

Linux 学习笔记-shell 脚本

一、shell 简介

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：

- 1、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。
- 2、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 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。其中，bash 是 Linux 标准默认的 shell(内部命令一共有 40 个)。我们常说有多少种 Shell，其实说的是 Shell 脚本解释器。

大体上，可以将程序设计语言可以分为两类：编译型语言和解释型语言。

1、编译性语言

类似于 C/C++ 就是编译性语言，这类语言需要预先将我们写好的源代码(source code)转换成目标代码(object code)，这个过程被称作“编译”。运行程序时，直接读取目标代码(object code)。由于编译后的目标代码(object code)非常接近计算机底层，因此执行效率很高，这是编译型语言的

优点。

但是，由于编译型语言多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象，往往实现一个简单的功能需要大量复杂的代码。例如，在 C++ 里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。

2、解释型语言

解释型语言也被称作“脚本语言”。执行这类程序时，解释器(interpreter)需要读取我们编写的源代码(source code)，并将其转换成目标代码(object code)，再由计算机运行。因为每次执行程序都多了编译的过程，因此效率有所下降。

使用脚本编程语言的好处是，它们多半运行在比编译型语言还高的层级，能够轻易处理文件与目录之类的对象；缺点是它们的效率通常不如编译型语言。不过权衡之下，通常使用脚本编程还是值得的：花一个小时写成的简单脚本，同样的功能用 C 或 C++ 来编写实现，可能需要两天，而且一般来说，脚本执行的速度已经够快了，快到足以让人忽略它性能上的问题。脚本编程语言的例子有 Perl、Python、Ruby 与 Shell。

因为 Shell 似乎是各 UNIX 系统之间通用的功能，并且经过了 POSIX 的标准化。因此，Shell 脚本只要“用心写”一次，即可应用到很多系统上。因此，之所以要使用 Shell 脚本是基于：

- 1、简单性：Shell 是一个高级语言；通过它，你可以简洁地表达复杂的操作。
- 2、可移植性：使用 POSIX 所定义的功能，可以做到脚本无须修改就可在不同的系统上执行。
- 3、开发容易：可以在短时间内完成一个功能强大又好用的脚本。

但是，考虑到 Shell 脚本的命令限制和效率问题，下列情况一般不使用 Shell

- 1、资源密集型的任务，尤其在需要考虑效率时（比如，排序，hash 等等）。
- 2、需要处理大任务的数学操作，尤其是浮点运算，精确运算，或者复杂的算术运算（这种情况一般使用 C++ 或 FORTRAN 来处理）。
- 3、有跨平台（操作系统）移植需求（一般使用 C 或 Java）。

- 4、复杂的应用，在必须使用结构化编程的时候（需要变量的类型检查，函数原型，等等）。
- 5、对于影响系统全局性的关键任务应用。
- 6、对于安全有很高要求的任务，比如你需要一个健壮的系统来防止入侵、破解、恶意破坏等等。
- 7、项目由连串的依赖的各个部分组成。
- 8、需要大规模的文件操作。
- 9、需要多维数组的支持。
- 10、需要数据结构的支持，比如链表或数等数据结构。
- 11、需要产生或操作图形化界面 GUI。
- 12、需要直接操作系统硬件。
- 13、需要 I/O 或 socket 接口。
- 14、需要使用库或者遗留下来的老代码的接口。

私人的、闭源的应用（shell 脚本把代码就放在文本文件中，全世界都能看到）。

如果你的应用符合上边的任意一条，那么就考虑一下更强大的语言吧——或许是 Perl、Tcl、Python、Ruby——或者是更高层次的编译语言比如 C/C++，或者是 Java。

第一个简单的 Shell 例子

在写 shell 小程序之前，这里有一个建议：阅读《鸟哥私房菜基础学习篇》第 10 章关于 vi 的使用，特别是常用的命令，我们花一点点记一下，会对我们编程有很大很大的好处。

上面说了这么多，先看一个例子，来感受下 Shell。

- 1、打开文本编辑器，新建一个文件，扩展名为 sh.例如：test01.sh.
- 2、添加内容并保存。如下：

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

```
echo "hello world"
```

上面代码的含义：“#!” 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。

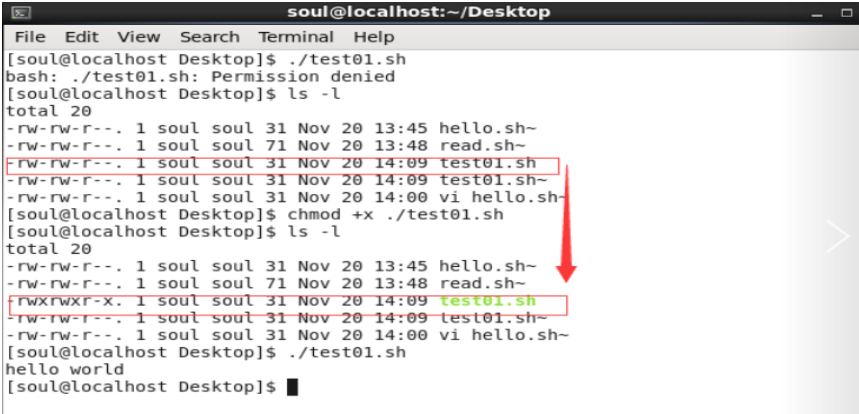
3、运行 test01.sh,命令为: ./test01.sh; 注意, 首先我们通过 ls -l 命令查看该文件的权限。

如果没有执行权限,则需要通过 chmod +x ./ test01.sh 来完成。

还有一点需要注意, 一定要写成./ test01.sh, 而不是 test01.sh。运行其它二进制的程序也一样, 直接写 test01.sh, linux 系统会去 PATH 里寻找有没有叫 test01.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 test01.sh 是会找不到命令的, 要用./hello.sh 告诉系统说, 就在当前目录找。

运行 shell 程序还有另外一种方式:/bin/sh test01.sh.(即指定解释器来执行我们的 shell 程序, 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。)

实践截图如下:



```
soul@localhost: ~/Desktop
File Edit View Search Terminal Help
[soul@localhost Desktop]$ ./test01.sh
bash: ./test01.sh: Permission denied
[soul@localhost Desktop]$ ls -l
total 20
-rw-rw-r--. 1 soul soul 31 Nov 20 13:45 hello.sh~
-rw-rw-r--. 1 soul soul 71 Nov 20 13:48 read.sh~
-rw-rw-r--. 1 soul soul 31 Nov 20 14:09 test01.sh
-rw-rw-r--. 1 soul soul 31 Nov 20 14:09 test01.sh~
-rw-rw-r--. 1 soul soul 31 Nov 20 14:00 vi hello.sh~
[soul@localhost Desktop]$ chmod +x ./test01.sh
[soul@localhost Desktop]$ ls -l
total 20
-rw-rw-r--. 1 soul soul 31 Nov 20 13:45 hello.sh~
-rw-rw-r--. 1 soul soul 71 Nov 20 13:48 read.sh~
-rwxrwxr-x. 1 soul soul 31 Nov 20 14:09 test01.sh
-rw-rw-r--. 1 soul soul 31 Nov 20 14:09 test01.sh~
-rw-rw-r--. 1 soul soul 31 Nov 20 14:00 vi hello.sh~
[soul@localhost Desktop]$ ./test01.sh
hello world
[soul@localhost Desktop]$
```

第二个 Shell 程序例子

功能: 读取控制台的数据, 并输出到控制台。

1、新建一个文件, test02.sh

2、输入内容, 如下:

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

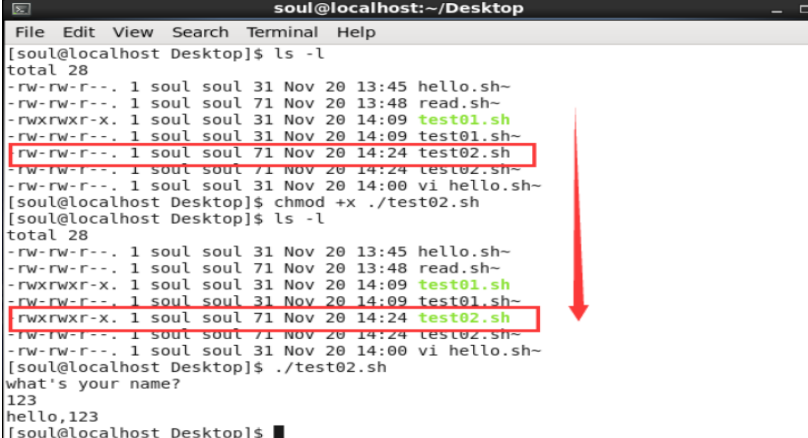
```
echo "what's your name?"
```

```
read PERSON
```

```
echo "hello,$PERSON"
```

read 命令从键盘读取变量的值

3、运行。



The screenshot shows a terminal window titled 'soul@localhost: ~/Desktop'. It displays the output of 'ls -l' twice. In the first listing, files 'test01.sh' and 'test02.sh' are highlighted with red boxes. In the second listing, after running 'chmod +x ./test02.sh', 'test02.sh' is highlighted with a red box. A red arrow points from the first 'test02.sh' entry to the second. Below the listings, the command './test02.sh' is executed, prompting 'what's your name?' and receiving the input '123'.

```
soul@localhost: ~/Desktop
File Edit View Search Terminal Help
[soul@localhost Desktop]$ ls -l
total 28
-rw-rw-r--. 1 soul soul 31 Nov 20 13:45 hello.sh~
-rw-rw-r--. 1 soul soul 71 Nov 20 13:48 read.sh~
-rwxrwxr-x. 1 soul soul 31 Nov 20 14:09 test01.sh~
-rw-rw-r--. 1 soul soul 31 Nov 20 14:09 test01.sh~
-rw-rw-r--. 1 soul soul 71 Nov 20 14:24 test02.sh~
-rw-rw-r--. 1 soul soul 71 Nov 20 14:24 test02.sh~
-rw-rw-r--. 1 soul soul 31 Nov 20 14:00 vi hello.sh~
[soul@localhost Desktop]$ chmod +x ./test02.sh
[soul@localhost Desktop]$ ls -l
total 28
-rw-rw-r--. 1 soul soul 31 Nov 20 13:45 hello.sh~
-rw-rw-r--. 1 soul soul 71 Nov 20 13:48 read.sh~
-rwxrwxr-x. 1 soul soul 31 Nov 20 14:09 test01.sh~
-rw-rw-r--. 1 soul soul 31 Nov 20 14:09 test01.sh~
-rwxrwxr-x. 1 soul soul 71 Nov 20 14:24 test02.sh~
-rw-rw-r--. 1 soul soul 71 Nov 20 14:24 test02.sh~
-rw-rw-r--. 1 soul soul 31 Nov 20 14:00 vi hello.sh~
[soul@localhost Desktop]$ ./test02.sh
what's your name?
123
hello,123
[soul@localhost Desktop]$
```

二、shell 变量

1、特殊变量

一般变量比较简单哈，与 Java、C/C++ 不一样的地方在于不需要定义，直接使用即可。

例如：

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

```
var= "hello world"
```

```
echo ${var}
```

需要注意的两点：

1、变量和等号之间不能有空格。

2、使用变量的时候我们只需要在变量前面加上一个美元符号\$即可。好的编程风格为：将变量用大括号括起来。例如：\${var}相比\$var 在风格上就更好。

特殊变量列表

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，下面将会讲到。
\$?	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。

我们先来看一个例子。

1、新建一个名为 testVar.sh 文件，并输入如下的内容

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

```
echo $0
```

```
echo $1
```

```
echo $2
```

```
echo $*
```

```
echo $@
```

```
echo $#
```

2、运行此文件：./testVar.sh AAAA BBBB CCCC.

结果如下：

```
[soul@localhost Desktop]$ ./testVar.sh AAAA BBBB CCCC
./testVar.sh
AAAA
BBBB
AAAA BBBB CCCC
AAAA BBBB CCCC
3
[soul@localhost Desktop]$
```

\$*和\$@的区别

这里有必要介绍下*和@的区别。

\$* 和 \$@ 都表示传递给函数或脚本的所有参数，当\$*和\$@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。

看到上面这句话可能还不太好理解\$*和\$@的区别, 下面通过一个例子就比较清楚的可以看出这两者的区别

1、新建一个文件 diff.sh, 并输入如下内容:

```
#!/bin/bash
echo "\${*}"={*}
echo "\${@}"={@}

echo "\"\${*}\""="{\"$*\}"
echo "\"\${@}\""="{\"$@\"}"

echo "print each element from \${*}:"
for var in $*
do
    echo $var
done

echo "print each element from \${@}:"
for var in $@
do
    echo ${var}
done

echo "print each element from \"\${*}\":"
for var in "$*"
do
    echo ${var}
done

echo "print each element from \"\${@}\":"
for var in "$@"
do
    echo ${var}
done
```

2、运行 diff.sh。

./diff.sh AAA BBB CCC 结果如下:

```
[soul@localhost Desktop]$ ./diff.sh AAA BBB CCC DDD
$*={AAA BBB CCC DDD}
$@={AAA BBB CCC DDD}
"$*="{AAA BBB CCC DDD}"
echo "$@="{AAA BBB CCC DDD}"
print each element from $*:
AAA
BBB
CCC
DDD
print each element from $@:
AAA
BBB
CCC
DDD
print each element from "$*":
AAA BBB CCC DDD
print each element from "$@":
AAA
BBB
CCC
DDD
[soul@localhost Desktop]$
```

看到上面的运行结果，再理解下下面这句话：

`$*` 和 `$@` 都表示传递给函数或脚本的所有参数

- 1、当`$*`和`$@`不被双引号(" ")包含时，都以"`$1`" "`$2`" ... "`$n`" 的形式输出所有参数。
- 2、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。

1、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。

例如：`date` 是 linux 是一个日期命令。我们在 Shell 编程中进行命令替换。

看一个例子：

- 1、新建一个 `testDate.sh`，并输入如下内容

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

```
DATE=`date`
```



```
echo "DATE is ${DATE}"
```

2、运行`./testDate.sh`。

结果如下：

```
[soul@localhost Desktop]$ ./testDate.sh
DATE is Mon Nov 20 15:17:33 PST 2017
[soul@localhost Desktop]$ date
Mon Nov 20 15:17:43 PST 2017
[soul@localhost Desktop]$
```

从结果中可以看到，确实进行了命令的替换。即 Shell 先执行命令 `date`，将输出结果保存到了 `DATE` 变量中。

2、变量替换

变量可以替换的形式有如下几种：

形式	说明
<code>\${var}</code>	变量本来的值
<code>\${var:-word}</code>	如果变量 <code>var</code> 为空或已被删除(unset)，那么返回 <code>word</code> ，但不改变 <code>var</code> 的值。
<code>\${var:=word}</code>	如果变量 <code>var</code> 为空或已被删除(unset)，那么返回 <code>word</code> ，并将 <code>var</code> 的值设置为 <code>word</code> 。
<code>\${var:?message}</code>	如果变量 <code>var</code> 为空或已被删除(unset)，那么将消息 <code>message</code> 送到标准错误输出，可以用来检测 <code>var</code> 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。
<code>\${var:+word}</code>	如果变量 <code>var</code> 被定义，那么返回 <code>word</code> ，但不改变 <code>var</code> 的值。

下面我们来写一个小 Demo，代码如下：

运行结果如下:

在运行结果中, 有点错误, 错误为: var= "today is friday " var= "hahah"

代码的等号右边有一个空格。这也就告诉我们变量赋值的 "=" 两边都不能有空格。否则都会报错。

三、shell 编程

1、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。

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

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

帅哥(*^▽^*)!

10 / 22

```
echo "Total value : $val"
```

注意：表达式和运算符之间要有空格。

算数运算符、关系运算符、布尔运算符、字符串运算符这些用法都比较简单哈，可能具体语法有一点点差异，在我们实际编程中，我们可以参考下网上的资料。

在自己实践过程中，再一次让我注意到了 Shell 编程中，像类似于 “=” 两边的表达式都不能有空格。不然会报错，这个需要我们注意。

在 Shell 中有的需要有空格，有的又不能有空格，确实比较蛋疼哈，因此就需要我们多实践总结

2、注释

以 “#” 开头的行就是注释，会被解释器忽略。

注意：sh 里没有多行注释，只能每一行加一个#号。

3、字符串

字符串可以用单引号，也可以用双引号，也可以不用引号。

单引号

例如：

```
str=hello world
```

```
echo $str    #输出：hello world
```

```
echo 'hello,$str' #输出：hello,$str.即单引号字符串中的变量是无效的
```

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

一般情况下，我们就是使用的是双引号，就没有单引号中的限制了,如下：

- 双引号里可以有变量
- 双引号里可以出现转义字符

示例如下：

```
1 #!/bin/bash
2
3 str='hello world'
4 echo $str      #输出:hello world
5 echo 'hello,$str'  #输出:hello,$str.即单引号字符串中的变量是无效的
6
7 #error----->str2='wojiushi\'mogui'
8 #echo $str2
9
10 # invalld ----->echo 'hello,$str'
11
12 str2="hello,shell"
13 echo "hello,$str2"|
```

```
File Edit View Search Terminal Help
[soul@localhost Desktop]$ ./test01.sh
hello world
hello,$str
hello,hello,shell
[soul@localhost Desktop]$
```

4、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。

4.1、定义数组

定义数组有两种格式：如下：

第一种

array_name=(value0 value1 value2 value3) #元素之间用空格分开

第二种：单独定义数组的各个分量

例如:

```
arr[0]="value0"
```

```
arr[1]="value1"
```

可以不使用连续的下标, 而且下标的范围没有限制

4.2、读取数组

获取数组中索引为 index 的语法如下:

语法: `${array_name[index]}`。

获取数组的全部元素的语法如下:

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

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

获取整个数组长度的语法如下:

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

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

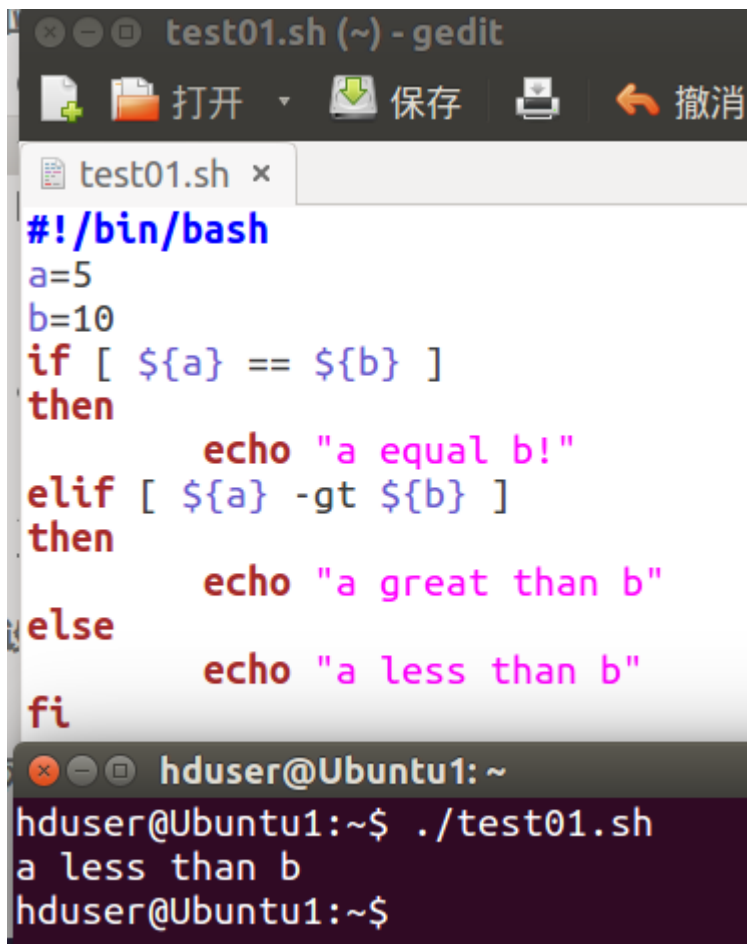
获取数组中索引为 index 元素的长度的语法如下:

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

5、条件语句(if)

在 shell 中有三种形式, 与其它语言类似。

直接看一个例子, 如下:



The screenshot shows a terminal window with a dark background. At the top, there's a window title bar for 'test01.sh (~) - gedit'. Below it is a menu bar with icons for '打开' (Open), '保存' (Save), and '撤消' (Undo). The main area shows the content of 'test01.sh', which is a shell script starting with a shebang line, followed by variable assignments and an if-elif-else block. Below the script, there's a terminal window showing the execution of the script, which outputs 'a less than b'.

```
#!/bin/bash
a=5
b=10
if [ ${a} == ${b} ]
then
    echo "a equal b!"
elif [ ${a} -gt ${b} ]
then
    echo "a great than b"
else
    echo "a less than b"
fi
```

```
hduser@Ubuntu1:~$ ./test01.sh
a less than b
hduser@Ubuntu1:~$
```

需要注意的是: `if [expression]` 中 `expression` 与方括号`[]`之间必须要有空格, 否则会报语法错误。

6、case 语句

case 值 in

模式 1)

command

;; #两个分号的作用与 Java/C 中的 break 语句的作用相同

模式 2)

command2

;;

*)

command3

```
;;
```

```
esac
```

case 语句的一点细节：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。

```
test01.sh x
#!/bin/bash
echo "please input a num(0~3)"
read inputNum

case ${inputNum} in
0)
    echo "input num is 0"
    ;;
1)
    echo "input num is 1"
    ;;
2)
    echo "input num is 2"
    ;;
3)
    echo "input num is 3"
    ;;
*)
    echo "input num is not in [0~3],please input again"
    ;;
esac

#两个分号的作用与Java/C中的break语句的作用相同
```

7、for(while)循环的基本语法

1.for ... in 语句

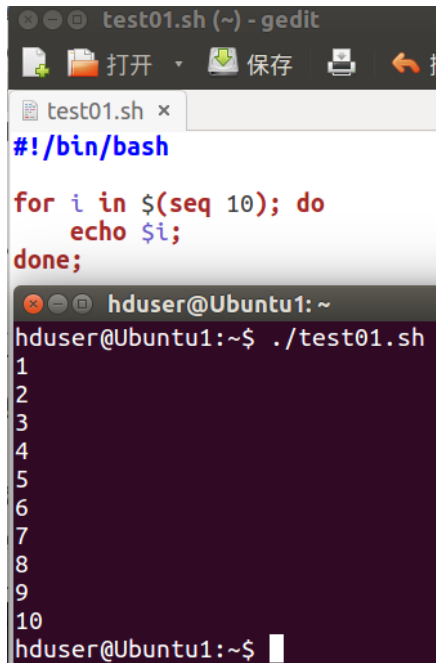
for 变量 in seq 字符串

do

action

done

说明：seq 字符串 只要用空格字符分割，每次 for...in 读取时候，就会按顺序将读到值，给前面的变量



```
test01.sh (~) - gedit
#!/bin/bash

for i in $(seq 10); do
    echo $i;
done;

hduser@Ubuntu1: ~
hduser@Ubuntu1:~$ ./test01.sh
1
2
3
4
5
6
7
8
9
10
hduser@Ubuntu1:~$
```

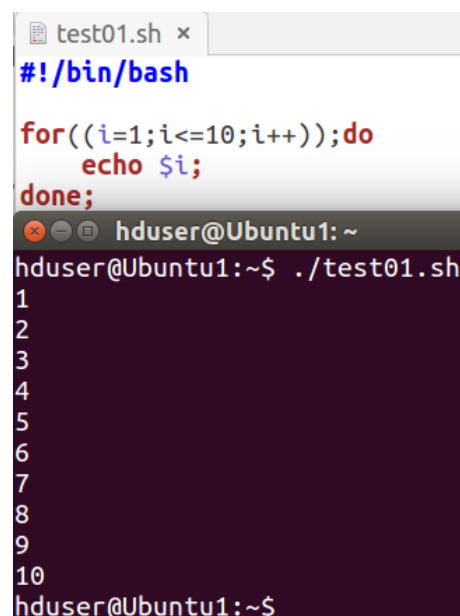
2.for((赋值; 条件; 运算语句))

for((赋值; 条件; 运算语句))

do

action

done;



```
test01.sh x
#!/bin/bash

for((i=1;i<=10;i++));do
    echo $i;
done;

hduser@Ubuntu1: ~
hduser@Ubuntu1:~$ ./test01.sh
1
2
3
4
5
6
7
8
9
10
hduser@Ubuntu1:~$
```

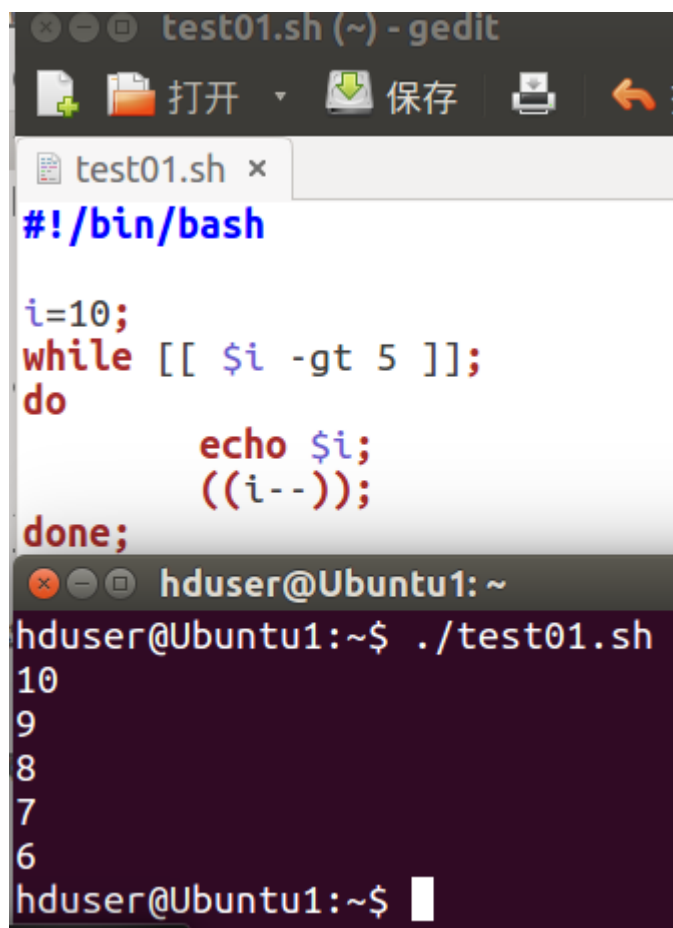

3. while 循环使用 (while/do/done)

while 条件语句

do

action

done;



The screenshot shows two windows. The top window is a text editor titled 'test01.sh (~) - gedit' with a toolbar containing icons for opening, saving, and printing files. The script content is as follows:

```
#!/bin/bash

i=10;
while [[ $i -gt 5 ]];
do
    echo $i;
    ((i--));
done;
```

The bottom window is a terminal titled 'hduser@Ubuntu1: ~'. It shows the command `./test01.sh` being executed, which results in the numbers 10, 9, 8, 7, and 6 being printed on separate lines. The prompt `hduser@Ubuntu1:~$` is visible at the bottom.

8、until 循环语句

语法结构:

until 条件;

do;

action;

done;

```
test01.sh x
#!/bin/bash

a=10;
until [[ $a -lt 5 ]];do
echo $a;
((a--));
done;

hduser@Ubuntu1: ~
hduser@Ubuntu1:~$ ./test01.sh
10
9
8
7
6
5
hduser@Ubuntu1:~$
```

9、函数

函数，基本和其它语言的函数一样。

函数定义的基本格式如下：

```
function function_name () {

    list of commands

    [ return value ]

}
```

有两点需要说明：

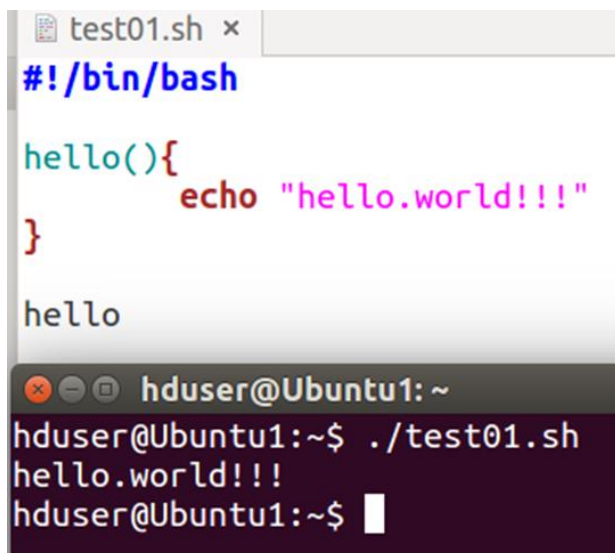
- 1、function 这个关键字可写可不写
- 2、return value 也是可写可不写，如果不写，则将函数最后一行的计算结果作为返回值。

注意：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。

看以下示例：输出“hello,world”的简单函数例子：



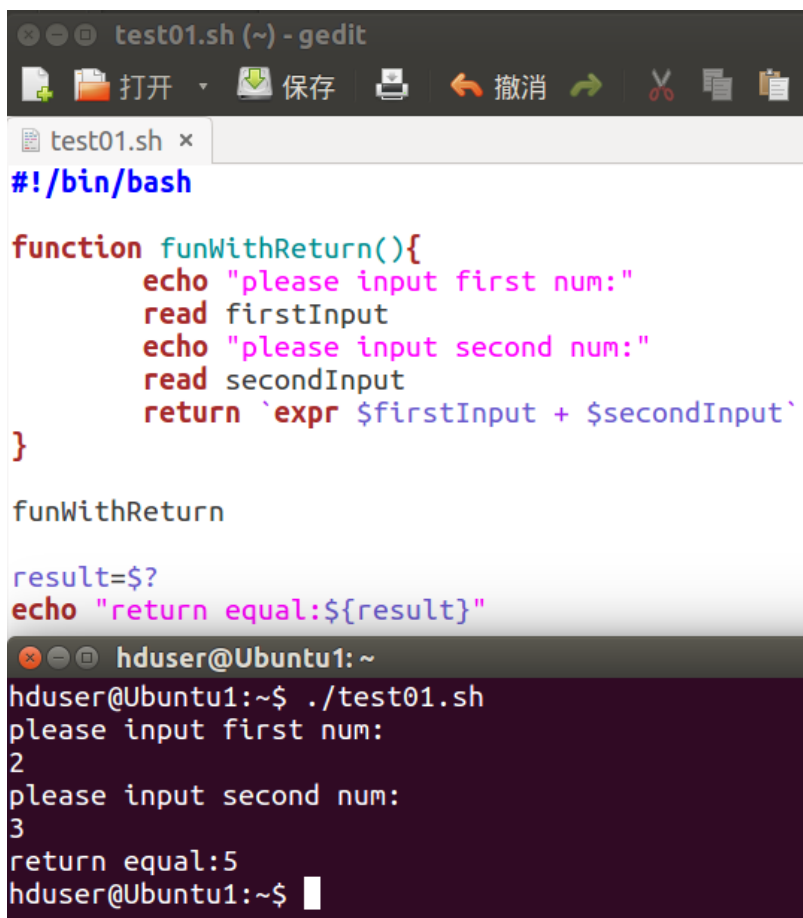
```
test01.sh x
#!/bin/bash

hello(){
    echo "hello.world!!!"
}

hello

hduser@Ubuntu1: ~
hduser@Ubuntu1:~$ ./test01.sh
hello.world!!!
hduser@Ubuntu1:~$
```

带返回值的函数例子：



```
test01.sh (~) - gedit
test01.sh x
#!/bin/bash

function funWithReturn(){
    echo "please input first num:"
    read firstInput
    echo "please input second num:"
    read secondInput
    return `expr $firstInput + $secondInput`
}

funWithReturn

result=$?
echo "return equal:${result}"

hduser@Ubuntu1: ~
hduser@Ubuntu1:~$ ./test01.sh
please input first num:
2
please input second num:
3
return equal:5
hduser@Ubuntu1:~$
```

函数的嵌套调用：

```
test01.sh x
#!/bin/bash

function fun(){
    echo "hello,world"
}

function fun1(){
    echo "hello,shell"
    fun      #executor function fun
}

fun1

hduser@Ubuntu1: ~
hduser@Ubuntu1:~$ ./test01.sh
hello,shell
hello,world
hduser@Ubuntu1:~$
```

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...

例子程序：求两个输入参数的和

```
test01.sh x
#!/bin/bash

function twoSum(){
    parTotal=$#
    echo $parTotal
    if [ ${parTotal} != 2 ]
    then
        echo "input para length is not 2"
        return
    fi
    echo "first para is:${1}"
    echo "first para is:${2}"
    return `expr ${1} + ${2}`
}

#twoSum 3 4 5
twoSum 3 4

sum=$?
echo "result is ${sum}"

hduser@Ubuntu1: ~
hduser@Ubuntu1:~$ ./test01.sh
2
first para is:3
first para is:4
result is 7
hduser@Ubuntu1:~$
```

10、包含文件

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。

看一个例子：

1、新建两个 Shell 文件，分别为 subFun.sh、mainFun.sh。

2、输入内容。

subTest.sh 文件中的内容截图如下：

```
#!/bin/bash

function hello(){
    echo "hello,world!!!"
}
hello
```

mainTest.sh

```
#!/bin/bash

function main(){
    . ./subTest.sh #contain subTest.sh file
    echo "hello,shell!!!"
}
main
```

3、执行

chmod +x ./mainTest.sh

./mainTest.sh

```
hduser@Ubuntu1:~$ ./mainTest.sh
bash: ./mainTest.sh: 权限不够
hduser@Ubuntu1:~$ chmod +x ./mainTest.sh
hduser@Ubuntu1:~$ ./mainTest.sh
hello,world!!!
hello,shell!!!
hduser@Ubuntu1:~$
```

注意：被包含脚本不需要有执行权限。例如：在本例中就不需要 subTest.sh 有执行权限