目　录

致谢

当前文档 《Shell脚本学习系列教程》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-03-01。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/shell-book

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

shell-book

Shell脚本学习

Shell是一种脚本语言，那么，就必须有解释器来执行这些脚本。

Unix/Linux上常见的Shell脚本解释器有bash、sh、csh、ksh等，习惯上把它们称作一种Shell。我们常说有多少种Shell，其实说的是Shell脚本解释器。

点击右上角的 **Watch** 订阅本书，点击 Star 收藏本书。

- 论坛

目录

开始阅读: Shell脚本学习–入门

打赏作者

欢迎微信扫码打赏我，感谢支持！

01- Shell脚本学习--入门

01- Shell脚本学习—入门

标签： Shell

简介

> *Shell*是一种脚本语言，那么，就必须有解释器来执行这些脚本。

Unix/Linux上常见的Shell脚本解释器有bash、sh、csh、ksh等，习惯上把它们称作一种Shell。我们常说有多少种Shell，其实说的是Shell脚本解释器。

Hello World

打开文本编辑器，新建一个文件 `test.sh` ，扩展名为 `.sh` （sh代表shell）。

输入一些代码：

```
1. #!/bin/bash
2. echo "Hello World !"
```

在命令行运行：

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

运行结果：

```
1. 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。

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。

```
1. # --------
2. # 这是注释块
3. # --------
```

打印输出

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。

```
1. echo arg
2. echo -e arg #执行arg里的转义字符。echo加了-e默认会换行
3. echo arg > myfile #显示结果重定向至文件，会生成myfile文件
```

注意，echo后单引号和双引号作用是不同的。单引号不能转义里面的字符。双引号可有可无，单引号主要用在原样输出中。

printf： 格式化输出语句。

`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。

如同 `echo` 命令， `printf` 命令也可以输出简单的字符串：

```
1. printf "hello\n"
```

`printf` 不像 `echo` 那样会自动换行，必须显式添加换行符(\n)。

注意： `printf` 由 POSIX 标准所定义，移植性要比 `echo` 好。

printf 命令的语法：

```
1. printf  format-string  [arguments...]
2.
3. #format-string 为格式控制字符串，arguments 为参数列表。功能和用法与c语言的 printf 命令类似。
```

这里仅说明与C语言printf()函数的不同：

- printf 命令不用加括号
- format-string 可以没有引号，但最好加上，单引号双引号均可。

- 参数比格式控制符(%)多时，格式控制符可以重用，可以将所有参数都转换。
- arguments 使用空格分隔，不用逗号。

```
# 双引号
printf "%d %s\n" 10 "abc"
10 abc
# 单引号与双引号效果一样
printf '%d %s\n' 10 "abc"
10 abc

# 没有引号也可以输出
printf %s abc
abc

# 但是下面的会出错：
printf %d %s 10 abc
#因为系统分不清楚哪个是参数，这时候最好加引号了。


# 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 a b c
abc
$ printf "%s\n" a b c
a
b
c

# 如果没有 arguments，那么 %s 用NULL代替，%d 用 0 代替
$ printf "%s and %d \n"
and 0

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

```
33. $
```

read: 命令行从输入设备读入内容

```
1. #!/bin/bash
2. 
3. # Author : lalal
4. 
5. echo "What is your name?"
6. read NAME #输入
7. echo "Hello, $NAME"
```

运行脚本:

```
1. chmod +x test.sh
2. ./test.sh
3. 
4. What is your name?
5. lalal
6. 
7. Hello, lalal
```

变量定义

Shell支持自定义变量。

定义变量

定义变量时，变量名不加美元符号（$），如:

```
1. variableName="value"
```

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。有空格会出错。

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）。

变量定义举例：

```
1. myUrl="lalal"
2. myNum=100
```

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：

```
1. your_name="lalal"
2. echo $your_name
3. 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：

```
1. for skill in C PHP Python Java
2. do
3.     echo "I am good at ${skill}Script"
4. 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。

在变量前面加 `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。

```
1. url="http://www.baidu.com"
2. readonly url
3. url="http://www.baidu.com"
```

使用 `unset` 命令可以删除变量。语法：

```
1. 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。

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。

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。

变量	含义
`$0`	当前脚本的文件名
`$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。被双引号(“ “)包含时，与 `$*` 稍有不同
`$?`	上个命令的退出状态，或函数的返回值。
`$$`	当前Shell进程ID。对于 Shell 脚本，就是这些脚本所在的进程ID。

示例：

```
#!/bin/bash
echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

运行结果：

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

`$*` 和 `$@` 的区别

`$*` 和 `$@` 都表示传递给函数或脚本的所有参数，不被双引号(“

“)包含时，都以 `"$1" "$2" … "$n"` 的形式输出所有参数。

但是当它们被双引号(“ “)包含时，” `$*` “ 会将所有的参数作为一个整体，以” `$1 $2 … $n` “的形式输出所有参数；” `$@` “ 会将各个参数分开，以 `"$1" "$2" … "$n"` 的形式输出所有参数。

示例：

```
#!/bin/bash
echo "\$*=" $*
echo "\"\$*\"=" "$*"
echo "\$@=" $@
echo "\"\$@\"=" "$@"
echo "print each param from \$*"
for var in $*
do
    echo "$var"
done
echo "print each param from \$@"
for var in $@
do
    echo "$var"
done
echo "print each param from \"\$*\""
for var in "$*"
do
    echo "$var"
done
echo "print each param from \"\$@\""
for var in "$@"
do
    echo "$var"
done
```

执行 `./test.sh "a" "b" "c" "d"` ，看到下面的结果：

```
$*=  a b c d
```

```
"$*"= a b c d
$@=  a b c d
"$@"= a b c d
print each param from $*
a
b
c
d
print each param from $@
a
b
c
d
print each param from "$*"
a b c d
print each param from "$@"
a
b
c
d
```

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。

示例：

```
if [[ $? != 0 ]];then
  echo "error"
  exit 1;
fi
```

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。

转义字符

```
转义字符    含义
\\    反斜杠
\a    警报，响铃
\b    退格（删除键）
\f    换页(FF)，将当前位置移到下页开头
\n    换行
\r    回车
\t    水平制表符（tab键）
\v    垂直制表符
```

shell默认是不转义上面的字符的。需要加 `-e` 选项。

举个例子：

```
#!/bin/bash
a=11
echo -e "a is $a \n"
```

运行结果：

```
Value of a is 10
```

这里 `-e` 表示对转义字符进行替换。如果不使用 `-e` 选项，将会原样输出：

```
Value of a is 10\n
```

可以使用 echo 命令的 `-E` 选项禁止转义，默认也是不转义的；使用 `-n` 选项可以禁止插入换行符。

命令替换

命令替换是指Shell可以先执行命令，将输出结果暂时保存，在适当的地方输出。

语法：

```
1.  `command`
```

注意是反引号，不是单引号，这个键位于 **Esc** 键下方。

下面的例子中，将命令执行结果保存在变量中：

```
1.  #!/bin/bash
2.  DATE=`date`
3.  echo "Date is $DATE"
```

变量替换

变量替换可以根据变量的状态（是否为空、是否定义等）来改变它的值。

可以使用的变量替换形式：

形式	说明
`${var}`	变量本来的值
`${var:-word}`	如果变量 `var` 为空或已被删除(unset)，那么返回 word，但不改变 `var` 的值。
`${var:=word}`	如果变量 `var` 为空或已被删除(unset)，那么返回 word，并将 `var` 的值设置为 word。
`${var:?message}`	如果变量 `var` 为空或已被删除(unset)，那么将消息 message 送到标准错误输出，可以用来检测变量 `var` 是否可以被正常赋值。若此替换出现在Shell脚本中，那么脚本将停止运行。
`${var:+word}`	如果变量 `var` 被定义，那么返回 word，但不改变 var 的值。

一个完整的shell示例

下面的脚本用于php安装过程中安装zip扩展。

`php_zip_ins.sh`

```
#!/bin/bash
#zip install

if [ -d php-5.4.25/ext/zip ];then
    cd php-5.4.25/ext/zip
else
    tar zxvf php-5.4.25.tar.gz
    cd php-5.4.25/ext/zip
fi
/usr/local/php/bin/phpize
./configure --with-php-config=/usr/local/php/bin/php-config
make
[ $? != 0 ] && exit
make install
echo
grep 'no-debug-zts-20100525' /usr/local/php/etc/php.ini
if [ $? != 0 ];then
        echo '' >> /usr/local/php/etc/php.ini
        echo 'extension_dir=/usr/local/php/lib/php/extensions/no-debug-zts-20100525' >> /usr/local/php/etc/php.ini
fi
grep 'zip.so' /usr/local/php/etc/php.ini
if [ $? != 0 ];then
    echo 'extension=zip.so' >> /usr/local/php/etc/php.ini
fi
echo "zip install is OK"


/usr/local/apache2/bin/apachectl restart
cd -
rm -rf php-5.4.25
echo "all ok!"
```

```
32.  ls /usr/local/php/lib/php/extensions/no-debug-zts-20100525/
```

02- Shell脚本学习--运算符

02- Shell脚本学习—运算符

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。

算术运算符

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。

```
# 命令行直接计算
expr 2 + 2   #4
expr 3 - 2   #1
expr 3 / 2   #1
expr 3 \* 2   #6

# 使用表达式
a=10
```

```
b=20
val=`expr $a + $b`
echo "a + b : $val"
```

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。
- 乘号(*)前边必须加反斜杠()才能实现乘法运算
- 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。

算术运算符列表

```
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 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。
```

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。

```
#!/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
```

缩成一行可以这样：

```
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
```

这里缩写，主要是为了让大家注意：

- if后面直到then前面的分号结束，都是有空格的：`if [ $a -eq $b ]`

关系运算符列表

```
运算符    说明
-eq    检测两个数是否相等，相等返回 true。同算数运算符`==`
-ne    检测两个数是否相等，不相等返回 true
-gt    检测左边的数是否大于右边的，如果是，则返回 true。
-lt    检测左边的数是否小于右边的，如果是，则返回 true。
-ge    检测左边的数是否大等于右边的，如果是，则返回 true。
-le    检测左边的数是否小于等于右边的，如果是，则返回 true。
```

布尔运算符

布尔运算符列表

```
运算符    说明
!    非运算，表达式为 true 则返回 false，否则返回 true。
-o    或运算(or)，有一个表达式为 true 则返回 true。
-a    与运算(and)，两个表达式都为 true 才返回 true。
```

```
if [ 3 -eq 3 -a 3 -lt 5 ]
```

```
2. then
3.     echo 'ok'
4. fi;
```

字符串运算符

字符串运算符列表

```
1. 运算符    说明    举例
2. =    检测两个字符串是否相等，相等返回 true。    [ $a = $b ] 返回 false。
3. !=    检测两个字符串是否相等，不相等返回 true。    [ $a != $b ] 返回
   true。
4. -z    检测字符串长度是否为0，为0返回 true。    [ -z $a ] 返回 false。
5. -n    检测字符串长度是否为0，不为0返回 true。    [ -n $a ] 返回 true。
6. str    检测字符串是否为空，不为空返回 true。    [ $a ] 返回 true。
```

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。

```
1. #!/bin/sh
2. file="/tmp/test.sh"
3.
4. if [ -e $file ]
5. then
6.    echo "File exists"
7. else
8.    echo "File does not exist"
9. fi
```

文件测试运算符列表

```
1. 操作符    说明    举例
2.
3. -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。
```

03- Shell脚本学习--字符串和数组

03- Shell脚本学习—字符串和数组

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：

单双引号的区别：

- 双引号里可以有变量，单引号则原样输出；
- 双引号里可以出现转义字符，单引号则原样输出；
- 单引号字串中不能出现单引号。

拼接字符串

```
#!/bin/bash

str1='i'
str2='love'
```

```
str3='you'

echo $str1 $str2 $str3
echo $str1$str2$str3
echo $str1,$str2,$str3
```

输出：

```
i love you
iloveyou
i,love,you
```

获取字符串长度

```
#!/bin/bash/

str='i love you'

echo ${#str}

# 输出：10
```

截取字符串

```
#!/bin/bash/

str='i love you'

echo ${str:1} # 从第1个截取到末尾。注意从0开始。
echo ${str:2:2} # 从第2个截取2个。
echo ${str:0} # 全部截取。
echo ${str:-3} # 负数无效，视为0。
```

输出：

```
love you
lo
i love you
i love you
```

查找字符串

```
#!/bin/bash/

str="i love you"

echo `expr index "$str" l`
echo `expr index "$str" love` #最后一个参数是字符，字符串只保留首字母
echo `expr index "$str" o`
echo `expr length "$str"` #字符串长度
echo `expr substr "$str" 1 6` #从字符串中位置1开始截取6个字符。索引是从0开始的。
```

输出：

```
3
3
4
10
i love
```

注意字符串变量需要加双引号。

*拓展：`expr`更多关于字符串用法：

```
STRING : REGEXP   anchored pattern match of REGEXP in STRING

match STRING REGEXP        same as STRING : REGEXP

substr STRING POS LENGTH   #从STRING中POS位置开始截取LENGTH个字符。POS索引是从1开始的。
```

```
6.
7.  index STRING CHARS          #在STRING中查找字符CHARS首次出现的位置，没有找
    到返回0
8.
9.  length STRING               #字符串长度
```

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。

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：

```
1.  array_name=(value1 value2 ... valuen)
```

例如：

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

或者

```
1.  array_name=(
2.      value0
3.      value1
4.      value2
5.      value3
6.  )
```

还可以单独定义数组的各个分量：

```
1.  array_name[0]=value0
2.  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。

下面来读取数组：

```
echo ${array_name[2]} #读取下标为2的元素
echo ${array_name[*]} #读取所有元素
echo ${array_name[@]} #读取所有元素


echo ${#array_name[*]} #获取数组长度
echo ${#array_name[@]} #获取数组长度
echo ${#array_name[1]} #获取数组中单个元素的长度
```

输出：

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

总结

对比shell里字符串和数组，我们发现：

字符串

```
str="hello"
${#str} # 读取字符串长度
echo ${str} # 读取字符串全部
echo ${str:1} # 截取字符串
```

数组:

```
arr=(a1,a2,a3)
${#str[*]} # 读取数组长度
${#str[1]} # 读取数组某个元素长度

echo ${str[*]} # 读取数组全部
echo ${str[1]} # 读取数组某个元素
```

`${#ele*}` 用来读取ele元素长度属性

`${ele*}` 用来读取或操作ele元素

04- Shell脚本学习--条件控制

04- Shell脚本学习—条件控制

标签： Shell

条件判断：if语句

语法格式：

```
1. if [ expression ]
2. then
3.    Statement(s) to be executed if expression is true
4. fi
```

注意：`expression` 和方括号([ ])之间必须有空格，否则会有语法错误。

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 语句
2. if ... else ... fi 语句
3. if ... elif ... else ... fi 语句
```

示例：

```
1. #!/bin/bash/
2.
3. a=10
4. b=20
5. if [ $a == $b ]
6. then
7.     echo "a is equal to b"
8. elif [ $a -gt $b ]
9. then
10.    echo "a is greater to b"
11. else
12.    echo "a is less to b"
13. fi
```

`if ... else` 语句也可以写成一行，以命令的方式来运行：

```
1. a=10;b=20;if [ $a == $b ];then echo "a is equal to b";else echo "a
   is not equal to b";fi;
```

`if ... else` 语句也经常与 `test` 命令结合使用，作用与上面一样：

```
1. #!/bin/bash/
2.
3. a=10
4. b=20
5. if test $a == $b
6. then
7.     echo "a is equal to b"
8. else
```

```
    echo "a is not equal to b"
fi
```

分支控制：case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。

示例：

```
#!/bin/bash/

grade="B"

case $grade in
    "A") echo "Very Good!";;
    "B") echo "Good!";;
    "C") echo "Come On!";;
    *)
        echo "You Must Try!"
        echo "Sorry!";;
esac
```

转换成C语言是：

```
#include <stdio.h>
int main(){
    char grade = 'B';
    switch(grade){
        case 'A': printf("Very Good!");break;
        case 'B': printf("Very Good!");break;
        case 'C': printf("Very Good!");break;
        default:
            printf("You Must Try!");
            printf("Sorry!");
            break;
```

```
    }
    return 0;
}
```

对比看就很容易理解了。很相似，只是格式不一样。

需要注意的是：
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。

再举一个例子：

```
#!/bin/bash
option="${1}"
case ${option} in
   "-f") FILE="${2}"
      echo "File name is $FILE"
      ;;
   "-d") DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      exit 1 # Command to come out of the program with status 1
      ;;
esac
```

运行结果：

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]

./test.sh -f index.html
File name is index.html
```

这里用到了特殊变量 `${1}`，指的是获取命令行的第一个参数。

for循环

shell的for循环与c、php等语言不同，同Python很类似。下面是语法格式：

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

示例：

```
#!/bin/bash/

for value in 1 2 3 4 5
do
    echo "The value is $value"
done
```

输出：

```
The value is 1
The value is 2
The value is 3
```

```
4. The value is 4
5. The value is 5
```

顺序输出字符串中的字符：

```
1. for str in 'This is a string'
2. do
3.     echo $str
4. done
```

运行结果：

```
1. This is a string
```

遍历目录下的文件：

```
1. #!/bin/bash
2. for FILE in *
3. do
4.    echo $FILE
5. done
```

上面的代码将遍历当前目录下所有的文件。在Linux下，可以改为其他目录试试。

遍历文件内容：

city.txt

```
1. beijing
2. tianjin
3. shanghai
```

```
1. #!/bin/bash
2.
3. citys=`cat city.txt`
```

```
for city in $citys
   echo $city
done
```

输出：

```
beijing
tianjin
shanghai
```

while循环

只要while后面的条件满足，就一直执行do里面的代码块。

其格式为：

```
while command
do
   Statement(s) to be executed if command is true
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。

示例：

```
#!/bin/bash

c=0;
while [ $c -lt 3 ]
do
    echo "Value c is $c"
    c=`expr $c + 1`
done
```

输出：

```
Value c is 0
Value c is 1
Value c is 2
```

这里由于shell本身不支持算数运算，所以使用 `expr` 命令进行自增。

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。

将上面while循环的例子改改，就能达到一样的效果：

```
#!/bin/bash

c=0;
until [ $c -eq 3 ]
do
    echo "Value c is $c"
    c=`expr $c + 1`
done
```

首先do里面的语句块一直在运行，直到满足了until的条件就停止。

输出：

```
Value c is 0
Value c is 1
Value c is 2
```

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，像

大多数编程语言一样，Shell也使用 break 和 continue 来跳出循环。

break

break命令允许跳出所有循环（终止执行后面的所有循环）。

```
#!/bin/bash

i=0
while [ $i -lt 5 ]
do
    i=`expr $i + 1`

    if [ $i == 3 ]
        then
            break
    fi
    echo -e $i
done
```

运行结果：

```
1
2
```

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。

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。

```
#!/bin/bash

i=0
while [ $i -lt 5 ]
do
    i=`expr $i + 1`

    if [ $i == 3 ]
        then
            continue
    fi
    echo -e $i

done
```

运行结果：

```
1
2
4
5
```

05- Shell脚本学习--函数

05- Shell脚本学习—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。

函数定义

Shell 函数的定义格式如下：

```
function function_name () {
    list of commands
    [ return value ]
}
```

其中 `function` 关键字是可选的。

```
#!/bin/bash

hello(){
    echo 'hello';
}

hello
```

运行结果：

```
hello
```

调用函数只需要给出函数名，不需要加括号。

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`。

```
#!/bin/bash

function hello(){
    return 'hello';
}

hello
```

运行结果：

```
line 4: return: hello: 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。

```
#!/bin/bash

function hello(){
    return 'hello';
}

str=hello

echo $str
```

运行结果：

```
hello
```

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
```

如果你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。

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...这就是前面讲的特殊变量。

```
#!/bin/bash

function sum(){
    case $# in
        0) echo "no param";;
        1) echo $1;;
        2) echo `expr $1 + $2`;;
        3) echo `expr $1 + $2 + $3`;;
        *) echo "$# params! It's too much!";;
    esac
}

sum 1 3 5 6
```

运行结果：

```
4 params! It's too much!
```

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：

```
特殊变量    说明
$#    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。
```

06- Shell脚本学习--其它

06- Shell脚本学习—其它

Shell输入输出重定向

Unix 命令默认从标准输入设备(stdin)获取输入，将结果输出到标准输出设备(stdout)显示。一般情况下，标准输入设备就是键盘，标准输出设备就是终端，即显示器。
输出重定向

命令的输出不仅可以是显示器，还可以很容易的转移向到文件，这被称为输出重定向。

命令输出重定向的语法为：

```
1.  command > file
```

这样，输出到显示器的内容就可以被重定向到文件。

例如，下面的命令在显示器上不会看到任何输出：

```
1.  who > users
```

打开 users 文件，可以看到下面的内容：

```
cat users

oko          tty01   Sep 12 07:30
ai           tty15   Sep 12 13:32
ruth         tty21   Sep 12 10:10
pat          tty24   Sep 12 13:07
steve        tty25   Sep 12 13:03
```

输出重定向会覆盖文件内容，请看下面的例子：

```
echo line 1 > users

cat users
line 1
```

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 line 2 >> users

cat users
line 1
line 2
```

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：

```
command < file
```

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(<)。

例如，计算 users 文件中的行数，可以使用下面的命令：

```
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。

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
```

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。

全部可用的重定向命令列表：

```
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。
```

Here Document

Here Document 目前没有统一的翻译，这里暂译为 `嵌入文档` 。Here Document 是 Shell 中的一种特殊的重定向方式，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。

下面的例子，通过 `wc -l` 命令计算 document 的行数:

```
wc -l << EOF
    This is a simple lookup program
    for good (and bad) restaurants
    in Cape Town.
EOF
```

输出: 3

也可以 将 Here Document 用在脚本中，例如:

```
#!/bin/bash
cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

运行结果:

```
This is a simple lookup program
```

```
2.  for good (and bad) restaurants
3.  in Cape Town.
```

/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`：

```
1.  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：

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

这样不会在屏幕打印任何信息。

Shell文件包含

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。

示例：
被包含文件：sub.sh

```
name="yjc"
```

主文件: test.sh

```
. ./sub.sh
echo $name
```

运行结果:

```
yjc
```