bash程序设计

Objectives

- 介绍shell编程的概念
- 讨论shell程序是如何执行的
- 描述shell变量的概念和使用方法
- 讨论命令行参数是如何传给shell程序的
- 解释命令替换的概念
- 介绍一些编程的基本原则
- 编写并讨论一些shell脚本
- 讨论数值数据处理，数组处理
- 描述如何将shell中的命令的标准输入重定向到脚本中的数据
- 解释bash中的函数

引言

引言

- Shell 脚本(script): 一个shell程序,由可执行的shell命令组成，以普通linux文件形式保存
- Shell 变量:shell允许使用一些读写存储区，为用户和程序设计人员提供一个暂存数据的区域。
- 程序流程控制命令 Program control flow commands (or statements): 提供了对shell脚本中的命令进行非顺序执行或循环执行的功能

运行一个 Shell 脚本

- 运行 Bourne Shell脚本的方法：
  - 为脚本文件加上可执行的权限

    $ chmod u+x script_file

    $./script_file

    - 设置当前目录为缺省查找路径：PATH=.:$PATH
  - 运行/bin/bash命令并且把脚本文件名作为它的参数。

    $ /bin/bash script_file

例

- 创建一个文件，其中包含了一个使用date和who命令的shell脚本，每条命令写在一行。使得文件可执行并运行这个shell脚本。写出完成这项工作的全部步骤。
  - 用编辑器（如vi）创建一个文件：

```
$ vi ch15ex2    #输入文本
$ cat ch15ex2
#! /bin/bash
date
who
$ chmod +x ch15ex2  #all加上x权限
$ ./ch15ex2
Mon Jul 31 17:49:42 PDT 2006
root pts/0 Jul 31 16:22
```

注释和程序头

注释和程序头

- 应该养成在程序中加入注释的习惯，以描述一些特殊命令的作用。
- 应该为你编写的每个shell脚本加入一个程序头。
- 一个好的程序头至少应包括下面列出的内容：
  - 包含脚本的文件名。
  - 作者的名字
  - 编写的日期
  - 最后一次修改的日期
  - 编写这个脚本的目的（用一两行来说明）
  - 对解决这个问题的算法的简要描述
- 另外，你可以把任何你认为重要的或是在你的组织团队中常用的条款加入到程序头中，通常称为编码规范
- # 注释命令，#后面的内容作为注释信息。

8.1 shell变量

Shell 变量

- shell变量：
  1. shell环境变量
  2. 用户定义变量

1. Shell环境变量

- 环境变量用来定制你的shell的运行环境，保证shell命令的正确执行。所有环境变量会传递给shell的子进程。
- 当登录Linux系统时，bash shell会作为登录shell启动。登录shell会从5个不同的启动文件里读取命令：
  - /etc/profile
  - $HOME/.bash_profile
  - $HOME/.bashrc
  - $HOME/.bash_login
  - $HOME/.profile

发行版不同可能会有所区别

环境变量

	一些常用的Bash环境变量
环境变量	变量的用途
BASH	bash的完整路径名
CDPATH	包含cd命令要逐个查找的路径，cd命令在这些路径下查找作为参数传递给他的目录名。如果CDPATH没有设置，cd命令则查找当前目录
EUID	有效的当前用户ID
LOGNAME	当前用户的登录名
HISTFILE	存放历史记录的文件的路径名
HOME	用户的家目录的名字
IFS	bash用来分割命令行中参数的分割符号
HOSTNAME	主机名
PATH	包含用户检索路径的变量—Shell依据这个变量在他指出的目录下面查找外部命令和程序
PPID	父进程的ID号
PS1	出现在命令行的主Shell提示符，通常被设置为$，但是这是可以修改的，可以通过在本节稍后讨论到的特殊符号来修改
PS2	出现在一个命令的第二行的二级Shell提示符号。典型的情况就是当一行命令以‘\’结束的时候，表示命令没有结束，还有下一行
PWD	当前工作目录的名字
TERM	用户的控制终端的类型
UID	用户ID号

shell环境变量

表	一些重要的只读Bash环境变量
环境变量	这些变量的用途
$0	程序的名字
$1~$9	命令行参数1~9的值。 大于9时，用大括号：${10}、${11}
$*	所有命令行参数的值（一起作为单个字符串）
$@	所有命令行参数的值（每个作为独立字符串） 如果$@被“”包括，即“$@”，这相当于其中的每一个参数的值被“”包括，相反，如果$*被“”包括，即“$*”，这就相当于所有的参数值作为一个串被“”包括。这就是$@同$*在被“”包括的时候的差别，其他时候这二者是等价的
$#	记录了命令行参数的总个数
$$	当前进程的ID号
$?	上一条命令的退出状态
$!	最近一次后台进程的ID号

```
# set `date`
# mv1=("$*")
# mv2=("$@")
# echo ${mv1[0]}
2019年 05月 2日 星期四 21:52:08 CST
# echo ${mv2[0]}
2019年
# echo ${mv2[1]}
05月
```

Shell环境变量

- echo $变量名，显示指定变量的值

  例：echo $PATH $PS1 $PPID

- set、env、declare、typeset、printenv命令显示环境变量和其值
- 用户定义的变量在shell脚本中用来作为临时的存储空间
- 一个没有初始化的shell变量自动地被初始化为一个空串
- 用shell内嵌命令declare，local，set和typeset来初始化用户变量

控制shell提示符的变量

- Linux可以指定一个或多个特殊字符作为提示符变量（PS1、PS2等）
- 指定当前工作目录作为提示符

```
$ PS1='\w$'
~/linuxbook/examples/bash$
```

表 一些有用的特殊字符及其描述

特殊字符	描述
\H	主机域名的全称
\T	时间，格式为hh:mm:ss，12小时格式
\d	日期，格式为“weekday month date”
\h	计算机的主机名的第一部分（第一个‘.’前面的部分），如upsunzo
\s	用户shell的名字
\t	时间，格式为hh:mm:ss，24小时格式
\u	当前用户的用户名
\v	Bash的版本号，如2.03
\w	当前的工作目录

2. 用户变量

- 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个。用户变量区分大小写，所以变量Var1和变量var1是不同的。

- bash并不要求你一定声明变量，但是你可以使用declare和typeset命令来声明变量，对它们进行初始化，并设定它们的属性

变量声明

- 使用declare和typeset命令声明变量及其属性：

  declare [options] [name[=value]]

  typeset [options] [name[=value]]

  常用选项：

  -a 每一个“name”是一个数组

  -f 每一个“name”是一个函数

  -i “name”是一个整数

  -r 给每一个“name”标记上只读属性

  -x 表示每一个“name”都可以被子进程访问到

```
declare  -i age=20
declare –rx os=linux
echo $age
  20
```

```
echo $os
  linux
os=windows
  bash: os: readonly variable
```

变量声明（续）

- 使用不带参数的declare和typeset命令打印所有shell变量的名字和他们的值
  - $typeset

    #列出预先定义的变量和函数等
  - $declare –x

    declare -x BASH_ENV="/root/.bashrc"

    declare -x G_BROKEN_FILENAMES="1"

    declare -x HISTSIZE="1000"

    declare -x HOME="/root"

    declare -x HOSTNAME="jijiangmin“

    ……

使用变量

- 可以改变一个变量的值,一个整型变量不能赋予非整型的值，非整型的值变量可以被赋予任何值

```
$ declare –i age =20
$ echo $age
20
$ age=“Forty”
$ echo $age
   0
```

```
$ name=Jhon
$ declare place =Mars
$ echo $name $place
Jhon Mars
$ name=2019 place=007
$ echo $name $place
2019  007
```

读写shell变量--赋值语句

- 变量赋值
  - variable1=value1 [variable2=value2…variableN=valueN]
    - 用法:把值‘value1,…,valueN’相应地赋给变量 ‘variable1,…, variableN’。
    - 注意在变量、等号和值之间不能出现空格
- 变量名前加上$符号，可以读取一个变量的值，即变量的引用。
- 例：

```
$ myhome=/usr/app1
$ echo $ myhome
      /usr/app1
$ ls $ myhome          #app1是子目录
      file1 file2 file3 file4
```

一些其他的语法和操作也可以用来读取一个shell变量的值

表 命令替代操作符及其描述		
特殊字符	用途	描述
$variable	得到一个变量的值，如果没有被初始化则为null	返回’variable’的值，如果没有被初始化则返回null
${variable}	得到一个变量的值，如果没有被初始化则为null，通常使用在数组变量。	返回’variable’的值，如果没有被初始化则返回null
${variable:-string}	得到一个变量的值，如果这个变量没有被定义则返回一个确定的值	当’variable’存在而且不是空值的时候返回变量的值，否则返回’string’
${variable:=string}	如果一个变量没有被定义则把一个已知的值赋给他并返回这个值	当’variable’存在而且不是空值的时候返回变量的值，否则把’string’赋给’variable’并返回’string’
${variable:?string}	如果变量没有被定义则显示一条消息	当’variable’存在而且不是空值的时候返回变量的值，否则显示字符串’varibale:’并在其后显示’string’
${variable:+string}	测试一个变量存在与否	当’variable’存在而且不是空值的时候返回’string’，否则返回null

例

```
$ echo $name
```

变量是空值

```
$ name=David
$ echo $name
David
$ echo $place
```

注意：空行

变量是空值

```
$ echo ${name:-John} ${place:-Portland}
David Portland
$ echo ${place:?”Not defined.”}
bash: place: Not defined.
$ echo ${name:+”Defined”}
Defined
$ echo ${place:+”Not defined”}
```

变量存在但是空值，返回**null**

```
$ echo ${place:=”San Francisco”}
San Francisco
$ echo ${name:-John} ${place:-Portland}
David San Francisco
$
```

3. 表达式中 "" '' \ * 和空格的使用

```
$ name=John
$ echo $name
John
$ name=John Doe
Doe: not found
$ name="John Doe"
$ echo $name
John Doe
$ name=file*
$ echo $name
file file00 file01
```

如果给变量赋值的内容中包含空格，而且没有用引号，shell就会尝试把空格后的赋值内容作为一个命令去执行。

把*当作元字符，当前目录下file*格式的文件

表达式中 “” ’ ’ \ * 和空格的使用

```
$ echo “$name”
file*
$ echo “The name $name sounds familiar!”
The name file* sounds familiar!
$ echo \$name
$name
$ echo ‘$name’
$name
```

双引号

\

单引号

读写 Shell 变量（续）

- 如果用双引号“”将值括起来，则允许使用$符对变量进行替换。对大多数的元字符（包括*）都将按字面意思处理。
- 如果用单引号‘’将值括起来，则不允许有变量替换，而不对它做shell解释。
- 反斜杠\可以用来去除某些字符的特殊含义并把它们按字面意思处理，其中就包括$。

读写 Shell 变量（续）

${var}使用

- 例：

```
BOOK="English book"
msg=$BOOK_1
echo $msg
                #空值
msg=${BOOK}_1
echo $msg
        English book_1
```

4、输出环境变量--全局变量

- export、declare -x 和typeset -x命令把一个变量的值传递给子进程。
- delcare –x [name-list]

  typeset –x [name-list]

  export [name-list]

  用途：把 name-list中所有变量名字和它们的值输出到从这点开始执行的每个命令中
- 例：

```
$ cat disiplay_name
echo $name
exit 0
$ declare –x name= “John Doe”
$ ./display_name
John Doe
$ echo $?
0
```

命令退出状态的值

输出环境变量

```
$ cat export_demo
#! /bin/bash
declare –x name=“John Doe”
display_change_name
display_name
exit 0
$ cat display_change_name
#! /bin/bash
echo $name
name=“Plain Jane”
echo $name
exit 0
$ export_demo
John Doe
Plain Jane
John Doe
$
```

注意：上机实验时，用”**bash export_demo**”命令运行**bash**脚本

5、重设变量Resetting Variables

- 可以把一个变量的值重设为null（这是所有变量的默认初始值），这可以通过显式的将其设置为null或通过unset命令来实现。
- unset [name-list]

  用途：重设或删去name-list中列出的变量值或者函数，name-list中有多个值则用空格分开

```
$ name=John place=Corvallis
$ echo "$name $place"
John Corvallis
$ unset name
$ echo "$name"

$ echo "$place"
Corvallis
$
```

重设为null

对变量赋空值：

```
$ country=
$ echo "$country"

$
```

6. 创建只读的用户定义变量

- 符号常量，在bash中，在初始化的时候加上只读的标记，就定义了符号常量。
- declare –r [name-list]

  typeset –r [name-list]

  readonly [name-list]
  - 用途：阻止对name-list中列出的变量赋新的值
- 当readonly，decalre –r或者typeset –r命令不带参数执行的时候，将打印出所有的只读变量和它们的当前值。
- 不能重设只读变量的值，下面给出一个例子。

  $ unset name

  bash: unset: name: cannot unset: readonly variable

例

```
$ declare –r name=Jim place=Ames
$ echo $name $place
Jim Ames
$ name=Art place="Ann Arbor"
bash: name: readonly variable
bash: place: readonly variable
$
```

8.2 命令替换

命令替换

- 当一个字符串包含在一对括号前加“$”符号，如$(command)，或者被括在反引号“`”中，该字符串将作为命令被shell解释执行，即用命令的执行结果替换这个字符串本身。要注意反引号与单引号的区别。
- 语法： $(command) 或 `command`
  - 用途:用相应的命令替换$(command) 或 `command`

命令替换(续)

- 例：

```
$ command=pwd
$ echo “The value of command is: $command.”
The value of command is: pwd
$ command=$(pwd)
$ echo “The value of command is: $command.”
The value of comomand is: /home/faculty/ji/linuxbook/examples.
$ echo “The date and time is $(date).”
The date and time is Wed Jan 31 22:51:25 PST 2001.
```

8.3 从标准输入设备读入

从标准输入设备读入

- 使用read命令把用户的输入读入到一个shell变量中。
- read [options] variable-list
  - 用途：从标准输入设备读入一行，把读入的词赋在name-list中的变量
  - 常用选项/功能：
    - -a name　把词读入到name数组中去
    - -e 把一整行读入到第一个变量中，其余的变量均为null
    - -p prompt　如果是从终端读入数据则显示prompt字符串
- 读入的一行输入由许多词组成，他们是用空格（或者制表符，这取决于shell环境变量*IFS*的值）分隔开的。
- 如果这些词的数量比列出的变量的数量多，则把余下的所有词赋值给最后一个变量。
- 如果列出的变量的数量多于输入的词的数量，这多余的变量的值被设置为null。

从标准输入设备读入

```
$ cat read_demo
#! /bin/bash
echo –n  “Enter input: ”
read line
echo “You entered: $line”
echo –n  “Enter another line: ”
read word1 word2 word3
echo “The first word is: $word1”
echo “The second word is: $word2”
echo “The rest of the line is: $word3”
exit 0
$
```

-n选项，用来表示在输出echo后的字符串后，光标仍然停留在同一行

```
$ read_demo
Enter input: LINUX rules the network computing world!
You entered: LINUX rules the network computing world!
Enter another line: LINUX rules the network computing world!
The first word is: LINUX
The second word is: rules
The rest of the line is: the network computing world!
$
```

8.4 给Shell脚本传递参数

给Shell脚本传递参数

- 给一个shell脚本传递命令行参数或位置参数。这些变量中的前九个的值被存放在shell变量*$1*到*$9*中。
- 环境变量$#包含了传递给一个执行中的shell脚本的参数的个数。
- 变量$*和$@都包含了所有参数的值。他们的区别在于，当被引号包括的时候，即“$@”，$@把每个参数的值做为一个字符串，而不是把所有的值作为一个字符串。
- 变量*$0*包含了脚本文件的名字（命令的名字）。
- 语法：shift [N]

  用途：把命令行的参数向左移动N个位置

给Shell脚本传递参数(续)

```
$ cat cmdargs_demo
    #! /bin/bash
    echo “The command name is: $0.”
    echo “The number of command line arguments passed as parameters are $#.”
    echo “The value of the command line arguments are: $1 $2 $3 $4 $5 $6 $7 $8 $9.”
    echo “Another way to display values of all of the arguments: $@.”
    echo “Yet another way is: $*.”
    exit 0
$ cmdargs_demo a b c d e f g h i
    The command name is: cmdargs_demo.
    The number of command line arguments passed as parameters are 9.
    The value of the command line arguments are: a b c d e f g h i.
    Another way to display values of all of the arguments: a b c d e f g h i.
    Yet another way is: a b c d e f g h i.
$ cmdargs_demo One Two 3 Four 5 6
    The command name is: cmdargs_demo.
    The number of command line arguments passed as parameters are 6.
    The value of the command line arguments are: One Two 3 Four 5 6 .
    Another way to display values of all of the arguments: One Two 3 Four 5 6.
    Yet another way is: One Two 3 Four 5 6.
```

给Shell脚本传递参数(续)

```
$ cat shift_demo
        #! /bin/bash
        echo “The program name is $0.”
        echo “The arguments are: $@”
        echo “The first three arguments are: $1 $2 $3”
        shift
        echo “The program name is $0.”
        echo “The arguments are: $@”
        echo “The first three arguments are: $1 $2 $3”
        shift 3
        echo “The program name is $0.”
        echo “The arguments are: $@”
        echo “The first three arguments are: $1 $2 $3”
        exit 0
$ shift_demo 1 2 3 4 5 6 7 8 9 10 11 12
        The program name is shift_demo.
        The arguments are: 1 2 3 4 5 6 7 8 9 10 11 12
        The first three arguments are: 1 2 3
        The program name is shift_demo.
        The arguments are: 2 3 4 5 6 7 8 9 10 11 12
        The first three arguments are: 2 3 4
        The program name is shift_demo.
        The arguments are: 5 6 7 8 9 10 11 12
        The first three arguments are: 5 6 7
```

给shell脚本传递参数(续)

- set命令可以用来修改位置参数的值。

句法：	**set [options][argument-list]**

用途：设置标志，选项和位置参数；当不带参数使用时，打印出所有shell变量（环境变量和用户定义的变量）的名字和当前值；使用在argument-list中的值来设置位置参数。

常用选项/功能：

--　　不把以’-’开头的词作为参数选项

-C　　把noclobber标志设置为on，不允许用户通过重定向覆盖已经存在的文件，如果一定要覆盖，应该用>|代替>。

-a　　在赋值的时候自动把变量输出（export）

-o [option]　设置options；当option为空的时候，打印出目前shell的设置。

其中的一些option是：

hash　在内部的哈希表中保存命令的位置（这是默认的设置，可以用-h选项来设置）

history　允许保存历史纪录

noclobber 不允许用输出重定向来覆盖已有的文件（和-C同样的作用）

-v　　详细模式：显示隐含的输入参数

给Shell脚本传递参数(续)

- 例：

```
$ date
2020年 12月 23日 星期三 10:51:03 CST
$ set $(date)
$ echo “$@”
2020年 12月 23日 星期三 10:51:03 CST
$ echo “$1 $4”
2020年 星期三
$set $(ls )
```

- **date命令执行的结果有6个域。set $(date)命令把date命令的输出设置为位置变量。**
- ***$1*被设置为2020年，*$2*被设置为12月，*$3*被设置为23日，*$4*对应星期三，*$5*对应10:51:03，最后，*$6*对应CST。echo “$@”命令显示所有位置参数的值。**

给Shell脚本传递参数(续)

```
$ cat set_demo
#! /bin/bash
filename="$1"
set -- $(ls –il $filename)
inode="$1"
size="$6"
echo "Name Inode  Size"
echo
echo "$filename $inode  $size"
exit 0
$ set_demo lab3
Name Inode  Size
lab3  856110 591
$
```

告诉set命令，如果第一个参数的第一个字符是-，它不应被视为set命令的一个选项。在这条命令不是必须，如：**set -- $(ls -l a.out)**命令中必须要加“--”

例：

- 编写一个Bash脚本，它显示出所有的命令行参数。把它们都左移两位，并再次显示所有的命令行参数。

```
$ cat ch15ex7
echo "$@"
shift 2
echo "$@"
$ ch15ex7 Hello, World!
Hello, World!

$ ch15ex7 One 2 three four
One 2 three four
three four
```

8.5 Source、点（.）和null命令

Source和点（.）命令

- 语法：

  source filename

  . filename

  - 使用当前shell程序执行fileName中的命令。通常使用在shell脚本程序中

- 例：

```
if [ -f ~/.bashrc ] ; then
    . ~/.bashrc
fi
```

null命令

- null命令的命令名为冒号（:），运行时什么也不做，退出状态总是0(正确退出)。它是一个内置命令。
- 例：判断当前目录下是否有file.txt文件。

```
$ cat null.sh
#!/bin/bash
if [ -f file.txt ]
then
   :
else
    echo “This doesn’t exit”
fi
```

8.6 内置字符串操作

字符串操作

表达式	含义	实例
${#variable} #计算字符串长度	返回字符串variable的长度	$foo= /home/ji/demo/2020 $echo ${#foo} 18
${variable:offset:length} #提取子串	返回变量variable的子字符串，从位置offset开始（从0开始计算），长度为length。	$foo= /home/ji/demo/2020 $echo ${foo:1:4} home $foo= /home/ji/demo/2020 $echo ${foo: -4} #-4前有空格 2020 $echo ${foo: -9:4} demo
${variable#pattern} #字符串头部的模式匹配(非贪婪匹配)	如果 pattern 匹配变量 variable 的开头，删除最短匹配（非贪婪匹配）的部分，返回剩余部分	$foo= /home/ji/demo/2020/file.12.txt $ echo ${foo#/*/} ji/demo/2020/file.12.txt
${variable##pattern} #字符串头部的模式匹配(贪婪匹配)	如果 pattern 匹配变量 variable 的开头，删除最长匹配（贪婪匹配）的部分，返回剩余部分	$foo= /home/ji/demo/2020/file.12.txt $echo ${foo##/*/} file.12.txt

字符串操作

表达式	含义	实例
${variable%pattern} #字符串尾部的模式匹配(非贪婪匹配)	如果 pattern 匹配变量 variable 的结尾，删除最短匹配（非贪婪匹配）的部分，返回剩余部分	$foo= /home/ji/demo/2020/file.12.txt $echo ${foo%.*} /home/ji/demo/2020/file.12
${variable%%pattern} #字符串尾部的模式匹配(贪婪匹配)	如果 pattern 匹配变量 variable 的结尾，删除最长匹配（贪婪匹配）的部分，返回剩余部分	$foo= /home/ji/demo/2020/file.12.txt $echo ${foo%%.*} /home/ji/demo/2020/file
${variable/pattern/string} #字符串任意替换(非贪婪匹配)	如果 pattern 匹配变量 variable 的一部分，最短匹配（非贪婪匹配）的那部分被 string 替换，但仅替换第一个匹配	$foo= /home/ji/demo/2020/file.12.txt $echo ${foo/./-} /home/ji/demo/2020/file-12.txt
${variable//pattern/string} #字符串任意替换(贪婪匹配)	如果 pattern 匹配变量 variable 的一部分，最长匹配（贪婪匹配）的那部分被 string 替换，所有匹配都替换	$foo= /home/ji/demo/2020/file.12.txt $echo ${foo//./-} /home/ji/demo/2020/file-12-txt

字符串操作

表达式	含义	实例
${variable/#substring/replacement} #字符串头部替换	如果substring匹配variable的开头部分，那么就用replacement来替换substring.	$f1=file.12.txt $echo ${f1/#file/file001} file001.12.txt
${variable/%substring/replacement} #字符串尾部替换	如果substring匹配variable的结尾部分，那么就用replacement来替换substring.	$f1=file.12.txt $echo ${f1/%txt/doc} file.12.doc
${varname^^}	转换为大写	$foo=heLLo $echo ${foo^^} HELLO
${varname,,}	转换为小写	$foo=heLLo $echo ${foo,,} hello

8.7 程序控制流程语句

程序控制流程命令

- 用程序控制命令/语句来决定shell脚本执行时的各个语句执行顺序
- 三类基本控制程序流程的命令：
  - 二路分支Two-way branching
  - 多路分支Multiway branching
  - 重复执行
- 在bash中，二路分支的语句是if语句，多路分支是if和case语句，重复执行的语句是for、while和until语句。

一、If语句

基本if语句

- 基本if-then语句有如下格式：

```
if command
then
    commands
fi
```

- bash shell的if语句会运行if后面的那个命令。如果该命令的退出状态码是0（该命令成功运行），位于then部分的命令就会被执行。如果该命令的退出状态码是其他值， then部分的命令就不会被执行，bash shell会继续执行脚本中的下一个命令。fi语句用来表示if-then语句到此结束。

基本if语句

- $cat if_demo0

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

1. if语句

If-then-elif-else-fi 语句

```
if expression
    then
        then-commands
fi
```

Purpose: To implement two-way branching

```
if expression
    then
      [elif expression
      then
        then-command-list]
      ...
      [else
        else-command-list]
fi
```

Purpose: To implement two-way or multiway branching

1. if语句

- test命令，检测一个表达式并返回true还是false

test [expression]
Or
[[expression]]

Purpose: To evaluate 'expression' and return true or false status

里面的方括号用来表示中间是一个可选的表达式，外面的方括号则表示test命令。在操作数和操作符或者括号的前后都要至少留一个空格。

Operators for the test Command

表 一些有用的test命令的操作符

文件测试		整数测试		字符串测试	
表达式	返回值	表达式	返回值	表达式	返回值
-d file	如果这个文件是一个目录，返回true	int1 –eq int2	如果int1等于int2，则返回true	str	如果str是一个非空字符串，则返回true
-f file	如果这个文件是一个普通文件，返回true	int1 –ge int2	如果int1大等于int2，则返回true	str1 = str2	如果str1等于str2，则返回true
-r file	如果这个文件是只读，返回true	int1 –gt int2	如果int1大于int2，则返回true	str1 != str2	如果str1不等于str2，则返回true
-s file	如果这个文件的长度非0，返回true	int1 –le int2	如果int1小等于int2，则返回true	-n str	如果str的长度大于0，则返回true
-t [filedes]	如果文件描述符filedes是联系着终端的，返回true	int1 –lt int2	如果int1小于int2，则返回true	-z str	如果str的长度为0，则返回true
-w file	如果这个文件是可写，返回true	int1 –ne int2	如果int1不等于int2，则返回true	str1 \< str2	如果str1小于str2，则返回true
-x file	如果这个文件是可执行，返回true	==	相等	str1 \> str2	如果str1大于str2，则返回true
-b file	如果这个文件是块特殊文件，返回true	!=	不等		
-c file	如果这个文件是字符特殊文件，返回true				
-e file	如果这个文件存在，返回true				
-L file	如果这个文件存在而且是一个符号链接文件，返回true				

Operators for the test Command

结合成更复杂的表达式的操作符			
!	逻辑非操作符：当后面的expression为false的时候返回true	(‘expression’)	用于把表达式分组的括号，在每个括号的前后至少要有一个空格
-a	逻辑与操作符：当前后的表达式均为true时才返回true	-o	逻辑或操作符：如果左边或者右边的表达式为true，则返回true

- $? 退出状态

[ ]测试：如果条件满足返回为0 ，否则返回1

Example Script

```
$ cat if_demo1
#! /bin/bash
if test $# -ne 1
      then
            echo “Usage: $0 ordinary_file”
            exit 1
fi
if test –f “$1”
      then
            filename=“$1”
            set $(ls –il $filename)
            inode=“$1”
            size=“$6”
            echo “Name Inode  Size”
            echo
            echo “$filename $inode $size”
            exit 0
fi
echo “$0: argument must be an ordinary file”
exit 1
```

Example Script

```
$ if_demo1
Usage: if_demo1 ordinary_file
$ if_demo1 lab3 lab4
Usage: if_demo1 ordinary_file
$ if_demo1 dir1
if_demo1: argument must be an ordinary file
$ if_demo1 lab3
Name Inode  Size
lab3 856110  591
$
```

if语句（第二种if语句 ）

```
if expression
    then
        then-command
    else
        else-command
fi
```

Purpose: To implement two-way branching

Example Script

```
$ cat if_demo2
#! /bin/bash
if [ $# -ne 1 ]
    then
        echo “Usage: $0 ordinary_file”
        exit 1
fi
if [ -f “$1” ]
    then
        filename=“$1”
        set $(ls –il $filename)
        inode=“$1”
        size=“$6”
        echo “Name  Inode  Size”
        echo
        echo “$filename  $inode  $size”
        exit 0
    else
        echo “$0: argument must be an ordinary file”
        exit 1
fi
```

在操作数和操作符或者括号的前后都要至少留一个空格。

Example Script

- 运行结果：

```
$ bash if_demo2 ifdemo
Name  Inode  Size

ifdemo  2252449  312
[root ~]
```

if语句（第三种if语句 ）

句法：	if expression1 then then-commands elif expression2 then elif1-commands elif expression3 then elif2-commands … else else-commands fi
用途：实现多路跳转	

Example Script

```
$ cat if_demo3
#! /bin/bash
if [ $# -ne 1 ]
      then
            echo “Usage: $0 file”
            exit 1
      else
            ls “$1” 2> /dev/null 1>&2
            if [ $? –ne 0 ]
            then
                  echo “$1: not found”
                  exit 1
            fi
            if [ -f “$1” ]
            then
                  filename=“$1”
                  set $(ls –il $filename)
                  inode=“$1”
                  size=“$6”
                  echo “Name  Inode  Size”
                  echo
                  echo “$filename  $inode  $size”
                  exit 0
```

命令用来检查从命令行输入的文件是否存在。标准错误输出被重定向到/dev/null（Linux中的黑洞），而1>&2把标准输出重定向到标准错误输出。这样，命令将不会产生任何的输出或错误信息

错误返回

正确返回

Example Script

```
    elif    [ -d "$1" ]
            then
                    nfiles=$(ls "$1" | wc –w)
                    echo "The number of files in the directory is $nfiles."
                    exit 0
    else
            echo "$0: argument must be an ordinary file or directory"
            exit 1
    fi
fi
```

```
$ if_demo3 file1
file1: not found
$ if_demo3 dir1
The number of files in the directory is 4.
$ if_demo3 lab3
Name  Inode  Size
lab3  856110  591
$
```

二、Exit命令

exit命令

- exit :退出脚本

  exit 0

Linux退出状态码

退出码	描述
0	命令成功结束
1	一般性未知错误
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
128+x	与Linux信号x相关的严重错误
130	通过Ctrl+C终止的命令
255	正常范围之外的退出状态码

三、复合条件测试

复合条件测试

- 布尔运算符：
  - [condition1] && [condition2]
  - [condition1] || [condition2]
- -a 和 -o 作为测试命令的参数用在测试命令的内部， && 和 || 则用来运算测试的返回值，！为两者通用。

复合条件测试

- $ cat if_demo4

```
#!/bin/bash
# testing compound comparisons
#
if [ -d $HOME ] && [ -w $HOME/testing ]
then
        echo "The file exists and you can write to it"
else
        echo "I cannot write to the file"
fi
```

```
$ ./if_demo4
  I cannot write to the file

$ touch $HOME/testing

$ ./if_demo4
  The file exists and you can write to it
```

四、if-then 的高级特性

if-then 的高级特性

- 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**双圆括号**
  - 用于高级字符串处理功能的**双方括号**

(1) 使用双括号

- 双括号命令允许在比较过程中使用高级数学表达式。
- 双括号命令的格式如下：

((expression))

  - expression可以是任意的数学赋值或比较表达式。

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

双括号命令符号

使用双括号

$ cat if_demo5

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

```
$ ./if_demo5
The square of 10 is 100
```

(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。双方括号命令的格式如下：

  [[expression]]

  - 双方括号里的expression可以使用了test命令中采用的标准字符串比较。它也能使用正则表达式。

(2) 使用双方括号

```
$ cat if_demo6
    #!/bin/bash
    # using pattern matching
    if [[ $USER == j* ]]
    then
        echo "Hello $USER"
    else
        echo "Sorry, I do not know you"
    fi
```

环境变量

```
$ ./if_demo6
Hello ji
```

五、 for 语句

for 语句

```
for variable [in argument-list]
do
   command-list
done
```

Purpose: To execute commands in 'command-list' as many times as the number of words in the 'argument-list'; without the optional part, 'in argument-list', the arguments are supplied at the command

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
```

- 在list参数中，需要提供迭代中要用到的一系列值。

for 命令

- for语句的for in遍历结构实例：

```
$ cat for_demo1
    #! /bin/bash
    for x in $(date)
    do
        echo “$x”
    done
    exit 0

$ for_demo1
    2020年
    05月
    01日
    星期五
    11ah
```

for 命令

- **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。文件扩展匹配是生成匹配指定通配符的文件名或路径名的过程。
- $ cat for_demo10

```
!/bin/bash
#遍历文件和目录
f=0 d=0
for file in ./*
do
  if [ -d "$file" ]
  then
      echo "$file is a directory“
      ((d++))
  elif [ -f "$file" ]
  then
      echo "$file is a file“
      ((f++))
  fi
Done
echo "文件有$f个"
echo "目录有$d个"
```

for 命令

- 遍历的对象来自于命令行参数
- $ cat user_info

```
#!/bin/bash
for user
do
    # Don't display anything if a login name is not in /etc/passwd
    grep “^$(echo $user):" /etc/passwd 1> /dev/null 2>&1
    if [ $? -eq 0 ]
     then
        echo -n "$user: "
        grep "^$(echo $user):" /etc/passwd | cut –f6 -d':'
    fi
done
exit 0
```

user的值来自命令行参数

把一系列已经存在的登录用户的名字作为命令行参数，并打印出每个登录名及其相应的拥有这个登录名的用户的全名。

C 语言风格的for命令

- for语句的另一种结构：

```
for ((a=1; a<=limit; a++ ))
do
    command
done
```

C 语言风格的for命令

- 利用 for 的功能，计算 1 到 100 的和。

$ cat for_demo0

```
#!/bin/bash
for (( i=1; i<=100; i++ ))
do
        s=$(($s+$i))
done
echo "the count is ==> $s"
```

注意，有些部分并没有遵循bash shell标准的for命令：

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

六、 while和until 语句

6. while 语句

SYNTAX

```
while expression
do
  command-list
done
```

Purpose: To execute commands in 'command-list' as long as 'expression' evaluates to true

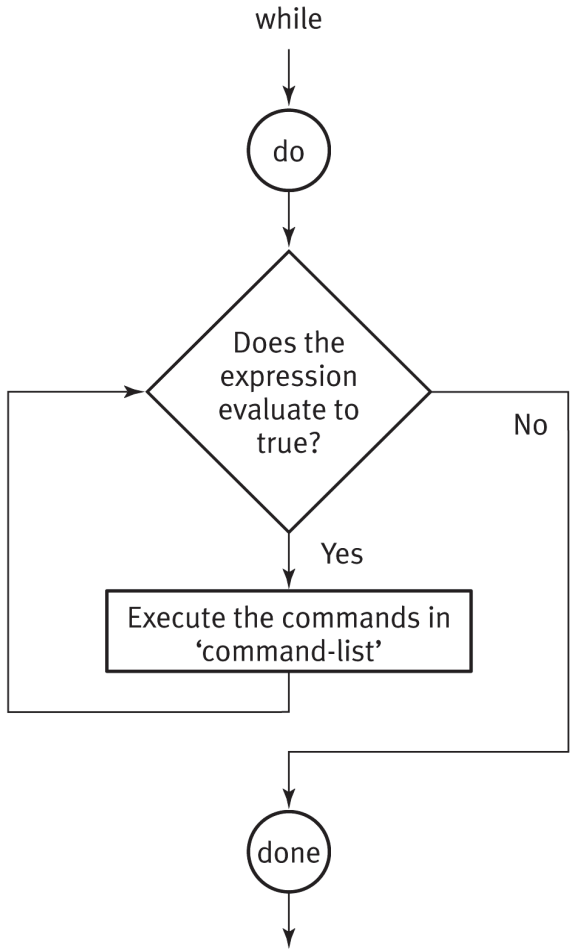

Figure 15.5 Semantics of the `while` statement

The while statement

```
$ cat while_demo
#! /bin/bash
secretcode=agent007
echo "Guess the code! "
echo -n "Enter your guess: "
read yourguess
while [ "$secretcode" != "$yourguess" ]
do
    echo "Good guess but wrong.  Try again! "
    echo -n "Enter your guess: "
    read yourguess
done
echo "Wow! You are a genius! "
exit 0
$ while_demo
Guess the code!
Enter your guess: star wars
Good guess but wrong. Try again!
Enter your guess: columbo
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!
```

The until Statement

SYNTAX

```
until expression
do
   command-list
done
```

Purpose: To execute commands in 'command-list' as long as 'expression' evaluates to false

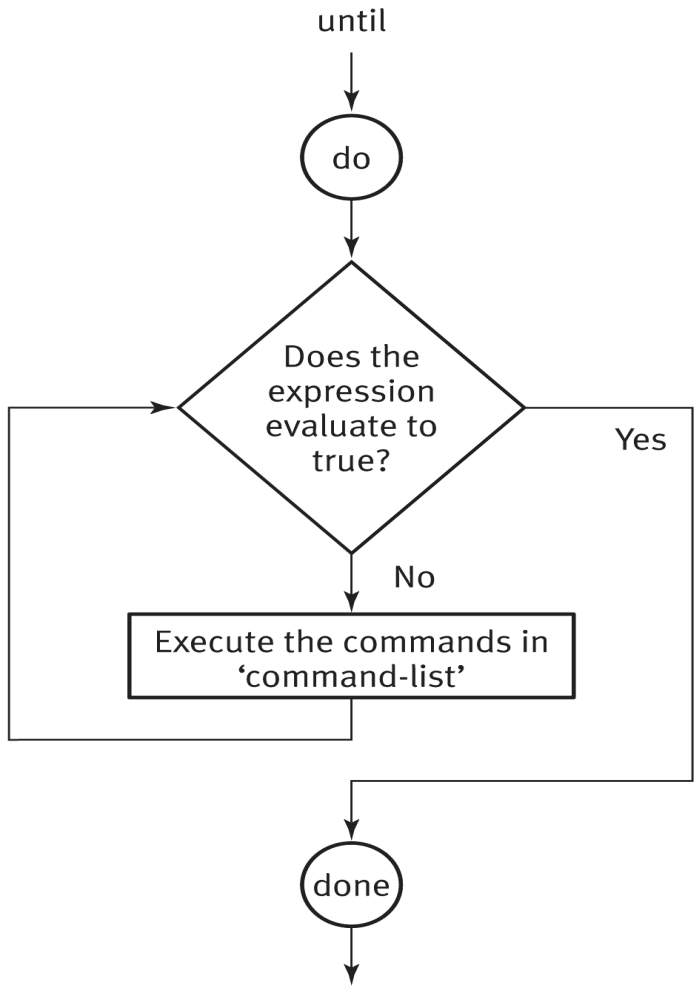

Figure 15.6 Semantics of the `until` statement

The until Statement

```
#! /bin/bash
secretcode=agent007
echo "Guess the code! "
echo -n "Enter your guess: "
read yourguess
until [ "$secretcode" = "$yourguess" ]
do
    echo "Good guess but wrong.  Try again! "
    echo -n "Enter your guess: "
    read yourguess
done
echo "Wow! You are a genius! "
exit 0
```

```
$ until_demo
Guess the code!
Enter your guess: Inspector Gadget
Good guess but wrong. Try again!
Enter your guess: Peter Sellers
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!!
$
```

7.break和continue命令

7. break和continue命令

- break命令和continue命令用来打断循环体的执行。
- break命令使得程序跳出循环,执行done后面的语句，这样就永久终止了循环。
- continue命令使得程序跳到done，这使得循环条件被再次求值，从而开始新的一次循环。

The `break` and `continue` Statements

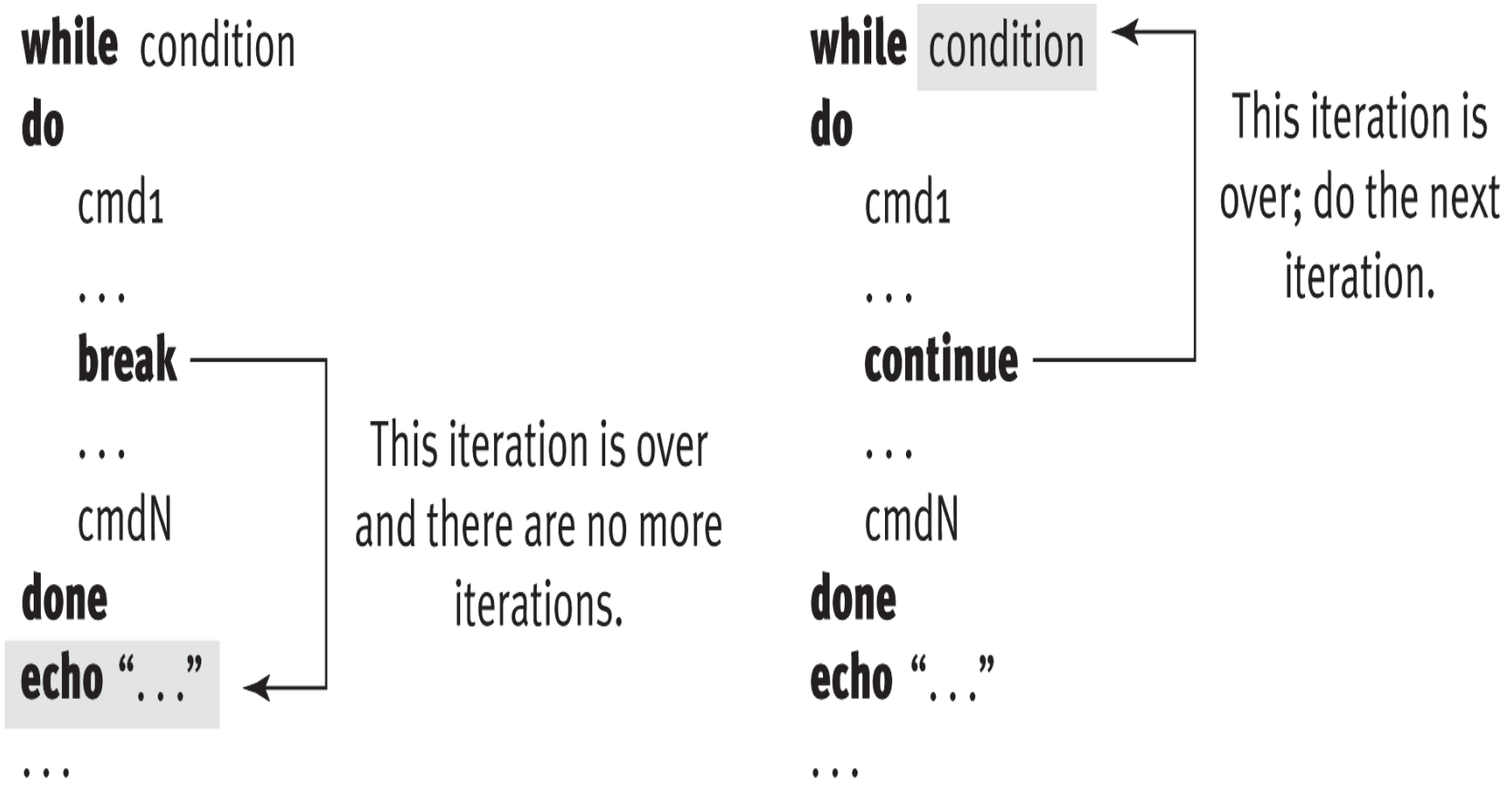

Figure 15.7 Semantics of the `break` and `continue` commands

8. case 语句

8. case 语句

- case语法结构

```
case test-string in
  pattern1）command-list1
          ;;
  pattern2）command-list2
          ;;
  …
  patternN）command-listN
          ;;
esac
```

The case Statement

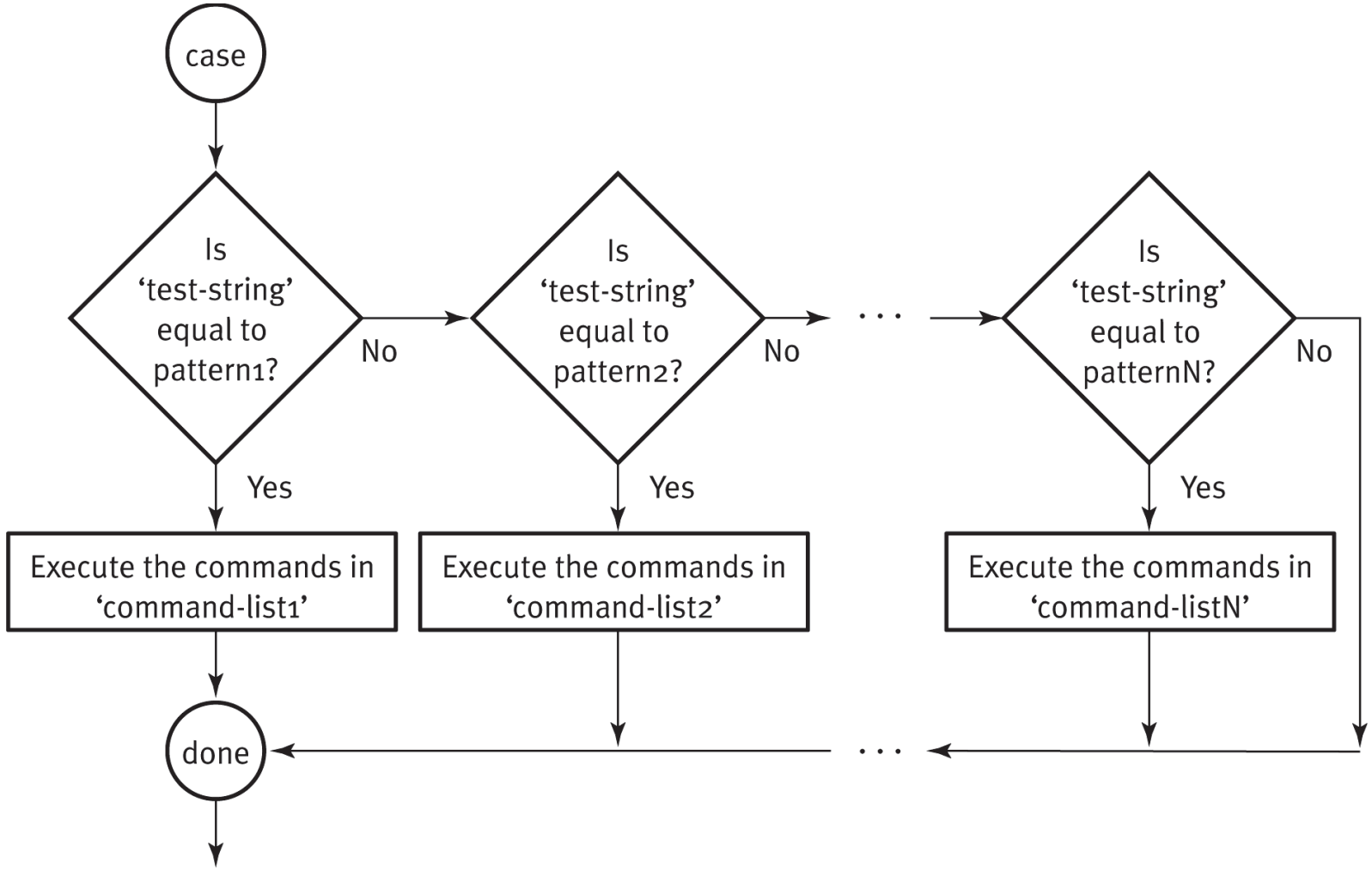

Figure 15.8 Semantics of the case statement

The case Statement

- case_demo脚本

```
#!/bin/bash
echo "Use one of the following options:"
echo "d or D:      To display today's date and present time"
echo "l or L:      To see the listing of files in your present working directory"
echo "w or W:      To see who is logged in"
echo "q or Q:      To quit this program"
echo -n "Enter your option and hit the <Enter> key: "
read option
case "$option" in
    d|D)            date
       ;;
    l|L)            ls
       ;;
    w|W)            who
       ;;
    q|Q)            exit 0
       ;;
    *) echo "Invalid option; try running the program again."
       exit 1
       ;;
esac
exit 0
```

The case Statement

```
$ case_demo
Use one of the following options:
        d or D:   To display today's date and present time
        l or L:   To see the listing of files in your present working directory
        w or W:   To see who is logged in
        q or Q:   To quit this program
Enter your option and hit <Enter>: D
Sat June 12 18:14:22 PDT 2004
$ case_demo
Use one of the following options:
        d or D:   To display today's date and present time
        l or L:   To see the listing of files in your present working directory
        w or W:   To see who is logged in
        q or Q:   To quit this program
Enter your option and hit <Enter>: a
Invalid option; try running the program again.
$
```

8.8 数值数据处理

数值数据处理

- 所有bash变量的值都以字符串方式存储。
- 如果需要对其进行算术和逻辑操作，必须先转换为整数，得到运算结果后再转回字符串，以便正确地保存于Shell变量中。

数值数据处理(续)

- bash提供多种数值数据进行算术运算的方法：
  - 使用let语句
  - 使用shell扩展 $((expression))
  - 使用expr命令
  - 使用方括号 $[operation]
  - 使用双圆括号 ((operation))
- 表达式求值以长整数进行，并且不作溢出检查
- bash运算符如表所示
- 八进制数以0为首的数值，十六进制数以0x或0X为首的数值

Arithmetic Operations Supported by Bash

表	Bash支持的算术运算符
运算符	含义
- +	一元运算（正负号）
! ~	逻辑非；补
**	指数
* / %	乘；除；取余数
+ -	加；减
<< >>	左移；右移
<= >= <>	小于等于；大于等于；不等于
== !=	等于；不等于
&	按位与
^	按位异或（XOR）
|	按位或
&&	逻辑与
||	逻辑或
= += -= *= /= &= ^= |= <<= >>=	赋值运算符：简单赋值；加赋值；减赋值；乘赋值；除赋值；与赋值；异或赋值；或赋值；左移赋值；右移赋值

1. let语句

- let语句
  - 语法：let express-list
  - 用途：求出算术表达式的值
- 例：

```
$ let “a = 8” “b = 13”  #若表达式有空格，要使用引号
$ let c=a+b
$ echo “the value of c is $c”
the value of c is 21
$ let “a *= b”
$ echo “The new value of a is $a; the product of a and b.”
The new value of a is 104; the product of a and b.
```

2. shell扩展$((expression))

- 使用shell扩展$((expression))
  - 语法： $((expression))
  - 用途：计算expression并返回它的值
- 例：

```
$ a=8 b=13
$ echo $((a*5+b))
53
$ echo “the new value of c is $((a+b))”
the new value of c is 21
$ echo “The new value of a is $((a*=b)); the product of a and b.”
The new value of a is 104; the product of a and b.
```

3. expr命令

- expr命令
  - 语法：expr　args
  - 用途:计算表达式的参数args的值，并返回他的值到标准输出
  - 常用选项/功能：

    算术、关系、逻辑运算符是最常用的。

expr命令操作符

- ARG1 | ARG2 如果ARG1既不是null也不是零值，返回ARG1；否则返回ARG2
- ARG1 & ARG2 如果没有参数是null或零值，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小于或等于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
- ARG1 >= ARG2 如果ARG1大于或等于ARG2，返回1；否则返回0
- ARG1 > ARG2 如果ARG1大于ARG2，返回1；否则返回0

expr命令操作符

- ARG1 + ARG2　返回ARG1和ARG2的算术运算和
- ARG1 - ARG2　返回ARG1和ARG2的算术运算差
- ARG1 * ARG2　返回ARG1和ARG2的算术乘积
- ARG1 / ARG2　返回ARG1被ARG2除的算术商
- ARG1 % ARG2　返回ARG1被ARG2除的算术余数

expr命令操作符

- STRING : REGEXP 如果REGEXP匹配到了STRING中的某个模式，返回该模式匹配
- match STRING REGEXP 如果REGEXP匹配到了STRING中的某个模式，返回该模式匹配
- substr STRING POS LENGTH 返回起始位置为POS（从1开始计数）、长度为LENGTH个字符的子字符串
- index STRING CHARS 返回在STRING中找到CHARS字符串的位置；否则，返回0
- length STRING 返回字符串STRING的数值长度
- + TOKEN 将TOKEN解释成字符串，即使是个关键字
- (EXPRESSION) 返回EXPRESSION的值

数值数据处理(续)

```
$ expr 1 + 2
$ 3
$ var1=10
$ expr $var1 + 1
11

$ var1=`expr $var1 \* $var1`
$ echo $var1
121
$ echo `expr $var1 / 10`
12
$ echo `expr $var1 % 10`
1
```

注意：运算符两边需要保留空格

数值数据处理(续)

- expr 关系运算结果，0 代表假，1 代表真：

```
$ i=6 j=8
$ expr $i = $j
0                #6 等于8 是假的
$ expr $i != $j
1                #6 不等于8 是真的
```

4. 使用方括号

- $[operation] 在bash中，在将一个数学运算结果赋给某个变量时，可以用美元符和方括号将数学表达式围起来。

```
$ var1=$[1 + 5]
$ echo $var1
6
$ var2=$[$var1 * 2]
$ echo $var2
12
$ var3=$[$var1 * ($var2 - $var1)]
$ echo $var3
36
```

数值数据处理(续)

```
$ cat countup
#!/bin/sh
if [ $# != 1 ]
   then
      echo "Usage: $0 integer-argument"
      exit 1
fi
target="$1"    # Set target to the number passed at the command line
current=1      # The first number to be displayed

# Loop here until the current number becomes greater than the target
while [ $current -le $target ]
do
      echo "$current \c"
      current=`expr $current + 1`
done
echo
exit 0
$ countup 5
1 2 3 4 5
$
```

数值数据处理(续)

- addall脚本

```
if [ $# = 0 ]
    then
     echo "Usage: $0 number-list"
     exit 1
fi
sum=0      # Running sum initialized to 0
count=0  #计算传入的参数的个数
```

数值数据处理(续)

```
while [ $# != 0 ]
do
    sum=`expr $sum + $1` #将下一个数加到当前的sum上
    if [ $? != 0 ]       #如果expr命令由于非整数的参数而失败，在此退出
     then
         exit 1
    fi
    count=$((count+1))       #更新目前已经累加的数字计数
    shift                    #将累加过的数字移走
done
echo "The sum of the given $count numbers is $sum."
exit 0
```

```
$add all
Usage: addall number-list
$add all 1 4 9 16 25 36 49
The sum of the given $count numbers is 140
$
```

8.9 数组处理

数组处理

- bash支持一维数组。
- 数组的元素是编号的，第一个元素是0号。
- 数组的大小没有限制，数组的元素不必连续赋值。
- 数组定义：
  - 语法：array_name[subscript]=value

    declare –a array_name

    declare –a array_name[subscript]

    local –a array_name

    readonly –a array_name
  - 用途：声明array_name为数组变量。下标是取值大于等于0的算术表达式。第二个和第三个句法是等价的，第二个声明中的下标被忽略。第四个句法在函数中用来声明一个函数内的局部数组变量。第五个句法用来声明只读数组。

数组处理

- 可以用${name[subscript]}引用数组中的元素。如果subscript是@或*，则数组中所有元素都被引用。
- 例：

注意括号

```
$ movies=(“Silence of the Lambs” “Malcolm X” [65]=“The Birds”)
$ echo ${movies[0]}
Silence of the Lambs
$ echo ${movies[1]}
Malcolm X
$ echo ${movies[2]}

$ echo ${movies[65]}
The Birds
$ echo ${movies[*]}
Silence of the Lambs Malcolm X The Birds
$ echo ${movies[@]}
Silence of the Lambs Malcolm X The Birds
$
```

数组处理

- 在下例中，files数组变量包含了当前目录中所有的文件名。numfiles变量包含了当前目录中的文件数目。文件的数目也可以用echo ${#files[*]}命令显示。

```
$files=(`ls`) ; numfiles=`ls | wc -w`
$echo ${files[*]}
cmdargs_demo foreach_demo1 if_demo1 if_demo_2 keyin_demo
$echo $numfiles
5
$echo ${#files[*]}
5
$echo $files[3]
if_demo1
```

括号

数组元素个数

8.10 函数

Bourne Shell中的函数

- bash中允许使用函数，函数由被称为函数体的一系列命令组成。
- 用函数名可以调用函数体中的命令。
- 函数定义的两种格式为：

```
function_name()
{
command-list
}
```

  - function_name是你选取的函数名。command-list中的命令为函数体。左括号{可以与函数名放在同一行。

```
function function_name()
{
command-list
}
```

Bourne Shell中的函数

```
$ machines ()
> {
> date
> echo “These are the machines on the network:”
> ruptime | cut -f1 -d’ ‘ | more
> }
$ machines
Thu Feb 19 17:05:00 PDT 2004
These are the machines on the network:
upibm0
...
upsun1
...
upsun29
$
```

End of the chpater