Shell编程

# Shell简介

Shell本身是一个用C语言编写的程序，它是用户使用Unix/Linux的桥梁，用户的大部分工作都是通过Shell完成的。Shell既是一种命令语言，又是一种程序设计语言。作为命令语言，它交互式地解释和执行用户输入的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

它虽然不是Unix/Linux系统内核的一部分，但它调用了系统核心的大部分功能来执行程序、建立文件并以并行的方式协调各个程序的运行。因此，对于用户来说，shell是最重要的实用程序，深入了解和熟练掌握shell的特性极其使用方法，是用好Unix/Linux系统的关键。

## Shell有两种执行命令的方式

交互式（Interactive）：解释执行用户的命令，用户输入一条命令，Shell就解释执行一条。

批处理（Batch）：用户事先写一个Shell脚本(Script)，其中有很多条命令，让Shell一次把这些命令执行完，而不必一条一条地敲命令。

## Shell的种类

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

bash是Linux标准默认的shell。bash由Brian Fox和Chet Ramey共同完成，是BourneAgain Shell的缩写，内部命令一共有40个。Linux使用它作为默认的shell是因为它有诸如以下的特色：

* 可以使用类似DOS下面的doskey的功能，用方向键查阅和快速输入并修改命令。
* 自动通过查找匹配的方式给出以某字符串开头的命令。
* 包含了自身的帮助功能，你只要在提示符下面键入help就可以得到相关的帮助。

sh 由Steve Bourne开发，是Bourne Shell缩写， 是Unix 标准默认的shell。

ash shell 是由Kenneth Almquist编写的，Linux中占用系统资源最少的一个小shell，它只包含24个内部命令，因而使用起来很不方便。

csh 是Linux比较大的内核，它由以William Joy为代表的共计47位作者编成，共有52个内部命令。该shell其实是指向/bin/tcsh这样的一个shell，也就是说，csh其实就是tcsh。

ksh 是Korn shell的缩写，由Eric Gisin编写，共有42条内部命令。该shell最大的优点是几乎和商业发行版的ksh完全兼容，这样就可以在不用花钱购买商业版本的情况下尝试商业版本的性能了。

***注意***：bash是 Bourne Again Shell 的缩写，是linux标准的默认shell ，它基于Bourne shell，吸收了C shell和Korn shell的一些特性。bash完全兼容sh，也就是说，用sh写的脚本可以不加修改的在bash中执行。

## Shell脚本语言与编译型语言

### 编译型语言

很多传统的程序设计语言，例如Fortran、Ada、Pascal、C、C++和Java，都是编译型语言。这类语言需要预先将我们写好的源代码(source code)转换成目标代码(object code)，这个过程被称作“编译”。运行程序时，直接读取目标代码(object code)。由于编译后的目标代码(object code)非常接近计算机底层，因此执行效率很高，这是编译型语言的优点。  
 但是，由于编译型语言多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象，往往实现一个简单的功能需要大量复杂的代码。例如，在C++里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。

### 解释型语言

解释型语言也被称作“脚本语言”。执行这类程序时，解释器(interpreter)需要读取我们编写的源代码(source code)，并将其转换成目标代码(object code)，再由计算机运行。因为每次执行程序都多了编译的过程，因此效率有所下降。  
 使用脚本编程语言的好处是，它们多半运行在比编译型语言还高的层级，能够轻易处理文件与目录之类的对象；缺点是它们的效率通常不如编译型语言。不过权衡之下，通常使用脚本编程还是值得的：花一个小时写成的简单脚本，同样的功能用C或C++来编写实现，可能需要两天，而且一般来说，脚本执行的速度已经够快了，快到足以让人忽略它性能上的问题。脚本编程语言的例子有awk、Perl、Python、Ruby与Shell。

## 运行Shell脚本方法

一般在shell脚本的第一行指定解析器参数：#!/bin/bash

### 作为可执行程序

chmod +x ./test.sh #使脚本具有执行权限

./test.sh #执行脚本

注意，一定要写成./test.sh，而不是test.sh。运行其它二进制的程序也一样，直接写test.sh，linux系统会去PATH里寻找有没有叫test.sh的，而只有/bin, /sbin, /usr/bin，/usr/sbin等在PATH里，你的当前目录通常不在PATH里，所以写成test.sh是会找不到命令的，要用./test.sh告诉系统说，就在当前目录找。

### 作为解释器参数

这种运行方式是，直接运行解释器，其参数就是shell脚本的文件名，如：

sh/bash test.sh

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

### Shell脚本的排错

1. shell脚本的三种调试方法

|  |
| --- |
| -n ：读一遍脚本中的命令但不执行，用来检查脚本中的语法错误  -v ：一边执行脚本，一边将执行过的脚本命令打印到标准输出端  -x ：提供跟踪执行信息，将执行的每一条命令和结果一次打印出来  这应该就像C语言的gdb一样有用吧，有助于检查错误 |

1. 使用这些选项有三种方法

|  |
| --- |
| 1. 在命令行提供参数：$sh -x script.sh 2. 脚本开头提供参数：#!/bin/sh -x 3. 在脚本中用set命令启用or禁用参数：其中set -x表启用，set +x表禁用 |

1. Shell脚本常见错误

|  |
| --- |
| 1. 在写脚本的时候，需特别注意添加空格或换行或分号。 2. test命令： $[ 5 -lt 10 ] 左方括号后须有空格，右方括号前也须有空格 3. shell脚本中函数定义： foo() {后须有空格或换行 4. shell脚本中，若同一行内写有两条命令须加分号。 |

# Shell的基本功能

## Bash命令的基本操作

**复制与粘贴**：鼠标左键复制，中键粘贴。

**bash快捷键：**

* Ctrl+A：切换到命令行开始
* Ctrl+E：切换到命令行结尾
* Ctrl+L：清除屏幕内容
* Ctrl+U：剪切清除光标之前的内容
* Ctrl+K：剪切清除光标之后的内容
* Ctrl+Y：粘贴刚才所删除的字符
* Ctrl+W：移除光标前一个单词
* Alt+D：移除光标后一个单词
* Ctrl+R：在历史命令中查找
* Ctrl+C：终止进程
* Ctrl+D：退出当前终端
* Ctrl+Z：后台运行，当用户退出时候进程终止。应使用nohup 命令&
* !!：重复执行最后一条命令
* 命令历史：↑↓或者使用history命令，通过”!数字”来访问历史命令。
* Tab键：自动补全

## 命令别名

命令别名定义：

范例：alias copy=cp

alias xrm= “rm -r ”

查看别名信息：alias

删除别名：unalias copy

注意：修改~/.bashrc文件，可以永久添加和删除别名。系统查找命令的顺序：命令绝对路径-->别名-->bash内部命令-->环境变量中查找命令

## 输入输出重定向

同标准I/O一样，Shell对于每一个进程预先定义3个文件描述字（0、1、2）。分别对应于：0 （STDIN）标准输入；1 （STDOUT）标准输出；2 （STDERR）标准错误输出。默认情况标准输入为键盘，标准输出和错误输出为显示器。

### 输出重定向

说明：>>会保留原来文件，在原来文件上追加内容。>会直接覆盖原来内容。

2代表错误输出重定向。>&表示标识输出重定向：将一个表示的输出重定向到另一个标识的输入。例如：

|  |
| --- |
| #命令执行成功时候输出到stdout.txt ，执行错误时候输出到stderr.txt  #文件不存在的时候会自动创建  find / -type f –name \*.txt > stdout.txt 2>stderr.txt  #执行命令发生错误时候，重定向到标准输出1  find / -type f –name \*.txt > stdout.txt 2>&1  #命令执行成功和失败都重定向到stdout.txt  find / -type f –name \*.txt &> stdout.txt |

**正确输出和错误输出同时保存**：命令 &>文件 或者 命令 &>> 文件

**正确数输出和错误输出保存在不同文件**：命令 >> 文件1 2>> 文件2

### 输入重定向(<)

标准输入重定向可以将原本应由从标准输入设备中读取的内容转由文件输入。

|  |
| --- |
| cat < install.log |

### Here Cocument

用于在命令或者脚本中按行输入文本。Here Cocument的格式为:

<<delimiter ,其中delimiter是一个用于标注的“分隔符”，该分隔符后所有输入都被当做文本来输入，知道下一个分隔符为止。

|  |
| --- |
| cat > hello.txt <<eof  hello  world  eof |

## 管道符

管道：将一个命令的输出传送给另一个命令，

作为另一个命令的输入。

使用方法：

命令1|命令2|命令3……|命令n

|  |
| --- |
| --举例  ls -l /etc | more  ls -l /etc | grep init | wc -l |

## 命令连接符

；：用；间隔的各命令按顺序依次执行。

**&&：**前后命令的执行存在逻辑与关系，只有&&前面的命令执行成功后，它后面的命令才被执行。

**||**：前后命令的执行存在逻辑或关系，只有||前面的命令执行失败后，它后面的命令才被执行。(短路逻辑)

# Shell内建命令

|  |
| --- |
| 确定内建命令：type    执行程序：.，source  别名：alias  删除别名：unalias  任务前后台切换：bg，fg，jobs  改变目录：cd  声明变量：declare,typeset  打印字符：echo  跳出循环：break  循环控制continue  将所跟的参数作为shell的输入，并执行产生的命令：eval  执行命令来取代当前的shell：exec  退出shell：exit  是变量能被子shell识别：export  发送信号给指定PID或进程：kill  整数运算：let  显示当前工作目录：pwd  声明局部变量：local  从标准输入读取一行到变量：read  定义函数的返回值：return  向左移动位置参数：shift  显示并设置进程资源限度：ulimit  测试表达式：test |

# Shell编程基础

## 标识符

标识符用于标识变量或者函数的名字，有以下限制：

* 必须以字母或者下划线开头
* 变量包含字符：字母，数字，下划线
* 长度不限
* 变量不能是shell的关键字，和环境变量同名

## 变量

### 局部变量

局部变量就是指在某个shell中生效，对其他shell来说是无效的，而且会随着当前Shell的消失而消失，局部变量的作用域被限定在声明它们的shell中，可以使用local内建命令来显示的声明局部变量，仅限于函数中使用。

### 环境变量

环境变量通常又称为全局变量。可以使用export var=value来设置环境变量，环境变量可以在子shell中传递。env用于查询环境变量。

|  |
| --- |
| export命令  语法：export [-fnp][变量名称]=[变量设置值]  参数说明：   * -f 　代表[变量名称]中为函数名称。 * -n 　删除指定的变量。变量实际上并未删除，只是不会输出到后续指令的执行环境中。 * -p 　列出所有的shell赋予程序的环境变量。   示例：  export -p //列出当前的环境变量值  export PATH=$PATH:$JAVA\_HOME/bin |
| 常见的环境变量  BASH：Bash Shell的全路径  BASH\_VERSION：Bash Shell版本  CDPATH：用于快速进入某目录  EUID：用于记录当前用UID  FUNCNAME：当前函数体的函数名  HISTCMD：下一条命令在history命令中的编号。  HISTFILE：记录history命令记录的文件位置  HISTFILESIZE：设置HISTFILE文件记录命令的行数  HOSTNAME：主机名  HOSTTYPE：主机的机构，有i386,i686,x86\_64等  LANG：设置 当前系统的语言环境  PWD：记录当前的工作目录  OLDPWD：记录之前的目录  PATH： 命令的搜索路径  PS1：命令提示符 |
| 环境变量配置文件的种类：/etc/profile;/etc/profile.d/\*.sh;~/.bash\_profile;~/.bashrc;/etc/bashrc  配置文件的读取顺序：   * /etc/profile--->~/.bash\_profile--->~/.bashrc--->/etc/bashrc--->/etc/profile.d/\*.sh--->命令提示符 * /etc/profile--->/etc/profile.d/\*.sh--->命令提示符   各配置文件的作用：   * /etc/profile的作用：USER变量，LOGNAME变量，MAIL变量，PATH变量，HOSTNAME变量，HISTSIZE变量，umask，调用/etc/profile.d/\*.sh文件 * ~/.bash\_profile的作用：调用了~/.bashrc文件；在PATH变量中加入了"$HOME/bin"这个目录 * ~/.bashrc的作用：定义了系统别名；调用了/etc/bashrc * /etc/bashrc的作用：PS1变量；umask；PATH;调用了/etc/profile.d/\*.sh文件   配置文件直接生效：source(.) 配置文件 |

### 自定义变量赋值和取值

|  |
| --- |
| 定义变量：变量名不加美元符号（$）。注意，变量名和等号之间不能有空格，这可能和你熟悉的所有编程语言都不一样。  同时，变量名的命名须遵循如下规则：   * 首个字符必须为字母（a-z，A-Z）。 * 中间不能有空格，可以使用下划线（\_）。 * 不能使用标点符号。 * 不能使用bash里的关键字（可用help命令查看保留关键字）。   变量定义举例：myNum=100  使用变量：要在变量名前面加美元符号（$）：echo ${myNum}。推荐给所有变量加上花括号，这是个好的编程习惯。  重新定义变量：myNum=200  只读变量：readonly myNum  查看所有变量：set  删除变量：unset [-fv] [变量或函数名称]  选项：   -f 　仅删除函数。   -v 　仅删除变量。 |
| declare变量的声明  格式：declare [+/-] [选项] 变量名  选项：   * [+/-]：给变量设定或者取消变量属性 * -i:给变量声明为整数型 * -x:给变量声明为环境变量 * -p:查看变量的类型 |
| 命令格式：read [选项] 变量  选项:   * -p:提示信息，等待read输入时，输出提示信息。 * -t 秒数:指定用户输入等待时间 * -n 字符数：指定接收的字符树 * -s:隐藏输入的内容，适用于机密的信息输入。 |
| ##Shell echo命令  显示变量：echo "${mouth}-1-2009"  显示转义字符：echo "\"It is a test\""  原样输出字符串：echo '$name\"'  显示命令执行结果：echo `date`  显示结果重定向至文件：echo "It is a test" > myfile  ###shell printf命令  printf 命令用于格式化输出， 是echo命令的增强版。它是C语言printf()库函数的一个有限的变形，并且在语法上有些不同。注意：printf 由 POSIX 标准所定义，移植性要比 echo 好。  命令格式：printf '输出类型输出格式' 输出内容  ***输出类型***：   * %ns:输出字符串。n是数字代表输出几个字符 * %ni:输出整数。n代表输出几个数字 * %m.nf:输出浮点数。m和n是数字，指代输出的整数位数和小数位数。   ***输出格式***：\a \b \f \n \r \t \v  例如：printf '%s %s %s' 1 2 3 4 5 6(会把三个当做一组)  注意：printf不能接收标准输入，不接受文件。通常和awk一起使用  awk中支持print和printf，区别在于print自动换行。 |

### 特殊变量

|  |
| --- |
| $0：当前脚本的文件名  $n：传递给脚本或函数的参数。n 是一个数字，表示第几个参数。大于9时候，需要使用${}  $#：传递给脚本或函数的参数个数。  $\*：传递给脚本或函数的所有参数。  $@：传递给脚本或函数的所有参数。被双引号(" ")包含时，与 $\* 稍有不同，下面将会讲到。  $?：上个命令的退出状态，或函数的返回值。退出状态是一个数字，一般情况下，大部分命令执行成功会返回 0，失败返回 1。  $$：当前Shell进程ID。对于 Shell 脚本，就是这些脚本所在的进程ID。 |

### 数组

|  |
| --- |
| #定义数组  array\_name=(value1 ... valuen)  #读取数组  读取数组元素值的一般格式是：${array\_name[index]}  ${array\_name[\*]} #获取所有元素  ${array\_name[@]} #获取所有元素  # 取得数组元素的个数  length=${#array\_name[@]}  length=${#array\_name[\*]}  # 取得数组单个元素的长度  lengthn=${#array\_name[n]}  #取消数组  unset array[n]  unset array |

### 变量的作用域

Shell变量的作用域是在本shell内，属于shell的全局变量，也就是从定义该变量的地方开始到shell结束，或到主动使用unset命令结束为止。所以，在本shell中多人维护的脚本是互相影响的，所以我们需要使用只能在函数中生效的local局部变量，这样这些变量将只能在局部的命令空间中

## 转义和引用

Linux中有两种字符，一类是普通字符，除了本身字面意思外没有其他特殊含义；另一类是元字符，是shell中的保留字符，有特殊含义。

### 转义

|  |
| --- |
| \'：'  \"："  \%：%  \$：$  \\：反斜杠  \a：警报，响铃  \b：退格（删除键）  \f：换页(FF)，将当前位置移到下页开头  \n：换行  \r：回车  \t：水平制表符（tab键）  \v：垂直制表符 |

### 引用

|  |
| --- |
| #部分引用：用双引号括起来的引用。这种引用中$符,反引号``,转义符\依然会被解析#为特殊字符    #全引用：单引号括起来的所有内容。单引号中的任意字符都会当成普通字符 |

### 命令替换

命令替换指命令可以先执行命令，将输出结果暂时保存，在适当的地方输出。

语法：`命令command`或者使用$(command)



## 运算符

### 算数运算符

|  |
| --- |
| + 加法 `expr $a + $b` 结果为 30。  - 减法 `expr $a - $b` 结果为 10。  \* 乘法 `expr $a \\* $b` 结果为 200。元字符需要转义  / 除法 `expr $b / $a` 结果为 2。  % 取余 `expr $b % $a` 结果为 0。  = 赋值 a=$b 将把变量 b 的值赋给 a。  == 相等。用于比较两个数字，相同则返回 true。 [ $a == $b ] 返回 false。  != 不相等。用于比较两个数字，不相同则返回 true。 [ $a != $b ] 返回 true。  ++,--：自增自减 |

注意：条件表达式要放在方括号之间，并且要有空格，例如 [$a==$b] 是错误的，必须写成 [ $a == $b ]。可以使用$((运算表达式))代替，这种方式进行计算，运算符允许空格。

#整数运算可以使用$(( 算数表达式 ))

#浮点数运算bc命令：其中scale代表有效数字，分号后面是运算式。

n=`echo "scale=3;77/3" | bc`

n=$(echo "scale=3;77/3" | bc)

### 关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

|  |
| --- |
| -eq：检测两个数是否相等，相等返回 true。 [ $a -eq $b ] 返回 true。  -ne：检测两个数是否相等，不相等返回 true。 [ $a -ne $b ] 返回 true。  -gt：检测左边的数是否大于右边的，如果是，则返回 true。 [ $a -gt $b ] 返回 false。  -lt：检测左边的数是否小于右边的，如果是，则返回 true。 [ $a -lt $b ] 返回 true。  -ge：检测左边的数是否大等于右边的，如果是，则返回 true。 [ $a -ge $b ] 返回 false。  -le：检测左边的数是否小于等于右边的，如果是，则返回 true。 [ $a -le $b ] 返回 true。 |

### 布尔运算符

|  |
| --- |
| !：非运算，表达式为 true 则返回 false，否则返回 true。 [ ! false ] 返回 true。  -o：或运算，有一个表达式为 true 则返回 true。 [ $a -lt 20 -o $b -gt 100 ] 返回 true。  -a：与运算，两个表达式都为 true 才返回 true。 [ $a -lt 20 -a $b -gt 100 ] 返回 false。 |

### 字符串运算符

|  |
| --- |
| =或者== 检测两个字符串是否相等，相等返回 true。 [ $a = $b ] 返回 false。  !=：检测两个字符串是否相等，不相等返回 true。 [ $a != $b ] 返回 true。  -z：检测字符串长度是否为0，为0返回 true。 [ -z $a ] 返回 false。  -n：检测字符串长度是否为0，不为0返回 true。 [ -z $a ] 返回 true。  str：检测字符串是否为空，不为空返回 true。 [ $a ] 返回 true。 |

### 文件测试运算符

|  |
| --- |
| -b file：检测文件是否是块设备文件，如果是，则返回 true。 [ -b $file ] 返回 false。  -c file：检测文件是否是字符设备文件，如果是，则返回 true。 [ -b $file ] 返回 false。  -d file：检测文件是否是目录，如果是，则返回 true。 [ -d $file ] 返回 false。  -f file：检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。 [ -f $file ] 返回 true。  -g file：检测文件是否设置了 SGID 位，如果是，则返回 true。 [ -g $file ] 返回 false。  -k file：检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。 [ -k $file ] 返回 false。  -p file：检测文件是否是具名管道，如果是，则返回 true。 [ -p $file ] 返回 false。  -u file：检测文件是否设置了 SUID 位，如果是，则返回 true。 [ -u $file ] 返回 false。  -r file：检测文件是否可读，如果是，则返回 true。 [ -r $file ] 返回 true。  -w file：检测文件是否可写，如果是，则返回 true。 [ -w $file ] 返回 true。  -x file：检测文件是否可执行，如果是，则返回 true。 [ -x $file ] 返回 true。  -s file：检测文件是否为空（文件大小是否大于0），不为空返回 true。 [ -s $file ] 返回 true。  -e file：检测文件（包括目录）是否存在，如果是，则返回 true。 [ -e $file ] 返回 true。 |

## 特殊字符

### 通配符

\*通配符：任意字符串（不包括.开头文件）

?通配符：任何单个字符

[ ]：指定字符中的一个字符，其中^表示取反，-表示连续

### 注释符和注释

#开头后面都是注释字符

### 花括号

引用变量原型，又叫变量扩展

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

# Shell选择结构

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

## 条件判断语句

方式一：test 条件表达式

方式二：[ 条件表达式 ]

例如：[ 1 -le 2 ] && echo yes || echo no

test 1 –le 2 && echo yes || echo no

## Shell if else语句

|  |
| --- |
| 格式：  if [ 条件表达式 ]; then  command  elif [ 条件表达式 ]; then  command  else  command  fi |

## Shell case esac语句

|  |
| --- |
| case 值 in  模式11 [ | 模式12]) commands;;  模式21 [ | 模式22]) commands;;  …  \*) commands;;  esac  注意：模式中可以使用通配符，可以有多个模式，每个分支以;;结尾，命令可以是多个。 |

## Select结构化语法格式

select是一种简单生成菜单的方式，生成的菜单具有编号，在选入选择的表好后，可根据用户的选择执行相应的代码。通常与case结构一起使用，允许用户在菜单中选择，并居于选择执行相应的命令。当执行break语句，退出select结构。格式如下：

|  |
| --- |
| select 变量 in 列表  do  command  done |

# Shell循环结构

## for语句

|  |
| --- |
| for 变量 in 列表  do  command  done  或者  for((exp1;exp2;exp3))  do  command  done  注意：列表是一组值（数字、字符串等）组成的序列，每个值通过空格分隔。每循环一次，就将列表中的下一个值赋给变量。in 列表是可选的，如果不用它，for 循环使用命令行的位置参数。连续的数字可以使用..。 |

## while循环

|  |
| --- |
| while 条件表达式  do  循环执行的命令  done  #无限死循环  while true  do  循环执行的命令  done |

## until循环

|  |
| --- |
| until 条件表达式  do  循环执行命令  done  注意：command 一般为条件表达式，如果返回值为 false，则继续执行循环体内的语句，否则跳出循环。表示直到条件不满足，就跳出循环。 |

## Shell 控制语句

|  |
| --- |
| ***break***：break命令允许跳出所有循环（终止执行后面的所有循环）。  ***continue***：跳出当前循环。  ***exit命令***：等同于退出shell，并返回给定值。在shell脚本中可以终止当前脚本执行。执行exit可使shell以指定的状态值退出。若不设置状态值参数，则shell以预设值退出。状态值0代表执行成功，其他值代表执行失败。  语法：exit(参数)  参数：返回值：指定shell返回值。  ***shift命令***  格式：shift [n] n默认是1  作用：变量的个数整体向前移动(删除)n个。  例如，$#=5，shift 3,则原来的5个参数剩下2个变量，值分别对应$4,$5。 |

# Shell函数的使用

## shell函数的基本知识

函数可以让我们将一个复杂功能划分成若干模块，让程序结构更加清晰，代码重复利用率更高。像其他编程语言一样，Shell 也支持函数。Shell 函数必须先定义后使用。函数格式：

[function] function\_name(){

commands

return value

}

函数返回值又叫函数的退出状态，实际上是一种通讯方式。函数返回值，可以显式增加return语句；如果不加，会将最后一条命令运行结果作为返回值。Shell 函数返回值只能是整数，一般用来表示函数执行成功与否，0表示成功，其他值表示失败。如果 return 其他数据，比如一个字符串，往往会得到错误提示：“numeric argument required”。

## 带参数函数

在Shell中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1表示第一个参数，$2表示第二个参数... 注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当n>=10时，需要使用${n}来获取参数。

|  |
| --- |
| $#：传递给函数的参数个数。  $\*：显示所有传递给函数的参数。  $@：与$\*相同，但是略有区别，请查看Shell特殊变量。  $?：函数的返回值。 |
| #!/bin/bash  total=0  for i in $\*  do  total=$(($total+$i))  done  printf $total |

## 函数库

对于某些常用的功能，必须考虑将它们独立出来，集中放到独立的文件中，这个文件称为函数库。很多linux发型版本中都有自己的函数库，Shell 中包含脚本都加载系统自带函数库：/etc/init.d/functions。

文件包含：. filename或者source filename

|  |
| --- |
| source /etc/init.d/functions  #常见函数  checkpid()：检查某个PID是否存在  deamon()：以deamon方式启动某个服务  killproc()：停止某个进程  pidfileofproc()：价差某个进程的pid文件  pidofproc()：检查某个进程pid  status()：判断某个服务的状态  strstr()：检查$1是否包含$2字符串  confirm()：提示是否启动某个服务 |