

## 目录

<b>Shell 教程</b>	<b>3</b>
Shell 脚本	3
Shell 环境	3
第一个 shell 脚本	3
运行 Shell 脚本有两种方法:	4
<b>Shell 变量</b>	<b>4</b>
使用变量	4
只读变量	5
删除变量	5
变量类型	5
Shell 字符串	6
单引号	6
双引号	6
拼接字符串	6
获取字符串长度	6
提取子字符串	6
查找子字符串	7
Shell 数组	7
定义数组	7
读取数组	7
获取数组的长度	8
Shell 注释	8
<b>Shell 传递参数</b>	<b>8</b>
<b>Shell 数组</b>	<b>11</b>
读取数组	11
获取数组中的所有元素	12
获取数组的长度	12
<b>Shell 基本运算符</b>	<b>13</b>
算术运算符	13
关系运算符	15
布尔运算符	16
逻辑运算符	18
字符串运算符	19
文件测试运算符	20
<b>Shell echo 命令</b>	<b>22</b>
1.显示普通字符串:	22
2.显示转义字符	22
3.显示变量	23
4.显示换行	23
5.显示不换行	23
6.显示结果定向至文件	23
7.原样输出字符串, 不进行转义或取变量(用单引号)	23
8.显示命令执行结果	23

<b>Shell printf 命令</b>	<b>24</b>
<i>printf 的转义序列</i>	26
<b>Shell test 命令</b>	<b>26</b>
数值测试	26
字符串测试	27
文件测试	27
<b>Shell 流程控制</b>	<b>28</b>
<i>if else</i>	29
<i>if else-if else</i>	29
<i>for 循环</i>	30
<i>while 语句</i>	31
无限循环	32
<i>until 循环</i>	32
<i>case</i>	33
跳出循环	34
<i>break 命令</i>	34
<i>continue</i>	35
<b>Shell 函数</b>	<b>35</b>
函数参数	37
<b>Shell 输入/输出重定向</b>	<b>38</b>
输出重定向	38
输入重定向	39
重定向深入讲解	40
<i>Here Document</i>	40
<i>/dev/null 文件</i>	41
<b>Shell 文件包含</b>	<b>41</b>

# Shell 教程

Shell 是一个用C语言编写的程序，它是用户使用Linux的桥梁。Shell既是一种命令语言，又是一种程序设计语言。

Shell 是指一种应用程序，这个应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

Ken Thompson的sh是第一种Unix Shell，Windows Explorer是一个典型的图形界面Shell。

[Shell 在线工具](#)

## Shell 脚本

Shell 脚本（shell script），是一种为shell编写的脚本程序。

业界所说的shell通常都是指shell脚本，但读者朋友要知道，shell和shell script是两个不同的概念。

由于习惯的原因，简洁起见，本文出现的"shell编程"都是指shell脚本编程，不是指开发shell自身。

## 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，常见的有：

- Bourne Shell（/usr/bin/sh或/bin/sh）

- Bourne Again Shell（/bin/bash）

- C Shell（/usr/bin/csh）

- K Shell（/usr/bin/ksh）

- Shell for Root（/sbin/sh）

.....

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。

## 第一个 shell 脚本

打开文本编辑器(可以使用vi/vim命令来创建文件)，新建一个文件test.sh，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，如果你用php写shell 脚本，扩展名就用php好了。

输入一些代码，第一行一般是这样：

### 实例

```
#!/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，即使用哪一种Shell。  
echo命令用于向窗口输出文本。

## 运行 Shell 脚本有两种方法：

### 1、作为可执行程序

将上面的代码保存为test.sh，并cd到相应目录：

```
chmod +x ./test.sh #使脚本具有执行权限
```

```
./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。

### 2、作为解释器参数

这种运行方式是，直接运行解释器，其参数就是shell脚本的文件名，如：

```
/bin/sh test.sh
```

```
/bin/php test.php
```

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

## Shell 变量

定义变量时，变量名不加美元符号（\$，PHP语言中变量需要），如：

```
your_name="runoob.com"
```

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：

- 首个字符必须为字母（a-z, A-Z）。

- 中间不能有空格，可以使用下划线（\_）。

- 不能使用标点符号。

- 不能使用bash里的关键字（可用help命令查看保留关键字）。

除了显式地直接赋值，还可以用语句给变量赋值，如：

```
for file in `ls /etc`
```

以上语句将 /etc 下目录的文件名循环出来。

## 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：

```
your_name="qinjx"
```

```
echo $your_name
```

```
echo ${your_name}
```

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，比如下面这种情况：

```
for skill in Ada Coffee Action Java; do
    echo "I am good at ${skill}Script"
done
```

如果不给`skill`变量加花括号，写成`echo "I am good at $skillScript"`，解释器就会把`$skillScript`当成一个变量（其值为空），代码执行结果就不是我们期望的样子了。

推荐给所有变量加上花括号，这是个好的编程习惯。

已定义的变量，可以被重新定义，如：

```
your_name="tom"
echo $your_name
your_name="alibaba"
echo $your_name
```

这样写是合法的，但注意，第二次赋值的时候不能写`$your_name="alibaba"`，使用变量的时候才加美元符（\$）。

## 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
myUrl="http://www.w3cschool.cc"
readonly myUrl
myUrl="http://www.runoob.com"
```

运行脚本，结果如下：

```
/bin/sh: NAME: This variable is read only.
```

## 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。

### 实例

```
#!/bin/sh
myUrl="http://www.runoob.com"
unset myUrl
echo $myUrl
```

以上实例执行将没有任何输出。

## 变量类型

运行shell时，会同时存在三种变量：

**1) 局部变量** 局部变量在脚本或命令中定义，仅在当前shell实例中有效，其他shell启动的程序不能访问局部变量。

**2) 环境变量** 所有的程序，包括shell启动的程序，都能访问环境变量，有些程序需要环境变量来保证其正常运行。必要的时候shell脚本也可以定义环境变量。

**3) shell变量** shell变量是由shell程序设置的特殊变量。shell变量中有一部分是环境变量，有一部分是局部变量，这些变量保证了shell的正常运行

## 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。单双引号的区别跟PHP类似。

### 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；
- 单引号字符串中不能出现单引号（对单引号使用转义符后也不行）。

### 双引号

```
your_name='qinx'
```

```
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

## 拼接字符串

```
your_name="qinx"
```

```
greeting="hello, \"$your_name\" !"
```

```
greeting_1="hello, ${your_name} !"
```

```
echo $greeting $greeting_1
```

## 获取字符串长度

```
string="abcd"
```

```
echo ${#string} #输出 4
```

## 提取子字符串

以下实例从字符串第 2 个字符开始截取 4 个字符：

```
string="runoob is a great site"
```

```
echo ${string:1:4} # 输出 unoo
```

## 查找子字符串

查找字符 **"i 或 s"** 的位置:

```
string="runoob is a great company"
```

```
echo `expr index "$string" is` # 输出 8
```

**注意：** 以上脚本中 ``` 是反引号，而不是单引号 `"`，不要看错了哦。

## Shell 数组

**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**。

## 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：

```
数组名=(值1 值2 ... 值n)
```

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[n]=valuen
```

可以不使用连续的下标，而且下标的范围没有限制。

## 读取数组

读取数组元素值的一般格式是：

```
${数组名[下标]}
```

例如：

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

使用**@**符号可以获取数组中的所有元素，例如：

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

## 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，例如：

```
# 取得数组元素的个数

length=${#array_name[@]}

# 或者

length=${#array_name[*]}

# 取得数组单个元素的长度

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

## Shell 注释

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。只能像这样：

```
#-----

# 这是一个注释

# author: 菜鸟教程

# site: www.runoob.com

# slogan: 学的不仅是技术，更是梦想！

#-----

##### 用户配置区 开始 #####

#

#

# 这里可以添加脚本描述信息

#

#

##### 用户配置区 结束 #####
```

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。

## Shell 传递参数

我们可以在执行 Shell 脚本时，向脚本传递参数，脚本内获取参数的格式为：**\$n**。**n** 代表一个数字，1 为执行脚本的第一个参数，2 为执行脚本的第二个参数，以此类推.....



## 实例

以下实例我们向脚本传递三个参数，并分别输出，其中 **\$0** 为执行的文件名：

```
#!/bin/bash

# author:菜鸟教程
# url:www.runoob.com

echo "Shell 传递参数实例！";

echo "执行的文件名: $0";

echo "第一个参数为: $1";

echo "第二个参数为: $2";

echo "第三个参数为: $3";
```

为脚本设置可执行权限，并执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
$ ./test.sh 1 2 3
```

Shell 传递参数实例！

执行的文件名: test.sh

第一个参数为: 1

第二个参数为: 2

第三个参数为: 3

另外，还有几个特殊字符用来处理参数：

参数处理	说明
<b>\$#</b>	传递到脚本的参数个数
<b>\$*</b>	以一个单字符串显示所有向脚本传递的参数。 如"\$*"用「"」括起来的情况、以"\$1 \$2 ... \$n"的形式输出所有参数。
<b>\$\$</b>	脚本运行的当前进程ID号
<b>\$_</b>	后台运行的最后一个进程的ID号
<b>@</b>	与\$*相同，但是使用时加引号，并在引号中返回每个参数。 如"\$@"用「"」括起来的情况、以"\$1" "\$2" ... "\$n" 的形式输出所有参数。
<b>-</b>	显示Shell使用的当前选项，与set命令功能相同。
<b>?</b>	显示最后命令的退出状态。 <b>0</b> 表示没有错误，其他任何值表明有错误。

```
#!/bin/bash

# author:菜鸟教程
# url:www.runoob.com

echo "Shell 传递参数实例！";
```

```
echo "第一个参数为: $1";
```

```
echo "参数个数为: $#";
```

```
echo "传递的参数作为一个字符串显示: $*";
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
```

```
$ ./test.sh 1 2 3
```

Shell 传递参数实例！

第一个参数为: 1

参数个数为: 3

传递的参数作为一个字符串显示: 1 2 3

**\$\*** 与 **\$@** 区别：

相同点：都是引用所有参数。

不同点：只有在双引号中体现出来。假设在脚本运行时写了三个参数 1、2、3，，则 "**\***" 等价于 "**1 2 3**"（传递了一个参数），而 "**@**" 等价于 "**1**" "**2**" "**3**"（传递了三个参数）。

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
echo "-- \${*} 演示 ---"
```

```
for i in "${*}"; do
    echo $i
done
```

```
echo "-- \${@} 演示 ---"
```

```
for i in "${@}"; do
    echo $i
done
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
```

```
$ ./test.sh 1 2 3
```

```
-- ${*} 演示 ---
```

```
1 2 3
```

```
-- @$ 演示 ---
```

```
1
```

```
2
```

```
3
```

# Shell 数组

数组中可以存放多个值。**Bash Shell** 只支持一维数组（不支持多维数组），初始化时不需要定义数组大小（与 **PHP** 类似）。

与大部分编程语言类似，数组元素的下标由**0**开始。

**Shell** 数组用括号来表示，元素用"空格"符号分割开，语法格式如下：

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

## 实例

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
my_array=(A B "C" D)
```

我们也可以使用下标来定义数组：

```
array_name[0]=value0
```

```
array_name[1]=value1
```

```
array_name[2]=value2
```

## 读取数组

读取数组元素值的一般格式是：

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

## 实例

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
my_array=(A B "C" D)
```

```
echo "第一个元素为: ${my_array[0]}"
```

```
echo "第二个元素为: ${my_array[1]}"
```

```
echo "第三个元素为: ${my_array[2]}"
```

```
echo "第四个元素为: ${my_array[3]}"
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
```

```
$ ./test.sh
```

第一个元素为： A

第二个元素为： B

第三个元素为： C

第四个元素为： D

## 获取数组中的所有元素

使用@ 或 \* 可以获取数组中的所有元素，例如：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
my_array[0]=A
```

```
my_array[1]=B
```

```
my_array[2]=C
```

```
my_array[3]=D
```

```
echo "数组的元素为： ${my_array[*]}"
```

```
echo "数组的元素为： ${my_array[@]}"
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
```

```
$ ./test.sh
```

数组的元素为： A B C D

数组的元素为： A B C D

## 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，例如：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
my_array[0]=A
```

```
my_array[1]=B
```

```
my_array[2]=C
```

```
my_array[3]=D
```

```
echo "数组元素个数为： ${#my_array[*]}"
```

```
echo "数组元素个数为： ${#my_array[@]}"
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
$ ./test.sh
```

数组元素个数为： 4

数组元素个数为： 4

## Shell 基本运算符

Shell 和其他编程语言一样，支持多种运算符，包括：

- 算数运算符
- 关系运算符
- 布尔运算符
- 字符串运算符
- 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(注意使用的是反引号 ``` 而不是单引号 `'`)：

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

```
val=`expr 2 + 2`
```

```
echo "两数之和为 : $val"
```

执行脚本，输出结果如下所示：

两数之和为 : 4

两点注意：

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。

## 算术运算符

下表列出了常用的算术运算符，假定变量 **a** 为 10，变量 **b** 为 20：

运算符	说明	举例
+	加法	<code>`expr \$a + \$b`</code> 结果为 30。
-	减法	<code>`expr \$a - \$b`</code> 结果为 10。
*	乘法	<code>`expr \$a \* \$b`</code> 结果为 200。
/	除法	<code>`expr \$b / \$a`</code> 结果为 2。
%	取余	<code>`expr \$b % \$a`</code> 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 ]**。

## 实例

算术运算符实例如下：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
a=10
```

```
b=20
```

```
val=`expr $a + $b`  
echo "a + b : $val"
```

```
val=`expr $a - $b`  
echo "a - b : $val"
```

```
val=`expr $a \* $b`  
echo "a * b : $val"
```

```
val=`expr $b / $a`  
echo "b / a : $val"
```

```
val=`expr $b % $a`  
echo "b % a : $val"
```

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

```
fi  
if [ $a != $b ]  
then
```

```
    echo "a 不等于 b"
```

```
fi
```

执行脚本，输出结果如下所示：

```
a + b : 30
```

```
a - b : -10
```

```
a * b : 200
```

```
b / a : 2
```

```
b % a : 0
```

```
a 不等于 b
```

## 注意:

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

## 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。  
下表列出了常用的关系运算符, 假定变量 a 为 10, 变量 b 为 20:

运算符	说明	举例
-eq	检测两个数是否相等, 相等返回 true。	[ \$a -eq \$b ] 返回 false。
-ne	检测两个数是否相等, 不相等返回 true。	[ \$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。
-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。

## 实例

关系运算符实例如下:

```
#!/bin/bash

# author:菜鸟教程
# url:www.runoob.com

a=10
b=20

if [ $a -eq $b ]
then
    echo "$a -eq $b : a 等于 b"
else
    echo "$a -eq $b: a 不等于 b"
fi
if [ $a -ne $b ]
then
    echo "$a -ne $b: a 不等于 b"
else
    echo "$a -ne $b : a 等于 b"
fi
if [ $a -gt $b ]
then
    echo "$a -gt $b: a 大于 b"
```

```

else
    echo "$a -gt $b: a 不大于 b"
fi
if [ $a -lt $b ]
then
    echo "$a -lt $b: a 小于 b"
else
    echo "$a -lt $b: a 不小于 b"
fi
if [ $a -ge $b ]
then
    echo "$a -ge $b: a 大于或等于 b"
else
    echo "$a -ge $b: a 小于 b"
fi
if [ $a -le $b ]
then
    echo "$a -le $b: a 小于或等于 b"
else
    echo "$a -le $b: a 大于 b"
fi

```

执行脚本，输出结果如下所示：

```

10 -eq 20: a 不等于 b

10 -ne 20: a 不等于 b

10 -gt 20: a 不大于 b

10 -lt 20: a 小于 b

10 -ge 20: a 小于 b

10 -le 20: a 小于或等于 b

```

## 布尔运算符

下表列出了常用的布尔运算符，假定变量 **a** 为 10，变量 **b** 为 20：



运算符	说明	举例
!	非运算，表达式为 true 则返回 false，否则返回 true。	[ ! false ] 返回 true。
-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。

## 实例

布尔运算符实例如下：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
a=10
```

```
b=20
```

```
if [ $a != $b ]
```

```
then
```

```
    echo "$a != $b : a 不等于 b"
```

```
else
```

```
    echo "$a != $b: a 等于 b"
```

```
fi
```

```
if [ $a -lt 100 -a $b -gt 15 ]
```

```
then
```

```
    echo "$a -lt 100 -a $b -gt 15 : 返回 true"
```

```
else
```

```
    echo "$a -lt 100 -a $b -gt 15 : 返回 false"
```

```
fi
```

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

```
then
```

```
    echo "$a -lt 100 -o $b -gt 100 : 返回 true"
```

```
else
```

```
    echo "$a -lt 100 -o $b -gt 100 : 返回 false"
```

```
fi
```

```
if [ $a -lt 5 -o $b -gt 100 ]
```

```
then
```

```
    echo "$a -lt 100 -o $b -gt 100 : 返回 true"
```

```
else
```

```
    echo "$a -lt 100 -o $b -gt 100 : 返回 false"
```

```
fi
```

执行脚本，输出结果如下所示：

```
10 != 20 : a 不等于 b
```

```
10 -lt 100 -a 20 -gt 15 : 返回 true
```

```
10 -lt 100 -o 20 -gt 100 : 返回 true
```

```
10 -lt 100 -o 20 -gt 100 : 返回 false
```

## 逻辑运算符

以下介绍 Shell 的逻辑运算符，假定变量 a 为 10，变量 b 为 20:

运算符	说明	举例
&&	逻辑的 AND	[[ \$a -lt 100 && \$b -gt 100 ]] 返回 false
	逻辑的 OR	[[ \$a -lt 100    \$b -gt 100 ]] 返回 true

## 实例

逻辑运算符实例如下:

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
a=10
```

```
b=20
```

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

```
    echo "返回 true"
```

```
else
```

```
    echo "返回 false"
```

```
fi
```

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

```
    echo "返回 true"
```

```
else
```

```
    echo "返回 false"
```

```
fi
```

执行脚本，输出结果如下所示:

```
返回 false
```

```
返回 true
```

## 字符串运算符

下表列出了常用的字符串运算符，假定变量 a 为 "abc"，变量 b 为 "efg"：

运算符	说明	举例
=	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。	[ -n \$a ] 返回 true。
str	检测字符串是否为空，不为空返回 true。	[ \$a ] 返回 true。

## 实例

字符串运算符实例如下：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
a="abc"
```

```
b="efg"
```

```
if [ $a = $b ]
```

```
then
```

```
    echo "$a = $b : a 等于 b"
```

```
else
```

```
    echo "$a = $b: a 不等于 b"
```

```
fi
```

```
if [ $a != $b ]
```

```
then
```

```
    echo "$a != $b : a 不等于 b"
```

```
else
```

```
    echo "$a != $b: a 等于 b"
```

```
fi
```

```
if [ -z $a ]
```

```
then
```

```
    echo "-z $a : 字符串长度为 0"
```

```
else
```

```
    echo "-z $a : 字符串长度不为 0"
```

```
fi
```

```
if [ -n $a ]
```

```
then
```

```
    echo "-n $a : 字符串长度不为 0"
```

```
else

    echo "-n $a : 字符串长度为 0"

fi
if [ $a ]
then

    echo "$a : 字符串不为空"

else

    echo "$a : 字符串为空"

fi
```

执行脚本，输出结果如下所示：

```
abc = efg: a 不等于 b

abc != efg : a 不等于 b

-z abc : 字符串长度不为 0

-n abc : 字符串长度不为 0

abc : 字符串不为空
```

## 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。  
属性检测描述如下：

操作符	说明
-b file	检测文件是否是块设备文件，如果是，则返回 true。
-c file	检测文件是否是字符设备文件，如果是，则返回 true。
-d file	检测文件是否是目录，如果是，则返回 true。
-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。
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true。
-k file	检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。
-p file	检测文件是否是具名管道，如果是，则返回 true。
-u file	检测文件是否设置了 SUID 位，如果是，则返回 true。
-r file	检测文件是否可读，如果是，则返回 true。
-w file	检测文件是否可写，如果是，则返回 true。
-x file	检测文件是否可执行，如果是，则返回 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。
-e file	检测文件（包括目录）是否存在，如果是，则返回 true。

## 实例

变量 **file** 表示文件"/var/www/runoob/test.sh"，它的大小为100字节，具有 **rw**x 权限。下面的代码，将检测该文件的各种属性：

```
#!/bin/bash

# author:菜鸟教程
# url:www.runoob.com

file="/var/www/runoob/test.sh"
if [ -r $file ]
then
    echo "文件可读"
else
    echo "文件不可读"
fi
if [ -w $file ]
then
    echo "文件可写"
else
    echo "文件不可写"
fi
if [ -x $file ]
then
    echo "文件可执行"
else
    echo "文件不可执行"
fi
if [ -f $file ]
then
    echo "文件为普通文件"
else
    echo "文件为特殊文件"
fi
if [ -d $file ]
then
    echo "文件是个目录"
else
    echo "文件不是个目录"
fi
```

```
if [ -s $file ]
then
    echo "文件不为空"
else
    echo "文件为空"
fi
if [ -e $file ]
then
    echo "文件存在"
else
    echo "文件不存在"
fi
```

执行脚本，输出结果如下所示：

```
文件可读
文件可写
文件可执行
文件为普通文件
文件不是个目录
文件不为空
文件存在
```

## Shell echo 命令

Shell 的 `echo` 指令与 PHP 的 `echo` 指令类似，都是用于字符串的输出。命令格式：

```
echo string
```

您可以使用 `echo` 实现更复杂的输出格式控制。

### 1.显示普通字符串：

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

这里的双引号完全可以省略，以下命令与上面实例效果一致：

```
echo It is a test
```

### 2.显示转义字符

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

结果将是：

```
"It is a test"
```

同样，双引号也可以省略

### 3.显示变量

**read** 命令从标准输入中读取一行,并把输入行的每个字段的值指定给 **shell** 变量

```
#!/bin/sh
```

```
read name
```

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

以上代码保存为 **test.sh**, **name** 接收标准输入的变量, 结果将是:

```
[root@www ~]# sh test.sh
```

```
OK                                #标准输入
```

```
OK It is a test                  #输出
```

### 4.显示换行

```
echo -e "OK! \n" # -e 开启转义
```

```
echo "It it a test"
```

输出结果:

```
OK!
```

```
It it a test
```

### 5.显示不换行

```
#!/bin/sh
```

```
echo -e "OK! \c" # -e 开启转义 \c 不换行
```

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

输出结果:

```
OK! It is a test
```

### 6.显示结果定向至文件

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

### 7.原样输出字符串, 不进行转义或取变量(用单引号)

```
echo '$name\''
```

输出结果:

```
$name\''
```

### 8.显示命令执行结果

```
echo `date`
```

结果将显示当前日期

Thu Jul 24 10:08:46 CST 2014

## Shell printf 命令

上一章节我们学习了 Shell 的 echo 命令，本章节我们来学习 Shell 的另一个输出命令 printf。

printf 命令模仿 C 程序库 (library) 里的 printf() 程序。

标准所定义，因此使用printf的脚本比使用echo移植性好。

printf 使用引用文本或空格分隔的参数，外面可以在printf中使用格式化字符串，还可以制定字符串的宽度、左右对齐方式等。默认printf不会像 echo 自动添加换行符，我们可以手动添加 \n。

printf 命令的语法：

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

参数说明：

**format-string:** 为格式控制字符串

**arguments:** 为参数列表。

实例如下：

```
$ echo "Hello, Shell"
```

```
Hello, Shell
```

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

```
Hello, Shell
```

```
$
```

接下来,我来用一个脚本来体现printf的强大功能：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
printf "%-10s %-8s %-4s\n" 姓名 性别 体重kg
```

```
printf "%-10s %-8s %-4.2f\n" 郭靖 男 66.1234
```

```
printf "%-10s %-8s %-4.2f\n" 杨过 男 48.6543
```

```
printf "%-10s %-8s %-4.2f\n" 郭芙 女 47.9876
```

执行脚本，输出结果如下所示：

```
姓名      性别    体重kg
```

```
郭靖      男       66.12
```

```
杨过      男       48.65
```

```
郭芙      女       47.99
```



`%s %c %d %f`都是格式替代符

`%-10s` 指一个宽度为10个字符（-表示左对齐，没有则表示右对齐），任何字符都会被显示在10个字符宽的字符内，如果不足则自动以空格填充，超过也会将内容全部显示出来。

`%-4.2f` 指格式化为小数，其中.2指保留2位小数。

更多实例：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
# format-string为双引号
```

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

```
# 单引号与双引号效果一样
```

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

```
# 没有引号也可以输出
```

```
printf %s abcdef
```

```
# 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
```

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

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

```
# 如果没有 arguments，那么 %s 用NULL代替，%d 用 0 代替
```

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

执行脚本，输出结果如下所示：

```
1 abc
```

```
1 abc
```

```
abcdefabcdefabc
```

```
def
```

```
a b c
```

```
d e f
```

```
g h i
```

```
j
```

```
and 0
```

## printf 的转义序列

序列	说明
\a	警告字符，通常为ASCII的BEL字符
\b	后退
\c	抑制（不显示）输出结果中任何结尾的换行字符（只在 <b>%b</b> 格式指示符控制下的参数字符串中有字符、任何接下来的参数以及任何留在格式字符串中的字符，都被忽略
\f	换页（formfeed）
\n	换行
\r	回车（Carriage return）
\t	水平制表符
\v	垂直制表符
\\	一个字面上的反斜杠字符
\ddd	表示1到3位数八进制值的字符。仅在格式字符串中有效
\Odd d	表示1到3位的八进制值字符

## 实例

```
$ printf "a string, no processing:<%s>\n" "A\nB"  
a string, no processing:<A\nB>
```

```
$ printf "a string, no processing:<%b>\n" "A\nB"  
a string, no processing:<A  
B>
```

```
$ printf "www.runoob.com \a"  
www.runoob.com $          #不换行
```

## Shell test 命令

Shell中的 **test** 命令用于检查某个条件是否成立，它可以进行数值、字符和文件三个方面的测试。

## 数值测试

参数	说明
-eq	等于则为真
-ne	不等于则为真
-gt	大于则为真
-ge	大于等于则为真
-lt	小于则为真
-le	小于等于则为真

实例演示：

```
num1=100
num2=100
if test ${num1} -eq ${num2}
then

    echo '两个数相等！'

else

    echo '两个数不相等！'

fi
```

输出结果：

两个数相等！

## 字符串测试

参数	说明
=	等于则为真
!=	不相等则为真
-z 字符串	字符串的长度为零则为真
-n 字符串	字符串的长度不为零则为真

实例演示：

```
num1="runoob"
num2="runoob"
if test num1=num2
then

    echo '两个字符串相等！'

else

    echo '两个字符串不相等！'

fi
```

输出结果：

两个字符串相等！

## 文件测试

参数	说明
-e 文件名	如果文件存在则为真
-r 文件名	如果文件存在且可读则为真
-w 文件名	如果文件存在且可写则为真
-x 文件名	如果文件存在且可执行则为真
-s 文件名	如果文件存在且至少有一个字符则为真
-d 文件名	如果文件存在且为目录则为真

<b>-f</b> 文件名	如果文件存在且为普通文件则为真
<b>-c</b> 文件名	如果文件存在且为字符型特殊文件则为真
<b>-b</b> 文件名	如果文件存在且为块特殊文件则为真

实例演示:

```
cd /bin
if test -e ./bash
then

    echo '文件已存在!'
else

    echo '文件不存在!'
fi
```

输出结果:

文件已存在!

另外, **Shell**还提供了与(**-a**)、或(**-o**)、非(!)三个逻辑操作符用于将测试条件连接起来, 其优先级为: "!"最高, "**-a**"次之, "**-o**"最低。例如:

```
cd /bin
if test -e ./notFile -o -e ./bash
then

    echo '有一个文件存在!'
else

    echo '两个文件都不存在'
fi
```

输出结果:

有一个文件存在!

## Shell 流程控制

和Java、PHP等语言不一样, **sh**的流程控制不可为空, 如(以下为PHP流程控制写法):

```
<?php
if (isset($_GET["q"])) {
    search(q);
}
else {

    // 不做任何事情
}
```

在**sh/bash**里可不能这么写, 如果**else**分支没有语句执行, 就不要写这个**else**。

## if else

### if

if 语句语法格式：

```
if condition
then
    command1
    command2
    ...
    commandN
fi
```

写成一行（适用于终端命令提示符）：

```
if [ $(ps -ef | grep -c "ssh") -gt 1 ]; then echo "true"; fi
```

末尾的fi就是if倒过来拼写，后面还会遇到类似的。

### if else

if else 语法格式：

```
if condition
then
    command1
    command2
    ...
    commandN
else
    command
fi
```

### if else-if else

if else-if else 语法格式：

```
if condition1
then
    command1
elif condition2
then
    command2
else
    commandN
fi
```

以下实例判断两个变量是否相等：

```
a=10
b=20
if [ $a == $b ]
then
    echo "a 等于 b"
elif [ $a -gt $b ]
```

```

then

    echo "a 大于 b"
elif [ $a -lt $b ]
then
    echo "a 小于 b"
else
    echo "没有符合的条件"
fi

```

输出结果:

a 小于 b

**if else**语句经常与**test**命令结合使用，如下所示:

```

num1=$((2*3))
num2=$((1+5))
if test $[num1] -eq $[num2]
then
    echo '两个数字相等!'

```

```

else

```

```

    echo '两个数字不相等!'

```

```

fi

```

输出结果:

两个数字相等!

## for 循环

与其他编程语言类似，**Shell**支持**for**循环。

**for**循环一般格式为:

```

for var in item1 item2 ... itemN
do
    command1
    command2
    ...
    commandN
done

```

写成一行:

```

for var in item1 item2 ... itemN; do command1; command2... done;

```

当变量值在列表里，**for**循环即执行一次所有命令，使用变量名获取列表中的当前取值。命令可为任何有效的**shell**命令和语句。**in**列表可以包含替换、字符串和文件名。

**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。

例如，顺序输出当前列表中的数字:

```

for loop in 1 2 3 4 5

```

```
do
    echo "The value is: $loop"
done
输出结果：
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5
顺序输出字符串中的字符：
for str in 'This is a string'
do
    echo $str
done
输出结果：
This is a string
```

## while 语句

**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。其格式为：

```
while condition
do
    command
done
```

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

```
#!/bin/sh
int=1
while(( $int<=5 ))
do
    echo $int
    let "int++"
done
```

运行脚本，输出：

```
1
2
3
4
5
```

使用中使用了 **Bash let** 命令，它用于执行一个或多个表达式，变量计算中不需要加上 **\$** 来表示变量，具体可查阅：[Bash let 命令](#)

。

**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**FILM**，按<Ctrl-D>结束循环。

```
echo '按下 <CTRL-D> 退出'
```

```
echo -n '输入你最喜欢的电影名： '

while read FILM
do

    echo "是的！$FILM 是一部好电影"

done
```

运行脚本，输出类似下面：

按下 <CTRL-D> 退出

输入你最喜欢的电影名： w3cschool菜鸟教程

是的！w3cschool菜鸟教程 是一部好电影

## 无限循环

无限循环语法格式：

```
while :
do
    command
done
或者
while true
do
    command
done
或者
for (( ; ; ))
```

## until 循环

until循环执行一系列命令直至条件为真时停止。

until循环与while循环在处理方式上刚好相反。

一般while循环优于until循环，但在某些时候——也只是极少数情况下，until循环更加有用。

until 语法格式：

```
until condition
do
    command
done
```

条件可为任意测试条件，测试发生在循环末尾，因此循环至少执行一次——请注意这一点。



## case

Shell **case**语句为多选择语句。可以用**case**语句匹配一个值与一个模式，如果匹配成功，执行相匹配的命令。**case**语句格式如下：

```
case 值 in
```

模式1)

```
    command1
    command2
    ...
    commandN
;;
```

模式2)

```
    command1
    command2
    ...
    commandN
;;
```

```
esac
```

**case**工作方式如上所示。取值后面必须为单词**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 **;;**。

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。

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 1 到 4 之间的数字:'
```

```
echo '你输入的数字为:'
```

```
read aNum
```

```
case $aNum in
```

```
    1) echo '你选择了 1'
```

```
;;
```

```
    2) echo '你选择了 2'
```

```
;;
```

```
    3) echo '你选择了 3'
```

```
;;
```

```
    4) echo '你选择了 4'
```

```
;;
```

```
*) echo '你没有输入 1 到 4 之间的数字'
```

```
;;
```

```
esac
```

输入不同的内容，会有不同的结果，例如：

输入 1 到 4 之间的数字：

你输入的数字为：

3

你选择了 3

## 跳出循环

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，Shell使用两个命令来实现该功能：**break**和**continue**。

### 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，需要使用**break**命令。

```
#!/bin/bash
while :
do
    echo -n "输入 1 到 5 之间的数字："

    read aNum
    case $aNum in
        1|2|3|4|5) echo "你输入的数字为 $aNum!"
            ;;
        *) echo "你输入的数字不是 1 到 5 之间的！游戏结束"
            break
            ;;
    esac
done
```

执行以上代码，输出结果为：

输入 1 到 5 之间的数字:3

你输入的数字为 3！

输入 1 到 5 之间的数字:7

你输入的数字不是 1 到 5 之间的！游戏结束

## continue

**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：

```
#!/bin/bash
while :
do

    echo -n "输入 1 到 5 之间的数字： "

    read aNum
    case $aNum in
        1|2|3|4|5) echo "你输入的数字为 $aNum!"
            ;;
        *) echo "你输入的数字不是 1 到 5 之间的!"
            continue
            echo "游戏结束"
            ;;
    esac
done
```

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，语句 **echo "Game is over!"** 永远不会被执行。

## esac

**case** 的语法和 **C family** 语言差别很大，它需要一个 **esac**（就是 **case** 反过来）作为结束标记，每个 **case** 分支用右圆括号，用两个分号表示 **break**。

## Shell 函数

linux shell 可以用户定义函数，然后在shell脚本中可以随便调用。

shell中函数的定义格式如下：

```
[ function ] funname [()]
```

```
{

    action;

    [return int;]

}
```

说明：

- 1、可以带function fun() 定义，也可以直接fun() 定义,不带任何参数。

2、参数返回，可以显示加：**return** 返回，如果不加，将以最后一条命令运行结果，作为返回值。**return**后跟数值n(0-255)

下面的例子定义了一个函数并进行调用：

```
#!/bin/bash

# author:菜鸟教程
# url:www.runoob.com

demoFun(){
    echo "这是我的第一个 shell 函数!"
}
```

```
echo "-----函数开始执行-----"
```

```
demoFun
```

```
echo "-----函数执行完毕-----"
```

输出结果：

```
-----函数开始执行-----
```

```
这是我的第一个 shell 函数!
```

```
-----函数执行完毕-----
```

下面定义一个带有**return**语句的函数：

```
#!/bin/bash

# author:菜鸟教程
# url:www.runoob.com

funWithReturn(){
    echo "这个函数会对输入的两个数字进行相加运算..."

    echo "输入第一个数字: "
    read aNum

    echo "输入第二个数字: "
    read anotherNum

    echo "两个数字分别为 $aNum 和 $anotherNum !"
    return $(( $aNum+$anotherNum ))
}

funWithReturn
```

```
echo "输入的两个数字之和为 $? !"
```

输出类似下面：

这个函数会对输入的两个数字进行相加运算...

输入第一个数字：

1

输入第二个数字：

2

两个数字分别为 1 和 2 ！

输入的两个数字之和为 3 ！

函数返回值在调用该函数后通过 `$?` 来获得。

注意：所有函数在使用前必须定义。这意味着必须将函数放在脚本开始部分，直至shell解释器首次发现它时，才可以使用。调用函数仅使用其函数名即可。

## 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

# author:菜鸟教程
# url:www.runoob.com

funWithParam() {
    echo "第一个参数为 $1 !"
    echo "第二个参数为 $2 !"
    echo "第十个参数为 $10 !"
    echo "第十个参数为 ${10} !"
    echo "第十一个参数为 ${11} !"
    echo "参数总数有 $# 个!"
    echo "作为一个字符串输出所有参数 $* !"
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

输出结果：

第一个参数为 1 ！

第二个参数为 2 ！

第十个参数为 10 ！

第十个参数为 34 ！

第十一个参数为 73 ！

参数总数有 11 个！

作为一个字符串输出所有参数 1 2 3 4 5 6 7 8 9 34 73 ！

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，还有几个特殊字符用来处理参数：

参数处理	说明
\$#	传递到脚本的参数个数
\$*	以一个单字符串显示所有向脚本传递的参数
\$\$	脚本运行的当前进程ID号
\$_	后台运行的最后一个进程的ID号
@	与\$*相同，但是使用时加引号，并在引号中返回每个参数。
-	显示Shell使用的当前选项，与set命令功能相同。
?	显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误。

## Shell 输入/输出重定向

大多数 UNIX 系统命令从你的终端接受输入并将所产生的输出发送回 到您的终端。一个命令通常从一个叫标准输入的地方读取输入，默认情况下，这恰好是你的终端。同样，一个命令通常将其输出写入到标准输出，默认情况下，这也是你的终端。

重定向命令列表如下：

命令	说明
command > file	将输出重定向到 file。
command < file	将输入重定向到 file。
command >> file	将输出以追加的方式重定向到 file。
n > file	将文件描述符为 n 的文件重定向到 file。
n >> file	将文件描述符为 n 的文件以追加的方式重定向到 file。
n >& m	将输出文件 m 和 n 合并。
n <& m	将输入文件 m 和 n 合并。
<< tag	将开始标记 tag 和结束标记 tag 之间的内容作为输入。

需要注意的是文件描述符 0 通常是标准输入（STDIN），1 是标准输出（STDOUT），2 是标准错误输出（STDERR）。

## 输出重定向

重定向一般通过在命令间插入特定的符号来实现。特别的，这些符号的语法如下所示：

```
command1 > file1
```

上面这个命令执行**command1**然后将输出的内容存入**file1**。

注意任何**file1**内的已经存在的内容将被新内容替代。如果要将新内容添加在文件末尾，请使用>>操作符。

## 实例

执行下面的 **who** 命令，它将命令的完整的输出重定向在用户文件中(**users**):

```
$ who > users
```

执行后，并没有在终端输出信息，这是因为输出已被从默认的标准输出设备（终端）重定向到指定的文件。

你可以使用 **cat** 命令查看文件内容：

```
$ cat users
```

```
_mbsetupuser console Oct 31 17:35
```

```
tianqixin console Oct 31 17:35
```

```
tianqixin ttys000 Dec 1 11:33
```

输出重定向会覆盖文件内容，请看下面的例子：

```
$ echo "菜鸟教程：www.runoob.com" > users
```

```
$ cat users
```

```
菜鸟教程：www.runoob.com
```

```
$
```

如果不希望文件内容被覆盖，可以使用 >> 追加到文件末尾，例如：

```
$ echo "菜鸟教程：www.runoob.com" >> users
```

```
$ cat users
```

```
菜鸟教程：www.runoob.com
```

```
菜鸟教程：www.runoob.com
```

```
$
```

## 输入重定向

和输出重定向一样，**Unix** 命令也可以从文件获取输入，语法为：

```
command1 < file1
```

这样，本来需要从键盘获取输入的命令会转移到文件读取内容。

注意：输出重定向是大于号(>)，输入重定向是小于号(<)。

## 实例

接着以上实例，我们需要统计 **users** 文件的行数,执行以下命令：

```
$ wc -l users
```

```
2 users
```

也可以将输入重定向到 **users** 文件：

```
$ wc -l < users
```

```
2
```

注意：上面两个例子的结果不同：第一个例子，会输出文件名；第二个不会，因为它仅仅知道从标准输入读取内容。

```
command1 < infile > outfile
```

同时替换输入和输出，执行**command1**，从文件**infile**读取内容，然后将输出写入到**outfile**中。

## 重定向深入讲解

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

标准输入文件(**stdin**)：**stdin**的文件描述符为0，Unix程序默认从**stdin**读取数据。

标准输出文件(**stdout**)：**stdout** 的文件描述符为1，Unix程序默认向**stdout**输出数据。

标准错误文件(**stderr**)：**stderr**的文件描述符为2，Unix程序会向**stderr**流中写入错误信息。

默认情况下，**command > file** 将 **stdout** 重定向到 **file**，**command < file** 将**stdin** 重定向到 **file**。

如果希望 **stderr** 重定向到 **file**，可以这样写：

```
$ command 2 > file
```

如果希望 **stderr** 追加到 **file** 文件末尾，可以这样写：

```
$ command 2 >> file
```

**2** 表示标准错误文件(**stderr**)。

如果希望将 **stdout** 和 **stderr** 合并后重定向到 **file**，可以这样写：

```
$ command > file 2>&1
```

或者

```
$ command >> file 2>&1
```

如果希望对 **stdin** 和 **stdout** 都重定向，可以这样写：

```
$ command < file1 >file2
```

**command** 命令将 **stdin** 重定向到 **file1**，将 **stdout** 重定向到 **file2**。

## Here Document

**Here Document** 是 Shell 中的一种特殊的重定向方式，用来将输入重定向到一个交互式 Shell 脚本或程序。

它的基本的形式如下：

```
command << delimiter
```

```
    document
```

```
delimiter
```

它的作用是将两个 **delimiter** 之间的内容(**document**) 作为输入传递给 **command**。

注意：

结尾的**delimiter** 一定要顶格写，前面不能有任何字符，后面也不能有任何字符，包括空格和 **tab** 缩进。

开始的**delimiter**前后的空格会被忽略掉。

## 实例

在命令行中通过 **wc -l** 命令计算 **Here Document** 的行数：

```
$ wc -l << EOF
```



```
欢迎来到
菜鸟教程
www.runoob.com
```

EOF

```
3          # 输出结果为 3 行
```

\$

我们也可以将 **Here Document** 用在脚本中，例如：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
cat << EOF
```

```
欢迎来到
```

```
菜鸟教程
```

```
www.runoob.com
```

```
EOF
```

执行以上脚本，输出结果：

```
欢迎来到
```

```
菜鸟教程
```

```
www.runoob.com
```

## /dev/null 文件

如果希望执行某个命令，但又不希望在屏幕上显示输出结果，那么可以将输出重定向到 **/dev/null**：

```
$ command > /dev/null
```

**/dev/null** 是一个特殊的文件，写入到它的内容都会被丢弃；如果尝试从该文件读取内容，那么什么也读不到。但是 **/dev/null** 文件非常有用，将命令的输出重定向到它，会起到"禁止输出"的效果。

如果希望屏蔽 **stdout** 和 **stderr**，可以这样写：

```
$ command > /dev/null 2>&1
```

**注意：**0 是标准输入（**STDIN**），1 是标准输出（**STDOUT**），2 是标准错误输出（**STDERR**）。

## Shell 文件包含

和其他语言一样，**Shell** 也可以包含外部脚本。这样可以很方便的封装一些公用的代码作为一个独立的文件。

**Shell** 文件包含的语法格式如下：

```
. filename    # 注意点号(.)和文件名中间有一空格
```

或

```
source filename
```

## 实例

创建两个 **shell** 脚本文件。

**test1.sh** 代码如下：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
url="http://www.runoob.com"
```

**test2.sh** 代码如下：

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

```
# author:菜鸟教程
```

```
# url:www.runoob.com
```

```
#使用 . 号来引用test1.sh 文件
```

```
. ./test1.sh
```

```
# 或者使用以下包含文件代码
```

```
# source ./test1.sh
```

```
echo "菜鸟教程官网地址: $url"
```

接下来，我们为 **test2.sh** 添加可执行权限并执行：

```
$ chmod +x test2.sh
```

```
$ ./test2.sh
```

```
菜鸟教程官网地址: http://www.runoob.com
```

**注：**被包含的文件 **test1.sh** 不需要可执行权限。