

---

```

if read -t 10 -sp "Enter secret passphrase > " secret_pass; then
    echo -e "\nSecret passphrase = '$secret_pass'"
else
    echo -e "\nInput timed out" >&2
    exit 1
fi

```

---

上述脚本会提示用户输入秘密通行短语，系统的等待时间为 10 秒。若 10 秒内用户没有完成输入，脚本以出错状态结束运行。又因为使用了-s 选项，输入的密码并不会显示到屏幕上。

### 28.1.2 使用 IFS 间隔输入字段

通常 shell 会间隔提供给 read 命令的内容。这也就意味着，在输入行，由一个或多个空格将多个单词分隔成为分离的单项，再由 read 命令将这些单项赋值给不同的变量。此行为是由 shell 变量 IFS（Internal Field Separator）设定的。IFS 的默认值包含了空格、制表符和换行符，每一种都可以将字符彼此分隔开。

我们可以通过改变 IFS 值来控制 read 命令输入的间隔方式。例如，文件 /etc/passwd 的内容使用冒号作为字段之间的间隔符。将 IFS 的值改为单个冒号即可使用 read 命令读取/etc/passwd 文件的内容，并成功将各字段分隔为不同的变量。下面的脚本就完成了此功能。

---

```

#!/bin/bash

# read-ifs: read fields from a file

FILE=/etc/passwd

read -p "Enter a username > " user_name

file_info=$(grep "^$user_name:" $FILE) ①

if [ -n "$file_info" ]; then
    IFS=: read user pw uid gid name home shell <<< "$file_info" ②
    echo "User =      '$user'"
    echo "UID =       '$uid'"
    echo "GID =       '$gid'"
    echo "Full Name = '$name'"
    echo "Home Dir. = '$home'"
    echo "Shell =     '$shell'"
else
    echo "No such user '$user_name'" >&2
    exit 1
fi

```

---

此脚本提示用户输入系统账户的用户名，根据用户名找到/etc/passwd 文件中相应的用户记录，并输出此记录的各个字段。此脚本包含了两行有趣的代码。

第一行位于<sup>①</sup>，它将 grep 命令的结果赋值给 file\_info 变量。grep 命令使用的正则表达式保证了用户名只会与/etc/passwd 文件中的一条记录相匹配。

第二行这有趣的代码位于<sup>②</sup>，是由三部分构成的，即一条变量赋值语句一条带有变量名作参数的 read 命令和一个陌生的重定向运算符。首先让我们看一下其中的变量赋值语句。

shell 允许在命令执行之前对一到多个变量进行赋值，这些赋值操作会改变接下来所执行命令的操作环境。但是赋值的效果是暂时性的，只有在命令执行周期内有效。本例中，IFS 的值被修改为一个冒号。我们也可以通过以下方式达到此效果。

---

```
OLD_IFS="$IFS"
IFS=:"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

---

首先我们储存了 IFS 的旧值，并将新值赋给 IFS，我们执行了 read 命令最后将 IFS 恢复原值。显然，对于做相同的事情，将变量赋值语句置于执行命令前，是更为简洁的方法。

操作符“<<<”象征一条嵌入字符串。嵌入字符串与嵌入文档类似，只不过更为简短，它包含的是一条字符串。本例将/etc/passwd 文件中读取的数据输送给 read 命令。读者或许会疑惑为什么使用这么复杂的方法而不是如下方法。

---

```
echo "$file_info" | IFS=: read user pw uid gid name home shell
```

---

原因如下。

#### read 不可重定向

通常 read 命令会从标准输入中获取输入，而不能采用以下方式。

```
echo "foo" | read
```

这种方法看起来可行，实则不然。看似命令能够执行成功，但是 REPLY 变量总是空值。为什么会出现这样的现象？

这主要是由于 shell 处理管道的方式。在 bash（和其他 shell，如 sh）中，管道会创造子 shell。子 shell 复制了 shell 及 pipeline 执行命令过程中使用到的 shell 环境。前例中，read 命令就是在子 shell 中执行的。

类 UNIX 系统的子 shell 会为执行进程复制所需的 shell 环境。进程结束后，所复制的 shell 环境即被销毁，这意味着子 shell 永远不会改变父类进程的 shell

环境。`read` 命令给变量赋值，这些变量会成为 shell 环境的一部分。而在上述范例中，`read` 将变量 `foo` 的值赋给子 shell 环境中的 `REPLY` 变量，但是当命令退出时，子 shell 与其环境就会被销毁，`read` 命令的赋值效果也就丢失了。

嵌入字符串是解决此类问题的方法之一，在第 36 章我们将介绍另外一种办法。

## 28.2 验证输入

在程序具备了读取键盘输入功能的同时，也带来了新的编程上的挑战——验证输入。通常来说，程序写得好与不好的差别仅仅在于程序是否能够处理突发意外情况。而意外情况经常以错误输入的形式出现。前一章节的评估程序涉及到了少量此类操作，程序检查整数变量的值并筛选出空值与非数值的字符。对所有程序接收的输入执行此类验证是防御无效数据的重要方式，且对于多用户共享的程序尤其重要。只有那些执行特殊任务且只会被作者使用一次的程序，出于经济学原理可以省略这些安全措施。尽管如此，若程序执行的是像删除文件这样的危险任务，为以防万一还是进行数据验证会比较好。

以下是一个对不同类型输入进行验证的程序。

---

```
#!/bin/bash

# read-validate: validate input

invalid_input () {
    echo "Invalid input '$REPLY'" >&2
    exit 1
}

read -p "Enter a single item > "

# input is empty (invalid)
[[ -z $REPLY ]] && invalid_input

# input is multiple items (invalid)
(( $(echo $REPLY | wc -w) > 1 )) && invalid_input

# is input a valid filename?
if [[ $REPLY =~ ^[[:alnum:]]\._]+$ ]]; then
    echo "'$REPLY' is a valid filename."
    if [[ -e $REPLY ]]; then
        echo "And file '$REPLY' exists."
    else
        echo "However, file '$REPLY' does not exist."
    fi
fi
```

```

# is input a floating point number?
if [[ $REPLY == ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then
    echo "'$REPLY' is a floating point number."
else
    echo "'$REPLY' is not a floating point number."
fi

# is input an integer?
if [[ $REPLY == ^-?[[:digit:]]+$ ]]; then
    echo "'$REPLY' is an integer."
else
    echo "'$REPLY' is not an integer."
fi
else
    echo "The string '$REPLY' is not a valid filename."
fi

```

---

此脚本首先提示用户进行输入，然后分析确定用户的输入。脚本使用了很多本书至今为止涵盖的概念，包括 shell 函数、[[ ]]、(( ))、控制操作符&&、if 以及适量的正则表达式。

## 28.3 菜单

菜单驱动是一种常见的交互方式。在菜单驱动的程序会呈现给用户一系列的选项，并请求用户进行选择。例如，可想象程序呈现的菜单如下所示。

---

```

Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

Enter selection [0-3] >

```

---

根据写 sys\_info\_page 程序得到的经验，我们可以构建菜单驱动的程序来执行上述菜单中各项任务。

---

```

#!/bin/bash

# read-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

```

```

        "
read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
    if [[ $REPLY == 0 ]]; then
        echo "Program terminated."
        exit
    fi
    if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        exit
    fi
    if [[ $REPLY == 2 ]]; then
        df -h
        exit
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
        exit
    fi
else
    echo "Invalid entry." >&2
    exit 1
fi

```

---

脚本在逻辑上分隔为两个部分。第一部分展示了菜单并获取用户的响应，第二部分对响应进行验证并执行相应的选项。需要注意脚本中 `exit` 命令的使用方法。在完成选定的功能后，`exit` 可防止脚本继续执行多余的代码。程序多出口通常并不是一个好现象（会导致程序逻辑难以理解），但是在上述脚本中适用。

## 28.4 本章结尾语

本章节迈出了程序交互的第一步，允许用户通过键盘向程序提供输入数据。使用迄今为止本书讲解过的技术，用户已经能够写出很多不同功效的程序，如专门的计算程序和易用的命令行工具前端。下一章我们将在菜单驱动程序概念的基础上对其进行改进。

## 28.5 附加项

因为接下来的程序会变得更加复杂，所以就需要用户认真学习并完全理解本章节程序的逻辑构建方式。作为练习，用户可使用 `test` 命令替代“`[[ ]]`”复合命令来改写本章节中的程序。提示：使用 `grep` 命令处理正则表达式，然后评估其退出状态。这将是个很好的练习。

# 第 29 章

## 流控制：WHILE 和 UNTIL 循环

前面的章节我们曾构建过一个菜单驱动的程序，用来提供不同的系统信息。此程序能有效地运作，但是却存在着重大的可用性问题——程序在执行一次选择之后即终止运行。更糟糕的是，若用户做出了无效的选择，那么程序会立刻以错误状态终止，用户没有再次尝试的机会。如果能够用某种方法重构程序，让程序能够一遍一遍地展示菜单选项，一直到用户选择退出程序，这样会好得多。

本章节将讲解叫做循环的编程概念，用于重复执行部分程序。`shell` 为循环提供了三种复合命令。本章节会介绍其中的两种，第 33 章中介绍第三种。

### 29.1 循环

日常生活充满了循环。每天上班、遛狗和切胡萝卜等都需要重复一系列的步骤。就比如用伪代码来描述切胡萝卜的过程，可能会是这样的。

1. 取案板。

2. 取刀。
3. 把胡萝卜放在案板上。
4. 拿起刀。
5. 前移胡萝卜。
6. 切胡萝卜。
7. 若整根胡萝卜都切好了，就退出程序，否则继续从第 4 步开始操作。

第 4 步到第 7 步构成了一个循环。循环中的动作会一直重复，直到条件“整根胡萝卜都切好了”为真。

## 29.2 while

bash 可以表达出类似的过程。若要按顺序显示 1~5 这 5 个数字，就可以构建这样的 bash 脚本。

---

```
#!/bin/bash

# while-count: display a series of numbers

count=1

while [ $count -le 5 ]; do
    echo $count
    count=$((count + 1))
done
echo "Finished."
```

---

脚本的执行结果如下所示。

---

```
[me@linuxbox ~]$ while-count
1
2
3
4
5
Finished.
```

---

while 命令的语法结构如下。

```
while commands; do commands; done
```

就如同 if 命令一样，while 会判断一系列指令的退出状态。只要退出状态为 0，它就执行循环内的命令。上述脚本中，我们创建了 count 变量并赋予 count 初始值 1。while 命令会判断 test 命令的退出状态。只要 test 命令返回的退出状态为 0，

那么循环内的指令继续执行。在每个循环周期的末尾，重复执行 test 命令。在经过 6 次循环迭代后，count 变量的值增加至 6，此时 test 命令返回的退出状态就不再是 0，循环终止。接下来程序会进一步执行循环后面的语句。

我们可以使用 while 循环改进第 28 章中的 read-menu 程序。

---

```
#!/bin/bash

# while-menu: a menu driven system information program

DELAY=3 # Number of seconds to display results

while [[ $REPLY != 0 ]]; do
    clear
    cat <<- _EOF_
        Please Select:

        1. Display System Information
        2. Display Disk Space
        3. Display Home Space Utilization
        0. Quit
_EOF_
    read -p "Enter selection [0-3] > "

    if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            sleep $DELAY
        fi
        if [[ $REPLY == 2 ]]; then
            df -h
            sleep $DELAY
        fi
        if [[ $REPLY == 3 ]]; then
            if [[ $(id -u) -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh $HOME
            fi
            sleep $DELAY
        fi
    else
        echo "Invalid entry."
        sleep $DELAY
    fi
done
echo "Program terminated."
```

---

将菜单封装到 while 循环内，程序就可以在用户每次选择后重复展示菜单项。只要 REPLY 值不为 0，重复循环，展示菜单项，给用户又一次进行选择的机会。而在每次动作结束时，系统执行 sleep 命令使程序暂停几秒，让用户能看

到选择执行的结果，随后程序清空屏幕显示并再次展示菜单项。一旦 REPLY 值为 0，也就意味着用户选择了退出项，循环终止，程序进一步执行 done 之后的代码。

## 29.3 跳出循环

bash 提供了两种可用于控制循环内部程序流的内建命令。其中 break 命令立即终止循环，程序从循环后的语句恢复执行。continue 命令则会导致程序跳过循环剩余的部分，直接开始下一次循环迭代。下面这个版本的 while-menu 程序实际应用了 break 和 continue。

---

```

#!/bin/bash

# while-menu2: a menu driven system information program

DELAY=3 # Number of seconds to display results

while true; do
    clear
    cat <<- _EOF_
        Please Select:

        1. Display System Information
        2. Display Disk Space
        3. Display Home Space Utilization
        0. Quit

_EOF_
    read -p "Enter selection [0-3] > "

    if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            sleep $DELAY
            continue
        fi
        if [[ $REPLY == 2 ]]; then
            df -h
            sleep $DELAY
            continue
        fi
        if [[ $REPLY == 3 ]]; then
            if [[ $(id -u) -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh $HOME
            fi
            sleep $DELAY
        fi
    fi
done

```

---

```

        continue
    fi
    if [[ $REPLY == 0 ]]; then
        break
    fi
else
    echo "Invalid entry."
    sleep $DELAY
fi
done
echo "Program terminated."

```

---

这个版本的程序脚本构建了一个无限循环（无限循环永远不会自动终止），利用 `true` 命令向 `while` 提供退出状态。因为 `true` 退出时的状态总为 0，所以循环永远都不会停止。这是一个很常见的脚本技术。因为循环永远不能自动终止，所以需要程序员提供在适当的时刻跳出循环的方式。当用户选择为 0 时，脚本使用 `break` 命令来终止循环。为了使脚本执行更加高效，可以在其他脚本选项的末端使用了 `continue`。在确认用户做出了选择后，`continue` 让脚本跳过了不需要执行的代码。例如，若确认用户选择了 1，那么没有必要验证其他选项了。

## 29.4 until

`while` 命令退出状态不为 0 时终止循环，而 `until` 命令则刚好相反。除此之外，`until` 命令与 `while` 命令很相似。`until` 循环会在接收到为 0 的退出状态时终止。在 `while-count` 脚本中，循环会一直重复到 `count` 变量小于等于 5。使用 `until` 改写脚本也可以达到相同的效果。

---

```

#!/bin/bash

# until-count: display a series of numbers

count=1

until [ $count -gt 5 ]; do
    echo $count
    count=$((count + 1))
done
echo "Finished."

```

---

将测试表达式改写为 `$count -gt 5`，`until` 就可以在合适的时刻终止循环。选择使用 `while` 还是 `until`，通常取决于哪种循环能够允许程序员写出最明了的测试表达式。

## 29.5 使用循环读取文件

`while` 和 `until` 可处理标准输入，这让使用 `while` 和 `until` 循环处理文件成为可能。本书前面的章节曾使用过 *distros.txt* 文件，接下来的示例显示了该文件的内容。

---

```
#!/bin/bash

# while-read: read lines from a file

while read distro version release; do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
        $distro \
        $version \
        $release
done < distros.txt
```

---

为将一份文件重定向到循环中，我们可在 `done` 语句之后添加重定向操作符。`done` 循环使用 `read` 命令读取重定向文件中的字段。在到达文件末端之前，退出状态为 0。在读取过文件中的每一行之后，`read` 命令退出，此时退出状态变为非 0，循环终止。将标准输入重定向到循环中也是可以做到的。

---

```
#!/bin/bash

# while-read2: read lines from a file

sort -k 1,1 -k 2n distros.txt | while read distro version release; do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
        $distro \
        $version \
        $release
done
```

---

此脚本获取 `sort` 命令的输出并显示文本流。但需要留意的是，因为管道是在子 shell 中进行循环操作。当循环终止时，循环内部新建的变量或者是对变量的赋值效果都会丢失。

## 29.6 本章结尾语

在介绍过循环，讲解过分支、子进程和队列之后，本书已经讲解了编程使用的几种主要的流控制方法。`bash` 还提供了许多其他的方法，但这些方法也只是对这些基本概念的改进。

# 第 30 章

## 故 障 诊 断

随着脚本的复杂度越来越高，当脚本出现错误或执行情况和预期不同的时候，就需要看看是哪里出的问题。本章将讲解一些脚本中常见的错误类型以及几种用于追踪和解除错误的有用技巧。

### 30.1 语法错误

语法错误是一种常见的错误类型，其中就包括了 shell 语句中一些元素的拼写错误。在大多数情况下，shell 会拒绝执行含有此种类型错误的脚本。

在接下来的讨论过程中，我们将使用以下脚本来演示常见的错误类型。

---

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
```

```

        echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi

```

可以看到，此脚本运行正常。

```
[me@linuxbox ~]$ trouble
Number is equal to 1.
```

### 30.1.1 引号缺失

现在修改上述脚本，删除第一个 echo 命令后实参后的双引号。

```

#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
    echo "Number is equal to 1.
else
    echo "Number is not equal to 1."
fi

```

观察出现的现象如下所示。

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 10: unexpected EOF while looking for matching `"'
/home/me/bin/trouble: line 13: syntax error: unexpected end of file
```

此删除操作使脚本产生了两个错误。有趣的是，错误报告指出的行并不是之前所删除双引号所在的行，而是之后的代码。可以看到，当系统读取到所删除双引号的位置之后，bash 将继续向下寻找与前双引号对应的引号，这样的行为会一直延续到 bash 找到目标，也就是在第二个 echo 命令后的第一个引号处。然后 bash 就陷入了混乱中，即 if 命令的语法结构被破坏了，fi 语句现在处于了引号标识（但是又只有一边存在引号）的字符串中。

这种类型的错误在长脚本中很难发现，而使用带有语法结构突出显示功能的编辑器能够帮助找到这类错误。如果系统配备的是完整版的 vim，可使用以下命令启用 vim 的语法结构突出显示功能。

```
:syntax on
```

### 30.1.2 符号缺失冗余

另一种常见的错误是如 if 或 while 这样的复合命令结构不完整。现在就删

除 if 命令中 test 部分后的分号，看看会产生什么情况。

---

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ] then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

---

结果如下所示。

---

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 9: syntax error near unexpected token `else'
/home/me/bin/trouble: line 9: `else'
```

---

再一次，报错信息指向的代码远在实际问题之后，这个现象的机理非常有趣。事实是，if 接收一系列的命令，并评估系列中最后一个命令为退出码。在本程序中，这个命令系列只由一条命令组成，“[，”即 test 的同义表示。“[”命令将其之后的部分视为一系列的参数——本例中也就是“\$number”、“=”、“1”和“]”。在分号被删除之后，单词 then 也就被添加到了参数系列中，这在语法中也是允许的。接下来的 echo 命令也是合法的，echo 被翻译为 if 接收的命令列表中的另一条命令，并成为 if 命令的退出码。最后遇到的就是 else，但是因为 else 是 shell 的保留单词（在 shell 中有特定的含义），并不是命令的名称，所以 else 不应该出现在这里。因此才有了如上的报错信息。

### 30.1.3 非预期的展开

有一些脚本错误是间歇出现的。脚本有时候会运行正常，有时候又会因为展开的结果而出错。现在将缺失的分号补回并改变 number 的值为空，即可以演示这类错误。

---

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

---

运行修改过的脚本得到的输出结果如下所示。

---

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 7: [: =: unary operator expected
Number is not equal to 1.
```

---

也就是一条看起来很深奥的报错信息，外加上脚本第二条 echo 命令的输出结果。造成问题的原因是 test 命令中 number 变量的展开。当命令

---

```
[ $number = 1 ]
```

---

expansion 情形为 number 变量为空，就造成了这样的情况。

---

```
[ = 1 ]
```

---

等式无效，也就产生了错误。“=”是一个二元操作符（要求符号两边都有值），但是本例中缺少了一个值，所以 test 命令要求程序改用一元操作符（如-z）。接下来，因为 test 不成立（因为上述错误），if 命令接收到一个非零的退出码，从而执行了第二个 echo 命令。

用户可以通过在 test 命令中使用双引号引用第一个参数方式更正这个错误。

---

```
[ "$number" = 1 ]
```

---

现在展开命令，结果是这样的。

---

```
[ "" = 1 ]
```

---

“=”的前后有了正确数量的参数。除了空字符串需要引用之外，类似包含空格的文件名这样的多字符的字符串也要前后加以引号。

## 30.2 逻辑错误

与语法错误不同的是，逻辑错误不会阻碍脚本的运行。因为脚本包含的逻辑问题，脚本尽管可以运行但是不能产生理想的结果。可能发生的逻辑错误有非常多的种，但是以下几种逻辑错误是脚本中最常见的。

- **条件表达错误。**编写代码中，很容易写出不正确的 if、then 和 else 语句，造成错误的逻辑，例如相反的或者不完整的逻辑表达。
- **“从 1 开始”错误。**在使用计数器的循环中，可能会忽略程序需要从 0 而不是 1 开始计数才能在正确的点结束循环。这种错误会导致循环次数过多，超过了预期的终点，或循环次数过少，缺失了最后一次的迭代过程。

- 非预期的情形。大多数此种错误来源于程序运行过程遇到了编写程序人员没有预期到的数据或环境。非预期展开也包括在内，如一个包括了空格的文件名本应该作为单个文件名存在，却扩展成为了多个命令的实参。

### 30.2.1 防御编程

在编程中核实各项假设是很重要的事情，这意味我们着需要对程序的退出状态以及脚本所用命令进行仔细评估。有一个真实的例子，一位倒霉的系统管理员写了一个脚本，用于对一台重要的服务器执行维护任务，这个脚本包含了这样两行代码。

---

```
cd $dir_name
rm *
```

---

只要名为 `dir_name` 的目录存在，以上两行代码并没有什么本质性的错误。但是，如果目标不存在又会发生什么呢？此情形下，`cd` 命令跳转失败，脚本继续读取下一行代码，删除的就是当前的工作目录。完全不是管理员想要的结果！出于这样的设计，这位管理员不幸地删除了服务器一个重要的部分。

现在就看看改进此设计的一些方法。第一种可能的方法是使执行 `rm` 命令以 `cd` 的成功执行为前提。

---

```
cd $dir_name && rm *
```

---

这样以来，如果 `cd` 命令跳转失败，`rm` 命令就不会执行。情况有所改善，但是仍然存在当 `dir_name` 为空的情况下，用户的主目录被删除的可能性。检查 `dir_name` 是否确实包含有效的目录名可以避免此情形。

---

```
[[ -d $dir_name ]]; && cd $dir_name && rm *
```

---

通常来说，若发生了上述情形之一，最好立刻终止脚本的运行。

---

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
```

---

以上代码中，不仅检查 `dir_name` 是否包含有效的目录名，且验证了 `cd` 命令

是否跳转成功。任一验证没有通过的话，将对应的描述性报错信息发送给系统标准错误，且脚本终止运行，其退出状态为 1，标志着运行失败。

### 30.2.2 输入值验证

一条普遍适合的编程法则是若程序需获得用户输入，则程序必须能够处理任何输入值。这通常意味着必须仔细检查输入值，以保证有效的输入可用于进一步的处理过程。在 29 章讲解 `read` 命令时，我们曾经遇到过类似的范例。某脚本包含了以下测试来验证菜单的选择结果是否有效。

---

```
[[ $REPLY =~ ^[0-3]$ ]]
```

---

这项测试非常明确，只有用户返回的字符串是 0~3 之间的数字时，程序才会返回值为 0 的退出状态，除此之外不会接受任何其他值。有时输入值验证类型的测试写起来很有挑战性，但却是创造高质量脚本的必经之路。

#### 时间雕琢而成的设计

当我还是一名工业设计专业的大学生时，曾听一位教授说过：“工程项目设计的质量等级取决于给予设计师的时间长度。若要用五分钟来设计一个杀死苍蝇的设备，结果可能是苍蝇拍。而给予设计师五个月的话，结果就可能是激光制导的‘杀苍蝇系统’”。

这项准则对编程同样适用。一方面，如果程序员只使用一次脚本，那么一个简陋劣质的脚本就能满足要求。这种类型的脚本很常见，且应开发得尽量快，使得效益最大化。而另一方面，如果脚本是用于产品，也就是说脚本会用于重要的任务，反复使用或被多个用户使用，那么脚本的开发过程就应当更加仔细小心。

## 30.3 测试

对任何软件的开发过程来说，包括脚本的开发过程，测试都是一个非常重要的步骤。在开源世界中有这么一种说法——“尽早发布，经常发布”。这种说法映射出了测试的重要性。通过尽早发布和经常发布新版本，软件就能够得到更多用户的使用和测试。经验表明如果能尽早在开发周期中发现 bug，那么剩余的 bug 会更加容易发现，修补 bug 的代价也会更小。

### 30.3.1 桩

前面讲到过使用桩(stub)核查程序流程的内容。从脚本开发的最初阶段开始，桩就是可用于查看工作进程的重要技术手段。

现在让我们回顾一下之前的文件删除问题，看看怎样提高代码的易测试性。因为初始代码的目的是删除文件，所以对代码直接进行测试具有一定的危险性，对代码进行一些修改可以解决这个问题。

---

```

if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        echo rm * # TESTING
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
exit # TESTING

```

---

因为出现错误的条件设置本身已经表达出了足够的信息，所以我们就不需要再添加什么了。最重要的变化是在 rm 命令之前放置了 echo 命令，使得 rm 命令和扩展的命令参数可以展示出来，而不是进行实际的删除操作，从而使代码能够安全地执行。在代码片段的末尾，我们添加了 exit 命令来结束测试，防止执行脚本的其他部分。是否需要添加 exit 命令取决于不同脚本的不同设计。

在测试的过程中，我们还需要添加一些扮演“记录者”角色的注释，来记录做出的改变。在测试结束后，可根据注释还原代码。

### 30.3.2 测试用例

开发和应用高质量的测试用例是执行测试过程中的重要部分。这要求测试人员要很小心地选择能反映边缘测试的输入数据和操作条件。在上述代码片段（算是很简单的片段）中，我们需要测试以下三种特定条件下代码的执行情况。

- `dir_name` 包含的是存在的目录名。
- `dir_name` 包含的是不存在的目录名。
- `dir_name` 为空。

在这三种条件下分别执行测试，就能够得到较高的测试覆盖率。

就像设计一样，测试也是由时间雕琢而成。并不是脚本的每个角落都需要测试，要紧的是确定什么是最重要的。因为出错的代码可能会造成毁灭性的后果，所以不管是代码设计还是测试都需要程序员仔细地斟酌。

## 30.4 调试

如果测试揭露出脚本存在问题，那么下一步就是调试。“问题”通常意味着，在某些情况下，脚本的运行结果和预期的效果不同。如果是这样的话，就需要仔细查明脚本实际上是怎样运作的以及为什么会出现这样的情况。寻找 bug 有时会需要很多侦查工作。

设计优良的脚本本身能够提供一些帮助，防御性的脚本遇到异常时会向用户反馈有用的信息。但是，解决预料之外的奇怪问题就需要介入其他的测试技术。

### 30.4.1 找到问题域

在一些脚本中，尤其是长脚本中，将问题相关的脚本域隔离出来是很有必要的。隔离的部分并不一定是实际问题所在之处，但是通常能提供通往实际原因的线索。其中一种能够用于隔离代码片段的方法是“注释”掉脚本的一部分。

---

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
# else
#     echo "no such directory: '$dir_name'" >&2
#     exit 1
fi
```

---

### 30.4.2 追踪

bug 还经常表现为脚本中异常的逻辑流程。也就是说，脚本的一部分或者从来没有执行过，或者以错误的顺序或在错误的时间执行了。追踪是一种用于查看程序实际运行流程的技术。

一种追踪技术是通过在脚本中添加通知信息的方式来展示程序执行之处。

我们可以在上述代码片段中添加一些信息：

---

```

echo "preparing to delete files" >&2
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
echo "deleting files" >&2
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
echo "file deletion complete" >&2

```

---

我们将这些信息发送给标准错误，从而与一般的程序输出区分开。这些信息所在的行没有缩进，使这些代码在需要删除的时候易于寻找。

在脚本执行之后，系统就已经执行了文件的删除操作。

---

```

[me@linuxbox ~]$ deletion-script
preparing to delete files
deleting files
file deletion complete
[me@linuxbox ~]$

```

---

bash 也提供了一种追踪的方法，即直接使用-x 选项或 set 命令加-x 选项。在之前的 trouble 脚本中，在脚本第一行添加-x 选项即可激活对整个脚本的追踪活动。

---

```

#!/bin/bash -x

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi

```

---

脚本执行的结果如下所示。

---

```

[me@linuxbox ~]$ trouble
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.

```

---

激活追踪之后，我们就可以看到变量展开的执行情况。行开端的加号表示此

行是系统的追踪信息，以区别于一般的输出。加号是追踪信息的默认特征，是由 shell 变量 PS4（提示符字符串 4）设定的，用户可以修改变量值使追踪活动的提示符提供更多帮助信息。接下来，对 PS4 做出修改，使提示符包含执行追踪活动的脚本行号。值得注意的是，单引号使得提示符真正被使用时才展开生效。

---

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

---

要对脚本选定的一部分而不是整个脚本执行追踪，可以使用 set 命令加-x 选项。

---

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

set -x # Turn on tracing
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

---

在这里使用 set 命令加-x 激活追踪，set 命令加+x 解除追踪。这项技术可以用来检验一个问题脚本的多个部分。

### 30.4.3 运行过程中变量的检验

在执行脚本并追踪脚本的过程中，显示各变量的值以体现脚本的内部活动会很有帮助。这项功能通常通过添加额外的 echo 语句完成。

---

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

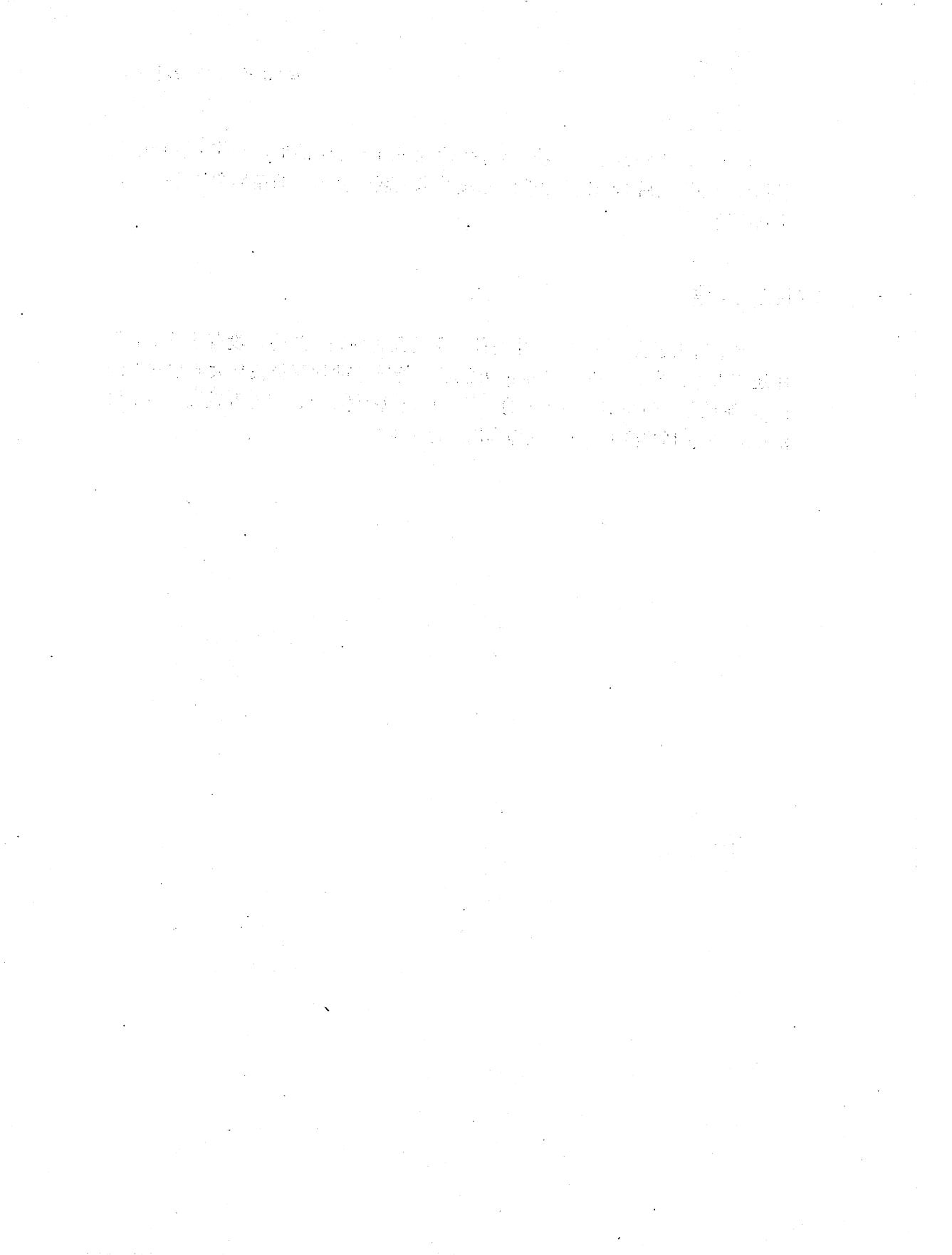
echo "number=$number" # DEBUG
set -x # Turn on tracing
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

---

在这个简单的例子中，我们只输出了变量 `number` 的值，并使用注释标注了新行，便于之后的识别和删除。这项技术在观察脚本中的循环和计算行为时尤其有用。

## 30.5 本章结尾语

本章讨论了脚本开发过程中可能出现的几种问题。当然，没有涉及到的问题还有很多。本章讲述的 `debug` 方法已经能够帮助程序员找到大多数常见的 `bug`。调试是一门在实践中成长的艺术，它既包括避免 `bug`（在开发过程中不停测试），也包括找到 `bug`（有效地利用追踪技术）。



# 第31章

## 流控制：case 分支

本章节将继续讲解流控制的内容。第 28 章构建了一些简单的菜单，并建立了用于响应用户选择的逻辑。为达到这个目的，我们使用了一系列的 if 命令来确定哪个是选中项。这种类型的程序构造经常被用到，所以许多编程语言（包括 shell）为基于多项选择的判断提供了流控制机制。

### 31.1 case

bash 的多项选择复合命令被称为 case，其语法如下所示。

```
case word in
    [pattern [| pattern]...) commands ;;]...
esac
```

回顾第 28 章中的 read-menu 程序，可以看到用于响应用户选择的逻辑如下所示。

---

```
#!/bin/bash
# read-menu: a menu driven system information program
```

```

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"

read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
    if [[ $REPLY == 0 ]]; then
        echo "Program terminated."
        exit
    fi
    if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        exit
    fi
    if [[ $REPLY == 2 ]]; then
        df -h
        exit
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
        exit
    fi
else
    echo "Invalid entry." >&2
    exit 1
fi

```

---

使用 case 可以简化以上逻辑。

---

```

#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "

Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"

read -p "Enter selection [0-3] > "

```

```

case $REPLY in
    0)      echo "Program terminated."
            exit
            ;;
    1)      echo "Hostname: $HOSTNAME"
            uptime
            ;;
    2)      df -h
            ;;
    3)      if [[ $(id -u) -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh $HOME
            fi
            ;;
        *)      echo "Invalid entry" >&2
            exit 1
            ;;
esac

```

---

case 命令将关键字的值——本例中，即变量 REPLY 的值——与特定的模式相比较，若发现吻合的模式，就执行与此模式相联系的命令。发现吻合的模式后，将不再比对剩余的模式。

### 31.1.1 模式

同路径名展开一样，case 使用以“)”字符结尾的模式。表 31-1 列出一些有效的模式。

表 31-1 case 模式范例

模式	描述
a)	若关键字为 a 则吻合
[[:alpha:]]	若关键字为单个字母则吻合
???	若关键字为三个字符则吻合
*.txt)	若关键字以.txt 结尾则吻合
*)	与任何关键字吻合。在 case 命令的最后的一个模式应用此项是个不错的做法，可对应所有前模式不吻合的关键字，也就是对应任何可能的无效值

下面是一个使用模式的范例。

---

```

#!/bin/bash

read -p "enter word > "

case $REPLY in
    [[:alpha:]])
        echo "is a single alphabetic character." ;;

```

```
[ABC][0-9])      echo "is A, B, or C followed by a digit." ;;
???)            echo "is three characters long." ;;
*.txt)          echo "is a word ending in '.txt'" ;;
*)              echo "is something else." ;;
esac
```

---

### 31.1.2 多个模式的组合

我们也可以使用竖线作为分隔符来组合多个模式，模式之间是“或”的条件关系。这对一些类似同时处理大写和小写字母的事件很有帮助。如下所示。

```
#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "
Please Select:

A. Display System Information
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "

case $REPLY in

    q|Q)    echo "Program terminated."
            exit
            ;;
    a|A)    echo "Hostname: $HOSTNAME"
            uptime
            ;;
    b|B)    df -h
            ;;
    c|C)    if [[ $(id -u) -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh $HOME
            fi
            ;;
    *)      echo "Invalid entry" >&2
            exit 1
            ;;

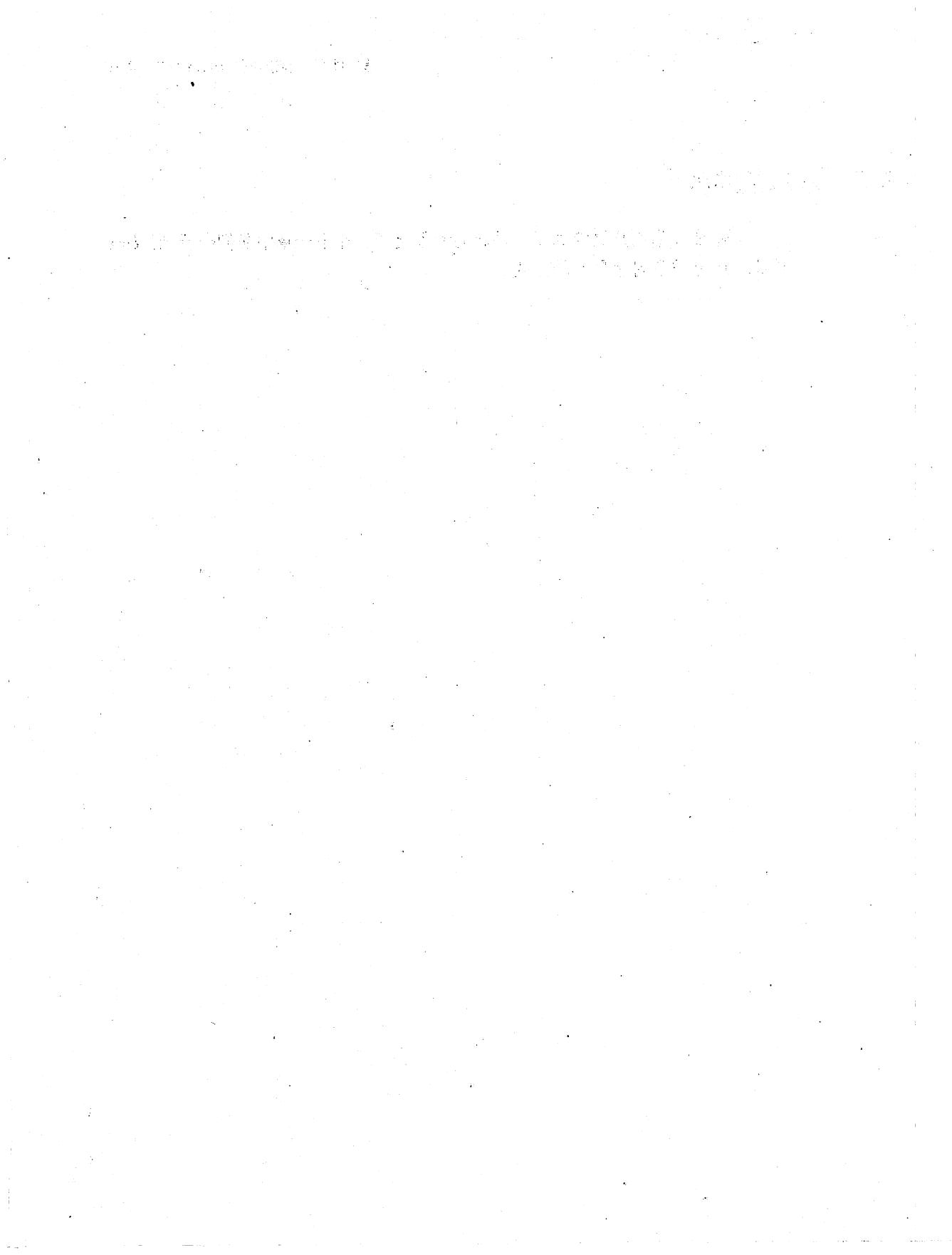
esac
```

---

以上代码将 case-menu 程序修改为通过字母而不是数字进行菜单选择。可以注意到在新模式下，用户进行选择时可以输入大写字母，也可以输入小写字母。

## 31.2 本章结尾语

case 命令为用户又增添了一种新的编程技巧。在下一章中我们会看到，case 是处理特定类型问题最好的方式。



# 第32章

## 位置参数

之前我们一直没有涉及程序接收和处理命令行选项及实参的能力，本章节将讲解允许程序访问命令行内容的 shell 功能。

### 32.1 访问命令行

shell 提供了一组名为位置参数的变量，用于储存命令行中的关键字，这些变量分别命名为 0~9。可以通过以下方法展示这些变量。

---

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
```

```
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
```

---

这个简单的脚本展示了从变量\$0 到变量\$9 的值。在没有任何命令行实参的情形下执行此脚本结果如下所示。

---

```
[me@linuxbox ~]$ posit-param

$0 = /home/me/bin/posit-param
$1 =
$2 =
$3 =
$4 =
$5 =
$6 =
$7 =
$8 =
$9 =
```

---

即便没有提供任何实参，变量\$0 总是会储存有命令行显示的第一项数据，也就是所执行程序所在的路径名。现在让我们，看一下提供实参情形下的程序执行结果：

---

```
[me@linuxbox ~]$ posit-param a b c d

$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

---

### 注意

使用参数扩展技术，用户实际可以获取多于 9 个的参数。为标明一个大于 9 的数字，将数字用大括号括起来，例如\${10}、\${55}和\${211}等。

## 32.1.1 确定实参的数目

shell 还提供了变量\$#以给出命令行参数的数目。

---

```
#!/bin/bash

# posit-param: script to view command line parameters
echo "
Number of arguments: $#
```

```
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
```

---

以上程序运行结果如下所示。

---

```
[me@linuxbox ~]$ posit-param a b c d

Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

---

### 32.1.2 shift——处理大量的实参

但是如果给程序提供大量的实参会是什么呢？如下所示。

---

```
[me@linuxbox ~]$ posit-param *

Number of arguments: 82
$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Desktop
$9 = dirlist-bin.txt
```

---

在本示例系统中，通配符“\*”扩展为82个实参。怎样才能处理这么多的实参呢？shell提供了一种略显笨拙的处理方法。每次执行shift命令后，所有参数的值均“下移一位”。实际上，通过shift命令我们就可以只处理一个参数（\$0之外的一个参数，\$0值恒定）而完成全部程序任务。

---

```

#!/bin/bash

# posit-param2: script to display all arguments

count=1

while [[ $# -gt 0 ]]; do
    echo "Argument $count = $1"
    count=$((count + 1))
    shift
done

```

---

每当执行一次 `shift` 命令时，变量`$2` 的值就赋给变量`$1`，而`$3` 的值则赋给变量`$2`，依次类推。变量`$#`的值同时减 1。

在程序 `posit-param2` 中，我们创建了一个循环，只要还存在 1 个实参，循环就不停止。首先输出当前的实参，其次，每当循环迭代一次，就将变量 `count` 值加 1，代表处理过的实参数目。最后，执行 `shift` 命令使`$1` 载入下一个实参的值。以下是 `posit-param2` 的运行范例。

---

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c
Argument 4 = d
```

---

### 32.1.3 简单的应用程序

不使用 `shift`，我们也可以写出使用位置参数的有用的应用程序。下面是一个简单的文件信息程序范例。

---

```

#!/bin/bash

# file_info: simple file information program

PROGNAME=$(basename $0)

if [[ -e $1 ]]; then
    echo -e "\nFile Type:"
    file $1
    echo -e "\nFile Status:"
    stat $1
else
    echo "$PROGNAME: usage: $PROGNAME file" >&2
    exit 1
fi

```

---

这个程序输出了单个特定文件的文件类型（来自 `file` 命令）和文件状态（来自 `stat` 命令）。程序中的 `PROGNAME` 变量是一个有趣的角色，其值来自 `basename`

\$0 命令的执行结果。basename 命令的作用是移除路径名的起始部分，只留下基本的文件名。在上述例子中，basename 移除了\$0 参数中范例程序所在路径全名的起始部分，得到的值在构建程序信息——如程序末尾的使用信息——时很有用处。若使用 basename 构建信息，在重命名脚本后，这条信息会自动调整所包含的程序名称。

### 32.1.4 在 shell 函数中使用位置参数

就像位置参数可用于向 shell 脚本传递实参一样，位置参数也可用于 shell 函数实参的传递。现在将 file\_info 脚本改写为 shell 函数，以进行如下演示说明。

---

```
file_info () {
    # file_info: function to display file information

    if [[ -e $1 ]]; then
        echo -e "\nFile Type:"
        file $1
        echo -e "\nFile Status:"
        stat $1
    else
        echo "$FUNCNAME: usage: $FUNCNAME file" >&2
        return 1
    fi
}
```

---

现在，如果一个包含了 file\_info 函数的脚本以一个文件名为实参调用 file\_info，则此实参就被传递给 file\_info 函数。

在这样的条件下，我们就可以写出很多不仅普通脚本可使用，而且.bashrc 文件也适用的有用的 shell 函数。

需要注意的是，PROGRAME 变量被换成了名为 FUNCNAME 的 shell 变量。shell 自动更新 FUNCNAME 以追踪当前执行的 shell 函数。但是变量\$0 包含的总是命令行第一项的路径全名（例如，程序名称），而并不像用户可能期待的那样包含了 shell 函数的名称。

## 32.2 处理多个位置参数

有时我们将所有的位置参数作为一个整体来处理会比较方便。例如，为目标程序写一个包装，也就是编写一个简化了目标程序运作过程的脚本或者 shell 函数的时候，包装就供应了一系列的复杂的命令行选项，并为下一层的程

序传递所需的实参。

shell 为这项功能提供了两种特殊的参数。两种参数都能够扩展为一个完整的位置参数列，但是又有着微妙的区别。表 32-1 描述了这两种参数。

表 32-1 特殊的参数\*和@

参数	描述
\$*	可扩展为从 1 开始的位置参数列。当包括在双引号内时，扩展为双引号引用的由全部位置参数构成的字符串，每个位置参数以 IFS shell 变量的第一个字符（默认情况下为空格）间隔开
\$@	可扩展为从 1 开始的位置参数列。当包括在双引号内时，将每个位置参数扩展为双引号引用的单独单词

下面的脚本演示了这两种特殊的参数功能。

---

```
#!/bin/bash

# posit-params3 : script to demonstrate $* and $@

print_params () {
    echo "\$1 = $1"
    echo "\$2 = $2"
    echo "\$3 = $3"
    echo "\$4 = $4"
}
pass_params () {
    echo -e "\n" '$*' ; print_params $*
    echo -e "\n" '"$*"' ; print_params "$*"
    echo -e "\n" '$@' ; print_params $@
    echo -e "\n" '"$@"' ; print_params "$@"
}
pass_params "word" "words with spaces"
```

---

在这个较为复杂的程序中，我们创建了两个实参——word 和 words with spaces，并将其传递给 pass\_params 函数。而 pass\_params 函数使用 4 种 “\$\*” 和 “\$@” 的方法，将这两个实参分别依次传递给 print\_params 函数。脚本执行的结果反映了它们之间的区别。

---

```
[me@linuxbox ~]$ posit-params3

$* :
$1 = word
$2 = words
$3 = with
$4 = spaces

"$*":
$1 = word words with spaces
$2 =
$3 =
```

```
$4 =
$@ :
$1 = word
$2 = words
$3 = with
$4 = spaces

"$@" :
$1 = word
$2 = words with spaces
$3 =
$4 =
```

---

对 word 和 words with spaces 两个实参，“\$\*”和“\$@”都产生了包含 4 个单词的结果，即 word、words、with 和 spaces。“\$\*”产生的是只包含一条字符串的结果，即 word words with spaces。“\$@”产生的则是包含了两条字符串的结果，即 word 和 words with spaces。

结果与实际期望相符。从中我们得到的结论就是尽管 shell 提供了四种不同的获得位置参数列的方法，因为“\$@”保持了每个位置参数的完整性，所以迄今为止，“\$@”是大多数情况下最令人满意的方法。

### 32.3 更完整的应用程序

在很长的间隔之后，现在我们重新回到 sys\_info\_page 程序的工作上来。接下来我们将为程序添加如下所示的命令行选项。

- **输出文件。** 指定程序输出文件的选项。使用形式是-f file 或者--file file。
- **交互模式。** 这个选项会提示用户输出文件的名称，并检验此文件是否已经存在。若文件已经存在，在覆盖文件之前提示用户。使用形式是-i 或者--interactive。
- **帮助。** 形式为-h 或者--help，可使程序输出帮助性质的使用说明。

下面是完善命令行处理功能所需的代码。

---

```
usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

# process command line options

interactive=
filename=
```

```

while [[ -n $1 ]]; do
    case $1 in
        -f | --file)
            shift
            filename=$1
            ;;
        -i | --interactive)
            interactive=1
            ;;
        -h | --help)
            usage
            exit
            ;;
        *)
            usage >&2
            exit 1
            ;;
    esac
    shift
done

```

---

首先，我们添加名为 `usage` 的 shell 函数，在使用 `help` 选项或者未知选项时，`usage` 会输出相应的提示信息。

接下来，我们将处理过程循环启动。只要位置参数\$1 不为空，循环不停。在循环的末尾，使用 `shift` 命令推进位置参数处理进程，确保不是死循环。

在循环中，我们使用 `case` 语句来检验当前位置参数是否与任何程序支持的选项相匹配。若发现支持的参数选项，则基于参数做出相应的行为。若没有发现匹配项，则输出程序使用信息，脚本以 `error` 结束运行。

程序中-f 参数的处理过程很有趣。在检测到-f 后，即执行附加的 `shift` 命令，使位置参数\$1 的内容置换为-f 选项后的文件名实参。

下一步我们将添加完善交互模式所需的代码。

---

```

# interactive mode

if [[ -n $interactive ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e $filename ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case $REPLY in
                Y|y)      break
                ;;
                Q|q)      echo "Program terminated."
                exit
                ;;
                *)       continue
                ;;
            esac
        elif [[ -z $filename ]]; then
            continue
        else
            break
        fi
    done
fi

```

```

        fi
done
fi
```

---

若 `interactive` 变量不为空，则开启一个无限循环，循环中包含了文件名提示符和接下来的当前存在的文件处理代码。若设想的输出文件已经存在，则提示用户覆盖文件。另选文件或者退出程序。若用户选择覆盖当前存在的文件，则使用 `break` 跳出循环。需要注意的是，只有在用户选择覆盖当前存在的文件或退出程序的时候，`case` 语句才适用。任何其他的选择都会使循环继续运行并再次提示用户。

为了完善程序指定输出文件名的功能，我们必须首先将现存的写页面的代码改写为 shell 函数，这样做的原因马上就会揭晓。

```

write_html_page () {
    cat <<- _EOF_
    <HTML>
        <HEAD>
            <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
            <H1>$TITLE</H1>
            <P>$TIME_STAMP</P>
            $(report_uptime)
            $(report_disk_space)
            $(report_home_space)
        </BODY>
    </HTML>
    _EOF_
    return
}

# output html page

if [[ -n $filename ]]; then
    if touch $filename && [[ -f $filename ]]; then
        write_html_page > $filename
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
fi
```

---

处理-f 选项的逻辑出现在上述代码的底部。在逻辑中，首先检验文件是否存在，若存在，则检验文件是否确实可写。为达到这样的目的，执行 `touch` 命令并检验结果文件是否是常规文件。这两项检验考虑到了在输入的是无效路径（`touch` 会执行失败）和文件已存在的情况下，文件是否是常规文件。

可以看到，在实际生成页面的执行过程中调用了 `write_html_page` 函数，并将函数的输出定向到标准输出（`filename` 变量为空时）或重定向到指定的文件。

## 32.4 本章结尾语

在位置参数的帮助下，用户可写出功能性更强的脚本。简单来说，对于重复性的任务，位置参数使得用户可以写出很有帮助的 shell 函数，并放置在用户的 `.bashrc` 文件中。

`sys_info_page` 程序在复杂性和先进性方面有了很大提升。下面是完整的程序，并突出显示了最近做出的改变。

---

```
#!/bin/bash

# sys_info_page: program to output a system information page

PROGNAME=$(basename $0)
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +%s %r %Z)
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    cat <<- _EOF_
        <H2>System Uptime</H2>
        <PRE>$(uptime)</PRE>
    _EOF_
    return
}
report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>
        <PRE>$(df -h)</PRE>
    _EOF_
    return
}
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}
```

```

usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}
write_html_page () {
    cat <<- _EOF_
    <HTML>
        <HEAD>
            <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
            <H1>$TITLE</H1>
            <P>$TIME_STAMP</P>
            $(report_uptime)
            $(report_disk_space)
            $(report_home_space)
        </BODY>
    </HTML>
_EOF_
    return
}

# process command line options

interactive=
filename=

while [[ -n $1 ]]; do
    case $1 in
        -f | --file)
            shift
            filename=$1
            ;;
        -i | --interactive)
            interactive=1
            ;;
        -h | --help)
            usage
            exit
            ;;
        *)
            usage >&2
            exit 1
            ;;
    esac
    shift
done

# interactive mode

if [[ -n $interactive ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e $filename ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case $REPLY in
                Y|y)
                    break
                    ;;
                Q|q)
                    echo "Program terminated."
                    exit
                    ;;
                *)
                    continue
                    ;;
            esac
        fi
    done
fi

```

```
        esac
    fi
done
fi

# output html page

if [[ -n $filename ]]; then
    if touch $filename && [[ -f $filename ]]; then
        write_html_page > $filename
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
fi
```

---

现在这个脚本已经很不错了，但是还没有结束。在下一章中，我们将会对脚本做出最后的改进。

# 第 33 章

## 流控制：for 循环

本章是关于流控制的最后一章，我们将会再学习一个新的 shell 循环结构。因为 for 循环采用在循环期间进行序列处理的机制，所以它不同于 while 循环和 until 循环。事实证明，这在编程时是非常有用的。因此，for 循环在 bash 脚本编程中是一种十分流行的结构。

自然地，实现一个 for 循环应使用 for 命令。在新版 bash 语言中，for 命令存在两种形式。

### 33.1 for：传统 shell 形式

原始的 for 命令语法如下。

```
for variable [in words]; do  
    commands  
done
```

其中，variable 是一个在循环执行时会增值的变量名，words 是一列将按顺

序赋给变量 variable 的可选项， commands 部分是每次循环时都会执行的命令。

for 命令在命令行上是很有用的。很容易就可以获知它的工作方式。

---

```
[me@linuxbox ~]$ for i in A B C D; do echo $i; done
A
B
C
D
```

---

在本例中，for 顺序执行了一个包含 4 个字符的列表：A、B、C 和 D。它采用包含 4 个字符的列表，循环执行了 4 次。在循环体内部 echo 命令显示变量 i 的值，以此表明赋值过程。像 while 循环和 until 循环一样，for 循环以关键词 done 结束。

for 循环真正强大的功能在于创建字符列表的方式有多种。例如，可以使用花括号扩展方式，如下所示。

---

```
[me@linuxbox ~]$ for i in {A..D}; do echo $i; done
A
B
C
D
```

---

或使用路径名扩展方式，如下所示。

---

```
[me@linuxbox ~]$ for i in distros*.txt; do echo $i; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros.txt
distros-vernums.txt
distros-versions.txt
```

---

再或者使用命令形式，如下所示。

---

```
#!/bin/bash

# longest-word : find longest string in a file

while [[ -n $1 ]]; do
    if [[ -r $1 ]]; then
        max_word=
        max_len=0
        for i in $(strings $1); do
            len=$(echo $i | wc -c)
            if (( len > max_len )); then
                max_len=$len
                max_word=$i
            fi
        done
        echo "$1: '$max_word' ($max_len characters)"
```

```

    fi
    shift
done

```

---

本例的功能是在一个文件中搜索最长的字符串。当在命令行输入一个或多个文件名时，本程序调用 strings 程序（包含在 GNU binutils 软件包里）并在每个文件中产生一个可读文本“字符”。for 循环按顺序处理每个字符，判断目前的字符是不是迄今为止找到的最长字符。当循环终止，将会打印出最长的字符。最后的字符将被打印出来。

如果 for 命令中字符部分的选项被忽略，for 循环默认处理该位置参数。我们可以使用如下方法，修改 longest-word 脚本。

```

#!/bin/bash

# longest-word2 : find longest string in a file

for i; do
    if [[ -r $i ]]; then
        max_word=
        max_len=0
        for j in $(strings $i); do
            len=$(echo $j | wc -c)
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
done

```

---

如上所示，最外层循环发生了改变，使用 for 代替了 while。因为 for 命令中语句列表采用默认值，所以使用位置参数。在循环内部，前例的变量 i 已经被变换为变量 j，并且 shift 也已弃之不用。

### 为什么变量名是 I?

你一定已经注意到以上每一个 for 循环中都选择使用变量 i。这是为什么呢？其实，除了惯例这个理由之外，没有其他原因。for 循环使用的变量可以是任何有效变量，不过 i 是最常见的，除此之外还有 j 和 k。

这种惯例来自于 Fortran 编程语言。在 Fortran 语言中，以字母 I、J、K、L 和 M 开头的未声明的变量自动被归类为整数，而以其他字母开头的变量被归类为实数（含十进制小数的数字）。因为当需要一个临时变量时（循环通常如此），这样做会比较省力，所以这种方式促使程序员使用变量 I、J 和 K 作为循环的变量。

为此也诞生了这样的 Fortran 双关语：“GOD 是真的（实数），除非被声

明为整数。”（译者注：GOD is real, unless declared integer. GOD 双关“变量 GOD”或者“上帝”，real 双关“真的”或者“实数”。）

## 33.2 for: C 语言形式

最近的 bash 版本已经加入了第二种 for 命令语法，它类似于 C 语言形式，并且许多其他编程语言都支持这种形式。

```
for (( expression1; expression2; expression3 )); do
    commands
done
```

其中 expression 1、expression 2 和 expression 3 为算术表达式，commands 是每次循环都要执行的命令。

就执行结果而言，这种形式等同于如下结构。

```
(( expression1 ))
while (( expression2 )); do
    commands
    (( expression3 ))
done
```

expression1 用来初始化循环条件，expression2 用来决定循环何时结束，expression3 在每次循环末尾执行。

这里有一个典型的应用。

---

```
#!/bin/bash

# simple_counter : demo of C style for command

for (( i=0; i<5; i=i+1 )); do
    echo $i
done
```

---

执行后，它将输出如下结果。

---

```
[me@linuxbox ~]$ simple_counter
0
1
2
3
4
```

---

本例中，expression1 对变量 i 初始化赋值为 0，只要 i 的值小于 5，expression2 就允许循环继续，expression3 在每次循环重复时使 i 的值增加 1。

每当需要数值序列时, for 的 C 语言形式就能发挥作用了。接下来的两章将学习这种情况下的几个实际应用。

### 33.3 本章结尾语

学习了 for 命令之后, 我们将最后的改进应用到 sys\_info\_page 脚本中。目前, report\_home\_space 函数应该是这样。

---

```
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}
```

---

接下来, 我们将重写上述脚本, 使其可以为每个用户的主目录提供更多的细节, 并且包含每个用户的文件总数和子目录总数。

---

```
report_home_space () {

    local format="%8s%10s%10s\n"
    local i dir_list total_files total_dirs total_size user_name

    if [[ $(id -u) -eq 0 ]]; then
        dir_list=/home/*
        user_name="All Users"
    else
        dir_list=$HOME
        user_name=$USER
    fi

    echo "<H2>Home Space Utilization ($user_name)</H2>"

    for i in $dir_list; do

        total_files=$(find $i -type f | wc -l)
        total_dirs=$(find $i -type d | wc -l)
        total_size=$(du -sh $i | cut -f 1)
        echo "<H3>$i</H3>"
        echo "<PRE>"
        printf "$format" "Dirs" "Files" "Size"
        printf "$format" "----" "----" "----"
        printf "$format" $total_dirs $total_files $total_size
        echo "</PRE>"

    done
}
```

---

```
done  
return  
}
```

---

这次重写应用了很多目前为止学习过的知识。虽然仍要求超级用户运行，但是设置了一些随后在一个 `for` 循环中用到的变量，而不是运行一整套作为 `if` 语句部分的程序。此外，在函数中我们加入了几个局部变量，并利用 `printf` 按格式输出内容。

# 第 34 章

## 字符串和数字

计算机程序其实就是处理数据。前面的章节主要从文件层面讲解了数据的处理。然而，很多编程问题需要用到更小的数据单元，例如字符串和数字，来解决。

本章将学习几个用于操纵字符串和数字的 shell 脚本特性。shell 提供了多种字符串操作的参数扩展。除了算术扩展（在第 7 章讲到），还有一个常见的名为 bc 的命令行程序，它能执行更高层次的数学运算。

### 34.1 参数扩展 (Parameter Expansion)

虽然参数扩展在第 7 章就已出现，但是因为大部分参数扩展使用在脚本文档，而非命令行中，所以我们未加详细解释，在这之前已经使用了某些形式的参数扩展，例如 shell 变量。shell 提供了多种参数的扩展形式。

### 34.1.1 基本参数

参数扩展的最简单形式体现在平常对变量的使用中。举例来说，\$a 扩展后成为变量 a 所包含的内容，无论 a 包含什么。简单参数也可以被括号包围，例如 \${a}。这对扩展本身毫无影响，但是，当变量相邻于其他文本时，则必须使用括号，否则可能让 shell 混淆。看下面这个例子，我们试图以附加字符串\_file 到变量 a 内容后的方式新建一个文件名。

---

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

---

由于 shell 会试图扩展名为 a\_file 的变量而不是 a 变量，所以如果按序执行这些命令，结果将是一无所获。这个问题可以通过加上括号加以解决。

---

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

---

同样可见，大于 9 的位置参数可以通过给相应数字加上括号来访问。例如，访问第 11 个位置参数，可以这样做——\${11}。

### 34.1.2 空变量扩展的管理

有的参数扩展用于处理不存在的变量和空变量。这些参数扩展在处理缺失的位置参数和给参数赋默认值时很有用处。这种参数扩展形式如下。

`${parameter:-word}`

如果 parameter 未被设定（比如不存在）或者是空参数，则其扩展为 word 的值；如果 parameter 非空，则扩展为 parameter 的值。

---

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
substitute value if unset
[me@linuxbox ~]$ echo $foo
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
bar
[me@linuxbox ~]$ echo $foo
Bar
```

---

以下是另一种扩展形式，在里面使用等号，而非连字符号。

---

`${parameter:=word}`

---

如果 parameter 未被设定或者为空，则其扩展为 word 的值；此外，word 的

值也将赋给 parameter。如果 parameter 非空，则扩展为 parameter 的值。

---

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
default value if unset
[me@linuxbox ~]$ echo $foo
default value if unset
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
bar
[me@linuxbox ~]$ echo $foo
Bar
```

---

### 注意

位置参数和其他特殊参数不能以这种方式赋值。

我们使用一个问号，如下所示。

`${parameter:?word}`

如果 parameter 未设定或为空，这样扩展会致使脚本出错而退出，并且 word 内容输出到标准错误。如果 parameter 非空，则扩展结果为 parameter 的值。

---

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:??"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:??"parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

---

如果我们使用一个加号，如下所示：

`${parameter:+word}`

若 parameter 未设定或为空，将不产生任何扩展。若 parameter 非空，word 的值将取代 parameter 的值；然而，parameter 的值并不发生变化。

---

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+substitute value if set}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+substitute value if set}
substitute value if set
```

---

### 34.1.3 返回变量名的扩展

shell 具有返回变量名的功能。这种功能在相当特殊的情况下才会被用到。

---

```
 ${!prefix*}
 ${!prefix@}
```

该扩展返回当前以 prefix 开头的变量名。根据 bash 文档，这两种扩展形式执行效果一模一样。下面的例子中，我们列出了环境变量中所有以 BASH 开头的变量。

---

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION BASH_COMPLETION_DIR
BASH_LINENO BASH_SOURCE BASH_SUBSHELL BASH_VERSINFO BASH_VERSION
```

---

### 34.1.4 字符串操作

对字符串的操作，存在着大量的扩展集合。其中一些扩展尤其适用于对路径名的操作。扩展式

```
 ${#parameter}
```

扩展为 parameter 内包含的字符串的长度。一般来说，参数 parameter 是个字符串。然而，如果参数 parameter 是“@”或“\*”，那么扩展结果就是位置参数的个数。

---

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."
'This string is long.' is 20 characters long.
```

---

```
 ${parameter:offset}
 ${parameter:offset:length}
```

这个扩展用来提取一部分包含在参数 parameter 中的字符串。扩展以 offset 字符开始，直到字符串末尾，除非 length 特别指定。

---

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo:5}
string is long.
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

---

如果 offset 的值为负，默认表示它从字符串末尾开始，而不是字符串开头。注意，负值前必须有一个空格，以防和“\$”{parameter:-word} 扩展混淆。如果有 length（长度）的话，length 不能小于 0。

如果参数是“@”，扩展的结果则是从 offset 开始，length 为位置参数。

---

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

---

---

```
 ${parameter%pattern}
 ${parameter%%pattern}
```

根据 pattern 定义，这些扩展去除了包含在 parameter 中的字符串的主要部分。pattern 是一个通配符模式，类似那些用于路径名的扩展。两种形式的区别在于“#”形式去除最短匹配，而“##”形式去除最长匹配。

---

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
Zip
```

---

```
 ${parameter%pattern}
 ${parameter%%pattern}
```

这些扩展与上述的“#”和“##”扩展相同，除了一点——它们从参数包含的字符串末尾去除文本，而非字符串开头。

---

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%.*}
file.txt
[me@linuxbox ~]$ echo ${foo%.*}
file
```

---

```
 ${parameter/pattern/string}
 ${parameter//pattern/string}
 ${parameter/#pattern/string}
 ${parameter/%pattern/string}
```

这个扩展在 parameter 的内容上执行搜索和替换非常有效。如果文本被发现和通配符 pattern 一致，就被替换为 string 的内容。通常形式下，只有第一个出现的 pattern 被替换。在“//”形式下，所有 pattern 都被替换。“/#”形式要求匹配出现在字符串开头，“/%”形式要求匹配出现在字符串末尾。“/string”可以省略，不过和 pattern 匹配的文本都会被删除。

---

```
[me@linuxbox ~]$ foo=JPG.JPG
[me@linuxbox ~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox ~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

参数扩展是一个比较重要的功能。进行字符串操作的扩展可以替代其他常用的命令，例如 set 和 cut 命令。扩展通过取代外部程序，改善了脚本的执行效率。比如，我们将改动前面章节中讨论过的 longest-word 程序，运用参数扩展\${#}代替在 subshell 中输出结果的\$(echo \$j | wc -c)，如下所示。

---

```

#!/bin/bash

# longest-word3 : find longest string in a file

for i; do
    if [[ -r $i ]]; then
        max_word=
        max_len=
        for j in $(strings $i); do
            len=${#j}
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
    shift
done

```

---

接着，我们将用 time 命令比较两个版本的效率。

---

```

[me@linuxbox ~]$ time longest-word2 dirlist/usr-bin.txt
dirlist/usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38 characters)

real    0m3.618s
user    0m1.544s
sys     0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist/usr-bin.txt
dirlist/usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38 characters)

real    0m0.060s
user    0m0.056s
sys     0m0.008s

```

---

原始版本的脚本花费了 3.618 秒来扫描 text 文件，而使用参数扩展的新版本只花费了 0.06 秒——这是一个比较大的改进。

## 34.2 算术计算和扩展

第 7 章我们学习了算术扩展，用来对整数进行算术运算。它的基本形式如下所示：

`$((expression))`

其中 `expression` 是一个有效的算术表达式。

这和用于算法计算（真实性测试）的复合命令 “`(( ))`” 有关，我们曾在第 27 章中遇到过。

通过前面章节的学习，我们已经了解了表达式和运算符的常见类型。下面，

我们将更为完整地学习它们。

### 34.2.1 数字进制

回顾第 9 章，我们已经学习了八进制（octal）和十六进制（hexadecimal）的数字。在算术表达式中，shell 支持任何进制表示的整数。表 34-1 列出了基本数字进制的描述。

表 34-1 不同的数字进制

符号	描述
Number	默认情况下，number 没有任何符号，将作为十进制数字
0number	在数字表达式中，以 0 开始的数字被认为是八进制数字
0xnumber	十六进制符号
'base#number	base 进制的 number

看一些例子，如下所示。

---

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

---

这些例子中，我们输出了十六进制数字 ff（最大的两位数字）的值以及最大的八位数（二进制）。

### 34.2.2 一元运算符

有两种一元运算符：+ 和 -。它们分别被用来指示一个数字是正或是负。

### 34.2.3 简单算术

表 34-2 列出了普通算术运算符。

表 34-2 算术操作符

操作符	描述
+	加法
-	减法
*	乘法
/	整除
**	求幂
%	取模（余数）

这里大部分操作符具有自描述性，但是整数除法和取模需要更深入的讨论。

由于 shell 的算术运算符仅适用于整数，除法的结果永远是完整的数字，如下所示。

---

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

---

这使得除法运算中余数的确定尤为重要，如下所示。

---

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))
1
```

---

通过运用除法和取模运算，可以确定 5 被 2 整除的结果为 2，余数为 1。

计算余数在循环中很有用。它使得一个运算符能够在循环的特定间隔中执行。在下例中，我们显示了一行数字，其中 5 的倍数突出显示。

---

```
#!/bin/bash

# modulo : demonstrate the modulo operator

for ((i = 0; i <= 20; i = i + 1)); do
    remainder=$((i % 5))
    if (( remainder == 0 )); then
        printf "<%d> " $i
    else
        printf "%d " $i
    fi
done
printf "\n"
```

---

运行后，结果如下。

---

```
[me@linuxbox ~]$ modulo
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

---

### 34.2.4 赋值

尽管算术表达式并非立等可见，但是它们可以用来进行赋值。通过前面不同的场景，我们已经进行了多次赋值。每当赋给变量一个值时，就是赋值操作。算术表达式可以这样使用。

---

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ if (( foo = 5 ));then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

---

上例中，先给变量 `foo` 赋空值，并确认它确实为空。接着，执行一个以复

杂命令 ((foo = 5)) 为条件的 if 语句。整个过程有两件有趣的事。(1) 它给变量 foo 赋值 5。(2) 因为赋值是成功的，所以条件为 true。

### 注意

记住上式中 “=” 的确切含义很重要。单个的 “=” 执行赋值，即 `foo = 5` 意味着 “让 `foo` 等于 5”。两个 “==” 用来判断是否相等，即 `foo == 5` 意味着 “`foo` 是否等于 5？”这可能十分令人费解，因为 `test` 命令认为单个的 “=” 判断字符串是否相等。但这也正是另一个使用新式的 “[[]]” 和 “(( ))” 混合命令代替 `test` 的理由。

此外，除了 “=” 之外，shell 还提供了一些相当有用的赋值语句，如表 34-3 所示。

表 34-3 赋值操作符

运算符	描述
<code>parameter = value</code>	简单赋值运算。赋予 <code>parameter</code> 值为 <code>value</code>
<code>parameter += value</code>	加法运算。等价于 <code>parameter = parameter + value</code>
<code>parameter -= value</code>	减法运算。等价于 <code>parameter = parameter - value</code>
<code>parameter *= value</code>	乘法运算。等价于 <code>parameter = parameter * value</code>
<code>parameter /= value</code>	整除运算。等价于 <code>parameter = parameter / value</code>
<code>parameter %= value</code>	取模运算。等价于 <code>parameter = parameter % value</code>
<code>parameter++</code>	变量后增量运算。等价于 <code>parameter=parameter+1</code> （查看后面的讨论）
<code>parameter--</code>	变量后减量运算。等价于 <code>parameter=parameter-1</code>
<code>++parameter</code>	变量前增量运算。等价于 <code>parameter=parameter+1</code>
<code>--parameter</code>	变量前减量运算。等价于 <code>parameter=parameter-1</code>

这些赋值操作作为很多常见算术任务提供了一种快捷方式。增量 (++) 和减量 (--) 运算特别有意义，它们以 1 为间隔增加或减少参数的值。这种风格的表示法是从 C 编程语言衍生而来，并且已经被其他几种编程语言采用，其中包括 bash。

---

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
2
```

---

这些操作既可能在参数前部也可能在参数尾部出现。虽然它们都以 1 为间隔增加或减少参数的值，两者的位置安排有一个微妙的区别。如果在参数前部，参数在返回前增加（或减少）。如果在参数尾部，该操作在参数返回后执行。这

十分奇怪，但却是故意为之。下面是一个示例。

---

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
2
[me@linuxbox ~]$ echo $foo
2
```

---

如果给变量 `foo` 赋值 1，然后用“`++`”操作增加它的值，“`++`”位置在参数名后，那么 `foo` 返回值为 1。然而，如果再次查看变量的值，会发现该值已经增加 1。如果“`++`”位置在参数名前，将得到比较期望的结果，如下所示。

对于大部分 shell 应用，前置运算操作则是最常用的。

“`++`”和“`--`”操作符经常和循环联合使用。下面我们将对 `modulo` 脚本做些改善，使它变得更为紧凑。

---

```
#!/bin/bash

# modulo2 : demonstrate the modulo operator

for ((i = 0; i <= 20; ++i )); do
    if (((i % 5) == 0 )); then
        printf "<%d> " $i
    else
        printf "%d " $i
    fi
done
printf "\n"
```

---

### 34.2.5 位操作

有一种操作符以一种非同寻常的方式巧妙地进行数字运算，这些操作符在位层面执行运算。它们被用于特定的低层任务中，常用来位标志的设置和读取。表 34-4 列出了位操作符。

表 34-4 位操作

操作符	描述
<code>~</code>	按位取反。将数字里的每一位取反
<code>&lt;&lt;</code>	逐位左移。将数字里的每一位向左移位
<code>&gt;&gt;</code>	逐位右移。将数字里的每一位向右移位
<code>&amp;</code>	按位与。对数字里的每一位执行与操作
<code> </code>	按位或。对数字里的每一位执行或操作
<code>^</code>	按位异或。对两个数字的每一位执行异或操作

注意，除了按位取反之外，也存在相应的赋值操作（例如“`<=>`”）。

这里将示范使用逐位左移操作，产生 2 的次方的一串数字。

---

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
1
24
8
16
32
64
128
```

---

### 34.2.6 逻辑操作

正如在第 7 章中所述，“`(( ))`” 复合命令支持多种比较操作。有另外几种可以被用于判断逻辑是否成立的操作。表 34-5 列出了完整的清单。

表 34-5 Comparison Operators

操作符	描述
<code>&lt;=</code>	小于或等于
<code>&gt;=</code>	大于或等于
<code>&lt;</code>	小于
<code>&gt;</code>	大于
<code>=</code>	等于
<code>!=</code>	不等于
<code>&amp;&amp;</code>	逻辑与
<code>  </code>	逻辑或
<code>expr1?expr2:expr3</code>	比较（三元组）操作。如果表达式 <code>expr1</code> 非零（算术为 <code>true</code> ），那么执行 <code>expr2</code> ，否则执行 <code>expr3</code>

---

当使用逻辑操作时，表达式遵循算术逻辑的规则。这就是说，值为零的表达式为 `false`，而非零表达式为 `true`。如下所示，“`(( ))`” 复合命令将结果映射到 shell 的正常退出代码。

---

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

---

最奇怪的逻辑操作是三元操作。这个操作（该操作模仿 C 编程语言中的相应操作）执行一个独立的逻辑测试。它可以被用作某种意义上的 `if/then/else` 语

句。它作用于三个算术表达式上（不可以是字符串），并且如果第一个表达式为真（或非零），就执行第二个表达式，否则执行第三个表达式。我们可以在命令行上尝试一下。

---

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

---

这是一个实际的三元操作，本例实现了一个来回切换。每次操作被执行，变量 a 的值从 0 变为 1，或从 1 变为 0。

请注意在表达式内的赋值操作并不能简单使用。当试图这样做时，`bash` 将输出一个错误。

---

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: (( a<1?a+=1:a-=1: attempted assignment to non-variable (error token is
"-=1"))
```

---

这个问题可以通过使用括号包围赋值表达式来解决。

---

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

---

接下来，我们将研究一个更为综合的例子，该例在脚本中使用算术操作产生一个简单的数字表。

---

```
#!/bin/bash

# arith-loop: script to demonstrate arithmetic operators

finished=0
a=0
printf "a\t${a**2}\t${a**3}\n"
printf "\t====\t====\n"

until ((finished)); do
    b=$((a**2))
    c=$((a**3))
    printf "%d\t%d\t%d\n" $a $b $c
    ((a<10?++a:(finished=1)))
done
```

---

在这个脚本中，基于 `finished` 变量的值实现了一个 `until` 循环。最初，该变量被设为 0（算术假），循环继续，直到变量成为非零值。在循环体内，计算计数变量 a 的平方和立方。在循环最后，判断计数变量的值。如果它小于 10（最大迭代次数），就加 1，否则给变量 `finished` 赋值 1，使得 `finished` 算术为真，从而

终止循环。运行脚本，将得到如下结果。

---

```
[me@linuxbox ~]$ arith-loop
a    a**2      a**3
=    ===      ===
0    0          0
1    1          1
2    4          8
3    9         27
4   16         64
5   25        125
6   36        216
7   49        343
8   64        512
9   81        729
10  100       1000
```

---

### 34.3 bc：一种任意精度计算语言

我们已经了解了 shell 可以处理所有种类的整数运算，但是如果需要执行更高级的数学运算，或者甚至使用浮点数怎么办呢？答案是无法实现，至少无法用 shell 直接实现。为了达到这个目的，我们需要使用一个外部程序。这里有几种方法可以使用，如嵌入 Perl 或 AWK 是一个可能的解决方案。但不幸的是，这已超出本书范围。

另一种方法是使用一个专门的计算器程序，大多数 Linux 系统都支持这种程序 bc。

bc 程序读取一个使用类 C 语言编写的程序文件，并执行它。bc 脚本可以是一个单独的文件，也可以从标准输入中读取。bc 语言支持很多功能，包括变量、循环以及由程序员自定义的函数。在这里我们不会完整地涵盖 bc 的知识点，而只是抛砖引玉。bc 的 man 手册已有充分详细的说明。

以一个浅显易懂的例子开始，我们下面将写一个 2 加 2 的 bc 脚本。

---

```
/* A very simple bc script */

2 + 2
```

---

脚本的第一行是一个注释。bc 使用和 C 编程语言同样的注释语法。注释可以跨多行，以 “\*” 开始，以 “\*/” 结束。

#### 34.3.1 bc 的使用

如果将上述 bc 脚本保存为 foo.bc，那么可以这样运行它。

---

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4
```

---

如果仔细看看，可以在最底部版权信息的后面看到运行结果。这些信息可以通过`-q`(quiet)选项禁止显示。

`bc` 也可以交互地使用，如下所示。

---

```
[me@linuxbox ~]$ bc -q
2 + 2
4
quit
```

---

当交互地使用 `bc` 时，只需简单地输入运算值，计算结果就会立刻被显示出来。使用 `bc` 命令中的 `quit` 结束交互会话。

通过标准输入传递一个脚本到 `bc` 亦是可行的，如下所示。

---

```
[me@linuxbox ~]$ bc < foo.bc
4
```

---

既然支持标准输入，那么意味着可以使用嵌入文档、嵌入字符串和管道传递脚本。下面是一个嵌入字符串的例子。

---

```
[me@linuxbox ~]$ bc <<< "2+2"
4
```

---

### 34.3.2 脚本例子

作为一个实际的例子，我们将构建一个常见运算的脚本——按月偿还贷款。以下脚本使用嵌入文档传递脚本到 `bc`。

---

```
#!/bin/bash

# loan-calc : script to calculate monthly loan payments

PROGNAME=$(basename $0)

usage () {
    cat <<- EOF
Usage: $PROGNAME PRINCIPAL INTEREST MONTHS
    Where:
        PRINCIPAL is the amount of the loan.
```

```

INTEREST is the APR as a number (7% = 0.07).
MONTHS is the length of the loan's term.

EOF
}
if (($# != 3)); then
    usage
    exit 1
fi

principal=$1
interest=$2
months=$3

bc <<- EOF
scale = 10
i = $interest / 12
p = $principal
n = $months
a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
print a, "\n"
EOF

```

---

执行后，结果如下。

---

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270.7222490000
```

---

本例计算了\$135,000 贷款的月付款数，180 个月（15 年）的年度百分率为 7.75%。注意答案的精度，它是由赋给 bc 脚本中 scale 特定变量的值决定的。bc 的 man 手册提供了 bc 脚本编程语言的完整描述。虽然它的数学表示法与 shell 的稍微不同（bc 更接近于 C 语言），但是就本书目前涵盖的知识而言，大部分内容都是相似的。

## 34.4 本章结尾语

本章我们学习了脚本中许多“实用的”小技巧。随着脚本编程经验的增长，有效而巧妙地操纵字符串和数字将会是极其珍贵的。本书中的 loan-calc 脚本，说明了哪怕是最简单的脚本也可以做一些真正有用的事。

## 34.5 附加项

虽然 loan-calc 脚本的基本功能已经实现，但是这个脚本远非完善。读者可以试着改善 loan-calc 脚本，使其包括下面的功能。

- 对命令行参数的充分验证。
- 用来实现“交互”模式的命令行选项，该模式提示用户输入贷款的本金、利率和偿还期。
- 更好的输出格式。

# 第 35 章

## 数    组

在上一章中，我们了解了 shell 如何操作字符串和数字。到目前为止，我们所接触到的数据类型在计算机科学领域被称为标量变量（scalar variable），也就是说，该变量包含一个单一值。

在本章中，我们将会学习一种包含多个值的数据结构——数组。事实上，数组几乎是所有程序设计语言的一大特点。尽管 shell 对数组的支持有限，但它对于解决程序设计问题是非常有帮助的。

### 35.1 什么是数组

数组是可以一次存放多个值的变量，数组的组织形式如同表格一样。下面以电子表格为例。一个电子表格就像一个二维数组一样。它由行和列组成，根据行与列的地址可以在电子表格里标识每一个独立单元的位置。数组也是以这种方式工作的。数组中的单元叫做元素，并且每个元素中含有数据。使用一种

叫做索引或是下标的地址就可以访问一个独立的数组元素。

大多数的程序设计语言支持多维数组。电子表格就是多维数组的一个实例，该数组是由宽度和高度两个维度组成的二维数组。尽管最经常使用的可能是二维和三维数组，但是很多语言支持任意维数的数组。

**bash** 中的数组是一维的。可以将它想象成只有一列的电子表格。尽管有这个限制，但是它们还是有很多的应用。**bash** 的第二个版本开始提供对数据的支持。而最初的 UNIX shell 程序 **sh** 是不支持数组的。

## 35.2 创建一个数组

命名数组变量同命名其他 **bash** 变量一样，当访问数组变量时可以自动创建它们。实例如下。

---

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

---

这里我们看到的是赋值和访问数组元素的例子。通过第一条命令，可将值 **foo** 赋给数组 **a** 的元素 1。第二条命令显示了元素 1 的存储值。在第二条命令中使用花括号是为了阻止 **shell** 在数组元素名里试图扩展路径名。

使用 **declare** 命令也可以创建数组，如下所示。

---

```
[me@linuxbox ~]$ declare -a a
```

---

这是使用选项-a 和 **declare** 创建数组 **a** 的实例。

## 35.3 数组赋值

赋值的方式可以有两种。使用下面的语法可以赋单一值。

*name[subscript]=value*

这里的 **name** 是数组名，并且 **subscript** 是大于或等于 0 的整数（或算术表达式）。要注意的是，数组的第一个元素是 **subscript0**，而不是 1。**value** 是赋给数组元素的字符串或整数。

使用下面的语法可以赋多值。

*name=(value1 value2...)*

这里的 name 是数组名，并且将 value1 value2... 等值依次赋予从元素 0 开始的数组元素。例如，如果想要将一星期中天数的缩写赋给数组 days，那么我们可以像下面这样赋值。

---

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

---

通过为每个值指定一个下标来给特定元素赋值也是可行的。

---

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu [5]=Fri [6]=Sat)
```

---

## 35.4 访问数组元素

那么数组有哪些应用呢？就像使用电子表格程序可以执行很多数据管理任务一样，使用数组可以完成很多程序设计任务。

以一个简单的数据采集和表示为例。创建一个脚本，用于校验特定目录中文件的修改次数。从这些数据来看，脚本将会输出一个表格来显示文件最后一次修改发生在一天中的什么时候。使用这样的一个脚本可以来检测什么时候系统是最活跃的。这个称之为 hours 的脚本产生的结果如下。

---

```
[me@linuxbox ~]$ ./hours .
Hour      Files     Hour      Files
-----
00          0        12       11
01          1        13        7
02          0        14        1
03          0        15        7
04          1        16        6
05          1        17        5
06          6        18        4
07          3        19        4
08          1        20        1
09         14        21        0
10          2        22        0
11          5        23        0
Total files = 80
```

---

执行 hours 程序，并指定当前目录为目标目录，将产生一个表格用来显示在一天中每个小时（0~23）有多少文件经过最后一次修改。产生表格的代码如下。

---

```
#!/bin/bash

# hours : script to count files by modification time

usage () {
    echo "usage: $(basename $0) directory" >&2
}
```

---

```

# Check that argument is a directory
if [[ ! -d $1 ]]; then
    usage
    exit 1
fi

# Initialize array
for i in {0..23}; do hours[i]=0; done

# Collect data
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
    j=$((i/#0)
        ((++hours[j]))
        ((++count)))
done

# Display data
echo -e "Hour\tFiles\thour\tFiles"
echo -e "----\t----\t----\t----"
for i in {0..11}; do
    j=$((i + 12))
    printf "%02d\t%d\t%02d\t%d\n" $i ${hours[i]} $j ${hours[j]}
done
printf "\nTotal files = %d\n" $count

```

---

这个脚本包含了一个函数（`usage`）和由 4 部分组成的一个主函数。在第一部分，我们检测到一个命令行参数并且这是一个目录。如果不是的话，显示 `usage` 信息，同时退出。

第二部分初始化数组 `hours`。通过给每个元素赋 0 值来实现。尽管在调用数组之前对数组中的元素没有特殊要求，但是脚本需要保证没有元素是空的。注意一下有趣的循环创建方式，通过使用花括号扩展（`{0..23}`），能够很容易地为 `for` 循环生成一系列初始值。

第三部分通过运行 `stat` 程序遍历目录中每个文件来采集数据。使用 `cut` 选项从结果中提取两位数的小时数（`hour`）。在循环内部，需要清除 `hour` 域中的前导值 0，这是因为 shell 将要试图（最终失败了）以八进制的形式来表示数值 00~09（见表 34-1）。接下来，将与一天中的小时数相对应的数组元素值增加 1。最后，使用一个计数器（`count`）来追踪目录里文件的总数目。

脚本的最后一部分显示了数组的内容。首先我们输出几个标题行，然后，进入一个产生两列输出的循环。最后，输出文件的最终统计结果。

## 35.5 数组操作

有很多常见的数组操作。比如删除数组、确定数组大小和排序等在脚本中

有很多应用。

### 35.5.1 输出数组的所有内容

我们可以使用下标“\*”和“@”来访问数组中的每个元素。对于定位参数来讲，符号“@”较之更有用。例证如下。

---

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done
a dog a cat a fish
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

---

我们创建了数组 `animals`，并使用 3 个双单词字符串为其赋值，然后执行 4 个循环以便观察单词拆分对数组内容的影响。如果对符号  `${animals[*]}`  和  `${animals[@]}`  加以引用，就会得到不同的结果。符号“\*”将数组所有内容放在一个字中，而符号“@”使用 3 个字来显示数组的真实内容。

### 35.5.2 确定数组元素的数目

使用参数扩展，我们可以采用类似获取字符串长度的方式来确定数组中元素的数目。实例如下。

---

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]} # number of array elements
1
[me@linuxbox ~]$ echo ${#a[100]} # length of element 100
3
```

---

我们首先创建了数组 `a`，并且将字符串 `foo` 赋给第 100 个元素。接下来，我们使用符号“@”通过参数扩展来确定数组长度。最后，我们查看包含字符串

foo 的元素 100 的长度。值得一提的是，当将字符串赋给元素 100 时，bash 报告数组中只有一个元素。这与其他一些编程语言的行为是不同的。在这些语言中，数组中未使用的元素（元素 0~99）初始化为空值并参与计数。

### 35.5.3 查找数组中使用的下标

由于 bash 允许在下标赋值中包含“空格”，有时这对确定实际存在的元素是很有用的。这可以过参数扩展来实现，其形式如下。

```
 ${!array[*]}
 ${!array[@]}
```

这里的 array 是数组变量名。就像符号“\*”和“@”等参数扩展一样，引用中含有的“@”形式是最有用的，因为它将数组内容扩展成独立的单词。

---

```
[me@linuxbox ~]$ foo=(2=a 4=b 6=c)
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done
a
bc
[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done
24
6
```

---

### 35.5.4 在数组的结尾增加元素

如果在数组的结尾需要添加元素的话，知道数组中元素的数目是没有用的，因为符号“\*”和“@”返回的值并不会告诉我们使用的最大数组索引是什么。幸运的是，shell 提供了一种解决方法。通过使用“+=”赋值运算符，可以在数组的尾部自动地添加元素。这里，我们将 3 个值赋给数组 foo，然后再添加 3 个元素。

---

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

---

### 35.5.5 数组排序操作

就像电子表格一样，通常需要将数据列中的值进行排序。shell 虽然没有直接的方式来完成排序功能，但是用一些代码来完成并非难事。

---

```
#!/bin/bash

# array-sort : Sort an array
```

---

```
a=(f e d c b a)
echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
echo "Sorted array: ${a_sorted[@]}"
```

---

当执行时，脚本会产生如下的内容。

---

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array: a b c d e f
```

---

脚本巧妙地使用一个替换命令将原数组（a）的内容复制到数组（a\_sorted）中。通过改变设计流程，我们可以这个基本的技术就可被用来执行数组中的多种操作。

### 35.5.6 数组的删除

使用 `unset` 命令，我们可以删除数组，如下所示。

---

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}

[me@linuxbox ~]$
```

---

我们也可以使用 `unset` 来删除单个数组元素。

---

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset 'foo[2]'
[me@linuxbox ~]$ echo ${foo[@]}
a b d e f
```

---

在这个例子里，我们删除了数组的第 3 个元素（下标为 2）。记住，数组是以下标 0 开始的，而不是 1 开始的！同时需要注意的是，我们必须引用数组元素来阻止 shell 执行路径名扩展。

很有趣的是，对数组元素赋一个空值并不意味着清空它的内容，如下所示。

---

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo[@]}
b c d e f
```

---

任何涉及到不含下标的数组变量的引用指的是数组中的元素 0，如下所示。

---

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

---

---

```
[me@linuxbox ~]$ foo=A  
[me@linuxbox ~]$ echo ${foo[@]}  
A b c d e f
```

---

## 35.6 本章结尾语

如果在 bash 手册文档中搜索 array，我们会发现在很多使用数组变量的实例。大多数实例是相当令人费解的，但是在一些特殊情况下可能会提供临时效用。事实上，在 shell 程序设计中，未能非常充分地涵盖数组的所有内容，很大程度上是因为传统的 UNIX shell 程序（比如 sh）缺乏对数组的支持。传统 shell 对数组普遍缺乏支持是很不幸的，因为数组广泛运用于其他的程序设计语言，并且提供强有力的工具来解决多种程序设计问题。

数组和循环本身具有密切的关系，并且经常被一块使用。下面的循环形式非常适合估算数组下标。

```
for ((expr1; expr2; expr3))
```

# 第36章

## 其他命令

在本书最后一章，我们将会讲解一些琐碎零散的知识。虽然我们在前面的章节中已经学习了很多知识，但是还有大量的 bash 特性没有涉及到。对于那些整合在 Linux 发行版中的 bash，其中的绝大多数是相当令人费解的，但是它们却很有帮助。除此之外，还有一些命令，虽然不经常用，却对特定的程序设计问题大有帮助。下面我们将学习这方面内容。

### 36.1 组命令和子 shell

bash 允许将命令组合到一起使用，这有两种方式，一种是利用组命令，另一种是使用子 shell。下面是这两种方式的语法实例。

组命令：

```
{ command1; command2; [command3; ...] }
```

子 shell：

---

```
(command1; command2; [command3; ...])
```

这两种形式的区别在于，组命令使用花括号将其命令括起来，而子 shell 则用圆括号。值得注意的是，在 bash 实现组命令时，必须使用一个空格将花括号与命令分开，并且在闭合花括号前使用分号或是换行来结束最后的命令。

### 36.1.1 执行重定向

那么组命令和子 shell 有什么用途呢？尽管它们有一处主要的区别（马上将会涉及这一点），但是它们都可以用来管理重定向。下面让我们看一个在多个命令中执行重定向的脚本段。

---

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

---

显而易见，3 条命令将输出重定向为 *output.txt* 文件。使用组命令，可以按照如下方式编码。

---

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

---

使用子 shell 时也是一样，如下所示。

---

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

---

使用这个技术，可以减少一些输入，但是组命令或是子 shell 真正有价值的地方在于管道的使用。当创建命令管道时，通常将多条命令的结果输出到一条流中，这很有用。组命令和子 shell 使得这一点变得简单，如下所示。

---

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

---

这里我们已经将 3 个命令的输出进行合并，并通过管道输出到 lpr 的输入以产生一个打印报告。

### 36.1.2 进程替换

虽然组命令和子 shell 看起来相似，都可以用来为重定向整合流，但是，它们有一处主要的不同。子 shell（正如名字所示）在当前 shell 的子拷贝中执行命令，而组命令在当前 shell 里执行所有命令。这意味着子 shell 复制当前的环境变量以创建一个新的 shell 实例。当子 shell 退出时，复制的环境变量也就消失了，因此，任何对子 shell 环境（包括变量赋值）的改变也同样丢失了。所以，大多数情况下，除非脚本需要子 shell，否则组命令比子 shell 更可取。组命令更快，并

且需要更少的内存。

在第 28 章中，我们已经接触到了子 shell 环境存在问题的一个实例，管道中的 `read` 命令没有像先前预期的那样工作。我们可以采用如下的方式重新创建管道。

---

```
echo "foo" | read
echo $REPLY
```

---

`REPLY` 变量的内容总是为空，因为 `read` 命令是在子 shell 中执行的，并且当子 shell 终止的时候，`REPLY` 的拷贝也遭到了破坏。

由于总是在子 shell 中执行管道中的命令，任何变量赋值的命令都会遇到这个问题。很幸运的是，shell 提供了一种叫做进程替换的外部扩展方式来解决这个问题。

实现进程替换有两种方式，一种是产生标准输出的进程，如下所示。

```
<(list)
```

另一种是吸纳标准输入的进程，如下所示。

```
>(list)
```

这里的 `list` 是一系列的命令。

为了解决上述 `read` 命令的问题，我们可以像这样使用进程替换。

---

```
read < <(echo "foo")
echo $REPLY
```

---

进程替换允许将子 shell 的输出当做普通文件，目的是为了重定向。事实上，这是一种扩展形式，我们可以查看它的真实值。

---

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

---

通过使用 `echo` 来查看扩展结果，可以看到文件 `/dev/fd/63` 正为子 shell 提供输出。

进程替换通常结合带有 `read` 的循环使用。这里有一个读循环的实例，该读循环用来处理子 shell 创建的目录列表的内容。

---

```
#!/bin/bash

# pro-sub : demo of process substitution

while read attr links owner group size date time filename; do
    cat <<- EOF
        Filename:  $filename
        Size:      $size
    EOF
done
```

---

---

```

Owner:      $owner
Group:      $group
Modified:   $date $time
Links:      $links
Attributes: $attr
EOF
done < (ls -l | tail -n +2)

```

---

循环在目录列表的每一行执行 read 操作。脚本的最后一行产生了列表本身。这一行将进程替换的输出重新定向到循环的标准输入。进程替换管道中的 tail 命令用来清除列表的第一行，之后这一行不再需要了。

当执行这个命令的时候，脚本产生如下的输出。

---

```

[me@linuxbox ~]$ pro_sub | head -n 20
Filename: addresses.ldif
Size:      14540
Owner:     me
Group:     me
Modified:  2012-04-02 11:12
Links:     1
Attributes: -rw-r--r-

Filename: bin
Size:      4096
Owner:     me
Group:     me
Modified:  2012-07-10 07:31
Links:     2
Attributes: drwxr-xr-x

Filename: bookmarks.html
Size:      394213
Owner:     me
Group:     me

```

---

## 36.2 trap

在第 10 章中，我们了解了程序如何响应信号。同样我们也可以将这种功能应用到脚本中。虽然到目前为止，我们所写的脚本还不需要这种功能（因为它们有着更短的执行时间，并且不产生临时文件），但是，拥有一个信号处理程序可能对庞大复杂的脚本大有裨益。

当设计一个庞大复杂的脚本时，我们一定要考虑到，如果在脚本正在运行时，用户注销或是关闭电脑时会发生什么情况。当这样的情况发生时，将会把信号发送到所有受影响的进程。相应地，执行那些进程的程序能够通过一些操作来保证程序合理有序地结束。设想一下，比如说，脚本在执行的时候创建了一个临时文件。在一个好的设计中，当脚本结束工作时，该临时文件会自动删

除。如果接收的信号表明将要过早地结束程序，这个时候让脚本删除这个文件也是很明智的。

为了实现这个目的，bash 提供了一种 trap 机制。内置命令 trap 可以恰如其分地实现 trap 机制。trap 命令使用的语法如下。

```
trap argument signal [signal...]
```

这里的 argument 是作为命令被读取的字符串，而 signal 是对信号量的说明，该信号量将会触发解释命令的执行。

下面是一个简单的例子。

---

```
#!/bin/bash

# trap-demo : simple signal handling demo

trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

---

每当运行中的脚本接收到 SIGINT 或者 SIGTERM 信号时，脚本定义的 trap 将执行 echo 命令。当用户通过按下 Ctrl-C 键来试图结束脚本时，程序的执行情况如下。

---

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
I am ignoring you.
Iteration 3 of 5
I am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

---

可以看出，每次用户试图中断程序时，都会输出这样的信息。

构造一个字符串来形成一系列有用的命令看起来是很笨拙的，因此，通常的做法是指定 shell 函数来代替命令。在下面的例子中，我们为每个将要处理的信号指定一个独立的 shell 函数。

---

```
#!/bin/bash

# trap-demo2 : simple signal handling demo

exit_on_signal_SIGINT () {
    echo "Script interrupted." 2>&1
    exit 0
```

```

}
exit_on_signal_SIGTERM () {
    echo "Script terminated." 2>&1
    exit 0
}

trap exit_on_signal_SIGNAL SIGINT
trap exit_on_signal_SIGTERM SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done

```

这个脚本为两个不同的信号定义了相应的 trap 命令。当接收到特定信号的时候，每个 trap 相应地执行指定的 shell 函数。注意到每个信号处理函数中的 exit 命令。如果没有 exit 命令，脚本会循环执行该函数。

在脚本执行的过程中，当用户按下 Ctrl-C 键时，产生的结果如下。

---

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
Script interrupted.
```

---

### 临时文件

在脚本中使用信号控制句柄，是为了删除脚本执行过程中用于保存中间变量的临时文件，一些术语称其为临时文件。传统意义上来说，类 UNIX 系统中的程序在 /tmp 目录里创建临时文件，该目录是用于保存临时文件的共享目录。但是，由于目录是共享的，不可避免地产生安全上的考虑，尤其是超级用户特权下运行的程序。除了为所有系统用户都可以访问的文件设置适当的权限之外，赋给临时文件一个不可预知的文件名是非常重要的。这就避免了临时快速攻击 (temp race attack) 的漏洞。创建一个不可预知（但是仍然是可以描述的）的名称的方法如下所示。

```
 tempfile=/tmp/${basename $0}.$$.RANDOM
```

这将会创建一个包含程序名的文件名，紧随其后的是进程 ID (PID) 以及一个随机整数。但是，要注意的是，\$RANDOM shell 变量返回一个范围仅在 1~32767 之间的值，在计算机领域里，这并不是一个很大的范围，因此，单一的变量实例不足以抵御一个不达目的不罢休的攻击者。

比较好的一个方法是使用 mktemp 程序（不要与 mktemp 标准库函数相混淆）来命名和创建临时文件。mktemp 程序使用模板作为参数来创建文件名。这个模板应该包括一系列的 X 字符，可以用相应的随机字母和数字来代替这

些 X 字符。X 字符的序列越长，随机字符的序列就会越长。下面是一个实例。

```
 tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

这就创建了一个临时文件，并且将其名称赋给了变量 `tempfile`。随机的字母和数字代替了模板中的字符 X。那么，最终的文件名（在这个例子里，最后的文件名也包含了特殊参数`$$`的扩展值以获得 PID）可能会如下所示。

```
/tmp/foobar.6593.U0ZuvM6654
```

尽管 `mktemp` 手册页声明，`mktemp` 构造了一个临时文件名，但是它同时也创建了这个文件。

如果是普通用户执行的脚本，更为明智的做法是避免使用`/tmp` 目录，而在用户主目录下而为临时文件创建一个目录，其代码如下。

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

## 36.3 异步执行

有的时候我们希望同时执行多项任务。众所周知，所有现代的操作系统即便不是多用户系统，至少也是多任务系统。脚本可以在多任务中运行。

这涉及到父脚本以及一个或多个子脚本的加载问题，子脚本可以在父脚本运行时执行其他额外的任务。但是，当一系列脚本以这种方式运行的时候，保持父脚本与子脚本的协调一致就会是一个问题。也就是说，试想这样一种情况，如果父脚本与子脚本彼此相互依赖，一个脚本必须等待另一个脚本任务完成之后才能继续完成自己的任务。

`bash` 提供了一个内置的命令来帮助管理异步执行。`wait` 命令可以让父脚本暂停，直到指定的进程（比如子脚本）结束。

### 36.3.1 wait 命令

首先，我们来演示 `wait` 命令。为此我们需要两个脚本来完成这个过程。下面是一个父脚本。

---

```
#!/bin/bash

# async-parent : Asynchronous execution demo (parent)

echo "Parent: starting..."
```

---

```

echo "Parent: launching child script..."
async-child &
pid=$!
echo "Parent: child (PID= $pid) launched."

echo "Parent: continuing..."
sleep 2

echo "Parent: pausing to wait for child to finish..."
wait $pid

echo "Parent: child is finished. Continuing..."
echo "Parent: parent is done. Exiting."

```

---

下面是一个子脚本。

---

```

#!/bin/bash

# async-child : Asynchronous execution demo (child)

echo "Child: child is running..."
sleep 5
echo "Child: child is done. Exiting."

```

---

在这个例子中，我们可以看出子脚本内容非常简单，父脚本执行实际的操作。在父脚本中，子脚本加载并在后台运行。通过将\$! shell 程序参数值赋给 pid 变量来记录子脚本的进程 ID，该参数值总是包含后台中最后一次运行的进程 ID。

父脚本继续运行，随后执行带有子进程 PID 的 wait 命令。这会导致父脚本暂停，直到子脚本退出；子脚本退出之后，父脚本也结束。

当执行的时候，父脚本和子脚本产生如下的输出。

---

```

[me@linuxbox ~]$ ./async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child: child is running...
Parent: pausing to wait for child to finish...
Child: child is done. Exiting.
Parent: child is finished. Continuing...
Parent: parent is done. Exiting.

```

---

## 36.4 命名管道

大多数类 UNIX 系统支持创建一个叫做命名管道的特殊类型的文件。使用命名管道可以建立两个进程之间的通信，并且可以像其他类型的文件一样使用。虽然它们没有其他类型的文件那么受欢迎，但是它们仍然值得了解。

客户端/服务器模式是一种常见的程序设计结构。它可以使用命名管道这样的通信方式，也可以使用网络连接这样的进程间通信方式。

使用客户端/服务器程序设计架构最为广泛的地方当然是 Web 服务器与 Web 浏览器之间的通信。Web 浏览器充当客户机，客户机对服务器提出请求，服务器通过网页的形式对浏览器做出回应。

命名管道的工作方式与文件雷同，但实际上是一块先进先出（FIFO）的缓冲区。与普通的（未命名的）管道一样，数据从一端进入，从另一端出来。使用命名管道，也可以以如下方式设置。

```
process1 > named_pipe
```

以及

```
process2 < named_pipe
```

执行效果等效于如下语句。

```
process1 | process2
```

### 36.4.1 设置命名管道

首先，我们必须创建一个命名管道。使用 mkfifo 命令即可完成，如下所示。

---

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me me 0 2012-07-17 06:41 pipe1
```

---

这里使用 mkfifo 命令创建一个名为 pipe1 的命名管道。使用 ls 命令查看文件属性，可以看到属性字段的第一个字母是 p，这表明它是一个命名管道。

### 36.4.2 使用命名管道

为了说明命名管道是如何工作的，我们需要两个终端窗口（或者两个虚拟的控制台）。在第一个终端中，我们输入一条简单的命令，并将其输出重新定位到这个命名管道，如下所示。

---

```
[me@linuxbox ~]$ ls -l > pipe1
```

---

按下 Enter 键之后，这个命令看起来像挂起来了。这是因为从管道的另一端还没有接收到数据。当这种情况发生时，也就是说命名管道被阻塞。一旦将一个进程连接到管道的另一端时，情况就会有所改变，进程会从管道中读取数据。使用第二种终端窗口，输入如下命令。

```
[me@linuxbox ~]$ cat < pipe1
```

---

第一个终端窗口产生的目录列表作为 cat 命令的输出出现在第二个终端窗口。一旦它不处于阻塞状态，第一个终端窗口中的 ls 命令就可以成功完成。

## 36.5 本章结尾语

到此我们已经结束了所有的学习内容。现在要做的唯一一件事就是练习，练习，再练习。尽管在学习的过程中已经涉及了大量的内容，但是，我们了解的仅仅是命令行方面的皮毛而已。仍然还有成千上万的命令行程序等着去发现和探索。深入地探索/usr/bin 目录内容之后，相信你一定会有所收获！