Linux学习笔记

1. Linux常用命令

1.1 文件处理命令

1.1.1 命令格式与目录处理命令 ls

命令格式: 命令 [-选项] [参数]

```
ls -la /etc
```

> 多个选项可以合并 如 ls -l -a 可合并成 ls -la

命令英文原意：list

所在路径：/bin/ls

选项：

- -a --all 显示所有文件（包括隐藏文件）。
- -l 显示文件详细信息，长列表格式 。
- -F 在目录名字后加 / ，在可执行文件后加 * ，易于分辨。
- -R 递归选项，列出目录及其子目录的文件。
- -i 查看目录或者文件的inode号。

文件详细信息：

```
drwxr-xr-x 2 root root 4096 1月  13 16:12 an
```

- 文件类型，如目录（d）、文件（-）、字符型文件（c）或块设备（b）
- 文件权限
- 文件硬链接总数
- 文件属主的用户名
- 文件属组的组名
- 文件大小（单位字节）
- 文件上次修改时间
- 文件名或者目录名

1.1.2 创建文件 touch

```
$ touch test_one
$ ls -l test_one
-rw-r--r-- 1 root root 0 1月  17 10:33 test_one
```

touch 命令创建你指定的文件并将你的用户名作为文件属主，并且此时文件是空的。

touch 还可以用来改变文件修改时间，并且不会改变文件内容。

复制文件 cp

基本用法： *cp source destination*

source 和 destination 都是文件名时，cp命令将源文件复制为一个新文件并且以desitation命名。并且新文件有新的修改时间。

若目标文件已存在，cp命令会直接覆盖而不提醒，最好是加上 -i 选项，强制shell询问是否需要覆盖已有文件。

```
$ cp test_one test_two
$ ls -l test_*
-rw-r--r-- 1 root root 4 1月  17 10:36 test_one
-rw-r--r-- 1 root root 4 1月  17 10:54 test_two
$ cp -i test_one test_two
cp: overwrite 'test_two'? n
$
```

输入y则覆盖，不输入或者输入n则不覆盖。

-R/-r选项可以递归复制整个目录内容。

1.1.3 链接文件 ln

- **符号链接** 一个文件，指向另一个文件，彼此内容互不相同。使用 ln命令和-s选项创建
- **硬链接** 独立的虚拟文件，包含原始文件的信息和位置，但是根本上而言还是同一个文件啊。引用硬链接等同于引用源文件。直接使用ln创建。

  > 硬链接只能在同一硬盘，同一分区下

```
$ ls -l file1
-rw-r--r-- 1 root root   21 1月  17 11:14 file1
$ ln -s file1 sl_file1
$ ln file1 hl_file1
$ ls -l *file1
-rw-r--r-- 1 root root   21 1月  17 11:16 file1
-rw-r--r-- 2 root root   23 1月  17 11:16 hl_file1
lrwxrwxrwx 1 root root    5 1月  17 11:15 sl_file1 -> file1
```

我们可以查看两个文件的inode编号来确认他们之间的关系，文件或目录的inode编号是用于标识的唯一数字，可用 ls 的 -i 选项查看。

```
$ ls -i *file1
132081 file1  132081 hl_file1  133196 sl_file1
```

1.1.4 移动文件 mv

mv命令可以移动文件或者重命名（即将文件移动到当前目录），但是inode编号和时间戳保持不变。

使用 -i 选项也可以得到提示，与cp命令相同。

```
$ touch test1
$ ls -l
-rw-r--r-- 1 root root    0 1月  17 11:38 test1
$ mv test1 test2
$ ls -l
-rw-r--r-- 1 root root 0 1月  17 11:38 test2
```

1.1.5 删除文件 rm

删除文件一定要小心，一旦删除就无法找回。

选项：

- **-i**： 询问是否删除
- **-f**： --force 强制删除
- **-r**： 递归删除，常用于删除目录

rm -rf /* 就是删除系统根目录下所有文件。~~跑路必备~~

1.2 目录处理命令

1.2.1 创建目录 mkdir

选项：

- **-m** 配置文件权限
- **-p** 递归创建目录

```
$ mkdir New_Dir
$ ls -l
drwxr-xr-x 2 root root 4096 1月  17 12:05 New_Dir
$ mkdir dir1/dir2
mkdir: cannot create directory 'dir1/dir2'
No such file or diretory
$ mkdir -p dir1/dir2
```

1.2.2 删除目录 rmdir

rmdir [-p] dirName 只能删除空的目录dirName

- **-p** 是指子目录被删除后使它成为空目录的话也一并删除

1.3 查看文件内容

1.3.1 查看文件类型 file

```
$ file my_file
myfile: ASCII text #ASICC编码的文本文件
$ file New_Dir
New_Dir: directory #目录
$ file sl_file
sl_file: symbolic link to 'file' #软连接
```

1.3.2 查看文件内容

1. **cat命令**

   选项：

   - **-n** 给所有行加上行号
   - **-b** 给有内容的行加上行号
   - **-T** 将制表符替换为^I

> 当使用cat命令查看大型文件时，文件文本会在显示器上一晃而过，这时候我们可以使用more命令

2. **more命令**

more命令一次显示一页，可以通过空格等按键逐页查看，具体可以通过man more命令了解。浏览完毕后按q键退出。

3. **less命令**

more命令的升级版，来自于"less is more"，可以通过上下按键翻页。具体使用man less命令查看。

1.3.3 查看部分文件

1. **tail命令**

tail命令默认显示最后十行，可以通过 tail -n 2 fileName 或者tail -2 fileName控制，此命令只显示最后两行。

2. **head命令**

head命令显示文件开头的内容，与tail使用方法一致。

1.4 监测程序

1.4.1 探查进程 ps

Linux系统使用的 ps 命令支持三种类型的命令行参数

- Unix 风格的参数，前面加单破折线；
- BSD 风格的参数，前面不加破折线；
- GNU 风格的参数，前面加双破折线。

其中常用的选项包括：

- a 显示与终端关联的所有进程
- u 采用基于用户的格式显示
- x 显示没有控制终端的进程
- -l 长格式显示更加详细的进程
- -e 显示所有进程

常用命令：

- ps -aux 查看系统中所有进程，可以通过grep过滤自己想看到的
- ps -le 查看系统中的所有进程及其父进程PID和进程优先级

1.4.2 实时监测进程 top

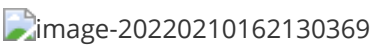

图4-2 top命令运行时的输出

第一行显示系统概况，包括当前时间，系统运行时间、登录用户数及系统平均负载（三个值分别表示最近1、5、15分钟的负载）。

第二行显示系统进程信息，分别为运行，休眠，停止和僵化（进程已完成，父进程没有响应）状态。

最后一部分：

- **PID**：进程ID。
- **USER**：进程所有者。
- **PR**：进程优先级。
- **NI**：进程谦让度。
- **VIRT**：进程占用的虚拟内存总量。
- **RES**：进程占用的物理内存总量
- **SHR**：进程和其他进程共享的内存总量。
- **S**：进程状态
    - D：可中断的休眠状态
    - R：运行状态
    - S：休眠状态
    - T：跟踪状态或停止状态
    - Z：僵化状态
- **%CPU**：进程使用的CPU时间比例。
- **%MEM**：进程使用的内存占可用内存比例。
- **TIME+**：自进程启动到目前为止的CPU时间总量。
- **COMMAND**：进程对应的命令行名称，也就是启动的程序名。

1.4.3 结束进程 kill与killall

信号	名称	描述
1	HUP	挂起
2	INT	中断
3	QUIT	结束运行
9	KILL	无条件终止
11	SEVG	段错误
15	TERM	尽可能终止
17	STOP	无条件停止运行，但不终止
18	TSTP	停止或者暂停，但继续在后台运行
19	CONT	在STOP或TSTP后恢复执行

表4-3 Linux进程信号

1. kill命令

kill命令通过进程ID（PID）给进程发信号，默认情况下发送TERM信号。发送信号需要你是进程所有者或者root用户。

如果没有关闭，可以使用 -s 选项发出其他信号

```
$ kill 3940
$ kill -s HUP 3940
```

kill没有任何输出，你可以使用ps或者top命令查看进程是否已经终止。

2. killall命令

killall命令支持通过进程名来结束进程，可以使用通配符。

```
# killall http*
```

结束所有以http开头的进程，使用root用户时需要小心，不要结束重要的系统进程从而破坏文件系统。

1.4.4 监测磁盘空间

1.挂载储存媒体

在使用新的储存媒体之前，需要把他放到虚拟目录下。这项工作称为**挂载**（mounting）。

1.mount命令

直接输入mount命令会显示当前系统上挂载的设备列表。包括四部分信息：

- 媒体设备文件名
- 媒体挂载到虚拟目录的挂载点
- 文件系统类型
- 已挂载媒体的访问状态

手动挂载需要root用户，基本命令为：

```
mount -t type device directory -o options
```

type参数指定了磁盘被格式化的文件系统类型。如果和Windows PC共用这些储存设备，通常使用一下文件系统类型

- vfat：Windows长文件系统。
- ntfs：Windows NT、XP、Vista以及Windows 7 中广泛使用的高级文件系统。
- iso96000：标准CD-ROM文件系统。

options选项之间用逗号隔开，常用选项：

- ro：只读。
- rw：读写。
- user：允许普通用户挂载。
- loop：挂载一个文件。

2. umount命令

卸载设备的命令是umount(没有n)，弹出一个设备前都需要先卸载该设备。

命令格式：

> umount [directory | device]

通过设备文件或者挂载点来指定需要卸载的设备。此时必须没有程序在使用设备上的文件。

3. df命令

查看所有已挂载磁盘的使用情况。

```
$ df
文件系统                                      1K-块      已用       可用 已用% 挂载点
/dev/mapper/user--data-vm--4072--disk--0  16337788  5359880  10122664   35% /
/dev/mapper/pve-vlab--software           131062788 93888296  30447224   76%
/opt/vlab
none                                           492        4       488    1% /dev
udev                                     115331888        0 115331888    0%
/dev/tty
tmpfs                                    115373808        0 115373808    0%
/dev/shm
tmpfs                                     23074764   254292  22820472    2% /run
tmpfs                                         5120        0       5120    0%
/run/lock
tmpfs                                     23074760        4  23074756    1%
/run/user/1000
tmpfs                                     23074760        0  23074760    0%
/run/user/0
```

df -h 将输出的磁盘空间采用用户易读的方式显示，如用M代替兆字节。

4. du命令

显示某个目录的磁盘使用情况。常用选项有：

- -c：显示列出文件的总大小。
- -h：按用户易读的方式输出大小。
- -s：显示每个输出参数的总计。

1.4.5 处理数据文件

1.排序数据 sort

> sort命令按照会话指定的默认语言的排序规则对文本文件进行行排序（默认升序）。

选项：

- -n 识别数字，从小到大排序。
- -M 识别三字符月份名。
- -b 排序时忽略起始空白。
- -c 不排序，检查输入数据是否已排序，未排序则报告。
- -d 仅考虑空白和字母。
- -f 忽略大小写。
- -g 按数值排序，数值当做浮点数。
- -o 输出到指定文件
- -r 反序排序

2. 搜索数据 grep

命令格式：grep [options] pattern [file]

在输入或者指定文件中查找包含匹配指定模式字符的行。可以使用Unix风格正则表达式匹配。

选项：

- -v 反向搜索，输出不匹配的行。
- -n 显示行号。
- -c 显示总行数。
- -e 指定多个匹配模式。

3. 压缩数据

gzip软件包含有以下工具：

- gzip：用来压缩文件。
- gzcat：用来查看压缩过的文本文件的内容。
- gunzip：用来解压文件。

4. 归档数据 tar

命令格式： tar function [options] object1 object2 ...

function参数定义tar命令应该做什么：

- -c --create 创建一个新的tar归档文件。
- -t --list 列出已有的归档文件。
- -x --extract 从已有的tar归档文件中提取文件。
- -C dir 切换到指定目录。
- -f file 输出结果到文件或者设备file。
- -p 保留所有文件权限。
- -v 在处理文件时显示文件。
- -z 将输出重定向给gzip命令来压缩内容。
- -x 从归档中解压文件。

```
$ touch a.c
$ tar -czvf test.tar.gz a.c
a.c
$ tar -tzvf test.tar.gz
-rw-r--r-- root/root    0 2022-01-18 15:39 a.c
$ tar -xzvf test.tar.gz #常用解压命令
a.c
```

1.5 Shell内建命令

1.5.1 外部命令

外部命令，有时也被称为文件系统命令是存在于bash shell之外的程序。通常位于/bin、/usr/bin、/sbin或/usr/sbin中。

ps就是一个外部命令，可以通过which和type命令找到它。

```
$ which ps
/bin/ps
$ type -a ps
ps is /bin/ps
```

当外部命令执行时，会创建出一个子进程。这种操作称为衍生（forking）。

1.5.2 内建命令

内建命令不需要使用子进程执行，它们已经和shell编译成一个整体。

cd和exit命令都内建于bash shell。可以使用type命令查看。

```
$ type cd
cd is a shell builtin
$ type exit
exit is a shell builtin
```

内建命令不需要衍生子进程，也不需要打开程序文件，它们的执行速度更快，效率也更高。

有些命令有多种实现，我们可以通过type的-a选项查看。

```
$ type -a pwd
pwd is a shell builtin
pwd is /bin/pwd
```

> 对于有多种实现的命令，要使用外部命令时直接指定对应文件就好了。如外部命令pwd可以直接输入/bin/pwd。

1.5.3 查看命令历史 history

直接输入history命令就可以查看最近使用的命令列表，通常历史记录中会包含最近的1000条命令。

> 可以通过修改名为HISTSIZE的环境变量来改变保存的命令数。

命令历史记录被保存在隐藏文件.bash_history中，它位于用户的主目录。bash命令的历史记录先被存放在内存中，当shell退出时才写入到历史文件。输入!num可以取出历史命令对应编号的命令并执行。

1.5.3 命令别名 alias

alias是shell的一个内建命令，允许你为常用的命令（及其参数）创建另一个名称。

使用alias命令及选项-p可以查看当前可用的别名。

```
$ alias -p
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
```

可以使用alias命令创建属于自己的别名。

```
$ alias li='ls -li'
$ li
272251 drwxr-xr-x 2 root root 4096 1月  18 20:42 an
155026 -rw-r--r-- 1 root root    0 1月  18 20:40 test
```

注意，因为命令别名属于内部命令，一个别名只在它被定义的shell进程中有效。

```
$ alias li='ls -li'
$ bash #重新启动一个shell进程
$ li
bash: li: command not found
$ exit
exit
```

2. Linux文件权限

2.1 理解文件权限

2.1.1 使用文件权限符

```
$ ls -l
drwxr-xr-x 2 root root 4096 1月  17 12:05 New_Dir
```

输出结果的第一个字段就是描述文件和目录权限的编码。第一个字符代表对象类型：

- - 代表文件
- d 代表目录
- l 代表链接
- c 代表字符型设备
- b 代表块设备
- n 代表网络设备

之后是三组三字符编码。每一组定义三种访问权限：

- r 代表对象可读
- w 代表文件可写
- x 代表文件可执行
- - 代表缺少该种权限

这三组权限分别对应对象的属主，属组和其他用户的权限。

2.1.2 默认文件权限

```
$ touch new_file
$ ls -l new_file
-rw-r--r-- 1 root root 0 1月  18 20:42 new_file
$ umask
0022
```

touch命令用默认权限创造文件，将全权限值减掉umask后三位值就是默认权限，上例中文件默认权限就是文件的全权限值666减去022即为644。

文件权限码一般用八进制表示，以下是可能遇到的组合：

表2.1.2 Linux文件权限码

权限	二进制值	八进制值	描述
---	000	0	没有任何权限
--x	001	1	只有执行权限
-w-	010	2	只有写入权限
-wx	011	3	有写入和执行权限
r--	100	4	只有读取权限
r-x	101	5	有读取和执行权限
rw-	110	6	有读取和写入权限
rwx	111	7	有全部权限

2.1.3 改变文件权限 chmod

命令格式： chmod options mode file

使用八进制模式或符号模式进行安全性设置。

```
$ chmod 760 new_file
$ ls -l new_file
-rwxrw---- 1 root root 0 1月  18 20:42 new_file
```

chmod [ugoa...] [[+-=]] [rwxXstugo...]

第一组字符限定权限作用对象：

- u代表用户
- g代表组
- o代表其他
- a代表所有

后面跟着的符号表示你是想在现有基础之上增加权限（+）、移除权限（-），还是将权限设置成后面的值。

第三个字符代表作用到设置上的权限，他比普通的rwx多，主要包括：

- X：如果对象是目录或者已经有执行权限，赋予执行权限。
- s：运行时重新设置UID或GID。
- t：保留文件或目录。
- u：将权限设置为和属主一样。
- g：将权限设置为和属组一样。
- o：将权限设置为和其他用户一样。

```
$ chmod o+r new_file
$ ls -l new_file
-rwxrw-r-- 1 root root 31 1月  18 20:57 new_file
```

3. Linux常用编辑器

3.1 vim编辑器

3.1.1 vim基础

vim有两种操作模式：

- 普通模式
- 插入模式

普通模式下的一些命令：

- PageDown（或Ctrl+F）：下翻一屏。
- PageUp（或Ctrl+B）：上翻一屏。
- G：移动到缓冲区最后一行。
- num G：移动到缓冲区第num行。
- gg：移动到缓冲区第一行。

在普通模式下按下冒号键进入命令行模式，左下角出现冒号并等待输入命令。

在命令行模式下有几个命令可以将缓冲区的数据保存到文件中并退出vim。

- q：未修改缓冲区数据则退出。
- q!：取消对缓冲区的修改并退出。
- w filename：将文件保存到另一个文件中。
- wq：将缓冲区数据保存到文件并退出。

3.1.2 编辑数据

普通模式下vim常用的编辑命令：

表1-2 vim常用编辑命令

命令	描述
x	删除当前光标所在位置的字符
dd	删除当前光标所在行
dw	删除光标当前所在位置的单词
d$	删除光标当前所在位置至行尾的内容
u	撤销前一编辑命令
a	在当前光标后追加数据

有些编辑命令运行使用数字修饰符来重复该命令，如命令5dd会删除光标当前所在行开始的5行。

3.1.3 复制和粘贴

剪切和粘贴相对容易一些，vim在删除数据时，实际上会将数据保存在单独的一个寄存器中。可以用p命令取回数据。

比如，在用dd命令删除一行文本后，将光标移动到缓冲区某个要放置该行文本的位置，再用p命令。

复制文本稍微复杂点，vim的复制命令是y（yank）。可以在y后面使用相同的第二字符（yw表示复制一个单词，y$表示复制到行尾）。再使用p命令粘贴。

按v可以进入可视模式，当你移动光标时会高亮显示文本。移动光标选中你需要复制的文本，按y键激活复制命令，再使用p（paste）命令来粘贴。

3.1.4 查找和替换

要查找一个字符串，就按下斜线（/）。光标会跑到消息行，再输入你要查找的文本并回车。编辑器将作出回应：

- 若要查找的文本出现在光标当前位置之后，则光标会调到该文本出现的第一个位置。
- 若当前查找的文本未在光标当前位置之后出现，则光标会绕过文件末尾，出现在该文本所在的第一个位置并说明。
- 输出一条错误消息，说明没有在文件中找到要查找的文本。

要继续查找同一个单词，按下斜线再按回车。或者使用n键，表示next。

替换命令允许你快速用一个单词替换文本中的单词，需要在命令行模式使用。命令格式为：

```
:s/old/new/
```

vim编辑器会调到该行old第一次出现的地方并用new来替换。还有一些替换命令：

- :s/old/new/g 命令替换该行所有old。
- :n,ms/old/new/g 替换行号n和m之间的所有old。
- :%s/old/new/g 替换整个文件所有的old。
- :%s/old/new/gc 替换整个文件所有old，但在每次出现时提示。

4. shell脚本基础

4.1 使用多个命令

要将两个命令同时运行，只需要把他们放在同一行中，用分号隔开。

```
$ date ; who
2022年 01月 19日 星期三 11:28:28 CST
ubuntu   tty7         2022-01-17 21:02 (:0)
root     pts/4        2022-01-19 10:23 (172.31.0.2)
```

4.2 创建shell脚本文件

创建shell脚本文件时，必须在第一行指定使用的shell。其格式为；

```
#!/bin/bash
```

在通常的shell脚本中，井号（#）用作注释行。但第一行是个例外，#后面的!会告诉shell用哪个shell来运行脚本。

下面将上节的例子写成一个shell脚本，保存在文件test中：

```
#!/bin/bash
# This script displays the date and who's logged in
date
who
```

但当你直接运行test时：

```
$ test
bash: test: command not found
```

shell会通过环境变量查找你的命令。PATH环境变量并不包含test所在的目录。我们可以通过绝对或相对文件路径来引用shell脚本文件，我们可以使用单点操作符。

```
$ ./test
-bash: ./test: Permission denied
$ ls -l test
-rw-r--r-- 1 root root   21 1月  19 11:44 test
```

还有一个问题，我们可以看到，所创建的shell脚本并没有执行权限，于是我们使用chmod命令赋予文件属主执行文件的权限，以后我们省略赋予权限的步骤，当然你也可以使用 `bash test` 的命令来执行该shell脚本文件。

```
$ chmod u+x test
$ ./test
2022年 01月 19日 星期三 11:44:50 CST
ubuntu   tty7         2022-01-17 21:02 (:0)
root     pts/4        2022-01-19 10:23 (172.31.0.2)
```

成功!

4.3 显示消息 echo

在echo命令后面加上一个字符串，就可以显示出来，默认情况下不需要将显示的文本字符串划定出来，但当文本里有引号时，就需要用另一种引号包起来（与python类似）。

```
$ echo This is a test
This is a test
$ echo Let's see if this'll work
Lets see if thisll work
$ echo "Let's see if this'll work"
Let's see if this'll work
```

我们将test脚本修改一下:

```
#!/bin/bash
# This script displays the date and who's logged on
echo The time and date are:
date
echo "Let's see who's logged into the system:"
who
```

运行这个脚本，将产生如下输出:

```
$ ./test
The time and date are:
2022年 01月 19日 星期三 12:06:51 CST
Let's see who's logged into the system:
ubuntu   tty7         2022-01-17 21:02 (:0)
root     pts/4        2022-01-19 10:23 (172.31.0.2)
```

如果想把文本字符串和命令输出显示在同一行中，只需要使用echo的-n参数。将第一个echo语句改成这样就行：

```
echo -n "The time and date are: "
```

你需要在字符串两侧使用引号。命令输出将会在紧接着字符串结束的地方出现。现在的输出结果：

```
$ ./test
The time and date are: 2022年 01月 19日 星期三 12:12:57 CST
Let's see who's logged into the system:
ubuntu   tty7         2022-01-17 21:02 (:0)
root     pts/4        2022-01-19 10:23 (172.31.0.2)
```

4.4 使用变量

4.4.1 环境变量

shell维护着一组环境变量，用来记录特定的系统信息。比如系统的名称，登录到系统上的用户等等。我们可以使用set命令来显示一份完整的当前环境变量列表。

```
$ set
BASH=/bin/bash
[...]
```

太多我们就不显示了。

在脚本中，你可以在环境变量前加上美元符($)来使用这些环境变量。如以下shell脚本：

```
#!/bin/bash
# display user information from the system
echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
```

执行该脚本，得到输出：

```
$ ./test
User info for userid: root
UID: 0
HOME: /root
```

我们再看以下一个例子：

```
$ echo "The cost of the item is $15"
The cost of the item is 5
```

这显然不是我们想要的输出，在这个例子中，脚本会尝试显示变量$1（未定义），再显示5。要显示美元符，我们需要在它前面放置一个反斜线（转义）。

```
$ echo "The cost of the item is \$15"
The cost of the item is $15
```

> 你可能见过通过${variable}形式引用的变量。变量名两侧额外的花括号通常用来帮助识别美元符后的变量名。

4.4.2 用户变量

shell脚本允许在脚本中定义和使用自己的变量。用户变量可以是任何由字母、数字或下划线组成的文本字符串，长度不超过20。并且用户变量区分大小写。

使用等号将值赋给用户变量。在变量、等号和值之间**不能出现空格**（非常令人困扰）。

shell脚本会自动决定变量值的数据类型。在脚本结束时会被删除掉。

```
#!/bin/bash
days=10
guest="Katie"
echo "$guest checked in $days days ago"
```

运行脚本有以下输出。

```
$ ./test
Katie checked in 10 days ago
```

引用一个变量时需要$，但是对一个变量赋值时不需要$

4.4.3 命令替换

shell脚本最有用的特性之一就是可以从命令输出中提取消息，并将其赋给变量。有两种方法可以让命令输出赋予给变量：

- 反引号字符(`)
- $( )格式

```
testing=`date`
testing=$(date)
```

shell命令运行命令替换符号中的命令，并将其赋值给变量testing。接下来是一个例子，它在脚本中通过命令替换获得当前日期并用他生成唯一文件名：

```
#!/bin/bash
# copy the /usr/bin directory to a log file
today=$(date +%y%m%d)
ls /usr/bin -al > log.$today
```

+%y%m%d格式告诉date命令将日期显示为两位数年月日的组合。

```
$ date +%y%m%d
220119
```

运行该脚本后，应该能在目录中看到一个新文件。

```
$ ls -l log.220119
-rw-r--r-- 1 root root 92516 1月  19 17:22 log.220119
```

4.5 重定向输入和输出

4.5.1 输出重定向 >

最基本的重定向命令将命令的输出发送到一个文件中。bash shell用大于号（>）来完成这项功能。

```
$ date > test2
cat test2
2022年 01月 19日 星期三 17:27:18 CST
$ who > test2
cat test2
ubuntu   tty7         2022-01-17 21:02 (:0)
root     pts/4        2022-01-19 17:22 (172.31.0.2)
```

有时候，你可能不想覆盖文件内容，而是想将命令输出追加到已有文件中，则可以用双大于号（>>）来追加数据。

```
$ date >> test2
$ cat test2
ubuntu   tty7         2022-01-17 21:02 (:0)
root     pts/4        2022-01-19 17:22 (172.31.0.2)
2022年 01月 19日 星期三 17:30:45 CST
```

4.5.2 输入重定向 <

输入重定向和输出重定向的方向相反。输入重定向将文件的内容重定向到命令。

输入重定向的符号是小于号（<）。

```
$ wc < test2
  3  16 139
```

wc命令对文本中的数据计数。默认情况下输出三个值：

- 文本行数。
- 文本词数。
- 文本字节数。

还有一种重定向方法叫做**内联输入重定向**(inline input redirection)，这种方法不需要文件进行重定向，只需要在命令行中指定用于输入重定向的数据就可以了。

内联输入重定向的符号是远小于号（<<）。除了这个符号，你必须指定一个文本标记来划分输入数据的开始和结尾。任何字符串都可以作为文本标记，但是数据的开始和结尾的文本标记必须一致。

```
$ wc << EOF
> test1
> test string 2
> test 3
> EOF
 3  6 27
```

4.6 管道 |

我们可以通过在两个命令之间加上单个竖线（|）将命令输出直接重定向到另一个命令，这个过程叫做**管道连接**（piping）。

```
command1 | command2
```

管道串起来的两个命令并不会依次执行。Linux系统实际上会同时运行两个命令，在系统内部将它们连接起来。在第一个命令产生输出的同时，输出会立即送给第二个命令。数据传输不会用到任何中间文件和缓冲区。

例如在Ubuntu系统中，我们使用apt list来列出已安装的包，由于显示的过快，我们根本来不及看到输出，所以我们可以使用文本分页命令来强行按屏显示。

```
$ apt list | more

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

Listing...
0ad-data-common/focal 0.0.23.1-1 all
0ad-data/focal 0.0.23.1-1 all
0ad/focal 0.0.23.1-4ubuntu3 amd64
0install-core/focal 2.15.1-1 amd64
0install/focal 2.15.1-1 amd64
0xffff/focal 0.8-1 amd64
2048-qt/focal 0.1.6-2build1 amd64
2ping/focal 4.3-1 all
2to3/focal 3.8.2-0ubuntu2 all
[...]
```

我们也可以搭配使用管道和重定向命令来将输出保存到文件中。

```
$ apt list | sort > apt.list
$ more apt.list
0ad-data-common/focal 0.0.23.1-1 all
0ad-data/focal 0.0.23.1-1 all
0ad/focal 0.0.23.1-4ubuntu3 amd64
0install-core/focal 2.15.1-1 amd64
0install/focal 2.15.1-1 amd64
0xffff/focal 0.8-1 amd64
2048-qt/focal 0.1.6-2build1 amd64
2ping/focal 4.3-1 all
2to3/focal 3.8.2-0ubuntu2 all
2vcard/focal 0.6-2 all
[...]
```

4.7 执行数学运算

4.7.1 expr命令

expr命令能够识别少数的数学和字符串操作符，主要包括简单的四则运算和字符串处理能力。

表4-7 expr命令操作符

操作符	描述
ARG1 | ARG2	如果ARG1既不是null也不是0，返回ARG1；否则返回ARG2
ARG1 & ARG2	如果没有参数是null或者0值，返回ARG1；否则返回0
ARG1 < ARG2	如果ARG1小于ARG2，返回1；否则返回0
ARG1 = ARG2	如果ARG1等于ARG2，返回1；否则返回0
ARG1 != ARG2	如果ARG1不等于ARG2，返回1；否则返回0
STRING : REGEXP	如果REGEXP匹配到STRING中的某个模式，返回该模式匹配
match STRING REGEXP	同上个
substr STRING POS LENGTH	返回起始位置为POS（从1开始），长度为LENGTH个字符的字符串
index STRING CHARS	返回在STRING中找到CHARS字符串的位置；未找到返回0
length STRING	返回字符串STRING的长度
+ TOKEN	将TOKEN解释成字符串，即使是关键字
(EXPRESSION)	返回EXPRESSION的值

表中还有一些简单的四则运算和比较没有列出，可以自己推出。

但有些操作符在shell中另有含义，在expr中会出现奇怪的结果，我们需要转义它们，如乘法运算。

```
$ expr 5 * 2
expr: syntax error
$ expr 5 \* 2
10
```

我们需要在*号前面加上反斜线\才能完成乘法运算。expr命令在shell脚本中也很复杂：

```
#!/bin/bash
# An example of using the expr command
var1=10
var2=20
var3=$(expr $var2 / $var1)
echo The result is $var3
```

我们运行这个脚本，可以得到：

```
$ chmod u+x test
$ ./test
The result is 2
```

4.7.2 使用方括号

我们可以使用美元符合方括号来将数学表达式围起来。即$[operation]

```
$ var1=$[1+5]
$ echo $var1
6
$ var2=$[$var1 * 2]
$ echo $var2
12
```

使用方括号计算公式时，不用担心shell误解乘号。*此时不是通配符，因为它在方括号内。

bash shell数学运算符只支持整数运算，在实际数学计算中，这是一个巨大的限制。

4.7.3 浮点数运算

有几种方案可以解决bash中数学运算的整数限制。最常见的是使用内建的bash运算器，叫做bc。

1. bc基本用法

bash计算器实际上是一种编程语言，它允许在命令行中输入浮点表达式，然后解释并计算该表达式。bash计算器能够识别：

- 数字（整数和浮点数）
- 变量（简单变量和数组）
- 注释（#或者C语言的/* */）
- 表达式
- 编程语句（如if-then）
- 函数

可以在shell提示符下通过bc命令直接访问bash计算器：

```
$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
100 * 0.123
12.300
```

要退出计算器可以输入quit。

可以通过设置变量scale来控制你需要保留的位数。

```
$ bc -q # quiet模式，不输出前面一大段话
3.44 / 5
0
scale=4
3.44 / 5
.6880
quit
```

bc还允许你通过print语句打印变量和数字。

2. 在脚本中使用bc

使用命令替换在shell脚本中运行bc命令，基本格式：

```
variable=$(echo "options; expression" | bc)
```

第一部分options允许你设置变量，使用分号隔开。

```
#!/bin/bash
var1=$(echo "scale=4; 3.44 / 5" | bc)
echo The answer is $var1
```

运行脚本：

```
$ chmod u+x test
$ ./test
The answer is .6880
```

对于比较复杂的运算，我们可以使用内联输入重定向来在命令行中重定向数据，仍然需要命令替换符号将bc命令的输出赋值给变量。如下shell脚本：

```
#!/bin/bash

var1=10.46
var2=43.67
var3=33.2
var4=47

var5=$(bc << EOF
scale = 4
a1 = ($var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
)

echo the final answer for this mass is $var5
```

运行脚本：

```
$ ./test
the final answer for this mass is 2017.1882
```

4.8 退出脚本

4.8.1 查看退出状态码

Linux有一个专门的变量$?来查看上个已执行命令的退出状态码。如下：

```
$ date
2022年 01月 19日 星期三 21:48:29 CST`
$ echo $?
0
$ assd
assd: command not found
$ echo $?
127
```

表4-8 Linux退出状态码

状态码	描述
0	命令成功结束
1	一般未知性错误
2	不适合的shell命令
126	命令不可执行
127	没找到命令
128	无效的退出参数
130	通过Ctrl+C终止的命令
255	正常范围之外的退出状态码

4.8.2 exit命令

默认情况下，shell脚本以脚本中的最后一个命令的退出状态码退出。但是通过exit返回自己的退出状态码。

```
#!/bin/bash
var1=10
var2=20
var3=$[$var1 + $var2]
echo The answer is $var3
exit 5
```

执行脚本并查看脚本退出码：

```
$ chmod u+x test
$ ./test
The answer is 30
$ echo $?
5
```

但注意退出状态码不能超过最大值255，当超过255时，会取256的余数。如 exit 300 则实际退出状态码为除以256的余数44。

5. 使用结构化命令

5.1 if-then语句

基本格式：

```
if command
then
    commands
fi
```

bash shell中的if语句会运行if后面的命令，如果命令成功运行（退出状态码为0），则位于then部分的命令就会被执行。如果if后面的退出状态码不是0，则then部分命令不会被执行。fi表示if then语句到此结束。下面是一个简单的例子：

```
#!/bin/bash
# testing the if statement
if pwd
then
    echo "It worked"
fi
```

这个脚本if行采用了pwd命令，如果命令成功执行，echo语句将会显示该行文本。运行该脚本（赋予执行权限不再写出），我们得到以下结果：

```
./test1.sh
/root/an
It worked
```

下面是另外一个例子：

```
$ cat test2.sh
#!/bin/bash
# testing a bad command
if adsba
then
    echo "It worked"
fi
echo "We are outside the if statement"
$ ./test2.sh
./test2/sh: line 3: adsba: command not found
We are outside the if statement
```

在这个例子中，我们在if中放了一个不能执行的命令。由于这是一个错误的命令，所以它会产生非零的退出状态码，且bash shell跳过then部分的echo语句。但是那个错误命令所生成的错误消息仍然会输出，后面我们会讨论如何避免它。

> if then 语句还有另一种形式：
>
> ```
> if command; then
>     commands
> fi
> ```
>
> 这样更像其它语言的if-then语句

5.2 if-then-else 语句

if-then-else语句格式：

```
if command
then
    commands
else
    commands
fi
```

当if语句的命令返回退出状态码0时，then部分中的命令会被执行，这跟普通的if-then语句一样。当if语句中的命令返回非零退出状态码时，bash shell会执行else部分的命令。下面我们看一个例子：

```
$ cat test3.sh
#!/bin/bash
# testing the else section
testuser=NoSuchUser

if grep $testuser /etc/passwd
then
    echo "The bash files for user $testuser are:"
    ls -a /home/$testuser/.b*
else
    echo "The user $testuser does not exist on this system."
fi
$ ./test3.sh
The user NoSuchUser does not exist on this system.
```

5.3 elif语句

命令格式：

```
if command1
then
    commands
elif command2
then
    commands
else command3
then
    commands
fi
```

每块命令都根据是否返回退出状态码0来执行。bash shell 会依次执行if语句，第一个返回退出状态码0的语句将会被执行。

5.4 test命令

命令格式：

```
if test condition
then
    commands
fi
```

如果test命令中列出的条件成立，就会退出并返回状态码0；如果不成立，test命令退出并返回非0的状态码，这使得if-then语句不再被执行；如果condition部分为空，则test以非零的退出状态码退出，并执行else语句。

```
$ cat test4.sh
#!/bin/bash
# Testing the test command
if test
then
    echo "No expression returns a True"
else
    echo "No expression returns a False"
fi
$ ./test4.sh
No expression returns a False
```

当你加入一个条件，test会测试该条件。如，利用test命令确定变量是否有内容。

```
$ cat test5.sh
#!/bin/bash
# Testing the test  command
my_variable="Full"
if test $my_variable
then
    echo "The $my_variable expression returns a True"
else
    echo "The $my_variable expression returns a False"
fi
$ ./test5.sh
The Full expression returns a True
```

bash shell 提供了另一种方法，无需在if-then语句中声名test命令。

```
if [ codition ]
then
    commands
fi
```

方括号中定义了测试条件。注意，方括号内开头和结尾都必须有一个空格，否则会报错。

test命令可以判断三类条件：

- 数值比较
- 字符串比较
- 文件比较

5.4.1 数值比较

test命令最常见的是对两个数值进行比较。下表列出了测试两个值时可使用的参数。

表5-4 test命令的数值比较功能

参数	描述
n1 -eq n2	检查n1是否与n2相等
n1 -ge n2	检查n1是否大于等于n2
n1 -gt n2	检查n1是否大于n2
n1 -le n2	检查n1是否小于等于n2
n1 -lt n2	检查n1是否小于n2
n1 -ne n2	检查n1是否不等于n2

我们举个例子：

```
$ cat numeric_test.sh
#!/bin/bash
# Using numeric test evaluations
value1=10
value2=11
#
if [ $value1 -gt 5 ]
then
    echo "The test value $value1 is greater than 5"
fi
#
if [ $value1 -eq $value2 ]
then
    echo "The values are equal"
else
    echo "The values are different"
fi
$ ./numeric_test.sh
The test value 10 is greater than 5
The values are different
```

第一个条件测试value1值是否大于5。第二个条件测试value1与value2是否相等。两个结果与预期一致。

5.4.2 字符串比较

条件测试允许比较字符串值。下表列出了可用的字符串比较功能：

表5-4 字符串比较

比较	描述
str1 = str2	检查str1是否和str2相同
str1 != str2	检查str1是否和str2不同
str1 < str2	检查str1是否小于str2
str1 > str2	检查str1是否大于str2
-n str1	检查str1的长度是否非零
-z str1	检查str1的长度是否为0

1. 字符串相等

注意，比较字符串是否相等时，会将所有标点和大小写都考虑进去。

```
$ cat test6.sh
#!/bin/bash
# testing string equality
testuser=root
if [ $USER = $testuser ]
then
    echo "Welcome $testuser"
else
    echo "This is not $testuser"
fi
$ ./test6.sh
Welocome root
```

2. 字符串顺序

在使用测试条件的大于或者小于功能时，就会出现两个常见的问题：

- 大于号和小于号必须转义，否则shell会把它们当做重定向符号，把字符串当做文件名。
- 大于和小于顺序和sort命令所采用的不同。

接下来我们先看第一个问题。

```
$ cat badtest.sh
#!/bin/bash
# mis-using string comparisons
val1=baseball
val2=honey
if [ $val1 > $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$ ./badtest.sh
baseball is greater than honey
$ ls -l honey
-rw-r--r-- 1 root root 0 1月  20 16:16 honey
```

这个脚本中只使用了大于号，没有出现错误，但结果是错的。脚本把大于号解释成了输出重定向。因此，它创建了一个名为honey的文件。由于重定向顺利完成，test命令返回退出状态码0，从而then中的语句被执行。

要解决这个问题，我们需要正确转义大于号。

```
$ cat badtest.sh
#!/bin/bash
# mis-using string comparisons
val1=baseball
val2=honey
if [ $val1 \> $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$ ./badtest.sh
baseball is less than honey
```

现在的结果符合预期。

第二个问题是由于sort命令与test命令处理大小写方式相反，我们来看一个例子。

```
$ cat test7.sh
#!/bin/bash
# testing string sort order
val1=Testing
val2=testing
if [ $val1 \> $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$ ./test7.sh
Testing is less than testing
$ echo Testing > testfile
$ echo testing >> testfile
$ sort testfile
testing
Testing
```

在比较测试中，大写字母被认为是小于小写字母的。但sort命令恰好相反。这是不同的命令使用的排序技术不同造成的。

3. 字符串大小

-n 和 -z 可以检查一个变量是否含有数据。

```
if [ -n $val1 ] #判断变量长度是否非零
if [ -z $val2 ] #判断变量长度是否为零
```

5.4.3 文件比较

shell 允许你测试Linux文件系统上文件和目录的状态。下表列出了这些比较：

比较	描述
-d file	检查file是否存在并是一个目录
-e file	检查file是否存在
-f file	检查file是否存在并是一个文件
-r file	检查file是否存在并可读
-s file	检查file是否存在并非空
-w file	检查file是否存在并可写
-x file	检查file是否存在并可执行
-O file	检查file是否存在并属当前用户所有
-G file	检查file是否存在并且默认组与当前用户相同
file1 -nt file2	检查file1是否比file2新
file1 -ot file2	检查file1是否比file2旧

1. 检查目录

-d 测试会检查指定的目录是否存在于系统中。如果你打算将文件写入目录或是准备切换到某个目录，先进行测试总是件好事。

```
$ cat test8.sh
#!/bin/bash
# Look before you leap
jump_directory=/home/ubuntu
if [ -d $jump_directory ]
then
    echo "The $jump_directory directory exists"
    cd $jump_directory
    ls
else
    echo "The $jump_directory does not exist"
fi
$ ./test8.sh
The /home/ubuntu directory exists
公共的  模板  视频  图片  文档  下载  音乐  桌面  test.c
```

上例使用了-d测试条件来检查jump_directory变量的目录是否存在。若存在，则使用cd命令切换到该目录并列出目录中的内容；若不存在，则输出一条警告信息再退出。

2. 检查对象是否存在

-e 允许你的脚本在使用文件或目录前检查它们是否存在。

```
$ cat test9.sh
#!/bin/bash
```

```
# Check if either a directory or file exists
location=$HOME
file_name="sentinel"
if [ -e $location ]
then #directory does exist
    echo "OK on the $location directory."
    echo "Now checking on the file, $file_name."
    #
    if [ -e $location/$file_name ]
    then # file does exists
        echo "OK on the filename"
        echo "Updating Current Date..."
        date >> $location/$file_name
    else # file does not exist
        echo "File does not exist"
        echo "Nothing to update"
    fi
else # Directory does not exist
    echo "The $location directory does not exist."
    echo "Nothing to update"
fi
$ ./test9.sh
OK on the /root directory.
Now checking on the file, sentinel.
File does not exist
Nothing to update
$ touch sentinel
$ ./test9.sh
OK on the /root directory.
Now checking on the file, sentinel.
OK on the filename
Updating Current Date...
$ cat sentinel
2022年 01月 21日 星期五 11:20:43 CST
```

上例中我们使用-e判断用户是否有$HOME目录。再判断在其中有没有sentinel文件，再追加写入当前的日期和时间。

3. 检查文件

-f 检查文件是否存在。

4. 检查是否可读

在尝试从文件中读取数据之前，最好先测试一下文件是否可读。可用-r比较测试。

```
$ cat test10.sh
#!/bin/bash
# testing if you can read a file
pwfile=/etc/shadow
#
# first, test if the file exists, and is a file
if [ -f $pwfile ]
then
    # now test if you can read it
    if [ -r $pwfile ]
    then
        tail $pwfile
```

```
    else
        echo "Sorry, I am unable to read the $pwfile file"
    fi
else
    echo "Sorry, the file $pwfile does not exist"
fi
$ ./test10.sh
Sorry, I am unable to read the /etc/shadow file
```

/etc/shadow 含有系统用户加密后的密码，所以它对系统上的普通用户来说是不可读的。-r测试失败从而执行else部分。

5. 检查空文件

使用-s命令检查文件是否为空，尤其在不想删除非空文件的时候。当文件中有数据时，-s才比较成功。

```
$ cat test11.sh
#!/bin/bash
# Testing if a file is empty
file_name=$HOME/sentinel
if [ -f $file_name ]
then
    if [ -s $file_name ]
    then
        echo "The $file_name file exists and has data in it."
        echo "Will not remove this file."
    else
        echo "The $file_name file exists, but it is empty."
    fi
else
    echo "File $file_name not exist"
fi
$ ls -l $HOME/sentinel
-rw-r--r-- 1 root root 43 1月  21 11:20 /root/sentinel
$ ./test11.sh
The /root/sentinel file exists and has data in it.
Will not remove this file.
$ > $HOME/sentinel
$ ./test11.sh
The /root/sentinel file exists, but it is empty.
Deleting empty file...
```

剩下几种文件测试就不一一赘述了。

5. 5 复合条件测试

if-then语句允许你使用布尔逻辑来组合测试。有两种布尔运算符可以用：

- [condition1] && [condition2]
- [condition1] || [condition2]

第一种布尔运算使用AND布尔运算符来组合两个条件。then部分命令要执行，两个条件都需要满足。

> 布尔逻辑是一种能够将可能的返回值简化为TRUE或FALSE的方法。

第二组布尔运算使用OR布尔运算符来组合两个条件。如果任意条件为TRUE，then部分命令就会执行。

```
$ cat test12.sh
#!/bin/bash
# testing compound comparisons
if [ -d $HOME ] && [ -w $HOME/testing ]
then
    echo "The file exists and you can write to it"
else
    echo "You can not write to the file"
fi
$ ./test12.sh
You can not write to the file
$ touch $HOME/testing
$ ./test12.sh
The file exists and you can write to it
```

使用AND布尔运算符时，两个条件必须都满足。第一个比较$HOME目录是否存在，第二个检查用户是否对$HOME/testing文件有写入权限。两个都通过时就执行then部分命令。

5.6 if-then语句高级特性

bash shell提供了两项可在if-then语句中使用的高级特性：

- 用于数学表达式的双括号
- 用于高级字符串处理的双方括号

5.6.1 使用双括号

双括号命令允许你在比较过程中使用高级数学表达式。test命令只能在比较中使用简单的算术操作。双括号命令的格式如下：

```
((expression))
```

`expression` 可以是任意的数学赋值或者比较表达式。除了test命令使用的标准数学运算符，下表列出了双括号命令中会使用的其它运算符。

表5-4 双括号命令运算符

符号	描述
val++	后增
val--	后减
++val	先增
--val	先减
!	逻辑求反
~	位求反
**	幂运算
<<	左位移
>>	右位移
&	位布尔和
|	位布尔或
&&	逻辑和
||	逻辑或

可以在if语句中使用双括号命令，也可以在脚本普通命令中使用来赋值：

```
$ cat test13.sh
#!/bin/bash
# using double parenthesis
val1=10
if (( $val1 ** 2 > 90 ))
then
    (( val2 = $val1 ** 2 ))
    echo "The square of $val1 is $val2"
fi
$ ./test13.sh
The square of 10 is 100
```

注意，不需要转义双括号表达式的大于号。这是双括号表达式提供的另一个高级特性。

5.6.2 使用双方括号

双方括号命令提供了针对字符串比较的高级特性。双方括号命令格式如下：

```
[[ expression ]]
```

双方括号里的expression使用了test命令中的标准字符串比较。但它提供了test命令未提供的另一个特性——**模式匹配**（pattern matching）

> 双方括号在bash shell中工作良好，但并不是所有的shell都支持双方括号

在模式匹配中，可以定义一个正则表达式来匹配字符串。

```
$ cat test14.sh
#!/bin/bash
# using pattern matching
if [[ $USER == r* ]]
then
    echo "Hello $USER"
else
    echo "Sorry, I do not know you"
fi
$ ./test14.sh
Hello root
```

在上面的脚本中，我们使用了双等号（==）。双等号将右边的字符串（r*）视为一个模式，并用相应模式匹配规则。双方括号命令$USER环境变量进行匹配，看它是否以字母r开头。如果是则执行then中的指令。

5.7 case命令

case命令会采用列表格式来检查单个变量的多个值。以下是case命令的格式：

```
case variable in
pattern1 | pattern2) command1;;
pattern3) command2;;
*) default commands;;
esac
```

case命令会将指定的变量与不同模式进行比较。如果变量和模式是匹配的，那么shell会执行为该模式匹配的命令。可以通过竖线操作符在一行中分割出多个模式模式。星号会捕获所有与已知模式不匹配的值。

```
$ cat test15.sh
#!/bin/bash
# using the case command
case $USER in
ubuntu | root)
    echo "Welcome, $USER"
    echo "Please enjoy your visit";;
testing)
    echo "Special testing account";;
an)
    echo "Don't forget to log off when you're down";;
*)
    echo "Sorry, you are not allowed here";;
esac
$ ./test15.sh
Welcome, root
Please enjoy your visit
```

case命令提供了一个更加清晰的方法来为变量每个可能的值提供不同的选项。

6. 更多的结构化命令

6.1 for 命令

bash shell 提供了for命令，允许你创建一个遍历一系列值的循环。每次迭代都使用其中一个值来执行已定义好的一组命令。下面是bash shell中for命令的基本格式，

```
for var in list
do
    commands
done
```

> 只要你愿意，也可以将do语句和for语句放在同一行，但必须用分号将其同列表中的值分开：for var in list; do。

6.1.1 读取列表中的值

for 命令最基本的用法就是遍历for命令自身所定义的一系列值。

```
$ cat test1.sh
#!/bin/bash
# basic for command for
for test in Alabama Alaska Arizona Arkansas California
do
    echo The next state is $test
done
$ ./test1
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
```

每次for遍历值列表，它都会将列表下个值赋给$test变量，$test变量与其他脚本语句的变量一样使用。在最后一次迭代后，$test变量的值仍然在shell脚本的剩余部分保持有效。

```
$ cat test2.sh
#!/bin/bash
# testing the for variable after looping

for test in Alabama Alaska Arizona Arkansas California
do
    echo "The next state is $test"
done
echo "the last state is $test"
$ ./test2.sh
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
the last state is California
```

6.1.2 读取列表中的复杂值

下面是一个有常见问题的例子：

```
$ cat test3.sh
#!/bin/bash
# another example of how to use the for command
for test in I don't know if this'll work
do
    echo "word:$test"
done
$ ./test3.sh
word:I
word:dont know if thisll
word:work
```

shell 看到了列表值中的单引号并且尝试使用他们来定义一个单独的数据值。有两种办法可以解决这个问题：

- 使用转义字符（反斜线）将单引号转义；
- 使用双引号来定义引用到单引号的值。

这两种解决方法都能解决这个问题。

```
$ cat test3.sh
#!/bin/bash
# another example of how to use the for command
for test in I don\'t know if "this'll" work
do
    echo "word:$test"
done
$ ./test3.sh
word:I
word:don't
word:know
word:if
word:this'll
word:work
```

我们在第一个有问题的地方添加反斜线来转义don't中的单引号。在第二个有问题的地方this'll使用了双引号圈起来。两种方法都可以正常识别出这个值。

另一个可能的问题是有多个词的值，由于for循环假定每个值都用空格分割，如果有包含空格的数据值就会出现问题。

```
$ cat badtest.sh
#~/bin/bash
# an example of how not to use the for command
for test in California Neveda New York
do
    echo "Now going to $test"
done
$ ./badtest.sh
Now going to California
Now going to Neveda
Now going to New
Now going to York
```

这不是我们想要的结果。for命令使用空格来划分列表的每个值。如果在单独的值中有空格，就必须使用双引号把他们圈起来。

```
$ cat test4.sh
#~/bin/bash
# an example of how not to use the for command
for test in California Neveda "New York"
do
    echo "Now going to $test"
done
$ ./test4.sh
Now going to California
Now going to Neveda
Now going to New York
```

现在for命令可以正确区分不同值了。另外注意，在某个值两边使用双引号时，shell不会讲双引号当做值的一部分。

6.1.3 从变量读取列表

我们可以将一系列值储存在一个变量中，然后遍历变量中的整个列表，通过for命令来完成。

```
$ cat test5.sh
#!/bin/bash
# using a variable to hold the list
list="Liu Li Guan Liang Wang"
list=$list" Hu"
for name in $list
do
    echo "Now the person is $name"
done
$ ./test5.sh
Now the person is Liu
Now the person is Li
Now the person is Guan
Now the person is Liang
Now the person is Wang
Now the person is Hu
```

$list变量包含了用于迭代的标准文本值列表。我们还使用了另一个赋值语句向$list变量包含的已有列表中添加（或者说是拼接）了一个值。这是向变量中储存的已有文本字符串尾部添加文本的常用方法。

6.1.4 从文本中读取值

生成列表中所需值的另外一个途径是使用命令的输出。可以用命令替换来执行任何能产生输出的命令，然后在for命令中使用该命令的输出。

```
$ cat test6.sh
#!/bin/bash
# reading values from a file
file="names"
for name in $(cat $file)
do
    echo "I Know $name"
done
$ cat names
Liu Zi an
Li
Hu
$ ./test6.sh
I Know Liu
I Know Zi
I Know an
I Know Li
I Know Hu
```

这个例子在命令替换中使用cat命令来输出文件names内容。你注意到names文件中每一行有一个名字，而不是通过空格分割的。for命令仍然以每次一行的方式遍历了cat命令的输出。但是这并未解决数据中有空格的问题。

> 引用文件时如果脚本和文件位于同一目录下，可以直接使用文件名，若不是，则需要使用全路径名来引用文件。

6.1.5 更改字符串分隔符

造成这个问题的原因是特殊的环境变量 IFS ，叫做内部字段分隔符（internal field separator）。IFS环境变量定义了bash shell用作字段分隔符的一系列字符。默认情况下bash shell将以下字符当做分隔符：

- 空格
- 制表符
- 换行符

我们可以在脚本中临时更改IFS环境变量的值来限制被bash shell当做字段分隔符的字符。如你想修改IFS的值，使其只能识别换行符，就必须怎么做：

```
IFS=$'\n'
```

对前一个脚本使用这种方法则获得如下输出：

```
$ cat test6.sh
#!/bin/bash
# reading values from a file
IFS=$'\n'
file="names"
for name in $(cat $file)
do
    echo "I Know $name"
done
$ ./test6.sh
I Know Liu Zi an
I Know Li
I Know Hu
```

现在，shell脚本可以处理列表中含有空格的值了。

> 在处理代码量较大的脚本时，我们可能需要在一个地方修改IFS的值，然后忽略这次修改，在脚本其它地方继续使用IFS的默认值，我们可以这样做：
>
> IFS.OLD=$IFS
>
> IFS=$'\n'
>
> <代码>
>
> IFS=$IFS.OLD

如果需要指定多个IFS字符，可以将它们直接串起来：

```
IFS=$'\n':;"
```

这个赋值会将换行符、冒号、分号和双引号作为字符分隔符。

6.1.6 用通配符读取目录

最后，可以用for命令来自动遍历目录中的文件。进行此操作时，必须在文件名或者路径名中使用通配符。它会强制shell使用**文件扩展匹配**。文件扩展匹配生成匹配制定通配符的文件名或者路径名的过程。

```
$ cat test7.sh
#!/bin/bash
# iterate through all the files in a directory
for file in /home/*
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
$ ./test7.sh
/home/dailyHealth.py is a file
/home/daily.sh is a file
/home/geckodriver.log is a file
/home/mcl is a directory
/home/output is a file
```

```
/home/ubuntu is a directory
```

for命令会遍历/home/*输出的结果。注意，我们在if语句中做了一些处理：

```
if [ -d "$file" ]
```

在Linux中，目录名和文件名包含空格当然是合法的。要适应这种情况，应该将$file使用双引号圈起来，否则遇到有空格的文件名或目录名会产生错误：

```
./test7.sh: line6: [: too many arguments
./test7.sh: line9: [: too many arguments
```

在test命令中，bash shell将另外的单词当做参数从而报错。

也可以列出多个目录通配符，将目录查找和列表合并成一个for语句。

```
$ cat test8.sh
#!/bin/bash
# iterating through multiple directories

for file in /home/ubuntu/.b* /home/ubuntu/output
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
$ ./test8.sh
/home/ubuntu/.bash_history is a file
/home/ubuntu/.bashrc is a file
/home/ubuntu/output is a file
```

6.2 C语言风格的for命令

6.2.1 C语言for命令

bsah shell 支持一种与C语言风格类似的for循环，也有一些细微的不同，以下是bash中C语言风格的for循环的基本格式。

```
for ((variable assignment; condition; iteration process))
for (( a = 1; a < 10; a++))
```

注意，C语言风格的for命令有一些不遵循bash shell的标准for命令：

- 变量赋值可以有空格
- 条件中的变量不以美元开头
- 迭代过程的算式未用expr命令格式

下面是一个例子：

```
$ cat test9.sh
#!/bin/bash
# testing the C-style for loop
for (( i = 1; i < 6; i++ ))
do
    echo "The number is $i"
done
$ ./test9.sh
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
```

6.2.2 使用多个变量

C语言风格的for命令允许为迭代使用多个变量。循环会单独处理每个变量，但只能在for循环中定义一种条件。

```
$ cat test10.sh
#!/bin/bash
# multiple variable
for (( a=1, b=1; a <= 5; a++, b-- ))
do
    echo "$a - $b = $[ $a-$b ]"
done
$ ./test10.sh
1 - 1 = 0
2 - 0 = 2
3 - -1 = 4
4 - -2 = 6
5 - -3 = 8
```

6.3 while命令

6.3.1 while命令的基本格式

while命令的格式是：

```
while test command
do
    other commands
done
```

while命令中的test command和if-then语句中的格式一样，可以使用bash shell命令或者test条件测试。

注意，使用while循环一定要保证循环能够结束，不要造成死循环。

```
$ cat test11.sh
#!/bin/bash
# while command test
var1=10
while [ $var1 -gt 5 ]
do
```

```
    echo $var1
    var1=$[ $var1 - 1 ]
done
$ ./test11.sh
10
9
8
7
6
```

6.3.2 使用多个测试命令

while命令允许你咋while语句行定义多个测试命令。只有最后一个命令的退出状态码会被用来决定什么时候结束循环。下面我们看一个例子：

```
$ cat test12.sh
#!/bin/bash
# testing a multicommand while loop

var1=5

while echo $var1
    [ $var1 -ge 0 ] #写两行，不然死循环，亲身尝试,应该是由于
do
    echo "This is inside the loop"
    var1=$[ $var1 - 1 ]
done
$ ./test12.sh
5
This is inside the loop
4
This is inside the loop
3
This is inside the loop
2
This is inside the loop
1
This is inside the loop
0
This is inside the loop
-1
```

我们可以看到，while语句中定义了两个测试命令。第一个简单的显示了var1变量的当前值，第二个测试使用方括号来判断var1的值。在循环内部，echo语句显示一条简单的消息说明循环被执行。注意结束时输出的 -1 。这说明在最后一次循环时（var1此时为0），var1值变成-1后再次进入测试命令从而显示-1。

6.4 until命令

until命令和while命令工作方式完全相反。until命令要求你指定一个通常返回非0退出状态码的测试命令。只有测试命令的退出状态码不为0，bash shell才执行循环中列出的命令。一旦测试命令返回退出状态码0，循环就结束了。

```
until test commands
do
    other commands
done
```

和while命令类似，你可以在until命令语句中放入多个测试命令。只有最后一个命令的退出状态码决定了bash shell是否执行已定义的other commands。

下面是一个例子：

```
$ cat test13.sh
#!/bin/bash
# using the until command

var1=100
until [ $var1 -eq 0 ]
do
    echo $var1
    var1=$[ $var1 - 25 ]
done
$ ./test13.sh
100
75
50
25
```

6.5 嵌套循环

循环语句中还可以使用循环命令，这种循环叫做**嵌套循环**（nested loop）。

这里有个在for循环中嵌套for循环的简单例子。

```
$ cat test14.sh
#!/bin/bash
# nesting for loops

for (( a=1; a<=3; a++ ))
do
    echo "Starting loop $a:"
    for (( b=1; b<=3; b++ ))
    do
        echo " Inside loop: $b"
    done
done
$ ./test14.sh
Starting loop 1:
 Inside loop: 1
 Inside loop: 2
 Inside loop: 3
Starting loop 2:
 Inside loop: 1
 Inside loop: 2
 Inside loop: 3
Starting loop 3:
 Inside loop: 1
 Inside loop: 2
```

```
  Inside loop: 3
```

shell能够区分两个循环的do和done命令。

6.6 循环处理文件数据

通常需要遍历储存在文件中的数据。我们需要结合学过的两种技术：

- 使用嵌套循环
- 修改IFS环境变量

修改IFS环境变量可以强制for命令将文件中的每行都当成单独的一个条目处理，即使数据中有空格。

典型的例子是处理/etc/passwd文件中的数据。要求你逐行遍历/ect/passwd文件，并将IFS值改为冒号，这样就能分隔开每行的各个数据段了。

```
$ cat test15.sh
#!/bin/bash
# changing the IFS value
IFS.OLD=$IFS
IFS=$'\n'
for entry in $(cat /etc/passwd)
do
    echo "Values in $entry -"
    IFS=:
    for value in $entry
    do
        echo "  $value"
    done
done
$ ./test15.sh
Values in root:x:0:0:root:/root:/bin/bash -
  root
  x
  0
  0
  root
  /root
  /bin/bash
Values in daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin -
  daemon
  x
  1
  1
  daemon
  /usr/sbin
  /usr/sbin/nologin
```

这种方法在处理外部导入电子表格所采用的逗号分隔的数据时也很方便。

6.7 控制循环

有两个命令能帮我们控制循环内部的情况：

- break命令
- continue命令

6.7.1 break命令

break命令是退出循环的一个简单方法。可以使用break命令来退出任意类型的循环，包括while和until循环。

1. 跳出单个循环

在shell执行break命令时，他会尝试跳出当前正在执行的循环。

```
cat test17.sh
#!/bin/bash
# breaking out of a for loop

for var1 in 1 2 3 4 5 6 7 8
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Iteration number: $var1"
done
echo "The for loop is completed"
$ ./test17.sh
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
The for loop is completed
```

for 循环通常会遍历列表中指定的所有值，但当满足 if-then 条件时，shell执行break语句命令，停止for循环。

这种方法同样适用于while和until循环。

2. 跳出内部循环

在处理多个循环时，break命令会自动终止你所在的最内层的循环。

```
$ cat test18.sh
#!/bin/bash
# breaking out of an inner loop
for (( a=1; a<4; a++ ))
do
    echo "Outer loop: $a"
    for (( b=1; b<100; b++ ))
    do
        if [ $b -eq 5 ]
        then
            break
        fi
        echo "  Inner loop $b"
```

```
    done
done
$ ./test18.sh
Outer loop: 1
  Inner loop 1
  Inner loop 2
  Inner loop 3
  Inner loop 4
Outer loop: 2
  Inner loop 1
  Inner loop 2
  Inner loop 3
  Inner loop 4
Outer loop: 3
  Inner loop 1
  Inner loop 2
  Inner loop 3
  Inner loop 4
```

3. 跳出外部循环

有时你在内部循环，需要停止外部循环。break命令接收单个命令行参数值

```
break n
```

其中n指定了要跳出的循环层级。默认为1表示跳出当前循环。如果将其设置为2，则break命令将会停止下一级的外部循环。

```
$ cat test19.sh
#!/bin/bash
# breaking out of an outer loop

for (( a=1; a<4; a++ ))
do
    echo "Outer loop: $a"
    for (( b=1; b<100; b++ ))
    do
        if [ $b -gt 4 ]
        then
            break 2
        fi
        echo "  Inner loop: $b"
    done
done
$ ./test19.sh
Outer loop: 1
  Inner loop: 1
  Inner loop: 2
  Inner loop: 3
  Inner loop: 4
```

当shell执行了break 2命令后，外部循环就停止了。

6.7.2 continue命令

continue 命令可以提前中止某次循环中的命令，但不会完全终止循环。这里有个在for命令中执行continue命令的简单例子。

```
$ cat test20.sh
#!/bin/bash
# using the continue command
for (( var1 = 1; var1 < 15; var1++ ))
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "Iteration number: $var1"
done
$ ./test20.sh
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Iteration number: 10
Iteration number: 11
Iteration number: 12
Iteration number: 13
Iteration number: 14
```

和break命令相同，continue命令也允许通过命令行参数指定要继续执行那一集循环。

```
continue n
```

其中n定义了要继续的循环层级。下面是一个例子：

```
$ cat test21.sh
#!/bin/bash
# continuing an outer loop
for (( a = 1; a <= 5; a++ ))
do
    echo "Iteration $a:"
    for (( b = 1; b < 3; b++ ))
    do
        if [ $a -gt 2 ] && [ $a -lt 4 ]
        then
            continue 2
        fi
        var3=$[ $a * $b ]
        echo "  The result of $a * $b is $var3"
    done
done
$ ./test21.sh
Iteration 1:
  The result of 1 * 1 is 1
  The result of 1 * 2 is 2
Iteration 2:
  The result of 2 * 1 is 2
```

```
  The result of 2 * 2 is 4
Iteration 3:
Iteration 4:
  The result of 4 * 1 is 4
  The result of 4 * 2 is 8
Iteration 5:
  The result of 5 * 1 is 5
  The result of 5 * 2 is 10
```

此处使用continue命令来停止处理循环内命令，但外部循环仍继续。在n为3时continue命令停止的处理过程。即没有内层循环的输出。

6.8 处理循环的输出

在shell脚本中，你可以对循环的输出使用管道或者重定向。这可以通过在done命令之后添加一个处理命令来实现。

```
$ cat test22.sh
#!/bin/bash
# redirecting the for output to a file

for (( a = 1; a < 6; a++ ))
do
    echo "The number is $a"
done > test22.txt
echo "The command is finished"
$ ./test22.sh
The command is finished
$ cat test22.txt
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
```

此shell脚本创建了文件test22.txt并将for命令的输出重定向到该文件。你还可以将循环结果通过管道给另一个命令。

6.9 实例

以下是一个例子：

6.9.1 查找可执行文件

当你从命令行中运行一个程序的时候，Linux系统会搜索一系列目录来查找对应的文件，这些目录被定义在环境变量PATH中。我们可以编写一个小小的脚本来查找系统中可使用的可执行文件。

```
$ cat test23.sh
#!/bin/bash
# finding files in the PATH
IFS=:
for folder in $PATH
do
    echo "$folder:"
    for file in $folder/*
    do
```

```
        if [ -x $file ]
        then
            echo "  $file"
        fi
    done
done
$ ./test23.sh
...
```

输出了许多在环境变量PATH中找到的可执行文件，我们就不放在上面了。

7. 处理用户输入

7.1 命令行参数

向shell脚本传递数据的最基本方法是使用命令行参数。命令行参数允许在允许脚本时向命令行添加数据。

```
$ ./addem 10 30
```

这个例子向脚本addme传递了两个命令行参数10和30，脚本会通过特殊变量来处理它们。

7.1.1 读取参数

bash shell 会将一些称为位置参数（positional parameter）的特殊变量分配给输入到命令行中的所有参数。位置参数是标准的数字。$0是程序名，$1是第一个参数，$2是第二个参数，以此类推，直到第九个参数$9。

下面是一个简单的例子：

```
$ cat test1.sh
#!/bin/bash
# using one command line parameter
factorial=1
for (( number = 1; number <= $1; number++ ))
do
    factorial=$[ $factorial * $number ]
done
echo The factorial of $1 is $factorial
$ ./test1.sh 5
The factorial of 5 is 120
```

使用多个参数时使用空格分隔开：

```
$ cat test2.sh
#!/bin/bash
# testing two command line parameters
total=$[ $1 * $2 ]
echo The first parameter is $1
echo The second parameter is $2
echo The total value is $total
$ ./test2.sh 2 5
The first parameter is 2
The second parameter is 5
The total value is 10
```

记住，每个参数是用空格分隔的，当参数值中包含空格时，需要使用引号（单引号或双引号都可以）。

```
$ cat test3.sh
echo $1
$ ./test3.sh 'Hello world'
Hello world
$ ./test3.sh "Hello world"
Hello world
```

如果脚本的命令行参数大于9个，你需要稍微修改变量名，在第九个变量之后，需要在变量数字周围加上花括号，如`${10}`。

7.1.2 读取脚本名

可以使用$0参数获取shell在命令行启动时的脚本名。

```
$ cat test3.sh
#!/bin/bash
# Testing the $0 parameter
echo "The zero parameter is set to: $0"
$ bash test3.sh
The zero parameter is set to: test3.sh
$ ./test3.sh
The zero parameter is set to: ./test3.sh
```

如果使用另一个命令来运行shell脚本，命令会和脚本名混在一起出现在$0参数中。

当传给$0变量的实际字符串不仅仅是脚本名字，而是完整的路径时，变量$0会使用整个路径。

```
$ /root/an/chapter14/test3.sh
The zero parameter is set to: /root/an/chapter14/test3.sh
```

有个方便的小命令可以帮助到我们。`basename`命令返回不包含路径的脚本名。

```
$ cat test3b.sh
#!/bin/bash
# Using basename with the $0 parameter
name=$(basename $0)
echo The script name is: $name
$ /root/an/chapter14/test3b.sh
The script name is: test3b.sh
$ ./test3b.sh
The script name is: test3b.sh
```

可以使用这种方法来编写基于脚本名执行不同功能的脚本。这里有一个简单的例子：

```
$ cat test4.sh
#!/bin/bash
# Testing a Multi-function script
name=$(basename $0)
if [ $name = "addem" ]
then
    total=$[ $1 + $2 ]
elif [ $name = "multem" ]
```

```
then
    total=$[ $1 * $2 ]
fi
echo The calculated value is $total
$ cp test4.sh addem
$ chmod u+x addem
$ ln -s test6.sh multem
$ ls -l *em
-rwxr--r-- 1 root root 210 2月   6 14:33 addem
lrwxrwxrwx 1 root root   8 2月   6 14:34 multem -> test4.sh
$ ./addem 2 5
The calculated value is 7
$ ./multem 2 5
The calculated value is 10
```

本例通过复制命令和链接命令创建了两个不同的文件名。两种情况下都会先获得脚本的基本名称，然后根据该值执行相应功能。

7.1.3 测试参数

在shell脚本中使用命令行参数要小心些。如果脚本参数不对可能会出现问题。

```
$ ./addem 2
./addem: 行 6: 2 +  : 语法错误: 需要操作数 (错误符号是 "+  ")
The calculated value is
```

当脚本认为参数变量中会有数据而实际上并没有时，脚本很有可能产生错误消息。在使用参数前一定要检查其中是否存在数据。

```
$ cat test5.sh
#!/bin/bash
# Testing parameters before use
if [ -n "$1" ]
then
    echo Hello $1, glad to meet you.
else
    echo "Sorry, you did not identify yourself. "
fi
$ ./test5.sh an
Hello an, glad to meet you.
$ ./test5.sh
Sorry, you did not identify yourself.
```

本例中使用了-n测试来检查命令行参数$1中是否有数据。在下一节中，我们还将看到另一种方法。

7.2 特殊参数变量

7.2.1 参数统计

特殊变量$#含有脚本运行时携带的命令行参数的个数。我们在使用命令行参数之前应该先检查一下命令行参数。

```
$ cat test6.sh
#!/bin/bash
# getting the number of parameters
echo There were $# parameters supplied.
$ ./test6.sh
There were 0 parameters supplied.
$ ./test6.sh 1 2 3 4 5
There were 5 parameters supplied.
$ ./test6.sh "zi an"
There were 1 parameters supplied.
```

现在我们可以再使用参数前测试参数总数了。

```
$ cat test7.sh
#!/bin/bash
# Testing parameters
if [ $# -ne 2 ]
then
    echo Usage: test7.sh number1 number2
else
    total=$[ $1 + $2 ]
    echo The total is $total
fi
$ ./test7.sh
Usage: test7.sh number1 number2
$ ./test7.sh 1 2
The total is 3
```

这个变量还提供了一种简单的方法来获取命令行中的最后一个参数，而不需要知道实际上到底用了多少个参数。既然$#提供了参数的总数，那么变量${$#}就代表了最后一个参数变量，但由于花括号中不能有美元符，我们必须将美元符换成感叹号，即${!#}。

```
$ cat test8.sh
#!/bin/bash
# Grabbing the last parameter
params=$#
eval a=\$$params
echo The last parameter is $a
echo The last parameter is ${!#}
$ ./test8.sh 2 4
The last parameter is 4
The last parameter is 4
```

这两种方法使用了间接引用，都显示了最后一个参数，但记住当没有参数时，两个变量都返回命令行用到的脚本名，即$0。

7.2.2 获取所有的命令行参数

有时候需要获取命令行上提供的所有参数。这时候不需要先用$#变量来判断命令行上的参数个数再进行遍历，我们可以直接使用两个特殊的变量来解决这个问题。

$*和$@变量可以用来轻松访问所有的参数，不同的是$*将命令行上提供的所有参数当做一个单词保存，它包含了命令行出现的每一个参数值。而$@变量则会将所有参数当做同一字符串的多个独立的单词。我们通常使用for循环遍历它来得到每一个参数。

```
$ cat test9.sh
#!/bin/bash
# testing $* and $@
count=1
for param in "$*"
do
    echo "\$* parameter #$count = $param"
    count=$[ $count + 1 ]
done
echo
count=1
for param in "$@"
do
    echo "\$@ parameter #$count = $param"
    count=$[ $count + 1 ]
done
$ ./test9.sh liu zi an 22 33
$* parameter #1 = liu zi an 22 33

$@ parameter #1 = liu
$@ parameter #2 = zi
$@ parameter #3 = an
$@ parameter #4 = 22
$@ parameter #5 = 33
```

在这个例子中，我们可以看到，"$*"会将所有参数当成一个，而"$@"会将参数分开，单独处理每一个参数。

7.3 移动变量

bash shell 还有一个`shift`命令，它能够操作命令行参数，可以根据它们的相对位置来移动参数。默认情况下会将每个参数往左移动一个位置。即将$2的值移动到$1，$1的值则会被删除。

> shift命令不会改变$0的值，即程序名。

在你不知道有多少个参数时，你可以只操作第一个参数，然后移动参数，继续操作第一个参数。这里有一个例子：

```
$ cat test10.sh
#!/bin/bash
# demostrating the shift command
count=1
while [ -n "$1" ]
do
    echo "Parameter #$count = $1"
    count=$[ $count + 1 ]
    shift
done
$ ./test10.sh 1 3 5 7 9
Parameter #1 = 1
Parameter #2 = 3
Parameter #3 = 5
Parameter #4 = 7
Parameter #5 = 9
```

> 使用shift命令要小心，某个参数被移出后无法恢复。

可以通过在shift命令后添加参数指明要移动的位置数。shift 2 表示移动两个位置。

7.4 处理选项

7.4.1 查找选项

1. 处理简单选项

我们可以使用处理参数的方法来简单处理命令行选项。我们使用`case`语句来处理命令行选项。

```
$ cat test11.sh
#!/bin/bash
# extracting command line options as parameters
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
         *) echo "$1 is not an option" ;;
    esac
    shift
done
$ ./test11.sh -a -b -c -d
Found the -a option
Found the -b option
Found the -c option
-d is not an option
```

2. 分离参数和选项

对Linux来说，可以使用双破折线（--）分离选项和参数。在双破折线之后，脚本就会将剩下的命令行参数当做参数而不是选项来处理。

要检查双破折线，只需要在case语句中加一项就好了。

```
$ cat test12.sh
#!/bin/bash
# extracting options and parameters
echo
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break ;;
        *) echo "$1 is not an option"
    esac
    shift
done
#
count=1
for param in $@
do
    echo "Parameter #$count: $param"
```

```
    count=$[ $count + 1 ]
done
$ ./test12.sh -a -c -b test1 test2

Found the -a option
Found the -c option
Found the -b option
test1 is not an option
test2 is not an option
```

结果表明脚本认为所有的命令行参数都是选项，接下来我们使用双破折线将命令行的选项和参数分离开来。

```
$ ./test12.sh -a -c -b test1 test2

Found the -a option
Found the -c option
Found the -b option
Parameter #1: test1
Parameter #2: test2
```

3. 处理带值的选项

有些选项会带上一个额外的参数值。这种情况下命令行像下面这样。

```
$ ./testing.sh -a test1 -b -c -d test2
```

我们要处理这种情况的命令行参数：

```
$ cat test13.sh
#!/bin/bash
# extracting command line options and values
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) param="$2"
            echo "Found the -b option, with parameter value $param"
            shift ;;
        -c) echo "Found the -c option";;
        --) shift
            break;;
        *) echo "$1 is not an option"
    esac
    shift
done
$ ./test13.sh -a -b test1 -d test2
Found the -a option
Found the -b option, with parameter value test1
-d is not an option
test2 is not an option
```

本例中，case语句定义了三个它需要处理的选项。其中-b选项需要一个额外的参数值。由于要处理的参数位于$1，额外的参数值就位于$2，只需要将参数值从$2值提取出来就行了。由于这个选项占用了两个参数位，我们还需要使用shift命令多移动一个位置。

但是当我们合并选项时，它就不能工作了。在Linux中，合并选项是一个很常见的做法，下面我们介绍一种方法。

7.4.2 使用getopt命令

getopt命令可以识别命令行参数，从而在脚本解析它们时更方便。

1.命令格式

`getopt` optstring paramteters

在optstring 中列出你要在脚本中用到的每个命令行选项字母，并且在每个需要参数的选项字母后加一个冒号。getopt命令会基于你定义的optstring解析提供的参数。

> getopt命令有一个更高级的命令getopts，后面会讲到，注意不要搞混。

下面是一个简单的例子：

```
$ getopt ab:cd -a -b test1 -cd test2 test3
 -a -b test1 -c -d -- test2 test3
```

optstring定义了四个有效字母：a、b、c、d、冒号（:）放在了字母b后，因为b选项需要一个参数值。注意，它会自动的将-cd选项分为两个单独的选项，并插入双破折线来分割行中的额外参数。

如果指定了一个不在optstring中的选项，默认情况下，getopt产生一条错误消息。

```
$ getopt ab:cd -a -b test1 -cde test2 test3
getopt: invalid option -- e
 -a -b test1 -c -d -- test2 test3
```

可以使用-q选项来忽略该条错误消息，需要在命令之后，optstring之前加入。

```
$ getopt -q ab:cd -a -b test1 -cde test2 test3
 -a -b test1 -c -d -- test2 test3
```

2. 在脚本中使用getopt

我们使用`set`命令来使用`getopt`命令。set命令的双破折线（--）选项能够将命令行参数替换成set命令的命令行值。

该方法可以将原始脚本的命令行参数传给`getopt`命令，之后再将`getopt`命令的输出传给`set`命令，从而将用`getopt`格式化后的命令行参数来替换原始的命令行参数。

```
set -- $(getopt -q ab:cd "$@")
```

现在我们可以写出帮我们处理命令行参数的脚本。

```
$ cat test14.sh
#!/bin/bash
# Extract command line options & values with getopt
set -- $(getopt -q ab:cd "$@")
#
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
```

```
        -b) param=$2
            echo "Found the -b option, with the parameter value $param"
            shift ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break ;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done
count=1
for param in "$@"
do
    echo "Parameter #$count: $param"
    count=$[ $count + 1 ]
done
$ ./test14.sh -ac -b test1 -d test2 test3
Found the -a option
Found the -c option
Found the -b option, with the parameter value 'test1'
-d is not an option
Parameter #1: 'test2'
Parameter #2: 'test3'
$ ./test14.sh -ac -b test1 -d "test2 test3" test4
Found the -a option
Found the -c option
Found the -b option, with the parameter value 'test1'
-d is not an option
Parameter #1: 'test2
Parameter #2: test3'
Parameter #3: 'test4'
```

但是仍然有一个小问题，getopt命令无法很好地处理带空格和引号的参数值，它会将空格当做分割符，而不是根据双引号将二者当成一个参数。我们可以使用getopts解决这个问题。

7.4.3 使用更高级的getopts

每次调用`getopts`时，它一次只处理命令行上检测到的一个参数。处理完所有的参数后会退出并返回一个大于0的状态码。这让它非常适合用于解析命令行的所有参数的循环中。

`getopts`命令的格式如下：

```
getopts opstring variable
```

有效的字母选项都列在opstring中，如果该选项要求有个参数值，就加一个冒号。要去掉错误消息可以在opstring前加一个冒号。`getopts`命令将当前参数保存在命令行中定义的variable中。

`getopts`命令会用到两个环境变量。如果选项需要跟一个参数值，OPTARG环境变量就会保存这个值。而OPTING则保存了参数列表中正在处理的参数位置。我们来看一个例子：

```
$ cat test15.sh
#!/bin/bash
# simple demonstraction of the getopts command
#
while getopts :ab:c opt
do
```

```
    case "$opt" in
        a) echo "Found the -a option" ;;
        b) echo "Found the -b option, with value $OPTARG" ;;
        c) echo "Found the -c option" ;;
        *) echo "Unknown option: $opt";;
    esac
done
$ ./test15.sh -ab test1 -c
Found the -a option
Found the -b option, with value test1
Found the -c option
```

`getopts`命令解析命令行选项时会移除开头的单破折线，所以case中不需要单破折线。

并且`getopts`还可以在参数中包含空格。

```
$ ./test15.sh -ab 'test1 test2' -c
Found the -a option
Found the -b option, with value test1 test2
Found the -c option
```

还可以将选项字母和参数值放在一起使用，不需要加空格。

```
$ ./test15.sh -abtest1
Found the -a option
Found the -b option, with value test1
```

除此之外，`getopts`还可以让命令行上所有未定义的选项统一输出为问号。

```
$ ./test15.sh -acde
Found the -a option
Found the -c option
Unknown option: ?
Unknown option: ?
```

在`getopts`命令处理每一个选项时，它会将OPTIND环境变量增1。在`getopts`完成处理时，你可以使用`shift`命令和OPTIND值来移动参数。

```
$ cat test16.sh
#!/bin/bash
# Processing options & parameters with getopts
#
while getopts :ab:cd opt
do
    case "$opt" in
    a) echo "Found the -a option" ;;
    b) echo "Found the -b option, with value $OPTARG" ;;
    c) echo "Found the -c option" ;;
    d) echo "Found the -d option" ;;
    *) echo "Unknown option $opt" ;;
    esac
done
shift $[ $OPTIND - 1 ]
count=1
for param in "$@"
```

```
do
    echo "Parameter $count: $param"
    count=$[ $count + 1 ]
done
$ ./test16.sh -a -b test1 -d test2 test3 test4
Found the -a option
Found the -b option, with value test1
Found the -d option
Parameter 1: test2
Parameter 2: test3
Parameter 3: test4
```

7.5 将选项标准化

下表列出了Linux中一些命令行选项的常用含义。

选项	描述
-a	显示所有对象
-c	生成一个计数
-d	指定一个目录
-f	指定读入数据的文件
-h	显示命令帮助信息
-i	忽略大小写
-l	产生输出的长格式版本
-o	将输出重定向到指定输出文件
-q	以安静模式运行
-y	对所有问题回答yes

7.6 获得用户输入

脚本可以与用户进行更好的交互。比如我们可以在脚本中提一个问题，并且等待运行脚本的人回答。bash shell 为此提供了`read`命令。

7.6.1 基本的读取

`read`命令从标准输入（键盘）或另一个文件描述符中接受输入。接受输入后将数据放入一个变量。下面是它的简单用法。

```
$ cat test17.sh
#!/bin/bash
# testing the read command
echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my program. "
$ ./test17.sh
Enter your name: an
Hello an, welcome to my program.
```

生成提示符的`echo`命令使用了 -n 选项，从而不会在字符串末尾输出换行符。

实际上`echo`命令包含了 -p 选项，允许你直接在read命令行指定提示符。

```
$ cat test18.sh
#!/bin/bash
# testing the read -p option
#
read -p "Please enter your age: " age
days=$[ $age * 365 ]
echo "That makes you over $days days old! "
$ ./test18.sh
Please enter your age: 17
That makes you over 6205 days old!
```

`read`命令可以指定多个变量，它会将输入的每个数据值分配给变量列表中的下一个，如果变量数量不够，剩下的数据将全部分配给最后一个变量。

```
$ cat test19.sh
#!/bin/bash
# entering multiple variables
read -p "Enter your name: " first last
echo "Checking data for $last, $first..."
$ ./test19.sh
Enter your name: Donald Trump
Checking data for Trump, Donald...
```

也可以在`read`命令中不指定变量。这样`read`命令就会将它收到的数据放进环境变量`REPLY`中。

```
$ cat test20.sh
#!/bin/bash
# Testing the REPLY Enviroment variable
#
read -p "Enter your name: "
echo Hello $REPLY, welcome to my program.
$ ./test20.sh
Enter your name: an
Hello an, welcome to my program.
```

`REPLY`环境变量会保存所有的数据，可以像使用其他变量一样使用它。

7.6.2 超时

使用`read`命令时要当心，脚本可能会一直等待用户的输入。我们可以通过 -t 选项来指定一个计时器。-t选项制定了`read`命令等待的秒数。计时器过期后，返回一个非零的状态码。

```
$ cat test21.sh
#!/bin/bash
# timing the data entry
if read -t 5 -p "Please enter your name: " name
then
    echo "Hello $name, welcome to my script"
else
    echo
    echo "Sorry, too slow! "
```

```
fi
$ ./test21.sh
Please enter your name: an
Hello an, welcome to my script
$ ./test21.sh
Please enter your name:
Sorry, too slow!
```

在本例中，当计时器过期时，`read`命令以非零状态码退出，则`if`语句执行之后的`else`部分的命令。

也可以不对输入过程计时，而是让`read`命令统计输入的字符数。当输入字符达到预期字符数时就自动退出，将输入的数据赋值给变量。

```
$ cat test22.sh
#!/bin/bash
# get just one character of input
read -n1 -p "Do you want to continue [Y/N]? " answer
case $answer in
Y|y) echo
     echo "fine, continue on" ;;
N|n) echo
     echo "OK, goodbye"
     exit ;;
esac
echo "This is the end of the script"
$ ./test22.sh
Do you want to continue [Y/N]? y
fine, continue on
This is the end of the script
$ ./test22.sh
Do you want to continue [Y/N]? n
OK, goodbye
```

本例中 -n 选项和值1一起使用，告诉`read`命令在接受单个字符后退出。只要按下单个字符回答后，`read`命令就会接受输出并将它传给变量，而不需要按回车。

7.6.3 隐藏方式读取

有时你想在脚本用户处得到输入，但是又不想在屏幕上显示输入信息。其中典型例子就是输入密码，但除此之外还有很多其他需要隐藏的数据。

-s 选项可以避免在`read`命令中输入的数据出现在显示器上（实际上，数据会被显示，只是文本背景色和文本颜色一样（不会验证，有什么方法吗））。这里有一个使用 -s 选项的例子。

```
$ cat test23.sh
#!/bin/bash
# hiding input data from the monitor
read -s -p "Enter your password: " pass
echo
echo "Is this password really $pass? "
$ ./test23.sh
Enter your password:
Is this password really 123456?
```

输入提示符后输入的数据不会出现在屏幕上，但是会赋给变量，以便使用。

7.6.4 从文件中读取

我们也可以用`read`命令来读取Linux系统上文件里保存的数据。每次调用`read`命令，它都会从文件中读取一行文本。当文件中没有内容时，`read`命令会退出并返回非零退出状态码。

最常见的方法是对文件使用`cat`命令，再将结果通过管道直接传给含有`read`命令的`while`命令。

```
$ cat test24.sh
#!/bin/bash
# reading data from a file
count=1
cat test | while read line
do
    echo "Line $count: $line"
    count=$[ $count + 1 ]
done
echo "Finished processing the file"
$ cat test
first line
second line
the end
$ ./test24.sh
Line 1: first line
Line 2: second line
Line 3: the end
Finished processing the file
```

`while`循环会持续通过`read`命令处理文件中的行，直到`read`命令以非零状态吗退出。

8. 呈现数据

8.1 理解输入和输出

现在你已经知道了两种显示脚本输出的方法：

- 在显示器屏幕上显示输出
- 将输出重定向到文件中

下面几节将会介绍如何使用标准的Linux输入和输出系统来将脚本输出导向特定位置。

8.1.1 标准文件描述符

Linux系统将每个对象当做文件处理，包括输入和输出进程。Linux使用**文件描述符**（file descriptor）来标识每一个文件对象。文件描述符是一个非负整数，可以唯一标识会话中打开的文件。每个进程一次最多九个文件描述符。bash shell保留了前三个文件描述符（0、1和2），见下表。

表8-1 Linux的标准文件描述符

文件描述符	缩写	描述
0	STDIN	标准输入
1	STDOUT	标准输出
2	STDERR	标准错误

1. STDIN

STDIN文件描述符代表shell的标准输入。对终端界面来说，标准输入是键盘。shell从STDIN文件描述符对应的键盘获得输入，并在用户输入时处理每个字符。

在使用输入重定向符号（<）时，Linux会使用重定向指定的文件来替换标准输入文件描述符。它会读取文件并提取数据，如同它是在键盘上键入的。

许多bash命令可以接受STDIN的输入，尤其是没有在命令行上指定文件的话。下面是一个用cat命令处理STDIN输入的例子。

```
$ cat
this is a test
this is a test
this is a second test
this is a second test
```

当在命令行上只输入`cat`命令时，它会从STDIN接受输入。输入一行，`cat`命令就会显示一行。

2. STDOUT

STDOUT文件描述符代表shell的标准输出。在终端界面上，标准输出就是终端显示器。shell的所有输出（包括shell中运行的程序和脚本）会被定向到标准输出中，也就是显示器。

默认情况下，大多数bash命令会将输出导向STDOUT文件描述符。但如前面所说的，我们也可以使用输出重定向来改变。

但是对脚本使用标准输出重定向时，你会遇到一个问题，例如：

```
$ ls -al badfile > test3
ls: cannot access badfile: No such file or directory
$ cat test3
```

命令生成错误消息时，shell并未将错误消息重定向到输出重定向文件。shell创建了输出重定向文件，但是错误消息缺显示在了显示器屏幕上。注意，test3文件创建成功了，只是里面是空的。

3. STDERR

shell通过特殊的STDERR文件描述符来处理错误消息。 STDERR文件描述符代表shell的标准错误输出。shell或者shell中运行的程序和脚本出错时生成的错误消息都会发送到这个位置。

默认情况下，STDERR文件描述符也与STDOUT指向相同的地方，即也会显示到显示器上，并且STDERR不随着STDOUT的改变而改变。但有时我们希望能够重定向错误消息到日志文件中。

8.1.2 重定向错误

我们只要在使用重定向符号时定义STDERR描述符就好了。有几种办法实现：

1.只重定向错误

可以选择只重定向错误消息，我们将STDERR对应的文件描述符2放在重定向符号前，并且注意要紧贴重定向符号。

```
$ ls -al badfile test 2> test4
-rw-r--r-- 1 root root 0 2月  22 21:35 test
$ cat test4
ls: 无法访问 'badfile': 没有那个文件或目录
```

ls命令正常STDOUT输出仍然发送到默认的STDOUT文件描述符，即显示器。由于将文件描述符2的输出（STDERR）重定向到另一个文件，shell会将生成的所有错误消息直接发送到指定的重定向文件。

2. 重定向错误和数据

如果想重定向错误和正常输出，必须使用两个重定向符号。需要在符号前面放上待重定向数据所对应的文件描述符，然后指向用于保存数据的输出文件。

```
$ ls
test2  test3  test4
$ ls -l test test2 badtest 2> test5 1> test6
$ cat test6
-rw-r--r-- 1 root root 0 2月  22 21:35 test2
$ cat test5
ls: 无法访问 'test': 没有那个文件或目录
ls: 无法访问 'badtest': 没有那个文件或目录
```

shell利用`1>`符号将`ls`命令的正常输出重定向到了test6文件，而错误输出重定向到到了test5文件。这种方法可以将脚本正常输出和错误输出消息分离开来。

另外，我们也可以将STDOUT和STDERR的输出重定向到同一个文件，为此bash shell提供了特殊的重定向符号`&>`。

```
$ ls -l test test2 badtest &> test7
$ cat test7
ls: 无法访问 'test': 没有那个文件或目录
ls: 无法访问 'badtest': 没有那个文件或目录
-rw-r--r-- 1 root root 0 2月  22 21:35 test2
```

通过`&>`符，命令生成的输出都发送到同一位置，包括数据和错误。我们还可以注意到它输出的顺序不一样。为了避免错误信息散落在输出文件中，bash shell自动赋予的错误消息更高的优先级。

8.2 在脚本中重定向输出

有两种方法在脚本中重定向输出：

- 临时重定向输出
- 永久重定向脚本中所有命令

8.2.1 临时重定向

如果有意在脚本生成错误信息，可以将单独一行重定向到STDERR。你需要做的就是使用输出重定向符号来将信息重定向到STDERR文件描述符。重定向到文件描述符时，你需要在**文件描述符数字之前加一个&**：

```
echo "This is an error message" >&2
```

这会在脚本的STDERR文件描述符所指向的位置显示文本，而不是通常的STDOUT。我们看一个例子：

```
$ cat test1.sh
#!/bin/bash
# tesing STDERR messages
echo "This is an error" >&2
echo "This is an normal output"
```

如果正常运行这个脚本，我们看不出什么区别。

```
$ ./test1.sh
This is an error
This is an normal output
```

默认情况下，Linux将STDERR导向STDOUT。但是，如果你在运行脚本时重定向了STDERR，脚本中所有导向STDERR的文本都会重定向。

```
$ ./test1.sh 2> test8
This is an normal output
$ cat test8
This is an error
```

这个方法方程适合在脚本中生成错误消息，我们可以轻松的通过STDERR文件描述符重定向错误消息。

8.2.2 永久重定向

如果脚本中有大量数据需要重定向，我们可以使用exec命令来告诉shell在脚本执行期间重定向某个特定文件描述符。

```
$ cat test2.sh
#!/bin/bash
# redirecting output to different locations
exec 2>testerror

echo "This is the start of the script"
echo "Now redirecting all output to another location"

exec 1>testout

echo "This output should go to the testout file"
echo "But this should go to the testerror file" >&2
$ ./test2.sh
This is the start of the script
Now redirecting all output to another location
$ cat testout
This output should go to the testout file
$ cat testerror
But this should go to the testerror file
```

这个脚本使用exec命令来将发给STDERR的输出重定向到文件testerror。接下来，脚本使用echo语句显示了几行文本。随后再次用exec命令将STDOUT重定向到testout文件，但是STDERR仍然指向testerror。

8.3 在脚本中重定向输入

你可以使用与脚本中重定向STDOUT和STDERR相同的方法来将STDIN从键盘上重定向到其他位置。exec命令允许你将STDIN重定向到Linux系统上的文件中：

```
exec 0< testfile
```

这个命令会告诉shell它应该从文件testfile中获得输入，而不是STDIN。这个重定向只要在脚本需要输入时就会起作用。

```
$ cat test3.sh
#!/bin/bash
# redirecting file input

exec 0< testfile
count=1

while read line
do
    echo "Line #$count: $line"
    count=$[ $count + 1 ]
done
$ ./test3.sh
Line #1: This is the first line
Line #2: This is the second line
Line #3: This is the thrid line
```

当STDIN重定向到文件后，当read命令试图从STDIN读入数据时，它会到文件去取数据，而不是键盘。

8.4 创建自己的重定向

shell中最多有9个打开的文件描述符。其他6个从3~8的文件描述符均可用作输入或输出重定向。

8.4.1 创建输出文件描述符

可以用exec命令给输出分配文件描述符。和标准文件描述符一样，一旦将另一个文件描述符分配给另一个文件，这个重定向就会一直有效，直到你重新分配。这里有个在脚本中使用其他文件描述符的简单例子：

```
$ cat test4.sh
#!/bin/bash
# using an alternative file descriptor

exec 3> test4out

echo "This should display on the monitor"
echo "and this should be stored in the file" >&3
echo "This should be back on the monitor"
$ ./test4.sh
This should display on the monitor
This should be back on the monitor
$ cat test4out
and this should be stored in the file
```

这个脚本用exec命令将文件描述符3重定向到另一个文件。当脚本执行echo语句时，输出内容如预想中那样显示在STDOUT上。但你重定向到文件描述符3的那行echo语句的输出却进入到了另一个文件。这样就可以将输出分开来。

也可以不创建新文件，使用exec命令来将输出追加到现有文件中。

```
exec 3>> test4out
```

现在输出将追加到test4out文件中，而不是创建一个新文件。

8.4.2 重定向文件描述符

现在介绍如何恢复已重定向的文件描述符。你可以分配另外一个文件描述符给标准文件描述符，反之亦然。这意味着你可以将STDOUT的原来位置重定向到另一个文件描述符，然后再利用该文件描述符重定向回STDOUT。下面是一个简单的例子：

```
$ cat test5.sh
#!/bin/bash
# storing STDOUT, then coming back to it

exec 3>&1
exec 1>test5out

echo "This should store in the output file"
echo "along with this line."

exec 1>&3

echo "Now things should be back to normal"
$ ./test5.sh
Now things should be back to normal
$ cat test5out
This should store in the output file
along with this line.
```

我们来看看这个例子。首先，脚本将文件描述符3重定向到文件描述符1当前位置，也即STDOUT。第二个exec命令再将STDOUT重定向到文件test5out，现在shell会将发送给STDOUT的输出直接重定向到test5out文件中，而文件描述符3仍然指向STDOUT原来的位置，也就是显示器。最后我们再将文件描述符1重定向到文件描述符3（此时是指向显示器），则STDOUT又指向了它原来指向的位置。

这是一种在脚本中临时重定向输出，然后恢复默认输出设置的常用方法。

8.4.3 创建输入文件描述符

可以用和重定向输出文件描述符相同的方法重定向输入文件描述符。在重定向到文件之前，现将STDIN文件描述符保存到另外一个文件描述符，然后在读取完文件之后再将STDIN恢复到它原来的位置。

```
$ cat test6.sh
#!/bin/bash
# redirecting input file descriptors

exec 6<&0

exec 0< testfile

count=1
while read line;do
    echo "Line #$count: $line"
    count=$[ $count + 1 ]
done
exec 0<&6
read -p "Are you done now? " answer
case $answer in
    Y|y) echo "Goodbye";;
    N|n) echo "Sorry, this is the end";;
```

```
esac
$ ./test6.sh
Line #1: This is the first line
Line #2: This is the second line
Line #3: This is the thrid line
Are you done now? y
Goodbye
```

这个例子中，文件描述符6用来保存STDIN的位置。然后脚本将STDIN重定向到一个文件夹。read命令的所有输入都来自重定向之后的STDIN（也就是输入文件）。

在读取了所有行之后，脚本会将STDIN重定向到文件描述符6，从而STDIN恢复到原先的位置（即键盘）。该脚本最后使用了另一个read命令来测试STDIN是否恢复正常了，这次它会等待键盘的输入。

8.4.4 关闭文件描述符

如果你创建了新的输入或输出文件描述符，shell脚本会在脚本退出时候自动关闭它们。然而在有些情况下，你需要在脚本结束前手动关闭文件描述符。

要关闭文件描述符，将它重定向到特殊符号&-。在脚本中看起来如下：

```
exec 3>&-
```

该语句会关闭文件描述符3，不再在脚本中使用它。这里有例子来说明当你尝试使用已关闭的文件描述符会怎么样。

```
$ cat badtest
#!/bin/bash
# testing closing file descriptors

exec 3> test7file

echo "This is a test line of data" >&3

exec 3>&-

echo "This won't work" >&3
$ ./badtest.sh
./badtest: 3: Bad file description
```

一旦关闭了文件描述符，就不能在脚本中向它写入任何数据，否则shell会生成错误消息。

在关闭文件描述符时还要注意另一件事，如果随后你在脚本中打开了同一个输出文件，shell会用一个新文件来替换已有文件。这意味着如果你输出数据，它就会覆盖已有文件。考虑下面这个例子：

```
$ cat test7.sh
#!/bin/bash
# testing closing file descriptors

exec 3> test7file
echo "This is a test line of data" >&3
exec 3>&-

cat test7file

exec 3> test7file
```

```
echo "This'll be bad" >&3
$ ./test7.sh
This is a test line of data
$ cat test7file
This'll be bad
```

在向 test7file 文件发送一个数据字符串并关闭文件描述符之后，脚本使用了`cat`命令来显示文件内容。下一步，脚本重新打开了该输出文件并向它发送了另一个数据字符串。当显示该输出文件的内容时，你所能看到的只有第二个数据字符串。shell覆盖了原来的输出文件。

8.5 列出打开的文件描述符 lsof

你能用到的文件描述符只有9个，你可能会觉得这没什么复杂的。但有时要记住哪个文件描述符被重定向到了哪里很难。为了帮助你理清条理，bash shell提供了`lsof`命令。

`lsof`命令会列出整个Linux系统打开的所有文件描述符。这是一个有争议的功能，以为它会向非系统管理员提供Linux系统的信息。

该命令会产生大量的输出。它会显示当前Linux系统上打开的每个文件的有关信息。这包括后台运行的所有进程以及登录到系统的任何用户。

有大量的命令行选项和参数可以用来帮助过滤`lsof`命令的输出。最常用的有`-p`和`-d`，前者允许指定进程ID（PID），后者允许指定要显示的文件描述符编号。

要想知道进程的当前PID，可以使用特殊环境变量`$$`（shell会将它设为当前PID）。`-a`选项用来对其他两个选项的结果执行布尔AND运算，这会产生如下输出。

```
$ lsof -a -p $$ -d 0,1,2
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
bash    2593 root    0u   CHR  136,3      0t0    6 /dev/pts/3
bash    2593 root    1u   CHR  136,3      0t0    6 /dev/pts/3
bash    2593 root    2u   CHR  136,3      0t0    6 /dev/pts/3
```

上例显示了当前进程（bash shell）的默认文件（0、1和2）。`lsof`的默认输出中有9项信息。见下表：

列	描述
COMMAND	正在运行的命令名的前9个字符
PID	进程的PID
USER	进程属主的登录名
FD	文件描述符号以及访问类型（r代表读，w代表写，u代表读写）
TYPE	文件类型（CHR代表字符型，BLK代表块型，DIR代表目录，REG代表常规文件）
DEVICE	设备的设备号（主设备号和从设备号）
SIZE	如果有，表示文件大小
NODE	本地文件的节点号
NAME	文件名

与STDIN、STDOUT和STDERR关联的文件类型是字符型。因为STDIN、STDOUT和STDERR文件描述符都指向终端，所以输出文件的名称就是终端的设备名。所有三种标准文件都支持读和写。（尽管向STDIN写数据以及从STDOUT读数据看起来有点奇怪）。

现在看一下在打开了多个替代性文件描述符的脚本中使用`lsof`命令的结果。

```
$ cat test8.sh
#!/bin/bash
# testing lsof with file descriptors

exec 3> test8file1
exec 6> test8file2
exec 7< testfile

lsof -a -p $$ -d 0,1,2,3,6,7
$ ./test8.sh
COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
test8.sh 2917 root    0u   CHR  136,3      0t0      6 /dev/pts/3
test8.sh 2917 root    1u   CHR  136,3      0t0      6 /dev/pts/3
test8.sh 2917 root    2u   CHR  136,3      0t0      6 /dev/pts/3
test8.sh 2917 root    3w   REG 253,26        0 299834
/root/an/chapter15/test8file1
test8.sh 2917 root    6w   REG 253,26        0 299835
/root/an/chapter15/test8file2
test8.sh 2917 root    7r   REG 253,26       70 263467
/root/an/chapter15/testfile
```

该脚本创建了三个可替代性文件描述符，两个作为输出（3和6），一个作为输入（7）。在脚本中运行lsof命令时，可以在输出中看到新的文件描述符。文件名显示了文件描述符所使用的文件的完整路径名。它将每个文件都显示成REG类型的，这说明它们是文件系统中的常规文件。

8.6 阻止命令输出

有时候，你可能不想显示脚本的输出，这在脚本作为后台进程运行时很常见。如果运行在后台的脚本出现错误信息，shell会通过电子邮件将它们发给进程的属主。这会很麻烦，尤其是当运行会生成很多繁琐的小错误的脚本时。

要解决这个问题，可以将STDERR重定向到一个叫做null文件的特殊文件。null文件顾名思义，文件里什么都没有。shell输出到null文件的任何数据都不会保存，全部被丢掉了。

在Linux系统上null文件的标准位置是`/dev/null`。你重定向到该位置的任何数据都会被丢掉,不会显示。

```
$ ls -al > /dev/null
$ cat /dev/null
```

这是避免显示错误消息，也无需保存它们的一个常用方法。

```
$ ls -al badfile test6 2> /dev/null
-rw-r--r-- 1 root root 46 3月   2 20:47 test6
```

也可以在输出重定向中将`/dev/null`作为输入文件。由于`/dev/null`文件不含有任何内容，程序员通常用它来快速清除现有文件中的数据，而不用先删除文件再重新创建。

```
$ cat testfile
This is the first line
This is the second line
This is the thrid line
$ cat /dev/null > testfile
$ cat testfile
```

文件testfile仍然在系统上，但现在它是空文件。这是清除日志文件的一个常用方法，因为日志文件必须时刻准备等待应用程序操作。

8.7 创建临时文件

Linux系统有特殊的目录，专供临时文件使用。Linux使用`/tmp`目录来存放不需要永久保留的文件，大部分Linux发行版配置了系统在启动时自动删除`/tmp`目录的所有文件。

系统上的任何用户账户都有权限读写`/tmp`中的文件。这个特性为你提供了一种创建临时文件的简单方法，而且还不用担心清理工作。

有个特殊命令可以用来创建临时文件。`mktemp`命令可以在`/tmp`目录中创建一个唯一的临时文件。shell会创建这个文件，但不用默认的`umask`值。它会将文件的读和写权限分配给文件的属主，并将你设成文件的属主。一旦创建了文件，你就在脚本中有了完整的读写权限，但其他人没法访问它（当然，root用户除外）。

8.7.1 创建本地临时文件

默认情况下，`mktemp`会在本地目录中创建一个文件。要用`mktemp`命令在本地目录中创建一个临时文件，你只要指定一个文件名模板就行了。模板可以包含任意文本文件名，在文件名末尾加上6个X（大写）就行了。（经验证，X数目不能小于3）

```
$ mktemp testing.XXXXXX
testing.CPOgsB
$ ls -al testing*
-rw------- 1 an an 0 Mar 28 14:51 testing.CPOgsB
```

`mktemp`命令会用6个字符码替换这6个X，从而保证文件名在目录中唯一。你可以创建多个临时文件，它可以保证每个文件都是唯一的。`mktemp`命令的输出正是它所创建的文件的名字。在脚本中使用`mktemp`命令时，可能需要将文件名保存到变量中，这样就能在后面的脚本中引用了。

```
$ cat test9.sh
#!/bin/bash
# creating and using a temp file

tempfile=$(mktemp test9.XXXXXX)

exec 3> $tempfile

echo "This script writes to temp file $tempfile"

echo "This is the first line" >&3
echo "This is the second line" >&3
echo "This is the thrid line" >&3
exec 3>&-

echo "Done creating temp file. The contents are: "
cat $tempfile
```

```
rm -f $tempfile 2> /dev/null
$ ./test9.sh
This script writes to temp file test9.X32lYa
Done creating temp file. The contents are:
This is the first line
This is the second line
This is the thrid line
```

这个脚本使用`mktemp`命令来创建临时文件并将文件名赋值给`$tempfile`变量。接着将这个临时文件作为文件描述符3的输出重定向文件。在将临时文件名显示在STDOUT之后，向临时文件中写入了几行文本，然后关闭了文件描述符。最后，显示出临时文件的内容，并用`rm`命令将其删除。

8.7.2 在/tmp目录创建临时文件

`-t`选项会强制`mktemp`命令来在系统的临时目录来创建该文件。在用这个特性时，`mktemp`命令会返回用来创建临时文件的全路径，而不是只有文件名。

```
$ mktemp -t test.XXXXXX
/tmp/test.14yhKr
$ ls -al /tmp/test*
-rw------- 1 an an 0 Mar 28 23:20 /tmp/test.14yhKr
```

由于`mktemp`命令返回了全路径名，你可以在Linux系统上的任何目录下引用该临时文件，不管临时目录在哪里。

```
$ cat test10.sh
#!/bin/bash
# creating a temp file in /tmp

tempfile=$(mktemp -t tmp.XXXXXX)

echo "This is a test file." > $tempfile
echo "This is the second line of the test." >> $tempfile

echo "The temp file is located at: $tempfile"
cat $tempfile
rm -f $tempfile
$ ./test10.sh
The temp file is located at: /tmp/tmp.5W6Ye7
This is a test file.
This is the second line of the test.
```

在`mktemp`创建临时文件时，它会将全路径名返回给变量。这样你就能在任何命令中使用该值来引用临时文件了。

8.7.3 创建临时目录

`-d`选项告诉`mktemp`命令来创建一个临时目录而不是临时文件。这样你就可以利用该目录进行你想要的操作了，比如创建其他的临时文件。

```
$ cat test11.sh
#!/bin/bash
# using a temporary directory

tempdir=$(mktemp -d dir.XXXXXX)
```

```
cd $tempdir
tempfile1=$(mktemp temp.XXXXXX)
tempfile2=$(mktemp temp.XXXXXX)
exec 7> $tempfile1
exec 8> $tempfile2

echo "Sending data to directory $tempdir"
echo "This is a test line of data for $tempfile1" >&7
echo "This is a test line of data for $tempfile2" >&8
$ ./test11.sh
Sending data to directory dir.Wj4jiZ
$ cd dir.Wj4jiZ
$ ls -al
total 16
drwx------ 2 an an 4096 Apr  1 16:42 .
drwxr-xr-x 3 an an 4096 Apr  1 16:42 ..
-rw------- 1 an an   44 Apr  1 16:42 temp.2d7pck
-rw------- 1 an an   44 Apr  1 16:42 temp.knoI98
$ cat temp.2d7pck
This is a test line of data for temp.2d7pck
$ cat temp.knoI98
This is a test line of data for temp.knoI98
```

这段脚本在当前目录创建了一个目录，然后它用`cd`命令进入该目录，并创建了两个临时文件。之后这两个临时文件被分配给文件描述符，用来储存脚本的输出。

8.8 记录消息

将输出消息发送到显示器和日志文件，这种做法有时候能派上用场。你不用将输出重定向两次，只需要用特殊的`tee`命令就行。

`tee`命令相当于管道的一个T型接头。它将从STDIN过来的数据同时发往两处。一处是STDOUT，另一处是`tee`命令行所指定的文件名：

```
tee filename
```

由于`tee`会重定向来自STDIN的数据，你可以用它配合管道命令来重定向命令输出。

```
$ date | tee testfile
Sat Apr  2 15:17:06 CST 2022
$ cat testfile
Sat Apr  2 15:17:06 CST 2022
```

输出出现在了STDOUT中，同时也写入了指定的文件中。注意，默认情况下，`tee`命令会在每次使用时覆盖输出文件内容。

```
$ who | tee testfile
root     pts/3        2022-04-03 19:19 (172.31.0.2)
$ cat testfile
root     pts/3        2022-04-03 19:19 (172.31.0.2)
```

如果想将数据追加到文件中，必须使用`-a`选项。

```
$ date | tee -a testfile
2022年 04月 03日 星期日 19:21:19 CST
$ cat testfile
root     pts/3        2022-04-03 19:19 (172.31.0.2)
2022年 04月 03日 星期日 19:21:19 CST
```

利用这个方法，既能将数据保存在文件中，也能将数据显示在屏幕上。

```
$ cat test12.sh
#!/bin/bash
# using the tee command for logging

tempfile='test12file'

echo "This is the start of the test" | tee $tempfile
echo "This is the second line of the test" | tee -a $tempfile
echo "This is the end of the test" | tee -a $tempfile
$ ./test12.sh
This is the start of the test
This is the second line of the test
This is the end of the test
$ cat test12file
This is the start of the test
This is the second line of the test
This is the end of the test
```

现在你就可以在为用户显示输出的同时再永久保存一份输出内容了。

8.9 实例

文件重定向常见于脚本需要读入文件和输出文件时。这个样例两件事都做了。它读取.csv格式的数据文件，输出SQL `INSERT` 语句来将数据插入数据库（以后会讲）。

`shell` 脚本使用命令行参数指定待读取的.csv文件。.csv格式用于从电子表格中导出数据，所以你可以把数据库数据放入电子表格中，把电子表格保存成.csv格式，读取文件，然后创建 `INSERT` 语句将数据插入MySQL数据库 。

```
$ cat test13.sh
#!/bin/bash
# read file and create INSERT statements for MySQL

outfile='members.sql'
IFS=','
while read lname fname address city state zip
do
    cat >> $outfile << EOF
    INSERT INTO members (lname,fname,address,city,state,zip) VALUES
    ('$lname', '$fname', '$address', '$city', '$state', '$zip');
EOF
done < ${1}
```

这个脚本出现了三处重定向操作。 `while` 循环使用了 `read` 语句从数据文件中读取文本。注意在 `done` 语句中出现的重定向符号：

```
done < ${1}
```

当运行程序`test13`时，`$1`代表第一个命令行参数。它指明了待读取数据的文件。`read`语句会使用IFS字符解析读入的文本，我们在这里将IFS指定为逗号。

脚本中另外两处重定向操作出现在同一条语句中：

```
cat >> $outfile << EOF
```

这条语句包含一个输出追加重定向（双大于号）和一个输入追加重定向（双小于号）。输出重定向将`cat`命令的输出追加到$outfile变量指定的文件中。`cat`命令的输入不再取自标准输入，而是被重定向到脚本中储存的数据。`EOF`符号标记了追加到文件中的数据的起止。

```
INSERT INTO members (lname,fname,address,city,state,zip) VALUES
('$lname', '$fname', '$address', '$city', '$state', '$zip');
```

上面的文本生成了一个标准的SQL`INSERT`语句。注意，其中的数据会由变量来替换，变量中的内容则是由`read`语句存入的。

在这个例子中，使用以下数据文件。

```
$ cat members.csv
Blum,Richard,123 Main St.,Chicago,IL,60601
Blum,Barbara,123 Main St.,Chicago,IL,60601
Bresnahan,Christine,456 Oak Ave.,Columbus,OH,43201
Bresnahan,Timothy,456 Oak Ave.,Columbus,OH,43201
$ ./test13.sh members.csv
$ cat members.sql
    INSERT INTO members (lname,fname,address,city,state,zip) VALUES
    ('Blum', 'Richard', '123 Main St.', 'Chicago', 'IL', '60601');
    INSERT INTO members (lname,fname,address,city,state,zip) VALUES
    ('Blum', 'Barbara', '123 Main St.', 'Chicago', 'IL', '60601');
    INSERT INTO members (lname,fname,address,city,state,zip) VALUES
    ('Bresnahan', 'Christine', '456 Oak Ave.', 'Columbus', 'OH', '43201');
    INSERT INTO members (lname,fname,address,city,state,zip) VALUES
    ('Bresnahan', 'Timothy', '456 Oak Ave.', 'Columbus', 'OH', '43201');
```

结果和我们预想的一样，现在可以将members.sql文件导入MySQL数据表中了。

9. 控制脚本

我们可以通过一些方法来控制脚本的运行。这些方法包括向脚本发送信号、修改脚本优先级以及在脚本运行时切换到运行模式。本章将会逐一介绍这些方法。

9.1 处理信号

Linux利用信号与运行在系统中的进程进行通信。前面介绍了不同的Linux信号以及Linux如何使用这些信号来停止、启动、终止进程。可以通过对脚本进行编程，使其在收到特定信号时执行某些命令，从而控制shell脚本的工作。

9.1.1 重温Linux信号

Linux系统和应用程序可以生成超过30个信号。表9-1列出了在Linux编程时会遇到的最常见的Linux系统信号。

信号	值	描述
1	SIGHUP	挂起进程
2	SIGINT	终止进程
3	SIGQUIT	停止进程
9	SIGKILL	无条件停止进程
15	SIGTERM	尽可能终止进程
17	SIGSTOP	无条件停止进程，但不是终止进程
18	SIGTSTP	停止或暂停进程，但不终止进程
19	SIGCONT	继续运行停止的进程

默认情况下，bash shell会忽略收到的任何SIGQUIT（3）和SIGTERM（5）信号（正因为这样，交互式shell才不会被意外终止）。但是bash shell会处理收到的SIGHUP（1）和SIGINT（2）信号。

如果bash shell收到了SIGHUP信号，比如当你要离开一个交互式shell，它就会退出。但在退出之前，它会将SIGHUP信号传给所有由该shell所启用的进程（包括正在运行的shell脚本）。

通过SIGINT信号，可以中断shell。Linux内核会停止为shell分配CPU处理时间。这种情况发生时，shell会将SIGINT信号传给所有由它所启用的进程，以此告知出现的状况。

shell会将这些信号传给shell脚本程序来处理。而shell脚本的默认行为是忽略这些信号。它们可能不利于脚本的运行。要避免这种情况，你可以在脚本中加入识别信号的代码，并执行命令来处理信号。