Sed & Awk 101 Hacks

Enhance Your UNIX / Linux Life with Sed and Awk

目录

简介

用 sed 和 awk 提高你的 UNIX 和 Linux 水平

如果你是一个开发者、系统管理员、数据库管理员或 IT 管理员，要花不少时间使用 UNIX/Linux,那么你应该精通 Sed 和 Awk

Sed 和 Awk 是两个很强大的工具，能以很少的几行代码快速解决很多复杂的问题——很多情况下，仅需要一行代码即可搞定。

本书包含以下内容：

- 第 1 至 7 章介绍 sed，第 8 至 13 章介绍 awk
- 第 1 至 5 章解释 sed 的各种命令，包括强大的替换命令、正则表达式以及执行这些命令的不同方式
- 第 6 至 7 章介绍保持空间和模式空间，sed 多行命令，以及循环，其他提供了一些简单的例子
- 第 8 至 11 章使用例子和 awk 内置变量，介绍 awk 的各种命令
- 第 12 至 13 章用简单明了的例子，解释 awk 中强大的关联数组，以及内置的函数和命令

提示：大部分示例会用下面的方式排版样式：

示例描述

这里是你要手动输入的代码，以及其输出的结果。

其它任何附带的解释和描述，都会以文本的方式出现在代码的后面。

此外还要注意，如果你不想手动输入，而是复制粘贴，那么请确保把命令粘贴到同一行中。

第一章：Sed 语法和基本命令

所有的示例都要用到下面的 employee.txt 文件，请先行创建该文件。

```
$ vi employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

上面的雇员数据，每行记录都包含下面几列:

- 雇员 ID

- 雇员姓名
- 雇员职位

Sed 代表 Strem Editor(流编辑器),是操作、过滤和转换文本内容的强大工具。Sed 可以从文件和管道中读取输入。在你的 bash 启动文件中，就可能有不少用来设置各种环境的 sed 命令，这些命令你或许并不是很理解。

对于新手来说，sed 脚本看起来可能比较晦涩难懂，一旦你详细理解了 sed 命令，你就可以写出能解决很多复杂问题的 sed 脚本。

本书将用简单易懂的例子解释所有的 sed 命令。

1.Sed 命令语法

本部分内容旨在带你熟悉 sed 命令的语法和结构，但并不解释命令的含义，后面会详细解释这些命令。
Sed 基本语法:
sed [options] {sed-commands} {input-file}

sed 每次从 input-file 中读取一行记录，并在该记录上执行 sed-commands

sed 首先从 input-file 中读取第一行，然后执行所有的 sed-commands；再读取第二行，执行所有 sed-commands,重复这个过程，直到 input-file 结束

通过制定[options] 还可以给 sed 传递一些可选的选项

下面的例子演示了 sed 的基本语法，它打印出/etc/passwd 文件中的所有行

```
sed –n ‘p’ /etc/passwd
```

该例子的重点在于，{sed-commands}既可以是单个命令，也可以是多个命令。你也可以把多个 sed 命令合并到一个文件中，这个文件被称为 sed 脚本，然后使用-f 选项调用它，如下面的例子：
使用 sed 脚本的基本语法:

```
sed [ options ] –f {sed-commands-in-a-file} {input-file}
```

下面的例子演示了使用 sed 脚本的用法，这个例子将打印/etc/passwd 问中以 root 和 nobody 开头的行：

```
$ vi test-script.sed
/^root/ p
/^nobody/ p

$ sed –n –f test-script.sed /etc/passwd
```

你也可以使用 –e 选项，执行多个 sed 命令，如下所示:

-e 的使用方法：

sed [options] –e {sed-command-1} –e {sed-command-2} {input-file}

下面的例子演示了-e 的使用方法，它打印/etc/passwd 中以 root 和 nobody 开头的行:

sed –n –e '/^root/ p' –e '/^nobody/ p' /etc/passwd

如果使用-e 执行多个命令，也可以使用反斜杠\把它们分割到多行执行：

```
Sed –n \
-e '/^root/ p' \
-e '/^nobody/ p' \
/etc/passwd
```

也可以使用{ }将多个命令分组执行：

{}的使用方法：

```
sed [options] '{
sed-command-1
sed-command-2
}' input-file
```

下面的例子演示了{}的使用方法，打印/etc/passwd 中以 root 和 nobody 开头的行:

```
sed –n '{
/^root/ p
/^nobody/ p
}' /etc/passwd
```

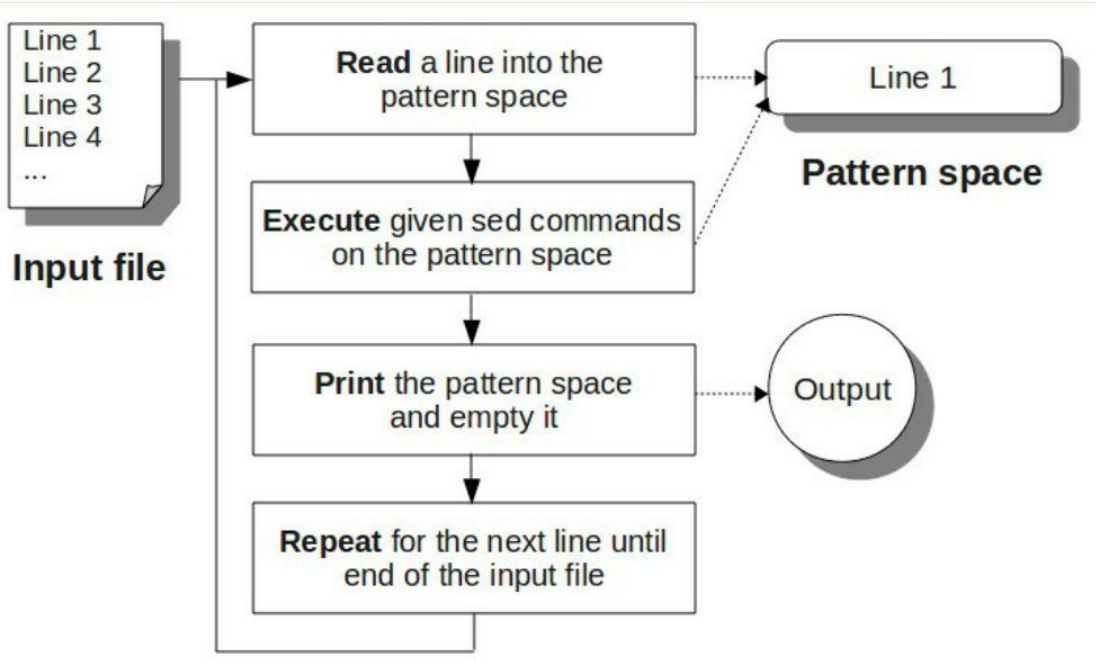
注意:sed 绝不会修改原始文件 input-file,它只是将结果内容输出到标准输出设备。如果要保持变更，应该使用重定向 > filename.txt

2.Sed 脚本执行流程

Sed 脚本执行遵从下面简单易记的顺序：Read,Execute,Print,Repeat(读取，执行，打印，重复)，简称 REPR

分析脚本执行顺序：

- 读取一行到模式空间(sed 内部的一个临时缓存，用于存放读取到的内容)
- 在模式空间中执行命令。如果使用了{ } 或 –e 指定了多个命令，sed 将依次执行每个命令
- 打印模式空间的内容，然后清空模式空间
- 重复上述过程，直到文件结束



图片：sed 执行流程图

3.打印模式空间(命令 p)

使用命令 p，可以打印当前模式空间的内容。
sed 在执行完命令后会默认打印模式空间的内容，既然如此，那么你可能会问为何还需要命令 p 呢。
有如下原因，命令 p 可以控制只输出你指定的内容。通常使用 p 时，还需要使用-n 选项来屏蔽 sed 的默认输出，否则当执行命令 p 时，每行记录会输出两次。
下面的例子打印 employee.txt 文件，每行会输出两次：

```
$ sed ‘p’ employee.txt
101,John Doe,CEO
101,John Doe,CEO
102,Jason Smith,IT Manager
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
105,Jane Miller,Sales Manager
```

输出 employee.txt 的内容，只打印一行(和 cat employee.txt 命令作用相同):

```
$ sed –n ‘p’ employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

105,Jane Miller,Sales Manager

指定地址范围

如果在命令前面不指定地址范围，那么默认会匹配所有行。下面的例子，在命令前面指定了地址范围：

只打印第 2 行:

```
$ sed -n '2 p' employee.txt
102,Jason Smith,IT Manager
```

打印第 1 至第 4 行:

```
$ sed -n '1,4 p' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

打印第 2 行至最后一行($代表最后一行):

```
$ sed -n '2,$ p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

修改地址范围

可以使用逗号、加号、和波浪号来修改地址范围。

上面的例子里面，就已经使用了逗号参与地址范围的指定。其意思很明了: n,m 代表第 n 至第 m 行。

加号+配合逗号使用，可以指定相的若干行，而不是绝对的几行。如 n,+m 表示从第 n 行开始后的 m 行

波浪号~也可以指定地址范围。它指定每次要跳过的行数。如 n~m 表示从第 n 行开始，每次跳过 m 行：

- 1~2 匹配 1,3,5,7,……
- 2~2 匹配 2,4,6,8,……
- 1~3 匹配 1,4,7,10,…..
- 2~3 匹配 2,5,8,11,…..

只打印奇数行:

```
$ sed -n '1~2 p' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
105,Jane Miller,Sales Manager
```

模式匹配

一如可以使用数字指定地址(或地址范围),也可以使用一个模式(或模式范围)来匹配，如下面

的例子所示。

打印匹配模式"Jane"的行:

```
$ sed -n '/Jane/ p' employee.txt
105,Jane Miller,Sales Manager
```

打印第一次匹配 Jason 的行至第 4 行的内容:

```
$ sed -n '/Jason/,4 p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

注意：如果开始的 4 行中，没有匹配到 Jason,那么 sed 会打印第 4 行以后匹配到 Jason 的内容

打印从第一次匹配 Raj 的行到最后的所有行：

```
$ sed -n '/Raj/,$ p' employee.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

打印自匹配 Raj 的行开始到匹配 Jane 的行之间的所有内容：

```
$ sed -n '/Raj/,/Jane/ p' employee.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

打印匹配 Jason 的行和其后面的两行:

```
$ sed -n '/Jane/,+2 p' employee.txt
105,Jane Miller,Sales Manager
```

4.删除行

命令 d 用来删除行，需要注意的是它只删除模式空间的内容，和其他 sed 命令一样，命令 d 不会修改原始文件的内容。

如果不提供地址范围，sed 默认匹配所有行，所以下面的例子什么都不会输出，因为它匹配了所有行并删除了它们：

```
sed 'd' employee.txt
```

指定要删除的地址范围更有用，下面是几个例子：

只删除第 2 行:

```
$ sed '2 d' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

删除第 1 至第 4 行：

```
$ sed '1,4 d' employee.txt
105,Jane Miller,Sales Manager
```

删除第 2 行至最后一行:

```
$ sed '2,$ d' employee.txt
101,John Doe,CEO
```

只删除奇数行：

```
$ sed '1~2 d' employee.txt
102,Jason Smith,IT Manager
104,Anand Ram,Developer
```

删除匹配 Manager 的行:

```
$ sed '/Manager/ d' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

删除从第一次匹配 Jason 的行至第 4 行：

```
$ sed '/Jason/,4 d' employee.txt
101,John Doe,CEO
105,Jane Miller,Sales Manager
```

如果开头的 4 行中，没有匹配 Jason 的行，那么上述命令将删除第 4 行以后匹配 Manager 的行

删除从第一次匹配 Raj 的行至最后一行:

```
$ sed '/Raj/,$ d' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
```

删除第一次匹配 Jason 的行和紧跟着它后面的两行:

```
$ sed '/Jason/,+2 d' employee.txt
101,John Doe,CEO
105,Jane Miller,Sales Manager
```

常用的删除命令示例:

删除所有空行

```
sed ‘/^$/ d’ employee.txt
```

删除所有注释行(假定注释行以#开头)

```
sed ‘/^#/ ‘ employee.txt
```

注意：如果有多个命令，sed 遇到命令 d 时，会删除匹配到的整行数据，其余的命令将无法操作被删除的行。

5.把模式空间内容写到文件中(w 命令)

命令 w 可以把当前模式空间的内容保存到文件中。默认情况下模式空间的内容每次都会打印到标准输出，如果要把输出保存到文件同时不显示到屏幕上，还需要使用-n 选项。

下面是几个例子：

把 employee.txt 的内容保存到文件 output.txt，同时显示在屏幕上

```
$ sed 'w output.txt' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager

$ cat output.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

把 **employee.txt** 的内容保存到文件 **output.txt**，但不在屏幕上显示

```
$ sed -n 'w output.txt' employee.txt

$ cat output.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

只保存第 **2** 行：

```
$ sed -n '2 w output.txt' employee.txt

$ cat output.txt
102,Jason Smith,IT Manager
```

保存第 **1** 至第 **4** 行：

```
$ sed -n '1,4 w output.txt' employee.txt

$ cat output.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

保存第 **2** 行起至最后一行：

```
$ sed -n '2,$ w output.txt' employee.txt

$ cat output.txt
102,Jason Smith,IT Manager
```

103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager

只保存奇数行：

```
$ sed -n '1~2 w output.txt' employee.txt

$ cat output.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
105,Jane Miller,Sales Manager
```

保存匹配 Jane 的行:

```
$ sed -n '/Jane/ w output.txt' employee.txt

$ cat output.txt
105,Jane Miller,Sales Manager
```

保存第一次匹配 Jason 的行至第 4 行：

```
$ sed -n '/Jason/,4 w output.txt' employee.txt
$
$ cat output.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

注意：如果开始的 4 行里没有匹配到 Jason,那么该命令只保存第 4 行以后匹配到 Jason 行

保存第一次匹配 Raj 的行至最后一行:

```
$ sed -n '/Raj/,$ w output.txt' employee.txt

$ cat output.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

保存匹配 Raj 的行至匹配 Jane 的行:

```
$ sed -n '/Raj/,/Jane/ w output.txt' employee.txt

$ cat output.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

保存匹配 Jason 的行以及紧跟在其后面的两行：

```
$ sed -n '/Jason/,+2 w output.txt' employee.txt

$ cat output.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

第二章：sed 替换命令

流编辑器中最强大的功能就是替换(substitute),其强大的功能和繁多的选项将占据下面整整一章的内容。

6.sed 替换命令语法

sed ‘[address-range|pattern-range] s/original-string/replacement-string/[substitute-flags]’ input file

上面提到的语法为：

- address-range 或 pattern-range(即地址范围和模式范围)是可选的。如果没有指定，那么 sed 将在所有行上进行替换。
- s 即执行替换命令 substitute
- original-string 是被 sed 搜索然后被替换的字符串，它可以是一个正则表达式
- replacement-string 替换后的字符串
- substitute-flags 是可选的，下面会具体解释

请谨记原始输入文件不会被修改，sed 只在模式空间中执行替换命令，然后输出模式空间的内容。

下面是一些简单的替换示例（替换的部分用黑体标明）

用 Director 替换所有行中的 Manager:

```
$ sed 's/Manager/Director/' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Director
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Director
```

只把包含 Sales 的行中的 Manager 替换为 Director:

```
$ sed '/Sales/s/Manager/Director/' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
```

103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales **Director**
注意：本例由于使用了地址范围限制，所以只有一个 Manager 被替换了

7.全局标志 g

g 代表全局(global) 默认情况下，sed 至会替换每行中第一次出现的 original-string。如果你要替换每行中出现的所有 original-string,就需要使用 g

用大写 A 替换第一次出现的小写字母 a：
$ sed 's/a/A/' employee.txt
101,John Doe,CEO
102,J**A**son Smith,IT Manager
103,R**A**j Reddy,Sysadmin
104,An**A**nd Ram,Developer
105,J**A**ne Miller,Sales Manager

把所有小写字母 a 替换为大写字母 A：
$ sed 's/a/A/g' employee.txt
101,John Doe,CEO
102,J**A**son Smith,IT M**A**nAger
103,R**A**j Reddy,Sys**A**dmin
104,An**A**nd R**A**m,Developer
105,J**A**ne Miller,S**A**les M**A**n**A**ger
注意：上述例子会在所有行上替换，因为没有指定地址范围。

8.数字标志(1,2,3 ....)

使用数字可以指定 original-string 出现的次序。只有第 n 次出现的 original-string 才会触发替换。每行的数字从 1 开始，最大为 512。

比如/11 会替换每行中第 11 次出现的 original-string
下面是几个例子：
把第二次出现的小写字母 a 替换为大写字母 A：
$ sed 's/a/A/2' employee.txt
101,John Doe,CEO
102,Jason Smith,IT M**A**nager
103,Raj Reddy,Sys**A**dmin

104,Anand R**A**m,Developer
105,Jane Miller,S**A**les Manager

为了方面下面示例，请先建立如下文件：
$ vim substitute-locate.txt
locate command is used to locate files
locate command uses database to locate files
locate command can also use regex for searching

使用刚才建立的文件，把每行中第二次出现的 locate 替换为 find:
$ sed 's/locate/find/2' substitute-locate.txt
locate command is used to **find** files
locate command uses database to **find** files
locate command can also use regex for searching
注意：第 3 行中 locate 只出现了一次，所以没有替换任何内容

9.打印标志 p(print)

命令 p 代表 print。当替换操作完成后，打印替换后的行。与其他打印命令类似，sed 中比较有用的方法是和-n 一起使用以抑制默认的打印操作。

只打印替换后的行：
$ sed -n 's/John/Johnny/p' employee.txt
101,**Johnny** Doe,CEO

在之前的数字标志的例子中，使用/2 来替换第二次出现的 locate。第 3 行中 locate 只出现了一次，所以没有替换任何内容。使用 p 标志可以只打印替换过的两行。

把每行中第二次出现的 locate 替换为 find 并打印出来：
$ sed -n 's/locate/find/2p' substitute-locate.txt
locate command is used to **find** files
locate command uses database to **find** files

10.写标志 w

标志 w 代表 write。当替换操作执行成功后，它把替换后的结果保存的文件中。多数人更倾向于使用 p 打印内容，然后重定向到文件中。为了对 sed 标志有个完整的描述，在这里把这个标志也提出来了。

只把替换后的内容写到 output.txt 中:

```
$ sed -n 's/John/Johnny/w output.txt' employee.txt
$
$ cat output.txt
101,Johnny Doe,CEO
```

和之前使用的命令 p 一样，使用 w 会把替换后的内容保存到文件 output.txt 中。

把每行第二次出现的 locate 替换为 find，把替换的结果保存到文件中，同时显示输入文件所有内容：

```
$ sed 's/locate/find/2w output.txt' substitute-locate.txt
locate command is used to find files
locate command uses database to find files
locate command can also use regex for searching

$ cat output.txt
locate command is used to find files
locate command uses database to find files
```

11.忽略大小写标志 i (ignore)

替换标志 i 代表忽略大小写。可以使用 i 来以小写字符的模式匹配 original-string。该标志只有 GNU Sed 中才可使用。

下面的例子不会把 John 替换为 Johnny,因为 original-string 字符串是小写形式:

```
$ sed 's/john/Johnny/' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

把 john 或 John 替换为 Johnny:

```
$ sed 's/john/Johnny/i' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

12.执行命令标志 e (excuate)

替换标志 e 代表执行(execute)。该标志可以将模式空间中的任何内容当做 shell 命令执行，并把命令执行的结果返回到模式空间。该标志只有 GNU Sed 中才可使用。

下面是几个例子：
为了下面的例子，请先建立如下文件:

```
$ cat files.txt
/etc/passwd
/etc/group
```

在 files.txt 文件中的每行前面添加 ls –l 并打击结果：

```
$ sed 's/^/ls -l/' files.txt
ls -l/etc/passwd
ls -l/etc/group
```

在 files.txt 文件中的每行前面添加 ls –l 并把结果作为命令执行:

```
$ sed 's/^/ls -l /e' files.txt
-rw-r--r-- 1 root root 1533 Dec 13 20:21 /etc/passwd
-rw-r--r-- 1 root root 682 Dec 13 20:21 /etc/group
```

13.使用替换标志组合

根据需要可以把一个或多个替换标志组合起来使用。

下面的例子将把每行中出现的所有 Manager 或 manager 替换为 Director。然后把替换后的内容打印到屏幕上，同时把这些内容保存到 output.txt 文件中。

使用 g,I,p 和 w 的组合:

```
$ sed -n 's/manager/Director/igpw output.txt' employee.txt
102,Jason Smith,IT Director
105,Jane Miller,Sales Director

$ cat output.txt
102,Jason Smith,IT Director
105,Jane Miller,Sales Director
```

14.sed 替换命令分界符

上面所有例子中，我们都是用了 sed 默认的分界符/,即 s/original-string/replacement-string/g

如果在 original-string 或 replacement-string 中有/,那么需要使用反斜杠\来转义。为了便于示例，请先建立下面文件:

```
$ vi path.txt
$ cat path.txt
reading /usr/local/bin directory
```

限制使用 sed 把/usr/local/bin 替换为/usr/bin。在下面例子中，sed 默认的分界符/都被\转义了:

```
$ sed 's/\/usr\/local\/bin/\/usr\/bin/' path.txt
reading /usr/bin directory
```

很难看，不是吗？如果要替换一个很长的路径，每个/前面都使用\转义，会显得很混乱。幸运的是，你可以使用任何一个字符作为 sed 替换命令的分界符，如 | 或 ^ 或 @ 或者 !。

下面的所有例子都比较易读。这里不再显示下面例子的结果，因为它们的结果和上面的例子是相同的。我个人比较倾向于使用@或者!来替换路径，但使用什么看你自己的偏好了。

```
sed ‘s|/usr/local/bin|/usr/bin|’ path.txt
sed ‘s^/usr/local/bin^/usr/bin^’ path.txt
sed ‘s@/usr/local/bin@/usr/bin@’ path.txt
sed ‘s!/usr/local/bin!/usr/bin!’ path.txt
```

15.单行内容上执行多个命令

一如之前所说，sed 执行的过程是读取内容、执行命令、打印结果、重复循环。其中执行命令部分，可以由多个命令执行，sed 将一个一个地依次执行它们。

例如，你有两个命令，sed 将在模式空间中执行第一个命令，然后执行第二个命令。如果第一个命令改变了模式空间的内容，第二个命令会在改变后的模式空间上执行(此时模式空间的内容已经不是最开始读取进来的内容了)。

下面的例子演示了在模式空间内执行两个替换命令的过程：

把 Developer 替换为 IT Manager,然后把 Manager 替换为 Director:

```
$ sed '{
> s/Developer/IT Manager/
> s/Manager/Director/
> }' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Director
103,Raj Reddy,Sysadmin
104,Anand Ram,IT Director
105,Jane Miller,Sales Director
```

我们来分析下第 4 行的执行过程：
1.读取数据：在这一步，sed 读取内容到模式空间，此时模式空间的内容为：
104,Anand Ram,Developer
2.执行命令：第一个命令，s/Developer/IT Manager/执行后，模式空间的内容为:
104,Anand Ram,IT Manager
现在在模式空间上执行第二个命令 s/Manager/Director/,执行后，模式空间内容为:
104,Anand Ram,IT Director
谨记：sed 在第一个命令执行的结果上，执行第二个命令。
3.打印内容:打印当前模式空间的内容，如下:
104,Anand Ram,IT Director
4.重复循环：移动的输入文件的下一行，然后重复执行第一步，即读取数据

16.&的作用——获取匹配到的模式

当在 replacement-string 中使用&时，它会被替换成匹配到的 original-string 或正则表达式，这是个很有用的东西。

看下面示例：
给雇员 ID(即第一列的 3 个数字)加上[ ],如 101 改成[101]

```
$ sed 's/^[0-9][0-9][0-9]/[&]/g' employee.txt
[101],John Doe,CEO
[102],Jason Smith,IT Manager
[103],Raj Reddy,Sysadmin
[104],Anand Ram,Developer
[105],Jane Miller,Sales Manager
```

把每一行放进< >中：

```
$ sed 's/^.*/<&>/' employee.txt
<101,John Doe,CEO>
<102,Jason Smith,IT Manager>
<103,Raj Reddy,Sysadmin>
<104,Anand Ram,Developer>
<105,Jane Miller,Sales Manager>
```

17.分组替换(单个分组)

跟在正则表达式中一样，sed 中也可以使用分组。分组以**\(**开始，以**\)**结束。分组可以用在回溯引用中。

回溯引用即重新使用分组所选择的部分正则表达式，在 sed 替换命令的 replacement-string

中和正则表达式中，都可以使用回溯引用。

单个分组：

```
$ sed 's/\([^,]*\).*/\1/g' employee.txt
101
102
103
104
105
```

上面例子中：

- 正则表达式\([^,]*\)匹配字符串从开头到第一个逗号之间的所有字符(并将其放入第一个分组中)
- replacement-string 中的\1 将替代匹配到的分组
- g 即是全局标志

下面这个例子只会显示/etc/passwd 的第一列，即用户名:

```
$ sed 's/\([^:]*\).*/\1/' /etc/passwd
```

下面的例子，如果单词第一个字符为大写，那么会给这个大写字符加上()

```
$ echo "The Geek Stuff"|sed 's/\(\b[A-Z]\)/\(\1\)/g'
(T)he (G)eek (S)tuff
```

请先建立下面文件，以便下面的示例使用：

```
$ vim numbers.txt
1
12
123
1234
12345
123456
```

格式化数字，增加其可读性:

```
$ sed 's/\(^\|[^0-9.]\)\([0-9]\+\)\([0-9]\{3\}\)/\1\2,\3/g' numbers.txt
1
12
123
1,234
12,345
123,456
```

18.分组替换(多个分组)

你可以使用多个\(和\)划分多个分组，使用多个分组时，需要在 replacement-string 中使用\n 来指定第 n 个分组。如下面的示例。

只打印第一列(雇员 ID)和第三列(雇员职位):

```
$ sed 's/^\([^,]*\),\([^,]*\),\([^,]*\)/\1,\3/' employee.txt
101,CEO
102,IT Manager
103,Sysadmin
104,Developer
105,Sales Manager
```

在这个例子中，可以看到，original-string 中，划分了 3 个分组，以逗号分隔。

- ([^,]*\) 第一个分组，匹配雇员 ID
- ，为字段分隔符
- ([^,]*\) 第二个分组，匹配雇员姓名
- ，为字段分隔符
- ([^,]*\) 第三个分组，匹配雇员职位
- ，为字段分隔符，上面的例子演示了如何使用分组
- \1 代表第一个分组(雇员 ID)
- ，出现在第一个分组之后的逗号
- \3 代表第二个分组(雇员职位)

注意：sed 最多能处理 9 个分组，分别用\1 至\9 表示。

交换第一列(雇员 ID)和第二列(雇员姓名)：

```
$ sed 's/^\([^,]*\),\([^,]*\),\([^,]*\)/\2,\1,\3/' employee.txt
John Doe,101,CEO
Jason Smith,102,IT Manager
Raj Reddy,103,Sysadmin
Anand Ram,104,Developer
Jane Miller,105,Sales Manager
```

19.GNU Sed 专有的替换标志

下面的标志，只有 GNU 版的 sed 才能使用。它们可以用在替换命令中的 replacement-string 里面.

\I 标志

当在 replacement-string 中使用\I 标志时，它会把紧跟在其后面的字符当做小写字符来处理。如你所知，下面的例子将把 John 换成 JOHNNY:

```
sed 's/John/JOHNNY/' employee.txt
```

下面的例子，在 replacement-string 中的 H 前面放置了\I 标志，它会把 JOHNNY 中的 H 换成小写的 h:

```
$ sed -n 's/John/JO\lHNNY/p' employee.txt
101,JOhNNY Doe,CEO
```

\L 标志

当在 replacement-string 中使用\I 标志时，它会把后面所有的字符都当做小写字符来处理。

下面的例子，在 replacement-string 中的 H 前面放置了\L 标志,它会把 H 和它后面的所有字符都换成小写：

```
$ sed -n 's/John/JO\LHNNY/p' employee.txt
101,JOhnny Doe,CEO
```

\u 标志

和\I 类似，只不过是把字符换成大写。当在 replacement-string 中使用\u 标志时，它会把紧跟在其后面的字符当做大写字符来处理。下面的例子中，replacement-string 里面的 h 前面有\u 标志，所以 h 将被换成大写的 H:

```
$ sed -n 's/John/jo\uhnny/p' employee.txt
101,joHnny Doe,CEO
```

\U 标志

当在 replacement-string 中使用\U 标志时，它会把后面所有的字符都当做大写字符来处理。下面的例子中，replacement-string 里面的 h 前面有\U 标志，所以 h 及其以后的所有字符，都将被换成大写:

```
$ sed -n 's/John/jo\Uhnny/p' employee.txt
101,joHNNY Doe,CEO
```

\E 标志

\E 标志需要和\U 或\L 一起使用，它将关闭\U 或\L 的功能。下面的例子将把字符串”Johnny Boy”的每个字符都以大写的形式打印出来，因为在 replacement-string 前面使用了\U 标志:

```
$ sed -n 's/John/\UJohnny Boy/p' employee.txt
101,JOHNNY BOY Doe,CEO
```

下面将把 John 换成 JOHNNY Boy:

```
$ sed -n 's/John/\UJohnny\E Boy/p' employee.txt
101,JOHNNY Boy Doe,CEO
```

这个例子只把 Johnny 显示为大写，因为在 Johnny 后面使用了\E 标志(关闭了\U 的功能)

替换标志的用法

上面的例子仅仅展示了这些标志的用法和功能。然而，如果你使用的是具体的字符串，那么这些选项未必有什么作用，因为你可以在需要的地方写出精确的字符串，而不需要使用这些标志进行转换。

和分组配合使用时，这些选项就显得很有用了。前面例子中我们已经学会了如何使用分组调换第一列和第三列的位置。使用上述标志，可以把整个分组转换为小写或大写。

下面的例子，雇员 ID 都显示为大写，职位都显示为小写：

```
$ sed 's/\([^,]*\),\([^,]*\),\([^,]*\)/\U\2\E,\1,\L\3/' employee.txt
JOHN DOE,101,ceo
JASON SMITH,102,it manager
RAJ REDDY,103,sysadmin
ANAND RAM,104,developer
JANE MILLER,105,sales manager
```

这个例子中：

- \U\2\E 把第二个分组转换为大写，然后用\E 关闭转换
- \L\3 把第三个分组转换为小写

第三章：正则表达式

20.正则表达式基础

很多*nix 的命令中，都用到了正则表达式，包括 sed。

行的开头（^）

^ 匹配每一行的开头

显示以 103 开头的行:

```
$ sed -n '/^103/ p' employee.txt
103,Raj Reddy,Sysadmin
```

只有^出现在正则表达式开头时，它才匹配行的开头。所以，^N 匹配所有以 N 开头的行。

行的结尾（$）
$匹配行的结尾。

显示以字符 r 结尾的行：
```
$ sed -n '/r$/ p' employee.txt
102,Jason Smith,IT Manager
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

单个字符（.）
元字符点 . 匹配除换行符之外的任意单个字符。

- . 匹配单个字符
- .. 匹配两个字符
- … 匹配三个字符
- ……以此类推

下面的例子中，模式”J 后面跟三个字符和一个空格”将被替换为”Jason 后面一个空格”。

所以，”J… “同时匹配 employee.txt 文件中的”John “和”Jane “,替换结果如下:
```
$ sed -n 's/J... /Jason /p' employee.txt
101,Jason Doe,CEO
105,Jason Miller,Sales Manager
```

匹配 0 次或多次（*）
星号*匹配 0 个或多个其前面的字符。如：1* 匹配 0 个或多个 1

先建立下面文件：
```
$ vim log.txt
log: input.txt
log:
log:    testing resumed
log:
log:output created
```

假设你想查看那些包含 log 并且后面有信息的行，log 和信息之间可能有 0 个或多个空格，同时不想查看那些 log:后面没有任何信息的行。

显示包含 log:并且 log 后面有信息的行，log 和信息之间可能有空格:
```
$ sed -n '/log: *./p' log.txt
log: input.txt
log:    testing resumed
log:output created
```

注意：上面例子中，后面的点.是必需的，如果没有，sed 只会打印所有包含 log 的行。

匹配一次或多次（\+）

“\+”匹配一次或多次它前面的字符，例如 空格\+ 或 “ \+”匹配至少一个或多个空格。

仍旧使用 log.txt 这个文件来作示例。

显示包含 log:并且 log:后面有一个或多个空格的所有行：

```
$ sed -n '/log: \+/ p' log.txt
log: input.txt
log:    testing resumed
```

注意:这个例子既没有匹配只包含 log:的行，也没有匹配 log:output craeted 这一行，因为 log:后面没有空格。

零次或一次匹配（\?）

\?匹配 0 次或一次它前面的字符。如：

```
$ sed -n '/log: \?/ p' log.txt
log: input.txt
log:
log:    testing resumed
log:
log:output created
```

转义字符（\）

如果要在正则表达式中搜寻特殊字符(如:*,.)，必需使用\来转义它们。

```
$ sed -n '/127\.0\.0\.1/ p' /etc/hosts
127.0.0.1               localhost
```

字符集（[0-9]）

字符集匹配方括号中出现的任意一个字符。

匹配包含 2、3 或者 4 的行:

```
$ sed -n '/[234]/ p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

在方括号中，可以使用连接符-指定一个字符范围。如[0123456789]可以用[0-9]表示，字母可以用[a-z],[A-Z]表示，等等。

匹配包含 2、3 或者 4 的行(另一种方式):

```
$ sed -n '/[2-4]/ p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

21.其他正则表达式

或操作符（|）
管道符号|用来匹配两边任意一个子表达式。子表达式 1|子表达式 2 匹配子表达式 1 或者子表达式 2

打印包含 101 或者包含 102 的行：
```
$ sed -n '/101\|102/ p' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
```
需要注意，| 需要用\转义。

打印包含数字 2~3 或者包含 105 的行：
```
$ sed -n '/[2-3]\|105/ p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
105,Jane Miller,Sales Manager
```

精确匹配 m 次（{m}）
正则表达式后面跟上{m}标明精确匹配该正则 m 次。

请先建立如下文件:
```
$ vi numbers.txt
1
12
123
1234
12345
123456
```

打印包含任意数字的行(这个命令将打印所有行):
```
$ sed -n '/[0-9]/ p' numbers.txt
1
12
123
1234
12345
123456
```

打印包含 5 个数字的行:
```
$ sed -n '/^[0-9]\{5\}$/ p' numbers.txt
12345
```
注意这里一定要有 s 开头和结尾符号，即^和$,并且{和}都要用\转义

匹配 m 至 n 次（{m,n}）:
正则表达式后面跟上{m,n}表明精确匹配该正则至少 m，最多 n 次。m 和 n 不能是负数，并且要小于 255.

打印由 3 至 5 个数字组成的行：
```
$ sed -n '/^[0-9]\{3,5\}$/ p' numbers.txt
123
1234
12345
```
正则表达式后面跟上{m,}表明精确匹配该正则至少 m，最多不限。(同样，如果是{,n}表明最多匹配 n 次，最少一次)。

字符边界（\b）
\b 用来匹配单词开头(\bxx)或结尾(xx\b)的任意字符，因此\bthe\b 将匹配 the,但不匹配 they. \bthe 将匹配 the 或 they.

请先建立如下文件:
```
$ cat words.txt
word matching using: the
word matching using: thethe
word matching using: they
```

匹配包含 the 作为整个单词的行：
```
$ sed -n '/\bthe\b/ p' words.txt
word matching using: the
```
注意：如果没有后面那个\b,将匹配所有行。

匹配所有以 the 开头的单词:
```
$ sed -n '/\bthe/ p' words.txt
word matching using: the
word matching using: thethe
word matching using: they
```

回溯引用（\n）
使用回溯引用，可以给正则表达式分组，以便在后面引用它们。

只匹配重复 the 两次的行:
```
$ sed -n '/\(the\)\1/ p' words.txt
word matching using: thethe
```
同理，”\([0-9]\)\1” 匹配连续两个相同的数字，如 11,22,33 .....

22.在 sed 替换中使用正则表达式

下面是一些使用正则表达式进行替换的例子。

把 employee.txt 中每行最后两个字符替换为",Not Defined":

```
$ sed -n 's/..$/,Not Defined/ p' employee.txt
101,John Doe,C,Not Defined
102,Jason Smith,IT Manag,Not Defined
103,Raj Reddy,Sysadm,Not Defined
104,Anand Ram,Develop,Not Defined
105,Jane Miller,Sales Manag,Not Defined
```

删除以 Manager 开头的行的后面的所有内容:

```
$ sed    's/^Manager.*//' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

这个示例好像有点问题啊，我觉得应该是 sed 's/^Manager.*/Manager/'

删除所有以#开头的行：

```
$ sed -e 's/#.*// ; /^$/ d' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

我觉得用 sed '/^#/ d' 更好

建立下面的 test.html 文件:

```
$ vim test.html
<html><body><h1>Hello World!</h1></body></html>
```

清除 test.html 文件中的所有 HTML 标签:

```
$ sed 's/<[^>]*>//g' test.html
Hello World!
```

删除所有注释行和空行:

```
$ sed -e 's/#.*// ; /^$/ d' /etc/profile
```

只删除注释行，保留空行:

```
sed '/#.*/ d' /etc/profile
```

使用 sed 可以把 DOS 的换行符(CR/LF)替换为 Unix 格式。当把 DOS 格式的文件拷到 Unix 上，你会发现，每行结尾都有\r\n .

使用 sed 把 DOS 格式的文件转换为 Unix 格式:

```
sed ‘s/.$//’ filename
```

第四章：执行 sed

23.单行内执行多个 sed 命令

第一章内已经讲过，单行内执行多个 sed 命令有多种方法。

1. 使用多命令选项 –e

多命令选项-e 使用方法如下:

```
sed –e ‘command1’ –e ‘command2’ –e ‘command3’
```

在/etc/passwd 文件中，搜索 root、nobody 或 mail:

```
$ sed -n -e '/^root/ p' -e '/^nobody/ p' -e '/^mail/ p' /etc/passwd
```

2. 使用\ 折行执行多个命令

在执行很长的命令，比如使用-e 选项执行多个 sed 命令时，可以使用\来把命令折到多行

```
sed –n –e ‘/^root/ p’ \
-e ‘/^nobody/ p’ \
-e ‘/^mail/ p’ \
/etc/passwd
```

3. 使用{ }把多个命令组合

如果要执行很多 sed 命令，可以使用{ }把他们组合起来执行，如:

```
sed –n ‘{
/^root/ p
/^nobody/ p
/^mail/ p
}’ /etc/passwd
```

24.sed 脚本文件

如果用重复使用一组 sed 命令，那么可以建立 sed 脚本文件，里面包含所有要执行的 sed 命令，然后用-f 选项来使用。

首先建立下面文件，里面包含了所有要执行的 sed 命令。前面已经解释过各个命令的含义，现在你应该知道所有命令的意思了。

```
$ vi mycommands.sed
s/\([^,]*\),\([^,]*\),\(.*\).*/\2,\1, \3/g
s/^.*/<&>/
s/Developer/IT Manager/
s/Manager/Director/
```

现在执行脚本里面的命令:

```
$ sed -f mycommands.sed employee.txt
<John Doe,101, CEO>
<Jason Smith,102, IT Director>
<Raj Reddy,103, Sysadmin>
<Anand Ram,104, IT Director>
<Jane Miller,105, Sales Director>
```

25.sed 注释

sed 注释以#开头。因为 sed 是比较晦涩难懂的语言，所以你现在写下的 sed 命令，时间一长，再看时就不那么容易理解了。因此，建议把写脚本时的初衷作为注释，写到脚本里面。如下所示：

```
$ vim mycommands.sed
#交换第一列和第二列
s/\([^,]*\),\([^,]*\),\(.*\).*/\2,\1, \3/g
#把整行内容放入<>中
s/^.*/<&>/
#把 Developer 替换为 IT Manager
s/Developer/IT Manager/
#把 Manager 替换为 Director
s/Manager/Director/
```

注意：如果 sed 脚本第一行开始的两个字符是#n 的话，sed 会自动使用-n 选项(即不自动打印模式空间的内容)

26.把 sed 当做命令解释器使用

一如你可以把命令放进一个 shell 脚本中，然后调用脚本名称来执行它们一样，你也可以把 sed 用作命令解释器。要实现这个功能，需要在 sed 脚本最开始加入”#!/bin/sed –f”，如下所示:

```
$ vi myscript.sed
#!/bin/sed -f
#交换第一列和第二列
s/\([^,]*\),\([^,]*\),\(.*\).*/\2,\1, \3/g
#把整行内容放入<>中
```

```
s/^.*/<&>/
#把 Developer 替换为 IT Manager
s/Developer/IT Manager/
#把 Manager 替换为 Director
s/Manager/Director/
```

现在，给这个脚本加上可执行权限,然后直接在命令行调用它:

```
$ chmod u+x myscript.sed
```

```
$ ./myscript.sed employee.txt
<John Doe,101, CEO>
<Jason Smith,102, IT Director>
<Raj Reddy,103, Sysadmin>
<Anand Ram,104, IT Director>
<Jane Miller,105, Sales Director>
```

也可以指定-n 选项来屏蔽默认输出:

```
$ vim testscript.sed
#!/bin/sed -nf
/root/ p
/nobody/ p
/mail/ p
```

然后加上可执行权限，执行:

```
$ chmod u+x testscript.sed
```

```
$ ./testscript.sed /etc/passwd
root:x:0:0:root:/root:/bin/bash
mail:x:8:12:Mailer daemon:/var/spool/clientmqueue:/bin/false
nobody:x:65534:65533:nobody:/var/lib/nobody:/bin/bash
```

处于测试目的，把 testscript.sed 里面的-n 去掉，然后再执行一次，观察它是如何运行的。

重要提示:使用-n 时，必须是-nf.如果你写成-fn,执行脚本时就会获得下面的错误:

```
$ ./testscript.sed /etc/passwd
/bin/sed: couldn't open file n: No such file or directory
```

27.直接修改输入文件

目前为止，你知道 sed 默认不会修改输入文件，它只会把输出打印到标准输出上。当想保存结果时，把输出重定向到文件中(或使用 w 命令)。

执行下面的例子之前，先备份一下 employee.txt 文件:

```
$ cp employee.txt employee.txt.orig
```

为了修改输入文件，通常方法是把输出重定向到一个临时文件，然后重命名该临时文件:
sed ‘s/John/Johnny/’ employee.txt > new-employee.txt
mv new-employee.txt employee.txt

相比这种传统方法，可以在 sed 命令中使用-i 选项，使 sed 可以直接修改输入文件:
在原始文件 employee.txt 中，把 John 替换为 Johnny:
$ sed -i 's/John/Johnny/' employee.txt

$ cat employee.txt
101,**Johnny** Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager

再次提醒: **-i 会修改输入文件**。或许这正是你想要的，但是务必小心。一个保护性的措施是，在-i 后面加上备份扩展，这一 sed 就会在修改原始文件之前，备份一份。

在原始文件 employee.txt 中，把 John 替换为 Johnny，但在替换前备份 employee.txt:
$ sed -ibak 's/John/Johnny/' employee.txt
备份的文件如下:
$ cat employee.txtbak
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
修改后的原始文件为:
$ cat employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager

除了使用-i,也可以使用完整样式 –in-place.下面两个命令是等价的:
sed –ibak ‘s/John/Johnny/’ employee.txt
sed –in-place=bak ‘s/John/Johnny/’ employee.txt

最后，为了继续下面的例子，把原来的 employee.txt 还原回去:
cp employee.txt.orig employee.txt

第五章：sed 附加命令

28.追加命令(命令 a)

使用命令 a 可以在指定位置的后面插入新行。

语法：

```
$ sed ‘[address] a the-line-to-append’ input-file
```

在第 2 行后面追加一行(原文这里可能有问题，没有写明行号):

```
$ sed '2 a 203,Jack Johnson,Engineer' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
203,Jack Johnson,Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

在 employee.txt 文件结尾追加一行:

```
$ sed '$ a 106,Jack Johnson,Engineer' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Jack Johnson,Engineer
```

sed 也可以追加多行。

在匹配 Jason 的行的后面追加两行:

```
$ sed '/Jason/a\
> 203,Jack Johnson,Engineer\
> 204,Mark Smith,Sales Engineer' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
203,Jack Johnson,Engineer
204,Mark Smith,Sales Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

追加多行之间可以用\n 来换行，这样就不用折行了,上面的命令等价于:

```
sed '/Jason/a 203,Jack Johnson,Engineer\n204,Mark Smith,Sales Engineer' employee.txt
```

29.插入命令(命令 i)

插入命令 **i**nsert 命令和追加命令类似，只不过是在指定位置之前插入行。

语法：

```
$ sed ‘[address] i the-line-to-insert’ input-file
```

在 employee.txt 的第 2 行之前插入一行:

```
$ sed '2 i 203,Jack Johnson,Engineer' employee.txt
101,Johnny Doe,CEO
203,Jack Johnson,Engineer
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

在 employee.txt 最后一行之前，插入一行:

```
$ sed '$ i 108,Jack Johnson,Engineer' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
108,Jack Johnson,Engineer
105,Jane Miller,Sales Manager
```

sed 也可以插入多行。

在匹配 Jason 的行的前面插入两行:

```
$ sed '/Jason/i\
> 203,Jack Johnson,Engineer\
> 204,Mark Smith,Sales Engineer' employee.txt
101,Johnny Doe,CEO
203,Jack Johnson,Engineer
204,Mark Smith,Sales Engineer
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

30.修改命令(命令 c)

修改命令 **c**hange 可以用新行取代旧行。

语法：

```
$ sed ‘[address] c the-line-to-insert’ input-file
```

用新数据取代第 2 行:

```
$ sed '2 c 202,Jack,Johnson,Engineer' employee.txt
101,Johnny Doe,CEO
202,Jack,Johnson,Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

这里命令 c 等价于替换：$ sed '2s/.*/202,Jack,Johnson,Engineer/' employee.txt

sed 也可以用多行来取代一行。

用两行新数据取代匹配 Raj 的行:

```
$ sed '/Raj/c\
> 203,Jack Johnson,Engineer\
> 204,Mark Smith,Sales Engineer' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
203,Jack Johnson,Engineer
204,Mark Smith,Sales Engineer
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

31.命令 a、i 和 c 组合使用

命令 a、i 和 c 可以组合使用。下面的例子将完成三个操作：

- a 在”Jason”后面追加”Jack Johnson”
- i 在”Jason”前面插入”Mark Smith”
- c 用”Joe Mason”替代”Jason”

```
$ sed '/Jason/ {
> a\
> 204,Jack Johnson,Engineer
> i\
> 202,Mark Smith,Sales Engineer
> c\
> 203,Joe Mason,Sysadmin
```

```
> }' employee.txt
101,Johnny Doe,CEO
202,Mark Smith,Sales Engineer
203,Joe Mason,Sysadmin
204,Jack Johnson,Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

32.打印不可见字符(命令 l)

命令 l 可以打印不可见的字符，比如制表符\t，行尾标志$等。

请先建立下面文件以便用于后续测试，请确保字段之间使用制表符(Tab 键)分开:

```
$ cat tabfile.txt
fname	First Name
lname	Last Name
mname	Middle Name
```

使用命令 l，把制表符显示为\t,行尾标志显示为 EOL:

```
$ sed -n 'l' tabfile.txt
fname\tFirst Name$
lname\tLast Name$
mname\tMiddle Name$
```

如果在 l 后面指定了数字，那么会在第 n 个字符处使用一个不可见自动折行，效果如下:

```
$ sed -n 'l 20' employee.txt
101,Johnny Doe,CEO$
102,Jason Smith,IT \
Manager$
103,Raj Reddy,Sysad\
min$
104,Anand Ram,Devel\
oper$
105,Jane Miller,Sal\
es Manager$
```

这个功能只有 GNU sed 才有。

33.打印行号(命令=)

命令=会在每一行后面显示该行的行号。

打印所有行号:

```
$ sed '=' employee.txt
1
101,Johnny Doe,CEO
2
102,Jason Smith,IT Manager
3
103,Raj Reddy,Sysadmin
4
104,Anand Ram,Developer
5
105,Jane Miller,Sales Manager
```

提示：把命令=和命令 N 配合使用，可以把行号和内容显示在同一行上(下文解释)。

只打印 1,2,3 行的行号:

```
$ sed '1,3 =' employee.txt
1
101,Johnny Doe,CEO
2
102,Jason Smith,IT Manager
3
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

打印包含关键字"Jane"的行的行号，同时打印输入文件中的内容：

```
$ sed '/Jane/ =' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
5
105,Jane Miller,Sales Manager
```

如果你想只显示行号但不显示行的内容，那么使用-n 选项来配合命令=:

```
$ sed -n '/Raj/ =' employee.txt
3
```

打印文件的总行数:

```
$ sed -n '$ =' employee.txt
5
```

34.转换字符(命令 y)

命令 y 根据对应位置转换字符。好处之一便是把大写字符转换为小写，反之亦然。

下面例子中，将把 a 换为 A，b 换为 B，c 换为 C，以此类推:

```
$ sed 'y/abcde/ABCDE/' employee.txt
101,Johnny DoE,CEO
102,JAson Smith,IT MAnAgEr
103,RAj REDDy,SysADmin
104,AnAnD RAm,DEvElopEr
105,JAnE MillEr,SAlEs MAnAgEr
```

把所有小写字符转换为大写字符:

```
$ sed 'y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/' employee.txt
101,JOHNNY DOE,CEO
102,JASON SMITH,IT MANAGER
103,RAJ REDDY,SYSADMIN
104,ANAND RAM,DEVELOPER
105,JANE MILLER,SALES MANAGER
```

35.操作多个文件

之前的例子都只操作了单个文件，sed 也可以同时处理多个文件。

在/etc/passwd 中搜索 root 并打印出来：

```
$ sed -n '/root/ p' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

在/etc/group 中搜索 root 并打印出来：

```
$ sed -n '/root/ p' /etc/group
root:x:0:
```

同时在/etc/passwd 和/etc/group 中搜索 root:

```
$ sed -n '/root/ p' /etc/passwd /etc/group
root:x:0:0:root:/root:/bin/bash
root:x:0:
```

36.退出 sed(命令 q)

命令 q 终止正在执行的命令并退出 sed。

之前提到，正常的 sed 执行流程是：读取数据、执行命令、打印结果、重复循环。
当 sed 遇到 q 命令，便立刻退出，当前循环中的后续命令不会被执行，也不会继续循环。

打印第 1 行后退出:

```
$ sed 'q' employee.txt
101,Johnny Doe,CEO
```

打印第 5 行后退出，即只打印前 5 行：

```
$ sed '5 q' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

打印所有行，直到遇到包含关键字 Manager 的行:

```
$ sed '/Manager/ q' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
```

注意：q 命令不能指定地址范围(或模式范围),只能用于单个地址(或单个模式)。

37.从文件读取数据(命令 r)

在处理输入文件是，命令 r 会从另外一个文件读取内容，并在指定的位置打印出来。下面的例子将读取 log.txt 的内容，并在打印 employee.txt 最后一行之后，把读取的内容打印出来。事实上它把 employee.txt 和 log.txt 合并然后打印出来。

```
$ sed '$ r log.txt' employee.txt
```

也可以给命令 r 指定一个模式。下面的例子将读取 log.txt 的内容，并且在匹配'Raj'的行后面打印出来：

```
$ sed '/Raj/ r log.txt' employee.txt
```

38.用 sed 模拟 Unix 命令(cat,grep,read)

之前的例子的完成的功能都很像标准的 Unix 命令。使用 sed 可以模拟很多 Unix 命令。完成它们，以便更好地理解 sed 的工作过程。

模拟 cat 命令

```
cat employee.txt
```

下面的每个 sed 命令的输出都和上面的 cat 命令的输出一样：

```
sed 's/JUNK/&/p' employee.txt
sed –n 'p' employee.txt
sed 'n' employee.txt
sed 'N' employee.txt
```

模拟 grep 命令

```
grep Jane employee.txt
```

下面的每个 sed 命令的输出都和上面的 grep 命令的输出一样：

```
sed –n 's/Jane/&/ p' employee.txt
sed –n '/Janme/ p' employee.txt
```

grep –v (打印不匹配的行):

```
sed –n '/Jane/ !p' employee.txt
```

注意：这里不能使用 sed –n 's/Jane/&/ !p' 了。

模拟 head 命令

```
head -10 /etc/passwd
```

下面的每个 sed 命令的输出都和上面的 head 命令的输出一样：

```
sed '11,$ d' /etc/passwd
sed –n '1,10 p' /etc/passwd
sed    '10 q' /etc/passwd
```

39.sed 命令选项

-n 选项

之前已经讨论过多次，并且在很多例子中也使用到了-n 选项。该选项屏蔽 sed 的默认输出。也可以使用—quiet,或者—silent 来代替-n，它们的作用是相同的。

下面所有命令都是等价的:

```
sed -n 'p' employee.txt
sed --quiet 'p' employee.txt
sed –silent 'p' employee.txt
```

-f 选项

可以把多个 sed 命令保存在 sed 脚本文件中，然后使用-f 选项来调用，这点之前已经演示过

了。你也可以使用—file 来代替-f

下面的命令是等价的:

```
sed –n –f test-script.sed /etc/passwd
sed –n –file=test-script.sed /etc/passwd
```

-e 选项

该选项执行来自命令行的一个 sed 命令，可以使用多个-e 来执行多个命令。也可以使用—expression 来代替。

下面所有命令都是等价的:

```
sed –n –e '/root/ p' /etc/passwd
sed –n –expression '/root/ p' /etc/passwd
```

-i 选项

下面所有命令都是等价的:

我们已经提到，sed 不会修改输入文件，只会把内容打印到标准输出,或则使用 w 命令把内容写到不同的文件中。我们也展示了如何使用-i 选项来直接修改输入文件。

在原始文件 employee.txt 中，用 Johnny 替换 John:

```
sed –i 's/John/Johnny/' employee.txt
```

执行和上面相同的命令，但在修改前备份原始文件:

```
sed –ibak 's/John/Johnny/' employee.txt
```

也可以使用--in-place 来代替-i:

下面的命令是等价的:

```
sed -ibak 's/John/Johnny/' employee.txt
sed --in-place=bak 's/John/Johnny/' employee.txt
```

-c 选项

该选项应和-i 配合使用。使用-i 时，通常在命令执行完成后，sed 使用临时文件来保持更改后的内容，然后把该临时文件重命名为输入文件。但这样会改变文件的所有者(奇怪的是我的测试结果是它不会改变文件所有者)，配合 c 选项，可以保持文件所有者不变。也可以使用--copy 来代替。

下面的命令是等价的:

```
sed -ibak -c 's/John/Johnny/' employee.txt
sed --in-place=bak --copy 's/John/Johnny/' employee.txt
```

-l 选项

指定行的长度，需要和 l 命令配合使用(注意**选项 l** 和**命令 l**，不要弄混了，上面提到的命令 i 和选项 i 也不要搞错)使用-l 选项即指定行的长度。也可以使用--line-length 来代替.

下面的命令是等价的:

```
sed -n -l 20 'l' employee.txt
sed -n --line-length=20 employee.txt
```

请注意，下面的例子不使用-l(原文是-n，应该是弄错)选项，同样可以获取相同的输出:

```
sed -n 'l 20' employee.txt --posix option
```

40.打印模式空间(命令 n)

命令 n 打印当前模式空间的内容，然后从输入文件中读取下一行。如果在命令执行过程中遇到 n，那么它会改变正常的执行流程。

打印每一行在模式空间中的内容：

```
$ sed n employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

如果使用了-n 选项，将没有任何输出：

```
$ sed -n n employee.txt
$
```

(再次提醒：不要把选项-n 和命令 n 弄混了)

前面提到，sed 正常流程是读取数据、执行命令、打印输出、重复循环。
命令 n 可以改变这个流程，它打印当前模式空间的内容，然后清除模式空间，读取下一行进来，然后继续执行后面的命令。

假定命令 n 前后各有两个其他命令，如下:

```
sed-command-1
sed-command-2
n
sed-command-3
sed-command-4
```

这种情况下，sed-command-1 和 sed-command-2 会在当前模式空间中执行，然后遇到 n，它打印当前模式空间的内容，并清空模式空间，读取下一行，然后把 sed-command-3 和 sed-command-4 应用于新的模式空间的内容。

提示：上面的例子中可以看到，命令 n 本身没有多大用处，然而当它和保持空间的命令配合使用时，就很强大了，后面会详细解释。

第六章：保持空间和模式空间命令

Sed 有两个内置的存储空间：

- **模式空间:**如你所知，模式空间用于 sed 执行的正常流程中。该空间 sed 内置的一个缓冲区，用来存放、修改从输入文件读取的内容。

- **保持空间：**保持空间是另外一个缓冲区，用来存放临时数据。Sed 可以在保持空间和模式空间交换数据，但是不能在保持空间上执行普通的 sed 命令。我们已经讨论过，每次循环读取数据过程中，模式空间的内容都会被清空，然而保持空间的内容则保持不变，不会在循环中被删除。

请先建立如下文件，以用于保持空间的示例:

```
$ vi empnametitle.txt
John
Doe
CEO
Jason Smith
IT Manager
Raj Reddy
Sysadmin
Anand Ram
Developer
Jane Miller
Sales Manager
```

可以看到，在这个文件中，每个雇员的名称和职位位于连续的两行内。

41.用保持空间替换模式空间(命令 x)

命令 x(Exchange)交换模式空间和保持空间的内容。该命令本身没有多大用处，但如果和其他命令配合使用，则非常强大了。

假定目前模式空间内容为”line 1”，保持空间内容为”line 2”。那么执行命令 x 后，模式空间的内容变为”line 2”,保持空间的内容变为”line 1”。

下面的例子打印管理者的名称，它搜索关键字’Manager’并打印之前的那一行:

```
$ sed –n -e '{x;n}’ –e ‘/Manager/{x;p}' empnametitle.txt
Jason Smith
Jane Miller
```

上面命令也可写成: sed -n 'x;n;/Manager/{x;p}' empnametitle.txt

提示：如果你的 empnametitle.txt 文件，雇员名称和职位不是连续的，那么得不到上面的结果。

上面例子中:

- **{x;n}** – x 交换模式空间和保持空间的内容；n 读取下一行到模式空间。在示例文件中，保持空间保存的是雇员名称，模式空间保存的是职位。
- **/Manager/{x;}** – 如果当前模式空间的内容包含关键’Manager’，那么就交换保持空间和模式空间的内容，然后打印模式空间的内容。这就意味着，如果雇员职位中包含 Manager,那么该雇员的名称将被打印出来。

你也可以把上述命令保存在 sed 脚本中，然后执行，如下所示：

```
$ vim x.sed
#!/bin/sed -nf
x;n
/Manager/{x;p}

$ chmod u+x x.sed

$ ./x.sed empnametitle.txt
Jason Smith
Jane Miller
```

42.把模式空间的内容复制到保持空间(命令 h)

命令 h(hold)把模式空间的内容复制到保持空间，和命令 x 不同，命令 h 不会修改当前模式空间的内容。执行命令 h 时，当前保持空间的内容会被模式空间的内容覆盖。

假定目前模式空间内容为”line 1”，保持空间内容为”line 2”。那么执行命令 h 后，模式空间的内容仍然为”line 1”,保持空间的内容则变为”line 1”。

打印管理者的名称:

```
$ sed -n -e '/Manager/!h' -e '/Manager/{x;p}' empnametitle.txt
Jason Smith
Jane Miller
```

上面例子中：

- **/Manager/!h –**如果模式空间内容不包含关键字’Manager’(模式后面的!表示不匹配该模式)，那么复制模式空间内容到保持空间。(这样一来，保持空间的内容可能会是雇员名称或职位，而不是’Manager’.)注意，和前面的例子不同，这个例子中没有使用命令 n 来获取下一行，而是通过正常的流程来读取后续内容。
- **/Manager/{x;p} –**如果模式空间内容包含关键字’Manager’，那么交换保持空间和模式空间的内容，并打印模式空间的内容。这和我们在命令 x 的示例中的用法是相同的。

你也可以把上面命令保存到 sed 脚本中执行：

```
$ vi h.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{x;p}

$ chmod u+x h.sed

$ ./h.sed empnametitle.txt
```

```
Jason Smith
Jane Miller
```

43.把模式空间内容追加到保持空间(命令 H)

大写 H 命令表示把模式空间的内容追加到保持空间，追加之前保持空间的内容不会被覆盖；相反，它在当前保持空间内容后面加上换行符\n，然后把模式空间内容追加进来。

假定目前模式空间内容为”line 1”，保持空间内容为”line 2”。那么执行命令 H 后，模式空间的内容没有改变，仍然为”line 1”,保持空间的内容则变为”line2\nline 1”。

打印管理者的名称和职位(在不同的行上):

```
$ sed -n -e '/Manager/!h' -e '/Manager/{H;x;p}' empnametitle.txt
Jason Smith
IT Manager
Jane Miller
Sales Manager
```

上面例子中:

- **/Manager/!h** –如果模式空间内容不包含关键字’Manager’(模式后面的!表示不匹配该模式)，那么复制模式空间内容到保持空间。(这样一来，保持空间的内容可能会是雇员名称或职位，而不是’Manager’.)。这和之前使用命令 h 的方法是一样的。
- **/Manager/{H;x;p}** –如果模式空间内容包含关键字’Manager’，那么命令 H 把模式空间的内容(也就是管理者的职位)作为新行追加到保持空间，所以保持空间内容会变为”雇员名称\n 职位”(职位包含关键字 Manager)。然后命令 x 交换模式空间和保持空间的内容，随后命令 p 打印模式空间的内容。

你也可以把上面命令保存到 sed 脚本中执行：

```
$ vi H-upper.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{H;x;p}

$ chmod u+x H-upper.sed

$ ./H-upper.sed empnametitle.txt
Jason Smith
IT Manager
Jane Miller
Sales Manager
```

如果想把雇员名称和职位显示在同一行，以分号分开，那么只需稍微修改一下即可，如下：

```
$ sed -n -e '/Manager/!h' -e '/Manager/{H;x;;s/\n/:/p}' empnametitle.txt
Jason Smith:IT Manager
Jane Miller:Sales Manager
```

这个例子除了在第二个-e 后面的命令中加入了替换命令之外，和前面的例子一样。H、x 和 p 都完成和之前相同的操作；在交换模式空间和保持空间之后，命令 s 把换行符\n 替换为分号，然后打印出来。

你也可以把上面命令保存到 sed 脚本中执行：

```
$ vi H1-upper.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{H;x;s/\n/:/;p}

$ chmod u+x H1-upper.sed

$ ./H1-upper.sed empnametitle.txt
Jason Smith:IT Manager
Jane Miller:Sales Manager
```

44.把保持空间内容复制到模式空间(命令 g)

命令 g(get)把保持空间的内容复制到模式空间。

这样理解：命令 h 保持(hold)住保持空间(hold space)，命令 g 从保持空间获取(get)内容。

假定当前模式空间内容为”line 1”,保持空间内容为”line 2”;执行命令 g 之后，模式空间内容变为”line 2”，保持空间内容仍然为”line 2”。

打印管理者的名称:

```
$ sed -n -e '/Manager/!h' -e '/Manager/{g;p}' empnametitle.txt
Jason Smith
Jane Miller
```

上面例子中:

- **/Manager/!h –**最近几个例子都在用它。如果模式空间内容不包含关键字’Manager’,那么就把他复制到保持空间。
- **/Manager/{g;p} –**把保持空间的内容丢到模式空间中，然后打印出来

你也可以把上面命令保存到 sed 脚本中执行：

```
$ vi g.sed
```

```
#!/bin/sed -nf
/Manager/!h
/Manager/{g;p}

$ chmod u+x g.sed

$ ./g.sed empnametitle.txt
Jason Smith
Jane Miller
```

45.把保持空间追加到模式空间(命令 G)

大写 G 命令把当前保持空间的内容作为新行追加到模式空间中。模式空间的内容不会被覆盖，该命令在模式空间后面加上换行符\n，然后把保持空间内容追加进去。

G 和 g 的用法类似于 H 和 h；小写命令替换原来的内容，大写命令追加原来的内容。

假定当前模式空间内容为”line 1”，保持空间内容为”line 2”;命令 G 执行后，模式空间内容变为”line 1\nline 2”,同时保持空间内容不变，仍然为”line 2”。

以分号分隔，打印管理者的名称和职位:

```
$ sed -n -e '/Manager/!h' -e '/Manager/{x;G;s/\n/:/;p}' empnametitle.txt
Jason Smith:IT Manager
Jane Miller:Sales Manager
```

上面例子中：

- **/Manager/!h –**和前面的例子一样，如果模式空间内容不包含关键字’Manager’,那么就把他复制到保持空间。
- **/Manager/{x;G;s/\n/:/;p} –**如果模式空间包含’Manager’,那么:
  - **x –**交换模式空间和保持空间的内容。
  - **G –**把保持空间的内容追加到模式空间。
  - **s/\n/:/** --在模式空间中，把换行符替换为分号:。
  - **p** 打印模式空间内容
  - 注意：如果舍去命令 x，即使用/Manager/{G;s/\n/:/;p},那么结果会由“雇员职位: 雇员名称”变成”雇员名称: 雇员职位”

也可把上述命令写到 sed 脚本中然后执行:

```
$ vi G-upper.sed
#!/bin/sed -nf
/Manager/!h
/Manager/{x;G;s/\n/:/;p}

$ chmod u+x G-upper.sed
```

```
$ ./G-upper.sed empnametitle.txt
Jason Smith:IT Manager
Jane Miller:Sales Manager
```

第七章：sed 多行模式及循环

Sed 默认每次只处理一行数据，除非使用 H,G 或者 N 等命令创建多行模式，每行之间用换行符分开。

本章将解释适用于多行模式的 sed 命令。

提示：在处理多行模式是，请务必牢记^只匹配该模式的开头，即最开始一行的开头，且$只匹配该模式的结尾，即最后一行的结尾。

46.读取下一行数据并附加到模式空间(命令 N)

就像大写的命令 H 和 G 一样，只会追加内容而不是替换内容，命令 N 从输入文件中读取下一行并追加到模式空间，而不是替换模式空间。

前面提到过，小写命令 n 打印当前模式空间的内容，并清空模式空间，从输入文件中读取下一行到模式空间，然后继续执行后面的命令。

大写命令 N，不会打印模式空间内容，也不会清除模式空间内容，而是在当前模式空间内容后加上换行符\n,并且从输入文件中读取下一行数据，追加到模式空间中，然后继续执行后面的命令。

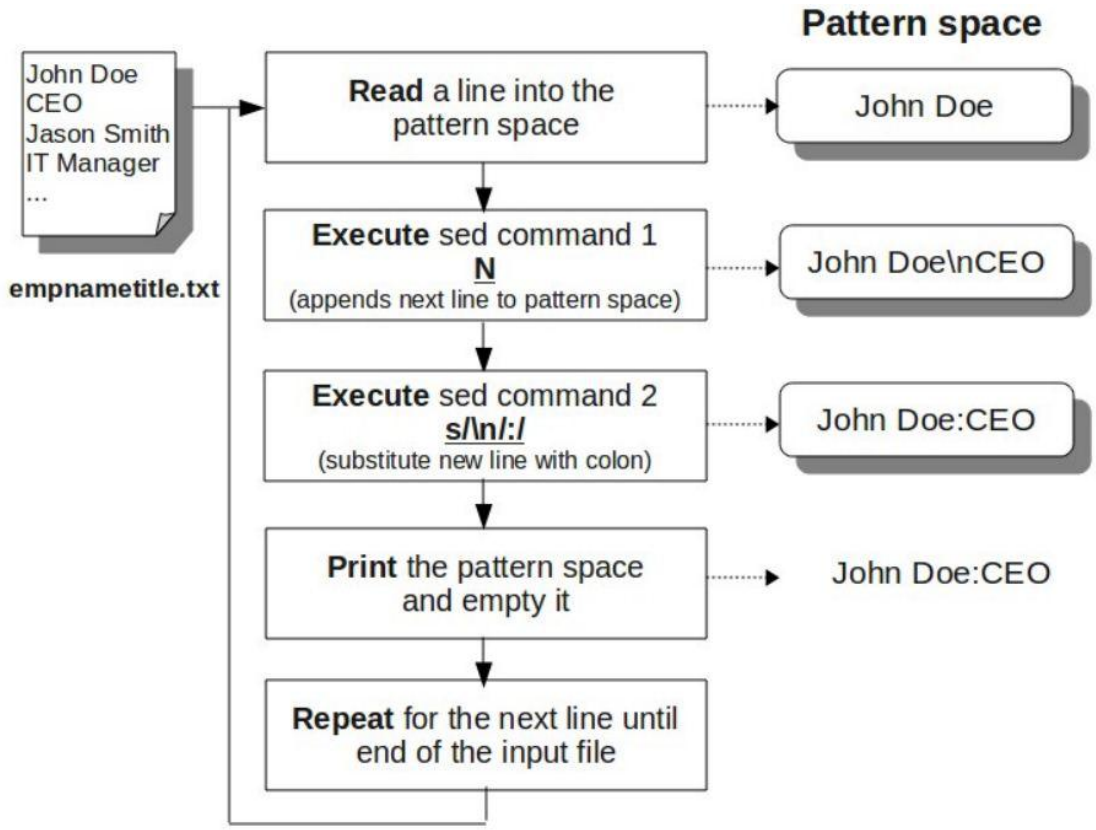
以分号分隔，打印雇员名称和职位:

```
$ sed -e '{N;s/\n/:/}' empnametitle.txt
John Doe:CEO
Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
Jane Miller:Sales Manager
```

这个例子中:

- **N** 追加换行符\n 到当前模式空间(雇员名称)的最后，然后从输入文件读取下一行数据，追加进来。因此，当前模式空间内容变为”雇员名称\n 雇员职位”。
- **s/\n/:/** 把换行符\n 替换为分号，把分号作为雇员名称和雇员职位的分隔符

流程如下图所示:



下面的例子将演示**在打印 employee.txt 文件内容的同时，以文本方式显示每行的行号：**

```
$ sed -e '=' employee.txt|sed '{N;s/\n/ /}'
1 101,Johnny Doe,CEO
2 102,Jason Smith,IT Manager
3 103,Raj Reddy,Sysadmin
4 104,Anand Ram,Developer
5 105,Jane Miller,Sales Manager
```

和之前的例子一样，命令=先打印行号，然后但印原始的行的内容。

这个例子中，命令 N 在当前模式空间后面加上\n(当前模式空间内容为行号),然后读取下一行，并追究到模式空间中。因此，模式空间内容变为”行号\n 原始内容”。然后用 s/\n/ /把换行符\n 替换成空格。

47.打印多行模式中的第一行(命令 P)

目前为止，我们已经学会了三个大写的命令(H,N,G)，每个命令都是追加内容而不是替换内容。现在我们来看看大写的 D 和 P，虽然他们的功能和小写的 d 和 p 非常相似，但他们在多行模式中有特殊的功能。

之前说到，小写的命令 p 打印模式空间的内容。大写的 P 也打印模式空间内容，直到它遇到换行符\n。

下面的例子将打印所有管理者的名称:

```
$ sed -n -e 'N' -e '/Manager/P' empnametitle.txt
Jason Smith
Jane Miller
```

48. 删除多行模式中的第一行(命令 D)

之前提到，小写命令 d 会删除模式空间内容，然后读取下一条记录到模式空间，并忽略后面的命令，从头开始下一次循环。

大写命令 D，既不会读取下一条记录，也不会完全清空模式空间(除非模式空间内只有一行)。它只会：

- 删除模式空间的部分内容，直到遇到换行符\n
- 忽略后续命令，在当前模式空间中从头开始执行命令

假设有下面文件，没个雇员的职位都用@包含起来作为注释。需要注意的是，有些注释是跨行的。如@Information Technology officer@就跨了两行。请先建立下面示例文件:

```
$ vi empnametitle-with-commnet.txt
John Doe
CEO @Chief Executive Officer@
Jason Smith
IT Manager @Infromation Technology
Officer@
Raj Reddy
Sysadmin @System Administrator@
Anand Ram
Developer @Senior
Programmer@
Jane Miller
Sales Manager @Sales
Manager@
```

现在我们的目标是，去掉文件里的注释:

```
$ sed -e '/@/{N;/@.*@/{s/@.*@//;P;D}}' empnametitle-with-commnet.txt
John Doe
CEO
Jason Smith
IT Manager
Raj Reddy
Sysadmin
Anand Ram
Developer
Jane Miller
```

```
Sales Manager
```

也可把上述命令写到 sed 脚本中然后执行:

```
$ vi D-upper.sed
#!/bin/sed -f
/@/{
N
/@.*@/{s/@.*@//;P;D}
}

$ chmod u+x D-upper.sed

$ ./D-upper.sed empnametitle-with-commnet.txt
John Doe
CEO
Jason Smith
IT Manager
Raj Reddy
Sysadmin
Anand Ram
Developer
Jane Miller
Sales Manager
```

这个例子中:

- **/@/{** 这是外传循环。Sed 搜索包含@符号的任意行，如果找到，就执行后面的命令；如果没有找到，则读取下一行。为了便于说明，以第 4 行，即”@Information Technology”(这条注释跨了两行)为例，它包含一个@符合，所以后面的命令会被执行。
- **N** 从输入文件读取下一行，并追加到模式空间，以上面提到的那行数据为例，这里 N 会读取第 5 行，即”Officer@”并追加到模式空间，因此模式空间内容变为”@Informatioin Technology\nOfficer@”。
- **/@.*@/** 在模式空间中搜索匹配/@.*@/的模式,即以@开头和结尾的任何内容。当前模式空间的内容匹配这个模式，因此将继续执行后面的命令。
- **s/@.*@//;P;D** 这个替换命令把整个@Information Technology\nOfficer@”替换为空(相当于删除)。P 打印模式空间中的第一行，然后 D 删除模式空间中的第一行，然后从头开始执行命令(即不读取下一条记录，又返回到/@/处执行命令)

49.循环和分支(命令 b 和 :label 标签)

使用标签和分支命令 b，可以改变 sed 的执行流程：

- :label 定义一个标签
- b lable 执行该标签后面的命令。Sed 会跳转到该标签，然后执行后面的命令。
- 注意：命令 b 后面可以不跟任何标签，这种情况下，它会直接跳到 sed 脚本的结尾

下面例子将把 empnametitle.txt 文件中的雇员名称和职位合并到一行内，字段之间以分号：分隔，并且在管理者的名称前面加上一个星号*。

```
$ cat label.sed
#!/bin/sed -nf
h;n;H;x
s/\n/:/
/Manager/!b end
s/^/*/
:end
p
```

这个脚本中，鉴于之前的例子，你已经知道 h;n;H;x 和 s/\n/:/的作用了。下面是关于分支的操作：

- /Manager/!b end　如果行内不包含关键字”Manager”,则 跳转到’end’标签，请注意，你可以任意设置你想要的标签名称。因此，只有匹配 Manager 的雇员名称签名，才会执行 s/^/*/(在行首加上星号*)。
- :end 即是标签

给这个脚本加上可执行权限，然后执行：

```
$ chmod u+x label.sed

$ ./label.sed empnametitle.txt
John Doe:CEO
*Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
*Jane Miller:Sales Manager
```

个人觉得脚本里面的 h;n;H;x 可以用一个 N 替代，这样就不用使用保持空间了。
如果不使用标签，还可以： sed 'N;s/\n/:/;/Manager/s/^/\.*/' empnametitle.txt

50.使用命令 t 进行循环

命令 t 的作用是，如果前面的命令执行成功，那么就跳转到 t 指定的标签处，继续往下执行后续命令。否则，仍然继续正常的执行流程。

下面例子将把 empnametitle.txt 文件中的雇员名称和职位合并到一行内，字段之间以分号：分隔，并且在管理者的名称前面加上三个星号*。

提示：我们只需把前面例子中的替换命令改为 s/^/***/即可带到该目的，下面这个例子仅仅是为了解释命令 t 是如何运行的。

```
$ vi lable-t.sed
#!/bin/sed -nf
h;n;H;x
s/\n/:/
: repeat
/Manager/s/^/*/
/\*\*\*/! t repeat

p

$ chmod u+x lable-t.sed

$ ./lable-t.sed empnametitle.txt
John Doe:CEO
***Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
***Jane Miller:Sales Manager
```

这个例子中:

- 下面的代码执行循环

```
:repeat
/Manager/s/^/*/
/\*\*\*/! t repeat
```

- **/Manager/s/^/*/** 如果匹配到 Manager,在行首加上星号*
- **/***/!t repeat** 如果没有匹配到三个连续的星号*(用/***/!来表示)，并且前面一行的替换命令成功执行了，则跳转到名为 repeat 的标签处(即 t repeat)
- **:repeat** 标签

第八章：Awk 语法和基础命令

Awk 是一个维护和处理文本数据文件的强大语言。在文本数据有一定的格式，即每行数据包含多个以分界符分隔的字段时，显得尤其有用。即便是输入文件没有一定的格式，你仍然可以使用 awk 进行基本的处理。Awk 当然也可以没有输入文件，那不是必须的。简言之，AWK 是一种能处理从琐碎的小事到日常例行公事的强大语言。

学习 AWK 的难度要比学习其他任意语言的难度都小。如果你已经掌握了 C 语言，那么你会发现学习 AWK 将会是如此简单和容易。

AWK 最开始由三个人开发——A.Aho、B.W.Kernighan 和 P.Weinberger。所有 AWK 的名字来自他们名字的第一个字母。

下面是 AWK 的几个变种：

- AWK 是最原始的 AWK。
- NAWK 是 new AWK
- GAWK 是 GNU AWK。所有 linux 发行版都默认使用 GAWK，它和 AWK 以及 NAWK 完全兼容。

本书包含了原始 AWK 的所有基础功能，以及 GAWK 特有的一些高级功能。在安装了 NAWK 或 GAWK 的操作系统上，你仍然可以直接使用 awk 命令，它会根据情况调用 nawk 或 gawk。

以 linux 系统为例，你会看到 awk 是一个指向 gawk 的符号链接，所以在 linux 上执行 awk 或 gawk 将会调用 gawk：

```
$ ls -l /bin/awk /bin/gawk
lrwxrwxrwx 1 root root           4 Apr  8   2011 /bin/awk -> gawk
-rwxr-xr-x 1 root root 319336 Dec  3   2008 /bin/gawk
```

本书示例将用到下面三个文件，请先建立它们，然后用它们来运行所有示例。

employee.txt 文件

employee.txt 文件以逗号作为字段分界符，包含 5 个雇员的记录，其格式如下：

```
employee-number,employee-name,employee-title
```

建立该文件:

```
$ vi employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

items.txt 文件

items.txt 是一个以逗号作为字段分界符的文本文件，包含 5 条记录，其格式如下：

```
items-number,item-description,item-category,cost,quantity-available
```

建立该文件:

```
$ vi items.txt
101,HD Camcorder,Video,210,10
102,Refrigerator,Appliance,850,2
103,MP3 Player,Audio,270,15
104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5
```

items-sold.txt 文件

items-sold.txt 是一个以空格作为字段分界符的文本文件，包含 5 条记录。每条记录都是特定商品的编号以及当月的销售量(6 个月)。因此每条记录有 7 个字段。第一个字段是商品编号，第二个字段到第七个字段是 6 个月内每月的销售量。其格式如下：

item-number qty-sold-month1 qty-sold-month2 qty-sold-month3 qty-sold-month4 qty-sold-month5 qty-sold-month6

建立该文件:

```
$ cat items-sold.txt
101 2 10 5 8 10 12
102 0 1 4 3 0 2
103 10 6 11 20 5 13
104 2 3 4 0 6 5
105 10 2 5 7 12 6
```

51.Awk 命令语法

Awk 基础语法:

```
Awk –Fs '/pattern/ {action}' input-file
(或者)
Awk –Fs '{action}' input-file
```

上面语法中：

- -F 为字段分界符。如果不指定，默认会使用空格作为分界符。
- /pattern/和{action}需要用单引号引起来。
- /pattern/是可选的。如果不指定，awk 将处理输入文件中的所有记录。如果指定一个模式，awk 则只处理匹配指定的模式的记录。
- {action} 为 awk 命令，可以是单个命令，也可以多个命令。整个 action(包括里面的所有命令)都必须放在{ 和 }之间。
- Input-file 即为要处理的文件

下面是一个演示 awk 语法的非常简单的例子:

```
$ awk -F: '/mail/ {print $1}' /etc/passwd
mail
mailnull
```

这个例子中:

- -F 指定字段分界符为冒号，即各个字段以冒号分隔。请注意，你也可以把分界符用双引号引住，-F":"也是正确的。
- /mail/ 指定模式，awk 只会处理包含关键字 mail 的记录
- {print $1} 动作部分，该动作只包含一个 awk 命令，它打印匹配 mail 的每条记录的第 1 个字段
- /etc/passwd 即是输入文件

把 awk 命令放入单独的文件中(awk 脚本)

当需要执行很多 awk 命令时，可以把/pattern/{action}这一部分放到单独的文件中，然后调用它:

```
awk –Fs –f myscript.awk    input-file
```

myscript.awk 可以使用任意扩展名(或者不用扩展名)。但是加上扩展名.awk 便于维护，也可以在这个文件中设置字段分界符(后面详述)，然后调用:

```
awk –f myscript.awk    input-file
```

52.Awk 程序结构(BEGIN,body,END)区域

典型的 awk 程序包含下面三个区域:

1. **BEGIN 区域**

   Begin 区域的语法:

   ```
   BEGIN { awk-commands }
   ```

BEGIN 区域的命令只最开始、在 awk 执行 body 区域命令之前执行一次。

- BEGIN 区域很适合用来打印报文头部信息，以及用来初始化变量。
- BEGIN 区域可以有一个或多个 awk 命令
- 关键字 BEGIN 必须要用大写
- BEGIN 区域是可选的

2. **body 区域**

   body 区域的语法:

   ```
   /pattern/ {action}
   ```

   body 区域的命令每次从输入文件读取一行就会执行一次

   - 如果输入文件有 10 行，那 body 区域的命令就会执行 10 次(每行执行一次)
   - Body 区域没有用任何关键字表示，只有用正则模式和命令。

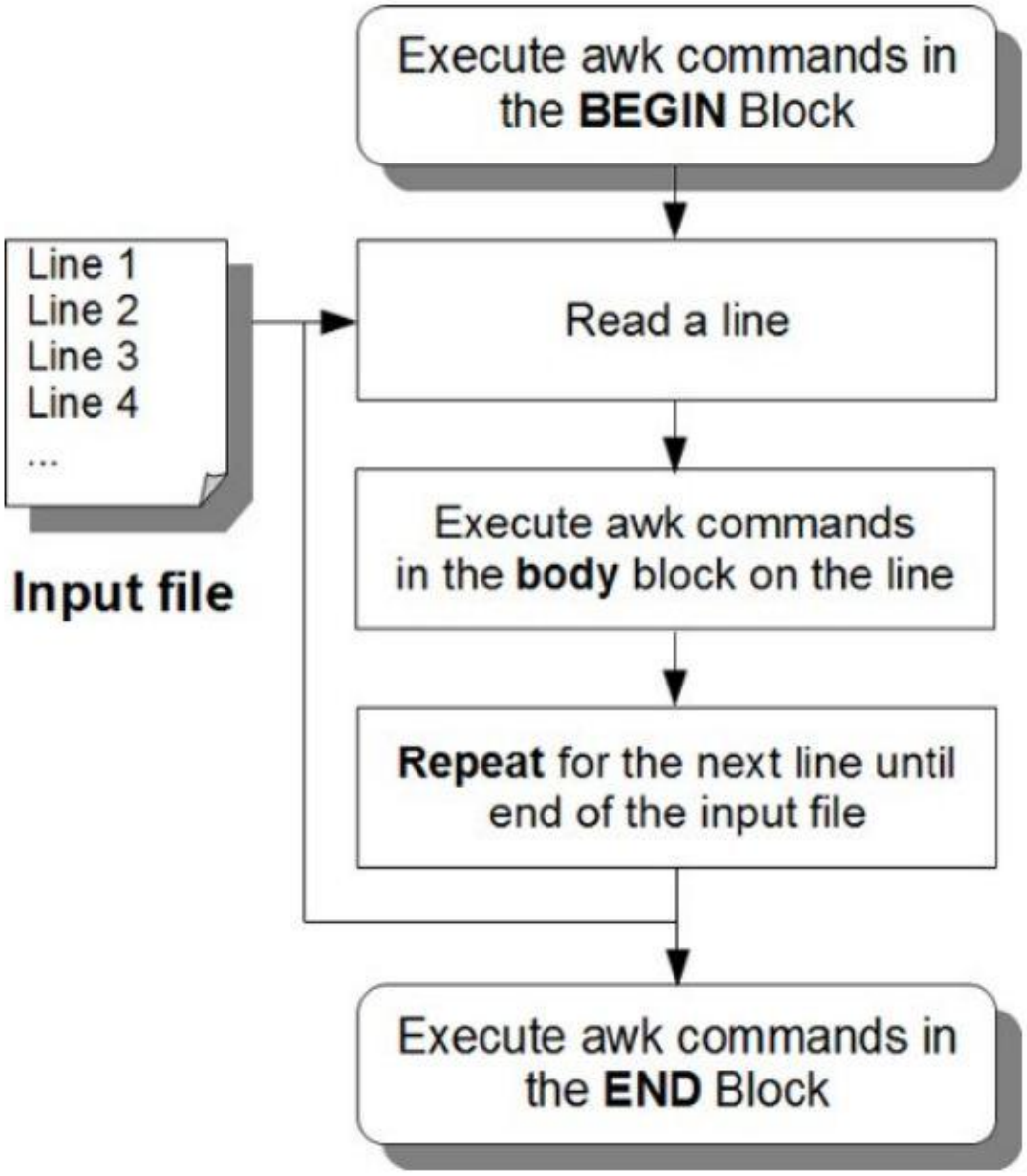
3. **END block**

   END 区域的语法:

   ```
   END { awk-commands }
   ```

   END 区域在 awk 执行完所有操作后执行，并且只执行一次。

   - END 区域很适合打印报文结尾信息，以及做一些清理动作
   - END 区域可以有一个或多个 awk 命令
   - 关键字 END 必须要用大写
   - END 区域是可选的



Awk 的执行流程

下面的例子包含上上述的三个区域：

```
$ awk 'BEGIN { FS=":";print "----header----" } \
/mail/ {print $1} \
END {print "----footer----"}' /etc/passwd
----header----
mail
mailnull
----footer----
```

提示：如果命令很长，即可以放到单行执行，也可以用\折成多行执行。上面的例子用\把命令折成了 3 行。

在这个例子中：

- BEGIN { FS=":";print "----header----" } 为 BEGIN 区域，它设置了字段分界符变量 FS(下文详述)的值，然后打印报文头部信息。这个区域仅在 body 区域循环之前执行一次。
- /mail/{print $1}是 body 区域，包含一个正则模式和一个动作，即在输入文件中搜索包含关键字 mail 的行，并打印第一个字段。
- END {print "----footer----" }是 END 区域，打印报文尾部信息。
- /etc/passwd 是输入文件，每行记录都会执行一次 body 区域里的动作。

上面的例子中，除了可以在命令行上执行外，还可以通过脚本执行。

首先建立下面的文件 myscript.awk,它包含了 begin,body 和 end ：

```
$vi myscript.awk
BEGIN {
FS=":"
print "---header---"
}
/mail/ {
print $1
}
END {
print "---footer---"
}
```

然后，如下所示，在/etc/passwd 上执行 myscript.awk 文件：

```
$awk -f myscript.awk /etc/passwd
---header---
mail
---footer---
```

请注意，awk 脚本中，注释以#开头。如果要编写复杂的 awk 脚本，最后接受下面的建议：在*awk 文件中写上足够多的注释，这样以后再次使用该脚本时，更易于读懂。

下面是随机列出的一些简单的例子，用例演示 awk 各个区域的不同组合方式：

只有 body 区域:

```
awk –F: '{ print $1 }' /etc/passwd
```

同时具有 begin,body 和 end 区域:

```
awk –F: 'BEGIN{printf "username\n-------\n"}\
{ print $1 }\
END {print "----------" }' /etc/passwd
```

只有 begin 和 body 区域:

```
Awk –F: 'BEGIN {print "UID"} {print $3}' /etc/passwd
```

关于使用 BEGIN 区域的提示:

只使用 BEGIN 区域在 awk 中是符合语法的。在没有使用 body 区域时，不需要指定输入文件，因为 body 区域只在输入文件上执行。所以在执行和输入文件无关的工作时，可以只使用 BEGIN 区域。下面的不少例子中，只包含 BEGIN 区域，用来说明 awk 的不同部分是如何执行的。你可以因地制宜地使用下面的例子。

只包含 BEGIN 的简单示例:

```
$awk ‘BEGIN { print “Hello,World!” }’
Hello World!
```

多个输入文件:

注意，可以为 awk 指定多个输入文件。如果指定了两个文件，那么 body 区域会首先在第一个文件的所有行上执行，然后在第二个文件的所有行上执行。

多个输入文件示例:

```
$awk 'BEING { FS=":";print "---header---" }\
/mail/ {print $1}\
END { print "---footer---"}' /etc/passwd /etc/group
mail:x:8:12:Mailer
mail:x:12:
maildrop:!:59:
---footer---
```

注意，即是指定了多个文件，BEGIN 和 END 区域，仍然只会执行一次。

53.打印命令

默认情况下，awk 的打印命令 print(不带任何参数)会打印整行数据。下面的例子等价于”cat employee.txt”命令.

```
$awk '{print}' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

也可以通过传递变量”$字段序号”作为 print 的参数来指定要打印的字段。我们猜想例子应该只打印雇员名称(第 2 个字段)

```
$awk '{print $2}' employee.txt
Doe,CEO
Smith,IT
Reddy,Sysadmin
Ram,Developer
Miller,Sales
```

等等，这个输出好像和预期不符。它打印了从姓氏开始直到记录结尾的所有内容。这是因为 awk 默认的字段分隔符是空格，awk 准确地执行了我们要求的动作，它以空格作为分隔符，打印第 2 个字段。当使用默认的空格作为字段分隔符时，101,Johne 变成了第一条记录的第一个字段，Doe,CEO 变成了第二个字段。因此上面例子中，awk 把 Doe,CEO 作为第二个字段打印出来了。

要解决这个文件，应该使用-F 选项为 awk 指定一个逗号”,”最为字段分隔符。

```
$awk -F ',' '{print $2}' employee.txt
John Doe
```

```
Jason Smith
Raj Reddy
Anand Ram
Jane Miller
```
当字段分隔符是单个字符时，下面的所有写法都是争取的，即可以把它放在单引号或双引号中，或者不使用引号：
```
awk –F ‘,’ ‘{print $2}’ employee.txt
awk –F “,” ‘{print $2}’ employee.txt
awk –F , ‘{print $2}’ employee.txt
```
提示:也可使用 FS 变量来达到同样的目的。后面会介绍这个 awk 内置变量的用法。

一个简单的例子，用来输出雇员姓名，职位，同时附带 header 和 footer 信息:
```
$awk 'BEGIN{FS=",";print "---------\nName    Title\n-----------\n";}\
{print $2,"\t",$3}\
END {print "------------------"}' employee.txt
---------
Name    Title
-----------
John Doe                CEO
Jason Smith             IT Manager
Raj Reddy               Sysadmin
Anand Ram               Developer
Jane Miller             Sale
```

这个例子中，输出结果各字段并没有很好地对齐，后面章节将会介绍如何处理这个问题。这个例子还展示了如何使用 BEGIN 来打印 header 以及如何使用 END 来打印 footer.

请注意，$0 代表整条记录。下面两个命令是等价的，都打印 employee.txt 的所有行:
```
awk ‘{print}’ employee.txt
awk ‘{print $0}’ employee.txt
```

54.模式匹配

你可以只在匹配特殊模式的行数执行 awk 命令。

下面的例子只打印管理者的姓名和职位：
```
$awk -F ',' '/Manager/ {print $2,$3}' employee.txt
Jason Smith IT Manager
Jane Miller Sales Manager
```

下面的例子只打印雇员 id 为 102 的雇员的信息：
```
$awk -F ',' '/^102/{print "Emp id 102 is",$2}' \
> employee.txt
```

```
Emp id 102 is Jason Smith
```

第九章：awk 内置变量

55. FS –输入字段分隔符

awk 默认的字段分隔符是空格，如果你的输入文件中不是一个空格作为字段分隔符，你已经知道可以在 awk 命令行上使用-F 选项来指定它:

```
awk –F ‘,’ ‘{print $2,$3}’ employee.txt
```

同样的事情，也可以使用 awk 内置变量 FS 来完成。FS 只能在 BEGIN 区域中使用。

```
awk ‘BEGIN {FS=”,”} {print $2,$3}’ employee.txt
```

BEGIN 区域可以包含多个命令，下面的例子中，BEGIN 区域包含一个 FS 和一个 print 命令.BEGIN 区域的多个命令之间，要用分号分隔。

```
$awk 'BEGIN { FS=",";\
> print "--------------------------\nName\tTitle\n----------------------"}\
> {print $2,"\t",$3;}\
> END {print "----------------------------------------"}' employee.txt
--------------------------
Name        Title
----------------------
John Doe                CEO
Jason Smith             IT Manager
Raj Reddy               Sysadmin
Anand Ram               Developer
Jane Miller             Sales Manager
----------------------------------------
```

注意：默认的字段分隔符不仅仅是单个空格字符，它实际上是一个或多个空白字符。

下面的 employee-multiple-fs.txt 文件，每行记录都包含 3 个不同的字段分隔符:

- ，雇员 id 后面的分隔符是逗号
- ：雇员姓名后面的分隔符是分号
- % 雇员职位后面的分隔符是百分号

创建文件:

```
$vi employee-multiple-fs.txt
101,John Doe:CEO%10000
102,Jason Smith:IT Manager%5000
103,Raj Reddy:Sysadmin%4500
104,Anand Ram:Developer%4500
105,Jane Miller:Sales Manager%3000
```

当遇到一个包含多个字段分隔符的文件时，不必担心，FS 可以搞定。你可以使用正则表达式来指定多个字段分隔符，如 FS = “[,:%]” 指定字段分隔符可以是逗号 ,或者分号 : 或者百分号 %。
因此，下面的例子将打印 employee-multiple-fs.txt 文件中雇员名称和职位.

```
$awk 'BEGIN {FS="[,:%]"}{print $2,$3}' employee-multiple-fs.txt
John Doe CEO
Jason Smith IT Manager
Raj Reddy Sysadmin
Anand Ram Developer
Jane Miller Sales Manager
```

56. OFS – 输出字段分隔符

FS 是输入字段分隔符，OFS 是输出字段分隔符。OFS 会被打印在输出行的连续的字段之间。默认情况下，awk 在输出字段中间以空格分开。

请注意，我们没有指定 IFS 作为输入字段分隔符，我们从简地使用 FS。

下面的例子打印雇员姓名和薪水，并以空格分开。当你使用单个 print 语句打印多个以逗号分开(如下面的例子所示)的变量是，每个变量之间会以空格分开。

```
$awk -F ',' '{print $2,$3}' employee.txt
John Doe CEO
Jason Smith IT Manager
Raj Reddy Sysadmin
Anand Ram Developer
Jane Miller Sales Manager
```

如果你尝试认为地在输出字段之间加上冒号，会有如下输出。请注意在冒号前后均有一个多余的空格，这是因为 awk 仍然以空格作为输出字段分隔符。

下面的 print 语句实际上会打印 3 个值(以逗号分割)——$2,:和$4.因为如你所知，当使用单个 print 语句打印多个变量时，输出内容会包含多余的空格。

```
$awk -F ',' '{print $2,":",$3}' employee.txt
John Doe : CEO
Jason Smith : IT Manager
Raj Reddy : Sysadmin
Anand Ram : Developer
Jane Miller : Sales Manager
```

正确的方法是使用 awk 内置变量 OFS(输出字段分隔符)，如下面了示例。请注意这个例子中分号前后没有多余的空格，因为 OFS 使用冒号取代了 awk 默认的分隔符。
下面的 print 语句打印两个变量($2 和$4)，使用都会分隔，然而输出结果却是以分号分隔(而不是空格),因为 OFS 被设置成了分号.

```
$awk -F ',' 'BEGIN {OFS=":"} {print $2,$3}' employee.txt
```

```
John Doe:CEO
Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
Jane Miller:Sales Manager
```

同时请注意在 print 语句中使用和不使用逗号的细微差别(打印多个变量时).当在 print 语句中指定了逗号，awk 会使用 OFS。如下面的例子所示，默认的 OFS 会被使用，所以你会看到输出值之间的空格。

```
$awk 'BEGIN { print "test1","test2"}'
test1 test2
```

不使用逗号是，awk 将不会使用 OFS，其输出变量之间没有任何空格。

```
$awk 'BEGIN { print "test1" "test2"}'
test1test2
```

57. RS – 记录分隔符

假定有下面一个文件，雇员的 id 和名称都在单一的一行内.

```
$vi employee-one-line.txt
101,John Doe:102,Jason Smith:103,Raj Reddy:104,Anand Ram:105,Jane, Miller
```

这个文件中，每条记录包含两个字段(empid 和 name),并且每条记录以分红分隔(取代了换行符).每条记录中的单独的字段(empid 和 name)以逗号分隔。

Awk 默认的记录分隔符是换行符。如果要尝试只打印雇员姓名，下面的例子无法完成：

```
$awk -F, '{print $2}' employee-one-line.txt
John Doe:102
```

这个例子把 employee-one-line.txt 的内容作为单独一行，把逗号作为字段分隔符，所以，它打印”John Doe:102 作为第二个字段。

如果要把文件内容作为 5 行记录来处理(而不是单独的一行)，并且打印每条记录中雇员的姓名，就必须把记录分隔符指定为分号，如下所示：

```
$awk -F, 'BEGIN { RS=":" } {print $2}' employee-one-line.txt
John Doe
Jason Smith
Raj Reddy
Anand Ram
Jane Miller
```

假设有下面的文件，记录之间用-分隔，独占一行。所有的字段都占单独的一行。

```
$vi employee-change-fs-ofs.txt
101
John Doe
CEO
```

```
-
102
Jason Smith
IT Manager
-
103
Raj Reddy
Sysadmin
-
104
Anand Ram
Developer
-
105
Jane Miller
Sales Manager
```

上面例子中，字段分隔符 FS 是换行符，记录分隔符 RS 是”-“和换行符。所以如果要打印雇员名称和职位，需要：

```
$awk 'BEGIN{RS="-\n";FS="\n";OFS=":"}{print $2,$3}' employee-change-fs-ofs.txt
John Doe:CEO
Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
Jane Miller:Sales Manager
```

58. ORS – 输出记录分隔符

RS 是输入字段分隔符，ORS 是输出字段分隔符。请注意，我们没有指定 IFS 作为输入字段分隔符，我们从简地使用 FS。

下面的例子在每个输出行后面追加”---------“.awk 默认使用换行符”\n”作为 ORS，这个例子中，我们使用”\n---\”作为 ORS。

```
$ awk 'BEGIN {FS=",";ORS="\n---\n"} {print $2,$3}' employee.txt
John Doe CEO
---
Jason Smith IT Manager
---
Raj Reddy Sysadmin
---
Anand Ram Developer
---
Jane Miller Sales Manager
```

下面的例子从 employee.txt 获取输入，把每个字段打印成单独一行，每条记录用”---“分隔

```
$ awk 'BEGIN { FS=",";OFS="\n";ORS="\n---\n"}{print $1,$2,$3}' employee.txt
101
John Doe
CEO
---
102
Jason Smith
IT Manager
---
103
Raj Reddy
Sysadmin
---
104
Anand Ram
Developer
---
105
Jane Miller
Sales Manager
---
```

59. NR – 记录序号

NR 非常有用，在循环内部标识记录序号。用于 END 区域时，代表输入文件的总记录数。

尽管你会认为 NR 代表”记录的数量(Number of Records)”，但它跟确切的叫法是”记录的序号(Number of the Record)”,也就是当前记录在所有记录中的行号。

下面的例子演示了 NR 在 block 和 END 区域是怎么运行的：

```
$ awk 'BEGIN {FS=","} \
> {print "Emp Id of record number",NR,"is",$1;}\
> END {print "Total number of records:",NR}' employee.txt
Emp Id of record number 1 is 101
Emp Id of record number 2 is 102
Emp Id of record number 3 is 103
Emp Id of record number 4 is 104
Emp Id of record number 5 is 105
Total number of records: 5
```

60. FILENAME – 当前处理的文件名

当使用 awk 处理多个输入文件时，FILENAME 就显得很有用，它代表 awk 当前正在处理的文件。

```
$ awk '{ print FILENAME }' \
> employee.txt employee-multiple-fs.txt
employee.txt
employee.txt
employee.txt
employee.txt
employee.txt
employee-multiple-fs.txt
employee-multiple-fs.txt
employee-multiple-fs.txt
employee-multiple-fs.txt
employee-multiple-fs.txt
```

如果 awk 从标准输入获取内容，FILENAME 的值将会是”-“，下面的例中，我们不提供任何输入文件，所以你应该手动输入内容以代替标准输入。

下面的例子中，我们只输入一个人名”John Doe”作为第一条记录,然后 awk 打印出该人的姓氏。这种情况下，必须按 Ctrl-C 才能停止标准输入。

```
$ awk '{print "Last name:",$2;\
> print "Filename:",FILENAME}'
John Deo
Last name: Deo
Filename: -
```

上面这个例子在使用管道向 awk 传递数据时，同样适用。如下所示，打印出来的 FILENAME 仍然是”-“

```
$ echo "Johe Doe" | awk '{print "Last name:",$2;\
> print "Filename:",FILENAME}'
Last name: Doe
Filename: -
```

注意:在 BEGIN 区域内，FILENAME 的值是空，因为 BEGIN 区域只针对 awk 本身，而不处理任何文件。

61. FNR – 文件中的 NR

我们已经知道 NR 是”记录条数”(或者叫”记录的序号”),代表 awk 当前处理的记录的行号。

在给 awk 传递了两个输入文件时 NR 会是什么？ NR 会在多个文件中持续增加，当处理到第二个文件时，NR 不会被重置为 1，而是在前一个文件的 NR 基础上继续增加。

下面的例子中，第一个文件有 5 条记录，第二个文件也有 5 条记录。如下所示，当 body 区域的循环处理到第二个文件时，NR 从 6 开始递增(而不是 1).最后在 END 区域，NR 返回两个文件的总记录条数。

```
$ awk 'BEGIN {FS=","}\
> {print FILENAME ": record number",NR,"is",$1;} \
> END {print "Total number of records:",NR}' \
> employee.txt employee-multiple-fs.txt
employee.txt: record number 1 is 101
employee.txt: record number 2 is 102
employee.txt: record number 3 is 103
employee.txt: record number 4 is 104
employee.txt: record number 5 is 105
employee-multiple-fs.txt: record number 6 is 101
employee-multiple-fs.txt: record number 7 is 102
employee-multiple-fs.txt: record number 8 is 103
employee-multiple-fs.txt: record number 9 is 104
employee-multiple-fs.txt: record number 10 is 105
Total number of records: 10
```

下面例子同时打印 NR 和 FNR:

```
$ awk -f fnr.awk employee.txt employee-multiple-fs.txt
FILENAME=employee.txt NR=1 FNR=1
FILENAME=employee.txt NR=2 FNR=2
FILENAME=employee.txt NR=3 FNR=3
FILENAME=employee.txt NR=4 FNR=4
FILENAME=employee.txt NR=5 FNR=5
FILENAME=employee-multiple-fs.txt NR=6 FNR=1
FILENAME=employee-multiple-fs.txt NR=7 FNR=2
FILENAME=employee-multiple-fs.txt NR=8 FNR=3
FILENAME=employee-multiple-fs.txt NR=9 FNR=4
FILENAME=employee-multiple-fs.txt NR=10 FNR=5
END Block:NR=10 FNR=5
```

第十章：awk 变量的操作符

62. 变量

Awk 变量以字母开头，后续字符可以是数字、字母、或下划线。关键字不能用作 awk 变量。

不像其他编程语言，awk 变量可以直接使用而不需事先声明。如果要初始化变量，最好在 BEGIN 区域内作，它只会执行一次。

Awk 中没有数据类型的概念，一个 awk 变量是 number 还是 string 取决于该变量所处的上下文。

employee-sal.txt 示例文件
emploeyee-sal.txt 以逗号作为字段分隔符，包含五个雇员的信息，格式如下:

```
employee-numer,employee-name,employee-title,salary
```

创建下面文件:

```
$ vi employee-sal.txt
101,John Doe,CEO,10000
102,Jason Smith,IT Manager,5000
103,Raj Reddy,Sysadmin,4500
104,Anand Ram,Developer,4500
105,Jane Miller,Sales Manager,3000
```

下面的例子将教你如何在 awk 中创建和使用自己的变量，该例中，”total”便是用户建立的用来存储公司所有雇员工资总和的变量。

```
$ vi total-company-salary.awk
BEGIN {
        FS=",";
        total=0;
}
{
        print $2 "'s slary is: " $4;
        total=total+$4
}
END {
        print "---\nTotal company salary =$"total;
}
$ awk -f total-company-salary.awk employee-sal.txt
John Doe's slary is: 10000
Jason Smith's slary is: 5000
Raj Reddy's slary is: 4500
```

```
Anand Ram's slary is: 4500
Jane Miller's slary is: 3000
---
Total company salary =$27000
```

63. 一元操作符

只接受单个操作数的操作符叫做一元操作符。

操作符(Operator)	描述(Description)
+	取正(返回数字本身)
-	取反
++	自增
--	自减

下面的例子使用取反操作:

```
$ awk -F, '{print -$4}' employee-sal.txt
-10000
-5000
-4500
-4500
-3000
```

下面的例子演示取正、取反操作符对文件中存放的复数的作用：

```
$ vi negative.txt
-1
-2
-3

$ awk '{print +$1}' negative.txt
-1
-2
-3

$ awk '{print -$1}' negative.txt
1
2
3
```

自增和自减操作

自增和自减改变变量的值，它可以在使用变量“之前”或“之后”改变变量的值。在表达式中，使用的可能是改变前的值(post)或改变后的值(pre).

使用改变后的变量值(pre)即是在变量前面加上++(或--)，首先把变量的值加 1(或减 1)，然后把改变后的值应用到表达式的其它操作中。

使用改变前的变量值(post)即是在变量后面加上++(或--)，首先把变量值应用到表达式中进行计算，然后把变量的值加 1(或减 1)。

Pre 自增示例:

```
$ awk -F, '{print ++$4}' employee-sal.txt
10001
5001
4501
4501
3001
```

Pre 自减示例:

```
$ awk -F, '{print --$4}' employee-sal.txt
9999
4999
4499
4499
2999
```

Post 自增示例:

(因为++ 在 print 语句中，所以变量的原始值被打印)

```
$ awk -F, '{print $4++}' employee-sal.txt
10000
5000
4500
4500
3000
```

Post 自减示例:

(因为++是单独语句，所以自增后的值被打印)

```
$ awk -F, '{$4++;print $4}' employee-sal.txt
10001
5001
4501
4501
3001
```

Post 自减示例:

(因为—在 print 语句中，所以变量原始值被打印)

```
$ awk -F, '{print $4--}' employee-sal.txt
10000
```

5000
4500
4500
3000

Post 自减示例:
(因为—在单独语句中，所以自减后的值被打印)
$ awk -F, '{$4--;print $4}' employee-sal.txt
9999
4999
4499
4499
2999

下面是一个有用的例子，显示所有登录到 shell 的用户数，即哪些用户可以登录 shell 并获得命令提示符。

- 使用算后自增运算符(尽管变量值只到 END 区域才打印出来，算前自增仍会产生同样的结果)
- 脚本的 body 区域包含一个模式匹配，因此只有最后一个字段匹配模式/bin/bash 时，body 的代码才会执行
- 提示:正则表达式应该包含在//之间，但如果正则表达式中的/必须转移，以避免被解析为正则表达式的结尾
- 当有匹配到模式的行时，变量 n 的值增加 1，最终的值在 END 区域打印出来。

打印所有可登陆 shell 的用户总数:
$ awk -F':' '$NF ~ /\/bin\/bash/ { n++ }; END { print n }' /etc/passwd
2

64. 算术操作符

需要两个操作数的操作符，成为二元操作符。Awk 中有多种基本二元操作符(如算术操作符、字符串操作符、赋值操作符，等等)。

下面是算术运算操作符:

操作符(Opertator)	描述(Description)
+	加
-	减
*	乘
/	除
%	取模(取余)

下面的例子展示+,-,*,/的用法.

下面例子完成两个事情:

1. 将每件单独的商品价格减少 20%
2. 将每件单独的商品的数量减少 1

创建并运行 awk 算术运算脚本:

```
$ vi arithmetic.awk
BEGIN {
        FS=",";
        OFS=",";
        item_discount=0;
}

{
        item_discount=$4*20/100;
        print $1,$2,$3,$4-item_discount,$5-1;
}

$ awk -f arithmetic.awk items.txt
101,HD Camcorder,Video,168,9
102,Refrigerator,appliance,680,1
103,MP3 Player,Audio,216,14
104,Tennis Racket,Sports,152,19
105,Laser Printer,Office,380,4
```

下面的例子只打印偶数行。打印前会检查行号是否能被 2 整除，如果整除，则执行默认的操作(打印整行).

取模运算演示:

```
$ awk 'NR % 2 == 0' items.txt
102,Refrigerator,appliance,850,2
104,Tennis Racket,Sports,190,20
```

65. 字符串操作符

(空格)是连接字符串的操作符。

下面例子中，有三处使用了字符串连接。在语句”string3=string1 string2”中，string3 包含了 string1 和 string2 连接后的内容。每个 print 语句都把一个静态字符串和 awk 变量做了连接。

提示:这个操作符解释了为什么在打印多个变量时，如果要使用 OFS 分隔每个字段，就需要在 print 语句中用逗号分隔每个变量。如果没有使用逗号分隔，那么会把所有值连接成一个字符串。

```
$ cat string.awk
BEGIN {
        FS=",";
        OFS=",";
        string1="Audio";
        string2="Video";
        numberstring="100";
        string3=string1 string2;
        print "Concatenate string is:" string3;
        numberstring=numberstring+1;
        print "String to number:" numberstring;
}

$ awk -f string.awk items.txt
Concatenate string is:AudioVideo
String to number:101
```

66. 赋值操作符

与其他大部分编程语言一样，awk 使用 = 作为赋值操作符。和 C 语言一样，awk 支持赋值的缩写方式。

操作符(Opertator)	描述(Descriptioin)
=	赋值
+=	加法赋值的缩写
-=	减法赋值的缩写
*=	乘法赋值的缩写
/=	除法赋值的缩写
%/	取模赋值的缩写

下面的例子演示如何使用赋值:

```
$ cat assignment.awk
BEGIN {
        FS=",";
        OFS=",";
        total1 = total2 = total3 = total4 = total5 = 10;
        total1 += 5; print total1;
        total2 -= 5; print total2;
        total3 *= 5; print total3;
        total4 /= 5; print total4;
        total5 %= 5; print total5;
}
```

```
$ awk -f assignment.awk
15
5
50
2
0
```

下面的例子使用加法赋值的缩写形式.

显示所有商品的清单：

```
$ awk -F ',' 'BEGIN { total=0 } { total += $5 } END {print "Total Quantity: " total }' items.txt
Total Quantity: 52
```

下面的例子统计输入文件中所有的字段数。Awk 读取每一行，并把字段数量增加到变量 total 中。然后在 END 区域打印该变量。

```
$ awk -F ',' 'BEGIN { total = 0 } { total += NF } END { print total }' items.txt
25
```

67. 比较操作符

Awk 支持下面标准比较操作符.

操作符(Operator)	描述(Description)
>	大于
>=	大于等于
<	小于
<=	小于等于
==	等于
!=	不等于
&&	且(and)
||	或(or)

提示：下面的例子，如果不指定操作，awk 会打印符合条件的整条记录。

打印数量小于等于临界值 5 的商品信息：

```
$ awk -F ',' '$5 <= 5' items.txt
102,Refrigerator,appliance,850,2
105,Laser Printer,Office,475,5
```

打印编号为 103 的商品信息:

```
$ awk -F "," '$1 == 103' items.txt
103,MP3 Player,Audio,270,15
```

提示:不要把==(等于)和=(赋值)搞混了。

打印编号为 103 的商品的描述信息:

```
$ awk -F "," '$1 == 103 { print $2}' items.txt
MP3 Player
```

打印除 Video 以外的所有商品:

```
$ awk -F "," '$3 != "Video"' items.txt
102,Refrigerator,appliance,850,2
103,MP3 Player,Audio,270,15
104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5
```

和上面相同，但只打印商品描述信息：

```
$ awk -F "," '$3 != "Video" { print $2}' items.txt
Refrigerator
MP3 Player
Tennis Racket
Laser Printer
```

使用&&比较两个条件，打印价钱低于 900 并且数量小于等于临界值 5 的商品信息：

```
$ awk -F "," '$4 < 900 && $5 <= 5' items.txt
102,Refrigerator,appliance,850,2
105,Laser Printer,Office,475,5
```

和上面相同，但只打印商品描述信息：

```
$ awk -F "," '$4 < 900 && $5 <= 5 {print $2}' items.txt
Refrigerator
Laser Printer
```

使用||比较两个条件，打印价钱低于 900 或者数量小于等于临界值 5 的商品信息：

```
$ awk -F "," '$4 < 900 || $5 <= 5' items.txt
101,HD Camcorder,Video,210,10
102,Refrigerator,appliance,850,2
103,MP3 Player,Audio,270,15
104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5
```

和上面相同，但只打印商品描述信息：

```
$ awk -F "," '$4 < 900 || $5 <= 5 {print $2}' items.txt
HD Camcorder
Refrigerator
MP3 Player
Tennis Racket
Laser Printer
```

下面例子使用>条件，打印/etc/password 中最大的 UID(以及其所在的整行)。Awk 把最大的 UID(第 3 个字段)放在变量 maxuid 中，并且把包含最大 UID 的行复制到变量 maxline 中。循环执行完后，打印最大的 UID 和其所在的行。

```
$ awk -F ':' '$3 > maxuid { maxuid = $3; maxline = $0 } END { print maxuid,maxline }' /etc/passwd
65534 nobody:x:65534:65533:nobody:/var/lib/nobody:/bin/bash
```

打印/etc/passwd 中 UID 和 GROUP ID 相同的用户信息

```
$ awk -F ':' '$3 == $4' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

打印/etc/passwd 中 UID >= 100 并且用户的 shell 是/bin/sh 的用户:

```
$ awk -F ':' '$3 >= 100 && $NF ~ /\/bin\/sh/ ' /etc/passwd
mygame:x:2300:0::/opt/game:/bin/sh
```

打印/etc/passwd 中没有注释信息(第 5 个字段)的用户:

```
mkey3g@mkey:/opt/mkey3g/ec> awk -F ':' '$5 == ""' /etc/passwd
mfs:x:2001:1000::/home/mfs:/bin/nologin
```

68. 正则表达式操作符

操作符(Opertator)	描述(Descriptioin)
~	匹配
!~	不匹配

使用==时，awk 检查精确匹配。 下面的例子不会打印任何信息，因为 items.txt 中，没有任何一条记录的第二个字段精确匹配关键字”Tennis”,”Tennis Racket”不是精确匹配。

打印第二个字段为”Tennis”的记录:

```
$ awk -F "," '$2 == "Tennis"' items.txt
```

当使用~时，awk 执行模糊匹配，检索”包含”关键字的记录.

打印第二个字段包含”Tennis”的行:

```
$ awk -F "," '$2 ~ "Tennis"' items.txt
104,Tennis Racket,Sports,190,20
```

!~ 对应 ~,即不匹配.

打印第二个字段不包含”Tennis”的行:

```
$ awk -F "," '$2 !~ "Tennis"' items.txt
101,HD Camcorder,Video,210,10
102,Refrigerator,appliance,850,2
103,MP3 Player,Audio,270,15
105,Laser Printer,Office,475,5
```

下面的例子打印 shell 为/bin/bash 的用户的总数，如果最后一个字段包含”/bin/bash”,则变量 n 增加 1

```
$ awk -F ':' '$NF ~ /\/bin\/bash/ { n++ } END { print n }' /etc/passwd
13
```

第十一章: awk 分支和循环

Awk 支持条件判断，控制程序流程。Awk 的大部分条件判断语句很像 C 语言的语法。

Awk 支持下面三种 if 语句.

- 单个 if 语句
- If-else 语句
- 多级 If-elseif 语句

69. if 结构

单个 if 语句检测条件，如果条件为真，执行相关的语句。

单条语句

语法:

```
if(conditional-expression )
        action
```

- if 是关键字
- conditional-expression 是要检测的条件表达式
- action 是要执行的语句

多条语句

如果要执行多条语句，需要把他们放在{ } 中，每个语句之间必须用分号或换行符分开,如下所示.

语法:

```
if (conditional-expression)
{
        action1;
        action2;
}
```

如果条件为真，{ }中的语句会依次执行。当所有语句执行完后，awk 会继续执行后面的语句。

打印数量小于等于 5 的所有商品:

```
$ awk -F ',' '{ if ($5 <= 5) print "Only",$5,"qty of",$2 "is available" }' items.txt
Only 2 qty of Refrigeratoris available
Only 5 qty of Laser Printeris available
```

使用多个条件，可以打印价钱在 500 至 100，并且总数不超过 5 的商品

```
$ awk -F ',' '{ if (($4 >= 500 && $4<= 1000) && ($5 <= 5)) print "Only",$5,"qty of",$2,"is available" }' items.txt
Only 2 qty of Refrigerator is available
```

70. if else 结构

在 if else 结构中，还可以指定判断条件为 false 时要执行的语句。下面的语法中，如果条件为 true，那么执行 action1,如果条件为 false,则执行 action2

语法:

```
if (conditional-expression)
        action1
else
        action2
```

此外，awk 还有个条件操作符(? :)，和 C 语言的三元操作符等价。

和 if-else 结构相同，如果 codintional-expresion 是 true,执行 action1,否则执行 action2

三元操作符:

```
codintional-expression ? action1 : action2 ;
```

如果商品数量不大于 5，打印”Buy More”,否则打印商品数量:

```
$ cat if-else.awk
BEGIN {
        FS=",";
}
{
        if( $5 <= 5)
                print "Buy More: Order",$2,"immediately!"
        else
                print "Shell More: Give discount on",$2,"immediately!"
}

$ awk -f if-else.awk items.txt
Shell More: Give discount on HD Camcorder immediately!
Buy More: Order Refrigerator immediately!
Shell More: Give discount on MP3 Player immediately!
Shell More: Give discount on Tennis Racket immediately!
Buy More: Order Laser Printer immediately!
```

下面的例子，使用三元操作符，把 items.txt 文件中的每两行都以逗号分隔合并起来。

```
$ awk 'ORS=NR%2?",":"\n"' items.txt
101,HD Camcorder,Video,210,10,102,Refrigerator,appliance,850,2
103,MP3 Player,Audio,270,15,104,Tennis Racket,Sports,190,20
```
(复杂的就是 awk '{if(NR%2==0) ORS="\n"; else ORS=",";print}' items.txt)

71. while 循环

Awk 循环用例执行一系列需要重复执行的动作，只要循环条件为 true，就一直保持循环。和 C 语言类似，awk 支持多种循环结构。

首先是 while 循环结构.
语法:

```
while (codition)
    Actions
```

- while 是 awk 的关键字
- condition 是条件表达式
- actions 是循环体，如果有多条语句，必须放在{ }中

while 首先检查 condtion,如果是 true，执行 actions,执行完后，再次检查 condition,如果是 true，再次执行 actions,直到 condition 为 false 时，退出循环。

注意，如果第一次执行前 condition 返回 false，那么所有 actions 都不会被执行。

下面的例子中，BEGIN 区域中的语句会先于其他 awk 语句执行。While 循环将 50 个'x'追加到变量 string 中。每次循环都检查变量 count,如果其小于 50，则执行追加操作。因此循环体会执行 50 次，之后，变量 string 的值被打印出来。

```
$ awk 'BEGIN { while(count++<50) string=string "x"; print string}'
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

下面的例子计算 items-sold.txt 文件中，每件商品出售的总数。

对于每条记录，需要把从第 2 至第 7 个字段累加起来(第一个字段是编号，因此不用累加)。所以，当循环条件从第 2 个字段(因为 while 之前设置了 i=2),开始，检查是否到达最后一个字段(i<=NF)。NF 代表每条记录的字段数。

```
$ cat while.awk
{
        i=2; total=0;
        while (i <= NF ){
                total = total + $i;
                i++;
        }
        print "Item",$1,":",total,"quantities sold";
}
```

```
$ awk -f while.awk items-sold.txt
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

72. do-while 循环

While 循环是一种进入控制判断的循环结构，因为在进入循环体前执行判断。而 do-while 循环是一种退出控制循环，在退出循环前执行判断。do-while 循环至少会执行一次，如果条件为 true，它将一直执行下去。

语法:

```
do
action
while(condition)
```

下面的例子中，print 语句仅会执行一次，因为我们已经确认判断条件为 false。如果这是一个 while 结构，在相同的初始化条件下，print 一次都不会执行。

```
$ awk 'BEGIN {
> count=1;
> do
> print "This gets printed at least once";
> while(count!=1)
> }'
This gets printed at least once
```

下面打印 items-sold.txt 文件中，每种商品的总销售量，输出结果和之前的 while.awk 脚本相同，不同的是，这次试用 do-while 结构来实现。

```
$ cat dowhile.awk
{
   i=2;
   total=0;
   do {
      total = total + $i;
      i++;
      }
   while(i<=NF)
print "Item",$1,":",total,"quantities sold";
}

$ awk -f dowhile.awk items-sold.txt
```

```
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

73. for 循环

Awk 的 for 循环和 while 循环一样实用，但语法更简洁易用。

语法：

```
for(initialization;condition;increment/decrement)
```

for 循环一开始就执行 initialization，然后检查 condition,如果 condition 为 true，执行 actions, 然后执行 increment 或 decrement.如果 condition 为 true，就会一直重复执行 actions 和 increment/decrement。

下面例子打印文件总字段数总和。I 的初始值为 1，如果 i 小于等于字段数，则当前字段会被追加到总数中，每次循环 i 的值会增加 1.

```
$ echo "1 2 3 4" | awk '{ for (i=1;i<=NF;i++) total = total + $i } END { print total }'
10
```

下面的例子，使用 for 循环把文件中的字段反序打印出来。注意这次在 for 中使用 decrement 而不是 increment。

提示：每读入一行数据后，awk 会把 NF 的值设置为当前记录的总字段数。

该例用相反的顺序，从最后一个自动开始到第一个字段，逐个输出每个字段。然后输出换行。

反转示例:

```
$ cat forreverse.awk
BEGIN {
   ORS="";
}
{
   for (i=NF;i>0;i--)
   print $i," "
   print "\n";
}

$ awk -f forreverse.awk items-sold.txt
12   10   8   5   10   2   101
2   0   3   4   1   0   102
```

```
13   5   20   11   6   10   103
5   6   0   4   3   2   104
6   12   7   5   2   10   105
```

之前我们用 while 和 do-while 循环，输出了 items-sold.txt 中每种商品的销售量，现在我们推出该程序的 for 循环版本。

```
$ cat for.awk
{
   total=0;
   for(i=2;i<=NF;i++)
      total = total + $i
   print "Item ",$1," : ",total," quantities sold"
}
```

```
$ awk -f for.awk items-sold.txt
Item   101   :   47   quantities sold
Item   102   :   10   quantities sold
Item   103   :   65   quantities sold
Item   104   :   20   quantities sold
Item   105   :   42   quantities sold
```

74. break 语句

Break 语句用来跳出它所在的最内层的循环(while,do-while,或 for 循环)。请注意，break 语句只有在循环中才能使用。

打印某个月销售量为 0 的任何商品，即从第 2 至第 7 个字段中出现 0 的记录。

```
$ cat break.awk
{
   i=2;total=0;
   while(i++<=NF)
   {
      if($i == 0)
      {
         print "Item",$0,"had a month without item sold"
         break
      }
   }
}
```

```
$ awk -f break.awk items-sold.txt
Item 102 0 1 4 3 0 2    had a month without item sold
Item 104 2 3 4 0 6 5    had a month without item sold
```

如果执行下面的命令，要按 Ctrl+c 才能停止。

```
$ awk ‘BEGIN{while(1) print “forever”}’
```

这条语句一直打印字符串”forever”,因为条件永远不为 false。尽管死循环会用于操作系统和进程控制，但通常不是什么好事。

下面我们修改这个死循环，让它执行 10 次后退出

```
$ awk 'BEGIN{
> x=1;
> while(1)
> {
> print "Iteration"
> if( x==10)
> break;
> x++
> }}'
```

其输出结果如下:

```
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
```

75. continue 语句

Continue 语句跳过后面剩余的循环部分，立即进入下次循环。请注意，continue 只能用在循环当中。

下面打印 items-sold.txt 文件中所有商品的总销售量。其输出结果和 while.awk、dowhile.awk 以及 for.awk 一样，但是这里的 while 循环中使用 contine，使循环从 1 而不是从 2 开始。

```
$cat continue.awk
{
   i=1;total=0;
   while(i++<=NF)
   {
      if(i==1)
      continue
      total = total + $i
```

```
    }
    print "Item",$1,":",total,"quantities sold"
}

$ awk -f continue.awk items-sold.txt
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

下面的脚本在每次循环时都打印 x 的值，除了第 5 次循环，因为 continue 导致跳打印语句。

```
$ awk 'BEGIN{
> x=1;
> while(x<=10)
> {
> if(x==5)
> {
> x++
> continue
> }
> print "Value of x:",x;x++;
> }
> }'
```

输出结果如下:

```
Value of x: 1
Value of x: 2
Value of x: 3
Value of x: 4
Value of x: 6
Value of x: 7
Value of x: 8
Value of x: 9
Value of x: 10
```

76. exit 语句

exit 命令立即停止脚本的运行，并忽略脚本中其余的命令。

exit 命令接受一个数字参数最为 awk 的退出状态码，如果不提供参数，默认的状态码是 0.

下面的脚本执行到第 5 次循环时退出，因为 print 命令位于 exit 之后，所以输出的值只到 4 为止，到第 5 次循环时就退出了。

```
$ awk 'BEGIN{
> x=1;
> while(x<=10)
> {
> if(x==5)
> {
> x++
> exit
> }
> print "Value of x:",x;x++;
> }
> }'
```

其输出结果如下:

```
Value of x: 1
Value of x: 2
Value of x: 3
Value of x: 4
```

下面例子打印第一次出现的有个月没有卖出一件的商品的信息。和 break.awk 脚本很相似，区别在于，遇到某月为出售的商品时，退出脚本，而不是继续执行。

```
$ cat exit.awk
{
   i=2;total=0;
   while(i++<=NF)
   if($i==0) {
      print "Item",$1,"had a month with no item sold"
      exit
   }
}

$ awk -f exit.awk items-sold.txt
Item 102 had a month with no item sold
```

提示：104 号商品有的月份也没有卖出一件，但是并没有被打印，因为我们在循环中使用了 exit 命令。

第十二章：awk 关联数组

相比较与其他编程语言中的传统数组，awk 的数组更为强大。

Awk 的数组，都是关联数组，即一个数组包含多个”索引/值”的元素。索引没必要是一系列连续的数字，实际上，它可以使字符串或者数字，并且不需要指定数组长度。

语法：

```
arrayname[string]=value
```

- arrayname 是数组名称
- string 是数组索引
- value 是为数组元素赋的值

访问 awk 数组的元素

如果要访问数组中的某个特定元素，使用 arrayname[index] 即可返回该索引中的值。

一个简单的数组赋值示例:

```
$ cat array-assign.awk
BEGIN {
   item[101]="HD Camcorder";
   item[102]="Refrigerator";
   item[103]="MP3 Player";
   item[104]="Tennis Racket";
   item[105]="Laser Printer";
   item[1001]="Tennis Ball";
   item[55]="Laptop";
   item["na"]="Not Available";
   print item["101"];
   print item[102];
   print item["103"];
   print item[104];
   print item["105"];
   print item[1001];
   print item["na"];
}

$ awk -f array-assign.awk
HD Camcorder
Refrigerator
MP3 Player
Tennis Racket
Laser Printer
Tennis Ball
Not Available
```

请注意:

- 数组索引没有顺序，甚至没有从 0 或 1 开始，而是直接从 101….105 开始,然后直接跳到 1001，又降到 55，还有一个字符串索引”na”.
- 数组索引可以是字符串，数组的最后一个元素就是字符串索引，即”na”
- Awk 中在使用数组前，不需要初始化甚至定义数组，也不需要指定数组的长度。
- Awk 数组的命名规范和 awk 变量命名规范相同。

以 awk 的角度来说，数组的索引通常是字符串，即是你使用数组作为索引，awk 也会当做字符串来处理。下面的写法是等价的：

```
Item[101]=”HD Camcorder”
Item[“101”]=”HD Camcorder”
```

78. 引用数组元素

如下所示，可以使用 print 命令直接打印数组的元素，也可以把元素的值赋给其他变量以便后续使用。

```
print item[101]
x=item[105]
```

如果视图访问一个不存在的数组元素，awk 会自动以访问时指定的索引建立该元素，并赋予 null 值。为了避免这种情况，在使用前最后检测元素是否存在。

使用 if 语句可以检测元素是否存在，如果返回 true，说明改元素存在于数组中。

```
if ( index in array-name )
```

一个简单的引用数组元素的例子:

```
$ cat array-refer.awk
BEGIN {
  x = item[55];
  if ( 55 in item )
     print "Array index 55 contains",item[55];
  item[101]="HD Camcorder";
  if ( 101 in item )
     print "Array index 101 contains",item[101];
  if ( 1010 in item )
     print "Array index 1010 contains",item[1010];
}

$ awk -f array-refer.awk
Array index 55 contains
Array index 101 contains HD Camcorder
```

该例中：

- Item[55] 在引用前没有赋任何值，所以在引用是 awk 自动创建该元素并赋 null 值
- Item[101]是一个已赋值的缘分，所以在检查索引值时返回 true，打印该元素
- Item[1010]不存在，因此检查索引值时，返回 false，不会被打印

79.使用循环遍历 awk 数组

如果要访问数组中的所有元素，可以使用 for 的一个特殊用法来遍历数组的所有索引：

语法：

for (var in arrayname)

actions

其中:

- var 是变量名称
- in 是关键字
- arrayname 是数组名
- actions 是一系列要执行的 awk 语句，如果有多条语句，必须包含在{ }中。通过把索引值赋给变量 var，循环体可以把所有语句应用到数组中所有的元素上。

在示例"for (x in item)"中，x 是变量名，用来存放数组索引。

请注意，我们并没有指定循环执行的条件，实际上我们不比关系数组中有多少个元素，因为 awk 会自动判断，在循环结束前遍历所有元素。

下面的例子遍历数组中所有元素并打印出来。

```
$ cat array-for-loop.awk
BEGIN {
   item[101]="HD Camcorder";
   item[102]="Refrigerator";
   item[103]="MP3 Player";
   item[104]="Tennis Racket";
   item[105]="Laser Printer";
   item[1001]="Tennis Ball";
   item[55]="Laptop";
   item["no"]="Not Available";

   for(x in item)
   print item[x]
}

$ awk -f array-for-loop.awk
Not Available
Laptop
HD Camcorder
Refrigerator
MP3 Player
Tennis Racket
Laser Printer
Tennis Ball
```

80. 删除数组元素

如果要删除特定的数组元素，使用 delete 语句。一旦删除了某个元素，就再也获取不到它的值了。

语法：

```
delete arrayname[index];
```

删除数组内所有元素

```
for (var in array)
    delete array[var]
```

在 GAWK 中，可以使用单个 delete 命令来删除数组的所有元素:

```
    Delete array
```

此外，下面例子中,item[103]=””并没有删除整个元素，仅仅是给它赋了 null 值。

```
$ cat array-delete.awk
BEGIN {
  item[101]="HD Camcorder";
  item[102]="Refrigerator";
  item[103]="MP3 Player";
  item[104]="Tennis Racket";
  item[105]="Laser Printer";
  item[1001]="Tennis Ball";
  item[55]="Laptop";
  item["no"]="Not Available";

  delete item[102]
  item[103]=""
  delete item[104]
  delete item[1001]
  delete item["na"]

  for(x in item)
  print "Index",x,"contains",item[x]
}

$ awk -f array-delete.awk
Index no contains Not Available
Index 55 contains Laptop
Index 101 contains HD Camcorder
Index 103 contains
Index 105 contains Laser Printer
```

81. 多维数组

虽然 awk 只支持一维数组，但其奇妙之处在于，可以使用一维数组来模拟多维数组。

假定要创建下面的 2X2 维数组:

```
10    20
30    40
```

其中位于”1,1”的元素是 10，位于”1,2”的元素是 20，等等…,下面把 10 赋值给”1,1”的元素:

```
item[“1,1”]=10
```

即使使用了”1,1”作为索引值，它也不是两个索引，仍然是单个字符串索引，值为”1,1”。所以上面的写法中，实际上是把 10 赋给一维数组中索引”1,1”代表的值。

```
$ cat array-multi.awk
BEGIN {
   item["1,1"]=10;
   item["1,2"]=20;
   item["2,1"]=30;
   item["2,2"]=40

   for (x in item)
   print item[x]
}

$ awk -f array-multi.awk
30
20
40
10
```

现在把索引外面的引号去掉，会发生什么情况？即 item[1,1](而不是 item[“1,1”]):

```
$ cat array-multi2.awk
BEGIN {
   item[1,1]=10;
   item[1,2]=20;
   item[2,1]=30;
   item[2,2]=40

   for (x in item)
   print item[x]
}
$ awk -f array-multi2.awk
10
30
20
40
```

上面的例子仍然可以运行，但是结果有所不同。在多维数组中，如果没有把下标用引号引住，awk 会使用”\034”作为下标分隔符。

当指定元素 item[1,2]时，它会被转换为 item[“1\0342”]。Awk 用把两个下标用”\034”连接起来并转换为字符串。

当使用[“1,2”]时，则不会使用”\034”，它会被当做一维数组。

如下示例

```
$ cat array-multi3.awk
BEGIN {
   item["1,1"]=10;
   item["1,2"]=20;
   item[2,1]=30;
   item[2,2]=40;

   for(x in item)
   print "Index",x,"contains",item[x];
}

$ awk -f array-multi3.awk
Index 1,2 contains 20
Index 21 contains 30
Index 22 contains 40
Index 1,1 contains 10
```

其中:

- 索引”1,1”和”1,2”放在了引号中，所以被当做一维数组索引，awk 没有使用下标分隔符，因此，索引值被原封不动地输出。
- 所以 2,1 和 2,2 没有放在引号中，所以被当做多维数组索引，awk 使用下标分隔符来处理，因此索引变成”2\0341”和”2\0342”,于是在两个下标直接输出了非打印字符“\034”

82. SUBSEP 下标分隔符

通过变量 SUBSEP 可以把默认的下标分隔符改成任意字符，下面例子中，SUBSEP 被改成了分号:

```
$ cat array-multi4.awk
BEGIN {
   SUBSEP=":";

   item["1,1"]=10;
   item["1,2"]=20;

   item[2,1]=30;
   item[2,2]=40;

   for(x in item)
   print "Index",x,"contains",item[x];
}
```

```
$ awk -f array-multi4.awk
Index 1,2 contains 20
Index 2:1 contains 30
Index 2:2 contains 40
Index 1,1 contains 10
```

这个例子中，索引”1,1”和”1,2”由于放在了引号中而没有使用 SUBSEP 变量。

所以，使用多维数组时，最好不要给索引值加引号，如:

```
$ cat array-multi5.awk
BEGIN {
   SUBSEP=":";

   item[1,1]=10;
   item[1,2]=20;

   item[2,1]=30;
   item[2,2]=40;

   for(x in item)
   print "Index",x,"contains",item[x];
}

$ awk -f array-multi5.awk
Index 1:1 contains 10
Index 2:1 contains 30
Index 1:2 contains 20
Index 2:2 contains 40
```

83. 用 asort 为数组排序

asort 函数重新为元素值排序，并且把索引重置为从 1 到 n 的值，此处 n 代表数组元素个数。

假定一个数组有两个元素:item[“something”]=”B - I’m big b”和 item[“notsure”]=”A – I’m big a” 调用 asort 函数手，数组会以元素值排序，变成:item[1]=”A – I’m big a” 和 item[2]=”B – I’m big b”

下面例子中，数组索引是非连续的数字和字符串，调用 asort 后，元素值被排序，并且索引值变成 1,2,3,4…. 请注意，asort 函数会返回数组元素的个数。

```
$ cat asort.awk
BEGIN {
   item[101]="HD Camcorder";
   item[102]="Refrigerator";
```

```
    item[103]="MP3 Player";
    item[104]="Tennis Racket";
    item[105]="Laser Printer";
    item[1001]="Tennis Ball";
    item[55]="Laptop";
    item["na"]="Not Available";

    print "---------- Before asort ----------"
    for(x in item)
    print "Index",x,"contains",item[x]
    total = asort(item);

    print "---------- After asort ----------"
    for(x in item)
    print "Index",x,"contains",item[x]
    print "Return value from asort:",total;
}

$ awk -f asort.awk
---------- Before asort ----------
Index 55 contains Laptop
Index 101 contains HD Camcorder
Index 102 contains Refrigerator
Index 103 contains MP3 Player
Index 104 contains Tennis Racket
Index 105 contains Laser Printer
Index na contains Not Available
Index 1001 contains Tennis Ball
---------- After asort ----------
Index 4 contains MP3 Player
Index 5 contains Not Available
Index 6 contains Refrigerator
Index 7 contains Tennis Ball
Index 8 contains Tennis Racket
Index 1 contains HD Camcorder
Index 2 contains Laptop
Index 3 contains Laser Printer
Return value from asort: 8
```

这个例子中，asort 之后，数组打印顺序不是按索引值从 1 到 8，而是随机的。可以用下面的方法，按索引值顺序打印:

```
$ cat asort1.awk
BEGIN {
    item[101]="HD Camcorder";
    item[102]="Refrigerator";
```

```
    item[103]="MP3 Player";
    item[104]="Tennis Racket";
    item[105]="Laser Printer";
    item[1001]="Tennis Ball";
    item[55]="Laptop";
    item["na"]="Not Available";
    total = asort(item);

    for(i=1;i<=total;i++)
    print "Index",i,"contains",item[i]
}

$ awk -f asort1.awk
Index 1 contains HD Camcorder
Index 2 contains Laptop
Index 3 contains Laser Printer
Index 4 contains MP3 Player
Index 5 contains Not Available
Index 6 contains Refrigerator
Index 7 contains Tennis Ball
Index 8 contains Tennis Racket
```

或许你已经注意到，一旦调用 asort 函数，数组原始的索引值就不复存在了。因此，你可能想在不改变原有数组索引的情况下，使用新的索引值创建一个新的数组。

下面的例子中，原始数组 item 不会被修改，相反，使用排序后的新索引值创建新数组 itemnew，即 itemnew[1],itemnew[2],itemnew[3],等等。

```
total = asort(item,itemnew);
```

再次申明，务必牢记 asort 函数按元素值排序,但排序后使用从 1 开始的新索引值，原先的索引被覆盖掉了。

84. 用 asorti 为索引排序

和以元素值排序相似，也可以取出所有索引值，排序，然后把他们保存在新数组中。

下面的例子展示了 asort 和 asorti 的不同,请牢记下面两点:

- asorti 函数为索引值(不是元素值)排序，并且把排序后的元素值当做元素值保存。
- 如果使用 asorti(state)将会丢失原始元素值，即索引值变成了元素值。因此为了保险起见，通常给 asorti 传递两个参数，即 asorti(state,statebbr).这样一来，原始数组 state 就不会被覆盖了。

```
$ cat asorti.awk
BEGIN {
   state["TX"]="Texas";
   state["PA"]="Pennsylvania";
   state["NV"]="Nevada";
   state["CA"]="California";
   state["AL"]="Alabama";

   print "---------- Function: asort ----------"
   total = asort(state,statedesc);
   for(i=1;i<=total;i++)
   print "Index",i,"contains",statedesc[i];

   print "---------- Function: asorti ----------"
   total = asorti(state,stateabbr);
   for(i=1;i<=total;i++)
   print "Index",i,"contains",stateabbr[i];
}

$ awk -f asorti.awk
---------- Function: asort ----------
Index 1 contains Alabama
Index 2 contains California
Index 3 contains Nevada
Index 4 contains Pennsylvania
Index 5 contains Texas
---------- Function: asorti ----------
Index 1 contains AL
Index 2 contains CA
Index 3 contains NV
Index 4 contains PA
Index 5 contains TX
```

第十三章：其他 awk 命令

85. 使用 printf 格式化输出

printf 可以非常灵活、简单地以你期望的格式输出结果。

语法:

```
printf “print format”, variable1,variable2,etc.
```

printf 中的特殊字符

printf 中可以使用下面的特殊字符

特殊字符	描述
\n	换行
\t	制表符
\v	垂直制表符
\b	退格
\r	回车符
\f	换页

使用换行符把 Line1 和 Line2 打印在单独的行里:

```
$ awk 'BEGIN {printf "Line 1\nLine 2\n"}'
Line 1
Line 2
```

以制表符分隔字段，Field 1 后面有两个制表符:

```
$ awk 'BEGIN { printf "Field 1\t\tField 2\tField 3\tField 4\n"}'
Field 1                 Field 2 Field 3 Field 4
```

每个字段后面使用垂直制表符:

```
$ awk 'BEGIN { printf "Field 1\vField 2\vField 3\vField 4\n" }'
Field 1
          Field 2
                    Field 3
                              Field 4
```

下面的例子中，除了第 4 个字段外，每个字段后面使用退格符，这会擦除前三个字段最后的数字。如”Field 1”会被显示为”Field”,因为最后一个字符被退格符擦除了。然而”Field 4”会照旧输出，因为它后面没有使用\b.

```
$ awk 'BEGIN { printf "Field 1\bField 2\bField 3\bField 4\n" }'
Field Field Field Field 4
```

下面的例子，打印每个字段后，执行一个“回车”，在当前打印的字段的基础上，打印下一个字段。这就意味着，最后只能看到”Field 4”,因为其他的字段都被覆盖掉了.

```
$ awk 'BEGIN { printf "Field 1\rField 2\rField 3\rField 4\n" }'
Field 4
```

使用 OFS,ORS

当使用 print(不是 printf)打印多个以逗号分隔的字段时，awk 默认会使用内置变量 OFS 和 ORS 处理输出。

下面例子展示 OFS 和 ORS 对单个 print 的影响:

```
$ cat print.awk
BEGIN {
   FS=",";
   OFS=":";
```

```
    ORS="\n--\n";
}
{
    print $2,$3
}

$ awk -f print.awk items.txt
HD Camcorder:Video
--
Refrigerator:Appliance
--
MP3 Player:Audio
--
Tennis Racket:Sports
--
Laser Printer:Office
--
```

Printf 不受 OFS,ORS 影响

printf 不会使用 OFS 和 ORS，它只根据”format”里面的格式打印数据，如下所示:

```
$ cat printf1.awk
BEGIN {
    FS=",";
    OFS=":";
    ORS="\n--\n";
}
{
     printf "%s^^%s\n",$2,$3
}

$ awk -f printf1.awk    items.txt
HD Camcorder^^Video
Refrigerator^^Appliance
MP3 Player^^Audio
Tennis Racket^^Sports
Laser Printer^^Office
```

printf 格式化字符

格式化字符	描述
s	字符串
c	单个字符
d	数值
e	指数
f	浮点数

g	根据值决定使用 e 或 f 中较短的输出
o	八进制
x	十六进制
%	百分号

下面展示各个格式化字符的基本用法:

```
$ cat printf-format.awk
BEGIN {
   printf "s--> %s\n", "String"
   printf "c--> %c\n", "String"
   printf "s--> %s\n", 101.23
   printf "d--> %d\n", 101,23
   printf "e--> %e\n", 101,23
   printf "f--> %f\n", 101,23
   printf "g--> %g\n", 101,23
   printf "o--> %o\n", 0x8
   printf "x--> %x\n", 16
   printf "percentage--> %%\n", 17
}

$ awk -f printf-format.awk
s--> String
c--> S
s--> 101.23
d--> 101
e--> 1.010000e+02
f--> 101.000000
g--> 101
o--> 10
x--> 10
percentage--> %
```

指定打印列的宽度

要指定打印列的宽度，必须在%和格式化字符之间设置一个数字。该数字代表输出列的最小宽度，如果字符串的宽度比该数字小，会在输出列左侧加上空格以凑足该宽度。

下面例子演示如何指定输出列宽度：

```
$ cat printf-width.awk
BEGIN {
   FS=","
   printf "%3s\t%10s\t%10s\t%5s\t%3s\n", "Num","Description","Type","Price","Qty"
   printf "-------------------------------------------------------------\n"
}
{
   printf "%3d\t%10s\t%10s\t%g\t%d\n", $1,$2,$3,$4,$5
}
```

```
$ awk -f printf-width.awk items.txt
Num        Description                Type          Price      Qty
-------------------------------------------------------------------
101        HD Camcorder               Video         210        10
102        Refrigerator               Appliance     850        2
103        MP3 Player                 Audio         270        15
104        Tennis Racket              Sports        190        20
105        Laser Printer              Office        475        5
```

注意，即是我们指定了输出列的宽度，输出结果仍然没有对齐。因为我们指定的是最小宽度，而不是绝对宽度；如果字符串长度超过了指定的宽度，整个字符仍然会打印出来。所以，要在到底打印多少宽度的字符上下点功夫。

如果想在字符串超出指定宽度时，仍然以指定的宽度把字符串打印出来，可以使用 substr 函数(或者)在指定宽度的数字前面加一个小数点(稍后详述)。

上面例子中，第二个字段的长度超过了 10 个字符，并不是我们期望的结果。

在左边补空格,把字符串”Good”打印成 6 个字符:

```
$ awk 'BEGIN { printf "%6s\n","Good" }'
  Good
```

指定宽度为 6，但仍然输出所有字符:

```
$ awk 'BEGIN { printf "%6s\n", "Good Boy!" }'
Good Boy!
```

打印指定宽度(左对齐)

当字符串长度小于指定宽度时，如果要让它靠左对齐(右边补空格),那么要在%和格式化字符之间加上一个减号(-)。

“%6s”是右对齐:

```
$ awk 'BEGIN { printf "|%6s|\n", "Good" }'
|  Good|
```

“%-6s”是左对齐:

```
$ awk 'BEGIN { printf "|%-6s|\n", "Good" }'
|Good  |
```

打印美元标识

如果要在价钱之前加上美元符号,只需在格式化字符串之前(%之前)加上$即可:

```
$ cat printf-width2.awk
BEGIN {
  FS=","
  printf "%3s\t%10s\t%10s\t%5s\t%3s\n", "Num","Description","Type","Price","Qty"
  printf "-------------------------------------------------------------------\n"
```

```
}
{
   printf "%3d\t%10s\t%10s\t$%-.2f\t%d\n", $1,$2,$3,$4,$5
}

$ awk -f printf-width2.awk    items.txt
Num         Description                Type          Price      Qty
-------------------------------------------------------------------
101         HD Camcorder               Video         $210.00 10
102         Refrigerator          Appliance          $850.00 2
103         MP3 Player                 Audio         $270.00 15
104         Tennis Racket              Sports        $190.00 20
105         Laser Printer              Office        $475.00 5
```

字符串长度不足时补 0

默认情况下，右对齐时左边会补空格

```
$ awk 'BEGIN { printf "|%5s|\n", "100" }'
|   100|
```

为了在右对齐是，左边补 0 (而不是空格),在指定宽度的数字前面加一个 0，即使用”%05s”代替”%5s”

```
$ awk 'BEGIN { printf "|%05s|\n", "100" }'
|00100|
```

下面的例子中，在打印的商品数量之前补 0 :

```
$ vim printf-width3.awk
BEGIN {
   FS=","
   printf "%-3s\t%-10s\t%-10s\t%-5s\t%-3s\n", "Num","Description","Type","Price","Qty"
   printf "-------------------------------------------------------------------\n"
}

{
   printf "%-3d\t%-10s\t%-10s\t$%-.2f\t%03d\n", $1,$2,$3,$4,$5
}

$ awk -f printf-width3.awk    items.txt
Num         Description         Type                Price      Qty
-------------------------------------------------------------------
101         HD Camcorder        Video               $210.00 010
102         Refrigerator        Appliance           $850.00 002
103         MP3 Player          Audio               $270.00 015
104         Tennis Racket       Sports              $190.00 020
105         Laser Printer       Office              $475.00 005
```

以绝对宽度打印字符串

通过之前的例子可以知道，如果字符串长度超过指定的宽度，字符串仍然会整个被打印出来。

```
$ awk 'BEGIN { printf "%6s\n", "Good Boy!" }'
Good Boy!
```

如果要最多打印 6 个字符，要在指定宽度的数字前面加一个小数点，即使用”%.6s”代替”%6s”，这样即使字符串比指定宽度长，也只打印字符串中的前 6 个字符。

```
$ awk 'BEGIN { printf "%.6s\n", "Good Boy!" }'
Good B
```

这个例子并非适用于所有版本的 awk，在 GAWK 3.1.5 上可以，但在 GAWK 3.1.7 上则不行。

当然，以绝对宽度打印字符串，最可取的方法是使用 substr 函数

```
$ awk 'BEGIN { printf "%6s\n", substr("Good Boy!",1,6) }'
Good B
```

控制精度

数字前面的点，用来指定其数值精度。

下面的例子说明如何控制精度，展示了使用 .1 和 .4 时数值”101.23”的精度(使用的格式化字符有 d,e,f 和 g)

```
$ cat dot.awk
BEGIN {
   print "---------- Using .1 ----------"
   printf ".1d--> %.1d\n", 101.23
   printf ".1e--> %.1e\n", 101.23
   printf ".1f--> %.1f\n", 101.23
   printf ".1g--> %.1g\n", 101.23
   print "---------- Using .4 ----------"
   printf ".4d--> %.4d\n", 101.23
   printf ".4e--> %.4e\n", 101.23
   printf ".4f--> %.4f\n", 101.23
   printf ".4g--> %.4g\n", 101.23
}
$ awk -f dot.awk
---------- Using .1 ----------
.1d--> 101
.1e--> 1.0e+02
.1f--> 101.2
.1g--> 1e+02
---------- Using .4 ----------
.4d--> 0101
.4e--> 1.0123e+02
.4f--> 101.2300
.4g--> 101.2
```

把结果重定向到文件

Awk 中可以吧 print 语句打印的内容重定向到指定的文件中。下面的例子中，第一个 print 语句使用”>report.txt”创建 report.txt 文件并把内容保存到该文件中。随后的所有 print 语句都使用”>>report.txt”,把内容追加到已存在的 report.txt 文件中。

```
$ cat printf-width4.awk
BEGIN {
   FS=","
   printf "%-3s\t%-10s\t%-10s\t%-5s\t%-3s\n", "Num","Description","Type","Price","Qty" >
"report.txt"
   printf "----------------------------------------------------------------------\n" >> "report.txt"
}

{
   if($5 > 10)
   printf "%-3d\t%-10s\t%-10s\t$%-.2f\t%03d\n", $1,$2,$3,$4,$5 >> "report.txt"
}
~
$ awk -f printf-width4.awk items.txt
$ cat report.txt
Num        Description        Type                Price     Qty
----------------------------------------------------------------------
103        MP3 Player         Audio               $270.00 015
104        Tennis Racket      Sports              $190.00 020
```

另一中方法是不在 print 语句中使用”>”或”>>”，而是在执行 awk 脚本时使用重定向

```
$ vim printf-width5.awk
BEGIN {
   FS=","
   printf "%-3s\t%-10s\t%-10s\t%-5s\t%-3s\n", "Num","Description","Type","Price","Qty"
   printf "----------------------------------------------------------------------\n"
}

{
   if($5 > 10)
   printf "%-3d\t%-10s\t%-10s\t$%-.2f\t%03d\n", $1,$2,$3,$4,$5
}

$ awk -f printf-width5.awk items.txt >report.txt
$ cat report.txt
Num        Description        Type                Price     Qty
----------------------------------------------------------------------
103        MP3 Player         Audio               $270.00 015
104        Tennis Racket      Sports              $190.00 020
```

86. awk 内置数值函数

Awk 有很多内置的数值、字符串、输入输出函数，下面介绍其中的一部分。

int(n)函数
int()函数返回给定参数的整数部分值。n 可以是整数或浮点数，如果使用整数做参数，返回值即是它本身，如果指定浮点数，小数部分会被截断。

int 函数示例:
```
$ awk 'BEGIN {
> print int(3.534);
> print int(4);
> print int(-5.223);
> print int(-5);
> }'
```
输出结果为:
```
3
4
-5
-5
```

log(n)函数:
log(n)函数返回给定参数的自然对数，参数 n 必须是正数，否则会抛出错误

log 函数示例:
```
$ awk 'BEGIN {
print log(12);
print log(0);
print log(1);
print log(-1);
> }'
2.48491
-inf
0
awk: cmd. line:4: warning: log: received negative argument -1
nan
```
可以看到，该例子中 log(0)的值是无穷大，显示为-inf, log(-1)抛出了错误(非数字)

注意：你可以同时会看到 log(-1)抛出如下错误信息 awk: cmd. line:4: warning: log: received negative argument -1

sqrt(n)函数
sqrt 函数返回指定整数的正平方根，该函数参数也必须是整数，如果传递负数将会报错。

sqrt 函数示例：

```
$ awk 'BEGIN { print sqrt(3.5) }'
1.87083
$ awk 'BEGIN {
> print sqrt(16);
> print sqrt(0);
> print sqrt(-12);
> } '
4
0
awk: cmd. line:4: warning: sqrt: called with negative argument -12
nan
```

exp(n)函数

exp 函数返回 e 的 n 次幂

exp 函数示例:

```
$ awk 'BEGIN {
> print exp(123434346);
> print exp(0);
> print exp(-12);
> }'
awk: cmd. line:1: warning: exp: argument 1.23434e+08 is out of range
inf
1
6.14421e-06
```

这个例子中，exp(1234346)返回的值是 inf,因为这个值已经超出范围(溢出)了。

sin(n)函数

sin(n)返回 n 的正弦值,n 是弧度值

sin 函数示例:

```
$ awk 'BEGIN {
> print sin(90);
> print sin(45);
> }'
0.893997
0.850904
```

cos(n)函数

cos(n)返回 n 的余弦值，n 是弧度值

cos 函数示例:

```
$ awk 'BEGIN {
```

```
> print cos(90);
> print cos(45);
> }'
-0.448074
0.525322
```

atan2(m,n)函数
该函数返回 m/n 的反正切值,m 和 n 是弧度值。

atan2 函数示例:
```
$ awk 'BEGIN { print atan2(30,45) }'
0.588003
```

87. 随机数生成器

rand()函数用于产生 0~1 之间的随机数，它只返回 0~1 之间的数，绝不会返回 0 或 1。这些数在 awk 运行时是随机的，但是在多次运行中，又是可预知的。

awk 使用一套算法产生随机数，因为这个算法是固定的，所以产生的数也有重复的。

下面的例子产生 1000 个 0 到 100 之间的随机数，并且演示了每个数是怎么产生的

产生 1000 个随机数(0 到 100 之间):
```
$ cat rand.awk
BEGIN {
while(i<1000)
  {
    n = int(rand()*100);
    rnd[n]++;
    i++;
  }

for(i=0;i<=100;i++)
{
  print i,"Occured",rnd[i],"times";
}
}

$ awk -f rand.awk
0 Occured 11 times
1 Occured 8 times
```

2 Occured 9 times
3 Occured 15 times
4 Occured 16 times
5 Occured 5 times
6 Occured 8 times
7 Occured 9 times
8 Occured 7 times
9 Occured 7 times
10 Occured 11 times
11 Occured 7 times
12 Occured 10 times
13 Occured 9 times
14 Occured 6 times
15 Occured 18 times
16 Occured 10 times
17 Occured 10 times
18 Occured 9 times
19 Occured 8 times
20 Occured 11 times
21 Occured 13 times
22 Occured 10 times
23 Occured 9 times
24 Occured 15 times
25 Occured 8 times
26 Occured 3 times
27 Occured 17 times
28 Occured 9 times
29 Occured 13 times
30 Occured 11 times
31 Occured 9 times
32 Occured 12 times
33 Occured 12 times
34 Occured 9 times
35 Occured 6 times
36 Occured 13 times
37 Occured 15 times
38 Occured 6 times
39 Occured 9 times
40 Occured 7 times
41 Occured 8 times
42 Occured 6 times
43 Occured 8 times
44 Occured 10 times
45 Occured 7 times

46 Occured 10 times
47 Occured 8 times
48 Occured 16 times
49 Occured 12 times
50 Occured 6 times
51 Occured 15 times
52 Occured 6 times
53 Occured 12 times
54 Occured 8 times
55 Occured 13 times
56 Occured 6 times
57 Occured 16 times
58 Occured 5 times
59 Occured 7 times
60 Occured 11 times
61 Occured 12 times
62 Occured 14 times
63 Occured 11 times
64 Occured 9 times
65 Occured 6 times
66 Occured 7 times
67 Occured 10 times
68 Occured 8 times
69 Occured 12 times
70 Occured 13 times
71 Occured 9 times
72 Occured 10 times
73 Occured 11 times
74 Occured 7 times
75 Occured 13 times
76 Occured 13 times
77 Occured 10 times
78 Occured 5 times
79 Occured 12 times
80 Occured 17 times
81 Occured 8 times
82 Occured 7 times
83 Occured 10 times
84 Occured 12 times
85 Occured 12 times
86 Occured 11 times
87 Occured 14 times
88 Occured 4 times
89 Occured 8 times

```
90 Occured 15 times
91 Occured 10 times
92 Occured 15 times
93 Occured 8 times
94 Occured 11 times
95 Occured 5 times
96 Occured 12 times
97 Occured 11 times
98 Occured 7 times
99 Occured 11 times
100 Occured  times
```

通过这个例子可以看出，rand()函数产生的随机数有很高的重复率。

srand(n)函数

srand(n)函数使用给定的参数 n 作为种子来初始化随机数的产生过程。不论何时启动，awk 只会从 n 开始产生随机数，如果不指定参数 n，awk 默认使用当天的时间作为产生随机数的种子。

产生 5 个从 5 到 50 的随机数:

```
$ cat srand.awk
BEGIN {
   #Initialize the sedd with 5.
   srand(5);

   #Totally I want to generate 5 numbers
   total = 5;

   #maximun number is 50
   max = 50;
   count = 0;
   while(count < total)
   {
      rnd = int(rand()*max);
      if( array[rnd] == 0 )
      {
         count++;
         array[rnd]++;
      }
   }

   for ( i=5;i<=max;i++)
   {
      if (array[i])
         print i;
```

```
   }
}

$ awk -f srand.awk
14
16
23
33
35
```

该例中:

- 首先使用 rand()函数产生随机数，然后乘以期望的最大值，获得一个小于 50 的数
- 检测产生的数是否存在于数组中，如果不存在，增加数组的索引和循环数。本例中产生 5 个数
- 最后在 for 循环中，从最小到最大一次打印每个索引对应的元素值。

88. 常用字符串函数

下面是一些可以在所有风格的 awk 上运行的常用字符串函数。

index 函数
index 函数用来获取给定字符串在输入字符串中的索引(位置)。

下面的例子中，字符串”Cali”在字符串”CA is California”中的位置是 7.

也可以用 index 来检测指定的字符串(或者字符)是否存在于输入字符串中。如果指定的字符串没有出现，返回 0，就说明指定的字符串不存在，如下所示。

```
$ cat index.awk
BEGIN {
   state="CA is California"
   print "String CA starts at location",index(state,"CA");
   print "String Cali starts at location",index(state,"Cali");
   if(index(state,"NY")==0)
   print "String NY is not found in:",state
}

$ awk -f index.awk
String CA starts at location 1
String Cali starts at location 7
String NY is not found in: CA is California
```

length 函数
length 函数返回字符串的长度，下面例子将打印 items.txt 文件中每行字符串的总数。

```
$ awk '{print length($0)}' items.txt
30
33
28
32
30
```

split 函数

语法:

```
    split(input-string,output-array,separator)
```

split 函数把字符串分割成单个数组元素，其接受如下参数：

- input-string:这个是需要被分割的字符串
- output-array:这个是分割后的字符串存放的数组
- separator:分割字符串的字段分隔符

为了演示这个例子，要对原先的 items-sold.txt 文件做小小的改动，使其包含不同的字段分隔符，即使用冒号分隔商品编号和销售量，在销售量列表中，每个数字之间以逗号分隔。

为了统计某件特定商品的销售量，我们需要取出第 2 个字段(以逗号分隔的销售量数字列表)，然后使用逗号作为分隔符，将其分隔并保存在一个数组中，然后使用循环遍历数组统计总和。

```
$ cat items-sold1.txt
101:2,10,5,8,10,12
102:0,1,4,3,0,2
103:10,6,11,20,5,13
104:2,3,4,0,6,5
105:10,2,5,7,12,6

$ cat split.awk
BEGIN {
        FS=":"
}
{
        split($2,quantity,",");
        total=0;
        for(x in quantity)
        total=total+quantity[x];
        print "Item",$1,":",total,"quantities sold";
}

$ awk -f split.awk items-sold1.txt
Item 101 : 47 quantities sold
Item 102 : 10 quantities sold
Item 103 : 65 quantities sold
Item 104 : 20 quantities sold
Item 105 : 42 quantities sold
```

substr 函数

语法：

```
substr(input-string,location,length)
```

substr 函数从字符串中提取指定的部分(子串)，上面语法中：

- input-string:包含子穿的字符串
- location:子串的开始位置
- length:从 location 开始起，出去的字符串的总长度。这个选项是可选的，如果不指定长度，那么从 location 开始一直取到字符串的结尾

下面的例子从字符串的第 5 个字符开始，取到字符串结尾并打印出来。开始的 3 个字符是商品编号，第 4 个字符时逗号。所以下面的例子会跳过商品编号，打印剩余的内容。

```
$ awk '{ print substr($0,5) }' items.txt
HD Camcorder,Video,210,10
Refrigerator,Appliance,850,2
MP3 Player,Audio,270,15
Tennis Racket,Sports,190,20
Laser Printer,Office,475,5
```

从第 2 个字段的第 1 个字符起，打印 5 个字符:

```
$ awk -F"," '{ print substr($2,1,5) }' items.txt
HD Ca
Refri
MP3 P
Tenni
Laser
```

89. GAWK/NAWK 的字符串函数

下面这些函数只能在 GAWK 和 NAWK 中使用。

sub 函数：

语法：

```
sub(original-string,replacement-string,string-variable)
```

- sub 代表 substitution(替换)的意思
- original-string:将被替换掉的字符串。也可以是一个正则表达式。
- replacement-string:用来替换的字符串
- string-variable:既是输入字符串，也是输出字符串。必须小心，一旦替换操作成功执行，你将丢失该字符串原来的值。

下面的例子中：

- original-string:这里是一个正则表达式 C[Aa],匹配 CA 或者 Ca
- replacement-string:如果匹配到 original-string，则用”KA”替换之
- string-variable:执行替换操作之前，该变量保存的是输入字符串(替换前的字符串)，一旦执行替换操作，该变量保存的是输出字符串(替换后的字符串)

需要注意的是，sub 函数只替换第一次出现的 original-string。

```
$ cat sub.awk
BEGIN {
    state="CA is California"
    sub("C[Aa]","KA",state);
    print state;
}
$ awk -f sub.awk
KA is California
```

第 3 个参数 string-variable 是可选的，如果没有指定，awk 会使用$0(当前记录)做为第 3 个参数，如下所示。这个例子把头两个字符从”10”替换为”20”,所有，商品编号 101 变成了 201,102 变成了 202，依此类推。

```
$ awk '{ sub("10","20"); print $0 }' items.txt
201,HD Camcorder,Video,210,10
202,Refrigerator,Appliance,850,2
203,MP3 Player,Audio,270,15
204,Tennis Racket,Sports,190,20
205,Laser Printer,Office,475,5
```

如果替换操作执行成功，sub 函数返回 1，否则返回 0.

仅打印替换成功的记录 ：

```
$ awk '{ if(sub("HD","High-Def")) print $0; }' items.txt
101,High-Def Camcorder,Video,210,10
```

gsub 函数

gsub 代表全局替换。和 sub 基本相同，只不过它把所有出现的 original-string 都替换为 replacement-string。

下面例子中，”CA”和”Ca”都将被替换成”KA”:

```
$ cat gsub.awk
BEGIN {
    state="CA is California"
    gsub("C[Aa]","KA",state);
    print state;
}
$ awk -f gsub.awk
KA is KAlifornia
```

和 sub 函数相同，第 3 个参数也是可选的，如果没有指定，则使用$0 作为第 3 个参数。

下面的例子把所有出现的”10”都替换为”20”，它不仅会替换商品编号，如果其他字段包含了 10，也会进行替换。

```
$ awk '{ gsub("10","20"); print $0 }' items.txt
201,HD Camcorder,Video,220,20
```

```
202,Refrigerator,Appliance,850,2
203,MP3 Player,Audio,270,15
204,Tennis Racket,Sports,190,20
205,Laser Printer,Office,475,5
```

match 函数和 RSTART,RLENGTH 变量

match 函数从输入字符串中检索给定的字符串(或正则表达式)，当检索到字符串时，返回一个正数值。

语法：

```
    match(input-string,search-string)
```

- input-string:这是需要被检索的字符串
- search-string:要检索的字符串，需要包含在 input-string 中，它可以是一个正则表达式

下面例子在 state 字符串中检索”Cali”,如果 Cali 出现，则打印一条检索成功的消息。

```
$ cat match.awk
BEGIN {
  state="CA is California"
  f(match(state,"Cali"))
  {
     print substr(state,RSTART,RLENGTH),"is present in:",state;
  }
}
$ awk -f match.awk
Cali is present in: CA is California
```

match 函数设置了两个特殊变量，这个例子在调用 substr 函数时使用了它们用来打印检索成功的消息。

- RSTART – search-string 的开始位置
- RLENGTH – search-string 的长度

90. GAWK 字符串函数

tolower 和 toupper 函数仅在 GAWK 中可以使用。正如函数名一样，这两个函数把给定的字符串转换成小写或大写形式，如下所示：

```
$ awk '{ print tolower($0) }' items.txt
101,hd camcorder,video,210,10
102,refrigerator,appliance,850,2
103,mp3 player,audio,270,15
104,tennis racket,sports,190,20
105,laser printer,office,475,5

$ awk '{ print toupper($0) }' items.txt
101,HD CAMCORDER,VIDEO,210,10
```

```
102,REFRIGERATOR,APPLIANCE,850,2
103,MP3 PLAYER,AUDIO,270,15
104,TENNIS RACKET,SPORTS,190,20
105,LASER PRINTER,OFFICE,475,5
```

91.处理参数(ARGC,ARGV,ARGIND)

之前我们已经讨论过 awk 内置变量，FS,NFS,RS,NR,FILENAME,OFS 和 ORS，这些变量在所有的 awk 版本(包括 nawk 和 gawk)上都可以使用。

- 本节中提到的环境变量仅仅适用于 nawk 和 gawk
- 可以使用 ARGC 和 ARGV 从命令行传递一些参数给 awk 脚本
- ARGC 保存着传递给 awk 脚本的所有参数的个数
- ARGV 是一个数组，保存着传递给 awk 脚本的所有参数，其索引范围从 0 到 ARGC
- 当传递 5 个参数是，ARGC 的值为 6
- ARGV[0]的值永远是 awk

下面的例子 arguments.awk 演示 ARGC 和 ARGV 的作用:

```
$ cat arguments.awk
BEGIN {
   print "ARGC=",ARGC
   for(i=0;i<ARGC;i++)
      print ARGV[i]
}
$ awk -f arguments.awk arg1 arg2 arg3 arg4 arg5
ARGC= 6
awk
arg1
arg2
arg3
arg4
arg5
```

在下面的例子中:

- 我们以”—参数名 参数值”的格式给 awk 脚本传递一些参数
- awk 脚本获取传递元素的内容和数量作为参数
- 如果把”—item 104 –qty 25”作为参数传递给 awk 脚本，awk 会把商品 104 的数量设置为 25
- 如果把”—item 105 –qty 3”作为参数传递给 awk 脚本，awk 会把商品 105 的数量设置为 3

```
$ cat argc-argv.awk
BEGIN {
   FS=",";
```

```
    OFS=",";
    for(i=0;i<ARGC;i++)
    {
        if(ARGV[i] == "--item")
        {
            itemnumber=ARGV[i+1];
            delete ARGV[i]
            i++;
            delete ARGV[i]
        }
        else if (ARGV[i]=="--qty")
        {
            quantity=ARGV[i+1]
            delete ARGV[i]
            i++;
            delete ARGV[i]
        }
    }
}

{
    if ($1==itemnumber)
    print $1,$2,$3,$4,quantity
    else
    print $0
}

$ awk -f argc-argv.awk --item 104 --qty 25 items.txt
101,HD Camcorder,Video,210,10
102,Refrigerator,Appliance,850,2
103,MP3 Player,Audio,270,15
104,Tennis Racket,Sports,190,25
105,Laser Printer,Office,475,5
```

在 gawk 中，当前处理的文件被存放在数组 ARGV 中，该数组在 body 区域被访问。ARGIND 是 ARGV 的一个索引，其对应的值是当前正在处理的文件名。

当 awk 脚本仅处理一个文件时，ARGIND 的值是 1，ARGV[ARGIND]会返回当前正在处理的文件名。

下面的例子只有 body 区域，打印 ARGIND 的值以及 ARGV[ARGIND]

```
$ cat argind.awk
{
    print "ARGIND:",ARGIND
```

```
    print "Current file:",ARGV[ARGIND]
}
```
调用这个脚本时，可以传递两个文件给它，没处理一行记录就会打印两条数据。这个例子意在让你搞清楚 ARGIND 和 ARGV[ARGIND]的值是怎么存储的。
```
$ awk -f argind.awk items.txt items-sold1.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 1
Current file: items.txt
ARGIND: 2
Current file: items-sold1.txt
ARGIND: 2
Current file: items-sold1.txt
ARGIND: 2
Current file: items-sold1.txt
ARGIND: 2
Current file: items-sold1.txt
ARGIND: 2
Current file: items-sold1.txt
```

92. OFMT

内置变量 OFMT 仅适用于 NAWK 和 GAWK。

当一个数值被转换成字符串并打印时，awk 适用 OFMT 格式来决定如何打印这些值。OFMT 默认值是”%.6g”,包括小数点两边的数字，它打印一共 6 个长度的字符。

在使用 g 时，必须数清楚小数点两边的数字位数，如”%.4g”表示包括小数点两侧，一共打印 4 个字符。

在使用 f 时，只需要数清楚小数点后面的数字位数，如”%.4f”表示小数点后面会打印 4 个字符。在此不必关系小数点左边有多少个字符。

下面的脚本 ofmt.awk 演示在使用不同的 OFMT 值(g 和 f)时，如何打印字符串
```
$ cat ofmt.awk
BEGIN {
    total=143.123456789;
```

```
    print "--- using g ---"
    print "Default OFMT:",total;
    OFMT="%.3g"
    print "%.3g OFMT:",total;
    OFMT="%.4g"
    print "%.4g OFMT:",total;
    OFMT="%.5g"
    print "%.5g OFMT:",total;
    OFMT="%.6g"
    print "%.6g OFMT:",total;
    print "--- using f ---"
    OFMT="%.0f";
    print "%.0f OFMT:",total;
    OFMT="%.1f";
    print "%.1f OFMT:",total;
    OFMT="%.2f";
    print "%.2f OFMT:",total;
    OFMT="%.3f";
    print "%.3f OFMT:",total;
}
$ awk -f ofmt.awk
--- using g ---
Default OFMT: 143.123
%.3g OFMT: 143
%.4g OFMT: 143.1
%.5g OFMT: 143.12
%.6g OFMT: 143.123
--- using f ---
%.0f OFMT: 143
%.1f OFMT: 143.1
%.2f OFMT: 143.12
%.3f OFMT: 143.123
```

93. GAWK 内置的环境变量

本节讨论的内置变量仅适用于 GAWK。

ENVIRON

如果能在 awk 脚本中访问 shell 环境变量会十分有用。ENVIRON 是一个包含所有 shell 环境变量的数组，其索引就是环境变量的名称。

如元素 ENVIRON[“PATH”]的值是环境变量 PATH 的值。

下面的例子打印所有环境变量名称和值。

```
$ cat environ.awk
BEGIN {
   OFS="="
   for(x in ENVIRON)
   print x,ENVIRON[x]
}
$ awk -f environ.awk
MAIL=/var/mail/root
CPU=x86_64
XDG_CONFIG_DIRS=/etc/xdg
LC_CTYPE=en_US.UTF-8
INPUTRC=/etc/inputrc
HOST=mkey
PWD=/root
FROM_HEADER=
……..
```

IGNORECASE

默认情况下，IGNORECASE 的值是 0，所有 awk 区分大小写。

当把 IGNORECASE 的值设置为 1 时，awk 则不区分大小写，这在使用正则表达式和比较字符串时很有效率。

下面的例子不会打印任何内容，因为它想用匹配小写的”video”，但 items.txt 中包含的却是大写的”Video”.

```
awk '/video/ {print}' items.txt
```

然而，当把 IGNORECASE 设置为 1 时，就能匹配并打印包含”Video”的行，因为现在 awk 不区分大小写。

```
$ awk 'BEGIN{IGNORECASE=1} /video/{print}' items.txt
101,HD Camcorder,Video,210,10
```

下面的例子同时支持字符串比较和正则表达式。

```
$ cat ignorecase.awk
BEGIN {
 FS=",";
 IGNORECASE=1;
}
{
 if ($3 == "video") print $0;
 if ($2 ~ "TENNIS") print $0;
}
$ awk -f ignorecase.awk items.txt
101,HD Camcorder,Video,210,10
104,Tennis Racket,Sports,190,20
```

ERRNO
当执行 I/O 操作(比如 getline)出错时，变量 ERRNO 会保存错误信息。

下面的例子试图用 getline 读取一个不存在的文件，此时 ERRNO 的内容将会是”No such file or directory”。

```
$ cat errno.awk
{
   print $0;
   x = getline < "dummy-file.txt"
   if ( x == -1 )
      print ERRNO
   else
      print $0
}

$ awk -f errno.awk items.txt
101,HD Camcorder,Video,210,10
No such file or directory
102,Refrigerator,Appliance,850,2
No such file or directory
103,MP3 Player,Audio,270,15
No such file or directory
104,Tennis Racket,Sports,190,20
No such file or directory
105,Laser Printer,Office,475,5
No such file or directory
```

94. pgawk – awk 运行分析器

pgawk 程序用来生成 awk 执行的结果报告。用 pgawk 可以看到 akw 每次执行了多少条语句(以及用户自定义的函数)。

首先建立下面的 awk 脚本作为样本，以供 pgawk 执行，然后分析其结果。

```
$ cat profiler.awk
BEGIN {
   FS=",";
   print "Report Generate On:",strftime("%a %b %d %H:%M:%S %Z %Y",systime());
}

{
   if ( $5 <= 5 )
      print "Buy More: Order",$2,"immediately!"
   else
```

```
        print "Sell More: Give discount on",$2,"immediately!"
}

END {
   print "----------"
}
```

接下来使用 pgawk(不是直接调用 awk)来执行该样本脚本。

```
$ pgawk -f profier.awk items.txt
Report Generate On: Tue Apr 09 15:56:26 CST 2013
Sell More: Give discount on HD Camcorder immediately!
Buy More: Order Refrigerator immediately!
Sell More: Give discount on MP3 Player immediately!
Sell More: Give discount on Tennis Racket immediately!
Buy More: Order Laser Printer immediately!
----------
```

pgawk 默认会创建输出文件 profier.out(或者 awkprof.out)，使用—profier 选项可以指定输出文件，如下所示。

```
$ pgawk --profile=myprofiler.out -f profier.awk items.txt
```

查看默认的输出文件 awkprof.out 来弄清楚每条单独的 awk 语句的执行次数。

```
$ cat awkprof.out
   # gawk profile, created Tue Apr    9 15:56:26 2013

   # BEGIN block(s)

   BEGIN {
        1       FS = ","
        1       print "Report Generate On:", strftime("%a %b %d %H:%M:%S %Z %Y", systime())
   }

   # Rule(s)

        5    {
        5       if ($5 <= 5) { # 2
        2           print "Buy More: Order", $2, "immediately!"
        3       } else {
        3           print "Sell More: Give discount on", $2, "immediately!"
      }
   }

   # END block(s)

   END {
```

```
        1       print "----------"
  }
```

查看 awkprof.out 文件时，务必牢记:

- 左侧一列有一个数字，标识着该 awk 语句执行的次数。如 BEGIN 区域里的 print 语句仅执行了一次(duh!)。而 while 循环执行了 5 次。
- 对于任意一个条件判断语句，左边有一个数字，括号右边也有一个数字。左边数字代表该判断语句执行了多少次，右边数字代表判断语句为 true 的次数。上面的例子中，由(# 2)可以判定，if 语句执行了 5 次，但只有 2 次为 true。

95. 位操作

和 C 语言类似，awk 也可以进行位操作。在日常工作中用不到，但可以说明你可以利用位操作来做什么。

下面表格列出了十进制数字及其二进制形式

十进制	二进制
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

AND (按位与)

要使 AND 结果为 1，两个操作数都必须为 1.

- 0 and 0 = 0
- 0 and 1 = 0
- 1 and 0 = 0
- 1 and 1 =1

例如，在十进制数 15 和 25 上执行 AND 操作，结果是二进制的 01001，也就是十进制的 9.

- 15 = 01111
- 25 = 11001
- 15 and 25 = 01001

OR(按位或)

要使 OR 结果为 1，任意一个操作数为 1 即可

- 0 or 0 = 0
- 0 or 1 = 1
- 1 or 0 = 1
- 1 or 1 = 1

例如，在十进制数 15 和 25 上执行 OR 操作，结果是二进制的 11111，也就是十进制的 31.

- 15 = 01111
- 25 = 11001
- 15 and 25 = 11111

XOR(按位异或)

要使 XOR 结果为 1，必须只有一个操作数为 1

- 0 xor 0 = 0
- 0 xor 1 = 1
- 1 xor 0 = 1
- 1 xor 1 = 0

例如，在十进制数 15 和 25 上执行 XOR 操作，结果是二进制的 10110，也就是十进制的 22.

- 15 = 01111
- 25 = 11001
- 15 and 25 = 10110

complement(取反码)

反码把 0 变成 1，把 1 变成 0

如，给 15 取反码。

- 15 = 01111
- 15 compl= 10000

Left Shift(左移)

该函数把操作数向左位移，可以指定位移多少次,位移后右边补 0。

例如把十进制数 15 向左位移(移两次),结果将是二进制的 111100，即十进制的 60.

- 15 = 1111
- lshift twice = 111100

Right Shift(右移)

该函数把操作数向右位移，可以指定位移多少次,位移后左边补 0。

例如把十进制数 15 向右位移(移两次),结果将是二进制的 0011，即十进制的 3.

- 15 = 1111
- lshift twice = 0011

awk 位移函数示例

```
$ cat bits.awk
BEGIN {
   number1=15
   number2=25
   print "AND: " and(number1,number2);
   print "OR: " or(number1,number2);
   print "XOR: " xor(number1,number2);
   print "LSHIFT: " lshift(number1,2);
```

```
    print "RSHIFT: " rshift(number1,2);
}

$ awk -f bits.awk
AND: 9
OR: 31
XOR: 22
LSHIFT: 60
RSHIFT: 3
```

96.用户自定义函数

awk 运行用户自定义函数，这在编写大量代码同时又要多次重复执行其中某些片段时特别有用，这些片段就适合定义成函数。

语法：

```
function fn-name(parameters)
{
    function-body
}
```

其中:

- fn-name: 函数名，和 awk 变量名一样，用户定义的函数名应该以字母开头，后续字符可以数字、字母或下划线，关键字不能用做函数名
- parameters:多个参数要使用逗号分开，也可以定义一个没有参数的函数
- function-body:一条或多条 awk 语句

如果你在 awk 中已经使用了某个名字作为变量名，那么它就不能再用来作函数名。

下面的例子创建了一个简单的用户自定义函数——discount,它返回商品打折后的价钱，如 discount(10)返回打九折后的价钱。

对于任意一种商品，如果数量不大于 10，则打九折，否则打 5 折。

```
$ cat function.awk
BEGIN {
    FS=","
    OFS=","
}
{
    if ($5 <= 10)
    print $1,$2,$3,discount(10),$5
    else
    print $1,$2,$3,discount(50),$5
}
```

```
function discount(percentage)
{
   return $4 - ($4*percentage/100);
}

$ awk -f function.awk items.txt
101,HD Camcorder,Video,189,10
102,Refrigerator,Appliance,765,2
103,MP3 Player,Audio,135,15
104,Tennis Racket,Sports,95,20
105,Laser Printer,Office,427.5,5
```

自定义函数另外一个作用是打印 debug 信息。

下面是一个简单的 mydebug 函数:

```
$cat function-debug.awk
{
   i=2; total=0;
   while (i <= NF ) {
   mydebug("quantity is "$i);
   total = total + $i;
   i++;
   }
   print "Item",$1,":",total,"quantities sold";
}

function mydebug( message )
{
   printf("DEBUG[%d]>%s\n",NR,message);
}

$ awk -f function-debug.awk items-sold.txt
DEBUG[1]>quantity is 2
DEBUG[1]>quantity is 10
DEBUG[1]>quantity is 5
DEBUG[1]>quantity is 8
DEBUG[1]>quantity is 10
DEBUG[1]>quantity is 12
Item 101 : 47 quantities sold
DEBUG[2]>quantity is 0
DEBUG[2]>quantity is 1
DEBUG[2]>quantity is 4
DEBUG[2]>quantity is 3
DEBUG[2]>quantity is 0
```

```
DEBUG[2]>quantity is 2
Item 102 : 10 quantities sold
DEBUG[3]>quantity is 10
DEBUG[3]>quantity is 6
DEBUG[3]>quantity is 11
DEBUG[3]>quantity is 20
DEBUG[3]>quantity is 5
DEBUG[3]>quantity is 13
Item 103 : 65 quantities sold
DEBUG[4]>quantity is 2
DEBUG[4]>quantity is 3
DEBUG[4]>quantity is 4
DEBUG[4]>quantity is 0
DEBUG[4]>quantity is 6
DEBUG[4]>quantity is 5
Item 104 : 20 quantities sold
DEBUG[5]>quantity is 10
DEBUG[5]>quantity is 2
DEBUG[5]>quantity is 5
DEBUG[5]>quantity is 7
DEBUG[5]>quantity is 12
DEBUG[5]>quantity is 6
Item 105 : 42 quantities sold
```

97. 使输出摆脱语言依赖(国际化)

在使用 awk 脚本打时，可能需要用 print 打印指定的 header 和 footer 信息。你或许会用英文把 header 和 footer 写成固定的内容。但在其他语言中，你希望它会打印什么信息？最终你可能会把脚本复制过去，然后修改要打印的固定信息，来适应当前的语言环境。

有个更容易的方法来实现这个目的——国际化，这样使用同一个脚本，仅需要在运行脚本时修改那些固定的输出信息即可。

在运行大型程序，而出于某些原因你要频繁地修改那些固定的输出信息时；或者希望用户能自行修改要输出的内容时，这个方法也很有用。

下面的例子演示了在 awk 中实现国际化的关键 4 步。

步骤 1 – 建立文本域

创建一个文本域文件，并把它和 awk 要搜寻的目录绑定。下面以当前目录为例。

```
$ cat iteminfo.awk
BEGIN {
   FS=","
   TEXTDOMAIN = "item"
```

```
    bindtextdomain(".")
    print _"START_TIME:" strftime("%a %b %d %H:%M:%S %Z %Y",systime());
    printf "%-3s\t",_"Num";
    printf "%-10s\t",_"Description"
    printf "%-10s\t",_"Type"
    printf "%-5s\t",_"Price"
    printf "%-3s\n",_"Qty"
    printf _"-------------------------------------------------\n"
}

{
    printf "%-3d\t%-10s\t%-10s\t%-.2f\t%03d\n",$1,$2,$3,$4,$5
}
```

注意：这个例子中，前面带”_”的字符串均可以自行定义。字符串前面的_(下划线)不会影响字符串内容的打印，即它和下面的输出完全相同。

```
$ awk -f iteminfo.awk items.txt
START_TIME:Thu Apr 11 11:37:40 CST 2013
Num         Description         Type                    Price      Qty
-------------------------------------------------
101         HD Camcorder        Video                   210.00     010
102         Refrigerator        Appliance               850.00     002
103         MP3 Player          Audio                   270.00     015
104         Tennis Racket       Sports                  190.00     020
105         Laser Printer       Office                  475.00     005
```

步骤 2：生成.po 文件

建立如下可移植对象文件(扩展名.po)，注意，除了使用—gen-po 之外，也可以使用”-W gen-po”

```
$ gawk --gen-po -f iteminfo.awk > iteminfo.po

$ cat iteminfo.po
#: iteminfo.awk:5
msgid "START_TIME:"
msgstr ""

#: iteminfo.awk:6
msgid "Num"
msgstr ""

#: iteminfo.awk:7
msgid "Description"
msgstr ""

#: iteminfo.awk:8
msgid "Type"
```

```
msgstr ""

#: iteminfo.awk:9
msgid "Price"
msgstr ""

#: iteminfo.awk:10
msgid "Qty"
msgstr ""

#: iteminfo.awk:11
msgid "-------------------------------------------------\n"
""
msgstr ""
```

现在修改该文件中相应的内容。例如要显示”Report Generated on:”(替换原来的”START_TIME”),那么修改 iteminfo.po 文件,把 START_TIME 右下方的 msgstr 修改为“Reprot Genterated On:”

```
$ cat iteminfo.po
#: iteminfo.awk:5
msgid "START_TIME:"
msgstr "Report Generated On:"
```

提示：这个例子中，其余的 msgstr 字符串均被置空。

步骤 3：生成消息对象

使用 msgfmt 命令(从可移植对象文件生成)生成消息对象文件。

如果 iteminfo.po 文件中所有的 msgstr 都是空，那么将不会生成任何消息对象文件，如：

```
$ msgfmt –v iteminfo.po
0 translated messages, 7 untranslated messages.
```

我们已经创建了一条消息，所以会生成 messages.mo 文件。

```
$ msgfmt -v iteminfo.po
1 translated message, 6 untranslated messages.
$ ls -l messages.mo
-rw-r--r-- 1 root root 89 Apr 11 12:11 messages.mo
```

把 message.mo 文件复制到消息目录下，消息目录应该在当前目录下创建

```
$ mkdir -p en_US/LC_MESSAGES

$ mv messages.mo en_US/LC_MESSAGES/item.mo
```

注意：目标文件的名字要和初始 awk 文件中的 TEXTDOAMIN 后面的值相同。之前的 awk 文件中 TEXTDOMAIN=”item”。

步骤 4：核证消息

现在可以看到，awk 将不再显示”START_TIME”,而是转换为”Report Generated On:”然后打印出来：

```
$ gawk -f iteminfo.awk items.txt
Reprot Generated On:Thu Apr 11 12:16:19 CST 2013
Num         Description         Type                    Price      Qty
-------------------------------------------------------
101         HD Camcorder        Video                   210.00     010
102         Refrigerator        Appliance               850.00     002
103         MP3 Player          Audio                   270.00     015
104         Tennis Racket       Sports                  190.00     020
105         Laser Printer       Office                  475.00     005
```

98. 双向管道

awk 可以使用”|&”和外部进程通信，这个过程是双向的。

下面的例子把关键字”Awk”替换为”Sed and Awk”。

```
$ echo "Awk is great" | sed 's/Awk/Sed and Awk/'
Sed and Awk is great
```

下面的例子使用”|&”了模拟实现上面的例子，以说明 awk 是如何双向管道的。

```
$ cat two-way.awk
BEGIN {
   command = "sed 's/Awk/Sed and Awk/'"
   print "Awk is Great!" |& command
   close(command,"to");
   command |& getline tmp
   print tmp;
   close(command);
}
$ awk -f two-way.awk
Sed and Awk is Great!
```

这个例子中：

- command = “sed ‘s/Awk/Sed and Awk/’” –这是要和 awk 双向管道对接的命令。它是一个简单的 sed 替换命令，把”Awk”替换为”Sed and Awk”。
- print “Awk is Great!” |& command – command 的输入，即 sed 替换命令的输入是”Awk is Great!”。”|&”表示这里是双向管道。”|&”右边命令的输入来自左边命令的输出。
- close(command,”to”) – 一旦命令执行完成，应该关闭”to”进程。
- command |& getline tmp –既然命令已经执行完成，就要用 getline 获取其输出。前面命令的输出会被存在变量”tmp”中。
- print tmp –打印输出
- close(command) –最后，关闭命令。

双向管道迟早会派上用场，尤其是当 awk 对外部程序的输出依赖比较大时。

99. 系统函数

可以使用操作系统内置的函数来执行操作系统命令，但请注意，调用系统命令和使用双向管道是不同的。

使用”|&”时，可以把任意 awk 命令的输出作为外部命令的输入；也可以接收外部命令的输出作为 awk 的输入(要不怎么叫双向管道呢)。

执行系统命令时，可以传递任意的字符串作为命令的参数，它会被当做操作系统命令准确第执行，并返回结果(这和双向管道有所不同)。

下面的例子在 awk 中调用 pwd 和 date 命令：

```
$ awk 'BEGIN { system("pwd") }'
/root
$ awk 'BEGIN { system("date") }'
Wed Apr 10 17:07:08 CST 2013
```

在执行比较大的 awk 脚本时，你或许想在脚本开始和结束时发送一封电子邮件。下面的例子展示如何在 BEGIN 和 END 区域中调用系统命令，在脚本开始和结束时发送邮件。

```
$ cat system.awk
BEGIN {
   system("echo 'Started' | mail -s 'Program system.awk started ..' ramesh@thegeekstuff.com");
}
{
   split($2,quantity,",");
   total=0;
   for (x in quantity)
   total=total+quantity[x];
   print "Item",$1,":",total,"quantities sold";
}

END {
   system("echo 'Completed' | mail -s 'Program system.awk completed..'
ramesh@thegeekstuff.com");
}

$ awk -f system.awk items-sold.txt
Item 101 : 2 quantities sold
Item 102 : 0 quantities sold
Item 103 : 10 quantities sold
Item 104 : 2 quantities sold
Item 105 : 10 quantities sold
```

100. 时间函数

这些命令仅适用于 GAWK。

看下面的例子，systime()函数返回系统的 POSIX 时间，即自 1970 年 1 月 1 日起至今经过的秒数。

```
$ awk 'BEGIN { print systime() }'
1365585325
```

如果使用 strftime 函数把 POSIX 时间转换为可读的格式，systime 函数就变动很有用了。

下面例子使用 systime 和 strftime 以可读的格式打印当前时间。

```
$ awk 'BEGIN { print strftime("%c",systime()) }'
Wed Apr 10 17:17:35 2013
```

下面的例子展示了多种可用的时间格式。

```
$ cat strftime.awk
BEGIN {
   print "--- basic formats ---"
   print strftime("Format 1: %m/%d/%Y %H:%M:%S",systime())
   print strftime("Format 2: %m/%d/%y %I:%M:%S %p",systime())
   print strftime("Format 3: %m-%b-%Y %H:%M:%S",systime())
   print strftime("Format 4: %m-%b-%Y %H:%M:%S %Z",systime())
   print strftime("Format 5: %a %b %d %H:%M:%S %Z %Y",systime())
   print strftime("Format 6: %A %B %d %H:%M:%S %Z %Y",systime())
   print "--- quick formats ---"
   print strftime("Format 7: %c",systime())
   print strftime("Foramt 8: %D",systime())
   print strftime("Format 9: %F",systime())
   print strftime("Format 10: %x",systime())
   print strftime("Format 11: %X",systime())
   print "--- single line format with %t ---"
   print strftime("%Y %t%B %t%d",systime())
   print "--- multi line format with %n ---"
   print strftime("%Y%n%B%n%d",systime())
}
$ awk -f strftime.awk
--- basic formats ---
Format 1: 04/10/2013 18:04:06
Format 2: 04/10/13 06:04:06 PM
Format 3: 04-Apr-2013 18:04:06
Format 4: 04-Apr-2013 18:04:06 CST
```

Format 5: Wed Apr 10 18:04:06 CST 2013
Format 6: Wednesday April 10 18:04:06 CST 2013
--- quick formats ---
Format 7: Wed Apr 10 18:04:06 2013
Foramt 8: 04/10/13
Format 9: 2013-04-10
Format 10: 04/10/13
Format 11: 18:04:06
--- single line format with %t ---
2013 April 10
--- multi line format with %n ---
2013
April
10

下面是 strftime 函数可用的时间格式标识符，需要注意的是，下面所有的标识符依赖于本地系统的设置，下面的示例是打印英文时间。

格式标识符	描述
%m	两位数字月份，一月显示为 01
%b	月份缩写，一月显示为 Jan
%B	月份完整单词，一月显示为 January
%d	两位数字日期，4 号显示为 04
%Y	年份的完整格式，如 2011
%y	两位数字的年份，如 2011 显示为 11
%H	24 小时格式， 1 p.m 显示为 13
%I	12 小时格式， 1 p.m 显示为 01
%p	显示 AM 或 PM，和%I 搭配使用
%M	两位数字分钟，9 分显示为 09
%S	两位数字描述，5 秒显示为 05
%a	三位字符星期，周一显示为 Mon
%A	完整的日期，周一显示为 Monday
%Z	时区，太平洋地区时区显示为 PST
%n	换行符
%t	制表符

组合时间格式：

格式标识符	描述
%c	**显示本地时间的完整格式，如 Fri 11 Feb 2011 02:45:03 AM PST**
%D	**简单日期格式，和%m/%d/%y 相同**
%F	**简单日期格式，和%Y-%m-%d 相同**
%T	**简单时间格式，和%H:%M:%S 相同**

%x	基于本地设置的时间格式
%X	基于本地设置的时间格式

101. getline 命令

如你所知，没从输入文件读取一行，body 区域的代码就会执行一次。你无法干预，awk 自动执行这个过程。

然而使用 geline 命令可以控制 awk 从输入文件(或其他文件)读取数据。注意，一旦 getline 执行完成，awk 脚本会重置 NF,NR,FNR 和$0 等内置变量。

getline 示例

```
$ awk -F"," '{getline;print $0;}' items.txt
102,Refrigerator,Appliance,850,2
104,Tennis Racket,Sports,190,20
105,Laser Printer,Office,475,5
```

当在 body 区域使用了 getline 时，会直接读取下一行数据。这个例子中，第一条语句便是 getline，所以即使 awk 已经读取了第一行数据，getline 也会继续读取下一行，因为我们强制它读取下一行，因此，getline 后面的 print $0 会打印输入文件的第二行。

这个例子中：

- 开始执行 body 区域时，执行任何命令之前，awk 从 items.txt 文件中读取第一行数据，保存在变量$0 中
- getline – 我们用 getline 命令强制 awk 读取下一行数据，保存在变量$0 中(之前的内容被覆盖掉了)
- print $0 –既然现在$0 中保存的是第二行数据，print $0 会打印文件第二行(而不是第一行)
- body 区域继续执行，只打印偶数行的数据。(注意到最后一行 105 也打印了么？)

把 getline 的内容保存的变量中

除了把 getline 的内容放到$0 中，还可以把它保存在变量中。

只打印奇数行内容

```
$ awk -F"," '{getline tmp; print $0;}' items.txt
101,HD Camcorder,Video,210,10
103,MP3 Player,Audio,270,15
105,Laser Printer,Office,475,5
```

这个例子如何工作：

- 开始执行 body 区域时，执行任何命令之前，awk 从 items.txt 文件中读取第一行数据，保存在变量$0 中
- getline tmp – 强制 awk 读取下一行，并保存在变量 tmp 中
- print $0 – 此时$0 仍然是第一行数据，因为 getline tmp 没有覆盖$0,因此会打印第

一行数据(而不是第二行)

- body 区域继续执行，只打印奇数行的数据。

下面的例子同时打印$0 和 tmp，可以看到，$0 是奇数行，而 tmp 是偶数行

```
$ awk -F"," '{getline tmp; print "$0->",$0;print "tmp->",tmp;}' items.txt
$0-> 101,HD Camcorder,Video,210,10
tmp-> 102,Refrigerator,Appliance,850,2
$0-> 103,MP3 Player,Audio,270,15
tmp-> 104,Tennis Racket,Sports,190,20
$0-> 105,Laser Printer,Office,475,5
tmp-> 104,Tennis Racket,Sports,190,20
```

从其他的文件 getline 内容

上面两个例子，getline 都是从当前输入文件获取内容，其实也可以从其他文件(非当前输入文件)读取内容，如下所示。

在两个文件中循环切换，打印所有内容。

```
$ awk -F"," '{print $0;getline <"items-sold.txt"; print $0;}' items.txt
101,HD Camcorder,Video,210,10
101 2 10 5 8 10 12
102,Refrigerator,Appliance,850,2
102 0 1 4 3 0 2
103,MP3 Player,Audio,270,15
103 10 6 11 20 5 13
104,Tennis Racket,Sports,190,20
104 2 3 4 0 6 5
105,Laser Printer,Office,475,5
105 10 2 5 7 12 6
```

这个例子中：

- 开始执行 body 区域时，执行任何命令之前，awk 从 items.txt 文件中读取第一行数据，保存在变量$0 中
- print $0 – 打印 items.txt 文件的第一行
- getline <"items-sold.txt" – 读取 items-sold.txt 中第一行并保存在$0 中
- print $0 – 打印 items-sold.txt 文件中的第一行
- body 区域继续执行，轮番打印 items.txt 和 items-sold.txt 的剩余内容

从其他的文件 getline 内容到变量中

除了把两个文件的内容都读到$0 中之外，也可以使用"getline var"把读取的内容保存到变量中。

继续使用上面那两个文件，打印所有内容(使用 tmp 变量)

```
$ awk -F"," '{print $0; getline tmp < "items-sold.txt";print tmp;}' items.txt
101,HD Camcorder,Video,210,10
101 2 10 5 8 10 12
```

```
102,Refrigerator,Appliance,850,2
102 0 1 4 3 0 2
103,MP3 Player,Audio,270,15
103 10 6 11 20 5 13
104,Tennis Racket,Sports,190,20
104 2 3 4 0 6 5
105,Laser Printer,Office,475,5
105 10 2 5 7 12 6
```

除了把第二个文件的内容保存到变量之外，这个例子和之前的例子是相同的。

getline 执行外部命令

getline 也可以执行 UNIX 命令并获取其输出。

下面的例子使用 getline 获取 date 命令的输出并打印出来。请注意这里也要使用 close 刚执行的命令，date 命令的输出保存在变量$0 中，如下所示。

使用这个方法可以在输出报文的 header 和 footer 中显示时间戳。

```
$ cat getline1.awk
BEGIN {
   FS=",";
   "date" | getline
   close("date")
   print "Timestamp:" $0
}
{
   if ( $5 <= 5)
   print "Buy More:Order",$2,"immediately!"
   else
   print "Sell More:Give discount on",$2,"immediatelty!"
}

$ awk -f getline1.awk items.txt
Timestamp:Thu Apr 11 11:07:47 CST 2013
Sell More:Give discount on HD Camcorder immediatelty!
Buy More:Order Refrigerator immediately!
Sell More:Give discount on MP3 Player immediatelty!
Sell More:Give discount on Tennis Racket immediatelty!
Buy More:Order Laser Printer immediately!
```

除了把命令输出保存在$0 中之外，也可以把它保存在任意的 awk 变量中(如 timestamp),如下所示。

```
$ cat getline2.awk
BEGIN {
```

```
    FS=",";
    "date" | getline timestamp
    close("date")
    print "Timestamp:" timestamp
}
{
    if ( $5 <= 5)
    print "Buy More: Order",$2,"immediately!"
    else
    print "Sell More: Give discount on",$2,"immediately!"
}
$ awk -f getline2.awk items.txt
Timestamp:Thu Apr 11 11:20:05 CST 2013
Sell More: Give discount on HD Camcorder immediately!
Buy More: Order Refrigerator immediately!
Sell More: Give discount on MP3 Player immediately!
Sell More: Give discount on Tennis Racket immediately!
Buy More: Order Laser Printer immediately!
```

Knowledge is not one man’s Entity

Book Hacked by VELOCIRAPTOR