



微计算机丛书

UNIX操作系统

[美] K·克里斯琴 著

孙玉芳 董士海 译

电子工业出版社



封面设计：薛太忠

统一书号：15290·289 定价：2.70 元

UNIX操作系统

[美] K·克里斯琴 著
孙玉方 董士海 译
杨芙清 校



000462

电子工业出版社

UNIX操作系统

[美] K·克里斯琴 著
孙玉方 董士海 译
杨芙清 校



000462

电子工业出版社

内 容 简 介

本书是UNIX系统方面较全面的著作。全书内容包括：UNIX系统的历史、基本原理、UNIX系统基础、shell、编辑程序、文件系统、实用程序的应用、正文文件实用程序、管理文件、高级编辑、正文格式加工、make和源代码控制系统、shell程序设计、C语言和UNIX系统、程序员用的实用程序、 yacc和lex、系统管理员用的实用程序、UNIX系统核心。

本书可作为大学、中专计算机专业教材，也可作为计算机领域中的科技人员和广大用户的学习材料。

[美] Kaare Christian 著

The UNIX Operating System

John Wiley & Sons, Inc. 1983版

UNIX 操作系统

孙玉方 董士海 译

杨美清 校

责任编辑：路石

※

电子工业出版社出版（北京市万寿路）

新华书店北京发行所发行 各地新华书店经售

人民卫生出版社印刷厂印刷

※

开本：850×1168 1/32 印张：12.3125 字数：327千字

1986年3月第1版 1986年5月第1次印刷

印数：1—15,000 册 定价：2.70元

统一书号：15290·289

前 言

UNIX是一组计算机操作系统的名字。许多专家认为,UNIX系统是过去十年中在计算机操作系统方面最重要的进展。UNIX系统的冲击力可以和FORTRAN或者IBM/360系统比较,前者是第一个主要的可移植的高级程序设计语言,后者是第一个具有广泛的性能范围的可兼容的计算机系列。UNIX系统因为可以在各种不同的计算机上运行并且有许多应用而变得日益重要。

最初,这个系统在数字设备公司(DEC)的小型机上用作科学研究、程序开发和资料准备。现在,UNIX的最新版本可以在各种计算机上运行。这些计算机小到并便宜到象Zilog的Z-80,大到象阿姆得尔(Amdahl)470/V7。UNIX系统当前应用在商业、科学和工业应用的各种领域里。

操作系统亦称为“经理”,因为它们(就象商业经理一样)管辖和控制一个复杂设备的运行。从狭义上说,UNIX系统把计算机的时间分成若干小的等分,并且在各个用户之间分配这些时间。此外,它在中央计算机和磁盘、磁带、终端和打印机之间控制信息流向并且管理信息的长期储存。

从广义上说,UNIX系统是依赖于上述基本服务的众多程序的大型集合体。这些程序允许用户建立并且考查文件,编写并且检验新的程序,执行复杂的资料准备工作,简言之,就是管理信息。UNIX系统是一个简单,有效并且易懂的系统。

UNIX最主要的长处之一是它可以在各种不同类型的计算机上运行。(在UNIX系统控制下)某类计算机上运行的普通程序通常不作修改或作很少的修改就可以在别的类型的计算机上运行。这样,出售计算机程序的公司就只要为广大的市场生产单独一种

产品，计算机用户在许多不同机器上也只需使用单独一种操作系统。虽然这看起来似乎很简单，但是，它却是在计算机方面花了三十年才获得的成果。

贝尔实验室的K·汤普逊 (Ken Thompson) 和D·M·里奇 (Dennis M. Ritchie)的工作在70年代早期开始了一场革命。70年代中期，对UNIX的兴趣扩展到学术界，他们当时正在为现代计算机科学中的许多困难寻找解决方法。80年代，兴趣扩展到商业界，它们把UNIX系统看成是为提供更好和更持久的软件产品和服务的一个重要部分。

汤普逊和里奇最初的目标是为开展计算机科学研究创造一个多产的环境。现在看来，不但最初的目标都已达到了，而且还达到了另外更多的目标，汤普逊和里奇的成就已经展示了，即使是个人，也能在一个困难的领域里作出重要的贡献。UNIX系统的新和更好的思想已经最终被商业界所采纳。

UNIX系统在第一个十年(1970~1980)中，主要用在学术界和贝尔系统内部。这是一个提炼和考验时期，它证明在通用的交互型操作系统领域中，这个系统具有无可置疑的优越性。最近的十年表明，UNIX系统进入了商业市场，在那里它迅速成为许多不同应用的各种计算机系统的标准操作系统。

要注意，与其它系统相比，UNIX系统有两个主要的不足之处。在核心部分，UNIX系统是无序的。如果系统中的每一个用户做的事都不同，那么UNIX系统工作得很好；但是，在每一个用户都要做同一事情时，就会引起麻烦。然而，如果知道每个人下面将要干些什么(支票处理系统，飞机票预约系统等)，那么，就可以优化该系统。优化UNIX系统会使得每个用户都能互不干扰地独立工作。

第二个不足之处是在实时方面。实时系统用于控制机械、工业过程、实验室仪器以及类似的东西。计算机在机器运行、过程进行或实验数据到达时必须迅速作出反应。虽然采用UNIX系统

来完成这些操作中的大部分工作具有一定的可能性；但是有另外一些操作系统，它们在实时应用中运行得要比UNIX系统更好。

过去，许多人学会了使用UNIX系统，70年代中期，在贝尔系统外部学会UNIX操作系统的大多数人，会告诉你辅导他们的人员的姓名。贝尔实验室以UNIX程序员手册(The UNIX Programmer's Manual)作为UNIX系统的资料，这是一份相当完整和精确的参考资料。但是很遗憾，贝尔实验室的这份手册太难了，它不适合于初学者和只有中等水平的用户使用。在使用方面，提供信息的其它主要来源的是一些学术刊物上的文章，它们向计算机科学家提供了UNIX系统和其主要程序的详细解释。

本书的第一部分是为了满足初学者和中等水平的用户有效地使用UNIX系统的需要，向他们提供了基本知识并且介绍了基本思想。第一部分的各章向用户介绍交互计算，解释shell命令和编辑程序的基本用法，解释隐含在UNIX文件系统中的某些思想，概述最有用的实用程序，并且不断用例子解释UNIX系统的思想，显示其能力和灵活性。如果你在UNIX系统中试验这些例子，那么，掌握UNIX系统会快得多。

UNIX系统手册作为参考材料很好，但作为介绍性读物却很差。例如，在我们得到的手册中，对pwd(印出工作目录)命令的全部说明只有“pwd印出工作目录的路径名”，实在太简单了。本书的一个作用是弥补这一点。它解释了路径名，解释了工作目录的表示，解释了为什么以及什么时候要知道工作目录的名字，并且提供使用pwd的例子。只要熟悉了UNIX系统，就可以通过使用UNIX系统手册来扩充知识，但是在开始，更为精细地介绍是会有所帮助的。

本书的另一个作用是把UNIX系统中普遍并且常用的实用程序划分了出来，UNIX系统提供了200多个实用程序，然而，我认为这些程序中只有大约40个是值得在本书中单独作解释的。第七、八和九章，按功能分别介绍了这些实用程序。这些程序也按

字母顺序在本书末尾UNIX系统简明手册中作了介绍。

本书的第二部分给中等水平和熟练的用户介绍了一批资料。本书这部分最重要的课题要算第七版的shell，即UNIX系统的标准命令的解释程序，shell极其重要，在文献中至今还未作过充分的介绍。本书第二部分还包括以下一些专题：UNIX系统的内部组织，程序员和管理员需要的信息以及某些最富于创新的UNIX系统程序的介绍性说明。

在UNIX系统简明手册后面列出了UNIX系统术语词汇表，这对于大多数初学者应该是有益的。

K·克里斯琴

译 者 的 话

UNIX 系统是一个通用、多用户的计算机分时系统，现已成为高档微型机和若干小型机系列上的主要操作系统，目前正广泛应用于教学、科研、工业和商业等多个领域。

本书是目前UNIX系统方面较为全面、系统的著作，它以大量的实例深入地介绍了UNIX系统外部面貌和内部结构。作者有比较丰富的使用UNIX系统和开发软件的实际经验，提出的许多见解中肯贴切，读后使人受益非浅。

全书共分十九章，前一部分十一章讲述了UNIX系统的基本部分，主要涉及了UNIX系统基础，基本shell命令，编辑程序，实用程序及正文格式加工；后一部分八章，论述了UNIX系统更为深入的一些课题，主要涉及软件开发工具——系统构造程序，源代码控制系统，词法和语法生成程序，shell程序设计方法和shell程序，C语言，程序员和系统管理员使用的实用程序，以及UNIX系统核心；最后是一个UNIX系统简明手册，列出了一些主要命令的格式、使用注意事项及有关例子。

在译校过程中发现原书有少量错误和遗漏，译文中均作了改正。专用术语的译名尽量与惯用的一致，但因为有些术语或者是第一次出现，或者因为至今仍无统一的译名，所以我们只好根据原义自己拟定。本书中的“我”代表本书作者，“你”代表读者或计算机用户。

限于译者水平，错误和不妥之处仍在所难免，请读者批评指正。

一九八三年十一月

目 录

第一部分 UNIX 系统基础

第一章	UNIX系统的历史	1
第二章	基本原理	5
2.1	低级的功能	5
2.2	典型的计算机	6
2.3	裸机	8
2.4	操作系统	10
2.5	分时	11
2.6	核心	11
2.7	程序	12
2.8	shell和编辑程序	13
第三章	UNIX系统基础	16
3.1	注册	16
3.2	某些简单的命令	19
3.3	文件和目录	21
3.4	与UNIX系统对话	23
3.5	注销	27
3.6	UNIX系统手册	27
第四章	UNIX系统shell	30
4.1	简单的shell命令	30
4.2	命令自变量	31
4.3	后台进程	33
4.4	标准输出和标准输入	34

4 . 5	输出重新定向	35
4 . 6	输入重新定向	37
4 . 7	管道	41
4 . 8	元字符和文件名生成	45
4 . 9	小结	49
第五章	UNIX 系统编辑程序	50
5 . 1	正文文件	52
5 . 2	行编辑	54
5 . 3	启动编辑程序	55
5 . 4	基本的编辑命令	56
5 . 5	把正文加到工作文件中	58
5 . 6	印出文件行	59
5 . 7	更新原先的文件	60
5 . 8	结束编辑工作	60
5 . 9	行和行号	61
5 . 10	删除正文行	64
5 . 11	插入和修改正文行	66
5 . 12	移动和传送正文行	67
5 . 13	替换正文	68
第六章	UNIX 文件系统	72
6 . 1	普通文件	73
6 . 2	目录文件	75
6 . 3	具有层次结构的文件系统	76
6 . 4	路径名	76
6 . 5	文件类型和方式	83
6 . 6	特别文件	85
6 . 7	目录存取方式	87
第七章	实用程序的应用	89
7 . 1	pwd和cd——显示和改变当前目录	90

7 . 2	ls——列出文件	91
7 . 3	file——推断文件类型	99
7 . 4	date和who——设置或显示日期和显示当前用户	99
7 . 5	ps——列出进程	100
7 . 6	kill——消灭后台进程	100
7 . 7	nohup——在退出系统之后运行程序	101
7 . 8	nice——以低优先权运行进程	102
7 . 9	time——计算进程的时间	103
7 . 10	man——印出手册的条目	103
7 . 11	passwd——改变注册口令	104
7 . 12	echo——回应命令行自变量	104
7 . 13	find——检索一个文件	105
7 . 14	mail和write——与其它用户通信	106
7 . 15	stty和tty——终端处理程序	108
7 . 16	du——查看磁盘使用情况	111
7 . 17	od——卸出 (dump) 文件	112
第八章 正文文件实用程序		114
8 . 1	正文实用程序	114
8 . 2	cat——印出文件	115
8 . 3	pr——给文件加标题并且进行格式加工	117
8 . 4	lpr——打印文件	117
8 . 5	wc——统计行数、字数和字符数	119
8 . 6	diff——比较文件	120
8 . 7	sort——重排文件	121
8 . 8	grep——在文件中寻找正文模式	122
8 . 9	cut和paste——重排文件的列	124
8 . 10	spell——寻找拼写错误	125
8 . 11	crypt——为文件加密	126
8 . 12	tee——复制输出	127

8 . 13	tail ——印出文件尾	127
第九章	管理文件	129
9 . 1	rm ——删除文件	129
9 . 2	mv, cp和ln ——移动和复制	131
9 . 3	chmod, chown和chgrp ——改变文件方式	134
9 . 4	mkdir和rmdir ——建立和删除目录	135
第十章	高级编辑	137
10 . 1	把正文读到工作文件中	137
10 . 2	文件命名命令	138
10 . 3	全局命令	139
10 . 4	粘连命令	142
10 . 5	正则表达式	143
10 . 5 . 1	在正则表达式中的特殊字符	143
10 . 5 . 2	单字符正则表达式	145
10 . 5 . 3	组合单字符正则表达式	146
10 . 6	再论替换命令	147
10 . 7	在编辑程序中使用shell命令	152
10 . 8	开放行编辑和屏幕编辑	153
第十一章	正文格式加工	155
11 . 1	nroff和 troff ——对正文进行格式加工	156
11 . 2	使用宏程序包	158
11 . 3	tbl ——加工表格	162
11 . 4	eqn ——加工数学公式	163

第二部分 UNIX 系统更深入的课题

第十二章	make 和源代码控制系统(SCCS)	165
12 . 1	大型程序	166
12 . 2	make	167

12.3	源代码控制系统 (SCCS)	173
第十三章	shell 程序设计语言	179
13.1	执行shell程序	180
13.2	shell变量	182
13.3	交互地使用shell变量	184
13.4	查找路径	185
13.5	引用	187
13.6	set命令	189
13.7	简单的条件	190
13.8	简单的命令、管道线和命令表	192
13.9	if条件	193
13.10	shell程序变量	195
13.11	while和until语句循环	197
13.12	结构化的命令	198
13.13	命令替换	199
13.14	shell替换	201
13.15	here文件	202
13.16	for语句	203
13.17	case语句	205
13.18	break和continue	206
第十四章	一些shell程序	208
14.1	何时使用shell程序设计语言?	208
14.2	多少用户?	209
14.3	更新一个记帐文件	210
14.4	列出子目录	214
14.5	列出当前子树中的文件	217
第十五章	C语言和UNIX系统	220
15.1	标准子程序	221
15.2	输入/输出系统调用	223

15.3	有关状态的系统调用	225
15.4	控制进程的系统调用	227
15.5	将自变量传递给程序	231
15.6	系统调用的实现	234
15.7	分别编译	236
15.8	lint——检查C语言程序	239
第十六章	程序员用的实用程序	240
16.1	编译	240
16.2	size——印出目标文件的特性	242
16.3	strip——从目标文件中删除符号表	243
16.4	nm——印出目标文件的符号表	243
16.5	ar——档案文件	244
16.6	ld——组合目标文件	246
第十七章	yacc和lex	248
17.1	词法分析和语法分析	248
17.2	lex	251
17.3	yacc	257
第十八章	系统管理员用的实用程序	263
18.1	安全性	264
18.2	su——成为超级用户	265
18.3	安装及拆卸文件系统	266
18.4	sync——誊清系统缓冲区	271
18.5	mknod——建立特别文件	272
18.6	df——印出磁盘空闲区	274
18.7	volcopy, labelit, dump, restor cpio——后援	275
18.8	dd——转换文件	276
18.9	fsck, fsdb——检查文件系统	277
18.10	cron——在指定时间运行程序	279
18.11	先进先出fifo文件	280

18.12	粘着位 (sticky bit)	281
18.13	调整用户标识 (set user id)	281
第十九章	UNIX 系统核心	283
19.1	概述	284
19.2	用户态和核心态	285
19.3	调度和对换	286
19.4	进程	287
19.5	引导、进程 0 和进程 1	292
19.6	文件系统	297
19.7	外部设备	306
	UNIX 系统简明手册	312
	小词典	359

第一部分 UNIX系统基础

第一章 UNIX系统的历史

计算机从高级技术的神秘王国进入了人类日常活动，UNIX系统是这一进展中的主要步骤之一。UNIX操作系统已经显示出，一个强有力的操作系统可以与机器基本上无关。UNIX系统也已经证明，在使用计算机解决问题的过程中，人们能够有效地使用强有力的软件工具。

UNIX系统提供的服务与其它操作系统所提供的基本上一样：它允许你运行程序；它为连接到大多数计算机的各种各样的外部设备（打印机、磁带、磁盘、终端等等）提供了方便和一致的接口；它还信息管理提供了文件系统。UNIX系统的独到之处主要是由于它所开拓的思想。通过历史回顾，我们可以更好地理解该系统越来越受欢迎的原因。

UNIX系统的流行其部分原因是它的可移植性、灵活性和精巧的设计。具有讽刺意义的是，UNIX系统是从世界上最强大的公司之一，即贝尔实验室的阁楼上产生出来的，贝尔实验室是美国电话电报公司的一个附属研究机构。绝大多数操作系统是计算机厂家为了出售计算机而开发出来的，而美国电话电报公司不经营出售计算机的业务，因此，UNIX系统原先并不想成为一种商品，它之所以成为商业竞争品，仅仅是为了满足日益扩大的市场需要。

在六十年代后期，贝尔实验室与一个称为Multics的操作系统有关。Multics是使用GE大型计算机的一种多用户交互系统。

1969年贝尔实验室从Multics工程中退出。但是Multics对UNIX系统具有重要的影响。事实上，UNIX这一名字不过是在Multics这一词义上玩的游戏。UNIX系统与Multics最明显的差别是在复杂性方面，UNIX操作系统相对地简单而Multics极端复杂。

大约与此同时，从Multics中退出的贝尔系统的一个成员，“UNIX系统的祖师爷”K·汤普森在DEC的PDP-7小型计算机上开始拼凑一项工程。汤普森企图创建一个操作系统，它能够在程序设计环境中支持程序员进行协调一致的工作。回顾起来，这一目标目前已经成功地达到了。另外，为了满足管理上的需要，汤普森建议贝尔实验室支持UNIX系统进一步的开发，以便为贝尔实验室的专利组织的资料准备工作提供支持。1971年，在一台PDP-11/20上的UNIX系统的早期版本最终提交给了贝尔实验室的专利组织。

从一开始，两个似乎不相容的科目：程序设计和资料准备就已经是UNIX系统的基石。在实践中，UNIX系统已经显示出，正文处理工具是许多工作包括程序设计在内的中心。有人批评说，UNIX系统仅仅是一种特制的字处理器。然而，即使有的应用可用非UNIX操作系统来支持，但是正文处理上的特长已经使UNIX系统成为极其通用的操作系统。正文是通信可以接受的介质，这是通用操作系统的一个关键特性。

K·汤普森原先的工作主要是创建一个操作系统、一个PDP-7汇编程序以及若干汇编语言实用程序。在1973年D·里奇用C程序设计语言重写了UNIX系统。C是由里奇开发的一种通用的高级程序设计语言。业已证明，它适合于许多不同类型的计算机体系结构。要是UNIX系统未用高级语言重写的话，它就会与它依赖于开发的机器（过时的PDP-7）完全连在一起了。在原先的汇编语言程序用C语言重写之后，没有多大困难就立即可以把整个UNIX系统从一个环境移到另一个环境中。

按惯例，操作系统是依赖于一台计算机或一个计算机系列的，

因为它们是用汇编语言编写的。虽然UNIX系统原先并未打算成为一个可移植的操作系统，但是一旦它用C语言写了以后，就有条件把所有内容都移到其它系统上。第一次移到不同类型的计算机上的工作是由里奇和S·约翰逊（S·Johnson）在1976年完成的。当时他们把UNIX移到了Interdata 8/32上。自从那时以来，UNIX系统已经移到十多种计算机上，范围从Zilog、Z-80、Z-8000、Motorola的MC68000和Intel 8086那样的单片微处理器到IBM 370和Amdahl 470那样的大型机。

在70年代早期，汤普森从他的同事和上级处获得认可以后，UNIX系统就开始在整个贝尔系统内部使用了。在有关这个操作系统的各种传说传开以后，若干有影响的高等院校对它产生了兴趣。1975年西方电气（Western Electric）开始颁发UNIX系统的许可证。对于高等院校，要求他们支付的费用极其微少，为的是鼓励它们使用并且进一步开发UNIX系统。不过因为UNIX系统被学术界的高级技术研究团体说得太好了，所以一开始商业界反而对它表示怀疑。但是近来商业界已经意识到。他们可以针对广泛的应用轻而易举地采纳UNIX系统。从70年代后期开始，一个支持UNIX系统的硬件、软件和有关服务的工业已经产生了。

UNIX系统已经开创若干重要的思想。最重要的发明之一是管道（pipe），由它进一步导致这样的思想：复杂的功能可以编制成一组在一起工作的程序。排版程序是一个很好的例子。UNIX系统包含若干不同的排版程序：一个用于一般事务处理，一个用于数学公式，一个用于表格数据，还有一个用于图表。管道连接使得用户需要多少程序就使用多少。每一个排版程序都不重复其它程序中的特性；相反，却有一种相互补充的关系。业已证明用一批协同操作的进程来解决一个复杂问题的方法，对于程序开发者和用户都是很方便的。

贯穿整个UNIX系统的另一个思想是软件工具概念。这种思想并不是UNIX系统所独有的，但是比起其它系统来说，这种思想

想在UNIX系统中得到了更进一步的发展。为了简化涉及命令语言识别的程序设计中的繁杂工作，UNIX系统备有yacc和lex。这两个程序使得程序员可以用表格形式实现命令语言解释程序，而不是为这语言单独编写一个解释程序。要掌握lex和yacc，先要对它们下功夫加以研究，但是一旦掌握了它们，就有可能方便地编制出涉及命令语言的新的应用程序。复杂工具的另外两个例子是make和源代码控制系统（SCCS）。make用于说明一个软件系统中各模块的相互依赖性，以便可以自动地维护这个系统。SCCS用于跟踪一个软件系统直到它变成熟为止，在此期间可以恢复它的老版本，也可以整理新版本资料。

至此，UNIX系统已作为各种计算机上的通用标准操作系统出现了。UNIX系统似乎不会广泛地使用在需要配置专门的操作系统、用作为特殊目的（例如，事务处理、预订系统、实时系统）的场合下。从实用角度看，UNIX系统从它诞生这些年来一直是很重要的，因为它得到了广泛的应用。按照这一说法，UNIX系统工业的确是方兴未艾。从金额上说，UNIX系统工业不能和那些已成为大型操作系统的服务工业相比。然而，在以后的几年内，对于UNIX系统服务工业中的那些公司来说，前景是很美好的，某些工业分析家还预言，UNIX系统工业会与今天存在的较早建成的操作系统的支持工业相匹敌。

UNIX系统未来的另一个侧面与计算机学术界的广泛的承认和使用有关。有点象程序设计语言ALGOL（以及最近的PASCAL）广泛地用在描述算法的学术著作中那样，UNIX操作系统在讨论到操作系统课题的各种著作中正得到广泛地引用。在衡量操作系统新发展时，UNIX系统已成为一种标准。

第二章 基本原理

计算机与其它机器的差别在于计算机是通用的。而计算机的通用性正成为学会使用计算机的主要困难。为理解UNIX系统或其它任何操作系统,第一步应该对于计算机的结构有一般的了解。本章介绍这些结构的一些基本原理。

操作系统的主要功能之一是可以不管计算机的结构。你不必为操作电子设备而要去懂得马迹和线路理论,你也不一定为了使用计算机而要去懂得计算机的体系结构。然而,有了对于计算机基本原理的一般了解,将使得你对以下各章中一些思想的理解来得更为容易。如果你对计算机已有了某些经验,那么可以跳过这一章。

2.1 低级的功能

电传打字机很容易使用,为了在纸上打印一个特定字符,只要按一个特定的键。象抬起打字机托架并且返回到行首那样一些普通操作虽然十分复杂,但已经构造到机械装置中而变成自动的了。电传打字机把按键信号翻译成一系列机械动作,从而产生预期的结果。这种翻译机构的目的是可以不管基本机械动作,使电传打字机使用起来更容易。

为了更为清晰地阐述上述思想,让我们进一步具体地加以解释。当你在一架典型的电传机键盘上敲了一个字母“a”时,就产生下列动作:(1)色带升起,(2)“a”打印锤撞击挡板,(3)色带落下,(4)电传机移动到下一个打印位置(除非已经走到了边缘),(5)当移到距右边缘有确定数目的空格时,电传打字机就响铃。这样电传打字机就把一次按键翻译成一系列内部动作,

打印出一个字母。

我们可以假想，有架缺少把按键信号翻译成一系列机械动作的电传机。词“proto”的一种意思是“最早的形式”。我们用术语“模型打字机 (prototypewriter)”来描述这种可以完成电传打字机的所有低级功能的机器。一架模型打字机可以升降色带，来回移动装置，对着挡板撞击符号等等。然而，我们假想的模型打字机缺少电传机响应单个按键打印一个字母的高级功能。为了使用这种模型打字机，你得记住一系列的基本操作，这些操作是为了完成每一个高级功能（例如，打印一个字母“a”）所需要的。人们可以要求模型打字机比一架通常的电传机更强有力、更通用，可以从左到右地、对角式地或垂直地打印。然而，这种模型打字机是极其简陋的，我想在市场上难以看到这种打字机。

计算机有点象模型打字机，它有非常有用的潜力，但是它不具有方便的高级控制机构。为了创建一种方便并且有用的设备，把翻译机建造到模型打字机中去是很容易的，这是因为电传机是一种用途单一的设备。然而要让计算机具有一组方便的操作却要困难得多，这是因为计算机是通用的机器。操作系统的作用就是使通用的计算机更容易使用，正如键盘翻译机构提供给模型打字机一组有用的操作那样，操作系统也提供给计算机一组有用的功能。

2.2 典型的计算机

已经生产了许多不同种类的计算机。虽然这些计算机有很大的不同，但是由于市场的压力及技术的进展已经导致对主要功能部件作了一些标准化。

计算机本质上是执行一系列指令的机器。它根据指令执行若干操作，比如把两个数字相加，把一些信息从一个地方移到另一个地方，或者改变指令执行的顺序等等。计算机中执行指令的这部分称为处理机（中央处理部件，简称为CPU），而指令存放的地

方称为存储器。CPU是计算机中处理信息的部分，但是在CPU中只存放相对来说比较少量的信息。存储器是存放信息的地方；存储器中的每一个存储单元被赋予唯一的一个号码，叫做地址。存储器常称为主存，因为它是CPU获得其指令的地方。

主存的主要优势是速度。信息可以非常迅速地从计算机的主存中检索出来。计算机主存的缺点是它的容量有限，相对来说也比较昂贵，而且对于大多数计算机来说，当计算机关机时，主存中的信息要丢失。

现在，人们已经开发了辅助存储器来弥补主存的缺点，补充主存的性能。辅助存储器有比较大的容量，比较便宜，并且当计算机关机后也不会丢失信息。在大多数使用UNIX系统的中型计算机上，辅助存储设备（也称海量存储设备）通常是磁盘和磁带。磁盘和磁带用磁性来存储数据，就象盒式录音带那样。辅助存储器的缺点是：存取放在辅助存储器中的信息比存取主存中的信息所花费的时间要长得多。信息几乎总是要从辅助存储器中移到主存中去。

有许多试图解释计算机工作的比喻。我最喜欢的是用菜谱这种比喻。菜谱正好是烹调某些菜肴的一系列指令。当你按照这个菜谱烹调时，你所干的与计算机运行一个程序时所干的一样。正如在菜谱中有许多不同的烹饪法，在计算机中也有许多不同的程序。

我猜想计算机之所以被人看得如此神秘，是因为它们是用电工作的，并且以0和1两种模式存放信息。要是计算机能够读菜谱并且准备饭菜（这是一项艰巨得多的任务），那么人们对计算机就不会有如此神秘的印象了。

在一个磁盘或一个磁带上，信息的一个集合叫做文件，文件通常用名字标识。文件组织的方式是计算机系统的主要特性之一。磁盘和磁带通常装有许多文件（经常是数千个），对于一个计算机来说，能够迅速地定位一个指定的文件是很重要的，因此，计算

机保存了文件表和它们的位置的信息。为了指挥计算机访问这些文件，你必须懂得这些表的基本组织。这个非常重要的课题将在第六章讨论。

使用计算机终端，你可以和计算机通信。计算机终端有一个象电传打字机那样的键盘和一个输出设备。显示终端一般采用电视输出设备，而打印终端采用打印机输出设备。少数终端在一个设备中既有显示又有打印机。显示终端也叫视觉终端或CRT（阴极射线管）终端。声音输入还是多年以后的事，因此，目前要使用计算机，就应该熟悉键盘的布局。

计算机终端可以通过电话线或直接连线与计算机相连。直接连线更可取，因为它们传输起来更快，但是只有在计算机终端与计算机实际距离很近（在一英里左右）时才能工作。

调制解调器是这样一种设备，它们允许计算机和终端进行远距离通信。调制解调器把声音信号调制成电信号或者把电信号解调成声音信号，调制的电信号可以通过公用电话系统发送出去。

2.3 裸机

如果计算机不带有任何可帮助用户程序，那么这种计算机称为裸机。很早的计算机的确是裸机，它们都通过操纵一系列开关来工作。操作系统就是因为用开关来控制计算机太笨拙而发展起来的。

操纵复杂的开关是专家即计算机行家们的工作。为了普及计算机，把它们从行家的掌握之中摆脱开来，已经花费了许许多多的心血。虽然这种摆脱还没有完成（或许永远也不可能完成），但是已经取得了重大进展。操作系统是在开展使计算机更容易使用的工作以来的主要进展之一，操作系统使得计算机的使用变得更为容易，而它本身的发展又增加了计算机行家的新成员——操作系统专家。

实际上，今天所有的通用计算机系统都使用操作系统来避免

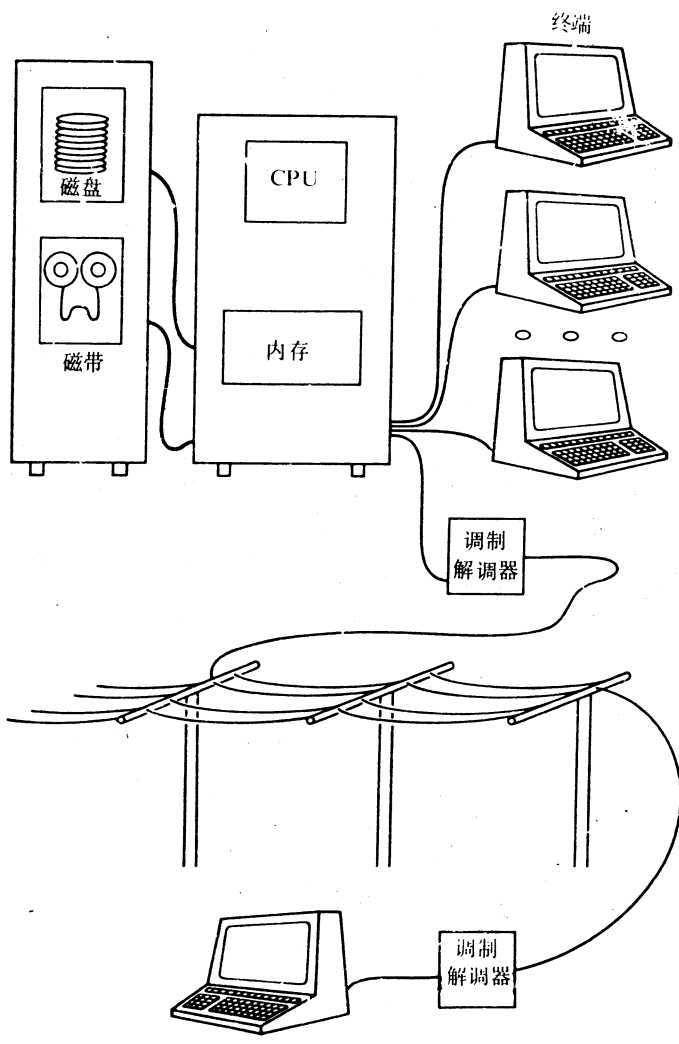


图2.1 一台典型的计算机

裸机带来的问题。操作系统的目标是增加通用计算机的有效性。而用于微波烤箱、缝纫机以及工业控制的那些专用计算机，就很少使用操作系统了。

2.4 操作系统

操作系统是管理计算机资源的程序。操作系统完成向通信设备发送信息、管理存储设备上的存储空间、把信息装入内存等工作。在允许若干人同时使用的计算机系统中，为了公平而且有效地分配计算机资源，操作系统裁决各种各样的请求。

要使用一个设计良好的操作系统一点也不神秘，你只要知道系统是如何组织的就行了。遗憾的是，由于某些早期的操作系统对于用户几乎都是妨碍多于帮助，所以有些人对操作系统的印象不好。

操作系统的复杂性通常随宿主计算机系统的复杂性而定。很简单的计算机系统常常只有很简单的操作系统。最简单的操作系统称为监控程序，它只有一些非常有限的功能。

具有中等程度的复杂性、价格及性能的计算机称为小型计算机。虽然现在已有许多用于很小的计算机（微型计算机）和非常大的计算机（大型计算机）的UNIX系统版本，但是原先设计的UNIX系统是在小型机上运行的。在一些小型机操作系统中，一次只能运行一个程序，而另一些小型机操作系统允许同时运行若干个程序。一次运行若干个程序的称为多道程序设计，实现起来很困难。因此，不包括多道程序设计的操作系统通常要比包括多道程序设计的操作系统简单得多。

UNIX系统是属中等复杂程度的操作系统。它远比运行在大型机上的那些操作系统简单，但是它比运行在微型机上的大多数操作系统具有更多功能。例如，UNIX系统允许你同时运行若干程序。

2.5 分时

分时是为了在若干用户之间共享一架计算机而发展起来的若干技术之一。象家用计算机那样，通常一次只运行一个程序的计算机不需要分时或任何其它的多道程序设计方法。分时的目的是为了给每一个用户这样一种假象：他们都在“独立”使用计算机。现代计算机所以能够分时是因为它们在一秒钟内可以完成数百万次操作。在这样一种速度下，计算机可以把成千上万次操作作用于你的任务，成千上万次操作作用于你的同伴的任务，成千上万次操作作用于协调计算机的各种任务——所有这些都在一秒钟内进行。

在某些计算机中，你一直要等到计算机已准备就绪才能运行你的程序。然而，在交互计算机系统中，计算机总是等你启动程序。当你请求计算机运行一个程序时，它就立刻开始运行，计算机勤奋地在这个程序上工作直到这个程序完成为止。该计算机可能并行地在干别的某些操作，但是，只要你一打入命令，通常你的程序就启动计算机继续运行。

分时是把每个时间单位划分成许多时间片来起作用的，每个执行程序享受一个时间片。当多个程序执行时，每个程序享受的时间片要比少量程序执行时的时间片小。由于计算机是非常快速的设备，所以它们可以迅速地从一作业转换到另一个作业，从而造成了这样一种现象：“计算机正在同时执行多个任务”。事实上，计算机在一个任务上进行，然后转换到下一个任务，再下一个，如此等等。

某些分时系统，在它们过载时，效率极低。当开销大到计算机把它的所有时间花费在程序之间的转换上而实际上没有什么时间运行程序时，效率就陡然下降。

2.6 核心

操作系统的某些功能每秒钟要被请求许多次。例如，在UNIX

系统中，从一个程序向另一个程序转换所涉及的那部分功能（分时）一秒钟要被请求许多次。在UNIX系统中所有立即需要的功能固定地放在内存中。操作系统的内存常驻部分称为核心。

许多操作系统的某些功能，比如把某些信息从一个海量存储设备中移到另一个海量存储设备中的功能，只是偶而用到。这种功能由实用程序提供，实用程序是鉴于用户需要而引用的标准程序。在UNIX系统中，用户编写新的和有用的程序并直接添加到实用程序库里是很容易做到的。

在许多操作系统中，核心包含许许多多功能。UNIX系统却试图用相对少的功能来武装核心，结果，操作系统的许多功能就由实用程序来提供。如果你对核心感到有兴趣，那么你应该读读第十九章。UNIX系统核心很简单，其原理完全可以为绝大多数用户掌握。

2.7 程序

程序是计算机为了获得某种结果而遵循的一个指令序列。当程序不执行时，这种指令序列就存放在一个海量存储设备（通常是磁盘）上。为了运行程序，程序指令的副本必须装入到内存中。

在UNIX系统中，一个程序正在运行，就称为是一个进程。如果有若干人差不多同时运行同样的程序，则存在若干进程，但却只有一个程序。

程序是很重要的，因为它是人和计算机之间的唯一界面。绝大多数程序需要从用户那儿得到信息，并且通常给用户某些信息作为回答。在一个设计良好的系统中，机器为用户工作所花的时间应该比用户为机器工作所花的时间多。

设计良好的程序工作起来很灵活。假如为了要把称为“alex”的文件名改为“alicia”而去专门编写一个程序，那是愚蠢的。这个程序可能用了一次后就废除了。相反却有一个程序，它能重新命名文件，当你运行这个程序时，只要提供两个名字就行了。

虽然设计良好的程序工作起来很灵活，但是所有程序都有一定局限性。有时，这种局限性似乎是很武断的。例如，你不能用来改变文件名的那个程序去改变系统中其它类型的名字（如用来和UNIX系统对话的注册名）。当你使用一个程序时，很重要的要知道，程序会从那儿需要什么样的信息，这个程序能做什么，不能做什么。

绝大多数UNIX系统程序仅仅完成一种功能。象使用计算机来编写并且发布一分备忘录那样复杂的操作就需要一系列程序。你的责任是把一个复杂的操作（比如编写一分备忘录）分解成相应于现有程序的一系列步骤。当你逐渐精通UNIX系统时，会认识到解决很复杂的操作常常有若干不同的途径。

程序可以笼统地划分成两类：实用程序和应用程序。实用程序通常完成通用的功能，而应用程序是为专门用途而设计的。例如，一个商店用来自动记帐的程序可能划归为应用程序，而显示时间的程序通常看成是一个实用程序。实用程序通常由操作系统支持，而应用程序常常要求单独编写。

本书的一个目的是让你熟悉最有用的UNIX系统实用程序。七、八、九三章介绍了UNIX系统中的大多数实用程序。对于完成简单的功能来说，这些程序大多数是简单、有效的工具。在这三章中出现的程序显示了其典型应用。列出这些程序的意思是让你熟悉这些程序，而不想对每一个程序作详尽的介绍。在读了本书的一般介绍之后，你可以通过随系统一起提供的资料掌握实用程序的细节。

2.8 shell和编辑程序

多数UNIX系统用户把大量时间花费在使用shell和编辑程序这两个程序上。为了灵活地使用UNIX系统，你需要许多其它程序如何工作的知识，但是你的很大一部分时间或许就是花在使用这两个程序上。

编辑程序是创建和修改正文文件的一种程序。标准的UNIX系统编辑程序是一种交互程序。用户通过打入命令来控制编辑程序。有打印文件行的编辑命令，有把正文加到一个文件上去的编辑命令，也有改变已经打入的正文的编辑命令。

你可以让编辑程序起各种各样的作用。例如，你可以用编辑程序建立一个文件，这个文件包含你用UNIX系统的mail程序发布的一个消息，或者你可能要用编辑程序建立一份资料或报告，这种资料或报告将在纸上打印并发布出去。

读了第五章你可以获得UNIX系统编辑程序的基本工作知识。有了基本了解，你也就学会了使用编辑程序，但是，如果打算透彻地使用编辑程序，那么你还应该熟悉在第十章中讨论的高级功能。这两章中讨论的编辑程序功能，在大多数UNIX系统的编辑程序中起作用。用在你的系统中的编辑程序，可能有若干特性没有在本书中提到，所以除了参看第五章和第十章外，你应该查一查你的系统所带的手册。

shell是UNIX系统中最重要的程序之一。象编辑程序一样，shell也是一种交互程序。用户通过打入shell解释（翻译）和执行的命令来控制shell。因此，shell的技术名称就是命令解释程序。

命令解释程序的功能是执行打入的命令。例如，如果要运行在显示屏上印出日期和时间的程序，那么就打入命令“date”，然后shell就让UNIX系统执行这个程序。

在许多系统中，命令解释程序是操作系统内部结构的一部分。然而，在UNIX系统中，shell仅仅是一种普通程序，类似于“date”程序或任何运行在UNIX系统中的其它程序。有关shell唯一特别的是：它是与UNIX系统绝大多数交互作用的中心。通常的用户，将把许多时间花在打命令上，shell却有许多可以用来增加有效性的功能。

UNIX系统实际上是信息管理的一种工具。UNIX系统的本

领归因于它有让程序一起工作来产生你所需要的信息的能力。在大多数计算机系统中，每一个程序自成一体。在UNIX系统中，绝大多数程序是简单工具，它可以和其它程序组合起来构成更强有力的工具。

shell是协调和组合UNIX系统程序的关键。有几章介绍shell的特性。第三章大约有一半是对shell作非常简单的介绍。第四章整章集中在shell上，把它作为交互命令解释程序。第四章中的信息初读起来显得很枯燥，但是在你对UNIX系统已经有了经验后，可以试试通过它进行工作。

除了作为交互命令解释程序之外，shell还是一种非常复杂的程序设计语言。绝大多数用户忽略了shell的程序设计特性，因为大多数用户不是程序员。然而，如果你要把shell作为一种程序设计语言来用，那么需要读第十三章并且通过第十四章中给出的例子来进行工作。

第三章 UNIX系统基础

学习一个新的计算机系统很象访问外国——一开始会感到很不习惯。即使UNIX系统已设计成了一个友好的有技术保证的操作系统，但是作为一个初学者，还是常会遭到失败。本章的目的是在你第一次与UNIX系统打交道时帮助你免遭挫折，并且帮助你加深对它的理解。

对有些人，需要读本章，本章告诉人们使用UNIX系统所需要知道的所有事情。也许一架打字机或烤箱简要地作个介绍就行了，但是一个计算机的操作系统却不可能一下子介绍清楚。为了有效地使用UNIX系统，你必须掌握大量的知识。然而，大多数概念是直截了当的，如果你勤快，那么，不用多久你就可以成为一个熟练的UNIX系统用户。

UNIX系统用起来比一个烤箱困难，但是它干的事也多。学习UNIX系统有点象学习一项复杂的技术，比如游泳或骑自行车一样，一开始都感觉对它不适应，但是最终都会掌握这门技术。

本章讨论使用UNIX系统的某些基本规则。如果你已经熟悉了UNIX系统，那么应该跳过这一章。

3.1 注册

使用UNIX系统要做的第一件事是注册。注册有两重目的：让UNIX系统验证你使用这个系统的权限，并且让UNIX系统为你设置环境。在允许通过电话访问的计算机系统中，对用户的使用权加以限制很有必要；而在需要对用户收费的系统中，知道谁在使用这台计算机以便收费，就可以精确地反映这个系统的使用情况。UNIX系统的功能之一是要管理计算机资源以便若干人

可以共享这台计算机。为了达到这一目的，UNIX 系统为每一个用户维持一个独立的环境。UNIX 系统记得每一个用户是谁，每个人什么时候注册，每个用户已经用了多少计算机时间，每个用户占用了多少文件，什么文件可以立即存取，正在使用什么型号的终端，等等。

在大多数单用户计算机系统（主要是家用计算机）中，不存在注册过程，因为实际地访问这个硬件就证实了你使用该系统的权力。在批处理操作系统中，不存在外表上的注册过程，相反，为了记帐和调度目的，每一个提交的作业都要加以标识。在UNIX 系统中，一旦你完成了注册过程，在每次运行一个程序时，就不再标识你自己了。

在你第一次注册之前，系统管理员或某些本地UNIX 系统雇主必须建立你的帐户。在不需要用户支付计算机机时费用的UNIX 系统装置中，设立一个帐户通常是很简单的。在需要支付计算机机时费用的装置中，设置一个帐户比较困难，因为需要编制有关帐单和财务信息。

从用户角度，设置一个帐户的主要问题在于注册名字。注册名是用户在与UNIX 交互作用时需要使用的名字。较短、小写的名字通常最容易。许多人使用他们姓名的第一个字母或者别名。名字“Betsy”，“kc”，和“m”都可以接受。

一旦你有了帐户，你就可以试试注册。如果你使用的是拨号终端，则你应该把全双工/半双工开关打到全双工位置，并且把终端的速度置成正确的速度。拨计算机号并且等待回答的嘟嘟声。当你听到了嘟嘟声时，应该把话筒放到音频调制解调器上，或者把“HOLD”合到多线电话上，或者把“ONLINE”打到数据电话上，究竟怎么办取决于你用的是什么样的电话硬件。从硬连线终端上进入就更容易了。为了获得新的“login:”消息，只要敲一下回车(return)键或者control-d 键就可以了。

一旦你已经建立了连接，计算机将会在终端上出现一些信息。

如果是

login:

或类似的什么消息，那么终端和计算机的通信速度是同步的，你就可以继续下去并且打入你的注册名。如果在你的屏幕上消息不是这样，那么在你的终端上敲一下break键。break导致UNIX系统改变它的通信速度，设法与你的终端同步。UNIX系统将印出一个新的消息。如果消息还不对，则再敲一下break键。为了和你的终端速度一致，UNIX系统通常循环地通过四、五种速度。如果试了四、五次后，你还是不能使UNIX系统在你的终端上印出“login:”字样，那么你就应该寻求值班专家的帮助。

最终，你会看到“login:”消息，此时打入你的注册名并且敲一下回车键。在一个简短的间歇之后，UNIX系统可能会要求你的口令。口令是为了证实你的身份而打入的一个保密字。这时你要打入你的口令并且敲一下回车键。

在与UNIX系统交互作用的绝大多数场合下，你会看到你所打入的每一个字符。然而，当你打入口令时，UNIX系统不让你打入的字符在屏幕上回应，通过这种方式来保证口令的安全性。UNIX系统接收了，但是不作回应。你必须非常小心地打入口令，因为在打入时你看不见它。如果打入了一个错误，就得从头开始注册过程。

一旦打入了口令，系统就检验它。如果口令被检查通过了，注册过程就可以继续下去。如果检查有错，那么会要求你再次打入注册名和口令。有些系统中不用口令。有些系统有额外的一层保密措施：在要求你打入注册名和个人的口令之前，得先打入一个拨号口令。

UNIX系统在注册过程结束处可能印出若干消息。这些消息可能是公布有关系统的新的调度消息，新的程序，用户会议，等等。在这些消息之后，UNIX系统将印出一个提示符，表明系统已经准备好接受你的命令。缺省的提示符通常是一个\$（在老的

系统上是一个%)。这时，你可以打入命令，并且可以和UNIX系统对话了。

3.2 某些简单的命令

学习UNIX系统的最好途径是使用它。

当打入UNIX系统命令

`date`

并且敲了回车键后。系统将显示日期和时间。跟在日期之后的下一行，系统将显示一个新的提示符，表明它已准备好接收下一条命令。在打入UNIX命令之前，你应该等待提示符。这本书中我们印出命令时没有印出UNIX系统提示符，这是因为提示符因系统而异，而且因为提示符并不是为运行命令而打入的部分。

由于在命令的结尾，总得打入回车或换行键，因此，从现在起我们将不再提它。然而，只要打入命令，就都得用回车或换行键来结束它。在本书中，只要我们印出一条命令，你就应该想到在这行的末尾有一个回车或换行符。我们不再在每一条印出的命令末尾使用笨拙的记号<CR>或<RETURN>了。

接着，打入命令

`who`

系统应该显示一组当前正在使用该系统的人、他们第一次注册的时间、以及他们正在使用的终端的标识名。要注意，你的注册名也在其中。打入命令

`echo hello`

系统应该显示消息“hello”。在这条命令中，字“echo”是命令名，而字“hello”是该命令的自变量。命令的自变量用来给命令提供额外的信息。echo命令只是重复它的自变量。echo命令的若干用法将会在本书后面章节中看到。在UNIX系统中，命令和它们的自变量（可能有几个自变量）由空格或制表符分开。空白格（空白或制表符）极其重要。如果你打入命令

```
echohello
```

那么你会得到一个出错消息。

在UNIX系统中，大小写也是极其重要的。UNIX系统懂得小写字母不同于大写字母。大多数UNIX系统命令的名字是用小写字母写的。如果你打入命令

```
Echo hello
```

那么，你也会得到一个出错消息，因为不存在一个名为“Echo”的命令。

date, who和echo是三个最普通的UNIX系统命令。一条命令就是要求UNIX系统去完成某件事的一次请求。UNIX系统通

```
login: kc
Password:
NOTICE —The system will be down
from 17:00 to 19:00 tomorrow
for routine maintenance.
Remember, monthly users meeting
this Weds at 5 pm.
% date
Weds July 5 11:08:17 EDT 1980
% who
td      tty10    Jul 5    7:03
kc      tty18    Jul 5    8:18
alvy    tty11    Jul 5    11:03
karl    tty03    Jul 5    11:03
% echo hello
hello
%
```

图3.1 在一次UNIX系统对话中开头的少量命令

在UNIX系统对话的这张图中，用户(kc)打入的字符下面加横线，而计算机回答的字符下面不加横线。要记住，在每一行的末尾有一个回车符。

常象上面所说的那样进行对话工作。在提示符后，你打入一条命令，后面跟一个回车键。然后，UNIX系统试图执行这条命令。当UNIX系统已经完成了这条命令的运行时，它就显示一个新的提示符。图3.1显示了UNIX系统对话开始的情况。

3.3 文件和目录

一个文件就是命了名的一组信息。正与UNIX系统交互作用时，你会用到各种各样的文件。术语“文件”用得非常之妙。计算机文件完全类似于存放在文件柜里的纸面文件。计算机文件有名字，有长度，可大可小，可以建立和撤消，也可以被检查。

在UNIX系统中，文件的重要性自不待言。每次运行一个程序就是在访问一个文件。这样，运行多个程序就是访问多个文件——通常是在命令行上提及的那些文件。

在一个UNIX系统中，可以有成千上万个文件，而一次只能看到其中的很少几个。在UNIX系统中，文件聚集成组，这个组称为目录。每一个目录有一个名字，例如，我的大多数文件的目录的全名（路径名）是‘/usa/kc’。在本书中，文件和目录名，象‘/usa/kc’那样，用单引号括起来。而所有其它引用的项用双引号括起来。第六章将详细讨论文件系统。

注册的一个目的是建立初始环境。环境的元素之一是当前目录名。当一开始注册时，系统把主目录作为当前目录。每一个用户通常有不同的主目录。刚刚建立的帐户，主目录下除了少量供管理用的文件外，可能没有别的。

`pwd`（打印工作目录）命令印出当前目录的名字。由于在不同的目录中可以得到不同的文件，因此，你应该知道当前目录的名字。打入命令

```
pwd
```

可了解当前目录的名字。在我的系统里，当注册以后马上运行`pwd`命令时，在我的终端上就会显示路径名‘/usa/kc’。在你的

系统里将会显示你本人的主目录名字。UNIX的文件系统组织和路径名(例如,‘usa/kc’)的使用将在下一章讨论。本章的剩下部分只要求你了解,文件是按组构成称为目录的单元的。

除了知道当前目录名以外,你还经常要知道什么文件在当前目录中。**ls(列表)命令**用来列出在一个目录中的文件。打入命令

```
ls
```

在当前目录中的一系列文件就印了出来。如果你的帐户刚刚建立,那么,只含有少量管理文件(在某些系统里没有)。

在大多数UNIX系统中,有一些标准目录。目录‘/bin’中通常包含许多你使用的程序。命令

```
ls /bin
```

将显示出在‘/bin’目录中的一组文件。只要你提供一个目录名作为ls命令的一个自变量,在这个目录中的所有文件就都列了出来。

UNIX系统允许你改变目录的前后关系,以致任何可访问的目录都可以变成当前目录。打入改变目录命令

```
cd /bin
```

就把目录‘/bin’作为当前目录(在老的系统里,改变目录命令常称为“chdir”)。接着可以打入命令

```
pwd
```

来验证新的当前目录。

只要在‘/bin’目录上,你就可以使用命令

```
ls
```

把‘/bin’里的所有文件列出来。回想一下,当你的当前目录不是‘/bin’时,为了获得同样的文件清单,使用ls命令时你得提供自变量‘/bin’。

现在你已经看到了。某些命令的操作要根据你的文件所在的目录而变化。许多人常常被极易改变的UNIX系统环境弄糊涂了。在一个目录中起作用的操作常常不能在另外的目录中起作用。你

应该始终清醒地了解当前目录的名字。一旦你理解了 UNIX 系统目录结构，你会明白这样做是有益的，而不是一种障碍。

3.4 与UNIX系统对话

用户以对话形式和UNIX系统打交道。一般情况是，用户打入一个命令，然后UNIX系统回答。对于简单命令，回答通常在一秒钟左右给出。复杂的命令花的时间要长得多。当UNIX系统严重过载时，即使简单的命令也要花费很长时间。因为对话是使UNIX系统有效工作的中心问题，所以下面我们稍微详细地讨论对话规则。

打入一个UNIX系统命令类似于在某些其它计算机系统上提交一个作业。当你打入一条UNIX系统命令时，是想要该计算机为你做某件事。至此，可以想象，“打入一条命令”就是意味着该计算机为你运行一个程序。当你对UNIX系统有更多的了解时，你会意识到，打入一条命令涉及的东西常常比仅仅运行一个程序更多，你也会开始理解UNIX系统环境怎样使其整个范围要比它的各部分之和还大。

对于一个初学者来说，你最好在一行上只打入一条命令。稍后的章节将向你显示如何在一行上打入若干条命令，或者一次运行若干命令。你可以通过打入一串字符，然后敲一下回车键或换行键来作为一行输入。回车（换行）表明一行输入结束，当你打入命令时，回车（换行）通常表明一个命令项的结束，并且告诉UNIX系统去执行该命令。

计算机和人之间的一个不同之处是：计算机要等你敲了回车键后它才真正正视你的话（命令）。与人谈话要容易些，因为当你与他们谈话时，他们会给你反馈信息。计算机和人之间的另一个差别是：计算机对于你打入的信息极为挑剔。在与人对话时，即使你的语法或发音有错，对方也理解你正在说的是什么。但是无声的计算机由于最简单的打字错误而生硬地停止对话。打入

命令时要小心，否则你要花费很多时间重打命令。

一旦你敲了回车键，UNIX 系统就马上对你已打入的信息感兴趣了。UNIX 系统立刻试图弄清楚你要做什么。命令的第一个字总是命令名，称为shell的程序试图解释这条特定命令。假定你要运行ls程序，但是由于错误，你打入了命令

```
lx
```

由于没有名为“lx”的命令，因此你将得到类似于

```
lx: not found
```

的出错消息，或者可能是

```
sh: lx: not found
```

在这两种情况下，名为“sh”(即shell)的程序告诉你，不能找到名为“lx”的命令。

如果命令找到了，那么它就进行工作，每一个特定的命令都用它自己的格式向你报告输入的错误。如果要列出在‘/bin’目录中的文件，但你错误地打入了命令

```
ls /bum
```

那么，ls命令会告诉你类似于

```
/bum: not found
```

的消息。当你在打入命令的自变量时犯了错误，命令就给你打印出一个出错消息。虽然已经作了不少努力力图使UNIX系统的出错消息尽量一致，但是，你肯定会遇到某些令人迷惑不解的消息。唯一可得到的安慰是，UNIX系统在标志错误输入时要比大多数别的系统好得多。

你打入的字符并不是直接从键盘送到屏幕上，也不是送去打印（如果你用的是打印终端的话）。相反，字符是先送到UNIX系统，然后再送回到你的屏幕上（或打印头上）。这种相当复杂的安排是为了使用灵活。在字符通过屏幕显示之前，UNIX系统获得每一个字符，所以它可以实行任何必要的转换。作为一个例子，UNIX系统可以把一个制表符翻译成适当数量的空格字符。在某

些时候，比如当你打入一个私有的口令时，UNIX 系统可能拒绝把字符返回给你。

在你打入输入行这段时间内，UNIX 系统正把它的大部分时间花在照应其它的事情上。然而，有两个特别的字符要立即照应，即抹字符和抹行符。抹字符一次抹去已经打入的一个字符，而抹行符将抹去打入的整个一行。

你可以规定，在你的终端上哪个键用作抹字符，哪个字符用作抹行符。在许多UNIX系统中，抹字符最初定为#，而抹行符最初定为商业中的@。使用#键和@键是因为它们在大多数终端上都有，并且很少作别的用处。如果你的终端上有更合适的键，那么可以重新指定抹字符和抹行符。在许多终端上，用 control-h 或 rubout 键来代替#键作为抹字符，并且常用 control-u 键来代替@键作为抹行符。一个控制字符（如 control-h 或 control-u）由压下CTRL键并且同时敲该特定的字符组成。

大多数系统中，在打入你的注册名和口令时，改正错误的唯一途径是使用#和@。这是因为，一直要到内部注册过程结束后，才有可能重新指定抹字符和抹行符。

这里有一个使用抹字符的例子（此例中，指定缺省的#号作为抹字符）：

```
qgi###wgo##ho
```

如果你遵照上面所示的字符序列工作，你将会看到用户实际上指定的 who 命令。

下面是使用抹行符的例子，它抹去一个打错了的输入行以便完全重打（在本例中指定用@符号作为抹行符）：

```
echohello@
```

```
echo hello
```

跟在抹行符之后的行，其外观表现随系统而变化。在某些系统中，如果你正在使用的是一个显示终端，那么抹行符抹除输入行。在另一些系统里，抹行符自动地把你推进到下一行（就如上一例子

所示的)。而在某些较老的系统里，抹行符逻辑上抹除该行，但是在屏幕上并没有任何反应。在一本书里要把这些情况一一罗列出来是很难的。

在你打入这一行时，你可以使用抹字符去抹除该行的一部分，而用抹行符去抹除整行并且从头开始。然而，一旦你敲入了回车键就不可以更改了。抹除前一行的字符是不可能的，所以在你敲回车键前要检查一下你打入的行是否正确。

许多命令执行一个功能，然后停止。例如，打印文件的程序将打印这个文件，然后就完成了这个命令。另外的命令将交互式地工作。交互命令的一个例子是UNIX正文编辑程序。该正文编辑程序交互式地从用户那儿接受命令，组成类似于我们在本章中介绍的UNIX系统命令对话那样的对话关系。

在你正在执行一个交互程序期间，只有这个程序的命令是直接有用的。当该交互程序终止时，你就返回到命令层，并且印出一个提示符，告诉你UNIX系统已准备好接受你的命令。你必须自始至终记住使用计算机的前后关系。一旦你熟悉了该系统，前后关系的转换将成为一种习惯性行为。当工作并不象你想象的那样展开时，你应该明确地想到所处情况的前后关系。也许你正打入编辑程序命令而你却处在UNIX系统命令方式中，也许你正打入UNIX系统命令而你却处在编辑程序中。

有时，有可能你要停止一个正在运行的程序。在UNIX系统中，敲一下中断字符就可以停止一个正在运行的程序。中断字符象抹字符和抹行符那样导致系统立即起作用。在大多数较新的系统中，中断字符是删除（在终端上通常标志为“DEL”）键，而在较老的系统中，中断字符是control-c。你也可以把你终端上的任何一个键指定为中断字符。

因为某些程序处在它们的临界区时是不允许中止的，所以UNIX系统允许程序阻塞中断。如果程序已经阻塞了中断，那么，中断键也就不起作用了。例如，正文编辑程序在它的绝大多数操

作中阻塞了中断，因为不希望由于不小心敲了中断键而丢失正在工作的文件（懂得计算机硬件内部工作的用户要小心，不要把硬件中断和UNIX系统中断功能混淆了）。

3.5 注销

注销比注册要容易得多。当你用完了UNIX系统时，应该注销。注销就是通知系统，你不打算向系统提出任何进一步的要求了。在一个以分钟记帐的系统中，一旦完成工作就要注销，这十分重要。在一个计算机时间比较“富余”的系统中，注销仅仅是出于对其它计算机用户的礼貌。

可以通过打入control-d注销。control-d是UNIX系统的文件结束符。打入文件结束符是告诉UNIX系统，不再有进一步的命令需要处理了。如果你用的是拨号终端，那么control-d并不挂起线路，系统将印出“login:”消息，表明它准备好接受一个新的注册。如果你是通过拨号线与系统连接的，那么注销的另一种方法是挂断电话线。

3.6 UNIX系统手册

对于正在使用UNIX系统的任何一个人，主要的参考材料之一是“UNIX程序员手册（UPM）”（某些版本用“UNIX用户手册”这个题目）。UPM包含了在你的系统上可以使用的大多数命令的有关信息。要核实一下你的手册是否适合你用的UNIX系统版本。

本书并不能代替UPM。UPM包含了有关你的系统的特殊信息，并且包含了许多难理解的命令的特殊信息以及并未在本书中讨论的特性。UPM的优点（也是其弱点）是，它太专门了。相比之下，本书试图介绍适合于所有UNIX系统的一般信息。本书另外一个作用是把普通和有用的程序与难懂的程序加以区分。而UPM对待所有的程序都是一律平等不加区分的。

UNIX 系统手册是由熟悉 UNIX 系统基本操作和服务的人设计的，也是为这些人设计的。如果你是一个初学者，你也许会发现，该手册中的大多数叙述太简练了，以至对你没有多大帮助。当你逐渐成为一个很熟悉的用户时，从另一个角度你又可能会发现，手册的简洁使它比一个繁琐的手册更有用。

大多数命令都在系统手册中作了适当的介绍。例如，在我的 UNIX 程序员手册中对于 pwd 命令的论述清楚地写成：

```
pwd prints the pathname of the working (current) directory
```

然而，对于一个新手来说，系统手册中许多很复杂的命令条目用处极少。某些很复杂的命令通常是在另外的资料，即在学术刊物上的论文中作介绍的，对于计算机科学家来说，这是些合适的描述，但是对于临时的用户来说，常常觉得不好懂。

UPM 分成八节。第一节介绍了在系统上可以得到的大多数命令。第二节到第八节介绍了程序员感兴趣的系统的各个方面。对于非程序员，第二节至第八节仅仅是些古董（第六节——游戏除外）。

UPM 的第一节包含了以字母顺序排列的 UNIX 系统命令的一组条文。对于在你的装置中可以得到的一般命令来说，第一节也应该是一组条文。对于你的装置是独有的命令，比如图形实验室中的图形命令，常常在局部发行的手册附录中介绍。某些 UNIX 系统是通过各种各样的来源获得程序而创建的。对于这些系统，手册可能不一定合适。

密切相关的命令有时合在单独一个条文中讨论。在我的 UPM 中，mv（移动）、cp（复制）和ln（联结）命令就都在一个条文中讨论了，因为它们都变动文件。如果你未能记住命令的确切名字，那么你应该试试在置换索引中查找所有相关的词。例如，为了揭示 cp 条文描述如何复制、联结或移动文件，你可以在置换索引中寻找词“move”。在 UPM 中的置换索引类似于上下文索引中的关键字，这种上下文索引由某些摘要编辑部门和学术刊物所采用。

UPM第一节中的每一条文,遵循一种相当统一的格式。条文的顶上给出名字和一种简洁的说明,后面跟随该命令的格式。对于ls命令来说,条文的顶部看起来象下面那样:

NAME

ls - list contents of directories

SYNOPSIS

ls [-ltasdriu] names

SYNOPSIS给你一种如何打入该命令的提示。ls的格式表明,打入词“ls”之后,后面可跟也可不跟一组任选项(字符串“ltasdriu”中的一个或多个字符),然后再跟一组文件或目录名。格式中的方括号表明:用方括号括起来的量是可以任选的。

命令的说明通常跟在格式后面。对于ls命令来说,其说明有一页多一些。一个命令的说明通常说明了该命令的基本操作以及你怎样可以通过使用各种任选项来改变这种基本操作。例如,ls命令具有这样的任选项:它们允许你在显示一组文件名的同时显示额外的信息。

在说明之后是若干简短的段:FILES段指定所有由该程序用到的文件,SEE ALSO段列出其条文中可能包含有用信息的相关命令,BUGS段可能包含某些有用的中止处理的原因,DIAGNOSTICS段可能帮助你辨认出错消息。这四段的任意一段或全部都可以省略。大多数命令的完整条文约为一页长。

UPM的第二节至第八节中的条文遵循同样的格式,这种格式是使用这些节的人都可以理解的,也就不在这里讨论了。要记住,如果你是一个新手,就只需用该手册的第一节。在第二节中有关系系统调用的那些说明,对你使用第一节中有类似名字的命令不会有什么帮助。

第四章 UNIX系统shell

计算机很擅长于执行高速运算，但是它们对于数值没有任何感觉。究竟是有用的工作还是空转，计算机不能加以区分。由于计算机没有任何主动权，因此人们必须精确地告诉它们应该怎样干。有时某人编写了一个程序，这个程序使计算机好象有了主动权，但实际上这个计算机是一个懒汉。

人们已经开发了命令语言，使得控制计算机来得更为容易些。命令语言中，有许多简便的命令来规定公共操作。命令精确地规定计算机应该执行什么程序。

计算机并不具有一种天生的本领来解释终端上打入的命令，这项功能是由操作系统提供的一种命令解释程序来完成的。在UNIX系统中，标准的命令解释程序称为shell。为了有效地使用UNIX系统。你必须知道如何打入命令。shell提供了丰富的功能；从而有可能规定非常强有力的命令。只要懂得一点shell，就能够使用UNIX系统。然而，如果你对shell有更深入的了解，那么对使用UNIX系统将更会有益。对于要更多地了解shell的用户来说，本书有好几章讨论shell程序设计和其高级特性。本章介绍的shell将对所有用户都有益处。

4.1 简单的shell命令

一个简单命令就是以空白或制表符隔开的一串(一个或多个)字。命令的第一个字指明命令名。后面的字指明命令自变量(回忆一下第三章可知，自变量是用来向命令传递额外的信息的)。最简单的命令就是单个字。ps(进程状态)命令将印出当前正在运行的一组程序。打入命令：

ps

然后观察结果。如果你没有干别的什么事，那么系统的输出将类似于在下面所示的我的系统上的输出：

PID	TTY	TIME	CMD
136	53	0:18	-sh
15390	53	0:04	ps

第一列总是进程标识号，第二列显示你的终端的标识号，第三列显示命令的累积执行时间，而最后一列显示命令名。上面显示的ps命令的输出展现了两个当前命令：-sh和ps。UNIX系统shell程序-sh是在注册过程的末尾就自动地启动了，而ps程序很明显是在你打入ps命令时，由你启动的。

你可以在一行上打入若干命令，命令之间用分号分开。当你预先知道将要运行的程序的顺序时，就可以使用这种特性。如果你想要知道当前目录的名字以及什么文件在当前目录中，那么你可以运行pwd程序及后面跟着的ls程序：

```
→pwd; ls
```

当你在一行上打入两个命令时，shell快速地接连运行这两个程序，先是pwd，然后是ls。该命令的等价形式是：

```
pwd; ls
```

因为你不必在shell专门字符（即命令名——译者注）前后放空格。

4.2 命令自变量

在第三章中，我们提到了命令的自变量，自变量用于向程序传递额外的信息。为了一个特定文件就编写一个完成某个功能（例如，在你的终端上显示一个文件）的程序，那是愚蠢的。应该写一些提供一般服务的程序。

在本章第一个例子中，ps命令不需要任何自变量，因为我们要使用ps的标准功能。你可以规定一个长列表的自变量来修改ps的操作。

对于每一个进程，长列表印出更多的信息。改变一个命令的操作所用的自变量常称为标志或任选项。在其它一些操作系统里，任选项称为开关或控制，你可以把任选自变量“-l”用在ps中来获得进程的长列表：

```
ps -l
```

命令名和自变量之间必须用一个或多个空格或制表符分隔开（如果你省略了分隔符，那么你将得到一个出错消息：“ps-l: not found”）。

你看到的仍是两个进程，但是对于每一个进程都印出了更多的信息。UNIX系统中，在任选自变量前面习惯上插入一个“-”（少数命令忽略这种字符）。

命令的自变量常常指明文件名。UNIX系统的cat程序(cat从单词concatenate派生来，含义是组合，串接)可以用来在你的终端上显示文件。你可以通过使用一个或若干个自变量来指明某个文件(cat的其它用途在8.2节中讨论)。cat最简单的使用是在终端上印出单个文件。命令：

```
cat /etc/motd
```

将在你的终端上印出日志文件的消息。

当你打入命令“cat /etc/motd”时，shell完成各种功能。该命令由“cat”和“/etc/motd”两部分组成，首先shell要确认存在一个名为“cat”的命令。然后，shell对于跟在命令名后的字作出一系列解释，解释（在4.8节详述）之后，该命令中的整个一组字被传递给程序，程序就开始执行。

命令的第一个字称为命令名。其后的字称为自变量(有人喜欢称命令名为第0个自变量，因为它处在编号为第1、2、3等等自变量的前头)。在命令中的字由空格或制表符分开。一个由引号括起来的字中可以包含空格或制表符。例如““the end””是由从“t”到“d”(其中包括一个空格)七个字符构成的单独一个字。引号在13.5节中讨论。

shell并不清楚特殊程序所需要的特定的自变量。在上述例子中，shell不用弄清楚字“/etc/motd”是否指的是一个文件。然而，cat程序却期望自变量指的正是一个文件，而且如果提供给cat的自变量不是文件，那么这就是一个错误。shell的责任是把自变量传递给cat程序，而cat程序的责任是弄清楚这些自变量是否合理。

4.3 后台进程

有时你可能需要运行一个花较长时间才能完成的程序。如果该程序的运行不需要用终端再输入信息，那么你可以不加监视地运行它。shell有一种特殊的功能，它可以让你启动一个程序，然后不加监视地让它运行，你可以继续打入别的shell命令。正在运行而不用监视的程序说成是在后台运行，而后来你打入的命令说成是在前台运行。后台进程和前台进程同时运行。平常，shell按顺序执行你的命令。然而，如果你在一条命令之后打了“&”，那么shell将在后台启动该程序，并且立即提示你打入另外的命令。

假定要花长时间运行的程序叫acctxx。因为估计acctxx程序运行要花很长时间，所以你可能愿意在它运行的同时做点别的事。在本例中，acctxx的实际功能并不重要。如果你打入了命令

```
acctxx &
```

那么shell将在后台启动acctxx程序运行，印出acctxx的进程标识号，然后立即返回来执行另外的命令。如果acctxx产生了送到终端的输出。那么这种迹象表明acctxx真正在运行。然而，如果acctxx把它的输出放到一个文件上，那么你可以打入下面命令来了解acctxx正在运行的某些迹象。当打入命令

```
ps
```

时，在终端上将得到你的一组进程。在我的系统中，输出如下：

PID	TTY	TIME	CMD
136	53	0:39	-sh
15388	53	0:14	acctxx
15390	53	0:04	ps

要注意，这里有三个程序在运行，而不是在前面例子中所显示的两个。这里acctxx程序和ps程序在同时运行。从ps命令的普通输出中，没有办法区分后台进程和前台进程（ps的长格式输出能让你推断出哪一个进程在前台，哪一个进程在后台）。

如果一个程序需要从终端输入，你就不应该在后台运行它，因为这个后台程序和shell将会争夺对于终端的访问。运行一个向终端发送长篇输出的后台进程虽然有可能，但是，这是很别扭的，因为这种长篇输出中夹杂着正常的前台输出消息。

这里就产生了一个非常有趣的问题。很明显，在你正打入一个命令的同时，shell是在前台运行。那么在你运行这条命令时，shell到哪儿去啦？在UNIX系统中，一个程序可以睡眠以等待一个确定的事件。当你打入一个正常的前台命令时，shell睡眠等待该命令的执行。当命令执行完成时，shell被唤醒并且提示你打入下一条命令。睡眠的shell和执行着的命令都是前台任务、但是，它们并不争夺终端，因为当命令执行时，shell正在睡眠（顺便提一下，睡眠、等待和唤醒都是UNIX系统的术语。UNIX系统中的术语通常十分直观）。

在后台启动acctxx，然后在前台运行ps命令的另一种途径是打入命令

```
acctxx & ps
```

在ps完成之后，shell将提醒你打入其它命令。acctxx将仍在后台运行。可以同时运行的后台进程个数通常有限制（常常是20）。

4.4 标准输出和标准输入

计算机终端是计算机和用户之间的基本通信设备。UNIX系统使得访问计算机终端来得非常容易，因为大多数实用程序在终端上产生输出，而且许多实用程序从终端上读输入。当一个程序在终端上打印了某些东西时，该程序通常在执行输出操作，把信息输出到所谓的标准输出上。当在终端上打入字符时，一个程序

通常正在从所谓的标准输入上读取打入的字符。标准输入和标准输出是UNIX系统中简化程序的约定(“通常”一词用在标准输入和标准输出的定义中,是因为有可能不用标准输入和标准输出来访问终端。然而,绝大部分程序用的是标准输入和标准输出)。图4.1 为标准输入/输出连接示意图。

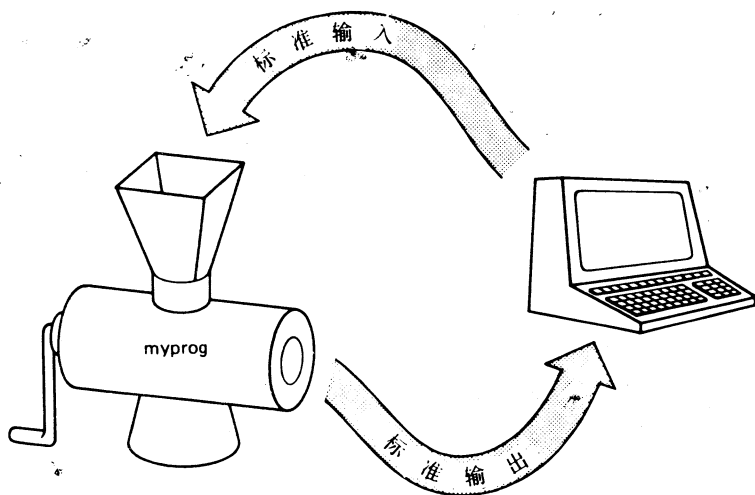


图4.1 标准 I/O 连接

程序的标准输入和标准输出通常指定给终端。

例如象ps, ls, who, date, pwd和echo那样的程序, 使用标准输出把它的信息发送给你。交互程序(例如, shell和编辑程序)从标准输入上读入命令, 并且把它们的答案写在标准输出上, 无论是UNIX系统还是别的系统, 标准化都是提高方便性和生产率的关键。

4.5 输出重新定向

标准输入和标准输出通常接到计算机终端上。然而, 由于标

准输入和标准输出是由 shell 建立的，因此有可能由 shell 重新指定标准输入和标准输出。shell 重新指定标准输入和标准输出的本领，是 UNIX 系统最重要的特性之一。

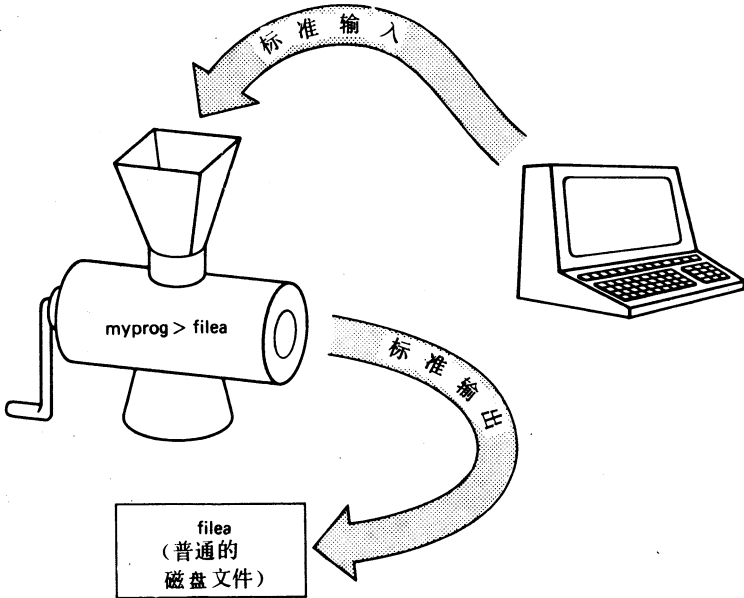


图4.2 输出重新定向

程序的输出可以重新指定给一个普通文件。在本例中，UNIX系统命令“myprog > filea”，使myprog命令的输出送到名为‘filea’的普通磁盘文件上。

让我们假定你要把ps命令的输出保存到一个文件中。如果你打入了命令

```
ps
```

那么进程状态程序将它的信息写到标准输出——你的终端上。

然而，如果你打入了命令

```
ps > posterity
```

那么结果有点不同，因为标准输出被重新定向了。进程状态程序仍然想把它的信息写到标准输出上，但是因为有了特殊记号“> posterity”，shell将把标准输出送到名叫‘posterity’的普通文件上。你将不会在屏幕上看到输出了。“>”是一个特殊的shell字符，它表明该命令的标准输出应该被定向到由命令下一个字表示的文件中去。这样的命令也可以打成：

```
ps> posterity
```

因为你不必在shell的特殊字符的前后加入空格或制表符。在上述两种情况下，文件‘posterity’都将包含通常应该在终端上印出的正文。你可以通过打入命令

```
cat posterity
```

加以验证。图4.2 为输出重新定向示意图。

通常输出的重新定向完全覆盖了那个输出文件。在上面给出的例子中，文件‘posterity’会被重写，它以前的内容都将丢失。有时，你想把输出定向加到那个文件的末尾。命令

```
ps>>ps.logfile
```

将把ps命令的输出附加到文件‘ps.logfile’的结尾处。你可以用一系列命令(ps>templ;cat ps.logfile templ>temp2;mv temp2 ps.logfile;rm templ)达到同样的结果，但是按上面所示的方式打命令要简单得多。

4.6 输入重新定向

标准输入也可以重新定向，至此，我们已经碰到的程序中，只有shell是从标准输入上读取信息的。而已经用过的其它程序(ls, who, ps和pwd)都是产生输出而不是从标准输入上读取信息的。shell通常从标准输入上读取命令，即shell读取终端上打入的命令。由于标准输入可以重新定向，因此，shell从普通文

件上获得它的命令也是可能的。图4.3 为输入重新定向示意图。

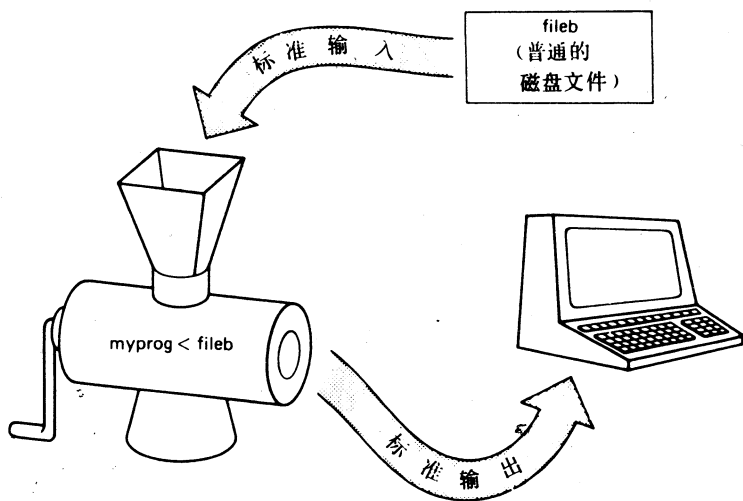


图4.3 输入重新定向

一个程序的输入可以由shell重新指定。在本例中,UNIX系统命令“`myprog < fileb`”使`myprog`命令从名为‘fileb’的文件中取得输入。

假定文件‘shellcommands’包含了以下三行:

```
ps
who
ls
```

(文件‘shellcommands’可以用正文编辑程序建立,参见第五章。)文件‘shellcommands’包含了三个熟悉的shell命令,它们可能是你从终端上打入的。如果你经常需要运行这三个命令,那末就不应每次都打入这些命令,相反,应把这些命令放进一个文件,并且让shell从该文件中读取这些命令,这可能更为容易。shell正是一个从它的标准输入中读取命令的普通程序。如果我们打入命令`sh`,那么我们将执行从标准输入中读取命令的shell的另

一种版本。然而，如果我们重新定向新的shell的输入，那么我们可以迫使它从文件而不是终端上读取命令。下面的命令能够建立具有从文件‘shellcommands’重新定向输入的shell的新版本：

```
sh < shellcommands
```

这个新shell的输出以及它运行的所有命令的输出都被定向到终端上。当shell到达文件‘shellcommands’的末尾时，它就终止并且把控制返回到你的交互式shell上（执行shell命令文件的其它方式在13.1节中讨论）。

当在系统上打入上述命令时，ps命令的输出、who命令的输出以及ls命令的输出就在终端上显示出来，然后，交互式shell提示打入别的命令。我们已经看到了who和ls的输出，所以，在这儿没有必要再作进一步讨论了。然而，ps命令的输出十分有趣。在我的系统上显示了下面的信息：

PID	TTY	TIME	CMD
136	53	0:39	-sh
14248	53	0:02	sh
14250	53	0:04	ps

让我们讨论一下由ps输出的每一个信息行。

1. ps输出的第一行详细列出了称为“-sh”的进程的信息。“sh”前面的“-”是一种暗示，表明shell是作为注册过程的一部分而被启动的。

2. ps输出的第二行详细列出了称为“sh”的进程的信息。这是为了执行来自文件‘shellcommands’中的命令而由我们启动的shell。

3. ps输出的第三行由后随名字“ps”的进程标识号组成。由于进程状态程序在它决定进程状态时运行，因此它是进程表中的一项。在我们的情况中，ps的父进程是我们开启的可见的shell，

而可见的shell的父进程是注册时初启的shell。

命令who和ls未在ps的输出中提及,因为它还没有开始执行。

上面提到的过程并不是shell处理存放在文件中的命令的唯一途径。然而,其它方法反映的是建立在shell中的特殊功能,而不是重新定向输入和输出的一般本领。这些另外的方法在13.1节中讨论。图4.4 是输入和输出重新定向的示意图。

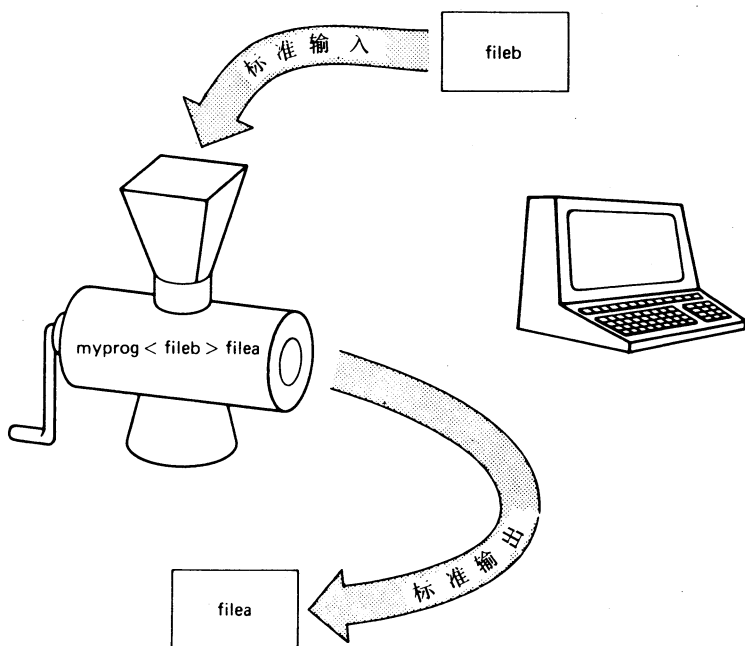


图4.4 输入和输出重新定向

输入和输出可以同时重新指定,就象在本例中那样,命令“myprog<fileb>filea”表示输入来自‘fileb’,而把输出赋给‘filea’。

4.7 管道

管道把一个程序的标准输出与另一个程序的标准输入连接起来。管道不同于 I/O 重新定向。输出重新定向是把一个程序的输出写到一个文件中，而输入重新定向导致一个文件包含一个程序的输入。另一方面，管道直接把一个程序的输出与另一个程序的输入相连接。

假定我们想知道在主目录（‘/usa/kc’）中有多少文件。也许最明显的方法是运行 `ls` 命令并且计算在终端上列出的文件的个数。对于一个只有很少一点文件的目录来说，这还可能凑合，但是对于一个塞满了文件的目录来说，这种简单方法就令人讨厌了。幸而，UNIX 系统包含了一个用于统计字数（和行数及字符数）称为 `wc` 的命令（见 8.5 节）。使用 `ls`，我们可以产生一组 ‘/usa/kc’ 目录中的文件；而使用 `wc`，我们可以统计在一张表中的字数。这样，我们就有了查找有多少文件在主目录中的基本工具了，但是我们怎样能够把这些工具组合起来一起工作呢？正如 UNIX 系统的绝大多数工作那样，这里至少有两种合理的方法，让我们来研究这两种方法。

第一种方法将使用前一节中的 I/O 重新定向技术，而第二种方法将使用称为管道线（pipeline）的一种特殊的 UNIX 系统设施。图 4.5 是管道线的示意图。

1. I/O 重新定向技术

方法：重新定向 `ls` 程序的输出以便把文件的清单保留在一个临时文件中。然后，使用 `wc` 统计在这个临时文件中的行数（-l 任选项让 `wc` 统计行数）。最后，删除这个临时文件。这些动作由下面三个命令完成。

```
ls /usa/kc > tempfile
wc -l tempfile
rm tempfile
```

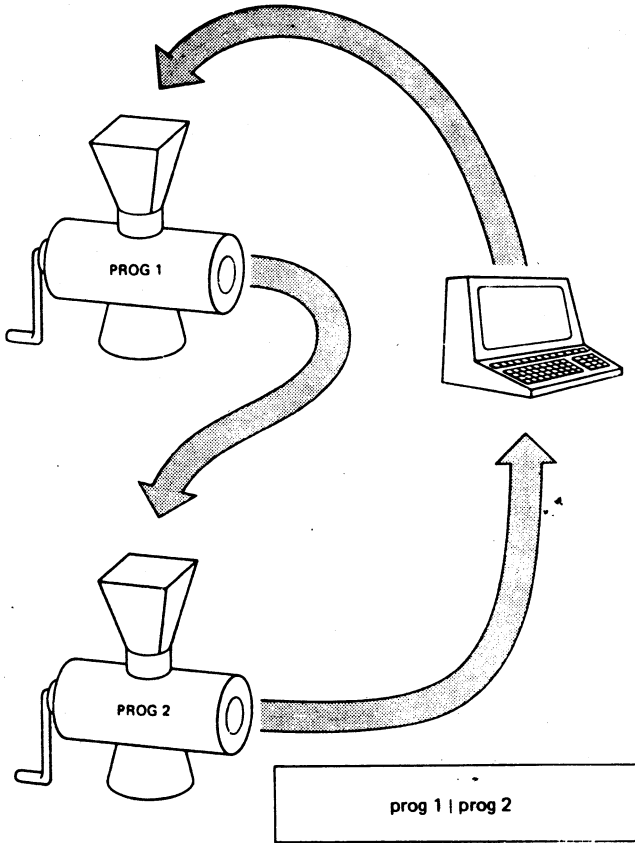


图4.5 管道线

管道线是一个命令的标准输出到另一个命令的标准输入的一种连接。在本例中，命令“prog1|prog2”导致命令prog1的输出作为命令prog2的输入而发送给prog2。虽然在本例中只看到两步（两个命令），但是实际上可以有步。

2 . 管道线

方法：把ls的输出连到wc。管道线连接的特殊shell记号是一个“|”或“^”。下面的命令把ls的输出与wc的输入连接起来：

```
ls /usa/kc | wc -l
```

正如你看到的。管道线方法更为简单。虽然 I/O 重新定向方法也可以达到目的，但是有某些令人讨厌的副作用。如果你没有得到在当前目录中建立文件的许可，那么，你就不能执行“ls /usa/kc>tempfile”命令，因为该命令建立并且写到‘tempfile’中（由于在ls程序可以运行之前，shell必须建立‘tempfile’，并且把标准输出重新定向到‘tempfile’，如果当前目录是‘/usa/kc’，那么‘tempfile’可能是一组文件中的一个文件。所以这组文件的数目可能太大）。

管道线是一个程序的标准输出和另一个程序的标准输入之间的连接。在上面的管道线例子中，ls程序在‘/usa/kc’目录中建立了一组文件，然后这组文件被传递给wc程序，wc程序统计在该组文件中的行数，并且把行数输出到标准输出即终端上。你所看到的全是最后计数值，而不是由ls产生的中间结果。

管道线使许多过程在概念上更为容易些。由于在一个管道线中的每一个程序可以连接作为一项任务的一个方面，因此，有可能书写连贯的、统一的程序。编写ls程序并能在它的输出上执行所有可以想象的操作是非常棘手的事。ls程序重点在列出文件，而wc程序重点在计数。在UNIX系统中，通过使用管道线接ls和wc，你可以建立一种新的动作，它统计在一个目录中的文件的数目。

（第八章中）许多正文文件实用程序可以随ls一起用在管道线中，目的是为了加强ls命令的本领。把所有这些本领都嵌入到ls程序中是不合理的，但是通过管道机构，所有这些本领都被嵌入到UNIX系统中了。你应该注意到，如果ls是管道线的一部分，那么它总是第一个元素。由于ls不需要来自标准输入的信息，因此，没有方法通过管理把信息发送给ls，图4.6 是 I/O 重新定向和管道线组合在一起的示意图。

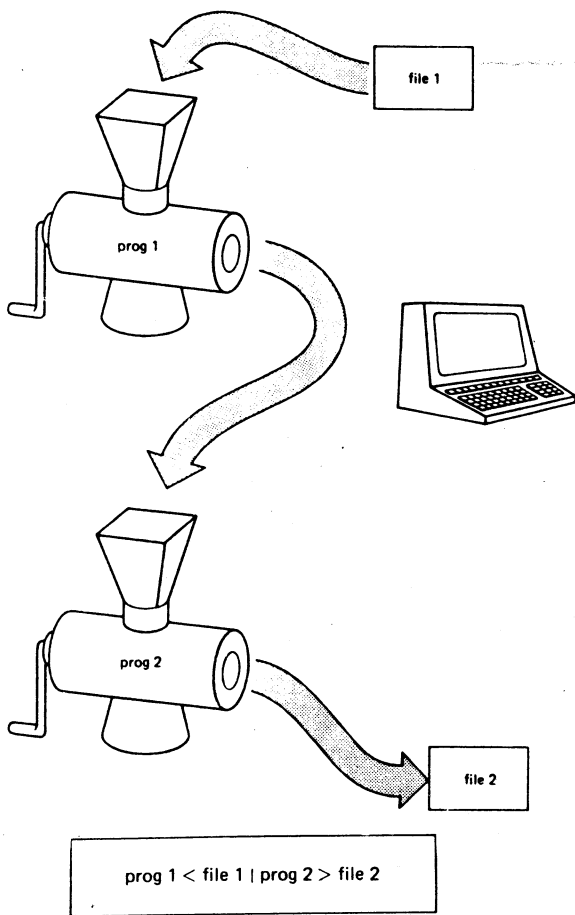


图4.6 I/O重新定向和管道线

有可能把输入输出重新定向与管道线组合在一起。在本例中，命令“`prog1 < file1 | prog2 > file2`”指定`prog1`的输入来自‘file1’，而`prog2`的输出赋给‘file2’并且把`prog1`的输出与`prog2`的输入连接起来。

`wc`是正文处理程序的一个典型例子。这种程序可以对作为自变量命名的文件进行处理（象前面例子中的命令“`wc - tempfile`”那样），或者对通过标准输入提供的正文进行处理（象前面例子中的命令“`ls | wc -l`”那样）。这种灵活性是UNIX系

统最强有力的特性之一。

4.8 元字符和文件名生成

提供给程序的大多数命令行自变量是文件名。命名文件时，一般要使用有关的文件有相应的名字。例如，所有C语言程序的名字以后缀“.c”结尾。本书的各章可以存在一系列名为‘chap1’、‘chap2’等的文件中。如果你想对一个目录中的所有C语言文件执行某种操作，那么，打入所有名字来作为命令行的自变量那是非常乏味的。为此，UNIX系统允许你同时规定一组文件名。当你打入命令自变量时，shell将检查你的自变量以确定你是否用了文件名生成缩写。可以通过规定文件名的一种模式来控制文件名生成。shell把你给的模式与当前目录的所有文件名进行比较。如果有什么文件名与这种模式匹配，那么一组按字母顺序排序的匹配文件的名称就被传送给该程序。如果在该目录中没有一个文件名与模式匹配，那么这个未作改变的模式就直接传送给该程序。

模式由普通字符和元字符组成。普通字符代表它们自己，而元字符具有特殊含义。完全由普通字符组成的模式（例如，“myfile”）不引用文件名生成。

应该了解文件名生成过程，因为每次打入一条命令都有文件名生成过程。至少应该确切地知道文件名生成的实质，并且要知道不应该在文件名中使用元字符“*”，“？”和方括号。进一步，你应该掌握元字符“*”和“？”，因为它们的使用的确十分简单，还应该知道怎样废除这些元字符的特殊含义。如果你真正要作为一个主人来使用UNIX系统，那么，要弄清楚字符类。

大多数UNIX系统资料用短语“正则表达式”表示模式。我喜欢术语“模式”。因为它对含糊的文件名生成的题目提供一种良好的直观认识。在某些别的操作系统中。用“通配符”来代表控制文件名生成进程的特殊字符。

下面的元字符在大多数UNIX系统上用来控制文件名生成。

- * 匹配任意一个字符串
- ? 匹配任意单一字符
- [引入一个字符组
-] 终止一个字符组
- 表明一种字符范围

*号和?号元字符使用起来很容易。*号将匹配任意一串字符,包括空串。这样,模式“*.c”将匹配文件名‘.c’或‘a.c’或‘aaaaaaaa.c’,但不能匹配文件名‘a.ca’。

?号匹配任意单独一个字符。因此,模式“???.c”将匹配文件名‘ab.c’或‘77.c’,但是不能匹配‘a.c’或‘abc.c’或‘bc.cc’。

方括号和“-”用于为字符组构成模式。在字符组中的字符由方括号括起来。模式“abc[aeiou]”将匹配任何以字符串“abc”开始而以单独一个元音字母结尾的文件名。“-”可以用在一对方括号内部,以便表明字符的范围。模式“def[0-9]”将匹配任何其前面三个字符是“def”第四个也就是最后一个字符是一个数字的文件名。范围是内含的(如上例中0和9都包括在其中),并且由ASCII字符集的数字顺序来定义。

当“-”用在方括号外面时,就失去了它的元字符的作用。而*号和?号,当它们用在方括号内部时,就失去了作为元字符的能力。在模式“-[*?]abc”中,只有方括号是起作用的元字符,其它字符只作一般字符用。这样,模式“-[*?]abc”仅仅匹配两个文件名:‘-*abc’和‘-?abc’。

shell的功能是一种宝贵的财富。你可使用它的特殊功能,比如元字符、I/O重新定向、管道线和后台执行等。然而,有些操作,需要一种用于控制这些功能的特殊shell字符。例如,你想使某一个特殊的shell字符失去它的能力,那么你可以在它前面加一个反斜线\字符。因此,命令

```
ls \*
```

将输出当前目录中其名字以一个星号结尾的所有文件名。

取消特殊的shell字符的能力的另一种途径是用引号。模式“answers\?”匹配‘answers?’。但不与‘answersl’匹配。当然，模式“answers?”既与‘answers?’匹配，也与‘answersl’匹配。如果‘answersl’和‘answers?’都是在当前目录里的正文文件，那么，命令

```
ls answers?
```

将把这两个文件都列出来，而命令

```
ls "answers?"
```

将只列出文件‘answers?’。你亦可以使用命令

```
ls answers\?
```

只列出文件‘answers?’。当需要转义的只是一个特别字符时。用反斜线进行转义比较容易；但是，当有若干个字符需要转义时，则用引号比较容易。在13.5节中会更加详细地讨论引号。

你应该理解在shell文件名生成过程中不只是元字符在起作用。当shell把模式与文件名比较时，在文件名前面的点必须显式匹配。如果你有一个名叫‘.invisible’的文件，那么，模式“*visibl”将不能匹配它。模式“.*visible”将匹配文件名‘.invisible’。模式“.*”将匹配所有以点打头的文件名。

在文件名生成中，使许多人感到迷惑不解的另一个问题是：在路径名中的斜线字符必须显式匹配。路径名是从目录到导至一个文件目录的一条路。较早提到的名字‘/etc/motd’是一个简单的路径名，它从‘/etc’目录导至文件‘motd’。在6.4节中将对路径名作详细讨论。

元字符只用于在一个目录内生成文件名。模式“/etc*.c”不与在‘/etc’目录内带有“.c”后缀的文件匹配。然而，模式“/etc/*.c”将匹配这些文件。在一个路径名中，斜线字符要显式匹配，这一限制基本上是合理的，因为当前目录是缺省的环境，除非你明确地另外加以说明。

假定当前目录包含下列文件：

```
ch1  ch2  ch3  ch4
33.doc abc ab.c ch3.a
```

命令

```
ls ch*
```

将列出下列文件：‘ch1’，‘ch2’，‘ch3’，‘ch3.a’和‘ch4’。要注意，此表是按字母顺序排列的，因为文件名生成就是按字母顺序进行的。

命令

```
ls *3*
```

将列出下列文件：‘33.doc’，‘ch3’和‘ch3.a’。要注意，星号*可以表示一串零个或多个字符。

命令

```
ls ch?
```

将列出下列文件：‘ch1’，‘ch2’，‘ch3’和‘ch4’，由于一个问号？只匹配一个字符，所以‘ch3.a’就列不出来了。

命令

```
ls ch[2-9]
```

将列出下列文件：‘ch2’，‘ch3’和‘ch4’。文件‘ch1’不列出了，因为“1”不在范围“2-9”之间。‘ch3.a’也列不出了，因为它有一个尾部“.a”。

下面一种情况是要掌握的，命令

```
ls ch[0-15]
```

将列出文件‘ch1’。虽然这个字符类似乎包括了数字0到15，但是实际上只包含了三个字符：“0”，“1”和“5”。应该记住，一个字符类是由一组字符组成的。序列“15”看起来象数15，但是在字符类中，它是两个字符。

4.9 小结

要有效地使用UNIX系统，就要理解shell的基本操作，应该知道怎样运行一个正常的前台进程以及怎样运行一个后台进程。也应该理解命令行自变量的思想，理解为了找到命令shell所应遵循的检索过程。I/O重新定向和管道线对于每一个人都是基本的——为什么使用UNIX系统而不利用它最强的特性呢？至少每一个人都应该了解文件名生成，如果你想从UNIX系统中获得更多的东西，就要学会使用文件名生成。

这几个方面还只是shell功能中的很小一部分。然而，对于把shell用作一种交互命令解释程序使用的人来说，在本章中介绍的这些内容就足以有效地使用UNIX系统了。shell是UNIX系统中最令人难忘的特性之一，因此，对于想知道得更多一些的用户来说，还要学习论述shell的另外几章。

第五章 UNIX系统编辑程序

为了完成任何有用的事情，计算机需要信息。一般说来，你（或某人）打入信息，然后，计算机完成某些事情，并且把信息传递给你。第一步，即信息进入，通常是最使人讨厌的。

计算机获取信息的来源有许多。有时，一台计算机从另外的计算机上获取信息。发送信息是没有什么问题的，但是决不会出现新的信息。在许多应用中，信息是自动收集的，就象现金出纳机和银行事务机器在收支资金时那样，前者记住每笔销售的每一项，而后者始终监视银行交易。在实验室中的计算机常常直接与实验仪器相连，当实验进行时，这些计算机自动地获得它们的信息。

当你不能用别的手段使信息自动进入计算机时，就得用手打入。把信息存放到计算机中的一种最容易和最普通的方法是：把信息放到一个正文文件中。正文编辑程序是这样一种程序：你可以用它把信息打入正文文件并且考查正文文件中的信息。正文编辑极为重要，因为对于大多数UNIX系统用户来说，这是信息进入的主要方法。如果你为了一个特别的工作而使用UNIX系统，那么你可以不用正文编辑，但是，对于大多数用户来说，正文编辑是主要的输入手段。

我们可以设想正文编辑程序就象一架电传打字机那样动作，性能一点不多也一点不少。让我们用名字“ugh”称呼我们假想的正文编辑程序。当你打入时，ugh接收正文并且把它放到一个文件中。ugh不能修正以前的正文，也不允许你移动正文、删除正文或者从其它资料中接收正文。ugh的主要优点是简单，但它不包含你必须掌握的大量命令。

实际上，所有的正文编辑程序做的工作远比我们假想的ugh

正文编辑程序所做的多。然而，正文编辑程序具备的性能越多，也就越难使用。大多数UNIX系统正文编辑程序允许你移动正文、添加正文、删除正文并且从其它正文文件中获得正文。你可以在一个位置、或一段文件、或整个文件中用另一个词来替换某个词（或一个词的一部分）。你也可以浏览一个正文文件，印出该文件某段中的所有行，或者印出含有一个特定字的所有行，等等。然而，为了使用强有力的正文编辑程序，你必须花一些时间来理解编辑程序。

正文编辑程序是一种交互程序。你可以通过打入命令来引入编辑程序。编辑命令允许你把正文加到文件中或者改变已经打入的正文。就象shell对话那样，编辑程序对话由你的命令后随编辑程序的回答组成。

遗憾地是，还没有标准的UNIX系统编辑程序（在这整个一章中，“编辑程序”指的就是编辑正文的一个程序）。我使用过的每一个不同的UNIX系统都有独特的一组正文编辑程序（它们确实不同）。大多数UNIX系统有若干正文编辑程序可用。对本书来说，我们把编辑程序分成两类：一类是与UNIX系统编辑程序相关的或类似的，另一类是与别的正文编辑程序版本相似的。本章介绍UNIX系统编辑程序以及类似的程序，其它编辑程序在这里略而

不说。

之所以有各种编辑程序的一个原因是有许多大学已经实现了对UNIX系统（包括对编辑程序）的修改。由这些编辑程序所提供的主要改善是屏幕编辑，它将在第十章结尾处讨论。

本章介绍为了使用正文编辑程序你应该掌握的基本命令。如果你打算做大量的正文编辑工作，那么学会使用你的正文编辑程序的所有特性一定是很有价值的。第十章介绍某些高级的、应用于很多编辑程序的特性，而系统的参考手册介绍了编辑程序中所有独有的特性。最艰难的部分是学会这些基本命令。只要理解了少量几个命令，其余的就都容易了。

本章的前面几节引入为了使用正文编辑程序而需要的某些一般概念。如果你以前用过正文编辑程序，那么可跳到5.3节去。

5.1 正文文件

一个字节是一个二进制数，它可以包含 256 种不同值中的某一个。在UNIX 系统中，一个文件是一个字节序列。当在一个文件的字节中包含所有 256 种可能的值时，我们说该文件是一个二进制文件。ASCII（美国信息交换标准码）是一个用大约 100 个不同的值编写正文的标准方法。其字节仅包含ASCII 码的文件称为正文文件。

正文用于存储许多类信息。下列三种类型包括了正文文件的大多数普通用法。

1. 资料

诸如信件、手册、手稿和书那样的资料都可以存放在普通正文文件中。让计算机辅助作资料准备是有好处的，因为正文只需打入一次；以后不必重新打入整个资料就可以在资料中作修改。此外，计算机可以用来检查拼写和语法，用于生成索引以及分类版本和修订编目等。例如，文件 ‘letter-to-john’ 就是存放在一个正文文件中的资料。

Dear John,

Feel free to come over
and use the system any
time after 5 pm next week.

Susan

2. 程序源代码

计算机程序是用语言编写的，语言被设计成让人们表达逻辑

的解法。程序由包含程序指令的正文文件所组成。文件‘me.c’包含了由C语言源代码指令构成的程序，该程序印出消息：

“I am a C-language Program”。

```
main ()
{ Printf("I am a C-language Program\n");}
```

3. 正文数据

简单的一组数据经常存放在普通的正文文件中。这儿是一个名为‘groceries’的正文文件，它包含了一组简短的杂货清单

```
yams
eggs (half dozen)
half pound of gruyere
Alpo
```

杂货清单存放在一个正文文件中，因为它只是一组杂货的简短的清单，把它们存入计算机的一个最简单方法就是放到一个正文文件中。用于美国军队的日用杂货清单，因为它太大并且太复杂了，大概不能放在一个正文文件中。

举正文文件的这组用途为例，仅仅是想给你一个可以在一个正文文件中存放各类东西这么一种感受。当然，货单可以是描述现代美国的饮食习惯的资料的一部分，而C语言程序可以是（或者是）论述UNIX系统的一本书的一部分。资料、程序源代码以及一组数据之间没有严格的区别。

有许多场合，信息并不存放在一个正文文件中。正文文件要浪费空间，这是因为在文件中每个字节的256种的可能值中，只用了大约100种可能值。在你很了解信息格式的场所（比如，支票处理系统、会计系统、飞机票预订系统、或数据库检索系统）中，你常常可以使用更有效的编码技术。当信息必须压缩存放或快速检索时，它们通常放在一个二进制文件中。

5.2 行编辑

一个正文文件可分成称为行的若干单位。正文文件中的行完全对应你直观理解的行。为了简单起见，一个正文文件的一行就是出现在计算机终端上一行的正文文件量。偶而，正文文件中包含有很长的行以致不适合你的终端。某些终端不能显示过分长的行。而有的终端可以把行折断并且在屏幕的下一行显示出来。虽然对行长没有严格的规定，但是一般不会长于256个字符。

大多数UNIX系统正文编辑程序称为“行编辑”，因为它们是在正文的行上操作。如果用纸和铅笔编辑某个正文，那么你总是把注意力集中在该正文的某个特定部分。用计算机编辑正文用的是同样的道理。当你编辑一个正文文件时，总是有一个“当前行”。当前行是正文中通常受编辑命令所影响的一部分。

在一个行编辑程序中，可以改变文件的单个字符，但是只能通过先确定该文件中包含该字符的行，然后再说明如何改变来进行。虽然这看起来象是事倍功半，但是你会明白，利用嵌入到行编辑中的这种限制进行工作是很容易也是很自然的。

其它类型的交互编辑是字符编辑和屏幕编辑。在字符编辑程序中，你可在文件中一个字符一个字符地移动字符指针。所有的改变都是相对于字符指针的当前位置来进行的。字符编辑程序是非常强有力的工具，但是常常难以使用。TECO（正文编辑和控制程序）大概是最有名的字符编辑程序。

屏幕编辑程序只能用在视频显示终端上。屏幕编辑程序提供进入正文文件的窗口。光标是在视频终端上的一个（常常是闪烁的）下线（ ）或小方框，它们用于指明当前字符位置。屏幕编辑程序使用简单，因为视频终端的光标可用很简单的光标定位命令在屏幕上定位，然后可以在该屏幕上相对于光标的位置作出修改。

屏幕编辑程序需要计算机与终端之间的大量通信。一个简单

的修改可以导致整个屏幕被更新。如果简单的改变就引起重画屏幕的长时间等待，那么编辑程序用起来就不会使人愉快。对于通过电话线与计算机连接的终端，屏幕编辑程序就不怎么有用，因为通过电话线的数据传送速率较低。然而，当终端通过高速宽带（高数据传送速率）系统直接与计算机连接时，屏幕编辑就很好。

除了需要通信带宽外，屏幕编辑程序要耗费计算机大量的资源。当编辑修改引起了复杂的显示修改时，处理机必须做大量的工作。许多UNIX系统装置不鼓励甚至禁止使用屏幕编辑，因为它对系统提出了很高的要求。然而，由于屏幕编辑程序对于操作人员的要求极小而抵消了对于计算机的很高要求。屏幕编辑程序因为它使用起来很容易而使人十分欣慰。

5.3 启动编辑程序

为了启动UNIX系统正文编辑程序，你必须注册到UNIX系统中。打入编辑程序名，后面跟上打算编辑的文件名，就启动了编辑程序。例如，shell命令

```
ed firstsession
```

把shell引导到名为ed的正文编辑程序，然后来编辑名为‘firstsession’的文件。如果你还从来没有用过文件‘firstsession’，那么，编辑程序将建立这个文件。只要你打入这个命令，该文件的内容就立即可以编辑。该文件的最后行将成为当前行，而编辑程序准备接收你的命令。

当编辑程序运行时，你只能打入编辑程序命令，在你使用编辑程序期间，不能直接使用shell命令。最后的编辑命令（除非你挂断电话或者计算机瘫痪了）将是停止该编辑程序，并且返回到shell。当编辑程序运行完成时，shell将提示你打入下一个shell命令。你必须记住，当编辑程序正在运行时，要打入编辑程序命令，而当你与shell对话时，要打入shell命令。

经过少量实践后，你就能够顺利地由shell命令方式转换到编

辑程序方式。改变工作方式是自然的事，我们大多数人不是在家里是一种方式，工作时是一种方式，休假时又是一种方式吗？

在你使用编辑程序期间，你使用的是此文件的副本，而不是文件的本身。如果你犯了一个重大的错误，原先的文件仍然可用。然而，正因为是在文件的副本上工作，因此你必须记住，在编辑工作结束时，要更新原先的文件。如果你忘了更新原来的文件，那么当你试图从编辑程序退出时，编辑程序会提醒你。

编辑程序有两种操作方式：命令方式和正文进入方式。在命令方式中，编辑程序等待你打入命令。在命令方式下，你打入的任何东西都被解释为编辑程序命令。编辑程序命令可以修改行、打印行、读或写磁盘文件、或引入正文进入方式。在正文进入方式中，编辑程序等待你打入正文行。在正文进入方式下，你打入的任何东西都将被加入到文件中去。在一行上，只打入一个句点，就可以离开正文进入方式，并且重新回到命令方式。你自始至终应该非常小心地记住你所处的方式。对于初学者来说，使用编辑程序最通常存在的问题是：在正文进入方式时打入了命令或者是在命令方式时打入了正文。

5.4 基本的编辑命令

本章下面若干小节介绍基本的编辑命令（参见图5.1），精通这些命令将使你能够使用编辑程序；打算认认真真使用编辑程序的用户应该懂得在第十章“高级编辑”中的命令。另外应该考查一下你的装置所带的编辑程序的随机资料，以便找出它独有的全部命令。

所有的编辑命令都是通过打入单个助记字符来引用的。大多数命令涉及文件中的一个特殊行或某个范围内的若干行。例如，“p”命令可以印出文件中的一行或多行。如果你打入命令

20 p

将印出该文件的第20行。使用命令

20,30 P

能够印出第20到30行的内容。

句点是表示当前行的一个特殊的编辑程序符号。如果你打入了命令

.p

那么就会印出当前行。如果打入命令

p

也会出现同样结果，因为对于打印命令来说，缺省行就是当前行。

不用地址的命令

q 退出编辑程序。工作文件不会自动地写到永久的磁盘文件中。

w 把工作文件写到永久的磁盘文件中。

使用一个地址的命令^①

.a 进入正文进入方式；新的正文被放在指定行的后面。只要你在进入方式，那么你打入的所有东西都会被加到工作文件中去。在单独一行上只打一个句点就可以离开进入方式。

.i 插入命令，导致你进入正文进入方式；添加的正文放在当前行之前。

\$ = 印出指定行的行号。

使用两个地址的命令^②

.,.c 该修改命令删除指定的行，然后编辑程序处于正文进入方式；进入的正文放在文件中被删除的正文的位置。

.,.d 指定的行从工作文件中删去。

.,.l 列表命令，毫不含混地印出指定行。所有控制字符用转义码印出。

.,.ma 移动命令，把指定行移到由 a 指定的后面。

.,.ta 传送命令，把指定行的一个副本放到由 a 指定的行

后面。

.,. p 印出指定的行。

.,. s/regexp/replacement/

本替换命令用来把指定行上的replacement正文替换匹配regexp的正文模式。

1, \$w 指定行被写到永久文件中去。

①缺省的行地址表示：1是正文缓冲区中的第1行，“.”是当前行，而“\$”是最后一行。

②缺省的行地址如①所示。

图5.1 基本的编辑命令

5.5 把正文加到工作文件中

附加命令(“a”)用于把编辑程序从命令方式改成正文进入方式。在命令方式时，你打入的每一行都假定是一条编辑程序命令。当你打入了附加命令时，假定就发生了改变：每一行都假定是加入到工作文件中去的正文。命令

a

将把编辑程序置成正文进入方式。所有随后打入的行都将被加入到正文文件中，直到打入了包含单独一个圆点的行：

为止。附加命令不太容易掌握，因为它导致编辑程序改变方式，使你打入的每样东西都突然作了不同的解释。这里通常犯的一种错误是在进入方式打入了编辑程序命令或者在命令方式打入了正文。如果发生的情况看来很可笑，那么就应该试试在一行单独打入一个圆点。此时如果你在输入方式，那么圆点将把你返回到命令方式；如果你已经在命令方式，那么圆点将只是印出当前行。

下面的例子展示了正在第10行后加入若干行正文。

10a

The UNIX System is a general purpose operating system for small to large computers.

The UNIX System is renowned for its simple construction, portability, and powerful command language.

5.6 印出文件行

“p”命令用于在终端上印出工作文件中的行。这个印出命令可以用于印出一行或若干行。例如，编辑命令

```
p
```

```
lp
```

```
l,$p
```

将分别印出文件的当前行，文件的第 1 行，以及该文件的所有行。

列表命令(“l”)是印出命令的变种。列表命令用于显示所有存放在该行中的特殊字符。所谓特殊字符，意思是在你印出该行时不能直接可见的那种字符(例如，制表符或退格符)。考虑包含下列八个字符的一行正文：

```
a, b, c<退格>, d, <制表>, e, f
```

如果你用印出命令显示这行，它看起来象下面那样：

```
a b d e f
```

c 不可见，因为跟在它后面的退格符把它抹去了，而制表符被扩充成一定数量的空格。然而，如果你用列表命令显示该行，那么退格和制表符就都可见了：

```
abc\bd\tef
```

约定“\b”和“\t”表示一个退格符和一个制表符。

5.7 更新原先的文件

只要你对文件作了实质性的修改或添加，就应该用写命令来更新文件的永久副本。要记住，当你编辑时，只是在文件的临时副本上工作。为了保留你所作的修改，就必须更新永久的副本。

写命令用于把一行（或多行）写到一个磁盘文件中去。写命令的普通形式是

```
n1, n2 w filename
```

这里“n1”和“n2”是行区分字，而“filename”是一个磁盘文件的名称。如果没有指定行。那么文件的所有行都写进去，如果没有指定文件名，那么缺省的文件是原先正在编辑的文件，所以，编辑命令

```
w
```

将把工作文件的所有行写到这个原先的文件中。

有时，你只想保留若干行，或者把若干行保留在另一个文件中。编辑命令

```
10,20 w safety
```

将把第10行到20行写到名为‘safety’的文件中。在对这些行作实质性改变之前，这可能是一个十分可靠的预防措施。

5.8 结束编辑工作

在你完成了对于原先文件的修改和更新之后，你可以用退出（“q”）命令离开编辑程序。退出命令没有什么任选项或行号：

```
q
```

如果你想保持所作的改变，那么你必须十分小心地把所有的修改都保留到永久文件中。当你打入退出命令时，编辑程序是不会自动地把正文缓冲区写到永久文件中去的。

绝大多数编辑程序有这样一种特性，如果你已经对工作文件作了修改而没有更新原先的文件，那么它将提醒你（或者是象问

号那样隐蔽的提醒符)。如果你打入命令

q

而编辑程序用一个“?”或用类似于“No write since last change”那样的消息回答你,那么你就知道了还没有更新永久文件。为了让你更新这个永久文件,编辑程序将忽略这个退出命令。此时,你可以打入一个写命令来保留临时文件,要不就重复退出命令从而真正地退出编辑程序。图5.2是一个编辑对话的简短例子。

通常命令

Q

可以用来退出编辑程序而不去检查从上次写以来是否已经作过修改。

%	
% <u>ed firstsession</u>	[编辑 ‘firstsession’ 的shell 命令]
a	[附加正文的编辑程序命令]
<u>This is line 1.</u>	[附加的正文的第1行]
<u>This is line 2.</u>	[附加的正文的第2行]
<u>This is the last line</u>	[附加的正文的第3行]
.	[重新进入命令方式的编辑程序命令]
<u>1,3p</u>	[打印第1到第3行的编辑命令]
This is line 1.	
This is line 2.	[由编辑程序印出这三行]
This is the last line	
w	[把内容写到磁盘上去的编辑命令]
<u>54</u>	[编辑程序印出写入的字节数]
q	[退出编辑过程的编辑命令]
%	[由shell印出的UNIX 系统shell提示符]

图5.2 一个简短的编辑对话

在这个简短的编辑对话中,编辑程序被调用来编辑文件‘first session’,少量几行正文被加到工作文件中,然后该文件被写回到永久的磁盘文件(‘firstsession’)中去,并且结束这段编辑对话。这一对话中的命令和正文在图中的左半边展开,而括在方括号中的注解(译文中以中文出现——译注)展示在图中的右半边。

5.9 行和行号

大多数编辑命令在若干行或一组行上操作。例如，印出命令可以印出一个指定行或一组指定的行。附加命令把正文附加到某行的后面，写命令把一组行写到输出文件中。

由于行对于UNIX系统的正文编辑程序是如此之重要，因此，你总期望有许多途径来标识正文行。自然，可以通过提及行号来标识行。在文件中的正文的第一行行号是1，第二行行号是2，等等。命令

```
2p
```

告诉编辑程序印出行号为2的行（不是两行）。

在一个小文件中，通过行号来标识行是容易的，因为只有少量的行。然而，在一个有几十行的文件中用行号就相当麻烦；而在有成千上万行的文件中用行号就更难了。

在大文件中，标识行的一种方法是使用相对行号。如果你用了数字“-1”，那么你表示的是当前行的前一行。只要一个数字前有正“+”或“-”号，编辑程序就把这个数字解释成是相对行号。相对行号指明了相对于当前行的一行。例如，如果当前行的行号是3，那么“-1”表示行号2，而“+2”表示行号5。

命令

```
-5, +5p
```

将印出当前行前面的五行、当前行及当前行后面的五行。当这个命令完结时，当前行行号将比原来增加5。

通常，当你在修改一个正文文件时，你仅集中在整个资料的一小部分上，相对行号使你可以在当前行邻近的指定行上修改。

标识行的另一条途径是采用上下文关系。上下文识别是一种非常强有力的技术。可通过在行号位置上打入

`/hello/`

来定位当前行后包含正文模式“hello”的第一行。你可以在使用行号的任何地方用上下文模式。如果你想印出跟在当前行后包含正文模式“program”的第一行，那么可以打入编辑命令

`/program/p`

如果你想印出从包含“program”的第一行到包含“PASCAL”后的第一行的所有行，那么可以打入编辑命令

`/program/,/PASCAL/p`

当你在行号的位置上打入了上下文模式时，编辑程序从当前行以后开始检索该模式。如果直到文件结束都未找到该模式，那么编辑程序跳到文件的头上，从该文件头上一直检索到当前行。模式找到的第一行用作为行号。如果正文模式未找到，那么编辑程序就打印出一个问号或者一个简短的出错消息。

除了我们已经提及的向前检索外，还可以用问号把正文模式括起来的方法来控制编辑程序执行反向的上下文检索。编辑命令

`?world?p`

将从当前行开始向后检索，印出遇到包含模式“world”的第一行。如果编辑程序检索到达了文件的开头而没有匹配成功，那么编辑程序就跳到文件的末尾，并且开始反向检索直到当前行。

术语“环绕”用于描述检索期间从开头跳到末尾（或从末尾跳到开头）这样一种过程。概念上，应该把文件想象为连接成环的一系列行。文件的最后行正是环中第一行之前的一行。当然，文件实际上并不是按环形存放的，但是在进行正文检索工作时记住环形是十分有用的。

你可以象

`?world?-3,.p`

那样，用相对地址来组合上下文地址，这条命令向后检索以寻找包含模式“world”的行，然后从该行开始向前退三行，再从那儿一直印到当前行。

编辑程序也允许你把尾随的正和负号加到一个行区分符上。

命令

```
?world? - - -, p
```

与前述的命令一样。你也可以通过打入命令

```
- - -
```

在文件中退回少数几行；或者用

```
+ + +
```

前进几行，对于浏览一个文件来说，使用尾随的正或负号是很有用的。

有时，你可能需要把一个上下文行号转换成一个绝对行号。
编辑命令

```
. =
```

将显示当前行的绝对行号。你也可以使用象

```
/PASCAL/=
```

那样的上下文模式来发现哪一行包含字“PASCAL”。

你可以在编辑程序中交替地使用绝对的、相对的和用上下文行来说明的行号。选择什么行号取决于你正在干什么。在任何不会含混的地方，绝对行号是最安全的；而如果了解你处在文件的什么地方，那么相对行号也是相当安全的；用上下文行说明的行号非常方便，但是相当危险。

5.10 删除正文行

编辑程序的删除命令(“d”)从正文中删除一行或多行。如果不指明行号，那么删除命令就删除当前行。如果指明了行号(或某种上下文模式)，那么删除命令就删除指明的行。如果指明了两个行号，那么删除命令就删除这两个行号之间整个范围的正文行。例如，考虑下列编辑命令：

```
10d
```

```
+ 10d
```

d

10,15 d

20, /there/ d

第一个命令删除行号为10的行，第二个命令删除当前行后的第10行，第三个命令删除当前行，第四个命令删除第10到第15行的内容，第五个命令从第20行一直删到包含模式“there”的行。

使用删除命令时，应当十分小心，因为很容易由于错误而删除了不该删去的行。在某些编辑程序中，复原命令(“u”)可用来消除前面命令的动作。然而，许多编辑程序并不包含复原命令，也可能你没有意识到这种损害。以至太迟了而不能使用复原命令。

为了避免灾难性的删除，一种良好的习惯是在你删除之前先把它们印出来。如果你想删除第10行到第20行的内容，那么应该在打入命令

10,20 d

真正删除之前，先打入命令

10,20 p

弄确实是否定位了正确的行。要记住，印出命令通常只印出当前行，所以如果你想删除当前行后的第10行，可以通过打入命令

+ 10p

弄确实这是正确的行，然后打入命令

d

来删除这行。

删除行时，另一个良好的习惯是使用绝对行号，除非你已成了使用编辑程序的专家。如果你想删除从包含模式“here”到包含模式“there”的一组行，那么最好不要草率地用命令

/here/, there/ d

较为谨慎的应该是先打入命令

/here/ p

找到且印出要删除的第一行。如果印出的行是正确的，那么再打

入命令

. =

印出当前行（要删除的第一行）的行号。作为例子让我们假定印出的行号为20。然后打入命令

/there/

找到且印出要删除的最后一行。再用命令

. =

印出当前行（要删除的最后一行）的行号。假定印出的行号为30。作为最后的防护措施，要使用命令

20,30p

检查一下以绝对保证从第20行到第30行是打算删除的行。最后打入命令

20,30d

真正地删除这些行。这里所示的途径是过分的小心了，但是采取这些预防措施通常要比重新构造失去的10行（或100行）信息来得更容易。

上下文行号对于检索正文来说是重要的，但是当你改换正文时，因为在一般正文中象“there”那样的正文模式出现的次数并不多，所以使用上下文行号是不聪明的。

5.11 插入和修改正文行

编辑程序的插入命令（“i”）用来在指定行前插入正文。插入类似于附加（“a”），只是正文是在指定行前而不是指定行后加上这点不同。如果没有指定行，那么在当前行前插入。

插入命令让编辑程序进入正文进入方式。在打入插入命令后，你打入的每样东西都解释成为正文，并且被加到文件中。通过打入由单个圆点组成的行，就能停止插入正文。

编辑程序的修改命令（“c”）首先从正文中删除一行或多行，然后正文交互式地附加到被删除行的地方。象插入和附加命令那

样，修改命令能使编辑程序进入正文进入方式。

图5.3展示了使用修改和插入命令的编辑对话的一个片段。

<u>3p</u>	[印出第3行的命令]
<u>Thursday</u>	[编辑程序印出第3行]
<u>3i</u>	[在第3行前插入的命令]
<u>Tuesday</u>	[插入的第1行]
<u>Wednesday</u>	[插入的第2行]
.	[退出正文插入方式]
<u>2,5p</u>	[印出第2至第5行的编辑命令]
<u>Monday</u>	[编辑程序印出第2行]
<u>Tuesday</u>	[编辑程序印出第3行]
<u>Wednesday</u>	[编辑程序印出第4行]
<u>Thursday</u>	[编辑程序印出第5行]
<u>3,4c</u>	[改变第3行到第4行]
<u>Tuesday,Wednesday</u>	[新的正文]
.	[退出正文进入方式]
<u>2,4p</u>	[印出第2到第4行的编辑命令]
<u>Monday</u>	[第2行]
<u>Tuesday,Wednesday</u>	[第3行]
<u>Thursday</u>	[第4行]

图5.3 插入和修正文行

本图展示了先是插入命令后是修改命令的编辑对话的一个片段，实际的编辑对话展示在左半边，而注解（已译为中文——译注）展示在右边的方括号中，由用户打入的项下边划线。要记住，这些仅是编辑对话的一个部分。

5.12 移动和传送正文行

编辑程序的移动命令（“m”）是用来在文件中把正文从一个地方移到另一个地方。如果没有指定源行，那么当前行就移到目的行的后面。如果只指定一个源行，那么该指定行就移到目的行后。如果两个源行都指定了，那么这组指定行就移到目的行后。

下列命令展示了所有三种形式的移动命令：

```
m 50
30 m 31
/hi/, 50 m 0
```

第一个命令把当前行移到第50行后，第二个命令把第30行移到第31行后，而第三个命令把一组行移到第0行后。第三个命令是很有趣的，因为指定的正文被移到了第0行后。在UNIX系统编辑程序中，正文的第1行行号总是1，但是在概念上第0行的确存在，以致正文可以移到一个文件的头上。任何放在第0行后的正文总是在该文件的开头。

传送命令(“t”)与移动命令几乎一样；唯一的区别是指定的正文在传送时并不损害。移动命令取走指定的行，并且从那里把它删除，再把取走的行放到其它什么地方；而传送命令取走指定行的正文，并且把它们放到目的地。移动命令对文件的大小没有影响，而传送命令却增加了文件的大小。图5.4展示了使用移动命令和传送命令的编辑对话的一个片段。

5.13 替换正文

替换命令用于在一行上把一种模式改成另一种模式。替换命令的一般格式是

```
n1, n2 s/pat1/pat2/
```

这里n1和n2是行的说明符，而pat1和pat2是正文模式。该命令导致在指定的若干行上用pat2替换pat1。

替换命令最简单的形式是

```
s/pat1//
```

它把由pat1代表的正文从当前行中删除掉。

如果工作文件的第20行是“The UNIX Operating System”。那么命令

```
20s/UNIX/UNIX/
```

将把错误“UNIX”改成“UNIX”。如果你习惯把“UNIX”

<u>45,47p</u>	[印出第45到第47行]
The goat	[第45行]
of the goatherder.	[第46行]
got the goat	[第47行]
<u>47m45</u>	[把第47行移到第45行后]
<u>45,47p</u>	[印出第45行到第47行]
The goat	[第45行]
got the goat	[第46行]
of the goatherder.	[第47行]
<u>45,47t47</u>	[把第45到47行传到第47行后]
<u>45,50p</u>	[印出第45到第50行]
The goat	[第45行]
got the goat	[第46行]
of the goatherder.	[第47行]
The goat	[第48行]
got the goat	[第49行]
of the goatherder.	[第50行]

图5.4 移动和传送正文行

本图包含了演示移动和传送命令的编辑对话片段。左边展示了实际的编辑过程，右边包含了注解（已译成中文——译注）。由用户打入的字符下边划线。

打成“URIX”，那么命令

1, \$s/URIX/UNIX/

将把文件每一行中第一次出现的“URIX”都改成“UNIX”。

如果在工作文件中的当前行是“Who'ss on first?”，那么命令

s/s/

将把多出的“s”从第一个字中去掉。

如果工作文件的当前行是“Don' tread on me”，那么，命令
s/Don' tread/Don't read/

将把原来的行改成“Don't read on me.”。你只需要提供足以使编辑程序能识别要改变正文的上下文就行了。因此，下列命令中任何一个都可以完成与上面命令相同的修改：

```
s/n'tr/n't tr/  
s/'t't t/  
s/'/t /  
s/t/t t/
```

当你替换时想省事而欲提供尽可能少的上下文时要特别小心，因为这很容易陷入困境。请考虑下列正文行：

The rain in Spain falls mainly on the plain.

正文模式“he”出现了两次，模式“ain”出现了三次，模式“in”出现了四次，而模式“a”和“i”各出现了五次。

假定前面的行已被错打成“The rain in Spain falls mainly on the plainly.”很清楚，你的目的是想把“plainly”中多出的“ly”去除，改成“plain”。你不能打入命令

```
s/ly//
```

因为这样一来“去除的是“mainly”中的“ly”而不是“plainly”中的“ly”。解决的办法是指明一个可以清楚地标明错误的“ly”的正文模式，命令

```
s/plainly/plain/
```

将能作出正确的修改。你必须记住，为了正确地指明“ly”要包括大量上下文，因为这里有两个“ly”。屏幕编辑之所以流行，原因之一是屏幕编辑允许你在一行上改动正文而不必使用替换命令。图5.5展示了使用替换命令的编辑对话的一个片段。

<u>1,3p</u>	[印出第 1 行到第 3 行的编辑命令]
Jack be nimble	[第 1 行]
Jack be quick.	[第 2 行]
Jack jump over the candlestick.	[第 3 行]
<u>1,3s/Jack/Jill/</u>	[把“jack”改成“jill”]
<u>1,3s/be/is/</u>	[把“be”改成“is”]
<u>3s/jump/jumps/</u>	[把“jump”改成“jumps”]
<u>1,3p</u>	[从第 1 行印到第 3 行]
Jill is nimble.	[第 1 行]
Jill is quick.	[第 2 行]
Jill jumps over the candlestick.	[第 3 行]
<u>s/candle/broom/p</u>	[把“candle”改成“broom”并打印结果]
Jill iumps over the broomstick.	[修改过的行]

图 5.5 替换命令

编辑对话的这一段展示了使用替换命令的若干例子。要注意，在第一行到第三行上把“be”替换成“is”的命令在第三行上是执行不成的。如果在这组的任何一行上执行替换，那么不会发生任何错误。然而，当你指明替换应该在一组各行上出现时，如果替换在所有行上都失败，那么这就是一个错误。要注意，编辑程序不会自动地印出修改过的行（或若干行）。然而，如果你想要编辑程序印出修改过的行，那么，就象在本图中最后的替换命令中那样，在命令的后面打入“p”。要注意，改变过的行被印出来了。如果你正在进行整个文件替换，那么，尾随“p”仅仅把最后替换的行印了出来。

第六章 UNIX文件系统

文件是命了名的一组信息，它们存放在象计算机磁盘或磁带那样的海量存储设备上。文件系统是一组文件的“组织”结构。在许多计算机系统中，所谓“组织”就是把文件归并成一堆一堆。遗憾的是，对于带有可以存放数十万文件的大型新式磁盘的多用户计算机系统来说，简单的文件系统是不适应的。

若干思想对 UNIX 文件系统的开发产生了影响。最重要的思想是方便性。UNIX 系统需要一个支持用户在逻辑上组织他们的文件的文件系统。一组文件称为一个目录，UNIX 系统鼓励你在必要时建立目录。

将文件组织成为更容易管理的小组是组织文件的良好开端，但是这并没有从根本上解决问题。如果你有几百个用户，每个用户有若干目录，外加十来个用于系统信息的目录，那么你就完全陷入了不可管理的困境。UNIX 文件系统的关键思想是它的层次结构。世界上还有某些其它的层次结构系统，比如普鲁士军队，组合的梯子以及全世界的基督教徒，等等。

文件存取权是多用户计算机中文件系统的—个主要部分。没有存取权确认系统，就没有人愿意把秘密的信息存到计算机中，也没有人能够担保系统文件和表格的安全性。存取权是一个共享的计算机系统必须有的功能。

UNIX 系统新奇的特性之一是所有的 I/O 硬件与特别文件连在一起。对 I/O 硬件本身的存取和对普通磁盘文件的存取相仿。每一个 I/O 设备（打印机、终端、磁盘等等）至少有一个特别文件。一个程序为了真正地存取 I/O 硬件，可以存取代表这个 I/O 的一个特别文件。虽然这看起来似乎很复杂。但是与别的大

多数计算机系统比较起来，这实际上是一个极大的简化。

UNIX 文件系统所有这些方面都将在本章的下面各部分加以讨论。为了有效地使用 UNIX 系统，你应该了解这些思想并有使用的体会。少量高级的文件系统课题在 18.11 至 18.13 节中介绍。对于大多数用户来说，并不强求他们去读这些节。UNIX 系统用来维持文件系统的某些数据结构在 19.6 节中讨论。

6.1 普通文件

普通文件用于存放信息。一个普通文件可能包含了你能执行的程序、资料的正文、公司的记录、或者计算机可以处理的任何其它类型的信息。在与 UNIX 系统对话期间，你会碰到大量普通文件。

普通文件是计算机系统中生气勃勃的一个部分，因为它们允许信息永久地存放。没有长期的信息存储器，计算机的信息处理能力就不会很有用。除了普通文件之外，UNIX 系统还包括目录文件、特别文件，某些系统还包括指定的管道（指定的管道也称为先进先出文件，参见 18.11 节）。普通文件是用作一般信息长期存储的唯一的文件类型。

文件名可长达 14 个字符，在同一个目录中，没有两个文件可共用同样的名字。然而，一个文件有几个名字倒是可能的（见 9.2 节中的 `ln` 命令）。你可以随使用你想要的字符作为文件名，但包含了不可打印的字符、空格符、以及 shell 元字符的文件名是很难用的。由于每一个 UNIX 系统目录总有文件名 `‘.’` 和 `‘..’` 在里边，因此，不能用这些名作为你的私有文件名。

UNIX 系统并不在文件命名上强加任何约定。然而，某些 UNIX 系统程序期望文件命名时带有后缀。例如，带有后缀 `“.sh”` 的文件（比如 `‘bakup.sh’`）通常是 shell 程序，带有后缀 `“.bas”` 的文件（比如 `‘aster.bas’`）通常是 BASIC 程序，而带有后缀 `“.c”` 的文件（比如 `‘xrefer.c’`）通常是 C 语言源程序文件。

按惯例，包含可执行程序的文件（比如‘who’）没有后缀。

有两种类型的普通文件：正文文件和二进制文件。正文文件只包含ASCII（美国标准信息交换码）字符。而二进制文件包含每个字节可能有的256种值。让我们先讨论一下正文文件。

普通终端和计算机可以识别大约100个ASCII字符。大多数终端可以显示下列可打印字符：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
!@#$%^&*()_+ = ~`{|
[]:;<>.,?/\|
```

除了上面展示的可打印字符之外，ASCII字符集还为空格符、水平和垂直制表符、换行符、格式馈给（换页）符以及全部的控制符定义了码子。

普通文件的一个例子是“日志消息”文件（‘/etc/motd’），每当你在系统上注册时，系统就会把其中的内容打印在你的终端上。通过打入命令

```
cat /etc/motd
```

可以在任何时候把日志消息文件打印在你的终端上。UNIX系统实用程序cat经常用来在你的终端上显示正文文件（参见8.7节）。

所包含的代码如果不属于ASCII字符集部分，那么这种文件就称为二进制文件。由于二进制文件使用字节的所有可能值，因此，对于存放信息来说，二进制文件是一种更为有效的方式。但是二进制文件不能在你的终端上直接打印出来，因为对于每个字节的256种可能值，大部分不是可打印的ASCII字符。

你可以通过使用八进制卸出（od）程序（参见7.17节）检查二进制文件的内容。od取出文件中的值并且把它们转换成可打印

的字符。如果你使用了命令

```
od /unix
```

那么你将卸出文件 /unix。在大多数系统中，文件 /unix 包含了当前正在操纵计算机的 UNIX 系统核心的一个副本。

实际上，你打入的每一个命令都涉及了普通文件。对于控制一组普通文件来说，有四个命令尤其重要，这四个命令是 mv (移动)、cp (复制)、ln (联结) 和 rm (删除)。这四个命令在第九章讨论。

6.2 目录文件

目录文件是包含一组文件的文件。UNIX 操作系统管理目录系统。执行程序可以读取目录文件，但是为了保证目录系统的完整性，操作系统阻止程序修改目录文件。

执行程序可以请求系统建立一个文件而把登记项加到目录中去。也可以请求系统删除一个文件而把登记项从目录中删除。系统自始至终有责任对目录文件进行修改。

在目录中列出的文件可以是普通文件、目录文件、特别文件或（在某些系统中的）先进先出文件。

每一个用户有一个称为主目录的特别目录。当你注册到系统中去时，你就在你的主目录上了。在与 UNIX 系统对话期间，你可以自由地从 一个目录移到另一个目录。如果你想移到名为 ‘/bin’ 的目录去，那么你就打入命令

```
cd /bin
```

你正在的那个目录称为当前目录或工作目录。

在 UNIX 系统第七版中，系统将记住你的主目录，以致你在任何时候都可以通过打入命令

```
cd
```

返回到你的主目录去（在较老的系统中，cd 命令通常取名为 chdir，并且为了返回到主目录，你通常必须指明主目录的名字）。如果你

打入了pwd命令，那么将印出当前目录的名字。cd和pwd在7.1节讨论。

mkdir（构造目录）命令用于建立一个目录，而**rmdir**命令用于删除一个目录。创建的目录中除了标准文件‘.’和‘..’以外，没有别的内容（标准登记项‘.’和‘..’在下面讨论）。你只能删去一个空目录（‘.’和‘..’除外）。**mkdir**和**rmdir**在9.4节中讨论。

6.3 具有层次结构的文件系统

在UNIX系统中的文件组成目录，而目录构成一个层次结构。层次的顶部是一个称为根目录的特殊目录。根目录包含了各种各样的与系统有关的文件，而且它通常包含一些诸如‘/bin’，‘/usr’，‘/dev’，‘/etc’和‘/lib’那样的标准目录。一个典型的、但是极为简化的文件系统层次结构如图6.1所示。

层次式文件系统的组织机构是优越的。让我们用一家公司打个比喻，在一家公司里，你可能允许每一个工人直接向总裁报告。这种组织形式对于小型的夫妻杂货店来说，会工作得很好；但是对于象通用汽车公司那样的巨型企业来说，这种组织形式可能是灾难性的。类似地，在UNIX系统中，松开文件和根目录之间的紧密连接，组织了层次式的系统是一个很大的优点。

UNIX文件系统结构常常称为树形结构，因为它的结构类似于一棵树。当前的子树是指在层次结构中比当前目录所处的层次为低的这么一部分，如果‘/usr’目录是当前目录，那么，所有用户目录及这些用户目录的子目录等等都是当前的子树，如图6.2所示。除非你指明了另外的目录，否则大多数UNIX系统命令用当前目录下的文件工作。少量UNIX系统命令用当前子树工作。

6.4 路径名

在当前目录中的文件是直接可存取的，它们可以通过直接打

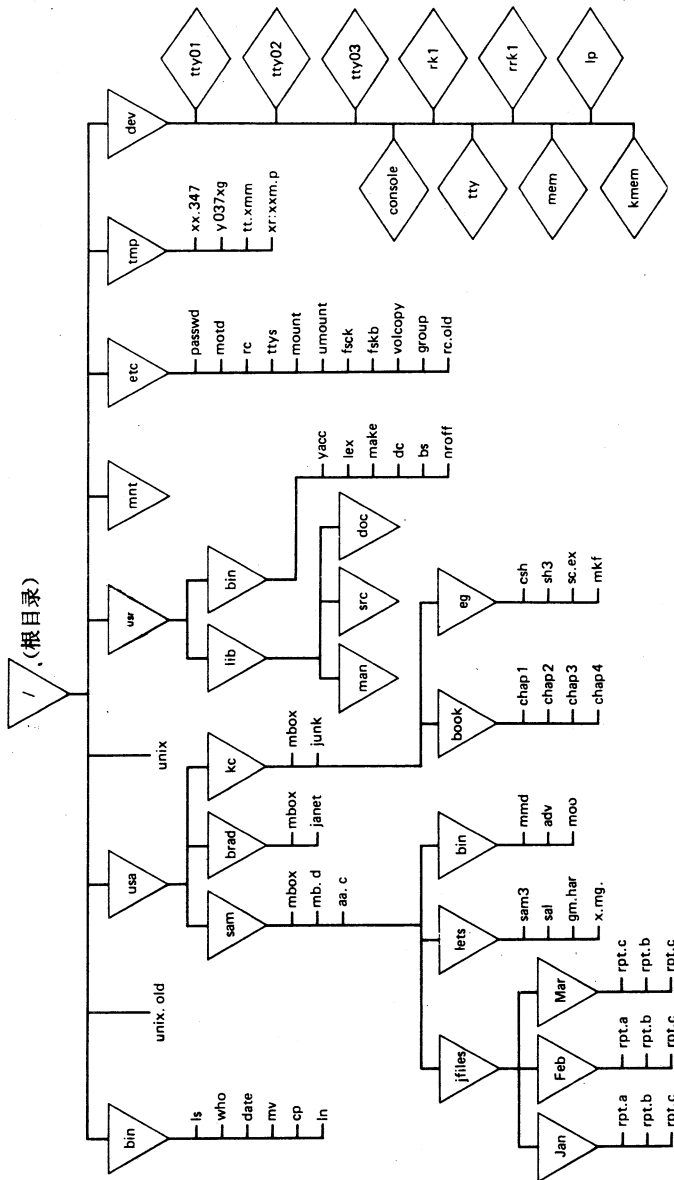


图6.1 一个典型的UNIX文件系统结构图
 在本图中，目录以三角形表示，特别文件以菱形表示，普通文件不加框。

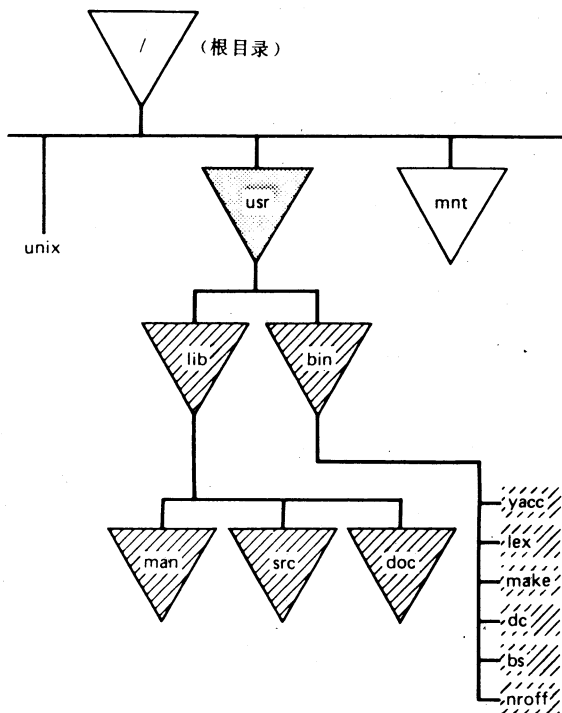


图6.2 当前目录和当前子树

在本图中，当前目录是 ‘/usr’（以阴影表示）。当前子树由文件系统层次结构中 ‘/usr’ 下的所有目录和文件组成（以斜线表示）。

入它们的名字来引用。不在当前目录中的文件必须用文件名来引用。路径名指明穿过文件系统直至所要文件的一条路径。穿过文件系统的路径只可以从两个地方，即你的当前目录或根目录中的一个地方开始。以 “/”（斜线）字符打头的路径名是绝对路径名，

指明从根目录开始的一条路。其它所有的路径名都是相对路径名，它们指明从你当前目录开始的一条路径。

除根目录以外，每一个目录都包含名字‘.’和‘..’的登记项。这两个项是把文件系统联合在一起的“粘合剂”。项‘.’是当前目录名的别名。想读取当前目录文件的程序可以直接使用名字‘.’而不必到处试探来决定以前创建时指定给这个当前目录的名字。

名字‘..’是当前目录的父目录的别名。在每一个目录中，允许用项‘..’指明“攀登”文件系统的路径名。要注意。在一个目录中，除‘.’和‘..’外，其它所有项都指明在文件系统层次中较低层的文件。用名字‘.’和‘..’的想法是很合适的，因为在与UNIX系统交互作用时，将非常频繁地使用这些名字。

下面几个简单的规则应用于所有的路径名：

1. 如果路径名以斜线打头，那么路径从根目录开始。除此以外，其它所有的路径都从当前目录开始。

2. 路径名或者是由斜线分开的一系列名字，或者是单个名字。在一串名字中，起始名字（如果有的话）是目录，最后名字是目标文件，目标文件可以是任何一种类型的文件。

3. 你可以通过在路径名中指明名字‘..’来往上“攀登”文件系统层次。在路径名中，除‘..’外的所有其它名字往下降低层次。

4. 在路径名中，不允许有空格。

让我们举出路径名的若干例子。路径名‘/usa/kc’是指明文件‘kc’的一个绝对路径名。由于路径名以斜线开始，因此，它是从根目录开始的一个绝对路径名。很明显，目录‘usa’是根目录的一个子目录（见图6.3）。

路径名‘jfiles/Jan/rpt.a’是一个相对路径名，因为它不是以斜线打头的。因此，路径从当前目录开始，在本例中，当前目

录是 '/usa/sam'(见图6.4)。目录 'jfiles' 是当前目录的一个子目录, 'Jan' 是 'jfiles' 的一个子目录, 而 'rpt.a' 是 'Jan' 中的一个文件。

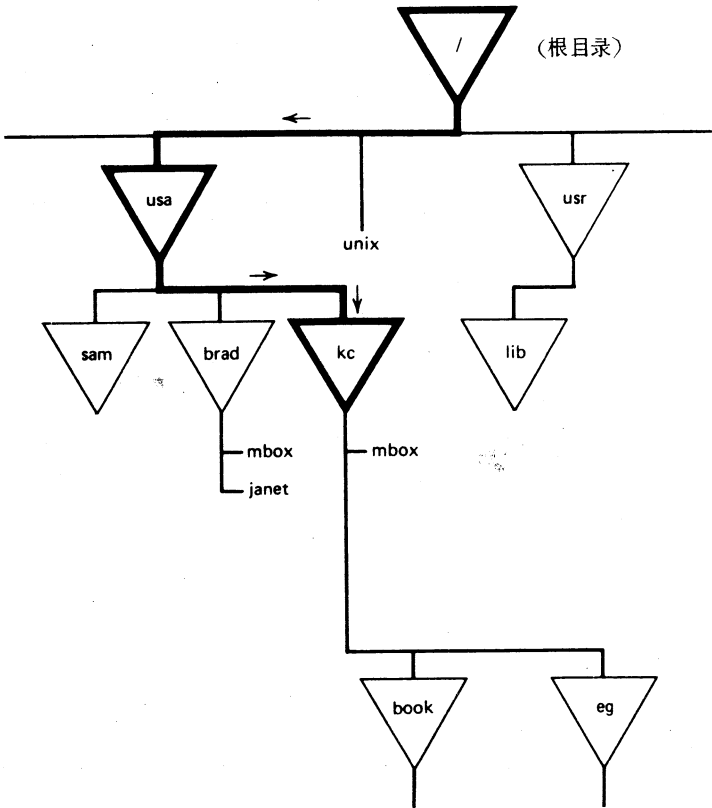


图6.3 路径名 '/usa/kc'

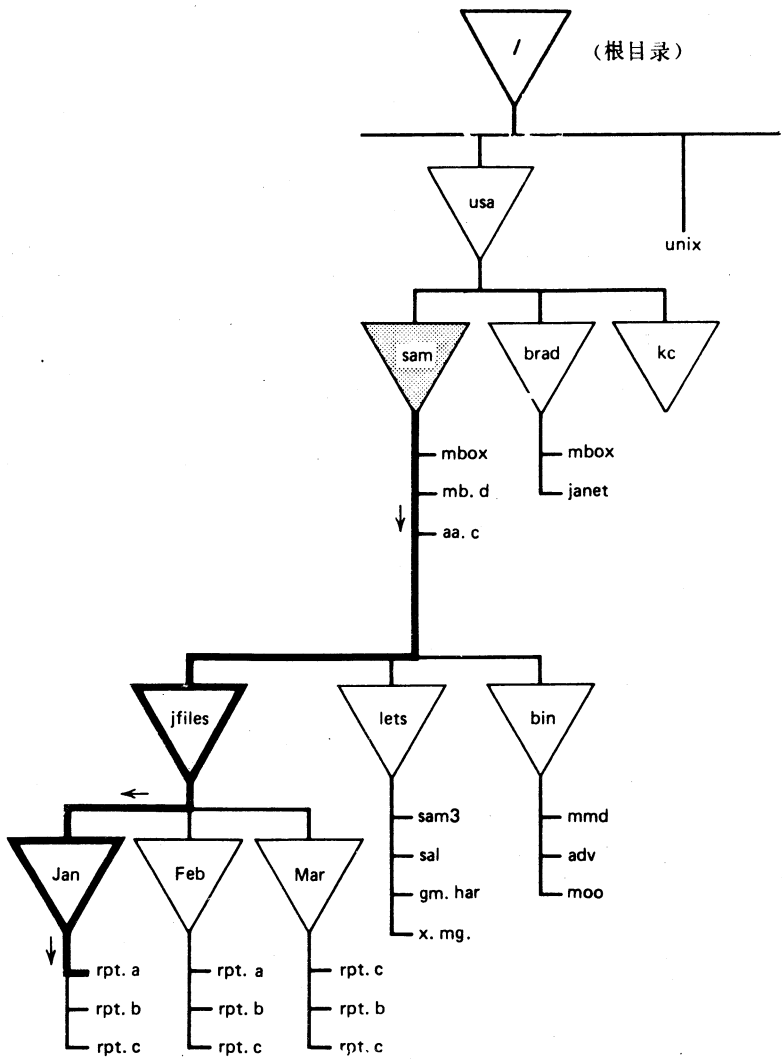


图6.4 路径名 'jfiles/Jan/rpt.a'

路径名 ‘.. /.. /brad/janet’ 理解起来较为困难。路径从当前目录（在本例中是 ‘/usa/kc/eg’，见图6.5）开始，这条路径导至 ‘/usa/kc/eg’ 的父目录，即 ‘/usa/kc’。然而路径

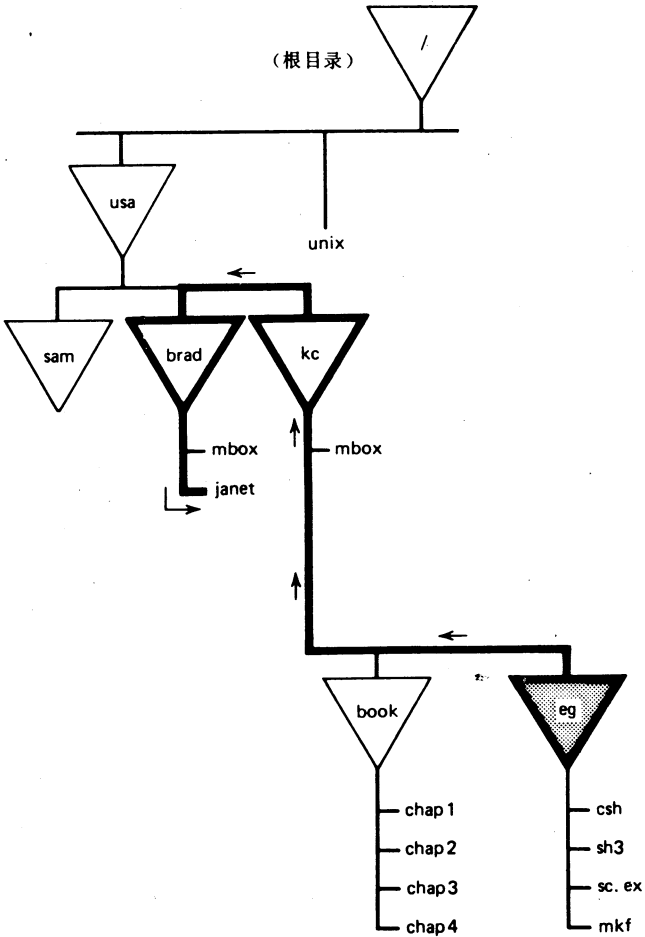


图6.5 路径名 ‘.. /.. /brad/janet’

进一步“攀登”到‘/usa’,然后再下降到目录‘brad’。在‘brad’目录中的目标文件是‘janet’。长的路径名很难使用,所以通常把目录改到动作发生的地点比较好。

6.5 文件类型和方式

我们已经说过,文件是信息的集合。就现在的情况看来,这仍然是正确的,但是,对一个文件来说,还有比它包含的信息更多的东西。UNIX 系统保持了描述每个文件的各色各样的信息。描述文件的信息包括存取权限、文件类型、有关文件的重要日期、文件的大小以及文件在磁盘上的精确位置。

文件类型和方式的意义遍布整个 UNIX 系统。许多操作都假定文件是这种或那种类型的。例如,你不应该使用命令

```
cd /usa/kc/mbox
```

因为‘/usa/kc/mbox’不是一个目录,而是包含电子邮件的一个普通文件。类似地,命令

```
cat /usa/kc
```

将产生莫名其妙的东西,因为‘/usa/kc’是一个目录,而不是一个正文文件。

每次你存取一个文件,系统就检验你执行这次存取的权限。正常设置的权限使得你有相当的自由来管辖你私有的文件,但是要存取你的近邻的文件就存在许多的限制。某些更秘密的 UNIX 系统,用出错消息直接告诉你,你不能存取某个文件,因为文件存取保护系统不允许你存取。

在 UNIX 系统中工作时,首先必须要记住每个文件的类型。文件分成几种类型:目录文件、普通文件、特别文件以及某些系统中的先进先出文件。目录文件是系统为了维持层次结构的文件系统而设置的文件,普通文件是用户存放信息的文件。允许用户读取目录文件中的信息,但是只允许系统写目录文件。

特别文件用于为 I/O 硬件提供连接。有字符特别文件和块特

别文件两类，这两类文件之间的区别在下一节讨论。特别文件常常用于提供对于磁盘或磁带存储设备的接口。然而，在特别文件中，实际上根本不存放信息，信息存放在磁盘或磁带上。当程序从一个特别文件读取信息时，信息实际上来自于有关的 I/O 设备。并且，当程序向一个特别文件写信息时，信息实际上也是送到有关的 I/O 设备上。每一种特别文件与主设备号和次设备号两个数字有关。当存取特别文件时，为了找出在逻辑上与每个特别文件连接的是什么 I/O 硬件，系统核心需要使用这种设备号。

从程序员看来，特别文件是大大简化了，但是对于大多数用户来说，并不需要知道有关特别文件的很多东西。例如，在某些装置中，为了获得打印输出，你必须重新定向输出到打印机的特别文件中去；但是，大多数装置有程序来处理这种打印机。程序通过特别文件开展工作，用户是不用过问的。

在 UNIX 系统中，在一个文件上可以做三种操作：读、写和执行。读一个文件意味着该文件的内容可以得到。写一个文件意味着文件的内容被改变了。执行一个文件意味着或者把文件装入了主存，或者执行存放在文件中的机器指令，或者从文件中读取 shell 命令并且执行这些命令（对于一个目录文件，执行许可意味着你在分析路径名的过程中可以检索该目录）。

每一个 UNIX 系统文件都归一个特定的用户所有，而且总是与一个特定的组相关（在 UNIX 系统中，一个组是有某些共同关系的一组用户——典型的组来自一个部门的用户、或在一项工程上一起工作的人，等等）。UNIX 系统文件存取保护机制取决于所要求的存取类型（读、写、或执行）以及谁在完成存取。有一组为文件所有者的特权，有另一组为该文件组中的其它成员的特权，还有第三组为其它人的特权。一个文件的特权可以用 ls 命令显示出来。有关例子参见 7.2 节。

通过使用 chmod（改变方式）命令，一个文件的所有者能够控制该文件的许可（在 9.3 节讨论）。所属关系及组间联系也可以

用chown（改变所有者）以及chgrp（改变组）命令来修改（也在9.3节讨论）。超级用户（系统管理员使用超级用户特权来执行普通用户所不能执行的操作）也可以修改普通用户的文件方式，但除非普通用户需要这种帮助，否则在正规管理的系统中，这多半不会产生。

UNIX系统中许多文件有几个名字。一个文件所具有的名字的个数也称联结数，因为每个别名是从一个目录项到文件系统内部管理部分的一个链，目录项都至少包含两个链，这是因为每个目录项至少包含其本身的别名‘.’。带有子目录的目录包含两个以上的链，这是因为每个子目录用别名‘.’定位其父目录。正如9.2节要讨论的，为了给一个给定的文件显式地建立若干名字，可以使用ln（联结）命令。

6.6 特别文件

普通文件的概念是容易理解的，这是因为它们包含的信息，就象在一个文件柜中的文件那样。特别文件理解起来要困难些，因为它们不包含信息，它们用于为存取I/O装置提供一个方便的通道。对于与计算机连接的每一种I/O装置（卡片读入机、终端、磁盘、磁带、等等），至少有一个特别文件。大多数特别文件存放在目录‘/dev’中。通常，特别文件的名称指示出它们与那种类型的设备有关。例如，‘/dev/pt’是纸带读入/穿孔机的特别文件，‘/dev/tty4’是终端（电传打字机）的特别文件，‘/dev/rp0’是RP06型磁盘的特别文件，而‘dev/lp’是一个行式打印机的特别文件。

当程序把数据写到诸如‘/dev/pt’那样的文件中去时，操作系统截取该数据并且把它们送到纸带穿孔机去。当程序从一个诸如‘/dev/pt’那样的文件读取数据时，操作系统实际上从纸带读入机上获得数据。

读或写‘/dev/pt’特别文件时，程序不必知道有关纸带

I/O 装置的任何细节。特别文件是一般应用程序和 UNIX 系统核心内部子程序之间的接口，一般应用程序要求忽略硬件细节，而 UNIX 系统核心内部子程序的存在却是为了要知道计算机硬件的细节。

某些 I/O 设备，一次处理一个字符，这就是面向字符的 I/O 设备。面向字符的 I/O 设备的代表者是终端。计算机一次送一个字符给终端。提供与字符 I/O 设备连接的特别文件称为字符特别文件。

某些 I/O 设备在大批数据一起传送时，工作起来效率很高。大多数磁盘要求 512 个字节一起传送；512 个字节称为一个数据块，用块工作起来更为有效的设备称为块 I/O 设备。提供与块 I/O 设备连接的特别文件称为块特别文件。

大多数块 I/O 设备也有字符特别接口。块 I/O 设备上的字符特别接口也称为原始接口，它们基本上由执行操作系统维护功能的程序使用。块 I/O 设备上的字符特别接口将在讲述 UNIX 系统内部结构的那章作更详细的讨论。

由于在特别文件中没有存放什么字符，因此，在长列表中的长度字段中，并不包含文件的长度。相反，却展示了主、次设备号。主设备号标识涉及的 I/O 设备类型。计算机常常与同类 I/O 设备中的若干台设备连接。比如，UNIX 系统计算机通常与若干（或几十个）终端连接。每一个终端都有一个特别文件，而每个特别文件的次设备号指定是哪一个终端。

UNIX 系统使用特别文件而不用别的什么机制来存取 I/O 硬件，这是因为 UNIX 系统想使 I/O 硬件的接口和普通文件的接口相一致。因此，特别文件作为目录中的项出现，它们是文件系统的一部分，并且由通常的三层保护系统控制对它们的存取。

当终端特别文件不使用时，它们归系统所有。在注册过程中，与你的终端对应的特别文件的所有权就交给你了。由于你占有了使程序与你的终端相连接的特别文件，因此，你可以控制它的存

取方式。当你注销时，你的终端特别文件的所有权也就还给了系统。

提供与磁盘和磁带连接的特别文件通常由系统占有，并且常常限制普通用户对它们的存取。提供与打印机、纸带读入机、阴极显示器、实验设备和类似的设备相连接的特别文件通常也由系统占有，它们的存取方式常常置成使所有的人都可以使用它们。这应该由你把这些资源的使用与其它用户对这些资源的使用协调一致起来，使得不会有两个人同时使用一种资源。

6.7 目录存取方式

对于所有者、同组用户和其它用户，目录有标准的读、写和执行许可。然而，在目录中，这些许可的解释与普通文件中的许可解释不同。

对于目录的读许可意味着允许打开该标准实用程序，并且读取该目录中的信息。例如，ls 程序读取目录以便寻找到它们的内容。如果没有给予读权限，那么就不可能发现什么文件包含在该目录中（直至重新恢复读权限为止）。

在未授予读许可的目录中，也有可能进行操作。例如，在我常常使用包括操作系统源代码在内的所有目录的那个 UNIX 系统上就禁止了读许可，然而，由于我非常熟悉这些目录的组织机构，因此在对这本书进行研究时，考察这些源文件对于我来说就没有任何问题。限制对于目录的读特权摆脱了无知动作的损害，但是这不是真正的保护——这是一件讨厌的事。

对于目录具有写特权意味着你被允许创建或删除目录中的文件。如果仅为了在一个目录中建立或删除文件，那么你不必要有读特权。当你在一个给定的目录中创建文件时，意味着系统已经把创建的文件的名字写到该目录文件中，以写特权控制建立和删除权利就具有很大的直观意义。类似地，在删除一个文件时，系统必须从目录文件中抹去要删除的文件的登记项。

对于某个目录的禁止写特权并不意味着在该目录中的文件不能被修改。只是在某个个别文件上的写特权才控制了修改该文件的权力。

对于一个目录的执行许可意味着系统在分析一个文件名的过程中将检索目录。当你在一个简单文件名的位置上指明了一个路径名时，系统将顺序地为下一个目录的名字检索该路径名中的每一个目录。如果你要请求系统

```
cat /usr/bin/source/README
```

那么，你必须对目录‘usr’，‘bin’和‘source’有执行(检索)许可。禁止一个目录的检索许可真正地防止用户使用该目录中的文件。你不能把目录改到禁止执行许可的目录上去，否则，以后你就不能检索该目录了。

第七章 实用程序的应用

UNIX操作系统的实力之一是：它有一大堆实用程序。不同的装置有不同的实用程序；所以，本书不可能作为系统中所有实用程序完整的指南。但是，我们试图讨论并且展示最有用的实用程序的某些例子。

本章集中讲述监视和控制与UNIX系统交互作用的实用程序。下一章讲述正文文件实用程序，再下一章讲述一般文件的实用程序。第十六章讲述程序员的实用程序，第十八章讲述系统管理员的实用程序。

这些章主要介绍组成UNIX系统有用而强有力的核心技术的一组程序。认真使用UNIX系统的任何人都应非常熟悉其中大多数程序；不准备下功夫的用户也至少要使用其中的一半。核心技术中大型、复杂的程序（例如，shell、编辑程序以及高级的字处理和程序设计辅助工具等等）将分别在本书的其它各章（shell在第四、十二、十三章；编辑程序在第五、十章；字处理在第十一章）中分别讨论。至于有关特定程序操作的专门知识，你应该查阅UNIX系统手册。

初学者常常感到难懂的问题之一是UNIX环境可变。可以举一个例子来说明这种情况：两个不同的用户通常有不同的主目录，并且对文件有不同的存取权限。他们可能有不同型号的终端，并且需要使用很不同的一组程序。UNIX系统的长处之一是它具有支持多种环境的能力。本章论述到的实用程序将有助于你理解当前环境。

7.1 `pwd` 和 `cd` ——显示和改变当前目录

当前目录的名字是指出当前环境的一组最基本的信息。`pwd`（印出工作目录）命令显示当前（工作）目录的名字。

UNIX 系统划分成目录（参见第六章）。在任何给定时刻，这些目录中只有一个是你的当前目录。当你向系统注册时，当前目录就是你的主目录。你可以使用 `cd` 命令（见下一节）从一个目录移到另一个目录。在每次移动之后，最好弄清楚你到达的是否正是你所希望的目录。命令

```
pwd
```

可印出当前目录的名字。注册之后，你立即就在你的主目录上，该目录是由系统管理员指定给你的。我的主目录是 `‘/usa/kc’`，如果我在注册后立即执行 `pwd` 命令，那么在我的终端上就印出 `‘/usa/kc’`。如果没有出现所希望的文件，或者如果事情并不象你期望的那样进行，那么使用 `pwd` 命令可以弄明白你是否处在你所希望的目录上。

对于处在错误目录上的一个补救办法是把目录改变到正确的目录上去。改变目录（`cd`）命令将把当前工作目录改到指定的目录上去。如果没有指定目录，那么新目录就是主目录。例如，命令

```
cd /usr/ron/source
```

将把你移到目录 `‘/usr/ron/source’` 上。

如果你打入 `cd` 命令时未指定目录，那么新目录将是主目录。下面两个命令中任何一个都可以用于从文件系统的任何地方移回到我的主目录（`‘/usa/kc’`）去：

```
cd
```

```
cd /usa/kc
```

`cd` 命令中的路径名自变量或者是象上面第二个命令中那样的绝对路径名，或者是象下列两个命令中那样的相对路径名：

```
cd ../../source
```

```
cd junk/programs
```

这两个命令中，第一个把当前目录置成 ‘source’，它是原当前目录父目录的父目录的子目录。第二个命令要把当前目录置成 ‘programs’，它是原当前目录的子目录 ‘junk’ 的子目录。

7.2 ls ——列出文件

ls 命令用于列出目录的内容并且打印有关文件的信息。ls 命令接受许多自变量和任选项，其中大多数就不在这里讨论了。

ls 命令的每一个自变量或者是一个普通文件（或者是特别文件）名，目录名，或者是任选项（任选项表）。任选项用于控制文件列表的次序以及控制每个文件打印的信息。对于每个普通（或特别）文件自变量来说，打印出所需要的信息。对于每个目录自变量来说，为该目录中所有文件印出所需要的信息（除非用了 “d” 任选项）。

下列四种任选项用得很频繁。更多的任选项在本书末尾的简明手册中提及，你还应该查阅随同你的系统一起发行的材料。

1. “l” 任选项

这是长列表任选项，用于印出大量的有关每个列出文件的信息。没有 “l” 任选项时，就只印出文件名。

2. “t” 任选项

这是时间排序任选项，它根据每个文件修改的日期排列这些文件，最新修改的文件最先印出。时间排序任选项用来展示当前哪些文件在活动。

3. “d” 任选项

这是目录任选项，用于迫使 ls 命令仅仅为自变量表中的每一个目录印出所需的信息。通常，所有在自变量表中指定的目录都要被检索，并且为这些目录中的所有文件印出所要求的信息。

4. “a” 任选项

通常，当你用 ls 命令印出在一个目录中的一列文件时，其名

字是以一个圆点开始的文件将从此组文件中略去不列。然而，用了“a”任选项，即使其名字以一个圆点开始（例如标准项‘.’和‘..’），对应的文件也将印出。

对于自变量表中的普通文件来说，只有该文件名和所要求的信息才被印出来。对于每一个在自变量表中指定的目录，包含在该目录中的文件都列出来。如果没有在自变量中指定文件或目录，那么就列出当前目录的内容。

命令

```
ls
```

将按字母顺序印出当前目录中的文件清单。你也可以用ls印出成组的文件清单。命令

```
ls *.c
```

将（以字母顺序）列出当前目录中其名以“.c”结尾的所有文件。“-t”任选项可以用来根据修改日期而不是根据字母顺序排列文件。命令

```
ls -t *.c
```

将印出在当前目录中，其名字以“.c”结尾的所有文件的清单，此清单根据文件的修改日期排序。

有一件事情迷惑了许多UNIX系统用户，这就是命令

```
ls
```

和命令

```
ls *
```

之间的不同。第一个命令没有自变量，所以，根据缺省原则，将产生当前目录中的文件清单。第二个命令用了元字符“*”，它匹配在当前目录中的所有文件的名字。因此，针对第二个命令，shell将为当前目录下的每一个文件提供一个自变量。如果当前目录下只有普通文件，那么这两个命令将产生同样的输出，但是，如果当前目录下有子目录的话，那么第一个命令只列出子目录名，而第二个命令却列出子目录的内容，因为子目录在自变量表中明

确地提到了（参见图7.1）。

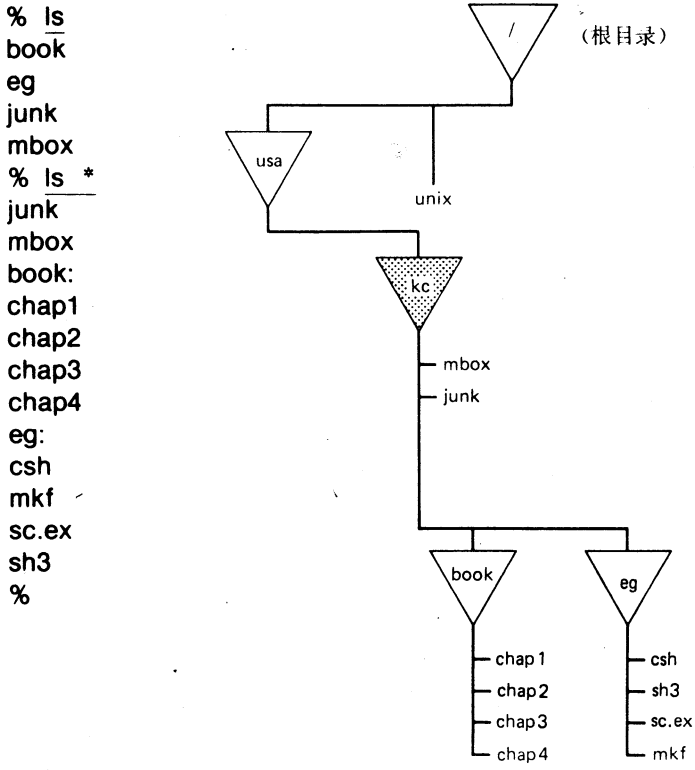


图7.1 使用ls的例子

当前目录包含子目录（在本例中的‘book’和‘eg’）时，命令“ls”和“ls*”作用完全不同。若没有自变量，ls命令只印出当前目录中的文件清单（如上面第一个命令所示）。当ls命令接收一个目录名作为自变量时，为该目录中的每一个文件印出所要求的信息。在这种情况下，命令“ls*”等同于命令“ls book eg junk mbox”，因为shell把元字符“*”扩展成在当前目录下的所有文件的清单。因此，在目录‘book’和‘eg’中的文件也出现在清单中（如上面第二个命令所示）。

当你需要知道有关一个文件的大量信息时，你可以使用“l”

ls -l /etc/rc

-rw-rw-r--	1	root	adm	3488	Jan 20	17:21	/etc/rc
	联结点	文件所有者	组名	文件大小 (按字节计算)	修改日期		文件名

其它用户特权 (可读, 但不可写, 也不可执行)
 同组用户特权 (可读, 可写, 但不可执行)
 文件所有者特权 (可读, 可写, 但不可执行)
 文件类型 (“-”) 表示这是一个普通文件

ls -ld /etc

drwxrwxr-x	1	root	adm	640	Jan 23	19:32	/etc
	联结点	文件所有者	组名	文件大小 (按字节计算)	修改日期		文件名

其它用户特权 (可读且可执行)
 同组用户特权 (可读, 可写, 可执行)
 文件所有者特权 (可写, 可读, 可执行)
 文件类型 (“d”) 表示这是一个目录文件

```

ls -l /dev/rk0
brw-rw---- 1 root adm 2, 0 Jan 23 19:32 /dev/rk0

```

文件所有者
 其它用户特权 (没有任何特权)
 同组用户特权 (可读, 可写)
 文件所有者特权 (可读, 可写)
 文件类型 (“b” 表示这是一个块特别文件)

1 联结数
 root 文件所有者
 adm 组名
 2, 0 主设备号 次设备号
 Jan 23 修改日期
 19:32 修改时间
 /dev/rk0 文件名

```

ls -l /dev/rrk0
crw-rw---- 1 root adm 6, 0 Jan 23 19:32 /dev/rrk0

```

文件所有者
 其它用户特权 (没有任何特权)
 同组用户特权 (可读, 可写)
 文件所有者特权 (可读, 可写)
 文件类型 (“c” 表示这是一个字符特别文件)

1 联结数
 root 文件所有者
 adm 组名
 6, 0 主设备号 次设备号
 Jan 23 修改日期
 19:32 修改时间
 /dev/rrk0 文件名

图 7.2 ls 命令的长格式输出

任选项。使用“l”任选项可以让你看见该文件的方式、对该文件的联结数、该文件的所有者、同组用户、该文件的大小以及该文件的修改时间。对于特别文件来说，在文件大小的位置上印出了主设备号和次设备号。图7.2 展出了用“l”任选项的ls命令的某些典型输出。

理解ls命令长格式输出的输出信息是十分重要的，因为与UNIX系统交互作用涉及到文件访问。ls的长格式输出是揭示每个文件关键信息（文件类型、方式、隶属关系及大小）的唯一途径。在你的系统中，ls命令的输出可能与图7.2 中所示的输出有点不同。因为长列表的格式是因系统而稍有变化的。

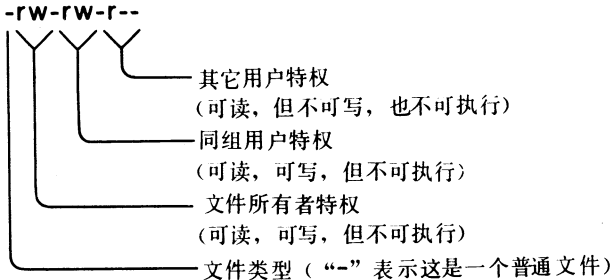
在长格式列表中的第一字段是方式字段。它通常由十个字符组成，第一个字符指明文件类型，余下九个字符指明文件存取特权。作为文件类型的编码（在方式字段中的第一个字符）都在下表中给了出来：

代码	含 义
-	普通文件
d	目录文件
c	字符特别文件
b	块特别文件
p	先进先出（称为管道）文件

文件类型是有关一个文件的最基本的信息。经过一个短时间学习后，你自然会理解在长列表中第一个字符的意义的。

在6.5 节中，我们讨论了可以在UNIX系统文件上进行的三种操作：读、写和执行。我们还要进一步讨论存取特权的三个等级：所有者特权，同组用户特权和其它用户特权。由于有三种存取操作（读、写和执行）以及有三种等级的保护（所有者、同组用户和其它用户），因此，存在与每个文件有关的九种（ 3×3 ）存取许可。许可的第一组三个字符展示文件所有者的读、写和执行特权，第二组三个字符展示同组成员的读、写和执行特权，而

第三组三个字符展示所有其它用户的读、写和执行特权。如果允许某种特权，那么就展示相应的字符（r、w或x）。如果禁止这某特权，那么就展示一个“-”。在图7.2的第一个例子中，存取特权是按下表那样展示的：



如果禁止了某种特权，而你又试图享受这种特权，那么这会受到操作系统的阻止。在许多UNIX系统环境中，与文件相关的读、写特权是较随便的，因为习惯上每个人都允许其它所有人自由地存取文件。在某些别的UNIX系统装置中，为了确保系统完整性，文件的存取方式受到小心的控制。

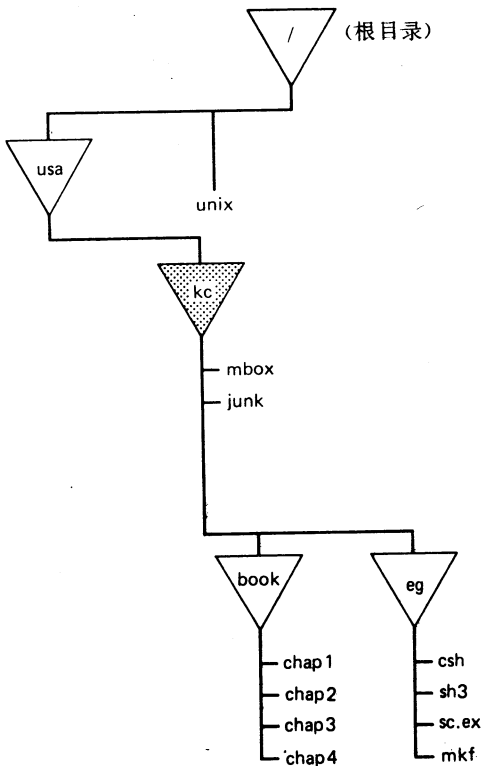
有时，你需要知道有关一个目录的存取方式或其它某些信息，这时如果打入命令

```
ls -l /usa/kc
```

那么你会获得在“/usa/kc”目录中所有文件的长格式列表。当ls命令接收一个目录名作为自变量时，标准的操作是列出在该目录中的所有文件。“d”任选项用于禁止这种正常的操作，而且迫使ls仅为指定的目录列出所要求的信息。命令

```
ls -ld /usa/kc
```

将为“/usa/kc”目录印出文件的长格式列表。这两个命令展示在图7.3中。



```

% ls -l book
-rw-rw---- 1 kc elec 11904 Jan 23 19:32 book/chap1
-rw-rw---- 1 kc elec 33381 Jan 23 09:59 book/chap2
-rw-rw---- 1 kc elec 21804 Jan 23 11:33 book/chap3
-rw-rw---- 1 kc elec 12556 Jan 23 18:08 book/chap4
% ls -ld book
drwxrwxrwx 2 kc elec 128 Jan 24 07:32 book

```

图7.3 列出目录内容

当传递给ls一个目录的名字时,ls命令的通常反应是为该目录中的每一个文件列出所要求的信息。上面第一个命令展示了ls按长格式列出目录‘book’中的所有文件。ls的“d”任选项阻止ls“下降”指定目录去列出它们的内容。在上面所示的第二条命令中,ls按长格式为目录‘book’列出信息,而不是为包含在‘book’中的文件列出信息,因为“d”任选项在场。

7.3 file——推断文件类型

file 命令试图确定是什么类型的信息存放在自变量指定的文件中。file 命令给出文件的内容使人得到有益的启示，而ls命令却难于对文件印出这些东西。file 命令最重要的用途或许是确定文件中究竟包含了正文还是二进制信息。正文文件可以打印在你的终端上，但是二进制文件却不行（在你的终端上试图打印二进制文件可能会导致灾难，因为在该文件中的某些二进制值可能被你的终端或调制解调器解释成控制码）。

如果一个文件中包含了二进制信息，那么 file 程序将确定该信息究竟是可执行的程序还是二进制数据。如果该文件包含的是正文，那么 file 程序将识别这种语言。在大多数系统中。file 程序认识下列若干种语言：shell,C,yacc和FORTRAN，以及英文、法文和西班牙文。自然，确定类别时可能会出错。命令

```
file *
```

将列出在当前目录中所有文件的文件类型。

7.4 date和who——设置或显示日期和显示当前用户

date 命令印出和设置当前日期和时间。一般用户打入命令
date

只能印出日期。超级用户可以引用date命令来设置日期。

who命令印出当前正在使用该系统的用户清单。打入命令
who

可以印出系统中的用户名、他们的终端标识以及他们注册的时间等等的清单。你的注册名也应该在此清单中。更老的某些系统擅长于提出象“你是谁”，“我是谁”（在有些系统中是“who am i”），以及“谁是上帝”那样的哲学问题（也许你的系统仍然认识这些问答）。

7.5 ps——列出进程

进程状态 (`ps`) 命令印出你的所有进程的清单。`ps` 命令常常由系统程序员 (偶尔是由好管闲事的系统管理员) 用于确定在整个系统中正在发生什么。你经常需要 `ps` 程序来确定错误进程的进程标识, 以便你可以消灭它们。在与 `learn` 程序对话期间不妨试一试 `ps` 命令, 以此作为一种实际体验。你可能会对正在进行的进程数目感到吃惊。为了印出你的所有进程的清单, 只要打入命令

```
ps
```

命令名、进程标识号、控制用 `tty` 终端名以及所有进程的累计执行时间就都印了出来 (控制用 `tty` 终端名是 UNIX 系统中与该进程有关的终端名, 例如, `‘/dev/tty34’`)。你将看见作为 `ps` 命令的进程、交互 `shell` 的进程以及任何为你运行的后台进程的信息。

如果你正在后台执行一个进程, 那么你能够用 `ps` 命令控制该进程。如果该后台进程的进程标识号是 2150, 那么命令

```
ps 2150
```

将印出进程 2150 的信息。当进程 2150 运行时。你可以反复地使用 `ps` 命令来检查递增的累计执行时间。在某一时刻进程 2150 完成时, 再打入命令

```
ps 2150
```

将印出类似于 “2150: no such process” 这样的信息。

7.6 kill——消灭后台进程

当一个程序正在前台运行时。通常你可以通过敲中断字符 (通常是 `control-d` 或 `DEL`) 来停止它。然而, 你不能通过敲中断字符来停止一个后台进程。UNIX 系统有一个称为 `kill` 的特殊命令, 可以用来消灭你私有的后台进程。只有超级用户可以消灭其它用户的进程。

当你在后台运行一个命令时，shell会自动地印出它的进程标识号。如果你的后台进程号是1234，那么你可以通过打入命令

```
kill 1234
```

来消灭它。如果你已经忘了进程标识(pid)号，那么可以使用ps命令来找到它。仅当进程1234存在，并且它是你的进程，而且该进程未捕捉也未忽略这个kill命令，才会被消灭。如果你正打算消灭的进程完全全是一个普通命令，比如一个正文处理作业，那么kill或许可以进行。而未加证实的、与机器相关的后台进程常常是不可能消灭的。

未捕捉或未忽略普通 kill 信号的进程可以确定无疑地通过发送信号号码 9 来消灭。命令

```
kill -9 1234
```

将发送信号给进程1234。使用“-9”任选项的kill命令，是停止一个进程（命令）的最可靠的途径。

由于进程在运行期间才能消灭它们自己（系统不能直接或间接消灭它们），因此，正在因某个事件而睡眠或等待的进程不能立即死亡。正在等待一个决不会发生的事件的进程是决不会死亡的。进程可以组织安排捕捉这个 kill 信号，或者忽略这个kill信号，或者以某种独特的方式来响应它。这里讨论的要点是：kill 程序只能消灭合理的进程，真正异常的进程还活着。

当你从系统中注销时，在后台运行的所有进程都将停止，除非你已经使用了nohup命令，或者除非该进程已安排了忽略这个信号。当系统的行为使人感到非常离奇或者不可控制时，就需要挂断终端并且从头开始。

7.7 nohup——在退出系统之后运行程序

nohup 命令允许你运行在挂断或者从系统注销以后希望继续运行的命令。对于大型的正文处理命令、非常大的文件排序以及大型程序的重新编译那样的大作业来说，nohup命令是非常有用的。通常，你可以在一个后合作业上用nohup命令。

如果你有一个称为 ‘nroffbook’ 的shell 命令文件，这个命令文件完成在一本书的手稿上的正文处理，那么你可以使用命令

```
nohup sh nroffbook &
```

来引用一个shell。它不会挂断在文件 ‘nroffbook’ 中处理命令的过程。在打入了这条命令之后，你可以立即从系统中注销，而工作将在你不在的情况下继续进行。如果你没有使用nohup，那么当你注销时，这个处理过程就会抹掉。如果你没有别的安排，那么执行nohup命令的输出将送到文件 ‘nohup.out’ 上。

7.8 nice——以低优先权运行进程

nice 命令通常用于降低一个进程的优先权。每当你在进行主要的处理并且想要降低对系统的要求时，你就应该使用**nice**。通常，使用**nice**的程序所花费的时间要比正常优先权上运行的程序花费的时间长得多。**nice**经常用于和后台任务、特别是与用nohup运行的任务连接。

从本质上讲，**nice**减少了分配给一个进程的时间片的大小，但是，该进程在所有情况下总要消耗一些CPU时间。遗憾的是，UNIX系统没有提供在系统没有任何别的事情可做时，安排首要工作的手段。为了尽可能减少对系统中正常处理的干扰，那些需要运行几小时，但是可以等一些时间运行的作业，应该在系统比较空闲时运行。

命令

```
nice sh nroffbook &
```

将引用shell在后台执行命令文件‘nroffbook’中的命令。如果你想注销，同时又让处理继续，那么命令

```
nice nohup sh nroffbook &
```

将以低优先权在后台执行这些事务，并免受挂断的影响。

7.9 time——计算进程的时间

time命令用于计算进程时间。为了比较两种不同方法所花的时间，或者你想知道干某件事花了多长时间，需要计算一个进程的时间。例如，你可能要计算ps命令的时间。命令

```
time ps
```

将计算ps命令的时间。在ps命令完成之后，time命令将印出三个关键的统计数字。第一个统计数字是执行这个命令经过的全部时间，第二个统计数字是执行这个命令所花费的时间，第三个统计数字是系统为这个命令花费的时间。经过的时间精确到秒，而执行时间和系统时间精确到一秒的六十分之一。在非常容易超负荷的系统上，这之间的不同偶而会导致古怪的结果。

统计而得出的时间依负载条件和其它随机影响而变。连续几次计算一个命令的时间多半会产生一致的数据；在一天的几个不同的时刻计算一个命令的时间多半会产生不太一致的数据。系统时间和执行时间的比率表明，在执行该进程时，系统调用相当重要（参见15.6节）。

7.10 man——印出手册的条目

man命令用来印出UNIX系统手册的条目。例如，为了重新产生解释ls命令的用途的UNIX系统手册条目，打入命令

```
man ls
```

手册条目将出现在你的终端上。如果你手头有一本通用的UNIX系统手册，那么man命令很少用得到。

在较老的系统中，man命令需要完成一大堆工作才能重新产生手册条目。如果你不久就需要再次查看其输出，那么你应该用下列命令

```
man ls > ls.man
```

把输出保留在一个文件中。在某些系统中，man的输出在照排机

上印出来。选择使用任选项“-n”或任选项“-Tterm”（参见你的UNIX系统手册而定），使输出在你的终端上看起来更漂亮一些。

7.11 passwd——改变注册口令

passwd 命令用于改变用户的注册口令。某些用户定期（某些系统管理员要求这样）改变他们的口令，以此来维持其安全性。当你打入命令

```
passwd
```

时，系统将提示你打入当前口令。这样做防止了某些人没有得到你允许就改变你的口令。当你已经打入了你原来的口令时，系统将请求你打入你新的口令。一个好的口令既包含大写字母也包含小写字母，它不能出现在目录中，要长于五个或六个字母。如果你打入口令时太马虎了，那么系统可能请你另选一个。当系统对你的选择感到满意时，它将请你再打一次来核实打入的是否正确。在这个过程中，如有一点打入错误，就要你重新来一遍，所以要小心地打入。要注意，在你打入口令时，终端上是不显示的。如果你忘了你的口令，那么系统管理员可以删除老的帐户文件并且临时给你另一个。

7.12 echo——回应命令行自变量

echo命令回应它的自变量。当它的自变量是简单的字，而且希望在终端上印出消息，尤其是它用在shell程序文件中时，echo命令是很有用的。

命令

```
echo Load second reel and THEN strike return
```

将印出消息“Load second reel and THEN strike return”。

echo命令也用于调查shell对自变量的处理能力。你提供给程序的自变量由shell扫描来判断是否用了shell的特殊字符。这些

特殊字符用于控制shell执行的各种替换。例如，shell支持了一个变量系统。字“\$PATH”是对名为“PATH”的shell变量的一次访问，“PATH”制订当前查找路径（shell变量在13.2节讨论，查找路径在13.4节讨论）。通过打入命令

```
echo $PATH
```

你可以得到\$ PATH变量的当前值。在许多系统中，输出将是“:/bin:/usr/bin”。

文件名生成是shell执行的自变量替换的另外一种形式。shell元字符“?”、“*”和“[”为生成文件清单提供手段。当你规定包含shell元字符的自变量时，你可以使用echo命令来观察传递给程序的是什么自变量表。命令

```
echo c*
```

```
echo c?
```

```
echo [cde]*xyz
```

将显示自变量表。第一个命令将显示在当前目录中以字母“c”打头的所有文件的清单，第二个命令将列出所有以字符“c”打头的两个字符的文件清单，第三个命令将列出在当前目录中以“c”、“d”、或“e”打头而以“xyz”正文序列结尾的所有文件的清单。要想把命令的注意力集中在某一组恰当当了名的文件上，文件名生成是一种强有力的技术，当你想准确地知道正在生成什么样的自变量表时，可使用echo命令。

7.13 find——检索一个文件

find命令是为了确定放错地方的文件。find考查一个文件系统统子树，以寻找与一些标准相匹配的文件。例如，命令

```
find . -name checklist -print
```

将从当前目录(名为‘.’)开始的文件系统树中检索名为checklist（因有-name任选项）的文件。如果文件找到了,将印出（因有-print任选项）其路径名。find命令的各种例子放在

书末的简明手册中，find 在 shell 程序中的若干使用情况参见 14.5 节。

针对文件系统所存在的危险或使用中的浪费。系统管理员常用 find 去检索文件系统。find 检索整个大型的子树要花费较长的时间；最好让子树尽可能的小。如果打算检索整个大型文件系统，最好在系统空闲时（比如，凌晨）运行 find。

7.14 mail 和 write——与其它用户通信

mail 程序用于读取送给你的信件或者发信件给计算机的其它用户或别组的用户。如果有人已经把信发给你了，当你注册时，系统将印出消息：“you have mail”。你可以通过打入命令

```
mail
```

来读取你的信件。系统将印出你的信件，一次一个消息。在每个消息的结尾处，mail 程序暂停并且印出一个“？”。你应该用一个表明你想要用 mail 做什么的命令作为回答（细节参见你的手册）。最通常的动作是，把消息保留在一个文件中、删除此消息、或者把消息传给其它用户。

如果你想把信件发送给某个人，那么你必须做的第一件事是寻找那个人的注册名字。注册名可以在 ‘/etc/passwd’ 文件中找到，你可以用 who 命令去查看已注册的用户注册名。如果你想把信件发给 ‘td’ 和 ‘alvy’，那么可以使用命令

```
mail td alvy
```

在你打入了这个命令之后，交互式地打入你的消息。当你打入时，mail 命令收下你的输入行——这时你不能退回几行并校正你先前的输入。当你已经完成了你的消息时，只要敲 EOF 键（control-d）来表示该消息的结束。

使用 mail 程序送信的情况演示如下。要注意，用户打入的部分下边加线，control-d 字符用标记 “^d” 来表示。

```
% mail e mike
Meeting tomorrow on documentation standards,
3pm in O'Flanagans Bar. (Bring Quarters)
^d
%
```

如上所示，对于短消息来说，交互式地打入你的消息是唯一的好办法，因为不可能退回几行来确定错误。如果你有一个长消息，那么你应该用正文编辑程序首先把这个消息放到一个文件里。正文编辑程序允许你推敲和提炼你的消息。通过使用输入重新定向：

```
mail tom alicia <message.fil
```

你可以把已经存放在称为‘message.fil’文件中的消息寄出去。如果mail程序不能确定一个接收者，那么你的消息会保留在一个名称为‘dead.letter’的文件中，以便可以找到正确的用户名并且重新发送该消息而不必重打这个正文。

write 命令用于在两个UNIX系统用户间通过打字机建立通信。电话、内部通信系统或两个罐头盒带一根线，通常都是通信的好途径。由于某些人可能出乎意料地在某日写信给你，因此你应该学会如何使用write命令来作好通信准备。

当某人写信给你时，消息“Message from harry on tty33”（或类似的消息）将出现在你的终端上。你应该通过暂停你正忙于处理的所有事情并且运行命令

```
write harry
```

来进行回答。一旦两个部分都已执行了write命令，两边任何人打入的所有东西都将出现在两边的终端上。因此，最好一次只有一个人打字。开始对话的那个人通常打入少量几行，然后打入表示“报文完，请回复”的单独一个“o”的一行。另一人现在可

自由地用少量几行加以答复。这种对话一直继续到某人打了具有“报文完”，并且退出的“00”的单独一行为止。你可以通过敲入EOF（control-d）字符来终止对话。如果有两个人同时打字，那么其输出将混杂在一起，几乎不可能把它们分清楚。

7.15 stty和tty——终端处理程序

`tty`命令用于印出连接在标准输入上的终端的特别文件名。如果标准输入并未接到一个终端上，那么会印出一个消息。有时终端和计算机之间的通信线断了。为了记录故障，需要知道你的特别文件名以便为你的通信硬件安排修复工作。命令

`tty`

将印出诸如“tty30”那样的消息，它表明你的终端相应于特别文件‘/dev/tty30’。

在shell程序中`tty`命令也用来确定标准输入究竟是不是一个终端。如果标准输入是终端，则出口状态为真，否则为假（参见13.7节）。

设置终端任选项命令（`stty`）让你控制系统去设置终端。因为有许多不同型号的终端，所以`stty`是非常重要的。在某些系统中，有一个程序根据TERM参数的值自动地使系统适应你的终端。在另一些系统中，在每次对话开始时，你应该在文件‘.profile’中用适当的`stty`命令设置合适的方式。

在UNIX系统中，为实现适应特殊型号终端而转换的那部分称为终端处理程序（也称为tty处理程序）本节讨论某些在使用`stty`命令的tty处理程序中可以起作用的任选项。各种完整的讨论收录在本书后面。

打入命令

`stty`

就印出少量关键方式的设置情况。在我的系统中。显示下列信息：

```
speed 300 baud; tabs
erase = #, kill = @, intr = ^?
```

这个消息表明通信速度是300波特，设置了制表符任选项，抹字符是#符，抹行字符是@符，中断字符是DEL（实际用“^?”标记表示）。制表符任选项表明系统相信我有一架能够处理制表符（在某些章节里详述）的终端。抹字符、抹行符和中断字符已在3.4节中讨论了。

抹字符用于抹去前面已打入的字符，抹行符用于抹去整个一行。因为标准的UNIX系统中，抹字符和抹行符都不太好，所以许多人就重新设置它们。通常，control-h字符用作抹字符，而control-u字符用作抹行符。你可以通过打入命令

```
stty erase \^h kill \^u〔译注〕
```

重新指定抹字符和抹行符。一个^后随一个字母，这种记号表明是给stty的控制符；反斜线用于转换^的含义，因为对shell来说，^有特殊含义。通过使用命令

```
stty
```

我们可以验证是否已经转换了。在我的系统中，新的输出如下所示：

```
speed 300 baud; tabs
erase = ^h, kill = ^u, intr = ^?
```

敲一下中断字符就发送一个中断信号给当前正在执行的前台进程。某些程序。如shell和编辑程序可以不理中断。但是其它大

〔译注〕——针对不是你当前的终端使用这条命令时，还应在后面加上重新定向符“<”和相应的特别文件名，例如，</dev/tty30。

多数程序，如 `cat` 和 `grep`，当它们接收一个中断时就暂停。许多人宁愿用 `control-c` 而不用 `DEL` 键作为中断键。命令

```
stty intr \ ^c
```

将把中断功能赋给 `control-c` 键。你可以使用 `stty` 命令来验证这次作的重新设置。

制表符的处理因终端不同而变。某些终端识别并且扩展制表符，但有的终端却不管它。命令

```
stty -tabs
```

通知系统，你正在使用一架不知道怎样扩展制表符的终端。UNIX 系统终端处理程序会把制表符扩展成若干空格，使它在你的终端上正确地出现。命令

```
stty tabs
```

通知系统，你的终端可以扩展制表符，这样，制表符将直接通过终端处理程序。你应该在有制表符的终端上使用这种设置，因为当遇到一个制表符时，它可以稍稍加快输出速度。

通过不产生小写字母的终端来使用 UNIX 系统，虽然很困难但是还是有可能的。命令

```
stty lcase
```

通知系统，你正在使用一架只支持大写字母的终端。当你在使用一架大写字母终端时，所有的输入将自动地从大写字母翻译成小写字母，而所有的输出将自动地从小写字母翻译成大写字母。在大写方式下，如果你打入了一个反斜线，后随一个字母，那么这个字母就不翻译成小写字母。例如，当你在大写方式下，输入“`\JOHN DOE`”就翻译成“`John Doe`”。命令

```
stty -lcase
```

将返回到大写和小写都可接收的正常方式。如果在注册时按下了终端的 `shift` 键，那么你可能不小心进入了只有大写的方式中。在注册过程中，系统试图确定终端的型号并且给出适当的设置。如果你都以大写字母打入你的注册名，那么系统就假定你在使用大

写终端，此时你就必须用 `stty` 命令重新转回到正常方式。

你可以用 `stty` 改变终端和计算机之间通信的波特率（波特率是终端和计算机间字符传送速度的一种度量。作为一种粗略地估计，波特率被10除就是每秒可以传送的字符数）。一旦你告诉了UNIX系统一种新的波特率，就必须把你的终端的速率改得与之匹配。你可以用命令

```
stty 1200
```

把通信速率改成1200波特。

`stty` 命令调节当前作为标准输入终端的设置。你可以利用这种功能来改变一个不是你当前正在交互使用的终端的设置。例如，如果你想显示 `tty33` 的设置，你可以用命令

```
stty </dev/tty33
```

（在大多数系统中，为了打开 `/dev/tty33` 来读，你需要超级用户特权）。类似地，你可以使用命令

```
stty 9600 </dev/tty33
```

在 `tty33` 上设置波特率为9600。对于控制诸如只接收打印机那样的特殊设备的通信通道来说，设置波特率这种方法是很有用的。用户很难去干涉已经受注册进程管制的通信通道（这可能是注册的通道）。

`stty` 包含了许多你可以用来让系统适应个别终端的其它任选项。为了使用 `stty` 命令的很多任选项，你必须知道许多有关终端、计算机和串行通信的知识。你应找出你需要用什么样的任选项。并在你的 `‘.profile’` 文件中放置适用的 `stty` 命令。

7.16 du——查看磁盘使用情况

有时，你想看看你的文件已经占用了多少磁盘空间。打入命令 `du`

就会得到一张表，它登记了在当前子树下每个目录所用的磁盘空间块数（当前子树由当前目录下的所有文件，或由任意一个子目录下的所有文件等等组成）。你也可以指定考察哪一个子树。
命令

```
du /usr/sys/src
```

将印出与‘/usr/sys/src’目录相连的子树中磁盘用途的一览表。

UNIX系统的一个准则是“对于用户的磁盘存储需要可以尽量满足直到填满可用的空间为止”。为了缩减你的存储负担，并且保持系统的空闲空间，必须定期删除老的文件。当用户正在清理他们的目录，以便压缩消耗很多空间的目录时，他们时常要用du命令。

df（磁盘空闲空间）命令可用于查看在一个特定的存储卷上还有多少空闲空间存在（见18.6节）。系统管理员时常定期地运行df以便掌握空闲空间的情况。当空闲空间减少到某种程度时，大多数系统管理员要请用户删去他们的目录。

7.17 od——卸出（dump）文件

有时，你想确切地知道在一个文件中包含了什么样的二进制代码。八进制卸出(od)程序用于产生一个文件的八进制、十进制、ASCII和十六进制格式的卸出信息。各种格式可以合在一起也可以分开来产生。

术语“卸出”许多年以前就已开始使用了，那时的程序调试通常是在程序出错时，将存储器中所有值打印输出来进行的。由于信息量很大并且程序员的识别工作又令人很不满意，因此，打印输出就称为“卸出”虽然现在仍用“卸出”这个词，但是许多调试手段已经比较灵活了。今天许多程序在失败时，可借助于更容易解释信息的程序来帮助考察“卸出”信息。一旦UNIX系统程序意外地失灵了，它就把信息卸出到程序当前目录中名为

‘core’的文件中，然后产生一个消息：“core dumped”。

八进制卸出程序常用于检索嵌入在正文文件中的控制字符。例如，你想知道某个文件中是否包含了制表符。如果你打算在屏幕上 cat 这个文件，那么终端处理程序或者你的终端将自动地把制表符扩展成数量合适的空格。然而，如果你用了od程序去卸出这个文件，那么你可以考察输出中表示制表符的记号“\t”。当以ASCII格式卸出时，可印刷字符正常地显示了出来，不可印刷的控制字符除了下面列出的少量很标准的字符以外，都以八进制形式印了出来：

退格	\b
制表	\t
换行	\n
回车	\r
空	\0
换页	\f

命令

```
od -c chapt
```

将以ASCII字符（因为有任选项“-c”）格式卸出文件‘chapt’。od也可象卸出普通文件一样卸出特别文件或目录文件。

通过打入命令

```
od /dev/rp0
```

超级用户用od来考查（以8为底）存放在磁盘上的信息。od也允许你通过指定一个位移量从中间某个地方开始考察一个文件。使用命令

```
od /dev/rp0 +1024
```

可以从第1024字节开始考察该磁盘。自变量“+1024”表明你应该跳过文件开头的1024个字节再开始卸出。通过打入命令

```
od /dev/rp0 +2b
```

也可以以块为单位（1块是512个字节）来指定同样的位移量。

第八章 正文文件实用程序

许多用户是因为UNIX系统有出色的正文处理功能而使用它的。许多程序员喜欢UNIX系统并不奇怪，因为程序设计的很多工作是处理正文。同样，秘书、科学家和商人也已经发现，UNIX系统的正文处理功能可以提高他们的工作效率。

自然，UNIX系统包含了在终端上打印文件或在打印机上打印文件的程序。UNIX系统还包含了排序文件，为正文模式检索整个文件，统计文件的行数、字符数、字数，以及在资料文件中查找拼写错等等的程序。这些程序都在本章中介绍。

UNIX系统还包含了各种十分复杂的用于正文文件工作的程序。本章第一节是UNIX系统复杂的正文处理程序的一个指南。所有这些程序都在本书的其它地方涉及到。

8.1 正文实用程序

正文文件常常通过正文编辑程序来创建。正文编辑程序允许用户在一个文件中打入并且修改正文。许多UNIX系统有几个正文编辑程序，在第五章中已经介绍了许多与UNIX系统标准的正文编辑程序相类似的正文编辑程序。标准编辑程序的某些高级特性在第十章介绍。

UNIX系统包含了可以加工正文文件格式的程序。格式加工包括下列功能：对齐边缘、插入标题和脚注、调整空白、排齐表格数据的列、并且处理在数学公式中的所有特殊字符。UNIX系统中用于格式加工的nroff、troff、eqn和tbl在第十一章中介绍。

UNIX系统也包含了这样的程序，它可以帮助你编写识别命令语言和语法的程序。如果你正在编写一个正文格式加工程序、

一种程序设计语言翻译程序或者是完成正文处理的各种动作的程序，那么，这些识别程序是很有用的。UNIX 系统中帮助你编写识别程序的那种程序称为 lex 和 yacc，它们在第十七章中介绍。

正文文件常常来源于其它正文文件。一个正文文件也可以有若干版本。为了帮助你维护正文文件系统，UNIX 系统包含了 make 程序。它保持一组文件相互依赖关系的记录，而源代码控制系统 (SCCS) 保持一个文件不同版本的记录。make 和 SCCS 在第十二章中介绍。

8.2 cat——印出文件

串接程序 (cat) 是最通用的正文处理程序之一。cat 程序的标准用途之一是在你的终端上打印文件。命令

```
cat /etc/motd
```

将在你的终端上打印文件 ‘/etc/motd’。由于 cat 是 concatenate (串接) 的缩略，所以 cat 就是把文件串接在一起。如果你打入了命令

```
cat /etc/greeting /etc/motd
```

那么 ‘greeting’ 文件和日志消息文件 motd 将被组合 (串接) 在一起，并且在你的终端上打印出来。用 cat 串接文件可以按下面的方式进行：

```
cat chapt1 chapt2 chapt3 chapt4 chapt5 > book
```

这个命令用输出重新定向，把一本书的第五章文件串接成称为 ‘book’ 的单独一个文件。这个命令可以象

```
cat chapt [12345] > book
```

那样以更简洁的方式打入，因为文件名生成按字母顺序出现 (正如你期望的，数字 1, 2, ... 按字母顺序排列)。如果在目录中只有五个编了号的章，这个命令也可以象

```
cat chapt? > book
```

那样打入，或者象

```
cat chapt * >book
```

那样打入。然而，你应该注意，命令

```
cat chapt3 chapt1 >newchapters
```

与命令

```
cat chapt[31] >newchapters
```

并不等价。第二个命令取决于为 cat 程序生成自变量表的shell的文件名生成过程。文件名生成总是产生一个按字母顺序排列的表，所以当你串接文件时，如果你依赖于文件名生成，那么你要记住按字母顺序排列。你可以用echo命令查看其中的区别。命令

```
echo chapt3 chapt1
```

将产生输出“chapt3 chapt1”，而命令

```
echo chapt [31]
```

将产生输出“chapt1 chapt3”。

cat命令的另一个用途是建立空文件。用命令

```
cat /dev/null >empty
```

将建立一个名为‘empty’的空文件。UNIX系统的空设备是名为‘/dev/null’的特别文件。如果把信息定向输出到该空设备上，那么信息就被抛弃了。如果你从该空设备上读入，那么，你立即就会碰到文件尾。因此，执行空设备的 cat 命令将会产生一个长度为 0 的输出（在上述命令中是定向到名为‘empty’的文件去）。

建立空文件的另一条途径是使用命令

```
>naught
```

它建立名为‘naught’的空文件。当你打入一条仅仅由输出重新定向的命令时，shell将建立指定的文件。

cat 程序的另一种用途是把少量几行正文放到一个文件中而不必打扰正文编辑程序。如果你执行一个没有自变量的cat，那么，它从标准输入中读取信息直至碰到文件尾为止。这样，命令

```
cat >quick
```

将使 cat 读取你在终端上打入的任何东西，并且把它们放到文件

‘quick’中。由于cat不是编辑程序，因此你不能退回几行来找出错误，但是对于只有一、两行输入来说，这个命令是很有用的。当你敲control-d时，cat将接收文件尾标记、关闭输出文件并且退出。然后，shell将提示你打入另外的命令。

如果在命令行中没有明确地提到输入文件，那么许多UNIX系统正文处理程序将从标准输入中读取它们的输入。当故意这样用时，这是一种强有力的技术，但是当你不小心漏掉输入文件时，程序一直在等待从终端上输入，而你却老是坐在那儿等待程序完成它的杂务工作！

8.3 pr——给文件加标题并且进行格式加工

pr命令用于为文件分页并且加标题。如果你用pr来打印一个文件，那么这个文件被分拆成若干页，每页加以编号并且加标题，在每页的上边和底下留下若干空白行。pr命令最普通的用途是为行式打印机准备正文文件。

pr也可以用来产生输出的若干列，通过用制表符替换空白来压缩文件、把制表符扩展成空白、统计文件中的行数、或者执行其它简单的重新格式加工任务。命令

```
ls | pr -5 -t
```

将把ls命令的输出用管道连到pr命令的输入上去。pr将产生五列输出（因为有自变量-5），但没有标题，也不进行分页（因为自变量是-t）。成列的ls输出将使更多的文件在一个屏幕上显示出来。

pr的标准用法是：如下列命令

```
pr chapt1 > chapt1.pr
```

那样给一个文件分页并且加标题。

8.4 lpr——打印文件

lpr命令在系统行式打印机上打印出文件。备有若干种行式打印机的系统，通常提供若干种lpr命令。在我的系统上，

lpr 命令是用于主行印机的，而vpr, dpr和npr是用于其它行式打印机的。

在若干用户之间，同时共享一台行印机是不可能的。在请求打印期间，该行印机只能单独分配给一个用户。lpr 命令的主要作用就是协调对于行式打印机的请求。如果当你打入 lpr 命令时，该行印机正忙着，那么你的文件将放在一个队列上，当该行印机空闲时再打印你的文件。当你的打印文件一放到队列中，lpr就把控制返回给你。

在打印文件的前面和后面，lpr 命令通常都要印出一个标识页，但是文件内容不作修改。lpr 并不在页的顶部和底部插入空白行，也不编页号，也不执行pr命令的任何其它动作。如果你想为文件内容编页或加标题，那么你应该先用pr命令。

假定你需要若干目录的打印清单。一种方法是打入下列四条命令：

```
ls -l /bin /usr/bin > dirfile
pr dirfile > dirfile.pr
lpr dirfile.pr
rm dirfile dirfile.pr
```

一个更简单的使用管道的方法是命令

```
ls -l /bin /usr/bin | pr | lpr
```

当然，lpr 命令最普通的用途是象下例那样打印一个或一组文件：

```
lpr chapt1.pr chapt2.pr chapt3.pr
```

图8.1 是cat, pr和lpr三个实用程序的示意图。

普通磁盘文件'fleas'

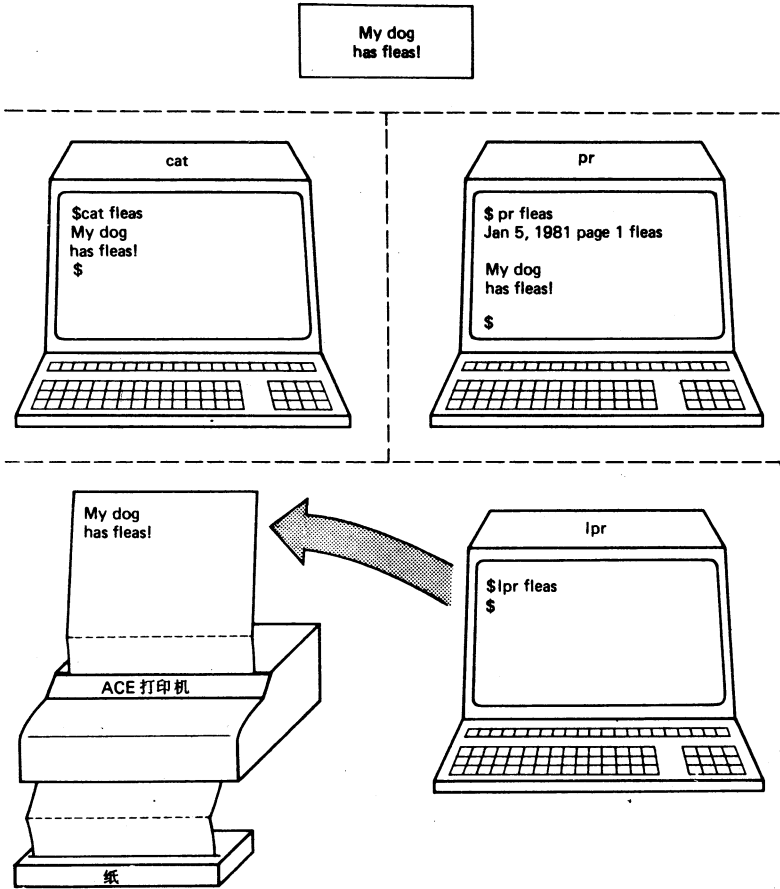


图8.1 cat, pr和lpr

8.5 wc——统计行数、字数和字符数

字数计 (wc) 程序将清点在一个正文文件中的字符数、字数和行数。命令


```
wc chapt?
```

将印出在当前目录下存放每一个 chapt 的文件的字符数、字数和行数。标志自变量可以用于控制字计数程序,让它只统计字符数,或只统计字数,或只统计行数。

8.6 diff——比较文件

diff 程序用于显示在两个文件中哪些行不同。diff 的输出类似于编辑程序命令的输出,后面跟随来自于两个文件的受影响的若干行。来自于第一个文件的行,前面有一个“<”,而来自于第二个文件的行,前面有一个“>”。

diff命令产生三类伪编辑程序命令行:

```
n1 a n3, n4
n1, n2 c n3, n4
n1, n2 d n3
```

行号n1和n2表示第一个文件中的行,而行号n3和n4表示第二个文件中的行。第一条伪编辑程序命令表明,第二个文件包含了(从n3到n4)那些行,它们并不在第一个文件的第n1行后出现。第二条命令表明,在第一个文件中的第n1到第n2行不同于第二个文件中的第n3行到第n4行。第三个命令表明第一个文件中包含的(第n1到n2)行不在第二个文件的第n3行后出现。

例如,如果文件‘ar1note’和‘ar1note2’,除了‘ar1note2’第10行后包含了行“Susan -586 -1234”外,其余两者都一样,那么命令

```
diff ar1note ar1note2
```

将产生下列输出:

```
10 a 11
> Susan -586 -1234
```

输出表明,如果行“Susan-586-1234”(‘arlinote2’的第11行)附加到‘arlinote’的第10行后,两个文件就相同。

8.7 sort——重排文件

`sort` 命令用来排序并且/或者合并正文文件。`sort` 根据你的命令行说明重新排列文件中的行。这些说明很繁杂,因而就不在这里一一讨论了。除非你另作了安排,否则排了序的输出将出现在你的终端上。自然,如果你想要排了序的永久副本,那么可以把排了序的输出重新定向到一个文件中。

每一个输入行可以包含若干字段。字段可以用一种字段分隔符定界。分隔符通常是制表符或空格符,但是可以指定某些别的字符作分隔符。在`sort`命令中,用于确定该文件排列次序的那个输入行部分,称为排序码。排序码可以是一个或多个字段,或者是字段的一部分。或者是整个行。

举一个例子,我们要排序一个文件,此文件包含一批人的电话号码和姓名的第一个字母。为了排序一个文件,你必须有一个具有一种正规结构的文件,并且你必须知道这种结构的细节。在我们举的例子中,假定文件的每一行的次序是:每个人的姓名的第一个字母、一个制表符和那个人的电话号码。为了简化,让我们考虑一个假想的名为‘telnos’的文件,它包含下列三行:

```
kc    362-4993
gmk   245 3209
arm   333-3903
```

命令

```
sort +0 -1 telnos > namesorted
```

将根据人的姓名的第一个字母排序文件‘telnos’。自变量“+0”和“-1”通知`sort`程序,我们想限定排序码的第一字段;“+0”表明排序码从行的起始开始,而“-1”表明该排序码在第一字段尾结束。如果我们想要根据电话号码排序文件,那么我们可以打

入命令

```
sort +1 telnos > numbersorted
```

“+ 1”自变量通知sort程序，我们想限定排序码从第一个字段以后开始。排序之后，(在我们的例子中)称为‘namesorted’的文件将包含下列排列：

```
arm 333-3903
gmk 245-3209
kc 362-4993
```

而称为‘numbersorted’的文件将包含

```
gmk 245-3209
arm 333-3903
kc 363-4993
```

当然，可以完成更复杂的排序工作（参见简明手册中的例子）。

8.8 grep——在文件中寻找正文模式

grep程序（和相关的 **fgrep** 和 **egrep**）用作在文件中寻找正文模式。只要正文模式在一行上找到，文件的该行就在标准输出上印出。你可以把grep程序想象成这样一种工具：它根据一种正文匹配标准来完成整个文件的水平切削。字grep是由短语“global regular expression print（全局的正则表达式打印）”派生而来的。

grep命令的第一个自变量是正文模式，后面的自变量指明应该被考察的文件。grep的正文模式可以使用UNIX系统标准的正文编辑程序的大多数正则表达式语法。对于**eqrep**来说，正文模式可以更为复杂些。而对于**fgrep**来说，正文模式限于固定的字符串。可以预料，**fgrep**（fast grep——快速的grep）要比**grep**快，而**grep**要比**egrep**（extended grep——扩展的grep）快。对于大多数普通使用来说，grep已足够强有力和快速的了。

假定你想要根目录的子目录的长格式清单。那么，打入命令

```
ls -l /
```

就会产生所要的清单，这个清单与所有包含在根目录中的普通文件纠缠在一起。grep命令可以用来过滤掉所有不需要的信息。如果没有文件作为自变量提供给grep，那么grep就从标准输入读入。因此，grep可以象下面那样用在管道线中：

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

这条命令把ls命令的输出通过管道连到grep命令去。grep的正文模式用引号括了起来，因为它包含了^，这是对于shell另有含义的字符。编辑程序正则表达式“^d”匹配所有以字母“d”开始的行。这样，由于在长列表中的目录行是以“d”开始的，而普通文件是以“-”开始的，因此只有目录被列了出来。当在我的系统上运行这个命令时，就印出了下列行：

drwxrwxr-x	4	bin	3136	Sep 17 11:28	bin
drwxr-xr-x	9	root	3648	Sep 16 17:24	dev
drwxrwx--x	4	root	2496	Sep 17 18:40	etc
drwxrwxr-x	5	bin	752	Jul 1 11:24	lib
drwxrwxr-x	6	root	160	Aug 26 13:37	mnt
drwxr-x--x	12	mel	528	Sep 2 15:48	source
drwxrwxrwx	2	root	896	Sep 17 19:27	tmp
drwxrwx--x	26	root	416	Jul 27 12:57	usr

要注意，所有行都以字母“d”打头。

grep命令的标准用法是找出资料中出现的某些字的全体。当我本想买打“the”时，由于坏习惯常打成了“teh，命令

```
grep teh chapt*
```

将在当前目录下为正文模式“teh”检索所有chapt文件。当然，我可以用编辑程序完成检索任务，但是用grep去标识那些包含这个错误的文件比用编辑程序确定这个错误更为容易。

8.9 cut和paste——重排文件的列

cut 和 paste 程序用于进行文件的垂直分割。cut 和 paste 对于包含表格数据的文件来说，是很有用的。cut 程序用于从文件中裁下一个垂直片段，而 paste 程序把若干垂直片段合并到一个文件中去。cut 和 paste 都处理输入文件并且在标准输出（终端）上产生它们的输出。通常，你会把它们的输出重新定向到一个文件上去的（要注意，cut 和 paste 并不象在本章讨论的其它程序那样用得如此广泛）。

为了使用 cut 和 paste，你必须理解文件的列（字段）的构造方法。虽然可以使用其它字符作字段分隔符，但是最容易的字段分隔符是制表符。

假定我们需要把以前所介绍的‘telnos’文件中的姓名第一个字母和电话号码分开。正如你还记得的。‘telnos’文件包含下列三行：

```
kc    362-4993
gmk   245-3209
arm   333-3903
```

使用命令

```
cut -f1 telnos > initials
```

将把第一个字段（姓名第一个字母）放到一个名为‘initials’的文件中，而命令

```
cut -f2 telnos > numbers
```

将把第二个字段（电话号码）放到一个名为‘numbers’的文件中。‘initials’文件包含

```
kc
gmk
arm
```

而‘numbers’文件将包含

362-4993

245-3209

333-3903

命令

```
paste numbers initials > newtelnos
```

重新把 ‘initials’ 文件和 ‘numbers’ 文件组合成一个新的电话号码文件, 这个文件我们将称之为 ‘newtelnos’。在 ‘newtelnos’ 中的每一行, 电话号码可以在姓名第一个字母的前面。paste 将自动地用一个制表符把两列分开。这样, 文件 ‘newtelnos’ 将包含下列三行信息:

```
362-4993 kc
```

```
245-3209 gmk
```

```
333-3903 arm
```

8.10 spell——寻找拼写错误

UNIX 系统 spell 程序检查正文文件以寻找可能的拼写错误。spell 使用含有常用字的字典。针对输入正文中的每一个字都要在这个字典里查阅一番。spell 极少受前缀、后缀及折转 (指一个字在一行写不下时, 一部分折转到下一行去——译者注) 的欺骗, 并且 spell 能够忽略 nroff/troff 格式加工命令 (参见第十一章)。

如果 “frequent” 在字典中, 那么 spell 将接受 “frequents”、 “frequently”、 “frequenting” 以及别的变形。因为某些字不遵循加前缀和后缀的正常规则, 所以另外有一个字典列出所有这些例外的字。非规则变化的字必须和这部字典 (称为 stop list) 的字完全一致。

在资料中有许许多多 spell 不能发现的可能的拼写错误。spell 也可能不承认许多正确的字, 因为它的字典不可能包含每一个字, 尤其在技术科目中。spell 也很可能反对专有名字、地点等等。

spell是宝贵的校对工具，但是它并不能代替细心的校对。

spell的正常使用是
spell chapt1.doc

用来核实文件 ‘chapt1 . doc’ 中的拼写。有疑问的字将在标准输出上打印出来。在一份中等规模的资料中，spell将会找出几千个问题，所以通常你应该用下列命令把这些字收集在一个文件中：

```
spell chapt1.doc > chapt1.errwds
```

如果没有提到文件，那么spell就从标准输入中读。对于会话式地核对拼写，或者在一个管道线中使用spell来说，这都是有用的。

8.11 crypt——为文件加密

crypt 程序读取标准输入、将它加密并且写到标准输出中。如果你想确保你的文件绝对可靠，那么你可以给它们加密。加密比使用文件保护机制更安全，因为超级用户可以存取任何文件，但是他不能给加了密码的文件解密。只有当你知道了口令或者精通密码破译方法或者你有大量的计算机时间专供解密时，才能把加密后的文件解密。命令

```
crypt xyZZy321 <chaptn.doc>chaptn.cry
```

将用口令“xyZZy321 ”为文件 ‘chaptn.doc’ 加密。其结果将放在文件 ‘chaptn.cry’ 中。类似地，你可以打入命令

```
crypt<chaptn.doc>chaptn.cry
```

在这种情况下，crypt 将在你的终端上印出一个信息，请求你打入口令。在你用口令回答时，crypt 将不给回应。程序需要从控制终端而不是从标准输入上读取输入，这是一种较为罕见的情况。

原先的文件可以用同样的口令

```
crypt xyZZy321 <chaptn.cry>chaptn.new
```

恢复。文件 ‘chaptn.new’ 应该与文件 ‘chaptn.doc’ 相同。

8.12 tee——复制输出

tee 程序读取标准输入并且把它转送到标准输出以及一个或多个指定的文件中。这与管工用 T 形管道接头，把一个管道分成两个管道的情况相类似。当你想把一个程序的输出传送到一个文件上并且还想在你的终端上看到它时，通常就用 **tee**。命令

```
spell mybook.n | tee errwords
```

将把来自文件‘mybook.n’中的疑难字收集到文件‘errwords’中并且在终端上显示出来。

tee 也可以用来收集管道线中的中间结果。如果你想通过打印一个五列加了标题的资料来展示一个目录的内容，那么可以使用命令

```
ls | pr -5 | lpr
```

通过在管道线中插入 **tee** 命令，你可能保留两个中间文件。

```
ls | tee lsfile | pr -5 | tee lsprfile | lpr
```

文件‘lsfile’将包含 **ls** 命令原先的输出，而文件‘lsprfile’将包含五列加了标题的 **ls** 输出版本。

8.13 tail——印出文件尾

tail 程序用于印出一个文件的尾。这就使你可以看到一个大型文件的结尾内容，而不必使人乏味地坐等显示整个文件。命令

```
tail book.new
```

将印出文件‘book.new’的最后少数行。**tail** 的各种任选项允许你控制你要打印多少行文件尾。命令

```
tail -132 book.new
```

将印出文件‘book.new’的最后132行，而命令

```
tail +66 chapt.1
```

将印出文件‘chapt.1’中除去前面开始的66行外的所有行。数字前面有+号，表明是从文件开头算起的行的位移，而数字前的

“-”号，表明是从文件尾算起的位移。

tail也可以用在管道线中。命令

```
ls /usr/bin | tail - 20
```

将显示 ‘/usr/bin’ 目录中最后20个(按字母顺序算的)文件。

第九章 管理文件

UNIX 操作系统是用于管理信息的工具。本章介绍UNIX 系统中管理文件的实用程序。所有这些程序完成非常简单的、类似于大扫除期间完成的那样普通的功能。当你与UNIX 系统对话时，你必须定期删除老的信息，为新的信息腾出空间，为某些文件调整文件存取权限，并且偶而从其它用户处获得文件。

也有某些信息自动管理的领域，但在大多数情况下，你必须提供管理功能。在UNIX 系统中，信息的单位是普通文件。大多数操作系统，包括UNIX 系统在内，包含下列功能：建立普通文件、把普通文件从一个地方移到另一个地方、重新命名文件、构造文件副本以及删除文件的程序等等。必要时，你就要在一组文件上实现这些功能。

在一个多用户操作系统中，文件分别由各个人占有，有必要维持一个特权系统，以便占有者能够保护它们的文件不受其它用户进行不希望有的存取。因此，UNIX 系统有控制文件的存取权限以及改变文件所有关系的程序。UNIX 文件系统要比单一一组文件还要多。在UNIX 系统中，文件组合成目录，目录再排列成一个逻辑层次结构。这种层次结构使得组织和安排你的那组文件更为容易。因为这个原因，UNIX 系统中有若干程序来维护这个目录系统。信息管理任务的一个主要方面是取决你打算怎样组织目录。一组良好的目录组织会使UNIX 系统的使用容易得多，尤其是如果你把它用于若干不同的场合的话。

9.1 rm——删除文件

删除 (rm) 命令可以让你删除普通文件 (rmdir 命令用于删

除目录。参见9.5节)。为了删除一个文件，你必须在包含这个文件的目录中有写许可，但是对于这个文件本身，你既不需要读许可，也不需要写许可。如果该文件是写保护的，那么如果你真正想删除这个文件，系统就会追问你。

命令

```
rm myfile
```

将删去文件 ‘myfile’。当你使用rm时要小心，因为删去的文件就真正丢了。恢复一个已删去文件的唯一途径是请系统管理员从最近的后援系统中恢复该文件的副本。一有怀疑，就应该停止文件的删除工作。一种良好的习惯是不要让许多文件用类似的名字，因为键盘打入时的错误或短暂的精神疏忽很容易删错文件。

命令

```
rm *
```

将会删去在当前目录下的所有普通文件。除非你想整个儿地清除一个目录，否则不要使用这条命令。

删除命令包含两个非常有用的任选项。如果出现“-i”标志，那么删除程序会以对话方式询问你是否真正打算删去每一个提及的文件。命令

```
rm -i chapt *
```

将请你保留或删除在当前目录下，其名字以 ‘chapt’打头的每一个文件。每一个文件（在这里是以名字 ‘chapt’打头的文件）的名字将印出来，后随一个？号。如果你用“y”或“yes”回答，那么这个文件就被删去了。任何其它的回答都会使文件得以保留。

如果在你的目录中有不可打印名字的文件，这种会话式的删除法尤其有用。任何包含控制字符的名字不太容易或者不可能打印出来。由于错误的程序或者由于其它瞬时的偶而会产生古怪的、说不出名字的文件。如果你想删除一个具有不可打印名字的文件，那么就打入命令

```
rm -i *
```

除非遇到一个具有不可打印名字的文件，否则这个命令不作任何回答。

另一个非常有用的任选项是“- r”，这个任选项用于删除一个目录及其所有的内容、以及在这个目录子树中的所有文件和目录。打入命令

```
rm -r bookdir
```

就删除目录 ‘bookdir’ 以及它的所有内容、子目录等等。这个递归任选项清除在所提及处的整棵树。自然，当使用任选项“- r”时要特别小心。作为一般规律，用 rm 删除普通文件，然后用 rmdir 删除目录，这是比较安全的。

9.2 mv, cp 和 ln——移动和复制

移动 (mv) 命令、复制 (cp) 命令和联结 (ln) 命令可移动和复制文件。移动命令把一个文件从一个地方移到另一个地方。如果两个地方都在同一个文件系统中，那么这种移动本质上是一种重新命名操作，因为在该文件中的数据并不需要重新定位。如果两个地方在不同文件系统中，那么在该文件中的数据必须从一个设备重新定位到另一个设备。命令

```
mv chapt3 chapt3.save
```

将重新命名一个普通文件。老名字是 ‘chapt3’ 而新名字是 ‘chapt3.save’。在这个命令执行后，就不再存在名为 ‘chapt3’ 的文件了。第一个文件 (‘chapt3’) 称为源文件，而第二个文件 (‘chapt3.save’) 称为目标或目的文件。

一般说来，你不能用 mv 来重新命名目录。但有一种例外，即源目录和目的目录有同样的父目录。如果 ‘mydir’ 是一个目录，那么命令

```
mv mydir mynewdir
```

是合法的，因为源目录文件 ‘mydir’ 和目的目录文件 ‘mynewdir’ 有同样的父目录。命令

```
mv mydir ../mynewdir
```

是不合法的，因为源目录和目标目录没有同一个父目录。

如果源文件是一个普通文件而目标文件是一个目录文件，那么mv可以把源文件移到目标目录中去。假设‘wkfile’是一个普通文件，而‘mydir’是当前目录的一个子目录，那么命令

```
mv wkfile mydir
```

将把普通文件‘wkfile’移到‘mydir’目录中去。打入命令

```
ls mydir/wkfile
```

你可以验证这个操作。我们可以进一步想到，如果目标文件是一个目录，那么可以有若干源文件。命令

```
mv wkfile1 wkfile2 wkfile3 mydir
```

将所有三个源文件移到‘mydir’目录中去。

复制命令构造一个文件的副本。mv和cp之间的区别是mv要清除源文件；而cp操作后，源文件和目的文件皆存在。命令

```
cp chapt4 chapt4.archive
```

将在名叫‘chapt4.archive’的文件中构造一个‘chapt4’文件的副本。文件‘chapt4’并不由于这个操作而改变。如果源文件是一个目录，则不能用复制操作。

如果cp操作的目标是一个目录，那么复制操作将把源文件复制到该目录中。在操作之前目标目录必须已经存在。有一个名叫‘bkpdir’的目录，其中保存了许多重要文件的副本。命令

```
cp chapt4 bkpdir
```

将把名叫‘chapt4’的文件的副本放到目录‘bkpdir’中。

象以前那样，当目标文件是一个已存的目录时，你也可以有若干源文件。命令

```
cp chapt* bkpdir
```

把这些章中的所有文件都放到‘bkpdir’目录中。

联结命令用于建立文件的别名。在UNIX系统中，一个文件可以有若干不同的名字。

当你使用复制命令时，将产生该文件的两个副本。改变该文件的一个副本并不影响另一个副本。当你使用联结命令时，就建立了引用这个文件的一个新名字。联结命令并不产生该文件中实际数据的新副本。

命令

```
In chapt8 genfilchapt
```

将为文件 ‘chapt8’ 建立第二个名字。新的第二个名字为 ‘genfilchapt’。两个名字都是有效的，用两个名字中任意一个都可以引用这个文件。有若干理由需要一个文件有两个名字。例如，对于一组文件我们想要一种命名系统，它可以反映该章编号（‘chapt8’）；还要另一种系统，它可以反映该章内容（‘genfilchapt’）。

通过使用ls命令你可以发现一个文件所有的联结数目。命令

```
ls -i chapt8 genfilchapt
```

将展示这两个名字都有联结数目为2。通过在ls命令中使用索引节点任选项（“-i”）；你可以看出这两个名字实际上仅仅联结到一个文件上（索引节点是系统用来定义单独一个文件特征的一种数据结构。参见9.6节）。如果两个名字引用同一个文件，那么这两个名字都将与同一个索引节点号相关。命令

```
ls -i chapt8 genfilchapt
```

将揭示这两个名字的索引节点号。如果就象目前这样，文件实际上只联结到一个文件上，那么索引节点号是相同的。如果名字表示不同的文件，那么索引节点号也将不同。

在目录层次结构中，别名（联结）也很重要。名字 ‘...’ 总是表示父目录。当一个目录创建时，系统把名字 ‘...’ 联结到父目录去，并且把名字 ‘.’ 联结到当前目录去。整个目录层次结构就是由目录文件间的联结进行维持的。你不能用ln命令来改变这种把文件系统捆扎在一起的联结。

9.3 chmod, chown和chgrp——改变文件方式

改变方式 (chmod) 命令、改变所有者 (chown) 命令和改变组名 (chgrp) 命令用于控制文件的存取权限。这些命令能灵活和有保护地存取文件、很好地协调文件系统，是UNIX系统的长处之一。所有这些命令只能由文件的所有者或由超级用户来操作。

在一个文件上可以实行三种操作——读、写和执行。每个文件有三层特权，即所有者特权、同组用户特权和其它用户特权。每一层特权，在三种基本操作的任一种中，都可以被允许或者被禁止（调整id方式和sticky方式也可以允许或禁止，但是这两种方式不在这里讨论，因为它们是系统程序设计属性）。

允许你自己决定属你所有的文件的方式。例如，如果你想构造一个除你本人以外任何人都不可读写的文件，那么你可以打入命令

```
chmod go-rw file
```

字“go-rw”是该文件的新方式。字母“g”和“o”表明你想控制同组用户和其它用户许可。字符“-”表明你想禁止特权，而字母“r”和“w”表明读、写特权。

一个符号方式控制字（在本例中是“go-rw”）由三部分组成：谁、操作符、许可。在本例中，“谁”这部分是“go”表明同组和其它用户；“操作符”这部分是“-”，表明不许可；而“许可部分”是“rw”表明读和写。构造符号方式控制的字符汇总于下表中（见下一页）。

命令

```
chmod a=rw myfile
```

将使得myfile可由所有者、同组用户以及其它用户读和写。

命令

```
chmod g+x newdoo
```

	谁	操作符	许 可
u	用户(所有者)	-	r 读
g	同组用户	+	w 写
o	其它用户	=	x 执行
a	全部 (ugo)		s 调整用户(或组)id方式
			t 保留正文(sticky)方式
			u 用户的目前许可
			g 同组用户的目前许可
			o 其它用户的目前许可

将增加同组用户对文件 ‘newdoo’ 的执行许可。命令

```
chmod o-rwx newdoo
```

将使 ‘newdoo’ 不能由其它用户访问。

用命令 `chown` 和 `chgrp` 来改变文件的所有者和同组用户。当一个用户继承另一个用户的文件或者当一个用户从另外的用户处得到文件的副本时，通常使用这些命令。命令

```
chown kc *
```

将把当前目录下所有文件占有关系交给名叫 “kc” 的用户。新的所有者的名字必须是一个合法的注册名或者是一个用户标识号。注册名及相应的标识号可在文件 ‘/etc/passwd’ 中找到。

命令

```
chgrp staff corelist
```

将把组名 “staff” 与名叫 ‘corelist’ 的文件联系在一起。在 `chgrp` 命令中涉及到的组名可以是组名或组标识号，这些组名和组标识号来自于文件 ‘/etc/group’。许多系统极少使用 UNIX 系统成组处理的特性。

9.4 mkdir和rmdir——建立和删除目录

构造目录 (`mkdir`) 命令用于建立一个目录。当系统建立一个目录时，它自动地插入有关名字 ‘.’ 和 ‘..’ 的项。名字

‘.’是该目录的别名，而名字‘..’是其父目录的别名。所有目录都包含这些项。而普通用户不得删除这些项。一个仅包含‘.’和‘..’项的目录被看成是空目录。

命令

```
mkdir morestuff
```

将建立一个名为‘morestuff’的目录，它将是当前目录的一个子目录。如果你打入命令

```
ls morestuff
```

将会发现‘morestuff’是空的。命令

```
ls -a morestuff
```

将展示‘morestuff’只包含两项：‘.’和‘..’。命令

```
ls -id . morestuff/..
```

将展示在‘morestuff’中的‘..’项和当前目录涉及到的同一个索引节点（为获得有关任选项a、i和d等更多的信息，请参见7.2节和简明手册中有关ls的描述）。

在使用‘morestuff’目录的过程中，它自然会被大量的文件填满的。如果你想删除目录‘morestuff’。那么你必须先删除它的所有内容。如果‘morestuff’只包含普通文件，那么命令

```
rm morestuff/*
```

就可以把它清空。如果‘morestuff’包含了子目录，那么，清空它需要做更多的工作。一旦‘morestuff’清空了，命令

```
rmdir morestuff
```

就会真正地把这个目录删除。

第十章 高级编辑

第五章介绍了打入和修改正文时用的基本命令。对于偶然使用编辑程序的用户，这些基本命令已经足够了，但是为了获得更高的效率，用户应该掌握这章所介绍的思想。

正如以前那样，本书试图介绍大多数UNIX系统可以得到的正文编辑程序所共有的正文编辑概念。要在本章中论述有关高级编辑的所有特性的目标很难达到，因为正是在高级编辑方面各种编辑程序往往会发生分歧。

本章末尾集中在UNIX系统编辑的两个方向，即开放行编辑和屏幕编辑上。虽然这两种技术都未被标准化，但是这两种技术都简化了编辑过程。

10.1 把正文读到工作文件中

在第五章中，讨论了一种最简单的方法，它从一个永久的磁盘文件中把正文取至编辑程序的工作文件中（工作文件常常称为编辑程序正文缓冲区）。如果启动编辑程序的shell命令涉及到一个磁盘文件，那么该文件的内容自动地读到工作文件中。如果永久文件不存在，那么当你打入第一个写命令时，可以建立这个文件。
shell命令

```
ed mydoc
```

将启动编辑程序并且把文件‘mydoc’中的内容读到工作文件中。

如果你打入shell命令

```
ed
```

来启动编辑程序，然后打入编辑命令

```
e mydoc
```

可以获得同样的结果。编辑程序的编辑命令 (“e”) 指挥编辑程序去清除工作空间, 然后把指定文件的内容读进工作空间。工作空间中以前的所有内容都丢失了。象本例中那样, “e” 命令可以在一次编辑对话的开始时使用, 也可以在一次对话的中间使用, 以便在一个新文件上开始工作。“e” 命令正象 “q”(quit 退出) 命令那样危险, 使用时要当心。如果老的工作空间已经被修改过, 许多编辑程序将提醒你, 以便在你开始用一个新文件工作之前, 把刚用的那个临时文件写到永久文件中去。

编辑程序并不要求有了文件名后才能工作。先不指明输入文件就启动编辑程序, 尔后再把正文加到空的工作空间中, 也是可以接受的。可以在以后用文件命名命令 (“f”) 或者用带有明确的文件名的写 (“w”) 命令来指明文件名。

有时你需要把磁盘文件的内容和工作文件的当前内容组合起来。编辑命令

```
Or headerfile
```

将把 ‘headerfile’ 的一个副本放到工作文件的第 0 行后。编辑程序的读 (“r”) 命令读进指定的文件并且把它的内容放到给定行后。如果没有提到行号, 信息就放在当前行后。编辑程序的读命令和编辑程序的编辑命令之间的区别是, 读命令把正文加到当前工作空间而不清除工作空间的内容。

UNIX 系统编辑程序不允许你只读进文件的一部分。如果你只需要读进一个文件的少量几行, 那么你必须构造这个文件的一个副本, 然后用编辑程序删去不想要的部分, 最后编辑这个目标文件并且读进被删剩下的副本。

10.2 文件命名命令

编辑程序记录当前正在编辑的文件名。文件命名命令 (“f”) 用来显示或修改记录的文件名。编辑命令

```
f
```

将导致编辑程序去印出被记录的文件名字。编辑程序命令

```
f myaltdoc
```

将把记录的名字改成 ‘myaltdoc’。

有三个可以修改被记录的文件名的命令：“f”、“e”和“w”。“r”(读)命令对于被记录的文件名没有影响。

10.3 全局命令

大多数编辑命令或者与文件中的单独一行有关，或者与附近的一些行有关。编辑程序的全局命令(“g”)用于修饰命令，使它们与包含某一正文模式的所有行有关。假定你想把包含字“help”的所有行印出来。编辑程序命令

```
g/help/p
```

将印出所有包含字“help”(或“helping”、或“helpless”等等)的行。短语“g/help/”修饰打印命令(“p”)以致该打印命令可以和所有包含正文模式“help”的行有关。“g”字符引入全局修饰符短语，而正文模式(在本情况中是“help”)用斜线括起来。

全局命令(以及下面讨论的它的变形)极其有用。在正文文件中的错误和问题常是很有规律的，如果你能识别与正文文件中的问题有关的一种正文模式，那么你就可以设计出一种包括全局命令的解决办法。

全局命令分两个阶段工作。第一个阶段，全局命令检索整个正文并且构造包含指定正文模式的所有行的清单。第二个阶段，对此清单中的所有行完成指定的命令。

也可以给全局命令规定一个地址范围。例如你想把第50行到第100行中所有包含字“alpha”的行都打印出来，那么可以用命令

```
50, 100g/alpha/p
```

也可以用上下文地址来规定全局命令的地址：

```
/beta/,/dopa/g/zeta/d
```

这条命令从包含“beta”的那一行（在当前行后）开始，到包含“dopa”的那一行为止，把其中包含正文模式“zeta”的所有行删除。在这个范围之外包含“zeta”的行不被删去。

“v”命令和全局命令“g”相反，“v”命令走遍整个正文并且构造不包含指定正文模式的所有行的一份清单，然后“v”命令再次通过这个正文并且在这份清单的所有行上执行指定的命令。例如，命令

```
v/-/s/330/340/p
```

将在不包含“-”的所有行上把“330”替换成“340”。

全局命令“g”和“v”除了更改一个命令外，也可以更改一个命令表。假定你需要把包含字“clothing”的每一行中的“hat”改成“cap”，把“sweater”改成“coat”，把“glove”改成“mitten”，你可以用三条单独的全局命令“g”，然而，用下列命令更为容易，给人印象更为深刻：

```
g/clothing/s/hat/cap/\
s/sweater/coat/\
s/glove/mitten/
```

这条命令将在包含字“clothing”的每一行上执行三条替换命令。要注意，在命令表中，每个命令（最后一条除外）都在行尾包含一个反斜线“\”。反斜线是一种暗示，它告知编辑程序，在这个命令表中有多个编辑程序命令。下面是一些高级编辑命令表。

不用地址的命令

e [文件名]

这个编辑命令用于清除正文缓冲区并且读进一个永久文件，有效地开始一个新的编辑对话而不用离开编辑程序。以前在该缓冲区的正文都丢失了。如果提到了文件名，那么正文就从该文件中读出，该提到的文件变成已记住的文件。然而，如果没提到文件名，那么正文从以前已

记住的文件中读进。

f [文件名]

这个文件命名命令用于改变或者显示已记住的文件名。当一个文件名提到时，这个命令改变已记住的文件名；当没有文件名时，本命令让以前已记住的名字打印出来。

! 命令 [自变量]

这个 shell 转义用于执行一个普通的 shell 命令而不必离开编辑程序。

用一个地址的命令①

. r 文件名

这个读命令读取指定文件的内容，并且把它加到给定行后的编辑程序正文缓冲区中。如果没有提到行，就加到当前行后。

用两个地址的命令②

1, \$ g / 正则表达式 / 编辑命令

这个全局命令标记在给定的界内的每一行，这个给定的界包含由正则表达式指明的模式。然后编辑程序命令在每一个标记行上执行。

., . + 1 j

粘连命令把两行或多行组合成一行。

1, \$ w [文件名]

如果没有提到文件名，那么这个写命令把指定的行写到以前已记住的文件中。然后，如果提到了文件名，那么指定的行就写到该文件中。

①缺省的行地址被显示出来

②缺省的地址被显示出来

10.4 粘连命令

粘连命令用于把两行粘接在一起。编辑程序命令

,.j

把前面一行和当前行组合在一起。(“-”号是当前行的前一行的缩写记号)。编辑程序命令

10.20j

把第10行和第20行粘连在一起。小心不要搞出太长(通常长度以255字符为限)的行来。

粘连命令之所以要存在,是因为人们用删除换行符来组合新行太危险了。编辑程序粘连命令不能处理非常长的行,所以编辑程序要进行检查,以确保所建立的行是合理的。

虽然不用粘连命令,编辑程序就不能粘连行,但是用替换命令来分裂行却是可能的。分裂行要比粘连行引起的麻烦少,因为分裂后得到的行决不会长到不能编辑。如果你想把一行分裂成两部分,那么你可以在合适的地方换上一个换行符。编辑程序替换命令

```
s/time/time \  
/
```

通过在字“time”之后放入一个换行符,将把行

```
A stitch in time saves nine.
```

分裂成

```
A stitch in time  
saves nine.
```

两行。在上述命令中,在打入了第二个“time”之后,又打入了一个反斜线。再打回车(或换行符)键,再打一个斜线,接着再打一个回车键来终止该命令。在回车键之前的反斜线解除(转换)了回车键对于系统的特定含义,因此输入的“行”直到第二个回车键才算完结。如果你希望命令写在单独一行上,那么应该注意

用熟悉的命令“s/pat1/pat2/”。当你敲转义的回车键时，虽然你仍在继续打这一条命令，但在终端上已转到下一行了。

10.5 正则表达式

在论述编辑程序的这些章中，都涉及到正文模式。在编辑程序中，正文模式非常重要。用正文模式可以标识和修改行。在模式匹配中涉及到的正文模式称为正则表达式。一个正则表达式匹配一个或多个字符串。

正则表达式用在编辑程序的两种场合中：作为上下文地址，以及作为一个替换命令的第一部分。当正则表达式用作为上下文地址时，它用一对斜线或一对问号括了起来。当正则表达式用在替换命令的第一部分时，虽然斜线是最常用的定界符，但是除了空格或换行符外，其它任何字符都可以用来定界这个表达式。

在替换命令中(s/pat1/pat2/)的第二个正文模式称为替代串。它不是一个正则表达式，因为它并不与模式匹配有关。正则表达式和替代串之间的区别是：正则表达式与正文匹配操作有关，而且它们可以包含特殊字符，这些特殊字符表明某种匹配类型。当这些字符包括在正则表达式中时，则是特殊的字符，而在替代串中就没有特殊含义了。

10.5.1 在正则表达式中的特殊字符

到现在为止，我们已经小心地使用了非常简单的正则表达式。为了熟练地使用编辑程序，你必须理解正则表达式规则。当下列字符用在正则表达式中时，有特殊意义：

- 圆点
- * 星号
- [左方括号
- \ 反斜线
- ^ 脱字号

\$ 货币号

在组成正则表达式时，如果你故意避免使用上面列出的这些字符，那么正则表达式总是与它们本身匹配的。如果把下列五个正文模式用作为正则表达式，那么它们将与本身匹配，因为这里没有使用任何特殊字符：

```
hello
bye
This is a long regular expression!
abcdefghijklmnopqrstuvwxy
01234
```

用反斜线可以转换特别字符特殊含义。例如，正文模式

```
end\.
```

仅当字“end”后跟一个圆点时才与“end”匹配。如果在编辑程序中正文的当前行包含句子“This is the end of the end.”，那么你可以通过打入下列替换命令

```
s/end\./middle./
```

把句子中的最后一个字改成“middle”。在本例中，正则表达式是“end\.”而替代串是“middle”。要注意，在替代串中的圆点不必转义。

假定你要寻找包含感叹语“!!&& *\$\$\$”的一行正文，打印编辑程序命令

```
/!!&&\*\$ \$ / p
```

将该行找出并且打印这行。如果你想把这个感叹语改成“heck”，那么你可以使用替换命令：

```
s/!!&&\*\$ \$ /heck/
```

只要知道编辑程序的特殊字符以及知道怎样转变它们的特定含义，就能让大多数人使用好编辑程序而不会出重大的意外。然而，要精通编辑程序，就要掌握正则表达式的语法。如果你打算回避（而不是掌握）正则表达式的话，就可以跳过本节余下部分去读

下一节。

10.5.2 单字符正则表达式

长而复杂的正则表达式是由单字符正则表达式构成的。所以首先我们必须弄清单字符正则表达式的规则。下面是编辑程序可接受的所有单字符正则表达式的一张清单：

1. 单个非特殊的字符。所有非特殊的字符组成表示它们本身的一个字符的正则表达式。字符“a”就表示“a”，字符“b”就表示“b”，等等。

2. 转义的特殊字符。特殊字符（圆点、星号、脱字号、货币符号、左方括号、反斜线）前面加了反斜线后就失去了其原先特定的含义，

因此，下列成对的字符实际上是匹配指定字符的单字符正则表达式：

- \\. 匹配圆点
- * 匹配星号
- \\^ 匹配脱字号
- \\\$ 匹配货币符号
- \\[匹配左方括号
- \\ 匹配反斜线

在某些编辑程序中，一些特殊字符可以通过指定“非奇异的”方式而永久地转义。在“非奇异的”方式中，特殊字符仅为它们前面有一个反斜线时才是特殊的。许多编辑程序没有“奇异的”、“非奇异的”选择。

3. 特殊字符圆点。圆点是一种除换行符外能匹配所有单个字符的单字符正则表达式。

4. 特殊字符左方括号。左方括号引入一个字符集，该字符集的末尾由一个右方括号表示，在括号之间的字符定义了一个字符集。字符集是一种与该字符集中的任意一个字符匹配的单字符

正则表达式。

让我们略微讲一讲字符集。字符集“[abcd]”匹配字母表中前四个小写字母中的任意一个。可用短划表示包含字符的一个范围，例如，字符集“[a-d]”将匹配字母表中前四个小写字母中的任意一个。在字符集末尾的“-”将失去范围的特殊含义，如字符集“[ab-]”，它只匹配“a”、“b”，或“-”。

如果在一个字符集的头有一个“^”，那么，正则表达式表示此字符集的补集。字符集“[^a-z]”匹配任何非小写字母的单个字符。“^”如果不出现在集合的头，那么将失去这种特殊含义。如字符集“[0-9^&]”，它只匹配任何数字、“^”或“&”号。

字符圆点、星号、左方括号以及反斜线在一个字符集中代表它们本身。字符集“[\.*]”匹配反斜线、或星号、或圆点。

10.5.3 组合单字符正则表达式

可以根据下列更精巧的正则表达式的组成规则把单字符正则表达式组合起来：

1. 串链。正则表达式的串链匹配单字符正则表达式匹配起来的字符串链。因此，正则表达式“abc”只匹配“abc”，而正则表达式“a.c”却匹配以“a”打头且以“c”结尾的任意三字符序列。

2. 运算符*。后随一个星号的单字符正则表达式匹配零次或多次出现的单字符正则表达式。因此，正则表达式“12*3”匹配

13

123

1223

12223

中的任意一个

3. 运算符\$。当\$放在正则表达式的末尾时，该正则表达

式只与一行正文的最后一段匹配，正则表达式末尾的\$ 匹配一个换行符。(但这么说不太正确，因为换行符不包含在任何正文替换中。)放在正则表达式的开头或中间的\$ 将只匹配\$ 。例如，如果字符串“123”出现在一行正文的末尾，那么，正则表达式“123\$”将匹配字符串“123”。如果“\$ 123”出现在一行正文的任何位置，那么，正则表达式“\$ 123”将匹配该字符串。

4. 运算符^。当^在一个正则表达式的开头、或作为正则表达式字符集的第一个字符时，则为特殊字符。除这两种情况外的其它场合，它只匹配本身。如果^出现在一个正则表达式的开头，那么该正则表达式限于匹配行的起始段。^匹配正文行的开头。(但这么说是太不正确的，因为这样的一种匹配结果，并不包含在替换中)。比如，正则表达式“^Hello”匹配字符串“Hello”，但是当它出现在正文行的任何其他地方时，意义就不同了。比如，正则表达式“Hello^”就匹配字符串“Hello^”。我们已经讨论过一个字符集的开头的用法。

某些编辑程序允许正则表达式分段。分段的正则表达式被分成若干片段，这些片段可以用一个特殊记号表示。这种技术用处不大也不普遍，所以就不在这儿进一步讨论。

要注意，UNIX 系统中有一个问题，即在编辑程序中的正则表达式匹配规则与文件名生成规则不同。在编辑程序中的* 表示前面单字符正则表达式出现零次或多次，而在文件名生成过程中，* 匹配任意的字符序列（可能为空）。在编辑程序中，. 匹配任意单个字符，而在文件名生成过程中，却以? 匹配任意单个字符。

10.6 再论替换命令

替换命令可能是最难的编辑程序命令。在一个短行上把一个字改成另一个字是容易的，但是要在一个C语言程序的长行上改变第三个星号所要求的技巧比写这个方程还难。对于这个问题的一个典型的解决方法是：删除，然后费力地手工重新打入这个出

错了的行（校正打字错误的困难常常是费力的另一原因）。本节主要讲解少量有用的、但是较为复杂的实现正文替换的技术。

^和\$ 常常用来简化替换表达式。考虑下面二进制数串中的一行：

```
1101 1101 1101 1101
```

如果你必须把最后一个数（字段）改成“1011”，那么你可以用下列替换命令中的任意一个：

```
s/1101 1101 1101 1101/1101 1101 1101 1011/  
s/1101$/1011/  
s/...$/1011/
```

第一个命令要打该行两次，很明显，这种方法不好。在后面两个命令中，\$迫使正则表达式在行尾出现匹配，因此必须打入的就很少。第二个替换命令明确地指出最后四个字符应是“1101”，它应该由“1011”替代。第三个替换命令仅仅用“1011”替换在行中的最后四个字符。如果尾部有空白或制表符，那么第三个替换命令将失去作用。

^迫使匹配出现在一行的开头。你可以用命令

```
s/^/-/
```

放‘-’在一行开头。在本例中正则表达式由一个单独的组成，要替换的串是一个‘-’。

当可能有若干个匹配时，编辑程序的“最长最左”原则决定了匹配哪一个正文。

1. 最左。总是假定匹配最左。把命令“s/1/2/”应用于正文行“111111”，将产生正文“211111”，因为假定了最左原则。

2. 最长。假定最长匹配。命令“s/1*2/3/”应用于正文行“1111124”将该行改成“34”，因为若干“1”后面跟一个“2”的整个串与一个“3”匹配而被替换掉了（要记住，*号重复前面单字符正则表达式零次或多次）。

较难的问题是把命令“s/1*2/3/”应用到正文行“21112”上。在这里最左原则与最长原则发生了矛盾。很明显，在该行上的最长匹配将包含“1112”。然而，在该行上的最左匹配是领头的“2”（要记住，正则表达式“1*2”将匹配“2”，“12”，“112”，等等）。由于总是假定最左匹配优先，因此将产生“31112”。

有时，你想在一行上进行多个替换。例如，你可能想把某行上的所有“jacks”改成“janes”。跟在替换命令后的字母“g”将控制编辑程序作出所有可能的替换。编辑命令

```
s /jacks/janes/g
```

将把行

```
jacks jacks and more jacks
```

改成

```
janes janes and more janes
```

你需要懂得对于替换命令的“g”任选项和“g”全局命令之间的区别。“g”任选项规定编辑程序应该在一行上完成每一种可能的替换，全局命令“g”规定了一条命令应该在一组行上面执行。如果你想把整个文件中所有正文串“teh”改成“the”，那么你可以用命令：

```
g /teh/s/teh/the/g
```

领头的“g”表明是一个全局命令，它构造包含一个“teh”的所有行的一个清单，然后在该清单的每行上执行替换命令，而最后的“g”任选项规定“teh”的每一个出现都应被改成“the”。

由于在一行上进行了所有可能的匹配，自然，“g”任选项修改了“最长最左”原则。因此，“最左”原则就不再适用了。而“最长”原则能使用，每一个匹配要尽可能地长，而且匹配从左开始进行。

当使用替换命令时，常常可用的其它任选项是“p”和“l”，它们用来印出（或列出）修改过的行。例如，命令

```
s /jack/jill/p
```

将把第一个“jack”改成“jill”，然后印出结果。

有时，正文文件有不可打印的字符。有因为击键错误而不知不觉地混入的古怪的字符，或因为拨号线路的噪声及不希望有的硬件问题所产生的古怪字符。在任何情况下，解决奇怪字符出现的问题要做两方面的工作：确定其存在，然后删除它们。

最普通的不可打印字符之一是退格。自信的新手（偶而还有专家）在他们试图打入UNIX系统抹字符符时，有时打入了退格字符。要记住抹字符符（常常是一个英镑符号、或一个删除键、或是control-h键）是这样一字符，系统把它解释成你要抹去前一个字符的命令。打入一个退格看起来常常象打入一个抹字符符一样，所以很容易把这两种操作混淆起来。在任何情况中，如果你的正文文件由于退格而增长了，那么你在终端上看到的行就不是包含在这个文件中的行了。

如果当你印出正文行时产生了噪声，或者如果你的替换命令不工作，或者如果发生了不可解释的东西，那么可以用“l”命令去寻找不可印出的字符。“l”列表命令类似于“p”印出命令，它们的区别是“l”可以生动地描述出所有不可印出的字符。

一旦在文件中找到了不希望有的不可印出字符，你可以用替换命令把它们删去（你应该删除并且重打那个拙劣的行）。使用替换命令所带来的问题是打入不可印出的字符通常是困难的。匹配一个不可印出的字符，最容易的方法是在正则表达式中使用圆点。如果你的正文行包含一个“l”后面跟一个不想要的和不可打印的字符，它们的后面再跟一个“2”，那么下列命令将删去不可打印的字符，然后输出最后的结果行：

```
s/1.2/12/1
```

要记住，在正则表达式中的圆点匹配包括不可打印的字符在内的任何单个字符（除换行符外）。

或许你已经注意到替代串常常包含非常类似于正则表达式的正文。当你把字符加到已存正文中时，替代串常常是原先正则表

达式加上这个附加的正文。例如，如果你要把字“awkward”改成“awkwardly”，可以使用下列替换命令：

```
s/awkward/awkwardly/p
```

如果你老是试图打入该命令，那么你会感到命令太笨拙。为了减少打入错误，UNIX 编辑程序允许你在替代串中使用&，把&作为由正则表达式匹配的整个被匹配项的缩写记号。使用&，上面展示的命令可以象下面那样打入：

```
s/awkward/&ly/p
```

在本情况中，&表示正文串“awkward”。

当你使用复杂的正则表达式时，&记号也能起作用，比如在下列命令中，这个命令把一个正号放到一行的第一个数字的前面并且把“.00”放在这个数字的后面：

```
s/[0-9][0-9]*/+&.00/p
```

上面这条命令应用于行

```
The year end balance is 550 dollars.
```

将产生行

```
The year end balance is +550.00 dollars.
```

对于在单独一行上进行一个修改，这种相当广义的形式并不是很有用的；但是当你在一个文件中改变许多行时，就有很多好处。

用*正则表达式操作符时，始终要小心。命令

```
s/[0-9]*/+&.00/p
```

用于上面原先的行，会产生行

```
+ .00The year end balance is 550 dollars.
```

因为正则表达式“[0-9]*”将匹配最左的空串。

在至今给出的所有替换命令例子中，正则表达式和替代串都是由斜线定界的。在替换命令（不是在上下文模式）中，任何非空格字符都允许作为定界符。平常使用斜线是因为它们更清晰地限定了表达式。

如果你想在当前行上把字“boy”改成“adolesoent”，那么，

你可以使用下列替换命令中的任何一个：

```
s/boy/adolescent/  
s,boy,adolescent,  
szboyzadolescentz  
saboya\adolescenta
```

在上面展示四个命令中，定界符分别是反斜线，逗号，“z”和“a”。无论什么字符一旦用作为定界符，就变成了该命令的一个特殊字符。在上面展示的最后命令中的“adolescent”中的“a”必须在它的前面加一个反斜线，消除它原来作为一个定界符所具有的特殊含义。要是没有反斜线，编辑程序将把“a”解释成表达式中的最后一个定界符，而尾随的字符“dolescent”将产生一个错误消息。一般来说，挑选可见的（例如，斜线或逗号）并且不在该表达式中出现的表达式定界符是最容易的。

10.7 在编辑程序中使用shell命令

当你在编辑时，有时又想执行一个标准的UNIX系统命令。当然你可以先写你的文件，再退出编辑程序，并且执行这条命令后再重新进入编辑程序，但是这就要做一大堆工作。编辑程序的shell转义命令允许你运行标准的UNIX系统命令而不必离开编辑程序。如果你打入了命令

```
! who
```

那么，将执行who命令。在该命令的末尾，编辑程序将印出一个感叹号表示它准备好接受更多的输入。

编辑对话并不受shell转义的影响。在使用shell转义之前你不必把正文缓冲区写到一个永久文件中。如果你想打入若干命令，那么你可以通过打入命令

```
! sh
```

为你建立一个新的shell，半永久地从编辑程序中摆脱出来。这个新的shell将允许你愿意打多少UNIX系统命令就打多少。你可以试验一下UNIX系统ps命令来印出所有你正在运行的一组进程的清单，你原先的shell，编辑程序，你的新shell，以及ps命令将都会印在这个清单中。通过敲入control-d 你可以终止新的shell。

UNIX系统之所以能够提供shell转义，是因为shell只不过是可以在任何时刻执行的普通程序。当你在编辑程序内部执行一个shell时，编辑程序在耐心地等待shell命令的终结，就象你原先的shell在耐心地等待编辑程序完成一样。

10.8 开放行编辑和屏幕编辑

因为替换命令使用困难，所以已经开发了更好的编辑形式。开放行编辑程序允许你沿一行移动光标然后修改光标附近的行，而不必费脑筋建立且打入一个替换命令来表明修改。

在开放行编辑程序中，一次只有一行可以修改，当你要对整个文件作修改时，你必须依靠全局替换。然而，即使在一个文件中的若干地方出现打入错误，对于一个非程序员来说，手工地修改每个错误也常常比用替换命令更容易并且更迅速。

为了利用开放行特性，必须用打开命令来打开一行。一旦该行被打开了，就有特殊控制码来回移动光标访问这行。虽然控制码必须记忆，但是操作是如此之简单以致可以迅速地学会。一旦光标定位在错误正文上面，就可以用于特殊命令删除该行的正文或者把正文加到该行中。

屏幕编辑是开放行编辑思想的一种扩展。在屏幕编辑中，文件的一部分在你的终端上显示出来。该终端起文件窗口的作用，对文件的任何修改都立即可在终端上显示出来。

当你在文件上改动时，你的窗口部分也就作了修改。象开放行编辑一样，在屏幕编辑时，对文件的大多数修改通过把光标定位在合适地方，然后送入修改命令来进行。如果你需要修改多次

出现的同一个错误，那么你仍然可以使用全局替换命令。

屏幕编辑不能在打印机终端或某些非常笨拙不灵的CRT终端上工作。屏幕编辑在电话线上进行是困难的，因为数据传送速度慢，尽管可以通过缩小观察口来得到比较好的结果。另外，屏幕编辑对处理机也提出了很高的要求。

开放行编辑和屏幕编辑通常是有附加特性的编辑程序，除了这些附加特性外，其它功能类似于UNIX系统标准的正文编辑程序。屏幕编辑和开放行编辑正在变得更为通用了。

第十一章 正文格式加工

UNIX 系统字处理并不是单独的一个程序，它是应用一大批实用程序的一个综合概念。正文处理（也称为字处理）是 UNIX 系统的一个关键特性。当然，对于正文文件来说，UNIX 系统有正文编辑程序、格式加工程序和实用程序。但是，更为重要的是，UNIX 系统有一系列一起工作以提高生产效率的正文处理程序。

UNIX 系统有一大批一起用于正文文件的简单程序——有统计文件中的行数、字数和字符数的程序；为正文模式而检索文件的程序；打印文件的程序；排序文件的程序；检验文件中拼写的程序；比较文件的程序以及重新组织正文文件的若干程序。所有这些程序都已在第八章中讨论了。UNIX 系统也有创建和更改正文文件的编辑程序。这种标准的 UNIX 系统编辑程序已在第五章讨论了，而该编辑程序的高级特性也在第十章讨论了。

本章把要点放在 UNIX 系统中主要的正文格式加工程序 `nroff`、`troff`、`eqn` 和 `tbl` 上。`nroff` 和 `troff` 是通用格式加工程序，`nroff` 和打印终端一起使用，`troff` 和照排机一起使用。`nroff` 和 `troff` 几乎是相同的，它们之间细微的差别是因为照排机和计算机打印机性能有所不同。我们可以用术语 `nroff/troff` 作为引用两个程序的语句。你应该始终把 `nroff/troff` 和一个好的宏程序包一起使用，宏程序包在 11.2 节中讨论。`eqn` 程序和 `nroff/troff` 一起用于对数学公式进行格式加工，而 `tbl` 程序和 `nroff/troff` 一起用于对表格数据进行格式加工。

没有办法用一章（或者甚至整本书）来刻划清楚 UNIX 系统正文加工程序可以使用的所有方法。每一种正文加工应用代表了不同的一组要求，而每一种或一组带有正文的工作又有一组不同

的选择。UNIX 系统有足够的正文加工任选项以致大多数操作可以用若干不同方法中的任意一种来执行。例如，如果你想在—组文件中用正文串“UNIX”替代正文串“Unix”，那么你可以（1）使用正文编辑程序手工地执行这种替代，（2）使用带有存放在一个文件中的的编辑草稿的正文编辑程序，（3）使用字符流编辑程序，（4）使用lex程序。

当你为执行字处理操作挑选—种方法时，应该考虑以下几个条件。—个条件是熟悉。不熟悉的工具经常比熟悉的工具更难使用。另—个条件是你预期执行—个操作的工作量。如果你打算为包含单独—个短表的数据进行格式加工，那么可以不用tbl程序而用熟悉的nroff/troff格式加工程序，手工地对这个表进行格式加工。然而，如果你打算加工更大的表格，那么，你应该花时间学会tbl。本章并不教你怎样在系统上有效地使用格式加工程序，本章仅仅是格式加工性能—种指南，这种格式加工性能通常可以在UNIX系统中找到，它也是对某些术语和思想—般介绍，为了使用为系统的格式加工程序提供的参考材料及辅导材料，你需要理解这些术语和思想。

11.1 nroff和troff——对正文进行格式加工

nroff和troff是UNIX系统的通用格式加工程序。正文格式加工程序取—个未被格式加工的正文文件，并且把它转换成—个经过格式加工的正文文件。未经过格式加工的正文有未加工的边缘和不适当的空格，而经过格式加工的正文有调整了的边缘、合理的空格、书眉、脚注、标题和其它细节。nroff用于为在打印机上打印的正文进行格式加工，而troff用于为在照排机上输出的正文进行格式加工。由于绝大多数UNIX系统不连接照排机，因此大多数人都使用nroff程序。这两个程序接受几乎相同的输入，

但是由troff产生的输出要比由nroff产生的输出漂亮得多，因为照排机比打印机更为灵活。

当你使用普通电传打字机时，你是通过把正文放到纸上来规定资料的格式的。如果在段落之间你需要两个空白行，那么在每一段落之前，敲两下回车键。如果你想有一个大的左边缘，那么你可以向中间移动左边缘。使用nroff/troff正文格式加工系统与使用一架电传打字机有很大不同，在正文格式加工系统中，你打入的正文的具体格式对于最终的资料影响很小，在这种系统中，最终资料的格式要通过在正文中嵌入格式控制行来规定。我们将在下面及下节展示格式控制行的若干例子。

在UNIX系统中，正文几乎总是不经加工地打入，因为如果在打字的同时过多地考虑右边缘、留空、脚注以及即将出现的其他（或许是扩充的）形式，那是很艰难的。由于正文和格式是两个独立项，为了在正文内部规定格式，已经开发了一个系统。以一个圆点或单引号打头的行被解释为格式加工命令而所有其它行解释成为正文。

这个系统实际上非常简单。如果你想跳过一行，那么只要在正文文件单独一行上打入nroff/troff命令

```
.sp
```

在未经过格式加工的文件中，格式控制行仅仅是正文行。然而，当你运行格式加工程序时，那些格式控制行就被删去并且完成相应的动作。格式控制行用来控制留空、提供标题和书眉，给页编号、缩进处理和大量其它特性。修改带有格式控制项的正文组成的文件比包含经过格式加工后的资料要容易得多。如果你在一个经过格式加工后的资料的页中改变句子的长度，那么整个页必须加以调整。然而，在一个未经格式加工的资料中修改正文是不难的。如果你在一个未经格式加工的资料中改变句子的长度，那么整个页就不必加以调整，因为该页还没有被格式加工程序装配。

nroff/troff 有大约80个内部的格式加工命令。每个这样的命令通常完成一个小而特殊的功能。象开始一个新段落那样的操作可以表达成一系列基本的内部功能。例如，开始一个新段落的这一系列命令可能由跳行功能和缩进少量空格功能组成。如果正好在页尾或页首，这种开始新段的工作怎么办呢？很明显，如果正好在一页的首部，就不必跳过一行。你也不要在一页的最后一行开始一段。当你考虑所有可能性时，那么，即使象开始一个新段那样的简单命令也会弄得很复杂。

nroff/troff 允许一系列基本操作组成一个组，称为宏程序。编写一个宏程序你就可以建立一个新的操作，该宏程序包含一系列原有内部的nroff/troff命令以及先前定义的宏程序。通过在未经格式加工的正文文件中打入它们的名字就可以引用这些宏程序；当某些条件出现，比如在一页中到达某一确定行时，宏程序也可以自动地被引用。

在nroff/troff内部设置命令的目的是控制输出设备的特殊功能。nroff/troff好比一个汇编语言，它内部的命令与机器特性有密切的关系。为了满足普通的正文处理要求，已经建立了若干宏程序包。这些宏程序包完成诸如开始一个段落、为页码编号、放置脚注、建立目录表等通用功能。

了解nroff/troff最重要的事情是：你不应该直接使用原有的命令。而是应该使用与正文格式加工的细节无关的宏程序包。ms和mm是两个最常用的通用宏程序包。如果你的UNIX系统渊源于贝尔实验室UNIX系统，那么你也许有这两个程序包中的一个。其它宏程序包在许多系统上可以得到，在选择一个宏程序包时，要听取用过这些程序包的人的意见。

11.2 使用宏程序包

编写宏程序包的人应该是直接使用大量nroff/troff的人。使用宏程序包的人应该找出可在他们系统上得到的那些宏程序包，

他们在开始时至少可以不管 nroff/troff。如果你已成了一个有更高要求且更老练的正文处理者，那么当现存宏程序包不能满足你提出的要求时，为了扩充你中意的宏程序包，你应该学会 nroff/troff 系统。

在使用一个与 nroff/troff 有关的宏程序包时，你仍在使用 nroff 或 troff 程序。其区别是：内部的 nroff/troff 命令由高级宏命令支持。

为了解释宏程序包的典型特性，让我们假定一个称为 mh 的宏程序包，它包含下列三个宏程序

- .P 开始一个缩进的段落
- .C 对准下一行
- .S 跳过一行

一个有用的宏程序包所包含的宏程序比三个要多。我们这里假定只有三个宏程序是为了叙述简单。

下面是假设的宏程序运用于某些未加工正文的非常简单的例子：

.C

Starting a Paragraph in MH

.P

Some people use the .P macro command to start a paragraph because they prefer indented paragraphs.

This paragraph was started with a .P command.

.S

Other people use .S to start a paragraph because they abhor indentation.

This paragraph was started with a .S command.

如果上面所示的正文存放在名为 ‘rawtext’ 的文件中，那么命令

```
nroff -mh rawtext
```

将在你的终端上产生下列输出：

Starting a Paragraph in MH

Some people use the `.P` macro command to start a paragraph because they prefer indented paragraphs. This paragraph was started with a `.P` command.

Other people use `.S` to start a paragraph because they abhor indentation. This paragraph was started with a `.S` command.

上面给出的 `nroff` 命令使用自变量 “-mh” 通知 `nroff`，文件 ‘rawtext’ 包含了有某些前面假设的宏程序的正文。

使用带有 `nroff/troff` 的宏程序包，可能包括了少量使人感到意外的事情。注意，在上面例子中，宏命令在一行的开头才算数，嵌入在行中间的宏命令不起作用而只被复制到输出中。

在用 `nroff` 进行正文格式加工中另一件使人感到意外的是：以空格或制表符打头的未加工正文的行将导致间断。间断后正文填写过程不可继续。作为一个例子，下列若干行展示以空白格打头的某些未加工正文：

P

A line beginning with white space will cause a break.
A break is a discontinuity in the text filling process.

This line begins with white space.

如果上面所示的未加工正文采用假设的宏程序由 `nroff/troff` 处理，输出将如下所示：

A line beginning with white space will cause a break.
A break is a discontinuity in the text filling process.
This line begins with white space.

上例第三个句以单独的行出现，因为它的未加工正文中打头的空白导致了间断。要是在未加工正文中没有打头的空白，第三句就会正常地在第二句结尾处开始填写。间断由许多nroff/troff命令自动产生。

除了以空白格打头的行的问题外，你亦应该谨防以圆点或单引号打头的行。圆点和单引号用来标志nroff/troff命令的启动，以这两个字符之中任一个打头的行必须是一个有效命令。

只要你使用宏程序包，你就必须告诉正文格式加工程序，你正在使用什么程序包。nroff/troff的“-mNAME”任选项通知格式加工程序，它将包括宏程序包“NAME”。命令

```
nroff -mm file1 file2...
```

将为指定的文件使用mm宏程序包，而命令

```
nroff -ms file1 file2...
```

将使用ms宏程序包。

由于nroff通常把结果写到标准输出上，因此你可以使用输出重新定向来把结果保留在一个文件中。命令

```
nroff -ms rawtext >results
```

将用ms宏程序包处理‘rawtext’，并且把格式加工后的正文放到名为‘results’的文件中，要注意，troff通常不写到标准输出上。

对于nroff/troff来说，可以使用若干任选项。“-s”任选项在每一页后让程序停止以允许你修改记录，“-o”任选项允许你控制打印哪一页，而“-n”任选项允许你控制赋给第一页的页号。正如上面讨论的，“-m”任选项允许你指定使用哪一个宏程序包。

nroff程序允许你用“-T”任选项指定使用的终端型号，如果

你不用“-T”任选，那么终端型号来自shell变量\$TERM

nroff和troff是令人讨厌的慢速程序，在某些装置中，在工作时间内，它们不让正文加工程序加重系统负担。如果你的正文文件比较小的话，那么，一般来说，正文处理工作进行得很好。但长于10页的文件是难于编辑的，并且格式加工要花费很长时间。在就程序设计的开头，先用小文件工作，然后在设计结束时把这些文件组合起来就比较容易。如果你保持这些文件小型化，那么诸如一本书那样的大工程就可以包含数百个小文件。请考虑一下用于管理几组文件并且使用良好的 make 程序（参见第十二章）。

11.3 tbl——加工表格

用传统（非UNIX系统）的字处理程序处理，最困难的问题之一是表格数据。数据表格较难处理有许多原因，比如，列的宽度就难于对齐，获得确定的或对准的列项常常也较难，并且当你接近表格的末尾时，你几乎总是发现某些东西要退回到表的开头去调整。围绕表格画一个方框，或者在某些项或行下划线，或者用竖线分开列等等工作，在大多数系统中几乎是不可能的。用剪子裁剪并且粘贴倒是通常的办法。

当然，UNIX系统中的办法不是一把剪子。程序tbl是nroff/troff的伙伴程序。tbl是为了产生一个表格而建立nroff/troff手稿的程序。如果你要建立包含表格数据的资料，那么你应该学会使用tbl程序。

nroff/troff手稿生成程序(tbl)和宏程序包(ms, mm)之间有重大区别。一个手稿生成程序就是一个UNIX系统命令。tbl输入文件中包含在正常nroff/troff正文内部嵌入的表格数据。tbl把表格数据转换成nroff/troff命令以使输出只包含nroff/troff正文。为了把表格数据转换成nroff/troff手稿，在这个表格中，对项的大小和类型必须加以分析。这类分析已超出了宏程序包的能力。在概念上宏程序包要简单得多，因为每一个宏程序总是扩充

成已知的一系列内部的nroff/troff命令。

通过tbl加以构造,使得表格数据和正常的正文数据可以在一个文件中混合。表格数据的开头由命令“.TS”指明,而表格的结尾由命令“.TE”指明。tbl命令和表格数据在“.TS”和“.TE”命令范围之内定位。

tbl程序转换在表格开始和表格结束命令之间出现的表格信息,并且把它们转换成讨厌的不可读的一系列nroff/troff命令。在表格开始和表格结束命令以外的文件内容不受tbl的影响。一旦文件受到tbl处理,它还必须由nroff/troff处理。一个典型的与nroff有关的tbl的应用是:

```
tbl reportdata | nroff -ms >prettyreport
```

在这个命令中,‘reportdata’文件首先由tbl处理,然后通过管道传给nroff,用ms宏程序包作进一步处理,再输出到文件‘prettyreport’中。

表格处理在nroff/troff上下文外面执行,因为象表格处理那样的专门应用并不属于通用正文格式加工程序。nroff/troff不必了解表格,而tbl也不必了解格式加工。tbl可以很容易地随nroff一起使用,这是UNIX系统让程序一起工作来解决困难问题这一宗旨的一个证明。

11.4 eqn——加工数学公式

对于大多数正文处理程序来说,另一个困难的工作是打入数学公式。数学公式难于加工是因为其特殊的符号,还因为与标准的正文处理的一维问题不同,数学公式表示了一种二维问题。因为nroff是与标准打印机紧紧连在一起的,所以用nroff产生数学公式是不好办的。然而,troff却有产生数学公式的照相制版副本。

eqn是nroff/troff的一种手稿生成程序。eqn取出放在命令“.EQ”和“.EN”之间的数学公式说明,并且产生合适的nroff/

troff手稿。通过用几乎是助记的方法写数学公式，可以用**eqn**说明一个数学公式。通过写**eqn**命令行

```
1 over 2
```

可以产生 $1/2$ 。为了使得**eqn**能被不懂数学的人（例如打字员）采纳，**eqn**没有数学知识。为了引导**eqn**产生一份手稿，**eqn**的用户必须用文字勾画出该数学公式的一幅图，这样的一份手稿可由**nroff/troff**来解释产生数学公式。

eqn的典型应用是

```
eqn eq.doc | troff
```

如果**nroff**正在使用，那么你可以用

```
neqn eq.doc | nroff
```

要注意，**eqn**是与**troff**一起使用的，而**neqn**是与**nroff**一起使用的。

eqn和**tdl**可以通过增加管道线的长度而一起使用：

```
tbl sigplan.ent | neqn | nroff
```

为了使通过管道线传递的数据尽可能少，**tbl**应放在**eqn**前面。

第二部分 UNIX系统更深入的课题

第十二章 make 和源代码

控制系统(SCCS)

在大型程序设计项目中，协调是一个主要问题。当许多程序员（几十个或几百个）一起工作时，相互间的交流要花费许多时间和精力。在大型程序设计项目中，效率的降低已经成为软件工业公认的事实。程序员之间协调不好的最严重后果是得到的产品质量很差。大型程序设计项目中某些最顽固的毛病往往和程序小组间没有很好地相互协调有关。

UNIX 系统并不是一个无错误的系统，但它是一个可靠的程序设计的生动例子。UNIX 系统的可靠，部分原因是由于它是真正的模块化设计。许多UNIX的实用程序小而简洁，并且比起其他系统中常见的那些庞大而复杂的实用程序来说，其维护要容易得多。可从UNIX系统学到的另一经验是，使用带有一组强有力工具的操作系统，将使可靠软件更容易得到开发。因为正文文件是所有软件的原始格式，所以在UNIX系统的工具库中，许多程序是处理正文文件的标准实用程序。UNIX系统工具库的其它程序用于分析编译程序产生的二进制代码文件（见第十六章）。然而，UNIX系统最独特的软件工具是本章要讨论的内容，它使很多通常在软件开发者头脑里的软件系统知识形式化。

本章第一节解释大型程序设计项目中的一些共同问题，后二节则专门讨论UNIX系统的实用程序 make 和构成源代码控制系

统 (SCCS) 的一组实用程序。

12.1 大型程序

一些程序设计语言允许程序分成几个独立工作的分离部分(这里不很严格地称作“模块”)。许多程序语言在精确定义模块和保持各模块的完整性方面工作得很好。问题出在一个模块和其它模块的相互关系上。独立的模块通常被一个有相互依赖关系的模块网所牵制,一个模块有一点改动就能使另一模块不能再用。

若程序模块分别放在各自的文件里,可能会产生几个问题:先考虑一个低层模块,它把相互商定的值送回到某些高层模块。在大型项目中,这两个模块可能在不同的文件中,这样它们可能分别进行编译。如果低层模块改动了,它将送回一组不同的值,自然高层模块也必须作改变。

协调不同模块的一种技术是把几个模块所需要的定义放在一个称为include的公用文件里。上面提到的那种情况中,在这个include文件内,应定义这个由低层模块传递到高层模块的值。由于所有模块引用同样的include文件,因此可以认为这两个模块将自动被同步。

虽然include文件解决了程序模块化的许多问题,但有时也会导致一个更困难的问题——定时问题。一旦你有一个公共的include文件和一些独立的程序模块,如果你在include文件中进行了改动,而不去重新编译受到影响的全部程序模块,那将冒很大的风险。若一个程序模块包含了某个文件,那么我们就说这个程序的目标文件既依赖于这个include文件也依赖于源程序文件。如果目标生成以后,include文件或者源程序文件被修改了,那么这个目标文件就算“过时”了。

在一个大型程序设计项目中,编译的顺序和范围取决于模块的内部引用关系。近来的一些语言(如MODULA)已经设计成按照模块的引用关系,让大量文件自动地进行协调。当使用的语

言和 MODULA 不同时，UNIX 系统 **make** 程序能用来实现模块间的协调关系。**make** 接受一个说明，此说明规定了模块间的相互关系，也规定了更新模块时必须进行的动作。**make** 将按照这个说明和有关文件的修改时间，自动地维护这些模块。

程序一般要通过一个调试阶段，将许多错误找出并加以修复。在调试以后留下来的错误一般更顽固，去除这些错误所花费的代价要比早期发现它们花费的代价多得多。在软件研制生存周期到达一定阶段后，修复错误将变得十分困难，因为每一次改动多半会引起一些不希望发生的问题。

在软件项目研制的早期，由于整个项目还未健全，所以错误的改正对整个项目只有很少的影响。可是对于一个成熟的产品，每次改动必须考虑十分仔细，因为这时产品作为一个整体已经基本正常。源代码控制系统（SCCS）是一组 UNIX 系统程序，在一个程序的整个生存周期中，它将使系统容易维修并形成相应文件。

在大型程序设计项目中的另一个问题是需要程序的不同版本。自然 SCCS 对于维护程序的不同版本是很有用的。

make 和 SCCS 是 UNIX 系统维护大型软件项目的两个最强有力的工具。它们是运行于 UNIX 系统下的程序构造和用在其他系统中的程序开发的重要工具。

12.2 **make**

make 是一个程序，它接受不同程序模块间相互关系的信息，以便根据文件的修改日期去推断哪个经过编译的模块已经“过时”了。当“过时”的一些模块被发现后，将执行说明所包含的命令，这些命令通常完成一些更新动作。

作为一个十分简单的情况，考虑存在文件 ‘network.c’ 里的一个主程序模块。它还有一些子程序，它们的源程序存在文件 ‘subrs.c’ 里。假定 ‘network.c’ 和 ‘subrs.c’ 均包含一个公

共定义的文件‘netdefs. h’。它们间的关系见图12.1 (include 文件在15.7节讨论)。

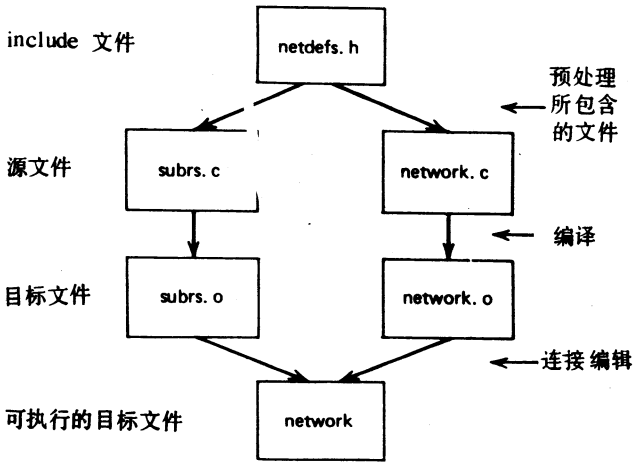


图12.1 一个小型软件系统的关系图

这个 network 软件一共包括了六个主要文件: ‘netdefs. h’ 是这软件系统的公共定义部分; ‘subrs. c’ 及 ‘network. c’ 是 C 源程序; ‘subrs. o’ 及 ‘network. o’ 是经过编译的目标程序; ‘network’ 是可执行的程序。

make 使用的相互关系说明放在文件 Makefile 内。在 Makefile 内的从属关系是这样来规定的: 冒号左边放从属模块, 冒号右边放独立模块。下面以 ‘subrs. c’ 模块为例说明从属关系。

由于在模块 ‘subrs. c’ 中有一个包含 ‘netdefs. h’ 文件的编译指令, 所以应编译的实际正文应来自两个文件: ‘subrs. c’ 和 ‘netdefs. h’。因为 ‘subrs. c’ 不是一个完整的程序, 因此我们执行部分编译而不是一个完整的编译, 部分编译的结果将放在一个 ‘subrs.o’ 的文件里。“o” 词尾指出文件 ‘subrs. o’ 是一个目标文

件（目标文件是一次编译的结果，见第十六章）。

由于 ‘subrs.o’ 是经过对包含 ‘netdefs.h’ 在内的正文 ‘subrs.c’ 编译而生成的，因此我们称 ‘subrs.o’ 依赖于这两个文件，在 Makefile 里它是这样表示的：

```
subrs.o:subrs.c netdefs.h
```

在 Makefile 中，除了从属说明外，还需要插入更新“过时”模块的 UNIX 系统命令。UNIX 系统命令放在从属说明下面缩进的一行中。因此目标模块 ‘subrs.o’ 的从属关系和重新生成的整个说明是：

```
subrs.o : subrs.c netdefs.h
        cc -c subrs.c
```

C 语言编译程序的 “-c” 任选项引导编译程序对 ‘subrs.c’ 执行部分编译，并且把目标代码放到文件 ‘subrs.o’ 中（因为 ‘subrs.o’，并不包含一个完整的程序，它只包含辅助的子程序，所以不可能是完整的编译）。同样 ‘network.o’ 目标模块的说明是：

```
network.o : network.c netdefs.h
        cc -c network.c
```

程序 ‘network’ 依赖于两个目标模块：‘network.o’ 及 ‘subrs.o’ 它能按 Makefile 所说明的项予以生成：

```
network : network.o subrs.o
        cc -o network network.o subrs.o
```

C 语言编译程序的 “-o” 任选项，要求编译程序将可执行的输出代码放在文件 ‘network’ 中而不是放在缺省文件 ‘a.out’ 中。全部维护 ‘network’ 程序的说明放在一起，成为如下页那样的 Makefile 文件。

这个文件由 Make 实用程序所使用，从而生成一个从属关系表以及一个为重构过时文件的修正表。当每一次（使用正文编辑程序）改动源程序或 include 模块时，我们可使用这个 Makefile

```

network : network.o subrs.o
        cc -o network network.o subrs.o
network.o : network.c netdefs.h
         cc -c network.c
subrs.o : subrs.c netdefs.h
         cc -c subrs.c

```

文件去重新编译相应模块。对于下面例子，我们假定上面给定的 Makefile 说明放在名为 ‘Makefile’ 的文件里（‘Makefile’ 是 make 程序专用的一个缺省文件名）。首先我们改动 include 文件 ‘netdefs. h’ 中的某些定义，然后再打入下面命令：

```
make network
```

上面这个 make 命令的目的是看文件 ‘network’ 是否过时；若已经过时，则生成一个新版本。make 命令首先打开文件 ‘Makefile’，并形成一从属关系表。根据这个表，make 指出表中哪一项已经过时（见图12.2）。所谓一个文件“过时”，是指它

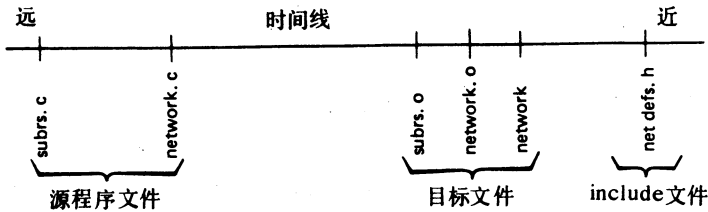


图12.2 一个网络软件系统的时间图

按照例子中 make 的说明，时间表指示有三个文件（‘subrs. o’，‘network. o’，及 ‘network’）已经过时，这是因为它们比 include 文件 ‘netdefs. h’ 更老。

所依赖的模块最近修改了，而它本身尚未更新。由于我们例子中的每一模块均依赖于 include 文件 ‘netdefs. h’，因此下列文件均是“过时”的：subrs. o，network. o及network。

为了重新生成新文件，下一步make将执行以下命令：

```
cc -c subrs. c
```

```
cc -c network. c
```

```
cc -o network network. o subrs. o
```

最后的结果是 ‘network’ 的新版本。

make 对单独一个模块的改动是其十分有用的例子。假定在 ‘subrs. c’ 文件中的源程序有一改动（图12.3）。现在命令
make network

将引导 make 找出有两个文件（subrs. o 及network）是过时的。为生成一个新的 ‘network’ 程序，make 将执行 UNIX 系统命令：

```
cc -c subrs. c
```

```
cc -o network network. o subrs. o
```

要注意，为了生成一个 ‘network’ 的更新版本，make 将执行最少量的再编译。

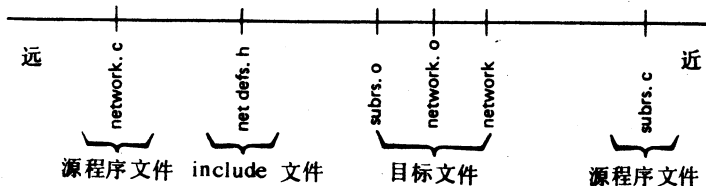


图12.3 网络软件系统的另一个时间表

在此例中，只有两个目标文件是过时的：‘subrs. o’ 是过时的，因为它比 ‘subrs. c’ 更老；‘network’ 是过时的，因为它依赖于已过时的 ‘subrs. o’ 文件。要注意 ‘network. o’ 并未过时。

在这个简单系统里,只包含了两个源程序文件和一个include文件,因此使用 make 并不带来很多好处。然而,维护一个包含了十多个源程序文件及一组不规则层次的 include 文件的系统时,使用make要容易得多。

你可能已注意到,上面给定的 make 说明是相当噜嗦的。由于目标文件 'subrs. o' 通常依赖于源文件 'subrs. c',因此make 包含了一个由 'subrs. c' 生成 'subrs. o' 的内部规定。例如,make 将知道文件 'subrs. o' 是使用任选项 "-c",通过编译文件 'subrs.c' 生成的。利用make的这个内部规定,能重写 Makefile 文件:

```
network : network.o subrs.o
        cc -o network network.o subrs.o
subrs.o network.o : netdefs.h
```

象UNIX系统 shell 的命名变量那样。make 也有一组命名变量。假定在Makefile中有如下一行:

```
CSOURCE = network. c subrs. c
```

那么你就能使用字 \$(CSOURCE) 来表示所有源程序。作为一例,考虑下列Makefile的从属关系和命令行:

```
listing : $(CSOURCE)
        pr $(CSOURCE) | lpr
```

使用上述特点, UNIX系统命令

```
make listing
```

将在行式打印机上印出全部源程序文件。这是因为在前面给定的

全部 make 例子中，UNIX 系统命令只是在相应项已经这时才予以执行，在此例中，因为并没有一个真正的名叫 ‘listing’ 的文件，它只是被假想为已过时的文件，于是 UNIX 系统命令将一直被执行。

类似地，考虑从 Makefile 中摘录来的下面一段：

```
network.lint : $(CSOURCE)
lint $(CSOURCE) > network.lint
```

命令

```
make network.lint
```

将使用 lint 程序来检查 C 语言的源程序，并把它的输出结果放在文件 ‘network.lint’ 里。

虽然有关程序设计的各种不同任务都能借助于 make 来实现，但它最重要的用处是当作一个复杂源程序系统的相互依赖关系的文件。

12.3 源代码控制系统 (SCCS)

SCCS 通过建立一些类似于帐目变动记录的文件，来控制正文文件并使它文档化。许多源程序代码文件因为发现错误和增加新的内容而需要在较长时间内更动和演变。SCCS 是一个使这些更动文档化，并且控制谁有权去作这些更动的系统，它也是恢复文件老版本的一个系统。SCCS 适用于任何类型的正文文件，但一般它用于带有许多注解的源程序文件。下面我们假定 SCCS 的正文实际上是一个程序。

SCCS 工作时，在一个专门的 SCCS 格式文件里保存一个正文文件的编码版本。编码格式中包含了足够信息以便重新生成老版本，也记载了谁将（或被允许）对文件进行专门修改。全部格式文件名字均使用前缀 “s.” 因此 SCCS 文件 ‘s. network.c’

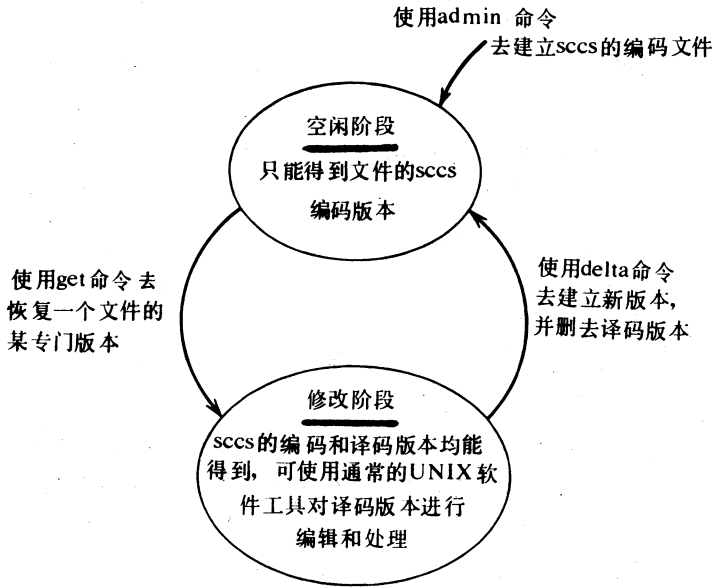


图12.4 SCCS修改周期

是 ‘network.c’ 的编码版本。

在用例子说明SCCS程序的使用之前，我们先一般地描述一下SCCS的修改周期。参考图12.4。对于一个成熟的软件产品，在它的两次修改之间通常有一个很长的空闲阶段。

在修改之间的空闲阶段，仅有文件的SCCS编码版本存在。当必须进行修改时，用`get`命令将已编码的SCCS格式文件恢复为文件的某专门版本，修改实际上是对已恢复的文件版本进行的。如果文件是一个源程序，则程序员可以通过一系列改动，并用编译及测试去证实对文件所作的改变。

在修改过程中，SCCS的编码文件和已恢复的译码文件都存

在。当这次修改完成时，使用 `delta` 命令，将包含在已恢复文件里的新信息加到 SCCS 格式编码文件里，并使用 `delta` 命令更新 SCCS 格式编码文件以删除未编码文件。在两次修改之间，保留打开的格式版本是危险的，因为此时可能做了一次不需要的修改，但主要复制件却被更改了，因此在空闲阶段只能用文件的一个 SCCS 的格式版本。

下面的图 12.5 是 SCCS 系统的主要命令表及其功能说明。

<code>get</code>	恢复一个已编码文件的版本
<code>admin</code>	在已编码文件上执行一些管理功能
<code>delta</code>	将一新版本放入一个已编码文件中
<code>prs</code>	打印一个已编码的文件

图 12.5 SCCS 系统的命令表

最好在软件项目生存周期的一个恰当时刻使用 SCCS 系统。如 SCCS 系统使用过早，则软件的许多早期版本将包含明显的修复痕迹；但若使用太晚，将会丢失许多必要的信息。

对一个软件系统，能用 `admin` 命令给 SCCS 予以初始化，`admin` 除建立一个 SCCS 格式文件以外，还能控制现有的 SCCS 文件的参数。命令：

```
admin -n s. SCCSsample
```

将建立 ‘s. SCCSsample’ 文件，并在文件里设置了标准参数。和 SCCS 系统信息不同，这个文件将是空的（这个命令的另一个变体，也能建立一个非空文件），并用 1.1 号版本予以标志。在向这个空文件中写正文之前，我们必须使用命令

```
get -e s. SCCSsample
```

来恢复文件。“-e” 标志位表明我们复原的文件可用于编辑。`get` 命令将印出已恢复文件的 (1.1) 版本以及在这个已恢复文件中的行号 (0)，此时 SCCS 的工作已经开始——建立一个 SCCS 的记录保存系统和一个空格文件。我们可使用标准的 UNIX 系统正文编辑程序将正文加到文件中：


```
ed SCCSsample
a
Doctor Foster went to Gloucester
In a shower of rain;
He stepped in a puddle,
Right up to his middle,
And never went there again.
```

```
wq
```

现在假定这次修改已完成。我们使用命令 `delta` 来保存文件的新版本：

```
delta s. SCCSsample
```

因为 `delta` 命令自动地将版本号加了 1，所以这次改动将被保存在 (1.2) 版本内，`delta` 程序将在终端上显示一个提示信息 “comments? ”，然后从终端上读一个注释到文件中。这个注释用来说明这次修改的原因。在这个例子中，假定送入的注释是 “placed rhyme in file”。一旦文件 ‘SCCSsample’ 的内容已经安全地加到文件 ‘s. SCCSsample’ 中，`delta` 程序将删去前一文件。`delta` 还印出一个改动的统计摘要及一个出错消息表，这种错误可能是我们忽视有关的标识关键字而引起的。

下面让我们恢复 ‘s. SCCSsample’ 的 (1.2) 版本，然后加入一些正文，以便建立 (1.3) 版本。命令

```
get -e s. SCCSsample
```

将复原最近的版本，可用正文编辑程序修改文件：

```
ed SCCSsample
$a
- Book of Nursery and Mother Goose Rhymes
by Marguerite deAngeli, Doubleday, 1953.
```

```
wq
```

```

<^a>h32774
<^a>s 00002/00000/00005
<^a>d D 1.3 82/04/17 11:57:30 kgc 3 2
<^a>c Added the citation
<^a>e
<^a>s 00005/00000/00000
<^a>d D 1.2 82/04/17 11:56:05 kgc 2 1
<^a>c Placed rhyme in file
<^a>e
<^a>s 00000/00000/00000
<^a>d D 1.1 82/04/17 11:47:26 kgc 1 0
<^a>e
<^a>u
<^a>U
<^a>t
<^a>T
<^a>l 2
Doctor Foster went to Gloucester
In a shower of rain;
He stepped in a puddle,
Right up to his middle,
And never went there again.
<^a>l 3
- Book of Nursery and Mother Goose Rhymes
by Marguerite deAngeli, Doubleday, 1953.
<^a>E 3
<^a>E 2
<^a>l 1
<^a>E 1

```

图12.6 文件 ‘s. SCCSsample’

按正文中提到的操作执行后所得到的 ‘s. SCCSsample’ 文件的内容，为 SCCS 系统的内部工作提供了一种启示。除了七行文件的实际内容外，其他各行均用不可印刷字符 control-a 开头，这里用符号 “<^a>” 表示。

这时，将加两行到已复原的文件里。我们可使用命令

```
delta s. SCCSsample
```

把修改过的文件存入新的（1.3）版本。最后当 delta 命令询问“comment?”时，假定我们送入注释“Added the citation”。

图12.6表示在前面操作已经执行完后文件‘s. SCCSsample’中的内容。图12.6给你一个有关SCCS系统内部工作的直观感觉；并不要求你理解所有各行。从图12.6中，你可以看到，正是用了存储器花的代价才换来了SCCS系统的能力。我们的七行正文文件，在SCCS编码格式文件中用了30行来存储。

使用 get 命令能恢复 ‘s. SCCSsample’ 三个版本中的任何一个。送入命令

```
get -s -p -r1.2 s. SCCSsample
```

能在终端上得到文件的第（1.2）版本。“-s”标志位将禁止改动的统计摘要的正常输出，“-p”表示输出在终端上，“-r1.2”表示我们要文件 ‘s. SCCSsample’ 的（1.2）版本。送入下列命令，能在终端上显示最新的版本。

```
get -s -p s. SCCSsample
```

虽然这个例子仅仅展示了SCCS的少量特点，但显而易见，使用SCCS能在一个软件产品的整个成熟期控制源程序代码。

第十三章 shell程序设计语言

shell 可能是最重要的 UNIX 实用程序；可是它也是最不易理解，资料最贫乏的实用程序之一。UNIX 系统 shell 既是一个交互命令解释程序，也是一个命令级程序设计语言解释程序。这是 UNIX 系统最强有力的特点之一。

某些计算机系统具有简单而有效的交互命令解释程序，但是它们缺少编制复杂命令程序的能力，另一些计算机具有精巧的命令级程序设计语言，但并不为直接运行一个程序提供手段。UNIX 系统将两者的功能结合在 shell 内。

我们已经（在第 4 章）讨论了 shell，你也已经交互地使用了 shell。许多交互式用户只是练习了少量的 shell 功能。shell 的典型交互式用法包括送入简单的命令（“ls”），使用 shell 的文件名字生成机构（“ls *.doc”），指定输入输出重新定向（“ls > myfile”），以及管道线（“ls | wc -l”）。这些技术是极其有效和有用的，但它们只是 shell 功能中的一小部分。

交互地控制一个任务和建立自动执行一个任务的程序，这两者间有十分重要的区别。当你交互地控制一个任务时，你能在开发时发挥你的智力作用。程序是没有智力的，它们的反应是由人预先决定好并编在程序里的。任何写过大程序的人都知道，要预知所有可能的情况是十分困难的。

让我们用一个例子说明交互处理和自动处理之间的差别。假定你要在终端上检查一个文件，同时你回想起文件的名称可能是 ‘groc.lst’，也许是 ‘grocer.l’，或其它和这相似的名称。同时你也记起文件可能是在目录 ‘data.lst’ 里，也可能在你当前目录里，或在目录 ‘groc.proj’ 里。在交互处理情形下，很容易通过浏览

目录，找出文件所在位置，然后使用 `cat` 程序在终端上显示这个文件。但要用自动处理时，要写一个程序找出并显示这个‘`gro...`’文件却是十分困难的。你必须考虑什么是确定这个文件位置的一个充分条件？如果没有（或多个）文件被定位，那么程序应该做些什么？一个定位及打印‘`gro...`’的程序必须回答上述及其他有关问题。

交互过程是一个简单的命令串，而要能产生同样结果的程序却是一个复杂的逻辑结构。经验表明，利用重复命令及测试条件的那些高级控制结构更易于写出好的程序，UNIX 系统 shell 包括了这些控制结构。经验也已表明，在一个程序运行时，具有其值可以改变的项（称作变量）是十分有用的。UNIX 系统 shell 包含了变量。

UNIX 系统的第 6 版本和第 7 版本之间的主要差别之一是 shell。第 6 版本的 shell 是个良好的交互命令解释程序，但它是一个功能较弱的程序设计语言。第 7 版本 shell 将第 6 版本的交互特点和一个功能较强的程序设计语言结合在一起。在这里讨论的特点是选自第 7 版本的 shell，它通常叫做 Bourne shell，因为它的作者是 S. R. Bourne。

这一章的前五节对多数 UNIX 系统用户是有用的。如果你希望写 shell 程序或者希望真正了解更多的 shell，那么你应该阅读本章的其它部分。如果你要看 shell 程序的一些例子，应该阅读第十四章。许多 UNIX 系统用户并不写复杂的 shell 程序，也就不需要了解本章所提及的那样深的 shell 了。

13.1 执行 shell 程序

任何一个命令或存放在一个文件里的 UNIX 系统命令串，称为一个 shell 程序或一个命令文件。通常，当一个文件包含一串简单命令时，就称为命令文件；当一个文件包含更为复杂的命令组合时，就称为 shell 程序（常常使用 shell 的条件命令和其它

高级特性)。

有三种方法让 shell 执行命令文件：第一种方法十分简单，可用输入重新定向，让 shell 从一个文件中读入命令。如

```
sh < lsdire
```

这里 shell 从文件 ‘lsdire’ 中读入命令串（在这个例子中，存放在 ‘lsdire’ 里的 shell 命令串的功能并不重要）。通常，由标准输入交互地读入命令所进行的工作，可改由一个命令文件经输入重新定向来进行控制。

第二种方法是把命令文件作为 shell 命令 sh 的命令行自变量来引用。如

```
sh lsdire
```

这里 shell 命令 sh 将接收一个文件名为 ‘lsdire’ 的命令行自变量，并从这个文件中读入命令串。你以后会看到，自变量对 shell 程序是很有用的。例如命令

```
sh lsdire /bin /etc
```

将使 shell 执行文件 ‘lsdire’ 中的命令，而 ‘lsdire’ 的 shell 程序能存取自变量 “/bin” 及 “/etc”。

许多命令可按上述两种方法构成，你可以将一个文件名指定为命令行的自变量或使用输入重新方向。不过，并不是所有命令均允许用这两种表示方式。

第三种方法更简炼，它先用改变文件的存取方式的命令（见第 9.3 节），为命令文件 ‘lsdire’ 建立允许执行的许可：

```
chmod a+x lsdire
```

一旦允许执行的许可建立，只要打入这个命令文件名，就可以执行这个文件中的命令。比如，

```
lsdire
```

若命令文件没有允许执行的许可，则 shell 将会受挫。通常 UNIX 系统的正文编辑程序形成的文件是不能执行的，这是因为许多正文文件包含了字母和资料，而每当 shell 遇到一个可执行的正文

文件时，shell就认为它包含了一个shell程序。

第三种方法的一个优点是只需送一个文件名作为一个命令，就可以执行一个 shell 程序。这样，对待 shell 程序与对待其它任何程序没有什么不一样。另一优点是 shell 将在全部可用的目录里（通常是 ‘/bin’， ‘/usr/bin’ 等）查找可执行命令。而当你采用前两种方法打入命令时（如“sh lsdire”或“sh< lsdire”），这个命令文件必须驻留在当前目录里（或用路径名加以说明）。

对于第三种方法，也可将自变量传递给这个可执行的命令文件，这和把自变量传递给普通程序是一样的。比如，

```
lsdir /usr/lib /usr/man/doc
```

打算反复使用的 shell 程序最好按第三种方法处理，这样，用户使用起来比较方便。

13.2 shell变量

一个程序设计语言使用变量来存放值。之所以称为“变量”是由于它所存储的值在执行过程中是能够改变的，UNIX 系统 shell变量能存储正字符串，也能使用赋值命令予以置值：

```
ux = u. UNIX
```

上面列出的赋值命令将值“u. UNIX” 赋予名为“ux”的 shell 变量。一个 shell 变量名字必须以字母开头，它可以包含字母，数字和下划线。由于“ux”是一个变量，因此它的值可使用另一个赋值语句加以改变：

```
ux = UNIX
```

如果赋给变量的值包括了空格或制表符或换行符，那么需要把值放在引号内

```
hero = "John Paul Jones"
```

可用 readonly 命令标记一个变量的值是不可改变的。例如用readonly 命令标记变量ux：

```
readonly ux
```

则变量“ux”只允许读出，不允许改写。而命令

```
ux = uu. UNIX
```

将产生出错消息“ux: is read only.”在 shell 的生存期间内，如要变量值不变，则最好使用 readonly 来标记它。如果你要列出当前只读变量，可打入下列命令：

```
readonly
```

除非给变量以移出标记，否则所建立的变量只能用于当前的 shell，已被标记为移出的变量可用于由 shell 建立的任一命令。

命令

```
export ux
```

将“ux”标记为移出，直到 shell 终止前，移出标记有效。你打入下面命令后，能得到一个当前可用的移出变量表

```
export
```

当你给一个变量赋值或给定方式时，这时使用的是变量的名字，可是当你使用存放在这个变量里的值时，你必须在这个变量名字前放一个“\$”符号。“\$”符号通知 shell，下面的名字表示变量而不是文件或字母。echo 命令能用来显示某些变量的值。

命令

```
echo $ux
```

将回应字“UNIX”，因为变量 ux 的值是“UNIX”。不曾明确地置过值的变量仅包含一个空字符串。因此命令

```
echo $abc
```

将没有任何东西回应。

当一个变量名字的后面紧接一些字符，而这些字符又不是这个名字的一部份时，应该用花括号或双引号将变量名括起来，使变量名和后面的字符分开。命令

```
echo ${ux}tm
```

将回应字“UNIXtm”。命令

```
echo "$ux"tm
```


也将回应字“UNIX™”。阅读第13.5节有关引号的解释，你就可以了解为什么有同样结果。

有五个shell变量自动地由shell置值：

1. 变量\$? 最近执行的命令返回的值。
2. 变量\$\$ shell的进程号码。
3. 变量\$! shell引用的最近后台进程的号码。
4. 变量\$# shell自变量（位置参数）的个数。
5. 变量\$- shell标志位。它在被引用时，是由shell传递来的，或由set命令设置的。

这些由shell自动置值的变量和由用户建立的变量的使用方法一样。命令

```
echo $$
```

将印出shell的进程标识号。可用ps命令来验证这个进程标识号。

13.3 交互地使用shell变量

shell变量除了在shell程序中作为存放数据或置值外，对简化交互使用也是十分有用的。假定有一个目录（我们用‘/usr/td/c/mon/src/doc’表示）包含了你将使用的一些文件，由于一些原因，你决定仍工作在自己的当前目录里，而在访问‘/usr/td/c/mon/src/doc’里的文件时，使用绝对路径名。打入涉及‘/usr/td/c/mon/src/doc’目录的命令，显然是笨拙的，因为这个命令太难打了。你可以通过将目录名存放在一个变量内，然后打入变量名来代替整个目录名，以简化这一过程。

```
docdir = /usr/td/c/mon/src/doc
```

可打入下面命令来验证你是否正确地打入了目录名

```
echo $docdir
```

可用下面命令得到该目录中的文件清单

```
ls $docdir
```

也可用下面命令显示该目录中的一个特定文件(如‘sema.txt’)

```
cat $docdir/sema.txt
```

为产生其名字以“.txt”结尾的全部文件的长格式清单, 你可以使用命令名生成功能

```
ls -l $docdir/*.txt
```

如果你要执行一个常驻在目录 \$docdir 里的程序 ‘mkdoc’, 那么你可以设立一个新变量来存放 ‘mkdoc’ 的绝对路径名。你可打入下面命令

```
mkdoc = /usr/td/c/mon/src/doc/mkdoc
```

或者用更短的命令

```
mkdoc = $docdir/mkdoc
```

来建立 “mkdoc” 变量。现在命令

```
$mkdoc $docdir/sema.txt
```

将执行带有自变量“/usr/td/c/mon/src/doc/sema.txt”的程序 ‘/usr/td/c/mon/src/doc/mkdoc’。采用变量有时能简化通常的交互任务。

13.4 查找路径

当你送入一个命令时, shell 做的第一件事是查找程序, 可是到哪儿去找呢? 许多 UNIX 系统包含了成千上万个目录, 因此查找每一个目录需花费很多时间。UNIX 系统 shell 将查找问题集中在管理查找路径上。查找路径实际是 shell 寻找命令用的一串目录。

多数查找路径包含了当前目录、目录 ‘/bin’ 及目录 ‘/usr/bin’。在 UNIX 系统里习惯将许多重复使用的命令存放在 ‘/bin’ 及 ‘/usr/bin’ 里。查找路径能够修改, 从而使得每当打入一条命令时可在另外的目录里查找。如果在查找路径的任一目录里均未找到该命令, 则 shell 会印出一个消息说明该程序未找到。

你可通过打入下面命令来显示当前查找路径:

echo \$PATH

它的回答可能是：

```
:/bin:/usr/bin
```

这个查找路径说明先查当前目录，然后查目录‘/bin’，最后查目录‘/usr/bin’。查找路径中的各个目录用冒号分隔。查找路径中的任何空目录（一行内连接两个冒号或一个开头为冒号）系指当前目录。说明先查‘/bin’，再查‘/usr/bin’的查找路径是

```
/bin:/usr/bin
```

如果你要先查‘/bin’，再查当前目录，最后查‘/usr/bin’那么查找路径是

```
/bin:./usr/bin
```

由于查找路径正好是一个 shell 变量，因此，你能在终端上通过命令置值构成查找路径。命令

```
PATH = :/bin:/usr/bin:/usr/kc/bin
```

将构成一查找路径。它包含了所有常用的目录及目录‘/usr/kc/bin’。

如果你的 login 目录里包含了名为‘.profile’的文件，那么当你注册时，该目录里的 shell 命令将被执行。在‘.profile’里的命令通常用于使终端处理程序适合于你的终端。为 \$PATH 那样的环境变量置初值，并且执行任何你要的命令。如果你要一个独特的查找路径，那么你应在文件‘.profile’中放入适当的命令。

查找尽可能少的目录并且按最适宜的次序来查找，这两点最重要。使用的查找路径越长，系统响应就越差，因为每打入一个命令，shell 就要忙于查许多目录。因此，一个重要的事情是：应使目录的大小易于处理。查找一个只有少量文件的目录比查找有几百个文件的庞大目录要省很多时间，shell 花费在查找命令上的时间是值得注意的，仔细地控制查找路径和目录大小能减少这个时间。

当一组人员都要访问某一程序体时, 查找路径是十分重要的。例如, 一组人员都使用一组 teletext 程序, 于是, 全部程序应放在一个目录(如`/usr/teletext/bin`)里。每一个使用这个程序的人应该修改他们的查找路径使其包含 teletext 目录。相反如果将这个程序放在`/bin`或`/usr/bin`里。那么系统的每个用户将经常地受到惩罚, 因为通过标准目录寻找要花费很长时间, 所以执行每一命令都需很长时间。这是一个用户有能力控制他们的环境以便建立一个有更高生产效率的环境的例子。

13.5 引 用

很遗憾, 许多 shell 所使用的特殊符号也为其它程序所使用。字符和符号很不够用。当你交互地打入命令或者你将命令放在文件里并执行这个文件时, shell 是获得这个信息的第一个程序。如果信息内包含了 shell 的任一特殊符号, 那么除非用引号把它括起来, 否则 shell 会改变这一信息的意义。

当你打入一个带有管道的命令

```
ls | wc -l
```

时, 这个竖线表示管道操作, shell 产生两个进程 ls 及 wc, 同时 ls 的标准输出与 wc 的标准输入相连接。当你象下面那样, 打入一个“在两个 shell 变量间进行逻辑比较”的命令时。

```
expr $var1 \ | $var2
```

这儿的竖线是一个数学符号, 而不是管道操作。当竖线用作数学符号时, 我们必须十分小心地用特殊标记引用它, 这样它就失去管道的意义了。在上面这个例子中, 我们在竖线的前面用一反斜字符来引用。

在 UNIX 系统 shell 中, 有三种引用方式:

1. 用反斜线 (\) 引用它后面紧跟的字符。
2. 用单引号 (') 将字符串括起来, 不作任何翻译。
3. 可用双引号 (") 把除了反斜线, \, 双引号, \$ 以外

的字符括起来。双引号内的内容用作命令及参数替换。

当必须引用单个字符时，像上面例子所示用一个反斜线最容易。当必须引用一组字符，而且又不作任何解释时，使用单引号最容易，它是UNIX系统 shell 中引用时的最好方式。双引号标记引用较少，在双引号内的内容产生命令及参数替换，假如你要阻止这种替换，那么控制命令及参数替换的字符（\, \$）需要加上引用符号。在双引号之内的反斜线和双引号还是特殊的。

如果你打入命令

```
echo '$HOME'
```

或者类似的命令

```
echo \" $HOME\"
```

则字“\$HOME”将印出，但如果你打入命令

```
echo "$HOME"
```

则会把你的主目录的名字印出（见下一节）。

引用也用于将一串带有空白的字符当作一个单词。如果你打入命令

```
macedonian = Alexander the Great
```

则将把值“Alexander”赋予 \$macedonian，并且 shell 将报告在定位名为“the”的程序时有错。代替上述命令，你可打入命令

```
macedonian = 'Alexander the Great'
```

或

```
macedonian = "Alexander the Great"
```

或

```
macedonian = Alexander\ the\ Great
```

上面三种命令都能将值“Alexander the Great”赋予 shell 变量 \$macedonian。

13.6 set 命令

set命令通常用于印出一个已经建立的变量表。命令

```
set
```

在我的系统里将产生下列输出：

```
HOME = /usa/kc
PATH = ./usa/kc/bin:/bin:/usr/bin
IFS =

PS1 = $
PS2 = >
TERM = vt100
```

上面这些变量是注册过程中建立的，它们是UNIX系统环境的一个重要部分。有一些系统还包含一些附加变量以支持系统本身的局部特性（注意：set命令不能列出自动变量 \$\$, \$?, \$!, \$#, 及\$-）。下面让我们分别讨论上面的每个变量：

1. “HOME”变量是你主目录的名字。
2. “PATH”变量是 shell 查找命令时的查找路径。查找路径在第13.4节中讨论。
3. “IFS”变量是内部字段的分隔符，通常是空格、制表符或换行符。这个内部字段的分隔符将命令按字段分开。
4. “PS1”及“PS2”是 shell 使用的基本提示符。“PS1”是 shell使用的正常提示符，“PS2”要求额外的输入，以便完成那些明显是不完整的命令。
5. “TERM”变量是终端的型号，有些命令需要知道所使用的终端型号，以便产生正确的输出。

当一个交互的 shell 开始运行时，它首先读入并且执行用户主目录下的文件‘.profile’中的命令。典型地，使用‘.profile’

文件中的stty命令去调整系统的终端配置，也调整某些变量值以适合你的选择。例如，‘.profile’文件有下列赋值命令

```
PS1 = "Yes boss->"
```

则系统将用“Yes boss->”提示符，而不是用标准提示符来询问你。当然你也可交互地对PS1重新赋值（或者对其它自变量重新赋值）。

set命令也能用于控制shell的几个内部方式。例如，命令set -v

将使shell使用“详细”(verbose)方式。当用shell输入命令读入时，输入行将全部被印出。“详细”方式可用下面命令废除：

```
set +v
```

当使用shell去执行文件中的一系列命令时。可用set命令控制的许多标志自变量也能作为命令行的自变量。命令

```
sh -v lsdire
```

将使用“详细”方式调用shell去执行文件‘lsdire’中的命令。若命令“set -v”是‘lsdire’中的第一个命令，则效果是一样的。

使用set命令或命令行自变量来控制的shell任选项，其完整的清单可参见UNIX系统手册。

13.7 简单的条件

决策能力是智力的标志。决策包含了在任选项中作出选择。路径的选择应比其它更为优先。在一个程序中作出决策后，一串命令就被执行，而其它命令串却不予理会。在一个程序设计语言里作出决策的结构称为条件(condition)。

一个系统的基础是系统内的操作。在计算机程序设计语言(如BASIC及FORTRAN)中，其基础是对各种表达式(包括基于布尔代数的条件表达式)和语句的操作，基本操作的结果决定了控制流方向。shell命令程序设计语言的基础是UNIX系统的实用程序。因此，在shell命令程序设计语言中，程序执行的成败

决定了控制流方向。

若 UNIX 系统程序执行成功，则它送回一个“0”出口状态；若程序执行遇到严重问题，则它送回一个“非 0”出口状态。假定在我的系统里打入一个命令

```
cd /usa/kc
```

使我的主目录成为当前目录，cd 的出口状态将是“0”。可是假定在你的系统里打入同样命令，shell 将可能回答“/usa/kc: bad directory”（因为你的系统可能没有这个目录），这时，cd 的出口状态将是“1”。一个管道线的出口状态是它最后一个命令的出口状态。从一个命令（或管道线）送回的出口状态，控制 shell 程序的执行方向。

许多系统有专门程序 true 及 false。程序 true 的唯一功能是送回一个 true (0) 的出口状态。同样程序 false 的唯一功能是送回一个 false (0) 的出口状态。

Bourne shell 有几个条件运算符，最简单的是 (“&&”)。在两个命令用 “&&” 条件运算符连接时，仅当第一个命令送回“0”出口状态时，第二个命令才执行。打入命令

```
test -d /usa/kc && echo success!
```

如果文件 ‘/usa/kc’ 是一个目录，则此命令将印出消息 “success!” test 程序用于测试不同条件。“-d” 是标志自变量，它说明测试应该判断后面的自变量是否是目录。其他不同条件也能被测试，请参见 UNIX 系统手册中 test 命令的叙述。

与 “&&” 条件运算符相反的是 “||”。在两个命令由 “||” 条件运算符连接时，仅当第一个命令送回非 0 出口状态时，第二个命令才执行。打入命令

```
test -d /usa/kc || echo failure!
```

若该文件不存在，或虽存在但不是目录，则此命令将印出消息 “failure!”。

13.8 简单命令、管道线和命令表

我们已经定义了由命令和它的自变量构成的简单命令。UNIX 系统 shell 还有两种其它结构类型：管道线和命令表。由于两者用于 shell 的控制结构，因此应该了解它。

管道线是由管道符号(|或 \wedge)连接的一个或一组简单命令。下面是一些管道线：

```
ls -l /bin /usr/bin
who | wc -l
a^b^c^d
ps
```

在 UNIX 系统中，一串管道线是一个命令表。因此刚才提到的四个管道线组成一个命令表。上面命令表的元素（管道线）分离在各行中（用 UNIX 系统的术语可说成管道线被换行符分开）。下列命令表和上面的命令表是等价的：

```
ls -l /bin /usr/bin ; who^wc -l;a|b|c|d ; ps
```

在这个命令表里，元素由分号分开。下列字符能用来分隔命令表的元素：

1. ; 或换行符 表示顺序执行。
2. && 表示根据条件 (true) 执行后面的管道线。
3. || 表示根据条件 (false) 执行后面的管道线。
4. & 表示前面的管道线在后台 (异步) 执行。

命令表是 UNIX 系统的一个基本结构。一个命令表可以简单到仅是一个命令，也可以复杂到随你怎样构造它。一个命令表送回的值是表中最后一个管道线的出口状态。了解命令、管道线和命令表之间的区别很重要。

- 1、一个简单命令执行一个程序。
- 2、一个管道线是一串由管道符号连接的简单命令。最简单的管道线是一个简单命令。
- 3、一个命令表是一串管道线。最简单的命令表是一个管道线（它可以是一个简单命令）。

13.9 if条件

&&条件和 | 条件对于建立十分简单的条件结构是很有用的。然而，shell有许多更高级的条件。shell最重要的特性之一是 Bourne shell的if条件，它对第6版本中if条件，作了很大的改进。if条件的语法如下：

```
if if-list
  then then-list
elif elif-list
  then then-list
else else-list
fi
```

字“if”，“then”，“elif”，“else”，及“fi”是关键字。关键字是shell（或任何程序设计语言）用以表示像if条件语句那样的内部结构的单词。字“if-list”，“then-list”，“elif-list”及“else-list”是UNIX系统的命令表。“elif..then..”部分是可任选的，“else”部分也是可任选的，必要时也可以有许多“elif..then..”部分。因此，最简单的if条件是

```
if if-list
  then then-list
fi
```

UNIX系统shell的if条件所起的作用与许多程序设计语言中if语句类似。我们用一简单例子来展示if语句是如何工作的。

假想有四个名为 winter, spring, summer及 fall 的程序，当处在它们的季节时将送回一个 true 出口状态，否则送回 false。同时假想有一组程序应该印出每个季节里应做的家务事。这个印出家务事备忘录的 shell 程序如下所示：

```
if winter
  then
    snowremoval
    weatherstrip
elif spring
  then
    startgarden
    mowlawn
elif summer
  then
    tendgarden
    painthouse
    mowlawn
    waterlawn
elif fall
  then
    harvest
    mowlawn
else
  echo Something is wrong.
  echo Check the 4 season programs
fi
```

在春季里，spring 命令为 true，并执行“startgarden”命令及“mowlawn”程序。在秋季里，fall 命令将送回一个 true 状态，并执行“harvest”命令和“mowlawn”命令。如果四个季节的程序出口均不是 true，则条件语句的 else 部分将被执行，并印出一个出错消息。

下面让我们举一个更真实的例子。假定有某个连续运行的程

序，每当操作中遇到错误时，它将把诊断消息写到文件‘errorfile’中。另一个程序每小时运行一次去记录这些错误。如果错误文件存在，则后一个程序将‘errorfile’在总部的行式打印机上复制输出；如果‘errorfile’不存在（因为没有错误产生），则后一个程序将送出一个“all is well”的消息。下面的 shell 命令文件将实现这个简单任务：

```
date > /dev/lp-to-hdq
if test -r errorfile
then
    cat errorfile > /dev/lp-to-hdq
    rm errorfile
else
    echo "No errors this hour" > /dev/lp-to-hdq
fi
```

（第二行的文件名应为 errorfile，原文错一译者注）

要注意，date，cat，及 echo 将把他们的标准输出重新定向到总部的行式打印机设备上。我们可以认为‘errorfile’将被复制或移动到一个专门的行式打印机接口文件中。可是复制（cp）或移动（mv）是磁盘操作，它把一个普通文件从磁盘的一处移动（或复制）到另一处。因此要复制一个特别文件时，需要用重新定向输出，使得执行中的程序把数据写到特别文件中。当程序把数据写到特别文件中时，实际上操作系统把数据送到输入/输出设备（本例中就是行式打印机）上。

13.10 shell 程序变量

我们已经看到实现通用功能的程序的重要性。许多实现通用功能的程序通过在命令行中提供一些自变量，控制它实现更专门的功能。例如命令

```
ls
```

将输出当前目录中的文件清单。如果你要更专门的清单，诸如目录 ‘/bin’ 的全部文件清单，你必须打入更专门的命令。命令

```
ls /bin
```

将输出目录 ‘/bin’ 中的全部文件清单。用自变量 “/bin” 引导程序 ls 去注意目录 ‘/bin’

shell 程序中的命令行自变量可以使用带编号的变量。\$1 是命令行的第一个自变量，\$2 为第二个自变量，等等。编号变量常常称作位置参数，所以 \$1 表示第一个位置的自变量，\$2 表示第二个位置的自变量，等等。专门的变量名 \$0 总是表示第 0 个自变量，它是正在执行的 shell 程序名。另一专门名字 \$#，它表示命令自变量的个数（见第 13.2 节）。

举一个使用简单的位置参数的例子，假定你需要一个程序来获取四个变量，并以逆序打印这些变量。如果程序（称作 ‘rev-4’）按下面命令引用：

```
rev-4 20 30 40 50
```

它将回应 “50 40 30 20”。我们用程序检查自变量的个数，若自变量的个数少了或多了，则打印一个出错消息。如个数是对的，则按逆序打印这些自变量：

```
if test $# = 4
then echo $4 $3 $2 $1
else echo $0 usage: arg1 arg2 arg3 arg4
fi
```

此处，我们使用简单的 shell 程序 test 命令来检测自动置值变量 \$# 是否为 4，我们用位置参数 \$1, \$2, \$3 及 \$4 表示实际自变量。某些 UNIX 系统版本，为方便起见，允许用方括号把表达式括起来，让 test 命令测试表达式的计算结果。在这些系统里，上面例子中包含 if 条件的那一行可以等价地写成 “if [\$# = 4]”。方括号经常使程序更易读，在这两种情况下，test 程序实际上均是

执行同一操作（注意UNIX系统的expr命令也用于实现算术比较）。

为了使用位置参数，for语句（见第13.16节）提供了一个更灵活的方法。

13.11 while和until语句循环

while和until语句允许你重复一组命令。首先让我们考察while语句。它的语法是：

```
while while-list
do do-list
done
```

这里的关键字是“while”，“do”，及“done”。首先执行while-list。如果它送回一个true出口状态，则执行do-list，然后从起始处重新开始操作。如果while-list送回一个false出口状态，则语句就算执行完了。

假定你必须写一个等待某文件被删除的shell程序（另外有程序负责删除文件），我们可用while命令等待一个条件变为true。由于我们还要让其它UNIX系统用户得到一些处理时间，因此我们应在两次连续测试之间间隔几秒而不是不断地测试，下面的shell程序是等待一个名叫‘lockfile’文件消失的程序：

```
while test -r lockfile
do sleep 5
done
```

这个程序检查名为‘lockfile’的文件是否可读。如果可读，则命令“sleep 5”中止执行5秒，即等待5秒后，再去执行下次测试。当‘lockfile’不可读，则测试失败而命令完成。

注意，前述的shell程序，是通过将命令放在分离的各行中来分隔命令表的。我们也能在一行里送入这个程序，使用分号来

分隔命令表：

```
while test -r lockfile ; do sleep 5 ; done
```

`until`语句是 `while` 语句的一个变种。当 `while-list` 送回一个 `true` 值时，`while` 语句一直重复执行。与此相反，当 `until-list` 送回一个 `false` 值时，`until`语句一直重复执行。`until`语句的语法是：

```
until until-list
do do-list
done
```

这儿的新关键字是“`until`”。

假定你必须写一个 shell 程序，它一直等待到一个文件建立为止。一种方法是使用 `while`语句及否定的测试。

```
while test ! -r proceedfile; do sleep 1; done
```

感叹号用于否定可读性测试：如果文件不可读，则 `test`送回一个 `true`值。另一种方法是使用 `until`语句：

```
until test -r proceedfile ; do sleep 1; done
```

使用这个方法，循环将继续直到 `test`命令送回一个 `true`值（即直到‘`proceedfile`’建立）为止。

13.12 结构化的命令

像 `while`和 `until`这样的语句结构，几乎被 shell 作为单一的命令来执行。在它们的任一部分执行前，整个结构被 shell 先扫描。在你的系统里，请交互地打入下列命令：

```
until test -r stopfile ; do
```

该命令显然是不完整的，因此 shell 将提示你进一步输入（一般用“`>`”）。请再打入下面行以完成命令：

```
echo Hello ; sleep 2 ; done &
```

最后的“&”符号表示整个 until 命令应在后台运行。该命令每二秒将打印一次“Hello”，直到你建立了一个‘stopfile’为止。因为你的命令将运行在后台，shell将准备接收其它命令。当你看终端看得累时，你可打入下列命令，自己去建立一个名叫‘stopfile’的文件：

```
>stopfile
```

因为打入命令的时间可能比两秒长，因此在你按键时可能被一两个“Hello”所中断。当系统周期地显示“Hello”时，系统仍将保持你所打入的字符串的结构。一旦你成功地建立了‘stopfile’，后台的进程将终止（为了不让目录变得杂乱，你应该删除文件‘stopfile’）。

所有 Bourne shell 语句均是结构化的命令。在这种结构里，所有可选的执行路径都是该语句的一部分，控制流向只局限于该语句。这种语句的对立面是 goto 语句，在 goto 语句里，可选的执行路径并不属于语句部分，控制流向可到达外部任一语句。

近代程序设计语言喜欢用没有 goto 的结构化语句。goto之所以不受欢迎是因为它的任意转向会导致不易理解的程序。Bourne shell 中没有 goto 语句，这是因为 goto 语句可能违背一次只处理一个命令（或一个结构化命令）的协定。

13.13 命令替换

shell总是让你得到你的命令的一个标准输出，并在 shell 过程中使用这个标准输出。当一个命令被\`括起来时，该命令就为 shell 所执行，其结果被替换。例如，你能将当前的日期及时间存在一个名叫now的变量里，即执行下列命令：

```
now = `date`
```

可以认为这个进程分为两步。第一步，date程序被执行，第二步，执行结果被替换。从概念上我们认为得到了 shell 替换命令“now = ‘09:30 Jan 1, 1980’”。这个命令执行后，将使结果存

入变量\$now里。命令

```
echo $now
```

将印出存在 shell 变量\$now里的日期及时间。

命令替换经常用于 shell 变量的算术操作，expr命令能在它的自变量之间实现算术操作。如果你送入命令

```
expr 5 + 13
```

在终端上将印出“18”。用于表达式的许多运算符（括号，星号，&等）是 shell 的专门符号，因此必须十分小心地去避开它们（见第13.5节）。如一个 shell 变量被赋予一个数

```
count=10
```

那么，使用命令替换

```
count=`expr $count + 1`
```

能使该值加1。expr命令将接受自变量“10”，“+”，及“1”。在 shell 变量\$count内将保存结果“11”。

我们已能用变量做算术操作了，现在我们再写出逆序印出自变量的程序（在第13.10节提到）：

```
count=$#
cmd=echo
while test $count -gt 0
do
    cmd="$cmd \$$count"
    count=`expr $count - 1`
done
eval $cmd
```

由于这个程序使用了 shell 的许多技巧，因此理解它可能会有一些困难。这个程序包含一个 while 条件循环，它对每个自变量执行一次，先从最后一个自变量开始，顺序降至第一个自变量，被处理的自变量由变量\$count所指示，变量的初始值是最后一个

自变量的号码，每通过一次循环，变量值减 1。

每通过一次循环，在变量 \$cmd 中将加入一些正文。开始变量 \$cmd 是正文 “echo”。如果程序包括了四个自变量，在第一次通过循环后，变量 \$cmd 将是 “echo \$4”，第二次以后变量将是 “echo \$4 \$3”，等等。当变量 \$count 的值减至零时，循环将终止。

do list 的第一个语句中的字 “\\$\$count” 值得注意。它的目的是产生一个字符串，它包含了一个 \$，并在后面紧跟变量 \$count 的当前值，由于 \$ 是 shell 的专门符号，因此要用反斜线转义，这是第一个反斜线和第一个 \$ 的解释。第二个 \$ 正好是变量 \$count 的开始字符。

程序最难的部分是以字 eval 开始的那一行。eval 是一个专门命令，它提供自变量的多层替换，并执行这些自变量。在循环完成时，shell 变量 \$cmd 中存储的是输出自变量逆序表的 echo 命令。对于四个自变量的情形，shell 变量 \$cmd 是 “echo \$4 \$3 \$2 \$1”。eval 将使 echo 命令重新定值并执行，用逆序来输出自变量的内容。

如果这个程序存放在名叫 ‘revargs’ 的文件里，使用 chmod 命令使 ‘revargs’ 变成可执行，于是命令

```
revargs n o t
```

将输出 “t o n”。revargs 用于迴文(顺读和倒读都一样的字)并不很好，但是，必须调整自变量顺序的那种情况还是用它。

13.14 shell 替换

到此我们已经讨论了 shell 实现的所有替换。现在按它们出现的次序总结一下全部替换。当一字适合几种替换的情形时，替换的次序是十分重要的。

1. 命令替换 由 \ 括起来的全部命令被 shell 执行，其结果正文被替换。见第 13.13 节的讨论。

2. 参数（或变量）替换。 在一个程序中，全部以\$开头的字是变量，它们将被替换。见第13.2节的讨论。

3. 空格解释。 为寻找字段分隔符，就要扫描前面替换后的结果正文。通常字段分隔符是空格、制表符和换行符。带有字段分隔符的任一字将分成许多字。被引号括起来的字中，若包含字段分隔符，则不予理会。空格（除了明显带引号的空格外）将被丢弃。

1. 文件名生成。 为寻找元字符“*”，“?”，及“[”，shell将检查每一个字。如果任一字中包含了上述元字符，则该字将被匹配这种模式的一组文件名所替换，这些文件名按字母次序排好序；如果不匹配，则仍是原字不变。文件名的生成见第4.8节的讨论。

shell的较强能力在于它能实现正文替换，可是这种技术对于习惯于程序设计语言 FORTRAN 和 BASIC 的人来说，则是易混淆的，因为那些语言都是面向数值的。shell的这种替换像已经在程序设计语言 LISP 和 SNOBOL 所具备的那样，与一般正文处理功能更相似。

13.15 here文件

here文件用于把 shell 程序中的标准输入暂时重新定向。here文件的写法类似于标准输入的重新定向。

```
cmd args <<symbol  
( the here document )  
symbol
```

here文件一开头由“<<”指示。它后面的符号（symbol）要记住，因为它要用于指示 here 文件的结束。here文件根据需要可以是多行的。包含符号（symbol）的那行指明here文件的结束。shell使here文件可作为命令的标准输入之用。

考虑在半夜运行的大型 shell 程序的错误处理问题。你可用 shell 程序把出错消息写到一个文件里，并送给每天负责看文件的人员。一个较好的方法是送一个信件给负责人员，这种通知是自动进行的。

如果 shell 程序的出错处理程序包含了下面命令，则某人（此处是“opsmanager”）将收到叙述问题的信件：

```
mail opsmanager <<!
***** PROBLEMS AGAIN *****
The midnight error has struck again! The
tdata file was missing - all processing
stopped.
!
```

另外的一种解决方法是将消息放在一个单独文件里，然后使用普通的输入重新定向。here 文件使你能在一个文件里保留每一事件。

13.16 for 语句

UNIX 系统 shell 包含了 for 语句，它对于名字表中的每一个字，执行一次命令串。for 语句的一般形式是：

```
for name in word1 word2 ...
do do-list
done
```

for 语句对于名字表的每一个字（word1, word2, ..., 等），将执行一次 do-list。在表中的当前字将赋给 shell 变量 \$name。关键字是“for”，“in”及熟悉的“do”和“done”。

用下面例子展示 for 语句比较笨拙的用法。

```
for fruit in apples pears oranges mangos
do
    echo $fruit are fruits
done
```

如果执行该程序，它将印出“apples are fruits”，接着是“pears are fruits”，等等。for语句的这一形式经常用于对一组文件或一组目录中的每一个执行某些功能。

没有关键字“in”及后面名字表的for语句也能使用。

```
for name
do do-list
done
```

在这种形式里，for语句对于shell的每一位置自变量，将执行一次do-list。这是顺序地将自变量给予shell程序的最简单的方法。

我们可用for语句的第二种形式重写逆序印出自变量的shell程序。

```
list = ""
for arg
do
    list = "$arg $list"
done
echo $list
```

在这个逆序印出自变量的shell程序里，for语句顺序地通过shell，将每个自变量放在以前自变量的前面，并存放在变量\$list里。在程序结束时，将印出这个逆序表。

13.17 case 语句

shell 的 case 语句是一个奇特的、基于模式匹配的多路分支结构。case 语句的一般形式是：

```
case word in
    pattern1) pat1-list ;;
    pattern2) pat2-list ;;
    ...
esac
```

word 将和所有模式进行比较，最先的匹配将使相应的模式清单被执行，执行以后语句即完成了。常用的 shell 元字符可用来组成模式：“*”可适合任何字符串，“?”可适合任何单字符，方括号将划定一类字符的边界。当多个模式对应同一个模式清单时，可用竖线表明它们是可供选择的。在 case 语句里，竖线的常用含义（管道线符号）没有了。下面的 shell 程序是一个试图决定一个动物的品种的程序：

```
for breed
do
    case $breed in
        arabian|palomino|clydesdale) echo $breed is a horse ;;
        jersey|guernsey|holstein) echo $breed is a cow ;;
        husky|shepherd|setter|labrador) echo $breed is a dog ;;
        siamese|persian|angora) echo $breed is a cat ;;
        *) echo $breed is not in our catalog ;;
    esac
done
```

注意，最后的模式是“*”，它将适合每一品种。每经过一次 case

语句，只能执行一个模式清单。如果该程序存放在一个可执行的文件 ‘breeds’ 里，则命令

```
breeds husky holstein terrier
```

将产生消息“husky is a dog”, “holstein is a cow”, 及“terrier is not in our catalog”。case 语句经常和for 语句连在一起使用。

13.18 break和continue

shell的 break和 continue 语句经常用于改变 for, while 和 until 循环的活动。break 语句将使 shell 走出整个循环，而 continue语句将使 shell 重新转到整个循环的开始，并进行另一次循环。

作为例子，先重写逆序印出自变量的 shell 程序。我们回忆一下使用while语句的这个程序：

```
count = $#
cmd = echo
while test $count -gt 0
do
    cmd = "$cmd \$$count"
    count = `expr $count - 1`
done
eval $cmd
```

这个程序也可使用break语句写成：

```

count = $#
cmd = echo
while true
do
    cmd = "$cmd \$$count"
    count = `expr $count - 1`
    if test $count -eq 0
    then break
    fi
done
eval $cmd

```

在这个程序里，由于只有一个循环出口条件及一个出口点，所以 `break` 语句并未提供很多优点。当一个循环有几个出口点，或者当出口条件十分复杂时，`break` 能提供仅有的更为清晰的解决办法。我们使用 `continue` 语句再重写这个问题的程序：

```

count = $#
cmd = echo
while true
do
    cmd = "$cmd \$$count"
    count = `expr $count - 1`
    if test $count -gt 0
    then continue
    fi
    eval $cmd
    exit
done

```

循环的最后两句（“`eval $cmd`”及“`exit`”）仅在程序结束时执行一次。在所有通过循环的头几遍里，这两句被跳过了。在一个 shell 程序里，`exit` 语句将终止处理。当交互地打入该语句时，通常是起不了什么作用的。

第十四章 一些shell程序

UNIX 系统中最有趣的特性之一是 shell。shell 是与复杂的高级程序设计语言混合成一体**的强有力的交互命令解释程序**。UNIX 系统命令是 shell 程序设计语言里的基本操作。因此 UNIX 操作系统的全部功能都能在 shell 程序里使用。

通常都把程序员训练成用普通的程序设计语言来工作的人。可是, shell 程序设计却需要一些不同的思路。由于 shell 对于 UNIX 系统是太重要了, 因此, 有必要在下面讨论一些 shell 程序的例子。所有这些例子均设计在 UNIX 系统第 7 版本上工作, 它们用到了第 7 版本的 shell 特性。

非程序员可以写些简单的 shell 程序, 但复杂的 shell 程序设计(事实上是所有的程序设计)仍是专门人员的工作。本章第一节解释了当你为一给定的应用考虑使用 shell 语言时, 应想到的某些问题。简单的 shell 程序通常采用方便地存放在文件里的 UNIX 系统命令。第 14.2 节展示了一个这种例子。其他 shell 程序概念上是简单的, 但包含了许多命令。在第 14.3 节里, 开发的记帐问题是一个很好的例子。最后, 在第 14.4 及 14.5 节, 我们可以看到使用某些 shell 程序设计语言特性的例子。

14.1 何时使用 shell 程序设计语言?

并没有固定的规则来决定什么时候用 shell 程序语言来写一个程序。若程序的执行速度很重要, 则应使用更有效的语言, 因为使用 shell 的高级特性时, 执行速度将受到很大影响。但对于一个给定的问题, 决定使用哪种程序设计语言时, 执行速度并不是唯一考虑的标准。

当一个问题的解法包含了许多UNIX系统的标准命令操作时，应该使用 shell 程序设计语言。UNIX 系统命令包括：在文件内查找、将文件排序、传送文件、建立文件、移动文件等。如果一个问题能用 UNIX 系统中已经建立的基本操作来表示，则使用 shell 程序设计语言能构成很强的功能。

一旦你能很熟练地使用 shell 进行程序设计时，你就能按照 shell 语言解题的潜力去观察问题。许多问题起初看起来似乎不易用 shell 语言来解，但实际上可以由 shell 程序很好地实现。

评价 shell 程序设计语言适用性的另一种方法是考察问题所具有的基本数据。若基本数据是正文行或是文件。则 shell 可构成一个很好的解法。若基本数据是数或字符，则用 shell 可能不是好的方法。

是否使用 shell 程序的最后一个准则是程序开发的成本。用象 C 或 FORTRAN 那种使用编译的语言来开发一个程序要花费许多钱。而交互式语言却更易测试和调试，这样对于只用一两次的程序，用 shell 进行程序开发成本是很便宜的，这既可以获得 shell 程序容易开发的优点而又能忍受它较慢的执行速度的缺点。

14.2 多少用户？

当一个UNIX系统运行十分慢，甚至毫无反应时，往往是系统有了故障，或者有许多人正在争夺计算机有限的资源。粗略地估计系统负荷的最简便的方法是观察一下有多少人正在使用这个系统。如果执行命令

```
who
```

那么将印出一个用户清单。为了决定有多少用户正在系统上工作，我们可用 wc 程序去统计 who 命令的输出行数。命令

```
who | wc -l
```

将印出用户的数目。“-l” 任选项是表明 wc 只统计行数。正常情况下 wc 可统计行数、字数及字符数。如果你想得到更明确的结果

果。则可用下面命令

```
echo `who | wc -l` users on the system
```

产生相同的数目,并在后面跟有下列消息“users on the system”。如果该命令放在文件‘nusers’里,并使它变成可执行的,则在有20个用户工作时,命令

```
nusers
```

将产生信息“20users on the system”。

用C语言如何解这个问题?用C解这个问题是困难的。因为要找出多少人使用这个系统。就必须辨别文件‘/etc/utmp’(该文件记录了运行的用户名)的内容。由于who命令已经知道如何去检查文件‘/etc/utmp’,因此使用现成的UNIX系统工具,能很容易地解决这个问题,而用C语言解决起来却相当困难,这仅仅是一个例子。

14.3 更新一个记帐文件

在我们系里(作者是洛克菲勒大学教授——译注)记帐信息存放在一个大文件里。在每个财政年度结束时,许多帐户已不用了,他们的记录必须从文件里删去,只留下少量文件以便管理。可是即使每年剔除已不用的帐户信息,这个文件还是很大的,正文约有30000行。

因为这个文件是一个普通的正文文件,解决这个问题的一个方法是使用UNIX系统的正文编辑程序手工进行删除。秘书使用编辑程序负责这个修改,估算至少要花费一周时间。由于每个帐户的记录按组分在一起,因此使用标准编辑程序修改这个文件相当容易。然而在一周之内修改一个30000行的文件,暗示了修改率大约是一小时1000个记录,或一分钟20个记录。要在每分钟决定20次记录的删除或保留,出错概率是很大的。此外,不但秘书工作的一周时间,而且在我们UNIX系统30小时的机时也是一个相当高的成本。

另一种方法是写一个专门程序去做这种修改。由于程序一年

仅运行一次，效率并不是考虑的主要因素。因为感兴趣的项是正文行，看来采用 shell 是一种很自然的解决方法。另一重要原因是，使用 shell 能减少程序开发时间。当然，如果要花一周时间写这个程序，就谈不上节省了。

解决这类问题的第一步是应决定一个专门标准以判定哪些记录应删除或应保留。秘书告诉你删去所有老记录，这是十分不够的，你需要一个十分专门的标准。在我们这里，秘书已准备了一个新年度要转到新帐户文件中的全部帐户清单。

针对秘书准备的清单中每一帐号，我们可以执行一个提取命令，找到并从帐户文件中提取全部记录，然后把它们放在一个当前帐户文件里。UNIX 系统命令 `grep` 可用于提取。一旦我们有了命令表。就可以执行它，也就是在实际上实现提取。最后一步是清除那些我们建立的、并已检验了结果的杂乱文件。

在实现操作前，先看一下两个已指定用作输入的文件——帐号清单文件和帐户文件。当前的帐号清单（文件 ‘currents’）是一组数：

```
48150
48677
56789
56790
```

上面只写了少数项。帐户文件（‘acctfile’）包含了下面信息。

```
jmx 48150 mathiasson gt 0      general maintenance
jmg 48150 mathiasson gt 0      electrometer
tmg 48150 mathiasson gt 0      general maint
jmm 48309 pickett    gt 1000 y-axis amp
jmm 48309 pickett    gt 0      y-axis amp repair
```

当然这里只写了30000行中的5行。

可以使用下面命令，来提取帐号48150 的记录：

```
grep 48150 acctfile
```

如果字符串“48150”不在帐号段而在其他段产生，此时可能得到无关的输出。因此这並不是一种绝对安全的方法——对某些应用将是不合适的。在现在这个应用中，对于那些熟悉帐户並能仔细地检查的人员，上面命令是一个合适的解决方法。

grep命令通常把输出放在标准输出上，现在我们必须重新定向输出到一个文件去。由于 grep 包括了许多我们所不需要的正则表达式的匹配，因此我们可用类似的运行更快的 fgrep 程序。请参见UNIX系统手册有关grep和fgrep的阐述。

综上所述，从老记帐文件中提取帐号为48150 的记录，並將它们加在新记帐文件的末尾，其命令如下：

```
fgrep 48150 acctfile > > nuacctfile
```

（记住，“>”表示输出重新定向后，重写目标文件，而“>>”表示输出应附加在目标文件的末尾）。

我们已有了上面所示的帐号文件，並且要将它传送到下面所示的提取命令的文件中：

```
fgrep 48150 acctfile >> nuacctfile
fgrep 48677 acctfile >> nuacctfile
fgrep 56789 acctfile >> nuacctfile
fgrep 56790 acctfile >> nuacctfile
```

最容易的方法或许是使用UNIX系统的正文编辑程序。如果实现这个任务仅仅一次，那就采用这个技术，然而，如果你要实现提取多次，那么你应该使用UNIX系统的流（stream）编辑程序。这种流编辑程序读一输入文件，并逐行对此文件进行变换。流编辑程序和标准UNIX系统编辑程序十分相似，因此你若熟悉标准编辑程序，则相对地也容易使用流编辑程序，请参见UNIX系统手册中有关流编辑程序（sed）的描述。

可用一个辅助文件或一个命令行来提供编辑原稿给流编辑程

序。对于这里很简单的变换，我们只用命令行来提供编辑原稿。由流编辑程序接受的编辑原稿十分类似于由标准的 UNIX 系统正文编辑程序接受的编辑命令。可是，控制编辑程序的特殊字符和用于 shell 的特殊字符相重，因此我们必须在编辑原稿上用引号把这些字符括起来，这样它们将送给 sed，而不是按 shell 起作用。

如果用 UNIX 系统正文编辑程序，那么我们可用编辑程序的替换命令

```
s/.*/fgrep & acctfile >> nuacctfile/
```

把包含一个帐号（如 48150）的行，改变为包含 shell 命令“fgrep 48150 acctfile >>nuacctfile”行。相同的语法可用于流编辑程序。唯一的差别是替换命令必须加引号。sed 中的标志位“-e”指明下一自变量是编辑原稿。因此把文件‘currents’帐号表变换到一串 shell 命令，可用命令

```
sed -e "s/.*/fgrep & acctfile>>nuacctfile/"currents
```

在引号之间的复杂内容是编辑程序的替换命令。用这个命令的唯一问题是其输出在终端上显示。我们可用下面命令将输出保留在文件内。

```
sed -e "s/.*/fgrep & acctfile>>nuacctfile/"\
currents>shellcmds
```

（我们在第一行结束处使用反斜线，是为了将命令延续至第二行）。如果我们执行上面这个流编辑程序命令，你可以发现文件‘shellcmds’内有下面内容：

```
fgrep 48150 acctfile >> nuacctfile
fgrep 48677 acctfile >> nuacctfile
fgrep 56789 acctfile >> nuacctfile
fgrep 56790 acctfile >> nuacctfile
```

当然，实际的正文要长得多，但是在此没有必要全部列出。文件

‘shellcmds’ 中的每一命令将提取一个专门帐户的全部记录，并将它们加在文件 ‘nuacctfile’ 的末尾。因为文件 ‘shellcmds’ 十分长，它最好在晚间或周末执行。当我在周末运行它时，约花费一小时CPU时间。

总结一下，这个方法的价值是有效且容易地写了一个 shell 程序，节省了秘书大约一周的劳动。请注意。这里并不需要 shell 的任何高级特性。我们在这儿用到的 UNIX 系统特性仅仅是输入/输出重新定向，而这在其他操作系统上是没的。在这个例子中，让 UNIX 系统 shell 提供一个解决办法是很容易的，而在其它系统中，同样的结果，很多是要在操作系统上才能够达到的。

14.4 列出子目录

UNIX 操作系统的 ls 命令可列出一个目录中的全部文件名。但 UNIX 系统没有现成的命令去列出一个目录中的全部目录文件。每当你进入文件系统的生疏部分，你就乐意知道当前目录的子目录名字。

第8.8节谈到的命令

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

将列出全部根目录中的目录文件名，这是因为在一个长格式列表中，所有的目录行将以字符 d 开头。我们可使用 shell 自变量扩展这个简单命令，以便列出任一目录的子目录，而不只是根目录。我们用一个位置参数代替根目录的名字，这样会得到更通用的解决方法：

```
ls -l $1 | grep '^d'
```

如果这个命令放在文件 ‘lsdir’ 内，并且使 ‘lsdir’ 成为可执行，则命令

```
lsdir /bin
```

将产生 ‘/bin’ 目录的子目录的长格式列表。

如果没有提供自变量 (/bin), 则位置参数 \$1 未被置值, 此时 ls 程序不接受任何目录名作为自变量, 就只列出当前目录中的子目录文件。这是一个很好的安排。命令

```
lsdir
```

列出当前目录中的子目录, 正像命令

```
ls
```

列出当前目录中的文件一样。

lsdir 只接收一个自变量。若送入命令

```
lsdir /etc /lib
```

将只有 '/etc' 的子目录列出。我们可用一个 for 循环来弥补这个缺陷:

```
for i
do
ls -l $i | grep '^d'
done
```

如果上面这个改进版本放在文件 'lsdir' 里, 则命令

```
lsdir /etc /lib
```

将列出 '/etc' 及 'lib' 两个目录的子目录。

当没有自变量提供时, 作了上面改进的 lsdir 版本将会怎样呢? for 循环在没有自变量时将不会执行, 因而也将没有输出产生。为了纠正这一不足, 我们需放一个 test 在 'lsdir' 文件的开头, 以便确认至少有一个自变量:

```
if test $# = 0
then lsdir .
else
for i
do
ls -l $i | grep '^d'
done
fi
```


lsdir的最后版本用来检查是否至少有一个自变量。如果至少有一个自变量，则 for 循环执行；如果没有自变量，则执行命令“lsdir。”列出当前目录的子目录。要注意，lsdir调用它自己，这是一种熟悉的递归。shell程序允许执行其它 shell 程序（嵌套）或递归地执行它自己。

其它哪些技术能用来实现同样功能呢？UNIX 系统的 test 程序能用来测试文件是否是目录。命令

```
for i in /etc/*
do
if test -d $i
then echo $i
fi
done
```

将测试 ‘/etc’ 目录中的全部文件是否是目录，并用 echo 命令把通过测试的全部文件名字印出。为识别目录我们可用这种技术重写一个lsdir版本。

```
if test $# = 0
then lsdir
else
for i
do
for j in $i/*
do
if test -d $j
then echo $j
fi
done
done
fi
```

14.5 列出当前子树中的文件

不经常使用UNIX系统的人可能放错一个文件，或者记不住文件的确切名字，或者把它放到了一些不熟悉的目录里。在上面任一情况下，文件丢失了，我们需找到它。另外，有时常常要知道已在文件系统中但又不知道确切地方的那些文件（典型情况是这种文件在你的主目录的一些不熟悉的子目录中）。最容易找文件的方法是列出当前文件系统子树的全部文件名（当前子树包括了当前目录，及它的全部子目录，及它们的子-子目录，等等）。

find 命令经常用于查找文件。可用自变量去寻找带有某个名字的文件，或寻找某一日期以后修改过的文件，或寻找带有一定特权的文件等等。如果你未用专门的任选项，则find将寻找子树中的全部文件。

find命令的最简单形式是

```
find . -print
```

它将印出当前子树里的全部文件。点“.”表示查找应从当前目录开始（记住，点“.”经常表示当前目录的名字），任选项“-print”表示文件名应被印出（参见UNIX系统手册中find命令及其任选项的完整解释）。你可记住find命令的语法，或把这个命令放到名为‘lstrree’的shell命令文件中。每当你要当前子树中的全部文件清单时，可送入命令

```
lstrree
```

以代替命令

```
find . -print
```

许多shell程序也象这样简单：它们正是一组放在某一文件里有用但又复杂的命令。

上面问题的一个变种是要列出当前子树中的全部目录。让我们用find命令作为第一种解法。这个find命令包含了一个目录的测试。命令

```
find . -type d -print
```

将寻找并印出当前子树中的全部目录。自变量“-type”指出我们试图寻找一定类型的文件——在我们例子中用自变量“d”指出是目录文件。可把这个命令放到一个文件里，每当我们要列出当前子树中的目录时就打入这个文件名。

下面表示这个问题的另一个解法。我们可用 shell 的 for 循环去实现对某一目录中每个文件的测试。命令

```
for i in *
do
    if test -d $i
    then echo $i
    fi
done
```

将测试当前目录里的全部文件，并印出已证实是目录的全部文件名。为了列出当前子树中的全部目录，我们需要用递归。递归是子程序调用它自己的一种技术。我们可修改上面所示的命令，让它下降到每一个目录，于是可列出该目录的全部子目录。再进一步下降，直到结束。我们把下列命令放到文件‘lstree1’里：

```
if test $# = 0
then dirname = .
else dirname = $1
fi
for i in *
do
    if test -d $i
    then echo $dirname/$i
    (cd $i ; lstree1 $dirname/$i)
    fi
done
```

‘lstreel’ 的自变量是用于保留路径名的踪迹，这些路径名将引向被查找的目录。当 ‘lstreel’ 不包含自变量时，则变量 \$dirname 将获得值 “.”，否则 \$dirname 将获得 ‘lstreel’ 的第一个自变量值。在这个程序内，有趣的程序行是“(cd \$i; lstreel \$dirname/ \$i)”。当用括号将一组 shell 命令括起来时，则这组命令将在一个子 shell 里执行。我们让 cd 命令及 lstreel 命令在子 shell 里执行，是因为在一个 for 循环中间实现一个 cd 会是灾难性的。当子 shell 完成了 cd 命令及 lstreel 命令时，它将转向出口，并将控制送回给在原有目录中的原来 shell。

让我们再来看看另一个方法。不用 for 循环而用由 read 命令组成的 ls 命令来产生当前目录的全部文件。这次我们把程序放在文件 ‘lstree2’ 里。

```
if test $# = 0
    then dirname = .
else dirname = $1
fi
ls | (
    while read i
    do
        if test -d $i
        then echo $dirname/$i
            (cd $i ; lstree2 $dirname/$i)
        fi
    done )
```

除了用 ls 命令代替 for 循环以外，这个程序与前面那个是相似的。在这里，我们用 ls 命令产生当前目录的文件清单。这个文件清单又经过管道操作送到括号内的命令表里。read 命令从管道读各行，并将它们放入变量 \$i 内。程序的结局和 ‘lstreel’ 相同。

对这三种方法的评价留给读者去作。

第十五章 C语言和UNIX系统

C程序设计语言和UNIX操作系统是十分紧密联系在一起。UNIX系统核心的90%以上及UNIX系统中的绝大部分实用程序都是用C写的。C是一个中级程序设计语言。它比汇编语言更易使用也更高产，但它不具备高级语言（如PL/1）的许多代价昂贵的功能。

C是D·M·里奇在七十年代早期为UNIX系统早期版本而开发的。C直接从K·汤普逊设计的语言B演变过来。B和C的主要不同之处在于B的对象限于本身的机器字，而C包含了几种基本类型：字符（或字节）；短、长及机器字长的整数；浮点数。C的类型使它能适应不同的机器。例如，PDP-11上短字长及机器字长的整数是16位，长字长整数是32位。而Interdata8/32的短字长整数是16位，机器字长及长字长的整数是32位。

汤普逊的B是从BCPL演变过来的，BCPL是由马丁·理查德设计的。使C变得很出名的很多特性是从BCPL来的，这些特性包括指针和数组的结合，表达式的简洁风格。从BCPL到B，再到C的演变，已使一个系统程序设计语言转化为一个通用语言。因为C已十分成功地用作系统程序设计，因此有时被称作系统程序设计语言，但事实上，C是一个极其通用的语言。

C作为通用语言在某种程度上是因为它没有导致语言成为专用的那些特性。C缺少内部的输入/输出子系统，缺少对记录及数组这类高级类型的内部操作，也缺少动态存储管理机构。当需要这些机构时，它们是作为外部子程序来提供的。

在单用户计算机系统里，允许或鼓励应用程序直接取得许多计算机资源。但对于象UNIX那样的多用户计算机系统，这是不

可能的。在UNIX系统内,核心负责管理程序与输入/输出设备及文件之间的全部传送。系统调用是程序为了得到操作系统的服务而采用的机制。UNIX系统所包含的系统调用有:实现输入/输出;利用系统提供的环境进行工作;建立新的进程等。

本章将用一般方法讨论C程序设计语言和UNIX操作系统之间的接口。对于非程序员,他会因了解程序是如何和系统接口而感到兴趣;对于程序员,他会因所提供的一般介绍而更熟悉UNIX系统。本章第一节讨论子程序,后面几节讨论一些更有兴趣的系统调用,最后一节讨论C编译程序和用于测试C程序的lint。

C源程序经编译得目标文件。目标文件既包含了相应于C程序的机器指令,也包含了其它一些信息,这些信息将使这个目标文件能和其它几个编译后的文件连接在一起,从而生成一个能执行的完整程序。以后几章将讨论目标文件、分别编译、管理目标文件的几个UNIX实用程序。

使用系统调用的全部细节在UNIX系统手册的第二部分已提供。标准子程序的使用细则,见手册的第三部分。这一章并不介绍用C进行程序设计,阅读本章只需很少C语言的知识。

15.1 标准子程序

在其它语言里具有的许多特性在C语言里被省略了。例如,C内部并不包含处理字符串的任何能力。公共的字符串处理功能在标准子程序库内提供。习惯于高级语言(PL/1或PASCAL)的人们会感到很惊奇:为了比较两个字符串,在C中必须调用子程序。在PASCAL里,比较变量“Str1”和字符串常量“Hello”可使用下面语句:

```
IF Str1 = 'Hello' THEN....
```

在C中,与其等价的是

```
if(strcmp(Str1,"Hello"))....
```

标准子程序“strcmp”用于比较两个字符串。

标准子程序可提供字符串处理功能，通用算术函数，从数的一种表示转换到另一种表示，以及象快速排序一类的通用算法。子程序的目标码存放在目录‘/lib’内的标准子程序库‘libc.a’里。‘/lib/libc.a’通常由C编译程序查找，因此程序员用标准子程序时不需作专门的安排。

最值得注意的C子程序之一是printf。printf用来产生格式输出。在一个程序里，可用调用子程序而输出一个消息。

```
printf("The world is not safe for democracy.\n");
```

上面情形中，printf的自变量是常数字符串。（在字符串末尾的“\n”是换行符在C中的缩写）。一般地，printf用于输出变量的值。printf的第一个自变量是一个格式符（或格式字符串），后面的自变量（如果有的话）是要输出的对象。在格式字符串中的正常字符正好被复制到输出上，而格式字符串中的转换标志将使得后面的一个自变量变换成可印刷的形式再输出。转换标志包括一个百分比符号及后面的一个或几个区分转换种类的字符。使用printf比叙述它更为容易，因此我们将举一些例子。若x，y，z是整数，则能用十进制形式将它们的价值输出

```
printf("Values of x,y,and z: %d %d %d\n",x,y,z);
```

转换标志“%d”表示是一个整数的十进制输出转换。若“Strptr”是一个字符串的指针，则下面将印出字符串的地址（用八进制），后面是字符串的内容：

```
printf("%o %s", Strptr, Strptr);
```

“Strptr”自变量出现两次，因为两次需要它，一次用“%o”八进制转换，另一次用“%s”字符串转换。printf还可用于输出双字长的数，浮点数及字符。在许多UNIX系统里，还有一些printf子程序的变体，它们可把结果输出到一个文件或一个字符串中。

当程序员把一个标准子程序编入某程序中时，执行这个子程

序的实际计算机指令存放在可执行的程序模块里。因此，可执行程序的大小依赖于程序中包含了多少子程序，有一些子程序极大，要占许多存储空间。但是，由于子程序是在可执行程序体内，因此，调用标准子程序只消耗很少时间。

15.2 输入/输出系统调用

在UNIX系统里，设备独立性是最重要的特性之一，这意味着一个程序存取一个磁盘文件与使用终端、纸带穿孔机或打印机一样容易。磁盘文件和专用输入/输出设备皆可用一组系统调用来存取。象普通磁盘文件一样，输入/输出设备也有名字。例如‘/dev/pt’是纸带设备的名字，‘/dev/lp’是行式打印机的名字。

除考虑设备独立性之外，UNIX对输入/输出的其它处理也采用了简明的方法。例如，任何试图随机存取一个象终端那样的顺序设备，都会引起一个错误。但出一次错在UNIX中是一种正常的反应，相反，在其它许多系统中，当一个程序用非正规方法去处理输入/输出设备时，会引起严重后果。

从程序实现输入/输出的观点看，所有输入/输出请求是同时处理的。请求输入的程序从发出请求开始即被挂起，直到输入完成为止。写操作更复杂。当一个程序要传送一些字符到终端时，输出程序从发出请求后即挂起，直到字符被系统接受，并往下执行为止，在UNIX系统中所实现的同步，并不意味着在输出程序恢复执行（撤消挂起）的时刻，字符已到达了它们的目的地（终端上），它只表示字符已为UNIX系统所接受，并至少开始了下面的执行过程。采用同步是因为输入/输出请求肯定要费一些时间，长的请求比短的请求花费更长的时间。

程序的每一个输入/输出接口可用一个象文件描述字那样的数来标识。许多UNIX系统允许每一个程序具有10至20个同时打开的文件。UNIX系统的文件可以是一个磁盘文件，或是终端，或是某些其它输入/输出设备，或者可以是与另一个程序的接口。

UNIX 系统的一个独特的难懂的特性是子程序继承它们的父程序所打开的文件。运行在典型的 UNIX 系统上的许多程序是从 shell 开始的。为了供子程序使用，shell 一般自动地打开三个文件。由于 shell 是所有交互执行命令的父程序，因此在 UNIX 系统里，由 shell 建立标准接口是十分重要的。

第一个可用的接口用文件描述字“0”来标识，它一般称为标准输入。虽然标准输入也容易接到一个磁盘文件或一个纸带读入机上，但它一般接至终端的键盘上。标准输入是交互输入到程序的通常来源。

第二个可用的接口用文件描述字“1”来标识，它一般称为标准输出。虽然标准输出能重新定向至任何可写的文件，但它一般接至终端的显示器上。标准输出一般被程序用作主要的正文输出。例如，ls 命令把它的文件清单写到标准输出上。

用文件描述字“2”来标识的第三个接口称为标准出错输出。它一般接至终端，它是接收许多错误消息的通道。有时，当一个程序的主要输出不在终端上时，出错消息就接到终端上。有时，当运行的程序来自很长的 shell 命令文件时，出错消息可收集在磁盘文件里。

如果无选择余地，则所有三个标准通道可接到终端的特别文件上。如果 shell 命令行包含一个“>”，则标准输出被重新定向到其它文件去；若命令行包含一个“<”，则标准输入被重新定向为由其它文件来。在一个 shell 命令行里，标准出错连接可用“2>”来表示其重新定向。

一个程序可使用这三个标准接口，也可不用。UNIX 系统使用强有力的重新定向机构仅仅是因为方便，一个程序可以另行通过打开与终端相关的文件（‘/dev/tty’）实现输入/输出，不用考虑输入/输出的重新定向，而把输出写到终端上。

实现输入/输出的系统调用包括：

open 和已经存在的文件建立接口。

`creat` 创建一个文件，並和这个文件建立接口。
`pipe` 在两个进程间建立一个可为管道线使用的接口。
`dup` 为一个至少已有一个接口的文件再建立另一个接口。
`fcntl` 在文件上实现几个控制操作。
`read` 从一个文件读数据。
`write` 写数据到一个文件。
`close` 和一个文件断开接口。

当程序执行一个 `open`, `creat`, `pipe` 或 `dup` 系统调用并执行包含 `fcntl` 系统调用的某些操作时，将送回一个文件描述字。被送回的文件描述字在使用 `read` 和 `write` 时，可用来实现对文件的输入与/或输出操作。`read` 和 `write` 系统调用只在文件和进程的数据区之间传送数据。UNIX 系统也包括了实现高级输入输出 (`printf` 和 `scanf`) 和实现经缓冲区输入输出 (`putchar` 和 `getchar`) 的子程序。当一个程序终止时，它所打开的文件自动地关闭。

通过使用 `pipe` 系统调用和有关的建立进程的系统调用，可在进程间建立管道。UNIX 系统的这个十分值得注意的特点在下面第 15.4 节中讨论。

15.3 有关状态的系统调用

为了很好地调整系统的性能，许多操作系统有不少需控制和管理的項目。例如，执行程序的优先权，程序的特权，文件的保护属性，程序执行的环境，文件系统（数据集合）的安装和拆卸。

在许多操作系统里，状态项仅仅从系统的控制面板，经过一个硬件编码的命令解释程序予以控制。由于 UNIX 系统不用硬件编码的命令解释程序，因此，仅有的途径就是使用一个初始的程序来控制 and 监视状态项。例如，任何执行程序可要求增加或减少

它的执行优先权。任何程序能减少它的执行优先权，但只有特权用户运行的程序能实际上增加它的优先权。在UNIX系统里，任何程序都能请求对系统状态功能的控制权，但只有核心允许的请求才能满足。

另一个例子是控制一个文件的存取方式。在UNIX系统里，为控制所有者、同组用户及其他人对文件的读写和执行的方式(参见第六章)，程序可执行 `chmod` 系统调用。下面的一个简短的C程序将使用 `chmod` 系统调用，使文件 ‘nopeeking’ 成为不可取(其他任何人对 ‘nopeeking’ 都无特权)，直到方式得到重新改变为止。

```
#define NOACCESS 0
main()
{
  chmod("nopeeking",NOACCESS);
}
```

这个程序将改变方式，使 ‘nopeeking’ 只能为该文件的所有者和特权用户运行。

控制和监督指定状态项的系统调用包括：

<code>time</code>	读时间。
<code>stime</code>	置时间。
<code>getpid</code>	决定进程的标识号。
<code>getuid</code>	给定用户的标识号。
<code>getgid</code>	给定小组的标识号。
<code>chown</code>	改变文件的所有权。
<code>chmod</code>	改变文件的方式。
<code>chdir</code>	改变当前的工作目录。
<code>link</code>	建立与文件的联结。
<code>unlink</code>	删除与文件的联结。

mount	安装文件系统。
unmount	拆卸文件系统。
nice	改变进程的优先权。
stat	获得文件的状态。

其它几个和状态有关的系统调用，只存在于不同的UNIX系统版本内。

15.4 控制进程的系统调用

UNIX系统的实力之一，是它有一组用于建立和协调进程的系统调用。在一个时刻只允许一个进程的系统里，这些系统调用（除exit外）可能都是多余的，但控制进程的系统调用是基本的，又是有效的，它们包括：

fork	把一个进程分为两个。
exec	改变进程的身份。
exit	终止一个进程。
kill	送一个信号给进程。
signal	当一个信号收到时，指定一个应执行的动作。
wait	等待一个子进程结束。

在许多系统里，可以将程序连在一起，让它们一个接着一个地执行。在UNIX系统里实现类似操作的进程称为exec。为了用新进程替代当前进程，可用系统调用exec的几个变体，调用exec的进程称为前驱进程，被创建的进程称为后继进程。见图15.2。

如果仅有建立进程的系统调用exec，那么UNIX系统将是一个十分虚弱的系统了。这是因为系统调用exec不增加进程的数量，只改变进程的形式。在UNIX系统里，要增加进程数目，必须用系统调用fork。系统调用fork建立新的进程，它是被调用进程的精确复制品（除了像进程的标识号这种和进程特性有关的一些参数以外）。调用fork的进程叫父进程，被建立的进程叫子进

程。系统调用fork不破坏父进程。在系统调用fork以后，两个进程将争夺系统资源（如执行时间）。见图15.1。

fork最普通的用法是建立一个立即执行系统调用 exec 的子进程。这时的fork的作用是为父进程创建一个具有新的身份的子进程。这比只产生双份进程更值得注意。见图15.3。

通常，一个子进程是为完成某些特定任务而建立的，父进程在继续工作前要等待这个任务完成。一个明显的例子是，使用fork-exec对的 shell 几乎执行了所有的打入命令（少量命令是专门由 shell 处理的）、除非你用“&”符号指明后台执行，否则shell（父进程）将执行系统调用 wait 去等待子进程的完成（死亡）（当你用“&”去进行后台执行时，shell 将不去等待子进程完成）。

系统调用 kill 用于从一个进程送信号到另一个进程（名称“kill”选择得不太恰当）。此信号是一个相互约定的值，它在进程间交换某些信息。系统调用 kill 的一个通常用法是把信号SIGKILL

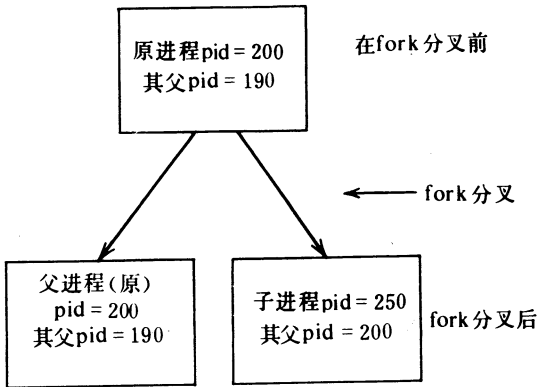


图15.1 系统调用fork

系统调用 fork 用于将一个进程变成两个。除了描述进程的一些关键参数（pid 是进程标识号）外，分叉后的两个进程是一样的。

送给进程。进程收到 SIGKILL 信号将停止它自己。在 UNIX 系统中，还有十几个其它标准信号可供使用。

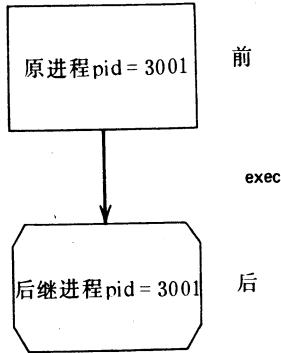


图 15.2 系统调用 exec

系统调用 exec 用于改变一个进程的身份。

由于在给定时刻只有一个进程在实际运行，因此系统调用 kill 的目的进程肯定正被挂起（偶然情况下它不存在）。系统调用 kill 送出的信号到目的进程开始活动以前，不起作用。每当在进程挂起后又重新开始执行时，系统将自动地检查一下信号表以便看看是否有信号已到达。若有信号，将进行相应的动作。例如，在信号 SIGKILL 到达后，要做的相应动作是执行系统调用 exit。名为 signal 的系统调用是用来指定当一个给定信号到达后所应该做的动作。连在一起的一对系统调用 kill-signal 是一个简单而又是功能很强的内部进程通信系统。

进程只在它们唤醒时才响应信号，这个事实的一个不良结果是一直等待着某些事情的进程会一直不死亡。在 UNIX 系统里，这类进程通常是有毛病的设备驱动器系统。虽然它们有些并不带来任何损害，但不能结束使正在等待的进程大量累积，将因核心进程表的最终装满而使这些进程轮流死亡。

UNIX 系统还有系统调用 exit，它将使正调用的进程死亡。

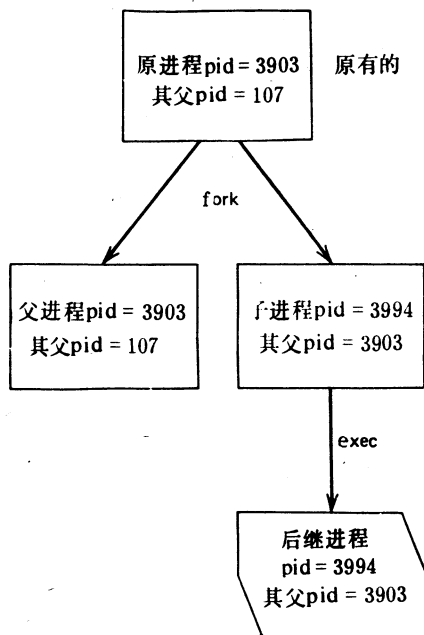


图15.3 建立一个不同身份的子进程

在 UNIX 系统里,为了建立一个有不同身份的新进程,常常在 fork 后安排一个 exec。这个技术被 shell 用于执行你的命令。

当响应各种信号以及在一个 C 程序的主模块结束时, exit 将自动被调用。exit 能直接用在程序的任何地方,同时它将自动地关闭进程所打开的全部文件。

下面用管道的讨论简单小结一下有关进程控制的系统调用。管道是有关进程间的连接。产生一个管道的第一步是为某一进程执行系统调用 pipe。返回两个文件描述符:一个读,一个写。接着为产生两个进程而执行 fork,通常子进程为改变其身份而执行 exec。由系统调用 pipe 所建立的已打开文件,在整个 fork 和

exec 过程中保留打开的状态。在 fork 及 exec 以后，一个进程使用读描述字，另一个使用写描述字。当 shell 在两个进程间建立管道连接时，shell 将安排两个管道描述字值“0”和“1”，一般它们相应于标准输入和标准输出。

15.5 将自变量传递给程序

自变量是为了控制一个命令的操作而在一个命令行内指定的信息。例如，当送入命令

```
cc myprog.c
```

时，字“myprog.c”是一个自变量，它将引导 C 编译程序去编译文件“myprog.c”中的代码。本节讨论在 UNIX 系统中把命令行中的自变量传递给程序的机制。

UNIX 系统最有价值的特性之一是：程序的自变量能直接地放在命令行内。将命令行的自变量传递给一个程序，并非 UNIX 系统所独有，然而，UNIX 系统 shell 所处理的自变量总数和这些自变量特别容易使用，则是 UNIX 所独有的特点。下面几段讨论自变量实际上是怎样传送的。本节的最后几段提供了一个从命令行取得自变量的 C 程序的例子。

这里的讨论将忽略由 shell 或其它程序给命令传送自变量表所做的全部工作。第 13.14 节我们已讨论了文件名的产生、参数替换及命令替换。这些工作的结果，生成传递给命令的自变量表。

让我们从下面情况开始来进行讨论。某些程序（典型的是 shell）有一组字符串（自变量）和一个用系统调用 exec 执行的程序名。和通常情况一样，这个程序为建立一个新进程进行了一次 fork，然后子进程准备执行 exec。

系统调用 exec 的几个变体允许将一个字符串数组传送给一个程序。当带有自变量数组的系统调用 exec 执行时，操作系统的核心从调用程序处取得了这个自变量数组。然后，后继程序被

装入到存储器（调用程序在 `exec` 成功后被覆盖），这个自变量数组被放在后继程序的存储空间中。见图 15.4。

程序可使用两个参数，一个是自变量的个数，一个是指针，这个指针指向一个由自变量指针（字符串的起始地址）所组成的数组。像 C 程序的所有参数一样，这两个参数放在栈中，而实际自变量放在高速存储器内。小心别混淆参数和自变量，参数的值可为程序所用，而自变量是被传送给程序的一个字符串。

一旦上述事情完成，程序就开始运行。事实上只用两个参数就可访问任意一个自变量，这似乎比较奇特。使用两个参数的优点是：程序写法统一；程序不必预先知道它要接收多少自变量。程序的主模块必须说明这两个参数。第一个自变量是一个整数，第二个自变量是一个指向字符串数组的指针。下面是一个短程序，它一次一行地印出它的自变量。

```
main(argc,argv)      /* echo arguments one per line */
int  argc;
char *argv[];
{
int  i;
for(i = 0; i < argc; i++)
    printf("%s \n", argv[i]);
}
```

C 语言的程序员会认识这是一个印出字符串数组的程序。如果你不熟悉 C，也不必担心这个程序严格的语法。这个程序存放

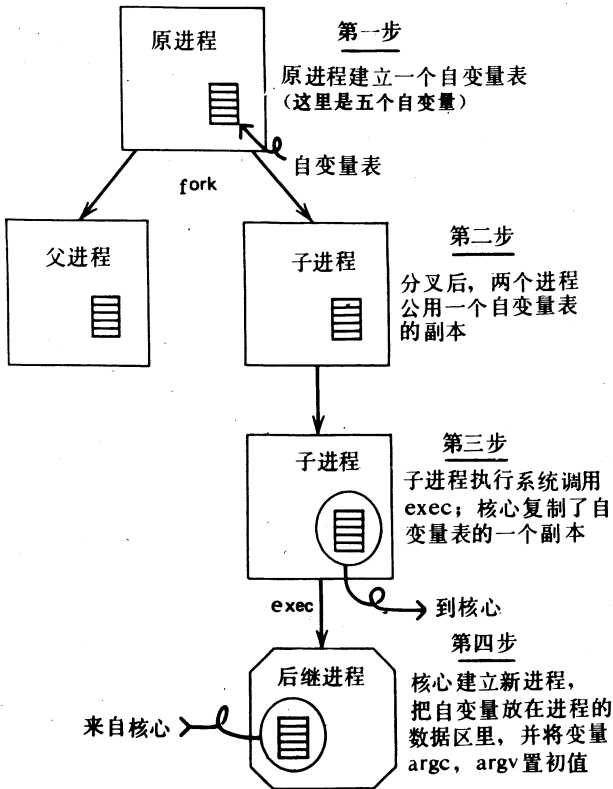


图 15.4 传送自变量给一个程序

在文件 'showargs.c' 里。命令

```
cc -o showargs showargs.c
```

将编译 'showargs.c', 并把可执行的输出放在文件 'showargs' 内, 如果你运行 showargs, 你将发现程序名经常在“真”的自变量前印出。这是因为程序名被当作 0 号自变量, 因此要注意 argc 至少是 1。如果你只要印出自变量, 而没有程序名, 则你必须改变 for 循环从 1 开始, 而不是从 0 开始。

两个自变量经常命名为`argc`（自变量个数）及`argv`（自变量向量），虽然严格地说这些名字是可任选的。参数“`argv`”常常等价地说明为

```
char ** argv;
```

（熟悉C的人或许能解释为什么它们是等价的）。说明的次序是重要的，自变量的个数必须在前。

15.6 系统调用的实现

系统调用的接口设计成和子程序接口相同。参数可传送给系统调用，而值可被送回。两个接口的基本差别是：在一个子程序执行期间，代码连接到可执行的程序模块内；而在一个系统调用执行期间，大多数代码常驻在核心地址空间（而不是程序的地址空间）里。因此增加一个系统调用到程序里，对程序大小影响极少；而当把一个新子程序加到程序中时，可能会增加几千字节的程序量。

调用一个子程序只花费很少时间。可是，执行一个系统调用是一个消耗时间的过程。在UNIX系统的PDP11版本上，系统调用工作时，用了一条`trap`指令。`trap`指令的作用与硬件中断有些相象。`trap`引起程序前后关系的很大的改变，这是很消耗时间的，并经常导致调用进程被挂起甚至被对换出去。

归根到底，系统调用是花费时间，应该少用。最浪费时间的可能要算用单字节传送方法来实现大量字节的读或写。例如，下面是使用单字符读操作来统计文件中的空格数目的简单程序：

```
/* count blanks in a file reading char by char */
main(argc,argv)
int argc; char *argv[];
{
int fd;
int count = 0;
char c;
fd = open(argv[1],0);
```

```

if (fd < 0)
    { printf("Cannot open %s. \n",argv[1]); exit(1); }
while(read(fd,&c,1) == 1)
    if (c == ' ')
        count++;
printf("There are %d blanks in %s. \n",count,argv[1]);
}

```

上面这个程序执行系统调用 `read` 去读每一个字符。当在一个轻负荷的PDP11/70上运行这个程序来统计大约20000字符的文件中空格的数目时，平均运行时间为33秒。约95%的时间花在20000次系统调用上。如果这个程序只需执行一次，那么还可认为是能接受的。然而，UNIX系统包括了经缓冲区的输入/输出，对于上面这个应用来说，用标准的经缓冲区的输入/输出子程序 `getchar` 来代替 `read`，它能工作得很好。其程序如下：

```

#include <stdio.h>
main(argc,argv) /* count blanks using buffered I/O */
int argc; char *argv[];
{
FILE *fd;
int count = 0;
char c;
fd = fopen(argv[1],"r");
if (fd == NULL)
    { printf("Cannot open %s. \n",argv[1]); exit(1); }
while((c = getc(fd)) != EOF)
    if (c == ' ')
        count++;
printf("There are %d blanks in %s. \n",count,argv[1]);
}

```

这个程序更复杂，因为他使用标准的经缓冲区的输入程序包（参见UNIX系统手册第三部分中的fopen和getc的描述）。可是，在代码上轻微的增加，远远补偿了在速度上巨大的浪费。这个程序运行约需一秒，花费的时间只有执行系统调用时间的三十分之一。这个差别是由于第二个程序仅执行了约40个系统调用来扫描这个20000字符的文件。

UNIX系统在系统级建立了缓冲文件，因此对于方便地读文件并没有什么困难。但是使用象read这种系统调用，而不是用像getc这种经缓冲区的输入子程序，一次一个字节地读大量输入，是没有什么好处的。

15.7 分别编译

C语言的整个源程序的代码可分散在多个文件里。将源程序放在分散的文件里是为了便于模块设计、开发和调试。象本章前面所示的echo或统计空格的小程序（最多几百行代码）大多存放在单独一个文件里。大程序（几百行甚至更多）一般存放在几个文件里。

包括几个子程序（典型的是几百行代码）的文件，将比包含几千行代码的大文件更容易了解和工作。一个文件的程序能分成几个文件，比如将子程序或数据结构收集成几个组，将它们分别放在独立的文件里。但子程序或一个数据说明这种逻辑实体是不能分割在两个文件中的。

把一个程序分成几个文件将引起几个新问题。一个问题是某个文件里的子程序需要知道另一文件里有关的子程序和数据说明。C语言有一种特别的说明叫外部定义。外部定义的目的是为了引用及描述那些不能在本文件里定义的项。如果引用其它文件中的项目个数很少，则少量外部定义可放在本文件里。可是，若这个数目太大，最好建立一个包括外部定义的文件，并使用C编译程序的include特性去蕴含这个文件。通过使用include伪指

令，把文件包括进去。在一个C语言的程序里，下面行

```
#include "defs.h"
```

将蕴含文件‘defs.h’的内容。

另一个问题是：整个程序需要知道的一些常量和单词。例如在一个表格处理的程序里，表格中项目个数的最大值是一个关键常量，需被所有子程序知道。在C里，define伪指令能用于建立一个指定的常量。下面行

```
#define LISTLEN 20
```

在C程序里将定义一个名叫“LISTLEN”的常量。在一个程序里，每当遇到名字“LISTLEN”时，就用常量“20”替代它。一个include文件通常包括了整个大文件都使用的全部定义。define伪指令实际上是一个宏置换机制，它可有参数，也能用于生成复杂的C代码序列。

在一个C程序里，“#”开头的各行均被用作象include或define一样的伪指令。所有这些伪指令均由一个称作C预处理程序的模块进行处理，这个预处理程序是C编译程序的一个部分。C预处理程序也实现其它工作，象从一个程序里删去注解等等。有时，你若想了解C预处理程序是否按你的要求去做，你可看一下它的输出。你也可用这个技术去核实一个注解，看它是否已经不知不觉地超出了它预定的限制。shell命令

```
cc -P myprog.c
```

将对文件‘myprog.c’进行预处理，并将输出放在文件‘myprog.i’里。

C编译程序的正常目标是编译一个完整的程序，以便产生一个可执行的模块。如果你已将一个大程序分成几个文件，那么你可以通过在命令行中提及全部文件以实现一个完整的编译：

```
cc fileA.c fileB.c fileC.c
```

如果都工作正常，没有错误查出，则将产生四个文件：名为‘a.out’的文件包括了可执行的程序，‘fileA.o’，‘fileB.o’及

‘fileC.o’三个文件包括了相应源文件的目标代码。

为了减少以后编译的工作量，几个独立的源文件的目标代码被保留下来。如果‘fileA.c’修改了，而‘fileB.c’和‘fileC.c’没有改变，则已过时的目标文件只有一个‘fileA.o’。目标模块‘fileB.o’和‘fileC.o’能够使用，因为它们所依赖的源代码并没有改动。通过只对‘fileA.c’的部分编译来实现了整个系统的重新编译，然后将这三个目标文件重新连接而产生一个新的可执行文件‘a.out’。上述工作可通过打入下列命令来完成：

```
cc fileA.c fileB.o fileC.o
```

C编译程序的一个很有用的特性是：它知道哪一种后缀用于哪一种文件。根据后缀“.c”，C编译程序知道‘fileA.c’是必须编译的源程序；根据后缀“.o”，它知道‘fileB.o’和‘fileC.o’是只需要（和‘fileA.o’）连接的目标文件，通过连接可产生一个可执行的‘a.out’文件。

要记住哪些文件已过时，哪些未过时是十分困难的。一种方法是经常将最新的目标文件保存下来。每当一个源文件改变时，相应目标文件立即予以重新建立。部分编译这个词是说明一次编译的目标仅仅是一个模块，而不是整个软件系统，C编译程序的任选项“-c”通常用于部分编译。命令

```
cc -c fileA.c
```

将产生一个最新的‘fileA.o’。如果不使用任选项“-c”，而当C编译程序发现文件‘fileA.c’中的源代码不是一个完整程序时，它将送出一个出错消息。

送入下列命令，可从一组最新的目标文件产生‘a.out’：

```
cc fileA.o fileB.o fileC.o
```

保留已过时文件踪迹的另一种方法是使用UNIX系统程序make，为了决定哪些需重新编译，make观察一个软件系统中全部文件的时间。要进一步了解程序make，可参见第十二章。

15.8 lint——检查C语言程序

关于在一个程序内检查错误这一点，C程序设计语言是十分不规则的，许多问题往往由下面情况引起：当一个程序分成几个模块时，编译程序要依靠外部定义去决定其它模块中变量的类型。如果外部定义不准确，将会产生可疑的代码。另外，问题也会由使用指针引起，因为这是一种会使变量的基本类型分辨不清的技术。

总之，C编译程序假定你写的代码已包含了你要实现的操作，如果一个操作在逻辑上是可能的，则大多数情况下，C编译程序将试图产生相应的机器指令。C编译程序的态度是：你比编译程序更聪明，无论什么情况都需要你的干预。

幸亏UNIX系统还有一个检查可疑情况的程序 lint。这个程序用于检查C程序的多种错误，包括：在不同文件里同一变量的类型有不同的定义；变量或变量值一直没有使用；变量在其值给定之前就明显被使用；及其它可疑的情况。lint不产生目标文件——它只在你的编码练习上产生警告及评注。有时 lint产生的警告并不表明有什么麻烦，但一般情况下，警告意味一些事情有错或不能移植。许多公司要求给他们写的所有软件都能通过 lint的分析检查。

用一个程序去实现编译，用另一程序去实现检验，可能会使两个程序失去同步。认识到C语言的这一特性，可以只选单独一种程序工作。

在单独一个程序里而不是在编译程序中实现校验，有几个优点。一个优点是：在一个独立程序里将实现详细的分析，而在编译程序里分析要花费很多时间。另一个优点是：lint能把一个软件系统当作一个整体进行分析，而一个语言理想的特点是分别编译。

第十六章 程序员用的实用程序

UNIX 系统有几个用于处理由编译程序生成的二进制文件的实用程序。这些实用程序主要由程序员使用。UNIX 系统的语言编译程序 (C, FORTRAN, PASCAL 等) 把一个包含源程序的正文文件翻译成一个包含二进制信息的目标文件。由于目标文件是二进制信息而不是正文文件, 因此不能用处理正文文件的 UNIX 标准实用程序来处理。相反用本章叙述的实用程序可以对目标文件实行标准的转换。

一些 UNIX 系统包含了这样一种程序设计系统, 它允许程序员在某一 UNIX 系统上编译, 而在另外其他系统上执行。这种系统被称作交叉系统, 对于它所产生的文件格式还没有一个标准。由交叉编译系统产生的可执行文件的处理本章将不予讨论。这里讨论的实用程序假定不工作在这类文件上。

本章第一节讨论使用 C 编译程序来产生目标文件。用 FORTRAN 或 PASCAL (或其它编辑程序) 产生的目标文件基本相同。最后一节叙述工作在目标文件上的多种实用程序。

16.1 编译

目标文件是高级语言程序编译的结果, 或是汇编语言程序汇编的结果。由标准的 UNIX 语言编译系统所生成的目标文件有四个部分: 头、程序 (指令和数据)、再定位信息及符号表 (见图 16.1)。头和程序总是要提供的。头说明不同部分的大小, 并且指明是否提供再定位信息和符号表。再定位信息和符号表对组合目标文件成为可执行程序 and 调试可执行程序是很有用的。

很短的源程序代码常常放在一个文件里。编译这样的程序,

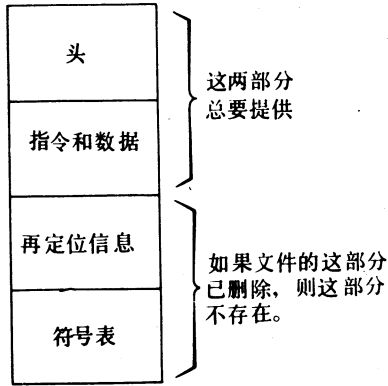


图16.1 一个目标文件的结构

可能立即产生一个可执行的目标文件。在UNIX系统里，可执行的缺省文件名字是‘a.out’。命令

```
cc myprog.c
```

是用cc（C编译程序）编译C语言的程序‘myprog.c’，并把目标文件放在‘a.out’里。如果‘myprog.c’中有其它程序模块尚未决定的引用，或者它所包含的源代码有错误，则‘a.out’就成为不可执行的。若源代码没有错误，并且没有未决定的引用，则‘a.out’是能成为可执行的。如果你使用“-o filename”任选项，那么可执行的输出将放入你选择的文件里。命令

```
cc -o myprog myprog.c
```

是把输出放在‘myprog’而不是‘a.out’里。

对于大型、复杂的程序，其源代码通常放在几个文件里，这样可简化程序的开发和维护。对几个源代码文件之中的一个代码文件进行的编译称部分编译。如果源文件命名为‘file.c’，则从一次部分编译产生的目标文件名是‘file.o’（目标文件常常称作“.o”文件）。

C编译程序的任选项“-c”引导编译程序实现部分编译。部分编译和完全编译的主要差别是：部分编译时，不认为尚未决定

的外部引用是一个错误。命令

```
cc -c fileA.c
```

将产生目标文件‘fileA.o’，部分编译的结果不能成为可执行的文件。

存放在几个源代码文件里的一个程序的完全编译，可用目标程序的任一联合来实现。源文件不再当作一个独立程序模块来表示。因此存放在源文件‘fileA.c’，‘fileB.c’和‘fileC.c’中的一个程序能用下面命令实现完全编译：

```
cc fileA.c fileB.c fileC.c
```

如果全部源代码文件已经分别编译并产生了目标文件，则完全编译也可用下面命令来实现。

```
cc fileA.o fileB.o fileC.o
```

这个命令的执行将比前面一个快得多，因为只有极少工作要做，主要工作是组合目标文件以产生一个可执行文件。在一次完全编译中，也可以混合目标文件和源程序文件：

```
cc fileA.c fileB.o fileC.c
```

在这个命令中，我们假定‘fileB.c’已经被编译产生了‘fileB.o’。

16.2 size ——印出目标文件的特性

size 程序印出目标文件的程序段长度。程序段包括三部分：指令（虽然与正文文件没有关系，但也常称作 text），已初始化的数据和未初始化的数据。**size** 程序用十进制印出程序段的这三个长度，接着用八进制及十进制印出这三部分长度的总和。

由程序**size** 印出的只是一个目标文件程序段的长度，而不是整个目标文件的长度。命令“ls -l”将显示一个目标文件的整个长度。命令

```
size file?.o
```

将印出前面例子中所有目标文件的长度。

16.3 strip——从目标文件中删除符号表

命令 `strip` 用于从一个目标文件中删除再定位信息和符号表。这个命令的结果有效地减少了文件的长度。命令

```
strip myprog
```

将从 ‘myprog’ 中删除再定位信息和符号表。编译时使用 C 编译程序的 “-s” 任选项也能达到同样的结果：

```
cc -s -o myprog myprog.c
```

经过这种删除后的文件更难于调试。删除部分编译的结果是毫无意义的，因为删除后的文件就不能和其它文件组合成一个可执行的程序了。

16.4 nm ——印出目标文件的符号表

程序 `nm`（列出符号表中的名字）考察指定的目标文件的符号表。被 `nm` 考察的目标文件必须是未用 `strip` 命令删除过的文件。不带有任选项的 `nm` 程序将印出符号表的全部内容：

```
nm myprog
```

这种 `nm` 形式已很少使用，除非你要用管道将符号表输出给 `grep` 以选择某些项目，或者你要考察的是一个十分小的目标文件。事实上，许多程序包含的符号比你想的要多很多。

`nm` 在每行印出一个符号。这行的第一个字段是符号的值（如果它已被定义的话），第二个字段是表明符号类型的一个字母，最后的一个字段是符号的名字。在 C 及其他高级程序设计语言中，从编译程序观点来看，一个符号的“值”是它的地址，因而被程序 `nm` 印出的“值”是该符号的地址。局部符号的类型字母是小写，而外部符号的类型字母是大写。

最重要的类型字母有：`D`（已被初始化的数据）；`B`（未初始化的数据）；`T`（正文）；`A`（绝对值）；以及 `U`（未定义的符号）。可从 UNIX 系统手册中找到完整的类型字母表。

对于 nm，最有用的任选项是“-u”，它让 nm 印出一个未定义的符号表：

```
nm -u fileA.o
```

未定义的符号是一个程序中已使用但未说明的那些符号。通常源程序文件是更好的信息来源，但当某些事情变得无法解释时，可以试试检查一下目标文件中的符号。

任选项“-g”将使 nm 印出符号表中的外部符号表：

```
nm -g fileA.o
```

外部符号表会使你知道文件中哪些项已被定义了。

16.5 ar——档案文件

UNIX 库管理程序称作 ar（档案）。一个程序库（也叫做一个档案）是一个含有一组文件的文件。虽然 UNIX 系统的程序库能包含任何类型的文件，但大多数库程序是目标文件。当单独的一个个文件加到库中时，这些文件并不作改动。

经常使用的目标文件组合到库中，使它们更容易引用。例如，C 的标准子程序及系统调用的全部目标文件就放在一个单独的库‘/lib/libc.a’中。其它的库有：图形库，数学函数库等。如果一个程序需要几个图形子程序，那么在图形库中叙述图形子程序比按其名称叙述每一个图形子程序要容易得多。

程序 ar 能用于：印出档案中的文件表；把文件加至档案；从档案中取出某文件的副本，等等。现在让我们先建立一个包含目标文件‘fileA.o’，‘fileB.o’，‘fileC.o’的档案，则可用下列命令：

```
ar rv libfile.a file?.o
```

与其它许多 UNIX 系统程序不一样，ar 程序任选项（这里是“r”和“v”）的前面没有“-”，这是因为 ar 要求至少有一个任选项。

“r”任选项表明被指定的文件将加入库‘libfile.a’中。必要时，将创建文件‘libfile.a’。通常“r”任选项用于把文件加至

一个已存在的库中，但它也能用来建立一个库。“v”任选项表明是“详细”(verbose)方式，即每一个ar完成的操作将都印出。当你交互地使用ar时，采用“详细”方式是一个很好的想法。

下面的命令执行后，将列出库‘libfile.a’中的文件‘fileA.o’，‘fileB.o’，及‘fileC.o’。命令

```
ar t libfile.a
```

将印出这个库的内容表，这三个文件也应列在其中。

在一个档案中，文件的次序常常是重要的。命令

```
ar mva fileA.o libfile.a fileC.o
```

将在库‘libfile.a’中，把文件‘fileC.o’移至（任选项“m”表明“移动”）文件‘fileA.o’的后面（任选项“a”表明“后”）。（与通常情况一样，任选项“v”表明“详细”方式。）命令

```
ar t libfile.a
```

将显示新的次序：‘fileA.o’，‘fileC.o’，‘fileB.o’。相应地，‘fileB.o’可移到档案的开始处，这可通过以下命令：

```
ar mvb fileA.o libfile.a fileB.o
```

将把‘fileB.o’移（任选项“m”）至库程序‘libfile.a’中‘fileA.o’的前面（任选项“b”表明“前”）。命令

```
ar t libfile.a
```

将显示新的次序：‘fileB.o’，‘fileA.o’，‘fileC.o’。ar最常用的任选项是“r”(replace)。如果‘fileB.c’被改动并重新编译，则‘libfile.a’外面的‘fileB.o’副本将比库程序内的‘fileB.o’更新，命令

```
ar rv libfile.a fileB.o
```

将用新的版本代替库内已过时的版本。替换时也能用“a”(后)或“b”(前)任选项改变文件的次序；若不能用这些任选项，则只作替换操作，并不改变次序。

有时要从库中得到一个文件的副本。用输出复制带源代码或正文的档案要比复制带目标文件的档案更加普遍。命令

```
ar xv libfile.a fileB.o
```

将从库中摘录出(“x”任选项表明摘录)‘fileB.o’的一个副本。摘录出副本并不改变库内容,唯一的改变是你的当前目录里的档案成员副本已被建立。

16.6 ld ——组合目标文件

ld是UNIX系统的连接编辑程序。这并不是一个正文文件的编辑程序,而是一个目标文件的编辑程序。ld组合目标文件的目的,通常是为了产生一个可执行的输出文件。偶然情况下,ld组合的文件是为了在进一步连接编辑期间使用。

目标文件的连接是通过组合全部的程序段来形成一个大的程序段。用符号表和再定位部分的信息来调整全部交叉访问。当目标文件组合到一个库里时,单独一个个文件并不会被改变:当目标文件用ld组合时,目标文件的四个相应部分(头,程序,再定位,符号表)合并到一个有同样四个部分的大目标文件中。

许多UNIX系统用户并不直接使用ld程序,而是使用某个编译程序(cc, f77, pc),该编译程序自动地调用ld程序。编译程序时,通常有好几个阶段,在完全编译时的最后阶段是连接编辑。在部分编译时不进行连接编辑。

可通过给编译程序提供任选项来控制ld程序的操作。编译程序将把任选项传到ld。例如,任选项“-s”使ld从输出文件中删除再定位信息和符号表。命令

```
cc -s myprog.c
```

将把任选项“-s”传送到ld程序。ld将删去输出中的有关部分。

有些程序经常被几个用户同时使用。典型例子是shell和编辑程序。当一个程序在几个不同的进程中同时要用到时,程序中的指令并不需要有几个副本,而是每一正在执行的进程必须保持一个程序的数据副本。

称一个程序为纯可执行的,是指几个进程能共享它的指令(正

文)。如果连接编辑程序提供任选项“-n”，则它将建立一个纯可执行的程序。纯可执行程序的特点是：它比非纯程序要求稍多一些存储单元。如果一个纯程序只有一个副本在执行，则可能使用了稍多一些存储单元，而当纯程序的几个副本皆在运行时，由于正文只有一个副本，因而内存空间有一定净节省。命令

```
cc -n -o nprocs nprocs.c
```

将产生一个纯可执行程序‘nprocs’。命令file能用于找出哪个文件是纯可执行的（信息是放在目标文件的头内）。命令

```
file nprocs
```

将印出“pure executable”，而不是“executable”，后者是为非纯程序而印出的。在纯和非纯程序之间，对于程序的用户来说并没有什么不同。两者的执行是相同的，也不可能发现别人在共享程序的正文。没有存储保护和存储映象的计算机系统上，这个特点很可能没有。

另一个经常使用的任选项是“-i”，它把程序正文和数据放在不同的地址空间。在支持指令和数据地址空间分离的计算机上，任选项“-i”用于更大的程序。

第十七章 yacc和lex

本章讨论UNIX系统两个最有魅力的工具：yacc (Yet Another Compiler Compiler) 及lex。yacc及lex在UNIX系统的文献中被广泛引用。在建立和维护UNIX系统某些关键的实用程序时，yacc和lex也是很重要的。不过只有很少UNIX系统的用户了解它们有什么用。本章所描述的yacc和lex将回答这样的问题：

“这两个工具最好用在哪儿？”。如果你要使用yacc或lex，那么应参阅更详细的手册和文章。

命令语言的识别问题是计算机程序设计中的一个最普通的问题。许多计算机都使用语言去控制某些操作，例如在UNIX系统中，有shell程序设计语言，C程序语言，make程序所使用的相互关系的说明等等。

有些命令语言很容易识别，而另一些却要在程序设计方面花很大精力才能识别。虽然完全能用C语言（或FORTRAN，PASCAL）写一个程序来识别一个复杂的命令语言，但是已经开发了更好的技术。本章将介绍其中的一些技术。

17.1 词法分析和语法分析

识别一个复杂的命令语言一般分为两个阶段：词法分析和语法分析。在词法分析阶段能识别低级的对象，如数、操作符及专门字；而在语法分析阶段能识别高级的对象，如程序设计语言中的语句。决定词法分析时识别哪些对象，语法分析时识别哪些对象，有一定的灵活余地，而经验常常能提供最好的帮助。

我们用逐项词法分析程序来描述实现词法分析的程序，用逐项语法分析程序来描述实现语法分析的程序。本节的后一部分有

词法分析和语法分析的更详细解释。本章的后两节讨论了产生词法分析程序和语法分析程序的UNIX系统实用程序lex和yacc。

在词法分析阶段，输入的是一串字符。这些字符可以交互地由用户从终端送入，也可以来自一个文件，也可以是另一程序的输出。不管哪种情形，词法分析程序均按一组规则扫描这一输入。当词法分析程序识别了一个对象时，它就输出所遇到对象的类型表示符。这个表示符一般叫作单词 (token)。若给定输入

$$25 * (16/2) + 15$$

词法分析程序将输出类似于下面这样的符号

NOLNONRON

这里，单词“N”表示数，“O”表示操作符，“L”表示左括号，“R”表示右括号。对于我们读者来说，宁可用上面算术表达式的普通形式，而不用下面这种序列“NOLNONRON”，但对于语法分析程序来说，下面这种形式是很合适的，因为它缩短了算术表达式的表示。

词法分析阶段的目的是减少或去掉输入字符串的许多不规则部分，如某些项的长度和空格，以便产生由所遇到的项编成的一个输出序列。产生同样语法分析序列“NOLNONRON”的另一个输入是：

$$0x19 * (0x10/0x2) + 0xF$$

在这个表达式中，数是16进制的，而不是10进制的，项的空格也是不同的。但基本表达式是相同的，因而词法分析输出也是相同的。

语法分析阶段将负责了解输入的高级特征。如果在一次识别过程中遇到上面所示的序列 (NOLNONRON)，则语法分析程序将负责证实这一序列是否代表一个有效的表达式，并产生一个相应的动作序列。

一个语法分析程序目标的说明就是要列出命令语言所必须遵循的一组规则。若一个命令与这个规则一致，则一些相应的动作

将进行；若命令不适合这一规则，则命令有错。例如，用一算术命令语言进行工作时，我们要求列出的这组规则允许象“7”，“1 + 1”，“5 / 8”，及“6 * 7 * 14”这样的命令，而应排除象“6 7”，“5 + * 8”，及“7K4”这类错误。

假定已有了一个数的定义，我们可以定义下面所示的一个算术表达式

```
expression:
    number
    |
    expression "+" expression
    |
    expression "-" expression
    |
    expression "*" expression
    |
    expression "/" expression
    ;
```

在上面表达式中的竖线表示一个可选择的规则，number 表示任何一数。

选择表的开始用冒号，表的结束用分号。加减乘除这类标准操作符用引号括起来，这是因为在一个表达式中，它们必须照字面形式出现。上面所示的定义规定了一个表达式或者是一个数，或者是两个进行加减乘除的表达式。

按照表达式的定义，我们已经建立了任意长短表达式的定义。计算机科学家已经发明了按照这类规则来写语法分析程序的技术。（在更实用的例子中，优先规则必须加到定义里，以便解决象“5 * 8 + 1”这类表达式的二义性问题。）

词法分析程序和语法分析程序的基本差别在于词法分析程序按照规则来识别一定的字符序列，而语法分析程序可以按照自引用的规则来识别更复杂的结构。

17.2 lex

lex 程序用以生成词法分析子程序。lex 读入词法分析程序说明，并产生 C 或者 RATFOR 子程序输出。为了使用词法分析程序，lex 输出必须编译并将目标代码和其它程序组合在一起。正如上一节所述的，词法分析程序再把一个字符串作为输入，并产生一串单词作为输出。这三个变换如图 17.1 所示。

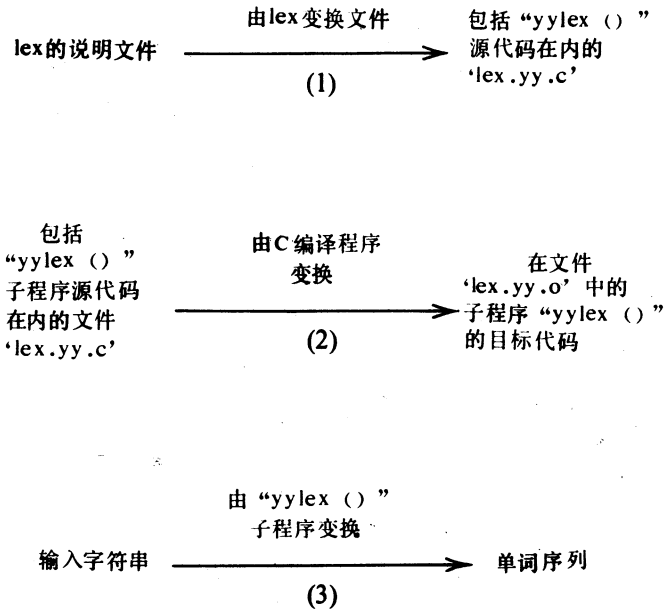


图 17.1 用 C 作为生成语言的 lex 系统要素

要了解的一个重要情况是，lex yacc 产生的是子程序，而不是完整的程序。产生子程序的优点是：这样可以在词法分析或语法分析以前、以后、或期间加进专用代码。和 lex 及 yacc 相似，其它词法、语法分析的生成程序将产生完整的程序，很难适应于不同的用户。

由lex 变换程序的词法分析规则类似于UNIX 系统中使用正文编辑程序形成正则表达式的规则。规则

```
[0-9]
```

是一个单字符的正则表达式，它与一个数字相对应。规则

```
[0-9] +
```

与一串数字相对应。在lex 中所使用的规则是编辑程序中说明正文模式规则的一个更强的扩充。对于lex 说明中的每一规则，能在目标语言中编出一个动作代码。当这一规则与一个模式匹配时，这个动作代码就被执行。例如，当每次输入中遇到一个数时，下面这行lex说明

```
[0-9] + { printf ("found a number!\n"); }
```

将会生成一个代码去印出一行“found a number!”信息。这类代码很有意思但实用价值有限。当lex 要用作语言翻译程序的一部分时，一个更好的方法是，在每次识别一个项时送回一个值。这个值叫作单词（像上面所讨论的那样）。由一些程序负责反复地调用生成“yylex”子程序的lex，以便获得每一个单词。

若适当地定义了名字“NUMBER”，则lex说明中的下面这一行

```
[0-9] + { return (NUMBER); }
```

将会在输入遇到一个数时生成代码。但数是多少？显然这个代码也必须指出遇到的数值。通常方法（也可以改变）是将数值放在一个整型变量中。这里把这个变量称作“yylval”。用一个字符串“yytext”表示lex 所识别的正文（这里是一个数）。下面使用的标准C子程序“atoi”把一串数字转换成一个整数值：

```
[0-9] + {  
    yyval = atoi(yytext);  
    return(NUMBER);  
}
```

上面这种代码在许多 lex 说明文件的内容中经常使用。在这种说明中，对象的类型与数值两者均可利用。

一个更完整的例子可以使这些想法更清晰。这个例子是一个能识别下列几项的词法分析程序：

1. 数（数字串）。
2. 字 “set” , “bit” , “on” , 及 “off” 。
3. 分号或换行符。

这个词法分析程序将不管空格符或制表符。

我们要做的第一件事是定义词法分析程序送回的单字。我们用名字 “SET” , “BIT” , “ONCMD” , “OFFCMD” , 及 “NUMBER” 来表示这些单字。分号或换行符识别后将送回单字名 “ENDCMD” 。只要规则遇到一些未定义的项目，就送回单字名 “UNKNOWN” 。下面的单字表放在文件 “y. tab. h” 中，这样分析程序及调用它的程序均能使用：

```
# define SET 257
# define BIT 258
# define ONCMD 259
# define OFFCMD 260
# define NUMBER 261
# define ENDCMD 262
# define UNKNOWN 263
```

对这些单字符号已选择了特殊的值，这样不会影响ASCII码的赋值。

一个词法分析说明至少包括两部分：说明及规则。这两部分用一对百分号分开。在说明部分中，C语言的说明必须用分界符 “% {” 及 “% }” 括起来。下面是识别上面各项的lex说明：

```

%{
/*
 * a lex specification to recognize
 * numbers, 4 words, and delimiters
 */
#include "y.tab.h"
extern int yyval;
%}
%%
[0-9]+      { /* rule 1 */
              yyval = atoi(yytext);
              return(NUMBER);
            }
;          return(ENDCMD); /* rule 2 */
\n        return(ENDCMD); /* rule 3 */
set ;      return(SET); /* rule 4 */
bit       return(BIT); /* rule 5 */
on        return(ONCMD); /* rule 6 */
off       return(OFFCMD); /* rule 7 */
[ \t]+    ; /* rule 8 */
;         return(UNKNOWN); /* rule 9 */

```

一些规则值得单独讨论一下。规则 8 将使空格符及制表符跳过。记号 “[\t] +”代表一串空格符或制表符，由于其动作部分是“空”，因此，任何这种序列都将跳过。规则 9 使用元字符“.”来匹配任何未被匹配的东西。在 lex 工作时，是从上到下按规则识别的，这样规则 9 只有在其它规则均不合用时才遇到。

lex 也能接受有二义性的说明。当有几个规则看来都适用时，则实施最长匹配的那个规则，如果几个匹配的规则具有相同长度，那么将实施最上面的规则。规则 9 指定了一个字符的识别，它是最后一个规则，因此具有最低优先级。

很容易写一个 C 语言子程序去识别上面所示的这些项。可是，即使对于十分简单的一些项，lex 说明也比等效的 C 程序短（更易改动）。对于了解 lex 的用户来说，这种说明将比 C 程序更容易写。这个说明只是体现了 lex 的能力。对于许多应用来说，写等效的 C 程序是十分困难的。一些更高要求的应用，可采用甚至鼓励使用 lex，而不使用一个普通的 C 程序。

如果这个 lex 说明放在文件 ‘lexdemo.l’ 中，则下面 shell 命令将生成包含 C 子程序 “yylex ()” 在内的文件 ‘lex.yy.c’：

```
lex lexdemo.l
```

名字 ‘lex.yy.c’ 是 lex 输出文件的标准名，和 C 编译程序输出可执行文件的标准名 ‘a.out’ 有点类似。使用下面命令能对 ‘lex.yy.c’ 进行编译并生成一个目标文件：

```
cc -c lex.yy.c
```

现在我们已有了一个 lex 子程序，我们需要一个方法去测试它。在本章的下一节，我们将会看到词法分析子程序是如何和 yacc 生成的语法分析程序连起来的。然而现在我们只是看一下词法分析程序是如何对不同输入进行回答的。下面 C 语言程序是为测试 “yylex ()” 子程序而写的。这个程序反复地调用 “yylex ()”，然后印出与单词有关的信息。

```
#include "y.tab.h"

int yylval;
extern char yytext[];

/*
 *          DEMONSTRATION
 * call yylex() to acquire tokens
 */
main()
{
    int token;
    while(token = yylex())
        switch(token)
        {
            case NUMBER:    printf("Number: %d \n",yylval); break;
            case SET:       printf("Set \n"); break;
```



```

    case BIT:           printf("Bit \ n"); break;
    case ONCMD:        printf("On \ n"); break;
    case OFFCMD:       printf("Off \ n"); break;
    case UNKNOWN:      printf("Unknown: %s \ n",yytext);
                       break;
    case ENDCMD:       printf("End marker \ n"); break;
    default:           printf("Unknown token: %d \ n",token);
                       break;
}
}

```

这个 main 子程序正好是按单词值印出不同信息的多路分支。如果 C 主程序(在文件 ‘lexst.c’ 中) 和子程序 “yylex () ” 被编译了:

```
cc lexst.c lex.yy.o
```

那末可执行程序运行时, 就接收下面输入:

```
set bit 5 on;set20
```

此时产生下面输出:

```

Set
Bit
Number: 5
On
End marker
Set
Number: 20
End marker

```

另一个例子的输入为

```
set bit 3 On
```

产生输出为

Set
Bit
Number: 3
Unknown: O
Unknown: n
End marker

17.3 yacc

UNIX 系统实用程序 yacc 用于生成一个语法分析子程序。
yacc 接受一个语法说明，然后生成一个语法分析子程序的 C 或
RATFOR 源代码。编译这个语法分析子程序，然后和一个调用它

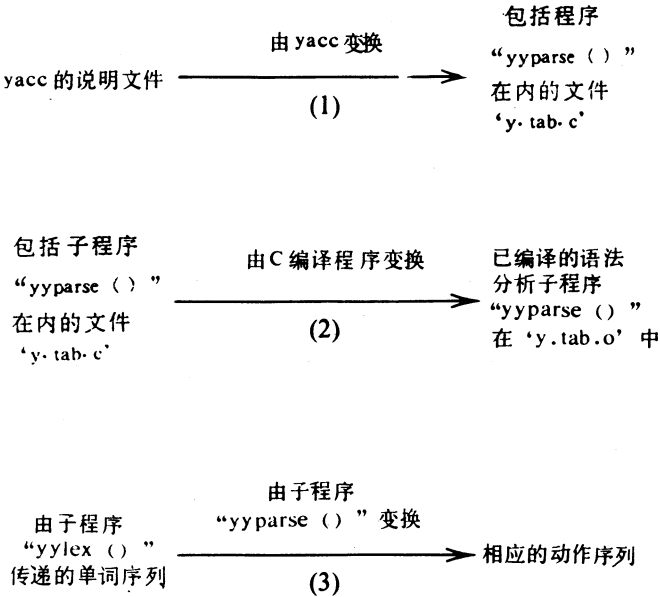


图17.2 用C作为生成语言的yacc系统的要素

以便对输入进行语法分析的程序组合在一起。语法分析子程序调用一个名为“yylex ()”的子程序以便得到单词。注意这个工作可顺利地由 lex 完成，因为 lex 生成一个名为“yylex ()”的子程序。然而，其它名为“yylex ()”的子程序也能使用，并不只是由 lex 生成的一种。使用 yacc 所涉及的三个变换已汇总在图 17.2 中。

lex 说明和 yacc 说明的主要差别是规则的格式不同。在 lex 说明中的规则格式类似于在正文编辑程序中的正则表达式，而在 yacc 说明中的规则是由一串定义链所组成的，这些定义常常可以是自引用的。

例如，一个命令可定义作

```
cmd:    SET BIT numb ONCMD ENDCMD
        |
        SET BIT numb OFFCMD ENDCMD
        ;
```

在 yacc 中，定义由冒号开始，竖线用于分隔定义中可选部分，分号指示定义结束。由“yylex ()”送回的单词通常是大写，由 yacc 说明中所定义的名字通常是小写。

由于这两个可选的定义 cmd 几乎是一样的，因此可象下面那样合并起来：

```
cmd:    SET BIT numb onoff ENDCMD
        ;
```

当然，合并后我们必须提供一个 onoff 的定义，以代替单词 ONCMD 或 OFFCMD。此外，还需要一个 numb 的定义。

```

onoff:   ONCMD
        |
        OFFCMD
        ;
numb:   NUMBER
        ;

```

yacc的说明采用了自顶向下的定义方法，在说明的顶部是最一般的定义，随后的定义是在已建立的定义上扩充和细化。和“yylex ()”送回的单词直接有关的定义通常放在说明文件的末尾。

yacc说明中的每一规则可有一个相应的动作。和lex相象，每当遇到与规则一致的输入时，就相应地执行一个yacc动作。由于yacc的规则可以用另外一些规则来写成，因此yacc有这样一种机制，它可以把一个规则的值送到另一规则。通过对cmd和onoff定义的考察，可以了解为什么需要从一个规则送值到另一规则。显然，cmd的动作需要了解onoff是否代替了单词ONCMD或OFFCMD，它也需要了解numb的值。

一个低级规则（如numb）可送一个值到高级规则（如cmd），这可以通过给一个伪变量\$\$赋值来实现。

```

numb:   NUMBER
        { $$ = yylval; }
        ;

```

一个高级规则能从伪变量取得数值，\$1是定义中第一个规则的伪变量，\$2是第二个，等等。

```

cmd:    SET BIT numb onoff ENDCMD
        {
          printf("Val %d returned by numb rule.\n",$3);
        }
        ;

```

至此yacc说明的规则部分已讨论了，现在可以举一个例子。在许多程序设计领域，必须交互地将值赋给复杂数据结构的元素。例如，一个控制生理刺激器的程序，要有控制刺激器的定时及振幅的数据结构。为了使用这个程序，数据结构必须给定相应的值。这个过程可以交互地通过一个命令语言来控制：

```
set trial 3 amplitude 10;set bit 5 on trial 3 csr
```

随后的yacc说明是这类应用的大概情况。不用控制整个抽象数据结构，而可以通过用一整型变量来置某位 on 或off，或赋值给变量。每当成功地识别了一个命令时，变量值就被印出来。由随后的 yacc 说明所识别的命令包括

```
set bit 3 on  
set 10  
set bit 4 off ; set bit 0 on
```

像 lex 说明一样，yacc说明文件也包含说明和规则。这两部分由分界符“%%”所分开。yacc说明的第三部分由一个十分简单的C子程序所组成，它调用已经生成的子程序“yyparse ()”去实际执行语法分析。

yacc说明文件的第一部分是说明，它由以分界符“% {”及“%}”括起来的C语言定义及单词说明所组成。为了使yacc和lex所用的单词定义一致，yacc能输出一个它本身定义了已经被其它程序说明过的单词值的文件。在我们这种情形下，就是包含在lex说明里的文件‘y.tab.h’。lex和yacc很容易在一起工作。

下面的yacc说明文件比本书的其它例子更长也更难懂。然而，如果了解了上面提及的许多思路，并且也熟悉C语言，那么你会发现，要弄懂它并不是太困难的。

```

%{
/*
 * Yacc Specification File
 * - The First Part -
 *   Declarations
 */
int testvar = 0;
int yyval;
#define Off 0
#define On 1
%}
%TOKEN SET,BIT,ONCMD,OFFCMD
%TOKEN NUMBER,ENDCMD,UNKNOWN
%%
/*
 * - The Second Part -
 *   Rules
 */
session: /* Rule 1 */
|
| cmds
|
| cmds cmd /* Rule 2 */
|
| cmds cmd
|
| cmd: ENDCMD /* Rule 3 - Alternative 1 */
| { /* the null cmd */ }
|
| error ENDCMD /* Rule 3 - Alternative 2 */
|
| SET BIT numb onoff ENDCMD /* Alternative 3 */
| {
| if (($3 <= 15) && ($3 >= 0))
| {
| if ($4 == Off)
| testvar = testvar & ~(1 << $3);
| else
| testvar = testvar | (1 << $3);
| }
| else printf("Illegal bit number: %d \n",$3);
| printf("Testvar - %o \n",testvar);
| }
|
| SET numb ENDCMD /* Rule 3 - Alternative 4 */
| {
| testvar = $2;
| printf("Testvar - %o \n",testvar);
| }

numb: NUMBER /* Rule 4 */
| { $$ = yyval; }

onoff: ONCMD /* Rule 5 - Alternative 1 */
| { $$ = On; }
| OFFCMD /* Rule 5 - Alternative 2 */
| { $$ = Off; }

%%
/*
 * - The Third Part -
 *   Support Subroutines
 */
main()
{
  yyparse();
}

```

如果这个说明文件存放在文件 ‘yaccdemo.y’ 中，则命令
yacc -d yaccdemo.y

将产生 C 语言的源文件 ‘y.tab.c’ 及单词定义文件 ‘y.tab.h’。命令

```
cc -o yaccdemo y.tab.c lex.yy.c
```

将产生文件 ‘yaccdemo’，它在使用时将被执行。

有几个规则是显而易见的。规则 1 和 2 简单解释如下：一个 session 可以是一个空事件或一串命令；一串命令 (cmds) 可以是一个命令或几个命令。

许多真正要完成的工作是在规则 3 中。规则 3 的第 3、4 任选项实现了两类命令：置位命令及赋值命令。第 1 任选项允许空命令，空命令很容易遇到，例如命令

```
set bit 3 on;
```

并不是非法的（这个输入导致一个空命令产生，因为命令 “set

bit 3 on” 是以分号作结束，后面又是一个重新开始的命令并立即以换行符作结束）。

规则 3 的第 2 任选项说明一个简单的错误恢复技术。名字 “error” 是 yacc 内固有的，任何时候只要遇到一个输入错误，就好象识别出错误单词一样。在我们这个简单语言里，如有以下输入则将引起错误

```
bit set 3 on
```

这是因为它的关键字的次序不对。当有下面输入时也将引起错误：

```
please set bit 3 off
```

这是因为字 “please” 是不能识别的。对上述所有错误的处理是按规则 3 的第 2 任选项来完成的，直到遇到单词 ENDCMD（分号或换行符）为止。这种做法是把一个已经出错的命令的剩余部分都扔掉，对于我们这个简单命令语言来说，这是一种简化而适当的办法。更加复杂的出错处理方案也是可以实现的。

第十八章 系统管理员用的实用程序

一个计算机系统的管理包含了许多工作，它远远不只是在每天早晨用户开始注册前开启电源而已。系统管理员要负责整个系统的维护，配置新的软件，改动软件以适合当地的条件，对用户文件周期地进行后援，恢复丢失的数据以及通知用户有哪些新的服务和设备。在一些单位，这些职责是分散给许多人干的，而另一些单位则有专人负责这些工作。

由系统管理员进行的许多操作不应该由普通用户执行。在UNIX系统里，这是由一个专门的特权阶层，称作超级用户来完成的。UNIX系统的超级用户并不受正常文件存取系统的限制，而被允许执行全部系统的维护功能。本章所讨论的全部程序应该为“root（根）”所有，改变系统状态的程序（volcopy，fsck，fsdb等）只能由具备超级用户特权的人员执行。

获得超级用户特权可有两种方法，一种是使用名字“root”来运行机器，另一种是使用命令su。在得到超级用户许可之前，用户不管使用上述哪种方法，都必须送入超级用户的口令。超级用户的口令应该只让那些得到委托而且有经验的UNIX系统程序员知道。

对于系统管理员来说，整个系统的维护可能是最繁重的活动；周期地执行文件的后援可能是最麻烦的。这一章并不解释如何去解决每个文件系统可能出现的问题，也不解释如何去建立一个可行的后援策略，这需要通过试验、出错、再改进的一个反复过程来研究，如果已经有了很好的后援，就不用担心文件系统损坏了。如果将一个极差的系统打了很多补丁，那么后援就很难理解。防止丢失数据的最有力的方法是保存一个好的后援，并了解文件系

统修复的过程，因为每一个UNIX系统装置总会有文件系统的损坏问题。为UNIX系统管理提供一个完整的介绍已超出了本书的范围。这一章的目标只是为系统管理员提供某些实用程序和过程的入门介绍。

18.1 安全性

一个系统的安全性指的是在不良情况下的可靠性。如果一个系统出现意想不到的失效，或丢失文件，或允许未被授权的文件的存取，那么这个系统是不安全的。一个安全的系统，工作可靠，并维持和保护用户的文件，安全性是一种相对的标准，在世界上尚未发现一个完全安全的计算机系统。

UNIX系统的可靠性（对抗系统瘫痪的能力）是很高的。UNIX系统由标准软件模块生成，常常运行几个月而不出问题。这个结果很好，可以和计算机硬件的可靠性相比。但由标准软件模块和经过局部修改的软件模块组合在一起的UNIX系统，可靠性一般较差。

UNIX系统不是一个容错计算机系统。UNIX很大程度上依赖于计算机硬件，当硬件失效时它将受影响。最近，UNIX系统包含了出错的记录，它在系统变坏前发现硬件故障方面具有更好的性能。

计算机系统丢失文件有三个基本原因：硬件故障，操作系统故障，计算机用户故障。硬件故障是不可避免的，操作系统故障可能会产生，计算机用户故障是可以查明的。计算机制造者和操作系统设计者的目标是尽可能减少由硬件和软件所造成的数据丢失；和用户意外地删除文件相比这种丢失是极少的。UNIX系统的一个特殊之处是：从用户观点来看，所有输入/输出都是同步的，而从系统观点来看，所有输入/输出都被暂存起来并且是异步的。由核心实现数据缓冲，很可能在计算机意外停机时引起数据丢失。这类丢失是不希望有的，但如果要求保持系统的有效性，

这又很难避免。

系统安全性受影响的另一方面是和那些有意破坏或恶作剧的用户有关。虽然敲打一个系统往往是人们的一种自然反应，但是敲打一个多用户计算机系统常常会干扰其他用户，因此要极力阻止。一些简单的预防措施可以阻止大多数问题的发生。

在较早的 UNIX 系统里，一个用户往往由于过多要求资源（如进程表的登记项，文件系统的存储空间等）而引起服务中止。在中止以后，调查进程表以发现问题的根源是很容易的，问题的根源多半是由于有问题的程序（buggy）引起的。这些技术在 UNIX 系统第七版本里没有什么作用，因为要求过多资源时，你需要有超级用户特权。

为保护系统的安全，有几个普通显而易见的对策：应该限制知道及使用超级用户口令的人数，应该限制对系统控制台的存取。也不要留下无人管理并具有超级用户特权的系统控制台（或任何这类终端）。计算机运行时，必须有受委托的人员的管理。例如，被允许在晚上单独使用计算机的人，能通过暂停计算机来正常地中断安全的系统，然后将计算机引导至单用户方式工作。

也应防止用户（通过安装磁带或磁盘）移入那些能以特权方式运行的程序。破坏 UNIX 系统安全性的一个最容易的方法是移入不要求口令的 su 命令（超级用户，见下面论述）的特权版本。使用根所拥有的调整用户标识（set user id）方式的程序（su, mail 等）是危险的。这些程序不应该具有写特权，没有写特权，它们就不能被一个普通用户来更改了。find 程序能用来定位所有这样的程序，它们在一个系统中使用调整用户标识方式。

18.2 su——成为超级用户

UNIX 系统的超级用户是具有最高特权的用户。超级用户能执行许多普通用户不能进行的操作。本章后面部分所讨论的许多操作都必须由超级用户来执行。另外，本书其它地方提及的几个

命令，当由超级用户执行时它的功能也是不同的。例如，普通用户使用date命令显示日期和时间。而超级用户却能用date命令置日期和时间。

普通用户受到UNIX系统的文件存取保护系统的约束。而超级用户不受这个保护系统的约束。超级用户能改变任一文件的方式，或用任何方法去存取任一文件。未加以正常约束的超级用户命令会引起很大的损害。超级用户特权应只限于个别细心并得到委托的人员所有。

有两种方法来获得超级用户特权：一种方法是使用专门名字（root）进行注册。另一种方法是使用正常的注册名字，然后执行su超级用户命令。这两种情况，系统都要求送入超级用户的口令，如你送入正确的口令，则系统将显示一个特殊的提示符（一般是“#”号）。当要结束超级用户的操作时，应该按control-d键。如果以“root”注册，按此键后，则将退出；如果用su命令，按此键后，则将恢复你正常的身份。

18.3 安装及拆卸文件系统

mount及umount命令用于控制磁盘存储卷与文件系统的逻辑连接。在讨论这两个命令前先讨论另一些问题。一个文件系统是文件及目录的集合。小型磁盘（如软盘）通常包含一个文件系统。大型磁盘通常分成几个区，每个区包含一个文件系统。大型磁盘是为了方便起见才分成几个区的，划分后的磁盘容易实现后援及修复损坏的文件系统。UNIX系统并不强求你划分大型磁盘，但许多管理者都喜欢划分磁盘。

将UNIX系统核心装到存储器，从而启动操作的过程叫引导（booting）。引导过程的细节，各个计算机是不一样的，但所有系统都是将结果装入存储器，对于UNIX系统来说，就是它的核心（一般存在文件‘/unix’中）由根文件系统装到存储器中。根文件系统包含了UNIX系统最基本的实用程序和文件。在引导

过程结束后，只有根文件系统中的文件能被存取。

mount命令用于通知UNIX系统核心，有另外一个文件系统存在，它应被合并到可存取的文件系统结构中。一个文件系统，存在但未合并到可存取文件系统结构中，则称为卸下的；一个文件系统被并入可存取的结构中，则称为已安装的。

未被安装的文件系统逻辑地被连接到一个可存取目录之下。通常该目录是空的，并专门是为安装此文件系统而提供的。例如，在一个小型系统里，根文件系统可存放在设备‘/dev/rk0’上，它可包括文件‘unix’及目录‘usr’，‘etc’，‘dev’及‘bin’。目录‘dev’包含了全部为输入/输出设备而建的特别文件。目录‘bin’包含了最频繁使用的命令。目录‘etc’包含了用于系统初始化及系统管理的文件。目录‘usr’是空目录。存放在设备‘/dev/rk1’上的文件能用下列命令将它连接到空目录‘usr’之下：

```
/etc/mount /dev/rk1 /usr
```

(注意：命令mount通常存放在目录‘/etc’内。在某些系统里，超级用户的查找路径包括了‘/etc’目录，因此可用名字“mount”来代替‘/etc/mount’)。这个mount命令的第一个自变量项(/dev/rk1)包含一个文件系统的专门设备名。第二个自变量项(/usr)是该文件系统将连接的目录名。在引导一个文件系统后立即能看到的情况如图18.1所示。图18.2所示的文件系统是一个已经将设备‘/dev/rk1’安装在目录‘/usr’上进行了逻辑扩充后的系统。

可以把一个文件系统安装成为只能读不能写的系统。这个功能可经常用在下述情况下：你要从后援中恢复数据，并要绝对地确保这个后援不被破坏。当一个文件系统被安装为可读/写，那么一旦文件被存取，即使文件未被明显写入，文件的存取日期也要被修改。把文件系统安装为只读，可防止这种修改，因此当你不想破坏文件的存取日期时，这个文件系统亦应以只读方式安装。

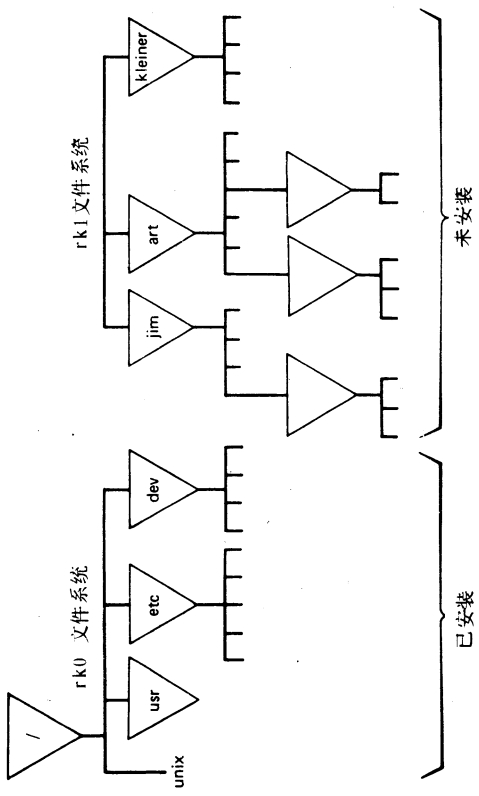


图18.1 在rk 1 安装前的文件系统rk 0 及 rk 1

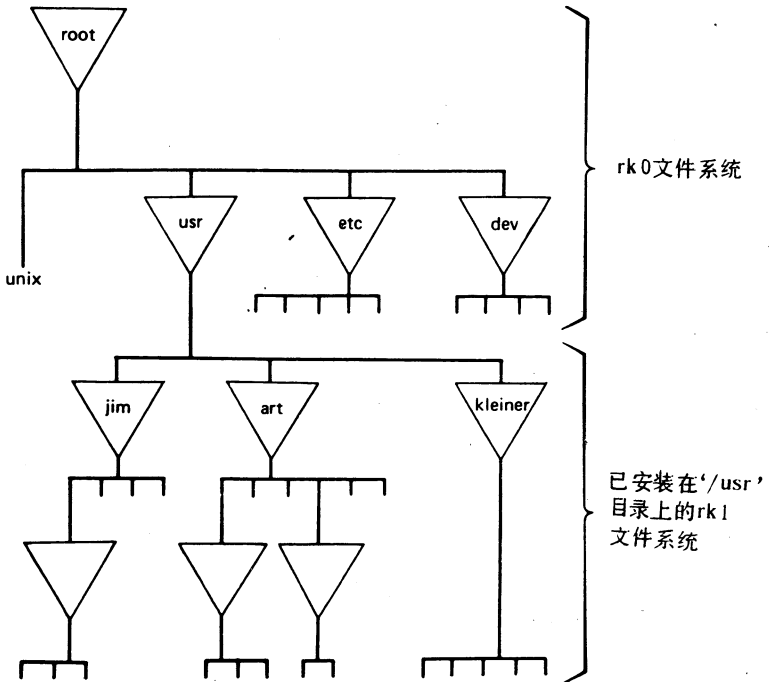


图18.2 rk 1 被安装在 ‘/usr’ 上以后的文件系统rk 0 及rk 1

命令

```
/etc/mount /dev/rk2 /mnt -r
```

将设备 ‘/dev/rk 2’ 上的文件系统，按只读方式安装在目录 ‘/mnt’ 之上。

UNIX 系统核心通常不识别磁盘或磁带驱动器上的任何写保护开关或机构。如你安装一个文件系统为读/写，然后使用磁盘驱动器上的开关来进行写保护，那么，当每次存取文件时，将产生一连串错误，这是因为硬件机构的写保护功能阻止操作系统修改文件的存取日期。如果使用磁盘驱动器上的写保护开关，那么你

也必须逻辑地安装文件系统为只读。根文件系统决不能安装为只读。

若mount命令不包含自变量:

```
/etc/mount
```

则全部已经安装的文件系统名字被印出:

```
/dev/rk0 on / read/write on Thu Dec 3 19:08:21
/dev/rk1 on /usr read/write on Thu Dec 3 19:09:03
/dev/rk2 on /mnt read/only on Thu Dec 3 19:10:55
```

这个例子中的根文件系统是‘/dev/rk0’,它决不能被拆卸下来,而其它文件系统,假如无人使用它们的话,则可以拆卸下来。命令

```
/etc/umount /dev/rk1
```

将试图拆卸‘/dev/rk1’文件系统。如果任何人在‘/dev/rk1’上有一个已打开的文件,或有一个当前的目录,那么umount命令将会指出‘/dev/rk1’文件系统正‘忙’,不能拆卸。仅有的解决办法是找出谁正在存取这个文件系统(使用ps命令),然后等文件系统空闲后再试umount命令。偶而,在需要拆卸的文件系统上,有一个是进程正在使用的打开文件,如果你不能终止这个进程,那么你就不能卸下文件系统。仅有的补救办法是你去暂停这个系统,然后再重新引导一次。

偶而,你要在一个磁盘上取出数据,而该磁盘上没有有效的UNIX文件系统。没有文件系统的磁盘或磁带,可使用特别设备文件进行存取,但不能被安装。安装是一个逻辑地扩充文件系统的操作,安装所限定的区域实际上是一个UNIX文件系统。虽然磁带能包含文件系统,但通常它被看作特大文件而不予以安装。

当你已安装了一个文件系统时,你要极其仔细,在进行逻辑拆卸操作以前,不要移走物理介质(磁盘)。当一个文件系统已经

安装时，有关这个文件系统上文件位置的重要信息与核心一起保存在存储器上。不执行拆卸操作就移走物理介质，可能引起有关该文件系统的存储器常驻信息的丢失。拆卸操作的一个目的是将全部存储器常驻信息送入物理介质上，这样，每一个内容就都不会丢失。UNIX 系统中的文件系统被弄坏的主要原因之一就是：没有做到先逻辑地拆卸再物理地拆下设备。

根文件系统是不能卸下的，因为它包含了系统运行所必须的程序和文件。如果你想改动根部分，只有先停止该系统，重新装入根部分，再重新引导该系统。因此，没有两个磁盘（或一个磁盘及一个磁带）实际上就不可能运行UNIX 系统。

当执行一个安装操作时，有关安装的信息被记在两个不同的地方。UNIX 系统的核心包含了已安装文件系统的存储器常驻信息表。一些附加信息亦记在这个表中。可是，UNIX 系统没有允许用户程序读内部安装表的内容的系统调用，因此用户程序不可能真正知道文件系统被安装了什么内容。一旦mount命令已安装了文件系统，则安装的记录就放到根文件系统的普通磁盘文件内（通常是‘/etc/mnttab’）内。当系统工作时，它工作得很正常，但任何时候信息都存放在两个不同的地方。因此信息有可能不同步。当送入命令：

```
/etc/mount
```

时，磁盘文件内的信息被印出，偶然它和内部系统安装表(gospel)包含的信息是不同的，这个问题很可能持续到系统重新引导为止。

18.4 sync——誊清系统缓冲区

sync命令将有关文件系统的存储器常驻信息送入物理介质内。在暂停系统之前总要去执行 sync 命令。UNIX 系统运行经验表明，为确保可靠起见，应执行两遍sync命令，这是因为sync命令完成时，并不保证信息已实际写到了磁盘上，虽然那时这个工作一定已被安排了。在执行sync命令以后，你要等待磁盘工作灯灭

了（假定有控制面板的话），再去真正暂停这个计算机。

一旦你感到系统失效是不可避免时，就应该使用sync命令，这可能是你要执行一个有某些疑点的程序，或许是你的直觉告诉你，系统将要失效。在任何情况下，慎重地执行sync决不会有任何坏处。

在暂停机器以前，应该执行sync命令，这条规则有一个例外，当你用‘icheck’程序或者‘fsck’程序重新构成空闲块表时，这时磁盘上的信息比存储器的信息更及时，因此在暂停机器之前不用sync。

18.5 mknod —— 建立特别文件

在UNIX系统里，特别文件连接输入/输出外部设备和操作系统。有两种不同的外部设备接口：块接口及字符接口。块接口按块（512字节）将数据传递给外部设备，它主要用于磁盘和磁带。字符接口用于将数据每次一个字符地传递给外部设备，它主要用于终端、打印机、纸带读入机、及其它按字符存取的设备。另外，具有块接口的磁盘及磁带，通常也有字符接口（也叫原始接口）提供给许多系统维护的程序使用。

通常，特别文件是在系统生成时或在硬件配置改变时建立的。特别文件一般在目录‘/dev’中可找到。如你送入命令

```
ls -l /dev
```

你会得到一个全部特别文件的长列表。这个表格的存取方式项中，若以字符“b”开头，则表示是一个块接口的特别文件，若以字符“c”开头，则表示是一个字符接口的特别文件。下面是我的系统上特别文件的部分表目：

crw--w--w-	1	bin	sys	0,	0	Jan	8	09:13	console
crw-rw----	1	bin	sys	3,	1	Jan	2	12:01	kmem
c-w--w----	1	bin	sys	5,	0	Jan	7	23:15	lp
crw-rw----	1	bin	sys	3,	0	Dec	7	11:11	mem
brw-rw----	1	bin	sys	1,	0	Jan	8	09:13	mt0
crw-rw-rw-	1	bin	sys	3,	2	Jan	8	09:00	null
brw-rw----	1	bin	sys	2,	0	Jan	8	10:15	rk0
brw-rw----	1	bin	sys	2,	1	Jan	8	10:15	rk1
crw-rw----	1	bin	sys	4,	0	Jan	8	09:44	rmt0
crw-rw----	1	bin	sys	6,	0	Jan	8	10:15	rrk0
crw-rw----	1	bin	sys	6,	1	Jan	8	10:15	rrk1
crw--w--w-	1	bin	sys	1,	0	Jan	2	03:20	tty
crw--w--w-	1	bin	sys	0,	1	Jan	5	15:49	ttyi
crw--w--w-	1	bin	sys	0,	2	Jan	8	10:17	tty2
crw--w--w-	1	bin	sys	0,	3	Jan	8	10:20	tty3

在一个普通文件的长列表里，文件的大小以字节为单位印在所有者及同组用户的有关信息之后。一个特别文件由于它不占存储器区域而仅是与输入/输出设备连接，因而没有文件的大小。而在长度位置上，印出了主设备号和次设备号。

主设备号表明哪类硬件和这个文件相联。下表指出上面提到的块特别文件及字符特别文件的主设备号。

主设备号

文件

字符特别文件

0	console, tty1, tty2, tty3
1	tty
3	mem, kmem, null
4	rmt0
5	lp
6	rrk0, rrk1

块特别文件

1	mt0
2	rk0, rk1

你可以看到常常有几个接口是同一种主设备号。四条通信线 (console, tty1, tty2, tty3) 都有主设备号 0。这意味着在这个计算机里有四个相同的硬设备在管理这四条通信线。为区分四条线路, 这四个特别文件的次设备号从 0 到 3。次设备号指明了几个相同设备中的哪一个。

这一部分我们可以通过看目录 '/dev' 中的信息, 去推断设备的主设备号和次设备号。要建立特别文件首先必须看一下系统组成文件 (常放在名为 'conf.c' 的文件内)。一旦你知道了主设备号及次设备号, 你就可以用命令 mknod 去建立特别文件。假定你已为纸带设备建立了一个字符特别文件。查阅系统组成文件显示主设备号是 7, 次设备号是 0 (只有一个纸带设备)。命令

```
/etc/mknod /dev/pt c 7 0
```

将为纸带设备建立字符特别文件 '/dev/pt'。

在带有已命名的管道 (fifo) 的系统上, mknod 命令也用于建立管道。命令

```
/etc/mknod /dev/fifo1 p
```

将建立 '/dev/fifo1' 文件。

18.6 df——印出磁盘空闲区

df 命令印出在联机文件系统上空闲区的情况。这个情况包括空闲块的数目和空闲节点 (inode) 的数目。命令

```
df /dev/rp3
```

将印出在 '/dev/rp3' 文件上的空闲区为:

```
/tmp on /dev/rp3 456 blocks 345 inodes
```

在这个例子里, '/dev/rp3' 文件安装在目录 '/tmp' 上, 它有 456 个空闲块及 345 个空闲节点。

如果你送入下面命令也能印出同样结果:

df /tmp

这是因为df允许自变量是文件系统（例如‘/dev/rp3’）名，或者是安装了文件系统的目录名（例如‘/tmp’）。注意，不能随便送入一个目录名，而必须是已安装了磁盘区的目录名。

也可以使用不指定文件系统的df命令，此时将列出全部已安装的文件系统上的空闲区情况。

系统管理员应该定期执行df，以监视全部文件系统上空闲区的情况。一些系统需要20~30%的空闲区，而另一些系统只要5~10%的空闲区，经验往往向你提供最好的帮助。一些系统对个人使用的磁盘空间要加以限制。命令du可用来检验是否遵从这些限制（见第7.16节）。

要建立特大文件的用户应使用df以确定是否有足够的空间。使用df并不需要超级用户的特权。

18.7 volcopy, labelit, dump, restor, cpio——后援

有很多方法来实现定期后援。后援可以恢复丢失的数据。不经常进行后援，会使原有后援老化或过时，而频繁地进行后援又会干扰正常的系统工作。每个系统管理员必须根据下述情况找出折衷的后援间隔：数据丢失的频度，数据的价值，及在后援过程中所花费的代价。同时还必须找到一个能和其余硬件一起工作的后援过程。对于只有很少一组磁盘的系统来说，用户进行后援工作，必须停止系统工作。而对于有较多的硬件的系统来说，用户只须停止系统的一部分工作，即能进行后援工作。

UNIX系统具有三个主要后援实用程序：volcopy/labelit系统，dump/restor系统及cpio程序。volcopy程序把整个文件系统从一处复制到另一处。volcopy在复制操作中，将按规格来检查文件系统的标号（用labelit标志），以确认是否安装了正确的文件卷。dump程序实现文件系统卸出保存操作，只有那些修改日

期比某一日期更新的文件才予以保存。**restor** 程序能检查由 **dump** 程序所建立的卸出区,以复原某一文件或整个文件系统。**cpio** 程序建立一个十分大的文件,它包括整个文件系统(或部分)的翻版。

volcopy 系统的一个优点是,每次执行后援时,整个文件系统被保存下来。这种冗余措施可以构成十分安全的后援。**volcopy** 系统的另一个优点是它能用来构造复制到磁盘或磁带上去的副本。盘到盘的后援操作是最频繁的,也是极快的。**volcopy** 的最后一个优点是极其容易恢复丢失的数据。由于保留了整个系统,因此,为了恢复整个文件系统,就可以安装保留的文件系统以便从后援介质上恢复某一文件,或者恢复整个文件系统。从盘到带的 **volcopy** 复制对于长期的数据存档是十分有用的,但对于每天的例行工作却不方便,因为从后援带上很难删除单个文件。

增量卸出系统的优点是:在每次后援期间只有很少的数据传送到后援介质上,这是因为只有最新修改的文件才被保留下来,因而实现后援所花费的时间是很少的。**dump/restor** 系统的一个缺点是只能把后援结果送到磁带。盘到盘的后援必须使用 **volcopy**。用 **restor** 恢复单个文件是复杂的,因为丢失的文件可能在几个磁带中的任何一个上,它依赖于该文件上次修改的时间。**restor** 程序将浏览各种清理后的磁带,并指出哪个磁带上丢失的文件。恢复文件系统也是困难的,因为后援可能散布在几个磁带上。

cpio 程序用于在磁带上保存一个文件系统是十分方便的。**cpio** 能建立一个很大的文件,它包含了整个文件系统的映象。因为磁带通常当作大文件来操作,用 **cpio** 来产生大文件是理想的。**cpio** 也能用于从档案中取出单个文件。当单个文件需要恢复时,用它对磁带操作比用 **volcopy** 更为方便。**find** 程序有一个任选项,能自动地引用 **cpio** 实现后援操作。

18.8 dd——转换文件

经常用磁带在 UNIX 系统计算机和其它计算机之间传送数据。

其间存在的主要问题是：UNIX 系统格式和非UNIX 格式之间的转换。例如，从大型系统到UNIX 系统经常要输送穿孔卡片的数据，传送给磁带的穿孔卡片数据一般是按80个字符作为一块存放的。在UNIX 系统中用这种数据时，其编码必须是ASCII 码（很多穿孔卡片采用EBCDIC码），并且每行（卡片的映象）需要尾随一个换行符。dd 程序执行这些转换。

在UNIX 系统的早期版本中，dd也作为后援使用，因为从一个文件系统把数据移动到另一个文件系统或磁带上，它都是一种十分有效的办法。在UNIX 系统的第七版本中，已用程序volcopy及dump代替dd来进行后援实现备份，但dd仍用于转换。

18.9 fsck, fsdb——检查文件系统

每个文件系统早晚要退化。系统管理员要负责维修退化的文件系统。有许多原因造成文件系统退化：硬件间歇性故障，电源线上浪涌电流或其它不规则情况，磁盘介质的问题，以及未执行sync命令就关闭计算机等等。退化了的文件系统应立即给予维修以减少数据的丢失。有时一个文件系统变得漏洞百出以至必须被抛弃，并要从后援中整个地予以复原。但通常只有一个或两个文件被丢失，有时甚至并未丢失什么。

一个文件系统有点象一个普通的分类帐。和分类帐相似，一个文件系统应该是一致的。在一个退化的文件系统中，有一些（或许是一个数据块）没有被记上，或许有的被记了两次。修复一个退化的文件系统要克服其不一致性。当计算机每次引导时，应检查一下文件系统，每当发现问题时就应立即维修。

维修文件系统有几种程序。在UNIX 系统第六版本中，程序icheck, dcheck, check, ncheck 及fcheck被用于检查和维修文件系统。在UNIX 系统第七版本中，这些程序已为程序fsck所代替。另外，在UNIX 系统第七版本中，还有程序fsdb，用它对一个文件系统进行查错及实现精细的维修。fsdb只能供那些完全

懂得文件系统结构的人员使用。

所有文件系统的检查和维护应在卸下（静态）的情况下进行。在未用fsck前就要维修文件系统是困难的，因为修复一个退化的文件系统需要用几种程序并要很好地判断。使用fsck要容易得多，因为当它发现文件系统有问题时，能提出解决办法。除非你对文件系统有足够的了解，才能使用fsdb，否则在大多数情况下，你应按上述有关fsck的建议去做。

如果你了解了UNIX文件系统的格式，那么文件系统的维修将更易理解，fsck的原理是删去已坏或将坏的文件。删去是简单并且有效的手段，但通常会造成少量数据丢失，偶尔也会造成数据严重丢失。一个合适的文件后援系统应使数据丢失不造成什么问题。

文件和它的名字偶尔变得不一致，在UNIX系统里这是可能的，因为文件名是存在目录里，而文件的其余信息存放在一个叫节点的结构里。没有名字的文件称为孤儿（orphaned）文件，fsck将把它们放在一个叫做‘lost + found’的目录里。在使用fsck之前，应在根目录上建立目录‘lost + found’，并要确认在这个目录里有空位置以便让某些文件可以复制到‘lost + found’中，然后再删除这个文件。如果未提供‘lost + found’，则孤儿文件被丢弃。在目录‘lost + found’中fsck将用一个数作为文件名。你应根据它的内容试图决定它的真正名字，然后再把此文件存到它的合法所有者那儿，或者删除它。命令

```
fsck
```

将检查在文件‘/etc/checklist’中提及的全部文件系统。通常‘/etc/checklist’包含了全部工作文件系统的清单。你也可检查单一的文件系统，这时要用该系统名字作为fsck的自变量：

```
fsck /dev/rk2
```

每个文件系统包含一个空闲块表。当一个文件建立时，用于文件的数据块就从这个空闲块表收集拢来。偶尔这个空闲块表受

到破坏：一些块是空闲的但却不在表内；或者一些块已在某文件中使用但又在这个表中登记了。fsck的任选项“-s”将重建这个空闲块表。命令：

```
fsck -s /dev/rk2
```

将重建/dev/rk 2的空闲块表（fsck的任选项“-s”和较老的程序‘icheck’任选项“-s”相同）。当重建一个已安装的文件系统（通常是根文件系统）的空闲块表时，你必须在重建后立即暂停计算机，然后从暂存文件中再引导系统。立即暂停计算机（不用通常的sync操作）将防止UNIX系统核心把老的（坏的）空闲块表写到磁盘上。通常当fsck在检查一个文件系统过程中发现了错误时，它将要求你同意去重建这个空闲块表。

18.10 cron——在指定时间运行程序

程序cron用于在指定时间执行程序。通常在系统引导后马上就要启动cron程序，并且在一个系统里应该只有一个cron进程在运行。cron从文件‘/usr/lib/crontab’内读命令。在crontab内的每一行，说明一个命令及它应该执行的时间。cron定期地考查‘/usr/lib/crontab’，以便跟上新加到文件里的命令。也就是当cron运行时，可改动‘crontab’文件。

在‘crontab’文件的每一项包括五个字段以指定命令应该运行的时间，按次序，这五个字段表示分（0—59），小时（0—23），日（1—31），月（1—12），及星期（0—6，星期日当作0）。这五个时间字段用空格或制表符予以分隔。五个字段的后面是一个命令。

下面是‘crontab’中的一行，它在控制台上每隔10分钟印一次日期：

```
0,10,20,30,40,50 * * * * date > /dev/console
```

代表时间的数可以是上面提及范围内的一个数，也可以是用逗号分开的一组数（如例子所示），或者是用“-”分开以表示一个时间

范围的两个数；星号代表一个合法的值。在控制台上星期一至星期五每晚 6 点到 10 点：每间隔一小时印一次日期的命令为：

```
0 6-10* * 1-5 date>/dev/console
```

cron 系统经常在半夜无人注意时去运行程序，如执行定期记帐操作等等。一些系统使用 cron 在高峰期间去掉一些程序以改善响应时间。

18.11 先进先出 (fifo) 文件

fifo 文件允许一个程序传输信息到另一个程序。典型的是一个程序写入 fifo，而另一个程序从这个 fifo 中读出。fifo 在这两个程序间建立了一个通信通道。在文件 fifo 中的信息只存了很短一个时间，也就是从一个程序输出给 fifo 到另一程序从 fifo 输入这段时间很短。fifo 只在很少的几个 UNIX 系统上存在。

显然，fifo 文件与 UNIX 系统管道类似。使用 fifo 的优点是任何两个无关的程序能用它进行通信。管道线要求在管道两侧的程序有同一个父程序，通常这个父程序是 UNIX 系统 shell。

能用命令 `mknod`（参见第 18.5 节）来建立 fifo。fifo 只由某些人所有，并由普通三级保护系统来管理对它们的存取。

在使用 fifo 之前必须先建立它。例如下列命令从通信线 ‘/dev/tty99’ 读数据，并把它们写到名为 ‘fifo1.tel’ 的一个 fifo 文件里：

```
cat</dev/tty99 > fifo1.tel
```

如果当命令送入时，文件 ‘fifo1.tel’ 并不存在，则 shell 将建立一个名为 ‘fifo1.tel’ 的普通文件，以接受 cat 命令的输出。

用输入重新定向命令可从 fifo 取得信息：

```
sh<fifo1.tel
```

在这两个例子中我们已用了 shell 的输入/输出重新定向来存取 fifo。

18.12 粘着位(sticky bit)

普通文件并不是连续地存在磁盘的相邻部位上的，而是把文件的信息划分成块，再把块分布存放在磁盘上。如果要使文件在终端上显示，则就得从磁盘的不同地方取出这个文件。这时，因为读这个文件的请求和其它用户输入/输出请求夹杂在一起，系统的开销是十分低的。可是，如果系统装入的某文件是执行文件，则由于磁盘存储分布所导致的系统开销是较高的。对于要频繁执行的程序来说，用粘着方式将有助于减少不希望的开销。

当一个程序暂时从执行状态挂起时，系统可把这个程序复制到磁盘的暂存区上，这个过程被称作对换 (swapping)。磁盘上的暂存区是相邻地组成的，这样可很快地进行存取。在UNIX系统中，可以给文件指定一种特殊方式，这就是当程序还未执行时，就把这个程序的可执行映象存放在对换空间中。因为这个程序是粘附在对换空间中的，因而这个方式称作粘着方式，控制这一过程的标志位，叫做粘着位。

只有很少一些程序被赋予粘着方式，因为对换空间是很宝贵的资源，如果全部程序都赋予粘着方式，则这些资源很快就被耗尽了。频繁使用的程序（如编辑程序，ls, cat等），常常赋予粘着方式。在大系统里，有更多对换空间可利用，这样可以有许多程序采用粘着方式。在小系统里磁盘空间受到限制，因而粘着方式极少使用。

粘着方式只能由系统管理员赋予一个文件。用户不能自己将粘着方式赋予自己的程序。在ls命令的长格式输出中，文件的类型/方式字段的最后位置上用字符“t”来指示粘着方式。

18.13 调整用户标识(set user id)

有时，UNIX系统的高级存取保护方式会给合法的存取要求带来不便。典型的例子是游戏程序。许多游戏程序要保持一个辅

助文件，它包含了一些属于游戏专有的信息。例如，在一个冒险程序里，辅助文件中的信息表格应该隐藏起来不让好奇的冒险者知道。UNIX 系统保护这类信息的机构可使这个辅助文件只为游戏创造者所有，以便限制对该文件的存取。可是，当有人要用这个游戏时，就需要辅助文件中的信息。

这个问题是简单的；你不玩这个游戏，就应禁止你存取这个游戏的秘密信息。然而，当你玩这个游戏时，应该允许你在游戏所控制的范围内存取这种秘密的信息。

调整用户标识方式解决了这个问题。如果一个文件具有调整用户标识方式，那么当你执行这个程序时即获得了文件所有者的全部特权。调整用户标识方式在许多游戏程序中已经使用了。当你不玩这个游戏时，你不能存取这个已保护的游戏文件；而当你玩这个游戏时（执行），就具有游戏创造者的存取特权。当玩这个游戏时，全部存取是按游戏的规则来构造的，游戏程序本身负责防止你受骗。

一个文件所有者能为他自己的文件置调整用户标识方式。调整用户标识方式是把字符“s”放在ls命令的长列表的类型/方式字段中所有者执行字符“x”的地方。

一个文件的调整同组用户标识方式（set group id）允许执行文件的人员获得象所有者同组用户的同样特权，调整同组用户标识方式基本上是调整用户标识方式的一个极好的变动。调整同组用户标识方式是用字符“s”来代替ls命令长列表中类型/方式字段的文件同组用户执行字符（“x”）。不存在其它的调整标识方式，因为没有别的事情可执行。

第十九章 UNIX系统核心

UNIX系统核心是UNIX的主要组织者。它负责调度进程，分配内存和磁盘存储器，主管主存和外部设备之间数据的传送并且受理进程所需要的服务。本章之所以要研究这个核心，其部分原因是为了回答这样的问题：UNIX系统究竟是怎样工作的？要注意，在本章中所介绍的某些比较特殊的情况，可能并未在所有的UNIX系统中使用。

本章的大部分内容对于UNIX系统管理员和系统程序员有直接的实际应用价值。UNIX系统中偏重于理论方面的问题，比如避免死锁、互斥技术等等，已在里奇和汤普森撰写的论文中作了讨论。

所谓核心，就是UNIX操作系统常驻内存的那部分。与其它操作系统比较，UNIX系统核心提供的服务较少。该核心根本不直接替用户干任何事情，所有的服务是由协调用户和核心之间关系的实用程序提供的。对实用程序进行创建、维护和改进要比对UNIX系统核心进行类似工作容易得多，在核心变得比较稳固之后，添加一些新的实用程序也比较容易。

UNIX系统核心包括约10000行的C语言代码和约1000行的汇编语言代码。这样大小的程序单独一个人就可以理解和维护。许多UNIX系统发行时，带有核心和所有实用程序的源代码，这就允许程序员研究并且根据他们自己的系统进行剪裁。比较起来，其它大多数操作系统就太大了，单独一个人不可能理解或维护这样大的一个系统，而且其它很多操作系统，发行时是怎样就是怎样，不允许剪裁。

19.1 概述

UNIX 系统核心由两个主要部分组成，一个是进程管理部分，另一个是设备管理部分。进程管理部分分配资源、调度进程并且受理进程所需要的服务；而设备管理部分掌管主存和外部设备之间数据的传送。UNIX 系统最大的成功之一是，对不同的计算机，使用几乎一样的进程管理方法。

对于一个给定的计算机，设备管理部分为每一个与该计算机连接的外部设备都保留一个程序模块。每次把一种新类型的 I/O 外部设备连接到计算机上，就得把一个程序模块加进设备管理部分。每当把 UNIX 系统移到一个不同类型的计算机上，就得大量地重写设备管理部分，这是因为不同的计算机通常由于不同的控制原理而有非常不同的外部设备。

针对一组特定的外部设备和缺省的任选项配置 UNIX 系统的过程称为系统生成。系统生成的主要工作是创建一张表，这张表刻划了精确的硬件环境并且为系统规定了一组确定的任选项。

有点象好的小说那样，典型的应用程序有开头、中间和结尾。比起小说来，UNIX 系统更象编织好的挂毯。UNIX 系统不是从头到尾地重复一个孤立的情节，相反，它有许多相互关联的情节，这些情节是为了响应每时每刻的请求而编好的。由于这个核心不是一个简单的顺序程序，因此理解起来比较困难。

UNIX 系统核心持有若干张重要的表格，这些表格协调执行过程中相互关联的动作。事实上，UNIX 系统是由数据来构造程序的极好例子。理解 UNIX 系统要从理解核心所持有的这些表格的信息开始。令人意想不到的，核心的大部分工作其实就是搜索和修改程序表格。本章的以下各节将介绍核心的主要功能以及信息的相应表格。

19.2 用户态和核心态

在任何一个给定的时刻，一台计算机或者在执行一个用户程序（一个进程），或者在执行系统代码（我倾向于把进程这个词留给表示在核心进程表中出现的实体）。当计算机正在执行用户进程中的指令时，我们就称计算机处在用户态，而当计算机正在执行核心中的指令时，我们称它处在核心态。某些计算机（例如PDP-11）处于核心态时具有的硬件特权不同于处于用户态时的特权。另外一些计算机（例如Z-80）就不支持具有硬件特权的两种态，它们处于两种态时的不同之点是指令的来源而不是硬件的特权级。

有若干情况可以导致从用户态向核心态的转换。

1. 从整个系统来看，最重要的也许是系统时钟。不管什么程序在运行，系统时钟总是定期地中断（通常每秒60次）。中断是一种硬件信号，它能够把计算机转向特定的程序。在系统时钟的服务程序运行过程中，进程优先级数被重新计算，并且可能改变进程。在没有其它中断时，系统时钟进行基本的时间片划分，这种划分的时间片可让各个用户共享计算机资源。

2. 每当用户程序需要操作系统服务时，就执行一个系统调用。系统调用的实现细节随各种计算机体系结构的不同而变化，但是，其中间结果总是从用户态变到核心态。执行I/O操作的系统调用经常导致调用进程的暂停执行，同时传递数据。在间歇期间，如果有可能，将会执行不同的用户进程。因此，程序正常的I/O请求，常常会引起分时调度程序动作，以调度新的进程。

3. I/O外部设备的服务请求是引起从用户态转换到核心态的第三种情况。通常，I/O外部设备有一个响应时间，这段时间要比通常的计算机的指令执行时间长得多。其细节随计算机的体系结构不同而有很大的不同，但是在大多数UNIX系统中，每次传送完成就有一个中断（一次传送或是512字节，或是单个字

节，或是一个可变长的字节串)。传送完成时的中断一般是更新某些表格中的状态值，然后可能启动另外一次传送。

19.3 调度和对换

在一台分时计算机系统中，进程要竞争可执行的时间片。调度就是裁决参加竞争的用户进程中哪一个去执行的一系列动作。很显然，调度是一个分时系统的关键成分之一。分时的基本动作是挂起一个进程，并且在以后某个时刻再重新开动它。在大多数分时系统（例如UNIX系统）中，每秒钟要出现多次挂起/重新继续开动，以致在外部看起来好象计算机正在同时执行若干功能。

在任何给定时刻，最多只有一个用户进程在活动，而其它所有进程都被挂起。我们可以把一组挂起的进程分成两个小组，一组是准备运行的，而另一组是被封锁的。一个被封锁的进程可看成是正在睡眠等待着某个事件发生。虽然进程也可能是因为下列原因而等待：（1）等待其子进程死亡，（2）等待一个指定的时间间隔，（3）等待来自其它进程的信号，但通常情况下的事件是一个I/O请求的完成。当事件发生时，该睡眠进程被唤醒。醒来的进程被标志为准备运行，但并不意味着这一个进程会立即开始执行。

理想的情况是所有的用户进程都驻留在内存中，调度程序的作用是从驻留在内存中的这组进程中挑选活动进程。遗憾的是，现代计算机的内存没有大到足以同时存放在UNIX系统中活动着的相当数量的进程。解决的办法是把某些挂起的进程存放到磁盘上，这个过程称为对换。存放在磁盘上的进程在它要执行之前必须重新装入内存。调度系统的责任有两个：调度执行的进程以及调度对换的进程。

简单地讲，分时系统是把时间分成一些时间片，并且把这些时间片分配给进程。UNIX系统还考虑了这样的情况，即等待I/O的进程能不能利用指定给它们的时间片。为了避免对执行

I/O进程的歧视,UNIX系统动态地计算进程的优先级。以便决定在当前活动着的进程被挂起时,哪一个不在活动但已就绪的进程可投入运行。

优先级的计算涉及到从该进程对换进内存后执行时间的总和。已执行了较长时间的进程,其优先级要比长久未执行的进程的优先级低。要注意,刚被换入的进程以及等待I/O没有在执行的进程,它们的优先级较低。对于执行系统调用的用户进程还有这样的情况:即当计算优先级可能有一些偏差时,普通用户进程只会降低它们的优先级,但是由超级用户运行的进程可以提高或降低它们的优先级。

优先级计算的最后效果是,需要大量I/O的进程趋向于执行,直到它们不再能继续为止,而需要大量计算的进程,在I/O进程等待其I/O完成的时间内,力图填补这个空隙。当各种工作都在进行时,系统力图使本身各项工作协调,以便让处理机不空闲,并且给所有进程某些执行时间。

在UNIX系统中,挂起的进程只要极小的开销。这个重要的事实已经对UNIX系统程序设计的本质产生了意义深远的影响。在UNIX系统中,程序都被写成能很好地完成一项工作,大型的、复杂的操作通常由若干一起工作的小型程序进行组合。挂起进程的低开销已促使许多“操作系统”功能从UNIX系统核心中移出并放到普通程序的位置上去。讲一下getty程序,该程序在通信线路上印出一个注册消息,然后等待用户注册。在UNIX系统的每一条通信线上都有一个(通常是挂起的)getty进程,可能还未被注册用户使用。许多系统有50到100个这样的getty进程。如果挂起的进程需要重大的开销的话,那么UNIX系统也许就会与现在极不相同了。

19.4 进程

对UNIX系统可有两种设想:文件系统有“空间”,而进程有“生命”。目录和“空间”牢固的内部连接使用户有可能控制文件

系统。设想进程有“生命”可以执行有用的工作，从而使考虑和控制进程比较容易。我们可以这样说：shell运行程序，编辑程序创建正文文件，ls程序列出目录的内容。上述的所有这些进程就好像它们都是活动的、生气勃勃的组织。我们赋予进程具有生命力不过是一种方便的设想。计算机科学家可能把进程说成为一种抽象数据结构的执行。计算机的活动能力在于硬件，但是智慧在于程序，所以，说软件具有生命力是比较合理的。

进程是UNIX系统中的一种基本概念。即使程序指定是由CPU执行并且存放在内存中，即使磁盘和磁带迅速地在旋转，即使计算机真正在进行工作，但是我们仍说进程在执行，而忽略上述很明显的事实。

UNIX系统核心的存在是为了支持进程的需要。从一个进程来看，核心操作是一种必要的开销；从核心内部看来，进程不过是分了类的、根据一组规则操纵的数据结构。下面有关进程的描述听起来可能比较枯燥，因为从系统内部观点看来，进程只不过是数据，更象会计员的工作单，而不是一种生气勃勃的力量。

进程是处在执行状态的程序。对于一个给定的程序，在某一特定的时刻，可能有零个，一个或若干个进程。在具体工作中，我们只把在UNIX系统核心的进程表中登记的项叫作进程。因此，UNIX系统核心的活动一般都作为进程来执行。

进程所必不可少的信息存放在两个地方：proc表（进程表）和user表（也称为每个进程数据段）中。proc表总在内存中，每个进程都在其中占一项；而每一项详细地登记了一个进程的状态信息。状态信息包括：进程的位置（内存地址或对换区地址），进程的大小，进程的标识号以及运行该进程的用户标识号。每一个UNIX系统由proc表中一定数量的登记项所生成，每一个进程在该表中占据一项，不可能存在比proc表中的登记项数量更多的进程。

每个进程的不常用的信息存放在user表中。为每一个进程分

配一个user表，而只有活动进程的user表才可由核心程序直接存取。参见图19.1 a 和19.1 b。

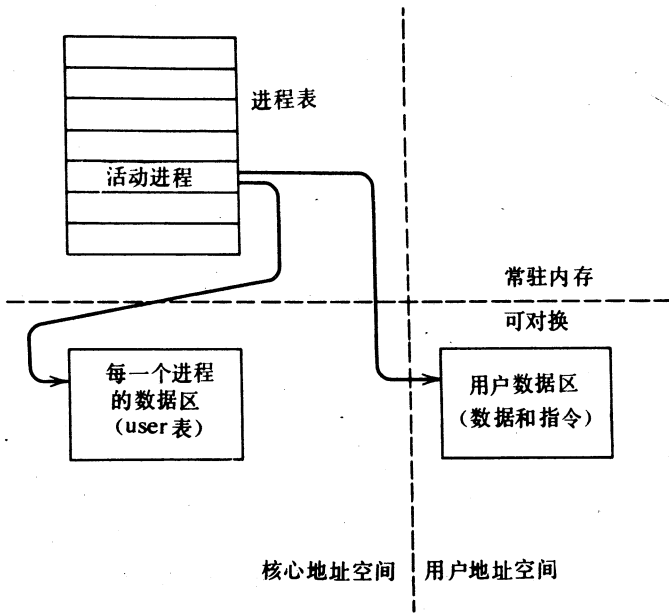


图19.1 a 管理普通进程的数据结构

在进程的所有生存转折点，都要查看proc表。创建一个进程包括初始化proc表中的一个登记项，初始化一个user表，并且为进程建立实际的正文及数据。当进程改变了它的状态（运行、等待、换出、换入等等）或者接收了一个信号时，相互作用都集中在proc表上。当一个进程死亡时，它在proc表中的登记项就擦去了，使得这一项可以由将来的某个进程使用。

proc表必须总在内存中，从而让核心能够管理进程的生存转折点，甚至进程对换出去也如此。在进程不活动时，其生命中的许多事件也会发生。例如，一个进程在等待 I/O 时要睡眠。I/O

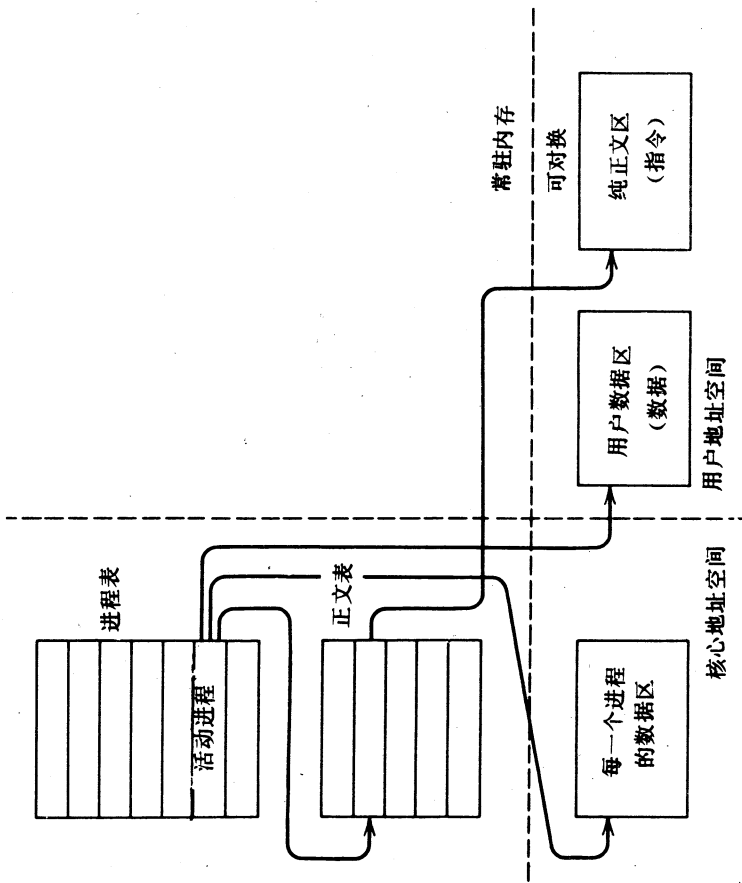


图19.1 b 管理具有纯正文段的进程的数据结构

的完成将导致该进程被唤醒并且被标志成准备重新恢复执行。唤醒一个睡眠进程所必需的信息就包含在proc表中。

核心为每一个活动进程分配一个user表。user表中包含了进程执行期间必须访问的信息。在进程挂起时，它的user表就不能访问或修改了。user表是进程数据区的一部分，并且当进程换出到磁盘上去时，user表要和进程映象的其它部分一起换出去。

user表中的大部分是有关进程的当前信息。例如，user表包含了如下内容：决定文件存取特权的用户和用户组标识号，指向所有进程打开文件的系统文件表（参见19.6节）的指针，一个指向索引节点表中当前目录（参见19.6节）的索引节点指针，以及一张各种信号的响应表。

进程的当前信息的操作是非常简单的。例如，如果进程执行改变目录的系统调用，则指向当前目录索引节点的指针值要改变。如果一个进程要忽略一个确定的信号，则在信号响应表中的相应项就置为0。在user表中大多数信息的管理很简单，你可以让一般程序而不必由核心来进行管理。这些简单管理由核心来完成，部分原因是为了一致性，但是更重要的理由是为了系统的完整性。在一个带有存储保护的计算机上，user表是不能由进程来访问的（除非通过系统调用），即使它是进程映象的一部分也是如此。

某些很通用的程序，比如shell和getty，通常是由若干个用户同时执行的。每个进程各自都必须有其进程映象的可变部分的副本，而固定部分，即程序正文，可以共享。为了共享，一个程序必须用一个特别的任选项来编译，这个任选项安排这个进程映象让可变部分和正文部分可以清楚地分离开。共享程序正文使UNIX系统可以更有效地使用计算机的主存。为了保持一个程序正文段的踪迹，UNIX系统维持了一张text表（正文表）。当一个程序使用共享的正文时，proc表项包含了一个指向text表的指针，这个text表才真正地指向该进程正文的位置。

让我们通过考察两个基本的系统调用来结束有关进程的讨论。

这两个系统调用由进程执行，目的是为了创建新的进程。fork系统调用被一个进程用来创建它自己的一个副本。在UNIX系统中增加进程数量的唯一途径是通过fork。fork之后就存在父进程和子进程两个进程。这两个进程之间主要的区别是两个进程有各自的进程标识号和各自的父进程标识号。这两个进程共享打开的文件，每一个进程有能力决定究竟哪一个是父进程，哪一个是子进程。

创建进程的第二个主要系统调用是exec。exec系统调用用于把调用者进程变换成一个新进程。在系统中，exec不能改变进程总数，只有调用者进程的特性被改变了。在系统调用exec之后，该进程的标识号（id号）未改变，并且打开的文件照样打开着。系统调用exec完成其它操作系统中的进程拉链功能，那些操作系统允许进程选择其后继者。

父进程通过系统调用fork，并后随exec来创建一个具有新的身份的子进程。在每次shell为你运行一个程序时，shell就使用这种调用序列。wait系统调用通常用来使fork和exec发生关联。wait系统调用允许父进程等待子进程死亡。当你在前台执行一个程序时，shell使用了wait。shell生成子进程，子进程执行所要求的处理，而父进程等待子进程的死亡。当子进程死亡时，父进程提示你打入下一个命令。shell在后台运行进程时，只要省掉对于子进程死亡的等待就行。

19.5 引导，进程 0 和进程 1

至此，我们已经对UNIX系统核心的某些基本概念作了介绍，现在，可以把注意力转向当核心一开始启动时，会发生些什么这样的问题上来了。若干特定的动作必须在一系列工作的早期就完成，这些动作是为了达到前面讨论的建立状态条件而执行的。等到进程 1 被初始化并且被启动后，系统就根据系统调用建立的规则正常运行和工作了。

把核心系统映象装入内存并且让它开始执行，这种动作称引导。只要给计算机硬件一加上电源，系统第一次启动，引导就发生。在UNIX系统瘫痪或故意停止之后，也要进行引导。

引导分几个阶段。第一个阶段，计算机硬件引导盘驱动器中的第一个磁盘块装入内存并且让其执行。在下一节你会发现，每一个文件系统的第一块是留作专用的，通常就是一个短小的自举装入程序。因此，为了引导，你必须让一个可引导的磁盘（即，在第一块中有一个有效的装入程序的磁盘）安置在用作引导的盘驱动器上。

这个短小的装入程序的目的是为了寻找在根目录中的名叫‘unix’的文件（‘/unix’），并且把‘unix’装到内存中。文件‘/unix’包含了操作系统核心的机器指令。‘/unix’是通过操作系统源代码文件的编译和连接而创建的。一旦文件‘/unix’被装入到内存，并且开始执行，就开始了引导过程的第一阶段。

核心首先要做的事情是初始化少量硬件接口。在带有存储管理硬件的机器上，需要初始化存储管理，在所有系统中，还要初始化提供定期中断的时钟。核心还要初始化少量数据结构，这些数据结构包括块设备缓冲池，字符设备缓冲池，索引节点缓冲池以及表明主存容量的变量。

在这些一般性的初始化工作之后，核心开始初始化进程0。一般，进程按规定是由fork系统调用创建的，fork指挥系统为调用者进程构成一个副本。但是，这种方法对于创建第一个进程，即进程0，是行不通的。为此，核心通过分配一个进程数据区结构，并且安置一个指向proc表中第一个空缺的数据结构的指针来创建进程0。进程0是较独特的，这有几个理由。首先要注意到，进程0没有代码段，它整个就是一个进程数据区结构。而所有其它进程都包含为了实现某些功能而要执行的代码。这些代码是通过编译得到的，并且是要顺序执行的一个程序的映象。但进程0

仅仅是由核心使用的一个进程数据区结构，而不是一个映象。还要注意，进程 0 的创建很特别，并且它还延续系统的生命。最后要注意，进程 0 是个地地道道的系统进程，它仅仅当处理机在核心态时才单独活动。你必须牢牢记住，进程 0 实际上只是一个核心数据结构，而不是通常意义下的一个进程。

在进程 0 创建并且被初始化后，系统就通过构造进程 0 的副本来创建进程 1。进程 0 副本的构造过程本质上与用户程序执行 fork 系统调用所遵循的过程相同。虽然进程 1 是精心构成的，但是这种精心制作过程与普通进程创建模式很相似。

最初，进程 1 的的确确就是进程 0 的复制品，它没有代码区。进程 1 创建后发生的第一个事件是它的长度要扩充。进程 1 的长度通过执行某段程序来增长，这段程序与系统调用 break（增加存储单元）所要执行的代码相同。进程 1 再次反常地动作，但是此动作仅是模仿后面跟着的一个普通程序执行系统调用的过程。要注意，直到此时，进程 0 和进程 1 都还得执行。

创建一个有生存力的进程 1 的第三个事件是，把一个非常简单的程序复制到你新建立的代码区中。复制到进程 1 代码区中的程序主要是包含了完成系统调用 exec 的机器指令，这组指令是为了执行一个称为 '/etc/init' 的程序的。

到此，进程 0 和进程 1 的初始化工作基本上完成了。进程 0 是一个进程数据区结构，在调度以及进程管理操作时由核心来使用。进程 1 是一个有生存力的映象，它是由核心特地创建的。一旦 UNIX 系统已经初始化了一个必要的数据库（进程 0）以及有生存力的第一个进程（进程 1），它就继续执行通常的调度子程序。

此时，核心的初始化完成了。然而，系统的初始化刚开始。我们通过描述系统生存周期中开始的几个事件来结束本节。调度程序的责任是决定什么进程去运行，哪些进程要换出或换进。第一次调用调度程序时，作出决定很容易，因为这时没有任何进程

需要换进或换出，只有一个进程渴望运行，这个进程就是进程 1。执行进程 1 立即引出 exec 系统调用，它用文件 ‘etc/init’ 中的代码覆盖进程 1 中原先的代码。此刻，进程 1 已经获得了它最终的形式，我们可以用它通常的名字，即 init 进程来称呼它。

init 进程的责任是建立 UNIX 系统的进程结构。init 通常能够创建至少两种不同的进程结构：单用户方式和多用户方式（在某些系统中，init 能够创建多于两个不同的进程环境）。通常，init 一开始就把 shell 连接到系统控制台，并且给 shell 以超级用户特权，这通常称为单用户方式。在单用户方式中，控制台以 root 特权自动地注册，其它通信线路都还不能接受注册。单用户方式通常用以检查和修补文件系统、执行基本的系统检测功能以及其它需要单独使用计算机的活动。

在单用户方式进程结构的后面，init 进程建立多用户进程结构。init 通过为每一条活动的通信线路创建一个 getty 进程来做到这一点（getty 在下一段讨论）。init 也创建一个 shell 进程来执行在文件 ‘/etc/rc’ 中的命令。‘etc/rc’ 文件包含了一段 shell 程序，其中包括安装文件系统、启动若干系统进程、删除过时的临时文件、以及启动记帐程序等命令。在 ‘/etc/rc’ 中的命令随 UNIX 系统具体配置不同而有很大不同。

在系统生存期间，init 进程睡眠，以等待它的所有子进程死亡。如果有一个 init 进程的子进程死亡了，那么 init 就醒来并且为相应的通信线路创建另一个 getty 程序。因此，init 不仅创建了多用户进程结构，而且还维持了系统生存期间的结构（见图 19.2）。

系统初始化进程中的最后一个活动者是 getty 程序。每一个 getty 程序耐心地等待某人在一个特定的通信线路上注册。当某人开始注册时，getty 程序就先完成对于线路协议的一些基本核实，然后执行 (exec) 注册程序来实际地检查口令。如果打入了正确的口令，那么注册程序执行 (exec) shell 程序以便实际从用

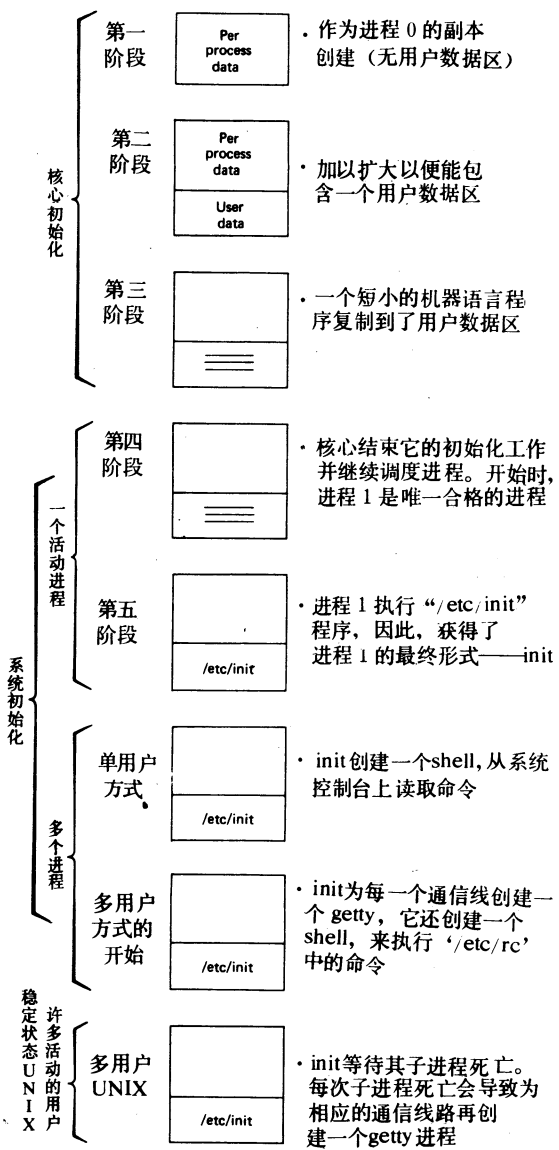


图19.2 进程 1 的生命转折点

户那里接受命令。当shell程序退出（死亡）时，init程序（它仅相对地活着）就醒来，而fork/exec就产生并且执行一个新的getty程序来监督通信线路并且等待下一个注册（见图19.3）。

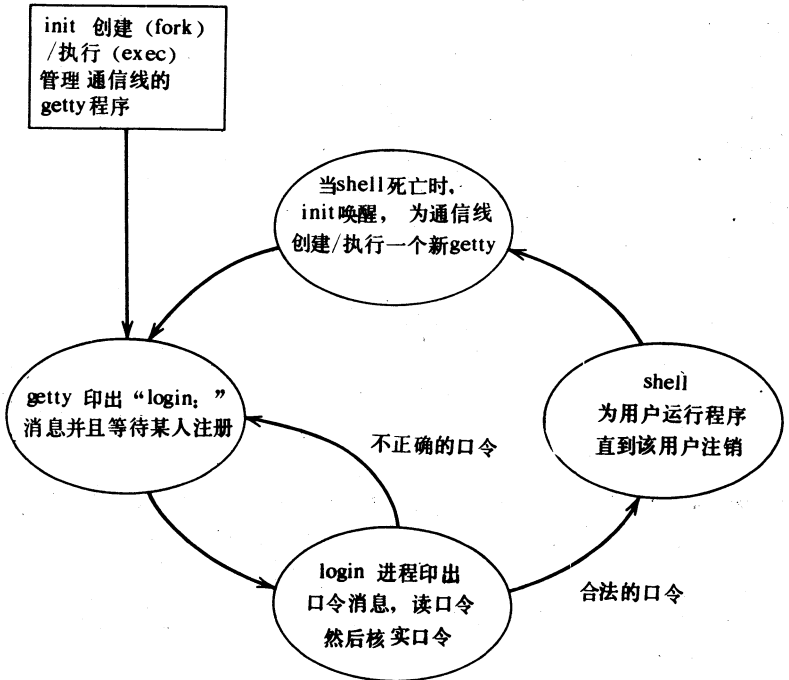


图19.3 每条通信线上的事件循环

19.6 文件系统

层次结构的文件系统是UNIX系统最重要的特性之一。任何文件系统最基本的划分办法是把磁盘和磁带存储器分成命名的单位，这单位我们称为文件。在许多系统中，都包含有若干种类的文件，各类文件都具有不同的存取方法。在UNIX系统中，所有

文件仅仅是一个简单的字节序列。有时，文件作为正文文件或二进制文件引用，但是这两种文件之间的区别仅仅是文件的内容（正文文件只包含ASCII值）而不是文件的结构或存取方法。

从系统的用户来看，目录是一组文件。在某些非UNIX的操作系统中，一个存储卷里的全部文件只属于一个目录。而另外一些文件系统，针对当前磁盘能存储很多文件的事实，将磁盘划分成一定数量的目录，再将文件分配至各个目录之下。上述两种方法所建立的是一种单层的文件系统，即全部文件都在同一层次上。单层的文件系统虽然可以使用，但却是凌乱的，因为每个目录包含很多种类文件。

UNIX文件系统是层次结构的文件系统。文件不是存放在一层上，而是存放在多层上，而且文件系统支持系统内部“空间”的设想。在普通的文件系统中，系统的主要组织结构是目录，目录包含了有关文件的所有信息，这些信息包括文件的名称、长度、位置、访问日期、方式和类型。因为在普通文件系统中的目录里包含了有关文件的所有重要的信息，所以这些信息由操作系统隐藏和保护。然而，在UNIX系统中，目录即是可由任何程序读取的文件。虽然目录是UNIX文件系统的可见结构，但是它们不是有关文件的全部信息的存储所在。在UNIX系统中，目录只包含每个文件的两类信息，即文件名和一个号，核心使用这些信息访问隐含的文件系统部分。

UNIX文件系统的隐含部分称索引节点（inode）。索引节点是UNIX文件系统的动作真正所在之处。每一个文件有一个索引节点。索引节点包含有关文件的位置、文件的长度、文件的存取方式、相应的日期、文件的所有者等等信息。随机的UNIX系统用户要与索引节点很好地隔离起来，至少要到索引节点结构变得不协调而且需要修补时为止。

让我们讨论一下UNIX文件系统开始部分的内部结构（见图19.4）。这里，我们将集中讨论磁盘存储器上的文件系统结构。本

节的后面，我们将集中讨论核心在内存中的结构。每个文件系统的第一块是引导块。对于引导过程涉及到的文件系统，其第一块包含了一个简短的引导程序。一般来说，这个引导程序读进一个更长的引导程序，或者读进的就是UNIX系统核心本身。引导的实际细节是与系统密切相关的。对于引导过程涉及不到的文件系统，第一块通常不用。

文件系统的第二块是文件系统头。这个头〔也称为专用块（superblock）〕包含了有关文件系统的种种信息。尤其是，专用块包含了文件系统的大小、在文件系统索引节点号、以及若干牵涉到空闲链表的参数。当文件系统用mount命令安装上时，要在UNIX系统核心的安装（mount）表中构造一个登记项，该文件系统的专用块被读进核心的一个大的缓冲区中。所有安装上的文件系统专用块都可由核心来存取，核心为了存取文件系统中的文件和索引节点需要用到专用块中的信息。

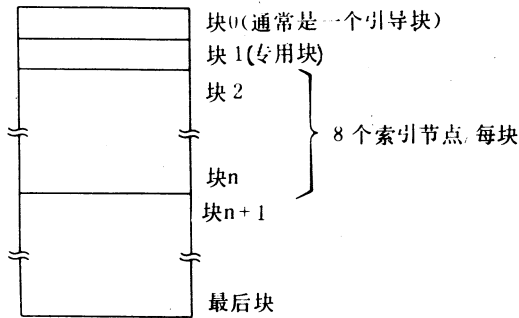


图19.4 文件系统安排

从块2开始存放文件系统的索引节点。大小不同的文件系统有不同个数的索引节点，索引节点所占大小的精确数目存放在专用块中。由于索引节点的大小是固定的，并且从0开始顺序编号，因此，指定索引节点号，就可以确定一个索引节点。

每一个文件都由一个索引节点定义，这个索引节点包含了系统所保持的有关这一个文件的所有信息。索引节点包含了如下信息：文件存取的方式和文件的种类，按字节计数的文件长度，文件所有者和同组用户的标识号，文件的位置以及文件创建、最近修改、和最近访问的时间。要注意，索引节点不包含文件的名字，名字是存放在目录中的。

UNIX系统通过一张文件块表来确定文件的位置。另一些操作系统的文件在磁盘上是连续存放的，它们通过起始块号和最后块号来确定文件的位置。由于文件长度的增长要受到下一个文件起始位置的限制，因此这种连续存放的文件系统并不令人满意。而且这种文件系统在大文件之间有小空隙，在无用单元收集程序收集之前，这些空间是要浪费的。UNIX系统通过保持一张文件块的表，避免了连续存放的文件系统所存在的问题。UNIX文件块可以分布在整个磁盘上，这些块组成了逻辑链，以保存文件的位置信息。

文件位置的关键信息是存放在索引节点中的由13个块号项组成的（块指针）表。这张表的前10个块号指明文件的前10块（每块512字节）。如果文件只有4块长，则此表的前4项为文件块号，后9项为0。如果文件长于10块（5120字节），则表的第11个块号用来指定一个磁盘块，此磁盘块中包含文件的下面128个块的块号。这个块称为间接块。对于长于138（128+10）块（70656字节）的文件，表中的第12项是一个磁盘块的地址，这个块称为二次间接块。该磁盘块中包含了128个间接磁盘块的地址。对于大于16522（10+128+128²）块（8459264字节）的文件，在这张表中的第13项包含了一个三次间接块的地址。在UNIX系统中，允许文件的最多块数是 $10 + 128 + 128^2 + 128^3$ ，即2113674块，或者说是1082201088字节（见图19.5）。

从非常大的文件中检索信息要比从小型文件中检索信息困难，因为为了决定这个文件的真正的块地址需要检索间接块。为了适

应容纳非常大的文件的能力，希望花费小的检索开销。例如，为了整个读10000块的文件，系统得检索1个二次间接块和79个间接块及10000文件块。还应当注意到，现今可用的大多数磁盘的容量要比UNIX系统所能确定的最大的单个文件的容量（10亿字节）小得多。

现在，让我们把注意力转移到目录上来。目录是一个驻留在盘上的文件，它包含一组文件名和相应的索引节点号。为了保持文件系统层次的完整性，不允许程序写进目录，读目录是允许的。当程序发出创建或删除文件的请求时，系统操纵目录的内容。正如任何文件都是由一个索引节点定义一样，每个目录也是由一个索引节点定义的。目录有16字节长，文件名占14个字节，索引节点编号占2个字节。

每一个目录的前两项是用于‘.’和‘..’的。‘.’项列出该目录本身的索引节点，而‘..’项列出其父目录的索引节点。（注意：在根目录中，‘.’和‘..’都指向根目录，因为根目录没有父目录。）当目录由系统建立时，这两个标准登记项就自动地放在一个目录中，并且不能由用户删除。当一个目录只包含‘.’和‘..’文件时，就看成是“空”的。

在UNIX系统核心看来，通过文件系统的一条路径实际上是目录和索引节点之间途经的路线。考虑路径‘.. / a / b’，此路径从当前目录开始，到达当前目录的父目录，再到达这个父目录的子目录‘a’，最后到达在目录‘a’中的名为‘b’的文件。为了沿此路径前进，系统要完成下列步骤：

1. 检索当前目录的索引节点（当前目录的索引节点指针在user结构中）。
2. 为了寻找和检索当前目录以便获得名字‘..’，就要使用当前目录的索引节点中的信息，并且返回‘..’的索引节点号。
3. 检索‘..’的索引节点。

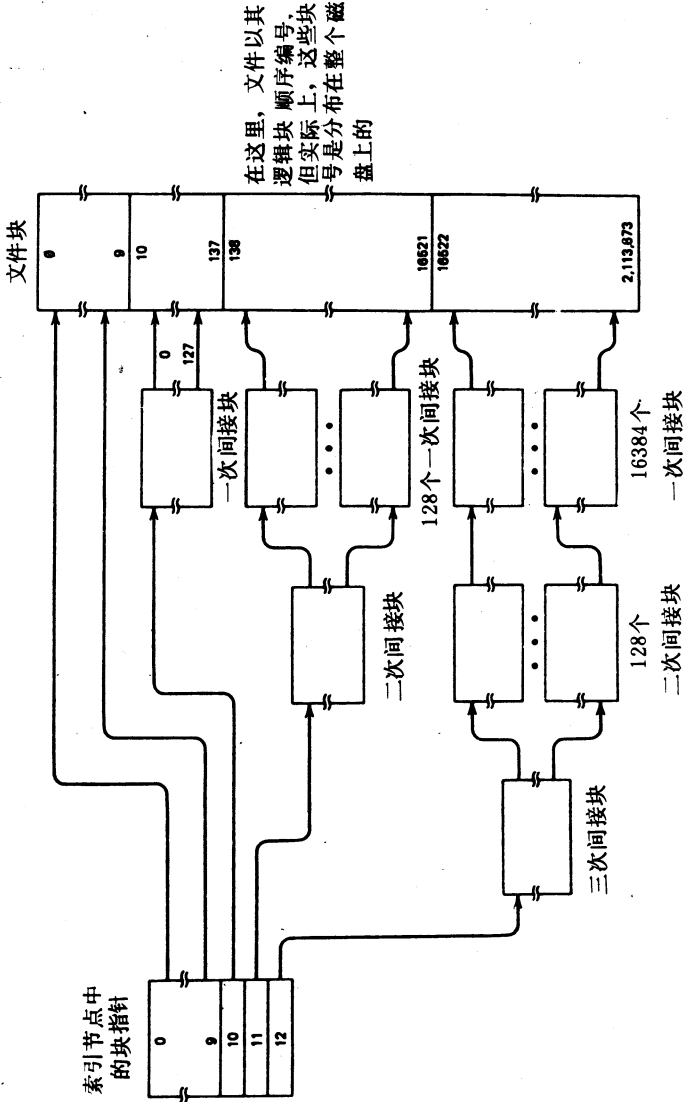


图 19.5 索引节点中的块指针实际是如何指向一个文件块的

4. 为了检索和寻找父目录以便获得文件 'a', 就要使用在 '..' 索引节点中的信息, 并且返回 'a' 的索引节点号。

5. 检索 'a' 的索引节点。

6. 为了寻找和检索 'a' 的目录以便获取文件 'b', 就要用 'a' 索引节点中的信息, 并且返回 'b' 的索引节点号。

7. 检索 'b' 的索引节点。

8. 存取文件 'b'。

这些就是检索文件要做的大量工作 (参见图 19.6)。在普通的文件系统中, 检索一个文件要容易得多。UNIX 系统核心所做的这些额外工作是我们采用层次结构文件系统所付出的代价。跟踪路径名是罕见的, 而存取已经定位的文件却要常见得多。

至此, 在文件系统的有关讨论中, 我们已经描述了为创建文件系统结构而存放在一个磁盘上的结构。这些结构包括专用块、索引节点、目录文件、普通文件及特别文件。这些结构可以由核心在通常的操作过程中处理, 或者由 fsck 和 fsdb 那样的程序来处理。在 fsck 和 fsdb 处理过程中, 文件系统要作修补。现在让我们通过查看一些结构来结束我们有关文件系统的讨论, 这些结构是为了存取文件系统而由核心保存在内存中的 (参见图 19.7)。

我们已经提到了由核心保持在内存中的两个结构: 每个安装上的文件系统的专用块以及一张索引节点表。专用块保持在内存中的原因是因为它包含了文件系统的若干关键性的参数, 其中最为重要的参数是空闲块的单元地址。在内存索引节点表中的每一项都包含了存取一个文件所用的关键性信息, 这些信息包括文件的存取方式及文件中块的地址。

还有一个为了存取文件而由系统核心保持在内存中的表, 叫文件表。文件表中的每一项都有一个指向索引节点表中的一个特定项的指针, 及一个文件的读/写指针。每个带有打开文件的进程的数据区都包含有指针, 这个指针指向文件表, 文件表的指针再指向索引节点表, 而索引节点表的指针才真正指向文件。

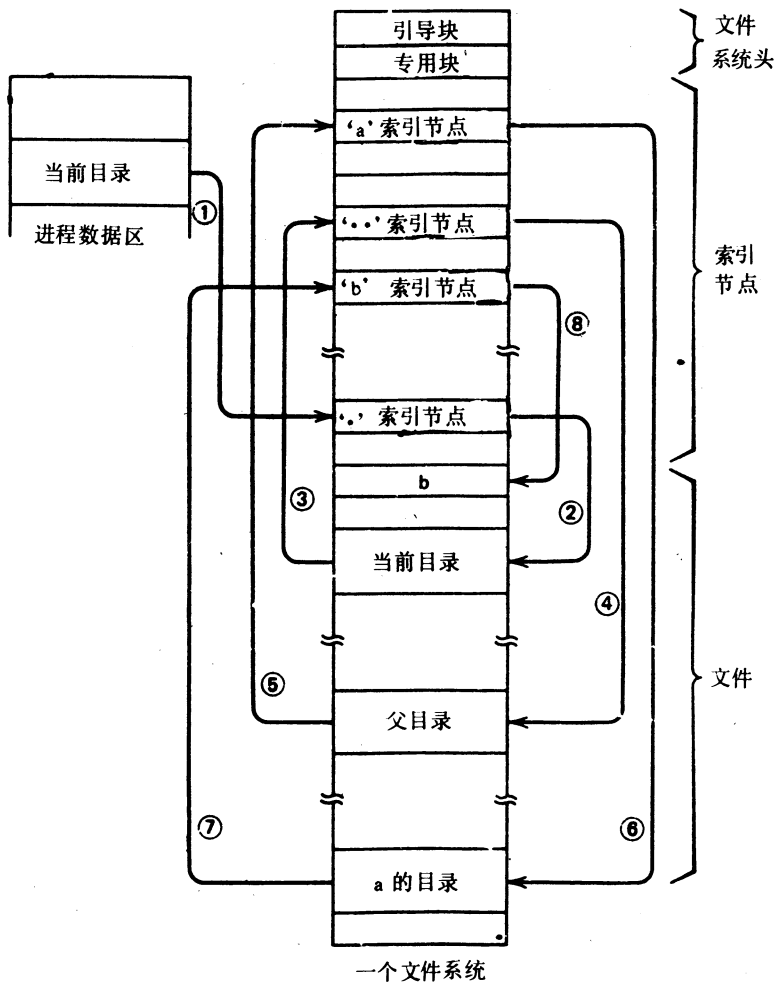


图 19.6 跟踪一个路径名 (跟踪路径名
 ‘.. / a / b’ 的八个步骤)

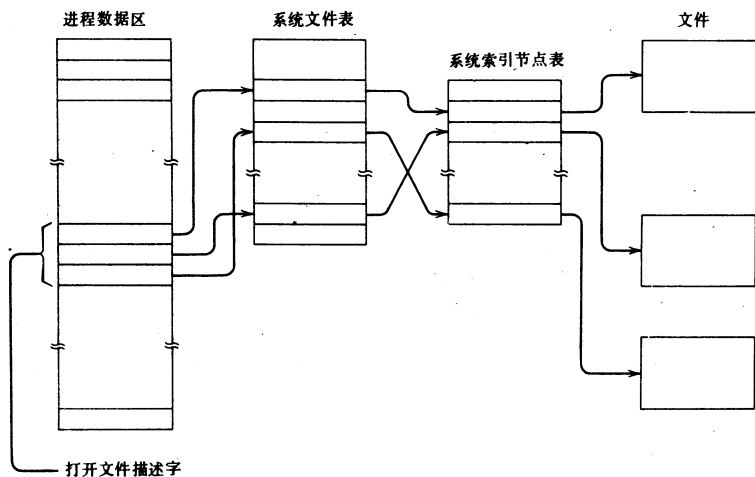


图19.7 存取文件时的核心数据结构

这看起来是相当复杂的，但你可以认为，每个进程数据区包含了直接指向索引节点的指针。设置文件表的真正目的是为了保存文件的读/写指针。当一个带有打开文件的进程用fork创建子进程时，这个进程和创建的子进程共享指向这个打开文件的一个读/写指针（存放在文件表中），这种特性经常用在shell中。只要shell运行一个程序，它就创建（fork）子进程，并执行（exec）新的进程，然后等待（wait）那个新进程的终止。在wait期间，那个新进程从标准输入中读入，并从标准输出中输出。shell和其子进程共享指向标准输入和输出的读/写指针（文件的读/写指针），这样，在shell再次获得控制时，能正确地定位读/写指针。

19.7 外部设备

外部设备通常是连接到计算机上的用于信息输入或输出（或两者皆有）的一种部件。磁盘、磁带、通信线、卡片读入机和打印机等是典型的外部设备。UNIX 系统包括管理 I/O 的系统。块方式通常用于可以由 512 字节组成的块来编址的设备，磁盘和磁带通常用块方式。

块方式的着眼点在于允许核心使用缓冲技术，以便减少 I/O 传输量。核心维持一组块缓冲区。只要程序需要 I/O 传输，就检索内部缓冲区以便查看一下该块是否已经在内存中。如果所需要的块不在内部缓冲区中，那么系统将释放一个内部缓冲区，并且在该内部缓冲区和 I/O 设备之间传送所要求的块。系统力图把频繁使用的块保留在内存里，从而可以减少 I/O 传输量。

字符方式用于所有不适合于块方式的设备。通常，字符方式用于通信线、打印机、纸带机、卡片读入机、等等。有块结构接口的大多数设备也有一个字符接口，以便不用核心的缓冲机制就可存取该设备。对于字符设备的存取，不在系统的块缓冲区池中进行缓冲。对于一次传送一个字符的字符设备的存取来说，通常由核心在字符缓冲区链上进行缓冲，而对于传送大块（通常是成块的）数据的字符设备的存取来说，核心一般不作任何缓冲处理。

在开发操作系统的 I/O 部分的软件时，有两个主要的困难。第一个困难是，每一类外部设备都有稍微不同的管理技术，所有这些技术，都要求把程序编制好放到操作系统中去。第二个困难是大多数计算机的外部设备经常要重新配置，每次添加或去除外部设备时，都需要修改操作系统。在 UNIX 系统中，通过使用不同的软件模块来控制各种不同的外部设备，并且使用一组表，从逻辑上把核心与不同的外设驱动程序相连接来解决上述两个问题。本节后面将更详细地讨论这些技术。

在操作系统内部工作、并控制计算机和特定外设之间的数据传送的一组子程序，称为驱动程序。在发行UNIX系统时，常带有十来个很通用的外设的驱动程序。在UNIX系统网络内部，还有许多不太常用的外设驱动程序。

为获得连接到操作系统中的正确的驱动程序，是系统配置时的主要工作目的。操作系统利用许多不同的途径解决重新配置问题。UNIX系统核心通过修改若干关键性的程序模块，然后重新编译来重新配置。UNIX系统使用bdevsw和cdevsw两张表来控制I/O配置过程。这些表通常被保留在名叫‘conf.c’的C语言源程序文件中。

在UNIX的最新版本中，有一个称为config的程序，它会自动地为任何指定的硬件配置创建一个‘conf.c’文件。在UNIX系统的早期版本中，文件‘conf.c’必须手工修改。除了bdevsw和cdevsw以外，‘conf.c’通常还包括若干参数，这些参数控制各种各样的资源，比如核心内部缓冲区的个数、对换空间的大小以及某些核心内部表格的长度，等等。

‘conf.c’文件的中心是一对结构：bdevsw和cdevsw。这两个结构通过安置不同的驱动程序模块，使它成为UNIX系统核心有能力很容易地适用于不同的硬件配置的关键。

先谈一下cdevsw。cdevsw是面向字符的I/O设备的驱动程序模块和UNIX系统核心的连接表。此表中的每一项用来在逻辑上把系统与驱动程序连接，该驱动程序和一个特定的主设备号相关。在cdevsw表中的第0项对应于具有主设备号0的I/O设备，下一项对应于具有设备号1的I/O设备，等等。当你用mknod命令创建特别设备文件时，必须检查这张表。

在cdevsw表中的每一项定义了驱动程序地址，这些程序是针对一个特定设备的打开、关闭、读、写和控制传送方式的。open和close子程序完成数据传送前后所要求的特殊处理。例如，在电话通信线上的open要等待线路上响铃，然后回答；在同一通

信线上的 close 子程序可能要挂起这条通信线。read和 write 子程序由核心内部调用，以便把数据从设备上取来或把数据送到设备上去。read和 write 子程序通常用来和中断服务子程序连接，这个中断服务子程序真正地控制数据的传送。传送方式子程序用在通信线路设备上，为的是使通道适合特定的终端和通信协议，这种传送方式子程序不用于字符设备，因为这种方式不能把字符发送到计算机终端上去。

UNIX 系统驱动程序设计成使得一个驱动程序可以为若干个相关的硬件服务。例如，在 PDP-11 计算机上，称为 KL11 的接口用于把字符送到一个终端上，即使该计算机包含若干个 KL11 硬件接口，也只需要一个 KL11 驱动程序。为了区分不同的接口，要传递给驱动程序一个号，这个号称为次设备号，它指定使用的接口，次设备号的解释留待各个设备模块自行处理。在 KL11 那样的驱动程序中，次设备号指定是哪一个 KL11 接口。UNIX 系统的其它驱动程序把次设备号用于其它目的。例如，磁带驱动程序，它使用不同的次设备号来指定记录密度，或者用来指定当磁带机关闭时，究竟要不要重绕。

驱动程序的名字通常带有两个字符的标准前缀，这个前缀暗示相关的硬件接口。例如，对于 DEC 的 DL11 串行接口，驱动程序使用前缀“kl”（因为这种接口以前称为 KL11）。在 cdevsw 表中，定义核心与 kl 驱动程序接口的那一行是

```
/* 0 */ &klopen,&kfclose,&kfread, &kfwrite,&kfsgtty,  
(&是 C 程序设计语言中一个对象的地址记号。)
```

驱动程序常常省略某些子程序，这或者是因为相应的设备不需要这种操作，或者是因为不允许这种操作。例如，不需要打开或关闭 UNIX 系统内存设备的特别操作，很明显不需要控制内存设备的传送特性，因为它不是一个通信线路的数据传送。当子程序不在驱动程序中出现时，在 cdevsw 表中就引用特殊的 nulldev 或 nodev 子程序。当不需要某个操作时，用 nulldev 子程序，而

当某个操作在逻辑上是不可能时，用 `nodev` 子程序，并且在引用时会出现一个错误。在 `cdevsw` 表中，与实际内存的接口项是

```
/* 8 */ &.nulldev, &.nulldev, &.mmread, &.mmwrite, &.nodev,
```

这一项指出，当打开或关闭内存设备时，不要执行操作，如果想去控制内存设备的传送方式，则是一种错误。

用 ‘`conf.c`’ 文件中的 `bdevsw` 表，把块设备的 I/O 子程序与 UNIX 系统核心连接起来。在此表中，包含打开和关闭子程序的地址、策略子程序的地址以及设备表的地址。当设备第一次被打开时，这种设备通常还是未用过的，因此，打开和关闭子程序必须进行一些处理。策略子程序被调用来完成块的读和写。调用策略子程序的理由是，通过安排块检索的次序可以在磁盘和磁带上优化整个的存取时间。

设备表是块设备执行 I/O 的主要依据。设备表包含了指向该设备的缓冲区的指针。完成块 I/O 设备的存取分两个阶段：第一阶段，分配一个缓冲区，初始化该块的首部。这个首部包含了该设备上的块号、若干标志、指向其它首部的指针和一个指向实际缓冲区的指针。第二阶段，进行缓冲区和设备之间的数据传送。因为考虑到了上面提到的策略，所以在缓冲区和设备间传送的次序可能不是发出请求的次序。

接到连 UNIX 系统上的许多外部设备，使用中断技术来传送数据。硬件中断是一个电子信号，这种信号导致处理机停止它正在做的所有工作，转到一个中断服务子程序。中断服务子程序通常处理外部设备的即时要求，并把控制返回到被中断的程序。面向字符的设备经常每传送一个字符就中断一次，偶而每传送一行或每传送一个数据块中断一次；面向块的设备通常每传送一个数据块中断一次。

中断处理程序是设备驱动程序的一个部分，它在外部设备的数据传输过程中，对数据的进/出予以中断管理。当 I/O 接口硬件产生一次中断时，就激活这些程序。在 PDP-11 计算机系统中，

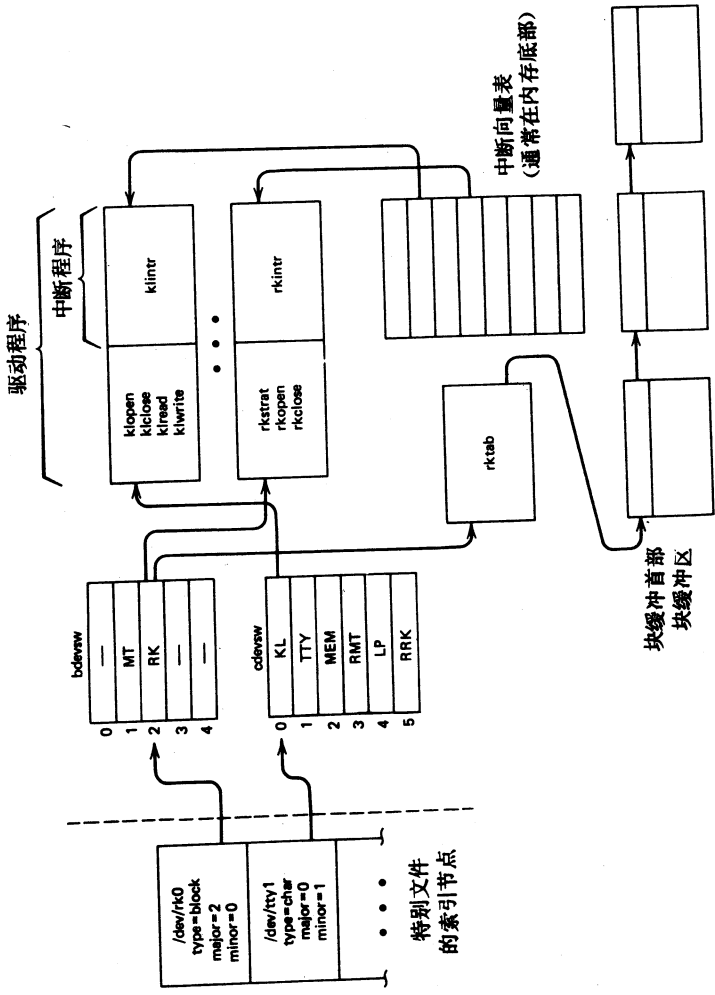


图19.8 访问 I/O 设备时的核心数据结构

中断服务程序的地址必须存放在内存低地址的特定单元中。这些地址单元可由接口线路插件上的可选件（某种连接线）来决定。怎样把特定的中断处理程序与特定类型I/O设备的中断联系起来，不同的计算机使用了不同的技术。由于PDP 11把地址放在内存低地址部分，因此在PDP 11计算机上的UNIX，使用称为‘low.s’的汇编语言文件，这个文件规定了中断服务程序的单元地址。

UNIX 系统简明手册

这份 UNIX 系统简明手册包含了以下40条最常用的实用程序：

at	file	mv	sort
cat	find	nice	spell
cd	grep	nohup	stty
chmod	kill	od	tail
chown, chgrp	ln	passwd	tee
cp	lpr	pr	time
crypt	ls	ps	tty
date	man	pwd	wc
diff	mail	rm	who
echo	mkdir	rmdir	write

为了增强UNIX系统的功能，许多UNIX系统命令都进行了局部修改，其细节可参见你身边的手册。当这里讲的用法和你身边手册里讲的用法不一致时，一般应以你身边的手册为准。有些命令存在几种版本。通常，其中一种版本是在贝尔系统中使用的，另外的是在贝尔系统之外使用的。这里描述了向外公布的各种版本的情况，并在条文的最后加了注解。

这个手册遵循原始的UNIX系统手册所建立的格式。在命令的格式中，方括号内是命令行的任选项（自变量），自变量后的省略号（...）表示该自变量可以重复。每个命令都给出了应用的例子。前面的例子解释了命令的通常用法，后面的例子展示了程序全部的能力。

at

名字

at——在将来某个指定时刻运行shell程序。

格式

at 时刻 [日期] [文件名]

说明

at程序用于在将来某个时刻执行 shell 程序。at构造了一个指定文件的副本（如没有指定文件，则是标准输入的副本），然后安排了当时刻到达时所要执行的文件。当 shell 程序执行时，它将运行在你的当前目录下。它的整个环境变量是一个应该执行at命令的时刻值。因此，命令实际执行时的环境，将和你执行at命令时的环境相同。

at命令的钟点自变量是一个1到4位的数字，其后面的尾随字母可选：a表示AM（上午），p表示PM（下午），n表示正午，m表示午夜。如果钟点自变量是一位或者两位数字，那么这个数目表示小时；如果有三位或四位数字，那么表示小时和分钟。如果数字后没有字母，那么表示24小时制的钟点。

任选的日子自变量取下面两种形式之一：月名后尾随一个日期，或者尾随一个星期几；星期几的后面可以尾随一个字“week”。在后一种情况下，命令将比提及的日期晚一周运行。月的名字和星期的名字均可以缩写。

举例

在星期二早上两点，执行shell程序‘nroffb.sh’：

```
at 2 tues nroffb.sh
```

在一月二日下午三点，执行shell程序‘cmds.sh’：

```
at 3p jan 2 cmds.sh
```

也可以用等价的命令：

```
at 15 jan 2 cmds.sh
```

在下一星期五的午夜执行shell程序 ‘cmds.sh’:

```
at 12m fri week cmds.sh
```

注

如果你欲减少系统的负荷，最好使用at程序，在半夜一两点钟运行大程序，而不要在高峰期间使用nice程序，以低优先权运行这种程序。

cat

名字

cat——串接並印出文件。

格式

```
cat [-u] 文件名...
```

说明

cat 程序用于两个场合：在你的终端上印出文件；使用输出重新定向符把几个文件组合（串接）成一个文件。任选项“-u”用来禁止cat实现正常块缓冲。

当你使用 cat 程序串接文件时，输出文件最好不要和某一输入文件同名，例如，命令“cat a b>a”及“cat a b>b”就不能达到期望的工作效果。

举例

在终端上印出文件 ‘ch3’:

```
cat ch3
```

在终端上印出几个文件:

```
cat ch1 ch2 ch3 ch4
```

组合几个文件到文件 ‘ch1-4’ 中:

```
cat ch1 ch2 ch3 ch4>ch1-4
```

创建一个名为 ‘file.new’ 的空文件:

```
cat /dev/null > file.new
```

cd

名字

cd——移至一个新的工作目录。

格式

```
cd [目录]
```

说明

cd命令用于从当前目录移至另一个工作目录。

举例

移至 '/usr/bin' 目录。

```
cd /usr/bin
```

移至当前目录的父目录：

```
cd ..
```

返回主目录：

```
cd
```

注

在较老的版本中，cd命令称为chdir。chdir命令并不记得你的主目录名，它要求你送入一个目录名自变量（包括主目录名）。

chmod

名字

chmod——改变文件的存取方式。

格式

```
chmod 方式 文件名...
```

说明

chmod命令用于改变文件及目录的存取方式。象第6.5节所讨论的那样，一个文件有三种存取方式（读、写及执行）和

三类用户（文件所有者、文件的同组用户、其他用户）。

方式用符号来表示，它通常由三部分组成：

谁 操作 许可

下表列出表示方式的各种符号及其解释：

	谁	操作	许可
u g o a	所有者（用户） 同组用户 其他用户 全体(ugo)(可缺省)		
r w x s t u g o		读 写 执行 调整用户（或小组）标识号方式 保存正文（粘着）方式 用户当前的许可 同组用户当前的许可 其他成员当前的许可	
- + =			删除许可 增加许可 赋予许可

只有文件所有者和超级用户可改变文件的方式。调整用户和小组标识号方式以及保留正文方式一般由系统管理员或系统程序员所使用，这些不在这里讨论。

方式也可以用一个八进制数来指定，以代替符号表示的方式，此种表示方式不在这里讨论，你可参阅UNIX系统手册。

举例

构造人人都能读的文件‘arli’：

```
chmod a + r arli
```

由于在符号表示方式中“谁”这部分的“a”可以缺省，因此用下面命令效果也是一样的：

```
chmod +r arli
```

对于文件‘britt’，使得同组成员和其他人获得和所有者当前同样的许可：

```
chmod go=u britt
```

使文件‘newsysdb’除所有者以外的任何人均不能读也不能写：

```
chmod go-rw newsysdb
```

使人人均能读写文件‘ptime’及‘qtime’，但均不能执行：

```
chmod a=rw ptime qtime
```

使文件‘chbeau.sh’人人皆能执行：

```
chmod +x chbeau.sh
```

使任何人皆不能在当前目录下创建文件：

```
chmod a-w.
```

chown chgrp

名字

chown——改变文件的所有者。

chgrp——改变文件的同组用户。

格式

```
chown 新所有者 文件名…
```

```
chgrp 新同组用户成员 文件名…
```

说明

命令 chown 及 chgrp 用来改变文件的所有者及同组用户。当你从另一个用户那里继承了文件时，或改变你的注册名字时，或将文件从一个系统转移到另一系统时，你需要做这一事情。

用户名通常可在文件‘/etc/passwd’中找到，而同组

用户名通常在文件 ‘/etc/group’ 中。

举例

将文件 ‘nycal’ 的所有者改成名为 “ralph” 的用户所有：

```
chown ralph nycal
```

将全部带有后缀 “.n” 的文件所有者改成名为 “george” 的用户所有：

```
chown george *.n
```

将以字母 “ch” 开头的全部文件的同组用户改成名为 “elec-
micro” 的同组用户：

```
chgrp elecmicro ch*
```

cp

名字

cp——复制文件。

格式

```
cp 文件1 文件2
```

```
cp 文件... 目录
```

说明

第一种格式是 cp 用 ‘文件 2’ 名字构造 ‘文件 1’ 的副本；‘文件 1’ 并不受复制操作的影响。如果 ‘文件 2’ 已经存在，则它的存取方式及所有权不予改变；否则它的方式和所有权由 ‘文件 1’ 复制过来。

第二种格式是用 cp 复制一个或几个文件到指定的 “目录” 下，同时保持原有的文件名。

举例

构造一个文件 ‘nuk.abm’ 的副本，具有文件名 ‘nk.2’：

```
cp nuk.abm nk.2
```

复制子目录 ‘disarm’ 下的全部文件到子目录 ‘newlit’ 下：

```
cp disarm/* newlit
```

复制在当前目录下全部带有后缀‘.doc’的文件到子目录‘disarm’下：

```
cp *.doc disarm
```

crypt

名字

crypt——给文件加密。

格式

```
crypt [口令]
```

说明

crypt 程序将给要保密的文件加密。在UNIX系统里，利用存取特权系统（参见chmod）来保护文件，将提供一定程度的保密性。然而，加了密的文件比具有存取保护的文件更安全得多。加密、解密机构是由一个口令来控制的。同一个口令用于加密一个文件或解密一个已经加了密的文件。如果在命令行中未提供口令，那么crypt程序将提示你提供口令，当你打入口令时，并不予回应。crypt从标准输入读进，并写到标准输出上。

举例

为文件‘salaryhist’加密，并把加了密后的版本放入文件‘salh.enc’中，然后删除原来的文件：

```
crypt abracadid<salaryhist>salh.enc; rm salaryhist
```

在终端上印出‘salh.enc’的原来形式（未加密前的版本）：

```
crypt abracadid<salh.enc
```

恢复‘salh.enc’的原有形式，并把它放在文件‘salaryhist’内：

```
crypt abracadid<salh.enc>salaryhist
```

在所有这些例子中，口令是由命令行来提供的。另一种写法，

口令可在命令行中省略，程序 crypt 将要求你提供口令。

date

名字

date——印出及设置钟点和日期。

格式

date [mmddhhmm [yy]]

说明

date 命令一般用来显示日期和钟点。超级用户也可使用任选自变量来置日期和钟点。自变量依次列出：月（01—12），日（01—31），小时（00—23），分（00—59），年份。年份的前面两位数字可要可不要。

举例

显示日期：

```
date
```

置日期为1980年1月21日上午9：15：

```
date 0121091580
```

注

具有超级用户特权者，才能设置日期和钟点。

diff

名字

diff——报告正文文件间的差别。

格式

diff [-efbh] 文件1 文件2

说明

diff 程序用来比较两个文件。diff 也能用于管道线，这时应该用特殊名字“-”来代替文件名字。如果两个文件不一样，那么diff就印出与编辑命令形式相似的一行，后面跟着两个

文件中受影响的正文行（去指出哪些行是不同的）。

diff印出下列三类伪编辑命令：

n1 a n3, n4

文件 1 中缺少文件 2 中存在的某些行。如果将文件 2 的第 n3 行至第 n4 行放在文件 1 的第 n1 行之后，那么两个文件就一致了。

n1, n2 d n3

文件 1 中有某些行是文件 2 中所缺少的。如果从文件 1 中删除第 n1 行至第 n2 行，则这种不一致将消失。类似地，也可通过将文件 1 的第 n1 行至第 n2 行加到文件 2 的第 n3 行之后来实现。

n1, n2, c n3, n4

文件 1 和文件 2 有一个区段不一致。如果文件 1 的第 n1 行至第 n2 行改为文件 2 的第 n3 行至第 n4 行（或反之），则这种不一致将消失。

对于这三类伪命令，如果 n1 等于 n2，或者 n3 等于 n4，则只印出一个号码。跟在伪命令后将印出文件里受影响的行。文件 1 的这类行是通过把“<”放在每行开头来标记的，而文件 2 的这类行则用“>”来标记。

diff识别四个任选项：

- e 将产生一个编辑手稿，它可以由标准 UNIX 系统编辑程序用来重新由文件 1 建立文件 2。
- f 产生一个同样的手稿，但它是相反的次序。用 -f 任选项产生的手稿不能被 UNIX 系统标准编辑程序所使用。
- h 快速地实现比较，而且文件大小不受限制。使用 -h 任选项的 diff 在重新同步时并不是很仔细的，-e 及 -f 的任选项不能与 -h 任选项共用。
- b 将忽略由于空格（空白及制表符）而引起的差别。

举例

比较文件 ‘notes.a’ 和文件 ‘notes.old’:

```
diff notes.a notes.old
```

印出一个编辑手稿, 由 ‘notes.a’ 重新建立 ‘notes.old’:

```
diff -e notes.a notes.old
```

在以前的某一时刻, 名为 ‘lsfile’ 的文件是通过执行 “ls > lsfile” 命令而建立的。为了判别目录中是否仍有建立文件 ‘lsfile’ 时的同样内容, 我们要将 ‘lsfile’ 和工作目录的当前内容进行比较:

```
ls | diff - lsfile
```

echo

名字

echo —— 复制命令行的自变量。

格式

```
echo [自变量]
```

说明

echo 命令复制它的命令行自变量到标准输出。echo 用于下列目的: 印出 shell 命令文件中的消息, 把少量已知的数据插到一个管道或文件中, 显示 shell 变量的值, 以及找出 shell 将和命令行自变量一起做些什么。

举例

在终端上印出消息 “Hello”:

```
echo Hello
```

将消息 “processing complete” 放在文件 ‘pmessage’ 中:

```
echo processing complete > pmessage
```

显示 shell 变量 \$PATH 的值:

```
echo $PATH
```

问题：当你打入命令“`expr \ (5 + 7 \) * 3`”时，希望见到结果“36”，但你却遇到一个难懂的出错消息。

解决方法：使用 `echo` 命令去看一下 `expr` 命令实际收到了什么样的自变量：

```
echo \ ( 5 + 7 \ ) * 3
```

输出将揭示出问题所在——星号被扩展为当前目录下的一组文件，这是由于 shell 的文件名生成过程所引起的。为了使星号传递给 `expr` 程序，你必须将它转义。

file

名字

`file`——猜出命名文件的类型。

格式

`file` 文件名...

说明

`file` 命令试图决定命名文件的文件类型。对于目录文件和特别文件，`file` 命令是比较准确的，而对于其它类型文件，`file` 命令采用经过训练的猜测方法。对于包含 ASCII 码的正文文件，`file` 命令试图决定它为源程序，这个结果常常是准确的。对于包含二进制信息的普通文件，`file` 命令试图决定文件是否是目标文件、库、`cpio` 映像文件，或者任何其它文件。不符合这些类型的文件通常归为“数据”一类。

举例

决定当前目录下全部文件的文件类型：

```
file *
```

决定目录 ‘`/usr/bin`’ 下全部文件的文件类型：

```
file /usr/bin/*
```

决定名为 ‘`unk`’ 文件的文件类型：

file unk

find

名字

find——在子树中查找文件。

格式

find 路径名... 条件...

说明

find 命令在以命令行“路径名”所指定的文件系统子树中查找满足指定“条件”的文件。至少要规定一个路径名（常常是‘.’）及一个条件。条件由下面提及的一个条件或多个条件所规定：

-atime n

规定在 n 天内已经被存取的文件。

-exec cmd

规定被执行的命令。这个命令的末尾用一个转义的分号来指示。在命令内，自变量“{ }”由当前路径名所替换。

-group 小组名

规定文件的同组用户。

-links n

规定文件别名（联结）的个数。

-mtime n

规定在 n 天内已经被修改的文件。

-name 文件名

规定用一般 shell 元字符的文件名。为使元字符能传递给 find 命令，元字符必须予以转义。

-newer 文件名

规定被考察的文件必须比指定的“文件”更新。

`-ok cmd`

除了命令被印出（前面用一个问号）并且对于给定的操作符希望得到回答 `yes` 或 `no` 以外，与 `-exec` 相同。

`-print`

印出当前路径名。

`-size n`

规定文件的大小为 `n` 个块。

`-type 字符`

用字符规定文件的类型：`f` 为普通文件，`d` 为目录，`c` 为字符特别文件，`b` 为块特别文件，`p` 为一个管道文件。

`-user 用户名`

规定文件的所有者。

可用括号把条件组成一个组，也可用一个惊叹号作为“`not`”操作符来取反条件。

在要求数值自变量的条件（`-links`，`-size`，`-atime` 及 `-mtime`）中，数前面带有“`-`”表示小于 `n`，数前面带有“`+`”表示大于 `n`，没有前缀则就表示 `n`。除非使用了 `-o` 这个“逻辑或”操作符，否则连在一起的条件被认为代表“逻辑与”条件。

使用 `find` 去组合一组条件象写一个程序一样复杂，一般用户（如果有的话）最好一次只用一个条件，同时用一个尾随的 `-print` 以便印出找到的文件名或者用一个尾随的 `-exec`（`ok`）以便当文件找到时执行某些简单的任务。在大型系统中，查找整个文件系统是十分花费时间的，应该力图把你的查找范围限制在一个尽可能小的子树内。

举例

印出在当前子树下的全部文件清单：

```
find . -print
```

找选项 -print 的作用可以通过使用 echo 命令来模拟：

```
find . -exec echo {} \;
```

找出在george的子树下属于john所有的全部文件：

```
find /usa/george -user john -print
```

找出在john的子树下为george或rik所有的全部文件：

```
find /usa/john \(-owner george -o -owner rik\)
-print
```

找出在文件系统中有两个或两个以上别名的所有的普通文件：

```
find / -type f -links + 1 -print
```

在文件系统中找出多于100 块长的全部文件，而且对于每一文件印出长格式列表：

```
find / -size +100 -exec ls -l {} \;
```

找出在文件系统中最近100 天内尚未存取的全部文件：

```
find / -atime +100 -print
```

找出在子树‘/usr’或‘/usr1’中的全部特别文件：

```
find /usr /usr1 \(-type b -o -type c\) -print
```

找出在当前子树下带有后缀“.c”的全部文件：

```
find . -name \*.c -print
```

grep

名字

grep——在文件中按模式查找。

格式

```
grep [任选项] 表达式 [文件……]
```

说明

grep程序在一个指定的文件中按正文模式查找，如果没有指定文件，则从标准输入中查找。正文模式由表达式自变量所

指定，此表达式类似于标准 UNIX 系统编辑程序的正则表达式。

不带任选项的 `grep` 程序将印出输入中包含与表达式匹配的正文模式的各行。可用任选项来稍稍改变这一功能：

-c 产生行数，而不是行本身。

-e 表达式

这个表达式几乎和一个简单表达式自变量相同。然而这个表达式是明显地以“-e”任选项作前导的，因而这个表达式允许以“-”开头。通常，表达式强制用某些字符开始而不允许以“-”开始。

-h 当有几个输入文件时，一般在每个文件匹配行前印上这个文件的名字。“-h”任选项将从输出中删除文件名。

-l 列出包含与表达式匹配的正文模式的文件的名字。

-s 使用 `grep` 的出口状态来指定匹配的存在，不产生输出。这个任选项对 `shell` 程序员很有用。

-v 印出不包含与表达式匹配的正文模式的所有行，而不是匹配的那些行。

-y 在表达式中的小写字母将匹配文件中的大写字母或小写字母。

-n 在输出行前加上行号。

举例

印出一组文件中包含字符串“nwords”的各行：

```
grep nwords chapt*
```

列出一组文件中包含字符串“nwords”的行数：

```
grep -c nwords chapt*
```

列出当前目录下包含字符串“artifact”的全部文件：

```
grep -l artifact*
```

印出文件‘rgb.c’或文件‘hsv.c’中包含字“bcount”全部

行及行号:

```
grep -n bcount rgb.c hsv.c
```

印出文件 'cmds.sh' 中不包含 "-" 的全部行:

```
grep -v -e - cmds.sh
```

判断是否有用户在tty30上注册:

```
who | grep tty30
```

kill

名字

kill——终止一个进程。

格式

```
kill [-信号号码] 进程标识号
```

说明

通常 kill 命令用来终止后台进程。为了终止进程，你应该知道进程标识号。当你是这个进程的所有者或超级用户时，你可用ps命令获取某一进程的进程标识号。当进程在后台运行时，你可记住由 shell 印出的这个进程标识号。

kill 一般送出终止信号（信号号码为15）给目标进程。偶而可把进程安排成能捉住信号但又不理会此信号，因而对于指定的进程，它可能接收了信号但不终止。通过在命令行上提供信号号码，也能送出其它信号号码。一个很有用的信号号码是9，它是不能被捉住或忽略的终止信号形式。

举例

终止进程1103:

```
kill 1103
```

送出口信号（信号3）给进程1106。退出（quit）信号一般将引起带有内存映象卸出（dump）的进程终止，它可用于调试:

```
kill -3 1116
```

ln

名字

ln——为一个存在的文件建立一个别名。

格式

```
ln 文件1 [文件2]
```

说明

ln命令用于为存在的文件建立一个新的名字（别名）。建立别名的技术名称是联结（link），因此取名为ln。如果在命令中出现两个文件：‘文件1’和‘文件2’，那么‘文件2’是‘文件1’的新名字；如果命令中没提到‘文件2’，那么在当前目录下建立了和‘文件1’有同样名字的别名，作为访问‘文件1’的路径名中的最后一个元素。

当一个文件有两个（或更多个）名字时，那么这两个名字有相同的地位。即使有几个名字，而这个文件也只有一个数据副本。

举例

为文件‘techrpt302’建立别名‘softrev’：

```
ln techrpt302 softrev
```

为很远的目录下的文件‘mkjuice’建立在当前目录下的别名‘mkjuice’：

```
ln ../fruit/tropical/mkjuice
```

lpr

名字

lpr——打印文件。

格式

```
lpr [任选项] [文件...]
```

说明

lpr 程序用于在行式打印机上打印文件。因为行式打印机不能同时为几个用户所共享，所以 lpr 程序就将各个用户的打印请求进行排队，一次安排打印一个文件。如果在命令行中没有提到文件，那么 lpr 就从标准输入上读进正文并且打印出来，因此 lpr 能被管道线用作最后一段。

lpr 不能用任何方法改动要打印的正文，如果你要按页打印文件或者加上标题，那么你在使用 lpr 之前，应该用 pr 命令预先处理正文。

在有几个行式打印机的系统中，通常有几个 lpr 的变体，每个打印机一个，常常给各种不同的变体指定类似于 lpr 的名字：如 npr, dpr, vpr, ppr 等。

可用下列任选项：

- c 立即复制文件，以防备打印前可能出现的改变。
- m 当打印完成时送出一个信件。
- n 当打印完成时不送信件，这是缺省条件。
- r 当文件已经为了打印而排上了队时，则删除此文件。

举例

在行式打印机上打印 ‘mydoc’：

```
lpr mydoc
```

在行式打印机上印出 ‘mydoc’ 的一个加标题 (pr) 的版本，用电子信件报告完成消息：

```
pr mydoc | lpr -m
```

ls

名字

ls——列出目录的内容。

格式

```
ls [-ltasdriu] (名字...)
```

说明

ls 命令用于列出目录中的文件及有关文件的信息。任选项可用于控制每个文件所印出的信息，以及清单的次序。“名字”可以是目录的名字，或是文件的名字。如果文件存在，那么印出每个命名的文件所要求的信息。如果文件不存在，那么印出一个简要的消息。对于每个命名的目录，将印出在目录中每个文件所要求的信息。如果没有名字，那么印出在当前目录下所要求的文件信息，因此命令

```
ls
```

等效于命令

```
ls .
```

文件清单一般是按字母先后顺序来排序的。通常以句点开头的文件名是不列出来的。下列任选项可用来更动这些功能：

- l 产生一个长格式列表（见第七章）。
- t 按文件修改日期而不是按字母次序来排序文件清单。
- a 列出命名目录下的全部文件，包括以句号开头的文件。
- s 按块为单位来印出文件的大小。
- d 对于每个命名的目录，列出目录文件本身的信息，而不是列出目录下每个文件的信息。
- r 将输出的次序倒过来。
- i 列出每个文件的索引节点数目。
- u 用存取时间而不是修改时间来排序，或在长格式显示上输出。

举例

列出当前目录下的文件：

```
ls
```

列出 ‘/etc’ 目录下的文件：

```
ls /etc
```

看文件 ‘/usa/bill/kill’ 是否存在：

```
ls /usa/bill/kill
```

列出文件 ‘/etc/passwd’ 所占块的多少：

```
ls -s /etc/passwd
```

列出文件 ‘/etc/passwd’ 的索引节点数目：

```
ls -i /etc/passwd
```

列出当前目录下的全部文件的登记项，包括名字以句号开头的登记项在内：

```
ls -a
```

列出目录 ‘/bin’ 下，按照修改时间排序、最近修改在前的全部文件清单：

```
ls -t /bin
```

列出当前目录下，以 “.doc” 作为文件名结尾、按最老的（最近一直未修改）文件在前的全部文件：

```
ls -rt *.doc
```

列出当前目录下，以 “.doc” 作为文件名结尾、按最近访问在前的全部文件：

```
ls -u *.doc
```

列出当前目录下，以 “chapt” 开头的、以逆字母次序排列的全部文件：

```
ls -r chapt*
```

列出父目录下全部文件的长格式清单：

```
ls -l ..
```

用长格式列出父目录。任选项 “-d” 用于强制列出目录本身，并禁止列出目录中单独的文件：

```
ls -ld ..
```

注

许多装置已经扩充了它们的 ls 命令，增加了一些任选项。详情可参阅你身边的 UNIX 系统手册。

mail

名字

mail——送信件给用户或读你所收到的信件。

格式

mail 用户名...

mail [-rpq] [-f信件的文件名]

说明

在 mail 命令中，如果用户名作为自变量提及时，那么信件被送至指定的用户处（上面格式中的第一种命令）。在所有其它情况下，mail命令用于读你收到的信件（上面格式中的第二种命令）。

让我们先讨论送信件给其他用户。mail从标准输入获得消息。你可以预先使用正文编辑程序准备消息，并使用shell的输入重新定向功能；或者你也可交互地打入消息，然后按control-d (EOF)去终止输入。如果指定的用户没能找到(例如你可能打错了用户名)，那么消息被保留在文件‘dead.letter’里，允许你决定正确的用户名，然后重新传递这个消息。

当你使用 mail 命令读你的信件时，其工作是不一样的。当你注册时，shell将印出消息“You have mail”，通知你有信件存在。mail接受四个命令行任选项以帮助你读你的信件：

- f 文件名 这个任选项说明命名的文件是信件的来源。而通常这个文件是在目录‘/usr/mail’下。
- p 不用暂停就印出全部消息。而通常消息是一次印出一个，在两个消息之间，系统提示你使用一个处置命令。
- q 通常，一个中断只是导致 mail 停止印出当前的消息。然而，当使用“-q”任选项时，一个中断接收时，程序将终止。

-r 先印出最老的消息。而通常却是先印出最新的消息。
mail通常在印出每个消息后停止，并等待你送入一个处理命令，mail可以识别下列处理命令：

<换行>或+ 印出下一个消息。

d 删除这个消息。

m 用户名 转递（邮寄）这个消息给指定的用户。

p 再次印出消息。

s [文件] 保存消息到指定的文件中（缺省时保存在‘mbox’中）。有一个首部将放在消息的前面，以便识别这个消息。

w [文件] 保存消息到指定的文件中，但省略首部。

- 返回到前面消息。

<control-d>或q 将未删除的信件放回到mailbox，并转出口。

? 印出一个求助的消息（某些装置使用星号代替问号以产生一个求助消息）。

通常只用 <换行>和“d”处理命令，你就能读你的信件，而不用命令行其它任选项。

举例

把文件‘msgfile’中的消息，送给Tom, Dick, 及Barry:

```
mail tom dick barry<msgfile
```

读你的信件:

```
mail
```

打入“d”去删除一个消息，并打“回车”印出下一个消息。

man

名字

man——印出UNIX系统手册的条文。

格式

man [-任选项] [节] 标题

说明

man命令用于定位印出UNIX系统手册的条文。当用户需很快地参考手册、而又没有一个印刷好的手册可用时，man命令能为用户产生手册的有关章节。系统管理员可把用man产生的手册的打印副本分发给用户们。正规印刷好的手册比重复使用man命令更为方便，但却比较陈旧（因为新的增强、修改等可能还未及反映在印好的手册中——译注）。

man命令识别下面几个任选项：

- t 使用 troff 正文格式加工程序，产生适合于照相排版的输出。
- n 使用 nroff 正文格式加工程序在标准输出上产生输出。
- e 使用 eqn（或当提供-n标志位时，使用 neqn）作为附加的正文预处理程序。
- w 印出手册登记项的路径名，而不是印出登记项。

“节”自变量指明在哪一节查找登记项。描述命令的手册登记项一般在第1节。如果“节”自变量被省略，则要查找手册的全部八节。

举例

印出手册中ls命令的条文：

```
man -n ls
```

印出手册中kill的条文：

```
man -n kill
```

对于上述命令将有两个引文产生：一个引文是UNIX系统手册第一节 kill命令，另一个是UNIX系统手册第二节 kill系统调用的引文。如果你只要kill命令引文，可打入命令：

```
man -n 1 kill
```

印出kill条文的路径名：

man -d kill

注

贝尔系统内部通常所使用 man 命令的任选项不同于上面所述的、通常在贝尔系统之外所用的man命令的任选项。

mkdir

名字

mkdir——建立一个目录。

格式

mkdir 目录名...

说明

mkdir 命令用于建立目录。为了建立目录，你必须在父目录中有写许可。

登记项 ‘.’ 和 ‘..’ 在目录建立时自动地被设置。

举例

建立命名为 ‘newsub’ 的子目录：

```
mkdir newsub
```

建立命名为 ‘/usa/kc/games/numoo’ 的目录：

```
mkdir /usa/kc/games/numoo
```

mv

名字

mv ——移动及重新命名文件。

格式

mv 文件 1 文件 2

mv 文件... 目录

说明

mv 命令用于管理文件。它最简单的形式是，改变一个文件的名字。mv 也能用于从一个目录移动一个文件（或一组文

件)到另一个目录。在一个文件系统里的移动实际上是复杂的重新命名操作:从一个文件系统移动至另一个文件系统包含了一个数据的实际传送。

在 mv 命令格式的第一种形式中,‘文件 1’被重新命名为‘文件 2’。在 mv 命令格式的第二种形式中,指定的文件被移至指定的目录下。文件保留它们原来的名字。

如果‘文件 2’已经存在,并且是被写保护的,那么 mv 将印出文件 1 的存取方式,并从标准输入读入一行。键入一个“y”,将引起操作继续进行,而键入“n”(或者其它)将暂停这个移动。如果‘文件 2’已经存在,并且不是写保护的, mv 将用‘文件 1’代替它。

举例

将文件‘newdb’重新命名为‘olddb’:

```
mv newdb olddb
```

将名为‘rjstat’的文件从当前目录移至子目录‘rjfiles’下:

```
mv rjstat rjfiles
```

将名为‘ddstat’的文件移至子目录‘rjfiles’下,取新名字‘xddstat’:

```
mv ddstat rjfiles/xddstat
```

将子目录‘rjfiles’下的全部文件移至父目录的子目录‘oldrje’下:

```
mv rjfiles/* .. /oldrje
```

nice

名字

nice——在低优先权下运行一个命令。

格式

```
nice [-增量] 命令 [自变量]
```

说明

`nice`命令用于降低进程的优先级,以便降低在系统里的这个进程对系统的要求。一般, `nice`和耗费时间的后台进程一起使用,以便防止这些进程降低系统的性能。

任选的“增量”自变量用于指定优先权降低的级数。在UNIX系统里,用增量“19”就能达到最低优先级,用增量“1”来达到在优先级上能察觉的微弱的降低。如果没有指定增量,那么假定缺省值是“10”。

普通用户只能降低他们进程的优先级,超级用户能通过指定一个负的增量,在提高了的优先级上运行作业。

举例

在后台以低优先级运行名为‘bigjob’的进程:

```
nice bigjob &
```

以最低优先级运行‘bigjob’:

```
nice -19 bigjob
```

以最高优先级运行名为‘import job’的进程:

```
nice --19 import job
```

注意:只有超级用户才能指定一个负的增量(在这里是-19)。

注

UNIX系统的所有进程,即使它们在低优先级上调度,也在竞争资源。在系统清闲期间(见`at`命令)运行进程而不是在高峰期间以低优先级运行进程是减轻系统负荷的一个更加有效的办法。

nohup

名字

`nohup`——运行一个程序使它免于被挂起。

格式

```
nohup 命令 [自变量]
```

说明

通常，在系统注销时，已经在后台工作的程序将会收到一个“挂起”信号。多数程序当它们收到挂起信号时就退出。nohup命令用于对进程初始化，使这些进程不理睬挂起和退出信号。因此当你从系统中注销时，任何已用过nohup命令，并运行在后台的进程将不死亡。

除非你使用输出重新定向来指定某些其它磁盘文件，否则，nohup将标准输出定向至文件‘nohup.out’。使用nohup运行的命令不需要用终端进行对话。

举例

使用nohup在后台对文件进行 nroff 命令，这样当处理正在进行时，你就可从系统中注销：

```
nohup nroff -ms chapt?.n >chaps.nr &
```

od

名字

od——卸出一个文件。

格式

```
od [格式] [文件] [位移量]
```

说明

od (octal dump)程序用于把一个命名文件(如无说明文件，则为标准输入)卸出。“格式”自变量允许你控制文件的卸出按八进制字(缺省情况)、八进制字节、ASCII字节、十六进制字、或者十进制字进行。除非提供了“位移量”，否则文件一般从起点开始卸出。

od可以识别下列“格式”控制自变量：

- b 按八进制解释字节。
- c 按ASCII码解释字节。
- d 按十进制解释字。

- o 按八进制解释字。
- x 按十六进制解释字。

当字节是按ASCII码解释时，下列转义被用于代表常用的不可印刷字符：

- \0 空白
- \b 退格
- \f 换页
- \n 换行
- \r 回车
- \t 制表符

按ASCII码解释时，如字节是不可印刷字符，而且没在上面表中列出，那么均用三位八进制数来表示。

“位移量”用于控制文件起点至卸出起点之间的距离。如果没有提到文件，则位移量必须用一个加号开始，否则位移量会按原有的数起始。除非有一个句点加在末尾，否则位移量被解释作八进制数。因此，位移量“10”解释作十进制8，而位移量“10.”解释作十进制10。如果位移量用一个“b”作后缀，那么这个数表示块（512字节）的数目，否则位移量以字节为单位。因此位移量“20”解释作十进制16个字节，而位移量“20 b”解释作十进制16块。

举例

按八进制字的格式卸出文件‘a.out’：

```
od a.out
```

或者等价地：

```
od -o a.out
```

按八进制字节格式卸出文件‘a.out’：

```
od -b a.out
```

按16进制格式从第10(十进制)块开始卸出文件‘a.out’：

```
od -x a.out 10.b
```

或者等价地:

```
od -x a.out 12b
```

用十进制格式,从第16字节开始卸出文件‘a.out’:

```
od -d a.out 16.
```

passwd

名字

passwd——改变注册口令。

格式

```
passwd [名字]
```

说明

passwd 命令用于为普通用户改变他们自己的注册口令,并且用于为超级用户改变普通用户的口令。当普通用户使用时,程序催问老的口令,以便证实是被授权的用户在改变口令,然后程序催问新的口令,以后程序再次询问新的口令以便确认口令正确地送入。为增强安全性,在口令送入时不产生回应。

超级用户在建立一个帐户时,使用 passwd 设置口令。而且在某些系统里,管理员为增强安全性,定期地改变用户的口令,所有的口令经过加密的版本保存在文件 ‘/etc/passwd’ 中。

举例

改变你自己的注册口令:

```
passwd
```

pr

名字

pr——将文件分页,加标题及格式化。

格式

pr [任选项] [文件...]

说明

通常pr命令用于为打印文件作准备。pr将文件分页、提供书眉、将文件划分成列、以及对不同的页长度和宽度进行调整。如果没有指定任选项，那么pr产生单列输出，每页66行，带有命名文件的一个短的书眉和一个短的尾部的输出。如果没有文件被命名，那就是指标准输入。标准书眉包括了日期，文件名，及页号。如果pr的输出是一个终端，那么在输出期间消息是被挂起的。

下列任选项能用来调整文件的格式：

- h 用尾随的自变量来替代书眉文件名。
- ln 产生 n 行长的页。缺省时是66行。
- m 同时印出全部文件，每个文件一列。
- n 产生 n 列输出。缺省时为单列输出。
- +n 在第 n 页开始输出。缺省时为第 1 页开始。
- sc 当多列输出时，用字符 c（通常是一个制表符）而不是用适当数量的空白间隔将各列分开。若没有 c，则假定为一个制表符。
- t 不产生书眉或尾部。
- wn 在多列输出时，用数 n 而不是缺省时的72作为页的宽度。

举例

在行式打印机上用方便的格式印出一个文件：

```
pr myfile | lpr
```

将ls程序的输出分列（删去书眉，使终端上的输出不至于超出视域）：

```
ls | pr -6t
```

从第10页开始印出文件‘myfile’，并把输出放在文件‘myfile.end’中：

```
pr + 10 myfile > myfile.end
```

注

许多系统已经扩充了pr命令，提供了几种附加的任选项。

ps

名字

ps——印出进程的状态信息。

格式

ps [任选项] 进程标识号

说明

ps命令用于印出有关活动进程的信息。ps为进程所做的也就是ls为文件所做的那些事情。没有任选项的ps命令为本用户的进程印出下列信息：终端设备名字、进程号、累计执行时间、以及命令行的一个缩写。下列任选项能用于修改ps命令的动作：

- a 印出与控制终端有关的全部进程信息，而不只是你自己的进程。
- k 为了检查使用文件‘/usr/sys/core’中的信息。这个任选项常常为系统管理员在一次系统瘫痪前所用。普通用户一般不需要ps的这个特性。
- l 产生一个长格式列表。在长格式的ps列表所提供的许多信息，对于普通用户来说，技术性太强了。
- number 如果一个进程号提及了，那么这个进程的信息就被产生。
- x 印出和控制终端没有联系的进程信息。

举例

产生一个当前进程的清单：

```
ps
```


产生当前进程的一个长格式列表：

```
ps -l
```

产生系统上全部进程的清单：

```
ps -ax
```

印出进程3402的有关信息：

```
ps 3402
```

注

UNIX 系统手册中对ps命令的长列表格式有解释。还要注意：普通在贝尔系统中使用的ps命令和这里描述的在贝尔系统之外所使用的ps命令，有不同的任选项。

pwd

名字

pwd——印出工作目录的名字。

格式

```
pwd
```

说明

pwd命令印出当前目录完整的路径名。

举例

印出工作目录的名字：

```
pwd
```

rm

名字

rm——删除文件。

格式

```
rm [-fri] 文件...
```

说明

rm 命令用于删除文件。为了删除文件，你对这个文件所处

的目录必须具备写许可，但你不需要有对该文件本身的写许可。如果对这个文件你没有写许可，则 `rm` 将印出文件方式，并等待你送入“y”或“n”以指示你是否真要删除这个文件。

可使用三个任选项：

- f 这个任选项将不管你是否具有这个文件的写许可而强制删除文件——将不再出现为了核准删除具有写保护文件的普通询问。
- i 这个用于交互的任选项，将使 `rm` 问你是否真的要删除每个指定的文件。回答是“y”或“n”。
- r 这个递归任选项用于删除文件系统的一棵子树。文件名自变量应该是目录名。这个目录，它的全部文件，子目录，等等都被删除。“-i”任选项，可以和“-r”合用，以便使递归删除能得到一些控制。

举例

删除文件 ‘mydocs’：

```
rm mydocs
```

删除几个文件：

```
rm nicotine caffeine tar
```

删除子目录 ‘xyresp’ 下的全部文件：

```
rm xyresp/ *
```

交互地删除子目录 ‘xyresp’ 下的全部文件：

```
rm -i xyresp/ *
```

不用询问写保护文件，而删除子目录 ‘zzresp’ 下的全部文件：

```
rm -f zzresp/ *
```

删除由目录 ‘/usa/kc/nudocs’ 开头的子树：

```
rm -r /usa/kc/nudocs
```

rmdir

名字

rmdir——删除目录。

格式

rmdir 目录名...

说明

rmdir 命令用来删除空目录，按定义，一个空目录是只包含两个登记项 ‘.’ 及 ‘..’ 的目录。为了查看一个目录的全部文件清单，你应使用命令：

```
ls -a dirname
```

如果只有两个登记项 ‘.’ 和 ‘..’，那么目录是空的。要删除目录，你必须具有它的父目录的写许可。

举例

删除目录 ‘/usa/kc/games/numoo’：

```
rmdir /usa/kc/games/numoo
```

删除子目录 ‘sortsh’：

```
rmdir sortsh
```

sort

名字

sort——排序与/或合并文件。

格式

```
sort [-cmu][[-tc][[-bdfinr][ + pos1[- pos2]]]...] [-o 输出文件] [文件名.....]
```

说明

sort程序对输入文件按行进行排序，然后将重新安排的行写到标准输出去，如果使用“-o”任选项，则写到指定的“输出文件”去。普通情况下，排序码是整个行。可是若用位置任选项，则排

序码限制在一行中被指定的字段上。有几种位置任选项来指定排序码；仅当较早的码相等时才使用较后的码。作为最后一着，当一行的其它方面相比较全部相等时，就要用全部有效的位置作为码来排序。

下列五个任选项用于控制 sort 程序的一般行为（简短地讨论六个次序任选项）：

- c 检查输入文件，验证它是否按规则被排序了。只有在文件正确地排序时才有输出产生。
- m 合并输入文件。可能这些输入文件已经排了序。
- o 输出文件 输出被写到命名的文件上、而不是标准输出上。
- tc 字符 c 用作字段分隔符。空格（空白及制表符）一般用作字段分隔符。
- u 在一组相等的行内只留下一行。作为这个任选项的目的，相等行定义为，在所有起作用的排序码比较时均相等的行。在排序码之外的字段以及非法的字符（见“-b”及“-i”任选项）相等判断时不加考虑。

如果不指定位置，则排序码是整行。当使用位置指示符 + pos1 时，强制排序码从指定位置开始。位置指示符 - pos2 强制排序码正好停止在指定位置之前。如果未指定 - pos2，则排序码停在行尾。位置指示符具有“m. n”形式，其中 m 表示从行的起点处所要跳过的字段数，n 表示还要跳过的字符数。位置指示符“+5.2”表示：从起点跳过五个字段，然后再跳过两个字符之后开始排序码。位置指示符“-0.2”表示排序码在从起点跳过零个字段，再跳过两个字符之后结束。如果一个位置指示符的 n 部分没有，则假定为 0。

有六个任选项用于控制项的次序。如果这六个任选项出现在任何位置指示符之前，那么它们是全程任选项。这六个次序任选项也能放在位置指示符之后，以便改变特别字段的缺省次序。

- b 当一个字段作比较时，忽略开头的空格和制表符。
- d 按字典的次序，而不是按缺省的 ASCII 码相应的值作次序。只有字母、数字、和空格在字段比较时有意义。
- f 为了进行字段比较，将大写字母并到小写字母里。输出时保存原有的大小写不变。
- i 在非数值比较时，ASCII 码范围 040—0176 之外的字符不予理会。
- n 执行数值比较。一个数可以有前导空格、一个可选的减号及带一个可选的小数点的零个或多个数字。按照数目的值而不是字典次序、或者 ASCII 码的相应序列排序。
- r 按递减次序将项排序。而正常情况下，项是按递增次序排序的。

举例

对于前面几个例子，假定文件 ‘tdata’ 包含下列四个字，每字一行：“apple”，“Balloon”，“apple” 及 “Apple”。因为处理重复是排序过程控制中的主要问题之一，所以我们采用二行一样的例子。

排序 ‘tdata’，并把输出放入 ‘tdata.1’：

```
sort -o tdata.1 tdata
```

产生的次序是“Apple”，“Balloon”，“apple”，“apple”，因为在 ASCII 码中，大写字母先于小写字母。

排序 ‘tdata’ 而不管大小写：

```
sort -f tdata
```

产生的次序是“Apple”，“apple”，“apple”，“Balloon”。三个 apple 字是按照最后一着规则来排序的，即当各行进行比较并相等时，按照每个位置哪一个先有效来排序。

排序 ‘tdata’ 并去掉相同的行：

```
sort -u tdata
```

产生的次序是“Apple”,“Balloon”,“apple”。

排序 ‘tdata’, 不管大小写, 去掉相同行, 如果大小写均有, 则输出大写字母。

```
sort -f tdata | sort -muf
```

这个管道线的第一次排序是将行以 ASCII 码的相应序列排序, 而管道线的第二次排序是将已排序的数据合并, 去掉重复行。合并决不会更动次序, 因此当提出“-u”任选项时, 单一文件的合并可让人预知那些项将被丢弃(后继项)。所产生的次序是“Apple”,“Balloon”。

排序 ‘tdata’ 来产生全部单一的字。当一个字用大写及小写两种形式时, 在最后清单中要求大写字跟在小写字的后面。

```
sort -u +0f +0r tdata
```

这实质上是一个两遍排序, 因为有两个排序码, 每个都是整行。仅对按第一码排序后相等的登记项才按第二码排序。在我们例子中, 由于“f”任选项紧跟在第一个位置指示符后, 三个 apple 字比较将为相同的。第二个码指定一个逆序(“r”任选项), 这将把大写 Apple 字放在小写 apple 字的后面, 而且“-u”任选项将去掉重复的小写 apple。产生的次序是“apple”, “Apple”, “Balloon”。

对于下面几个例子, 假定文件 ‘numbs’ 包含下面三行:

“ab: 40”, “cd: -20” 及 “ab: .30”。

按字典次序排序 ‘numbs’:

```
sort -d numbs
```

产生的次序将是“ab: 40”, “ab: .30”, “cd: -20”。冒号,“-”, 句号, 加号在字典次序下全部被忽略。

按第二字段来排序 ‘numbs’ (字段用冒号分开):

```
sort -t: +1 numbs
```

产生的次序将是“ab:40”, “cd:-20”, “ab:.30”。

因为在ASCII序列中,空白先于“-”,“-”先于句号。
按照第二字段的数值次序排序 ‘numbs’:

```
sort -t: +1n numbs
```

产生的次序将是“cd:-20”,“ab:.30”,“ab:40”。

主要按照第一字段对 ‘numbs’ 排序,再者按照第二字段的数值以逆序来排列。

```
sort -t: +0 -1 +1rn numbs
```

产生的次序将是“ab:40”,“ab:.30”,“cd:-20”。

spell

名字

spell——在正文文件中检查拼法。

格式

```
spell [任选项] [文件...]
```

说明

spell程序从指定的文件里取出所有字,当没有文件名时,就从标准输入里取,并在字典里查寻它们。所有不在字典里的字,以及不能从字典字中经过标准变化、加前缀、后缀而派生出来的字,均被作为可能的拼错列在标准输出上。某些字虽被spell标志为正确,而在正文内的一些错误却未被检查到。然而,即使spell不很完美,但它对定位某些拼写错误仍是一种有价值的辅助手段。

spell支持三个任选项:

- b 检查英国拼法。
- v 印出所有不在字典里的,或不是字典字派生出的字。
- x 对于每个字印出每个可能的词干。

这些任选项极少为很多用户所使用。

举例

对于一组正文文件,检查拼错的字:

```
spell xydocs?
```

对于和上面同样的一组正文文件，检查英国拼法：

```
spell -b xydocs?
```

stty

名字

stty——设置或显示终端任选项。

格式

```
stty [任选项]
```

说明

stty命令用于控制不同的任选项，使系统能正确地运用你的终端。安排stty命令的一个原因是：有许多类型的终端，并且它们都需要有稍微不同的对待。用stty命令的另一个原因是：允许你去规定某些UNIX系统特性（如规定erase和kill控制字符）。如果没有指定任选项，则stty将报告有关一些关键任选项的当前设置。

很明显，stty是实用程序中最依赖于机器的一种。下面任选项清单是运行在数字设备公司(DEC)小型计算机系列上的UNIX系统第七版本所具备的。在不同机器上的UNIX系统，很可能具有不同的或另加的设置，新的设置很可能是计算机终端及接口电子学方面进展的成果。

下列任选项通常用于控制你的终端运行：

even(-even) 串行传输时进行（不进行）偶校验。

odd(-odd) 串行传输时进行（不进行）奇校验。奇校验及偶校两者通常不能同时进行。在一给定时刻，只能用一种（奇或偶）校验。

raw(-raw) 启动（禁止）原始输入方式。在原始输入方式中，系统不执行正常输入处理，特别是，字符erase, kill, interrupt, quit, eof将被提交给正在

运行的程序,所有带奇偶位的字符也被提交。你的终端处理程序不会经常在原始输入方式下工作的。

cooked 同-raw

cbreak 一次读一个输入字符。erase及kill是无效的,而interrupt, quit及eof像平常一样起作用。

-cbreak 只有当一行结束按换行或回车键时,输入的字符才有用。erase, kill, quit及eof均能正常工作。这是许多UNIX系统的正常工作方式。

-nl 允许用回车或换行表示输入行的结束。

nl 只允许用换行表示输入行的结束。

echo (-echo) 当打入每个字符时回应(不回应)。

-echo 方式只用于送入口令及其它敏感信息的场合。其它情况下在终端上通常有回应。

lcase (-lcase) 映射(不映射)大写输入到小写,也映射(不映射)小写输出为大写。lcase方式只用于没有小写字母的终端; -lcase方式用于具有大小写字母的多数终端。

tabs (-tabs) 在输出上保存tabs(用空格代替tabs)。

-tabs任选项用于不知如何扩展tabs的终端, tabs任选项用于懂得tab字符的终端。

ek 置erase字符为#, 置kill字符为@。

erase c 置erase字符为c(c表示任一个键盘字符)。一个控制字符能用一个^放在字符前来指示(由shell来转义)。

kill c 置kill字符为c。

cr0 cr1 cr2 cr3 回车延迟(分别为0, 0.08, 0.16及0秒)。一些终端要求接收回车后有一点延迟。多数终端使用cr0设置。

n10 n11 n12 n13 换行延迟(分别为0, x, 0.1 及

0 秒)。n11延迟依赖于列数,偶尔用于打印终端。

tab0 tab1 tab2 tab3 tab延迟说明: tab0说明无延迟, tab1指定依赖于列数的延迟, tab2指定一个0.1 秒延迟, tab3与tabs相同,即用空格代替tabs。

ff0 ff1 换页延迟 (分别为 0, 2 秒)。

bs0 bs1 退格延迟 (分别为 0, 0.05秒)。

hup (-hup) 当最后一次结束时,挂断(不挂断)数据电话线的连接。

0 立即挂断数据电话线的连接。

50 75 110 134 150 200 300 600 1200
1800 2400 4800 9600 置波特率。

举例

印出当前任选项的清单:

```
stty
```

置计算机的通信接口速率为300 波特:

```
stty 300
```

置erase字符为control-h:

```
stty erase \^h
```

置kill字符为control-u:

```
stty kill \^u
```

在输出期间,让计算机把tab扩展为空格:

```
stty -tabs
```

置erase及kill字符为其缺省值#及@。

```
stty ek
```

tail

名字

tail——显示一个正文文件的尾部。

格式

`tail` [位移量] [文件]

说明

`tail`程序印出命名文件的最后一部分,在无文件提及,就印出标准输入的最后部分。如未提及位移量,则`tail`印出文件的最后几行(通常10行)。

位移量任选项用于控制印出文件末尾的行数。当位移自变量由一个`-`号前导时,位移量就相对于文件尾计算;当位移自变量由一个`+`号前导时,位移量指定为从文件头开始跳过的数量,位移量的单位用“`l`”,“`b`”及“`c`”分别指定为行、块(512字节)及字符,如未提到单位,则假定以行为单位。

举例

印出文件‘`wrdstr.l`’的最后几行:

```
tail wrdstr.l
```

印出‘`wrdstr.l`’的最后200个字符:

```
tail -200c wrdstr.l
```

印出文件‘`bell.5.1980`’的500行后的全部行:

```
tail +500 bell.5.1980
```

印出目录‘`/bin`’的长格式列表的最后23行:

```
ls -l /bin | tail -23l
```

tee

名字

`tee`——重复标准输入。

格式

`tee` [-i] [-a] [文件名...]

说明

`tee`程序重复标准输入。在管道线中的`tee`命令,相当于管道线路中的T型接头。`tee`通常用于一个程序输出既要放在一个文

件内，又要看见它的场合下，tee也能用在管道线中去保存处理的中间阶段，正常情况下它是不能看到的。

“-i”任选项导致tee不去理会中断，“-a”任选项导致tee将输出加到原来输出文件的后面，而不是将原输出文件盖掉。

举例

把spell程序的输出保存至一个文件，并同时可在屏幕上观察输出：

```
spell chl.doc | tee chl.errs
```

把wc程序的输出保存至一个文件，同时加这个输出至另一个文件的后面，并在终端上看到这个输出：

```
wc chl.doc | tee -a chl.logerrs | tee chl.errs
```

time

名字

time——为一个进程计时。

格式

```
time 命令名 [自变量]
```

说明

time命令允许你对一个进程计时。当进程完成时，time程序印出三个指示：总共过去的时间、进程的执行时间及进程的系统时间。总共过去的时间精确到秒，而执行时间和系统时间以六十分之一秒来度量。这个时间依赖于各种不同的随机因素，包括最显著的系统负荷。

举例

为who进程计时：

```
time who
```

为产生目录‘/bin’的长格式列表的ls程序计时：

```
time ls -l /bin
```

tty

名字

tty——印出终端的特别文件名。

格式

tty

说明

tty命令印出用作标准输入的特别文件名。如果标准输入不是一个文件，则tty命令印出一个类似于“not a tty”的消息。

举例

印出终端特别文件名：

```
tty
```

wc

名字

wc——对文件中的字、行、字符计数。

格式

```
wc [-lwc] [文件名……]
```

说明

字计数程序用于在正文文件中统计正文的单元。一般情况，wc报告它的输入文件的行、字、及字符的数目，如果在命令行中未提及文件，则指标准输入。任选项导致wc只统计字(-w)、行(-l)、或字符(-c)。

举例

对文件‘mydoc’统计行、字、字符个数：

```
wc mydoc
```

等效地可用命令：

```
wc -lwc mydoc
```

统计mydoc’的字数：

```
wc -w mydoc
```

统计当前目录下文件的个数：

```
ls | wc -l
```

who

名字

who——列出使用系统的用户名。

格式

```
who
```

说明

who命令产生当前系统已经注册的全部用户名单。这个清单包括注册名字、注册时间及每个用户的终端号。

专门命令“who am i”(在另一些系统中用“whoami”)通常产生你的注册名字。

举例

列出当前用户名：

```
who
```

列出你的注册名字：

```
who am i
```

write

名字

write——参加用打字方法进行的双路对话。

格式

```
write 用户名 [终端名]
```

说明

write命令允许你和另一个已经在你系统上注册的用户开始(或响应)对话。开始和一个用户名为“tom”的人对话，应打入命令：

```
write tom
```

用户名能用who命令或通过考察文件‘/etc/passwd’来推断。如果tom在几个不同的终端上注册,那么你可用命令:

```
write tom tty50
```

用‘/dev/tty50’连接写给tom。

一旦你已送入命令,类似于“Message from ralph on tty 30”的消息将出现在Tom的终端上。Tom应该放下他正在做的工作,并送入命令:

```
write ralph
```

去完成这种连接。在这以后, Tom或Ralph打入的任何东西将在他们两个的终端上出现。为了避免混淆, 在一个时刻应只有一个用户打字。通常开始对话的人将打入一个消息, 然后打入一个“o”表明“结束”。然后另一方打入消息并尾随“o”。这类似于CB无线电通信中所用的协议。在对话结束时, 你应该送入一个“oo”表明“结束并退出”, 并且应打入control-d以便停止write程序。

小 词 典

Access Mode 存取方式 (也称存取权限)

一种文件的保护信息。在UNIX系统中,文件可以由文件的所有者、同组用户或其它用户读、写或执行。存取方式详细地说明了提供给三类用户(所有者、同组用户和其它用户)的操作(读、写或执行)。在一个共享的计算机系统中,存取方式在一定程度上确保了用户文件的保密和安全性。

Acoustic Coupler 声耦合器

把电信号转换成电话声或者把电话声转换成电信号的一种设备。参见modem。

Application Program 应用程序

一种专用的计算机程序,如会计程序等等。

Argument 自变量

传递给命令的附加信息。命令名和它的自变量之间用空格或制表符分开。自变量通常也用来控制命令的操作。

ASCII 美国信息交换标准码

很多计算机和数据终端使用的标准码。

Assembly Language 汇编语言

直接与特定计算机的基本指令系统相关的程序设计语言。一般汇编语言程序用在需要高效率程序的场合下。汇编程序设计语言的缺点是:汇编语言程序难于编写,并且很难从一种型号的计算机移植到另一种型号的计算机上。

Background Process 后台进程

在一定意义上说,这是一种运行时无需监视的进程,当后台进程运行时,其它程序仍可开始及对话。当UNIX系统引导时有

一些后台进程启动，以便执行系统管理功能。另一些后台进程由用户以交互方式启动，以便执行该用户的工作。

Batch processing 批处理

一种非会话型的数据处理方式。在批处理系统中，程序由用户正式提交，然后由操作系统调度执行。为了优化机器的使用效率，程序进入系统不一定能立即执行，期间可能有明显的延迟（经常有数小时）。

Baud Rate 波特率

计算机与计算机或计算机与仪器设备之间的传输速率，以每秒多少位来度量。波特率被10除后的结果大致就是每秒传输的字符数。

‘/bin’ bin目录

在大多数UNIX系统中，它是包含有最常用的命令的目录。

Binary 二进制

这是一种以数2为基数的计数系统。在二进制系统中，数字就是0和1。二进制在计算机中很重要，因为计算机是由逻辑部件构成的，而这些逻辑部件取两种状态中的一种，相应于二进制的数字0和1。

Binary File 二进制文件

这是些包含了不是ASCII字符集代码的文件。二进制文件的每个字节可用所有256种可能值（ASCII代码文件的每个字节只能用128种可能值一译注）。遗憾的是，它们不能在你的终端上打印出来，因为这256种值中大部分不是可打印的ASCII字符。二进制文件可以用od程序把二进制代码转换成相应的可打印的ASCII字符加以检验。

Bit 位

一个二进制数字，它是数据的最小单位。

Bit Bucket 空洞

UNIX系统的空洞是一个称为‘/dev/null’的特殊文件。在

计算机术语中，空洞是那样一个地方：如从那儿输入，将得不到任何东西；如输出到那儿，则输出将销毁。参见Null Device。

Block Special File 块特别文件

这是一种特别文件，它提供了与某种设备的接口，这种设备具有支持文件系统的功能。

Booting 引导

启动系统的过程。

Bourne Shell

这是用在UNIX系统第七版中的shell程序，因作者S·R·Bourne得名，参见shell。

Break Key 中断键

这是终端键盘上的一个键，它向宿主机发送一个明确的代码。在注册期间，可用中断键使UNIX系统改变通信速度，以便与你的终端同步。

Buffer 缓冲区

数据暂时存放的地方。

Bug 故障

在计算机软件或计算机硬件中的错误。

Byte 字节

一定数目（通常是8）的位。目前，大多数实用的海量存储设备和I/O设备都设计成传送8位（字节）的系列设备。

Character 字符

用来组织、控制或表示数据的符号。在终端键盘上有相应的键，包括所有字母数字键，标点符号，以及其它特殊符号。字符通常以单字节存放。

Character Special File 字符特别文件

要作为与I/O设备接口的特别文件。字符接口供不能支持文件系统的设备用，也可作能支持文件系统的设备的另一种接口用。

C-Language C语言

这是UNIX系统的基本语言，是一种通用程序设计语言。它不是非常高级的语言，但因它表达简洁，没有很多限制以及通用性好而受到人们的称赞。C语言是D. M. 里奇开发的。

Command 命令

要控制系统完成某个功能的指令。虽然大多数命令归结为程序的执行，但是某些命令只在shell内部处理。

Command File 命令文件

包含若干shell命令的普通文件。当文件只包含一个或少量几个命令时，用术语“命令文件”；而当有大量命令或者使用了shell的循环和条件执行等功能时，就用术语“shell程序”。

Command Interpreter 命令解释程序

这是操作系统的一个组成部分，它解释并且执行由用户打入的命令。UNIX系统的命令解释程序称为shell。

Command Name 命令名

命令的第一个字。跟在命令名后面的字称为自变量。有时，命令名也叫做第0个自变量，因为在程序中，随后的自变量以1开始依次编号。

Compiler 编译程序

把用某种高级程序设计语言编写的程序正文文件翻译成可以执行的机器语言输出的一种计算机程序。在UNIX系统中，由编译程序编译出的机器语言称为目标文件。

Concatenate 串接

把若干个文件一个挨一个地组合在一起的操作。这种操作通常由cat程序来完成。

Conditional 条件

仅当一个确定的条件存在时，才能使一个语句(或若干语句)执行的一种程序设计语言结构。

Context Search 上下文检索

通过打入一种你要系统定位的正文模式，在一种指定的文件

中检索正文。可以在编辑程序内部完成上下文检索，也可以用grep命令完成上下文检索。

Control Character 控制字符

用于控制光标移动或屏幕清除等动作的功能码。控制字符通常混在正文中。在终端上常通过按下控制键（CTRL）的同时按下字母键来打入控制字符。

Control-d

参见EOF字符。

CPU 中央处理部件

计算机的控制器、运算器和逻辑部件的总称。

Crash 瘫痪

没有料到的计算机服务中断，这通常是由于严重的硬件或软件故障引起的。

CRT 阴极射线管

可以在上面显示信息的电视屏幕。

Current Directory 当前目录

其文件可以直接存取的目录。在你与UNIX系统会话的整个期间，只有一个当前目录，当前目录的名字可以用pwd命令显示。也可以用cd命令把当前目录改成另一个目录。

Current Subtree 当前子树

其根是当前目录的子树。即，这棵树包括当前目录、当前目录的所有子目录和文件、这些子目录的子目录和文件，等等。

Cursor 光标

这是显示终端上的一个特别符号，它指示下一个字符将要出现的地方。光标通常是一个小方框或者是一个下划线，并且可能是一闪一闪的。

Data 数据

信息的基本元素，它可以被计算机处理或产生。

Data Processing 数据处理

使用机器（通常是计算机）来操作信息。术语“数据处理”通常指使用计算机的商业活动。

‘/dev’ dev目录

通常可以从中确定特别文件的目录。

Diagnostic 诊断信息

由程序产生的一种出错消息，用来提供与程序中的故障有关的信息，或者指出程序环境中的问题的出错消息。

Dialogue 对话

用户和UNIX系统之间的一种会话。在通常的UNIX系统对话中，shell先显示提示符，然后用户打入一条命令，再按一个返回键，命令被执行。随后，shell显示另一个提示符。

Dial-Up Terminal 拨号终端

通过公用转换电话网与计算机连接的一种终端。

Directory 目录

是一种比较特殊的文件。目录用于组织和构造文件系统。如果没有提供UNIX层次目录系统这种组织形式，要管理在典型的UNIX系统装置中存在的几千个文件是非常困难的。ls命令用于列出目录中的文件清单。当你第一次注册进入系统时，你就在你的主目录中。可以用cd命令移动到别的目录上去，也可以用pwd命令打印当前目录的名字。

Disk 磁盘

在磁盘驱动器上使用的介质。磁盘通常是带磁性材料涂层的圆盘片。根据盘片的硬度，磁盘分成硬盘和软盘。硬盘的存储容量通常比软盘大。

Disk Drive 磁盘驱动器

一种使用旋转磁性介质（磁盘）存放信息的硬件设备；一种海量存储设备。

Disk File 磁盘文件

存放在海量存储设备上的命了名的信息集合。磁盘文件被说

成是永久性的，因为即使电源从海量存储设备上切断，这些文件也不会丢失。

Display Terminal 显示终端

使用CRT作为输出设备的一种计算机终端。

Echoing 回应

UNIX系统再现用户打入的输入。你打入的字符通常发送给UNIX系统，然后UNIX系统回应，从而，你打入的字符出现在你的终端上。有时禁止回应（例如当你打入你的口令时）。

Edit 编辑

修改或更换信息的动作。通常指通过正文编辑程序在正文文件中进行的有关操作。

Editor 编辑程序

参见text editor。

Electronic Mail 电子邮寄

向本系统内的其他用户或向其它系统内的用户传送信息（备忘录、报文、信件等等）的系统。

End of File(EOF) Character 文件结束(EOF)字符

control-d是UNIX系统的文件结束符（通过同时按下控制键和字母“d”得到）。由于shell遇到文件结束符通常要停止处理，因此，从UNIX系统中注销的一种方法是在shell提示符后按下control-d。

Erase Character 抹字符符

抹字符符抹去输入行上当前打入的前一个字符，一次抹一个。原来它被指定是#键，但可以用stty命令重新指定为其它的键。

‘/etc’ etc目录

这是一个UNIX系统目录，此目录中含有各种各样系统管理所要用到的文件。

‘/etc/passwd’

这是一个UNIX系统文件，此文件中含有系统中每一个用户

的主要注册信息（口令、注册名、用户主目录号，以及用户shell名）。

Execute 执行

完成存放在一个普通文件中的指令的操作。

Execute Permission 执行许可

对于普通文件来说，执行许可是允许你执行该文件的一种存取方式。对于目录文件来说，执行许可是允许你在跟踪路径名过程中检索目录的一种存取方式。

Execution Time 执行时间

计算机完成一条已知命令所需的时间。

Fifo 先进先出

fifo就是先进先出。在UNIX系统中，先进先出文件就是一种命了名的永久性管道。fifo允许两个无关的进程通过管道连接交换信息。普通的管道只在相关的进程间起作用。并不是所有系统都有fifo的。

File 文件

命了名的信息的集合。文件通常存放在一个海量存储设备上，UNIX系统把文件收集成称做目录的若干组。

Filename 文件名

用于识别特定文件的名字。

Filename Generation 文件名生成

shell把含有元字符的命令行扩展为相应的一组文件名的过程。例如，在含有文件‘x.doc’和‘nm.doc’的目录中，shell文件名生成过程把字“*.doc”扩展成文件‘nm.doc’和‘x.doc’。

File System 文件系统

在海量存储设备上文件和文件管理结构(索引节点)的集合。在UNIX系统中，文件系统是层次结构的。

Filter 过滤程序，过滤器

从标准输入上读取信息，并且把结果写到标准输出去的一种

程序。

Foreground Process 前台进程

相对于后台进程而言的一种交互运行的进程。可以有几个活动的后台命令，但是在正常环境下，只能有一个活动的前台进程，或许还有几个不在活动的前台进程。

Flag

见option。

Graphics 图形学

研究图形利用和图象显示的科学。

Group 同组用户

同一个部门，在同一项工程上工作的，或者以其它方式相关的若干用户成员。每一个UNIX系统文件与一个确定的组相关，规定该组成员有存取该文件的权限。

Hardware 硬件

计算机系统机械和电子部件。

Hardwired Terminal 硬连线终端

通过专用线路连接到计算机上的终端。

Header 标题

在文件的开头，详细说明文件的大小、位置等等信息的一种记录。

Hexadecimal Radix 十六进制

一种数基为16的计数系统，它的数字是0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E和F。

Hierarchy 层次结构

可以一层压一层地排列的人员组织或其他事物构成的系统。UNIX文件系统是层次结构的。

High Level Language 高级语言

程序设计语言的通称，它支持计算过程的形象思维，与此对应的是汇编语言，它直接与特定的计算机体系结构有关。C，

FORTRAN, BASIC和PASCAL等都是高级语言。

Home Directory 主目录

注册过程完了时用户所处的目录。

Inode i节点或索引节点

在UNIX系统中，这是管理文件的关键性内部结构。i节点包含了有关文件的方式、类型、所有者和位置的所有信息。i节点表存放在每个文件系统中靠近开头的地方。

Input Redirection 输入重新定向

shell从一个文件上而不是从终端上重新获得输入。参见standard inpnt。

Interactive Computer System. 交互计算机系统

一种会话系统，它允许用户和计算机同时对话。

i-number i节点号

在一个文件系统中指明一个特定i节点的号码。

I/O 输入/输出

描述计算机和海量存储设备、终端及打印机等外部设备之间传送信息的过程的一般说法。

I/O Device 输入/输出设备

可以与计算机交换信息的外部设备（如打印机、终端、磁盘等）。在UNIX系统中，I/O设备通过特别文件存取。

kernel 核心

UNIX操作系统中常驻内存的那部分程序和数据，其中包括所有立即且频繁需要的UNIX系统功能程序。核心主管I/O传送、管理和控制硬件并且调度用户进程去执行。UNIX系统核心是由大约10000行C语言和大约1000行汇编语言代码行编译而成的。在UNIX系统中，核心只含有相对少的功能（相对其它操作系统而言），以致可以通过一个个实用程序，更方便地发挥核心功能的作用。

Kill Character 抹行符

抹行符让用户抹去整行。原来它被指定为@键，但可以用stty命令重新指定为别的键。

Login 注册

为用户提供访问系统的权利的过程。

Login Directory 注册目录

参见home directory。

Login Name 注册名

在注册过程中用户使用的名字。

Logout 注销

通知系统，用户将要退出并且不再对系统提出进一步要求的过程。

Machine Language 机器语言

计算机的基本语言。

Macro Package 宏程序包

在UNIX系统中，术语“宏程序包”通常用来描述一组高级nroff/troff正文格式加工命令。内部的nroff/troff命令是非常低级的，并且不方便，宏程序包引入了一组更为方便的正文处理功能。

Mail 邮寄

参见electronic mail。

Mass Storage Device 海量存储设备

储存大量信息的部件，通常是磁盘、磁带或盒式带等部件。储存在海量存储设备上的信息可由CPU存取，虽然存取时间比存取在主存上的信息所花的时间要长得多。大多数信息是用磁性储存在海量存储设备上的。海量存储设备也称辅助存储器或二级存储器。

Memory 存储器

可以把信息复制到上面、存储到上面并且在以后可以从中取出的设备。通常这个术语指的是计算机的主存，它是可以存放适

量信息的电子设备，CPU可以十分迅速地存取其中的信息。主存也称为一级存储器。

Metacharacter 元字符

在某种场合下具有特殊含义的键盘字符。例如，当你打入“ls *.c”那样的 shell 命令时，* 可用来匹配任何一系列字符。如果你希望使用元字符而又不它的特殊含义，那么你必须使用引号把它括起来。

Modem(modulator-demodulator) 调制解调器

这种设备把数据信号从一种与数据处理仪器兼容的形式转换成另一种可以远距离传送的形式或其相反，传送通常经由公共电话线。

motd(Message of the Day) 日志消息

系统管理员适时放置消息的正文文件(‘/etc/motd’)。在许多系统中，当你注册时，motd文件将打印出来。

Multiprogramming 多道程序设计

在单独一个计算机上同时运行若干程序或例程的能力。

Multuser 多用户

同时支持若干用户的能力。

Named Pipe 命名管道

参见fifo。

Null 空

经常用来表示空的或不存在的內容的一个术语。

Null Device 空设备

UNIX系统的空设备称为‘/dev/null’。当你把输出定向到空设备时，输出就销毁了；当你从空设备上读取输入时，你立即会遇到文件结束符。有时，为了销毁输出而把它定向到空设备上，而为了空读，有时从空设备上读输入。

Null String 空串

不包含任何正文的正文串。空串的长度为0。

Object File 目标文件

含有可以由计算机执行的机器指令的文件。在UNIX系统中，目标文件是编译后的结果。

Octal Radix 八进制

一种数基为8的记数系统，其数字是0，1，2，3，4，5，6和7。

Operating System 操作系统

管理计算机资源的程序。操作系统简化输入/输出过程、进程调度和文件系统那样的内务工作。

Option 任选项

改变命令操作的一种自变量。通常，任选项是前面带-的单个字符。例如，在shell命令“ps -l”中，任选项是字母“l”，它控制ps命令去完成进程的长格式列表而不是通常的短格式列表。

Ordinary File 普通文件

普通文件用于存放数据。普通文件经常包含程序、资料、信件、数据库和其它类型的信息。

Output Redirection 输出重新定向

shell把标准输出重新与一个特定的文件相连。参见standard output。

Password 口令

用户在识别过程中打入的一组独特的字母或数字。

Pathname 路径名

通过文件系统到达一个文件的一条路径。为了确定该路径，由一列由/分开的目录名组成路径名。例如，路径名‘/usr/bin/lex’表示了一条路径，它从根目录开始，走至‘usr’目录，然后走至‘bin’目录，最后到达文件‘lex’。

Permissions 许可

文件的存取方式。参见access mode。

Permuted Index 置换索引

一种用来定位 UNIX 用户手册条文的索引关键字形式。

Pipe 管道

一个程序的标准输出和另一个程序的标准输入间的连接。例如,在ls命令和wc命令间创建一个管道的shell命令是“ls | wc”。

Pipe Fitting 管道安装

一种管道连接。

Pipeline 管道线

通过管道连接的一组命令。

Primary Store 主存储器

参见memory。

Printer 打印机

一种为了产生打印输出而连接到计算机系统上的设备。

printing Terminal 打印终端

使用打印装置作为输出的一种计算机终端。术语“打印机”通常指产生打印输出的设备,而术语“打印终端”通常指与计算机交互作用的打印设备。

process 进程

正在执行的程序。它在系统进程表中占一项。

process Identification Number 进程标识号

UNIX 系统核心赋给每个进程一个独一无二的进程标识号。当前进程的标识号可由ps命令印出。当你在后台运行一个程序时,shell印出它的进程标识号pid。

‘profile’

一个可以驻留在用户主目录中的shell命令文件。如果在主目录中有‘.profile’文件,那么,每当用户注册时,shell在执行终端来的命令之前,先执行这个文件中的命令。在‘.profile’中的命令通常用来初始化shell变量(例如查找路径),设置终端处理程序方式等等。

Program 程序

为了完成某个有用功能的一系列计算机指令。

Prompt 提示 (符)

由程序印出表示程序已经准备好从用户接受另一条命令的一个消息。shell提示符可以通过向变量 `$ps1` 赋一个值来加以改变。在很多系统中，UNIX系统shell的缺省提示符是 `^%` 或 `$`。

Quotation 引用

通过它元字符可以丧失其特殊含义的过程。

Read 读

获取信息（通常是从一个文件或 I/O 设备上）的动作。

Read Permission 读许可

允许用户执行一个程序，此程序从文件上读取数据。

Regular Expression 正则表达式

一个正则表达式指定了一组字符串。

Search String 检索串 (查找路径)

UNIX系统中shell保存一个检索串，它指导shell在一组目录中为打入的每条命令检索。检索串通常包括当前目录、`/bin` 目录以及 `/usr/bin` 目录。可以通过向变量 `$PATH` 赋新的值来更换检索串。

Secondary Store 辅助存储

参见 `mass storage devices`。

Shell

提供与UNIX操作系统接口的一种命令程序设计语言。作为一种命令语言，它交互地从用户那里接受命令并且安排所需的动作。作为一种程序设计语言，它包含控制流和串赋值的变量。实现shell的程序称 `/bin/sh`。

Shell Program shell程序

用shell程序设计语言编写的程序。虽然大多数shell程序存放在普通文件中，但是shell程序可以交互地编写和执行。

Software 软件

存放在计算机可存取介质中的程序和程序设计系统。

Source Code 源代码

程序的正文形式。编译程序把源代码变换成目标代码。

Single user 单用户

只能支持一个用户。在UNIX系统中，通常只在系统引导后进入单用户方式，这种方式通常用来进行文件系统修复和维护以及其它需要单独使用计算机才能完成的功能。即使在单用户方式下，UNIX系统也可以运行若干进程（多道程序设计）。

Special File 特别文件

在UNIX系统中，用来提供与I/O设备接口的文件。每个UNIX系统对于每一个与计算机连接的I/O设备至少有一个特别文件。可以利用存取普通文件的同样的技术来存取特别文件。特别文件通常驻留在‘/dev’目录中，有两种类型的特别文件，块特别文件针对能够支持文件系统的设备，而字符特别文件针对其它所有设备。

Standard Error 标准出错

许多程序放置出错消息的地方。

Standard I/O 标准输入/输出

从用户那里读取命令和数据、把消息写给用户以及写出出错消息的标准通道。在UNIX系统中，shell为每个程序准备了三种标准I/O连接，即标准输入、标准输出和标准出错。这些标准通道通常与用户终端相连接，虽然它们可以用重新定向重新指定。

Standard Input 标准输入

许多程序的输入源。参见input redirection。

Standard Output 标准输出

许多程序放置正文输出的地方。参见output redirection。

Subdirectory 子目录

处在文件系统层次结构中的一个目录的下层目录。例如，目录‘/usr/bin’是目录‘/usr’的子目录。

Subtree 子树

UNIX文件系统的一个分支。

Superuser 超级用户

存在于UNIX系统中的一个特殊的特权级，它使系统管理员能执行某些禁止普通用户执行的功能。超级用户不受通常的文件存取方式系统的约束。

Swapping 对换

有时主存里存放不下太多的进程，此时，一些进程就临时存放在海量存储设备上，这一过程就称为对换。把进程从主存换到海量存储设备上的动作称为换出，相反的过程称为换入。

Swap Space 对换空间

在海量存储设备上进程换出后的存储空间。

Syntax 语法

在语言中，控制语句构造的规则。在计算机中，术语“语法”用来描述在程序设计语言或命令语言中编写合法语句的规则。

System 系统

按一定方法统一起来的一个有组织的整体。

System Call 系统调用

活动进程提出要UNIX系统核心提供服务的请求。UNIX系统有完成I/O、控制、协调和创建进程，以及读取或设置系统各种状态的系统调用。

Tape Drive 磁带驱动器

一种海量存储设备，它用磁带来储存信息。

Terminal 终端

一种包含打字机键盘及显示设备或打印机的I/O设备。终端通常连接到计算机上，并且允许用户与计算机会话。

Text Editor 正文编辑程序

用于准备正文文件的一种通用程序。正文编辑程序使用户能够打入并且校正正文。正文编辑程序中有若干命令，这些命令包

括：在正文中定位特定的行或字，附加、删除、修改和打印正文行等等。

Text Formatter 正文格式加工程序

为出版或印刷准备正文的程序。格式加工完成调整间距、排齐表格、控制留空并且加标题等等工作。基本的UNIX系统正文格式加工程序是nroff, troff, eqn, neqn和tbl。

Text File 正文文件

只用ASCII字符组成的文件。

Time Sharing 分时

开发在若干用户间分享计算机资源的一种技术，这种技术给用户这个假象：只有他一个人在使用计算机。这是通过从一个任务很迅速地转换到另一个任务，从而看起来好象所有活动都在同时进行的方法来达到的。

The UNIX Programmer's Manual (UPM) UNIX程序员手册

也称UNIX用户手册。UPM介绍UNIX系统特定的功能。对于每一个UNIX系统版本都有相应的手册。

User 用户

使用UNIX系统的人。

‘/usr’ usr 目录

一个通用目录，它是包含UNIX系统大多数软件和资料的子树的根。

‘/usr/bin’ usr 目录下的bin子目录

一个通常用来存放不常使用的UNIX系统实用程序的目录。

Utility 实用程序

一种标准程序，它能提高完成一般数据处理任务的能力。

Variable 变量

其值允许修改的量。除赋值操作外，shell变量名前有美元号，一些标准的shell变量如\$PATH, \$PS1和\$TERM。

Word Processing 字处理

这是一个通用术语，它表示用计算机产生资料的正文数据处理。字处理系统的软件通常包括正文编辑程序和正文格式加工程序。

Write 写

把数据送到一个文件或 I / O 设备上的操作。

Write Permission 写许可

允许用户程序往文件上写数据。

Zeroeth Argument 第 0 个自变量

参见 command name。