第3章

查找与替换

我们在1.2节里曾提及UNIX程序员偏好处理文本的行与列。文本型数据比二进制数据更具灵活性，且UNIX系统也提供许多工具，让用户可以轻松地剪贴文本。

在本章中，我们要讨论的是编写Shell脚本时经常用到的两个基本操作：文本查找（searching）——寻找含有特定文本的行，文本替换（substitution）——更换找到的文本。

虽然你可以使用简单的固定文本字符串完成很多工作，但是正则表达式（regular expression）能提供功能更强大的标记法，以单个表达式匹配各种实际的文本段。本章会介绍两种由不同的UNIX程序所提供的正则表达式风格，然后再进一步介绍提取文本与重新编排文本的几个重要工具。

3.1 查找文本

以`grep`程序查找文本（以UNIX的专业术语来说，是匹配文本（matching text））是相当方便的。在POSIX系统上，`grep`可以在两种正则表达式风格中选择一种，或是执行简单的字符串匹配。

传统上，有三种程序，可以用来查找整个文本文件：

`grep`
: 最早的文本匹配程序。使用POSIX定义的基本正则表达式（Basic Regular Expression，BRE），本章稍后会提到这部分。

`egrep`
: 扩展式`grep`（Extended `grep`）。这个程序使用扩展正则表达式（Extended Regular Expression，ERE）——这是一套功能更强大的正则表达式，使用它的代价就是会

耗掉更多的运算资源。在早期出现的PDP-11的机器上，这点事关重大，不过以现在的系统而言，在性能影响上几乎没有太大的差别。

`fgrep`

快速`grep`（Fast `grep`）。这个版本匹配固定字符串而非正则表达式，它使用优化的算法，能更有效地匹配固定字符串。最初的版本，也是唯一可以并行（in parallel）地匹配多个字符串的版本；也就是说，`grep`与`egrep`只能匹配单个正则表达式；而`fgrep`使用不同的算法，却能匹配多个字符串，有效地测试每个输入行里，是否有匹配的查找字符串。

1992 POSIX标准将这三个改版整合成一个`grep`程序，它的行为是通过不同的选项加以控制。POSIX版本可以匹配多个模式 —— 不管是BRE还是ERE。`fgrep`与`egrep`两者还是可用，只是标记为不推荐使用（deprecated），即它们有可能在往后的标准里删除。果然，在2001 POSIX标准里，就只纳入合并后的`grep`命令。不过实际上，`egrep`与`fgrep`在所有UNIX与类UNIX的系统上都还是可用的。

3.1.1 简单的grep

`grep`最简单的用法就是使用固定字符串：

```
$ who                                                  有谁登录了
tolstoy  tty1         Feb 26 10:53
tolstoy  pts/0        Feb 29 10:59
tolstoy  pts/1        Feb 29 10:59
tolstoy  pts/2        Feb 29 11:00
tolstoy  pts/3        Feb 29 11:00
tolstoy  pts/4        Feb 29 11:00
austen   pts/5        Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6        Feb 29 15:39 (mansfield-park.example.com)
$ who | grep -F austen                          austen登录于何处
austen   pts/5        Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6        Feb 29 15:39 (mansfield-park.example.com)
```

范例中使用`-F`选项，以查找固定字符串**`austen`**。事实上，只要匹配的模式里未含有正则表达式的meta字符（metacharacter），则`grep`默认行为模式就等同于使用了`-F`：

```
$ who | grep austen                                    不具 -F，但结果一样
austen   pts/5        Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6        Feb 29 15:39 (mansfield-park.example.com)
```

3.2 正则表达式

本节提供有关正则表达式构造与匹配方式的概述。特别会提及POSIX BRE与ERE构造，因为它们想要将大部分UNIX工具里的两种正则表达式基本风格（flavors）加以正式化。

grep

语法

grep [*options ...*] *pattern-spec* [*files ...*]

用途

显示匹配一个或多个模式的文本行。时常会作为管道（pipeline）的第一步，以便对匹配的数据作进一步处理。

主要选项

-E

使用扩展正则表达式进行匹配。grep -E可取代传统的egrep。

-F

使用固定字符串进行匹配。grep -F可取代传统的fgrep命令。

-e *pat-list*

通常，第一个非选项的参数会指定要匹配的模式。你也可以提供多个模式，只要将它们放在引号里并以换行字符分隔它们。模式以减号开头时，grep会混淆，而将它视为选项。这就是-e选项派上用场的时候，它可以指定其参数为模式——即使它以减号开头。

-f *pat-file*

从*pat-file*文件读取模式作匹配。

-i

模式匹配时忽略字母大小写差异。

-l

列出匹配模式的文件名称，而不是打印匹配的行。

-q

静默地。如果模式匹配匹配，则grep会成功地离开，而不将匹配的行写入标准输出；否则即是不成功。（我们尚未讨论成功/不成功；可参考6.2节）。

-s

不显示错误信息。通常与-q并用。

-v

显示不匹配模式的行。

行为模式

读取命令行上指名的每个文件。发现匹配查找模式的行时，将它显示来。当指明多个文件时，grep会在每一行前面加上文件名与一个冒号。默认使用BRE。

警告

你可以使用多个-e与-f选项，建立要查找的模式列表。

我们期望你在阅读这本书前，已经接触过正则表达式与文本匹配，并已有些了解。如果是这样，下面的段落将澄清如何使用正则表达式完成具有可移植性的Shell脚本。

若你完全没接触过正则表达式，那么这里提到的东西对你来说可能太简略了，你应该先去看看介绍性的资料，例如《Learning the UNIX Operating System》（O'Reilly）或是《sed & awk》（O'Reilly）。因为正则表达式是UNIX工具使用和构建模型上的基础，花些时间学习如何使用它们并且好好利用它们，你会不断地从各个层面得到充分的回报。

如果你使用正则表达式处理文本已有多年经验，可能会觉得这里所介绍的内容略嫌粗略。在这种情况下，我们会建议你浏览了第一部分POSIX BRE与ERE的表格式概括之后，就直接跳到下一节，然后找一些比较深入的资料来阅读，例如《Mastering Regular Expressions》（O'Reilly）。

3.2.1 什么是正则表达式

正则表达式是一种表示方式，让你可以查找匹配特定准则的文本，例如，“以字母a开头”。此表示法让你可以写一个表达式，选定或匹配多个数据字符串。

除了传统的UNIX正则表达式表示法之外，POSIX正则表达式还可以做到：

- 编写正则表达式，它表示特定于locale的字符序列顺序和等价字符。
- 编写正则表达式，而不必关心系统底层的字符集是什么。

很多的UNIX工具程序沿用某一种正则表达式形式来强化本身的功能。这里列举一部分例子：

- 用来寻找匹配文本行的`grep`工具族：`grep`与`egrep`，以及非标准但很好用的`agrep`工具（注1）。
- 用来改变输入流的`sed`流编辑器（stream editor），本章稍后将会介绍。
- 字符串处理程序语言，例如`awk`、Icon、Perl、Python、Ruby、Tcl等。
- 文件查看程序（有时称为分页程序，pagers），例如`more`、`page`，与`pg`，都常出现在商用UNIX系统上，另外还有广受欢迎的`less`分页程序（注2）。

注1：1992年原始的UNIX版本是在*ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z*。Windows版本则在*http://www.tgries.de/agrep/337/agrep337.zip*。`agrep`不同于我们在本书中介绍的大部分可自由下载的软件，它并不能随意地用于任何目的；你可以参考程序所附的许可文件。

注2：与`more`对应的双关语。见*ftp://ftp.gnu.org/gnu/less/*。

- 文本编辑器，例如历史悠久的ed行编辑器、标准的vi屏幕编辑器，还有一些插件（add-on）编辑器，例如emacs、jed、jove、vile、vim等。

正因为正则表达式对于UNIX的使用是这么的重要，所以花些时间把它们弄熟绝对不会错，越早开始就能掌握得越好。

从根本上来看，正则表达式是由两个基本组成部分所建立：一般字符与特殊字符。一般字符指的是任何没有特殊意义的字符，正如下表中所定义的。在某些情况下，特殊字符也可以视为一般字符。特殊字符常称为元字符（metacharacter），本章接下来的部分都会以 meta 字符表示。表 3-1 为 POSIX BRE 与 ERE 的 meta 字符列表。

表 3-1：POSIX BRE 与 ERE 的 meta 字符

字符	BRE/ERE	模式含义
\	两者都可	通常用以关闭后续字符的特殊意义。有时则是相反地打开后续字符的特殊意义，例如 \(...\)与\{...\}。
.	两者都可	匹配任何单个的字符，但 NUL 除外。独立程序也可以不允许匹配换行字符。
*	两者都可	匹配在它之前的任何数目（或没有）的单个字符。以ERE而言，此前置字符可以是正则表达式，例如：因为.（点号）表示任一字符，所以 .* 代表“匹配任一字符的任意长度”。以 BRE 来说，* 若置于正则表达式的第一个字符，不具任何特殊意义。
^	两者都可	匹配紧接着的正则表达式，在行或字符串的起始处。BRE：仅在正则表达式的开头处具此特殊含义，ERE：置于任何位置都具特殊含义。
$	两者都可	匹配前面的正则表达式，在字符串或行结尾处。BRE：仅在正则表达式结尾处具特殊含义。ERE：置于任何位置都具特殊含义。
[...]	两者都可	方括号表达式（bracket expression），匹配方括号内的任一字符。连字符（-）指的是连续字符的范围（注意：范围会因 locale 而有所不同，因此不具可移植性）。^ 符号置于方括号里第一个字符则有反向含义：指的是匹配不在列表内（方括号内）的任何字符。作为首字符的一个连字符或是结束方括号（]），则被视为列表的一部分。所有其他的 meta 字符也为列表的一部分（也就是：根据其字面上的意义）。方括号表达式里可能会含有排序符号（collating symbol）、等价字符集（equivalence class），以及字符集（character class）（文后将有介绍）。

表 3-1：POSIX BRE 与 ERE 的 meta 字符（续）

字符	BRE/ERE	模式含义
\{*n*,*m*\}	BRE	区间表达式（interval expression），匹配在它前面的单个字符重现（occurrences）的次数区间。\{*n*\}指的是重现 *n* 次；\{*n*,\}则为至少重现（occurrences）*n* 次，而\{*n*,*m*\}为重现 *n* 至 *m* 次。*n* 与 *m* 的值必须介于 0 至 RE_DUP_MAX（含）之间，后者最小值为 255。
\(\)	BRE	将\(与\)间的模式存储在特殊的“保留空间（holding space）”。最多可以将 9 个独立的子模式（subpattern）存储在单个模式中。匹配于子模式的文本，可通过转义序列（escape sequences）\1 至 \9，被重复使用在相同模式里。例如**\(ab\).*\1**，指的是匹配于 ab 组合的两次重现，中间可存在任何数目的字符。
n	BRE	重复在\(与\)方括号内第 *n* 个子模式至此点的模式。*n* 为 1 至 9 的数字，1 为由左开始。
{*n*,*m*}	ERE	与先前提及 BRE 的\{*n*,*m*\}一样，只不过方括号前没有反斜杠。
+	ERE	匹配前面正则表达式的一个或多个实例。
?	ERE	匹配前面正则表达式的零个或一个实例。
|	ERE	匹配于|符号前或后的正则表达式。
( )	ERE	匹配于方括号括起来的正则表达式群。

表 3-2 列举了一些简单的范例。

表 3-2：简单的正则表达式匹配范例

表达式	匹配
tolstoy	位于一行上任何位置的 7 个字母：tolstoy
^tolstoy	7 个字母 tolstoy，出现在一行的开头
tolstoy$	7 个字母 tolstoy，出现在一行的结尾
^tolstoy$	正好包括 tolstoy 这 7 个字母的一行，没有其他的任何字符
[Tt]olstoy	在一行上的任意位居中，含有 Tolstoy 或是 tolstoy
tol.toy	在一行上的任意位居中，含有 tol 这 3 个字母，加上任何一个字符，再接着 toy 这 3 个字母
tol.*toy	在一行上的任意位居中，含有 tol 这 3 个字母，加上任意的 0 或多个字符，再继续 toy 这 3 个字母（例如，toltoy、tolstoy、tolWHOtoy 等）

3.2.1.1 POSIX 方括号表达式

为配合非英语的环境，POSIX 标准强化其字符集范围的能力（例如，**[a-z]**），以匹配

非英文字母字符。举例来说，法文的è是字母字符，但以传统字符集**[a-z]**匹配并无该字符。此外，该标准也提供字符序列功能，可用以在匹配及排序字符串数据时，将序列里的字符视为一个独立单位（例如，将 locale 中 ch 这两个字符视为一个单位，在匹配与排序时也应这样看待）。越来越广为使用的Unicode字符集标准，进一步地增加了在简单范围内使用它的复杂度，使得它们对于现代应用程序而言更加不适用。

POSIX也在一般术语上作了些变动，我们早先看到的范围表达式在UNIX里通常称为字符集(character class)，在POSIX的标准下，现在叫做方括号表达式(bracket expression)。在方括号表达式里，除了字面上的字符（例如 z、;等等）之外，另有额外的组成部分，包括：

字符集（Character class）
: 以**[:**与**:]**将关键字组合括起来的 POSIX 字符集。关键字描述各种不同的字符集，例如英文字母字符、控制字符等，见表 3-3。

排序符号（Collating symbol）
: 排序符号指的是将多字符序列视为一个单位。它使用**[.**与**.]**将字符组合括起来。排序符号在系统所使用的特定 locale 上各有其定义。

等价字符集（Equivalence class）
: 等价字符集列出的是应视为等值的一组字符，例如 e 与 è。它由取自于locale的名字元素组成，以[= 与 =]括住。

这三种构造都必须使用方括号表达式。例如**[[:alpha:]!]**匹配任一英文字母字符或惊叹号 (!)；而**[[.ch.]]**则匹配于 ch（排序元素），但字母 c 或 h 则不是。在法文 French 的 locale 里，**[[=e=]]**可能匹配于 e、è、ë、ê 或 é。接下来会有字符集、排序符号，以及等价字符集的详细说明。

表 3-3 描述 POSIX 字符集。

表 3-3：POSIX 字符集

类别	匹配字符	类别	匹配字符
[:alnum:]	数字字符	[:lower:]	小写字母字符
[:alpha:]	字母字符	[:print:]	可显示的字符
[:blank:]	空格（space）与定位（tab）字符	[:punct:]	标点符号字符
[:cntrl:]	控制字符	[:space:]	空白（whitespace）字符
[:digit:]	数字字符	[:upper:]	大写字母字符
[:graph:]	非空格（nonspace）字符	[:xdigit:]	十六进制数字

BRE 与 ERE 共享一些常见的特性，不过仍有些重要差异。我们会从 BRE 的说明开始，再介绍 ERE 附加的 meta 字符，最后针对使用相同（或类似）meta 字符但拥有不同语义（或含义）的情况进行说明。

3.2.2 基本正则表达式

BRE是由多个组成部分所构建，一开始提供数种匹配单个字符的方式，而后又结合额外的 meta 字符，进行多字符匹配。

3.2.2.1 匹配单个字符

最先开始是匹配单个字符。可采用集中方式做到：以一般字符、以转义的meta字符、以 .（点号）meta 字符，或是用方括号表达式：

- 一般字符指的是未列于表 3-1 的字符，包括所有文字和数字字符、绝大多数的空白（whitespace）字符以及标点符号字符。因此，正则表达式 **a**，匹配于字符 a。我们可以说，一般字符所表示的就是它们自己，且这种用法应是最直接且易于理解的。所以，**shell** 匹配于 shell；**WoRd** 匹配于 WoRd，但不匹配于 word。
- 若 meta 字符不能表示它们自己，那当我们需要让 meta 字符表示它们自己的时候，该怎么办？答案是转义它。在前面放一个反斜杠来做到这一点。因此，***** 匹配于字面上的 *，**** 匹配于字面上的反斜杠，还有 **\[** 匹配于左方括号（若将反斜杠放在一般字符前，则 POSIX 标准保留此行为模式为未定义状态。不过通常这种情况下反斜杠会被忽略，只是很少人会这么做）。
- .（点号）字符意即“任一字符”。因此，**a.c** 匹配于 abc、aac 以及 aqc。单个点号用以表示自己的情况很少，它多半与其他 meta 字符搭配使用，这一结合允许匹配多个字符，这部分稍后会提及。
- 最后一种匹配单个字符的方式是使用方括号表达式（bracket expression）。最简单的方括号表达式是直接将字符列表放在方括号里，例如，**[aeiouy]** 表示的就是所有小写元音字母。举例来说，**c[aeiouy]t** 匹配于 cat、cot 以及 cut（还有 cet、cit，与 cyt），但不匹配于 cbt。

  在方括号表达式里，^ 放在字首表示是取反（complement）的意思；也就是说，不在方括号列表里的任意字符。所以 **[^aeiouy]** 指的就是小写元音字符以外的任何字符，例如：大写元音字母、所有辅音字母、数字、标点符号等。

将所有要匹配的字母全列出来是一件无聊又麻烦的事。例如 **[0123456789]** 指所有数字，或 **[0123456789abcdefABCDEF]** 表示所有十六进制数字。因此，方括号表达式可以包括字符的范围。像前面提到的两个例子，就可以分别以 **[0-9]** 与 **[0-9a-fA-F]** 表示。

警告： 一开始，范围表示法匹配字符时，是根据它们在机器中字符集内的数值。因此字符集的不同（ASCII v.s EBCDIC），会使得表示法无法百分之百地具有可移植性，但实际上问题不大，因为几乎所有的 UNIX 系统都是使用 ACSII。

以 POSIX 的 locale 来说，某些地方可能会有问题。范围使用的是各个字符在 locale 排序序列里所定义的位置，与机器字符集里的数值不相关。因此，范围表示法仅在程序运行在 locale 设置为“POSIX”之下，才具可移植性。前面所提及的 POSIX 字符集表示法，提供一种可移植方式表示概念，例如“所有数字”，或是“所有字母字符”，因此在方括号表达式内的范围不建议用在新程序里。

在前面的 3.2.1 节里，我们曾简短地介绍 POSIX 的排序符号（collating symbol）、等价字符集（equivalence class）以及字符集（character class）。这些是方括号表达式最后出现的组成部分。接下来我们就要说明它们的构造方式。

在部分非英语系的语言里，为了匹配需要，某些成对的字符必须视为单个字符。像这样的成对字符，当它们与语言里的单个字符比较时，都有其排序的定义方式。例如，在 Czech 与 Spanish 语系下，ch 两个字符会保持连续状态，在匹配时，会视为单个独立单位。

排序（collating）是指给予成组的项目排列顺序的操作。一个 POSIX 的排序元素由当前 locale 中的元素名称组成，并由 [. 与 .] 括起来。以刚才讨论的 ch 来说，locale 可能会用 [.ch.]（我们说“可能”是因为每一个 locale 都有自己定义的排序元素）。假定 [.ch.] 是存在的，那么正则表达式 **[ab[.ch.]de]** 则匹配于字符 a、b、d 或 e，或者是成对的 ch；而单独的 c 或 h 字符则不匹配。

等价字符集（equivalence class）用来让不同字符在匹配时视为相同字符。等价字符集将类别（class）名称以 [= 与 =] 括起来。举例来说，在 French 的 locale 下，可能有 [=e=] 这样的等价字符集，在此类别存在的情况下，正则表达式 **[a[=e=]iouy]** 就等同于所有小写英文字母元音，以及字母 è、é 等。

最后一个特殊组成部分：字符集，它表示字符的类别，例如数字、小写与大写字母、标点符号、空白（whitespace）等。这些类别名称定义于 [: 与 :] 之间。完整列表如表 3-3 所示。前 POSIX（pre-POSIX）范围表达式对于十进制与十六进制数字的表示（应该）是具有可移植性的，可使用字符集：**[[:digit:]]** 与 **[[:xdigit:]]** 达成。

注意： 排序元素、等价字符集以及字符集，都仅在方括号表达式的方括号内认可，也就是说，像 **[:alpha:]** 这样的正则表达式，匹配字符为 a、l、p、h 以及 :，表示所有英文字母的正确写法应为 **[[:alpha:]]**。

在方括号表达式中，所有其他的 meta 字符都会失去其特殊含义。所以**[*\.]**匹配于字面上的星号、反斜杠以及句点。要让]进入该集合，可以将它放在列表的最前面：**[]*\.]**，将]增加至此列表中。要让减号字符进入该集合，也请将它放到列表最前端：**[-*\.]**。若你需要右方括号与减号两者进入列表，请将右方括号放到第一个字符、减号放到最后一个字符：**[]*\.-]**。

最后，POSIX 明确陈述：NUL 字符（数值的零）不需要是可匹配的。这个字符在 C 语言里是用来指出字符串结尾，而 POSIX 标准则希望让它是直截了当的，通过正规 C 字符串的使用实现其功能。除此之外，另有其他个别的工具程序不允许使用 .（点号）meta 字符或方括号表达式来进行换行字符匹配。

3.2.2.2 后向引用

BRE 提供一种叫后向引用（backreferences）的机制，指的是“匹配于正则表达式匹配的先前的部分”。使用后向引用的步骤有两个。第一步是将子表达式包围在\(与\)里；单个模式里可包括至多 9 个子表达式，且可为嵌套结构。

下一步是在同一模式之后使用*digit*，*digit*指的是介于 1 至 9 的数字，指的是“匹配于第 *n* 个先前方括号内子表达式匹配成功的字符”。举例如下：

模式	匹配成功
\(ab\)\(cd\)[def]*\2\1	abcdcdab、abcdeeecdab、abcdddeeffcdab、...
\(why\).*\1	一行里重现两个 why
\([[:alpha:]_][[:alnum:]_]*\) = \1;	简易 C/C++ 赋值语句

后向引用在寻找重复字以及匹配引号时特别好用：

```
\(["']\).*\1        匹配以单引号或双引号括起来的字，例如 'foo' 或 "bar"
```

在这种方法下，就无须担心是单引号或是双引号先找到。

3.2.2.3 单个表达式匹配多字符

匹配多字符最简单的方法就是把它们一个接一个（连接）列出来，所以正则表达式**ab**匹配于 ab，**..**（两个点号）匹配于任意两个字符，而**[[:upper:]][[:lower:]]**则匹配于任意一个大写字符，后面接着任意一个小写字符。不过，将这些字符全列出来只有在简短的正则表达式里才好用。

虽然 .（点号）meta 字符与方括号表达式都提供了一次匹配一个字符的很好方式，但正则表达式真正强而有力的功能，其实是在修饰符（modifier）meta 字符的使用上。这类 meta 字符紧接在具有单个字符的正则表达式之后，且它们会改变正则表达式的含义。

最常用的修饰符为星号（*），表示“匹配 0 个或多个前面的单个字符”。因此，**ab*c** 表示的是“匹配 1 个 **a**、**0** 或多个 **b** 字符以及 **a c**”。这个正则表达式匹配的有 **ac**、**abc**、**abbc**、**abbbc** 等。

注意：“匹配 0 或多个”不表示“匹配其他的某一个……”，了解这一点是相当重要的。也就是说，正则表达式 **ab*c** 下，文本 aQc 是不匹配的 —— 即便是 aQc 里拥有 0 个 b 字符。相对的，以文本 ac 来说，b* 在 **ab*c** 里表述的是匹配 a 与 c 之间含有空字符串（null string）—— 意即长度为 0 的字符串（若你先前没遇过字符串长度为 0 的概念，这里可能得花点时间消化。总之，它在必要的时候会派得上用场，这在本章稍后会有所介绍）。

* 修饰符是好用的，但它没有限制，你不能用 * 表示“匹配三个字符，而不是四个字符”，而要使用一个复杂的方括号表达式，表明所需的匹配次数 —— 这也是件很麻烦的事。区间表达式（interval expressions）可以解决这类问题。就像 *，它们一样接在单个字符正则表达式后面，控制该字符连续重复几次即为匹配成功。区间表达式是将一个或两个数字，放在 \{ 与 \} 之间，有 3 种变化，如下：

\{*n*\}　前置正则表达式所得结果重现 *n* 次

\{*n*,\}　前置正则表达式所得的结果重现至少 *n* 次

\{*n*,*m*\}　前置正则表达式所得的结果重现 *n* 至 *m* 次

有了区间表达式，要表达像“重现 5 个 a”或是“重现 10 到 42 个 q”就变得很简单了，这两项分别是：**a\{5\}** 与 **q\{10,42\}**。

n 与 *m* 的值必须介于 0 至 RE_DUP_MAX（含）之间。RE_DUP_MAX 是 POSIX 定义的符号型常数，且可通过 getconf 命令取得。RE_DUP_MAX 的最小值为 255；不过部分系统允许更大值，在我们的 GNU/Linux 系统中，就遇到很大的值：

```
$ getconf RE_DUP_MAX
32767
```

3.2.2.4 文本匹配锚点

再介绍两个 meta 字符就能完成整个 BRE 的介绍了。这两个 meta 字符是脱字符号（^）与货币符号（$），它们叫做锚点（anchor），因为其用途在限制正则表达式匹配时，针对要被匹配字符串的开始或结尾处进行匹配（^ 在此处的用法与方括号表达式里的完全不同）。假定现在有一串要进行匹配的字：abcABCdefDEF，我们以表 3-4 列举匹配的范例。

表 3-4：正则表达式内锚点的范例

模式	是否匹配	匹配文本（粗体）/ 匹配失败的理由
ABC	是	居中的第 4、5 及 6 个字符：abc**ABC**defDEF
^ABC	否	限定匹配字符串的起始处
def	是	居中的第 7、8 及 9 个字符：abcABC**def**DEF
def$	否	限制匹配字符串的结尾处
[[:upper:]]\{3\}	是	居中的第 4、5 及 6 个字符：abc**ABC**defDEF
[[:upper:]]\{3\}$	是	结尾的第 10、11 及 12 个字符：abcDEFdef**DEF**
^[[:alpha:]]\{3\}	是	起始的第 1、2 及 3 个字符：**abc**ABCdefDEF

^ 与 $ 也可同时使用，这种情况是将括起来的正则表达式匹配整个字符串（或行）。有时 **^$** 这样的简易正则表达式也很好用，它可以用来匹配空的（empty）字符串或行列。例如在 grep 加上 -v 选项可以用来显示所有不匹配于模式的行，使用上面的做法，便能过滤掉文件里的空（empty）行。

例如，C 的源代码在经过处理后，变成了 #include 文件与 #define 宏时，这种用法就很有用了，因为这样一来你可以了解 C 编译器实际上看到的是什么（这是一种初级的调试法，但有时你就是要这么做）。扩展文件（expanded file）里头时常包含的空白或空行通常会比原始码更多，因此要排除空行只要：

```
$ cc -E foo.c | grep -v '^$' > foo.out      预先删除空行
```

^ 与 $ 仅在 BRE 的起始与结尾处具有特殊用途。在 BRE 下，ab^cd 里的 ^ 表示的，就是自身（^）；同样地，ef$gh 里的 $ 在这里表示的也就是字面上的货币字符。它也可能与其他正则表达式连用，例如 **\^** 与 **\$**，或是 **[$]**（注 3）。

3.2.2.5 BRE 运算符优先级

在数学表达式里，正则表达式的运算符具有某种已定义的优先级（precedence），指的是某个运算符（优先级较高）将比其他运算符先被处理。表 3-5 提供 BRE 运算符的优先级 —— 由高至低。

表 3-5：BRE 运算符优先级，由高至低

运算符	表示意义
[..] [==] [::]	用于字符排序的方括号符号
metacharacter	转义的 meta 字符

注 3：**[^]** 并非有效的正则表达式。请确认你了解是为什么。

表 3-5：BRE 运算符优先级，由高至低（续）

运算符	表示意义
[ ]	方括号表达式
\(\) *digit*	子表达式与后向引用
* \{ \}	前置单个字符重现的正则表达式
无符号（no symbol）	连续
^ $	锚点（Anchors）

3.2.3 扩展正则表达式

ERE（Extended Regular Expressions）的含义就如同其名字所示：拥有比基本正则表达式更多的功能。BRE 与 ERE 在大多数 meta 字符与功能应用上几乎是完全一致，但 ERE 里有些 meta 字符看起来与 BRE 类似，却具有完全不同的意义。

3.2.3.1 匹配单个字符

在匹配单个字符的情况下，ERE 本质上是与 BRE 是一致的。特别是像一般字符、用以转义 meta 字符的反斜杠，以及方括号表达式，这些行为模式都与先前提及的 BRE 相同。较有名的一个例外出现在 awk 里：其 \ 符号在方括号表达式内表示其他的含义。因此，如需匹配左方括号、连字符、右方括号或是反斜杠，你应该用 `[\[\-\]\\]`。这是使用上的经验法则。

3.2.3.2 后向引用不存在

ERE 里是没有后向引用的（注 4）。圆括号在 ERE 里具特殊含义，但和 BRE 里的使用又有所不同（这点稍后会介绍）。在 ERE 里，\(与 \) 匹配的是字面上的左括号与右括号。

3.2.2.3 匹配单个表达式与多个正则表达式

ERE 在匹配多字符这方面，与 BRE 有很明显的不同。不过，在 * 的处理上和 BRE 是相同的（注 5）。

注 4： 这在 grep 与 egrep 命令下有不同的影响，这并非正则表达式匹配能力的问题，只是 UNIX 的一种处理方式而已。

注 5： 有一个例外是，* 作为 ERE 的第一个字符是“未定义”的，而在 BRE 中它是指“符合字面的 *”。

区间表达式可用于ERE中，但它们是写在花括号里（{}），且不需前置反斜杠字符。因此我们先前的例子“要刚好重现5个a”以及“重现10个至42个q”，写法分别为：**a{5}**与**q{10,42}**。而**\{**与**\}**则可用以匹配字面上的花括号。当在ERE里{找不到匹配的}时，POSIX特意保留其含义为“未定义（undefined）状态”。

ERE另有两个meta字符，可更细腻地处理匹配控制：

?	匹配于0个或一个前置正则表达式
+	匹配于1个或多个前置正则表达式

你可以把?想成是“可选用的”，也就是说，匹配前置正则表达式的文本，要么出现，要么没出现。举例来说，与**ab?c**匹配的有ac与abc，就这两者！（与**ab*c**相较之下，后者匹配于中间有任意个b）。

+字符在概念上与* meta字符类似，不过前置正则表达式要匹配的文本在这里至少得出现一次。因此，**ab+c**匹配于abc、abbc、abbbc，但不匹配于ac。你当然可以把**ab+c**的正则表达式形式换成**abb*c**；无论如何，当前置正则表达式很复杂时，使用+可以少打一点字，当然也减少了打错字的几率！

3.2.3.4 交替

方括号表达式易于表示“匹配于此字符，或其他字符，或…”，但不能指定“匹配于这个序列（sequence），或其他序列（sequence），或…”。要达到后者的目的，你可以使用交替（alternation）运算符，即垂直的一条线，或称为管道字符（|）。你可以简单写好两个字符序列，再以|将其隔开。例如**read|write**匹配于read与write两者、**fast|slow**匹配于fast与slow两者。你还可以使用多个该符号：**sleep|doze|dream|nod off|slumber**匹配于5个表达式。

|字符为ERE运算符里优先级最低的。因此，左边会一路扩展到运算符的左边，一直到一个前置|的字符，或者是到另一个正则表达式的起始。同样地，|的右边也是一路扩展到运算符的右边，一直到后续的|字符，或是到整个正则表达式的结尾。这部分将在下一节探讨。

3.2.3.5 分组

你应该已经发现，在ERE里，我们已提到运算符是被应用到“前置的正则表达式”。这是因为有圆方括号（(...)）提供分组功能，让接下来的运算符可以应用。举例来说，**(why)+**匹配于一个或连续重复的多个why。

在必须用到交替（alternation）时，分组的功能就特别好用了（也是必需的）。它可以让你用以构建复杂并较灵活性的正则表达式。举例来说，**[Tt]he (CPU|computer) is** 指的就是：在 The（或 the）与 is 之间，含有 CPU 或 computer 的句子。要特别注意的一点是，圆括号在这里是 meta 字符，而非要匹配的输入文本。

将重复（repetition）运算符与交替功能结合时，分组功能也是一定用得到的。**read|write+** 指的是正好一个 read，或是一个 write 后面接着任意数个 e 字符（writee、writeee 等），比较有用的模式应该是 **(read|write)+**，它指的是：有一个或重现多个 read，或者一个或重现多个 write。

当然，**(read|write)+** 所指的字符串中间，不允许有空白。**((read|write)[[:space:]]*)+** 的正则表达式看起来虽然比较复杂，不过也比较实际些。乍看之下，这可能会搞不清楚，不过若把这些组成部分分隔开来看，其实就不难理解了。图 3-1 为图解说明。

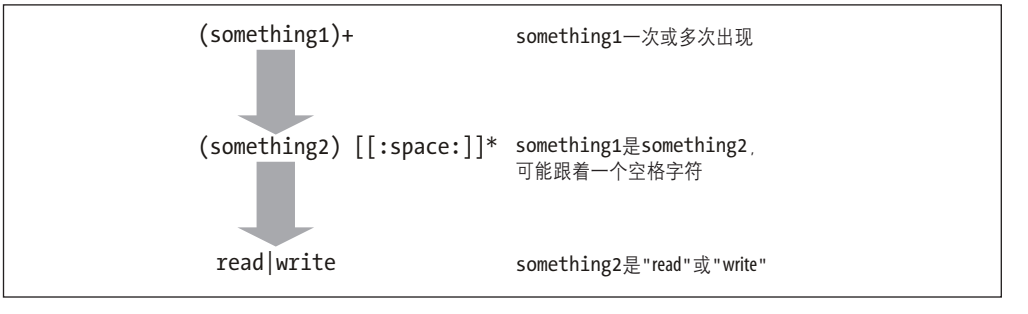

图 3-1：读取一个复杂的正则表达式

结论就是：这个单个正则表达式是用以匹配多个连续出现的 read 或是 write，且中间可能被空白字符隔开。

在 [[:space:]] 之后使用 * 是一种判断调用（judgment call）。使用一个 * 而非 +，此匹配可以取得在行（或字符串）结尾的单词。但也可能可以匹配中间完全没有空白的单词。运用正则表达式时常会需要用到这样的判断调用（judgment call）。该如何构建正则表达式，需要根据输入的数据以及这些数据的用途而定。

最后要说的是：当你将交替（alternation）操作结合 ^ 与 $ 锚点字符使用时，分组就非常好用了。由于 | 为所有运算符中优先级最低的，因此正则表达式 **^abcd|efgh$** 意思是“匹配字符串的起始处是否有 a b c d，或者字符串结尾处是否有 e f g h”，这和 **^(abcd|efgh)$** 不一样，后者表示的是“找一个正是 abcd 或正是 efgh 的字符串”。

3.2.3.6 停驻文本匹配

^ 与 $ 在这里表示的意义与BRE里的相同：将正则表达式停驻在文本字符串（或行）的起始或结尾处。不过有个明显不同的地方就是：在ERE里，^ 与 $ 永远是meta字符。所以，像 **ab^cd** 与 **ef$gh** 这样的正则表达式仍是有效的，只是无法匹配到任何东西，因为 ^ 前置了文本，与 $ 后面的文本，会让它们分别无法匹配到“字符串的开始”与“字符串结尾”。正如其他的meta字符一般，它们在方括号表达式中的确失去了它们特殊的意义。

3.2.3.7 ERE 运算符的优先级

在ERE里运算符的优先级和BRE一样。表 3-6 由高至低列出了ERE运算符的优先级。

表 3-6：ERE 运算符优先级，由高至低

运算符	含义
`[..] [= =] [: :]`	用于字符对应的方括号符号
\metacharacter	转义的meta字符
`[]`	方括号表达式
`()`	分组
`* + ? {}`	重复前置的正则表达式
无符号（no symbol）	连续字符
`^ $`	锚点（Anchors）
`|`	交替（Alternation）

3.2.4 正则表达式的扩展

很多程序提供正则表达式语法扩展。这类扩展大多采取反斜杠加一个字符，以形成新的运算符。类似POSIX BRE里 **\(...\)** 与 **\{...\}** 的反斜杠。

最常见的扩展为 \< 与 \> 运算符，分别匹配“单词（word）”的开头与结尾。单词是由字母、数字及下划线组成的。我们称这类字符为单词组成（word-constituent）。

单词的开头要么出现在行起始处，要么出现在第一个后面紧跟一个非单词组成（nonword-constituent）字符的单词组成（word-constituent）字符。同样的，单词的结尾要么出现在一行的结尾处，要么出现在一个非单词组成字符之前的最后一个单词组成字符。

实际上，单词的匹配其实相当直接易懂。正则表达式 **\<chop** 匹配于 `use chopsticks`，

但 eat a lambchop 则不匹配；同样地，**chop\>** 则匹配于第二个字符串，第一个则不匹配。需要特别注意的一点是：在 **\<chop\>** 的表达式下，两个字符串都不匹配。

虽然 POSIX 标准化的只有 ex 编辑器，但在所有商用 UNIX 系统上，ed、ex 以及 vi 编辑器都支持单词匹配，而且几乎已是标准配备。GNU/Linux 与 BSD 系统上附带的克隆程序（“clone” version）也支持单词匹配，还有 emacs、vim 与 vile 也是。除此之外，通常 grep 与 sed 也会支持，不过最好在系统里再通过手册页（manpage）确认一下。

可处理正则表达式的标准工具的GNU版本，通常还支持许多额外的运算符，如表 3-7 所示。

表 3-7：额外的 GNU 正则表达式运算符

运算符	含义
\w	匹配任何单词组成字符，等同于 **[[:alnum:]_]**
\W	匹配任何非单词组成字符，等同于 **[^[:alnum:]_]**
\<\>	匹配单词的起始与结尾，如前文所述
\b	匹配单词的起始或结尾处所找到的空字符串。这是 \< 与 \> 运算符的结合
	注意：由于 awk 使用 \b 表示后退字符，因此 GNU awk（gawk）使用 \y 表示此功能
\B	匹配两个单词组成字符之间的空字符串
\' \`	分别匹配 emacs 缓冲区的开始与结尾。GNU 程序（还有 emacs）通常将它们视为与 ^ 及 $ 同义

虽然 POSIX 明白表示了 NUL 字符无须是可匹配的，但 GNU 程序则无此限制。若 NUL 字符出现在输入数据里，则它可以通过 .meta 字符或方括号表达式来匹配。

3.2.5 程序与正则表达式

有两种不同的正则表达式风格是经年累月的历史产物。虽然egrep风格的扩展正则表达式在 UNIX 早期开发时就已经存在了，但 Ken Thompson 并不觉得有必要在 ed 编辑器里使用这样全方位的正则表达式（由于 PDP-11 的小型地址空间、扩展正则表达式的复杂度，以及实际应用时大部分的编辑工作使用基本正则表达式已足够了，这样的决定其实相当合理）。

ed 的程序代码后来成了 grep 的基础（grep 为 ed 命令中 g/*re*/p 的缩写，意即全局性匹配 *re*，并将其打印）。ed 的程序代码后来也成为初始构建 sed 的根基。

就在 pre-V7 时期，Al Aho 创造了 egrep，Al Aho 是贝尔实验室的研究人员，他为正则表达式匹配与语言解析的研究奠定了基础。egrep 里的核心匹配程序代码，日后也被 awk 的正则表达式拿来使用。

\< 与 \> 运算符起源于滑铁卢大学的 Rob Pike、Tom Duff、Hugh Redelmeier，以及 David Tilbrook 所修改的 ed 版本（Rob Pike 是这些运算符的发明者之一）。Bill Joy 在 UCB 时，便将这两个运算符纳入 ex 与 vi 编辑器，自那时起，它就广为流传。区间表达式源起于 Programmer's Workbench UNIX（注 6），之后通过 System III 以及此后的 System V，特别将其取出用于商用 UNIX 系统上。表 3-8 列出的是各种 UNIX 程序与其使用的正则表达式。

表 3-8：UNIX 程序及其正则表达式类型

类型	grep	sed	ed	ex/vi	more	egrep	awk	lex
BRE	•	•	•	•	•			
ERE						•	•	•
\< \>	•	•	•	•	•			

lex 是一个很特别的工具，通常是用于语言处理器中的词法分析器的构建。虽然已纳入 POSIX，但我们不会在这里进一步讨论，因为它与 Shell 脚本无关。less 与 pg 虽然不是 POSIX 的一部分，但它们也支持正则表达式。有些系统会有 page 程序，它本质上和 more 是相同的，只是在每个充满屏幕的输出画面之间，会清除屏幕。

正如我们在本章开头所提到的：要（试图）解决多个 grep 的矛盾，POSIX 决定以单个 grep 程序解决。POSIX 的 grep 默认行为模式使用的是 BRE。加上 -E 选项则它使用 ERE，及加上 -F 选项，则它使用 fgrep 的固定字符串匹配算法。因此，真正地遵循 POSIX 的程序应以 grep -E ... 取代 egrep ...。不过，因为所有的 UNIX 系统确实拥有它，且可能已经有许多年了，所以我们继续在自己的脚本中使用它。

最后要注意的一点就是：通常，awk 在其扩展正则表达式里不支持区间表达式。直至 2005 年，各种不同厂商的 awk 版本也并非全面支持区间表达式。为了让程序具有可移植性，若需要在 awk 程序里匹配大方括号，应该以反斜杠转义它，或将它们括在方括号表达式里。

注 6：Programmer's Workbench（PWB）UNIX 是用在 AT&T 里以支持电信交换软件开发的变化版。它也可以用于商业用途。

3.2.6 在文本文件里进行替换

很多Shell脚本的工作都从通过grep或egrep取出所需的文本开始。正则表达式查找的最初结果，往往就成了要拿来作进一步处理的“原始数据（raw data）”。通常，文本替换（text substitution）至少需要做一件事，就是将一些字以另一些字取代，或者是删除匹配行的某个部分。

一般来说，执行文本替换的正确程序应该是sed —— 流编辑器（Stream Editor）。sed的设计就是用来以批处理的方式而不是交互的方式来编辑文件。当你知道要做好几个变更 —— 不管是对一个还是数个文件时，比较简单的方式是将这些变更部分写到一个编辑中的脚本里，再将此脚本应用到所有必须修改的文件。sed存在的目的就在这里（虽然你也可以使用ed或ex编辑脚本，但用它们来处理会比较麻烦，而且用户通常不会记得要存储原先的文件）。

我们发现，在Shell脚本里，sed主要用于一些简单的文本替换，所以我们先从它开始。接下来我们还会提供其他的后台数据，并说明sed的功能，特意不在这里提到太多细节，是因为sed所有的功能全都写在《sed & awk》（O’Reilly）这本书里了，该书已列入参考书目。

GNU sed可从*ftp://ftp.gnu.org/gnu/sed/*获取。这个版本拥有相当多有趣的扩展，且已配备使用手册，还附带软件。GNU的sed使用手册里有一些好玩的例子，还包括与众不同的程序测试工具组。可能最令人感到不可思议的是：UNIX dc任意精确度计算程序（arbitrary-precision calculator）竟是以sed所写成的！

当然绝佳的sed来源就是*http://sed.sourceforge.net/*了。这里有连接到两个sed FAQ文件的链接。第一个是*http://www.dreamwvr.com/sed-info/sed-faq.html*，第二个比较旧的FAQ则是*ftp://rtfm.mit.edu/pub/faqs/editor-faq/sed*。

3.2.7 基本用法

你可能会常在管道（pipeline）中间使用sed，以执行替换操作。做法是使用s命令 —— 要求正则表达式寻找，用替代文本（replacement text）替换匹配的文本，以及可选用的标志：

```
sed 's/:.*//' /etc/passwd |          删除第一个冒号之后的所有东西
 sort -u                             排序列表并删除重复部分
```

sed

语法

```
sed [ -n ] 'editing command' [ file ... ]
sed [ -n ] -e 'editing command' ... [ file ... ]
sed [ -n ] -f script-file ... [ file ... ]
```

用途

为了编辑它的输入流，将结果生成到标准输出，而非以交互式编辑器的方式来编辑文件。虽然sed的命令很多，能做很复杂的工作，但它最常用的还是处理输入流的文本替换，通常是作为管道的一部分。

主要选项

`-e 'editing command'`

将*editing command*使用在输入数据上。当有多个命令需应用时，就必须使用-e了。

`-f script-file`

自*script-file*中读取编辑命令。当有多个命令需要执行时，此选项相当有用。

`-n`

不是每个最后已修改结果行都正常打印，而是显示以p指定（处理过的）的行。

行为模式

读取每个输入文件的每一行，假如没有文件的话，则是标准输入。以每一行来说，sed会执行每一个应用到输入行的*editing command*。结果会写到标准输出（默认状态下，或是显示地使用p命令及-n选项）。若无-e或-f选项，则sed会把第一个参数看作是要使用的*editing command*。

在这里，/字符扮演定界符（delimiter）的角色，从而分隔正则表达式与替代文本(replacement text)。在本例中，替代文本是空的（空字符串null string），实际上会有效地删除匹配的文本。虽然/是最常用的定界符，但任何可显示的字符都能作为定界符。在处理文件名称时，通常都会以标点符号字符作为定界符（例如分号、冒号或逗点）：

```
find /home/tolstoy -type d -print   |    寻找所有目录
 sed 's;/home/tolstoy/;/home/lt/;'  |    修改名称；注意：这里使用分号作为定界符
  sed 's/^/mkdir /'                 |    插入mkdir命令
   sh -x                                 以Shell跟踪模式执行
```

上述脚本是将/home/tolstoy目录结构建立一份副本在/home/lt下（可能是为备份而

做的准备)。(find 命令在第 10 章将会介绍，在本例中它的输出是 /home/tolstoy 底下的目录名称列表：一行一个目录。)这个脚本使用了产生命令(generating commands)的手法，使命令内容成为 Shell 的输入。这是一个功能很强且常见的技巧，但却很少人这么用（注 7)。

3.2.7.1 替换细节

先前已经提过，除斜杠外还可以使用其他任意字符作为定界符；在正则表达式或替代文本里，也能转义定界符，不过这么做可能会让命令变得很难看懂：

```
sed 's/\/home\/tolstoy\//\/home\/lt\//'
```

在前面的3.2.2 节里，我们讲到POSIX 的BRE时，已说明后向引用在正则表达式里的用法。sed 了解后向引用，而且它们还能用于替代文本中，以表示“从这里开始替换成匹配第 *n* 个圆括号里子表达式的文本”：

```
$ echo /home/tolstoy/ | sed 's;\(/home\)/tolstoy/;\1/lt/;'
/home/lt/
```

sed将\1替代为匹配于正则表达式的 **/home**部分。在这里的例子中，所有的字符都表示它自己，不过，任何正则表达式都可括在 \(与 \) 之间，且后向引用最多可以用到 9 个。

有些其他字符在替代文本里也有特殊含义。我们已经提过需要使用反斜杠转义定界符的情况。当然，反斜杠字符本身也可能需要转义。最后要说明的是：& 在替代文本里表示的意思是“从此点开始替代成匹配于正则表达式的整个文本”。举例来说，假设处理 Atlanta Chamber of Commerce 这串文本，想要在广告册中修改所有对该城市的描述：

```
mv atlga.xml atlga.xml.old
sed 's/Atlanta/&, the capital of the South/' < atlga.xml.old > atlga.xml
```

(作为一个跟得上时代的人，我们在所有的地方都尽可能使用 XML，而不是昂贵的专用字处理程序)。这个脚本会存储一份原始广告小册的备份，做这类操作绝对有必要 —— 特别是还在学习如何处理正则表达式与替换（substitutions）的时候，然后再使用 sed 进行变更。

如果要在替代文本里使用 & 字符的字面意义，请使用反斜杠转义它。例如，下面的小脚本便可以转换 DocBook/XML 文件里字面上的反斜杠，将其转换为 DocBook 里对应的 \

```
sed 's/\\/\&bsol;/g'
```

注 7： 这个脚本有小瑕疵，它无法处理目录名称含有空格的情况。这个问题是可以解决的，只是要有点小技巧，这部分我们将在第 10 章介绍。

在 s 命令里以 g 结尾表示的是：全局性（global），意即以“替代文本取代正则表达式中每一个匹配的”。如果没有设置 g，sed 只会取代第一个匹配的。这里来比较看看有没有设置 g 所产生的结果：

```
$ echo Tolstoy reads well. Tolstoy writes well. > example.txt   输入样本
$ sed 's/Tolstoy/Camus/' < example.txt                         没有设置 g
Camus reads well. Tolstoy writes well.
$ sed 's/Tolstoy/Camus/g' < example.txt                        设置了 "g"
Camus reads well. Camus writes well.
```

鲜为人知的是（可以用来吓吓朋友）：你可以在结尾指定数字，指示第 *n* 个匹配出现才要被取代：

```
$ sed 's/Tolstoy/Camus/2' < example.txt    仅替代第二个匹配者
Tolstoy reads well. Camus writes well.
```

到目前为止，我们讲的都是一次替换一个。虽然可以将多个 sed 实体以管道（pipeline）串起来，但是给予 sed 多个命令是比较容易的。在命令行上，这是通过 -e 选项的方式来完成。每一个编辑命令都使用一个 -e 选项：

```
sed -e 's/foo/bar/g' -e 's/chicken/cow/g' myfile.xml > myfile2.xml
```

不过，如果你有很多要编辑的项目，这种形式就很恐怖了。所以有时，将编辑命令全放进一个脚本里，再使用 sed 搭配 -f 选项会更好：

```
$ cat fixup.sed
s/foo/bar/g
s/chicken/cow/g
s/draft animal/horse/g
...
$ sed -f fixup.sed myfile.xml > myfile2.xml
```

你也可以构建一个结合 -e 与 -f 选项的脚本；脚本为连续的所有编辑命令，依次提供所有选项。此外，POSIX 也允许使用分号将同一行里的不同命令隔开：

```
sed 's/foo/bar/g ; s/chicken/cow/g' myfile.xml > myfile2.xml
```

不过，许多商用 sed 版本还不支持此功能，所以如果你很在意可移植性的问题，请避免使用此法。

ed 与其先驱 ex 与 vi 一样，Sed 会记得在脚本里遇到的最后一个正则表达式 —— 不管它在哪。通过使用空的正则表达式，同一个正则表达式可再使用：

```
s/foo/bar/3        更换第三个 foo
s//quux/           现在更换第一个
```

你可以考虑一个 html2xhtml.sed 的简单脚本，它将 HTML 转换为 XHTML。该脚本会将标签转换成小写，然后更改
 标签为自我结束形式
：

```
s/<H1>/<h1>/g                          斜杠为定界符
s/<H2>/<h2>/g
s/<H3>/<h3>/g
s/<H4>/<h4>/g
s/<H5>/<h5>/g
s/<H6>/<h6>/g
s:</H1>:</h1>:g                        冒号为定界符，因数据内容里已有斜杠
s:</H2>:</h2>:g
s:</H3>:</h3>:g
s:</H4>:</h4>:g
s:</H5>:</h5>:g
s:</H6>:</h6>:g
s:</[Hh][Tt][Mm][LL]>:<html>:g
s:</[Hh][Tt][Mm][Ll]>:</html>:g
s:<[Bb][Rr]>:<br/>:g
...
```

像这样的脚本就可以自动执行大量的HTML转XHTML了，XHTML为标准化的、以XML为主的 HTML 版本。

3.2.8 sed 的运作

sed 的工作方式相当直接。命令行上的每个文件名会依次打开与读取。如果没有文件，则使用标准输入，文件名“-”（单个破折号）可用于表示标准输入。

sed读取每个文件，一次读一行，将读取的行放到内存的一个区域 —— 称之为模式空间（pattern space）。这就像程序语言里的变量一样：内存的一个区域在编辑命令的指示下可以修改，所有编辑上的操作都会应用到模式空间的内容。当所有操作完成后，sed会将模式空间的最后内容打印到标准输出，再回到开始处，读取另一个输入行。

这一工作过程如图3-2所示。脚本使用两条命令，将The UNIX System替代为The UNIX Operating System。

3.2.8.1 打印与否

-n 选项修改了 sed 的默认行为。当提供此选项时，sed 将不会在操作完成后打印模式空间的最后内容。反之，若在脚本里使用p，则会明白地将此行显示出来。举例来说，我们可以这样模拟 grep：

```
sed -n '/<HTML>/p' *.html          仅显示 <HTML> 这行
```

虽然这个例子很简单，但这个功能在复杂的脚本里非常好用。如果你使用一个脚本文件，可通过特殊的首行来打开此功能：

```
#n                                 关闭自动打印
/<HTML>/p                          仅打印含 <HTML> 的行
```

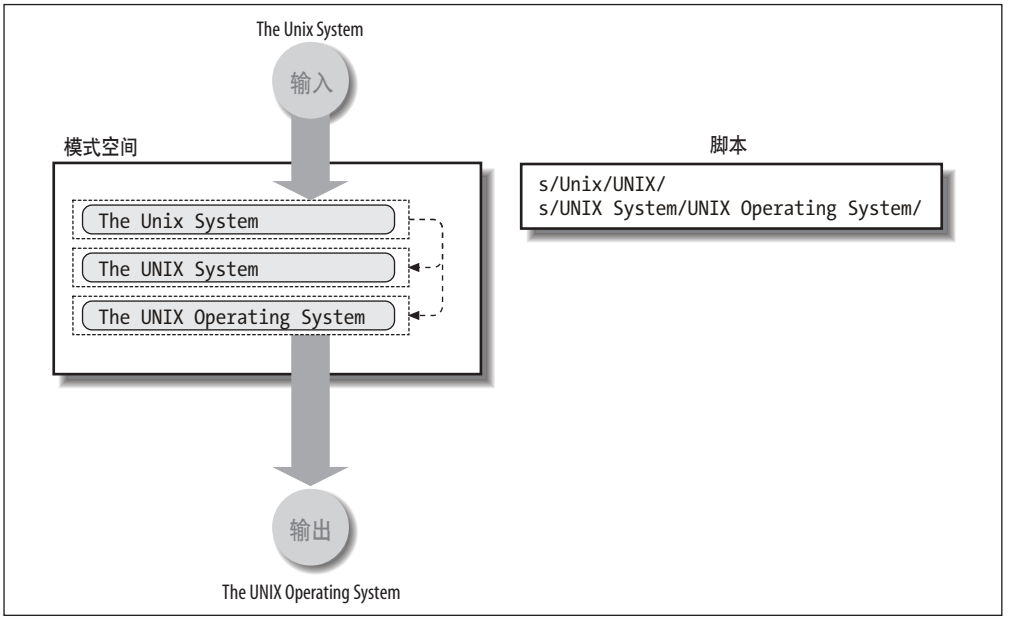

图3-2：在sed脚本中的命令改变了模式空间

在Shell中，与很多其他UNIX脚本式语言一样：#是注释的意思。sed注释必须出现在单独的行里，因为它们是语法型命令，意思是：它们是什么事也不做的命令。虽然POSIX指出，注释可以放在脚本里的任何位置，但很多旧版sed仅允许出现在首行，GNU sed则无此限制。

3.2.9 匹配特定行

如前所述，sed默认地会将每一个编辑命令（editing command）应用到每个输入行。而现在我们要告诉你的是：还可以限制一条命令要应用到哪些行，只要在命令前置一个地址（address）即可。因此，sed命令的完整形式就是：

```
address command
```

以下为不同种类的地址：

正则表达式

将一模式放置到一条命令之前，可限制命令应用于匹配模式的行。可与s命令搭配使用：

```
/oldfunc/ s/$/# XXX: migrate to newfunc/     注释部分源代码
```

s命令里的空模式指的是“使用前一个正则表达式”：

```
/Tolstoy/ s//& and Camus/g          提及两位作者
```

最终行

符号 $（就像在 ed 与 ex 里一样）指“最后一行”。下面的脚本指的是快速打印文件的最后一行：

```
sed -n '$p' "$1"                    引号里为指定显示的数据
```

对 sed 而言，“最后一行”指的是输入数据的最后一行。即便是处理多个文件，sed 也将它们视为一个长的输入流，且 $ 只应用到最后一个文件的最后一行（GNU 的 sed 具有一个选项，可使地址分开地应用到每个文件，见其说明文档）。

行编号

可以使用绝对的行编号作为地址。稍后将有介绍。

范围

可指定行的范围，仅需将地址以逗点隔开：

```
sed -n '10,42p' foo.xml             仅打印 10～42 行
sed '/foo/,/bar/ s/baz/quux/g'      仅替换范围内的行
```

第二个命令为“从含有 **foo** 的行开始，再匹配是否有 **bar** 的行，再将匹配后的结果中，有 **baz** 的全换成 quux”（像 ed、ex 这类的检阅程序，或是 vi 内的冒号命令提示模式下，都认识此语法）。

这种以逗点隔开两个正则表达式的方式称为范围表达式（range expression）。在 sed 里，总是需要使用至少两行才能表达。

否定正则表达式

有时，将命令应用于不匹配于特定模式的每一行，也是很有用的。通过将!加在正则表达式后面就能做到，如下所示：

```
/used/!s/new/used/g                 将没有 used 的每个行里所有的 new 改成 used
```

POSIX 标准指出：空白（whitespace）跟随在 ! 之后的行为是“未定义的（unspecified）”，并建议需提供完整可移植性的应用软件，不要在 ! 之后放置任何空白字符，这明显是由于某些 sed 的古董级版本仍无法识别它。

例 3-1 说明的是使用绝对的行编号作为地址的用法，这里是以 sed 展现的 head 程序简易版。

例 3-1：使用 sed 的 head 命令

```
# head --- 打印前 n 行
#
# 语法：  head N file

count=$1
sed ${count}q "$2"
```

当你引用head 10 foo.xml之后，sed会转换成sed 10q foo.xml。q命令要求sed马上离开；不再读取其他输入，或是执行任何命令。后面的7.6.1节里，我们将介绍如何让这个脚本看起来更像真正的head命令。

迄今为止，我们看到的都是sed以/字符隔开模式以便查找。在这里，我们要告诉你如何在模式内使用不同的定界符：这通过在字符前面加上一个反斜杠实现：

```
$ grep tolstoy /etc/passwd                                    显示原始行
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
$ sed -n '\:tolstoy: s;;Tolstoy;p' /etc/passwd                改变定界符
Tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

本例中，以冒号隔开要查找的模式，而分号则扮演s命令的定界符角色（编辑上的操作其实不重要，我们的重点是介绍如何使用不同的定界符）。

3.2.10 有多少文本会改动

有个问题我们一直还没讨论到：有多少文本会匹配?事实上，这应该包含两个问题。第二个问题是：从哪儿开始匹配？执行简单的文本查找，例如使用grep或egrep时，则这两个问题都不重要，你只要知道是否有一行是匹配的，若有，则看看那一行是什么。至于在这个行里，是从哪儿开始匹配，或者它扩展到哪里，已经不重要了。

但如果你要使用sed执行文本替换，或者要用awk写程序，这两个问题的答案就变得非常重要了（如果你每天都在文本编辑器内工作，这也算是个重要议题，只是本书的重点不在文本编辑器）。

这两个问题的答案是：正则表达式匹配可以匹配整个表达式的输入文本中最长的、最左边的子字符串。除此之外，匹配的空（null）字符串，则被认为是比完全不匹配的还长（因此，就像我们先前所解释的，正则表达式：**ab*c**匹配文本ac，而b*则成功地匹配于a与c之间的null字符串）。再者，POSIX标准指出：“完全一致的匹配，指的是自最左边开始匹配、针对每一个子模式、由左至右，必须匹配到最长的可能字符串”。（子模式指的是在ERE下圆括号里的部分。为此目的，GNU的程序通常也会在BRE里以\(...\)提供此功能）。

如果sed要替代由正则表达式匹配的文本，那么确定该正则表达式匹配的字不会太少或太多就非常重要了。这里有个简单例子：

```
$ echo Tolstoy writes well | sed 's/Tolstoy/Camus/'    使用固定字符串
Camus writes well
```

当然，sed可以使用完整的正则表达式。这里就是要告诉你，了解“从最长的最左边（longest leftmost）”规则有多的重要：

```
$ echo Tolstoy is worldly | sed 's/T.*y/Camus/'      试试正则表达式
Camus                                                结果呢 ?
```

很明显，这里只是要匹配 Tolstoy，但由于匹配会扩展到可能的最长长度的文本量，所以就一直找到 worldly 的 y 了！你需要定义的更精确些：

```
$ echo Tolstoy is worldly | sed 's/T[[:alpha:]]*y/Camus/'
Camus is worldly
```

通常，当开发的脚本是要执行大量文本剪贴和排列组合时，你会希望谨慎地测试每样东西，确认每个步骤都是你要的 —— 尤其是当你还在学习正则表达式里的微妙变化的阶段的时候。

最后，正如我们所见到的，在文本查找时有可能会匹配到 null 字符串。而在执行文本替代时，也允许你插入文本：

```
$ echo abc | sed 's/b*/1/'          替代第一个匹配成功的
1abc
$ echo abc | sed 's/b*/1/g'         替代所有匹配成功的
1a1c1
```

请留意，**b*** 是如何匹配在 abc 的前面与结尾的 null 字符串。

3.2.11 行 v.s. 字符串

了解行（line）与字符串（string）的差异是相当重要的。大部分简易程序都是处理输入数据的行，像 grep 与 egrep，以及 sed 大部分的工作（99%）都是这样。在这些情况下，不会有内嵌的换行字符出现在将要匹配的数据中，^ 与 $ 则分别表示行的开头与结尾。

然而，对可应用正则表达式的程序语言，例如 awk、Perl 以及 Python，所处理的就多半是字符串。若每个字符串表示的就是独立的一行输入，则 ^ 与 $ 仍旧可分别表示行的开头与结尾。不过这些程序语言，其实可以让你使用不同的方式来标明每条输入记录的定界符，所以有可能单独的输入“行”（记录）里会有内嵌的换行字符。这种情况下，^ 与 $ 无法匹配内嵌的换行字符；它们只用来表示字符串的开头与结尾。当你开始使用可程序化的软件工具时，这一点，请牢记在心。

3.3 字段处理

很多的应用程序，会将数据视为记录与字段的结合，以便于处理。一条记录（record）指的是相关信息的单个集合，例如以企业来说，记录可能含有顾客、供应商以及员工等数据，以学校机构来说，则可能有学生数据。而字段（field）指的就是记录的组成部分，例如姓、名或者街道地址。

3.3.1 文本文件惯例

由于UNIX鼓励使用文本型数据，因此系统上最常见的数据存储类型就是文本了，在文本文件下，一行表示一条记录。这里要介绍的是在一行内用来分隔字段的两种惯例。首先是直接使用空白（whitespace），也就是用空格键（space）或制表（tab）键：

```
$ cat myapp.data
# model     units sold     salesperson
xj11        23             jane
rj45        12             joe
cat6        65             chris
...
```

本例中，#字符起始的行表示注释，可忽略（这是一般的习惯，注释行的功能相当好用，不过软件必须可忽略这样的行才行）。各字段都以任意长度的空格（space）或制表（Tab）字符隔开。第二种惯例是使用特定的定界符来分隔字段，例如冒号：

```
$ cat myapp.data
# model:units sold:salesperson
xj11:23:jane
rj45:12:joe
cat6:65:chris
...
```

两种惯例都有其优缺点。使用空白作为分隔时，字段内容就最好不要有空白（若你使用制表字符（Tab）作分隔，字段里有空格是不会有问题的，但这么做视觉上会混淆，因为你在看文件时，很难马上分辨出它们的不同）。反过来说，若你使用显式的定界符，那么该定界符也最好不要成为数据内容。请你尽可能小心地选择定界符，让定界符出现在数据内容里的可能性降到最低或不存在。

注意：这两种方式最明显的不同，便是在处理多个连续重复的定界符之时。使用空白(whitespace)分隔时，通常多个连续出现的空格或制表字符都将看作一个定界符。然而，若使用的是特殊字符分隔，则每个定界符都隔开一个字段，例如，在myapp.data的第二个版本里使用的两个冒号字符（“::”）则会分隔出一个空的字段。

以定界符分隔字段最好的例子就是 /etc/passwd 了，在这个文件里，一行表示系统里的一个用户，每个字段都以冒号隔开。在本书中，很多地方都会以 /etc/passwd 为例，因为在系统管理工作中，很多时候都是在处理这个文件。如下是该文件的典型例子：

```
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

该文件含有 7 个字段，分别是：

1. 用户名称
2. 加密后的密码（如账号为停用状态，此处为一个星号，或是若加密后的密码文件存储于另外的 /etc/shadow 里，则这里也可能是其他字符）。
3. 用户 ID 编号。
4. 用户组 ID 编号。
5. 用户的姓名，有时会另附其他相关数据（办公室号码、电话等）。
6. 根目录。
7. 登录的 Shell。

某些UNIX 工具在处理以空白界定字段的文件时，做得比较好，有些则是以定界符分隔字段比较好，更有其他的工具两种方式都能处理得当，这部分我们稍后会介绍。

3.3.2 使用 cut 选定字段

cut 命令是用来剪下文本文件里的数据，文本文件可以是字段类型或是字符类型。后一种数据类型在遇到需要从文件里剪下特定的列时，特别方便。请注意：一个制表字符在此被视为单个字符（注 8）。

举例来说，下面的命令可显示系统上每个用户的登录名称及其全名：

```
$ cut -d : -f 1,5 /etc/passwd          取出字段
root:root                              管理者账号
...
tolstoy:Leo Tolstoy                    实际用户
austen:Jane Austen
camus:Albert Camus
...
```

通过选择其他字段编号，还可以取出每个用户的根目录：

```
$ cut -d : -f 6 /etc/passwd            取出根目录
/root                                  管理账号
...
/home/tolstoy                          实际用户
/home/austen
/home/camus
...
```

通过字符列表做剪下操作有时是很方便的。例如，你只要取出命令 ls -l 的输出结果中的文件权限字段：

注 8： 这可通过 expand 与 unexpand 改变其定义。见 *expand*(1)手册页。

Cut

语法

```
cut -c list [ file ... ]
cut -f list [ -d delim ] [ file ... ]
```

用途

从输入文件中选择一或多个字段或者一组字符，配合管道（pipeline），可再做进一步处理。

主要选项

`-c list`

以字符为主，执行剪下的操作。*list*为字符编号或一段范围的列表（以逗点隔开），例如1,3,5-12,42。

`-d delim`

通过-f选项，使用*delim*作为定界符。默认的定界符为制表字符（Tab）。

`-f list`

以字段为主，作剪下的操作。*list*为字段编号或一段范围的列表（以逗点隔开）。

行为模式

剪下输入字符中指定的字段或指定的范围。若处理的是字段，则定界符隔开的即为各字段，而输出时字段也以给定的定界符隔开。若命令行没有指定文件，则读取标准输入。见正文中的范例。

警告

于POSIX系统下，cut识别多字节字符。因此，“字符（character）”与“字节（byte）”意义不同。详细内容见*cut*(1)的手册页。

有些系统对输入行的大小有所限制，尤其是含有多字节字符（multibyte characters）时，这点请特别留意。

```
$ ls -l | cut -c 1-10
total 2878
-rw-r--r--
drwxr-xr-x
-r--r--r--
-rw-r--r--
...
```

不过这种用法比使用字段的风险要大。因为你无法保证行内的每个字段长度总是一样的。一般来说，我们偏好以字段为基础来提取数据。

3.3.3 使用 join 连接字段

join命令可以将多个文件结合在一起，每个文件里的每条记录，都共享一个键值（key），键值指的是记录中的主字段，通常会是用户名称、个人姓氏、员工编号之类的数据。举例来说，有两个文件，一个列出所有业务员销售业绩，一个列出每个业务员应实现的业绩：

join

语法

join [*options* ...] *file1 file2*

用途

以共同一个键值，将已存储文件内的记录加以结合。

主要选项

-1 *field1*

-2 *field2*

标明要结合的字段。-1 *field1* 指的是从 *file1* 取出 *field1*，而 -2 *field2* 指的则为从 *file2* 取出 *field2*。字段编号自 1 开始，而非 0。

-o *file.field*

输出 *file* 文件中的 *field* 字段。一般的字段则不打印。除非使用多个 -o 选项，即可显示多个输出字段。

-t *separator*

使用 *separator* 作为输入字段分隔字符，而非使用空白。此字符也为输出的字段分隔字符。

行为模式

读取 *file1* 与 *file2*，并根据共同键值结合多笔记录。默认以空白分隔字段。输出结果则包括共同键值、来自 *file1* 的其余记录，接着 *file2* 的其余记录（指除了键值外的记录）。若 *file1* 为 -，则 join 会读取标准输入。每个文件的第一个字段是用来结合的默认键值；可以使用 -1 与 -2 更改之。默认情况下，在两个文件中未含键值的行将不打印（已有选项可改变，见 *join*(1)手册页）。

警告

-1 与 -2 选项的用法是较新的。在较旧的系统上，可能得用：-j1 *field1* 与 -j2 *field2*。

```
$ cat sales                          显示sales文件
# 业务员数据                         注释说明
# 业务员 量
joe     100
jane    200
herman  150
chris   300

$ cat quotas                         显示quotas文件
# 配额
# 业务员 配额
joe     50
jane    75
herman  80
chris   95
```

每条记录都有两个字段：业务员的名字与相对应的量。在本例中，列与列之间有多个空白，从而可以排列整齐。

为了让 join 运作得到正确结果，输入文件必须先完成排序。例 3-2 里的程序 merge-sales.sh 即为使用 join 结合两个文件。

例 3-2：merge-sales.sh

```
#! /bin/sh

# merge-sales.sh
#
# 结合配额与业务员数据

# 删除注释并排序数据文件
sed '/^#/d' quotas | sort > quotas.sorted
sed '/^#/d' sales  | sort > sales.sorted

# 以第一个键值作结合，将结果产生至标准输出
join quotas.sorted sales.sorted

# 删除缓存文件
rm quotas.sorted sales.sorted
```

首先，使用 sed 删除注释，然后再排序个别文件。排序后的缓存文件成为 join 命令的输入数据，最后删除缓存文件。这是执行后的结果：

```
$ ./merge-sales.sh
chris 95 300
herman 80 150
jane 75 200
joe 50 100
```

3.3.4 使用 awk 重新编排字段

awk 本身所提供的功能完备，已经是一个很好用的程序语言了。我们在第 9 章会好好地介绍该语言的精髓。虽然 awk 能做的事很多，但它主要的设计是要在 Shell 脚本中发挥所长：做一些简易的文本处理，例如取出字段并重新编排这一类。本节，我们将介绍 awk 的基本概念，随后你看到这样的“单命令行程序（one-liners）”就会比较了解了。

3.3.4.1 模式与操作

awk 的基本模式不同于绝大多数的程序语言。它其实比较类似于 sed：

```
awk 'program' [ file ... ]
```

awk 读取命令行上所指定的各个文件（若无，则为标准输入），一次读取一条记录（行）。再针对每一行，应用程序所指定的命令。awk 程序基本架构为：

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

pattern 部分几乎可以是任何表达式，但是在单命令行程序里，它通常是由斜杠括起来的 ERE。*action* 为任意的 awk 语句，但是在单命令行程序里，通常是一个直接明了的 print 语句（稍后有范例说明）。

pattern 或是 *action* 都能省略（当然，你不会两个都省略吧？）。省略 *pattern*，则会对每一条输入记录执行 *action*；省略 *action* 则等同于 { print }，将打显示整条记录（稍后将会介绍）。大部分单命令行程序为这样的形式：

```
... | awk '{ print some-stuff }' | ...
```

对每条记录来说，awk 会测试程序里的每个 *pattern*。若模式值为真（例如某条记录匹配于某正则表达式，或是一般表达式计算为真），则 awk 便执行 *action* 内的程序代码。

3.3.4.2 字段

awk 设计的重点就在字段与记录上：awk 读取输入记录（通常是一些行），然后自动将各个记录切分为字段。awk 将每条记录内的字段数目，存储到内建变量 NF。

默认以空白分隔字段 —— 例如空格与制表字符（或两者混用），像 join 那样。这通常就足够使用了，不过，其实还有其他选择：你可以将 FS 变量设置为一个不同的值，也就可以变更 awk 分隔字段的方式。如使用单个字符，则该字符出现一次，即分隔出一个字段（像 cut -d 那样）。或者，awk 特别之处就是：也可以设置它为一个完整的 ERE，这种情况下，每一个匹配在该 ERE 的文本都将视为字段分隔字符。

如需字段值，则是搭配 $ 字符。通常 $ 之后会接着一个数值常数，也可能是接着一个表达式，不过多半是使用变量名称。列举几个例子如下：

```
awk '{ print $1 }'                    打印第 1 个字段（未指定 pattern）
awk '{ print $2, $5 }'                打印第 2 与第 5 个字段（未指定 pattern）
awk '{ print $1, $NF }'               打印第 1 个与最后一个字段（未指定 pattern）
awk 'NF > 0   { print $0 }'           打印非空行（指定 pattern 与 action）
awk 'NF > 0'                          同上（未指定 action，则默认为打印）
```

比较特别的字段是编号 0：表示整条记录。

3.3.4.3 设置字段分隔字符

在一些简单程序中，你可以使用 -F 选项修改字段分隔字符。例如，显示 /etc/passwd 文件里的用户名称与全名，你可以：

```
$ awk -F: '{ print $1, $5 }' /etc/passwd      处理 /etc/passwd
root root                                     管理账号
...
tolstoy Leo Tolstoy                           实际用户
austen Jane Austen
camus Albert Camus
...
```

-F 选项会自动地设置 FS 变量。请注意，程序不必直接参照 FS 变量，也不用必须管理读取的记录并将它们分割为字段：awk 会自动完成这些事。

你可能已经发现，每个输出字段是以一个空格来分隔的 —— 即便是输入字段的分隔字符为冒号。awk 的输入、输出分隔字符用法是分开的，这点与其他工具程序不同。也就是说，必须设置 OFS 变量，改变输出字段分隔字符。方式是在命令行里使用 -v 选项，这会设置 awk 的变量。其值可以是任意的字符串。例如：

```
$ awk -F: -v 'OFS=**'  '{ print $1, $5 }' /etc/passwd     处理 /etc/passwd
root**root                                                管理者账号
...
tolstoy**Leo Tolstoy                                      实际用户
austen**Jane Austen
camus**Albert Camus
...
```

稍后就可以看到设置这些变量的其他方式。或许那些方式更易于理解，根据你的喜好而定。

3.3.4.4 打印行

就像我们已经所介绍过的：大多数时候，你只是想把选定的字段显示来，或者重新安排其顺序。简单的打印可使用print语句做到，只要提供给它需要打印的字段列表、变量或字符串即可：

```
$ awk -F: '{ print "User", $1, "is really", $5 }' /etc/passwd
User root is really root
...
User tolstoy is really Leo Tolstoy
User austen is really Jane Austen
User camus is really Albert Camus
...
```

简单明了的print语句，如果没有任何参数，则等同于print $0，即显示整条记录。

以刚才的例子来说，在混合文本与数值的情况下，多半会使用awk版本的printf语句。这和先前在2.5.4节所提及的Shell（与C）版本的printf语句相当类似，这里就不再重复。以下是把上例修改为使用printf语句的用法：

```
$ awk -F: '{ printf "User %s is really %s\n", $1, $5 }' /etc/passwd
User root is really root
...
User tolstoy is really Leo Tolstoy
User austen is really Jane Austen
User camus is really Albert Camus
...
```

awk的print语句会自动提供最后的换行字符，就像Shell层级的echo与printf那样，然而，如果使用printf语句，则用户必须要通过\n转义序列的使用自己提供它。

注意：请记得在print的参数间用逗点隔开！否则，awk将连接相邻的所有值：

```
$ awk -F: '{ print "User" $1 "is really" $5 }' /etc/passwd
Userrootis reallyroot
...
Usertolstoyis reallyLeo Tolstoy
Useraustenis reallyJane Austen
Usercamusis reallyAlbert Camus
...
```

这样将所有字符串连在一起应该不是你要的。忘了加上逗点，这是个常见又难找到的错误。

3.3.4.5 起始与清除

BEGIN与END这两个特殊的“模式”，它们提供awk程序起始（startup）与清除（cleanup）操作。常见于大型awk程序中，且通常写在个别文件里，而不是在命令行上：

```
BEGIN     {  起始操作程序代码(startup code) }

pattern1  { action1 }

pattern2  { action2 }

END       {  清除操作程序代码(cleanup code) }
```

BEGIN与END的语句块是可选用的。如需设置，习惯上（但不必须）它们应分别置于awk程序的开头与结尾处。你可以有数个BEGIN与END语句块，awk会按照它们出现在程序的顺序来执行：所有的BEGIN语句块都应该放在起始处，而所有END语句块也应放在结尾。以简单程序来看，BEGIN可用来设置变量：

```
$ awk 'BEGIN { FS = ":" ; OFS = "**" }      使用BEGIN设置变量
> { print $1, $5 }' /etc/passwd             被引用的程序继续到第二行
root**root
...
tolstoy**Leo Tolstoy                        输出，如前
austen**Jane Austen
camus**Albert Camus
...
```

警告： POSIX标准中描述了awk语言及其程序选项。POSIX awk是构建在所谓的“新awk”上，首度全球发布是在1987年的System V Release 3.1版，且在1989年的System V Release 4版中稍作修正。

但是，直到2005年底，Solaris的/bin/awk仍然还是原始的、1979年的awk V7版！在Solaris系统上，你应该使用/usr/xpg4/bin/awk，或参考第9章，使用awk自由下载版中的一个。

3.4 小结

如需从输入的数据文件中取出特定的文本行，主要的工具为grep程序。POSIX采用三种不同grep变体：grep、egrep与fgrep的功能，整合为单个版本，通过不同的选项，分别提供这三种行为模式。

虽然你可以直接查找字符串常数，但是正则表达式能提供一个更强大的方式，描述你要找的文本。大部分的字符在匹配时，表示的是自己本身，但有部分其他字符扮演的是meta字符的角色，也就是指定操作，例如“匹配0至多个的……”、“匹配正好10个的……”等。

POSIX的正则表达式有两种：基本正则表达式（BRE）以及扩展正则表达式（ERE）。哪个程序使用哪种正则表达式风格，是根据长时间的实际经验，由POSIX制定规格，简化

到只剩两种正则表达式的风格。通常，ERE比BRE功能更强大，不过不见得任何情况下都是这样。

正则表达式对于程序执行时的locale环境相当敏感；方括号表达式里的范围应避免使用，改用字符集，例如`[[:alnum:]]`较佳。另外，许多GNU 程序都有额外的 meta 字符。

`sed`是处理简单字符串替换（substitution）的主要工具。在我们的经验里，大部分的Shell脚本在使用`sed`时几乎都是用来作替换的操作，我们特意在这里不介绍`sed`所能提供的其他任务，是因为已经有《sed & awk》这本书（已列于参考书目中），它会介绍更多相关信息。

“从最左边开始，扩展至最长（longest leftmost)”，这个法则描述了匹配的文本在何处匹配以及匹配扩展到多长。在使用`sed`、`awk`或其他交互式文本编辑程序时，这个法则相当重要。除此之外，一行与一个字符串之间的差异也是核心观念。在某些程序语言里，单个字符串可能包含数行，那种情况下，`^` 与 `$` 指的分别是字符串的开头与结尾。

很多时候，在操作上可以将文本文件里的每一行视为一条单个记录，而在行内的数据则包括字段。字段可以被空白或是特殊定界符分隔，且有许多不同的UNIX 工具可处理这两种数据。`cut` 命令用以剪下选定的字符范围或字段，`join` 则是用来结合记录中具有共同键值的字段的文件。

`awk` 多半用于简单的“单命令行程序”，当你想要只显示选定的字段，或是重新安排行内的字段顺序时，就是`awk`派上用场的时候了。由于它是编程语言，即使是在简短的程序里，它也能发挥其强大的功能、灵活性与控制能力。