

计算机科学丛书

UNIX编程环境

(美) Brian W.Kernighan 著
Rob Pike

陈向群 等译



机械工业出版社
China Machine Press

本书对UNIX操作系统的编程环境做了详细而深入的讨论,内容包括UNIX的文件系统、Shell、过滤程序、I/O编程、系统调用等,并对UNIX中的程序开发方法做了有针对性的指导。本书内容深入浅出,实例丰富,无论是UNIX系统的初学者还是专业人员都可从本书受益。本书亦可作为大学生、研究生学习UNIX的教材。

Brian W.Kernighan & Rob Pike: The UNIX Programming Environment.

Authorized translation from the English language edition published by Prentice Hall.

Copyright © 1984 by Bell Telephone Laboratories, Incorporated.

All rights reserved. For sale in Mainland China only.

本书中文简体字版由机械工业出版社出版。未经出版者书面许可,不得以任何方式复制或抄袭本书的任何部分。

版权所有,翻印必究。

本书版权登记号:图字:01-1999-1567

图书在版编目(CIP)数据

UNIX编程环境/(美)柯尼汉(Kernighan, B.W.), (美)派克(Pike, R.)著;陈向群等译.
—北京:机械工业出版社,1999.10

(计算机科学丛书)

书名原文: The UNIX Programming Environment

ISBN 7-111-07115-8

I.U… II.①柯… ②派… ③陈… III.UNIX操作系统—程序设计 IV.TP316

中国版本图书馆CIP数据核字(1999)第23881号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑:陈 谊

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2006年2月第1版第8次印刷

787mm×1092mm 1/16·16.5印张

印数:16 001-17 000册

定价:24.00元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换

译者序

UNIX是当代计算机操作系统的一个成功典范。从计算机操作系统的发展史来看，UNIX超过了以往所有的操作系统，取得了前所未有的成功。

UNIX系统的诞生已经有30年了。许多操作系统在这期间早已寿终正寝，然而没有一位计算机工作者会预言UNIX在可见的今后年代中走下坡路。UNIX不仅在操作系统发展的历史上具有里程碑作用，而且，它在学术上和教学中也是人们首先研究和探讨的为数不多的操作系统之一。UNIX极大地推动了操作系统理论的发展，是所有计算机院校、系所必须开设和研讨的一门课程。

UNIX不仅在学术上取得了成功，多年来，它在商业上的成功也是极为出色的。成百万台的机器在运行着它。而近年来，随着互联网的迅猛发展，它的装机量更是可观，而更重要的是，凡是重要的互联网上的站点，几乎都把UNIX作为首选的Web服务器操作系统。从这些事实来看，UNIX的重要性是不言而喻的。

与其他系统不同的一点是，UNIX的成功和C语言的成功密不可分。UNIX自身就是C高级程序设计语言的编程典范。UNIX推动了C语言的应用和普及，而C语言又反过来促进了UNIX在更广阔领域里的成功。

如今，世界正处于世纪之交，随着Linux操作系统在Internet上的出现，人们又一次看到UNIX操作系统的新的山峰出现在我们面前。Linux是“free UNIX”，即免费的UNIX。Linux遵从国际上相关组织制定的UNIX标准，即POSIX。它的结构、功能以及界面都与经典的UNIX并无二致。Linux不但继承了UNIX的全部优点，而且还增加了一条其他操作系统不曾具备的优点，即Linux操作系统源码全部开放。从某种意义上讲，Linux是UNIX在国际互联网环境下的一次新生。想要学习90年代的Linux吗？请从UNIX开始，请从这本书开始。

应该说本书不是一本普通的有关UNIX的书籍。本书的写作日期是1984年，十多年岁月的考验更显露其华彩。在书籍的海洋中，本书是有关UNIX编程环境的一本举世公认的经典著作。迄今为止，在这个领域中，还没有其他的书能够替代它。这是一本深入浅出地讲解如何在UNIX环境下进行程序设计的优秀书籍。书中引用大量编程实例，由浅入深地讲解如何使用UNIX及其各种工具，以及如何用C语言在UNIX环境下写出高质量的程序来。正因为本书的重要价值，所以在国际上，许多一流大学的相关专业都采用本书作为首选教科书。

想要成为一名UNIX系统上的程序设计高手吗？请仔细阅读本书，并上机实践。想要掌握C语言的精妙技巧吗？请在机器上试验本书中的实例并尝试做出改进。

本书对初学者也同样适用。跟随本书的讲解，一边学习，一边上机实践，初学者就一定会成为编程好手。

本书适合作为大专院校相关专业的教科书。对于想深入掌握UNIX和C语言的程序设计人员，以及想学习和掌握Linux的人员，本书也不失为一本难得的参考书。

参加本书翻译工作的还有：张乃琳、李茹、汤敏、王梦秋、叶松和翁念龙。由于时间有限，书中难免有误，请读者指出，以便再版时改正。

1999年2月于北京

序 言

“UNIX安装的数量已经增至10台，超过了预期。”

《UNIX程序员手册》，第2版，1972年6月

UNIX[⊖]操作系统于1969年首次在贝尔实验室一台闲置的DEC PDP-7计算机上启用。当时，Ken Thompson从Rudd Canaday、Doug McIlroy、Joe Ossanna和Dennis Ritchie处得到启示，并在他们的支持下，编写了小型通用分时系统，其适用性好、可信度高，吸引了热心的用户，并最终装备在一台较大的计算机——PDP-11/20上。系统早期的用户之一是Ritchie，他在1970年曾帮助将该系统转到PDP-11计算机上。Ritchie也曾设计和编写了C程序语言的编译程序。1973年，Ritchie和Thompson用C语言重写了UNIX核心，打破了系统软件用汇编语言编写的传统。经过此次重写，该系统最终成为今天必不可少的操作系统。

大约在1974年，UNIX获准进入大学“用于教育目的”，而且，几年之后便进入了商业领域。在此期间，UNIX系统在贝尔实验室日益发展壮大，成功地进入到实验室、软件开发项目、字处理中心和电话公司运行支持系统。从那时起，UNIX开始向全球扩散，从微型计算机到大型机，已安装了数以万计的UNIX系统。

是什么促使UNIX系统如此成功呢？首先，由于是用C语言编写，因此它是可移植的——UNIX系统的运行范围可以从微型机到大型机，这是一个强大的商业优势。第二，源代码是用高级语言编写的，且很容易得到，从而使系统容易适应特殊的需求。最后，也是最重要的一点，它是一个良好的操作系统，特别是对程序员而言。UNIX编程环境非常丰富而且容易取得成效。

尽管UNIX系统引入了很多创新的程序和技术，但并非其中某个单独的程序或想法使得工作完美无缺。相反，是程序设计的方法，即应用计算机的基本原则使系统富有成效。虽然这个基本原则不能用一句话概括，但其核心就是，系统的能力更多地来自程序之间的关系，而不是程序本身。许多单独的UNIX程序只能完成相当简单的工作，然而，在与其他程序结合起来使用时，它们就成为通用而有效的工具。

本书的目的是传授UNIX程序设计的基本原则，由于这个基本原则是以程序间的关系为基础的，因此，除了程序组合和运用程序进行创建程序的主题外，我们还用了大量篇幅对单个工具进行讨论。要想很好地使用UNIX系统及其组成部件，不仅必须了解如何使用程序，还要了解它们是怎样与环境相匹配的。

尽管UNIX系统得到了广泛应用，但精通的用户却越来越难找到。我们经常发现富有经验的用户(包括我们自己在内)只能找到问题的笨解法，或者只会写程序做现有的工具易于处理的工作。当然，没有一定的经验和对其透彻的理解，更不易获得精巧的解法。我们希望无论是新手还是老用户，通过阅读本书，都能够增强理解力，从而使系统的使用更加有效和有趣。

⊖ UNIX是贝尔实验室的商标。“UNIX”不是首字母缩拼词，它是一个同MULTICS稍微有联系的双关语，在UNIX之前，Thompson和Ritchie使用MULTICS操作系统。

希望读者更好地使用UNIX。

我们把焦点对准程序员，希望通过本书的学习，能使其工作更有效，从而也使程序组的工作更有成果。尽管我们的主要对象是程序员，但学习本书的前4章或前5章并不需要具有编程经验，因此，对于其他读者来说，本书也是很有帮助的。

我们尽可能尝试使用真实的而不是假设的例子，来表述我们的观点。有些程序尽管在书中作为实例，但也可成为常用程序的一部分。所有实例都经过了机器可读格式文本的直接测试。

本书的编排如下。第1章为系统使用的基础入门。其中包括登录、邮件、文件系统、常用命令和命令解释的基本原理。有经验的用户可以略过此章。

第2章讨论UNIX文件系统。文件系统是系统操作和使用的核心，因此必须充分理解才能更好地使用。这一章讲述文件和目录、容许权限和文件模式以及I节点。章末是文件系统层次的浏览和设备文件的解释。

命令解释程序，即shell，是一个基本工具，不仅用于执行程序，也用于编写程序。第3章讲述怎样按要求使用shell：创建新命令、命令变量、shell变量、基本控制流和输入输出重定向。

第4章介绍过滤程序，它是当数据流通过时，对数据执行简单转换的程序。首先涉及grep模式搜索命令和其对应程序；接下来讨论诸如sort的一些更多的常用过滤程序；其余部分重点讨论两个通用数据转换程序，称为sed和awk。sed是流编辑程序，是当数据流通过时，作出编辑更改的一个程序。awk是用于简单信息检索和报告生成任务的一种编程语言。使用这些程序，有时再加以与shell的配合，常常能完全避免流于常规的程序设计。

第5章讨论如何使用shell编程，以供他人使用。主要内容包括更高级的控制流和变量、陷阱和中断处理。本章的实例充分利用了sed、awk和shell。

至此读者将完全领会运用shell和其他现有程序所能进行的工作。第6章讲述有关运用标准I/O库编写新程序的内容。这些程序用C语言编写，我们假设读者了解或至少最近学过C语言。我们会展示用于设计和组织新程序的实用策略，介绍怎样用易于管理的步骤创建程序如何利用现有的工具。

第7章涉及系统调用，这是其他所有软件层的基础，主要内容包括输入输出、文件创建、出错处理、目录、I节点、进程和信号。

第8章讨论程序开发工具。yacc是一种语法分析程序生成器；make可以控制编译大程序的过程；lex可以生成词法分析程序。上述讨论是以一个大程序的开发为基础的，该大程序是一种类似C语言的可编程计算器。

第9章讨论文档处理工具，本章可以独立于其他章节阅读。

附录A概括了标准编辑器ed。尽管许多读者日常工作中喜欢运用其他编辑器，但ed是非常通用、有益而且高效的。它的正则表达式是其他程序(如grep和sed)的核心，而且只为此理由就值得一读。

附录B是用于第8章计算器语言的参考手册。

附录C是计算器程序最后版本，列出了所有的源代码以便于阅读。

另外，还有一些实用的内容。首先，UNIX系统的使用已经很普遍，有多种广泛应用的版本。例如，第7版出自UNIX系统的发源地——贝尔实验室计算机科学研究中心。系统III和系统V是贝尔实验室支持的正式版本。加利福尼亚大学Berkeley分校提供的系统源自第7版，通

常称为UCB 4.xBSD。另外，特别是对于小型计算机，有大量变种UNIX系统，它们也是从第7版派生的。

我们尽可能地把不同版本中完全一致的内容放在一起，来对付这些版本差异。尽管我们将要讲授的课程是独立于任何特定版本的，但对于特定的细节，我们选择了和第7版中有关内容一致的描述，因为它构成了广泛应用的绝大多数UNIX系统的基础。我们也已经在贝尔实验室的系统V上和Berkeley 4.1BSD中运用了这些例子；只需做一些轻微的改动，而且改动只限于少数例子。不管机器运用何种版本，读者会发现其间的差异是很小的。

其次，虽然本书中有很多参考资料，但它不是一本参考手册。我们知道讲授使用、方法和风格要比仅仅讲授细节重要得多。《UNIX程序员手册》是信息的标准来源，书中的许多参见内容是指参见这个手册。读者需要使用手册去解决那些没有包含在本书中的问题，或者检测读者的系统与本书作者的系统之间的不同之处。

最后，我们相信，实践出真知。本书应在终端旁阅读，这样便于对所讲的内容进行实验、检查或者指正，探索其局限和变化。请边阅读、边实践，再回顾小结，然后进一步地阅读。

我们相信，尽管UNIX系统不是完美无缺的，但它是一个相当优秀的计算机环境，我们希望通过对本书的阅读，读者也得出这一结论。

Brian Kernighan

Rob Pike

目 录

译者序

序言

第1章 初学UNIX 1

1.1 起步 1

1.1.1 有关终端和击键的一些

预备知识 1

1.1.2 与UNIX会话 2

1.1.3 登录 3

1.1.4 键入命令 3

1.1.5 异常的终端行为 4

1.1.6 键入错误 4

1.1.7 继续键入 5

1.1.8 中止程序 5

1.1.9 注销 6

1.1.10 邮件 6

1.1.11 用户间通信 6

1.1.12 新闻 7

1.1.13 手册 7

1.1.14 计算机辅助教学 8

1.1.15 游戏 8

1.2 文件和常用命令 8

1.2.1 创建文件 8

1.2.2 列出文件 9

1.2.3 打印文件 11

1.2.4 移动、复制和删除文件 12

1.2.5 文件名 13

1.2.6 有用的命令 13

1.2.7 文件系统命令小结 15

1.3 目录 16

1.4 shell 19

1.4.1 文件名简写 19

1.4.2 I/O重定向 21

1.4.3 管道 23

1.4.4 进程 24

1.4.5 剪裁环境 26

1.5 UNIX系统的其余部分 28

第2章 文件系统 30

2.1 文件系统的基础 30

2.2 文件结构 33

2.3 目录和文件名 35

2.4 权限 37

2.5 I节点 41

2.6 目录层次 45

2.7 设备 47

第3章 shell的使用 51

3.1 命令行结构 51

3.2 元字符 53

3.3 创建新命令 57

3.4 命令参数 59

3.5 程序输出作为参数 62

3.6 shell变量 63

3.7 进一步讨论I/O重定向 66

3.8 shell程序里的循环 68

3.9 bundle合并 69

3.10 为什么说shell是可编程的 71

第4章 过滤程序 72

4.1 grep系列 72

4.2 其他过滤程序 75

4.3 流编辑程序sed 77

4.4 模式扫描与处理语言awk 81

4.4.1 字段 82

4.4.2 打印 83

4.4.3 模式 83

4.4.4 BEGIN与END模式 84

4.4.5 算术运算与变量 84

4.4.6 控制流 86

4.4.7 数组 87

4.4.8 关联数组 88

4.4.9 字符串	89	7.5 信号和中断	164
4.4.10 与shell的交互作用	91	第8章 程序开发	169
4.4.11 基于awk的日历服务	91	8.1 第一阶段: 四功能计算器	169
4.4.12 附注	94	8.1.1 语法	170
4.5 好的文件与过滤程序	94	8.1.2 yacc概述	170
第5章 shell编程	96	8.1.3 第一阶段的程序	171
5.1 定制cal命令	96	8.1.4 在程序中增加单目减	174
5.2 which命令	100	8.1.5 关于make	175
5.3 while和until循环	104	8.2 第二阶段: 变量和错误恢复	175
5.4 trap: 捕获中断	109	8.3 第三阶段: 任意变量名和内部函数	178
5.5 overwrite: 改写文件	110	8.3.1 再谈make	184
5.6 zap: 使用名字终止进程	114	8.3.2 关于lex	185
5.7 pick命令: 空格和参数	116	8.4 第四阶段: 编译成机器	187
5.8 news命令: 社团服务信息	118	8.5 第五阶段: 控制流和关系运算符	193
5.9 get和put: 追踪文件变动	120	8.6 第六阶段: 函数、过程和I/O	197
5.10 小结	124	8.7 性能评价	205
第6章 使用标准I/O编程	126	8.8 小结	206
6.1 vis: 标准I/O	126	第9章 文档处理	208
6.2 vis第2版: 程序参数	128	9.1 宏程序包ms	209
6.3 vis第3版: 访问文件	130	9.1.1 阵列文本	211
6.4 p: 一次显示一屏	133	9.1.2 改变字体	212
6.5 pick	137	9.1.3 其他命令	212
6.6 错误与调试	137	9.1.4 宏程序包mm	214
6.7 zap	139	9.2 troff	214
6.8 idiff: 交互式文件比较程序	141	9.2.1 字符	214
6.9 获取环境变量	145	9.2.2 改变字体和尺寸	215
第7章 UNIX系统调用	147	9.2.3 基本troff命令	216
7.1 低级I/O	147	9.2.4 定义宏	217
7.1.1 文件描述符	147	9.3 tbl和eqn预处理程序	217
7.1.2 文件I/O	148	9.3.1 表格	218
7.1.3 创建文件	149	9.3.2 数学表达式	219
7.1.4 错误处理	151	9.3.3 输出	220
7.1.5 随机访问	151	9.4 排印手册	222
7.2 文件系统: 目录	152	9.5 其他文档处理工具	225
7.3 文件系统: I节点	156	第10章 结束语	228
7.4 进程	160	附录A 编辑器概述	230
7.4.1 创建低级进程	160	附录B hoc手册	239
7.4.2 控制进程	161	附录C hoc清单	243

第1章 初学UNIX

什么是UNIX? 狭义地看, 它是一个分时操作系统内核, 即一个控制计算机的资源并分配给用户的程序。它让用户运行其程序; 它控制与机器连接的外围设备(硬盘、终端、打印机等等); 它提供一个文件系统用以管理诸如程序、数据及文档一类信息的长期存储。

广义地看, UNIX通常不仅包含内核, 还包括一些基本程序, 诸如编译器、编辑器、命令语言、用于复制和打印文件的程序等。

从更广的角度来看, UNIX可以包括由用户开发的、用于运行用户定制系统的程序, 如文档处理工具、统计分析程序以及图像软件包。

这些有关UNIX的解释究竟正确与否, 取决于读者所面对系统的应用级别。本书其他部分提到UNIX时, 会在上下文指示其内在含义。

UNIX系统有时看起来比实际上更复杂——对于新用户而言, 很难以最佳方式使用可用资源。所幸它并不难入门——只要了解很少的几个程序, 就可以开始工作了。本章帮助读者尽快地学会使用UNIX系统, 是概述, 而不是手册; 后续的章节将详细介绍各种内容。本章涉及以下主要内容:

- 基本操作——登录和退出、简单的命令、纠正输入错误、邮件及终端间通信。
- 日常使用——文件及文件系统、文件打印、目录以及常用命令。
- 命令解释器或shell——文件名缩写、输入输出重定向、管道、删字符及消行符设置、命令查询路径定义。

如果你用过UNIX, 那么对本章的多数内容应该是熟悉的, 可以直接跳至第2章。

即使阅读本章, 也需要一份《Unix程序员手册》。对作者而言, 告诉你翻阅手册中的某些内容, 比在书中重复这些内容更为方便。本书不是为了替代手册, 而是为了教会你更好地使用手册中的命令。另外, 本书中所叙述的内容可能同你所使用的系统有所差别。在手册中的开始处有索引, 可以协助找出解决某一问题的程序; 应学会如何使用它。

最后, 有一句忠告: 不要害怕做试验。如果你是初学者, 那么放心, 不会出现伤害自身或其他用户的事。要通过试验了解事物原理。本章篇幅较长, 最好的方法是一次读几页, 并在学习过程中不断试验。

1.1 起步

1.1.1 有关终端和击键的一些预备知识

为了避免繁冗解释有关计算机使用的所有事项, 作者假定你已经熟悉计算机终端, 并知道使用方法。如果对下面的叙述迷惑不解, 请询问身旁的专家们。

UNIX系统是全双工的: 你在键盘上敲入的字符送至系统, 然后系统回送给终端并在屏幕上显示出来。通常echo进程把字符直接复制到屏幕上, 这样就可以看到所输入的是什么。但是有的时候, 比如在键入保密口令时, echo关闭, 字符就不会在屏幕上显示出来。

多数键盘字符是普通字符, 没有什么特殊的含义。但有些字符是通知计算机怎样解释所

键入的内容的。到目前为止，这些键中最重要的是Return键。Return键说明一行输入的结束；系统作出反应并把屏幕上的光标移到下一行的起始处。必须按下Return键后，系统才会对键入的字符作出解释。

Return键是一种控制字符——即不可见的字符，它控制着终端上输入输出的某种状况。在任一台终端上，Return都有自己的按键，但是其他大多数控制字符并非如此。相反，它们必须通过按住Control键(Control键有时被称作为Ctl键、Cntl键或Ctrl键)，同时按下另一个键，通常是一个字母来输入。例如，Return可以通过按下Return键输入，或者，按住Control键并键入m键。所以Return键又可称为Control-m键，它可以写成Ctl-m。其他的控制字符有：Ctl-d，表示程序输入到此结束；Ctl-g，终端上的振铃鸣响；Ctl-h，通常称为Backspace(退格键)，它可在纠正错误的输入时使用；还有Ctl-i，通常称作Tab，它使光标往前跳到下一个Tab的停止位，这一点非常类似于普通的打字机。在UNIX系统中Tab间隔为8个空字符。在多数终端上Backspace和Tab都有各自的键符。

有两个键具有特殊的含义：一个是Delete，有时又称作为Rubout或某些缩写；另一个是Break，有时又称作为Interrupt。在多数UNIX系统中，按Delete键可立即中止程序，而不等待程序完成。在某些系统中，Ctl-c提供这项功能。而在有些系统中，根据终端的连接方式，Break可能是Delete或Ctl-c的同义词。

1.1.2 与UNIX会话

让我们从与UNIX系统的会话开始。在本书的例子中，你键入的内容用斜体字符表示，计算机回应的用正体表示。

建立连接：必要时拨号或打开电源。

系统应该显示

login: you	键入名称后按回车
Password:	口令在键入时不会显示
You have mail.	登录后可看邮件
\$	系统等待用户命令
\$	按几次回车
\$ date	日期和时间
Sun Sep 25 23:02:57 EDT 1983	
\$ who	谁在使用机器
jlb tty0 Sep 25 13:59	
you tty2 Sep 25 23:01	
mary tty4 Sep 25 19:03	
doug tty5 Sep 25 19:22	
egb tty7 Sep 25 17:17	
bob tty8 Sep 25 20:48	
\$ mail	读邮件
From doug Sun Sep 25 20:53 EDT 1983	
give me a call sometime monday	
?	回车以到下一条消息
From mary Sun Sep 25 19:07 EDT 1983	下一条消息
Lunch at noon tomorrow?	
? d	删除该消息
\$	没有消息了
\$ mail mary	向mary发邮件
lunch at 12 is fine	
ctl-d	邮件结束
\$	挂断电话或关闭终端
	全部结束

上述内容就是一段会话的全部，当然，有时人们也进行一些其他任务。本节的下面部分将讨论这段会话，顺便分析其他一些有用的程序段。

1.1.3 登录

登录必须有登录名和口令，这些可以从系统管理员处得到。UNIX系统可以适用于多种类型的终端，但最习惯于小写字体的设备；它区分大小写！如果你的终端只有大写字符(类似某些视频和便携式终端)，那么日子就难过了，应该另外换一台终端。

要确认设备的开关设置为大小写、全双工，以及身边专家们建议的其他设置，诸如速率，即波特率。要为终端建立连接，这也许要拨通电话，或者只是拨一下开关。不论哪种情形，系统都会出现

```
login:
```

如果出现乱码，则可能是速率不对，请检查速率和其他设置。如果仍旧不成功，轻按几次Break键或Interrupt键。如果仍不出现登录消息，那就只好请人帮忙了。

出现login: 登录消息后，用小写字母键入登录名，然后按回车键。如果要求口令，系统会有提示，而且在输入口令时，屏幕不会显示所输入的口令。

登录一旦成功，系统便响应一个提示符，通常是一个单个字符，表示系统已经准备好接收用户的命令。提示符多半是美元符号(\$)或百分比符号(%), 但是可以将其改为任意的符号，这一点以后再讨论。提示符实际是由一个名为命令解释器或称为shell的程序打印出来的，它是系统对用户的主界面。

在提示符之前可能会有日期消息，或者有关于电子邮件的通知。有时系统会询问你使用的终端类型；这有助于系统利用终端所具有的特性。

1.1.4 键入命令

一旦有了提示符，就可以键入命令了，命令是请系统完成某项工作的要求。我们将使用“程序”这一词作为命令的代名词。当看到提示符(假设是\$)时，键入date并按下Return键。系统应该回应日期和时间，然后显示下一个提示符，所以整个的处理过程在终端上看起来如下所示：

```
$ date
Mon Sep 26 12:20:57 EDT 1983
$
```

不要忘了回车键Return，不要键入\$。如果你觉得系统没有反应，可以按Return键；应该会有响应。以后不再提Return键，但是在每一行的结尾都需要它。

下一个要试用的命令是who，它说明当前有哪些人登录上机了：

```
$ who
rlm      tty0      Sep 26 11:17
pjw      tty4      Sep 26 11:30
gerard   tty7      Sep 26 10:27
mark     tty9      Sep 26 07:59
you      ttya      Sep 26 12:20
$
```

首列是用户名。第二列是连接的终端的名称(tty表明teletype，终端的一个学名)。其余信

息告知登录的日期和时间。读者还可试验

```
$ who am i
you      ttya      Sep 26 12:20
$
```

如果键入出错，输入了一条不存在的命令，会被告知没有发现该命令：

```
$ whom                      拼错命令名.....
whom: not found             .....所以系统不知如何执行
$
```

当然，如果不恰当地键入了一条实际存在的命令，它会运行，也许会出现奇怪的结果。

1.1.5 异常的终端行为

有时终端的行为会是很奇怪的，比如，一个字符也许会显示两次，或者Return键可能不把光标置到下一行的首列。通常可以把终端关闭后再开启，或者退出登录然后再次登录以消除这些现象。也可以阅读手册第1节中有关stty(set terminal options)命令的叙述。如果终端没有Tab键，要想巧妙地处理该键，可键入命令

```
$ stty -tabs
```

接着系统会把tabs转换成正确数量的空格。如果终端具有计算机可设置的Tab跳格键功能，命令tabs会正确地为us户设置跳格位置。(这样读者就需要键入

```
$ tabs 终端类型
```

从而使系统正确工作——请参阅手册中有关tabs命令的说明)

1.1.6 键入错误

如果出现了键入错误，并且在按Return键前发现了它，可有两个方法纠正：一次删除一个字符或者删行重新键入。

如果键入消行符，其缺省值是@，会取消整行，就好像没有键入该行一样，从而可以开始新的一行：

```
$ ddtae@                      命令输入错，整行删除；开始新的一行
date
Mon Sep 26 12:23:39 EDT 1983
$
```

字符#删除所键入的最末一个字符，每个#字符删去一个字符，一直到行的起始处(但不会超越它)。所以如果键入出错太多，可以进行如下纠正：

```
$ dd#atte##e                  纠正错误
Mon Sep 26 12:24:02 EDT 1983
$
```

专用删字符和消行符是与系统密切相关的。在很多系统中(包括作者使用的系统)，删字符都已改为Backspace键，这在视频终端上很成功。你可以迅速检查所使用的系统是哪种情况：

```
$ date←                      试一下Backspace键
date←: not found              不是Backspace键
$ date#                       试一下#
Mon Sep 26 12:26:08 EDT 1983  正是#
```

\$

(我们把Backspace键印刷为←，这样就可以看到它。)消行符的另一种常见形式是Ctrl-u。

在本节的其余部分，将使用字符#作为删字符，因为它较为清晰可见，如果系统不一致，请做好思想准备。稍后，在1.4.5节将介绍如何设置你所喜欢的删字符和消行符。

假如必须把删字符或者消行符作为文本的一部分输入，该怎么办呢？如果在#或@之前加入反斜杠\，它们就失去了特有的含义。所以要输入#或@，只需键入\#或\@。系统会先把终端光标移到在@之后的下一行，即使它前面有反斜杠。不过不用担心，该符号已被记录下来了。

反斜杠有时被称作escape字符，它主要用于指明后续字符是特殊字符。要消除反斜杠，需要键入两个删除字符：\##。知道这是为什么吗？

键入的字符在到达最终目的地之前，要经过一系列程序的检查和解释，至于如何解释它们，则完全取决于它们如何结束以及如何到达终点。键入的每一字符立即在终端显示，除非回应功能被关闭(而这是极少见的)。在按回车键之前，字符还未被内核保留，所以键入的错误可以用删字符和消行符纠正。当删字符和消行符前端有反斜杠时，内核取消反斜杠，并且不加解释地保留后续字符。

按下Return键之后，保留的字符被送到从终端读取字符的程序。程序也许会以特殊的方式依次解释字符，例如，假若字符的前面有反斜杠，shell就关闭专用的解释器。这在第3章会进一步探讨。总之应该认识到，内核处理删字符和消行符，而反斜杠只有在删字符和消行符之前，才会被内核处理；其他剩下的字符都可以被别的程序解释。

练习1-1 说明下面命令的结果：

```
$ date \@
```

练习1-2 多数shell(尽管第7版shell并非如此)把#解释为注释，并忽略从#开始至行尾的全部文字。按照这一点，说明下列的文本，假设删字符也是#：

```
$ date
Mon Sep 26 12:39:56 EDT 1983
$ #date
Mon Sep 26 12:40:21 EDT 1983
$ \#date
$ \\#date
#date: not found
$
```

1.1.7 继续键入

在键入的同时，内核读取所键入的内容，即使系统正忙于其他事务时也是如此，所以你可以用最快速度键入想输入的内容，即使有命令正在显示也没有关系。如果此时系统正在输出，那么输入字符会同输出字符混在一起，但它们会另行存储起来并以正确的次序解释。你可以一条接一条地输入命令，而无需等待它们完成或是开始。

1.1.8 中止程序

通过键入字符Delete，可以中止大多数命令。在某些终端上，Break也起作用，当然这是

与系统相关的。在少数程序中，如文本编辑器，Delete中止了程序的执行过程但仍留在该程序中。关闭终端或挂断电话会中止大多数的程序。

如果只打算暂停输出，例如，在要避免某些关键信息出现在屏幕上时，可以键入Ctl-s，输出会立即停下来，程序被挂起来直到再次启动。要打算继续输出，可键入Ctl-q。

1.1.9 注销

注销的正确方法是按下Ctl-d，而不用输入命令。这样就通知shell，输入中止了(至于它到底是如何起作用，下一章再作说明)。实际上，可关闭终端或挂断电话，但这样做是否真正注销了，取决于不同的系统。

1.1.10 邮件

系统提供一套邮件系统用于与其他用户通信，所以有时在登录时，在第一个提示符之前会看到消息

```
You have mail.
```

要阅读邮件，请键入

```
$ mail
```

邮件消息会打印出来，一次一条消息，最新的消息首先出现。在每一项后面，邮件等待用户的行动指示。有两种基本响应方式：d，删除该条消息；Return，不删除(消息会保留，下次可以阅读消息)。其他处理包括：p，打印消息；s filename(文件名)，以规定的名字保存消息；q，从mail中退出。(假如不知道文件这个概念，简单地把它看成一个用选定的名字保存信息的地方，以备今后再使用。文件是1.2节的主题，而且本书很多部分都在讨论它。)

同已经介绍的程序相比，mail可能有所不同，它有许多变量。请阅读手册以了解细节。

给他人发邮件很简单。假设要邮往登录名为nico的人，最方便的方法是这样的：

```
$ mail nico
```

在此键入邮件文本，可键入任意长度的文本，输入完毕后按

```
Ctrl-d
```

```
$
```

Ctl-d指明信件的结尾，通知mail没有输入了。如果在编辑信件的中途改变了主意，可以按Delete键，而不须使用Ctl-d。写到一半的信件将存放在名为dead.letter的文件中而不是邮出。

作为练习，给自己发封邮件，然后键入mail读取它(它不像听起来那么奇怪——这是一个简便的提醒机制)。

发邮件还有其他一些方法——发出一封事先准备好的信件，或一次给许多人同时发信，也可以发信给使用其他机器的人们。有关细节请参阅《UNIX程序员手册》第1节中的有关mail的叙述。今后我们将用标记mail(1)来表征手册第一节中有关mail的叙述。本章讨论的所有命令都可以在第1节中找到。

系统还有日历服务(参见(calendar(1)))；如果你没有使用过，我们会在第4章介绍如何设置它。

1.1.11 用户间通信

如果你的UNIX系统是多用户的，某一天，蓝色屏幕上可能会出现如下类似的语句

```
Message from mary tty7 . . .
```

并伴随着一声蜂鸣声。Mary打算写信给你，但是除非采取明确的行动，否则不能写回信。若要回应，请键入

```
$ write mary
```

这样，就建立了一条双向通信途径。现在Mary在她的终端上键入的内容，会出现在你的终端上，反之也一样，尽管通信很慢，但毕竟不象是同月球通话。

如果你正在进行某项工作，那么必须进入可以键入命令的状态。通常，正在运行的程序必须中止，但是有些程序，诸如编辑器以及write本身，利用“l”命令可暂时切换到shell——请参阅附录A中的表A-2。

write命令没有规则，所以要有一个协议，以免你键入的内容同Mary键入的内容混到一起。通常的做法是，每一方以(o)结束，它表明“完毕”，打算退出的信号是“oo”，其含义是“完毕并退出”。

Mary的终端:

```
$ write you
```

```
Message from you ttya...
did you forget lunch? (o)
```

```
ten minutes (o)
ok (oo)
```

```
EOF
ctl-d
```

```
$
```

你的终端:

```
$ Message from mary tty7...
write mary
```

```
did you forget lunch? (o)
five@
ten minutes (o)
```

```
ok (oo)
ctl-d
```

```
$ EOF
```

要退出write也可以按Delete键。注意，键入的错误不会在Mary的终端上出现。

如果要写信给某个未登录的人，或者收件人不想被打扰，系统会通知你。如果目标登录了，但是在一个间隔后没有回答，也许对方太忙或者不在终端旁，只要按Ctl-d或Delete键就可退出。如果你不想被打扰，可利用mesg(1)。

1.1.12 新闻

许多UNIX系统提供了一种新的服务，可使用户随时了解一些有趣无趣的事件。试试键入

```
$ news
```

也有大量的UNIX系统网络存在，它们可以通过电话拨号连接，有关netnews和USENET，请询问你身边的专家。

1.1.13 手册

《UNIX程序员手册》描述了大量读者需要了解的系统信息。第1节介绍命令，包括本章讨论的内容。第2节讨论系统调用，这是本书第7章的主题，而第6节是关于游戏的内容。其余章节讨论C程序员使用的函数、文件格式，以及系统维护(这些章节的编号随着系统的不同而变

化)。在开始使用手册时,不要忘记使用各种形式的手册索引。可以快速地翻阅手册索引,以了解相关命令的内容。还有关于UNIX系统运行的原理。

通常手册用在线形式存放,这样可以在终端上阅读。如果你不知道做法,又找不到专家协助,可以在终端上用命令man打印出有关页面。下面例子是读取who命令:

```
$ man who
```

而

```
$ man man
```

则说明了man命令。

1.1.14 计算机辅助教学

你的系统中也许有learn命令,这个命令提供了有关文件系统、基本命令、编辑器、文档准备,甚至包括C语言程序设计的计算机辅助教学。请试一下

```
$ learn
```

它会告诉你从哪儿开始,可以做什么。如果不行,可以再试一下teach。

1.1.15 游戏

游戏往往不被正式承认,但熟悉计算机和终端的最好方式之一,就是玩游戏。UNIX系统提供了适量的游戏,通常在本地上就有。问问周围的人,或者查看手册的第6节。

1.2 文件和常用命令

在UNIX系统中信息存储在文件中,它很像日常的办公室文件。每个文件有名字,内容,存放地点以及某些管理信息,诸如所有者以及文件大小等。文件可能是一封信,或者是人名及地址清单,或者是源程序,或者是供某个程序用的数据,甚至是程序的可执行形式以及其他非文本类型材料。

UNIX文件组织结构使你可以维护自己的文件而不会妨碍其他人的文件,并且也防止他人干预你的文件。有大量的程序可对文件进行操作,但是现在,我们只介绍最频繁使用的那些命令。第2章是关于文件系统的具体讨论,并引入了许多与文件有关的其他命令。

1.2.1 创建文件

如果想录入一篇文章、一封信或一个程序,那么怎样把这些信息存放在机器中呢?这类任务大多是用文本编辑器完成的,它是一个在计算机中存放和操作信息的程序。在每个UNIX系统中几乎都有一个屏幕编辑器,它利用现代终端的特点,显示你对文件所进行的编辑效果。两个最流行的文本编辑器是vi和emacs。在这里我们将不介绍任何屏幕编辑器,其原因部分是由于篇幅所限,部分是由于不存在一个标准的文本编辑器。

不过,有一个名为ed的编辑器肯定在你的系统中存在。它不需要特定的终端功能,因此可以在任何终端上工作。它也构成了其他基本程序的基础(包括一些屏幕编辑器),所以这个程

序实在是值得学习的。附录A中有关于这个程序的简要说明。

无论我们偏好哪个编辑器，都得熟练掌握它以便能够创建文件。这里我们以ed为例来进行讨论，这也可保证你在自己的系统中完成这些例子，当然你可以使用自己最喜好的编辑器。

要用ed建立一个文件junk，可按如下操作进行：

<code>\$ ed</code>	调出文本编辑器
<code>a</code>	用以加入文本的ed命令
键入文本.....	
<code>.</code>	键入.以停止增添文本
<code>w junk</code>	将文本内容写入文件junk
<code>39</code>	ed显示出写入的字符数
<code>q</code>	退出ed
<code>\$</code>	

命令a(append)通知ed开始接收文本。.指明文本结束，但必须出现在一行的开始处。不要忘掉这个命令，否则任何其他ed命令都不会被识别——所键入的一切都会被认为是所添加的文本。

编辑命令w(write)存储所键入的信息。w junk 是把内容存放在名为junk的文件中。文件名可以任意起(之所以选择junk这个词，是因为该文件不重要)。

ed回送键入到文件中的字符数。在w命令之前，不会有永久性的存储，所以如果退出系统下班回家，有关的信息不会存储到文件中去(如果在编辑时退出系统，所有的数据会存放在名为ed.hup的文件中，这样下一次可以继续使用)。如果正在编辑时系统崩溃(比如由于软件或硬件的故障而突然停机)，该文件只存放最后一次用写命令时所保存的内容。而在w之后，信息就永久地保存了。以后可以键入如下命令访问该文件：

```
$ ed junk
```

当然，你可以编辑所键入的文本，纠正拼写错误，修改用词，重新排列段落等等。当这些都完成之后，可用q(quit)命令退出编辑器。

1.2.2 列出文件

以下创建两个文件，junk和temp：

```
$ ed
a
To be or not to be
.
w junk
19
q
$ ed
a
That is the question.
.
w temp
22
q
$
```

字符数从ed开始统计到每一行的结尾，结尾称为newline，它是系统表征回车键Return用的。

ls命令可列出文件的名字(并非内容):

```
$ ls
junk
temp
$
```

这就是刚创建的那两个文件(当然也还会有他人建立的其他文件)。名字自动按字母顺序排列。

如同多数命令一样, ls有各种选项用以替代默认功能。在命令行中选项跟随着命令名, 并通常由-和一个代表其含义的字母构成。例如, ls -t使文件按“时间”顺序排列, 最后一次修改的文件, 最先列出。

```
$ ls -t
temp
junk
$
```

-l给出了一个“长”的列表, 为每个文件提供了更多信息:

```
$ ls -l
total 2
-rw-r--r-- 1 you      19 Sep 26 16:25 junk
-rw-r--r-- 1 you      22 Sep 26 16:26 temp
$
```

total 2说明文件占据硬盘的数据块数量, 一个数据块通常有512或1024个字符。字符串-rw-r--r--说明读写权限。在本例中, 所有者(你自己)可以读写, 但其他人只能读。后面跟随的1是该文件的链接数量, 将在第2章讨论。you是文件的所有者, 即创建文件的人。19和22是对应文件的字符数, 它们同从ed所得到的数字是一致的。而日期和时间则说明了文件是何时进行最后修改的。

选项可以组合在一起: ls -lt会得到同ls -l一样的数据, 但是按照较新文件先列出的原则分类。-u选项说明文件何时被使用: ls -lut按照最新使用的次序, 列出一个长(-l)的清单。-r选项反转输出的顺序, 所以ls -rt按最早使用的次序排列。也可以只列出感兴趣的文件名字, 命令如下:

```
$ ls -l junk
-rw-r--r-- 1 you      19 Sep 26 16:25 junk
$
```

在命令行中命令后面的字符串, 诸如上面例子中的-l和junk, 称为程序变量。变量通常是命令所使用的文件选项或名字。

通常的惯例是用负号和单个字母指定选项, 诸如-t, 或字母组合-lt。一般而言, 如果命令接收这类选项, 它们应置于文件名变量之前, 但可以按任意的次序出现。UNIX程序在对多选项的处理上是不一致的。例如, 标准第7版的ls不会接受如下写法

```
$ ls -l -t
```

在第7版中无效

作为ls -lt的同义词, 而在别的程序中, 多重选择项则要求被分开书写。

在学习更多例子之后, 你会发现对于选择变量不存在什么规则或系统性。每个命令都有其各自的语法规则, 以及关于字母的各种含义(通常同一个功能在不同的命令中有不同的含义)。这种不可预见的行为使人不适应, 而且常常被看作系统的缺陷而加以引证。尽管情况正在改

善——新的版本通常有较好的一致性——我们仍旧建议，在编写程序时尽可能做得更好些，同时在手边保留一份手册。

1.2.3 打印文件

若现在有了一些文件，那么怎么查看其内容呢？有太多的程序可以完成这项工作。一种方式是使用编辑器：

```
$ ed junk
19          ed报告在junk中有19个打印字符
1,$p       从第1行打印到最后一行
To be or not to be  文件只有一行
q          完成
$
```

ed先报告junk中的字符数目。命令1,\$p通知打印出文件中所有的行，在明确如何使用编辑器后，你可以选择要打印出的部分。

有时候使用编辑器打印并不太合适。举例来说，编辑器ed有个处理文件大小的限制——仅限于数千行的文件。另外，它一次只能打印一个文件，而有时可能需要打印若干个文件，一个接一个，中间不间断。所以这里介绍几个其他办法。

首先是cat，在所有打印命令中，它是最简单的。cat打印其变量列出的所有文件名的内容：

```
$ cat junk
To be or not to be
$ cat temp
That is the question.
$ cat junk temp
To be or not to be
That is the question.
$
```

给定名字的文件在终端上被连接(catenated[⊖]，从而有了名称cat)在一起，其间一个接一个相连，没有间隙。

对短小的文件这么做没有问题，但是对于较长的文件，如果计算机连接的速率较快，则必须用ctl-s迅速停止由cat导出的输出，否则它们就全部从屏幕上流逝了。尽管每个UNIX系统几乎都有这样的命令，但并不存在一个“标准”的命令，可以在视频终端上一次一屏地打印一份文件。你所用的系统中也许有一个称为pg或者more的命令。我们所用的是p，在第6章中将介绍它。

类似于cat，pr命令也可打印出文件名清单中文件的全部内容，但它是以一种在行式打印机上适用的模式打印：每页66行(11英寸)，附有最后一次修改文件的日期和时间，页码，页眉上的文件名，以及跨越两页间的附加行。这样，可以只打印junk，然后跳到新的一页开始打印temp：

```
$ pr junk temp

Sep 26 16:25 1983   junk page 1
```

⊖ catenate是concatenate的一个偏僻的同义词。

```
To be or not to be
      (60多行空行)
```

```
Sep 26 16:26 1983    temp page 1
```

```
That is the question.
      (60多行空行)
```

```
$
```

pr可以输出多列格式

```
$ pr -3 文件名清单
```

这表示以3列的形式打印每个文件。你可以使用其他合理的列数，pr会尽可能完成。pr -m可以按并行列的形式，打印若干文件。请参见pr(1)。

需要注意的是，pr并不是可以重新排列行及调整边距的排版程序。真正的排版程序是nroff以及troff，将在第9章讨论。

还有可以在高速打印机上打印文件的命令。请在手册中查看名称类似为lp及lpr的命令，或者在手册索引中查找“printer(打印机)”。至于使用哪一个命令，取决于连接到系统的设备是什么。pr和lpr经常一起用，用pr对信息做合适的格式处理，再用lpr将其传送至行式打印机上。稍后，还会讨论这些内容。

1.2.4 移动、复制和删除文件

现在来看看其他命令。首先是修改文件的名称。改名是通过把文件名“变动”到另一个名称来完成的，比如：

```
$ mv junk precious
```

其作用是把名为junk的文件改名为precious；而文件内容并没有改变。如果再运行ls，会看见一个不同的清单，junk没有了，而precious出现了。

```
$ ls
precious
temp
$ cat junk
cat: can't open junk
$
```

注意，如果把一个文件移往另一个已存在的文件，目标文件会被替换掉。

要复制一个文件副本(也就是，需要某文件的两个版本)，可以使用cp命令：

```
$ cp precious precious.save
```

对precious 制作一个名为precious.save的副本。

最后，如果你已厌倦了文件的创建和移动操作，rm命令可以清除列出的所有文件：

```
$ rm temp junk
rm: junk nonexistent
$
```

如果要删去的某个文件并不存在，会出现警告，除此以外rm在像多数UNIX命令一样安静地工作。没有提示符，也没有对话，出错信息很简洁，有时甚至没有用。太简单的表达对于

初学者而言有些困难，而经验丰富的用户却认为对话多的命令让人厌烦。

1.2.5 文件名

到目前为止，我们并没有说明一个合法文件名应符合哪些要求，现在是说明这些规则的时候了。首先，文件名长度不应超过14个字符。其次，尽管可以选择几乎所有的字符，但通常应选择那些明显的，并避免可能引起歧义的字符。例如，我们已看见在ls命令中，ls-t意味着按时间顺序列出文件。所以，如果有一个名为-t的文件，那么把它按名称列出来就很困难。(怎样做?)除了把负号作为首字符之外，还有一些字符也有特别的含义。在熟练掌握它们之前，为避免弄混，应该只使用字母、数字、点号和下划线(点号和下划线通常用于把文件名分成若干部分，如前例中的precious.save)。最后，不要忘了大小写敏感方式——junk、Junk和JUNK是三个不同的名称。

1.2.6 有用的命令

我们有了创建文件，列出文件名称并打印其内容的基本命令，现在我们可以学习几个文件处理的命令了。为使讨论更集中，将使用Augustus De Morgan的诗歌当作名为poem的文件。先用ed创建该文件：

```
$ ed
a
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
  and so ad infinitum.
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
.
w poem
263
q
$
```

下面第一个命令用于计算文件的行数、词数和字符数。类似于计数命令w，我们把这一命令称为wc(word-counting)：

```
$ wc poem
      8      46     263   poem
$
```

也就是说，poem有8行，46个字，263个字符。“字”的定义很简单：任何不含空格、跳格键和新行的字符串。

wc可以为多个文件计数(并打印全部结果)，它也可以在需要时，略去计数功能。参见wc(1)。

第二个命令为grep，它搜寻一个文件中符合给定匹配模式的行(该名称来源于ed命令“g/regular-expression/p”，这在附录A中解释)。假如打算在poem中寻找字fleas：

```
$ grep fleas poem
Great fleas have little fleas
```

```

And little fleas have lesser fleas,
And the great fleas themselves, in turn,
    have greater fleas to go on;
$

```

grep在使用-v选项时, 会寻找所有不匹配给定模式的行(它来源于编辑器命令v, 可以认为, 它是匹配的逆向使用)。

```

$ grep -v fleas poem
    upon their backs to bite 'em,
    and so ad infinitum.
While these again have greater still,
    and greater still, and so on.
$

```

grep可以用于在多个文件中查询。此时, 对每一查询的结果行, 会加上文件名做为前缀, 这样就知道该匹配来源于哪个文件。另外grep还有计数、编号等功能。它还能处理比一个字复杂得多的模式匹配, 我们将在第4章中讨论该问题。

第3个命令是sort, 它把输入按字母次序一行行地排序。对于诗句, 这样做不是很有意义, 但我们也进行一下处理, 从而可以看看结果:

```

$ sort poem
    and greater still, and so on.
    and so ad infinitum.
    have greater fleas to go on;
    upon their backs to bite 'em,
And little fleas have lesser fleas,
And the great fleas themselves, in turn,
Great fleas have little fleas
While these again have greater still,
$

```

排序是逐行的, 但默认排序次序依次是空白行、大写字母、小写字母, 所以它并不是严格地按字母排序的。

sort有很多控制排序方式的选项——逆序、数字序、字典序、略去前异空格序、按行内的字段排序等——不过通常只需要掌握部分选项, 以下是最常用的选项:

sort -r	按正常排序反序
sort -n	按数字次序排序
sort -nr	按逆向数字排序
sort -f	将大小写混合在一起排序
sort +n	从第n+1个字段开始排序

第4章会进一步讨论sort。

另一个文件检查命令是tail, 它打印文件的最后10行。所以这一命令对于上面的8行诗来说是太多了, 但它对于较大型的文件是有用的。另外, tail有指定行数的选项。这样可以打印poem的最后一行:

```

$ tail -1 poem
    and greater still, and so on.
$

```

tail也可以用来指定从某行开始打印文件:

```
$ tail +3 文件名
```

从第3行开始打印文件(请注意, 作为变量的负号通常有逆向含义)。

最后一对命令用于比较文件。假设有一个poem的变种文件new_poem:

```
$ cat poem
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
  and so ad infinitum.
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
$ cat new_poem
Great fleas have little fleas
  upon their backs to bite them,
And little fleas have lesser fleas,
  and so on ad infinitum.
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
$
```

这两个文件没有多大的区别，实际上，要仔细观察才能找出差别。这就是要使用比较命令的原因了。cmp找出两个文件不同处的第一个有区别的地方：

```
$ cmp poem new_poem
poem new_poem differ: char 58, line 2
$
```

该结果说明，两文件不同之处在第2行，这是确实的，但它没有说明其差别是什么，而且除了第一个差别之外，它就不再作其他的比较了。

另一个文件比较命令是diff，它报告所有被修改、增加或减少的行：

```
$ diff poem new_poem
2c2
<  upon their backs to bite 'em,
---
>  upon their backs to bite them,
4c4
<  and so ad infinitum.
---
>  and so on ad infinitum.
$
```

该结果说明，在第一个文件(poem)的第2行已被修改成了第2个文件(new-poem)的第2行，而第4行也存在不同之处。

一般而言，cmp用于判断两个文件是否确实内容相同。它工作快，适用于任何类型的文件，而不仅仅是文本文件。diff用于已知道文件之间可能有所差别，进而准确地想知道哪些行有什么不同的情况，它只适用于文本文件。

1.2.7 文件系统命令小结

表1-1列出了到目前为止已讨论的处理文件的命令。

表1-1 常用文件系统命令

命 令	含 义
ls	列出当前目录的文件名
ls 文件名	只列出指定的文件

(续)

命 令	含 义
ls -t	按时间次序列出文件, 时间最近的文件排在最前面
ls -l	长列表, 包括较多信息; 还可使用 ls -lt
ls -u	按最后使用的时间次序列出文件; 也可使用 ls -lu , ls -lut
ls -r	以逆序列出文件; 可组成 -rt , -rlt 等等
ed 文件名	编辑文件
cp 文件1 文件2	把文件1拷贝到文件2, 若文件2存在, 则覆盖文件2
mv 文件1 文件2	文件1换名为文件2, 若文件2存在, 则覆盖文件2
rm 文件名	删除文件, 不能恢复
cat 文件名	打印文件
pr 文件名	按打印机格式(表头, 每行66页)打印文件
pr -n 文件名	以n列形式打印文件
pr -m 文件名	并排打印指定的若干文件(多列打印)
wc 文件名	统计文件的行、单词(字)和字符
wc-l 文件名	统计每个文件的行
grep 模式 文件名	打印匹配模式的行
grep-v 模式 文件	打印不匹配模式的行
sort 文件名	逐行按字母顺序排列
tail 文件名	打印文件最后10行
tail -n 文件名	打印文件最后n行
tail +n 文件名	从第n行开始打印
cmp 文件1 文件2	打印两个文件差异第一处
diff 文件1 文件2	打印两个文件所有不同之处

1.3 目录

系统可以把你的名为junk的文件同任何其他人的相同名字的文件区分开。这个区分是依靠把文件划分到各个目录中, 而不是像图书馆中那样把书放在架上, 所以在不同目录中的文件可有相同的名字而不至于出现矛盾。

通常, 每个用户都有其个人或主目录, 有时也称为登录目录, 其中只包含了属于用户的文件。登录时用户位于自己的主目录中。用户可以改变工作用的目录——通常称为用户的工作目录或当前目录——但用户的主目录始终不会变化。除非用户特别指明, 当创建一个新文件时, 它会位于用户主目录中。这是用户主目录初始特性, 文件不会与其他人的相同名称的文件相关。

目录中可以包含其他的目录, 当然也可包含一般的文件(“好目录中有较少的目录……”), 描述这一结构的自然方式是目录和文件树。可以从树根开始沿着恰当的分支移动, 从而找到系统中的任何文件。相反, 也可以从任何一处开始, 最终到达根部。

我们先从后者开始。所用的基本工具是pwd(print working directory), 它打印用户当前所在的目录名:

```
$ pwd
/usr/you
$
```

它说明用户当前在目录you中, you在usr目录中, 而usr就在根目录中, 根目录通常用/表示。/

字符分隔了名称的部件，前面所述的应用于文件名的14个字符的限制也适用它。在很多系统中，`/usr`是包含系统所有普通用户目录的一个目录。（即使用户的主目录不是`/usr/you`，`pwd`也会打印类似的内容，所以用户能够理解下面的内容是什么。）

如果用户键入

```
$ ls /usr/you
```

应该得到如同使用`ls`所得到的完全一样的名单。在没有提供变量时，`ls`列出当前目录的内容；如果给定目录的名称，它就列出该目录的内容。

其次，试一试

```
$ ls /usr
```

这应该列出一串目录名称，其中有你自己登录的目录。

下一步是试验列出根本身。可以得到一个类似于下面的响应：

```
$ ls /
bin
boot
dev
etc
lib
tmp
unix
usr
$
```

（不要混淆`/`字符的两个含义：它既是根的名称，又是文件名的分隔符）。它们多数是目录，但`unix`实际是一个含有UNIX内核的可执行形式的文件。有关它的更多内容在第2章中讨论。

现在试一下

```
$ cat /usr/you/ junk
```

（如果`junk`仍在你的目录中。）

```
/usr/you/ junk
```

称作文件的路径名。路径名表征了从根通过目录树到特定文件的全部路径。在UNIX系统中有一个通用规则，就是既可以使用普通的文件名，也可以使用路径名。

文件系统构造得像一个通用逻辑树，用一幅图表示可能会更清晰一些：

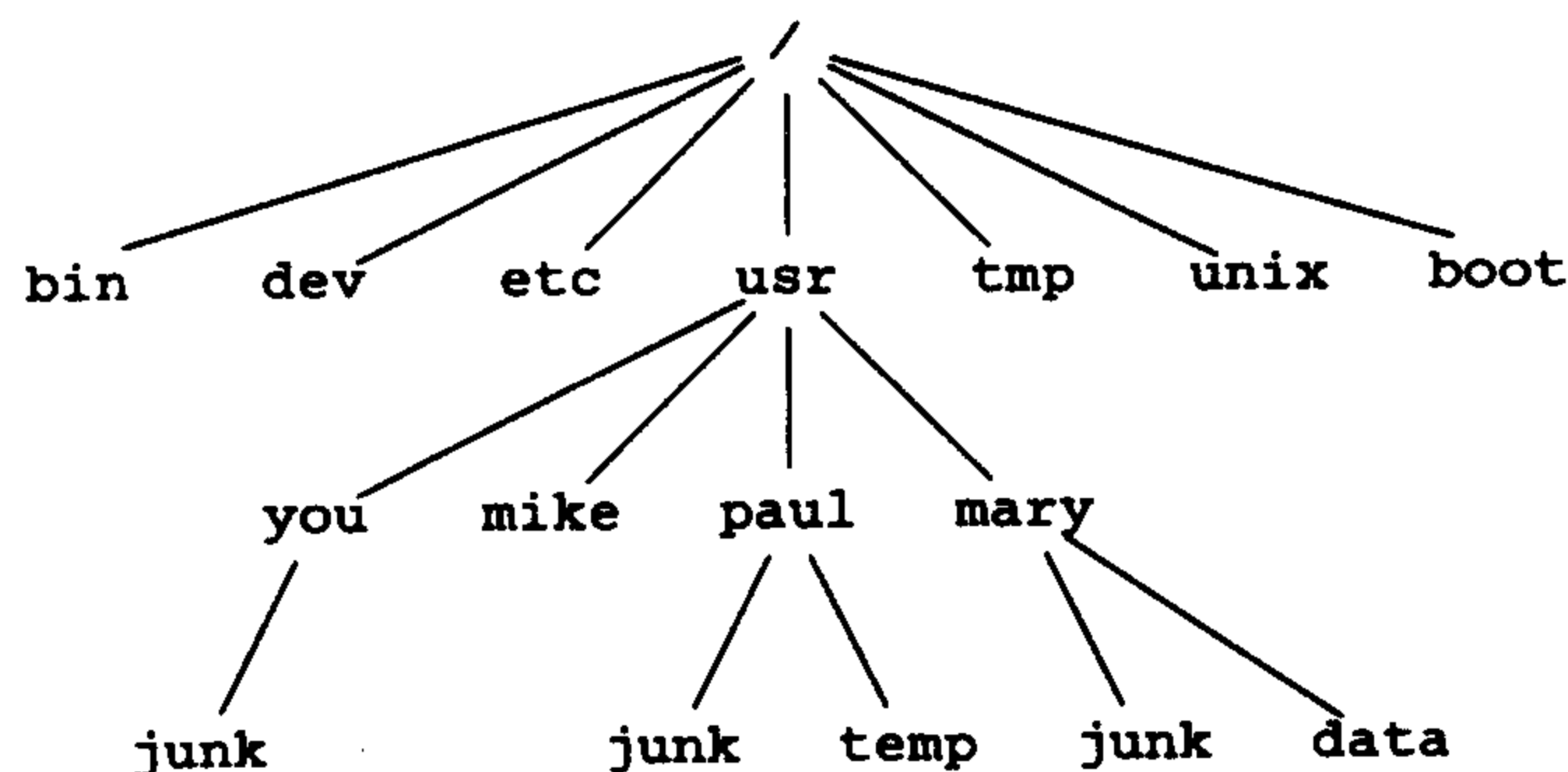


图1-1 文件系统构造

从图中可以看出，你的junk文件同Paul的或Mary的junk文件没有任何关系。

如果所关心的所有文件都在你自己的目录中，那么路径名并没有多少作用，但是如果你同其他人合作，或者同时在进行几个项目，那么路径名就很有用了。例如，你的朋友可以通过如下操作打印你的junk文件：

```
$ cat /usr/you/junk
```

类似地，你可以用如下操作找出Mary文件的内容

```
$ ls /usr/mary
data
junk
$
```

或者对她的文件制作一份自己的副本

```
$ cp /usr/mary/data data
```

或者编辑她的文件：

```
$ ed /usr/mary/data
```

如果Mary不想让你搅和她的文件，那么可以做出私用性设置。每个文件和目录都有对应其所有者、组以及其他人的读-写-执行的允许权限，用以控制存取(请回忆ls -l)。在作者的系统中，多数用户在多数时间里发现开放性还是比私用性更为有益，但是在你的系统中可能有不同的策略，有关内容会在第2章中讨论。

作为路径名有关试验的最后一个，请试一下

```
$ ls /bin /usr/bin
```

有什么名称看似熟悉吗？当在提示符后键入命令名，然后运行命令时，系统寻找具有该名称的文件。它通常先寻找当前目录(也许多数不会在此目录中)，然后在/bin中寻找，最后是在/usr/bin中寻找。目录收集在一起是为了便于查询和管理。要验证这一点，通过使用它们的全路径名执行下面这些程序：

```
$ /bin/date
Mon Sep 26 23:29:32 EDT 1983
$ /bin/uho
srm      tty1   Sep 26 22: 20
cvw      tty4   Sep 26 22: 40
you      tty5   Sep 26 23: 04
$
```

练习1-3 尝试

```
$ ls /usr/games
```

并按所要求的执行。可能会有比通常的上班更为有趣的事发生。

改变目录

如果你经常与Mary一起处理她的主目录中的信息，你可以说“我打算在Mary的文件上而不是在自己文件上工作”。这是通过用cd命令改变当前目录而实现的：

```
$ cd /usr/mary
```

现在，当你使用文件名(不带/)作为cat或pr的变量时，它会引向Mary的目录。改变目录并不会影响文件有关的允许权限——如果不能从自己的目录中访问某一文件，改变至另一个目录并

不能改变这一事实。

把同某一事物有关的文件都安排在一个目录中而与其他项目分隔开会带来便利。例如，如果打算写一本书，也许可以把所有的文件都放在一个称为book的目录中。命令mkdir建立一个新的目录：

```
$ mkdir book           新建一目录
$ cd book             进入该目录
$ pwd                 确认在正确位置
/usr/you/book

...
$ cd ..               写书(过了若干分钟)
                        在文件系统中上移一层
$ pwd
/usr/you
$
```

..表示当前所在目录的父目录，即更接近根一层的目录。.指当前目录。

```
$ cd      返回主目录
```

上述操作把你带回主目录，即登录目录。

一旦你的书出版了，就可以清除掉有关文件。要取消目录book，删去其中的所有文件(我们会简单地展示)，再用cd指向book的父目录，请键入：

```
$ rmdir book
```

rmdir只删去空目录。

1.4 shell

当系统出现提示符\$，你可键入命令并得到执行，此时并不是内核在与你对话，而是与一个称为命令解释器或外壳shell的对话。shell是同date或who一样的普通程序，尽管它可以处理一些有特色的事务。shell存在于你和内核机制之间是有益的，有些会在这里说明。它有三个主要优点：

- 文件名简写：可以通过指令文件名的模式来选取一套文件名作为程序的变量——shell会找出匹配该模式的文件名。
- 输入输出重定向：可以把任何程序的输出送到一文件中而不是终端上，并当作来自文件的而不是终端的输入。输入和输出甚至可以连接到其他程序上。
- 环境个性化：可以定义自己的命令和同义词。

1.4.1 文件名简写

现在讨论文件名模式。假设键入了许多文档，像书一样多。逻辑上，会把它分成许多小部分，类似于章、节一类。从物理上看，也应该被划分开，因为编辑一个大文件是很麻烦的，所以应该把文本作为一批文件键入。可以每章建立一个文件，称为ch1、ch2，等等。或者每一章拆分成节，建成如下的文件：

```
ch1.1
ch1.2
ch1.3
...
```

```
ch2.1
ch2.2
...
```

这也是用于本书的结构。由于有一个系统的命名惯例，只要看一眼就可说出某一个特定的文件在整个系统中的位置。

如果打算打印全本书，那么该怎么办呢？可以执行

```
$ pr ch1.1 ch1.2 ch1.3...
```

但这样很快就使人厌烦，而且容易出现错误。这就是要使文件名简写的原因。如果键入

```
$ pr ch*
```

shell认为*的含义是“任何字符串”，所以ch*是一个匹配当前目录中的所有以ch开头的文件的一个模式。shell会按字母顺序建立清单^①，并把清单传递给pr。pr命令决不会看到*，shell程序在当前目录中的模式匹配生成一个传递给pr字符串清单。

关键之处在于，文件名简写并不是pr命令的特性，而是shell的一项服务。所以你可以用它为任何命令生成文件名序列。例如，要计算第1章的字数：

```
$ wc ch1.*
  113    562   3200 ch1.0
  935   4081  22435 ch1.1
  974   4191  22756 ch1.2
  378   1561   8481 ch1.3
 1293   5298  28841 ch1.4
   33    194   1190 ch1.5
   75    323   2030 ch1.6
 3801  16210  88933 total
$
```

有一个名为echo的程序特别有必要同简写字符共同试验。正如你所猜测的，echo除了照搬其变量之外，什么也不做：

```
$ echo hello world
hello world
$
```

但是该变量可以由模式匹配所生成：

```
$ echo ch1. *
```

它列出第1章中的所有文件名，而

```
$ echo *
```

以字母顺序列出在当前目录中的所有文件名，

```
$ pr *
```

打印按字母顺序排列的所有的文件，而

```
$ rm *
```

则把当前目录中的文件全部清除。（最好确认打算做什么？）

*并不是只能用在文件的最后位置，它可以放在任何位置，并可以使用多次。如

```
$ rm *.save
```

① 顺序也并不是严格按字母序的，实际上大写字母先于小写字母。对于在sort中所使用字符的顺序，请参考ascii(7)。

把所有结尾为save的文件除去。

注意，文件名按字母顺序排序同按数字排序是不一样的。如果一本书有10章，次序可能不是你当初所想要的，因为ch10在ch2之前：

```
$ echo *
ch1.1 ch1.2 ... ch10.1 ch10.2 ... ch2.1 ch2.2 ...
$
```

*不是shell提供的唯一匹配模式，尽管它被频繁地使用。模式[...]匹配括号内的任何字符。连续的字母或数字范围可以简写为：

```
$ pr ch [12 346789] *      打印1、2、3、4、6、7、8、9章，但没有第5章。
$ pr ch [1-46-9]          同上
$ rm temp [a-z]            删除存在的任何tempa, ..., tempz
```

? 模式匹配任何单个字符：

```
$ ls ?                    列出单个字符的文件
$ ls -l ch?.1             列出ch1.1、ch2.1、ch3.1等等，但没有ch10.1
$ rm temp?                删除templ, ..., tempa, 等等
```

注意模式只匹配已存在的文件。特别地，不能用模式建立新的文件名。例如，如果打算把每一章的ch扩展为chapter，不能采用如下方法：

```
$ mv ch.* chapter.*      不工作!
```

因为没有与chapter.*匹配的已存在文件。

模式字符如*之类既可用于单个文件名也可用于路径名，模式按含有特殊字符路径的每个分量进行。这样/usr/mary/*完成在/usr/mary中的匹配，而/usr/*/calendar生成所有用户的calendar文件的路径名清单。

如果打算关闭*、?等等的特殊作用，可以把整个变量放在单引号内，如下所示

```
$ ls '?'
```

也可以在特殊字符前加上一个反斜杠：

```
$ ls \?
```

(注意，由于?不是删除符也不是消行符，该斜杠由shell解释，而不是由内核解释)有关引号的讨论在第3章中进行。

练习1-4 以下这些命令有何不同？

```
$ ls junk                $ echo junk
$ ls /                   $ echo /
$ ls                     $ echo
$ ls *                   $ echo *
$ ls '*'                 $ echo '*'
```

1.4.2 I/O重定向

到目前为止所讨论的命令多数可在终端上产生输出，如编辑器，也从终端上取其输入。而终端可以被一个文件所替代供输入或输出，举例来说，

```
$ ls
```

在你的终端上列出文件名。但是，如果说

```
$ ls >filelist
```

那么同样的文件名清单会放在filelist文件中。符号>含义为“把输出写到如下文件中，而不是在终端上”。如果该文件原先不存在，则创建它，如果原先已存在，则该文件会被重写，在终端上什么也没创建。另一个例子是，通过把cat的输出都写进一个文件的途径，可以把若干个文件组合成一个文件：

```
$ cat f1 f2 f3 > temp
```

符号>的操作与>类似，但它的含义是“添加在某文件的尾部”。也就是说

```
$ cat f1 f2 f3 >>temp
```

把f1, f2以及f3 的文件内容复制到temp文件原有文件的结尾之后，而不是重写原有的内容。在使用>时，如果temp原先不存在，它会创建一个新的空文件。

相似地，<的含义是程序从文件中取输入，而不是从终端上取。这样，就可以在文件let中准备一封信函，然后用下列方法发送给若干人

```
$ mail mary joe tom bob <let
```

在所有这些例子中，在>或<两边的空格是可选的，但本书中使用的格式是传统格式。

用>获得了重定向输出的能力，就有可能把命令组合起来以实现原来不可能达到的效果。例如，要打印用户的字母顺序清单，可以

```
$ who >temp
$ sort <temp
```

由于who为每一个登录用户打印一行输出，而wc-l计算行数(不使用计算字数和字符数功能)，于是就可以计算用户数，这可用如下方法：

```
$ who >temp
$ wc -l <temp
```

也可用下列方式统计当前目录中文件数：

```
$ ls >temp
$ wc -l <temp
```

用这个方法，计数中包含temp自身。用下列方法可以以3列形式打印文件名：

```
$ ls >temp
$ pr-3 <temp
```

而通过组合运用who以及grep，可以看到登录的某一特定用户：

```
$ who >temp
$ grep mary<temp
```

在所有这些例子中，在使用诸如 * 一类文件名模式匹配符时，重要的一点是请记住，它们是由shell解释的，而不是由各个程序来解释的。功能的集成化使得shell的这一输入和输出重定向功能可供任何程序使用，而这些程序自身并不知道发生了什么不寻常的事。

这样就引出了一项重要的惯例。命令

```
$ sort <temp
```

对文件temp的内容进行排序，而

```
$ sort temp
```

做类似的工作，但有所差别。在前一个例子中，因为字符串 <temp 由shell解释，sort并没有把

文件名 temp 看成一个变量，它只是对它的标准输入进行了排序，该输入来自一个文件，并由 shell 进行了重定向。然而后一个例子，把名字 temp 作为变量传递给了 sort，由 sort 读入文件并对其进行排序。sort 可以被指定一个文件名清单，如

```
$ sort temp1 temp2 temp3
```

但是如果没有指定文件名，它就对其标准输入进行排序。这是大多数命令的一个基本特性：如果没有指定文件名，就对标准输入进行处理。这里，你可以直接键入命令，观看它们如何工作。例如

```
$ sort
ghi
abc
def
ctl-d
abc
def
ghi
$
```

在下一节中，将讨论这一原理如何应用。

练习1-5 解释为什么

```
$ ls >ls.out
```

导致 ls.out 被包括在名单中。

练习1-6 解释来自如下的输出

```
$ wc temp >temp
```

如果拼错了命令名，比如

```
$ woh >temp
```

会发生什么？

1.4.3 管道

上一节中所有例子都依赖于相同的技巧：把一个程序的输出，通过一个临时文件送到另一个文件的输入，而该临时文件并没有其他的作用。实际上，使用这样一个文件是很笨拙的。这导致了UNIX系统基础性的贡献之一——管道的想法。管道是不使用临时文件而把一个程序的输出直接连接到另一个程序的输入的途径，管道线是通过管道对两个或两个以上的程序的连接。

下面通过管道而不是临时文件重新检查前面的一些例子。垂直线字符 | 通知 shell 建立一条管道线：

\$ who sort	打印已排序的用户名单
\$ who wc -l	统计用户数
\$ ls wc -l	统计文件
\$ ls pr -3	3列式文件名列表
\$ who grep mary	寻找特定用户

任何从终端读取的程序都可改由管道读取，任何写到终端上的程序都可以改为写向管道。当没有指定文件名时，任何读取标准输入的程序均可使用管道线。grep、pr、sort 以及 wc 都以

上述方式使用管道线。

在管道线中可以有任何多个程序：

```
$ ls / pr-3 / lpr
```

此行命令，在行式打印机上输出3列形式的文件名清单，而

```
$ who / grep mary / wc -l
```

统计Mary登录的次数。

管道线上的程序实际上是同时运行，而不是一个接一个地运行。这意味着管道线中的程序是交互式的，内核对调度和同步进行处理，以使它们全部运行。

当需要使用管道时，shell会作出安排，显然各个程序要进行重定向。当然，此时程序必须能够有所反应。大多数命令遵循一个公共格式，所以它们多半能在任何位置上适应管道的要求。通常，命令按如下方式实施：

命令 变量选项 文件名选项

如果没有指定文件名，则命令读进标准输入，其缺省设备是终端(便于从事试验)，但输入可以被重定向为来自别的文件或者管道。同时，在输出一方，多数命令把输出写到标准设备，其缺省设备也是终端，但它也可以重定向到文件或者管道。

命令的出错消息有不同的处理方式，它们也许会在文件或管道中消失。因此每一个命令都有一个标准错误输出，它通常被导向终端。如图1-2所示：

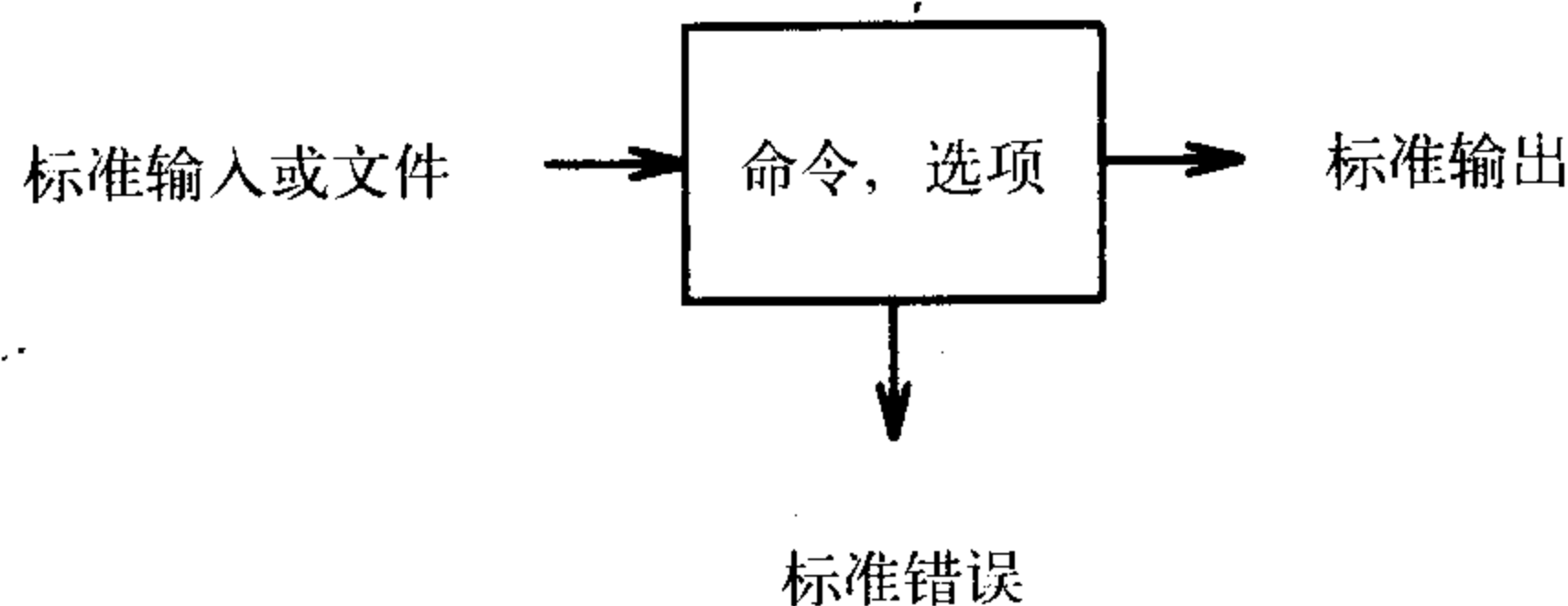


图1-2 命令标准I/O模式

到现在为止，所讨论过的命令几乎都符合这一模式，唯一例外的是类似date和who这种命令，它们没有输入；还有一些如cmp及diff，它们有固定数量的文件输入。(请注意它们的 - 选项。)

练习1-7 解释如下两者的不同

```
$ who / sort
```

与

```
$ who > sort
```

1.4.4 进程

shell除了设置管道之外，它还处理大量的工作。现在让我们来看看同时运行多个程序的基本情况，前面我们已经看到了管道中存在这种情况。例如，可以在一个命令行中运行用分号分隔的两个命令，shell会识别分号并把命令分成两个命令：

```
$ date; who
Tue Sep 27 01:03:17 EDT 1983
```

```

ken      tty0      Sep 27 00:43
dmr      tty1      Sep 26 23:45
rob      tty2      Sep 26 23:59
bwk      tty3      Sep 27 00:06
jj       tty4      Sep 26 23:31
you      tty5      Sep 26 23:04
ber      tty7      Sep 26 23:34
$

```

在shell返回提示符之前，两个命令都执行了(顺序地)。

也可以同时运行多个程序，例如，如果需要做一些费时的工作，如计算书本中的字数，但在进行其他工作之前不想等待wc完成工作。那么可以这样做：

```

$ wc ch* >wc.out &
6944                               进程标识符
$

```

命令行结尾的&字符通知shell“开始该命令的运行，然后立即从终端取下一个命令”，也就是说，不等该命令完成，就取下一个命令。这样一来，命令开始工作，在它运行时，用户可以去做其他的事。把输出导向至wc.out，做其他事时就不会受到干扰。

一个运行的程序实例称为进程。由&初始化的命令，经由shell打印出的数字称为进程标识符。该进程标识符可以在其他命令中使用，用于引用某一个运行着的程序。

重要的是把程序和进程区分开，wc是一个程序；每次运行程序wc时，就创建了一个新的进程。如果同一个程序在同时有若干个实例在运行，那么每一个实例都是一个独立的进程，且有一个不同的进程标识符。

如果管道线被&初始化了，如下所示：

```

$ pr ch* /lpr &
6951                lpr的进程标识符
$

```

那么其中的进程都会立即开始——\$适用于整个管道线。然而只打印一个进程标识符，它对应于最后一个进程。

命令

```
$ wait
```

要等到所有进程都被&初始化之后。如果它不立即返回，命令会一直运行。可用Delete中断wait。

可以用shell打印的进程标识符来停止被&初始化的进程：

```
$ kill 6944
```

如果忘记了进程标识符，可以用命令ps显示正在运行的一切内容。如果着急的话，kill0会终止除了所登录的shell之外的全部进程。如果想知道别的用户在做什么，ps -ag会展示当前运行的所有进程。下面是一些输出的例子：

```

$ ps -ag
  PID TTY  TIME CMD
   36 co   6:29 /etc/cron
 6423 5     0:02 -sh
 6704 1     0:04 -sh
 6722 1     0:12 vi paper
 4430 2     0:03 -sh
 6612 7     0:03 -sh
 6628 7     1:13 rogue

```

```

6843 2    0:02 write dmr
6949 4    0:01 login bimmler
6952 5    0:08 pr ch1.1 ch1.2 ch1.3 ch1.4
6951 5    0:03 lpr
6959 5    0:02 ps -ag
6844 1    0:02 write rob

```

```
$
```

PID表示进程标识符；TTY为同进程相关的终端(如在who中一般)；TIME是以分秒计的处理器时间；其他是在运行的命令。ps在系统的不同版本中均有所不同，所以你得到的输出格式可能会与本书的不同，变量甚至也会有所不同——请参看手册的ps(1)页。

进程有和文件相同的层次结构，每个进程有一父进程，也可能还有子进程。用户shell由与连接到系统的终端线相关的进程创建。当运行命令时，这些进程就是shell的直接子进程。如果运行其中一个程序，比如，运行从ed中退出的!命令，ed创建自己的子进程，这样就有了shell的孙进程。

有时一个进程花费的时间是如此之长，以至于可以在这个进程运行时，关闭终端然后回家，而不是等待它完成。但是在关闭终端或断开连接时，即使使用了&，进程通常也会被终止。命令nohup(no hangup)用来处理这种情形。如果执行如下操作

```
$ nohup 命令 &
```

那么在用户退出登录后，命令仍继续运行。命令产生的任何输出会保存在名为nohup.out的文件之中。但不能使用nohup追溯一条命令。

如果进程耗费大量的处理机资源，比较合适的是赋予该进程低于正常的优先级，这可以通过另一个名为nice的程序来完成：

```
$ nice 耗费大的命令 &
```

nohup自动调用nice，用户一旦退出了登录，便可以以较长的时间来执行该命令。

最后，可以直接通知系统，在一清早别人还在睡觉没有进行计算的时候，启动用户的进程。该命令名为at(1)：

```

$ at 时间
你打算运行的命令
ctl-d
$

```

以下是一种典型的使用方法，命令当然可以来自文件：

```

$ at 3am < file
$

```

时间time可以是24小时制，如2130；或12小时制，如930 pm。

1.4.5 剪裁环境

UNIX系统的优点之一是，可通过多种方式调整系统，使之更接近个人习惯或适合本地计算环境。举例说，我们曾讨论过删字符和消行符，它们通常约定为#和@。可用下列操作在任何时间修改它们

```
$ stty erase e kill k
```

这里e是打算用于删除字符而k用于行删除。但是每次登录时却要键入它们，也是件烦人的事。

Shell可以补救这一不足。如果在用户登录目录中有一个名为.profile的文件，那么登录时shell就会执行其中的命令，然后显示第一个提示符。所以可以把命令放入.profile中，从而建立所希望的环境，这样每次登录时，它们都会被执行。

多数人会把下列操作首先放进.profile:

```
stty erase ←
```

本书这里使用←是为了方便读者，这样就可以在 .profile文件中放进一个Backspace键。stty也理解表示ctl-x的^x，所以用下列处理可以得到相同的效果：

```
stty erase '^h'
```

因为ctl-h也表示Backspace。(^ 字符是管道操作符|的一个作废的同义词，所以必须用引号保护。)

如果终端没有快捷停止键，可以加入：

```
stty erase '^h' -tabs
```

如果希望在登录时看看系统是否忙碌，可加入

```
who | wc -l
```

以便统计用户数。如果要知道有什么新闻，可以加入news。某些人还喜欢试试自己的运气：

```
/usr/games/fortune
```

过一会儿，如果觉得登录时间太长了，在.profile中删掉一些内容，只剩下最必要的。

Shell的某些特性实际上由所谓的shell变量所控制，其值可被访问也可被设置。例如，如前所用的提示符\$，实际上存放在shell变量PS1中，可以设置为任何用户希望的提示符，操作如下：

```
PS1='Yes dear?'
```

由于提示字符串中有空格，所以引号是必需的。在该语句中，=旁不允许有空格。

Shell也把HOME及MAIL视作特别变量。HOME是用户当前目录名，它通常不必用.profile文件就会正常地设置。变量MAIL命名了保存mail的标准文件，如果shell对它作了定义，就可以在每一命令之后得到是否有新邮件到达的通知[⊖]：

```
MAIL=/usr/spool/mail/you
```

(在你的系统中，mail系统可能与这里描述的不同，但/usr/mail/you是常见的形式。)

也许shell变量中最有用的是用来控制搜寻命令的变量。请回忆，当键入命令名时，shell通常首先查询当前目录，而后是/bin，再是/usr/bin。这一目录序列称为查询路径，并存放在名为PATH的shell变量中。如果缺省的查询路径不是所希望的，可以修改它，而且通常也是在.profile中修改。例如，下面的操作把缺省路径及/usr/games设置为路径：

```
PATH=../bin:/usr/bin:/usr/games
```

一种方式...

其语法有一点奇怪：由冒号分隔的目录名序列。请记住，.是当前目录，可以略去.，在PATH中的空的分量意指当前目录。

⊖ 这在shell中的实现是很糟的。在每一个命令之后，可看到有关文件明显地装载给系统。而且，如果读者长期在一个编辑器上工作，是不会知道有新邮件到达的，因为读者没有在所登录的shell上运行新的命令。一个较好的设计是每隔数分钟检查一次，而不是在每一个命令之后。在第5章和第7章中介绍了怎样实现这类邮件检查器。第三种可能的方式(但这并不对每个人都合适)，是有自己的邮件通知程序，这个程序当然知晓读者的邮件是何时到达的。

设置PATH变量的一种简化方式是把变量设定为先前的值:

```
PATH=$PATH:/usr/games
```

...另一方式

可以在变量名前缀以\$, 从而得到变量的值。在上例中, 用表达式\$PATH提取PATH当前值, 并加入新值, 然后将结果赋给PATH。这一点可以用echo进行验证:

```
$ echo PATH is $PATH
PATH is :/bin:/usr:/bin:/usr/games
$ echo $HOME          用户登录目录
/usr/you
$
```

如果用户有自用的命令, 也许打算把它们收集到自用的目录中, 并把它们加入到查询目录中。在这种情形下, PATH语句如下:

```
PATH=: $HOME/bin:/bin:/usr/bin:/usr/games
```

在第3章中将讨论如何编写用户自己的命令。

另一种变量是TERM, 通常由文本编辑器使用, 且比ed更时新。它命名用户所使用的终端种类, 可以用来供程序更有效地管理屏幕。可以将

```
TERM=adm3
```

添加到用户的.profile中去。

也有可能使用缩写变量。如果你有常用的、名称较长的目录, 那么有必要添加

```
d=/horribly/long/directory/name
```

到.profile文件中, 这样就可使用如下语句:

```
$ cd $d
```

类似于d的、私用的变量通常用小写方式拼写, 以便同shell自身使用的变量相区别, 例如PATH。

最后, 有必要把用户打算在其他程序中使用的变量通知shell, 这可以通过export 命令实现, 有关内容在第3章中进一步讨论:

```
export MAIL PATH TERM
```

现在作一小结, 典型的.profile文件可能如下例所示:

```
$ cat .profile
stty erase '^h' -tabs
MAIL=/usr/spool/mail/you
PATH=$HOME/bin:/bin:/usr/bin:/usr/games
TERM=adm3
b=$HOME/book
export MAIL PATH TERM b
date
who | wc -l
$
```

本书没有必要穷尽shell所提供的服务。最有用的功能之一, 是把已有的命令嵌入shell处理的文件中, 从而创建一条新命令。这种方法简单却能产生显著的效果, 有关讨论从第3章开始。

1.5 UNIX 系统的其余部分

UNIX系统中的内容远远多于本章所讨论的内容, 也远比本书的内容多。现在, 你可以放心地使用系统, 特别是, 要尽量地使用手册。当对如何使用命令有疑问时, 应查询手册。

经常浏览手册是有必要的，这样可以深入了解已知的命令并认识新命令，包括类似FORTRAN77的语言编译器、诸如bc(1)的计算器程序、用于机器间通信的cu(1)及uucp(1)、图像包、统计软件以及units(1)等等。

正如前所述，本书不是要替代手册，只是手册的一种补充。在本章之后，我们会讨论UNIX系统中的内容及程序，从手册中的信息开始，然后是系统的各部分及程序。尽管在手册中，程序间关系不明显，但它们构成了UNIX编程环境的综合基础。

有关文献

由D. M. Ritchie及K.L. Thompson合著的最早关于UNIX的文章“The UNIX Time-Sharing System”，1974年7月首次发表在Communication of the ACM杂志上，1983年1月CACM再次刊登了该篇文章(第89页刊登在1983年3月刊上)。这篇文章为对操作系统感兴趣的人而写，值得所有编程人员阅读。

Bell System Technical Journal(BSTJ)有关UNIX的专刊(1978年7月)，其中有许多讨论后续发展的文章以及一些回顾文章，包括Ritchie及Thompson的对他们发表在CACM上的文章的一些修改。BSTJ的第2本专刊出版于1984年，其中包含新的UNIX论文。

“The UNIX Programming Environment”，作者为B.W.Kernishan及J. R.Mashey(IEEE Computer Magazine, 1981年4月)，意图向程序员们介绍该系统的基本特征。

《UNIX程序员手册》，不论哪一版，都是实用的文献，该文献列出了命令、系统子程序及接口、文件格式以及维护工作。没有手册是不行的，尽管在开始编程之前也许只需要阅读第1卷中的部分内容就行了。手册第7版的第1卷由Holt、Rinehart及Winston出版。

《UNIX程序员手册》第2卷，名为“Documents for Use with the UNIX Time-sharing System”，包含有关主要命令的教材及参考手册，特别地，它介绍了文档准备程序以及程序开发工具，用户有必要阅读它。

“A UNIX Primer”(Prentice-Hall, 1983年)，由Ann及Nico Lomuto编著，是为初学者，特别是为非程序员写的一本入门书。

第2章 文件系统

UNIX系统由文件组成，但这并非像某些人所想象的那样经过刻意的简化。当设计UNIX第1版的时候——当时它甚至连一个名字都没有——讨论都集中在文件系统的结构，即它应该既清晰又易于使用。UNIX系统的成功与它的方便易用，关键就在文件系统。它是“保持简单”这种哲学思想的一个最好的示例，同时向我们展示了这些精心选择的思想被认真实现以后所取得的巨大成就。

为了更好地讨论命令以及它们之间的联系，我们需要对文件系统的结构和它的外部工作有一个很好的背景知识。本章涵盖了使用文件系统的大部分细节——什么是文件系统，它们怎样表示，目录和文件系统层次，权限，I节点(文件的系统内部记录)和设备文件。由于大多数情况下使用UNIX系统都会涉及到对文件操作的处理，因而系统中包含有许多涉及文件使用和重新排列的命令。本章对经常使用的命令进行了介绍。

2.1 文件系统的基础

文件即字节序列(一个字节即一小块信息，长度一般为8位。在这里，一个字节就等于一个字符)。系统并没有给文件强加任何结构，也没有给它的内容附加任何意义——这些字节的意义仅依赖于解释文件的程序。而且，正如我们将要看到的一样，不仅磁盘文件，外设文件也如此。磁带、邮件、键盘键入的字符、行式打印机的输出、管道中的数据流——对系统而言，这些文件不论形式都是字节序列，它只关心其中的程序。

学习文件最好的方式就是实践，所以我们从创建一个小文件开始：

```
$ ed
a
now is the time
for all good people
.
w junk
36
q
$ ls -l junk
-rw-r--r-- 1 you          36 Sep 27 06:11 junk
$
```

junk是一个具有36个字节的文件——在添加文件内容时所键入的36个字符(当然，不包括对任何敲入错误的更正)。现在查看文件

```
$ cat junk
now is the time
for all good people
$
```

cat显示文件的内容。命令od(八进制输出)以可视形式打印文件所有字节：

```
$ od -c junk
0000000  n   o   w           i   s           t   h   e           t   i   m   e  \n
0000020  f   o   r           a   l   l           g   o   o   d           p   e   o
```

```
0000040  p  l  e  \n
0000044
$
```

-c选项意味着“把字节解释成字符”。-b选项按八进制数^①（以8为基数）显示字节：

```
$ od -cb junk
0000000  n  o  w      i  s      t  h  e      t  i  m  e  \n
      156 157 167 040 151 163 040 164 150 145 040 164 151 155 145 012
0000020  f  o  r      a  l  l      g  o  o  d      p  e  o
      146 157 162 040 141 154 154 040 147 157 157 144 040 160 145 157
0000040  p  l  e  \n
      160 154 145 012
0000044
$
```

左边的7位数是文件中的位置，即以八进制表示的下一个字符的序号。顺便提一句，这种对八进制的偏重遗留自PDP-11，在该种机型中一般采用八进制。但十六进制使用更普遍；-x选项会告诉od按十六进制输出。

注意，在每一行后面都有一个字符，它的八进制值为012，这是ASCII码的换行符。当按下Return键的时候，系统自动将其放入输入中。这借鉴于C语言，用\n表示换行符，但这只是od这类程序的习惯用法，是为了阅读方便，实际上存在文件中的值只是一个简单的字节012。

换行符是特殊字符中最普通的例子。与一些终端控制操作相关的其他字符包括Backspace(八进制值010，输出为\b)、Tab(011,\t)以及Return(015, \r)。

在每个事务中区分文件中的字符如何存储以及不同情况下字符如何被解释非常重要。例如，当在键盘上键入一个Backspace键时(假设Backspace键代表删除字符)，内核对它的解释是放弃前面所键入的一个字符，字符和Backspace键都消失，但是，Backspace键在终端上的反应是使光标向回移动一个位置。

如果键入

\←

(即\后跟着一个Backspace键)，此时，内核对它的解释是想要在输入中有一个Backspace键字符，所以\被扔掉而字节010被保存在文件中。当Backspace键响应到终端上时，它使光标落到\的上面。

当输出一个包含Backspace键的文件时，Backspace键被原样传递给终端，它又会使光标向前移动一个位置。当使用od命令显示包含Backspace键的文件时，它显示字节值010，或者如果有-c选项则显示为\b。

Tab键的情况与此相同：在输入端，一个Tab键被响应到终端上并送给正在读它的程序；在输出端，Tab键只送给终端去解释。然而，这里有一点不同——你可以告诉内核让它在输出端解释Tab键。在这种情况下，代替Tab键输出的是到达下一个Tab停止点所需的正确空格数。Tab停止点被设置在第9、17、25等列。命令

```
$ stty -tabs
```

① 文件中的每一个字节都包含一个足够大的数字，以对每一个可打印的字符进行编码。大多数UNIX系统中的编码方法被称作ASCII(American Standard Code for Information Interchange，美国信息交换标准码)，但是有一些机器使用的编码方法称作EBCDIC(Extended Binary-Coded-Decimal Interchange Code，扩充的二-十进制交换码)。本书自始至终都将使用ASCII编码方法，从cat/usr/pub/ascii目录或者ascii(7)可以查到所有字符的八进制值。

在终端上输出空格代替Tab键。参见stty(1)。

Return键的处理方法与此类似。内核对Return键的解释是一个回车和一个换行，但在输入端只保持一个换行。在输出端，一个换行符被扩展成一个回车和一个换行。

UNIX系统对控制信息的表示方法与众不同，特别是它使用换行来结束一行。许多系统代替的方法是每行提供一个“记录”，每一个记录不仅包含数据还包含行中的字符数(没有换行)。其他系统用一个回车符和一个换行来结束每一行，因为在大多数终端上的输出都需要此序列(单词linefeed是换行的同义词，所以这个序列又经常被称作CRLF)。

UNIX系统什么也不做——既没有记录，也没有记录统计，而且你的程序可以放在任何文件中的任何字节。当一个换行送到终端时，它被扩充成一个回车和一个换行，但是程序只需要处理单个换行即可，因为那就是它们所看到的全部。对大多数的用途来说，这种简单的方案就已经足够了。当需要一个更复杂的结构时，很容易在其上进行扩展，反过来，从复杂中创建简单就很难实现了。

既然行的结束由一个换行来标识，那么文件的结束就应以另一个特殊字符来标识，称作\0，代表“文件尾”。然而，查看od的输出，在文件尾没有看到任何特殊字符——它仅仅是停了下来。系统对文件尾的标识不是使用一个特殊的代码，而只是简单地说明文件中没有任何数据了。内核跟踪文件的长度，所以当程序处理完文件中的所有字节时就到达了文件尾。

程序使用一个系统调用(内核中的一个子例程)取得文件中的数据，称作read。每一次read被调用，它都返回文件中的下一部分——例如，终端上键入的下一行。read同样说明返回了文件中多少个字节，所以当它说明“返回了零字节”时，就假设到达了文件尾。如果还有任何遗留字节，read将返回其中的某些字节。事实上，不使用特殊的字节值来表示文件尾很有意义，因为正如我们早先所说的那样，字节的含义依赖于对文件的解释，但是，所有的文件都必须结束，并且所有文件都必须通过read来访问，返回零就成为一个不依赖解释的表示文件尾的方式，所以不需引入一个新的特殊字符。

当从终端读入程序时，内核只有在你键入新行(即按下Return键)时才把每一个输入行传送给程序。如果你犯了个输入错误，若在键入换行符之前意识到，就可以回退并更正。但若在意识到之前已按下了Return键，该行就已经被系统读入，不能再更正了。

我们可以使用cat来查看这种一次一行的输入是怎样工作的。cat通常会保存或缓冲它的输出，以便更有效地按大块输出，但是如果使用“无缓冲”输出cat -u，它就会在读入的同时输出：

\$ cat	cat的缓冲输出
123	
456	
789	
ctl-d	
123	
456	
789	
\$ cat -u	cat的无缓冲输出
123	
123	
456	
456	
789	
789	
ctl-d	
\$	

在按下Return键时，cat接收每一行。若没有缓冲，它会在接收到数据的同时输出。

现在，尝试一些不同之处：键入一些字符，然后按下Ctrl-d来代替Return键：

```
$ cat -u
123ctl-d123
```

cat立即输出字符，Ctrl-d说明“立即把我敲入的字符送给正从我的终端读数据的程序”。与换行符不同，Ctrl-d本身并没有送给程序。现在键入第2个Ctrl-d，不要带任何字符：

```
$ cat -u
123ctl-d123ctl-d$
```

shell响应一个提示符，因为cat没有读入任何字符，确定这就意味着文件结束，并停止。Ctrl-d把从终端读入的你所键入的任何字符都送给程序。如果什么都没有键入，程序就不会读入任何字符，看上去就像到达了文件的末尾。这就是为什么键入Ctrl-d会退出——shell没有看到任何输入。当然，Ctrl-d通常用于标识文件结束，但有意思的是，它还有一些更普通的功能。

练习2-1 在ed中键入Ctrl-d会怎样？把它和以下命令进入比较：

```
$ ed <file
```

2.2 文件结构

文件的格式由使用它的程序来决定。因为存在很多种程序，所以有很多种文件类型。既然文件类型不由文件系统来决定，内核就不能告知文件的类型，因为它并不知道。file命令做经验猜测(下面将说明这一过程有多么简短)：

```
$ file /bin /bin/ed /usr/src/cmd/ed.c /usr/man/man1/ed.1
/bin:          目录文件
/bin/ed:       纯可执行文件
/usr/src/cmd/ed.c: C源程序文本
/usr/man/man1/ed.1: roff, nroff, 或eqn输入文本
$
```

这是四种比较典型的文件，都和编辑器有关：它驻留的目录(/bin)，‘二进制’或可执行程序本身(/bin/ed)，定义程序的‘源码’或C语句(/usr/src/cmd/ed.c)，以及手册页(/usr/man/man1/ed.1)。

在决定类型时，file并不关心文件名(尽管它可以)，因为命名传统仅仅是传统，并不完全可靠。例如，后缀.c的文件通常是C源码，但是并没有任何东西可以防止你创建一个.c文件却包含别的任意的内容。file采用的方法是读入文件的前几百个字节，查看文件类型的线索。(正如我们在后面将要展示的一样，文件拥有特定的系统属性，例如目录，可以通过询问系统进行识别，但是file却能通过读目录来识别它。)

有时线索很明显。一个可执行程序的标识是在它的开始有一个二进制的‘魔力数’。不带任何选项的od命令按16位或2字节的单词输出文件，可使魔力数可见：

```
$ od /bin/ed
0000000 000410 025000 000462 011444 000000 000000 000000 000001
0000020 170011 016600 000002 005060 177776 010600 162706 000004
0000040 016616 000004 005720 010066 000002 005720 001376 020076
...
$
```

八进制值410标识一个纯的可执行程序，该程序可能被几个进程所共享(特定的魔力数与

系统有关)。410的编码方式不是ASCII，所以这个值不会被类似编辑器这样的程序无意中创建出来。但是可以通过运行自己的程序来创建这样一个文件，系统将明白按照惯例这样的文件是二进制代码。

对于文本文件，文件中的线索可能会较深，所以file通过查看诸如#include这类单词来标识C源码，或者是以一个句号开始的行来标识nroff或troff输入。

你也许会疑惑，为什么系统不仔细地跟踪文件的类型呢？一个原因就是避免预先中止一些有用的计算。例如，sort就永远不会把/bin/ed作为输入，尽管

```
$ sort /bin/ed
```

没有任何意义，但毕竟还有许多别的命令可以对任何文件进行操作，没有任何理由限制它们的能力。od、wc、cp、cmp、file和其他一些命令处理文件时并不考虑它们的内容。如果nroff输入与C源码不同，编辑器在创建文件时将被迫进行区分，而且当它读入一个文件进行再编辑时可能还要如此，而这必然会对我们排版第6章到第8章的C程序造成很大困难。

不但没有造成不同，UNIX系统还试图消除文件的差别。所有文本都由以换行符标识结束的行组成，大多数的程序都能理解这种简单的格式。在本书的编写过程中，我们多次运行命令创建文本文件，用上面所列出的那样的命令对它们进行处理，并使用一个编辑器将它们合并到本书的troff输入。你在大多数的页上所看到的脚本都由下面这样的命令完成：

```
$ od -c junk >temp
$ ed ch2.1
1534
r temp
168
...
```

od在它的标准输出上生成文本，该文本可以被用在任何地方。这种一致性是与众不同的。大多数的系统，甚至文本都有多种文件格式，因此要求程序或者用户协商创建一个特殊类型的文件。而在UNIX系统中只有一种类型的文件，因而访问一个文件所需的全部就是它的名字[⊖]。

没有文件格式总体上来说是个优点——程序可以不需考虑文件的类型，而且所有标准程序可以对任何文件进行处理。但是也存在一点缺点，如排序、搜索和编辑程序只希望文本文件作为它们的输入：grep不能正确地检查二进制文件；sort无法对其排序；任何标准编辑器都无法对其进行操作。

对大多数希望将文本文件作为输入的程序都存在着实现上的限制。我们用一个不包含换行符的30 000字节的文本文件对许多程序进行了测试。奇怪的是，仅有很少的程序能够正常地工作。这是因为大多数的程序在不经意之间对文件中一行的最大长度进行了假设。（有一个例外，参见sort(1)的BUGS节。）

非文本文件无疑也有它们的有用之处。例如，大型数据库通常需要另外的地址信息以进行快速访问，而且出于效率的考虑它必须是二进制的。但是非文本文件的每一种格式必须有它自己的一组支持程序，才能实现标准工具在文本格式上所能完成的工作。在机器循环中，文本文件也许不那么高效，但是必须在它和维持更多特殊格式所需附加软件的费用之间取得平衡。如果设计一个文件格式，那么在选择非文本表示之前应仔细地考虑这点。（你还应当考虑在面对较长输入行时程序的健壮性。）

⊖ 对文件系统的一致性有个很好的测试方法，最初由Doug McIlroy进行。UNIX的文件可以很轻巧地进行传送。一个FORTRAN程序的输出可用作FORTRAN编译器的输入吗？大多数系统都不能完成此项测试。

2.3 目录和文件名

所有文件都有一个以/usr/you开始的明确的名称。但如果只有一个名为junk的文件，此时键入ls，并不输出/usr/you/junk，而是输出不带任何前缀的文件名：

```
$ ls
junk
$
```

这是因为每个运行程序，即每个进程，都有一个当前目录，而且所有文件名都被系统假设从该目录开始，除非它们直接以一个反斜线开始。登录shell，然后使用ls，因而也就有了一个当前目录。命令pwd(print working directory)可以识别当前目录：

```
$ pwd
/usr/you
$
```

当前目录是进程，而不是一个人或程序的属性——用户有登录目录，进程有当前目录。如果进程创建一个子进程，则子进程继承其父进程的当前目录。但如果子进程改变到一个新的路径下，父进程不受影响——不论子进程做了什么父进程的当前路径都保持不变。

当前目录的概念带来了记忆上的方便，因为它可以省去许多输入，但其真正的目的是利于组织。相关的文件被置于同一个目录下。/usr通常是用户文件系统的顶层目录(usr是user的简写，与cmp、ls等相似)；/usr/you是你的登录目录，亦即第一次登录时的当前目录；/usr/src包含了系统程序的源码；/usr/src/cmd/sh包含了shell的源文件；等等。无论何时你着手从事一项新工程，或拥有一组相关文件，假定为一组recipes(食谱)，都可使用mkdir创建一个新目录并把文件放在该目录下。

```
$ pwd
/usr/you
$ mkdir recipes
$ cd recipes
$ pwd
/usr/you/recipes
$ mkdir pie cookie
$ ed pie/apple
...
$ ed cookie/choc.chip
...
$
```

注意，对子目录的引用非常简单。pie/apple有一个很明显的含义：苹果派食谱，位于目录/usr/you/recipes/pie下。也可以直接将其放在recipes目录中，即recipes/apple.pie，而不是创建一个recipes的子目录，但是把所有的派组织在一起似乎更好。例如，饼干食谱可以保存在目录recipes/pie/crust下，而不是在每个派的食谱中重复。

尽管文件系统是一个强有力的组织工具，但还是可能忘记把一个文件放在何处，或者甚至不知道有什么文件。一个简单的解决方案就是提供搜索命令。ls命令对于寻找文件很有用，但它不搜索子目录。

```
$ cd
$ ls
junk
recipes
$ file *
junk:          ascii 文本
```

```

recipes:                                目录
$ ls recipes
cookie
pie
$ ls recipes/pie
apple
crust
$

```

这一小段文件系统结构显示如下:

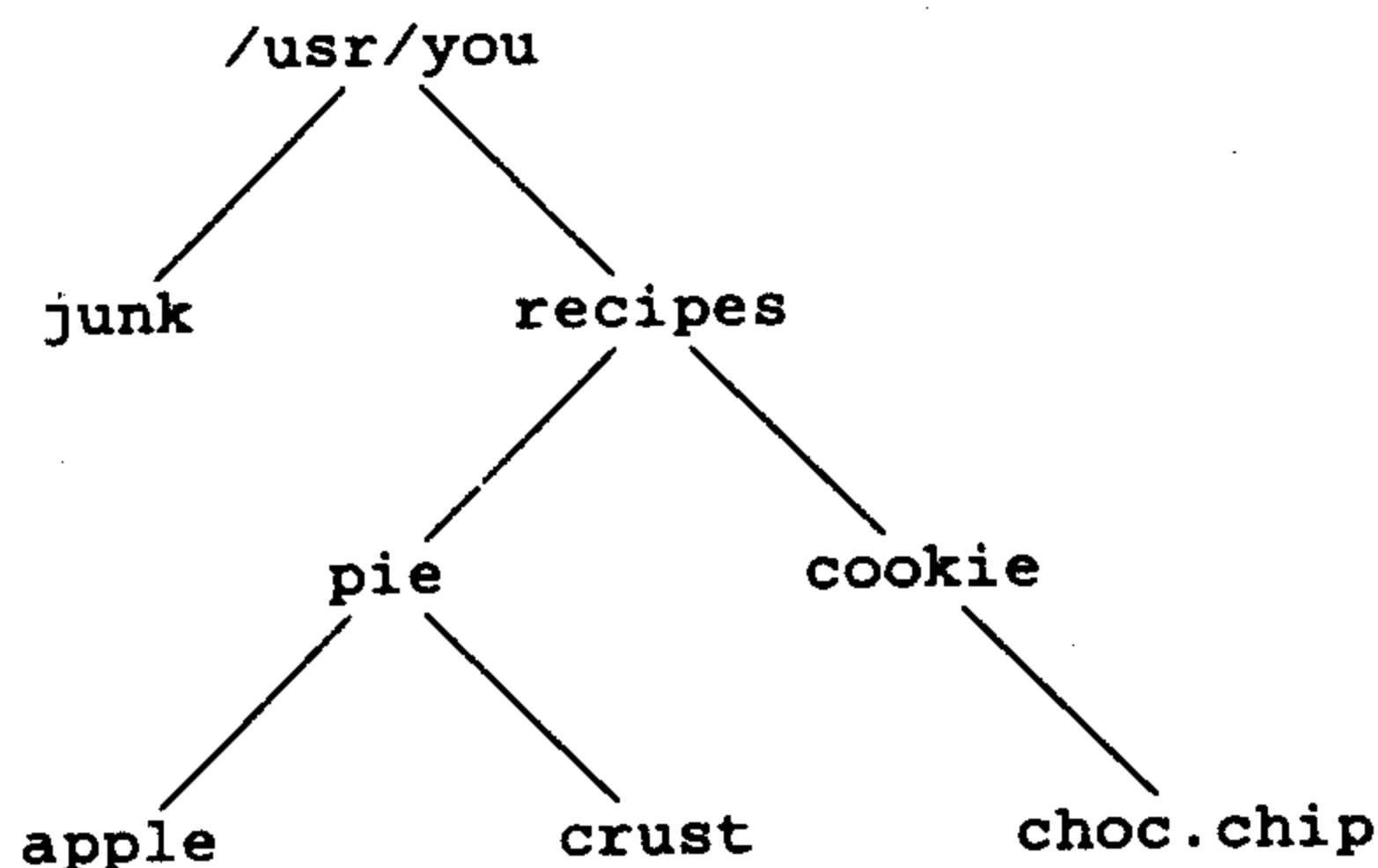


图2-1 文件系统的一部分

命令du(disc usage)显示一个目录下的文件, 包括其所有的子目录, 占据了多少磁盘空间。

```

$ du
6      ./recipes/pie
4      ./recipes/cookie
11     ./recipes
13     .
$

```

文件名很显而易见, 数字指出每个文件所占用的磁盘块数——一般来说每块为512或1024个字节。目录所对应的值表明该目录和它的子目录中所有文件占用的磁盘块数。

du有个选项为-a(all), 它输出一个目录中的所有文件。如果其中包括目录, du也将列出:

```

$ du -a
2      ./recipes/pie/apple
3      ./recipes/pie/crust
6      ./recipes/pie
3      ./recipes/cookie/choc.chip
4      ./recipes/cookie
11     ./recipes
1      ./junk
13     .
$

```

du-a的输出可从管道输入给grep, 以查找特定的文件:

```

$ du -a | grep choc
3      ./recipes/cookie/choc.chip
$

```

回忆第1章中我们说过, . 是引用目录自身的目录入口, 它允许在不知道全名的情况下访问目录。du在目录中查找文件, 若没有告诉它是哪个目录, 则假设是 ., 即当前所处的目录。因而junk和./junk是同一文件。

不考虑在内核内的基本属性，目录和普通文件一样都处在文件系统里。它们可以像普通文件一样被访问。但是，它们不可以和普通文件一样被创建和写——为不致引起混乱和保护用户文件，内核把对目录内容的所有控制权都留给了自己。

现在，可以查看一下目录中的字节：

```
$ od -cb .
0000000  4  ;  .  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
          064 073 056 000 000 000 000 000 000 000 000 000 000 000 000 000
0000020 273  (  .  .  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
          273 050 056 056 000 000 000 000 000 000 000 000 000 000 000 000
0000040 252  ;  r  e  c  i  p  e  s  \0 \0 \0 \0 \0 \0 \0 \0 \0
          252 073 162 145 143 151 160 145 163 000 000 000 000 000 000 000
0000060 230  =  j  u  n  k  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
          230 075 152 165 156 153 000 000 000 000 000 000 000 000 000
0000100
$
```

看到文件名在何处了吗？目录格式是二进制和文本数据的组合。一个目录由16字节的块组成，最后14个字节保存文件名，并用ASCII NUL(值为0)填充，前两个用于告诉系统文件的管理信息在哪儿——我们将回到那儿。每个目录由两个入口开始：.和..。

```
$ cd                      回到注册目录
$ cd recipes
$ pwd
/usr/you/recipes
$ cd ..; pwd              向上一层目录
/usr/you
$ cd ..; pwd              再向上一层
/usr
$ cd ..; pwd              再向上一层
/
$ cd ..; pwd              再向上一层
/
$                          到达最顶层
```

/称作文件系统的根目录，系统中每一个文件都位于根目录或者它的一个子目录中。根目录是它自己的父目录。

练习2-2 按照本节给出的信息，应能够粗略地理解ls命令是怎样运作的。提示：cat . > foo; ls -f foo。

练习2-3 (较难)pwd命令怎样运作？

练习2-4 du命令用来监视磁盘的使用情况。使用它在文件层次结构中查找文件充其量是一个奇特的用法，而且可能并不适宜。换一种方法，在手册页中查看find(1)，并比较这两个命令。特别是，比较du -a | grep ...和相应的find命令。哪个运行得更快？是创建一个新工具，还是使用旧工具中的侧面效应？

2.4 权限

每个文件都有一组权限与其相连，它决定谁可以对这个文件进行什么样的操作。也许你把情书保留在系统中，按层次安排在一个目录下，并且不想让其他人看到。这样，你就可以改变每封信件的权限以防止流言蜚语，或者只改变包含信件的目录的权限，以这种方式来阻止窥探者。

但我们必须警告你：在每个UNIX用户中都有一个特殊的用户，称作超级用户(super-user)，

他可以读或修改系统中的任何文件。特殊的登录名root拥有超级用户特权，它由系统管理员在做系统维护时使用。知道root口令的任何人都可以读你的情书，因此，不要把敏感的材料放在文件系统中。

若需要更多的隐私权，可以改变文件中的数据，这样即使是超级用户也不可读它(或至少是不能理解它)，这可以通过crypt命令来实现(crypt(1))。当然，甚至crypt也不是彻底安全的。一个超级用户可以改变crypt命令自身，并且对crypt算法存在着密码攻击，前者利用职务之便就可做到，而后者则要花费很大力气，但毕竟crypt在实际使用中还算安全。

在现实生活中，大多数安全的破坏都是由于密码的泄露或者是被轻易猜到。偶然地，也有管理员的失职，使怀有恶意的用户得到了超级用户的权限。在本章后面“有关文献”中的一些论文对安全问题进行了更深入的讨论。

当登录时，输入一个名字，接着输入一个密码来验证你就是这个人。名字是登录标识，或称login-id。但系统实际上是用一个数字来识别，称作user-id，或uid。事实上，不同的login-id可能有着同一个uid，使它们对于系统来说不可区分，尽管这相对很少见或者出于安全的考虑而不可取。除了uid，用户还被赋给一个标识group-id，用于把该用户放入一类用户中。在许多系统上，所有普通的用户(相对类似root的login-id而言)都被放入一个单个组，称作others，但也有的系统与此不同。文件系统，也就是总体意义上的UNIX系统，根据赋给用户的uid和group-id的特权决定该用户可以做什么。

文件/etc/passwd是口令文件(password file)，它包含关于每个用户的所有注册信息。用户可以和系统一样，通过在/etc/passwd中查找自己的名字，找到自己的uid和group-id：

```
$ grep you /etc/passwd
you:gkmbCTrJ04COM:604:1:Y.O.A.People:/usr/you:
$
```

口令文件中的各域以冒号分隔，并如下所示(参见passwd(5))：

```
login-id:encrypted-password:uid:group-id:miscellany:login-directory:shell
```

文件是普通文本，但是域定义和分隔符遵循使用它们的程序所一致认可的规则。

shell域通常为空白，暗示使用系统设定的shell：/bin/sh。miscellany域可以包含任何东西，通常为你的名字、地址或电话号码。

注意，口令出现在第二个域，但是是以保密格式出现。任何人都可以读口令文件(你刚才所生成的)，所以如果口令以它原来的形式存放在那里，任何人都可以使用它来冒充你。在登录期间给出口令后，对其编码，并将结果与/etc/passwd中已编码的口令进行比较。如果一致，则允许登录。之所以这样做是因为保密算法具有这样一种属性，它很容易从一种清晰的形式转变成保密的形式，但反过来却很难。例如，如果口令是ka-boom，它可能被编码成gkmbCTrJ04COM，但是对于后者，却没有一种很容易的方法使其还原成前者。

内核通过查看与文件相连的权限决定你能否能读/etc/passwd。每个文件具有三种权限：读(即查看它的内容)、写(即改变它的内容)和执行(即把它作为一个程序运行)。此外，不同的权限可以赋给不同的人。作为文件的所有者，可以具有读、写和执行三种权限。同组用户有另一种权限设置。其他人有第三种权限设置。

ls命令的-l选项在输出其他信息的同时，也输出权限信息：

```
$ ls -l /etc/passwd
-rw-r--r-- 1 root 5115 Aug 30 10:40 /etc/passwd
```

```
$ ls -lg /etc/passwd
-rw-r--r-- 1 adm          5115 Aug 30 10:40 /etc/passwd
$
```

这两行可以被共同解释成：/etc/passwd的所有者是login-id为root的根用户，小组标识符为adm，长5115个字节，最后的修改日期是8月30日上午10:40分，有一条链(文件系统中的一个小名字，我们将在下一节中讨论)。有一些版本的ls命令同时列出文件所有者及其所属小组的标识符。

串-rw-r--r--是ls对文件权限的表示。第一个短横表明它是一个普通文件。如果是一个目录，则应该显示d。接着的三个符号表明文件所有者(基于uid)的读、写和执行权限。rw-表明root(文件所有者)可以读或写，但是不可以执行文件。可执行文件用x来表示。

下面的3个字符(r--)包含组权限，在本例中是小组adm中的人，假设是系统管理员，可以读文件但是不可以写和执行。再下面的3个字符(r--)定义了其他人的权限——系统中其他的用户。那么，在本机上，只有root可以改变一个用户的登录信息，但是任何人都可以读取文件以查阅信息。一种值得推荐方法是让adm组成员同样具有对/etc/passwd的写权限。

文件/etc/group包含组名和组标识，并定义了哪个用户是在哪个组；/etc/passwd仅识别登录组。newgrp命令可以改变组权限。

任何人都可以输入：

```
$ ed /etc/passwd
```

并编辑口令文件，但只有根用户可以写回所作的改变。那么，既然改变口令要涉及编辑口令文件，怎样才能改变口令呢？改变口令的程序叫做passwd，可以在/bin下找到：

```
$ ls -l /bin/passwd
-rwsr-xr-x 1 root      8454 Jan  4 1983 /bin/passwd
$
```

(注意，/etc/passwd是包含注册信息的文本文件，而/bin/passwd是一个包含可执行程序的文件，运行该程序可以改变口令信息。两者位于不同的路径下。)这里的权限说明任何人都可以执行命令，但是只有root可以改变passwd命令。但对于文件所有者而言，在执行域中用s代替x，当程序运行时，由文件所有者赋给其权限，在这个例子中文件所有者是根用户root。由于/bin/passwd对root进行了set-uid，任何用户都可以运行passwd命令编辑口令文件。

set-uid位是一个简单但很完美的想法^①，它解决了许多安全问题。例如，一个游戏程序的作者可以使程序对所有者set-uid，这样它就可以更新一个记分文件，否则该文件将防止其他用户的访问。但是，set-uid概念具有潜在的危险。/bin/passwd必须被更正，如果不这样，它会在根用户的帮助下毁坏系统信息。如果它具有权限-rwsrwxrwx，它就可以被任何用户重写，因而，这些用户就可以用实现任意事情的程序来替代它。对于一个set-uid程序来说，这一点特别严重，因为root对系统中的每个文件都有访问权限(某些UNIX系统在文件被修改时关掉其set-uid位，以减少这个安全漏洞的危险)。

set-uid位的功能很强大，但是基本上只用于很少的几个系统程序上，诸如passwd。让我们看一个较普通的文件。

```
$ ls -l /bin/who
-rwxrwxr-x 1 root      6348 Mar 29 1983 /bin/who
$
```

① set-uid位的专利权归Dennis Ritchie所有。

任何人都可以执行who，而且root和文件所有者小组的成员可以写。这里“可执行”意味着，当键入

```
$ who
```

给shell时，它搜索几个目录，其中之一就是/bin，查找名为who的文件。如果它找到这样一个文件，而且如果这个文件拥有执行权限，shell就会调用内核去运行它。内核检查权限，如果它们有效，则运行程序。注意，程序只是一个拥有执行权限的文件。下一章我们介绍只是文本文件的程序，但是它可以作为命令来执行，因为它拥有执行权限。

目录权限的操作有些不同，但基本思想是一样的。

```
$ ls -ld .
drwxrwxr-x 3 you          80 Sep 27 06:11 .
$
```

ls的-d选项是让它说明目录本身的信息，而不是它的内容。输出中的第一个字母d说明，实际上是一个目录。r说明可以读此目录，这样就可以用ls(或od，就此而言)查找目录中的文件。w意味着可以在这个目录中创建或删除文件，因为这要求修改因而要写目录文件。

事实上，不可以简单地写一个目录——甚至root也不可以这样做：

```
$ who >.          试图覆盖.目录(当前目录)
.: cannot create   不能创建目录
$
```

但可以用系统调用来创建和删除文件，而且只有通过它们才有可能改变一个目录中的内容。然而，权限的思想仍在应用：w说明谁可以使用系统例程来修改目录。

删除一个文件的权限与文件自身无关。如果对一个目录拥有写权限，就可以从那里删除文件，甚至是那些不允许写的文件。然而rm命令在删除一个受保护的文件之前要求证实，询问用户是否确实想这样做——这是UNIX程序双重检查用户意图的一种少见情形。(-f选项强迫rm不必在删除文件时询问。)

目录权限中的x并不意味着执行，它的意思是“搜索”。目录的执行权限决定是否可以在这个目录中查找文件。因此，就有可能给其他用户创建一个具有--x模式的目录，暗示这些用户可以访问他们所知道的该目录下的任何文件，但是不可以在其上运行ls，或是读它以查看该目录下有什么文件。类似地，如果目录拥有权限r--，用户就可以运行ls以查看文件，但是不可以使用目录中的内容。一些安装程序使用这项功能在繁忙的时候关掉/usr/games。

chmod(change mode)命令用于改变文件的权限：

```
$ chmod 权限 文件名 ...
```

权限的语法非常繁琐。它们可以用两种方式：八进制数或者符号来描述。符号描述可以说明权限的相对改变，因而有时更为方便，但是八进制数要更容易一些。如果采用这种方式：

```
$ chmod rw-rw-rw- junk          这种方式无效
```

显然要比

```
$ chmod 666 junk
```

好，但却不能这样做。在八进制模式下通过给读权限赋值4，给写权限赋值2，给执行权限赋值1来改变权限。和在ls中一样，三个数字分别说明文件所有者、组成员和其他成员的权限。

符号代码较难解释，必须查看chmod(1)以得到一个正确的描述。在这里，只要知道+打开一个权限，而-关闭一个权限即可。

例如：

```
$ chmod +x command
```

允许每个人执行命令，而

```
$ chmod -w file
```

关闭每个人的写权限，包括文件所有者本人。除了超级用户，就只有文件所有者本人可以改变一个文件的权限，而不考虑权限本身是什么。即使别人允许你写一个文件，系统也不允许你改变它的权限。

```
$ ls -ld /usr/mary
drwxrwxrwx 5 mary          704 Sep 25 10:18 /usr/mary
$ chmod 444 /usr/mary
chmod: can't change /usr/mary
$
```

如果一个目录是可写的，人们就可以从中删除文件，而不管文件本身的权限是什么。如果要确保你的朋友永远不能从一个目录下删除文件，就从该目录中去掉写权限：

```
$ cd
$ date >temp
$ chmod -w .           使目录不可写
$ ls -ld .
dr-xr-xr-x 3 you          80 Sep 27 11:48 .
$ rm temp
rm: temp not removed    不能删除文件
$ chmod 775 .           恢复权限
$ ls -ld .
drwxrwxr-x 3 you          80 Sep 27 11:48 .
$ rm temp
$                       现在可以删除文件了
```

现在temp没有了。注意改变一个目录的权限并没有改变它的修改日期。修改日期的改变反映了文件内容的改变，而不是它的模式。权限和日期并非保存在文件自身中，而是保留在系统结构中，该结构所做索引节点，或称I节点，是下一节的主题。

练习2-5 练习使用chmod，尝试各种简单的模式，如0和1。小心不要毁坏登录目录。

2.5 I节点

一个文件有几项内容：名称、内容和管理信息如权限和修改时间。管理信息存放在I节点中，同时存放的还有一些基本的系统数据如文件的长度、文件内容存放在磁盘上的位置等等。

在I节点中有三个时间：文件内容最后一次修改(写)的时间；文件最后一次使用(读或执行)的时间；以及I节点自身最后一次改变的时间，例如设置权限。

```
$ date
Tue Sep 27 12:07:24 EDT 1983
$ date >junk
$ ls -l junk
-rw-rw-rw- 1 you          29 Sep 27 12:07 junk
$ ls -lu junk
-rw-rw-rw- 1 you          29 Sep 27 06:11 junk
$ ls -lc junk
-rw-rw-rw- 1 you          29 Sep 27 12:07 junk
$
```

改变文件的内容并不改变文件的使用时间,使用时间可以通过ls -lu得到。改变文件的权限只改变I节点的改变时间,改变时间可以通过ls -lc得到。

```
$ chmod 444 junk
$ ls -lu junk
-r--r--r-- 1 you          29 Sep 27 06:11 junk
$ ls -lc junk
-r--r--r-- 1 you          29 Sep 27 12:11 junk
$ chmod 666 junk
$
```

ls的-t选项可以把文件按日期排序,缺省情况下按最后一次改动时间排序,-t可以与-c或-u连用,表示按I节点改动或文件被读取的时间排序:

```
$ ls recipes
cookie
pie
$ ls -lut
total 2
drwxrwxrwx 4 you          64 Sep 27 12:11 recipes
-rw-rw-rw- 1 you          29 Sep 27 06:11 junk
$
```

上面的recipes是最近被使用过的,因为刚看过它的内容。

为了理解I节点,不仅要理解ls的选项,更重要的是要认识到I节点就是文件。所有的目录层次确实为文件提供了方便的名称。文件的系统内部名称即它的I节点号,I节点保存文件信息的号码。ls -i按十进制输出I节点号:

```
$ date >x
$ ls -i
15768 junk
15274 recipes
15852 x
$
```

I节点号存储在目录的头两个字节中,在名称的前面。od -d可以把每个字节按十进制,而不是八进制输出,因而可以看到I节点号:

```
$ od -c .
0000000 4 ; . \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000020 273 ( . . \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040 252 ; r e c i p e s \0 \0 \0 \0 \0 \0 \0 \0
0000060 230 = j u n k \0 \0 \0 \0 \0 \0 \0 \0
0000100 354 = x \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000120
$ od -d .
0000000 15156 00046 00000 00000 00000 00000 00000 00000
0000020 10427 11822 00000 00000 00000 00000 00000 00000
0000040 15274 25970 26979 25968 00115 00000 00000 00000
0000060 15768 30058 27502 00000 00000 00000 00000 00000
0000100 15852 00120 00000 00000 00000 00000 00000 00000
0000120
$
```

每个目录入口处的头两个字节是文件的名称和内容之间的唯一联系。目录中的文件名因而被称作链,因为它把目录层次结构中的名称链接到它的I节点,因而也就链接到数据。同一个I节点号可以出现在多个目录中。rm命令并不真正删除I节点,它删除目录入口或链。只有当链接到文件的最后链消失以后,系统才删除I节点,也就是文件本身。

如果目录入口处的I节点号为零,说明链已被删除,但并不见得是文件的内容消失了——

也许在某处还有一条链指向了此文件。可通过删除文件来验证I节点号变为零：

```
$ rm x
$ od -d .
0000000 15156 00046 00000 00000 00000 00000 00000 00000
0000020 10427 11822 00000 00000 00000 00000 00000 00000
0000040 15274 25970 26979 25968 00115 00000 00000 00000
0000060 15768 30058 27502 00000 00000 00000 00000 00000
0000100 00000 00120 00000 00000 00000 00000 00000 00000
0000120
$
```

在此目录中创建的下一个文件会占用未设置的槽，尽管它可能有一个不同的I节点号。

ln命令使一个链指向一个现存的文件，语法是：

```
$ ln 旧文件名 新文件名
```

链的目的是让同一个文件具有两个名称，这样它就可以出现在不同的目录中。在很多系统中都有一个链指向/bin/ed，叫做/bin/e，这样人们就可以用e来调用编辑器。一个文件的两个链指向同一个I节点，因而有着相同的I节点号：

```
$ ln junk linktojunk
$ ls -li
total 3
15768 -rw-rw-rw- 2 you          29 Sep 27 12:07 junk
15768 -rw-rw-rw- 2 you          29 Sep 27 12:07 linktojunk
15274 drwxrwxrwx 4 you          64 Sep 27 09:34 recipes
$
```

在权限和文件所有者之间出现的整数是指向文件的链的数目。因为每条链都同样指向I节点，因而每条链都同等重要——在第一条链和随后的链之间没有任何区别。（注意ls所计算的整个磁盘空间会出错，因为它计算了两次。）

在改变了一个文件以后，以任何名称引用此文件都会显示出所做的改变。因为所有的链都指向了同一个文件。

```
$ echo x >junk
$ ls -l
total 3
-rw-rw-rw- 2 you          2 Sep 27 12:37 junk
-rw-rw-rw- 2 you          2 Sep 27 12:37 linktojunk
drwxrwxrwx 4 you          64 Sep 27 09:34 recipes
$ rm linktojunk
$ ls -l
total 2
-rw-rw-rw- 1 you          2 Sep 27 12:37 junk
drwxrwxrwx 4 you          64 Sep 27 09:34 recipes
$
```

在linktojunk被删除以后，链的数目变回到1。正如我们在前面所说的那样，删除一个文件只是打破一条链，文件依旧存在直到最后一条链被删除。当然，实际上大多数的文件都只有一条链，不过这使我们又一次看到了一种简单的思想产生的巨大灵活性。

顺便提一下：一旦文件最后一条链被删除，数据就无法再恢复了。删除文件是将其放入焚化炉，而不是废纸篓，也就没有任何办法把它们从灰烬恢复成原样。（这里还存在有恢复的微弱希望。大多数大型UNIX系统都有一个正式的备份过程，它周期性地将改变的文件复制到一个安全的地方，如磁带等，从这里可以对它们进行检索。为了保护自己 and 心里有底，你应该弄清楚系统中提供了多少备份，如果一点也没有，那就要小心了——一些磁盘的失误就有可能导致大灾祸。）

当两个人想要共享一个文件时，提供到文件的链就非常方便，但有时候确实想要一个单独的拷贝——具有相同信息的不同文件。例如，在对一个文件做修改之前一般应该复制一份，这样当要取消所做的修改时可以从原始文件中恢复。这时链不会有任何帮助，因为当数据改变时，所有的链都会反映出这种变化。cp完成对文件的复制：

```
$ cp junk copyofjunk
$ ls -li
total 3
15850 -rw-rw-rw- 1 you          2 Sep 27 13:13 copyofjunk
15768 -rw-rw-rw- 1 you          2 Sep 27 12:37 junk
15274 drwxrwxrwx 4 you        64 Sep 27 09:34 recipes
$
```

junk和copyofjunk有着不同的I节点号，因为它们是不同的文件，尽管当前它们有着相同的内容。建议改变备份的权限，这样它就不会在无意中被删掉。

```
$ chmod -w copyofjunk          关闭写权限
$ ls -li
total 3
15850 -r--r--r-- 1 you          2 Sep 27 13:13 copyofjunk
15768 -rw-rw-rw- 1 you          2 Sep 27 12:37 junk
15274 drwxrwxrwx 4 you        64 Sep 27 09:34 recipes
$ rm copyofjunk
rm: copyofjunk 444 mode n      不！不能删除
$ date >junk
$ ls -li
total 3
15850 -r--r--r-- 1 you          2 Sep 27 13:13 copyofjunk
15768 -rw-rw-rw- 1 you          29 Sep 27 13:16 junk
15274 drwxrwxrwx 4 you        64 Sep 27 09:34 recipes
$ rm copyofjunk
rm: copyofjunk 444 mode y      好吧！可以删除
$ ls -li
total 2
15768 -rw-rw-rw- 1 you          29 Sep 27 13:16 junk
15274 drwxrwxrwx 4 you        64 Sep 27 09:34 recipes
$
```

改变一个文件的备份并不会改变它的原件，同时删除备份对它的原件也没有影响。注意，由于copyofjunk的写权限被关闭，rm在删除文件之前会要求确认。

还有许多通用的命令用于操纵文件：mv可以通过简单地重新安排链来移动或重命名文件。它的语法与cp相同：

```
$ mv junk sameoldjunk
$ ls -li
total 2
15274 drwxrwxrwx 4 you        64 Sep 27 09:34 recipes
15768 -rw-rw-rw- 1 you        29 Sep 27 13:16 sameoldjunk
$
```

sameoldjunk与我们原有的junk是同一个文件，I节点号相同，只是名称——与I节点15768相连的目录入口——被改变了。

我们一直是在一个目录下做这些文件重组的工作，在跨目录的情况下它同样适用。ln通常用于在不同目录下相同名称的文件之间建立链，例如当几个人同时使用一个程序或文档时。mv可以把一个文件或目录从一个目录移到另一个目录下。事实上，mv和cp更普通的用法是：

```
$ mv (或 cp) 文件1 文件2 ... 目录
```

移动(或复制)一个或更多的文件到最后一个参数所指明的目录下。链或备份有相同的文件名。例如，如果要强化编辑器，你可以使用：

```
$ cp /usr/src/cmd/ed.c .
```

来得到要使用的资源的备份。若要修改具有多个不同源文件的shell, 可以使用:

```
$ mkdir sh
$ cp /usr/src/cmd/sh/* sh
```

而且cp会将shell所有的源文件复制到子目录sh下(cp 并不是非常聪明, 它假设在/usr/cmd/sh下没有任何子目录)。在一些系统中, ln同样接受多个文件参数, 也是以一个目录作为最后的参数。而且在某些系统中, mv、cp和ln自身也链接到一个独立的文件上, 然后检查它的名称来看提供哪项服务。

练习2-6 为什么ls -ld报告recipes有4条链? 提示: 试一下

```
$ ls -ld /usr/you
```

为什么这是一个有用的信息?

练习2-7 下面两者何不同

```
$mv junk junk1
```

和

```
$cp junk junk1
$rm junk
```

提示: 给junk 建立一条链, 然后试一下。

练习2-8 cp不能复制子目录中的文件, 只复制给出的第一级目录中的文件。那如果参数文件是一个目录它会怎样做呢? 这是合理或者甚至明智的吗? 讨论下面三种可能性的优点: 给cp提供一个选项以进入到子目录; 用一个单独的命令rcc(递归复制)来完成工作; 或让cp在发现目录后递归地复制。有关这项功能的帮助参见第7章。此外还有哪些程序会从遍历目录树的能力中获益呢?

2.6 目录层次

在第1章里, 我们从/usr/you开始, 非正式地介绍了一下文件系统的层次结构, 现在, 我们要以一种更为有序的方式对其进行探讨, 从树的顶部——根——开始。

目录的顶部是/:

```
$ ls /
bin
boot
dev
etc
lib
tmp
unix
usr
$
```

/unix是用于UNIX内核本身的程序。当系统启动时, /unix被从磁盘上读入到内存中并启动。事实上, 启动过程分为两步: 首先文件boot被读入, 接着读取/unix中的信息。有关这个“引导程序”进程更多的信息可以参见boot(8)。/中其余的文件, 在这里都是目录, 在某种程度上每一个都是整个文件系统包含的一部分。下面对层次结构作简短浏览的同时将会对已经提到的目录作一些探讨。对文件系统的布局越熟悉, 使用起来就更有效。表2-1列出了一些有意义的目录, 其中一些目录名称与系统有关。

表2-1 有意义的目录(参见hier(7))

目 录	含 义
/	文件系统的根
/bin	可执行形式的基本程序(二进制)
/dev	设备文件
/etc	系统各种管理文件
/etc/motd	有关日期的登录消息
/etc/passwd	口令文件
/lib	基本库等
/tmp	临时文件, 当系统重新启动时清除
/unix	操作系统的可执行形式
/usr	用户文件系统
/usr/adm	系统管理: 计数信息等
/usr/bin	用户库: troff等
/usr/dict	词典(单词)和拼写spell(1)支持
/usr/games	游戏程序
/usr/include	C程序的头文件, 如math.h
/usr/include/sys	C程序的系统头文件, 例如inode.h
/usr/lib	C、FORTRAN等的库
usr/man	在线手册
/usr/man/man1	在线手册第1节
/usr/mdec	硬件诊断程序、引导程序等
/usr/news	公众服务信息
/usr/pub	公共信息, 见ascii(7)和eqnchar(7)
/usr/src	工具和库源代码
/usr/src/cmd	/bin和/usr/bin中命令资源
/usr/src/lib	子例程库源代码
/usr/spool	通信程序的工作目录
/usr/spool/lpd	行式打印机临时目录
/usr/spool/mail	邮箱
/usr/spool/uucp	uucp 程序的工作目录
/usr/sys	操作系统内核资源
/usr/tmp	后备临时目录(很少使用)
/usr/you	你的登录目录
/usr/you/bin	你的个人程序

- /bin(binaries) 我们以前已经见过, 该目录中存放最基本的程序, 如who和ed。
- /dev(devices) 将在下一节中讨论。
- /etc(et cetera) 前面也已经见过。它包含各种管理文件, 如口令文件和一些系统程序如/etc/getty, 该程序为/bin/login初始化终端连接。/etc/rc是一个shell命令文件, 它在系统启动后执行。/etc/group 列出每组的成员。
- /lib(library) 包含C编译器的主要部分, 如C预处理器lib/cpp和C的子例程库/lib/libc.a。
- /tmp(temporaries) 是在程序的执行过程中创建临时文件的仓库。例如, 在启动编辑器ed时, 它创建一个名字类似于/tmp/e00512的文件作为正在编辑的文件的备份, 而不是直接工作在原始文件上。当然, 它也可以把文件创建到当前目录下, 但是把它放在/tmp内要更好一点: 尽管不太可能, 但还是应该考虑到在当前目录下已经有了一个文件e00512。系统启动时, /tmp自动清除, 所以即使系统崩溃, 你的目录也不会有个未预料到的文件。通常/tmp被安排

在磁盘上以进行快速访问。

当然，这里还存在一个问题，当几个程序同时在/tmp内创建文件时，它们有可能互相干扰对方的文件。这就是为什么ed的临时文件会有一个特殊名称，它的构造方式保证了没有其他程序会选择与它的临时文件相同的名称。在第5章和第6章，我们会看到实现的方法。

/usr 被称作“用户文件系统”，尽管它也许和系统真正的用户没关系。在我们的机器上，我们的登录目录是/usr/bwk和/usr/rob，但是在你的机器上/usr部分很可能不同，这一点已经在第1章中作过解释。无论个人目录是否位于/usr的一个子目录下，你还是可以从那里找到许多东西。和在/中一样，这里也有目录被称作/usr/bin、/usr/lib和/usr/tmp。这些目录所具有的功能类似于它们在/中的同名目录，只是它们所包含的程序对系统而言不那么关键。例如，nroff通常位于/usr/lib而不是/bin。当然，什么称作“关键”随系统的不同而不同。有些系统，如分布式的第7版，所有的程序都位于/bin下，而去掉了/usr/bin；而某些系统根据使用频率的不同把/usr/bin分成两个目录。

/usr中其他的目录是/usr/adm和/usr/dict，前者包含了帐目而后者包含了一个简单的字典(见spell(1))。在线手册保存在/usr/man中，例如，可以参见/usr/man/man1/spell.1。如果系统中有在线源码，就有可能在/usr/src中找到它。

值得花一点儿时间来研究文件系统，特别是/usr，这样就对文件系统的组织和在哪里可以找到东西有一些感性认识。

2.7 设备

在上一节中，我们忽略掉了/dev，因为那只是在总体上对文件提供了一个很好的浏览。顾名思义，/dev 包含了设备文件。

在UNIX系统中非常绝妙的想法之一就是它对外设——磁盘、磁带设备、行式打印机、终端等的处理。例如，在读磁带时它并没使用特殊的系统例程，而是提供了一个称作/dev/mt0(同样，不同系统可能不同)的文件。在内核内部，对该事件的引用转变成访问磁带的硬件命令，所以如果一个进程读/dev/mt0，安装在驱动器上的磁带的內容就被返回。例如：

```
$ cp /dev/mt0 junk
```

把磁带的內容复制到一个称作junk的文件中，cp并没有意识到/dev/mt0有什么不同，它只是一个文件——一个字节序列。

设备文件有些像一个动物园，每个动物都有些不同，但是文件系统的基本思想仍然在每个上面得到了体现。下面是/dev的一个很重要的简短列表：

```
$ ls -l /dev
crw--w--w- 1 root      0,  0 Sep 27 23:09 console
crw-r--r-- 1 root      3,  1 Sep 27 14:37 kmem
crw-r--r-- 1 root      3,  0 May  6 1981 mem
brw-rw-rw- 1 root      1, 64 Aug 24 17:41 mt0
crw-rw-rw- 1 root      3,  2 Sep 28 02:03 null
crw-rw-rw- 1 root      4, 64 Sep  9 15:42 rmt0
brw-r----- 1 root      2,  0 Sep  8 08:07 rp00
brw-r----- 1 root      2,  1 Sep 27 23:09 rp01
crw-r----- 1 root     13,  0 Apr 12 1983 rrp00
crw-r----- 1 root     13,  1 Jul 28 15:18 rrp01
crw-rw-rw- 1 root      2,  0 Jul  5 08:04 tty
```

```
crw--w--w- 1 you      1,  0 Sep 28 02:38 tty0
crw--w--w- 1 root     1,  1 Sep 27 23:09 tty1
crw--w--w- 1 root     1,  2 Sep 27 17:33 tty2
crw--w--w- 1 root     1,  3 Sep 27 18:48 tty3
$
```

首先，要引起注意的是这里用来代替字节计数的是一对数值较小的整数，而且模式的第一个字符通常为 b 或 c。ls就是这样从一个说明设备而不是普通文件的I节点输出信息。普通文件的I节点包含了一些用于存储文件内容的磁盘块。而对于一个设备文件，I节点中包含的是设备的内部名称，它由设备的类型——字符(c)或块(b)——和一对数字组成，这对数字称作主和次设备号。磁盘和磁带是块设备，其他的——如终端、打印机、电话线等等——都是字符设备。主设备号包含了设备的类型，而次设备号则用于区分设备的实例。例如，/dev/tty0和/dev/tty1是同一个终端控制器上的两个端口，所以它们有着相同的主设备号和不同的次设备号。

磁盘文件通常是它们所表示的特定硬件变量来命名的。/dev/rp00和/dev/rp01是根据与系统相连的DEC rp06磁盘驱动器来命名的。只有一个物理驱动器，但在逻辑上分成两个文件系统。如果有第2台驱动器，与它相连的驱动器应该被命名为/dev/rp10和/dev/rp11。头一个数字说明物理设备，第二个说明设备分区。

你有可能想知道，为什么会有几个磁盘文件系统而不是只有一个。出于历史的原因或者是为了易于维护，文件系统被分成了更小的子系统。子系统文件可以通过系统中的一个目录来访问。程序/etc/mount报告设备文件和目录之间的对应关系：

```
$ /etc/mount
rp01 on /usr
$
```

在上述例子中，根系统占用了/dev/rp00(尽管/etc/mount没有报告出这一点)，而用户的文件系统——在/usr和它的子目录中的文件——保留在/dev/rp01中。

系统执行时根文件系统必须存在。/bin、/dev和/etc通常都保留在根系统中，因为在系统启动的时候只有根系统中的文件才是可以访问的，而一些文件诸如/bin/sh自始至终都需要运行。在启动操作期间，要检查所有文件系统的自一致性(见icheck(8)或fsck(8))，并且把它们附加到根系统。这种附加操作被称作安装，与在一个驱动器上安装一个新的磁盘包相同，通常它只能由超级用户来完成。在/dev/rp01作为/usr被装上以后，用户文件系统中的文件就可以被访问，就好像它们是根系统的一部分一样。

对于普通用户来说，哪个文件子系统该装在哪儿这种细节问题不必考虑，但还有几点需要注意。首先，由于子系统可以被装上和卸载，那么做一条链使它指向另一个子系统文件就是不合法的。例如，不可能把/bin下的程序链接到用户私有的bin目录中的文件上去，因为/usr和/bin位于不同的文件子系统。

```
$ ln /bin/mail /usr/you/bin/m
ln: Cross-device link
$
```

还存在的一个困难是不同文件系统中的I节点号也不相同。

其次，一些子系统大小(文件可以使用的块数)和I节点有着固定的上限。如果一个子系统被填满了，就不可能再扩大该子系统中的文件，除非回收一些空间。df(disc free space)命令报告已装上的文件子系统上可利用空间的情况：

```
$ df
/dev/rp00 1989
/dev/rp01 21257
$
```

/usr有21257个空余块。空间是多还是少依赖于系统怎样被使用。一些安装程序要比其他程序需要更多的文件空间。顺便提一句，在所有的命令中，df的输出格式可能具有最大的变化。你的df输出可能看上去完全不同。

让我们转向一些更经常使用的东西。在登录的时候，你得到了一个终端行，因而也就在/dev下有一个文件，通过这个文件你所键入和接收的字符可以被发送出去。tty命令用于报告正在使用的是哪个终端：

```
$ who am i
you      tty0      Sep 28 01:02
$ tty
/dev/tty0
$ ls -l /dev/tty0
crw--w--w- 1 you      1, 12 Sep 28 02:40 /dev/tty0
$ date >/dev/tty0
Wed Sep 28 02:40:51 EDT 1983
$
```

注意，既然你拥有设备，那么就只有你才允许去读它。换句话说没有人可以直接读你刚才所键入的字符。然而，任何人都可以在你的终端上写。为防止这一点，可以chmod设备，从而防止人们使用write命令与你联系，或者只使用mesg。

```
$ mesg n          切断消息
$ ls -l /dev/tty0
crw----- 1 you      1, 12 Sep 28 02:41 /dev/tty0
$ mesg y          恢复
$
```

能够按名称来引用正在使用的终端通常都很有用，但不方便的是无法决定它究竟是哪一个。设备/dev/tty是你所登录的终端的同义词，而不论实际使用的到底是哪个。

```
$ date >/dev/tty
Wed Sep 28 02:42:23 EDT 1983
$
```

当程序需要和用户进行交互时，/dev/tty特别有用，尽管它的标准输入和输出是与文件而不是终端相连。crypt是使用/dev/tty的一个程序。“明码文本”从标准输入进来，加密后的数据被送到标准输出，所以crypt从/dev/tty中读取密钥：

```
$ crypt <cleartext >cryptedtext
Enter key:          输入密钥
$
```

在本例中对/dev/tty的使用并不很清楚，但它确实在那儿。如果 crypt从标准输入中读密钥，会把明码文本的第一行读入。所以，作为一种替换的方法，crypt打开/dev/tty，关闭自动字符回显——这样密钥就不会在屏幕上显示，并读入密钥。在第5章和第6章，我们会讨论/dev/tty的其他一些用法。

有时你可能会想要运行一个程序，却并不关心它的输出是什么。例如，可能你已看过了今天的新闻，不想再看了。把news重定向到文件/dev/null会使它的输出被抛弃：

```
$ news >/dev/null
$
```

写到/dev/null的数据不带任何注释地被扔掉，当程序从/dev/null读的时候，会立刻得到一个文

件结束信息，因为从/dev/null读总是返回零字节。

/dev/null的一个常用用法是抛弃正规的输出使诊断消息可见。例如，time命令 (time(1))报告程序的CPU使用程度。消息被打印出到标准错误输出设备，这样可以通过把标准输出发送到/dev/null来对产生过多输出的命令计时：

```
$ ls -l /usr/dict/words
-r--r--r-- 1 bin 196513 Jan 20 1979 /usr/dict/words
$ time grep e /usr/dict/words >/dev/null

real    13.0
user    9.0
sys     2.7
$ time egrep e /usr/dict/words >/dev/null

real    8.0
user    3.9
sys     2.8
$
```

time输出中的数字是逝去的时钟时间、程序所花费的CPU时间以及程序运行时内核所花费的CPU时间。egrep是grep的一个功能更强大的变体，我们将在第4章进行讨论。当在一个大文件中搜索的时候，它的速度要比grep快两倍。如果grep和egrep的输出没有被送到/dev/null或一个实际的文件，我们就需要等到上千个字符在终端上出现以后才能发现要寻找的时间信息。

练习2-9 通过阅读手册的第4节寻找/dev中的其他文件。在/dev/mt0和/dev/rmt0之间有什么区别？对在/dev下为磁盘、磁带等分别建立子目录的潜在优点进行评论。

练习2-10 非UNIX系统写出的磁带通常具有不同的块尺寸，如800字节——10个80-字符的卡片映像，但是磁带设备/dev/mt0期待的是512-字节的块。查看dd命令(dd(1))以解决怎样读这种磁带的问题。

练习2-11 为什么/dev/tty不是一条指向登录终端的链呢？如果它和登录终端一样模式为rw-w--w-会怎样呢？

练习2-12 write(1)是怎样工作的？提示：参见utmp(5)。

练习2-13 你怎样判断是否一个用户已经在终端激活？

有关文献

在Ken Thompson(BSTJ, 1978年7月)所著的“UNIX implementation”中讨论了文件系统。Dennis Ritchie所作的论文，名为“The evolution of the UNIX time-sharing system”(Symposium on Language Design and Programming Methodology, Sydney, Australia, 1979年9月)描述了文件系统在最初的PDP-7 UNIX系统上的设计与实现，以及它怎样发展成现在的形式。

UNIX文件系统从MULTICS文件系统中借鉴了一些思想。“The MULTICS System: An Examination of its Structure”(E.I.Organick, MIT出版社, 1972年)提供了MULTICS的全面方案。

Bob Morris和Ken Thompson所著的“Password security :a case history”对各种系统上的口令机作了一个有趣的比较，这篇文章可以在《UNIX 程序员手册》卷2B中找到。

在同一卷中，Dennis Ritchie所著的“On the security of UNIX”解释了系统安全是怎样更依赖于管理员的小心谨慎，而不是依赖于像crypt这样的程序细节。

第3章 shell的使用

shell是对用户提出的运行程序的请求进行解释的程序。对大多数UNIX用户而言, shell是系统中最重要程序。除了使用文本编辑器以外, 用户大部分时间都在shell中工作。本章和第5章将主要讲述shell的功能特性。一旦学会使用shell, 就可以不必求助于传统语言(如C)编程, 而解决很多复杂的问题。

有关shell的内容将分成两章讲述。本章在第1章的基础上, 进一步讨论shell的常用特性, 例如元字符、引号、创建新命令、传递shell参数、使用shell变量以及一些基本的控制流等。第5章的内容更加深入, 它介绍了一些重要的shell程序, 这些程序可以防止其他用户使用。应该通读两章, 才能完全地掌握 shell。

3.1 命令行结构

作为shell的预备知识, 首先用户需要了解什么是命令行, shell又是如何解释命令行的。与第1章中的基本介绍相比, 本节的内容更深入。

最简单的命令就是一个字(word), 通常是一个可执行的文件名(以后还会见到其他类型的命令):

```
$ who                                执行文件/bin/who
you      tty2      Sep28 07:51
jpi      tty4      Sep28 08:32
$
```

一条命令通常以换行结束, 但有时分号; 也可用作命令结束符:

```
$ date;
Wed Sep 28 09:07:15 EDT 1983
$ date; who
Wed Sep 28 09:07:23 EDT 1983
You      tty 2     Sep 28 07:51
jpi      tty 4     Sep 28 08:32
$
```

尽管分号可以用作命令结束符, 但通常只有按回车键后命令才开始执行。注意, shell在多个命令执行后通常只打印一个提示符, 而且

```
$ date; who
```

执行的效果等同于在不同行上键入命令, 即who命令要等到date执行完后才开始执行。

可以将date;who的输出送到一个管道:

```
$ date; who | wc
Wed Sep 28 09:08:48 EDT 1983
      2      10      60
$
```

这样的结果也许和用户所想的不大一样, 因为只有who的输出被送到了wc。通过管道将who和

wc连成了一条命令，称为管道线命令，它在date之后运行。shell分析命令行时，认为|的优先级高于;。

括号可以用来组合命令：

```
$ (date; who)
Wed Sep 28 09:11:09 EDT 1983
you      tty2      Sep 28 07:51
jpl      tty4      Sep 28 08:32
$ (date; who) / wc
      3      16      89
$
```

date和who的输出被串连成单个数据流，送入一个管道。

tee命令可以将通过管道的数据流截获并放进某个文件(而不是另一个管道)。tee不是shell的一部分，但它有助于对管道进行操作。它的用途之一就是将中间结果保存到一个文件中去：

```
$ (date; who) | tee save | wc
      3      16      89          WC的输出
$ cat save
Wed Sep 28 09:13:22 EDT 1983
you      tty2      Sep 28 07:51
jpl      tty4      Sep 28 08:32
$ wc <save
      3      16      89
$
```

tee将它的输入复制到一个命名的文件或文件组，同时把输入复制到输出，因此wc可以接收到同样的数据，就好像tee在管道线中并不存在一样。

另一个命令结束符是&。它与换行符或分号的功能基本一致，但它可以告诉shell不必等待命令结束。通常，&将一个需长时间运行的命令放入后台，而用户可以在此期间继续在前台与shell交互：

```
$ long-running-command &
5273          long-running-command的进程id
$             立即出现提示符
```

利用命令组合，可以实现后台进程的一些有趣的应用。sleep 命令用来等待一段时间，然后退出：

```
$ sleep 5
$          5秒钟后出现提示
$ (sleep 5; date) & date
5278
Wed Sep 28 09:18:20 EDT 1983      第二个date的输出
$ Wed Sep 28 09:18:25 EDT 1983    出现提示符，5秒钟后出现第一个date的结果
```

该后台进程一开始运行就立即进入睡眠状态，同时，第二个date命令打印出当前时间和 shell 提示符，等待键入新的命令。5秒钟以后，sleep命令完成，第一个 date命令打印出新时间。这个过程的时间变化很难在纸上描述清楚，用户应该在机器上试一下这个例子。(由于机器系统的负载情况不同，以及其他一些因素的影响，这两个时间之间的差可能不会恰好是5秒。) 这样可以很容易写出一个在将来运行的命令：

```
$ (sleep 300; echo Tea is ready) &          5分钟后打印出 "Tea is ready"
5291
$
```

上述命令可以用于提醒功能。(正如ctrl_g将引起终端响铃一样。)由于 & 的优先级高于 ; , 所以上述例子中的括号是必须的。

&字符表示命令的终止, 既然管道线也是命令, 因此安排管道线到后台执行的时候可以不用括号。如下命令:

```
$ pr file / lpr &
```

将一个文件放到打印机上输出, 而用户无需等待该命令的结束。将该管道线用括号括起来会达到同样的效果:

```
$ (pr file / lpr) &          结果与上例相同
```

大多数程序接受命令行参数, 如上例中的file(作为pr的一个参数)。参数是由空格和表格符分隔的单词, 通常是命令所要处理的文件名, 程序可以用任意合理的方式解释这些参数。例如, pr接受文件名进行打印, echo不加解释地返回它的参数, 而grep的第一个参数指出了要进行搜索的文本模式。并且, 大多数程序提供了选项, 这些选项均以带负号的参数表示。

由shell解释的各种特殊字符, 如<、>、|、; 和&等, 不能用作在shell中程序的参数。相反, 它们指示了shell运行这些程序的方式。例如:

```
$ echo Hello >junk
```

告诉shell, 运行一条带单个参数 Hello的命令 echo, 并把输出放入文件 junk。字符 >junk 不是echo的参数, 它由 shell解释, 而 echo 根本看不见它。事实上, 它甚至不一定放在命令行的最后, 如:

```
$ >junk echo Hello
```

结果与上例是一样的, 但不如上例直观。

练习3-1 下列三条命令有什么不同?

```
$ cat file | pr
$ pr <file
$ pr file
```

(近年来, 重定向符<已经逐渐为管道所取代, 人们更倾向于使用“cat file |”而不是“<file”。)

3.2 元字符

shell认为有一些字符是特殊字符, 最常见的就是字符 *, 它指示 shell查找这样的文件目录, * 的位置上可为任何字符串。例如,

```
$ echo *
```

的功能与ls大致相同。在第1章中忘了提及一点, 就是文件名匹配字符串不能匹配以点 . 开头的文件名, 以避免与每个目录中的 . 和 .. 相匹配, 而带来副作用。如果匹配模式中出现了 .., 则只匹配以 . 开头的文件名。以echo命令为例, 可区分这两种情况:

```
$ ls
.profile
junk
temp
$ echo *
junk temp
$ echo .*
. . . .profile
$
```

像*这样在shell中有特殊用途的字符称为元字符(metacharacter)。这样的字符有很多, 表3-1列出了全部元字符, 其中的一些将在第5章讨论。

表3-1 shell元字符

元 字 符	含 义
>	prog>文件 将标准输出重定向到文件
>>	prog>> 文件 将标准输出附加到文件
<	prog< 文件 从文件file中获取标准输入
	$p_1 p_2$ 将 p_1 的标准输出作为 p_2 的标准输入
<<串	here document: 标准输入从here document读入, 直到出现串
*	匹配文件名中的零个或多个字符
?	匹配文件名中的任何单个字符
{ccc}	匹配文件名中ccc范围内的任何字符, 如0-9或a-z都是合法的
;	命令结束符, 例如 $p_1; p_2$ 表示先执行 p_1 , 再运行 p_2
&	与; 类似, 但不等 p_1 结束
`...`	运行...中的命令; 输出结果代替`...`
(...)	在子shell里运行...中的命令
{...}	在当前shell 中运行...中的命令(很少使用)
\$1, \$2 等	\$0...\$9 可代表shell 文件的参数
\$变量	shell 变量的值
\${变量}	变量的值; 为避免在文本联接时混淆, 请参照表5-3
\	\C 将C 字符作为字符, 但\后面加换行符无效
'...'	...表示文字
"..."	在...中的\$, `...` 和\得到解释后, 将...作为文本文字
#	表示注释的开始(第7版除外)
变量 = 值	为变量赋值
$p_1 \& \& p_2$	运行 p_1 ; 若成功, 再运行 p_2
$p_1 p_2$	运行 p_1 ; 若不成功, 再运行 p_2

元字符不作元字符用时, 可以有几种方法来表示它们。防止它们被shell解释的最简便的方法, 就是用单引号将它们括起来:

```
$ echo '***'
***
$
```

也可以用双引号"...", 但shell会在双引号中寻找\$, `...` 以及\符号, 因此如果用户不想处理引号中的字符串, 就不要使用双引号。

第三种方法是在每个元字符前放一个反斜杠符\, 如

```
$ echo \* \* \*
```

尽管***看起来不太符合英语的习惯, 但shell认为它仍然是一个词。shell接受这样的词作为处理的基本单位, 而且引号中的词还可以包括空格。

一种引号可以保护另一种引号:

```
$ echo "Don't do that !"
Don't do that !
$
```

而且引号也不必包含整个参数:

```
$ echo x '*' y
```

```
x*y
$ echo '* A ?'
*A?
$
```

在上例中，引号被用作功能符号，echo将忽略它们，而输出一个无引号的参数。

引号中的字符串可以包括换行符：

```
$ echo 'hello
> world'
hello
world
$
```

字符>是一个辅助提示符，由shell给出，意思是希望用户键入更多的信息以完成该命令行的输入。在这个例子中，第一行的引号应该与另一行的引号配对。辅助提示符存放在shell变量PS2中，可以根据需要修改。

在所有这些例子中，一个元字符被引号括起来的作用就是防止shell解释它。命令

```
$ echo x*y
```

将返回所有以x开头，以y结尾的文件名。实际上echo命令并不区分文件名或shell元字符，*的解释是由shell执行的。

如果没有文件匹配该模式，会出现什么结果呢？shell会将该字符串直接放在参数位置，就好像它被引号括起来了一样。这样的处理虽然不太合理，但它可以用来帮助发现匹配某模式的文件名：

```
$ ls x*y
x*y not found          来自ls的消息：没有这样的文件
$ >xyzzy               创建xyzzy文件
$ ls x*y
xyzzy                  文件xyzzy与x*y匹配
$ ls 'x*y'
x*y not found          ls不解释*
$
```

在一行末尾加入反斜杠\可以表示该行未完，这可以避免shell中的输入行过长。

```
$ echo abc\
> def\
> ghi
abcdefghi
$
```

注意反斜杠后面的换行符将被忽略，但如果换行符出现在引号中，则仍然保留。

元字符#用作shell注释符，若一个shell词以#开始，该行余下的部分则被忽略：

```
$ echo hello # there
hello
$ echo hello#there
hello#there
$
```

#号不是第7版的内容，但它已经被大多数版本广泛接受，所以在本书余下的部分将用#作为注

释符。

练习3-2 描述下列命令的输出

```
$ ls .*
```

关于echo的话题

echo在输出后通常都会换行，而不管用户是否需要，更合理的 echo命令应该只打印出用户要求打印的内容，这样引起下列结果：

```
$ pure-echo Enter a command:
Enter a command:$
```

结尾不换行

实际上，大部分情况下需要换行，因此就必须特别地指明：

```
$ pure-echo 'Hello!'
>
Hello!
$
```

既然一条命令在缺省状态下应该完成它最常用的功能，因此echo命令总是自动换行。

但如果用户并不希望换行呢？第7版中的echo有一个选项参数-n，可以去掉最后的换行：

```
$ echo -n Enter a command:
Enter a command:$
```

提示符出现在同一行上

```
$ echo -
-
$
```

只有-n时例外

若要打印出-n并后换行，则：

```
$ echo -n '-n'
>
-n
$
```

虽然看起来很奇怪，但也只好如此了，毕竟这种情况十分罕见。

系统V中采用另一种方法，让echo 用与C类似的方法解释 \开头的字符，如\b表示退格，\c(实际上C语言中没有这个字符)表示去掉换行：

```
$ echo 'Enter a command:\c'
Enter a command:$
```

系统V版本

尽管这种机制可以避免在打印-号时引起混乱，但它会造成其他问题。echo通常用于帮助诊断错误，但许多程序都对反斜杠开始的字符时进行解释，若echo也这样做，则只有增加混乱而已。

因此，echo命令的两种设计方案都各自有优缺点，我们将使用版本7的echo(-n)，所以，如果你的系统没有采用这个约定，一些程序将需要做轻微的改动。

另一个问题是，如果没有参数，echo应该输出什么结果？它是应该打印一个空行呢，还是什么都不输出？现在几乎所有的实现方案都将打印空行，但在过去的版本中则不是这样。实际上，历史上曾就此有过相当大的争议。Doug McIlroy在讨论这一主题时诉说了对神秘主义的正确感受：

UNIX和回应

在新泽西住着一位受到各地来的智者们赞赏的美丽的少女，名叫UNIX。他们都被她的美

丽所倾倒，被她的优雅或举止所迷恋，佩服她轻松地完成那些在富人圈子中看来极困难的工作的能力。具有包容心和天性宽厚的UNIX，接受了所有富有的求婚者。

用发自内心的微笑回答是UNIX最愿做的事。实际上，谁在乎那些宫廷方式呢，人们喜欢她准确而又清晰的回答，因为都知道她不会象岩石和森林那样，对他们在野外的呼喊给予扭曲的回应。

当急性子的年青情人呼唤UNIX时，“回答没有”，UNIX按规矩开口，之后又闭口。“不论你的意思是什么，”急性年青人命令道，“就那样开口怎么行？今后，如果要回答没有，那就决不要张开嘴巴！”于是UNIX照办了。

“但是我需要一种完美的方式，那怕是回答没有。”一名敏感的青年恳求说。“从闭着的嘴里不会有完美的回声。”为了不冒犯他们中的任何一位，UNIX同意对急性青年和敏感青年以不同的方式说没有。她把对敏感青年用的没有称为\n。

这样当她说\n时，并不是真的什么也没说，她不得不开口两次，一次得说\n，一次得说没有，这样，敏感的青年人还是不高兴，他立刻说，“我觉得第一个看来很完美，但是第二个却把一切都毁了。我要你收回其中的一个。”UNIX可是不能接受这种冒犯，答应有些回应可以修改，改为\c。现在这位敏感的青年可以通过要求\n和\c，从而听到一种对没有的完美的回应了。只不过人们说，在听到这一串过分长的标记之前，这位年青人就死了。

练习3-3 试指出下述哪一个grep命令会成功执行，并验证结论。

grep \\$	grep \\\
grep \\\$	grep \\\\\$
grep \\\\$	grep "\\$"
grep '\\$'	grep '\$'
grep '\\$'	grep "\$"

包含这些命令的文件将是一个好的测试用例。

练习3-4 用户应该如何让grep搜寻以-开始的一个匹配模式？为什么不能将参数用引号括起来？提示：参看-e选项。

练习3-5 考虑

```
$ echo */*
```

这会打印出所有目录中的所有文件名吗？这些文件会以何种顺序出现？

练习3-6 你如何将/放入某文件名(也就是说，/不用作路径的分隔符)？

练习3-7 以下命令分别得到什么结果？在运行前先考虑一下。

```
$ cat x y >y
```

和

```
$ cat x >>x
```

练习3-8 若你键入

```
$ rm *
```

为什么rm不会警告你，这将会删除你的所有文件？

3.3 创建新命令

现在来讨论一下在第1章提过的，如何利用现有命令创建新命令。

假如一个命令序列要被反复执行多次，那么最好把它组织成一个新命令，这样就可以像使用常规命令那样执行它。举例来说，假如想要用管道线命令统计用户数量：

```
$ who | wc -l
```

这在第1章提到过，你也可以创建一个名为nu的新程序来完成这件工作。

第一步是创建一个包含who | wc -l的普通文件。可以选用任意喜欢的文本编辑器，或用下述方法生成：

```
$ echo 'who | wc -l' > nu
```

(考虑一下，若没有引号，nu中将会是什么内容？)

shell和编辑器、who或wc一样，也是一个程序，名字叫sh。既然这是一个程序，你就可以运行它，并重定向它的输入。因此，运行shell，以文件nu作为输入，而不是从终端输入：

```
$ who
you      tty2      Sep 28 07:51
rhh      tty4      Sep 28 10:02
moh      tty5      Sep 28 09:38
ava      tty6      Sep 28 10:17
$ cat nu
who | wc -l
$ sh <nu
      4
$
```

输出结果与在终端上键入who | wc -l 所得到的一样。

同时，也如同大多数程序一样，shell可以从一个文件中取得输入，只要该文件名作为一个参数出现。下列语句

```
$ sh nu
```

也可得到同样的结果。但命令中的sh并不是必需的，有了它之后，命令太长，且这样一来，就不能以同样的方式对待高级语言和shell命令编写的程序[⊖]。因此，如果一个文件是可执行的，且它包含文本文件，则shell认为这是一个shell命令组成的文件，称为shell文件。用户所需要做的只是使nu可执行，进行如下操作：

```
$ chmod +x nu
```

然后可以执行它：

```
$ nu
```

到目前为止，仅仅从运行的角度，nu的用户并不能分辨出你这么容易地实现了nu。

但实际上shell仍要创建一个新的shell进程来运行nu，其效果就如同你键入了

```
$ sh nu
```

一样。这个新的shell称为“子shell”，它是由当前shell激活的子shell进程。sh nu与sh <nu不同，因为它的标准输入仍然来自终端。

一般，nu只有在当前工作目录中才能执行(假设工作目录由PATH设置)。为了使nu可以在任何工作目录下运行，可将它移到bin目录下，并将/usr/you/bin放入查找路径：

```
$ pwd
/usr/you
$ mkdir bin
```

创建一个bin目录，若当前还没有该目录的话

⊖ 在其他大多数操作系统中，高级语言和系统shell命令编写的程序确实存在着区别。

```

$ echo $PATH          检查PATH
: /usr/you/bin:/bin:/usr/bin 显示PATH
$ mv nu bin           安装nu
$ ls nu
nu not found          nu已从当前目录中移走
$ nu
4                     但shell可以找到它
$

```

当然, PATH值应该在.profile中正确设置, 这样就不必在每次登录时都手工重置PATH。

还可以用这种方法创建一些其他的简单命令, 以便根据自己的需要配置工作环境。这包括:

- cs 回送清除终端显示屏幕的字符序列(通常每幅为24行)。
- what 运行 who 和ps -a, 返回当前登录的用户以及他们正在做的工作。
- where 打印用户当前使用的UNIX系统的名字。(设置PS1也可达到同样的目的。)

练习3-9 查看一下/bin和/usr/bin目录中有多少是真正的shell文件。可以用一条命令完成这件事吗? 提示: 参见file(1)。若用文件长度作依据, 准确性如何?

3.4 命令参数

尽管nu不需要参数, 但大部分shell程序都有参数, 例如运行程序所需的文件名和选项等等。

假设用户想要创建一个名为cx的程序, 将一个文件的权限改为可执行的, 即

```
$ cx nu
```

是

```
$ chmod +x nu
```

的缩写。创建的步骤你已经很清楚了, 首先需要创建一个文件, 其内容是:

```
chmod +x 文件名
```

与原来不同的是, 需要告诉cx文件名, 这样cx的每一次运行才能对不同的文件进行操作。

Shell执行一个命令文件时, 用第一个参数代替\$1, 用第二个参数代替\$2, 如此下去直到\$9。因此若文件cx内容如下:

```
chmod +x $1
```

那么运行命令

```
$ cx nu
```

时, 子shell就用nu代替\$1。

现在看一下整个创建过程:

```

$ echo 'chmod +x $1' >cx
$ sh cx cx
$ echo echo Hi, there! >hello
$ hello
hello: cannot execute
$ cx hello
$ hello
Hi, there!
$ mv cx /usr/you/bin
$ rm hello
$

```

首先创建cx
使cx本身可执行
创建一个测试程序hello
测试

使hello可执行
再次测试
它可以工作
安装cx
去掉hello

注意若cx已经可执行，shell自动进行操作的准确表述应是：

```
$ sh cx cx
```

将cx变为可执行文件。这等同于cx已经可执行时，shell自动所做的动作，或用户键入

```
$ cx cx
```

如果想要处理多个参数，例如想让cx程序能一次处理多个文件，又该怎么办呢？一种办法是在shell程序中放入9个参数，如下：

```
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```

(这只能处理到\$9，因为\$10被shell分析成“第一个参数\$1，后接0”！)如果该shell文件的使用者提供了少于9个的参数，缺少的部分作为空串处理，这样只有实际提供的参数值被shell传递给chmod命令。因此这个方法是可以采纳的，但它显然是不够的，若用户提供了多于9个的参数时，文件就不能正确运行。

针对这一问题，shell提供了一个缩写\$*，表示“所有的参数”。这样，就可以如下定义cx：

```
chmod +x $*
```

无论给出多少参数，该命令都能正确运行。

利用\$*，可以更方便的生成shell文件，如lc或m：

```
$ cd /usr/you/bin
$ cat lc
# lc: count number of lines in files
wc -l $*
$ cat m
# m: a concise way to type mail
mail $*
$
```

这两个文件都可以不带参数。无参数时，\$*为空串，没有参数会传给wc或mail。不论有无参数，该命令都可以正确执行：

```
$ lc /usr/you/bin/*
 1 /usr/you/bin/cx
 2 /usr/you/bin/lc
 2 /usr/you/bin/m
 1 /usr/you/bin/nu
 2 /usr/you/bin/what
 1 /usr/you/bin/where
 9 total
$ ls /usr/you/bin | lc
6
$
```

这些命令和本章中的其他命令都是个人程序的例子，是根据个人需要编写的，并放入个人的bin目录中。它们不是通用的，因为这完全取决于个人喜好。在第5章中将介绍如何编写公用的shell程序。

一个shell文件的参数不一定是文件名。例如，如想要查找个人电话目录，假设你有一个名为/usr/you/lib/phone-book的文件，包含以下行：

```
dial-a-joke 212-976-3838
dial-a-prayer 212-246-4200
dial santa 212-976-3636
dow jones report 212-976-4141
```

可以使用grep命令进行查找。(lib目录是存放这些个人数据的好地方。)既然grep不关心信息的格式，你就可以搜索名字、地址、邮编或其他任何想要的内容。现在写一个查号员程序，用

查号员的代号411命名:

```
$ echo 'grep $* /usr/you/lib/phone-book' >411
$ cx 411
$ 411 joke
dial-a-joke 212-976-3838
$ 411 dial
dial-a-joke 212-976-3838
dial-a-prayer 212-246-4200
dial santa 212-976-3636
$ 411 'dow jones'
grep: can't open jones
$
```

出错

最后一个例子揭示出一些潜在的问题: 尽管dow jones对411而言是一个参数, 但它包含了一个空格, 而引号不再起作用, 从而解释命令411的子shell将其转化成两个参数, 并传递给grep, 就好像你键入了:

```
$ grep dow jones /usr/you/lib/phone-book
```

一样, 而这显然是错的。

一个补救措施是使用shell的双引号。尽管单引号内部的字符串不加解释, 但shell会在双引号里寻找\$、\和`...`等特殊字符。因此可以修改411:

```
grep "$*" /usr/you/lib/phone-book
```

这样, \$*会被参数代替, 但它将作为一个参数传给grep, 即使它包含空格。

```
$ 411 dow jones
dow jones report 212-976-4141
$
```

此外, 可以使用-y选项, 使grep(也使411)不区分大小写:

```
$ grep -y 模式 ...
```

通过使用-y选项, 模式中的小写字母也可以和输入中的大写字母匹配。(这个选项在第7版的grep中出现, 但在某些其他版本中没有。)

命令参数还有一些精彩的用法, 将在第5章中介绍, 但这里还有一点需要注意。参数\$0是指正在执行的程序的名字——在cx中, \$0就是cx。\$0的新用法用在程序2、3、4等的实现中, 这些程序将它们的输出打印成多列:

```
$ who / 2
drh      tty0      Sep 28 21:23      cvw      tty5      Sep 28 21:09
dmr      tty6      Sep 28 22:10      scj      tty7      Sep 28 22:11
you      tty9      Sep 28 23:00      jlb      ttyb      Sep 28 19:58
$
```

程序2、3等的实现是类似的, 事实上它们被链接到同一个文件:

```
$ ln 2 3; ln 2 4; ln 2 5; ln 2 6
$ ls -li [1-9]
16722 -rwxrwxrwx 5 you      51 Sep 28 23:21 2
16722 -rwxrwxrwx 5 you      51 Sep 28 23:21 3
16722 -rwxrwxrwx 5 you      51 Sep 28 23:21 4
16722 -rwxrwxrwx 5 you      51 Sep 28 23:21 5
16722 -rwxrwxrwx 5 you      51 Sep 28 23:21 6
$ ls /usr/you/bin / 5
2          3          4          411        5
6          cx          lc          m          nu
what      where
$ cat 5
# 2, 3, ....: print in n columns
```

```
pr -$0 -t -l1 $*
$
```

-t选项表示取消打印页眉，-ln选项则设置页长为n行。程序的名字成了传递给pr的列数，因此输出将会被打印成\$0所指的列数。

3.5 程序输出作为参数

现在从 shell文件的命令参数转向参数的生成。当然通过使用元字符如*等扩展文件名，是产生命令参数的有效方式(不同于直接提供文件名的方法)，但是，还有另一种方法，就是运行一个程序。任何程序的输出都可以放入命令行，只需要将其用反引号`括住：

```
$ echo At the tone the time will be `date`
At the tone the time Will be Thu Sep 29 00:02:15 EDT 1983
$
```

对上面的例子稍作改动就可以得到如下形式，它表明反引号中的内容可以在双引号里得到解释：

```
$ echo "At the tone
> the time will be `date`."
At the tone
the time will be Thu Sep 29 00:03:07 EDT 1983.
$
```

再举一个例子。假设想要发送邮件给一系列的人，他们的登录名放在文件mailinglist里。比较笨拙的方法是将 mailinglist编辑成一个合适的mail命令，并用shell运行之，但采用如下方案更容易：

```
$ mail `cat mailinglist` <letter
```

它运行cat产生一个用户名的列表，该列表成为mail命令的参数。(反引号中的程序输出解释成参数时，shell把换行符看作单词的分隔符。而不是命令行结束符。在第5章中再详细论述。)反引号使用起来十分方便，它使得不必再为mail命令设置一个单独的mailinglist选项。

另外一种将文件mailinglist从一个包含名字列表的文件，转换成打印该列表的命令的方法，如下：

```
$ cat mailinglist                                新版本
echo don whr ejs mb
$ cx mailinglist
$ mailinglist
don whr ejs mb
$
```

现在若要将信发送给列表中的人，只需

```
$ mail `mailinglist` <letter
```

利用另外一个程序 pick，甚至可以交互式修改用户列表：

```
$ pick 参数 ...
```

pick程序一次回送一个参数，并在每一个参数后等待响应。pick的输出是那些响应为 y 的参数，其他的则都被忽略。例如：

```
$ pr `pick *.c` / lpr
```

回送每一个以c结尾的文件名，那些被选中的文件则用pr和lpr打印出来。(pick并不是第7版的内容，但它方便易用，本书的第5章和第6章中会再次提及它)。

假如采用了mailinglist的第二种实现，那么：

```
$ mail 'pick \'mailinglist\'' <letter
don? y
whr?
ejs?
mb? y
$
```

将信送到don和mb。注意这里有嵌套的反引号。而反斜杠使得shell在分析时，对内层的`...`不加解释。

练习3-10 如果在下列句子中，反斜杠被省略：

```
$ echo `echo\`date\``
```

将会得到什么结果？

练习3-11 在机器上尝试

```
$ `date`
```

练习3-12

```
$ grep -l 模式 文件名
```

列出了与模式相匹配的文件名，而没有任何其他结果。如果改成如下形式

```
$ command `grep -l 模式 文件名`
```

将会如何？

3.6 shell变量

和大多数编程语言一样，shell程序也有变量，有时也称作参量。字符串\$1就是位置参量，它保存了传给一个shell文件的参数。而数字1则指明了它在命令中的位置。我们已经介绍过其他一些shell变量：PATH保存了shell查找命令的目录列表，HOME是你的登录路径，等等。与常规语言中的变量不同的是，参数变量的值是不能修改的。尽管PATH是个变量，其值为\$PATH，但不能说变量1的值为\$1。\$1只是对应于命令的第一个参数的缩写符而已。

除了位置参量外，可以创建、访问和修改shell变量。例如，

```
$ PATH=:/bin:/usr/bin
```

对PATH赋值，改变了查找路径。在等号 = 左右不能有空格，而赋给的值也必须是一个单字，也就是说，若它含有不应该被解释的shell元字符，就必须用引号括起来。在变量名前加一个\$符号，可得到变量的值：

```
$ PATH=$PATH:/usr/games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games
$ PATH=:/usr/you/bin:/bin:/usr/bin          恢复原值
$
```

不是所有的shell变量都有特定的含义。你可以创建新的变量，给它们赋值。通常有特殊意义的变量都用大写，因此普通变量名就用小写。变量常用于存储长的字符串，例如路径名：

```
$ pwd
/usr/you/bin
$ dir=`pwd`          记下目前所处的位置
$ cd /usr/mary/bin   转到其他地方
$ ln $dir/cx .       在文件名中使用变量
$ ...               做其他工作
$ cd $dir            返回
$ pwd
```

```
/usr/you/bin
$
```

shell的内部命令set可以显示出你定义的所有变量的值。如果只想查看一两个变量，使用echo就足够了。

```
$ set
HOME=/usr/you
IFS=

PATH=/usr/you/bin:/bin:/usr/bin
PS1=$
PS2=>
dir=/usr/you/bin
$ echo $dir
/usr/you/bin
$
```

一个变量的值与创建它的shell有关，而且该值并不会自动地传递给子shell：

```
$ x=Hello          创建x
$ sh              新的shell
$ echo $x         只有换行符：x在子shell里没定义

$ ctrl-d         离开子shell
$                回到原来的shell
$ echo $x
Hello            x仍然有定义
$
```

这意味着一个shell文件不能修改变量的值，因为shell文件是由子shell运行的。

```
$ echo 'x="Good Bye"'  创建一个两行的shell文件...
> echo $x' >setx      ...来设置和打印x
$ cat setx
x="Good Bye"
echo $x
$ echo $x
Hello              在原来的shell中，x的值是hello
$ sh setx
Good Bye          在子shell中x是Good Bye
$ echo $x
Hello            但在原shell中，x仍是hello
$
```

但是，有时通过shell文件来改变shell变量的值是很有用的。一个典型的例子就是使用一个文件，将新的目录加到用户的PATH变量中去。为此shell提供了一个.命令，使得可以在当前shell中执行某shell文件，而不是在子shell里。这个命令的出现，一开始是为了使用户能方便地执行他们的.profile文件，而无需重新登录，但此外它还有别的用途：

```
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games      将/usr/games附加到PATH里
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin
$ . games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games
$
```

.命令所在的文件也通过PATH来搜索，故它可以放入用户的bin目录中去。

当一个文件用.执行时，表面看来像是执行一个shell文件，但实际上并非如此。相反，该文件中的命令被如此解释，就好像用户在终端上交互式键入它们一样，即shell的标准输入被暂时重定向到了该文件。既然文件只是被读取而不是被执行，它甚至不需要有可执行权限。另外一个区别在于这样的文件不接受命令行参数，而且\$1、\$2以及其他位置参量都为空。假

如这类命令能够接受参数，就更好了，但目前还做不到这一点。

在子shell里设置变量值的另一个方法，是在一个命令行的命令之前显式地给它赋值：

```
$ echo 'echo $x' >echox
$ cx echox
$ echo $x
Hello                                和以前一样
$ echox                               x不在子shell里

$ x=Hi echox
Hi                                    x被传给了子shell
$
```

(原来在命令行里的任意地方赋值都会被传递给该命令，不过这样与dd(1)命令不符合。)

命令用于永久地改变一个变量的值，而行内赋值适合于临时地改变。作为一个例子，再次考虑一下搜寻/usr/games的问题，该目录并不在用户的PATH中：

```
$ ls /usr/games | grep fort
fortune
$ fortune                               Fortune cookie命令
fortune: not found
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin           /usr/games不在PATH中
$ PATH=/usr/games fortune
Ring the bell; close the book; quench the candle.
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin           PATH没有被改变
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games                 games命令仍然存在
$ . games
$ fortune
Premature optimization is the root of all evil - Knuth
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games 这次PATH被改变了
$
```

可以在同一个文件里用这两种不同的机制来改变变量的值。稍微改动一下games命令，就可以不用改变PATH的值而运行一个游戏，或是可以永久地改变PATH，以使其包含/usr/games：

```
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games $*              注意$*
$ cx /usr/you/bin/games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin           没有/usr/games目录
$ games fortune
I'd give my right arm to be ambidextrous.
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin           仍然没有/usr/games目录
$ . games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games 现在有了/usr/games目录
$ fortune
He who hesitates is sometimes saved.
$
```

对games的第一次调用是在子shell里运行一个shell文件，这里PATH只是暂时被改变以包含/usr/games。相反，第二个例子是在当前shell里解释这个文件，\$1是空串，因此在命令行中没有命令，且PATH被永久地改变了。games的这两种使用方法都比较巧妙，而且也很方便自然。

当希望可在子shell中访问某变量值时，可以使用shell的export命令。(你可能会想，为什么不能将一个变量值从某个子shell传送给它的父shell。)看一下以前的一个例子，只是现在将

变量输出了:

```
$ x=Hello
$ export x
$ sh
$ echo $x
Hello
$ x='Good Bye'
$ echo $x
Good Bye
$ ctrl-d
$
$ echo $x
Hello
$
```

新的shell
x在子shell中有定义
改变它的值
离开子shell
回到原来的shell
x仍是hello

export有着微妙的语义，但在日常应用中只要能遵循以下规则即可：不要贪图一时方便而输出临时变量，而应该输出那些希望能出现在所有shell和子shell中的变量(如用ed的！命令开头的shell)。因此，与shell有关的变量，如PATH和HOME，都应该被输出。

练习3-13 为什么总要在PATH中包括当前目录？它应该被置于何处？

3.7 进一步讨论I/O重定向

标准错误输出总是出现在终端上:

```
$ diff file1 file2 >diff.out
diff: file2: No such file or directory
$
```

当然，错误信息应该这么输出，否则，如果它输出到diff.out，你就会以为错误的diff命令得到了正确的执行。

缺省状态下，每个程序启动时总要建立3个文件，分别用名为“文件描述符”的小整数标识(在第7章再详细讨论文件描述符)。标准输入和标准输出分别用0和1表示，它们经常被重定向到文件和管道。而2表示标准错误输出，它通常都被送往终端。

有时即使程序正确运行也会产生标准错误输出。一个典型的例子是程序time，它运行一条命令，然后在标准错误输出上报告使用的时间。

```
$ time wc ch3.1
  931   4288   22691 ch3.1

real    1.0
user    0.4
sys     0.4
$ time wc ch3.1 >wc.out

real    2.0
user    0.4
sys     0.3
$ time wc ch3.1 >wc.out 2>time.out
$ cat time.out

real    1.0
user    0.4
sys     0.3
$
```

结构2>filename(在2和>之间没有空格)将标准错误输出定向到该文件。从语法上，它看起来比较怪异，但这个结构大有用处。(time产生的时间对如此短的例子而言，也许不太精确，但如果换成一系列较长的例子，这些数字就会相当准确可靠，可以保存起来以便以后分析。例如，

见表8-1。)

也可以将两个程序的输出流合并:

```
$ time wc ch3.1 >wc.out 2>&1
$ cat wc.out
  931   4288   22691 ch3.1

real      1.0
user      0.4
sys       0.3
$
```

符号2>&1让shell将标准错误输出合并到标准输出流中。这里&仅是一个记号。还可以通过1>&2把标准输出加入到标准错误输出中:

```
echo ... 1>&2
```

在错误输出上打印结果。在shell文件里, 这样可以将信息送往终端, 而不是被偶然地送到一个管道或文件。

shell提供了把命令的标准输入和命令放在一起的功能, 而不必分开存放, 因此shell文件本身即是齐全的。程序411的目录信息可写作:

```
$ cat 411
grep "$*" <<End
dial-a-joke 212-976-3838
dial-a-prayer 212-246-4200
dial santa 212-976-3636
dow jones report 212-976-4141
End
$
```

shell称这种结构为here document, 它表明文件的输入就是该文件的一部分, 而不是别的文件。<<指示了这种结果, 后面紧接着的单字(在本例中是End)用于终止输入, 即信息从文件输入, 直到出现该终止词为止。shell要替换here文件里的\$, `...`和\, 除非该单字用引号或反斜杠括起来了。在后面这种情况下, 整个文件被看作文本文件。

在本章结束时, 还会讨论here document, 那时可以看到一些更有趣的例子。

练习3-14 试将411的here document版本与原始文件作比较, 哪一个更容易维护? 对一般服务性而言哪一个基础更好?

表3-2列出了shell支持的各种I/O重定向。

表3-2 shell I/O重定向

形 式	含 义
>文件	将标准输出定向到文件
>>文件	将标准输出附加到文件
<文件	从文件file中取得标准输入
p ₁ p ₂	将程序p ₁ 的输出连接到p ₂ 的输入
^	与 的意义相同
n>文件	将输出从文件描述符n定向到文件
n>>文件	将输出从文件描述符n附加到文件
n>&m	将文件描述符n和m的输出合并
n<&m	将文件描述符n和m的输入合并
<<s	here document, 作为标准输入, 直到首次出现s字符串为止; 替换文件中的\$, `...`、和\
<<\s	不进行替换的here document
<<'s'	不进行替换的here document

3.8 shell程序里的循环

shell实际上是一种编程语言，它有变量、循环、条件转移等等。这里只讨论基本的循环，在第5章将进一步讨论控制流。

shell的循环通常是对一组文件名进行的，shell的for语句是唯一可能在终端上键入，而不是放入文件以备以后执行的控制流语句。语法是：

```
for 变量 in 文件列表
do
    命令
done
```

例如，可以用一个for语句，在一行输出一个文件名，如下：

```
$ for i in *
> do
>     echo $i
> done
```

i 可以是任意shell变量，但习惯上用i表示。注意该变量的值用\$i表示，但for循环中就用i引用该变量。用*取出当前目录中的所有文件，但也可使用其他的参数。通常我们并不仅仅是想打印出文件名，而往往要对一组文件的新旧版本进行比较。例如，第2章中，为了比较目录old中的旧版本和当前版本，可进行如下操作：

```
$ ls ch2.* | 5
ch2.1      ch2.2      ch2.3      ch2.4      ch2.5
ch2.6      ch2.7
$ for i in ch2.*
> do
>     echo $i:
>     diff -b old/$i $i
>     echo
> done | pr -h "diff `pwd`/old `pwd`" | lpr &
3712
$
```

为增强可读性，加入一个空行

进程标识符

通过管道将输出放入pr和lpr只是为了说明，有可能把for语句中程序的标准输出放在for循环本身的输出上。在pr命令里放了一个-h选项，打印输出标题，并用了两个嵌入的pwd命令。程序的整个运行序列设计成异步的，因此用户不必等待它结束，&号作用于整个循环和管道线。

按上述格式书写for语句是比较合适的，但你也可以写得紧凑一些。不过必须注意，关键字do和done只有紧接在一个换行符或分号后才能被识别。根据for语句的长短，有时可以把它写成一行：

```
for i in 文件列表; do 命令; done
```

for循环一般用于多命令的情况，或者单个命令不适宜处理内部参数的时候。但如果单独的命令已经能够遍历文件名，就不必使用for循环：

```
# Poor idea:
for i in $*
do
    chmod +x $i
done
```

就不如

```
chmod +x $*
```

因为for循环为每一个文件都执行一次chmod，这极大地浪费了计算机资源。（首先应该弄清楚

```
for i in *
```

和

```
for i in $*
```

的区别,前者对当前目录中的所有文件名进行循环,而后者则是对该shell文件的所有参数进行循环)。

一个for语句的参数列表通常来自于与文件名匹配的模式,但也可以有其他来源。它可以是

```
$ for i in `cat...`
```

或者是键入的参数。例如,在本章的前面,有一组多列打印的程序,名为2、3等。一旦写出了文件2:

```
$ for i in 3 4 5 6; do ln 2 $i; done
$
```

它们就被链接到了同一个文件。

而for语句的一个更为有趣的应用是,可以使用pick来选择文件,并同备份目录的相应文件进行比较:

```
$ for i in `pick ch2.*`
> do
>     echo $i:
>     diff old/$i $i
> done | pr | lpr
ch2.1? y
ch2.2?
ch2.3?
ch2.4? y
ch2.5? y
ch2.6?
ch2.7?
$
```

很显然,这个循环应该置于一个shell文件中,以节省键入时间,否则若想执行两遍,就不得不键入两次。

练习3-15 若diff循环被放入一个shell文件中, pick也将放入文件吗?为什么?

练习3-16 如果上述循环的最后一行改为

```
> done / pr / lpr &
```

结果如何?

3.9 bundle合并

为了给出开发shell程序的更有趣的应用,现在来具体实现一个较大的例子。假设你收到一个朋友somewhere!bob[⊖]从另外一台机器上发来的信件,他想复制你的bin目录中的文件。最简单的方式是通过信件回送文件,如下:

```
$ cd /usr/you/bin
$ for i in `pick *`
> do
>     echo ===== This is file $i =====
>     cat $i
> done | mail somewhere!bob
$
```

⊖ 有很多种对远端机器地址的表示法,机器名!人名的形式最为常见。参见mail(1)。

但从somewhere!bob的角度看来，他得到的邮件尽管已将各文件区分清楚，他仍需要用编辑程序将它们拆成一个个单独的文件。最好能如此组织信件，使得在接收端信件信息能自动还原成原来的文件，而不必要接收者再做工作。这就意味着，实现这个功能的shell文件应包括所有要传送的文件和拆分还原的指令。

第二个想法是，可以用shell的here document来很方便地将命令和命令所需数据结合起来。余下的工作只是正确使用引号。下面是程序bundle，它在标准输出上把文件组织到一个可自我解释的shell文件中：

```
$ cat bundle
# bundle:  group files into distribution package

echo '# To unbundle, sh this file'
for i
do
    echo "echo $i 1>&2"
    echo "cat >$i <<'End of $i'"
    cat $i
    echo "End of $i"
done
$
```

End of \$ i用引号括住保证了文件中的任何元字符都将被忽略。

当然，在送给somewhere!bob之前，应该先测试一下该程序：

```
$ bundle cx lc >junk          做一个合并
$ cat junk
# To unbundle, sh this file
echo cx 1>&2
cat >cx <<'End of cx'
chmod +x $*
End of cx
echo lc 1>&2
cat >lc <<'End of lc'
# lc: count number of lines in files
wc -l $*
End of lc
$ mkdir test
$ cd test
$ sh ../junk                  测试
cx
lc
$ ls
cx
lc
$ cat cx
chmod +x $*
$ cat lc
# lc: count number of lines in files
wc -l $*                      看起来不错
$ cd ..
$ rm junk test/*; rmdir test   清除测试文件
$ pwd
/usr/you/bin
$ bundle 'pick *' | mail somewhere!bob  发送文件
```

如果发送的一个文件的某行恰好具有如下形式，就会产生问题：

End of 文件名

但这是一种很少出现的情况。要使bundle程序更安全可靠，需要用到后面章节所讲述的内容。

bundle显示了UNIX环境的灵活性，它使用了shell循环、I/O重定向、here document以及

shell文件，它直接为mail提供了接口，甚至它就是一个能创建程序的程序。bundle是个设计得很精巧的程序，仅用几行代码就完成了一系列文件操作。

练习3-17 应该怎样利用bundle传送一个目录里的所有子目录和文件？提示：shell文件可以是递归的。

练习3-18 试修改bundle，使之包括ls -l命令列出的文件的各种信息，特别是访问权限和最近修改日期。并将bundle与文档管理程序ar(1)做一比较。

3.10 为什么说shell是可编程的

UNIX的shell并不是一般的命令解释器：尽管它允许以通常的方式运行命令，但它还是一种编程语言，它还可以完成许多另外的功能。这里，有必要简单回顾一下本章所讲内容，因为本章的知识涉及面广，而且强调了shell的“常用特性”，并给出了许多编程实例。大多数时候，你只需在终端上键入一行的简单命令，比如管道线命令也是一个程序，如前所述Tea is ready的例子。经常使用shell，就会发现shell编程十分容易、自然，以至于都意识不到这是在编程。

shell可以使用循环、I/O重定向，以及用*通配符扩展文件名，这些功能适用于所有程序。其他一些特性，如shell文件和管道线，是真正由内核提供的，但shell对它们给出了一个自然的文法结构。它们使用起来都非常方便，并增加了系统处理问题的能力。

shell大部分的功能和特点都来自UNIX系统内核。比如说，尽管由shell建立管道，但实际上是内核在移动数据。系统按这样的方式来处理可执行文件，即用户可以书写shell文件，它们运行起来就像编译过的二进制文件那样。用户根本不需要知道它们是命令文件，shell无需调用像run这样的特殊命令就可以运行它们。同样，shell本身也是一个程序，而不是内核的一部分，所以它可以修改，扩充，象其他程序一样使用。这种思想并不是UNIX独有的，但它在UNIX中得到了最完善的体现。

第5章，还将讲述shell编程，但应记住，无论在shell中做什么，实际上都是在进行编程。这也就是系统能工作得如此好的主要原因。

有关文献

一开始shell就是可编程的。最初有一些单独的命令，如if、goto和标号，goto命令从头扫描输入文件，以找到合适的标号。（因为重新读取一个管道是不可能的，所以将管道结果放入一个有某种控制流的shell文件也是不可能的。）

第7版的shell最初是由Steve Bourne写的，他借鉴了John Mashey的一些思想。在第5章将会看到，它包含了编程所需的所有特性。此外，输入输出也更合理：可以对shell程序进行不受限制的I/O重定向。文件名元字符的分析也成为shell的一个功能，它原来是由一个单独的程序来完成的。

你可能还会用到(或者已经使用了)csh，即Bill Joy在建立第6版shell时开发的C shell。在交互上，C shell比Bourne的shell更深入一步，它提供了一个历史记录机制，允许用户只用缩写就可以执行以前执行过的命令，语法上也有些不同。但因为它是基于较早的shell版本开发的，它在编程方面不如第7版的shell方便，更像是一个交互的命令解释器，而不是一种编程语言。尤其是，它不允许控制流结构出入管道。

pick是Tom Duff开发的，而bundle则是由Alan Hewett和James Gosling独立完成的。

第4章 过滤程序

在UNIX系统程序中，有这样一类程序，它们读取输入数据，对其进行简单的变换，然后将其结果写到输出，如：grep和tail对输入数据做必要的选择；sort对输入数据进行排序；wc对其进行计数等等，这样的程序统称为过滤程序。

本章讨论一些最常用的过滤程序。我们从最常用的grep开始，说明模式匹配问题，与第1章相比，它要复杂一些。同时，我们还将介绍grep类中的另两个程序：egrep和fgrep。

第2节简要地介绍其他几个有用的过滤程序，包括：用于字符翻译的tr、处理其他系统中数据的dd以及检索重复文本行的uniq，同时，还将比第1章更为详细地介绍sort。

本章的其余部分集中讨论两个常用的程序：数据转换或可编程的过滤程序。之所以称之为可编程，是因为这些特殊的转换是以用简单的程序设计语言编写的程序来表述的。不同的程序可产生不同的转换。

这两个程序是：sed(stream editor)，流编辑程序；awk，这是以其作者的名字命名的。二者都是由grep派生而成。

```
$ program 模式行为 文件名
```

顺序扫描文件，查找与模式相匹配的行，找到时，则执行相应的操作。对于grep，其模式是如ed中那样的正则表达式，缺省的操作是打印与模式相匹配的每一行。

sed和awk对模式匹配的范围和实现相应的操作都进行了扩展。sed由ed派生而来，它从流经其上的文件中获得由编辑程序命令和流数据组成的“程序”，执行每一行上的程序指令。对于文本替换，awk不如sed方便，但它包含数学表达式、变量、内部函数以及类似于C语言的设计语言。本章未对这两种程序做全面的介绍，《UNIX程序员手册》2B卷中有关于这二者的基础教程。

4.1 grep系列

第1章中，我们简要地提及了grep，并曾在一些例子中使用到。

```
$ grep 模式 文件名...
```

搜索所指定的文件或标准输入，并打印与模式相匹配的每一行。grep在查找程序中的变量、文档中的单字或者选择一个程序的部分输出等方面，具有特殊功效。

\$ grep -n variable *.ch]	在C源程序中定位变量
\$ grep From \$MALL	打印信箱中的信箱报头
\$ grep From \$MALL / grep -v mary	不是来自于mary的信箱报头
\$ grep -y mary \$HOME/lib/phone-book	查找mary的电话号码
\$ who / grep mary	查看mary是否登录
\$ ls / grep -v temp	不包含temp的文件名

选项-n表示打印行号，-v表示对测试结果求反，-y使得模式中的小写字母与文件中的大写或小写字母都相匹配(大写字母仍与大写字母相匹配)。

迄今为止，在所有例子中，grep都是查找一般的字母或数字串，但实际上，grep可以搜索更为复杂的模式：grep用描述字符串的简单语言来解释表达式。

从技术上讲，模式是按正则表达式写的字符串，grep对正则表达式的解释与ed一样。事实上，grep最初是通过直接在ed上进行外部处理而形成的。

正则表达式是通过赋予某些字符特殊意义而确定的，如shell所用的*等。遗憾的是，还有几个元字符在意义上是有所差别的。表4-1列出了所有的正则表达式，这里仅做简要的介绍。

元字符^和\$分别表示行首和行尾。如：

```
$ grep From $MAIL
```

搜索信箱中包含From的行。而

```
$ grep '^From' $MAIL
```

则打印起始于From的行，这更有可能是邮件的报头行。正则表达式元字符与shell元字符有重叠的部分，因此，最好用单引号将grep模式括起。

grep支持类似于shell中的字符组，因此，[a-z]匹配任何的小写字母。但也存在差别。若grep字符组起始于符号^，则模式匹配除了组以外的任何其他字符，因此，[^0-9]匹配任何非数字。同样，在shell中，反斜杠保护字符组中的]和-，但是，grep和ed却要求在其意义明确时，使用这些字符。如，[]与左或右中括号或一个负号相匹配。

点号.相当于shell的?，它可以匹配任何字符。(点号恐怕是不同UNIX程序中意义最为不同的字符。)下面是两个例子：

```
$ ls -l / grep '^d'
```

列出子目录名

```
$ ls -l / grep '^.....rw'
```

列出其他用户可以读写的文件名

符号^和7个点号匹配行中的任何头7个字符，当这种表示应用到ls-l的输出时，其意义是任意许可字符串。

延续操作符*与表达式中前面的字符或元字符(包括一个字符组)有关，它们共同匹配任意多个连续重复的前一字符或元字符。如：x*匹配尽可能长的x序列；[a-zA-Z]*匹配一字母表字符串；.*匹配任意字符直至一回车；.*x匹配行中到x为止并包括x的字符串。

关于延续操作符，有两点重要说明。首先，延续操作符仅适用于前一个字符，因此，xy*匹配的是x开始的任意个y的字符串，而不是像xyxyxy这样的序列；其次，“任意多个”包含零个，因此，若想要匹配至少一个字符，必须将其重复。如：为匹配一字母字符串，正确的表达式是[a-zA-Z][a-zA-Z]* (一字母接着零个或多个字母)。shell的*在做文件名匹配时，与任何文件名匹配，这类似于grep的正则表达式.*。

grep的正则表达式不能与换行匹配，表达式被独立地应用于每一行。

有了正则表达式，grep成为一种简单的程序设计语言，如：查看一下password文件，其第二个域是加密口令，以下命令搜索没有口令的用户：

```
$ grep '^[^:]*::' /etc/passwd
```

此模式表示：行的开始，任何数量的非冒号字符，最后是两个冒号。

grep实际上是程序族中最早的一个，这一族程序还包括fgrep和egrep，其基本特性是一样的，但fgrep可同时搜索许多文字字符串，而egrep解释真正的正则表达式——这与grep相同，不同的是egrep可用or操作符和圆括号来聚合表达式，这一点将在下面说明。

fgrep和egrep都用-f选项表示从指定的文件中读取模式。在此文件中，用换行分隔了可并

行搜索的模式。如，若有习惯性误拼的单词，则可以通过将其保存在一文件中，使用fgrep一次一行地来核对文档中这些单词的出现。

```
$ fgrep -f common-errors document
```

由egrep解释的正则表达式(亦列于表4-1中)与grep中的相同，只是另外又附加了两个：圆括号可被用于聚合，从而(xy)*匹配空字符串、xy、xyxy、xyxyxy等字符串中的任何一个；符号|为“或”操作符，today|tomorrow同于(today|tomorrow)，匹配today或者tomorrow。最后，egrep中还有两个延续操作符：+和?。模式x+匹配一个或多个x，x?仅匹配零个或一个x。

egrep很适合于这样的字游戏：从字典中查找具有特殊性质的单词。我们的字典是Webster's Second International，它被联机存储作为单词表，一行一个，没有解释。你的系统可能装有/usr/dict/words，这是一个较小的字典，目的在于核对拼写，或查看一下以了解其格式。以下模式用来查找包含所有五个元音，且按字母顺序排列的单词：

```
$ cat alphvowels
^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*$
$ egrep -f alphvowels /usr/dict/web2 / 3
abstemious          abstemiously        abstentious
acheilous            acheirous            acleistous
affectious          annelidous           arsenious
arterious            bacterious           caesious
facetious            facetiously          fracedinous
majestious
$
```

在文件alphvowels中，模式并未被封装在引号内。当用引号括起egrep模式时，shell保证命令不被解释，只是将引号去掉，egrep是看不到引号的。然而，由于文件并不是由shell检查，因此，不需要引号来括起模式的内容。对于这个例子，我们也可以用grep命令，但是，由于egrep的工作方式，当搜索包含终结符的模式，尤其是扫描大文件时，它还是要快得多。

另一个例子是查找有六个或更多个字母，且按字母表顺序排列的所有的单词：

```
$ cat monotonic
^a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?$
$ egrep -f monotonic /usr/dict/web2 /grep '.....' / 5
abdest      acknow      adipsy      agnosy      almost
befist      behint      beknow      bijoux      biopsy
chintz      dehors      dehort      deinos      dimpsy
egilops     ghosty
$
```

(egilops是侵袭小麦的一种疾病。)注意使用grep来过滤egrep的输出的用法。

为什么要有三个grep程序呢？因为fgrep不能解释元字符，但却可有效地并行检索成千上万的单词(一经初始化，其运行时间与单词数无关)，因此，主要用于类似于文件检索这样的任务。fgrep的典型模式在算术运算方面超出了grep和egrep的功能。grep和egrep难以明确区分，grep出现的稍早些，使用ed中所熟悉的正则表达式，并且具有带括号的正则表达式和很多选项。egrep可解释更一般的表达式(除了带括号的)，并且运行速度极快(其速度与模式无关)，但是，其标准版在表达式较为复杂时，需要较长的起动时间，现已有起动迅速的新的版本，因此，egrep和grep可组合成单个模式匹配程序。参见表4-1。

表4-1 grep与egrep正则表达式(按优先级的降序排列)

表 达 式	含 义
c	任何与其自己匹配的非特殊字符c
\c	去掉字符c的任何特殊意义
^	行首
\$	行尾
.	任意单个字符
[...]	...字符中的任一字符; 像a-z是合法的
[^...]	不在...中的任意单个字符; 范围是合法的
\n	与第n个\(...\)所匹配的字符(仅于grep中)
r*	r出现零或多次
r+	r出现一或多次(仅于egrep中)
r?	r出现零或一次(仅于egrep中)
r1r2	r2紧随r1
r1 r2	r1或r2(仅于egrep中)
\(r\)	带括号的正则表达式r(仅于grep中); 可嵌套
(r)	正则表达式r(仅于egrep中); 可嵌套

注: 没有正则表达式与换行匹配。

练习4-1 在附录A或ed(1)中查找带括号的正则表达式(\(和\)), 并用grep搜索回文——即顺读逆读拼写都一样的单词。提示: 对每种长度的单词写一个不同的模式。

练习4-2 grep的结构是读一行, 核对一个匹配, 然后循环。若正则表达式可匹配换行, grep会受到怎样的影响?

4.2 其他过滤程序

本节的目的在于提示大家, 系统确实提供了丰富的小型过滤程序, 并给出了许多应用实例。这里无法罗列出其所有内容——第7版还有更多, 而且在每次安装过程中都会创建一些属于自己的东西。所有的标准程序在手册的第一节中都有所介绍。

我们从sort开始, 它可能是所有过滤程序中最有用的一个。第1章中, 涉及了sort的基本内容: 它按照ASCII字母顺序逐行对其输入进行排序。虽然缺省时是这样, 但是人们可能还需要许多其他的数据排序方法。sort通过提供较多的选项来满足这些需要, 如: -f选项使得大、小写被“封装”, 从而不区分大、小写; -d选项(字典顺序)在比较时, 忽略除了字母、数字和空格外的所有字符。

虽然按字母顺序的比较是最常用的, 有时, 也需要对数字进行比较。-n选项表示按数值排序, -r选项对任意比较的结果求反, 因此,

```
$ ls / sort -f          按字母表的顺序将文件名排序
$ ls -s / sort -n       最小文件优先的排序
$ ls -s / sort -nr      最大文件优先的排序
```

sort通常对整行排序, 但也可使其注意力仅指向特定的域。符号+m的意义是比较时跳过头m个域; +0是行的开始。因此, 如:

```
$ ls -l / sort +3nr     按字节计数最大优先的排序
$ who / sort +4n       按登录时间最早优先的排序
```

其他有用的选项包括: -o, 用于为输出指定一文件名(可以是输入文件中的一个); -u, 删

除排序区域中的重复行，对相同的行仅保留一行。

可使用多个排序码，手册上sort(1)中关于加密的例子就说明了这种情况：

```
$ sort +0f +0 -u 文件名
```

+0f对行排序，将大、小写封装，但是相同的行可能并不相邻，因此，+0是第二个排序码，它对第一次排序得到的相同的行再按ASCII顺序排序，最后，-u删除任何相邻的重复行。因此，给出一列单词，每行一个，此命令打印出唯一的单词。某些图书的索引即是用类似的sort命令，使用了甚至更多的sort的功能精制而成的。参阅sort(1)。

命令uniq受sort的-u标志的启发：它删除每组相邻的重复行，对相同行仅保留一行。使用独立的程序实现此功能可以完成与排序无关的任务，如：uniq可以删除多个空行，无论其输入是否排序。uniq提供了多个选项专门来处理重复的行：uniq-d仅打印那些重复的行；uniq-u仅打印那些唯一的行(即不重复的行)；uniq-c求每行出现的次数。我们将简略地看一个例子。

comm命令是文件比较程序，给定两个已排序的文件f1和f2，comm打印三列输出：仅在f1中出现的行、仅在f2中出现的行以及在两个文件中出现的行。这些列中的任何一个都可以通过选项控制。

```
$ comm -12 f1 f2
```

仅打印在两个文件中都出现的行，而

```
$ comm -23 f1 f2
```

仅打印在第一个文件中出现，且不在第二个文件中出现的行。这在比较目录以及将单词表与字典相比较时很有用。

tr命令对其输入字符进行翻译，其最常用的用途是大、小写转换：

```
$ tr a-z A-Z
```

将小写字母转换成大写字母

```
$ tr A-Z a-z
```

将大写字母转换成小写字母

dd命令与我们已看到的其他所有命令都不相同，其主要目的是处理来自其他系统的磁带数据，它的很多名称均沿自OS/360作业控制语言。dd可作大、小写转换(其语法完全不同于tr)；它可将ASCII转换为EBCDIC或相反；它可读或写带有空填充的具有非UNIX系统特性的固定大小记录中的数据。实际上，dd常被用于处理原始的、未格式化的数据，无论是什么数据源。它包括一系列处理二进制数据的程序。

为了说明组合的过滤程序所能实现的功能，考虑下面的管道，它打印输入中最常用的10个单词：

```
cat $* |
tr -sc A-Za-z '\012' |      非字母字符转换为换行
sort |
uniq -c |
sort -n |
tail |
5
```

由于tr仅读取标准输入，所以要用cat收集文件。tr命令来自于手册，它将相邻的非字母字符转换为换行，因此，将输入转换为一行一个单词，然后，将单词排序，uniq -c命令将每组相同的单词删成为一行，并以其计数为前缀，这将成为sort -n的排序域，(这种在uniq的前后各使用一个sort的结合经常出现，足以成为固定用法。)结果是文档中的不同单词按其出现频次升

序排列。Tail选取10个最常用的单词(排序表中的最后), 5是按5列将其打印出来。

附带提一下, 以|结束一行是使其继续的有效方法。

练习4-3 用本节的工具写一个简单的拼写检查程序, 可利用/usr/dict/words。其缺点是什么? 如何解决?

练习4-4 用你喜欢的程序设计语言写一个单词计数程序, 比较它与上述单词计数管道的大小、速度和可维护性。你怎样才能轻易地将其转换为拼写检查程序?

4.3 流编辑程序sed

现在让我们转到sed。由于它是由ed直接派生而来, 应该是很容易掌握的, 并且还可由此进一步巩固你的ed知识。

sed的基本思想很简单:

```
$ sed '一系列ed命令' 文件名...
```

从输入文件中一次一行地读取, 按顺序将列表中的命令应用到每一行, 并将其编辑过的行写到标准输出。例如, 可用以下命令在一系列文件中任意出现UNIX的地方将其转换为UNIX(TM):

```
$ sed 's/UNIX/UNIX (TM )/g' 文件名... >output
```

不要误解, sed并不改变其输入文件的内容, 而仅把修改内容写到标准输出上, 因此, 原始文件并没有改变。至此, 你已有足够的shell经验, 可以意识到以下命令:

```
$ sed '....' file >file
```

是一个错误的命令: 为替换文件的内容, 一定要使用临时文件, 或另一程序。(后面, 我们会讨论一程序, 它包括了覆写现有文件的思想, 见第5章5.5节)。

sed自动输出每一行, 因此, 在上面的替换命令的后面不需要p; 若是加上, 那么每个被修改的行会被打印两次。然而, 引号总是必须的, 因为sed的很多元字符也是shell的元字符。如: 考虑使用du -a来生成文件列表, 通常du打印文件的大小和文件名:

```
$ du -a ch4.*
18      ch4.1
13      ch4.2
14      ch4.3
17      ch4.4
2       ch4.9
$
```

可使用sed删除文件大小, 但编辑命令需要用引号来保护*和Tab键不为shell所注释:

```
$ du -a ch4.* / sed 's / .*→//'
ch4.1
ch4.2
ch4.3
ch4.4
ch4.9
$
```

此替换删除所有的字符.* 直至并包含最右边的Tab键(模式中用→表示)。

类似地, 可从who的输出中选择用户名和注册时间:

```
$ who
lr      tty1      Sep  29   07:14
ron     tty3      Sep  29   10:31
you     tty4      Sep  29   08:36
td      tty5      Sep  29   08:47
$ who | sed 's/ .*//'
lr 07:14
ron 10:31
you 08:36
td 08:47
$
```

s命令以单个空格置换一个空格和紧随其后的直至另一个空格的一切字符(尽可能的多, 包括更多的空格), 同样, 此命令也需要引号。

可以使用几乎与此相同的sed命令生成一个程序getname, 用于返回你的用户名:

```
$ cat getname
who am i | sed 's/ .*//'
$ getname
you
$
```

另一个sed序列, 由于经常使用, 我们将其作成是一个shell文件, 称为ind。ind命令将其输入数据缩进一个Tab键, 这可用于方便地移动某些行以更适合行式打印机的打印格式。

ind的实现比较简单——在每行的前面敲一下Tab键:

```
sed 's/^/→/' $*                                ind第一版
```

此版本甚至在每个空行中也输入一个Tab, 这是不必要的。较新的版本是利用sed的功能来选取要修改的行, 若事先为命令指定了一个模式, 则只有与模式相匹配的行才会受到影响:

```
sed '/./s/^/→/' $*                               ind第二版
```

模式/./与任意至少有一个字符(不包括换行)的行相匹配。s命令作用于这样的行, 而非空行。注意sed输出所有的行, 无论它们是否被修改, 因此, 空行仍保留在输出部分。

还可有另一种方法写ind, 通过在命令前加感叹号, 使命令仅作用在与所选取的模式不匹配的行上。在以下命令中:

```
sed '/^$/!s/^/→/' $*                             ind第三版
```

模式/^\$/与空行相匹配(行尾紧跟着行首), 因此, /^\$/!的意义是: 不要将命令作用在空行上。

如上所述, sed自动打印每行, 无论对其施加了怎样的操作(除非是删除操作), 进而, 也可使用大多数的ed命令。所以, 很容易写一段sed程序, 打印出其输入的比如说前三行, 然后停止:

```
sed 3q
```

虽然3q不是一个合法的ed命令, 但在sed中却有意义: 拷贝行, 然后在第三行后退出。

你可能想对数据进行其他处理, 如对其进行缩进, 一种方法是将sed的输出数据送到ind, 然而, 由于sed可以处理多个命令, 因此, 调用一次sed即可实现:

```
sed 's/^/→/
    3q'
```

注意引号与换行所在的位置，每条命令必须放在单独的行上，但sed忽略前导的空格和Tab键。

有了这些思想后就可以写一段程序head，来打印每个文件名字段的头几行，这似乎是很 有意义的。但是，使用sed 3q(或10q)命令的如此简单的实现，使人感觉不到它的必要性。然而，我们确实有一个ind命令，因为与其相对应的sed命令的输入很困难。(在写书的过程中，我们用前面所示的用一行实现的第二版本代替了现有的30行C程序)。没有明确的标准来确定什么时候值得用单个命令，而什么时候又用比较复杂的命令行。现在所发现的最好的规则是将其放到bin中，以查看实际上是否经常使用它。

也可以将sed命令置于一个文件中，并在那里执行：

```
$ sed -f 命令文件...
```

可使用行数而不是像3这样的数字选择程序：

```
$ sed '/ 模式/q'
```

打印其输入直至并包含与模式相匹配的第一行，并且

```
$ sed '/ 模式/d'
```

删除包含模式的每一行，删除发生在自动打印行之前，因此，删除的行被丢弃。

虽然自动打印通常是比较方便的，但有时也会变得碍事，可用-n选项将其关闭。在这种情况下，只有用p命令显示说明的行才出现在输出中，例如：

```
$ sed -n '/ 模式/p'
```

的作用同于grep，由于可后加一感叹号！使匹配条件变反，故

```
$ sed -n '/ 模式/!p'
```

等同于grep -v。(因此也等同于sed '/ 模式 /d'。)

为什么既要有sed，又要有grep呢，grep毕竟只是sed的一个简单的特例。部分原因是历史造成的——grep先于sed而出现，但是grep能生存下来，而且袭用至今，是因为对于二者都可完成的特定工作，使用grep较sed 要容易得多：它尽可能简洁地完成一个又一个的普通事例。(grep亦可实现一些sed所不能实现的功能，如-b选项。)然而，确实有些程序销声匿迹了，曾有个gres，可以做一些简单的替换，但当sed出现后，它就被淘汰了。

利用sed可插入新行，其语法要求与ed相同：

```
$ sed 's/$/\n'
> /'
```

在每行的末尾增加一回车，从而使其输入扩展为双倍行距，又如：

```
$ sed 's/[ →][ →]*\n'
> /g'
```

用换行替换每个空格或Tab串，从而将其输入分为一行一个单词。(正则表达式[→]匹配一个空格或Tab，[→]*匹配零或多个空格或Tab，因此整个模式匹配一个或多个空格和(或)Tab)。

也可使用成对的正则表达式或行号来选取一定范围内的行，使得任一命令都可作用于其上。

```
$ sed -n '20,30p'
$ sed '1,10d'
```

仅打印20至30行

删除1至10行(=tail+11)

```
$ sed '1,/^$/d'           删除直至并包括第一个空行
$ sed -n '/^$/,/^end/p'    打印每个从一空行至一始于end的这样的行组
$ sed '$d'                 删除最后一行
```

行号从输入的起始记起，在新文件的开始处并不重置。

然而，sed有个基本的限制，是ed所没有的：它不支持相对行数的计算。实际上，在行数表达式中，不能识别+和-，因此，不可能在输入中向反方向进行查找：

```
$ sed '$-1d'              不合法的：不能反方向查找
Unrecognized command: $-1d
$
```

一旦读了一行，前面的一行就永远过去了：没有办法识别前面一行，而这正是上面的命令所要求的。（本着公平，有一方法处理sed的这个限制，但相当高深，可在手册中查找hold命令。）同样，也没有办法用相对地址向前查找：

```
$ sed '/thing/+1d'         不合法的：不能向前查找
```

sed提供了写出多行输出文件的能力，如：

```
$ sed -n '/ 模式/w file1
>          / 模式/!w file2' 文件名...
$
```

将与模式相匹配的行写到file1上并将不与模式相匹配的行写到file2上。回到第一个例子：

```
$ sed 's/UNIX/UNIX(TM)/gw u.out' 文件名... >output
```

像以前一样，将整个输出写到文件output，但它还要将改变了的行写到文件u.out。

有时，有必要与shell相结合，在sed命令中间取得shell文件参数。一个例子是程序newer，列出目录中比指定的文件更新的所有文件：

```
$ cat newer
# newer f: list files newer than f
ls -t | sed '/^'$1'$/q'
$
```

引号保护sed里使用的各种特殊字符，然而将\$1留在外面，从而shell可用文件名替换它。另一种写参数的方法是：

```
"/^$1\$/q"
```

由于\$1会为参数所替换，而\ \$恰好变为\$。

同样，也可写older，它列出所有比指定的文件更早的文件：

```
$ cat older
# older f: list files older than f
ls -tr | sed '/^'$1'$/q'
$
```

唯一的区别在于ls上的-r选项，将顺序变反。

虽然sed的功能比已说明的要得多，包括测试条件、循环和分支、保留先前的行，当然还有许多附录A中介绍的ed命令，但是，sed的大多数用途类似于这里已说明的，由一个或两个简单的编辑命令，而不是较长或复杂的序列构成。表4-2总结了sed的一些功能，其中省略了处理多行命令的功能。

表4-2 sed命令汇总

命 令	含 义
a \	将行添加到输出直至不以\为终结的行
b 标号	转至命令: 标号
c \	对随后的文本, 如同a命令那样逐行修改
d	删除行, 读下一个输入行
i \	在下一个输出前插入下面的文本
l	按行列表, 使所有的非打印字符可见
p	打印行
q	退出
r 文件	读文件, 拷贝内容到输出
s /旧字符串/新字符串/f	用新字符串代替旧字符串, 若f=g, 置换所有的变化; f=p, 打印; f=w 文件, 写入文件
t 标号	测试: 若对当前行作了替换, 转至标号
w 文件	将行写到文件
y /串1/串2/	用串2中对应的字符替换串1中的每个字符(不支持范围值)
=	打印当前输入行号
! 命令	仅当行未被选取时, 执行sed 命令
: 标号	为b和t命令设置标号
{	将{ }之间的命令视作一组

因为sed可以处理任意长的输入, 而且运行速度较快, 还因为它的正则表达式与ed十分类似, 并且也是一次一行地处理, 所以sed使用起来很方便。然而, 另一方面, 由于sed仅提供相当有限的存储记忆形式(难以记住从一行到另一行的文本), 所以它仅能扫描数据一次, 不能往回扫描, 没有办法执行诸如/.../+1这样的向前关联, 并且没有提供处理数字的工具——它仅仅是一个纯粹的文本编辑程序。

练习4-5 修改older和newer, 使得它们在输出中不包含参数指定的文件。修改它们使得文件以逆序列表。

练习4-6 利用sed使得bundle更完备。提示: 在here document中, 仅当结束标志与行完全匹配时才有效。

4.4 模式扫描与处理语言awk

sed的一些局限性可由awk来弥补。awk的思想几乎与sed相同, 但其操作细节更多的是基于C程序设计语言, 而不是文本编辑程序, 用法也同于sed:

```
$ awk '程序' 文件名...
```

但是程序不同:

```
模式 { action }
模式 { action }
...
```

awk一次一行地读文件名中的输入, 依次将每行与每个模式相比较, 对每个与行相匹配的模式, 执行其对应的操作。和sed一样, awk并不改变其输入文件。

awk的模式可以是完全同于egrep中的正则表达式, 也可以是使人联想到C 的更复杂的条件。以下是一简单的例子:

```
$ awk '/正则表达式/{ print }' 文件名 ...
```

实现egrep所实现的功能：打印与正则表达式相匹配的每一行。

模式或操作都是可选的，若省略操作，缺省的操作是打印相匹配的行，因此，

```
$ awk '/正则表达式/' 文件名...
```

与前面的例子做同样的事情。相反，若省略模式，则操作部分被作用到任何输入行，因此，

```
$ awk '{ print }' 文件名 ...
```

的功能同于cat，尽管要慢得多。

在开始有意义的例子之前，最后需要注意的一点是，对于sed，有可能从文件将程序提交给awk：

```
$ awk -f 命令文件 文件名 ...
```

4.4.1 字段

awk自动将每个输入行分成字段，即由空格或Tab分隔的非空格字符串。由此定义，who的输出有五个字段：

```
$ who
you      tty2      Sep 29 11:53
jim      tty4      Sep 29 11:57
```

awk称这些字段为\$1、\$2、...、\$NF，其中NF为一变量，其值被设置为字段的个数。在此例中，两行的NF皆为5。（注意NF与\$NF两者之间的差别，NF指字段的个数，\$NF指行中的最后一个字段。不同于shell，在awk中只有字段是以\$开始，变量是不加此修饰的。）例如：抛弃由du -a产生的文件的大小：

```
$ du -a / awk '{ print $2 }'
```

下面的程序一行一个地打印登录人的姓名和登录时间：

```
$ who / awk '{ print $1, $5 }'
you 11:53
jim 11:27
$
```

下面的这段程序按时间顺序打印姓名与时间：

```
$ who / awk '{ print $5, $1 }' / sort
11:27 jim
11:53 you
$
```

这些是本章前面所示例的sed各版本的另一种实现方法。虽然，对于这类操作，awk比sed更容易使用，但当有许多输入时，其起始与执行的速度都很慢。

awk通常假设用空白空间(任意数量的空格与Tab)分隔字段，但是，分隔符可以用任意单个字符代替，一种方法是用-F(大写)命令行选项，如：口令文件/etc/passwd中的字段是由冒号分隔的：

```
$ sed 3q /etc/passwd
root:3D.fHR5KoB.3s:0:1:S.User::
ken:y.68wd1.iJayz:6:1:K.Thompson:/usr/ken:
dmr:z4u3dJWbg7wCk:7:1:D.M.Ritchie:/usr/dmr:
```

```
$
```

为了打印来自于第一个字段的用户名，可以用

```
$ sed 3q /etc/passwd / awk -F: '{ print $1 }'
root
ken
dmr
$
```

空格与Tab的处理是有意这样特殊设置的。缺省地，空格与Tab都是分隔符，且忽略前导的分隔符。然而，若分隔符被设置为非空格的任意其他字符，则在确定字段时，前导的分隔符将起作用。特别地，若分隔符是Tab，则空格不作为分隔符字符，前导空格作为字段的一部分，每个Tab定义一个字段。

4.4.2 打印

除了输入字段的个数外，awk还记载其他有意义的数量。内嵌变量NR是当前输入“记录”或行的序数。因此，为了把行数添加到一输入数据流中，可使用：

```
$ awk '{ print NR , $0 }'
```

字段\$0是未加更改的整个输入行。在print语句中，由逗号分隔的项在打印时是由输出字段分隔符(缺省时是空格)相分隔的。

print命令的打印格式可以实现一般的打印要求，若不然，也可以使用printf完全控制输出，例如：为了在四个数字宽的字段中打印行数，可以使用以下命令：

```
$ awk '{ printf "%4d %s\n", NR, $0 }'
```

%4d指定字段中的十进制整数(NR)为四个数字宽；%s表示字符串(\$0)；\n表示换行，这是因为printf不自动打印任何空格或换行。awk中的printf语句类似于C函数。参见printf(3)。

我们已写过第一版的ind命令(参见本章前面的部分)，如：

```
awk '{ printf "\t%s\n", $0 }' $*
```

在输入记录前打印Tab(\t)。

4.4.3 模式

假设想要在/etc/passwd中查找没有口令的用户。加密的口令位于第二个字段，因此，程序可用如下模式：

```
$ awk -F: '$2 == ""' /etc/passwd
```

此模式查看第二个字段是否为空字符串(== 是相等测试操作符)。可用几种方法写此模式：

\$2 == ""	第二个字段为空
\$2 ~ /^\$/	第二个字段与空串相匹配
\$2 !~ /. /	第二个字段不与任何字符相匹配
length(\$2) == 0	第二个字段的长度为零

符号~表示与正则表达式匹配，!~ 的意义是“不匹配”。正则表达式自身被包括在两个斜杠内。

length是awk的内部函数，它得到字符串的长度。可以在模式前加!，使其变反，如在

```
!($2 == " ")
```

中，操作符!同于C中的!，但与sed相反，在sed中，!遵循原模式。

awk中模式通常用于简单的数据验证任务。最多也不过是查找不满足某些规则的行。若没有任何输出，则数据是可接受的(没有消息就是最好的消息)。例如，以下模式使用操作符%来计算余数，以确保每个输入记录具有奇数个字段：

```
NF % 2 != 0      # 若字段数为奇数则打印
```

以下则是利用内部函数length，打印过长的行：

```
length($0) > 72   # 若太长，则打印
```

awk使用与shell一样的注释规则：以#作为注释的开始。

使用另一个内部函数substr，通过打印一个警告和过长的行的一部分，可使输出稍有些不那么格式化：

```
length($0) > 72 { print "Line", NR, "too long:", substr($0, 1, 60) }
```

substr(s, m, n)产生s的子串，它起始于位置m且具有n个字符长(字符串始于位置1)。若n被省略，则使用从位置m至末尾的子串。substr也可用于提取确定位置的字段，如从date的输出中选取小时与分钟：

```
$ date
Thu Sep 29 12:17:01 EDT 1983
$ date / awk '{ print substr($4, 1, 5) }'
12:17
$
```

练习4-7 你可以写多少个awk程序，用以像cat一样，将输入拷贝到输出？哪个最短？

4.4.4 BEGIN与END模式

awk提供了两个特殊的模式，BEGIN和END。BEGIN在读入第一个输入行之前就被执行，可使用BEGIN模式初始化变量，打印标题头或通过指定变量FS设置字段分隔符：

```
$ awk 'BEGIN { FS = ":" }
>      $2 == " " ' /etc/passwd
$                                     (没有输出：都使用口令)
```

END动作在处理完最后一行后执行：

```
$ awk 'END { print NR }' ...
```

打印输入的行数。

4.4.5 算术运算与变量

迄今所举的例子都仅涉及简单的文本处理，awk的真正实力还表现在对输入数据进行计算的能力，使用它可以很容易地实现计数、累计总和和求平均数等功能。awk的常用功能是求各列的总和。例如，为求第一列中所有数字之和：

```
    { s = s + $1 }
END    { print s }
```

由于值的个数可从参数NR中得到，因此，将最后一行改为：

```
END    { print s, s/NR }
```

则可打印总和与平均数。

这个例子也说明了awk中变量的用法，s不是一个内部变量，是在第一次调用时定义的一个变量，缺省地，变量被初始化为零，因此，通常不必考虑初始化的问题。

awk提供与C一样的算术操作符，因此，上面的例子通常写作：

```
{ s += $1 }
END { print s }
```

s += \$1与s = s + \$1一样，但书写格式更紧凑。

可以用以下方法对输入行求和：

```
{ nc += length($0) + 1 # number of chars, 1 for \n
  nw += NF              # number of words
}
END { print NR, nw, nc }
```

本例计算其输入中的行数、单词数和字符数，因此，其功能同于wc(虽然它并不通过文件将总和分解)。

再来看另一个算术运算的例子，此程序计算文件页数，每页66行，该页是通过pr打印一组文件而产生的。它可以打包在prpages命令中：

```
$ cat prpages
# prpages: Compute number of pages that pr will print
wc $* |
awk '!/total$/ { n += int(($1+55) / 56) }
END { print n }'
$
```

pr将56行的文本置于每页(由经验确定)。对与行尾的total不匹配的每一wc的输出行，页数被汇总，然后由内部函数int截为一整数。

```
$ wc ch4.*
 753   3090   18129   ch4.1
 612   2421   13242   ch4.2
 637   2462   13455   ch4.3
 802   2986   16904   ch4.4
  50    213    1117   ch4.9
2854 11172   62847   total
$ prpages ch4.*
53
$
```

为验证此结果，直接在awk中运行pr：

```
$ pr ch4.* / awk 'END { print NR/66 }'
53
$
```

awk中的变量还可以存放字符串。变量是被视为数字串还是字符串，取决于上下文。粗略地说，在诸如s += \$1这样的算术表达式中，常用数字串；而在字符串环境下，如x = "abc"，则使用字符串值；在诸如x > y这样模棱两可的情况下，除非操作数很显然是数值，否则使用字符串。(在awk手册中有精确定义的规则。)字符串变量被初始化为空字符串，下一节我们将充分利用字符串。

awk自身内含许多这两种类型的内嵌变量，如NR和FS，表4-3给出了其完整的列表，表4-4列出了操作符。

表4-3 awk内部变量

变 量	含 义
FILENAME	当前输入的文件名
FS	字段分隔字符(缺省为空格和Tab)
NF	输入记录中字段的个数
NR	输入记录数
OFMT	数字的输出格式(缺省为%g，参见printf(3))
OFS	输出字段分隔字符串(缺省为空格)
ORS	输出记录分隔字符串(缺省为换行)
RS	输入记录分隔符(缺省为换行)

表4-4 awk操作符(按优先级的升序排列)

操 作 符	含 义
= += -= *= /= %=	赋值；v op= expr即为v = v op (expr)
	或：expr1 expr2为真，若任意为真； 若expr1为真，则不计算expr2
&&	与：expr1 && expr2为真，若二者皆为真； 若expr1为假，则不计算expr2
!	表达式求反
> >= < <= == != ~ !~	关系操作符；~和!~是匹配和不匹配
nothing	字符串连接
+ -	加、减
* / %	乘、除、求余数
++ --	增量、减量(前缀或后缀)

练习4-8 试用不同的方法实现prpages，比较哪种方法较快。

4.4.6 控制流

在编辑大型文档时，很容易写重一个单词(从经验上说)，即相邻的两个单词相同。很显然，这通常都不是有意识产生的。为防止这类问题，Writer's Workbench程序类中一个称为double的程序，用于查找相邻的、成对的、完全相同的单词。以下是在awk中double的实现：

```
$ cat double
awk '
FILENAME != prevfile { # new file
    NR = 1              # reset line number
    prevfile = FILENAME
}
NF > 0 {
    if ($1 == lastword)
        Printf "double %s, file %s, line %d\n", $1, FILENAME, NR
    for ( i = 2 ; i <= NF ; i++)
        if ($i == $(i-1))
            printf "double %s, file %s, line %d\n", $i, FILENAME, NR
    if (NF > 0)
        lastword = $NF
}
```

```
} ' $*
$
```

运算符++递增其操作数，--则递减其操作数。

内部变量FILENAME表示当前输入文件名，由于NR从输入开始时就进行行计数，因此，每当文件名改变时，将其复位，从而可正确地识别不恰当的行。

awk的if语句与C中的相同：

```
if (条件)
    语句1
else
    语句2
```

如果条件为真，则执行语句1，否则，若有else部分，则执行语句2。else部分是可选的。

for语句也同于C中的循环语句，但与shell的有所不同：

```
for (表达式1; 条件; 表达式2)
    语句
```

for同于以下的while语句，它在awk中也有效：

```
表达式1
while (条件) {
    语句
    表达式2
}
```

例如，

```
for ( i = 2; i <= NF; i++ )
```

i依次被设置为2、3、…，直至域的总数NF，执行循环。

break语句的作用是从while或for循环中退出；continue语句跳到下次循环的开始(在while语句中从condition开始，for语句中从expression2开始)；next语句促成在awk程序起始处重新开始读入下一个输入行和匹配模式；exit语句则立即转向END模式。

4.4.7 数组

和大多数程序设计语言一样，awk也提供数组，举一个不是很有意义的例子，下面的awk程序将每个输入行收集到单个数组元素中，以行数为索引，然后以逆序将其打印输出：

```
$ cat backwards
# backwards: print input in backward line order
awk ' { line[NR] = $0 }
END   { for (i = NR; i > 0 ; i--) print line[i] } ' $*
$
```

注意，与变量相同，数组不必声明。一个数组的大小仅受所用机器可用内存的限制，若要将一个相当大的文件读入数组，可能会导致内存溢出。如果要以逆序打印一个庞大的文件的尾部，需要与tail相结合：

```
$ tail -5 /usr/dict/web2 / backwards
zymurgy
zymotically
```

```

zymotic
zymosthenic
zymosis
$

```

为了不去读介于中间的数据而直接进到文件的尾部，tail充分利用称为seeking的文件系统操作，查看第7章中关于lseek的讨论。(当前版本的tail有一个-r选项，它代替了backwards，以逆序打印行)。

标准输入处理将每个输入行分成若干字段，可以用内部函数split对任意字符串执行同样的字段分割操作：

```
n = split (s, arr, sep)
```

将字符串s分成若干字段，并把这些字段分别保存在数组arr从1至n的元素中。若提供了分隔符字符sep，则使用它；否则，使用FS的当前值。例如，split(\$0, a, ":"), 以冒号为分隔符分隔输入行，它适合于处理/etc/passwd，而split("9/29/83", date, "/")则以斜杠作为分隔符。

```

$ sed 1q /etc/passwd / awk '{split ($0, a, ":"); print a[1]}'
root
$ echo 9 / 29 / 83 / awk '{split($0, date, "/"); print date[3]}'
83
$

```

表4-5列出了awk中的内嵌函数。

表4-5 awk中的内嵌函数

函 数	含 义
cos(表达式)	表达式的余弦
exp(表达式)	对表达式取自然指数
getline()	读入下一个输入行：若文件结束，则返回0；否则为1
index(字符串1, 字符串2)	字符串1中字符串2的位置：若不存在，则返回0
int(表达式)	表达式的整数部分，向0截取
length(字符串)	字符串的长度
log(表达式)	表达式的自然对数
sin(表达式)	表达式的正弦
split(字符串,a,c)	以字符c为分隔符将s分隔成a[1]...a[n]并返回n
sprintf(格式, ...)	格式化...
substr(字符串,m,n)	起始于位置m的字符串的n个字符的子串

4.4.8 关联数组

数据处理中很平常的一个问题是对名字与数值对的集合进行累加，如，从以下的输入中计算每个名字对应数值的总和：

```

Susie      400
John       100
Mary       200
Mary       300
John       100
Susie      100
Mary       100

```

即：

```
John      200
Mary      600
Susie     500
```

awk提供了实现此功能的一个简单方法：关联数组。虽然人们通常认为数组的下标为整数，但在awk中，任意数值都可作为下标。因此，

```
    { sum[$1] += $2 }
END   { for(name in sum) print name , sum[name] }
```

对于诸如以上那些名字与数值对，无论它们是否排序，求和并打印总和。每个名字(\$1)被用作sum中的一个下标，最后，使用一个特殊形式的for语句对所有的sum元素进行循环，并将其打印出来。在语法结构上，for语句的这个变形是：

```
for (数组中的变量)
    语句
```

虽然从表面上看，可能有些像shell中的for循环，但实际上是没有联系的。它对数组的下标循环，而不是元素。依次设置var为每个下标。下标是以不可预计的顺序产生的，因此，可能有必要对其进行排序。在上面的例子中，其输出可传送到sort，从而将具有最大值的人名列在最上面。

```
$ awk '...' / sort +1nr
```

关联数组的存储是用散列方法实现的，以确保对任何元素的存取都使用一样的时间，且时间不依赖于数组中元素的个数(至少对于中等规模的数组是这样)。

关联存储方式对于诸如求输入中所有单词的出现次数这样的任务，处理效率很高：

```
$ cat wordfreq
awk ' { for (i = 1; i <= NF; i++) num[$i]++ }
END   { for (word in num) print word, num[word] }
' $*
$ wordfreq ch4.* / sort +1 -nr / sed 20q / 4
the 372      .CW 345      of 220      is 185
to 175       a 167       in 109      and 100
.pl 94       .p2 94      .pp 90      $ 87
awk 87       sed 83      that 76     for 75
The 63       are 61     line 55    print 52
$
```

第一次for循环查看输入行中的每个单词，将以此单词为下标的数组num的元素递增(不要将awk的\$i，即输入行的第i个字段，与任意shell变量相混淆)。读完文件后，第二个for循环以任意顺序打印单词及其个数。

练习4-9 如上，wordfreq的输出包括了诸如.CW这样的文本格式命令，怎样才能去掉像这样无意义的单词？如何使用tr，使得wordfreq不考虑输入单词的大、小写而能正确工作？试比较wordfreq与4.2节的管道以及下面的程序的实现与性能：

```
sed 's/[ →]* /\n'
/g' $* | sort | uniq -c | sort -nr
```

4.4.9 字符串

虽然sed和awk都可用于处理诸如选取单个字段这样的小工作，但只有awk可用于真正需要

编程的各种工作。举一个例子，这个例子将较长的行调整为80列，任何超过80个字符的行，第80个字符之后的字符被截断，附加一个\以示警告，再处理剩余的部分。一个经调整的行是采取向右对齐，而非从左对齐，因为这样使处理结果更方便于程序排列输出，这也正是我们经常使用fold的目的所在。作为例子，我们用20个字符的行代替80个字符的行：

```
$ cat test
A short line.
A somewhat longer line.
This line is quite a bit longer than the last one.
$ fold test
A short line.
A somewhat longer li\
                      ne.
This line is quite a\
bit longer than the\
                    l ast one.
$
```

奇怪的是，第7版并没有提供添加或删除Tab的程序，但在系统V中，pr可以实现这两个功能。在fold中，用sed来将Tab转换为空格，使得awk的字符计数保持正确。对于行首的Tab(这在源程序中很典型)这也完全适用，但是不支持在一行中间出现的Tab。

```
# fold : fold long lines
sed 's/\t/ /g' $* | # convert tabs to 8 spaces
awk '
BEGIN {
    N = 80      # folds at column 80
    for (i = 1; i <= N; i++) # make a string of blanks
        blanks = blanks " "
}
{
    if ((n = length($0)) <= N)
        print
    else {
        for (i = 1; n > N; n -= N ) {
            printf "%s\\n", substr($0, i, N)
            i += N;
        }
        printf "%s%s\\n", substr(blanks, 1, N-n), substr($0, i)
    }
}
```

在awk中，没有明显的字符串连接操作符，当字符串相邻时，就被连接。初始时，blanks是一个空字符串，BEGIN部分的循环通过连接产生了一个很长的空格字符串：循环的每一步都在blanks的后面增加一个空格。第二个循环处理相当长的输入行，直至剩余的部分足够短为止。如同C中，赋值语句也可用作表达式，因此，语句：

```
if ((n = length($0)) <= N) ...
```

在测试数值前，将输入行的长度赋给n。注意语句中括号的使用。

练习4-10 修改fold使得它在空格或Tab处调整行，而不会将一个单词一分为二。同时，要求对于长的单词，其性能也足够好。

4.4.10 与shell的交互作用

如果要写一个程序field n打印输入的每一行的第n个字段，例如，想用命令；

```
$ who / field 1
```

打印登录名。显然，awk提供字段选取功能，主要的问题是如何将字段号n传给awk程序，下面是一种实现方法：

```
awk '{ print $' $1' }'
```

\$1不在任何引号之间，因此成为awk可以知道的字段号。另外，还有一种使用双引号的方法：

```
awk "{ print \$$1 }"
```

在这种情况下，参数由shell解释，因此，\ \$变为一个\$，\$1为n的值所替换。对于这两种方法，我们更倾向于使用单引号，因为在一个典型的awk程序中，对于双引号，需要过多额外的\。

第二个例子是addup n，它将第n个字段的数字求和：

```
awk '{ s += $' $1' }
END { print s }'
```

第三个例子是对n列中的每列分别单独求和，然后再将各列之和相加：

```
awk '
BEGIN { n = $1 }
{
    for (i = 1; i <= n; i++)
        sum[i] += $i
}
END {
    for (i = 1; i <= n; i++) {
        printf "%6g ", sum[i]
        total += sum[i]
    }
    printf "; total = %6g\n", total
}'
```

我们使用BEGIN将n的值插入到变量中，而不在程序的其余部分插入引号，避免程序很混乱。

所有这些例子的主要问题是弄明白哪些内容在引号内，哪些在引号外(虽然这是一件麻烦事)，但对于现在所写的程序，只能从标准输入读入，无法将参数n和任意长的文件名列表传给它们。这需要一些shell程序设计，在下一章我们将会作介绍。

4.4.11 基于awk的日历服务

最后看一个使用关联数组的例子，同时也说明如何与shell交互，并展示程序的开发过程。

程序的任务是让系统每天早上将邮件发给你，提醒你一些未来事件(已有很多这样的日历服务，参见calendar(1)，本节介绍另一种方法)。程序提供的基本服务包括告知你今天的日程安排，第二步是给出一天的预告——今天及明天的事件，对于周末及节日的适当处理留作练习。

第一个需要解决的是设置保存日历的地方，最简单的是使用一个在/usr/you中称为calendar的文件。

```
$ cat calendar
```

```

Sep 30  mother's birthday
Oct 1   lunch with joe, noon
Oct 1   meeting 4pm
$

```

第二个需要解决的问题是，需要一个扫描日历以查找日期的方法，这有很多选择，可以用awk，因为它最擅长于计算从“今天”推算到“明天”这样的算术运算，尽管像sed或egrep这样的程序也可用。当然，从日历中选取的行是通过邮件发送的。

第三个需要解决的问题是，需要一个可以每天，很可能是在早上，自动、可靠地扫描日历的方法，这可由at来完成，在第1章中我们简要地提及过它。

若我们限制日历的格式，使每行以月、日开始，如同date所产生的那样，这样一个简单的日历程序是很容易编写出来的：

```

$ date
Thu Sep 29 15:23:12 EDT 1983
$ cat bin/calendar
# calendar:  version 1 -- today only
awk <$HOME/calendar '
    BEGIN { split ("`date`", date) }
    $1 == date[2] && $2 == date[3]
' | mail $NAME
$

```

BEGIN部分把由date产生的日期分成几个部分并存入一个数组的不同元素中，数组的第二、第三个元素为月和日，这里假设shell变量NAME包含你的登录名。

引号字符序列是在awk程序中以字符串里提取“日期”所必需的，另一个比较容易理解的方法是将日期传入，作为输入的第一行：

```

$ cat bin/calendar
# calendar:  version 2 -- today only, no quotes
(date; cat $ HOME/calendar) |
awk '
    NR == 1    { mon = $2; day = $3 } # set the date
    NR > 1 && $1 == mon && $2 == day # print calendar lines
' | mail $NAME
$

```

下一步是编排calendar来查找今天和明天。大多数时候所需的只是得到今天的日期，并把天数加1。但是，在月末时，就必须取得下一个月，并将日期值设置回1，当然，每个月有不同的天数。

这就是关联数组发挥作用之处。两个数组，days和nextmon，其下标是月份名，每个月的天数保存在日期数组days内，每个月的下一个月份名保存在数组nextmon中，从而，days["Jan"]等于31，nextmon["Jan"]等于Feb。数组初值的设置并不是创建像下面这样的整个语句序列：

```

days["Jan"] = 31; nextmon["Jan"] = "Feb"
days["Feb"] = 28; nextmon["Feb"] = "Mar"
...

```

而是采用split将一种方便的数据结构转换为真正所需的数据形式：

```

$ cat calendar
# calendar:  version 3 -- today and tomorrow
awk <$HOME/calendar `
BEGIN {
    X= "Jan 31 Feb 28 Mar 31 Apr 30 May 31 Jun 30" \
        "Jul 31 Aug 31 Sep 30 Oct 31 Nov 30 Dec 31 Jan 31"
    split(x, data)
    for (i = 1; i < 24; i += 2) {
        days[data[i]] = data[i+1]
        nextmon[data[[i]]] = data[i+2]
    }
    split("`date`", date)
    mon1 = date[2]; day1 = date[3]
    mon2 = mon1; day2 = day1 + 1
    if (day1 >= days[mon1]) {
        day2 = 1
        mon2 = nextmon[mon1]
    }
}
$1 == mon1 && $2 == day1 || $1 == mon2 && $2 == day2
` | mail $NAME
$

```

注意在数据中Jan出现了两次，像这样的“哨兵”数据值简化了December的处理。

最后一步是安排日历程序的每天运行，你所希望的是每天早上大约5点有人醒来运行calendar。其实你完全可以自己来做——通过记住说明(每天!):

```

$ at 5am
calendar
ctl-d
$

```

但这并不完全自动或可靠，关键是不仅要告知at运行calendar，还要告知其安排下次运行。

```

$ cat early.morning
calendar
echo early.morning | at 5am
$

```

第二行语句为第二天安排下一个at命令，因此，一旦启动，此序列就永久存在。at命令还可为你设置路径PATH、当前目录以及它处理的命令的其他参数，因此，你不必做特别的事。

练习4-11 修改calendar，使其分辨Friday是周末，tomorrow包括Saturday、Sunday和Monday。修改calendar以处理闰年。calendar能分辨节日吗？如何实现？

练习4-12 calendar能分辨一行之内，而不是正好在行首的日期吗？对于以其他格式，如10/1/83表示的日期又如何？

练习4-13 calendar为什么不以getname代替\$NAME？

练习4-14 写一个个人版本的rm，将文件拷贝到临时目录，而不是将其删除，并用一个at命令写一个程序，在你在睡觉时，清空目录。

4.4.12 附注

awk是一种内容十分广泛的语言，一章的篇幅是不可能介绍其所有功能的，下面是一些在手册中可读到的有关awk的其他方面的内容：

- 将print的输出重定向到文件和管道中：任意print或printf语句后可跟一个 > 和一个文件名(一个带引号的字符串或一个变量)，则输出被发送到指定文件。对于shell，>>是附加而不是覆盖。打印到管道则用|，而不是>。

- 多行记录：若记录分隔字符RS被设为换行，则输入记录会为一个空行所分隔，这样的话，几个输入行可被视作单个记录。

- 可选用pattern, pattern格式：和ed与sed中一样，可由模式对来确定行的范围。它匹配从第一个模式匹配的行直至第二个模式匹配的行之间的所有行。一个简单的例子是：

```
NR == 10, NR == 20
```

匹配从第10行到第20行之问的所有行。

4.5 好的文件与过滤程序

虽然最后几个awk的例子包括一些独立的命令，但许多awk的使用都是简单的一到两行的程序，它们作为很长的管道的一部分，做些过滤的工作。大多数的过滤程序都是这样，有时，手边的一些问题可以由单个的过滤程序来解决，但更普遍的做法是，将问题分成若干可由过滤程序解决的子问题，而这些过滤程序被联合成一个管道。像这样使用工具的方法常被认为是UNIX程序设计环境的核心。不过，这个观点过于有限，过滤程序的使用遍及整个系统，其为何有用，倒是值得一番论述。

UNIX程序产生的输出，可以作为其他程序的输入，其格式能为其他输入程序所理解。可过滤的文件包含文本行、文件头、文件尾或空行。每一行都是一个有意义的对象——文件名、单词，对运行过程的描述——因此，像wc和grep这样的程序可计算有意义的项的总和或者可以按名查找。当描述对象更多的信息时，文件仍是按行显示，但按列由空格或Tab把行分隔成字段，如同ls-l的输出那样。假设数据分割成这样的字段，像awk这样的程序就可以很容易地选取、处理或重新排列这些信息。

过滤程序有着相同的设计思想，每个都是将对参数文件的处理结果写到标准输出，若是没有给定参数则使用标准输入。参数只用于指定输入，而不是指定输出^①，因此，一个命令的输出总可以送往管道。可选参数(或诸如grep这样的无文件名参数的模式)位于任意文件名之前。最后，错误信息总是写于标准错误输出上，因此不会被送到管道中。

这些规则对单个命令是无效的，但当所有的程序都遵循这些规则时，会简化它们的相互连接，本书的许多例子都说明了这点，最明显的是4.2节后面的单词计数的例子。若任一程序都要求用命名文件进行输入输出，交互地请求参数，或者都生成文件头和尾，管道便无法工作。当然，若UNIX系统不提供管道功能，人们就不得不写一常规程序来做此工作。但现在有管道，且管道工作良好，如果能熟悉地使用这些工具，编写程序就更容易了。

练习4-15 ps打印说明性文件头，ls-l显示文件中块的总数。解释这一结果。

① 一个早期的UNIX文件系统就是被违背此规则的一段维护程序所破坏的，因为一个看上去无害的命令破坏了所有的磁盘。

有关文献

在论文“Pattern matching in string” (Proceedings of the Symposium on Formal Language Theory, Santa Barbara, 1979年)中有模式匹配算法的详细说明, 其作者是egrep的作者Al Aho。

sed是由Lee McMahon设计与实现的, 它以ed为基础。

awk是由Al Aho、Peter Weinberger和Brian Kernighan设计与实现的, 它远不是一个一流的设计, 以作者的名字命名一种语言, 也表明了其想象力的缺乏。实现者的一篇论文“AWK—a pattern scanning and processing language” (Software—Practice and Experience, 1978年7月), 论述了其设计。awk来源于多个方面, 但肯定是来源于SNOBOL4、sed、Marc Rochkind设计的有效性语言、语言工具yacc和lex, 当然还有从C中窃取了一些好的思想。确实, awk与C的相似性正是问题的根源——它看上去像C, 却又不是。一些语法结构没有了, 其他的也只有微小的差别。

Doug Comer 的文章“The flat file system FFG: a database system consisting of primitives” (Software—Practice and Experience, 1982年11月)论述了用shell和awk建立数据库系统。

第5章 shell编程

尽管大多数用户认为shell是一个交互式命令解释器，但实际上它是一种程序设计语言，它的每一条语句运行一条可执行的命令。shell要同时满足交互执行命令和编程运行命令两种使用方式的要求，是一种特殊的语言，造成这种局面，既有历史的原因，也有设计方面的影响。它的应用程序范围广泛，而且从语言的观点看目前还存在着许多争议，但不了解这些争议的细节并不影响用户高效地使用shell进行程序设计。本章的目的是通过逐步开发一些有用的shell程序，说明shell编程的基本原理。本章不是一部shell的参考手册，阅读本章时应该在手边准备好《UNIX程序员手册》中关于sh(1)的部分，以便随时参考。

和其他很多命令一样，shell程序的使用细节通常可以通过实践很快地掌握。shell手册常常不易于理解，而一个好的例子是将问题解释清楚的最好方法。鉴于这一原因，本章围绕着程序实例而不是shell的特性来组织，是shell编程指南，而不是shell所有内容的罗列。这里不仅讨论shell的各种功能，而且讨论shell程序的开发和设计，同时强调交互式程序测试的思想。

当用shell或者其他语言编写程序时，如果恰有其他人也需要使用这个程序，将有助于程序的进一步改进和完善。因为他人对使用程序的要求往往要比程序编写者本人更苛刻，因此，shell编程的一个重要原则在于提高程序的健壮性，要能处理不正确的输入，当程序运行出现错误时，也要能给出有用的提示信息。

5.1 定制cal命令

shell程序的一个通常用途是增强或改善应用程序的用户界面。作为一个增强程序界面的例子，考虑cal(1)命令：

```
$ cal
usage: cal [moth] year    到目前为止还不错
$ cal October 1983
Bad argument              参数不对
$ cal 10 1983
    October 1983
S  M  Tu  W  Th  F  S
                1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$
```

这里的缺点是月份必须使用数字输入。当只键入cal 10时，将打印10年的年历，而不是打印当年10月份的月历。因此，要打印一个月的月历，必须同时输入相应的年份。

重要的是，不论cal命令提供的是什么接口，都可以只改变用户接口而不需要改动cal程序

本身。可以把命令放在自己的bin目录里，将一个更易于使用的参数语法转化为实际的cal命令要求的参数格式，甚至还可以调用自己的cal版本，这样就更加直接一些。

在编制新的cal命令时，首先要考虑的问题是：cal应该做些什么？我们的基本设想是要求cal能够更合理地工作。cal应该能通过名字识别月份，当有两个参数时，除了将月份名转换为相应数字以外，其他功能和原来的cal命令完全一样。当仅给定一个参数时，cal应打印当年相应月份的月历，而当不提供参数时，它只打印当月的月历。这些正是cal命令最常用的情况。综上所述，要解决的主要问题是：确定有多少个参数，如何将这些参数转换为标准cal要求的格式。

shell提供的case语句正好适用于进行上述判断：

```
case word in
  模式)  命令 ;;
  模式)  命令 ;;
  ...
esac
```

case语句将word和pattern从头至尾进行比较，当遇到第一个匹配的模式时，执行与该模式相应的命令，而且仅仅执行这一命令。模式均按照shell模式匹配规则书写，对文件名匹配规则稍微作了一些推广。每项匹配所对应的命令均以双重分号;;结尾。(最后一个匹配项的命令之后可以不用;;，但为了编辑的方便，通常还是保留它。)

我们的cal版本能确定出现在命令行中参数的个数，处理以字母表示的月份名，然后再调用前述真正的cal。shell变量\$#保存了调用shell文件时参数的个数，其他特殊shell变量列在表5-1中。

```
$ cat cal
#cal:  nicer interface to/usr/bin/cal

case $# in
0)      set `date`; m=$2; y=$6;;          # no args: use today
1)      m=$1; set `date`; y=$6;;          # 1 arg: use this year
*)      m=$1; y=$2 ;;                     # 2 args: month and year
esac

case $m in
jan* | Jan*)    m=1 ;;
feb* | Feb*)    m=2 ;;
mar* | Mar*)    m=3 ;;
apr* | Apr*)    m=4 ;;
may* | May*)    m=5 ;;
jun* | Jun*)    m=6 ;;
jul* | Jul*)    m=7 ;;
aug* | Aug*)    m=8 ;;
sep* | Sep*)    m=9 ;;
oct* | Oct*)    m=10 ;;
nov* | Nov*)    m=11 ;;
dec* | Dec*)    m=12 ;;
[1-9] | 10 | 11 | 12) ;;                  # numeric month
*)              y=$m; m=" " ;;            # plain year
```

表5-1 shell内部变量

变 量	含 义
\$#	参数个数
\$*	shell的所有参数
\$@	类似\$, 参见5.7节
\$-	shell用的选择项
\$?	上次执行的命令的返回值
\$\$	shell的进程标识符
\$_	用&启动的最后一个命令的进程标识符
\$HOME	cd命令的缺省参数
\$IFS	参数分隔词的字符表
\$MAIL	信件文件, 当其改变时给出信息you have mail
\$PATH	搜索命令的目录表
\$PS1	提示字符串, 缺省为\$
\$PS2	后继命令行提示字符串, 缺省为>

```
esac
```

```
/usr/bin/cal $m $y
```

```
# run the red one
```

```
$
```

第一个case检查参数个数 \$ #, 并选择对应的操作。第一个case最后的模式*表示匹配所有的情况, 即当参数个数既非0又非1时, 执行最后一种情况。(因为各个模式是顺序扫描, 所以与所有情形匹配的模式一定是最后扫描的。)m和y变量分别作为月份和年份——当给定两个参数时, 我们的这个cal命令与原始的cal命令执行相同的操作。

第一个case语句有两行都包括了这样一条语句:

```
set `date`
```

虽然字面上这一语句的功能不太明显, 但是通过下面几条命令很容易看出这一语句的作用:

```
$ date
Sat Oct 1 06:05:18 EDT 1983
$ set `date`
$ echo $1
Sat
$ echo $4
06:05:20
$
```

set是shell的一个内部命令, 它能够处理相当多的事情。当没有参数时, set给出环境变量值, 正如第3章中所提到的。set还能实现重置基本参数如\$1、\$2等的功能, Set `date`把\$1重置为星期几, \$2重置为月份, 等等。因此, 在cal程序的第一个条件语句case中, 当月份或年份缺省时, 按照当前日期设置月份或年份。只有一个参数时, 将该参数作为月份, 而年份从当前日期中取得。

set还可以识别多个选项。使用最多的是-v和-x, 设置了这些选项后, 每条命令在由shell处理执行时, 会返回所运行的命令。这对于调试复杂的shell程序是必不可少的。

剩下的问题是月份的转换。即当月份是文字形式时, 要将它转换成数字形式。第二个case语句正好完成这一工作。无需多说, 这段程序的功能是一目了然的。这里唯一不易理解的是

case语句中的|符号，它与egrep中的号相同，表示选择，如big|small表示或与big匹配，或与small匹配。当然这一条件也可以写成[jJ]an*等其他形式，程序能接受全部小写的月份名或者以大写开头的月份名，前者是因为UNIX系统主要接收小写，而后者是因为date命令打印的月份的第一个字母就是大写。shell模式匹配的规则列在表5-2内。

表5-2 shell模式匹配规则

符 号	含 义
*	匹配任意字符串，包括空字符串
?	匹配任意单个字符
[CCC]	匹配CCC中的任意字符，如[a-d0-3]等于[abcd0123]
"..."	完全与...匹配。引号保护特殊符号，也可以写成'...'
\C	匹配文字C
a b	只用于条件表达式，与a或b匹配
/	对于文件名，仅与表达式中显式的/符号相匹配；对于case语句，与其他字符一样匹配
.	作为文件名的第一个字符，仅与表达式中显式的·符号相匹配

第二个case语句中的最后两种情形用于处理单个的参数，它可能是年份，但第一个case语句会将它当成月份，如果它可以用来表示月份，则作为月份处理；否则，作为年份处理。

最后一行使用转换后的参数，调用/usr/bin/cal(真正的cal命令)。我们最后得到的这一cal程序版本可能正是初学者所设想的。

```
$ date
Sat Oct 1 06:90:55 EDT 1983
$ cal
October 1983
S M Tu W Th F S
          1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$ cal dec
December 1983
S M Tu W Th F S
          1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
$
```

若键入cal 1984，将打印出1984年全年日历。

这个改进了的cal程序完成与原来的cal程序相同的工作，但是实际使用起来却更简单，更容易记忆，在名字选取上也尽量采用简单的方式，即用cal，而不用calendar(它已经是另一个命令)或任何其他不利于记忆的名称，如ncal。不改变命令的名字还有一个优点，即用户不会

因为新名字产生新的印象。

在结束关于case语句的讨论之前，还需说明一点：即为什么shell程序的模式匹配规则不同于ed程序及其衍生程序中的匹配规则。毕竟采用两种不同的模式意味着要学习两组规则，要求采用两种代码来处理这两组模式。有些区别源自最初的错误选择，后来也从未改正——例如，匹配任意符号的模式在ed里使用`.`，在shell里使用`?`，这仅仅是为了保持对过去的兼容性，除此之外没有任何其他原因。另一方面，有时，不同的模式完成不同的工作。在ed中的正则表达式可以搜索行内任何位置出现的字符串，特殊符号`^`和`$`的作用是将搜索定位到行首或行尾。然而在缺省情况下，对文件名的搜索总是定位的，在这种情况下，如果不得不用命令

```
$ ls ^?*.c$
```

这种格式无效

来代替命令

```
$ ls *.c
```

将是非常麻烦的。

练习5-1 如果其他用户希望使用你的cal版本，如何使它被所有用户共享？把它放在/usr/bin目录中需要执行哪些操作？

练习5-2 有必要使cal程序在cal 83时打印1983年的年历吗？如果有必要的话，你怎样实现打印83年的年历。

练习5-3 修改cal使之接受多个月份，如：

```
$ cal oct nov
```

或者一个连续范围的月份：

```
$ cal oct - dec
```

假如现在是12月，而运行cal Jan，应该得到的是今年的1月还是明年的1月？应如何考虑这个问题？

5.2 which命令

建立自己的命令版本，如cal命令的新版本，会带来一些其他问题。最明显的例子是，如果和Mary一起工作，并且以mary登录，则此时的cal还是标准的版本，除非Mary把新的cal命令连接到她的bin目录里。你可能会非常疑惑——原先的cal命令给出的错误信息不足以使人弄清发生错误的原因。这是一类问题的一个典型例子。因为shell通过PATH指定的一组目录搜索命令，得到的可能不是所期望的版本。例如，键入一条命令，例如echo，而实际运行的文件的完整的目录可能是./echo、/bin/echo、/usr/bin/echo或是别的什么目录，这取决于搜索命令的目录PATH以及文件存在的具体位置。如果存在一个名字相同，但行为完全不同的命令，就可能把人弄糊涂。最常见的情况是test，关于这一命令在以后还要讨论，因为大家往往喜欢用test来命名程序的临时版本，但通常会执行错误的test程序[⊖]。这种情况下，一个能报告即将执行程序的版本的命令能为用户提供很有用的服务。

一种实现方法是对PATH里的目录进行循环搜索，找出给定名字的每一个执行文件。在第3章里用for语句实现对文件名和参数的循环，这里我们给出所要求的循环结构：

⊖ 后面将介绍如何在经常使用test的shell文件中避免这一问题。

```
for i in PATH 的各个部分
do
    如果给定的名称在目录之中,
    则打印它的完整的目录
done
```

```

0                                0表示运行正常：文件相同
$ cmp /usr/you/.profile /usr/mary/.profile
/usr/you/.profile /usr/mary/.profile differ: char 6, line 3
$ echo $?
1                                非0表示文件不同
$

```

有几个命令，如cmp和grep，都有选项-s，使他们仅以适当的状态返回，但不产生任何输出。shell的if语句按照命令的退出状态执行选择，如：

```

if 命令
then
    如果条件为真时的命令
else
    如果条件为假时的命令
fi

```

其中换行是很重要的，fi、then和else仅在换行或分号之后才能被识别。else部分是可选的。

if语句至少要运行一条命令——条件中的命令，而case语句直接在shell中做模式匹配。在某些UNIX版本，包括System V中，test是shell的内部函数，所以if和test能和case运行得一样快。如果test不是内部函数，case语句就比if语句更有效，在模式匹配时应该使用case语句来执行：

```

case "$1" in
hello) 命令
esac

```

这一语句要比下面的语句运行得快：

```

if test "$1" =hello                除非test是一个shell内部命令，否则会运行较慢
then
    命令
fi

```

这就是为什么有时我们使用shell中的case语句进行条件测试，尽管在大多数程序设计语言中使用if语句就可以了。但另一方面，case语句不能方便地判断文件是否可以读，这时最好使用test命令和if语句。

现在我们开始编写which命令的第一个版本，它将报告与给定命令相匹配的文件：

```

$ cat which
# which cmd:  which cmd in PATH is executed, version 1

case $# in
0)      echo 'Usage: Which command' 1>&2; exit 2
esac
for i in `echo $PATH | sed 's/^:/:./
                s/:/:./g
                s/:$/:./
                s:/ /g`
do
    if test -f $i/$1                # use test -x if you can
    then
        echo $i/$1
    fi
done

```

```

        exit 0                # found it
    fi
done
exit 1                # not found
$

```

让我们试验一下这个命令:

```

$ cx which                *使其成为可执行文件
$ which which
./which
$ which ed
/bin/ed
$ mv which /usr/you/bin
$ which which
/usr/you/bin/which
$

```

开始的case语句进行错误检查。注意echo后的重定向语句1>&2, 表示不把错误信息送往管道。shell内部命令exit用于返回退出状态。当命令不工作时, exit 2用于返回错误状态; 当没有找到文件时返回exit 1; 当找到文件时返回exit 0。如果程序中这一位置没有给出exit语句, 则退出状态就是shell文件最后一个命令执行的状态。

假如在当前目录中有一个test程序, 那么会发生什么情况呢?(假定test不是shell内部命令。)

```

$ echo 'echo hello' >test    做一个假的test
$ cx test                    使它可执行
$ which which                现在试验which
hello                        失败
./which
$

```

上述程序调用了更多的错误检查。可以运行which(假设在当前目录里没有test程序!)去寻找test文件的全路径, 并打印它。但这还不是十分令人满意: 因为在不同的系统中, test可能位于不同的目录里, 而且which要依靠sed和echo, 所以还要指定它们的路径名。一个比较简单易行的办法是: 在shell文件里固定PATH, 使得在执行which命令时, 仅搜索/bin和/usr/bin目录。当然, 必须保存以前的PATH中的目录搜索顺序。

```

$ cat which
# which cmd: which cmd in PATH is executed, final version

opath=$PATH
PATH=/bin:/usr/bin

case $# in
0)      echo 'Usage: Which command' 1>&2; exit 2
esac
for i in `echo $opath | sed 's/^:/:./
        s/:/:./g
        s/:$/:./
        s/:/ /g`
do

```

```

        if test -f $i/$1          # this is /bin/test
        then                      # or /usr/bin/test
            echo $i/$1
            exit 0                 # found it
        fi
    done
    exit 1                         # not found
$

```

这样which可以工作了，即使在搜索途径中存在一个假的test(或假的sed和假的echo)。

```

$ ls -l test
-rwxrwxrwx 1 you          11 oct 1 06:55 test          仍在那里
$ which which
/usr/you/bin/which
$ which test
. /test
$ rm test
$ which test
/bin/test
$

```

shell提供了另外两个操作符用于组合命令，这就是||和&&，它们的使用形式比if语句更加简练方便。例如，||能够代替一些if语句。

```
test -f 文件名 || echo file 文件名 does not exist
```

等价于：

```

if test ! -f 文件名              !表示对条件求反
then
    echo file 文件名 does not exist
fi

```

尽管形式上相似，但操作符||并不是管道，而是表示“或”的条件操作符。执行||左边的命令后，若退出状态为零(成功)，则忽略||右边的命令；若退出状态为非零(失败)，执行右边的命令，并且整个表达式的值是右边命令退出时的状态值。换言之，||是“或”条件操作符，左边的命令执行成功时不执行右边的命令。对应的&&条件操作符表示“与”，仅当左边的命令成功才执行右边的命令。

练习5-4 为什么which文件不在退出前把路径PATH恢复成opath?

练习5-5 shell使用esac终止case，使用fi终止if，为什么使用done终止do呢?

练习5-6 把选项-a加到which命令中，使它打印所有在PATH里的文件，而不是打印第一个文件之后便退出。提示：match='exit 0'

练习5-7 修改which，使之可以识别shell的内部函数，如exit。

练习5-8 修改which，使之能检查文件的执行权限。当不能找到文件，改变which使之打印错误信息。

5.3 while和until循环

在第3章中，for循环用于一些简单的迭代程序。一般地，for对一组文件名进行循环，如在for i in *.c中；或对shell程序的所有参数进行循环，如for i in \$*。但实际上，shell中的for循

环可以有更大的用途，例如在which里的for循环。

shell有三种循环：for、while和until，而其中for是最常用的一种循环。对于循环变量的每一个元素，它执行一次循环体内的一组命令。最常用的循环变量是文件名。while和until循环利用命令的退出状态来控制循环体内命令的执行。循环体一直执行，直至作为条件的命令返回状态为非0(对于while循环)或0(对于until循环)。while和until循环除了对命令的退出状态进行不同的解释之外，其他完全相同。下面是各种循环的基本格式：

```
for i in 文件列表
do
    循环体，$i依次取表中的元素值
done

for i
    隐含地表示对shell文件的全部参数，即$*
do
    循环体，$i依次取参数值
done

while 命令
do
    循环体，只要命令返回状态为真便继续执行
done

until 命令
do
    循环体，只要命令返回状态为假便继续执行
done
```

在第二种形式的for循环中，空的循环参数表示\$*，这是常用的缩写形式。

控制while或until的条件命令可以是任何命令。例如，下面是一个观察某人登录情况的循环：

```
while sleep 60
do
    who | grep mary
done
```

其中sleep命令用于挂起60秒，它总能正确执行(除非发生中断)，因此总返回“成功”，这样，这个循环每分钟检查一次，查看Mary是否已登录。

这一版本有个缺点，如果mary已经登录，也必须等60秒钟后才能知道Mary登录的信息。还有，当Mary已经登录到系统中，你就会每60秒钟接收到一个报告。可以将这个程序修改一下，使用until循环来编写，此时，如果Mary已登录到系统，你不需延迟就可以立即得到Mary的登录信息：

```
until who | grep mary
do
    sleep 60
done
```

这是一个更有意思一些的条件。如果Mary已经登录，who | grep mary打印出who列表中Mary这一行，并返回“真”值，因为grep返回的状态表示它是否找到了匹配的内容，管道命令的

退出状态是最后一个元素的退出状态。

最后，我们可以将这条命令包装起来，给它取一个名字，并安装到系统中：

```
$ cat watchfor
# watchfor : watch for someone to log in

PATH=/bin; /usr/bin

case $# in
0)      echo `Usage: watchfor person`1>&2; exit 1
esac

until who | egrep "$1"
do
    sleep 60
done

$ cx watchfor
$ watchfor you
you      tty0      Oct   1 08:01      成功了
$ mv watchfor /usr/you/bin      进行安装
$
```

可以将grep改成egrep，这样可以输入：

```
$ watchfor 'joe / mary'
```

来同时观察多个人的登录情况。

作为一个更复杂的例子，我们考虑一个功能更强的who命令：它不仅能观察所有人的登录情况，并能及时报告用户登录或退出系统的情况。它的基本结构很简单：每分钟运行一次who命令，并与一分钟前的输出情况相比较，如果有不同，报告发生的改变。who命令的输出将保存在一个文件里，我们将这个文件保存在目录/tmp中。为了区分我们的文件和其他进程的文件，可以将shell变量\$\$ (shell命令的进程标识符)放在文件名里，这是一种常用的方法。将命令的名字编码到临时文件名中是系统管理员常用的手段之一。各种命令(包括这个watchwho命令)常常把文件放在/tmp目录，最好了解是哪个命令在这么做。

```
$ cat watchwho
#watchwho: watch who logs in and out

PATH=/bin:/usr/bin
new=/tmp/wwho1.$$
old=/tmp/wwho2.$$
>$old      # create an empty file

while :      # coop forever
do
    who>$new
    diff $old $new
    mv $new $old
    sleep 60
done | awk '/>/ { $1 = "in:      ";print}
        />/ { $1 = "out:      ";print}'
$
```

: 是shell的内部命令, 它仅仅计算参数值, 然后返回“真”。在这里, 我们也可以用true命令代替它。true命令仅仅返回一个“真”的退出状态。(还有一个false命令。)但是,: 不需要执行文件系统中的命令, 所以它比true命令效率更高一些。

diff命令的输出用<和>来区分来自两个文件中的不同数据。awk程序处理diff输出, 并以更容易理解的格式显示出来。注意, 整个while循环的结果通过管道送入awk, 而不是每分钟刷新一次awk。sed不适用于这种处理情况, 因为它的输出总是比输入落后一行, 即总有一行的输入正在处理, 没有打印, 这样会导致不必要的延迟。

old生成时是一个空文件, 所以watchwho第一次输出的内容是当前所有登录的用户清单。如果用命令who > \$old建立old的初值, 那么watchwho将只打印用户登录的变化情况, 这取决于习惯的不同。

另一个循环程序可以定时地查看信箱, 当信箱的内容变化时, 程序打印you have mail。shell的内部机制使用参数MAIL, 我们的实现方法使用shell变量而不使用文件, 要了解其差别请做实验。

```
$ cat checkmail
# checkmail: watch mailbox for growth

PATH=/bin: /usr/bin
MAIL=/usr/spool/mail/`getname` # system dependent

t=${1-60}

x=`ls -l $MAIL`
while :
do
    y=`ls -l $MAIL`
    echo $x $y
    x="$y"
    sleep $t
done | awk '$4<$12 { print "you have mail" }'
$
```

这里再次用到了awk程序, 确保当mailbox变大而不仅仅是改变时能及时将信息打印出来。否则, 只能在删除信件时接收到信息。(shell的内部版本就受到这一缺点的限制。)

时间间隔仍然设为60秒。但在命令行提供了一个参数, 如果键入下面的命令

```
$ checkmail 30
```

就使用新的时间参数。如果提供了时间参数, shell变量t被设为对应的时间间隔; 如果未提供时间参数, 缺省的时间间隔为60秒, 由下面这一行确定:

```
t=${1-60}
```

这是shell的另一个特点。

\${var}等价于\$var, 它可以用来避免变量名与字母或数字组成的字符串相混杂所引起的问题:

```
$ var=hello
$ varx=goodbye
$ echo $var
hello
```

```
$ echo $varx
goodbye
$ echo ${var}x
hellox
$
```

花括号内的某些特殊字符可以指定对变量进行特殊处理。如果该变量没有定义，并且在变量名之后带有一个问号，则打印问号后面的字符串，然后退出shell(除非是交互情况)。如果没有提供问号后的信息，则打印标准的输出信息：

```
$ echo ${var?}
hello
$ echo ${junk?}
junk: parameter not set
$ echo ${junk? error!}
junk: error!
```

变量var已被设置为hello
缺省信息
提供的信息

注意，shell产生的标准输出信息中总是包括未定义变量的名字。

另一种形式是变量\${var=thing}。当var有定义时，其值为\$var；当var没有定义时，其值为thing。\${var=thing}的情形与其相仿，但会把\$var的值置为thing：

```
$ echo ${junk-'Hi there'}
Hi there
$ echo ${junk?}
junk; parameter not set
$ echo ${junk='Hi there'}
Hi there
$ echo ${junk?}
Hi there
$
```

junk没有受到影响
将junk置为Hi there

变量取值规则在表5-3中列出。

回到我们原来的例子，

```
t=${1-60}
```

把t设为\$1。如果没有提供参数，t设为60。

表5-3 shell变量的赋值

变 量	赋 值
\$var	var的值；若var无定义则无值
\${var}	同上；用于变量后面跟着字母数字串的情况
\${var=thing}	var有定义时，取值var；否则取值thing，而\$var不改变
\${var=thing}	var有定义时，取值var；否则取值thing，\$var的值设为thing
\${var? message}	\$var有定义时，取值\$var；否则打印 message并退出shell。 如果message为空字符串，打印：var: parameter not set
\${var+thing}	\$var有定义时，取值thing；否则无值

练习5-9 查看目录/bin或/usr/bin中命令true和false的实现过程。(怎样才能找出它们?)

练习5-10 修改watchfor，使之能支持多个参数，查看多个用户的信息，而不需要用户输入joe | mary。

练习5-11 写一个watchwho的版本，用comm代替awk，比较新旧数据。你喜欢哪个版本?

练习5-12 写一个watchwho的版本，用shell变量存放who的输出，而不用文件存放，你喜欢哪一个版本？哪个版本运行得更快？watchwho及checkmail自动做&吗？

练习5-13 shell的两种情况：不做任何工作的命令和注解字符#有何不同？两者都需要吗？

5.4 trap: 捕获中断

如果在运行watchwho时突然按下DEL键或挂断电话，在目录/tmp中，将保存一个或两个临时文件。watchwho应该在退出之前清除这些暂存文件。我们需要一定的手段来检测各种中断事件，并进行恢复处理。

按DEL键时，一个中断信号会送给终端上正在运行的所有进程；同样地，当挂断电话线时，会传送一个挂断信号。其他信号发生的情形亦同。除非程序有专门处理中断信号的措施，否则，中断信号将一律终止程序的运行。如果是中断信号，后台运行的进程(使用&运行)能得到保护，但如果是挂断信号，则得不到保护。

第7章将详细讨论各种中断信号，但无须知道在shell中如何处理中断信号。shell的内部命令trap能生成中断信号发生时所要执行的命令序列。

trap 命令序列 信号值

命令序列是一个单个参数，一般要用引号括起来。信号值是小整数，用于标识中断信号。例如，2表示DEL键产生的信号，1表示电话挂断信号。shell程序使用的大多数通用信号值列在表5-4中。

表5-4 shell信号值

信 号 值	含 义
0	shell退出(各种正常退出，包括文件结束)
1	挂断
2	中断(DEL键)
3	退出(ctl-\; 导致突然撤出内核)
9	强迫终止(不能捕获或不予执行)
15	终止，由kill(1)产生的缺省信号

为了清除暂存文件，watchwho程序要在循环之前加上调用trap的语句，以捕获挂断、中断和终止信号：

```
...
trap 'rm -f $new $old; exit 1' 1 2 15

while :
...
```

trap的第一个参数是一个命令序列，这个序列类似中断信号发生时立即调用的子程序。除非中断信号终止了它，命令结束后，程序返回到断点继续执行。因此，trap命令序列必须显式地调用exit，否则在中断之后将继续执行shell程序。实际上trap中的命令序列将被读取两次，一次是建立trap时，一次是trap被调用时。因此，命令序列最好用单引号来保护，这样变量仅在trap程序执行时才被赋值。在这个例子里没有关系，但在有的情况下，这种区别非常重要，后面我们将看到一个例子。另外，选项-f表示rm不进行询问。

有时，trap在交互上很有用，最常用的情况是保护程序不会由于电话断线引起的挂断信号

异常终止:

```
$ (trap ' 1; long-running-command) &
2134
$
```

空的命令序列意味着在进程及其子进程中“不响应中断”。括号的作用是使trap和命令一起在一个后台子shell中运行，如果没有括号，trap将同时作用到登录的shell和long-running command中的命令上。

命令nohup(1)是一个很短的shell程序，它能提供这种服务。下面是在第7版中，nohup的实现:

```
$ cat `which nohup`
trap "" 1 15
if test -t 2>&1
then
    echo "Sending output to 'nohup.out' "
    exec nice -5 $* >>nohup.out 2>&1
else
    exec nice -5 $* 2>&1
fi
$
```

test -t测试标准输出是否为终端，以决定输出是否需要保存。带nice运行的后台进程的优先级比交互程序的优先级低。(注意: nohup没有设置PATH。是否应该设置?)

使用exec主要是为了提高效率，没有exec，程序的运行也不会有问题。exec是一个shell内部命令，它用给定程序代替运行shell的进程，因此节省了一个shell进程，而一般这个shell需要等待程序运行结束。我们还可以在其他一些地方使用exec，例如在增强的cal程序结尾处调用/usr/bin/cal时可以使用exec。

附带提一下，信号9是一个不能捕获也不能忽略，而且必须执行的信号，它总是要终止程序，在shell环境下，可用如下方法发出信号9:

```
$ kill -9 进程标识符...
```

kill -9的选项不是缺省设置，因为按这样的方式来终止进程，进程在结束之前没有任何机会处理善后工作。

练习5-14 上述nohup版本把命令的标准错误输出和标准输出结合在一起，这样设计好吗？如果不好，怎样将它们分开呢？

练习5-15 查阅shell内部函数times，在你的.profile文件里增加一行命令，当退出系统时打印出所使用的CPU时间。

练习5-16 写一个程序，使之能寻找在/etc/passwd文件里下一个可用的用户标识符。如果有兴趣(并有访问权力)，将这个程序改写成一个命令，在系统里添加新用户。完成上述工作需要什么访问权限？应该如何处理中断？

5.5 overwrite: 改写文件

sort排序命令有一个选项-o，表示覆盖文件:

```
$ sort file1 -o file2
```

它等价于下面这个命令

```
$ sort file1 >file2
```

如果file1和file2是同一个文件，重定向符号 >在排序之前就已经把输入文件截断。然而，加上-o选项的命令却能正常工作，因为在输出文件建立前，sort先将输入排序并存放在一个临时文件中。

很多其他命令也可用-o选项。例如，sed可以编辑文件如下：

```
$ sed 's/UNIX/UNIX(TM)/g' ch2 -o ch2
```

不能这样做！

要对所有这样的命令都加上选项-o显然是不现实的。另外，这种做法也不可取：最好是将功能集中起来处理，就像shell使用运算符>那样。我们给出一个程序overwrite完成这项工作。第一种设计如下：

```
$ sed 's/UNIX/UNIX(TM)/g' ch2 / overwrite ch2
```

基本的实现思路直截了当——保存全部输入直到文件尾，然后把数据复制到参数文件：

```
# overwrite: copy standard input to output after EOF
# version 1. BUG here

PATH=/bin:/usr/bin

case $# in
1)      ;;
*)      echo 'Usage: overwrite file' 1>&2; exit 2
esac

new=/tmp/overwr.$$
trap 'rm -f $new; exit 1' 1 2 15

cat >$new                # collect the input
cp $new $1                # overwrite the input file
rm -f $new
```

这里使用cp命令，而不使用mv命令，从而当输出文件已存在时，访问权限和所属关系保持不变。

尽管这个版本有个诱人的优点：简单，但是它有一个致命的缺点：如果用户在cp过程中按下DEL键，原来的输入文件将被破坏。必须防止对输入文件的覆盖被任何中断终止。

```
# overwrite: copy standard input to output after EOF
# version 2. BUG here too
```

```
PATH=/bin:/usr/bin

case $# in
1)      ;;
*)      echo 'Usage: overwrite file' 1>&2; exit 2
esac

new=/tmp/overwr1.$$
old=/tmp/overwr2.$$
trap 'rm -f $new $old; exit 1' 1 2 15
```

```

cat >$new                # collect the input
cp $1 $old                # save original file

trap ' ' 1 2 15          # we are committed; ignore signals
cp $new $1                # overwrite the input file

rm -f $new $old

```

如果到达原文件末尾前发生了一个DEL中断，临时文件将被删除而原文件保留。备份完成后信号将被忽略，因此前一个cp不会被中断——一旦cp开始，overwrite将全力进行原文件的修改工作。

还有一个小问题。考虑下面的程序：

```

$ sed 's/UNIX/UNIX(TM)g'precious / overwrite precious
command garbled: s/UNIX/UNIX(TM)g
$ls -l precious
-rw-rw-rw- 1 you          0 Oct 1 09:02 precious    #$$@*!
$

```

如果为overwrite提供输入的程序出错导致输出为空，overwrite仍会尽职尽责地破坏参数文件。

这一问题有几种解决办法：overwrite可以在重写文件前请求确认，但是增加交互功能会大大削弱overwrite的优越性；overwrite可以检查其输入是否为空(利用test -z),但这种编程风格很糟糕，而且也不正确，因为一些输出数据会在探测到错误之前生成。

最好的解决方案是由overwrite运行并控制数据生成程序，以便检查它的退出状态。尽管这与传统和直觉相左——在一个管道中，overwrite通常放在最后，但是为了正确工作，这里它必须最先运行。所幸的是，overwrite不产生标准输出，所以这种处理没有丧失一般性。它的语法也并不陌生：time、nice和nohup都是以其他命令作为参数的命令。

下面是这个安全的版本：

```

# overwrite: copy standard input to output after EOF
# final version

opath=$PATH
PATH=/bin:/usr/bin

case $# in
0 | 1)  echo 'Usage: overwrite file cmd [args]' 1>&2; exit 2
esac

file=$1; shift
new=/tmp/overwr1.$$; old=/tmp/overwr2.$$
trap 'rm -f $new $old; exit 1' 1 2 15    # clean up files

if PATH=$opath "$@" >$new                # collect input
then
    cp $file $old                        # save original file
    trap " 1 2 15"                       # we are committed; ignore signals
    cp $new $file
else
    echo "overwrite:$1 tailed, $file unchanged" 1>&2
    exit 1

```

```
fi
rm -f new $old
```

shell的内部命令shift将整个参数表向左移动一个位置：\$2成为\$1，\$3成为\$2，依次类推。\$@提供所有参数(移动后的)，与\$*类似，但不对参数作解释。在第5.7节还要讨论这一问题。

注意，为运行用户命令而对PATH进行了重置，否则，overwrite将不能访问那些不在/bin和/usr/bin目录下的命令。

现在overwrite可以工作了(尽管有些笨拙)：

```
$ cat notice
UNIX is a Trademark of Bell Laboratories
$ overwrite notice sed 's /UNIXUNIX(TM)/g' notice
command garbled: s/UNIXUNIX(TM)/g
overwrite: sed failed, notice unchanged
$ cat notice
UNIX is a Trademark of Bell Laboratories          没有改变
$ overwrite notice sed 's/UNIX/UNIX(TM)/g' notice
$ cat notice
UNIX(TM) is a Trademark of Bell Laboratories
$
```

sed命令经常用于词的替换。利用现有的overwrite，很容易写出一个shell程序实现替换功能：

```
$ cat replace
# replace: replace str1 in files with str2, in place

PATH=/bin:/usr/bin

case $# in
0 | 1 | 2) echo 'Usage: replace str1 str2 files' 1>&2; exit 1
esac

left="$1"; right="$2"; shift; shift

for i
do
    overwrite $i sed "s@$left@$right@g" $i
done
$ cat footnote
UNIX is not an acronym
$ replace UNIX Unix footnote
$ cat footnote
Unix is not an acronym
$
```

(回忆当for语句循环表为空时，缺省地等于\$*。)使用@代替/表示定界符，@与输入字符串冲突的可能性更小。

replace把PATH设为/bin:/usr/bin，而不包括\$HOME/bin。这表明overwrite必须在/usr/bin目录运行下才能使replace工作。做这一假定是为了处理简化。如果不能在/usr/bin里安装overwrite，就必须在replace里将\$HOME/bin放入PATH，或者明确地写出overwrite的路径名。

以后，我们假定所有命令都被有意地放在/usr/bin目录下。

练习5-17 为什么overwrite在trap中不用信号值0，从而在退出时删除临时文件？提示：试着在运行下列程序时按DEL键：

```
trap "echo exiting; exit 1" 0 2
sleep 10
```

练习5-18 为replace命令加上选项-v，用于在/dev/tty上打印所有变动的行。提示：s/\$left/ \$right/g\$flag。

练习5-19 修改replace以忽略替换串中的任何字符。

练习5-20 replace能否把程序中出现的所有变量i改为index？应该怎样修改？

练习5-21 把replace程序放在/usr/bin目录是否方便和完全满足使用需要？需要时简单地键入正确的sed命令是否更加可取？为什么？

练习5-22 (难)下一命令不能正常工作。解释原因并修改。

```
$ overwrite file 'who / sort'
```

提示：参阅sh(1)中的eval。修改后程序对命令中元字符的解释产生了什么影响？

5.6 zap: 使用名字终止进程

kill命令只能通过指定进程标识符来终止进程。要终止某个后台进程时，一般要运行ps命令以得到进程标识符，然后再把它作为kill的参数输入。通过一个命令程序打印一个参数，再把这个参数手工输入到另一个命令中，这个方法似乎有些笨拙。为什么不写一个程序，如zap，自动完成这些工作呢？

原因之一是终止进程是个危险的操作，执行时必须小心谨慎。一个保险的办法是交互地运行zap，用pick命令选择要终止的进程。

先简要回顾一下pick的功能：pick顺次打印它的每个参数，并请求用户响应，只有用户响应为y时才打印参数。(pick是下一节讨论的主题。)zap使用pick来确认那些通过名字选择的进程确实为用户所要终止的进程。

```
$ cat zap
# zap pattern: kill all processes matching pattern
# BUG in this version

PATH=/bin:/usr/bin

case $# in
0)      echo 'Usage: zap pattern' 1>&2; exit 1
esac

kill `pick \`ps -ag | grep "$*" | awk '{print $1}'`
$
```

注意经反斜杠转义的嵌套的引号。awk程序从pick对ps输出的选择结果中选取进程标识符：

```
$ sleep 1000 &
22126
$ ps -ag
PID TTY TIME CMD
...
```

```
22126 0 0:00 sleep 1000
```

```
...
```

```
$ zap sleep
```

```
22126?
```

```
0? q
```

发生了什么?

```
$
```

问题是ps输出的是一些分隔开来的单词，pick将它们视作分离的参数处理，而不是一次处理一行。shell的标准处理是以空格为分割符分裂字符串的，如：

```
for i in 1 2 3 4 5
```

在该程序里，我们要控制shell对字符串的分裂，只用换行分隔两个相邻的“词”。

shell变量IFS(内部字段分隔符)是一个字符串，串中是那些用于在参数列表和for语句中分割词的字母。通常，IFS包括空格、Tab和换行，我们也可以把它改成任何有用的形式，诸如只有一个换行：

```
$ echo `echo $#` >nargs
```

```
$ cx nargs
```

```
$ who
```

```
you      tty0      Oct    1 05:59
```

```
pjw      tty2      Oct    1 11:26
```

```
$ nargs `who`
```

```
10
```

十个空格和换行分隔的字段

```
$ IFS='
```

```
'
```

只是换行

```
$ nargs `who`
```

```
2
```

两行，两个字段

```
$
```

把IFS设置为换行，zap就能很好地工作：

```
$ cat zap
```

```
# zap pat: kill all processes matching pat
```

```
# final version
```

```
PATH=/bin:/usr/bin
```

```
IFS='
```

```
'
```

just a newline

```
case $1 in
```

```
"") echo `Usage: zap [-2] pattern` 1>&2; exit 1 ;;
```

```
*) SIG=$1; shift
```

```
esac
```

```
echo `PID TTY TIME CMD`
```

```
kill $SIG `pick \`ps -ag | egrep "$*"` | awk '{print $1}`
```

```
$ ps -ag
```

```
PID TTY TIME CMD
```

```
...
```

```
22126 0 0:00 sleep 1000
```

```
...
```

```
$ zap sleep
```

```
PID TTY TIME CMD
```

```
22126 0 0:00 sleep 1000? y
```

```
23104      0 0:02 egrep sleep? n
$
```

这里我们又使用了两个shell编程技巧：首先是通过一个可选参数来指定信号(注意如果不给出参数值，则SIG无定义，并作为空字符串处理)，其次是用egrep代替grep，以便接受更复杂的模式，如‘lsleep | date’。echo语句用于打印ps输出的标题栏。

你也许会想到，为什么这个命令称作zap而不直接称作kill。主要原因是，与cal不同，我们实际上没有提供一个新的kill命令：zap需要交互地操作，此其一；而且我们还想保留原来真正的kill程序。同时zap的执行速度较慢，所以附加程序的开销可想而知，其中ps(它是必须运行的)是开销最大的。我们将在下一章提供一个更有效的实现方法。

练习5-23 修改zap，使它由管道打印ps标题，而不必理会ps输出格式的改变。这将在多大程度上使程序复杂化？

5.7 pick命令：空格和参数

我们已经接触了shell中书写pick命令需要的多数命令。我们只需要一种新的机制来接收用户输入。shell内部命令read提供了这一功能，即从标准输入读一行文本，并把读到的文本(不含换行)赋给命名变量：

```
$ read greeting
hello, world          为greeting输入新参数
$ echo $greeting
hello, world
$
```

read最常用于注册时在.profile文件里设置环境，主要是建立shell环境变量如TERM。

read只能读取标准输入，而且不能被重定向。shell内部命令(与控制流原语，如for，相反)都不能使用>或<重定向：

```
$ read greeting </etc/passwd
goodbye              必须输入一值
illegal io           shell报告出错
$ echo $greeting
goodbye              greeting有一个不是来自文件的输入值
$
```

这也许可以说是shell的一个缺陷，但这就是现实。幸运的是，重定向read外围的循环可以达到同样的目的。这是pick命令实现的关键：

```
# pick: select arguments

PATH=/bin:/usr/bin

for i                # for each argument
do
    echo -n "$i? " >/dev/tty
    read response
    case $response in
    y*)      echo $i ;;
    q*)      break
    esac
```

```
done </dev/tty
```

echo的-n选项抑制最后一个换行，因此用户的响应可以打印在提示符的同一行。当然，提示符打印在/dev/tty上，因为这里的标准输出大多数情形下并不是终端。

break语句取自C语言，表示终止内部循环。上例中，输入q键时终止for循环。q为quit的缩写，易用，这一退出响应几乎已成为惯例，其他很多程序也采用q。

pick命令行参数里的空格是很有趣的：

```
$ pick '1 2' 3
1 2?
3?
$
```

要观察pick程序如何读取参数，运行pick并在每次提示符出现后按Return键。程序工作情况很好：for i适当地处理参数。我们还可以用其他方法写循环：

```
$ grep for pick          看看这个版本做什么工作
for i in $*
$ pick '1 2' 3
1?
2?
3?
$
```

工作异常，因为循环中的操作数被重复扫描，这样第一个参数中的空格将它变成了两个参数。试用引号括住参数\$*：

```
$ grep for pick          试验一个不同的版本
for i in "$*"
$ pick '1 2' 3
1 2 3?
$
```

还是不能正常工作，因为"\$*"是由空格分隔的所有参数连在一起构成的单个词。

可能的解决办法是，shell对字符串"\$@"作特殊处理，并将它转换为严格的shell文件参数：

```
$ grep for pick          试验第三个版本
for i in "$@"
$ pick '1 2' 3
1 2?
3?
$
```

如果不用引号括住\$@，它与\$*相同，它只有在用双引号括住时才具有特殊性质。我们在overwrite程序中用它来保护用户的命令参数。

上述内容可以归纳为如下规则：

- S*和\$@扩展为参数，并被重复扫描；参数的空格将字符串分成多个参数。
- "\$*"表示shell文件的所有参数及其空格连在一起作为单个词处理。
- "\$@"与shell文件接收的参数等价，参数中的空格被忽略，其结果是等同于原来参数的一个单词列表。

如果pick没有参数，可以读取标准输入，所以我们可以使用命令

```
$ pick <mailinglist
```

来代替

```
$ pick `cat mailinglist`
```

这里我们不研究pick的这个版本：它的复杂度和难以理解的程度远远超过下一章将要介绍的用C语言书写的程序。

下面练习中的前两个题目比较难，属于高级shell程序员教程的内容。

练习5-24 写一个pick程序，当命令行中没有参数支持时它从标准输入读取参数。程序应能正确处理空格。程序能否处理q响应？如果不能，试做下一个练习。

练习5-25 虽然shell的内部命令，如read和sed不能重定向，但shell本身可以暂时地重定向。阅读sh(1)节中对exec命令的描述，考虑如果不调用子shell如何将read 定向到/dev/tty。（可以先参阅第7章。）

练习5-26 （比较容易）在.profile文件里使用read初始化TERM和其他相关参数，如Tab停止位。

5.8 news命令：社团服务信息

我们在第1章曾提到你的系统中可以有一个news命令用以报告用户社团方面的信息。大部分系统都提供新闻服务，尽管命令名和命令细节有所不同。这里给出一个news命令，不是要代替原有的本地命令，而旨在说明在shell里编写这样一个程序是多么的容易。比较两个news命令的实现会是件有趣的事情。

程序利用这样一个基本事实，即各个新闻项目是分开存放的，每个文件包含一个新闻项目，存放在特定的目录下，譬如/usr/news。news(我们自己的news程序)将/usr/news目录里的文件和你的注册目录(.news_time)中下一个作为时间戳的文件进行比较。为便于调试，我们用.代替新闻文件和.news_time两者的目录，在使用时，可以方便地修改为/usr/news目录。

```
$ cat news
# new: print news files, version 1

HOME=.          # debugging only
cd .             # place holder for /usr/news
for i in `ls -t * $HOME/.news_time`
do
    case $i in
        */.news_time) break;;
        *)             echo news: $i
    esac
done
touch $HOME/.news_time
$ touch .news_time
$ touch x
$ touch y
$ news
news:y
news:x
$
```

touch的功能是把参数文件的最后修改时间改为当前时间，而不改动文件本身。为调试方

便，我们只打印新闻文件的名称，而不打印文件内容。循环遇到`news_time`便终止，因此所列文件都是比`news_time`新的文件。注意，`case`语句中的符号`*`可以与`/`匹配，而`/`是不会出现在文件名模式中的。

如果`news_time`不存在，会出现什么情况？

```
$ rm .news_time
$ news
$
```

毫无响应，说明程序有错。这一错误的发生是因为如果`ls`命令找不到文件，它会在打印现有文件的有关信息之前，在标准输出上给出错误信息。不可否认这是一个缺陷——诊断信息应该打印在标准错误输出上——不过我们可以识别循环中的问题，通过重定向标准错误输出到标准输出使所有版本同样地工作，从而避开这个问题。（这一问题在更新版本中已经得到适当处理，但是我们仍然将它保留下来，作为一个如何应付小麻烦的示例。）

```
$ cat news
# news: print news files, version 2

HOME=.          # debugging only
cd .             # place holder for /usr/news
IFS='
'               # just a newline
for i in `ls -t * $HOME/.news_time 2>$1`
do
    case $i in
        *' not found') ;;
        */.news_time) break ;;
        *)           echo news: $i ;;
    esac
done
touch $HOME/.news_time
$ rm .news_time
$ news
news: news
news:y
news:x
$
```

我们必须将`IFS`设置为换行，以免下面的信息被分为三个单独的词：

```
./news_time not found
```

`news`必须打印新闻文件，而不是仅仅打印文件名。发送者和发送时间往往是有用的信息，因此可以使用`set`命令和`ls -l`命令在前面打印标题：

```
$ ls -l news
-rwxrwxrwx 1 you          208 Oct 1 12:05 news
$ set `ls -l news`
-rwxrwxrwx: bad option(s)      出现错误
$
```

这是一个shell中程序和数据互相交换的例子。`set`不接收参数是因为参数`-rwxrwxrwx`以负号开始，与选项相混淆。一个简易可行的修改方法是在参数前加上一个字符作为前缀：

```
$ set X`ls -l news`
$ echo "news: ($3) $5 $6 $7"
news: (you) Oct 1 12:05
$
```

这是一个可行的格式，它在给出文件名的同时给出了作者和日期信息。

下面是news命令的最后版本：

```
# news: print news files, final version

PATH=/bin:/usr/bin
IFS=
# just a newLine

cd/usr/news

for i in `ls -t * $HOME/.new.time 2>&1`
do
    IFS=' '
    case $ i in
        *`not found`) ;;
        */.news_time) break ;;
        *)
            set X`ls -l $i`
            echo "
$i: ($3) $5 $6 $7
"
            cat $i
        esac
    done
touch $HOME/.news_time
```

标题中附加的换行将新闻项逐条分行打印。第一个IFS的值只有一个换行，因此第一个ls命令产生的信息not found(如果有的话)被作为单个参数处理。下一个赋值将IFS置为空格，所以第二个ls的输出被分成若干个参数。

练习5-27 在news中增加选项-n(notify)，用于报告新闻项而不打印，并且不改变.news_time。可以把这个新的news放在.profile文件中。

练习5-28 把上一练习的news的设计和实现与系统中类似的命令作一比较。

5.9 get和put: 追踪文件变动

这一节是本章的最后一节，我们将介绍一个更大更复杂的例子，以此说明shell如何与awk和sed进行合作。

程序在开发过程中不断地改正错误和增加新功能。版本追踪往往是不可少的。特别是，有的用户把程序移植到其他的机器上使用——他们常常跑回来问“在我们的版本之后又有哪些改动”，或者“这个那个错误是怎么改的”。保留备份也会使尝试新思想变得更加安全稳妥：如果新的程序不成功，利用备份很容易回到原来的程序。

保存所有版本是一种备份方法。然而这样做不易组织，同时占用磁盘空间太多。我们可以利用一个可能性，即相邻两版本间通常大部分是重复不变的，从而只要保存一次即可。用diff-e命令生成一个ed命令列表，可以把老版本转换成新版本：

```
$ diff -e old new
```

这样，通过维护一个完整的版本，以及把这个版本转换为任何其他版本的一组编辑命令，就可以把文件的所有版本都保留在一个独立的(不同的)文件中。

这有两种组织方式：一种是保留最新的完整版本和转换到老版本的编辑命令，另一种是保留最老的版本和转换到新版本的编辑命令。虽然后一种方式比较容易实现，但是当有很多版本时，前一种方式使用较快，因为我们一般总是对当前版本感兴趣。

我们选择第一种组织方式。在一个我们称之为“历史文件”的文件中，保存当前版本和多组编辑命令，每组编辑命令把某一版本转换为前一版本。各组编辑命令以下一行信息开始：

```
@@@ 个人 数据 摘要
```

摘要为单行，由个人描述文件的改动。

有两个维护版本的命令：get和put。get命令从历史文件里获得版本；put命令在得到一个单行的变动摘要后，把一个新的版本放入历史文件中。

在介绍具体实现之前，我们先举例说明get和put如何工作，以及历史文件是怎样维护的：

```
$ echo a line of text >junk
$ put junk
Summary: make a new file          输入摘要信息
get: no file junk.H              历史文件不存在
put: creating junk.H             ...所以put命令创建历史文件
$ cat junk.H
a line of text
@@@ you Sat Oct 1 13:31:03 EDT 1983 make a new file
$ echo another line>>junk
$ put junk
Summary: one line added
$ cat junk.H
a line of text
another line
@@@ you Sat Oct 1 13:32:28 EDT 1983 one line added
2d
@@@ you Sat Oct 1 13:31:03 EDT 1983 make a new file
$
```

其中“编辑命令”是由2d行命令组成的，表示删除文件的第二行，这样就可以将新版本转换为老版本。

```
$ rm junk
$ get junk                        最新的版本
$ cat junk
a line of text
another line
$ get -1 junk
$ cat junk                        最新版本之前的版本
a line of text
$ get junk                        又是最新的版本
$ replace another 'a different' junk  改变它
$ put junk
Summary: second line changed
$ cat junk.H
```

```

a line of text
a different line
@@@ you Sat Oct 1 13:34:07 EDT 1983 second line changed
2c
another line

@@@ you Sat Oct 1 13:32:28 EDT 1983 one line added
2d
@@@ you Sat Oct 1 13:31:03 EDT 1983 make a new file
$

```

编辑命令从上到下编辑历史文件，用来产生所要求的版本：第一组编辑命令把最新的版本转换为次新的版本，下一组命令把次新的版本换成第三新的版本，依此类推。因此，实际上每调用一次ed命令就把文件向前推进了一个版本。

如果被修改的文件包含行首为@@@的行，会引起一些麻烦，另外diff(1)命令的BUGS段将对只含一个句号的行给出警告。我们选择@@@作为编辑命令的标题头记号，因为标准文本一般不会用到这种序列。

讨论get和put命令的使用说明及命令的各种不同形式需要很多篇幅，所以我们只给出它们最后的形式。put命令比较简单，其程序为：

```

# put: install file into history

PATH=/bin:/usr/bin

case $# in
    1) HIST=$1.H ;;
    *) echo 'Usage: put file' 1>&2; exit 1 ;;
esac
if test ! -r $1
then
    echo "put: can't open $1" 1>&2
    exit 1
fi
trap 'rm -f /tmp/put.[ab]$$; exit 1' 1 2 15
echo -n 'Summary: '
read Summary

if get -o /tmp/put .a$$ $1                                # previous version
then                                                         # merge pieces
    cp $1 /tmp/put.b$$                                       # current version
    echo "@@@ `getname` `date` $Summary" >>/tmp/put.b$$
    diff -e $1 /tmp/put.a$$ >>/tmp/put.b$$                  # latest diffs
    sed -n '/^@@@/, $p' <$HIST >>/tmp/put.b$$               # old diffs
    overwrite $HIST cat/tmp/put.b$$                         # put it back
else                                                         # make a new one
    echo "put: creating $HIST"
    cp $1 $HIST
    echo "@@@ `getname` `date` $Summary" >>$HIST
fi
rm -f /tmp/put.[ab]$$

```

读完一行摘要之后，put调用get从历史文件中提取前一版本。get的-o选项表示给出一个不同的输出文件名。如果get找不到历史文件，它返回一个错误状态，而put将创建一个新的历史文件。如果历史文件已经存在，then子句以临时文件的形式创建一个新的历史文件，文件创建依次利用下列内容：最新版本、@@@行、新版本转换为次新版本的编辑命令，以及老的编辑命令和@@@行。最后，用overwrite把临时文件复制到历史文件。

get程序比put程序更复杂，主要是因为有一些选项。

```
# get: extract file from history

PATH=/bin:/usr/bin

VERSION=0
while test "$1" != ""
do
    case "$1" in
        -i) INPUT=$2; shift ;;
        -o) OUTPUT=$2; shift ;;
        -[0-9]) VERSION=$1 ;;
        -*) echo "get: Unknown argument $1" 1>&2; exit 1;;
        -*) case "$OUTPUT" in
                "") OUTPUT=$1 ;;
                *) INPUT=$1.H ;;
            esac
        esac
    shift
done
OUTPUT=$ {OUTPUT?"Usage: get[-o outfile] [-i file.H] file" }
INPUT=$ {INPUT-$OUTPUT.H}
test -r $INPUT || { echo "get: no file $INPUT" 1>&2; exit 1; }
trap 'rm -f /tmp/get.[ab]$$; exit 1' 1 2 15
# split into current version and editing commands
sed<$INPUT -n< '1, /^@@@/w /tmp/get.a'$$'
/^@@@/, $w /tmp/get.b'$$

# perform the edits
awk </tmp/get.b$$ '
    /^@@@/ { count++ }
    !/^@@@/ && count>0 && count <= - ' $VERSION'
    END { print "$d"; print "w", "'$OUTPUT'" }
' | ed -/tmp/get.a$$
rm -f /tmp/get.[ab]$$
```

这些选项很普通。-i和-o指定输入和输出文件名。-[0-9] 选择特定的版：0表示最新版本(缺省值)，-1表示次新版本，依次类推。循环是while语句，用test和shift实现对全部参数的循环，循环不用for语句，因为某些选择项(-i, -o)要使用其他参数，所以要用shift把对应参数移出，而位于for循环语句内部的shift语句不能与for很好地合作。ed的-选项关闭了在读写文件时通常使用的符号计数功能。

下面一程序

```
test -r $INPUT || { echo "get: no file $INPUT" 1>&2; exit 1; }
```

等价于

```

if test ! -r $ INPUT
then
    echo "get: no file $INPUT" 1>&2
    exit 1
fi

```

(这是我们在put中使用的形式), 但是更紧凑, 对于熟悉操作符||的程序员来说也更清楚。{}之间的命令表示在当前shell下执行, 而不是在子shell下执行, 这样就保证了exit是从get退出, 而不从子shell退出。字符{和}很像do和done——它们只在分号、换行或命令终止符之后才具有特殊作用。

最后, 我们介绍get程序的工作代码。首先, sed把历史文件分成两部分: 最新版本和编辑命令集合。然后由awk程序处理编辑命令。对@@@行进行计数(但不打印), 同时传递编辑命令, 直到计数值超过所期望的版本数(回忆一下, awk命令的缺省动作是打印输入行)。两个ed命令\$d和w附加在历史文件中的命令后: \$d删除sed留在当前版本里的@@@行, 而w命令完成文件的最后写入。这里不需要overwrite, 因为get程序只改变文件的版本, 而不改变先前的历史文件。

练习5-29 编写一个命令version, 完成下述两件工作:

```
$ version -5 file
```

报告摘要、修改日期和修改人, 对历史文件中选定版本进行修改。

```
$ version sep 20 file
```

报告9月20日的版本号。这被典型地用于下面命令中(version可以方便地回送历史文件名):

```
$ get 'version sep 20 file'
```

练习5-30 修改get和put, 使之可以操作其他目录下的历史文件, 而不至于因为.H文件把工作目录弄乱。

练习5-31 一旦工作完成, 不再需要保留所有的文件版本, 怎样实现从历史文件中删除一些版本呢?

5.10 小结

当需要编写一个新程序时, 自然立刻会想到如何用你最喜欢的语言来编写这个程序。对我们来说, 最常用的语言是shell。

shell是一种很好的编程语言, 虽然它的语法有些特殊。shell属于高级语言, 它的操作对象为整个程序。由于shell是交互式语言, 所以shell程序能够交互式地开发, 可以逐级求精直至它能够令人满意地工作。如果要使其面向更多的用户, 可以对shell程序进一步改造, 使之更精巧和更实用, 以满足广泛使用的需要。不能用shell程序高效地解决问题的情况微乎其微。如果遇到这种例外情况, 在手中拥有可靠的设计和工作实现的情况下可以用C语言编写全部或部分程序(我们在下一章将会遵循这种方法)。

UNIX环境下编程有其特有原则——将工作建筑在已有成果之上, 而不是白手起家; 从小做起, 逐步发展; 使用各种工具验证新的想法。

我们在本章列举了很多例子, 用现有的程序和shell很容易实现它们。有时, 只需要重新设置参数就可以工作, 如cal程序。有时, shell在一组文件名或一个命令执行序列上执行循环,

如watchfor程序和checkmail程序。比较复杂的例子，用shell编程比用C语言编程要简洁得多，例如，我们用20行的shell程序news版本完全可以代替350行的C语言程序[sic]版本。

有了一个可编程的命令语言和许多实用的程序还不够，重要的是所有这些组件能够协同工作！它们采用相同的信息表示和通信格式。每一个组件都出色地完成自己的份内工作。shell则根据你的意愿将它们简单高效地连接起来。这种合作能力正是UNIX程序设计环境如此富有成效的关键所在。

有关文献

get和put来源于Source Code Control System(SCCS)，这最早是由Marc Rochkind提出来的(“The source code control system”，IEEE Trans. on Software Engineering,1975年)。SCCS是一个比我们上面这些简单的程序要强大和灵活得多的系统，它的目标是在工业环境中维护大型程序。然而，SCCS的基础是同一个diff程序。

第6章 使用标准I/O编程

到目前为止，我们已学会用已有的工具来创建新的工具，但这样做往往受到shell、sed和awk的限制。在本章中，我们将用C语言写一些简单的程序。把完成不同功能的部件组合在一起仍将是本书进行以下讨论和编程的主要思想，它将指导我们编出一些工具而为他人所用。本章的每个例子将按如下的方式展开：从最基本的开始，随着需求的增加而逐步添加功能和选项。

很多时候，我们不得不从头写一个新程序，因为很可能没有现成的程序来解决某个现有问题，例如，当程序要处理非文本文件时(这类情况时常发生)。而到目前为止我们所介绍的大多数程序都是对文本信息进行操作的。有时仅用shell和其他一般工具也很难获得足够的健壮性或高效性，因此也需重新编写新程序。在这种情况下，一个shell版本也许适于定义变量和用户接口(如果它工作得很好，就没有必要重新编写)。上一章所介绍的zap程序就是一个很好的例子：编写其shell版本只需很少的时间，而最终版本则有一个完善的用户接口，但运行速度太慢。

我们采用C语言是因为它是UNIX系统的标准语言——内核以及所有的用户程序都是由C编写的，而且其他语言都没有像它一样被如此广泛地支持。我们假定读者了解C，至少能读懂程序。如果不是这样，请参阅The C Programming Language(B.W.Kernighan, D.M.Ritchie, Prentice-hall, 1978年)。

我们还将使用标准I/O库。它包含了一系列用于I/O和系统服务的C语言例程。在许多支持C语言的非UNIX系统中也带有标准I/O库，因此用它编写的程序很容易移植。

本章中的示例有一个共同点：它们都是常用的小工具，但并不是第7版UNIX的一部分。如果你的系统中有类似的程序，对它们进行比较将很有帮助。如果以前没有使用过类似的工具，将会发现它们很有用。总之，没有一个系统是完美的，但通常可以通过一定的努力改善它。

6.1 vis: 标准I/O

许多程序只进行一次读操作和一次写操作，对这类程序而言，使用标准I/O就可以了，同时，从标准I/O入手也便于我们学习。

下面给出一段名为vis的程序，它的功能是将其标准输入拷贝到标准输出，其中无法打印的字符以\nnn(nnn为字符的八进制代码)代替。因此，vis无法显示文件中的怪字符，例如，它将Backspace键显示为\010(010为Backspace键的八进制代码)：

```
$ cat x
abc
$ vis <x
abc\010\010\010___
$
```

为了使vis一次读入多个文件，可用cat命令将这些文件收集到一起：

```
$ cat file1 file2 ... | vis
...
$ cat file1 file2 ... | vis | grep '\\'
```

这种做法避免了学习在程序中如何使用文件。

另外，为实现以上vis程序的功能，也可使用sed命令。sed中的 l 子命令可将不可打印字符以一种可以理解的方式显示出来：

```
$ sed -n 1 x
abc<<<_--
$
```

sed命令的输出此时会显得比vis程序的输出更易懂，但sed的缺点在于它不能用于非文本文件：

```
$ sed -n 1 /usr/you/bin
$
```

没有任何显示

(以上显示结果是基于PDP-11机器的。在VAX系统中，sed的运行将会异常终止。可能是因为这里的输入比较类似于一个长的字符串。)因此，sed命令不能满足要求，必须写一个新的程序。

最简单的I/O例程名为getchar和putchar。每次调用getchar时将从标准输入(如文件、管道或终端，程序对此并不加以区别)中读取下一个字符。同样，putchar(c)将字符c送入标准输出，缺省为终端。

函数printf(3)用于格式化输出。对printf和putchar的调用可按任意顺序进行。与printf相对应的格式化输入函数是scanf(3)，它将标准输入按指定方式分为多个字符串和数字等。同样，对scanf和getchar的调用也可按任意顺序进行。

下面给出程序vis的第一个版本：

```
/* vis:  make funny characters visible (version 1) */

#include <stdio.h>
#include <ctype.h>

main()
{
    int c;

    while ((c = getchar()) != EOF)
        if (isascii(c) &&
            (isprint(c) || c=='\n' || c=='\t' || c==' '))
            putchar(c);
        else
            printf("\\%03o", c);
    exit(0);
}
```

此例中，getchar返回输入的下一字符，或是在遇到文件结束或错误时返回EOF。EOF不是文件中的一个字节，相关说明请参阅第2章。EOF的值与其他任何字节的值都不同，因此，可将其正确地辨认出来。变量c被声明为int而非char类型，因此可用它来保存EOF的值。

程序行

#include <stdio.h>必须在每个源程序的开头出现。它的作用是使C的编译器读取有关标准例程和标识符(包括EOF)的头文件 /usr/include/stdio.h，编程时可用<stdio.h>代替。

文件<ctype.h>是/usr/include中的另一个头文件，它定义了用来决定字符属性的测试，该测试与设备无关。本例中使用了isascii和isprint来检查输入字符是否是ASCII字符(值小于0200)

以及是否可打印。其他测试如表6-1中所示。在<ctype.h>中, 换行、Tab和空格是不可打印的。

表6-1 <ctype.h>字符测试宏

名 称	含 义
isalpha(c)	字母:a-z A-Z
issupper(c)	大写: A-Z
islower(c)	小写: a-z
isdigit(c)	数字: 0-9
isxdigit(c)	十六进制数字: 0-9 a-f A-F
isalnum(c)	字母或数字
isspace(c)	空格、Tab、换行、垂直Tab、换页、回车
ispunct(c)	非字母或控制字符或空白
isprint(c)	可打印字符: 任何图形符
isctrl(c)	控制字符: $0 \leq c < 040$ $c == 0177$
isascii(c)	ASCII字符: $0 \leq c \leq 0177$

在程序末尾对exit的调用对程序的正确执行来说并不是必须的, 但它能确保vis的调用者得到一个正常的退出状态(一般为0)。另一种解决方式是让主程序return 0, 因为主程序的返回值就是程序的退出状态。如果没有exit或return, 那么退出状态是无法预测的。

为编译一个C程序, 应先将源代码保存在.c文件中, 比如vis.c, 然后用cc编译, 得到可执行文件a.out(a 表示汇编程序assembler):

```
$ cc vis.c
$ a.out
hello world ctl-g
hello world\007
ctl-d
$
```

一般情况下, 当程序可正确运行后, 应将a.out改名或直接使用cc的选项-o:

```
$ cc -o vis vis.c
```

输出文件是vis, 而非a.out

练习6-1 在vis中, Tab被当作特殊字符处理, 而不是将其显示为\011或\t。另一种处理方式是将每个字符都平等看待, 请将vis进行必要的修改, 以便它能将Tab、Backspace和空格等显示为C语言中的正常表示方式(\t、\b等)。并将你的设计与如下语句相比较:

```
$ sed -n 1
```

练习6-2 对vis进行修改, 使其具有将长句自动折行的功能。思考如何按这种格式处理练习6-1的输出。

6.2 vis第2版: 程序参数

当C程序执行时, 命令行参数可以以字符串数组argv和数组长度argc的形式传递给主程序。其中, argv[0]代表命令名, 因此, argc总是大于0。对程序有用的参数储存于argv[1]到argv[argc-1]中。我们已讲过, 用<和>进行重定向是由shell而非各个程序管理的, 因此对程序来说, 重定向对参数个数没有影响。

为说明如何对程序参数进行处理, 现在对vis进行修改, 加上一个可选的参数-s: 它的功能是跳过不可打印字符。这个参数对处理其他系统(如用回车作为一行结尾的系统)的文件很有用。

```

/* vis:  make funny characters visible (version 2) */

#include <stdio.h>
#include <ctype.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    int c, strip = 0;

    if (argc > 1 && strcmp(argv[1], "-s") == 0)
        strip = 1;
    while ((c = getchar()) != EOF)
        if (isascii(c) &&
            (isprint(c) || c=='\n' || c=='\t' || c==' '))
            putchar(c);
        else if (!strip)
            printf("\\%03o", c);
    exit(0);
}

```

argv是指向一个数组的指针，此数组的每一项又是一个指向某一字符数组的指针。这些字符数组以ASCII码NUL(‘\0’)结尾，因此可看作字符串。修改后的vis先检查程序是否有参数以及此参数是否为-s。无效的参数将被忽略。函数strcmp(3)用来比较两个字符串，若相同则返回0。

表6-2列出了对字符串进行处理的函数，strcmp是其中之一。一般最好使用这些已有的函数，因为它们是正确的并且是针对特定机器作了优化的，所以运行速度一般比自己编的函数要快。

表6-2 标准字符串函数

函 数	含 义
strcat(s,t)	把字符串t连接到字符串s的后面，返回s
strncat(s,t,n)	把字符串t的前n个字符连接到字符串s的后面，返回s
strcpy(s,t)	将字符串t拷贝到s中，返回s
strncpy(s,t,n)	将字符串t的前n个字符拷贝到s中，返回s
strcmp(s,t)	比较s和t,根据是小于、等于或大于返回<0、0、>0
strncmp(s,t,n)	至多比较n个字符
strlen(s)	返回s的长度
strchr(s,c)	返回字符串中第一个指向字符c的指针，否则返回NULL
strrchr(s,c)	返回字符串中最后一个指向字符c的指针，否则返回NULL
atoi(s)	返回s的整数值
atof(s)	返回s的浮点值；需声明double atof()
malloc(n)	返回一个指向n字节内存的指针，如分配失败，则返回NULL
calloc(n,m)	返回一个指向n*m字节的指针，并设置为0，如果分配失败，则返回NULL
free(p)	释放由malloc或calloc分配的内存

练习6-3 修改vis的-s参数，使得vis -sn的运行结果是只显示那些包含长度大于等于n的连续可打印字符的字符串。利用这一功能可将非文本文件(如可执行文件)中的文本部分提取出来。

某些版本的系统提供strings程序实现相同功能。请考虑：是用一个独立的程序来实现这项功能好，还是仅修改vis的参数好？

练习6-4 UNIX系统的优点之一就是可获取C语言写的源代码，从这些代码中可学到很多好的解决问题的方法，但同时，系统的某些部分是用汇编语言进行优化的。你如何看待C语言的可读性与汇编语言的高效率之间的矛盾呢？

6.3 vis第3版：访问文件

前两个版本的vis都是从标准输入读入后再写进标准输出，这些功能是由shell控制的。下面将对vis进行修改，使其能通过文件名来读取文件。如下语句：

```
$ vis file1 file2 . . .
```

将从相应文件读入信息，而非标准输入。但是，如果vis语句中不包括文件名参数，则它仍从标准输入中读数据。

现在的问题是如何读文件，也就是说，如何将文件名与真正执行读数据功能的I/O过程联系起来。

解决此问题的操作规则比较简单。在读写一个文件之前，必须先用库函数fopen将其打开。fopen得到一个文件名(如temp或/etc/passwd)后，将与系统内核进行通讯，以得到此文件的内部名称，并在以后的操作中一直使用此名称。

这个内部名称实际上是一个指针，称作文件指针。它指向一个含有此文件相关信息的结构，这些信息包括缓冲区的地址、缓冲区中字符的当前位置以及文件是否正在被读或写等。在<stdio.h>中定义了一个称为FILE的结构。定义一个文件指针可用如下语句：

```
FILE *fp;
```

它表示fp是一个指向FILE结构的指针。函数fopen的返回值就是一个此类型的指针。在<stdio.h>中有对fopen返回类型的定义。

在程序中，对fopen的调用形式为：

```
char *name, *mode;
```

```
fp = fopen(name, mode);
```

fopen的第一个参数是文件名字符串，第二个参数也是一字符串，它代表将要对文件进行的操作：r代表读；w代表写；a代表添加。

若当前被打开的用于写操作或添加操作的文件并不存在，则它将作为一个新文件被创建。打开一个已存在的文件用于写操作将删除此文件已有内容。读一个不存在的文件或读写一个未经授权的文件都将发生错误。发生错误时，fopen将返回NULL值(在<stdio.h>中定义为(char *)0)。

文件被打开后，就可按某种方式进行读写操作，其中最简单的就是使用getc和putc函数。getc函数的功能是从文件中读取下一字符：

```
c = getc(fp)
```

fp是指向某一文件的指针。当读到文件末尾时，返回EOF。putc的功能与getc相对应，它将一字符写入文件并返回此字符：

```
putc(c, fp)
```

当错误发生时，getc和putc都返回EOF。

当程序开始执行时，三个文件及相应的文件指针被打开。这三个文件是：标准输入、标准输出和标准错误输出。相应的文件指针是stdin、stdout和stderr，它们都是在<stdio.h>中定义的，可在任何FILE *类型对象可出现的地方使用。需要注意的是，这三个指针是常量而非变量，因此不能给它们赋值。

getchar()与getc(stdin)相同，putchar(c)与putc(c,stdout)相同。实际上，这四个“函数”在<stdio.h>中被定义为宏，因为这样运行速度较快。表6-3给出了<stdio.h>中其他有关的定义。

表6-3 一些<stdio.h>中的定义

名 称	含 义
stdin	标准输入
stdout	标准输出
stderr	标准错误输出
EOF	文件结束，一般是-1
NULL	无效指针，一般是0
FILE	用于定义文件指针
BUFSIZ	正常的I/O缓冲区大小(一般是512或1024)
getc(fp)	从流fp中返回一个字符
getchar()	getc(stdin)
putc(c,fp)	向流fp输出一个字符c
putchar(c)	putc(c,stdout)
feof(fp)	当文件结束时得到一个不为0的数
ferror(fp)	当文件出错时得到一个不为0的数
fileno(fp)	流fp的文件描述符(见第7章)

学习了以上知识，现在就可以编写vis的第三个版本了。如果命令行中有参数，则对其按顺序操作；若无参数，则使用标准输入：

```
/* vis: make funny characters visible (version 3) */

#include <stdio.h>
#include <ctype.h>
int strip = 0;      /* 1 => discard special characters */

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    FILE *fp;

    while (argc > 1 && argv[1][0] == '-') {
        switch (argv[1][1]) {
            case 's': /* -s: strip funny chars */
                strip = 1;
                break;
            default:
                fprintf(stderr, "%s: unknown arg %s\n",
                    argv[0], argv[1]);
                exit(1);
        }
        argc--;
        argv++;
    }
}
```

```

    }
    if (argc == 1)
        vis(stdin);
    else
        for (i = 1; i < argc; i++)
            if ((fp=fopen(argv[i], "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n",
                    argv[0], argv[i]);
                exit(1);
            } else {
                vis(fp);
                fclose(fp);
            }
        exit(0);
    }
}

```

以上代码要求命令行中可选参数(如-s)必须放在前面。当所有这些可选参数被处理完后, argc和argv的内容将有所改变, 以使剩余程序与这些可选参数的存在与否无关。虽然vis现在只有一个可选参数-s, 但我们在编码中仍然使用了一个循环以展示如何对选项进行处理。在第1章中我们谈到了UNIX程序如何不按顺序地处理可选参数, 而排除程序员个人的偏好因素, 现在这样处理参数, 比较容易编写分析参数的代码。某些系统上的getopt(3)函数是用来使参数的处理合理化, 读者可在编制自己的程序前了解一下。

例程vis将一个文件的内容打印出来:

```

vis(fp) /* make chars visible in FILE *fp */
FILE *fp;
{
    int c;

    while ((c = getc(fp)) != EOF)
        if (isascii(c) &&
            (isprint(c) || c=='\n' || c=='\t' || c==' '))
            putchar(c);
        else if (!strip)
            printf("\\%03o", c);
}

```

除了需要一个文件指针类型的参数指定写入的文件外, 函数fprintf与printf是相同的。

函数fclose用来中断函数fopen建立的文件指针与文件外部名称(用户所看到的名称)之间的联系, 使此文件指针能用于其他文件。一个程序可同时打开的文件数目是有限的(约20个), 因此当一个文件不再被使用时, 最好把它关掉。一般来说, 通过调用标准库函数(如printf、putc等)而产生的输出都将被送入缓冲区, 以便批量输出以提高效率(到终端屏幕的输出是一个例外)。对fclose的调用也会迫使系统输出缓冲区的内容。当程序调用exit或从主程序返回时, 将对每一个打开的文件自动调用fclose。

程序与stderr的联系方式和与stdin或stdout的联系方式相同。即使标准输出被重定向, 写到stderr的输出也会显示到用户的终端屏幕上。vis将错误信息写入stderr中而不是stdout中, 这样做的好处是当某一文件由于某种原因不能被访问时, 反映此情况的信息可在终端上显示出来, 而不是消失于一个管道或输出文件中。(当错误信息确实消失于管道后, 将会在管道之后发现一个标准错误。)

在这一版本的vis中, 当它不能打开一个输入文件时将退出, 这种处理方式适合于只有一个输入文件的交互式程序。

练习6-5 编写一个名为printable的程序, 其功能是显示命令行参数中, 那些只包含可打印

字符的文件名称，忽略那些包含任何非打印字符的文件。在以下情况中，`printable`是很有用的：

```
$ pr `printable *` / lpr
```

再加入选项-v，其作用是使`printable`的功能颠倒过来，即显示那些含有非打印字符的文件名称。要考虑命令行中没有文件名参数的情况，以及`printable`的返回值。

6.4 p: 一次显示一屏

到目前为止，我们已用过`cat`命令来查看文件。但如果文件过长，或是与系统的连接速度很快，那么即使使用`ctl-s`和`ctl-q`键，也可能会来不及阅读`cat`的输出。

因此，需要一个能分段显示文件的程序。也许是因为早期的UNIX 是用于硬拷贝(纸)终端，并且使用低速连线，还没有一个标准命令来实现以上功能。我们下一个示例程序名为`p`，它的功能就是一次显示一屏的文件信息，等待用户确认再显示下一屏。同前几个程序相似，它可以从作为命令行参数的文件中或从标准输入中读入数据。

```
$ p vis.c
...
$ grep '#define' *.*[ch] | p
...
$
```

用C语言编写这个程序很容易，而用系统提供的标准工具则不合适，因为这些工具不适于将来自文件或管道的输入与来自终端的输入进行混合处理。

最基本的设计就是将输入分成小块显示。合适的块大小为22行：它小于大多数显示终端的24行的屏幕，并且是66行的标准页的1/3。提示用户的一个简单方法：对每个22行的小块文本都不显示最后一个换行，此时，光标就可停留在最后一行的右端末尾处，而不是下一行的左端起始处。当用户按Return键后，将显示下一屏。若用户在一屏末尾按`ctl-d`或`q`键，则`p`退出。

对于超长行将不作特殊处理。对多个文件处理时，对文件的切换将不作提示。因此，如下语句：

```
$ p 文件名 ...
```

的功能类似于：

```
$ cat 文件名 ... / p
```

对多个文件操作时，可使用for循环：

```
$ for i in 文件名 ...
> do
>     echo $i:
>     cat $i
> done / p
```

实际上，对此程序可有多种改进方式。最好先设计一个最基本的程序，然后再逐步完善它。这样使得程序最后实现的功能是人们真正需要的，而非编程者空想的。

`p`的基本结构与`vis`类似：主程序对多个文件进行循环，并调用例程`print`对每个文件进行进一步处理。

```
/* p:  print input in chunks (version 1) */

#include <stdio.h>
```

```

#define PAGESIZE    22
char    *progrname; /* program name for error message */

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    FILE *fp, *efopen();

    progrname = argv[0];
    if (argc == 1)
        print(stdin, PAGESIZE);
    else
        for (i = 1; i < argc; i++) {
            fp = efopen(argv[i], "r");
            print(fp, PAGESIZE);
            fclose(fp);
        }
    exit(0);
}

```

例程efopen封装了一个常用操作：试图打开一个文件，若不能打开，则显示错误信息并退出。为了显示出错的程序名，efopen用一个外部字符串progrname来储存此名称，它是在主程序中被赋值的。

```

FILE *efopen(file, mode)    /* fopen file, die if can't */
    char *file, *mode;
{
    FILE *fp, *fopen();
    extern char *progrname;

    if ((fp = fopen(file, mode)) != NULL)
        return fp;
    fprintf(stderr, "%s: can't open file %s mode %s\n",
        progrname, file, mode);
    exit(1);
}

```

在此之前，本书还设计了efopen的其他几个版本。一种是在显示错误信息后，返回一个空指针，这就使其调用者可选择是继续还是退出。另一种是为efopen设计第三个参数以指明出错后是否返回。但是在几乎所有的示例中，当一个文件不能被读取时都不能继续，因此efopen的当前版本是符合需求的。

p中核心部分是print例程：

```

print(fp, pagesize) /* print fp in pagesize chunks */
    FILE *fp;
    int pagesize;
{
    static int lines = 0;    /* number of lines so far */
    char buf[BUFSIZ];

    while (fgets(buf, sizeof buf, fp) != NULL)
        if (++lines < pagesize)
            fputs(buf, stdout);
        else {
            buf[strlen(buf)-1] = '\0';
            fputs(buf, stdout);
            fflush(stdout);
            ttyin();
            lines = 0;
        }
}

```

```
    }
```

```
}
```

其中, BUFSIZ是在<stdio.h>中定义的, 在这里用于指明输入缓冲区的大小。fgets(buf,size,fp)从fp指向的文件中读取下一行, 将其存入buf并加入字符串终结符\0。读取的字符个数最大为size-1。遇到文件末尾时, 它返回NULL。(可对fgets加以改进: 返回buf而非字符数目。另外, 如果输入行过长, 它并不提出警告。字符不会丢失, 但必须查看buf的内容以确认是否正确。)

函数strlen返回字符串的长度, 可以使用它来将最后一个输入行的换行屏蔽掉。fputs(buf,fp)将buf中的内容写入文件fp。对fflush的调用迫使系统输出缓冲区中的所有内容。

例程ttyin用于在每屏显示完后读入用户的响应。它不能从标准输入读入, 因为p的输入源有可能是文件或管道。对此的处理方法是打开文件/dev/tty, 无论怎样对标准输入进行重定向, 此文件都代表用户终端。程序还将ttyin的返回值设定为用户响应的第一个字符, 但在这里并未使用此功能。

```

ttyin() /* process response from /dev/tty (version 1) */
{
    char buf[BUFSIZ];
    FILE *efopen();
    static FILE *tty = NULL;

    if (tty == NULL)
        tty = fopen("/dev/tty", "r");
    if (fgets(buf, BUFSIZ, tty) == NULL || buf[0] == 'q')
        exit(0);
    else /* ordinary line */
        return buf[0];
}

```

文件指针devtty被定义为静态变量, 因此, 下次调用ttyin时, 它仍保留上一次的值, 同时, 文件/dev/tty只需在ttyin第一次被调用时打开。

p还有很多可改进的地方, 但以上版本已可实现所需功能: 打印22行并等待。

一个简单的改进为: 每屏所显示的行数可由命令行中的参数指定:

```
$ p -n ...
```

此语句一次显示n行。为实现此功能, 只需在主程序开始处加入一些熟悉的代码:

```

/* p: print input in chunks (version 2) */
...
int i, pagesize = PAGESIZE;

programe = argv[0];
if (argc > 1 && argv[1][0] == '-') {
    pagesize = atoi(&argv[1][1]);
    argc--;
    argv++;
}
...

```

函数atoi将一字符转化为整数(参见atoi(3))。

另外一种改进为: 在显示完每一屏后, 用户可暂时退出以执行其他命令。类似于ed等命令, 当用户在行首输入! 时, 此行余下的部分将被看作是一条命令, 并交给shell来执行。但由于已存在函数system(3)完成类似功能, 因此这一改进又显得微不足道。改进后的ttyin程序为:

```

ttyin() /* process response from /dev/tty (version 2) */
{
    char buf[BUFSIZ];
    FILE *efopen();
    static FILE *tty = NULL;

    if (tty == NULL)
        tty = efopen("/dev/tty", "r");
    for (;;) {
        if (fgets(buf, BUFSIZ, tty) == NULL || buf[0] == 'q')
            exit(0);
        else if (buf[0] == '!') {
            system(buf+1); /* BUG here */
            printf("!\\n");
        }
        else /* ordinary line */
            return buf[0];
    }
}

```

不幸的是，这种版本的ttyin有一个不易察觉但却有害的错误。由system函数运行的命令将从p继承标准输出，因此，若p正在从一个管道或文件中读数据，则此命令将会与p的输入发生冲突：

```

$ cat /etc/passwd | p -1
root:3D.fHR5KoB.3s:0:1:S.User:/:!ed
?
!

```

从p命令内部调用ed
ed读/etc/passwd...
...混乱了，退出

为解决这一问题，应了解系统是如何控制UNIX进程的，对此本书将在7.4节介绍。现在只需知道调用库函数system可能会带来问题，但如果ttyin在第7章所示的系统中编译，它将会正确运行。

到目前为止，我们已写了两个程序：vis和p，它们可看作是cat的变体。那么它们是否应该作为cat命令的一部分呢(比如-v或-p的形式)? 当人们有了新的想法时，就会遇到究竟是编写一个新程序还是在已有程序中加入一新功能的问题。对此并没有一个明确的答案，但的确有一些原则可以帮助人们作出决定。

最主要的原则是：一个程序只应做一项基本工作。因为若功能太多，程序就会变得更大、更慢、更难维护并且更难使用。实际上，许多功能都不常被使用，因为人们根本记不住那么多的选项。

因此，cat与vis是不应合并到一起的。cat只是拷贝输入，并不加以改变，而vis则要改变其输入。将二者合并起来会产生一个做两件不同事情的程序。对cat和p也是同样道理。cat意味着迅速高效的拷贝，而p则意味着浏览。另外，p要对输出做一些变形：每22行要停顿一下。因此，设计三个独立的程序，完成三种不同功能是最好的选择。

练习6-6 当变量pagesize不是正数时，p还能正常工作吗？

练习6-7 思考p还有哪些可改进的地方。如何将p已显示的部分重新显示呢？(这个功能是用用户所需要的。)或者修改p，使其每次显示都少于一屏。为p加入以下功能，使其向前或向后扫描某一行，该行由行代码或关键字指定。

练习6-8 利用exec的文件处理能力改进ttyin对system的调用。

练习6-9 如果忘记为p指定一个输入，则p将一直等待下去。有必要检测这种情况吗？如果有必要，如何检测？提示：isatty(3)。

6.5 pick

第5章中的pick扩展了shell的功能。如下所示的C版本又有所改进：当有命令参数时，它同原来一样运行；当仅有参数-时，pick将对标准输入进行处理。

为何不在没有参数的情况下读取标准输入呢？请思考5.6节中所示的zap命令：

```
kill $SIG `pick \`ps -ag | egrep "$*" \` | awk '{print $1}'`
```

如果egrep模式找不到匹配对象将会发生什么事情呢？在这种情况下，pick没有任何参数，因此就要从标准输入读取数据；而zap命令则会以一种令人疑惑的方式失败。由此看来，加入一个-作为参数可以方便地区分上述情况。

```
/* pick: offer choice on each argument */

#include <stdio.h>
char    *programe; /* program name for error message */

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    char buf[BUFSIZ];

    programe = argv[0];
    if (argc == 2 && strcmp(argv[1], "-") == 0) /* pick - */
        while (fgets(buf, sizeof buf, stdin) != NULL) {
            buf[strlen(buf)-1] = '\0'; /* drop newline */
            pick(buf);
        }
    else
        for (i = 1; i < argc; i++)
            pick(argv[i]);
    exit(0);
}

pick(s) /* offer choice of s */
    char *s;
{
    fprintf(stderr, "%s? ", s);
    if (ttyin() == 'y')
        printf("%s\n", s);
}
```

pick着重于交互式地选择命令参数。它不仅提供了有用的功能，还满足了其他命令对交互式选项的需求。

练习6-10 对pick来说，是否需要rm -i？

6.6 错误与调试

如果你以前编过程序的话，就一定对错误(bug)的出现并不陌生。现在还没有一个好的方法来写一个完全无错的程序，只能先做出一个简单明了的设计，在编程时倍加小心，在修改时保证其正确性。

有许多UNIX工具可帮助你找到错误，但没有一个是最佳的。为了说明这些工具的用法，需要以一个错误为例子，但本书中给出的程序示例都是没有错误的。因此，现在需要创建一个典型的错误。考虑以上介绍的pick程序，下面将介绍一个含有错误的pick程序：

```

pick(s) /* offer choice of s */
char *s;
{
    fprintf("%s? ", s);
    if (ttyin() == 'y')
        printf("%s\n", s);
}

```

如果对其进行编译和运行，将会发生下列情况：

```

$ cc pick.c -o pick
$ pick *.c
Memory fault - core dumped
$

```

尝试
出错!

Memory fault意味着该程序试图去引用一个被禁止访问的内存区域。多数情况下这意味着一个指针指向了一个无效的内存区域。Bus error也表明发生了类似的事情，引起它的原因通常是读一个没有终止符的字符串。

core dumped意味着内核将正在执行的程序的状态储存到当前目录的一个名为core的文件。可以强迫程序进入dump core状态：若程序是在前台运行，按ctl-\键；若程序是在后台运行则输入命令kill-3。

adb和sdb是用来处理错误的两个程序。同大多数调试器相同，它们是神秘、复杂而又不可缺少的。adb目前已到了第七个版本；而sdb在大多数新出的系统中都可以找到。总之，用户总可以找到这两个程序中的一个。

由于篇幅所限，在这里只讨论它们的最简单应用：显示一个栈的跟踪，也就是说，显示程序死掉时其正在执行的函数、此函数的调用者，等等。最先显示的就是当程序发生错误时的当前函数。

用adb获取栈跟踪的命令为\$C：

```

$ adb pick core
$C
~_strout(0175722,011,0,011200)
    adjust:      0
    fillch:      060542
__doprnt(0177345,0176176,011200)
~fprintf(011200,0177345)
    iop:         011200
    fmt:         0177345
    args:        0
~pick(0177345)
    s:           0177345
~main(035,0177234)
    argc:        035
    argv:        0177234
    i:           01
    buf:         0
ctl-d
$

```

调用adb
请求堆栈跟踪
退出

以上结果说明，主程序调用pick，pick调用fprintf，fprintf调用_doprnt，_doprnt调用_strout。由于_doprnt在pick.c中并未出现，因此，错误必定发生在fprintf或其调用者中。（栈跟踪结果中，子例程后面的行显示了局部变量的值。而\$c命令以及部分版本的adb的\$C命令不显示这些信息。）

现在试一下sdb的运行结果：

```

$ sdb pick core
Warning: 'a.out' not compiled with -g
lseek: address 0xa64          程序出错的例程
*t                            请求堆栈跟踪
lseek()
fprintf(6154,2147479154)
pick(2147479154)
main(30,2147478988,2147479112)
*q                            退出
$

```

以上信息的格式与adb的运行结果不太一样，但也指出了问题的发生处：fprintf。(在不同机器上，堆栈跟踪的显示结果是不同的，因为它们的标准I/O库不同。)现在可以确定的是，在这一版本的pick中，如下语句引起错误：

```
fprintf("%s? ", s),
```

由于不存在stderr，格式说明"%s?"被用作一个文件指针，因此错误会发生。

选择这种错误加以说明是因为它的普遍性，它的起因是程序员的疏忽，而并非设计不好。下面的错误也经常发生：调用一函数时使用了错误的参数。这种错误可由C语言的检测工具lint(1)来发现。lint检查C程序中的潜在错误、移植性问题以及重复定义等。如果对pick.c文件使用lint，它会指出如下错误：

```

$ lint pick.c
...
fprintf, arg. 1 used inconsistently "llib-1c"(69) :: "pick. c"(28)
...
$

```

以上结果说明，在程序的第28行，fprintf的第一个参数的使用与标准库中所定义的相矛盾。这对找到错误很有帮助。

lint很有用，它能准确地指出错误的发生处，但同时也显示了许多无关信息，在上例中已被我们忽略掉了。只有当用户有一定经验时才能辨认出什么信息是有用的，什么信息是没用的。具有这种经验非常有用，因为lint能找到一些程序员不易找到的错误。编写了一大段的程序之后，运行lint是很有帮助的，当然，首先得明白它所给出的每行信息的意思。

6.7 zap

在第5章中我们介绍过zap程序，它被看作一个shell文件，其功能是有选择地杀死进程。这一版本的zap最主要的问题是它的运行速度：它创建的进程数目过多，使得其运行速度很慢，而这一点对于杀死错误进程的程序来说很不好。用C语言重新写zap程序会使其运行速度加快。但是，我们并不准备重新编写所有的代码，而是仍使用ps来找到进程信息。这样做比直接从内核中寻找信息要容易得多。在输入端，zap用ps来打开一个管道，并从中读取信息。函数popen(3)类似于fopen，但它的第一个参数是一条命令而非一个文件名。与此相对应的是pclose函数，在这里并不需要使用它。

```

/* zap:  interactive process killer */

#include <stdio.h>
#include <signal.h>
char    *programe; /* program name for error message */
char    *ps = "ps -ag"; /* system dependent */

```

```

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fin, *popen();
    char buf[BUFSIZ];
    int pid;

    progname = argv[0];
    if ((fin = popen(ps, "r")) == NULL) {
        fprintf(stderr, "%s: can't run %s\n", progname, ps);
        exit(1);
    }
    fgets(buf, sizeof buf, fin);    /* get header line */
    fprintf(stderr, "%s", buf);
    while (fgets(buf, sizeof buf, fin) != NULL)
        if (argc == 1 || strindex(buf, argv[1]) >= 0) {
            buf[strlen(buf)-1] = '\0'; /* suppress \n */
            fprintf(stderr, "%s? ", buf);
            if (ttyin() == 'y') {
                sscanf(buf, "%d", &pid);
                kill(pid, SIGKILL);
            }
        }
    exit(0);
}

```

上例中，程序使用了ps-ag(此选项与系统有关)命令，但除非是超级用户，否则你只能删除自己的进程。

对fgets的第一次调用是从ps中提取头行。请推断试图杀死与此行有关的进程时会发生什么。

函数sscanf是scanf(3)函数族中的一员，它也用于对输入格式进行转换。它的处理对象是字符串而不是文件。调用kill命令时，系统会将指定的信号发给相应进程，在<signal.h>中定义的信号SIGKILL不能被捕捉或忽略。你也许还记得在第5章中介绍过此信号的值为9，但最好在程序中使用头文件里定义的符号常量，不要直接使用数值。

如果没有命令参数，zap将按所有可能的选择对ps的每行输出进行处理。如果有参数，则只处理符合此参数的ps输出行。函数strindex(s1,s2)通过使用strncmp(见表6-2)来检测参数是否匹配ps输出行的某一部分。strindex返回s1中匹配s2的位置，如果不能匹配，则返回-1。

```

strindex(s, t) /* return index of t in s, -1 if none */
    char *s, *t;
{
    int i, n;

    n = strlen(t);
    for (i = 0; s[i] != '\0'; i++)
        if (strncmp(s+i, t, n) == 0)
            return i;
    return -1;
}

```

表6-4包括了常用的标准I/O库中的函数。

练习6-11 改写zap，使其能处理任意数目的参数。在以上版本的zap中，它通常会响应与其对应的行，作为选择之一。这样做是否必要？如果不必要，请作出相应修改。提示：getpid(2)。

练习6-12 用fgrep(1)代替strindex。对于含有10个词的文档，比较二者对复杂查询的运行时间。思考为何fgrep的运行速度更快？

表6-4 常用的标准I/O函数

函 数	含 义
fp=fopen(s, 模式)	打开文件，模式为r, w, a时分别表示读、写、添加(当出错时返回NULL)
c=getc(fp)	得到一个字符。getchar()即getc(stdin)
putc(c,fp)	放入一个字符。putchar(c)即putc(c,stdout)
ungetc(c,fp)	将字符放回输入文件fp，每次至多可以放回一个字符
scanf(格式, a1,...)	把字符从stdin读入到a1,...。ai必须是一个指针。返回EOF或者转换后的域值
fscanf(fp,...)	从文件fp中读入
sscanf(s,...)	从字符串s中读入
printf(格式,a1,...)	对a1, ...进行格式化，输出到stdout
fprintf(fp,...)	输出到文件fp
sprintf(s,...)	输出到字符串s
fgets(s,n,fp)	从文件fp中至多读n个字符到s中，在文件末尾返回NULL
fputs(s,fp)	在文件fp中输出字符串s
fflush(fp)	将缓冲中的数据写到fp中
fclose(fp)	关闭文件fp
fp=popen(s, 模式)	为命令s打开一个管道，参见fopen
pclose(fp)	关闭管道fp
system(s)	运行命令s,等待其结束

6.8 idiff: 交互式文件比较程序

一个较普遍的问题是，同一文件的两个版本存在差异，但它们又各自包含所需文件的部分内容。这种情况一般是由于两个人各自独立地修改某文件造成的。diff可显示两个文件有哪些不同，但对于从两个文件中分别选取部分内容来说，此命令没有直接的帮助。

在这一节中，将编写一个名为idiff(interactive diff)的程序，它对diff的每一个输出块都进行处理，使用户能选择from部分和to部分，并且能对其进行编辑。它将用户所选出的部分按正确的规则排列于文件idiff.out中。例如，对如下两个文件：

```

file1:                file2:
This is               This is
a test                not a test
of                    of
your                  our
skill                 ability.
and comprehension.

```

diff显示如下结果：

```

$ diff file1 file2
2c2
< a test
---
> not a test
4,6c4,5
< your
< skill
< and comprehension.

```

```

---
> our
> ability.
$

```

而idiff的输出如下所示:

```

$ idiff file1 file2
2c2                                第一处不同
< a test
---
> not a test
? >                                用户选择第二个(>)版本
4,6c4,5                            第二处不同
< your
< skill
< and comprehension.
---
> our
> ability.
? <                                用户选择第一个(<)版本
idiff output in file idiff.out
$ cat idiff.out                    显示输出这个文件
This is
not a test
of
your
skill
and comprehension.
$

```

如果用户输入的是e而非<或>, idiff将会调用ed命令, 并将已读入的两组文本行传递给它。如果用户输入的第二个参数是e, 则编辑器的缓冲区内容为:

```

your
skill
and comprehension.
---
our
ability.

```

ed的编辑结果将会写入文件, 作为最终输出。

另外, 在idiff的运行过程中, 可使用! 命令来暂时退出以执行其他外部命令。

从技术上讲, 最困难的部分是由diff实现的, 这在上一章中已经讲述过了。因此, idiff实际上就是分析diff的输出, 并且在合适的时候打开、关闭、读或写所要求的文件。idiff的主要例程用于建立文件并调用diff:

```

/* idiff:  interactive diff */

#include <stdio.h>
#include <ctype.h>
char    *programe;
#define HUGE    10000    /* large number of lines */

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fin, *fout, *f1, *f2, *efopen();
    char buf[BUFSIZ], *mktemp();
    char *diffout = "idiff.XXXXXX";

    programe = argv[0];
    if (argc != 3) {
        fprintf(stderr, "Usage: idiff file1 file2\n");
    }
}

```

```

    exit(1);
}
f1 = efopen(argv[1], "r");
f2 = efopen(argv[2], "r");
fout = efopen("idiff.out", "w");
mktemp(diffout);
sprintf(buf, "diff %s %s >%s", argv[1], argv[2], diffout);
system(buf);
fin = efopen(diffout, "r");
idiff(f1, f2, fin, fout);
unlink(diffout);
printf("%s output in file idiff.out\n", progname);
exit(0);
}

```

函数mktemp(3)创建了一个与当前所有文件都不同名的临时文件。它重写了其参数：将6个X用idiff进程的进程标识符和一个字母代替。系统调用unlink(2)删除此临时文件。

函数idiff用于循环处理diff所报告的文件差异。基本思想很简单：显示一部分diff的输出，忽略文件中所不需要的部分，并从另一文件中拷贝需要的部分。实现起来则比较烦琐，所以代码很长，但还是比较容易理解的。

```

idiff(f1, f2, fin, fout)    /* process diffs */
    FILE *f1, *f2, *fin, *fout;
{
    char *tempfile = "idiff.XXXXXX";
    char buf[BUFSIZ], buf2[BUFSIZ], *mktemp();
    FILE *ft, *efopen();
    int cmd, n, from1, to1, from2, to2, nf1, nf2;

    mktemp(tempfile);
    nf1 = nf2 = 0;
    while (fgets(buf, sizeof buf, fin) != NULL) {
        parse(buf, &from1, &to1, &cmd, &from2, &to2);
        n = to1-from1 + to2-from2 + 1; /* #lines from diff */
        if (cmd == 'c')
            n += 2;
        else if (cmd == 'a')
            from1++;
        else if (cmd == 'd')
            from2++;
        printf("%s", buf);
        while (n-- > 0) {
            fgets(buf, sizeof buf, fin);
            printf("%s", buf);
        }
        do {
            printf("? ");
            fflush(stdout);
            fgets(buf, sizeof buf, stdin);
            switch (buf[0]) {
                case '>':
                    nskip(f1, to1-nf1);
                    ncopy(f2, to2-nf2, fout);
                    break;
                case '<':
                    nskip(f2, to2-nf2);
                    ncopy(f1, to1-nf1, fout);
                    break;
                case 'e':
                    ncopy(f1, from1-1-nf1, fout);
                    nskip(f2, from2-1-nf2);
                    ft = efopen(tempfile, "w");
                    ncopy(f1, to1+1-from1, ft);

```

```

        fprintf(ft, "---\n");
        ncopy(f2, to2+1-from2, ft);
        fclose(ft);
        sprintf(buf2, "ed %s", tempfile);
        system(buf2);
        ft = fopen(tempfile, "r");
        ncopy(ft, HUGE, fout);
        fclose(ft);
        break;
    case 'l':
        system(buf+1);
        printf("l\n");
        break;
    default:
        printf("< or > or e or l\n");
        break;
}
} while (buf[0]!='<' && buf[0]!='>' && buf[0]!='e');
nf1 = to1;
nf2 = to2;
}
ncopy(f1, HUGE, fout); /* can fail on very long files */
unlink(tempfile);
}

```

函数parse所做的工作平凡而又巧妙，它用来分析diff的输出行，提取4个行号和命令(a、c和d中的一个)。它的复杂之处在于diff既可输出1个行号，又可在命令字符的两侧输出两个行号。

```

parse(s, pfrom1, pto1, pcmd, pfrom2, pto2)
    char *s;
    int *pcmd, *pfrom1, *pto1, *pfrom2, *pto2;
{
    #define a2i(p) while (isdigit(*s)) p = 10*(p) + *s++ - '0'

    *pfrom1 = *pto1 = *pfrom2 = *pto2 = 0;
    a2i(*pfrom1);
    if (*s == ',') {
        s++;
        a2i(*pto1);
    } else
        *pto1 = *pfrom1;
    *pcmd = *s++;
    a2i(*pfrom2);
    if (*s == ',') {
        s++;
        a2i(*pto2);
    } else
        *pto2 = *pfrom2;
}

```

其中，宏a2i将ASCII码转换为整数。

nskip和ncopy从一文件中忽略或拷贝指定数量的文本行：

```

nskip(fin, n) /* skip n lines of file fin */
    FILE *fin;
{
    char buf[BUFSIZ];

    while (n-- > 0)
        fgets(buf, sizeof buf, fin);
}

ncopy(fin, n, fout) /* copy n lines from fin to fout */
    FILE *fin, *fout;

```

```

{
    char buf[BUFSIZ];

    while (n-- > 0) {
        if (fgets(buf, sizeof buf, fin) == NULL)
            return;
        fputs(buf, fout);
    }
}

```

idiff在被中断时,不能正确地退出,因为/tmp中还有一些文件等待处理。下一章将介绍如何捕获中断以删除临时文件。

zap和idiff的关键之处在于,大多数复杂的工作是由其他命令完成的,只需提供一个方便的接口来和这些已有的命令联系。编程之前想想哪些部分可由别人完成是很有好处的。

练习6-13 在idiff中加入q命令: q<的功能是自动获取所有剩余的<选项;而q>的功能是自动获取所有剩余的>选项。

练习6-14 修改idiff,使其可传递所有diff的参数,比如-b和-h。修改idiff,使其可指定一个不同于ed的编辑器,如下所示:

```
$ idiff -e 另一个编辑器 file1 file2
```

这两个修改如何交互作用?

练习6-15 用popen和pclose代替用于diff输出的临时文件。这两种处理方式的运行速度与复杂性有何不同?

练习6-16 当diff的某一参数为一目录时,它应该在此目录中寻找一个与另一个参数同名的文件。但当idiff遇到此种情况时,将会以一种奇怪的方式失败。请解释这种现象并加以改正。

6.9 获取环境变量

在C程序中,获取shell环境变量是很方便的,这种方法常用于使程序能适应其运行环境,并且不需用户更多的参与。比如说,你正在使用一台屏幕显示范围大于24行的终端,如果在运行p时要充分利用屏幕的显示属性该怎么做呢?每次运行p时都指定屏幕尺寸是件很烦人的事情:

```
$ p -36 . . .
```

另一种解决方式是在bin目录下放一个shell文件:

```

$ cat /usr/you/bin/p
exec /usr/bin/p -36 $*
$

```

第三种解决方式是在p中使用环境变量来定义终端的属性。可以在.profile文件中定义一个名为PAGESIZE的变量:

```

PAGESIZE=36
export PAGESIZE

```

例程getenv("var")用于搜寻名为var的环境变量并以字符串的形式返回此变量的值,若此变量不存在,则返回NULL。因此,使用getenv可以方便地修改p。只需在主程序的开始处增加一些定义以及对getenv的调用。

```

/* p:  print input in chunks (version 3) */
...
    char *p, *getenv();

    progame = argv[0];
    if ((p=getenv("PAGESIZE")) != NULL)
        pagesize = atoi(p);
    if (argc > 1 && argv[1][0] == '-') {
        pagesize = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
...

```

由于可选参数在环境变量之后被处理，因此，任何一个明确给定的页大小将覆盖掉缺省的说明。

练习6-17 修改idiff，使其搜寻与所用编辑器名称相对应的环境变量。对PAGESIZE赋不同值，如2、3等。

有关文献

标准I/O库是由Dennis Ritchie在Mike Lesk的I/O库出现后设计的。这两个库的设计目标都是为了提供足够的标准工具，使得当程序从UNIX系统向非UNIX系统移植时不需改变。

本章对p的设计基于Henry Spencer所编的一段程序。

adb由Steve Bourne编写，sdb由Howard Katseff编写，lint由Steve Johnson编写。

idiff基于Joe Maranzano编写的一段程序。diff本身由Doug McIlroy编写，它基于Harold Stone、Wayne Hunt和Tom Szymanski所发明并实现的一个算法(参见“A fast algorithm for computing longest common subsequences”，J.W.Hunt和T.G.Szymanski著，CACM, 1977年5月)。M.D.McIlroy和J.W.Hunt在“An algorithm for differential file comparison” (Bell labs Computing Science Technical Report 41, 1976年)中阐述了diff算法。McIlroy说：“在给出最终算法之前，我至少试过了完全不同的三种算法。而diff则集中了它们中的精华。”

第7章 UNIX系统调用

本章集中讨论UNIX操作系统最底层的界面：系统调用。它们是通往内核的入口，是操作系统所提供的工具，任何其他功能都以它们为基础构建。

下面将讨论几个主要的领域。首先是I/O系统，它们是像fopen和putc这样的库函数的基础。本章还将更多地讨论文件系统，特别是目录和I节点。其次讨论进程，即怎样在一个程序内运行其他程序。最后，将讨论信号和中断：当按下Delete键时会发生什么，以及在程序内部如何处理这种情况。

和第6章中一样，本章的例子中很多都是有用的程序，它们并不是第7版本的一部分。即使它们并不能给你带来直接的帮助，你也可以在阅读的过程中学到一些东西，而且在为自己的系统构建类似的工具时，它们可以作为很好的范例。

有关系统调用完整的细节可以参考《UNIX程序员手册》第2节。本章仅描述最重要的部分。

7.1 低级I/O

最底层的I/O是进入操作系统的一个直接入口。你的程序可按任何大小的块读写文件。内核缓冲用户数据，将其分成块以匹配外围设备，并调度设备上的操作，优化执行性能。

7.1.1 文件描述符

所有的输入和输出都通过读或写文件来完成，因为所有的外设，甚至是终端都被看作文件系统中的文件。这意味着一个单一接口就能处理程序和外设之间的所有通信。

大多数的情况下，读或写一个文件之前，都必须首先通知系统，程序有这样做的意图，这通过调用打开文件的进程来实现。如果想要写一个文件，就必须先创建它。系统检查用户的权限，（文件存在吗？用户是否有权访问它？）如果一切都正常，则返回一个非负整数，称作文件描述符。无论何时要在文件上做I/O操作，都用文件描述符代替文件名来识别文件。一个打开了的文件的所有信息都要由系统来维护，你的程序只能通过文件描述符来引用文件。正如在第6章讨论的那样，一个FILE指针指向一个结构，该结构包含了文件描述符和其他一些信息。宏fileno(fp)定义在<stdio.h>中，它返回文件描述符。

为了使终端的I/O更为方便，还存在一些特殊安排。shell启动一个程序时，该程序会自动继承三个打开的文件，其文件描述符为0、1和2，分别称作标准输入、标准输出和标准错误输出。所有这些都缺省连接到终端，因此如果一个程序只读文件描述符0，只写描述符1和2，那它不必打开任何其他文件就可以完成I/O操作。如果程序打开任何其他文件，就会有文件描述符3、4等等。

如果I/O被重定向到管道或文件，shell就会把对文件描述符0和1的缺省值从终端改变到指定的文件。通常文件描述符2依然连到终端上，这样错误信息就可以在那里显示。shell命令，诸如2 > 文件名和2 >&1，会导致I/O重定向。（如果需要的话，程序本身可以对这些做进一步

的重定向，但这是很少见的。)

7.1.2 文件I/O

所有的I/O都是由两个系统调用完成：read和write，它们来自C的同名函数。对于这两个函数，第一个参数都是文件描述符；第二个参数是字符数组，用作数据源或数据目标；第三个参数是要传输的字节数。

```
int fd, n, nread, nwritten;
char buf[SIZE];

nread = read(fd, buf, n);
nwritten = write(fd, buf, n);
```

每次调用均返回传输的字节数。在读的时候，返回的字节数可能要比要求读的数目少，这是因为剩余可读的字节数可能会小于n。(当文件是一个终端时，read通常只读到下一个换行符，而这通常都会比要求读的少。)返回值0说明文件结束，而-1则说明发生了某种错误。对于写来说，返回值是真正写入的字节数，如果该字节数与预期要写入的字节数不等，则说明发生了错误。

当要读或写的字节数未指定时，通常认为是1，这意味着一次一个字符(“非缓冲”方式)。一般磁盘块的大小为512或1024个字节。(在<stdio.h>中的BUFSIZE参数规定此值。)

作为演示，下面的程序把输入复制到输出。由于输入和输出可以被重定向到任何文件或设备，所以它实际上可以复制任何源到任何目的地：它是实现cat的雏形。

```
/* cat: minimal version */
#define SIZE 512 /* arbitrary */

main()
{
    char buf[SIZE];
    int n;

    while ((n = read(0, buf, sizeof buf)) > 0)
        write(1, buf, n);
    exit(0);
}
```

如果文件的大小不是SIZE的整数倍，一些读操作就会返回较少的字节数，在这以后的read将返回0。

读写操作可以以块为单位进行，块大小与磁盘匹配时，这样做的效率非常高。甚至一次一个字符的方式也适用于很多情况，因为内核会将用户数据存入缓冲区。这样做主要的系统开销就是系统调用。例如ed，它在获得其标准输入时采用一次一个字符的方式。下面是针对一个54000字节的文件，6个不同的SIZE值，测试cat的现有版本得出的时间代价：

SIZE	Time (user+system, sec.)	
	PDP-11/70	VAX-11/750
1	271.0	188.8
10	29.9	19.3
100	3.8	2.6
512	1.3	1.0
1024	1.2	0.6
5120	1.0	0.6

其中，在PDP-11系统上磁盘大小为512字节，而在VAX机上为1024字节。

允许多个进程同时访问同一个文件，事实上，一个进程在对文件进行写操作时，另一个进程可以对该文件进行读操作。这在某些时候可能会非常有用。甚至是在一个读操作已经返回0以后(意味着已到文件末尾)，只要别的写操作将更多的数据写入文件，那么后续的读操作仍然读出更多的字节。这个事实是程序readslow的基础，它持续读它的输入，而不管是否已经到了文件结束。它可以用于观察一个程序的处理过程：

```
$ slowprog >temp &                               进程标识符
5213
$ readslow <temp | grep something
```

换句话说，一个程序在一个文件内产生持续的输出。readslow和其他程序结合在一起，可以观察数据的积累。

从结构上说，除了当readslow遇到当前输入的结束符时它会循环而不是退出，它与cat是类似的。它必须使用低级I/O，因为标准库函数在遇到第一个文件结束符后会报告EOF。

```
/* readslow: keep reading, waiting for more */
#define SIZE 512 /* arbitrary */

main()
{
    char buf[SIZE];
    int n;

    for (;;) {
        while ((n = read(0, buf, sizeof buf)) > 0)
            write(1, buf, n);
        sleep(10);
    }
}
```

函数sleep的功能是使程序挂起，以等待特定的时间，它的详细描述见sleep(3)。我们并不想让readslow不停地访问文件以寻找更多的数据，因为这要极大地耗费CPU的时间。因而，这个版本readslow复制它的输入直到文件结束，然后睡眠一会儿，再尝试。如果在它睡眠的过程中到达了更多数据，就会在下次的read中读入。

练习7-1 给readslow添加一个-n参数，这样缺省的睡眠时间可以改变成n秒。一些系统给程序tail提供了-f(forever)选项，而把tail的功能和readslow的功能结合在一起。试对这样的设计进行论述。

练习7-2 如果正在读的文件被截短了，readslow会怎样做？怎样修改它？提示：参考第7.3节的fstat。

7.1.3 创建文件

除了缺省的标准输入、输出和错误文件以外，对于其他文件，都必须明确地打开该文件才能对它们进行读和写。这可通过两种系统调用来实现：open和creat[⊖]。

open与上一章讲的fopen很相像，但它返回的是文件描述符而不是文件指针，文件描述符的类型为int。

⊖ 有一次Ken Thompson被问到：“如果让你重新设计UNIX系统，会有什么不同？”他的回答是：“把creat改成create。”

```
char *name;
int fd, rwmode;
```

```
fd = open(name, rwmode);
```

和fopen一样，name参数是一个包含文件名的字符串。然而，访问模式参数不同：rwmode为0代表读；为1代表写；为2表示所打开的文件既可读又可写。如果有错误发生，open返回-1，否则返回一个有效的文件描述符。

试图打开一个不存在的文件会产生错误。系统调用creat创建一个新文件，或重写一个旧文件。

```
int perms;
```

```
fd = creat(name, perms);
```

如果creat创建名为name的文件成功，它将返回一个文件描述符，否则返回-1。如果文件不存在，creat按照perms参数指定的权限创建该文件。如果文件已经存在，creat把它的长度截短为0。创建一个已经存在的文件并不是一个错误。(权限也不会被改变。)无论perms是什么，一个创建好的文件都为写而打开。

正如第2章所描述的那样，有9位保护信息与每个文件相连，控制读、写和执行权限，所以用一个3位的八进制数说明它们就很方便。例如，0755说明文件所有者具有读、写和执行权限，而同组成员和其他人只具有读和执行权限。不要忘记，前导的0是C中用于说明八进制数的。

为了说明，下面是个简化版的cp。最主要的简化是该版本只复制一个文件，而且不允许第2个参数是一个目录。另一个缺陷是该版本不保护源文件的权限，后面将会看到怎样来补救这一点。

```
/* cp: minimal version */
#include <stdio.h>
#define PERMS 0644 /* RW for owner, R for group, others */
char *progname;

main(argc, argv) /* cp: copy f1 to f2 */
{
    int argc;
    char *argv[];

    {
        int f1, f2, n;
        char buf[BUFSIZ];

        progname = argv[0];
        if (argc != 3)
            error("Usage: %s from to", progname);
        if ((f1 = open(argv[1], 0)) == -1)
            error("can't open %s", argv[1]);
        if ((f2 = creat(argv[2], PERMS)) == -1)
            error("can't create %s", argv[2]);

        while ((n = read(f1, buf, BUFSIZ)) > 0)
            if (write(f2, buf, n) != n)
                error("write error", (char *) 0);
        exit(0);
    }
}
```

本章会在后面的小节中讨论error。

一个程序可以同时打开的文件数有一个限制(典型的是20左右，参看<sys/param.h>中

NOFILE)。相应地，想要处理多个文件的程序必须为文件描述符的重新使用做准备。系统调用CLOSE终止文件名和文件描述符之间的联系，释放文件描述符供其他程序使用。程序的终止可由EXIT来完成，或关闭所有打开的文件后从主程序返回。

系统调用unlink从文件系统中删除文件。

7.1.4 错误处理

本节所讨论的系统调用，而且事实上是所有的系统调用，都有可能产生错误。通常它们通过返回值-1说明发生了错误。有些时候需要知道到底发生了哪种错误。出于这个目的，所有的系统调用在合适的时候，都会在一个称作errno的外部整数中留下一个错误号。(各种错误号的含义见《UNIX程序员手册》第2节。)例如，通过使用errno，程序可以判定，打开文件的失败是由于它不存在，还是由于没有读权限。另外还有一个字符串数组sys_errlist，它以errno为下标，可以把数字转换成有意义的字符串。以下的errno版本使用了这些数据结

```
error(s1, s2)    /* print error message and die */
{
    char *s1, *s2;

    extern int errno, sys_nerr;
    extern char *sys_errlist[], *progrname;

    if (progrname)
        fprintf(stderr, "%s: ", progrname);
    fprintf(stderr, s1, s2);
    if (errno > 0 && errno < sys_nerr)
        fprintf(stderr, " (%s)", sys_errlist[errno]);
    fprintf(stderr, "\n");
    exit(1);
}
```

errno初始为0，并且通常会小于sys_nerr。在运行正常时它不会被重置为0，所以，如果程序想要继续执行，就必须在每次产生错误后把它重置为0。

下面是运行目前版本的cp时所出现的错误消息：

```
$ cp foo bar
cp: can't open foo (No such file or directory)
$ date >foo; chmod 0 foo
$ cp foo bar
cp: can't open foo (Permission denied)
$
```

创建一个不可读文件

7.1.5 随机访问

文件的I/O操作通常是顺序的：每次在文件内的读或写都紧跟在上次操作之后。然而，如果必要，一个文件可以按任意的顺序读写。系统调用lseek提供了一种方式，可以不必完成实际的读或写操作就在一个文件内随机移动：

```
int fd, origin;
long offset, pos, lseek();

pos = lseek(fd, offset, origin);
```

lseek使得描述符fd所指定的文件的当前读写位置移动到offset处，它是相对于origin所指定的位置计算得出的相对位置。随后的读和写将从此处开始。origin可以为0、1或2，分别说明相

对于文件的开始、文件的当前位置或文件的尾开始移动。返回值是新的绝对位置，如果出错，则返回-1。例如，为了在文件后添加内容，写之前要定位到文件的末尾：

```
lseek (fd, 0L, 2) ;
```

重新返回到开始：

```
lseek (fd, 0L, 0) ;
```

取得当前位置：

```
pos = lseek (fd, 0L, 1) ;
```

注意参数0L：offset是一个长整型。(在lseek中的l代表了long，以区分第6版中使用短整型值的seek系统调用。)

使用lseek，可以像对待一个大的数组一样对待文件，代价是访问速度变慢。例如，下面的函数从文件的任意位置读出任意数目的字节：

```
get(fd, pos, buf, n) /* read n bytes from position pos */
{
    int fd, n;
    long pos;
    char *buf;
    if (lseek(fd, pos, 0) == -1) /* get to pos */
        return -1;
    else
        return read(fd, buf, n);
}
```

练习7-3 修改readslow，使它对只存在一个文件名参数的情况进行处理。添加选项-e：

```
$ readslow -e
```

使readslow在开始读之前定位到输入的末尾。lseek可以在管道上做什么？

练习7-4 重写第6章的efopen以调用error。

7.2 文件系统：目录

下一个主题是怎样穿越目录层次。实际上，它并没有使用任何新的系统调用，只是在新的环境下使用旧的知识。本节通过编写一个名为spname的程序进行说明，它试图处理拼写错误的文件名。函数

```
n = spname (名称, 新名称);
```

搜索名字与name最为接近的文件。如果找到一个，就把该文件复制到文件newname中去。如果没有找到任何文件，spname返回值-1；如果有一个精确的匹配，返回为0；如果作了一次更正，则返回1。

对于p命令，spname是一个方便的附加品：如果想要打印一个文件，却又拼错了名字，p就会询问用户，真正想要打印的是不是别的什么文件：

```
$ p /urs/srx/ccmd/p/spnam.c          糟糕，拼错了名字
"/usr/src/cmd/p/spname.c"? y        建立接受这一修正
/* spname:  return correctly spelled filename */
...
```

spname会试图更正发生在文件名每一个部分的不匹配，如漏掉或多加了一个字母，或某个字母被拼错了，再或者是一对字母交换了位置。所有这些上面都作了说明。对于粗心大意的打

字员来说，这可真是上帝的恩赐。

在编写代码之前，最好先对文件系统的结构做一简单的回顾。一个目录是一个文件，它包含了文件名和位置的列表。“位置”实际上是一个指针，用于在另一个称作I节点表的表格内检索信息。文件的I节点保留了该文件除文件名之外的全部信息。一个目录项只由两项组成，一个I节点号和文件名。关于I节点准确的说明可在文件<sys/dir.h>中找到：

```
$ cat /usr/include/sys/dir.h
#define DIRSIZ 14 /* max length of file name */

struct direct /* structure of directory entry */
{
    ino_t d_ino; /* inode number */
    char d_name[DIRSIZ]; /* file name */
};
$
```

ino_t是一个typedef类型，描述了I节点表的索引。在PDP-11和VAX版的系统内，它恰好是无符号短整型，但它绝不能直接嵌入到程序中：在不同的机器上它有可能不同。一个完整的“系统”类型集合可以在<sys/types.h>中找到，它必须被包括在<sys/dir.h>之前。

spname的操作十分明白直接，但还有许多边界条件需要仔细考虑。假设文件名为/d1/d2/f。基本思想是去掉第一个部分(/)，然后搜索目录找到和下一个部分(d1)接近的名字，接着再搜索目录找到与d2近似的，等等，直到找到与每一个部分匹配的。如果在任何阶段，目录内没有一个恰当的候选者，搜索就将放弃。

现在把工作分成三个函数，spname本身分离路径部分，并把它们构造成一个“迄今为止最佳匹配”的文件名；调用mindist函数搜索一个指定的目录，寻找与当前最接近的文件；第三个函数spdlist用于计算两个名字之间的距离。

```
/* spname: return correctly spelled filename */
/*
 * spname(oldname, newname) char *oldname, *newname;
 * returns -1 if no reasonable match to oldname,
 *          0 if exact match,
 *          1 if corrected.
 * stores corrected name in newname.
 */

#include <sys/types.h>
#include <sys/dir.h>

spname(oldname, newname)
    char *oldname, *newname;
{
    char *p, guess[DIRSIZ+1], best[DIRSIZ+1];
    char *new = newname, *old = oldname;

    for (;;) {
        while (*old == '/') /* skip slashes */
            *new++ = *old++;
        *new = '\0';
        if (*old == '\0') /* exact or corrected */
            return strcmp(oldname, newname) != 0;
        p = guess; /* copy next component into guess */
        for ( ; *old != '/' && *old != '\0'; old++)
            if (p < guess+DIRSIZ)
                *p++ = *old;
        *p = '\0';
        if (mindist(newname, guess, best) >= 3)
```

```

        return -1; /* hopeless */
    for (p = best; *new = *p++; ) /* add to end */
        new++; /* of newname */
    }
}

mindist(dir, guess, best) /* search dir for guess */
char *dir, *guess, *best;
{
    /* set best, return distance 0..3 */
    int d, nd, fd;
    struct {
        ino_t ino;
        char name[DIRSIZ+1]; /* 1 more than in dir.h */
    } nbuf;

    nbuf.name[DIRSIZ] = '\0'; /* +1 for terminal '\0' */
    if (dir[0] == '\0') /* current directory */
        dir = ".";
    d = 3; /* minimum distance */
    if ((fd=open(dir, 0)) == -1)
        return d;
    while (read(fd, (char *) &nbuf, sizeof(struct direct)) > 0)
        if (nbuf.ino) {
            nd = spdist(nbuf.name, guess);
            if (nd <= d && nd != 3) {
                strcpy(best, nbuf.name);
                d = nd;
                if (d == 0) /* exact match */
                    break;
            }
        }
    close(fd);
    return d;
}

```

如果赋给mindist的目录名为空，则搜索 . 目录。mindist一次读入一个目录项。请注意read的缓冲区是一个结构，而不是一个字符数组。使用SIZEOF来计算字节数，并强行将地址转换为字符指针。

如果目录中的一个槽当前未被使用(因为一个文件已被删除)，那么I节点为零，该位置被跳过。距离测试为：

```
if (nd <= d ...)
```

而不是

```
if (nd < d ...)
```

所以任何其他单个字符都是比 . 更好的匹配，. 表示当前目录，它通常是目录中的第一项。

```

/* spdist: return distance between two names */
/*
 * very rough spelling metric:
 * 0 if the strings are identical
 * 1 if two chars are transposed
 * 2 if one char wrong, added or deleted
 * 3 otherwise
 */

#define EQ(s,t) (strcmp(s,t) == 0)

spdist(s, t)

```

```

char *s, *t;
{
    while (*s++ == *t)
        if (*t++ == '\0')
            return 0;          /* exact match */
    if (*--s) {
        if (*t) {
            if (s[1] && t[1] && *s == t[1]
                && *t == s[1] && EQ(s+2, t+2))
                return 1;      /* transposition */
            if (EQ(s+1, t+1))
                return 2;      /* 1 char mismatch */
        }
        if (EQ(s+1, t))
            return 2;          /* extra character */
    }
    if (*t && EQ(s, t+1))
        return 2;              /* missing character */
    return 3;
}

```

一旦完成了spname, 在p中加入拼写检查就很容易了:

```

/* p:  print input in chunks (version 4) */

#include <stdio.h>
#define PAGE_SIZE 22
char    *programe; /* program name for error message */

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fp, *efopen();
    int i, pagesize = PAGE_SIZE;
    char *p, *getenv(), buf[BUFSIZ];

    programe = argv[0];
    if ((p=getenv("PAGE_SIZE")) != NULL)
        pagesize = atoi(p);
    if (argc > 1 && argv[1][0] == '-') {
        pagesize = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
    if (argc == 1)
        print(stdin, pagesize);
    else
        for (i = 1; i < argc; i++)
            switch (spname(argv[i], buf)) {
                case -1: /* no match possible */
                    fp = efopen(argv[i], "r");
                    break;
                case 1: /* corrected */
                    fprintf(stderr, "\"%s\"? ", buf);
                    if (ttyin() == 'n')
                        break;
                    argv[i] = buf;
                    /* fall through... */
                case 0: /* exact match */
                    fp = efopen(argv[i], "r");
                    print(fp, pagesize);
                    fclose(fp);
            }
    exit(0);
}

```

拼写更正不可以盲目地用于所有使用文件名的程序。它能够与p合作是因为p是交互性的，它不适合于不可交互的程序。

练习7-5 在spname选择了最好匹配的情况下，还可以做哪些改进？例如，把一个普通文件当作一个目录进行处理是很愚蠢的，但在当前的版本上有可能发生。

练习7-6 tx匹配目录中最后一个出现的tc，其中c代表任何单个字符。你可以发明一个更好地测量距离的方法吗？实现它并看一下实际用户对它的使用情况。

练习7-7 mindist一次只读一个目录项。如果按照较大的块来读目录，p的运行是否会明显地快一点？

练习7-8 修改spname，使它在找不到任何近似匹配时，把与其前缀相匹配的名字返回。如果有几个名字同时与其前缀匹配时，怎么办？

练习7-9 其他的程序可从spname中得到什么好处？设计一个单独运行的程序，使它在将参数送给其他程序之前，对它们进行更正，如：

```
$ fix prog 文件名 ...
```

你可以编写一个使用spname的新版cd命令吗？怎样安装它？

7.3 文件系统：I节点

这一节中讨论的系统调用主要处理文件系统，特别是处理与文件有关的信息，如文件大小、日期、权限等等。通过这些系统调用可以获得在第2章中讨论过的全部信息。

现在深入到I节点内部。部分I节点由一个stat结构来描述，该结构定义在<sys/stat.h>中：

```
struct stat /* structure returned by stat */
{
    dev_t    st_dev;      /* device of inode */
    ino_t    st_ino;      /* inode number */
    short    st_mode;     /* mode bits */
    short    st_nlink;    /* number of links to file */
    short    st_uid;      /* owner's userid */
    short    st_gid;      /* owner's group id */
    dev_t    st_rdev;     /* for special files */
    off_t    st_size;     /* file size in characters */
    time_t   st_atime;    /* time file last read */
    time_t   st_mtime;    /* time file last written or created */
    time_t   st_ctime;    /* time file or inode last changed */
};
```

其中大多数的域，注释都已作出了解释。dev_t和ino_t这样的类型定义在<sys/types.h>中，这一点上面已作了讨论。st_mode项包含用于描述文件的标志集，为了方便起见，标志定义也是文件<sys/stat.h>的一部分：

```
#define S_IFMT    0170000 /* type of file */
#define S_IFDIR  0040000 /* directory */
#define S_IFCHR  0020000 /* character special */
#define S_IFBLK  0060000 /* block special */
#define S_IFREG  0100000 /* regular */
#define S_ISUID  0004000 /* set user id on execution */
#define S_ISGID  0002000 /* set group id on execution */
#define S_ISVTX  0001000 /* save swapped text even after use */
#define S_IREAD  0000400 /* read permission, owner */
#define S_IWRITE 0000200 /* write permission, owner */
#define S_IXEC   0000100 /* execute/search permission, owner */
```

对一个文件I节点的访问是通过一对系统调用来完成的：stat和fstat。stat接收一个文件名，

返回该文件的I节点信息(如果发生错误则返回-1)。fstat从一个打开的文件或文件描述符完成同样的工作(而不是从一个file指针)。也就是说:

```
char *name;
int fd;
struct stat stbuf;

stat(name, &stbuf);
fstat(fd, &stbuf);
```

用文件name或文件描述符fd的I节点信息填充结构stbuf。

有了这些知识,就可以写一些有用的程序了。现在以一个C语言的checkmail程序开始,它监视你的邮箱。如果文件变大,checkmail会打印出You have mail并振铃。(如果文件变短,那就假设你刚刚已经阅读并删除了一些邮件,不提供任何消息。)作为编程的第一步,这已经足够了。完成了这项工作,可以再向前扩展。

```
/* checkmail: watch user's mailbox */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
char *progname;
char *maildir = "/usr/spool/mail"; /* system dependent */

main(argc, argv)
    int argc;
    char *argv[];
{
    struct stat buf;
    char *name, *getlogin();
    int lastsize = 0;

    progname = argv[0];
    if ((name = getlogin()) == NULL)
        error("can't get login name", (char *) 0);
    if (chdir(maildir) == -1)
        error("can't cd to %s", maildir);
    for (;;) {
        if (stat(name, &buf) == -1) /* no mailbox */
            buf.st_size = 0;
        if (buf.st_size > lastsize)
            fprintf(stderr, "\nYou have mail\007\n");
        lastsize = buf.st_size;
        sleep(60);
    }
}
```

函数getlogin(3)返回你的登录名,否则返回NULL。checkmail使用系统调用chdir切换到邮件目录,这样随后的stat调用就不必搜索从根到邮件目录的每一个目录。你必须改变自己的maildir,使它在自己的系统中能正确工作。让checkmail在即使没有邮箱的情况下也不停地尝试,因为大多数版本的mail程序会在邮箱为空的时候删掉它。

在第5章说明shell循环时也写了这样一个程序。那个程序在每次查看邮箱时就会创建几个进程,所以它可能要进行比你所预期的更多次的系统载入。这个C版本程序只有一个进程,它每分钟在文件上做一次stat调用。如果让checkmail在后台永不停止地运行,代价会怎样呢?笔者对其进行了测试,结果是每小时中它所占用的时间不到1秒,这已经足够低了,完全可以忽略。

sv: 错误处理示例

下面要写的程序叫做sv，类似于cp，它将几个文件复制到一个目录下，但仅当目标文件不存在或者比源文件旧时才改变目标文件。sv代表了save，它不会覆盖最新的東西。它要比checkmail使用更多I节点中的信息。

设想像下面这样使用sv：

```
$ sv file1 file2 ... dir
```

将file1复制到dir/file1；将file2复制到dir/file2，等等，如果目标文件比源文件新，则不作任何复制，只输出一个警告。为了避免对链接文件做多份复制，sv不允许/在任何源文件名中出现。

```
/* sv: save new files */
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
char *programe;

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    struct stat stbuf;
    char *dir = argv[argc-1];

    programe = argv[0];
    if (argc <= 2)
        error("Usage: %s files... dir", programe);
    if (stat(dir, &stbuf) == -1)
        error("can't access directory %s", dir);
    if ((stbuf.st_mode & S_IFMT) != S_IFDIR)
        error("%s is not a directory", dir);
    for (i = 1; i < argc-1; i++)
        sv(argv[i], dir);
    exit(0);
}
```

I节点内的时间是从以前某个时刻(格林威治平时1970年1月1日0:00)到现在的秒数，所以旧文件在它们的st_mtime域的值要比新文件的小。

```
sv(file, dir) /* save file in dir */
    char *file, *dir;
{
    struct stat sti, sto;
    int fin, fout, n;
    char target[BUFSIZ], buf[BUFSIZ], *index();

    sprintf(target, "%s/%s", dir, file);
    if (index(file, '/') != NULL) /* strchr() in some systems */
        error("won't handle /'s in %s", file);
    if (stat(file, &sti) == -1)
        error("can't stat %s", file);
    if (stat(target, &sto) == -1) /* target not present */
        sto.st_mtime = 0; /* so make it look old */
    if (sti.st_mtime < sto.st_mtime) /* target is newer */
        fprintf(stderr, "%s: %s not copied\n",
            programe, file);
    else if ((fin = open(file, 0)) == -1)
        error("can't open file %s", file);
    else if ((fout = creat(target, sti.st_mode)) == -1)
        error("can't create %s", target);
}
```

```

else
    while ((n = read(fin, buf, sizeof buf)) > 0)
        if (write(fout, buf, n) != n)
            error("error writing %s", target);
    close(fin);
    close(fout);
}

```

使用creat代替标准I/O函数，这样sv就可以保留输入文件的模式。(请注意index和strchr是同一函数的不同名字。检查手册中string(3)中的内容，看一下你的系统使用的是哪个名字。)

尽管sv程序的针对性很强，但它确实显示了一些重要的思想。许多程序不是“系统程序”，但仍然使用操作系统所维护的信息，并通过系统调用对它们进行访问。对于这样的程序，信息的表示仅出现在像<stat.h>和<dir.h>这样的头文件中，这是非常关键的，而且这些程序也只引用这些文件，而不是在自己的程序内嵌入实际的定义。这样代码就会更有可能从一系统移植到另一系统。

另一点值得注意的是，在sv中至少有三分之二的代码是错误检查。在编写程序的早期阶段，人们试图跳过错误检查，因为它会分散对主要任务的注意力。一旦程序可以“工作”了，再想返回去加入检查的代码，以使程序从一个私有程序转变成为一个无论什么情况下都可以继续工作的程序，就不那么容易了。

sv并没有防止所有可能的错误——例如，它没有处理在关键时刻所发生的中断，但比起大多数的程序来说，它已经很完备了。为了集中讨论一点，考虑最后的write语句。write语句发生错误的情况比较少见，所以许多程序员忽略了这种可能性。但是磁盘没有空间、用户超过限额、传输线路中断等等，所有这些都会导致write出错，如果能够处理这些，要比程序沉默地假装一切都运行良好要好的多。

因此结论就是出错检查虽然很烦琐，却很重要。由于篇幅所限，本书中的大部分程序都没有做错误检查，只是集中在更有意义的主题上讨论。但是现实生活中，尤其是对于商品程序，如果忽略了错误检查，后果将无法担负。

练习7-10 修改checkmail程序，识别邮件的发送者，并把它作为You have mail消息的一部分。提示：使用sscanf和lseek。

练习7-11 修改checkmail程序，使它在进入循环之前不切换到邮件目录。这对它的性能有什么显著的影响吗？(较难)能否写一个checkmail版本，它只需一个进程就可以通知所有用户？

练习7-12 写一个程序watchfile，它监视一个文件，并在每次改变文件时输出文件内容。什么时候你会使用它？

练习7-13 sv在错误处理方面比较严格。修改它，使它在即使不能处理一些文件时仍能继续运行。

练习7-14 使sv递归：如果源文件之一是一个目录，该目录和它下面的文件将以同样的方式进行处理。使cp也递归。考虑cp和sv是否应该为一个程序，以使cp-v命令在目标文件较新时不做复制。

练习7-15 编写程序random：

\$ random 文件名

从文件中随机地选出一行并输出。给定名为people的文件，其内容是人名，random可以被用

在一个名为scapegoat的程序中，该程序用于分配责任：

```
$ cat scapegoat
echo "It's all `random people`'s fault!"
$ scapegoat
It's all Ken's fault!
$
```

确保无论行的长度怎样分布，random都应保持公平。

练习7-16 在I节点中还有其他的信息，特别是，文件块所在的磁盘地址。检查文件<sys/ino.h>，然后写一个程序icat，它读取由I节点号和磁盘设备所指定的文件。在什么样的情况下会用到icat呢？

7.4 进程

本节描述怎样在一个程序内执行另一个程序。最简单的方式是使用标准库例程system，在第6章中已经提到过它并做了一些介绍。system接受一个参数，该参数是一个命令行，就像在终端所键入的那样(除了末尾的换行)，并且在子shell内执行它。如果命令行由几部分构成，那么sprintf这种内存格式化能力也许会很有用。在本节的末尾，会给出system的一个较安全的版本，供交互式程序使用，但首先我们必须看一下构成它的各个部分。

7.4.1 创建低级进程

最基本的操作是执行另一个程序而不必返回，这是通过系统调用execlp来完成的。例如，一个正在运行的程序的最后一步可以打印出日期，如下：

```
execlp("date", "date", (char *) 0);
```

execlp的第一个参数是命令的文件名：execlp从用户的环境中抽取搜索路径(即\$PATH)，并进行搜索，搜索的方式与shell相同。第二个及其后继参数是命令名及命令的参数。这些就是新程序的argv数组。列表的最后用一个0参数作为标识。(阅读exec(2)，熟悉execlp的设计。)

execlp调用一个新程序覆盖正在运行的程序，运行该新程序，然后退出。只有当发生错误时，原先的程序才重新获得控制，例如，如果没有找到文件或该文件不能被执行：

```
execlp("date", "date", (chr*) 0);
fprintf(stderr, "Couldn't execute date\n");
exit(1)
```

execlp的一个变体称作execvp，当事先不知道将有多少个参数时，可以使用execvp。调用方法是：

```
execvp(filename, argp);
```

其中，argp是指向参数(如argv)的指针数组，数组中的最后一个指针必须是NULL，这样execvp可以告诉程序列表的结束位置。和execlp一样，filename是程序所在的文件；argp是新程序的argv数组。argp[0]是程序名。

这些例程都没有提供元字符的扩展，像<、>、*、引号等等。如果想在参数列表中使用这些字符，使用execlp激活shell程序/bin/sh，它会接下来完成所有的工作。构造一个字符串commandline，让它包含完整的命令，就像应该在终端上键入的那样，然后使用：

```
execlp("/bin/sh", "sh", "-c", commandline, (char *) 0);
```

参数-c说明, 把下一个参数作为整个命令行, 而不是单个参数来对待。

作为exec的一个示例, 考虑程序waitfile。命令是:

```
$ waitfile 文件名 [命令]
```

它周期性地检查指定的文件。若从上次检查以来未做修改, 则执行命令。如果没有指定任何命令, 就把文件复制到标准输出。可以使用waitfile监视troff过程, 正如:

```
$ waitfile troff.out echo troff done &
```

在waitfile的实现中, 使用fstat抽取文件最后一次被修改的时间。

```
/* waitfile: wait until file stops changing */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
char *progname;

main(argc, argv)
    int argc;
    char *argv[];
{
    int fd;
    struct stat stbuf;
    time_t old_time = 0;

    progname = argv[0];
    if (argc < 2)
        error("Usage: %s filename [cmd]", progname);
    if ((fd = open(argv[1], 0)) == -1)
        error("can't open %s", argv[1]);
    fstat(fd, &stbuf);
    while (stbuf.st_mtime != old_time) {
        old_time = stbuf.st_mtime;
        sleep(60);
        fstat(fd, &stbuf);
    }
    if (argc == 2) { /* copy file */
        execlp("cat", "cat", argv[1], (char *) 0);
        error("can't execute cat %s", argv[1]);
    } else { /* run process */
        execvp(argv[2], &argv[2]);
        error("can't execute %s", argv[2]);
    }
    exit(0);
}
```

上述例子说明了execlp和execvp的用法。

之所以选用这项设计是因为它很有用, 其他方案也是可行的。例如, waitfile可以在文件改变结束后简单地返回。

练习7-17 修改watchfile(练习7-12), 使它和waitfile具有相同的属性: 如果没有command, 它复制文件; 否则, 它执行命令。watchfile和waitfile可以共享源代码吗? 提示: argv[0]。

7.4.2 控制进程

下一步就是在使用execlp或者execvp运行完一个程序后, 原来程序重新获得控制。由于这些例程只是简单地把新程序覆盖在旧的上, 因而为了保护旧的程序首先就需要把它分割成两个副本: 其中的一个被覆盖, 而另一个则等待新的覆盖进程的结束。分割操作通过名为fork的

系统调用来完成:

```
proc_id = fork( )
```

把程序分割成两个副本，它们都继续运行。两者之间唯一的区别在于fork返回的值不同，即进程标识符不同。这两个进程中一个是子进程，proc_id为0；另一个是父进程，proc_id为非0，它是子进程的进程标识符。因而调用另一个程序并从另一个程序返回的基本方式是：

```
if (fork( ) == 0)
    execlp("/bin/sh", "sh", "-c", commandline, (char *) 0);
```

事实上，除了还需要错误处理外，这就已经足够了。fork制作程序的两个副本。在子进程，fork返回的值为0，所以它调用execlp，完成commandline，然后退出。在父进程，fork返回非0值所以它跳过execlp。(如果发生了任何错误，fork返回-1。)

更多的时候，父进程等待子进程结束以后才开始运行它自己。这通过系统调用wait来完成：

```
int status;

if (fork() == 0)
    execlp(...);    /* child */
wait(&status);      /* parent */
```

这里依然没有处理任何意外情况，如execlp或者fork的失败，或多个子程序同时运行的可能性。(wait返回结束了的子进程的进程标识符，可以拿它与fork返回的值作对照。)同时，该程序片断也没有处理任何子进程的异常行为。然而，这两行仍然是标准system函数的核心。

wait返回的status的低8位，被系统用来表示子进程的退出状态，它为0表示正常终止，非0则表示出现了某种问题。次高8位取自调用exit的参数或者main函数的返回值。

当shell调用一个程序时，三个文件描述符0、1和2都已被设定指向正确的文件，并且所有其他的文件描述符都可以使用。当该程序调用另一个程序时，建议确保同样的状态被保留下来。fork或exec的调用不能以任何方式影响打开的文件，父进程和子进程有相同的打开文件。如果父进程要在子进程之前输出缓冲区的内容，则必须在调用execlp之前刷新其缓冲区。相反地，如果父进程正在缓冲输入流，子进程就会丢失任何已被父进程读入的信息。输出可以被刷新，但是输入不能被送回。如果使用第6章所讨论的标准I/O库完成I/O操作的话，就必须进行这些考虑，因为这些库通常对I/O都要缓冲。

正是这种穿越execlp的文件描述符之间的继承关系破坏了system：如果调用程序不让它的标准输入和输出连接到终端，system所调用的命令也不能。然而，这有时正是用户所想要的，例如，在一个ed脚本内，以一个!开始的命令的输入可能就来自于脚本。尽管这样，ed也必须以每次一个字母的方式来读它的输入，避免输入缓冲问题。

对于像p这样的交互式程序，system应该把标准输入和输出重新连接到终端。有一种方式就是把它们连接到/dev/tty。

系统调用dup(fd)在最低序号的未分配的文件描述符上复制文件描述符fd，返回指向相同打开文件的一个新的文件描述符。下面的代码把一个程序的标准输入连接到一个文件：

```
int fd;

fd = open("file", 0);
close(0);
```

```
dup(fd);
close(fd);
```

close(0)去掉对文件描述符0，即标准输入的分配，但在通常情况下它不影响父进程。

下面是用于交互式程序的system版，它使用programe处理错误信息。这里关于信号处理的那部分函数在下一节将会讨论。

```
/*
 * Safer version of system for interactive programs
 */
#include <signal.h>
#include <stdio.h>

system(s) /* run command line s */
char *s;
{
    int status, pid, w, tty;
    int (*istat)(), (*qstat)();
    extern char *programe;

    fflush(stdout);
    tty = open("/dev/tty", 2);
    if (tty == -1) {
        fprintf(stderr, "%s: can't open /dev/tty\n", programe);
        return -1;
    }
    if ((pid = fork()) == 0) {
        close(0); dup(tty);
        close(1); dup(tty);
        close(2); dup(tty);
        close(tty);
        execlp("sh", "sh", "-c", s, (char *) 0);
        exit(127);
    }
    close(tty);
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return status;
}
```

注意/dev/tty被以模式2打开——可读和可写——然后进行复制，形成标准输入和输出。这实际上就是系统在你登录时装配标准输入、输出和错误输出的情况。因而，用户的标准输入是可写的：

```
$ echo hello 1>&0
hello
$
```

这意味着可以让被复制的文件描述符2重新连接到标准输入和输出，但是打开/dev/tty更清楚更安全。即使如此，system还存在潜在的问题：调用者打开的文件，如p中例程ttyin中的tty，将会被传递给子进程。

本书在这里并没有说你应该在所有的程序中使用我们的system版本——例如，它会破坏一个非交互性的ed，但你应该理解进程是怎样被管理的，以及怎样正确地使用原语；“正确”的含义随着应用程序的不同而不同，也许并不符合system实现的标准。

7.5 信号和中断

本节着重于怎样巧妙地处理外部世界的信号(如中断), 以及程序故障。程序故障主要是由于非法引用内存引起, 如执行了特定的指令, 或是浮点错误。最普通的外来信号有: 中断, 当敲入DEL字符时发出; 退出, 通过FS字符(ctl-\)产生; 挂起, 挂起电话线所引起; 终止, 通过kill命令产生。当这些事件之一发生时, 信号被送到同一终端开始的所有进程, 除非已做出其他安排, 否则信号将终止进程。对大多数信号而言, 一个核心映像文件被生成, 以备调试。(见abd(1)和sbd(1)。)

系统调用signal改变缺省的动作。它有两个参数。第一个是一个数字, 用于说明信号。第二个或者是一个函数的地址, 或者是一段代码, 该代码要么要求信号被忽略, 要么给出缺省的处理。文件<signal.h>包含了各种参数的定义。因而

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

将使中断被忽略而

```
signal(SIGINT, SIG_DFL);
```

恢复缺省的处理使进程终止。在所有的情况下, signal返回前一个信号值。如果signal的第2个参数是函数名(必须已经在同一个源文件中进行了定义), 当信号发生时将调用此函数。大多数情况下这项功能允许程序在结束之前处理未完成的工作, 例如, 删除一个临时文件:

```
#include <signal.h>
char *tempfile = "temp.XXXXXX";

main()
{
    extern onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    mktemp(tempfile);

    /* Process ... */

    exit(0);
}

onintr()    /* clean up if interrupted */
{
    unlink(tempfile);
    exit(1);
}
```

程序为什么要在主程序中测试并两次调用signal呢? 信号被送给从一特定终端启动的所有进程。相应地, 当一个程序将以非交互式的方式运行时(由&启动), shell将保护该程序, 忽略中断, 所以它不会被想要成为前台进程的中断所终止。如果该程序在开始时宣告要把所有的中断毫不考虑地送给 onintr例程处理, 那么当它运行在后台时, 就会破坏shell对它的保护。

上述解决方案是测试中断处理的状态, 如果中断已经被忽略那就继续忽略它们。编写的代码依赖于signal返回的特定信号的前一种状态。如果信号已经被忽略, 进程将继续忽略它们; 否则, 它们将被捕获。

比较高级的程序可能希望截获中断, 解释中断请求的内容, 并返回到程序本身的命令处

理循环。考虑一个文本编辑器：中断一个长时间的打印输出不应使它退出，也不应丢失已经完成的工作。该例的代码可以如下：

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main()
{
    int onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    setjmp(sjbuf); /* save current stack position */

    for (;;) {
        /* main processing loop */
    }
    ...
}

onintr() /* reset if interrupted */
{
    signal(SIGINT, onintr); /* reset for next interrupt */
    printf("\nInterrupt\n");
    longjmp(sjbuf, 0); /* return to saved state */
}
```

文件<setjmp.h>定义类型jmp_buf 为一个对象，在该对象内可以保存栈的位置，sjbuf就被定义为这样一个对象。函数setjump(3)保存一个记录，它记载了程序在哪里执行。变量的值没有被保存。当发生了一个中断时，强制调用例程onintr，该例程打印一个消息，设置标志，或做其他的一些什么事。longjump先作为setjump存入的一个对象的参量，然后恢复控制到调用setjmp之后的位置。这样控制(和栈层次)会回退到主程序例程中主循环进入的位置。

注意，在中断发生以后，在onintr中信号被重新设置。这是必要的：当信号发生时，它们会自动重置到它们的缺省状态。

一些仅想检测信号的程序不能在任意点上被终止，例如，在更新一个复杂数据结构的过程中。解决方案是让中断例程设置一个标志并返回，而不是调用exit或longjump。该程序在中断点处继续执行，中断标志可稍后再测。

这个方法中还存在一个困难。假设当中断被送出时，程序正在读终端。这时指定的例程被按时调用，它设置标志并返回。如果这确实为真，正如我们上面所说的那样，执行就在“中断的那个点”处继续，程序继续读终端直到用户键入另外一行。这种情况可能会很复杂，由于用户可能并不知道程序正在读，并片面地希望信号立即生效。为解决这个困难，系统终止read，但使用一个错误状态以指明发生了什么事情：errno设置为EINTR，在<errno.h>中定义，指明一个被中断的系统调用。

捕获信号并在信号后继续执行的程序应准备好处理由中断的系统调用所引起的“错误”。(用于监视的系统调用是来自终端的read、wait和pause命令。)这样的程序在读标准输入的时候，可以使用如下代码：

```
#include <errno.h>
extern int errno;

...
if (read(0, &c, 1) <= 0) /* EOF or interrupted */
```

```

    if (errno == EINTR) { /* EOF caused by interrupt */
        errno = 0; /* reset for next time */
        ...
    } else { /* true end of file */
        ...
    }
}

```

当信号捕获与执行其他程序相结合时，要牢记它们之间的细微差别。假设一个程序捕获中断，同时又包括一个执行其他程序的方法(像ed中的!)。代码看上去就会像下面这样：

```

if (fork() == 0)
    execlp(...);
signal(SIGINT, SIG_IGN); /* parent ignores interrupts */
wait(&status);           /* until child is done */
signal(SIGINT, onintr);  /* restore interrupts */

```

为什么会是这样？信号被送给所有的进程。假设你调用的程序捕获了它自己的中断，就像一个编辑器所做的那样。如果你中断了子程序，它就会取得信号并返回到它的主循环，并有可能读取你的终端。但是调用程序也会跳出等待子程序的wait，也去读你的终端。两个读终端的进程是很容易混淆的，事实上系统是用抛硬币的方法来决定谁该得到输入的哪一行。解决方案是让父进程忽视中断直到子进程完成。这个方案在system中的信号处理部分得到体现：

```

#include <signal.h>

system(s) /* run command line s */
char *s;
{
    int status, pid, w, tty;
    int (*istat)(), (*qstat)();

    ...
    if ((pid = fork()) == 0) {
        ...
        execlp("sh", "sh", "-c", s, (char *) 0);
        exit(127);
    }
    ...
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return status;
}

```

很明显函数signal的第二参量很奇怪，实际上它是一个整数函数的指针，是信号程序本身的类型。SIG_IGN和SIG_DFL这两值有正确的类型，但是所作的选择要使它们不能与真正的函数一起运行。下面给出了它们在PDP-11和VAX中的定义，这定义是如此的不方便，因此推荐使用<signal.h>。

```

#define SIG_DFL (int (*)( ))0
#define SIG_IGN (int (*)( ))1

```

报警

系统调用alarm(n)会在n秒后发送一个信号SIGALRM给进程。该报警信号可用来确保在一

定时间内发生一些事情：如果发生了，报警信号就可以关闭；如果没有，程序就可以在捕获报警信号后重新获得控制。

为了说明这一点，这里给出一个叫timeout的程序，它运行一个命令：如果该命令在指定的时间内没有完成，就在报警信号结束时终止它。例如，第5章的watchfor命令，可以为它设置一个小时的时间限制，而不让它无休止地运行下去：

```
$ timeout -3600 watchfor dmg &
```

下面给出的timeout程序几乎概括了前两节讨论的所有内容：子进程建立，父进程设置报警，然后等待子进程结束。如果报警先到，子进程就被杀掉。程序中还试图返回子进程的退出码：

```
/* timeout: set time limit on a process */
#include <stdio.h>
#include <signal.h>
int pid;          /* child process id */
char *progrname;

main(argc, argv)
    int argc;
    char *argv[];
{
    int sec = 10, status, onalarm();

    progrname = argv[0];
    if (argc > 1 && argv[1][0] == '-') {
        sec = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
    if (argc < 2)
        error("Usage: %s [-10] command", progrname);
    if ((pid=fork()) == 0) {
        execvp(argv[1], &argv[1]);
        error("couldn't start %s", argv[1]);
    }
    signal(SIGALRM, onalarm);
    alarm(sec);
    if (wait(&status) == -1 || (status & 0177) != 0)
        error("%s killed", argv[1]);
    exit((status >> 8) & 0377);
}

onalarm() /* kill child when alarm arrives */
{
    kill(pid, SIGKILL);
}
```

练习7-18 你能推测出sleep是怎样实现的吗？提示：pause(2)。在什么情况下，sleep和alarm会相互影响？

有关文献

对于UNIX系统的实现没有任何详细的描述，部分是因为代码的所有权。Ken Thompson的论文“UNIX implementation” (BSTJ, 1978年7月)描述了其基本思想。相关的其他论文还有“The UNIX system——a retrospective”发表在同一期的BSTJ，和“The evolution of the UNIX time-sharing system” (Symposium on Language Design and Programming Methodology,

Springer-Verlag出版社, Lecture Notes in Computer Science #79, 1979年), 作者都是Dennis Ritchie。

程序readslow的作者是Peter Weinberger, 它是在象棋比赛的过程中, 使用该程序作为一种低消耗的方式, 观察Belle、Ken Thompson和Joe Condon的象棋机的进展。Belle在一个文件中记录它的游戏的状态, 观察者使用readslow选择文件, 就不必从Belle那里盗用过多珍贵的循环。(Belle 硬件的最新版本只在主机上实现很少的计算, 因而问题也就不复存在了。)

spname的灵感来自于Tom Duff。Ivor Durham、David Lamb 和James Saxe所著的一篇论文“Spelling correction in user interface” (CACM, 1983年10月), 在一个邮件程序的环境内, 提出了拼写检查的某些不同的设计。

第8章 程序开发

UNIX系统最初是作为一个程序开发环境出现的。本章将介绍一组用于开发程序的特定工具。我们将完成一个实际的程序——一个程序设计语言的解释程序，这个语言类似于BASIC。之所以选择实现一种语言，是因为它包括了在一个大型程序中所能遇到的各种有代表性的问题。而且，许多程序都可以视为一种将系统输入转换为一连串的操作和输出的语言，所以本章用语言开发工具作为例子。

在本章中，我们将讨论下列工具：

- yacc 一个语法分析程序生成器，可以从语言的语法描述生成语法分析程序。
- make 一个通过指定和控制进程对复杂程序进行编译的程序。
- lex 一个类似于yacc的程序，用于生成词法分析程序。

本章体现这样一种观点：要完成一个项目，应该先从小做起，然后不断发展壮大，在语言功能不断进展的过程中，展现开发工具的使用。

本章将分六个阶段介绍UNIX程序设计方法，每一个阶段都很有用，即使开发不再进行下去也是如此。这些阶段非常类似于程序员实际编写程序的步骤。

1) 一个四种功能的计算器，提供了+、-、*、/和括号，可以进行浮点运算。每行输入一个表达式，并立即打印出结果。

2) 增加a到z变量名，该版本还具备负数运算，并提供一些纠错功能。

3) 增加长度任意的变量名，增加内部函数如sin、exp等，有用的常量如 π （由于印刷限制一般拼写成PI）和指数运算。

4) 对内部实现的一个改变：为每一条语句生成相应代码，然后对其进行解释，而不是随时进行求值。这主要为下一步做准备，没有增加新特性。

5) 增加控制流：if-else和while，带有{和}的复合语句，以及关系运算如>、<=等。

6) 增加带参数的递归函数和过程，还增加I/O字符串和数字的语句。

在第9章里描述了该语言，它将作为UNIX文档预备软件演示中的主要例子。附录B给出了该语言的参考手册。

由于本章涉及到正确编写大程序过程中的许多细节，所以篇幅很长，这也是理所当然的。我们假定你懂得C语言，并且手边备有《UNIX程序员手册》第2卷，因此不再解释每一个细节。在附录C中我们给出了最后版本的所有代码，使读者可以更容易地了解到各个部分怎样彼此衔接。

此外，我们花费了许多时间考虑给该语言起个什么样的名字，但总没有十分满意的结果。最后我们将该语言命名为hoc(high-order calculator)，表示“高级计算器”。因此它的版本就成为hoc1、hoc2等。

8.1 第一阶段：四功能计算器

本节介绍hoc1的实现，hoc1是一个提供类似小型便携计算器功能的程序，hoc1有四个功

能: +、-、* 和/, 并且具备括号功能以使表达式以任意深度嵌套, 而这是一般便携计算器所没有的。如果输入一个表达式, 按Return后, 将在下一行显示出答案。

```
$ hoc1
4*3*2
      24
(1+2) * (3+4)
      21
1/2
      0.5
355/113
      3.1415929
-3-4
hoc1: syntax error near line 4      它还没有执行单日减功能
$
```

8.1.1 语法

自从采用巴科斯-诺尔范式(Backus-Naur Form)描述Algol语言以来, 一直使用形式语法来描述语言。hoc1的语法描述很小而且简单, 其抽象形式表示如下:

```
list:  expr \n
      list expr \n
expr:  NUMBER
      expr + expr
      expr - expr
      expr * expr
      expr / expr
      ( expr )
```

换句话说, 一个list是一个表达式序列, 每个表达式后面跟一个换行符。表达式或者是一个数字; 或者是用运算符连接的两个表达式; 或者是一个用括号括起来的表达式。

按照严格的语法, 上面的形式描述是不完全的。该文法没有指定操作符的优先关系和运算符的结合性, 也没有给任何结构附加任何意义。尽管用expr定义list, 用NUMBER定义expr, 而NUMBER本身却无定义。这些细节将在从语言框架到一个可工作程序的发展过程中补充。

8.1.2 yacc概述

yacc是一个语法分析生成器[⊖], 它将一种语言的规范化描述(如上述语言)转换成一个语法分析程序, 这个语法分析程序可以对语言中的语句进行语法分析。yacc提供了将该语义与语法成分相关联的一种方法, 利用这一方法, 当进行语法分析时, 语义也可以被“评价”。使用yacc可分为以下几个阶段。

1) 将语法写下来, 像上面所写的那样, 但要更精确。它给出了语言的语法, 此时, 用yacc可以查出语法中的错误及二义性。

2) 用一个操作扩充语法的每一个规则或产生式。该操作是一条语句, 当对程序进行语法分析时, 对发现的语法形式实例, 由该操作决定执行什么动作。“执行什么动作”部分是用C语言写的, 其语法符合C代码标准。这样语言的语义就确定了。

3) 需要一个词法分析程序, 该程序读入要进行语法分析的内容, 并把它分解成对语法分析有意义的块。例如, NUMBER就是一个若干字符长的词法块, 每个单运算符, 如+和*也是

⊖ yacc即yet another compiler-compiler, 这个名称是由它的创造者Steve Johnson在发布它时, 鉴于已有许多类似的程序而决定的(1972年)。yacc是其中最有力的工具之一。

块。习惯上，将词法块称为记号(token)。

4) 需要一个控制例程，调用由yacc生成的语法分析程序。

yacc将该语法和语义操作处理成一个语法分析函数，称为yyparse，并写成C程序文件。如果yacc没有发现任何错误，则语法分析程序、词法分析程序及控制例程便可进行编译，也可与其他C程序连接并运行。这一程序的操作就是不断调用词法分析器，取得记号，识别出输入中的语法结构，并完成相应的语义动作。语法分析器的入口必须被命名为yylex，因为 yyparse每次要求下一个时，都会调用名为yylex的函数(yacc使用的所有名字都是y开头的)。

为了更精确地表示，yacc的输入采用下列形式：

```
%{
C语句，如#include 语句、定义语句等。这部分可省。
}%

yacc定义：词法记号、语法变量、优先级和结合顺序。

%%
语法规则和动作。
%%
其他C语句(可选)。
main() { ...; yyparse(); ... }
yylex() { ... }
...
```

这是yacc处理的源文件格式，yacc处理后的结果写入一个名为y.tab.c的文件中，该文件的格式如下：

```
%{和}%之间的C语句
第2个%%之后的C语句
main() { ...; yyparse(); ... }
yylex() { ... }
...
yyparse() { 分析器，调用    yylex() }
```

yacc产生C文件，而不是已被编译的目标文件，这是一种典型的UNIX处理方法。这种安排最为灵活——因为这样一来，当程序员有了一个好的想法时，可以移植生成的代码，并保持一致性。

yacc本身是一个强有力的工具。可能要花费相当的努力才能掌握它，但一旦会用了yacc，就会发现这是值得的。yacc产生的语法分析程序规模小、效率高，而且正确(当然其中的语义动作是由用户自己写的)，许多难以解决的语法分析问题都能自动处理。可以很容易地建立起语言识别程序，而且(可能更重要的是)，这样建立起来的语言识别程序可以随着语言定义变化而反复修改。

8.1.3 第一阶段的程序

hoc1的源代码由一个带动作的语法分析程序、词法分析程序yylex和main组成，放在文件hoc.y里，(yacc的文件名传统上以.y结束。但这并不是yacc本身所要求的，不像cc和.c。)语法部分位于hoc.y的开头：

```
$ cat hoc.y
%{
#define YYSTYPE double /* data type of yacc stack */
}%
%token  NUMBER
```

```

%left '+' '-' /* left associative, same precedence */
%left '*' '/' /* left assoc., higher precedence */
%%
list:      /* nothing */
| list '\n'
| list expr '\n' { printf("\t%.8g\n", $2); }
;
expr:      NUMBER { $$ = $1; }
| expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
;
%%
/* end of grammar */
...

```

这部分程序包含了许多新的信息，这里不打算逐一讲述，也不打算讲述分析器是如何工作的——关于这些，可以参阅yacc手册。

供选择的规则用|隔开。任意语法规则可以有一个与之关联的动作，当输入中的语法规则实例被识别出时，该动作被执行。一个动作是一对花括号({和})中的一组C语句。在一个动作内部，\$n(如\$1、\$2，等等)表示规则的第n个子成分的返回值，\$\$是整个规则的返回值。例如，在如下规则之中

```
expr:  NUMBER { $$ = $1; }
```

\$1是识别NUMBER时的返回值，该值同时也被作为整个规则expr的值返回。赋值\$\$=\$1可以省略，因为\$\$总被缺省地置为\$1，除非用户显式地将其设为其他值。

在下一层，当规则为：

```
expr :  expr '+' expr { $$ = $1 + $3; }
```

此时结果expr的返回值为两个子expr的值之和。注意这里+是\$2，每个成分都要被编号。

在此基础上，一个以换行(\n)结束的表达式被识别为一个list，并打印出它的结果。若输入的末尾紧跟着一个这样的结构，则语法分析过程终止。一个list可以是空字符串。这是hoc1能处理空输入行的原因所在。

yacc的输入可以采用任意格式，但建议使用上述标准格式。

在这种实现中，识别和分析输入动作的同时，也会引起对表达式的立即求值。在更复杂的情形下(例如hoc4及其更新版本)，分析过程将产生延后计算的代码。

图8-1是一棵分析树，显示了由树叶到根自下而上地求值传递过程，分析树有助于理解语法分析过程。

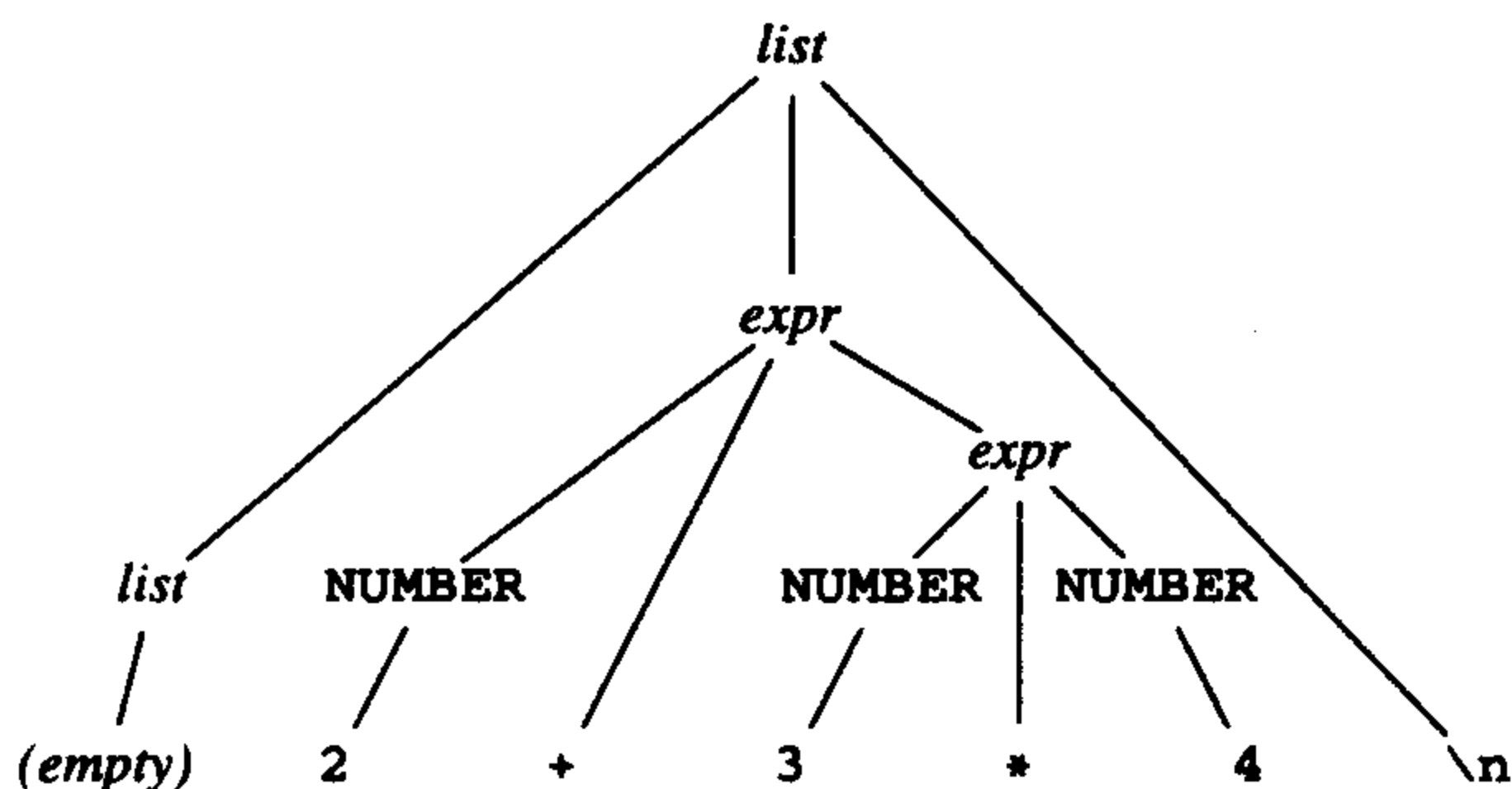


图8-1 2+3*4的语法树

实际上, 尚未完全识别的规则的值被放入一个栈中, 通过栈, 规则的值在规则之间传递。这种栈的数据类型通常是int, 但既然需要处理浮点数, 就必须推翻默认定义。下述定义

```
#define YYSTYPE double
```

设置栈类型为双精度数。

由词法分析器识别的语法单位, 除了单字符, 如 + 和-等, 都应该定义。%token定义了一个或多个这样的对象。通过适当使用%left或%right来代替%token, 可以指定左结合或右结合方式(左结合意味着a-b-c将被分析为(a-b)-c, 而不是a-(b-c))。优先级顺序由出现的次序决定, 在同一个定义中出现的记号具有同一优先级。这样定义的语法具有二义性(即可以有多种路径分析某些输入), 但可以在定义中加入一些额外的信息可以解决二义性。

代码余下的部分是文件hoc.y第二部分的程序:

继续hoc.y

```
#include <stdio.h>
#include <ctype.h>
char    *programe;    /* for error messages */
int      lineno = 1;

main(argc, argv)      /* hoc1 */
{
    char *argv[];
    {
        programe = argv[0];
        yyparse();
    }
}
```

main调用yyparse分析输入。根据list序列, 在语法内部就完成了从一个表达式到下一个表达式的循环。同样也可以在main内部进行对yyparse的循环调用, 而list的动作是打印出它的值并立即返回。

yyparse反复调用 yylex以获得输入记号。这里的 yylex很简单: 它跳过空格和Tab, 将数字序列转换为数值, 计算输入行数以备报错, 并返回其他任何字符。因为语法中仅出现 +、-、*、/、(、)和\, 其他任何字符都将导致yyparse报错。返回0表明yyparse的输入文件结束。

继续hoc.y

```
yylex()      /* hoc1 */
{
    int c;

    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) {    /* number */
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    if (c == '\n')
        lineno++;
    return c;
}
```

变量yylval被用于在语法分析器和词法分析器中传递信息, 它是由yyparse定义的, 与yacc的栈有相同的数据类型。yylex返回一个记号的类型作为它的函数值, 而将 yylval置为该记号的值(如果有的话)。例如, 一个浮点数类型为NUMBER, 其值为12.34。对某些记号而言, 尤

其是单字符 + 和 \n, 语法分析并不需要它们的值, 而只需要类型。在这种情况下, yylval 不必设置。

yacc 的定义 %token NUMBER 在 yacc 的输出文件 y.tab.c 中被转换为 #define 语句, 因此 NUMBER 可以在 C 程序中任何地方被作为常量使用。yacc 选择的 NUMBER 值不会同 ASCII 字符值发生冲突。

若出现了一个语法错误, yyparse 调用 yyerror, 并传递给它一个包含 syntax error 错信息的字符串。yacc 希望用户能够提供 yyerror 函数。在这里, yyerror 仅将该字符串传递给另一个函数, 名为 warning, 它打印出更为详细的信息。后面的 hoc 版本将直接调用 warning。

```
yyerror(s)      /* called for yacc syntax error */
    char *s;
{
    warning(s, (char *) 0);
}

warning(s, t)   /* print warning message */
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

这段程序标志着 hoc.y 中例程的结束。

yacc 程序的编译分两步进行:

\$ yacc hoc.y	输出生成 y.tab.c 文件
\$ cc y.tab.c -o hoc1	输出生成可执行程序 hoc1
\$ hoc1	
2/3	
0.66666667	
-3-4	
hoc1: syntax error near line 1	
\$	

练习8-1 考查 y.tab.c 文件的结构。(hoc1 大约有 300 行。)

8.1.4 在程序中增加单目减

前面提到过, 使用 yacc 可以使得修改语言很容易。作为一个例子, 下面在 hoc1 中增加单目减运算, 使得如下表达式

-3-4

可以被求值, 而不会产生语法错。

只需要增加两行到 hoc.y 里。在定义优先级部分的末尾加入一个新的记号 UNARYMINUS, 使得单目减运算具有最高优先级:

```
%left '+' '-'
%left '*' '/'
%left UNARYMINUS          /* new */
```

语法部分增加一个 expr 的产生式:

```

expr:      NUMBER      { $$ = $1; }
      | '-' expr %prec UNARYMINUS {$ $ = -$2; } /* new */

```

%prec表明, 单目减符号(即在一个表达式之前的负号)具有优先级UNARYMINUS(高), 执行的动作是改变值的符号。而两个表达式之间的负号仍采用原来的优先级。

练习8-2 把操作符%(求模或取余)和单目加添加到hoc1中。建议参考frexp(3)。

8.1.5 关于make

为了编译hoc1的新版本, 每次都必须键入两条命令, 这是极不方便的。虽然可以很容易地创建一个shell文件来完成这件工作, 但这里有一个更好的办法, 它尤其适用于程序中源文件不止一个的情形。程序make读入一个程序说明, 该说明指出了程序各部分之间相互依赖的关系, 以及如何处理它们以生成该程序的最新版本。make检查不同部分最后被修改的时间, 确定为创建相应的最新版本所需编译的最小文件集合, 并编译之。同时, make也能够处理如yacc这样复杂的多步程序, 因此任务可以放入一个make说明而不用写出每一步的过程。

当程序相当大, 大到有多个源文件时, make最为适用, 但它也可以用于像hoc1这样的小程序。下面是hoc1的make说明, 通常称之为makefile。

```

$ cat makefile
hoc1:  hoc.o
      cc hoc.o -o hoc1
$

```

这意味着hoc1依赖于hoc.o, 而通过运行C编译器cc将hoc.o转换为hoc1。make已经知道如何将yacc源文件hoc.y转换成目标文件hoc.o:

```

$ make                                     编译makefile里的第一个目标hoc1
yacc  hoc.y
cc  -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cc hoc.o -o hoc1
$ make                                     再做一次
`hoc1' is up to date                       make认识到不必再做一遍
$

```

8.2 第二阶段: 变量和错误恢复

下一步(简单的一步)便是为hoc1增加“内存”, 新程序名为hoc2。该内存分配了26个变量, 分别命名为a到z。这当然不十分确切, 但它可作为一个简便有用的中间步骤。此外还要加入一些错误处理。用户在使用hoc1时会发现, hoc1对错误的处理方式就是打印出一个信息并退出, 它对算术错误, 如类似除以0这样的错的处理更是不合理的:

```

$ hoc1
1/0
Floating exception - core dumped
$

```

hoc2的新特点引起的改动不大, 大约需35行左右的代码。词法分析器yylex需要将字母识别为变量, 语法分析器则需包括下列形式的产生式:

```

expr:      VAR
        | VAR '=' expr

```

一个表达式可以包含一个赋值，甚至允许下列多重赋值

```
x = y = z = 0
```

保存变量值最简单的方法是将值存入一个26个元素的数组中，而单字母变量名可作为该数组的下标。但若语法分析要同时处理栈里的变量名和值，yacc就需被告知它的栈里包含了双精度型和整型的联合类型，而不止是双精度型。这可以在程序头部加入一条%union定义来实现。若只想将栈的类型置为某种数据类型，如双精度型，那么一个#define或typedef语句就足够了，但要想使用联合类型，则%union是必须的，因为yacc要检查形如\$\$=\$2这样的表达式的一致性。

下面是hoc2的hoc.y语法分析部分：

```

$ cat hoc.y
%{
double  mem[26];          /* memory for variables 'a'...'z' */
%}
%union {
    double  val;          /* actual value */
    int      index;       /* index into mem[] */
}
%token <val>  NUMBER
%token <index> VAR
%type <val>  expr
%right '='
%left '+' '-'
%left '*' '/'
%left UNARYMINUS
%%
list:      /* nothing */
        | list '\n'
        | list expr '\n'      { printf("\t%.8g\n", $2); }
        | list error '\n'     { yyerror; }
        ;
expr:      NUMBER
        | VAR                { $$ = mem[$1]; }
        | VAR '=' expr       { $$ = mem[$1] = $3; }
        | expr '+' expr      { $$ = $1 + $3; }
        | expr '-' expr      { $$ = $1 - $3; }
        | expr '*' expr      { $$ = $1 * $3; }
        | expr '/' expr      {
            if ($3 == 0.0)
                execerror("division by zero", "");
            $$ = $1 / $3; }
        | '(' expr ')'       { $$ = $2; }
        | '-' expr %prec UNARYMINUS { $$ = -$2; }
        ;
%%
/* end of grammar */
...

```

%union定义该栈的元素可以用双精度型数，或整型数作为数组mem的下标；%token定义用一个类型指示作为参数；%type指定expr是union的<val>成员类型，即双精度型数。这样的类型信息使得yacc可以产生对union的成员类型的正确引用。同时也应注意，=是右结合的，而其他操作符是左结合的。

错误处理分成几个程序段，其中最重要的是检查除数为0的部分。如果出现除0错，则调用execerror出错例程。

其次是检查“浮点数溢出”信号，当一个浮点数发生溢出时产生该信息，它在main中设置。

错误恢复的最后部分是为error加入一个产生式。在yacc的语法分析器中，error是一个保留字，它提供了预测和恢复一个语法错误的方法。一个错误发生时，yacc将使用该产生式，识别错误并按语法上“正确”的处理，从而从错误中恢复。动作yyerrok设置分析器中的一个标志，该标志允许分析器返回到正常的分析状态。对任何分析器而言，错误恢复都是很困难的。现在对hoc2所做的不过是一些最基本的步骤，然而也已大大超过了yacc的处理能力。

hoc2语法分析部分的动作无需做大的改动。在main里加入setjump，以保存状态，从错误中恢复。execerror与longjump匹配。(参见7.5节关于setjump和longjump的说明。)

```
...
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;

main(argc, argv)      /* hoc2 */
    char *argv[];
{
    int fpecatch();

    progname = argv[0];
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    yyparse();
}

execerror(s, t) /* recover from run-time error */
    char *s, *t;
{
    warning(s, t);
    longjmp(begin, 0);
}

fpecatch() /* catch floating point exceptions */
{
    execerror("floating point exception", (char *) 0);
}
```

让execerror调用abort(参见abort(3))能便于跟踪调试，由此能产生可被adb或sdb读取的内存转储。一旦被调试好，就可用longjump代替abort。

hoc2里的词法分析器有少许改动。加入了一个专门测试小写字母的语句，而且因为yyylval现在是联合类型，故在yylex返回之前必须设置正确的成员类型。下面是改动后的部分：

```
yylex()      /* hoc2 */
...
    if (c == '.' || isdigit(c)) { /* number */
        ungetc(c, stdin);
        scanf("%lf", &yyylval.val);
        return NUMBER;
    }
    if (islower(c)) {
        yyylval.index = c - 'a'; /* ASCII only */
        return VAR;
    }
...
}
```

再次提醒注意记号的类型(例如NUMBER)与其值的区别(例如3.1416)。

现在举例说明hoc2的新功能, 如变量和错误恢复:

```
$ hoc2
x = 355
      355
y = 113
      113
p = x/z
hoc2: division by zero near line 4          z未定义故而为0
x/y                                          错误恢复
      3.1415929
1e30 * 1e30                                溢出
hoc2: floating point exception near line 5
...
```

事实上, PDP-11机需要专门的处理来检测浮点数溢出错, 但在大多数机器上, hoc2会得到如上的运行结果。

练习8-3 增加一个功能, 使程序能记忆最近计算的值, 使得在进行一系列相关运算时, 该值无需被再次键入。一种解决方案是设置一个变量, 例如p(pervious)。

练习8-4 修改hoc, 使得分号可以作为一个表达式终止符, 与换行功能相同。

8.3 第三阶段: 任意变量名和内部函数

在hoc3中, 加入了一些新的功能及相应代码。程序主要的新特性是可以访问内部函数:

sin	cos	atan	exp	log	log10
sqrt	int	abs			

此外, 还添加了一个指数操作符[^], 它具有最高优先级, 而且是右结合的。

因为词法分析器必须处理不止一个字符的内部名称, 所以可以容易地将变量名扩展到任意长度。现在需要一个更复杂的符号表来保存这些变量, 一旦有了符号表, 程序就可以把变量和一些有用的常量装入表内, 例如:

PI	3.14159265358979323846	圆周率
E	2.71828182845904523536	自然对数的底
GAMMA	0.57721566490153286060	欧拉-Mascheroni常量
DEG	57.29577951308232087680	弧度比率
PHI	1.61803398874989484820	黄金分割率

这可使hoc3成为一个有用的计算器:

```
$ hoc3
1.5^2.3
      2.5410306
exp(2.3*log(1.5))
      2.5410306
sin(PI/2)
      1
atan(1)*DEG
      45
...
```

对程序的行为也作了一些调整。在hoc2里, 赋值表达式x=expr不仅仅赋值, 还会打印出

该值，因为hoc2打印所有的表达式：

```
$ hoc2
x = 2 * 3.14159
        6.28318
...

```

打印赋给变量的值

在hoc3中，赋值和表达式有明显的区别，只有表达式的值才会被打印：

```
$ hoc3
x = 2 * 3.14159
x
        6.28318
...

```

赋值：不打印任何值
表达式：
值被打印出来

作了这些改动以后的程序相当大(大约250行)，最好将它分成多个单独的文件，便于编辑和较快地编译。现在将其分成五个文件：

hoc.y	语法分析，main，yylex(和以前的一样)
hoc.h	全局数据结构，便于各文件引用
symbol.c	符号表例程：lookup、install
init.c	内部量和常量：init
math.c	对数学例程的接口：Sqrt、Log，等等

这就要求进一步学习如何组织多文件的C程序，以及make的用法。

现在简短回顾一下make。首先，考察符号表代码。一个符号有自己的名字、类型(变量或内部量)以及自己的值。若该符号为一个变量，它的值就应该为一个双精度型数；若该符号为一个内部量，则它的值是指向某函数的指针，该函数返回一个双精度型数值。这样的信息在hoc.y、symbol.c和init.c中都会用到。可以简单地做成三份副本，但这样一来当发生改变时，容易出错或有可能会漏掉修改其中的一份。因此，这里把这些公用信息放入头文件hoc.h中，只要在需要用到它的文件中包括它即可(后缀.h是习惯上的称呼，并不是强制性的)。此外，还需在makefile中加入表示这些文件依赖hoc.h的语句，以便hoc.h被修改后，重做必要的编译。

```
$ cat hoc.h
typedef struct Symbol { /* symbol table entry */
    char    *name;
    short   type;      /* VAR, BLTIN, UNDEF */
    union {
        double val;          /* if VAR */
        double (*ptr)();     /* if BLTIN */
    } u;
    struct Symbol *next; /* to link to another */
} Symbol;
Symbol *install(), *lookup();
$
```

类型UNDEF表明这是一个尚未赋值的变量。

通过Symbol里的next域，各符号被连接到了一起，成为一个链表。该链表局限于symbol.c，只有通过函数lookup和install才能访问它。这样在有必要时，比较容易修改符号表的组织(我们曾这样做过一次)。lookup函数在表中查找某个特定的名字，若找到则返回一个指向Symbol的指针；否则返回0。符号表采用顺序查找，这对交互式计算器已完全足够，因为变量是在分析时刻进行查找，而不是运行时刻。install函数将一个变量相应的量值和类型填入符号表表头。

emalloc调用malloc，它是标准的内存分配函数(参见malloc(3))，并检查分配结果。这三个函数组成了symbol.c的全部内容。文件y.tab.h由运行yacc -d产生，它包括一系列的#define语句，如NUMBER、VAR、BETIN，等等。

```
$ cat symbol.c
#include "hoc.h"
#include "y.tab.h"

static Symbol *symlist = 0; /* symbol table: linked list */

Symbol *lookup(s)          /* find s in symbol table */
char *s;
{
    Symbol *sp;

    for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
        if (strcmp(sp->name, s) == 0)
            return sp;
    return 0;              /* 0 ==> not found */
}

Symbol *install(s, t, d) /* install s in symbol table */
char *s;
int t;
double d;
{
    Symbol *sp;
    char *emalloc();

    sp = (Symbol *) emalloc(sizeof(Symbol));
    sp->name = emalloc(strlen(s)+1); /* +1 for '\0' */
    strcpy(sp->name, s);
    sp->type = t;
    sp->u.val = d;
    sp->next = symlist; /* put at front of list */
    symlist = sp;
    return sp;
}

char *emalloc(n)          /* check return from malloc */
unsigned n;
{
    char *p, *malloc();

    p = malloc(n);
    if (p == 0)
        execerror("out of memory", (char *) 0);
    return p;
}
$
```

文件init.c包括了对常量(如PI等)和指向内部量的指针的定义，它们通过函数init被填入符号表中。init由main调用。

```
$ cat init.c
#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double Log(), Log10(), Exp(), Sqrt(), integer();
static struct {          /* Constants */
    char    *name;
    double  cval;
} consts[] = {
    "PI",    3.14159265358979323846,
```

```

        "E",      2.71828182845904523536,
        "GAMMA",  0.57721566490153286060, /* Euler */
        "DEG",    57.29577951308232087680, /* deg/radian */
        "PHI",    1.61803398874989484820, /* golden ratio */
        0,        0
    };
    static struct {          /* Built-ins */
        char    *name;
        double  (*func)();
    } builtins[] = {
        "sin",  sin,
        "cos",  cos,
        "atan", atan,
        "log",  Log,      /* checks argument */
        "log10", Log10,   /* checks argument */
        "exp",  Exp,      /* checks argument */
        "sqrt", Sqrt,     /* checks argument */
        "int",  integer,
        "abs",  fabs,
        0,      0
    };
    init() /* install constants and built-ins in table */
    {
        int i;
        Symbol *s;

        for (i = 0; consts[i].name; i++)
            install(consts[i].name, VAR, consts[i].cval);
        for (i = 0; builtins[i].name; i++) {
            s = install(builtins[i].name, BLTIN, 0.0);
            s->u.ptr = builtins[i].func;
        }
    }
}

```

数据被保存在表中，而不是硬编码到代码中，这样表更容易读和修改。这些表被声明为静态的，使得它们仅在本文件内部可见，而不是整个程序中都可可见。稍后会谈到数学函数，如Log和Sqrt等。

在此基础上，就可以对语法分析器进行修改了。

```

$ cat hoc.y
%{
#include "hoc.h"
extern double Pow();
%}
%union {
    double val;      /* actual value */
    Symbol *sym;     /* symbol table pointer */
}
%token <val> NUMBER
%token <sym> VAR BLTIN UNDEF
%type <val> expr asgn
%right '='
%left '+' '-'
%left '*' '/'
%left UNARYMINUS
%right '^' /* exponentiation */
%%
list:      /* nothing */
| list '\n'
| list asgn '\n'
| list expr '\n'      { printf("\t%.8g\n", $2); }
| list error '\n'     { yyerrok; }

```

```

;
asgn:  VAR '=' expr { $$=$1->u.val=$3; $1->type = VAR; }
;
expr:  NUMBER
      | VAR { if ($1->type == UNDEF)
                execerror("undefined variable", $1->name);
                $$ = $1->u.val; }
      | asgn
      | BLTIN '(' expr ')' { $$ = (*($1->u.ptr))($3); }
      | expr '+' expr { $$ = $1 + $3; }
      | expr '-' expr { $$ = $1 - $3; }
      | expr '*' expr { $$ = $1 * $3; }
      | expr '/' expr {
          if ($3 == 0.0)
              execerror("division by zero", "");
          $$ = $1 / $3; }
      | expr '^' expr { $$ = Pow($1, $3); }
      | '(' expr ')' { $$ = $2; }
      | '-' expr %prec UNARYMINUS { $$ = -$2; }
;

%%
/* end of grammar */
...

```

该语法分析器现在具有用于赋值的asgn和expr。若有输入行如下：

```
VAR = expr
```

它是一个赋值，因此不打印任何值。注意这里很容易地就将指数运算及右结合加入到了语法分析中。

yacc的栈采用了和原来 %union不同的类型，它不再通过下标引用26个元素数组中的变量，而是一个指向符号类型的指针。头文件hoc.h包含了这种类型的定义。

词法分析器识别出变量名，在符号表中进行查找，判断它是变量还是内部量，yylex返回其中一种类型。用户定义的变量和预定义变量(如PI)均为变量。

变量的一个特性是指明它是否已被赋值，这样使用一个未赋值的变量时yyparse就会报告出错。测试变量是否被定义是在语法分析而不是词法分析中进行的。当一个变量在词法上被识别时，它的上下文环境是未知的；若其环境完全合乎语法，例如x出现在赋值x=1的左边时，就认为x是有定义的。

下面是yylex修改过的部分：

```

yylex()          /* hoc3 */
...
    if (isalpha(c)) {
        Symbol *s;
        char sbuf[100], *p = sbuf;
        do {
            *p++ = c;
        } while ((c=getchar()) != EOF && isalnum(c));
        ungetc(c, stdin);
        *p = '\0';
        if ((s=lookup(sbuf)) == 0)
            s = install(sbuf, UNDEF, 0.0);
        yylval.sym = s;
        return s->type == UNDEF ? VAR : s->type;
    }
...

```

在main中增加了一行，调用初始化函数init以将内部量和预定义变量(如PI)填入到符号

表中。

```
main(argc, argv)      /* hoc3 */
    char *argv[];
{
    int fpecatch();

    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    yyparse();
}
```

现在只剩下了math.c未讨论。某些标准数学函数需要一个错误检查接口，发送错误信息，并从错误中恢复——例如，标准函数 sqrt 的参数若为负数，则返回0。math.c里的代码利用了错误测试程序，这比用户自己动手写的程序更可靠，可移植性更好，可以认为在“正式的”代码中程序的特定限制都已得到最充分反映。头文件<math.h>包括标准数学函数所需要的类型定义；<errno.h>包括各种可能的错误名称。

```
$ cat math.c
#include <math.h>
#include <errno.h>
extern int      errno;
double  errcheck();

double Log(x)
    double x;
{
    return errcheck(log(x), "log");
}
double Log10(x)
    double x;
{
    return errcheck(log10(x), "log10");
}
double Exp(x)
    double x;
{
    return errcheck(exp(x), "exp");
}
double Sqrt(x)
    double x;
{
    return errcheck(sqrt(x), "sqrt");
}
double Pow(x, y)
    double x, y;
{
    return errcheck(pow(x,y), "exponentiation");
}
double integer(x)
    double x;
{
    return (double)(long)x;
}
double errcheck(d, s)  /* check result of library call */
    double d;
    char *s;
{
    if (errno == EDOM) {
        errno = 0;
        execerror(s, "argument out of domain");
    }
}
```

```

    } else if (errno == ERANGE) {
        errno = 0;
        execerror(s, "result out of range");
    }
    return d;
}
$

```

若在新的语法下运行yacc, 会出现一些有趣的诊断信息(非语法方面的):

```

$ yacc hoc.y

conflicts: 1 shift/reduce
$

```

shift/reduce(移进/归约)信息表明hoc3的语法是有二义性的: 输入行

```
x = 1
```

可以用图8-2所示的两种方法进行分析:

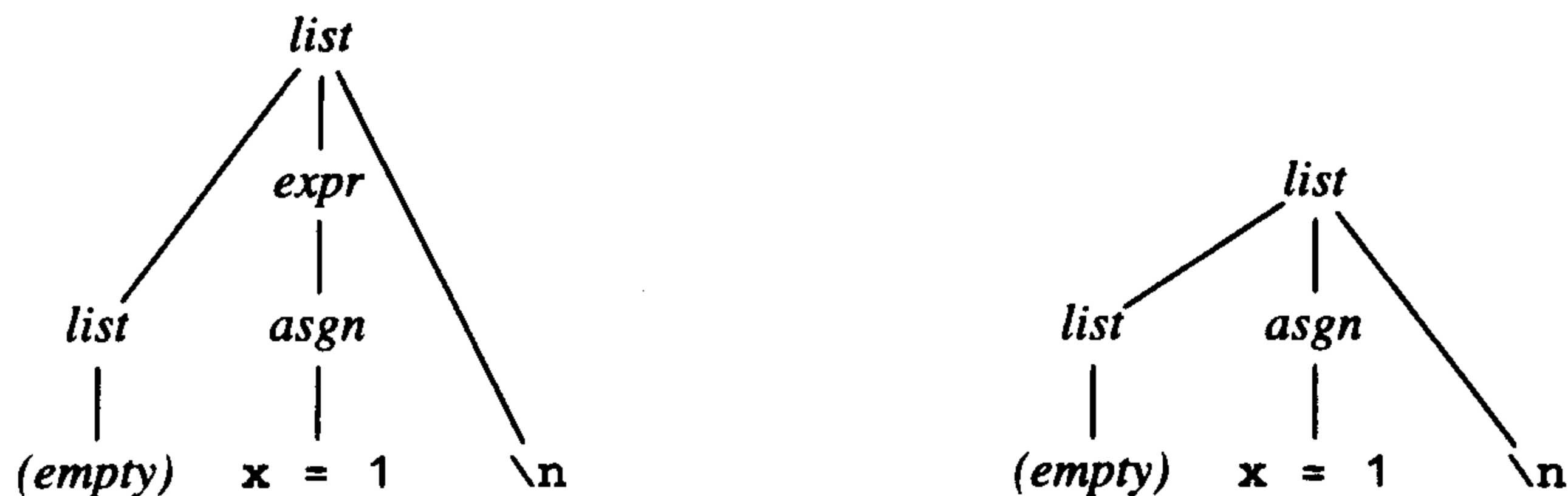


图8-2 归约与移进

分析器可决定采用哪种方法: 一种是asgn归约为expr然后归约为list, 如左边的分析树所示; 另一种是采用后续的\n(“移进”), 而将整个句子不用中间规则直接归约到list, 如右边的分析树所示。对于二义性, yacc选择移进, 因此实际的语法含义是右边的分析树。你应尽可能理解这一信息, 以保证yacc做出了正确的决定[⊖]。用-v选项运行yacc, 会产生一个名为y.output的很大的文件, 它在产生二义性的地方给出了提示。

练习8-5 根据hoc3, 下式是合法的:

```
PI = 3
```

这样好吗? 请修改hoc3, 以禁止对“常量”赋值。

练习8-6 增加内部函数atan2(y, x), 它返回正切值为y/x的角度。增加内部函数rand(), 它返回一个浮点随机变量, 其值在(0,1)区间上均匀分布。如何修改语法, 使得程序能够处理带不同数目参数的内部函数?

练习8-7 如何在hoc内加入一段执行命令的程序, 类似于其他UNIX程序的! 的功能?

练习8-8 修改在math.c的代码, 用一个表代替基本等价的函数的集合。

8.3.1 再谈make

因为hoc3的程序分散到了5个文件里, 它的makefile也就更为复杂:

[⊖] yacc的信息reduce/reduce conflict指出了一个问题, 多半是有直接的语法错误而不是故意的二义性。

```

$ cat makefile
YFLAGS = -d          # force creation of y.tab.h
OBJS = hoc.o init.o math.o symbol.o    # abbreviation

hoc3:  $(OBJS)
       cc $(OBJS) -lm -o hoc3

hoc.o:  hoc.h

init.o symbol.o:      hoc.h y.tab.h

pr:
       @pr hoc.y hoc.h init.c math.c symbol.c makefile

clean:
       rm -f $(OBJS) y.tab.[ch]

$

```

YFLAGS = -d语句把选项-d加到了make生成的yacc命令行中，它通知yacc产生包含#define语句的y.tab.h文件。OBJS=…语句定义了一个结构的缩写，该结构在后面会被反复使用多次。这里的语法与shell变量不同，在引用时必须加括号。标志-lm指示在调用数学函数时，需查找数学库。

hoc3依赖于四个.o文件，其中有些.o文件又依赖于.h文件。考虑到这些依赖关系，当其中某些文件被修改后，make可以判断出需要重做的编译工作。若只想观察一下make的流程，而不实际执行make，可键入

```
$ make -n
```

另一方面，若想要使文件的时间保持一致，用-t(touch)选项就可以修改它们，而不用做任何编译。

注意到文件中不仅增加了一组源文件的依赖关系，还添加了许多各种各样的应用例程，它们都封装到了一起。缺省情况下，make作用于makefile中所列的第一个目标，但假如指明了某个标识依赖规则的项目，如symbol.o或pr，那么它将被首先处理。空的依赖表示该项目总是过期的，因此一旦有请求，总要执行规定动作。所以

```
$ make pr / lpr
```

将在打印机上输出用户所要的文本。(@pr中的@表示make执行的命令不作回送。)而且

```
$ make clean
```

将清除yacc的输出文件和.o文件。

makefile中的空依赖机制往往比在一个shell文件里放入所有相关计算要好。make的应用也不仅限于程序开发——它可以将任何在时间上有相互依赖关系的操作集合封装在一起。

8.3.2 关于lex

程序lex的功能是以类似yacc产生语法分析器的方式产生词法分析器：使用正则表达式和C程序段来输入语言的词法规则说明，以便当找到一个匹配字符串时执行相应的表达式和C程序段。lex把这些规则转换成一个识别程序。lex和yacc结合起来使用，它在这里的作用同前面所讨论的词法分析器一样。这里不打算对lex进行深入详细的探讨，下面的讨论是为了吸引读者，进行进一步的学习。请参见《UNIX程序员手册》2B卷中关于lex的说明。

首先来看一个lex程序，它放在文件lex.l中，可替代以前所用的函数yylex:

```
$ cat lex.l
%{
#include "hoc.h"
#include "y.tab.h"
extern int lineno;
%}
%%
[ \t] { ; } /* skip blanks and tabs */
[0-9]+\.[0-9]*[0-9]+ {
    sscanf(yytext, "%lf", &yylval.val); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* {
    Symbol *s;
    if ((s=lookup(yytext)) == 0)
        s = install(yytext, UNDEF, 0.0);
    yyval.sym = s;
    return s->type == UNDEF ? VAR : s->type; }
\n { lineno++; return '\n'; } /* everything else */
. { return yytext[0]; }
$
```

和egrep或awk中的类似，每一条“规则”都是一个正规表达式。只是lex还能识别C语言风格的符号，如\t和\n。其动作被括在花括号里。规则会试图顺序处理形如*和+匹配结构的、任意长度的字符串。如果规则匹配了输入的后续部分，相应的动作会被完成。被匹配的字符串放入lex中的字符串yytext中。

为了使用lex，makefile需改动如下:

```
$ cat makefile
YFLAGS = -d
OBJS = hoc.o lex.o init.o math.o symbol.o

hoc3: $(OBJS)
    cc $(OBJS) -lm -ll -o hoc3

hoc.o: hoc.h

lex.o init.o symbol.o: hoc.h y.tab.h
...
$
```

make知道如何从.l文件中得到正确的.o文件，它只需要获得依赖信息。(这里也将lex库-ll加到了cc的搜索列表中，因为lex产生的识别程序不是自持的。)输出完全是自动的:

```
$ make
yacc -d hoc.y

conflicts: 1 shift/reduce
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
lex lex.l
cc -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc -c init.c
cc -c math.c
cc -c symbol.c
cc hoc.o lex.o init.o math.o symbol.o -lm -ll -o hoc3
$
```

若单个文件被修改了，仅使用make就可以生成一个最新的版本:

```

$ touch lex.1          改变文件lex.1的最近修改时间
$ make
lex lex.1
cc -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc hoc.o lex.o init.o math.o symbol.o -ll -lm -o hoc3
$

```

对于lex有两种观点，一种认为lex是额外的工具，可简要说明一下，不用也罢；另一种则认为一旦语言变得复杂起来，lex是生成词法分析器的首选工具。两种说法各有道理。lex的主要问题(除了需要用户学习另一种语言之外)是，它运行可能较慢，而且与等价的C版本相比生成了更大更慢的识别器。如果发现异常，比如在错误恢复甚至是从文件输入时，它都较难调整其输入机制。这些问题就hoc而言都不严重，主要的问题是这里在描述lex时花费了太多的页面，所以(很抱歉)本章将会转向C进行实际的词法分析。不过，制作lex版本仍是不错的练习手段。

练习8-9 比较hoc3的两个版本的大小。提示：参阅size(1)。

8.4 第四阶段：编译成机器

下面将谈到hoc5，它是一个带有控制流的语言解释器。hoc4是一个中间产品，它具有同hoc3一样的功能，但是是用hoc5的解释框架实现的。之所以采用这种方式编写hoc4是因为它提供了两个程序，它们完成相同的工作，对于调试非常有用。在分析输入的同时，hoc4生成适应于简单计算机的代码而不立即计算结果。一旦语句输入完毕，生成的代码被执行(“被解释”)以计算结果。

这种简单计算机是堆栈机器：当遇到一个操作数时，把它压入到堆栈中(更精确地说，生成了把它压入堆栈的代码)。大多数运算符对栈顶的元素进行操作。例如，处理语句

```
x = 2 * y
```

将生成下面的代码：

constpush	把一个常量压入堆栈
2	...常量2
varpush	把变量压入堆栈
y	...变量y
eval	计算：用值代替指针
mul	用栈顶两个元素的乘积代替这两个元素
varpush	把变量压入堆栈
x	...变量x
assign	将值保存到变量并弹出指针
POP	栈顶值出栈
STOP	指令序列结束

当这些代码被执行时，表达式被赋值并且结果被存储到x中。执行过程由注释说明。最后一次弹出清除堆栈的值，因为它已经没用了。

堆栈机器通常导致产生简单的解释器，而这里也不例外——它只是包含操作符和操作数的数组，操作符是机器指令，每一条是带参数的(如果有参数)函数调用，随后是指令。其他的操作数可以是已经存放到堆栈中的，就如上面的例子所述。

hoc4的符号表代码、init.c的初始化和math.c的数学函数与hoc3一模一样，语法也一样，但动作截然不同。基本上，每个动作生成机器指令及其参数。例如，在处理表达式中的变量

时，生成三条机器指令：varpush指令、指向变量的符号表指针以及eval指令。eval指令的作用是，执行时用符号表指针所指的值得代替该指针。因为操作数已经在堆栈中，对应 * 的代码就是mul。

```
$ cat hoc.y
%{
#include "hoc.h"
#define code2(c1,c2)    code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
    Symbol *sym; /* symbol table pointer */
    Inst *inst; /* machine instruction */
}
%token <sym>    NUMBER VAR BLTIN UNDEF
%right '='
%left '+' '-'
%left '*' '/'
%left UNARYMINUS
%right '^' /* exponentiation */
%%
list: /* nothing */
    | list '\n'
    | list asgn '\n' { code2(pop, STOP); return 1; }
    | list expr '\n' { code2(print, STOP); return 1; }
    | list error '\n' { yyerrok; }
    ;
asgn:  VAR '=' expr { code3(varpush, (Inst)$1, assign); }
    ;

expr:  NUMBER      { code2(constpush, (Inst)$1); }
    | VAR          { code3(varpush, (Inst)$1, eval); }
    | asgn
    | BLTIN '(' expr ')' { code2(bltin, (Inst)$1->u.ptr); }
    | '(' expr ')'
    | expr '+' expr { code(add); }
    | expr '-' expr { code(sub); }
    | expr '*' expr { code(mul); }
    | expr '/' expr { code(div); }
    | expr '^' expr { code(power); }
    | '-' expr %prec UNARYMINUS { code(negate); }
    ;

%%
/* end of grammar */
...
```

Inst是机器指令的一种数据类型(函数返回值为整型的指针)，不久将再谈到它。请注意code的参数是函数名，即函数指针，或者硬性规定的函数指针值。

main有一些变化。现在，在每一条语句或表达式后，分析器都将返回，生成的代码被执行。yyparse在文件结束处返回值0。

```
main(argc, argv) /* hoc4 */
    char *argv[];
{
    int fpecatch();

    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    for (initcode(); yyparse(); initcode())
        execute(prog);
}
```

```

    return 0;
}

```

词法分析器只有一小部分不同。主要的变化是数字必须被保存，而不是立即使用。最容易的方法是把它们同变量一样保存在符号表中。下面是yylex的改变部分：

```

yylex()          /* hoc4 */
...
    if (c == '.' || isdigit(c)) { /* number */
        double d;
        ungetc(c, stdin);
        scanf("%lf", &d);
        yylval.sym = install("", NUMBER, d);
        return NUMBER;
    }
...

```

解释器堆栈中的每个元素或者是浮点型数值，或者是指向符号表项的指针，堆栈数据类型是这些类型组成的联合类型。机器本身也是指针的数组，这些指针可以指向完成具体操作的程序，如mul等，或者指向符号表中的数据。头文件hoc.h也必须包含解释器的数据结构和函数定义，以保证整个程序中它们能够被识别。（所有这些信息被放在一个文件，而不是两个文件中。在较大的程序中，把头文件信息分解为多个文件会更好。这样，每个文件仅在真正需要时才会被包含进去。）

```

$ cat hoc.h
typedef struct Symbol { /* symbol table entry */
    char    *name;
    short   type; /* VAR, BLTIN, UNDEF */
    union {
        double val; /* if VAR */
        double (*ptr)(); /* if BLTIN */
    } u;
    struct Symbol *next; /* to link to another */
} Symbol;
Symbol *install(), *lookup();

typedef union Datum { /* interpreter stack type */
    double val;
    Symbol *sym;
} Datum;
extern Datum pop();

typedef int (*Inst)(); /* machine instruction */
#define STOP (Inst) 0

extern Inst prog[];
extern eval(), add(), sub(), mul(), div(), negate(), power();
extern assign(), bltin(), varpush(), constpush(), print();
$

```

执行机器指令和操作堆栈的例程被保存到一个新的文件code.c中，它有150行左右，因此这里将它分成多个部分。

```

$ cat code.c
#include "hoc.h"
#include "y.tab.h"

#define NSTACK 256
static Datum stack[NSTACK]; /* the stack */
static Datum *stackp; /* next free spot on stack */

#define NPROG 2000

```

```

Inst    prog[NPROG];    /* the machine */
Inst    *progp;          /* next free spot for code generation */
Inst    *pc;             /* program counter during execution */

initcode()    /* initialize for code generation */
{
    stackp = stack;
    progp = prog;
}
...

```

栈的操作通过调用push和pop来完成。

```

push(d)      /* push d onto stack */
Datum d;
{
    if (stackp >= &stack[NSTACK])
        execerror("stack overflow", (char *) 0);
    *stackp++ = d;
}

Datum pop()   /* pop and return top elem from stack */
{
    if (stackp <= stack)
        execerror("stack underflow", (char *) 0);
    return *--stackp;
}

```

在分析过程中，通过调用函数code生成机器，函数code只是把一条指令放到数组prog的下一个空闲单元。它返回这条指令的地址(hoc4中没有使用)。

```

Inst *code(f) /* install one instruction or operand */
Inst f;
{
    Inst *oprogp = progp;
    if (progp >= &prog[NPROG])
        execerror("program too big", (char *) 0);
    *progp++ = f;
    return oprog;
}

```

机器的执行很简单，实际上，它不仅小而且很利索，一旦被建立就可以“运行”。

```

execute(p)    /* run the machine */
Inst *p;
{
    for (pc = p; *pc != STOP; )
        ((*pc++))();
}

```

每次循环执行一个函数，该函数由一个指令指向，而该指令由程序计数器pc指向，并且pc的增量已指向下一条指令。带操作码STOP的指令终止循环。诸如constpush和varpush等指令，也增量pc值，用以跳过紧接在指令后的参数。

```

constpush()   /* push constant onto stack */
{
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d);
}

varpush()     /* push variable onto stack */
{
    Datum d;

```

```

        d.sym = (Symbol *)(*pc++);
        push(d);
    }

```

其他部分也同样简单。比如，算术操作都是类似的，可通过对一个原型的编辑来构造。

下面是add:

```

add()          /* add top two elems on stack */
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push(d1);
}

```

其余的函数同样简单。

```

eval()         /* evaluate variable on stack */
{
    Datum d;
    d = pop();
    if (d.sym->type == UNDEF)
        execerror("undefined variable", d.sym->name);
    d.val = d.sym->u.val;
    push(d);
}
assign()       /* assign top value to next value */
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
                    d1.sym->name);
    d1.sym->u.val = d2.val;
    d1.sym->type = VAR;
    push(d2);
}
print()        /* pop top value from stack, print it */
{
    Datum d;
    d = pop();
    printf("\t%.8g\n", d.val);
}
bltin()        /* evaluate built-in on top of stack */
{
    Datum d;
    d = pop();
    d.val = (*(double (*)(double)))(*pc++))(d.val);
    push(d);
}

```

最难的部分是bltin中的计算，它把*pc计算为“返回值为双精度的函数指针”，且该函数执行时把d.val作为参数使用。

如果每部分都执行正确，eval和assign中的错误诊断不会执行，保留它们是为了避免程序错误导致堆栈堵塞。它们在时间与空间上的开销，都比检测由于在程序上的粗心改动而引起的错误要小。

C操作函数指针的能力对于压缩代码和提高代码有效性非常有效。此外还有一种办法，固定操作符并将语义函数合并到一个大的switch语句中。这项工作很容易，留作练习。

make讨论之三

随着hoc源代码的增加,保持对改变以及依赖于这种改变的部分的跟踪越来越重要。make的优点就是能自动完成这些任务。否则程序员得亲手做(有时还会做错),或者建立一个专门的解释文件。

这里对makefile做了两个改进。第一个是建立在如下基础上:尽管许多文件依赖于由yacc定义的、放在y.tab.h中的常量,但除非这些常量改变了,否则没有必要重新编译这些文件——hoc.y中C代码的改变不会影响到别的部分。新的makefile,即.o文件依赖于一个新文件x.tab.h,当y.tab.h对应的常量改变时,x.tab.h才会被修改。第二个改进是制定pr(printing the source files)的规则,pr依赖于源文件,因此,只有改变的文件才被打印。

这一改进对于大型的程序节约了大量时间,因为语法是静态而语义不是(通常情形下)。第二个改进是节约了大量纸张。

下面是一个新的hoc4的makefile:

```
YFLAGS = -d
OBSJS = hoc.o code.o init.o math.o symbol.o

hoc4: $(OBSJS)
    cc $(OBSJS) -lm -o hoc4

hoc.o code.o init.o symbol.o: hoc.h

code.o init.o symbol.o: x.tab.h

x.tab.h: y.tab.h
    -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h

pr: hoc.y hoc.h code.c init.c math.c symbol.c
    @pr $?
    @touch pr

clean:
    rm -f $(OBSJS) [xy].tab.[ch]
```

cmp前的符号-的作用是:即使cmp失败,make也继续执行。这样一来,即使x.tab.h不存在,操作仍能进行。(选项-s的作用是:使cmp只设置退出状态而不产生输出。)符号\$?通过规则列出还未更新的元素。遗憾的是,make的记号规则最多只能部分类似于shell。

下面举例说明这些操作,假设所有一切都是更新过的。

```
$ touch hoc.y                改变hoc.y的日期
$ make
yacc -d hoc.y

conflicts: 1 shift/reduce
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cc hoc.o code.o init.o math.o symbol.o -lm -o hoc4
$ make -n pr                  打印修改过的文件
pr hoc.y
touch pr
$
```

注意,除hoc.y外,没有任何文件需要重新编译,因为y.tab.h和以前一模一样。

练习8-10 动态设置stack和prog的大小,保证hoc4不会越界,假设可通过调用malloc获得

内存。

练习8-11 修改hoc4, 使其在执行中对操作类型使用switch语句, 而不是调用函数。这样在源代码大小和执行速度上有何区别? 在维护及发展上又将怎样?

8.5 第五阶段: 控制流和关系运算符

版本hoc5吸取了解释器的优点。像C一样, 它包含if-else和while语句、用{和}括起来的复合语句以及打印语句。还有一套关系运算符(>、>=等)以及AND和OR操作符&&和||。(最后两个操作符不一定从左向右计算, 这与C是一样的。它们总要计算两个条件, 即使没有必要这样做。)

由于增加了记号、非终端以及if、while、分支和关系运算符等情况, 语法也随之增加, 并且非常长, 但并不是很复杂(除了if和while外):

```
$ cat hoc.y
%{
#include "hoc.h"
#define code2(c1,c2)    code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
    Symbol *sym;    /* symbol table pointer */
    Inst *inst;    /* machine instruction */
}
%token <sym>    NUMBER PRINT VAR BLTIN UNDEF WHILE IF ELSE
%type <inst>    stmt asgn expr stmtlist cond while if end
%right '='
%left OR
%left AND
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/'
%left UNARYMINUS NOT
%right '^'
%%
list:    /* nothing */
        | list '\n'
        | list asgn '\n' { code2(pop, STOP); return 1; }
        | list stmt '\n' { code(STOP); return 1; }
        | list expr '\n' { code2(print, STOP); return 1; }
        | list error '\n' { yyerrok; }
        ;
asgn:    VAR '=' expr { $$=$3; code3(varpush,(Inst)$1,assign); }
        ;
stmt:    expr { code(pop); }
        | PRINT expr { code(preexpr); $$ = $2; }
        | while cond stmt end {
            ($1)[1] = (Inst)$3;    /* body of loop */
            ($1)[2] = (Inst)$4; } /* end, if cond fails */
        | if cond stmt end { /* else-less if */
            ($1)[1] = (Inst)$3;    /* thenpart */
            ($1)[3] = (Inst)$4; } /* end, if cond fails */
        | if cond stmt end ELSE stmt end { /* if with else */
            ($1)[1] = (Inst)$3;    /* thenpart */
            ($1)[2] = (Inst)$6;    /* elsepart */
            ($1)[3] = (Inst)$7; } /* end, if cond fails */
        | '{' stmtlist '}' { $$ = $2; }
        ;
cond:    '(' expr ')' { code(STOP); $$ = $2; }
        ;
```

```

;
while:  WHILE { $$ = code3(whilecode, STOP, STOP); }
;
if:     IF     { $$=code(ifcode); code3(STOP, STOP, STOP); }
;
end:    /* nothing */          { code(STOP); $$ = prog; }
;
stmtlist: /* nothing */      { $$ = prog; }
| stmtlist '\n'
| stmtlist stmt
;
expr:   NUMBER      { $$ = code2(constpush, (Inst)$1); }
| VAR              { $$ = code3(varpush, (Inst)$1, eval); }
| asgn
| BLTIN '(' expr ')'
|          { $$ = $3; code2(bltin, (Inst)$1->u.ptr); }
| '(' expr ')' { $$ = $2; }
| expr '+' expr { code(add); }
| expr '-' expr { code(sub); }
| expr '*' expr { code(mul); }
| expr '/' expr { code(div); }
| expr '^' expr { code(power); }
| '-' expr %prec UNARYMINUS { $$ = $2; code(negate); }
| expr GT expr { code(gt); }
| expr GE expr { code(ge); }
| expr LT expr { code(lt); }
| expr LE expr { code(le); }
| expr EQ expr { code(eq); }
| expr NE expr { code(ne); }
| expr AND expr { code(and); }
| expr OR expr { code(or); }
| NOT expr      { $$ = $2; code(not); }
;
%%

```

和hoc3中的一样，这个语法有5处移进/归约冲突。

注意在很多地方都用STOP指令用来中止程序。和以前一样，prog存放着下一条执行指令的地址。执行过程中，STOP将会中止循环部分。end结束部分实际是一个子程序，并在多处被调用。这个子程序有STOP指令，并返回调用者的下一条语句。

为while和if生成的代码需要特别讨论。当遇到关键字while时，就生成whilecode操作，而其在机器中的位置则作为结果值返回：

```
While: WHILE
```

同时，随后的两处位置则保留，以备今后再填写。下一条要生成的代码是while的条件部分。cond的返回值是条件代码的开始位置。当整个while语句读完后，whilecode保留的两个位置被填写，一个是循环体的地址，一个是while语句的下一条语句地址。（这条语句的代码紧接着被生成。）

```

| while cond stmt end {
    ($1)[1] = (Inst)$3;    /* body of loop */
    ($1)[2] = (Inst)$4; } /* end, if cond fails */

```

\$1是whilecode的地址，因此(\$1)[1]和(\$1)[2]就是其后的两个地址。

用图来描述会显得更加清晰，参见图8-3。

if和while相似，不过有三个位置需要保留，分别对应then、else和if的下一条语句，我们很快就会看到其操作方式。

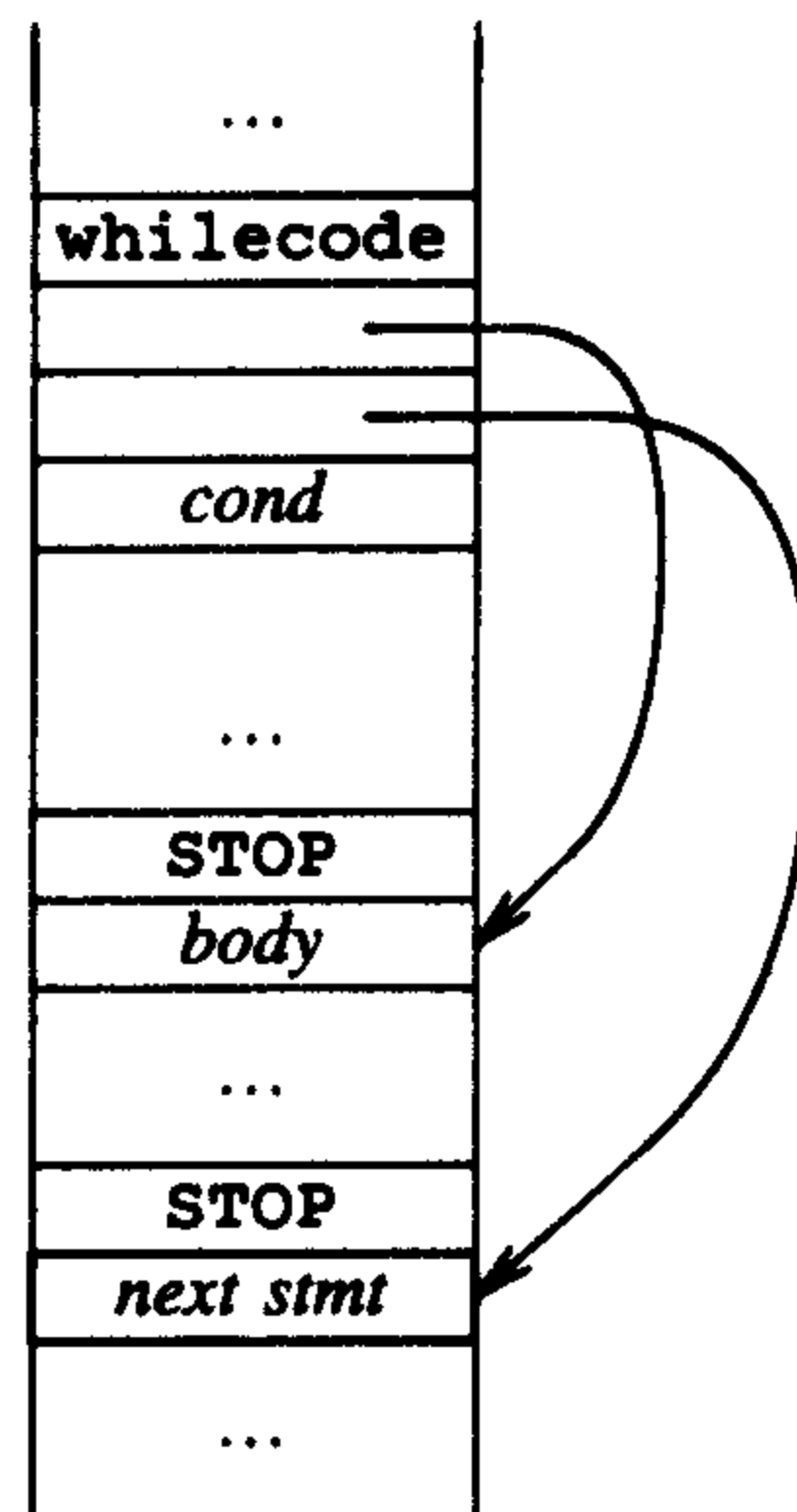


图8-3 whilecode及其后两个地址

词法分析会变得稍长一些，主要因为加入了额外的操作符：

```

yylex()          /* hoc5 */
...
    switch (c) {
    case '>':      return follow('=', GE, GT);
    case '<':      return follow('=', LE, LT);
    case '=':      return follow('=', EQ, '=');
    case '!':      return follow('=', NE, NOT);
    case '|':      return follow('|', OR, '|');
    case '&':      return follow('&', AND, '&');
    case '\n':     lineno++; return '\n';
    default:       return c;
    }
}

```

follow查看前一个字符，如果这个字符并不是所期望的，就通过ungetc放回到输入流中。

```

follow(expect, ifyes, ifno) /* look ahead for >=, etc. */
{
    int c = getchar();

    if (c == expect)
        return ifyes;
    ungetc(c, stdin);
    return ifno;
}

```

hoc.h中有更多的函数定义——例如所有的关系操作——但它们与hoc4在想法上是一致的。

这是最后的一部分：

```

$ cat hoc.h
...
typedef int (*Inst)(); /* machine instruction */
#define STOP (Inst) 0

extern Inst prog[], *progp, *code();
extern eval(), add(), sub(), mul(), div(), negate(), power();
extern assign(), bltin(), varpush(), constpush(), print();
extern preexpr();
extern gt(), lt(), eq(), ge(), le(), ne(), and(), or(), not();
extern ifcode(), whilecode();
$

```

code.c的大部分代码也是一样的，尽管增加了许多处理关系运算的例程。函数 le(less than or equal to)就是一个典型的例子。

```
le()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val <= d2.val);
    push(d1);
}
```

较为难懂的两个例程是whilecode和ifcode。要真正理解它们，必须认识到execute是按一串指令流进行的，直至遇到STOP，然后返回。在分析的同时，代码生成就在每个指令流中安排了一个STOP，而这是由一个简单的execute调用处理的。while的循环体、条件部分，if的then和else是通过对execute的递归调用处理的，即某一调用完成任务后又返回上一层。递归任务的控制是由直接对应while和if的whilecode和ifcode完成的。

```
whilecode()
{
    Datum d;
    Inst *savepc = pc;      /* loop body */

    execute(savepc+2);      /* condition */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc))); /* body */
        execute(savepc+2);
        d = pop();
    }
    pc = *((Inst **)(savepc+1)); /* next statement */
}
```

回想前面所谈的，whilecode的操作有两个指针，一个指向循环体，一个指向下一条语句，然后是条件部分。当调用whilecode时，pc已经增量了，因此它指向循环体指针。而pc+1指向下一条语句，pc+2指向条件。

ifcode非常相似：pc指向then部分，pc+1指向else，pc+2指向下一条语句，而pc+3则指向条件。

```
ifcode()
{
    Datum d;
    Inst *savepc = pc;      /* then part */

    execute(savepc+3);      /* condition */
    d = pop();
    if (d.val)
        execute(*((Inst **)(savepc)));
    else if (*((Inst **)(savepc+1))) /* else part? */
        execute(*((Inst **)(savepc+1)));
    pc = *((Inst **)(savepc+2)); /* next stmt */
}
```

init.c中的初始化代码也相应增加，关键字表和其他部分一起存储在符号表中。

```
$ cat init.c
...
static struct {          /* Keywords */
    char    *name;
    int     kval;
```

```

} keywords[] = {
    "if",          IF,
    "else",        ELSE,
    "while",       WHILE,
    "print",       PRINT,
    0,            0,
};
...

```

为存储关键字，init中需要增加一个循环。

```

for (i = 0; keywords[i].name; i++)
    install(keywords[i].name, keywords[i].kval, 0.0);

```

符号表管理不需要任何改动。code.c中包含例程prexpr，当遇到print expr这种形式的语句时被调用。

```

prexpr()          /* print numeric value */
{
    Datum d;
    d = pop();
    printf("%.8g\n", d.val);
}

```

这和打印计算结果时自动调用的打印函数不同，打印函数有弹栈操作并把Tab键加入到输出中去。

hoc5目前是一个相当有用的计算器，虽然对于严谨的编程来说，它需要增加更多的功能。下面的练习提供了关于hoc5的一些可能建议。

练习8-12 修改hoc5，使之可以识别形式打印它生成的机器，供程序调试用。

练习8-13 加入C赋值符，如+=、*=，等，以及递增、递减算符++和--。修改&&和||以保证自左向右计算，并能提前中止，就像C一样。

练习8-14 在hoc5中加入for语句，类似于C。加入break和continue。

练习8-15 如何修改hoc5语法或词法分析器(或两者)，使对换行的设置更灵活些？如何用分号代表换行？如何加入注解？使用什么句法？

练习8-16 加入一个中断，当hoc5的计算未完成时能被中止，但必须保存已计算变量的状态。

练习8-17 如果以文件的形式建立一个程序并运行它，此后再编辑文件时哪怕做很小的改动也是一件很麻烦的事。修改hoc5，提供一个编辑命令，它能装入hoc的程序副本。提示：考虑text操作码。

8.6 第六阶段：函数、过程和I/O

至少对此书而言，hoc发展的最后一步主要在于增加了新功能：函数和过程。至今已增加了打印字符串和数字的功能并能从标准输入中读取值。hoc6接受文件名作为参数，包括以-代替标准输入。所有的改动增加了235行代码，总代码长度达810行左右，而实际上就完成了从计算器到编程语言的转化。在此就不列出每一行代码了，附录C列出全部程序，可从那里看到各部分程序是如何配合的。

从语法上讲，函数调用的是表达式，而过程调用的是语句。附录B对它们作了详细的解释，还有更多的例子。例如，打印小于某数的斐波拉契(Fibonacci)数，其过程的定义和用法如下所示：

```

$ cat fib
proc fib() {
    a = 0
    b = 1
    while (b < $1) {
        print b
        c = b
        b = a+b
        a = c
    }
    print "\n"
}
$ hoc6 fib -
fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
...

```

这也说明了文件的用法：文件名-代表标准输入。

以下是一个阶乘函数：

```

$ cat fac
func fac() {
    if ($1 <= 0) return 1 else return $1 * fac($1-1)
}
$ hoc6 fac -
fac(0)
1
fac(7)
5040
fac(10)
3628800
...

```

在函数和过程中，通过\$1等引用参数，这与shell一样，但能对它们赋值。函数和过程是可递归的，但参数是局部变量。其他变量是全局变量，可在程序的任何地方引用。

hoc区分函数和过程，这样做可在实现堆栈时做到分层检测。程序员总是容易忘掉写return，或多加一个表达式，而把堆栈弄乱。

在从hoc5到hoc6的转化中，语法有多处变化，但都是局部的。hoc6的语法需要新的记号和非终端，并且%union定义了一个新成员以保留参数个数的统计：

```

$ cat hoc.y
...
%union {
    Symbol *sym; /* symbol table pointer */
    Inst *inst; /* machine instruction */
    int narg; /* number of arguments */
}
%token <sym> NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE
%token <sym> FUNCTION PROCEDURE RETURN FUNC PROC READ
%token <narg> ARG
%type <inst> expr stmt asgn prlist stmtlist
%type <inst> cond while if begin end
%type <sym> procname
%type <narg> arglist
...
list: /* nothing */
    | list '\n'
    | list defn '\n'
    | list asgn '\n' { code2(pop, STOP); return 1; }
    | list stmt '\n' { code(STOP); return 1; }
    | list expr '\n' { code2(print, STOP); return 1; }
    | list error '\n' { yyerrok; }

```

```

;
asgn:  VAR '=' expr { code3(varpush,(Inst)$1,assign); $$=$3; }
      | ARG '=' expr
        { defnonly("$"); code2(argassign,(Inst)$1); $$=$3; }
;
stmt:  expr { code(pop); }
      | RETURN { defnonly("return"); code(procret); }
      | RETURN expr
        { defnonly("return"); $$=$2; code(funcrct); }
      | PROCEDURE begin '(' arglist ')'
        { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
      | PRINT prlist { $$ = $2; }
...
expr:  NUMBER { $$ = code2(constpush, (Inst)$1); }
      | VAR { $$ = code3(varpush, (Inst)$1, eval); }
      | ARG { defnonly("$"); $$ = code2(arg, (Inst)$1); }
      | asgn
      | FUNCTION begin '(' arglist ')'
        { $$ = $2; code3(call,(Inst)$1,(Inst)$4); }
      | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
...
begin: /* nothing */ { $$ = progp; }
;
prlist: expr { code(prexp); }
      | STRING { $$ = code2(prstr, (Inst)$1); }
      | prlist ',' expr { code(prexp); }
      | prlist ',' STRING { code2(prstr, (Inst)$3); }
;
defn:  FUNC procname { $2->type=FUNCTION; indef=1; }
      | '(' ')' stmt { code(procret); define($2); indef=0; }
      | PROC procname { $2->type=PROCEDURE; indef=1; }
      | '(' ')' stmt { code(procret); define($2); indef=0; }
;
procname: VAR
        | FUNCTION
        | PROCEDURE
;
arglist: /* nothing */ { $$ = 0; }
        | expr { $$ = 1; }
        | arglist ',' expr { $$ = $1 + 1; }
;
%%
...

```

arglist统计参数个数。初看起来，好像必须以某种方式收集参数，但这并不需要，因为参数表中的每个expr都将值放到堆栈中，所以只需知道有多少个参数在堆栈中就可以了。

defn规则引入了yacc的一个新特性——嵌入的活动。在识别规则时，嵌入一个活动到规则中执行是可能的。使用这个性质可记录函数和过程定义。(也就是说建立起类似begin的新符号，它会在恰当的时候被识别。)如果结构出现在函数和过程的定义之外，defnonly打印一条警告信息。检测句法错误还是语义错误一般是可选择的，一般先处理未定义变量。defnonly函数是一个语义检查比语法检查更容易的典型例子。

```

defnonly(s) /* warn if illegal definition */
char *s;
{
    if (!indef)
        execerror(s, "used outside definition");
}

```

变量indef在hoc.y中定义，由defn的动作设置。

词法分析器增加了参数——\$后跟一数字——以及引号内的串的测试。像\n这样的反斜杠串由函数backslash解释。

```

yylex()          /* hoc6 */
...
    if (c == '$') { /* argument? */
        int n = 0;
        while (isdigit(c=getc(fin)))
            n = 10 * n + c - '0';
        ungetc(c, fin);
        if (n == 0)
            execerror("strange $...", (char *)0);
        yylval.narg = n;
        return ARG;
    }
    if (c == '"') { /* quoted string */
        char sbuf[100], *p, *emalloc();
        for (p = sbuf; (c=getc(fin)) != '"'; p++) {
            if (c == '\n' || c == EOF)
                execerror("missing quote", "");
            if (p >= sbuf + sizeof(sbuf) - 1) {
                *p = '\0';
                execerror("string too long", sbuf);
            }
            *p = backslash(c);
        }
        *p = 0;
        yylval.sym = (Symbol *)emalloc(strlen(sbuf)+1);
        strcpy(yylval.sym, sbuf);
        return STRING;
    }
...

backslash(c)      /* get next char with \'s interpreted */
    int c;
{
    char *index(); /* 'strchr()' in some systems */
    static char transtab[] = "b\bf\fn\nr\rt\t";
    if (c != '\\')
        return c;
    c = getc(fin);
    if (islower(c) && index(transtab, c))
        return index(transtab, c)[1];
    return c;
}

```

词法分析器就是一个有限状态自动机的例子，用C或类似lex的程序生成器编写。ad hoc的C版本已改进得相当复杂；除此之外，无论在源代码大小，还是修改性方面，lex可能都更优秀些。

此外，大部分变化是在code.c中，它在hoc.h中增加了一些函数名。机器和以前一样，只是增加了另一个堆栈，用以跟踪嵌入的函数和过程调用。(增加一个堆栈比向已有的堆栈放信息更容易。)下面是code.c的开始部分：

```

$ cat code.c
#define NPROG    2000
Inst    prog[NPROG];    /* the machine */
Inst    *progp;         /* next free spot for code generation */
Inst    *pc;            /* program counter during execution */
Inst    *progbase = prog; /* start of current subprogram */

```

```

int    returning;    /* 1 if return stmt seen */

typedef struct Frame { /* proc/func call stack frame */
    Symbol *sp;    /* symbol table entry */
    Inst *retpc; /* where to resume after return */
    Datum *argn; /* n-th argument on stack */
    int nargs; /* number of arguments */
} Frame;
#define NFRAME 100
Frame frame[NFRAME];
Frame *fp;    /* frame pointer */

initcode() {
    progp = progbase;
    stackp = stack;
    fp = frame;
    returning = 0;
}
...
$

```

因为符号表保留有指向函数和过程的指针，还有待打印串的指针，因此必须在hoc.h中增加联合类型：

```

$ cat hoc.h
typedef struct Symbol { /* symbol table entry */
    char *name;
    short type;
    union {
        double val;    /* VAR */
        double (*ptr)(); /* BLTIN */
        int (*defn)(); /* FUNCTION, PROCEDURE */
        char *str; /* STRING */
    } u;
    struct Symbol *next; /* to link to another */
} Symbol;
...
$

```

编译时，函数通过define进入符号表，符号表存储原始部分，并在编译成功生成代码后，修改下一个空闲位置。

```

define(sp)    /* put func/proc in symbol table */
    Symbol *sp;
{
    sp->u.defn = (Inst)progbase;    /* start of code */
    progbase = progp;    /* next code starts here */
}

```

执行过程中，函数或过程被调用时，所有参数已计算出来并压入堆栈(第一个参数最深)。call的操作码后跟着符号表指针和参数数目。一帧(Frame)就是指栈中包含的对一个程序有用的信息的全部——包括程序的符号表入口、返回地址、表达式栈中的参数地址和被调用的参数数量。帧由call建立，然后执行例程代码。

```

call()    /* call a function */
{
    Symbol *sp = (Symbol *)pc[0]; /* symbol table entry */
    /* for function */
    if (fp++ >= &frame[NFRAME-1])
        execerror(sp->name, "call nested too deeply");
    fp->sp = sp;
    fp->nargs = (int)pc[1];
    fp->retpc = pc + 2;
}

```

```

        fp->argn = stackp - 1; /* last argument */
        execute(sp->u.defn);
        returning = 0;
    }

```

图8-4描述了这个结构。

最后，被调用的例程通过procret或funcrret返回：

```

funcrret()      /* return from a function */
{
    Datum d;
    if (fp->sp->type == PROCEDURE)
        execerror(fp->sp->name, "(proc) returns value");
    d = pop(); /* preserve function return value */
    ret();
    push(d);
}

procret()      /* return from a procedure */
{
    if (fp->sp->type == FUNCTION)
        execerror(fp->sp->name,
                    "(func) returns no value");
    ret();
}

```

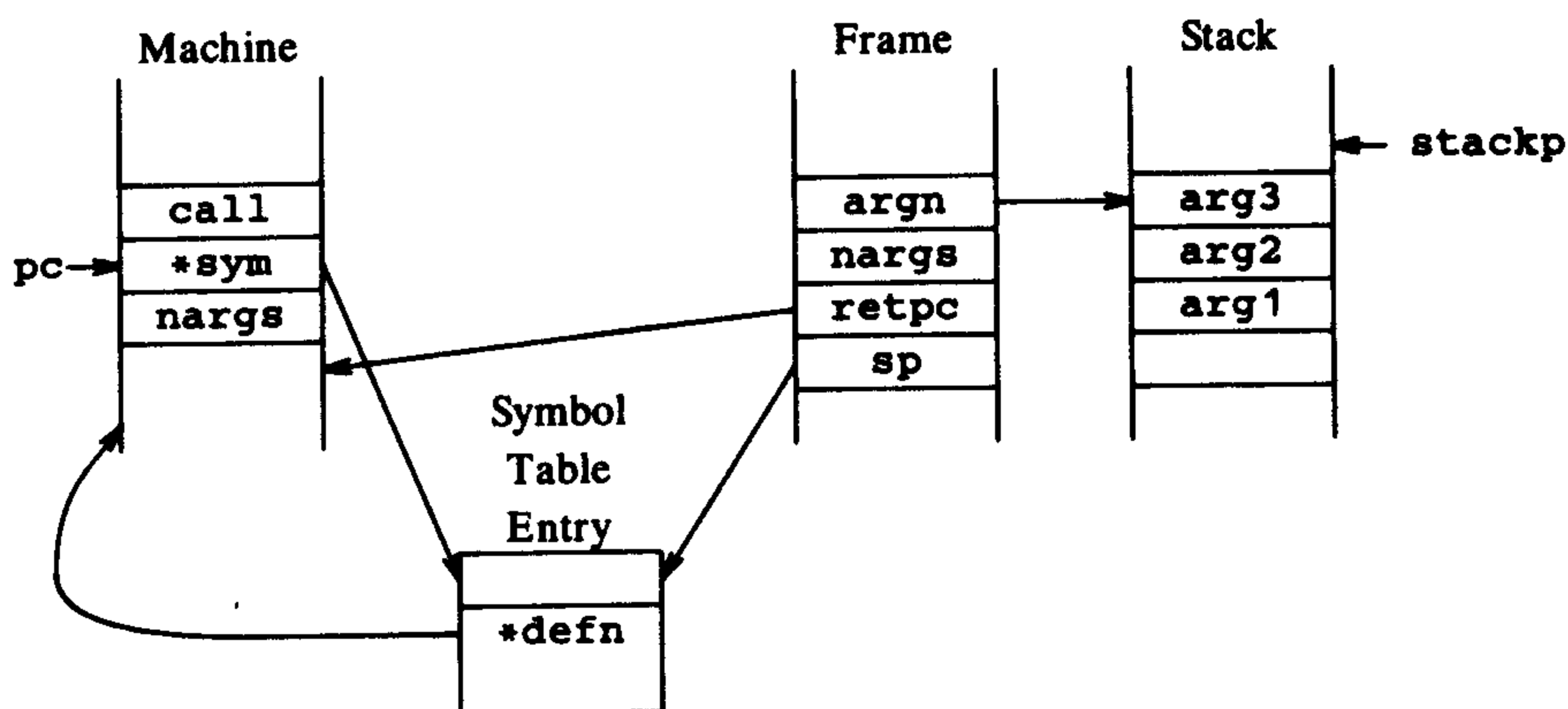


图8-4 过程调用的数据结构

函数ret把参数弹出栈，恢复帧指针fp，并设置程序计数器。

```

ret()          /* common return from func or proc */
{
    int i;
    for (i = 0; i < fp->nargs; i++)
        pop(); /* pop arguments */
    pc = (Inst *)fp->retpc;
    --fp;
    returning = 1;
}

```

在处理嵌入式语句的返回时，一些解释器例程需要做些改动。这里设置标识returning来完成这个功能，尽管有些不妥，但这种做法很常见。当执行语句return时，returning的值被置为真。若returning被设置，则ifcode、whilecode和execute会提前终止，然后call将returning置为0。

```

ifcode()

```

```

{
    Datum d;
    Inst *savepc = pc;      /* then part */

    execute(savepc+3);      /* condition */
    d = pop();
    if (d.val)
        execute(*((Inst **)(savepc)));
    else if (*((Inst **)(savepc+1))) /* else part? */
        execute(*((Inst **)(savepc+1)));
    if (!returning)
        pc = *((Inst **)(savepc+2)); /* next stmt */
}
whilecode()
{
    Datum d;
    Inst *savepc = pc;

    execute(savepc+2);      /* condition */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc))); /* body */
        if (returning)
            break;
        execute(savepc+2);    /* condition */
        d = pop();
    }
    if (!returning)
        pc = *((Inst **)(savepc+1)); /* next stmt */
}
execute(p)
    Inst *p;
{
    for (pc = p; *pc != STOP && !returning; )
        ((*pc++)());
}

```

getarg用于获取参数，以使用参数或对其赋值，并且对栈施加正确的算术操作。

```

double *getarg()      /* return pointer to argument */
{
    int nargs = (int) *pc++;
    if (nargs > fp->nargs)
        execerror(fp->sp->name, "not enough arguments");
    return &fp->argn[nargs - fp->nargs].val;
}

arg() /* push argument onto stack */
{
    Datum d;
    d.val = *getarg();
    push(d);
}

argassign() /* store top of stack in argument */
{
    Datum d;
    d = pop();
    push(d); /* leave value on stack */
    *getarg() = d.val;
}

```

prstr和prexpr分别完成打印字符串和打印数字。

```

prstr() /* print string value */
{

```

```

        printf("%s", (char *) *pc++);
    }

    prexpr()      /* print numeric value */
    {
        Datum d;
        d = pop();
        printf("%.8g ", d.val);
    }

```

函数varread读入变量。读到文件尾时返回值0，否则返回值1并设置指定的变量。

```

varread()      /* read into variable */
{
    Datum d;
    extern FILE *fin;
    Symbol *var = (Symbol *) *pc++;
    Again:
    switch (fscanf(fin, "%lf", &var->u.val)) {
        case EOF:
            if (moreinput())
                goto Again;
            d.val = var->u.val = 0.0;
            break;
        case 0:
            execerror("non-number read into", var->name);
            break;
        default:
            d.val = 1.0;
            break;
    }
    var->type = VAR;
    push(d);
}

```

如果遇到当前输入文件的末尾，varread调用moreinput，moreinput打开下一个参数文件(如果有的话)。moreinput具备很多输入处理，比这里表现的要多，附录C给出了全部细节。

hoc的情况，就讨论到这里，作为比较，这里列出了各版本的非空行数：

hoc1	59	
hoc2	94	
hoc3	248	(lex版本229)
hoc4	396	
hoc5	574	
hoc6	809	

当然，这些数字是由以下程序统计的：

```
$ sed '/^$/d' `pick * .[chy1]` / wc -l
```

这种语言决不会停止发展的，至少在一定意义上还很容易想到一些有用的扩展，在此就不作深究了。下面的练习提出了一些有价值的参考方法。

练习8-18 修改hoc6，允许在子例程中命名形参，就像\$1，等。

练习8-19 如前所述，除了参数外，所有变量都是全局变量。目前，很多机制都提供了用栈维护局部变量的功能。一个方法就是用auto说明，它为列出的变量分配栈空间，不这样命名的是全局变量。符号表也必须扩充，使得搜索符号时先搜索局部变量，再搜索全局变量。这与命名的参数如何相互作用？

练习8-20 如何在hoc中加入数组？如何把它们传递给函数和过程？数组如何返回？

练习8-21 扩大串处理，使变量能表示串而不单是数字。需要什么操作才能实现？难点在于存储管理：当不需要时串被释放，以充分利用空间。作为中间步骤，可加入较好的输出格式处理，例如，接近C的printf语句的格式。

8.7 性能评价

将hoc与UNIX的其他计算程序相比，可以大致明白它的工作情况。表8-1只涉及了一小部分，但它足以表明本章的实现是合理的。所有时间是基于PDP-11/70的用户时间，以秒计算。在两个任务中，第一个是计算阿克曼(Ackermann)函数ack(3.3)，一个很好的函数调用机制的测试函数：它共有2432次调用，嵌套层次很深。

```
func ack() {
    if ($1 == 0) return $2+1
    if ($2 == 0) return ack($1-1, 1)
    return ack($1-1, ack($1, $2-1))
}
ack(3,3)
```

第二个是计算小于1000的菲波那契数，共计算100次。主要是测试算术运算和偶然的函数调用。

```
proc fib() {
    a = 0
    b = 1
    while (b < $1) {
        c = b
        b = a+b
        a = c
    }
}
i = 1
while (i < 100) {
    fib(1000)
    i = i + 1
}
```

参与比较的语言有4种：hoc、bc(1)、bas(BASIC语言的早期版本，仅运行于PDP-11上)和C(所有变量均为双精度型)。

表8-1 用户时间(PDP-11/70)

程 序	ack(3,3)	100×fib(1000)
hoc	5.5	5.0
bas	1.3	0.7
bc	39.7	14.9
C	<0.1	<0.1

表中的数据是由time计算的用户时间与系统CPU时间总和。也可以用C程序来检测每个函数使用了多少时间。每个程序必须打开概要重新编译，这只需在C编译和载入中加入选项-p即可。如果修改makefile用以读入

```
hoc6: $(OBJS)
cc $(CFLAGS) $(OBJS) -lm -o hoc6
```

以便cc命令使用变量CFLAGS，然后执行

```
$ make clean; make CFLAGS=-p
```

则程序结果中包含了概要代码。当程序运行时，生成一个文件mon.out，它由程序prof解释。

为了对这些概念作一简明的论述，这里用上面的Fibonacci程序基于hoc6作了一个测试：

```
$ hoc6 <fibtest
$ prof hoc6 | sed 15q
```

name	%time	cumsecs	#call	ms/call
_pop	15.6	0.85	32182	0.03
_push	14.3	1.63	32182	0.02
mcount	11.3	2.25		
csv	10.1	2.80		
cret	8.8	3.28		
_assign	8.2	3.73	5050	0.09
_eval	8.2	4.18	8218	0.05
_execute	6.0	4.51	3567	0.09
_varpush	5.9	4.83	13268	0.02
_lt	2.7	4.98	1783	0.08
_constpu	2.0	5.09	497	0.22
_add	1.7	5.18	1683	0.05
_getarg	1.5	5.26	1683	0.05
_yyvsparse	0.6	5.30	3	11.11

```
$
```

运行测试程序
分析

与time类似，以概要得到的数据只是大致的时间，而不是绝对精确的。然而，它们确实给出了如何使hoc更快的信息。从上可知大约有一分之一的运行时间都用于压栈弹栈。如果加上C子程序链接函数csv和cret的时间，系统开销会更大。(mcount是用cc -p编译的概要代码中的一段。)用宏代替函数调用，开销有相当显著的区别。

修改code.c，用宏代替栈操作过程push和pop可以看到区别：

```
#define push(d) *stackp++ = (d)
#define popm()  *--stackp      /* function still needed */
```

(函数pop仍需作为一个机器操作码，因此并不能代替所有pop。)修改后的程序速度提高了35%，表8-1中的数据分别从5.5秒和5.0秒缩减到3.7秒和3.1秒。

练习8-22 宏push和popm没有错误检测。请分析为什么不需要。怎样能将函数检测与宏的速度结合起来？

8.8 小结

这一章有许多重要的知识。首先，语言开发工具很有用，它使我们集中精力于一项工作中最有趣的部分——语言设计——因为它很容易实验。语法的使用为实现提供了有组织结构——通过语法，例程连接为整体，在分析过程必要的时候调用。

其次，把工作当作语言开发，而不仅是“写一个程序”，这种思考方式具有一定意义。一个程序将组织为语言处理器要求句法规整(这是用户接口)，并构造其实现，这对于新特征与原有语言平滑啮合也是有帮助的。“语言”并不局限于传统的编程语言——它还包括eqn、pic、yacc、lex和make。

本章讨论了许多工具的使用方法。例如，make的作用可以说是无价的，它基本上消除了由于忘记重新编译例程所引起的一类错误，它确保了不做额外的工作，并提供一种把相关的、甚至相依的操作打包到单一文件中的便利途径。

当有多于一个的文件必须使用数据定义时，头文件是管理它们的一个好办法。把消息集

中到一起，消除了不同版本造成的错误，尤其是和make相关时。把数据和程序放到文件中是很重要的，必要时可以实现信息隐藏。

限于篇幅，还有一些概念就不强调了。其中一个是在hoc系列的发展过程中，对UNIX其他工具的使用程度。程序的每个版本放在分离的目录下，通过唯一标识的文件链接到一起。ls和du被频繁使用，以获知什么地方有什么。许多问题可由程序解决，例如，变量在什么地方定义？用grep；新版本中有什么变化？用diff；怎样将不同版本通用？用idiff；文件有多大？用wc；该做备份吗？用cp；如何只备份上次备份以来改动过的文件？用make。以下风格绝对是UNIX系统上程序开发的典型：大量的小工具；单独或联合使用；完成机械性的工作以节省人力。

有关文献

yacc由Steve Johnson开发。yacc所生成的分析器能识别的语言，称作LALR(1)：从左向右分析，向前最多看一个符号。采用分离描述解决语法先后次序及二义性问题，对于yacc还是件新鲜事。请参见“Deterministic parsing of ambiguous grammars”，A. V. Aho、S. C. Johnson和J.D.Ullman著，CACM，1975年8月。文中有许多独创性的算法和数据结构用以建立、存储分析表。

由A. V. Aho 和 J. D. Ullman所著“Principles of Compile Design” (Addison-Wesley, 1977年)描述了yacc和其他分析器产生器的基本理论。在《UNIX程序员手册》卷2B中对yacc也有描述，并且还描述了一个类似 hoc2的计算器，将它们作一个比较相当有指导价值。

lex最初由Mike Lesk编写，此后又被Aho和Ullman所描述，并且在《UNIX程序员手册》中也有关于lex语言的文档。

yacc及较低级别的lex，已被用于开发多种语言处理器，包括可移植C编译器、Pascal、FORTRAN 77、Ratfor、awk、bc、eqn以及pic。

make由Stu Feldman 编写。请参见“MAKE—a program for maintaining computer programs” (Software—Practice & Experience, 1979年4月)。

Jon Bentley 所写的Writing Efficient Programs(Prentice-Hall, 1982年)描述了提高程序速度的技术，重点在于首先找到正确的算法，然后在必要时精炼代码。

第9章 文档处理

UNIX系统最初的一个应用程序是编辑和格式化文档，正是由于签定了研制文档处理系统(而不是操作系统)的合同才促使贝尔实验室管理部门购买了第一个PDP-11硬件。(值得庆幸的是，他们的收效比预计的大。)

第一个格式化程序称作roff。roff很小，易于使用，能快速生成行式打印机所需的简单文档。另一格式化程序称作nroff，它是由颇为能干的Joe Ossanna研制的。nroff是可编程的，而不是向用户提供文档的每一种式样，从而可使用nroff语句编程处理更多的格式化任务。

自1973年有了小型排字机以来，nroff便被扩展为新的程序troff(nroff念成en-roff，类似地，troff念成tee-roff)，troff用于处理由小型排字机提供的多尺寸、多字体的较丰富的字符集。nroff和troff这两个程序大致相同，接受同一种输入语言，但是，nroff不接受改变尺寸这类命令。我们着重讨论troff，但大部分讨论同样也适用于nroff，只是在输出设备方面有一定的限制。

troff的强大功能在于基本语言的灵活性和可编程能力——它几乎可以完成所有的格式化任务。但这种灵活性要付出很高的代价——troff通常极难使用。凭心而论，几乎所有的UNIX文档编排软件多多少少都受到troff的影响。

来看一个例子，页面格式——文档的一般式样——即文档标题、页眉、段落的编排形式；页码的位置；页面尺寸大小等等。这些设置不是预先准备好并保存在机器内部的，而是必需通过编程实现。然而对于每个用户来说，不必专门指定每份文档格式编排的所有细节，因为它们已由标准的格式化命令包提供。当使用这一程序包时，不必说明“下一行排在中间位置，用较大号的字母，并用粗体字”等，而只需指出“下一行是标题”，自动套用程序包定义的标题格式。换言之，用户定义文档的逻辑部件——文档标题、子标题、段落、附注等等，而不使用具体的文档尺寸、字体和位置。

不幸的是，最初“标准”的程序包现在已不标准，几个广泛使用的程序包增加了很多局部变量。这里，只讨论两个通用程序包：ms——最初的标准程序包；mm——System V的新版本。还将介绍用于打印帮助手册的man程序包。

本章主要讨论ms，因为它是UNIX系统第7版的标准程序包，是所有这类程序包的一个典型例子。ms具有很强的功能，不过，我们还需要对它作一点扩充，例如，增加在文本中处理in this font这样的词组的命令。

宏程序包适用于很多格式化任务，但有时需要还原成基础的troff命令。本章只描述troff命令的少许内容。

虽然troff提供了控制输出格式的全部能力，但它很难适用于数学、表格、图形等复杂的文档应用。这些文档和页面格式一样难于实现，然而，可以用不同的形式来解决这些问题。某些适合于数学、表格和图形的专用语言，可以代替格式命令程序包，因为它们很容易描述这类文档。数学、表格和图形诸方面的文档可用一个独立程序处理，该程序将专用语言翻译成troff命令。独立程序之间通过管道传送信息。

这些预处理是UNIX解决问题的范例——将独立程序与troff命令有机地结合起来，而不是把troff命令扩充得更大、更复杂。（当然，第8章提供的语言开发工具已经提供了具体帮助。）我们将介绍两个程序，它们是格式化表格程序tbl和格式化数学表达式程序tqn。

本章还将给出关于文档编排的提示信息及其他支援编排的工具。采用的具体例子是第8章中描述hoc语言的文档和hoc手册。hoc语言的详细文档见附录B。

9.1 宏程序包ms

宏程序包的核心思想是按照文档的逻辑成分，即题目、节、标题和段落，来说明文档，而不是通过基线间距离、字体式样和字母尺寸来对文档进行描述。这样就把你从十分费力的工作中解脱出来，使文档与毫不相干的细节脱离。事实上，使用一组具有相同逻辑名的不同的宏定义，可使文档显示成不同格式。例如，用同一格式命令，但是使用四个不同的宏程序包进行格式化，可以使一份文档以四种形式出现：技术报告、会议论文、杂志文章和书籍。

无论troff是否包括宏程序包，troff的输入皆是由普通文本和格式命令组成，格式命令散布在文本中。格式命令有两种形式：第一种是以.开头，后接一至两个字母或数字，或者是参数，如下：

```
.PP
.ft B
This is a little bold font paragraph.
```

troff的内部命令均用小写字母表示，因此按照惯例，宏程序包中的命令以大写字母表示。在这个例子中，.PP是关于段的ms命令，.ft B是troff命令，它把字体改变为粗字体(bold font)。（字体用大写名表示，可能因不同排字机而异。）

troff命令的第二种形式为以反斜杠\开始的字符序列，可以出现在任何输入位置，例如，\fB，也是把字体转换为粗体。这种形式是纯粹的troff命令形式。我们还会回过头来作简短的叙述。

可以在每个段落之间只用一个.PP命令进行格式化。对大多数文档来说，使用大约12种ms命令就可以解决编程。例如，附录B中给出的hoc文档有题目、作者名、摘要、节和段标题的自动顺序编号。它只用了14个不同的命令，其中的12个命令是成对出现的。ms命令的一般形式如下：

```
.TL
文档题目(一行或多行)
.AU
作者名, 一行给出一个作者名
.AB
摘要, 用命令.AE结束
.AE
.NH
节标题编号(自动顺序编号)
.PP
段落 ...
.PP
另一段落 ...
.SH
子标题(不编号)
.PP
...
```

格式命令必须出现在行首。命令之间的输入形式是自由的，输入的换行位置无关紧要，因为troff可以把字从一行移动到另一行，使行填满，这一处理就叫填满(filling)。并可把多余的空格均匀分派到行中各单词间，以使页边整齐，这一处理叫调整(justification)。实际使用时，每个句子最好从换行后开始，这样可以使随后的编辑工作容易些。

下面是实际的hoc文档的开始部分：

```
.TL
Hoc - An Interactive Language For Floating Point Arithmetic
.AU
Brian Kernighan
Rob Pike
.AB
.I Hoc
is a simple programmable interpreter
for floating point expressions.
It has C-style control flow,
function definition and the usual
numerical built-in functions
such as cosine and logarithm.
.AE
.NH
Expressions
.PP
.I Hoc
is an expression language,
much like C:
although there are several control-flow statements,
most statements such as assignments
are expressions whose value is disregarded.
...
```

这里的 .I 命令表示用斜体字印刷它的参数，没有参数时表示将下文的字体转换为斜体。

如果使用宏程序包，可将它作为troff命令的参数：

```
$ troff -ms hoc.ms
```

-m后的字符指明所用的宏程序包^①。当用ms 格式化时，hoc文档的排版形式如下：

Hoc - An Interactive Language For Floating Point Arithmetic

Brian Kernighan

Rob Pike

ABSTRACT

Hoc is a simple programmable interpreter for floating point expressions. It has C-style control flow, function definition and the usual numerical built-in functions such as cosine and logarithm.

1. Expressions

Hoc is an expression language, much like C: although there are several control-flow statements, most statements such as assignments are expressions whose value is disregarded.

① ms 宏程序包在文件/usr/lid/tmac/tmac.s中，man宏程序包位于文件/usr/bin/tmac/an中。

9.1.1 阵列文本

虽然用troff填满和调整文本很方便,但有时不是很合适——例如,程序文档的两头就不必对齐。这种非格式化文本称为阵列文本。ms命令.DS(display start)和.DE(display end)将按照输入格式的样子输出文本,这时文本只作缩进,而不作重新编排。下面是hoc手册的下一部分排版形式,其中包括一些阵列文本命令:

```
.PP
.I Hoc
is an expression language,
much like C:
although there are several control-flow statements,
most statements such as assignments
are expressions whose value is disregarded.
For example, the assignment operator
= assigns the value of its right operand
to its left operand, and yields the value,
so multiple assignments work.
The expression grammar is:
.DS
.I
expr:          number
              |
              | variable
              | ( expr )
              | expr binop expr
              | unop expr
              | function ( arguments )
.R
.DE
Numbers are floating point.
```

排印成

Hoc is an expression language, much like C: although there are several control-flow statements, most statements such as assignments are expressions whose value is disregarded. For example, the assignment operator = assigns the value of its right operand to its left operand, and yields the value, so multiple assignments work. The expression grammar is:

```
expr:          number
              |
              | variable
              | ( expr )
              | expr binop expr
              | unop expr
              | function ( arguments )
```

Numbers are floating point.

阵列文本一般不作填满或调整。进一步说,如果在当前页中没有足够的空间,那么阵列的材料(命令后的信息)便移到下一页。.DS可以有几个选项,例如,选项L表示左对齐,选项C表示每行分别居中排列;选项B表示整段阵列内容居中。

上述阵列内容中的各项均用Tab分隔。缺省时,troff Tab设置在每隔半英寸的位置上,而不是按8个空格的通常方式。即使Tab每隔8个空格停一下,由于字符宽度不同,troff处理的Tab也并不总是出现在预想的位置上。

9.1.2 改变字体

ms宏程序包提供三种改变字体的命令：.R产生罗马字体——最经常使用的字体；.I产生斜体字；.B产生粗体字。每个命令只是为其后的文本选择字体。

例如，

```
This text is roman, but
.I
this text is italic,
.R
this is roman again, and
.B
this is boldface.
```

排印成：

This text is roman, but *this text is italic*, this is roman again, and **this is boldface**.

.I和.B的参数是可选的，所做的字体改变仅在参数上起作用。在troff中，参数中包含空格的部分必须用引号引住，唯一可用的引用字符是双引号"。

```
This is roman, but
.I this
is italic, and
.B "these words"
are bold.
```

排印成：

This is roman, but *this* is italic, and **these words** are bold.

命令.I和.B的第二个参数均打印成罗马体，并原封不动地附在第一个参数之后。这一特性最适合于在适当的字体中加标点符号。下述内容：

```
(parenthetical
.I "italic words")
```

中的最后一个括号位置不当，所以排印成：

(parenthetical *italic words*)

其中最后的圆括号排成斜体，而使用

```
(parenthetical
.I "italic words" )
```

就可排印成正确格式

(parenthetical *italic words*)

由此可知，在troff命令参数中必须正确使用双引号。

nroff命令可识别不同的字体，只是效果不太理想。nroff的斜体字要下划线，而且nroff没有粗体字符，尽管某些nroff版本能够通过重复打印的方法来模拟粗体字。

9.1.3 其他命令

脚注由命令.FS引入，并以命令.FE结束。如果想用任何标志符号，如用星号(*)或匕首式符号(†)作为脚注标记，那么脚注可以这样生成：

```
identifying mark like an asterisk or a dagger.\(dg
.FS
```

```
\(dg Like this one.
```

```
.FE
```

```
This footnote was created with ...
```

缩进段落可用.IP命令产生,可能要带一个数字或其他页边空白标记。为了产生如下文本:

(1) First little paragraph.

(2) Second paragraph, which we make longer to show that it will be indented on the second line as well as the first.

需要输入:

```
.IP (1)
```

```
First little paragraph.
```

```
.IP (2)
```

```
Second paragraph, ...
```

命令.PP或.LP(左对齐段落)用于终止.IP命令。IP的参数可以是任意字符串,必要的话,可用引号保护空格。第2个参数用于指定缩进量。

命令对.KS和.KE使文本保持完整,如果包括在两个命令之间的文本不能全部排印在当前页,则将它们排印在下一页。假如用命令.KF代替命令.KS,若要保证一页的完整形式的话,则会把刚刚读过的文本的后续部分移到下一页的顶端,本书中所有的表格都是用.KF命令进行编排的。

通过设置数字寄存器可以改变ms命令的大部分缺省值,数字寄存器是ms所用的troff变量。最常用的是控制文本尺寸和行距的寄存器。一般文本尺寸是10磅,一磅约为1/72英寸,这是从印刷工业沿袭下的标准单位。行间距离通常为12磅。这些尺寸可以改变,例如,将文本尺寸和行距改为9磅和11磅(正如我们给出的阵列文本那样),通过设置数字寄存器PS和VS实现:

```
.nr PS 9
```

```
.nr VS 11
```

其他数字寄存器还有控制每行长度的LL、段落缩进的PI和段间距的PD,它们一般在命令.PP或.LP之后才起作用。

表9-1 常用ms格式命令(参见ms(7))

命 令	含 义
.AB	开始摘要; 其终止命令为.AE
.AU	作者名写在下一行; 允许多个作者名
.B	开始粗体字, 即如果给出参数则将其印刷成粗体字
.DS <i>t</i>	开始阵列(不作填满)文本; 由命令.DE结束; <i>t</i> =L(左对齐)、C(居中)、B(文本段居中)
.EQ <i>s</i>	开始数学公式(<i>s</i> =eqn 输入); 由.EN命令结束
.FS	开始附注; 由.FE命令结束
.I	开始斜体字, 即如果给出参数则将其印刷成斜体字
.IP <i>s</i>	缩进段落; 页左的空白处用 <i>s</i> 作标注
.KF	将文本排印在一起; 需要时换页; 用.KE命令结束
.KS	将文本排印在同一页; 用.KE命令结束
.LP	左对齐新段落
.NH <i>n</i>	第 <i>n</i> 层的标题编号; 命令后是标题; 直到出现.PP命令或.LP命令
.PP	新段落

(续)

命 令	含 义
.R	返回罗马字体
.SH	子标题; 命令后是标题; 直到出现 .PP命令
.TL	命令后是文本题目; 直到出现下一个 ms命令
.TS	开始表格(tbl输入), 由.TE命令结束

9.1.4 宏程序包mm

我们在这里不详细讨论mm宏程序包, 因为它的基本思想与ms宏程序包非常类似, 具体内容也大致相同。与ms相比, mm提供的控制参数更多, 排版功能更强(例如对表项进行自动顺序和编号), 以及更好的错误信息处理。表9-2给出的mm命令与表9-1中的ms命令等价。

表9-2 常用mm格式命令

命 令	含 义
.AS	开始摘要, 由.AE命令结束
.AU	命令后的第一个参数是作者名
.B	开始粗体字, 即如果给出参数则将其印刷为粗体字
.DF	将文本排印在一起; 需要时换页; 用.DE 命令结束
.DS	开始阵列文本; 由.DE 命令结束
.EQ	开始印刷数学公式(eqn 输入); 由 .EN 命令结束
.FS	开始脚附注; 由.FE 命令结束
.I	开始斜体字, 即如果提供参数则将其印刷为斜体
.H <i>n</i> "..."	第 <i>n</i> 层编号标题" ... "
.HU "..."	未编号标题" ... "
.P	开始段落; 用 nr Pt 1缩进段落
.R	返回罗马字体
.TL	命令后面是标题, 直到下一个mm命令
.TS	开始表格(tbl 输入); 由.TE命令结束

练习9-1 遗漏结束命令(如.AE或.DE)通常会造成严重后果。写一个检查程序mscheck(或你喜欢的程序包)检查ms的输入错误。建议使用awk。

9.2 troff

实际编排文档时, 人们有时不得不超越ms、mm或其他程序包而直接使用troff命令的某些功能。然而, 这样做有点类似于用汇编语言编程, 所以必须谨慎地编写, 因为类似汇编的语言是不容易驾轻就熟的。

常见的有以下三种情况: 访问特殊字符、改变行内尺寸和字体, 此时将调用某些基本的格式化函数。

9.2.1 字符

访问特殊字符(如希腊字母 π 、图形字符如 $\bullet$ 和 $\dagger$ 以及各种行和空格的间距)尽管不是很有规律, 但还是比较容易实现的。每个特殊字符都可表示成\c, c是单个字符, 也可以表示为\cd, 这里cd是字符对。

troff命令将ASCII码表中的负号-打印成连字号-。真正的减号必须键入\-, 而要想键入一

个破折号，则用\em，它代表“em破折号”，即字符——。

表9-3列出一些最常用的专用字符，troff手册中有很多这种特殊字符(这个列表与你使用的系统可能不同)。

troff命令常常被禁止解释字符，特别是字符反斜杠和前导符句点。两个常用的“不许解释”字符是\e和\&。序列\e保证印刷为反斜杠，而且不作解释，它用于在输出时得到反斜杠。而\&则什么操作也不做，它是一个零宽度空格。\&的主要作用是阻止troff命令解释位于命令行首的句点。本章中多次用到\e和\&。例如，印刷本章开始的ms宏程序包所键入的命令为：

```
\&.TL
.I "Title of document"
\&.AU
.I "Author name"
\&.AB
\&...
...
```

当然，印刷上面部分的几行，需要键入的命令为：

```
\e&.TL
\&.I "Title of document"
\e&.AU
...
```

你大概能够想象出需要键入什么命令才能印出上面的三个命令。

另一个很少碰见的专用字符是非填充空格(unpaddable blank)，反斜杠\后随一个空格。一般情况下，troff命令把通常的空白与行页的空白一起调整，而非填充空白不进行调整，它类似于其他字符，有固定宽度。它还可用于把多音节单词作为一个单独参数进行传送

```
.I Title\of\document
```

表9-3 troff专用字符表

字 符	含 义
-	连字号-
\(hy	连字号-, 同上
\-	当前字体中的负号-
\(mi	数学字体中的负号-
\(em	em 破折号——
\&	什么都不是；只用于保护句点
\blank	未填充空格
\	未填充的半空格
\e	文字转义符，通常为\
\(bu	粗点号•
\(dg	匕首符†
\(*a	α ; \(*b= β 、\(*c= ξ 、\(*p= π 、等等
\fX	改变字体为X；X=P表示使用前面的字体
\f(XX	改变字体为XX
\sn	将字体尺寸改为n磅，n=0表示使用前面的字体尺寸
\s ± n	字体尺寸相对改变n磅

9.2.2 改变字体和尺寸

多数字体和格式的变化可用位于行首的宏命令实现，例如.I，不过有时这种改变要在行内进行。特别地，换行符是一个字分隔符，所以如果要在词中间作字体变化时，就不能使用宏

程序包。本小节介绍troff命令是如何解决这一问题的。注意，这是由troff命令提供的解决方法，而不是ms宏程序包提供的。

troff用反斜杠引入行内命令，两个最常用的命令是\f和\s，前者用于改变字体，后者用于改变字体尺寸。

在命令\f中，紧跟在f后的字符指定字体，例如下述命令：

```
a \fBfriv\fIolous\fR \fIvar\fBiety\fR of \fIfonts\fP
```

输出字体为：

```
a friv olous var iety of fonts
```

\fP返回到前面使用的字体——不论最后一次字体转换之前的字体是什么。(troff 只有一种先前字体，而不是以堆栈形式存放字体。)

有些字体的名称由两个字符组成，由格式\f(XX指定，XX是字体名。例如，排字机的字体CW(Courier Coustant Width)的关键字keyword可以写成：

```
\f(CWkeyword\fP
```

显然，这样键入命令很不方便，为此，我们对ms扩充了.CW宏命令，这样就不必键入和读出反斜杠。可以用它排印一行中的词，如troff一词，可排印如下：

```
The
.CW troff
formatter ...
```

宏定义的格式也很容易改变。

尺寸改换由字符序列\s n引入，n是一位或两位数字，用以指定新的尺寸的磅数，例如，\s8为8磅。经常要对尺寸作相对的改变，可通过在尺寸前放置一个加号或减号来实现尺寸的相对改变。例如要印刷出SMALL CAPS，可以键入命令：

```
\S-2SMALL CAPS\s0
```

\s0使字体尺寸恢复为先前值。\s0类似于\fP，按照troff命令的习惯用法，不使用\sP。我们对ms的扩充还有宏命令.UC(大写)。

9.2.3 基本troff命令

即使对性能良好的宏程序包，也必须了解用于控制间距、填满和设置表格位等的几个重要troff命令。命令.br用于分段，即下一行输入的内容将换行输出。例如，可以使用.br将一个长题目分开放在适当的位置：

```
.TL
Hoc - An Interactive Language
.br
For Floating Point Arithmetic
...
```

命令.nf表示不对输出行作填满操作，每一行输入直接作为一行输出；命令.fi表示对输出行作填满操作；命令.ce将下面一行输入放置在中央。

命令.bp进行换页；命令.sp输出一行空格，其之后可以有一个参数，指出空多少行或空多少空间。

```
.sp 3          留三个空白行
.sp .5        留半个空白行
```

<code>.sp 1.5i</code>	留1.5英寸的空白
<code>.sp 3p</code>	留3磅宽的空白
<code>.sp0 3.1c</code>	留3.1厘米的空白

该页底部多余的空白被去除，这样，`.sp`参数大多数情况下就相当于`.bp`的作用。

命令`.ta`设置表格位(初始值为每隔半英寸停一下)。

`.ta n n n ...`

Tab停止位置按照页左边空白的指定距离计算。当用`sp`命令时，如果参数后随一个`i`时，表示数字`n`以英寸为计数单位。若设置Tab命令后缀`R`，则把文本调整到下一个表格停止位；后缀`C`则使位置居中。

命令`.ps n`设字体尺寸为`n`磅；命令`.ft X`设字体为`X`。递增尺寸和返回先前的规则与`\s`和`\f`的规则相同。

9.2.4 定义宏

就其普遍规则来说，定义宏涉及比`troff`更加复杂的内容，不过我们还是可以介绍某些基本的宏定义。例如，下面是宏`.CW`的定义：

<code>.de CW</code>	开始定义
<code>\&\f(CW\\\$1\fP\\\$2</code>	关于第1个参数的字体改变
<code>..</code>	定义结束

当调用宏时，`\$n`给第`n`个参量赋值；如果不提供第`n`个参量，则为空。双反斜杠延迟宏定义过程中的`\$n`计算。当遇到参数是以点为前导符的情况，如：

`.CW .sp`

则`\&`防止把参数解释为`troff`命令。

9.3 tbl和eqn预处理程序

`troff`是一个庞大的且复杂的程序，对`troff`进行修改决不是一件轻而易举的事情。因此，为了开发程序以实现数学公式和表格的编排，采取了一种实用的方法，即用相互独立的程序`eqn`和`tbl`分别实现数学和表格语言的设计。程序`eqn`和`tbl`的作用如同`troff`命令的“预处理程序”一样。实际上，`troff`对于排字机来说就是汇编语言，`eqn`和`tbl`是编译生成它的程序。

`eqn`比`tbl`开发得早，它是第一个用`yacc`开发出来的非程序设计语言。随后出现了`tbl`，它与`eqn`实质相同，尽管在语法上它们毫不相干。`tbl`不用`yacc`语言，因为其语法非常简单，没有使用`yacc`的必要。

由于UNIX具有很强的管道功能，因而把程序分成各个独立的部分是值得提倡的方法，除了将工作分解成小部分以外(无论怎样都是必需的——`troff`程序本身与在PDP-11上所能运行的最大程序相比已相差无几)，管道还可以减少各部分之间和相互关联的程序之间的通信。减少程序之间的通信是很重要的，这样就不必存取源代码生成预处理程序。进一步说，用管道不必为巨大的中间文件担忧，除非为了调试而有意使中间文件各部分分开运行。

独立程序通过管道通信时还存在一些问题。通信速度不理想，因为输入和输出量很大。`eqn`和`tbl`从输入到输出一般要进行8:1的扩展。更重要的是，信息只能单向传输。例如，没有

办法使eqn可以确定当前字体尺寸，而这一点使eqn语言的使用受到一些限制。最后一个问题是出错报告比较困难。当执行eqn或tbl程序出现错误时，从troff命令执行状态返回到诊断显示状态，有时是难以实现的。

然而，独立程序的优越性远远超过了它的缺陷，所以以下基于同一模型编写了另外几个预处理程序。

9.3.1 表格

我们先简短讨论tbl程序，因为首先要排印的是hoc文档的操作符表。tbl从输入文件或标准输入读进命令，并把命令.TS和命令.TE之间的文本转换成排印表格的troff命令，按列对齐并处理所有内容。.TS和.TE所在行也要复制，以便宏程序包能适当安排表格，例如，使表格编排在同一页上，并从有关文本开始印刷。

在安排复杂表格时需要查阅tbl手册，但是借助一个实例就可以介绍大多数常用命令的使用方法。下面是编排hoc文档中操作符表的程序：

```
.TS
center, box;
c s
lfcW 1.
\fbTable 1:\fp Operators, in decreasing order of precedence
.sp .5
^      exponentiation (\s-1FORTRAN\s0 **), right associative
! \-   (unary) logical and arithmetic negation
* /    multiplication, division
+ \-   addition, subtraction
> >=   relational operators: greater, greater or equal,
< <=   less, less or equal,
\&== != equal, not equal (all same precedence)
&&     logical AND (both operands always evaluated)
||      logical OR (both operands always evaluated)
\&=    assignment, right associative
.TE
```

它排印出下面这张表：

Table 1: Operators, in decreasing order of precedence

^	exponentiation (FORTRAN **), right associative
! -	(unary) logical and arithmetic negation
* /	multiplication, division
+ -	addition, subtraction
> >=	relational operators: greater, greater or equal,
< <=	less, less or equal,
== !=	equal, not equal (all same precedence)
&&	logical AND (both operands always evaluated)
	logical OR (both operands always evaluated)
=	assignment, right associative

分号前的词(center, box)描述整个表的特性：把表置于页面水平方向的中心位置，并画一个表的外框。分号前的词还可以写成doublebox, allbox(将表格中每一条目都框起来)以及expand(把表的宽度扩展成页面宽度)。

接着几行，一直到句点为止，说明表的各个部分的格式，包括标题行和表体。通常第

一个说明针对表的第一行，第二个说明针对表的第二行，最后的说明适用于表的所有其余行。在表1中只有两个说明行，所以第2个说明适用于表中第一行以后所有的行。格式符c指示将排印项目置于列的中心位置；r和l表示右对齐和左对齐；n表示此数值排印项中的数按十进制小数点位置对齐。s指示一个扩展列。格式 c s 表示扩展第一列和第二列，从而将标题置于表体正中。字体可以按列定义，tbl的说明lfCW表示列左对齐，并按CW字体打印。

表文本接在格式指示信息后面。Tab把各列分隔开，有些troff命令(如cp)即使位于表内也可以识别。(注意\&的出现表明：列中出现的无保护前导符的-和=将告诉tbl在表的那一点上画线。)

tbl编排表格要比这个简单例子采用的格式更丰富，它可将文本填入框内，垂直调整列标题，等等。如果用tbl印制复杂一些的表格，最简单的方法是查找《UNIX程序页手册》卷2A中的类似例子，并改写有关命令以达到实际要求。

9.3.2 数学表达式

第2个troff预处理程序是eqn，它把描述数学表达式的语言转换为troff命令用于排版。eqn自动处理字体和尺寸改变，并且能够为标准数学字符串命名。与tbl的输入出现在.TS和.TE命令行之间类似，eqn的输入通常出现在.EQ和.EN命令行之间。例如，

```
.EQ
x sub i
.EN
```

可产生 x_i 。如果使用宏程序包ms，公式可按原样打印，并且可以使用.EQ的可选参数指定方程编号，例如，柯西(Cauchy)积分公式

$$f(\zeta) = \frac{1}{2\pi i} \int_C \frac{f(z)}{z - \zeta} dz$$

可用eqn语言表示成

```
.EQ (9.1)
f( zeta ) ==~ 1 over {2 pi i} int from C
      f(z) over {z - zeta} dz
.EN
```

eqn语言以数学的口头表达方法为基础。口头表达数学公式和eqn输入之间的一个差别在于eqn的括号用大括号{}表示，它们使语言的缺省优先规则无效，而通常的括号是没有什么专门意义的。然而，空格是有重要意义的。注意上例中第一次出现的zeta两边各有一个空格。像zeta和over这种关键字只有用空格和大括号括住时才能被识别，空格和大括号在输出中不出现。要使输出中出现空格，需用~符，正如例中~ = ~，要得到大括号，需用{和}。

eqn的关键字分为几类。希腊字母既可用大写字母拼写，也可以用小写字母拼写，如lambda和LAMBDA(λ 和 Λ)。其它的数学字符用名称表示，例如sum、int、infinity、grad: Σ 、 $\int$ 、 ∞ 、 ∇ 。还有一些位置操作符，例如sub、sup、from、to和over。所以数学公式：

$$\sum_{i=0}^{\infty} x_i^2 \rightarrow \frac{1}{2\pi}$$

可用eqn的关键字描述为：

```
sum from i=0 to infinity x sub i sup 2 --> 1 over {2 pi}
```

还有一些操作数，如sqrt、可扩展括弧、大括号等等。eqn还可以生成列向量和矩阵。当缺省值不适用时，为了处理尺寸、字体和位置还专门设置了一些命令。

诸如 $\log_{10}(x)$ 这样简单的数学表达式通常放在文本体里，而不是直接放在阵列中。eqn关键字delim定义一对字符用于括起行内表达式。通常左定界符和右定界符的字符为同一字符，很多情况下使用美元符\$。由于hoc用\$作为参量，所以我们在下例中使用@作定界符。%也是一个合适的定界符，但要避免与其他定界符相混淆。有很多字符在不同的程序里具有其特殊用途，以致于它们的用途比较混乱。(我们在写本章时的确已感到这一点。)

因此，在给出如下说明后：

```
.EQ
delim @@
.EN
```

行内表达式 $\sum_{i=0}^{\infty} x_i$ 便可如下编排：

```
in-line expressions
such as @sum from i=0 to infinity x sub i@ can be printed:
```

行内表达式可用于表格内数学公式的排印。以hoc文档为例其编排如下：

```
.TS
center, box;
c s s
lfcW n 1.
\fbTable 3:\fp Built-in Constants
.sp .5
DEG      57.29577951308232087680    @180/ pi@, degrees per radian
E        2.71828182845904523536    @e@, base of natural logarithms
GAMMA    0.57721566490153286060    @gamma@, Euler-Mascheroni constant
PHI      1.61803398874989484820    @(\sqrt{5} + 1)/2@, the golden ratio
PI       3.14159265358979323846    @pi@, circular transcendental number
.TE
```

这个表还显示了tbl如何将数字列按照小数点对齐。上面程序的表输出如下：

Table 3: Built-in Constants		
DEG	57.29577951308232087680	180/ π , degrees per radian
E	2.71828182845904523536	e , base of natural logarithms
GAMMA	0.57721566490153286060	γ , Euler-Mascheroni constant
PHI	1.61803398874989484820	$(\sqrt{5}+1)/2$, the golden ratio
PI	3.14159265358979323846	π , circular transcendental number

最后要说明的是，由于eqn将不能识别的字符串用斜体字印刷，所以习惯上用eqn斜体印刷普通字，例如，@Word@印刷成Word。但是要小心：eqn识别某些常用字(如from和to)，对它们要作特殊处理。此外，eqn将会丢掉空格，对此必须谨慎使用。

9.3.3 输出

一旦准备好文档，就必须安排好所有的预处理程序，使troff进行格式编排。命令的顺序必须是tbl，eqn，然后为troff。如果只使用troff，可键入命令

```
$ troff -ms 文件名 (或者-mm)
```

否则，必须指定管道中第一个命令的参数文件名，使别的命令可以读取标准输入，正如

```
$ eqn 文件名 / troff -ms
```

或

```
$ tbl 文件名 / eqn / troff -ms
```

记录打印特定文档真正需要的预处理程序是很麻烦的。对此，程序 doctype 将很有帮助，它可以推导出合适的命令序列：

```
$ doctype ch9.*
cat ch9.1 ch9.2 ch9.3 ch9.4 | pic | tbl | eqn | troff -ms
$ doctype hoc.ms
cat hoc.ms | tbl | eqn | troff -ms
$
```

doctype 用第4章讨论的工具实现。特别地，awk 程序查询预处理程序使用的命令序列，并打印命令行，调用格式化文档所需的命令。doctype 还查找格式化请求的宏程序包 ms 中使用的 .PP(paragraph) 命令。

```
$ cat doctype
# doctype: synthesize proper command line for troff
echo -n "cat $* | "
egrep -h '^\. (EQ|TS|\\[|PS|IS|PP)' $* |
sort -u |
awk '
/^\.PP/ { ms++ }
/^\.EQ/ { eqn++ }
/^\.TS/ { tbl++ }
/^\.PS/ { pic++ }
/^\.IS/ { ideal++ }
/^\.\\[/ { refer++ }
END {
    if (refer > 0) printf "refer | "
    if (pic > 0)   printf "pic | "
    if (ideal > 0) printf "ideal | "
    if (tbl > 0)   printf "tbl | "
    if (eqn > 0)   printf "eqn | "
    printf "troff "
    if (ms > 0) printf "-ms"
    printf "\n"
}
$
```

(egrep 选项 -h 的作用是删除每一行的文件名标题。不幸的是，并非系统的所有版本都支持这一选项。) doctype 扫描输入，搜集命令行中所有组件类别信息。检查过所有信息之后，按适当的次序进行打印输出。前面给出了利用 troff 排印文档的步骤，虽然其细节是针对于 troff 的，但其思想具有普遍性，即让机器处理繁琐的细节。

和 bundle 程序一样，doctype 生成命令后，还要求用户在 shell 中重新键入命令，这个问题的解决留作练习。

实际运行 troff 时，必须牢记 troff 的工作特点是与系统有关：在某些安装设备上，troff 直接驱动排字机，而对另外的系统，它产生标准输出信息，这些信息必须通过另外的程序送到排字机。

顺便提一下，doctype 程序的第1版没有使用 egrep 和 sort，而用 awk 扫描所有的输入。由于对于大型文档 doctype 运行得很慢，所以我们增加了 egrep，以便快速检索，并增加了 sort -u 用于排除重复信息。对于典型文档，产生两个建立数据窗口的附加进程所需的额外开销要少于运行 awk 程序在大量输入上的开销。为了说明这一点，给出 doctype 和只运行 awk 的 doctype 两个版本的比较，用它们编排一段约为 52000 个字符的内容：

```

$ time awk '... doctype 不包含 egrep ...' ch9.*
cat ch9.1 ch9.2 ch9.3 ch9.4 | pic | tbl | eqn | troff -ms

real      31.0
user       8.9
sys        2.8
$ time doctype ch9.*
cat ch9.1 ch9.2 ch9.3 ch9.4 | pic | tbl | eqn | troff -ms

real       7.0
user       1.0
sys        2.3
$

```

比较结果明显展示了用三个处理程序的doctype版本的优越性。(这个比较是在只有一个用户的机器上执行的。对于重负载系统,从实际使用时间的比值来看,使用有egrep的版本更好一些。)注意,这里我们的使用原则是:先开发简单而能正确工作的版本,然而再着手做优化改进工作。

练习9-2 如果eqn定界符是美元符号,怎样在输出中得到它呢?提示:考查eqn的引号和预定义字。

练习9-3 为什么

```
$ `doctype 文件名`
```

得不到正确结果?修改doctype,使之运行生成的命令而不只是打印命令。

练习9-4 doctype中附加的命令cat的额外开销大吗?重写doctype使之不附加进程。哪一个版本更简单些?

练习9-5 要为一个具体文档排版,既可使用doctype程序,也可以写一个包含各种命令的shell文件,试比较两者优缺点。

练习9-6 试用grep、egrep、fgrep、sed、awk和sort的不同组合,生成一个排印最快的doctype版本。

9.4 排印手册

通常来说,排版命令加工的主要文档是手册,例如页式的《UNIX程序员手册》(参见图9-1)。手册存放在标准目录(通常为/usr/man)下,分成若干按照手册章节编号的子目录。例如,我们的hoc手册就保存在目录/usr/man/man1/hoc.1中,因为hoc描述的是用户命令。

手册用man(1)命令打印,这是一个运行nroff -man的shell文件,所以 man hoc 打印hoc手册。同一名字在手册中可能不止一处出现(例如man的章节1为命令说明,而章节7为宏命令说明),可以对man制定章节。下面这个命令:

```
$ man 7 man
```

仅仅打印宏命令说明。man缺省定义是打印包含指定名字的所有nroff生成的页,而man -t 打印所有troff生成的页。

手册的作者可建立一个文件,放在/usr/man的适当子目录里。man命令用宏程序包调用nroff或troff打印手册,正如我们为调用格式化程序而搜索man命令所看到的那样,得到的结果是:

```
$ grep roff 'which man'
    nroff $opt -man $all ;;
    neqn $all | nroff $opt -man ;;
    troff $opt -man $all ;;
    troff -t $opt -man $all | tc ;;
    eqn $all | troff $opt -man ;;
    eqn $all | troff -t $opt -man | tc ;;
$
```

不同的选项有不同的含义：用nroff还是用troff；是否运行eqn，等等。由troff-man调用的手册宏命令定义troff的命令，这些命令决定手册的格式。手册宏命令基本上和ms宏命令相同，但也有差别，尤其在设置标题和字体改变命令方面。man(7)给出了简要的宏命令文件，它们都是很容易记的。手册各页的格式设计如下：

```
.TH 命令(大写) 章节号
.SH NAME
命令(小写) \- 函数的简要描述
.SH SYNOPSIS
.B 命令(小写)
选项
.SH DESCRIPTION
程序和选项的详细解释
.PP 引用的段落
.PP
新段落
.SH FILES
指定命令(如 passwd(1))所用的文件/etc/passwd
.SH "SEE ALSO"
相关文档参考，包括其他手册
.SH DIAGNOSTICS
描述异常输出(参见cmp(1))
.SH BUGS
令人惊异的特性(并不总是错误，见下)
```

如果某一节没有内容，则省略其标题。.TH行和NAME、SYNOPSIS和DESCRIPTION都是必须的。

行

```
.TH 命令(大写) 章节号
```

命名命令并说明章节编号。各个.SH行标识手册各章节。NAME和SYNOPSIS节是特殊内容，其他包含一般信息。NAME一节指定命令(小写)并给予一行说明；SYNOPSIS一节指定选项，但不作说明。而在其他节，输入格式是自由的，所以字形变化可以用宏命令.B、.I或.R来指定。在SYNOPSIS一节里，名字和选项是粗体字，其余信息为罗马体。例如，ed(1) NAME和SYNOPSIS节是：

```
.SH NAME
ed \- text editor
.SH SYNOPSIS
.B ed
[
.B \-
] [
.B \-x
] [ name ]
```

输出为：

NAME

ed - text editor

SYNOPSIS

ed [-] [-x] [name]

注意要用\表示减号-，而不能只写-号。

DESCRIPTION一节描述命令及其选项。在多数情况下，它是命令的说明，而不是命令定义的语言。手册cc(1)部分没有定义C语言，而是说明了如何运行cc命令编译C程序；怎样调用优化程序；输出放在什么地方；等等。C参考手册给出了C语言说明，在cc(1)的SEE ALSO节中引用了这些说明。换句话说，类型也不是绝对的：man(7)即描述了手册宏语言。

为方便起见，在**DESCRIPTION**一节中，命令名和选项的标识符(例如ed页中的name)按斜体字印刷，这因宏命令.I(按斜体印刷第一参量)和.IR(按斜体印刷第一个参量，罗马体印刷第二个参量)很容易实现。之所以能用宏命令.IR是因为man程序包中的宏命令.I与ms程序包中的不同，ms宏程序包很适合于处理第二参量。

FILES一节指定命令所用的文件。**DIAGNOSTICS**仅当命令产生异常输出时才有用的，这些输出可以是命令执行出现的诊断信息、退出状态或异常情况。**BUGS**一节名字起得不大合适，这里报告的缺点并不像真正的错误那样有问题，事实上，在命令安装之前，简单错误便得到了校正。为了解**DIAGNOSTICS**和**BUGS**节的具体内容，可以翻阅标准手册。

```
.TH HOC 1
.SH NAME
hoc \- interactive floating point language
.SH SYNOPSIS
.B hoc
[ file ... ]
.SH DESCRIPTION
.I Hoc
interprets a simple language for floating point arithmetic,
at about the level of BASIC, with C-like syntax and
functions and procedures with arguments and recursion.
.PP
The named
.IR file s
are read and interpreted in order.
If no
.I file
is given or if
.I file
is '\-'
.I hoc
interprets the standard input.
.PP
.I Hoc
input consists of
.I expressions
and
.IR statements .
Expressions are evaluated and their results printed.
Statements, typically assignments and function or procedure
definitions, produce no output unless they explicitly call
.IR print .
.SH "SEE ALSO"
.I
Hoc \- An Interactive Language for Floating Point Arithmetic
by Brian Kernighan and Rob Pike.
.br
.IR bas (1),
.IR bc (1)
and
.IR dc (1).
.SH BUGS
```

```
Error recovery is imperfect within function and procedure definitions.
.br
The treatment of newlines is not exactly user-friendly.
```

通过具体例子可以弄清手册排印的方法。上面给出了hoc(1)的源程序，它放在目录/usr/man/man1/hoc.1中，图9-1是由命令

```
$ man -t hoc
```

得到的印刷形式。

HOC(1)	HOC(1)
NAME hoc – interactive floating point language SYNOPSIS hoc [file ...] DESCRIPTION Hoc interprets a simple language for floating point arithmetic, at about the level of BASIC, with C-like syntax and functions and procedures with arguments and recursion. The named <i>files</i> are read and interpreted in order. If no <i>file</i> is given or if <i>file</i> is '-' hoc interprets the standard input. Hoc input consists of <i>expressions</i> and <i>statements</i>. Expressions are evaluated and their results printed. Statements, typically assignments and function or procedure definitions, produce no output unless they explicitly call <i>print</i>. SEE ALSO Hoc – <i>An Interactive Language for Floating Point Arithmetic</i> by Brian Kernighan and Rob Pike. bas(1), bc(1) and dc(1). BUGS Error recovery is imperfect within function and procedure definitions. The treatment of newlines is not exactly user-friendly.	
8th Edition	1

图9-1 命令 \$ man -t hoc输出

练习9-7 试编写doctype的手册。编排一个man命令，用它在你的man目录中寻找有关个人程序的文档。

9.5 其他文档处理工具

另外还有几个程序支持文档处理。refer(1)命令用关键字查询，并在文档末尾添加内部引用和索引节。通过定义合适的宏，可以让refer按照要求的特殊格式打印。refer中有各种计算机科学杂志名称。refer属于UNIX第7版的一部分，某些其他版本没有refer。

pic(1)和ideal(1)对图片所做工作类似eqn对公式所做工作。图片比公式更复杂(至少对排版

来说是这样)。由于没有说明图片的习惯的口头表达法, 所以这两种语言都要花费一些功夫才能领会和使用。为了说明pic的实用性, 这里提供一个用pic描述的简单图片及其描述语句。

```
.PS
.ps -1
box invis "document"; arrow
box dashed "pic"; arrow
box dashed "tbl"; arrow
box dashed "eqn"; arrow
box "troff"; arrow
box invis "typesetter"
[ box invis "macro" "package"
  spline right then up -> ] with .ne at 2nd last box.s
.ps +1
.PE
```

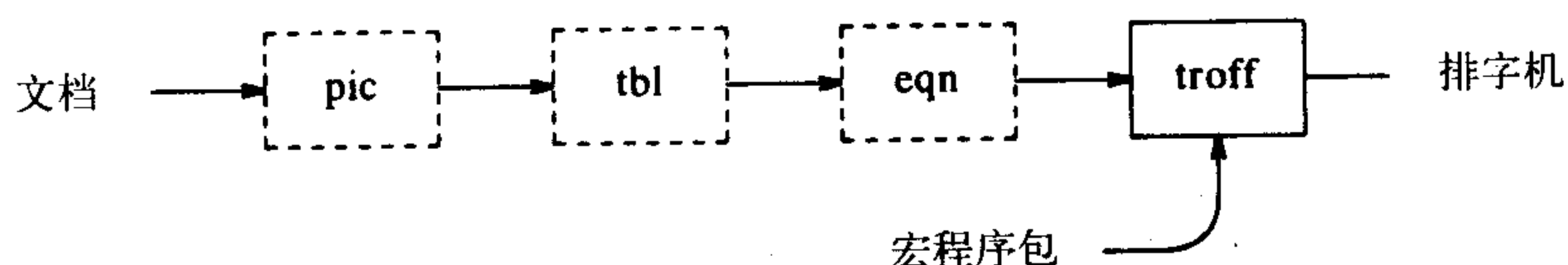


图9-2 pic描述

本书中的图片全是用pic完成的。pic和ideal不属于第7版的范畴, 但是目前是可用的。

refer、pic和ideal都是troff预处理程序, 此外还有文档检查和文档评注程序。最著名的是spell(1), 它指出文件中可能的拼写错误, spell(1)用途很广泛。style(1)和diction(1)用于分析标点、语法和语言使用。这些预处理程序的开发, 逐渐演变成Writer's Workbench程序, 它是一组协助改进书写格式的程序。Writer's Workbench程序非常适合于排除陈腐的题材、多余文字和不合时宜的措辞。

spell是标准程序。其他程序可能系统上也有, 用man不难查出系统上装有哪些程序:

```
$ man style diction ww
```

也可以通过列出目录/bin/usr/bin来查询。

有关文献

troff是已故的Joe Ossanna为Graphics System GAT-4排字机所编制的, 具有悠久的历史, 可以回溯到RUNOFF, 后者是J.E.Saltzer在60年代早期在MIT为CTSS所写的程序。这些程序共享基本的命令语法和思想, 尽管troff是其中最复杂的也是最强大的, eqn和其他预处理器的存在也极大地加强了它的功能。有几个较新的排字程序具有良好的输入格式: TEX, 作者Den Knuth (TEX and Metafont: New Directions in Typesetting, Digital Press, 1979年)和Scribe, 作者是Brian Reid ("Scribe: a highlevel approach to computer document formatting", 7th Symposium on the Principles of Programming Language, 1980年), 可能是最有名的。论文"Document Formatting Systems: Survey, Concepts and Issues", 作者Richard Furuta、Jeffrey Scofield 和Alan Shaw (Computing Surveys, 1982年9月)对该领域作了细致的调查。

有关eqn最早的论文是"A system for typesetting mathematics" (CACM, 1975年3月), 作者Brian Kernighan和Lorinda Cherry。ms宏程序包、tbl和refer均为Mike Lesk所开发; 它们仅记载在《UNIX程序员手册》卷2A中。

pic的描述出自“PIC—a language for typesetting graphics” (Brian Kernighan, Software-Practice and Experience, 1982年1月)。ideal的描述出自“A high-level language for describing pictures” (Chris Van Wyk, ACM transactions on Graphics, 1982年4月)。

spell是一个命令，它是从Steve Johnson所写的一个 shell文件转变来的一个 C 程序，作者是Doug McIlroy。第7版的spell程序使用了一个哈希(hashing)机制用于快速查找，并制定了自动剥离后缀和前缀的规则以保持字典最小。参见“Development of a spelling list” (M. D. McIlroy, IEEE Transactions on Communications, 1982年1月)。

style和 diction程序的描述位于“Computer aids for writers” (Lorinda Cherry, SIGPLAN Symposium on Text Manipulation, 1981年6月)。

第10章 结 束 语

UNIX操作系统已有十多年的历史了，但目前运行UNIX的计算机数量的增长远比以往快得多。对于一个设计初衷并没有考虑市场目标或意图的系统而言，它取得了异乎寻常的成功。

UNIX在商业上成功的主要原因也许是它的可移植性——即除了编译器和内核的小部分程序之外，所有程序都可不经修改地运行在任何计算机上。那些在其计算机上运行UNIX软件的制造商只需少量的工作就可让系统运行在新的硬件上，并可从拓展UNIX程序的商业市场方面受益。

但UNIX系统早在它取得商业上的成功之前就很受欢迎，甚至在它还只是运行在PDP-11上时就如此了。1974年Ritchie和Thompson发表在CACM上的论文引起了学术界的注意，到1975年第6版系统便在各大学盛行。整个70年代中期对UNIX的了解是通过口头传播的：尽管系统不被支持并且没有保证，但使用它的人们依然热情十足地说服别人去尝试，一旦人们试用过它之后，便往往喜欢上了它。UNIX获得目前的成功的另一个原因，是那些使用过学院UNIX系统的那一代程序员现在期望在他们工作的地方也有UNIX环境。

首先，它为什么如此受欢迎？最主要的因素是该系统是由两个极富天才的人设计并构建的，他们的唯一目的是创建一个便于程序开发的环境，而且他们也有完成该设想的自由。没有市场的压力，早期的系统小得足以令一个人明白和理解。John Lions曾在澳大利亚的新南威尔士大学为本科生的操作系统课程讲授第6版的UNIX内核，他在讲义中写到：“.....整个文档可合理地携带在学生的公文包中。”（最新的版本对这一点进行了修正。）

在早期的系统中封装了许多计算机科学中很富创造性的应用，包括流处理机制(管道)、正则表达式、语言原理(lex、yacc等)和更多的诸如diff算法这样的特定实例，把它们全都绑在一起就是一个“具有即使在较大的操作系统中也很少见的功能”的内核。作为例子，考虑一下I/O结构：一个层次型的文件系统在那时是很罕见的；设备按文件系统中的文件处理，所以不需要特别的实用程序；一打重要的系统调用，诸如带有两个参数的open原语。该软件全部用高级语言编写并随系统一起发布，所以可被研究和修改。

之后UNIX成为计算机市场上标准的操作系统之一，伴随市场优势而来的是达到竞争系统提供的“功能”所需要的责任和需要。结果是内核的大小在过去十年内增长了10倍左右，而性能并没有按相应的比例提高，这种增长伴随着过多构思拙劣、不按已有环境构建的程序。功能的蔓延为命令增加了很多的选项，从而使得原本的程序意图模糊不清，而且由于源码经常不随系统一同提供，好的风格原型难以为继。

但幸运的是，即使大的变种仍然保留了让早期版本通用的想法。UNIX基于的原则——结构的简单性、不采用不成比例的方法、基于已有的程序搭建而不是重建新的程序、命令解释器的可编程性、树型结构的文件系统等等——扩展和取代了它之前的单片机系统的概念。UNIX系统也许不会永远存在，但那些希望取代它的系统将不得不继承它的许多基本思想。

我们在前言中说过，UNIX有它自己的方法和宗旨，一种如何完成编程任务的风格。回顾

本书，你也许已从我们的示例说明中了解到这些风格的要点。

首先，让机器来完成工作。使用诸如grep、wc和awk这样的程序来使某些工作自动化，而这在其他系统中也许得手工进行。

其次，让其他人来完成工作。在构建自己的程序时使用已存在的程序，通过shell和可编程的过滤程序把它们组合在一起。编写一个小的接口程序，以便与某个已有的可真正执行任务的程序衔接起来，如我们在idiff中所做的那样。UNIX环境存在大量可用各种方法组合的工具，你的工作常常只是考虑怎样进行合理的组合。

再次，分阶段完成工作。先建立有用的最简单的程序，并根据经验来决定下一步该做的事情(如果有的话)。不急于增加功能和选项，直到使用方式能告诉你需要增加哪一些时再动手。

最后，建立工具。编写与现有环境相吻合的程序，扩展提高而不仅仅是往其中添加。扩展程序建立得好的话，这样的程序本身又可成为常用工具箱的一部分。

我们在前言中还说过UNIX并不是完美无缺的。在阅读了9章满是奇怪的约定、无意义的区别和武断的限制的程序之后，你肯定也会有同感。尽管有这样的缺点，它的正面效益仍远大于偶尔恼人的小毛病。UNIX系统确实很好地完成了当初的设计初衷：提供一个舒适的开发环境。

所以尽管UNIX已开始显露人到中年的某些征兆，但它依然富有朝气并获得了广泛的欢迎。而这种欢迎可追溯到1969年那几个人的清晰的思路，他们在黑板上画出了一个舒适的编程环境的设计概图。尽管当初他们并没有料到他们的系统会广泛应用到成千上万台计算机上，但整整一代程序员却很乐意它能够如此。

附录A 编辑器概述

“标准”的UNIX文本编辑器是一个称为ed的程序，最初由Ken Thompson编写。ed是在70年代初期设计的，适用于那些在拥有以非常低速(每秒10到15个字符)运行的硬拷贝终端的小型机(第一个UNIX系统限制用户程序为8K字节)上的计算环境。它起源于当时很流行的早期编辑器qed。

随着技术的进步，ed基本保留了原来的风貌。几乎可以肯定在你使用的系统上可以找到其他具有诱人功能的编辑器，其中最常见的是“可见”或“全屏”的编辑机制，在这样的工具中你的终端屏幕可以反映你所做的编辑修改。

那么我们为什么还要花时间讨论这样一个过时的程序？答案是尽管有些陈旧，ed确实能够很好地完成某些功能。ed是在所有UNIX平台上都通用的工具，你可以确信当从一个系统移植到另一个系统上时它就在你的手边；它可在低速电话线和任何类型的终端上良好地工作；它还可以很容易地从脚本中运行，大多数的屏幕编辑器采用自己驱动终端的方式，不能方便地从文件中获取输入。

ed为模式匹配提供正则表达式。基于ed的正则表达式渗透了整个UNIX系统：grep和sed使用的正则表达式几乎与ed完全相同；egrep、awk和lex扩展了ed的正则表达式；shell的文件名匹配正则表达式虽然语法不同但基本思想都是一致的。有些屏幕编辑器具有“行方式”以回到ed，以便使用正则表达式。

最后一点是ed运行得很快。调用ed程序、修改文件的某一行、保存改后的新版本之后退出ed，所有这些操作总共所花费的时间可能比一个较大的、引人入胜的全屏编辑器的启动时间还短。

A.1 基础

ed每次编辑一个文件。实际上它在文件的副本上工作，若要将所做的修改保存在原来的文件中，需要给出一个明确的命令。ed提供命令来操纵连续的行或与模式匹配的行，并在这些行上进行修改。

每个ed命令都是单个字符，通常是一个字母。多数命令前可冠以一个或两个行号，用于指明该命令涉及哪个或哪些行，否则使用缺省的行号。行号的说明可以用其在文件中的绝对位置(1、2、...)指定，其他特殊的行号有：符号\$代表最后一行；.表示当前行或采用正则表达式的模式搜索得到的行，以及所有这些行号的组合。

让我们来看一下如何通过ed创建一个文件，仍采用第1章中De Morgan的诗句为例。

```
$ ed poem
? poem
a
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
```

警告：文件poem不存在
开始添加行

```

    and so ad infinitum.
.
w poem
121
q
$

```

键入. 结束添加命令
把添加的行写到文件poem中
ed 报告写了121个字符
退出命令

命令a表示增加或添加行，添加方式由一个本身为.的行来结束。在ed中没有显示你正处于何种方式的标志，所以两种常见的错误是没有用a命令而键入文本以及在键入.之前键入命令。

ed不能自动地把文本写入文件，你必须用w命令来告诉它。如果试图在没有保存改变时使用退出命令q，ed会显示？给予警告，询问是否真要退出。此时，再键入q命令将直接退出编辑程序而不写文件，q命令表示无论有无改变均退出ed。

```

$ ed poem
121
a
And the great fleas themselves, in turn,
  have greater fleas to go on ;
While these again have greater still,
  and greater still, and so on.
.
q
?
w
263
q
$ wc poem
      8      46      263 poem
$

```

文件已存在，有121个字符
在文件尾添加更多的行
键入. 结束添加命令
试图退出
警告：没有保存文件
没有给文件名，则缺省值为poem
现在可以正常退出
检查文件以确认

A.2 用!暂时进入shell

当你正在使用ed时，可以暂时离开ed去运行一个shell命令，而不必退出ed。实施这个操作的ed命令是!。

```

$ ed poem
263
! wc poem
      8      46      263 poem
!
q
$

```

不退出ed而运行wc
从shell命令中返回ed
不写文件而直接退出是可行的，因为没做改变

A.3 显示

文件的行号是以数字1、2、...编排的，可以用命令np或直接用数字n来显示第n行的内容，用m, np可显示行m到行n之间的内容。行号\$表示最后一行，所以不必对行进行计数。

```

1
$

```

显示第1行，同lp
显示最后一行，同\$lp

1, \$p

从第1行开始显示到最后一行

可以按Return键每次显示一行，用命令-可以回退一行，行号还可以与符号+和-组合起来使用：

\$-2, \$p

显示最后3行

1, 2+3p

显示第1行到第5行

但不能逆序或超越最后一行进行显示，例如命令\$, \$+1p 和\$, 1p都是非法的。

显示命令l表示以所有字符都可见的方式显示文本内容，这在查找文件中的控制字符、区分空白与Tab等时很有用。(参见第6章的vis。)

A.4 模式

一旦文件长度超过几行，通过显示所有的行以查找某个特定行就很麻烦，所以ed提供了一种匹配某个特定模式的搜索方法：/模式/，以查找指定模式的下一次出现。

\$ ed poem

263

/flea/

搜索下一包含flea的行

Great fleas have little fleas

/flea/

搜索下一包含flea的行

And little fleas have lesser fleas,

//

搜索下一具有相同模式的行

And the great fleas themselves, in turn

??

反向搜索具有相同模式的行

And little fleas have lesser fleas,

ed能记住你上次使用的模式，所以可以只用//来重复搜索，反向搜索使用?模式?，或??。

用/.../ 和?...?的搜索在文本的任何一端都将“回绕”(首尾相连)。

\$p

显示最后一行。(p可有可无)

and greater still, and so on.

/flea/

下一个flea在文本开始附近

Great fleas have little fleas

??

在开始处反向回绕到文本尾部

have greater fleas to go on;

像/flea/这样的模式搜索如同1或\$一样可作为行号，也可用在同样的上下文中：

1, /flea/p

显示从第1行到下一flea出现的行

?flea?+1, \$p

显示前一flea出现的行的下一行到文本最后一行

A.5 当前行

ed记录了最后操作所在的行：如最后显示、添加文本或读一个文件时的行号，这一行被命名为.，称为当前行。每个命令都对.有一个定义，通常是将其设置为该命令影响的最后一行，可以按照与行号\$或1相同的方式使用：

\$ ed poem

263

.

显示当前行，如同读以后的\$

and greater still, and so on.

.-1, . p

显示前一行和当前行

```
While these again have greater still,
    and greater still, and so on.
```

行号表达式可以用缩写形式:

缩写	等价于	缩写	等价于
-	.-1	+	+.1
-- 或-2	.-2	++或+2	+.2
-n	.-n	+n	+.n
\$-	\$.-1	.3	+.3

A.6 添加、修改、删除、插入

添加命令a在指定的行后添加新行; 删除命令d删除行; 插入命令i在指定行前插入新行, 而修改命令c则修改行, 是删除和插入命令的组合。

n a	在第n行后添加文本
n i	在第n行前插入文本
m,n d	删除第m行到第n行
m,n c	修改第m行到第n行

如果没有给出行号, 则使用., 即对当前行进行操作。a、c和i命令的新文本内容也用本身为.的行结束, .是添加的最后一行。在删除命令后, .被设置成被删除的最后一行的下一行, 除非超过\$行号。

0a	在开始处添加文本(与命令li相同)
dp	删除当前行, 显示下一行(或为\$时显示最后一行)
., \$dp	删除当前行到结束行, 显示新的最后一行
1, \$d	删除所有的内容
?模式?, .-1d	从上一个“模式”处删除到.之前的一行
\$dp	删除最后一行, 并显示新的最后一行
\$c	修改最后一行(\$a表示在最后一行后面添加文本)
1, \$c	修改所有的行

A.7 替换、取消

仅仅为了修改其中的几个字符而重新键入一整行是很痛苦的, 替换命令s用于将某个字符串替换成另一个字符串:

s/old/new/	把当前行中的第一个old改成new
s/old/new/p	把第一个old改成new, 并显示相应行
s/old/new/g	把当前行中的每一个old改成new
s/old/new/gp	把每一个old改成new, 并显示相应行

除非跟着命令g, 否则只有行中最左边出现的模式被替换, s命令本身并不显示被修改的行, 除非在后面使用p。事实上, 大多数的ed命令都是默默地工作, 但几乎任何命令后都可以用p来显示结果。

如果替换没有按你所期望的那样做, 可使用取消命令u来取消最近的替换。.必须设置为被替换的行。

u	取消最近的替换
up	取消最近的替换并显示

正如p和d命令前可以冠以一个或两个行号以表示受影响的行一样, s命令也可以这样

使用:

<code>/old/s/old/new/</code>	查找下一个old, 并替换成new
<code>/old/s//new/</code>	查找下一个old, 并替换成new (模式已被记住)
<code>1,\$s/old/new/p</code>	把每一行中的第一个old换成new, 并显示最后被改变的行
<code>1,\$s/old/new/gp</code>	把每一行中的每一个old换成new, 并显示最后被改变的行

注意1,\$s表示将s命令应用于每一行, 但是它仍只表示替换每行最左边的匹配模式字符串, 若要替换每行中所有的模式字符串, 则需要后续的 g, 此外p只显示最后受影响的行, 要显示所有改变的行则需要全局命令, 稍后我们将作介绍。

字符&是一个缩写, 如果它出现在一个s命令的右边某处, 则表示与左边匹配的内容。

<code>s/big/very &/</code>	用very big替换big
<code>s/big/& &/</code>	用big big替换big
<code>s/./(&)/</code>	把整行用括号括起来(参见下面的.*)
<code>s/and/\&/</code>	用&(\用于取消特殊含义)替换and

A.8 元字符和正则表达式

在shell中, 符号 *、> 和 | 有特殊含义, 同样, 当ed中的某些字符出现在一个搜索模式或s命令的左边部分时, 也具有特殊意义。这类字符称为元字符, 而使用它们的模式称为正则表达式。表A-1列出了这些字符及其含义, 下面的例子应与该表结合起来阅读。任何字符前冠以反斜线符号可以取消其特殊含义。

表A-1 编辑器正则表达式

字 符	含 义
c	任何非特殊字符c都与自身匹配
\c	关闭字符c的特殊含义
^	当模式以^开始时, 匹配行首
\$	当模式以\$结束时, 匹配行尾
.	匹配任何单个字符
[...]	匹配 ... 中的任何一个字符, 像a-z这样的字符范围是合法的
[^...]	匹配不在 ... 中的任何一个字符, 范围是合法的
r*	匹配0个或多个r, 其中r为一个字符、. 或 [...]
&	只在s的右边出现, 代表匹配的模式
\(...\)	标记正则表达式; 匹配的字符串可在其左右两边, 如\1等等

注: 没有匹配换行的正则表达式。

模式	匹配内容
<code>/^\$/</code>	空行, 即只有换行
<code>/./</code>	非空行, 即至少包含一个字符
<code>/^/</code>	所有行
<code>/thing/</code>	包含thing的行
<code>/^thing/</code>	行首为thing的行
<code>/thing\$/</code>	行尾为thing的行
<code>/^thing\$/</code>	只包含thing的行
<code>/thing.\$/</code>	行尾为thing加上任意一个字符
<code>/thing\.\$/</code>	行尾为thing.
<code>/\ /thing \//</code>	包含/thing/的行

<code>/[tT]hing/</code>	包含Thing或thing的行
<code>/thing[0-9]/</code>	thing后跟一个数字
<code>/thing[^0-9]/</code>	thing后跟一个非数字
<code>/thing[0-9][^0-9]/</code>	thing后跟一个数字及一个非数字
<code>/thing1.*thing2/</code>	thing1之后为任意字符串, 然后为thing2
<code>/^thing1.*thing2\$/</code>	行首为thing1, 行尾为thing2

包含*的正则表达式匹配最左边最长的串, 注意x*可以匹配0个字符, 而xx*匹配一个或多个。

A.9 全局命令

全局命令g和v适用于由正则表达式选择的多行的一个或多个其他命令。g命令最常用于多行的显示、替换或删除:

<code>m,n g/re/ 命令</code>	对m和n之间的所有与re匹配的行执行操作cmd
<code>m,n v/re/ 命令</code>	对m和n之间的所有不与re匹配的行执行操作cmd

g或v命令之前可冠以行号来限制范围, 缺省范围为1,\$。

<code>g/.../p</code>	显示所有与正则表达式...匹配的行
<code>g/.../d</code>	删除所有与...匹配的行
<code>g/.../s//repl/p</code>	用repl替换每行中的第一个..., 并显示被改变的行
<code>g/.../s//repl/gp</code>	用repl替换每一个..., 并显示被改变的行
<code>g/.../s/pat/repl/</code>	对匹配...的行, 用repl替换第一个pat
<code>g/.../s/pat/repl/p</code>	对匹配...的行, 用repl替换第一个pat, 并显示
<code>g/.../s/pat/repl/gp</code>	对匹配...的行, 用repl替换所有的pat, 并显示
<code>v/.../s/pat/repl/gp</code>	对不匹配...的行, 用repl替换第一个pat, 并显示
<code>v/^\$/p</code>	显示所有的非空行
<code>g/.../命令1\命令2\命令3</code>	用一个g命令实施多个命令; 除了最后一个命令, 在每个命令后添加\

由g或v命令所控制的命令也可使用行号, .被依次设置为每个被选中的行。

<code>g/thing/, .+1p</code>	显示每个包含thing的行及其下一行
<code>g/^\.EQ/.1, /^ \. EN/-s/alpha/beta/gp</code>	只把.EQ和.EN之间的alpha改成beta, 并显示被改变的行

A.10 移动和复制行

命令m移动一组连续的行, 命令t可复制一组行到另一指定地方。

<code>m,n m d</code>	把第m行到第n行移到第d行之后
<code>m,n t d</code>	把第m行到第n行复制到第d行之后

如果没有指定源文本行, 则可使用.(当前行)。目的行d不能落在范围m, n-1之内。下面是使用m和t时的习惯用法:

<code>m+</code>	把当前行移到下一行之后(互换)
<code>m-2</code>	把当前行移到上一行之前
<code>m--</code>	同上, --等同于-2
<code>m-</code>	什么也不做
<code>m\$</code>	把当前行移到结尾(m0移到文本开始处)
<code>t.</code>	复制当前行(t\$复制到结尾处)
<code>-,.t.</code>	复制上一行和当前行

<code>1,\$t\$</code>	复制所有的行
<code>g/^/m0</code>	将行序逆排

A.11 标记和行号

命令=显示最后一行(\$)的行号(一个糟糕的缺省方式); .= 则显示当前行的行号, 等等, 显示后, 当前行不改变。

命令kc 用小写字母c标记被定位的行, 以后可用 'c来指定该行, k命令不改变当前行。标记适用于文本的大块移动, 因为它们(指标记)保持在行上不变, 如下面的序列:

<code>/.../ka</code>	查找行 ... 并标记为a
<code>/.../kb</code>	查找行 ... 并标记为b
<code>'a, 'bp</code>	显示整个范围以确认
<code>/.../</code>	查找目标行
<code>'a, 'bm.</code>	把选中的行移到其后

A.12 连接、分解和重组行

行可通过j命令(没有附加的空格)连接起来:

<code>m,nj</code>	把第m行到第n行连接成一行
-------------------	---------------

缺省的范围是.,.+1, 所以

<code>jp</code>	把当前行与下一行连接并显示
<code>-,.jp</code>	把上一行与当前行连接并显示

用替换命令插入一个换行来对行进行分解:

<code>s/part1part2/part1\</code>	把行分解成两行
<code>part2/</code>	...
<code>s/ / \</code>	把每个空格替换成换行
<code>/g</code>	使每行只包含一个单词

当前行被设置成新创建的最后一行。

若涉及匹配正则表达式的一部分而不是整个的话, 可使用带标记的正则表达式: 如果一个正则表达式中出现结构\(...\), 则匹配的整个表达式的这一部分可以用\1表示, 它可出现在结构的左右两边。最多可有九个标记表达式, 分别记为\1、\2, 等等。

<code>s/\(...\)\(.*\)/\2\1/</code>	把开始3个字符移到行尾
<code>/\(...*\)\1/</code>	查找一个包含相邻的重复字符串的行

A.13 文件处理命令

读和写命令r和w前可冠以行号:

<code>nr</code> 文件	读文件, 并将其添加在行n后, 把当前行设置成读取的最后一行
<code>m,nw</code> 文件	把行m到行n写到文件中, 不改变当前行
<code>m,n W</code> 文件	把行m到行n添加到文件后, 不改变当前行

w和W命令的缺省范围是整个文件; r命令中n的缺省是\$, 一个不太好的选择, 请小心。

ed记得使用的第一个文件名, 或者从命令行中得知, 或者从r或w命令中获得。文件命令f 显示或改变被记住的文件名:

f 显示被记住的文件名
f 文件 设置被记住的文件名

编辑命令**e**用记住的文件或新的文件来重新设置**ed**:

e 开始编辑被记住的文件
e 文件 开始编辑文件

e命令的保护方式同**q**命令: 如果你没有写出你的改变, 第一个**e**将引起一个错误信息, **E**则无论有无改变均重置**ed**。在某些系统上, **ed**连接到**e**上以便在编辑器内外均可使用相同的命令(**e** 文件名)。

A.14 加密

文件可通过命令**x**在写出时加密而在读入时解密, 并要求提供一个密码。加密如同**crypt(1)**中的方法, 在某些系统上**x**命令改变成**X**以使无意的加密不太容易。

A.15 命令小结

表A-2是**ed**命令的小结, 表A-3则列出了有效行号的表示法。每个命令前可冠以零个、一个或两个行号, 用以指明可提供的文本行范围以及没有指定时的缺省值。多数命令后可跟随命令**p**来显示最后受影响的行, 或跟一个**l**命令来把最后受影响的行中不可见字符标记成可见的。**.**(当前行)通常被设置为最后受影响的行, 但不受**f**、**k**、**w**、**x**、**=**和**!**命令的影响。

表A-2 ed命令一览表

命 令	含 义
.a	增加文本, 直到一个只包含.的行为止
.,.c	改变行, 新文本的结束方式同 a
.,.d	删除行
e 文件	用文件重新初始化, E 重新设置文件并且不包括文件修改
f 文件	设置记忆文件
1,\$g/正则表达式/命令	对每个匹配正则表达式的行执行 ed 的命令, 多个命令之间用\换行隔开
.i	在行前插入文本, 结束方式同 a
.,.+1j	把多行连接成一行
.kc	用字母 c 标记行
.,.l	显示行, 把不可见的字符标记成可见的
.,.m 行	把当前行移到指定行之后
.,.p	显示行
q	退出, Q 为不保存改变而退出
\$r 文件	读文件
.,.s/正则表达式/新表达式/	把任何与正则表达式匹配的内容替换成新表达式
.,.t 行	把当前行复制到指定行之后
.u	取消行中上一次的替换(仅一次)
1,\$v/正则表达式/命令	在每个不与正则表达式匹配的行中执行命令 ed 命令
1,\$w 文件	把所有行写到文件中, W 表示添加操作而不进行覆盖
x	进入加密方式(或 ed -x 文件名)
\$=	显示行号
! 命令行	执行UNIX命令
(.+1) 换行	显示行

表A-3 ed行号表示法一览表

表 示 法	含 义
<i>n</i>	绝对行号 <i>n</i> , <i>n</i> = 0、1、2 ...
.	当前行
\$	文本的最后一行
/正则表达式/	与正则表达式匹配的下一行, 当在\$行时回绕到1(从第一行开始)
?正则表达式?	与正则表达式匹配的上一行, 当在第1行时回绕到\$(从\$行开始)
' <i>c</i>	标记为 <i>c</i> 的行
<i>N1+n</i>	行 <i>N1+n</i> (附加的组合)
<i>N1</i> , <i>N2</i>	行 <i>N1</i> 到行 <i>N2</i>
<i>N1</i> ; <i>N2</i>	把当前行设置为 <i>N1</i> , 然后变到 <i>N2</i> 其中 <i>N1</i> 和 <i>N2</i> 可用上述的任何表示法指定

练习 若你认为自己对ed有所了解, 可试试编辑器quiz, 参见quiz(6)。

附录B hoc手册

hoc——交互式浮点运算语言

Brain Kernighan

Rob Pike

摘要

hoc是一个简单的有关浮点表达式的可编程解释器，具有C风格的控制流、函数定义和诸如余弦、对数等常用内部函数。

B.1 表达式

hoc是一个表达式语言，很像C：只有若干个控制流语句，诸如赋值等的大多数语句都是其值被忽略的表达式。譬如，赋值运算符 = 把右边操作数的值赋给左边的操作数，并产生值，所以多个赋值可以连接在一起工作。表达式语法为：

表达式：	数
	变量
	(表达式)
	表达式 二元操作符 表达式
	一元操作符 表达式
	函数(参数)

hoc的数是浮点数，输入格式可由scanf(3)识别，即由以下部分组成：数字、小数点、数字、e或E、带符号的指数。至少要求包含一位数字或小数点，其他成分是可选的。

变量的名字是由一个字母后跟一个字母或数字串组成，二元操作符表示诸如加法或逻辑比较等的操作，一元操作符则表示两个非操作符：！（逻辑非）和-（算术非）。表B-1列出了所有操作符。

表B-1 操作符(按优先级递减的次序排列)

操 作 符	含 义
^	指数运算，即求幂次(FOJRTAN为**), 右结合性
! -	(一元)逻辑非；算术非
* /	乘法；除法
+ -	加法；减法
> >=	关系操作符：大于；大于等于
< <=	小于；小于等于
= = !=	等于；不等于(优先级相同)
&&	逻辑与(AND, 两个操作数都要求被赋值)
	逻辑或(OR, 两个操作数都要求被赋值)
=	赋值，右结合性

下面将要介绍的函数可由用户自定义，函数的参数可用逗号分隔。hoc本身有许多内部函数，它们都带有单一的参数，如表B-2所示。

表B-2 内部函数

函 数	含 义
abs(x)	x , x 的绝对值
atan(x)	x 的反正切函数
cos(x)	x 的余弦函数
exp(x)	e^x
int(x)	对x取整，向零方向取整
log(x)	$\log(x)$, x 的以e为底的对数函数
log10(x)	$\log_{10}(x)$, x 的以10为底的对数函数
sin(x)	x 的正弦函数
sqrt(x)	x 的平方根

逻辑表达式具有两个值1.0(真)和0.0(假)，如同C语言一样，任何非0值都被认为是真值。由于所有计算都使用浮点数，相等比较只是近似的。

hoc也有一些内部常量，如表B-3所示。

表B-3 内部常量

常 量 名	数 值	含 义
DEG	57.29577951308232087680	$180/\pi$, 弧度数
E	2.71828182845904523536	e, 自然对数
GAMMA	0.57721566490153286060	γ , Euler-Mascheroni常量
PHI	1.61803398874989484820	$(\sqrt{5}+1)/2$, 黄金比例
PI	3.14159265358979323846	π , 圆周率

B.2 语句和控制流

hoc 的语句语法如下：

语句:	表达式
	变量 = 表达式
	过程(参数表)
	while(表达式)语句
	if (表达式)语句
	if(表达式)语句else语句
	{语句列表}
	print表达式列表
	return可选表达式
语句列表:	(无)
	语句列表语句

在语法分析时，缺省情况下赋值运算作为语句而不是作为表达式处理，所以交互式输入的赋值语句并不显示其值。

注意：在hoc中分号没有特殊意义，语句以换行结束，这会引起hoc的某些怪异行为。下面是合法的if语句：

```
if (x < 0) print (y) else print (z)
```

```

if (x < 0) {
    print (y)
} else {
    print (z)
}

```

在第二个例子中，花括号是必须的：if后的换行将结束语句并在省略花括号时产生语法错误。

hoc控制流机制的语法和语义与C语言基本相同，除了没有break或continue语句之外，while和if语句与C语言的while和if相同。

B.3 输入和输出：read和print

像其他内部函数一样，输入函数read也带有一个参数，但与内部函数不同的是，read函数的参数不是一个表达式，而是变量名。下一个数(如上述所定义)从标准输入读取并赋给命名变量。当读到值时，read的返回值为1(真)；当遇到文件结束或发生错误时，read返回0(假)。

输出由print语句产生，同C语言一样，print的参数是一组由逗号隔开的表达式和用双引号括起的字符串，必须提供换行，因为print不自动提供换行。

值得注意的是，read是一个特殊的内部函数，因而带有一个括起来的单一参数，而print是一个语句，带有一组用逗号分隔、且无括号的列表。

```

while (read(x)) {
    print " value is ", x, "\n"
}

```

B.4 函数和过程

hoc中函数和过程是不同的，尽管它们的定义机制相同。不同之处只在于运行时的错误检查：对过程来说，有返回值是错的；而对函数来说，没有返回值则是错的。

定义的语法是：

```

函数:      func 函数名() 语句
过程:      proc 过程名() 语句

```

其中函数名与过程名可以是任意变量名——内部函数除外。如同上面的if语句一样，定义(包括花括号或语句)部分不能超过一行。

与C语言不同的是，函数体或过程体可以是任何语句，而不必是一个复合语句(花括号括起的)。因为在hoc中分号没有特殊意义，所以空过程体由空括号组成。

函数和过程在调用时可以带用逗号分隔的参数。参数的引用方法同shell：\$3表示第3个(从1开始编址)参数；参数采用值方式传递；在函数内，它们在语义上等价于变量。如果引用一个序数大于传递给例程的参数个数的参数，将导致错误，错误检查是动态完成的。但若初始化参数影响了被引用的参数个数(如同C语言的printf)的话，一个例程就可能具有可变的参数数目。

函数和过程可以递归，但栈的深度有限(大约一百个调用)。下面给出hoc的阿克曼(Ackermann)函数的定义：

```
$ hoc
```

```

func ack() {
    if ($1 == 0 ) return $2 +1
    if ($2 == 0) return ack($1 -1, 1)
    return ack ($1-1, ack ($1, $2-1))
}
ack (3, 2)
    29
ack (3, 3)
    61
ack (3, 4)
hoc : stack too deep near line 8
...

```

B.5 实例

斯特林(Stirling)公式:

$$n! \sim \sqrt{2n\pi}(n/e)^n(1 + \frac{1}{12n})$$

```

$ hoc
func stirl() {
    return sqrt(2*$1*PI) * ($1/E)^$1*(1 + 1/(12*$1))
}
stirl(10)
    3628684.7
stirl(20)
    2.4328818e+18

```

阶乘函数n!:

```
func fac() if ($1 <= 0) return 1 else return $1 * fac($1-1)
```

斯特林公式逼近阶乘的精度:

```

i = 9
while ((i = i+1) <= 20) {
    print i, " ", fac(i)/stirl(i), "\n"
}
10 1.0000318
11 1.0000265
12 1.0000224
13 1.0000192
14 1.0000166
15 1.0000146
16 1.0000128
17 1.0000114
18 1.0000102
19 1.0000092
20 1.0000083

```

附录C hoc 清单

以下是完整的hoc6清单:

```
***** hoc.y *****

%{
#include "hoc.h"
#define code2(c1,c2)    code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
    Symbol *sym; /* symbol table pointer */
    Inst *inst; /* machine instruction */
    int narg; /* number of arguments */
}
%token <sym>  NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE
%token <sym>  FUNCTION PROCEDURE RETURN FUNC PROC READ
%token <narg> ARG
%type <inst>  expr stmt asgn prlist stmtlist
%type <inst>  cond while if begin end
%type <sym>   procname
%type <narg>  arglist
%right '='
%left OR
%left AND
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/'
%left UNARYMINUS NOT
%right '^'
%%
list: /* nothing */
    | list '\n'
    | list defn '\n'
    | list asgn '\n' { code2(pop, STOP); return 1; }
    | list stmt '\n' { code(STOP); return 1; }
    | list expr '\n' { code2(print, STOP); return 1; }
    | list error '\n' { yyerrok; }
    ;
asgn:  VAR '=' expr { code3(varpush,(Inst)$1,assign); $$=$3; }
    | ARG '=' expr
      { defnonly("$"); code2(argassign,(Inst)$1); $$=$3; }
    ;
```

```

stmt:      expr { code(pop); }
          | RETURN { defnonly("return"); code(procret); }
          | RETURN expr
            { defnonly("return"); $$=$2; code(funcrct); }
          | PROCEDURE begin '(' arglist ')'
            { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
          | PRINT prlist { $$ = $2; }
          | while cond stmt end {
            ($1)[1] = (Inst)$3; /* body of loop */
            ($1)[2] = (Inst)$4; } /* end, if cond fails */
          | if cond stmt end { /* else-less if */
            ($1)[1] = (Inst)$3; /* thenpart */
            ($1)[3] = (Inst)$4; } /* end, if cond fails */
          | if cond stmt end ELSE stmt end { /* if with else */
            ($1)[1] = (Inst)$3; /* thenpart */
            ($1)[2] = (Inst)$6; /* elsepart */
            ($1)[3] = (Inst)$7; } /* end, if cond fails */
          | '(' stmtlist ')' { $$ = $2; }
          ;

cond:      '(' expr ')' { code(STOP); $$ = $2; }
          ;

while:     WHILE { $$ = code3(whilecode, STOP, STOP); }
          ;

if:        IF { $$ = code(ifcode); code3(STOP, STOP, STOP); }
          ;

begin:     /* nothing */ { $$ = progp; }
          ;

end:       /* nothing */ { code(STOP); $$ = progp; }
          ;

stmtlist:  /* nothing */ { $$ = progp; }
          | stmtlist '\n'
          | stmtlist stmt
          ;

expr:      NUMBER { $$ = code2(constpush, (Inst)$1); }
          | VAR { $$ = code3(varpush, (Inst)$1, eval); }
          | ARG { defnonly("$"); $$ = code2(arg, (Inst)$1); }
          | asgn
          | FUNCTION begin '(' arglist ')'
            { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
          | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
          | BLTIN '(' expr ')' { $$=$3; code2(bltin, (Inst)$1->u.ptr); }
          | '(' expr ')' { $$ = $2; }
          | expr '+' expr { code(add); }
          | expr '-' expr { code(sub); }
          | expr '*' expr { code(mul); }
          | expr '/' expr { code(div); }
          | expr '^' expr { code(power); }
          | '-' expr %prec UNARYMINUS { $$=$2; code(negate); }
          | expr GT expr { code(gt); }
          | expr GE expr { code(ge); }
          | expr LT expr { code(lt); }
          | expr LE expr { code(le); }
          | expr EQ expr { code(eq); }
          | expr NE expr { code(ne); }
          | expr AND expr { code(and); }
          | expr OR expr { code(or); }
          | NOT expr { $$ = $2; code(not); }
          ;

prlist:    expr { code(prexp); }
          | STRING { $$ = code2(prstr, (Inst)$1); }
          | prlist ',' expr { code(prexp); }
          | prlist ',' STRING { code2(prstr, (Inst)$3); }
          ;

defn:      FUNC procname { $2->type=FUNCTION; indef=1; }
          | '(' ')' stmt { code(procret); define($2); indef=0; }
          | PROC procname { $2->type=PROCEDURE; indef=1; }
          | '(' ')' stmt { code(procret); define($2); indef=0; }
          ;

procname:  VAR
          | FUNCTION
          | PROCEDURE
          ;

arglist:   /* nothing */ { $$ = 0; }
          | expr { $$ = 1; }
          | arglist ',' expr { $$ = $1 + 1; }
          ;

%%

```

```

/* end of grammar */
#include <stdio.h>
#include <ctype.h>
char *progname;
int lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;
int indef;
char *infile; /* input file name */
FILE *fin; /* input file pointer */
char **gargv; /* global argument list */
int gargc;

int c; /* global for use by warning() */
yyllex() /* hoc6 */
{
    while ((c=getc(fin)) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) { /* number */
        double d;
        ungetc(c, fin);
        fscanf(fin, "%lf", &d);
        yylval.sym = install("", NUMBER, d);
        return NUMBER;
    }
    if (isalpha(c)) {
        Symbol *s;
        char sbuf[100], *p = sbuf;
        do {
            if (p >= sbuf + sizeof(sbuf) - 1) {
                *p = '\0';
                execerror("name too long", sbuf);
            }
            *p++ = c;
        } while ((c=getc(fin)) != EOF && isalnum(c));
        ungetc(c, fin);
        *p = '\0';
        if ((s=lookup(sbuf)) == 0)
            s = install(sbuf, UNDEF, 0.0);
        yylval.sym = s;
        return s->type == UNDEF ? VAR : s->type;
    }
    if (c == '$') { /* argument? */
        int n = 0;
        while (isdigit(c=getc(fin)))
            n = 10 * n + c - '0';
        ungetc(c, fin);
        if (n == 0)
            execerror("strange $...", (char *)0);
        yylval.narg = n;
        return ARG;
    }
    if (c == '"') { /* quoted string */
        char sbuf[100], *p, *emalloc();
        for (p = sbuf; (c=getc(fin)) != '"'; p++) {
            if (c == '\n' || c == EOF)
                execerror("missing quote", "");
            if (p >= sbuf + sizeof(sbuf) - 1) {
                *p = '\0';
                execerror("string too long", sbuf);
            }
            *p = backslash(c);
        }
        *p = 0;
        yylval.sym = (Symbol *)emalloc(strlen(sbuf)+1);
        strcpy(yylval.sym, sbuf);
        return STRING;
    }
    switch (c) {
        case '>': return follow('=', GE, GT);
        case '<': return follow('=', LE, LT);
        case '=': return follow('=', EQ, '=');
        case '!': return follow('=', NE, NOT);
        case '|': return follow('|', OR, '|');
        case '&': return follow('&', AND, '&');
        case '\n': lineno++; return '\n';
        default: return c;
    }
}

```

```

}
backslash(c)    /* get next char with \'s interpreted */
    int c;
{
    char *index(); /* 'strchr()' in some systems */
    static char transtab[] = "b\bf\fn\nr\rt\t";
    if (c != '\\')
        return c;
    c = getc(fin);
    if (islower(c) && index(transtab, c))
        return index(transtab, c)[1];
    return c;
}
follow(expect, ifyes, ifno)    /* look ahead for >=, etc. */
{
    int c = getc(fin);
    if (c == expect)
        return ifyes;
    ungetc(c, fin);
    return ifno;
}
defnonly(s)    /* warn if illegal definition */
    char *s;
{
    if (!lindf)
        execerror(s, "used outside definition");
}
yyerror(s)    /* report compile-time error */
    char *s;
{
    warning(s, (char *)0);
}
execerror(s, t) /* recover from run-time error */
    char *s, *t;
{
    warning(s, t);
    fseek(fin, 0L, 2);          /* flush rest of file */
    longjmp(begin, 0);
}
fpecatch()    /* catch floating point exceptions */
{
    execerror("floating point exception", (char *) 0);
}
main(argc, argv)    /* hoc6 */
    char *argv[];
{
    int i, fpecatch();
    progname = argv[0];
    if (argc == 1) {            /* fake an argument list */
        static char *stdinonly[] = { "-" };
        gargv = stdinonly;
        gargc = 1;
    } else {
        gargv = argv+1;
        gargc = argc-1;
    }
    init();
    while (moreinput())
        run();
    return 0;
}

```

```

moreinput()
{
    if (gargc-- <= 0)
        return 0;
    if (fin && fin != stdin)
        fclose(fin);
    infile = *gargv++;
    lineno = 1;
    if (strcmp(infile, "-") == 0) {
        fin = stdin;
        infile = 0;
    } else if ((fin=fopen(infile, "r")) == NULL) {
        fprintf(stderr, "%s: can't open %s\n", progname, infile);
        return moreinput();
    }
    return 1;
}

run() /* execute until EOF */
{
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    for (initcode(); yyparse(); initcode())
        execute(progbase);
}

warning(s, t) /* print warning message */
char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, " %s", t);
    if (infile)
        fprintf(stderr, " in %s", infile);
    fprintf(stderr, " near line %d\n", lineno);
    while (c != '\n' && c != EOF)
        c = getc(fin); /* flush rest of input line */
    if (c == '\n')
        lineno++;
}

***** hoc.h *****

typedef struct Symbol { /* symbol table entry */
    char *name;
    short type;
    union {
        double val; /* VAR */
        double (*ptr)(); /* BLTIN */
        int (*defn)(); /* FUNCTION, PROCEDURE */
        char *str; /* STRING */
    } u;
    struct Symbol *next; /* to link to another */
} Symbol;
Symbol *install(), *lookup();

typedef union Datum { /* interpreter stack type */
    double val;
    Symbol *sym;
} Datum;

```

```

extern Datum pop();
extern eval(), add(), sub(), mul(), div(), negate(), power();
typedef int (*Inst)();
#define STOP (Inst) 0

extern Inst *progp, *progbase, prog[], *code();
extern assign(), bltin(), varpush(), constpush(), print(), varread();
extern preexpr(), prstr();
extern gt(), lt(), eq(), ge(), le(), ne(), and(), or(), not();
extern ifcode(), whilecode(), call(), arg(), argassign();
extern funcrret(), procrret();

***** symbol.c *****

#include "hoc.h"
#include "y.tab.h"

static Symbol *symlist = 0; /* symbol table: linked list */
Symbol *lookup(s)          /* find s in symbol table */
    char *s;
{
    Symbol *sp;
    for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
        if (strcmp(sp->name, s) == 0)
            return sp;
    return 0; /* 0 ==> not found */
}

Symbol *install(s, t, d) /* install s in symbol table */
    char *s;
    int t;
    double d;
{
    Symbol *sp;
    char *emalloc();
    sp = (Symbol *) emalloc(sizeof(Symbol));
    sp->name = emalloc(strlen(s)+1); /* +1 for '\0' */
    strcpy(sp->name, s);
    sp->type = t;
    sp->u.val = d;
    sp->next = symlist; /* put at front of list */
    symlist = sp;
    return sp;
}

char *emalloc(n)          /* check return from malloc */
    unsigned n;
{
    char *p, *malloc();
    p = malloc(n);
    if (p == 0)
        execerror("out of memory", (char *) 0);
    return p;
}

***** code.c *****

#include "hoc.h"
#include "y.tab.h"
#include <stdio.h>

#define NSTACK 256

static Datum stack[NSTACK]; /* the stack */
static Datum *stackp;       /* next free spot on stack */

#define NPROG 2000
Inst prog[NPROG]; /* the machine */
Inst *progp;      /* next free spot for code generation */
Inst *pc;         /* program counter during execution */
Inst *progbase = prog; /* start of current subprogram */
int returning;    /* 1 if return stmt seen */

typedef struct Frame { /* proc/func call stack frame */
    Symbol *sp; /* symbol table entry */
    Inst *retpc; /* where to resume after return */
    Datum *argn; /* n-th argument on stack */
    int nargs; /* number of arguments */
} Frame;

```

```

#define NFRAME 100
Frame frame[NFRAME];
Frame *fp;          /* frame pointer */

initcode() {
    progp = progbase;
    stackp = stack;
    fp = frame;
    returning = 0;
}

push(d)
Datum d;
{
    if (stackp >= &stack[NSTACK])
        execerror("stack too deep", (char *)0);
    *stackp++ = d;
}

Datum pop()
{
    if (stackp == stack)
        execerror("stack underflow", (char *)0);
    return *--stackp;
}

constpush()
{
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d);
}

varpush()
{
    Datum d;
    d.sym = (Symbol *)(*pc++);
    push(d);
}

whilecode()
{
    Datum d;
    Inst *savepc = pc;
    execute(savepc+2);      /* condition */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc))); /* body */
        if (returning)
            break;
        execute(savepc+2);      /* condition */
        d = pop();
    }
    if (!returning)
        pc = *((Inst **)(savepc+1)); /* next stmt */
}

ifcode()
{
    Datum d;
    Inst *savepc = pc;      /* then part */
    execute(savepc+3);      /* condition */
    d = pop();
    if (d.val)
        execute(*((Inst **)(savepc)));
    else if (*((Inst **)(savepc+1))) /* else part? */
        execute(*((Inst **)(savepc+1)));
    if (!returning)
        pc = *((Inst **)(savepc+2)); /* next stmt */
}

define(sp) /* put func/proc in symbol table */
Symbol *sp;
{
    sp->u.defn = (Inst)progbase; /* start of code */
    progbase = progp;           /* next code starts here */
}

```

```

call()          /* call a function */
{
    Symbol *sp = (Symbol *)pc[0]; /* symbol table entry */
                                   /* for function */
    if (fp++ >= &frame[NFRAME-1])
        execerror(sp->name, "call nested too deeply");
    fp->sp = sp;
    fp->nargs = (int)pc[1];
    fp->retpc = pc + 2;
    fp->argn = stackp - 1; /* last argument */
    execute(sp->u.defn);
    returning = 0;
}

ret()           /* common return from func or proc */
{
    int i;
    for (i = 0; i < fp->nargs; i++)
        pop(); /* pop arguments */
    pc = (Inst *)fp->retpc;
    --fp;
    returning = 1;
}

funcrct()       /* return from a function */
{
    Datum d;
    if (fp->sp->type == PROCEDURE)
        execerror(fp->sp->name, "(proc) returns value");
    d = pop(); /* preserve function return value */
    ret();
    push(d);
}

procrct()       /* return from a procedure */
{
    if (fp->sp->type == FUNCTION)
        execerror(fp->sp->name,
                    "(func) returns no value");
    ret();
}

double *getarg() /* return pointer to argument */
{
    int nargs = (int) *pc++;
    if (nargs > fp->nargs)
        execerror(fp->sp->name, "not enough arguments");
    return &fp->argn[nargs - fp->nargs].val;
}

arg() /* push argument onto stack */
{
    Datum d;
    d.val = *getarg();
    push(d);
}

argassign()     /* store top of stack in argument */
{
    Datum d;
    d = pop();
    push(d); /* leave value on stack */
    *getarg() = d.val;
}

bltin()
{
    Datum d;
    d = pop();
    d.val = (*(double (*)(void)) *pc++) (d.val);
    push(d);
}

```

```

eval()          /* evaluate variable on stack */
{
    Datum d;
    d = pop();
    if (d.sym->type != VAR && d.sym->type != UNDEF)
        execerror("attempt to evaluate non-variable", d.sym->name);
    if (d.sym->type == UNDEF)
        execerror("undefined variable", d.sym->name);
    d.val = d.sym->u.val;
    push(d);
}

add()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push(d1);
}

sub()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val -= d2.val;
    push(d1);
}

mul()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val *= d2.val;
    push(d1);
}

div()
{
    Datum d1, d2;
    d2 = pop();
    if (d2.val == 0.0)
        execerror("division by zero", (char *)0);
    d1 = pop();
    d1.val /= d2.val;
    push(d1);
}

negate()
{
    Datum d;
    d = pop();
    d.val = -d.val;
    push(d);
}

gt()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val > d2.val);
    push(d1);
}

lt()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val < d2.val);
    push(d1);
}

```

```
ge()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val >= d2.val);
    push(d1);
}

le()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val <= d2.val);
    push(d1);
}

eq()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val == d2.val);
    push(d1);
}

ne()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != d2.val);
    push(d1);
}

and()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != 0.0 && d2.val != 0.0);
    push(d1);
}

or()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != 0.0 || d2.val != 0.0);
    push(d1);
}

not()
{
    Datum d;
    d = pop();
    d.val = (double)(d.val == 0.0);
    push(d);
}

power()
{
    Datum d1, d2;
    extern double Pow();
    d2 = pop();
    d1 = pop();
    d1.val = Pow(d1.val, d2.val);
    push(d1);
}
```

```

assign()
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
                    d1.sym->name);
    d1.sym->u.val = d2.val;
    d1.sym->type = VAR;
    push(d2);
}

print() /* pop top value from stack, print it */
{
    Datum d;
    d = pop();
    printf("\t%.8g\n", d.val);
}

prexpr() /* print numeric value */
{
    Datum d;
    d = pop();
    printf("%.8g ", d.val);
}

prstr() /* print string value */
{
    printf("%s", (char *) *pc++);
}

varread() /* read into variable */
{
    Datum d;
    extern FILE *fin;
    Symbol *var = (Symbol *) *pc++;
Again:
    switch (fscanf(fin, "%lf", &var->u.val)) {
    case EOF:
        if (moreinput())
            goto Again;
        d.val = var->u.val = 0.0;
        break;
    case 0:
        execerror("non-number read into", var->name);
        break;
    default:
        d.val = 1.0;
        break;
    }
    var->type = VAR;
    push(d);
}

Inst *code(f) /* install one instruction or operand */
Inst f;
{
    Inst *oprogp = progp;
    if (progp >= &prog[NPROG])
        execerror("program too big", (char *)0);
    *progp++ = f;
    return oprog;
}

execute(p)
Inst *p;
{
    for (pc = p; *pc != STOP && !returning; )
        ((*pc++)());
}

```

```

*****  init.c  *****

#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double  Log(), Log10(), Sqrt(), Exp(), integer();

static struct {          /* Keywords */
    char  *name;
    int    kval;
} keywords[] = {
    "proc",      PROC,
    "func",      FUNC,
    "return",    RETURN,
    "if",        IF,
    "else",      ELSE,
    "while",     WHILE,
    "print",     PRINT,
    "read",      READ,
    0,          0,
};

static struct {          /* Constants */
    char  *name;
    double cval;
} consts[] = {
    "PI",        3.14159265358979323846,
    "E",         2.71828182845904523536,
    "GAMMA",     0.57721566490153286060, /* Euler */
    "DEG",       57.29577951308232087680, /* deg/radian */
    "PHI",       1.61803398874989484820, /* golden ratio */
    0,          0
};

static struct {          /* Built-ins */
    char  *name;
    double (*func)();
} builtins[] = {
    "sin",      sin,
    "cos",      cos,
    "atan",     atan,
    "log",      Log, /* checks range */
    "log10",    Log10, /* checks range */
    "exp",      Exp, /* checks range */
    "sqrt",     Sqrt, /* checks range */
    "int",      integer,
    "abs",      fabs,
    0,          0
};

init() /* install constants and built-ins in table */
{
    int i;
    Symbol *s;
    for (i = 0; keywords[i].name; i++)
        install(keywords[i].name, keywords[i].kval, 0.0);
    for (i = 0; consts[i].name; i++)
        install(consts[i].name, VAR, consts[i].cval);
    for (i = 0; builtins[i].name; i++) {
        s = install(builtins[i].name, BLTIN, 0.0);
        s->u.ptr = builtins[i].func;
    }
}

*****  math.c  *****

#include <math.h>
#include <errno.h>
extern int    errno;
double errcheck();

double Log(x)
    double x;
{
    return errcheck(log(x), "log");
}

```

```

double Log10(x)
    double x;
{
    return errcheck(log10(x), "log10");
}

double Sqrt(x)
    double x;
{
    return errcheck(sqrt(x), "sqrt");
}

double Exp(x)
    double x;
{
    return errcheck(exp(x), "exp");
}

double Pow(x, y)
    double x, y;
{
    return errcheck(pow(x,y), "exponentiation");
}

double integer(x)
    double x;
{
    return (double)(long)x;
}

double errcheck(d, s) /* check result of library call */
    double d;
    char *s;
{
    if (errno == EDOM) {
        errno = 0;
        execerror(s, "argument out of domain");
    } else if (errno == ERANGE) {
        errno = 0;
        execerror(s, "result out of range");
    }
    return d;
}

```

```

***** makefile *****

YFLAGS = -d
OBS = hoc.o code.o init.o math.o symbol.o
hoc6: $(OBS)
    cc $(CFLAGS) $(OBS) -lm -o hoc6

hoc.o code.o init.o symbol.o: hoc.h
code.o init.o symbol.o: x.tab.h
x.tab.h: y.tab.h
    -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
pr: hoc.y hoc.h code.c init.c math.c symbol.c
    @pr $?
    @touch pr

clean:
    rm -f $(OBS) [xy].tab.[ch]

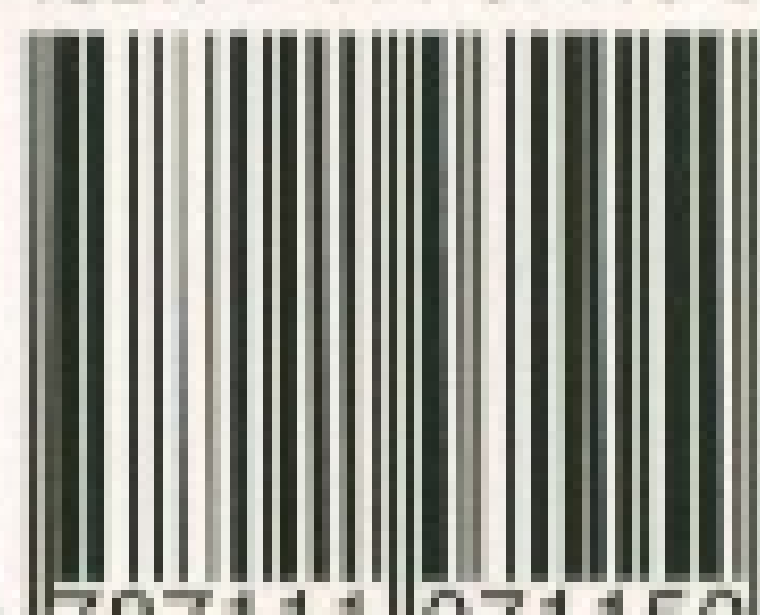
```



计 算 机 科 学 丛 书

- ❖ Internet技术基础 (18.00元)
- ❖ 数据结构算法与应用——C++语言描述 (英文版 65.00 元)
(中文版)
- ❖ 高级计算机体系结构 (英文版 59.00元)
- ❖ 软件工程: Java语言实现 (英文版 51.00 元)
(中文版 38.00 元)
- ❖ 软件工程: 实践者的研究方法 (英文版 68.00 元)
(中文版 48.00 元)
- ❖ 数据库系统概念 (英文版 63.00 元)
(中文版)
- ❖ 通信网络基础 (英文版 32.00元)
- ❖ 现代操作系统
- ❖ 编译原理: 结构与实践
- ❖ 专家系统原理与编程
- ❖ 计算理论基础
- ❖ UNIX编程环境 (24.00元)
- ❖ Java编程思想 (60.00元)
- ❖ C++编程思想
- ❖ 数据库管理系统基础 (20.00元)
- ❖ 离散数学及其应用 (英文版 59.00元)
- ❖ 数据通信与网络 (英文版 59.00元)
- ❖ 并行计算机体系结构 (英文版 88.00元)
- ❖ 计算机体系结构: 量化研究方法 (英文版 88.00元)
- ❖ 计算机组织与设计: 硬件/软件接口 (英文版 80.00元)
- ❖ 人工智能 (英文版 45.00元)

ISBN 7-111-07115-8



9 787111 071150



北京市西城区百万庄大街22号 100037
购书热线: (010) 8006100280

www.hzbook.com

ISBN 7-111-07115-8/TP · 1034
定价: 24.00 元

计 算 机 科 学 丛 书

UNIX 编程环境

The UNIX
Programming
Environment

(美) Brian W. Kernighan Rob Pike 著
陈向群 等译



机械工业出版社
China Machine Press

Prentice Hall

[G e n e r a l I n f o r m a t i o n]

书名=U N I X 编程环境

作者=B r i a n W . K e r n i g h a n & R o b P i k e

页数= 2 5 5

出版日期= 1 9 9 9 - 1 0

出版社= 机械工业出版社

I S B N = 7 - 1 1 1 - 0 7 1 1 5 - 8

书名
版权
前言
目录
正文