一. AWK入门指南

Awk是一种便于使用且表达能力强的程序设计语言，可应用于各种计算和数据处理任务。本章是个入门指南，让你能够尽快地开始编写你自己的程序。第二章将描述整个语言，而剩下的章节将向你展示如何使用Awk来解决许多不同方面的问题。纵观全书，我们尽量选择了一些对你有用、有趣并且有指导意义的实例。

1.1 起步

有用的awk程序往往很简短，仅仅一两行。假设你有一个名为 *emp.data* 的文件，其中包含员工的姓名、薪资（美元/小时）以及小时数，一个员工一行数据，如下所示：

Beth	4.00	0
Dan	3.75	0
kathy	4.00	10
Mark	5.00	20
Mary	5.50	22
Susie	4.25	18

现在你想打印出工作时间超过零小时的员工的姓名和工资（薪资乘以时间）。这种任务对于awk来说就是小菜一碟。输入这个命令行就可以了：:

```
awk '$3 >0 { print $1, $2 * $3 }' emp.data
```

你应该会得到如下输出：

Kathy 40
Mark 100
Mary 121
Susie 76.5

该命令行告诉系统执行引号内的awk程序，从输入文件 *emp.data* 获取程序所需的数据。引号内的部分是个完整的awk程序，包含单个模式-动作语句。模式 *$3>0* 用于匹配第三列大于0的输入行，动作:

```
{ print $1, $2 * $3 }
```

打印每个匹配行的第一个字段以及第二第三字段的乘积。

如果你想打印出还没工作过的员工的姓名，则输入命令行：:

```
awk '$3 == 0 { print $1 }' emp.data
```

这里，模式 *$3 == 0* 匹配第三个字段等于0的行，动作:

```
{ print $1 }
```

打印该行的第一个字段。

当你阅读本书时，应该尝试执行与修改示例程序。大多数程序都很简短，所以你能快速理解awk是如何工作的。在Unix系统上，以上两个事务在终端里看起来是这样的：

$ awk '$3 > 0 { print $1, $2 * $3 }' emp.data
Kathy 40
Mark 100
Mary 121
Susie 76.5
$ awk '$3 == 0 { print $1 }' emp.data
Beth
Dan
$

行首的 *$* 是系统提示符，也许在你的机器上不一样。

AWK程序的结构

让我们回头看一下到底发生了什么事情。上述的命令行中，引号之间的部分是awk编程语言写就的程序。本章中的每个awk程序都是一个或多个模式-动作语句的序列：

pattern { action }
pattern { action }
...

awk的基本操作是一行一行地扫描输入，搜索匹配任意程序中模式的行。词语“匹配”的准确意义是视具体的模式而言，对于模式 *$3 >0* 来说，意思是“条件为真”。

每个模式依次测试每个输入行。对于匹配到行的模式，其对应的动作（也许包含多步）得到执行，然后读取下一行并继续匹配，直到所有的输入读取完毕。

上面的程序都是模式与动作的典型示例。：

```
$3 == 0 { print $1 }
```

是单个模式-动作语句；对于第三个字段为0的每行，打印其第一个字段。

模式-动作语句中的模式或动作（但不是同时两者）都可以省略。如果某个模式没有动作，例如：：

```
$3 == 0
```

那么模式匹配到的每一行（即，对于该行，条件为真）都会被打印出来。该程序会打印 *emp.data* 文件中第三个字段为0的两行

Beth 4.00 0
Dan 3.75 0

如果有个没有模式的动作，例如：：

```
{ print $1 }
```

那么这种情况下的动作会打印每个输入行的第一列。

由于模式和动作两者任一都是可选的，所以需要使用大括号包围动作以区分于其他模式。

执行AWK程序

执行awk程序的方式有多种。你可以输入如下形式的命令行：：

```
awk 'program' input files
```

从而在每个指定的输入文件上执行这个program。例如，你可以输入：：

```
awk '$3 == 0 { print $1 }' file1 file2
```

打印file1和file2文件中第三个字段为0的每一行的第一个字段。

你可以省略命令行中的输入文件，仅输入：：

```
awk 'program'
```

这种情况下，awk会将program应用于你在终端中接着输入的任意数据行，直到你输入一个文件结束信号（Unix系统上为control-d）。如下是Unix系统的一个会话示例：

$ awk '$3 == 0 { print $1 }'
Beth 4.00 0

Beth

Dan 3.75 0

Dan

Kathy 3.75 10
Kathy 3.75 0

Kathy

...

加粗的字符是计算机打印的。

这个动作非常便于尝试awk：输入你的程序，然后输入数据，观察发生了什么。我们再次鼓励你尝试这些示例并进行改动。

注意命令行中的程序是用单引号包围着的。这会防止shell解释程序中 *$* 这样的字符，也允许程序的长度超过一行。

当程序比较短小（几行的长度）的时候，这种约定会很方便。然而，如果程序较长，将程序写到一个单独的文件中会更加方便。假设存在程序 *progfile* ，输入命令行：：

```
awk -f progfile     optional list of input files
```

其中 *-f* 选项指示awk从指定文件中获取程序。可以使用任意文件名替换 *progfile* 。

错误

如果你的awk程序存在错误，awk会给你一段诊断信息。例如，如果你打错了大括号，如下所示：：

```
awk '$3 == 0 [ print $1 }' emp.data
```

你会得到如下信息：

```
awk: syntax error at source line 1
    context is
        $3 == 0 >>> [ <<<
        extra }
        missing ]
awk: bailing out at source line 1
```

“Syntax error”意味着在 >>> <<< 标记的地方检测到语法错误。“Bailing out”意味着没有试图恢复。有时你会得到更多的帮助-关于错误是什么，比如大括号或括弧不匹配。

因为存在句法错误，awk就不会尝试执行这个程序。然而，有些错误，直到你的程序被执行才会检测出来。例如，如果你试图用零去除某个数，awk会在这个除法的地方停止处理并报告输入行的行号以及在程序中的行号（这话是什么意思？难道输入行的行号是忽略空行后的行号？）。

1.2 简单输出

这一节接下来的部分包含了一些短小，典型的awk程序，基于操纵上文中提到的 *emp.data* 文件. 我们会简单的解释程序在做什么，但这些例子主要是为了介绍 awk 中常见的一些简单有用的操作 – 打印字段, 选择输入, 转换数据. 我们并 没有展现 awk 程序能做的所有事情, 也并不打算深入的去探讨例子中的一些细节. 但在你读完这一节之后, 你将能够完成一些简单的任务, 并且你将发现在阅读后 面章节的时候会变的容易的多.

我们通常只会列出程序部分, 而不是整个命令行. 在任何情况下, 程序都可以用 引号包含起来放到 awk 命令的地一个参数中运行, 就像上文中展示的那样, 或者 把它放到一个文件中使用 awk 的 *-f* 参数调用它.

在 awk 中仅仅只有两种数据类型: 数值 和 字符构成的字符串. *emp.data* 是 一个包含这类信息的典型文件 – 混合了被空格和(或)制表符分割的数字和词语.

Awk 程序一次从输入文件的中读取一行内容并把它分割成一个个字段, 通常默认 情况下, 一个字段是一个不包含任何空格或制表符的连续字符序列. 当前输入的 行中的地一个字段被称做 *$1*, 第二个是 *$2*, 以此类推. 整个行的内容被定 义为 *$0*. 每一行的字段数量可以不同.

通常, 我们要做的仅仅只是打印出每一行中的某些字段, 也许还要做一些计算. 这一节的程序基本上都是这种形式.

打印每一行

如果一个动作没有任何模式, 这个动作会对所有输入的行进行操作. *print* 语 句用来打印(输出)当前输入的行, 所以程序

```
{ print }
```

会输出所有输入的内容到标准输出. 由于 $0 表示整行,

```
{ print $0 }
```

也会做一样的事情.

打印特定字段

使用一个 *print* 语句可以在同一行中输出不止一个字段. 下面的程序输出了每 行输入中的第一和第三个字段

```
{ print $1, $3 }
```

使用 *emp.data* 作为输入, 它将会得到

```
Beth 0
Dan 0
Kathy 10
Mark 20
Mary 22
Susie 18
```

在 *print* 语句中被逗号分割的表达式, 在默认情况下他们将会用一个空格分割 来输出. 每一行 *print* 生成的内容都会以一个换行符作为结束. 但这些默认行为都可以自定义; 我们将在第二章中介绍具体的方法.

NF, 字段数量

很显然你可能会发现你总是需要通过 $1, $2 这样来指定不同的字段, 但任何表 达式都可以使用在$之后来表达一个字段的序号; 表达式会被求值并用于表示字段 序号. Awk会对当前输入的行有多少个字段进行计数, 并且将当前行的字段数量存 储在一个内建的称作 *NF* 的变量中. 因此, 下面的程序

```
{ print NF, $1, $NF }
```

会依次打印出每一行的字段数量, 第一个字段的值, 最后一个字段的值.

计算和打印

你也可以对字段的值进行计算后再打印出来. 下面的程序

```
{ print $1, $2 * $3 }
```

是一个典型的例子. 它会打印出姓名和员工的合计支出(以小时计算):

```
Beth 0
Dan 0
Kathy 40
Mark 100
Mary 121
Susie 76.5
```

我们马上就会学到怎么让这个输出看起来更漂亮.

打印行号

Awk提供了另一个内建变量, 叫做 *NR*, 它会存储当前已经读取了多少行的计数. 我们可以使用 *NR* 和 *$0* 给 *emp.data* 的没一行加上行号:

```
{ print NR, $0 }
```

打印的输出看起来会是这样:

```
1 Beth   4.00    0
2 Dan    3.75    0
3 Kathy  4.00   10
4 Mark   5.00   20
5 Mary   5.50   22
6 Susie  4.25   1 8
```

在输出中添加内容

你当然也可以在字段中间或者计算的值中间打印输出想要的内容:

```
{ print "total pay for", $1, "is", $2 * $3 }
```

输出

```
total pay for Beth is 0
total pay for Dan is 0
total pay for Kathy is 40
total pay for Mark is 100
total pay for Mary is 121
total pay for Susie is 76.5
```

在打印语句中,双引号内的文字将会在字段和计算的值中插入输出.

1.3 高级输出

print 语句可用于快速而简单的输出。若要严格按照你所想的格式化输出，则需要使用 *printf* 语句。正如我将在2.4节所见， *printf* 几乎可以产生任何形式的输出，但在本节中，我们仅展示其部分功能。

字段排队

printf 语句的形式如下：:

```
printf(format, value1, value2, ..., valuen)
```

其中 *format* 是字符串，包含要逐字打印的文本，穿插着 *format* 之后的每个值该如何打印的规格(specification)。一个规格是一个 *%* 符，后面跟着一些字符，用来控制一个 *value* 的格式。第一个规格说明如何打印 *value1* ，第二个说明如何打印 *value2* ，... 。因此，有多少 *value* 要打印，在 *format* 中就要有多少个 *%* 规格。

这里有个程序使用 *printf* 打印每位员工的总薪酬：:

```
{ printf("total pay for %s is $%.2f\n", $1, $2 * $3) }
```

printf 语句中的规格字符串包含两个 *%* 规格。第一个是 *%s* ，说明以字符串的方式打印第一个值 *$1* 。第二个是 *%.2f* ，说明以数字的方式打印第二个值 *$2*$3* ，并保留小数点后面两位。规格字符串中其他东西，包括美元符号，仅逐字打印。字符串尾部的 *\n* 代表开始新的一行，使得后续输出将从下一行开始。以 *emp.data* 为输入，该程序产生：

```
total pay for Beth is $0.00
total pay for Dan is $0.00
total pay for Kathy is $40.00
total pay for Mark is $100.00
total pay for Mary is $121.00
total pay for Susie is $76.50
```

printf 不会自动产生空格或者新的行，必须是你自己来创建，所以不要忘了 *\n* 。

另一个程序是打印每位员工的姓名与薪酬：:

```
{ printf("%-8s $%6.2f\n", $1, $2 * $3) }
```

第一个规格 *%-8s* 将一个姓名以字符串形式在8个字符宽度的字段中左对齐输出。第二个规格 *%6.2f* 将薪酬以数字的形式，保留小数点后两位，在6个字符宽度的字段中输出。

```
Beth     $  0.00
Dan      $  0.00
Kathy    $ 40.00
Mark     $100.00
Mary     $121.00
Susie    $ 76.50
```

之后我们将展示更多的 *printf* 示例。一切精彩尽在2.4小节。

排序输出

假设你想打印每位员工的所有数据，包括他或她的薪酬，并以薪酬递增的方式进行排序输出。最简单的方式是使用awk将每位员工的总薪酬置于其记录之前，然后利用一个排序程序来处理awk的输出。Unix上，命令行如下:

```
awk '{ printf("%6.2f    %s\n", $2 * $3, $0) }' emp.data | sort
```

将awk的输出通过管道传给 *sort* 命令，输出为：

```
  0.00    Beth  4.00 0
  0.00    Dan   3.75 0
 40.00    Kathy 4.00 10
 76.50    Susie 4.25 18
100.00    Mark  5.00 20
121.00    Mary  5.50 22
```

1.4 选择

Awk的模式适合用于为进一步的处理从输入中选择相关的数据行。由于不带动作的模式会打印所有匹配模式的行，所以很多awk程序仅包含一个模式。本节将给出一些有用的模式示例。

通过对比选择

这个程序使用一个对比模式来选择每小时赚5美元或更多的员工记录，也就是，第二个字段大于等于5的行：：

```
$2 >= 5
```

从 *emp.data* 中选出这些行：：

```
Mark    5.00    20
Mary    5.50    22
```

通过计算选择

程序

```
$2 * $3 > 50 { printf("$%.2f for %s\n", $2 * $3, $1) }
```

打印出总薪资超过50美元的员工的薪酬。

通过文本内容选择

除了数值测试，你还可以选择包含特定单词或短语的输入行。这个程序会打印所有第一个字段为 *Susie* 的行：：

```
$1 == "Susie"
```

操作符 == 用于测试相等性。你也可以使用称为 *正则表达式* 的模式查找包含任意字母组合，单词或短语的文本。这个程序打印任意位置包含 *Susie* 的行：：

```
/Susie/
```

输出为这一行：：

```
Susie    4.25    18
```

正则表达式可用于指定复杂的多的模式；2.1节将会有全面的论述。

模式组合

可以使用括号和逻辑操作符与 *&&* ， 或 **||** ， 以及非 *!* 对模式进行组合。程序:

```
$2 >= 4 || $3 >= 20
```

会打印 *$2* (第二个字段) 大于等于 *4* 或者 *$3* (第三个字段) 大于等于 *20* 的行：：

```
Beth    4.00    0
```

```
kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

两个条件都满足的行仅打印一次。与如下包含两个模式程序相比：：

```
$2 >= 4
$3 >= 20
```

如果某个输入行两个条件都满足，这个程序会打印它两遍：：

```
Beth    4.00    0
Kathy   4.00    10
Mark    5.00    20
Mark    5.00    20
Mary    5.50    22
Mary    5.50    22
Susie   4.25    18
```

注意如下程序:

```
!($2 < 4 && $3 < 20)
```

会打印极不满足 *$2* 小于4也不满足 *$3* 小于20的行；这个条件与上面第一个模式组合等价，虽然也许可读性差了点。

数据验证

实际的数据中总是会存在错误的。在数据验证-检查数据的值是否合理以及格式是否正确-方面，Awk是个优秀的工具。

数据验证本质上是否定的：不是打印具备期望属性的行，而是打印可疑的行。如下程序使用对比模式 将5个数据合理性测试应用于 *emp.data* 的每一行：：

```
NF != 3     { print $0, "number of fields is not equal to 3" }
$2 < 3.35   { print $0, "rate is below minimum wage" }
$2 > 10     { print $0, "rate exceeds $10 per hour" }
$3 < 0      { print $0, "negative hours worked" }
$3 > 60     { print $0, "too many hours worked" }
```

如果没有错误，则没有输出。

BEGIN与END

特殊模式 *BEGIN* 用于匹配第一个输入文件的第一行之前的位置， *END* 则用于匹配处理过的最后一个文件的最后一行之后的位置。这个程序使用 *BEGIN* 来输出一个标题：：

```
BEGIN { print "Name    RATE    HOURS"; print ""}
      { print }
```

输出为：：

```
NAME    RATE    HOURS

Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

程序的动作部分你可以在一行上放多个语句，不过要使用分号进行分隔。注意 普通的 *print* 是打印当前输入行，与之不同的是 *print “”* 会打印一个空行。

1.5 使用AWK进行计算

一个动作就是一个以新行或者分号分隔的语句序列。你已经见过一些其动作仅是单个 *print* 语句的例子。本节将提供一些执行简单的数值以及字符串计算的语句示例。在这些语句中，你不仅可以使用像 *NF* 这样的内置变量，还可以创建自己的变量用于计算、存储数据诸如此类的操作。awk中，用户创建的变量不需要声明。

计数

这个程序使用一个变量 *emp* 来统计工作超过15个小时的员工的数目：：

```
$3 > 15 { emp = emp + 1 }
END     { print emp, "employees worked more than 15 hours" }
```

对于第三个字段超过15的每行， *emp* 的前一个值加1。以 *emp.data* 为输入，该程序产生：：

```
3 employees worked more than 15 hours
```

用作数字的awk变量的默认初始值为0，所以我们不需要初始化 *emp* 。

求和与平均值

为计算员工的数目，我们可以使用内置变量 *NR* ，它保存着到目前位置读取的行数；在所有输入的结尾它的值就是所读的所有行数。

```
END { print NR, "employees" }
```

输出为：：

```
6 employees
```

如下是一个使用 *NR* 来计算薪酬均值的程序：：

```
    { pay = pay + $2 * $3 }
END { print NR, "employees"
      print "total pay is", pay
      print "average pay is", pay/NR
    }
```

第一个动作累计所有员工的总薪酬。 *END* 动作打印出

```
6 employees
total pay is 337.5
average pay is 56.25
```

很明显， *printf* 可用来产生更简洁的输出。并且该程序也有个潜在的错误：在某种不太可能发生的情况下， *NR* 等于0，那么程序会试图执行零除，从而产生错误信息。

处理文本

awk的优势之一是能像大多数语言处理数字一样方便地处理字符串。awk变量可以保存数字也可以保存字符串。这个程序会找出时薪最高的员工：：

```
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print "highest hourly rate:", maxrate, "for", maxemp }
```

输出

```
highest hourly rate: 5.50 for Mary
```

这个程序中，变量 *maxrate* 保存着一个数值，而变量 *maxemp* 则是保存着一个字符串。（如果有几个员工都有着相同的最大时薪，该程序则只找出第一个。）

字符串连接

可以合并老字符串来创建新字符串。这种操作称为 *连接（concatenation）* 。程序

```
    { names = names $1 " "}
END { print names }
```

通过将每个姓名和一个空格附加到变量 *names* 的前一个值， 来将所有员工的姓名收集进单个字符串中。最后 *END* 动作打印出 *names* 的值：:

```
Beth Dan Kathy Mark Mary Susie
```

awk程序中，连接操作的表现形式是将字符串值一个接一个地写出来。对于每个输入行，程序的第一个语句先连接三个字符串： *names* 的前一个值、当前行的第一个字段以及一个空格，然后将得到的字符串赋值给 *names* 。因此，读取所有的输入行之后， *names* 就是个字符串，包含所有员工的姓名，每个姓名后面跟着一个空格。用于保存字符串的变量的默认初始值是空字符串(也就是说该字符串包含零个字符)，因此这个程序中的 *names* 不需要显式初始化。

打印最后一个输入行

虽然在 *END* 动作中 *NR* 还保留着它的值，但 *$0* 没有。程序

```
    { last = $0 }
END { print last }
```

是打印最后一个输入行的一种方式：:

```
Susie    4.25    18
```

内置函数

我们已看到awk提供了内置变量来保存某些频繁使用的数量，比如：字段的数量和输入行的数量。类似地，也有内置函数用来计算其他有用的数值。除了平方根、对数、随机数诸如此类的算术函数，也有操作文本的函数。其中之一是 *length* ，计算一个字符串中的字符数量。例如，这个程序会计算每个人的姓名的长度：:

```
{ print $1, length($1) }
```

结果：:

```
Beth 4
Dan 3
Kathy 5
Mark 4
Mary 4
Susie 5
```

行、单词以及字符的计数

这个程序使用了 *length* 、 *NF* 、以及 *NR* 来统计输入中行、单词以及字符的数量。为了简便，我们将每个字段看作一个单词。

```
    { nc = nc + length($0) + 1
      nw = nw + NF
    }
END { print NR, "lines,", nw, "words,", nc, "characters" }
```

文件 *emp.data* 有:

```
6 lines, 18 words, 77 characters
```

$0 并不包含每个输入行的末尾的换行符，所以我们要另外加个1。

1.6 控制语句

Awk为选择提供了一个 *if-else* 语句，以及为循环提供了几个语句，所以都效仿C语言中对应的控制语句。它们仅可以在动作中使用。

if-else语句

如下程序将计算时薪超过6美元的员工的总薪酬与平均薪酬。它使用一个 *if* 来防范计算平均薪酬时的零除问题。

```
$2 > 6 { n = n + 1; pay = pay + $2 * $3 }
END    { if (n > 0)
           print n, "employees, total pay is", pay,
                    "average pay is", pay/n
         else
           print "no employees are paid more than $6/hour"
       }
```

emp.data 的输出是：：

```
no employees are paid more than $6/hour
```

if-else 语句中，*if* 后的条件会被计算。如果为真，执行第一个 *print* 语句。否则，执行第二个 *print* 语句。注意我们可以使用一个逗号将一个长语句截断为多行来书写。

while语句

一个 *while* 语句有一个条件和一个执行体。条件为真时执行体中的语句会被重复执行。这个程序使用公式 $value = amount(1 + rate)^{years}$

来演示以特定的利率投资一定量的钱，其数值是如何随着年数增长的。

```
# interest1 - 计算复利
#   输入: 钱数    利率    年数
#   输出: 复利值

{   i = 1
    while (i <= $3) {
        printf("\t%.2f\n", $1 * (1 + $2) ^ i)
        i = i + 1
    }
}
```

条件是 *while* 后括弧包围的表达式；循环体是条件后大括号包围的两个表达式。 *printf* 规格字符串中的 \t 代表制表符； ^ 是指数操作符。从 # 开始到行尾的文本是注释，会被awk忽略，但能帮助程序的读者理解程序做的事情。

你可以为这程序输入三个一组的数字，看看不一样的钱数、利率、以及年数会产生什么。例如，如下事务演示了1000美元，利率为6%与12%，5年的复利分别是如何增长的：：

```
$ awk -f interest1
1000 .06 5
        1060.00
        1123.60
        1191.02
        1262.48
        1338.23
1000 .12 5
        1120.00
        1254.40
        1404.93
        1573.52
        1762.34
```

for语句

另一个语句， *for* ，将大多数循环都包含的初始化、测试、以及自增压缩成一行。如下是之前利息计算的 *for* 版本：：

```
# interest1 - 计算复利
#   输入: 钱数    利率    年数
#   输出: 每年末的复利

{ for (i = 1; i <= $3; i = i + 1)
    printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

初始化 *i = 1* 只执行一次。接下来，测试条件 *i <= $3* ；如果为真，则执行循环体的 *printf* 语句。循环体执行结束后执行自增 *i = i + 1* ，接着由另一次条件测试开始下一个循环迭代。代码更加紧凑，并且由于循环体仅是一条语句，所以不需要大括号来包围它。

1.7 数组

awk为存储一组相关的值提供了数组。虽然数组给予了awk很强的能力，但在这里我们仅展示一个简单的例子。如下程序将按行逆序打印输入。第一个动作将输入行存为数组 *line* 的连续元素；即第一行放在 *line[1]* ，第二行放在 *line[2]* , 依次继续。 *END* 动作使用一个 *while* 语句从后往前打印数组中的输入行：：

```
# 反转 - 按行逆序打印输入

    { line[NR] = $0 }  # 记下每个输入行

END { i = NR           # 逆序打印
      while (i > 0) {
        print line[i]
        i = i - 1
      }
    }
```

以 *emp.data* 为输入，输出为

```
Susie     4.25   18
Mary      5.50   22
Mark      5.00   20
Kathy     4.00   10
Dan       3.75   0
Beth      4.00   0
```

如下是使用 *for* 语句实现的相同示例：：

```
# 反转 - 按行逆序打印输入

    { line[NR] = $0 }   # 记下每个输入行

END { for (i = NR; i > 0; i = i - 1)
        print line[i]
    }
```

二. AWK语言详解

本章将主要通过示例来解释构成awk程序的概念。因为这是对语言的全面描述，材料会很详细，因此我们推荐你浏览略读，需要的时候再回来核对细节。

最简单的awk程序就是一个模式-动作语句的序列：：

```
pattern    { action }
pattern    { action }
...
```

某些语句中，可能没有模式；另一些语句中，可能没有动作及其大括号。awk检查你的程序以确认不存在语法错误后，一次读取一行输入，并对每一行按序处理模式。对于每个匹配到当前输入行的模式，执行其关联的动作。不存在模式，则匹配每个输入行，因此没有模式的每个动作对于每个输入行都要执行。一个仅包含模式的模式-动作语句将打印匹配该模式的每个输入行。本章的大部分内容中，名词"输入行(input-line)"和"记录(record)" 是同义的。2.5小节中，我们将讨论多行记录，即一个记录包含多行输入。

本章的第一节将详细描述模式。第二节通过表达式、赋值以及控制语句来描述动作。剩下的章节覆盖函数定义，输出，输入，以及awk程序如何调用其他程序等内容。多数章节包含了主要特性的概要。

输入文件 countries

本章中，我们使用一个名为 *countries* 的文件作为许多awk程序的输入。文件的每行包含一个国家的名字，以千平方英里为单位的面积，以百万为单位的人口数，以及属于哪个洲。数据是1984年的，苏联(USSR)被武断地归入了亚洲。文件中，四列数据以制表符tab分隔；以单个空格将 *North* 、 *South* 与 *America* 分隔开。

文件 *countries* 包含如下数据行：：

```
USSR    8649    275     Asia
Canada  3852    25      North America
China   3705    1032    Asia
USA     3615    237     North America
Brazil  3286    134     South America
India   1267    746     Asia
Mexico  762     78      North America
France  211     55      Europe
Japan   144     120     Asia
Germany 96      61      Europe
England 94      56      Europe
```

本章的其余部分，如果没有明确说明输入文件，那么就是使用文件 *countries* 。

程序的格式

模式-动作语句以及动作中的语句通常以换行分隔，如果它们以分号分隔，则多个语句可以出现在一行中。分号可以放在任意语句的尾部。

动作的开大括号必须与其对应的模式处于同一行；动作的其余部分，包括闭大括号，则可以出现接下来的行中。

空行会被忽略；一般为了提高程序的可读性会在语句的前面或者后面插入空行。在操作符和操作数的两边插入空格和制表符也是为了提高可读性。

任意行的末尾可能会有注释。注释以符号 # 开始，结束于行尾，就像这样

```
{ print $1, $3 }        # print country name and population
```

长语句可以跨越多行，但要在断行的地方加入一个反斜杠和一个换行符：：

```
{ print \
                $1,             # country name
                $2,             # area in thousands of square miles
                $3 }    # population in millions
```

如上例所示，语句也可以逗号断行，在每个断行的末尾也可以加入注释。

本书中，我们使用了多种格式风格，部分是为了说明相异之处，部分是为了避免程序占用太多的行空间。类似于本章中的简短程序，格式并不是很重要，但一致性与可读性可以帮助更长的程序保持可控。

2.1 模式

模式控制着动作的执行：模式匹配，其关联的动作则执行。本节将描述6种模式及其匹配条件。

模式摘要

1. BEGIN { 语句 }
   在读取任何输入前执行一次 *语句*
2. END { 语句 }
   读取所有输入之后执行一次 *语句*
3. 表达式 { 语句 }
   对于 *表达式* 为真（即，非零或非空）的行，执行 *语句*
4. /正则表达式/ { 语句 }
   如果输入行包含字符串与 *正则表达式* 相匹配，则执行 *语句*
5. 组合模式 { 语句 }
   一个 *组合模式* 通过与（&&），或（||），非（|），以及括弧来组合多个表达式；对于组合模式为真的每个输入行，执行 *语句*
6. 模式1，模式2 { 语句 }
   范围模式(range pattern)匹配从与 *模式1* 相匹配的行到与 *模式2* 相匹配的行（包含该行）之间的所有行，对于这些输入行，执行 *语句*。

BEGIN和END不与其他模式组合。范围模式不可以是任何其他模式的一部分。BEGIN和END是仅有的必须搭配动作的模式。
