

# Linux命令大全

当前位置: 首页 » 软件·打印·开发·工具 » awk

## awk命令

**awk**是一种编程语言，用于在linux/unix下对文本和数据进行处理。数据可以来自标准输入(stdin)、一个或多个文件，或其它命令的输出。它支持用户自定义函数和动态正则表达式等先进功能，是linux/unix下的一个强大编程工具。它在命令行中使用，但更多是作为脚本来使用。awk有很多内建的功能，比如数组、函数等，这是它和C语言的相同之处，灵活性是awk最大的优势。

### awk命令格式和选项

#### 语法形式

```
awk [options] 'script' var=value file(s)
awk [options] -f scriptfile var=value file(s)
```

#### 常用命令选项

- **-F fs** fs指定输入分隔符，fs可以是字符串或正则表达式，如-F:
- **-v var=value** 赋值一个用户定义变量，将外部变量传递给awk
- **-f scripfile** 从脚本文件中读取awk命令
- **-m[fr] val** 对val值设置内在限制，-mf选项限制分配给val的最大块数目；-mr选项限制记录的最大数目。这两个功能是Bell实验室版awk的扩展功能，在标准awk中不适用。

### awk模式和操作

awk脚本是由模式和操作组成的。

#### 模式

模式可以是以下任意一个：

- **/正则表达式/**: 使用通配符的扩展集。

#### 本文索引

[隐藏]

- [awk命令格式和选项](#)
- [awk模式和操作](#)
  - [模式](#)
  - [操作](#)
- [awk脚本基本结构](#)
  - [awk的工作原理](#)
- [awk内置变量（预定义变量）](#)
- [将外部变量值传递给awk](#)
- [awk运算与判断](#)
  - [算术运算符](#)
  - [赋值运算符](#)
  - [逻辑运算符](#)
  - [正则运算符](#)
  - [关系运算符](#)
  - [其它运算符](#)
  - [运算级优先级表](#)
- [awk高级输入输出](#)
  - [读取下一条记录](#)
  - [简单地读取一条记录](#)
  - [关闭文件](#)
  - [输出到一个文件](#)
- [设置字段定界符](#)
- [流程控制语句](#)
  - [条件判断语句](#)
  - [循环语句](#)
    - [while语句](#)
    - [for循环](#)
    - [do循环](#)

- 关系表达式: 使用运算符进行操作, 可以是字符串或数字的比较测试。
- 模式匹配表达式: 用运算符 `~` (匹配) 和 `~!` (不匹配)。
- BEGIN语句块、pattern语句块、END语句块: 参见[awk的工作原理](#)

- [其他语句](#)
- [数组应用](#)
  - [数组的定义](#)
  - [数组相关函数](#)
  - [二维、多维数组使用](#)
- [内置函数](#)
  - [算术函数](#)
  - [字符串函数](#)
  - [一般函数](#)
  - [时间函数](#)

## 操作

操作由一个或多个命令、函数、表达式组成, 之间由换行符或分号隔开, 并位于大括号内, 主要部分是:

- [变量或数组赋值](#)
- [输出命令](#)
- [内置函数](#)
- [控制流语句](#)

## awk脚本基本结构

```
awk 'BEGIN{ print "start" } pattern{ commands } END{ print "end" }' file
```

一个awk脚本通常由: **BEGIN语句块**、**能够使用模式匹配的通用语句块**、**END语句块**3部分组成, 这三个部分是可选的。任意一个部分都可以不出现在脚本中, **脚本通常是被单引号或双引号中**, 例如:

```
awk 'BEGIN{ i=0 } { i++ } END{ print i }' filename
awk "BEGIN{ i=0 } { i++ } END{ print i }" filename
```

## awk的工作原理

```
awk 'BEGIN{ commands } pattern{ commands } END{ commands }'
```

- 第一步: 执行 `BEGIN{ commands }` 语句块中的语句;
- 第二步: 从文件或标准输入(stdin)读取一行, 然后执行 `pattern{ commands }` 语句块, 它**逐行扫描文件**, 从第一行到最后一行重复这个过程, 直到文件全部被读取完毕。
- 第三步: 当读至输入流末尾时, 执行 `END{ commands }` 语句块。

**BEGIN语句块**在awk开始从输入流中**读取行之前被执行**, 这是一个可选的语句块, 比如变量初始化、打印输出表格的表头等语句通常可以写在BEGIN语句块中。

**END**语句块在awk从输入流中读取完所有的行之后即被执行，比如打印所有行的分析结果这类信息汇总都是在END语句块中完成，它也是一个可选语句块。

**pattern**语句块中的通用命令是最重要的部分，它也是可选的。如果没有提供**pattern**语句块，则默认执行 `{ print }`，即打印每一个读取到的行，awk读取的每一行都会执行该语句块。

## 示例

```
echo -e "A line 1\nA line 2" | awk 'BEGIN{ print "Start" } { print } END{ print "End" }'
```

Start  
A line 1  
A line 2  
End

当使用不带参数的 `print` 时，它就打印当前行，当 `print` 的参数是以逗号进行分隔时，打印时则以空格作为定界符。在awk的**print**语句块中双引号是被当作拼接符使用，例如：

```
echo | awk '{ var1="v1"; var2="v2"; var3="v3"; print var1,var2,var3; }'
```

v1 v2 v3

双引号拼接使用：

```
echo | awk '{ var1="v1"; var2="v2"; var3="v3"; print var1"="var2"="var3; }'
```

v1=v2=v3

`{ }`类似一个循环体，会对文件中的每一行进行迭代，通常变量初始化语句（如：`i=0`）以及打印文件头部的语句放入BEGIN语句块中，将打印的结果等语句放在END语句块中。

## awk内置变量（预定义变量）

说明：`[A][N][P][G]`表示第一个支持变量的工具，`[A]=awk`、`[N]=nawk`、`[P]=POSIXawk`、`[G]=gawk`

**\$n** 当前记录的第n个字段，比如n为1表示第一个字段，n为2表示第二个字段。

**\$0** 这个变量包含执行过程中当前行的文本内容。

**[N] ARGC** 命令行参数的数目。

**[G] ARGIND** 命令行中当前文件的位置（从0开始算）。

**[N] ARGV** 包含命令行参数的数组。

**[G] CONVFMT** 数字转换格式（默认值为%.6g）。

**[P] ENVIRON** 环境变量关联数组。

**[N] ERRNO** 最后一个系统错误的描述。

**[G] FIELDWIDTHS** 字段宽度列表（用空格键分隔）。

**[A] FILENAME** 当前输入文件的名。

**[P] FNR** 同NR，但相对于当前文件。

- [A] **FS** 字段分隔符 (默认是任何空格)。
- [G] **IGNORECASE** 如果为真, 则进行忽略大小写的匹配。
- [A] **NF** 表示字段数, 在执行过程中对应于当前的字段数。
- [A] **NR** 表示记录数, 在执行过程中对应于当前的行号。
- [A] **OFMT** 数字的输出格式 (默认值是%.6g)。
- [A] **OFS** 输出字段分隔符 (默认值是一个空格)。
- [A] **ORS** 输出记录分隔符 (默认值是一个换行符)。
- [A] **RS** 记录分隔符 (默认是一个换行符)。
- [N] **RSTART** 由match函数所匹配的字符串的第一个位置。
- [N] **RLENGTH** 由match函数所匹配的字符串的长度。
- [N] **SUBSEP** 数组下标分隔符 (默认值是34)。

## 示例

```
echo -e "line1 f2 f3\nline2 f4 f5\nline3 f6 f7" | awk '{print "Line No:"NR", No of fields:"NF, "$0="$0, "$1="$1, "$2="$2, "$3="$3}'
Line No:1, No of fields:3 $0=line1 f2 f3 $1=line1 $2=f2 $3=f3
Line No:2, No of fields:3 $0=line2 f4 f5 $1=line2 $2=f4 $3=f5
Line No:3, No of fields:3 $0=line3 f6 f7 $1=line3 $2=f6 $3=f7
```

使用 `print $NF` 可以打印出一行中的最后一个字段, 使用 `$(NF-1)` 则是打印倒数第二个字段, 其他以此类推:

```
echo -e "line1 f2 f3\n line2 f4 f5" | awk '{print $NF}'
f3
f5
```

```
echo -e "line1 f2 f3\n line2 f4 f5" | awk '{print $(NF-1)}'
f2
f4
```

打印每一行的第二和第三个字段:

```
awk '{ print $2,$3 }' filename
```

统计文件中的行数:

```
awk 'END{ print NR }' filename
```

以上命令只使用了END语句块, 在读入每一行的时, awk会将NR更新为对应的行号, 当到达最后一行NR的值就是最后一行的行号, 所以END语句块中的NR就是文件的行数。

一个每一行中第一个字段值累加的例子:

```
seq 5 | awk 'BEGIN{ sum=0; print "总和: " } { print $1"+"; sum+=$1 } END{ print "等于"; print sum }'
```

总和:  
1+  
2+  
3+  
4+  
5+  
等于  
15

## 将外部变量值传递给awk

借助 `-v` 选项, 可以将外部值 (并非来自stdin) 传递给awk:

```
VAR=10000  
echo | awk -v VARIABLE=$VAR '{ print VARIABLE }'
```

另一种传递外部变量方法:

```
var1="aaa"  
var2="bbb"  
echo | awk '{ print v1,v2 }' v1=$var1 v2=$var2
```

当输入来自于文件时使用:

```
awk '{ print v1,v2 }' v1=$var1 v2=$var2 filename
```

以上方法中, 变量之间用空格分隔作为awk的命令行参数跟随在BEGIN、{}和END语句块之后。

## awk运算与判断

作为一种程序设计语言所应具有的特点之一, awk支持多种运算, 这些运算与C语言提供的基本相同。awk还提供了一系列内置的运算函数 (如log、sqr、cos、sin等) 和一些用于对字符串进行操作 (运算) 的函数 (如length、substr等等)。这些函数的引用大大的提高了awk的运算功能。作为对条件转移指令的一部分, 关系判断是每种程序设计语言都具备的功能, awk也不例外, awk中允许进行多种测试, 作为样式匹配, 还提供了模式匹配表达式~ (匹配) 和~! (不匹配)。作为对测试的一种扩充, awk也支持用逻辑运算符。

### 算术运算符

运算符	描述
+ -	加, 减
* / %	乘, 除与求余
+ - !	一元加, 减和逻辑非
^ ***	求幂
++ --	增加或减少, 作为前缀或后缀

例:

```
awk 'BEGIN{a="b";print a++,++a;}'
0 2
```

**注意: 所有用作算术运算符进行操作, 操作数自动转为数值, 所有非数值都变为0**

## 赋值运算符

运算符	描述
= += -= *= /= %= ^= **=	赋值语句

例:

```
a+=5; 等价于: a=a+5; 其它同类
```

## 逻辑运算符

运算符	描述
	逻辑或
&&	逻辑与

例:

```
awk 'BEGIN{a=1;b=2;print (a>5 && b<=2),(a>5 || b<=2);}'
0 1
```

## 正则运算符

运算符	描述
~ ~!	匹配正则表达式和不匹配正则表达式

例:

```
awk 'BEGIN{a="100testa";if(a ~ /^100*/){print "ok";}}'
ok
```

## 关系运算符

运算符	描述
< <= > >= != ==	关系运算符

例:

```
awk 'BEGIN{a=11;if(a >= 9){print "ok";}}'
ok
```

注意: > < 可以作为字符串比较, 也可以用作数值比较, 关键看操作数如果是字符串就会转换为字符串比较。两个都为数字才转为数值比较。字符串比较: 按照ASCII码顺序比较。

## 其它运算符

运算符	描述
\$	字段引用
空格	字符串连接符
?:	C条件表达式
in	数组中是否存在某键值

例:

```
awk 'BEGIN{a="b";print a=="b"? "ok": "err";}'
ok
```

```
awk 'BEGIN{a="b";arr[0]="b";arr[1]="c";print (a in arr);}'
0
```

```
awk 'BEGIN{a="b";arr[0]="b";arr["b"]="c";print (a in arr);}'
1
```

## 运算级优先级表

级别	运算符	说明
1	=, +=, -=, *=, /=, %=, &=, ^=,  =, <=, >=	赋值、运算、赋值
2		逻辑或
3	&&	逻辑与
4		按位或
5	^	按位异或
6	&	按位与
7	==, !=	等于、不等于
8	<=, >=, <, >	小于等于、大于等于、小于、大于
9	<<, >>	按位左移、按位右移
10	+, -	加、减
11	*, /, %	乘、除、取模
12	!, ~	逻辑非、按位取反或补码
13	-, +	正、负

级别越高越优先

## awk高级输入输出

### 读取下一条记录

awk中 `next` 语句使用：在循环逐行匹配，如果遇到 `next`，就会跳过当前行，直接忽略下面语句。而进行下一行匹配。`next` 语句一般用于多行合并：

```
cat text.txt
a
b
c
d
e
```



```
awk 'NR%2==1{next}{print NR,$0;}' text.txt
2 b
4 d
```

当记录行号除以2余1, 就跳过当前行。下面的 `print NR,$0` 也不会执行。下一行开始, 程序有开始判断 `NR%2` 值。这个时候记录行号是 `: 2`, 就会执行下面语句块: `'print NR,$0'`

分析发现需要将包含有“web”行进行跳过, 然后需要将内容与下面行合并为一行:

```
cat text.txt
web01[192.168.2.100]
httpd          ok
tomcat         ok
sendmail       ok
web02[192.168.2.101]
httpd          ok
postfix        ok
web03[192.168.2.102]
mysqld         ok
httpd          ok
0
awk '/^web/{T=$0;next;}{print T":t"$0;}' test.txt
web01[192.168.2.100]: httpd          ok
web01[192.168.2.100]: tomcat         ok
web01[192.168.2.100]: sendmail       ok
web02[192.168.2.101]: httpd          ok
web02[192.168.2.101]: postfix        ok
web03[192.168.2.102]: mysqld         ok
web03[192.168.2.102]: httpd          ok
```

## 简单地读取一条记录

`awk getline` 用法: 输出重定向需用到 `getline` 函数。 `getline` 从标准输入、管道或者当前正在处理的文件之外的其他输入文件获得输入。它负责从输入获得下一行的内容, 并给 `NF,NR` 和 `FNR` 等内建变量赋值。如果得到一条记录, `getline` 函数返回1, 如果到达文件的末尾就返回0, 如果出现错误, 例如打开文件失败, 就返回-1。

**getline语法:** `getline var`, 变量 `var` 包含了特定行的内容。

`awk getline` 从整体上来说, 用法说明:

- 当其左右无重定向符 `|` 或 `<` 时: `getline` 作用于当前文件, 读入当前文件的第一行给其后跟的变量 `var` 或 `$0` (无变量), 应该注意到, 由于 `awk` 在处理 `getline` 之前已经读入了一行, 所以 `getline` 得到的返回结果是隔行的。

- 当其左右有**重定向符** `|` 或 `<` 时: `getline`则作用于定向输入文件, 由于该文件是刚打开, 并没有被`awk`读入一行, 只是`getline`读入, 那么`getline`返回的是该文件的第一行, 而不是隔行。

### 示例:

执行linux的 `date` 命令, 并通过管道输出给 `getline`, 然后再把输出赋值给自定义变量`out`, 并打印它:

```
awk 'BEGIN{ "date" | getline out; print out }' test
```

执行shell的`date`命令, 并通过管道输出给`getline`, 然后`getline`从管道中读取并将输入赋值给`out`, `split`函数把变量`out`转化成数组`mon`, 然后打印数组`mon`的第二个元素:

```
awk 'BEGIN{ "date" | getline out; split(out,mon); print mon[2] }' test
```

命令`ls`的输出传递给`getline`作为输入, 循环使`getline`从`ls`的输出中读取一行, 并把它打印到屏幕。这里没有输入文件, 因为`BEGIN`块在打开输入文件前执行, 所以可以忽略输入文件。

```
awk 'BEGIN{ while( "ls" | getline) print }'
```

## 关闭文件

`awk`中允许在程序中关闭一个输入或输出文件, 方法是使用`awk`的`close`语句。

```
close("filename")
```

`filename`可以是`getline`打开的文件, 也可以是`stdin`, 包含文件名的变量或者`getline`使用的确切命令。或一个输出文件, 可以是`stdout`, 包含文件名的变量或使用管道的确切命令。

## 输出到一个文件

`awk`中允许用如下方式将结果输出到一个文件:

```
echo | awk '{printf("hello word!\n") > "datafile"}'  
或  
echo | awk '{printf("hello word!\n") >> "datafile"}'
```

## 设置字段定界符

---

默认的字段定界符是空格, 可以使用 `-F "定界符"` 明确指定一个定界符:

```
awk -F: '{ print $NF }' /etc/passwd
或
awk 'BEGIN{ FS=":" } { print $NF }' /etc/passwd
```

在 `BEGIN` 语句块 中则可以用 `OFS="定界符"` 设置输出字段的定界符。

## 流程控制语句

在linux awk的while、do-while和for语句中允许使用break,continue语句来控制流程走向, 也允许使用exit这样的语句来退出。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。if 是流程选择用法。awk中, 流程控制语句, 语法结构, 与c语言类型。有了这些语句, 其实很多shell程序都可以交给awk, 而且性能是非常快的。下面是各个语句用法。

### 条件判断语句

```
if (表达式)
    语句1
else
    语句2
```

格式中语句1可以是多个语句, 为了方便判断和阅读, 最好将多个语句用{}括起来。awk分枝结构允许嵌套, 其格式为:

```
if (表达式)
    {语句1}
else if (表达式)
    {语句2}
else
    {语句3}
```

示例:

```
awk 'BEGIN{
test=100;
if(test>90){
    print "very good";
}
else if(test>60){
    print "good";
}
else{
```

```
    print "no pass";  
  }  
}'  
  
very good
```

每条命令语句后面可以用 `;` 分号结尾。

## 循环语句

### while语句

```
while (表达式)  
{ 语句}
```

示例:

```
awk 'BEGIN{  
  test=100;  
  total=0;  
  while(i<=test){  
    total+=i;  
    i++;  
  }  
  print total;  
}'  
5050
```

### for循环

for循环有两种格式:

**格式1:**

```
for(变量 in 数组)  
{ 语句}
```

示例:

```
awk 'BEGIN{  
  for(k in ENVIRON){  
    print k=="ENVIRON[k];
```

```
}  
  
}'  
TERM=linux  
G_BROKEN_FILENAMES=1  
SHLVL=1  
pwd=/root/text  
...  
logname=root  
HOME=/root  
SSH_CLIENT=192.168.1.21 53087 22
```

注: ENVIRON是awk常量, 是子典型数组。

## 格式2:

```
for (变量;条件;表达式)  
{语句}
```

示例:

```
awk 'BEGIN{  
total=0;  
for(i=0;i<=100;i++){  
total+=i;  
}  
print total;  
}'  
5050
```

## do循环

```
do  
{语句} while(条件)
```

例子:

```
awk 'BEGIN{  
total=0;  
i=0;  
do {total+=i;i++;} while(i<=100)  
print total;  
}'  
5050
```

## 其他语句

- **break** 当 break 语句用于 while 或 for 语句时，导致退出程序循环。
- **continue** 当 continue 语句用于 while 或 for 语句时，使程序循环移动到下一个迭代。
- **next** 能够导致读入下一个输入行，并返回到脚本的顶部。这可以避免对当前输入行执行其他的操作过程。
- **exit** 语句使主输入循环退出并将控制转移到END,如果END存在的话。如果没有定义END规则，或在END中应用exit语句，则终止脚本的执行。

## 数组应用

---

数组是awk的灵魂，处理文本中最不能少的就是它的数组处理。因为数组索引（下标）可以是数字和字符串在awk中数组叫做关联数组(associative arrays)。awk 中的数组不必提前声明，也不必声明大小。数组元素用0或空字符串来初始化，这根据上下文而定。

### 数组的定义

数字做数组索引（下标）：

```
Array[1]="sun"  
Array[2]="kai"
```

字符串做数组索引（下标）：

```
Array["first"]="www"  
Array["last"]="name"  
Array["birth"]="1987"
```

使用中 `print Array[1]` 会打印出sun；使用 `print Array[2]` 会打印出kai；使用 `print ["birth"]` 会得到1987。

### 读取数组的值

```
{ for(item in array) {print array[item]}; } #输出的顺序是随机的  
{ for(i=1;i<=len;i++) {print array[i]}; } #Len是数组的长度
```

### 数组相关函数

得到数组长度：

```
awk 'BEGIN{info="it is a test";lens=split(info,tA," ");print length(tA),lens;}'
4 4
```

length返回字符串以及数组长度，split进行分割字符串为数组，也会返回分割得到数组长度。

```
awk 'BEGIN{info="it is a test";split(info,tA," ");print asort(tA);}'
4
```

asort对数组进行排序，返回数组长度。

输出数组内容（无序，有序输出）：

```
awk 'BEGIN{info="it is a test";split(info,tA," ");for(k in tA){print k,tA[k];}}'
4 test
1 it
2 is
3 a
```

for...in 输出，因为数组是关联数组，默认是无序的。所以通过 for...in 得到是无序的数组。如果需要得到有序数组，需要通过下标获得。

```
awk 'BEGIN{info="it is a test";tlen=split(info,tA," ");for(k=1;k<=tlen;k++){print k,tA[k];}}'
1 it
2 is
3 a
4 test
```

注意：数组下标是从1开始，与C数组不一样。

判断键值存在以及删除键值：

#错误的判断方法：

```
awk 'BEGIN{tB["a"]="a1";tB["b"]="b1";if(tB["c"]!="1"){print "no found";};for(k in tB){print k,tB[k];}}'
no found
a a1
b b1
c
```

以上出现奇怪问题，tB["c"] 没有定义，但是循环时候，发现已经存在该键值，它的值为空，这里需要注意，awk数组是关联数组，只要通过数组引用它的key，就会自动创建改序列。

**#正确判断方法:**

```
awk 'BEGIN{tB["a"]="a1";tB["b"]="b1";if( "c" in tB){print "ok";};for(k in tB){print
k,tB[k];}}'
a a1
b b1
```

`if(key in array)` 通过这种方法判断数组中是否包含 `key` 键值。

**#删除键值:**

```
[chengmo@localhost ~]$ awk 'BEGIN{tB["a"]="a1";tB["b"]="b1";delete tB["a"];for(k in tB)
{print k,tB[k];}}'
b b1
```

`delete array[key]` 可以删除, 对应数组 `key` 的, 序列值。

**二维、多维数组使用**

awk的多维数组在本质上是一维数组, 更确切一点, awk在存储上并不支持多维数组。awk提供了逻辑上模拟二维数组的访问方式。例如, `array[2,4]=1` 这样的访问是允许的。awk使用一个特殊的字符串

`SUBSEP (034)` 作为分割字段, 在上面的例子中, 关联数组array存储的键值实际上是2◆344。

类似一维数组的成员测试, 多维数组可以使用 `if ( (i,j) in array)` 这样的语法, 但是下标必须放置在圆括号中。类似一维数组的循环访问, 多维数组使用 `for ( item in array )` 这样的语法遍历数组。与一维数组不同的是, 多维数组必须使用 `split()` 函数来访问单独的下标分量。

```
awk 'BEGIN{
for(i=1;i<=9;i++){
for(j=1;j<=9;j++){
tarr[i,j]=i*j; print i,"*",j,"=",tarr[i,j];
}
}
}'
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
...
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
```



可以通过 `array[k, k2]` 引用获得数组内容。

另一种方法:

```
awk 'BEGIN{
for(i=1;i<=9;i++){
  for(j=1;j<=9;j++){
    tarr[i,j]=i*j;
  }
}
for(m in tarr){
  split(m,tarr2,SUBSEP); print tarr2[1],"*",tarr2[2],"=",tarr[m];
}
}'
```

## 内置函数

awk内置函数，主要分以下3种类似：算数函数、字符串函数、其它一般函数、时间函数。

### 算术函数

格式	描述
<code>atan2( y, x )</code>	返回 $y/x$ 的反正切。
<code>cos( x )</code>	返回 $x$ 的余弦； $x$ 是弧度。
<code>sin( x )</code>	返回 $x$ 的正弦； $x$ 是弧度。
<code>exp( x )</code>	返回 $x$ 幂函数。
<code>log( x )</code>	返回 $x$ 的自然对数。
<code>sqrt( x )</code>	返回 $x$ 平方根。
<code>int( x )</code>	返回 $x$ 的截断至整数的值。
<code>rand( )</code>	返回任意数字 $n$ ，其中 $0 \leq n < 1$ 。
<code>srand( [expr] )</code>	将 <code>rand</code> 函数的种子值设置为 <code>Expr</code> 参数的值，或如果省略 <code>Expr</code> 参数则使用某天的时间。返回先前的种子值。

举例说明:

```
awk 'BEGIN{OFMT="%.3f";fs=sin(1);fe=exp(10);fl=log(10);fi=int(3.1415);print fs,fe,fl,fi;}'
0.841 22026.466 2.303 3
```

OFMT 设置输出数据格式是保留3位小数。

获得随机数:

```
awk 'BEGIN{srand();fr=int(100*rand());print fr;}'
78
awk 'BEGIN{srand();fr=int(100*rand());print fr;}'
31
awk 'BEGIN{srand();fr=int(100*rand());print fr;}'
41
```

## 字符串函数

格式	描述
gsub( Ere, Repl, [ In ] )	除了正则表达式所有具体值被替代这点, 它和 sub 函数完全一样地执行。
sub( Ere, Repl, [ In ] )	用 Repl 参数指定的字符串替换 In 参数指定的字符串中的由 Ere 参数指定的扩展正则表达式的第一个具体值。sub 函数返回替换的数量。出现在 Repl 参数指定的字符串中的 & (和符号) 由 In 参数指定的与 Ere 参数的指定的扩展正则表达式匹配的字符串替换。如果未指定 In 参数, 缺省值是整个记录 (\$0 记录变量)。
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 (零)。
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)。
blength [(String)]	返回 String 参数指定的字符串的长度 (以字节为单位)。如果未给出 String 参数, 则返回整个记录的长度 (\$0 记录变量)。
substr( String, M, [ N ] )	返回具有 N 参数指定的字符数量子串。子串从 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, 则子串的长度将是 M 参数指定的位置到 String 参数的末尾 的长度。

格式	描述
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 数组中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。大写和小写的映射由当前语言环境的 LC_CTYPE 范畴定义。
toupper( String )	返回 String 参数指定的字符串，字符串中每个小写字符将更改为大写。大写和小写的映射由当前语言环境的 LC_CTYPE 范畴定义。
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。

注：Ere都可以是正则表达式。

## gsub,sub使用

```
awk 'BEGIN{info="this is a test2010test!";gsub(/[0-9]+/, "!");print info}'
this is a test!test!
```

在 info中查找满足正则表达式， `/[0-9]+/` 用 `"!"` 替换，并且替换后的值，赋值给info 未给info值，默认是 `$0`

## 查找字符串 (index使用)

```
awk 'BEGIN{info="this is a test2010test!";print index(info,"test")?"ok":"no found";}'
ok
```

未找到，返回0

## 正则表达式匹配查找(match使用)

```
awk 'BEGIN{info="this is a test2010test!";print match(info,/[0-9]+)?"ok":"no found";}'
ok
```

## 截取字符串(substr使用)

```
[wangsl@centos5 ~]$ awk 'BEGIN{info="this is a test2010test!";print substr(info,4,10);}'
s is a tes
```

从第 4 个 字符开始，截取10个长度字符串

## 字符串分割 (split使用)

```
awk 'BEGIN{info="this is a test";split(info,tA," ");print length(tA);for(k in tA){print
k,tA[k];}}'
4
4 test
1 this
2 is
3 a
```

分割info，动态创建数组tA，这里比较有意思，`awk for ...in` 循环，是一个无序的循环。并不是从数组下标 1...n，因此使用时需要注意。

## 格式化字符串输出 (sprintf使用)

### 格式化字符串格式：

其中格式化字符串包括两部分内容：一部分是正常字符，这些字符将按原样输出；另一部分是格式化规定字符，以 `"%"` 开始，后跟一个或几个规定字符,用来确定输出内容格式。

格式	描述
%d	十进制有符号整数
%u	十进制无符号整数
%f	浮点数
%s	字符串
%c	单个字符
%p	指针的值
%e	指数形式的浮点数

格式	描述
%x	%X 无符号以十六进制表示的整数
%o	无符号以八进制表示的整数
%g	自动选择合适的表示法

```
awk 'BEGIN{n1=124.113;n2=-1.224;n3=1.2345;
printf("%.2f,%.2u,%.2g,%X,%on",n1,n2,n3,n1,n1);}'
124.11,18446744073709551615,1.2,7C,174
```

## 一般函数

格式	描述
close( Expression )	用同一个带字符串值的 Expression 参数来关闭由 print 或 printf 语句打开的或调用 getline 函数打开的文件或管道。如果文件或管道成功关闭, 则返回 0; 其它情况下返回非零值。如果打算写一个文件, 并稍后在同一个程序中读取文件, 则 close 语句是必需的。
system(command )	执行 Command 参数指定的命令, 并返回退出状态。等同于 system 子例程。
Expression   getline [ Variable ]	从来自 Expression 参数指定的命令的输出中通过管道传送的流中读取一个输入记录, 并将该记录的值指定给 Variable 参数指定的变量。如果当前未打开将 Expression 参数的值作为其命令名称的流, 则创建流。创建的流等同于调用 popen 子例程, 此时 Command 参数取 Expression 参数的值且 Mode 参数设置为一个是 r 的值。只要流保留打开且 Expression 参数求得同一个字符串, 则对 getline 函数的每次后续调用读取另一个记录。如果未指定 Variable 参数, 则 \$0 记录变量和 NF 特殊变量设置为从流读取的记录。
getline [ Variable ] < Expression	从 Expression 参数指定的文件读取输入的下一个记录, 并将 Variable 参数指定的变量设置为该记录的值。只要流保留打开且 Expression 参数对同一个字符串求值, 则对 getline 函数的每次后续调用读取另一个记录。如果未指定 Variable 参数, 则 \$0 记录变量和 NF 特殊变量设置为从流读取的记录。
getline [ Variable ]	将 Variable 参数指定的变量设置为从当前输入文件读取的下一个输入记录。如果未指定 Variable 参数, 则 \$0 记录变量设置为该记录的值, 还将设置 NF、NR 和 FNR 特殊变量。

## 打开外部文件 (close用法)

```
awk 'BEGIN{while("cat /etc/passwd"|getline){print $0;};close("/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
```

## 逐行读取外部文件(getline使用方法)

```
awk 'BEGIN{while(getline < "/etc/passwd"){print $0;};close("/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
```

```
awk 'BEGIN{print "Enter your name:";getline name;print name;}'
Enter your name:
chengmo
chengmo
```

## 调用外部应用程序(system使用方法)

```
awk 'BEGIN{b=system("ls -al");print b;}'
total 42092
drwxr-xr-x 14 chengmo chengmo    4096 09-30 17:47 .
drwxr-xr-x 95 root    root      4096 10-08 14:01 ..
```

b返回值，是执行结果。

## 时间函数

格式	描述
函数名	说明
mktime( YYYY MM dd HH MM ss[ DST])	生成时间格式
strftime([format [, timestamp]])	格式化时间输出，将时间戳转为时间字符串 具体格式，见下表.
systeme()	得到时间戳,返回从1970年1月1日开始到当前时间(不计闰年)的整数秒数

## 建指定时间(mktime使用)

```
awk 'BEGIN{tstamp=mktime("2001 01 01 12 12 12");print strftime("%c",tstamp);}'
2001年01月01日 星期一 12时12分12秒
```

```
awk 'BEGIN{tstamp1=mktime("2001 01 01 12 12 12");tstamp2=mktime("2001 02 01 0 0 0");print
tstamp2-tstamp1;}'
2634468
```

求2个时间段中间时间差, 介绍了strftime使用方法

```
awk 'BEGIN{tstamp1=mktime("2001 01 01 12 12 12");tstamp2=systemtime();print tstamp2-tstamp1;}'
308201392
```

## strftime日期和时间格式说明符

格式	描述
%a	星期几的缩写(Sun)
%A	星期几的完整写法(Sunday)
%b	月名的缩写(Oct)
%B	月名的完整写法(October)
%c	本地日期和时间
%d	十进制日期
%D	日期 08/20/99
%e	日期, 如果只有一位会补上一个空格
%H	用十进制表示24小时格式的小时
%I	用十进制表示12小时格式的小时
%j	从1月1日起一年中的第几天
%m	十进制表示的月份
%M	十进制表示的分钟
%p	12小时表示法(AM/PM)
%S	十进制表示的秒

格式	描述
%U	十进制表示的一年中的第几个星期(星期天作为一个星期的开始)
%w	十进制表示的星期几(星期天是0)
%W	十进制表示的一年中的第几个星期(星期一作为一个星期的开始)
%x	重新设置本地日期(08/20/99)
%X	重新设置本地时间(12: 00: 00)
%y	两位数字表示的年(99)
%Y	当前月份
%Z	时区(PDT)
%%	百分号(%)

---

## 相关命令

---

talk    whoami    mesg    sum    mattrib    cksum    dircolors    bc    ls – List    users

wall    yes    man    ngrep    vdfuse    gpm    rm – Remove    clockdiff    cal

Decompression Bomb

命令直达(输入完整命令)

[进入](#)

模糊搜索(输入关键词)

[搜索](#)

---

## Linux下载

---

[Ubuntu下载](#)

[CentOS下载](#)

[返回顶部 ↑](#)



[Linux命令大全](#)   [关于/联系](#)   [收藏本站请使用Ctrl+D](#)   [Shell脚本攻略](#)   [Shell正则表达式](#)   [网站地图](#)   [共收录](#)

**597** 条Linux系统命令

在Linux命令大全 ([man.linuxde.net](http://man.linuxde.net)) 可以查询您所需要的Linux命令教程和相关实例。如果您觉得本站内容对您有所帮助, 请推荐给更多需要帮助的人。