第6学时　模 式 匹 配

在上个学时中，我们介绍了如何从文件中读取数据的方法。懂得这个方法后，再加上标量、数组和运算符方面的知识，就可以准备对该数据进行操作，以便做你想要做的任何事情。有时，文件中的数据没有采用便于使用的格式进行格式化，它不能使用简单的 split函数对数据进行分割，或者有的文件行包含了你不感兴趣的数据，你想通过编辑将它删除。

你必须具备一种能力，来识别输入数据流中的模式。根据这些模式来选择数据，并且对数据进行编辑，使之变成比较容易使用的格式。 Perl的工具中有一个工具可以用来执行这项任务，即正则表达式。在本学时的课文中，正则表达式与模式几乎可以互换使用。

正则表达式本身几乎是一种语言，它是用于描述要匹配模式的正式方法。在本学时中，我们将要介绍一些关于这种模式匹配的语言。

在线文档对Perl使用的完整的正则表达式语言进行了更加深入的（但是比较扼要的）描述。你可以查看 Perl中包含的perlre文档。这个问题涉及的内容非常广泛，因此有人出版了整整一本书来介绍正则表达式。 Perl界大力推荐的这本书名叫“ Mastering Regular Expressions ”，它是由Jeffery E.F. Friedl撰写的，1997年出版。它全面介绍了正则表达式，并且对 Perl给予了很大的关注。

正则表达式也可以用于其他编程语言，如 TCL、JavaScript、Pathon和C语言。UNIX操作系统的许多实用程序也使用正则表达式。 Perl正好配有一组非常丰富的正则表达式，这与其他操作系统的情况非常相似，学习这些正则表达式不仅有助于你在 Perl中的应用，而且可以用于其他语言。在本学时中，你将学习：

- 如何创建简单的正则表达式。
- 如何使用正则表达式进行模式匹配。
- 如何使用正则表达式来编辑字符串。

6.1 简单的模式

在Perl中，模式被括在模式匹配运算符中间，有时该运算符采用 m//的形式。下面就是一种简单的模式：

```
m/simon/
```

上面这个模式依次与字母S-i-m-o-n相匹配。但是究竟在什么地方才能找到 Simon呢？以前我们讲过， Perl变量$_常常用于Perl需要默认值的时候。模式匹配是根据 $_来进行的，除非你告诉Perl用别的方式来进行匹配（后面我们将要介绍）。因此前面的模式可以在标量变量 $_中寻找S-i-m-o-n。

如果由m//规定的模式可以在变量 $_中的任何地方找到，那么匹配运算符返回真。这样，

能够看到匹配模式的正常位置是在条件表达式中，如下所示：

```
if (m/Piglet/) {
    : # the pattern "Piglet" is in $_
}
```

在这个模式中，除非字符是个元字符，否则每个字符均与自己相匹配。大多数“标准”字符均与自己相匹配，这些字符包括 A至Z、a至z和数字。元字符是指改变了模式匹配运行特性的那些字符。下面是元字符的列表：

```
^ $ ( ) \ | @ [ { ? . + *
```

下面我们很快就要介绍元字符能够做些什么。在你的模式中，如果想要匹配元字符的原义值，只需要在元字符的前面加上一个反斜杠即可，如下所示：

```
m/I won \$10 at the fair/;      # The $ is treated as a literal dollar-sign.
```

前面我们已经讲过模式匹配运算符通常用 m//来表示。实际上，可以用你想要的任何其他字符来代替斜杠，如下面这个例子所示：

```
if (m,Waldo,) { print "Found Waldo.\n"; }
```

在许多情况下，当模式中包含斜杠（ /）时且模式的结尾则可能与模式内的斜杠相混淆，可用另一个字符来代替它，因此括号里面的斜杠的前面必须加上反斜杠，如下所示：

```
if (m/\/usr\/local\/bin\/hangman/) { print "Found the hangman game!" }
```

可以编写下面这个代码，使上面的代码更加容易阅读：

```
if (m:/usr/local/bin/hangman:) { print "Found the hangman game!" }
```

如果将模式括起来的字符（称为界限符）是斜杠，那么编写模式匹配代码时也可以不带 m。因此，也可以将 m/Cheetos写成/Cheetos/。通常情况下，除非需要使用不是斜杠（ //）的其他界限符，否则，可以只使用斜杠而不使用 m来编写模式匹配代码。

变量也可以用在正则表达式中。如果在正则表达式中看到一个标量变量， Perl首先计算该标量，然后查看正则表达式。这个功能使你能够动态地创建正则表达式。下面这个 if语句中的正则表达式是根据用户输入创建的：

```
$pat=<STDIN>;  chomp $pat;
$_="The phrase that pays";
if (/$pat/) {    #Look for the user's pattern
    print "\"$_\" contains the pattern $pat\n";
}
```

联机手册页和其他文档中的正则表达式有时称为 RE或regexp。为了清楚起见，在本书中将继续将它们称为正则表达式。

匹配的规则

当你开始在Perl中编写正则表达式时，应该知道它必须遵循几条规则。不过，规则并不多，大多数规则在你理解它们之后才具有更大的意义。这些规则是：

- 通常情况下，模式匹配从目标字符串的左边开始，然后逐步向右边进行匹配。
- 如果并且只有当整个模式能够用于与目标字符串相匹配时，模式匹配才返回真（在任何

上下文中均如此）。

- 目标字符串中第一个能够匹配的字符串首先进行匹配。正则表达式不会漏掉某一个能够匹配的字符串，而去寻找另一个更远的字符串来进行匹配。
- 进行第一次最大字符数量的匹配。你的正则表达式可能迅速找到一个匹配的模式，然后设法尽可能延伸能够匹配的字符范围。正则表达式是“贪婪的”，也就是说，它会尽可能多地寻找能够匹配的字符。

6.2 元字符

在下面的所有例子中，与模式相匹配的文本部分用下划线标出。请记住，即使目标字符串中只有一部分与正则表达式相匹配，整个目标字符串也可以说是匹配的。下划线标记用于帮助说明究竟哪些部分是匹配的。

阅读下列各节内容是非常重要的，但是，如果有些内容你无法立即理解，请不要担心。你很快就会理解的。这些元字符的应用将会有所明确。

6.2.1 一个简单的元字符

第一个元字符是圆点（.）。在正则表达式中，圆点用于匹配除了换行符外的任何单个字符。例如，在模式/p.t/中，‘.’用于匹配任何单个字符。这个模式用于匹配pot、pat、pit、carpet、python和pup_tent。‘.’要求存在一个字符，但是不能有更多的字符。因此，该模式不能与 apt 相匹配（p与t之间没有任何字符），也不能与expect相匹配（pt之间的字符太多）。

6.2.2 非输出字符

前面我们讲过，若要将元字符纳入正则表达式，应该在字符前面加上一个反斜杠，使它失去“元”的含义，如下所示：

```
/\^\$/;    # A literal caret and dollar sign
```

当普通字符的前面加上反斜杠后，它就变成了元字符。正如你在第 2学时中看到的字符串那样，有些字符的前面加上反斜杠后，它在字符串中就具备了特殊的含义。在正则表达式中，所有这些字符几乎代表相同的值，如表6-1所示：

表6-1 特殊字符

字 符	匹配的字符
\n	换行符
\r	回车符
\t	制表符
\f	换页符

6.2.3 通配符

到现在为止，模式中的所有字符与它们要匹配的目标字符串之间存在一种一对一的关系。例如，在/Simon/中，s与S匹配，i与i匹配，m与m匹配，等等。通配符是一种元字符，它告诉正则表达式有多少字符需要匹配。通配符可以放在任何单个字符或一组字符的后面（后面我们将要详细介绍这方面的内容）。

最简单的通配符是+元字符。+用于使前面的字符与后面的字符至少匹配一次，也可以任意次地进行匹配，并且仍然拥有匹配的表达式。因此，/do+g/将能够与下面的字符串匹配：

hounddog
hotdog
doogie howser
doooooogdoog
但是不能与下面的字符串匹配：
badge (因为没有o)
doofus (因为没有g)
Doogie （因为D与d不同）
pagoda （因为d、o和g的顺序不对）

与元字符+的作用类似的是*。元字符*使得前面的字符可以进行0次或多次匹配。换句话说，模式/t*/可以进行任意次的匹配，但是，如果没有匹配的字符存在，这也没有问题。因此，/car*t/将能够与下面的字符匹配：

carted
cat
carrrt
但是不能与下面的字符匹配：
carrot (多了一个字符o)
carl (模式中的t不是可有可无的)
caart （多出来的a不能匹配）

下一个元字符是？。元字符？用于使前面的字符进行0次或一次匹配（但是不能超过一次）。因此，模式/c?ola/用于对c进行匹配，如果c存在的话。然后对o、l和a进行匹配。实际上，该模式可以对带有ola在内的任何字符串进行匹配。如果ola的前面是一个c，那么该字符串也是匹配的。

元字符？与*之间的区别是：模式/c?ola/可以匹配cola和ola，但是不能与ccola匹配。多出来的c需要进行两次匹配。模式/c*ola/可以匹配cola、ola和ccola，因为c可以根据需要重复匹配任意次，而不只是0次或一次。

如果对一个模式进行0次、一次或许多次匹配不能满足你的需要，那么Perl允许根据你需要的具体次数为你进行匹配，方法是使用花括号{}。花括号的格式如下：

```
pat{n, m}
```

这里的n是匹配的最小次数，m是匹配的最大次数，pat是你试图量化匹配的字符或字符组。可以省略n，也可以省略m，但是不能同时省略n和m。请看下面这些例子：

/x{5，10}/ x至少出现5次，但是不超过10次。
/x{9，}/ x至少出现9次，也可能出现更多次。
/x{0，4}/ x最多出现4次，也可能根本不出现。
/x{8}/ x必须正好出现8次，不能多，也不能少。

正则表达式中常用的一个通配符是.*。可以用它来匹配任何东西，通常是你感兴趣的其他两样东西之间的任何东西。例如/first.*last/。这个模式设法匹配单词first，再匹配它后面的任何东西，然后匹配单词last。请观察/first.*last/是如何匹配下面的字符串的：

first then last

The good players get picked first, the bad last.

The first shall be last, and the last shall be first.

请仔细观察上面第3行中的匹配过程。这个匹配过程首先从单词 first开始。接着对单词last进行匹配，然后继续进行匹配，直到第二次出现单词 last。这时，通配符*遵循“匹配规则”这一节中列出的第4条规则，即它要匹配数量最大的字符串，并且仍然要完成该匹配过程。许多情况下，匹配数量最大的字符串并不是你想要进行的操作，因此Perl提供了另一个解决方案，称为最小数量匹配，这在perlre手册页中有详细的描述。

6.2.4 字符类

正则表达式中的另一个常用做法是要求匹配“这些字符中的任何字符”。如果你想要匹配数字，那么能够编写一个匹配“0~9的任何数字”的模式将是非常好的。或者，如果你要搜索一个名字列表，想要匹配Von Beethoven与von Beethoven，那么使用能够匹配“v或者V”的模式，将对你是有帮助的。

Perl的正则表达式拥有这样一个工具，它称为字符类。若要编写一个字符类，可以用方括号[ ]将这些字符括起来。进行匹配时，字符类中的所有字符被视为单个字符。在一个字符类中，可以设定字符的范围（在范围有意义的时候），方法是在上限与下限之间加一个连字符。下面是一些例子：

字 符 类	说 明
[abcde]	用于匹配a、b、c、d或e中的任何一个字符
[a-e]	与上面相同。用于匹配a、b、c、d或e中的任何一个字符
G	用于匹配大写字母G或小写字母g
[0-9]	用于匹配一个数字
[0-9]+	用于顺序匹配一个或多个数字
[A-Za-z]{5}	用于匹配任何一组5个字母字符
[*！@ #$%&()]	用于匹配这些符号中的任何一个

最后一个例子非常有意思，因为在字符类中，大多数通配符会失去它们的“通配符性质”，换句话说，它们的运行特性将类似其他任何一个普通字符。因此，* 实际上代表一个普通的*字符。

如果插入记号（^）作为字符类中的第一个字符，该字符类将变为无效。也就是说，该字符类可以匹配不在该字符类中的任何单个字符。如下面的例子所示：

```
/[^A-Z]/;        # Matches non-uppercase-alphabetic characters.
```

由于]、^和-等字符都是字符类中的特殊字符，因此，如果按照字符的原义来匹配字符类中的这些字符，就要使用某些规则。若要匹配字符类中的原义字符 ^，必须确保它不出现在字符类中的第一个字符的位置上；若要匹配字符]，你既必须将它放在字符类中的第一个字符位置上，也可以在它的前面加上一个反斜杠（例如 /[abc\ ]]/）；若要将一个原义连字符（-）放在字符类中，你只需要将它放在字符类中的第一个字符位置上，或者在它的前面放一个反斜杠。

Perl包含了某些常用字符类的快捷方式。它们用反斜杠和通配符来表示，如表 6-2所示。

下面是一些例子：

```
/\d{5}/;         # Matches 5 digits
/\s\w+\s/;       # Matches a group of word-characters surrounded by white space
```

表6-2 特殊字符类

模 式	用于匹配
\w	一个单词字符，与[a-zA-z0-9_]相同
\W	一个非单词字符（与\w相反）
\d	一个数字，与[0-9]相同
\D	一个非数字
\s	一个白空间字符，与[\t\f\r\n]相同
\S	一个非白空间字符

不过请注意，上面的最后一个例子不一定匹配一个单词，它也可以匹配一个前后是空格的下划线。同时，并不是所有单词都可以被最后一个模式来匹配的，它们的前后必须加上白空间，并且“ don ’ t ”之类的单词不能匹配，因为它包含一个省字号。本学时的后面部分的内容还要进一步介绍用于单词匹配的更好的模式。

6.2.5 分组和选择

有时在正则表达式中，你可能想要知道是否找到了一组模式中的任意一个模式。例如，这个字符串包含dogs还是cats？正则表达式对这个问题的解决办法称为选择。当可能的匹配项之间用一个 | 字符隔开时，正则表达式中就出现了选择，如下例所示：

```
if (/dogs|cats/) {
    print "\$_ contains a pet\n";
}
```

选择可能是非常有趣的，但是，当想要匹配许多类似的东西时，它也可能很麻烦。例如，如果你想要匹配frog、bog、log、flog或clog等单词时，可以试用/frog|bog|log|flog|clog/这个表达式，不过它包含许多重复的选择操作。而你真正想要进行的操作只是比较字符串的第一部分，如下所示：

```
/fr|b|l|fl|clog/;      # Doesn't QUITE work.
```

上面这个例子不一定能够运行，因为 Perl没有办法知道选择是你要进行匹配的一个对象，而og是要匹配的另一个对象。为了解决这个问题，可以使用 Perl的正则表达式，用括号将模式的各个部分组合起来，如下所示：

```
/(fr|b|fl|cl)og/;
```

可以对括号进行嵌套，使一个组中包含另一个组。例如，也可以将上面的表达式写成/(fr|b| (f|c) |) og/。

在列表上下文中，匹配运算符返回括号中匹配的表达式的各个部分的一个列表。每个加括号的值都是列表的返回值，如果模式不包含括号，则返回 1。请看下面这个例子：

```
$_="apple is red";
($fruit, $color)=/(.*)\sis\s(.*)/;
```

在上面这个代码段中，该模式先对任意对象（作为一个组）进行匹配，然后对白空间进行匹配，再对单词is进行匹配，然后匹配更多的白空间，再对任意对象（也作为一个组）进行匹配。这两个分组的表达式返回左边的列表，并赋予 $fruit和$color。

6.2.6 位置通配符

最后两个通配符（相信你可能认为通配符是没有止境的）是位置通配符。可以使用位置

通配符来告诉正则表达式，你要查找的模式的准确位置是在字符串的开头，还是在字符串的结尾。

第一个位置通配符是插入记号（^）。正则表达式开头的插入记号告诉正则表达式只匹配一行开头的字符。例如，/^video/只匹配单词video，如果它出现在一行的开头的话。

与它相对应的通配符是美元符号（$）。正则表达式结尾处的美元符号能够使模式只匹配一行结尾的字符。例如/earth$/用于匹配earth,不过它只能位于行尾。

模　式	作　用
/^Help/	只匹配以Help开头的行
/^Frankly.*darn$/	用于匹配以Frankly开头和以darn结尾的行。它们中间的所有字符也进行匹配
/^hysteria$/	用于匹配只包含单词hysteria的行
/^$/	用于匹配一行的开头，紧接着匹配该行的结尾。它只用于匹配空行
/^/	用于匹配带有开头字符的行（所有行）。/$/的作用也相同

6.3 替换

仅仅查找字符串中的模式和输入的信息行是不够的，有时也需要修改数据。方法之一（当然不是惟一的方法）是使用替换运算符 s///。它的句法如下：

```
s/searchpattern/replacement/;
```

替换运算符用于默认搜索 $_，找出searchpattern，并且用replacement来替换整个匹配的正则表达式。该运算符返回匹配的数量或进行替换的数量，如果没有进行任何匹配，则返回 0。下面是一个例子：

```
$_="Our house is in the middle of our street".
s/middle/end/;          # Is now: Our house is in the end of our street
s/in/at/;               # Is now: Our house is at the end of our street.
if (s/apartment/condo/) {
    #  This code isn't reached, see note.
}
```

在这个例子中，替换按照你希望的那样进行。单词 middle替换成end，in替换成at。但是if语句运行失败了，因为单词apartment没有出现在$_中，因此无法替换。

替换运算符也可以使用非斜杠（/）的界限符，使用的方法与匹配运算符相同。只需要直接在s的后面加上你想要的界限符即可，如下所示：

```
s#street#avenue#;
```

6.4 练习：清除输入数据

当你试图更改数据的时候，常常会进行上面这个例子中的“盲目”替换，即替换是进行了，但是不检查退出状态。更改数据是从用户或文件中取出没有完全按照你的要求进行格式化的数据，然后对数据重新进行格式化。程序清单 6-1显示了一个例程，用于将你在地球上的体重转换成月球上的体重，这可以展示数据操作时的情况。

使用文本编辑器，键入程序清单 6-2中的程序，并将它保存为 Moon。务必按照第 1学时中的说明使该程序成为可执行程序。

当完成上述操作后，键入下面这个命令行，设法运行该程序：

```
perl Moon
```

程序清单6-1显示了某些输出示例。

程序清单6-1 月球体重转换程序的输出示例

```
1:   $  perl Moon
2:   Your weight:  150lbs
3:   Your weight on the moon: 25.00005 lbs
4:   $ perl Moon
5:   Your weight: 90 kg
6:   Your weight on the moon: 90.9090 lbs
```

程序清单6-2 你的月球体重转换程序

```
1:   #!/usr/bin/perl -w
2:
3:   print "Your weight:";
4:   $_=<STDIN>;  chomp;
5:   s/^\s+//;   # Remove leading spaces, if any.
6:
7:   if (m/(lbs|kg|kilograms|pounds?)/i) {
8:           if (s/\s*(kg|kilograms?).*//) {
9:                   $_*=2.2;
10:           } else {
11:                  s/\s*(lbs|pounds?).*//;
12:           }
13:   }
14:   print "Your weight on the moon: ", $_*.16667, " lbs\n";
```

第1行：这一行包含到达解释程序的路径（可以修改该路径，使之适合你的系统的需要）和开关-w。请始终使警告特性处于激活状态。

第3~4行：这几行用于提示用户输入他的体重，将输入的值赋予 $_，并且用chomp命令删除换行符。请记住，如果没有设定其他变量，那么 chomp将改成$_。

第5行：模式/^\s+/对该行的开头的白空间进行匹配。它没有列出任何替换字符串，因此，匹配该模式的$_部分被删除了。

第7行：如果在用户的输入中发现了计量单位，那么该 if代码块便删除该计量单位，并在适当的情况下对它进行转换。

第8~9行：模式/ \s*(kgs?|kilograms?) / i对白空间进行匹配，然后对 kg或kilograms（每个元素的结尾都有一个选项 s）进行匹配。这意味着如果输入中包含 kg或kg（没有空格），那么它就被删除。如果该模式被发现和删除了， $_中留下的数据将与2.2相乘，或者转换成磅。

第11行：否则，从$_中删除lbs或pounds（并删除前导白空间）。

第14行：$_中的重量（已经转换成磅）乘以 1/6，然后输出。

6.5 关于模式匹配的其他问题

现在你已经能够对 $_进行模式匹配，并且懂得替换的基本概念，因此可以学习更多的功能。为了使正则表达式的运行做到真正有效，必须对 $_之外的变量进行匹配，并且进行更加复杂的替换，还要使用适用于（但不是只能用于）正则表达式的 Perl函数。

6.5.1 对其他变量进行操作

在程序清单6-2中，用户输入的重量存放在$_中，并用替换运算符和匹配运算符进行操作。

不过该程序清单存在一个问题，那就是 $_并不是用来存放“重量”的最佳变量名。对于初学者来说，$_并不十分直观，在不经意中，$_也许被改变了。

一般来说，将数据长期存放在 $_中是非常危险的，最终你会感到非常恼火。Perl的许多运算符都使用$_作为默认参数，其中有些运算符也会修改 $_。$_是Perl的通用变量，如果试图将一个值长时间存放在 $_中（尤其是当你学习了第8学时的内容后），最终将会导致某些错误。

在程序清单6-2中使用变量$weight就比较好。如果要对非$_的变量使用匹配运算符和替换运算符，则必须将它们与该变量连接起来。为此可以使用连接运算符 =~，如下所示：

```
$weight="185 lbs";
$weight=~s/ lbs//;      # Do substitution against $weight
```

=~运算符并不进行赋值，它只是取出右边的运算符，并使它对左边的变量进行操作。整个表达式拥有的值与使用$_时所拥有的值是相同的，正如你在下面这个例子中看到的那样：

```
$poem="One fish, two fish, red fish";
$n=$poem=~m/fish/;      # $n is true, if $poem has fish7
```

6.5.2 修饰符与多次匹配

到现在为止，你看到的所有正则表达式都是区分大小写字母的。也就是说，在模式匹配中，大写字符与小写字符是不一样的。如果在匹配单词时不考虑它们是大写字母还是小写字母，那么需要使用下面的代码：

```
/[Mm][Aa][Cc][Bb][Ee][Tt][Hh]/;
```

这个例子看上去不仅很笨，而且很容易出错，因为很容易在键入大写和小写字母对时发生错误。替换运算符（s///）和匹配运算符（m//）能够在匹配正则表达式时不考虑大小写字母，如果匹配项的后面跟一个字母 i的话。

```
/macbeth/i;
```

上面这个例子可以对 Macbeth进行匹配，无论它是使用大写字母、小写字母还是大小写混合字母（MaCbEtH）。

用于匹配和替换的另一个修饰符是全局匹配修饰符 g。正则表达式（或替换）的匹配操作不是一次完成的，它要重复通过整个字符串，第一次匹配后，立即进行下一次匹配（或替换）。

在列表上下文中，全局匹配修饰符可使匹配代码返回一个放在括号中的正则表达式的各个部分的列表：

```
$_="One fish, two frog, red fred, blue foul";
@F=m/\W(f\w\w\w)/g;
```

该模式首先匹配一个非单词字符，然后匹配字母 f，接着匹配4个单词字符。字母f和4个单词用括号分组。该表达式被计算后，变量 @F将包含4个元素，即fish、frog、fred和foul。

在标量上下文中，修饰符 g使得匹配操作迭代通过整个字符串，为每个匹配操作返回真，当不再进行更多的匹配操作时，返回假。现在请看下面这个代码：

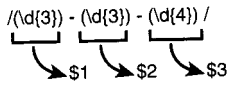
```
$letters=0;
$phrase="What's my line?";
while($phrase=~/\w/g) {
    $letters++;
}
```

上面这个代码段使用匹配运算符（ //），它带有标量上下文中的修饰符 g。while循环这个条件提供了标量上下文。该模式用于匹配一个单词字符。 While循环将继续运行(并且letters 被递增)，直到匹配代码返回假为止。当该代码段运行完成后， $letters的结果将是11。

在第9学时中，我们将要展示更加有效的对字符进行计数的方法。

6.5.3 反向引用

当将括号用于Perl的正则表达式中时，由每个带括号的表达式进行匹配的目标字符串的这个部分将被记住。 Perl将把这个匹配的文本记录在一些特殊的变量中，这些变量的名字是 $1（用于第一组括号）、$2（用于第二组括号）、$3、和$4等等。现在请看下面这个例子：



上面这个模式用于匹配格式很好的美国 /加拿大电话号码，比如 800-555-1212，同时将电话号码的每个部分记录在$1、$2和$3中。这些变量可以用在下面的表达式的后面：

```
if (/(\d{3})-(\d{3})-(\d{4})/) {
    print "The area code is $1";
}
```

它们也可以用作替换操作中的替换文本的组成部分，如下所示：

```
s/(\d{3})-(\d{3})-(\d{4})/Area code $1 Phone $2-$3/;
```

不过应该注意，模式匹配运行成功时，变量 $1、$2和$3的值将被清除（不管它是否使用括号），如果并且只有当模式匹配运行成功时，这些变量才被设置。基于这个情况，请看下面这个例子：

```
m/(\d{3})-(\d{3})-(\d{4})/;
print "The area code is $1";  # Bad idea.  Assumes the match succeeded:
```

在上面这个例子中，使用$1时根本没有确定模式匹配是否可行。如果模式匹配运行失败，将会带来麻烦。

6.5.4 一个新函数：grep

Perl中的一个常见操作是搜索数组，寻找某些模式。例如，如果将一个文件读入一个数组，然后你想要知道哪一行包含某个单词。 Perl有一个特殊的函数，可以用来进行这项操作，这个函数称为grep。grep函数的句法如下：

```
grep expression, list
grep block list
```

grep函数迭代运行通过 list中的每个元素，然后执行 expression或block。在expression或block中，$_被设置为要计算的列表中的每个元素。如果该表达式返回真， grep就返回该元素。请看下面这个例子：

```
@dogs=qw(greyhound bloodhound terrier mutt chihuahua);
@hounds=grep /hound/, @dogs;
```

在上面这个例子中， @dogs的每个元素被依次赋予 $_。然后根据$_对表达式/hound/进行测试。返回真的每个元素被grep返回，并存放在@hounds中。

这里你必须记住两点。首先，在表达式中，$_是对列表中的实际值的引用。如果修改$_，就会改变列表中的原始元素：

```
@hounds=grep s/hound/hounds/, @dogs;
```

当运行这个代码后，@hounds将包含greyhounds和bloodhounds（请注意它们结尾处的字母s）。通过修改$_，原始数组@dogs也被修改了，同时，它现在包含了greyhounds、bloodhounds、terrier、mutt和chihuahua。

需要记住的另一点（Perl程序员有时忘记了这一点）是：grep不一定必须与模式匹配或替换运算符一道使用，它可以与任何运算符一道使用。下面这个例子用于检索长度超过8个字符的犬名：

```
@longdogs=grep length($_)>8, @dogs;
```

grep函数与UNIX的一个命令同名，该命令用于搜索文件中的模式。UNIX的grep命令在UNIX中的用处是如此之大（因此在Perl中用处也很大），以至于它已经变成了一个动词，即“to grep”（进行模式搜索）。如果我们说to grep through a book，那么这句话的意思是翻阅每一页，寻找某个模式。

一个相关函数map的句法与grep基本相同，不过它的表达式（或语句块）返回的值是从map返回的，而不是$_的值。可以使用map函数，根据第一个数组来产生第二个数组。下面是该函数的一个例子：

```
@words= map {split ' ',$_} @ input;
```

在这个例子中，数组@input的每个元素（作为$_传递给语句块）均用空格隔开。这意味着@input的每个元素均产生一个单词列表。该列表存放在@words中。@input的每个相邻行均被分隔开来，并在@words中进行累加。

6.6 课时小结

在本学时中，我们介绍了什么是正则表达式，它们采用什么样的结构，以及如何在Perl中使用正则表达式。正则表达式是由标准字符和元字符构成的。标准字符通常是指字符本身的原义，而元字符则用于改变标准字符的含义。正则表达式可以用于测试是否存在某些模式，或者用于替换模式。

6.7 课外作业

6.7.1 专家答疑

问题：模式/ \w（\w）+W/似乎不能与文本行上的所有单词相匹配，它只能与中间的几个单词进行匹配。为什么？

解答：你查找的是用非单词字符括起来的单词字符。文本行的第一个单词的前面没有非单词字符。它的前面根本就没有任何字符。

问题：m//与//之间有什么差别。我不明白。

解答：它们之间几乎根本没有差别。当你决定设置一个不是 / 的模式界限符时，两者之间就会显示出惟一的一个差别。如果你在该模式的前面放置一个 m，比如m!pattern!，那么你才能不使用 / 界限符。

问题：我想要检查用户键入的一个数字，但是/\d*/似乎不起作用。它总是返回真！

解答：它返回真，因为仅仅使用通配符 *的模式总是能够运行成功的。它既能够对出现 0 次的\d进行匹配，也能够对出现2次、100次或1000次的\d进行匹配。使用/\d+/，能够确保你至少对一个数字进行匹配。

如果你已经开始学习正则表达式的模式，请设法做一下下面这些思考题，以了解学到了哪些知识。

6.7.2 思考题

1) 如果你拥有一些格式化为“ x=y ”的代码行，那么使用什么表达式可以将表达式的左边与右边相交换？

a. s/（.+）=（.+）/$2=$1/；

b. s/（*）=（*）/$2=$1/；

c. s/（.*）=（.*）/$2$1/；

2) 下面这个代码运行后， $2中的值是什么？

```
$foo=“Star Wars: The Phantom Menace
$foo=~/Star\s((Wars):The Phantom Menace)/
```

a. 模式匹配后$2没有被设置，因为匹配失败。

b. Wars

c. Wars:The Phantom Menace

3) 模式m/^[-+]?[0-9]+（\.[0-9]*）?$/匹配的结果是什么？

a. 日期，格式为04-03-1969

b. 格式很好的数字，如45，15.3，-0.61

c. 类似加法的模式，如4+12或89+2

6.7.3 解答

1) 答案是a。如果选择c，那么在替代字符串中将不包括符号 =，它在$1或$2中将不能被捕获，因为=出现在括号的外面。选择 b是无效的，一个字符必须出现在*的前面。选择 a，就能够正确地执行该操作。

2) 答案是a。匹配失败是因为star没有使用大写，同时，匹配代码不包含区分大小写字母的修饰符i。由于这个原因，你始终都应该在使用 $1、$2等之前测试匹配代码运行是否成功。（如果模式匹配使用了i修饰符，或者star已经变成大写字母，那么选择b就可以得到正确的结果。）

3) 答案是b。在匹配行的开头，该模式应该是一个选项 +或-，接着是一个或多个数字，然后（可选项）在行的结尾是一个小数点并且可能是多个数字。该模式可以匹配简单的、格式很好的数字。

6.7.4 实习

• 看一看你是否能够创建一个用于匹配标准时间格式的模式。下面的所有格式应该都是可

行的：12:00am、5:00pm、8:30AM。下面这些格式是不行的：3:00、2:60am、99:00am、3:0pm。

• 编写一个短程序，使它能够执行下列操作：

1) 打开一个文件。

2) 将所有文件行读入一个数组。

3) 从每个行中取出所有单词。

4) 找出至少拥有4个连续辅音或非元音字母的所有单词（比如“ thoughts ”或“ yardstick ”这样的单词）。