awk以及shell编程简介

awk：输出

print expr1, expr2,...,exprn或print(expr1, expr2,...)

- 输出多个参数的值，值之间缺省以空格隔开。print等价于print $0
- OFS： print输出时参数之间的分隔符

printf format, expr-list或printf(format, expr-list)

- 类似于C语言的printf，第一个参数为格式化字符串，后面为要格式化的多个参数

- print/printf都允许输出到文件中，后面附加 > FILE，表示写入到FILE。如果第一次写，而FILE存在，则内容被覆盖。后面附加>> FILE，表示附加到文件中

```
$ awk 'BEGIN {print 1,2,3}'
1 2 3
$ awk 'BEGIN {OFS=":"; print(1,2,3)}'
1:2:3
$ ls /etc/ | awk '
BEGIN { print "List of config files:"}
/\.conf$/ {
    total += 1
    printf("%-3d %s\n", total, $0)
}
END { print "Total", total, "Files"}
'
```

awk：输入

- awk根据FS将当前记录分解为各个字段
- FS缺省为单个空格，实际上多个连续的空格类字符看成一个分隔符，且首尾的空格也会去除
- 可通过awk -F sep 指定FS；-v FS=sep；也可在BEGIN部分更改FS为其他分隔符
- FS是一个正则表达式
- $i = value: 修改字段或者增加字段，相应的$0也会改变

```
$ echo -e '  1 \t 2\t3 ' | awk -v OFS=: '{print $1,$2,$3}'
1:2:3
$ ps ax | grep [s]sh | sed -E  's/^\s+//' | cut -d' ' -f 1
$ ps ax | grep [s]sh | awk '{print $1}'
$ awk -F :  '{print $1,$6}' /etc/passwd
root /root
...
$ awk 'BEGIN { FS=":"} {print $1,$6}' /etc/passwd
...
$ echo '1,,,,,, 2;;;;' |awk 'BEGIN {FS="[,;]+";OFS=":"}
{print $1,$2,$3 ":"}'
1: 2::
$ echo 1 2 3 4 |awk '{$1 = -1; print}'
-1 2 3 4
$ echo 1 2 3 4 |awk '{OFS=","; $6 = -1; print}'
1,2,3,4,,-1
```

```
$ ps ax | grep [s]sh
  894 ?        Ss     0:00 /usr/sbin/sshd -D
 1513 ?        Ss     0:00 sshd: dlmao [priv]
 1638 ?        S      0:04 sshd: dlmao@pts/0
```

awk: 数组

- awk支持一维数组，与我们传统的数组不一样，不是按照顺序存储的
- awk支持的数组是关联数组，类似于python语言的字典，C语言中的哈希表，下标为字符串类型，整数下标自动转换为字符串；值为数值或字符串。
- 不需要声明某个变量是数组，也不需要声明其长度，也不需要声明数组的元素类型
- a[index] = value 如果a不存在，首先创建一个数组；添加元素对：index/value
- a[index] 返回index对应的值，如果index不存在，则返回0或空字符串，同时加入到数组a中
- delete a[index] 删除数组a中index对应的元素
- 如何遍历数组？ 注意数组下标的顺序是不确定的，由系统决定顺序

  for (idx in a) statement

```
$ seq 4 | awk '{ a[NR] = $0 }
END {
  for (i = NR; i > 0; i--)
    print(a[i])
}'
4
3
2
1
```

```
$ awk 'BEGIN {
  a[1] = 1
  a[2] = 4
  a[3] = 8
  delete a[3]
  a["test"] = 1
  for (i in a)
    print(i, a[i])
}'
test 1
1 1
2 4
```

`for (init;condition;update) statement`
循环结构：

- 首先执行`init`
- 重复直到`condition`不满足为止
  - 如果满足执行`statement`部分
  - 执行`update`
- `statement`部分有多条语句时，用`{ }`括起来

awk：程序结构

- 许多内置函数：数学函数(sqrt/exp/sin/cos/rand/int等)、字符串处理(tolower/toupper/length/index/substr/gsub/split等)
- 可以定义函数： function function_name(args) statement
  - return语句：从函数中返回
- 条件语句：支持单分支和双分支结构

```
if (expr) statement
if (expr) statement else statement
```

- 循环语句：while/for循环，也支持continue/break语句

```
while (expr) statement
do statement while (expr)
```

- exit语句：exit expr 如果在END部分中表示退出awk，否则不再继续执行所在的action部分，跳到END部分执行

```
$ awk 'BEGIN {
  t = int(rand() * 10)
  if (t > 5) print(t, ">5")
  if (t % 2 == 0)
    print(t, "is even number")
  else
    print(t, "is odd number")
}'
```

```
$ awk 'BEGIN {
  i = 1
  while (i < 6) {
    print i
    i += 1
  }
}'
```

```
$ cat marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
```

```
$ awk 'NR > 1 {
  if ($3 >= 60)
    passed += 1
  else
    failed += 1
  total += $3
}
END {
  printf("passed:%d, failed:%d\n", passed, failed)
  printf("average:%.2f\n", total / (NR -1))
}
' marks2.txt
passed:3, failed:1
average:68.50
```

shell编程简介: 算术运算

bash用于分割单词的元字符如下:
< > | & ; ( ) space tab newline

- 所有常量都是字符串，引号/双引号为引用，表示引起来的内容是一个整体
- 所有变量都是字符串类型，$var访问变量var的值
- 整数算术运算:
  - expr命令:
    - 运算数/运算符都作为expr的参数，如expr 4 + 5
    - 还支持字符串运算(长度/子串/下标)和正则表达式匹配
  - 内置命令let: 可有多个参数，每个参数是一个算术表达式，如let t=4+5
  - ((expression)): 类似于let，但运算数/运算符前后可有空格，建议采用
    - 可以通过名字访问变量的值，比如t+4，当然也可以$t + 4
    - 类似于C语言，支持算术运算(幂运算符为**)、比较运算、逻辑运算
    - 支持赋值和复合赋值运算，如t+=4
    - 程序语言中，非0表示True，0表示False，命令状态码为0表示成功，非0表示失败
    - 表达式计算的结果为0时，状态码为1；结果为非0时，状态码为0
- 算术运算扩展$((expression)): 替代以expression运算后的结果
- 浮点运算一般使用bc命令

```
$ t=4+5
$ echo $t
4+5
$ ((t = 4 + 5))
$ echo $t
9
$ (( t >= 10))
$ echo $?
1
$ echo $((4 + 5))
9
```

shell编程简介: 分支结构

if COMMANDS; then COMMANDS; [elif COMMANDS; then COMMANDS;]... [else COMMANDS;] fi

- 如果命令COMMANDS的状态码为0(成功)，则执行相应的语句
- 注意分号也可用换行符代替

```
$ t=8; if (( t > 0)); then printf "$t > 0\n"; fi
8 > 0
```

```
t=8
if (( t > 0)); then
  printf "$t > 0\n"
fi

t=8
if (( t > 0))
then
  printf "$t > 0\n"
fi
```

```
LC_ALL=C
if date | grep -q Fri; then
    echo "It's Friday!"
fi

if date | grep -q Fri; then
    echo "It's Friday!"
else
    echo "It is not Friday!"
fi
```

```
read -p "Please input score: "
if [[ $REPLY -ge 90 ]]; then
    echo "RANK A"
elif [[ $REPLY -ge 85 ]]; then
    echo "RANK A+"
elif [[ $REPLY -lt 60 ]]; then
    echo "RANK F"
fi
```

shell编程简介: 条件判断

- 条件表达式判断命令:
  - test命令： test expr
    - 文件状态判断
    - 整数比较
    - 逻辑运算
    - 字符串比较
  - [expr]， 等价于 test expr
  - [[expr]]: 建议采用的方式
    - 不进行路径名扩展和单词分割
    - test命令中支持的表达式
    - 支持逻辑运算： && || !
    - 文件通配符匹配： ==
    - 正则表达式匹配： =~

```
-e file    file存在
-r file    file存在且可读
-w file    file存在且可写
-x file    file存在且可执行
-f file    file是普通文件
-d file    file是目录
-s file    file存在且长度大于0
```

```
int1 -eq int2    int1等于int2
int1 -ne int2    int1不等于int2
int1 -gt int2    int1大于int2
-gt greater than   -lt   less than
-ge greater/equal -le   less/equal
```

test	[[ ]]	描述
-a	&&	逻辑and
-o	||	逻辑or
!	!	逻辑not
( )		改变运算顺序

```
-z string          空字符串(长度为0)
-n string          非空字符串(长度>0)
string             非空字符串(长度>0)
str1 = str2        字符串相等
str1 == str2
str1 != str2       字符串不相等
str1 > str2        str1在str2之后
str1 < str2        str1在str2之前
```

```
str1 = str2        字符串相等
str == pattern     基于通配符扩展，str与pattern匹配
str =~ regexp      str与扩展正则表达式regexp匹配
```

```
FILE=tmp.py
if [[ "$FILE" == *.py ]]; then
    echo "$FILE is a python program."
fi
```

```
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    echo "$INT is integer"
fi
```

shell编程简介: 循环结构

while COMMANDS; do COMMANDS; done

- 当命令执行成功时，一直执行循环体
- 循环体内支持 break/continue语句

until COMMANDS; do COMMANDS; done

- 执行循环体，直到命令执行成功时，结束循环
- until循环中循环体至少执行一次

for ((exp1; exp2; exp3)); do COMMANDS; done

- 类似于C语言的for循环。首先执行exp1初始化；判断表达式exp2是否为真，如果为假，结束循环；如果为真，执行循环体，然后执行exp3，再次判断条件exp2看是否继续循环

for NAME [in WORDS ...] ; do COMMANDS; done

- 对于后面列表(用空格隔开的多个参数）中的每个元素，NAME设置为该元素，然后执行循环体
- in WORDS省略时，相当于 for NAME in "$@"

```
count=1
while [[ $count -le 5 ]]; do
    echo $count
    (( count += 1))
done
```

```
for ((i = 1; i <= 5; i += 1 )); do
    echo $i
done
```

```
for i in 1 2 3 4; do
    echo $i
done
for i in `seq 1 4`; do
    echo $i
done
for i in {1..4}; do
    echo $i
done

for i in /etc/p*.conf; do
    echo $i
done
```

shell编程简介: 函数和位置参数

function name { COMMANDS ; }

name () { COMMANDS ; } 建议的形式

- 调用函数: name arg1 arg2....
- 调用函数或调用shel脚本时的位置参数:
  - $0: 程序名
  - $FUNCNAME: 函数名
  - $1/$2...: 第1/2个位置参数
  - $#: 参数个数，不包括$0
  - 所有参数组成的列表: $*或$@
  - 建议"$@"，等价于 "$1" "$2" ...
  - "$*"相当于 "$1 $2 $2"
- 函数体返回:
  - return语句: return [n]
  - n为状态码，如果没有，表示return $?, 即最后一个命令的状态码
  - 调用者通过echo $?得到状态码

```
$ cat func
args() {
    echo "\$0=[$0]"
    echo "\$#=[$#]"
    echo "\$1=[$1]"
    echo "\$2=[$2]"
    echo "\$3=[$3]"
    echo "\$4=[$4]"
    echo "\$*=[$*]"
    echo "\$@=[$@]"
    for i in $@; do
        echo $i
    done
    for i in "$*"; do
        echo $i
    done
    echo
    for i in "$@"; do
        echo $i
    done
}
args 1 a "b c"
```

```
$0=[func]
$#=[3]
$1=[1]
$2=[a]
$3=[b c]
$4=[]
$*=[1 a b c]
$@=[1 a b c]
1
a
b
c

1 a b c

1
a
b c
```

shell编程简介: 重定向

- if/while等语句都支持重定向
- HERE文本: 标准输入不是来自于键盘或某个文件，而是来自命令行的多行文本，常用在shell脚本中。 EOF可为任意不在文本中出现的字符串

<< EOF

多行,直到某行的内容为EOF时结束

EOF

<< "EOF" : 如果EOF用引号包含，则表示多行文本中不支持参数和变量扩展

<<- EOF(表示忽略文本块每行最前面的制表符，注意不是空格)

- HERE字符串: <<< WORD 表示输入来自于后面的单词WORD，如果WORD有空格，用引号包含

```
count=1
IFS=:
while read user shadow uid t; do
    echo $count $user $uid
    (( count += 1 ))
done < /etc/passwd  >/dev/tty
```

```
$ cat <<EOF > log.txt
This is a test log file
blah...
EOF
$ cat log.txt
This is a test log file
blah...
$ cat << 'DONE'
echo $LANG
show locale
DONE
echo $LANG
show locale
$ cat <<< "nice to meet you"
nice to meet you
```

shell编程简介: 其他程序结构

case WORD in [PATTERN [| PATTERN]...) COMMANDS ;;]... esac

- 类似于C语言的switch语句，多分支结构

select NAME [in WORDS ... ;] do COMMANDS; done

- 循环结构，一直重复：显示各个选项供用户选择，然后执行循环体

shift [n]

- 位置参数左移移位，用于逐个访问位置参数

${parameter:-word} ${parameter#pattern} ${parameter%pattern}

- 参数扩展还支持更多的格式，比如缺省值、截取字符串等

trap [-lp] [[arg] signal_spec ...]

- 支持信号处理，在收到信号时执行相应的命令
- shell也支持数组的访问