普通高校本科计算机专业特色教材精选·计算机原理

实用UNIX教程

蒋砚军　高占春　编著

清华大学出版社

第 1 章 UNIX 简介

1.1 UNIX 的发展过程和标准化

1.1.1 UNIX 的发展过程

二十世纪六十年代末期，由美国的通用电气，麻省理工学院和AT&T的贝尔实验室联合创立了一个庞大的项目，就是开发一种功能强大的操作系统MULTICS(多路信息与计算系统MULTiplexed Information and Computing System)。该系统设计目标宏伟，功能强大，是一个分时和友好的作业环境。MULTICS的原始版本于1969年在GE645计算机上运行了，但由于效率低，功能庞杂，没能取得预期的成功。一些热衷于拼字游戏的批评者将MULTICS讥讽为“Many Unnecessarily Large Tables In Core Simultaneously”。但是，MULTICS的研究对操作系统的发展起到了很重要的作用。

1969年，从事MULTICS研究的贝尔实验室的Ken Thompson和他的同事Dennis Ritchie，在DEC的PDP-7型计算机上编写了一个简易的新型操作系统，是一个两用户的多任务操作系统。整个系统都用汇编语言编写的。从技术角度上，这个新的操作系统并没有引用更新的技术，主要对MULTICS的技术做了合理的裁减，不追求大而全，追求小而精。Brian Kernighan在1971年给这套系统取名时，针对于它的前身MULTICS，半开玩笑式的命名为UNICS(UNiplexed Information and Computing System)。UNICS的尾音-ICS与-IX类似，于是被改写成UNIX，名字就这样被确定下来。众所周知，UNIX在后来取得了巨大成功。

UNIX最初用汇编语言编制，在PDP-7上实现，在把UNIX移植到更新的计算机系统PDP-11上时遇到了一些麻烦。PDP-11不完全兼容PDP-7的指令，不象今天的Pentium完全兼容80386那样。这样，由于汇编语言不兼容，在汇编语言级上的移植变得比较麻烦。而且以汇编语言编写的程序由于可读性差，维护起来又很困难。因此，必须寻找一种高级语言，提高可读性和可移植性，同时又要求不丧失汇编语言高效的优点。C语言在这种需求下应运而生。Dennis Ritchie发明了C语言，并与Thompson用C语言改写了UNIX的源程序，为以后的开发和移植奠定了基础。DEC公司的PDP-11是七十年代小型机的主流机型，被广泛用在大学的实验室中。贝尔实验室把UNIX的C语言源程序代码和说明书赠送给美国的大学，使得UNIX成为许多大学操作系统课程的范例，为UNIX的普及和以后的成功奠定了基础。

C语言最初是为了能够在不同的硬件平台上移植UNIX而诞生的。UNIX和C语言都取得了巨大成功，UNIX自身就是C程序设计语言在系统软件领域的成功范例。反过来，UNIX

推动了C语言的应用和普及。C语言后续的衍生版本C++和Java都是现在非常活跃的编程语言。

1983年美国计算机协会的最高年度奖“图灵奖”，颁发给了UNIX和C语言的主要奠基人Ken Thompson 和 Dennis Rithie，以表彰他们在UNIX和C语言上的贡献。

随后，UNIX被移植到各种各样的计算机系统上，UNIX也经历了二三十年的发展，引进了许多新的技术，成为更加成熟的操作系统。

UNIX发展过程中，曾经有两大流派，一个是以UNIX的鼻祖AT&T最早于1983年发表的UNIX System V,这里的V是罗马数字5,最新的版本是Release 4,简写为SVR4。另一个是学术派的加州大学伯克利分校计算机系统研究小组(CSRG)发表的BSD UNIX(Berkeley Software Distributions),比较有影响的版本是4.3，记作4.3BSD。它在UNIX中引进了许多新技术，风靡全球的Internet上的TCP/IP协议，最早就是在BSD上开发的，它的后继版本就是现在的FreeBSD。早期的Internet上的大部分计算机都运行UNIX。两种流派的UNIX随着不断的发展，互相吸收对方的新技术，趋于统一到逐渐完备的POSIX标准。

UNIX后来成为了AT&T贝尔实验室操作系统产品的注册商标，所以，许多操作系统由于商业上的原因，在命名时不能够使用UNIX这个字眼，如IBM RISC/6000小型机上的AIX, Sun工作站上的SunOS以及后续版本Solaris，CDC小型机上的EP/IX，惠普Hewlett-Packard的HP-UX,等等。几乎所有的工作站平台上都运行UNIX操作系统。

UNIX成为一种软件商品，其源代码也就不再随意公开。早期的UNIX用作许多大学操作系统课程的范例，澳大利亚的John Lions教授还为UNIX的源程序逐行写了注释, 这是著名的《莱昂氏UNIX源代码分析》一书。后来由于其源代码不公开，为了教学上的需要,Andrew S. Tanenbaum教授在UNIX的基础上开发了教学版的UNIX, 取名MINIX,意为Mini-Unix。它的功能简单，模块清晰，结构易懂，可以运行在流行的Intel x86系列的PC上，非常适于操作系统课程的教学使用。1991年，芬兰学生Linus Benedict Torvalds, 对于MINIX系统十分熟悉，在PC上独立编写了基于Intel 80386体系结构的UNIX系统，相对简单易懂的纯教学工具MINIX有更好的性能，命名为freax。作者Linus承认这套系统有点象MINIX，但是完全没有用到MINIX的源代码。Linus将这套系统的所有源程序免费公布在芬兰最大的FTP站点，由于这套系统是“Linus的MINIX”,因此，就建立了一个名为Linux的目录存放这些代码，并通过Internet请世界各地的软件爱好者参加Linux的开发工作。最终Linux成了这套系统的名字。世界各地的软件爱好者都可以通过Internet免费获得Linux的全部源代码和二进制发行版本。现在，LINUX更是发展迅速。从比PC还要简单的多的小型嵌入式环境,一直到大型的服务器，都有Linux操作系统在运行。

1.1.2 什么是 UNIX

狭义的说，UNIX指的是一个多用户多任务分时操作系统内核（Kernel）。内核的功能是用于控制并管理计算机的资源，使多个用户可以同时访问这些资源。内核负责进程的创建、控制、调度，为进程分配内存和外设，提供文件系统的管理功能。广义的说，UNIX

不仅指系统内核，它还是一个程序设计环境，为程序员提供了丰富的软件开发工具，包括许多UNIX的实用命令，编辑器，编译程序，调试工具，数据复制和备份，打印，数据库等等。

1.1.3 UNIX 的标准化

由于UNIX被移植到各种各样的硬件平台上,即使同一种硬件平台上,也会有许多种不同的UNIX操作系统软件产品，x86系统上就有SCO UNIX，Solaris，Linux等。UNIX的版本种类越来越多，达到几十种，而且出自不同厂家。许多厂商在他们的UNIX中扩充了新的内容，这使得各种UNIX之间的可移植性，尤其是用户编写的源程序和命令在各种UNIX之间的可移植性越来越低。这样，就要求制定一个UNIX标准以供各厂商遵守。于是，就有一些国际性组织制订了一些UNIX标准。这些标准也常常出现在UNIX产品的使用手册中，供用户进行可移植性程序设计时作为参考。这些标准都是仅仅定义系统的界面，不关心系统内部的实现。主要有以下几种：

(1) 1986年，IEEE制定了IEEE P1003标准，这套标准被称为POSIX(Potable Operating System Interface)。POSIX定义了一整套的作业接口，包括系统调用，库函数，公共命令等等。这套标准非常活跃，一直在不断演化和完备。

(2) 另一个促使UNIX标准化的组织是X/Open。X/Open最早在1984年由几家欧洲计算机公司组成。1989年发表了X/Open Portability Guide第3版，称作XPG3。

(3) AT&T制订的UNIX标准SVID(System V Interface Definition)，与POSIX兼容。

(4) 1988年，IBM，HP，DEC等几家计算机厂商成立了OSF: Open Software Foundation，1990年OSF发表了OSF/1。

(5) FIPS，是联邦信息处理标准的缩写(Federal Information Processing Standard),由美国政府出版，用于指导美国政府的计算机系统采购。1989年出版FIPS 151-1，规定了必须支持的POSIX可选功能。因此，它实质上是个更严格的POSIX标准。

1.2 系统的登录与退出

1.2.1 UNIX 的主机和终端

UNIX系统被设计的同时可以连接多台终端。在PC中,可以直接利用PC内置的RS-232串行通信口连接终端。当需要安装多台终端时,需要在PC内总线扩展槽内安装多用户卡,一个多用户卡可以引出多个(如八个或六十四个)串行通信口,使得PC可以连接多台终端。传统的终端(或者叫做“哑终端”)主要由一个键盘和显示器构成,并有一个串行通信口与主机相连,不含有任何磁盘存储设备。结构简单,通常仅仅是一台PC价格的1/5到1/10。

除了直接使用终端外,也可以使用另外一台PC通过运行终端仿真软件来仿真终端。例如: DOS操作系统下的CrossTalk,PROCOM软件就可使一台PC通过串行通信口成为UNIX主机的一台仿真终端。在WINDOWS系列操作系统中,也含有终端仿真程序“超级终端”。

另外,还可以通过网络,例如UNIX主机与多台PC通过网络相连,运行TCP/IP网络协议,那么,PC可以运行TCP/IP协议之上的网络终端仿真程序,成为UNIX的一个网络虚拟终端,在大部分系统中使用命令TELNET。

当多个终端，无论是一个真正意义上的终端，或者仿真终端，还是网络虚拟终端，同时登录到同一台UNIX系统时，UNIX就可以容纳多个用户同时上机。这也是UNIX作为多用户多任务分时操作的操作系统设计的初衷和基本功能。

在上机过程中，终端只负责把用户输入的按键信息送到UNIX主机,并把主机发来的信息在屏幕上显示。所有其它处理均在UNIX主机上完成，数据及程序全部存放在UNIX主机的硬盘等存储设备上,所有程序的运行也都由UNIX主机内的CPU占用UNIX主机的内存来完成。仿真终端和网络虚拟终端，也只是模仿一个真正的终端只完成这两项任务。

例如，在上机过程中，用户在终端上按下d键，那么，终端将d的一字节ASCII码(十六进制64，附录中列出了ASCII表)送到UNIX主机，UNIX主机回送一个字节的数据，这便是操作员能够看到的回显d，如果UNIX主机不回送，就不能看到回显。UNIX主机通过向终端发送数据的方式，将处理结果发送到终端，就可以看到命令执行结果。

主机和终端除了交换这些简单的可见的字符信息以外，还有一些控制信息，叫做“终端转义序列”。例如：主机向终端连续发送四个字节的序列(用十六进制表示) 1b 5b 32 4a，第一字节是键盘左上角的按键Esc的代码，终端转义序列都以这个控制字符打头。后续的分别是[2J三个字符。常将上述的序列记作Esc[2J。当终端收到这四个字节的时候，不再将左中括号，数字2和大写字母J显示在终端屏幕上，而是在收到Esc之后，将这四个字节作为一个整体解析为一个终端转义序列，实现控制功能。这一终端转义序列的功能是将屏幕清为空白，清屏。在这四字节序列之后的信息会照常在终端屏幕上显示。再如，4字节终端转义序列Esc[8A的功能是将光标光标上移8行，7字节序列Esc[16,8H将光标移动到16行8列。

终端支持很多这样的转义序列，完成光标移动，清屏，设置字符的颜色，闪烁，下划线，字体大小等等，甚至可以绘图，可以控制连接在终端上的打印机，刷卡器等等。不同的终端类型，会支持不同的转义序列，对转义序列有不同的解析。因此，必要的时候还需要设置终端的“终端类型”，常见的终端类型有ansi，vt100，vt220等等。UNIX了解了终端类型后，才能够根据终端类型发送相应的转义序列。在仿真终端和虚拟终端软件中，也有这样的终端类型设置。终端类型的设置，必须和UNIX的设置兼容，否则，一些全屏幕操作的软件，就无法正常工作。

主机和终端之间的流量控制功能是必要的。例如：主机送来数据，终端需要把这些数据在终端的打印机上打印出来，但是打印的速度没有主机发送数据的速度快。或者，主机送来的显示内容，操作员需要暂停显示，进行仔细分析。这样就需要一种机制，可以控制主机方向来的数据流量。流控方式有硬件方式和软件方式。硬件方式利用RS-232

接口的RTS和CTS信号线，软件方式利用Xon和Xoff两个用于流量控制的控制字符。大部分终端都设置Xon/Xoff软件流控方式。当终端希望主机暂停发送数据时，终端向主机发送Xoff字符，当终端希望主机继续发送数据时，终端向主机发送Xon字符。这两个控制字符分别被定义为ASCII码的17和19(十进制表示)，用户键盘按键Ctrl-S和Ctrl-Q分别产生这两个字符。操作员经常使用的按下Ctrl-S键暂停显示，按Ctrl-Q键继续显示。依赖于终端的设定，按下Ctrl-S后暂停显示，有的终端按下任意键继续显示；有的终端则要求必须按下Ctrl-Q后才继续显示，否则其它按键都不能使得终端继续显示。

1.2.2 登录

用户在上机前应向系统管理员申请一个帐号，也叫登录名或用户名,由系统管理员在系统中创建用户。随后，用户就可以使用这一登录名，从任意终端上登录进入系统。

UNIX系统有一个超级用户，用户名为root,在系统安装时创建，作为系统管理员。随后创建的用户，是普通用户。超级用户负责系统的维护，包括创建和删除用户账号等等。在使用UNIX时，应当尽量以普通用户的身份登录进入系统。只有在进行系统管理操作时才以超级用户root的身份作为系统管理员登录进入系统。以root用户身份操作时，要特别小心，因为他不受UNIX文件权限的制约，可以随意修改和删除文件，而普通用户不允许。root用户的误操作，比如：删除了重要的文件，可能会给系统带来严重的后果。

当终端上出现登录提示符`login`：后，键入登录名，然后系统给出提示`password:`，提示输入口令。为了保密起见，口令在键入过程中不回显。对于那些不会键盘盲打的使用者，键入用户名，应当确认`password`：提示符出现之后，再输入口令。如果用户在输入用户名之后未出现`password:`提示符之前迅速输入口令，口令会回显在屏幕上，从而可能会被其他人看到而泄密。

只有在上述两项都答对了之后，才登录成功。登录成功之后，就出现了Shell提示符，如同DOS中的C>。

有些UNIX系统登录成功后会询问用户的终端类型，这时输入的终端类型，UNIX主机将按照这个终端类型定义的转义序列发送控制信息，所以必须和所使用的终端类型兼容，否则，在使用一些全屏幕操作的命令和其他一些命令时，会遇到麻烦。

Shell的种类由系统管理员创建用户时选定的。常见Shell的提示符：

`$` Bourne Shell (`/bin/sh`)
`%` C Shell (`/bin/csh`)
`$` Korn Shell (`/bin/ksh`)
`#` 当前用户为超级用户root

在System V中，默认的shell为`/bin/sh`。在其它的UNIX中，系统根据自身情况设定。同DOS一样，shell提示符可以根据用户的需要和喜好，通过shell命令改变。

1.2.3 退出

在Shell提示符下按Ctrl-D键，或者键入logout命令，或者键入exit命令，退出本次登录。退出成功的标志是UNIX再次给出login:提示符。

使用RS-232串行通信口连接UNIX的传统终端和仿真终端的用户，有时主机和多个终端都安置在距离几米到几十米的同个房间或相近房间里，直接通过电缆连接主机和终端，而不是通过调制解调器连接，这种连接方法很多。应当注意的是，在某些设置情况下，终端直接关闭电源，UNIX一侧并没有自动完成退出登录。UNIX会以为终端只是暂时没有按键信息。但是，当重新打开终端电源或者重新启动仿真终端软件的时候，UNIX会在以前已经登录的基础上继续工作，UNIX能感受到的是该终端在长时间没有按键信息之后，现在又继续工作。这样，一个用户未退出登录，下个用户会直接进入到前个用户已登录的状态，会对信息安全造成威胁。使用DOS/Windows仿真终端的用户，DOS或者Windows的突然崩溃，或者正常关机，只要没有正常退出登录，都会出现类似直接关掉终端电源的后果。因此，使用RS-232通信的传统终端和仿真终端的用户要特别注意，退出成功的标志是UNIX再次给出login:提示符。这时，才可以安全离去。

使用TELNET的网络虚拟终端，由于实现的机制不同，不会出现这样的情况。

1.2.4 关机

UNIX主机的开机和关机工作由系统管理员完成,普通用户不允许关机。在关掉UNIX主机电源之前必须先执行关机命令shutdown，否则，可能会导致系统中文件数据的丢失，甚至导致系统瘫痪，下次无法启动系统。例如，在UNIX系统上运行一个程序，向磁盘写20块(每块512字节)数据，每写一块前都要查找文件的不同位置，只用了300毫秒时间。而磁盘的实际性能是平均存取时间为75毫秒，那么，20块的写操作应需要1.5秒,但是，只用了300毫秒。这是因为write系统调用只把数据传送到内核的文件高速缓冲区，然后立即返回，并不等待实际的磁盘操作的结束。关机前shutdown命令的其中一项功能是将高速缓冲区数据真正写到磁盘上。

1.3 使用系统命令

在Shell提示符下就可以键入UNIX命令，就如同在DOS系统的C>提示符下可以输入DOS的命令一样。与DOS/Windows不同的是在UNIX系统中组成命令的英文字母大小写有区别。在UNIX中，最常用的命令一般由两到三个字母构成。在这里介绍几个简单的命令。

1. 3.1 man:查阅联机手册

几乎所有的UNIX系统都提供联机手册(Online manual)。内容包括:各种命令的使用说明书，系统调用的使用手册，C语言和其它语言的库函数使用手册，系统配置文件的格式，等等。

命令：man

命令名man,取自manual的前三个字母。

用法：

man 名字

man 章节号 名字

一般UNIX系统章节编号为：

1 用户命令

2 系统调用

3 库函数

不同UNIX系统中联机手册的章节编号可能不同，可以使用命令

man man

查命令man的使用手册，在这一手册中，会给出联机手册的章节编号方案。例如：SCO UNIX是一种在Intel x86体系结构计算机上广泛使用的商用UNIX版本。在SCO UNIX系统中，联机手册的章节编号方案如下：

```
C     Commands
S     Development System routines and libraries
ADM   System administration
F     File formats
K     Device driver routines and libraries
```

如：man man 查命令man的使用手册。

man ls 查命令ls的使用手册。

man strcpy 查strcpy函数的使用手册。

man sleep 由于有一个命令名为sleep,也有一个函数调用名为sleep,在这种情况下，man可能只列出命令sleep的使用说明，不列出函数调用sleep的使用说明，有的系统会同时列出两者。如果要查阅函数调用sleep的使用说明，则可以在man命令中指定章节号，在SCO UNIX系统中的命令为：

man S sleep

该命令列出的内容如下：

```
sleep(S)    6 January 1993       sleep(S)

Name

   sleep - suspend execution for interval
```

```
Syntax

   cc . . .  -lc

   unsigned sleep (seconds)
   unsigned seconds;

Description

   The current process is suspended from execution for the number of seconds
specified by the argument. The actual suspension time may be less than that
requested for two reasons: (1) Because scheduled wakeups occur at fixed
1-second intervals, (on the second, according to an internal clock) and
(2) because any caught signal terminates the sleep following execution of
that signal's catching routine.  Also, the suspension time may be longer
than requested by an arbitrary amount due to the scheduling of other activity
in the system.  The value returned by sleep is the ``unslept'' amount (the
requested time minus the time actually slept) in case the caller had an
alarm set to go off earlier than the end of the requested sleep time, or
premature arousal due to another caught signal.
   The routine is implemented by setting an alarm signal and pausing until
it (or some other signal) occurs.  The previous state of the alarm signal
is saved and restored.  The calling program may have set up an alarm signal
before calling sleep.  If the sleep time exceeds the time till such alarm
signal, the process sleeps only until the alarm signal would have occurred.
The caller's alarm catch routine is executed just before the sleep routine
returns. But if the sleep time is less than the time till such alarm, the
prior alarm time is reset to go off at the same time it would have without
the intervening sleep.

See also

   alarm(S), pause(S), signal(S)

Standards conformance

   sleep is conformant with:
   AT&T SVID Issue 2;
   X/Open Portability Guide, Issue 3, 1989;
   IEEE POSIX Std 1003.1-1990 System Application Program Interface (API)
[C Language] (ISO/IEC 9945-1);
   and NIST FIPS 151-1.
```

在man列出的说明书中，首先列出函数的基本功能和语法格式。对于C语言的调用，列出需要#include的头文件名，连接时需要的函数库。

说明书中列出了sleep调用的基本功能，将程序的执行暂停若干秒，详述了实际暂停时间和函数参数指定时间会有出入的原因。说明书中还说明了这一函数是用alarm信号和pause实现。

说明书的“See also”段中，会含有与此命令和调用有关的其它项目的名字和章节号。例如：上例中列出与函数sleep有关的内容有S章的alarm和pause以及signal。

最后的“Standards conformance”段列出了该函数调用所遵循的UNIX标准。

1.3.2 获取系统情况

1. date: 读取系统日期和时间

该命令列出系统当前的日期和时间值。例如：

```
$ date
Wed May 19 13:54:55 BEIJING 2004
```

这里的$号是shell的提示符，date是用户键入的内容。为了区分用户键入的内容和系统给出的显示，所有用户手工键入的内容都是用黑体字，并且加下划线，以示区别。本书中的shell提示符，都用$号。

date命令还可以根据用户的需求，打印出符合用户需求的时间和日期格式。如：

```
$ date "+%Y.%m.%d  %H:%M:%S   Day %j"
2004.05.19  13:55:47   Day 140
```

这种方式打印出当前的日期和时间，140指的是今天是今年的第140天。date的格式控制字符串，第一个字母必须为+号，%Y代表年号，%m代表月份，%M代表分钟。没必要一项项记住这些控制符。UNIX的命令往往有很多选项和复杂的功能，date命令的完整手册至少需要密密麻麻地写满5张A4纸。需要的时候可以通过man date查阅联机手册。

2. who: 查看已登录系统的用户

命令who列出当前已登录进入系统的所有用户，输出结果如下所示：

```
$ who
root    tty1    Jul 5   10:30am
liang   tty2    Jul 5   10:18am
jiang   tty3    Jul 5   10:19am
song    tty4    Jul 5   10:20am
fang    tty5    Jul 5   10:20am
$ who am i
jiang   tty3    Jul 5  10:19am
$ whoami
jiang
```

其中,第一列列出用户名,第二列列出该用户登录系统所使用的终端设备的设备文件名。在UNIX系统中设备和文件统一管理，每一设备都在文件系统中有一个文件名，同普通磁盘文件所不同的是，文件类型属于特殊文件。一般都将设备文件放在目录/dev下。终端设备文件的名字一般有tty前缀,tty是teletype(电传打字机)的缩写，网络虚拟终端有的用ttyp的前缀，p是pseudo的缩写，也有用vty前缀的。终端设备文件类似于DOS中的设备文

件CON,但UNIX中的终端设备文件取名不像DOS那样必须强制为CON，而且DOS文件系统中不存在一个名为CON的文件名。UNIX的文件系统中的确有相应命名的文件，用户还可以使用命令mknod根据自己的喜好改成别的名字。

使用命令who am i，可以列出当前终端上的登录用户，登录终端和登录时间。命令whoami和who am i命令不同的是，前者仅列出当前终端上登录的用户名。

使用命令who可以得知某用户的登录时间。对整个系统来说，命令uptime可以得知UNIX系统自启动后到现在的运行时间(系统的年龄)，当前登录入系统的用户数，以及近期1分钟，5分钟，15分钟内系统CPU的负载情况。

与who命令类似的有w命令(Who & What)。w命令列出终端的空闲时间(IDLE)；JCPU是终端上正在运行的作业占用的CPU时间，包括前台程序和后台程序；PCPU是终端上正在运行的前台程序占用的CPU时间；WHAT列出终端上的用户正在执行什么命令。

```
$ uptime
10:35:22  up 18 min,  5 users,  load average: 0.55, 0.73, 0.43
$ w
10:35:26  up 18 min,  5 users,  load average: 0.51, 0.72, 0.43
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
root     tty1     -                10:30am  0.00s  0.15s  0.01s  w
liang    tty2     -                10:18am  1:43   0.05s  0.01s  ftp
jiang    tty3     -                10:19am 28.00s  0.08s  0.08s  -bash
song     tty4     -                10:20am  1:47   0.04s  0.00s  telnet
fang     tty5     -                10:20am 23.00s  0.11s  0.08s  vim .profile
```

3. tty: 打印出当前终端的设备文件名

使用命令tty可以打印出当前终端的设备文件名。

```
$ tty
/dev/tty1
```

1.3.3 passwd:更换用户口令

对于普通用户可以使用passwd命令更改自己的登录口令，在更改前系统会先验证你原来的口令。与一般的信息系统一样，输入口令时，没有回显，输入新口令时，要求输入两遍，两遍完全相同时才确认。在口令字符中，英文字母的大小写有区别。

对于超级用户，也可以使用passwd命令修改自己的口令，另外，root用户还可以使用类似下面的命令：

```
passwd liu
```

根据系统的提示，可以将用户liu的口令强迫设置为某一已知口令，但是，超级用户root无法读取其他用户的当前口令。当一个普通用户忘记了口令时，可以请求超级用户执行这一操作，然后使用一个设定口令登录后，再修改口令。

在修改超级用户root的口令时要特别注意，超级用户的口令丢失会很麻烦，有的系统会需要一套很复杂的操作，才能重新获取一个新的有效口令，有的系统甚至会永远无法再次以超级用户的身份登录进入系统。

无论是UNIX的登录用户，或者，其它的信息系统，很多时候要求用户设置自己的口令。用户口令的设置，应当本着对自己容易记忆，对其他人不容易记忆的原则，以免被别人猜到。对于那些打字不熟练的用户来说，口令组成的各字符的按键还应当尽量零散分布在键盘的不同区域。输入时，左右手的不同手指迅速地交替击键，以免被别人通过观察按键位置窃取口令。例如：选择akigo2就比qwerty好些。口令的输入应当尽量做到熟练。在为系统选择口令时选用仅含字母和数字的组合，以免遇到不必要的麻烦。

1.3.4 与其他用户通信

1. write 命令

使用这一命令可以直接给登录在系统的其它用户发消息。步骤：

用who命令确定接收消息的用户现在是否已注册。如果用户未注册则无法使用write命令。然后输入命令：write 用户名，如：

```
write liang
```

键入消息,多行信息时用回车分开，结束消息，用Ctrl-D，Ctrl-D是“文件结束”键。如果对方处于拒绝消息状态，则通信失败。对方收到的消息会直接显示在终端屏幕上。

write命令也可以使用输入重定向，格式为：write 用户名 < 文件名，如：

```
write liang<msg.liang
```

2. talk 命令

可以用于网络环境，同一UNIX主机内两用户可以talk，用网络连在一起的两台UNIX主机上的用户间也可以talk。talk是个很简易的两用户对话程序

设用户liu希望与同一台UNIX主机上的用户wang通信，talk的通信步骤为，通信发起者liu执行下面命令：

```
talk wang
```

那么，在用户wang的屏幕上出现一个消息，消息中含有应答talk应使用的命令。接收者wang执行下列命令：

```
talk liu
```

然后liu和wang，从键盘输入通信的内容。屏幕被分成上下两部分，两用户可以交互式键盘会话。对话结束，按Ctrl-C键或Ctrl-D键，终止程序talk的执行。

如果对方此时没有登录入系统或者虽然已经登录入系统但是处于消息拒绝状态，则通信失败。与其它UNIX系统上的用户进行键盘会话的talk命令中还应含有对方主机名，如：与cdc.xynet.edu.cn主机上的用户wang通信：

```
talk wang@cdc.xynet.edu.cn
```

3. mesg 命令

设置当前终端的消息接受状态。

不希望别人打扰，使用命令mesg n。

允许接受消息，使用命令mesg y。

列出当前状态，使用命令mesg。

当设置了mesg n后，超级用户write的信息，仍然会在终端上提示。这是超级用户的特权。

4. wall 命令

用于向所有用户广播消息(write to all)。例如，超级用户会使用这个命令，通知所有在线用户十五分钟之后要关机。

命令的用法与write类似，从键盘输入消息，结束时按Ctrl-D键。或者，使用类似wall < info.text的重定向方式，从文件中获取消息。

登录用户设置的mesg n，不能阻止超级用户或者其他用户通过wall命令发送来的消息显示在终端上。有些系统中，系统管理员会将wall的执行权限设置为只允许超级用户使用。

1.3.5 与其他主机通信

1. telnet:远程登录

这一命令是个很常用的命令，在DOS/WINDOWS中也有功能相似的程序版本。Windows XP系统的命令行方式TELNET程序的使用方法和UNIX的命令用法类似。使用这一命令可以登录到远程的其它UNIX主机上，作为远程计算机的一个网络虚拟终端上机操作。telnet命令给定参数可以是远程计算机的IP地址，或者，是一个名字。远程的计算机，必须已启动telnet服务。例如：

```
telnet 202.172.122.135
```

或者：

```
telnet cdc.xynet.edu.cn
```

【例 1-1】使用 telnet 登录到远程一台 AIX 系统的例子。

括号内的楷体字内容是一些注释。

```
$ (在 shell 提示符下输入命令)
$ telnet 202.172.122.135
Trying...
Connected to 202.172.122.135.
Escape character is '^]'.

AIX Version 5
(C) Copyrights by IBM and by others 1982, 2000.
login: jiang (出现登录提示符，输于远程计算机用户的用户名/口令)
jiang's Password:***
*************************************************************
*  Welcome to AIX Version 5.1!
*  Please see the README file in /usr/lpp/bos for information
*  pertinent to this release of the AIX Operating System.
*************************************************************
Last unsuccessful login: Wed Jun 25 12:38:45 2003 on /dev/pts/0 from
61.149.71.89
Last login: Wed May 19 08:04:53 2004 on /dev/pts/0 from 61.149.159.49

$
(登录到了远程的计算机，这里的 shell 提示符$是远程计算机给出的，在此输入的命令，在远程
计算机上执行)
$ tty
/dev/pts/1
$ who am i
jiang     pts/1      May 19 08:45
$
$ ^](在此按下转义字符 Ctrl-]键,会出现 telnet 的提示符，开始和本地 telnet 仿真程序会
话)
telnet> help
Commands may be abbreviated.  Commands are:
close       close current connection
displ       display operating parameters
ay
emula       emulate a vt100 or 3270 terminal
te
mode        try to enter line-by-line or character-at-a-time mode
open        connect to a site
quit        exit telnet
send        Transmit special characters ('send ?' for more)
set         set operating parameters ('set ?' for more)
statu       print status information
s
toggl       toggle operating parameters ('toggle ?' for more)
e
z           suspend telnet
?           print help information
telnet> (在此直接按回车，退出和 telnet 程序的会话，继续与远程计算机会话)
```

```
$ tty
/dev/pts/1
$
$ exit
Connection closed.
```

从键入telnet命令，到退出远程登录期间，根据前边的提示：

```
Escape character is '^]'.
```

随时可以键入Ctrl-]键，调出终端仿真程序telnet的命令提示符，可以使用一些命令控制终端仿真程序的运行。这些命令随着系统的不同，差别也很大。有的系统会在这里允许用户设置仿真终端的终端类型；允许用户设置一个捕获文件的文件名，从此以后，与远端计算机的所有会话内容除了在屏幕上显示之外，还存入到一个文本文件中，以备事后查阅和分析。大部分的终端仿真程序，包括Windows系统的“超级终端”（可用于RS-232异步口或者TELNET协议），都有这种捕获功能。

连接到远程计算机，出现login:提示符，如果无法输入正确的用户名和口令，目标系统会反复询问用户名和口令，也就无法成功登录到shell中以输入exit命令的方式中止这次登录。或者，登录成功后，在远端主机上运行的程序出错或者其他原因，无法回到shell提示符状态。这样，就可以按下Ctrl-]键调出终端仿真程序telnet的命令提示符telnet>，输入close关闭这个连接,结束远程登录。

2. ftp:文件传送与文本文件的格式

使用ftp命令可以直接将文件传到其它计算机上，或者，从其它计算机上获取文件。这是个常用命令，在DOS/WINDOWS中也有功能相似的命令行程序。和telnet命令一样，远程的计算机，必须已启动ftp服务，才可以使用ftp命令与对方计算机传送文件。一般UNIX都默认有这项服务，Windows系统常常需要经过特意的设置，或者安装专门的ftp的服务软件，才能够支持ftp的服务。

【例 1-2】ftp 会话的过程。

```
$ ftp 202.162.120.115（在 shell 中输入 ftp 命令）.
Connected to 202.162.120.115.
220 RISC6000 FTP server (Version wu-2.6.1(1) Thu Jul 20 19:10:14 DFT 2000)
ready.
Name (jiang): jiang(输入远程计算机的用户名和口令)
331 Password required for jiang.
Password:***
230 User jiang logged in.
ftp> dir（在 ftp>提示符下，可以输入 ftp 的命令,dir 用于察看远端计算机的目录）
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 522776
-rw---       1                    Aug 30         .sh_history
-r--        100       328         2003
```

```
-rw-r-      1                    Sep  9         paper01.txt
-r--        100      181248      2003
-rw-r-      1                    Sep 28         paper02.txt
-r--        100      996987      2003
-rw-r-      1                    Mar 13         linux-2.4.ta
-r--        100      277032      2003           r
                     98
-rw-r-      1                    Apr  9         linux.tar.Z
-r--        100      276992      02:13
                     83
-rw-r-      1                    May 13         books.tar.Z
-r--        100      207575      22:55
                     995
-rw-r-      1                    Sep  9         src.tar.Z
-r--        100      992094      2003
226 Transfer complete.
ftp> help（输入 help 命令，打印出 ftp 的所有命令列表）
Commands may be abbreviated.  Commands are:
!                   image               recv
$                   lcd                  reget
account             local               reinitialize
                ...(省略)
ftp> get src.tar.Z （get 命令将远程计算机的文件传到本地）
200 PORT command successful.
150 Opening ASCII mode data connection for src.tar.Z  (992094 bytes).
226 Transfer complete.
995934 bytes received in 0.4596 seconds (2116 Kbytes/s)
local: src.tar.Z  remote: src.tar.Z
ftp>
ftp> put myfile（put 命令将本地计算机的文件传到远程主机）
200 PORT command successful.
150 Opening ASCII mode data connection for myfile.
226 Transfer complete.
995934 bytes sent in 0.2397 seconds (4057 Kbytes/s)
local: myfile remote: myfile
ftp> bye（bye 命令退出 ftp）
221-You have transferred 1988028 bytes in 2 files.
221-Total traffic for this session was 1989427 bytes in 3 transfers.
221 Goodbye.
```

有些ftp服务器，允许匿名登录。匿名登录时，使用用户名anonymous,口令输入时应当输入自己的E-mail地址，有的系统允许随意输入，或者不问口令。

连接到远程计算机之后，就可以传送文件。如果操作员在一定时间内(例如：15分钟)没有任何操作，那么远程计算机会自动关闭连接。

在ftp>提示符下，可以使用FTP的命令。输入help可以打印出ftp支持的命令列表。表1-1列出一些常用命令。

表 1-1 FTP 常用命令

命令	功能描述

dir	列出远端计算机的目录。
get *filename*	将远端计算机文件传输到本地。
put *filename*	将本地计算机文件传输到远程计算机。
cd *dir*	更改远程计算机的当前目录。
lcd *dir*	更改本地计算机的当前目录。
close	关闭一个FTP连接。
open *host*	建立一个新的FTP连接，指定IP地址或主机名。
user *username*	在已建立的FTP连接上登录到其他用户。
mkdir *dir*	创建新目录。
rmdir *dir*	删除目录。
delete *filename*	删除文件。
rename *old new*	文件改名。
ascii	设置ASCII码方式传送文件（默认方式）。
binary	设置二进制方式传送文件。
hash	文件传送过程中，打印#号标志文件传送的进度。每打印一个#号，代表1K或者2K数据传送完成。当网络速度较慢时，操作员可以实时了解传送的进度。
bye	退出ftp。

使用FTP在UNIX和Windows之间传送文件时，会有这样的现象，将一个Windows下3722字节的文件report.txt传送到UNIX后，文件大小变为3628字节。文件的大小发生了变化，打印出UNIX下的文件，又发现数据没有丢失。反过来，将UNIX文件传送到Windows时，文件大小又会增加。

使用FTP传送文件时，应当特别注意“传送模式”。在前面的FTP会话举例中，执行put myfile命令，系统会给出下列的提示：

```
150 Opening ASCII mode data connection for myfile.
```

在这里，文件传送使用ASCII方式，这也是默认的传输模式。使用FTP在UNIX和Windows之间ASCII码方式传送文件时，FTP会自动在UNIX和Windows之间进行文本文件的格式转换。UNIX和Windows的文本文件的格式有些不同。例如：设想一个很小的文件mini.txt，文件只有两行，第一行为ab,第二行为xyz，那么，在Windows中，文件的大小为9字节，这9字节的十六进制代码分别是：

```
61 62 0d 0a 78 79 7a 0d 0a
```

在Unix中，文件的大小为7字节，这7字节的十六进制代码分别是：

```
61 62 0a 78 79 7a 0a
```

其中，61,62分别是字符a和b的ASCII码，78,79,7a分别是xyz的ASCII码，0d是“回车(CR)”的ASCII码，0a是“换行(LF)”的ASCII码。从这里的比较中可以看出两种不同的操作系统文本文件存储的格式不同。UNIX的文本文件存储时，每行结尾处仅存储换行字符， 而Windows的文本文件存储时，每行结尾处存储回车和换行两个字符。

使用ASCII方式，在UNIX和Windows之间传送文件时，ftp会自动在UNIX和Windows之间进行文本文件的格式转换，接收时ftp检索出文件中的CR-LF序列，替换为LF;发送

时，把LF替换为CR-LF序列。所以导致传输前后文件的大小不同。这种自动转换，有时会在不期望的时候发生。例如：从Windows传送一个805603字节的JPG照片文件，操作员并没有特意设置传送模式，系统使用默认传输模式,就是ASCII方式，收到的文件变成了805594字节，这显然会是错误的。所以要特别注意。

在ftp命令提示符下使用binary命令，就可以设置为二进制模式，或者叫IMAGE模式。设置了binary之后，随后在该FTP连接上的文件传送都使用BINARY模式，不再进行转换。如果要期望再使用ASCII方式，那么键入命令ascii。

【例 1-3】ftp 使用二进制方式传输文件的例子。

```
ftp> binary
200 Type set to I.
ftp> get A301211.JPG
200 PORT command successful.
150 Opening BINARY mode data connection for A301211.JPG (805603 bytes).
226 Transfer complete.
805603 bytes received in 0.2035 seconds (3865 Kbytes/s)
local: A301211.JPG remote: A301211.JPG
```

在UNIX/UNIX之间，或者，Windows/Windows之间传送时，无论使用什么模式，都不会发生转换，所以不会有什么问题。Unix/Windows之间传送文件时，必要的时候检查文件大小，以防止不期望的转换发生。

在Windows中，使用图形界面的IE,输入类似下边的URL,可以登录到UNIX的FTP服务。

ftp://jiang:akigoo2@201.203.112.108

其中，jiang为用户名，akigoo2为该用户的口令，201.203.112.108是远端计算机的地址，也可以使用类似cdc.xynet.edu.cn主机名字。或者,使用匿名ftp，直接输入下边的网址：ftp://201.203.112.108。

在使用文件图标拖拽的方式上传/下传文件时，默认使用BINARY方式，有些FTP工具，如FlashGet，操作员可以自行设置，根据文件名的后缀，自动选择ASCII方式或者BINARY方式。

1.3.6 几个实用工具

1. cal: 打印日历

命令cal的用法为：

cal [[*month*] *year*]

【例 1-4】cal 命令用法举例。

cal	打印当前月份的日历。
cal 2005	打印2005年的日历。
cal 10 2006	打印 2006年10月份的日历。

根据命令的语法，要想指定月份，必须指定年份。命令cal 12，系统会以为要打印出公元12年全年的日历，而不是打印今年12月的日历。

```
$ cal
        May 2004
Sun Mon Tue Wed Thu Fri Sat
                      1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19  20  21  22
23  24  25  26  27  28  29
30  31
$ cal 2005
                                2005

          January                        February
Sun Mon Tue Wed Thu Fri Sat  Sun Mon Tue Wed Thu Fri Sat
                      1                1   2   3   4   5
 2   3   4   5   6   7   8    6   7   8   9  10  11  12
 9  10  11  12  13  14  15   13  14  15  16  17  18  19
16  17  18  19  20  21  22   20  21  22  23  24  25  26
23  24  25  26  27  28  29   27  28
30  31                    .
                          .
                          .
          November                       December
Sun Mon Tue Wed Thu Fri Sat  Sun Mon Tue Wed Thu Fri Sat
          1   2   3   4   5                    1   2   3
 6   7   8   9  10  11  12    4   5   6   7   8   9  10
13  14  15  16  17  18  19   11  12  13  14  15  16  17
20  21  22  23  24  25  26   18  19  20  21  22  23  24
27  28  29  30               25  26  27  28  29  30  31

$ cal 10 2006
       October 2006
Sun Mon Tue Wed Thu Fri Sat
 1   2   3   4   5   6   7
 8   9  10  11  12  13  14
15  16  17  18  19  20  21
22  23  24  25  26  27  28
29  30  31
```

2. bc: 计算器

Windows中图形界面的计算器，是经常使用的一种便捷工具。在UNIX中也有一个类似的计算器程序bc,这个bc程序的功能非常复杂和强大，它支持数学函数库，变量，循环等许多编程功能。另外，这个计算器程序还可以进行任意精度的计算。例如，在Pentium III计算机上LINUX中，将精度设置为小数点后1万个有效数字，有理数计算比较快，计算正弦函数s(1.0)，其中1.0为弧度，需要半小时时间。精度2万个小数位，计算正弦函数s(1.0)需要3小时。可见精度是用时间换来的。

使用这一命令时，输入bc命令，然后输入表达式，每输入一个表达式回车后，系统给出表达式的值。退出计算器程序，按Ctrl-D（文件结束）键。

【例 1-5】使用 bc 命令的例子。

其中的圆括号内的楷体字提示使用bc时的注意事项。

```
$ bc        (启动 bc 命令)
18 * (5 + 5)   (输入一个表达式，表达式中可以使用括号)
180         (打印出表达式的计算结果)
1024 * 4096
4194304
(下边是一个高精度计算的例子。在 32 位系统中，数据线宽度 32 比特，CPU 一次可运算的最大整数是 4G，这也是 C 语言中无符号整数的最大值，下边的两个数以及它们的乘积都超过了 4G)
12345678901234567890123 45 * 98765432109876543210987654321
12193263113702179522618496034720321071359549253925
2.5 * 2.1     (浮点数运算)
5.2         (运算结果应当是 5.25,但是由于精度太低，未能取得期望的值)
1/113
0           (浮点数运算结果的精度太低，未能取得期望的值)
scale=40
(上述语句设置 bc 的内部控制变量，使得计算结果可保留到小数点后 40 位。很糟糕，scale 的默认值为零，才有上述两次计算结果的不如意)
2.5 * 2.1
5.25        (重新计算，得到正确结果)
1/113
.0088495575221238938053097345132743362831
(运算结果是无限循环小数，保留 40 个小数位)
(在这里按下 Ctrl-D 键，退出 bc 程序，返回 shell，出现 shell 提示符)
$
$ bc -l
(使用bc命令带-l选项，bc的内部控制变量scale自动被设置为20,再进行浮点运算2.5*2.1或1/113会得到期望的结果，期望保留更多的小数位数，可以自行设置 scale 控制变量。-l 选项原意是 library，定义了一个 bc 使用的任意精度的数学函数库，包括正弦函数 s(x)，余弦函数 c(x)等，另外，将 scale 初始设置为 20)
2.5 * 2.1
5.25
1/113
.00884955752212389380
p = 2.346901
```

（bc 允许使用 a~z 的 26 个寄存器，后边的计算如果经常使用这个值，将它存入到寄存器 p 中）

```
17.8 * p
41.7748378
198.23 * p
465.22618523
$
```

在使用浮点数运算的时候，最好直接使用bc -l选项。有关bc命令的更丰富功能，可参阅命令手册。

第 2 章 基本 UNIX 实用程序

众所周知，UNIX提供了很多的实用程序工具，有很多的命令，每个命令又往往有很多复杂的选项。掌握所有的命令几乎是不可能的，有很多厚厚的UNIX参考书详细的介绍这些命令。对于UNIX的命令参考手册来说，所有命令，每个命令的所有功能，都不分轻重的叙述的非常详尽。但并不是所有命令都等概率的被使用到，所以，命令手册仅适合于象字典一样地查阅。许多不太熟悉的命令可以借助联机手册查阅。熟练使用UNIX，必须掌握一些常用的命令和常用选项。有很多UNIX的教科书介绍这些命令。一个UNIX的使用者应当不停的根据自己的实际需要，学习和熟悉所需要的各种命令。程序员也可以根据自己的实际需要，编写适合自己特殊领域使用的新命令，但是，这些新设计的命令应当做到和UNIX已存的命令有相同的风格。使用UNIX的目的不同，所需要掌握的命令集自然会有不同的侧重。

UNIX的命令很多，有几百个，初看起来，显得非常杂乱。掌握这些命令，需要对它们分类。能够灵活使用这些命令，还必须了解隐藏在命令背后的一些基本概念。所有的命令操作的都是系统中的一些软件对象，如：文件系统，进程，信号量，等等。从操作系统的角度了解这些对象的特点性质，对理解为什么系统会设置一组处理这些对象的命令，会很有帮助。正如面向对象的系统中，对象本身的性质和特点是根本的，命令就是作用于对象之上的“软件方法”。

另外，UNIX的许多命令，有相似的风格。这些命令功能的设计和使用方法，并不是孤立的，往往可以和系统中其它的功能或者其它的命令配合，组装出更强的功能。了解这些特点之后，才能更灵活地使用UNIX。

在这一章中，介绍几个最基本的UNIX实用程序。这些命令的操作对象都是文本文件中的文本数据。在后续的章节中,还介绍与文件系统有关的命令。其它的命令，如进程操作，IPC对象的操作，在需要时介绍。本书对命令的介绍，不是命令的使用手册，更侧重于如何使用和在什么场合下需要这些命令。至于命令更复杂的功能，这里不做详尽的介绍。

UNIX中有很多文本文件的处理程序， 如：`more`，`less`，`pg`，`cat`，`hd`，`od`，`head`，`tail`，`sort`，`wc`，`grep`，`cut`，`paste`，`cb`，`pr`，`awk`，……。这些命令普遍有下列特点：

(1) 当不指定文件名(处理对象)时，从标准输入`stdin`获得数据。

(2) 当指定文件名时，从文件中获取数据,而且可以同时指定多个文件。

(3) 处理结果在标准输出`stdout`显示。

这些命令设计成这样的风格，使得用户可以方便地利用UNIX的重定向和管道功能组合出所需要的命令。可以指定多个文件名的做法，与shell的文件名通配符展开有关，在后面4.3节介绍。

2.1 more,less,pg:逐屏显示文件内容

在DOS中有同名的MORE命令。pg命令最先由AT&T UNIX开发。more命令最先由BSD UNIX开发。一般UNIX系统都提供这两个实用程序。more命令，最初的功能是用于显示满一屏之后，等待用户按键，再显示“更多”(more)一点。LINUX中开发了命令less,除了允许逐屏显示之外，还允许用户按键，再回退显示,显示比当前已显示内容“更少”(less)一点。使用举例：

```
more server.c
more *.[ch]
ls -l | more
pg *.[ch]
```

第一个命令，指定了一个文件server.c作为处理对象。第二个命令，指定多个文件作为处理对象，*号是文件名通配符。方括号括起来的两个字符，是UNIX文件名通配符的一种描述，要求文件名有c或者h后缀。第三个命令，指定了0个处理对象，这样more从标准输入获取数据。这里的管道符|，使得标准输入来自于上个命令的标准输出。ls -l命令用于列出当前目录，在随后的章节介绍。

more 与pg的使用：

(1)more显示满一屏后，屏幕最后一行为反转显示--more--或反转显示后再附加一个百分率，如--more--(15%)。当显示暂停之后，可以使用表2-1中的子命令。

表 2-1 more 命令的子命令

按键	功能
空格	显示下一屏。
回车	上滚一行，当你所感兴趣的段落内容正好处于当前屏幕的尾部，另有一部分在下一页中时，可以连续按回车，将感兴趣的部分滚动上来。
q	(quit)退出程序，后面的内容不再显示。
/*pattern*	搜索指定模式的字符串，模式描述使用正则表达式。
/	继续查找指定模式的字符串。
h	(Help)帮助信息。打印出more命令的所有功能列表。
Ctrl-L	屏幕刷新。

屏幕刷新功能经常用在这样的场合，突然有其它用户通过write或者talk送达本终端的信息，或者，超级用户用wall发送来的十五分钟后要关机的通知。还有一种情况是，事先在当前终端启动了一个后台进程，后台进程经过一段时间运行后，现在有了屏幕输出。这些信息到达之后，直接显示在终端屏幕上，并且会覆盖掉当前的more命令在终端上逐页显示的信息。在读完这些信息之后，按Ctrl-L键，刷新屏幕，恢复原来的显示。类似的，在全屏幕编辑程序vi中也使用Ctrl-L。有的应用软件也提供这种redraw的功能，通过按下终端的某个键，重新刷新终端屏幕。

more命令还有很多其它的功能，甚至可以后退回去浏览那些已经浏览过的页。这些功能，在不同系统中会有些差距。根据more的操作对象的不同，是磁盘文件(如:more

server.c)或者是标准输入(如:ls –l | more)，回退浏览的功能,会受到限制。more命令比DOS系统中的同名命令的功能强得多。

在LINUX系统中有命令less，和more的功能类似，但是，回退浏览的功能更强,可以直接使用键盘的上下箭头键，或者j,k,类似vi的光标定位键，以及PgUp，PgDn，或者Ctrl-F，Ctrl-B，Home，End键，使用起来更方便，是对more命令的增强。在LINUX中使用广泛。许多UNIX没有less命令，但是，这些UNIX中的more命令的增强功能融入了less命令的部分功能。

(2)pg显示满一屏后,屏幕最后一行为冒号(:)提示符，显示暂停，等待按键命令，每个命令之后还需要按回车键，这使得这个命令用起来比more更麻烦。表2-2列出了可以使用的按键命令。

表 2-2 pg 命令的子命令

按键	功能
回车	下一屏。
L	上滚一行。
Q	退出。
/*pattern*	查找指定模式的字符串，模式描述用正则表达式规则。
/	继续查找。
h	(help)帮助信息。
Ctrl-L	屏幕刷新。

比较而言，more比pg少按键，pg的每个命令要比more多按回车键。man命令也使用more或pg。一般来说，man先查找磁盘文件，再解压缩文件，最后用more或pg显示。根据满一屏后最后一行的提示,可以判断用的是pg还是more，然后使用相应的命令翻页或滚动。System V系统中，一般默认不太好用的pg为man的分屏浏览器。系统管理员可以设定man命令中使用more或pg。例如：在SCO UNIX中将文件/etc/default/man中的行

```
PAGER=/usr/bin/pg
```

改为

```
PAGER=/usr/bin/more
```

那么，man将使用more。

2.2 cat:列出文本文件内容

cat命令用于列出文本文件的内容，功能和DOS中的TYPE命令类似。例如：

```
cat try.c
```

指定处理对象一个，打印文本文件try.c的内容。

```
cat > try.txt
```

指定处理对象0个，程序从标准输入（键盘）获取数据，直到按下Ctrl-D标志输入结束。程序输出被重定向到文件try.txt。这是一种很简单的建立新文件的方法。

```
cat tryl.c try2.c try.h
```

命令行参数:3个，指定处理对象为三个文件，程序顺序打印出三个文件的内容。

```
cat tryl.c try2.c try.h > trysrc
cat makefile *.[ch] > src
```

上述两个命令都是将多个文件，输出重定向到一个文件，完成文件合并。cat命令得名于“串接”(catenate)，在指定多个文件时，cat就是将这些文件串接起来打印。

2.3 od:列出文件每个字节的内容

od得名于(octal dump)，八进制打印。这个命令也有一些选项。使用这个命令，可以看到文件的存储结构，可以用于打印可执行文件，或者非文本格式的数据文件。

```
od a.dat
```

以八进制打印出文件。

od命令的-x选项，使用十六进制打印文件内容。由于正好两个十六进制数字描述出一个字节，所以，十六进制打印经常被使用。od命令的-c选项,对可打印字符打印出字符，不可打印字符，打印出ASCII码。

当不指定处理对象时，od命令从标准输入获取数据。下边的组合命令，可以查出字符abcdABCD的ASCII码。echo命令将命令行参数输出到标准输出。

```
echo abcdABCD | od -x
```

【例 2-1】使用 od 命令的例子。

```
$ od a.dat
0000000  046532 131400 004000 000000 020000 000153 016220 002000
0000020  000400 051571 071564 062555 020126 062562 071551 067556
0000040  020064 027071 030015 005044 075007 000016 017570 004630
0000060  020533 000514 036441 000000 000000 056011 000000 004000
0000100  000000 000000 000000 005000 000000 047105 002474 107510
0000120  072025 002035 007566 007152 062000
（最左列以十六进制形式指出数据在文件中的偏移量）
$ od -c a.dat
0000000    M   Z   ? \0  \b  \0  \0  \0      \0  \0   k 034 220 004  \0
0000020  001  \0   S   y   s   t   e   m       V   e   r   s   i   o   n
0000040        4   .   9   0  \r  \n   $   z  \a  \0 016 037   x  \t 230
0000060    !   [ 001   L   =   !  \0  \0  \0  \0   \  \t  \0  \0  \b  \0
0000100   \0  \0  \0  \0  \0  \0  \n  \0  \0  \0   N   E 005   < 217   H
0000120    t 025 004 035 017   v 016   j   d
（从这里能看到，文件中有 System Version 字符串）
$ od -x a.dat
0000000  4d5a b300 0800 0000 2000 006b 1c90 0400
0000020  0100 5379 7374 656d 2056 6572 7369 6f6e
0000040  2034 2e39 300d 0a24 7a07 000e 1f78 0998
0000060  215b 014c 3d21 0000 0000 5c09 0000 0800
0000100  0000 0000 0000 0a00 0000 4e45 053c 8f48
0000120  7415 041d 0f76 0e6a 6400
$ echo abcdABCD | od -x
```

```
0000000  6162 6364 4142 4344 0a00
$
```

2.4 head 与 tail:打印文件头或尾

head和tail的用法类似，可以打印出指定文件头部，或者尾部的一部分内容。如果感兴趣的内容仅在头部和尾部，可以使用这个命令。UNIX的选项一般以减号开头，这里减号后边的数字是期望看到的行数。对head和tail命令若未指定行数，默认为显示10行。

【例 2-2】使用 head 和 tail 命令的例子。

```
$ head -15 ab.c
(显示文件 ab.c 中前 15 行)
$ head -23 a.c b.c c.c | more
(显示三个文件各自的前 23 行共显示 69 行)
$ tail -20 liu.mail
(打印出文件尾部的 20 行，看看邮件尾部的发信者签名)
$ netstat -s -p tcp | head -5
tcp:
   5902873 packets sent
      5118107 data packets (3423705271 bytes)
      35445 data packets (18222213 bytes) retransmitted
      329697 ack-only packets (35578 delayed)
```

命令netstat是个很重要的网络命令，-s选项(Statistic)是打印统计信息，-p选项(protocol)指定协议，在这里指定TCP协议。TCP协议的详细统计信息有很多，这里仅列出前5行，因为，这5行中含有操作员感兴趣的TCP数据重传数据量，一段时间内反复执行这一命令，可以看出当前一段时间的通信质量。

tail命令有个重要的选项-f(forever)，经常会被用到。具体用法为：

tail –f *filename*

其中，这里的文件名是一个普通的磁盘文件名。

【例 2-3】使用 tail 命令实时显示文件新追加的内容。

```
tail –f /usr/adm/pppd.log
```

上述命令列出文件/usr/adm/pppd.log的尾部(UNIX文件系统路径名分割用正斜线/，与DOS中使用反斜线\不同)。文件尾部的10行打印完毕之后，tail命令并不退出，继续等待。UNIX允许多个活动程序同时操作磁盘上的同一个文件。如果系统中其它的活动进程会在pppd.log文件的尾部追加，那么，tail会实时的打印出这些追加的内容。随着其它进程对pppd.log文件的不断追加，tail命令会随着系统其它进程对文件追加的进度，及时打印出这些信息。停止tail进程，按下Ctrl-C键。

UNIX的许多程序，例如：实现PPP通信协议的进程pppd(Point-to-Point Protocol Daemon)，在运行的时候，可以根据这个程序的某些配置信息，有选择性的打开或者关闭一些日志(log)信息，并把这样的信息存储到一个文本格式的日志文件。这样，可以观察协议运行的过程，提供了一个调试手段。如果程序员自己设计的比较复杂的应用程序，也应当采用类似这样的做法。这样的程序，正常情况下，关闭大部分的日志信息。正常运行情况下，日志信息很少，不影响程序执行的效率。当在系统初装调试阶段或者运行中发现有异常情况时，通过设置程序的某些配置开关，打开所需要观察的日志信息，例如：所有收发报文的内容，计时器的活动等等。那么，程序记录它的活动过程到一个日志文件中，这些信息包括，在什么时间，出现了什么事件，完成了什么动作，操作员可以通过事后检查这一日志文件的内容，发现系统配置中的错误，甚至是程序的BUG。调试结束正常运行时，修改配置，关闭这些信息，以免影响效率和过分挤占磁盘空间。至于日志信息的格式，打开和关闭日志信息的配置方法，与具体的程序相关。程序活动时，及时记录了程序的活动过程（日志信息）到磁盘文件以供事后检查，操作员可以使用tail -f命令，实时看到程序的活动过程。执行和关闭tail -f不影响日志文件的生成。

2.5 wc:字计数

字计数(word count)命令wc,可以列出文件中一共有多少行,有多少个单词,多少字符,当指定的文件数大于1时,最后还列出一个合计。

【例 2-4】使用 wc 命令的例子。

```
wc sum.c                    (1个文件)
wc x.c makefile stat.sh     (多个文件)
```

以下是上机操作实例。

```
$ wc *.[ch] makefile *.sh
1912         6532         49143           auth.c
1227         4038         32394           ccp.c
860          2558         22487           chap.c
124          695          4702            chap.h
762          2129         17159           fsm.c
144          792          5237            fsm.h
2168         7487         56500           lcp.c
87           288          2035            magic.c
1827         5833         44234           main.c
306          1901         11841           md5.c
58           349          3048            md5.h
390          1343         9138            multilink.c
1545         5220         37149           options.c
738          4142         28320           pppd.h
876          2712         18623           utils.c
347          1324         9673            makefile
78           735          5125            ppp.sh
```

```
13449        48078        356808             total
$ ps -ef | wc -l
108
$ who | wc -l
10
```

列出的内容，第一列为文本文件的行数。第二列，是“词”数，wc有内部的一个规则来定义怎样的字符序列算作一个词，如果不是一篇英文文章，词数统计往往参考意义不太大。第三列，是字符数。最后一列是文件名。当处理对象为两个或者更多时，最后一行有个合计。在上例中，可以统计出所有的源程序文件共有13449个文本行，共356808字节。

wc有些选项，其中，-l选项，只给出行计数，-c选项，只给出字符计数。wc经常被用作其它的统计用途，而且经常用来和其它的命令组合起来使用。例如：

```
ps -ef | wc -l
```

上述命令，指定了0个处理对象，那么，以标准输入作为输入，利用UNIX的管道功能，将上个命令的输出，作为下个命令的输入。ps –ef命令，列出系统中的所有进程（ps是process status），每个进程占用一行，所以wc最终统计出的行数，就是系统中的进程总数，可以看出系统中有107个进程（ps的输出第一行是表头，其余的，每进程一行，关于ps命令，后边的7.1.8节中介绍）。

```
who | wc –l
```

统计出系统已成功登录的用户个数。

2.6 sort: 对文件内容排序

例: sort telnos>namesorted

上述命令，将文件telnos的内容排序，输出信息重定向到文件namesorted中。默认的排序方法是，将文件每行作为一个整体，按照ASCII码字符串的方式比较，从小到大排列。字符串比较时，注意会出现32 > 123。

sort有许多选项，可以根据需要选择一行中某一部分作为排序关键字，选择升序或降序，取消相同内容的行，字符串比较时对字母是否区分大小写等等。-n选项（Numberic），对于数字按照算术值大小排序，而不是按照字符串比较规则。

下面是使用sort排序的例子。列出当前目录下存储空间最大的10个文件。其中，ls命令的-s选项(size)列出文件时，文件名前边的数字是文件大小，单位不是字节数，是“块”数。不同的系统，1块的大小可能会不同，有的是512字节，有的是1024字节。ls命令类似DOS的DIR命令，在后边的4.4.1节中介绍。

【例 2-5】使用 sort 命令的例子。

```
$ ls -s | sort | tail -10 (默认的按照字符串方式比较进行排序)
  44 main.c
```

```
  48 auth.c
  56 lcp.c
1268 BUGS.report
1720 paper.pdf
202712 document.pdf
27052 disk.img
27056 linux-src.tar.Z
3532 pppd.log
total 263724
$ ls -s | sort -n | tail -10
  40 options.c
  44 main.c
  48 auth.c
  56 lcp.c
1268 BUGS.report
1720 paper.pdf
3532 pppd.log
27052 disk.img
27056 linux-src.tar.Z
202712 document.pdf
```

2.7 tee:三通

将从标准输入stdin得到的数据抄送到标准输出stdout显示，同时存入磁盘文件中。这一功能类似水管或电线的T型三通，命令取名一个字母t太短，根据字母T的发音[ti:]取名tee。这也是一个常用的命令。

许多程序员都有过这样的经历，编写源程序，编译后运行，运行过程中有许多打印，但是，打印的信息较多，很快地滚动过屏幕，程序员又想再重新看看已经滚过屏幕的信息，这时，就可以使用tee命令。例:

```
./myap | tee myap.log
```

其中,当前目录下命令myap是程序员自编程序的可执行文件（目录./代表当前目录，与DOS同）。这样，在执行myap，当前终端的键盘输入仍然为myap的输入，原先myap的输出仍然能照常实时显示输出，同时，又将看到的信息存盘到文件myap.log,可以事后查阅。

2.8 正则表达式的概念

在UNIX的文本文件处理中，广泛地使用正则表达式(Regular Expressions)的概念，使用正则表达式描述一个字符串模式(Text Patterns)，在模式匹配(Pattern matching)操作中使用。抽象地说，模式匹配是一种二元运算，两个操作数分别称为“模式”和“字符串”，同除法运算一样，这种运算不遵守交换律。模式描述使用正则表达式，运算结果是一个逻辑值。

正则表达式的概念被广泛应用在文本处理中，UNIX系统中vi的模式查找与替换命令中,使用正则表达式来描述“模式”。前面介绍的more命令，也使用正则表达式。正则表达式的概念也适用于lex，grep，awk，expr等用于处理文本数据的其它命令中。在Windows的许多软件中也使用正则表达式。例如：在Turbo C和Borland C++, Visual C++编辑器，等等。

在不同的软件中，对正则表达式的定义会稍微有些不同，必要时应查阅这些软件的手册，了解它们自己对正则表达式的定义，这些软件的正则表达式定义，常常是对基本正则表达式定义的扩展。下面给出的是最基本的正则表达式定义。

1. 正则表达式中的特殊字符

正则表达式的特殊字符，共6个

. * [\ ^ $

除此之外的其它字符与其自身匹配。如：hello,bye。用反斜线可以取消特殊字符的特殊含义(转义)。如：正则表达式end\.只与字符串end.匹配

2. 单字符正则表达式

长的,复杂的正则表达式是由单字符正则表达式构成的.

⑴ 普通字符

除了前边列出的六个特殊字符外，其它字符与其自身匹配，如:a与a匹配，b与B不匹配，?与j不匹配。

⑵ 转义字符(\)

在特殊字符前,缀以反斜线,则丧失字符的特殊含义，与其自身匹配。

\. * \$ \^ \[\\

⑶ 圆点(·)

圆点匹配任意单字符。

⑷ 单字符集合的定义

使用中括号。左中括号与其后的右中括号一起定义一个集合，描述一个字符。

① 在左中括号与右中括号之间的字符为集合的内容，

如：单字符正则表达式［abcd］与a，b，c，d中任一字符匹配。

② 可以用减号定义一个区间

如[a-d] [A-Z] [a-zA-Z0-9]

若减号在最后,则失去表示区间的意义,如:[ad-]只与3个字符a,d,-之一匹配。

③ 可以用^表示补集

若^在开头,则表示与集合内字符之外的任意其它单字符匹配,如:[^a-z]匹配任一非小写字母。若^不在开头,则失去其表示补集的特殊意义。如:[a-z^]能匹配27个单字符。

④ 正则表达式的特殊字符，在方括号内时，仅代表它们自己。

如:[*.]可以匹配2单字符

3. 单字符正则表达式的组合

⑴ 简单串结

正则表达式abc，是三个单字符正则表达式的串结，仅能匹配字符串abc。正则表达式[A-Z].[0-9].，是四个单字符正则表达式的串结，要求字符串的第一个字符是大写字母，第三个字符是数字，第二第四字符可以是任意字符。

正则表达式[Mm]akefile匹配Makefile或者makefile.

正则表达式a\[i]*3\.14匹配字符串a[i]*3.14。后边的介绍，可以看出正则表达式a[i]*3.14并不匹配字符串a[i]*3.14，对于C语言的程序员来说应当注意正则表达式的这些处理。

⑵ 星号(*)

单字符正则表达式后跟*，则匹配此单字符正则表达式的0次或任意多次出现。

⑶ $行尾符$和行首符^

$只有出现在正则表达式最尾部时才有特殊意义，否则与其自身匹配。类似地，^只有出现在正则表达式最首部时才有特殊意义,否则与其自身匹配。

【例 2-6】正则表达式中符号*的作用。

(1) 正则表达式12*4

1234	不匹配
1224	匹配
12224	匹配
14	匹配

此例中*作用于它左面的单字符正则表达式2。注意正则表达式12*4与字符串14是匹配的。

(2) 正则表达式[A-Z][0-9]*

此例中*作用于它左侧的单字符正则表式为[0-9],代表：

[A-Z]
[A-Z][0-9]
[A-Z][0-9][0-9]
[A-Z][0-9][0-9][0-9]
……

与A,A1,C45,D768匹配，与b64512,T546t不匹配

(3) 正则表达式[Cc]hapter *[1-4]

在*号前有一个空格，允许数字1-4之前有多个或者0个空格。可匹配Chapter2, chapter 3,等等。类似的，正则表达式：

```
a\[i] *= *b\[j] *\* *c\[k]
```
可以匹配字符串a[i]=b[j]*c[k]，并允许等号和星号两侧有多个或者0个空格。

【例 2-7】正则表达式中$和^的作用。

(1) 123$ 则匹配文件中行尾的123,不在行尾的123字符与正则表达式123$不匹配。

(2) $123与字符串$123匹配

(3) .$匹配行尾的任意字符。

(4) 正则表达式^Hello匹配行首的Hello字符串，不在行首的Hello串不匹配。

(5) 正则表达式Hel^lo与字符串Hel^lo匹配。

(6) ^号后跟四个空格，那么，仅匹配行首的连续4个空格。

需要注意的是，正则表达式规则与文件名匹配规则是不同的。一般来说，正则表达式规则用于文本处理的场合，文件名匹配规则用于文件处理的场合。文件名通配符中的星号，问号，圆点，在正则表达式中有不同的解释。

2.9 grep,egrep 与 fgrep:在文件中查找字符串

grep（Global regular expression print）命令是一种文本过滤程序，按照正则表达式的规则，仅筛选出含有指定模式字符串的文本行。语法：

grep *pattern file-list*

1. grep

如果指定的文件数>1,那么，当查找到指定字符串时,整个行，连同该行处的文件名一起显示。如果指定的文件数≤1，那么，只列出含有指定模式的整个行的内容，但不显示文件名。

【例2-6】grep命令的使用。

```
grep O_RDWR /usr/include/*.h
```

用于查找C语言中宏定义O_RDWR在哪些头文件中定义,查找范围为多个文件。类似的命令用于查找C语言的struct类型定义，等等。

```
grep routed /etc/tcp 指定文件数=1
who | grep liang       指定的文件数=0
```

2. egrep

扩展grep，在描述模式时,使用扩展的正则表达式，egrep对基本的正则表达式进行了扩展，可以用圆括号()和表示“或”的符号|，其次，还定义了和正则表达式中的星号

地位类似的+和?。*号表示它左边的单字符正则表达式的0次或多次重复，对应的，+号表示1次或多次，?表示0次或一次。

【例 2-8】egrep 的扩展正则表达式的使用。

egrep可以使用扩展的正则表达式，下面是几个扩展正则表达式以及他们可以匹配的字符串示例。

(xy)*	可匹配空字符串，xy，xyxy，xyxyxy,等等。
(pink|green)	与pink或green匹配。
[0-9]+	不匹配空字符串，匹配长度至少为一数字串。
a?	匹配零个或一个a。

下面的两个命令按扩展的正则表达式规则检索字符串。

```
egrep '(SEEK_|IPC_)' *.h
egrep '[0-9]:[0-9][0-9] (client|server)$'
```

egrep在指定模式方面比grep更灵活，但算法需要稍多的处理时间。

3. fgrep

快速grep，按字符串搜索而不是按模式搜索。fgrep运算速度快，适合于从大量的数据中进行检索。但只能指定字符串，不可按模式查找。

4. 选项

grep/fgrep/egrep有若干选项用以控制输出格式。

-n　显示时每行前面显示行号

-i　字母比较时忽略字母的大小写。

-v　显示所有不包含模式的行

【例 2-9】grep 选项的使用。

(1)grep -n __DATE__ *.c

在所有的后缀为.c的文件中查找含有正则表达式__DATE__的行，并打印行号。当文件数超过一个时，除了输出行号，还输出文件名。

(2) grep -v '[Dd]isable' device.stat>device.active

将文件device.stat中取消所有含有指定模式的行，生成新文件device.active。

(3) grep -i richard telnos

在文件telnos中检索字符串richard，不顾字母的大小写。

(4) grep '[0-9]*' chapter1

由于[0-9]*与空字符串匹配，上述命令打印出chapter1文件中所有行，而不是仅打印出含数字的行。正确的用法应当是：grep '[0-9][0-9]*' chapter1。

打印出文件chapter1中所有含有数字的行。或者使用egrep的扩展正则表达式：

```
egrep '[0-9]+' chapter1
```

这里给出了grep命令的三个常用选项，UNIX的grep命令共有十几个选项。前面介绍的命令中，也介绍了一些命令选项。这些选项都以减号开头，丰富的选项，为命令提供了丰富的功能选择，尽管有些功能并不经常用到。UNIX风格的大部分命令都是在一个命令行内，通过命令行参数的形式提供程序处理所需要的数据，处理对象，以及描述处理方法的命令选项，程序开始运行后，就不再需要任何其它的交互式输入，直到命令执行完毕。

以这里的grep命令为例，不善于使用命令行参数的程序员会将命令设计成这样的交互式界面：

首先，输入grep命令，按下回车键程序开始执行。然后，程序提示：“输入待查找的模式:”，等待用户输入。完成后，再问：“显示时每行前加行号吗(Y/N)?”，等待用户输入Y或者N。然后，再问：“模式匹配时忽略英文字母的大小写吗(Y/N)?”,等待用户输入。完成后,再提示:“1-显示包含模式的行 2-显示不包含模式的行,请选择(1/2):”。等这些问题都回答完之后，再提示：“请输入文件名:”，操作员输入文件名。处理结束后，再问：“还需要查找其它的文件吗(Y/N)？”，……。

这样的命令风格，通过交互式问答，界面非常友好，简单易用，即使是事先对这个命令毫不知晓的“傻瓜”也可以轻松使用。由于命令有很多可选功能，那么，可以改进上面的众多问答，组织成一个菜单的形式，用户选择菜单的项目，设置一些必要的开关。但是，UNIX风格的命令不这样做。尽管这种友好的界面适合初学者，但是对熟练使用命令的用户来说，过分啰嗦。许多软件项目的实践表明，菜单方式或者交互式提示输入对初学者很方便，但是当系统变得非常复杂时，远没有使用命令方式更有效。UNIX风格的命令不使用这样的交互式问答的重要原因是，这样的交互式输入非常不便于把命令用于批处理程序，不便于利用文件名通配符，不便于利用重定向和管道功能与其它的命令协同工作。类似grep这样的程序，可以在Windows等图形系统中，设计出非常直观和便于使用的图形界面，选项可以用复选框代替。但是，图形界面的缺点是不便于批处理程序在不需要任何人工干预的情况下自动地处理数据。所以，许多Windows的图形界面程序常常会有另一个便于批处理操作的命令行程序版本。程序员在设计一些命令时，应当考虑到这样一些使用方面上的因素，设计自己的应用程序时界面风格应当遵守惯例，并且把程序的使用界面与核心处理分割清晰。

2.10 awk:文本处理语言

a.w.k分别为该实用程序的三位设计者姓氏的第一个字母，awk的三位作者是Alfred V. Aho, Peter J. Weinberger, Brain W. Kernighan。

awk程序逐行扫描文本文件，并进行处理。和其它的文本处理程序一样，awk可以指定0到多个文件作为处理对象。和grep类似，awk也是一种对文本文件进行过滤的程序，通过逐行扫描筛选出满足指定条件的文本行，并且，能够生成报表。报表格式非常灵活，是awk得到广泛使用的主要原因。

awk有很强的功能，它是一种模式扫描和处理语言，支持条件控制，循环控制，变量定义，函数等功能。由于UNIX和C语言的特殊关系，awk的很多地方设计的类似于C语言。在这里仅仅介绍awk的几种简单并常用的用法。

用法1: awk '*Program*' *file-list*

用法2: awk -f *ProgramFile* *file-list*

awk是一种文本处理语言，可以用这种语言编写文本处理程序。awk的用法1是在awk的命令行中直接提供程序内容。这用在程序很简单的情况下，如：只有一行程序，用起来很方便。由于shell的原因（在第六章的shell编程中6.5.9节解释），用法1的程序两侧需要用单引号括起，以避免许多不必要的麻烦。当书写的程序非常复杂时，可以将程序存放到一个文件中，使用用法2引用一个实现编制好的程序文件。

awk程序的写法是：

condition {*action*}

awk的做法是，对满足“条件*condition*”的行，执行花括号中指定的这些“动作*action*”。为了解释“条件”和“动作”，首先介绍awk的内置程序变量。表2-3列出的内置程序变量，是awk文本处理程序中不需要定义就可以直接使用的变量。

表 2-3 awk 编程中的内置变量

变量	含义
NR	当前记录的记录编号。(No. of Record)
$0	当前记录。
$1, $2, ……	当前记录中的域。
FILENAME	当前输入的文件名。

awk把输入文件的每一行作为一个“记录”,变量NR就是行号。每行中，用空格或者制表符分隔开的部分，叫做每个记录中的“域”。变量$1指的是第1个域内容，依次，变量$2指的是第2个域内容，……。特别的,$0指的是整个这一行的内容。

描述“条件”时，有几种方法：

⑴ 如果不指定任何条件，那么对文本文件的所有行进行处理。

⑵ 可以使用与C语言了类似的关系算符,见表2-4。

表 2-4 awk 编程中的运算符

算符	意义
<	小于
<=	小于或等于
==	等于
>	大于
>=	大于或等于

!=	不等于
||	条件或
&&	条件与

⑶ 正则表达式的模式匹配

如果文本行中含有这一模式描述的字符串，就对这行进行处理，执行相应的“动作”。这种按照正则表达式模式匹配规则筛选文本行并进行处理的方法很常用。模式描述方法为：

/*pattern*/

⑷ 特殊的条件

BEGIN和END。BEGIN之后的动作，在awk开始处理所有文本行之前执行。同样，END之后的动作，在awk处理完所有文本行之后执行。

描述“动作”时，简单的用法有：

print 变量1， 变量2，……

printf("格式串"，变量1， 变量2，……)

由于awk允许使用用户变量，条件和循环，动作部分可以设计得很灵活。大部分情况下，上述两种简单的用法就够用。print将用逗号隔开的变量打印，打印时用空格隔开。printf的用法和C语言里的printf函数用法类似，熟悉C语言的用户，可以灵活地使用printf的格式控制字符串。

awk命令覆盖了grep命令的功能，可以对处理对象按行筛选，按列提取所需要的信息，因而awk是一个非常重要的命令。

【例 2-10】使用 awk 命令的例子。

```
$ date
Thu May 27 22:02:22 BEIDT 2004
$ date | awk '{print $4}'
（未指定条件，处理所有的文本行）
22:02:42
$ who
zhang   ttylb   Sep 29  11:20
liang   ttyla   Sep 29  11:53
zhang   ttylf   Sep 29  12:04
feng    tty1c   Sep 29  12:54
$ who | awk '/^ *zhang / {printf("%s ", $2)}'
(仅处理含有正则表达式 zhang 字符串的文本行。由于 printf 的格式字符串尾不含\n,程序执行完之后，不换行，导致下个命令的提示符$在打印行的行尾)
tty1b tty1f $
$ ls -s | awk '$1 > 2000 { print $2 }'
(这里的描述条件为：第一列的取值大于 2000 的文本行)
disk.img
document.pdf
linux-src.tar.Z
pppd.log
$ cat list.awk
```

（事先编辑好的 awk 程序文件，程序中含有三组“条件 {动作}”描述。其中：BEGIN 的动作有三个，程序执行时，awk 处理所有文本行之前，执行 BEGIN 指定的三个动作；处理完所有文本行之后，执行 END 指定的一个动作；最后一个程序块，未指定任何条件，对所有文本行执行这个动作。在这段程序中使用了内置变量 FILENAME 和 NR，$0）

```
BEGIN {
    printf("====================================\n")
    printf("FILENAME %s\n", FILENAME)
    printf("------------------------------------\n")
}

END { printf("====================================\n") }

{ printf("%3d: %s\n", NR, $0) }
```

```
$ awk -f list.awk md5.c
====================================
FILENAME md5.c
------------------------------------
  1:
  2: #include "md5.h"
  3:
  4: /* forward declaration */
  5: static void Transform ();
  6:
  7: /* F, G, H and I are basic MD5 functions */
  8: #define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
 ……
298:   buf[2] += c;
299:   buf[3] += d;
300: }
301:
====================================
```

2.11 sed:流编辑

sed(stream editor)是一个流编辑程序，当指定的处理对象为0个文件时，它从标准输入获取输入字符流，否则，将文件中的数据作为作为输入字符流。对输入字符流进行编辑处理，加工处理后再输送到标准输出。

用法1: sed '命令' 文件名列表

用法2: sed -f 文件名 文件名列表

这两种用法的区别和awk命令类似。

【例 2-11】使用 sed 命令的例子。

(1) `tail -f pppd.log | sed 's/145\.37\.123\.26/QiaoXi/g'`

sed的编辑命令有很多,这里的s命令是“替换（substitute）”，三个斜线分割的第一部分是正则表达式145\.37\.123\.26，第二部分是替换字符串QiaoXi，最后的g是global flag，这一特征字符，使得s命令在一行中遇到多个模式描述的字符串时，都替换为QiaoXi，否则，仅替换一次。

上述命令中，sed将IP地址转为一个名字。

(2) `tail -f pppd.log | sed -f sed.script`

其中sed.script文件内容如下：

```
s/145\.37\.123\.26/QiaoXi/g
s/102\.157\.23\.109/LiuYin/g
s/145\.37\.123\.57/DaTun/g
 ……
```

使用这样的方法，定义一张IP地址/名字的对照表，sed可以将一些程序的输出内容进行编辑替换，加工之后再显示出来。

2.12 tr:翻译字符

用法：`tr` *string1 string2*

把stdin拷贝到stdout,对选出的字符作替换。在字符串*string1*中出现的输入字符被替换为字符串*string2*中的对应字符。

【例 2-12】使用 tr 命令的例子。

(1) 将大写UVX改写为小写uvx。

```
cat telnos | tr UVX uvx
```

(2) 可以使用［ ］指定一个集合。

```
cat report | tr '[a-z]' '[A-Z]'
```

将小写字母改为大写字母。

(3) 也可以使用\加三个八进制数字（类似C语言中描述字符常数的方法）表示一个字符。下面的命令将%改为换行符：

```
cat file1 | tr % '\012'
```

在第一章1.3.5节中介绍过UNIX和DOS文本文件的区别，下面的命令将按照二进制格式从DOS拷贝来的文件中多余的回车改为空格，回车的ASCII码是八进制的015。

```
cat myap.c | tr '\015' ' ' > myap1.c
```

这几个例子的命令中，注意不要漏掉必需的单引号。

2.13 cmp 和 diff:比较两个文件

用法：cmp *file1 file2*

用法：diff *file1 file2*

这两个命令用于比较两个文件是否相同。

cmp命令逐个字节比较两个文件是否完全相同，两个文件完全相同时，不给出任何提示。当两个文件不同时，打印出第一个不同之处。这个命令常用来判断两个文件的内容是否完全一致，无论是ASCII码文件还是二进制格式的程序或数据文件。在Windows中有类似的命令COMP。

对于程序员来说，有时候需要比较两个版本的源程序文件，以寻找出这两个源程序文件之间的差别。这样可以判断出新版本的源程序文件在原先版本的源程序文件基础上作了何种改动。许多读者使用过Windows下类似功能的FC命令(File Comparison)。UNIX用于完成这项功能的命令是diff。无论是Windows的FC命令，还是UNIX的diff命令，都可以逐行比较包括源程序文件在内的任意内容的文本格式文件。Windows的FC命令使用时经常需要/N选项，在列出文本文件行时打印行号。

命令diff *file1 file2*每发现两个文件中的一处不同，就列出一个如何将f1.c转化为f2.c的指令，这些指令有a(Add)，c(Change)和d(Delete)，指令的格式见表2-5。指令用一个字母a,c或d表示，指令字母左边的行号是*file1*的行号，指令右面的行号是*file2*的行号。列出内容时，大于号后边的内容是需要在*file1*文件中增加的内容，小于号后边的内容是需要从*file1*中删除的内容。

表 2-5 由命令 diff *file1 file2* 产生的文件转化指令

指令	如何将文件*file1*转化为文件*file2*
*l1*a*l2*,*l3* > (*file2*第*l2*~*l3*行内容)	将*file2*的第*l2*~*l3*行追加到*file1*的第*l1*行之后。
l1,*l2*c*l3*,*l4* < (*file1*第*l1*~*l2*行内容) --- > (*file2*第*l3*~*l4*行内容)	将文件*file1*的第*l1*~*l2*行换成*file2*的第*l3*~*l4*行。
l1,*l2*d*l3* < (*file1*第*l1*~*l2*行内容)	将文件*file1*的第*l1*~*l2*行删除以后，就与*file2*的第*l3*行以后内容相同。

【例 2-13】 比较两个不同版本的 C 语言源程序文件，找出文件的改动之处。

源程序文件 f1.c 被修改后的新版本源程序文件为 f2.c，下面是 UNIX 下比较两个文本文件 f1.c 和 f2.c 的结果。

```
$ cmp f1.c f2.c
f1.c f2.c differ: char 69, line 3
$ diff f1.c f2.c
3,5d2
<     tmp->vm_mm = mm;
<     mm->map_count++;
```

```
<    tmp->vm_next = NULL;
260c257
<    i = (i+1) * 8 * sizeof(long);
---
>    i = i * 8 * sizeof(long);
528c525
<    p->swappable = 1;
---
>    p->swappable = 0;
548a546,547
>    retval = p->pid;
>    p->tgid = retval;
```

diff给出的报告，列出了两个文件f1.c和f2.c的四处不同。其它内容，两个文件完全相同。这个报告很简洁，初看起来，不太容易理解。这四处不同，描述出文件f2.c对f1.c都作了哪些方便的修改。

第一处修改是，3,5d2，将文件f1.c的第3~5行删除了，删除的三行被列了出来。

第二处修改是，260c257，文件f1.c的第260行，被换成了f2.c的第257行内容。然后分别列出了f1.c中原先的程序行和被f2.c修改后的新的程序行，可以看出将i+1换成了i。

第三处修改是，528c525，文件f1.c的第528行，被换成了f2.c的第525行内容。然后分别列出了f1.c中原先的程序行和被f2.c修改后的新的程序行，可以原先的赋值1改为了0。

第四处修改是，548a546,547，文件f1.c的第548行之后增加了两行内容。然后列出了增加的这两行。

diff命令常用的选项有：

-b 逐行比较两个文件时忽略每行结尾处的多余空格。

-e 为UNIX的行编辑程序ed生成脚本文件。ed命令使用这个脚本文件编辑*file1*文件，就可以变成文件*file2*。

第 3 章 全屏幕编辑程序 vi

早期的UNIX提供的编辑器是行编辑ed，早期的DOS中也有类似的行编辑器edlin，现在的Windows已经不再提供。UNIX的全屏幕编辑器vi(名字取自visual)最早出现于BSD UNIX，vi出现于上世纪七十年代，现在所有的UNIX版本都支持vi编辑器。目前，行编辑ed还经常用于shell脚本程序中，在脚本程序中，根据用户的输入信息修改一个文本文件的内容，在6.3.2节中给出了一个脚本文件中使用ed命令的例子。行编辑程序ed的显示是面向行的，对终端的类型和特性没有任何的特殊要求和限制，ed程序本身也比较简单。vi可以交互式编辑文本文件，编辑是面向屏幕的，终端的类型设置必须正确，否则，无法正常工作。vi编辑器有很多忠实的爱好者，也有很多不认同者。由于vi在所有UNIX之间通用，也可以应用于各种各样的终端，占用系统资源很少，所以，vi仍然被广泛使用。

3.1 vi 的启动方法

用法：vi *filename*

例如：vi abc.c，启动vi编辑文件abc.c。

vi有搜索命令和卷动功能，可以用来浏览文本文件，比more,less等更方便，但是，应当避免在浏览时对文件无意中做出错误的修改，可以使用view代替vi进入编辑程序，编辑程序就不允许修改文件内容。

3.2 vi 的选项

vi有40多个选项控制vi的运行。设定方法：

(1) 用.exrc文件控制

vi一启动后就自动读取用户自己主目录(Home Directory)下的文件.exrc，获取用户自设定的vi选项，未指定的选项按默认值处理。

如：建立文件.exrc,其中含有如下两行：

```
set number
set showmode
```

其中第一条命令使得vi在列出每一行时，在左边列出行号。第二条命令使得vi在屏幕右下角标志出当前是否处于输入状态。

在vi中使用set命令

例：

```
:set number    在列出每一行时，左边列出行号。
:set showmode    在屏幕右下脚标志出当前是否处于输入状态。
:set nonumber    在列出每一行时，在左边不列出行号。
:set noshowmode    在屏幕右下脚不标志出当前是否处于输入状态。
```

:set all 列出所有开关的当前状态。

一般的，设置选项和取消选项的命令格式分别为（其中*option*为选项名）

:set *option*

:set no*option*

3.3 vi 的工作方式

vi的工作方式是命令方式和输入方式。vi启动后就进入命令方式，参见图3-1。

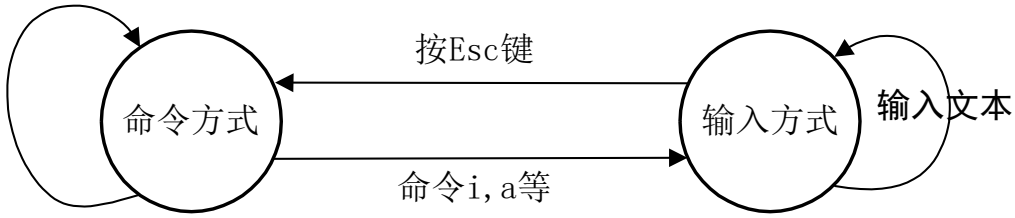

图 3-1 vi 的工作方式转换

处于命令方式时，用户键入的内容被当作vi的命令来解释，一般处于命令方式下按键无回显（以冒号打头的命令和查找命令/除外）。编辑命令i，a等，可以从命令方式转到输入方式。

处于输入方式时，用户键入的所有内容全部作为输入的正文内容，用户可以输入多行，每输入完一行后按回车转入下一行，正文输入时有回显。输入完毕按键盘左上角的Esc键，返回到命令方式。

3.4 vi 的编辑命令

当vi处于命令状态时，用户的按键不回显，被解释成编辑命令，vi大约有一百多个编辑命令。下面介绍的vi命令子集，足可以完成一般的编辑任务。

3.4.1 正文插入命令

在当前光标处插入(insert)正文段，直至按Esc键：命令i。

在命令方式下，按下i键后，进入输入方式。从此以后，输入的文本在屏幕上回显，输入完一整行后，按下回车键，继续输入下一行。输入结束后，按Esc键，退出输入方式，回到命令方式。回到命令方式之后，按键信息不再回显，所有的按键被解释为命令。例如：在输入方式下输入i,有回显，就是输入了一个字母i作为编辑的内容；在命令方式下输入i,无回显，i被解释为编辑命令。同一个按键，在不同的vi状态下，会有不同的解释。现在的全屏编辑器，都属于vi出现之后的新生代软件产品，广泛地使用功能键，不再有这样的命令模式和输入模式的切换，操作起来也更容易。使用过这些软件的操作者，还要注意的是，在Windows类的全屏幕文本编辑器中，输入文本正文时，如果顺便发现，

前面的几行中有个输入错误，就可以很方便的移动光标定位到出错的地方修改。而许多传统的UNIX中，vi必须首先按下Esc键退出输入模式之后再移动光标到其它行，修改后，移动光标回来，再按下i命令（或者a,o,O）重新进入输入方式。在当前输入行的错误修正，使用Backspace键，不需要退出输入方式。

除了i命令外，在当前光标后追加(append)正文段的命令a，也可以进入输入模式，直至按Esc键。

命令o，在当前行之下插入新行(open)，进入输入模式，直至按Esc键。

大写字母O命令，在当前行之上插入新行(open)，进入输入模式，直至按Esc键。

3.4.2 光标移动命令

① 单字符移动

h 光标左移一列

j 光标下移一行

k 光标上移一行

l 光标右移一列

这四个字母在键盘上是相邻的四个按键。在vi中有许多命令可以在命令前加上一个整数，标志这个命令连续执行多少遍，例：

5h 光标左移5列

6j 光标下移6行

23k 光标上移23行

10l 光标右移10列

如前所述，在vi命令状态下的按键命令没有回显。以前面的命令23k为例，用户在按下键2后，没有回显，而且屏幕上没有任何反应，再次按下键3后，仍然没有任何反应，最后按下键k后，光标立即上移23行。

vi中有许多命令允许在命令前加上一个整数，例如：你可以使用3000i命令,在连续键入3000i五个字符过程中，没有回显，vi进入输入方式，这时，输入的内容，可以回显，输入一行文本，按下回车，然后，按Esc退出输入方式，回到命令方式。在按下Esc之后，系统会将刚刚做完的插入操作重复执行3000遍，这样很容易创建一个大个子垃圾文件。

有的终端可以直接使用键盘上的箭头按键代替这四个字母，更便于用户使用。vi被设计成对终端特性的依赖性最小，所以，不依赖于终端的功能键，仅使用键盘的字母、数字和符号键vi就能完成编辑工作。

② 翻页

Ctrl-B: 向后翻页(Backward)

Ctrl-F: 向前翻页(Forward)

Ctrl-U: 向上翻半页(Up)

Ctrl-D: 向下翻半页(Down)

在vi中，把向文件尾方向定义为“向前”，向文件头方向定以为“向后”，这与许多人的习惯不同。在有的PC上的UNIX允许用PgDn键代替Ctrl-F,用PgUp键代替Ctrl-B。

当你所感兴趣的内容正好集中在当前页上部和上页下部时，Ctrl-U命令会派上用场。当你所感兴趣的内容正好集中在当前页下部和下页上部时，Ctrl-D命令会派上用场。

也可以使用下面的命令：

6Ctrl-F：向前翻6页

15Ctrl-B：向后翻15页

③ 将光标移至当前行首 ^

将光标移至当前行尾 $

④ 移到右一个单词 w W，移到左一个单词 b B

使用这种方法,向左/右移动光标,尤其是需要连续移动光标定位到某一位置时,比单字符移动命令更快和更有效。

w,b与W,B的区别是它们对“单词”的定义不同。小写命令的命令w和b,以非字母，数字，下划线之外的所有字符作为“单词”分界符。大写命令的命令W和B,仅以空白符（空格或者制表符）作为“单词”分界符。同前述其它命令类似，也可以使用类似6w 3W 5b 10B命令。

⑤ 移到指定的行

在编写C语言源程序时，编译器通常报出出现语法错误的源程序行号，使用这种方法可以立即将光标定位到需要修改的行。例：

:476 将光标定位于第476行

:1 将光标定位于第1行(文件首)

:$ 将光标定位于文件尾

:$-10 将光标定位于文件倒数第10行

在描述行号时，可以使用圆点(.)代表当前行号，使用$代表最后一行的行号，而且可以使用整数加减法，如前述的最后一例。

⑥ 括号配对命令 %

先把光标移到一个花括号(或圆括号，或方括号)上，按%键，则光标自动定位到与它配对的那一个括号，对编写和检查C语言的源程序非常有用。

3.4.3 设置书签

vi允许设置以单个英文字母命名的最多26个标记(mark)，许多编辑器把这种功能叫“书签(bookmark)”。vi的书签记忆了一个行号。

设置书签的命令是m。例如：按下按键ma,尽管终端上没什么特别的显示信息，但是，vi已经将当前行号记为名字为a的书签。类似的，按下两个按键me将当前行号记为名字为e的书签。最多可以使用26个书签，设置的所有书签，在vi退出后，不再保存。

vi许多编辑命令可以使用命名的书签，将光标移动到指定书签处的命令是'(单引号)

例： 'a 连续按下单引号和字母a，光标会移动到书签a处。

'e 光标移动到书签e处。

3.4.4 删除

① 删除当前字符的命令：x

类似的，命令5x删除从当前光标开始的5个字符。

② 删除当前行的命令：dd

类似的，命令3dd删除从当前行开始的3行。

③ 与光标移动命令相关的删除命令

d'e 从当前光标处删除到书签e处（书签e需要事先用命令m设置好）

d$ 从当前光标处删除到行尾

d^ 从当前光标处删除到行首

dw 删除一个单词

d% 将光标移动到一个括号字符上，删除和它配对的括号括起的段落。

3.4.5 字符替换

替换光标处字符的命令是r。

例：ra命令将当前光标处字符替换为a。

如果希望将当前光标处开始的三个字符依次替换为abc,则需要按命令rarbrc。

替换多个字符的命令是R。

例：命令Rabcdef，然后按Esc键

该命令把从当前光标开始的字符依次替换为abcdef，用Esc来结束多字符替换命令。这类似于以“覆盖”(overwrite)方式进入编辑状态。

3.4.6 取消和重复

① 命令u

取消上一次的编辑操作。例如：误删除了一段正文，用u命令可以恢复到删除前的状态(undo)。再如：把文件中的所有abc字符串替换成xyz字符串，用u命令可恢复到替换前状态。

vi的取消操作，只能回退一次，不像现在Windows上流行的很多编辑软件那样，可以回退很多步。许多新版本的vi对这个问题作了改进。

② 命令.

重复上一次的编辑操作。按圆点键，可以重复上一次的编辑操作。例如：按3dd命令删除了三行，然后按圆点键就再删除三行，如果接着连续按圆点键，则每按一次删除三行。

3.4.7 文件命令

① 存盘退出　　ZZ
存盘退出　　:wq

② 存盘不退出　　:w

③ 不存盘退出　　:q!

④ 读入一个文件插入到当前行之下

```
:r a.c
```

⑤ 写文件

把第50行至文件尾的内容写到文件junk中

```
:50,$w junk
```

如果文件junk事先已经存在，使用下述命令强制把它覆盖掉

```
:50,$w! junk
```

如果编辑了文件之后，无法存盘(例如：文件没有写权限)，那么可以用

```
:w file1
```

将当前编辑好的文件内容存到一个另个文件中。

3.4.8 段落的删除,复制和移动

① 删除(delete)

`:10,50d` 删除第10-50行

`:1,.d` 删除文件首至当前行的部分

② 复制(copy)

`:5,10co56` 复制第5-10行到第56行之下

③ 移动(move)

`:8,34m78` 移动第8-34行到第78行之下

行号描述时除了可以使用圆点代表当前行，$代表最后一行，还可以使用“书签”，例如:下面的命令中'e代表书签e的行号。

```
:'e,.d
```

3.4.9 剪贴板

vi有一个通用的缓冲区和用单个的英文字母命名的26个有名缓冲区，用于保留一些文本。前面介绍的删除命令，vi会在删除了这些信息之后，自动把这些信息保留在一个缓冲区中。下面的命令都会执行两个操作：删除信息和将删除的信息放置到通用缓冲区中。

dd	当前行
3dd	当前行开始的3行
d'e	从当前光标处到书签e处（书签e需要事先用命令me设置好）
d$	从当前光标处到行尾
d^	从当前光标处到行首
dw	一个单词
d%	首先将光标定位到一个括号字符上,从此开始到和它配对的括号处
:1,.d	文件首到当前行的段落

将缓冲区中保留的信息粘贴到光标处，使用p命令(put或paste)。

除了上述的d命令之外，还有“抽取(yank)”命令y,它仅仅把指定的信息拷贝到缓冲区，但不删除它们。用法和d命令类似。

yy	当前行
3yy	当前行开始的3行
y'e	从当前光标处到书签e处（书签e需要事先用命令me设置好）
y$	从当前光标处到行尾

y^　　从当前光标处到行首
yw　　一个单词
y%　　首先将光标定位到一个括号字符上,从此开始到和它配对的括号处
:1,.y　文件首到当前行的段落

在信息保留到缓冲区之后，就可以使用p命令，将它们粘贴到文件合适的位置。

vi除了使用这个默认的缓冲区之外，还有26个用单个英文字母命名的有名字的缓冲区。与缓冲区有关的三个命令是d,y,p，在使用有名字缓冲区时，在这些命令前加两个字符的前缀,一个字符是双引号，一个字符是英文字母代表的缓冲区名字。例如：

"a3dd　删除当前行开始的3行，并把信息保留到a缓冲区中。
"by'e　拷贝当前光标到书签e处内容到b缓冲区中。
"ky%　光标定位在一个括号字符上,拷贝从此开始到和它配对的括号处的段落到缓冲区k中.
"kp　粘贴出缓冲区k中的内容。

3.4.10 其它命令

① 两行合并J （大写字母，Join）

把当前行下面的行合并到当前行。

② 刷新屏幕显示Ctrl-L

当正在编辑一个文件时，由于其它用户给本终端发送的信息显示到屏幕上，或者某一后台进程的输出冲掉了当前的vi显示。那么，在阅读完这些“突然”出现的信息后，按Ctrl-L键，恢复vi原先的屏幕显示。在more命令中也介绍过类似的功能。

③ 状态显示 Ctrl-G

在屏幕最下面一行列出正在编辑的文件的名字，总行数，当前行号，文件是否被修改过等信息。

3.4.11 模式查找

在vi的模式查找命令中，使用“正则表达式”来描述一个字符串模式。命令格式为：

/模式

例：/[0-9][0-9]*

从文件当前位置开始向下查找能与正则表达式[0-9][0-9]*匹配的字符串，找到后，光标将定位于该处。

继续查找命令：n和N。小写字母n键，向下继续查找下一个(next),查到文件尾后，自动折到文件首继续向下查找（循环搜索）。大写字母N，向上继续查找下一个,查到文件头后自动折到文件尾继续向上查找（循环搜索）。

3.4.12 模式替换

① 基本用法

在vi的模式替换命令(substitution)中，也使用“正则表达式”来描述一个字符串模式。命令格式为：

:行号，行号s/模式/替换字符串/g

其中，模式描述使用正则表达式。替换命令结束后在屏幕的最下面的状态行显示替换次数。

【例 3-1】 将 abc 字符串替换为 xyz。

```
:1,50s/abc/xyz/
```

将第1-50行的字符串abc换为xyz，如果同一行内有多个abc字符串，则只替换第一个。

```
:1,50s/abc/xyz/g
```

将第1-50行的字符串abc换为xyz，如果同一行内有多个abc字符串，则替换所有的abc字符串。

```
:s/abc/xyz/g
```

仅将当前行的字符串abc换为xyz，如果当前行中有多个abc字符串，则替换所有的abc字符串。

【例 3-2】在编辑 C 语言源程序时，把一部分行右移/左移四列。

```
:50,80s/^/    /g
```

这样，在50~80行的行首增加了四个空格，从效果上看，第50~80行右移4列。

```
:50,80s/^    //g
```

这样，在50~80行的行首的四个空格被替换成空字符串，从效果上看，第50~80行左移4列（每行首部必须有四个空格）。

② 模式描述使用正则表达式

使用替换命令时应特别注意在描述模式时使用的是正则表达式，尤其是在编辑C语言源程序时，不要忘记在应该加转义符时加上转义符。下面的例子中的转义符\是必不可少的。

【例 3-3】注意 vi 替换命令中用于模式描述的正则表达式中的特殊字符。

(1) 将end.改为middle.。

```
:1,$s/end\./middle./g
```

指定范围为从第一行到最后一行，就是在整个文件中替换。middle后边的圆点前不需要加反斜线。这里的middle.是替换字符串，不是正则表达式，圆点就没有任何特殊含义，因此不能在它前边加反斜线。

(2) 删除每行尾部的一个字符。

```
:1,$s/.$//g
```

将Windows中的文本文件按照二进制格式拷贝到Unix系统中，由于两种系统文本文件存储格式上的差异，在UNIX看来每行尾部都多余一个ASCII码为13的字符，在vi显示的时候，每行尾部显示一个^M，使用上述的命令，将行尾的一个字符替换为空字符串，就可以将行尾的^M消除。但使用:1,$s/\^M//g不能达到目的。

(3) 将a[i]*b[j]替换为x[k]*y[n]。

```
:1,$s/a\[i]\*b\[j]/x[k]*y[n]/g
```

将buf.length/1000替换为 buffer.size/1024。

```
:1,$s/buf\.length\/1000/buffer.size\/1024/g
```

在这里模式和替换字符串中的斜线前加转义符\,以区别于替换命令格式中所必须的斜线。

(4) 在编辑C语言程序的时候，将f[n]替换为f[i]。

使用下面的命令:

```
:1,$s/f[n]/f[i]/g
```

结果，会发现源程序中的f[n]未被替换，程序员会误以为可能打字错误，重新键入命令，发现程序文件中的f[n]仍然未被替换。既然这个命令不能达到期望的结果，着急调试程序的程序员，会手工逐个找到f[n]一个个的改为f[i]。上述的替换命令失败的原因是,正则表达式f[n]与字符串f[n]并不匹配，它只能与fn匹配。在一个较大的程序文件中，如果凑巧有个变量的名字叫fn,那么，由于这段时间程序员把注意力都集中在他试图修改的f[n]，fn被悄悄地替换成了不期望的f[i]。如果编译源程序时未发生任何错误，就会给原本正确的程序引入一个错误。正确的命令是：

```
:1,$s/f\[n]/f[i]/g
```

类似的，将当前行开始到文件尾的段落中所有的x*y替换为x+y,使用下面的命令：

```
:.,$s/x*y/x+y/g
```

结果是，源程序中的x*y被替换为x*x+y,而且，其它地方的y都被替换成了x+y。因为正则表达式x*y可以与字符串y匹配，星号可以是它左边的单字符正则表达式的0次重复。正确的用法是：

```
:.,$s/x\*y/x+y/g
```

③ 替换字符串中的符号&

在替换字符串中特殊字符&,代表被模式所匹配的那部分。

【例 3-4】替换字符串中&符的作用。

设文件当前只含有四行，每行为一个整数，内容为

```
5
2
10
18
```

执行下边的命令

```
:1,$s/[0-9][0-9]*/192.168.24.&    host&/
```

然后，文件内容变为

```
192.168.24.5    host5
192.168.24.2    host2
192.168.24.10    host10
192.168.24.18    host18
```

上述命令描述模式时使用[0-9][0-9]*,如果仅使用[0-9]*，那么，是错误的。由于星号作用于它前边的单字符正则表达式，可以是0次出现，所以[0-9]*可以与任意“空字符串”匹配。

&是C语言中常用的符号，所以要特别注意。将*pointer替换为&record的命令是：

```
:1,$s/\*pointer/\&record/g
```

③ 使用正则表达式的更灵活替换

这种替换方法，扩展了基本的正则表达式，增加\(和\)，在左右圆括号前面缀以转义符。在正则表达式中出现的圆括号，仍然代表它自身。在正则表达式中出现的\(和\)不影响匹配操作。

下边的两个正则表达式匹配相同的字符串。

```
[a-zA-Z_][a-zA-Z0-9_]*->number
\([a-zA-Z_][a-zA-Z0-9_]*\)->number
```

字符串ptr->number和Term370_blk->number都与上述两个正则表达式匹配。使用\(和\)的目的不在于匹配操作，而是在于替换操作。对于能够匹配的字符串来说，用\(和\)标注出来的一段，可以在替换字符串中使用。上例中，在替换字符串中使用\1，就代表了\(和\)标注出来的那段正则表达式所匹配的部分，上面给出的两个字符串的例子中分别是ptr和Term3270_blk。如果正则表达式中有多段这样标记出的部分，在替换字符串中分别使用\2,\3,……。

【例 3-5】使用正则表达式中\(和\)实现更灵活的替换功能。

这种正则表达式替换在其它命令中和其它的非UNIX系统中也广泛使用，Windows的Visual C++编辑器也支持此种替换。UNIX的expr命令，也是用这种方法从一个字符串中提取部分内容，参见6.7.2节的“字符串运算”部分。

(1) 下边的命令完成了number到num的替换，但是这种替换仅限于在一个C语言指针引用之后的number，对于其它地方出现的number不进行替换。

```
:1,$s/\([a-zA-Z_][a-zA-Z0-9_]*\)->number/\1->num/g
```

(2) 将文件中的日期格式“月-日-年”改为“年.月.日”,比如：04-26-1997替换为1997.04.26，使用的命令为：

```
:1,$s/\([0-9][0-9]\)-\([0-9][0-9]\)-\([0-9][0-9]*\)/\3.\2.\1
/g
```

(3) 替换字符串中可以使用特殊的\0代表被整个正则表达式匹配的部分，前面例3-4中的命令:

```
:1,$s/[0-9][0-9]*/192.168.24.&    host&/
```

也可以等价的写作下列的命令:

```
:1,$s/[0-9][0-9]*/192.168.24.\0    host\0/
```

使用正则表达式的替换方法，除了在vi和前面介绍的流编辑命令sed中使用之外，在许多其它的信息系统中，包括非UNIX的系统中，也得到了广泛的使用。Cisco路由器的IOS配置，Windows的Visual C++的编辑器，也采用与上面介绍的方法，进行灵活的替换。这几乎成为了一种事实上的标准，但是，在某些细节上，可能会有些差别。如果你自己设计的软件系统也需要类似的功能，最好也遵循这样多数人都熟悉的成熟规则，以便于简化设计，开发和使用的培训。许多软件开发套件中含有一些正则表达式模式匹配和模式替换的函数库，甚至，可以很方便的从因特网上搜索到完成这样功能的程序源代码。

3.4.13 编辑命令小结

表 3-1 vi 编辑命令

命令	举例	说明
i a o O		(Insert/Append/Open)正文插入命令，进入输入模式，直到按下Esc
h j k l		光标移动一行或一列
Ctrl-F　Ctrl-B		(Forward/Backward)翻页
Ctrl-U　Ctrl-D		(Up/down)翻半页
^ $		光标移到行首和行尾
w W b B		(Word)光标移动一个单词
:n	:25	光标移动到指定行
%		括号配对
m	mq	(Mark)设置一个命名的书签
'	'q	引用书签，光标移动到指定名字的书签处
x		

d	dd d'q dw d%	(Delete)删除
y	yy y% y'q	(Yank)抽取命令，将正文段拷贝到剪贴版
p		(Put)将剪贴板内容粘贴到当前位置
"	"a3dd "ap "by'e "bp "cd% "cp	使用命名剪贴板，用于d,y,p命令，做这些命令前缀
r	rarb	(Replace)替换
R		替换，直到按下Esc
u		(Undo)撤销上一次操作
.		重复上次操作
ZZ		存盘退出
:wq		存盘退出
:w		(Write)存盘不退出
:q!		(Quit)不存盘退出
:r file		(Read)读入文件
:w file	:10,35w fa	(Write)写入到文件
:行号,行号d	:.$d	(Delete)删除
:行号,行号co行号	:1,30co42	(Copy)拷贝
:行号,行号m行号	:70,100m43	(move)移动
J		(Join)两行合并
Ctr-L		刷新屏幕显示
Ctrl-G		显示当前状态（文件名，行号等）
/模式	/Test	模式查找
n N		(Next)查找下一个
:行号，行号s/模式/替换字符串/	:1,$s/abc/xyz/g	(Substitute)模式替换

第 4 章 UNIX 的文件和目录

4.1 文件和目录的层次结构

UNIX系统通过目录管理文件，文件系统组织成树状结构，目录中可以含有多个文件，也可以含有子目录。这些特点，和普通的Windows系统类似。UNIX系统中，路径名分割符用正斜线/,而不是Windows中的反斜线\。下面是一些常见的目录和文件。与系统有关的一些主要目录的取名和在层次结构中的位置，几乎在所有UNIX系统中都相同。

/unix	程序文件，unix内核。
/etc	供系统维护管理用的命令和配置文件。例如： 文件/etc/passwd:存放的是用户相关的配置信息。 文件/etc/issue:登录前在login之上的提示信息。 文件/etc/motd:存放登录成功后显示给用户的信息。 文件系统管理的程序有fsck,mount,shutdown等等。 许多系统维护的命令，在不同的UNIX系统之间区别很大。
/tmp，/usr/tmp	存放临时文件。
/bin	系统常用命令,如ls,ln,cp,cat等。
/dev	存放设备文件,如终端设备文件,磁带机,打印机等等.
/usr/include	C语言头文件存放目录。
/usr/bin	存放一些常用命令,如ftp,make等。
/lib,/usr/lib	存放各种库文件，包括C语言的链接库文件，动态链接库，还包括与终端类型相关的terminfo终端库，等等。静态链接库文件有.a后缀，动态链接库文件的后缀不是.DLL，而是.so。UNIX很早就广泛地使用动态链接库，静态链接库逐渐过时。.a取名于archive(存档)，.so取名于shared objects(共享对象)。
/usr/spool	存放与用户有关的一些临时性文件,如：打印队列,已收到但未读的邮件等等。

4.2 文件和目录的命名

UNIX的文件和目录的命名规则：

(1) 名字长度：

现在的UNIX都支持长文件名，文件名长度的最大值都在200以上，早期的UNIX不支持长文件名，但至少可以支持长度为14个字符的文件名。

(2)取名的合法字符：

除斜线外的所有字符都是命名的合法字符，空格，星号，甚至不可打印字符也可以做文件名。斜线被用作了路径名分割符，不许用于文件名。一个字节的取值0~255之中，47是斜线的ASCII码，不可作为文件名，ASCII码0用作C语言的字符串结束标志，其余的254种取值都可以作为文件名。尽管这些取值都是“合法”的，但是，在为一个文件或目录取名时，尽量避免使用一些特殊字符，以免使用时不方便。

(3)大小写字母有区别

UNIX和Windows系统不同，在UNIX中文件名字母的大小写是严格区分的，这点，和C语言源程序类似。例如：makefile，Makefile，MAKEFILE是三个不同的文件名。

4.3 shell 的文件名通配符

4.3.1 规则

UNIX的文件名通配符是由shell程序解释的，不同的shell会有不同的文件名通配符规则。例如：在C-shell中，波浪线~符号代表的是用户的主目录路径，而在B-shell中，字符~没有这种功能。但是，对几乎所有的shell来说，表4-1列出的有关文件名通配符的规则几乎都一致。

表 4-1 常用的 shell 文件名通配符

符号	含义
*	匹配任意长度(包括空字符串)。如：try*c匹配try1.c，try.c，try.basic。例外的情况是当文件通配串的第一个字符为*时，*不匹配以圆点（.）开头的文件。例：*file匹配文件file，tempfile，makefile，但不匹配.profile文件。
?	匹配任一单字符。例如：p?.c可以匹配p1.c，p2.c，pa.c。
[]	匹配括号内任一字符,也可以用减号指定一个范围。例如:[A-Z]*匹配所有名字以大写字母开头的文件，*.[ch]匹配所有含.c或者.h后缀的文件，[Mm]akefile匹配Makefile或者makefile。

4.3.2 与 DOS 文件名通配符的区别

1．UNIX的文件名通配符要比DOS中的要严格的多，无二义性。

设当前目录下有两文件xcom.exe和xcom.obj，DOS中DIR XCOM*命令会列出两个文件，而DEL XCOM*删不掉任一个文件。DOS也不允许使用类似的通配串*temp*。在UNIX中,文件名通配符允许用于任何命令,而DOS中只能用于dir/del/copy等有限的

几个命令中。在UNIX中，命令cat *.c可以列出所有的.c文件内容，而在DOS中，命令TYPE *.c不可。现在的Windows系统中，这种情况有所改观。

2．关于文件扩展名。

DOS中*.*匹配所有文件，UNIX中*.*要求文件名中必须含有圆点，否则不匹配，如：*.*与makefile不匹配。UNIX中并没有把文件名中的圆点做特殊的处理，它和普通的字符有着相同的地位。UNIX文件名处理中，没有所谓“扩展名”的待遇。UNIX的某些程序，象C语言编译器，要求它处理的文件必须具有特定的文件名后缀，但那是编译程序自己的约定，而与UNIX系统无关。

3．匹配子目录中的文件。

在UNIX中可以使用*/*.[ch]通配符，匹配当前目录下所有一级子目录中文件名后缀为.c和.h的文件，在DOS中不许。

4.3.3 文件名通配符的处理过程

UNIX和DOS的文件名通配符看起来类似，但是，在处理过程上有所不同。了解shell的工作机理，会更深入更准确地把握shell的特性，便于正确地使用它，也能够理解shell的某些行为。UNIX处理文件名通配符的过程分几步：

(1) 在shell提示符下，从键盘输入命令，输入的命令被shell所接受。

(2) shell对所键入的内容作若干种加工处理,其中含有对文件名通配符的扩展工作，生成结果命令。

(3) 执行生成的结果命令。

shell程序也是在操作系统内核支持下的应用级程序，与普通用户编写的源程序编译链接之后产生的可执行程序在系统中有相同的地位。甚至，用户也可以根据自己的喜好编写自己的shell程序。从设计shell软件的程序员角度看，在shell提示符下输入命令，会在程序中得到一个字符串的输入，这便是第一步；随后，将得到的这个字符串进行加工处理，其中，含有对文件名通配符的展开工作，生成结果命令，这是第二步；第三步，调用操作系统的系统调用，创建新的进程执行命令，并把参数传递给新进程。

shell在第二步中，含有文件名生成工作，把用空格分开的每一段作为一个“单词”，扫描每个词，从中寻找 * ？ []。如果其中之一出现，则该词被识别为一个文件名通配符，用与文件名通配符相匹配的文件名表取代该词，可以匹配多个名字时，按字母序排列多个名字。如果没有找到与文件名通配符相匹配的文件名，在B-shell 中,不改变该词；在C-shell中，产生错误。

【例 4-1】体验 shell 对文件名通配符的展开处理。

(1) 设当前目录下只有try.c, zap.c, arc.c三文件，在shell提示符下，键入命令

```
cat *.c
```

那么，shell根据当前目录下的所有文件的文件名集合，将*.c扩展为arc.c try.c zap.c，扩展后的多个文件名按照字典序排列。这样，cat *.c被加工成了cat arc.c try.c zap.c，实际执行加工之后的命令。就是说，键入命令cat *.c与手工键入命令cat arc.c try.c zap.c的效果是完全等价的。从cat命令的角度来说，都是指定了三个文件作为处理对象,cat程序在执行的时候，已经看不到*.c，它看到的是三个文件名。

(2) 设当前目录下有0131.rpt, 0130.rpt,wang.mail, lee.mail四个文件。在两个mail文件中查找数字串的命令为：

```
grep '[0-9][0-9]*' *.mail
```

这里，[0-9][0-9]*的两侧，应当用单引号括起来。用单引号括起来的部分，shell就不再进行文件名展开。如果漏掉单引号，错误的使用了下面的命令：

```
grep [0-9][0-9]* *.mail
```

那么，经过shell替换之后，命令变成了：

```
grep 0130.rpt 0131.rpt lee.mail wang.mail
```

键入前边的命令，与手工键入上述命令效果是完全相同的。这样，grep并不了解发生的这些替换，按照grep命令的语法格式，grep在执行的时候，就会在0131.rpt,lee.mail,wang.mail三个文件中搜寻与正则表达式0130.rpt匹配的字符串。这样的结果，不符合操作员的初衷，系统的行为显得有些怪异。因此，输入命令中必需的单引号是必不可少的。关于单引号等元字符的介绍，在“B-shell及编程”一章6.5.9节中详细介绍。

(3) 鉴于shell对文件名通配符的处理方式，文件名通配符可以使用在任何命令中，如：vi m*e 替换成 vi makefile，而用户的输入cd *work.d 替换成 cd configure_network.d。cd是改变当前工作目录的命令，要求后面只能有一个参数，如果文件名展开后cd有了两个参数，cd命令就会失败。只要所选择的文件名通配符只能匹配一个名字，cd就可以正确工作。使用文件名通配符，可以简化一些命令的输入，尤其是那些较长的名字。

在UNIX中，文件名通配的过程由shell完成，命令本身并无文件名通配的功能，而DOS中，文件名通配的过程由命令自己完成，如:DIR *.C由命令DIR对文件名通配符作解释，DEL *.C由DEL命令解释，因而可能会产生二义性，不同命令的解释结果不同，而且，也只能用于有限的几个命令中。

从shell对文件名通配符的处理方式上，也不难看出为什么UNIX的许多命令允许指定多个文件作为处理对象，如前面介绍过的cat, od, wc, grep等命令，本章介绍的文件管理命令也一样具有这样的特点。这些命令的格式如此设计，也是为了能够配合shell的文件名通配符处理方式，以提供更强的功能。

4.3.4 验证文件名匹配的结果

【例 4-2】从程序员的角度理解 shell 对通配符的处理。

编写一个很简短的C语言程序，站在与cat命令同等的位置，观察一下shell对文件名通配符的处理。如果将这个程序进一步扩展，增加相应的处理，就可以实现诸如cat,grep等命令。

用vi 编辑下列文件arg.c。

```
int main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < argc; i++)
        printf("%d: [%s]\n", i, argv[i]);
}
```

编译，链接，生成可执行文件。使用命令：

```
cc arg.c -o arg
```

或者使用命令

```
make arg
```

在当前目录下生成名为arg可执行文件,没有Windows系统中那样的.EXE扩展名。执行这个程序的命令，如：

```
./arg abc ABCDEF
```

在DOS系统中，直接键入ARG abc ABCDEF命令，系统会首先在当前目录下搜索ARG.EXE。在UNIX系统中，如果直接键入arg abc ABCDEF，默认情况下，shell会只在系统规定的目录中搜索指定文件arg,搜索不到也不会到当前目录下搜索。因此，需要用命令./arg显式的指定程序文件存储的路径为点目录(./)，shell就会到./目录下搜寻执行文件arg。

目录./就是当前目录，目录../是上级目录,这一点DOS和UNIX相同。用户可以通过设置shell环境变量PATH将当前目录增加到系统的搜索路径中去。一般默认情况下不搜索当前目录。设置方法在6.3.5节中介绍。

在C语言中，从主函数main的两个参数，可以获得命令行参数的内容。第一个参数argc是命令行参数的个数，第二个参数argv可以理解是一个指向数组的指针，就是说，argv是一个指针，所指向的内存位置存放一个数组，数组的每个元素是个指针，指向一个字符串。数组中有效元素个数是argc个，argv[0]～argv[argc-1]分别是字符串指针。按照C语言的字符串格式，每个字符串结束处都有字节0标志。

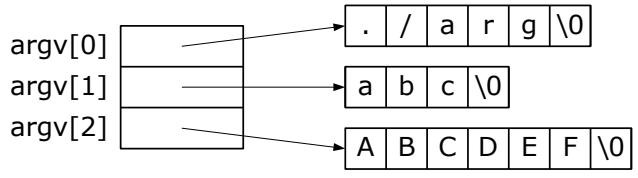

图 4-1 main 函数的 argv 参数的存储结构

针对输入的./arg abc ABCDEF，argv数组的布局如图4-1所示。在一个32位系统中，一个指针一般占用4字节，存储上图4-1中数据至少需要29字节。这些数据安排在进程的堆栈底部，在程序arg的main函数执行之前，系统根据命令行参数已经做好了上述的内存安排，在main函数中直接使用即可。其中，argv[0]字符串是命令自身，其余的是命令行的参数。

下面是这个命令上机执行的情况：

```
$ cat arg.c
int main(int argc, char *argv[])
  {
     int i;
     for (i = 0; i < argc; i++)
        printf("%d: [%s]\n", i, argv[i]);
  }

$ cc arg.c -o arg
$ ./arg abc ABCDEF
0: [./arg]
1: [abc]
2: [ABCDEF]
$ ./arg *.[ch]
0: [./arg]
1: [arg.c]
2: [auth.c]
3: [ccp.c]
4: [chap.c]
……
$ ./arg '*.[ch]'
0: [./arg]
1: [*.[ch]]
```

将执行结果与同样的arg.c在DOS下执行结果相比较，在DOS系统中，.\arg *.c的执行时，不会进行文件名展开，执行结果为：

```
0: [.\arg]
1: [*.c]
```

4.4 文件管理

4.4.1 ls:文件名列表

命令ls类似DOS系统中的DIR命令，列出目录中的文件名。ls之后可以跟0个到多个名字。

(1) 不给出任何名字时，列出当前目录下所有文件和子目录。

(2) 名字为文件时，列出文件名。

(3) 名字为目录时，不列出目录名，而是列出目录下的所有文件和子目录。

(4) 在同一命令行中可以指定多个名字，这点可以配合shell文件名通配符工作。

例如:

```
$ ls
(ls 未指定名字，列出当前目录下的所有文件和子目录名)
arg       bak.d     document   pipe1     xsh2.c
arg.c     config.ap  inc       xsh2
$ ls arg.c
(为 ls 指定一个实参，是普通文件)
arg.c
$ ls /
(为 ls 指定一个实参，是目录，列出根目录下的所有文件和目录)
TT_DB             ftphome1         lpp             tftpboot
WebSM.pref        ftphome2         mbox            tmp
audit             ftphome3          mnt            u
bin               home              nsmail         unix
bosinst.data      image.data       opt              usr
cdrom            inst.log          proc             var
dead.letter       lab               sbin             websm.log
dev               lib               smit.log         websm.script
etc               lost+found        smit.script      wsmmon.dat
$ ls arg*
(为 ls 指定多个实参，均为普通文件)
arg    arg.c
$ ls /usr/include/*
(为 ls 指定多个实参，既有文件也有目录)
/usr/include/60x_regs.h           /usr/include/lvm.h
/usr/include/NLchar.h             /usr/include/lvmrec.h
/usr/include/NLctype.h            /usr/include/macros.h
/usr/include/NLregexp.h           /usr/include/malloc.h
/usr/include/NLxio.h              /usr/include/math.h
/usr/include/a.out.h              /usr/include/mbstr.h
/usr/include/acl.h                /usr/include/memory.h
/usr/include/aio.h                /usr/include/mesg.h
.
.
.
/usr/include/Motif2.1:
Dt   Mrm  Xm   uil

/usr/include/Mrm:
MrmAppl.h    MrmDecls.h   MrmPublic.h  MrmWidget.h  MrmosI.h
```

ls有几十个选项，控制每个文件的列表格式，以及列表的范围包括那些文件。

(1) -a：列出所有(all)项，包括那些名字以圆点打头的文件，默认情况下，名字以圆点打头的文件不被列出。

```
$ ls -a
.         .cshrc    arg       bak.d     document   pipe1     xsh2.c
```

```
..        .profile   arg.c     config.ap  inc       xsh2
```

(2) -R：递归地列出碰到的子目录(Recursion)。就是说，在列出子目录时，如果子目录还有子目录，就一直如此追究下去，直到所有的分支目录。如：ls -R / 将列出系统中所有文件，这一命令执行时间会很长，列出的内容也很多。ls -R . 列出当前目录下所有文件。

```
$ ls -R .
arg       bak.d     document   pipe1     xsh2.c
arg.c     config.ap  inc        xsh2
./bak.d:
1s.c     cld.c    cld2.c    client.c  clock.c

./document:
manual.pdf  paper.pdf
```

(3) -F：标记(Flag)每个文件。若列出的是目录，就在名字后面缀以/；若列出的是可执行文件，就在名字后面缀以*；若列出的是符号连接文件，就在名字后面缀以@；若列出的是管道文件，则名字面后缀以|；若列出的是普通文件，则名字面后无任何标记。ls命令允许同时指定多个选项，ls -aF命令就是同时使用两个选项a和F。

```
$ ls -F
arg*       bak.d/     document/   pipe1|     xsh2.c
arg.c      config.ap*  inc@       xsh2*
$ ls -aF
./         .profile    bak.d/     inc@       xsh2.c
../        arg*       config.ap*  pipe1|
.cshrc     arg.c       document/   xsh2*
```

(4) -i：列出文件的i节点号。例：

```
$ ls -i
184323 arg      184393 config.ap 184326 pipe1
184321 arg.c    206873 document  184391 xsh2
184327 bak.d    184325 inc      184392 xsh2.c
```

arg文件的i节点号为184323，xsh2.c文件的i节点号为184392。关于i节点，在4.7和4.8节有详细的介绍。

(5) -d: 若实参是目录，则只列其名字(不列内容)。

例：设当前目录结构如图4-2所示。

ls abc*　列出abc,f1,f2,abc.rpt四个文件。

ls -d abc*　列出abc,abc.dir,abc.rpt三个文件。

```
$ ls *
abc     abc.rpt

abc.dir:
f1  f2
$ ls -d abc*
```

```
abc     abc.dir  abc.rpt
```

无参数的ls命令，与ls *执行结果并不同，与ls -d *功能相同。

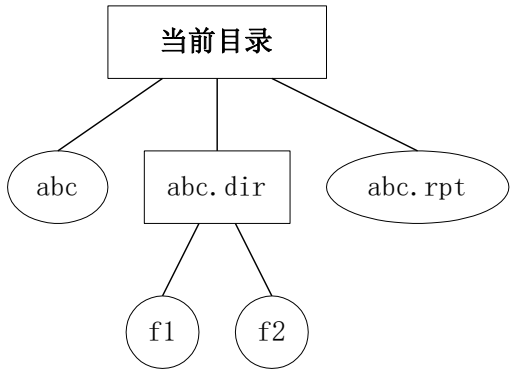

图 4-2 文件和目录结构举例，展示 ls 的-d 选项的功能

(6) -l (long)长格式列表。例如：

```
$ ls -l
-rwxr-xr-x   1 jiang   usr          4423 Aug 26 13:31 arg
-rw-r--r--   1 jiang   usr           116 Aug 26 13:31 arg.c
drwxr-x---   2 jiang   usr          1536 Aug 26 14:00 bak.d
-rwxr-xr-x   1 jiang   usr           128 Aug 26 13:38 config.ap
drwxr-x---   2 jiang   usr           512 Aug 26 13:59 document
lrwxrwxrwx   1 jiang   usr            12 Aug 26 13:33 inc -> /usr/include
prw-r--r--   1 jiang   usr             0 Aug 26 13:33 pipe1
-rwxr-xr-x   1 jiang   usr          8661 Aug 26 13:36 xsh2
-rw-r--r--   1 jiang   usr          1076 Aug 26 13:36 xsh2.c
```

第1列为文件属性。

第1列的第1字符描述文件类型。常见的文件类型有：- 普通文件，d 目录文件，l 符号连接文件，b 块设备文件，c 字符设备文件，p 命名管道文件(pipe)。

第1列的第2~10字符，描述文件的访问权限。其中第2~4字符描述文件所有者对文件的访问权限；第5~7字符描述文件所有者的同组用户对文件的访问权限；第8~10字符描述其他用户对文件的访问权限。权限描述方式为rwx，分别对应读权限,写权限,可执行权限,相应位置显示的是字母，表明有相应权限，显示减号，表明没有相应的权限。

第2列是文件的link数,涉及到此文件的目录项数。

第3列,第4列分别是文件主的名字和组名。

第5列是文件的大小。对于普通磁盘文件，列出文件内容大小；对于目录，列出目录表大小（而不是目录下各文件长度之和）；对于符号连接文件，列出符号连接文件自身的长度；对于字符设备和块设备文件，列出主设备号和次设备号；对于管道文件，列出管道内的数据长度。

第6列是文件最后一次被修改的日期和时间。

第7列是文件名。对于符号连接文件，还附带列出符号连接文件的内容。

例：

ls -l 将以长格式列出当前目录下所有文件

ls -l . 列出当前目录下所有文件,可查知各文件权限

ls -ld . 列出当前目录自身,可查知当前目录自身的权限

ls -l * 列出当前目录下所有文件（但不包含目录）和一级子目录中所有文件名

ls -ld f* 与DOS中的DIR F*.*功能类似

ls -Flad rpt* 可以在同一命令中指定多个选项

ls -l | grep '^d' | wc -l 统计当前目录下有多少子目录

4.4.2 cp:拷贝文件

格式1:cp *file1 file2*

格式2:cp *file1 file2 ... filen dir*

其中，*file1*, *file2* ... *filen*为文件名(n>0), *dir*为已有目录的名字。命令的第一种格式,把文件*file1*拷成*file2*。若*file2*存在,则覆盖，否则，创建*file2*。cp命令的第二种格式,将*file1*~*filen*一个或多个文件拷至目录*dir*下。当n为1时，第二种格式的命令看起来和第一种格式相同，但是完成的操作不同。

例如： cp *.c backup.d

其中 backup.d 为一个子目录,符合第二种格式。此命令与DOS中copy *.c backup.d 执行结果相同，但执行过程不同，UNIX的文件名通配符由shell负责展开。

cp fa.c backup.d命令也符合上述的格式2，把文件fa.c拷贝到现有的一个目录backup.d中。

【例 4-3】cp 命令和 DOS 的 COPY 命令的区别。

设文件目录结构图4-3所示,

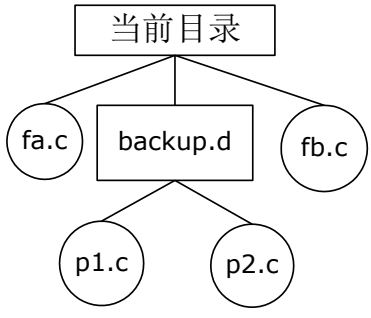

图 4-3 文件和目录结构举例，展示 cp 命令的处理过程

在DOS中，如果希望将backup.d下的两个文件p1.c和p2.c复制到当前目录，使用命令COPY P?.C。但是，在UNIX中执行cp backup.d/p?c命令,经过shell完成文件名生成之后，实际执行：

cp backup.d/p1.c backup.d/p2.c

执行结果是p1.c和p2.c没有拷贝到当前目录,而是文件p1.c覆盖掉文件p2.c。

错误的用法cp backup.d/p?.c会带来不期望的后果。如果运气好的话，backup.d目录下还有p3.c，展开后cp有三个参数，既不符合格式1，也不符合格式2,命令无效，拒绝作任何操作。

将上述两文件拷贝到当前目录的方法，使用cp命令的格式2，最后一个参数圆点是当前目录:

```
cp backup.d/p?.c .
```

再如: cp /usr/include/*.h .

4.4.3 mv:移动文件

格式1: mv *existing-file-or-dir new--name*

格式2: mv *file1 file2 ... filen directory*

mv命令的第一种格式可以将文件和目录改名，或移动到另个目录，并使用新名字。第二种格式，*directory*是一个已存在的目录，可以把一个或多个已有的文件或目录移动到目录*directory*中，并保持原先的名字。其中的*file~filen*可以是已有文件，也可以是已有的目录。

从后面介绍的文件系统的存储结构中可以得知，在同一个文件系统内的mv操作比先拷贝成新文件再删除旧文件的方法更高效，因为mv操作只需要修改目录表，不需要访问文件内容。

4.4.4 rm:删除文件

格式:rm *file1 file2 ... filen*

例:
```
rm core a.out
   rm *.o *.tmp
```

(1) 选项-r

允许*file1~filen*是已有的文件名或者目录名。当它是一个目录时，递归地(recursively)删除子目录中的所有文件和目录。经常使用这一命令删除一棵已有的目录树。

(2) 选项-i

交互方式(interactive)。每次删除前，经过操作员确认,选择y，删除，选择n不做删除。例如:当前目录下产生了一个名字怪异的文件。UNIX中除了斜线和字节0之外的所有其它取值1~255的字节都是合法文件名，在用户自己的C语言程序中创建新文件，如果程序有错，文件名字符串没有赋初值，可能会是随机值，就有可能创建出名字怪异的文件。这种文件，可以用命令rm -i *删掉，删除前正常的文件一律回答n。或者，如果凑巧名字中有字符A的话，使用rm -i *A*命令。

(3) 选项-f

强迫删除(force)。只读文件也可以被删除。rm在删除文件时,如果文件没有写权限,则以八进制形式显示权限的整个设置,并后跟一个问号提示用户进行确认。若用户答y,则删除,否则,保留文件。当选用-f选项时,将强迫删除文件,不管它有无写权限,并且不提示用户。

【例 4-4】rm 命令选项的功能。

(1) rm -r backap.d

删除当前目录下的整个子目录backup.d。

(2) rm -rf xx*

清除所有以xx打头的文件。程序员应当为一段时间内临时使用的一些文件的取名符合某一特定规律，例如名字的前两个字母为x，以便于清理临时文件。

(3) rm -i *.test

有选择地删除若干文件。

(4) rm * .bak

误操作，星号之后多出了一个空格，那么，经shell文件名展开之后，rm会忠实的删除所有文件,并且可能会通知你，企图删除的文件.bak不存在。

【例 4-5】处理以减号-打头的文件。

设当前目录下只有a,b,c三个文件。执行命令rm -i，rm会将-i理解为命令选项，没有指定文件名,不能删除任何文件。执行命令who > -i， 将生成文件-i，因为这个名字符合文件命令规则。但是，使用rm -i 不能删除文件-i。

如果使用命令rm *， 文件名通配符展开后的命令变成rm -i a b c，那么，rm将提示删除文件a,或b,c,唯独不提示删除文件-i。

如果使用命令ls -l *，文件名通配符展开后的命令变成ls -l -i a b c，那么，ls命令以长格式列出a,b,c三个文件并附带其i节点号,但不列出-i文件。

如何处理这种情况？在很多UNIX命令中，这也是个普遍存在的问题。 UNIX的许多命令，都允许使用选项，选项是可选择的，并不是每次使用该命令时都需要所有选项。按照惯例，选项都以减号开头，当程序从左向右扫描所有命令行参数的时候，如果命令行的参数的第一个字符是减号，程序就将它后边的内容理解为选项，否则，会理解为要处理的对象。当要处理的对象以减号开头时，例如文件取名-i，大部分命令都遵循的惯例是用一个独立的命令行参数--显式地标志命令行选项的结束，从--参数之后的任何命令行参数，都不再解释为选项。这样命令就可以识别以“-”开头的文件名。许多UNIX命

令中都采用这种方式显示地区分选项和处理对象，这些命令有cp，ls，mv，rm，grep，set，等等。如果要设计自己的应用程序，也使用选项，最好也遵守这些惯例。

命令 rm -- -i 删除文件-i。当然，也可以用 rm ./-i的方式，删除当前目录下的文件-i。在C语言源程序文件中搜索-delta字符串，命令grep -delta *.c 会把-delta理解为grep的选项，这是个grep不能识别的选项，导致grep抱怨你提供了不认识的选项，因而无法工作。正确的用法该是：

```
grep -- -delta *.c
```

4.4.5 find:查找文件

find命令在一个指定的范围内查找符合条件的文件或者目录，然后执行相应的动作。find功能很强，描述“条件”和“动作”选项较多。

例：find ver1.d ver2.d -name '*.c' -print

查找范围为：两个目录ver1.d和ver2.d，find的查找范围可以为1个或者多个目录。条件为与名字*.c匹配，这里的*.c应当用单引号括起。对于在规定范围内找到的符合条件的文件，执行的动作是把查找到的文件的路径名打印出来。

find从指定的路径开始,递归地查找其下属的所有子目录,凡满足条件的文件或目录,执行规定的动作。

① 关于“条件”的选项有

-name选项: 文件名的匹配,允许使用文件名通配符 * ? []，应当把这个文件名通配符描述串传递到find程序中，因此，应当用引号括起来，以免被shell展开。

-type 类型 f:普通文件，d:目录，l:符号连接文件，c:字符设备文件，b:块设备文件，p:管道文件，如：-type d。

-size ±n[c]: 文件大小，正号表示大于，负号表示小于，无符号表示等于，其它与数量有关的选项，也采用这样的方式。例：

-size +100　　表示长度大于100块(1块=512字节,或者1024字节，与具体的系统有关)。

-size -100　　长度小于100块。

-size +100000c　表示长度大于100000字符(100k字节)。

-size -100000c　表示长度小于100k字节

-mtime ±n　文件最近修改时间(modify),单位为天。如：-mtime -10，表示10天之内曾经修改过。

-atime ±n　文件最近访问(access)时间,单位为天。文件“修改(Modify)”指的是对文件的内容进行了修改。文件“访问(Access)”指读取了文件的内容，或者，文件作为一个程序被执行。仅仅写文件，文件的“访问”时间不变。

find还有许多其它的条件选项，可以指定文件主(-user)，组（-group）,文件的link数(-links),i节点号(-inum),等等。

find中可以指定多个条件，罗列出的多个条件默认为多条件的“与”，参见下面的例4-6(2)(3)。可以用!和-o表示条件“非”和两条件的“或”,还可以使用圆括号表示更复杂的复合条件，参见下面的例4-6(5)(6)。

② 关于“动作”的选项：

-print　打印查找到的符合条件的文件的路径名。

-exec，-ok　对查找到的符合条件的文件执行某一命令。

find命令稍微有些复杂，初学UNIX的操作员，在使用find命令的时候，由于对shell的特性还不是很了解，常常会在键入命令时缺少了必需的引号或者空格等符号，导致命令执行失败，达不到期望的结果。许多UNIX的书籍中，由于印刷格式等原因，对空格的作用不够突出，应当注意。

find命令由于可以用-exec和-ok选项引入其它的命令，允许对搜索到的文件执行所需要的操作。多命令的这种组合，使得find命令可以配合其它命令提供灵活又强大的功能，因而find命令成为一个非常重要的命令。

【例 4-6】几个使用 find 命令的例子。

(1) find . -type d -print

从当前目录开始查找，查找所有目录，找到后，打印路径名。这种方法可以按层次列出当前的目录结构。

(2) find / -name 'stud*' -type d -print

指定了两个条件。名字与stud*匹配，类型为目录。为两个条件的“逻辑与”。同时符合这两个条件的的项目，打印路径名。

(3) find / -type f -mtime -10 -print

从根目录开始检索最近10天之内曾经修改过的普通磁盘文件。

(4) find . -atime +30 -mtime +30 -print

从当前目录开始检索最近30天之内既没有读过，也没有写过，而且也没有被当作命令执行过的文件。这种方法可以筛选出一个时间周期内不活跃的文件。

(5) find . ! -type d -links +2 -print

从当前目录开始检索link数大于2的非目录文件。

条件的“非”用!。注意：!号与-type之间必须保留一空格。

(6) find / -size +100000c \(-name core -o -name '*.tmp' \) -print

从根目录开始检索那些文件尺寸大于100K字节，并且，文件名叫core或者文件名有.tmp后缀的文件。在这个命令中，使用了两条件的“或”(-o)及组合(圆括号)，注意：不要遗漏了所必需的引号，反斜线和空格，尤其是圆括号前和圆括号后。圆括号是shell的特殊字符，因此，必须在圆括号字符前面加转义符\。shell在见到转义符之后不再对紧跟其后的字符进行特殊解释,才使得find命令可以看到这些括号。左括号和右括号必须作为一个单独的命令行参数传递给find命令，因此,必需的空格不可缺少。读者可以尝试着将find命令替换成前面介绍的简单程序arg,提供跟find一样的参数，观察一下不同的参数书写格式，给find带来的不同感受。上述命令也可以写作下面的形式。关于shell的这些问题，后面的6.4~6.5节详细介绍。

```
find / -size +100000c '(' -name core -o -name \*.tmp ')' -print
find / -size +100000c  \( -name core –o -name \*.tmp ')' -print
```

(7) find / -name make -print -exec ls -l {} \;

在-exec及随后的分号之间的内容作为一条命令，上述命令以长格式列出文件。由于在shell中分号有特殊含义,因此,在此前面加\以取消shell对分号的特殊解释,使得find命令可以见到分号。{}代表所查到的符合条件的路径名。注意，两花括号间没有空格，而{}之后的空格不可省略，这些格式上的细节，命令输入时如果不正确，就无法完成期望的动作。-ok选项和-exec选项类似，只是在执行指定的命令前等待用户确认。再如：

```
find . -name '*[ch]' -exec cat {} \;
find / -size +100000c \( -name core -o name '*.tmp' \) -ok rm {}
\;
```

find命令的“递归式”穷尽搜索一个子目录，-exec选项又提供了处理的灵活性，使得在find框架下，组合其它的命令，提供许多更方便的功能。例如，grep就没有类似rm命令的-r选项，grep无法递归式的搜索一个目录中的所有文件，寻找一个字符串，使用find命令和grep命令的组合，可以达到这样的目的。

```
find /usr/include -name '*.h' -exec grep AF_INET6 {} \;
```

这样就可以在/usr/include目录下属的所有子目录中的C语言头文件里检索字符串AF_INET6。

(8) 将当前目录下所有文件拷贝到目录../bak中去，可以执行cp * ../bak命令。shell在执行这一命令时，会将*扩展为当前目录下的所有文件名，但是，如果当前目录下文件数目太多，shell在进行文件名展开的时候，展开后的名字太多，有的系统中就会执行失败。在后面章节的图7.1“用户堆栈底部”的解释，说明了这种失败的原因。这时，可以使用这样的命令：

```
find . -type f -maxdepth 1 -exec cp {} ../bak \;
```

选项-maxdepth 1将find的搜索深度限制为最多1层，如果当前目录有子目录，就不再检索子目录。

4.5 目录管理

4.5.1 路径名

(1) 绝对路径名与相对路径名

绝对路径名与相对路径名的概念，与Windows系统中的概念相同。

(2) 当前工作目录。

UNIX中，每个进程都有一个当前工作目录。当前工作目录是进程属性的一部分,不象Windows那样，每个逻辑盘都对应一个当前工作目录并且有 “当前盘”的概念。UNIX没有当前盘的概念。

(3) 文件.与..

在目录表的存储结构中，确实有两个文件,一个名字叫做.另外一个叫..。分别代表当前目录和上一级目录。设置这两个目录项的目的是为了便于用户使用，这两个目录项只允许由系统创建和删除。

(4) 主目录(Home Directory)

UNIX是一个多用户系统，每个用户都对应一个主目录。不同的用户有不同的主目录。主目录的设置可以从/etc/passwd文件中看到，一般在系统管理员创建用户时指定他的主目录和登录shell类型。用env命令查环境变量HOME的值，可以看到当前用户的主目录路径。

4.5.2 pwd:打印当前工作目录

pwd命令打印当前工作目录的名字(Print Working Directory)。

4.5.3 cd:改变当前工作目录

cd命令用于修改当前工作目录的路径(Change Directory)。

例:cd /usr/include

cd / 斜线/前必须要有空格,不象DOS那样可以省略，直接写作cd\。

cd .. 返回上级目录。

cd命令之后未给出任何参数，在DOS中,打印出当前工作目录;而在UNIX中,默认回到用户的主目录。

cd命令是shell的一个内部命令，而且它只可能是内部命令。用于修改“shell进程”的进程属性中的当前工作目录。在C语言程序中可以用系统调用chdir函数修改进程的当前目录。

4.5.4 mkdir:创建目录

```
例:mkdir  sun/work1.d
  mkdir tmp
```

mkdir除创建目录外,还在所创建的目录中自动建立文件.与..。

4.5.5 rmdir:删除目录

```
例: rmdir   sun/workl.d
    rmdir   tmp
```

要求被删除的目录除.与..外没有其它文件或目录。否则，rmdir命令会失败。删除一个含有文件或者子目录的目录树,可以直接使用命令rm,例如：rm -r sun/workl.d

4.5.6 cp:复制目录

cp命令有个选项-r,用于递归(Recursively)的复制一个目录。就如同rm命令的-r选项一样。命令格式为： cp -r *dir1 dir2*

(1) 若*dir2*不存在,则新建子目录,并将*dir1*下所有文件拷入。

(2) 若*dir2*已存在,则将所有文件拷入目录*dir2*。

还可以使用选项-v, 冗长(verbose)方式,执行时列出所拷贝的文件名。目录拷贝常常被用于备份目录的目的。备份目录时，cp命令的一个有用选项是-u,用于增量拷贝(update)。-u选项在拷贝*dir1*目录中的一个文件时，检查在*dir2*目录中文件是否已经存在。如果不存在，立即拷贝；如果存在，就检查*dir2*和*dir1*目录下的两个同名文件的最后一次修改时间，如果*dir1*中的文件比*dir2*中文件的最后一次修改时间更晚，才进行拷贝，否则，如果*dir2*中的文件和比*dir1*中文件最后一次修改时间相等甚至更晚，就不进行拷贝。这样达到增量拷贝的目的，在目录中的文件数比较多的情况下可以大大提高效率。而且，如果操作员的失误把源目录和备份目录的名字在使用cp命令时颠倒了位置，那么也不会灾难性的用旧的文件覆盖掉所有新修改的文件。Windows中也有类似功能的命令XCOPY，选项/D可以用来实现增量拷贝(Date)。

【例 4-6】目录的拷贝与增量拷贝。

(1) 拷贝目录work.d。

```
cp -r work.d  bak.d
```

(2) 增量拷贝。

设bak.d是work.d的备份目录,将work.d中的内容增量拷贝到备份目录中。

```
cp -ruv work.d bak.d
```

顺便提及的是，UNIX有一个命令touch可以将文件的最后一次修改时间设置为当前时间，但是不修改文件内容。例如：

```
touch *.[ch]
```

4.6 文件的归档与压缩处理

4.6.1 tar:文件归档

文件归档程序tar最早是为顺序访问的磁带机设备而设计的(Tape ARchive，磁带归档)，用于保留和恢复磁带上的文件。早期的计算机系统，硬盘容量不大，相对而言，磁带容量大，而且价格便宜，使用磁带机进行数据备份和恢复数据。

命令用法：

tar [ctxu][v][f *device*] *file-list*

选项的第一个字母指定要执行的操作(功能字母)是必须的。

c: create创建新磁带。从磁带的头上开始写，以前存于磁带上的数据会被覆盖掉。

t: table列表。磁带上的文件名列表,当不指定文件名时,将列出所有的文件，否则，将列出指定的文件。

x: extract抽取。从磁带中抽取指定的文件。当不指定文件名时，抽取所有文件。如果磁带上有几个名字相同的文件时,则最后一个文件将覆盖所有较早的同名文件。

u: update更新。把文件追加到磁带的尾部，这个文件的某个版本也可能曾经存放到磁带上。为了兼顾磁带设备的顺序访问特点，新版本文件追加到文件尾部，旧版本文件仍保留在磁带中。

除功能字母外，还可附加其它字符以选用所想要的功能。

v: verbose冗长。tar每处理一个文件，就打印出文件的文件名,并在该名前冠以功能字母。若用t功能,则以长格式列出磁带中的文件名。

f: file指定设备文件名。

【例 4-7】使用 tar 命令的例子。

(1) tar cvf /dev/rct0 .

将从当前目录开始的整棵目录树,备份到设备/dev/rct0中，圆点目录是当前目录。

(2) tar xvf /dev/rct0

将磁带设备/dev/rct0上的数据恢复到文件系统中。

(3) tar tvf /dev/rct0

打印磁带设备/dev/rct0上的文件目录。

(4) tar uf /dev/rct0 test.c

磁带备份完成之后，文件test.c又发生了变化，将更新过的test.c追加到磁带尾部。

(5) tar cv *

将当前目录下所有文件备份到默认的设备中。

(6) tar命令可以指定一个普通文件代替设备文件,那么,就可将多个文件或一棵目录树存储成一个文件。这是早期出现的一种将一棵目录树打包成一个文件的软件工具。

tar cvf my.tar *.[ch] makefile

将多个文件存成单一的my.tar文件。这一命令的错误用法是tar cvf *.[ch] makefile，漏掉了必须的设备文件名，按照shell对文件名的展开规则，会冲掉现存的某一文件。

tar cvf work1.tar work1

其中，work1是一个复杂的子目录,有多个目录层次。结果，打包成一个文件work1.tar。

(7) tar xvf work1.tar

从归档文件中恢复数据。

4.6.2 compress:文件压缩

采用LZW算法对文件压缩,普通文本文件可压掉50-80%，这是一种字典压缩算法。。压缩算法对于那些文件中的数据很有规律的内容压缩效率很高，有许多字段是空白的一些数据库文件压缩后文件体积甚至可以减少90%以上。

例1:compress chapt5　　压缩,生成新文件chapt5.z

zcat chapt5.z　　读取压缩格式的文件。

uncompress chapt.z 解压缩,还原文件chapt5。

4.6.3 应用

文件归档tar与压缩处理compress在不同的UNIX系统之间有通用性。因此,可以使用这些方法在不同UNIX中交换程序或数据。

【例 4-8】使用 tar 命令和 compress 命令通过网络中拷贝一棵目录树。

在主机A:

```
tar  cvf  xapi.tar  xapi
```

将整个xapi目录树,存到一个文件xapi.tar中。

```
compress xapi.tar
```

压缩生成文件xapi.tar.Z。

将文件xapi.tar.Z通过网络传到主机B(用ftp或者E-mail)。

在主机B上执行下面的操作:

```
uncompress xapi.tar.Z
tar  xvf  xapi.tar
```

用这种方法可以把整个一棵目录树复制到另一台UNIX主机上。

因特网上有许多后缀为.tar.Z的文件，就是用这种方法制作出来的。现在的Windows中WINZIP，WINRAR等工具也都可以识别UNIX的这种tar格式的文件和.Z格式的压缩文件。

4.7 文件系统的存储结构

4.7.1 基本文件系统与子文件系统

UNIX的整个文件系统分成基本文件系统(也叫根文件系统root filesystem)和子文件系统。基本文件系统是整个文件系统的基础,不能 “脱卸(umount)”。子文件系统以基本文件系统中某一子目录的身份出现，包括用作子文件系统的硬盘，软盘，USB盘，网络文件系统NFS等，不象DOS那样使用逻辑盘的概念。根文件系统和子文件系统都有自己独立的一套存储结构和目录结构，甚至文件系统的格式都不一样。

【例 4-9】文件系统的创建和安装。

在SCO UNIX中创建文件系统的命令。

```
mkfs /dev/fd0135ds18
```

命令mkfs创建文件系统(make filesystem),块设备文件/dev/fd0135ds18指3.5英寸容量1.44MB的A盘。此命令类似DOS中format命令，在磁盘上建立文件系统。

安装一个子文件系统的命令是mount,例如:

```
mount /dev/fd0135ds18 /mnt
```

其中，mount的第二个参数/mnt可以是任一个事先建好的空目录名，允许处于根文件系统的任何目录中。这个操作，类似Windows的“映射虚拟盘”。从此以后，凡是操作子目录/mnt，就是对子文件系统的访问。对于应用程序来说，从所操作的文件或目录名，看不出和其它根文件系统的对象有什么区别。

下面的命令都是操作软盘设备。

```
cp /usr/include/*.h /mnt
rm /mnt/stdio.h
mkdir /mnt/jiang
cp /usr/jiang/*.[ch] /mnt/jiang
vi /mnt/jiang/fn.c
ls /mnt
```

不带参数的mount命令，列出当前所有的子文件系统。umount命令的功能与mount命令相反，拆除一个已安装的子文件系统。以后系统就无法访问子文件系统中的文件。当某个运行的进程的当前工作目录是位于一个子文件系统中的目录时，umount命令就会失败。所以，有的C语言程序中会有chdir("/");语句以确保自己的当前工作目录不在一个子文件系统中。下面的命令使得软盘A不再与/mnt目录有联系。

```
umount  /dev/fd0135ds18
```

【例 4-10】NFS 网络文件系统的安装。

建立一个网络文件系统。设有两台主机，主机C和主机S,两主机上均已安装好了NFS软件包，主机C期望共享主机S上的文件目录/usr/jiang。在主机S的文件/etc/exports中添加行/usr/jiang。然后在主机C上,执行下面的命令：

```
mount -f NFS  203.123.54.189:/usr/jiang  /xbg
```

其中，203.123.54.189是主机S的地址，/xbg是事先在主机C上已建好的空目录。这样，主机C上访问的文件/xbg/makefile，实际上访问的是主机B上的文件/usr/jiang/makefile。使用局域网或者其它高速网络时,用NFS可以非常方便地共享其他主机上的目录。当使用低速广域网时，仍然可以使用NFS，但是速度很慢。

【例 4-11】LINUX 系统中引用 Windows 格式的磁盘分区的例子。

在LINUX系统中,事先建立空文件目录/a，/c，/d，/u,/cdrom。分区1是FAT32格式的C盘，分区4是FAT32格式的扩展D盘。

```
mount /dev/fd0 /a
mount /dev/hdc1 /c
mount /dev/hdc4 /d
mount /dev/cdrom /cdrom
mount /dev/sda1 /u
```

LINUX可以自动设别并支持FAT32格式的Windows文件系统。设备/dev/cdrom是CDROM盘，设备/dev/sda1是USB接口Flash盘,设备/dev/fd0是软驱A,设备/dev/hdc1是硬盘分区1上的C盘，设备/dev/hdc4是硬盘分区4上的D盘。从此以后，

访问/a，/c，/d，/cdrom，/u子目录，就是对其它文件系统的访问。不同的硬件环境下，这些设备文件的名字会有些差异。

UNIX的设备文件分字符设备文件和块设备文件，只有块设备文件，才可以使用mkfs命令创建文件系统和使用mount命令安装到根文件系统中。

4.7.2 文件系统的结构

块设备提供的界面是使得整个外存设备看起来是简单的线性块的组合。使用块设备时不再关心底层物理磁盘的尺寸和特性。对块设备的特定编号的块的读取或者写入请求，需要映射到对于设备有意义的参数，即：这个块对应的磁盘上的磁道号，柱面号，扇区号，并且启动相应的控制器完成读写操作，这些，都是块设备驱动程序的任务。文件系统使用块设备，要求所有不同种类的设备提供相同的界面，这种功能上的分层次安排，使得文件系统的设计者不再关注硬件驱动器，物理界质等方面的差异。设备提供商，也只需要提供设备驱动程序就可以用文件系统管理它的设备上的信息。

UNIX的每个逻辑块设备，如：硬盘的一个分区，一张软盘，USB接口的flash盘，光盘，等等，都对应一个块设备文件，如：/dev/hdc4，/dev/cdrom，/dev/sda1等等，在每个逻辑设备上构造一个独立的子文件系统。系统在这个设备上创建这个子文件系统的时候，把整个逻辑设备以“块”为单位划分，编号为0,1,2,...。磁盘设备读写的最小单位是“扇区”。典型的，一个扇区会有512字节，逻辑设备的“块”一般是一个扇区，或者2^n个扇区，这样，一块会可能会是512字节，1024字节，4096字节，……。文件系统使用设备时，空间分配或释放的最小单位是 “块”。例如，块大小为4096字节的系统中，存放大小4097字节的信息，需要两块。

参见图4-4，一个文件系统由下述几部分构成:

(1) 引导块(Boot block)

0号块。用于启动系统,存放引导程序，它含有的程序代码用于系统启动时引导执行操作系统的内核。当整个文件系统由多个文件系统构成时，只有根文件系统的引导块有效。

(2) 超级块(Super block)

1号块，也叫管理块。存放与整个文件系统的管理有关的信息。如:文件系统的大小，i节点区的大小,空闲空间大小,空闲块链表的头等等。这些是整个文件系统对块设备中“块”进行分配和释放操作的重要信息。

(3) i节点区

i节点 (index node),简记为i-node。i节点区由若干块构成，专用于存放i节点。系统中的每个文件都对应一个i节点。每块可容多个i节点,每个i节点有固定大小。例如：每块512

字节，i节点固定大小64字节,每块容纳8个i节点。i节点中最重要的信息是“索引”信息，这也是取名为i节点的原因。这些索引信息，是由一组指针构成的索引表，指向文件存储区中实际存放数据的存储块。顺着这些指针，经过直接或者间接的索引，能够到磁盘的指定块中找到文件的信息，这正是内核中的“文件管理”模块要实现的最重要的功能之一：逻辑文件到物理磁盘块的映射。除了索引信息之外，i节点中还记录了一些文件属性信息，包括:文件类型，属主，组，权限，link数，大小，最近访问和修改的时间。注意：i节点内不含有文件的文件名。

i节点的数目：在使用命令mkfs创建文件系统时，根据整个块设备的大小，由系统管理员手工指定，或者采用默认的大小。一旦创建之后，对这个存储设备来说，i节点的数目就固定为一个常数。

i节点编号:从1开始。1，2，3，...。不使用编号为0的i节点。

(4) 文件存储区

用于存放文件中数据的区域，除了普通磁盘文件之外，还包括目录表。一个存储设备的文件存储区占整个存储空间的绝大部分。

引导块	专用块	i 节 点 区	文 件 存 储 区

图 4-4 文件系统的布局

文件系统创建(mkfs)，安装(mount),脱卸(umount)，完整性检查和修复(fsck,类似Windows的“磁盘扫描程序”)，程序存于/etc目录下。/etc目录主要存放系统维护和管理使用的命令及配置文件。

4.7.3 目录结构

UNIX的目录结构不是严格意义上的树形结构，是一种允许带交叉勾链的目录结构。每个目录表在UNIX中也被组织的跟一个普通文件一样,存于“文件存储区”中，有其自己的i节点。

用ls命令列出的目录的大小是目录表本身的长度，而不是目录下所有文件的长度总和。

目录表的基本组成单位是“目录项”，每个目录项由一个“文件名-i节点号”对构成。UNIX采用这样的存储结构，部分原因是为了提高目录检索的效率。

以一个最简单的文件系统为例，每块是一个磁盘扇区，512字节大小。考虑不采用将文件名和文件的索引信息分离的办法，而是将文件名和文件的索引信息存放在一起的情况。设存储文件名使用定长的14字节，索引信息需要64字节，那么，每个目录项大小78字节，每块可以容纳512÷8=6个目录项。设当前目录下有100个文件，需要访问的文件的文件名mydata.bin存放在目录表的最末尾处。为了能够根据文件名获取该文件在磁盘上的存储块索引，系统需要调入目录。第一次，读入一块，含6个目录项，比较这6个文件名，结果都不是需要访问的文件mydata.bin，这样，需要继续读入下一块。在读到第17块时，

才找到文件名mydata.bin，根据索引信息访问磁盘上的块。另外一种方案是，将文件名和索引信息分开，索引信息存放在i节点中，在目录中仅记录文件名和i节点的编号。文件名14字节，i节点号两字节，每个目录项16字节，一个磁盘块含512÷16=32个目录项。这样，读入4块就检索到文件mydata.bin的i节点号，根据i节点号读取i节点，只需要读入i节点区的一块。总共磁盘操作5块，就可以根据名字找到文件的索引信息。显然后一种方案需要进行的磁盘操作次数更少。相对于内存来说磁盘操作需要较长的时间，因此，后一种方案效率更高。UNIX采用这样的方案还可以很方便的实现这样的功能，即给一个文件多个名字。

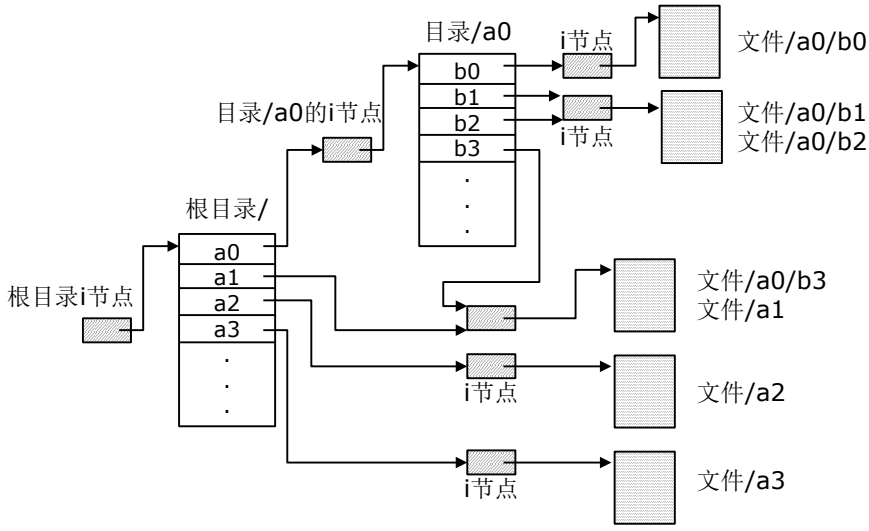

图 4-5 文件系统目录表的构造

在图4-5的例子中，一个文件可以有多个名字。如：/a0/b1和/a0/b2是同一个文件，/a1和/a0/b3也是同一个文件。从UNIX文件系统的存储结构上看，达到这样的效果非常容易，只要在不同的目录项中填写“文件名-i节点号”的时候将i节点号填写的相同就可以了。

命令ls的-i选项，可以列出文件的i节点号。如果两个文件的i节点号相等，它们就引用同一个文件。在介绍ls -l命令时，每个文件列出的项目之一是“文件的link数”，就是同一个i节点被引用的次数。link数被记录在i节点中，便于文件的删除操作。上例中，删除文件/a1,不可以删除磁盘上数据信息，因为/a0/b3还访问这个文件。但是，将i节点的link数减1，当删除文件/a0/b3时，link数再减1，变为0。于是，将文件存储区的存储块释放，同时也释放这个i节点。这种方法是多个不同的引用共享同一个数据对象时常用的算法，在UNIX操作系统内核和网络协议软件中广泛使用。根据i节点的link数，可以判断出文件被多少个目录项所引用，普通文件的link数一般为1。当文件的link数不为1的时候，从文件系统的存储结构上看，仅仅根据link数，无法直接找出系统中在哪些地方还有哪些目录项引用了这个i节点。根据i节点号，find命令的-inum选项可以查找指定i节点号的文件。

4.7.4 命令 df 与 du

1. df:显示可利用的磁盘空间

df命令用于显示文件系统的空闲空间(disk free space)。DOS的DIR命令会在显示结束的时候报告文件系统的空闲空间大小，UNIX的ls命令没有这种功能，检查文件系统空闲空间应使用命令df。

df可以列出每个子文件系统的设备文件名，mount安装的路径，文件存储区和i节点区的总长度,空闲空间和百分比。不同的系统df的显示不完全相同，常用选项有-v和-i。-v选项(verbose)使得系统显示更多的信息，-i显示与i节点有关的信息。

【例 4-12】 使用 df 命令的例子。

```
$ df
Filesystem    512-blocks      Free %Used    Mounted on
/dev/hd4         2359296   2167496    9%     /
/dev/hd2        15007744   8842688   42%     /usr
/dev/hd11      128188416 107533208   17%     /home/malaba
$ df -i
Filesystem   Iused    Ifree %Iused Mounted on
/dev/hd4      2104   587720     1% /
/dev/hd2     75444  1800524     5% /usr
/dev/hd11    11441 16012111     1% /home/malaba
```

df在列出文件系统的文件存储区时，含有总块数和空闲块数，根据系统的不同，一块可能是512字节，或者其它数目。上述的例子中，设备文件/dev/hd2总空间7.5GB，将15007744除以2，就将大小为512字节的单位“块”转换为“Kbytes”，还剩余约4.4GB空闲，已占用42%。/dev/hd2设备的文件系统中i节点已占用75444，一般系统中大部分文件是普通磁盘文件，link数为1，可以推算系统中大概有7.5万个文件和目录。

2 du:显示磁盘使用信息

du命令(disk usage)显示包括所有下属子目录在内的某一目录树中文件使用的块数总和，显示单位仍然是“块”。

【例 4-13】 使用 du 命令的例子。

```
$ du /etc
8       /etc/rc.d/rc2.d
8       /etc/rc.d/rc3.d
8       /etc/rc.d/rc4.d
8       /etc/rc.d/rc5.d
8       /etc/rc.d/rc6.d
```

```
8       /etc/rc.d/rc7.d
8       /etc/rc.d/rc8.d
8       /etc/rc.d/rc9.d
32      /etc/rc.d/samples
112     /etc/rc.d
272     /etc/methods
24      /etc/locks
8       /etc/snmpinterfaces
8       /etc/rpm
16      /etc/vatools/vac
16      /etc/vatools/vacpp
48      /etc/vatools
112     /etc/vacpp/C
112     /etc/vacpp/en_US
232     /etc/vacpp
3408    .
```

对/etc下每一个下属子目录,都列出其下属文件所占用磁盘空间的块数,最后列出一个总合计。

4.8 硬连接与符号连接

4.8.1 硬连接

1. 什么是硬连接

存于文件存储区的每一个文件都有一个i节点。目录表由目录项构成,目录项就是一个“文件名-i节点号”对。因此，可以在同一目录表中有两个目录项，有不同的文件名，但有相同的i节点号。在不同的目录表中也可以有两个目录项有相同的i节点号。每个目录项指定的文件名-i节点号的映射关系,叫做硬连接。

硬连接数目(link数):同一i节点被目录项引用的次数。

2. 命令 ln

ln命令(link)可以用来创建一个硬连接。

【例 4-14】普通数据文件的硬连接。

```
$ ls -l chapt0
-rw-r--r--   1 kc   kermit         54332 Jun 01 12:28 chapt0
$ ln chapt0 intro
```

(chapt0 是一个已存在的文件,新生成目录项 intro,ln 命令创建新的目录项 intro 时，文件名为 intro，i 节点号照抄 chapt0 的 i 节点号。ln 命令源文件在前，目的文件名在后，

与 cp 命令的参数排列顺序一致，便于记忆。）

```
$ ls -l intro chapt0
-rw-r--r--   2 kc   kermit         54332 Jun 01 12:28 chapt0
-rw-r--r--   2 kc   kermit         54332 Jun 01 12:28 intro
```

（link 数为 2，列出的前面的几项必相同，因是从同一 i 节点提取的数据。chapt0 与 intro 两文件同时存在时，有相同的 i 节点，两者地位完全平等，不存在谁主谁次，谁先谁后的问题。）

```
$ ls -i intro chapt0
88077 chapt0
88077 intro
```

（两文件的 i 节点号均为 88077）

```
$ vi intro （修改文件中的内容并存盘）
$ ls -li intro chapt0
88077 -rw-r--r--   2 kc   kermit         54140 Jun 01 12:30 chapt0
88077 -rw-r--r--   2 kc   kermit         54140 Jun 01 12:30 intro
```

（两文件显示的文件信息，文件大小，最后一次修改时间，同时发生变化）

```
$ rm chapt0
$ ls -l intro
-rw-r--r--   1 kc   kermit         54140 Jun 01 12:30 intro
```

（删除文件 chapt0 后，文件所占用的存储空间并不释放，还可以引用文件 intro 来访问它，文件的 link 数减 1）

【例 4-15】程序文件的硬连接。

用ln可以为程序文件建立硬连接。在本章的4.3.4节的例4-2，介绍了一个很简单的程序arg.c,编译链接之后生成可执行文件arg。

```
$ ln arg arg1
$ ls -li arg*
88090 -rwxr-xr-x   2 kc   kermit         11628 Jun 01 09:27 arg
88090 -rwxr-xr-x   2 kc   kermit         11628 Jun 01 09:27 arg1
87302 -rwxr--r--   1 kc   kermit           164 Jun 01 09:20 arg.c
$ ./arg abc ABCDEF
0: [./arg]
1: [abc]
2: [ABCDEF]
$ ./arg1 abc ABCDEF
0: [./arg1]
1: [abc]
2: [ABCDEF]
```

尽管arg和arg1都是磁盘上的同一个文件，但是由于名字不同，执行时，从程序中感受到的参数argv[0]是不同的。因此，程序完全可以做到根据argv[0]的不同，使得程序有不同的表现。在有的系统中，的确用这种方法，使得同一个程序的执行，有不同的表现。在AIX操作系统中，grep和egrep是一个程序文件，尽管使用不同名字时程序的表现不同。

下面是AIX系统中程序文件硬连接的使用。

```
$ pwd
/usr/bin
$ ls -li grep [ef]grep
```

```
30997 -r-xr-xr-x   3 bin     bin          20434 Feb 11 2002  egrep
30997 -r-xr-xr-x   3 bin     bin          20434 Feb 11 2002  fgrep
30997 -r-xr-xr-x   3 bin     bin          20434 Feb 11 2002  grep
$ ls -li sh [kpt]sh
31015 -r-xr-xr-x   4 bin     bin         241436 Oct 24 2001  ksh
31015 -r-xr-xr-x   4 bin     bin         241436 Oct 24 2001  psh
31015 -r-xr-xr-x   4 bin     bin         241436 Oct 24 2001  sh
31015 -r-xr-xr-x   4 bin     bin         241436 Oct 24 2001  tsh
$ ls -li vi
30991 -r-xr-xr-x   5 bin     bin         232114 Sep 24 2001  vi
$ find . -inum 30991 -exec ls -li {} \;
30991 -r-xr-xr-x   5 bin     bin         232114 Sep 24 2001  ./ex
30991 -r-xr-xr-x   5 bin     bin         232114 Sep 24 2001  ./edit
30991 -r-xr-xr-x   5 bin     bin         232114 Sep 24 2001  ./vedit
30991 -r-xr-xr-x   5 bin     bin         232114 Sep 24 2001  ./vi
30991 -r-xr-xr-x   5 bin     bin         232114 Sep 24 2001  ./view
```

用ln建立硬连接时,只限于文件,两个文件名的路径名也必须处于同一文件系统中。不允许对目录用ln命令建立硬连接,从文件系统的存储结构上看，建立这种硬连接很容易实现，但是，如果对目录允许这样的操作，就会破坏目录的树型结构，使得目录结构变成网状结构。因此，对磁盘目录的硬连接操作被禁止。

3. 目录表的硬连接

尽管不允许用户对目录使用ln命令建立硬连接，但是，创建目录的操作，操作系统内核会自动在所创建的目录中创建.和..文件，使得目录的硬连接数目也不再是1。例：图4-6示例出一个目录结构和这个目录结构对应的硬连接情况。

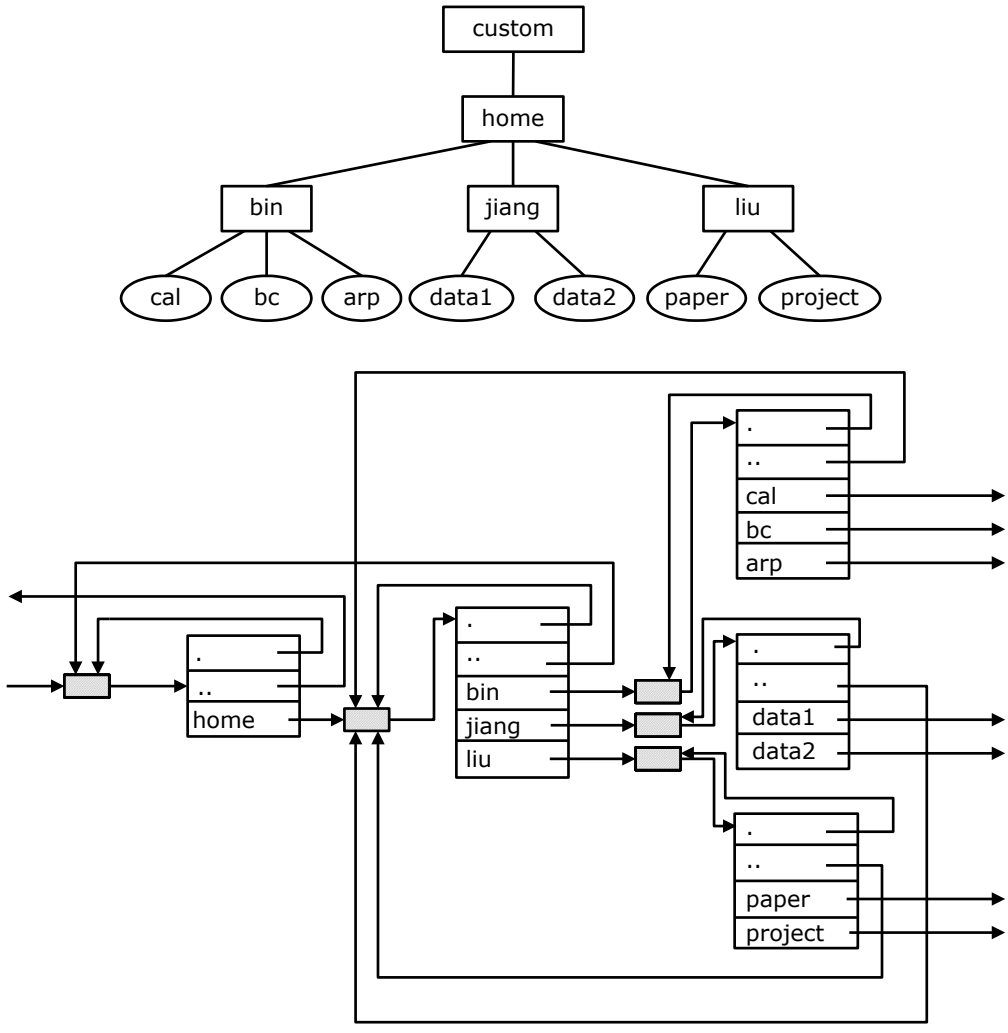

图 4-6 目录文件的硬连接

目录home的link数为5，分别对应上级目录的home目录项，自己目录中的.目录项，以及三个子目录中的..目录项。一般来说,对于一个普通目录，它的link数等于直属子目录数加2。命令ls -ld . 可以查当前目录的link数,目录的大小指的是目录表的大小。

根据上述的存储结构，UNIX在创建目录时自动在所创建的目录中增加了.和..两个目录项,i节点号分别填写新创建目录的i节点号和上级目录的i节点号。从此以后，对.和..目录项的访问不再有特殊性，它们和普通目录项的访问一样,这两个目录项只有创建和删除时系统需要特殊处理。

4.8.2 符号连接

符号连接也叫软连接,最早在BSD UNIX中实现。符号连接是一种非常有用的功能，符号连接的思想被广泛应用到用名字进行管理的信息系统中。为了便捷地管理信息系统

中的信息对象，符号连接允许给一个对象取多个符号名字，可以通过不同的路径名共享同一个信息对象。在UNIX系统中，用一个特殊文件“符号连接文件”来实现符号连接。在“符号连接文件”中仅包括了一个描述路径名的字符串。创建符号连接的命令是使用带-s选项(Symbol link)的ln命令。类似cp命令和建立硬连接的命令，符号参数在前，新建的文件名在后。

【例 4-16】符号连接的使用举例。

```
$ who > users_on
$ ln -s users_on active_users
$ ls -l active_users
   lrwxrwxrwx 1 fang kermit 8 Jul 26 16:57 active_users->users_on
$ cat active_users
   fang    tty00  Jul 5  14:49
   sun     tty01  Jul 5  11:31
   liang   tty03  Jul 5  15:50
   dong    tty11  Jul 5  09:45
```

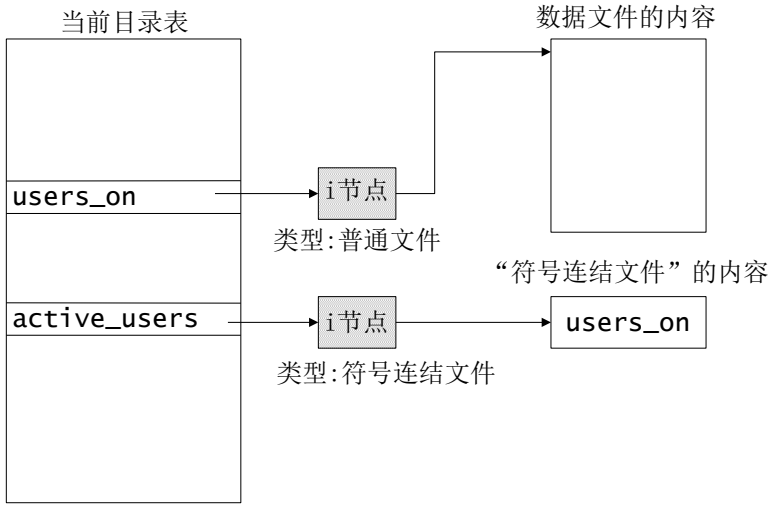

图 4-7 实现符号连接的存储结构

参见图4-7，命令ln -s users_on active_users创建新的符号连接文件active_users, 文件系统中active_users文件不再是普通的磁盘文件，对应的i节点中记录了文件类型为符号连接文件，用ls -l列出时，类型域为l。文件大小仅为8字节，因为在文件存储区存储的“符号连接文件”active_users中只存放了8字节长的users_on字符串。箭头后列出符号连结文件的内容。ls -l也是读取“符号连接文件”自身内容的命令。符号链接文件的文件最后一次修改时间以后不再变化。

使用ls -l命令在显示文件名的位置有->符号，->后边的内容是符号连接的内容。可以使用下面的组合命令，列出当前目录树中所有的符号连接文件。

```
$ ls -lR | grep -- '->'
```

```
$ find . -exec ls -l {} \; | grep -- '->'
```

grep后面的--和这里的单引号不可少。这些在前面的4.4.4节的例4-5中介绍过。

执行命令cat active_users实际访问文件users_on。在打开文件时,系统实际打开文件users_on。这样，用户读写文件active_users，跟直接使用文件users_on一样。将active_users展开成该符号连接文件存储的字符串users_on的操作，不是由shell完成，而是由操作系统内核的文件管理模块完成。

操作系统内核模块处理符号连接的方法是，open，creat等系统调用，提供一个文件的路径名。在逐个翻译路径名中用斜线分割开的路径分量时,若系统发现符号连接,就把符号连接的内容加到路径名的剩余部分的前面,翻译这个名字产生结果路径名。若符号连接包含绝对路径名，使用绝对路径名。否则，根据文件层次结构中该连接的位置(不是根据调用进程的当前的工作目录)和符号连接的内容，继续分析路径名。

【例 4-17】符号连接的展开方式。

设/a/b/c是一个符号连接文件内容为p/g/r，不是绝对路径，那么,访问文件/a/b/c/d/e则实际访问 /a/b/p/g/r/d/e。文件c记录了符号p/g/r,处理时将符号粘贴替换到路径分量c的相应位置。同样的，设a/b/c记录符号../p/g，那么,a/b/c/d/e/实际访问a/b/../p/g/d/e。

设/a/b/c是一个符号连接文件内容为/p/g/r，是绝对路径，那么,访问文件/a/b/c/d/e 则实际访问 /p/g/r/d/e。

【例 4-18】符号连接与硬连接的区别。

设目录结构如图4-8所示，当前目录为d。

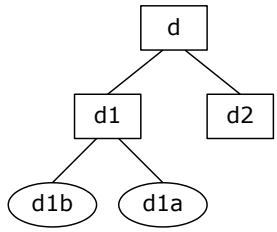

图 4-8 目录结构举例

(1) ln -s d1/d1b d1/dx

那么，在d1目录下新建文件dx，在创建d1/dx符号连接文件的时候，系统根本不解读d1/d1b字符串的含义,仅仅是原封不动的将这个符号字符串存储在文件d1/dx对应的文件存储区中，任务就算完成。只有在打开文件d1/dx读写访问时，操作系统的文件系统才进行符号连接处理，实际访问d1/d1/d1b，而不是d1/d1b。如果这样的文件不存在，访问才会失败。

(2) ln d1/d1b d1/dx

那么，在d1目录下新建文件dx。硬连接的处理方式是检索出文件d1/d1b的i节点号，在d1目录下新创建的目录项dx使用这个i节点号，硬连接操作完成。如果d1/d1b文件事先不存在，硬连接就会失败。访问文件d1/dx和访问d1/d1b是等价的。在这点上，符号连接和硬连接的处理方式的不同。

【例 4-19】观察 UNIX 命令对符号连接文件的操作。

(1) 创建一符号连接文件:

```
ln -s users_on  active_users
```

无论事先文件users_on是否存在，总可以建立符号连接文件active_users。

(2) 读

读取active_users，例如cat active_users， 实际上读取文件users_on。文件users_on如果不存在，读操作就会失败。

(3) 写

用vi编辑器改写active_users，实际改写了文件users_on。文件users_on的最后一次修改时间也随之变化。而ls -l active_users的最后一次修改时间不变。UNIX系统专有一个命令touch，被touch命令“摸”过的文件。内容不发生任何变化，但文件的最后一次修改时间被设置为当前时刻。执行命令touch active_users同样改变users_on的最后一次修改时间。

(4) 覆盖文件

使用命令who > active_users，则实际上文件users_on被覆盖。如果文件users_on事先不存在，那么就会创建新的文件users_on。

(5) 执行硬连接

执行ln active_users file2,那么file2的i-node号与文件users_on相同。

(6) 修改文件的权限

执行chmod a+w active_users，那么文件users_on的权限发生变化。

(7) 删除文件

执行rm active_users，则实际上符号连接文件被删除,文件users_on不变。

总之，一旦建立了符号连接,只有删除操作删除的是符号连接文件自己。在符号连接存在期间，所有对符号连接的操作都将访问符号连接所引用的文件,而不是符号连接文件本身。

【例 4-20】符号连接与硬连接的比较，符号连接与目录层次的关系。

对图4-9的文件层次结构，设当前目录为d，

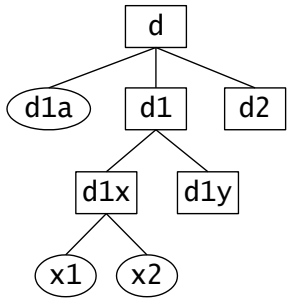

图 4-9 目录结构举例

(1) `ln -s d1a dx`

`cat dx`实际访问`d1a`，如果`rm d1a`后，`cat dx`仍然试图访问`d1a`文件。由于文件`d1a`不存在，会出错。反过来，如果`rm dx`，那么，对`d1a`没有任何影响。

(2) `ln d1a dx`

`rm d1a`后，使用硬连接`cat dx`仍能访问以前的文件。

(3) `ln -s d1/d1x dx`

`cd dx`后当前目录变为`d1x`，再执行`cd ..`，则当前目录变为`d1`而不是`d`。

了解符号连接和硬连接实现的原理，会很容易理解系统的这些特性。

4.8.3 硬连接与符号连接的比较和应用

硬连接利用了文件系统内部的存储结构，很简单的实现了信息共享。符号连接依靠操作系统中文件系统模块的软件处理，实现了信息共享。硬连接，只适用于文件，不允许用户通过命令对目录实现硬连接，以保障文件系统中目录的树型结构不被破坏。每个文件系统有自己独立的一套i节点，因此，不同子文件系统之间，不可能用硬连接实现文件的共享。

硬连接能够完成的功能，使用符号连接都可以做到。符号连接在操作系统内核里用软件实现，同硬连接相比要多占用系统的一部分开销，包括软件处理占用的CPU时间和调入符号连接文件需要的磁盘操作的时间。根据符号连接实现的原理，符号连接完全可以适用于目录，也可以将符号连接文件和符号连接引用的目的文件安排在不同的文件系统中，从而比硬连接更灵活。

由于符号连接可以用于任意的目录，目录下的文件可以是符号连接文件，当然可以是任意目录，如果是它的上级目录，就完全可能构造出符号连接的死循环。UNIX内核的`open`等系统调用，在分析一个路径名时，逐个分析用斜线隔开的多个路径名分量。系统

保留一个计数器，每进行一次符号连接处理，计数器加1，当计数器的值到达某一值（比如：20）时，就拒绝再进行进一步的符号连接处理，并返回错误，从而，避免这种死循环出现。这也是一种避免递归死循环的经典算法。

硬连接和符号连接的使用可以带来很多方便。

【例 4-21】使用硬连接和符号连接共享数据文件以节约存储空间。

在家庭相册中有几百个照片文件，每个文件都有大约700K字节。可以为这些文件建立多个目录，比如，按照时间顺序，每年的照片组织为一个目录，还可以根据家庭成员，或者，某一次的活动来组织目录。使用符号连接或者硬连接都能做到两个人合影的同一张照片，同时处在某个年度目录中，两个不同的家庭成员目录中，还可以放入某次活动的专题目录中。这样，同一个文件，都可以从四个目录中检索到，而不需要将一个文件复制4份。使用符号连接和硬连接还是有些区别，使用符号连接时，删除某个700K字节的文件，系统就会释放出700K字节的空间，可能会导致某些引用这个文件的符号连接不再能正常打开文件；但是，使用硬连接，需要删除所有的相关目录项才能够释放文件的存储空间。

【例 4-22】使用符号连接对程序和用户透明地变更目录组织结构。

在UNIX系统中有/usr/adm目录，一般用于存储一些系统软件的日志文件，/usr/tmp可以存放一些临时文件，/usr/spool存放系统收到但用户未读的邮件和打印任务队列。如果系统管理员需要对目录系统重新进行规划，希望这些文件不再存储在/usr目录中，而是把这些每日都会变化的数据信息存放到新建立的/var目录下，用/var/adm, /var/tmp, /var/spool分别代替原先的目录。但是，许多用户已经习惯于使用/usr/adm, /usr/tmp, /usr/spool, 甚至某些事先编好的程序软件在操作数据时，就固定地从这些目录下访问数据。这些软件没有提供配置手段，而且，修改这些事先编制好的软件也不可能。这种情况下，符号连接使得系统管理员对文件目录的这种重新规划提供了可能。创建三个符号连接

```
ln -s /var/adm /usr/adm
ln -s /var/tmp /usr/tmp
ln -s /var/spool /usr/spool
```

那么，系统管理员的对系统的这种重新安排，变得几乎完全透明。其他用户仍然可以按照他们原来的习惯使用/usr目录下的这三个目录，原先的程序软件也不需要任何改动。甚至，如果/var目录是另外一个子文件系统的mount点，就可以将这样的信息转移到另个磁盘的文件系统中。操作员还可以根据需要，使用符号连接在不同的时期，把这些目录导向不同的存储位置。

前面在4.8.1节介绍硬连接的时候，可以为同一个命令文件建立多个目录项，在命令文件的程序内部，根据`argv[0]`可以判断出命令名的不同，从而，程序执行时的表现不同。如：AIX系统中的`ksh`和`sh`。使用符号连接也可以达到相同的效果，在LINUX系统中，`/usr/bin/sh`就是一个符号连接，符号连接的内容是`bash`。用户键入`bash`时，程

序表现出bash的特性；键入sh时，表现出B-shell的特性，隐藏bash对B-shell的扩展，尽管真正的可执行程序文件是同一个磁盘文件。

类似的，Windows系统中有“快捷方式”，尽管在很多地方使用起来有些限制，还不能做到和UNIX的符号连接一样在访问文件系统中的对象时对软件和用户都透明，但是，它的基本做法和出发点跟符号连接相似。

4.9 系统调用

前面介绍了一些文件操作和目录操作的命令，以及上一章介绍的实用程序，这里从程序员的角度体会一下UNIX的命令是怎样实现的。以后的章节会陆续介绍一些编程接口。一般来说，UNIX的命令可以实现的功能，都可以编程实现，因为这些命令和我们编写的程序一样，都属于UNIX内核支撑下的应用程序。有些UNIX的命令实现的功能，程序员难以编程实现的原因是，部分内核提供的接口功能不公开，或者缺少相关的文档。使用man查到的命令手册中的“SEE ALSO”部分，经常会列出一些与这些命令相关的系统调用和库函数名。在使用系统调用和库函数时，还应当注意这些函数定义遵循的UNIX标准，以便于移植。在介绍几个UNIX关于文件和目录的系统调用之前，首先，了解一下什么是系统调用。

“系统调用”(System call)是一个专用名词，是操作系统的内核提供的编程界面。使用C语言时，提供一些C语言的函数。通过这些函数，调用操作系统提供的功能。系统调用是应用程序和操作系统进行交互的唯一手段，应用程序只能通过系统调用，才能够申请和访问硬件资源。前面介绍的命令，都使用了UNIX的系统调用，都是运行在用户态的程序。系统调用函数，例如：用于文件操作的open，read，write，close等等，和普通的函数库调用，例如：字符串操作的strcpy,strlen等等，从C语言程序员的角度看起来用法一样，都是调用一个函数，但是，处理却不一样。系统调用的函数体一般非常简单，依赖于具体的操作系统，准备好系统调用所需要的参数之后，立刻产生一个软件中断，系统从用户态进入到内核状态，函数的功能由操作系统的内核代码实现。普通函数库如同程序员自己编写的程序一样，通过函数调用，利用程序的堆栈，在用户态完成函数内部的处理后返回。有些库函数，例如：缓冲文件操作的fopen，scanf，printf，fclose等，在函数里完成一些必要处理之后再引用系统调用。Windows系统很少会直接将系统调用提供给程序员使用，它提供的是类似缓冲文件操作那样的既有用户态代码，又会有系统调用的函数库，Windows的术语，叫“应用编程接口(API)”(Application Programming Interface)。

为了便于不同UNIX系统之间C语言源程序之间的移植，UNIX的系统调用函数的名字，参数排列顺序，参数的类型定义，返回值的类型，以及实现的功能，都属于POSIX标准来规范的内容。最初UNIX第7版只有大约50多个系统调用，其中只有一半的调用是常用的，到UNIX SVR4(System V Release 4)系统调用扩充到120个。UNIX系统的内核界面非常简洁。Windows也支持大部分的POSIX函数标准，所以，许多UNIX的系统调用函数在Windows系统下也可以使用，Windows提供了功能等价的函数。

UNIX的系统调用一般都返回一个整数值。例如：删除一个文件的系统调用是unlink，系统调用返回值为-1时，表示系统调用失败。为了说明为什么失败，有一个整数errno说明了失败的原因，errno记录了一个错误代码。程序员直接在C语言源程序中#include <errno.h>之后，就可以直接使用这个变量。这个变量在C语言的标准程序库中保留存储空间，系统调用失败后会被填写为错误代码。计算机很擅长识别这些整数类型的代码，但不方便程序员记忆和理解。为便于记忆，在errno.h头文件中定义了许多宏，用名字代替这些整数值。这些宏都有E前缀，例如：EACCESS，EIO，ENOMEM等等，对每个系统调用来说，在手册页中都列出可能的出错情况和对应的错误代码的宏名字。man 2 unlink可以列出系统调用unlink的用法，以及列举出失败的可能原因和错误代码的宏名字。errno便于用程序识别错误原因，但是，这样的数字代码很不便于操作员理解失败原因。库函数strerror可以将数字形式的错误代码，转换成一个可以供操作员阅读的字符串，这是一个重要的函数。

```
char *strerror(int errno);
```

与系统调用执行失败有关的另一个函数是perror(),函数原型为：

```
void perror(char *string);
```

它会在标准错误输出设备stderr上产生一条消息,描述在系统调用或使用库函数(库函数中使用了系统调用)期间所遇到的最后一个错误。先打印出字符串*string*,再打印出一个冒号和空格,然后再打印出这条消息和换行符。

前面介绍了一些文件操作和目录操作的命令，这里介绍几个UNIX关于文件和目录操作的一组系统调用和库函数。

4.10 文件和目录的访问

4.10.1 文件存取

操作系统的文件管理模块的主要功能是提供了一种便捷的手段管理存储在外存中的数据。基本方法是，将用户看来逻辑上相关的一组数据组织在一起，叫做“文件”，不同的文件长度可能会不同。系统提供了便于记忆的“文件名”，应用程序就可以根据文件名来访问存储在外存中的数据。如果没有文件系统的支持，那么，应用程序必须清楚地记住自己的数据存放在磁盘的哪一块中。数据较多需要多个存储块时，必须记住数据的每一部分分别存储在磁盘的哪些位置，它们在磁盘上可能是不连续的存储块。需要存储新数据时，还要挑选一个合适的存储块，不要错误地把已有的数据抹掉。没有文件系统的支持，实现这些功能非常麻烦。操作系统的文件管理模块，负责管理外存的分配和释放，负责检索出文件中数据在磁盘中的存储位置。这样，应用程序才能做到根据文件名访问存储在外存的信息。UNIX操作系统内核提供给应用程序员的文件，看起来是前后相继的

连续的无格式字节流。应用程序可以按照自己的意愿去解释这个字节流，如：文本文件，数据库文件，但这种解释与操作系统内核的文件管理模块无关。

1. 文件描述符

有了文件系统的支持，应用程序就可以根据文件名，访问文件中的数据，而不用操心这些数据怎样存放在外存上。那么，这种访问应当提供一个什么样的编程接口呢？例如：文件中有1000个数据，每个数据512字节，需要顺序的读出这1000个数据。最简单最朴素的编程界面是应用程序向内核提供文件名，存储起始位置，和希望得到的字节数等信息。类似下面的函数界面:

```
get_data(char *filename, int position, void *buf, int len);
```

其中,*buf*是用于存放读来的数据的缓冲区，调用时*len*设为512。这样，这个函数需要执行1000次。而且，每次调用时应用程序需要自己记着下次存取时该从文件的哪个位置获得数据，正确地填写*position*。前面介绍过文件的i节点，i节点的最重要作用就是索引功能。如果操作系统提供一个这样的编程界面访问文件，那么，每次这个函数执行时，就需要执行一系列的操作：

① 根据函数参数提供的文件名字符串，查找目录，得到文件对应的i节点号；

② 从文件系统的i节点区调入i节点；

③ 判断用户对文件有没有读操作权限；

④ 再根据*position*，查找i节点里存放的索引表，得到文件的这个数据块在磁盘上的位置；

⑤ 从磁盘上读入这个数据块，将数据返回给应用程序的*buf*缓冲区。

至此，操作完成。如果函数需要执行1000次，这些操作就需要重复地执行1000次。稍微分析一下，上述的①②③操作，尤其是①②，非常浪费时间，在1000次操作中都重复执行是没必要的。在这1000次的循环之前，仅作一次就足够了。这便是程序员都熟悉的文件操作采用`open/read/write/.../close`模式的主要原因。

所谓的文件“打开(open)”操作，就是完成包括上述的①②③在内的文件访问前的准备工作，把i节点等数据读入到内核中。在内核中还可以维持一个文件的访问位置的指针，这样用户在连续访问文件的时候就不用再提供*position*参数了。为了使应用程序可以在一个文件访问未完成的时候，同时访问别的文件，当多个文件交叉访问时，操作系统内核必须知道到底要访问哪个文件。需要对已经“打开”的文件建立一个索引。对于计算机来说，处理一个字符串结构的名字远不如处理一个整数值更有效，所以，这个索引是个整数，叫做“文件描述符”(file descriptor)。内核根据它很容易定位到相关的诸如文件当前读写位置和文件i节点的信息。UNIX系统中，每个进程有自己独立的一套文件描述符，按照UNIX和C语言的惯例，文件描述符是从0开始编号的整数，它是操作系统内核中进程PCB(Process Control Block进程控制块)结构里的一个数组的下标。从此以后，再操作文件时，多次的`read`或`write`直接提供“文件描述符”作参数而不是文件名，内核就可以做

到高效地访问文件。文件访问结束之后，没有必要再保留那些信息，于是，可以执行所谓的“关闭(close)”操作，释放这些信息占用的存储空间。

打开文件的系统调用open有两种格式：

```
#include <fcntl.h>
int open(char *filename, int flags);
int open(char *filename, int flags, mode_t mode);
```

其中，字符串*filename*提供一个文件名，*flags*在这里是个比特位图，可以使用<fcntl.h>中的定义的宏名字用|操作符连接起来，达到各种组合效果。但是，并不是所有的组合是有意义的。表4-2是open函数的*flags*参数取值的宏名字的简单说明。

表 4-2 文件打开方式

打开方式	作用
O_RDONLY	只读方式打开文件。
O_WRONLY	只写方式打开文件。
O_RDWR	读写方式打开文件。
O_CREAT	如果提供的文件名对应文件已存在，该位无意义。否则，创建新文件，只有这时候，open的第三个参数才有意义。第三个参数，是创建新文件时使用的访问权限代码。
O_TRUNC	如果文件原先有数据，打开时截为零。其实就是删光了原先的数据。
O_APPEND	每次向文件中写入数据，都写到文件尾。这个标志很有用，尤其是用于向已有的日志文件中追加数据，或者记账和计费数据。这个标志可以用于多个并发的进程都想写在同一个指定文件尾的情形。由于多个并发进程会有自己独立的当前写位置，使用这个标志打开文件，就可以做到每次写文件之前，首先寻找到文件尾，达到多个并发进程按时间累计数据的效果。为了达到这样的效果，每个进程都必须按这种方式打开文件。

如果在Windows上使用同样的函数，open中的*flags*还应当有O_BINARY。

第三个参数*mode*仅在创建新文件时使用，它是一个整数，用数字方式给出文件的存储权限。如：八进制C语言常数0666，表示任何人可以读写该文件。

函数返回值是一个整数，是新打开文件的“文件描述符”，以后的read和write都使用这个文件描述符。如果open返回-1，那么，文件打开失败。

UNIX还有个系统调用creat，注意，不是create,省掉了e。系统调用creat用于创建文件，在open中加O_CREAT标志，就能完成creat的功能，使得creat更少被用到。

为了使终端的I/O更方便，系统在启动一个程序时自动为程序准备好了三个文件描述符，分别是0，1，2，分别称作标准输入，标准输出，标准错误输出。所有这三个文件默认地指向当前终端。程序不必要执行open就可以直接使用这三个文件描述符。0是只读方式打开的文件描述符，对应当前终端的键盘输入；1和2是只写方式打开的文件描述符，对应终端的屏幕输出。如果程序用open打开其他文件，就会有文件描述符3，4等等。

2. 读写文件

读写文件的系统调用分别是read和write。

```
int write(int fd, void *buf, int len);
```

*fd*是已经打开的文件的文件描述符，*buf*是写缓冲区指针,指向要写的数据，*len*是期望写的字节数。返回值是真正写入的字节数。一般情况下，如果该字节数小于*len*,则说明发生了错误。如果系统调用的返回值是-1，那么说明系统调用失败。

```
int read(int fd, void *buf, int len);
```

*fd*是已经打开的文件的文件描述符，缓冲区*buf*是存放读来的数据的缓冲区指针，*len*是期望读的字节数。返回值是实际读的字节数。对于读操作来说，返回的字节数小于*len*是正常的，因为在文件尾时，当前剩余可读的字节数可能会小于*len*。如果系统调用的返回值是-1，那么说明系统调用失败。

read的返回值，如果为零，说明读到了文件尾部。由于UNIX允许多个进程同时访问同一个文件，一个进程在对文件进行写操作时，另个进程可以对该文件进行读操作。一个读操作已经返回0以后，意味着已到文件尾，随后，只要别的写操作会将更多的数据追加到文件尾部，那么，等待了一段时间后，后续的读操作仍然可以读出更多的字节。tail -f命令便利用这点实现。

使用read或者write读写文件时，每次读写的字节数太少，会导致效率低。表4-3是一个测试的例子，文件大小100M,硬件环境为Pentium III,主频750MHZ,LINUX系统。完成的操作为读取文件中的所有内容。当每次read操作传送的数据尺寸大于磁盘操作的缓冲区块大小时，效率变化很小。在注重效率的操作中，最好不要使用长度*len*过小的缓冲区*buf*。

表 4-3 不同大小缓冲区对文件操作效率的影响

缓冲区大小（字节）	读出整个文件的时间（秒）
1	130.45
4	33.56
8	17.44
16	9.64
64	6.21
256	5.83
512	5.71
1024	5.65
4096	5.64
8192	5.64
16384	5.64

在内核中，为每个打开的文件保留一个记录当前操作位置的指针。无论是read还是write,都从文件的当前位置读取或写入，操作之后，文件当前指针自动向后移动。这个当前指针也叫做文件的当前读写位置指针，读写共用一个指针，影响read和write操作的起始位置。这非常适合于按顺序访问文件中的数据。在文件访问中，顺序操作占绝大部分。在某些情况下，要求“随机访问”，可以按照任意的顺序读写。系统调用lseek提供了一种手段，用于强制设定文件的当前读写位置指针，下次的read或者write从这个位置开始。

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

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

其中，*fd*是文件描述符，*offset*是一个字节偏移量，函数返回值是下次读写位置。off_t是整数类型，不同系统中可能整数的位宽度不同，可能是32位整数，或者64位整数。off_t在头文件<sys/types.h>中定义。*whence*是指偏移量*offset*的参考基准，在<unistd.h>中有几个相关的宏定义。参见表4-4。

表 4-4 lseek 指定文件偏移量的方法

whence	文件读写位置指针的确定
SEEK_SET	读写指针移动到*offset*指定的位置（绝对值）。
SEEK_CUR	读写指针移动到当前读写指针加*offset*后的位置。
SEEK_END	读写指针移动到文件尾加*offset*后的位置，也就是文件中字节数加*offset*后的位置。

按照UNIX和C语言的惯例，文件的首字节编号为0号字节，而不是1号字节。后两种方式*offset*可以指定为负数。lseek甚至允许将文件读写位置指针移动到文件尾之后。如果超越了文件尾，随后在这个位置的写操作，可能会造成“文件空洞”。例如，有个5120大小的文件，占用10个存储块，lseek将文件指针定位到51200,然后写入512字节。那么，对使用系统调用操作文件的应用程序来说，能感觉到的是它拥有了一个51712字节大小的文件，其中编号5120~51199字节的内容都是字节0。但是，操作系统内核真正只为这个文件分配了11个存储块，0~9号和100号，中间的“空洞”不占空间，系统处理后让应用程序感觉不到空洞的存在。在空洞位置的写操作会导致分配真正的磁盘存储块。但是，在复制这个文件或者备份这个文件时，复制品会真正的占有较大的存储空间。没有特殊需求，尽量不要创建出含空洞的文件。程序中错误操作也可能会创建出含空洞的文件。

函数ftruncate可以截断一个文件，下面是这个函数的原型。

```
int ftruncate(int fd, off_t length);
```

文件截断后的剩余长度为*length*个字节。如果*length*为0，那么就将文件长度截短为0。

3. 关闭文件

关闭文件的调用很简单，提供一个“文件描述符”作为参数。

```
int close(int fd);
```

不使用的文件，应当关闭。每个程序允许同时打开的文件数目是有限的，整个系统允许打开的文件总数，也是有限的。每个有效的文件描述符都对应内核的部分资源。如果每次打开文件使用完毕后都不关闭，可能会导致以后的文件无法不开。

程序无论正常还是异常退出时，所有由程序打开但尚未关闭的有效的文件描述符，系统都会自动关闭。

文件访问的另一种方式是使用C语言的标准函数fopen，fprintf，fscanf，fgets，fread，fwrite，...，fclose，这些函数在C语言的教科书中有详细地介绍。这一组函数，是在系统调用open，read，write,...，close基础上构建的库函数。

4. 文件操作举例

【例 4-23】 使用系统调用函数编程序以十六进制打印数据文件的内容。

这里的例子是一个使用open/read/close方式访问文件的例子。程序比UNIX的命令od要简单得多。允许提供多个文件名，将文件内容以十六进制打印出来。打开文件时，有简单的出错处理。

```
$ cat myod.c
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
   int fd, k, i, len;
   unsigned char buf[512];

   for (k = 1; k < argc; k++) {
      fd = open(argv[k], O_RDONLY);
      if (fd == -1) {
         fprintf(stderr, "Open file \"%s\": %s (ERROR %d)\n",
            argv[k], strerror(errno), errno);
         continue;
      }
      while ( (len = read(fd, buf, sizeof(buf))) > 0) {
         for (i = 0; i < len; i++)
            printf("%02x ", buf[i]);
      }
      if (len < 0)
         perror("read data");
      close(fd);
   }
   printf("\n");
   return 0;
}
$ cc myod.c -o myod （编译链接生成可执行文件）
$ ./myod ../liang/.profile xyz
Open file "../liang/.profile": Permission denied (ERROR 13).
Open file "xyz": No such file or directory (ERROR 2).
$ ./myod *
90 36 59 ea 60 c4 90 a1 db 61 33 09 f6 b1 37 a9 d4
……
```

4.10.2 目录访问

在文件系统中，目录和普通文件一样，占用文件存储区中的空间。目录中的数据就是“文件名-i节点”对。能够像普通文件那样打开目录文件读取其中的数据吗？在早期的

UNIX中，的确就是用这样的方式获取一个目录中的所有文件名。每个目录项占16字节，包括两字节的i节点号和最长14字节的文件名。i节点号为0的条目，标志这一目录项已经失效，这是由于删除文件导致的。这也是i节点编号从1开始而不是从0开始编号的原因。这样，像一个普通文件那样用open以只读方式打开一个目录文件，read每得到16字节，就是一个目录项。程序自己来分析i节点号，文件名。使用了长文件名之后，目录文件的存储结构发生了变化，应用程序就必须了解目录文件的存储结构，这很不方便。因此，系统提供了几个库函数，用于操作目录表文件。用read读取目录表文件的方式，在LINUX中作了限制，但在大多数UNIX都允许这样的操作。尽管可以将目录表文件和普通文件一样打开读，但是对目录表，内核保留写目录的权利，任何用户态程序，就算是超级用户，也不允许以write调用改写目录内容，这样，保证其中数据的结构正确和有效。

【例 4-24】与普通数据文件相同的方式访问目录文件。

在AIX系统中执行上一节4.10.1中的应用程序myod，使用普通的文件打开open和读取read调用读取目录文件的内容。

```
$ ls -ia /usr/adm
 2048 .            2072 dev_pkg.fail   2055 ras          2071 wtmp
    2 ..           4098 ffdc           4119 sa           2103 xferlog
 8193 SRC          2102 ftp.pids-all   4096 streams
 4118 acct         4106 invscout       2073 sulog
 2049 cron         4097 nim            4101 sw
$ ./myod /usr/adm
08 00 2e 00 00 00 00 00 00 00 00 00 00 00 00 00 02 2e 2e 00 00 00 00
00 00 00 00 00 00 00 00 08 01 63 72 6f 6e 00 00 00 00 00 00 00 00 00
08 07 72 61 73 00 00 00 00 00 00 00 00 00 00 10 00 73 74 72 65 61 6d
73 00 00 00 00 00 00 00 08 17 77 74 6d 70 00 00 00 00 00 00 00 00 00
10 01 6e 69 6d 00 00 00 00 00 00 00 00 00 00 10 02 66 66 64 63 00 00
00 00 00 00 00 00 00 00 10 05 73 77 00 00 00 00 00 00 00 00 00 00 00
08 18 64 65 76 5f 70 6b 67 2e 66 61 69 6c 00 00 10 0a 69 6e 76 73 63 6f
75 74 00 00 00 00 00 00 20 01 53 52 43 00 00 00 00 00 00 00 00 00 00
10 16 61 63 63 74 00 00 00 00 00 00 00 00 00 00 10 17 73 61 00 00 00 00
00 00 00 00 00 00 00 00 08 19 73 75 6c 6f 67 00 00 00 00 00 00 00 00 00
08 36 66 74 70 2e 70 69 64 73 2d 61 6c 6c 00 00 08 37 78 66 65 72 6c 6f
67 00 00 00 00 00 00 00
$ od -c /usr/adm
0000000   \b  \0   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020   \0 002   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040   \b 001   c   r   o   n  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000060   \b  \a   r   a   s  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000100  020  \0   s   t   r   e   a   m   s  \0  \0  \0  \0  \0  \0  \0
0000120   \b 027   w   t   m   p  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000140  020 001   n   i   m  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000160  020 002   f   f   d   c  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000200  020 005   s   w  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000220   \b 030   d   e   v   _   p   k   g   .   f   a   i   l  \0  \0
0000240  020  \n   i   n   v   s   c   o   u   t  \0  \0  \0  \0  \0  \0
0000260     001   S   R   C  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
```

```
0000300  020 026   a   c   c   t  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000320  020 027   s   a  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000340   \b 031   s   u   l   o   g  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000360   \b   6   f   t   p   .   p   i   d   s   -   a   l   l  \0  \0
0000400   \b   7   x   f   e   r   l   o   g  \0  \0  \0  \0  \0  \0  \0
```

程序象处理普通的磁盘文件那样，打开目录读取目录文件的数据，根据数据目录表的存储结构判断文件名，i节点号。这很不方便，而且，目录表结构变化时，还需要重新编写程序。源程序也不便于在多种UNIX之间移植。UNIX提供了目录操作的库函数，隐藏了这些细节，使用起来更方便。这套函数的设计风格类似C语言访问文件的标准函数集。

```
#include <dirent.h>
DIR *opendir(char *dirname);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

提供给opendir一个目录名，得到一个句柄（返回值NULL,打开目录失败）。以后的操作，就直接使用这个句柄代替目录名。这点，跟fopen获得一个FILE *句柄类似。

readdir使用已经打开的目录句柄，每次调用获得一个目录项的数据。返回值是一个指针，指针指向的结构体记录了i节点号和文件名，分别放在dirent结构体的d_ino和d_name成员中。readdir返回NULL,标志目录表已经读到尾。

不再使用的目录句柄，用closedir关闭。

【例 4-25】目录表的访问举例。

下面的例子，使用目录操作的函数，给定一个目录名，列出该目录下的所有文件名和对应的i节点。类似ls -i的执行结果。

```
$ cat myls.c
#include <stdio.h>
#include <dirent.h>
#include <errno.h>

int main(int argc, char **argv)
{
   DIR *dir;
   struct dirent *entry;

   if (argc != 2) {
      fprintf(stderr, "Usage : %s <dirname>\n", argv[0]);
      exit(1);
   }

   dir = opendir(argv[1]);
   if (dir == NULL) {
      printf("Open directory \"%s\": %s (ERROR %d)\n",
         argv[1], strerror(errno), errno);
      return 1;
   }
```

```
    while ((entry = readdir(dir)) != NULL)
        printf("%d %s \n", entry->d_ino, entry->d_name);

    closedir(dir);
    return 0;
}
$ cc myls.c -o myls
$ ./myls /etc
4110 .
2 ..
4111 consdef
4112 csh.cshrc
4113 csh.login
4114 dlpi.conf
4115 dumpdates
4116 environment
4117 filesystems
4118 group
4250 inittab
4120 magic
4121 motd
8564 objrepos
4122 passwd
4123 profile
 ...
$ ./myls ../liaing
Open directory "../liaing ": No such file or directory (ERROR 2)
$ ./myls ../liang
Open directory "../liang": Permission denied (ERROR 13)
$
```

与目录表有关的操作函数还有许多，在此不再一一叙述。参见表4-5。

表 4-5 与目录表相关的操作函数

功能	函数原型
获取目录表的访问句柄	DIR *opendir(char *dirname);
读出目录中的一个目录项	struct dirent *readdir(DIR *dir);
关闭目录表的访问句柄	int closedir(DIR *dir);
删除文件	int unlink(char *pathname);
建立硬连接	int link(char *oldpath, char *newpath);
建立符号连接	int symlink(char *symbol, char *path);
文件改名	int rename(char *oldpath, char *newpath);
修改当前工作目录	int chdir(char *path);
当前工作目录名	int getcwd(char *buf, int size);
创建目录	int mkdir(char *pathname, mode_t mode);
删除目录	int rmdir(char *pathname);

4.10.3 获取文件系统的信息

使用df命令可以获得文件系统剩余的空闲块数目和空闲i节点数。在unix中还提供了系统调用ustat()以及一组statfs()函数，可以在程序中从文件系统的超级块中获得文件系统的统计信息，如同df命令的输出信息一样。这些调用，在不同的系统中会有些差异。

4.11 获取文件的状态信息

UNIX系统提供的系统调用stat，可以获取文件或者目录的状态信息，这些信息大部分来自于i节点中的管理信息。

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

int stat(char *path, struct stat *statbuf);
   /* return 0 on success or -1 on error */

int fstat(int fd, struct stat *statbuf);
   /* returns 0 on success or -1 on error */
```

这两个函数都用于从i节点中获得文件的状态信息。stat利用给出的路径名寻找指定文件的i节点；fstat利用给出的一个事先已经打开的文件描述符*fd*从内核中存储在内存中的活动i节点表中寻找文件中的i节点。如果使用缓冲文件I/O方式，用函数fopen()打开了文件FILE *f,调用宏fileno(f)可以获得已打开的缓冲文件的“文件描述符”。stat需要逐个分析路径名分量，最终查找到文件的i节点号，然后从磁盘的i节点区调入i节点，速度比fstat慢。但是，fstat仅适用于那些已经打开的文件。stat和fstat都将从i节点提取的状态数据放入调用者提供的结构体*statbuf*中。

结构体stat的定义在/usr/include目录下的头文件sys/stat.h中。

```
struct stat {
   dev_t   st_dev;
   ino_t   st_ino;
   mode_t  st_mode;
   nlink_t st_nlink;
   uid_t   st_uid;
   gid_t   st_gid;
   dev_t   st_rdev;
   off_t   st_size;
   time_t  st_atime;
   time_t  st_mtime;
   time_t  st_ctime;
};
```

随着系统的不同，可能会对上述结构体扩展，含有一些其它的与文件的管理信息相关的成员。前边列出的这些结构体的成员都是整数类型(16位整数，32位整数，或者64位整数)，为了系统移植的便利，不使用C语言的诸如short，int，或者long定义。以_t为后缀的这些类型定义，都在头文件<sys/types.h>中用C语言的typedef定义，例如：结构体的st_size成员存放文件的大小，单位为字节。32位无符号整数最大4G,有的系统允许大于4G的文件，那么在这些系统中，类型off_t就会被定义为64位整数。

st_dev: i节点所在设备的设备号。例如：SCO UNIX系统里，这是个16比特整数，两个字节，高字节为主设备号,低字节为次设备号。例：根文件系统下文件状态中的st_dev为0x0128,是根文件系统设备/dev/root的主设备号(0x01)和次设备号(0x28)。有了st_dev就可以知道文件存储在哪个设备的文件系统中。

st_ino: i节点的编号。

st_mode: 文件方式。总共16比特,含有文件的9比特基本存取权限，SUID权限和SGID权限各占一个比特；以及文件的类型，占用若干比特。在<sys/stat.h>中定义了若干个常量,可以判断文件的访问权限和文件类型。判断文件类型的方法，检查表达式st_mode & S_IFMT，可能为下列几种值之一:

S_IFREG	普通磁盘文件(Regular file)
S_IFDIR	目录文件(Directory)
S_IFCHR	字符设备文件(Character device)
S_IFBLK	块设备文件(Block device)
S_IFIFO	管道文件(先进先出FIFO)
S_IFLNK	符号连接文件(Symbolic link)

st_nlink: i节点的link数。

st_uid和st_gid: 文件主ID,用户组ID。在系统内部存储时不存储那些长度不定的字符串格式的用户名或者组名。这不利于节约存储空间和程序处理的时间。在创建用户和组的时候，每个组名字或者用户名字，都对应一个整数ID。系统提供一些名字和ID之间相互转换的函数。

st_size: 文件大小，文件中有效数据的字节计数。

st_atime: 最近访问(Access)时间。

st_mtime: 最近修改(Modify)时间。

st_ctime: 最近改变(Change)时间。

每个文件的i节点中都记录了这三个时间。一般的系统中，包括Windows 系统，记录时间都使用一个32比特的整数，时间精度为秒。32比特整数记录了事件发生时刻相对于格林威治时间1970年1月1日00:00:00的秒数。例如：整数1,000,000,000(1G),就是北京时间2001年9月9日上午09:46:40。系统提供一些函数，根据这个整数值和系统安装时设定的时区，推算出年月日时分秒。32比特表示的有符号整数最大值是±2G，到2038年1月后溢出，溢出后返回到1901年12月。如果系统用这种方式表示时间，会在2038年前后出现类似“千年虫”那样的问题。

文件的“访问(Access)”时间，指最后一次读取或者把文件当作一个程序文件加载执行的时间。这两个操作会影响文件的访问时间。

文件的“修改(Modify)”时间，最后一次写文件的时间。

文件的“改变(Change)”时间，最后一次写文件，会影响这个时间。另外，文件i节点记录的一些文件属性信息发生改变，这个时间也发生变化。例如：link数变了，访问权限变化，文件主或者组变化了，也会影响文件的“改变时间”，读和执行两操作不影响这个时间。

st_rdev: 当文件是字符设备文件或块设备文件时,该项中记录了设备号，包括主设备号和次设备号。例如：SCO UNIX中，st_dev占16比特，共两个字节，高位字节是主设备号，低位字节是次设备号。LINUX中是64比特,也是含有主设备号和次设备号。对于其它类型文件，该项无意义。

使用目录操作的一组函数(opendir，readdir，closedir)以及stat调用可以编写出象ls这样的命令。

使用文件操作的系统调用open，read……，或者，在这组系统调用之上的缓冲I/O函数库fopen，fgets，……，可以构筑出诸如od，wc，grep，cp，awk，...等命令。

4.12. 设备文件

这里简单介绍一下前边多次提到的设备文件，以及主设备号和次设备号。将设备组织的看起来同磁盘文件一样，可以通过文件系统里的文件名进行访问，是UNIX设备管理的一大特点。UNIX的设备分为块设备和字符设备。块设备面向磁盘等可以随机访问的存储设备，统一定义一种可以随机的根据块号进行操作的驱动界面，提供给文件系统使用，以便于使用UNIX的文件系统管理外存设备上的信息。文件系统中的命令mkfs，mount等，使用块设备文件名，映射块设备。除了块设备之外的所有设备，终端，打印机，等等，都算作字符设备，也叫做“原始(raw)”设备。所有的设备在文件系统中都拥有一个文件名，这是特殊类型的文件，在文件的i节点中记录了文件类型是块设备文件或者字符设备文件。每个设备都对应一个主设备号和次设备号，也记录在i节点中。按照惯例，设备文件都放在/dev目录下。使用ls -l时，对于设备文件，列出主设备号和次设备号。例如：

```
$ ls -l /dev/hdc1 /dev/fd[01] /dev/tty1
brw-rw---- 1 root   floppy   2, 0 Jan 30  2003 /dev/fd0
brw-rw---- 1 root   floppy   2, 1 Jan 30  2003 /dev/fd1
brw-rw---- 1 root   disk    22,1 Jan 30  2003 /dev/hdc1
crw------- 1 root   root    4, 1 Jun  2 20:32 /dev/tty1
```

对/dev/fd0来说，这行最左侧的字符b说明它是一个块设备文件，在普通文件列出文件大小的地方列出了主设备号2，次设备号0。

主设备号对应物理设备的一类。设备能够正常工作，需要提供设备驱动程序。所谓的“设备驱动程序(Device Driver)”是按照操作系统内核规定的接口标准事先编制好的一组子程序库，这些子程序库中含有多个事先约定好的函数入口，通过静态或者动态链接

的手段，加载到操作系统内核中，等待操作系统内核在合适的时机调用相关的函数。这些函数会以核心态模式工作。核心态的程序对所有内存操作没有限制，糟糕的驱动程序会使得系统崩溃。

例如，将一个含有八个RS-232异步串行通信口的PCI接口卡插入到主机箱里的PCI扩展槽中。设备驱动程序中有多个入口函数，假定是xxopen()，xxwrite()，xxclose()，xxintr()。设备驱动程序安装时注册这些函数，安装完成之后，会得到一个主设备号，假设是16。主设备号与驱动程序相关联，根据主设备号，可以检索到已经登记了的这个设备的设备驱动程序的入口函数。主设备号常常是内核中的一个数组下标，数组的每个元素是个结构体，每个结构体对应一个设备驱动程序，结构体的多个成员都是C语言的函数指针，指向加载到内核的驱动程序的函数xxopen()，xxwrite()，xxclose()，xxintr()等等。这个数组被称作“设备驱动程序入口表”。在文件系统中创建8个设备文件/dev/ttyE0 ~ ttyE7。那么，执行命令 who > /dev/ttyE0，系统会像普通文件操作一样通过open系统调用打开文件/dev/ttyE0，分析了文件的i节点，它不是普通磁盘文件，而是一个字符设备文件，于是，同样以主设备号16为数组下标，检索设备驱动程序入口表，找到open操作时应调用的函数的指针，函数指针指向设备驱动程序的函数xxopen()。调用这个设备驱动程序的xxopen()函数，设备驱动程序根据需要，对硬件作某些初始化等等，然后返回，open结束。who命令的输出，导致write系统调用，系统分析出这是对字符设备的操作，以主设备号16为数组下标，检索设备驱动程序入口表，找到write操作时应调用的函数指针，函数指针指向设备驱动程序的函数xxwrite()，调用设备驱动程序的函数xxwrite()。设备驱动程序通过函数被操作系统调用而得到数据，如何处理这些数据，就是设备驱动程序自己的任务。它可以根据这一特定硬件的功能，控制某些设备寄存器，将数据通过串行通信口一个个发送出去。设备产生中断时，操作系统调用设备驱动程序的xxintr()函数。主设备号被用来定位所使用的设备驱动程序，次设备号的作用很简单，用来区分这一类物理设备的哪一个。本例中有八个串行通信口，内核调用驱动程序入口函数时，提供了次设备号，驱动程序才知道该向哪个串行通信口输送数据。创建设备文件的命令是mknod,例如：

```
mknod  /dev/ttyE0 c 16 0
```

其中，/dev/ttyE0是要创建的设备文件名，c指的是字符设备，16是主设备号，0是次设备号。

如果一个设备驱动程序不操作任何物理硬件，也是完全可以的。它像普通设备驱动程序一样提供设备驱动程序入口，通过这些入口，可以和内核交互作用，得到某些数据。这些设备叫做“虚拟设备”。例如：设备文件/dev/mem和/dev/kmem可以通过某些操作获取内核中的一些数据表格内容，常常被随操作系统软件带来的一些运行在用户态的工具使用。

在LINUX系统中，虚拟设备文件/dev/random是一个随机数生成器，根据系统键盘和鼠标等外设的活动生成二进制的随机数数列。可以在用户自己编写的程序中使用这个设备文件。命令od -x /dev/random可以观察这些随机数序列。在系统外设鼠标都不

活动时，这个设备暂停供应随机数序列。另一个虚拟设备文件/dev/urandom，无论外设是否活动，只要你的程序读取该设备，都会源源不断地供应随机数序列。命令od -x /dev/urandom可以观察到这些随机数序列。

在1.3.2节中，介绍过终端设备文件。每个终端设备，无论是网络虚拟终端设备，还是一个真正的终端设备，都对应一个名字。可以按名称像普通文件那样使用终端，但是不方便的是无法决定当前终端到底是哪个终端。特殊的设备文件/dev/tty就是当前终端，而无论实际使用的终端到底是哪个。尽管scanf和printf等函数可以从当前终端获得输入和从当前终端输出，但是标准输入和标准输出的重定向功能，会使得程序不再从当前终端输入和输出数据。对于程序员来说，当程序在这种情况下仍然需要和用户交互的时，/dev/tty文件特别有用。程序可以打开/dev/tty文件输入和输出数据，在重定向环境中，仍然保持从当前终端输入和输出数据。

最著名的虚拟设备文件是/dev/null，所有的UNIX系统都会提供。所有写入该设备的数据统统被丢弃；任何时候都可以打开该设备读取，每次都读不到任何数据，就遇到文件结束，同读取一个含有0字节的磁盘文件效果一样。

【例 4-26】设备文件/dev/null 的使用举例。

(1) 自编的程序文件myap有许多输出，将输出显示在终端屏幕上会导致显示太多而且程序执行很慢，通过./myap > myap.log的方法，因为程序过多的输出挤占磁盘空间，这样可以使用命令./myap > /dev/null丢弃所有输出。需要丢弃程序输出时经常使用虚拟设备/dev/null。

(2) 检索当前目录以及当前目录的所有下属子目录中的C语言头文件，查找所有含有inode的行。这种需要“递归式”遍历一个目录树中的所有文件的操作，常常用find命令。可以使用下面的命令：

```
find . -name '*.h' -exec grep -n inode {} \;
```

然而，每次grep命令的执行，都处理一个文件。grep在处理只有一个文件的时候，-n选项仅仅打印出行号，不打印文件名，显然，find可能会命中多个文件，这样就无法知道是在哪个文件中找到了inode字符串。grep只有在两个或两个以上的文件检索时才打印文件名。这样，可以让/dev/null陪着，每次读取/dev/null都是一个空文件。下面的命令可以达到所期望的结果。

```
find . -name '*.h' -exec grep -n inode /dev/null {} \;
```

许多UNIX都提供虚拟设备文件/dev/zero，向这个文件中写入数据，如同写入/dev/null，所有数据被丢弃。和/dev/null不同的是从文件/dev/zero中读取时，程序的任何一次read系统调用，都会返回一连串的字节，每个字节都是二进制0字节。这个文件常常会用在程序中。命令od -v /dev/zero可以观察到这个效果。-v选项是

必需的(verbose)，否则，由于每行内容相同，od仅输出一个*号，代表数据和上一行相同，等待下行和上行不同时才打印。

4.13 文件和目录的权限

4.13.1 权限控制的方法

UNIX的每个文件和目录，都有一组权限与这个文件或者目录相对应，存放在它的i节点中，用于控制用户对它的访问。每个文件都有唯一的属主和组号，都记录在i节点中。这些信息在进行权限判断时使用。UNIX有命令chown和chgrp可以修改文件的属主和组，也有相应的系统调用函数chown(),在此不再介绍，用到的机会也不多，需要时查阅相关手册。

在i节点中记录的权限有9个比特，描述三个级别，分别是文件主，同组用户,其他用户。每个级别的权限都包括三部分：读权限，写权限，执行权限。

1. 普通文件的权限

普通文件的权限比较容易理解。如果用户对文件具有读权限，那么用户可以读取文件。如果用户对文件具有写权限，那么它可以修改文件的内容。如果用户对文件具有可执行权限，那么，就可以执行这个文件。

例如:将一个文件的权限设为不许读,不许写但可执行。那么,用户可以执行这个文件,但无法取走,或拷贝。可以将文件设置为可读，但是不可写，这样可以保护文件被误写。

2. 目录的权限

从UNIX的文件系统的组织结构上看，所谓“目录”，也组织成象普通磁盘文件一样的文件，其中存储了多个由“文件名-i节点号”组成的目录项。目录的读写权限，就是对这个目录表文件的读写权限。

如果对目录没有读权限，那么，目录表文件不许读，ls列出目录的操作会失败。

如果对目录没有写权限，那么，所有会导致目录文件修改的操作都被禁止。例:创建文件，删除文件，文件改名，拷贝文件到当前在目录下，创建子目录，删除子目录，这些操作必须在当前目录表中增加，删除，或者修改条目，这些操作都被禁止。许多人会想当然的认为，目录不可写，就可以保护目录下的所有的文件不可写，其实不然。在一个只允许读的目录下修改一个已经存在的文件，不需要修改目录表中的任何数据。但是，i节点中的数据和文件内容要发生变化，只要文件自身有可写属性，这些操作就能正常完成。类似的问题是，一个文件具备只读属性，但是，文件所处的目录又就可写权限，那

么，删除这个文件就完全可能。也就是说文件的只读权限可以保护文件不被误写，但不能保护文件被误删。因为删除文件，不需要打开文件写，只需要修改文件所处的目录，只要目录可写就可以了。rm命令会在删除一个只读文件时给出一个善意的提示，但是删除照样可以完成。

目录不可能像程序一样执行，所以，目录的执行权限位用作其它的用途。对目录有执行权限，意味着在分析路径名的过程中可以检索该目录。例: cat /a/b/c 那么,要求/，/a，/a/b三个目录必须有x权限，c文件有读权限；否则，命令执行失败。
cd ../stud4628那么，要求当前目录，上级目录和stud4628三个目录必须有x权限。

3. 访问合法性验证的顺序

用户在操作一个文件的时候，系统根据登录用户的用户名，组，以及文件i节点中存储的文件主和组，判断该使用三级权限的哪一级。判断的方法是：

① 若文件主与登录用户相同,则只使用文件主权限,不再查组和其他用户的权限。

② 若文件主与登录用户不同,但文件i节点中记录的组号与登录用户的组号相同，则只使用组权限,不使用关于其他用户的权限。

③ 若文件主与登录用户不同,文件i节点中记录的组号与登录用户的组号也不同,则使用文件关于其他用户的权限。

系统严格的按照上述的三条原则进行权限判断。因此，完全可以把文件权限设置为文件主不可读但同组用户可读,即使文件主是该组用户之一也不行。甚至可以将文件权限设置成文件主不许任何操作，同组用户可以读，其他用户既可以读也可以写。这样的设置显得很怪异，但是UNIX并不阻止那些怪异的用户这样做。

应当特别注意的是，对文件设置了权限，不是对文件中的信息进行了彻底的保护。超级用户root不受任何权限的制约，它可以肆意的读写系统中的任何文件。因此，非常私密性的个人信息，或者一些绝密信息，放到文件系统中，就有可能会被超级用户读取。为保护隐私，应当对信息加密，使得超级用户可以读取，但是不知道解密密码，信息还是无法还原。

4.13.2 查看文件和目录的权限

使用ls命令。有关选项-l和-d。 例如:

ls -l 可以查当前目录下所有文件和子目录的权限。

ls -ld . 列出当前目录自身的权限。

4.13.3 chmod: 修改权限

只有超级用户和文件主，才允许修改文件的存取权限。修改文件存取权限的命令是chmod。该命令有两种方式修改文件的权限。

1. 字母形式

命令格式为：

chmod [ugoa][+-=][rwx] *file-list*

描述用户级用一个或者多个下列的字母表示。

- u (user)文件主的权限
- g (group)同组用户的权限
- o (other)其他用户权限
- a (all)所有上述三级权限

下面的符号定义要执行的操作：

- + 给指定用户增加存取权限。
- - 取消指定用户的存取权限。
- = 给指定用户设置存取权限，其他权限都取消。

下面的字母定义存储权限：

- r 读权限。
- w 写限。
- x 执行权限

例如:

```
chmod u+rw *
chmod go-rwx *.[ch]
chmod a+x batch
chmod u=rx try2
```

2. 数字形式

数字形式的命令格式为：

chmod *mode file-list*

权限的描述，使用三个八进制数字，分别描述文件主，同组用户，其他用户的权限。这种方法将文件的权限强迫设置为某一个值。

例如：权限640,三个八进制数字分别6，4，0，分别是文件主，同组用户，其他用户的操作权限设置。文件主可读些，同组用户可读，其他用户不可读写。

八进制：　　6　　4　　0

二进制：　110　100　000

权限：　　rw-　r--　---

例：chmod 640 *.[ch] makefile *.log

【例 4-27】文件和目录的权限对文件和目录操作的限制。

下面是一个上机操作的例子，可以看出文件和目录权限的作用。

```
(文件的写权限)
$ who am i
jiang     pts/2      Jun 06 08:34
$ who > mydata
$ ls -l mydata
-rw-r--r--   1 jiang   usr           58 Jun 06 09:04 mydata
$ chmod u-w mydata
$ who >> mydata (只读文件不许写)
mydata: The file access permissions do not allow the specified action.
$ rm mydata (只读文件可以被删除)
rm: Remove mydata? y
$ ls -l mydata
ls: 0653-341 The file mydata does not exist.


(文件的读权限)
$ who > mydata
$ chmod u-rw mydata
$ cat mydata (无法读取不允许读的文件中内容)
cat: 0652-050 Cannot open mydata.
$ chmod 644 mydata


(目录写权限)
$ chmod u-w . (当前目录不许写)
$ who > mydata2 (不能创建新文件)
mydata2: The file access permissions do not allow the specified action.
$ who >> mydata (但是可以修改已有的文件,追加一部分数据)
$ rm mydata     (不能删除文件)
rm: 0653-609 Cannot remove mydata.
The file access permissions do not allow the specified action.
$ cp /etc/passwd mydata (可以覆盖旧文件)
$ cp /etc/passwd mydata2 (不能创建新文件))
cp: mydata2: The file access permissions do not allow the specified action.
$ mv mydata MyData  (文件不许改名)
mv: 0653-401 Cannot rename mydata to MyData:
    The file access permissions do not allow the specified action.
$ mkdir Test (不可创建子目录)
mkdir: 0653-357 Cannot access directory ..
.: The file access permissions do not allow the specified action.


(目录读权限)
$ pwd
```

```
/usr/jiang
$ chmod u-r .
$ ls (不可读的目录无法列表出其中文件)
ls: .: The file access permissions do not allow the specified action.
$ chmod 000 . (取消当前目录所有权限)
$ ls
ls: 0653-345 .: Permission denied.
$ chmod 755 . (试图恢复当前目录权限，但失败，因为试图访问当前目录下的.文件)
chmod: .: The file access permissions do not allow the specified action.
$ chmod 755 /usr/jiang (这种访问不需要当前目录的权限，可恢复当前目录权限)


(子目录 ttt 没有读写权限，但是保留了 x 权限)
$ chmod u=x ttt
$ cat ttt/ccp.c
main(int argc, char **argv)
{
   ...
}
$ rm ttt/arg.c (子目录没有写权限，不能删除其中的文件)
rm: 0653-609 Cannot remove ttt/arg.c.
The file access permissions do not allow the specified action.
$ ls ttt  (子目录没有读权限，不能列出其中的文件)
ls: ttt: The file access permissions do not allow the specified action.


(子目录有读写权限,但没有 x 权限)
$ chmod u=rw ttt
$ ls ttt
BUGS.report  arg.c       ccp.c       chap.h      mydata
arg         auth.c      chap.c      disk.img
$ cat ttt/arg.c
cat: 0652-050 Cannot open ttt/arg.c.


(试图设置其他用户的文件或目录的权限)
$ chmod 777 /
chmod: /: Operation not permitted.
$
```

4.13.4 umask: 改变文件创建状态掩码

1. umask 命令

用于决定文件和目录的初始权限。例如:用vi初创一个文件，或用ls -l>file.list创建文件file.list。文件的初始权限受umask值的限制。

umask是一个shell内部命令。在UNIX中，umask是进程属性的一部分，每个进程都对应一个umask值，umask命令用于修改shell进程自身的umask属性，同cd命令一样，只可能是内部命令。

例: umask　　打印当前的umask值

例: umask 022　　强制将umask值设置为八进制的022

掩码值的含义：在新创建的文件和目录中，不含有掩码值中列出的权限。

掩码值:　022

二进制:　000010010

掩掉的权限 ----w--w-

也就是说，所创建的文件和目录的初始权限，不含有对同组用户和其他用户的写权限，这些用户对新生成的文件不可写，不可修改新创建的目录的目录表。

比较严厉的命令是umask 077，除了文件主外，不许其他用户的访问。一般常将umask命令,放到自动执行批处理文件中,如:用csh作登录shell时,可以将umask命令加入到.login文件或.cshrc文件中; 用sh作登录shell时,可以将umask命令加入到.profile文件中。

2. 系统调用 umask

掩码值是进程属性的一部分，系统调用函数umask用于修改当前进程的掩码值。系统调用的函数原型为:

```
int  umask(int new_mask);
```

*new_mask*为指定的新掩码值,函数返回值为原先的掩码值。为了读出进程的掩码值,而不改变它,需要调用umask两次。

【例 4-28】umask 和 open 中的权限参数共同确定了新文件的初始权限。

设umask为077时,用C程序

```
fd = open(filename, O_CREAT | O_WRONLY, 0666);
```

open创建的文件的权限为0666,屏蔽掉077后为0600，即rw-------。可见，初始创建的文件的权限受open调用的规定值和进程自身属性的umask值影响。文件的权限只有在初始创建时受上述因素影响，一个已存在的文件的权限，不会受open/umask的影响。

系统调用chmod可以修改已存文件的权限,它不受umask的影响。和stat函数一样，提供两个版本的系统调用，函数原型为:

```
int chmod(char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

4.13.5 SUID 权限和 SGID 权限

前面讲述了文件和目录的权限，一个文件的权限分成三个级别，对于系统中的某一用户来说，他对这一文件中的全部内容要么有某一种访问权（如：读权限），要么没有这种权限。一旦该用户拥有了访问权限，那么这一用户就可以访问这一文件中的所有内容。这在有些场合下非常不方便。UNIX提供了SUID和SGID权限解决这样的问题，下面是使用SUID权限的例子。通过这个例子可以了解SUID权限解决问题的方法。

【例 4-29】SUID 权限的使用。

设用户liu记录了在本UNIX系统中的某些用户的月工资清单，记录在文件list.txt中，假定文件list.txt的内容如下：

```
$ cat list.txt
#=============================================================
# 登录名 工作证号  姓名   月份    工资    奖金    补助    扣除   总额
#-------------------------------------------------------------
  tian    2076   田晓星   03    1782   1500    200    175   3307
  liang   2074   梁振宇   03    1560   1400    180     90   3050
  sun     3087   孙东旭   03    1804   1218    106    213   2915
  tian    2076   田晓星   04    1832   1450    230    245   3267
  liang   2074   梁振宇   04    1660   1450    230     70   3270
  sun     3087   孙东旭   04    1700   1310    283    270   3023
#=============================================================
#         注：煤气费不再从工资中扣除，由煤气公司自行收缴。
```

对于这一文件，希望能够限制某一用户，如：用户liang，只能访问文件中的部分内容，而不是文件的全部内容。只允许用户liang读取行首字符为#的行和与用户liang有关的行，与其它用户有关的行对用户liang保密。这样，使用前面提到的文件权限设置方式，用户liang要么拥有对整个文件list.txt的读权限，要么不许他读取该文件中的任何内容。为了达到前述的保密要求，就不得不将该文件拆成若干个文件，每个用户拥有自己的文件，并且还要设置权限以防其他用户读取。这对于用户liu来说，很不方便，对这么多的文件进行修改和维护非常困难。需要有一种机制限制用户liang只能访问用户liu的文件list.txt中的一部分内容，UNIX提供了这样一种机制。

先看下面用户liu编写的C源程序query.c。

```
$ awk '{printf("%2d %s\n",NR,$0)}' query.c
 1 #include <stdio.h>
 2 #include <string.h>
 3 void main(void)
 4 {
 5   FILE *f;
 6   char line[512], login_name[16];
 7   f = fopen("list.txt", "r");
 8   if (f == NULL) {
 9      perror("*** ERROR: Open file \"list.txt\" ");
10      exit(1);
```

```
11    }
12    while (fgets(line, sizeof(line), f)) {
13       if (line[0] == '#') {
14          printf("%s", line);
15       } else {
16          if (sscanf(line, "%s", login_name) > 0) {
17             if (strcmp(login_name, getlogin()) == 0)
18                printf("%s", line);
19          }
20       }
21    }
22    exit(0);
23 }
$ cc query.c -o query
$
```

为了在列出源程序文件时每行左边打印行号，这里使用了awk命令。UNIX中有个简单的nl命令(number lines)，可以在列出文件时每行增加行号，但没有awk更灵活，本书后面的源程序例子列出时，都使用awk命令。

函数调用getlogin()返回当前用户的用户名。sscanf类似fscanf, scanf,只是sscanf从字符串中获得输入，而fscanf和scanf分别从文件和标准输入获取数据。

源程序经过编译后，生成可执行文件query。为了保密期间用户liu将他的文件list.txt的权限设为rw-------，文件query的权限为rwx--x–x，下面的内容是用户liu执行的操作。

```
$ chmod 600 list.txt
$ chmod 711 query
$ ls -l list.txt query
-rw-------   1 liu   leader    722 Dec 10 23:04 list.txt
-rwx--x--x   1 liu   leader   56134 Dec 10 23:07 query
$
```

这样，用户liang执行命令query，结果如下。用户liang也无法用cat列出文件list.txt的内容。下面的内容是用户liang执行的操作。

```
$ query
*** ERROR: Open file "list.txt" : Permission denied
$ cat list.txt
cannot open list.txt: Permission denied
$
```

可见，用户liang在执行用户liu的程序query时，由于用户liang没有对用户liu的文件list.txt的读权限，在源程序的第7行为读而打开文件时失败，执行第9-10行，程序退出。

在UNIX中文件query的文件主用户liu可以给文件query增加SUID权限（设置用户标识）。用户liu的操作过程如下：

```
$ chmod u+s query
$ ls -l query
-rws--x--x   1 liu   leader   56134 Dec 10 23:07 query
$
```

增加了SUID权限以后，ls -l命令显示的文件主对该文件的操作权限为rws,在执行权限处显示字母s而不是字母x。

文件query的SUID权限，由它的i节点中的一个比特来记录。在执行chmod u+s命令给一个文件施加SUID权限时，该文件对文件主必须有可执行权限。

这样，用户liang就可以通过query命令查询只许liu可读的文件list.txt，尽管用户liang没有对文件list.txt的读权限。用户liang的操作过程如下：

```
$ ls -l list.txt query
-rw-------   1 liu   leader    722 Dec 10 23:04 list.txt
-rws--x--x   1 liu   leader   56134 Dec 10 23:07 query
$ query
#============================================================
# 登录名 工作证号  姓名   月份    工资    奖金   补助   扣除   总额
#------------------------------------------------------------
  liang   2074   梁振宇   03    1560   1400   180    90   3050
  liang   2074   梁振宇   04    1660   1450   230    70   3270
#============================================================
#        注：煤气费不再从工资中扣除，由煤气公司自行收缴
$ cat list.txt
cannot open list.txt: Permission denied
$
```

用户liang仍无法用cat命令列出文件list.txt的全部内容。文件主liu对可执行文件query授予SUID权限意味着，无论哪个用户只要拥有对文件query的可执行权，query程序执行时，使用文件主liu的权限来访问文件。这样，用户liang虽然无法通过其它程序如cat命令读取文件list.txt的内容，但是可以通过query来查询文件list.txt的内容，他可以看到的内容受限于可执行文件query内部的程序设计。使用这种方法，限制了其它用户只能以文件主提供的程序来访问文件。Dennis Ritchie发明的这种方法，很简单的解决了很多安全问题。这种“简单”在于操作系统处理起来很简单，它仅仅提供了这样一种机制，在这种机制下，应用程序自己编写应用程序，有充分的灵活性，可以按照自己需要的访问策略访问文件。

UNIX每启动一个程序，系统就创建一个进程，每个进程都有两个UID。UID是用户标识号，每个用户都有一个名字，在系统内部对应一个标识号，是一个整数，同名字一样UID代表一个用户。进程的两个UID是：实际的UID和有效的UID，记录在内核进程proc结构中的p_uid域和p_suid域。一般情况下，用户启动一个可执行的程序时，进程的实际UID和有效UID相等，就是用户自己。进程在打开一个文件时，文件系统将根据进程的有效UID而不是实际UID，与文件所有者UID之间的关系和文件自身的权限进行访问合法性验证。当一个可执行程序有SUID权限后，启动这一程序创建新进程，新进程的实际UID和有效UID就不再相等，实际UID就是启动这一程序的用户的UID，而有效UID为可执行文件的文件主的UID，而UNIX对文件访问权限的检查使用进程的有效UID。这使得一个用户可以通过文件主提供的程序来访问一个文件主未授予他访问权限的文件，这种访问依赖于文件主提供的程序，进行有限的访问。

UNIX系统提供的/bin/passwd命令，就是使用SUID权限的例子，允许任何用户执行这个命令，修改用户自己的口令，而所有的加密后的口令都存储在仅对超级用户有访问权限的数据文件中。每个用户在修改口令时，仅可以修改属于他自己的口令字段。

删除一个文件的SUID权限使用chmod u-s命令。例：

```
chmod u-s query
```

SGID权限（设置组标志）与SUID权限类似。一个拥有SGID权限的可执行文件在执行过程中，当需要进行文件访问合法性验证时，如果需要判断进程是否与被访问文件的文件主同组时，进程的组标识符GID将使用可执行文件的文件主的组GID，而不是象普通程序的执行那样使用启动这一命令的用户的组GID。

增加SGID权限时，文件必须事先有组执行权限，命令格式为：

```
chmod g+s file-list
```

删除SGID权限的命令：

```
chmod g-s file-list
```

有SGID权限的文件在ls -l列表时，组执行权限处显示小写字母s，而不是x。如果在ls -l列表时，组执行权限处显示大写字母S,则不是SGID权限，而是与文件和记录锁定有关。与文件和记录锁定有关的命令还有chmod +l，这在有关文件和记录锁定的7.9.4节中介绍。

第 5 章 C-shell 的交互功能

5.1 UNIX 的 shell

UNIX系统的重要特性之一就是提供了大量的公用程序，这些程序都经过精心设计，完成一定的功能，作为UNIX的命令。UNIX的命令手册提供这些命令的使用方法。这些命令，在系统中运行时，同程序员自编的应用程序处在相同的地位，都是利用操作系统内核(Kernel)提供的系统调用，完成处理任务。在这些公用程序种，有一组被称为外壳(shell)的程序，shell的取名相对于kernel。shell程序是用户和系统之间的接口，是一个交互式命令解释器。用户通过它可以输入命令，然后shell调用这些命令，利用kernel的功能，完成用户的任务。shell中的许多功能和特点也都来自于kernel，例如，管道，重定向，执行命令创建新进程等等。除了可以交互式的输入命令之外，shell还是一种程序设计语言，提供了变量，循环结构和条件结构，用户可以通过它设计批处理的命令。UNIX提供了几种shell,主要有：

(1) /bin/sh

B-shell,由Stephen R. Bourne在贝尔实验室开发，是最早被普遍认可的shell,也是UNIX的标准shell，设计得非常简练。它的风格被后来出现的其它shell所继承，影响很大。但是，它的命令行编辑功能很弱，交互操作起来非常不方便。

(2) /bin/csh

C-shell,最先由加利福尼亚大学的William N. Joy(也叫Bill Joy)在20世纪70年代开发，最初运用在BSD2.0版本的UNIX，是Berkeley UNIX的主要特征之一。Joy在1982年和斯坦福的三人共同创办了Sun Microsystems。C-shell提供了历史机制和别名替换，相对B-shell来说交互起来更方便。在编程方面也更灵活，许多编程结构的风格类似C语言，所以取名C-shell。

(3) /bin/ksh

K-shell,由贝尔实验室的David Korn在1986年开发。是B-shell的超集，支持带类型的变量，数组，等等，同sh相比，提供了更强的功能。

(4) /bin/bash

Bourne Again shell, 这是LINUX上的标准shell, 它兼容Bourne Shell,并且在标准B-shell上进行了扩展，吸收了C shell的某些特点。它的命令行编辑方法非常方便，可以直接使用键盘上的上下箭头等全屏幕编辑操作的功能键，便于交互式操作，得到许多用户的喜爱。但是，bash没能在很多其他UNIX中作为标准工具提供。

每个用户登录成功后，进入shell。这里的shell种类选择，由系统管理员在创建用户时设置，也可以进行修改。在配置文件/etc/passwd中记录了每个用户的用户名，用户ID,组号，主目录，注册shell等。超级用户直接编辑这个文件，也可以修改用户的注册shell。下面是/etc/passwd中的若干行：

```
tian:x:1289:100::/usr/tian:/bin/csh
jiang:x:1306:100::/usr/jiang:/bin/sh
liang:x:2167:100::/usr/liang:/bin/ksh
sun:x:1283:100::/usr/sun:/bin/sh
```

象众多的编辑软件一样，每种shell也有自己忠实的爱好者和不认同者。在一般的UNIX中，sh和csh都作为标准命令提供。在本章，介绍csh的便于交互使用的特点，不再介绍csh编程。在第6章中主要介绍标准B-shell编程。

有必要在这里介绍一下“内部命令”和“外部命令”的概念。使用过DOS的用户都熟悉“内部命令”和“外部命令”，例如：DIR，COPY，DEL等命令是内部命令，不需要磁盘上有一个类似DIR.COM或者DIR.EXE的程序文件，就可以直接执行DIR命令，而XCOPY就是一个外部命令，必须在搜索路径PATH列出的某个子目录中有XCOPY.EXE文件，才能执行。所谓的内部命令，就是输入了命令之后，直接由shell“内部”来理解的命令，不需要到系统中搜寻可执行程序文件。有的命令必须设计成内部命令，前面介绍过的命令中，cd命令和umask命令都是内部命令。诸如ls，cp，rm等命令，在UNIX中都是外部命令。既然“内部命令”是由shell解释的，那么，内部命令与具体的shell相关，不同的shell支持不同的内部命令集。外部命令没有这样的限制。有的shell为了提高执行效

率，会把一些外部命令实现为同等语义的内部命令。例如：test，echo原来都是外部命令，许多shell把它们实现为内部命令。

5.2 csh 启动与终止

csh启动时，将自动执行用户主目录下.cshrc文件中命令。如果它作为注册shell运行,再执行主目录中.login文件中命令。不同的用户有自己独立的主目录(Home Directory),所以，不同用户有自己独立的.cshrc文件。这些特性类似DOS中的AUTOEXEC.BAT。许多程序一启动就会搜寻一个批命令文件，这些文件的命名大都以圆点开头， 如vi的.exrc，mail的.mailrc，B-shell的.profile。

作为注册shell的csh终止时，执行主目录下 .logout文件中的命令。

如果你的系统不是以csh作为注册shell启动的，那么，直接执行命令csh就可以进入一个交互式的c-shell。

5.3 使用 csh 的历史机制

所谓历史机制，指的是csh将最近一段时间内输入的命令保存起来，这样就可以重复使用前面已经输入的命令，或者前面的命令有错时，用一种简化的方法修改，而不需要将命令重新输入一遍。

5.3.1 历史表大小

以前键入的命令行被存在历史表中,该表的大小由csh的变量history设定。使用内部命令set history=30，可以设定历史表大小为30个命令行。许多系统默认的历史表大小为1。

5.3.2 查看历史表

用csh的内部命令history，可以列出历史表的内容。例如：

```
% history
    87  ls -l /etc/passwd
    88  vi /etc/passwd
    89  ls -l /bin/passwd
    90  man passwd
    91  vi /etc/security/passwd
    92  cd /usr/include
    93  grep -n termio *.h
    93  history
```

左侧的编号是命令号，csh为每个命令分配一个编号，csh的提示符一般是%.

5.3.3 引用历史机制

引用历史机制的方法，是shell进行“历史替换”。与历史替换有关的符号是!。表5-1列举了一些典型的历史机制的引用方法。

表 5-1 C-Shell 历史替换的方法

引用方法	历史替换操作
!!	引用上一命令(如同DOS中按F3键)
!*str*	引用 以*str*开头的最近使用过的命令。如:用!v引用刚不久执行过的命令vi disp_stat.c命令；用!fin引用最近用过的find . -name core -print命令；用!cc引用最近用过的cc disp_stat.c -o disp_stat命令；用!.引用最近用过的./disp_stat命令。
!45	引用历史表中第45号命令。
!20:s/*str1*/*str2*/	把历史表中第20号命令中的*str1*串替为*str2*后执行。这种方法可以修改一个以前用过的命令。
^*str1*^*str2*	把历史表最后一行中*str1*串替为*str2*串,并执行。这种方法可以直接修改刚刚键入命令中的错误。例如：刚刚输入的命令find . -nmae core -print，可以用^nmae^name 修改。
!55:2	引用55号命令的第二个单词。命令行中的单词从左向右编号0，1，2...。如果55号命令是上述的find命令。那么命令 ls -l !!:3实际上执行 ls -l core
!55:^	引用55号命令的第一号个单词
!55:$	引用55号命令的最后一个单词
!55:2-4	引用55号命令的第2~4号单词

5.4 别名

使用csh的别名机制，可以为一个我们经常使用的命令取一个别名，帮助提高效率和减少重复劳动。UNIX系统中最常用的命令的命令名长度2~3个字符。可以为那些一段时间内经常使用的命令取一个单字符的别名，简化命令。当然，别名可以是多个字符。与别名有关的csh内部命令是alias和unalias。

5.4.1 在别名表中增加一个别名

csh的内部命令alias用于定义一个个别名。例如：

```
alias h    'history'
alais t    'tail -f /usr/adm/pppd.log'
alias l    'ls -Fl'
alias rm   'rm -i'
alias dir  'ls -Flad'
alias type 'cat'
```

```
alias n    'netstat -p tcp -s | head -10'
alias r    'netstat -rn'
alias p    'ping 202.143.12.189'
alias rt   'traceroute 217.226.227.27'
```

通过定义一些别名，可以不用输入这么复杂的命令。其它的命令，前面都介绍过。后三个命令都是与网络相关的命令，netstat -rn打印当前的IP路由表(r:route路由，n:numeric数字格式显示IP地址，而不是显示名字)。ping命令测试与对方主机的连通性。traceroute试图探测出从本地出发到目的地的路径。这三个命令在Windows中也有相应版本的命令NETSTAT,PING和TRACERT。如果网络状态不稳定需要调试，那么这些命令就会需要反复键入，为它们选择一个简短的别名，更能够提高效率。

上面的rm被定义为别名，从而，在csh中键入rm已经不再是原来意义上的rm,在这里对rm重新进行了定义。利用命令的同名别名进行重定义功能，带来了很多方便。当你发现某个命令执行起来和从前不同，有必要检查别名表，看看是否已经通过“别名替换”被偷梁换柱。

用户自己定义的别名，会提高自己的工作效率。然而，这些别名设置在csh中止后就不再存在。最好将这些别名设置，连同前面介绍的set history=30这样的命令放到.cshrc文件中，每次执行csh时重新设置。

5.4.2 查看别名表

使用不带任何参数的alias命令，能打印出当前的别名表。

5.4.3 给别名传递参数

许多命令带有参数，使用别名带有参数时，就将参数传递给命令。例如：

```
alias dir 'ls -Flad'
dir ~  那么，实际执行ls -Flad ~
```

在csh中，~是个特殊符号，它将被csh展开为当前用户的主目录。这种处理是csh的处理，其它的shell未必会这样做。

这种简单的参数传递很容易理解，需要传递的参数都排列在别名定义之后。但是，有时这种方法并不能满足我们的需求。

【例 5-1】给别名命令传递参数的方法。

下面的命令递归式地在系统头文件目录下检索所有的头文件，查找含有termio字符串的程序行。

```
find /usr/include -name "*.h" -exec grep -n termio {} /dev/null
\; | more
```

键入这么长的一串命令很费力气，而且，一段时间内会经常用到这个命令，不如定义一个别名用起来简便。但是，每次检索的字符串不同。这次检索termio，下次检索的是tcp_hdr，或者别的字符串。使用上述的简单别名无法达到要求。

别名参数的使用，是和历史机制相同的方式。用一个惊叹号表示当前的输入。和历史命令一样，在冒号后边的数字或者^号$号代表参数号。

```
alias f 'find /usr/include -name "*.h" -exec grep -n \!:$ {} /dev/null \; | more'
```

这里的惊叹号前面加上转义符\,使得csh不再把他们解释为历史替换，而是把惊叹号作为一个字符，传递给alias命令，alias真正得到的是!:$。

这样，有了别名，直接使用f termio就可以了。

也可以传递多个参数。

```
alias scan 'find \!:3 -name "*.h" -exec grep -n \!:1 {} /dev/null \; | more'
```

引用时，使用下面的命令能达到前面的命令相同的效果：

```
scan termio in /usr/include
```

5.4.4 取消别名

使用内部命令unalias，可以取消别名。用法是：

unalias 别名

例如：unalias n　在别名表中取消别名n。

5.5 csh 提示符

csh的提示符默认是%，用户可以自己使用csh的变量prompt控制，用户可以自定义csh提示符。prompt变量值是一个字符串，在这个字符串中的!，shell实际显示命令提示符时会以命令号代替。

```
set prompt="[\!]%"
```

在!前加\以取消csh对!的特殊解释，对变量prompt赋值为由四个字符构成的[!]%字符串。

5.6 csh 的管道和重定向

在交互式命令中，经常会使用到管道和重定向。标准输入和标准输出的输入重定向以及管道，在不同的shell中，用法都相同。例如：

```
ls -l | more
man ls > man.paper
man rm >> man.paper
tr UVW uvw < file1 > file2
```

重定向符号>>是向文件中追加。除了标准输入和标准输出的输入重定向和管道之外，标准错误输出的重定向在不同的shell之间，有所不同。标准输出和标准错误输出，看起来都是输出到终端屏幕上，但是有所不同。

例如：C语言的程序员很容易会发现标准错误输出和标准输出的不同之处。

```
cc  myap.c -o myap | more
```

这样的命令并不能使得编译程序产生的错误信息停下来逐屏显示，而是照常滚动。同样的

```
cc  myap.c -o myap > err.list
```

也不能把编译程序产生的错误信息写到文件err.list中，而是照常在屏幕上滚动显示。究其原因，上述的两个命令的管道和重定向操作，第一个命令是将cc命令的标准输出管道到命令more的标准输入，第二个命令是将标准输出重定向到文件err.list，而cc给出的错误信息是在标准错误输出上输出的，所以根本不起作用。

5.6.1 标准输入,标准输出,标准错误输出

C语言中操作文件有两种模式，一种是标准的缓冲I/O方式，一种是低级的直接使用系统调用的方式。使用缓冲方式时，在stdio.h中已经声明了三个变量，分别对应标准输入，标准输出，标准错误输出。

```
extern FILE *stdin, *stdout, *stderr;
```

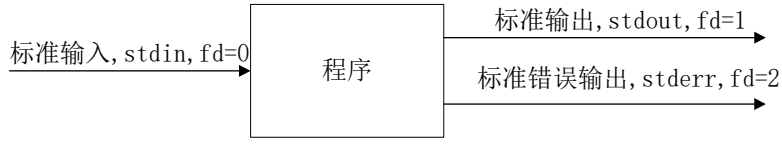

图 5-1 标准输入,标准输出,标准错误输出

这三个变量对应的存储区在C语言的函数库中分配，而且，在main函数调用之前已经赋值好，分别与当前终端的键盘输入，屏幕输出相关联，见图5-1。程序员可以直接使用这三个缓冲文件的句柄stdin，stdout和stderr输入/输出数据。执行一条命令，在屏幕上产生的输出，到底属于标准输出，还是标准错误输出，要看实现这一命令的程序是如何编制的。下面的例子，在屏幕上输出的所有STRING1属于标准输出，所有的STRING2都属于标准错误输出。C语言中scanf()，getchar()从标准输入获取数据。

【例 5-2】使用缓冲 I/O 的标准输出和标准错误输出。

```
#include <stdio.h>
main()   /* 使用缓冲 I/O */
{
   static char  *str1= "STRING1\n";
   static char  *str2= "STRING2\n";
   int i;
   for (i=0;i<20;i++) {
      printf(str1);  /* 或:fprintf(stdout,str1); */
```

```
        fprintf (stderr,str2);
    }
    exit(0);
}
```

直接使用系统调用的方式操作文件，文件描述符0,1,2分别对应标准输入，标准输出和标准错误输出。这三个文件描述符对应的文件，分别与当前终端的键盘输入，屏幕输出相关联。程序员可以直接使用它们read或write数据。

【例 5-3】使用原始 I/O 的标准输出和标准错误输出。

下面的例子与例5-2有等价的效果。

```
main()      /* 使用原始 I/O */
{
    static  char  *strl= "string1\n";
    static  char  *str2= "string2\n";
    int i;
    for(i=0;i<20;i++) {
        write(1,strl,strlen(str1));
        write(2,str2,strlen(str2));
    }
    exit(0);
}
```

5.6.2 标准输出和标准错误输出重定向

1. 未设置 csh 变量 noclobber 情况下

未设置csh变量noclobber情况下，重定向用法有以下几种：

用法：>文件

把stdout重定向到到一个文件中，但不影响stderr仍然在终端输出。见图5-2。

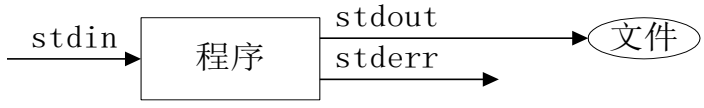

图 5-2 标准输出重定向

用法：>& 文件

把stderr合并到stdout,然后又重定向到一个文件中，见图5-3。

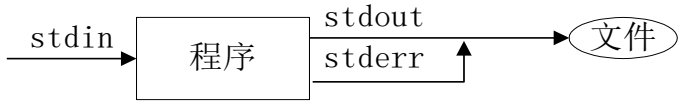

图 5-3 标准输出和标准错误输出合并后重定向

例如：cc myop.c -o myop >& err.list

注意，在>和&之间不许有多余的空格字符。

用法：>和>&

若文件不存在则创建新文件,否则覆盖掉原文件。

用法：>>文件　>>&文件

若文件不存在,则创建;若文件已存在则追加到文件尾。

2. 设置 csh 变量 noclobber 情况下

执行命令ls -l > filelist时，如果文件filelist事先已经存在，shell首先删除文件中的所有数据，使文件长度为0，然后把命令的标准输出的数据流存入到该文件中。文件被覆盖，先前的内容全部丢失，这被称作clobber(击毁)一个文件。为了防止这样的操作会无声息地覆盖掉重要的文件，csh提供了一个变量noclobber防止这种现象发生，给予重定向的文件一个简单的保护机制。设置和取消这个变量的csh命令分别是：

```
set noclobber
unset  noclobber
```

查阅这个变量值的命令是不带任何参数的set命令。set命令是csh的内部命令。

设置csh变量noclobber情况下，重定向用法有以下几种：

用法：>文件　>&文件

若文件已存在则出错；否则创建。

用法：>!文件 >&!文件

若文件已存在则强制覆盖掉原文件；否则创建。

用法：>>文件 >>&文件

若文件不存在则出错；否则追加到尾部。

用法：>>!文件 >>&!文件

若文件不存在，强制创建；否则追加到尾部。

5.6.3 管道

1. 标准输出管道 |

用法: 命令1 | 命令2

例如：`ls -l | more`

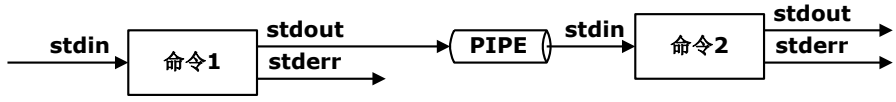

图 5-4 将标准输出管道到下个命令

把前面命令的stdout管道成下一命令的stdin，不影响stderr输出，见图5-4。

2. 标准输出和标准错误输出合并后管道 |&

用法: 命令1 |& 命令2

把stderr合并到stdout然后管道到下一命令，见图5-5。

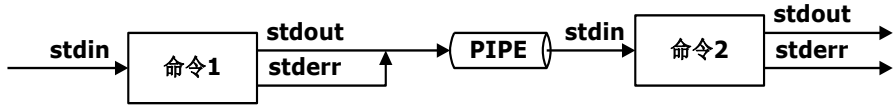

图 5-5 将标准输出和标准错误输出合并后管道到下个命令

举例:

```
cc myjob.c -o myjob | more
```

若有100个错误行,不能逐屏显示错误信息。

```
cc myjob.c -o myjob |& more
```

可以逐屏显示错误信息。

第 6 章 B-Shell 及编程

shell是对用户提出的运行程序请求进行解释执行。对于UNIX用户来说，文本编辑程序和shell程序是最常用的程序。本章介绍B-shell的特性，包括元字符，引号，shell的变量替换，命令替换，文件名生成，以及shell变量，流程控制，子程序。使用shell可以编写脚本程序,在交互方式下也可以使用shell的流程控制编写复合命令。熟悉C语言或者PASCAL语言等其它算法语言的shell用户，通过对shell的学习，了解shell编程风格和一般的算法语言之间的区别。shell是面向命令处理的语言，shell提供的流程控制结构是通过对一些内部命令的解释实现的，另外一些灵活的功能是通过shell替换实现。至于流程控制所需要的条件判断，shell仅仅提供了一种如何表示条件的机制，条件判断的具体方法则由shell之外的命令根据自己的要求完成，因而，提供了更大的灵活性。如同C语言的设计思路一样，shell本身设计得非常精炼，但是它提供了灵活的机制。

6.1 启动 B-shell

6.1.1 启动一个交互式 B-shell

当注册shell不是B-shell(如`csh`)时,在命令提示符下键入`sh`命令,即进入了B-shell。

当`sh`作为注册shell被启动时,会自动执行用户主目录下的`.profile`文件中的命令，记作`$HOME/.profile`。类似`umask`之类的命令，应当写在`.profile`文件中。

6.1.2 #!/bin/sh:脚本文件的执行

就像一部戏剧的排演需要一个剧本，剧本中预选设定了演员出场顺序和对话先后顺序。shell脚本(script)是预先定义好的一个命令序列，由需要执行的命令构成的文本文件。在DOS系统中，被称作“批处理文件(batch)”。

脚本文件本身是一个文本文件，文本文件本身是由ASCII码构成的，不象普通的命令文件那样由CPU可以执行的指令代码构成，不可能直接执行。当一个文件具有可执行属性，用户将它执行的时候，系统判断这个文件的格式，首先看它是不是一个经源程序编译链接后产生的，这种文件都满足系统的某一种特定格式。如果是，那么系统就加载这一程序，启动执行这一程序文件中的CPU指令。否则的话，系统会认为这是一个脚本文件，启动shell程序文件`/bin/sh`,运行`/bin/sh`文件中的CPU指令来解释执行脚本文件中的文本。

UNIX的shell除了/bin/sh外，还有其它的ksh，csh等，每种shell都可以有自己的文本格式的脚本文件。这就要求，对脚本文件的解释程序应当有一种可以选择的方法。为此，系统规定，如果脚本文件的第一行的头两个字符是#!，那么，就用这行后面的说明启动一个命令来解释这个脚本文件中的文本。

如:用vi或者其它的文本编辑器，编辑下列的脚本文件lsdir，并且使用chmod u+x lsdir赋予文件可执行属性。

```
#!/bin/sh
if [ $# = 0 ]
then
   dir=.
else
   dir=$1
fi
find $dir -type d -print
```

由于脚本文件lsdir首行#!/bin/sh的开头两个字母是#!,那么，执行命令./lsdir时，系统就启动/bin/sh程序来解释脚本文件的内容。如果第一行是#!/bin/ksh或者#!/bin/csh,那么，就启动/bin/ksh或者/bin/csh来解释脚本文件中的文本。#!必须位于脚本文件的第一行的开头两个字符。也就是说必须是文件的前两个字节。在#!之后甚至可以跟一个用户自己的可执行程序，那么系统就会用用户规定的这个程序来解释脚本文件的文本。对#!的理解不是由shell完成的，而是7.1.3节将要介绍的负责加载程序文件运行的系统调用exec()的职责。

【例 6-1】使用#!为脚本文件自设定解释程序。

```
$ cat lsdir
#!/bin/od -c
if [ $# = 0 ]
then
   dir=.
else
   dir=$1
fi
find $dir -type d -print

$ chmod u+x lsdir
$ ./lsdir
0000000   #   !   /   b   i   n   /   o   d       -   c  \n   i   f
0000020   [       $   #       =       0       ]  \n   t   h   e   n
0000040  \n           d   i   r   =   .  \n   e   l   s   e  \n
0000060   d   i   r   =   $   1  \n   f   i  \n   f   i   n   d       $
0000100   d   i   r       -   t   y   p   e       d       -   p   r   i
0000120   n   t  \n  \n
```

脚本文件中第一行如果缺少了#!，那么，系统就会用默认的shell程序来解释执行脚本文件的文本。一般系统的默认shell是/bin/sh，LINUX的默认shell是/bin/bash。

有三种方法可以执行脚本文件。

(1) `sh < lsdir`

(2) `sh lsdir`
`sh lsdir /bin`

(3) `chmod u+x lsdir` 给文件`lsdir`可执行属性
`./lsdir`

6.2 重定向与管道

6.2.1 输入重定向

1. 输入重定向自文件

用法：<文件

将标准输入重定向到一个磁盘文件，而不是从键盘输入。

【例 6-2】标准输入重定向的使用举例。

```
./myap < try.in
```

如果你在编写一个程序`btree.c`完成一个数据结构的作业，根据一棵二叉树的先序，中序，求后序。程序运行的时候需要输入先序序列，中序序列，然后打印出后序序列。由于程序逻辑较复杂，先序序列和后序序列分别为下列顺序时，程序运行不正确。

```
ABFGEHILMOPCDJKNQR
FGBHELIOPMACJDQNRK
```

这样，就需要反复地修改程序文件`btree.c`,编译后运行`./btree`,然后根据程序提示，从键盘输入上述的两个字符串数据。反复的调试需要反复的输入上述数据，这种重复性的劳动效率很低，而且容易出错。这样，就可以将上述两行文本编辑成文件`btree.tst1`，利用shell的输入重定向，运行`./btree < btree.tst1`。这种重定向功能在Windows中也可以使用。

2. Here Document

用法： <<定界符

这种方法从shell脚本中获取数据,直到再次遇到定界符为止。UNIX把这种输入重定向的方法叫做“Here document”。这种重定向方法在C-shell中也可以使用。

【例 6-3】简单的 Here document。

```
cat << TEXT
```

```
**********
* Hello! *
**********
TEXT
```

上述命令执行结果,在屏幕上显示：

```
**********
* Hello! *
**********
```

这里，<<符号后面是定界符，在shell输入中下一个TEXT之前的这段文本，算作cat命令的标准输入。

【例 6-4】在 Here document 中进行命令替换和变量替换。

```
cat << TOAST
Hello! Time: `date`
My Home Directory  is  $HOME
Bye!
TOAST
```

上述命令执行结果为：

```
Hello! Time: Sat Jul 27 14:47:56 BEIJING 2004
My Home Directory is /usr/jiang
Bye!
```

注意由shell所进行的变量替换和命令替换。在两定界符之间的内容,只用变量替换,命令替换,不作文件名生成。shell对由两个定界符所界定的内容作的加工处理同shell对双引号所括起内容的加工处理一样。关于变量替换，命令替换，双引号，在6.4和6.5节介绍。

当定界符自身被单引号括起来,或者前面增加反斜线时，就不再对Here document进行这些替换。

【例 6-5】在 Here document 中禁止命令替换和变量替换。

```
cat << \TOAST
Hello! Time: `date`
My Home Directory  is  $HOME
Bye!
TOAST

cat << 'TOAST'
Hello! Time: `date`
My Home Directory  is  $HOME
Bye!
TOAST
```

上述两种情况，输出结果都是：

```
Hello! Time: `date`
My Home Directory  is  $HOME
Bye!
```

6.2.2 输出重定向

1. 标准输出重定向

用法： >文件 >>文件
将标准输出重定向到一个磁盘文件。例如：

```
ls -l > file1
```

将命令ls标准输出定向到文件file1中，如果文件已经存在，就覆盖它。

```
ls -l >> file1
```

将命令ls标准输出定向到文件file1中，如果文件已经存在，就追加到文件尾部。

2. 标准错误输出重定向

用法： 2>文件
将标准错误输出重定向到文件。标准错误的重定向方法和csh不同。使用不同的shell应当注意选用shell能识别的重定向符。

【例 6-6】B-shell 的标准错误输出重定向举例。

(1) `cc  myap.c -o myap 2> myap.err`
将cc命令的stderr重定向到文件myap.err中。

(2) 设try是程序员设计的某个应用程序。

```
try > try.out 2>try.err
try 1> try.out 2>try.err
```

将try程序执行后的stdout或stderr分别定向到两个不同的文件中(其中的1和2,是文件描述符,分别是stdout和stderr，如果换成其它数字也可以对try的其它文件描述符输出改向)。

3. 指定文件描述符的输出重定向

用法： 文件描述符>&文件描述符

【例 6-7】B-shell 指定文件描述符的输出重定向。

```
myap >rpt 2>&1
```

或者：

```
myap 1>rpt 2>&1
```

shell从左至右进行重定向处理。上例中,先把文件描述符1(是stdout)与文件rpt结合,然后再把文件描述符2复制的和文件描述符1一样，和文件rpt结合。

这样stdout和stderr均存入文件rpt中。图6-1列举出改向前，1>rpt之后，2>&1之后，三步操作的文件描述符和相关文件的对应情况。

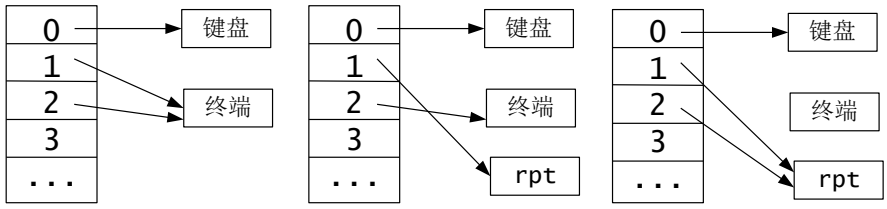

图 6-1 指定文件描述符的输出重定向处理方式㈠

如果重定向的顺序反过来：

```
myap 2>&1  > rpt
```

那么,重定向文件描述符2，和描述字1相同，假定文件描述符1是终端，就与终端结合。然后的操作，就是文件描述符1与文件rpt结合。图6-2列举出改向前，2>&1之后，>rpt之后，三步操作的文件描述符和相关文件的对应情况。这样，2>&1操作就没有能够将标准错误输出定向到文件rpt中。

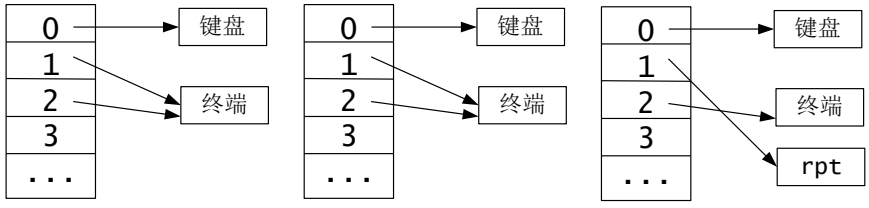

图 6-2 指定文件描述符的输出重定向处理方式㈡

下章的 7.3.2 节介绍的系统调用函数 dup2 会有助于理解 shell 的这些处理方式。

6.2.3 管道

【例 6-8】 B-shell 的管道处理举例。

(1) `ls  -l  |  grep '^d'`

前一命令的stdout作后一命令的stdin。

(2) `cc myap.c -o myap 2>&1 |  more`

前一命令的stdout+stderr作为下一命令的stdin。这里管道操作的优先级高，先完成cc的标准输出管道到下个命令的标准输入，然后，将文件描述符重定向的和文件描述符1一样，就完成了将标准输出和标准错误输出都管道到下个命令的目的。

6.3 变量

6.3.1 变量赋值和引用

shell支持有名字的变量，以提供更多的灵活性。shell变量只有一种类型，存储字符串。shell变量可以赋值，内容可以被修改，也可以被引用。

变量名的第一个字符必须为字母,其余字符可以是字母,数字,下划线。变量不需要事先定义，直接赋值就可以定义一个新变量，或者修改原变量的值。引用时,在变量名前加$符，代表变量的内容。

【例 6-9】shell 变量的定义和引用举例。

定义一个名为addr的变量，存放一个IP地址字符串。

```
$ addr=20.1.1.254   (最左侧的$为 sh 命令提示符)
$ echo $addr        (引用变量 addr,echo 执行前,sh 完成变量替换)
20.1.1.254
```

在等号两侧不允许有多余的空格，对惯于在运算符两侧加空格，书写类似a = 1;之类语句的C语言的程序员尤其应该注意。在shell中空格的作用很重要，多一个或者少一个空格，会引起系统的不同解释。上例中如果等号两侧有空格，系统就会以为要启动一个名叫addr的命令，它有两个命名行参数，分别是等号和20.1.1.254。

echo命令在shell脚本文件中经常被使用，用做输出命令，打印变量值，用户提示。类似一般的算法语言中的输出语句。它的功能就是将给它的命令行参数输出，echo有多个参数时，两个参数之间加一个空格，输出结束后再输出一个换行符。为了提高效率，许多shell在具体实现时，echo被做成了内部命令，实现与/bin/echo程序等价的功能。

引用时,在变量名前加$符，代表变量的内容。shell采用变量替换的方式工作。上例中，shell会把它的命令行echo $addr替换成echo 21.1.1.254然后执行这一结果命令。就跟手工键入echo 21.1.1.254命令一样。

赋值时,等号右侧的字符串中含有空格或者制表符，换行符时,要用引号将打算赋值的字符串括起来。

```
$ city="Beijing, China"
$ echo $city
Beijing, China
```

引用一个未定义的变量,变量值为空字符串。

```
$ echo Connected to $proto Network
Connected to Network
$ proto=TCP/IP
$ echo Connected to $proto Network
Connected to TCP/IP Network
```

在DOS中也有变量替换，设置变量用“set 变量名=字符串”命令，察看变量直接用无参数的set命令，DOS中的变量替换引用变量的方法是“%变量名%”,例如：

```
CD %TEMP%
```

6.3.2 read:读用户的输入

内部命令read，可以从标准输入上读入一行，并将这行的内容赋值给一个变量。这在shell脚本文件中非常有用，可以接受用户的输入信息。

【例 6-10】shell 读取用户的输入，并使用输入的信息。

```
$ read name
ccp.c
$ echo $name
ccp.c
$ ls -l $name
-rw-r--r--   1 jiang   usr         32394 May 27 10:10 ccp.c
$
```

【例 6-11】在脚本程序中获取用户输入，并根据用户输入修改程序的配置文件。

假设有个应用程序myap运行时需要从文件myap.cfg中读取配置参数。使用下面的脚本文件，在程序安装时，根据用户的输入修改配置文件myap.cfg中的内容。

```
$ cat myap.cfg
ID 3098
SERVER 192.168.0.251
TCP-PORT 3450
TIMEOUT 10
LOG-FILE /usr/adm/myap.log
$ chmod u+x config.ap; cat config.ap
#!/bin/sh
echo 'Input IP address of server computer: \c'
read addr
ed myap.cfg > /dev/null <<TOAST
/SERVER
.d
i
SERVER $addr
.
w
q
TOAST
$ ./config.ap
Input IP address of server computer: 202.112.67.213
$ cat myap.cfg
ID 3098
```

```
SERVER 202.112.67.213
TCP-PORT 3450
TIMEOUT 10
LOG-FILE /usr/adm/myap.log
$
```

这里使用echo给出一个提示，\c表示echo在显示完字符串之后不再换行，这是echo命令的功能。另外，echo还支持\r,\t,\n等其它转义符，反斜线后跟三个八进制数字可以描述任意ASCII值的字节,这些表达方法跟C语言的字符串常数的表达方法相近。

read命令将从键盘上输入一个IP地址字符串，放到addr变量中。

这里使用了行编辑程序ed来修改文件myapp.cfg，编辑过程的所有输出信息重定向到/dev/null中被丢弃不再在终端上显示。在两个TOAST之间夹着的Here document段落作为ed的标准输入。这也是行编辑程序ed可以发挥作用的地方。行编辑命令的这几个操作，/SERVER是检索含有正则表达式SERVER的行；.d删除当前行，圆点代表当前行，d为删除命令；i为插入命令，随后的输入内容作为插入内容，其中的$addr会被变量addr的值取代，插入工作直到遇到行首为圆点(.)的行，插入结束，回到ed的命令状态；w存盘；q退出命令执行。需要行编辑ed的其它命令，可以查阅相关手册。在shell脚本文件中经常会用到ed命令，并用<<重定向输入，来自动编辑修改一个文本文件的内容，这些文本文件通常是些配置文件。

6.3.3 环境变量和局部变量

所创建的shell变量,默认为局部变量。使用sh内部命令export可以将一个局部变量转换为环境变量。例如：exort proto

局部变量和环境变量是有区别的。在当前shell下启动的子进程只继承环境变量,不继承局部变量。所谓“继承”，就是指子进程有自己的一整套独立存储的环境变量，但是这些环境变量的初始状态，是从父进程那里原封不动抄写来的。从此以后，父子进程各保留一套，子进程中对全局变量的修改,不影响父进程中的同名变量的值，子进程继续创建它自己的子进程时，这些值生效。

【例 6-12】观察 shell 的局部变量和环境变量的不同。

```
$ chmod u+x stat.report; cat stat.report
    echo Connected to $proto Network
$ proto=AppleTalk
$ ./stat.report    (启动一个子进程sh)
    Connected to Networks
$ export proto
$ ./stat.report
    Connected to AppleTalk Networks
```

上边的chmod和cat命令写在一行内，shell允许把多个命令写在一行，中间用分号隔开。

在上例中,子进程仍为一个shell进程,去访问变量proto。如果子进程的程序是用C语言编译而成的程序时，C语言访问环境变量可以用库函数getenv()，在7.1.3节介绍了另外一种方法，但是getenv()用法最方便程序员使用。

【例 6-13】在 C 语言程序中访问环境变量。

下面的程序例子使用getenv函数获取名为proto的环境变量的值。

```
int main(void)
{
   char *envstr;
   envstr = getenv("proto");
   if (envstr)
      printf("Protocol is  %s\n", envstr);
   else
      printf("Protocol is ??\n");
}
```

不附带任何参数的内部命令set会列出当前所有变量及其值,包括环境变量和局部变量。使用内部命令unset可以删除一个变量。

下面的命令删除名为proto的变量。

```
unset proto
```

与环境变量相关的重要的命令是env，env是一个外部命令，程序文件/bin/env，列出所有环境变量及其值。

6.3.4 内置变量

在shell中有几个内置的变量，在shell脚本文件中可以直接使用这些内置变量，而且不允许对这些变量直接赋值,见表6-1。

表 6-1 B-shell 的内置变量

内置变量	含义
$?	最后一次执行的命令的返回码。
$$	shell进程自己的PID。
$!	shell进程最近启动的后台进程的PID。
$#	命令行参数的个数（不包括脚本文件的名字在内）。
$0	脚本文件本身的名字。
$1 $2, ……	第一个，第二个,...,命令行参数。
"$*"	"$1 $2 $3 $4 ..."，将所有命令行参数组织成一个整体，作为一个“单词”。
"$@"	"$1" "$2" "$3" ...，将多个命令行参数看作是多个“单词”。

表6-1最后几项，从$#到"$@"，是与shell脚本文件启动时附带的命令行参数相关的，也叫“位置变量”。在本章后面的6.9节，介绍了位置变量可以使用内部命令set和shift进行重新设置。另外，位置变量还用在shell的子程序中，作为传递的参数。

【例 6-14】位置变量$*与$@的区别。

```
$ cat arg.c
    main(int argc, char **argv)
    {
       int i;
       for (i = 0; i < argc; i++)
          printf("%d:[%s]\n", i, argv[i]);
    }
    $ cc arg.c -o arg
$ chmod u+x param; cat param
    #!/bin/sh
    echo $$
    echo $#
    echo "Usage: $0 arg1 arg2 ..."
    ./arg  "$@"
    ./arg  "$*"
$ ./param Copy Files to $HOME
    19752
    3
    Usage: ./param arg1 arg2 ...
    0:[./arg]
    1:[Copy]
    2:[Files]
    3:[to]
    4:[/usr/jiang]
    0:[./arg]
    1:[Copy Files to /usr/jiang]
```

6.3.5 shell 的标准变量

这是一些系统一登录成功后就由系统自动创建好的环境变量。

1. HOME

用户主目录的路径名。在C语言程序中可以用getenv("HOME")得到用户的主目录。

2. PATH

使用这个变量的值作为命令的查找路径。例:

PATH=/bin:/usr/bin:/etc

多个查找路径之间用冒号隔开。那么,在用户建入一个命令时，如myap1，没有指定路径名。shell首先判断myap1不是它的内部命令，然后，先找/bin/myap1，再找/usr/bin/myap1，然后/etc/myap1。与DOS不同的是,它不首先搜索当前目录下的myap1，如果在上述三个路径下找不到myap1，但当前目录下有myap1，也不去执行它。

```
PATH=./:/bin:/usr/bin:/etc     先搜索当前目录
PATH=/:bin/usr/bin:/etc:./     最后搜索当前目录
```

也可以使用变量替换：

```
PATH=./:$PATH
PATH=$PATH:./
```

3. PS1 和 PS2

这两个变量设置B-shell提示符，分别设置B-shell的主提示符和副提示符。PS1="C>"，那么，主提示符变为C>。

副提示符用在这样的情况，按照shell的语法解释，一个命令在一行时输不完需要几行输入一个命令时,第2行及其它行的提示符用副提示符。B-shell一般的PS1="$"，PS2=">"。例如：第一行有单引号，未碰到第二个单引号，shell理解为当前命令输入尚未结束，随后给出的提示符就是大于号，等shell理解输入结束之后，立即执行命令。命令执行完之后，提示符再次变为$号。

【例 6-15】B-shell 的主提示符和副提示符的用途。

```
$ msg='The echo command writes character strings
> to standard output. Strings are separated by spaces,
> a new-line character follows the last String.'
$ echo "$msg"
The echo command writes character strings
to standard output. Strings are separated by spaces,
a new-line character follows the last String.
$ cat <<TEXT
> ********
> Hello!
> ********
> TEXT
********
Hello!
********
$
```

4. TERM

终端类型名,许多全屏幕操作的软件(如vi),使用这一变量的值以确定当前使用的终端的类型,去搜索终端库，以确定需要光标移动或其它控制操作时，主机该送往终端的转义字符序列的格式。

5. LANG 或 LANGUAGE

语言。有的命令会根据语种不同，给出相应的语种的提示信息，如：中文或英文。

6.4 替换

shell对脚本文件中的命令进行的替换工作有，命令替换，变量替换和文件名生成。“替换”是shell提供灵活性的最重要手段。shell先替换用户输入或者脚本文件中的命令行，然后再执行命令。文件名生成和变量替换，在前面的已经提到过。

6.4.1 文件名生成

shell将文件名通配符展开成多个文件名的工作，叫“文件名生成(Filename generation)”。例如：命令ls *.c，shell会将*.c进行展开，替换为arg.c myap.c。展开后有多个文件时，按照字符串比较时从小到大的顺序排序。然后执行 ls arg.c myap.c。这样，执行命令ls *.c和手工键入命令ls arg.c myap.c的执行效果是一样的。

6.4.2 变量替换:

所谓变量替换就是将脚本文件中的$打头的单词，替换为变量值。下面的例子shell会用变量HOME的值替换$HOME,然后再执行find命令；用变量TERM的值替换$TERM,然后执行echo命令。

```
find $HOME –name "*.c" -print
echo Terminal type is $TERM
```

6.4.3 命令替换

用两个反撇号括起来一个命令，用命令执行的标准输出代替两个反撇号标志出的字段,这就是所谓的“命令替换”。反撇号是一个很少用到的符号，一般在键盘最左上角Esc按键下方。

【例 6-16】shell 命令替换的例子。

```
$ now=`date`    (以命令date 的stdout替换`date`)
    $ ./arg 'date' (实际执行./arg Sun Dec 4 14:54:38 BEIJING 2004)
    0:[./arg]
    1:[Sun]
    2.[Dec]
    3:[4]
    4:[14:54:38]
    5:[BEIJING]
    6:[2004]
$ frames=`expr 5 + 13`
```

```
$ echo $frames
18
$ count=10
$ count=`expr $count + 1`
$ echo $count
11
```

应当说expr是一个与shell无关的外部命令，用于完成表达式计算。这个命令在后边的6.7.2节还会详细介绍。shell自身连四则运算这样的基本功能都没有，但是它提供的命令替换机制却可以借助其它的命令来完成。

6.5 元字符

所谓“元字符”（metacharacter）是指在shell中有特殊含义的字符，shell对这些特殊字符进行特殊解释。初学shell的用户，不了解shell中的元字符和对元字符处理的特点，往往会非常困惑。UNIX从一开始的设计，就没打算设计得像傻瓜相机一样便于毫无专业知识的人使用。在前面使用命令时，多处强调必须注意命令格式中必需的空格，引号和转义符\，通过这一节的解释，应当了解为什么必需这些符号,并且能够正确的使用这些符号，有效地驾驭shell，以完成所期望的任务。

shell有许多元字符，表6-2是部分元字符的一个列表。

表 6-2 B-shell 的常用元字符

元字符	功能
\	转义符，取消后继字符的特殊作用。
空格,制表符	命令行参数的分隔符。
回车	执行键入的命令。
;	用于一行内输入多个命令。
* [] ?	文件名通配符。
$	引用shell变量。
` `	反撇号,用于命令替换。
> < | >> <<	重定向与管道。
&	后台运行。
( )	用于定义shell函数,以及定义命令表。
"	双引号，除$和`外，其它特殊字符的特殊含义被取消。
'	单引号,对所括起的所有字符,不做特殊解释,特殊字符的特殊含义被取消。

() > < | ; & 等除了它们自身的特殊含义外还同时起到单词分隔符的作用。尽管许多命令对命令行参数中第一个字符为减号的参数，理解成是命令执行功能的选项，但是，这仅仅是命令程序自身的行为，与shell无关。减号不是元字符。

6.5.1 空格、制表符和转义符

和类似C语言一样的普通算法语言不同，空格和制表符在shell中起着很重要的作用，shell使用它们做单词之间的分隔符。

【例 6-17】shell 中空格的作用。

下面的例子可以看出空格符在shell中的重要作用。

```
$ expr 33+44
33+44
$ expr 33 + 44
77
```

在普通的算法语言中，33+44和33 + 44没什么区别，但是，在shell中，它们的区别很大。第一种用法，expr可以获得一个命令行参数，是字符串33+44，加号两侧带空格的第二种方法，expr可以得到三个字符串的参数，分别是33，+，44。联想到我们自己做的简单程序arg.c，体谅一下程序expr的设计者对参数的处理，两种不同的书写方法，在程序中会有完全不同的感受。UNIX简洁的特点，使得expr程序不把一个独立的单词33+44理解成两个整数相加。同样，[$a=$b]和[$a = $b]也是完全不同的。

在shell脚本中，水平制表符，即按下Tab键产生的字符，和空格一样，起着“单词分隔符”的作用。空格作为单词分隔符使用时，连续的多个空格和一个空格的效果是一样的。

【例 6-18】shell 中连续多个空格的作用。

```
$ expr   33    + 44
77
$ echo UNIX System V
UNIX System V
$ echo  UNIX    System V
UNIX System V
```

后两个命令，echo都是得到三个命令行参数，echo把三个命令行参数顺序打印出来，每两个之间用空格分开，最后打印一个换行。

6.5.2 回车和分号

回车的作用是标志一个命令输入的结束。键入回车后，shell读取当前行，当前行的第一个单词是命令名，其余单词是该命令的参数，然后执行命令。UNIX允许把多个命令书写在一行，命令之间用分号分开。

【例 6-19】用分号串结两个命令。

```
$ date; who am i
Wed Jun  9 20:58:16 BEIJING 2004
jiang     pts/2      Jun 09 20:58
```

那么，先执行命令date，执行完毕之后，再执行who am i。分号是shell的元字符，而且，分号也有分隔符的功能，因此，分号前后有无空格均可。

6.5.3 文件名通配符

出现在命令行中的文件名通配符* [] ?会被shell展开成多个文件名。

6.5.4 美元符和反撇号

美元符$和配对使用的反撇号分别用于变量替换和命令替换。

6.5.5 重定向和管道

重定向和管道功能给系统的使用带来了很多灵活性。可以使用它们组合多个命令以构造出更多功能。UNIX 的许多命令设计的风格，甚至许多命令的输出格式，在设计时都考虑了便于重定向和管道处理。重定向符号和管道符号，都可以兼做命令分隔符。

【例 6-20】重定向和管道符的命令分隔符作用。

(1) `./myap < my.in | ./myap2 > myap2.out`

也可以写作

```
./myap<my.in|./myap2>myap2.out
```

(2) 重定向符在可能会产生歧义的地方，应当使用空格。

列出文件2的属性，并存入文件2.list，使用下面的命令：

```
ls -l 2>2.list
```

那么，shell会理解为执行ls -l并把标准错误输出重定向到文件2.list.应当使用

```
ls -l 2 >2.list
```

符号>的左边的空格是必不可少的。同样，大于号左边有单独一个数字构成单词时，大于号的左边一定要多用一个空格。类似的：

```
./myap 8>8.list
```

那么，shell会理解为将文件描述符8重定向到文件8.list。

B-shell除了对标准输入，标准输出和标准错误输出可以施行重定向之外，对除了0,1,2之外的其它文件描述符3~9也可以重定向。前面的8>对文件描述符8重定向，也可以使用8>&1使得文件描述符8重定向到文件描述符1的文件。除了几种常用的重定向功能外，shell还有其它的几种重定向功能，例如，<&,用法和>&类似，5<&0将文件描述符5作为输入文件，并且文件描述符5重定向的和文件描述符0一样指向同一文件，这样程序中的read(5,……),就会和read(0,……)有相同的效果，都从文件描述符0引用的文件中获取数据。shell重定向的这些灵活的功能，应当和所设计的应用程序配合起来工作，才能有更有意义。必要的时候，在设计应用程序时可以考虑利用shell的这些灵活功能。

【例 6-21】shell 对标准输入，标准输出和标准错误输出之外的其它文件描述符的重定向。

(1) 将文件描述符8重定向到标准输出。

```
$ cat myap.c
main(int argc, char **argv)
{
   if (write(8, "Hello!\n", 7) < 0)
      perror("write fd8");
}
$ cc myap.c -o myap
$ ./myap 8 >`tty`
write fd8: Bad file number
$ ./myap 8>`tty`
Hello!
$
```

上面的例子将文件描述符8重定向到当前的终端设备文件。在执行./myap前，shell打开当前终端设备文件，并且和文件描述8相关联; 否则，直接使用文件描述符8执行write系统调用，会返回错误，指出文件描述符错。

(2) 将文件描述符5通过重定向归并到标准输入。

```
$ cat myap.c
int main(void)
{
   char buf[128];
   int len;
   len = read(5, buf, sizeof(buf));
   if (len < 0)
      perror("read fd5");
   else
      write(1, buf, len);
}

$ cc myap.c -o myap
$ ./myap
read fd5: Bad file number
```

```
$ ./myap 5<&0
Hello!
Hello!
```

6.5.6 启动程序后台执行

&符作为后台启动程序的元字符。

shell中输入一个命令，在这个命令执行完之后，shell才给出新的提示符，允许输入新的命令。shell在具体实现时，启动这个命令的进程，然后shell进程等待命令进程运行结束。命令运行结束后，shell才给出新的shell提示符，允许输入新的命令。在命令的结尾处加&符，那么，shell启动了这个命令进程之后，不等待命令运行结束，就立刻给出新的提示符，可以输入下个命令。如果这个命令的执行需要很长的时间，就会在后台运行。随后输入的命令执行的同时，后台程序也在运行。

例如：在后台执行一个需要运行时间较长的排序操作，排序结果记入文件。

```
sort telnos > telnos1  &
```

Windows也是多任务系统，在DOS窗口中的内部命令start和UNIX的后台命令类似。如：

```
START PING -t 202.12.112.87
```

当前窗口可以继续输入其它的命令，ping命令在另一个新打开的窗口中执行。

6.5.7 圆括号

圆括号是shell的元字符，配对的圆括号之间的所有命令，会作为一个整体。

shell在实现圆括号的功能时，会首先创建子shell进程,在子shell进程中执行圆括号内的命令。

【例 6-22】shell 元字符圆括号的使用举例。

```
date;who>>users_on
(date;who)>>users_on
(ls -l;grep '^[^#]' data1)|wc -l
```

第一个命令，先执行date命令，标准输出输出到终端，执行结束后再执行who命令，who命令的输出重定向到文件users_on。

第二个命令，date和who两个命令相继执行，它们的标准输出都重定向到文件users_on。

第三个命令，ls列出当前目录下的文件，grep列出文件data1中所有行首字符不是#的行，两个命令相继执行，它们的标准输出都管道到下个命令wc的输入。

由于圆括号，分号，重定向符，管道符，都兼有单词分隔符的作用，所以上述命令，这些符号前后加不加空格都可以。

6.5.8 转义符

shell中的反斜线，用作转义符，取消紧跟其后的元字符的特殊作用。如果反斜线加在不是元字符的其它字符前面，那么，这个反斜线跟没有一样。或者说，反斜线使得它后面的任何字符不再有特殊意义。如果很难确定某个字符是否是元字符，在它的前面多加个反斜线，没有影响。但这仅限于shell的直接输入，对于引号或者反撇号之内的内容中的反斜线，处理与此不同。

【例 6-23】shell 元字符\的使用举例。

(1) `find / -size +100 \( -name core -o -name \*.tmp \) -exec rm -f {} \;`

查找系统中所有文件尺寸大于100块，名字是core或者有.tmp后缀的文件，删除它们。这里圆括号，星号和分号之前都有反斜线，以阻止shell对它们的特殊解释，find命令看不到这些反斜线，它能得到的是真正的圆括号，星号，分号。试着将上述命令的find换成我们自编的小程序./arg，可以站在find命令的角度体验程序所能接受到的命令行参数。

(2) 空格是shell的很重要的元字符，空格前面加反斜线，就取消了空格作为单词分隔符的特殊作用。

`ls -l > file\ list`

这样会创建一个文件名中包含空格的文件。

`vi 2\>\&1`

会创建一个名为2>&1的文件。当然，为了避免不必要的麻烦，最好不用这样的文件名，尽管这样的名字在UNIX中是合法的。

(3) `echo Unix\ \ System\ V`

这样，空格不再是单词分隔符，而是命令参数的有效组成部分，echo不是得到3个命令行参数，而是仅得到一个命令行参数，由14个字符组成的字符串。那么，echo把这14个字符打印出来，Unix和System之间就有两个空格。

(4) 反斜线加在其它元字符前面，元字符的特殊作用被取消。

echo *会打印出当前目录下的所有文件名，而echo *会打印一个星号。echo $HOME打印出当前用户的主目录，但是，echo \$HOME打印出的就是字符串$HOME。

(5) 非元字符前的反斜线的作用。

```
$ echo DOS Directory is C:\WINDOWS\DESKTOP
DOS Directory is C:WINDOWSDESKTOP
$ echo DOS Directory is C:\\WINDOWS\\DESKTOP
```

```
DOS Directory is C:\WINDOWS\DESKTOP
```

6.5.9 双引号和单引号

由于shell有这么多的元字符,如果需要的时候都用反斜线来转义，很不方便。

例如：打印10个星号，并询问(1+1)*2>3?那么需要的命令是：

```
echo \*\*\*\*\*\*\*\*\*\*
echo \(1+1\)\*2\>3\?
```

这样很麻烦，于是，shell提供了单引号，在单引号内的所有字符都不再解释为元字符。

使用单引号可以取消元字符作用。下面的命令和前面的命令运行结果相同。

```
echo '**********'
echo '(1+1)*2>3?'
```

配对的单引号括起来的内容不再对元字符做特殊的解释，这在有的时候不够方便。于是，引入了双引号。双引号和单引号的用法差不多，只是在配对的双引号括起的内容中还保留了元字符$和反撇号`，只允许变量替换和命令替换。

【例 6-24】单引号和双引号使用上的区别。

```
$ a=10
$ b=20
$ c=30
$ echo '($a+$b)*$c=?'
($a+$b)*$c=?
$ echo "($a+$b)*$c=?"
(10+20)*30=?
```

6.5.10 转义符与引号及反撇号

在这里需要特别的解释shell转义符\。在双引号或者单引号内，都没有保留反斜线的元字符地位，所以，echo '\A'和echo "\A"的输出都是\A，而echo \A的输出却是A。

如果在配对的单引号括起来的内容中有单引号字符，会怎么样？shell怎样判断遇到的第二个单引号不是第一个单引号的配对，而是正常的文本？为了解决这个问题，shell仍然允许在单引号括起的内容中使用\'代替单引号自身，允许\\代表反斜线自身。概括起来说，就是在配对的单引号括起来的内容中，只允许\'和\\两个转义序列。其它情况下的反斜线保持原文不变。类似地，在配对的双引号括起来的内容中，只允许\"，\$，\'，\\四个转义序列。

【例 6-25】单引号或双引号括起的内容中容许的转义。

```
$ echo 'Don\'t remove Peter\'s DOS dir "C:\PETER"!'
```

```
    Don't remove Peter's DOS dir "C:\PETER"!
$ echo "`who am i | awk '{print \$1}'`'s \$HOME is \"$HOME\""
    jiang's $HOME is "/usr/jiang"
$ echo "\`\`pipeline'' is commands separated by |"
    ``pipeline'' is commands separated by |
```

应当注意反斜线字符在引号内和引号外的这种不同表现。echo '\A'和echo "\A"的输出都是\A，但是，echo \A的输出却是A。echo "\$"和echo \$都输出符号$。

同单引号和双引号问题类似的还有反撇号，在配对的反撇号内括起的内容中如果有反撇号，那么就用\`,同样，反撇号内允许\\代表\自己。也就是说，反撇号内仅允许\`和\\转义序列，其它情况下，反斜线代表的是自己。

【例 6-26】反撇号内的转义处理。

下面的一个实用例子是编写一段脚本程序，给出程序名字，终止系统中正在运行的进程。注意这段程序中，转义符\的用法。

终止一个正在运行的程序一般的方法，首先使用ps命令列出当前的活动进程，然后，根据ps提供的进程PID，用kill命令终止这一进程。ps命令和kill命令，在下章的7.1.8节和7.2.2节还会详细介绍。下面的例子终止程序myap的进程。

```
$ ps -e | grep myap
31650  pts/2  0:00 myap
$ kill 31650
$ ps -e | grep myap
$
```

ps -e命令列出的第一列是进程的PID号，最后一列是启动这一程序的程序名。为了挑选出期望终止的程序的PID，用awk命令。挑选出myap单词在行尾，并且前面有空格和表示时间的*m:ss*格式的串，用正则表达式便是[0-9]:[0-9][0-9] myap$，必须将类似这样的正则表达式字符串传递给awk，直接使用awk命令便是：

```
ps -e | awk '/[0-9]:[0-9][0-9] myap$/{printf("%d ",$1)}'
```

为了编写可以灵活指定程序名字的脚本程序，这里不能直接用myap，而是使用脚本程序执行时得到的命令行参数$1，因此，上面的awk就不能再使用单引号，必须使用双引号以允许变量替换。原先的$和双引号前面一律加反斜线，就变成了

```
ps -e | awk "/[0-9]:[0-9][0-9] $1\$/{printf(\"%d \",\$1)}"
```

打印一个进程PID号不是最终目的，而是希望这个输出做kill的参数，应当使用“命令替换”功能。在反撇号内的文字，只有反撇号自身和反斜线是需要特殊对待的。将上述的命令，做为成对反撇号括起来的文字，必须对反斜线进行处理，于是变成了

```
kill `ps -e | awk "/[0-9]:[0-9][0-9] $1\\$/{printf(\\"%d \\",\\$1)}"`
```

如果反方向理解这个命令，首先将反撇号内的文字作为命令来执行，考虑\\会被换成\，反撇号内的文字实际内容是ps -e | awk "/[0-9]:[0-9][0-9] $1\$/{printf(\"%d \",\$1)}"。设脚本文件执行时提供的命令行参数$1是myap，根据双引号的处理特点，awk命令真正能得到的命令行参数只有一个，就是字符串

/[0-9]:[0-9][0-9] myap$/{printf("%d ",$1)}。awk会把第一个$理解为正则表达式中的行尾符，把第二个$1理解成打印第1列内容。

下面是脚本文件k的执行的情况。这里的这个程序还很不完善，在本章后面6.7.3节介绍for循环结构和6.9.3节介绍shift命令时，会给出改进。

```
$ cat k
PIDs=`ps -e | awk "/[0-9]:[0-9][0-9] $1\\$/{printf(\\"%d \\",\\$1)}"`
echo "kill $PIDs"
kill $PIDs
$ chmod u+x k
$ ps -e | myap
27248  pts/2  0:00 myap
31714  pts/2  0:00 myap
36926  pts/2  0:00 myap
$ ./k myap
kill 27248 31714 36926
$ ps -e | grep myap
$
```

转义符\的这些使用方法，看起来很蹩脚，但是，shell程序就是按照这样的规则进行处理。无论编写shell的脚本程序文件，或者交互式输入命令，都有这样的问题。对于这个问题，也没有更简洁而且完美的替代方案。如果不能理解shell程序处理这些问题的基本思路和做法，就不能准确的驾驭它按照自己的意图完成期望的任务。初学shell的操作员，应当逐渐掌握shell的这些规律，了解shell处理问题的风格。

6.6 条件判断

shell作为一种编程语言，程序的分支结构和循环处理是必需的要素。条件判定，使得程序的流程根据不同情况进行不同的处理。

6.6.1 条件

shell是一种命令语言，它设计得非常简练。shell变量只有字符串一种变量类型，就连加减乘除基本的算术运算，比较两个数大小这样的逻辑判断功能，都不提供。但是，它提供了命令替换等机制，使用这些机制，利用外部的命令，可以完成所需要的功能。这是一种“策略和机制分离”的方法。shell仅仅提供一种机制，但不提供解决问题的策略，所有策略问题“外包”给其它的命令，或者用户自己编写的应用程序。这种开放性设计，使系统有极大的灵活性，而且，大大简化了shell自身的设计。尽管这样会带来一些效率上的问题，但是，对于shell这样面向命令处理的脚本语言来说，效率上的损失可以忍受。为了提高效率，有些shell把脚本程序中某些常用的命令，如expr，test，true，false等，改进成内部命令，但这种改进是透明的，shell仍不失它一贯的风格。C语言本身也是这样一种思路，C语言连基本的输入输出语句都没有，它把这样的功能“外包”给了诸如

printf，scanf，fgets这样的C语言基本要素之外的库函数。如果你对这些函数不满意，完全可以编制自己的输入输出函数，它们和printf之类库函数在C语言中具有完全相同的地位。这种，“策略”和“机制”相分离的方法，值得我们在设计其它软件系统时参考。

shell条件判定也一样，只提供了一种机制。条件判断的唯一依据是判定一条命令是否执行成功。判断方法是根据命令执行的返回码，返回0，就算是条件成立，返回非0的任意值，都算是条件不成立。

所谓的“命令执行的返回码”是由命令自身的行为决定的。

2.13节介绍的cmp命令比较两个文件是否相同。两文件相同时，cmp命令返回码为0；否则，不为0。cmp命令的返回码可以用于shell的条件判断。

【例 6-27】在 C 语言程序中不同的分支结束程序时给出不同的返回码。

下边是源程序myap.c的一个框架。

```
1 int main(int argc, char *argv[])
2 {
3    ……
4   if (……) {
5        ……
6       exit(4);
7   }
8   ……
9   if (……) {
10       ……
11       exit(19);
12   }
13   ……
14   if (……) {
15       ……
16       exit(0);
17   }
18   ……
19   return 0;
20 }
```

其中，库函数exit终止程序运行。exit的参数值，就是程序运行结束后产生的“返回码”。这一源程序文件，编辑链接之后，产生可执行程序myap。程序myap在执行时，如果运行到第6行的分支而运行结束，那么，程序的返回码就是4，同样的，如果程序运行到第11行的分支而运行结束，那么，程序的返回码就是19，第16行，返回码0。主函数main的函数返回值也是返回码，程序因执行19行而运行结束，返回码是0。如果没有第19行，main函数执行到函数尾而结束，那么，返回码就会是一个随机值。返回码的取值在0~255之间。UNIX建议把程序执行完后的返回码理解成“错误码”，0表示成功，其它值，就是错误代码。

在shell中，有一个内置变量$?，它是上个命令执行结束后的返回码的值。设当前目录下有目录xyz并且不存在一个名为abc的文件或目录。

【例 6-28】ls 命令的返回码，以及 shell 内置变量$?。

```
$ ls -d xyz
xyz
$ echo $?
0
$ ls -d abc
abc: not found
$ echo $?
2
```

用管道线连接在一起的若干命令，shell仅采用最后一个命令执行的返回码。

6.6.2 最简单的条件判断

最简单的条件判断仅含有一个分支，条件成立或者不成立时执行相应的命令。具体做法是用&&或||连结两个命令。

命令1 && 命令2

若命令1执行成功(返回码为0)则执行命令2,否则不执行命令2。

命令1 || 命令2

若命令1执行失败(返回码不为0)则执行命令2,否则不执行命令2。

【例 6-29】利用 ls 命令的返回码进行条件判断。

```
$ ls -d xyz && echo FOUND
xyz
FOUND
$ ls -d xyz > /dev/null && echo FOUND
FOUND
$ ls -d abc || echo No dir \'abc\'
ls: abc not found
No dir 'abc'
$ ls -d abc 2> /dev/null > /dev/null || echo No dir \'abc\'
No dir 'abc'
```

6.6.3 命令 true 与 false

命令true和false不是shell中的关键字。许多UNIX系统的确存在/bin/true和/bin/false两个命令文件。true命令的返回码总为0，除此之外不做任何操作。可以设想true.c一定是个很简单的C语言程序：

```
int main(void) { return 0; }
```

false命令的返回码总不为0。

shell中有个内部命令，名字为冒号(:)，冒号命令和true命令有相同的效果。6.7.1节例6-39中有冒号命令的使用举例。

6.6.4 命令 test 与[

严格的说，test命令是一个独立于shell之外的命令，就如同printf是独立于C语言基本要素之外的库函数一样。shell表示条件的唯一准则是判断命令执行的返回码是否为0。至于命令自己在什么条件下给出什么样的返回码，是命令自己的行为，与shell无关。程序员完全可以根据需要，编写自己的命令程序，通过返回码进行shell的流程控制。UNIX系统自带的命令test，可以提供一些常用的条件判断。除此之外，还有一个与test功能等价的命令[，和test命令不同的是，命令[要求其最后一个命令行参数必须为右中括号。早期的UNIX中的确依靠两个程序文件/bin/test和/bin/[。注意，在这里的独立的左中括号是一个程序文件的名字，和普通文件名一样，没什么特殊性。在书写和阅读shell脚本时,必须把方括号理解成一个命令,而不要同其它算法语言(如:C语言)中那样,简单地将方括号理解成一个词法符号。既然两个程序的功能一样，可以猜测文件/bin/test和文件/bin/[可能是硬连接关系或者符号连接关系,在LINUX中就是符号连接。有些shell为了效率上的考虑，将test和[实现为内部命令，但是，无论是内部命令还是外部命令，使用方法是一样的。

例如: 下面两个命令的执行结果完全相同。注意，左中括号后面，和右中括号前面，空格是必不可少的。

```
test -r /etc/motd
[ -r  /etc/motd ]
```

test命令主要提供了以下的判断功能。

1. 文件特性检测

-f 普通文件
-d 目录文件
-s size>0
-r 可读
-w 可写
-x 可执行

例如：

```
test -r /etc/motd  && echo readable
[ -r /etc/motd ] && echo readable
```

2. 字符串比较

-z *str1*　　*str1*串长度等于0　(zero)
-n *str1*　　*str1*串长度不等于0　(non-zero)
str1 = *str2*　　*str1*与*str2*串相等
str1 != *str2*　　*str1*串与*str2*串不等
需要注意的是，等号和不等号两侧的空格是必不可少的。
例如： `test $# = 0 && echo "No argument"`

【例 6-30】判断 name 变量的值是否为空字符串。

```
[ -n "$name" ] || echo "empty string."
```

这里括起$name的引号是必需的，因为，如果没有$name两侧的引号而且shell变量name真的是空字符串，那么，[-n $name]会被替换成[-n]，这样，对名字是[的命令来说，只有两个命令行参数，分别是-n和右中括号，从而不符合命令[的要求。加上引号之后，[命令就会得到三个命令行参数，其中第二个命令行参数是空字符串。类似的，判断两个字符串相等或不等时，如果两个字符串之一有可能是空字符串，那么，也应当使用双引号将必要的内容括起来。

3. 整数的比较

-eq	equal	＝
-gt	greater than	＞
-ge	greater or equal	≥
-ne	not equal	≠
-lt	less than	＜
-le	less or equal	≤

例如： `test `ls | wc -l` -ge 1000 && echo "Too many files"`

4. 逻辑运算

!　NOT（非）
-o　OR （或）
-a　AND （与）
例如： 判断$cmd是一个具有可执行属性的普通文件。需要判断它不是目录文件，并且又具有可执行属性。

```
[ ! -d $cmd -a -x $cmd ]
```

【例 6-31】条件判断命令中空格符的重要作用。

下面的例子，说明脚本文件中必需的空格不可省略。这也是shell中使用[……]结构时经常出现的错误。

设有一脚本文件t1，当给它的命令行参数正好两个时，打印出一条信息。但是，实际执行的结果却是，给定了9个命令行参数，而且，这条信息照样打印出来。

```
$ cat t1
echo "count=$#"
[ $#=2 ] && echo There are 2 files.
$ ./t1 *.c
count=9
There are 2 files.
```

正确的判断命令该是[$# = 2]

必需的空格有四个,每个空格都不可省略。命令[要求等号必须单独作为一个命令行参数，所以，等号两侧必须有空格。以保证等号作为一个独立的命令行参数，传递给命令[。如果缺少了[号后面的空格,系统实际执行命令名为[9的命令，系统会给出错误,说找不到一个名字叫[9的命令。

如果缺少了]前面的空格,命令/bin/[执行时，因找不到独立的命令行参数]而失败。

6.6.5 { }与( )

当使用&&或||时，需要在条件分支中完成多个动作，执行若干个命令，就需要使用类似复合语句的构造，在shell中使用花括号。书写规则为

{ *list*;}

左花括号后面必须有一个空格，右花括号前面必须有分号。*list*是由一个或者多个命令构成的命令表。

【例 6-32】在 B-shell 中使用花括号实现复合语句的构造。

下面的例子使用了花括号，在条件满足的时候执行多个命令。

```
DIR=/usr/include/sys/netinet
pwd
[ -d $DIR ] && {
   cd $DIR
   echo "Current Directory is `pwd`"
   echo "`ls -l *.h | wc -l` files (*.h)"
}
pwd
```

在上例中，左花括号后,换行代替了左花括号后的空格,右花括号前的换行代替了规则中的分号(;)。书写规则的这些要求，保障了左中括号和右中括号都作为一个独立命令的首个单词，也就是命令名。可以设想shell是把左中括号和右中括号作为一个内部命令来处理的，所以花括号从来都不是元字符，而且，可以猜测，正是由于UNIX和C语言的关

系，shell才会选用花括号以内部命令的方式完成这样的功能，要是参照Pascal语言，该选用begin代替左花括号，用end代替右花括号了。

前面脚本程序的执行结果如下:

```
/usr/jiang
Current Directory is /usr/include/sys/netinet
27 files (*.h)
/usr/include/sys/netinet
```

圆括号也有将多个命令合成一个整体的功能。圆括号是shell的元字符，所以书写格式上不必要象花括号那样让圆括号一定要单独作为一个独立命令的行首单词。书写规则为

(*list*)

左括号后不需要空格，右括号前面也不需要有换行。圆括号和花括号不仅在书写格式上不同，更主要的不同在于，花括号括起的一组命令是在shell进程中执行，但是，圆括号括起的一组命令，却是在子shell中执行。shell会首先创建子shell进程，然后，在这一子shell中执行命令，圆括号内的命令执行完毕之后，子shell就会终止，返回到shell。同样执行效果的前提下，用{}会比()执行效率更高些。

【例 6-33】{}与()的不同之处。

将上例中的{}改成()。那么，执行结果会有所不同（如下所示），脚本程序最后一行的输出不同。

```
/usr/jiang
Current Directory is /usr/include/sys/netinet
27 files (*.h)
/usr/jiang
```

在UNIX中，当前目录是进程自身的属性，cd命令只可能是内部命令，用于改变shell进程自身的当前目录。上述的例子中，如果将{}改为()，那么会启动子shell进程。子shell进程的当前目录是从父进程那里“继承”来的，这样，父子进程的当前目录都是/usr/jiang。子shell中执行的cd命令会改变子shell进程的当前目录，不会影响父进程。子shell进程终止后，父进程的当前目录仍然是/usr/jiang。

但是.profile 中的命令却是在当前 shell 进程中执行了。shell 提供了点命令(.)，在执行脚本文件的命令之前增加圆点和空格，那么，脚本文件的命令就在当前 shell 中执行，而不是启动一个新的子 shell 进程来解释脚本文件中的命令。这里，点命令是 shell 的一个内部命令。

【例 6-34】点命令，以及 shell 与子 shell 的区别。

下面的例子中，如果期望每次键入命令cdn而不用键入这个长串的命令就进入期望的子目录，也会失败。因为，脚本文件是系统单独启动的另个子shell进程执行的，而不是和我们交互式对话的这个shell。使用点命令可以用当前shell解释脚本文件中的命令。

脚本文件中的shell内置变量$$是shell进程的进程号。

```
$ pwd
/usr/jiang
$ cat cdn
echo $$
cd /usr/include/sys/netinet
pwd
$ ./cdn
4650
/usr/include/sys/netinet
$ echo $$
6626
$ pwd
/usr/jiang
$ . ./cdn
6626
/usr/include/sys/netinet
$ pwd
/usr/include/sys/netinet
$
```

【例 6-35】使用{}时，多行合并为一行书写的例子。

依照{}书写时的语法{ *list*;},多行合并为一行书写时不要漏掉所必需的空格和分号。

```
[ -f core ] && {
   echo "Remove core file"
   rm -f  core
}
```

写成一行应当为

```
[ -f core ] && { echo ”Remove core”;rm -f core;}
```

在交互式输入shell命令时，经常使用这种一行内输入所有命令的方式。注意左花括号后的空格和右花括号前面的分号都必不可少。

6.6.6 条件结构 if

使用&&和||的最简单方法，只可以有一个分支，条件结构if可以提供多个分支。条件if的语法是:

```
if  condition
    then  list
elif  condition
    then  list
else
    list
fi
```

这里的*list*可以是多个命令构成的命令表。*condition*仍然是一个命令，根据返回码判定为条件满足或者不满足。then左边的空格是为了书写格式的美观而增加的缩进排列，从语法角度，这些空格是可有可无的。实现条件if的关键字是if，then，elif，else，fi。从上面的格式要求中，可以很容易的分析出，这些所谓的关键字，都表现为一个独立命令的首个单词，作为命令名。可见，shell是把它们处理成内部命令，然后再赋以shell的特殊解释，用来进行流程控制的。这是shell处理这样问题的基本方法，循环结构控制也是使用类似的手法。在多个命令构成*list*时，没有必要在由多个命令构成的*list*的开始和结尾用花括号或者圆括号括起来，因为，在then和elif之间夹着的多个命令算作一个分支要执行的程序块，或者叫分程序。在else和fi之间夹着的多个命令算作另个分支要执行的程序块。fi是将if反过来写的形式。

【例 6-36】if 结构的使用举例，条件满足和不满足分别执行不同的命令。

将系统中现有文件errfile合并到文件errlog的尾部，合并时加入合并的日期。

```
$ cat errmonitor
LOGFILE=./errlog
date>>$LOGFILE
if test -r errfile
then
   cat errfile>>$LOGFILE
   rm errfile
else
   echo "No error">>$LOGFILE
fi
```

注意，书写这一文件时，then行可以和它下面的cat行合并成一行：

```
then cat errfile>>$LOGFILE
```

但是，if行不可以和then行合并成一行：

```
if test -r errfile then
```

如果这样，then没有机会作为一个新命令的首个单词，而是成为了test命令的第三个参数。命令test -r errfile then 的执行结果，将影响if的条件认可。then成为了test命令的第三个命令行参数，脚本文件缺了then，从而语法错误。这与普通的高级语言(如:Pascal语言)不同。如果将两行合并成上述形式的一行，那么，将不符合if结构的语法。在sh中，将应当写成两行的内容并成一行的方法是加分号，在这里分号与换行有相同的作用。例如：

```
if test -r errfile; then
```

是可以的。按照元字符分号(;)的作用，分号后是另一个命令的开始，保证了then作为一个独立命令的行首单词，从而使得shell可以将它解释为一个内部命令，赋予它特别的含义，从而实现流程控制的功能。

6.6.7 case 结构

case结构是基于模式匹配基础上的多条件分支结构，在很多情况下比使用if结构更简练。语法规则如下，其中esac是case四个字母的反序。case和esac是关键字，同样的，shell是通过把它们解释为内部命令的方式实现流程控制。

```
case  word  in
    pattern1)  pat1_list;;
    pattern2)  pat2_list;;
esac
```

(1) 模式描述时，使用shell的文件名匹配规则，这样使用起来更方便。

(2) ;;是一个整体,不可分隔,不能在两分号间加空格,也不能用两个连续的空行代替它(这一点如同&&和||)。在右圆括号和;;之间可以夹着多个命令定义的一个程序块，这个程序块可以有多个命令，也没必要用花括号或者圆括号括起来。

(3) 可以使用竖线罗列出多个模式。

(4) 当*word*可以与多个模式匹配时，只执行它所遇到的第一个命令表。

【例 6-37】case 结构的使用举例。

```
case "$1" in
START|start)
    .
    . (一段程序)
    .
    ;;
STOP|stop)
    .
    . (一段程序)
    .
    ;;
*)
    echo "Usage: $0 [start|stop]"
    ;;
esac
```

这种结构的代码经常出现在一些脚本程序中，脚本程序根据命令行参数的不同，完成不同的动作序列，对于不认识的参数在最后列出命令该怎样使用的提示。本例中case句中$1最好加上双引号，如果在引用这个脚本文件时没有携带任何参数，那么$1就会是空字符串，这种情况下省略了双引号就会导致case行语法错误。

6.7 循环结构

6.7.1 while 结构

循环结构while的语法是：

```
while condition
    do list
done
```

其中，while，do，done是关键字，必须以独立命令行的首个单词的身份出现，以确保shell会有机会把它们处理成内部命令，达到流程控制的目的。do和done之间的一段程序算作循环体。因此，也不需要在有多个命令时加花括号或者圆括号。while结构是在条件满足的前提下，循环执行do和done框起来的循环体内的命令。

【例 6-38】shell 脚本程序使用 while 结构的例子。

每隔10秒钟检查文件lockfile是否可读，并打印出这个文件的属性，直到这个文件被其它的任务删除后循环才退出。

```
$ cat waitlock
while test -r lockfile
do
   ls -l lockfile
   sleep 10
done
```

这里的sleep是shell的一个内部命令，后面跟的参数是秒数。shell会“睡眠”规定的秒数，暂停执行，然后再继续执行。循环体中加入的sleep命令，避免循环体被过渡频繁的“疯狂”执行。比如：1秒钟内循环体被执行几百次或者上千次，会严重浪费处理器的处理能力，多任务系统中的这种“忙等待”应当尽量避免。

书写这一脚本程序时,行与行合并时应注意的问题见上一节。下面的写法是错误的：

```
while test -r lockfile do
   ls -l lockfile
   sleep 10
done
```

下面的写法是正确的：

```
while [ -r lockfile ];do ls -l lockfile; sleep 5;done
```

【例 6-39】交互式中使用 while 结构。

在交互式shell中，也经常会使用循环控制的命令。使用ftp命令从远程计算机传输文件到本地文件mydata，如果线路速度不是很快，用下面的命令监视文件mydata的增长速度。

```
while true;do ls -l mydata;sleep 10;done
```

在前面的6.6.3节中介绍过shell的内部命令冒号命令(:)，冒号命令执行起来和true命令有相同的效果，所以上述命令也可以改写为下面的形式。注意，这里冒号是个命令名，不是什么特殊的元字符，所以，while后面的空格不可缺少。

```
while :;do ls -l mydata;sleep 10;done
```

6.7.2 expr: 计算表达式的值

命令expr用来求表达式的值。和test命令一样，从地位上来说，它是个独立于shell之外的外部命令，尽管这个命令经常被用到。

在B-shell编程中，B-shell本身没有提供数学运算和字符串运算的能力,所有这些运算都是借助于命令expr完成的。其它的一些shell，如C-shell和K-shell,为了效率上的考虑在shell内部支持数学运算。expr本来是外部命令，通过将它改造成内部命令的方法也可以提高效率。

1. 算数运算和关系运算

expr支持算术运算加减乘除，取余数，以及数值比较的关系运算。要求操作数是包含数字0到9的字符串，前面还可以带负号。expr将字符串转化为整数后运算，对数字执行的算术运算，显示之前再转换回字符串。另外，作为运算符的符号，必须作为单独的一个命令行参数，因此，运算符两侧的空格是必不可少的。考虑到shell的元字符，应该转义的地方必须加反斜线转义，以确保expr命令会得到它所期望的符号，而不是被shell作特殊解释。表6-3是一些算符和算符实现的功能。

表 6-3 expr 的运算符

算符	运算
*	乘法运算
/	除法运算
%	取余数运算
+	加法运算
-	减法运算
<	小于
<=	小于或等于
=	等于
!=	不等于
>=	大于或等于
>	大于

运算优先级，和 C语言一样。乘除法优先级最高，其次加减法，然后是关系运算。关系运算的结果是expr打印1或者0。也可以使用括号。

【例 6-40】shell 中使用 expr 命令时不可漏掉 shell 必需的转义符。

假设a,b,c是三个shell变量。求a*(b+c)，正确的写法为：

```
expr $a \* \( $b + $c \)
```

判断x是否大于20。正确的写法为：

```
[ `expr $x \> 20` = 1 ] && echo OK
```

使用时注意转义符和必须有的空格。

【例 6-41】 每秒一次倒计数到 0。

```
$ cat count
    count=10
    [ $# = 0 ] || count=$1
    while [ $count -gt 0 ]
    do
       echo "$count \c"
       count=`expr $count - 1`
       sleep 1
    done
    echo 0
    $ ./count 15
    15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    $
```

2. 字符串运算

用法：expr *string* ： *pattern*

这是一种字符串匹配运算。用正则表达式*pattern*去匹配字符串*string*，从最左字符开始,尽量匹配，看能匹配多长。最终打印出匹配的长度值，不匹配时打印0。这里的运算符冒号，必须单独作为一个命令行参数传递给expr命令。

【例 6-42】获取正则表达式与字符串的匹配长度。

```
$ expr 123 : "[0-9]*"
3
$ expr A123 : "[0-9]*"
0
$ expr 123A : "[0-9]*"
3
$ expr 123A : "[0-9]*$"
0
```

在正则表达式中有部分文字用\(和\)括起来,这两个符号在匹配时不起作用，那么，字符串能与正则表达式匹配时,打印括号内能匹配的部分；否则打印空字符串。这种正则表达式替换和3.4.12节的vi的替换命令类似。

【例 6-43】抽取正则表达式可匹配的字符串中的字符串片段。

下面的例子中，首先获取当前终端的设备文件名编号，后面的操作获取当前目录的路径名分量最后一个目录名。使用了命令替换的嵌套，而且非常夸张地使用了转义符。应当掌握shell对转义符，以及具体的命令获得了反斜线后正则表达式的处理。只有把握了shell的这些特点，在编写shell程序时，才不会遇到一些让人恼火的麻烦。

```
$ tty
/dev/tty6
$ expr `tty` : .\*  返回字符串长度
9
$ expr `tty` : '/dev/tty\(.*\)'  获取终端文件名字中的编号
6
$ termno=`expr \`tty\` : '/dev/tty\\(.*\\)'` 注意反撇号和转义符的关系
$ echo $termno
6
$ expr `tty` : /dev/tty\\\(.\*\\\)  只使用转义符的格式
6
$ termno=`expr \`tty\` : /dev/tty\\\\\\(.\\*\\\\\\)`
$ echo $termno
6
$ pwd
/usr/include/arpa
$ expr `pwd` : '.*/\([^/]*\)$'  截取路径名的最后一个分量
arpa
```

在上述的例子中，按照反撇号和转义符的关系(参见6.5.10节),那么，

```
`expr \`tty\` : /dev/tty\\\\\\(.\\*\\\\\\)`
```

就是用下列命令的执行结果进行命令替换。

```
expr `tty` : /dev/tty\\\(.\*\\\)
```

进一步，按照shell的元字符处理方式，expr的第一个参数是tty命令的执行结果，第二个参数是由冒号独立组成的字符串，expr实际可以得到的第三个参数是字符串/dev/\(.*\)，由expr命令内部自行解释第三个参数中的正则表达式。

6.7.3 for 结构

for结构的循环，循环体被执行多次。要求给出一个由多个单词构成的表格。每次循环，循环控制变量取值是表格中的一个单词。语法为：

```
for  name  in word1 word2 ...
    do list
done
```

其中，*name*是循环控制变量，在循环体内用*$name*引用变量的值。

for循环的另种格式是：

```
for name
    do list
done
```

这种格式没有指定循环控制变量的取值表，那么，系统就会用shell的位置变量中的$1, $2,...来作为循环控制变量的取值表。相当于:

```
for name in $1 $2 ...
    do list
done
```

【例 6-44】使用 for 循环的例子。

这是SCO UNIX中一段开机时系统自动执行的脚本程序。检索/etc/rc.d目录下所有的直属或一级子目录中的所有可执行文件，执行它们。根据shell文件名通配符的展开规则，文件名按照自小到大的顺序排列，所以，/etc/rc.d/0目录下的命令会比/etc/rc.d/1目录下的命令先执行。同一个目录中的文件，文件名小的，先被执行。

```
if [ -d /etc/rc.d ]
then
   for cmd in  /etc/rc.d/*/* /etc/rc.d/*
   do
      [ ! -d $cmd -a -x $cmd ] && $cmd
   done
fi
```

【例 6-45】将所有的命令行参数逆序显示出来。

脚本程序中使用for循环。

```
$ cat rev1
list=""
for arg
do
   list="$arg $list"
done
echo "$list"
$ ./rev aa bb cc
cc bb aa
```

【例 6-46】给出一组程序名，终止这些程序文件启动的所有进程。

获取程序启动的进程PID的方法，在前面与“元字符”有关的6.5.10节中介绍过。下面的脚本程序中使用了for循环结构。

```
$ cat k
for name
do
   echo "$name: \c"
   PID=`ps -e | awk "/[0-9]:[0-9][0-9] $name\\$/{printf(\\"%d\\",\\$1)}"`
   if [ -n "$PID" ]
   then
      echo kill $PID
      kill $PID
   else
      echo No process
   fi
done
$ k myap findkey sortdat
myap: kill 20608 27336 28072 29720
findkey: kill 36994 37948
sortdat: No process
```

6.7.4 break 与 continue

shell的内部命令break和continue用在循环结构for和while中使用，与C语言中的beak和continue流程控制功能类似。

【例 6-47】break 使用的例子。

将命令行参数逆序输出，脚本程序中使用了跳出循环的内部命令break。

```
$ cat rev2
count=$#
cmd=echo
while true
do
    cmd="$cmd \$$count"
    count=`expr $count - 1`
    [ $count -eq 0 ] && break
done
eval $cmd
$ ./rev2 aa bb cc
cc bb aa
```

针对上述脚本文件执行的例子，命令行有三个参数，这样，count的初值是3，在循环体执行结束的时候，cmd的内容是echo $3 $2 $1。如果脚本文件的最后一行没有eval，那么，输出结果会是$3 $2 $1。shell在将$cmd进行了变量替换之后，变成echo $3 $2 $1，但是，shell不会因为替换后的结果命令中含有$符号，而再继续对替换后的结果命令进一步递归式替换。这样，输出结果就是$3 $2 $1。

shell的内部命令eval，会将eval的实参作为shell的输入读入，再经过一轮变量替换，文件名生成，命令替换后，执行所得的命令。在这里，eval的实参就是echo $3 $2 $1，然后变成echo cc bb aa，才能打印出期望的结果。

【例 6-48】continue 使用的例子。

将命令行参数逆序输出，脚本程序中使用了提前终止循环体的内部命令continue。

```
$ cat rev3
count=$#
cmd=echo
while true
do
    cmd="$cmd \$$count"
    count=`expr $count - 1`
    [ $count -gt 0 ] && continue
    eval $cmd
    exit 0
done
$ ./rev3 aa bb cc dd
dd cc bb aa
```

其中的内部命令exit，终止shell程序, exit后面的参数，就是shell脚本程序结束的返回码。

【例 6-49】 从脚本程序中输入一个有效的 IP 地址。

从键盘读取一个IP地址值，默认值192.168.0.112, 直接输入回车时，取默认值。

判断一个输入是否是一个满足点分格式的IP地址，用了expr命令。但是这种判断能力还很弱，最后面的运行实例列出了漏掉错误的情况，257.222.340.231是个无效的IP地址。要想精确的判断一个合法的IP地址，可以设计一段更复杂的shell脚本程序，或者自己设计一个命令。

脚本程序中$addr两侧的引号必须保留，以防止用户输入的addr中会有空格。

```
$ cat getip
ADDR=192.168.0.112
while true
do
   echo "Please Input IP Address [$ADDR]: \c"
   read addr
   [ "$addr" = "" ] && addr=$ADDR
   if [ `expr "$addr" :
"[0-9][0-9]*\.[0-9][0-9]*\.[0-9][0-9]*\.[0-9][0-9]*$"` -gt 0 ]
   then
      break
   else
      echo "   **** Invalid IP Address !"
   fi
done
echo "IP Address is $addr"
$ ./getip
Please Input IP Address [192.168.0.112]:
IP Address is 192.168.0.112
$ ./getip
Please Input IP Address [192.168.0.112]: abcd
   **** Invalid IP Address !
Please Input IP Address [192.168.0.112]: 123.12.12.190
IP Address is 123.12.12.190
$ ./getip
Please Input IP Address [192.168.0.112]: 257.222.340.231
IP Address is 257.222.340.231
$
```

6.8 函数

从UNIX System V 2.0开始，才允许使用shell函数。

语法: *name*() { *list*;}

在调用函数时,引用函数的名字，可以附加上0到多个参数,在函数体内部以位置变量$1,$2,...或$*,$@方式引用函数的参数。

在函数体内部可以使用内部命令return,使函数有返回码,返回码0代表成功,非零表示失败。函数体内一个函数不能调用它自己。shell函数不允许递归调用。函数体的执行不创建新的子shell进程，它和脚本文件的其它部分一样，在同一个shell进程中执行。

shell脚本中可以使用#号作注释,如果#号出现在一个词的首部,那么，从#号至行尾的所有字符被忽略。类似C++的//注释。

有了函数机制，脚本文件中的某些代码就可以共享同一个函数。

【例 6-50】使用 shell 函数的例子。

这个例子是一个广域网通信适配卡驱动程序安装脚本的一部分。

这个通信适配卡安装之前要求输入硬件的中断号，I/O基地址和通信速率。然后，列出操作员的输入，等待确认后才执行安装操作。

安装操作在41~85行之间，被省略。

第4~16行的函数get_answer打印出一条消息，然后，强制输入y或者n，否则继续要求用户输入。函数体内使用了$1引用调用函数get_answer的命令行参数，用return使得shell函数象普通命令一样有返回码，供条件判断使用。

第20~34行的函数get_val有四个参数。第一个参数$1存放用户输入值的shell变量的名字，第二个参数$2是输入之前给用户的提示信息，第三个参数$3是用户直接按下回车时的缺省值，最后一个参数$4是允许取值的有效值列表。函数get_val强制用户输入一个有效值列表中的有效值，否则，就给出有效值列表做提示，并进一步要求用户重新输入。函数体内使用了while循环和for循环结构。第29行的break命令含有参数2，可以跳出两层循环。

脚本程序执行时从第36行开始执行。由于脚本文件前面有了函数说明，shell记下了函数名字get_val和get_answer作为内部命令，在执行命令get_val和get_answer时，shell都转去执行函数体，而不是到磁盘上寻找这样名字的命令文件。

```
$ awk '{print NR,$0}' WanCom
 1  # Shell function to read a Y/N response
 2  # Usage: get_answer <message>
 3  #
 4  get_answer()
 5  {
 6     while  true
 7     do
 8        echo "$1? (y/n) \c"
 9        read yn
10        case $yn in
11           [yY]) return 0;;
12           [nN]) return 1;;
13              *) echo "Please answer y or n" ;;
14        esac
15     done
16  }
```

```
 17  # Shell function to get a value
 18  # Usage: get_val <var_name> <message> <default_val> <list>
 19  #
 20  get_val()
 21  {
 22     while true
 23     do
 24        echo "$2 [$3] : \c"
 25        read val
 26        [ "$val" = "" ] && val=$3
 27        for i in $4
 28        do
 29           [ "$val" = "$i" ] && break 2
 30        done
 31        echo "**** Invalid choice $val, must be in $4"
 32     done
 33     eval "$1=$val"
 34  }

 35  #  main program

 36  get_val INTR "Interrupt Number" 10 "2 3 4 5 7 10 11 12 14 15"
 37  get_val PORT "I/O Base Address" 320 "200 210 220 230 300 310 320 330"
 38  get_val BAUD "Baud Rate" 9600 "2400 9600 14400 33600 64000"

 39  echo "Interrupt $INTR, I/O base address $PORT, Baud rate is $BAUD"
 40  get_answer "Do you want to install WanCom adapter driver" && {
 41     echo "Please Wait ...\c"
        ……
 85  }
$ ./WanCom
Interrupt Number [10] : 22
**** Invalid choice 22, must be in 2 3 4 5 7 10 11 12 14 15
Interrupt Number [10] : 14
I/O Base Address [320] :
Baud Rate [9600] : 33.6
**** Invalid choice 33.6, must be in  2400 9600 14400 33600 64000
Baud Rate [9600] : 33600
Interrupt 14, I/O base address 320, Baud rate is 33600
Do you want to install WanCom adapter driver? (y/n) y
Please Wait ...
...
$
```

6.9 shell 开关和位置变量

set命令后不跟任何参数时,列出shell的所有变量,包括局部变量和环境变量。set是内部命令,在B-shell和C-shell中用法会有区别。

6.9.1 set:设置 B-shell 内部开关

内部命令set可以用来设置一些shell开关，以影响shell的某些行为。常用的这些开关有：

-x　在每执行一条命令时,先打印出这个命令及命令参数,为区别于正常的shell输出,还在前面冠以+号。

+x　取消上述设置。

-u　当引用一个未赋值的变量时,产生一个错误。

+u　当引用一个未赋值的变量时,认为是一个空串。

【例 6-51】shell 脚本程序中-x 开关的作用。

脚本程序中打开-x 开关，可以观察到程序执行的流程。

```
$ cat chmod1
set -x
echo "$*"
for i
do
   [ -f $i ] && {
      chmod a+r $i
      eoho "$i is readable"
   }
done
$ ./chmod1 a*
+ echo a1 aa8 abcd
a1 aa8 abcd
+ [ -f a1 ]
+ chmod a+r a1
+ echo a1 is readable
a1 is readable
+ [ -f aa8 ]
+ [ -f abcd ]
+ chmod a+r abcd
+ echo abcd is readable
abcd is readable
```

本例中第一行的set -x也可以省略，使用命令sh -x chmodl a*也能达到相同的效果。或者，将脚本文件chmod1的第一行修改为!/bin/sh -x也可以。

使用shell的-x选项对shell脚本程序的调试(debug)非常有用。可以看出程序流程执行的轨迹。而且，使用-x选项，可以清楚地看到被变量替换，命令替换，文件名生成以及转义符的处理之后，真正要投入执行的最终命令是什么。

【例 6-52】交互式 shell 中-x 开关的作用。

交互式shell中打开-x开关，可以观察shell的命令替换，变量替换，文件名生成，以及转移符的作用。每次shell在真正执行一个命令之前都把要执行的命令显示出来。

```
$ set -x
$ tty
+ tty
/dev/tty6
$ termno=`expr \`tty\` : /dev/tty\\\\\\(.\\*\\\\\\)`
+ + tty
+ expr /dev/tty6 : /dev/tty\(.*\)
termno=6
$ echo $termno
+ echo 6
6
$ find $HOME -size +100 \( -name \*.c -o -name xxxx \) -exec rm -i {} \;
+ find /usr/jiang -size +100 ( -name *.c -o -name xxxx ) -exec rm -i {} ;
```

set -u开关也非常有用。一般情况下，程序员不会引用一个未赋值的shell变量。凡是出现了这种情况，往往是因为程序中引用变量时的变量名拼写错误。设置-u开关，一旦引用一个未赋值的shell变量，shell脚本程序就会立刻停下来，可以提前发现程序中的错误。

【例 6-53】shell 的-u 开关的使用。

可以检查出引用变量时的变量名拼写错误。

```
$ name=CSPTF
$ echo Connecting to $nmae Network
Connecting to Network
$ echo Connecting to $NAME Network
Connecting to Network
$ set -u
$ echo Connecting to $nmae Network
nmae: 0402-009 Parameter is not set.
$ echo Connecting to $NAME Network
NAME: 0402-009 Parameter is not set.
$ echo Connecting to $name Network
Connecting to CSPTF Network
$
```

6.9.2 set:设置 shell 位置变量

set命令后边跟多个参数，可以修改shell的位置变量。位置变量就是$1,$2,$3, ...，以及$#，$@和$*。位置变量还会影响未罗列出循环控制变量取值表的for结构循环。

【例 6-54】使用 set 命令设置 shell 的位置变量。

使用set命令可以重新设置shell的位置变量，先前的位置变量全部被新的值代替。

```
$ echo $#
0
$ date
Sun Jul 28 11:00:40 BEIJING 2004
$ set `date`
$ echo $1 $2 $3 $4
Sun Jul 28 11:00:40
$ ls -l /etc/motd
-rw-r--r-- 1 root staff 316 Jan 5 08:42 /etc/motd
$ set `ls -l /etc/motd`
sh:-rw-r-r-: bad option(s)
$ set -- `ls -l /etc/motd`
$ echo $9:$5 $1
/etc/motd:316 -rw-r--r--
```

注意--的使用，显式地指定set命令选项的结束，否则set会把以减号开头的命令行参数理解成set的选项。关于--，在前面的4.4.4节介绍过。

6.9.3 shift: 位置变量的移位

除了set命令外，内部命令shift，也可以影响位置变量。它的功能是使位置变量“移位”。例如：$#为4， $1,$2,$3,$4分别为aa,bb,cc,dd。那么，执行shift命令后，$#变为3,而$1,$2,$3分别变成bb,cc,dd。

shift命令还可以跟一个整数做参数，说明“移位”几个位置，上例中，如果执行shift 2命令，那么，$#变为2,而$1,$2分别变成cc,dd。

shift命令也影响位置变量的$*和$@。

【例 6-55】逐个打印源程序文件。:

打印多个源程序文件，每打印个文件之前列出文件名，打印文件时每行带上行号。

```
$ cat prt
while  [ $# -gt 0 ]
do
   echo ====================
   echo FILE NAME: $1
   echo ====================
   awk '{printf("2d %s\n",NR,$0)}' $1
   shift
done
$ ./prt makefile *.[ch]
```

【例 6-56】给出若干个程序名，终止这些程序文件启动的所有进程。

获取程序启动的进程的PID的方法，在前面的“元字符”6.5.10节中介绍过。这里的脚本程序中使用了位置变量和shift命令。

```
$ cat k
while [ $# != 0 ]
do
   echo "$1: \c"
   PID=`ps -e | awk "/[0-9]:[0-9][0-9] $1\\$/{printf(\\"%d \\",\\$1)}"`
   if [ -n "$PID" ]
   then
      echo kill $PID
      kill $PID
   else
      echo No process
   fi
   shift
done
$ ./k myap findkey sortdat
myap: kill 26506 38020
findkey: kill 31542
sortdat: No process
$
```

【例 6-57】软件安装时调整操作系统内核的部分参数。

下面的一段脚本程序，在SCO UNIX系统中安装某个软件包时，调整操作系统内核参数。第一行和第二行列出了相应的参数和期望的配置值。例如：要求MSGMNB参数取值至少32768, NQUEUE参数取值至少64。

命令configure的格式：

configure -y 参数名

configure 参数名=参数值 参数名=参数值 ……

第一种格式，打印出指定名字的内核参数的当前取值，第二种格式，调整指定名字的内核参数为指定的参数值，并且可以一次调整多个参数。这个命令是SCO UNIX专用的，在其它UNIX中没有通用性。

脚本程序使用set命令和shift命令对位置变量的影响。内核参数当前取值已经满足要求的参数不再调整。

```
 1 PARA="MSGMNB MSGTQL MSGSEG NBLK4096 NBLK2048 NBLK1024 NQUEUE"
 2 VAL=" 32768  600    16384   16       128       100      64"
 3 CHANGE=

 4 cd /etc/conf/cf.d
 5 set $VAL
 6 for i in $PARA; do
 7     x=`./configure -y $i`
 8     if [ $x -lt $1 ]
 9     then
```

```
10         echo "Adjusting parameter $i from $x to $1"
11         CHANGE="$i=$1 $CHANGE"
12     fi
13     shift
14 done

15 if [ -n "$CHANGE" ]
16 then
17     ./configure $CHANGE
18 fi
```