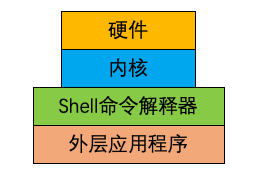
# Shell概述



shell的两种语法类型有Bourne和C，这两种语法彼此不兼容。Bourne家族主要包括sh、ksh、Bash、psh、zsh；C家族主要包括：csh、tsh（Bash和zsh在不同程度上支持csh的语法）。我们可以通过vi /etc/shells命令来查询Linux支持的shell。

| **shell类别** | **易学性** | **可移植性** | **编辑性** | **快捷性** |
| --- | --- | --- | --- | --- |
| Bourne Shell（sh） | 容易 | 号 | 较差 | 较差 |
| Korn Shell（ksh） | 较难 | 较好 | 好 | 较好 |
| Bourne Again（Bash） | 难 | 较好 | 好 | 好 |
| POSIX Shell（psh） | 较难 | 好 | 好 | 较好 |
| C Shell | 较难 | 差 | 较好 | 较好 |
| TC Shell（tcsh） | 难 | 差 | 好 | 好 |

# Shell语法

## 变量

变量名的命名须遵循如下规则：

* 命名只能使用英文字母，数字和下划线，首个字符不能以数字开头。
* 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）。

变量分类：

* 用户自定义变量（局部变量）：这种变量是最常见的变量，由用户自由定义变量名和变量的值
* 环境变量：这种变量中主要保存的是和系统操作环境相关的数据，比如当前登录用户，用户的家目录，命令的提示符等。不是太好理解吧，那么大家还记得在Windows中，同一台电脑可以有多个用户登录，而且每个用户都可以定义自己的桌面样式和分辨率，这些其实就是Windows的操作环境，可以当做是Windows的环境变量来理解。环境变量的变量名可以自由定义，但是一般对系统起作用的环境变量的变量名是系统预先设定好的
* 位置参数变量：这种变量主要是用来向脚本当中传递参数或数据的，变量名不能自定义，变量作用是固定的
* 预定义变量：是Bash中已经定义好的变量，变量名不能自定义，变量作用也是固定的

## 字符串

字符串是shell编程中最常用最有用的数据类型，字符串可以用单引号，也可以用双引号，也可以不用引号。

1. 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独一个的单引号（对单引号使用转义符后也不行），但可成对出现，作为字符串拼接使用。

1. your\_name="runoob"
2. str="Hello, I know you are \"$your\_name\"! \n"
3. echo -e $str

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

获取字符串长度：

1. string="abcd"
2. echo ${#string}
3. #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

1. string="runoob is a great site"
2. echo ${string:1:4}
3. # 输出 unoo

查找子字符串：查找字符 i 或 o 的位置(哪个字母先出现就计算哪个)

1. string="runoob is a great site"
2. echo `expr index "$string" io`
3. # 输出 4

以上脚本中 ` 是反引号，而不是单引号 '。

## 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。类似于 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。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)

读取数组元素值的一般格式是：

${数组名[下标]}

使用 @ 符号可以获取数组中的所有元素，例如：

echo ${array\_name[@]}

获取数组长度的方法与获取字符串长度的方法相同，例如：

1. # 取得数组元素的个数
2. length=${#array\_name[@]}
3. # 或者
4. length=${#array\_name[\*]}
5. # 取得数组单个元素的长度
6. lengthn=${#array\_name[n]}

## 参数传递

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……

1. echo "Shell 传递参数实例！";
2. echo "执行的文件名：$0";
3. echo "第一个参数为：$1";
4. echo "第二个参数为：$2";
5. echo "第三个参数为：$3";
7. #输出结果为
8. #Shell 传递参数实例！
9. #执行的文件名：./transmit.sh
10. #第一个参数为：1
11. #第二个参数为：2
12. #第三个参数为：3

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 |
| $\* | 以一个单字符串显示所有向脚本传递的参数。 如"$\*"用「"」括起来的情况、以"$1 $2 … $n"的形式输出所有参数。 |
| $$ | 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。 如"$@"用「"」括起来的情况、以"$1" "$2" … "$n" 的形式输出所有参数。 |
| $- | 显示Shell使用的当前选项，与[set命令](https://www.runoob.com/linux/linux-comm-set.html)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。 |

$\* 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）。

1. echo "-- \$\* 演示 ---"
2. **for** i in "$\*"; **do**
3. echo $i
4. done
6. echo "-- \$@ 演示 ---"
7. **for** i in "$@"; **do**
8. echo $i
9. done
11. #输出结果为
12. -- $\* 演示 ---
13. 1 2 3
14. -- $@ 演示 ---
15. 1
16. 2
17. 3

## 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。

1. val=`expr 2 + 2`
2. echo "两数之和为 : $val"

* 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，注意使用的是反引号 ` 而不是单引号 '

### 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 | `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 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。 |

1. a=10
2. b=20
4. val=`expr $a + $b`
5. echo "a + b : $val"
7. val=`expr $a - $b`
8. echo "a - b : $val"
10. val=`expr $a \\* $b`
11. echo "a \* b : $val"
13. val=`expr $b / $a`
14. echo "b / a : $val"
16. val=`expr $b % $a`
17. echo "b % a : $val"
19. **if** [ $a == $b ]
20. then
21. echo "a 等于 b"
22. fi
23. **if** [ $a != $b ]
24. then
25. echo "a 不等于 b"
26. fi
28. a + b : 30
29. a - b : -10
30. a \* b : 200
31. b / a : 2
32. b % a : 0
33. 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；
* 在 MAC 中 shell 的 expr 语法是：$((表达式))，此处表达式中的 "\*" 不需要转义符号 "\" 。

### 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 | 检测两个数是否不相等，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于等于右边的，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。 |

1. a=10
2. b=20
4. **if** [ $a -eq $b ]
5. then
6. echo "$a -eq $b : a 等于 b"
7. **else**
8. echo "$a -eq $b: a 不等于 b"
9. fi
10. **if** [ $a -ne $b ]
11. then
12. echo "$a -ne $b: a 不等于 b"
13. **else**
14. echo "$a -ne $b : a 等于 b"
15. fi
16. **if** [ $a -gt $b ]
17. then
18. echo "$a -gt $b: a 大于 b"
19. **else**
20. echo "$a -gt $b: a 不大于 b"
21. fi
22. **if** [ $a -lt $b ]
23. then
24. echo "$a -lt $b: a 小于 b"
25. **else**
26. echo "$a -lt $b: a 不小于 b"
27. fi
28. **if** [ $a -ge $b ]
29. then
30. echo "$a -ge $b: a 大于或等于 b"
31. **else**
32. echo "$a -ge $b: a 小于 b"
33. fi
34. **if** [ $a -le $b ]
35. then
36. echo "$a -le $b: a 小于或等于 b"
37. **else**
38. echo "$a -le $b: a 大于 b"
39. fi
41. 10 -eq 20: a 不等于 b
42. 10 -ne 20: a 不等于 b
43. 10 -gt 20: a 不大于 b
44. 10 -lt 20: a 小于 b
45. 10 -ge 20: a 小于 b
46. 10 -le 20: a 小于或等于 b

### 逻辑运算符

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 |

1. a=10
2. b=20
4. **if** [[ $a -lt 100 && $b -gt 100 ]]
5. then
6. echo "返回 true"
7. **else**
8. echo "返回 false"
9. fi
11. **if** [[ $a -lt 100 || $b -gt 100 ]]
12. then
13. echo "返回 true"
14. **else**
15. echo "返回 false"
16. fi
18. 返回 **false**
19. 返回 **true**

### 字符串运算符

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不相等，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 $a ] 返回 false。 |
| -n | 检测字符串长度是否不为 0，不为 0 返回 true。 | [ -n "$a" ] 返回 true。 |
| $ | 检测字符串是否为空，不为空返回 true。 | [ $a ] 返回 true。 |

1. a="abc"
2. b="efg"
4. **if** [ $a = $b ]
5. then
6. echo "$a = $b : a 等于 b"
7. **else**
8. echo "$a = $b: a 不等于 b"
9. fi
10. **if** [ $a != $b ]
11. then
12. echo "$a != $b : a 不等于 b"
13. **else**
14. echo "$a != $b: a 等于 b"
15. fi
16. **if** [ -z $a ]
17. then
18. echo "-z $a : 字符串长度为 0"
19. **else**
20. echo "-z $a : 字符串长度不为 0"
21. fi
22. **if** [ -n "$a" ]
23. then
24. echo "-n $a : 字符串长度不为 0"
25. **else**
26. echo "-n $a : 字符串长度为 0"
27. fi
28. **if** [ $a ]
29. then
30. echo "$a : 字符串不为空"
31. **else**
32. echo "$a : 字符串为空"
33. fi
35. abc = efg: a 不等于 b
36. abc != efg : a 不等于 b
37. -z abc : 字符串长度不为 0
38. -n abc : 字符串长度不为 0
39. 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** | **举例** |
| -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。 | [ -c $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 | 检测文件是否是有名管道，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。 |

其他检查符：

* -S: 判断某文件是否 socket。
* -L: 检测文件是否存在并且是一个符号链接。

1. file="/var/www/runoob/test.sh"
2. **if** [ -r $file ]
3. then
4. echo "文件可读"
5. **else**
6. echo "文件不可读"
7. fi
8. **if** [ -w $file ]
9. then
10. echo "文件可写"
11. **else**
12. echo "文件不可写"
13. fi
14. **if** [ -x $file ]
15. then
16. echo "文件可执行"
17. **else**
18. echo "文件不可执行"
19. fi
20. **if** [ -f $file ]
21. then
22. echo "文件为普通文件"
23. **else**
24. echo "文件为特殊文件"
25. fi
26. **if** [ -d $file ]
27. then
28. echo "文件是个目录"
29. **else**
30. echo "文件不是个目录"
31. fi
32. **if** [ -s $file ]
33. then
34. echo "文件不为空"
35. **else**
36. echo "文件为空"
37. fi
38. **if** [ -e $file ]
39. then
40. echo "文件存在"
41. **else**
42. echo "文件不存在"
43. fi
45. 文件可读
46. 文件可写
47. 文件可执行
48. 文件为普通文件
49. 文件不是个目录
50. 文件不为空
51. 文件存在

## 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

echo 输出默认自带换行符，常用参数:

* -n：不要在最后自动换行
* -e：若字符串中出现以下字符，则特别加以处理，而不会将它当成一般文字输出（允许解释加反斜线的转义字符）
* \a：发出警告声
* \b：删除前一个字符，相当于退格（backspace）键，前提是 \b 后存在字符
* \c：不换行输出，最后不加上换行符号，相当于 echo -n
* \f：换行但光标仍旧停留在原来的位置
* \n：换行且光标移至行首
* \r：光标移至行首，但不换行，相当于使用"\r"以后的字符覆盖"\r"之前同等长度的字符;但是当"’\r"后面不存在任何字符时，" \r"前面的字符不会被覆盖
* \t：插入tab
* \v：与 \f 相同
* \：表示插入\字符本身

1. echo 123
2. echo -n 123
3. echo "123\b\b456"
4. echo -e "1234\b456"
5. echo -e "123\n456"

## printf 命令

printf 命令模仿 C 程序库（library）里的 printf() 程序。printf 由 POSIX 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。默认的 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...]

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

1. printf "%-10s %-8s %-4s\n" 姓名 性别 体重kg
2. printf "%-10s %-8s %-4.2f\n" 郭靖 男 66.1234
3. printf "%-10s %-8s %-4.2f\n" 杨过 男 48.6543
4. printf "%-10s %-8s %-4.2f\n" 郭芙 女 47.9876
6. #姓名     性别   体重kg
7. #郭靖     男      66.12
8. #杨过     男      48.65
9. #郭芙     女      47.99

* %s %c %d %f 都是格式替代符，％s 输出一个字符串，％d 整型输出，％c 输出一个字符，％f 输出实数，以小数形式输出。
* %-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。

1. # format-string为双引号
2. printf "%d %s\n" 1 "abc"
4. # 单引号与双引号效果一样
5. printf '%d %s\n' 1 "abc"
7. # 没有引号也可以输出
8. printf %s abcdef
10. # 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
11. printf %s abc def
13. printf "%s\n" abc def
15. printf "%s %s %s\n" a b c d e f g h i j
17. # 如果没有 arguments，那么 %s 用NULL代替，%d 用 0 代替
18. printf "%s and %d \n"

转义序列表格：

|  |  |
| --- | --- |
| **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格式指示符控制下的参数字符串中有效），而且，任何留在参数里的字符、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 |
| \0ddd | 表示1到3位的八进制值字符 |

## test 命令

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。

|  |  |
| --- | --- |
| **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 |

1. num1=100
2. num2=100
3. **if** test $[num1] -eq $[num2]
4. then
5. echo '两个数相等！'
6. **else**
7. echo '两个数不相等！'
8. fi

代码中的 [ ] 执行基本的算数运算，如：

1. #!/bin/bash
2. a=5
3. b=6
4. result=$[a+b] # 注意等号两边不能有空格
5. echo "result 为： $result"

### 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 |

1. num1="ru1noob"
2. num2="runoob"
3. **if** test $num1 = $num2
4. then
5. echo '两个字符串相等!'
6. **else**
7. echo '两个字符串不相等!'
8. fi

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

1. cd /bin
2. **if** test -e ./bash
3. then
4. echo '文件已存在!'
5. **else**
6. echo '文件不存在!'
7. fi

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。

1. cd /bin
2. **if** test -e ./notFile -o -e ./bash
3. then
4. echo '至少有一个文件存在!'
5. **else**
6. echo '两个文件都不存在'
7. fi

## 流程控制

和 Java、PHP等语言不一样，sh的流程控制不可为空

### if else

if 语句语法格式：

if condition

then

command1

command2

...

commandN

elif condition2

then

command2

else

command

fi

1. a=10
2. b=20
3. **if** [ $a == $b ]
4. then
5. echo "a 等于 b"
6. elif [ $a -gt $b ]
7. then
8. echo "a 大于 b"
9. elif [ $a -lt $b ]
10. then
11. echo "a 小于 b"
12. **else**
13. echo "没有符合的条件"
14. fi

if else 语句经常与 test 命令结合使用，如下所示：

1. num1=$[2\*3]
2. num2=$[1+5]
3. **if** test $[num1] -eq $[num2]
4. then
5. echo '两个数字相等!'
6. **else**
7. echo '两个数字不相等!'
8. fi

### 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

1. **for** loop in 1 2 3 4 5
2. **do**
3. echo "The value is: $loop"
4. done

### 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。其语法格式为：

while condition

do

command

done

1. #!/bin/bash
2. **int**=1
3. **while**(( $**int**<=5 ))
4. **do**
5. echo $**int**
6. let "int++"
7. done

### until 循环

until 循环执行一系列命令直至条件为 true 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

1. a=0
2. until [ ! $a -lt 10 ]
3. **do**
4. echo $a
5. a=`expr $a + 1`
6. done

### case ... esac

case ... esac 为多选择语句，与其他语言中的 switch ... case 语句类似，是一种多分支选择结构，每个 case 分支用右圆括号开始，用两个分号 ;; 表示 break，即执行结束，跳出整个 case ... esac 语句，esac（就是 case 反过来）作为结束标记。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 ... esac 语法格式如下：

case 值 in

模式1)

command1

command2

...

commandN

;;

模式2)

command1

command2

...

commandN

;;

esac

1. echo '输入 1 到 4 之间的数字:'
2. echo '你输入的数字为:'
3. read aNum
4. **case** $aNum in
5. 1)  echo '你选择了 1'
6. ;;
7. 2)  echo '你选择了 2'
8. ;;
9. 3)  echo '你选择了 3'
10. ;;
11. 4)  echo '你选择了 4'
12. ;;
13. \*)  echo '你没有输入 1 到 4 之间的数字'
14. ;;
15. esac

### 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）。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。

1. #!/bin/bash
2. **while** :
3. **do**
4. echo -n "输入 1 到 5 之间的数字:"
5. read aNum
6. **case** $aNum in
7. 1|2|3|4|5) echo "你输入的数字为 $aNum!"
8. ;;
9. \*) echo "你输入的数字不是 1 到 5 之间的! 游戏结束"
10. **break**
11. ;;
12. esac
13. done

### 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。

1. #!/bin/bash
2. **while** :
3. **do**
4. echo -n "输入 1 到 5 之间的数字: "
5. read aNum
6. **case** $aNum in
7. 1|2|3|4|5) echo "你输入的数字为 $aNum!"
8. ;;
9. \*) echo "你输入的数字不是 1 到 5 之间的!"
10. **continue**
11. echo "游戏结束"
12. ;;
13. esac
14. done

## 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;]

}

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

定义了一个函数并进行调用：

1. demoFun(){
2. echo "这是我的第一个 shell 函数!"
3. }
4. echo "-----函数开始执行-----"
5. demoFun
6. echo "-----函数执行完毕-----"
8. funWithReturn(){
9. echo "这个函数会对输入的两个数字进行相加运算..."
10. echo "输入第一个数字: "
11. read aNum
12. echo "输入第二个数字: "
13. read anotherNum
14. echo "两个数字分别为 $aNum 和 $anotherNum !"
15. **return** $(($aNum+$anotherNum))
16. }
17. funWithReturn
18. echo "输入的两个数字之和为 $? !"

函数返回值在调用该函数后通过 $? 来获得。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。

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：

1. funWithParam(){
2. echo "第一个参数为 $1 !"
3. echo "第二个参数为 $2 !"
4. echo "第十个参数为 $10 !"
5. echo "第十个参数为 ${10} !"
6. echo "第十一个参数为 ${11} !"
7. echo "参数总数有 $# 个!"
8. echo "作为一个字符串输出所有参数 $\* !"
9. }
10. funWithParam 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。还有几个特殊字符用来处理参数：

|  |  |
| --- | --- |
| **参数处理** | **说明** |
| $# | 传递到脚本或函数的参数个数 |
| $\* | 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。 |

# 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。 |

## 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

这个命令执行command1然后将输出的内容存入file1。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，使用>>操作符。

1. who > users
3. echo "菜鸟教程：www.runoob.com" > users
5. echo "菜鸟教程：www.runoob.com" >> users

## 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

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(<)。

1. $ wc -l users
2. 2 users
4. $  wc -l < users
5. 2

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。

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

* 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

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)。

## 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。

* 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。

1. $ wc -l << EOF
2. 欢迎来到
3. 菜鸟教程
4. www.runoob.com
5. EOF
6. 3          # 输出结果为 3 行

## /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）。这里的 2 和 > 之间不可以有空格，2> 是一体的时候才表示错误输出。

# 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

1. url="http://www.runoob.com"
2. ---------------------------------
3. . ./test1.sh
4. # 或者使用以下包含文件代码
5. # source ./test1.sh
6. echo "菜鸟教程官网地址：$url"

# shell实例

## 监听端口

#!/bin/bash

####################################################

#检测服务器是否开启web服务(监听80)，并判断服务种类#

###################################################

#捕获监听80端口的服务名称

netstat -lntp|grep ":80"|awk -F '/' '{print $2}'|sort|uniq > /tmp/80.log

n=`wc -l /tmp/80.log|awk '{print $1}'`

s1=`cat /tmp/80.log`

#判断服务种类函数

judge\_server(){

case $1 in

httpd)

echo "this server is apache"

;;

nginx)

echo "this server is nginx"

;;

tomcat)

echo "this server is tomcat"

;;

\*)

echo "this server is other"

;;

esac

}

if [ $n -eq 0 ]

then

echo "no web server!"

exit

fi

if [ $n -eq 1 ]

then

judge\_server $s1

else

echo "there are $n web servers"

for web in `cat /tmp/80.log`

judge\_server $web

fi

## 计算内存

#!/bin/bash

#########################################

#计算一下linux系统所有进程占用内存大小的和

#########################################

count=0

# 这个循环会遍历出每个进程占用的内存大小

for i in `ps aux |awk '{print $6}' |grep -v 'RSS'`

do

# 将遍历出来的数字进行累加

count=$[$count+$i]

done

# 就得到所有进程占用内存大小的和了

echo "$count/kb"