

Shell 笔记

郑华

2018 年 5 月 13 日

1 Shell 基本知识

1.1 Shell

Shell 就是一个命令行解释器，它的作用是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，这种方式称为交互式 (Interactive)

Shell 还有一种执行命令的方式称为批处理 (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 脚本和编程语言很相似，也有变量和流程控制语句，包括循环和分支。但 Shell 脚本是解释执行的，不需要编译，Shell 程序从脚本中一行一行读取并执行这些命令，相当于一个用户把脚本中的命令一行一行敲到 Shell 提示符下执行。作为程序设计语言，它虽然不是 Linux 系统内核的一部分，但它调用了系统内核的大部分功能来执行程序、创建文档并以并行的方式协调各个程序的运行

1.2 Shell 执行脚本

shell 执行脚本是一门解释性语言、批量化处理语言，大大的节省了工作成本，shell 脚本第一行必须以 #! 开头，它表示该脚本使用后面的解释器解释执行。

Example: //script.sh 注：这是一个文本文件

```
#!/bin/bash
#注意 echo 默认为换行输出，如果不换行+\c
echo "this_is_a_test"
ls
ls -l
echo "there_are_all_files"
```

1.2.1 执行方式

```
//第一种执行方式:  
[admin@localhost Shell]$ chmod +x script.sh  
[admin@localhost Shell]$ ./script.sh  
  
//第二种执行方式:  
[admin@localhost Shell]$ /bin/bash script.sh  
//等价于:  
[admin@localhost Shell]$ shell script.sh
```

1.2.2 执行过程

Shel 会 fork 一个子进程并调用 exec 执行./script.sh 这个程序,exec 系统调用应该把子进程的代码段替换成./script.sh 程序的代码段,并从它的 __start 始执行。然而 script.sh 是个文本文件,根本没有代码段和 __start 函数,怎么办呢? 其实 exec 还有另外一种机制,如果要执行的是一个文本文件,并且第一行指定了解释器,则用解释器程序的代码段替换当前进程,并且从解释器的 __start 开始执行,而这个文本文件被当作命令行参数传给解释器。因此,执行上述脚本相当于执行程序!

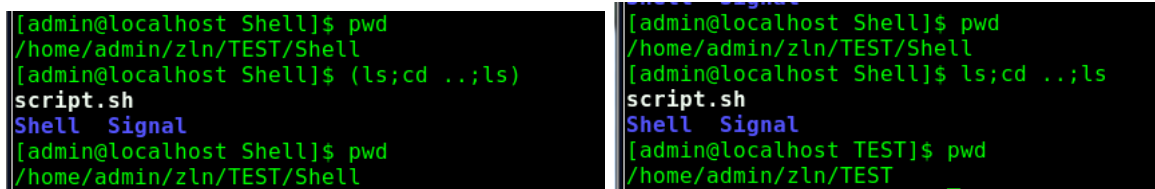
什么是“子程序”呢? 就是说,在我目前这个 shell 的情况下,去启用另一个新的 shell , 新的那个 shell 就是子程序

这个程序概念与变量有啥关系啊? 关系可大了! 因为子程序仅会继承父程序的环境变量,子程序不会继承父程序的自订变量

1. 交互式进程（父进程）创建一个子进程用于执行脚本，父进程等待子进程终止
2. 子进程程序替换 bash 解释器
3. 读取 shell 脚本的命令，将其以参数传递的方式传递给 bash 解释器
4. 子 bash 对 shell 脚本传入的参数进行读取，读一行识别到它是一个命令，则再创建一个子进程，子 bash 等待该新进程终止
5. 新进程执行该命令，执行完后将结果交给子进程
6. 子进程继续读取命令，创建新进程，新进程执行该命令，将结果返回给子进程，直到执行完最后一条命令
7. 子进程终止，将结果返回给交互式父进程

注意 注意：像 export、cd、env、set 这些内置命令，在键入命令行后，交互式进程不会创建子进程，而是调用 bash 内部的函数执行这些命令，改变的是交互式进程。

如果在命令行下，将多个命令用括号括起来，并用分号隔开来执行，交互式进程依然会创建一个子 shell 执行括号中的命令：



```
[admin@localhost Shell]$ pwd
/home/admin/zln/TEST/Shell
[admin@localhost Shell]$ (ls;cd ../ls)
script.sh
Shell Signal
[admin@localhost Shell]$ pwd
/home/admin/zln/TEST/Shell

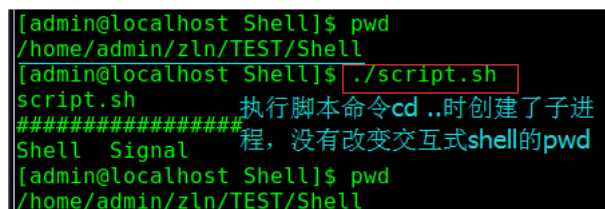
[admin@localhost Shell]$ pwd
/home/admin/zln/TEST/Shell
[admin@localhost Shell]$ ls;cd ../ls
script.sh
Shell Signal
[admin@localhost TEST]$ pwd
/home/admin/zln/TEST
```

图 1: (comands) VS Comands

或者 source 这两个命令是 Shell 的内建命令，这种方式不会创建子 Shell，而是直接在交互式 Shell 下逐行执行脚本中的命令

Example

```
#!/bin/bash
ls
echo "#####"
cd ..
ls
```



```
[admin@localhost Shell]$ pwd
/home/admin/zln/TEST/Shell
[admin@localhost Shell]$ ./script.sh
script.sh
#####
Shell Signal
[admin@localhost Shell]$ pwd
/home/admin/zln/TEST/Shell
```

图 2: without use . Or source Command



```
[admin@localhost Shell]$ pwd
/home/admin/zln/TEST/Shell
[admin@localhost Shell]$ . ./script.sh
script.sh
#####
Shell Signal
[admin@localhost TEST]$ pwd
/home/admin/zln/TEST

[admin@localhost Shell]$ pwd
/home/admin/zln/TEST/Shell
[admin@localhost Shell]$ source ./script.sh
script.sh
#####
Shell Signal
[admin@localhost TEST]$ pwd
/home/admin/zln/TEST
```

图 3: Use . Or source Command

1.3 Shell 变量

shell 变量不需要进行任何声明，直接定义即可，因为 shell 变量的值实际上都是字符串（对于没有定义的变量默认是一个空串）。定义的时候 shell 变量由大写字母加下划线组成，并且定义的时候等号两边不能存在空格，否则会被认为是命令！

1.3.1 shell 变量的种类

环境变量： shell 进程的环境变量可以从当前 shell 进程传给 fork 出来的子进程

本地变量： 只存在于当前 shell 进程

利用 `printenv` 可以显示当前 shell 进程的环境变量；利用 `set` 命令可以显示当前 shell 进程中的定义的所有变量（包括环境变量和本地变量）和函数。

一个 shell 变量定义后仅存在于当前 Shell 进程，是一个本地变量。用 `export` 命令可以把本地变量导出为环境变量。用 `unset` 命令可以删除已定义的环境变量或本地变量。

```
//分步 先定义后导出
COUNT=5
export COUNT

//一步完成定义和导出环境变量
export COUNT=5

//删除已经定义的环境变量
unset COUNT
```

变量引用： 引用 shell 变量要用到 `$` 符号，加 `{}` 可以防止歧义。

```
COUNT=5
echo $COUNT
echo ${COUNT}911
```

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"
```

```

[admin@localhost ~]$ COUNT=5
[admin@localhost ~]$ echo $COUNT
5
[admin@localhost ~]$ echo $COUNT911
5911
[admin@localhost ~]$ echo ${COUNT}911
5911

```

引起歧义

加花括号避免歧义

图 4: Shell Variable Use

//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。

```

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

// 以上没输出

```

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，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。

- 单引号字符串的限制:

1. 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；
2. 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）。

- 双引号字符串的好处:

1. 双引号里可以有变量
2. 双引号里可以出现转义字符

```

str='this_is_a_string'
your_name='qinix'
str="Hello, I know your are \"$your_name\"!\n"

```

```

// 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

```

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**：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)，还可以单独定义数组的各个分量
- **读取数组**：\${数组名[下标]}
- **求数组长度**:获取数组长度的方法与获取字符串长度的方法相同,length=\${#array_name[@]}

```

// 定义数组
array_name=(value0 value1 value2 value3)
array_name=(
    value0
    value1
    value2
    value3
)

// 单独定义数组的各个分量
array_name[0]=value0
array_name[1]=value1
array_name[n]=valuen

// 读取数组各元素和所有元素

```

```

valuei=${array_name[i]}
// 使用@符号可以获取数组中的所有元素
echo ${array_name[@]}

// 获取数组的长度
# 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# 开头的行为注释行

1.4 Shell 通配符、命令代换、单引号、双引号

见基础相关拓展.

1.5 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……

- \$#: 参数个数
- \$\$: 当前 shell 的 PID 进程 ID
- \$@和\$* 均表示所有参数，形式有所不同。\$@: "\$1" "\$2" ... "\$n"; \$*: "\$1 \$2 ... \$n"。

```

//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
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";

$ 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

1.6 Shell 各种括号

<http://blog.csdn.net/taiyang1987912/article/details/39551385>

1.6.1 ()-小括号

1. **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, 各命令和括号之间不必有空格
2. **命令替换**: 等同于`cmd`, shell 扫描一遍命令行, 发现了\$(cmd) 结构, 便将\$(cmd) 中的cmd 执行一次, 得到其标准输出, 再将此输出放到原来命令。有些shell 不支持, 如 tcsh。
3. 用于初始化数组。如: array=(a b c d)

1.6.2 (())-双小括号

1. **整数扩展**。这种扩展计算是整数型的计算, 不支持浮点型。((exp)) 结构扩展并计算一个算术表达式的值, 如果表达式的结果为 0, 那么返回的退出状态码为 1, 或者是"假", 而一个非零值的表达式所返回的退出状态码将为 0, 或者是"true"。若是逻辑判断, 表达式exp 为真则为 1, 假则为 0
2. 只要括号中的运算符、表达式符合 C 语言运算规则, 都可用在\$((exp)) 中, 甚至是三目运算符。作不同进位 (如二进制、八进制、十六进制) 运算时, 输出结果全都自动转化成了十进制。如: echo \$((16#5f)) 结果为95 (16 进位转十进制)
3. 单纯用(()) 也可重定义变量值, 比如 a=5; ((a++)) 可将 \$a 重定义为 6
4. 常用于算术运算比较, 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, 比如可以直接使用for((i=0;i<5;i++)), 如果不使用双括号, 则为for i in `seq 0 4`或者for i in {0..4}。再如可以直接使用if ((\$i<5)), 如果不使用双括号, 则为if [\$i -lt 5]

1.6.3 []-中括号

1. `bash` 的内部命令, `[`和`test` 是等同的。如果我们不用绝对路径指明, 通常我们用的都是 `bash` 自带的命令。`if/test` 结构中的**左中括号**是调用`test` 的命令标识, **右中括号**是关闭条件判断的。这个命令把它的参数作为比较表达式或者作为文件测试, 并且根据比较的结果来返回一个退出状态码。`if/test` 结构中并不是必须右中括号, 但是新版的 `Bash` 中要求必须这样
2. `Test`和`[]` 中可用的比较运算符**只有`==`和`!=`**, 两者都是用于字符串比较的, 不可用于整数比较, **整数比较只能使用`-eq`, `-gt` 这种形式**。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"`: `[ab \< bc]`, 结果为真, 也就是返回状态为 0。`[]` 中的逻辑与和逻辑或使用`-a` 和`-o` 表示。
3. **字符范围**。用作正则表达式的一部分, 描述一个匹配的字符范围。作为`test` 用途的中括号内**不能使用正则**
4. 在一个`array` 结构的上下文中, 中括号用来引用**数组中每个元素的编号**

1.6.4 [[]]-双中括号

1. `[[`是 `bash` 程序语言的关键字。并不是一个命令, `[[]]` 结构比`[]` 结构更加通用。在`[[`和`]]` 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**
2. 支持字符串的模式匹配, 使用`=~` 操作符时甚至支持 `shell` 的正则表达式。字符串比较时可以把右边的作为一个模式, 而不仅仅是一个字符串, 比如`[[hello == hell?]]`, 结果为真。`[[]]` 中匹配字符串或通配符, 不需要引号。
3. 使用`[[...]]` **条件判断结构**, 而不是`[...]`, **能够防止脚本中的许多逻辑错误**。比如, `&&`、`|`、`<` 和 `>` 操作符能够正常存在于`[[]]` 条件判断结构中, 但是如果出现在`[]` 结构中的话, 会报错。比如可以直接使用`if [[$a != 1 && $a != 2]]`, 如果不适用双括号, 则为`if [$a -ne 1] && [$a != 2]` 或者`if [$a -ne 1 -a $a != 2]`
4. `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

```
if ($i<5)
if [ $i -lt 5 ]
if [ $a -ne 1 -a $a != 2 ]
if [ $a -ne 1 ] && [ $a != 2 ]
if [[ $a != 1 && $a != 2 ]]
```

```
for i in $(seq 0 4);do echo $i;done
for i in `seq 0 4`;do echo $i;done
for ((i=0;i<5;i++));do echo $i;done
for i in {0..4};do echo $i;done
```

1.6.5 { }-大括号

{ } 常规用法

- 1.
- 2.
- 3.

{ } 特殊的替换用法

- 1.
- 2.
- 3.

{ } 模式匹配替换用法

- 1.
- 2.
- 3.

{ } 字符串提取和替换

- 1.
- 2.
- 3.

1.7 参考

基本使用参考博客<http://www.cnblogs.com/Lynn-Zhang/p/5758287.html>。

使用教程: <http://c.biancheng.net/cpp/view/6998.html>

菜鸟教程-使用教程 2: <http://www.runoob.com/linux/linux-shell.html>

Bash 在线运行网址: <http://www.runoob.com/try/runcode.php?filename=helloworld&type=bash>

2 Shell 运算操作

2.1 算数运算

```
#!/bin/bash

a=10
b=20

val=`expr $a + $b`
echo "a_+_b_:_$val"

val=`expr $a - $b`
echo "a_-_b_:_$val"

//乘号(*)前边必须加反斜杠(\)才能实现乘法运算
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

    //////////////// Result -->////////////////////

a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a 不等于 b

```

2.2 关系运算

```

#!/bin/bash

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

////////////////////Result ->////////////////////////////////////
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

```

2.3 布尔运算

```

#!/bin/bash

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-lt100-a$b-gt15: 返回true"
else
    echo "$a-lt100-a$b-gt15: 返回false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
    echo "$a-lt100-o$b-gt100: 返回true"
else

```

```

        echo "$a<100-o$b>100: 返回false"
    fi
    if [ $a -lt 5 -o $b -gt 100 ]
    then
        echo "$a<100-o$b>100: 返回true"
    else
        echo "$a<100-o$b>100: 返回false"
    fi
    //////////////////////////////////Result ->////////////////////////////////
    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

```

2.4 逻辑运算

```

#!/bin/bash

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

////////////////////////////////Result->////////////////////////////////
返回 false
返回 true

```

2.5 字符串运算

```

#!/bin/bash
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
//////////////////////////Result-->////////////////////////////////
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

```

2.6 文件测试运算

```
#!/bin/bash

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

//////////Result //////////
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

```

3 Shell 脚本控制结构

3.1 if-else

```

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 "没有符合条件的条件"
fi

```

3.2 while

let 命令 let 命令是 BASH 中用于计算的工具，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。如果表达式中包含了空格或其他特殊字符，则必须引起来

```

#!/bin/sh
int=1

```

```

while(( $int<=5 ))
do
    echo $int
    let "int++"
done

# let 使用方法
let a=5+4
let b=9-3
let no--
let ++no
let no+=10
let no-=20

```

3.3 for

```

for loop in 1 2 3 4 5
do
    echo "The_value_is:_$loop"
done
//////////Result is ....
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5

for str in 'This_is_a_string'
do
    echo $str
done
//////////Result is .....
This is a string

```

3.4 until

```

until condition
do
    command
done

```

3.5 case

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

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
```

3.6 continue,break

4 Shell 函数

4.1 函数定义

shell 中函数的定义格式如下：

```
[ function ] funname [()]
{
    action;
    [return int;]
}
```

注意

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)
3. 函数返回值在调用该函数后通过 \$? 来获得

```
#!/bin/bash
// Without Return
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-----"

---->Result:

-----函数开始执行-----
这是我的第一个 shell 函数!
-----函数执行完毕-----

// With Return
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_ $?_!"
```

---->Result:

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 !

4.2 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...

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

```
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
```

----->Result:

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：

参数类型	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。

表 1: 参数说明

5 Shell 脚本调用已有脚本

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
```

Example

```
//--->test1.sh
#!/bin/bash
url="http://www.runoob.com"

//--->test2.sh 引用test1.sh
#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"

--->Exec and Result:
$ chmod +x test2.sh
$ ./test2.sh
菜鸟教程官网地址: http://www.runoob.com
```

6 Shell 示例

How would you print just the 10th line of a file? From LeetCode

```
#!/bin/bash
count=0

while read line
do
    let ++count;
    if [ $count -eq 10 ] // if [[ $count <= 10 ]] if((count <= 10))
        then echo $line
        break
    fi
done < file.txt
```

7 参考文献

练习题: <https://wenku.baidu.com/view/e6664cafb64783e08122b4f.html>

<https://zhangge.net/4023.html>

参考: <http://www.cnblogs.com/90zeng/p/shellNotes.html>

在线解释王者: <https://explainshell.com>