# linux shell编程

## shell编程主要内容

### 变量

|  |
| --- |
| 变量命名：  变量名必须以字母或下划线开头，后面可以跟字母、数字或下划线。任何其他字符都标志变量名结束。  变量名大小写敏感。 |
| 变量类型：  根据变量的作用域，变量可以分为本地变量和环境变量。  本地变量只有在创建它们的shell程序中可用。  环境变量则在shell中的所有用户进程中可用，通常也称为全局变量。(export M1变成全局) |
| 变量赋值： variabl=value  等号两边不能有空格。  如果要变量附空值，可以在等号后面跟一个换行符。 |
| 显示变量的值：  echo $var 或 echo ${var} -->推荐这种写法  清除变量：  # unset variable  显示所有变量：  # set (env显示环境变量) |
| 只读变量：（不能重新赋值，不能被清除）  # readonly myvar |
|  |

#### 环境变量：（称为全局变量，按照惯例需要大写）

|  |
| --- |
| # export LANG  可被所有的shell环境下访问；如果父shell进程产生了子shell进程，则环境变量可被“继承”并复制。 |

#### 位置参量列表

|  |  |
| --- | --- |
| 位置参量（命令行参数）：  是一组特殊的内置变量，通常被shell脚本用来从命令行接收参数，或被函数用来保存传递他的参数。  执行shell脚本时，用户可以通过命令向脚本传递参数，跟在脚本后面的用空格隔开的每个字符串都成为位置变量。  $1表示第一个，$2第二个依次类推，直到$9，$10以后要用${10}表示 | |
| $0 | 当前脚本的文件名 |
| $1-$9 | 1-9位置参量 |
| ${10} | 第10个参量，类似有${12}... |
| $# | 位置参量的个数 |
| $\* | 以单字符显示所有的位置参量 |
| $@ | 未加双引号时与$\*含义相同，加双引号时有区别 |
| $$ | 脚本运行当前进程号 |
| $! | 最后一个后台运行的进程的进程号 |
| $? | 最后一个命令退出码 0表示没有错误 |

### 数组

|  |
| --- |
| 数组定义  arr=(math english chinese) |
| 数组初始化  arr=(math english chinese) |
| 数组引用  引用变量：${arr[0]}  数组个数：${#arr[\*]}  所有元素：${arr[\*]} |
| 数组赋值  arr[0]=chemical |

|  |  |
| --- | --- |
| 输入输出 |  |
| read |  |
| echo |  |
| printf |  |

|  |  |
| --- | --- |
| 条件测试 |  |
| 整数测试 |  |
| 逻辑测试 |  |
| 字符串测试 |  |

|  |  |
| --- | --- |
| 控制语句 |  |
| 条件 |  |
| 循环 |  |
| 分支 |  |

|  |
| --- |
| 函数 |
|  |

### 函数

|  |
| --- |
| 和其它编程语言一样， Bash 也可以定义函数 |
| 一个函数就是一个子程序，用于完成特定的任务，当有重复代码，或者一个任务只需要很少的修改就被重复几次执行时, 这时你应考虑使用函数。 |
| 函数的一般格式  function function\_name {  commands  }  function\_name() {  commands  }  #! /bin/bash  func1()  {  echo "the parameter's count:$#"  echo "the first parameter:$1"  echo "the second parameter:$2"  }  func1 a b |

#### 字符串操作

|  |
| --- |
|  |
| **注：pattern，old 中可以使用通配符。** |

#### 随机数和 expr 命令

|  |
| --- |
| echo $RANDOM |

#### expr：通用的表达式计算命令

|  |
| --- |
| 表达式中参数与操作符必须以空格分开，表达式中的运算可以是算术运算，比较运算，字符串运算和逻辑运算 |
| expr 5 % 3 |
| expr 5 \\* 3 # 乘法符号必须被转义 |
| 注意：目前比较少使用，可用$[ ... ]替换 |

#### shitf命令

|  |
| --- |
| 一般用于函数或者脚本程序参数处理，特别是参数多于10以上的时候 |
| 将所有参数变量向下移动一个位置,$2变成$1,$3变成$2，依次递进，但$0保持不变 |
| **while [ "$1" != "" ]**  **do**  **echo $1**  **shift**  **done** |

#### eval命令

|  |
| --- |
| 将所有的参数连接成一个表达式，并计算或执行该表达式，参数中的任何变量都将被展开 |
| eval arg1 [arg2] ... [argN] |
| listpage="ls -l | more"  $listpage |
| listpage="ls -l | more"  eval $listpage |

#### rap命令

|  |
| --- |
| trap command signal |
| command：  1．一般情况下是Linux命令  2．’ ’表示发生陷阱时为空指令，不做任何动作  3．’-’表示发生陷阱时采用缺省指令 |
| signal：  HUP(1) 挂起；一般因终端掉线或用户退出而引发  INT(2) 中断；一般因按下”Ctrl+C”组合键引发  QUIT(3) 退出；一般因按下”Ctrl+\”组合键引发  ABRT(6) 异常中止；一般因某些严重的执行错误而引发  ALRM(14) 闹钟；一般是超时时钟到来而引发  TREM(15) 中止；一般由系统在关机的时候发出 |
| #!/bin/bash  #安装2、3号信号  #处理代码为"rm -f tmp$$;exit 0"  trap "rm -f tmp$$;exit 0" 2 3  #生成文件，文件名为tmp+当前进程号  touch tmp$$  #睡眠60秒，以便向当前进程发送信号  sleep 60 |

### sed简单用法

|  |
| --- |
| sed 是一个精简的、非交互式的流式编辑器，它在命令行中输入编辑命令和指定文件名，然后在屏幕上查看输出 |
| **逐行**读取文件内容存储在临时**缓冲区**中，称为“模式空间”（pattern space），接着用**sed命令**处理缓冲区中的内容，处理完成后，把缓冲区的内容送往屏幕。接着处理下一行，这样不断重复，直到文件末尾。原文件内容并没有改变。 |

#### 用法

|  |
| --- |
|  |
| 1.sed ‘4,$d’ test.in  2.sed ‘3q’ test.in  3.sed ‘s/public/PUBLIC/’ test.in  4.sed -n ‘s/public/PUBLIC/p’ test.in |

#### 指定命令的方法

|  |
| --- |
| 1、用分号分隔命令  sed 's/public/PUBLIC/;s/north/NORTH/' test.in |
| 2、每个命令前放置-e  sed -e 's/public/PUBLIC/' -e 's/north/NORTH/' test.in |
| 3、使用分行命令功能，在输入单引号后按return键就会出现多行提示符（>）  sed '  > s/public/PUBLIC/  > s/north/NORTH/' test.in |

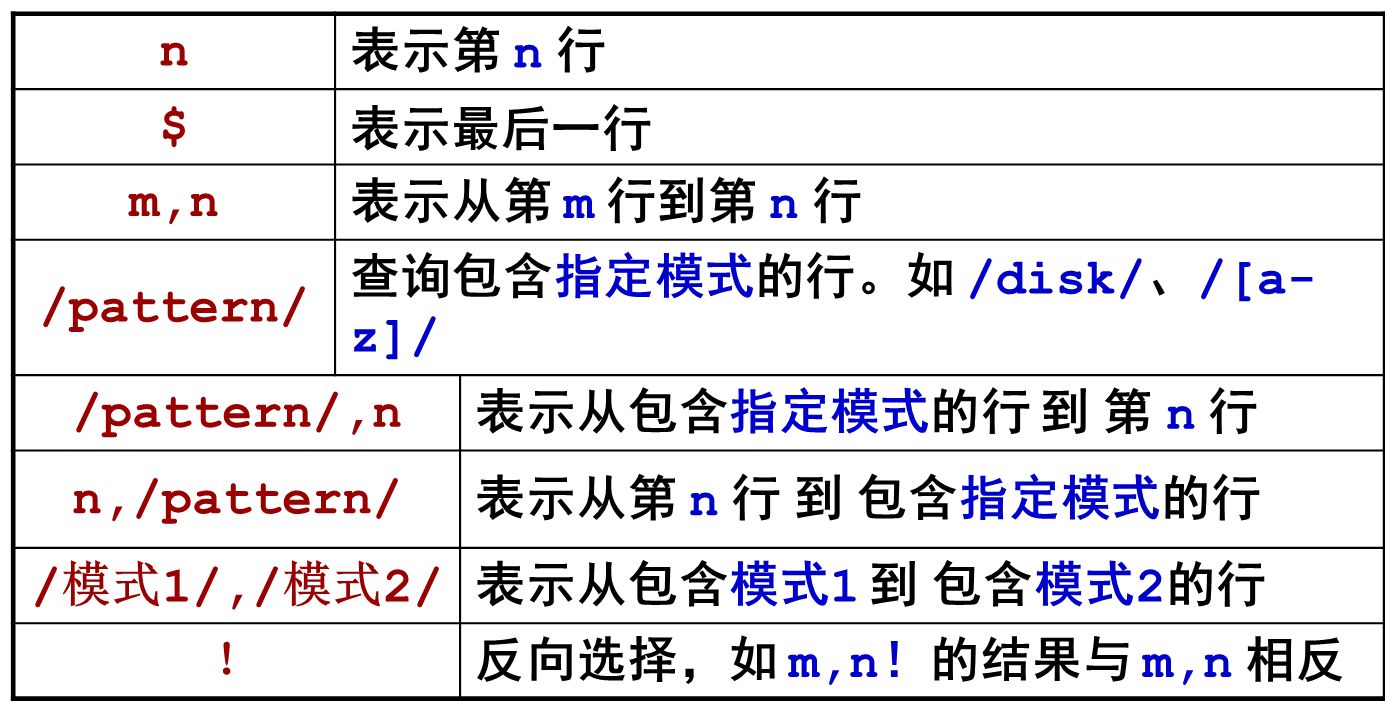
#### 脚本中使用命令

|  |
| --- |
| test.sed文件内容如下：  s/public/PUBLIC/  s/north/NORTH/  sed -f test.sed test.in |

#### 命令语法

|  |
| --- |
| **[address]sed\_edit\_cmd** (通常用单引号括起来)，其中 **address** 为 **sed** 的行定位模式，用于指定将要被  **sed** 编辑的行。如果省略，**sed** 将编辑所有的行。**sed\_edit\_cmd** 为 **sed** 对被编辑行将要进行的**编辑操作**。 |

##### 定义语法



##### 编辑命令

|  |
| --- |
| p ：打印匹配行  sed -n '3,5p' test.in // ('3,5!p')  sed -n '$p' test.in  sed -n '/north/p' test.in |
| **=** ：显示匹配行的行号  sed -n '/north/=' test.in |
| d ：删除匹配的行  sed '/north/d' test.in |
| 在指定行后面追加一行或多行文本，并显示添加的新内容，该命令主要用于 **sed** 脚本中。  sed '/north/a\AAA\  >BBB\  >CCC' test |
| i\ ：在指定行前插入一行或多行，并显示添加的新内容，使用格式同 a\ |
| c\ ：用新文本替换指定的行，使用格式同 a\ |
| l ：显示指定行中所有字符，包括控制字符(非打印字符)  sed -n '/north/l' test.in |
| s ：替换命令，使用格式为：  **[address]s/old/new/[gpw]**  **address** ：如果省略，表示编辑所有的行。  **g** ：全局替换  **p** ：打印被修改后的行  **w fname** ：将被替换后的行内容写到指定的文件中  **sed -n 's/north/NORTH/gp' test.in**  **sed -n 's/north/NORTH/w data' test.in**  **sed 's/[0-9][0-9]$/&.5/' datafile**  **& 符号用在替换字符串中时，代表 被替换的字符串** |
| r ：读文件，将另外一个文件中的内容附加到指定行后。  **sed '$r data' test.in** |
| **w** ：写文件，将指定行写入到另外一个文件中。  **sed -n '/public/w data2' test.in** |
| n ：将指定行的下面一行读入编辑缓冲区。  **sed -n '/public/{n;s/north/NORTH/p}' test.in**  对指定行同时使用多个 sed 编辑命令时，需用大括号 “ {} ” 括起来，命令之间用分号 “ ; ” 格开。注意与 -e 选项的区别 |
| **q** ：退出，读取到指定行后退出 **sed**  **sed '3q' test.in** |

#### 命令小结

|  |
| --- |
|  |

#### sed命令示例

|  |
| --- |
| 以下两个命令都能显式vsftpd.conf中的配置行  grep -v '^#' /etc/vsftpd/vsftpd.conf  cat /etc/vsftpd/vsftpd.conf | sed 's/^#.\*$//g'| sed '/^$/d' |

### awk简单用法

|  |
| --- |
| awk是什么？  **awk** 是一种用于处理数据和生成报告的编程语言  **awk** 可以在命令行中进行一些简单的操作，也可以被写成脚本来处理较大的应用问题  **awk** 与 **grep、sed** 结合使用，将使 shell 编程更加容易  **Linux** 下使用的 **awk** 是 **gawk** |
| awk如何工作？  **awk** 逐行扫描输入 ( 可以是文件或管道等 )，按给定的模式查找出匹配的行，然后对这些行执行 **awk** 命令指定的操作 |
| 与 **sed** 一样，**awk** 不会修改输入文件的内容。  可以使用重定向将 **awk** 的输出保存到文件中。 |

#### 用法

|  |
| --- |
|  |

|  |
| --- |
| awk -F: '{ print $1 }' /etc/passwd  awk -F: '{ print $1, $3 }' /etc/passwd  awk -F: '{ print $1 "|" $3 }' /etc/passwd  awk -F: '/root/{ print $1 "|" $3 }' /etc/passwd  awk -F: '/^root/' /etc/passwd |

#### awk\_script语法

|  |
| --- |
| **awk 'BEGIN {actions}**  **/pattern1/{actions}**  **......**  **/patternN/{actions}**  **END {actions}' input\_file**  **注意 BEGIN、和 END都是大写字母。**  其中 **BEGIN** **{actions}** 和 **END** **{actions}** 是可选的 |
| **awk\_script 可以由一条或多条 awk\_cmd 组成，每条 awk\_cmd 各占一行。** |
| **每个 awk\_cmd 由两部分组成：/pattern/{actions}** |
| **awk\_cmd 中的 /pattern/ 和 {actions} 可以省略，但不能同时省略；/pattern/ 省略时表示对所有的输入行执行指定的 actions；{actions} 省略时表示打印整行** |

#### awk 的执行过程

|  |
| --- |
| 1. 如果存在 **BEGIN** ，**awk** 首先执行它指定的 **actions** 2. **awk** 从输入中读取一行，称为一条输入记录 3. **awk** 将读入的记录分割成数个字段，并将第一个字段放入变量  **$1** 中，第二个放入变量 **$2** 中，以此类推；**$0** 表示整条记录；  字段分隔符可以通过选项 **-F** 指定，否则使用缺省的分隔符。 4. 把当前输入记录依次与每一个 **awk\_cmd** 中 **pattern** 比较：  如果相匹配，就执行对应的 **actions**；  如果不匹配，就跳过对应的 **actions**，直到完成所有的 **awk\_cmd** 5. 当一条输入记录处理完毕后，**awk** 读取输入的下一行，重复上  面的处理过程，直到所有输入全部处理完毕。 6. **awk** 处理完所有的输入后，若存在 **END**，执行相应的 **actions** 7. 如果输入是文件列表，**awk** 将按顺序处理列表中的每个文件。 |

#### awk 举例

|  |
| --- |
| **ifconfig | awk '/inet addr/{ print $2 }' | awk -F: '{ print $2 }'** |
| **ifconfig | awk '/inet addr/{ print $2 }' | awk -F: 'BEGIN { print "begin..."} { print $2 } END { print "end..."} '** |

#### 模式匹配

|  |
| --- |
| ①：使用正则表达式：/rexp/，如 /^A/、/A[0-9]\*/  **awk** 中正则表达式中常用到的元字符有: |
| ②：使用布尔 ( 比较 ) 表达式，表达式的值为真时执行相应的操作 (**actions**)  表达式中可以使用变量 ( 如字段变量 **$1,$2** 等 ) 和 **/rexp/**  表达式中的运算符有  关系运算符:  **< > <= >= == !=**  匹配运算符:  **~ !~   x ~ /rexp/** 如果 **x** 匹配 **/rexp/**，则返回真； **x!~ /rexp/** 如果 **x** 不匹配 **/rexp/**，则返回真。  **awk '$1 > 20 {print $0}' test.in**  **awk '$2 ~ /^6/ {print $0}' test.in** |
| ②：使用布尔 ( 比较 ) 表达式，表达式的值为真时执行相应的操作 (**actions**)  表达式中可以使用变量 ( 如字段变量 $1,$2 等 ) 和 /rexp/  表达式中的运算符有：  关系运算符:  **< > <= >= == !=**  匹配运算符:  **~ !~   x ~ /rexp/** 如果 **x** 匹配 **/rexp/**，则返回真； **x!~ /rexp/** 如果 **x** 不匹配 **/rexp/**，则返回真。  **awk '$1 > 20 {print $0}' test.in**  **awk '$2 ~ /^6/ {print $0}' test.in** |
| ③：复合表达式：**&&** ( 逻辑与 )、**||** ( 逻辑或 )、**!** ( 逻辑非 )  **expr1 && expr2** 两个表达式的值都为真时，返回真  **expr1 || expr2** 两个表达式中有一个的值为真时，返回真  **!expr** 表达式的值为假时，返回真  **awk '($1<20)&&($2~/^6/){print $0}' test.in**  **awk '($1<20)||($2~/^6/){print $0}' test.in**  **awk '!($2~/^6/){print $0}' test.in**  **awk '/^#/ && /#$/{ print }' test.in**  **注：表达式中有比较运算时，一般用圆括号括起来** |

#### 字段分隔符、重定向和管道

|  |
| --- |
| 字段分隔符：  **awk** 中的字段分隔符可以用 **-F** 选项指定，缺省是空格。  **awk '{print $1}' test.in**  **awk –F: '{print $1}' test.in**  **awk -F'[ :]' '{print $1}' test.in** |
| 重定向与管道：  **awk '{print $1, $2 > "output"}' test.in**  **awk 'BEGIN{"cal" | getline a; print a}'** |

#### 更多awk

|  |
| --- |
| awk不仅是一个命令，它更是一个编程语言 |
| 变量  内部变量  awk ' {print NR,$0} ' #给文件加上行号  自定变量 |
| 函数  内置函数  自定义函数  awk '{ print sum($1,$2) } function sum(x,y) { s=x+y ; return s }' grade.txt |
| 数组  awk 'BEGIN { print split("123#456",arr,"#") ; for (i in arr) { print arr[i] } }' |