awk 实例

李会民
lihm@qibebt.ac.cn

中国科学院青岛生物能源与过程研究所 超级计算中心

2009 年 11 月

awk 简介

- awk 名称得于它的创始人 Alfred Aho、Peter Weinberger 和 Brian Kernighan 姓氏的首字母
- awk 是 Unix/Linux 环境中现有的功能最强大的数据处理引擎之一
- awk 设计思想来源于 SNOBOL4、sed、Marc Rochkind 设计的有效性语言、语言工具 yacc 和 lex，还从 C 语言中获取了一些优秀的思想
- awk 具有完全属于其本身的语法，在很多方面类似 Unix shell 编程语言
- awk 提供了极其强大的功能：可以进行正则表达式的匹配，样式装入、流控制、数学运算符、进程控制语句甚至于内置的变量和函数
- awk 的目的是用于文本处理，只要在输入数据中有模式匹配，就执行一系列指令，扫描文件中的每一行，查找与命令行中所给定内容相匹配的模式，如发现匹配内容，则进行下一个编程步骤，如果找不到匹配内容，则继续处理下一行

awk 语法

awk 命令的语法基本是：

```
awk '{pattern + action}' {filenames}
```

- pattern 表示 awk 在数据中查找的内容，支持正则表达式
- action 是在找到匹配内容时所执行的一系列命令
- 花括号（{}）不需要在程序中始终出现，但它们用于根据特定的模式对一系列指令进行分组
- pattern 和 action 整体之外需用 '' 括起来

gawk 是 awk 的 gnu 版本，本培训以 gawk 为主，其余 awk 实现版本略有不同

第一个 awk

```
awk '{ print }' /etc/passwd
```

- 执行此 awk 命令时，它依次对输入文件 /etc/passwd 文件中的每一行执行 **print** 命令，所有输出都发送到标准输出（stdout）
- 花括号用于将几块代码组合到一起，类似于 C 语言
- 在 awk 中，如只出现 **print** 命令，那么将打印当前行的全部内容

```
awk '{ print $0 }' /etc/passwd
```

$0 变量表示整个当前行，所以 **print** 和 **print** $0 的作用完全一样
以下代码可输出与输入数据完全无关（其实输入文件的行数决定输出的行数）的数据：

```
awk '{ print "hiya" }' /etc/passwd
```

多个字段

- awk 善于处理分成多个逻辑字段的文本，第 n 字段用 $n 引用，其中 $0 表示整行
- 调用 awk 时，默认使用空格做为分隔符，利用 −F 选项可来指定其它字符做为字段分隔符
- \ 表示转义，比如 \t 表示制表符

```
awk −F":" '{ print "username: " $1 "\t\tuid:" $3 }' /etc/passwd
```

将产生以下输出：

```
username: halt          uid:7
username: operator      uid:11
username: root          uid:0
username: shutdown      uid:6
username: sync          uid:5
username: bin           uid:1
...
```

外部脚本

- 将脚本作为命令行自变量传递给 awk 对于单行程序来说非常简单，而对于多行程序将比较复杂
- 可在外部文件中撰写脚本，然后 awk −f 脚本文件执行，将脚本放入文本文件还可使用附加 awk 功能：

```
awk −f myscript.awk myfile.in
```

```
#myscript.awk
BEGIN {
    FS=":"
}
{ print $1 }
```

在此脚本中，字段分隔符在代码自身中指定（通过设置 FS 变量），最好在脚本自身中设置字段分隔符，这样可以少输入一个命令行自变量

awk 脚本

awk 还可类似普通的 shell 脚本

```
#!/usr/bin/awk -f
#myscript.awk
BEGIN {
    FS=":"
}
{ print $1 }
```

- 通过第一行为 *#!/usr/bin/awk −f* 设置，其中 /usr/bin/awk 为使用的 awk 的具体位置
- # 后面的为注释

其运行也可类似普通 shell 脚本运行：

```
./myscript.awk myfile.in
```

BEGIN 和 END 块

- **BEGIN** 块：通常对于每个输入行，awk 都会执行每个脚本代码块一次。在许多编程情况中，可能需要在 awk 开始处理输入文件中的文本之前执行初始化代码。对于这种情况，awk 允许您定义一个 **BEGIN** 块。awk 在开始处理输入文件之前会执行 **BEGIN** 块，它是初始化 FS 变量、打印页眉或初始化其它在程序中以后会引用的全局变量的极佳位置。
- **END** 块：awk 在处理了输入文件中的所有行之后执行这个块。通常 END 块用于执行最终计算或打印应该出现在输出流结尾的摘要信息。

正则表达式和块

awk 允许使用正则表达式，根据正则表达式是否匹配当前行来选择执行独立代码块。
只输出包含字符序列 foo 的那些行：

```
/foo/ { print }
```

只打印包含浮点数的行：

```
/[0-9]+\.[0-9]*/ { print }
```

表达式和块

可以将任意一种布尔表达式放在一个代码块之前，以控制何时执行某特定块，仅当对前面的布尔表达式求值为真时，awk 才执行代码块。
输出第一个字段等于 fred 的所有行中的第三个字段，如当前行的第一个字段不等于 fred，awk 将继续处理文件而不对当前行执行 **print** 语句：

```
$1 == "fred" { print $3 }
```

awk 提供了完整的比较运算符，包括 "==="、"<"、">"、"<="、">=" 和 "!="，还提供了分别表示“匹配”和“不匹配”的 "~" 和 "!~" 运算符。其用法是在运算符左边指定变量，在右边指定正则表达式。
只打印第五个字段包含字符序列 root 中的行的第三个字段：

```
$5 ~ /root/ { print $3 }
```

if 条件语句

awk 提供了非常好的类似于 C 语言的 **if** 语句。
使用 **if** 语句重写前一个脚本：

```
{
    if ( $5 ~ /root/ ) {
        print $3
    }
}
```

更复杂的 if 语句示例

尽管使用了复杂、嵌套的条件语句， if 语句看上去仍与相应的 C 语言语句一样：

```
{
    if ( $1 == "foo" ) {
        if ( $2 == "foo" ) {
            print "uno"
        } else {
            print "one"
        }
    } else if ($1 == "bar" ) {
        print "two"
    } else {
        print "three"
    }
}
```

if 语句实例

使用 **if** 语句还可以将代码：

```
! /matchme/ { print $1 $3 $4 }
```

转换成：

```
{
    if ( $0 !~ /matchme/ ) {
        print $1 $3 $4
    }
}
```

这两个脚本都只输出不包含 matchme 字符序列的那些行。

布尔运算符 "||"（逻辑或）和 "&&"（逻辑与）

awk 允许使用布尔运算符 " || "（逻辑或）和 "&&"（逻辑与），以创建更复杂的布尔表达式：

```
( $1 == "foo" ) && ( $2 == "bar" ) { print }
```

这个示例只打印第一个字段等于 foo 且第二个字段等于 bar 的那些行。

数值变量

awk 允许执行整数和浮点运算。通过使用数学表达式，可很方便地编写计算文件中空白行数量的脚本。

```
BEGIN { x=0 }
/^$/    { x=x+1 }
END   { print "I found " x " blank lines. :)" }
```

在 **BEGIN** 块中，将整数变量 x 初始化成零。然后 awk 每次遇到空白行时，将执行 x=x+1 语句，递增 x。处理完所有行之后，执行 **END** 块，awk 将打印出最终摘要，指出它找到的空白行数量。

字符串化变量

awk 的优点之一就是“简单和字符串化”。awk 变量“字符串化”是因为所有 awk 变量在内部都是按字符串形式存储的。同时，awk 变量是“简单的”，因为只要变量包含有效数字字符串就可以对它执行数学操作，awk 会自动处理字符串到数字的转换步骤。

```
x="1.01"
# We just set x to contain the *string* "1.01"
x=x+1
# We just added one to a *string*
print x
```

awk 将输出：

```
2.01
```

如想要对每个输入行的第一个字段乘方并加一，可以使用以下脚本：

```
{ print ($1^2)+1 }
```

如某变量不含有效数字，awk 在对数学表达式求值时会将该变量当作零处理。

运算符

awk 有完整的数学运算符集合：

- 标准的加、减、乘、除
- 指数运算符 "^"、模（余数）运算符 "%"
- 从 C 语言中借入的易于使用的赋值操作符：
  - 前后加减（i++、--foo）
  - 加 / 减 / 乘 / 除赋值运算符（+=、*=、/=、-=）
  - 模 / 指数赋值运算符（^=、%=）

字段分隔符：FS

awk 有它自己的特殊变量集合，其中一些允许调整 awk 的运行方式，而其它变量可以被读取以收集关于输入的有用信息。

- 缺省情况下，FS 设置成单一空格字符，awk 将这解释成表示“一个或多个空格或 tab”
- FS 变量可以设置 awk 要查找的字段之间的字符序列
- FS 值并没有被限制为单一字符，可以通过指定任意长度的字符模式，将它设置成正则表达式

处理由一个或多个 tab 分隔的字段，可按以下方式设置 FS：

```
FS="\t+"
```

特殊 "+" 正则表达式字符表示“一个或多个前一字符”
复杂的正则表达式也可，下面表示分隔符为 "foo" 后面跟着三个数字：

```
FS="foo[0-9][0-9][0-9]"
```

字段数量：NF

NF 变量，也叫做“字段数量”变量，awk 会自动将该变量设置成当前记录中的字段数量。可以使用 NF 变量来只显示某些输入行：

```
NF == 3 { print "this particular record has three fields : " $0 }
```

也可在条件语句中使用 NF 变量：

```
{
    if ( NF > 2 ) {
        print $1 " " $2 ":" $3
    }
}
```

记录号：NR

记录号（NR）包含当前记录的编号（第一个记录算作记录号 1）。
使用 NR 来只打印某些输入行：

```
(NR < 10 ) || (NR > 100) { print "We are on record number 1-9 or 101+" }
```

另一个示例：

```
{
    #skip header
    if ( NR > 10 ) {
        print "ok, now for the real information!"
    }
}
```

多行记录分隔符：RS

如要分析占据多行的记录，仅依靠设置 FS 是不够的，还需修改多行记录分隔符 RS 记录分隔符变量。RS 变量告诉 awk 当前记录什么时候结束，新记录什么时候开始。
“联邦证人保护计划”所涉及人员的地址列表：

Jimmy the Weasel
100 Pleasant Drive
San Francisco, CA 12345

Big Tony
200 Incognito Ave.
Suburbia, WA 67890

希望 awk 将每 3 行看作是一个独立的记录，而不是三个独立的记录，将地址的第一行看作是第一个字段 $1，街道地址看作是第二个字段 $2，城市、州和邮政编码看作是第三个字段 $3。

```
BEGIN {
    FS="\n"
    RS=""
}
```

在上面这段代码中，将 FS 设置成 "\n" 告诉 awk 每个字段都占据一行；
通过将 RS 设置成 ""，告诉 awk 每个地址记录都由空白行分隔。

多行记录实例分析

```
#address.awk
BEGIN {
    FS="\n"
    RS=""
}
{
    print $1 ", " $2 ", " $3
}
```

如脚本保存为 address.awk，地址数据存储在 address.txt，可通过输入 "awk −f address.awk address.txt" 来执行这个脚本，将产生以下输出：

```
Jimmy the Weasel, 100 Pleasant Drive, San Francisco, CA 12345
Big Tony, 200 Incognito Ave., Suburbia, WA 67890
```

输出字段分隔符 OFS 和输出记录分隔符 ORS

- 输出字段分隔符 OFS 和输出记录分隔符 ORS 可用于设置 **print** 输出时的字段分隔符和多行记录分隔符
- 缺省情况：OFS 设置成 " "（单个空格），ORS 设置成换行（"\n"）
- 如想使输出的间隔翻倍，可将 ORS 设置成 "\n\n"；如想用单个空格分隔记录（而不换行），将 ORS 设置成 " "

```
#address.awk
BEGIN {
    FS="\n"
    RS=""
    OFS=", "
}
{
    print $1, $2, $3
}
```

输出：

```
Jimmy the Weasel, 100 Pleasant Drive, San Francisco, CA 12345
```

将多行转换成用 tab 分隔的格式

以上 address.awk 只适合于三行的地址，如果 awk 遇到以下地址，将丢掉第四行，并且不打印该行：

Cousin Vinnie

Vinnie's Auto Shop

300 City Alley

Sosueme, OR 76543

将多行转换成用 tab 分隔的格式

适合具有任意多字段的地址的 address.awk 版本

```
BEGIN {
    FS="\n"
    RS=""
    ORS=""
}
{
        x=1
        while ( x<NF ) {
                print $x "\t"
                x++
        }
        print $NF "\n"
}
```

- 将字段分隔符 FS 设置成 "\n"，将记录分隔符 RS 设置成 ""，awk 可以象以前一样正确分析多行地址。
- 将输出记录分隔符 ORS 设置成 ""，它将使 **print** 语句在每个调用结尾不输出新行，这意味着如希望任何文本从新的一行开始，那么需明确写入 **print** "\n"。
- 在主代码块中，创建了一个变量 x 来存储正在处理的当前字段的编号，起初设置成1。
- 使用 **while** 循环（等同于 C 语言中的 **while** 循环），对于所有记录（最后一个记录除外）重复打印记录和 tab 字符。
- 最后，打印最后一个记录和换行

程序输出如下：

```
Jimmy the Weasel    100 Pleasant Drive    San Francisco, CA 12345
Big Tony    200 Incognito Ave.    Suburbia, WA 67890
Cousin Vinnie    Vinnie's Auto Shop    300 City Alley    Sosueme, OR 76543
```

循环结构：while

- awk 的 **while** 循环结构，等同于相应的 C 语言 **while** 循环
- 当遇到普通 **while** 循环时，如条件为假，将永远不执行该循环
- "**do…while**" 循环与类似于其它语言中的 "repeat … until" 循环，与一般的 **while** 循环不同，将在代码块之后对条件求值，因此永远都至少执行一次

do…while 示例：

```
{
    count=1
    do {
        print "I get printed at least once no matter what"
    } while ( count != 1 )
}
```

for 循环

awk 的 **for** 循环，类似 **while** 循环，也等同于 C 语言的 **for** 循环：

```
for ( initial assignment; comparison; increment ) {
    code block
}
```

以下是一个简短示例：

```
for ( x = 2; x <= 4; x++ ) {
    print "iteration",x
}
```

此段代码将打印：

```
iteration 2
iteration 3
iteration 4
```

跳出循环：break

awk 提供 **break** 语句，可跳出循环，以更好地控制 awk 的循环结构。

```
while (1) {
    print "forever and ever ..."
}
```

因为 1 永远代表是真，上述 **while** 循环将永远运行下去。
以下是一个只执行十次的循环：

```
x=1
while(1) {
    print "iteration",x
    if ( x == 10 ) {
        break
    }
    x++
}
```

break 语句用于“逃出”最深层的循环，**break** 使循环立即终止，并继续执行循环代码块后面的语句。

返回循环开始：continue

continue 语句可以跳过此循环中剩余的语句返回循环开始，适合各种 awk 迭代循环。

```
x=1
while (1) {
    if ( x == 4 ) {
        x++
        continue
    }
    print "iteration",x
    if ( x > 20 ) {
        break
    }
    x++
}
```

以上代码打印 "iteration 1" 到 "iteration 21"，但 "iteration 4" 除外。

```
for ( x=1; x<=21; x++ ) {
    if ( x == 4 ) {
        continue
    }
    print "iteration",x
}
```

数组

awk 可使用数组，数组无需指明大小，数组下标通常从 1 而不是 0 开始，也不需要连续的数字序列（可定义 myarr[1] 和 myarr[1000]，但不定义其它所有元素），数组可不提前定义直接使用

```
myarray[1]="jim"
myarray[2]=456
```

awk 遇到第一个赋值语句时，它将创建 myarray，并将元素 myarray[1] 设置成 "jim"。执行了第二个赋值语句后，数组就有两个元素了。

数组迭代

awk 有一个便利的机制来迭代数组元素，如下所示：

```
for ( x in myarray ) {
    print myarray[x]
}
```

这段代码将打印数组 myarray 中的每一个元素。当对于 **for** 使用这种特殊的 "in" 形式时，awk 将 myarray 的每个现有下标依次赋值给 x，每次赋值以后都循环一次循环代码。但它有一个缺点：当 awk 在数组下标之间轮转时，它不会依照任何特定的顺序。
套用 Forrest Gump 的话来说，迭代数组内容就像一盒巧克力– 您永远不知道将会得到什么。

数组下标字符串化：关联数组

awk 实际上以字符串格式来存储数字值，awk 并没限制使用纯整数下标，可使用字符串下标，如 myarr["name"]，这种数组称为关联数组。

```
myarr["1"]="Mr. Whipple"
print myarr["1"]
print myarr[1]

myarr["name"]="Mr. Whipple"
print myarr["name"]
```

awk 将 myarr["1"] 和 myarr[1] 看作数组的同一个元素，将打印 "Mr. Whipple"

数组工具

awk 提供了一些实用功能有助于使数组变得更易于管理
使用 **delete** 删除数组元素：

```
delete fooarray[1]
```

使用特殊的 "in" 布尔运算符查看是否存在某个特定数组元素

```
if ( 1 in fooarray ) {
    print "Ayep! It's there."
} else {
    print "Nope! Can't find it."
}
```

格式化输出

awk 提供了 **printf()** 和 **sprintf()**，**printf()** 会将格式化字符串打印到 stdout，而 **sprintf()** 则返回可以赋值给变量的格式化字符串。
以下是 **printf()** 和 **sprintf()** 的样本代码，几乎与 C 语言完全相同：

```
x=1
b="foo"
printf("%s got a %d on the last test\n","Jim",83)
myout=sprintf("%s-%d",b,x)
print myout
```

此代码将打印：

```
Jim got a 83 on the last test
foo-1
```

字符串函数

awk 有许多字符串函数，以方便对字符串进行处理，下面仅对常用的几个做解释，更多的请参看 awk 手册。
以下以 mystring 的内容为 "How are you doing today?" 为例

字符串的长度：length()

返回字符串的长度 **length()**：

```
print length(mystring)
```

awk 会打印：

```
22
```

子字符串在另一个字符串中出现的位置：index()

index() 函数，将返回子字符串在另一个字符串中出现的位置，如果没有找到该字符串则返回0：

```
print index(mystring,"you")
```

awk 会打印：

```
9
```

转换大小写：tolower() 和 toupper()

tolower() 和 toupper() 将返回字符串并且将所有字符分别转换成小写或大写，不会修改原始字符串。

```
print tolower(mystring)
print toupper(mystring)
print mystring
```

将产生以下输出：

```
how are you doing today?
HOW ARE YOU DOING TODAY?
How are you doing today?
```

字符串中选择子串：substr()

字符串中选择子串：substr()：

mysub=**substr**(mystring,startpos,maxlen)

mystring 应该是要从中抽取子串的字符串变量或文字字符串。startpos 应该设置成起始字符位置，maxlen 应该包含要抽取的字符串的最大长度，如果 **length**(mystring) 比 startpos+maxlen 短，那么得到的结果就会被截断。**substr**() 不会修改原始字符串，而是返回子串。

print substr(mystring,9,3)

awk 将打印：

---

you

返回匹配的起始位置：match()

- **match()** 与 **index()** 非常相似，与 **index()** 的区别在于它并不搜索子串，它搜索的是正则表达式
- **match()** 函数将返回匹配的起始位置，如没有找到匹配，则返回 0
- **match()** 还将设置两个变量，RSTART（包含返回第一个匹配的位置的值）和 RLENGTH（指定它占据的字符跨度，如没有找到匹配，则返回 -1）。通过使用 RSTART、RLENGTH、**substr()** 和一个小循环，可以轻松地迭代字符串中的每个匹配。

match() 调用示例：

```
print match(mystring,/you/), RSTART, RLENGTH
```

将打印：

```
9 9 3
```

字符串替换：sub() 和 gsub()

字符串替换函数 **sub()** 和 **gsub()**，修改原始字符串

```
sub(regexp,replstring,mystring)
```

- 调用 **sub()** 时，它将在 mystring 中匹配 regexp 的第一个字符序列，并且用 replstring 替换该序列。
- **sub()** 和 **gsub()** 用相同的自变量；唯一的区别是 **sub()** 将替换第一个 regexp 匹配（如果有的话），**gsub()** 将执行全局（global）替换，换出字符串中的所有匹配。

sub() 和 **gsub()** 调用示例：

```
sub(/o/,"O",mystring)
print mystring
mystring="How are you doing today?"
gsub(/o/,"O",mystring)
print mystring
```

必须将 mystring 复位成其初始值，因为第一个 **sub()** 调用直接修改了 mystring。此代码将输出：

```
HOw are you doing today?
HOw are yOu dOing tOday?
```

字符串分割：split()

split() 函数对字符串进行分割，并将各部分放到整数下标的数组中。
split() 调用示例：

```
numelements=split("Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec",mymonths,",")
```

调用 **split()** 时：

- 第一个自变量包含要切开文字字符串或字符串变量
- 在第二个自变量中，应该指定 **split()** 将填入片段部分的数组名称
- 在第三个元素中，指定用于切开字符串的分隔符
- **split()** 返回时，它将返回分割的字符串元素的数量
- **split()** 将每一个片段赋值给下标从 1 开始的数组

```
print mymonths[1],mymonths[numelements]
```

将打印：

Jan Dec

特殊字符串形式

调用 **length()**、**sub()** 或 **gsub()** 时，可以去掉最后一个自变量，这样将对 $0（整个当前行）应用函数调用。
要打印文件中每一行的长度：

```
{
    print length()
}
```

调用系统命令：system()

利用 **system()** 可以调用系统命令
打印系统时间：

```
system("date")
```

```
Tue Nov 10 23:14:52 CST 2009
```

自定义函数

awk 中利用关键词 **function** 可以自定义函数，然后在脚本中调用，自定义函数中的变量是共用的

```
function monthdigit(mymonth) {
    return (index(months,mymonth)+3)/4
}
 ...
print monthdigit($2)
```

参考资料

本文基于 Daniel Robbins 的《awk 实例》：
http://www-900.ibm.com/developerWorks/cn/linux/shell/awk/awk-1/
Daniel Robbins 建议参考：

- O'Reilly 的 sed & awk, 2ndEdition 是极佳选择：
  http://www.oreilly.com/catalog/sed2/
- comp.lang.awkFAQ 包含许多附加 awk 链接：
  http://www.faqs.org/faqs/computer-lang/awk/faq/
- Patrick Hartigan 的 awk tutorial 还包括了实用的awk 脚本：
  http://sparky.rice.edu/~hartigan/awk.html
- The GNU Awk User's Guide：http://www.gnu.org/manual/gawk/gawk.html

联系方式

- 中科院青岛生物能源与过程研究所：
  - http://www.qibebt.cas.cn
- 青能所超算中心：
  - http://124.16.151.186，当前 IP 地址
  - http://scc.qibebt.cas.cn，将来的域名
- 李会民：
  - http://staff.ustc.edu.cn/~hmli/
  - lihm@qibebt.ac.cn
  - hmli@ustc.edu.cn
  - li.huimin@gmail.com
  - 0532-80662613