ 存在且是目录
$-e file$	$file$ 存在

16. 在实际中，这些操作现在都内置于 shell 中，不需要真的运行**/bin/test**。

<code>-f file</code>	<code>file</code> 存在且是普通文件
<code>-r file</code>	用户有 <code>file</code> 的读权限
<code>-s file</code>	<code>file</code> 存在且不为空
<code>-w file</code>	用户有 <code>file</code> 的写权限
<code>file1 -nt file2</code>	<code>file1</code> 比 <code>file2</code> 新
<code>file1 -ot file2</code>	<code>file1</code> 比 <code>file2</code> 旧

虽然 `elif` 的形式能用，但是为了清楚起见，用 `case` 语句做选择是更好的方法。`case` 的语法如下面的这个函数例程所示，该函数集中给一个脚本写日志。特别值得注意的是，每一个选择条件之后有个右括号，而在条件符合时每个要执行的语句块之后有两个分号。`case` 语句以 `esac` 结尾。

```
# The log level is set in the global variable LOG_LEVEL. The choices
# are, from most to least severe, Error, Warning, Info, and Debug.

function logMsg {
    message_level=$1
    message_itself=$2

    if [ $message_level -le $LOG_LEVEL ]; then
        case $message_level in
            0) message_level_text="Error" ;;
            1) message_level_text="Warning" ;;
            2) message_level_text="Info" ;;
            3) message_level_text="Debug" ;;
            *) message_level_text="Other"
        esac
        echo "${message_level_text}: $message_itself"
    fi
}
```

这个函数演示了许多系统管理应用经常采取的“日志级别”方案。脚本的代码产生详尽程度不同的日志消息，但是只有那些在全局设定的阈值 `$LOG_LEVEL` 之内的消息才被真正记录到日志里，或者采取相应的行动。为了阐明每则消息的重要性，在消息文字之前用一个标签说明其关联的日志级别。

2.2.6 循环

bash 的 `for...in` 结构可以让它很容易对一组值或者文件执行若干操作，尤其是和文件名通配功能（对诸如`*`和`?`这样的模式匹配字符进行扩展，形成文件名或者文件名的列表）联合起来使用的时候。在下面这个 `for` 循环里，其中的`*.sh` 模式会返回当前目录下能够匹配的文件名列表。`for` 语句则逐一遍历这个列表，接着把每个文件名赋值给变量`$script`。

```
#!/bin/bash
suffix=BACKUP--`date +%Y%m%d-%H%M`
for script in *.sh; do
    newname="$script.$suffix"
    echo "Copying $script to $newname..."
    cp $script $newname
done
```

输出结果如下：

```
$ sh forexample
Copying rhel.sh to rhel.sh.BACKUP--20091210-1708...
Copying sles.sh to sles.sh.BACKUP--20091210-1708...
...
```

在这里的上下文关系中，对文件名做扩展并没有什么玄妙之处；它的做法就和在命令行上一模一样。也就是说，先扩展，然后再由解释器处理已经扩展过的这一行¹⁷。也可以静态地输入文件名，就像下面这行一样。

```
for script in rhel.sh sles.sh; do
```

实际上，任何以空白分隔的对象列表，包括一个变量的内容，都可以充当 `for ...in` 语句的目标体。

bash 也有从传统编程语言看来更为熟悉的 `for` 循环，在这种 `for` 循环里，可以指定起始、

¹⁷. 更准确地说，文件名扩展是有一点儿小玄机的，因为它确实保持了每个文件名完整不被分割。带有空格的文件名也在 `for` 循环的一次循环中处理。

增量和终止子句。例如：

```
for (( i=0 ; i < $CPU_COUNT ; i++ )); do
    CPU_LIST="$CPU_LIST $i"
done
```

接下来的例子演示了 **bash** 的 `while` 循环，这种循环也能用于处理命令行参数，以及读取一个文件里的各行。

```
#!/bin/bash
exec 0<$1
counter=1
while read line; do
    echo "$counter: $line"
    $((counter++))
done
```

下面是输出结果：

```
ubuntu$ sh whileexample /etc/passwd
1: root:x:0:0:Superuser:/root:/bin/bash
2: bin:x:1:1:bin:/bin:/bin/bash
3: daemon:x:2:2:Daemon:/sbin:/bin/bash
...
```

这个脚本片段有两个有趣的功能。`exec` 语句重新定义了该脚本的标准输入，变成由第一个命令行参数指定的任何文件¹⁸。这个文件必须要有，否则脚本就会出错。

在 `while` 子句里的 `read` 语句实际上是 `shell` 的内置命令，但它的作用就和一条外部命令一样。外部命令也可以放在 `while` 子句里；在这种形式下，当外部命令返回一个非零的退出状态时，就会结束 `while` 循环。

表达式`$((counter++))`实际上是个丑小鸭。`$((...))`这样的写法要求强制进行数值计算。它还可以利用`$`来标记变量名。`++`是人们在 C 和其他语言中熟悉的后置递增运算符。它返回它前面的那个变量的值，但返回之后还要把这个变量的值再加 1。

¹⁸ . 根据这里调用的方式，`exec` 也有大家熟悉的“停止这个脚本，把控制权交给另一个脚本或者表达式”这样的含义。通过同一条语句就能访问两个函数，这也是 `shell` 的奇特之处。

`$((...))` 的技巧在双引号里也起作用，所以可以把整个循环体紧凑地写到一行里。

```
while read line; do
    echo "$((counter++)): $line"
done
```

2.2.7 数组和算术运算

复杂的数据结构和计算不是 **bash** 的特长。但它的确至少提供了数组和算术运算。

所有 **bash** 变量的值都是字符串，所以 **bash** 在赋值的时候并不区分数字 1 和字符串 “1”。不同之处在于如何使用变量。下面几行代码展示出了其中的差异：

```
#!/bin/bash
a=1
b=$((2))
c=$a+$b
d=$((a+b))
echo "$a + $b = $c \t(plus sign as string literal)"
echo "$a + $b = $d \t(plus sign as arithmetic addition)"
```

这个脚本产生的输出如下：

```
1 + 2 = 1+2 (plus sign as string literal)
1 + 2 = 3   (plus sign as arithmetic addition)
```

注意给\$c 赋值的语句，其中的加号（+）连字符串的连接运行符都不是。它仅仅就是一个字符而已。那行代码等价于

```
c="$a+$b"
```

为了强制进行数值计算，要把这个表达式放在`$((...))`里面，就像上面给\$d 赋值那样。但即便如此，也不会让\$d 获得一个数值；它的值仍然保存为字符串“3”。

bash 通常能够混合使用算术、逻辑和关系运算符；参考手册页了解详情。

bash 中的数组有点儿怪，所以不常用到它们。然而，如果需要它们也依然可以用。数组用括号括起来，数组元素之间用空白隔开。数组元素中的空白要用引号引起来。

```
example=(aa 'bb cc' dd)
```

单个数组元素用 `${array_name[subscript]}` 来访问。下标从 0 开始。下标`*`和`@`指整个数组， `${#array_name[*]}` 和 `${#array_name[@]}` 这两种特殊形式表示数组里元素的个数。不要把它们和似乎更合乎逻辑的 `${#array_name}` 搞混了；后者实际上是数组第一个元素的长度（等价于 `${#array_name[0]}`）。

读者可能会以为`$example[1]` 是指数组的第二个元素，这一点无可争议，但 **bash** 对这个字符串的分析结果却是：`$example`（即`$example[0]` 的简洁引用形式）加上一个字符串`[1]`。在访问数组变量的时候，一定要带花括号——这一点无一例外。

下面是一个快速脚本，它演示了 **bash** 中数组管理的一些功能和缺陷：

```
#!/bin/bash
example=(aa 'bb cc' dd)
example[3]=ee
echo "example[@] = ${example[@]}"
echo "example array contains ${#example[@]} elements"
for elt in "${example[@]}"; do
    echo " Element = $elt"
done
```

这个脚本的输出如下：

```
$ sh arrays
example[@] = aa bb cc dd ee
example array contains 4 elements
Element = aa
Element = bb cc
Element = dd
Element = ee
```

这个例子似乎很直观易懂，但只是因为我们已经把这个脚本构造得循规蹈矩了。人们一不小心就会犯错误。例如，用下面这一句替换 for 语句那行代码：

```
for elt in ${example[@]}; do
```

(在数组表达式外面没有用引号引起来) 也能行 , 但它却不是输出 4 个数组元素 , 而是 5 个 : aa 、 bb 、 cc 、 dd 和 ee 。

这背后的问题是 , 因为所有 **bash** 变量实质上仍然是字符串 , 所以数组的表象充其量还是不确定的。字符串什么时候分割成数字元素 , 怎样分割成数组元素 , 都有很多细微变化。读者可以使用 Perl 或者 Python , 或是用谷歌搜索 Mendel Cooper 的 *Advanced Bash-Scripting Guide* 来研究这些细微差别。

2.3 正则表达式

尽管有些语言比其他语言对正则表达式更关注 , 但大多数现代语言都支持正则表达式。诸如 **grep** 和 **vi** 这样的 UNIX 命令也使用正则表达式。正则表达式如此常见 , 以至于通常都把其名称缩写为 “regex” 。有几本书全篇就讲如何发掘正则表达式的威力 , 而且无数博士论文的主题也都是正则表达式。

在解释诸如 **wc -l *.pl** 这样的命令行时 , shell 会做文件名匹配和扩展 , 但这并不是正则表达式匹配的一种形式。它是一种称之为 “shell 通配” 的不同体系 , 并且它采用了一种不同的、更为简单的语法。

正则表达式非常强大 , 但是它们不能支持所有可能的语法。其最著名的弱点就是 : 不能识别嵌套的限定符。例如 , 当允许用圆括号组织算术表达式的时候 , 就不可能写出一个正则表达式 , 它能够辨别出这样的算术表达式是否有效。

正则表达式在 Perl 语言中达到了其威力和完美的巅峰。实际上 , Perl 的模式匹配功能如此精致 , 以至于把它们称为正则表达式的一种实现 , 都显得不够准确。Perl 的模式能够匹配

嵌套的限定符，能够识别回文，还能够匹配由若干个 A 后跟相同数量的 B 所组成的任意字符串——所有这些都超出了正则表达式能够支持的范围。当然，Perl 也能处理普通的正则表达式。

Perl 的模式匹配语言保持为业界的标准，它已经被其他语言和工具广泛采纳。Philip Hazel 写的 PCRE 库（Perl-compatible regular expression，兼容 Perl 的正则表达式）让开发人员能够相当容易地把这种语言加入到他们自己的项目中。

正则表达式自身并不是一种脚本编程语言，但是它们作用很大，以至于只要讨论脚本编程就要专门介绍它们的特色功能；因此也就有了本节内容¹⁹。本节我们讨论它们的基本形式，但带有一些 Perl 的改进。

2.3.1 匹配过程

当代码要判断一个正则表达式的值时，它会尝试用一个给定的文本字符串去匹配一个给定的模式。要匹配的“文本字符串”可以很长，其中可以包括换行符。用一个正则表达式去匹配整个文件或者 HTML 文档的内容，往往是非常方便的。

整个搜索模式必须和一段连续的搜索文本匹配，匹配程序才宣布匹配成功。不过，这个模式可以在任意位置进行匹配。成功匹配一次之后，取值程序就返回匹配出来的文本，以及该模式里所有特别限定的部分所匹配的结果列表。

2.3.2 普通字符

一般而言，在正则表达式中的字符就匹配它们自己。所以下面这个模式

I am the walrus

就匹配字符串“I am the walrus”，并且只匹配这个字符串。因为它能够匹配搜索文本中的任意位置，所以这个模式也能匹配成功字符串“I am the egg man. I am the walrus. Koo koo ka-choo!”不过，实际匹配的地方仅限于“I am the walrus”这部分。匹配是区分大小写的。

19. Perl 语言的专家 Tom Christiansen 评论道，“我不知道‘脚本编程语言’是什么，但我同意正则表达式既不是程式语言，也不是函数式语言的提法。正则表达式是一种基于逻辑的或者声明性的语言，这类语言还包括 Prolog 和 Makefile。还有 BNF[译者注：Backus-Naur Form，巴科斯范式]。有人可能也把它们称为基于规则的语言。我自己更愿意把它们叫做声明性语言。”

表 2.4 给出了在正则表达式中常见的一些特殊符号的含义。它们都是基本字符——还有许多其他字符。

表 2.4

正则表达式里的特殊字符

符 号	匹配或者作用
.	匹配任何字符
[chars]	匹配给定字符集内的任何字符
[^chars]	匹配给定字符集之外的任何字符
^	匹配一行的开头
\$	匹配一行的结尾

续表

符 号	匹配或者作用
\w	匹配任何“单词”字符（和[A-Za-z0-9_]等价）
\s	匹配任何空白字符（和[\f\t\n\r]等价） ^a
\d	匹配任何数字（和[0-9]等价）
	匹配该符号左边或者右边元素的任何一个
(expr)	限定范围、元素成组，从而可以捕获到匹配
?	匹配前面元素的零个或者一个
*	匹配前面元素的零个、一个或者多个
+	匹配前面元素的一个或者多个
{ n }	匹配前面元素的 n 个实例
{ min, }	匹配前面元素的至少 min 个实例（注意有个逗号）
{ min,max }	匹配前面元素从 min 到 max 之间任意个数的实例

a.也就是说，一个空格、一个换页、一个制表符、一个换行符或者一个回车

许多特殊结构，像`+`和`|`，都会影响它们左边或者右边“东西”的匹配。一般而言，这个“东西”可以是一个字符、用括号括起来的子模式，或者用方括号括起来的字符类。不过，对于`|`这个字符来说，前面所说的“东西”的范围可以向左右无限制地扩展。如果想要限制这个竖线的作用域，那么可以把这个竖线和左右两边的东西，都一起放到它们自己的一对圆括号里。例如，

```
I am the (walrus|egg man)\.
```

就匹配“`I am the walrus.`”或者“`I am the egg man.`”。这个例子也演示了特殊字符（在这里是点）的转义。下面的模式

```
(I am the (walrus|egg man)\. ?){1,2}
```

匹配下面所有的句子：

- `I am the walrus.`
- `I am the egg man.`
- `I am the walrus. I am the egg man.`
- `I am the egg man. I am the walrus.`

它偏偏也匹配“`I am the egg man. I am the egg man.`”这一句（但那句话有什么意义呢？）。更重要的是，它还匹配“`I am the walrus. I am the egg man. I am the walrus.`”这句话，即使重复数明确限制为至多两次。那是因为这个模式不需要匹配整个搜索文本。本例中的正则表达式匹配两个句子之后就终止了，并宣告匹配成功。它根本不关心还有一次重复。

有一种常见的错误，把正则表达式的元字符`*`（零次或者多次的量词）和 shell 的通配符搞混。正则表达式中的星号需要有东西去修饰；否则，它并不会按照预期去起作用。如果任

何字符序列（包括根本没有字符）都是能够接受的匹配结果，那么就用`*`。

2.3.4 正则表达式的例子

在美国，邮政编码是 5 个数字，或者 5 个数字后面加一个短划线和另外 4 个数字。要匹配一个常规的邮政编码，就必须匹配一个有 5 位数字的数。下面的正则表达式刚好符合要求：

```
^\d{5}$
```

`^` 和 `$` 匹配搜索文本的开头和结尾，但是没有实际对应文本中的字符；它们是“零宽界定符（zero-width assertion）”。这两个字符确保了一点，只有正好 5 个数字才能匹配这个正则表达式——对于在一个更长字符串里面包含的 5 个数字，这个正则表达式不会去匹配它们。`\d` 这个转义符匹配一个数字，量词`{5}` 表明必须正好匹配 5 个数字。

为了既能匹配 5 位数字的邮政编码，也能匹配那种扩展形式（即邮政编码 +4 位数字），就要加上一个可有可无的短划线，以及另外 4 位数字：

```
^\d{5}(-\d{4})? $
```

圆括号把短划线和多出来的数字放到一起，这样一来，它们就被当做一个整体上可有可无的单元。例如，这个正则表达式不会匹配 5 位数字后跟一个短划线。如果出现了短划线，那么也必须出现 4 位数字的扩展，否则就不会匹配。

下面的表达式是演示正则表达式匹配的一个经典例子：

```
M[ou]l?am+[ae]r ([AEae]l[- ])?[GKQ]h?[aeu]+([dtz][dhz]?)+af[iy]
```

它匹配了利比亚领导人卡扎菲名字的大多数不同拼法，包括：

- Muammar al-Kaddafi (BBC)
- Moammar Gadhafi (美联社)
- Muammar al-Qadhafi (卡塔尔半岛电视台)
- Mu'ammar Al-Qadhafi (美国国务院)

能看出这些名字是怎样匹配该模式的吗？

这个正则表达式也展示出：达到人们能够阅读清楚的上限有多快。许多正则表达式体系（包括 Perl 中的体系）都支持一个 `x` 选项，它忽略模式里的空白，并支持注释，从而使得模式能够被隔开，分成多行显示。于是用户就可以用空白把逻辑组划分开，搞清楚其中的关系，就像用一种过程式语言那样。例如：

```
M [ou] ? a m+ [ae] r  # First name: Mu'ammar, Moamar, etc.
\s                      # Whitespace; can't use a literal space here
(
  [AEae] l            # Group for optional last name prefix
  [-\s]                #   Al, El, al, or el
)?                     #   Followed by dash or space
[GKQ] h? [aeu]+        # Initial syllable of last name: Kha, Qua, etc.
(                      # Group for consonants at start of 2nd syllable
  [dtz] [dzh]?        #   dd, dh, etc.
)+                     #
af [iy]
```

这样做有点儿帮助，但是仍然很容易折磨以后阅读代码的人。所以要做得友善些：如果可以，就用层次匹配或者多个小的匹配，而不用一个较大的正则表达式覆盖所有可能的情况。

2.3.5 捕获

一次匹配成功之后，每一对圆括号都变成了一个“捕获组”，它们记录下了该正则表达式匹配的实际文本。这些结果的使用方式则取决于具体实现和上下文环境。在 Perl 中，可以把匹配结果当做一个列表或者一系列被编了号的变量来访问。

因为圆括号可以嵌套，怎样知道哪个匹配哪个呢？这很简单——匹配的次序和左括号的次序一样。捕获的数量和左括号的数量一样多，不管每个用括号括起来的捕获组在实际匹配中扮演什么角色（或者不扮演什么角色）。当括号括起来的捕获组没有用到的时候（例如当用 Mu(')?ammar 匹配“Muammar”的时候），它对应的捕获组就为空。

如果一个捕获组匹配了不止一次，那么只返回最后一次匹配的内容。例如，用下面这个模式

```
(I am the (walrus|egg man)\. ?){1,2}
```

匹配文本

I am the egg man. I am the walrus.

的话，就会得到两个结果，一个结果对应一对括号。

```
I am the walrus.  
walrus
```

注意，这两个捕获组实际都匹配了两次。不过，实际只返回每对括号最后一次匹配的文本。

2.3.6 贪心、懒惰和灾难性的回溯

正则表达式从左到右进行匹配。模式的每个成分都要匹配尽可能长的字符串，然后再让位给下一个成分，这种特性称为“贪心”。

如果正则表达式匹配器达到了一种不能完成一次匹配的状态，那么它就从候选的匹配结果那里退回一点儿，让一个贪心的原子成分少匹配一点儿自己的文本。例如，考虑用正则表达式 a^*aa 匹配输入的文本“aaaaaa”。

首先，正则表达式匹配器把整个输入都分配给这个正则表达式中的 a^* 部分，因为 a^* 是贪心的。在没有可匹配的 a 之后，匹配器就继续尝试匹配正则表达式接下来的部分。不过情况不妙，接下来的部分是个 a ，再没有输入文本能匹配一个 a 的了；这就到了回溯的时候。正则表达式的 a^* 部分不得不匹配一个它已经匹配过的 a 。

现在匹配器能够匹配 a^*a 了，但它仍然不能匹配这个模式里的最后那个 a 。所以它又得回溯，让 a^* 再次腾出一个 a 。现在该模式里的第二个和第三个 a 都有匹配的 a 了，匹配也就完成了。

这个简单的例子展示出一些要点。首先，在处理整个文件时，贪心的匹配方式加上回溯会让明显很简单的模式（如 `<img.*></tr>`）开销很大²⁰。正则表达式中 * 的部分一开始就开始匹配了从第一个 `<img` 到输入结束的所有内容，只有通过反复回溯它才缩到和局部标签相吻合。

而且，这个模式绑定的 `></tr>` 部分是输入中“最后一个可能”的有效匹配，这或许并不是用户想要的效果。更可能的情况是，用户想要匹配一个 `` 后跟一个 `</tr>` 标签。这个模式更好的写法是 `<img[^>]*></tr>`，让一开始的通配符只匹配到当前这个标签的结尾，因为它不能超过右尖括号形成的界限。

用户还可以使用懒惰（和贪心正好相反）通配符：用 $^?$ 来代替 $*$ ，用 $^+?$ 来代替 $^+$ 。这两个版本的通配符匹配尽可能少的输入字符。如果不能匹配更少的话，它们就多匹配一些字符。在许多情况下，这两个运算符比贪心的版本更有效，而且更接近用户想要的结果。

不过要注意，它们得到的匹配结果和贪心运算符的不一样；差异不仅仅是这一种实现。对于我们给的 HTML 那个例子，懒惰模式是 `<img.^?></tr>`。但即便在这里， $^?$ 最终都会

20. 虽然这一节节选了 HTML 片段作为要匹配的文本样例，但是正则表达式确实不是做这项工作的正确工具。我们外聘的评审人无一例外地对此表示吃惊。Perl 和 Python 都带有优秀的插件，专门分析 HTML 文档。于是用户可以用 XPath 选择器访问到自己感兴趣的部分。参考 XPath 的维基网页以及各种语言的模块库了解详情。

扩大到包括不想要的>，因为之后接下来的标签可能不是一个</tr>。这又可能是用户想要的结果。

有多个通配部分的模式会导致正则表达式匹配器的处理量呈指数增长，如果文本的各个部分能够匹配几个通配表达式，而搜索文本实际上并不匹配该模式的话，处理量都会尤其大。这种情况可没有听上去的那么少见，特别是对HTML做模式匹配的时候。想匹配某些标签后面跟着其他标签，其间可能还被更多的标签隔开，这是频繁遇到的情形，但这种模式可能会要求正则表达式匹配器尝试许多可能的组合。

正则表达式专家Jan Goyvaerts把这种现象称为“灾难性的回溯”，他在自己的博客里写下了有关这种现象的内容；参考regular-expressions.info/catastrophic.html了解详情以及一些好的解决方案。

其中几个能用到的办法是：

- 如果可以一行一行地进行模式匹配，而不是一次整个文件进行模式匹配，那么性能差的风险就会小好多；
- 即使正则表达式的写法默认采用了贪心运算符，但是可能不应该用它们，用懒惰运算符吧；
- 出现.*的地方本身全都值得怀疑，应该仔细检查。

2.4 Perl 编程

Larry Wall发明了Perl语言，它第一种真正伟大的脚本编程语言。它的能耐要比**bash**大得多，而且编写良好的Perl代码也相当容易阅读。另一方面，Perl没有给开发人员强加太多的风格规范，所以不考虑可读性的Perl代码显得很神秘。Perl也被诟病为只适合写（不

适合读) 的语言。

我们在这里介绍 Perl 5 , 这个版本成为标准已经有 10 年了。Perl 6 是一个仍处在开发之中的主要版本。参考 perl6.org 了解详情。

对于系统管理工作来说 , Perl 或者 Python (2.5 节开始讨论) 都是比传统的编程语言 (如 C 、 C++ 、 C# 和 Java) 更好的选择。它们做得更多 , 但代码行数更少 , 程序员调试的痛苦更少 , 而且省却了编译的麻烦。

选择哪种语言通常取决于个人的偏好 , 或者取决于雇主强加给雇员的标准。Perl 和 Python 都提供了由用户群编写的模块库和语言扩展库。Perl 存在的年头更长 , 所以它提供的库长尾效应更为明显 [译者注 : 即功能繁多影响力不大的库数量更庞大 , 累积起来的效果也更大] 。不过 , 对于常见的系统管理任务而言 , 两者的支持库大致上等价。

Perl 的口号是“条条大路通罗马。”要记住本节读到的大多数例子都能用别的方法来实现。

Perl 的语句用分号分隔²¹。注释以一个井号 (#) 开头 , 并且一直到这一行的结尾。语句块用花括号括起来。下面是一个简单的“hello , world !”程序 :

```
#!/usr/bin/perl  
print "Hello, world!\n";
```

和 **bash** 程序一样 , 必须用 **chmod +x** 将这个文件改为可执行 , 或者直接调用 Perl 的解释程序执行它。

```
$ chmod +x helloworld  
$ ./helloworld  
Hello, world!
```

Perl 脚本中的代码行都不是 shell 命令 ; 它们是 Perl 代码。**bash** 可以让用户把一系列命令组合起来 , 把它叫做脚本 , 但 Perl 和 **bash** 不一样 , 除非让 Perl 去看外面的世界 , 否

21. 因为分号是分隔符而不是终结符 , 所以在一个代码块里的最后一个分号可有可无。

则 Perl 自己不会看。也就是说，Perl 提供了许多和 **bash** 一样的惯例，如使用撇号来获得一条命令的输出结果。

2.4.1 变量和数组

Perl 有 3 种基本数据类型：标量（也就是说，像数和字符串这样的一元量）、数组和哈希（hash）。哈希也叫做关联数组。变量的类型总是一目了然，因为它体现在变量名上：标量的变量以\$开头，数组变量以@开头，而哈希变量以%开头。

在 Perl 语言里，“列表（list）”和“数组（array）”这两个术语经常混用，不过或许更准确的说法是，列表是一系列的值，而数组是能够保存这样一个列表的变量。数组里的各个元素都是标量，所以和普通的标量变量一样，它们的名字都以\$开头。数组下标从零开始，数组@a 里元素的最大下标是\$#a。这个值加 1 就等于数组的长度。
数组@ARGV 保存着该脚本的命令行参数。可以像访问其他任何数组那样来访问它。

下面的脚本展示了数组的用法：

```
#!/usr/bin/perl  
@items = ("socks", "shoes", "shorts");  
printf "There are %d articles of clothing.\n", $#items + 1;  
print "Put on @{$items[2]} first, then ", join(" and ", @items[0..1]), ".\n";
```

输出为：

```
$ perl clothes  
There are 3 articles of clothing.  
Put on shorts first, then socks and shoes.
```

仅仅在这几行代码中，看点就很多。冒着分散我们注意力的风险，我们在所举的每个 Perl 例程中都包含了几种常见的习惯用法。在每个例子之后的文字中，我们都会阐述其中的窍

门。如果仔细阅读这些例子（不要胆怯，它们都不长！），在看完本章的内容之后，就会对 Perl 最常碰到的形式有了一些经验。

2.4.2 数组和字符串文字

在本例中，首先要注意（…）创建了一个列表。这个列表里的每个元素都是字符串，它们由逗号隔开。创建好这个列表之后，就把它赋值给变量 @items。

Perl 不严格要求所有的字符串都要用引号引起。在这个具体的例子里，没有引号也能给 @items 赋初值。

```
@items = (socks, shoes, shorts);
```

Perl 把这些没有用引号引起的字符串称为“裸单词（bareword）”，它们就按上次访问的方式来解释。如果用任何其他方式解释都没有意义，Perl 就尝试把它按一个字符串来解释。在有限的几种情况下，这样解释有意义，并且能让代码保持整洁。不过，这里不一定正好就是上述几种情况之一。即使自己能一直保持用引号把字符串引起，也要有所准备，去分析别人写的没有用引号引起的代码。

要初始化这个数组，更 Perl 化的办法是用 qw（指 quote words 的缩写）操作符。它实际上是把字符串用引号引起的一种形式，而且和 Perl 里大多数被引号引起的实体一样，可以选择自己的限定符。下面的形式

```
@items = qw(socks shoes shorts);
```

是最传统的办法，但是它容易误导人，因为 qw 之后的部分不再是一个列表了，它实际上是一个用空白分隔字符串形成的一个列表。下面的这个版本的写法

```
@items = qw[socks shoes shorts];
```

也能用，而且对于真正要做什么来说，这种写法可能还更正确一点儿。注意逗号没了，因为 `qw` 已经包括了它们的功能。

2.4.3 函数调用

`print` 和 `printf` 都能接受任意数量的参数，这些参数由逗号分隔。但是 `join(...)` 看上去更像是某种函数调用；它和 `print` 和 `printf` 有怎样的不同呢？

实际情况并非如此；`print`、`printf` 和 `join` 都是普通的函数。如果不会引起歧义，Perl 允许省略函数调用的圆括号，所以两种调用形式（带括号和不带括号）都很常用。在上面例子的 `print` 那句中，圆括号把 `join` 的参数同 `print` 的参数给区分开了。

我们可以说表达式 `@items[0..1]` 必定要按某种列表来算，因为它以 @ 开头。这实际上是一个“数组段”或者叫子数组，0 和 1 两个下标列出了在这个数组段里包含的数组元素的索引。Perl 在这里也能接受一个范围值，就像其等价表达式 `@items[0..1]` 里出现的那样。这里也能接受单个数值下标：`@items[0]` 是一个列表，里面只有一个标量，即字符串“socks”。在这种情况下，它等价于 ("socks") 这个常量。

函数调用会自动扩展数组，如下面的表达式中

```
join(" and ", @items[0..1])
```

`join` 接收到 3 个字符串参数：“and”、“socks”和“shoes”。它把第二个及后面的参数连起来，在两个参数之间插入第一个参数。结果是“socks and shoes”。

2.4.4 表达式里的类型转换

在 **printf** 那一行，`$#items + 1` 计算得到数字 3。由此看来，`$#items` 是一个数值，但这并不是这个表达式要按数值计算的原因；"2" + 1 也是可以的。关键在于运算符 +，它总是暗指算术运算。它把自己的参数转为数字，并计算得到数值结果。类似地，点运算符 (·) 把两个字符串连接起来，它根据需要转换自己的操作数："2" . (12 ** 2) 得到 "2144"。

2.4.5 字符串表达式和变量

和 **bash** 里的情况一样，双引号引起的字符串可以进行变量扩展。和 **bash** 里一样的还有，如果必要，可以用花括号把变量名括起来，如 `${items[2]}` ，从而避免歧义（这里的花括号只用于演示；它们不是必须的）。\$ 暗示这个表达式应该按标量来算。@items 是数组，但是它的任何一个元素都是标量，起名字的习惯就反映出了这一事实。

2.4.6 哈希

哈希（也称为关联数组）表示一组“键/值”对。可以把一个哈希当作下标（键）是任意标量值的数组；它们不一定是数字。但在实际中，数字和字符串都是常用的键。

哈希变量的第一个字符是 %（例如，%myhash），但是和在数组中的情况一样，哈希里的单个值都是标量，所以也以 \$ 开头。哈希的下标用花括号而不是方括号括起来，例如

```
$myhash{'ron'}。
```

哈希是系统管理员的一个重要工具。系统管理员编写的几乎每个脚本都会用到它们。在下面的代码中，我们会读取一个文件的内容，按照/etc/passwd 里的规则分析它，然后用分析得到的若干项结果构造一个叫做%names_by_uid 的哈希。这个哈希中每一项都是用户名和它对应的 UID。

```
#!/usr/bin/perl  
while ($_ = <>) {  
    ($name, $pw, $uid, $gid, $gecos, $path, $sh) = split /:/;  
    $names_by_uid{$uid} = $name;  
}  
  
%uids_by_name = reverse %names_by_uid;  
  
print "$names_by_uid[0] is $names_by_uid[0]\n";  
print "$uids_by_name['root'] is $uids_by_name['root']\n";
```

和上面的那个示例脚本一样，我们在这几行代码里也加入了几点新思想。在开始介绍这几处细节之前，先看一下这个脚本的输出：

```
$ perl hashexample /etc/passwd  
$names_by_uid[0] is root  
$uids_by_name['root'] is 0
```

while (\$_ = <>) 这条语句一次一行读取输入，把它赋值给名为\$_的变量；和 C 语言一样，整个赋值语句的值是等号右边的值。当碰到输入的结尾时，<> 返回一个出错值，然后结束循环。

Perl 对<>的解释为，检查命令行看是否在那里给出了任何文件。如果提供了文件，那么它就依次打开每个文件，然后在循环里运行这个文件的内容。如果没有在命令行给出任何文件，那么 Perl 就考虑从标准输入获得输入。

在循环体内，把 split 返回的值赋值给一系列变量，split 是个函数，它用传给它的正则表达式作为域分隔符，对输入字符串进行分隔。这里的正则表达式用斜线来界定；这只是另

一种形式的引用符号，这种形式的引用专门用于正则表达式，但和双引号的解释很类似。

我们同样也可以写为 `split ':'` 或者 `split ":"`。

这个 `split` 要用冒号分割的字符串到底是什么，没有明确地指定。当 `split` 没有第二个参数的时候，Perl 就认为要分割 `$_` 的值。说实话，即使这个匹配模式（即用冒号做分隔符）也是可有可无的；默认用空白做分隔符，但却会忽略开头的所有空白。

不过稍等一下，还要多说一点。甚至回到循环一开始的地方，给 `$_` 赋值的操作也是不必要的。如果简单地写

```
while (<>) {
```

那么 Perl 会自动把每行都保存在 `$_` 里。用户不必明确地去访问保存输入行的变量，就可以处理这些行。把 `$_` 用作默认操作数的做法很常见，只要在 `$_` 或多或少有意义的地方，Perl 都会允许用 `$_`。

在获得 **passwd** 文件内各个域值的多重赋值语句里

```
($name, $pw, $uid, $gid, $gecos, $path, $sh) = split /:/;
```

等式左边出现的列表创建了 `split` 的“列表上下文”，告诉 `split` 返回结果是由所有的域构成的一个列表。如果是给一个标量赋值，例如，

```
$n_fields = split /:/;
```

`split` 则运行在“标量上下文”里，只返回它找到的域的个数。通过使用 `wantarray` 函数，用户自己编写的函数也能区分标量上下文和列表上下文。该函数在列表上下文中返回一个真值，而在标量上下文中返回一个假值，在空 (`void`) 上下文里则返回一个不定值。

下面这行代码

```
%uids_by_name = reverse %names_by_uid;
```

也有一些深意。列表上下文里的哈希（这里是 `reverse` 函数的一个参数）算成(`key1, value1, key2, value2, ...`)形式的一个列表。`reverse` 函数颠倒该列表的次序，得到(`valueN, keyN, ..., value1, key1`)。最后，给哈希变量`%uids_by_name`赋值的时候把这个列表按(`key1, value1, ...`)这样转换，因此得到了颠倒的索引。

2.4.7 引用和自动生成

虽然这两方面都是高级话题，但是如果我不提一下它们的话，那就是我们的工作怠慢了。这里给出它们的简短总结。数组和哈希只保存标量，但用户经常想在其中再保存别的数组和哈希。例如，回到我们前面分析 **/etc/passwd** 文件的那个例子，用户可能想要把 **passwd** 文件里每一行所有的域都保存到一个用 UID 来索引的哈希结构里。

虽然不能直接保存数组和哈希，但是却可以保存对数组和哈希的引用，因为引用本身是标量。只要在变量名之前加上一个反斜线（例如，`\@array`），或是用引用数组或引用哈希的语法，就可以建立对数组或者哈希的引用。例如，分析口令的循环就可以变成下面的样子：

```
while (<>) {
    $array_ref = [ split ':/' ];
    $passwd_by_uid[$array_ref->[2]] = $array_ref;
}
```

尖括号返回对一个数组的引用，这个数组里保存有分割后的结果。`$array_ref->[2]` 的记法引用 UID 域，即`$array_ref` 所指数组中的第三个成员。

这里不能用`$array_ref[2]`，因为我们没有定义`@array_ref` 这个数组；`$array_ref` 和

`@array_ref` 是不同的变量。再进一步说，如果在这里错误地用了`$array_ref[2]`，那么也不会得到出错消息，因为`@array_ref` 是一个完全合法的数组名；只是不能给它赋值而已。缺少报警消息似乎是个问题，但它可以说是 Perl 最好的特性之一，这个特性叫做“自动生成（autovivification）”。因为变量名和引用语法总是会清楚地表明要访问的数据结构，所以就不用手工随时创建任何数据结构了。只要进行最低可能层面的赋值操作，就会自动生成中间结构。例如，只用一次赋值操作，就可以创建一个指向数组的哈希结构，数组的内容又是指向哈希的引用。

2.4.8 Perl 语言里的正则表达式

用`=~`操作符把字符串“绑定”到正则表达式上，就可以在 Perl 里使用正则表达式了。例如，下面这一行代码

```
if ($text =~ m/ab+c/) {
```

检查保存在`$text` 里的字符串，看是否能够匹配正则表达式 `ab+c`。要对默认字符串`$_`进行操作，只要省略掉变量名和绑定操作符即可。实际上，还可以省去 `m`，因为默认就是执行匹配操作：

```
if (/ab+c/) {
```

替换操作也和匹配操作类似：

```
$text =~ s/etc\./and so on/g;      # Substitute text in $text, OR
s/etc\./and so on/g;              # Apply to $_
```

我们插入了一个 `g` 选项，用“`and so on`”替换所有出现的“`etc.`”，而不是只替换第一次出现

的“etc.”。其他常见的选项有忽略大小写的 i、用点号 (.) 匹配换行的 s，还有 m，即用 ^ 和 \$ 匹配各行行首和行尾，而不只是要搜索的文本的开头和结尾。

下面这个脚本演示了另外两个要点：

```
#!/usr/bin/perl  
  
$names = "huey dewey louie";  
$regex = '(\w+)\s+(\w+)\s+(\w+)';  
  
if ($names =~ m/$regex/) {  
    print "1st name is $1\n2nd name is $2.\n3rd name is $3.\n";  
    $names =~ s/$regex/\2 \1/;  
    print "New names are \"${names}\"\n";  
} else {  
    print qq["$names" did not match "$regex".\n];  
}
```

该脚本的输出为：

```
$ perl testregex  
1st name is huey.  
2nd name is dewey.  
3rd name is louie.  
New names are "dewey huey".
```

这个例子展示了由 // 括起来的变量怎样做扩展，这样一来，正则表达式就不必是一个固定的字符串了。qq 是双引号操作符的另一种写法。

在执行一次匹配或者替换操作之后，\$1、\$2 等变量的内容就和正则表达式里括号中捕获的内容相对应。这些变量的内容在做替换操作时也能用，此时用 \1、\2 等来引用它们。

2.4.9 输入和输出

打开一个文件执行读或者写操作时，就要定义一个“文件句柄”来标识这个通道。在下面的例子里，INFILE 是 /etc/passwd 的文件句柄，而 OUTFILE 是关联到 /tmp/passwd 的文件句柄。while 语句的循环条件是 <INFILE>，它和我们已经见过的 <> 很相似，但它专

指一个文件句柄。这条语句读取文件句柄 `INFILE` 中的每一行，直到文件结尾，这时 `while` 循环结束。每行的内容都放入 `$_` 变量。

```
#!/usr/bin/perl  
open(INFILE, "</etc/passwd") or die "Couldn't open /etc/passwd";  
open(OUTFILE, ">/tmp/passwd") or die "Couldn't open /tmp/passwd";  
  
while (<INFILE>) {  
    ($name, $pw, $uid, $gid, $gecos, $path, $sh) = split '/';  
    print OUTFILE "$uid\t$name\n";  
}
```

如果成功打开了这个文件，则 `open` 返回一个真值，从而绕过对 `die` 子句的判断（让这个子句不必执行）。Perl 的 `or` 操作符和 `||`（Perl 也有这个操作符）的作用很像，但是优先级更低。如果要特意先对操作符左边的全部内容做判断，然后 Perl 才关注失败的结果，那么采用操作符 `or` 一般是更好的选择。

Perl 用来指定如何使用每个文件（读？写？追加？）的语法和 shell 一模一样。还可以用像 `"/bin/df |"` 这样的“文件名”来打开和 shell 命令联系的管道。

2.4.10 控制流程

下面的例子用 Perl 重新实现了一个 **bash** 脚本，我们早先用后者来验证命令行参数是否有效。读者可以参照 2.2.3 节里该脚本的 **bash** 版本。注意，Perl 的 `if` 语句结构没有 `then` 这个关键字，也没有终结关键字，它只是一个用花括号括起来的语句块。

还可以在单个语句的后面追加 `if` 子句（或者 `unless` 子句，它是 `if` 子句的否定版本），使得该语句有条件地执行。

```

#!/usr/bin/perl
sub show_usage {
    print shift, "\n" if scalar(@_);
    print "Usage: $0 source_dir dest_dir\n";
    exit scalar(@_) ? shift : 1;
}
if (@ARGV != 2) {
    show_usage;
} else { # There are two arguments
    ($source_dir, $dest_dir) = @ARGV;
    show_usage "Invalid source directory" unless -d $source_dir;
    -d $dest_dir or show_usage "Invalid destination directory";
}

```

在这个例子中，有两行代码使用了 Perl 的单目操作符 `-d`，用来判断 `$source_dir` 和 `$dest_dir` 是否为目录。第二种形式（`-d` 操作符在行首）有优势，它把实际的判断语句放在行首，这个位置最显眼。不过，用 `or` 来表示“否则”的意思有点儿难懂；有些读到这一代码的人可能会发现它容易让人误会。

按标量上下文（本例中由标量运算符来指出）取数组变量的值，返回的是该数组内元素的个数。这个值比 `$#array` 的值正好多 1；在 Perl 里，有不止一种方法可以得到这个值。Perl 的函数从名为 `@_` 的数组里获得它们的参数。用 `shift` 操作符是访问这些参数最常用的方法，`shift` 操作符删除参数数组里的第一个元素，并返回它的值。

这个版本的 `show_usage` 函数接受一则要打印的出错消息，但也可以不提供这个出错消息。如果提供了一则出错消息，那么还可以提供一个特殊的退出码。三目操作符 `?:` 计算其中第一个参数的值；如果值为真，那么返回结果就是第二个参数；否则就返回第三个参数。和 **bash** 里一样，Perl 也有一种专门的“`else if`”条件，但是它的关键字是 `elsif` 而不是 `elif`（对于 **bash** 和 Perl 两种语言都用的人来说，这些有意思的小差别要么让人头脑聪颖，要么让人抓狂）。

如表 2.5 所示，Perl 的比较运算符正好和 **bash** 的相反；字符串比较用文本运算符，而数值比较用传统的代数表达式。读者可以和 2.2.7 节的表 2.2 进行比较。

表 2.5

Perl 的基本比较运算符

字符串	数值	为真，如果
x eq y	x = y	x 等于 y
x ne y	x != y	x 不等于 y
x lt y	x < y	x 小于 y
x le y	x <= y	x 小于等于 y

续表

字符串	数值	为真，如果
x gt y	x > y	x 大于 y
x ge y	x >= y	x 大于等于 y

在 Perl 里，可以使用表 2.3 中除 -nt 和 -ot 之外的所有文件测试操作符，-nt 和 -ot 只有 **bash** 才支持。

和 **bash** 一样，Perl 也有两种类型的循环。比较常见的循环形式是通过一个明确的参数列表。例如，下面的代码通过一个动物的列表来循环，每行打印一个动物的名字。

```
@animals = qw(lions tigers bears);
foreach $animal (@animals) {
    print "$animal \n";
}
```

也可以采用更传统的 C 风格的循环：

```
for ($counter=1; $counter <= 10; $counter++) {
    printf "%d", $counter;
}
```

我们给出了传统的 **for** 和 **foreach** 两种写法，但实际上，它们两个在 Perl 里是相同的关键字，可以根据自己的偏好选用其中任何一个。

在 Perl 5.10 (2007 年) 版之前都没有明确的 case 或者 switch 语句，但可以用几种办法取得相同的效果。用多层嵌套的 if 语句显然是一种办法，但除了这种不太好的做法之外，另一种可能的方法是用一条 for 语句设置 \$_ 的值，然后提供一种上下文，让 last 语句可以从 for 语句块里立即跳出：

```
for ($ARGV[0]) {
    m/^websphere/ && do { print "Install for websphere\n"; last; };
    m/^tomcat/ && do { print "Install for tomcat\n"; last; };
    m/^geronimo/ && do { print "Install for geronimo\n"; last; };
    print "Invalid option supplied.\n";
    exit 1;
}
```

用多个正则表达式和 \$_ 里保存的参数进行比较。匹配不成功的话，就绕过 && 并直接落入下一个测试条件。只要匹配了一个正则表达式，那么就执行相应的 do 语句块。然后由 last 语句从 for 语句块里立即跳出。

2.4.11 接受和确认输入

下面的脚本把我们在前面碰到的许多 Perl 结构都结合了起来，包括例程（函数）、后缀形式的 if 语句，以及 for 循环。这个程序本身只是主函数 get_string 的一个封装程序，get_string 是一个检查输入有效性的通用例程。这个例程提示输入一个字符串，删除结尾的所有换行符，然后核对该字符串是否为空。空字符串会提示重新输入，三次之后脚本就放弃尝试。

```
#!/usr/bin/perl
$maxatt = 3; # Maximum tries to supply valid input
sub get_string {
    my ($prompt, $response) = shift;
    # Try to read input up to $maxatt times
    for (my $attempts = 0; $attempts < $maxatt; $attempts++) {
        print "$prompt ";
        $response = readline("STDIN");
        chomp($response);
        return $response if $response;
    }
    die "Too many failed input attempts";
}
```

```

#!/usr/bin/perl
$maxatt = 3; # Maximum tries to supply valid input
sub get_string {
    my ($prompt, $response) = shift;
    # Try to read input up to $maxatt times
    for (my $attempts = 0; $attempts < $maxatt; $attempts++) {
        print "Please try again.\n" if $attempts;
        print "$prompt ";
        $response = readline(*STDIN);
        chomp($response);
        return $response if $response;
    }
    die "Too many failed input attempts";
}

# Get names with get_string and convert to uppercase
$fname = uc get_string "First name";
$lname = uc get_string "Last name";
printf "Whole name: $fname $lname\n";

```

这个脚本的输出为：

```

$ perl validate
First name: John Ball
Last name: Park
Whole name: JOHN BALL PARK

```

在函数 `get_string` 和 `for` 循环中，都用 `my` 操作符创建有局部作用域的变量。在默认情况下，Perl 中的变量都是全局变量。

对 `get_string` 里的局部变量列表进行初始化的时候，只从该函数的参数数组中获得了一个标量。对于初始化列表中的变量，如果没有获得相应的值（本例中是 `$response`），那么就保持未定义的状态。

传给函数 `readline` 的 `*STDIN` 是一个“类型通配（typeglob）”，这在语言设计上是一个让人讨厌的缺点。最好不要太深入地去搞清楚它的确切含义，以免搞得自己头大。简单解释说，就是 Perl 的文件句柄不是一流的数据类型，所以一般必须在它们的名字之前加一个星号，才能当做参数传给函数。

在给 `$fname` 和 `$lname` 赋值的语句里，`uc`（代表“convert to uppercase”，即转为大写）和 `get_string` 两个函数调用都不带括号。因为在一条语句里不可能出现混淆，所以这样写没问题。

2.4.12 Perl 用作过滤器

不通过脚本也能用 Perl，只要在命令行写单独的表达式就行。这是做快速文本转换的一种好方法，这种方法几乎取代了比较老的过滤器程序，如 **sed**、**awk** 和 **tr**。

Use the **-pe** command-line option to loop through STDIN, run a simple expression on each line, and print the result. For example, the command

用命令行选项 **-pe** 可以对 STDIN 循环处理，对每行做一次简单匹配，然后打印结果。例如，

下面这条命令

```
...
ubuntu$ perl -pe 's#/bin/sh#/bin/bash#' /etc/passwd
root:x:0:0:root:/root/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/bash
...
```

把 **/etc/passwd** 里每行末尾的 **/bin/sh** 替换为 **/bin/bash**，然后把转换后的 **passwd** 文件送到 STDOUT。看到用斜线作为定界符（例如，**s/foo/bar/**），配合文本替换操作符一起使用，这种形式对读者来说可能更习惯，但是 Perl 允许用任何字符。在这里，搜索文本和替换文本都带有反斜线，所以用 # 作为定界符更简单。如果用成对的定界符，那么必须用 4 个，而不是通常的 3 个，例如，**s(foo)(bar)**。

Perl 的 **-a** 选项打开了自动分隔模式，这种模式把输入行分成若干个域，保存在名为 @F 的数组里。默认的域分隔符是空白，但可以用 **-F** 选项设置另一种分隔符模式。

自动分隔模式和 **-p** 或者 **-n** (**-p** 的变体，它表示不自动打印) 配合使用很方便。例如，下面的前两条命令用 **perl -ane** 把 **df** 两种不同形式的输出进行分隔。第三行命令接着执行 **join** 操作，把两组结果按 Filesystem 域拼接到一起，生成的组合表里包括两个 **df** 输出版本中的所有域。

```

suse$ df -h | perl -ane 'print join("\t", @F[0..4]), "\n" > tmp1
suse$ df -i | perl -ane 'print join("\t", @F[0,1,4]), "\n" > tmp2
suse$ join tmp1 tmp2
Filesystem  Size  Used   Avail   Use%  Inodes  IUse%
/dev/hda3    3.0G  1.9G   931M   68%  393216  27%
udev        126M  172K   126M   1%   32086   2%
/dev/hda1    92M   26M    61M   30%  24096   1%
/dev/hda6   479M   8.1M   446M   2%   126976  1%
...

```

不用临时文件的脚本如下：

```

#!/usr/bin/perl
for (split(/\n/, `df -h`)) {
    @F = split;
    $h_part[$F[0]] = [ @F[0..4] ];
}
for (split(/\n/, `df -i`)) {
    @F = split;
    print join("\t", @{$h_part[$F[0]}], $F[1], $F[4]), "\n";
}

```

真正厉害的地方在于，可以把**-i** 和**-pe** 连起来用，就地编辑文件；Perl 把文件读进来，提供文件里要编辑的那些行，然后再把结果保存回原来的文件。可以给**-i** 提供一个模式，告诉 Perl 怎样备份每个文件原来的版本。例如，**-i.bak** 把 **passwd** 文件备份为 **passwd.bak**。要小心——如果没有提供一个备份模式，那么就根本得不到备份了。注意，在**-i** 和后缀名之间没有空格。

2.4.13 Perl 的附加模块

位于 cpan.org 的 CPAN (Comprehensive Perl Archive Network , 综合 Perl 存档网络) 是一个 Perl 语言的大宝库，里面有用户贡献的第三方库。用 **cpan** 命令安装新的模块非常简单，就像把 **yum** 或者 **APT** 这样的包管理软件用到 Perl 模块上一样。如果在 Linux 系统上，那么检查一下，看该 Linux 发行版本是否把要找的模块已经放在软件包里，成为一个标准的功能——在系统层面只安装一次，然后让系统以后负责该模块的升级，这种做

法更简单。

在没有 **cpan** 命令的系统上，可以运行 **perl -MCPAN -e shell**，从另一条途径达到同样的效果：

```
$ sudo perl -MCPAN -e shell
cpan shell -- CPAN exploration and modules installation (v1.9205)
ReadLine support available (maybe install Bundle::CPAN or Bundle::CPANxxl?)

cpan[1]> install Class::Date
CPAN: Storable loaded ok (v2.18)
CPAN: LWP::UserAgent loaded ok (v5.819)
CPAN: Time::HiRes loaded ok (v1.9711)
... several more pages of status updates ...
```

用户还可以把 Perl 模块安装到自己的主目录里，供个人使用，但是这个过程并不简单。我们推荐采用一种自由的策略，把 CPAN 的第三方模块安装到系统层面上；Perl 社区提供了一个发布模块的中心点，放开代码供人们检查，贡献模块的人都要给出自己的名字。Perl 模块不会比任何别的开源代码更危险。

为了获得更好的性能，许多 Perl 模块都用到了 C 写的组件。安装这样的模块就会涉及去编译这些组件，所以需要一个完整的开发环境，其中要包括 C 编译器和一整套库。

和大多数语言一样，Perl 程序里最常见的错误是，重复实现已经由社区编写的模块所提供的功能²²。养成习惯，在解决任何 Perl 的问题时首先访问 CPAN，这样做会节省开发和调试的时间。

2.5 Python 脚本编程

随着项目变得越来越大、越来越复杂，面向对象的设计和实现所带来的好处，也就变得越

22. Tom Christiansen 评论道，“那不是我的第一选择，但它不失为一种好选择。我认为，程序中最常见的错误是，人们通常从不重写程序。当你写英语作文时，经常会被要求分开交一份初稿，然后再交一份终稿。在编程时这个过程也同样重要。你听到过这样的格言：‘不要交原型。’可偏偏交的就是原型：人们把程序钻研出来，但却从不为了清晰和高效而重写它们。”

来越清楚。Perl 错过了大概 5 年时间，没有提供 OO 特性，虽然它后来又拼命去追赶，但 Perl 版的面向对象编程仍然显得有点儿牵强。

本节介绍 Python 2。Python 3 尚在开发之中，可能在本书没过时之前就能发布。但是和 Perl 6 不一样的是，它看上去更像是一种增量更新。

有着很强 OO 背景的工程师通常会喜欢 Python 和 Ruby，这两种脚本编程语言都有一种明显的 OO 特质。目前，Python 似乎正处于采纳曲线的下降沿，所以对于系统管理来说，它是一种相当流行的工具。包括 OpenSolaris 在内的几种操作系统，主要利用了 Python 的脚本功能。与此相对照，Ruby 仍然主要用于 Web 开发，很少用作一般性的脚本编程。

Guido van Rossum 发明了 Python。与 Perl 相比，Python 的代码更好写，也更好读。Python 提供了一种易于理解的语法，即使没开发过这种代码，也很容易掌握。如果觉得要记住用哪种比较操作符很累人，那么会喜欢上 Python 整齐划一的方式。Python 还提供更多数据类型，有些系统管理员会发现它们很有用。

如果系统上还没装好 Python，可以看看操作系统提供商或者发布方给出的软件包清单。它是一种极其常见的软件包，应该到处都有。如果没找到，可以从 python.org 下载 Python 的源代码。这个地方也是一个集中位置，可以找到其他人开发的附加模块。对于 Python 而言，要想获得比我们在这儿给出的介绍更全面的内容，Mark Pilgrim 的 *Dive Into Python* 是一个非常好的起步点。从 diveintopython.org 可以（免费）阅读或者下载这份文档，也可以购买 Apress 出版的印刷版图书。在 2.7 节里可以找到完整的参考文献。

2.5.1 Python 快速入门

还像往常一样，我们先从一个简单“Hello , world !”脚本开始。果不其然，Python 的“Hello, world!”几乎和 Perl 的一模一样。

```
#!/usr/bin/python
print "Hello, world!"
```

要让这个脚本可以运行，只要设置它的可执行位，扩展直接调用 **python** 来解释这个脚本：

```
$ chmod +x helloworld
$ ./helloworld
Hello, world!
```

这样的一行程序不能体现出 Python 对传统的突破，这一突破恶名在外，也就是说，缩行在逻辑上很重要。Python 不用花括号、方括号，或者 begin 和 end 来界定代码块。处于同一缩进级别的语句自动构成代码块。缩行的风格（空格或者制表符，缩进的深度）则无关紧要。Python 的代码分块最好用例子来展示，所以下面就用一条 if-then-else 语句来

说明：

```
#!/usr/bin/python
import sys
a = sys.argv[1]
if a == "1":
    print 'a is one'
    print 'This is still the then clause of the if statement.'
else:
    print 'a is', a
    print 'This is still the else clause of the if statement.'
print 'This is after the if statement.'
```

第三行导入 sys 模块，它包含数组 argv。then 和 else 子句都有两行，每一部分都缩进到相同的层次。最后的 print 语句在 if 语句之外。和在 Perl 里的情况一样，Python 的 print 语句也接受任意数量的参数。但和 Perl 不一样的是，Python 自动在每对参数之间都插入一个空格，并提供一个换行符。可以在 print 行的末尾多加一个逗号，消除这个换行符；参数为空则告诉 print 不要输出换行符。

在一行末尾的冒号一般是该行引入的一个联系符，它和随后的一个缩进块关联到一起。

```
$ python blockexample 1
a is one
This is still the then clause of the if statement.
This is after the if statement.

$ python blockexample 2
a is 2
This is still the else clause of the if statement.
This is after the if statement.
```

Python 的缩行惯例使得代码格式上的灵活度更低了，但它也有优点，即不同的人所编写的代码看上去都一样，而且还意味着不需要搞得代码里到处都有分号，而这些烦人的分号就只为了结束语句。

Python 里的注释用一个井号 (#) 开头，一直持续到行尾，同 **bash** 和 Perl 里的用法一样。

在代码行用反斜线结尾，就可以把长长的代码行分成多行来写。这样做的时候，只有第一行代码的缩进才重要。不过，如果愿意，可以让后续的代码行都缩进。即使没有出现反斜线，但圆括号、方括号或者花括号不配对的代码行仍会自动触发续行，但如果出现这样的情况，可以在代码里包含反斜线，让代码的结构更清晰。

有些剪切和粘贴操作会把制表符 (tab) 转换为空格，除非知道自己要什么，否则这会让人抓狂。黄金法则是，绝对不要混用制表符和空格；缩进要么用制表符，要么用空格。许多软件采用传统的假设，让制表符应该等于 8 个空格的距离，对于代码的可读性来说，这确实缩进得太厉害了。大多数 Python 用户似乎都偏向于采用空格，而且缩进 4 个字符。

不管决定怎样处理缩行的问题，大多数编辑器都有若干选项，要么只用空白让制表符为非法，要么让空格和制表符显示得不一样，从而帮助用户搞得清楚。在万不得已的情况下，还可以用 **expand** 命令把制表符转为空格，或者通过 **perl -pe** 命令用一个更容易看到的字符串替换制表符。

2.5.2 对象、字符串、数、列表、字典、元组和文件

Python 中所有的数据类型都是对象，比起 Perl 中的数据类型来说，这一点让它们更强大，也更灵活。

在 Python 里，列表用方括号而不是圆括号括起来。数组的索引（下标）从 0 开始，在本章介绍的 3 种脚本编程语言中，这是为数不多的几种没有变化的概念之一。

Python 里新出现了一种叫做“元组（tuple）”的数据类型，它实质上是不能变的列表。元组比数组更快，对于实际上不应该被修改的数据来说，用元组表示更合适。除了用圆括号而不是方括号做定界符之外，元组的语法和列表的一样。因为(thing)看上去是一个简单的代数表达式，因此，对于只包含一个元素的元组，需要用一个额外的逗号来消除它们的含糊意思：(thing,)。

下面是 Python 中出现的几种基本变量和数据类型：

```
#!/usr/bin/python
name = 'Gwen'
rating = 10
characters = [ 'SpongeBob', 'Patrick', 'Squidward' ]
elements = ( 'lithium', 'carbon', 'boron' )

print "name:\t%s\nrating:\t%d" % (name, rating)
print "characters:\t%s" % characters
print "elements:\t%s" % (elements, )
```

这个例子产生的输出如下：

```
$ python objects
name:      Gwen
rating:    10
characters: ['SpongeBob', 'Patrick', 'Squidward']
elements:  ('lithium', 'carbon', 'boron')
```

Python 中的变量不会从语法上体现出来，也不会做类型声明，但是它们所指的对象的确

有一种支持类型。在大多数情况下，Python 不会替用户自动做类型转换，但是单个函数或者操作符可以做转换。例如，不显式地把数值转换为它的字符串表示形式的话，就不能把一个数和一个字符串（用操作符 +）连接起来。不过，格式化操作符和语句会强制把所有东西都转为字符串形式。每个对象都有一个字符串表示。

字符串格式化操作符 % 很像 C 或者 Perl 语言里的 **sprintf** 函数，但它可以用在字符串出现的任何地方。它是一个双目操作符，左边是字符串，右边是要插入的数值。如果要插入一个以上的数值，那么必须用元组来表示这些值。

Python 字典和 Perl 的哈希一样；也就是说，它是“键/值”对的一个列表。字典常量用花括号括起来，每一对“键/值”都用一个冒号分隔。

```
#!/usr/bin/python
ordinal = { 1 : 'first', 2 : 'second', 3 : 'third' }
print "The ordinal array contains", ordinal
print "The ordinal of 1 is", ordinal[1]
```

在使用上，Python 的字典又非常像数组，区别在于下标（键）可以是对象而不只是整数。

```
$ python dictionary
The ordinal array contains {1: 'first', 2: 'second', 3: 'third'}
The ordinal of 1 is first
```

Python 用对象所关联的方法把打开的文件按对象来处理。顾名思义，**readline** 方法读取一行，所以下面的例子从 /etc/passwd 文件读取并打印两行内容。

```
#!/usr/bin/python
f = open('/etc/passwd', 'r')
print f.readline(),
print f.readline(),
f.close()

$ python fileio
at:x:25:Batch jobs daemon:/var/spool/atjobs:/bin>true
bin:x:1:bin:/bin:/bin>true
```

`print` 语句里最后的逗号消除了换行符，因为每行在从原来的文件读入的时候就已经带一个换行符了。

2.5.3 确认输入的例子

下面的脚本片段是 Python 版的确认输入的程序，我们现在已经很熟悉这个例子了。它展示了子例程和命令行参数的用法，还体现了其他两种 Python 化的特性。

```
#!/usr/bin/python

import sys
import os

def show_usage(message, code = 1):
    print message
    print "%s: source_dir dest_dir" % sys.argv[0]
    sys.exit(code)

if len(sys.argv) != 3:
    show_usage("2 arguments required; you supplied %d" % (len(sys.argv) - 1))
elif not os.path.isdir(sys.argv[1]):
    show_usage("Invalid source directory")
elif not os.path.isdir(sys.argv[2]):
    show_usage("Invalid destination directory")

source, dest = sys.argv[1:3]

print "Source Directory is", source
print "Destination Directory is", dest
```

除了导入 `SYS` 模块之外，我们还导入了 `OS` 模块，从而可以使用 `os.path.isdir` 这个例程。

注意，对于模块定义的任何代号来说，`import` 命令不会提供访问它们的捷径；必须使用从模块名开始的全名。

例程 `show_usage` 的定义里给退出码赋予了一个默认值，万一调用程序没有显式地指定这个参数，就用这个默认值退出。既然所有的数据类型都是对象，所以用引用来给函数传参。数组 `sys.argv` 在第 0 个位置保存有该脚本的名字，所以它的长度比实际提供的命令行参数个数正好多 1。`sys.argv[1:3]` 这样的形式表示一个数组段。有意思的是，数组段不包括

指定范围里最后的那个元素，所以这个数组段只有 `sys.argv[1]` 和 `sys.argv[2]` 两个元素。

可以简单地用 `sys.argv[1:]` 把从第二个开始的所有元素都包括进来。

同 **bash** 和 Perl 一样，Python 也有专门的一种“`else if`”条件；其关键字是 `elif`。Python 也没有 `case` 或者 `switch` 语句。

给 `soure` 和 `dest` 变量的平行赋值与 Perl 有点儿不一样，因为这两个变量本身不在一个列表里。两种形式的平行赋值在 Python 里都可以。

Python 给数值和字符串的比较运算符都一样。“不相等”的比较运算符是 `!=`，但却没有 `!` 这样的单目运算符；这样的单目运算符是 `not`。也要搞清楚布尔运算符 `and` 和 `or`。

2.5.4 循环

下面的代码片段用一个 `for...in` 结构从 1 循环到 10。

```
for counter in range(1, 10):
    print counter,
```

和前面例子里的数组段一样，这个范围的右端点实际上没有包括进来。输出值只有 1 到 9。

1 2 3 4 5 6 7 8 9

这是 Python 里唯一的 `for` 循环类型，但它功能很强大。Python 的 `for` 语句有几项特性，有别于其他语言。

- 数值范围没有什么特殊之处。任何对象都可以支持 Python 的循环模型，而且最常见
- 的对象都支持。可以通过一个字符串（按逐个字符）、一个列表、一个文件（按逐个字符、逐行或者逐块），以及一个数组段等。

- 循环可以产生多个值，循环变量也可以有多个。在每次循环的开头进行的赋值，就和 Python 正常的多重赋值一样。
- for 和 while 循环都可以在末尾加上 else 子句。只有当循环正常终止之后，才执行这个 else 子句，这和通过一条 break 语句退出正好相反。这一功能乍看起来似乎与直觉相反，但它能很好地处理某些用例。

下面的脚本示例接受命令行上的一个正则表达式，把它同一个列表进行匹配，列表里是白雪公主中七个小矮人的名字及其衣服的颜色。该脚本打印第一个匹配的结果，而且匹配正则表达式的部分两边用下划线区分出来。

```
#!/usr/bin/python

import sys
import re

suits = { 'Bashful':'red', 'Sneezy':'green', 'Doc':'blue', 'Dopey':'orange',
          'Grumpy':'yellow', 'Happy':'taupe', 'Sleepy':'puce' }
pattern = re.compile("(%s)" % sys.argv[1])

for dwarf, color in suits.items():
    if pattern.search(dwarf) or pattern.search(color):
        print "%s's dwarf suit is %s." % \
              (pattern.sub(r"\1", dwarf), pattern.sub(r"\1", color))
        break
else:
    print "No dwarves or dwarf suits matched the pattern."
```

下面是一些输出的例子：

```
$ python dwarfsearch '[aeiou]{2}'
Sn_ee_zy's dwarf suit is gr_ee_n.

$ python dwarfsearch go
No dwarves or dwarf suits matched the pattern.
```

给 suits 赋值的语句，展示了 Python 用于字典常量的编码方式。suits.items()方法是“键/值”对的迭代器——每次循环都提取一个小矮人的名字和一种衣服颜色。如果只想通过键做循环，只需要把代码写为 for dwarf in suits。

Python 通过它的 re 模块实现对正则表达式的处理。Python 语言本身没有任何有关正则表达式的功能，所以用 Python 处理正则表达式比用 Perl 要稍微麻烦一点儿。在本例中，

正则表达式的 pattern —开始由 `compile` 方法，用圆括号括起来第一个命令行参数进行编译，形成了一个捕获组。接着，用正则表达式对象的 `search` 和 `sub` 方法测试和修改字符串。还可以像函数那样直接调用 `re.search` 等方法，把正则表达式当做第一个参数来用。替换字符串里的`\1` 是反过来引用第一个捕获组的内容。

2.6 脚本编程的最佳实践

虽然本章里的代码片段几乎不带注释，而且很少打印用法说明，只是因为我们已经列出了每个例子的大纲，从而体现出若干关键点。实际的脚本应该有更好的表现。有几本书通篇就讲编码的最佳实践，不过其中的基本指导原则如下。

- 如果运行脚本时带了不合适的参数，脚本应该打印一则用法说明，然后再退出。更好的做法是，也以这样的方式实现`--help` 选项。
- 验证输入的有效性，并检查获得的输入值。例如，在对算出来的一个路径执行 `rm -fr` 操作之前，可能要让脚本复查这条路径是否符合期望的模式。此时会发现脚本编程语言的“污点（taint）”功能很有帮助。
- 返回一个恰当的退出码：`0` 表示成功，非 `0` 表示不成功。但是，没有必要非要给每种不成功的模式分配一个唯一的退出码，考虑调用程序实际想要的是什么。
- 用适当的命名约定来给变量、脚本以及函数起名字。名字要符合该语言的惯例，符合代码库中大部分代码的习惯，最重要的是，符合当前项目里定义的其他变量和函数的惯例。用大小写或者下划线来让长名字可读性更好²³。
- 用变量名反映变量保存的值，但要保持简洁。`number_of_lines_of_input` 这样就太长了；

23. 给脚本本身起名字也很重要。在这种情况下，对于模仿空格而言，短划线比下划线更常见，如 `system-config-printer`。

试试用 `n_lines`。

- 考虑形成一种指导风格，这样一来，你和你的团队成员都可以按照相同的规范来编写代码。有了这样的指导，在阅读别人写的代码，或者别人阅读你写的代码时，都会变得更容易。
- 每个脚本开头有一段注释，说明该脚本的作用以及它接受的参数。注释里要包括作者的名字和编写日期。如果这个脚本需要在系统上安装非标准的工具、库或者模块，那么也要把它们列出来。
- 注释要达到的程度是，过一两个月再来看这个脚本，发现注释很有帮助就行了。有关注释的一些要点如下：选择的算法、没有按显然更好的方式去做的理由、代码里不常见的路径，以及在开发期间成为障碍的任何东西。不要到处乱写没用的注释；要假定阅读代码的人不傻，而且熟悉语言。
- 最好做到代码块级或者函数级的注释。对变量功能的注释说明应该出现在变量声明或者首次使用变量的地方。
- 以 `root` 身份运行脚本是可以的，但要避免设置这些脚本的 `setuid` 位；保证 `setuid` 的脚本彻底安全相当费事儿。所以代之以用 `sudo` 来实现适当的访问控制策略。
- 对于 `bash` 来说，在执行命令之前，先用 `-x` 回显命令，用 `-n` 检查命令的语法，却不执行它们。
- Perl 的 `-w` 选项可以把某些可疑行为报告给用户，如变量未设置就先使用这样的问题。可以把这个选项加到脚本的“`#!`”一行里，或者用 `use warnings` 打开该程序的文字提示。
- 在 Python 中，除非在命令行用 `-0` 参数显式地关闭 `debug`（调试）模式，否则就在这个模式下。这意味着，可以在打印诊断输出之前，先测试特殊的 `__debug__` 变量。

对于产生有用的出错消息而言，Tom Christiansen 提出了下面 5 条黄金法则：

- 出错消息应该送到 STDERR 而不是 STDOUT；
- 包括发布该出错消息的程序名；
- 说明什么函数或者操作未成功；
- 如果一次系统调用失败，那么就要包括 perror 这个字符串（在 Perl 里是\$!）；
- 用 0 之外的其他一些出错码退出。

Perl 可以轻易遵守所有 5 条法则：

```
die "can't open $filename: $!";
```

2.7 推荐读物

BROOKS, FREDERICK P., JR. *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1995.

Shell 基础知识和 bash 脚本编程

ALBING, CARL, JP VOSSEN, AND CAMERON NEWHAM. *Bash Cookbook*. Sebastopol, CA: O'Reilly Media, 2007.

KERNIGHAN, BRIAN W., AND ROB PIKE. *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

NEWHAM, CAMERON, AND BILL ROSENBLATT. *Learning the bash Shell (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2005.

POWERS, SHELLEY, JERRY PEEK, TIM O'REILLY, AND MIKE LOUKIDES. *Unix PowerTools, (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2002.

正则表达式

FRIEDL, JEFFREY. *Mastering Regular Expressions (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2006.
GOYVAERTS, JAN, AND STEVEN LEVITHAN. *Regular Expressions Cookbook*. Sebastopol, CA: O'Reilly Media, 2009.

Perl 脚本编程

WALL, LARRY, TOM CHRISTIANSEN, AND JON ORWANT. *Programming Perl (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2000.
SCHWARTZ, RANDAL L., TOM PHOENIX, AND BRIAN D FOY. *Learning Perl (5th Edition)*, Sebastopol, CA: O'Reilly Media, 2008.
BLANK-EDELMAN, DAVID. *Automating System Administration with Perl*, Sebastopol, CA: O'Reilly Media, 2009.
CHRISTIANSEN, TOM, AND NATHAN TORKINGTON. *Perl Cookbook (2nd Edition)*. Sebastopol, CA: O'Reilly Media, 2003.

Python 脚本编程

BEAZLEY, DAVID M. *Python Essential Reference (4th Edition)*, Reading, MA: Addison-Wesley, 2009.
GIFT, NOAH, AND JEREMY M. JONES. *Python for Unix and Linux System Administrators*, Sebastopol, CA: O'Reilly Media, 2008.
MARTELLI, ALEX, ANNA MARTELLI RAVENSCROFT, AND DAVID ASCHER. *PythonCookbook (2nd Edition)*, Sebastopol, CA: O'Reilly Media, 2005.
PILGRIM, MARK. *Dive Into Python*. Berkeley, CA: Apress, 2004. 这本书也可以从 web 网站 diveintopython.org 自由获得。

2.8 习题

E2.1 UNIX 允许文件名里有空格。怎样找出名字里有空格的那些文件？如何删除它们？

bash、Perl 和 Python 都能很好地处理文件名中的空格吗？否则的话，

需要预先注意什么特别之处？列出一些编写脚本所适用的法则。

E2.2 编写一个简单的 **bash** 脚本（或者两个脚本），备份和恢复你的系统。

E2.3 编写一个 Perl 或者 Python 脚本，利用正则表达式，分析 **date** 命令生成的

日期格式（例如，Tue Oct 20 18:09:33PDT 2009），判断它是否合法

（例如，2月没有30号，确认时区等）。有现成的库或者模块，可以在

一行代码里做到前面那一点吗？如果能，试述如何安装它，如何重写脚本

来使用它。

E2.4 编写一个脚本，从/etc/passwd 和/etc/group（及其等价的网络数据

库）里列出系统的用户和用户组。对于每个用户而言，打印他们的 UID，

再打印该用户所属的用户组。

E2.5 重写改写 2.4.11 节中 get_string 函数的例子，让它只接受整数。它应该接

受 3 个参数：提示字符串、能接受的整数的下限和上限。

E2.6 在你的环境里找一个没有文档的脚本。阅读该脚本，确保掌握了其中的功能。

给该脚本增加注释，或者给该脚本写一个手册页。

★E2.7 编写一个脚本，在屏幕上显示状态数据汇总，状态数据和下面某类有关：

CPU、内存、磁盘或者网络。这个脚本应该利用操作系统的命令和文件，

构造易于理解、包括尽可能多信息的仪表板。

★E2.8 构造一个菜单驱动的界面，让它很方便地选择 **top**、**sar** 或者自己选的性

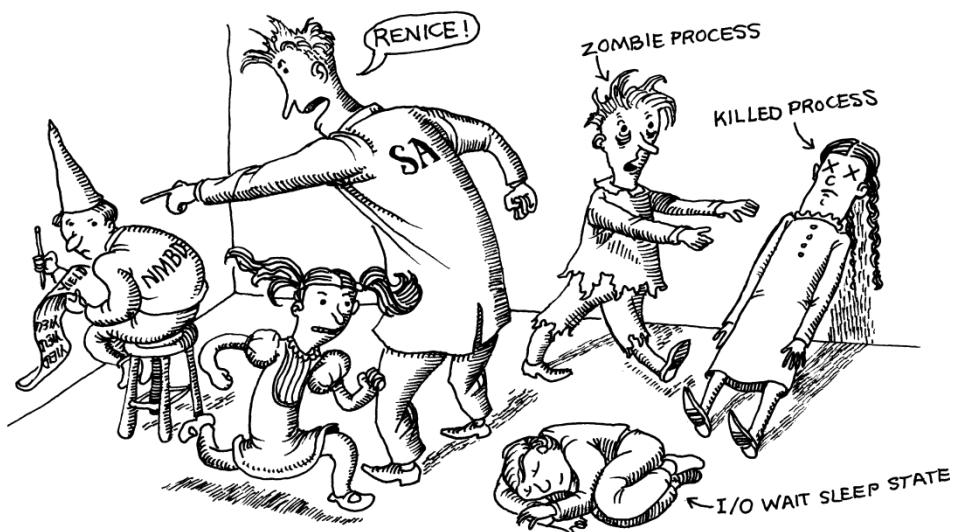
能分析工具的命令行选项。

★E2.9 编写一个脚本，测试一台服务器的网络是否连通，还要测试它所依靠的上游

服务（例如，DNS、文件服务、LDAP 或者其他目录服务）。如果发现了

问题，它会向你发电子邮件或者文本消息吗？

第 5 章 进程控制



进程是 UNIX 和 Linux 用来表示正在运行的程序的一种抽象概念。程序的内存使用、处理器时间和 I/O 资源就是通过这个对象进行管理和监视的。

UNIX 设计思想的一部分内容就是让尽可能多的工作在进程的上下文 (context) 中完成，而不是由内核专门来进行处理。系统进程和用户进程都遵守同样的规则，因此可以用一套工具控制这两种进程。

5.1 进程的组成部分

一个进程由一个地址空间和内核内部的一组数据结构共同组成。地址空间是由内核标记出来供进程使用的一组内存页面²⁴。它包含进程正在执行的代码和库、进程变量、进程栈以及在进程正在运行时内核所需要的各種其他信息。因为 UNIX 和 Linux 都是采用虚拟内存的系统，因此一个内存页面在进程的地址空间中的位置和它在机器的物理内存或交换空间中的位置之间没有关系。

内核的内部数据结构记录了有关每个进程的各种信息，其中非常重要的信息有：

- 进程的地址空间映射；
- 进程的当前状态（睡眠状态、停止状态、可运行状态等）；
- 进程执行的优先级；
- 进程已用资源的信息；
- 进程已打开的文件和网络端口的信息；
- 进程的信号掩码（一个记录，确定要封锁哪些信号）；
- 进程的属主。

一个执行线程（execution thread）通常简称为一个线程（thread），它是在一个进程中执行一次 **fork** 的结果。线程继承了包含它的进程的许多属性（例如，进程的地址空间），多个线程在同一个进程中按照一种称为多线程（multithreading）的机制并发执行。在老式单处理器系统的内核可以模拟并发执行（concurrent execution），而在多核和多 CPU 体系结构上，多个线程可以在不同的核心（core）上同时运行。像 BIND 和 Apache 这样的多线程应用在多核系统上受益最大，因为这些应用能够同时处理多个请求。

24. 页面是管理内存的单位，页面大小通常是 1KiB 和 8KiB。

我们的所有示例操作系统都支持多线程机制。

与进程有关的许多参数直接影响进程的执行：进程所获处理器的时间量、进程能够访问的文件等。在下面各小节中，我们将讨论系统管理员最感兴趣的那些参数的意义及其重要性这些属性对于所有版本的 UNIX 和 Linux 来说都是相同的。

5.1.1 PID：进程的 ID 号

内核给每个进程分配一个独一无二的 ID 号²⁵。控制进程的大多数命令和系统调用需要用户指定 PID 来标识操作的目标。PID 按照创建进程的顺序来分配。

5.1.2 PPID：父 PID

UNIX 和 Linux 都没有提供创建新进程去运行某个特定程序的系统调用，现有进程必须克隆自身去创建一个新进程。克隆出的进程能够把它正在运行的那个程序替换成另一个不同程序。

当一个进程被克隆时，原来的进程就叫做父进程，而克隆出的副本则叫做子进程。进程的 PPID 属性就是克隆它的父进程的 PID²⁶。

当遇到无法辨认（以及可能运行失常）的进程时，父进程 PID 就成了一项很有用的信息。回溯该进程的来源（是一个 shell 还是另一个程序），就能更好地了解它的目的和作用。

25 .

我们的评审人 Jon Corbet 指出，Linux 的 2.6.24 版内核引入了进程 ID 名字空间的概念，能够让 PID 相同的多个进程并发存在。实现这个特性是为了支持基于容器的虚拟化技术。

26 . 至少最初是这样的。如果原来的父进程终止，那么 init（进程 1）就成为新的父进程，请参见 5.3 节。

5.1.3 UID 和 EUID：真实的和有效的用户 ID

进程的 UID 就是其创建者的用户标识号，或者更确切地说，就是复制了父进程的 UID 值。

通常，只允许创建者（也称为属主）和超级用户对进程进行操作。有关 UID 的更多信息，请参考 7.1.3 节。

EUID 是“有效（effective）”的用户 ID，这是一个额外的 UID，用来确定进程在任何给定的时刻对哪些资源和文件具有访问权限。对于大多数进程来说，UID 和 EUID 是一样的，例外的情况是 setuid 程序。

为什么同时采用 UID 和 EUID 呢？这只是因为要保持标识和权限之间的区别，还因为 setuid 程序可能不希望一直以扩大了的权限运行。在大多数系统上，可以设置和重置进程的有效 UID 以便启用或限制它所享有的额外权限。

大多数系统也保留一种“保存 UID（saved UID）”，它是进程刚开始执行时刻，进程的 EUID 的副本。除非进程采取措施删除这个保存下来的 UID，否则它就留了下来，作为真实的或者有效的 UID 来用。因此，编写严谨的 setuid 程序可以让其大部执行操作与它的特殊权限无关，只有在需要额外特权的特定时刻才用上它们。

Linux 还定义了一种非标准的进程参数 FSUID，它控制着对文件系统权限的判断，但在内核之外并不常用，而且不能移植到其他的 UNIX 系统上。

5.1.4 GID 和 EGID：真实的和有效的组 ID

GID 就是进程的组标识号。EGID 与 GID 的关系跟 EUID 与 UID 的关系相同，EGID 也可

以由一个 setgid 程序来“升级”。还有一种保存 (saved) GID，它的目的和保存 UID 一样。

进程的 GID 属性基本没什么用处了。为了规定访问权限，一个进程可以同时是多个组的成员。组的完整列表与 GID 和 EGID 分开保存。判断访问权限一般要考虑 EGID 和补充的组清单，但不考虑 GID。参考 7.1.4 节了解组的更多知识。

只有在一个进程要创建新文件的时候，GID 才会起作用。根据文件系统的权限设定情况，新文件可能要采用创建该文件的进程的 GID。参考 6.5.3 节的内容了解详情。

5.1.5 谦让度

进程的调度优先级决定了它所接收到的 CPU 时间有多少。内核使用动态算法来计算优先级：

它考虑一个进程近来已经消耗 CPU 的时间量以及考虑该进程已经等待运行的时间等因素。

内核还会关注为管理目的而设置的值，这种值通常叫做“谦让值 (nice value) ”或“谦让度 (niceness) ”，之所以这么叫是因为它表明了管理员计划对待系统其他用户的友好程度。

我们将在 4.6 节中详细介绍这个主题。



为了给低延迟的应用提供更好的支持，Linux 向传统的 UNIX 调度模型中增加了“调度类 (scheduling class) ”的概念。目前有三种调度类，每个进程都属于一种调度类。遗憾的是，实时类 (real-time class) 既没有得到广泛使用，命令行对它的支持也不好。系统进程都使用传统的（基于谦让度）调度机制，我们在本书只讨论这一种。参考 www.realtimelinuxfoundation.org 了解有关实时调度机制的更多讨论。

5.1.6 控制终端

大多数不是守护进程（daemon）的进程都有一个与自己相关联的控制终端。控制终端决定了标准输入、标准输出和标准错误通道都默认连到哪儿。当用户从 shell 启动一个命令时，他的终端通常就成为该进程的控制终端。控制终端的概念还影响到信号的发布，有关信号的概念将在 5.3 节中讨论。

5.2 进程的生命周期

为了创建一个新进程，一个进程就会用系统调用 **fork** 来复制自身。**fork** 创建原进程的一个副本，这个副本与父进程大致相同。新进程拥有一个不同的 PID 和它自己的记账信息。

fork 具有一个独一无二的特性，它能够返回两个不同值。从子进程的角度来看，它返回 0。另一方面，对于父进程来说，则返回新创建的子进程的 PID。由于这两个进程在其他方面是相同的，所以它们都必须检查返回值来确定它们要扮演什么角色。

在 **fork** 以后，子进程经常使用 **exec** 族系统调用中的一个成员开始执行新的程序²⁷。这些调用能改变进程正在执行的程序正文（text），并把内存段重置为预先定义的初始状态。

exec 各种不同形式的区别仅在于它们采用的不同方法给新程序指定命令行参数和环境。

当系统引导时，内核会自主创建并安装几个进程。其中最知名的就是 **init**，它的进程号总是 1。**init** 负责执行系统的启动脚本，虽然在 UNIX 和 Linux 上完成上述工作的确切方式稍有不同。除了内核创建的那几个进程以外，其他所有进程都是 **init** 的后代。有关 **init** 守护进程的更多信息，请参考第 3 章。

init 在进程管理中还担负着另外一个重要角色。当一个进程执行完毕时，它调用一个名为 **_exit** 的例程来通知内核它已经做好“消亡”的准备了。它提供一个退出码（一个整数）表

²⁷ 实际上，这个家族除了一个成员之外，都是库例程，而不是系统调用。

明它准备退出的原因。按照惯例，0 用来表示正常的或者说“成功”的终止。

在允许进程完全消失以前，内核要求该进程的消亡得到其父进程的确认，父进程是通过调用 **wait** 来确认的。父进程接收到子进程退出码的一个副本（或者是一个通知，在子进程不是自愿退出的情况下，说明子进程被终止的原因），如果父进程愿意，它还可以得到子进程对资源使用情况的一个总结。

如果父进程比子进程的生命期长并能负责地调用 **wait** 清理死亡的进程，那么这种方案没问题。但是，如果父进程比子进程先消亡，那么内核会意识到将来不会有 **wait** 调用处理子进程了，于是就调整子进程使它这个“孤儿”成为 **init** 的子进程。内核要求 **init** 接受这些“孤儿”进程，并在它们消亡的时候执行所需的 **wait** 调用清除它们。

5.3 信号

信号是进程级的中断请求。系统定义了大约 30 种不同种类的信号，使用这些信号的方式可以有以下几种：

- 作为一种通信手段在进程之间发送信号；
- 当键入特殊的按键时，例如<Control-C>和<Control-Z>，可以由终端驱动程序发送信号去终止、中断或挂起进程²⁸；
- 可以由管理员（使用 **kill**）发送信号来达到各种结果；
- 当进程出错，例如出现“用零作除数”的错误时，可以由内核发送信号；
- 可以由内核发送信号，通知一个进程有某种“受关注的”条件出现，比如一个子进程死亡或者在 I/O 通道上有数据。

28. 可以使用 **stty** 命令将<Control-Z>和<Control-C>的功能重新分配给其他键，但实际上很少有人这样做。在本章中，我们提到它们的时候，都是用它们习惯上的绑定关系。

当收到信号时，可能发生两种情况之一。如果接收进程已经为这个特定的信号指派了一个信号处理例程，那么就使用和传递该信号的上下文有关的信息来调用这个信号处理例程。否则，内核代表该进程采取某种默认措施。根据信号的不同，采取的默认措施也不同。许多信号会终止进程，有些信号也会产生一次内存转储（core dump）。内核转储是一个进程的内存映像文件。它可用于调试程序错误[译者注：core 代表老式计算机上用作主存储器的磁芯“magnetic core”。现代计算机已经不再使用磁芯存储器了，或许称之为 memory dump 会更恰当些，但是导线 core 一词却沿袭下来，而且人们常常把 core 同 kernel 这个词混淆，以为是内核或核心的同义词，但实际上两者的含义大相径庭]。

在一个程序内为信号指定信号处理例程的做法通常叫做“捕获”该信号。当信号处理例程完成时，执行流程从接收到信号的位置重新开始。

为了防止收到信号，程序可以要求忽略（ignore）信号或者封锁（block）信号。被忽略的信号被简单地丢弃，它对进程没有什么影响。被封锁的信号排队等待发送，但内核不要求进程对该信号采取措施，一直到该信号被明确地解除封锁为止。处理新的不被封锁的信号的信号处理器只被调用一次，即使在封锁接收的同时多次接收到该信号的情况下，也只调用一次。

表 5.1 列出了所有系统管理员都应该知道的信号。信号名称采用大写的习惯源于 C 语言的传统，有时候还可能看到信号名称以 SIG 开头（例如，SIGHUP），也是出于类似原因。

表 5.1 每个系统管理员都应该知道的信号 ^a						
编号	名称	描述	默认	是否能捕获	是否能封锁	是否转储内存映像
1	HUP	挂起	终止	是	是	否
2	INT	中断	终止	是	是	否
3	QUIT	退出	终止	是	是	是

9	KILL	杀死	终止	否	否	否
_b	BUS	总线错误	终止	是	是	是
11	SEGV	段错误	终止	是	是	是
15	TERM	软件终止	终止	是	是	否
_b	STOP	停止	停止	否	否	否
_b	TSTP	键盘停止	停止	是	是	否
_b	CONT	停止以后继续	忽略	是	否	否
_b	WINCH	窗口改变	忽略	是	是	否
_b	USR1	用户定义	终止	是	是	否
_b	USR2	用户定义	终止	是	是	否

a . 执行 **bash** 的内置命令 **kill -l** 也能获得一份信号名称和编号的清单

b . 随系统不同而异 , 请参见 /usr/include/signal.h 或者 man signal 命令了解更详细的信息

还有其他一些信号没有在表 5.1 中列出来 , 其中大多数用来报告一些含糊的错误 , 例如 “illegal instruction (非法指令)”。对这类信号的默认处理方式是终止其执行并产生一个内存转储。一般也可以捕获和封锁这类信号 , 因为有些程序可能够聪明 , 会先尝试清除导致错误的任何问题 , 再继续运行。

BUS 和 SEGV 信号也是错误信号。我们之所以把这两个信号包括在表中 , 是因为这两个信号非常常见 : 一个程序崩溃的次数中有 99% 最终都可以归结为是由这两种错误中的一种导

致的。这两个信号本身并没有特定的诊断值，它们都表明有不正确地使用或访问内存的企业²⁹。

KILL 和 STOP 信号不能够被捕获、封锁或忽略。KILL 信号杀死接收到它的进程。STOP 信号挂起正在执行的进程，一直到该进程接收到 CONT 信号为止。CONT 信号可以被捕获和忽略，但不能够被封锁。

TSTP 信号是“软”版的 STOP 信号，把它描述成一个“停止请求”是最贴切不过的了。当用户在键盘上键入 <Control-Z> 时，就会由终端驱动程序发出这个信号。捕获到这个信号的程序通常清除它们的状态，然后给自己发送一个 STOP 信号来完成停止操作。另外，TSTP 可以被简单地忽略，以防止程序被键盘输入所停止。

终端模拟器在其配置参数（例如虚拟终端中的行数）改变时，都将发送一个 WINCH 信号。这个规定可以让那些理解模拟器的程序（比如文本编辑器）根据相应的变化而自动对自身进行重新配置。如果不能够让窗口正确地改变大小，那么请确认 WINCH 是否正确地产生和发送了³⁰。

信号 KILL、INT、TERM、HUP 和 QUIT 听起来似乎表示同一回事，但实际上它们的使用方法非常不同。遗憾的是，人们竟然为这些信号选择了那样含义模糊的术语，下面是确切的解释。

- KILL 不可以封锁，它在内核的层面上终止进程。进程实际上从来就不能够“接收”到这个信号。

29. 更具体地说，是由于没有满足对齐要求，或者使用无意义的地址所引发的总线错误。“segmentation violation”表明出现了诸如向地址空间的只读部分写数据这样的保护错。

30. 这一点说起来要比做起来容易。终端模拟程序（例如，xterm）、终端驱动程序和用户级命令在出现 SIGWINCH 信号时都可能要负有一定的责任。常见的问题有，只向一个终端的前台进程（而不是所有和这个终端关联的进程）发送信号，没有把终端大小的变化通过网络告诉远端的一台计算机。像 TELNET 和 SSH 这样的协议都能识别本地终端大小的变化，并且把这一信息告诉远端的主机。比较简单的协议（例如，直连的串口线）就做不到。

- INT 是当用户键入<Control-C>时由终端驱动程序发送的信号。这是一个终止当前操作的请求。如果捕获了这个信号，一些简单的程序应该退出，或者只是让自己被杀死，这也是程序没有捕捉到这个信号时的默认处理方法。拥有命令行模式的那些程序应该停止它们正在做的事情，清除状态，并等待用户的再次输入。
- TERM 请求彻底终止某次执行。它期望接收该信号的进程清除自己的状态并退出。
- HUP 有两种常见的解释。第一种，它被许多守护进程理解为一个重置的请求。如果一个守护进程不用重新启动就能够重新读取它自己的配置文件并调整自己以适应变化，那么 HUP 信号通常可以用来触发这种行为。第二种，HUP 信号有时候由终端驱动程序生成，试图用来“清除”（也就是“杀死”）跟某个特定终端相关联的那些进程。例如，当一个终端会话结束时，或者当一个调制解调器连接被不经意地断开（因而得名为“挂断”）时，就可能出现这种情况。
C shell 家族中的 shell (tcsh 等) 通常让后台程序不受 HUP 信号的影响，这样，它们可以在用户注销后继续运行。Bourne 风格 shell (ksh、 bash 等) 的用户可以使用 nohup 命令来模仿这种行为。
- QUIT 与 TERM 类似，不同之处在于，如果没有被捕获的话，它的默认行为是产生一个内存转储。有一些程序借用了它，把这个信号解释为其他意思。
信号 USR1 和 USR2 没有设定意义。程序可以按照自己需要的方式，任意使用这两个信号，例如，Web 服务器 Apache 把信号 USR1 解释为一次要求妥善重启的请求。

5.4 kill: 发送信号

顾名思义，kill 命令最常见的用法是终止一个进程。kill 能够发送任何信号，但在默认情

况下，它发送一个 TERM 信号。**kill** 可以被普通用户用在他们自己的进程上，或者被超级用户用在任何进程上。语法是：

```
kill[-signal] pid
```

这里的 *signal* 就是要发送信号的编号或符号名称（如表 5.1 所示），*pid* 就是目标进程的进程标识号。*pid* 为 -1 会把这个信号广播给除了 **init** 以外的所有进程。

没有带信号编号的 **kill** 命令不保证进程会被杀死，因为 TERM 信号可能被捕获、封锁或忽略。下面的命令：

```
kill -9 pid
```

将“保证”进程的消亡，因为信号 9，即 KILL 不能够被捕获到。我们给“保证”加引号是因为进程的生命力有时候能够变得相当“旺盛”，以致于连 KILL 也不能够影响到它们（通常是由有些退化的 I/O 虚假锁定，例如等待已经停止旋转的磁盘）。重新启动系统通常是解决这些“不听话”的进程的唯一方法。

killall 命令在 UNIX 和 Linux 上的功能差别很大。在 Linux 上，按名字杀死所有进程。例如，下面的命令杀死所有的 Apache Web 服务器进程：

```
ubuntu$ sudo killall httpd
```

Solairs、HP-UX 和 AIX 自带的标准 UNIX **killall** 命令不带参数，只杀死当前用户的所有进程。以 root 身份执行这条命令会杀死 **init** 并关机。

Solaris、HP-UX 和 Linux（但不包括 AIX）上的 **pgrep** 和 **kill** 命令按名字（或者其他属性，例如 EUID）查找进程，显示这些进程，或者向它们发信号。例如，下面这条命令向所有以用户 ben 身份运行的进程发一个 TERM 信号：

```
$sudo kill -u ben
```

5.5 进程的状态

进程不会仅仅因为其存在就自动地具有获得 CPU 时间的资格。管理员需要注意 4 种最基本的执行状态，这些状态将在表 5.2 中列出。

表 5.2

进程的状态

状态	意义
Runnable (可运行状态)	进程可以被执行
Sleeping (睡眠状态)	进程正在等待某些资源
Zombie (僵化状态)	进程试图消亡
Stopped (停止状态)	进程被挂起 (不允许执行)

处于可运行状态的进程只要有 CPU 时间可用，就准备执行。处于这种状态的进程已经得到它需要的全部资源，而只是在等待获得 CPU 时间去处理它的数据。一旦进程执行了一个不能够立即完成的系统调用（例如请求读取文件的一部分），内核会把这个进程转入睡眠状态。

处于睡眠状态的进程等待特定的事件发生。交互式的 shell 和系统守护进程把它们的大多数时间都花在睡眠、等待终端输入或网络连接上。由于正在睡眠的进程在其请求被满足之前，都被有效地阻塞，所以，除非它接收到一个信号，或者接收到一次对其 I/O 请求的响应，否则它将得不到 CPU 时间。

有些操作让进程进入到一种不能中断的睡眠状态。这种状态通常是临时性的，从 ps 命令的输出看不到（在 STAT 列用一个 D 来指示，参见 5.4 节）。不过，在个别不对劲儿的情

形下，会让这种状态保持下去。最常见的原因是，以“hard”选项通过 NFS 挂载文件系统时，服务器出现问题。因为处于不能中断睡眠状态的进程甚至不能被唤醒去处理一个信号，那么也就不能去杀死它们。为了清除这样的进程，必须纠正背后的问题，或者重启系统。

僵尸进程是已经执行完毕但还没有让它们的状态被收集起来的进程。如果看到有僵尸进程挂在那里，那么就要用 **ps** 查看它们的 PPID，找出它们的来源。

处于停止状态的进程从管理上来说是被禁止运行的。进程一接收到 STOP 或 TSTP 信号，就进入停止状态，并可以使用 CONT 信号来重新启动处于停止状态的进程。处于停止状态与睡眠状态类似，但除了让另外某个进程来唤醒（或者终止）进程以外，它是不能够脱离停止状态的。

5.6 nice 和 renice : 影响调度优先级

进程的“谦让度”是给内核的数字暗示，通过它来表明一个进程在同其他进程竞争 CPU 时内核应该如何对待这个进程。它的这个奇怪名称源自于这样的事实：它决定了准备对待系统上其他用户的谦让程度。高谦让值表示进程具有低优先级：准备很友好地对待其他进程。低谦让值或者负值表示进程具有高优先级：对其他进程就不那么谦让了。

谦让值的允许范围在不同系统上不一样。最常见的取值范围是 $-20 \sim +19$ 。有些系统使用相同大小的取值范围，但起始值是 0 而不是负数（一般是 $0 \sim 39$ ）。在我们的示例系统上使用的范围如表 5.3 所示。

尽管各个系统的谦让度取值不一样，但是所有的系统都已同样的方式处理谦让值。除非用户采取特殊动作，否则，新创建的进程就从它的父进程那里继承谦让值。进程的属主可以增加其谦让值，但不能够降低谦让值，哪怕是让进程返回到默认谦让值也不行。这种限制防止了具有低优先级的进程派生出高优先级的子进程。超级用户可以任意设置谦让值。

如今已经很少通过手工设置进程的优先级了。在 20 世纪 70、80 年代那些功能并不强大的系统上，影响性能的最重要因素就是在 CPU 上运行的是什么进程。现在，由于每一台台式机的 CPU 处理能力都有富余，所以调度程序通常能够很好地完成服务所有进程的工作。

在需要快速响应的情况下，增加调度类型给开发人员更多的控制能力。

I/O 性能一直跟不上日益快速的 CPU，所以现在大多数系统上的主要瓶颈已经变成了磁盘驱动器的速度问题了。遗憾的是，一个进程的谦让值并不会影响内核对其内存或 I/O 的管理；高谦让度的进程仍然以不合适的比例地独占着部分这类资源。

进程的谦让值可以在创建进程时用 **nice** 命令来设置，并可以在执行时使用 **renice** 命令进行调整。**nice** 带一个命令行作为参数，而 **renice** 带 PID 或者（有时候）带用户名作为

参数。

下面举几个例子：

```
$nice -n 5 ~/bin/longtask      // 把优先级降低（提高谦让度）5
$sudorenice -5 8829          // 把谦让值设为-5
$sudorenice 5 -u boggs       // 把boggs的进程的谦让值设为5
```

遗憾的是，就如何指定所需优先级来说，各种系统之间很不统一；实际上，甚至同一种系统上的 **nice** 和 **renice** 也不统一。有些命令的参数是谦让值的增量，而其他命令的参数是一个绝对的谦让值。有些命令要带标志（**-n**），而有些则要值，不带标志。

C shell 和另外一些常见的 shell 中（但不是 **bash**）内置了一种版本的 **nice**，这让情况变得更复杂。如果不键入 **nice** 命令的完整路径，将得到 **nice** 的 shell 版本而不是操作系统的版本。这可能会令人困惑，因为 shell 的 **nice** 和操作系统的 **nice** 几乎总是使用不同的语法：shell 的 **nice** 要求它的优先级增量用 *+incr* 或者 *-incr* 来表达，而独立的 **nice** 命令则要求用 **-n** 标志，后跟优先级增量³¹。

表 5.3 总结了所有这些变化：*prio* 是一个绝对的谦让值，而 *incr* 是一个相对的优先级增量，要把它加上 shell 的当前优先级，或者从 shell 的当前优先级中减去。无论用 *-incr* 还是 *-prio*，都可以用两个短划线输入负值（例如，*--10*）。只有 shell 的 **nice** 才能理解加号（实际上，它需要加号）；所有别的环境，则不予考虑。

在现代世界中，最常见的使用 **nice** 的进程是 **ntpd**，它是时钟同步守护进程。由于 CPU 的及时性对 **ntpd** 的任务非常重要，因此它通常运行在比默认值低 12 左右的谦让值上（也就是说，它的优先级比普通进程更高）。

³¹ 实际上情况甚至更糟：独立的 **nice** 命令把 **nice -5** 解释成值为正 5 的增量，而 shell 的内置 **nice** 命令会把同一形式解释成值为负 5 的增量。

表 5.3 各种版本的 nice 和 renice 如何表示优先级

系 统	范 围	OS 的 nice	csh 的 nice	renice
Linux	[-20 ~ 19]	-incr 或-nincr	+incr 或-incr	prio
Solaris	0 ~ 39	-incr 或-nincr	+incr 或-incr	incr 或-nincr
HP-UX	0 ~ 39	-prio 或-nprio	+incr 或-incr	-nprio ^a
AIX	[-20 ~ 19]	-incr 或-nincr	+incr 或-incr	-nincr

^a. 使用绝对优先级，但要给指定的值加 20

如果某个进程“发狂”而使得系统的负荷平均值达到 65 的话，在能够运行命令去调查问题以前，可能需要使用 nice 来启动一个高优先级的 shell，否则，可能难以运行哪怕是简单的命令。

5.7 ps : 监视进程

ps 是系统管理员监视进程的主要工具。虽然不同版本的 **ps** 其参数和输出也不同，但它们都提供的信息本质上是一样的。造成各个 **ps** 版本间差别那么大的部分原因可追溯到 UNIX 发展历史的不同上。不过，**ps** 也是一条各个厂商出于其他原因想要做定制的命令。它和内核对进程的处理联系紧密，所以它反映出了某个厂商对下层内核所做的各种改动。

ps 可以显示进程的 PID、UID、优先级和控制终端。它还给出了有关一个进程正在使用多少内存、已经消耗了多少 CPU 时间以及它的当前状态（运行中、已停止、在睡眠等）的信息。在 **ps** 中，僵尸进程显示为<exiting>或<defunct>。

在过去 10 年里，**ps** 的实现变得相当复杂。有几个厂商已经放弃为定义有意义的输出显示而做的努力，让它们的 **ps** 完全可配。只需要做一点定制工作，就能配出几乎任何想要的输出。Linux 使用的 **ps** 就是这方面的一个例子，它一种混合型的 **ps** 版本，能理解多种选

项的集合，而且 Linux 能用一个环境变量告诉它采用哪一种风格。

不要被所有这些复杂性所吓住：复杂主要是对开发人员而言的，而不是对系统管理员而言的。即便会频繁用到 **ps**，也只需知道几条特定的用法就够了。



在 Linux 和 AIX 上，可以用命令 **ps aux** 了解正在系统上运行的所有进程的全貌。选项 **a** 的意思是显示所有的进程，**x** 的意思是显示没有控制终端的进程，而 **u** 的意思是选择“面向用户”的输出格式。下面举个在一台运行 RedHat 的机器上 **ps aux** 输出的例子（相同命令在 AIX 上的输出略有不同）：

```
redhat$ ps aux
USER   PID %CPU%MEM   VSZ   RSS   TTY STAT TIME COMMAND
root     1  0.1  0.2  3356  560 ? S 0:00 init [5]
root     2  0  0  0  0 ? SN 0:00 [ksoftirqd/0]
root     3  0  0  0  0 ? S< 0:00 [events/0]
root     4  0  0  0  0 ? S< 0:00 [khelper]
root     5  0  0  0  0 ? S< 0:00 [kacpid]
root    18  0  0  0  0 ? S< 0:00 [kblockd/0]
root    28  0  0  0  0 ? S 0:00 [pdflush]
...
root   196  0  0  0  0 ? S 0:00 [kjournald]
root  1050  0  0.1 2652  448 ? S<s 0:00 udevd
root  1472  0  0.3 3048 1008 ? S<s 0:00 /sbin/dhclient -1
root  1646  0  0.3 3012 1012 ? S<s 0:00 /sbin/dhclient -1
root  1733  0  0  0  0 ? S 0:00 [kjournald]
root  2124  0  0.3 3004 1008 ? Ss 0:00 /sbin/dhclient -1
root  2182  0  0.2 2264  596 ? Ss 0:00 syslogd -m 0
root  2186  0  0.1 2952  484 ? Ss 0:00 klogd -x
rpc   2207  0  0.2 2824  580 ? Ss 0:00 portmap
rpcuser 2227  0  0.2 2100  760 ? Ss 0:00 rpc.statd
root  2260  0  0.4 5668 1084 ? Ss 0:00 rpc.idmapd
root  2336  0  0.2 3268  556 ? Ss 0:00 /usr/sbin/acpid
root  2348  0  0.8 9100 2108 ? Ss 0:00 cupsd
root  2384  0  0.6 4080 1660 ? Ss 0:00 /usr/sbin/sshd
root  2399  0  0.3 2780  828 ? Ss 0:00 xinetd -stayalive
root  2419  0  1.1 7776 3004 ? Ss 0:00 sendmail: accept
```



方括号括起来的命令名不是真正的命令，而是按进程的方式来调度运行的内核线程。表 5.4 解释了每个字段的意义。

表 5.4 ps aux 命令输出的解释

字 段	内 容
USER	进程属主的用户名

PID	进程 ID
%CPU	该进程正在使用的 CPU 百分比
%MEM	该进程正在使用的实际内存百分比
VSZ	进程的虚拟大小
RSS	驻留集的大小 (内存中页的数量)
TTY	控制终端的 ID
STAT	<p>当前进程的状态 :</p> <p>R=可运行 D=在等待磁盘 (或者短期等待)</p> <p>S=在睡眠 (<20 秒) T=被跟踪或者被停止</p> <p>Z=僵进程</p> <p>附加标志 :</p> <p>W=进程被交换出去</p> <p><= 进程拥有比普通优先级更高的优先级</p> <p>N=进程拥有比普通优先级更低的优先级</p> <p>L=有些页面被锁在内存中</p> <p>S=进程是会话的先导进程</p>
TIME	进程已经消耗掉的 CPU 时间
COMMAND	命令的名称和参数 ^a

^a . 程序能够修改这项信息 , 因此它未必确切表示实际命令行

在 Linux 和 AIX 上的另一组有用的选择是 **lax** , 它提供了技术性更强的信息。

a 和 **X** 选项的含义同上 (显示每个进程) , **I** 选择“长格式”输出。 **ps lax** 的

运行速度也比 **ps aux** 稍快 , 因为它不必把每个 UID 都转换为用户名——

如果系统已经因别的某个进程而停顿了 , 那么效率就显得很重要了。

下面给出了一个缩短后的例子 , **ps lax** 的输出包括父进程 ID (PPID) 、谦

让值 (NI) 字段以及进程正在等待的资源 (WCHAN)。

```
redhat$ ps lax
  F  UID   PID  PPID PRI NI   VSZ   RSS WCHAN STAT TIME COMMAND
  4    0     1      0 16  0 3356  560 select  S  0:00 init [5]
  1    0     2      1 34 19  0   0 ksofti  SN  0:00 [ksoftirqd/0]
  1    0     3      1 5-10  0   0 worker  S<  0:00 [events/0]
  1    0     4      3 5-10  0   0 worker  S<  0:00 [khelper]
  5    0   2186      1 16  0 2952  484 syslog  Ss  0:00 klogd -x
  5   32  2207      1 15  0 2824  580 -      Ss  0:00 portmap
  5   29  2227      1 18  0 2100  760 select  Ss  0:00 rpc.statd
  1    0   2260      1 16  0 5668 1084 -      Ss  0:00 rpc.idmapd
  1    0   2336      1 21  0 3268  556 select  Ss  0:00 acpid
  5    0   2384      1 17  0 4080 1660 select  Ss  0:00 sshd
  1    0   2399      1 15  0 2780  828 select  Ss  0:00 xinetd -sta
  5    0   2419      1 16  0 7776 3004 select  Ss  0:00 sendmail: a
...
...
```

在 Solaris 和 HP-UX 上，**ps -ef** 是一条入手的好命令。选项 **e** 选择所有的进程，而 **f** 选项设置输出格式 (**ps-eF** 在 AIX 和 Linux 系统上也能用；注意短划线)。

```
solaris$ ps -ef
  UID   PID  PPID   C  STIME   TTY   TIME  COMD
root    0     0   80 Dec 21 ?  0:02 sched
root    1     0     2 Dec 21 ?  4:32 /etc/init-
root    2     0     8 Dec 21 ?  0:00 pageout
root   171     1   80 Dec 21 ?  0:02 /usr/lib/sendmail-bd
trent  8482  8444  35 14:34:10 pts/7  0:00 ps-eF
trent  8444  8442  203 14:32:50 pts/7  0:01 -csh
...
...
```

ps-eF 输出里各列信息的含义在表 5.5 中说明。

表 5.5 **ps-eF** 输出的说明

字段	内 容	字段	内 容
UID	属主的用户名	STIME	启动进程的时间
PID	进程 ID	TTY	控制终端
PPID	父进程的 PID	TIME	消耗的 CPU 时间

C	CPU 的使用/调度信息	COMD	命令和参数
---	--------------	------	-------

和 Linux 和 AIX 上的 ps lax 一样 , Solaris 和 HP-UX 系统上的 ps-elf 也给出了更丰富的细节 :

```
% ps -elf
F S  UID   PID  PPID   C   P   NI   ADDR     SZ   WCHAN   TIME   COMD
19 T  root    0     0   80   0   SY  f00c2fd8  0          0:02  sched
  8 S  root    1     0   65   1   20 ff26a800  88 ff2632c8 4:32  init-
  8 S  root   142    1   41   1   20 ff2e8000 176 f00cb69  0:00  syslogd
...
...
```

为了适合在本页显示 , STIME 和 TTY 两列被省略掉了 ; 它们和 ps-ef 的输出一样。表 5.6 里介绍了一些含义不直观的字段。

表 5.6 **ps-elf** 输出的说明

字 段	内 容
F	进程标志 ; 其取值随系统的不同而不同 (系统管理员很少用到)
S	进程状态 : O=当前正在运行 S=正在睡眠 (等待事件) R=有资格运行 T=被停止或者被跟踪 Z=僵尸 D=不能中断的睡眠 (通常是对磁盘)
C	进程的 CPU 利用率 / 调度信息
P	调度优先级 (对内核的内部优先级 , 不同于谦让值)
NI	系统进程的谦让值或者 SY
ADDR	进程的内存地址

SZ	进程在主存中的大小 (按页面数计算)
WCHAN	进程正在等待的对象的地址

5.8 用 top、prstat 和 topas 动态监视进程

由于 **ps** 这样的命令只提供系统的一次性快照，因此，要获得系统上正在发生事情的“全景”，往往很困难。**top** 命令是一个免费的工具，它能在许多系统上运行，对活动进程及其所使用的资源情况提供定期更新的汇总信息。在 AIX 上的等价工具是 **topas**，在 Solaris 上的类似工具是 **prstat**。例如：

```
ubuntu$ top
top - 16:37:08 up 1:42, 2 users, load average: 0.01, 0.02, 0.06
Tasks: 76 total, 1 running, 74 sleeping, 1 stopped, 0 zombie
Cpu(s): 1.1% us, 6.3% sy, 0.6% ni, 88.6% id, 2.1% wa, 0.1% hi, 1.3% si
Mem: 256044k total, 254980k used, 1064k free, 15944k buffers
Swap: 524280k total, 0k used, 524280k free, 153192k cached

 PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
3175 root 15 0 35436 12m 4896 S 4.0 5.2 01:41.9 X
3421 root 25 10 29916 15m 9808 S 2.0 6.2 01:10.5 rhn-applet-gui
 1 root 16 0 3356 560 480 S 0.0 0.2 00:00.9 init
 2 root 34 19 0 0 0 S 0.0 0 00:00.0 ksoftirqd/0
 3 root 5 -10 0 0 0 S 0.0 0 00:00.7 events/0
 4 root 5 -10 0 0 0 S 0.0 0 00:00.0 khelper
 5 root 15 -10 0 0 0 S 0.0 0 00:00.0 kacpid
18 root 5 -10 0 0 0 S 0.0 0 00:00.0 kblockd/0
28 root 15 0 0 0 0 S 0.0 0 00:00.0 pdflush
29 root 15 0 0 0 0 S 0.0 0 00:00.3 pdflush
31 root 13 -10 0 0 0 S 0.0 0 00:00.0 aio/0
19 root 15 0 0 0 0 S 0.0 0 00:00.0 khubd
30 root 15 0 0 0 0 S 0.0 0 00:00.2 kswapd0
187 root 6 -10 0 0 0 S 0 0 00:00.0 kmirrord/0
196 root 15 0 0 0 0 S 0 0 00:01.3 kjournald
...
...
```

默认设置下，这些显示内容每 10s 更新一次。那些最占 CPU 的进程显示在顶部。**top** 还接受键盘的输入并允许用户向进程发送信号和调整进程的谦让值，因此用户可以观察自己的操作是如何影响到机器的整体状态。

root 用户能够带**-q** 选项去运行 **top**，以便把它提升到可能的最高优先级。当试图跟踪一个已经把系统拖垮的进程时，这种提升就非常有用。

5.9 /proc 文件系统



Linux 版的 **ps** 和 **top** 命令都从 **/proc** 目录读取进程的状态信息，内核把有关系统状态的各种有意义的信息都放在这个伪目录里。虽然这个目录叫做 **/proc**（下面的文件系统类型也叫做“proc”），但是它里面的信息却并不局限于进程信息——内核产生的所有状态信息和统计数据都在这里。用户可以通过向 **/proc** 下的适当文件写入数据的方法来修改某些参数——参见 13.3 节的例子。

虽然使用诸如 **vmstat** 和 **ps** 这样的前端命令来访问某些信息更方便，但是有些不那么常用的信息就必须从 **/proc** 目录下直接读取。值得花时间多浏览一下这个目录，熟悉里面的各项内容。**man proc** 命令也能给出一些有用处的提示和窍门。

由于内核随时都在创建 **/proc** 下文件的内容（在读它们的同时），所以 **/proc** 下的大多数文件在用 **ls -l** 命令列出时都显示为空。用户应该用 **cat** 或者 **more** 去看这些文件里的内容是什么。但是要注意——有几个文件包含或者链接到了二进制数据，如果直接去看的话，会搞乱终端模拟程序的显示。进程特有的信息都分别被放到了按 PID 起名字的子目录里。例如，**/proc/1** 一定是包含 **init** 信息的目录。表 5.7 列出了各个进程最有用的文件。

表 5.7 /proc 目录下的进程信息文件（数字编号的子目录）

文 件	内 容
-----	-----

cmd	进程正在执行的命令或者程序
cmdline ^a	进程的完整命令行（以 null 分隔）
cwd	链到进程当前目录的符号链接
environ	进程的环境变量（以 null 分隔）
exe	链到正被执行的文件的符号链接
fd	子目录，其中包含链到每个打开文件的描述符的链接
maps	内存映射信息（共享段、库等）
root	链到进程的根目录（由 chroot 设置）的符号链接
stat	进程的总体状态信息（ps 最擅长解析这些信息）
statm	内存使用情况的信息

^a . 如果进程被交换出内存的话可能得不到

在 **cmdline** 和 **environ** 文件里的各个部分用空字符（null）而不是换行符（newline）分隔。用户可以借助命令 **tr "\000" "\n"** 过滤这些文件的内容，使之可读性更好。

子目录 **fd** 以符号链接形式表示进程打开的文件。连接到管道或者网络套接口的文件描述符没有相关联的文件名。内核代之以提供一种通用的描述形式来作为链接目标。

maps 文件用于确定一个程序链接到哪些库或者依赖于哪些库。

Solaris 和 AIX 也有一个/**proc** 文件系统，但是它里面没有 Linux 上有的那些状态和统计信息。有一组合起来称为 **proc** 工具的程序能够给出正在运行的进程的一些有用信息。例如，AIX 上的 **procsig** 命令及其在 Solaris 上的等价命令 **psig** 可以输出一个给定进程的信号动作和处理例程。表 5.8 给出了大多数有用的 **proc** 工具及其功能。

表 5.8 AIX 和 Solaris 上读取/proc 信息的命令

Solaris ^a	AIX	说 明
pcred [<i>pid</i> <i>core</i>]	procrcrd [<i>pid</i>]	打印/设置真实、有效和保存 UID/GID
pldd [-F] [<i>pid</i> <i>core</i>]	procldd [<i>pid</i>]	显示依赖的库（类似 ldd ）
psig [<i>pid</i>]	procsig [<i>pid</i>]	列出信号的动作和处理 pfiles
pfiles [<i>pid</i>]	procfiles [<i>pid</i>]	打印打开的文件
pwdx [<i>pid</i>]	procwdx [<i>pid</i>]	打印当前的工作目录
pwait [<i>pid</i>]	procwait [<i>pid</i>]	等待一个进程退出

a . 有些 Solaris 的 proc 工具程序可以把一个 core 文件（内存转储）作为输入。这主要是调试工具

HP-UX 没有/proc 文件系统或者对应的东

西。

5.10 strace、truss 和 tusc：追踪信号和系统调用

有时候判断一个进程实际正在做什么相当困难。用户可能不得不根据从文件系统以及 **ps** 这样的工具取得的间接数据和经验来推测。

Linux 能让用户通过 **strace** 命令直接观察一个进程，进程每调用一次系统调用，以及每接收到一个信号，这个命令都能显示出来。Solaris 和 AIX 的类似命令叫做 **truss**。HP-UX 的等价命令是 **tusc**，不过必须单独安装这个命令。

用户甚至可以把 **strace** 或者 **truss** 附在一个正在执行的进程上，监视一会儿该进程，再从这个进程脱离，整个过程都不会影响那个进程³²。

虽然这些系统调用出现在相当低级的位置，但是用户通常都可以从输出里了解到进程的一些活动情况。例如，下面的记录是由 **strace** 附在一个活动的 **top** 进程上获得的：

```
redhat$ sudo strace -p 5810
gettimeofday({1116193814, 213881}, {300, 0})          = 0
open("proc", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 7
fstat64(7, {st_mode=S_IFDIR|0555, st_size=0, ...})    = 0
fcntl64(7, F_SETFD, FD_CLOEXEC)                      = 0
getdents64(7, /* 36 entries */, 1024)                = 1016
getdents64(7, /* 39 entries */, 1024)                = 1016
stat64("/proc/1", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
open("/proc/1/stat", O_RDONLY)                         = 8
read(8, "1 (init) S 0 0 0 0 -1 4194560 73...", 1023) = 191
close(8)                                              = 0
...
...
```

strace 不仅能够显示进程每次调用的系统调用名，它还能解析参数，给出由内核返回的结果代码。

strace 还有不少好处，在 man 手册里有其中大多数功能的文档说明。例如，-f 标志后面跟 **fork** 出来的进程，这个可以帮助跟踪像 httpd 这样生出很多子进程的守护进程。-e 这

32. 通常是这样的。strace 会中断系统调用，被监视的进程接下来必须准备重新发起调用，这是 UNIX 软件应遵守的标准规则，但不是总能看到这样的结果。

个文件选项只显示文件操作，对应找到难以定位的配置文件的位置特别方便。

在本例中，**top** 先检查当前时间。然后用 **open** 打开/**proc** 目录，用 **stat** 获得其信息，然后读取该目录的内容，由此获得当前正在运行的进程清单。**top** 接着用 **stat** 获得代表 **init** 进程的那个目录的信息，然后打开/**proc/1/stat** 读取 **init** 的状态信息。

下面是一个在 Solaris 上用 **truss** 的简单例子（对 **date** 命令）：

在这个例子里，**date** 先分配内存和打开依赖的库（没有显示出来），然后使用系统调用 **time** 读取系统时间，打开恰当的时区文件得到正确的偏移量，再调用系统调用 **write** 打印日期和时间戳。

5.11 失控进程

失控进程有两类：一类是过度占用了某种系统资源（例如 CPU 时间或磁盘空间等）的用户进程，另一类是突然发狂并呈现狂暴行为的系统进程。第一种失去控制的类型未必是一种故障，它可能仅仅是过度占用了资源。系统进程始终应该具有合理的行为方式。要了解有关失控进程的更多信息，请参考 29.5 节。

通过查看 **ps** 或者 **top** 的输出，能够确认是哪些进程占用了过多的 CPU 时间。如果明显发现某个用户进程正在消耗的 CPU 时间比预计的合理值多，那么就要调查这个进程。留意正等待运行的进程的数目也有用。使用 **uptime** 命令给出 1 分钟、5 分钟和 15 分钟间隔的平均

负载（可运行的进程的平均数）。

之所以要在对进程做出反应以前，先查明该进程正试图要做的事情，原因有两个。第一，该进程可能不但是合法的，而且对用户来说非常重要，这时候仅仅因为进程碰巧占用了大量的 CPU 时间而杀死它们并不合理。第二，进程可能是恶意的或具有破坏性的，如果是这样的话，管理员已经知道进程正在做的是什么事情（例如破译口令）的话，就可以修好受损的地方。

如果进程使用的内存相对于系统的物理 RAM 来说太多的话，就会造成严重的性能问题。

用 **top** 命令可以查看各个进程使用的内存大小。VIRT 列给出每个进程分配的虚拟内存总量，RES 列给出了该内存中当前被映射到特定内存页（“驻留集”）的比例。在 Linux 系统上，使用显卡的应用（例如 X 服务器）会因为显存也计入内存用量而使得数据看上去不好。这两个数字都包括了像库这样的共享资源，这使得它们可能会误导人。在 DATA 列里能找到对进程的内存消耗量更为直接的测定值。为了在 **top** 的输出里加上这一列的显示，要在运行 **top** 之后键入 **f**，从列表中选择 DATA。DATA 值表明了每个进程的数据和堆栈段占用的内存量，所以这个值对单个进程的针对性相当强（模块共享内存段）。在观察这个值随时间的增加情况的同时，也要观察绝对的内存量。

那些产生输出信息的失控进程能够填满整个文件系统，从而导致出现大量问题。文件系统

被填满以后，控制台上将记录大量的消息，而试图向文件系统写入数据则会产生错误消息。

在这种情况下，要做的第一件事情是停止正在填满磁盘的那个进程。用 **df-k** 命令显示文件系统的使用情况。寻找用了大概 100% 满的文件系统³³。在找出来的这个文件系统上使用 **du** 命令，寻找使用最多空间的目录。重复使用 **du** 命令直到把大文件都找出来为止。

如果不能判定哪个进程正在用那个文件，可以试试用 **fuser** 和 **lsof** 命令（在 6.2 节详细

³³ 大多数文件系统的实现都保留一定比例（大约 5%）的存储空间用作“回旋余地”，但是以 root 身份运行的进程可以蚕食这块空间，导致报告的使用率超过 100%。

介绍) 了解更多的信息。

作为管理员 , 可能想把所有看起来有问题的进程都挂起 , 一直到自己找出导致故障的那个进程为止。但当确定问题症结所在以后 , 要记得把其他没有问题的进程重新启动起来。一旦找到导致故障的那个进程 , 就要删除该进程创建的所有文件。有时候 , 更明智的做法是用 gzip 压缩这个文件 , 并起个别的名字 , 以防万一它包含有用或者重要的数据。

5.12 推荐读物

BOVET, DANIEL P. AND MARCO CESATI.*Understanding the Linux Kernel (3rd Edition)*. Sebastopol, CA: O'Reilly Media, 2006.

MCKUSICK, MARSHALL KIRK, AND GEORGE V. NEVILLE-NEIL.*The Design and Implementation of the FreeBSD Operating System*. Reading, MA: Addison-Wesley Professional, 2004.

5.13 习题

E5.1 解释一个文件的 UID 同一个正在运行的进程的真实 UID 和有效 UID 之间的关系。除了文件的访问控制之外 , 进程的有效 UID 有什么用途 ?

E5.2 假定您所负责的站点中 , 有一个用户已经启动了一个长期运行的进程 , 它消耗了机器的很大一部分资源。

a) 如何认识到某个进程正在消耗资源 ?

b) 假定异常的进程可能是合法的 , 不应该杀死它。给出您要把它“冻结”起来 (在您调查期间 , 暂时停止它的运行) 应该使用的命令。

c) 随后 , 您发现这个进程属于您的老板 , 必须继续运行下去。给出您要继续执行这个任务应该使用的命令。

d) 另一种可能是假定需要杀死这个进程。您会发送什么信号 ? 为什么 ? 如果您要保证这个进程确实已经被杀死了 , 该怎么做 ?

E5.3 找出一个能导致内存泄漏的进程 (如果手头没有 , 就自己写一个) 。用 ps 或者 top 来监视程序运行时的内存使用情况。

★E5.4 编写一个处理 ps 输出的 Perl 脚本 , 确定在系统上正在运行的进程总的 VSZ 和 RSS 值。这些数字与系统物理内存和交换空间的实际量有什么样的关系 ?