编　　制：北京万邦易嵌科技有限公司-嵌入式研发部肖龙

版　　本：V2.0

编制日期：2017 年 1 月 10 日

修改日期：2018 年 1 月 10 日

第一章 Shell 简介

Linux 命令在线查看网站：http://man.linuxde.net/

Shell 本身是一个用 C 语言编写的程序，它是用户使用 Unix/Linux 的桥梁，用户的大部分工作都是通过 Shell 完成的。Shell 既是一种命令语言，又是一种程序设计语言。作为命令语言，它交互式地解释和执行用户输入的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

它虽然不是 Unix/Linux 系统内核的一部分，但它调用了系统核心的大部分功能来执行程序、建立文件并以并行的方式协调各个程序的运行。因此，对于用户来说，shell 是最重要的实用程序，深入了解和熟练掌握 shell 的特性极其使用方法，是用好 Unix/Linux 系统的关键。

- Shell 有两种执行命令的方式：

交互式（Interactive）：解释执行用户的命令，用户输入一条命令，Shell 就解释执行一条。

批处理（Batch）：用户事先写一个 Shell 脚本(Script)，其中有很多条命令，让 Shell 一次把这些命令执行完，而不必一条一条地敲命令。

- Shell 环境

Shell 编程跟 java、php 编程一样，只要有一个能编写代码的文本编辑器和一个能解释执行的脚本解释器就可以了。

Linux 的 Shell 种类众多，常见的解释器有：

① Bourne Shell（/usr/bin/sh 或/bin/sh）

② Bourne Again Shell（/bin/bash）

③ C Shell（/usr/bin/csh）

④ K Shell（/usr/bin/ksh）

⑤ Shell for Root（/sbin/sh）

⑥ ……

本教程关注的是 Bash，也就是 Bourne Again Shell，由于易用和免费，Bash 在日常工作中被广泛使用。同时，Bash 也是大多数 Linux 系统默认的 Shell。

在一般情况下，人们并不区分 Bourne Shell 和 Bourne Again Shell，所以，像 #!/bin/sh，它同样也可以改为 #!/bin/bash。

#!告诉系统其后路径所指定的程序即是解释此脚本文件的 Shell 程序。

- 查看当前系统默认的 shell:

```
[xiao@localhost file_2]$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 05-25 19:24 /bin/sh -> bash
```

第二章 Shell 语法

2.1 第一个 Shell 程序

打开文本编辑器，新建一个文件，扩展名为 sh（sh 代表 shell），扩展名并不影响脚本执行，见名知意就好。

输入一些代码：

```
#!/bin/bash
echo "Hello World !"
```

“#!” 是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，使用哪一种 Shell。echo 命令用于向窗口输出文本。

脚本编写成功后，需要修改该脚本的权限为可执行，然后再执行。

示例：

```
chmod 777 test.sh    #修改权限为 777
./test.sh            #执行脚本
```

注意，一定要写成./test.sh，而不是 test.sh。运行其它二进制的程序也一样，直接写 test.sh，linux 系统会去 PATH 里寻找有没有叫 test.sh 的，而只有/bin, /sbin, /usr/bin，/usr/sbin 等在 PATH 里，你的当前目录通常不在 PATH 里，所以写成 test.sh 是会找不到命令的，要用./test.sh 告诉系统说，就在当前目录找。

系统没有指定脚本的情况下，通过这种方式运行 bash 脚本，第一行一定要写对，好让系统查找到正确的解释器。如果系统一开始就指定了默认的 shell 脚本，第一行指定解析器的代码就可以省掉。

当提示命令解析找不到的时候可以在运行时，加上解析器：

```
[xiao@localhost file]$ bash bash_shell.sh
```

- 在 shell 脚本中以“#”开头的行就是注释，会被解释器忽略。

sh 里没有多行注释，只能每一行加一个#号

如果在开发过程中，遇到大段的代码需要临时注释起来，过一会儿又取消注释，怎么办呢？每一行加个#符号太费力了，可以把这一段要注释的代码用一对花括号括起来，定义成一个函数，没有地方调用这个函数，这块代码就不会执行，达到了和注释一样的效果。

2.2 Shell 变量

Shell 支持自定义变量。

2.2.1 定义变量

定义变量时，变量名不加美元符号（$），如：ABC="123"

注意，变量名和等号之间不能有空格，这可能和你熟悉的所有编程语言都不一样。同时，变量名的命名须遵循如下规则：

① 首个字符必须为字母（a-z，A-Z）。

② 中间不能有空格，可以使用下划线（_）。

③ 不能使用标点符号。

④ 不能使用 bash 里的关键字（可用 help 命令查看保留关键字）。

- 变量定义举例

```
A="ABCD "
B=100
```

2.2.2 使用变量

使用一个定义过的变量，只要在变量名前面加美元符号（$）即可，如：

```
A ="123"
echo $A
echo ${A}
```

变量名外面的花括号是可选的，加不加都行，加花括号是为了帮助解释器识别变量的边界，

推荐给所有变量加上花括号，这是个好的编程习惯。

2.2.3 只读变量

使用 readonly 命令可以将变量定义为只读变量，只读变量的值不能被改变。

下面的例子尝试更改只读变量，结果报错：

```
#!/bin/bash
A="123"
readonly A   #声明只读变量
A="345"
```

运行脚本，结果如下：

./bash_shell.sh: line 3: A: readonly variable

2.2.4 删除变量

使用 unset 命令可以删除变量。语法：unset variable_name

变量被删除后不能再次使用；unset 命令不能删除只读变量。

举个例子：

```
#!/bin/sh
A="1234"
unset A
echo $A
```

上面的脚本没有任何输出。

2.2.5 Shell 特殊变量

前面已经讲到，变量名只能包含数字、字母和下划线，因为某些包含其他字符的变量有特殊含义，这样的变量被称为特殊变量。

- 特殊变量列表

$0	当前脚本的文件名
$n	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是$1，第二个参数是$2。
$#	传递给脚本或函数的参数个数。
$*	传递给脚本或函数的所有参数。
$@	传递给脚本或函数的所有参数。
$?	上个命令的退出状态，或函数的返回值。$? 也可以表示函数的返回值。 比如: ls -l echo $? 输出上一次命令的状态 或者直接获取: **data=`ls -l`**
$$	当前 Shell 进程 ID。对于 Shell 脚本，就是这些脚本所在的进程 ID。

- 命令行参数

运行脚本时传递给脚本的参数称为命令行参数。命令行参数用 $n 表示，例如，$1 表示第一个参数，$2 表示第二个参数，依次类推。

- 退出状态

$? 可以获取上一个命令的退出状态。所谓退出状态，就是上一个命令执行后的返回结果。

退出状态是一个数字，一般情况下，大部分命令执行成功会返回 0，失败返回 1。不过，也有一些命令返回其他值，表示不同类型的错误。

- 程序示例

```
#!/bin/bash
echo "脚本文件名称：$0"
echo "第一个参数：$1"
echo "第二个参数：$2"
echo "传进来参数总数量：$# 个"
echo "传进来的所有参数：$*"
echo "传进来的所有参数：$@"
echo "命令执行的状态：$?"
echo "当前脚本的进程 ID 号：$$"
```

- 运行: # ./bash_shell.sh 12 13 14

2.3 Shell 运算符

Bash 支持很多运算符，包括算数运算符、关系运算符、布尔运算符、字符串运算符和文件测试运算符。原生 bash 不支持简单的数学运算，但是可以通过其他命令来实现，例如 awk 和 expr，expr 最常用。expr 是一款表达式计算工具，使用它能完成表达式的求值操作。

例如，两个数相加：

```
#!/bin/bash
A=`expr 2 + 2`
echo "sum = $A"
```

运行脚本输出：SUM= 4

注意：表达式和运算符之间要有空格，例如 2+2 是不对的，必须写成 2 + 2，这与我们熟悉的大多数编程语言不一样。完整的表达式要被 `` 包含，注意这个字符不是常用的单引号，在 Esc 键下边。

2.3.1 算术运算符

- 算术运算符示例代码：

```
#!/bin/sh
a=10
b=20
val=`expr $a + $b`       #加法运算
echo "a + b : $val"

val=`expr $a - $b`     #减法运算
echo "a - b : $val"

val=`expr $a \* $b`    #乘法运算
echo "a * b : $val"

val=`expr $b / $a`     #除法运算
echo "b / a : $val"

val=`expr $b % $a`    #取余运算
echo "b % a : $val"
if [ $a == $b ]        #判断变量 a 和 b 是否相等
then
    echo "a is equal to b"    #相等
fi

if [ $a != $b ]       #判断变量 a 和 b 是否相等
then
    echo "a is not equal to b"    #不相等
fi
```

运行结果：

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

注意：乘号(*)前边必须加反斜杠(\)才能实现乘法运算；其中的 if...then...fi 是条件语句，后续将会讲解。

- 算术运算符列表

+	加法 `expr $a + $b` 结果为 30。
-	减法 `expr $a - $b` 结果为 10。
*	乘法 `expr $a \* $b` 结果为 200。
/	除法 `expr $b / $a` 结果为 2。

%	取余 `expr $b % $a` 结果为 0。
=	赋值 a=$b 将把变量 b 的值赋给 a。
==	相等。 用于比较两个数字，相同则返回 true。 [$a == $b] 返回 false。
!=	不相等。用于比较两个数字，不相同则返回 true。 [$a != $b] 返回 true。

注意：条件表达式要放在方括号之间，并且要有空格，例如 [$a==$b] 是错误的，必须写成 [$a == $b]。

2.3.2 关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

- 关系运算符示例代码

```
#!/bin/sh
a=10
b=20
if [ $a -eq $b ]
then
    echo "$a -eq $b : a is equal to b"
else
    echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
    echo "$a -ne $b: a is not equal to b"
else
    echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
    echo "$a -gt $b: a is greater than b"
else
    echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
    echo "$a -lt $b: a is less than b"
else
    echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
    echo "$a -ge $b: a is greater or    equal to b"
else
    echo "$a -ge $b: a is not greater or equal to b"
fi
```

```
if [ $a -le $b ]
then
    echo "$a -le $b: a is less or    equal to b"
else
    echo "$a -le $b: a is not less or equal to b"
fi
```

运行结果

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or    equal to b
```

- 关系运算符列表

-eq	检测两个数是否相等，相等返回 true。（正确--1） [$a -eq $b] 返回 true。
-ne	检测两个数是否相等，不相等返回 true。(1) [$a -ne $b] 返回 true。
-gt	检测左边的数是否大于右边的，如果是，则返回 true。 [$a -gt $b] 返回 false。(错误 0)
-lt	检测左边的数是否小于右边的，如果是，则返回 true。 [$a -lt $b] 返回 true。
-ge	检测左边的数是否大等于右边的，如果是，则返回 true。 [$a -ge $b] 返回 false。
-le	检测左边的数是否小于等于右边的，如果是，则返回 true。 [$a -le $b] 返回 true。

2.3.3 布尔运算符

- 布尔运算符使用示例

```
#!/bin/sh
a=10
b=20

if [ $a != $b ]
then
    echo "$a != $b : a is not equal to b"
else
    echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
    echo "$a -lt 100 -a $b -gt 15 : returns true"
else
    echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
```

```
    echo "$a -lt 100 -o $b -gt 100 : returns true"
else
    echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
    echo "$a -lt 100 -o $b -gt 100 : returns true"
else
    echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

运行结果

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

- 布尔运算符列表

-o	或运算，有一个表达式为 true 则返回 true。 [$a -lt 20 -o $b -gt 100] 返回 true。
-a	与运算，两个表达式都为 true 才返回 true。 [$a -lt 20 -a $b -gt 100] 返回 false。

2.3.4 字符串运算符

- 字符串运算符示例代码

```
#!/bin/sh
a="abc"
b="efg"

if [ $a = $b ]
then
    echo "$a = $b : a is equal to b"
else
    echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
    echo "$a != $b : a is not equal to b"
else
    echo "$a != $b: a is equal to b"
fi

if [ -z $a ]
then
    echo "-z $a : string length is zero"
else
    echo "-z $a : string length is not zero"
```

```
fi

if [ -n $a ]
then
    echo "-n $a : string length is not zero"
else
    echo "-n $a : string length is zero"
fi

if [ $a ]
then
    echo "$a : string is not empty"
else
    echo "$a : string is empty"
fi
```

运行结果：

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

- 字符串运算符列表

=	检测两个字符串是否相等，相等返回 true。 [$a = $b] 返回 false。
!=	检测两个字符串是否相等，不相等返回 true。[$a != $b] 返回 true。
-z	检测字符串长度是否为 0，为 0 返回 true。 [-z $a] 返回 false。
-n	检测字符串长度是否为 0，不为 0 返回 true。[-z $a] 返回 true。
str	检测字符串是否为空，不为空返回 true。 [$a] 返回 true。

注意：字符串比较必须时，必须加双引号。

例如: 变量 A 与”*”之间比较是否相等。 必须这样写： if [“$A” == “*”]

2.3.5 文件测试运算符

文件测试运算符用于检测 Unix 文件的各种属性。

例如，变量 file 表示文件“test.sh”，它的大小为 100 字节，具有 rwx 权限。下面的代码，将检测该文件的各种属性：

```
#!/bin/sh

file="/test.sh"

if [ -r $file ]
then
    echo "File has read access"
else
    echo "File does not have read access"
```

```
fi

if [ -w $file ]
then
    echo "File has write permission"
else
    echo "File does not have write permission"
fi

if [ -x $file ]
then
    echo "File has execute permission"
else
    echo "File does not have execute permission"
fi

if [ -f $file ]
then
    echo "File is an ordinary file"
else
    echo "This is sepcial file"
fi

if [ -d $file ]
then
    echo "File is a directory"
else
    echo "This is not a directory"
fi

if [ -s $file ]
then
    echo "File size is zero"
else
    echo "File size is not zero"
fi

if [ -e $file ]
then
    echo "File exists"
else
    echo "File does not exist"
fi
```

运行结果：

```
File has read access
File has write permission
```

```
File has execute permission
File is an ordinary file
This is not a directory
File size is zero
File exists
```

- 文件测试运算符列表

-b file	检测文件是否是块设备文件，如果是，则返回 true。 [-b $file] 返回 false。
-c file	检测文件是否是字符设备文件，如果是，则返回 true。 [-b $file] 返回 false。
-d file	检测文件是否是目录，如果是，则返回 true。 [-d $file] 返回 false。
-f file	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。 [-f $file] 返回 true。
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true。 [-g $file] 返回 false。
-k file	检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。[-k $file] 返回 false。
-p file	检测文件是否是具名管道，如果是，则返回 true。 [-p $file] 返回 false。
-u file	检测文件是否设置了 SUID 位，如果是，则返回 true。 [-u $file] 返回 false。
-r file	检测文件是否可读，如果是，则返回 true。 [-r $file] 返回 true。
-w file	检测文件是否可写，如果是，则返回 true。 [-w $file] 返回 true。
-x file	检测文件是否可执行，如果是，则返回 true。 [-x $file] 返回 true。
-s file	检测文件是否为空（文件大小是否大于 0），不为空返回 true。[-s $file] 返回 true。
-e file	检测文件（包括目录）是否存在，如果是，则返回 true。 [-e $file] 返回 true。

2.4 Shell 字符串

字符串是 shell 编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。

2.4.1 单引号

```
str='this is a string'
```

- 单引号字符串的限制：

单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；

单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

2.4.2 双引号

```
your_name='大侠'
str="Hello, I know your are \"$your_name\"! \n"
```

双引号的优点：

双引号里可以有变量

双引号里可以出现转义字符

2.4.3 拼接字符串

```
your_name="大侠"
greeting="hello, "$your_name" !"
greeting_1="hello, ${your_name} !"
echo $greeting $greeting_1
```

输出结果：

hello, 大侠 ！hello, 大侠 ！

2.4.4 获取字符串长度

```
string="abcd"
echo ${#string}     #输出 4
```

2.4.5 提取子字符串

```
string="123456789"
echo ${string:1:4} #输出 2345
```

2.4.6 查找子字符串

```
string="123ABC567 "
echo `expr index "$string" ABC`   #输出 1
```

ABC 是要查找的字符串，$string：是存放字符串的变量

2.5 Shell 数组

Shell 在编程方面比 Windows 批处理强大很多，无论是在循环、运算。

bash 支持一维数组（不支持多维数组），并且没有限定数组的大小。类似与 C 语言，数组元素的下标由 0 开始编号。获取数组中的元素要利用下标，下标可以是整数或算术表达式，其值应大于或等于 0。

2.5.1 定义数组

在 Shell 中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的一般形式为：

```
array_name=(value1 ... valuen)
```

例如：

```
array_name=(value0 value1 value2 value3)
```

或者

```
array_name=(
value0
value1
value2
value3
)
```

还可以单独定义数组的各个分量：

```
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2
```

可以不使用连续的下标，而且下标的范围没有限制。

2.5.2 读取数组

读取数组元素值的一般格式是：

```
${array_name[index]}
```

例如：

```
valuen=${array_name[2]}
```

- 举个例子：

```
#!/bin/sh
NAME[0]="12"
NAME[1]="13"
NAME[2]="14"
NAME[3]="15"
NAME[4]="16"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

运行脚本，输出：

```
$./test.sh
First Index: 12
Second Index: 13
```

- 使用@ 或 * 可以获取数组中的所有元素，例如：

```
${array_name[*]}
${array_name[@]}
```

- 数组示例代码：

```
#!/bin/sh
BUFF[0]="1"
BUFF[1]="2"
BUFF[2]="3"
BUFF[3]="4"
BUFF[4]="5"
echo "One: ${BUFF[*]}"
echo "Two: ${BUFF[@]}"
```

运行脚本输出：

```
[root@localhost file_1]# ./bash_shell.sh
One: 1 2 3 4 5
Two: 1 2 3 4 5
```

2.5.3 获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同。

① 取得数组元素的个数，方式 1

```
length=${#array_name[@]}
```

② 取得数组元素的个数，方式 1

```
length=${#array_name[*]}
```

③ 取得数组单个元素的长度

```
lengthn=${#array_name[n]}
```

- 举例:

```
array_name=(value0 value1 value2 value3)
length=${#array_name[*]}  #得到数组长度
lengthn=${#array_name[0]} #得到单个下标存放的字符长度

echo ${array_name[*]}
echo $length
echo $lengthn
```

输出结果：

```
value0 value1 value2 value3
4
6
```

2.6 Shell echo 命令

echo 是 Shell 的一个内部指令，用于在屏幕上打印出指定的字符串。

命令格式：echo arg

2.6.1 显示转义字符

```
echo "\"It is a test\""
```

结果将是："It is a test"

2.6.2 显示变量

```
name="OK"
echo "$name It is a test"
```

结果将是：OK It is a test　　说明：变量赋值时双引号也可以省略。

- 如果变量与其它字符相连的话，需要使用大括号（{}）：

```
mouth=8
echo "${mouth}-1-2009"
```

结果将是：8-1-2009

2.6.3 显示换行

```
echo "OK!\n"
echo "It is a test"
```

输出：

OK!

It is a test

2.6.4 显示不换行

```
echo "OK!\c"
echo "It is a test"
```

输出：OK!It si a test

2.6.5 显示结果重定向至文件

```
echo "It is a test" > myfile
```

如果 myfile 这个文件没有，会自动创建，每次写入会覆盖之前的内容。

如果是>>符号，就表示追加。

2.7 Shell printf 命令

printf 命令用于格式化输出，是 echo 命令的增强版。它是 C 语言 printf()库函数的一个有限的变形，并且在语法上有些不同。

注意：printf 由 POSIX 标准所定义，移植性要比 echo 好。

如同 echo 命令，printf 命令也可以输出简单的字符串：

```
$printf "Hello, Shell\n"
Hello, Shell
```

printf 不像 echo 那样会自动换行，必须显式添加换行符(\n)。

- printf 命令的语法：

```
printf  format-string  [arguments...]
```

format-string 为格式控制字符串，arguments 为参数列表。

示例：printf "%s\n" "1234"

printf()在 C 语言中已经讲到，Shell 中的 print 功能和用法与 C 语言类似。这里仅说明与 C 语言 printf()函数的不同：

① printf 命令不用加括号

② format-string 可以没有引号，但最好加上，单引号双引号均可。

③ 参数多于格式控制符(%)时，format-string 可以重用，可以将所有参数都转换。

④ arguments 使用空格分隔，不用逗号。

- 示例代码

(1) format-string 为双引号

```
printf "%d %s\n" 1 "abc"
```

输出: 1 abc

(2) 单引号与双引号效果一样

```
printf '%d %s\n' 1 "abc"
```

输出: 1 abc

(3) 没有引号也可以输出

```
printf %s abcdef
```

输出: abcdef

(4) 格式只指定了一个参数，但多出的参数仍然会按照该格式输出，format-string 被重用

```
printf %s abc def
```

输出: abcdef

```
printf "%s\n" abc def
```

输出:

abc
def

```
printf "%s %s %s\n" a b c d e f g h i j
```

输出:

a b c
d e f
g h i
j

(5) 如果没有 arguments，那么 %s 用 NULL 代替，%d 用 0 代替

```
printf "%s and %d \n"
```

输出:

and 0

(6) 如果以 %d 的格式来显示字符串，那么会有警告，提示无效的数字，此时默认置为 0

```
$ printf "The first program always prints'%s,%d\n'" Hello Shell
```

输出:

-bash: printf: Shell: invalid number
The first program always prints 'Hello,0'

(7) 打印浮点数

```
printf "%f %s\n" 1.1234 "abc"
```

输出：1.123400 abc

```
printf "%0.4f %s\n" 1.1234 "abc"
```

输出：1.1234 abc

注意，根据 POSIX 标准，浮点格式%e、%E、%f、%g 与%G 是“不需要被支持”。这是因为 awk 支持浮点预算，且有它自己的 printf 语句。这样 Shell 程序中需要将浮点数值进行格式化的打印时，可使用小型的 awk 程序实现。然而，内建于 bash、ksh93 和 zsh 中的 printf 命令都支持浮点格式。

其中: awk 是一种编程语言，用于在 linux/unix 下对文本和数据进行处理

2.8 Shell read 命令

read 命令接收标准输入（键盘）的输入，或其他文件描述符的输入（后面在说）。得到输入后，read 命令将数据放入一个标准变量中。

2.8.1 read 命令基本用法

- 下面是 read 命令用法：

```
#!/bin/bash
echo -n "输入数据:"                 #参数-n 的作用是不换行，echo 默认是换行
read   data                         #从键盘输入
echo "你输入的数据为: $data"        #显示信息
exit 0                              #退出 shell 程序。
```

read 命令提供了-p 参数，允许在 read 命令行中直接指定一个提示。

- 所以上面的脚本可以简写成下面的脚本：

```
#!/bin/bash
read   -p "输入数据:" data          #从键盘输入
echo "你输入的数据为: $data"        #显示信息
exit 0                              #退出 shell 程序。
```

在上面 read 后面的变量只有一个，也可以有多个，这时如果输入多个数据，则第一个数据给第一个变量，第二个数据给第二个变量，如果输入数据个数过多，则最后所有的值都给第一个变量。如果太少输入不会结束。

- 示例

```
#!/bin/bash
read   -p "输入数据:" data1 data2 data3
echo "你输入的数据为: $data1 $data2 $data3"
exit 0
```

输入数据时，多个数据之间使用空格隔开。

2.8.2 环境变量 REPLY

在 read 命令行中也可以不指定变量.如果不指定变量，那么 read 命令会将接收到的数据放置在环境变量 REPLY 中。

例如：read -p "输入一个数字 "

环境变量 REPLY 中包含输入的所有数据，可以像使用其他变量一样在 shell 脚本中使用环境变量 REPLY.

示例：

```
read -p "输入一个数字:"
echo $REPLY
```

2.8.3 计时输入

使用 read 命令存在着潜在危险。脚本很可能会停下来一直等待用户的输入。如果无论是否输入数据脚本都必须继续执行，那么可以使用-t 选项指定一个计时器。

-t 选项指定 read 命令等待输入的秒数。当计时满时，read 命令返回一个非零退出状态;

示例：

```
#!/bin/bash
if read -t 5 -p "请输入你的名字:" name
then
        echo "你好  $name,欢迎来到我的脚本"
else
        echo "对不起,你输入的太慢了"
fi
exit 0
```

如果 5 秒内没有输入，脚本自动退出。

2.8.4 输入计数

除了输入时间计时，还可以设置 read 命令计数输入的字符。当输入的字符数目达到预定数目时，自动退出，并将输入的数据赋值给变量。

示例：

```
#!/bin/bash
read -n1 -p "是否继续  [Y/N]?" data
case $data in
Y | y)
         echo "很好,继续";;
N | n)
         echo "好吧,再见";;
*)
        echo "错误的选择";;
esac
exit 0
```

该例子使用了-n 选项，后接数值 1，指示 read 命令只要接受到一个字符就退出。只要按下一个字符进行回答，read 命令立即接受输入并将其传给变量。无需按回车键。case 是 shell 中的多分枝判断语句，类似 C 语言的 switch，后面会讲到。

2.8.5 默读（输入不显示在监视器上）

有时会需要脚本用户输入，但不希望输入的数据显示在监视器上。典型的例子就是输入密码，当然还有很多其他需要隐藏的数据。

-s 选项能够使 read 命令中输入的数据不显示在监视器上（实际上，数据是显示的，只是 read 命令将文本颜色设置成与背景相同的颜色）。

示例：

```
#!/bin/bash
read    -s    -p "输入你的密码:" pass
echo "你的密码是: $pass"
exit 0
```

2.8.6 读文件

最后，还可以使用 read 命令读取 Linux 系统上的文件。

每次调用 read 命令都会读取文件中的"一行"文本。当文件没有可读的行时，read 命令将以非零状态退出。

读取文件的关键是如何将文本中的数据传送给 read 命令。

最常用的方法是对文件使用 cat 命令并通过管道将结果直接传送给包含 read 命令的 while 命令。

示例：

```
#!/bin/bash
count=1
cat 123.c | while read line          #cat 命令的输出作为 read 命令的输入,read 读到的值放在 line 中
do
    echo "Line $count:$line"
    count=$[ $count + 1 ]            #注意中括号中的空格。
done
echo "finish"
exit 0
```

其中 123.c 是读取的文件。While 是 shell 中的循环语句，后面会讲到。

第三章 语句与函数

3.1 Shell if else 语句

if 语句通过关系运算符判断表达式的真假来决定执行哪个分支。

Shell 有三种 if ... else 语句：

① if ... fi
② if ... else ... fi
③ if ... elif ... else ... fi

3.1.1 if ... else 语句

if ... else 语句的语法：

```
if [ 表达式 ]
then
    <语句块>
fi
```

如果<表达式>返回 true，then 后边的语句将会被执行；如果返回 false，不会执行任何语句。最后必须以 fi 来结尾闭合 if，fi 就是 if 倒过来拼写。

注意：<表达式> 和方括号([ ])之间必须有空格，否则会有语法错误。

- 注意 if 多个表达式之间的判断:

```
#!/bin/bash
data1=123
data2=456
data3=888
```

```
if [ ${data1} == 123 -a ${data2} == 456 ] || [ ${data3} == 888 ]
then
echo "为真"
else
echo "为假"
fi


if [ ${data1} == 123 -a ${data2} == 456 ] && [ ${data3} == 888 ]
then
echo "为真"
else
echo "为假"
fi

#说明: if 语句里两个独立的表达式可以使用|| 或者 && 进行区分
```

- 举例：

```
#!/bin/bash
a=10
b=20

if [ $a == $b ]
then
    echo "a 等于 b"
fi

if [ $a != $b ]
then
    echo "a 不等于 b"
fi
```

运行结果：a 不等于 b

3.1.2 if ... else ... fi 语句

if ... else ... fi 语句的语法：

```
if [ <表达式> ]
then
    语句块 1
else
    语句块 2
fi
```

如果<表达式> 返回 true，那么 then 后边的语句将会被执行；否则，执行 else 后边的语句。

- 举例：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
    echo "a 等于 b"
else
    echo "a 不等于 b"
fi
```

执行结果：a 不等于 b

3.1.3 if ... elif ... fi 语句

if ... elif ... fi 语句可以对多个条件进行判断，语法为：

```
if [ 表达式 1 ]
then
    语句块 1
elif [ 表达式 2 ]
then
    语句块 2
elif [ 表达式 3 ]
then
    语句块 3
else
    语句块 4
fi
```

哪一个 <表达式> 的值为 true，就执行哪个<表达式> 后面的语句；如果所有条件都为 false，那么执行语句块 4。

- 举例：

```
#!/bin/sh
a=10
b=20

if [ $a == $b ]
then
    echo "a = b"
elif [ $a -gt $b ]
then
    echo "a 大于 b"
elif [ $a -lt $b ]
then
    echo "a 小于 b"
else
    echo "没有一个条件满足"
```

```
fi
```

运行结果：a 小于 b

- if ... else 语句也可以写成一行，以命令的方式来运行，像这样：

```
if test $[2*3] -eq $[1+5]; then echo '这两个数字是相等的!'; fi;
```

- if ... else 语句也经常与 test 命令结合使用，如下所示：

```
num1=$[2*3]
num2=$[1+5]
if test $[num1] -eq $[num2]
then
    echo '这两个数字是相等的!'
else
    echo '这两个数字是不相等的!'
fi
```

输出：这两个数字是相等的!

其中的 test 命令用于检查某个条件是否成立，与方括号([ ])类似。

3.2 Shell case esac 语句

case ... esac 与其他语言中的 switch ... case 语句类似，是一种多分枝选择结构。

case 语句匹配一个值或一个模式，如果匹配成功，执行相匹配的命令。

3.2.1 case 语句格式

```
case <值> in
模式 1)
    command1
    command2
    command3
    ;;
模式 2）
    command1
    command2
    command3
    ;;
*)
    command1
    command2
    command3
    ;;
esac
```

case 工作方式如上所示。取值后面必须为关键字 in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。;; 与其他语言中的 break 类似，意思是跳到整个 case 语句的最后。

取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令。

3.2.2 示例 1

下面的脚本提示输入 1 到 4，与每一种模式进行匹配：

```
echo '输入一个 1 到 4 之间的数字'
read aNum
case $aNum in
    1)  echo '你选择 1'
    ;;
    2)  echo '你选择 2'
    ;;
    3)  echo '你选择 3'
    ;;
    4)  echo '你选择 4'
    ;;
    *)  echo '你不选择一个数字不在 1 到 4 之间'
    ;;
esac
```

```
1  echo ’输入一个1到4之间的数字’
2  read aNum
3  case $aNum in
4      1)  echo ’你选择1’
5      ;;
6      2)  echo ’你选择2’
7      ;;
8      3)  echo ’你选择3’
9      ;;
10     4)  echo ’你选择4’
11     ;;
12     *)  echo ’你不选择一个数字不在1到4之间’
13     ;;
14 esac
```

3.2.3 示例 2

```
#!/bin/bash
option="${1}"
case ${option} in
   -f)
      FILE="${2}"
      echo "输入的文件名称为: $FILE"
      ;;
   -d)
      DIR="${2}"
      echo "输入的目录名称为: $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      ;;
esac
```

运行结果：

```
[root@xiaolong linux-share-dir]# ./123.sh -f 123.c
输入的文件名称为: 123.c
[root@xiaolong linux-share-dir]# ./123.sh -d /work/
输入的目录名称为: /work/
```

3.3 Shell for 循环

与其他编程语言类似，Shell 支持 for 循环。

3.3.1 for 循环一般格式

```
for <变量> in <列表>
do
    command1
    command2
    ...
    commandN
done
```

列表是一组值（数字、字符串等）组成的序列，每个值通过空格分隔。每循环一次，就将列表中的下一个值赋给变量。in <列表>是可选的，如果不用它，for 循环使用命令行的位置参数。

3.3.2 顺序输出当前列表中的数字

```
for loop in 1 2 3 4 5
do
    echo "值: $loop"
done
```

运行结果：

```
[root@xiaolong linux-share-dir]# ./123.sh
值: 1
值: 2
值: 3
值: 4
值: 5
```

3.3.3 顺序输出字符串中的字符

```
for str in 'This is a string'
do
    echo $str
done
```

运行结果：

```
[root@xiaolong linux-share-dir]# ./123.sh
This is a string
```

3.3.4 循环遍历文件

```
#!/bin/bash
for FILE in /test/*.c
do
```

```
    echo $FILE
done
```

显示/test/目录下以.c 结尾的文件。

3.4 Shell while 循环

while 循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。

3.4.1 while 循环格式

```
while <command>
do
    <语句块>
done
```

命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假。

3.4.2 基本的 while 循环运用

以下是一个基本的 while 循环，测试条件是：如果 COUNTER 小于 5，那么返回 true。COUNTER 从 0 开始，每次循环处理时，COUNTER 加 1。运行上述脚本，返回数字 1 到 5，然后终止。

```
COUNTER=0
while [ $COUNTER -lt 5 ]
do
    COUNTER=`expr $COUNTER + 1`
    echo $COUNTER
done
```

运行结果：

```
[root@xiaolong linux-share-dir]# ./123.sh
1
2
3
4
5
```

3.4.3 循环读取键盘信息

while 循环可用于读取键盘信息。下面的例子中，输入信息被设置为变量 DATA，按<Ctrl-D>、<CTRL-C>结束循环。

```
echo '输入<CTRL-D> 或者<CTRL-C>结束程序 '
echo -n '输入你最喜欢的数字: '
while read DATA
do
    echo "吉祥数字: $DATA"
done
```

运行结果：

```
[root@xiaolong linux-share-dir]# ./123.sh
输入<CTRL-D> 或者<CTRL-C>结束程序
输入你最喜欢的数字： 8888
吉祥数字： 8888
66666
吉祥数字： 66666
```

3.5 Shell break 和 continue 命令

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，像大多数编程语言一样，Shell 也使用 break 和 continue 来跳出循环。

3.5.1 break 命令

break 命令允许跳出所有循环（终止执行后面的所有循环）。

下面的例子中，脚本进入死循环直至用户输入数字大于 5。要跳出这个循环，返回到 shell 提示符下，就要使用 break 命令。

```
#!/bin/bash
while :
do
    echo -n "输入一个 1 到 5 之间的数字:"
    read aNum
    case $aNum in
        1|2|3|4|5)
        echo "你输入的数字是 $aNum!"
        ;;
        *)
        echo "你不选择一个 1 到 5 之间的数字,程序就结束了!"
            break
        ;;
    esac
done
echo "程序结束
```

在嵌套循环中，break 命令后面还可以跟一个整数，表示跳出第几层循环。

例如：

break n 表示跳出第 n 层循环。

3.5.2 break 跳出嵌套循环

下面是一个嵌套循环的例子，如果 var1 等于 2，并且 var2 等于 0，就跳出循环：

```
#!/bin/bash
for var1 in 1 2 3
do
    for var2 in 0 5
    do
        if [ $var1 -eq 2 -a $var2 -eq 0 ]
        then
            break 2
```

```
        else
            echo "$var1 $var2"
        fi
    done
done
```

如上，break 2 表示直接跳出外层循环。

运行结果：
1 0
1 5

3.5.3 continue 命令

continue 命令与 break 命令类似，只有一点差别，它不会跳出所有循环，仅仅跳出当前循环。

对上面的例子进行修改：

运行代码发现，当输入大于 5 的数字时，该例中的循环不会结束。

```
#!/bin/bash
while :
do
    echo -n "输入一个 1 到 5 之间的数字: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
        *) echo "你输入的数字不在 1 到 5 之间!"
            continue
            echo "永远不会执行!"
        ;;
    esac
done
```

同样，continue 后面也可以跟一个数字，表示跳出第几层循环。

3.6 Shell 函数

函数可以让我们将一个复杂功能划分成若干模块，让程序结构更加清晰，代码重复利用率更高。像其他编程语言一样，Shell 也支持函数。Shell 函数必须先定义后使用。

3.6.1 Shell 函数的定义格式

```
function_name () {
    <语句块>
    [ return value ]
}
```

也可以在函数名前加上关键字 function：

```
function function_name () {
    <语句块>
```

```
    [ return value ]
}
```

3.6.2 函数返回值

函数返回值，可以显式增加 return 语句；如果不加，会将最后一条命令运行结果作为返回值。

Shell 函数返回值只能是整数，一般用来表示函数执行成功与否，0 表示成功，其他值表示失败。如果 return 其他数据，比如一个字符串，往往会得到错误提示：“numeric argument required”。

如果一定要让函数返回字符串，那么可以先定义一个变量，用来接收函数的计算结果，脚本在需要的时候访问这个变量来获得函数返回值。

3.6.3 函数的定义与调用

```
#!/bin/bash
# 定义函数
Hello () {
    echo "函数调用成功!"
}
Hello   #调用函数
```

运行结果：

```
[root@xiaolong linux-share-dir]# ./123.sh
函数调用成功!
```

调用函数只需要给出函数名，不需要加括号。

3.6.4 接收函数返回值

```
#!/bin/bash
#定义函数
Hello () {
    echo "函数调用成功!"
    return 88
}
Hello
echo $?      #接收函数返回值
```

函数返回值在调用该函数后通过 $? 来获得。

注意：函数返回值最大不能超过 255，且不能返回字符串。

这里说的字符串，就是包含了除了数字之外的数据。”0”~”255”这样的字符串是可以返回的

3.6.5 函数嵌套例子

```
#!/bin/bash

# 调用一个函数
number_one() {
    echo "123456789"
    number_two
}
```

```
number_two() {
    echo "abcdefg"
}

number_one   #调用函数
```

像删除变量一样，删除函数也可以使用 unset 命令，不过要加上 .f 选项，如下所示：

```
unset .f function_name
```

如果希望直接从终端调用函数，可以将函数定义在主目录下的 .profile 文件，这样每次登录后，在命令提示符后面输入函数名字就可以立即调用。

3.6.6 Shell 函数参数

在 Shell 中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值。

例如，$1 表示第一个参数，$2 表示第二个参数...

- 带参数的函数示例：

```
#!/bin/bash
fun(){
    echo "第 1 个参数的值  $1 "
    echo "第 2 个参数的值  $2 "
    echo "第 3 个参数的值  $3 "
    echo "参数个数  $# "
    echo "传递给函数的所有参数: $*"
}
fun 1 2 3 4 5
```

注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当 n>=10 时，需要使用${n}来获取参数。

另外，还有几个特殊变量用来处理参数，前面已经提到：

$#	传递给函数的参数个数。
$*	显示所有传递给函数的参数。
$@	与$*相同，但是略有区别，请查看 Shell 特殊变量。
$?	函数的返回值。

3.7 Shell 调用外部脚本

像其他语言一样，Shell 也可以包含外部脚本，将外部脚本的内容合并到当前脚本。

Shell 中包含脚本可以使用：

```
. filename
```

或

```
source filename
```

两种方式的效果相同，简单起见，一般使用点号(.)，但是注意点号(.)和文件名中间有一空格。

注意：被包含的脚本不需要有执行权限。

- 示例：

```
. 456.sh
source 456.sh
```

第四章 Shell 程序案例

4.1 变量定义与使用案例

4.1.1 变量基本定义

```
#!/bin/bash
data1=12345
data2="67890"
echo "data1 变量的值=${data1}" #取出变量的值
echo "data2 变量的值=${data2}"
data2+="abcdf"
echo "data2 变量的值=${data2}"
```

运行结果:

```
[root@wbyq linux-share-dir]# ./shell.sh
data1变量的值=12345
data2变量的值=67890
data2变量的值=67890abcdf
```

4.2 if 语句使用案例

4.2.1 C 语言风格 if 语句

```
#!/bin/bash
if((123==456 || 888==888))
then
    echo "为真"
else
    echo "为假"
fi

#!/bin/bash
if((123==456 && 888==888))
then
    echo "为真"
else
    echo "为假"
fi
```

输出结果:

```
[root@wbyq linux-share-dir]# ./shell.sh
为真
为假
```

4.2.2 if 语句多个表达式判断

```
#!/bin/bash
data1=123
data2=456
data3=888
if [ ${data1} == 123 -a ${data2} == 456 ] || [ ${data3} == 888 ]
then
echo "为真"
else
echo "为假"
fi

if [ ${data1} == 123 -a ${data2} == 456 ] && [ ${data3} == 888 ]
then
echo "为真"
else
echo "为假"
fi
#说明: if 语句里两个独立的表达式可以使用|| 或者 && 进行区分
```

4.2.3 if 语句基本用法

```
#!/bin/bash

#示例 1: 判断两个整数是否相等
a=123
b=123
if [ ${a} == ${b} ] #判断变量 a 和 b 是否相等
then
echo "相等" #相等
else
echo "不相等" #相等
fi

#示例 2: 判断字符串是否相等
str1="123"
str2="123"
if [ ${str1} == ${str2} ]
then
echo "str 相等" #相等
else
echo "str 不相等" #相等
fi
```

4.3 数组使用案例

4.3.1 数组基本定义

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

```
BUFF=(123 456 789 100)      #定义一个数组
echo "BUFF[0]=${BUFF[0]}" #输出下标 0 的值
echo "BUFF[1]=${BUFF[1]}" #输出下标 1 的值
echo "BUFF[2]=${BUFF[2]}" #输出下标 2 的值
echo "BUFF[*]=${BUFF[*]}" #输出全部数据



DATA[0]="1232"
DATA[1]="6767"
DATA[3]="yhhj"
echo "DATA[0]=${DATA[0]}" #输出下标 0 的值
echo "DATA[1]=${DATA[1]}" #输出下标 1 的值
echo "DATA[2]=${DATA[2]}" #输出下标 2 的值
echo "DATA[*]=${DATA[*]}" #输出全部数据
```

4.3.2 使用 if 语句判断数组元素

```
#!/bin/bash
BUFF=(123 456 789 123)
if [ ${BUFF[0]} == ${BUFF[3]} ]
then
    echo "BUFF[0]与{BUFF[3]相等:${BUFF[0]},${BUFF[3]}"
else
    echo "BUFF[0]与{BUFF[3]不相等:${BUFF[0]},${BUFF[3]}"
fi

if [ ${BUFF[1]} == ${BUFF[2]} ]
then
    echo "BUFF[1]与{BUFF[2]相等  :${BUFF[1]},${BUFF[2]}"
else
    echo "BUFF[1]与{BUFF[2]不相等  :${BUFF[1]},${BUFF[2]}"
fi
```

4.4 read 语句使用案例

4.4.1 read 基本用法

```
#!/bin/bash
echo -n "输入数据 1:"  #参数-n 的作用是不换行， echo 默认是换行
read data1    #从键盘输入值存放到 data 中
echo "你输入的数据为: $data1" #显示信息

echo -n "输入数据 2:"  #参数-n 的作用是不换行， echo 默认是换行
read data2    #从键盘输入值存放到 data 中
echo "你输入的数据为: $data2" #显示信息
```

4.5 算术运算使用案例

4.5.1 expr 连续运算

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

```
data=`expr 1 + 1 + 1 + 1`
echo "data=$data"

data=`expr 4 - 1 + 1 + 1`
echo "data=$data"
```

4.5.2 let 运算工具

```
#!/bin/bash
let data=1+2+3+4+5+6
echo $data
```

4.5.3 判断闰年和平年

```
#!/bin/bash
echo -n "输入一个年份:"  #参数-n 的作用是不换行， echo 默认是换行
read year   #从键盘输入值存放到 year 中

#判断闰年与平年
if [ $(expr ${year} % 4) -eq 0 -a $(expr ${year} % 100) -ne 0 ] || [ $(expr ${year} % 400) -eq 0   ]
then
echo "${year} 是闰年"
else
echo "${year} 是平年"
fi

#说明: if 语句里两个独立的表达式可以使用
```

4.5.4 自增自减方法

```
#!/bin/bash
let data=1+2+3+4+5+6
echo $data

let data+=1
echo $data

((data++))
echo $data

data=$((data+1))
echo $data

data=$[data+1]
echo $data

#!/bin/bash
let data=1+2+3+4+5+6
echo $data
```

```
let data-=1
echo $data

((data--))
echo $data

data=$((data-1))
echo $data

data=$[data-1]
echo $data
```

4.6 for 循环使用案例

4.6.1 列表循环使用方法

```
#!/bin/bash
for loop in 1 2 3 4 5
do
echo "loop=${loop}"
done

#说明: in 后面是数据列表，for 循环每循环一次就将列表中的数据给 loop 变量
#直到所有数据都打印完毕之后，for 循环才结束

for data in 1234 5667 dfgvfgbvf dfdfvbd
do
echo "data=${data}"
done
```

4.6.2 C 语言风格 for 循环

单层循环示例:

```
#!/bin/bash
for((i=0;i<10;i++))
do
    echo "i=$i"
done
```

嵌套循环示例:

```
#!/bin/bash
cnt=0
for((i=0;i<5;i++))
do
    for((j=0;j<5;j++))
    do
        cnt=`expr $cnt + 1`
    done
done
echo "cnt=$cnt"
```

4.6.3 循环遍历目录文件

```
#!/bin/bash
for FILE_NAME in /test/*.sh
do
echo "文件名称=${FILE_NAME}"
done
#循环输出指定目录下的所有.sh 文件
```

4.6.4 列表形式输出 99 乘法口诀表

```
#!/bin/bash
for i in "1" "2" "3" "4" "5" "6" "7" "8" "9"
do
    for j in "1" "2" "3" "4" "5" "6" "7" "8" "9"
    do
            if [ ${j} -lt  ${i} ]
            then
                k=$((i * j))
                echo -n ${i}*${j}=${k}$'\t'
            fi
            if [ ${j} -eq ${i} ]
            then
                k=$((i * j))
                echo    ${i}*${j}=${k}
            fi
     done
done
```

4.7 while 循环使用案例

4.7.1 while 死循环方式

示例 1:

```
#!/bin/bash
while :
do
    echo "<-----数据输出------>"
    sleep 1   #延时 1 秒钟
done
#死循环示例，无限循环
```

示例 2:

```
#!/bin/bash
while :
do
    A=`expr ${A} + 1`
    echo "A=${A}"
    if [ ${A} == 5 ]
```

```
        then
            break 1     #1 表示跳出一层循环
        fi
        sleep 1    #延时 1 秒钟
    done
    #死循环示例，无限循环
```

4.7.2 while 循环 C 语言风格调用

```
示例 1:
    #!/bin/bash
    i=0
    while((i<=5))
    do
        echo $i
        let i++
    done

示例 2:
    #!/bin/bash
    a=0
    while((a!=5))
    do
        echo "a=$a"
        ((a++))
    done
```

4.7.3 循环读取键盘数据

```
    #!/bin/bash
    echo "<输入<CTRL-D>  或者<CTRL-C>结束程序>"
    echo -n "输入你最喜欢的数字: "
    while read DATA
    do
        echo "吉祥数字: $DATA"
    done
```

4.7.4 循环常规使用

```
    #!/bin/bash
    data=0
    while [ ${data} != 5 ]
    do
        data=`expr ${data} + 1`
        echo ${data}
    done
    #使用 while 循环将数据打印出来，条件不等于 5
```

4.7.5 输出 99 乘法口诀表

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

```
#输出 9*9 乘法口诀表
i=1
j=1
while [ $i -le 9 ]   #le 判断左边是否小于等于右边的
do
    while [ $j -le $i ]
    do
        printf "%d*%d=%d " $i $j `expr $i '*' $j`
        j=`expr $j + 1`
    done
    printf "\n"
    j=1
    i=`expr $i + 1`
done
```

4.7.6 冒泡排序

```
#!/bin/bash
echo "<-----下面展示的是冒泡排序程序---->"
echo -n "请输入排序数的数量:"
read s_cnt

echo -n "请输入数据(输入一个按下回车输入第 2 个):"

cnt=0 #定义计数的变量值
while :
do
    read buff[$cnt]   #将读取的数据存放到数组中
    cnt=`expr $cnt + 1`
    if [ $cnt == $s_cnt ]   #判断数据是否输入完毕
    then
        break   #跳出循环
    fi
done

echo "你输入的数据如下: ${buff[*]}"


#冒泡排序程序
s_cnt=`expr $s_cnt - 1 ` #最大长度减一
i=0
j=0
temp=0
while [ $i -lt $s_cnt ] #判断左边数是否小于右边数
do
    while [ $j -lt $s_cnt ]
    do
        if [ ${buff[$j]} -lt ${buff[`expr $j + 1`]} ]   #比较谁大谁小
```

```
            then
                temp=${buff[$j]}
                buff[$j]=${buff[`expr $j + 1`]}
                buff[`expr $j + 1`]=$temp
            fi
            j=`expr $j + 1`    #j++
        done
        i=`expr $i + 1`    #i++
        j=0    #j 变量归 0
    done

    echo "排序之后的数= ${buff[*]}"
```

4.7.7 局域网在线用户数量检测

```
#!/bin/bash
#写一个脚本， 通过 ping 命令测试 192.168.1.0 到 192.168.1.254 之间的所有主机是否在线
#          如果在线，就显示“在线”
#          如果不在线，就显示“离线”
#    注意：ping -c1 -w1  中-c1 是指 ping 的次数，-w 是指执行的最后期限，也就是执行的时间，单位为秒

cnt=0
count=0
while [ $cnt -lt 255 ]
do
    ping -c1 -w1 192.168.1.$cnt >log.txt
    if [ $? == 0 ]
    then
        echo "192.168.1.$cnt  在线!"
        count=`expr $count + 1`
    else
        echo "192.168.1.$cnt  不在线!"
    fi
    cnt=`expr $cnt + 1`
done

echo "一个共有$count  台设备在线!"
```

4.8 case 多分支语句使用案例

4.8.1 case 基本运用

```
#!/bin/bash
echo -n "输入一个 1~4 范围的数字:"
read Num
case $Num in
    1) echo "你选择  1"
    ;;
```

```
        2) echo "你选择 2"
        ;;
        3) echo "你选择 3"
        ;;
        4) echo "你选择 4"
        ;;
        *) echo "数字范围在 1 到 4 之间"
        ;;
esac
```

4.9 函数使用案例

4.9.1 函数参数传递

```
#!/bin/bash
#定义函数
fun()
{
    echo "第 1 个参数的值 ${1} "
    echo "第 2 个参数的值 ${2} "
    echo "第 3 个参数的值 ${3} "
    echo "参数个数 $# "
    echo "传递给函数的所有参数: $*"
}
fun 1 2 3 4 5
```

4.9.2 函数嵌套调用

```
#!/bin/bash
#定义函数
Hello()
{
    echo "Hello 函数调用成功!"
    func
}

func()
{
    echo "func 函数调用成功!"
}
Hello #调用函数
```

4.9.3 函数递归调用

```
#!/bin/bash
#定义函数
Hello()
{
    cnt=`expr ${cnt} + 1`
    echo "函数调用成功! cnt=${cnt}"
    sleep 1
```

```
    Hello      #递归调用
}
Hello #调用函数
```

4.9.4 接收函数返回值

```
#!/bin/bash
#定义函数
Hello()
{
    echo "函数调用成功!"
    return 88
}
Hello
echo "函数返回值:$?" #接收函数返回值
```

4.9.5 常规函数调用

```
#!/bin/bash
#定义函数
Hello()
{
    echo "Hello 函数调用成功!"
}

#定义函数
func()
{
    echo "func 函数调用成功!"
}
Hello #调用函数
func   #调用函数
```

4.9.6 判断闰年平年

```
#!/bin/bash
#判断平年润年函数封装
echo -n "请输入年份:"   #-n 表示不换新行
read year
get_year()
{
    if [ `expr $1 % 400` == 0 ] || [ `expr $1 % 4` == 0 -a `expr $1 % 100` != 0 ]
    then
        return 1
    else
        return 0
    fi
}
get_year $year
```

```
if [ $? == 0 ]
then
    echo "$year 是平年"
else
    echo "$year 是润年"
fi
```

4.9.7 输出水仙花数

```
#!/bin/bash
#输出所有水仙花数(个位+十位+百位的立方和=本身)例如: 1*1*1+2*2*2+3*3*3=123
cnt=100
a=0
b=0
c=0
sum=0
while [ $cnt -le 999 ]
do
    a=`expr $cnt / 100`
    b=`expr $cnt % 100 / 10`
    c=`expr $cnt % 10`
    sum=`expr $a '*' $a '*' $a`
    sum=`expr $b '*' $b '*' $b + $sum`
    sum=`expr $c '*' $c '*' $c + $sum`
    if [ $sum == $cnt ]
    then
        echo $cnt
    fi
    sum=0
    cnt=`expr $cnt + 1`
done
```

4.10 当前脚本调用其他脚本文件

4.10.1 基本运用

```
#!/bin/bash
. 123.sh   #方式 1
source 123.sh #方式 2
echo ${a}
```