awk实用程序

此PPT介绍大家一个强大的shell工具，可以帮我们实现很多复杂的处理，是每一个Linux爱好者必须要精通的利器

指导老师：杨坤 QQ: 12238747
蓝狐教育系统培训中心 0731-84125710

awk简介

- awk是3个姓氏的首字母，代表该语言的3个作者
- awk的版本有很多，包括: 旧版awk，新版awk(nawk), GNU awk(gawk)等
- awk程序有awk命令、括在引号或写在文件中的指令以及输入文件这几个部分组成

从文件输入

- 本书是基于Linux版本，之后的例子都采用 gawk命令，你也可以使用 awk
- 格式：
  - gawk '/匹配字符串/' 文件名
  - gawk '{处理动作}' 文件名
  - gawk '/匹配字符串/ {处理动作}' 文件名

awk工作原理(一)

- 以下面的内容的names文件名举例按步骤解析awk的处理过程
  - vi ~/names

```
Tom Savage 100
Molly Lee 200
John Doe 300
```

:wq

使用下面awk命令处理

```
gawk '{ print $1 $3 }'  ~/names
```

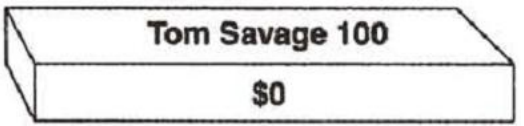
从命令输入

- awk还可以处理通过管道接收到的Linux命令的结果，shell程序通常使用awk做深处理
- 格式：
  - 命令 | gawk '/匹配字符串/'
  - 命令 | gawk '{处理动作}'
  - 命令 | gawk '/匹配字符串/ {处理动作}'

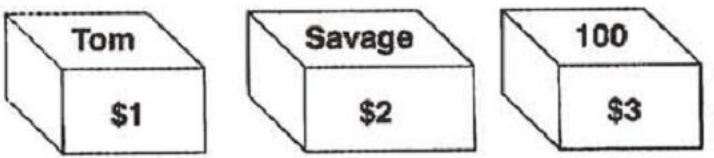
```
df | gawk '$4 > 200000'  #剩余空间大于200000的磁盘
```

awk工作原理(二)

- 第一步：awk对文件或管道的内容一次只处理一行，将获取到的这一行赋给内部变量 $0



- 第二步：这一行的内容按awk内部变量FS定义的分隔符，缺省为空格（包括tab制表符）分解成字段,每一段存储在从 $1 开始的变量中



awk工作原理(三)

- 第三步：awk中print命令打印字段
  - {print $1,$3}　#只取有用的第一段和第三段
  - 在打印时$1和$3之间由空格间隔。","逗号是一个映射到内部的输出字段分隔符（OFS），OFS变量缺省为空格，逗号在输出时被空格替换

```
Tom 100
Molly 200
John 300
```

- 接下来，awk处理下一行数据，直到所有的行处理完

格式化输出 print 函数

- awk命令操作处理部分是放在 "{}"(括号)中
- print函数将变量和字符夹杂着输出，如同linux中的echo命令

```
shell> date
Wed Feb 24 10:22:28 CST 2010
shell>   date | gawk '{ print "Month: "$2 "\nYear:", $6 }'
Month: Feb
Year: 2010
```

注意上面的例子，一种是直接在Month后连接$2,另一种是在Year和$6之间使用了逗号，都由OFS决定

OFMT变量

- 在OFMT中定义数字的格式
- 默认为"%.6gd"，只会打印小数点后6位
- shell> df

```
Filesystem      1K-blocks     Used      Available Use% Mounted on
/dev/hda1        19599892   4212460    14391808  23%    /
none            103652        0        103652    0%     /dev/shm
/dev/hdc         2191906   2191906        0       100%   /media/cdrom
```

- shell> df | grep -v 'Available'| gawk '$4 > 200000 { OFMT="%.2f"; print $1, $4/1024,"M" } '

  /dev/hda1 14054.50 M

printf函数转义字符

- printf与C语言中的printf雷同
  - 转义字符
  - %c 字符

    printf("The charcter is %c\n",x)
  - %s 字符串
  - %d 十进制整数

    printf("The boy is %d years old\n",50)
  - %f 浮点数

printf函数修饰符

- 打印时需要对齐，下面提供一些打印输出时所用到的修饰符

-(横杠)　左对齐

```
echo "Bluefox" | gawk '{ printf "|%-15s|\n",$1}'
    |Bluefox         |
echo "Bluefox" | gawk '{printf "|%15s|\n",$1}'
    |         Bluefox|
```

#(井号)　显示8进制时前面加0，显示16进制时加0x

+(加号)　显示正负值时的正+负-号

0（零）　用0对显示值填充空白处

文件中的awk命令

- 将awk写入一个文件中，更适合复杂的程序
- 使用 -f 选项指定awk的文件名
- awk一次读取一条记录，测试文件中的每一条命令这样循环
- vi awkfile

```
# !/bin/awk            #标识为awk程序
/^Mary/ { print "Hello Mary!" }
{print $1, $2 , $3}
```

- gawk -f awkfile employees_db_file

记录与字段

- 记录分隔符：默认行输入和输出的分隔符都是回车 ，保存在 RS 和 ORS 内部变量中
- 变量$0: awk每次一行取得整条记录，$0随之改变，同时内部变量 NF（字段的总数）也随之变化
- 变量NR: 每条记录的行号，处理完一行将会加1，所以全部处理完后可以理解成行数的总数

```
gawk '{print NR,": ->",$0}' /etc/passwd
```

字段分隔符

- FS内部变量：
  - 保存着输入字段的分隔符的值 (OFS则代表输出的分隔符)
  - 默认使用空格或制表符来分隔字段
  - 在BEGIN语句段中设置FS的值

```
gawk 'BEGIN {FS=":"}  $3 > 499 { print $1; count++ } END {
    print "Total normal users", count,"\n"}'  /etc/passwd
```

- 也可以在命令行中指定 -F 选项改变分隔符

```
gawk -F:  '{ print $1 }'  /etc/passwd
```

- 使用多个字符分隔符，写在括号中,下面的例子使用空格，冒号和制表符
  - `gawk -F'[ :\t]'  '{print $1, $5, $7 }'  /etc/passwd`

模式

- awk模式用来控制输入的文本行执行什么样的操作
- 模式为正则表达式
- 模式具有着隐式 if 语句
- 模式写在模式操作符两个 "//"中
  - gawk '/^root/' /etc/passwd

操作

- 格式

  模式 { 操作语句1; 操作语句2; ..... ;}

- 或者

  模式

  {

  操作语句1

  操作语句2

  .....

  }

正则表达式

- 很多地方都是用到正则表达式来匹配特定的信息

^	串首
$	串尾
.	匹配单个任意字符
*	匹配零个或多个前面的字符
+	匹配一个或多个前面的字符
?	匹配零个或一个前面的字符
[ABC]	匹配括号中给出的任一个字符
[A-Z]	匹配A到Z之间的任一个字符
A|B	匹配二选一，或者的意思，等同于[AB]
(AB)+	匹配一个或多个括号中的组合
*	星号本身，转义星号本身

匹配操作符

- 前面介绍了模式，也可以对特定的列使用模式匹配符"~"，与条件做比较
- 语法：
  - ~ / /　　或者　　!~ / /
    - gawk 'BEGIN { FS=":" } $7 ~ /bash/ { print $1 }' /etc/passwd
    - gawk ' $1 ~ /^[Bb]ill/' employees

POSIX字符类表达式

- [[:allnum:]] 字母和数字字符 [A-Za-z0-9]
- [[:alpha:]] 字母字符等同于[A-Za-z]
- [[:cntrl:]] 控制字符
- [[:digit:]] 数字字符 [0-9]
- [[:graph:]] 非空白字符(空格,控制字符等)
- [[:lower:]] [[:upper:]] 小写/大写字母
- [[:space:]] 所有的空白字符(换行符，空格，制表符)

```
gawk 'BEGIN { FS=":"} $1 ~ /^[[:digit:]]+$/ {print $0 }'
    /etc/inittab
```

awk脚本

- 如果有多条awk的模式或指令要处理，将它们都写在脚本中
- #为注释符
- 一行多条命令要用分号隔开
- 操作跟在模式之后的话，必须一行书写完（即左大括号在同一行）
- vi info

```
#!/bin/gawk
# The first awk script
/student/ { print "Student id is" , $2}
/vistor/ {print "vistor id is", $2}
```

- gawk -F: -f info /etc/passwd

比较表达式

- 用来对文本做比较，只有条件为真，才执行指定的动作
- < gawk '$3 < 500' /etc/passwd #系统帐户
- <= gawk '$3 <= 499' /etc/passwd #同上
- == gawk '$3 == 0' /etc/passwd #id为0，root帐户
- != gawk '$3 != 500' /etc/passwd #非id=500的帐户
- >= gawk '$3 >= 500' /etc/passwd #普通帐户
- > gawk '$3 > 499' /etc/passwd #同上
- ~ gawk '$1 ~ /^root/' /etc/passwd #root帐户
- !~ gawk '$1 !~ /^root/' /etc/passwd #非root帐户

条件表达式

- 格式：条件表达式1？表达式2：表达式3
- 与之等同的代码段如下

```
{
 if (条件表达式1成立)
      表达式2
else
      表达式3
}
awk '{ max=($1 > $2) ? $1 : $2 } ; print max}'  filename
awk '{ print ($7 > 4 ? "high " $7 :  "low "$7)}' datafile
```

算术运算

- 可以在模式中执行计算操作

```
awk '$3 / 1024 > 2000'  filename
```

+	加	10+2	=12
-	减	10-2	=8
*	乘	10*1024	=10240
/	除	10240/1024	=10
%	求模（求余数）	10%3	=1
^	次方	2^3	=8

逻辑操作符

- 逻辑操作符用来测试模式的真假

&&	逻辑与	1&&0	FALSE
||	逻辑或	1 || 0	TRUE
!	逻辑非	!0	TRUE

```
awk -F: '$3 >500 && $3 <= 550' /etc/passwd
awk -F:  '$3 ==100 || $3 > 50'    /etc/passwd
```

范围模式

- 范围模式提供了选择一段数据操作的可能
- 先匹配第一个模式，将作为开始部分
- 接着匹配第二个模式，作为结束
- 之间的这一段将选中做操作
- 模式与模式之间使用","逗号间隔

```
awk '/operator/,/nobody/'  /etc/passwd
```

验证数据的有效性

- 验证数据的有效性可以综合我们之前所学的判断方式
- 判断有效数据是否为七列，可以通过检测 NF 内部变量

```
awk 'NF ! = 7 {.....}' filename
```

- 判断是否为字母开头

```
awk '$1 ~ /^[a-zA-z]+/ {.....}' filename
```

- 判断数值是否大于某值

```
awk '$4 > 200 {.....}' filename
```

数值变量和字符串变量

- 字符串写下双引号中,比如"Hello world"
- 默认将一个字符串转化成数字时，将变成0
  - name = "Nancy"　　#name此时为字符串
  - x++　　#x是数字，初始值为1
  - number = 35　　#number是数字
- 强行转化
  - name + 0　　#name此时变成数字0
  - number " "　　#number此时为字符

用户自定义变量

- 变量名可以是 字母 数字 和 下划线 组成，但不能以数字开头。
- awk 将字符串变量初始化为空字符串 ""
- 数值变量初始化为 0
- 格式
  - 变量 = 表达式

```
gawk  '$1 ~ /Tom/ { wage = $2 * $3; print wage}' filename
```

BEGIN模式

- BEGIN模式后面跟了一个程序段
- 对输入文件进行任何处理之前，先执行BEGIN程序段
- 用来修改内部变量（OFS，RS，FS）等的 值
- 用来初始化用户变量和打印标题等

```
gawk 'BEGIN { FS=":"; OFS="\t";  ORS="\n\n" }  {
  print $1,$2,$3} '  filename
```

- 甚至无需文件名

```
gawk 'BEGIN { print "Helllo World" }'
```

END模式

- 不匹配任何输入行
- 在awk处理完输入行之后执行

```
gawk 'END { print "Total Users: " NR } '
/etc/passwd
```

```
  Total Users:  35
```

```
gawk '$0 ~ /^[^#]+<Directory/ { word_count++ }
END { print "Directory was found " word_count "
times" }'  /etc/httpd/httpd.conf
```

输出重定向

- 可以将 awk 的输出重定向到Linux文件中
- 目标文件必须用双引号括起来
- "> " 清空文件,写入awk信息，awk程序执行完毕后才关闭文件
- ">>" 追加内容

```
cat -n /etc/passwd
gawk 'NR > 10 && NR < 15 { print $0  >
   "/tmp/newfile" }' /etc/passwd
cat /tmp/newfile          #提取10到15行
```

输入重定向-读输入 getline

- 从标准输入，管道或文件中读取输入
- 读取每一行，重置 NF，NR 和 FNR 内部变量
- 有内容读取到，返回 1 （为真值）
- 读到 EOF（文件末尾） 返回 0 (为假值）
- 发生错误，返回 -1

```
gawk 'BEGIN { "date" | getline d; print d}'
gawk 'BEGIN { print "What is your name?"; getline name <
   "/dev/tty" } $1 ~ name { print "Found " name " on line ", NR " .
   " ; print "Nice to meet  you " name " @_@"}'  /etc/passwd
```

Bash向awk发送参数

- getline 函数也可以从 Bash 命令发送的管道中获取参数
- 语法：
  - getline 变量 < "-"　　# "-" 如同 tar 一样代表着管道过来的信息
- 示例：
  - echo "$(date +%F)" | gawk 'BEGIN{ getline DATE < "-" } { if($2 ~ DATE){ print }}'

管道

- 打开管道的前提是其他管道必须关闭，每次只能打开一个管道，这和 Linux 中一条命令无限次的使用管道不同
- 管道右边的命令必须写在双引号之间

```
gawk '/^[^#]+/ { print $1 | "ping -c 3 " $1 }'
    /etc/hosts
```

关闭文件和管道

- 要对awk程序中的某个文件或管道多次读写，得先关闭程序
- awk程序在处理文件时会保持打开状态直到脚本结束。
- END块中也受管道影响。
- 关闭文件或管道时的引号中的内容与打开文件或管道时的名称一致，甚至包括参数，空格
- 使用 close（ ）函数实现

```
gawk '/^[^#]+/ { print $1 | "ping -c 3 " $1 } END {
    close("ping -c 3 ")}'  /etc/hosts
```

system函数

- system（ ）函数以Linux命令作为参数
- 执行Linux命令之后将Linux命令 $? 退出值返回给awk
- 注意：函数中Linux命令必须用双引号

```
awk ' $1 ~ /^[^#]+/
{   if ( system("ping -c 2 " $1) == 0 )      {
        printf ("%s is online",$1);
    }else{
        printf ("%s is down",$1);
    }
}'  /etc/hosts
```

if 语句

- awk就如C语言，有着变量，条件判断和循环
- if 语句是条件判断的关键字,默认为隐藏，
- 格式

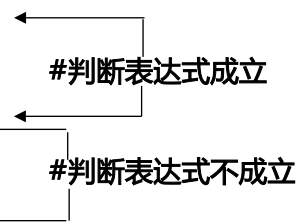
```
if (判断的表达式)
{
    语句1；语句2；.....   #判断表达式（为真）成立而执行的代码段
}
```

```
gawk -F: '{if ( $3 >= 500) print  $1 " is a normal user\n"}'
  /etc/passwd
gawk -F: '{ if ($3 >=500) {count++;} } END { print "Total
  normal user: " count }' /etc/passwd
```

if/else语句

- if/else语句实现了真与假的两重处理
- 判断表达式为1(真)，与前面if 语句相同
- 为0 (假)则执行else后面的代码段
- 格式:

```
if (判断表达式) {
        语句1；语句2；.....        #判断表达式成立
} else {
        语句10,语句11;.....        #判断表达式不成立
}
```



- gawk -F: '{ if($3 >=500) { print $1 " is a normal user\n" } else { print $1 " is a system user\n" }' /etc/passwd
- gawk -F: '{ if($3 < 500) { print $1 " is a system user\n" } else { print $1 "is a normal user\n" }' /etc/passwd

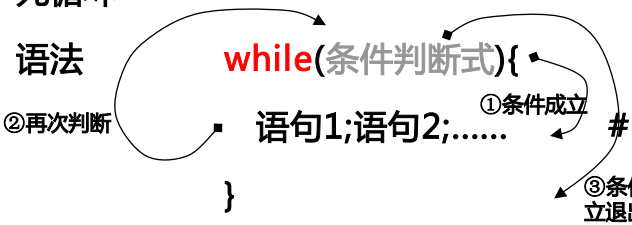
if/else和else if语句

- 如果在计算多重判断的时候,我们还需要对if/else语句做扩充,在其后再加上 if else,做下一个判断
- 语法

```
if (判断表达式1) {
    语句1; 语句2;.....          #表达式1成立
}else if (判断表达式2) {
    语法10;语法11;.....         #表达式2成立
}else if (判断表达式N...) {
    语法N0;语法N1;.....         #表达式N成立
}else {
    语法Y1;语法Y2;....          #以上都不成立
}
```

while循环

- 首先给初始一个变量,接着在 while 判断表达式中测试该变量,为0(假)退出 while 循环;
- 注意:代码段中要在一些情况下修改初始的变量值,否则是一个消耗CPU的死循环
- 语法




```
gawk -F: '{ print NR," : User info:\n============";  i=1;
while(i<=NF) { print $i; i++ } ; print "\n\n"}' /etc/passwd
```

for循环

- 有着三个表达式,第一个为初始化变量,第二个做测试,第三个用来改变初始变量(如果缺少此部分,就得到代码段修改,否则是死循环)
- 语法　　for(初始表达式; 判读表达式; 更新表达式) {

　　语法1; 语法2; .....

　　}

```
gawk -F: '{ print NR," : User info:\n==============="; for ( i=0 ; i<NF; i++) { print $i } { print "\n\n"} } '  /etc/passwd
```

循环控制

- break 用来终止循环
- continue 语句 用来不做后续操作,绕过此次循环,继续下一循环

```
gawk -F: '{ i = 1;
        while(i++ < NF){
            if ($i ~ /operator/)
            {print NR; break;}
         }
} ' /etc/passwd
```

next语句

- 在循环处理时，使用 next 语句，将会使 awk 中跳过 next 段以后本次的处理，执行下一个循环
- 下面的例子将打印出除系统帐户以外的所有用户

```
cat /etc/passwd  | gawk -F:  '{ if ($3 < 500) {  next; } \
else { print $1 } }'
```

exit语句

- exit 用来终止 awk 对后续内容的处理
- exit 也可以返回具体的值，提供Bash做状态判断
- 要注意，exit退出不会绕过END块，换句话说END块总会执行，对于要向BASH返回值的处理，使用以下的方法
- 语法：
  - { exit (1) }

exit示例纠错

- 下面的例子检测是否存在用户，存在返回0，否则返回255
  - gawk -F: ' BEGIN { "read -p \"Input a username: \" USER; echo $USER" | getline USER }{ if ($1 ~ USER) { exit (0) } } END { exit(-1) }' /etc/passwd
  - 错误上例永远返回 -1
  - gawk -F: 'BEGIN{ "read -p \"Input a username: \" USER; echo $USER" | getline USER }{ if ($1 ~ USER) { EXIST=1;exit (0) } } END{ if (EXIST) { exit(0) } else{ exit(-1) } }' /etc/passwd

关联数组的下标

- awk中的数组下标可以是数字，也可以是字符串
- 数组和变量一样，需要的时候直接创建
- 数组的下标数值由0开始

```
vi employees
Tom         4424    05/19/2010      543354
Mary        5346    05/11/2008      28765
Bill 1683   09/23/2009      233103
```

- gawk '{ name[x++]=$1 }; END { for(i=0;i<NR;i++){print $i,name[$i] } }' employees

用字符串作为数组的下标

- 专门用于数组的 for 循环，遍历数组
- for(索引 in 数组)
- {
  - 语句
- }

处理命令行参数

- 从命令行获得参数，ARGC代表参数的总数，ARGV数组用来保存输入的参数

```
BEGIN{
        for ( i=0; i < ARGC; i++){
                printf("argv[%d] is %s\n", i, ARGV[i])
        }
        printf("The number of arguments, ARGC=%d\n", ARGC)
}
```

```
[root@localhost ~]# gawk -f argvs /etc/passwd
argv[0] is gawk
argv[1] is /etc/passwd
The number of arguments, ARGC=2
```

处理命令行参数二

- awk不会把 -f 以及后面的脚本认定为参数
- 之前的例子

  gawk -f argvs /etc/passwd "Operator Li" 56

  再观察结果

字符串sub和gsub函数

- sub和 gsub 函数，可以在条目中查找与给定的正则表达式匹配的字符串，并取代它
- sub 和 gsub的区别是，前者只对匹配部分一次替换，后者为全部替换
- 语法：
  - sub(正则表达式，替换字符串） #默认为$0,整条记录
  - sub(正则表达式，替换字符串，目标字段）；#指定的字段

sub 函数示例

- 注意正则表达式的使用方法
- gawk '{ sub(/172\.168\.0\./, "192.168.0."); print}' /etc/sysconfig/iptables
- gawk '{ sub(/Mac/,"MacIntosh",$1) ; print }' filename

- gawk '{ gsub(/[Tt]om/, "Thomas" , $1); print }' datafile

字符串长度 length 函数

- length函数取回字符串的字符数量
- 格式
  - length（字符串）
  - length #不带参数，返回记录中的字符个数

  - gawk '{ print length($1) }' filename

字符 substr函数

- substr函数返回从字符串指定位置开始的一个子字符串。
- 如果指定了子字符串的长度，返回字符串的对应的部分
- 语法：
  - substr(字符串，起始位置）
  - substr(字符串，起始位置，子字符串长度）

  - gawk ' { print substr($1,7,length) }' filename

字符 match函数

- match函数根据正则表达式返回其在字符串中出现的位置，未出现，返回0
- match函数中变量
  - RSTART　　子字符串出现的起始位置
  - RLENGTH　　子字符串的长度
  - 而这些变量之后可以提供给substr来提取子字符串

```
gawk 'BEGIN { line="Good ole USA"};\
END {match(line,/[A-Z]+$/); \
print substr(line,RSTART,RLENGTH) }' filename
```

字符 split 函数

- split函数用来将一个字符串拆分成一个数组
- 语法：#注意Split函数切割的数组下标从1开始，0永远为空
  - split (字符串, 保存数据的数组，字段分隔符)
  - split (字符串, 保存数据的数组) #使用FS默认值

vi database

2010-04-22	car	10	1000
2010-05-10	car	7	700
2010-05-13	dog	8	80
2010-06-11	bike	1	100

```
gawk '{ split($1,date,"-"); if(date[2] == 05 ) {
   count+=$4 }} END { print count}' datebase
```

整数 int 函数

- int函数去掉小数点后面的数字，仅仅留下整数部分
- 它不会做四舍五入操作
  - gawk 'END {print 31/3}' filename
  - gawk 'END {print int(31/3)}' filename

生成随机数

- rand函数生成一个大于或等于0，小于1的浮点数
  - gawk '{print rand() }' filename
- 当多次执行上面的脚本，每次都生成相同的数字
- srand函数，以当前时刻为rand()生成一个种子
- srand(x)把x设置为种子，通常，程序应该在运行过程中不断的改变x的值
- gawk 'BEGIN{srand()}; {print rand()} ' filename

systime时间函数

strftime格式化时间函数

用户自定义函数

指导老师：杨坤 QQ: 12238747
蓝狐教育系统培训中心 0731-84125710

实验一

- 分析下面数据中，打印出每个销售员5月销售的总额

vi sales

Tom	2010-04-09	car	6	6000
Mary	2010-05-07	car	1	1000
Tom	2010-05-20	bike	15	1500
Mary	2010-05-22	car	2	2000
Tom	2010-06-17	car	1	1000

实验二

- 以下的数据需要转成SQL语句，方便插入到数据库中，注意年月日的格式

vi sales

Tom	04/09/2010	car	6	6000
Mary	05/07/2010	car	1	1000
Tom	05/20/2010	bike	15	1500
Mary	05/22/2010	car	2	2000

SQL格式

```
insert into sales value('Tom',2010-04-09，
'car',6,6000)
```

谢谢!

任何问题请 email：yangwawa0323@163.com