Shell 编程（Linux）

> Author：Marlous（整理、编写）
>
> E-mail：goonecat@foxmail.com

> Date：2018/10/18

> 参考教程 1 参考博文 1 参考博文 2 参数博文 3

一 shell 注释模板

```
#!/bin/bash
# ------------------------------------------------------------------
# name: login.sh
# version: 1.0
# createTime: 2018/10/12
# description: shell 脚本的功能描述
# author: Marlous
# email: example@gmail.com
# github: https://github.com/test
# ------------------------------------------------------------------
```

二 运行 shell 脚本

```
cd 到该脚本目录下
chmod +x ./test.sh    #给脚本执行权限
./test.sh             #执行该脚本
```

三 shell 编程

1 变量

1. 变量赋值与引用：

给变量赋值： num=12 或 string="zifuchuan"

引用变量： ${num} 用花括号把变量包裹起来，看情况也可以不加，推荐加。

只读变量： readonly num

删除变量： unset num

```
# 系统环境变量使用
curTime=$(date "+%H%M%S")   #将时间  时 分 秒 赋给变量 curTime，注意 date 和 双引号之间有空格。
curDate=$(date "+%Y%m%d")   #将时间  年 月 日 赋给变量 curDate。
```

2. 字符串:

可以不用引号，也可以用单引号或双引号（用双引号，其中可以有变量、转义字符；不用双引号则原样输出）。

获取字符串长度： ${#string}

提取子字符串： ${string:1:4} 表示从第 2 个字符开始，长度为 4。

`expr index "$string" io` 表示查找字符 i 或 o 的位置，值为位置数字，哪个字母先出现就计算哪个（用反引号）。

3. 数组:

给数组赋值： array_name=(value0 value1 value2 value3) 或 array_name[0]=value0

读取数组元素：单个元素 ${array_name[n]} 全部值用 ${array_name[@]}

获取数组长度： ${#array_name[@]}

4. 注释:

- 单行注释： `#`
- 多行注释：

```
:<<符号


符号
```

```
# 例：EOF 也可以是 !、'（感叹号、反引号等） 。
:<<EOF


EOF
```

2 接收参数

- $0 为执行的文件名，$1 到 $n 为第一个参数到第 n 个参数。
- 其他特殊参数：

```
$#    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。
```

3 运算符

1. 算术运算：

```
加、减、乘、除、取模、赋值、相等、不相等。
+ - * / % = == !=
```

2. 关系运算符：

```
-eq    检测两个数是否相等，相等返回 true。
-ne    检测两个数是否相等，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于等于右边的，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。
```

3. 布尔（逻辑）运算符:

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

4. 逻辑运算符:

```
&&    逻辑与
||    逻辑或
```

5. 字符串运算符：

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

6. 文件测试运算符:

```
-b file     检测文件是否是块设备文件，如果是，则返回 true。    例：  [-b $file] 或 test -b $file
-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     检测文件是否是有名管道，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。
```

7. shell 中的运算小结：

```
# 用 expr 改变运算顺序
echo `expr 1+2`

# 用 let 指示数学运算
b=let 1+2

# 用 $[] 表示数学运算
echo $[1+2]
```

示例：计算 S=3(xy)+4x2+5y+6

```
s=0              # 初始化一个变量
t=`expr$1**$2`   # 用 expr 改变运算顺序，求x的y次
t=$[t*3]         # t 乘以 3
s=$[s+t]         # 结果相加
t=$[$1**2]       # 求x的平方
t=$[t*4]         # 结果乘以 4
s=$[s+t]         # 结果相加
t=`expr$2*5`     # 求5y的值
s=$[s+t]         # 结果相加
s=$[s+6]         # 结果加上 6
echo$s           # 输出结果
echo$((a%b))     # 取余
```

4 一些命令

命令详见 Linux 命令。

- read 命令：`read variable`，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。
- echo 命令：输出。
- 显示命令执行结果：用反引号括起来。如 `echo 反引号 某命令 反引号`
- printf 命令，格式化输出字符：与 C 语言相同。

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

例:
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

输出结果:
姓名      性别    体重kg
郭靖      男      66.12
杨过      男      48.65
郭芙      女      47.99
```

- test 命令：判断某个条件是否成立。使用和关系运算符相同。

```
例:

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
```

- let 命令：用于指定算术运算。

```
例:
a=3
let "a+=1"
echo "a+=1 is $a"`

b=let 1 + 2
```

- expr：
  expr 用法参考博文
  expr 命令用来计算表达式的值。

数值运算、数值或字符串比较、字符串匹配、字符串提取、字符串长度计算等功能。几个特殊功能，判断变量或参数是否为整数、是否为空、是否为 0 等。

```
用于数值计算：
# 例子：计算（2 + 3）×4 的值。（必须要有空格，乘法要在星号前有个斜杠。）
分步计算：
s=`expr 2 + 3`
expr $s \* 4
一步完成计算：
expr  `expr 2 + 3 `  \* 4

增量计数：
# 例子：
$LOOP=0
$LOOP=`expr $LOOP + 1`

数值测试：
# 将一个值赋予变量，进行运算，并将输出导入 dev/null；然后测试最后命令状态，如果为 0，证明这是一个数，其他则表明为非数值。
$rr=2
$expr $rr + 1
3

模式匹配：
# 使用 expr 通过指定冒号选项计算字符串中字符数。 .* 即任何字符重复 0 次或多次。
```

补充模式匹配处理（见 expr 命令）：

模式匹配记忆方法：

是去掉左边(在键盘上#在$之左边)

% 是去掉右边(在键盘上%在$之右边)

#和%中的单一符号是最小匹配，两个相同符号是最大匹配。

> ${var%pattern}, ${var%%pattern}, ${var#pattern}, ${var##pattern}

第一种模式：`${variable%pattern}`，这种模式时，shell在variable中查找，看它是否一给的模式pattern结尾，如果是，就从命令行把variable中的内容去掉右边最短的匹配模式

第二种模式：`${variable%%pattern}`，这种模式时，shell在variable中查找，看它是否一给的模式pattern结尾，如果是，就从命令行把variable中的内容去掉右边最长的匹配模式

第三种模式：`${variable#pattern}` 这种模式时，shell在variable中查找，看它是否一给的模式pattern开始，如果是，就从命令行把variable中的内容去掉左边最短的匹配模式

第四种模式：`${variable##pattern}` 这种模式时，shell在variable中查找，看它是否一给的模式pattern结尾，如果是，就从命令行把variable中的内容去掉右边最长的匹配模式

这四种模式中都不会改变variable的值，其中，只有在pattern中使用了*匹配符号时，%和%%，#和##才有区别。结构中的pattern支持通配符，*表示零个或多个任意字符，?表示仅与一个任意字符匹配，[...]表示匹配中括号里面的字符，[!...]表示不匹配中括号里面的字符。

补充字符串提取与替换处理（见 expr 命令）：

> ${var:num}, ${var:num1:num2}, ${var/pattern/pattern}, ${var//pattern/pattern}

第一种模式：`${var:num}`，这种模式时，shell在var中提取第num个字符到末尾的所有字符。若num为正数，从左边0处开始；若num为负数，从右边开始提取字串，但必须使用在冒号后面加空格或一个数字或整个num加上括号，如${var: -2}、${var:1-3}或${var:(-2)}。

第二种模式：`${var:num1:num2}`，num1是位置，num2是长度。表示从$var字符串的第$num1个位置开始提取长度为$num2的子串。不能为负数。

第三种模式：`${var/pattern/pattern}` 表示将var字符串的第一个匹配的pattern替换为另一个pattern。

第四种模式：`${var//pattern/pattern}` 表示将var字符串中的所有能匹配的pattern替换为另一个pattern。

5 一些符号

参考资料

- 表达式中包含了空格或其他特殊字符，则必须引起来。
- 分号：多个语句之间的分割符号，当只有一个语句（或多行）的时候，末尾无需分号。
- 一些转义字符：`\c` 为显示不换行。
- 原样输出字符串，不进行转义或取变量：用单引括起来。
- 单小括号 `()`：

命令组：
括号中的命令将会新开一个子 shell 顺序执行，所以括号中的变量不能够被脚本余下的部分使用。
括号中多个命令之间用分号隔开，最后一个命令可以没有分号。

命令替换：
等同于 `cmd`，执行命令，用输出替换原命令位置。

用于初始化数组：
如：array=(a b c d)

- 双小括号 `(())`：

整数扩展：
((exp)) 结构扩展并计算一个算术表达式的值（整型）。
表达式的结果为零，返回假或状态码 1；表达式的结果为非零，返回 true 或状态码 0。

括号中的运算符、表达式符合 C 语言运算规则，都可用在 $((exp)) 中。
如 echo $((16#5f)) 结果为 95，十六进制转十进制输出。

用 (( )) 也可重定义变量值。如 ((a++))

用于算术运算比较：
双括号中的变量可以不使用 $ 符号前缀。括号内支持多个表达式用逗号分开。
只要括号中的表达式符合 C 语言运算规则，可以直接使用。如，直接使用 if (($i<5)) 如果不使用双括号，则为 if [$i -lt 5]

- 单中括号 `[]`：

bash 内部命令：
等同于 test 命令。
[] 中关系（整数）比较只能使用 -eq 这种形式。
[] 中的逻辑与和逻辑或使用- a 和 -o 表示。
[] 中可用字符串比较运算符只有 == 和 != 两种。
无论是字符串比较还是整数比较都不支持大于号小于号；
如果实在想用，对于字符串比较可以使用转义形式，如果比较 "ab" 和 "bc" 为 [ab \< bc]，结果为真，也就是返回状态为 0。

字符范围：
用作正则表达式的一部分，test 中不能用正则。

引用数组元素：
引用数组中每个元素的编号。

- 双中括号 `[[]]`：

[[ ]] 结构比[ ]结构更加通用。
在[[ ]]之间所有的字符都不会发生文件名扩展或者单词分割，但是会发生参数扩展和命令替换。

[[ ]] 中匹配字符串或通配符，不需要引号。
字符串比较时可以把右边的作为一个模式，不仅是一个字符串，如 [[hello == hell?]]，结果为真。

使用 [[ ]] 条件判断结构，能防止脚本中的许多逻辑错误。
可直接如 C 语言，if [[$a != 1 && $a != 2]]

bash 把双中括号中的表达式看作一个单独的元素，并返回一个退出状态码。

- 大括号 `{}`：

大括号拓展：
对以逗号 ，分割的文件列表进行拓展，如 touch {a,b}.txt 结果为 a.txt b.txt
对以点点 .. 分割的顺序文件列表起拓展作用，如 touch {a..d}.txt 结果为 a.txt b.txt c.txt d.txt

代码块：
这个结构事实上创建了一个匿名函数 。
与小括号中的命令不同，大括号内的命令不会新开一个子 shell 运行，即脚本余下部分仍可使用括号内变量。
括号内的命令间用分号隔开，最后一个也必须有分号。{} 的第一个命令和左括号之间必须要有一个空格。

用来对字符串做处理。

6 符号使用小结

1. $ 后的括号：

${a} 变量 a 的值，在不引起歧义的情况下可以省略大括号。

$(cmd) 命令替换，和 `cmd` 效果相同，结果为 shell 命令 cmd 的输，
过某些 Shell 版本不支持 $() 形式的命令替换，如 tcsh。

$((expression)) 和用反引号包裹效果相同，
计算数学表达式 exp 的数值，其中 exp 只要符合 C 语言的运算规则即可，甚至三目运算符和逻辑表达式都可以计算。

2. 多条命令执行：

单小括号，(cmd1;cmd2;cmd3) 新开一个子 shell 顺序执行命令 cmd1 cmd2 cmd3
各命令之间用分号隔开，最后一个命令后可以没有分号。

单大括号，{ cmd1;cmd2;cmd3;} 在当前 shell 顺序执行命令 cmd1 cmd2 cmd3，
各命令之间用分号隔开，最后一个命令后必须有分号，第一条命令和左括号之间必须用空格隔开。

对 {} 和 () 而言，括号中的重定向符只影响该条命令， 而括号外的重定向符影响到括号中的所有命令。

7 控制结构

不可为空，else 无操作则不写。

1. 选择:（写成一行需要在条件、命令结尾加分号）

- if 选择:

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

- case 选择:

```
case 值 in
模式1)
    command1
    command2
    ...
    commandN
    ;;
模式2)
    command1
    command2
    ...
    commandN
    ;;
esac
```

2. 循环:

- for 循环:

```
# 将 item1 item2 等依次赋给变量 var。
for var in item1 item2 ... itemN
do
    command1
    command2
    ...
    commandN
done

# 将字符串中的每个字符分别赋给变量 str。
for str in 'This is a string'
do
```

```
    echo $str
done
```

- while 循环：

```
while condition
do
    command
done

# 无限循环：
while true
do
    command
done
```

- until 循环：循环执行一系列命令直至条件为 true 时停止。

```
until condition
do
    command
done
```

3. break 与 continue

8 输入输出与重定向

```
# 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）。
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。
例：
command << tag
    document
tag

----------------------------------------------------------------
重定向到 /dev/null
> /dev/null
command > /dev/null 2>&1  # 屏蔽 stdout 和 stderr
```

9 函数

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。

```
funname()
{
    action;
    return int;
}
```

10 调用脚本

- fork: 如果脚本有执行权限的话为 `path/to/test.sh`；如果没有为 `sh path/to/test.sh`，子 Shell 中执行，子从父 Shell 单向继承环境变量，执行完返回。（类似于批处理中的 start）
- source: `source path/to/test.sh`，同一个 Shell 中执行，环境变量可双向获取。（类似于批处理中的 call）
- exec: `exec path/to/test.sh`，同一个 Shell 内执行，调用一个新脚本以后, 父脚本中 exec 行之后的内容就不会再执行。

四 shell 脚本示例

```
#/bin/bash
#模拟 linux 登录 shell
echo -n "login:"
read name
echo -n "password:"
read passwd
if [ $name = "cht" -a $passwd = "abc" ];then
echo "the host and password is right!"
else echo "input is error!"
fi
```

```
#/bin/bash
#删除当前目录下大小为 0 的文件
for filename in `ls`
do
    if test -d $filename
    then b=0
    else
       a=$(ls -l $filename | awk '{ print $5 }')
            if test $a -eq 0
             then rm $filename
             fi
        fi
done
```