ShellScript Documentation

发布 *1.0.0*

HappyAnony

2018 年 **06** 月 **20** 日

Contents

参考文档

- Shell脚本
- man bash文档

shell可以理解为一种脚本语言，像javascript等其它脚本语言一样，只需要一个能编写代码的文本编辑器和一个能解释执行的脚本解释器就可以

shell脚本的本质是：以某种语法格式将shell命令组织起来的由shell程序解析执行的脚本文本文件

由本质可知，要想掌握shell脚本，就需要了解并掌握下列三部分内容

- **shell命令**：即ls/cd等linux命令，详细可参考shell命令
- **shell解释器**：即sh/bash/csh等shell应用程序，详细可参考shell应用程序
- **shell语法**：即数据类型/变量/控制流语句/函数等编程语法

关于shell命令和shell解释器可参考上述指定的文档，本系列主要是对shell语法进行相关讲解，将从以下方面展开介绍：

CHAPTER 1

语法基础

shell语法基础有

1.1 脚本结构

我们在学习每一种编程语言时，都会先学习写一个hello world的demo程序，下面我们将从这个小demo程序来窥探一下我们shell脚本的程序结构

```
#!/bin/bash

# 注释信息

echo_str="hello world"

test(){
        echo $echo_str
}

test echo_str
```

首先我们可以通过文本编辑器(在这里我们使用linux自带文本编辑神器vim)，新建一个文件demo.sh，文件扩展名sh代表shell，表明该文件是一个shell脚本文件，并不影响脚本的执行，然后将上述代码片段写入文件中，保存退出

然后使用bash -n demo.sh命令可以检测刚才脚本文件的语法是否错误，如果没有回显结果就代表脚本文件没有语法错误

关于上述脚本文件中的代码语法，这里我们简单说明下，详细说明介绍将在下述文档中一一展开

- 脚本都以#!/bin/bash开头，#称为sharp，!在unix行话里称为bang，合起来简称就是常见的shabang。#!/bin/bash 指定了shell脚本解释器bash的路径，即使用bash程序作为该脚本文件的解释器，当然也可以使用其它的解释器/bin/sh等，根据具体环境进行相应选择
- echo_str是字符串变量，通过$进行引用变量的值，
- test是自定义函数名，通过函数名 传入参数格式进行函数的调用
- echo是shell命令，相对于c中的printf
- #字符用来注释shell脚本的

最后可以使用下列两种方式执行上述脚本

- 将脚本作为bash解释器的参数执行：此时首行的`#!/bin/bash`shabang可以不用写
  - `bash demo.sh`：直接将脚本文件作为bash命令的参数
  - `bash -x demo.sh`：使用`-x`参数可以查看脚本的详细执行过程
- 将脚本作为独立的可执行文件执行：此时首行的`#!/bin/bash`shabang必须写，用来指定shell解释器路径；同时脚本必须可执行权限
  - `chmod +x demo.sh`：给脚本添加执行权限
  - `./demo.sh`：执行脚本文件，在这里需要使用`./demo.sh`表明当前目录下脚本，因为`PATH`环境变量中没有当前目录，写成`demo.sh`系统会去`/sbin`、`/sbin`等目录下查找该脚本，无法找到该脚本文件执行，造成报错

1.2 数据类型

数据类型的本质：固定内存大小的别名

数据类型的作用：

- 确定对应变量分配的内存大小
- 确定对应变量所能支持的运算或操作

shell脚本是弱类型解释型的语言，在脚本运行时由解释器进行解释变量在什么时候是什么数据类型

在bash中，变量默认都是字符串类型，都是以字符串方式存储，所以在本章主要是介绍各数据类型变量所支持的运算或操作

虽说变量默认都是字符串类型，但是按照其使用场景可将数据类型分为以下几种类型：

- 数值型
- 字符串型
- 数组型
- 列表型

1.2.1 0x00 数值型

首先我们来声明定义一个数值型变量：`declare -i Var_Name`

- 虽说声明是一个数值型变量，但是存储依然是按照字符串的形式进行存储
- 该种方式声明，变量默认是本地全局变量，可以通过`local Var_Name`关键字将变量修改为局部变量，可以通过`export Var_Name`关键字将变量导出为环境变量
- 除了使用`declare -i`显式声明变量数据类型为数值型，还可以像`Var_Name=1`由解释器动态执行隐式声明该变量数据类型为数值型

数值型变量一般支持以下运算操作

- 算术运算
- 比较运算
- 数组索引

1.2.2 0x0000 算术运算

算术运算代码示例如下

```
#!/bin/bash

declare -i val=5    # 显式声明数值变量
num=2               # 隐式声明数值变量

# 使用[]运算符执行算术表达式$val+$num
# 使用$引用表达式执行结果
echo "val+num=$[$val+$num]"
echo "val++: $[val++]"   # 这里不需要加$，不是引用变量的值，而是修改变量的值
echo "val--: $[val--]"   # 这里不需要加$，不是引用变量的值，而是修改变量的值
echo "++val: $[++val]"   # 这里不需要加$，不是引用变量的值，而是修改变量的值
echo "--val: $[--val]"   # 这里不需要加$，不是引用变量的值，而是修改变量的值

# 使用(())运算符执行算术表达式
# 使用$引用表达式执行结果
echo "val-num=$(($val-$num))"
echo "val%num=$(($val%$num))"

# 使用let关键字执行算术表达式$val*$num
# 使用=运算符将执行结果赋值给变量
let ret=$val*$num
echo "var*num=$ret"

# 使用expr命令执行算术表达式$val/$num但是$val / $num之间需要用空格隔开
# 此时该表达式中的各个部分将作为参数传递给expr命令，最后使用``运算符引用命令的执行结果
# 使用=运算符将命令引用结果赋值给变量
ret=`expr $val / $num`
echo "val/num=$ret"

# 使用let关键字执行算术表达式+=、-=、*=、/=、%=
let val+=$num
echo "var+=num:$val"
let val-=$num
echo "var-=num:$val"
let val*=$num
echo "val*=num:$val"
let val/=$sum          # 貌似let不支持/=运算符
echo "val/=num:$val"
let val%=$num
echo "val%=num:$val"

# 执行结果如下
# val+num=7
# val++: 5
# val--: 6
# ++val: 6
# --val: 5
# val-num=3
# val%num=1
# var*num=10
# val/num=2
# var+=num:7
# var-=num:5
# val*=num:10
# ./test.sh: line 19: let: val/=: syntax error: operand expected (error token is "/
↪=")
# val/=num:10
# val%=num:0
```

由上述示例可知：数值类型变量支持的算术运算以及对应的算术运算符如下

- 加：+、+=、++
- 减：-、-=、--
- 乘：*、*=
- 除：/
- 取余：%、%=

1.2.3 0x0001 比较运算

比较运算有以下几种类型

- 用于条件测试
- 用于*for*循环

用于条件测试的示例代码如下

```
#!/bin/bash

declare -i val=5    # 显式声明数值变量
num=2               # 隐式声明数值变量

# -eq: 判断val变量的值是否等于5
# []运算符用来执行条件测试表达式，其执行结果要么为真，要么为假
# []运算符和条件测试表达式之间前后有空格
if [ $val -eq 5 ]; then
        echo "the value of val variable is 5"
fi

# -ne: 判断num变量的值是否不等于5
# [[]]运算符用来执行条件测试表达式，其执行结果要么为真，要么为假
# [[]]运算符和条件测试表达式之间前后有空格
if [[ $num -ne 5 ]];then
        echo "the value of num variable is not 5"
fi

# -le: 判断num变量的值是否小于或等于val变量的值
# test命令关键字用来执行条件测试表达式，其执行结果要么为真，要么为假
if test $num -le $val ;then
        echo "the value of num variable is lower or equal than val variable"
fi

# -ge: 判断val变量的值是否大于或等于num变量的值
# [[]]运算符用来执行条件测试表达式，其执行结果要么为真，要么为假
# [[]]运算符和条件测试表达式之间前后有空格
if [[ $val -ge $num ]];then
        echo "the value of val variable is growth or equal than num variable"
fi

# -gt: 判断val变量的值是否大于5
# []运算符用来执行条件测试表达式，其执行结果要么为真，要么为假
# []运算符和条件测试表达式之间前后有空格
if [ $val -gt 2 ];then
        echo "the value of val variable is growth than 2"
fi

# -lt: 判断num变量的值是否小于5
# [[]]运算符用来执行条件测试表达式，其执行结果要么为真，要么为假
# [[]]运算符和条件测试表达式之间前后有空格
```

(continues on next page)

(续上页)

```
if [[ $num -lt 5 ]];then
        echo "the value of num variable is lower than 5"
fi

# 执行结果如下
# the value of val variable is 5
# the value of num variable is not 5
# the value of num variable is lower or equal than val variable
# the value of val variable is growth or equal than num variable
# the value of val variable is growth than 2
# the value of num variable is lower than 5
```

由上述示例可知：数值类型变量用于条件测试时支持的比较运算以及对应的运算符如下

- 等于：-eq
- 不等于：-ne
- 小于等于：-le
- 大于等于：-ge
- 大于：-gt
- 小于：-lt
- 逻辑与：&&
- 逻辑非：！
- 逻辑或：||

用于用于for循环的示例代码如下

```
#!/bin/bash

# ==判断变量i的值是否等于1
for ((i=1; i==1; i++));do
        echo $i
done

# !=判断变量i的值是否不等于3
for ((i=1; i!=3; i++)); do
        echo $i
done

# <=判断变量i的值是否小于等于4
for ((i=1; i<=4; i++)); do
        echo $i
done

# >=判断变量i的值是否大于等于1
for ((i=5; i>=1; i--));do
        echo $i
done

# <判断变量i的值是否小于7
# >判断变量i的值是否大于0
# &&表示逻辑与
# ||表示逻辑或
# !表示逻辑非
# 非的优先级大于与，与的优先级大于或
for ((i=1; i>0 && i<7; i++)); do
        echo $i
done
```

由上述示例可知：数值类型变量用于for循环时支持的比较运算以及对应的运算符如下

- 等于：==
- 不等于：!=
- 小于等于：<=
- 大于等于：>=
- 大于：>
- 小于：<
- 逻辑与：&&
- 逻辑非：!
- 逻辑或：||

1.2.4 0x0002 数组索引

数组类型变量当做数组索引可参考数组型变量一节

1.2.5 0x01 字符串型

首先我们来声明定义一个字符串型变量：`Var_Name="anony"`

- 在bash中，变量默认都是字符串类型，也都是以字符串方式存储，所以字符串可以不需要使用""，除非特殊声明，否则都会解释成字符串
- 该种方式声明，变量默认是本地全局变量，可以通过`local Var_Name`关键字将变量修改为局部变量，可以通过`export Var_Name`关键字将变量导出为环境变量
- 该种声明定义方式是由shell解释器动态执行隐式声明该变量数据类型为字符串型

字符串型变量一般支持以下运算操作

- 返回字符串长度：`${#Var_Name}`(长度包括空白字符)
- 字符串消除
  - `${var#*word}`：查找var中自左而右第一个被word匹配到的串，并将此串及向左的所有内容都删除；此处为非贪婪匹配
  - `${var##*word}`：查找var中自左而右最后一个被word匹配到的串，并将此串及向左的所有内容都删除；此处为贪婪匹配
  - `${var%word*}`：查找var中自右而左第一个被word匹配到的串，并将此串及向右的所有内容都删除；此处为非贪婪匹配
  - `${var%%word*}`：查找var中自右而左最后一个被word匹配到的串，并将此串及向右的所有内容都删除；此处为贪婪匹配
- 字符串提取
  - `${var:offset}`：自左向右偏移offset个字符，取余下的字串；例如：name=jerry，`${name:2}`结果为rry
  - `${var:offset:length}`：自左向右偏移offset个字符，取余下的length个字符长度的字串。例如：‘‘name=’hello world’ ${name:2:5}结果为llo w‘‘
- 字符串替换
  - `${var/Pattern/Replaceplacement}`：以Pattern为模式匹配var中的字串，将第一次匹配到的替换为Replaceplacement；此处为非贪婪匹配，Pattern模式可参考正则表达式

- `${var//Pattern/Replaceplacement}`：以`Pattern`为模式匹配`var`中的字串，将全部匹配到的替换为`Replaceplacement`；此处为贪婪匹配，`Pattern`模式可参考正则表达式

代码示例如下：

```
#!/bin/bash
echo "PATH variable is $PATH"
echo "the length of PATH variable is ${#PATH}"

file_name="linux.test.md"
echo "${file_name%%.*}"
echo "${file_name%.*}"
echo "${file_name##*.}"
echo "${file_name#*.}"
echo "${file_name:0:5}"
echo "${file_name:2}"

test_str="/usr/bin:/root/bin:/usr/local/apache/bin:/usr/local/mysql:/usr/local/
↪apache/bin"
echo "${test_str/:\/usr\/local\/apache\/bin/}"    # 此处需要使用\对/进行转义，替换值为空表
示删除前面匹配到的内容
echo "${test_str//:\/usr\/local\/apache\/bin/}"   # 此处需要使用\对/进行转义，替换值为空表
示删除前面匹配到的内容

# 执行结果如下
# PATH variable is /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
# the length of PATH variable is 59
# linux
# linux.test
# md
# test.md
# linux
# nux.test.md
# /usr/bin:/root/bin:/usr/local/mysql:/usr/local/apache/bin
# /usr/bin:/root/bin:/usr/local/mysql
```

1.2.6 0x02 数组型

数组是一种数据结构，也可以叫做数据序列，它是一段连续的内容空间，保存了连续的多个数据(数据类型可以不相同)，可以使用数组index索引来访问操作数组元素

根据数组index索引的不同可将数组分为

- 普通数组：数组index索引为整型数
- 关联数组：数组index索引为字符串

0x0200 普通数组

普通数组也可以称为整型索引数组，它的声明定义方式有以下几种

```
#!/bin/bash

# 使用declare -a显式声明变量数据类型为整型索引数组型
# 数组中各元素间使用空白字符分隔
# 字符串类型的元素使用引号
declare -a array1=(1 'b' 3 'a')
# 依次引用数组的第一、二、三、四个元素
# 不加下标时默认引用第一个元素
# 引用时必须加上{}，否则$array1[0]的值为1[0]
echo "the first element of array1 is ${array1[0]}"
```

(continues on next page)

(续上页)

```
echo "the second element of array1 is ${array1[1]}"
echo "the third element of array1 is ${array1[2]}"
echo "the fourth element of array1 is ${array1[3]}"
# 查看数组所有元素
echo "all elements of array1 is ${array1[*]}"
echo "all elements of array1 is ${array1[@]}"


# 由解释器动态解释变量数据类型为整型索引数组型
# 如果数组中各元素间使用逗号，则它们将作为一个整体，也就是数组索引0的值
array2=(1,'b',3,'a')
echo "the first element of array2 is ${array2[0]}"


# 由解释器动态解释变量数据类型为整型索引数组型
# 数组元素使用自定义下标赋值
# 以下数组定义中，第一个元素是1，第二个元素是'b'，第3个元素为空，第4个元素为'a'
array3=(1 'b' [3]='a')
# 依次引用数组的第一、二、三、四个元素
# 不加下标时默认引用第一个元素
echo "the first element of array3 is ${array3[0]}"
echo "the second element of array3 is ${array3[1]}"
echo "the third element of array3 is ${array3[2]}"
echo "the fourth element of array3 is ${array3[3]}"
# 查看数组中所有有效元素(不为空)的整型索引号
echo "the index of effective element is ${!array3[*]}"
echo "the index of effective element is ${!array3[@]}"
# 查看数组中的有效元素个数(只统计值不为空的元素)
echo "the num of array3 is ${#array3[*]}"
echo "the num of array3 is ${#array3[@]}"


# 由解释器动态解释变量数据类型为整型索引数组型
# 数组中每个元素被逐渐赋值
array4[0]=1
array4[1]='bc'
array4[2]=3
array4[3]='a'
# 依次引用数组的第一、二、三、四个元素
# 不加下标时默认引用第一个元素
echo "the first element of array4 is ${array4[0]}"
echo "the second element of array4 is ${array4[1]}"
echo "the third element of array4 is ${array4[2]}"
echo "the fourth element of array4 is ${array4[3]}"
# 查看第二个元素的字符长度
echo "the length of second element is ${#array4[1]}"


# 执行结果如下
# the first element of array1 is 1
# the second element of array1 is b
# the third element of array1 is 3
# the fourth element of array1 is a
# all elements of array1 is 1 b 3 a
# all elements of array1 is 1 b 3 a
# the first element of array2 is 1,b,3,a
# the first element of array3 is 1
# the second element of array3 is b
# the third element of array3 is
# the fourth element of array3 is a
# the index of effective element is 0 1 3
```

(continues on next page)

(续上页)

```
# the index of effective element is 0 1 3
# the num of array3 is 3
# the num of array3 is 3
# the first element of array4 is 1
# the second element of array4 is bc
# the third element of array4 is 3
# the fourth element of array4 is a
# the length of second element is 2
```

另外普通数组还支持以下运算操作

- 返回数组长度(即有效元素的个数，不包括空元素)
  - `${#Array_Name[*]}`
  - `${#Array_Name[@]}`
- 数组元素消除，该操作不会修改原数组元素，操作执行结果用数组来接收
  - `Array_Name1=${Array_Name[*]#*word}`：功能同下
  - `Array_Name1=${Array_Name[*]##*word}`：自左而右查找`Array_Name`数组中所有被匹配到的`word`匹配到的元素，并将所有匹配到的元素删除(并不会删除原数组中的元素)，最后返回剩余的数组元素
  - `Array_Name1=${Array_Name[*]%word*}`：功能同下
  - `Array_Name1=${Array_Name[*]%%word*}`：自右而左查找`Array_Name`数组中所有被匹配到的`word`匹配到的元素，并将所有匹配到的元素删除(并不会删除原数组中的元素)，最后返回剩余的数组元素
- 数组元素提取，该操作不会修改原数组元素，操作执行结果用数组来接收
  - `Array_Name1=${Array_Name[*]:offset}`：返回`Array_Name`数组中索引为`offset`的数组元素以及后面所有元素；其中`offset`为整型数
  - `Array_Name1=${Array_Name[*]:offset:length}`：返回`Array_Name`数组中索引为`offset`的数值元素以及后面`length-1`个元素；其中`offset`和`length`都为整型数
- 数组元素替换，该操作不会修改原数组元素，操作执行结果用数组来接收
  - `Array_Name1=${Array_Name[*]/Pattern/Replaceplacement}`：功能同下
  - `Array_Name1=${Array_Name[*]//Pattern/Replaceplacement}`：以`Pattern`为模式匹配`Array_Name`数组中的元素，将全部匹配到的替换为`Replaceplacement`(不会修改原数组中的元素)，并返回全部数组元素；`Pattern`模式可参考正则表达式

代码示例如下

```
#!/bin/bash

array_test=(/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin)

# 返回数组长度(即有效元素的个数，不包括空元素)
echo "the length of array_test is ${#array_test[*]}"
echo "the length of array_test is ${#array_test[@]}"

# 数组元素消除，该操作不会修改原数组元素，操作执行结果用数组来接收
array_test1=${array_test[*]#*/usr/apache/bin}
echo "array_test:${array_test[*]}"
echo "array_test1:${array_test1[@]}"
array_test2=${array_test[*]##*/usr/apache/bin}
echo "array_test:${array_test[*]}"
echo "array_test2:${array_test2[@]}"
array_test3=${array_test[*]%/usr/apache/bin*}
echo "array_test:${array_test[*]}"
```

(continues on next page)

(续上页)

```
echo "array_test3:${array_test3[@]}"
array_test4=${array_test[*]%%/usr/apache/bin*}
echo "array_test:${array_test[*]}"
echo "array_test4:${array_test4[@]}"

# 数组元素提取，该操作不会修改原数组元素，操作执行结果用数组来接收
array_test5=${array_test[*]:2}
echo "array_test:${array_test[*]}"
echo "array_test5:${array_test5[@]}"
array_test6=${array_test[*]:2:2}
echo "array_test:${array_test[*]}"
echo "array_test6:${array_test6[@]}"

# 数组元素替换，该操作不会修改原数组元素，操作执行结果用数组来接收
array_test7=${array_test[*]/\/usr\/apache\/bin/}    # 需要用\对/进行转义，替换值为空表示删除前面匹配到的
echo "array_test:${array_test[*]}"
echo "array_test7:${array_test7[@]}"
array_test8=${array_test[*]//\/usr\/apache\/bin/}   # 需要用\对/进行转义，替换值为空表示删除前面匹配到的
echo "array_test:${array_test[*]}"
echo "array_test8:${array_test8[@]}"

# 执行结果如下
# the length of array_test is 5
# the length of array_test is 5
# array_test:/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin
# array_test1:/usr/bin /root/bin /usr/mysql
# array_test:/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin
# array_test2:/usr/bin /root/bin /usr/mysql
# array_test:/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin
# array_test3:/usr/bin /root/bin /usr/mysql
# array_test:/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin
# array_test4:/usr/bin /root/bin /usr/mysql
# array_test:/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin
# array_test5:/usr/apache/bin /usr/mysql /usr/apache/bin
# array_test:/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin
# varray_test6:/usr/apache/bin /usr/mysql
# array_test:/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin
# array_test7:/usr/bin /root/bin /usr/mysql
# array_test:/usr/bin /root/bin /usr/apache/bin /usr/mysql /usr/apache/bin
# array_test8:/usr/bin /root/bin /usr/mysql
```

同时普通数组也可用于for循环遍历

代码示例如下

```
#!/bin/bash

# 获取家目录下文件列表，转换成普通数组
array_test=(`ls ~`)
echo ${array_test[@]}
echo "-----------------"

# 以数组元素值的方式直接遍历数组
for i in ${array_test[*]};do
        echo $i
done
echo "-----------------"

# 以数组index索引的方式遍历数组
```

(continues on next page)

(续上页)

```
for i in ${!array_test[*]};do
        echo ${array_test[$i]}
done
echo "----------------"

# 以数组元素个数的方式遍历数组
for ((i=0;i<${#array_test[*]};i++));do
        echo ${array_test[$i]}
done

# 执行结果如下
# anaconda-ks.cfg demo.sh test1.sh test.sh
# ----------------
# anaconda-ks.cfg
# emo.sh
# est1.sh
# test.sh
# ----------------
# anaconda-ks.cfg
# demo.sh
# test1.sh
# test.sh
# ----------------
# anaconda-ks.cfg
# demo.sh
# test1.sh
# test.sh
```

0x0201 关联数组

关联数组也可以称为字符索引数组，它的声明定义方式有以下几种

```
#!/bin/bash

# 声明定义字符索引数组时必须使用declare -A
# 数组中各元素间使用空白字符分隔
declare -A array1=([name1]=jack [name2]=anony)
# 依次引用name1和name2对应的值
echo "the value of name1 element is ${array1[name1]}"
echo "the value of name2 element is ${array1[name2]}"


# 声明定义字符索引数组时必须使用declare -A
# 如果数组中各元素间使用逗号，则它们将作为一个整体
declare -A array2=([name1]=jack,[name2]=anony)
echo "the value of name1 element is ${array2[name1]}"
# 查看name1对应值的字符长度
echo "the length of name1 element is ${#array2[name1]}"


# 声明定义字符索引数组时必须使用declare -A
declare -A array3=([name1]=jack [name2]=anony)
echo "the value of name1 element is ${array3[name1]}"
echo "the value of name2 element is ${array3[name2]}"
# 通过字符索引进行赋值
array3[name3]=zhangsan
echo "the value of name3 element is ${array3[name3]}"
# 通过字符索引进行赋值
array3[name5]=lisi
```

(continues on next page)

(续上页)

```
# 查看数组所有元素
echo "the all effective element is ${array3[*]}"
echo "the all effective element is ${array3[@]}"
# 查看数组中所有有效元素(不为空)的字符索引号，默认是对应值的排列顺序
echo "the index of all effective element is ${!array3[*]}"
echo "the index of all effective element is ${!array3[@]}"
# 查看数组中的有效元素个数(只统计值不为空的元素)
echo "the length of array is ${#array3[*]}"
echo "the length of array is ${#array3[@]}"

# 执行结果如下
# the value of name1 element is jack
# the value of name2 element is anony
# the value of name1 element is jack,[name2]=anony
# the length of name1 element is 18
# the value of name1 element is jack
# the value of name2 element is anony
# the value of name3 element is zhangsan
# the all effective element is zhangsan anony jack lisi
# the all effective element is zhangsan anony jack lisi
# the index of all effective element is name3 name2 name1 name5
# the index of all effective element is name3 name2 name1 name5
# the length of array is 4
# the length of array is 4
```

和普通数组一样，关联数组也支持以下运算操作

- 返回数组长度(即有效元素的个数，不包括空元素)
  - `${#Array_Name[*]}`
  - `${#Array_Name[@]}`
- 数组元素消除，该操作不会修改原数组元素，操作执行结果用数组来接收
  - `declare -A Array_Name1=${Array_Name[*]#*word}`: 功能同下
  - `declare -A Array_Name1=${Array_Name[*]##*word}`: 自左而右查找`Array_Name`数组中所有被匹配到的`word`匹配到的元素，并将所有匹配到的元素删除(并不会删除原数组中的元素)，最后返回剩余的数组元素
  - `declare -A Array_Name1=${Array_Name[*]%word*}`: 功能同下
  - `declare -A Array_Name1=${Array_Name[*]%%word*}`: 自右而左查找`Array_Name`数组中所有被匹配到的`word`匹配到的元素，并将所有匹配到的元素删除(并不会删除原数组中的元素)，最后返回剩余的数组元素
- 数组元素提取，该操作不会修改原数组元素，操作执行结果用数组来接收
  - `declare -A Array_Name1=${Array_Name[*]:offset}`: 返回`Array_Name`数组中索引为`offset`的数组元素以及后面所有元素；其中`offset`为整型数
  - `declare -A Array_Name1=${Array_Name[*]:offset:length}`: 返回`Array_Name`数组中索引为`offset`的数值元素以及后面`length-1`个元素；其中`offset`和`length`都为整型数
- 数组元素替换，该操作不会修改原数组元素，操作执行结果用数组来接收
  - `declare -A Array_Name1=${Array_Name[*]/Pattern/Replaceplacement}`: 功能同下
  - `declare -A Array_Name1=${Array_Name[*]//Pattern/Replaceplacement}`: 以`Pattern`为模式匹配`Array_Name`数组中的元素，将全部匹配到的替换为`Replaceplacement`(不会修改原数组中的元素)，并返回全部数组元素；`Pattern`模式可参考正则表达式

代码示例如下

```
#!/bin/bash

declare -A array_test=([ele1]=/usr/bin [ele2]=/root/bin [ele3]=/usr/apache/bin␣
↪[ele4]=/usr/mysql [ele5]=/usr/apache/bin)

# 返回数组长度(即有效元素的个数，不包括空元素)
echo "the length of array_test is ${#array_test[*]}"
echo "the length of array_test is ${#array_test[@]}"

# 数组元素消除，该操作不会修改原数组元素，操作执行结果用数组来接收
declare -A array_test1=${array_test[*]#*/usr/apache/bin}
echo "array_test:${array_test[*]}"
echo "array_test1:${array_test1[@]}"
declare -A array_test2=${array_test[*]##*/usr/apache/bin}
echo "array_test:${array_test[*]}"
echo "array_test2:${array_test2[@]}"
declare -A array_test3=${array_test[*]%/usr/apache/bin*}
echo "array_test:${array_test[*]}"
echo "array_test3:${array_test3[@]}"
declare -A array_test4=${array_test[*]%%/usr/apache/bin*}
echo "array_test:${array_test[*]}"
echo "array_test4:${array_test4[@]}"

# 数组元素提取，该操作不会修改原数组元素，操作执行结果用数组来接收
declare -A array_test5=${array_test[*]:2}
echo "array_test:${array_test[*]}"
echo "array_test5:${array_test5[@]}"
declare -A array_test6=${array_test[*]:2:2}
echo "array_test:${array_test[*]}"
echo "array_test6:${array_test6[@]}"

# 数组元素替换，该操作不会修改原数组元素，操作执行结果用数组来接收
declare -A array_test7=${array_test[*]/\/usr\/apache\/bin/}
echo "array_test:${array_test[*]}"
echo "array_test7:${array_test7[@]}"
declare -A array_test8=${array_test[*]//\/usr\/apache\/bin/}
echo "array_test:${array_test[*]}"
echo "array_test8:${array_test8[@]}"

# 执行结果如下
# the length of array_test is 5
# the length of array_test is 5
# array_test:/usr/mysql /usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test1:/usr/mysql  /usr/bin /root/bin
# array_test:/usr/mysql /usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test2:/usr/mysql  /usr/bin /root/bin
# array_test:/usr/mysql /usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test3:/usr/mysql  /usr/bin /root/bin
# array_test:/usr/mysql /usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test4:/usr/mysql  /usr/bin /root/bin
# array_test:/usr/mysql /usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test5:/usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test:/usr/mysql /usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test6:/usr/apache/bin /usr/bin
# array_test:/usr/mysql /usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test7:/usr/mysql  /usr/bin /root/bin
# array_test:/usr/mysql /usr/apache/bin /usr/bin /root/bin /usr/apache/bin
# array_test8:/usr/mysql  /usr/bin /root/bin
```

关联数组和普通数组一样，也可用于for循环遍历

先创建`test.log`文件，内容如下

```
#cat ~/test.log
portmapper
portmapper
portmapper
portmapper
portmapper
portmapper
status
status
mountd
mountd
mountd
mountd
mountd
mountd
nfs
nfs
nfs_acl
nfs
nfs
nfs_acl
nlockmgr
nlockmgr
nlockmgr
nlockmgr
nlockmgr
nlockmgr
```

代码示例如下：统计文件中重复行的次数

```
#!/bin/bash

declare -A array_test

for i in `cat ~/test.log`;do
        let ++array_test[$i]  # 修改数组元素值
done

for j in ${!array_test[*]};do
        printf "%-15s %3s\n" $j :${array_test[$j]}
done

# 执行结果如下
# status            :2
# nfs               :4
# portmapper        :6
# nlockmgr          :6
# nfs_acl           :2
# mountd            :6
```

1.2.7 0x03 列表型

列表型变量常用来for循环遍历，但是一般是在for循环中直接使用，当然也可以通过变量进行引用

代码示例如下

```
#!/bin/bash

# 生成数字列表：使用{}运算符
for i in {1..4};do
```

(continues on next page)

(续上页)

```
        echo $i
done
echo "--------------------"

# 生成数字列表：使用seq命令
for i in `seq 1 2 7`;do
        echo $i
done
echo "--------------------"

# 生成文件列表：直接给出列表
for fileName in /etc/init.d/functions /etc/rc.d/rc.sysinit /etc/fstab;do
        echo $fileName
done
echo "--------------------"

# 生成文件列表：使用文件名通配机制生成列表
dirName=/etc/rc.d
for fileName in $dirName/*.d;do
        echo $fileName
done
echo "--------------------"

# 生成文件列表：使用``运算符引用相关命令的执行结果
for fileName in `ls ~`;do
        echo $fileName
done

# 执行结果如下
# 1
# 2
# 3
#4
# --------------------
# 1
# 3
# 5
# 7
# --------------------
# /etc/init.d/functions
# /etc/rc.d/rc.sysinit
# /etc/fstab
# --------------------
# /etc/rc.d/init.d
# /etc/rc.d/rc0.d
# /etc/rc.d/rc1.d
# /etc/rc.d/rc2.d
# /etc/rc.d/rc3.d
# /etc/rc.d/rc4.d
# /etc/rc.d/rc5.d
# /etc/rc.d/rc6.d
# --------------------
# anaconda-ks.cfg
# demo.sh
# test1.sh
# test.log
# test.sh
```

1.3 变量

变量是一种逻辑概念，变量有三要素(也可称为三种属性)

- 数据类型：变量存储数据的类型；用来确定该变量存储数据的内存大小以及存储数据所能支持的运算操作(解释器执行解释)
- 变量类型：变量名的类型；用来确定该变量的作用域以及生命周期(关键字修饰决定)
- 变量名：访问变量存储的数据；用来访问一段可读可写的连续内存空间(自定义命名)

其中数据类型属性将在数据类型一章内容介绍

在本章内容中主要介绍

- 变量名
- 变量类型
  - 本地变量
  - 局部变量
  - 环境变量
  - 位置变量
  - 特殊变量
  - 变量属性
  - 变量赋值

1.3.1 0x00 变量名

变量是通过变量名进行声明、定义、赋值和引用；变量存在于内存中，对于shell变量而言，设置或修改变量属性以及变量值时，不需要带`$`，只有引用变量的值时才使用`$`

变量名的本质是：一段可读可写的连续内存空间的别名

通过对变量名的引用就可以读写访问连续的内存空间

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`内嵌的关键字(可用`help`命令查看保留关键字)
- 不能使用shell命令

1.3.2 0x01 变量类型

shell脚本是弱类型解释型的语言，变量类型由不同关键字声明决定；根据变量类型可将变量分为：(变量类型即变量名的类型，它决定变量的作用域以及定义引用的方式)

- 本地变量
- 局部变量
- 环境变量
- 位置变量
- 特殊变量

- 变量属性

0x0100 本地变量

本地变量可以理解为全局变量，它的作用域为：只对当前shell进程有效，对其子shell以及其他shell都无效

该类型变量的声明定义方式为：`[set]Var_Name=Value`

- `set`关键字可以省略
- 等号左右没有空格；如果有空格就是进行比较运算符的比较运算
- 该变量可以声明定义在脚本的任何地方
- 变量Var_Name可以是任意数据类型

该类型变量的引用方式(获取变量的值)为：`$Var_Name`或`${Var_Name}`

- 可以在脚本的任意地方引用

该类型变量的赋值方式(修改变量的值)为：`Var_Name=Value`

- 在脚本中任意地方的赋值都会覆盖之前的变量值

该类型变量的撤销释放方式为：`unset Var_Name`

- 变量名前不加前缀$
- 撤销该变量后，引用该变量就会为空

需要注意的是：

- 如果使用`readonly`关键字修饰变量`Var_Name`，即`readonly Var_Name[=Value]`，此时将无法修改变量值也无法unset变量
- 不接收任何参数的`set`或者`declare`关键字命令，将会输出当前所有有效的本地变量、局部变量和环境变量

示例程序如下

```
#!/bin/bash

test_str="hello world"
readonly ro_str="test"
test_one(){
        echo "test_str in test_one is $test_str"
        test_str="happy"
        test_name="anony"
        unset test_set    # 撤销变量test_set，之后引用该变量就会为空
        echo "test_set in test_one is $test_set"
        ro_str="tset"     # 该变量被readonly修饰，不能修改其变量值，将会出现语法错误，直接退出函数，不执行下列命令
        echo "ro_str"     # 上述直接退出函数，该命令不会执行
}

test_two()
{
        echo "test_str in test_two is $test_str"
        echo "test_name in test_two is $test_name"
        echo "test_set in test_two is $test_set"

}

test_one
test_two
```

(continues on next page)

(续上页)

```
# 执行结果如下
# test_str in test_one is hello world
# test_set in test_one is                # echo显示为空
# ./demo.sh: line 9: ro_str: readonly variable
# test_str in test_two is happy
# test_name in test_two is anony
# test_set in test_two is                # echo显示为空
```

0x0101 局部变量

局部变量的作用域为：只对变量声明定义所在函数内有效

该类型变量的声明定义方式为：`loca Var_Name=Value`

- `local`关键字不能省略，否则就是本地全局变量
- 等号左右没有空格；如果有空格就是进行比较运算符的比较运算
- 该变量只能声明定义在函数体内，否则会语法报错
- 变量Var_Name可以是任意数据类型

该类型变量的引用方式(获取变量的值)为：`$Var_Name`或`${Var_Name}`

- 只能在声明定义的函数体内引用，其它地方引用将为空

该类型变量的赋值方式(修改变量的值)为：`Var_Name=Value`

该类型变量的撤销释放方式为：`unset Var_Name`

- 变量名前不加前缀`$`
- 撤销该变量后，引用该变量就会为空

需要注意的是：

- 如果使用`readonly`关键字修饰变量`Var_Name`，即`readonly Var_Name[=Value]`，此时将无法修改变量值也无法unset变量
- 不接收任何参数的`set`或者`declare`关键字命令，将会输出当前所有有效的本地变量、局部变量和环境变量

示例程序如下

```
#!/bin/bash

test_str="anony"
test_one(){
        local test_str="happy"    # 局部变量test_str会覆盖全局变量test_str
        local test_local="test"
        echo "test_str in test_one is $test_str"
        echo "test_local in test_one is $test_local"
        unset test_str           # 只会撤销局部变量test_str，不会撤销全局变量test_str
}

test_two()
{
        echo "test_str in test_two is $test_str"       # unset没有撤销全局变量test_str
        echo "test_local in test_two is $test_local"   # test_local是定义在test_one函
数中的局部变量，该处引用将会为空

}

test_one
test_two
```

(continues on next page)

(续上页)

```
# 执行结果如下
# test_str in test_one is happy
# test_local in test_one is test
# test_str in test_two is anony
# test_local in test_two is
```

0x0102 环境变量

环境变量可以用来

- 定义bash的工作特性
- 保存当前会话的属性信息

关于环境变量的生命周期和作用域可以参考：bash环境配置

shell环境变量有两种来源

- 系统环境变量
  - 该环境变量已经由bash定义初始化，不用重新声明定义，只要引用就可以
    - 使用`env`、`export`、`set`、`declare`或`printenv`可以查看当前用户的环境变量(包括系统环境变量和自定义环境变量)，以下列出部分bash默认系统环境变量(`set`和`declare`可以查看所有环境变量，其它三个命令只能查看部分环境变量)
      - `$BASH`：bash二进制程序文件的路径
      - `$BASH_SUBSHELL`：子shell的层次说明，说明用户在哪一个层次中
      - `$BASH_VERSION`：bash的版本
      - `$EDITOR`：指定默认编辑器
      - `$EUID`：有效的用户ID
      - `$UID`：当前用户的ID号
      - `$USER`：当前用户名
      - `$PATH`：自动搜索路径
      - `$LANG`：系统使用语系
      - `LOGNAME`：当前登录的用户
      - `$FUNCNAME`：当前函数的名称，在函数中引用想判断自己是什么函数
      - `$GROUPS`：当前用户所属的组
      - `$HOME`：当前用户的家目录
      - `$HOSTTYPE`：主机架构类型，用来识别系统硬件平台
      - `$MACHTYPE`：平台类型，系统平台依赖的编译平台
      - `$OSTYPE`：OS系统类型
      - `$IFS`：输入数据时的默认字段分隔符，默认是空白符(空格、制表符、换行符)
      - `$OLDPWD`：上次使用的目录
      - `$PWD`：当前目录
      - `$PPID`：父进程
      - `$PS1`：主提示符，即bash命令窗口提示符
      - `$PS2`：第二提示符，主要用于补充完全命令输入时的提示符

            - `$PS3`：第三提示符，用于select命令中
            - `$PS4`：第四提示符，当使用-X选项调用脚本时，显示的提示符，默认为+号
            - `$SECONDS`：当前脚本已经运行的时长，单位为秒
            - `$SHLVL`：shell的级别，bash被嵌入的深度
        * 系统环境变量常用大写字母表示
    - 系统环境变量作用域
        * 执行脚本前，原始系统环境变量对当前用户所有shell进程(包含不同终端bash进程以及其子shell进程)都有效
        * 执行脚本时，系统环境变量对当前shell进程以及子shell进程都有效
        * 执行脚本后
            - 如果使用source命令执行脚本，修改后的系统环境变量会覆盖之前的系统环境变量，但是修改后的变量值只对当前终端bash进程以及其子shell进程才有效；原始变量值依然对当前用户所有shell进程(包含不同终端bash进程以及其子shell进程)都有效
            - 如果使用`./demo.sh`和`bash demo.sh`执行脚本，修改后的系统环境变量不会覆盖之前的系统环境变量，即所以系统环境变量依然保持原值，依然对当前用户所有shell进程(包含不同终端bash进程以及其子shell进程)都有效
- 自定义环境变量
    - 该环境变量是使用`export`命令将全局变量或局部变量导出成环境变量，需要手动声明定义
        * 方式一：`export Var_Name=Value`
        * 方式二：`Var_Name=Value`、`export Var_Name`
        * 自定义环境变量名尽量避免与系统环境变量名冲突；等号左右没有空格；如果有空格就是进行比较运算符的比较运算
        * 变量`Var_Name`可以是全局变量或局部变量，也可以是任意数据类型
    - 自定义环境变量作用域
        * 执行脚本时，自定义环境变量才被声明定义，同时继承全局变量或局部变量的作用域
        * 执行脚本后
            - 如果使用`./demo.sh`和`bash demo.sh`执行脚本，自定义环境变量不会导出成系统环境变量，即脚本执行完胡，该类环境变量会自动撤销
            - 如果使用`source demo.sh`执行脚本，只有全局环境变量才能导出成bash环境变量，局部环境变量会自动被撤销；但是导出后的全局环境变量只对当前终端bash进程以及其子shell进程才有效

不管是系统环境变量还是自定义环境变量都可以通过以下方式进行引用(获取环境变量的值)：`$Var_Name`或`${Var_Name}`

- 在环境变量的作用域之内引用
- 变量名`Var_Name`可以是系统环境变量名，又可以是自定义环境变量名

不管是系统环境变量还是自定义环境变量都可以通过以下方式进行赋值(修改环境变量的值)：对当前shell进程来说通过该方式赋值修改的环境变量继承之前的作用域

- 方式一：`export Var_Name=Value`
- 方式二：`Var_Name=Value`、`export Var_Name`

不管是系统环境变量还是自定义环境变量都可以通过下列方式进行撤销释放：`unset Var_Name`

- 变量名前不加前缀`$`
- 撤销该变量后，引用该变量就会为空

需要注意的是：

- 如果使用readonly关键字修饰变量Var_Name，即readonly Var_Name[=Value]，此时将无法修改变量值也无法unset变量
- 不接收任何参数的set或者declare关键字命令，将会输出当前所有有效的本地变量、局部变量和环境变量

示例程序如下

```
#!/bin/bash

test_one(){
        PATH=./:$PATH             # 修改系统环境变量的值
        export PATH               # 导出系统环境变量使其生效
        export MYNAME="anony"     # 将全局变量导出成环境变量
        local MYSEX="man"         # 定义局部变量
        export MYSEX              # 将局部变量导出成环境变量
        export MYBLOG="blog"
        export MYAGE="22"
        echo "PATH in test_one is $PATH"       # 上述所有定义的环境变量都有效
        echo "MYNAME in test_one is $MYNAME"
        echo "MYSEX in test_one is $MYSEX"
        echo "MYBLOG in test_one is $MYBLOG"
        echo "MYAGE in test_one is $MYAGE"
        unset MYBLOG              # 撤销全局变量导出成的环境变量
        readonly MYAGE            # 将全局变量导出成的环境变量修改为只读变量
        MYAGE="23"                # 对只读变量进行赋值修改会造成语法错误
}

test_two()
{
        echo "PATH in test_two is $PATH"           # 系统变量的作用域
        echo "MYNAME in test_two is $MYNAME"       # 全局环境变量的作用域
        echo "MYSEX in test_two is $MYSEX"         # 局部环境变量的作用域
        echo "MYBLOG in test_two is $MYBLOG"       # 全局环境变量已经撤销
        echo "MYAGE in test_two is $MYAGE"         # 全局环境变量只读
}

test_one
test_two


# 执行结果如下
# PATH in test_one is ./:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/
↪bin
# MYNAME in test_one is anony
# MYSEX in test_one is man
# MYBLOG in test_one is blog
# MYAGE in test_one is 22
# ./demo.sh: line 20: MYAGE: readonly variable
# PATH in test_two is ./:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/
↪bin
# MYNAME in test_two is anony
# MYSEX in test_two is
# MYBLOG in test_two is
# MYAGE in test_two is 22
```

0x0103 位置变量

位置变量无需声明定义，直接引用即可；该变量也不能被赋值修改，甚至不能被unset撤销

位置变量是用来实现

- 在函数体外直接引用脚本的传入参数，它引用方式如下

  - `$0`：引用脚本名
  - `$1`：引用脚本的第1个传入参数
  - `$n`：引用脚本的第n个传入参数
- 在函数体内直接引用函数的传入参数，它引用方式如下
  - `$0`：引用脚本名
  - `$1`：引用函数的第1个传入参数
  - `$n`：引用函数的第n个传入参数

示例程序如下

```
#!/bin/bash
echo "script name is $0"

echo "the script first arg is $1"  # 引用脚本的第一个传入参数
test(){
        echo "script name is $0"
        echo "the func first arg in test is $1" # 引用函数的第一个传入参数，不是脚本的第一
个参数
}
test 26

# 执行结果如下：./test.sh 12
# script name is ./test.sh
# the script first arg is 12
# script name is ./test.sh
# the func first arg in test is 26
```

0x0104 特殊变量

特殊变量也无需声明定义，直接引用即可；该变量也不能被赋值修改，甚至不能被unset撤销

特殊变量的引用方式如下

- `$?`：引用上一条命令的执行状态返回值，状态用数字表示：0-255
  - `0`：表示成功
  - `1-255`：表示失败；需要注意的是`1/2/127/255`是系统预留的，自己写脚本时要避开与这些值重复
- `$$`：引用当前shell的PID。除了执行bash命令和shell脚本时，$$不会继承父shell的值，其他类型的子shell都继承
- `$BASHPID`：引用当前shell的PID，这和`$$`是不同的，因为每个shell的$BASHPID是独立的，而`$$`有时候会继承父shell的值
- `$!`：引用最近一次执行的后台进程PID，即运行于后台的最后一个作业的PID
- `$#`：引用所有位置参数的个数
- `$*`：引用所有位置参数的整体，即所有参数被当做一个字符串
- `$@`：引用所有单个位置参数，即每个参数都是一个独立的字符串
- `$_`：引用上一条命令的最后一个参数的值
- `$-`：引用传递给脚本的标记

示例程序如下

```
#!/bin/bash

echo '$# is:'$#
echo '$* is:'$*
echo '$@ is:'$@
echo '$! is:'$!
echo '$$ is:'$$
echo '$BASHPID is:'$BASHPID
echo '$? is:'$?
test(){
        echo '$# in func is:'$#
        echo '$* in func is:'$*
        echo '$@ in func is:'$@
        echo '$! in func is:'$!
        echo '$$ in func is:'$$
        echo '$BASHPID in func is:'$BASHPID
        echo '$? in func is:'$?
}
test 26 23 47

# 执行结果如下：[root@localhost ~]# ./test.sh 1 3 4 5 6 7
# $# is:6
# $* is:1 3 4 5 6 7
# $@ is:1 3 4 5 6 7
# $! is:
# $$ is:4002
# $BASHPID is:4002
# $? is:0
# $# in func is:3
# $* in func is:26 23 47
# $@ in func is:26 23 47
# $! in func is:
# $$ in func is:4002
# $BASHPID in func is:4002
# $? in func is:0
```

0x0105 变量属性

此处的变量属性是指数据类型和变量类型，这两个属性可以通过相关命令关键字进行修改，例如：

`Var_Name=Value`语句中声明定义的变量`Var_Name`默认的数据类型是字符串类型，变量类型是本地全局变量

- `local Var_Name`声明该变量为局部变量
- `export Var_Name`声明该变量为环境变量
- `declare -x Var_Name`声明该变量为环境变量
- `declare +x Var_Name`取消该变量的环境变量属性
- `declare -i Var_Name`声明该变量为整型变量
- `declare +i Var_Name`取消该变量的整型变量属性
- `declare -p Var_Name`显式指定变量被声明的类型
- `declare -r Var_Name`声明该变量为只读变量，不能撤销，不能修改，相当于readonly，只有当前进程终止才消失
- `declare +r Var_Name`取消该变量的只读变量属性

可以使用`man declare`查看`declare`命令的详细使用方法

0x0106 变量赋值

除了上述介绍的`Var_Name=Value`赋值方式，还有以下变量赋值的方式，以下赋值方式常用来给变量赋默认值

- `${var:-default}`：如果var没有声明或者声明了为空，则返回default代表的值；如果var声明了不为空，则返回var代表的值
- `${var-default}`：如果var没有声明，则返回default代表的值；如果var声明了但是为空，则返回null；如果var声明了不为空，则返回var代表的值
- `${var:+default}`：如果var没有声明或者声明了为空，不做任何操作，返回空；如果var声明了不为空，则返回default代表的值
- `${var:=default}`：如果var没有声明或者声明了为空，则返回default代表的值，并将default的值赋值给var；如果var声明了不为空，则返回var代表的值
- `${var:?default}`：如果var没有声明或者声明了为空，则以default为错误信息返回；如果var声明了不为空，则返回var代表的值

1.4 操作符

shell中常用的操作符有

- 引用操作符
- 算术操作符
- 条件测试操作符
  - 整数条件测试
  - 字符条件测试
  - 文件条件测试
- 逻辑操作符
- 括号操作符

1.4.1 0x00 引用操作符

引用操作符如下

- 变量引用：引用变量值，两者等效
  - `$variable`
  - `${variable}`
- 命令引用：引用命令的执行结果
  - `` `command` ``
  - `$(command)`
- 字符引用：引用字符串值
  - `''`：强引用，该操作符的优先级大于`$`，即不会进行变量替换，直接引用显示全部字符
  - `""`：弱引用，该操作符的优先级小于`$`，即先进行变量替换，然后再引用显示全部字符

1.4.2 0x01 算术操作符

组成算术表达式的操作符有

- 加：`+`、`+=`、`++`
- 减：`-`、`-=`、`--`
- 乘：`*`、`*=`
- 除：`/`
- 取余：`%`、`%=`

执行算术表达式的操作符有

- `$[算术表达式]`
- `$((算术表达式))`

1.4.3 0x02 条件测试操作符

条件测试有以下几种情况

- 整数条件测试
- 字符条件测试
- 文件条件测试

1.4.4 0x0200 整数条件测试

组成整数条件测试表达式的操作符有

- `-eq`：等于
- `-ne`：不等于
- `-le`：小于等于
- `-ge`：大于等于
- `-lt`：小于
- `-gt`:大于

执行整数条件测试表达式的操作符有

- `[ 整数条件测试表达式 ]`：前后有空格
- `[[ 整数条件测试表达式 ]]`：前后有空格

1.4.5 0x0201 字符条件测试

组成字符条件测试表达式的操作符有

- `>`：大于
- `<`：小于
- `==`：等于，等值比较
- `=~`：左侧是字符串，右侧是一个模式，判断左侧的字符串能否被右侧的模式所匹配：但是必须在`[[]]`中执行模式匹配。模式中可以使用行首、行尾锚定符，但是模式不要加引号，有时候可能不需要转义
- `!=`，`<>`：不等于

- `-n`: 判断字符串是否不空，不空为真，空则为假
- `-z`：判断字符串是否为空，空则为真，不空则假

执行字符条件测试表达式的操作符有

- `[ 字符条件测试表达式 ]`：前后有空格
- `[[ 字符条件测试表达式 ]]`：前后有空格

1.4.6 0x0202 文件条件测试

组成文件条件测试表达式的操作符有

- `-e file`：测试文件是否存在
- `-a file`：测试文件是否存在
- `-f file`：测试是否为普通文件
- `-d directory`： 测试是否为目录文件
- `-b file`：测试文件是否存在并且是否为一个块设备文件
- `-c file`：测试文件是否存在并且是否为一个字符设备文件
- `-h|-L file`：测试文件是否存在并且是否为符号链接文件
- `-p file`：测试文件是否存在并且是否为管道文件：
- `-S file`：测试文件是否存在并且是否为套接字文件：
- `-r file`：测试其有效用户是否对此文件有读取权限
- `-w file`：测试其有效用户是否对此文件有写权限
- `-x file`：测试其有效用户是否对此文件有执行权限
- `-s file`：测试文件是否存在并且不空
- `file1 -nt file2`：测试file1是否比file2更new一些
- `file1 -ot file2`：测试file1是否比file2更old一些

执行文件条件测试表达式的操作符有

- `[ 文件条件测试表达式 ]`：前后有空格
- `[[ 文件条件测试表达式 ]]`：前后有空格

1.4.7 0x03 逻辑操作符

逻辑操作符有

- 逻辑与`&&`
  - 真 `&&` 真 = 真
  - 真 `&&` 假 = 假
  - 假 `&&` 真 = 假
  - 假 `&&` 假 = 假
- 逻辑或`||`
  - 真 `||` 真 = 真
  - 真 `||` 假 = 真
  - 假 `||` 真 = 真

  - 假 || 假 = 假
- 逻辑非!
  - ! 真 = 假
  - ! 假 = 真

注意：各种编译语言对逻辑真、假的定义不同，在shell中，状态值为0代表真，状态值为非0代表假

1.4.8 0x04 括号操作符

括号操作符有以下几种

- ()
  - 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，即`(cmd1;cmd2;cmd3)`
  - 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，例如`tcsh`
  - 数组初始化：用来初始化数组
- (())
  - 执行算术表达式：这种算术表达式是整数型的计算，不支持浮点型
  - 执行进制运算：`$((16#5f))`结果为95(16进位转十进制)
  - 重定义变量值：`a=5;((a++))`结果a被重定义为6
  - 算术运算比较：双括号中的变量可以不使用$符号前缀，括号内支持多个表达式用逗号分开；比如可以直接使用`for((i=0;i<5;i++))`
- []
  - 执行测试表达式：前后有空格
  - 执行算术表达式：前后没有空格
- [[]]
  - 执行测试表达式：前后有空格
- {}
  - 命令组：括号中的命令将会在当前shell顺序执行，所以括号中的变量能被脚本余下的部分使用；括号中多个命令之间用分号隔开，最后一个命令后必须有分号，并且第一条命令和左括号之间必须用空格隔开，即`{ cmd1;cmd2;cmd3;}`
  - 变量引用：`${}`
  - 生成列表：`{a..d}.txt`表示`a.txt`、`b.txt`、`c.txt`、`d.txt`；在括号中，不允许有空白，除非这个空白被引用或转义
  - 扩展：`{a,b}.txt`表示`a.txt`和`b.txt`；在括号中，不允许有空白，除非这个空白被引用或转义

1.5 控制流程语句

和其它编程语言一样，shell的控制流程语句大体上也分为三种

- 顺序执行语句
- 条件执行语句

1.5.1 0x00 顺序执行语句

顺序执行语句是默认法则，即按照自上而下、自左往右的顺序逐条执行各命令，每执行一次就会得到对应的结果，然后退出该次执行操作

1.5.2 0x01 条件执行语句

条件执行语句会根据判断条件选择符合条件的分支执行对应的`cmd_list`命令列表，执行完命令后就会退出该分支；条件执行语句有以下几种

- *if*条件语句
- *case*条件语句
- *select*条件语句

0x0100 if条件语句

`if`条件语句的语法结构如下(使用`help if`命令可以查看)

```
if TEST_COMMANDS; then
        COMMANDS_LIST;
[elif TEST_COMMANDS; then
        COMMANDS_LIST;]
...
[else
        COMMANDS_LIST;]
fi
```

其执行逻辑是

- **1.** 先执行`if`分支下的`TEST_COMMANDS`条件测试命令，如果执行完的状态返回值为非`0`，则执行第2步；如果执行完的状态返回值为`0`，即`TEST_COMMANDS`条件测试命令执行成功，则执行该分支下的`COMMANDS_LIST`命令列表，执行完后就直接退出，此时整个if语句结构体的状态返回值取决于`COMMANDS_LIST`命令列表中最后一个命令的状态返回值
- **2.**如果存在`elif`分支，则按照第一步的流程依次执行`elif`分支下的`TEST_COMMANDS`条件测试命令，如果没有一个`elif`分支的状态返回值为`0`，则执行第3步；如果存在一个`elif`分支的状态返回值为`0`，即该分支下的`TEST_COMMANDS`条件测试命令执行成功，则执行该分支下

的COMMANDS_LIST命令列表，执行完后就直接退出，此时整个if语句结构体的状态返回值取决于COMMANDS_LIST命令列表中最后一个命令的状态返回值

- **3.**如果else分支不存在，那么整个if语句结构体的状态返回值为0；如果存在else分支，则执行该分支下的COMMANDS_LIST命令列表，执行完后就直接退出，此时整个if语句结构体的状态返回值取决于COMMANDS_LIST命令列表中最后一个命令的状态返回值

在整个if语句结构体中有两个地方需要注意

- COMMANDS_LIST：表示待执行的命令列表，即一系列shell命令的集合，类型格式多种多样，在一系列示例代码中可见一斑
  - 注意：在命令列表中不能使用 () 操作符改变优先级，它的作用是让括号内的语句成为命令列表进入子shell中执行，它的具体作用可参考：括号操作符
- TEST_COMMANDS：表示条件测试命令，即通过引用条件测试命令的执行状态返回值是否为0来判断是否执行上述COMMANDS_LIST命令列表；**这里需要特别注意的是，和其它语言不通，shell的条件测试命令只有以下三种类型**
  - 命令执行：命令本身执行后就会产生对应的执行状态返回值，所以可以直接用来做条件判断
    - 此时不能使用‘‘操作符来引用命令，因为该操作引用的是命令的执行结果，而不是命令的执行状态返回值
    - 通常是直接使用命令，然后在命令后面添加s&> /dev/null，表示将命令的执行结果重定向至/dev/null，只引用其状态返回值；例如：`if grep "^root" /etc/passwd &> /dev/null; then`
  - 执行条件测试表达式：在shell中，条件测试表达式是由条件测试操作符以及对应的操作数组成，详细介绍可参考下列：条件测试表达式，执行条件测试表达式有以下三种格式
    - `test Test_Expression`：通过test命令执行
    - `[ Test_Expression ]`：通过[]操作符执行，注意Test_Expression 前后有空格
    - `[[ Test_Expression ]]`：通过[[]]操作符执行，注意Test_Expression 前后有空格
  - 组合条件测试：即对多个命令执行状态返回值或者执行条件测试表达式状态返回值做逻辑运算，组合条件测试有以下三种格式
    - 逻辑与操作：只有当&&操作符两边执行结果都为真(状态值为0)，最后组合条件测试结果才为真(状态值为0)
      - `[ Test_Expression1 ] && [ Test_Expression2 ]`：此处使用[]或[[]]都行
      - `COMMAND &> /dev/null && [ Test_Expression2 ]`：此处使用[]或[[]]都行
      - `COMMAND1 &> /dev/null && COMMAND2 &> /dev/null &&`
      - `[ Test_Expression1 -a Test_Expression2 ]`：此处使用[]或[[]]都行
      - `[[ Test_Expression1 && Test_Expression2 ]]`：此处只能使用[[]]操作符，因为&&运算符不允许用于[]操作符中
    - 逻辑或操作：只要||操作符两边执行结果有一个为真(状态值为0)，最后组合条件测试结果就为真(状态值为0)
      - `[ Test_Expression1 ] || [ Test_Expression2 ]`：此处使用[]或[[]]都行
      - `COMMAND &> /dev/null || [ Test_Expression2 ]`：此处使用[]或[[]]都行
      - `COMMAND1 &> /dev/null || COMMAND2 &> /dev/null &&`
      - `[ Test_Expression1 -0 Test_Expression2 ]`：此处使用[]或[[]]都行

        - `[[ Test_Expression1 || Test_Expression2 ]]`：此处只能使用`[[]]`操作符，因为`||`运算符不允许用于`[]`操作符中

    * 逻辑非操作：对`!`右侧执行结果取反

        - `! [ Test_Expression ]`：此处使用`[]`或`[[]]`都行
        - `! COMMAND1 &> /dev/null`
        - `! ([ Test_Expression1 ] || [ Test_Expression2 ])`：此处相当于`! [ Test_Expression1 ] && ! [ Test_Expression2 ]`
        - `! ([ Test_Expression1 ] && [ Test_Expression2 ])`：此处相当于`! [ Test_Expression1 ] || ! [ Test_Expression2 ]`

    * 注意：**非的优先级大于与，与的优先级大于或**

示例代码如下

- 输出两个传入参数中的最大值

```
#!/bin/bash
if [ $# -lt 2 ]; then
        echo "`basename $0` arg1 arg2"
        exit 1
fi
if [ $1 -gt $2 ]; then
        echo "the max num is $1"
else
        echo "the max num is $2"
fi
```

- 计算1~200之间偶数之和

```
#!/bin/bash

declare -i sum=0
for i in {1..200};do
        if [ $[$i%2] -eq 0 ]; then
                let sum+=$i
        fi
done

echo "the sum is : $sum"
```

- 让用户输入一个用户名，先判断该用户是否存在，不存在，则以7为退出码；如果存在，判断用户的shell是否为`/bin/bash`，如果是，则显示为`Bash User`，退出码为0，否则显示为`Not Bash User`，退出码为1

```
#!/bin/bash

read -p "please input username: " username

echo $username
if ! grep "^$username\>" /etc/passwd &> /dev/null; then
        echo "User not exist"
        exit 7
elif [[ `grep "^$username\>" /etc/passwd | cut -d: -f7` =~ /bin/bash ]];then
        echo "Bash User"
        exit 0
else
        echo "Not Bash User"
        exit 1
fi
```

- 统计输入文件的空白行数

```
#!/bin/bash

read -p "Enter a file path: " filename

if grep "^$" $filename &> /dev/dull; then
        linesCount=`grep "^$" $filename | wc -l`
        echo "$filename has $linesCount space lines"
else
        echo "$filename has no space linse"
fi
```

0x0101 case条件语句

case条件语句的语法结构如下(使用help case命令可以查看)

```
case WORD in
        PATTERN1)
                COMMANDS_LIST
                ;;
        PATTERN2)
                COMMANDS_LIST
                ;;
        PATTERN3)
                COMMANDS_LIST
                ;;
        ...
esac
```

其执行逻辑是：WORD依次匹配PATTERN1、PATTERN2、PATTERN3.....；如果所有模式都没有匹配上，则直接退出case语句，此时执行状态返回值为0；如果匹配上任意一个PATTERN就执行该分支下面的COMMANDS_LIST命令列表，执行完后就直接退出，此时整个case语句结构体的状态返回值取决于COMMANDS_LIST命令列表中最后一个命令的状态返回值；模式的匹配优先级是PATTERN1> PATTERN2> PATTERN3> ......

在以上结构中，有以下几点需要注意

- case中的每个小分支都以双分号;;结尾，表示执行完该分支后直接退出case语句；但最后一个小分句的双分号可以省略。实际上，小分句除了使用;;结尾，还可以使用;&和;;&结尾，只不过意义不同，如下
  - ;;符号表示小分支执行完成后立即退出case语句
  - ;&符号表示继续执行下一个小分支中的COMMANDS_LIST部分，而无需进行匹配动作，并由此小分支的结尾符号来决定是否继续操作下一个小分句
  - ;;&符号表示继续向后(不止是下一个，而是一直向后)匹配小分支，如果匹配成功，则执行对应小分支中的COMMANDS_LIST部分，并由此小分支的结尾符号来决定是否继续向后匹配
- 每个小分支中的PATTERN部分都使用括号 () 包围，只不过左括号 (不是必须的
- 一般最后一个小分支使用的PATTERN是*，表示无法匹配前面所有小分支时，将匹配该小分支；用来避免case语句无法匹配的情况，在shell脚本中，此小分支一般用于提示用户脚本的使用方法，即给出脚本的Usage

这里也需要说明下以下两个关键组成成分

- WORD：一般是字符串类型
- PATTERN：该模式支持通配符机制(注意不是正则表达式)
  - *：匹配任意长度的任意字符
  - ?：匹配单个任意字符
  - []：匹配指定字符范围内的任意单个字符，不区分大小写

        * [a-z]：不区分大小写，可以匹配大写字母
        * [A-Z]：不区分大小写，可以匹配小写字母
        * [0-9]：匹配0到9任意单个数字
        * [a-z0-9]：匹配单个字母或数字
        * [[:upper:]]：匹配单个大写字母
        * [[:lower:]]：匹配单个小写字母
        * [[:alpha:]]：匹配单个大写或小写字母
        * [[:digit:]]：匹配单个数字
        * [[:alnum:]]：匹配单个字母或数字
        * [[:space:]]：匹配单个空格字符
        * [[:punct:]]：匹配单个标点符号
    - [^]：匹配指定字符范围外的任意单个字符
        * [^a-z]：匹配字母之外的单个字符
        * [^A-Z]：匹配字母之外的单个字符
        * [^0-9]：匹配数字之外的单个字符
        * [^a-z0-9]：匹配字母和数字之外的单个字符
        * [^[:upper:]]：匹配大写字母之外的单个字符
        * [^[:lower:]]：匹配小写字母之外的单个字符
        * [^[:alpha:]]：匹配字母之外的单个字符
        * [^[:digit:]]：匹配数字之外的单个字符
        * [^[:alnum:]]：匹配字母和数字之外的单个字符
        * [^[:space:]]：匹配空格字符之外的单个字符
        * [^[:punct:]]：匹配标点符号之外的单个字符
    - |：用来分隔上述*、?、[]、[^]通配元字符；例如([yY] | [yY][eE][sS])表示即可以输入单个字母的y或Y，还可以输入yes三个字母的任意大小写格式

示例代码如下

```
#!/bin/bash
set -- y

case "$1" in
    ([yY]|[yY][eE][sS])
        echo yes;&
    ([nN]|[nN][oO])
        echo no;;
    (*)
        echo wrong;;
esac

# 执行结果如下
# yes
# no
```

其中set -- string_list的作用是将string_list按照IFS分隔后分别赋值给位置变量$1、$2、$3...，因此此处是为$1赋值字符y

在此示例中，$1能匹配第一个小分支，但第一个小分支的结尾符号为;&，所以无需判断地直接执行第二个小分支的echo no，但第二个小分支的结尾符号为;;，于是直接退出case语句。因此，即使$1无法匹配第二个小分句，case语句的结果中也输出了yes和no

```
#!/bin/bash
set -- y

case "$1" in
    ([yY]|[yY][eE][sS])
        echo yes;;&
    ([nN]|[nN][oO])
        echo no;;
    (*)
        echo wrong;;
esac

# 执行结果如下
# yes
# wrong
```

在此示例中，$1能匹配第一个小分支，但第一个小分支的结尾符号为;;&，所以继续向下匹配，第二个小分支未匹配成功，直到第三个小分支才被匹配上，于是执行第三个小分支中的echo wrong，但第三个小分支的结尾符号为;;，于是直接退出case语句。所以，结果中输出了yes和wrong

0x0102 select条件语句

select条件语句是一种可以提供菜单选择的条件判断语句，其语法结构如下(使用help select命令可以查看)

```
select NAME [in WORDS ... ;] do
        COMMANDS_LIST
done
```

其执行逻辑是

- **1.**如果in WORDS部分存在，则会将WORDS部分根据环境变量IFS进行分割，对分割后的每一项依次进行编号作为菜单项输出；如果in WORDS部分不存在，则使用in $@代替，即将位置变量的内容进行编号作为菜单项输出
- **2.**当输入内容能够匹配输出菜单序号时，该序号将会保存到变量NAME中，该序号对应的内容将会保存到特殊变量REPLY中；当输入内容不能匹配输出菜单序号时，比如随便几个字符，变量NAME将会被置空，特殊变量REPLY将会保存所有输入内容
- **3.**每次输入选择保存NAME和REPLY变量后，就会直接执行COMMANDS_LIST部分；如果没有break命令，则会跳回第一步，循环重复执行，直到遇到break命令或者ctrl+c退出select语句

示例代码如下

```
#!/bin/bash

select fname in cat dog sheep mouse;do
        echo your choice: \"$REPLY\) $fname\"
done

# 执行结果如下
[root@localhost ~]# ./test.sh
1) cat
2) dog
3) sheep
4) mouse
#? 1                        # 输入序号1
```

(continues on next page)

(续上页)

```
your choice: "1) cat"
#? 2                        # 输入序号2
your choice: "2) dog"
#? 3                        # 输入序号3
your choice: "3) sheep"
#? 4                        # 输入序号4
your choice: "4) mouse"
#? 5                        # 输入序号5，没有该序号值，所有fname变量置空
your choice: "5) "
#? anony                    # 输入anony，不是序号值，所以fname变量置空
your choice: "anony) "
#? ^C                       # select语句中没有break命令，通过ctrl+c退出select语句
```

0x0103 条件测试表达式

条件测试表达式有以下几种类型

- 整数测试表达式
- 字符测试表达式
- 文件测试表达式

整数测试表达式的格式为：`NUM1 操作符 NUM2`

- NUM1和NUM2是整数，可以直接是整数值(例如：2)，可以是变量引用(例如：$#)，也可以是算术运算得到的值(参考算术运算)
- 整数测试操作符有
  - -eq：等于
  - -ne：不等于
  - -le：小于等于
  - -ge：大于等于
  - -lt：小于
  - -gt:大于

字符测试表达式的格式有两种格式

- 双目测试格式：`STR1 双目操作符 STR2`
  - STR1和STR2是字符串，shell中默认数据类型是字符串，即不带""默认都会被当做字符串类型；但是在此处，必须使用""(除非是模式匹配中的模式字符串，才不用引号)
  - 双目测试操作符有
    - >：表示左边的字符串大于右边的字符串
    - <：表示左边的字符串小于右边的字符串
    - ==：表示左边的字符串等于右边的字符串
    - !=、<>：表示左右两边的字符串完全不相等
    - =~：左侧是普通字符串，右侧是一个模式字符串，用来判断左侧的字符串能否被右侧的模式所匹配：但是必须在[[]]中才能执行模式匹配；模式中可以使用行首、行尾锚定符，但是**模式不要加引号**，有时候可能不需要转义，具体模式书写格式可参考正则表达式
- 单目测试格式：单目操作符 STR
  - STR是字符串，shell中默认数据类型是字符串，即不带""默认都会被当做字符串类型；但是在此处，必须使用""

  - 单目测试操作符有
    * -n: 判断字符串是否不空，不空为真，空则为假
    * -z：判断字符串是否为空，空则为真，不空则假

文件测试表达式的格式也有两种

- 单目测试格式：单目操作符 FILE
  - FILE是文件名，一般使用绝对路径
  - 单目操作符有
    * -e FILE：测试文件是否存在
    * -a FILE：测试文件是否存在
    * -f FILE：测试是否为普通文件
    * -d FILE： 测试是否为目录文件
    * -b FILE：测试文件是否存在并且是否为一个块设备文件
    * -c FILE：测试文件是否存在并且是否为一个字符设备文件
    * -h|-L FILE：测试文件是否存在并且是否为符号链接文件
    * -p FILE：测试文件是否存在并且是否为管道文件：
    * -S FILE：测试文件是否存在并且是否为套接字文件：
    * -r FILE：测试其有效用户是否对此文件有读取权限
    * -w FILE：测试其有效用户是否对此文件有写权限
    * -x FILE：测试其有效用户是否对此文件有执行权限
    * -s FILE：测试文件是否存在并且不空
- 双目测试格式：FILE1 双目操作符 FILE2
  - FILE1和FILE2是文件名，一般使用绝对路径
  - 双目操作符有
    * FILE1 -nt FILE2：测试FILE1是否比FILE2更new一些
    * FILE1 -ot FILE2：测试FILE1是否比FILE2更old一些

1.5.3 0x02 循环执行语句

循环执行语句会根据判断条件循环多次执行对应的循环体cmd_list命令列表，当判断条件不满足时就会退出该循环体，需要注意的是：**循环必须有退出条件，否则将陷入死循环**；循环执行语句有以下几种

- *for*循环语句
- *while*循环语句
- *until*循环语句

1.5.4 0x0200 for循环语句

for循环语句在shell脚本中应用及其广泛，它有两种语法结构(使用help for命令可以查看)

```
# 结构一
for NAME [in WORDS ... ] ; do
        COMMANDS_LIST
done


# 结构二
for (( exp1; exp2; exp3 )); do
        COMMANDS_LIST
done
```

语法结构一的执行逻辑是

- **1.**如果`in WORDS`部分存在，则会将`WORDS`部分根据环境变量`IFS`进行分割，依次赋值给变量`NAME`(**如果WORD中使用引用包围了某些单词，则将引号包围的内容分隔为一个单词**)；如果`in WORDS`部分不存在，则默认使用`in $@`代替，即将位置变量依次赋值给变量`NAME`
- **2.**`NAME`变量每被赋值一次，就会执行一次循环体`COMMANDS_LIST`，直到第一步中所有分隔部分给`NAME`变量赋值完毕，才会结束循环
- **3.**如果在循环体中遇到`continue`命令，则退出当前for循环，直接进行下一for循环；如果遇到`break`命令，则直接退出for循环结构体
- **4.**整个for语句结构体的状态返回值取决于退出整个for循环结构体时最后一个命令的执行状态返回值

语法结构二的执行逻辑是

- **1.**首先执行算术表达式`exp1`
- **2.**然后判定算术表达式`exp2`的状态返回值是否为`0`，如果为`0`则执行循环体`COMMANDS_LIST`，执行完之后，执行算术表达式`exp3`，然后再次判定算术表达式`exp2`的状态返回值是否为`0`；直到其状态返回值为非`0`才退出整个for循环结构体，否则就会循环执行第2步，此时整个for循环的状态返回值为退出整个for循环结构体时最后一个算术表达式`exp2`的状态返回值
- **3.**如果在循环体中遇到`continue`命令，则退出当前for循环，直接进行下一for循环(即直接执行上述第二步)，此时整个for循环的状态返回值为退出整个for循环结构体时最后一个算术表达式`exp2`的状态返回值；如果遇到`break`命令，则直接退出整个for循环结构体，此时整个for语句结构体的状态返回值取决于退出整个for循环结构体时最后一个命令的执行状态返回值

`for`循环语句的循环退出机制有：

- `continue`：跳出当前循环进入下一循环
- `break[n]`：默认跳出整个循环；n可以指定跳出几层循环
- 列表遍历：使用一个变量去遍历给定列表中的每个元素(以环境变量`IFS`为分隔符)，在每次变量赋值时执行一次循环体，直至赋值完成所有元素退出循环
- 算术执行：引用算术表达式的执行状态返回值来判断是否退出整个循环

for循环语句适用于已知循环次数的场景

语法结构一中的`WORDS`有多种表现形式

- 列表变量
  - 数字列表：数字列表示例代码
    - `{start..end}`
    - `` `seq start step end` ``
  - 其它列表：其它列表示例代码
    - 使用空白分隔符直接给出列表
    - 使用文件名通配机制生成列表
    - 使用命令生成列表

- 数组变量
  - 普通数组：普通数组示例代码
  - 关联数组：关联数组示例代码

语法结构二种的exp只支持数学计算和比较，因为它被包含在执行算术运算的(())操作符之内

- exp1：一般是赋值表达式，例如for ((i=1,j=3;i<=3 && j>=2;++i,--j));do echo $i $j;done
- exp2：一般是比较表达式，例如for ((i=1,j=3;i<=3 && j>=2;++i,--j));do echo $i $j;done，比较表达式可参考数值类型比较运算for循环部分
- exp3：一般是计算表达式，例如for ((i=1,j=3;i<=3 && j>=2;++i,--j));do echo $i $j;done，计算表达式可参考数值类型算术运算

示例代码如下

- 计算当前系统所有用户ID之和

```
#!/bin/bash

declare -i uidSum=0

for i in `cut -d: -f3 /etc/passwd`; do
        uidSum=$[$uidSum+$i]
done

echo "the UIDSum is: $uidSum"
```

- 新建用户tmpuser1-tmpuser10，并计算它们的id之和

```
#!/bin/bash

declare -i uidSum=0

for i in {1..10}; do
        useradd tmpuser$i
        let uidSum+=`id -u tmpuser$i`
done

echo "the UIDSum is: $uidSum"
```

- 输出1-10之间的所有偶数

```
#!/bin/bash

for ((i=1;i<=10;i++));do
        let tmp=i%2
        if [ $tmp -eq 0 ]; then
                echo $i
        fi
done
```

1.5.5 0x0201 while循环语句

while循环语句的语法结构如下(使用help while命令可以查看)

```
while TEST_COMMANDS_LIST; do
        COMMANDS_LIST
done
```

其执行逻辑是

- **1.**先执行TEST_COMMANDS_LIST条件测试命令，如果其最后一个命令的执行状态返回值为0，则执行循环体COMMANDS_LIST，执行完后，再次执行TEST_COMMANDS_LIST条件测试命令，直到其最后一个名的状态返回值为非0才会退出整个while循环体，否则将一直循环执行该步，此时整个while循环的状态返回值为退出循环结构体时最后一个TEST_COMMANDS_LIST条件测试命令的最后一个命令的状态返回值
- **2.**如果在循环体中遇到continue命令，则退出当前while循环，直接进行下一while循环(即直接执行上述第一步)，此时整个while循环的状态返回值为退出循环结构体时最后一个TEST_COMMANDS_LIST条件测试命令的最后一个命令的状态返回值；如果遇到break命令，则直接退出整个while循环结构体，此时整个while语句结构体的状态返回值取决于退出整个循环结构体时最后一个命令的执行状态返回值

在上述while循环语句结构中需要注意的是

- COMMANDS_LIST：表示待执行的命令列表(也称为while循环体)，即一系列shell命令的集合，类型格式多种多样，在一系列示例代码中可见一斑
  - 注意：在命令列表中不能使用()操作符改变优先级，它的作用是让括号内的语句成为命令列表进入子shell中执行，它的具体作用可参考：括号操作符
- TEST_COMMANDS_LIST：表示条件测试命令，即通过引用条件测试命令的执行状态返回值是否为0来判断是否执行上述COMMANDS_LIST循环体；**这里需要特别注意的是，和其它语言不通，shell的条件测试命令只有以下三种类型**
  - 命令执行：命令本身执行后就会产生对应的执行状态返回值，所以可以直接用来做条件判断
    - 此时不能使用‘‘操作符来引用命令，因为该操作引用的是命令的执行结果，而不是命令的执行状态返回值
    - 通常是直接使用命令，然后在命令后面添加s&> /dev/null，表示将命令的执行结果重定向至/dev/null，只引用其状态返回值；例如：`if grep "^root" /etc/passwd &> /dev/null; then`
  - 执行条件测试表达式：在shell中，条件测试表达式是由条件测试操作符以及对应的操作数组成，详细介绍可参考下列：条件测试表达式，执行条件测试表达式有以下三种格式
    - `test Test_Expression`：通过test命令执行
    - `[ Test_Expression ]`：通过[]操作符执行，注意Test_Expression 前后有空格
    - `[[ Test_Expression ]]`：通过[[]]操作符执行，注意Test_Expression 前后有空格
  - 组合条件测试：即对多个命令执行状态返回值或者执行条件测试表达式状态返回值做逻辑运算，组合条件测试有以下三种格式
    - 逻辑与操作：只有当&&操作符两边执行结果都为真(状态值为0)，最后组合条件测试结果才为真(状态值为0)
      - `[ Test_Expression1 ] && [ Test_Expression2 ]`：此处使用[]或[[]]都行
      - `COMMAND &> /dev/null && [ Test_Expression2 ]`：此处使用[]或[[]]都行
      - `COMMAND1 &> /dev/null && COMMAND2 &> /dev/null &&`
      - `[ Test_Expression1 -a Test_Expression2 ]`：此处使用[]或[[]]都行
      - `[[ Test_Expression1 && Test_Expression2 ]]`：此处只能使用[[]]操作符，因为&&运算符不允许用于[]操作符中
    - 逻辑或操作：只要||操作符两边执行结果有一个为真(状态值为0)，最后组合条件测试结果就为真(状态值为0)
      - `[ Test_Expression1 ] || [ Test_Expression2 ]`：此处使用[]或[[]]都行

        - COMMAND &> /dev/null || [Test_Expression2]：此处使用[]或[[]]都行
        - COMMAND1 &> /dev/null || COMMAND2 &> /dev/null &&
        - [Test_Expression1 -0 Test_Expression2]：此处使用[]或[[]]都行
        - [[Test_Expression1 || Test_Expression2]]：此处只能使用[[]]操作符，因为||运算符不允许用于[]操作符中
    * 逻辑非操作：对!右侧执行结果取反
        - ! [Test_Expression]：此处使用[]或[[]]都行
        - ! COMMAND1 &> /dev/null
        - ! ([Test_Expression1] || [Test_Expression2])：此处相当于! [Test_Expression1] && ! [Test_Expression2]
        - ! ([Test_Expression1] && [Test_Expression2])：此处相当于! [Test_Expression1] || ! [Test_Expression2]
    * 注意：**非的优先级大于与，与的优先级大于或**

while循环语句的循环退出机制有：

- continue：跳出当前循环进入下一循环
- break[n]：默认跳出整个循环；n可以指定跳出几层循环
- 条件测试：此时为了避免死循环，TEST_COMMANDS_LIST条件测试里必须有控制循环次数的变量；COMMANDS_LIST循环体里必须有改变条件测试中用于控制循环次数变量的值操作

while循环语句适用于循环次数未知的场景，示例代码如下

```
#!/bin/bash

let i=1,sum=0

# 此处TEST_COMMANDS_LIST有多个命令
# 需要注意的是[ $i -le 10 ]才是判定是否退出循环的命令
# 而echo $i命令的执行状态返回结果跟退出循环无关
while echo $i;[ $i -le 10 ]; do
        let sum=sum+i;
        let ++i
done

echo $sum
```

对于while循环，有另外两种常见的用法

- 实现无限死循环

```
# 格式一：TEST_COMMANDS_LIST部分使用:
while :; do
        COMMANDS_LIST
done

# 格式二：TEST_COMMANDS_LIST部分使用true
while true; do
        COMMANDS_LIST
done
```

- 实现read命令从标准输入中按行读取值，然后保存到变量line中(既然是read命令，就可以保存到多个变量中)，读取一行就是一个循环

```
##############################方法一#######################################
# 标准输入来自于管道
# 每读取一行内容就会进入一次while循环，此处有两行内容所以进行两次while循环
# 此处通过-e选项实现多行输入
# 读取的每行内容将会按照IFS分隔，并赋值给两个变量
declare -i linenum=0
echo -e "abc xyz\n2abc 2xyz" | while read field1 field2; do
        echo $field1
        echo $field2
        linenum+=1
done
echo "there are $linenum lines"
# 此处使用的是管道符号，这样使得while语句在子shell中执行，这也意味着while语句内部设置的变量、数
组、函数等在while循环外部都不再生效
# 执行结果如下
# abc
# xyz
# 2abc
# 2xyz
# there are 0 lines


##############################方法二#######################################
# 标准输入来自于重定向
# 每读取一行内容就会进入一次while循环，此处有两行内容所以进行两次while循环
# 此处通过EOF标志实现多行输入
# 读取的每行内容将会按照IFS分隔，并赋值给两个变量
declare -i linenum=0
while read field1 field2; do
        echo $field1
        echo $field2
        linenum+=1
done << EOF
abc xyz
2abc 2xyz
EOF
echo "there are $linenum lines"
# 此处while语句内部设置的变量、数组、函数等在while循环外部依然生效
# 执行结果如下
# abc
# xyz
# 2abc
# 2xyz
# there are 2 lines


##############################方法三#######################################
# 标准输入来自于重定向
# 常用来重定向文件输入，读取文件内容
# 每读取文件一行内容，就会进入一次while循环，直到读完文件尾部退出循环
while read line; do
        echo $line
done < /etc/passwd


##############################方法四#######################################
# 读取文件的另一种写法
exec </etc/passwd;while read line; do
echo $line
done
```

关于read命令从标准输入中按行读取值的几种while循环的写法，还有一点需要注意

- 方法一传递数据的源是一个单独的进程，它传递的数据只要被while循环读取一次，所有剩余的数据就会被丢弃
- 方法二、三、四是以实体文件作为重定向传递的数据，while循环读取一次之后并不会丢弃剩余数据，直到数据完全读取完毕

也就是说当标准输入是非实体文件时(如管道传递、独立进程产生的)只供一次读取；当标准输入是直接重定向实体文件时，可供多次读取，但只要某一次读取了该文件的全部内容就无法再提供读取

回到IO重定向上，无论什么数据资源，只要被读取完毕或者主动丢弃，那么该资源就不可再得

- 对于独立进程传递的数据(管道左侧进程产生的数据、进程替换产生的数据)，它们都是虚拟数据，要不被一次读取完毕，要不读一部分剩余的丢弃，这是真正的一次性资源；其实这也是进程间通信时数据传递的现象
- 实体文件重定向传递的数据，只要不是一次性被全部读取，它就是可再得资源，直到该文件数据全部读取结束，这是伪一次性资源

大多数情况下，独立进程传递的数据和文件直接传递的数据并没有什么区别，但有些命令可以标记当前读取到哪个位置，使得下次该命令的读取动作可以从标记位置处恢复并继续读取，特别是这些命令用在循环中时。这样的命令有`head -n N`和`grep -m`，经测试，`tail`并没有位置标记的功能，因为`tail`读取的是后几行，所以它必然要读取到最后一行并计算要输出的行，所以`tail`的性能比`head`要差

- 示例一：通过管道将实体文件的内容传递给head

```
#!/bin/bash
declare -i i=0

cat /etc/passwd | while head -n 2; [[ $i -le 3 ]]; do
        echo $i
        let ++i
done

# 执行结果如下
# root:x:0:0:root:/root:/bin/bash
# bin:x:1:1:bin:/bin:/sbin/nologin
# 0
# 1
# 2
# 3
```

- 示例二：将实体文件重定向传递给head

```
#!/bin/bash
declare -i i=0

while head -n 2; [[ $i -le 3 ]]; do
        echo $i
        let ++i
done < /etc/passwd

# 执行结果如下
# root:x:0:0:root:/root:/bin/bash
# bin:x:1:1:bin:/bin:/sbin/nologin
# 0
# daemon:x:2:2:daemon:/sbin:/sbin/nologin
# adm:x:3:4:adm:/var/adm:/sbin/nologin
# 1
# lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
# sync:x:5:0:sync:/sbin:/bin/sync
# 2
# shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
# halt:x:7:0:halt:/sbin:/sbin/halt
```

(continues on next page)

(续上页)

```
# 3
# mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
# operator:x:11:0:operator:/root:/sbin/nologin
```

分析上述结果可以看到

- 示例一中：本该head应该每次读取2行，但实际执行结果中显示总共就只读取了2行
- 示例二中：head每次读取2行，而且每次读取的两行是不同的，后一次读取的两行是从前一次读取结束的地方开始的，这是因为head有读取到指定行数后做上位置标记的功能

要想确定命令、工具是否具有做位置标记的能力，只需像下面例子一样做个简单的测试。以head和sed为例，即使sed的q命令能让sed匹配到内容就退出，但却不做位置标记，而且数据资源使用一次就丢弃

```
[root@localhost ~]# (head -n 2;head -n 2) </etc/fstab

#
# /etc/fstab
# Created by anaconda on Sun May 27 09:35:43 2018
[root@localhost ~]# (sed -n /default/'{p;q}' ;sed -n /default/'{p;q}') </etc/fstab
/dev/mapper/centos-root /                       xfs     defaults        0 0
[root@localhost ~]#
```

其实在实际应用过程中，这根本就不是个问题，因为搜索和处理文本数据的工具虽然不少，但绝大多数都是用一次文本就丢一次，几乎不可能因此而产生问题。之所以说这么多废话，主要是想说上面的read读取数据while写法中，管道传递数据是使用最广泛的写法，但其实也是最烂的一种

1.5.6 0x0202 until循环语句

until循环语句的语法结构如下(使用help until命令可以查看)

```
until TEST_COMMANDS_LIST; do
        COMMANDS_LIST
done
```

until循环和while循环的执行思路大致相同，只不过效果相反

- **1.**先执行TEST_COMMANDS_LIST条件测试命令，如果其最后一个命令的执行状态返回值为非0，则执行循环体COMMANDS_LIST，执行完后，再次执行TEST_COMMANDS_LIST条件测试命令，直到其最后一个命令的状态返回值为0才会退出整个until循环体，否则将一直循环执行该步，此时整个until循环的状态返回值为退出循环结构体时最后一个TEST_COMMANDS_LIST条件测试命令的最后一个命令的状态返回值
- **2.**如果在循环体中遇到continue命令，则退出当前until循环，直接进行下一until循环(即直接执行上述第一步)，此时整个until循环的状态返回值为退出循环结构体时最后一个TEST_COMMANDS_LIST条件测试命令的最后一个命令的状态返回值；如果遇到break命令，则直接退出整个until循环结构体，此时整个until语句结构体的状态返回值取决于退出整个循环结构体时最后一个命令的执行状态返回值

在上述until循环语句结构中需要注意的是

- COMMANDS_LIST：表示待执行的命令列表(也称为until循环体)，即一系列shell命令的集合，类型格式多种多样，在一系列示例代码中可见一斑
  - 注意：在命令列表中不能使用 () 操作符改变优先级，它的作用是让括号内的语句成为命令列表进入子shell中执行，它的具体作用可参考：括号操作符
- TEST_COMMANDS_LIST：表示条件测试命令，即通过引用条件测试命令的执行状态返回值是否为0来判断是否执行上述COMMANDS_LIST循环体；**这里需要特别注意的是，和其它语言不通，shell的条件测试命令只有以下三种类型**
  - 命令执行：命令本身执行后就会产生对应的执行状态返回值，所以可以直接用来做条件判断

    * 此时不能使用‘‘操作符来引用命令，因为该操作引用的是命令的执行结果，而不是命令的执行状态返回值
    * 通常是直接使用命令，然后在命令后面添加s`&> /dev/null`，表示将命令的执行结果重定向至`/dev/null`，只引用其状态返回值；例如：`if grep "^root" /etc/passwd &> /dev/null; then`
  - 执行条件测试表达式：在shell中，条件测试表达式是由条件测试操作符以及对应的操作数组成，详细介绍可参考下列：条件测试表达式，执行条件测试表达式有以下三种格式
    * `test Test_Expression`：通过`test`命令执行
    * `[ Test_Expression ]`：通过`[]`操作符执行，注意`Test_Expression` 前后有空格
    * `[[ Test_Expression ]]`：通过`[[]]`操作符执行，注意`Test_Expression` 前后有空格
  - 组合条件测试：即对多个命令执行状态返回值或者执行条件测试表达式状态返回值做逻辑运算，组合条件测试有以下三种格式
    * 逻辑与操作：只有当`&&`操作符两边执行结果都为真(状态值为0)，最后组合条件测试结果才为真(状态值为0)
      - `[ Test_Expression1 ] && [ Test_Expression2 ]`：此处使用`[]`或`[[]]`都行
      - `COMMAND &> /dev/null && [ Test_Expression2 ]`：此处使用`[]`或`[[]]`都行
      - `COMMAND1 &> /dev/null && COMMAND2 &> /dev/null &&`
      - `[ Test_Expression1 -a Test_Expression2 ]`：此处使用`[]`或`[[]]`都行
      - `[[ Test_Expression1 && Test_Expression2 ]]`：此处只能使用`[[]]`操作符，因为`&&`运算符不允许用于`[]`操作符中
    * 逻辑或操作：只要`||`操作符两边执行结果有一个为真(状态值为0)，最后组合条件测试结果就为真(状态值为0)
      - `[ Test_Expression1 ] || [ Test_Expression2 ]`：此处使用`[]`或`[[]]`都行
      - `COMMAND &> /dev/null || [ Test_Expression2 ]`：此处使用`[]`或`[[]]`都行
      - `COMMAND1 &> /dev/null || COMMAND2 &> /dev/null &&`
      - `[ Test_Expression1 -0 Test_Expression2 ]`：此处使用`[]`或`[[]]`都行
      - `[[ Test_Expression1 || Test_Expression2 ]]`：此处只能使用`[[]]`操作符，因为`||`运算符不允许用于`[]`操作符中
    * 逻辑非操作：对`!`右侧执行结果取反
      - `! [ Test_Expression ]`：此处使用`[]`或`[[]]`都行
      - `! COMMAND1 &> /dev/null`
      - `! ([ Test_Expression1 ] || [ Test_Expression2 ])`：此处相当于`! [ Test_Expression1 ] && ! [ Test_Expression2 ]`
      - `! ([ Test_Expression1 ] && [ Test_Expression2 ])`：此处相当于`! [ Test_Expression1 ] || ! [ Test_Expression2 ]`
    * 注意：**非的优先级大于与，与的优先级大于或**

`until`循环语句的循环退出机制有：

- `continue`：跳出当前循环进入下一循环
- `break[n]`：默认跳出整个循环；n可以指定跳出几层循环

- 条件测试：此时为了避免死循环，TEST_COMMANDS_LIST条件测试里必须有控制循环次数的变量；COMMANDS_LIST循环体里必须有改变条件测试中用于控制循环次数变量的值操作

until循环语句也是适用于循环次数未知的场景，示例代码如下

```
#!/bin/bash

declare -i i=5

until echo hello;[ "$i" -eq 1 ]; do
        let --i
        echo $i
done

# 执行结果如下
# hello
# 4
# hello
# 3
#hello+
# 2
# hello
# 1
# hello
```

1.5.7 0x0203 循环退出命令

循环退出命令有

- continue [n]：表示退出当前循环进入下一次循环，适用于for、while、until、select语句；n表示退出的循环的次数，默认n=1
- break [n]：表示退出整个循环，适用于for、while、until、select语句；n表示退出的循环层数，默认n=1
- return [n]：表示退出整个函数，适用于函数体内的for、while、until、select语句，同样也适用于函数体内的if、case语句；数值n表示函数的退出状态码，如果没有定义退出状态码，则函数的状态退出码为函数的最后一条命令的执行状态返回值
- exit [n]：表示退出当前shell，适用于脚本的任何地方，表示退出整个脚本；数值n表示脚本的退出状态码，如果没有定义退出状态码，则脚本的状态退出码为脚本的最后一条命令的执行状态返回值

1.6 函数

在编程语言中，函数是能够实现模块化编程的工具，每个函数都是一个功能组件，但是函数必须被调用才能执行

函数存在的主要作用在于：最大化代码重用，最小化代码冗余

在shell中，函数可以被当做命令一样执行，它的本质是命令的组合结构体，即可以将函数看成一个普通命令或一个小型脚本。接下来本章内容将从以下几个方面来介绍函数

- 函数定义
- 函数调用
- 函数退出
- 示例代码

1.6.1 0x00 函数定义

在shell中函数定义的方法有两种(使用`help function`命令可以查看)

```
# 方法一
function FuncName {
        COMMANDS_LIST
} [&>/dev/null]

# 方法二
FuncName() {
        COMMANDS_LIST
} [&>/dev/null]
```

上面两种函数定义方法定义了一个名为`FuncName`的函数

- 方法一中：使用了`function`关键字，此时函数名`FuncName`后面的括号可以省略
- 方法二中：省略了`function`关键字，此时函数名`FuncName`后面的括号不能省略

`COMMANDS_LIST`是函数体，它与以下特点

- 函数体通常使用大括号`{}`包围，由于历史原因，在shell中大括号本身也是关键字，所以为了不产生歧义，函数体和大括号之间必须使用空格、制表符、换行符分隔开来；一般我们都是通过换行符进行分隔
- 函数体中的每一个命令必须使用`;`或换行符进行分隔；如果使用`&`结束某条命令，则表示该条命令会放入后台执行

需要注意的是

- `&>/dev/null`表示将函数体执行过程中可能输出的信息重定向至`/dev/null`中，该功能可选
- 定义函数时，还可以指定可选的函数重定向功能，这样当函数被调用的时候，指定的重定向也会被执行
- 当前shell定义的函数只能在当前shell使用，子shell无法继承父shell的函数定义，除非使用`export -f`将函数导出为全局函数；如果想取消函数的导出可以使用`export -n`
- 定义了函数后，可以使用`unset -f`移除当前shell中已定义的函数
- 可以使用`typeset -f [func_name]`或`declare -f [func_name]`查看当前shell已定义的函数名和对应的定义语句；使用`typeset -F`或`declare -F`则只显示当前shell中已定义的函数名
- 只有先定义了函数，才可以调用函数；不允许函数调用语句在函数定义语句之前
- 在shell脚本中，函数没有形参的概念，使用方法二定义函数时，括号里什么都不用写，只需要在函数体内使用相关的调用机制调用接收参数即可

1.6.2 0x01 函数调用

函数的调用格式如下

```
FuncName ARGS_LIST
```

其中

- `FuncName`：表示被调用函数的函数名，需要注意的是在shell中函数调用时函数名后面没有`()`操作符
- `ARGS_LIST`：表示被调用函数的传入参数，在shell中给函数传入参数和脚本接收参数的方法相似，直接在函数名后面加上需要传入的参数即可

函数调用时需要注意以下几点

- 如果函数名和命令名相同，则优先执行函数，除非使用command命令。例如：定义了一个名为rm的函数，在bash中输入rm执行时，执行的是rm函数，而非/bin/rm命令，除非使用command rm ARGS，表示执行的是/bin/rm命令
- 如果函数名和命令别名相同，则优先执行命令别名，即在优先级方面：别名别名>函数>命令自身

当函数调用函数被执行时，它的执行逻辑如下

- 接收参数：shell函数也接受位置参数变量，但函数的位置参数是调用函数时传递给函数的，而非传递给脚本的参数，所以脚本的位置变量和函数的位置变量是不同的；同时shell函数也接收特殊变量。函数体内引用位置参数和特殊变量方式如下
  - 位置参数
    - $0：和脚本位置参数一样，引用脚本名称
    - $1：引用函数的第1个传入参数
    - $n：引用函数的第n个传入参数
  - 特殊变量
    - $?：引用上一条命令的执行状态返回值，状态用数字表示0-255
      - 0：表示成功
      - 1-255：表示失败；其中1/2/127/255是系统预留的，写脚本时要避开与这些值重复
    - $$：引用当前shell的PID。除了执行bash命令和shell脚本时，$$不会继承父shell的值，其他类型的子shell都继承
    - $!：引用最近一次执行的后台进程PID，即运行于后台的最后一个作业的PID
    - $#：引用函数所有位置参数的个数
    - $*：引用函数所有位置参数的整体，即所有参数被当做一个字符串
    - $@：引用函数所有单个位置参数，即每个参数都是一个独立的字符串
- 执行函数体：在函数体执行时，需要注意的是
  - 函数内部引用变量的查找次序：内层函数自己的变量>外层函数的变量>主程序的变量>bash内置的环境变量
  - 函数内部引用变量的作用域
    - 本地变量：函数体引用本地变量时，重新赋值会覆盖原来的值，如果不想覆盖值，可以使用local进行修饰
    - 局部变量：函数体引用局部变量时，函数退出，将会被撤销
    - 环境变量：函数体引用环境变量时，重新赋值会覆盖原来的值，如果不想覆盖值，可以使用local进行修饰
    - 位置变量：函数体引用位置变量表示引用传递给函数的参数
    - 特殊变量
- 函数返回：函数返回值可分为两类
  - 执行结果返回值：正常的执行结果返回值有以下几种
    - 函数中的打印语句：如echo、print等
    - 最后一条命令语句的执行结果值
  - 执行状态返回值：执行状态返回值主要有以下几种
    - 使用return语句自定义返回值，即return n，n表示函数的退出状态码，不给定状态码时默认状态码为0
    - 取决于函数体中最后一条命令语句的执行状态返回值

在shell中不仅可以调用本脚本文件中定义的函数，还可以调用其它脚本文件中定义的函数

- 先使用`. /path/to/shellscript`或`source /path/to/shellscript`命令导入指定的脚本文件
- 然后使用相应的函数名调用函数即可

1.6.3 0x02 函数退出命令

函数退出命令有

- `return [n]`：可以在函数体内的任何地方使用，表示退出整个函数；数值n表示函数的退出状态码
- `exit [n]`：可以在脚本的任何地方使用，表示退出整个脚本；数值n表示脚本的退出状态码

此处需要注意的是：`return`并非只能用于`function`内部

- 如果`return`在`function`之外，但在`.`或者`source`命令的执行过程中，则直接停止该执行操作，并返回给定状态码n(如果未给定，则为0)
- 如果`return`在`function`之外，且不在`source`或`.`的执行过程中，则这将是一个错误用法

可能有些人不理解为什么不直接使用`exit`来替代这时候的`return`。下面给个例子就能清楚地区分它们

先创建一个脚本文件`proxy.sh`，内容如下，用于根据情况设置代理的环境变量

```
#!/bin/bash

proxy="http://127.0.0.1:8118"
function exp_proxy() {
        export http_proxy=$proxy
        export https_proxy=$proxy
        export ftp_proxy=$proxy
        export no_proxy=localhost
}

case $1 in
        set) exp_proxy;;
        unset) unset http_proxy https_proxy ftp_proxy no_proxy;;
        *) return 0
esac
```

首先我们来了解下`source`的特性：即`source`是在当前shell而非子shell执行指定脚本中的代码

当进入bash

- 需要设置环境变量时：使用`source proxy.sh set`即可
- 需要取消环境变量时：使用`source proxy.sh unset`即可

此时如果不清楚该脚本的用途或者一时手快直接输入`source proxy.sh`，就可以区分`exit`和`return`

- 如果上述脚本是`return 0`，那么表示直接退出脚本而已，不会退出bash
- 如果上述脚本是`exit 0`，则表示退出当前bash，因为`source`是在当前shell而非子shell执行指定脚本中的代码

可能你想象不出在source执行中的return有何用处：从source来考虑，它除了用在某些脚本中加载其他环境，更主要的是在bash环境初始化脚本中使用，例如`/etc/profile`、`~/.bashrc`等，如果你在`/etc/profile`中用`exit`来替代`function`外面的`return`，那么永远也登陆不上bash

1.6.4 0x03 示例代码

- 随机生成密码

```
#!/bin/bash

genpasswd(){
        local l=$1
        [ "$1" == ""  ]&& l=20
        tr -dc A-Za-z0-9_</dev/urandom | head -c ${l} | xargs
}

genpasswd $1    # 将脚本传入的位置参数传递给函数，表示生成的随机密码的位数
```

- 写一个脚本，完成如下功能：
  - 1、脚本使用格式：`mkscript.sh [-D|--description "script description"] [-A|--author "script author"] /path/to/somefile`
  - 2、如果文件事先不存在，则创建；且前几行内容如下所示：
    - #!/bin/bash
    - # Description: script description
    - # Author: script author
  - 3、如果事先存在，但不空，且第一行不是`#!/bin/bash`，则提示错误并退出；如果第一行是`#!/bin/bash`，则使用vim打开脚本；把光标直接定位至最后一行
  - 4、打开脚本后关闭时判断脚本是否有语法错误；如果有，提示输入y继续编辑，输入n放弃并退出；如果没有，则给此文件以执行权限

```
#!/bin/bash
read -p "Enter a file: " filename
declare authname
declare descr

options(){
if [[ $# -ge 0 ]];then
        case $1 in
    -D|--description)
        authname=$4
        descr=$2
        ;;
    -A|--author)
        descr=$4
        authname=$2
        ;;
        esac
fi
}

command(){
if  bash -n $filename &> /dev/null;then
        chmod +x $filename
else
    while true;do
        read -p "[y|n]:" option
        case $option in
        y)
                vim + $filename
                ;;
        n)
                exit 8
                ;;
        esac
        done
```

(continues on next page)

(续上页)

```
fi
exit 6
}

oneline(){
if [[ -f $filename ]];then
        if [ `head -1 $filename` == "#!/bin/bash" ];then
        vim + $filename
        else
        echo "wrong..."
        exit 4
        fi
else
        touch $filename && echo -e "#!/bin/bash\n# Description: $descr\n# Author:
↪$authname" > $filename
        vim + $filename
fi
command
}

options $*
oneline
```

- 写一个脚本，完成如下功能：
  - 1、提示用户输入一个可执行命令
  - 2、获取这个命令所依赖的所有库文件(使用ldd命令)
  - 3、复制命令至/mnt/sysroot/对应的目录中；如果复制的是cat命令，其可执行程序的路径是/bin/cat，那么就要将/bin/cat复制到/mnt/sysroot/bin/目录中，如果复制的是useradd命令，而useradd的可执行文件路径为/usr/sbin/useradd，那么就要将其复制到/mnt/sysroot/usr/sbin/目录中
  - 4、复制各库文件至/mnt/sysroot/对应的目录中

```
#!/bin/bash

options(){
        for i in $*;do
                dirname=`dirname $i`
                [ -d /mnt/sysroot$dirname ] || mkdir -p /mnt/sysroot$dirname
                [ -f /mnt/sysroot$i ]||cp $i /mnt/sysroot$dirname/
        done
}

while true;do
        read -p "Enter a command : " pidname
        [[ "$pidname" == "quit" ]] && echo "Quit " && exit 0
        bash=`which --skip-alias $pidname`
        if [[ -x $bash ]];then
                options `/usr/bin/ldd $bash |grep -o "/[^[:space:]]\{1,\}"`
                options $bash
        else
                echo "PLZ a command!"
        fi
done

# 说明
# 将bash命令的相关bin文件和lib文件复制到/mnt/sysroot/目录中后
# 使用chroot命令可切换根目录，切换到/mnt/sysroot/后可当做bash执行复制到该处的命令，作为bash中
的bash
```

- 写一个脚本，用来判定172.16.0.0网络内有哪些主机在线，在线的用绿色显示，不在线的用红色显示

```
#!/bin/bash
Cnetping(){
        for i in {1..254};do
        ping -c 1 -w 1 $1.$i
        if [[ $? -eq 0 ]];then
                echo -e -n "\033[32mping 172.16.$i.$j ke da !\033[0m\n"
        else
                echo -e -n "\033[31mping 172.16.$i.$j bu ke da !\033[0m \n"
        fi
        done
}

Bnetping(){
        for j in {0..255};do
                Cnetping $1.$j
        done
}

Bnetping 172.16
```

- 写一个脚本，用来判定随意输入的ip地址所在网段内有哪些主机在线，在线的用绿色显示，不在线的用红色显示

```
#!/bin/bash
Cnetping(){
        for i in {1..254};do
        ping -c 1 -w 1 $1.$i
        if [[ $? -eq 0 ]];then
                echo -e -n "\033[32mping 172.16.$i.$j ke da !\033[0m\n"
        else
                echo -e -n "\033[31mping 172.16.$i.$j bu ke da !\033[0m \n"
        fi
        done
}

Bnetping(){
        for j in {0..255};do
                Cnetping $1.$j
        done
}

Anetping(){
        for m in {0.255};do
                Bnetping $1.$m
        done
}

netType=`echo $1 | cut -d'.' -f1`

if [ $netType -gt 0 -a $netType -le 126 ];then
        Anetping $1
elif [ $netType -ge 128 -a $netType -le 191 ];then
        Bnetping $1
elif [ $netType -ge 192 -a $netType -le 223 ];then
        Cnetping $1
else
        echo "Wrong"
        exit 3
fi
```

1.7 知识碎片

shell脚本是一种纯面向过程的脚本编程语言

在编写shell脚本时，需要注意以下几点

- 标准输出：在编写shell脚本的时候，要考虑下该命令语句是否存在标准输出
  - 如果有，是否需要输出到标准输出设备上
  - 如果不需要，那就输出重定向至`/dev/null`
- 常见逻辑错误
  - 用户输入是否为空问题
  - 用户输入字符串大小写问题
  - 用户输入是否存在问题
- 编程思想
  - 明确脚本的输入、输出是什么
  - 根据输入考虑可能存在逻辑错误的地方
  - 根据输出判断使用什么控制流程
  - 在保证功能实现的前提下进行优化精简代码

在编写shell脚本时，常用到的一些命令语句

- 判断用户是否存在
  - `grep "^$userName\>" /etc/passwd &> /dev/null`
  - `id $userName`
- 获取用户的相关信息(用户名，UID，GID或者默认shell)
  - 对`/etc/passwd`文件进行处理
  - 使用`id`命令
- 脚本文件中导入调用其它脚本文件
  - `source config_file`
  - `. config_file`

```
#!/bin/bash
# configurefile: /tmp/script/myscript.conf

# 先判断对导入文件是否有读权限，然后尝试导入
[ -r /tmp/script/myscript.conf ] && . /tmp/script/myscript.conf
# 如果导入文件没有成功或者导入文件中对引用变量没有相关定义时，需定义默认值，防止出错
userName=${userName:-testuser}

echo $userName
```

- 读取文件内容

```
while read line; do
        CMD_LIST
done < /path/to/somefile
```

- 下载文件

```
#!/bin/bash

url="http::/mirrors.aliyun.com/centos/centos6.5.repo"

which wget &> /dev/null || exit 5 # 如果wget命令不存在就退出

downloader=`which wget`  # 获取wget命令的二进制文件路径

[ -x $downloader ] || exit 6 # 如果二进制文件没有执行权限就退出

$downloader $url
```

- 创建临时文件或目录

```
mktemp [-d] /tmp/file.XX    # X指定越多，随机生成的后缀就越长，其中-d表示创建临时目录
```

CHAPTER 2

常用类库

shell脚本中常用的类库可以分为三大类

2.1 常用环境变量

shell脚本中常用的环境变量有

- *IFS*
- *RANDOM*

2.1.1 0x00 IFS

shell下的很多命令都会分割单词，绝大多数时候默认是采用空格作为分隔符，有些时候遇到制表符、换行符也会进行分隔；这种分隔符是由IFS环境变量指定的

IFS是shell内部字段分隔符的环境变量

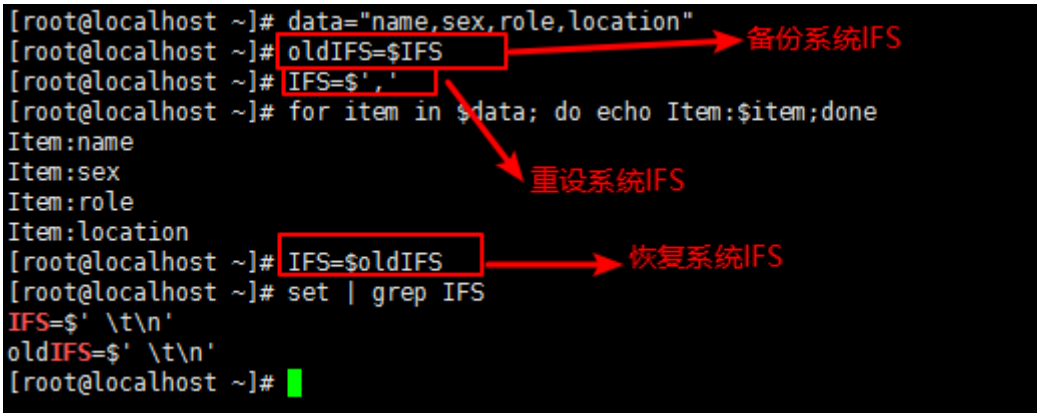
```
[root@localhost ~]# set | grep IFS
IFS=$' \t\n'
_=IFS
[root@localhost ~]#
```

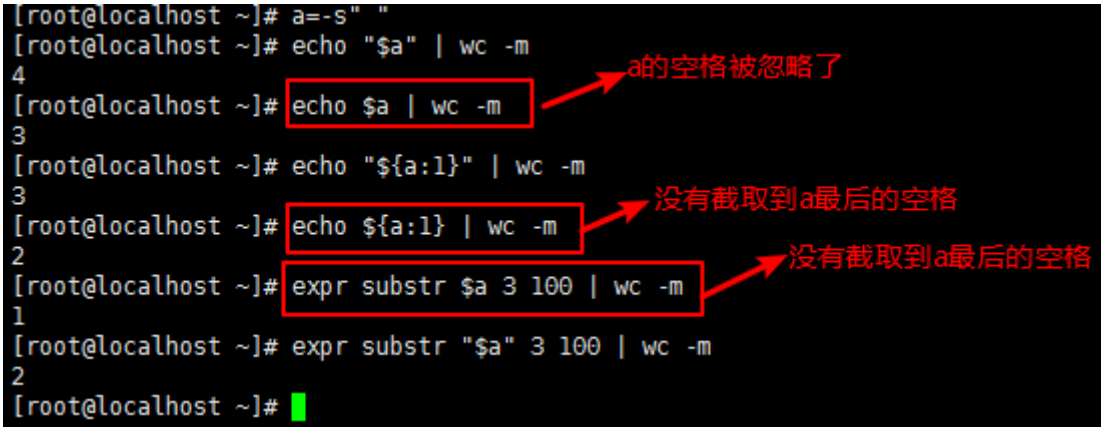
由上图可知：默认的IFS在碰到空格、制表符\t或 分行符\n就会自动分隔进入下一步；但是对空格处理有点不一样，对行首和行尾两边的空格不处理，并且多个连续的空格默认当作一个空格

有些时候在编写脚本或执行循环的时候，修改IFS可以起很大作用。如果要修改IFS，最好记得先备份系统IFS，再需要的地方再还原IFS

大多数时候，我们都不会去修改IFS来达到某种目的，而是采用其他方法来替代实现。这样就需要注意默认IFS的一个特殊性，它会忽略前导空白和后缀空白，并压缩连续空白；但是在某些时候，这会出现意想不到的问题

因此，**在可以对变量加引号的情况下，一定要加上引号来保护空白字符**





2.1.2 0x01 RANDOM

RANDOM环境变量是bash的伪随机数生成器

- $RANDOM=======生成0~32767之间的随机数
- $[RANDOM%num]===生成0~num之间的随机数；对算术表达式的值进行引用时需要使用[]

代码示例：通过脚本生成n个随机数(N>5),对这些随机数按从小到大排序

```
#!/bin/bash
declare -a arrynumber
read -p "Enter a number:" opt
opt=$[opt-1]
for i in `seq 0 $opt`;do
  arrynumer[$i]=$[RANDOM%1000]
done
let length=${#arrynumer[@]}
length=$[length-1]
for i in `seq 0 $length`;do
  let j=i+1
  for j in `seq $j $length`; do
      if [ ${arrynumer[$j]} -lt ${arrynumer[$i]} ];then
        temp=${arrynumer[$j]}
        arrynumer[$j]=${arrynumer[$i]}
        arrynumer[$i]=$temp
      fi
  done
  echo ${arrynumer[$i]}
done
```

2.2 常用命令

shell脚本可以理解为shell命令的集合，shell命令可以分为两大类

- `shell内部命令(builtin command)`：在bash中内部实现的命令叫做内建命令，在文件系统上没有对应的可执行文件
- `shell外部命令(binary command)`：在文件系统上的某个位置(/bin、/sbin等)有一个与命令名称对应的可执行文件

关于shell脚本中可能会用到的shell命令可以参考：linux工具集之shell命令

在此处我们主要介绍下shell脚本中使用频次最高的几个命令

- *read*：获取用户输入
- *echo*：打印输出
- *printf*：打印输出
- *shift*：剔除位置参数

2.2.1 0x00 read

参考文档：shell脚本之read命令

`read`命令是用来获取用户输入内容，即标准输入设备(键盘)输入内容，它是shell内建命令，使用`help read`命令可以查看其语法格式和使用说明，它的语法格式如下

```
read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-
↪t timeout] [-u fd] [var_name ...]
```

其执行逻辑如下

- `read`命令从标准输入设备中读取输入单行，默认单行的结束符号为回车换行符
  - 此处需要注意的是：不带任何选项的read命令，只有按下回车键才能结束read命令的读取
- 然后将读取的单行根据`IFS`环境变量分裂成多个字段，并将分割后的字段分别赋值给`read`命令后面指定的变量列表`var_name`，其赋值逻辑如下
  - 第一个字段分配给第一个变量`var_name1`，第二个字段分配给第二个变量`var_name2`，依次到结束
  - 如果指定的变量名少于字段数量，则多出的字段数量也同样分配给最后一个`var_name`
  - 如果指定的变量命令多于字段数量，则多出的变量赋值为空
  - 如果没有指定任何`var_name`，则分割后的所有字段都存储在特定变量`REPLY`中

read命令的常用选项有

- `-a`：将分隔后的字段依次存储到`-a`指定的数组中，存储的起始位置从数组的`index=0`开始
- `-d`：指定读取行的结束符号，默认结束符号为换行符
- `-n`：限制读取`N`个字符就自动结束读取，如果没有读满`N`个字符就按下回车或遇到换行符，则也会结束读取
- `-N`：严格要求读满`N`个字符才自动结束读取，即使中途按下了回车或遇到了换行符也不结束，其中换行符或回车算一个字符
- `-p`：输出提示符或提示语，默认不支持`\n`换行，要换行需要特殊处理
- `-r`：禁止反斜线的转义功能，这意味着`\`会变成文本的一部分
- `-s`：静默模式，输入的内容不会回显在屏幕上，常用来获取密码输入

- -t：给出超时时间，在达到超时时间时，read退出并返回错误，也就是说不会读取任何内容，即使已经输入了一部分

-a选项将读取的内容分配给数组变量，从索引号0开始分配

```
[root@localhost ~]# read -a array_test
my name is anony
[root@localhost ~]# echo ${array_test[@]}
my name is anony
[root@localhost ~]# echo ${array_test[0]}
my
[root@localhost ~]#
```

-d选项指定读取行的结束符号，而不再使用换行符

-n和-N选项限制输入字符

-p选项输出提示字符串

-p选项默认不带换行功能，且也不支持\n换行，但通过$'string'的方式特殊处理，就可以实现换行的功能；关于$'String'和$"String"的作用，详见shell中加引号有什么用

-s选项用来获取密码输入

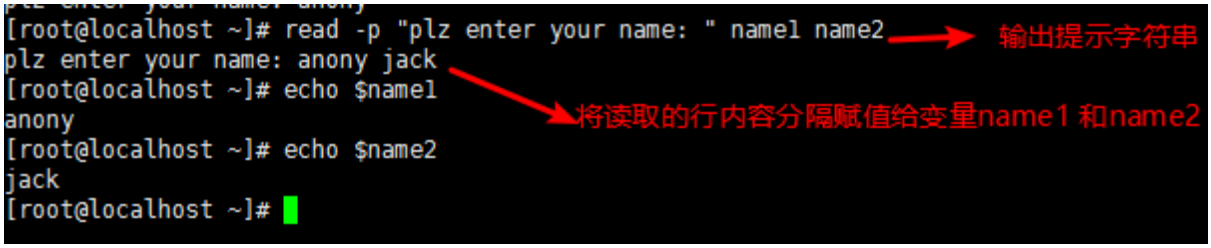
-t选项给出输入时间限制，没完成的输入将被丢弃，所有变量将赋值为空(如果在执行read前，变量已被赋值，则此变量在read超时后将被覆盖为空)

read也可以用来在shell脚本中读取文件内容

```
# 每读取文件一行内容，就会进入一次while循环，直到读完文件尾部退出循环

# 读取文件方法一
while read line; do
        echo $line
done < /etc/passwd

# 读取文件方法二
exec </etc/passwd;while read line; do
echo $line
done
```




```
[root@localhost ~]# read -p $'Enter your name: \n'
Enter your name:
anony
[root@localhost ~]#
```

2.2.2 0x01 echo

echo命令类似于c中printf，用于标准输出，它是shell内建命令，使用help read命令可以查看其语法格式和使用说明，它的语法格式如下

```
echo [-neE] [arg ...]
```

它的执行逻辑是：将给定arg内容按照-neE选项指定的不同方式输出

echo命令常用的选项有

- -n：取消分行输出
- -e：支持字符串内转义字符的显示输出

关于echo命令的使用，主要关注一些几点

- echo中的引号和感叹号：在bash环境中，感叹号只能通过单引号包围来输出，不能通过双引号来包围输出，原因有
  - 在bash环境中，感叹号表示引用历史命令，除非设置set +H关闭历史命令的引用
  - ''：单引号表示强引用，该操作符的优先级大于!，即不会进行历史命令的引用，直接引用显示全部字符
  - ""：双引号表示弱引用，该操作符的优先级小于!，即先进行历史命令的引用，然后再引用显示全部字符
- echo中的转义：通过-e选项识别转义和特殊意义的符号，如换行符\n、制表符\t、转义符\等

```
echo "hello world"    # 打印字符串

# -e选项支持字符串内转义字符的显示输出
echo -e "hello\bworld"          # 删除前面的字符，输出hellworld
echo -e "hello\tworld"          # 制表符，输出hello     world
echo -e "hello\vworld"          # 垂直制表符
echo -e "hello\nworld"          # 换行符
```

- echo中的分行处理：默认情况下echo会在每行行尾加上换行符号，使用-n选项可以取消分行输出
- echo中的颜色输出：echo可以控制字体颜色和背景颜色输出，因为需要使用特殊符号，所以需要配合-e选项来识别特殊符号

```
[root@localhost ~]# read -s -p "please enter your password: "
please enter your password: [root@localhost ~]#
[root@localhost ~]# echo $REPLY
515151
[root@localhost ~]#
```

```
[root@localhost ~]# var=5
[root@localhost ~]# read -t 3 var
1[root@localhost ~]# 1
-bash: 1: command not found
[root@localhost ~]# echo $var

[root@localhost ~]#
```

```
[root@localhost ~]# echo hello world!
hello world!
[root@localhost ~]# echo 'hello world!'
hello world!
[root@localhost ~]# echo "hello world!"
-bash: !": event not found
[root@localhost ~]# echo hello world!;echo 'hello world!'
-bash: !: event not found
[root@localhost ~]# echo 'hello world!';echo hello world!
hello world!
hello world!
[root@localhost ~]# set +H
[root@localhost ~]# echo "hello world!"
hello world!
[root@localhost ~]#
```

默认双引号不能包围！

！只能在最尾部部

取消！的历史命令引用功能

```
[root@localhost ~]# str=hello
[root@localhost ~]# echo "$str"'!' "world"
hello! world
[root@localhost ~]#
```

只用单引号无法扩展变量，使用双引号不好输出感叹号，于是解决方法就是：对这种特殊符号分开引用

```
[root@localhost ~]# echo 'hello world!' > 1.txt
[root@localhost ~]# echo 'hello world!' >> 1.txt
[root@localhost ~]# cat 1.txt
hello world!
hello world!
[root@localhost ~]# echo -n 'hello world!' > 1.txt
[root@localhost ~]# echo -n 'hello world!' >> 1.txt
[root@localhost ~]# cat 1.txt
hello world!hello world![root@localhost ~]#
[root@localhost ~]#
```

输入完添加了换行符

取消了换行符

  - 常见的字体颜色：重置=0，黑色=30，红色=31，绿色=32，黄色=33，蓝色=34，紫色=35，天蓝色=36，白色=37
  - 常见的背景颜色：重置=0，黑色=40，红色=41，绿色=42，黄色=43，蓝色=44，紫色=45，天蓝色=46，白色=47
  - 字体控制选项：1表示高亮，4表示下划线，5表示闪烁
  - 着色显示字符串格式为："\033[@;@mSTRING\033[0;0m"从左往右各字段的含义依次是
    * \033表示定义一个转义序列，也可以使用\e
    * [表示开始定义颜色
    * @;@表示颜色定义，第一个@表示字背景颜色，颜色范围40–47；;用来分隔字背景颜色和文字颜色；第二个@表示文字颜色，颜色范围30–37。如果没有相关定义则表示默认颜色
    * m表示颜色定义完毕
    * STRING表示要输出的字符串
    * \033表示定义一个转义序列，也可以使用\e
    * [表示再次开启颜色定义
    * 0;0m表示将前面定义的背景颜色和文字颜色重置为默认颜色；注意定义了颜色之后就需要使用此项来重置关闭颜色，否则会继续影响bash环境的颜色，前面定义了几个@，该处就应该使用几个0来重置对应的颜色

```
echo -e "\033[32mhello\033[0m"          # 着色显示，默认背景颜色，字颜色为32绿色
echo -e "\033[34m 蓝色字 \033[0m"
echo -e "\033[35m 紫色字 \033[0m"
echo -e "\033[36m 天蓝字 \033[0m"
echo -e "\033[37m 白色字 \033[0m"
echo -e "\033[40;37m 黑底白字 \033[0m"
echo -e "\033[41;37m 红底白字 \033[0m"
echo -e "\033[42;37m 绿底白字 \033[0m"
echo -e "\033[43;37m 黄底白字 \033[0m"
echo -e "\033[44;37m 蓝底白字 \033[0m"
echo -e "\033[45;37m 紫底白字 \033[0m"
echo -e "\033[46;37m 天蓝底白字 \033[0m"
echo -e "\033[47;30m 白底黑字 \033[0m"
echo -e "\033[41;37;0m 关闭所有属性 \033[0m"
echo -e "\033[41;37;1m 设置高亮度 \033[0m"
echo -e "\033[41;37;4m 下划线 \033[0m"
echo -e "\033[41;37;5m 闪烁 \033[0m"
echo -e "\033[41;37;7m 反显 \033[0m"
echo -e "\033[41;37;8m 消隐 \033[0m"
```

2.2.3 0x02 printf

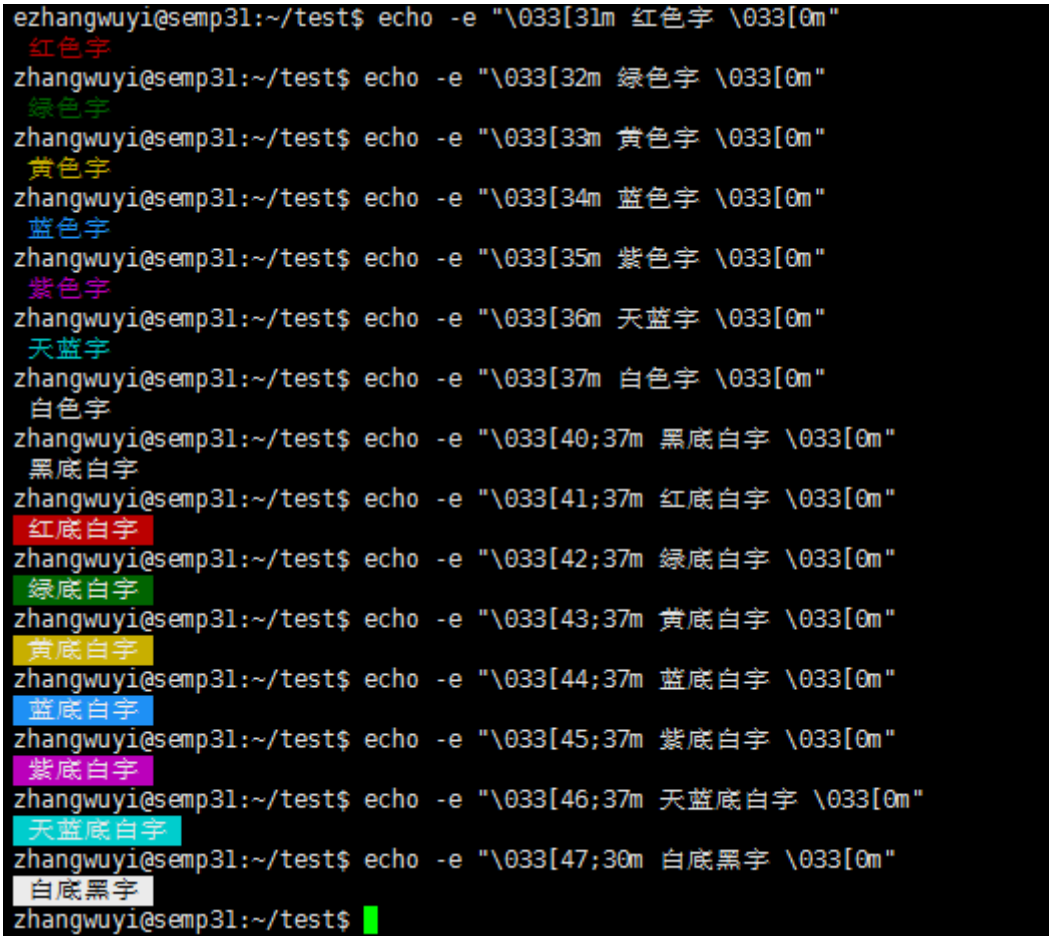
使用printf命令可以输出比echo更规则更格式化的结果，它引用于C语言的printf函数，但是有些许区别；它也是shell内建命令，使用help printf命令可以查看其语法格式和使用说明，它的语法格式如下

```
printf [-v var] format [arguments]
```

其执行逻辑是：按照format定义的输出格式将arguments输出到指定位置；默认是输出到标准输出，如果使用了-v选项表示将arguments按照指定格式赋值给该选项指定的变量var

使用printf最需要注意以下两点

- printf默认不在结尾加换行符，它不像echo一样，所以要手动加\n换号符
- printf只是格式化输出，不会改变任何结果，所以在格式化浮点数的输出时，浮点数结果是不变的，仅仅只是改变了显示的结果


ezhangwuyi@semp31:~/test$ echo -e "\033[31m 红色字 \033[0m"
红色字
zhangwuyi@semp31:~/test$ echo -e "\033[32m 绿色字 \033[0m"
绿色字
zhangwuyi@semp31:~/test$ echo -e "\033[33m 黄色字 \033[0m"
黄色字
zhangwuyi@semp31:~/test$ echo -e "\033[34m 蓝色字 \033[0m"
蓝色字
zhangwuyi@semp31:~/test$ echo -e "\033[35m 紫色字 \033[0m"
紫色字
zhangwuyi@semp31:~/test$ echo -e "\033[36m 天蓝字 \033[0m"
天蓝字
zhangwuyi@semp31:~/test$ echo -e "\033[37m 白色字 \033[0m"
白色字
zhangwuyi@semp31:~/test$ echo -e "\033[40;37m 黑底白字 \033[0m"
黑底白字
zhangwuyi@semp31:~/test$ echo -e "\033[41;37m 红底白字 \033[0m"
红底白字
zhangwuyi@semp31:~/test$ echo -e "\033[42;37m 绿底白字 \033[0m"
绿底白字
zhangwuyi@semp31:~/test$ echo -e "\033[43;37m 黄底白字 \033[0m"
黄底白字
zhangwuyi@semp31:~/test$ echo -e "\033[44;37m 蓝底白字 \033[0m"
蓝底白字
zhangwuyi@semp31:~/test$ echo -e "\033[45;37m 紫底白字 \033[0m"
紫底白字
zhangwuyi@semp31:~/test$ echo -e "\033[46;37m 天蓝底白字 \033[0m"
天蓝底白字
zhangwuyi@semp31:~/test$ echo -e "\033[47;30m 白底黑字 \033[0m"
白底黑字
zhangwuyi@semp31:~/test$


zhangwuyi@semp31:~/test$ echo -e "\033[41;37;0m 关闭所有属性 \033[0m"
关闭所有属性
"zhangwuyi@semp31:~/test$ echo -e "\033[41;37;1m 设置高亮度 \033[0m"
设置高亮度
zhangwuyi@semp31:~/test$ echo -e "\033[41;37;4m 下划线 \033[0m"
下划线
zhangwuyi@semp31:~/test$ echo -e "\033[41;37;5m 闪烁 \033[0m"
闪烁
zhangwuyi@semp31:~/test$ echo -e "\033[41;37;7m 反显 \033[0m"
反显
zhangwuyi@semp31:~/test$ echo -e "\033[41;37;8m 消隐 \033[0m"
zhangwuyi@semp31:~/test$

使用printf可以实现

- 指定字符串的宽度
- 实现左对齐(使用-)
- 实现右对齐(默认值)
- 格式化小数输出

```
#!/bin/bash

# 三个%分别对应后面的三个参数
# 减号"-"表示左对齐，默认表示右对齐
# 减号"-"后面的数字n表示占用n个字符
# 点号"."后面的数字m表示取小数点后m位
# s表示对应一个字符串变量
# f表示对应一个浮点数变量
# d表示对应一个整数变量
# \t表示制表符
# \n表示换行符
printf "%-s\t %-s\t %s\n" No Name Mark
printf "%-s\t %-s\t %4.2f\n" 1 Sarath 80.34
printf "%-s\t %-s\t %4.2f\n" 2 James 90.998
printf "%-s\t %-s\t %4.2f\n" 3 Jeff 77.564

# 执行结果如下
# No            Name            Mark
# 1             Sarath          80.34
# 2             James           91.00
# 3             Jeff            77.56
```

2.2.4 0x03 shift

shift命令在shell脚本中用来剔除脚本的位置变量，它是shell内建命令，使用help shift命令可以查看其语法格式和使用说明，它的语法格式如下

```
shift [n]
```

该命令常用来解析脚本的传入参数

- shift表示剔除脚本的第一个传入参数，后面参数往前排
- shift n表示剔除脚本的前n个传入参数，后面参数往前排

写一个脚本，使用形式：userinfo.sh -u username [-v {1|2}]

- -u选项用于指定用户，而后脚本显示用户的UID和GID
- -v选项后面是1，则显示用户的家目录路径；如果是2，则显示用户的家目录路径和shell

```
#!/bin/bash

[ $# -lt 2 ] && echo "less arguments" && exit 3

if [[ "$1" == "-u" ]]; then
        userName="$2"
        shift 2         # 剔除前2个位置参数
fi

if [[ $# -ge 2 ]] && [ "$1" == "-v" ]; then
        verFlag=$2
fi
```

(continues on next page)

(续上页)

```
verFlag=${verFlag:-0}

if [ -n $verFlag ]; then
        if ! [[ $verFlag =~ [012] ]]; then
                echo "Wrong Parameter"
                echo "Usage: `basename $0` -u UserName -v {1|2}"
                exit 4
        fi
fi

if [ $verFlag -eq 1 ];then
        grep "^$userName" /etc/passwd | cut -d: -f1,3,4,6
elif [ $verFlag -eq 2 ];then
        grep "^$userName" /etc/passwd | cut -d: -f1,3,4,6,7
else
        grep "^$userName" /etc/passwd | cut -d: -f1,3,4
fi
```

2.3 常用函数

参考文档：functions文件详细分析和说明

shell中函数和命令不一样，它没有对应的二进制文件，只有相关的声明定义

shell中函数可以大致分为两大类：自定义函数和库函数

自定义函数好说，直接在脚本中自行声明定义和调用即可；在这里我们主要是介绍库函数，shell中所谓库函数就是`/etc/rc.d/init.d/functions`文件中定义的系统函数，这些系统函数几乎被`/etc/rc.d/init.d/`下所有的sysv服务启动脚本加载，在该文件中提供了以下几个非常有用的函数

- 显示函数
  - *success*：显示绿色的`OK`，表示成功
  - *failure*：显示红色的`FAILED`，表示失败
  - *passed*：显示黄色的`PASSED`，表示pass该任务
  - *warning*：显示黄色的`warning`，表示警告
  - *confirm*：提示`(Y)es/(N)o/(C)ontinue? [Y]`并判断、传递输入的值
  - *is_true*：`$1`的布尔值代表为真时，返回状态码`0`，否则返回`1`；包括`t/y/yes/true`，不区分大小写
  - *is_false*：`$1`的布尔值代表为假时，返回状态码`0`，否则返回`1`；包括`f/n/no/false`，不区分大小写
  - *action*：根据进程退出状态码自行判断是执行`success`还是`failure`
- 进程函数
  - *checkpid*：检查`/proc`下是否有给定`pid`对应的目录，给定多个`pid`时，只要存在一个目录都返回状态码`0`
  - *__pids_var_run*：检查`pid`是否存在，并保存到变量`pid`中，同时返回几种进程状态码
  - *__pids_pidof*：获取进程`pid`
  - *pidfileofproc*：获取进程`pid`，但只能获取`/var/run`下的`pid`文件中的值
  - *pidofproc*：获取进程`pid`，可获取任意给定`pidfile`或默认`/var/run`下`pidfile`中的值
  - *status*：检查给定进程的运行状态
  - *daemon*：启动一个服务程序，启动前还检查进程是否已在运行

- *killproc*：杀掉给定的服务进程

以下是/etc/init.d/functions文件的开头定义的语句(本文分析的/etc/init.d/functions文件是CentOS 7上的，和CentOS 6有些许区别)

- 设置umask值，使得加载该文件的脚本所在shell的umask为22
- 导出PATH路径变量，但这个导出的路径变量并不理想，因为要为非rpm包安装的程序设计服务启动脚本时，必须写全路径命令，例如/usr/local/mysql/bin/mysql，因此，可以考虑将/etc/init.d/functions中的该语句注释掉

```
# Make sure umask is sane
umask 022

# Set up a default search path.
PATH="/sbin:/usr/sbin:/bin:/usr/bin"
export PATH
```

2.3.1 0x00 显示函数

显示函数常用在编写系统服务启动脚本，便于提示相关启动信息

2.3.2 0x0000 success

除了success函数，还有echo_success函数也可以显示绿色的OK，表示成功

以下是echo_success和success函数的定义语句

```
echo_success() {
    [ "$BOOTUP" = "color" ] && $MOVE_TO_COL
    echo -n "["
    [ "$BOOTUP" = "color" ] && $SETCOLOR_SUCCESS
    echo -n $"  OK  "
    [ "$BOOTUP" = "color" ] && $SETCOLOR_NORMAL
    echo -n "]"
    echo -ne "\r"
    return 0
}


success() {
        [ "$BOOTUP" != "verbose" -a -z "${LSB:-}" ] && echo_success
        return 0
}
```

这两个函数的功能就是：不换行带绿色输出[OK]字样；效果如下

```
[root@localhost init.d]# . /etc/init.d/functions
[root@localhost init.d]# success
[root@localhost init.d]#                                   [  OK  ]
[root@localhost init.d]# echo_success
[root@localhost init.d]#                                   [  OK  ]
```

2.3.3 0x0001 failure

除了failure函数，还有echo_failure函数也可以显示红色的FAILED，表示失败

以下是echo_failure和failure函数的定义语句

```
echo_failure() {
    [ "$BOOTUP" = "color" ] && $MOVE_TO_COL
    echo -n "["
    [ "$BOOTUP" = "color" ] && $SETCOLOR_FAILURE
    echo -n $"FAILED"
    [ "$BOOTUP" = "color" ] && $SETCOLOR_NORMAL
    echo -n "]"
    echo -ne "\r"
    return 1
}


failure() {
    local rc=$?
    [ "$BOOTUP" != "verbose" -a -z "${LSB:-}" ] && echo_failure
    [ -x /bin/plymouth ] && /bin/plymouth --details
    return $rc
}
```

这两个函数的功能就是：不换行带红色输出[FAILED]字样；效果如下

```
[root@localhost init.d]# . /etc/init.d/functions
[root@localhost init.d]# failure
[root@localhost init.d]#                                   [FAILED]
[root@localhost init.d]# echo_failure
[root@localhost init.d]#                                   [FAILED]
```

2.3.4 0x0002 passed

除了passed函数，还有echo_passed函数也可以显示黄色的PASSED，表示pass该任务

以下是echo_passed和passed函数的定义语句

```
echo_passed() {
    [ "$BOOTUP" = "color" ] && $MOVE_TO_COL
    echo -n "["
    [ "$BOOTUP" = "color" ] && $SETCOLOR_WARNING
    echo -n $"PASSED"
    [ "$BOOTUP" = "color" ] && $SETCOLOR_NORMAL
    echo -n "]"
    echo -ne "\r"
    return 1
}


passed() {
    local rc=$?
    [ "$BOOTUP" != "verbose" -a -z "${LSB:-}" ] && echo_passed
    return $rc
}
```

这两个函数的功能就是：不换行带黄色输出[PASSED]字样；效果如下

```
[root@localhost init.d]# . /etc/init.d/functions
[root@localhost init.d]# passed
[root@localhost init.d]#                                   [PASSED]
[root@localhost init.d]# echo_passed
[root@localhost init.d]#                                   [PASSED]
```

2.3.5 0x0003 warning

除了warning函数，还有echo_warning函数也可以显示黄色的warning，表示警告

以下是echo_warning和warning函数的定义语句

```
echo_warning() {
    [ "$BOOTUP" = "color" ] && $MOVE_TO_COL
    echo -n "["
    [ "$BOOTUP" = "color" ] && $SETCOLOR_WARNING
    echo -n $"WARNING"
    [ "$BOOTUP" = "color" ] && $SETCOLOR_NORMAL
    echo -n "]"
    echo -ne "\r"
    return 1
}


warning() {
    local rc=$?
    [ "$BOOTUP" != "verbose" -a -z "${LSB:-}" ] && echo_warning
    return $rc
}
```

这两个函数的功能就是：不换行带黄色输出[WARNING]字样；效果如下

```
[root@localhost init.d]# . /etc/init.d/functions
[root@localhost init.d]# warning
[root@localhost init.d]#                                   [WARNING]
[root@localhost init.d]# echo_warning
[root@localhost init.d]#                                   [WARNING]
```

2.3.6 0x0004 confirm

这个函数一般用不上，因为脚本本来就是为了避免交互式的。在CentOS 7的functions中已经删除了该函数定义语句。不过，借鉴下它的处理方法还是不错的

以下摘自CentOS 6.6的/etc/init.d/functions文件

```
# returns OK if $1 contains $2
strstr() {
        [ "${1#*$2*}" = "$1" ] && return 1    # 参数$1中不包含$2时，返回1，否则返回0
        return 0
}

# Confirm whether we really want to run this service
confirm() {
        [ -x /bin/plymouth ] && /bin/plymouth --hide-splash
        while : ; do
                echo -n $"Start service $1 (Y)es/(N)o/(C)ontinue? [Y] "
                read answer
                if strstr $"yY" "$answer" || [ "$answer" = "" ] ; then
                return 0
        elif strstr $"cC" "$answer" ; then
                rm -f /var/run/confirm
                [ -x /bin/plymouth ] && /bin/plymouth --show-splash
                return 2
        elif strstr $"nN" "$answer" ; then
                return 1
        fi
```

(continues on next page)

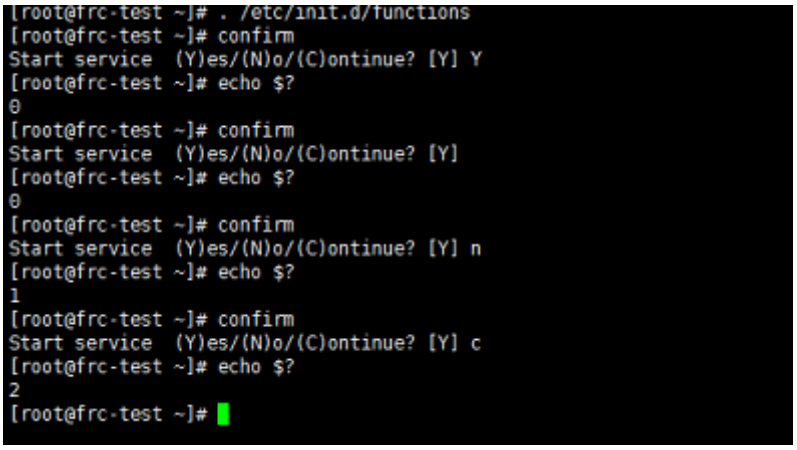
(续上页)

```
        done
}
```

上述代码中

- 第一个函数strstr的作用是判断第一个参数$1中是否包含了$2，如果包含了则返回状态码0,，这函数也是一个不错的技巧
- 第二个函数confirm的作用是根据交互式输入的值返回不同的状态码，如果输入的是y或Y或不输入时，返回0。输入的是c或C时，返回状态码2，输入的是n或“N“时返回状态码1

于是可以根据confirm的状态值决定是否要继续执行某个程序，用法和效果如下

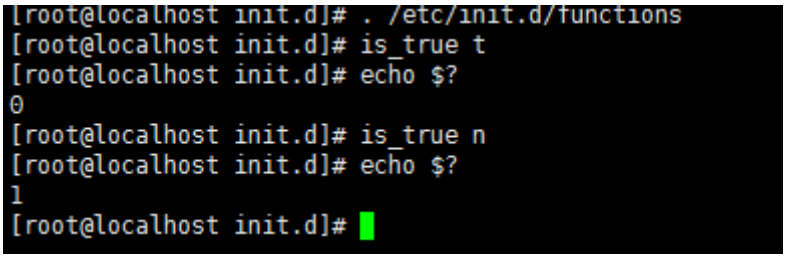


2.3.7 0x0005 is_true

以下是is_true函数的定义语句

```
# Evaluate shvar-style booleans
is_true() {
    case "$1" in
    [tT] | [yY] | [yY][eE][sS] | [oO][nN] | [tT][rR][uU][eE] | 1)
        return 0
        ;;
    esac
    return 1
}
```

由以上代码可知：这个函数的作用就是转换输入的布尔值为状态码；$1第一个函数参数的布尔值代表为真(包括t/y/yes/true，不区分大小写)时，返回状态码0，否则返回1



2.3.8 0x0006 is_false

以下是is_false函数的定义语句

```
# Evaluate shvar-style booleans
is_false() {
    case "$1" in
    [fF] | [nN] | [nN][oO] | [oO][fF][fF] | [fF][aA][lL][sS][eE] | 0)
        return 0
        ;;
    esac
    return 1
}
```

由以上代码可知：这个函数的作用就是转换输入的布尔值为状态码；$1第一个函数参数的布尔值代表为假(包括f/n/no/false，不区分大小写)时，返回状态码0，否则返回1

```
[root@localhost init.d]# . /etc/init.d/functions
[root@localhost init.d]# is_false false
[root@localhost init.d]# echo $?
0
[root@localhost init.d]# is_false true
[root@localhost init.d]# echo $?
1
[root@localhost init.d]#
```

2.3.9 0x0007 action

该函数在写脚本时还比较有用，可以根据退出状态码自动判断是执行success还是执行failure函数

以下是action函数的定义语句

```
# Run some action. Log its output.
action() {
    local STRING rc

    STRING=$1
    echo -n "$STRING "
    shift
    "$@" && success $"$STRING" || failure $"$STRING"
    rc=$?
    echo
    return $rc
}
```

这个函数定义的很有技巧

- 先将第一个参数保存并踢掉，再执行后面的命令("$@"表示执行后面的命令)
- 当action函数只有一个参数时，action直接返回OK，状态码为0；当超过一个参数时，第一个参数先被打印，再执行从第二个参数开始的命令

在脚本中使用action函数时，可以让命令执行成功与否的判断显得更专业，效果如下

```
[root@localhost init.d]# . /etc/init.d/functions
[root@localhost init.d]# action
                                                           [  OK  ]
[root@localhost init.d]# action 5
5                                                          [  OK  ]
[root@localhost init.d]# action sleeping sleep 3
sleeping                                                   [  OK  ]
[root@localhost init.d]#
```

通常，该函数会结合/bin/true和/bin/false命令使用，它们无条件返回0或1状态码；例如，mysqld启动脚本中，判断mysqld已在运行时，直接输出启动ok的消息，但实际上根本没做任

何事

```
# action函数使用格式
# action $"MESSAGES: " /bin/true
# action $"MESSAGES: " /bin/false

if [ $MYSQLDRUNNING = 1 ] && [ $? = 0 ]; then
        # already running, do nothing
        action $"Starting $prog: " /bin/true
        ret=0
fi
```

2.3.10 0x01 进程函数

启动进程时，pid文件非常重要

- pid文件不仅可以用来判断进程是否在运行，还可以从中读取pid号用来杀进程
- pid文件中可能有多行，表示多实例

2.3.11 0x0100 checkpid

checkpid函数是用来检测给定的`pid值在/proc下是否有对应的目录存在

以下是函数checkpid的定义语句

```
# Check if any of $pid (could be plural) are running
checkpid() {
    local i

    for i in $* ; do       # 检测/proc目录下是否存在给定的进程目录
        [ -d "/proc/$i" ] && return 0
    done
    return 1
}
```

每个进程都必有一个pid，但并不一定都记录在pid文件中，例如线程的pid；但无论如何，在/proc/目录下，一定会有pid号命名的目录，只要有对应pid号的目录，就表示该进程已经在运行

在检查/proc下是否有给定pid对应的目录，无论给定多少个pid，只要有一个有目录，都返回0

该函数的调用方法如下

```
checkpid pid_list
```

效果图如下

```
[root@localhost ~]# source /etc/init.d/functions
[root@localhost ~]# sleep 10 & a="$!";sleep 10 & a="$a $!";sleep 10 & a="$a $!";checkpid $a
[1] 7882
[2] 7883
[3] 7884
[root@localhost ~]# echo $?
0
[root@localhost ~]#
```

2.3.12 0x0101 __pids_var_run

__pids_var_run函数是用来判断给定程序的运行状态以及对应的pid文件是否存在

以下是函数__pids_var_run的定义语句

```
# __proc_pids {program} [pidfile]
# Set $pid to pids from /var/run* for {program}.  $pid should be declared
# local in the caller.
# Returns LSB exit code for the 'status' action

# 通过检测pid判断程序是否已在运行
__pids_var_run() {
    local base=${1##*/}                         # 获取进程名的basename
    local pid_file=${2:-/var/run/$base.pid}     # 定义pid文件路径
    local pid_dir=$(/usr/bin/dirname $pid_file > /dev/null)
    local binary=$3

    [ -d "$pid_dir" -a ! -r "$pid_dir" ] && return 4

    pid=
    if [ -f "$pid_file" ] ; then    # 判断给定的pid文件是否存在
            local line p

        [ ! -r "$pid_file" ] && return 4 # "user had insufficient privilege"
        while : ; do                        # 将pid文件中的pid值赋值给pid变量
            read line
            [ -z "$line" ] && break
            for p in $line ; do
                if [ -z "${p//[0-9]/}" ] && [ -d "/proc/$p" ] ; then
                    if [ -n "$binary" ] ; then
                        local b=$(readlink /proc/$p/exe | sed -e 's/\s*(deleted)$//
↪')
                        [ "$b" != "$binary" ] && continue
                    fi
                    pid="$pid $p"
                fi
            done
        done < "$pid_file"

            if [ -n "$pid" ]; then # pid存在，则返回0，否则表示pid文件存在，但/proc下没有
对应命令
                    return 0        # 即进程已死，但pid文件却存在，返回状态码1
            fi
        return 1 # "Program is dead and /var/run pid file exists"
    fi
    return 3 # "Program is not running"pid文件不存在时，表示进程未进行，返回状态码3
}
```

由函数定义可知：只有当pid文件存在，且/proc下有pid对应的目录时，才表示进程在运行(当然线程没有pid文件)，该函数的调用方法是：__pids_var_run program [pidfile]

- program为程序进程名
- pidfile为进程pid文件名，如果不给定pidfile，则默认为/var/run/$base.pid文件
  - pidfile的路径可能为/var/run/$base.pid文件($base表示进程名的basename)，此路径为默认值
  - pidfile的路径也可能是自定义的路径，例如mysql的pid可以自定义为/mysql/data/mysql01.pid
- 函数的执行结果有4种状态返回码
  - 0：表示program正在运行
  - 1：表示program进程已死，pid文件存在，但/proc目录下没有对应的文件
  - 3：表示pid文件不存在
  - 4：表示pid文件的权限错误，不可读

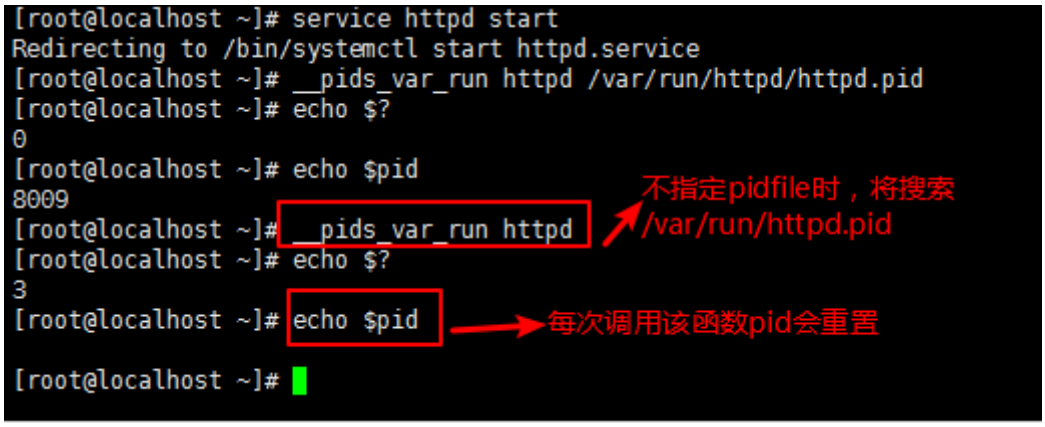
- 函数还会保存变量pid的结果，以供其他程序引用

这个函数非常重要，不仅可以从pidfile中获取并保存pid号码，还根据情况返回几种状态码，这几个状态码是status函数的重要依据，在SysV服务启动脚本中使用非常广泛

该函数的调用方法如下

```
__pids_var_run program [pidfile]
```

以下是httpd进程的测试结果，分别是指定pid文件和不指定pid文件的情况



2.3.13 0x0102 __pids_pidof

__pids_pidof函数是用来获取给定进程的pid

以下是函数__pids_pidof的定义语句

```
# Output PIDs of matching processes, found using pidof
# 忽略当前shell的PID，父shell的PID，调用pidof程序shell的PID
__pids_pidof() {
    pidof -c -m -o $$ -o $PPID -o %PPID -x "$1" || \
        pidof -c -m -o $$ -o $PPID -o %PPID -x "${1##*/}"
}
```

由以上代码可知：该函数使用了pidof命令，获取给定进程的pid值会更加精确，其中使用了几个-o选项，它用于忽略指定的pid

- -o $$表示忽略当前shell进程PID，大多数时候它会继承父shell的pid，但在脚本中时它代表的是脚本所在shell的pid
- -o $PPID表示忽略父shell进程PID
- -o %PPID表示忽略调用pidof命令的shell进程PID

关于pidof命令我们在这里简单介绍下，示例脚本如下

```
#!/bin/bash

echo 'pidof bash: '`pidof bash`
echo 'script shell pid: '`echo $$`
echo 'script parent shell pid: '`echo $PPID`
echo 'pidof -o $$ bash: '`pidof -o $$ bash`
echo 'pidof -o $PPID bash: '`pidof -o $PPID bash`
echo 'pidof -o %PPID bash: '`pidof -o %PPID bash`
echo 'pidof -o $$ -o $PPID -o %PPID bash: '`pidof -o $$ -o $PPID -o %PPID bash`
```

效果如下

上述效果图中

```
[root@localhost ~]# pidof bash
7306 6942 1552
[root@localhost ~]# (echo 'parent shell: '$$;echo "current bash pid: `pidof bash`";./test.sh)|cat -n
     1  parent shell: 6942
     2  current bash pid: 7337 7306 6942 1552
     3  pidof bash: 7340 7337 7306 6942 1552
     4  script shell pid: 7340
     5  script parent shell pid: 7337
     6  pidof -o $$ bash: 7337 7306 6942 1552
     7  pidof -o $PPID bash: 7340 7306 6942 1552
     8  pidof -o %PPID bash: 7337 7306 6942 1552
     9  pidof -o $$ -o $PPID -o %PPID bash: 7306 6942 1552
[root@localhost ~]#
```

- 第一个pidof命令显示结果中说明当前已有3个bash，pid分别为3306、2436、2302
- 第二个命令显示结果中
  - 行1说明括号的父shell为6942
  - 行5说明脚本的父shell为7337。即括号的父shell为当前bash环境，脚本的父shell为括号所在shell
  - 行2减第一个命令的结果说明括号所在子shell的pid为7337
  - 行3减行2说明shell脚本所在子shell的pid为7340
  - -o $$忽略的是当前shell，即脚本所在shell的pid，因为在shell脚本中时，$$不继承父shell的pid
  - -o $PPID忽略的是pidof所在父shell，即括号所在shell
  - -o %PPID忽略的是调用pidof程序所在的shell，即脚本所在shell

2.3.14 0x0103 pidfileofproc

pidfileofproc函数用来获取给定程序的pid，注意该函数不是获取pidfile，而是获取pid值

以下是函数pidfileofproc的定义语句

```
# A function to find the pid of a program. Looks *only* at the pidfile
pidfileofproc() {
    local pid

    # Test syntax.
    if [ "$#" = 0 ] ; then
        echo $"Usage: pidfileofproc {program}"
        return 1
    fi

    __pids_var_run "$1"          # 不提供pidfile，因此认为是/var/run/$base.pid
    [ -n "$pid" ] && echo $pid
    return 0
}
```

由以上代码可知：pidfileofproc函数只能获取/var/run下的pid值

该函数用的比较少，但确实有使用它的脚本；如crond启动脚本中借助pidfileofproc来杀进程

```
echo -n $"Stopping $prog: "
if [ -n "`pidfileofproc $exec`" ]; then
        killproc $exec
        RETVAL=3
else
        failure $"Stopping $prog"
fi
```

2.3.15 0x0104 pidofproc

pidofproc函数也可以用来获取给定程序的pid，注意该函数不是获取pidfile，而是获取pid值

以下是函数pidofproc的定义语句

```
# A function to find the pid of a program.
pidofproc() {
    local RC pid pid_file=

    # Test syntax.
    if [ "$#" = 0 ]; then
        echo $"Usage: pidofproc [-p pidfile] {program}"
        return 1
    fi
    if [ "$1" = "-p" ]; then     # 既可以获取/var/run/$base.pid中的pid
        pid_file=$2              # 也可以获取自给定pid文件中的pid
        shift 2
    fi
    fail_code=3 # "Program is not running"

    # First try "/var/run/*.pid" files
    __pids_var_run "$1" "$pid_file"
    RC=$?
    if [ -n "$pid" ]; then       # $pid不为空时，输出program的pid值
        echo $pid
        return 0
    fi

    [ -n "$pid_file" ] && return $RC   #  $pid为空，但使用了"-p"指定pidfile时，返回$RC
    __pids_pidof "$1" || return $RC    #  $pid为空，且$pidfile为空时，获取进程号pid并输出
}
```

由以上代码可知：pidofproc函数既可以获取/var/run下的pid值，又可以获取自给定pidfile中的pid值

该函数用的比较少，但确实有使用它的脚本；如dnsbind的named服务启动脚本中借助pidofproc来判断进程是否已在运行

```
pidofnamed() {
        pidofproc -p "$ROOTDIR$PIDFILE" "$named";
}

if [ -n "`pidofnamed`" ]; then
        echo -n $"named: already running"
        success
        echo
        exit 0
fi
```

2.3.16 0x0105 daemon

daemon函数用于启动一个程序，并根据结果输出success或failure

daemon函数的定义语句如下

```
# A function to start a program.
daemon() {
    # Test syntax.
    local gotbase= force= nicelevel corelimit
    local pid base= user= nice= bg= pid_file=
```

(continues on next page)

(续上页)

```
    local cgroup=
    nicelevel=0
    while [ "$1" != "${1##[-+]}" ]; do
        case $1 in
        '')
            echo $"$0: Usage: daemon [+/-nicelevel] {program}" "[arg1]..."
            return 1
            ;;
        --check)
            base=$2
            gotbase="yes"
            shift 2
            ;;
        --check=?*)
            base=${1#--check=}
            gotbase="yes"
            shift
            ;;
        --user)
            user=$2
            shift 2
            ;;
        --user=?*)
            user=${1#--user=}
            shift
            ;;
        --pidfile)
            pid_file=$2
            shift 2
            ;;
        --pidfile=?*)
            pid_file=${1#--pidfile=}
            shift
            ;;
        --force)
            force="force"
            shift
            ;;
        [-+][0-9]*)
            nice="nice -n $1"
            shift
            ;;
        *)
            echo $"$0: Usage: daemon [+/-nicelevel] {program}" "[arg1]..."
            return 1
            ;;
      esac
    done

    # Save basename.
    [ -z "$gotbase" ] && base=${1##*/}

    # See if it's already running. Look *only* at the pid file.
    __pids_var_run "$base" "$pid_file"

    [ -n "$pid" -a -z "$force" ] && return

    # make sure it doesn't core dump anywhere unless requested
    corelimit="ulimit -S -c ${DAEMON_COREFILE_LIMIT:-0}"

    # if they set NICELEVEL in /etc/sysconfig/foo, honor it
```

(continues on next page)

(续上页)

```
    [ -n "${NICELEVEL:-}" ] && nice="nice -n $NICELEVEL"

    # if they set CGROUP_DAEMON in /etc/sysconfig/foo, honor it
    if [ -n "${CGROUP_DAEMON}" ]; then
        if [ ! -x /bin/cgexec ]; then
            echo -n "Cgroups not installed"; warning
            echo
        else
            cgroup="/bin/cgexec";
            for i in $CGROUP_DAEMON; do
                cgroup="$cgroup -g $i";
            done
        fi
    fi

    # Echo daemon
    [ "${BOOTUP:-}" = "verbose" -a -z "${LSB:-}" ] && echo -n " $base"

    # And start it up.
    if [ -z "$user" ]; then
       $cgroup $nice /bin/bash -c "$corelimit >/dev/null 2>&1 ; $*"
    else
       $cgroup $nice runuser -s /bin/bash $user -c "$corelimit >/dev/null 2>&1 ; $*
↪"
    fi

    [ "$?" -eq 0 ] && success $"$base startup" || failure $"$base startup"
}
```

daemon函数调用方法

```
daemon [--check=servicename] [--user=USER] [--pidfile=PIDFILE] [--force] program␣
↪[prog_args]
```

其中需要注意的是

- 只有--user选项可以用来控制program启动的环境
- --check和--pidfile选项都是用来检查是否已运行的，不是用来启动的，如果提供了--check，则检查的是名为`servicename的进程，否则检查的是program名称的进程
- --force则表示进程已存在时仍启动
- prog_args是向program传递它的运行参数，一般会从/etc/sysconfig/$base文件中获取

例如httpd的启动脚本中

```
echo -n $"Starting $prog: "
daemon --pidfile=${pidfile} $httpd $OPTIONS
```

其执行结果大致如下

```
[root@xuexi ~]# service httpd start
Starting httpd:                                [  OK  ]
```

还需注意，通常program的运行参数可能也是--开头的，要和program前面的选项区分。例如

```
daemon --pidfile $pidfile --check $servicename $processname --pid-file=$pidfile
```

其中

- 第二个--pid-file是$processname的运行参数
- 第一个--pidfile是daemon检测$processname是否已运行的选项

- 由于提供了--check $servicename，所以函数调用语句__pids_var_run $base [pidfile]中的$base等于$servicename，即表示检查$servicename进程是否允许；如果没有提供该选项，则检查的是$processname

在SysV脚本中，daemon会配合以下几个语句同时执行

```
echo -n $"Starting $prog: "
daemon --pidfile=${pidfile} $prog $OPTIONS
RETVAL=$?
[ $RETVAL = 0 ] && touch ${lockfile}
return $RETVAL
```

daemon函数启动程序时，自身就会调用success或failure函数，所以就不需再使用action函数了；如果不使用daemon函数启动服务，通常会配合action函数，例如：

```
$prog $OPTIONS
RETVAL=$?
[ $RETVAL -eq 0 ] && action "Starting $prog" /bin/true && touch ${lockfile}
```

2.3.17 0x0106 killproc

killproc函数的作用是根据给定程序名杀进程；中间它会获取程序名对应的pid号，且保证/proc目录下没有pid对应的目录才表示进程关闭成功

killproc函数的定义语句如下

```
# A function to stop a program.
killproc() {
    local RC killlevel= base pid pid_file= delay try binary=

    RC=0; delay=3; try=0
    # Test syntax.
    if [ "$#" -eq 0 ]; then
        echo $"Usage: killproc [-p pidfile] [ -d delay] {program} [-signal]"
        return 1
    fi
    if [ "$1" = "-p" ]; then
        pid_file=$2
        shift 2
    fi
    if [ "$1" = "-b" ]; then
        if [ -z $pid_file ]; then
            echo $"-b option can be used only with -p"
            echo $"Usage: killproc -p pidfile -b binary program"
            return 1
        fi
        binary=$2
        shift 2
    fi
    if [ "$1" = "-d" ]; then
        if [ "$?" -eq 1 ]; then
            echo $"Usage: killproc [-p pidfile] [ -d delay] {program} [-signal]"
            return 1
        fi
        shift 2
    fi


    # check for second arg to be kill level
    [ -n "${2:-}" ] && killlevel=$2
```

(continues on next page)

(续上页)

```
    # Save basename.
    base=${1##*/}

    # Find pid.
    __pids_var_run "$1" "$pid_file" "$binary"
    RC=$?
    if [ -z "$pid" ]; then
        if [ -z "$pid_file" ]; then
            pid="$(__pids_pidof "$1")"
        else
            [ "$RC" = "4" ] && { failure $"$base shutdown" ; return $RC ;}
        fi
    fi

    # Kill it.
    if [ -n "$pid" ] ; then
        [ "$BOOTUP" = "verbose" -a -z "${LSB:-}" ] && echo -n "$base "
        if [ -z "$killlevel" ] ; then
            __kill_pids_term_kill -d $delay $pid
            RC=$?
            [ "$RC" -eq 0 ] && success $"$base shutdown" || failure $"$base␣
↪shutdown"
        # use specified level only
        else
            if checkpid $pid; then
                kill $killlevel $pid >/dev/null 2>&1
                RC=$?
                [ "$RC" -eq 0 ] && success $"$base $killlevel" || failure $"$base
↪$killlevel"
            elif [ -n "${LSB:-}" ]; then
                RC=7 # Program is not running
            fi
        fi
    else
        if [ -n "${LSB:-}" -a -n "$killlevel" ]; then
            RC=7 # Program is not running
        else
            failure $"$base shutdown"
            RC=0
            __kill_pids_term_kill -d $delay $pid
            RC=$?
            [ "$RC" -eq 0 ] && success $"$base shutdown" || failure $"$base␣
↪shutdown"
        # use specified level only
        else
            if checkpid $pid; then
                kill $killlevel $pid >/dev/null 2>&1
                RC=$?
                [ "$RC" -eq 0 ] && success $"$base $killlevel" || failure $"$base
↪$killlevel"
            elif [ -n "${LSB:-}" ]; then
                RC=7 # Program is not running
            fi
        fi
    else
        if [ -n "${LSB:-}" -a -n "$killlevel" ]; then
            RC=7 # Program is not running
        else
            failure $"$base shutdown"
            RC=0
        fi
```

(continues on next page)

(续上页)

```
    fi

    # Remove pid file if any.
    if [ -z "$killlevel" ]; then
        rm -f "${pid_file:-/var/run/$base.pid}"
    fi
    return $RC
}
```

由上述代码可知：关闭进程时，需要再三确定pid文件是否存在，/proc下是否有和pid对应的目录。直到/proc下已经没有了和pid对应的目录时，才表示进程真正杀死了；但此时pid文件仍可能存在，因此还要保证pid文件已被移除

该函数的调用方法如下

```
killproc [-p pidfile] [ -d delay] {program} [-signal]
```

其中

- -p pidfile：用于指定从此文件中获取进程的pid号，不指定时默认从/var/run/$base.pid中获取
- -d delay：指定未使用-signal时的延迟检测时间；有效单位为秒、分、时、日("smhd")，不写时默认为秒
- -signal：用于指定kill发送的信号；如果不指定，则默认先发送TERM信号，在-d delay时间段内仍不断检测是否进程已经被杀死，如果还未死透，则delay超时后发送KILL信号强制杀死

需要明确的是，只有/proc目录下没有了pid对应的目录才算是杀死了； 一般来说，killproc前会判断进程是否已在运行，最后还要删除pid文件和lock文件；当然，killproc函数可以保证pid文件被删除；所以，killproc函数大致会同时配合以下语句用来杀进程

```
status -p ${pidfile} $prog > /dev/null
if [[ $? = 0 ]]; then
        echo -n $"Stopping $prog: "
        killproc -p ${pidfile} -d ${STOP_TIMEOUT} $httpd
else
        echo -n $"Stopping $prog: "
        success
fi
RETVAL=$?
[ $RETVAL -eq 0 ] && rm -f ${lockfile} ${pidfile}
```

同样注意，killproc中已经自带success和failure函数；如果不使用killproc杀进程，则通常会配合action函数或者success、‘‘failure‘‘；大致如下

```
killall $prog ;  usleep 50000 ;  killall $prog
RETVAL=$?
if [ "RETVAL" -ne 0 ];then
    action $"Stopping $prog: " /bin/true
    rm -rf ${lockfile} ${pidfile}
else
    action $"Stoping $prog: " /bin/false
fi
```

以上由于采用的是killall命令，如果采用的是kill命令，则需要先获取进程的pid，在此之前还要检查pid文件是否存在

2.3.18 0x0107 status

status函数用于获取进程的运行状态，有以下几种状态

- ${base} (pid $pid) is running...
- ${base} dead but pid file exists
- ${base} status unknown due to insufficient privileges
- ${base} dead but subsys locked
- ${base} is stopped

status函数定义语句如下(注意：此为CentOS 7上语句，比CentOS 6多了一段systemctl的处理，用于Sysv的status状态向systemd的status状态转换)

```
status() {
    local base pid lock_file= pid_file= binary=

    # Test syntax.
    if [ "$#" = 0 ] ; then
        echo $"Usage: status [-p pidfile] {program}"
        return 1
    fi
    if [ "$1" = "-p" ]; then
        pid_file=$2
        shift 2
    fi
    if [ "$1" = "-l" ]; then
        lock_file=$2
        shift 2
    fi
    if [ "$1" = "-b" ]; then
        if [ -z $pid_file ]; then
            echo $"-b option can be used only with -p"
            echo $"Usage: status -p pidfile -b binary program"
            return 1
        fi
        binary=$2
        shift 2
    fi
    base=${1##*/}

    if [ "$_use_systemctl" = "1" ]; then
        systemctl status ${0##*/}.service
        ret=$?
        # LSB daemons that dies abnormally in systemd looks alive in systemd's␣
↪eyes due to RemainAfterExit=yes
        # lets adjust the reality a little bit
        if systemctl show -p ActiveState ${0##*/}.service | grep -q '=active$' && \
        systemctl show -p SubState ${0##*/}.service | grep -q '=exited$' ; then
            ret=3
        fi
        return $ret
    fi

    # First try "pidof"
    __pids_var_run "$1" "$pid_file" "$binary"
    RC=$?
    if [ -z "$pid_file" -a -z "$pid" ]; then
        pid="$(__pids_pidof "$1")"
    fi
    if [ -n "$pid" ]; then
        echo $"${base} (pid $pid) is running..."
        return 0
    fi
```

(continues on next page)

(续上页)

```
    case "$RC" in
    0)
        echo $"${base} (pid $pid) is running..."
        return 0
        ;;
    1)
        echo $"${base} dead but pid file exists"
        return 1
        ;;
    4)
        echo $"${base} status unknown due to insufficient privileges."
        return 4
        ;;
    esac
    if [ -z "${lock_file}" ]; then
        lock_file=${base}
    fi
    # See if /var/lock/subsys/${lock_file} exists
    if [ -f /var/lock/subsys/${lock_file} ]; then
        echo $"${base} dead but subsys locked"
        return 2
    fi
    echo $"${base} is stopped"
    return 3
}
```

该函数的调用方法如下

```
status [-p pidfile] [-l lockfile] program
# 如果同时提供了-p和-l选项，-l选项必须放在-p选项后面
```

CHAPTER 3

脚本示例

脚本示例有

3.1 示例脚本

- 写一个脚本，实现如下功能：让用户通过键盘输入一个用户名，如果用户存在，就显示其用户名和UID，否则，就显示用户不存在

```
#!/bin/bash

read -p "please input userName: " userName
if grep "^$userName\>" /etc/passwd &> /dev/null;then
        echo "$userName :`id -u $userName`";
else
        echo "$userName is not exist !!";
fi
```

- 写一脚本，实现如下功能
  - 1、让用户通过键盘输入一个用户名，如果用户不存在就退出
  - 2、如果用户的UID大于等于500，就说明它是普通用户
  - 3、否则，就说明这是管理员或系统用户

```
#!/bin/bash

read -p "please input userName: " userName

if ! grep "^$userName\>" /etc/passwd &> /dev/null;then
        echo "$userName  not exist"
        exit 6
fi

uid=`id -u $userName`
if [ $uid -ge 500 ];then
        echo "The $userName is common user"
else
```

(continues on next page)

(续上页)

```
        echo "The $userName is system user"
fi
```

- 写一脚本，实现如下功能
  - 1、让用户通过键盘输入一个用户名，如果用户不存在就退出
  - 2、如果其UID等于其GID，就说它是个”good guy”
  - 3、否则，就说它是个”bad guy”

```
#!/bin/bash
read -p "please input userName: " userName

if ! grep "^$userName\>" /etc/passwd &> /dev/null;then
        echo "$userName not exist"
        exit 62
fi

if [ `id -u $userName` -eq `id -g $userName` ];then
        echo "$userName is good guy"
else
        echo "$userName is bad guy"
fi
```

- 判断当前系统的所有用户是goodguy还是badguy

```
#!/bin/bash

for userName in `cut -d: -f1 /etc/passwd`;do
        if [ `id -u $userName` -eq `id -g $userName` ];then
                echo "$userName is good guy"
        else
                echo "$userName is bad guy"
        fi
done
```

- 写一个脚本，实现如下功能
  - 1、添加10个用户stu1-stu10；但要先判断用户是否存在
  - 2、如果存在，就用红色显示其已经存大在
  - 3、否则，就添加此用户；并绿色显示
  - 4、最后显示一共添加了几个用户

```
declare -i userCount=0

for i in {1..10};do
        if grep "^stu$i\>" /etc/passwd &> /dev/null;then
                echo -e "\033[31mstu$i\033[0m exist"
        else
                useradd stu$i && echo -e "useradd \033[32mstu$i\033[0m finished"
                let userCount++
        fi
done

echo "Add $userCount users"
```

- 判断当前系统中所有用户是否拥有可登录shell

```
#!/bin/bash

for userName in `cut -d: -f1 /etc/passwd`; do
        if [[ `grep "^$userName\>" /etc/passwd | cut -d: -f7` =~ sh$ ]];then
                echo "login shell user: $userName"
        else
                echo "nologin shell user: $userName"
        fi
done
```

- 写一个脚本，实现如下功能
  - 1.显示如下菜单
    - cpu) show cpu info
    - mem) show memory info
    - quit) quit
  - 2.如果用户选择cpu，则显示/proc/cpuinfo的信息
  - 3.如果用户选择mem，则显示/proc/meminfo的信息
  - 4.如果用户选择quit，则退出，且退出码为5
  - 5.如果用户键入其它字符，则显示未知选项，请重新输入

```
#!/bin/bash

info="cpu) show cpu info\nmem) show memory info\nquit) quit"
while true;do
        echo -e $info

        read -p "Enter your option: " userOption
        userOption=`echo $userOption | tr 'A-Z' 'a-z'`

        if [[ "$userOption" == "cpu" ]];then
                cat /proc/cpuinfo
        elif [[ "$userOption" == "mem" ]];then
                cat /proc/meminfo
        elif [[ "$userOption" == "quit" ]];then
                echo "quit"
                retValue=5
                break
        else
                echo "unkown option"
                retValue=6
        fi
done

[ -z $retValue ] && retValue=0

exit $retValue
```

- 写一个脚本，实现如下功能
  - 1.分别复制/var/log下的文件至/tmp/logs目录中
  - 2.复制目录时，使用cp -r
  - 3.复制文件时，使用cp
  - 4.复制链接文件时，使用cp -d
  - 5.余下的类型，使用cp -a

```
#!/bin/bash

targetDir='/tmp/logs'

[ -e $targetDir ] && mkdir -p $targetDir

for fileName in /var/log/*;do
        if [ -d $fileName ]; then
                copyCmd='cp -r'
        elif [ -f $fileName ]; then
                copyCmd='cp'
        elif [ -h $fileName ]; then
                copyCmd='cp -d'
        else
                copyCmd='cp -a'
        fi

        $copyCmd $fileName $targetDir
done
```

- 写一个脚本，使用形式：`userinfo.sh -u username [-v {1|2}]`
  - -u选项用于指定用户，而后脚本显示用户的UID和GID
  - -v选项后面是1，则显示用户的家目录路径；如果是2，则显示用户的家目录路径和shell

```
#!/bin/bash

[ $# -lt 2 ] && echo "less arguments" && exit 3

if [[ "$1" == "-u" ]]; then
        userName="$2"
        shift 2         # 剔除前2个位置参数
fi

if [[ $# -ge 2 ]] && [ "$1" == "-v" ]; then
        verFlag=$2
fi

verFlag=${verFlag:-0}

if [ -n $verFlag ]; then
        if ! [[ $verFlag =~ [012] ]]; then
                echo "Wrong Parameter"
                echo "Usage: `basename $0` -u UserName -v {1|2}"
                exit 4
        fi
fi

if [ $verFlag -eq 1 ];then
        grep "^$userName" /etc/passwd | cut -d: -f1,3,4,6
elif [ $verFlag -eq 2 ];then
        grep "^$userName" /etc/passwd | cut -d: -f1,3,4,6,7
else
        grep "^$userName" /etc/passwd | cut -d: -f1,3,4
fi
```

- 写一个脚本，实现功能如下
  - 提示用户输入一个用户名，判断用户是否登录了当前系统
  - 如果没有登录，则停止5秒之后，再次判定；直到用户登陆系统，显示用户来了，然后退出

```
#!/bin/bash

read -p "Enter a user name: " userName

# 判断输入是否为空并且是否存在该用户
until [ -n "$userName" ] && id $userName &> /dev/null; do
        read -p "Enter a user name again: " userName
done

until who | grep "^$userName" &> /dev/null; do
        echo "$userName is offline"
        sleep 5
done

echo "$userName is online"
```

- 写一个脚本，实现功能如下
  - 1.提示用户输入一个磁盘设备文件路径不存在或不是一个块设备，则提示用户重新输入，知道输入正确为止，或者输入quit以9为退出码结束脚本
  - 2.提示用户”下面的操作会清空磁盘的数据，并提问是否继续”。如果用户给出字符y或yes，则继续，否则，则提供以8为退出码结束脚本
  - 3.将用户指定的磁盘上的分区清空，而后创建两个分区，大小分别为100M和512M
  - 4.格式化这两个分区
  - 5.将第一个分区挂载至/mnt/boot目录，第二个分区挂载至/mnt/sysroot目录

```
#! /bin/bash
read -p "Enter you dev " devdir
umonut /mnt/boot
umonut /mnt/sysroot

while [[ "$devdir" != "quit" ]];do
        [ -a $devdir ] && [ -b $devdir ]
        if [[ $? -eq 0 ]];then
                read -p "Are you sure[y|yes]: " option
                if [[ "$option" == "y" || "$option" == "yes" ]];then
                        dd if=/dev/zero of=$devdir bs=512 count=1 &> /dev/null
                        echo -e "n\np\n1\n\n+100M\nn\np\n2\n\n+512M\nw" | fdisk
↪$devdir
                        mke2fs -t ext4 ${devdir}1
                        mke2fs -t ext4 ${devdir}2
                        mount ${devdir}1 /mnt/boot
                        mount ${devdir}2 /mnt/sysroot
                        echo "${devdir}1 /mnt/boot ext4 default 0 0" >> /etc/fstab
                        echo "${devdir}2 /mnt/sysroot ext4 default 0 0" >> /etc/
↪fstab
                        exit 7
                else
                        exit 8
                fi
    else
        read -p "Enter you dev again:  "  devdir
    fi
done

exit 9
```

- 写一个脚本，实现功能如下
  - 提示用户输入一个目录路径

  - 显示目录下至少包含一个大写字母的文件名

```
#! /bin/bash

while true; do
        read -p "Enter a directory: " dirname
        [ "$dirname" == "quit" ] && exit 3
        [ -d "$dirname" ] && break || echo "wrong directory..."
done

for filename in $dirname/*;do
        if [[ "$fileName" =~ .*[[:upper:]]{1,}.* ]]; then
        echo "$fileName"
    fi
done
```

- 写一个脚本，实现功能如下(前提是配置好yum源)
  - 1、如果本机没有一个可用的yum源，则提示用户，并退出脚本(4)；如果此脚本非以root用户执行，则显示仅有root才有权限安装程序包，而后退出(3)
  - 2、提示用户输入一个程序包名称，而后使用yum自动安装之；尽可能不输出yum命令执行中的信息；如果安装成功，则绿色显示，否则，红色显示失败
  - 3、如果用户输入的程序包不存在，则显示错误后让用户继续输入
  - 4、如果用户输入quit，则正常退出(0)
  - 5、正常退出前，显示本地共安装的程序包的个数

```
#!/bin/bash

while true;do
        if [ $UID -ne 0 ]; then
                echo "`basename $0` must be running as root"
                exit 3
        fi

        yum repolist &> /dev/null
        if  [[ $? -eq 0 ]];then
                while true; do
                        read -p "Enter a pakage: " pacName
                        if [[ "$pacName" == "quit" ]];then
                                rpm -qa | wc -l
                                exit 0
                fi

                        yum list | grep "^$pacName.*" &> /dev/null
                if [[ $? -eq 0 ]];then
                        yum install $pacName -y &> /dev/null
                        if [[ $? -ne 0 ]];then
                                echo "$pacName install fail"
                        else
                                echo "$pacName install success"
                        fi
                else
                        echo "$pacName is not exist"
                        continue
                fi
                done
        else
                echo "yum repo is not ok!"
                exit 4
        fi
done
```

- 写一个脚本，完成功能如下
  - 1.提示用户输入一个nice值
  - 2.显示指定nice指定进程名及pid
  - 3.提示用户选择要修改nice值的进程的pid和nice值
  - 4.执行修改
  - 5.别退出，继续修改

```
#!/bin/bash

if [[ $UID -eq 0 ]];then
   echo "keyi suibian tiao nice !"
else
   echo "zhineng tiaoda nice !"
fi

while true;do
        read -p "Enter a nice : " nicename
        [ "$nicename" == "quit" ] && exit 3
        /bin/ps axo nice,user,command,pid| grep "^[[:space:]]${nicename}\>"
        read -p "Enter a nice : " niceid
        read -p "Ener a PID :  " pidid
        /usr/bin/renice $niceid $pidid
done
```

- 写一个脚本，实现功能如下：能对/etc/进行打包备份，备份位置为/backup/etc-日期.后缀
  - 1.显示如下菜单给用户
    - xz) xz compress
    - gzip) gzip compress
    - bzip2) bzip2 compress
  - 2.根据用户指定的压缩工具使用tar打包压缩
  - 3.默认为xz，输入错误则需要用户重新输入

```
#!/bin/bash

# 方法一
[ -d /backup ] || mkdir /backup
cat << EOF
xz) xz compress
gzip) gzip compress
bzip2) bzip2 compress
EOF

while true;do
        read -p "Enter a options :" tarname
        [[ "$tarname" == "quit" ]] && exit 5
        tarname=${tarname:-xz}                # tarname为空时给定默认值

        case $tarname in
                xz)
                        tar Jcf /backup/etc-`date +%F-%H-%M-%S`.tar.xz /etc/*
                        break
                        ;;
                gzip)
                        tar zcf /backup/etc-`date +%F-%H-%M-%S`.tar.gz /etc/*
                        break
```

(continues on next page)

(续上页)

```
                        ;;
                bzip2)
                        tar jcf /backup/etc-`date +%F-%H-%M-%S`.tar.bz2 /etc/*
                        break
                        ;;
                *)
                        echo "you Enter is wrong option!"
        esac
done


# 方法二

#!/bin/bash

[ -d /backup ] || mkdir /backup

cat << EOF
plz choose a compress tool:

xz) xz compress
gzip) gzip compress
bzip2) bzip2 compress
EOF

while true; do
        read -p "your optopn: " option
        option=${option:-xz}

        case $option in
        xz)
                compressTool="J"
                suffix='xz'
                break
                ;;
        gzip)
                compressTool="z"
                suffix='gz'
                break
                ;;
        bzip2)
                compressTool="j"
                suffix='bz2'
                break
                ;;
        *)
                echo "wrong option"
                ;;
        esac
done

tar ${compressTool}cf /backup/etc-`date +%F-%H-%M-%S`.tar.$suffix /etc/*
```

3.2 实用脚本

shell脚本常用来启动相关系统服务

- *memcached*服务启动脚本

3.2.1 0x00 memcached服务启动脚本

以下是memcached服务启动脚本的示例，是一个非常简单但却非常通用的SysV服务启动脚本

- 关于SysV服务启动脚本的详解请参考：如何写SysV服务管理脚本

```
#!/bin/bash
#
# chkconfig: - 86 14
# description: Distributed memory caching daemon

## Default variables
PORT="11211"
USER="nobody"
MAXCONN="1024"
CACHESIZE="64"
OPTIONS=""

RETVAL=0
prog="/usr/local/memcached/bin/memcached"
desc="Distributed memory caching"
lockfile="/var/lock/subsys/memcached"

. /etc/rc.d/init.d/functions
[ -f /etc/sysconfig/memcached ] && source /etc/sysconfig/memcached

start() {
        echo -n $"Starting $desc (memcached): "
        daemon $prog -d -p $PORT -u $USER -c $MAXCONN -m $CACHESIZE "$OPTIONS"
        RETVAL=$?
        echo
        [ $RETVAL -eq 0 ] && touch $lockfile
        return $RETVAL
}

stop() {
        echo -n $"Shutting down $desc (memcached): "
        killproc $prog
        RETVAL=$?
        echo
        [ $RETVAL -eq 0 ] && rm -f $lockfile
        return $RETVAL
}

restart() {
        stop
        start
}

reload() {
        echo -n $"Reloading $desc ($prog): "
        killproc $prog -HUP
        RETVAL=$?
        echo
        return $RETVAL
}

case "$1" in
  start)
        start
        ;;
  stop)
        stop
```

(continues on next page)

(续上页)

```
        ;;
  restart)
        restart
        ;;
  condrestart)
        [ -e $lockfile ] && restart
        RETVAL=$?
        ;;
  reload)
        reload
        ;;
  status)
        status $prog
        RETVAL=$?
        ;;
   *)
        echo $"Usage: $0 {start|stop|restart|reload|condrestart|status}"
        RETVAL=1
esac

exit $RETVAL
```