awk进阶

我们所学的centos7，awk，也就是gawk

```
[root@node02 tmp]# ll /usr/bin/awk
lrwxrwxrwx. 1 root root 4 Jun  4 19:05 /usr/bin/awk -> gawk
```

awk能够对原始数据进行格式化展示，适合处理各种数据格式化任务。

使用变量

awk该编程语言一特性就是使用变量存取值，支持两种类型变量

- 内置变量
- 自定义变量

awk的一些内置变量，存放处理数据文件中的数据字段和记录的信息。

内置变量

字段和记录分隔符

已知awk使用 `$1 $2 $3` 的形式记录字段的位置，以此类推，awk默认分隔符是 `空格` 。

以及可以使用 `-F` 选项修改分隔符， `NR` 内置变量指定行号。

```
[root@node02 tmp]# awk -F ":" 'NR==1,NR==5{print $1}' /etc/passwd
root
bin
daemon
adm
lp
```

awk数据字段和记录变量

FIELDWIDTHS	由空格分隔的一列数字，定义了每个数据字段确切宽度
FS	输入字段分隔符
RS	输入记录分隔符
OFS	输出字段分隔符
ORS	输出记录分隔符

案例

> awk逐行处理文本的时候，以输入分割符为准，把文本切成多个片段，默认符号是空格

> 当我们处理特殊文件，没有空格的时候，可以自由指定分隔符特点
>
> FS变量就是控制分隔符的作用

```
[root@node02 tmp]# cat num.txt
data11,data12,data13,data14,data15
data21,data22,data23,data24,data25
data31,data32,data33,data34,data35

[root@node02 tmp]# awk 'BEGIN{FS=","}{print $1,$2,$3}' num.txt
data11 data12 data13
data21 data22 data23
data31 data32 data33
```

还可以通过修改OFS变量，控制输出时的分隔符。

```
[root@node02 tmp]# gawk 'BEGIN{FS=",";OFS="|"}{print $1,$2,$3}' num.txt
data11|data12|data13
data21|data22|data23
data31|data32|data33

[root@node02 tmp]# gawk 'BEGIN{FS=",";OFS=" | "}{print $1,$2,$3}' num.txt
data11 | data12 | data13
data21 | data22 | data23
data31 | data32 | data33
```


数据变量

除了字段和记录分隔符变量，awk还提供了些内置变量用于了解数据的变化。

变　量	描　述
ARGC	当前命令行参数个数
ARGIND	当前文件在ARGV中的位置
ARGV	包含命令行参数的数组
CONVFMT	数字的转换格式（参见printf语句），默认值为%.6 g
ENVIRON	当前shell环境变量及其值组成的关联数组
ERRNO	当读取或关闭输入文件发生错误时的系统错误号
FILENAME	用作gawk输入数据的数据文件的文件名
FNR	当前数据文件中的数据行数
IGNORECASE	设成非零值时，忽略gawk命令中出现的字符串的字符大小写
NF	数据文件中的字段总数
NR	已处理的输入记录数
OFMT	数字的输出格式，默认值为%.6 g
RLENGTH	由match函数所匹配的子字符串的长度
RSTART	由match函数所匹配的子字符串的起始位置

ARGC和ARGV变量允许awk从shell中获取命令行参数的总数，但是awk不会把脚本文件当作参数的一部分

ARGC变量表示命令行上的参数，包括awk命令和文件名

```
[root@node02 tmp]# awk 'BEGIN{print ARGC}'
1
[root@node02 tmp]#
[root@node02 tmp]# awk 'BEGIN{print ARGC}' data.txt
2
```

ARGV数组值从索引0开始，表示awk本身，索引1表示第一个命令行参数

```
[root@node02 tmp]# awk 'BEGIN{print ARGV[0]}' data.txt
awk
[root@node02 tmp]# awk 'BEGIN{print ARGV[0],ARGV[1]}' data.txt
awk data.txt
[root@node02 tmp]# awk 'BEGIN{print ARGV[0],ARGV[1],ARGV[2]}' data.txt
awk data.txt
[root@node02 tmp]# awk 'BEGIN{print ARGV[0],ARGV[1],ARGV[2]}' data.txt xxxx
awk data.txt xxxx
```

awk内置变量的引用不用加美元符。

ENVIRON变量

该变量用关联数组提取shell环境变量，注意点：关联数组用 文本字符串 作为数组的索引值，而不是数值。

> 在 **计算机科学** 中，**关联数组**（英语：**Associative Array**），又称**映射**（**Map**）、**字典**（**Dictionary**）是一个抽象的 **数据结构**，它包含着类似于（键，值）的有序对。一个关联数组中的有序对可以重复（如C++中的multimap）也可以不重复（如C++中的map）。

数组索引中的key是shell的环境变量名，值是shell环境变量的值。

```
[root@node02 tmp]# awk 'BEGIN{print ENVIRON["HOME"],ENVIRON["PATH"]}'
/root /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
```

awk跟踪数据字段和记录时，变量`FNR,NF和NR`用起来就很方便了，比如你不知道awk到底分隔了多少个数据字段，可以根据NF变量获取最后一个数据字段。

```
FNR FNR：各文件分别计数的行号
NF  NF：number of Field，当前行的字段的个数(即当前行被分割成了几列)，字段数量
NR  NR：行号，当前处理的文本行的行号。
```

案例

```
[root@chaogelinux ~]# awk 'BEGIN{FS=":";OFS=" - "}{print $1,$NF}' /etc/passwd | head -3
root - /bin/bash
bin - /sbin/nologin
daemon - /sbin/nologin
```

NF变量就记录了字段的数量，因此`$NF`也就是打印最后一个字段。

###NR和FNR变量

> `FNR`和`NR`变量类似，FNR变量含有当前数据文件中已经被处理过的记录数量
>
> NR变量含有已处理过的记录总数。

看下案例差别

```
[root@chaogelinux ~]# awk 'BEGIN{FS=":"}{print $1,"FNR="FNR}' /etc/passwd | head -5
root FNR=1
bin FNR=2
daemon FNR=3
adm FNR=4
lp FNR=5
```

可以看出，`FNR`变量是记录处理的记录数量，也就是行数。

> 那NR和FNR的区别在哪?

```
[root@chaogelinux ~]# cat /tmp/pwd.txt
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin



[root@chaogelinux ~]# awk 'BEGIN{FS=":"}{print $1,"FNR="FNR,"NR="NR}END{print "There
ware",NR,"records processed"}' /tmp/pwd.txt /tmp/pwd.txt
root FNR=1 NR=1
```

```
bin FNR=2 NR=2
daemon FNR=3 NR=3
adm FNR=4 NR=4
lp FNR=5 NR=5
root FNR=1 NR=6
bin FNR=2 NR=7
daemon FNR=3 NR=8
adm FNR=4 NR=9
lp FNR=5 NR=10
There ware 10 records processed
```

我们会发现，FNR变量的值在awk处理第二个文件数据的时候被重置，而NR变量则在处理第二个数据文件时继续统计。

> 也就是我们利用AWK处理多个文件的时候，NR和NFR的区别就出来了

自定义变量

> shell脚本与awk变量

awk允许自定义变量在程序中使用，awk自定义的变量可以是任意数目的字母，数字，下划线，不得已数字开头，而且区分大小写。

例如

```
[root@chaogelinux ~]# awk 'BEGIN{testing="Hello chaoge.";print testing}'
Hello chaoge.

[root@chaogelinux ~]# awk 'BEGIN{v1="超哥nb";print v1;v1="超哥不错哦";print v1}'
超哥nb
超哥不错哦
```

数值计算

```
[root@chaogelinux ~]# awk 'BEGIN{x=4;x=x*2+3;print x}'
11
```

> 命令行与变量赋值，花式用法
>
> 使用awk命令可以给脚本中的变量赋值
>
> 该作用可以不改变脚本的情况下，改变脚本的作用。

```
[root@chaogelinux ~]# cat data
data11,data12,data13,data14,data15
data21,data22,data23,data24,data25
data31,data32,data33,data34,data35
```

```
[root@chaogelinux ~]#
[root@chaogelinux ~]#
[root@chaogelinux ~]#
[root@chaogelinux ~]#
[root@chaogelinux ~]# awk -f script1 n=2 data
data12
data22
data32
[root@chaogelinux ~]#
[root@chaogelinux ~]# cat script1
BEGIN{FS=","}
{print $n}

[root@chaogelinux ~]# awk -f script1 n=3 data
data13
data23
data33
```

使用命令行参数定义变量会有一个问题，设置了变量之后，这个值在代码的BEGIN部分不可用。例如

```
[root@chaogelinux ~]# cat script2
BEGIN{print "The starting value is",n;FS=","}
{print $n}
[root@chaogelinux ~]#
[root@chaogelinux ~]#
[root@chaogelinux ~]# awk -f script2 n=3 data
The starting value is
data13
data23
data33
```


发现这里只是打印了第三列的值，但是明没有在BEGIN里输出n的值

这里可以用-v选项解决，允许在awk的BEGIN开始之前设定变量。

```
[root@chaogelinux ~]# awk -v n=3 -f script2 data
The starting value is 3
data13
data23
data33
```

处理数组

为了能够在单个变量中，存储多个值，许多编程语言都提供了数组，awk也支持`关联数组`功能，也就是可以理解为是`字典`的作用。

```
例如

"name":"chaoge"
"age":18
```

关联数组的索引可以是任意文本字符串，每一个字符串都可以对应一个数值。

> 定义数组变量
>
> 语法
>
> var[index]=element
>
> var是变量名字，index是索引，element是值

案例

```
[root@chaogelinux ~]# awk 'BEGIN{student["name"]="超哥";print student["name"]}'
超哥
```

关联数组计算

```
[root@chaogelinux ~]# awk 'BEGIN{num[1]=6;num[2]=7;sum=num[1]+num[2];print sum}'
13
```

遍历数组变量

关联数组的问题是必须要知道索引是什么，否则无法取值。

可以利用for循环遍历出所有的索引。

```
for (var in array)
{
  语句
}
```


例如

```
[root@chaogelinux ~]# awk 'BEGIN{
> var["a"]=1
> var["b"]=2
> var["d"]=3
> var["h"]=4
> for (s in var)
> {print "Index: ",s," - Value:",var[s]}}'
Index:  h  - Value: 4
Index:  a  - Value: 1
Index:  b  - Value: 2
Index:  d  - Value: 3
```

注意，索引值的返回是没有顺序的，但是对应的值是唯一的。

###删除数组变量

```
语法
delete array[index]
```

一旦删除了索引，就无法用用它提取元素了。

```
[root@chaogelinux ~]# awk 'BEGIN{
var["a"]=1
var["b"]=2
var["d"]=3
var["h"]=4
for (s in var)
{print "Index: ",s," - Value:",var[s]};delete var["d"];print "----";for (s in var){print
"Index:",s,"Value:",var[s]}}'
```

使用模式

awk的模式，我们已知有BEGIN和END俩关键字来处理，数据流开始与结束两个模式。

正则表达式

正则表达式必须出现在要控制的脚本左花括号前面。

```
# 匹配含有data的记录
[root@chaogelinux ~]# awk 'BEGIN{FS=","}/data/{print $1}' data
data11
data21
data31
```


匹配操作符

(matching operator)匹配操作符是波浪线 ~ ，来看下如何用

匹配操作符（~） 用于对记录或字段的表达式进行匹配

```
$1 ~ /^data/

$1表示记录中的第一个数据字段，该正则会过滤出第一个字段以文本data开头的所有记录。
```

> 例如

```
[root@chaogelinux sed_awk]# tail -10 /etc/passwd
pyyu01:x:1005:1005::/home/pyyu01:/bin/bash
pyyu02:x:1006:1006::/home/pyyu02:/bin/bash
pyyu03:x:1007:1007::/home/pyyu03:/bin/bash
pyyu04:x:1008:1008::/home/pyyu04:/bin/bash
pyyu05:x:1009:1009::/home/pyyu05:/bin/bash
pyyu06:x:1010:1010::/home/pyyu06:/bin/bash
pyyu07:x:1011:1011::/home/pyyu07:/bin/bash
pyyu08:x:1012:1012::/home/pyyu08:/bin/bash
pyyu09:x:1013:1013::/home/pyyu09:/bin/bash
pyyu10:x:1014:1014::/home/pyyu10:/bin/bash

# 匹配出有关pyyu的行
```

```
# $1 表示针对第一列处理
#  ~ 按照后面的表达式进行匹配
# $NF 最后一个字段的值，NF获取有几列数据
[root@chaogelinux sed_awk]# awk -F ":" '$1 ~ /^pyyu/{print $1,$2,$3,$NF}' /etc/passwd
pyyu01 x 1005 /bin/bash
pyyu02 x 1006 /bin/bash
pyyu03 x 1007 /bin/bash
pyyu04 x 1008 /bin/bash
pyyu05 x 1009 /bin/bash
pyyu06 x 1010 /bin/bash
pyyu07 x 1011 /bin/bash
pyyu08 x 1012 /bin/bash
pyyu09 x 1013 /bin/bash
pyyu10 x 1014 /bin/bash
```

排除语法

搜索出除了以a,b,c开头的行

```
[root@chaogelinux ~]# head -5 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin

[root@chaogelinux ~]# head -5 /etc/passwd | awk -F : '$1 !~ /^[a-c]/{print $1,$NF}'
root /bin/bash
daemon /sbin/nologin
lp /sbin/nologin
```


数学表达式

除了正则，还可以用数学表达式，过滤如UID,GID寻找用户信息。

```
# 找出所有组ID为0的用户。
[root@chaogelinux ~]# awk -F :  '$4==0{print $1}' /etc/passwd
root
sync
shutdown
halt
operator
```

> 常见的数学表达式

```
 x == y:值x等于y。
 x <= y:值x小于等于y。
 x < y:值x小于y。
 x >= y:值x大于等于y。
 x > y:值x大于y。
```

> 例如找出uid大于1000的用户信息

```
[root@chaogelinux ~]# awk -F :  '$3>1000{print $0}' /etc/passwd
yu1:x:1001:1004::/home/yu1:/bin/bash
yu2:x:1002:1002::/home/yu2:/bin/bash
pyyu:x:1500:1500::/home/pyyu:/bin/bash
tom:x:1501:1500::/home/tom:/bin/bash
jerry:x:1502:1502::/var/jerry:/sbin/nologin
eva:x:1503:1503:The girl eva userinfo:/home/eva:/bin/bash
mjj:x:1504:1504::/home/mjj:/bin/bash
xiaomage:x:1505:1505::/home/xiaomage:/bin/bash
pyyuc:x:2000:2000::/home/pyyuc:/bin/bash
alex:x:2001:1500::/home/alex:/bin/bash
virtual_chao:x:2003:2003::/var/ftpdir:/sbin/nologin
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
cc:x:2004:2004::/home/cc:/bin/bash
only:x:2005:2005::/home/only:/bin/bash
test1:x:2006:2006::/home/test1:/bin/bash
chaoge:x:2007:2007::/home/chaoge:/bin/bash
chao:x:2008:2008::/home/chao:/bin/bash
susu:x:2009:2010::/home/susu:/bin/bash
```


结构化命令

awk也支持逻辑判断

if语句

awk支持标准的if语句

```
if (条件)
  语句
```

> 案例，如果uid在1000,2000之间就打印出用户信息

```
[root@chaogelinux ~]# awk -F: '{if ($3 > 1000 && $3 < 2000)print $0}' /etc/passwd
yu1:x:1001:1004::/home/yu1:/bin/bash
yu2:x:1002:1002::/home/yu2:/bin/bash
pyyu:x:1500:1500::/home/pyyu:/bin/bash
tom:x:1501:1500::/home/tom:/bin/bash
jerry:x:1502:1502::/var/jerry:/sbin/nologin
eva:x:1503:1503:The girl eva userinfo:/home/eva:/bin/bash
mjj:x:1504:1504::/home/mjj:/bin/bash
xiaomage:x:1505:1505::/home/xiaomage:/bin/bash
```

> 执行多条语句

```
[root@chaogelinux ~]# cat data
10
5
13
50
34
[root@chaogelinux ~]#
[root@chaogelinux ~]#
[root@chaogelinux ~]#
[root@chaogelinux ~]#
[root@chaogelinux ~]# awk '{
> if ($1>20)
> {
> x=$1*2
> print x
> }
> }' data
100
68
```


if else

awk也支持if语句不成立，执行其他语句。

```
[root@chaogelinux ~]# awk '{
> if ($1 > 20)
> { x= $1 *2;print x}
> else {x=$1/2;print x}
> }' data
5
2.5
6.5
100
68
```

单行写法

单行写法，要注意分号;和花括号{}的使用。

```
[root@chaogelinux ~]# awk '{if($1>20) print $1*2;else print $1/2}' data
5
2.5
6.5
100
68
```

while语句

awk也支持while的循环功能。

```
语法
while (条件)
{
  语句
}
```

while循环会遍历数据，且检查结束条件。

```
[root@chaogelinux ~]# cat data
130 120 135
160 113 140
145 170 215

# 该循环作用是相加三个列的值，求平均值
[root@chaogelinux ~]# awk '{
> total=0
> i=1
> while (i<4)
> {
> total+=$i
> i++
> }
> avg=total/3
> print "Average:",avg
> }' data
Average: 128.333
Average: 137.667
Average: 176.667
```


循环中断

awk支持在while循环里使用break和continue跳出循环。

```
[root@chaogelinux ~]# awk '{
> total=0
> i=1
> while (i<4)
```

```
> {
> total+=$i
> if (i==2)
> break
> i++
> }
> avg=total/2
> print "The average of the first tow data elements is:",avg
> }' data


The average of the first tow data elements is: 125
The average of the first tow data elements is: 136.5
The average of the first tow data elements is: 157.5
```

for循环

awk也支持for循环，且是c语言风格。

```
[root@chaogelinux ~]# awk '{
> total=0
> for (i=1;i<4;i++)
> {
> total+=$i
> }
> avg=total/3
> print "Average:",avg
> }' data
Average: 128.333
Average: 137.667
Average: 176.667
```


for循环的计数器比起while要好用了。

awk内置函数

awk内置的函数功能非常强大，可以进行常见的数学，字符串等运算。

数学函数

atan2(x, y)	x/y的反正切，x和y以弧度为单位
cos(x)	x的余弦，x以弧度为单位
exp(x)	x的指数函数
int(x)	x的整数部分，取靠近零一侧的值
log(x)	x的自然对数
rand( )	比0大比1小的随机浮点值
sin(x)	x的正弦，x以弧度为单位
sqrt(x)	x的平方根
srand(x)	为计算随机数指定一个种子值

> int()函数用法，得到整数，如同其他编程语言的floor函数
>
> floor函数，其功能是“向下取整”，或者说“向下舍入”、“向零取舍”，即取不大于x的最大整数，与“四舍五入”不同，下取整是直接取按照数轴上最接近要求值的左边值，即不大于要求值的最大的那个整数值。
>
> Int()函数会生成值和0之间最接近该值整数。
>
> 例如int()函数值为5.6返回5，值为-5.6时取-5

> rand()函数用于创建随机数，但是只会在0和1之间，要得到更大的数，就要放大返回值。

> srand() 随机数种子，计算机无法产生绝对的随机数，生成只能是伪随机数，也就是根据某规则生成的，因此可以加入随机数种子，根据系统时间的变化，产生不同的随机数。

具体用法，注意随机数种子必须写在BEGIN里，这是awk的机制，我们必须在awk开始计算前，加入随机种子。

> 获取随机数，且判断，尝试多少次后，得到小于10的数

```
[root@chaogelinux ~]# awk -F "\t" 'BEGIN{
srand();
}{
value=int(rand()*100)
print value
if(value<=10)
print "值: "value"\t次数: "NR
}'
```


> 随机数简单写法

```
[root@chaogelinux ~]# awk 'BEGIN{srand();print rand()}'
0.547909
[root@chaogelinux ~]# awk 'BEGIN{srand();print rand()}'
0.999358

[root@chaogelinux ~]# awk 'BEGIN{srand();print int(100*rand())}'
29
[root@chaogelinux ~]# awk 'BEGIN{srand();print int(100*rand())}'
4
```

字符串函数

asort(*s* [,*d*])	将数组*s*按数据元素值排序。索引值会被替换成表示新的排序顺序的连续数字。另外，如果指定了*d*，则排序后的数组会存储在数组*d*中
asorti(*s* [,*d*])	将数组*s*按索引值排序。生成的数组会将索引值作为数据元素值，用连续数字索引来表明排序顺序。另外如果指定了*d*，排序后的数组会存储在数组*d*中
gensub(*r*, *s*, *h* [, *t*])	查找变量$0或目标字符串*t*（如果提供了的话）来匹配正则表达式*r*。如果*h*是一个以*g*或*G*开头的字符串，就用*s*替换掉匹配的文本。如果*h*是一个数字，它表示要替换掉第*h*处*r*匹配的地方
gsub(*r*, *s* [,*t*])	查找变量$0或目标字符串*t*（如果提供了的话）来匹配正则表达式*r*。如果找到了，就全部替换成字符串*s*
index(*s*, *t*)	返回字符串*t*在字符串*s*中的索引值，如果没找到的话返回0
length([*s*])	返回字符串*s*的长度；如果没有指定的话，返回$0的长度
match(*s*, *r* [,*a*])	返回字符串*s*中正则表达式*r*出现位置的索引。如果指定了数组*a*，它会存储*s*中匹配正则表达式的那部分
split(*s*, *a* [,*r*])	将*s*用FS字符或正则表达式*r*（如果指定了的话）分开放到数组*a*中。返回字段的总数
sprintf(*format*, *variables*)	用提供的*format*和*variables*返回一个类似于printf输出的字符串
sub(*r*, *s* [,*t*])	在变量$0或目标字符串*t*中查找正则表达式r的匹配。如果找到了，就用字符串*s*替换掉第一处匹配
substr(*s*, *i* [,*n*])	返回*s*中从索引值*i*开始的*n*个字符组成的子字符串。如果未提供*n*，则返回*s*剩下的部分
tolower(*s*)	将*s*中的所有字符转换成小写
toupper(*s*)	将*s*中的所有字符转换成大写


```
[root@chaogelinux sed_awk]# cat data.txt
This is an apple.
This is a boy.

This is a gril.
[root@chaogelinux sed_awk]#
[root@chaogelinux sed_awk]# awk '{print toupper($0)}' data.txt
THIS IS AN APPLE.
THIS IS A BOY.

THIS IS A GRIL.
```

> 函数使用案例
>
> 大写转换，统计长度

```
[root@chaogelinux ~]# awk 'BEGIN{x="chaoge";print toupper(x);print length(x)}'
CHAOGE
6
```

全局替换函数

```
[root@chaogelinux ~]# awk '
BEGIN{
str="Hello,chaoge"
print "替换前的字符串：",str
gsub("chaoge","超哥",str)
print "替换后的字符串：",str
}'
替换前的字符串： Hello,chaoge
替换后的字符串： Hello,超哥
```

排序函数asort()，经过排序后的数组，索引会被重置

asort根据value进行排序

```
# 生成关联数组
[root@chaogelinux ~]# awk 'BEGIN{t["a"]=66;t["b"]=88;t["c"]=22;for(i in t){print i,t[i]}}'
a 66
b 88
c 22

# asort()排序，新数组
[root@chaogelinux ~]# awk 'BEGIN{t["a"]=66;t["b"]=88;t["c"]=22;asort(t,newt);for(i in newt)
{print i,newt[i]}}'
1 22
2 66
3 88
```


排序函数asorti()，排序的是索引

当关联数组的索引是字符串时，可以使用asorti()函数排序，如果是数字，直接for循环即可

```
# 当前关联数组
[root@chaogelinux ~]# awk 'BEGIN{t["z"]=66;t["q"]=88;t["a"]=3;for(i in t){print i,t[i]}}'
z 66
a 3
q 88

# 排序后
[root@chaogelinux ~]# awk 'BEGIN{t["z"]=66;t["q"]=88;t["a"]=3;\
> len=asorti(t,newt);\
> for(i=1;i<=len;i++){print i,newt[i]} }
> '
1 a
2 q
3 z
```

```
# 那么可以根据排序后的索引，对原关联数组再进行排序
[root@chaogelinux ~]# awk 'BEGIN{t["z"]=66;t["q"]=88;t["a"]=3;\
> len=asorti(t,newt);\
> for(i=1;i<=len;i++){print i,newt[i],t[newt[i]]}}'
1 a 3
2 q 88
3 z 66
```

时间函数

mktime(*datespec*)	将一个按YYYY MM DD HH MM SS [DST]格式指定的日期转换成时间戳值[①]
strftime(*format* [,*timestamp*])	将当前时间的时间戳或timestamp（如果提供了的话）转化格式化日期（采用shell函数date()的格式）
systime( )	返回当前时间的时间戳

时间函数用在日志文件格式化处理非常有用。

```
[root@chaogelinux ~]# awk 'BEGIN{
> date=systime()
> day=strftime("%A,%B %d,%Y",date)
> print day
> }'
星期一,十月 12,2020
```


自定义函数

自定义函数

```
function name([variables])
{
  语句
}
```

自定义函数必须写在awk最开始的地方。

定义awk脚本

```
[root@chaogelinux ~]# cat func.awk
function find_min(num1,num2)
{
  if (num1<num2)
    return num1
  return num2
}

function find_max(num1,num2)
{
  if (num1>num2)
    return num1
```

```
  return num2
}

function main(num1,num2)
{
  # 找最小值
  result=find_min(num1,num2)
  print "最小值= ",result

  # 找最大值
  result=find_max(num1,num2)
  print "最大值= ",result
}
BEGIN {
main(10,30)
}


# 执行
[root@chaogelinux ~]# awk -f func.awk
最小值=  10
最大值=  30
```


awk实践

现有一个数据文件，可以使用awk进行格式化数据处理。

```
[root@chaogelinux ~]# cat scores.txt
Rich Blum,team1,100,115,95
Barbara Blum,team1,110,115,100
Christine Bresnahan,team2,120,115,118
Tim Bresnahan,team2,125,112,116
```

> 对每只队伍的成绩排序，且计算总平均分

```
# 脚本
[root@chaogelinux ~]#
c[root@chaogelinux ~]# cat bowling.sh
#!/bin/bash
# for循环首先迭代出队名然后去重
for team in $(awk -F, '{print $2}' scores.txt|uniq)
do
  # 循环内部计算，传递shell变量给awk
  awk -v team=$team 'BEGIN{FS=",";total=0}
  {
    # 如果队名一致，就计算三场总分
    if ($2==team)
```

```
  {
    total+=$3+$4+$5;
}
}
  END {
    # 求平均数
    avg=total/6;
    print "Total for",team,"is",total,",the average is ",avg

}' scores.txt
done
```

执行结果

```
[root@chaogelinux ~]# bash bowling.sh
Total for team1 is 635 ,the average is  105.833
Total for team2 is 706 ,the average is  117.667
```