SED 与 AWK 学习笔记

张逸群 jeantoe@gmail.com

Blog www.zhangyiqun.cn

前言

开卷有益。

在 it 领域有很多学习的途径,个人认为最好的学习教材是原版图书，但由于语言上的问题，大多数人只能望书兴叹。这 39 页的文档是根据我自学时的实际情况，和大多数的实际需要情况所写。

其中示例大部分参考了 O'Reilly.SED and AWK 。

每篇文章中所带的小练习或来自工作或来自网络，希望读者能够多加练习，实践才是王道。

文档完成于 2009-1-31

基本概念

Sed 是一个“非交互式”的**面向字符流的编辑器**。
这个面向字符流就像是我去餐厅点了一道菜让厨师做，菜都是字符，厨师做好后直接送到我面前。所以我只要下达命令就好。

Sed 的优点是可以在一个地方指定所有的编辑指令，然后通过文件传递一次来执行他们。但是它在每次多于一行的处理能力方面有限制。
Awk 的典型应用是将数据转换成格式化的报表。增强可读性。因此当数据有某种结构时就能最好的体现 awk 的好处。Awk 的功能是非常强大的，甚至可以说成是程序设计语言。

基本操作

框架：
命令 选项 工作内容 文件名
Sed 和 awk 的输出都不允许送到向程序提供输入的同一个文件，否则会使它变成乱码。如果工作内容中包含 shell 可执行的字符如“$和*”，那么必须用单引号引起。
Sed 和 awk 都可以用-f 来指定工作内容，这通常就是脚本存放的位置。
使用过程中，短的脚本可以在命令行上指定，长的脚本通常放在容易被修改和测试的文件中。

在 sed 和 awk 中，每个指令都包含两个部分，**模式和语句。模式是由/分隔的正则**。语句指定一个或多个将被执行的动作。
Awk 不自动输出行，脚本中的指令控制 awk 最终所做的事情。
Sed 的语句由类似于行编辑器中使用的那些编辑命令组成。大部分命令由单个字母组成。
Awk 的语句由程序设计语句和函数组成，语句必须用大括号括起。

初识 sed

最常见的 s 。替换字符串。
```
$ sed 's/MA/Massachusetts/' list
```
找出 MA 并替换成 Massachusetts
```
John Daggett, 341 King Road, Plymouth Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston Massachusetts
```
并不是在任何情况下都要将指令用单引号包围起来，但你应该养成这个习惯。
在城市和州之间放置逗号，这时候就需要单引号。
```
$ sed 's/ MA/, Massachusetts/' list
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```
如果不加单引号，那么会得到一个错误提示 sed: -e expression #1, char 2: unterminated `s' command
关闭自动输出，只打印被修改行。这里用了参数-n(关闭自动输出)和 p(打印被修改行)
```
$ sed -n -e 's/MA/Massachusetts/p' list
John Daggett, 341 King Road, Plymouth Massachusetts
Eric Adams, 20 Post Road, Sudbury Massachusetts
Sal Carpenter, 73 6th Street, Boston Massachusetts
```

在命令行上编写多个语句。
使用;分隔
```
sed 's/ MA/, Massachusetts/; s/ PA/, Pennsylvania/' list
```
使用-e
```
sed -e 's/ MA/, Massachusetts/' -e 's/ PA/, Pennsylvania/' list
```

初识 awk

为了能和 shell 区分开，**awk 的指令都必须包括单引号**，因为$这类符号在 shell 中是有特殊意义的。虽然 awk 与 sed 指令的结构相同，但 awk 中用语句和函数取代了使用一个或两个字符组成的命令。

Awk 将每个输入行识别成一条记录，而将那一行上的每个单词识别成一个字段。

```
$ awk '{ print $1 }' list
John
Alice
Orville
Terry
Eric
Hubert
Amy
Sal
```

打印含有 MA 的行

```
$ awk '/MA/' list
John Daggett, 341 King Road, Plymouth MA
Eric Adams, 20 Post Road, Sudbury MA
Sal Carpenter, 73 6th Street, Boston MA
```

打印含有 MA 的行的第一个字段

```
$ awk '/MA/ { print $1 }' list
John
Eric
Sal
```

使用-F 指定字段分隔符为逗号。意思是说逗号前的字段是$1 或者$其他。这就使得原来可能 $1 $2 的内容都合并成了$1。

```
$ awk -F, '{ print $1; print $2; print $3 }' list
John Daggett
 341 King Road
 Plymouth MA
Alice Ford
 22 East Broadway
 Richmond VA
Orville Thomas
 11345 Oak Bridge Road
 Tulsa OK
Terry Kalkas
 402 Lans Road
 Beaver Falls PA
Eric Adams
 20 Post Road
```

```
 Sudbury MA
Hubert Sims
 328A Brook Road
 Roanoke VA
Amy Wilde
 334 Bayshore Pkwy
 Mountain View CA
Sal Carpenter
 73 6th Street
 Boston MA
```

新手常见错误

没有用大括号{}。没有用单引号 ’’。没有用斜杠将正则括起来//。

正则表达式

引语

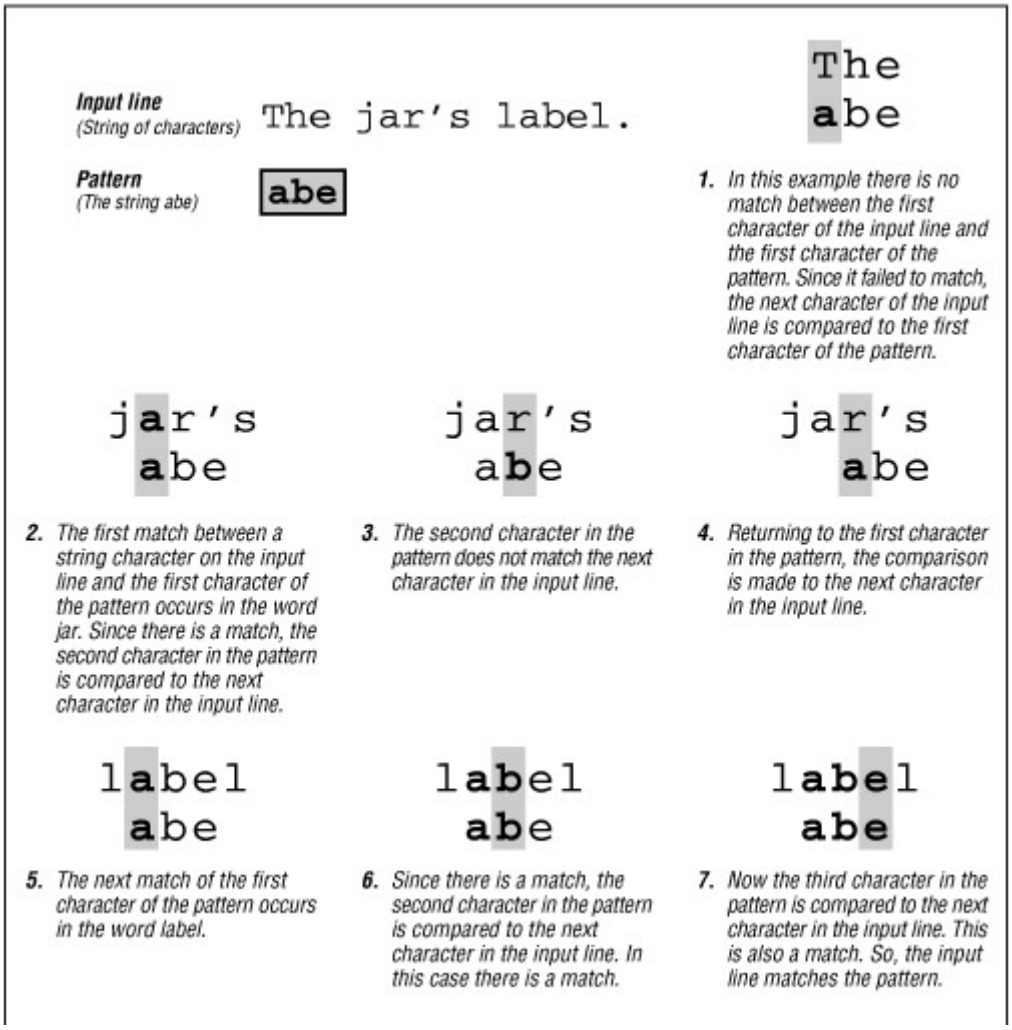
Grep、sed、awk 都使用正则，**然而这 3 个程序并不能完全使用正则表达式语法中的所有元字符。**所谓**元字符就是指那些在正则表达式中具有特殊意义的专用字符**，可以用来规定其前导字符（即位于元字符前面的字符）在目标对象中的出现形式。

注：本篇十分简陋，想深入学习正则可以找一本《精通正则表达式》。

表达式

表达式告诉计算机如何产生结果。但和加减乘除一样，在式子中也存在着优先级的问题。需要注意的一点是**正则是区分大小写的**。

那么正则是如何工作的呢，请看下图。



图中描述 abe 是如何进行匹配的。

元字符汇总

.

```
[root@localhost]# grep --color -n 'r..t' /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
12:operator:x:11:0:operator:/root:/sbin/nologin
15:ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

大部分情况下这个.**匹配除换行符外的任意一个字符。但在 awk 中也能匹配换行符。**

这里使用--color 可以非常方便地查看 grep 具体匹配了哪些字符串，使用-n 标出行数。

*

```
[root@test root]# grep -n 'ooo*' regular_express.txt
1:"Open Source" is a good mechanism to develop programs.
2:apple is my favorite food.
3:Football game is not use feet only.
9:Oh! The soup taste good.
18:google is the best tools for search keyword.
19:goooooogle yes!
```

这个例子来自鸟哥的网页，表达式为了匹配至少两个 o 以上的字符串。从本例应该能够理解*是匹配任意一个在它前面的字符。留一个思考题， *ooo 与 o.*o 匹配的结果有什么不同？

[...]

中的

\转义（awk 中使用）

-表示范围。例如[0-9]包含任何数字

^取反。例如[^0-9]不匹配数字。注意在 grep 中是使用 grep –v 来取反

```
grep '^[0-9]' /etc/inittab          查出以数字开头的行
grep '[a-Z]' /etc/inittab          查出有字母（大小通用）的行
grep '^[^0-9]'     /etc/inittab     ^[^ ]取反的
grep 'x:[0-9][0-9]:' /etc/passwd          取出 UID 为 0－99 的用户
ls -l /dev |grep 'tty[0-9]*$'      （＊多个或 0 个前面的字符）
```

^

^ n – 以 n 开头的行

$

从行尾开始任务。

注意

在 sed 和 grep 中^和$并不一定总保持着自己的个性。当使用 ab^c 或者是 ab$c 匹配时它们就确实代表着字面意思，没有任何别的含义。但在 awk 中则不同，^和$永远保持着自己的个性，所以在 awk 中如果要匹配它俩时都需要使用\来转义。

\{n,m\}

```
101
1001
10001
100001
10000001
100000001
```

这时候想把 1001，10001，100001 过滤出来。

```
# grep "10\{2,4\}1" file
1001
10001
100001
```

1001 中有 2 个 0，100001 中有 4 个 0，所以这个大家应该明白了吧？

\

转义符。取消字符的特殊效果。比如我要匹配文件中的 5.6。这时候我需要这样写

```
# grep '5\.6' file
5.6
```

如果写成

```
grep '5.6' file
5.6
506
```

则列出了 5.6 和 506。

拓展的元字符（egrep 和 awk）

+

```
egrep 'o+'     /etc/passwd
```

?

| 或运算

egrep '^mysql|^root' /etc/passwd

egrep '^(mysql|root)' /etc/passwd

指定单词的单复数 compan(y|ies)

()

用于对正则进行分组并设置优先级

基础

ls -l /dev |grep '^b'

grep -v '^#' /etc/httpd/conf/httpd.conf (-v 取反)

grep 'bash$' /etc/passwd 找出可以登录的用户

grep -c 'bash$' /etc/passwd (一c 统计行数)

ls -lR /etc/ |grep 'conf$' (查找/etc/下的以 conf 结尾的文件)

找出 httpd.conf 文件的有效行

grep -v '^$' /etc/httpd/conf/httpd.conf |grep -v '^#'

单词分割符：[^a-Z] +

在 vim 编辑器下使用正则：

1,$s/^\([a-z] [a-z]*\).*/\1/ 显示第一个单词

1,$s/^\([a-z] [a-z]*\)\([^a-z].*[^a-z]\)\([a-z]*[a-z]\)$/\3\2\1/

第一个单词与最后一个单词互换

1,$s/[0-9]//g 把数字替换为空 g 全局

1,$s#:/[a-z] [a-z]*/[a-z] [a-z]*$## 删掉 passwd 后面的字段

1,$s#.*:## 只留路径

练习

找出 passwd 文件中的 uid 为三位数的能登录的用户

```
grep 'x:[0-9][0-9][0-9]*:..*bash$' /etc/passwd
```

匹配否定句

```
I can do it
I cannot do it
I can not do it
I can't do it
I cant do it
```

答案

```
# grep "can[n 'o]*t" file
```

匹配单词 book

```
This file tests for book in various places, such as
book at the beginning of a line or
at the end of a line book
as well as the plural books and
handbooks.  Here are some
phrases that use the word in different ways:
"book of the year award"
to look for a line with the word "book"
A GREAT book!
```

```
A great book? No.
told them about (the books) until it
Here are the books that you requested
Yes, it is a good book for children
amazing that it was called a "harmful book" when
once you get to the end of the book, you can't believe
A well-written regular expression should
avoid matching unrelated words,
such as booky (is that a word?)
and bookish and
bookworm and so on.
```

答案

```
# egrep -n "^book |[ \"]book[ \!\?\"\,s)]| book$" bookwords
```

匹配结尾处有一个或多个空格的行

被括起来的是空格( )*$

统计空格数

```
$ grep -c '^$' ch04
```

匹配空行^ *$

匹配整个行^.*$

英文中的常用标点? . , ! ; : '

详解 sed

引语

本篇主要是让大家对 sed 的脚本编写有一个整体的了解，在大脑中能有个框架。

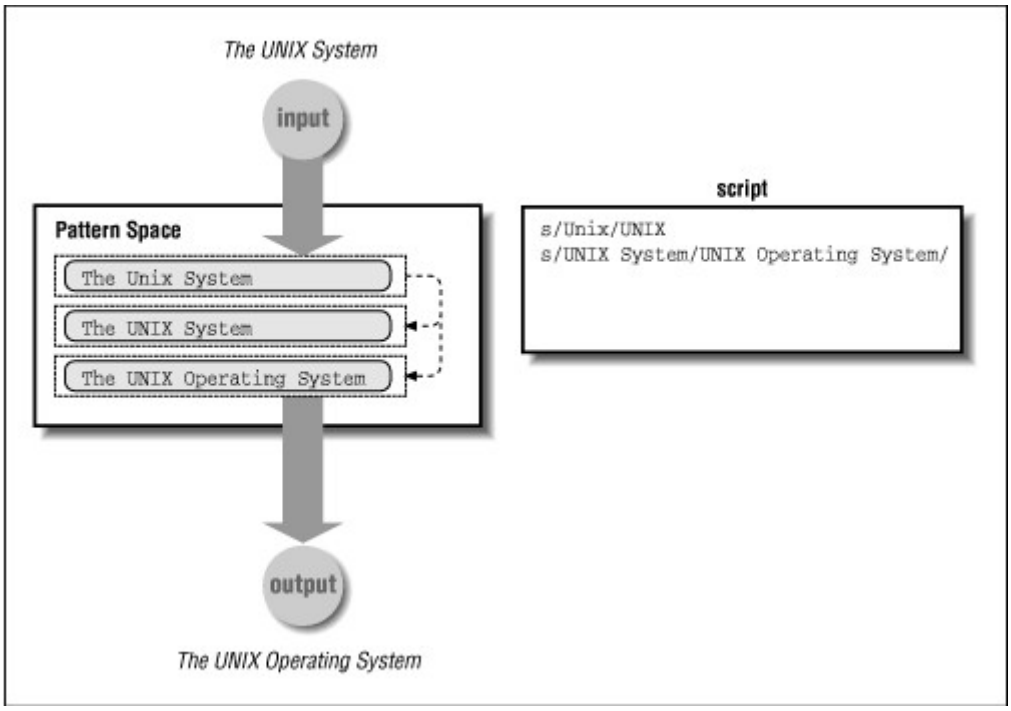
写脚本前一定要

1．具体的分析清楚自己想做什么

2．明确处理的过程

3．在应用于生产环境前要反复测试

下面图中的示例是 sed 如何匹配字符串并替换



从图中可以看出 sed 首先将整个编辑脚本应用于第一个输入行，然后在读取第二个输入行并对其应用整个脚本。这种做法的优点显而易见，跟少食多餐一个性质，一顿饭吃一百个馒头恐怕你的胃就爆了，这就是内存溢出。

Sed 脚本的 3 种用途

1 对同一文件的编辑。热身，包含了以前章节中的知识点。

```
                    HORSEFEATHERS SOFTWARE PRODUCT BULLETIN

  DESCRIPTION
+   ___________

  BigOne Computer  offers three  software packages from the  suite
  of Horsefeathers  software products  --  Horsefeathers  Business
  BASIC, BASIC  Librarian,  and LIDO.  These software products can
  fill  your    requirements    for    powerful,    sophisticated,
  general-purpose business  software providing you with a base for
  software customization or development.

  Horsefeathers  BASIC is  BASIC optimized for use on  the  BigOne
  machine with UNIX  or MS-DOS operating systems.  BASIC Librarian
  is a full screen program editor, which also provides the ability
```

要求

1 用 sea 取代所有空行

这个不用多说，在上次的文章中已经写过

```
s/^$/sea/g
```

2 删除每行前面的空格

```
s/^  *//g
```

3 删除+后的___

```
/^+  *___*/d
```
4 删除在两个单词之间的多个空格
```
s/  */ /g
```
5 保留.号后的多个空格
```
s/\. */\.  /g
```
6 干掉,号后的空格
```
s/\,  */\,/g
```
2 改变一组文件。

类似于这样
```
  sed -i -e '74 s/^/#/' -i -e '76 s/^/#/' $ssh_cf
  sed -i "s/#UseDNS yes/UseDNS no/" $ssh_cf
  sed -i -e '44 s/^/#/' -i -e '48 s/^/#/' $ssh_cf
  sed -i '/expose_php/s/On/Off/' $fcgi_cf
  sed -i '/display_errors/s/On/Off/' $fcgi_cf
  sed       -i       's#extension_dir       =       "./"#extension_dir       =
"/usr/local/php-fcgi/lib/php/extensions/no-debug-non-zts-20060613/"\nextension
```
这样的脚本可以节省大量的时间，但如果写的不好造成的错误恐怕会让你花更多的时间去解决。所以制作这种脚本时最好多做测试以免以后出乱子。

注：这种脚本通常出现在安装程序脚本的制作上。如果经常看文档，应该对此不会陌生。

3 提取文件的内容

例：提取 C 源文件中的 main() 函数
```
$ sed -n -e '/main[[:space:]]*(/,/^}/p' sourcefile.c | more
```
注：这里的[[:space:]] 只是一个特殊的关键字，它告诉 sed 与 TAB 或空格匹配。

基本 sed 命令

想要用好 sed 首先就要了解它的框架，知道每一段都是干什么的。

框架

[*address*]**s**/*pattern*/*replacement*/*flags*

Flags 可以看做是古代打仗作战时的令旗。

Flags 段

以下是该段可能出现的参数

n 1-512 间的一个数字，指定对第 n 次出现的情况进行替换

g 全局替换

p 打印

w 将样本写入到某个文件中

替换指令应用于与 address 匹配的行。如果没有指定地址，那么就应用于与 pattern 匹配的所有行。

Flag 段中的指令和组合使用。

replacement 段

& 表示用正则表达式匹配的内容进行替换

这东西以前我非常难以理解，因为所看的大部分中文文档对此都没有过详述。

比如一篇文章里面有很多关键字如：社会主义。现在我想在每个社会主义后面都加一个好并且用括号括起来。

s/社会主义/(&好）/g

现在大家应该明白了它的意思吧？（生产环境估计是没有中文，这里主要是为了大家好理解）

\n 匹配第 n 个子串（n 是一个数字），这个子串需要在 pattern 中用“ \(\) ”包围。
例现在文章中的关键字有司马光，小李，猴子。现在想变成司马光砸缸子，小李飞刀，猴子偷桃。
s/\(司马光\)\(小李\)\(猴子\)/\1 砸缸子\2 飞刀\3 偷桃/
\ 转义特殊符号。在相关文档中提到\也可以当作换行符来用。
注：The backslash is generally used to escape the other metacharacters but it is also used to include a newline in a replacement string.

追加，插入和更改

这些命令的语法在 sed 中不常用，因为它们必须在多行上来指定。语法如下
追加 [*line-address*]a\
text
插入 [*line-address*]i\
text
修改 [*address*]c\
text

替换文件中不可正常输出的字符

曾经使用过 man 查看文档，并把文档重定向的朋友一定知道用 firefox 或者 gedit 这类工具打开重定向的文档时会出现很多小方块。那些小方块是用来调节格式的，而 ff 和 gedit 不认识而已。
查看不可见字符。如换行。
Sed －n －e “ l ” file
这时便可以看见了。针对文档拿掉那些符号即可。

读和写文件

语法如下
[*line-address*]r *file*
[*address*]w *file*
非常实用的功能。
例如在代码中结尾处有;的地方给予提示。
代码如下

```
/;$/r file
```

执行效果

```
sed -f scr index.php
<?php
/**
 * Front to the WordPress application. This file doesn't do anything, but loads
 * wp-blog-header.php which does and tells WordPress to load the theme.
 *
 * @package WordPress
 */

/**
 * Tells WordPress to load the WordPress theme and output it.
 *
```

```
 * @var bool
 */
define('WP_USE_THEMES', true);
this is (;)

/** Loads the WordPress Environment and Template */
require('./wp-blog-header.php');
this is (;)
```

好，现在来看看如何写文件
有一份人员清单，里面包括必要的信息，现在按地理位置进行划分。
清单如下

```
Adams, Henrietta        Northeast
Banks, Freda            South
Dennis, Jim             Midwest
Garvey, Bill            Northeast
Jeffries, Jane          West
Madison, Sylvia         Midwest
Sommes, Tom           South
```

代码如下

```
/Northeast$/{
s///
w region.northeast
}
/South$/{
s///
w region.south
}
/Midwest$/{
s///
w region.midwest
}
/West$/{
s///
w region.west
}
```

字符串的转换

将 abc 中的每个字符都转换成 xyz 中的等价字符。

```
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

打印行号

我想知道 require 都在哪几行出现，实现这个功能需要用到=
代码如下

```
/require/{
=
```

p
}
执行效果
sed -n -f scr index.php
17
require('./wp-blog-header.php');
打印 n 行后退出
Sed '10q' file
打印 10 行后退出。

Sed 中的高级命令

引语
在写这部分前，我对那些命令可以说是没有丝毫了解的，只能看着 e 文档死磕，每个命令都花费不少时间来理解，如果有错误请一定提出。
最后引用作者的一句话 You can consider yourself a true sed-master once you understand the commands presented here.当你把这些命令玩明白，你就可以说精通 sed 了。

改变不连贯的语句

本例中要用的是 N 命令。N 通过读取新的输入行，并将它添加到 pattern 段的现有内容之后来创建多行 replacement 空间。通过语言实在很难描述清楚，大家来看底下的例子
Consult Section 3.1 in the Owner and Operator
Guide for a description of the tape drives
available on your system.

Look in the Owner and Operator Guide shipped with your system.

Two manuals are provided including the Owner and
Operator Guide and the User Guide.

The Owner and Operator Guide is shipped with your system.
在这篇文中中我想把 Owner and Operator Guide 都变为 just do it 。这里我用橙色标出了最难解决的部分。N 就是用来解决他们的。
现在来编写脚本。
s/Owner and Operator Guide/just do it
用这行语句先把比较简单的两行解决掉。
/Owner/{
N
s/Owner and Operator\nGuide/just do it\
找到 Owner and Operator 就给我结束，然后去下一行找 Guide
/
}
注：\n 是换行符，在这里起到分隔的作用。
实例中 Owner and Operator 共出现 4 次，所以下面我将用 1，2，3，4 来代替，这样也比较清楚。

重新执行脚本。可以发现 1，2，4 都已经被替换了，但是 3 还是没有变化，问题出在哪里呢？

```
s/Owner and Operator\nGuide/just do it\
```

对了，就是红色的部分。那如何解决呢？用或来解决

```
s/((Owner and Operator|Owner and)|(\nGuide|\nOperator Guide))/just do it\
```

实验后发现并没有效果，所以只能把语句分开写了，如果哪位朋友用一句话实现了记得告诉一下。

全部代码如下

```
s/Owner and Operator Guide/just do it/
/Owner/{
N
s/Owner and Operator\nGuide/just do it\
/
}
/Owner and/{
N
s/Owner and\nOperator Guide/just do it\
/
}
```

后来又把代码改良了一下

```
s/Owner and Operator Guide/just do it/
/Owner/{
N
s/\n/ /
```

精华就在这里了，哈哈，实际上是偷梁换柱把换行符给弄成空格了。

```
s/Owner and Operator Guide/just do it\
/
}
```

匪夷所思的删除

文章中的空行不合理，现在进行调整使其合理。这次要用到的指令是 D

文章如下

```
This line is followed by 1 blank line.

This line is followed by 2 blank lines.


This line is followed by 3 blank lines.



This line is followed by 4 blank lines.
```

This is the end.
我想得到的效果是
```
This line is followed by 1 blank line.

This line is followed by 2 blank lines.

This line is followed by 3 blank lines.

This line is followed by 4 blank lines.

This is the end.
```
代码如下
```
/^$/{
N
/^\n$/d
}
```
在执行之前希望各位能仔细想想该脚本的含义是什么。
执行后效果如下
```
# sed -f scr sample
This line is followed by 1 blank line.

This line is followed by 2 blank lines.
This line is followed by 3 blank lines.

This line is followed by 4 blank lines.
This is the end.
```
好，现在我来分析一下产生这种效果的原因。让我们先来看一下 sample 文档吧
```
# sed -n -e "l" sample
This line is followed by 1 blank line.$
$
This line is followed by 2 blank lines.$
$
$
This line is followed by 3 blank lines.$
$
$
$
This line is followed by 4 blank lines.$
$
$
$
$
This is the end.$
```

在代码中我用^$匹配空行，并使用了 N 把下面出现的空行也匹配上（通俗点讲 N 的作用就是把下一行拉上来看是不是空行），然后使用 d 来清除。
为了方便说明我将 5 句话按从上到下的顺序分为 1，2，3，4，5.
代码在 1，2 匹配时条件不成立，因为下一行是 2 不为空，所以 N 不成立，于是跳。
到了 2，3 行后条件匹配，这里我们很清晰的看到有 3 个空行，条件成立！于是 2 个空行被干掉了。
现在到了 3，4 之间又有两个空行被干掉了，这时的情况和在 1，2 之间的一样，N 不成立于是最后一个空行被放生，3，4 间便有了空行。
现在到了 4，5 之间，我想大家应该能推测出结果就是空行被全部干掉了。
这时，用 D 替代原来 d 的位置，就可以得到理想的效果了。当 D 遇到两个空行时只删除两个空行中的第一个，当一个空行后面跟我有文本时就正常输出。

Sed 补遗

```
h     复制进暂存区，一次只能存一行
    Sed '1,5h' passwd
H     追加到暂存区,可以多行
    Sed '1,5H' passwd
$G     粘贴到最后一行
    Sed '$G'
$g     使用暂存区覆盖最后一行
    Sed -e '1,5h' -e '$g' passwd
按顺序执行 {} 括号里的命令
    Sed '/^root/{ s/^root/blues/;s/bash$/nologin/; }'     passwd
    查找以 root 开头的行把 root 和 bash 替换成 blues 和 nologins
sed -r     使用扩展正则字符集
    sed -r 's/^([a-z]+)([^a-z].*[^a-z])([a-z]+)$/\3\2\1/' passwd
    把 passwd 文件的首末单词互换，并输出到屏幕
    sed -r 's/^([a-zA-Z0-9_]+):.*/\1/' passwd     只输出用户名
```

sed技巧小结

删除行首空格

```
sed 's/^[ ]*//g' filename
sed 's/^ *//g' filename
sed 's/^[[:space:]]*//g' filename
```

1、删除行首空格

```
sed 's/^[ ]*//g' filename
sed 's/^ *//g' filename
sed 's/^[[:space:]]*//g' filename
```

2、行后和行前添加新行

```
行后：sed 's/pattern/&\n/g' filename
行前：sed 's/pattern/\n&/g' filename
```

&代表pattern

3、使用变量替换(使用双引号)

```
sed -e "s/$var1/$var2/g" filename
```

4、在第一行前插入文本

```
sed -i '1 i\插入字符串' filename
```

5、在最后一行插入

```
sed -i '$ a\插入字符串' filename
```

6、在匹配行前插入

```
sed -i '/pattern/ i "插入字符串"' filename
```

7、在匹配行后插入

```
sed -i '/pattern/ a "插入字符串"' filename
```

8、删除文本中空行和空格组成的行以及#号注释的行

```
grep -v ^# filename | sed /^[[:space:]]*$/d | sed /^$/d
```

SED 单行脚本快速参考（Unix 流编辑器）

英文标题：USEFUL ONE-LINE SCRIPTS FOR SED (Unix stream editor)

原标题：HANDY ONE-LINERS FOR SED (Unix stream editor)

整理：Eric Pement - 电邮：pemente[at]northpark[dot]edu 版本 5.5

译者：Joe Hong - 电邮：hq00e[at]126[dot]com

在以下地址可找到本文档的最新（英文）版本：

http://sed.sourceforge.net/sed1line.txt

http://www.pement.org/sed/sed1line.txt

文本间隔：

```
--------

# 在每一行后面增加一空行
sed G

# 将原来的所有空行删除并在每一行后面增加一空行。
# 这样在输出的文本中每一行后面将有且只有一空行。
sed '/^$/d;G'

# 在每一行后面增加两行空行
sed 'G;G'

# 将第一个脚本所产生的所有空行删除（即删除所有偶数行）
sed 'n;d'

# 在匹配式样“ regex ”的行之前插入一空行
```

```
sed '/regex/{x;p;x;}'

# 在匹配式样“ regex ”的行之后插入一空行
sed '/regex/G'

# 在匹配式样“ regex ”的行之前和之后各插入一空行
sed '/regex/{x;p;x;G;}'
```

编号：

```
# 为文件中的每一行进行编号（简单的左对齐方式）。这里使用了“ 制表符 ”
# （tab，见本文末尾关于'\t'的用法的描述）而不是空格来对齐边缘。
sed = filename | sed 'N;s/\n/\t/'

# 对文件中的所有行编号（行号在左，文字右端对齐）。
sed = filename | sed 'N; s/^/     /; s/ *\(.\{6,\}\)\n/\1  /'

# 对文件中的所有行编号，但只显示非空白行的行号。
sed '/./=' filename | sed '/./N; s/\n/ /'

# 计算行数 （模拟 "wc -l"）
sed -n '$='
```

文本转换和替代：

```
# Unix 环境：转换 DOS 的新行符（CR/LF）为 Unix 格式。
sed 's/.$//'                    # 假设所有行以 CR/LF 结束
sed 's/^M$//'                   # 在 bash/tcsh 中，将按 Ctrl-M 改为按 Ctrl-V
sed 's/\x0D$//'                 # ssed、gsed 3.02.80，及更高版本

# Unix 环境：转换 Unix 的新行符（LF）为 DOS 格式。
sed "s/$/`echo -e \\\r`/"       # 在 ksh 下所使用的命令
sed 's/$'"/`echo \\\r`/"        # 在 bash 下所使用的命令
sed "s/$/`echo \\\r`/"          # 在 zsh 下所使用的命令
sed 's/$/\r/'                   # gsed 3.02.80 及更高版本

# DOS 环境：转换 Unix 新行符（LF）为 DOS 格式。
sed "s/$//"                     # 方法 1
sed -n p                        # 方法 2

# DOS 环境：转换 DOS 新行符（CR/LF）为 Unix 格式。
# 下面的脚本只对 UnxUtils sed 4.0.7 及更高版本有效。要识别 UnxUtils 版本的
```

```
#  sed 可以通过其特有的“ --text ”选项。你可以使用帮助选项（“ --help ”）看
# 其中有无一个“ --text ”项以此来判断所使用的是否是 UnxUtils 版本。其它 DOS
# 版本的的 sed 则无法进行这一转换。但可以用“ tr ”来实现这一转换。
sed "s/\r//" infile >outfile     # UnxUtils sed v4.0.7 或更高版本
tr -d \r <infile >outfile        # GNU tr 1.22 或更高版本

# 将每一行前导的“ 空白字符 ”（空格，制表符）删除
# 使之左对齐
sed 's/^[ \t]*//'                # 见本文末尾关于'\t'用法的描述

# 将每一行拖尾的“ 空白字符 ”（空格，制表符）删除
sed 's/[ \t]*$//'                # 见本文末尾关于'\t'用法的描述

# 将每一行中的前导和拖尾的空白字符删除
sed 's/^[ \t]*//;s/[ \t]*$//'

# 在每一行开头处插入 5 个空格（使全文向右移动 5 个字符的位置）
sed 's/^/     /'

# 以 79 个字符为宽度，将所有文本右对齐
sed -e :a -e 's/^.\{1,78\}$/ &/;ta'  # 78 个字符外加最后的一个空格

# 以 79 个字符为宽度，使所有文本居中。在方法 1 中，为了让文本居中每一行的前
# 头和后头都填充了空格。 在方法 2 中，在居中文本的过程中只在文本的前面填充
# 空格，并且最终这些空格将有一半会被删除。此外每一行的后头并未填充空格。
sed  -e :a -e 's/^.\{1,77\}$/ & /;ta'                     # 方法 1
sed  -e :a -e 's/^.\{1,77\}$/ &/;ta' -e 's/\( *\)\1/\1/'  # 方法 2

# 在每一行中查找字串“ foo ”，并将找到的“ foo ”替换为“ bar ”
sed 's/foo/bar/'                 # 只替换每一行中的第一个“ foo ”字串
sed 's/foo/bar/4'                # 只替换每一行中的第四个“ foo ”字串
sed 's/foo/bar/g'                # 将每一行中的所有“ foo ”都换成“ bar ”
sed 's/\(.*\)foo\(.*foo\)/\1bar\2/' # 替换倒数第二个“ foo ”
sed 's/\(.*\)foo/\1bar/'            # 替换最后一个“ foo ”

# 只在行中出现字串“ baz ”的情况下将“ foo ”替换成“ bar ”
sed '/baz/s/foo/bar/g'

# 将“ foo ”替换成“ bar ”，并且只在行中未出现字串“ baz ”的情况下替换
sed '/baz/!s/foo/bar/g'

# 不管是“ scarlet ”“ ruby ”还是“ puce ”，一律换成“ red ”
sed 's/scarlet/red/g;s/ruby/red/g;s/puce/red/g'  #对多数的 sed 都有效
gsed 's/scarlet\|ruby\|puce/red/g'               # 只对 GNU sed 有效
```

```
# 倒置所有行，第一行成为最后一行，依次类推（模拟“ tac ”）。
# 由于某些原因，使用下面命令时 HHsed v1.5 会将文件中的空行删除
sed '1!G;h;$!d'               # 方法 1
sed -n '1!G;h;$p'             # 方法 2

# 将行中的字符逆序排列，第一个字成为最后一字，……（模拟“ rev ”）
sed '/\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//'

# 将每两行连接成一行（类似“ paste ”）
sed '$!N;s/\n/ /'

# 如果当前行以反斜杠“ \ ”结束，则将下一行并到当前行末尾
# 并去掉原来行尾的反斜杠
sed -e :a -e '/\\$/N; s/\\\n//; ta'

# 如果当前行以等号开头，将当前行并到上一行末尾
# 并以单个空格代替原来行头的“ = ”
sed -e :a -e '$!N;s/\n=/ /;ta' -e 'P;D'

# 为数字字串增加逗号分隔符号，将“ 1234567 ”改为“ 1,234,567 ”
gsed ':a;s/\B[0-9]\{3\}\>/,&/;ta'                     # GNU sed
sed -e :a -e 's/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta'  # 其他 sed

# 为带有小数点和负号的数值增加逗号分隔符（GNU sed）
gsed -r ':a;s/(^|[^0-9.])([0-9]+)([0-9]{3})/\1\2,\3/g;ta'

# 在每 5 行后增加一空白行 （在第 5，10，15，20，等行后增加一空白行）
gsed '0~5G'                   # 只对 GNU sed 有效
sed 'n;n;n;n;G;'              # 其他 sed
```

选择性地显示特定行：

```
--------

# 显示文件中的前 10 行 （模拟“ head ”的行为）
sed 10q

# 显示文件中的第一行 （模拟“ head -1 ”命令）
sed q

# 显示文件中的最后 10 行 （模拟“ tail ”）
sed -e :a -e '$q;N;11,$D;ba'

# 显示文件中的最后 2 行（模拟“ tail -2 ”命令）
```

```
sed '$!N;$!D'

# 显示文件中的最后一行（模拟“ tail -1 ”）
sed '$!d'                           # 方法 1
sed -n '$p'                         # 方法 2

# 显示文件中的倒数第二行
sed -e '$!{h;d;}' -e x              # 当文件中只有一行时，输入空行
sed -e '1{$q;}' -e '$!{h;d;}' -e x  # 当文件中只有一行时，显示该行
sed -e '1{$d;}' -e '$!{h;d;}' -e x  # 当文件中只有一行时，不输出

# 只显示匹配正则表达式的行（模拟“ grep ”）
sed -n '/regexp/p'                  # 方法 1
sed '/regexp/!d'                    # 方法 2

# 只显示“ 不 ”匹配正则表达式的行（模拟“ grep -v ”）
sed -n '/regexp/!p'                 # 方法 1，与前面的命令相对应
sed '/regexp/d'                     # 方法 2，类似的语法

# 查找“ regexp ”并将匹配行的上一行显示出来，但并不显示匹配行
sed -n '/regexp/{g;1!p;};h'

# 查找“ regexp ”并将匹配行的下一行显示出来，但并不显示匹配行
sed -n '/regexp/{n;p;}'

# 显示包含“ regexp ”的行及其前后行，并在第一行之前加上“ regexp ”所
# 在行的行号 （类似“ grep -A1 -B1 ”）
sed -n -e '/regexp/{=;x;1!p;g;$!N;p;D;}' -e h

# 显示包含“ AAA ”、“ BBB ”或“ CCC ”的行（任意次序）
sed '/AAA/!d; /BBB/!d; /CCC/!d'  # 字串的次序不影响结果

# 显示包含“ AAA ”、“ BBB ”和“ CCC ”的行（固定次序）
sed '/AAA.*BBB.*CCC/!d'

# 显示包含“ AAA ”“ BBB ”或“ CCC ”的行 （模拟“ egrep ”）
sed -e '/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d    # 多数 sed
gsed '/AAA\|BBB\|CCC/!d'                        # 对 GNU sed 有效

# 显示包含“ AAA ”的段落 （段落间以空行分隔）
# HHsed v1.5 必须在“ x; ”后加入“ G; ”，接下来的 3 个脚本都是这样
sed -e '/./{H;$!d;}' -e 'x;/AAA/!d;'

# 显示包含“ AAA ”“ BBB ”和“ CCC ”三个字串的段落 （任意次序）
```

```
sed -e '/./{H;$!d;}' -e 'x;/AAA/!d;/BBB/!d;/CCC/!d'

# 显示包含“AAA”、“BBB”、“CCC”三者中任一字串的段落 （任意次序）
sed -e '/./{H;$!d;}' -e 'x;/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d
gsed '/./{H;$!d;};x;/AAA\|BBB\|CCC/b;d'         # 只对 GNU sed 有效

# 显示包含 65 个或以上字符的行
sed -n '/^.\{65\}/p'

# 显示包含 65 个以下字符的行
sed -n '/^.\{65\}/!p'            # 方法 1，与上面的脚本相对应
sed '/^.\{65\}/d'                # 方法 2，更简便一点的方法

# 显示部分文本——从包含正则表达式的行开始到最后一行结束
sed -n '/regexp/,$p'

# 显示部分文本——指定行号范围（从第 8 至第 12 行，含 8 和 12 行）
sed -n '8,12p'                   # 方法 1
sed '8,12!d'                     # 方法 2

# 显示第 52 行
sed -n '52p'                     # 方法 1
sed '52!d'                       # 方法 2
sed '52q;d'                      # 方法 3，处理大文件时更有效率

# 从第 3 行开始，每 7 行显示一次
gsed -n '3~7p'                   # 只对 GNU sed 有效
sed -n '3,${p;n;n;n;n;n;n;}'     # 其他 sed

# 显示两个正则表达式之间的文本（包含）
sed -n '/Iowa/,/Montana/p'       # 区分大小写方式
```

选择性地删除特定行：

```
--------

# 显示通篇文档，除了两个正则表达式之间的内容
sed '/Iowa/,/Montana/d'

# 删除文件中相邻的重复行（模拟“uniq”）
# 只保留重复行中的第一行，其他行删除
sed '$!N; /^\(.*\)\n\1$/!P; D'

# 删除文件中的重复行，不管有无相邻。注意 hold space 所能支持的缓存
# 大小，或者使用 GNU sed。
```

```
sed -n 'G; s/\n/&&/; /^\([ -~]*\n\).*\n\1/d; s/\n//; h; P'

# 删除除重复行外的所有行（模拟“ uniq -d ”）
sed '$!N; s/^\(.*\)\n\1$/\1/; t; D'

# 删除文件中开头的 10 行
sed '1,10d'

# 删除文件中的最后一行
sed '$d'

# 删除文件中的最后两行
sed 'N;$!P;$!D;$d'

# 删除文件中的最后 10 行
sed -e :a -e '$d;N;2,10ba' -e 'P;D'   # 方法 1
sed -n -e :a -e '1,10!{P;N;D;};N;ba'  # 方法 2

# 删除 8 的倍数行
gsed '0~8d'                           # 只对 GNU sed 有效
sed 'n;n;n;n;n;n;n;d;'                # 其他 sed

# 删除匹配式样的行
sed '/pattern/d'                      # 删除含 pattern 的行。当然 pattern
                                      # 可以换成任何有效的正则表达式

# 删除文件中的所有空行（与“ grep '.' ”效果相同）
sed '/^$/d'                           # 方法 1
sed '/./!d'                           # 方法 2

# 只保留多个相邻空行的第一行。并且删除文件顶部和尾部的空行。
# （模拟“ cat -s ”）
sed '/./,/^$/!d'          #方法 1，删除文件顶部的空行，允许尾部保留一空行
sed '/^$/N;/\n$/D'        #方法 2，允许顶部保留一空行，尾部不留空行

# 只保留多个相邻空行的前两行。
sed '/^$/N;/\n$/N;//D'

# 删除文件顶部的所有空行
sed '/./,$!d'

# 删除文件尾部的所有空行
sed -e :a -e '/^\n*$/{$d;N;ba' -e '}'  # 对所有 sed 有效
sed -e :a -e '/^\n*$/N;/\n$/ba'        # 同上，但只对 gsed 3.02.*有效
```

```
# 删除每个段落的最后一行
sed -n '/^$/{p;h;};/./{x;/./p;}'
```

特殊应用：

```
# 移除手册页（man page）中的 nroff 标记。在 Unix System V 或 bash shell 下使
# 用'echo'命令时可能需要加上 -e 选项。
sed "s/.`echo \\\b`//g"    # 外层的双括号是必须的（Unix 环境）
sed 's/.^H//g'             # 在 bash 或 tcsh 中，按 Ctrl-V 再按 Ctrl-H
sed 's/.\x08//g'           # sed 1.5，GNU sed，ssed 所使用的十六进制的表示方法

# 提取新闻组或 e-mail 的邮件头
sed '/^$/q'                # 删除第一行空行后的所有内容

# 提取新闻组或 e-mail 的正文部分
sed '1,/^$/d'              # 删除第一行空行之前的所有内容

# 从邮件头提取“Subject”（标题栏字段），并移除开头的“Subject:”字样
sed '/^Subject: */!d; s///;q'

# 从邮件头获得回复地址
sed '/^Reply-To:/q; /^From:/h; /./d;g;q'

# 获取邮件地址。在上一个脚本所产生的那一行邮件头的基础上进一步的将非电邮
# 地址的部分剃除。（见上一脚本）
sed 's/ *(.*)//; s/>.*//; s/.*[:<] *//'

# 在每一行开头加上一个尖括号和空格（引用信息）
sed 's/^/> /'

# 将每一行开头处的尖括号和空格删除（解除引用）
sed 's/^> //'

# 移除大部分的 HTML 标签（包括跨行标签）
sed -e :a -e 's/<[^>]*>//g;/</N;//ba'

# 将分成多卷的 uuencode 文件解码。移除文件头信息，只保留 uuencode 编码部分。
# 文件必须以特定顺序传给 sed。下面第一种版本的脚本可以直接在命令行下输入；
# 第二种版本则可以放入一个带执行权限的 shell 脚本中。（由 Rahul Dhesi 的一
# 个脚本修改而来。）
sed '/^end/,/^begin/d' file1 file2 ... fileX | uudecode   # vers. 1
sed '/^end/,/^begin/d' "$@" | uudecode                    # vers. 2
```

```
# 将文件中的段落以字母顺序排序。段落间以（一行或多行）空行分隔。GNU sed 使用
# 字元“ \v ”来表示垂直制表符，这里用它来作为换行符的占位符——当然你也可以
# 用其他未在文件中使用的字符来代替它。
sed '/./{H;d;};x;s/\n/={NL}=/g' file | sort | sed '1s/={NL}=//;s/={NL}=/\n/g'
gsed '/./{H;d};x;y/\n/\v/' file | sort | sed '1s/\v//;y/\v/\n/'

# 分别压缩每个.TXT 文件，压缩后删除原来的文件并将压缩后的.ZIP 文件
# 命名为与原来相同的名字（只是扩展名不同）。（DOS 环境：“ dir /b ”
# 显示不带路径的文件名）。
echo @echo off >zipup.bat
dir /b *.txt | sed "s/^\(.*\)\.TXT/pkzip -mo \1 \1.TXT/" >>zipup.bat
```

使用 SED：Sed 接受一个或多个编辑命令，并且每读入一行后就依次应用这些命令。当读入第一行输入后，sed 对其应用所有的命令，然后将结果输出。接着再读入第二行输入，对其应用所有的命令……并重复这个过程。上一个例子中 sed 由标准输入设备（即命令解释器，通常是以管道输入的形式）获得输入。在命令行给出一个或多个文件名作为参数时，这些文件取代标准输入设备成为 sed 的输入。sed 的输出将被送到标准输出（显示器）。因此：

```
cat filename | sed '10q'          # 使用管道输入
sed '10q' filename                # 同样效果，但不使用管道输入
sed '10q' filename > newfile      # 将输出转移（重定向）到磁盘上
```

要了解 sed 命令的使用说明，包括如何通过脚本文件（而非从命令行）来使用这些命令，请参阅《sed & awk》第二版，作者 Dale Dougherty 和 Arnold Robbins（O'Reilly，1997；http://www.ora.com），《UNIX Text Processing》，作者 Dale Dougherty 和 Tim O'Reilly（Hayden Books，1987）或者是 Mike Arst 写的教程——压缩包的名称是“ U-SEDIT2.ZIP ”（在许多站点上都找得到）。要发掘 sed 的潜力，则必须对“ 正则表达式 ”有足够的理解。正则表达式的资料可以看《Mastering Regular Expressions》作者 Jeffrey Friedl（O'reilly 1997）。Unix 系统所提供的手册页（“ man ”）也会有所帮助（试一下这些命令“ man sed ”、“ man regexp ”，或者看“ man ed ”中关于正则表达式的部分），但手册提供的信息比较“ 抽象 ”——这也是它一直为人所诟病的。不过，它本来就不是用来教初学者如何使用 sed 或正则表达式的教材，而只是为那些熟悉这些工具的人提供的一些文本参考。

括号语法：前面的例子对 sed 命令基本上都使用单引号（'...'）而非双引号（"..."）这是因为 sed 通常是在 Unix 平台上使用。单引号下，Unix 的 shell（命令解释器）不会对美元符（$）和后引号（`...`）进行解释和执行。而在双引号下美元符会被展开为变量或参数的值，后引号中的命令被执行并以输出的结果代替后引号中的内容。而在“ csh ”及其衍生的 shell 中使用感叹号（!）时需要在其前面加上转义用的反斜杠（就像这样：\!）以保证上面所使用的例子能正常运行

（包括使用单引号的情况下）。DOS 版本的 Sed 则一律使用双引号（"..."）而不是引号来圈起命令。

'\t'的用法：为了使本文保持行文简洁，我们在脚本中使用'\t'来表示一个制表符。但是现在大部分版本的 sed 还不能识别'\t'的简写方式，因此当在命令行中为脚本输入制表符时，你应该直接按 TAB 键来输入制表符而不是输入'\t'。下列的工具软件都支持'\t'做为一个正则表达式的字元来表示制表符：awk、perl、HHsed、sedmod 以及 GNU sed v3.02.80。

不同版本的 SED：不同的版本间的 sed 会有些不同之处，可以想象它们之间在语法上会有差异。具体而言，它们中大部分不支持在编辑命令中间使用标签（:name）或分支命令（b,t），除非是放在那些的末尾。这篇文档中我们尽量选用了可移植性较高的语法，以使大多数版本的 sed 的用户都能使用这些脚本。不过 GNU 版本的 sed 允许使用更简洁的语法。想像一下当读者看到一个很长的命令时的心情：

```
sed -e '/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d
```

好消息是 GNU sed 能让命令更紧凑：

```
sed '/AAA/b;/BBB/b;/CCC/b;d'      # 甚至可以写成
sed '/AAA\|BBB\|CCC/b;d'
```

此外，请注意虽然许多版本的 sed 接受象“ /one/ s/RE1/RE2/ ”这种在's'前带有空格的命令，但这些版本中有些却不接受这样的命令:“ /one/! s/RE1/RE2/ ”。这时只需要把中间的空格去掉就行了。

速度优化：当由于某种原因（比如输入文件较大、处理器或硬盘较慢等）需要提高命令执行速度时，可以考虑在替换命令（“ s/.../.../ ”）前面加上地址表达式来提高速度。举例来说：

```
sed 's/foo/bar/g' filename          # 标准替换命令
sed '/foo/ s/foo/bar/g' filename    # 速度更快
sed '/foo/ s//bar/g' filename       # 简写形式
```

当只需要显示文件的前面的部分或需要删除后面的内容时，可以在脚本中使用“ q ”命令（退出命令）。在处理大的文件时，这会节省大量时间。因此：

```
sed -n '45,50p' filename            # 显示第 45 到 50 行
sed -n '51q;45,50p' filename        # 一样，但快得多
```

如果你有其他的单行脚本想与大家分享或者你发现了本文档中错误的地方，请发电子邮件给本文档的作者（Eric Pement）。邮件中请记得提供你所使用的 sed 版本、该 sed 所运行的操作系统及对问题的适当描述。本文所指的单行脚本指命令行的长度在 65 个字符或 65 个以下的 sed 脚本〔译注 1〕。本文档的各种脚本是由以下所列作

者所写或提供：

```
Al Aab                          # 建立了“ seders ”邮件列表
Edgar Allen                     # 许多方面
Yiorgos Adamopoulos             # 许多方面
Dale Dougherty                  # 《sed & awk》作者
Carlos Duarte                   # 《do it with sed》作者
Eric Pement                     # 本文档的作者
Ken Pizzini                     # GNU sed v3.02 的作者
S.G. Ravenhall                  # 去 html 标签脚本
Greg Ubben                      # 有诸多贡献并提供了许多帮助
--------------------------------------------------------------------------
```

译注 1：大部分情况下，sed 脚本无论多长都能写成单行的形式（通过`-e'选项和`;'号）——只要命令解释器支持，所以这里说的单行脚本除了能写成一行还对长度有所限制。因为这些单行脚本的意义不在于它们是以单行的形式出现。而是让用户能方便地在命令行中使用这些紧凑的脚本才是其意义所在。

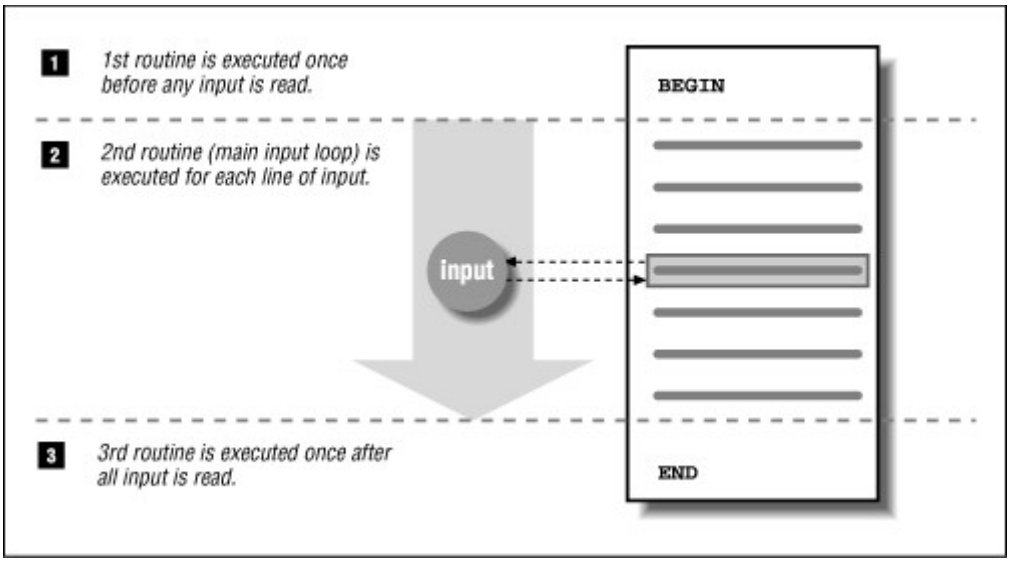
详解 awk

简介

AWK 是一种优良的文本处理工具。它不仅是 Linux 中也是任何环境中现有的功能最强大的数据处理引擎之一。这种编程及数据操作语言（其名称得自于它的创始人 Alfred Aho 、Peter Weinberger 和 Brian Kernighan 姓氏的首个字母）的最大功能取决于一个人所拥有的知识。

AWK 提供了极其强大的功能：可以进行样式装入、流控制、数学运算符、进程控制语句甚至于内置的变量和函数。它具备了一个完整的语言所应具有的几乎所有精美特性。实际上 AWK 的确拥有自己的语言： AWK 程序设计语言， 三位创建者已将它正式定义为“ 样式扫描和处理语言 ”。它允许您创建简短的程序，这些程序读取输入文件、为数据排序、处理数据、对输入执行计算以及生成报 表，还有无数其他的功能。

Awk 由循环组成，一个循环是一个历程，它将一直重复执行直到有一些存在的条件终止它。我们不用写这个循环，在 awk 中它作为一个框架存在，在这个框架中你编写的代码能够执行。Awk 的代码由 3 个主要部分构成（如下图）



初探 awk

匹配空行

```
# awk ' /^$/ {print this is kh }' sample
```

注：sample 文件请自行建立。

如果在 shell 中直接使用 awk，那么在单引号之间不能再使用单引号否则会出错

通过这个实例，大家可以自行试试匹配数字[0-9],匹配字母[a-z[A-Z]

Awk 中字段的分割

Awk 假设它的输入都是有结构的，而不只是一串无规则的字符。默认情况下 awk 用空格作为分隔符，对于/etc/passwd 这样的文件使用默认显然就比较愚蠢，因为 awk 会把它看做一个整体。使用-F 参数可以设定自己想要的分隔符，定义好后如何调用呢？这时候就用到了$。好现在用实例来说明一下。

匹配 passwd 文件中的用户

```
# awk -F : '{print "username " $1 }' /etc/passwd
```

注：这里使用-F 指定了:为分割符（我这里 F 后带了空格，实际上不带也可以），使用$1 引用第一段（众所周知 passwd 的第一段就是用户名）。朋友们可以可以尝试下用$0 看出来什么结果。

实例二

现在咱来打印第四段，通过这个例子向大家说明$后面可以是非常灵活的（整数）。

写脚本

```
# vim awk
BEGIN{
        FS = ":"
        one = 1
        three = 3
}
{
print $(one + three)
}
```

执行

```
# awk -f awk /etc/passwd
```

注：在简介中的图中已经提到过 BEGIN。FS 是 awk 变量用来指定分隔符。下面的 one 和 three 都是自己定义的变量，并且被我赋值。需要注意的一点是在 awk 中变量区分大小写，并且不可以数字开头。

计算空行的数目

本例主要演示赋值运算符++和+= 。这类符号还有很多如表中所示

Operator	Description
++	Add 1 to variable.
--	Subtract 1 from variable.
+=	Assign result of addition.
-=	Assign result of subtraction.
*=	Assign result of multiplication.
/=	Assign result of division.
%=	Assign result of modulo.
^=	Assign result of exponentiation.
**=	Assign result of exponentiation.[6]

如果对编程有些了解的朋友应该对它们并不陌生。

还是以刚刚自行建立的 sample 为例。

写脚本

```
vim awk
/^$/{
print x = x + 1
}
```

执行后便可统计出空行数。

这里的 x=x+1 完全可以被 x +=1 和 x++ 替代，使代码更简洁。

注：如此统计不够完美，因为数是排列出来的，可以自行尝试使用 END 来显示最后的结果。

计算学生的平均成绩

```
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

代码如下

```
{
        #总成绩
        sum = $2+$3+$4+$5+$6
        #平均分
        avg = sum / 5
        #输出
        print $1 , avg
}
```

结果

```
# awk -f 脚本的位置 成绩单的位置
john 87.4
andrea 86
jasper 85.6
```

银行账单

账单如下

```
总资产 1000
编号      地点      数目
125     Market  125.45
126     Hardware Store  34.95
127     Video Store     7.45
128     Book Store      14.32
129     Gasoline        16.10
```

代码如下

```
BEGIN{
        FS = "\t"
        blance = 1000
}

{
        blance  = blance- $3
        print blance
}
```

注：此题不难，请朋友们按照自己的思路写一个脚本来计算。

简洁的写法

```
awk 'BEGIN{FS="\t"}{count+=$3}END{print 1000-count}' checks.data
```

统计 LS 的信息

Ls –l 输出的结果

```
total 52
-rwxr-xr-x 1  502 games  92 Mar  2  1997 acro
-rw-r--r-- 1  502 games 247 Mar  2  1997 acronyms
-rw-r--r-- 1 root root   76 Feb  4 09:49 awk
-rw-r--r-- 1  502 games 298 Mar  2  1997 checkbook.awk
-rw-r--r-- 1  502 games 109 Mar  2  1997 checks.data
-rwxr-xr-x 1  502 games 163 Mar  2  1997 filesum1
-rwxr-xr-x 1  502 games 749 Mar  2  1997 filesum2
-rwxr-xr-x 1  502 games 766 Mar  2  1997 filesum3
-rwxr-xr-x 1  502 games  42 Mar  2  1997 fls1
-rw-r--r-- 1  502 games 244 Mar  2  1997 fls.data
-rw-r--r-- 1  502 games  92 Mar  2  1997 grades1.awk
-rw-r--r-- 1  502 games 100 Mar  2  1997 grades2.awk
-rw-r--r-- 1  502 games  64 Mar  2  1997 grades.data
```

现在我来写一个脚本统计 ls –l 中文件的数量及大小。

脚本的框架应该是这样的

```
ls -l $* | awk '{
        print $5, "\t", $9
}'
```

$*是 shell 里的一个变量用来扩展通过命令行传递的所有变量。这些参数可能是文件名，目录或 ls 的附加选项。

好，让我们先来统计文件的个数吧。

```
NF==9 && /^-/{
         Filenum ++
}
END { print  "there are",filenum,"files here" }
```

注：NF==9 用来过滤第一行 total 52。

/^-/是正则，匹配文件，在 Linux 中文件是用-来表示。

现在来统计大小

```
NF==9 && /^-/{
         filenum ++
         total += $5
}
END { print  "there are",filenum,"files here" , "total",total, "bytes"}
```

到此为止文件个数和大小都已经被统计。但是为了更清楚我们到底统计了哪些文件，我们还需要完善一下该脚本。

```
BEGIN {
        print "files","bytes"
}
NF==9{
        filenum ++
        total += $5
        print $9,$5
}
END { print  "there are",filenum,"files here" , "total",total, "bytes"}
```

运行后输出结果格式比较混乱

```
files bytes
acro 92
acronyms 247
awk 76
checkbook.awk 298
checks.data 109
filesum1 163
filesum2 749
filesum3 766
fls 172
fls.data 244
grades1.awk 92
grades2.awk 100
```

grades.data 64
there are 13 files here total 3172 bytes
好，现在使用 printf 来整理一下

格式化输出 printf

Awk 的 printf 与 c 一样。以下是用在 printf 中的格式说明符

c ASCII character

d Decimal integer 十进制整数

i Decimal integer. (Added in POSIX)

e Floating-point format ([-]*d.precision***e**[+-]*dd*)

E Floating-point format ([-]*d.precision***E**[+-]*dd*)

f Floating-point format ([-]*ddd.precision*)

g **e** or **f** conversion, whichever is shortest, with trailing zeros removed

G **E** or **f** conversion, whichever is shortest, with trailing zeros removed

o Unsigned octal value

s String 字符串

x Unsigned hexadecimal number. Uses **a-f** for 10 to 15

X Unsigned hexadecimal number. Uses **A-F** for 10 to 15

% Literal %

这次主要用 printf 来规定对齐方式。

右对齐

```
# awk 'END {printf ("|%10s|\n","hello")}' /etc/passwd
|     hello|
```

左对齐

```
# awk 'END {printf ("|%-10s|\n","hello")}' /etc/passwd
|hello     |
```

好了不知道大家有没有点感觉，如何让上例中的格式对齐呢？

格式化后的完整代码如下

```
ls -l $* | awk '
BEGIN{
        printf("%-15s\t%10s\n","files","bytes")
}

NF==9{
        filenum ++
        total += $5
        printf ("%-15s\t%10d\n",$9,$5)
}
END { print  "There are",filenum,"files here\n" "Total",total, "bytes"}
'
```

统计学生成绩单

首先我们要使用的是 for 循环，利用它制造出一个“万能”平均分计算器。

For 循环的框架

for (*变量初值* ; *条件（范围）* ; *计数方法*)
动作
以该成绩单为例
mona 70 77 85 83 70 89
john 85 92 78 94 88 91
andrea 89 90 85 94 90 95
jasper 84 88 80 92 84 82
dunce 64 80 60 60 61 62
ellis 90 98 89 96 96 92
代码如下

```
        total = 0
        for (i=2 ;i<=NF;++i )
        {
                total = total + $i
        }
        avg = total / (NF-1)
```

注：total 是总成绩
(NF -1)因为第一列是名字，所以需要-1 。这里的括号一定不能少，不然除法是比加法优先的。
好了，现在不管分有多少我们都可以用它算出成绩来了！
成绩出来了就一定有好有坏，现在来用 if 语句来把及格和不及格的成绩过滤出来。
If 语句的框架如下
if (*条件*){
动作 1 }
else
动作 2
现在我们用 if 来判断学生的成绩是否及格。这里以 60 为底线。

```
        if ( avg >= 60)
                grade = "good job!"
        else
                grade = "sorry!"
```

有时遇到的情况可能会更复杂一些，比如要为成绩分类，统计出每个层次的学生数量等。这时可以用 else if 来设置多个条件。
现在假设要把成绩分为 4 类。A,B,C,D

```
        if ( avg >= 90) grade = "A"
        else if ( avg >= 80 ) grade = "B"
        else if ( avg >= 60 ) grade = "C"
        else if ( avg <= 60 ) grade = "D"
```

结合上面的代码，一个统计学生成绩单的程序就出来了！完整代码如下

```
{
        total = 0
        for (i=2 ;i<=NF;++i )
        {
                total = total + $i
```

```
        }
        avg = total / (NF-1)
        if ( avg >= 90) grade = "A"
        else if ( avg >= 80 ) grade = "B"
        else if ( avg >= 60 ) grade = "C"
        else if ( avg <= 60 ) grade = "D"

        print $1,avg,grade
}
```

请朋友们自己试着统计出每个层次的学生数量。

技巧：在代码顶部加入`#!/bin/awk -f`可以直接调用 awk 来执行人物。有时候比使用 *awk -f 脚本 目标* 的方式更简洁一些。

好，现在来做下练习巩固前面所学到的东西，之后进行数组的学习。

练习

打印输入文件第八行

```
#!/bin/awk -f
{
  if ( NR == 8 )
  { print $0 }
}
```

打印输入行的总数：`awk -F: 'END{print NR}' passwd`

打印字段数大于等于 4 个的行：`awk -F: 'NF >=4 {print $0}' passwd`

打印文件所有字段的总数 `awk -F ":" 'BEGIN { N=0 } {n+=NF}END{ print n}' /etc/passwd`

打印 UID 在 30～40 范围内的用户名：`awk 'BEGIN {FS=":"} { if ($3 >= 30 && $3 <= 40) {print $0}}' /etc/passwd`

倒序排列文件的所有字段

注：标记为红色的是我个人曾未做出的题目

```
BEGIN {
        FS=":|:/"
}
{
        for (x=NF ; x>=1 ; --x)
        printf("%s:",$x)
        printf("\n") #这是重点
}
```

隔行删除：`awk -F ":" '{if ( NR%2==1 ) print $0}' /etc/passwd`

抽取每行第一次出现的单词

```
awk -F "[^a-zA-Z]+" '/.$/{if ($1 ~ /[a-zA-Z]+/) print $1 ; else print $2}' /etc/passwd
```

打印字段大于 5 个的行总数

```
BEGIN {
        FS=":"
}
NR > 5{
        num ++
```

```
}
END {print num}
```
输出文件的每一行的倒数第二个字段：
```
BEGIN {
        FS=":|/"
}
{
        for (i=0 ; i <= NR ; ++i)
        NR == i
        print $(NF -1)
}
```
输出可以登录与不可以登录的用户数量：
```
BEGIN {
        FS = ":|:/"
}
{
if (/bash/){ 可以登陆
        ++num
}
else if (/nologin/){ 不可登陆
        ++num2
}
else{ 其他
        ++num3
}
}
END {
        print num,num2,num3
}
```
数组
数组是可以用来存储一组数据的变量。通常这些数据之间具有某种联系。数组中的每一个元素通过它们在数组中的下标来访问。下面是数据的框架

array[*下标*] = *元素*

在 awk 中不必指明数组的大小，只需要为数组指定标识符。

下面的例子为数组 flavor 的一个元素指定了一个字符串 “ cherry ”

```
flavor[1] = "cherry"
```

这个数组的下标是 “ 1 ”。下面的语句将打印 “ cherry ”

```
print flavor[1]
```

好，现在让我们利用数组将学生平均分计算程序更加强大！（**编写时能用数组的地方都用了数组，主要是为了向大家介绍数组。但是大家要明白，那些功能并非一定要用数组才能解决**）

本次需要实现的功能有 1 计算班级平均分 2 统计高于和低于平均分的人数 3 统计 A,B,C,D 中各有多少人。

成绩单

```
mona 70 77 85 83 70 89
```

```
john 85 92 78 94 88 91
andrea 89 90 85 94 90 95
jasper 84 88 80 92 84 82
dunce 64 80 60 60 61 62
ellis 90 98 89 96 96 92
```

执行效果。（着色部分是本次加入的新功能）

```
$ awk -f grades.awk grades.test
mona    79      C
john    88      B
andrea  90.5    A
jasper  85      B
dunce   64.5    D
ellis   93.5    A

Class Average:  83.4167
At or Above Average:    4
Below Average:  2
A:      2
B:      2
C:      1
D:      1
```

代码如下

```
{
        total = 0
        for (i=2;i<=NF;i++)
                total += $i
        avg = total / (NF -1)
        if (avg >= 90) grade = "A"
        else if (avg >= 80) grade = "B"
        else if (avg >= 70) grade = "C"
        else grade = "D"

        student_avg_total[NR] = avg
        ++level[grade]

        print $1,avg,grade
}
END{
        for (x=1 ; x <= NR ; x++)
                class_avg_total += student_avg_total[x]
                class_avg = class_avg_total / NR
        print "Class Avg:",class_avg
```

```
        for (x=1; x<=NR;x++)
                if (student_avg_total[x] >= class_avg)
                        ++niubi
                else
                        ++yiban
        print "At or Above Average:",niubi
        print "Below Average:",yiban

        for (num in level)
                print num ":" level[num]

}
```

班级平均分的实现

将所有平均分都放入了数组中，并以 NR 作为下标（因为 NR 值是递增的）。

在 END 中通过一个 for 循环将元素调出，相加。用和除以 NR 便可得出班级平均分。

统计高于和低于平均分的人数

通过 for 循环将平均分调出，然后使用 if 语句进行判断。

统计 A,B,C,D 中各有多少人

在本例中实际上最难理解的点应该在这里 ++level[grade]

在 grade 中存储的是 A,B,C,D。而++level 负责统计字母出现的个数。

```
        for (num in level)
```

这里 num（可以是任意名称）可看做是和普通 for 循环计数器（i++）一样递增的临时变量，in 指定了它作用在哪个数组。

```
                print num ":" level[num]
```

这里 num 调出了 level 的元素名，而 level[num]调出了统计结果。

注：awk 中所有的数组都是关联数组。关联数组的独特之处在于它的下标可以是一个字符串或一个数值。

词汇搜索

本程序根据用户提交的缩略词将文件中的完整写法提取。

文件如下

```
USGCRP  U.S. Global Change Research Program
NASA    National Aeronautic and Space Administration
EOS     Earth Observing System
```

代码

```
BEGIN { FS = "\t"
        printf("Enter a glossary term: ")
}

FILENAME == "glossary" {
        entry[$1] = $2 #将第二段与第一段的缩写对应
        next #将数组载入完成后进入下面的代码段
}
#如果输入内容不为空则进行下面的判断
$0 != "" {
```

```
        #in 是一个操作符，用在条件表达式中来测试一个小标是否是数组的成员
        if ( $0 in entry ) {
                print entry[$0]
        } else
                print $0 " not found"
}
#如果输入内容为空
{
  printf("Enter another: ")
}
```

好，基本功能实现了。因为本脚本从标准输入中读取，所以在执行的时候需要这样写 awk -f lookup1 glossary glossary

本程序有一个缺点就是用户无法主动退出，现在来补充这个内容

```
$0 ~ /^(quit|[qQ]|exit|[Xx])$/ { exit }
```

不需要记，以后自己在写脚本的时候直接复制粘贴即可！

用 split()创建数组

内置函数 split()能够将任何字符串分解到数组的元素中。这个函数对于从字段中提取“子字段”是很有用的。函数 split()的框架如下

n = split(*字符串*，*数组*，*分隔符（或者正则）*)

n 为数组中元素的个数，所以数组中的下标从 1 开始到 n。

打印罗马数字

输入从 1 到 10 的数字并转换为罗马数字。

根据 split 的框架我们可以这样写

```
split ("I,II,III,IV,V,VI,VII,VIII,IX,XI",number,",")
```

这样就把罗马数字存入了数组中，此时 number[1]=I.number[2]=II……

```
#判断$1 是否在 1-10 之间。
$1>0 && $1<=10 {
        #过滤小数
        if (/[0-9]\.[0-9]*/) {
                print "not a good number"
        }else
                print number[$1]
        #exit 告诉程序执行到这里就结束，不然即使输入正确也会报错
        exit
}
```

如果不是 1-10 之间的数字，则报错

```
{
        print "faild"
        exit
}
```

转换日期格式

将“mm-dd-yy”或“mm/dd/yy”转换为“月日，年”

代码如下

```
awk '
```

```
#与打印罗马数字一样的思路一样。首先在 BEGIN 中将 1-12 与 12 月份的英文单词对应。
BEGIN {
        listmonths =
"January,February,March,April,May,June,July,August,September,October,November,De
cember"
        split( listmonths , month ,",")
}
#判断输入
$1 != ""{
        dateg = split($1 , date , "-") #将$1 打散放入数组
        if (dateg == 1) #判断是否有内容的
                datexg == split( $1 , date , "/")
        if (datexg ==1)
                exit
         date[1] += 0 #处理类似于 12/05/09 这样的操作，awk 认为 05 和 5 是两个不一样
的字符，最终结果将导致以 05 表示的五月无法被正常输出。
        print month[date[1]],date[2]",",date[3]
}'
```

处理文章的缩写词

文章如下

```
The USGCRP is a comprehensive
research effort that includes applied
as well as basic research.
The NASA program Mission to Planet Earth
represents the principal space-based component
of the USGCRP and includes new initiatives
```

类似于黄色字体的都是缩写词，本次编写的程序就目的就是把这些词转换。

缩写词的对应关系存储在 acronyms 中

```
USGCRP  U.S. Global Change Research Program
NASA    National Aeronautic and Space Administration
EOS     Earth Observing System
```

代码如下

```
awk 'FILENAME == "acronyms" #读取存储缩写词对应关系的文件
{
         #制作缩写词对应关系的数组（之前也做过，并且比这个方法简单，看以参阅“ 词
汇搜索 ”）
        split($0,entry,"\t")
        acro[entry[1]]=entry[2]
        next
}
#匹配包含多个大写字母的行
/[A-Z][A-Z]+/{
        for (i=1;i<=NF;i++)
                #一段一段的截取出来并判断是否有缩写存在于 acro 中
```

```
                if ($i in acro){
                        #如果存在则进行替换。"("$i")"用来显示被替换的缩写词
                        $i=acro [$i] "("$i")"
                }
}
{
        print $0
}' acronyms $*
```

常用

1.按内存从大到小排列进程:

```
ps -eo "%C : %p : %z : %a"|sort -k5 -nr
```

2.查看当前有哪些进程；查看进程打开的文件:

```
ps -A ；lsof -p PID
```

3.获取当前IP地址（从中学习grep,awk,cut的作用）

```
ifconfig eth0 |grep "inet addr:" |awk '{print $2}'|cut -c 6-
```