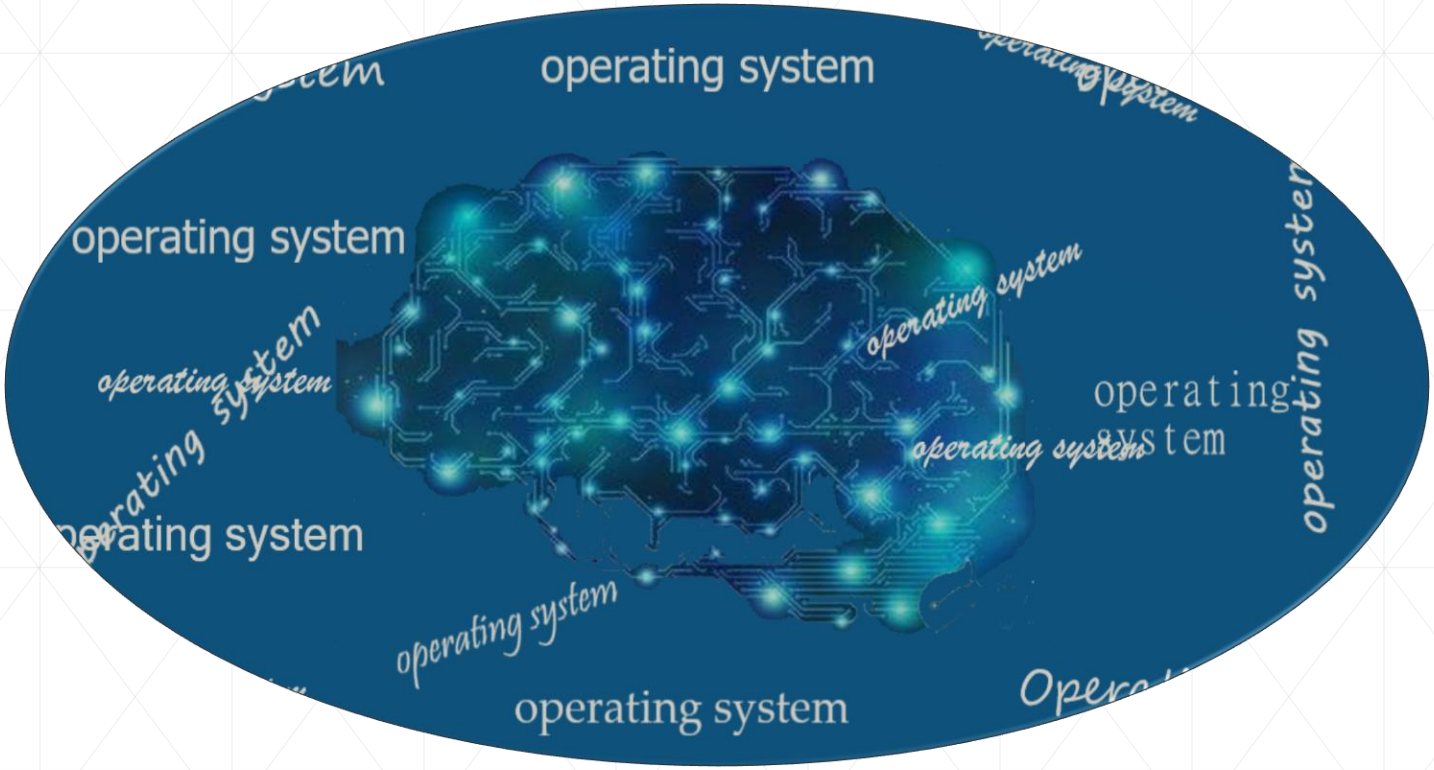


第八章 shell的交互功能与shell程序设计

授课教师：

电子邮箱：

本章主要内容

- Shell的启动和功能简介
- shell识别的命令形式
- 输入输出重定向和管道
- shell变量和引用符
- Shell脚本程序的建立与运行
- shell的语句类别
- ~~流编辑器sed和报表生成器awk简介~~

UNIX/Linux操作系统的概念

Linux的版本分为两类：内核版本和发行版本

1. 内核版本

- 内核是系统的心脏，是运行程序和管理像磁盘和打印机等硬件设备的核心程序，它提供了一个在裸设备与应用程序间的抽象层。

2. 发行版本

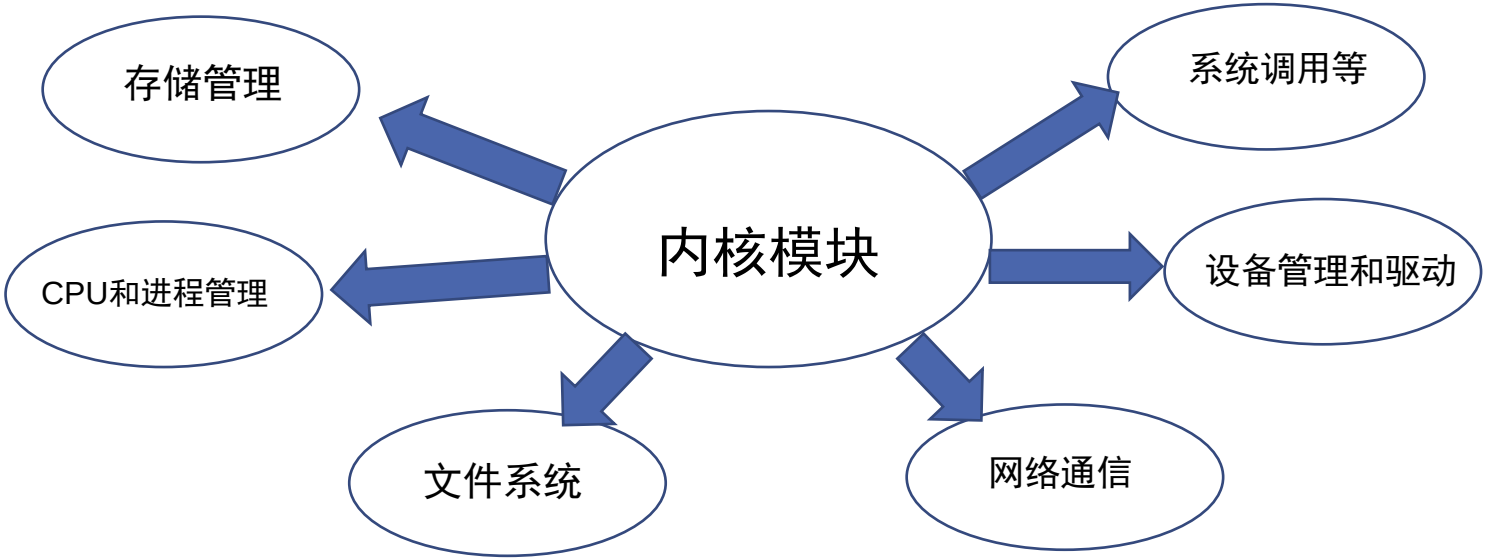
- 发行版是由发行商搜索一系列的应用程序打包发售时的编号。一个完整的Linux由“**内核程序 + 系统程序 + 应用程序**”组成。比较著名的几个发行版本有如下几个：
- 1）RedHat Linux或Fedora Core Linux； 2）Slackware Linux；
- 3）SuSE Linux； 4）Debian Linux； 5）红旗Linux（国产）

Linux操作系统的组成

- Linux操作系统由Linux内核，LinuxShell，Linux文件系统，Linux应用程序四大主要部分组成。
  - 内核是操作系统的核心，提供了操作系统最基本的功能
  - Shell是系统的用户界面，提供了用户与内核进行交互操作的一种接口。
  - 文件系统是文件存放在磁盘等存储设备上的组织方法。
  - 标准的Linux系统一般都有一套称为应用程序的程序集，即Linux应用程序

操作系统内核的概念

- 内核是操作系统的核心，提供了操作系统最基本的功能，如支持虚拟内存、多任务、共享库、需求加载、可执行程序和TCP/IP网络等。



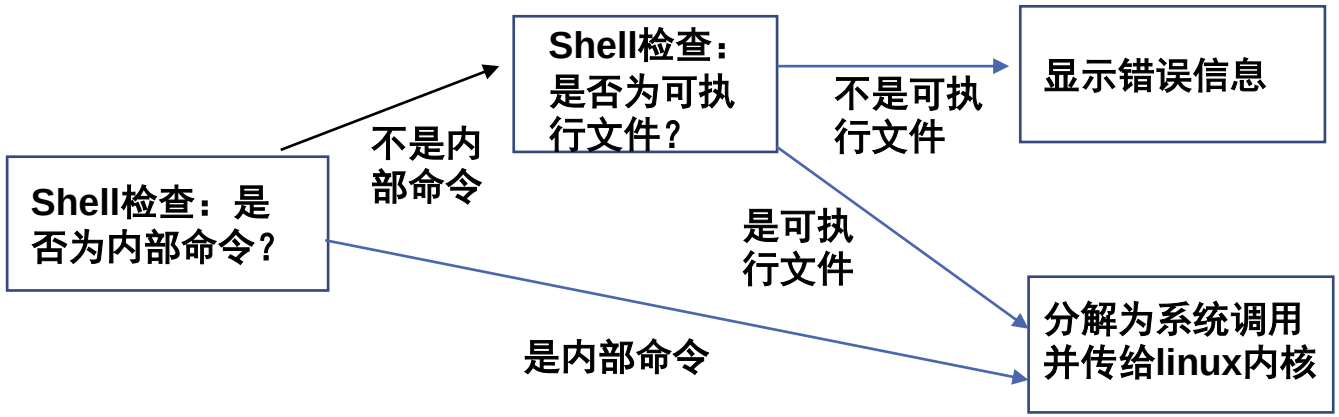
LinuxShell相关概念

- Shell是系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。实际上Shell是一个命令解释器，它解释由用户输入的命令并且把它们送到内核。

- Linux系统的Shell是命令语言、命令解释程序及程序设计语言的统称。

LinuxShell相关概念

- Shell是功能特点如下：
- （1）Shell是一个命令语言解释器：它拥有自己内建的Shell命令集，Shell也能被系统中其他应用程序所调用。用户在提示符下输入的命令都由Shell先解释然后传给Linux核心。使用户不必关心一个命令是建立在Shell内部还是一个单独的程序。



LinuxShell相关概念

- （2）Shell的另一个重要特性是它自身就是一个解释型的程序设计语言。

- Shell程序设计语言支持绝大多数在高级语言中能见到的程序元素，如函数、变量、数组和程序控制结构。shell编程语言简单易学，任何在提示符中能键入的命令都能放到一个可执行的Shell程序中。

第八章 shell的交互功能与shell程序设计

UNIX系统中的Shell具有两大功能:

- 命令解释器: 解释用户发出的各种操作系统命令
- 程序设计语言: 功能强大, 可包容引用所有的操作系统命令和可执行程序。

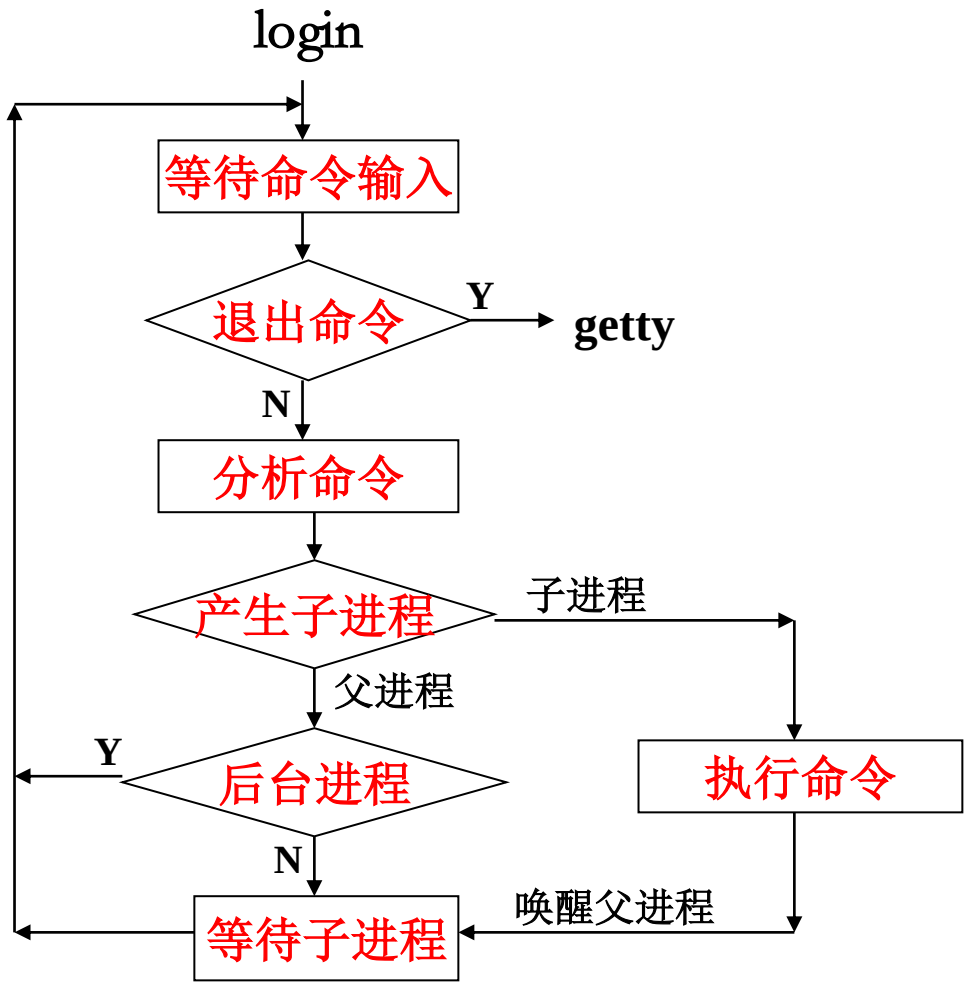
8.1 shell 的启动和终止

用户登录(login)进入系统时，系统根据

/etc/passwd

文件中的配置参数，为该用户启动一个指定类型的shell进程，专门用于解释执行该用户发出的各种命令. 当用户发出exit或Logout命令时，该shell程序终止运行，该用户退出系统，回到getty状态.

8.1 shell 的启动和终止



8.2 输入输出重定向和管道

标准输入：　键盘　fd＝0

标准输出：　荧光屏　fd＝1

标准错误输出：荧光屏　fd＝2

如果一个进程在运行时需要输入输出数据，在缺省状况下是从标准输入上读入数据，向标准输出上输出结果。利用shell的重定向操作符，可以把进程的输入和/或输出数据重新定向的任意其它文件。例如：

$ cat file　　运行结果（file的内容）送到标准输出荧光屏

$ passwd　　所需数据（新老口令）从标准输入键盘读入

$ ls -Y abc

ls: invalid option – Y　　错误信息送到标准错误输出（荧光屏）上显示

8.2 输入输出重定向和管道

8.2.1 输出重定向

command > filename 进程输出覆盖文件 filename

或 command >> filename 进程输出追加到文件filename后面，不覆盖filename

例如:

$ cat myfile

把文件myfile的内容输出到标准输出文件----荧光屏上

$ cat myfile > newfile

把文件myfile的内容输出到文件newfile中(标准输出已被重新定向到newfile). 其结果相当于拷贝文件.

$ cat abc >> xyz

把abc添加到xyz已有内容后面，而不是覆盖xyz

8.2 输入输出重定向和管道

应用举例:

① 按字母顺序显示当前系统中所有已登录的用户:

```
$ who  >  temp1          把当前登录用户的名单保存在temp1中
$ sort  temp1  >  temp2  把排序后的名单保存在temp2中
$ more  temp2            逐屏显示排好序的用户名单
$ rm  temp1  temp2       删除不用的临时文件
```

② 记录长时间运行程序的日志:

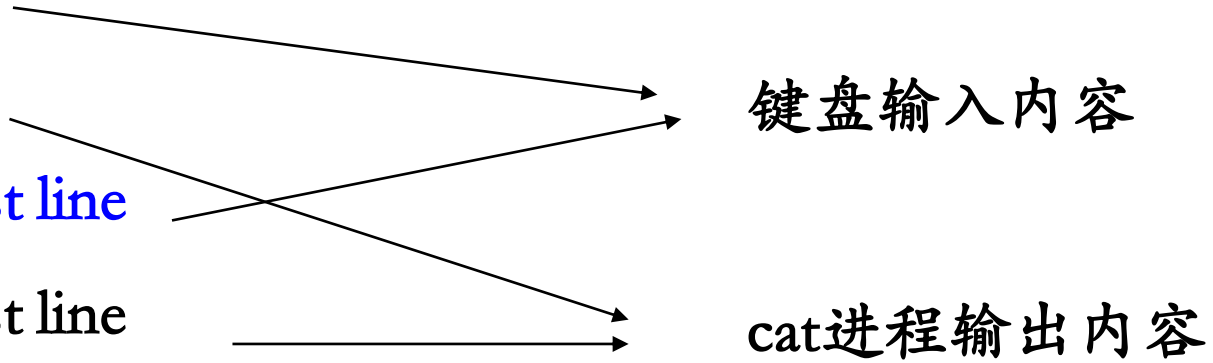
```
$ sort  temp1 >>  temp2   把排序后的名单累加(而不是覆盖)到文件temp2中。
```

8.2.2 输入重定向

command < filename

进程的输入来自文件filename，例如：

```
$ cat ↙          cat命令后无文件名，cat等待键盘输入
abcde
abcde
this is a test line
this is a test line
Ctrl+d
$
```



8.2 输入输出重定向和管道

```
$ cat  < abc
aaaaaaaaaaaaaa
bbbbbbbb
ccccccccccccccccc
$
```

cat进程的输入不是来自命令行参数，而是来自重定向文件abc

cat进程的输出送到标准输出荧光屏上

8.2.3 常见输入输出重定向形式

命 令	输入	输出	效果
cat	键盘	屏幕	将键盘输入显示在屏幕上
cat file1	file1	屏幕	file1的内容显示在屏幕上
cat file1 > file2	file1	file2	file1的内容写入file2中
cat > file2	键盘	file2	键盘输入的内容写入file2
cat < file1	file1	屏幕	file1的内容显示在屏幕上
cat < file1 > file2	file1	file2	file1的内容写入file2中

8.2.4 标准错误输出重定向

command 2> filename（2和>之间没有空格）

进程运行中的错误信息重定向到文件filename，例如：

```
$ cc -o core_prt core.c 2> err.log
```

在编译过程中如果出现core.c不存在或不能读、core_prt没有写权限等错误，则把错误信息保存在文件err.log中.

```
$ cat file1 file2 > file3 2> errfile
```

如果命令运行正常，则把结果(连接file1和file2)存入file3中；如果出现错误，则把错误信息存放到errfile中.

8.2 输入输出重定向和管道

```
$ grep  string6  data_sav  >  count_log  2> &1
```

把进程的出错信息存放到标准输出(已重定向到count_log)中，即把标准输出和标准错误输出都定向到一个文件中.

本例中“2> &1”表示把标准错误输出送到标准输出文件中去。由于前面先已把标准输出重定向到了count_log文件，所以本进程运行时如果有错误，则错误信息也送到count_log 中。

8.2 输入输出重定向和管道

8.2.5 管道

管道用于连接两个命令，它把前一个命令的标准输出重定向给后一个命令作为标准输入，其格式为：

command1 | command2

对command1来说标准输出被重新定向到管道，对command2来说标准输入也被重新定向为管道.

例题1，在4.2.1节中的例题"按字母顺序显示当前系统中所有已登录的用户"

```
$ who  > temp1
$ sort  temp1 > temp2
$ more temp2
$ rm temp1 temp2
```

```
$ who | sort | more
```

8.2 输入输出重定向和管道

例题2:

$ who | wc -l

查看系统当前有几个用户在上机使用系统.

$ pr myfile | lp

把文件myfile按标准打印格式处理后,送到打印机打印出来(原文件myfile并未作任何修改).

$ grep student user_list | sort > stu_list

在包含所有用户名单的文件user_list中,查找包含student的行,并把结果排序后存放在文件stu_list中.(管道和输出重定向混合使用)

8.3 shell 可识别的命令形式

8.3.1 单条命令

```
$ cat   file1
```

这是最常用命令形式, 本命令执行完成后出现shell提示符, 再接收下一条键盘命令.

8.3.2 多条命令

```
$ pwd; who; date
```

第一条命令执行完成后, (无停顿)再执行第二条命令, 如此下去. 运行功能和效果与在键盘上逐条输入命令并运行是完全一样的, 其主要目的是提高键盘命令输入效率.

8.3 shell 可识别的命令形式

8.3.3 复合命令

① $ ps －e | grep student2

管道前后的命令任意组合、同时运行,形成功能更强大灵活的复合命令.

② $ (ls ; cat file3 ; pwd) > run_log

括号内的命令任意组合、顺序执行，且由一个子shell来单独控制运行,相当于一个小的功能程序. 方便灵活,运行效率高.

8.3 shell 可识别的命令形式

8.3.4 后台命令

```
$ ls -lR > file_list &
  [1]  7981
$
```

普通命令行的行尾加上&符号，就表示该命令在后台执行。

[1]：当前shell的后台作业（进程）序号

7981：当前这个后台进程（ls进程）的进程号（PID）。

Shell 启动该后台进程后不再等待该进程结束，立即开始接受新的键盘命令——多进程并发，数量不限，充分利用系统资源。

8.4 shell 变量和引用符

每一个shell都可以设定一组变量来限定shell及其子程序的功能模式和取值范围，这些变量中有些是系统设定的，有些是由用户设定的. 每个shell都可以有完全不同的变量设置，由此构成各具特色的运行环境。

系统的基本环境变量放在/etc/profile中，用户环境变量放在用户主目录下的.profile文件中，用户shell启动时，先执行/etc/profile，再执行用户主目录下的.profile。

环境变量可在shell运行时动态修改。

8.4 shell 变量和引用符

8.4.1 环境变量

常用的shell环境变量及实例:

HOME=/usr/computer/student6　　用户主目录,注册时的初始目录

PATH=/bin:/usr/bin:$HOME/bin:./　　键盘命令的搜索路径

SHELL=/bin/sh　　用户的初始shell的路径名称

TERM=vt100　　当前所用的终端类型

PS1=$　　shell的主提示符

IFS=　　域分隔符,通常为空白符,用来分隔命令行各个域

8.4 shell 变量和引用符

echo 命令的使用

echo命令的基本功能就是在标准输出上显示后面的字符串，或变量的值。当字符串中带空白符或其它控制字符时，用引号将其括起来。例如：

```
$ echo  12345
12345
$ echo  "department computer"
department computer
$ echo "My home directory is: $HOME"
My home directory is:  /usr/teacher/david
$ echo –n "Input your choice (y/n) [  ]\b\b"
Input your choice (y/n) [ _ ]
```

8.4.2 系统变量

常用系统变量:

$0	当前shell程序的名字
$1 ~ $9	命令行上的第一到第九个参数
$#	命令行上的参数个数
$*	命令行上的所有参数
$@	分别用双引号引用命令行上的所有参数
$$	当前进程的进程标识号(PID)
$?	上一条命令的退出状态
$!	最后一个后台进程的进程标识号

系统变量只能引用不能修改!

8.4 shell 变量和引用符

系统变量应用举例：

```
$ echo aa  bb  cc  dd  $$
aa  bb  cc  dd  2391
$ cat  file1  file2  > file3  2> errlog
$ echo $?    (非0表示命令运行失败，错误信息在 errlog 文件中)
$ echo
                    (空行，即echo输出串尾隐含的换行符)
$ echo  This is      a      test.      (单词间多个空格)
This is a test.
$ echo  "This is      a      test."     (用引号包括时结果如何?)
```

8.4.3 局部变量(用户变量)

局部变量是由用户根据需要任意创建的. 变量名通常由一个字母后跟零个到多个字母、数字或下划线组成。引用变量的值时，在变量名前面加上$符号. 例如:

```
$ AA=123                          定义变量AA
$ echo  $AA                       引用变量AA的值
123                               (变量AA的值)
$  B="this is a string"           定义变量B，(字符串中有空格时用引号)
$  echo  $B                       引用变量B的值
this is a string                  (变量B的值)
```

8.4 shell 变量和引用符

8.4.4 单引号、双引号、反撇号和花括号

```
$ a="he is a student"
$ echo "She said: $a"
```

She said: he is a student　　echo执行时，替换了变量$a的值

```
$ b='The value of a is $a'
$ echo $b
```

The value of a is $a　　echo执行时，未替换了变量$a的值

shell规定单引号禁止变量替换，元字符$和*等保持其符号本身；而双引号允许元字符变量替换.

```
$ c="The value of a is $a"
$ echo $c
```

The value of a is he is a student

8.4 shell 变量和引用符

8.4.4 单引号、双引号、反撇号和花括号

```
$ a=date
$ echo $a
date                    (变量a的值是字符串date)
$ b=`date`              (反撇号中的字符串作为命令名)
$ echo $b
Sat Feb 1 16:28:19 Beijing 2003
                        (变量b的值是反撇号中命令的执行结果)
```

8.4 shell 变量和引用符

8.4.4 单引号、双引号、反撇号和花括号

```
$ c="There is a teach"
$ echo "$cer reading room"
reading room          (未定义变量cer, 其值用空串替代)

$ echo "${c}er reading room"
There is a teacher reading room
(花括号将变量名和后面的字符串区分开)
```

8.4.5 变量输出命令 export

新的shell变量定义后或已有的shell变量修改值后，如果未经export命令输出，则只在当前的shell中起作用，对其各个子shell不产生任何影响. 经过 export 命令输出的变量才能对当前shell的各个子shell、以及子shell的子shell起作用。例如：

```
$ PATH=$PATH:./
$ export  PATH
```

说明:

1. export后面的变量名前不加$符号
2. 经export输出给子shell的变量如果在子shell中被修改，则只影响子shell，不影响父shell；如果在子shell中被输出，则只影响子shell的子shell
3. export命令常用在.profile文件中

8.5 Shell的内部命令

shell 的内部命令包含在shell内部, 不是一条单独的操作系统命令, 因此无法在文件系统中查找到.

例如:

cd　　改变当前工作目录

pwd　显示当前工作目录

time 显示当前shell运行命令所花费的时间，例如:

```
$ time  ls –lR /usr  >  flist
real    2m 17.32s     该ls进程的总运行时间
user    0m 7.63s      其中用户程序部分的运行时间
sys     0m 6.79s      其中操作系统核心部分运行时间
```

8.6 进程监控

8.6.1 进程及进程状态

UNIX/Linux系统中的“进程”是可运行程序在内存中的一次运行实例。

进程和程序的主要区别是:

. 进程是动态的,它有自己的生命周期和不同状态;而程序是静态的,通常存放在某种介质(如磁盘或纸张等)上。

. 进程具有运行控制结构和作用数据区;程序没有。

. 一个程序可以同时在内存中有多个运行实例,即同时作为多个进程的组成部分。

8.6 进程监控

8.6.1 进程及进程状态

每个进程运行时都有如下生命周期:

创建→运行→等待→运行…等待→运行→结束

生命周期大致分为三种状态:

. 运行态

进程正占用CPU和其它资源进行运算.

. 就绪态

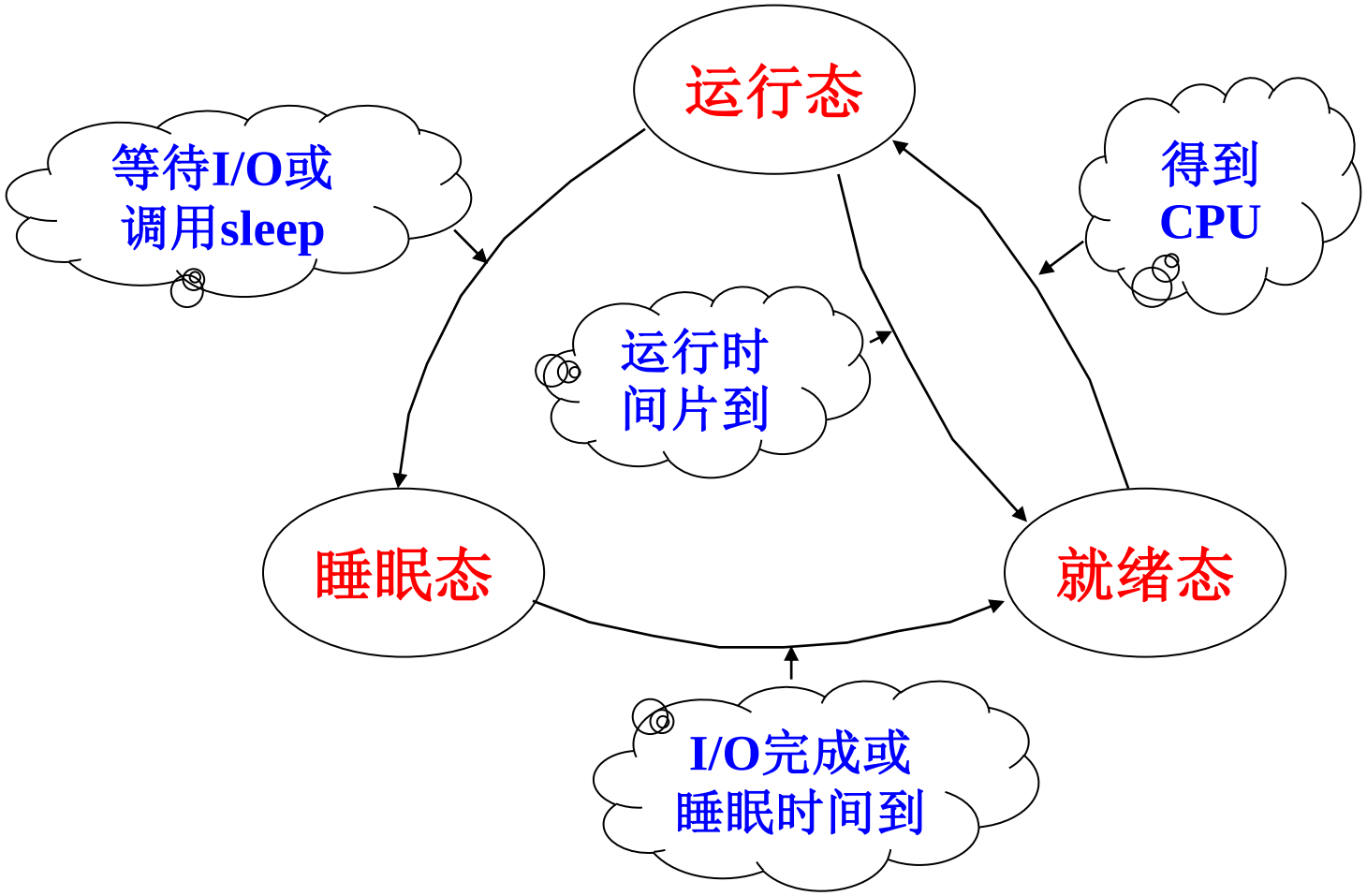
进程已做好一切准备, 等待获得CPU投入运行.

. 睡眠态

进程因等待输入输出或其它系统资源, 而让出CPU资源, 等待运行条件满足。

8.6 进程监控

8.6.1 进程及进程状态



8.6.2 获取进程状态信息： ps 命令

不带参数的ps命令运行时，显示该用户当前活动进程的基本信息：

```
$ ps
  PID      TTY      TIME      COMMAND
  612      tty08    0:37      sh
  931      tty08    0:01      ps
  $
```

PID　　进程标识号. 系统每个进程在其生命周期都有一个唯一的PID.
TTY　　启动该进程的终端号
TIME　　进程累计占用CPU的时间
COMMAND　产生该进程的命令

8.6.2 获取进程状态信息： ps 命令

ps命令的常用任选项 -e (或-a)　显示系统中所有活动进程的信息.

```
$ ps  -e
PID        TTY        TIME       COMMAND
0            ?        0:00       swapper
1            ?        0:01       init
358          00       0:01       sh
695          01       0:01       sh
23           ?        0:00       logger
732          03       0:00       vi
25           ?        0:00       cron
681          02       0:00       getty
623          03       0:01       csh
732          01       0:01       ps
$
```

8.6 进程监控

8.6.2 获取进程状态信息： ps 命令

-f 显示该进程的所有信息. 例如:

```
$ ps -f
UID   PID   PPID   C   STIME    TTY   TIME    COMMAND
liu    298   1      0   14:57:02   02    0:02    sh
liu    395   298    16  16:31:19   02    0:00    ps -f
```

其中:

UID	进程所有者的用户标识数
PID	进程标识数
PPID	本进程的父进程标识数
C	进程调度参数, 反映本进程使用CPU的状况
STIME	进程的启动时间
TTY	启动进程的终端
TIME	进程累计占用CPU的时间
COMMAND	启动该进程的命令名

8.6.3 暂停进程运行: sleep 命令

sleep time

sleep命令使运行它的进程暂停time指定的秒数.

例如:

```
$ sleep 5
   [进程暂停5秒钟,什么也不作]
$
$ sleep 10; who
   [进程暂停10秒钟后,显示系统中登录的用户名]
$ echo "I am sleeping..."; sleep 100; echo "I am awake"
I am sleeping ...    [等待100秒钟]
I am awake
$
```

8.6.4 终止进程运行: kill 命令

通常在三种情况下进程被终止运行:

. 进程运行完成, 自动消亡;

. 用户按^c 或 Del 等中断键, 强行终止前台进程的运行;

. 用户发出 kill 命令, 强行终止后台进程或键盘锁住了的前台进程的运行.

kill 命令的三种常用格式为:

kill PID

正常结束进程, 自动完成所有善后工作, 作用类似于按 Del 键.

kill -1 PID

先挂起该进程, 终止子进程, 完成善后工作, 再终止该进程.

kill -9 PID

立即强行终止该进程, 不作任何善后工作. 可能出现资源浪费和"孤儿"进程.

8.7 shell 编程

8.7.1 shell 编程的基本过程

主要包含以下三步:

1. 建立 shell 文件

包含任意多行操作系统命令或shell命令的文本文件;

2. 赋予shell文件执行权限

用chmod命令修改权限;

3. 执行shell文件

直接在命令行上调用该shell程序.

8.7 shell 编程

8.7.2 实例

1. 建立shell文件（可用任何建立文本文件的方法）：

```
$ cat  prog1

 who  |  grep  $1
```

2. 赋予执行权限：（初始文本文件无执行权限）

```
$ chmod  740  prog1
```

3. 执行该shell程序

```
$ prog1  student5
prog1: not found
```

（shell在标准搜索目录中找不到prog1命令）

4. 指定路径或修改环境变量PATH后执行shell程序

```
$ ./prog1  student5
student5   tty06   Feb 8   09:12
```

8.7.3 shell 程序和语句

shell 程序由零至多条shell语句构成. shell语句包括三大类: 功能性语句、结构性语句和说明性语句.

说明性语句:

以#号开始到行尾的部分，不被解释执行。

功能性语句:

任意的操作系统命令、shell内部命令、自编程序、其它shell程序名等。

结构性语句:

条件测试语句、多路分支语句、循环语句、循环控制语句等。

8.7.3.1 说明性语句(注释行)

注释行可以出现在程序中的任何位置,既可以单独占用一行，也可以接在执行语句的后面. 以#号开始到所在行的行尾部分，都不被解释执行. 例如:

```
#! /bin/sh
#
# 本程序说明
#
command_1
command_2                    # command_2的语句说明
……
# 下面程序段的说明
command_m
……
command_n                    # command_n语句的说明
……
```

告诉OS用哪种类型的shell来解释执行该程序

8.7.3.2 几个常用的功能性语句(命令)

1、read命令

read从标准输入读入一行, 并赋值给后面的变量, 其语法为:

. read var

把读入的数据全部赋给var

. read var1 var2 var3

把读入行中的第一个参数赋给var1, 第二个参数赋给var2, ……, 把其余所有的参数赋给最后一个变量.

如果执行read语句时标准输入无数据, 则程序在此停留等侯, 直到数据的到来或被终止运行.

8.7 shell 编程

read应用实例（命令行执行）

read 读取标准输入（键盘），输入的内容赋给后面的变量.

```
$ read name age
    yilan 23                                  从键盘输入的内容
$ echo "student $name is $age years old"
    student yilan is 23 years old             屏幕显示的内容
```

8.7 shell 编程

read应用实例（shell脚本程序）

```
# example1  for read
echo  -n "Input  your  name: "
read  username
echo  "Your name is  $username"
```

read应用实例（shell脚本程序）

```
#example2 for read
echo –n "Input date with format yyyy mm dd: "
read year month day
echo "Today is $year/$month/$day, right?"
echo -n "Press any key to confirm and continue"
read answer
echo "I know the date, bye!"
```

2、 expr 命令

算术运算命令expr主要用于进行简单的整数运算，包括加(+)、减（-）、乘（*)、整除（/）和求模（%）等操作。例如:

```
$ expr 12 + 5 \* 3
27
$ expr 3 - 8 / 2
-1
$ expr 25 % 4
1
$ num=9
$ sum=`expr $num \* 6`
$ echo $sum
54
```

反斜线去掉*号的元字符含义

反撇号引用命令的运行结果

8.7 shell 编程

3、 tput 命令

tput命令主要用于设置终端工作模式，或读出终端控制字符。tput命令与终端控制代码数据库terminfo相连，根据shell环境变量TERM的值，读出这种终端的指定功能控制代码。常用的终端显示功能控制符如下表：

选项	功　能	选项	功　能
bel	终端响铃	el	光标位置到行末清字符
blink	闪烁显示	smso	启动突出显示模式
bold	粗体字显示	smul	启动下划线模式
clear	清屏	rmso	结束突出显示模式
cup r c	光标移到 r 行 c 列	rmul	结束下划线模式
dim	显示变暗	rev	反白显示
ed	光标位置到屏幕底清屏	sgr0	关闭所有属性

8.7 shell 编程

tput应用实例1:

```
#
# program1   for   tput
#
tput  clear
tput  cup  11  30
tput  rev
echo  "Hello,  everybody!"
tput  sgr0
tput  cup  24  1
```

该程序先清屏，并在屏幕中央位置(11行30列)用反极性显示字符串“Hello, everybody!”，恢复正常显示极性后光标停留在屏幕左下角。

8.7 shell 编程

tput应用实例2:

```
#
# program2   for   tput
bell=`tput  bel`
s_uline=`tput  smul`
e_uline=`tput  rmul`
tput  clear
echo  $bell   $s_uline
tput  cup  10  20
echo  "Computer  Department"
echo  $e_uline
```

功能: 响一声铃后，在清空的屏幕中央以下划线模式显示字符串“Computer Department”，完成后重置正常显示模式。

8.7.4 结构性语句

结构性语句主要根据程序的运行状态、输入数据、变量的取值、控制信号以及运行时间等因素来控制程序的运行流程。主要包括以下几类语句：

- 条件测试语句（两路分支）
- 多路分支语句
- 循环语句
- 循环控制语句

无任何结构性语句的shell脚本程序是一种特例。

8.7 shell 编程

条件语句 if...then...fi

语法结构:

```
if   表达式
then 命令表
fi
```

如果表达式为真,则执行命令表中的命令;否则退出if语句,即执行fi后面的语句。 if和fi是条件语句的语句括号,必须成对使用。

命令表中的命令可以是一条,也可以是若干条;既可以直接在then语句后面写,也可以提行书写。

8.7 shell 编程

if语句应用实例

shell程序prog2, 测试命令行参数是否为一个已存在的文件或目录。 用法为:

prog2 file

prog2脚本内容如下:

```
#The statement of  if…then…fi                (注释语句)
if [  -f   $1  ]                             (测试命令行第一个参数是否为文件)
then
        echo "File  $1  exists"              (引用变量值)
fi
if [  -d   $HOME/$1  ]                       (测试参数是否为目录)
then
        echo "File  $1  is  a  directory"    (引用变量值)
fi
```

8.7 shell 编程

执行prog2程序:

$ prog2 prog1

File prog1 exists

$0为prog2; $1为prog1, 是一个已存在的文件.

$ prog2 backup

File backup is directory

$0为prog2; $1为backup,是一个已存在的目录.

如果不带参数，或大于一个参数运行prog2，例如:

$ prog2 (或 $ prog2 file1 file2)

将会出现什么结果?

8.7 shell 编程

条件语句 if...then...else...fi

语法结构为:

```
if    表达式
then 命令表1
else  命令表2
fi
```

如果表达式为真,则执行命令表1中的命令,退出if语句;否则执行命令表2中的语句,退出if语句。

注意: 无论表达式是否为真, 都有特定的语句要执行。

8.7 shell 编程

应用实例： shell程序prog3，用法为：

prog3 file

prog3的内容如下：

```
#The statement of if…then…else…fi
if [ -d $1 ]
then
        echo "$1 is a directory"
        exit                                    (立即退出当前的shell程序)
else
        if [ -f $1 ]
        then
                echo "$1 is a common file"
        else
                echo "unknown"
        fi
fi
```

（如果没有exit语句，程序的执行结果有什么不同？）

8.7 shell 编程

运行prog3程序:

```
$ prog3  backup
backup  is  a  directory
$ prog3  prog1
prog1  is  a  common  file
$ prog3  abc
unknown
```

prog3是对prog2的优化, 逻辑结构更加清晰合理!

prog2的主要缺点是什么?

测试语句 test

test语句可测试三种对象:

字符串　整数　文件属性

每种测试对象都有若干测试操作符，例如:

```
test "$answer" = "yes"
```

变量answer的值是否为字符串yes

```
test $num -eq 18
```

变量num的值是否为整数18

```
test -d tmp
```

测试tmp是否为一个目录名

8.7 shell 编程

test命令的应用:

test命令测试的条件成立时,命令返回值为真(0)，否则返回值为假(非0).

用法一.

```
test  $name  -eq  $1
echo  $?
```

用法二.

```
if  test  -f  $filename
then
      ……
fi
```

用方括号替代test语句

通常简写为:

```
if [  -f  $filename ]
```

至少一个空格

至少一个空格

8.7 shell 编程

多路分支语句 case…esac

多路分支语句case用于多重条件测试，语法结构清晰自然．其语法结构为：

```
case  字符串变量  in
    模式1)
        命令表1
        ;;
    模式2)
        命令表2
        ;;
    ……
    模式n)
        命令表n
        ;;
esac
```

case语句只能检测字符串变量

各模式中可用文件名元字符,以右括号结束

命令表以单独的双分号行结束,退出case语句

模式 n常写为字符* 表示所有其它模式

最后一个双分号行可以省略

8.7 shell 编程

实例. 程序prog4检查用户输入的文件名，用法为:

```
          prog4    string_name
# The statement of  case…esac
if [  $#  -ne  1  ]
then
     echo  “One argument must be declared”
     exit
fi
case  $1  in
     file1)
          echo  ”User selects file1”
          ;;
     file2)
          echo  ”User selects file2”
          ;;
     *)
       echo  ”You must select either file1 or file2!”
          ;;
esac
```

8.7 shell 编程

循环语句 for…do…done

当循环次数已知或确定时，使用for循环语句来多次执行一条或一组命令. 循环体由语句括号do和done来限定。 格式为:

```
for  变量名  in  单词表
do
        命令表
done
```

变量依次取单词表中的各个单词，每取一次单词，就执行一次循环体中的命令. 循环次数由单词表中的单词数确定. 命令表中的命令可以是一条，也可以是由分号或换行符分开的多条。

如果单词表是命令行上的所有位置参数时，可以在for语句中省略“in 单词表”部分。

8.7 shell 编程

实例. 包含for语句的程序prog5寻找指定文件, 或拷贝当前目录下的所有文件到backup子目录下. 使用语法及程序为:

prog5 [filename]

```
# The statement of for…do…done
if [ ! -d $HOME/backup ]
then
        mkdir $HOME/backup
fi
flist=`ls`
for file in $flist
do
        if [ $# = 1 ]
        then
                if [ "$1" = "$file" ]
                then
                        echo "$file found"; exit
                fi
        else
                cp $file $HOME/backup
                echo "$file copied"
        fi
done
echo '***Back up completed***'
```

flist的值是ls的执行结果即当前目录下的文件名

命令行上有一个参数时

命令行上不带参数时

8.7 shell 编程

循环语句while…do…done

语法结构为：

```
while    命令或表达式
do
        命令表
done
```

while语句首先测试其后的命令或表达式的值，如果为真，就执行一次循环体中的命令，然后再测试该命令或表达式的值，执行循环体，直到该命令或表达式为假时退出循环。

while语句的退出状态为“命令表”中被执行的最后一条命令的退出状态。

8.7 shell 编程

实例. 创建文件程序prog6, 用法为:

prog6 file [number]

```
# The statement for while
if [ $# -eq 2 ]
then
        loop=$2
else
        loop=5
fi
i=1
while [ $i -le $loop ]
do
        >$1$i
        i=`expr $i + 1`
done
```

根据命令行参数的个数确定循环的次数

建立以file开头, 变量i的值结尾的空文件名. 参见命令 **cmd > file**

8.7 shell 编程

循环语句until…do…done

语法结构为： until 命令或表达式

do

命令表

done

until循环与while循环的功能相似，所不同的是只有当测试的命令或表达式的值是假时，才执行循环体中的命令表，否则退出循环。 这一点与while命令正好相反。

8.7 shell 编程

循环控制语句 break 和 continue

break语句从包含该语句的最近一层循环中*跳出*一层，break n 则跳出n层；continue语句则马上转到最近一层循环语句的*下一轮*循环上，continue n则转到最近n层循环语句的下一轮循环上.

实例. 程序prog7的用法为:

prog7 整数 整数 整数 …

参数个数不确定，参数的数量为1~10个，每个参数都是正整数。

prog7的代码如下:

8.7 shell 编程

```
if [ $# -eq 0 ]
then
        echo "Numeric arguments required"
        exit
fi
if [ $# -gt 10 ]
then
        echo "Only ten arguments allowed"
        exit
fi
for number
do
        count=`expr $number % 2`
        if [ $count -eq 1 ]
        then
                continue
        else
                output="$output $number"
        fi
done
echo "Even numbers: $output"
```

取所有位置参数

用2求模, count的值只能是0或1

是奇数

下轮循环

偶数放到偶数队列中

8.7 shell 编程

实例. 包含while、until和break语句的prog8 程序.

```
while true
do
        echo level1
        until false
        do
                echo level2
                while true
                do
                        echo level3
                        break
                done
        done
done
```

无限循环程序，只能由人工中断

改为continue 3 开始最外层循环

如果改为break 3 跳出最外层循环

8.8 shell 函数

在shell程序中，常常把完成固定功能、且多次使用的一组命令（语句）封装在一个函数里，每当要使用该功能时只需调用该函数名即可。

函数在调用前必须先定义，即在顺序上函数说明必须放在调用程序的前面。

调用程序可传递参数给函数，函数可用return语句把运行结果返回给调用程序。

函数只在当前shell中起作用，不能输出到子shell中。

8.8 shell 函数

shell函数的说明格式:

```
function_name ()
{
    command1
    ……
    commandn
}
```

函数的调用格式:

```
value_name=`function_name  [arg1 arg2 … ]`
```

或者:

```
function_name  [arg1  arg2  …  ]
echo   $value_name
```

函数的所有标准输出都传递给了主程序的变量

函数的返回值隐含在变量中,由主程序使用该变量的值

8.8 shell 函数

shell函数调用实例:

```
check_user()    #查找已登录的指定用户
{
     user=`who | grep $1`
      if [ -n  “$user”  ]
      then
           return 0      #找到指定用户
      else
           return 1      #未找到指定用户
      fi
}
while true        # MAIN, Main, main:  program begin here
do
        echo -n "Input username: "
        read  uname
        check_user $uname    # 调用函数, 并传递参数uname
        if [ $? -eq 0 ]              # $?为函数返回值
        then  echo "user $uname online"
        else  echo "user $uname offline"
        fi
done
```

8.9 shell编程 — 实例1

```
#!/bin/sh
# Program name: numberit
# Put line numbers on all lines of a file
if [ $# -ne 1 ]
then
      echo "Usage: $0 filename " >&2
      exit 1
fi
count=1                  # Initialize count
cat $1 | while read line
# Input is coming from file on command line
do
    [ $count -eq 1 ] && echo "Processing file $1..." > /dev/tty
    echo $count $line
   count='expr $count + 1'
done > tmp$$          # Output is going to a temporary file
mv tmp$$ $1
```

8.9 shell编程 — 实例1

运行情况:

```
$ cat  test_file
abc
def
ghi

$ numberit  test_file
 Processing  file  test_file …

$ cat  test_file
1  abc
2  def
3  ghi
```

8.9 shell编程 — 实例2

```
#!/bin/sh
# Scriptname: speller
# Purpose: Check and fix spelling errors in a file
>file.new
while read line                    # Read from the tmp file
do
    echo $line
   echo -n "Is this word correct? [Y/N] "
    read answer < /dev/tty         # Read from the terminal
    case "$answer" in
    [Yy]*)
        echo $line >> file.new
        ;;
      *)
        echo "What is the correct spelling? "
        read word < /dev/tty
        echo $word >> file.new
        echo $line has been changed to $word.
    esac
done < file.old
```

8.10 流编辑器 sed

一. 什么是流编辑器?

流编辑器是一种流水线型的、非交互式的文本编辑器。它使用户可以在命令行上（而不是编辑器中）对文件进行无破坏性编辑。

8.10 流编辑器 sed

屏幕编辑器与流编辑器的区别

项目	vi	sed
1. 用户操作方式	交互式	非交互式
2. 文本处理模式	全局并行(可逆行)	逐行串行(不可逆行)
3. 编辑命令地点	编辑器中	命令行上
4. 编辑空间	临时文件(文件缓存)	模式空间(行缓存)
5. 对原文本影响	破坏性的	非破坏性的
6. 批量发出命令	不能	可以
7. 基本编辑单位	字符	行
8. 主要应用场合	人工编辑	程序自动编辑
9. 可编文件大小	较小	较大

8.10 流编辑器 sed

8.10.1 sed命令的基本格式

1. sed ‘command’ file
2. sed -n ‘command’ file
3. sed -e ‘command1’ –e ‘command2’ file
4. sed -f cmd_file file

任选项说明:

command: 普通行编命令

-n: 只显示与模式匹配的行(缺省都显示)

-e: 在同一命令行上进行多次编辑

-f: 编辑命令放在随后的命令表文件中

file: 被编辑的文本文件

8.10 流编辑器 sed

注意:

sed命令的结果是送到标准输出上，即荧光屏上，如果要将结果保存在文件中，应该使用重定向功能！

例如：

sed ‘s/student/teacher/g’ oldfile > newfile

如果要把结果保留在被编辑的原文件中，该如何办？

8.8.2 sed中常用的行编辑命令格式

1. [行定位符][编辑命令元字符]

例如: sed ‘1,9d’ abc

sed -n ‘196p’ abc

2. /正则表达式/[编辑命令元字符]

例如: **sed -n ‘/student/p’ filename**

sed ‘/xyz/d’ filename

3. **[定位符][元字符]/正则表达式/[元字符]**

例如: **sed –n ‘3,8s/east/west/’ filename**

sed –n ‘1,$s/computer/network/g’ filename

8.8.3 sed中常见的出错信息和退出状态

1. 操作系统命令出错:

sed -r 's/this/that/' myfile

显示: sed: ERROR: Illegal option – r

退出状态值: 1

例如:

sed -n 's/this/that/' newfile

显示: sed: ERROR: Cannot open newfile:
No such file or directory

退出状态值: 2

8.10 流编辑器 sed

2. 正则表达式出错和模式不匹配:

sed -n ‘s/this/that’ newfile

显示: sed: ERROR: Command garbled:

s/this/that

退出状态值: 0

例如:

sed -n ‘s/this/that/’ newfile

显示: 无 (文件newfile中无this字符串匹配)

退出状态值: 0

8.10 流编辑器 sed

3. 出错信息保存和退出状态值检测

保存出错信息:

sed －n ‘1,$s/abc/xyz/’ file 2> err_log

或: sed －n ‘1,$s/abc/xyz/’ file 2>> err_log

(这两条命令功能上的区别是什么?)

常用的退出状态值检测方式:

① **echo $?**
其它间接处理

② **if [$? -eq 0]**
then
正常处理
else
出错处理
fi

8.8.4 sed应用实例

1. 打印文件内容：p命令

sed -n ‘22, 35p’ file1

打印file1的第22~35行

sed -n ‘/string/p’ file2

打印file2中包含string的行

sed -n ‘9, /^uestc/p’ file3

打印file3中第9行到以uestc开头的行

sed -n ‘/[Cc]hina/p’ file4

打印file4中包含China或china的行

8.10 流编辑器 sed

2. 删除文件内容：d命令

sed ‘76d’ file5

删除file5中的第76行

sed ‘9,$d’ file6

删除file6中第8行以后的所有行

sed ‘/co*ool/d’ file7

删除file7中包含cool, coool, cooool, ……等等的行

8.10 流编辑器 sed

3. 替换文件内容: s命令

sed -n ‘s/beijing/shanghai/g’ table1

将table1中所有的beijing替换为shanghai

sed -n ‘s/^*uid/username/p’ ulist

将ulist中以零至多个空格开头后跟uid的字符串替换为username

4. 多次编辑: e命令

sed -e ’1,5d’ -e ‘s/good/bad/’ report

将report中的第1~5行删除, 同时将good替换为bad

8.10 流编辑器 sed

5. 添加行: a命令

```
sed  '/^operation/a\
this is an inserted line'  course
```

在course中的以operation开头的行后加入this is an inserted line一行。

- course中有多行以operation开头时会怎样?
- 不是另加一个新行, 而是在某行中加入字符串, 如何操作?

8.10 流编辑器 sed

特别说明

1. 不同的UNIX操作系统版本中，sed的格式和语法可能有少量的差异，使用时可参照联机手册(man命令)。

2. 教材中的例子大部分是在C_shell下进行的，在B_shell或K_shell下可能有少量的差异. 但这些差异只反映在shell命令中，而不会反映在编辑命令(表达式)中。

8.11 编程工具 awk

什么是awk

awk 是一种程序设计语言, 主要用来处理文本类数据并产生报表。

它执行时对输入数据(文件、标准输入或命令的输出）逐行进行扫描，匹配指定的模式，并执行指定的操作。

8.11 编程工具 awk

8.11.1 awk的基本格式

awk ‘pattern {action}’ filename

awk扫描filename中的每一行，对符合模式pattern的行执行操作action。

特例:

① awk ‘pattern’ filename

显示所有符合模式pattern的行

② awk ‘{action}’ filename

对所有行执行操作action

8.11 编程工具 awk

8.11.2 数据文件中记录和域的标识

	$1	$2	$3	$4	$5	
NR=1	Tom	Jones	4424	5/12/66	543354	NF=5
NR=2	Mary	Adams	5436	11/4/63	28765	NF=5
NR=3	Sally	Chang	1654	7/22/54	650000	NF=5
NR=4	Billy	Black	1683	9/23/44	336500	NF=5

$0

$0: 整个记录(整个行)

$1: 记录中的第一个域

$2: 记录中的第二个域 ….

NF: 记录中域的个数

NR: 输入数据中的记录号

8.11 编程工具 awk

8.11.3 应用实例

```
$cat  employees
Tom    Jones    4424     5/12/66       543354
Mary   Adams    5436     11/4/63       28765
Sally  Chang    1654     7/22/54       650000
Billy  Black    1683     9/23/44       336500
$awk   '/Mary/'    employees
Mary   Adams   5436     11/4/63      28765
$awk   '{print  $1}'    employees
Tom
Mary
Sally
Billy
$awk   '/Sally/  {print  $1, $2}'    employees
Sally   Chang
```

8.11 编程工具 awk

8.11.4 awk的输入重定向形式

1. 从其它命令输入

格式: command | awk ‘pattern’

command | awk ‘{action}’

command | awk ‘pattern {action}’

实例:

```
$who
zhanglan    tty01    Jan  12   18:36
yuexi       tty02    Jan  12   17:03
liuzhen     tty15    Jan  12   08:45
$who | awk ‘/tty01/ {print $1}’   (谁在1号终端上)
zhanglan
```

8.11 编程工具 awk

2. 从标准输入设备(键盘)输入

格式: awk ‘pattern {action}’

由于未指定输入数据来源, 缺省情况下从标准输入设备(键盘)读取数据. 键盘上每输入一行, awk就处理一行, 直到遇到^D为止.

例如:

```
$ awk ‘/aaa/ {print $0, NF}’
bbbb  bbbbb  bbbbb
aaaa  aaaaaa  aa  aaaaaa
aaaa  aaaaaa  aa  aaaaaa  4
xxx  xxxxx  yyyyyyy  xyz
Ctrl+D
$
```

查找包含字符串aaa的行,打印整行内容和这行包含的域数

(这是awk的输出行)

8.11.5 awk的格式化输出

1. print 函数

用于不需要复杂格式的简单输出。

例如:

```
$ ps -e | awk '/tty05/ {print "Terminal 05: " $4}'
```

(查看5号终端上的用户现在正在干什么)

```
Terminal 05:  sh
Terminal 05:  cc
Terminal 05:  find
$
```

2. printf 函数

高级格式化输出函数. 用法与C语言中的用法相同。

例如:

```
$awk  '{printf  "uname: %-8s ID: %6d\n", $1,
 $3}' employees
uname:   Tom                    ID:    4424
uname:   Mary                   ID:    5346
uname:   Sally                  ID:     1654
uname:   Billy                  ID:     1683
$
```

8.11.6 awk命令文件

格式:

```
awk -f awk_file data_file
```

当需要对输入数据中的一行执行多项操作时,常把这些操作命令放在一个命令文件awk_file中,而不是在命令行上发出.

awk运行时,对输入文件中的每一行执行命令文件中的所有操作后,再对下一行数据进行同样的处理过程,以此类推,直到输入文件中的最后一行。

8.11 编程工具 awk

应用实例:

```
$ cat  my_awk
/Sally/ {print  “**** found  Sally!  ****”}
{print  $1, $2, $3}
$ awk  -f  my_awk   employees
Tom  Jones     4424
Mary Adams    5436
**** found  Sally!  ****
Sally Chang      1654
Billy  Black      16
```

课后作业

自行练习shell命令和编程，并给出可以执行的程序，此次作业不用提交

- 1）编写程序列出当前登录用户的账号，并在有新用户登录时显示新用户的登录信息。
- 2）编写程序列出指定目录下文件大小大于指定大小的文件名及其完整信息。