讲讲awk：awk从入门到精通

我：骏马金龙

博客：骏马金龙www.junmajinlong.com

Shell技术交流QQ群：921383787

课程介绍

awk是一个文本处理工具

awk在运维工作的面试当中，出现频率非常高

课程内容	awk的掌握程度	
1.awk的基本用法：80%	30%	入门阶段
2.awk的语法细节	80%	大成阶段
3.大量原理分析、工作机制分析，awk示例、技巧	90%	精通阶段
	100%	大师阶段

awk基本用法

测试文件a.txt：

```
1   ID  name    gender  age  email          phone
2   1   Bob     male    28   abc@qq.com     18023394012
3   2   Alice   female  24   def@gmail.com  18084925203
4   3   Tony    male    21   aaa@163.com    17048792503
5   4   Kevin   male    21   bbb@189.com    17023929033
6   5   Alex    male    18   ccc@xyz.com    18185904230
7   6   Andy    female  22   ddd@139.com    18923902352
8   7   Jerry   female  25   exdsa@189.com  18785234906
9   8   Peter   male    20   bax@qq.com     17729348758
10  9   Steven  female  23   bc@sohu.com    15947893212
11  10  Bruce   female  27   bcbd@139.com   13942943905
```

铺垫：读取文件的几种方式

1. 按字符数量读取：每一次可以读取一个字符，或者多个字符，直到把整个文件读取完
   - `while read -n 1 char;do echo $char;done <a.txt`
2. 按照分隔符进行读取：一直读取直到遇到了分隔符才停止，下次继续从分隔的位置处向后读取，直到读完整个文件
   - `while read -d "m" chars;do echo "$chars";done <a.txt`
3. 按行读取：每次读取一行，直到把整个文件读完
   - 是按照分隔符读取的一种特殊情况：将分隔符指定为了换行符`\n`
   - `while read line;do echo "$line";done <a.txt`
4. 一次性读取整个文件
   - 是按字符数量读取的特殊情况，也是按分隔符读取的特殊情况
   - `read -N 10000000 data <a.txt`

    - `read -d '_' data <a.txt`
5. 按字节数量读取

awk用法入门

```
awk 'awk_program' a.txt
```

- a.txt是awk要读取的文件，可以是0个文件或一个文件，也可以多个文件
    - 如果不给定任何文件，但又需要读取文件，则表示从标准输入中读取
- **单引号**包围的是awk代码，也称为awk程序
    - 尽量使用单引号，因为在awk中经常使用`$`符号，而$符号在Shell是变量符号，如果使用双引号包围awk代码，则$符号会被Shell解析成Shell变量，然后进行Shell变量替换。使用单引号包围awk代码，则`$`会脱离Shell的魔掌，使得$符号留给了awk去解析
- awk程序中，大量使用大括号，大括号表示代码块，代码块中间可以之间连用，代码块内部的多个语句需使用分号";"分隔

```
awk '{print $0}' a.txt
awk '{print $0}{print $0;print $0}' a.txt
```

BEGIN和END语句块

```
awk 'BEGIN{print "我在前面"}{print $0}' a.txt
awk 'END{print "我在后面"}{print $0}' a.txt
awk 'BEGIN{print "我在前面"}{print $0}END{print "我在后面"}' a.txt
```

BEGIN代码块：

- 在读取文件之前执行，且执行一次
- 在BEGIN代码块中，无法使用`$0`或其它一些特殊变量

END代码块：

- 在读取文件完成之后执行，且执行一次
- 有END代码块，必有要读取的数据(可以是标准输入)
- END代码块中可以使用$0等一些特殊变量，只不过这些特殊变量保存的是最后一轮awk循环的数据

main代码块：

- 读取文件时循环执行，(默认情况)每读取一行，就执行一次main代码块
- main代码块可有多个

安装新版本gawk

```
# 1.下载
wget --no-check-certificate https://mirrors.tuna.tsinghua.edu.cn/gnu/gawk/gawk-4.2.0.tar.gz

# 2.解压、进入解压后目录
```

```
tar xf gawk-4.2.0.tar.gz
cd gawk-4.2.0/

# 3.编译
./configure --prefix=/usr/local/gawk4.2 && make && make install

# 4.创建一个软链接：让awk指向刚新装的gawk版本
ln -fs /usr/local/gawk4.2/bin/gawk /usr/bin/awk

# 此时，调用awk将调用新版本的gawk，调用gawk将调用旧版本的gawk
awk --version
gawk --version
```

👇系统性深入awk👇

awk命令行结构和语法结构

在Shell命令行当中，双短横线 `--` 表示选项到此结束，后面的都是命令的参数。

```
awk [ -- ] program-text file ...
awk -f program-file [ -- ] file ...
awk -e program-text [ -- ] file ...

cmd -x -r root -ppassword a.txt b.txt c.txt
# 1.选项分为长选项和短选项
# 2.选项分为3种：
#    (1).不带参数的选项
#    (2).是带参数的选项，如果该选项后面没有给参数，则报错
#    (3).参数可选的选项，选项后面可以跟参数，也可以不跟参数
#        参数可选选项，如果要接参数，则必须将参数紧紧跟在选项后面，不能使用空格分隔选项和参数

# 3.两种参数：
#    (1).选项型参数
#    (2).非选项型参数
```

awk的语法充斥着 `pattern{action}` 的模式，它们称为awk rule：

- `awk 'BEGIN{n=3} /^[0-9]/{$1>5{$1=333;print $1} /Alice/{print "Alice"} END{print "hello"}' a.txt`
- pattern部分用于测试筛选数据，action表示在测试通过后执行的操作
- pattern和action都可以省略
  - 省略 `pattern` ，等价于对每一行数据都执行action
    - `awk '{print $0}' a.txt`
  - 省略代码块 `{action}` ，等价于 `{print}` 即输出所有行
    - `awk '/Alice/' a.txt` 等价于 `awk '/Alice/{print $0}' a.txt`
  - 省略代码块中的 `action` ，表示对筛选的行什么都不做
    - `awk '/Alice/{}' a.txt`
  - `pattern{action}` 任何一部分都可以省略

    - `awk '' a.txt`
- 多个`pattern{action}`可以直接连接连用
- action中多个语句连用需使用分号分隔

pattern和action

对于`pattern{action}`语句结构(都称之为语句块)，其中的pattern部分可以使用下面列出的模式：

```
# 特殊pattern
BEGIN
END

# 布尔代码块
/regular expression/    # 正则匹配成功与否 /a.*ef/{action}
relational expression   # 即等值比较、大小比较 3>2{action}
pattern && pattern      # 逻辑与 3>2 && 3>1 {action}
pattern || pattern      # 逻辑或 3>2 || 3<1 {action}
! pattern               # 逻辑取反 !/a.*ef/{action}
(pattern)               # 改变优先级
pattern ? pattern : pattern  # 三目运算符决定的布尔值

# 范围pattern，非布尔代码块
pattern1, pattern2      # 范围，pat1打开、pat2关闭，即flip,flop模式
```

action部分，可以是任何语句，例如print语句。

awk读取文件

详细分析awk如何读取文件

awk读取输入文件时，每次读取一条记录(record)(默认情况下按行读取，所以此时记录就是行)。每读取一条记录，将其保存到`$0`中，然后执行一次main代码段。

```
awk '{print $0}' a.txt
```

如果是空文件，则因为无法读取到任何一条记录，将导致直接关闭文件，而不会进入main代码段。

```
touch x.log  # 创建一个空文件
awk '{print "hello world"}' x.log
```

可设置表示输入记录分隔符的预定义变量RS(Record Separator)来改变每次读取的记录模式。

```
# RS="\n" 、 RS="m"
awk 'BEGIN{RS="\n"}{print $0}' a.txt
awk 'BEGIN{RS="m"}{print $0}' a.txt
```

RS通常设置在BEGIN代码块中，因为要先于读取文件就确定好RS分隔符。

> RS指定输入记录分隔符时，所读取的记录中是不包含分隔符字符的。例如 `RS="a"` ，则 `$0` 中一定不可能出现字符a。

RS两种可能情况：

- RS为单个字符：直接使用该字符来分割记录
- RS为多个字符：将其当做正则表达式，只要匹配正则表达式的符号，都用来分割记录
  - 设置预定义变量IGNORECASE为非零值，正则匹配时表示忽略大小写
  - 兼容模式下，只有首字符才生效，不会使用正则模式去分割记录

特殊的RS值用来解决特殊读取需求：

- `RS=""` ：按段落读取
- `RS="\0"` ：一次性读取所有数据，但有些特殊文件中包含了空字符 `\0`
- `RS="^$"` ：真正的一次性读取所有数据，因为非空文件不可能匹配成功
- `RS="\n+"` ：按行读取，但忽略所有空行

```
# 按段落读取：RS=''
$ awk 'BEGIN{RS=''}{print $0"------"}' a.txt

# 一次性读取所有数据：RS='\0' RS="^$"
$ awk 'BEGIN{RS='\0'}{print $0"------"}' a.txt
$ awk 'BEGIN{RS='^$'}{print $0"------"}' a.txt

# 忽略空行：RS='\n+'
$ awk 'BEGIN{RS='\n+'}{print $0"------"}' a.txt

# 忽略大小写：预定义变量IGNORECASE设置为非0值
$ awk 'BEGIN{IGNORECASE=1}{print $0"------"}' RS='[ab]' a.txt
```

> 预定义变量RT：
>
> 在awk每次读完一条记录时，会设置一个称为RT的预定义变量，表示Record Termination。
>
> 当RS为单个字符时，RT的值和RS的值是相同的。
>
> 当RS为多个字符(正则表达式)时，则RT设置为正则匹配到记录分隔符之后，真正用于划分记录时的字符。
>
> 当无法匹配到记录分隔符时，RT设置为控制空字符串(即默认的初始值)。
>
> `awk 'BEGIN{RS="(fe)?male"}{print RT}' a.txt`

两种行号：NR和FNR

在读取每条记录之后，将其赋值给$0，同时还会设置NR、FNR、RT。

- NR：所有文件的行号计数器
- FNR：是各个文件的行号计数器

```
awk '{print NR}' a.txt a.txt
awk '{print FNR}' a.txt a.txt
```

详细分析字段分割

awk读取每一条记录之后，会将其赋值给 `$0` ，同时还会对这条记录按照**预定义变量FS**划分字段，将划分好的各个字段分别赋值给 `$1 $2 $3 $4...$N` ，同时将划分的字段数量赋值给**预定义变量NF**。

引用字段的方式

`$N` 引用字段：

- `N=0` ：即 `$0` ，引用记录本身
- `0<N<=NF` ：引用对应字段
- `N>NF` ：表示引用不存在的字段，返回空字符串
- `N<0` ：报错

可使用变量或计算的方式指定要获取的字段序号。

```
awk '{n = 5;print $n}' a.txt
awk '{print $(2+2)}' a.txt  # 括号必不可少，用于改变优先级
awk '{print $(NF-3)}' a.txt
```

分割字段的方式

读取record之后，将使用预定义变量FS、FIELDWIDTHS或FPAT中的一种来分割字段。分割完成之后，再进入main代码段(所以，在main中设置FS对本次已经读取的record是没有影响的，但会影响下次读取)。

FS或-F

`FS` 或者 `-F` ：字段分隔符

- FS为单个字符时，该字符即为字段分隔符
- FS为多个字符时，则采用正则表达式模式作为字段分隔符
- 特殊的，也是FS默认的情况，FS为单个空格时，将以连续的空白（空格、制表符、换行符）作为字段分隔符
- 特殊的，FS为空字符串""时，将对每个字符都进行分隔，即每个字符都作为一个字段
- 设置预定义变量IGNORECASE为非零值，正则匹配时表示忽略大小写(只影响正则，所以FS为单字时无影响)
- 如果record中无法找到FS指定的分隔符(例如将FS设置为"\n")，则整个记录作为一个字段，即 `$1` 和 `$0` 相等

FIELDWIDTHS

指定预定义变量FIELDWIDTHS按字符宽度分割字段，这是gawk提供的高级功能。在处理某字段缺失时非常好用。

用法：

- `FIELDWIDTHS="3 5 6 9"` 表示第一个字段3字符，第二字段5字符...

- `FIELDWIDTHS = "8 1:5 6 2:33"`表示：
  - 第一个字段读8个字符
  - 然后跳过1个字符再读5个字符作为第二个字段
  - 然后读6个字符作为第三个字段
  - 然后跳过2个字符在读33个字符作为第四个字段(如果不足33个字符，则读到结尾)
- `FIELDWIDTHS="2 3 *"`：
  - 第一个字段2个字符
  - 第二个字段3个字符
  - 第三个字段剩余所有字符
  - 星号只能放在最后，且只能单独使用，表示剩余所有
- 设置该变量后，FS失效
- 之后再设置FS或FPAT，该变量将失效

示例1：

```
# 没取完的字符串DDD被丢弃，且NF=3
$ awk 'BEGIN{FIELDWIDTHS="2 3 2"}{print $1,$2,$3,$4}' <<<"AABBBCCDDDD"
AA BBB CC

# 字符串不够长度时无视
$ awk 'BEGIN{FIELDWIDTHS="2 3 2 100"}{print $1,$2,$3,$4"-"}' <<<"AABBBCCDDDD"
AA BBB CC DDDD-

# *号取剩余所有，NF=3
$ awk 'BEGIN{FIELDWIDTHS="2 3 *"}{print $1,$2,$3}' <<<"AABBBCCDDDD"
AA BBB CCDDDD

# 字段数多了，则取完字符串即可，NF=2
$ awk 'BEGIN{FIELDWIDTHS="2 30 *"}{print $1,$2,NF}' <<<"AABBBCCDDDD"
AA BBBCCDDDD 2
```

示例2：处理某些字段缺失的数据。

如果按照常规的FS进行字段分割，则对于缺失字段的行和没有缺失字段的行很难统一处理，但使用FIELDWIDTHS则非常方便。

假设a.txt文本内容如下：

```
ID  name    gender  age  email          phone
1   Bob     male    28   abc@qq.com     18023394012
2   Alice   female  24   def@gmail.com  18084925203
3   Tony    male    21   aaa@163.com    17048792503
4   Kevin   male    21   bbb@189.com    17023929033
5   Alex    male    18                  18185904230
6   Andy    female  22   ddd@139.com    18923902352
7   Jerry   female  25   exdsa@189.com  18785234906
8   Peter   male    20   bax@qq.com     17729348758
9   Steven  female  23   bc@sohu.com    15947893212
10  Bruce   female  27   bcbd@139.com   13942943905
```

因为email字段有的是空字段，所以直接用FS划分字段不便处理。可使用FIELDWIDTHS。

```
# 字段1：4字符
# 字段2：8字符
# 字段3：8字符
# 字段4：2字符
# 字段5：先跳过3字符，再读13字符，该字段13字符
# 字段6：先跳过2字符，再读11字符，该字段11字符
awk '
BEGIN{FIELDWIDTHS="4 8 8 2 3:13 2:11"}
NR>1{
    print "<"$1">","<"$2">","<"$3">","<"$4">","<"$5">","<"$6">"
}' a.txt

# 如果email为空，则输出它
awk '
BEGIN{FIELDWIDTHS="4 8 8 2 3:13 2:11"}
NR>1{
    if($5 ~ /^ +$/){print $0}
}' a.txt
```

FPAT

FS是指定字段分隔符，来取得除分隔符外的部分作为字段。

FPAT是取得匹配的字符部分作为字段。它是gawk提供的一个高级功能。

FPAT根据指定的正则来全局匹配record，然后将所有匹配成功的部分组成 `$1、$2...` ，不会修改 `$0` 。

- `awk 'BEGIN{FPAT="[0-9]+"}{print $3"-"}' a.txt`
- 之后再设置FS或FPAT，该变量将失效

FPAT常用于字段中包含了字段分隔符的场景。例如，CSV文件中的一行数据如下：

```
Robbins,Arnold,"1234 A Pretty Street, NE",MyTown,MyState,12345-6789,USA
```

其中逗号分隔每个字段，但双引号包围的是一个字段整体，即使其中有逗号。

这时使用FPAT来划分各字段比使用FS要方便的多。

```
echo 'Robbins,Arnold,"1234 A Pretty Street, NE",MyTown,MyState,12345-6789,USA' |\
awk '
    BEGIN{FPAT="[^,]*|(\"[^\"]*\")"}
    {
        for (i=1;i<NF;i++){
            print "<"$i">"
        }
    }
'
```

最后，patsplit()函数和FPAT的功能一样。

检查字段分隔的方式

有FS、FIELDWIDTHS、FPAT三种获取字段的方式，可使用 `PROCINFO` 数组来确定本次使用何种方式获得字段。

PROCINFO是一个数组，记录了awk进程工作时的状态信息。

如果：

- `PROCINFO["FS"]=="FS"`，表示使用FS分割获取字段
- `PROCINFO["FPAT"]=="FPAT"`，表示使用FPAT匹配获取字段
- `PROCINFO["FIELDWIDTHS"]=="FIELDWIDTHS"`，表示使用FIELDWIDTHS分割获取字段

例如：

```
if(PROCINFO["FS"]=="FS"){
    ...FS spliting...
} else if(PROCINFO["FPAT"]=="FPAT"){
    ...FPAT spliting...
} else if(PROCINFO["FIELDWIDTHS"]=="FIELDWIDTHS"){
    ...FIELDWIDTHS spliting...
}
```

修改字段或NF值的联动效应

注意下面的分割和计算两词：分割表示使用FS（field Separator），计算表示使用预定义变量OFS（Output Field Separator）。

1. 修改`$0`，将使用`FS`重新分割字段，所以会影响`$1、$2...`
2. 修改`$1、$2`，将根据`$1`到`$NF`来重新计算`$0`
    - 即使是`$1 = $1`这样的原值不变的修改，也一样会重新计算`$0`
3. 为不存在的字段赋值，将新增字段并按需使用空字符串填充中间的字段，并使用`OFS`重新计算`$0`
    - `awk '{$(NF+2)=5;print $0}' OFS='-' a.txt`
4. 增加NF值，将使用空字符串新增字段，并使用`OFS`重新计算`$0`
    - `awk '{NF+=3;print $0}' OFS='-' a.txt`
5. 减小NF值，将丢弃一定数量的尾部字段，并使用`OFS`重新计算`$0`
    - `awk '{NF-=3;print $0}' OFS='-' a.txt`

关于$0

当读取一条record之后，将原原本本地被保存到`$0`当中。

```
awk '{print $0}' a.txt
```

但是，只要出现了上面所说的任何一种导致`$0`重新计算的操作，都会立即使用OFS去重建`$0`。

换句话说，没有导致`$0`重建，$0就一直是原原本本的数据，所以指定OFS也无效。

```
awk '{print $0}' OFS="-" a.txt  # OFS此处无效
```

当`$0`重建后，将自动使用OFS重建，所以即使没有指定OFS，它也会采用默认值(空格)进行重建。

```
awk '{$1=$1;print $0}'  a.txt  # 输出时将以空格分隔各字段
awk '{print $0;$1=$1;print $0}' OFS="-" a.txt
```

如果重建`$0`之后，再去修改OFS，将对当前行无效，但对之后的行有效。所以如果也要对当前行生效，需要再次重建。

```
# OFS对第一行无效
awk '{$4+=10;OFS="-";print $0}' a.txt

# 对所有行有效
awk '{$4+=10;OFS="-";$1=$1;print $0}' a.txt
```

关注`$0`重建是一个非常有用的技巧。

例如，下面通过重建`$0`的技巧来实现去除行首行尾空格并压缩中间空格：

```
$ echo "   a  b  c   d   " | awk '{$1=$1;print}'
a b c d
$ echo "     a   b  c   d   " | awk '{$1=$1;print}' OFS="-"
a-b-c-d
```

awk数据筛选示例

筛选行

```
# 1.根据行号筛选
awk 'NR==2' a.txt   # 筛选出第二行
awk 'NR>=2' a.txt   # 输出第2行和之后的行

# 2.根据正则表达式筛选整行
awk '/qq.com/' a.txt       # 输出带有qq.com的行
awk '$0 ~ /qq.com/' a.txt  # 等价于上面命令
awk '/^[^@]+$/' a.txt      # 输出不包含@符号的行
awk '!/@/' a.txt           # 输出不包含@符号的行

# 3.根据字段来筛选行
awk '($4+0) > 24{print $0}' a.txt  # 输出第4字段大于24的行
awk '$5 ~ /qq.com/' a.txt   # 输出第5字段包含qq.com的行

# 4.将多个筛选条件结合起来进行筛选
awk 'NR>=2 && NR<=7' a.txt
awk '$3=="male" && $6 ~ /^170/' a.txt
awk '$3=="male" || $6 ~ /^170/' a.txt

# 5.按照范围进行筛选 flip flop
# pattern1,pattern2{action}
awk 'NR==2,NR==7' a.txt        # 输出第2到第7行
awk 'NR==2,$6 ~ /^170/' a.txt
```

处理字段

修改字段时，一定要注意，可能带来的联动效应：即使用OFS重建$0。

```
awk 'NR>1{$4=$4+5;print $0}' a.txt
awk 'NR>1{$6=$6"*";print $0}' a.txt
```

awk运维面试试题

从ifconfig命令的结果中筛选出除了lo网卡外的所有IPv4地址。

```
# 1.法一:
ifconfig | awk '/inet / && !($2 ~ /^127/){print $2}'

# 2.法二:
ifconfig | awk 'BEGIN{RS=""}!/lo/{print $6}'

# 3.法三:
ifconfig | awk 'BEGIN{RS="";FS="\n"}!/lo/{$0=$2;FS=" ";$0=$0;print $2}'
```

awk工作流程

参考自: `man awk`的"AWK PROGRAM EXECUTION"段。

```
man --pager='less -p ^"AWK PROGRAM EXECUTION"' awk
```

执行步骤:

1. 解析`-v var=val...`选项中的变量赋值
2. 编译awk源代码为awk可解释的内部格式，包括-v的变量
3. 执行BEGIN代码段
4. 根据输入记录分隔符RS读取文件（根据ARGV数组的元素决定要读取的文件），如果没有指定文件，则从标准输入中读取文件，同时执行main代码段
    - 如果文件名部分指定为`var=val`格式，则声明并创建变量，此阶段的变量在BEGIN之后声明，所以BEGIN中不可用，main代码段可用
    - 每读取一条记录:
        - 都将设置NR、FNR、RT、$0等变量
        - (默认)根据输入字段分隔符FS切割字段，将各字段保存到`$1、$2...`中
        - 测试main代码段的pattern部分，如果测试成功则执行action部分
5. 执行END代码段

getline用法详解

除了可以从标准输入或非选项型参数所指定的文件中读取数据，还可以使用getline从其它各种渠道获取需要处理的数据，它的用法有很多种。

getline的返回值:

- 如果可以读取到数据，返回1
- 如果遇到了EOF，返回0
- 如果遇到了错误，返回负数。如-1表示文件无法打开，-2表示IO操作需要重试(retry)。在遇到错误的同时，还会设置`ERRNO`变量来描述错误

为了健壮性，getline时强烈建议进行判断。例如：

```
if( (getline) <= 0 ){...}
if((getline) < 0){...}
if((getline) > 0){...}
```

上面的getline的括号尽量加上，因为 `getline < 0` 表示的是输入重定向，而不是和数值0进行小于号的比较。

无参数的getline

getline无参数时，表示立即读取下一条记录保存到 `$0` 中，并进行字段分割，然后**继续执行后续代码逻辑**。

此时的getline会设置NF、RT、NR、FNR、$0和$N。

next也可以读取下一行。

- getline：读取下一行之后，继续执行getline后面的代码
- next：读取下一行，立即回头awk循环的头部，不会再执行next后面的代码

它们之间的区别用伪代码描述，类似于：

```
# next
exec 9<> filename
while read -u 9 line;do
  ...code...
  continue  # next
  ...code...  # 这部分代码在本轮循环当中不再执行
done

# getline
while read -u 9 line;do
  ...code...
  read -u 9 line  # getline
  ...code...
done
```

例如，匹配到某行之后，再读一行就退出：

```
awk '/^1/{print;getline;print;exit}' a.txt
```

为了更健壮，应当对getline的返回值进行判断。

```
awk '/^1/{print;if((getline)<=0){exit};print}' a.txt
```

一个参数的getline

没有参数的getline是读取下一条记录之后将记录保存到 `$0` 中，并对该记录进行字段的分割。

一个参数的getline是将读取的记录保存到指定的变量当中，并且不会对其进行分割。

```
getline var
```

此时的getline只会设置RT、NR、FNR变量和指定的变量var。因此$0和$N以及NF保持不变。

```
awk '
/^1/{
  if((getline var)<=0){exit}
  print var
  print $0"--"$2
}' a.txt
```

从指定文件中读取数据

- `getline < filename`：从指定文件filename中读取一条记录并保存到`$0`中
  - 会进行字段的划分，会设置变量`$0 $N NF`，不会设置变量`NR FNR`
- `getline var < filename`：从指定文件filename中读取一条记录并保存到指定变量var中
  - 不会划分字段，不会设置变量`NR FNR NF $0 $N`

filename需使用双引号包围表示文件名字符串，否则会当作变量解析`getline < "c.txt"`。此外，如果路径是使用变量构建的，则应该使用括号包围路径部分。例如`getline < dir "/" filename`中使用了两个变量构建路径，这会产生歧义，应当写成`getline <(dir "/" filename)`。

注意，每次从filename读取之后都会做好位置偏移标记，下次再从该文件读取时将根据这个位置标记继续向后读取。

例如，每次行首以1开头时就读取c.txt文件的所有行。

```
awk '
  /^1/{
    print;
    while((getline < "c.txt")>0){print};
    close("c.txt")
}' a.txt
```

上面的`close("c.txt")`表示在`while(getline)`读取完文件之后关掉，以便后面再次读取，如果不关掉，则文件偏移指针将一直在文件结尾处，使得下次读取时直接遇到EOF。

从Shell命令输出结果中读取数据

- `cmd | getline`：从Shell命令cmd的输出结果中读取一条记录保存到`$0`中
  - 会进行字段划分，设置变量`$0 NF $N RT`，不会修改变量`NR FNR`
- `cmd | getline var`:从Shell命令cmd的输出结果中读取数据保存到var中
  - 除了var和RT，其它变量都不会设置

如果要再次执行cmd并读取其输出数据，则需要close关闭该命令。例如`close("seq 1 5")`，参见下面的示例。

例如：每次遇到以1开头的行都输出seq命令产生的`1 2 3 4 5`。

```
awk '/^1/{print;while(("seq 1 5"|getline)>0){print};close("seq 1 5")}' a.txt
```

再例如，调用Shell的date命令生成时间，然后保存到awk变量cur_date中：

```
awk '
  /^1/{
    print
    "date +\"%F %T\""|getline cur_date
    print cur_date
    close("date +\"%F %T\"")
}' a.txt
```

可以将cmd保存成一个字符串变量。

```
awk '
  BEGIN{get_date="date +\"%F %T\""}
  /^1/{
    print
    get_date | getline cur_date
    print cur_date
    close(get_date)
}' a.txt
```

更为复杂一点的，cmd中可以包含Shell的其它特殊字符，例如管道、重定向符号等：

```
awk '
  /^1/{
    print
    if(("seq 1 5 | xargs -i echo x{}y 2>/dev/null"|getline) > 0){
      print
    }
    close("seq 1 5 | xargs -i echo x{}y 2>/dev/null")
}' a.txt
```

awk中的coprocess

awk虽然强大，但是有些数据仍然不方便处理，这时可将数据交给Shell命令去帮助处理，然后再从Shell命令的执行结果中取回处理后的数据继续awk处理。

awk通过 `|&` 符号来支持coproc。

```
awk_print[f] "something" |& Shell_Cmd
Shell_Cmd |& getline [var]
```

这表示awk通过print输出的数据将传递给Shell的命令Shell_Cmd去执行，然后awk再从Shell_Cmd的执行结果中取回Shell_Cmd产生的数据。

例如，不想使用awk的substr()来取子串，而是使用sed命令来替换。

```
awk '
    BEGIN{
      CMD="sed -nr \"s/.*@(.*)$/\\1/p\"";
    }

    NR>1{
        print $5;
        print $5 |& CMD;
        close(CMD,"to");
        CMD |& getline email_domain;
        close(CMD);
        print email_domain;
}' a.txt
```

对于 `awk_print |& cmd; cmd |& getline` 的使用，须注意的是：

- `awk_print |& cmd` 会直接将数据写进管道，cmd可以从中获取数据
- 强烈建议在awk_print写完数据之后加上 `close(cmd,"to")` ，这样表示向管道中写入一个EOF标记，避免某些要求读完所有数据再执行的cmd命令被永久阻塞
- 如果cmd是按块缓冲的，则getline可能会陷入阻塞。这时可将cmd部分改写成 `stdbuf -oL cmd` 以强制其按行缓冲输出数据
    - `CMD="stdbuf -oL cmdline";awk_print |& CMD;close(CMD,"to");CMD |& getline`

对于那些要求读完所有数据再执行的命令，例如sort命令，它们有可能需要等待数据已经完成后（遇到EOF标记）才开始执行任务，对于这些命令，可以多次向coprocess中写入数据，最后 `close(CMD,"to")` 让coprocess运行起来。

例如，对age字段(即 `$4`)使用sort命令按数值大小进行排序：

```
awk '
    BEGIN{
      CMD="sort -k4n";
    }

    # 将所有行都写进管道
    NR>1{
      print $0 |& CMD;
    }

    END{
      close(CMD,"to");  # 关闭管道通知sort开始排序
      while((CMD |& getline)>0){
        print;
      }
      close(CMD);
} ' a.txt
```

close()

```
close(filename)
close(cmd,[from | to])  # to参数只用于coprocess的第一个阶段
```

如果close()关闭的对象不存在，awk不会报错，仅仅只是让其返回一个负数返回值。

close()有两个基本作用：

- 关闭文件，丢弃已有的文件偏移指针
  - 下次再读取文件，将只能重新打开文件，重新打开文件会从文件的最开头处开始读取
- 发送EOF标记

awk中任何文件都只会在第一次使用时打开，之后都不会再重新打开。只有关闭之后，再使用才会重新打开。

例如一个需求是只要在a.txt中匹配到1开头的行就输出另一个文件x.log的所有内容，那么在第一次输出x.log文件内容之后，文件偏移指针将在x.log文件的结尾处，如果不关闭该文件，则后续所有读取x.log的文件操作都从结尾处继续读取，但是显然总是得到EOF异常，所以getline返回值为0，而且也读取不到任何数据。所以，必须关闭它才能在下次匹配成功时再次从头读取该文件。

```
awk '
  /^1/{
    print;
    while((getline var <"x.log")>0){
      print var
    }
    close("x.log")
}' a.txt
```

在处理Coprocess的时候，close()可以指定第二个参数"from"或"to"，它们都针对于coproc而言，from时表示关闭 `coproc |& getline` 的管道，使用to时，表示关闭 `print something |& coproc` 的管道。

```
awk '
BEGIN{
  CMD="sed -nr \"s/.*@(.*)$/\\1/p\"";
}
NR>1{
    print $5;
    print $5 |& CMD;
    close(CMD,"to");   # 本次close()是必须的
    CMD |& getline email_domain;
    close(CMD);
    print email_domain;
}' a.txt
```

上面的第一个close是必须的，否则sed会一直阻塞。因为sed一直认为还有数据可读，只有关闭管道发送一个EOF，sed才会开始处理。

执行Shell命令system()

多数时候，使用awk的 `print cmd | "sh"` 即可实现调用shell命令的功能。

但也可以使用system()函数来直接执行一个Shell命令，system()的返回值是命令的退出状态码。

```
$ awk 'BEGIN{system("date +\"%s.%N\"")}'
1572328598.653524342

$ awk 'BEGIN{system("date +\"%s.%N\" >/dev/null")}'

$ awk 'BEGIN{system("date +\"%s.%N\" | cat")}'
1572328631.308807331
```

system()在开始运行之前会flush gawk的缓冲。特别的，空字符串参数的 `system("")` ，它会被gawk特殊对待，它不会去启动一个shell来执行空命令，而是仅执行flush操作。

关于flush的行为，参考 **flush**。

输出操作

awk可以通过print、printf将数据输出到标准输出或重定向到文件。

print

```
print elem1,elem2,elem3...
print(elem1,elem2,elem3...)
```

逗号分隔要打印的字段列表，各字段都**会自动转换成字符串格式**，然后通过预定义变量OFS(output field separator)的值(其默认值为空格)连接各字段进行输出。

```
$ awk 'BEGIN{print "hello","world"}'
hello world
$ awk 'BEGIN{OFS="-";print "hello","world"}'
hello-world
```

print要输出的数据称为输出记录，在print输出时会自动在尾部加上输出记录分隔符，输出记录分隔符的预定义变量为ORS，其默认值为 `\n` 。

```
$ awk 'BEGIN{OFS="-";ORS="_\n";print "hello","world"}'
hello-world_
```

括号可省略，但如果要打印的元素中包含了特殊符号 `>` ，则必须使用括号包围(如 `print("a" > "A")`)，因为它是输出重定向符号。

如果省略参数，即 `print;` 等价于 `print $0;` 。

print输出数值

print在输出数据时，总是会先转换成字符串再输出。

对于数值而言，可以自定义转换成字符串的格式，例如使用sprintf()进行格式化。

print在自动转换数值（专指小数）为字符串的时候，采用预定义变量OFMT（Output format）定义的格式按照sprintf()相同的方式进行格式化。OFMT默认值为 `%.6g` ，表示有效位(整数部分加小数部分)最多为6。

```
$ awk 'BEGIN{print 3.12432623}'
3.12433
```

可以修改OFMT，来自定义数值转换为字符串时的格式：

```
$ awk 'BEGIN{OFMT="%.2f";print 3.99989}'
4.00

# 格式化为整数
$ awk 'BEGIN{OFMT="%d";print 3.99989}'
3
$ awk 'BEGIN{OFMT="%.0f";print 3.99989}'
4
```

printf

```
printf format, item1, item2, ...
```

格式化字符：

```
%c      将ASCII码转换为字符，例如printf "%c",65将输出A
%d, %i  转换为整数，直接截断而不会四舍五入，例如printf "%d",23.9输出23
%e, %E  科学计数法方式输出数值
%f, %F  浮点数方式输出，会四舍五入，例如printf "%4.3f",123.4128输出123.413
%g, %G  输出为浮点数或科学计数法格式
%o      将数字识别为8进制，然后转换为10进制，再转换为字符串输出，例如printf "%o",8输出10
%s      输出字符串
%x, %X  将数字识别为16进制，然后转换为10进制，再转换为字符串输出，例如printf "%x",16输出10
%%      输出百分号%
```

修饰符：均放在格式化字符的前面

```
N$      N是正整数。默认情况下，printf的字段列表顺序和格式化字符
        串中的%号顺序是一一对应的，使用N$可以自行指定顺序。
        printf "%2$s %1$s","world","hello"输出hello world
        N$可以重复指定，例如"%1$s %1$s"将取两次第一个字段

宽度     指定该字段占用的字符数量，不足宽度默认使用空格填充，超出宽度将无视。
         printf "%5s","ni"输出"___ni"，下划线表示空格

-       表示左对齐。默认是右对齐的。
        printf "%5s","ni"输出"___ni"
        printf "%-5s","ni"输出"ni___"

空格     针对于数值。对于正数，在其前添加一个空格，对于负数，无视
        printf "% d,% d",3,-2输出"_3,-2"，下划线表示空格

+       针对于数值。对于正数，在其前添加一个+号，对于负数，无视
        printf "%+d,%+d",3,-2输出"+3,-2"，下划线表示空格

#       可变的数值前缀。对于%o，将添加前缀0，对于%x或%X，将添加前缀0x或0X
```

```
20
21   0       只对数值有效。使用0而非默认的空格填充在左边，对于左对齐的数值无效
22           printf "%05d","3"输出00003
23           printf "%-05d","3"输出3
24           printf "%05s",3输出____3
25
26   '       单引号，表示对数值加上千分位逗号，只对支持千分位表示的locale有效
27           $ awk "BEGIN{printf \"%'d\n\",123457890}"
28           123,457,890
29           $ LC_ALL=C awk "BEGIN{printf \"%'d\n\",123457890}"
30           123457890
31
32   .prec   指定精度。在不同格式化字符下，精度含义不同
33           %d,%i,%o,%u,%x,%X 的精度表示最大数字字符数量
34           %e,%E,%f,%F 的精度表示小数点后几位数
35           %s 的精度表示最长字符数量，printf "%.3s","foob"输出foo
36           %g,%G 的精度表示表示最大有效位数，即整数加小数位的总数量
```

sprintf()

sprintf()采用和printf相同的方式格式化字符串，但是它不会输出格式化后的字符串，而是返回格式化后的字符串。所以，可以将格式化后的字符串赋值给某个变量。

```
awk '
    BEGIN{
        a = sprintf("%03d", 12.34)
        print a  # 012
    }
'
```

重定向输出

```
print[f] something >"filename"
print[f] something >>"filename"
print[f] something | "Shell_Cmd"
print[f] something |& "Shell_Cmd_Coprocess"
```

`>filename`时，如果文件不存在，则创建，如果文件存在则首先截断。之后再输出到该文件时将不再截断。

> awk中只要不close()，任何文件都只会在第一次使用时打开，之后都不会再重新打开。

```
awk '{print $2 >"name.txt";print $4 >"name.txt"}' a.txt
```

`>>filename`时，将追加数据，文件不存在时则创建。

`print[f] something | Shell_Cmd`时，awk将创建一个管道，然后启动Shell命令，print[f]产生的数据放入管道，而命令将从管道中读取数据。

```
# 例1:
awk '
    NR>1{
      print $2 >"name.unsort"
      cmd = "sort >name.sort"
      print $2 | cmd
      #print $2 | "sort >name.sort"
    }
    END{close(cmd)}
' a.txt

# 例2: awk中构建Shell命令，通过管道交给shell执行
awk 'BEGIN{printf "seq 1 5" | "bash"}'
```

`print[f] something |& Shell_Cmd`时，print[f]产生的数据交给Coprocess。之后，awk再从Coprocess中取回数据。这里的`|&`有点类似于能够让Shell_Cmd后台异步运行的管道。

stdin、stdout、stderr

awk重定向时可以直接使用`/dev/stdin`、`/dev/stdout`和`/dev/stderr`。还可以直接使用某个已打开的文件描述符`/dev/fd/N`。

例如:

```
awk 'BEGIN{print "something OK" > "/dev/stdout"}'
awk 'BEGIN{print "something wrong" > "/dev/stderr"}'
awk 'BEGIN{print "something wrong" | "cat >&2"}'

awk 'BEGIN{getline < "/dev/stdin";print $0}'

$ exec 4<> a.txt
$ awk 'BEGIN{while((getline < "/dev/fd/4")>0){print $0}}'
```

gawk语法

变量

awk的变量是动态变量，在使用时声明。

所以awk变量有3种状态:

- 未声明状态：称为untyped类型
- 引用过但未赋值状态：unassigned类型
- 已赋值状态

引用未赋值的变量，其默认初始值为空字符串或数值0。

在awk中未声明的变量称为untyped，声明了但未赋值(只要引用了就声明了)的变量其类型为unassigned。

gawk 4.2版提供了 `typeof()` 函数，可以测试变量的数据类型，包括测试变量是否声明。

```
awk 'BEGIN{
  print(typeof(a))            # untyped
  if(b==0){print(typeof(b))}  # unassigned
}'
```

除了typeof()，还可以使用下面的技巧进行检测：

```
awk 'BEGIN{
  if(a=="" && a==0){    # 未赋值时，两个都true
    print "untyped or unassigned"
  } else {
      print "assigned"
  }
}'
```

变量赋值

awk中的变量赋值语句也可以看作是一个**有返回值**的表达式。

例如， `a=3` 赋值完成后返回3，同时变量a也被设置为3。

基于这个特点，有两点用法：

- 可以 `x=y=z=5` ，等价于 `z=5 y=5 x=5`
- 可以将赋值语句放在任意允许使用表达式的地方
  - `x != (y = 1)`
  - `awk 'BEGIN{print (a=4);print a}'`

问题： `a=1;arr[a+=2] = (a=a+6)` 是怎么赋值的，对应元素结果等于？ `arr[3]=7` 。但不要这么做，因为不同awk的赋值语句左右两边的评估顺序有可能不同。

awk中使用变量

1. 在BEGIN或main或END代码段中直接引用或赋值
2. 使用 `-v var=val` 选项，可定义多个，必须放在awk代码的前面
   - 它的变量声明早于BEGIN块
   - 普通变量： `awk -v age=123 'BEGIN{print age}'`
3. 在awk代码后面使用 `var=val` 参数
   - 它的变量声明在BEGIN之后
   - `awk '{print n}' n=3 a.txt n=4 b.txt`
   - `awk '{print $1}' FS=' ' a.txt FS=":" /etc/passwd`

引用Shell变量

三种方式：

```
# 1. -v选项
awk -v age=$age 'BEGIN{print age}'

# 2. 非选项型参数的变量赋值方式
awk '{print age}' age=$age a.txt

# 3.从单引号中脱离，直接暴露给Shell解析
awk '{print '$age'}' a.txt
awk '{print '"$age"'}' a.txt
```

数据类型

gawk有两种基本的数据类型：数值和字符串。在gawk 4.2.0版本中，还支持第三种基本的数据类型：正则表达式类型。

数据是什么类型在使用它的上下文中决定：**在字符串操作环境下将转换为字符串，在数值操作环境下将转换为数值。**

隐式转换：

- 算术加0操作可转换为数值类型
  - `"123" + 0` 返回数值123
  - `" 123abc" + 0` 转换为数值时为123
  - 无效字符串将转换成0，例如 `"abc"+3` 返回3
- 连接空字符串可转换为字符串类型
  - `123""` 转换为字符串"123"

```
awk 'BEGIN{a="123";print typeof(a+0)}' # number
awk 'BEGIN{a=123;print typeof(a"")}'   # string

awk 'BEGIN{a=2;b=3;print(a b)+4}' # 27
```

显式转换：

- 数值->字符串：
  - CONVFMT或sprintf()：功能等价。都是指定数值转换为字符串时的格式

```
awk 'BEGIN{a=123.4567;CONVFMT="%.2f";print a""}' #123.46
awk 'BEGIN{a=123.4567;print sprintf("%.2f", a)}' #123.46
awk 'BEGIN{a=123.4567;printf("%.2f",a)}'
```

- 字符串->数值：strtonum()

```
gawk 'BEGIN{a="123.4567";print strtonum(a)}' # 123.457
```

awk字面量

awk中有3种字面量：字符串字面量、数值字面量和正则表达式字面量。

数值字面量

- 整数、浮点数、科学计数
  - 105、105.0、1.05e+2、1050e-1
- awk内部总是使用浮点数方式保存所有数值，但用户在使用可以转换成整数的数值时总会去掉小数点
  - 数值12.0面向用户的值为12，12面向awk内部的值是12.0000000...0

```
# 结果是123而非123.0
awk 'BEGIN{a=123.0;print a}'
```

算术运算

```
++ --     自增、自减，支持i++和++i或--i或i--
^         幂运算(**也用于幂运算)
+ -       一元运算符(正负数符号)
* / %     乘除取模运算
+ -       加减法运算

# 注:
# 1.++和--既可以当作独立语句，也可以作为表达式，如:
#     awk 'BEGIN{a=3;a++;a=++a;print a}'
# 2.**或^幂运算是从右向左计算的: print 2**1**3得到2而不是8
```

赋值操作(优先级最低):

```
= += -= *= /= %= ^= **=
```

疑惑：`b = 6;print b += b++` 输出结果？可能是12或13。不同的awk的实现在评估顺序上不同，所以不要用这种可能产生歧义的语句。

字符串字面量

awk中的字符串都以双引号包围，不能以单引号包围。

- `"abc"`
- `""`
- `"\0"`、`"\n"`

字符串连接（串联）：awk没有为字符串的串联操作提供运算符，可以直接连接或使用空格连接。

```
awk 'BEGIN{print ("one" "two")}'  # "onetwo"
awk 'BEGIN{print ("one""two")}'
awk 'BEGIN{a="one";b="two";print (a b)}'
```

注意：字符串串联虽然方便，但是要考虑串联的优先级。例如下面的:

```
# 下面第一个串联成功，第二个串联失败，
# 因为串联优先级低于加减运算，等价于`12 (" " -23)`
# 即: 先转为数值0-23，再转为字符串12-23
$ awk 'BEGIN{a="one";b="two";print (12 " " 23)}'
12 23
$ awk 'BEGIN{a="one";b="two";print (12 " " -23)}'
12-23
```

正则表达式字面量

普通正则：

- `/[0-9]+/`
- 匹配方式：`"str" ~ /pattern/` 或 `"str" !~ /pattern/`
- 匹配结果返回值为0(匹配失败)或1(匹配成功)
- 任何单独出现的 `/pattern/` 都等价于 `$0 ~ /pattern/`
  - `if(/pattern/)` 等价于 `if($0 ~ /pattern/)`
  - 坑1：`a=/pattern/` 等价于将 `$0 ~ /pattern/` 的匹配返回值（0或1）赋值给a
  - 坑2：`/pattern/ ~ $1` 等价于 `$0 ~ /pattern/ ~ $1`，表示用 `$1` 去匹配0或1
  - 坑3：`/pattern/` 作为参数传给函数时，传递的是 `$0~/pat/` 的结果0或1
  - 坑4.坑5.坑6...

强类型的正则字面量(gawk 4.2.0才支持)

- `@/pattern/` 作为独立的一种数据类型：正则表达式类型
- 在使用正则字面量变量进行匹配的时候，不能简写 `a=@/Alice/;a{print}`，只能写完整的匹配 `a=@/Alice/;$0 ~ a{print}`
- 解决上面的坑
- 可使用 `typeof()`(也是4.2才支持的)检查类型，得到的结果将是 `regexp`
  - `awk 'BEGIN{re=@/abc/;print typeof(re)}'`

gawk支持的正则

```
.       # 匹配任意字符，包括换行符
^
$
[...]
[^...]
|
+
*
?
()
{m}
{m,}
{m,n}
{,n}

[:lower:]
[:upper:]
[:alpha:]
[:digit:]
[:alnum:]
[:xdigit:]
[:blank:]
[:space:]
```

```
[:punct:]
[:graph:]
[:print:]
[:cntrl:]

以下是gawk支持的:
\y    匹配单词左右边界部分的空字符位置 "hello world"
\B    和\y相反，匹配单词内部的空字符位置，例如"crate" ~ `/c\Brat\Be/`成功
\<    匹配单词左边界
\>    匹配单词右边界
\s    匹配空白字符
\S    匹配非空白字符
\w    匹配单词组成字符(大小写字母、数字、下划线)
\W    匹配非单词组成字符
\`    匹配字符串的绝对行首  "abc\ndef"
\'    匹配字符串的绝对行尾
```

gawk不支持正则修饰符，所以无法直接指定忽略大小写的匹配。

如果想要实现忽略大小写匹配，则可以将字符串先转换为大写、小写再进行匹配。或者设置预定义变量IGNORECASE为非0值。

```
# 转换为小写
awk 'tolower($0) ~ /bob/{print $0}' a.txt

# 设置IGNORECASE
awk '/BOB/{print $0}' IGNORECASE=1 a.txt
```

awk布尔值

在awk中，没有像其它语言一样专门提供true、false这样的关键字。

但它的布尔值逻辑非常简单:

- 数值0表示布尔假
- 空字符串表示布尔假
- 其余所有均为布尔真
    - 字符串"0"也是真，因为它是字符串
- awk中，正则匹配也有返回值，匹配成功则返回1，匹配失败则返回0
- awk中，所有的布尔运算也有返回值，布尔真返回值1，布尔假返回值为0

```
awk '
BEGIN{
    if(1){print "haha"}
    if("0"){print "hehe"}
    if(a=3){print "hoho"}  # if(3){print "hoho"}
    if(a==3){print "aoao"}
    if(/root/){print "heihei"}  # $0 ~ /root/
}'
```

awk中比较操作

strnum类型

awk最基本的数据类型只有string和number(gawk 4.2.0版本之后支持正则表达式类型)。但是，对于用户输入数据(例如从文件中读取的各个字段值)，它们理应属于string类型，但有时候它们看上去可能像是数值(例如`$2=37`)，而有时候有需要这些值是数值类型。

awk的数据来源：1.awk内部产生的，包括变量的赋值、表达式或函数的返回值。2.从其它来源获取到的数据，都是外部数据，也是用户输入数据，这些数据理应全部都是string类型的数据。

所以POSIX定义了一个名为"numeric string"的"墙头草"类型，gawk中则称为strnum类型。当获取到的用户数据看上去是数字时，那么它就是strnum类型。strnum类型在被使用时会被当作数值类型。

注意，strnum类型只针对于awk中除数值常量、字符串常量、表达式计算结果外的数据。例如从文件中读取的字段`$1`、`$2`、ARGV数组中的元素等等。

```
$ echo "30" | awk '{print typeof($0) " " typeof($1)}'
strnum strnum
$ echo "+30" | awk '{print typeof($1)}'
strnum
$ echo "30a" | awk '{print typeof($1)}'
string
$ echo "30 a" | awk '{print typeof($0) " " typeof($1)}'
string strnum
$ echo " +30 " | awk '{print typeof($0) " " typeof($1)}'
strnum strnum
```

大小比较操作

比较操作符：

```
< > <= >= != ==  大小、等值比较
in      数组成员测试
```

比较规则：

```
        |STRING NUMERIC STRNUM
-------|----------------------
STRING |string string  string
NUMERIC|string numeric numeric
STRNUM |string numeric numeric
```

简单来说，string优先级最高，只要string类型参与比较，就都按照string的比较方式，所以可能会进行隐式的类型转换。

其它时候都采用num类型比较。

```
$ echo ' +3.14' | awk '{print typeof($0) " " typeof($1)}'  #strnum strnum
$ echo ' +3.14' | awk '{print($0 == " +3.14")}'     #1
$ echo ' +3.14' | awk '{print($0 == "+3.14")}'      #0
$ echo ' +3.14' | awk '{print($0 == "3.14")}'       #0
$ echo ' +3.14' | awk '{print($0 == 3.14)}'         #1
$ echo ' +3.14' | awk '{print($1 == 3.14)}'         #1
$ echo ' +3.14' | awk '{print($1 == " +3.14")}'     #0
$ echo ' +3.14' | awk '{print($1 == "+3.14")}'      #1
$ echo ' +3.14' | awk '{print($1 == "3.14")}'       #0
$ echo 1e2 3|awk '{print ($1<$2)?"true":"false"}'  #false
```

采用字符串比较时需注意，它是逐字符逐字符比较的。

```
"11" < "9"  # true
"ab" < 99   # false
```

逻辑运算

```
&&          逻辑与
||          逻辑或
!           逻辑取反

expr1 && expr2  # 如果expr1为假，则不用计算expr2
expr1 || expr2  # 如果expr1为真，则不用计算expr2

# 注:
# 1. && ||会短路运算
# 2. !优先级高于&&和||
#    所以`! expr1 && expr2`等价于`(! expr1) && expr2`
```

`!`可以将数据转换成数值的1或0，取决于数据是布尔真还是布尔假。`!!`可将数据转换成等价布尔值的1或0。

```
$ awk 'BEGIN{print(!99)}'   # 0
$ awk 'BEGIN{print(!"ab")}' # 0
$ awk 'BEGIN{print(!0)}'    # 1
$ awk 'BEGIN{print(!ab)}'   # 1，因为ab变量不存在

$ awk 'BEGIN{print(!!99)}'   # 1
$ awk 'BEGIN{print(!!"ab")}' # 1
$ awk 'BEGIN{print(!!0)}'    # 0
$ awk 'BEGIN{print(!!ab)}'   # 0
```

由于awk中的变量未赋值时默认初始化为空字符串或数值0，也就是布尔假。那么可以直接对一个未赋值的变量执行`!`操作。

下面是一个非常有意思的awk技巧，它通过多次`!`对一个flag取反来实现只输出指定范围内的行。

```
# a.txt
$1==1{flag=!flag;print;next}    # 在匹配ID=1的行时，flag=1
flag{print}                # 将输出ID=2,3,4,5的行
$1==5{flag=!flag;next}     # ID=5时，flag=0
```

借此，就可以让awk实现一个多行处理模式。例如，将指定范围内的数据保存到一个变量当中去。

```
$1==1{flag=!flag;next}
flag{multi_line=multi_line$0"\n"}
$1==5{flag=!flag;next}
END{printf multi_line}
```

运算符优先级

优先级从高到低：man awk

```
()
$      # $(2+2)
++ --
^ **
+ - !   # 一元运算符
* / %
+ -
space  # 这是字符连接操作 `12 " " 23`  `12 " " -23`
| |&
< > <= >= != ==   # 注意>即是大于号，也是print/printf的重定向符号
~ !~
in
&&
||
?:
= += -= *= /= %= ^=
```

对于相同优先级的运算符，通常都是从左开始运算，但下面2种例外，它们都从右向左运算：

- 赋值运算：如 `= += -= *=`
- 幂运算

```
a - b + c  =>  (a - b) + c
a = b = c  =>  a =(b = c)
2**2**3    =>  2**(2**3)
```

再者，注意print和printf中出现的 `>` 符号，这时候它表示的是重定向符号，不能再出现优先级比它低的运算符，这时可以使用括号改变优先级。例如：

```
awk 'BEGIN{print "foo" > a < 3 ? 2 : 1)'   # 语法错误
awk 'BEGIN{print "foo" > (a < 3 ? 2 : 1)}' # 正确
```

流程控制语句

注：awk中语句块没有作用域，都是全局变量。

```
if (condition) statement [ else statement ]
expr1?expr2:expr3
while (condition) statement
do statement while (condition)
```

```
for (expr1; expr2; expr3) statement
for (var in array) statement
break
continue
next
nextfile
exit [ expression ]
{ statements }
switch (expression) {
    case value|regex : statement
    ...
    [ default: statement ]
}
```

代码块

```
{statement}
```

if...else

```
# 单独的if
if(cond){
    statements
}

# if...else
if(cond1){
    statements1
} else {
    statements2
}

# if...else if...else
if(cond1){
    statements1
} else if(cond2){
    statements2
} else if(cond3){
    statements3
} else{
    statements4
}
```

搞笑题：妻子告诉程序员老公，去买一斤包子，如果看见卖西瓜的，就买两个。结果是买了两个包子回来。

```
# 自然语言的语义
买一斤包子
if(有西瓜){
    买两个西瓜
}

# 程序员理解的语义
if(没有西瓜){
    买一斤包子
}else{
    买两个包子
}
```

```
awk '
  BEGIN{
    mark = 999
    if (mark >=0 && mark < 60) {
      print "学渣"
    } else if (mark >= 60 && mark < 90) {
      print "还不错"
    } else if (mark >= 90 && mark <= 100) {
      print "学霸"
    } else {
      print "错误分数"
    }
  }
'
```

三目运算符?:

```
expr1 ? expr2 : expr3

if(expr1){
    expr2
} else {
    expr3
}
```

```
awk 'BEGIN{a=50;b=(a>60) ? "及格" : "不及格";print(b)}'
awk 'BEGIN{a=50; a>60 ? b="及格" : b="不及格";print(b)}'
```

switch...case

```
switch (expression) {
    case value1|regex1 : statements1
    case value2|regex2 : statements2
    case value3|regex3 : statements3
    ...
    [ default: statement ]
}
```

awk 中的switch分支语句功能较弱，只能进行等值比较或正则匹配。

各分支结尾需使用break来终止。

```
{
    switch($1){
        case 1:
            print("Monday")
            break
        case 2:
            print("Tuesday")
            break
        case 3:
            print("Wednesday")
            break
        case 4:
            print("Thursday")
            break
        case 5:
            print("Friday")
            break
        case 6:
            print("Saturday")
            break
        case 7:
            print("Sunday")
            break
        default:
            print("What day?")
            break
    }
}
```

分支穿透：

```
{
    switch($1){
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
            print("Weekday")
            break
        case 6:
        case 7:
            print("Weekend")
            break
        default:
            print("What day?")
            break
    }
}
```

while和do...while

```
while(condition){
    statements
}

do {
    statements
} while(condition)
```

while先判断条件再决定是否执行statements，do...while先执行statements再判断条件决定下次是否再执行statements。

```
awk 'BEGIN{i=0;while(i<5){print i;i++}}'
awk 'BEGIN{i=0;do {print i;i++} while(i<5)}'
```

多数时候，while和do...while是等价的，但如果第一次条件判断失败，则do...while和while不同。

```
awk 'BEGIN{i=0;while(i == 2){print i;i++}}'
awk 'BEGIN{i=0;do {print i;i++} while(i ==2 )}'
```

所以，while可能一次也不会执行，do...while至少会执行一次。

一般用while，do...while相比while来说，用的频率非常低。

for循环

```
for (expr1; expr2; expr3) {
    statement
}

for (idx in array) {
    statement
}
```

break和continue

break可退出for、while、do...while、switch语句。

continue可让for、while、do...while进入下一轮循环。

```
 awk '
 BEGIN{
   for(i=0;i<10;i++){
     if(i==5){
       break
     }
     print(i)
   }

   # continue
   for(i=0;i<10;i++){
     if(i==5)continue
     print(i)
   }
```

```
15   }'
```

next和nextfile

next会在当前语句处立即停止后续操作，并读取下一行，进入循环顶部。

例如，输出除第3行外的所有行。

```
1   awk 'NR==3{next}{print}' a.txt
2   awk 'NR==3{getline}{print}' a.txt
```

nextfile会在当前语句处立即停止后续操作，并直接读取下一个文件，并进入循环顶部。

例如，每个文件只输出前2行：

```
1   awk 'FNR==3{nextfile}{print}' a.txt a.txt
```

exit

```
1   exit [exit_code]
```

直接退出awk程序。

注意，END语句块也是exit操作的一部分，所以在BEGIN或main段中执行exit操作，也会执行END语句块。

如果exit在END语句块中执行，则立即退出。

所以，如果真的想直接退出整个awk，则可以先设置一个flag变量，然后在END语句块的开头检查这个变量再exit。

```
1   BEGIN{
2       ...code...
3       if(cond){
4           flag=1
5           exit
6       }
7   }
8   {}
9   END{
10      if(flag){
11          exit
12      }
13      ...code...
14  }
15
16  awk '
17      BEGIN{print "begin";flag=1;exit}
18      {}
19      END{if(flag){exit};print "end2"}
20  '
```

exit可以指定退出状态码，如果触发了两次exit操作，即BEGIN或main中的exit触发了END中的exit，且END中的exit没有指定退出状态码时，则采取前一个退出状态码。

```
$ awk 'BEGIN{flag=1;exit 2}{}END{if(flag){exit 1}}'
$ echo $?
1

$ awk 'BEGIN{flag=1;exit 2}{}END{if(flag){exit}}'
$ echo $?
2
```

数组

awk数组特性：

- awk的数组是关联数组(即key/value方式的hash数据结构)，索引下标可为数值(甚至是负数、小数等)，也可为字符串
  - 在内部，awk数组的索引全都是字符串，即使是数值索引在使用时内部也会转换成字符串
  - awk的数组元素的顺序和元素插入时的顺序很可能是不相同的
- awk数组支持数组的数组

访问、赋值数组元素

```
arr[idx]
arr[idx] = value
```

索引可以是整数、负数、0、小数、字符串。如果是数值索引，会按照CONVFMT变量指定的格式先转换成字符串。

例如：

```
awk '
  BEGIN{
    arr[1]   = 11
    arr["1"] = 111
    arr["a"] = "aa"
    arr[-1]  = -11
    arr[4.3] = 4.33

    print arr[1]     # 111
    print arr["1"]   # 111
    print arr["a"]   # aa
    print arr[-1]    # -11
    print arr[4.3]   # 4.33
  }
'
```

通过索引的方式访问数组中不存在的元素时，会返回空字符串，同时会创建这个元素并将其值设置为空字符串。

```
awk '
  BEGIN{
    arr[-1]=3;
    print length(arr);  # 1
    print arr[1];
    print length(arr)   # 2
  }'
```

数组长度

awk提供了`length()`函数来获取数组的元素个数，它也可以用于获取字符串的字符数量。还可以获取数值转换成字符串后的字符数量。

```
awk 'BEGIN{arr[1]=1;arr[2]=2;print length(arr);print length("hello")}'
```

删除数组元素

- `delete arr[idx]`：删除数组`arr[idx]`元素
  - 删除不存在的元素不会报错
- `delete arr`：删除数组所有元素

```
$ awk 'BEGIN{arr[1]=1;arr[2]=2;arr[3]=3;delete arr[2];print length(arr)}'
2
```

检测是否是数组

`isarray(arr)`可用于检测arr是否是数组，如果是数组则返回1，否则返回0。

`typeof(arr)`可返回数据类型，如果arr是数组，则其返回"array"。

```
awk 'BEGIN{
    arr[1]=1;
    print isarray(arr);
    print (typeof(arr) == "array")
}'
```

测试元素是否存在于数组当中

不要使用下面的方式来测试元素是否在数组中：

```
if(arr["x"] != ""){...}
```

这有两个问题：

- 如果不存在arr["x"]，则会立即创建该元素，并将其值设置为空字符串
- 有些元素的值本身就是空字符串

应当使用数组成员测试操作符in来测试：

```
# 注意，idx不要使用index，它是一个内置函数
if (idx in arr){...}
```

它会测试索引idx是否在数组中，如果存在则返回1，不存在则返回0。

```
awk '
    BEGIN{
        arr[1]=1;
        arr[2]=2;
        arr[3]=3;

        arr[1]="";
        delete arr[2];

        print (1 in arr);  # 1
        print (2 in arr);  # 0
    }'
```

遍历数组

awk提供了一种for变体来遍历数组：

```
for(idx in arr){print arr[idx]}
```

因为awk数组是关联数组，元素是不连续的，也就是说没有顺序。遍历awk数组时，顺序是不可预测的。

例如：

```
awk '
    BEGIN{
        arr["one"] = 1
        arr["two"] = 2
        arr["three"] = 3
        arr["four"] = 4
        arr["five"] = 5

        for(i in arr){
            print i " -> " arr[i]
        }
    }
'
```

此外，不要随意使用 `for(i=0;i<length(arr);i++)` 来遍历数组，因为awk数组是关联数组。但如果已经明确知道数组的所有元素索引都位于某个数值范围内，则可以使用该方式进行遍历。

例如：

```
awk '
```

```
    BEGIN{
        arr[1] = "one"
        arr[2] = "two"
        arr[3] = "three"
        arr[4] = "four"
        arr[5] = "five"
        arr[10]= "ten"

        for(i=0;i<=10;i++){
            if(i in arr){
                print arr[i]
            }
        }
    }
'
```

复杂索引的数组

在awk中，很多时候单纯的一个数组只能存放两个信息：一个索引、一个值。但在一些场景下，这样简单的存储能力在处理复杂需求的时候可能会捉襟见肘。

为了存储更多信息，方式之一是将第3份、第4份等信息全部以特殊方式存放到值中，但是这样的方式在实际使用过程中并不方便，每次都需要去分割值从而取出各部分的值。

另一种方式是将第3份、第4份等信息存放在索引中，将多份数据组成一个整体构成一个索引。

gawk中提供了将多份数据信息组合成一个整体当作一个索引的功能。默认方式为 `arr[x,y]` ，其中x和y是要结合起来构建成一个索引的两部分数据信息。逗号称为下标分隔符，在构建索引时会根据预定义变量SUBSEP的值将多个索引组合起来。所以 `arr[x,y]` 其实完全等价于 `arr[x SUBSEP y]` 。

例如，如果SUBSEP设置为"@"，那么 `arr[5,12] = 512` 存储时，其真实索引为 `5@12` ，所以要访问该元素需使用 `arr["5@12"]` 。

SUBSEP的默认值为 `\034` ，它是一个不可打印的字符，几乎不可能会出现在字符串当中。

如果我们愿意的话，我们也可以自己将多份数据组合起来去构建成一个索引，例如 `arr[x" "y]` 。但是awk提供了这种更为简便的方式，直接用即可。

为了测试这种复杂数组的索引是否在数组中，可以使用如下方式：

```
arr["a","b"] = 12
if (("a", "b") in arr){...}
```

例如，顺时针倒转下列数据：

```
1 2 3 4 5 6
2 3 4 5 6 1
3 4 5 6 1 2
4 5 6 1 2 3

结果:
4 3 2 1
5 4 3 2
6 5 4 3
1 6 5 4
2 1 6 5
3 2 1 6
```

```
{
  nf = NF
  nr = NR
  for(i=1;i<=NF;i++){
    arr[NR,i] = $i
  }
}

END{
  for(i=1;i<=nf;i++){
    for(j=nr;j>=1;j--){
      if(j%nr == 1){
        printf "%s\n", arr[j,i]
      }else {
        printf "%s ", arr[j,i]
      }
    }
  }
}
```

子数组

子数组是指数组中的元素也是一个数组，即Array of Array，它也称为子数组(subarray)。

awk也支持子数组，在效果上即是嵌套数组或多维数组。

```
a[1][1] = 11
a[1][2] = 12
a[1][3] = 13
a[2][1] = 21
a[2][2] = 22
a[2][3] = 23
a[2][4][1] = 241
a[2][4][2] = 242
a[2][4][1] = 241
a[2][4][3] = 243
```

通过如下方式遍历二维数组：

```
for(i in a){
    for (j in a[i]){
        if(isarray(a[i][j])){
            continue
        }
        print a[i][j]
    }
}
```

指定遍历顺序

由于awk数组是关联数组，默认情况下，`for(idx in arr)`遍历数组时顺序是不可预测的。

但是gawk提供了`PROCINFO["sorted_in"]`来指定遍历的元素顺序。它可以设置为两种类型的值：

- 设置为用户自定义函数
- 设置为下面这些awk预定义好的值：
  - `@unsorted`：默认值，遍历时无序
  - `@ind_str_asc`：索引按字符串比较方式升序遍历
  - `@ind_str_desc`：索引按字符串比较方式降序遍历
  - `@ind_num_asc`：索引强制按照数值比较方式升序遍历。所以无法转换为数值的字符串索引将当作数值0进行比较
  - `@ind_num_desc`：索引强制按照数值比较方式降序遍历。所以无法转换为数值的字符串索引将当作数值0进行比较
  - `@val_type_asc`：按值升序比较，此外数值类型出现在前面，接着是字符串类型，最后是数组类(即认为`num<str<arr`)
  - `@val_type_desc`：按值降序比较，此外数组类型出现在前面，接着是字符串类型，最后是数值型(即认为`num<str<arr`)
  - `@val_str_asc`：按值升序比较，数值转换成字符串再比较，而数组出现在尾部(即认`str<arr`)
  - `@val_str_desc`：按值降序比较，数值转换成字符串再比较，而数组出现在头部(即认`str<arr`)
  - `@val_num_asc`：按值升序比较，字符串转换成数值再比较，而数组出现在尾部(即认`num<arr`)
  - `@val_num_desc`：按值降序比较，字符串转换成数值再比较，而数组出现在头部(即认为`num<arr`)

例如：

```
awk '
  BEGIN{
    arr[1] = "one"
    arr[2] = "two"
    arr[3] = "three"
    arr["a"] ="aa"
    arr["b"] ="bb"
    arr[10]= "ten"

    #PROCINFO["sorted_in"] = "@ind_num_asc"
    #PROCINFO["sorted_in"] = "@ind_str_asc"
    PROCINFO["sorted_in"] = "@val_str_asc"
    for(idx in arr){
```

```
14          print idx " -> " arr[idx]
15        }
16    }'
17
18    a -> aa
19    b -> bb
20    1 -> one
21    2 -> two
22    3 -> three
23    10 -> ten
```

如果指定为用户自定义的排序函数，其函数格式为：

```
1   function sort_func(i1,v1,i2,v2){
2       ...
3       return <0;0;>0
4   }
```

其中，i1和i2是每次所取两个元素的索引，v1和v2是这两个索引的对应值。

如果返回值小于0，则表示i1在i2前面，i1先被遍历。如果等于0，则表示i1和i2具有等值关系，它们的遍历顺序不可保证。如果大于0，则表示i2先于i1被遍历。

例如，对数组元素按数值大小比较来决定遍历顺序。

```
1   awk '
2   function cmp_val_num(i1, v1, i2, v2){
3     if ((v1 - v2) < 0) {
4       return -1
5     } else if ((v1 - v2) == 0) {
6       return 0
7     } else {
8       return 1
9     }
10    # return (v1-v2)
11  }
12
13  NR > 1 {
14    arr[$0] = $4
15  }
16
17  END {
18    PROCINFO["sorted_in"] = "cmp_val_num"
19    for (i in arr) {
20      print i
21    }
22  }' a.txt
```

再比如，按数组元素值的字符大小来比较。

```
1   function cmp_val_str(i1,v1,i2,v2) {
2       v1 = v1 ""
3       v2 = v2 ""
4       if(v1 < v2){
5           return -1
```

```
    } else if(v1 == v2){
        return 0
    } else {
        return 1
    }
    # return (v1 < v2) ? -1 : (v1 != v2)
}

NR>1{
    arr[$0] = $2
}

END{
    PROCINFO["sorted_in"] = "cmp_val_str"
    for(line in arr)
    {
        print line
    }
}
```

再比如，对元素值按数值升序比较，且相等时再按第一个字段ID进行数值降序比较。

```
awk '
function cmp_val_num(i1,v1,i2,v2,    a1,a2) {
    if (v1<v2) {
        return - 1
    } else if(v1 == v2){
        split(i1, a1, SUBSEP)
        split(i2, a2, SUBSEP)
        return a2[2] - a1[2]
    } else {
        return 1
    }
}

NR>1{
    arr[$0,$1] = $4
}

END{
    PROCINFO["sorted_in"] = "cmp_val_num"
    for(str in arr){
        split(str, a, SUBSEP)
        print a[1]
    }
}

' a.txt
```

上面使用的 `arr[x,y]` 来存储额外信息，下面使用 `arr[x][y]` 多维数组的方式来存储额外信息实现同样的排序功能。

```
NR>1{
  arr[NR][$0] = $4
```

```
}

END{
  PROCINFO["sorted_in"] = "cmp_val_num"
  for(nr in arr){
    for(line in arr[nr]){
      print line
    }
  }
}

function cmp_val_num(i1,v1,i2,v2,   ii1,ii2){
  # 获取v1/v2的索引，即$0的值
  for(ii1 in v1){ }
  for(ii2 in v2){ }

  if(v1[ii1] < v2[ii2]){
    return -1
  }else if(v1[ii1] > v2[ii2]){
    return 1
  }else{
    return (i2 - i1)
  }
}
```

此外，gawk还提供了两个内置函数asort()和asorti()来对数组进行排序。

ARGC和ARGV

预定义变量ARGV是一个数组，包含了所有的命令行参数。该数组使用从0开始的数值作为索引。

预定义变量ARGC初始时是ARGV数组的长度，即命令行参数的数量。

ARGV数组的数量和ARGC的值只有在awk刚开始运行的时候是保证相等的。

```
$ awk -va=1 -F: '
  BEGIN{
    print ARGC;
    for(i in ARGV){
      print "ARGV[" i "]= " ARGV[i]
    }
}' b=3 a.txt b.txt

4
ARGV[0]= awk
ARGV[1]= b=3
ARGV[2]= a.txt
ARGV[3]= b.txt
```

awk读取文件是根据ARGC的值来进行的，有点类似于如下伪代码形式：

```
while(i=1;i<ARGC;i++){
    read from ARGV[i]
}
```

默认情况下，awk在读完ARGV中的一个文件时，会自动从它的下一个元素开始读取，直到读完所有文件。

直接减小ARGC的值，会导致awk不会读取尾部的一些文件。此外，增减ARGC的值，都不会影响ARGV数组，仅仅只是影响awk读取文件的数量。

```
# 不会读取b.txt
awk 'BEGIN{ARGC=2}{print}' a.txt b.txt

# 读完b.txt后自动退出
awk 'BEGIN{ARGC=5}{print}' a.txt b.txt
```

可以将ARGV中某个元素赋值为空字符串""，awk在选择下一个要读取的文件时，会自动忽略ARGV中的空字符串元素。

也可以 `delete ARGV[i]` 的方式来删除ARGV中的某元素。

用户手动增、删ARGV元素时，不会自动修改ARGC，而awk读取文件时是根据ARGC值来确定的。所以，在增加ARGV元素之后，要手动的去增加ARGC的值。

```
# 不会读取b.txt文件
$ awk 'BEGIN{ARGV[2]="b.txt"}{print}' a.txt

# 会读取b.txt文件
$ awk 'BEGIN{ARGV[2]="b.txt";ARGC++}{print}' a.txt
```

对ARGC和ARGV进行操刀

判断命令行中给定文件是否可读

awk命令行中可能会给出一些不存在或无权限或其它原因而无法被awk读取的文件名，这时可以判断并从中剔除掉不可读取的文件。

1.排除命令行尾部(非选项型参数)的var=val、-、和/dev/stdin这3种特殊情况
2.如果不可读，则从ARGV中删除该参数
3.剩下的都是可在main代码段正常读取的文件

```
BEGIN{
  for(i=1;i<ARGC;i++){
    if(ARGV[i] ~ /[a-zA-Z_][a-zA-Z0-9_]*=.*/ \
    || ARGV[i]=="-" || ARGV[i]=="/dev/stdin"){
      continue
    } else if((getline var < ARGV[i]) < 0){
      delete ARGV[i]
    } else{
      close(ARGV[i])
    }
  }
}
```

自定义函数

可以定义一个函数将多个操作整合在一起。函数定义之后，可以到处多次调用，从而方便复用。

使用function关键字来定义函数：

```
function func_name([parameters]){
    function_body
}
```

对于gawk来说，也支持func关键字来定义函数。

```
func func_name(){}
```

函数可以定义在下面使用下划线的地方：

```
awk '_ BEGIN{} _ MAIN{} _ END{} _'
```

无论函数定义在哪里，都能在任何地方调用，因为awk在BEGIN之前，会先编译awk代码为内部格式，在这个阶段会将所有函数都预定义好。

例如：

```
awk '
    BEGIN{
        f()
        f()
        f()
    }
    function f(){
        print "星期一"
        print "星期二"
        print "星期三"
        print "星期四"
        print "星期五"
        print "星期六"
        print "星期日"
    }
'
```

函数的return语句

如果想要让函数有返回值，那么需要在函数中使用return语句。

return语句也可以用来立即结束函数的执行。

例如：

```
awk '
    function add(){
        return 40
    }
    BEGIN{
        print add()
        res = add()
        print res
    }
'
```

如果不使用return或return没有参数，则返回值为空，即空字符串。

```
awk '
  function f1(){        }
  function f2(){return  }
  function f3(){return 3}
  BEGIN{
    print "-"f1()"-"
    print "-"f2()"-"
    print "-"f3()"-"
  }
'
```

函数参数

为了让函数和调用者能够进行数据的交互，可以使用参数。

```
awk '
  function f(a,b){
    print a
    print b
    return a+b
  }
  BEGIN{
    x=10
    y=20
    res = f(x,y)
    print res
    print f(x,y)
  }
'
```

例如，实现一个重复某字符串指定次数的函数：

```
awk '
    function repeat(str,cnt  ,res_str){
        for(i=0;i<cnt;i++){
            res_str = res_str""str
        }
        return res_str
    }
    BEGIN{
        print repeat("abc",3)
        print repeat("-",30)
    }
'
```

调用函数时，实参数量可以比形参数量少，也可以比形参数量多。但是，在多于形参数量时会给出警告信息。

```
awk '
  function f(a,b){
    print a
    print b
    return a+b
  }
  BEGIN{
    x=10
    y=20

    print "---1----"
    print "-"f()"-"          # 不传递参数

    print "---2----"
    print "-"f(30)"-"        # 传递1个参数

    print "---3----"
    print "-"f(10,20,30)"-"  # 传递多个参数
  }
'
```

参数数据类型冲突问题

如果函数内部使用参数的类型和函数外部变量的类型不一致，会出现数据类型不同而导致报错。

```
awk '
    function f(a){
        a[1]=30
    }
    BEGIN{
        a="hello world"
        f(a)   # 报错

        f(x)
        x=10   # 报错
    }
'
```

函数内部参数对应的是数组，那么外面对应的也必须是数组类型。

参数按值传递还是按引用传递

在调用函数时，将数据作为函数参数传递给函数时，有两种传递方式：

- 传递普通变量时，是按值拷贝传递
  - 直接拷贝普通变量的**值**到函数中
  - 函数内部修改不会影响到外部
- 传递数组时，是按引用传递
  - 函数内部修改会影响到外部

```
# 传递普通变量：按值拷贝
awk '
  function modify(a){
    a=30
    print a
  }
  BEGIN{
    a=40
    modify(a)
    print a
  }
'

# 传递数组：按引用拷贝
awk '
  function modify(a){
    a[1]=20
  }

  BEGIN{
    a[1]=10
    modify(a)
    print a[1]
  }
'
```

awk作用域问题

awk只有在函数参数中才是局部变量，其它地方定义的变量均为全局变量。

函数内部新增的变量是全局变量，会影响到全局，所以在函数退出后仍然能访问。例如上面的e变量。

```
awk '
  function f(){
    a=30  # 新增的变量，是全局变量
    print "in f: " a
  }
  BEGIN{
    a=40
    f()
    print a  # 30
  }
'
```

函数参数会遮掩全局同名变量，所以在函数执行时，无法访问到或操作与参数同名的全局变量，函数退出时会自动撤掉遮掩，这时才能访问全局变量。所以，参数具有局部效果。

```
awk '
  function f(a){
    print a    # 50，按值拷贝，和全局a已经没有关系
    a=40
    print a    # 40
  }
  BEGIN{
    a=50
    f(a)
    print a     # 50，函数退出，重新访问全局变量
  }
'
```

由于函数内部新增变量均为全局变量，awk也没有提供关键字来修饰一个变量使其成为局部变量。所以，awk只能将本该出现在函数体内的局部变量放在参数列表中，只要调用函数时不要为这些参数传递数据即可，从而实现局部变量的效果。

```
awk '
  function f(a,b       ,c,d){

    # a,b是参数，调用时需传递两个参数
    # c,d是局部变量，调用时不要给c和d传递数据
    a=30
    b=40
    c=50
    d=60
    e=70  # 全局变量

    print a,b,c,d,e  # 30 40 50 60 70
  }
  BEGIN{
    a=31
```

```
    b=41
    c=51
    d=61
    f(a,b)  # 调用函数时值传递两个参数
    print a,b,c,d,e  # 31 41 51 61 70
  }
'
```

所以，awk对函数参数列表做了两类区分：

- arguments：调用函数时传递的参数
- local variables：调用函数时省略的参数

为了区分arguments和local variables，约定俗成的，将local variables放在一大堆空格后面来提示用户。例如 `function name(a,b, c,d)` 表示调用函数时，应当传递两个参数，c和d是本函数内部使用的局部变量，不要传递对应的参数。

区分参数和局部变量：

- 参数提供了函数和它调用者进行数据交互的方式
- 局部变量是临时存放数据的地方

arguments部分体现的是函数调用时传递的参数，这些参数在函数内部会遮掩全局同名变量。例如上面示例中，函数内部访问不了全局的a和b，所有对a和b的操作都是函数内部的，函数退出后才能重新访问全局a和b。因此，arguments也有局部特性。

local variables是awk实现真正局部变量的技巧，只是因为函数内部新增的变量都是全局变量，所以退而求其次将其放在参数列表上来实现局部变量。

自定义函数示例

1.一次性读取一个文件所有数据

```
function readfile(file    ,rs_bak,data){
  rs_bak=RS
  RS="^$"
  if ( (getline data < file) < 0 ){
    print "read file failed"
    exit 1
  }
  close(file)
  RS=rs_bak
  return data
}


/^1/{
  print $0
  content = readfile("c.txt")
  print content
}
```

将RS设置为`^$`是永远不可能出现的分隔符，除非这个文件为空文件。

2.重读文件

实现一个rewind()功能来重置文件偏移指针，从而模拟实现重读当前文件。

```
function rewind(    i){
    # 将当前正在读取的文件添加到ARGV中当前文件的下一个元素
    for(i=ARGC;i>ARCIND;i--){
        ARGV[i] = ARGV[i-1]
    }

    # 随着增加ARGC，以便awk能够读取到因ARGV增加元素后的最后一个文件
    ARGC++

    # 直接进入下一个文件
    nextfile
}
```

要注意可能出现无限递归的场景：

```
awk -f rewind.awk 'NR==3{rewind()}{print FILENAME, FNR, $0}' a.txt

# 下面这个会无限递归，因为FNR==3很可能每次重读时都会为真
awk -f rewind.awk 'FNR==3{rewind()}{print FILENAME, FNR, $0}' a.txt
```

3.格式化数组的输出

实现一个a2s()函数。

```
BEGIN{
  arr["zhangsan"]=21
  arr["lisi"]=22
  arr["wangwu"]=23
  print a2s(arr)
}

function a2s(arr       ,content,i,cnt){
  for(i in arr){
    if(cnt){
      content=content""(sprintf("\t%s:%s\n",i,arr[i]))
    } else {
      content=content""(sprintf("\n\t%s:%s\n",i,arr[i]))
    }
    cnt++
  }
  return "{"content"}"
}
```

4.禁用命令行尾部的赋值语句

`awk '{}' ./a=b a.txt`中，`a=b`会被awk识别为变量赋值操作。但是，如果用户想要处理的正好是包含了等号的文件名，则应当去禁用该赋值操作。

禁用的方式很简单，只需为其加上一个路径前缀`./`即可。

为了方便控制，可通过`-v`设置一个flag类型的选项标记。

```
function disable_assigns(argc,argv,    i){
    for(i=1;i<argc;i++){
        if(argv[i] ~ /[[:alpha:]_][[:alnum:]_]*=.*/){
            argv[i] = ("./"argv[i])
        }
    }
}

BEGIN{
    if(assign_flag){
        disable_assigns(ARGC,ARGV)
    }
}
```

那么，调用awk时采用如下方式：

```
awk -v assign_flag=1 -f assigns.awk '{print}' a=b.txt a.txt
```

awk选项、内置变量、内置函数

选项

```
-e program-text
--source program-text
指定awk程序表达式，可结合-f选项同时使用
在使用了-f选项后，如果不使用-e，awk program是不会执行的，它会被当作ARGV的一个参数

-f program-file
--file program-file
从文件中读取awk源代码来执行，可指定多个-f选项

-F fs
--field-separator fs
指定输入字段分隔符(FS预定义变量也可设置)

-n
--non-decimal-data
识别文件输入中的8进制数(0开头)和16进制数(0x开头)
echo '030' | awk -n '{print $1+0}'

-o [filename]
格式化awk代码。
不指定filename时，则默认保存到awkprof.out
指定为`-`时，表示输出到标准输出
```

```
23
24    -v var=val
25    --assign var=val
26    在BEGIN之前，声明并赋值变量var，变量可在BEGIN中使用
```

预定义变量

预定义变量分为两类：控制awk工作的变量和携带信息的变量。

第一类：控制AWK工作的预定义变量

- `RS`：输入记录分隔符，默认为换行符`\n`，参考**RS**
- `IGNORECASE`：默认值为0，表示所有的正则匹配不忽略大小写。设置为非0值（例如1），之后的匹配将忽略大小写。例如在BEGIN块中将其设置为1，将使FS、RS都以忽略大小写的方式分隔字段或分隔record
- `FS`：读取记录后，划分为字段的字段分隔符。参考**FS**
- `FIELDWIDTHS`：以指定宽度切割字段而非按照FS。参考**FIELDWIDTHS**
- `FPAT`：以正则匹配匹配到的结果作为字段，而非按照FS划分。参考**FPAT**
- `OFS`：print命令输出各字段列表时的输出字段分隔符，默认为空格" "
- `ORS`：print命令输出数据时在尾部自动添加的记录分隔符，默认为换行符`\n`
- `CONVFMT`：在awk中数值隐式转换为字符串时，将根据CONVFMT的格式按照sprintf()的方式自动转换为字符串。默认值为"%.6g
- `OFMT`：在print中，数值会根据OFMT的格式按照sprintf()的方式自动转换为字符串。默认值为"%.6g

第二类：携带信息的预定义变量

- `ARGC`和`ARGV`：awk命令行参数的数量、命令参数的数组。参考**ARGC和ARGV**
- `ARGIND`：awk当前正在处理的文件在ARGV中的索引位置。所以，如果awk正在处理命令行参数中的某文件，则`ARGV[ARGIND] == FILENAME`为真
- `FILENAME`：awk当前正在处理的文件(命令行中指定的文件)，所以在BEGIN中该变量值为空
- `ENVIRON`：保存了Shell的环境变量的数组。例如`ENVIRON["HOME"]`将返回当前用户的家目录
- `NR`:当前已读总记录数，多个文件从不会重置为0，所以它是一直叠加的
  - 可以直接修改NR，下次读取记录时将在此修改值上自增
- `FNR`:当前正在读取文件的第几条记录，每次打开新文件会重置为0
  - 可以直接修改FNR，下次读取记录时将在此修改值上自增
- `NF`：当前记录的字段数，参考**NF**
- `RT`：在读取记录时真正的记录分隔符，参考**RT**
- `RLENGTH`：match()函数正则匹配成功时，所匹配到的字符串长度，如果匹配失败，该变量值为-1
- `RSTART`：match()函数匹配成功时，其首字符的索引位置，如果匹配失败，该变量值为0
- `SUBSEP`：`arr[x,y]`中下标分隔符构建成索引时对应的字符，默认值为`\034`，是一个不太可能出现在字符串中的不可打印字符。参考**复杂数组**

```
awk '
  BEGIN{
    for(idx in PROCINFO){
      if(typeof(PROCINFO[idx]) == "array"){
        continue
      }
      print idx " -> "PROCINFO[idx]
    }
  }'
```

预定义函数

预定义函数分为几类：

- 数值类内置函数
- 字符串类内置函数
- 时间类内置函数
- 位操作内置函数
- 数据类型相关内置函数：isarray()、typeof()
- IO类内置函数：close()、system()、fflush()

数值类内置函数

```
int(expr)      截断为整数：int(123.45)和int("123abc")都返回123，int("a123")返回0
sqrt(expr)     返回平方根
rand()         返回[0,1)之间的随机数，默认使用srand(1)作为种子值
srand([expr]) 设置rand()种子值，省略参数时将取当前时间的epoch值(精确到秒的epoch)作为种子值
```

例如：

```
$ awk 'BEGIN{srand();print rand()}'
0.0379114
$ awk 'BEGIN{srand();print rand()}'
0.0779783
$ awk 'BEGIN{srand(2);print rand()}'
0.893104
$ awk 'BEGIN{srand(2);print rand()}'
0.893104
```

生成`[10,100]`之间的随机整数。

```
awk 'BEGIN{srand();print 10+int(91*rand())}'
```

字符串类内置函数

注意，awk中涉及到**字符索引的函数，索引位都是从1开始计算**，和其它语言从0开始不一样。

- `sprintf(format, expression1, ...)`：返回格式化后的字符串，参考`sprintf`
  - `a=sprintf("%s\n","abc")`
- `length()`：返回字符串字符数量、数组元素数量、或数值转换为字符串后的字符数量

```
awk '
    BEGIN{
        print length(1.23)     # 4   # CONVFMT %.6g

        print 1.234567         # 1.23457
        print length(1.234567) # 7
        print length(122341223432.1213241234)  # 11
    }'
```

- `strtonum(str)`：将字符串转换为十进制数值
  - 如果str以0开头，则将其识别为8进制
  - 如果str以0x或0X开头，则将其识别为16进制
- `tolower(str)`：转换位小写
- `toupper(str)`：转换位大写
- `index(str,substr)`：从str中搜索substr(子串)，返回搜索到的索引位置(索引从1开始)，搜索不到则返回0

substr()

- `substr(string,start[,length])`：从string中截取子串

start是截取的起始索引位(索引位从1开始而非0)，length表示截取的子串长度。如果省略length，则表示从start开始截取剩余所有字符。

```
awk '
    BEGIN{
        str="abcdefgh"
        print substr(str,3)   # cdefgh
        print substr(str,3,3) # cde
    }
'
```

如果start值小于1，则将其看作为1对待，如果start大于字符串的长度，则返回空字符串。

如果length小于或等于0，则返回空字符串。

split()和patsplit()

- `split(string, array [, fieldsep [, seps ] ])`：将字符串分割后保存到数组array中，数组索引从1开始存储。并返回分割得到的元素个数

其中fieldsep指定分隔符，可以是正则表达式方式的。如果不指定该参数，则默认使用FS作为分隔符，而FS的默认值又是空格。

seps是一个数组，保存了每次分割时的分隔符。

例如：

```
split("abc-def-gho-pq",arr,"-",seps)
```

其返回值为4。同时得到的数组a和seps为：

```
arr[1] = "abc"
arr[2] = "def"
arr[3] = "gho"
arr[4] = "pq"

seps[1] = "-"
seps[2] = "-"
seps[3] = "-"
```

split在开始工作时，会先清空数组，所以，将split的string参数设置为空，可以用于清空数组。

```
awk 'BEGIN{arr[1]=1;split("",arr);print length(arr)}'  # 0
```

如果分隔符无法匹配字符串，则整个字符串当作一个数组元素保存到数组array中。

```
awk 'BEGIN{split("abcde",arr,"-");print arr[1]}' # abcde
```

- `patsplit(string, array [, fieldpat [, seps ] ])`：用正则表达式fieldpat匹配字符串string，将所有匹配成功的部分保存到数组array中，数组索引从1开始存储。返回值是array的元素个数，即匹配成功了多少次

如果省略fieldpat，则默认采用预定义变量FPAT的值。

```
awk '
    BEGIN{
        patsplit("abcde",arr,"[a-z]")
        print arr[1]   # a
        print arr[2]   # b
        print arr[3]   # c
        print arr[4]   # d
        print arr[5]   # e
    }
'
```

match()

- `match(string,reg[,arr])`：使用reg匹配string，返回匹配成功的索引位(从1开始计数)，匹配失败则返回0。如果指定了arr参数，则arr[0]保存的是匹配成功的字符串，arr[1]、arr[2]、...保存的是各个分组捕获的内容

match匹配时，同时会设置两个预定义变量：RSTART和RLENGTH

- 匹配成功时：
  - RSTART赋值为匹配成功的索引位，从1开始计数
  - RLENGTH赋值为匹配成功的字符长度
- 匹配失败时：
  - RSTART赋值为0
  - RLENGTH赋值为-1

例如：

```
awk '
    BEGIN{
        where = match("foooobazbarrrr","(fo+).*(bar*)",arr)
        print where   # 1
        print arr[0]  # foooobazbarrrr
        print arr[1]  # foooo
        print arr[2]  # barrrr
        print RSTART  # 1
        print RLENGTH # 14
    }
'
```

因为match()匹配成功时返回值为非0，而匹配失败时返回值为0，所以可以直接当作条件判断：

```
awk '
  {
    if(match($0,/A[a-z]+/,arr)){
      print NR " : " arr[0]
    }
  }
' a.txt
```

sub()和gsub()

- `sub(regexp, replacement [, target])`
- `gsub(regexp, replacement [, target])`：sub()的全局模式

sub()从字符串target中进行正则匹配，并使用replacement对第一次匹配成功的部分进行替换，替换后保存回target中。返回替换成功的次数，即0或1。

target必须是一个可以赋值的变量名、$N或数组元素名，以便用它来保存替换成功后的结果。不能是字符串字面量，因为它无法保存数据。

如果省略target，则默认使用 `$0` 。

需要注意的是，如果省略target，或者target是 `$N` ，那么替换成功后将会使用OFS重新计算 `$0` 。

```
awk '
    BEGIN{
        str="water water everywhere"
        #how_many = sub(/at/, "ith", str)
        how_many = gsub(/at/, "ith", str)
        print how_many   # 1
        print str        # wither water everywhere
    }
'
```

在replacement参数中，可以使用一个特殊的符号 `&` 来引用匹配成功的部分。注意sub()和gsub()不能在replacement中使用反向引用 `\N` 。

```
awk '
    BEGIN{
        str = "daabaaa"
        gsub(/a+/,"C&C",str)
        print str  # dCaaCbaaa
    }
'
```

如果想要在replacement中使用`&`纯字符，则转义即可。

```
sub(/a+/,"C\\&C",str)
```

> 两根反斜线：
> 因为awk在正则开始工作时，首先会扫描所有awk代码然后编译成awk的内部格式，扫描期间会解析反斜线转义，使得`\\`变成一根反斜线。当真正开始运行后，sub()又要解析，这时`\&`才表示的是对&做转义。 扫描代码阶段称为词法解析阶段，运行解析阶段称为运行时解析阶段。

gensub()

gawk支持的gensub()，完全可以取代sub()和gsub()。

- `gensub(regexp, replacement, how [, target])`：

可以替代sub()和gsub()。

how指定替换第几个匹配，例如指定为1表示只替换第一个匹配。此外，还可以指定为`g`或`G`开头的字符串，表示全局替换。

gensub()返回替换后得到的结果，而target不变，如果匹配失败，则返回target。这和sub()、gsub()不一样，sub()、gsub()返回的是替换成功的次数。

gensub()的replacement部分可以使用`\N`来引用分组匹配的结果，而sub()、gsub()不允许使用反向引用。而且，gensub()在replacement部分也还可以使用`&`或`\0`来表示匹配的整个结果。

```
awk 'BEGIN{
    a = "abc def"
    b = gensub(/(.+) (.*)/, "\\2 \\1, \\0 , &", "g", a)
    print b  # def abc, abc def , abc def
}'
```

asort()和asorti()

- `asort(src,[dest [,how]])`
- `asorti(src,[dest [,how]])`

asort对数组src的值进行排序，然后将排序后的值的索引改为1、2、3、4...序列。返回src中的元素个数，它可以当作排序后的索引最大值。

asorti对数组src的索引进行排序，然后将排序后的索引值的索引改为1、2、3、4...序列。返回src中的元素个数，它可以当作排序后的索引最大值。

```
arr["last"] = "de"
arr["first"] = "sac"
arr["middle"] = "cul"
```

asort(arr)得到：

```
arr[1] = "cul"
arr[2] = "de"
arr[3] = "sac"
```

asorti(arr)得到：

```
arr[1] = "first"
arr[2] = "last"
arr[3] = "middle"
```

如果指定dest，则将原始数组src备份到dest，然后对dest进行排序，而src保持不变。

how参数用于指定排序时的方式，其值指定方式和 `PROCINFO["sorted_in"]` 一致：可以是预定义的排序函数，也可以是用户自定义的排序函数。参考 **指定数组遍历顺序**。

IO类内置函数

- `close(filename [, how])`：关闭文件或命令，参考 **close**
- `system(command)`：执行Shell命令，参考 **system**
- `fflush([filename])`：gawk会按块缓冲模式来缓冲输出结果，使用fflush()会将缓冲数据刷出

从gawk 4.0.2之后的版本(不包括4.0.2)，无参数fflush()将刷出所有缓冲数据。

此外，终端设备是行缓冲模式，此时不需要fflush，而重定向到文件、到管道都是块缓冲模式，此时可能需要fflush()。

此外，system()在运行时也会flush gawk的缓冲。特别的，如果system的参数为空字符串 `system("")`，则它不会去启动一个shell子进程而是仅仅执行flush操作。

没有flush时：

```
# 在终端输入，将不会显示，直到按下Ctrl + D
awk '{print "first";print "second"}' | cat
```

使用fflush()：

```
# 在终端输入
awk '{print "first";fflush();print "second"}' | cat
```

使用system()来flush：

```
awk '{print "first";system("echo system");print "second"}' | cat
awk '{print "first";system("");print "second"}' | cat
```

也可以使用 `stdbuf -oL` 命令来强制gawk按行缓冲而非默认的按块缓冲。

```
stdbuf -oL awk '{print "first";print "second"}' | cat
```

fflush()也可以指定文件名或命令，表示只刷出到该文件或该命令的缓冲数据。

```
# 刷出所有流向到标准输出的缓冲数据
awk '{print "first";fflush("/dev/stdout");print "second"}' | cat
```

最后注意，fflush()刷出缓冲数据不代表发送EOF标记。

时间类内置函数

awk常用于处理日志，它支持简单的时间类操作。有下面3个内置的时间函数：

- `mktime("YYYY MM DD HH mm SS [DST]")`：构建一个时间，返回这个时间点的秒级epoch，构建失败则返回-1
- `systime()`：返回当前系统时间点，返回的是秒级epoch值
- `strftime([format [, timestamp [, utc-flag] ] ])`：将时间按指定格式转换为字符串并返回转的结果字符串

注意，awk构建时间时都是返回秒级的epoch值，表示从`1970-01-01 00:00:00`开始到指定时间已经过的秒数。

```
awk 'BEGIN{print systime();print mktime("2019 2 29 12 32 59")}'
1572364974
1551414779
```

mktime

mktime在构建时间时，如果传递的DD给定的值超出了月份MM允许的天数，则自动延申到下个月。例如，指定"2019 2 29 12 30 59"中2月只有28号，所以构建出来的时间是`2019-03-01 12:30:59`。

此外，其它部分也不限定必须在范围内。例如，`2019 2 23 12 32 65`的秒超出了59，那么多出来的秒数将进位到分钟。

```
awk 'BEGIN{
    print mktime("2019 2 23 12 32 65") | "xargs -i date -d@{} +\"%F %T\""
}'
2019-02-23 12:33:05
```

如果某部位的数值为负数，则表示在此时间点基础上减几。例如：

```
# 2019-02-23 12:00:59基础上减1分钟
$ awk 'BEGIN{print mktime("2019 2 23 12 -1 59") | "xargs -i date -d@{} +\"%F %T\""}'
2019-02-23 11:59:59

# 2019-02-23 00:32:59基础上减1小时
$ awk 'BEGIN{print mktime("2019 2 23 -1 32 59") | "xargs -i date -d@{} +\"%F %T\""}'
2019-02-22 23:32:59
```

strftime

```
strftime([format [, timestamp [, utc-flag] ] ])
```

将指定的时间戳tiemstamp按照给定格式format转换为字符串并返回这个字符串。

如果省略timestamp，则对当前系统时间进行格式化。

如果省略format，则采用PROCINFO["strftime"]的格式，其默认格式为`%a %b %e %H:%M%:S %Z %Y`。该格式对应于Shell命令`date`的默认输出结果。

```
$ awk 'BEGIN{print strftime()}'
Wed Oct 30 00:20:01 CST 2014

$ date
Wed Oct 30 00:20:04 CST 2014

$ awk 'BEGIN{print strftime(PROCINFO["strftime"], systime())}'
Wed Oct 30 00:24:00 CST 2014
```

支持的格式包括：

```
 %a 星期几的缩写，如Mon、Sun Wed Fri
 %A 星期几的英文全名，如Monday
 %b 月份的英文缩写，如Oct、Sep
 %B 月份的英文全名，如February、October
 %C 2位数的世纪，例如1970对应的世纪是19
 %y 2位数的年份(00-99)，通过年份模以100取得，例如2019/100的余数位19
 %Y 四位数年份(如2015)
 %m 月份(01-12)
 %j 年中天(001-366)
 %d 月中天(01-31)
 %e 空格填充的月中天
 %H 24小时制的小时(00-23)
 %I 12小时制的小时(01-12)
 %p 12小时制时的AM/PM
 %M 分钟数(00-59)
 %S 秒数(00-60)
 %u 数值的星期几(1-7)，1表示星期一
 %w 数值的星期几(0-6)，0表示星期日
 %W 年中第几周(00-53)
 %z 时区偏移，格式为"+HHMM"，如"+0800"
 %Z 时区偏移的英文缩写，如CST

 %k 24小时制的时间(0-23)，1位数的小时使用空格填充
 %l 12小时制的时间(1-12)，1位数的小时使用空格填充
 %s 秒级epoch

 ##### 特殊符号
 %n 换行符
 %t 制表符
 %% 百分号%

 ##### 等价写法:
 %x 等价于"%A %B %d %Y"
 %F 等价于"%Y-%m-%d"，用于表示ISO 8601日期格式
 %T 等价于"%H:%M:%S"
 %X 等价于"%T"
 %r 12小时制的时间部分格式，等价于"%I:%M:%S %p"
 %R 等价于"%H:%M"
 %c 等价于"%A %B %d %T %Y"，如Wed 30 Oct 2015 12:34:48 AM CST
 %D 等价于"%m/%d/%y"
 %h 等价于"%b"
```

例如：

```
$ awk 'BEGIN{print strftime("%s", mktime("2077 11 12 10 23 32"))}'
3403909412

$ awk 'BEGIN{print strftime("%F %T %Z", mktime("2077 11 12 10 23 32"))}'
2077-11-12 10:23:32 CST

$ awk 'BEGIN{print strftime("%F %T %z", mktime("2077 11 12 10 23 32"))}'
2077-11-12 10:23:32 +0800
```

将日期时间字符串转换为时间：strptime1()

例如：

```
2019-11-11T03:42:42+08:00
```

1.将日期时间字符串中的年月日时分秒全都单独保存起来
2.将年月日时分秒构建成mktime()的字符串格式"YYYY MM DD HH mm SS"
3.使用mktime()可以构建出时间点

```
function strptime(str,    time_str,arr,Y,M,D,H,m,S){
    time_str = gensub(/[-T:+]/," ","g",str)
    split(time_str, arr, " ")
    Y = arr[1]
    M = arr[2]
    D = arr[3]
    H = arr[4]
    m = arr[5]
    S = arr[6]
    # mktime失败返回-1
    return mktime(sprintf("%d %d %d %d %d %d", Y,M,D,H,m,S))
}

BEGIN{
  str = "2019-11-11T03:42:42+08:00"
  print strptime(str)
}
```

将日期时间字符串转换为时间：strptime2()

下面是更难一点的，月份使用的是英文或英文缩写，日期时间分隔符也比较特殊。

```
Sat 26. Jan 15:36:24 CET 2013
```

```
function strptime(str,     time_str,arr,Y,M,D,H,m,S){
    time_str = gensub(/[.:]+/, " ", "g", str)
    split(time_str, arr, " ")
    Y = arr[8]
    M = month_map(arr[3])
    D = arr[2]
    H = arr[4]
    m = arr[5]
    S = arr[6]
```

```
10      return mktime(sprintf("%d %d %d %d %d %d", Y,M,D,H,m,S))
11  }
12
13  function month_map(str,   mon){
14      # mon = substr(str,1,3)
15      # return (((index("JanFebMarAprMayJunJelAugSepOctNovDec", mon)-1)/3)+1)
16      mon["Jan"] = 1
17      mon["Feb"] = 2
18      mon["Mar"] = 3
19      mon["Apr"] = 4
20      mon["May"] = 5
21      mon["Jun"] = 6
22      mon["Jul"] = 7
23      mon["Aug"] = 8
24      mon["Sep"] = 9
25      mon["Oct"] = 10
26      mon["Nov"] = 11
27      mon["Dec"] = 12
28      return mon[str]
29  }
30
31  BEGIN{
32      str = "Sat 26. Jan 15:36:24 CET 2013"
33      print strptime(str)
34  }
```

数据类型内置函数

- `isarray(var)`：测试var是否是数组，返回1(是数组)或0(不是数组)
- `typeof(var)`：返回var的数据类型，有以下可能的值：
  - "array"：是一个数组
  - "regexp"：是一个真正表达式类型，强正则字面量才算是正则类型，如`@/a.*ef/`
  - "number"：是一个number
  - "string"：是一个string
  - "strnum"：是一个strnum，参考**strnum类型**
  - "unassigned"：曾引用过，但未赋值，例如"print f;print typeof(f)"
  - "untyped"：从未引用过，也从未赋值过

几个常见的gawk扩展

使用扩展的方式：

```
1   awk -l ext_name 'BEGIN{}{}END{}'
2   awk '@load "ext_name";BEGIN{}{}END{}'
```

1.文件相关的扩展

awk和文件相关的扩展是"filefuncs"。

它支持chdir()、stat()函数。

2.文件名匹配扩展

"fnmatch"扩展提供文件名通配。

```
@load "fnmatch"
result = fnmatch(pattern, string, flags)
```

3.多进程扩展

"fork"扩展提供多进程相关功能。

```
@load "fork"

pid = fork()
创建一个子进程，对子进程返回值为0，对父进程返回值为子进程的PID，返回-1表示错误。
在子进程中，PROCINFO["pid"]和PROCINFO["ppid"]会随之更新。

ret = waitpid(pid)
等待某个子进程退出。awk的waitpid是非阻塞的，如果等待的进程还未退出，则返回值为0，等待的进程已经退出，则返回该进程pid。

ret = wait()
等待任意一个子进程退出。wait()是阻塞的，必须等待到一个子进程退出，同时返回该子进程PID。
```

例如：

```
awk '
    @load "fork"
    BEGIN{
        if( (pid=fork()) == 0 ){
            print "Child Process"
            print "CHILD PID: "PROCINFO["pid"]
            print "CHILD PPID: "PROCINFO["ppid"]
            system("sleep 1")
        } else {
            while(waitpid(pid) == 0){
                system("sleep 1")
            }
            print "Parent PID: "PROCINFO["pid"]
            print "Parent PPID: "PROCINFO["ppid"]
            print "Parent Process"
        }
    }
'
```

4.日期时间扩展

"time"扩展提供了两个函数。mktime systime strftime

```
@load "time"

the_time = gettimeofday()
    获取当前系统时间，以浮点数方式返回，精确的浮点小数位由操作系统决定

res = sleep(sec)
    睡眠指定时间，可以是小数秒
```

```
$ awk '@load "time";BEGIN{printf "%.9f\n",gettimeofday()}'
1572422333.740148067

$ awk '@load "time";BEGIN{printf "%.19f\n",gettimeofday()}'
1572422391.5475890636444091797
```

睡眠是很好用的功能：

```
awk '@load "time";BEGIN{sleep(1.2);print "hello world"}'
```

awk实战示例

去除重复行

```
 abc
 def
 ghi
 abc
 ghi
 xyz

 mnopq
 abc

```

```
# 1.会打乱顺序
awk '{arr[$0]++}END{for(i in arr){print i}}' x.log

# 2.保证顺序
awk '!arr[$0]++{print}' x.log
```

统计行出现的次数

```
$ awk '{arr[$0]++}END{for(i in arr){print i"->"arr[i]}}' x.log
->1
def->1
mnopq->1
abc->3
ghi->2
xyz->1
```

统计单词出现的次数

```
awk '{for(i=1;i<=NF;i++){arr[$i]++}}END{for(idx in arr){print idx": "arr[idx]}}
' x.log
```

统计TCP连接状态数量

```
$ netstat -tnap
Proto Recv-Q Send-Q Local Address   Foreign Address  State       PID/Program name
tcp        0      0 0.0.0.0:22      0.0.0.0:*       LISTEN      1139/sshd
tcp        0      0 127.0.0.1:25    0.0.0.0:*       LISTEN      2285/master
tcp        0     96 192.168.2.17:22 192.168.2.1:2468 ESTABLISHED 87463/sshd
tcp        0      0 192.168.2017:22 192.168.201:5821 ESTABLISHED 89359/sshd
tcp6       0      0 :::3306         :::*            LISTEN      2289/mysqld
tcp6       0      0 :::22           :::*            LISTEN      1139/sshd
tcp6       0      0 ::1:25          :::*            LISTEN      2285/master
```

统计得到的结果：

```
5: LISTEN
2: ESTABLISHED
```

思路： 1.将出现的状态作为数组索引
2.将出现的状态次数作为值，每出现一次就+1

```
 netstat -tnap |\
 awk '
     /^tcp /{
         arr[$6]++
     }
     END{
         for(state in arr){
             print arr[state] ": " state
         }
     }
 '
```

一行式：

```
netstat -tna | awk '/^tcp /{arr[$6]++}END{for(state in arr){print arr[state] ": " state}}'
netstat -tna | /usr/bin/grep 'tcp ' | awk '{print $6}' | sort | uniq -c
```

根据字段取最大值

已知文件a.txt，第一列是文件名，第二列是版本号，打印出每个文件最大的版本号一行。

```
$ cat a.txt
file 100
dir 11
file 100
dir 11
file 102
dir 112
file 120
dir 119
```

```
$ awk '{if(code[$1]<$2){code[$1]=$2}}END{for(i in code){print i,code[i]} }' a.txt
file 120
dir 119
```

去掉 /**/ 注释内容

示例数据：

```
/*AAAAAAAAAA*/
1111
222

/*aaaaaaaaa*/
32323
12341234
12134 /*bbbbbbbbbb*/ 132412

14534122
/*
    ccccccccc
    ccccccccc
*/
yyyyyy /*ddddddddddd
    ccccccccc
xxxxx*/ zzzzz
5642341
```

```
index($0, "/*"){
  # 同行是否有*/
  if(index($0,"*/")){
```

```
    print gensub("^(.*)/\\*.*\\*/(.*)","\\1\\2","g",$0)
  } else {
    # 去掉/*后面的内容，输出/*前面的内容
    print gensub("^(.*)/\\*.*","\\1","g",$0)

    # 读取下一行，直到遇到*/，并输出*/后面的内容
    while( (getline var) > 0 ){
      if(index(var, "*/")){
        print gensub("^.*\\*/(.*)","\\1","g",var)
        break
      }
    }
  }
}

!index($0, "/*"){
  print
}
```

前后段落判断

从如下类型的文件中，找出false段的前一段包含i-order的段，同时输出这两段。

```
2019-09-12 07:16:27 [-][
  'data' => [
    'http://192.168.100.20:2800/api/payment/i-order',
  ],
]
2019-09-12 07:16:27 [-][
  'data' => [
    false,
  ],
]
2019-09-21 07:16:27 [-][
  'data' => [
    'http://192.168.100.20:2800/api/payment/i-order',
  ],
]
2019-09-21 07:16:27 [-][
  'data' => [
    'http://192.168.100.20:2800/api/payment/i-user',
  ],
]
2019-09-17 18:34:37 [-][
  'data' => [
    false,
  ],
]
```

```
BEGIN{
  RS="]\n"
  ORS=RS
}
{
  if(/false/ && prev ~ /i-order/){
    print prev
    print
  }
  prev=$0
}
```

行列转换

原始数据：

```
ID  name    gender  age  email phone
1   Bob     male    28   abc   *2233
2   Alice   female  24   def   *3849
3   Tony    male    21   aaa   *1487
4   Kevin   male    21   bbb   *4239
5   Alex    male    18   ccc   *5859
```

得到结果：

```
ID      1       2       3       4       5
name    Bob     Alice   Tony    Kevin   Alex
gender  male    female  male    male    male
age     28      24      21      21      18
email   abc     def     aaa     bbb     ccc
phone   *2233   *3849   *1487   *4239   *5859
```

思路：将每行的各个字段信息都追加到数组的一项中。

代码：

```
{
    for(i=1;i<=NF;i++){
        arr[i] = arr[i]"\t"$i
    }
}

END{
    for(i in arr){
        print arr[i]
    }
}
```

结果：

```
ID      1       2       3       4       5
name    Bob     Alice   Tony    Kevin   Alex
gender  male    female  male    male    male
age     28      24      21      21      18
email   abc     def     aaa     bbb     ccc
phone   *2233   *3849   *1487   *4239   *5859
```

缺点：不能处理空字段。

为了处理可能的空字段，使用FIELDWIDTHS。

例如：

```
ID  name    gender  age  email phone
1   Bob     male    28   abc   *2233
2   Alice   female  24   def   *3849
3   Tony            21   aaa   *1487
4   Kevin   male    21   bbb   *4239
5   Alex    male    18         *5859
```

```
BEGIN{
  FIELDWIDTHS = "4 8 8 5 6 *"
}

{
  for(i=1;i<=NF;i++){
    # 初始赋值时，不要追加制表符
    if(typeof(arr[i]) == "unassigned"){
      arr[i] = gensub(/ +$/,"","g",$i)
    }else {
      arr[i] = arr[i]"\t"gensub(/ +$/,"","g",$i)
    }
  }
}

END{
  for(i=1;i<=NF;i++){
    print arr[i]
  }
}
```

结果：

```
ID      1       2       3       4       5
name    Bob     Alice   Tony    Kevin   Alex
gender  male    female          male    male
age     28      24      21      21      18
email   abc     def     aaa     bbb
phone   *2233   *3849   *1487   *4239   *5859
```

筛选日志时间实战案例

当使用grep/sed/awk按时间来筛选日志时，筛选要求的时间精确度越低，筛选越方便，时间精确度越高，筛选越麻烦。

例如，想要筛选2019年11月9号及之后的日志，它精确到天，只需使用`2019 11 9`作为筛选条件即可。

```
awk '/2019-11-09/{print;while(getline > 0){print}}' a.txt
sed -nr '/2019-11-09/,$p' a.txt
```

但如果要求筛选2019-11-09 06:12:31秒之后的日志，那就难了，因为这个时间点的31秒又没有日志是不确定的，甚至06点的日志有没有也是不确定的。甚至更难一点的，要求筛选指定时间范围内的日志，难于上青天。这时候除了用perl/ruby一行式或写编程脚本来筛选，已经没有好办法。

好在，awk为我们提供了相关的解决方案，awk支持mktime()，它能够根据给定的时间字符串构建出一个秒级的epoch时间戳。如此一来，通过比较数值大小即可确定时间范围。

现在有两种日志文件，access.log文件中日志格式：

```
1.1.1.1 - - [2019-11-11T02:11:39+08:00] "GET / HTTP/1.1" 301 169 "-" "Mozilla/5.0 zgrab/0.x" "-"
```

access1.log文件日志格式：

```
1.1.1.1 - - [11/Nov/2019:02:11:39+08:00] "GET / HTTP/1.1" 301 169 "-" "Mozilla/5.0 zgrab/0.x" "-"
```

先处理access.log：

```
BEGIN{
    tmp_time = mktime("2019 11 9 6 12 31")
}

{
    match($0,/\[(2019.*)\]/,arr)
    # arr[1] is the datetime

    # dt format "2019-11-11T02:11:39+0800"
      mytime = strptime1(arr[1])
        if( mytime >= tmp_time){
            count1++
        }
}

END{
    print count1
}

# for dt format "2019-11-11T02:11:39+0800"
function strptime1(dt_str){
    split(dt_str,dt, "[-:+T]")
    Y = dt[1]
    M = dt[2]
    D = dt[3]
    H = dt[4]
    m = dt[5]
    S = dt[6]
```

```
    return mktime(sprintf("%d %d %d %d %d %d",Y,M,D,H,m,S))
}
```

再处理access1.log:

```
BEGIN{
    tmp_time = mktime("2019 11 9 6 12 31")
}

{
    match($0,/\[(.*)\]/,arr)
    # arr[1] is the datetime

    # dt format "11/Nov/2019:02:11:39+08:00"
    if(strptime(arr[1]) >= tmp_time){
        #print
        count++
    }
}

END{
    print count
}


# for dt format "11/Nov/2019:02:11:39+08:00"
function strptime(dt_str,   dt,Y,M,D,H,m,S){
    split(dt_str,dt, /[:+/]/)
    Y = dt[3]
    M = mon_map(dt[2])
    D = dt[1]
    H = dt[4]
    m = dt[5]
    S = dt[6]
    return mktime(sprintf("%d %d %d %d %d %d",Y,M,D,H,m,S))
}

function mon_map(mon,    str) {
    str = "JanFebMarAprMayJunJulAugSepOctNovDec"
    return ( ( ( index(str, mon) - 1 ) / 3 ) + 1)
}
```

将两种情况结合在一起:

```
awk '
BEGIN{
    tmp_time = mktime("2019 11 9 6 12 31")
}

{
    match($0,/\[(.*)\]/,arr)
    # arr[1] is the datetime

    # dt format "2019-11-11T02:11:39+0800"
```

```
    if(ARGIND == 1){
        if(strptime1(arr[1]) >= tmp_time){
            #print
            count1++
        }
    }

    # dt format "11/Nov/2019:02:11:39+08:00"
    if(ARGIND == 2){
        if(strptime2(arr[1]) >= tmp_time){
            #print
            count2++
        }
    }
}

END{
    print "count1: " count1
    print "count2: " count2
}

# for dt format "2019-11-11T02:11:39+0800"
function strptime1(dt_str,   dt,Y,M,D,H,m,S){
    split(dt_str,dt, /[-:+T]/)
    Y = dt[1]
    M = dt[2]
    D = dt[3]
    H = dt[4]
    m = dt[5]
    S = dt[6]
    return mktime(sprintf("%d %d %d %d %d %d",Y,M,D,H,m,S))
}

# for dt format "11/Nov/2019:02:11:39+08:00"
function strptime2(dt_str,   dt,Y,M,D,H,m,S){
    split(dt_str,dt, /[:+/]/)
    Y = dt[3]
    M = mon_map(dt[2])
    D = dt[1]
    H = dt[4]
    m = dt[5]
    S = dt[6]
    return mktime(sprintf("%d %d %d %d %d %d",Y,M,D,H,m,S))
}

function mon_map(mon,    str) {
    str = "JanFebMarAprMayJunJulAugSepOctNovDec"
    return ( ( ( index(str, mon) - 1 ) / 3 ) + 1)
}

' access.log access1.log
```