19 文本处理工具awk编程

文本处理工具awk编程

`awk` 的工作流程： `awk '[ BEGIN{动作} ] 模式{动作} [ END {动作} ]' 文件...`

一、流程控制语句

流程控制语句是所有程序设计语言必不可少的部分，每一门语言都支持某些执行流程控制的语句。

`awk` 提供了完整的流程控制语句。

1.1 if条件语句

`if…else` 语句其语法格式为

```shell
if （条件表达式）
    语句1
else
    语句2
```

或：

```shell
if （条件表达式） 语句1; else 语句2
```

如果条件表达式的判断结构为真，则执行语句1，否则执行语句2。

格式中的 `语句1` 可以是**多个语句**，为了方便 `awk` 判断，可以将**多个语句用 `{}` 括起来**。

`awk` 分支结构允许嵌套，其格式为

Shell | 复制代码

```shell
if （条件表达式1）
  {if （条件表达式2）
    语句1
  else
    语句2
  }
  语句3
else {if （条件表达式3）
    语句4
  else
    语句5
  }
  语句6
```

案例1：根据UID判断系统中用户类别

Shell | 复制代码

```
# 根据/etc/passwd第三列UID的大小判断是否为系统用户
# 1 ~ 200：由Linux发行版自行建立的系统账号
# 201 ~ 999：若用户有系统账号需求时，就可以使用的账号UID
# 1000以上：普通用户
[root@Shell ~]# awk -F: '{if($3<=200){name="system"} else{name="user"} print $1,name}' /etc/passwd
root system
bin system
daemon system
adm system
lp system
sync system
shutdown system
halt system
mail system
operator system
games system
ftp system
nobody system
systemd-network system
dbus system
polkitd user
sshd system
postfix system
chrony user

# 语句过长可用\续行
[root@Shell ~]# awk -F: '{if($3<=200){name="system"}else{name="user"} \
> print $1,name}' /etc/passwd
root system
bin system
daemon system
adm system
lp system
sync system
shutdown system
halt system
mail system
operator system
games system
ftp system
nobody system
systemd-network system
dbus system
polkitd user
```

```
sshd system
postfix system
chrony user

# 模式部分可自动续行，注意'位置
[root@Shell ~]# awk -F: '
> {
> if($3<=200)
> {name="system"}
> else
> {name="user"}
> print $1,name
> }' /etc/passwd
root system
bin system
daemon system
adm system
lp system
sync system
shutdown system
halt system
mail system
operator system
games system
ftp system
nobody system
systemd-network system
dbus system
polkitd user
sshd system
postfix system
chrony user
```

案例2：统计系统用户数

```shell
[root@Shell ~]# awk -F: '{if ($3>0 && $3<1000){++i}}{print i}' /etc/passwd

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
# END在所有行处理完才输出
[root@Shell ~]# awk -F: '{if ($3>0 && $3<1000){++i}} END {print i}' /etc/passwd
18
```

案例3：分别统计管理员个数，普通用户个数及系统用户个数

```shell
[root@Shell ~]# awk -F: '{if ($3==0){i++} else if ($3>999){j++} else {k++}} END {print "管理员"i,"普通用户"j,"系统用户"k}' /etc/passwd
管理员1 普通用户 系统用户18
```

1.2 while循环

`while` 循环语法结构为：

```shell
while 条件表达式
  {语句}
```

案例4：while循环

```shell
# 注意，模式为BEGIN
[root@shell ~]# awk 'BEGIN {i=1; while(i<=10) {print i;i++}}'
1
2
3
4
5
6
7
8
9
10
```

案例5：使用while循环打印文件/etc/passwd中以root开头的行出每个字段

```shell
[root@shell ~]# awk -F: '/^root/ {i=1;while(i<=NF){print $i;i++}}' /etc/passwd
root
x
0
0
root
/root
/bin/bash
```

`awk` 除 `while` 循环结构外，还有 `do...while` 循环结构。它在代码块结尾处对条件求值，而不像标准 `while` 循环那样在开始处求值，其语法结构为:

```shell
do
  语句
while （条件表达式）
```

与一般的 `while` 循环不同，由于在代码块之后对条件求值， **`do...while` 循环结构永远都至少执行一次**。

换句话说，当第一次遇到普通 `while` 循环时，如果条件为假，将永远不执行该循环。

案例6：使用do while循环打印文件/etc/passwd中以root开头的行出每个字段

```shell
[root@shell ~]# awk -F: '/^root/ {i=1;do {print $i;i++} while(i<=NF)}' /etc/passwd
root
x
0
0
root
/root
/bin/bash
```

1.3 for循环

`for` 循环中数组遍历的格式为：

```shell
for (变量 in 数组)
  {语句}
```

`for` 循环中固定循环的格式为：

```shell
for (变量;条件;表达式)
  {语句}
```

`for` 语句首先执行初始化语句，然后再检查条件。如果条件为真，则执行语句，然后执行递增或者递减操作。只要条件为真，循环就会一直执行。每次循环结束都会进行条件检查，若条件为假则结束循环。

案例7：使用for循环输出数字1至5

```shell
[root@shell ~]# awk 'BEGIN {for(i=1;i<=5;i++) {print i}}'
1
2
3
4
5
```

案例8：使用for循环打印文件/etc/passwd中以root开头的行出每个字段

```shell
[root@shell ~]# awk -F: '/^root/ {for(i=1;i<=NF;i++){print $i}} ' /etc/passwd
root
x
0
0
root
/root
/bin/bash
```

1.4 break命令、continue命令、exit命令

在 `awk` 的 `while` 语句、 `do...while` 和 `if` 语句中可以使用 `break` 命令、 `continue` 命令控制流程走向。 `break` 中断当前正在执行的循环并跳到循环外执行下一条语句， `continue` 用于在循环体内部结束本次循环，从而直接进入下一次循环迭代， `exit` 用于结束脚本程序的执行。

案例9：当累加和大于50时，使用break结束循环

```shell
[root@shell ~]# awk 'BEGIN{ sum=0;for(i=0;i<20;++i){sum +=i;if(sum>50)break;else print"Sum=",sum}}'
Sum= 0
Sum= 1
Sum= 3
Sum= 6
Sum= 10
Sum= 15
Sum= 21
Sum= 28
Sum= 36
Sum= 45
```

案例10：输出1到20之间的偶数

```shell
[root@shell ~]# awk 'BEGIN{for(i=1;i<=20;++i){if(i%2==0) print i;else continue}}'
2
4
6
8
10
12
14
16
18
20
```

案例11：当计算的和大于50时，结束awk程序

```shell
[root@shell ~]# awk 'BEGIN{ sum=0;for(i=0;i<20;++i){sum +=i;if(sum>50) exit(10);else print"Sum=",sum}}'
Sum= 0
Sum= 1
Sum= 3
Sum= 6
Sum= 10
Sum= 15
Sum= 21
Sum= 28
Sum= 36
Sum= 45
[root@shell ~]# echo $?
10
```

二、数组

`awk` 处理文本中数组是必不可少的，由于数组索引(下标)可以是**数字和字符串**，索引(下标)一般称作 `key` ，并且与对应数组元素的值**关联**。因此， `awk` 中的**数组**称为**关联数组**(Associative Arrays)。

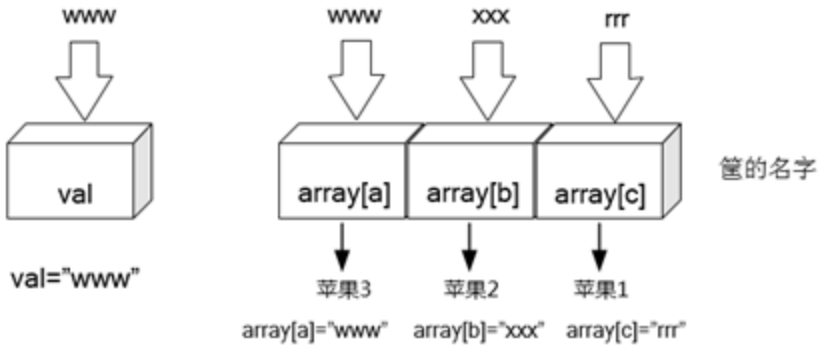
数组元素的 `key` 和值都放在 `awk` 内部程序的某一张表中，通常使用一定**散列算法**来存放，所以**数组元素并不是按照一定顺序来放**的。同理，也不是按照一定的顺序打印出来的，但可以使用管道来对所需的数据再次操作来达到效果。

`awk` 中的数组**不必提前声明**，也**不必声明大小**，因为它在运行时可以自动地增加或减少。数组元素用 `0` 或**空字符串**来初始化，这根据上下文而定。

通常 `awk` 数组用来从记录中收集信息，可以用于计算总和、统计单词以及跟踪模板被匹配的次数等。

`awk` 数组的结构如下所示。

如上图可以发现，`awk`数组就和酒店一样，数组的名称就像是酒店名称，数组元素名称就像酒店房间号码，每个数组元素里面的内容就像是酒店房间里面的人。



定义数组的格式为：`数组名[下标]=元素值`，其中索引下标可以是数字，也可以是字符。
使用数组的格式为：`数组名[下标]`
输出数组元素的值：`print 数组名[下标]`
遍历数组循环结构：`for(变量名in 数组名){print 数组名[变量名]}`

案例12：遍历数组，显示/etc/passwd的root账户

Shell | 复制代码

```
[root@shell ~]# awk -F: '{username[++i]=$1} END{print username[1]}' /etc/passwd
root
[root@shell ~]# awk -F: '{username[++i]=$1} END{print username[2]}' /etc/passwd
bin
```

案例12：遍历数组，显示/etc/passwd的所有账户

```shell
[root@shell ~]# awk -F: '{user[j++]=$1} END{for (i in user){print i,user[i]}}' /etc/passwd
17 postfix
4 lp
5 sync
6 shutdown
7 halt
8 mail
9 operator
10 games
11 ftp
12 nobody
13 systemd-network
0 root
14 dbus
1 bin
15 polkitd
2 daemon
16 sshd
3 adm
```

案例13：使用数组统计文件/etc/passwd中各种类型Shell的数量

```shell
[root@shell ~]# awk -F: '{shells[$NF]++} END{for (i in shells)print i,shells[i]}' /etc/passwd
/bin/sync 1
/bin/bash 1
/sbin/nologin 14
/sbin/halt 1
/sbin/shutdown 1
```

三、函数

定义、调用用户本身的函数是每个高级语言都具有的功能，`awk` 也不例外。

`awk` 函数分为**内置函数**和**自定义函数**。

3.1 内置函数

常用字符串内置函数如下表所示。

内置函数	作用
`gsub(x,y,z)`	在字串z中使用字串y替换与正则表达式x相匹配的所有字串，z默认为$0。相当于sed中的s `///g`
`sub(x,y,z)`	在字串z中使用字串y替换与正则表达式x相匹配的第一个字串，z默认为$0。相当于sed中的 `s///`
`length(string)`	返回string参数指定的字符串的长度（字符形式）。如果未给出string参数，则返回整个记录的长度（$0记录变量）。
`getline`	从输入中读取下一行内容。
`index(string1,string2)`	在由string1参数指定的字符串（其中有出现string2指定的参数）中，返回位置，从1开始编号。如果string2参数不在string1参数中出现，则返回0。
`substr(string,M,[N])`	返回具有N参数指定的字符数量字串。字串从string参数指定的字符串取得，其字符以M参数指定的位置开始。M参数指定为将string参数中的第一个字符作为编号1。如果未指定N参数，则字串的长度将是M参数指定的位置到string参数的末尾的长度。
`match(string,Ere)`	在string参数指定的字符串（Ere参数指定的扩展正则表达式出现在其中）中返回位置（字符形式），从1开始编号，或如果Ere参数不出现，则返回0（零）RSTART特殊变量设置为返回值，RLENGTH特殊变量设置为匹配的字符串的长度，或如果未找到任何匹配，则设置为-1（负一）。
`split(string,A,[Ere])`	将string参数指定的参数分割为数组元素A[1],A[2],...A[n]，并返回n变量的值。此分隔可以通过Ere参数指定的扩展正则表达式进行，或用当前字段分隔符（FS特殊变量）来进行（如果没有给出Ere参数）。除非上下文指明特定的元素还应具有一个数字值，否则A数组的元素用字符串值来创建。
`tolower(string)`	返回string参数指定的字符串，字符串中每个大写字符将更改为小写。

`sprintf(Format,Expr,Expr,…)`	根据Format参数指定的printf子例程格式字符串来格式化Expr参数指定的表达式并返回最后生成的字符串。

常用算术内置函数如下表所示。

内置函数	说明
`rand()`	产生0-1之间的浮点类型的随机数， `rand()` 产生随机数时需要设置一个参数，否则单独的 `rand()` 每次产生的随机数都是一样的。
`init(x)`	返回 `x` 的整数部分的值。
`sqrt()`	返回 `x` 的平方根。
`srand()`	建立 `rand()` 新的种子数，如果没有指定就用当天的时间。

案例14：统计用户名为4个字符的用户

```shell
# 传统方法
[root@shell ~]# awk -F: '$1~/^....$/ {count++;print $1} END{print count}' /etc/passwd
root
sync
halt
mail
dbus
sshd
6
# 使用awk内建函数length()计算并返回字串的长度
[root@shell ~]# awk -F: 'length($1)==4 {count++;print $1} END{print count}' /etc/passwd
root
sync
halt
mail
dbus
sshd
6
```

3.2 自定义函数

函数定义包括**函数名**、**函数参数**、**函数体**。

`awk` 函数的定义方法如下：

```shell
function 函数名（参数表）{
函数体
}
```

函数名是用户自定义函数的名称，函数名称以字母开头，后面可以是数字、字母或下画线的自由组合，`awk` 保留的关键字不能作为用户自定义函数的名称。

自定义函数可以接受参数，参数之间用逗号分隔，**参数不是必需的**，用户也可以定义没有任何输入参数的函数，函数体包含 `awk` 程序代码。

案例15：简易函数

```shell
[root@shell ~]# vi awk_test
11 22
[root@shell ~]# awk 'function test(a,b) { printf a "\t" b "\n"}
> { test($1,$2) } ' awk_test
11      22
```

当函数代码量比较大时，将函数写在外部文件中更合适。

案例16：返回两个数中的最大值和最小值

```shell
[root@shell ~]# vi functions.awk
function find_min(num1,num2)
{
    if (num1<num2)
        return num1
    else
        return num2
}

function find_max(num1,num2)
{
    if (num1>num2)
        return num1
    else
        return num2
}

function main(num1,num2)
{
    result1=find_max(num1,num2)
    result2=find_min(num1,num2)
    print "最小值：",result1,"最大值：",result2
}

BEGIN{
    main(10,20)
}
[root@shell ~]# awk -f functions.awk
最大值：  20 最小值：  10
```

案例17：自定义函数应用

上面的案例函数的参数是固定的，下面演示如何通过文件向函数传递参数。

Shell | 复制代码

```
[root@shell ~]# vi functions.awk
function find_min(num1,num2)
{
    if (num1<num2)
        return num1
    else
        return num2
}

function find_max(num1,num2)
{
    if (num1>num2)
        return num1
    else
        return num2
}

function main(num1,num2)
{
    result1=find_max(num1,num2)
    result2=find_min(num1,num2)
    print "最大值: ",result1,"最小值: ",result2
}

main($1,$2)

[root@shell ~]# awk -f functions.awk awk_test
最大值:  22 最小值:  11
# 修改样例文件中数据的分隔符
[root@shell ~]# vi awk_test
11:22
[root@shell ~]# awk -f functions.awk -F":" awk_test
最大值:  22 最小值:  11
# 增加样例文件中的记录
[root@shell ~]# vi awk_test
11:22
33:44
[root@shell ~]# awk -f functions.awk -F":" awk_test
最大值:  22 最小值:  11
最大值:  44 最小值:  33
```

注意：在无输入文件或标准输入时，函数调用部分要放在 `BEGIN` 模式中，为什么？

小结

- 流程控制：语法格式
- 数组：语法
- 函数：内置函数功能，自定义函数定义及调用

课程目标

- 知识目标：熟练掌握 `awk` 命令的基本语法。
- 技能目标：能够利用 `awk` 命令完成实战场景的处理。

课外拓展

- 进一步了解 `awk` 命令的应用场景。

参考资料

- `awk --help` 或 `man awk`
- 《Linux Shell核心编程指南》，丁明一，电子工业出版社