Perl 正则表达式讲解

摘自《Perl 编程详解》

目录：

9．3．1 原则 1

正则表达式有三种形式：匹配、替换和转换。

在表 9-1 中列有三种正则表达式运算符。

表9-1. 正则表达式运算符.

运 算 符	含 义
m//	“匹配”
s///	“替换”
tr///	“转换”

接下来对每一个表达式给出详尽解释。

匹配：m/<regexp>/这种形式表明在//内部的正则表达将用于匹配 = ~或 !~左边的标量。为了语法上的简化用/<regexp>/，略去 m。

替换：s/<regexp>/<substituteText>/这种形式表明正则表达式<regexp>将被文本<substituteText>替换，为了语法的简化用/<regexp>/<substituteText>略去 s。

·转换：tr/<charClass>/<substituteClass>/这种形式包含一系列的字符—/<charClass>—同时把它们替换为<substituteClass>。

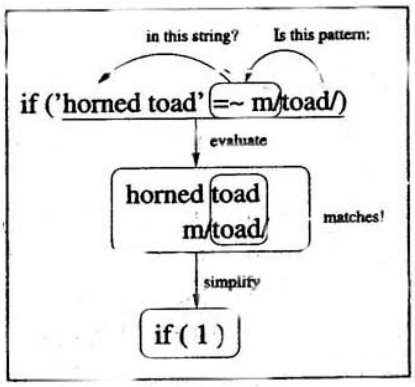

图9-1 简单模式匹配

注意转换<tr>并不真正是一个正则表达式，但是对于用正则表达式难于处理的数据常使用它来进行操纵。因此，tr/[0-9]/9876543210.组成 1223456789，987654321 等字符串。

通过使用＝~(用英语讲:does，与“进行匹配”同)和!~(英语:doesn't，与“不匹配”同)把这些表达式捆绑到标量上。作为这种类型的例子，下面我们给出六个示例正则表达式及相应的定义：

```
$scalarName =~ s/a/b;     # substitute the character a for b, and return true if this can happern
$scalarName =~ m/a;            # does the scalar $scalarName have an a in it?
$scalarName =~ tr/A-Z/a-z/;  # translate all capital letter with lower case ones, and return ture if this happens
$scalarName !~ s/a/b/;    # substitute the character a for b, and return false if this indeed happens.
$scalarName !~ m/a/;           # does the scalar $scalarName match the character a? Return false if it does.
$scalarName !~ tr/0-9/a-j/;   # translate the digits for the letters a thru j, and return false if this happens.
```

如果我们输入像 horned toad =~ m/toad/ 这样的代码，则出现图 9-1 所示情况：

另外，如果读者正在对特定变量 $_ 进行匹配(读者可能在 while 循环，map 或 grep 中使用)，则可以不用!~和=~。因而，以下所有代码都会合法：

```
my @elemente = (' a1' ,' a2' ,' a3' ,' a4' ,' a5' );
foreach (@elements) {s/a/b/;}
```

程序使 @elements 等于 b1，b2. b3，b4，b5。另外：

```
while(<$FD>) {print if (m/ERBOR/);}
```

打印所有包含 error 字符串的行：

```
if (grep(/pattern/, @lines)) {print " the variable \@lines has pattern in it!\n";}
```

打印所有包含模式 pattern 内容的行，这直接引入下一原则。

9．3．2 原则 2

正则表达式仅在标量上匹配。

注意这里标量的重要性，如果读者试一试如下代码：

```
@arrayName = (' variable1', 'variable2');
@arrayName =~ m/variable/; # looks for ' variable' in the array? No! use grep instead
```

那么@arrayName 匹配不成功! @arrayName 被 Perl 解释为 2，于是这意味着读者在输入：

```
' 2' =~ m/variable/;
```

至少讲这不能给出预想的结果。如果读者想这样做，输人为：

```
grep(m/variable/, @arrayName);
```

该函数通过@arrayName 中的每一个元素进行循环，返回(在标量环境中)匹配的次数，同时在数组环境中返回匹配元素的实际列表。

9．3．3 原则 3

对于给定的模式串，正则表达式只匹配最早出现的匹配项。匹配时缺省一次只匹配或替换一次。

这个原则使用称为“回溯”的过程指出如何匹配一个给定的字符串；如果发现了一个局部匹配进而找到使该匹配无效的东西，正则表达式在字符串中“回溯”最小的可能数量，这个数量的字符要保证不丢失任何匹配。

对于理解正则表达式正在做什么，这个原则是最有帮助的一个，同时不需要与 Perl 一样的形式来理解它正在做什么。假定有如下模式：

```
' Silly people do silly things if in silly moods'
```

同时想匹配如下模式：‘

```
' silly moods'
```

那么正则表达式引擎匹配 silly，接着遇到 people 的 P，至此，正则表达式引擎知道第一个 silly 不匹配，于是正则表达式引擎移到 P 且继续寻求匹配。它接着遇到第二个 silly，于是来匹配 moods。然而得到的是字母 t(在 thing 中)，于是移到 things 中的 t 处，继续进行匹配。当引擎遇到第三个 silly 并且尽力匹配 moods 时，匹配成功，匹配最后完成。所发生的情况如图 9-2 所示。

当我们遇到通配符时回溯将变得更加重要。如果在同一正则表达式中有几个通配符，且所有的通配符交织在一起，那么这里就有病态情形出现，在这种情形下，回溯变得非常昂贵。看如下表达式：　：

```
$line = m/expression.*matching.*could.*be.*very.*expensive.*/
```

.* 代表一个通配符，它意味着“匹配任意字符(换行符除外)零次或多次”。这个过程有可能花很长时间；如果在未匹配过的字符串末尾有可能匹配，那么引擎将发狂地回溯。为得到这方面的更多信息，请留意关于通配符方面的原则。

如果读者发现类似于上面的情形，那么通配符需将正则表达式分解成小功部分。换句话

讲，简化自己的正则表达式。

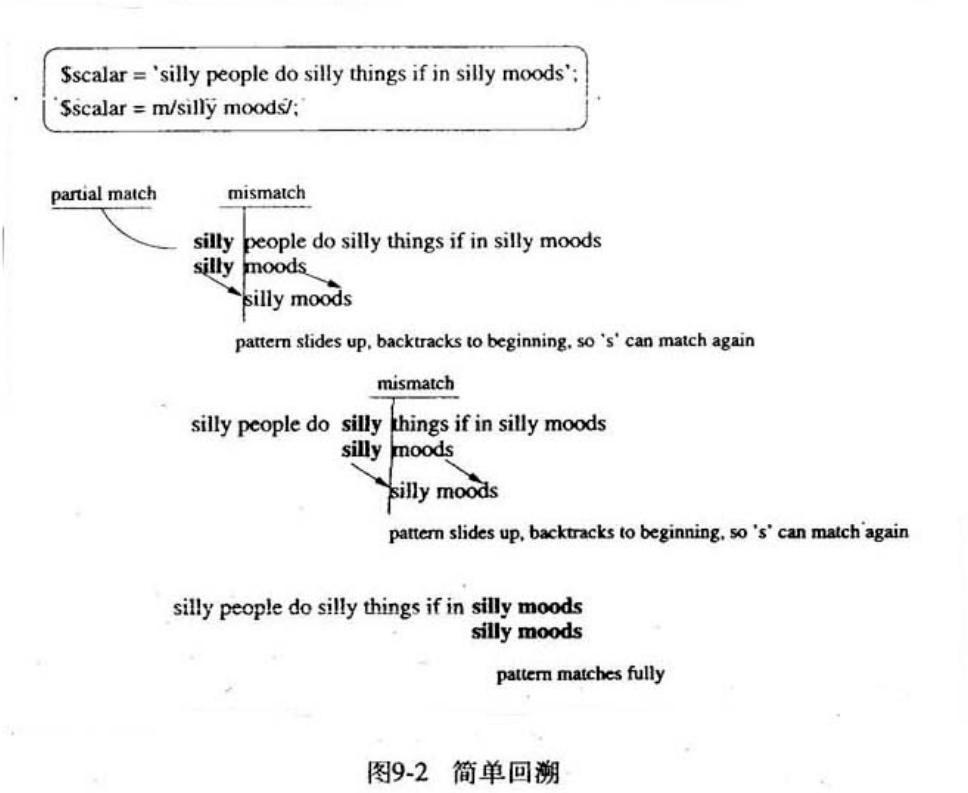

图9-2 简单回溯

9．3．4 原则4

正则表达式能够处理双引号字符串所能处理的任意和全部的字符。

在s///运算符(s/*//)，或者m//运算符m/*/的第一个分隔区，位于其中的条目确实能像双引号字符串一样对待(带有一些额外的附加功能，名义上的特殊正则表达式字符!后面描述)。读者可用他们进行内插：

```
$variable = ' TEST' ;
$a =~ m/${variable}aha/;
```

和：

```
$a = " ${variable}aha" ;
```

二者都指向同一字符串：前者在$a 中匹配字符串 TESTaha. 后者把$a 设置为字符串TESTaha。因为正则表达式处理双引号字符串能处理的每个字符，所以可以执行下列操作：

```
$expression = ' hello';
@arrayName = (' elem1', ' elem2');
$variable =~ m/$expression/; # this equals m/hello/;
```

在这里，我们简单地把$expression扩展为hello而得到m/hello/。这个技巧也可用于数组：

```
$variable =~ m/@arrayName/; # this equals m/elem1 elem2/;
```

在这里，表达式等价于 m/elem1 elem2/。如果特殊变量$"被设置为 |. 则表达式将等价于 m / elem | elem2 / ，正如我们看到的，它匹配字符串的elem或者elem2。这种方法也可应用于特殊字符：

```
$variable =~ m/\x01\27/;    # match binary character x01, and
                # octal character 27.
$variable =~ s/\t\t\t//;    # substitute three tabs for three spaces.
```

实际上，这里所讨论的除极少数例外以外，Perl 处理在 m//中的过程的确像处理在双引号中的一样。但是有例外：有某些对正则表达式引擎有明确意义的字符。那么，如果想匹配类似于正斜杠(/)或者园括引(())这样的字符会发生什么呢？这些字符对正则表达式引取有特殊意义：因而不能使用如下语句：

```
$variable=~ m//usr/local/bin/;   # matches /usr/local/bin? NO! SYNTAX ERROR
```

因为 Perl 将把/解释为正则表达式的结束标记。这里有三种方法去匹配类似于上述特殊字符的方法。第一种方法是利用反料杠来“转义”想匹配的任意特殊字符一包括反斜杠。因而刚才给出的例子可变为：

```
$path =~ m/\/usr\/local\/bin/;
```

该程序尽力匹配 $path 中的/usr/local/bin。第二种方法是使用一个不同的正则表达式字符。如果有许多字符要匹配，那么使用反斜杠则会变得很难看(路径字符尤其不好)。

幸运的是，Perl 以一种合成形式来确决这个问题。因为在当读者输入 m//或 s///时需要给每个/加反斜杠，所以正则表达式允许读者将正则表达式的定界符(/)改为自己喜欢的任意字符。例如，我们可以使用双引号(")来避免大量的反斜杠：

```
$variable =~ m"/usr/local/bin";  # Note the quotation marks.
$variable =~ m"\"help\"";        # If you are going to match quotation
                                 # marks, you need to backslash them here. (as per\")
$variable =~ S" $variable" $variable";   # works in s///too.
```

出于好的初衷，我们在本书的前几章使用了这一约定。如果使用"作为读者的正则表达式字符，那么在用起来时它充当了好的记忆法，记住在这里所处理的实际上是字符串的变相反插入；否则，引号就远不如斜杠常用。

Perl 允许使用{}()[]来书写正则表达式：

```
$variable =~ m{this works well with vi or emacs because the parens bounce};
$variable =~ m(this also works well);
$variable =~ s(substitute pattern) {for this pattern}sg;
```

这一原则对我们处理多行正则表达式非常方便。因为在这里可以不用圆括号，所以读者可以开始把表达式作为“微型函数”对待(如果读者有像 emacs 或 vi 这样的合理的智能编辑器)，换句话讲，读者可在表达式的开始和结尾部分之间往返。

第三种方法是利用函数 quotemeta()来自动地加反斜杠。如果输入如下代码：

```
$variable =~ m" $scalar";
```

则$scalar 将为被插且被转变为标量的值。这里有一个警告：任何一个特殊字符都将被正则表达式引擎影响，并且可能引起语法错误。因此，如果标量为：

```
$scalar = " ({";
```

那么输入如下代码：

```
$variabie =~ m" $scalar" ;
```

就等价于是说：$variable =~ m"({"，而这是一个运行时语法错误。如果表达式为如下形式：

```
$scalar = quotemeta(' ({');
```

则表达式会使$scalar 变为\(\{，且把$scalar 替换为：

```
$variable =~ m" \(\{" ;
```

这样才可以匹配到读者愿意匹配的字符串({。

9．3．5 原则 5

正则表达式在求值的过程中产生两种情况：结果状态和反向引用。

每次对正则表达式求值时会得到：

. 指示正则表达式匹配字符串的次数(结果状态)。

. 如果希望保存部分匹配，则有一系列称为反向引用的变量。

接下来让我们依次学习他们：

1. 结果状态

结果状态表示正则表达式匹配字符的次数。得到结果状念的方法是在标量环境下求正则表达式的值。以下所有例子使用了这一结果变量。

```
$pattern = ' simple always simple';
$result = ($pattern =~ m"simple");
```

这里,result 为 1,因为模式 simple 位于 simple always simPle 中。同样的,给定 simple always simple：

```
$result = ($pattern =~ m" complex");
```

将使 result 为空，因为 complex 不是 simple always simple 的子字符串，接着：

```
$result = ($pattern =~ s" simple" complex");
```

使 result 为 1，因为把 simple 替换为 complex 成功了。更进一步：

```
$pattern = ' simple simple';
$result = ($pattern =~ s" simple" complex" g);
```

情况变得更复杂。在这里，$result 为 2，因为在 simple always simple 中 simple 出现两次，同时正则表达式的 g 修饰符被使用，这意味着“匹配尽可能多的次数”。(要更详细材料参看本章后面的修饰符)。同样地：

```
$pattern = ' simple still';
if ($pattern =~ m" simple")
{
    print " MATCHED!\n";
}
```

在 if 子句中使用$pattern =~ m" simple",而该子句基本上告诉 Perl,如果模式$pattern 包含有子串 simple 则打印 Matched! 。

2. 反向引用

反向引用有点复杂。假定想保存一些匹配供后用，那么为达到该目的，Perl 有一个运算符(圆括号())，该运算符可用于包围读者希望匹配的一系列给定的字符。

在正则表达式中用圆括号括住某模式就是告诉解释器“嗨，我希望保存那个数据。”Perl 解释器再应请求，且将查找到的匹配保存在一系列特珠的变量中($1,$2,$3…$65536)，这些变量可用来查询第一个、第二个、第三个等等圆括号匹配，这些变量于是可以通过查看相应的变量或在数组环境下对正则表达式进行求值而且进行访问。例如：

```
$text = " this matches ' THIS' not 'THAT' ";
$text =~ m" ('TH..' )";
print " $1\n";
```

在这里，字柳 HIS 被打印出来 —— Perl 已经将它们保存在$1 中，以后再打印$1。 然而，该例子揭示了更多内容，例如：

1)通配符(字符点(.)匹配任意字符)。如果 THIS 不在字符串中，模式(TH..)将欣然匹配 THAT。

2)正则表达式匹配一行上出现的第一处模式。THIS 因为首先出现，所以被匹配。同时，按缺省 regexp 行为，THIS 将总是被匹配的第一个字符串。(可以用修饰符改变缺省值，详细

情况稍后介绍)。

图 9-3 表示了这一匹配过程如何进行。

在图 9-3 中每个圆括号与自己的数字变量一道运行。

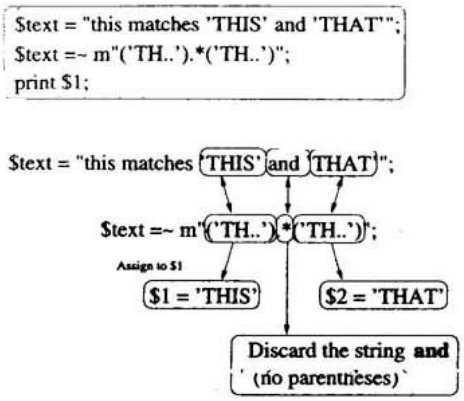

图9-3 简单反向引用

这里有更多的例子：

```
$text = ' This is an example of backreferences';
($example, $backreferences) = ($text =~ m" (example).*(backreferences)" );
```

这里又用了通配符来分开两个文本字符串$example 和$backreferences。这些字符串存放在$1 和$2 中，且随后立即赋给$example 和$backrefercences。该过程在图 9—4 中说明。

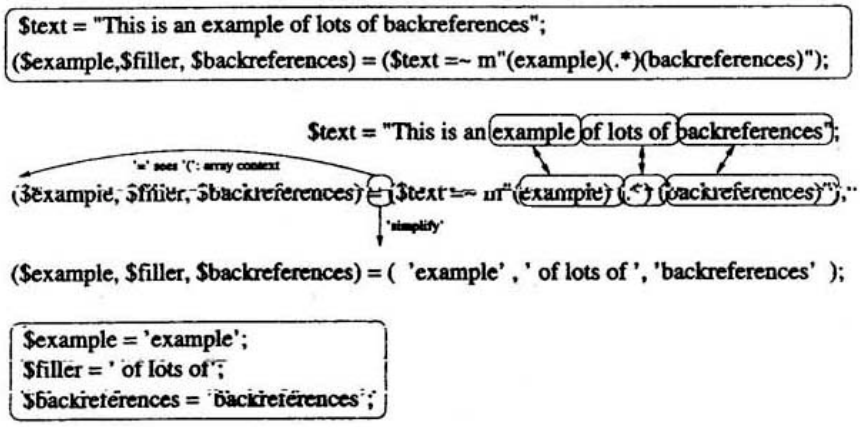

图9-4 引用的直接赋值

然而应注意的是仅当文本字符串匹配时,给$example 和$bacbreference 赋值的过程才发生。当文本字符串不匹配时，$example 和 backreferences 是空的。这里有更好的同样的例子，该例子包含在 if 语句中，仅在匹配时打印$1 和$2。

```
if ($text =~ m" (example).*(back)")
{
    print $1;   # prints ' example' -- since the first parens match the text example.
    print $2;   # prints ' back' -- since the second parens match the text back
}
```

这样，如果正则表达式根本不匹配将发生什么？如果使用下面的模式：

```
$text = ' This is an example of backreferences';
$text =~ s" (examplar).*(back)" doesn't work";
print $1;
```

$1 因正则表达式匹配不成功而不能被赋值。更重要地，Perl 不会告诉读者它没有给$1 赋任 何值。最后一例展示了关于正则表达式的两点重要内容：

1)正则表达式是“要么全部要么什么也没有”的处理，只因为 back 字符串在模式内才能匹 配，所以：

```
This is an example of backreferences'
```

并不意味着整个表达式达到匹配。因为 exemplar 不在字符串中，因而替换失败。

2)如果正则表达式失败，反向引用不能得到赋值。因此，不能肯定将打印出什么内容。当跟踪逻辑问题时，这就是让人吃惊的原因；且经常是 Perl gotcha。$1 只是一个正则变量，并且(与 Perl 语法相反)如果正则表达式失败则反向引用不被设置为“空白”。有人认为这是一个缺陷，然而另有人认为这是一个特色。不过，当分析下面的代码时第二点变得非常明显。

```
1 $a = ' bedbugs bite';
2 $a =~ m" (bedbug)";      # sets $1 to be bedbug.
3
4 $b = ' this is nasty';
5 $b =~ m" (nasti)";       # does NOT set $1 (nasti is not in ' this is nasty' ).
6                          # BUT $1 is still set to bedbug!
7 print $1;                # prints ' bedbug'.
```

在这种情况下，$1 为字符串 bedbug，因为第 5 行的匹配失败！如果希望得到 nasti，好吧，那是自己的问题。这种 Perl 化的行为可能让人不知所措。考虑的是自己要当心。

3．使用反向引用的一般构造法

如果想避免这种很平常的缺陷(读者想得到一个匹配,但是没有得到并且用前面的匹配为替代而结束)，在把反向引用赋给变量时只要应用下列三个构造法之一：

1)短路方法。核查匹配，如果匹配发生，此时且只有此时用' &&'进行赋值，例如：

```
($scalarName =~ m" (nasti)" ) {$matched = $1;}
```

2)if 子句。将匹配放于 if 子句中，如果 if 子句为真，此时且只有此时才为模式赋值。

```
if ($scalarName =~ m" (nasti)" ) {$matched = $1;}
else {print "$scalarName didn't match"; }
```

3)直接赋值。因为可以直接把正则表达式赋给一个数值，所以可始终利用这一点。

```
($match1, $match2) = ($scalarName =~ m" (regexp1).*(regexp2)" );
```

读者的所有模式的匹配代码看起来应该与前述三个例子中的一个相似。缺少这些形式，那么就是在没有安全保证的条件下进行编码。如果读者从不想有这种类型的错误的话，那么这些形式将节省读者的大量时间。

4．在正则表达式中使用反向引用

当希望使用 s" " "运算符或者用 m" "运算符对一些复杂模式进行匹配很困难时，Perl 提供了读者应意识到的有用功能。这个功能就是反向引用可以用于正则表达式自身。换句话说，如果用圆括号括住一组字符，那么就可以在正则表达式结束之前使用反向引用。如果想在 s" " "的第二部分(带下划线)中使用反向引用，那么要使用语法$1，$2 等。如果想在 m" " 或者 s" " "的第一部分(带下划线)使用反向引用，那么使用语法\1\2 等。下面是一些例子：

```
$string = ' far out';
$string =~ s "(far)(out)" $2 $1";       # This makes string ' out far'.
```

我们在该例中只是将单词 far out 转换为 out far。

```
$string = ' sample examples';
if ($string =~ m" (amp..) ex\1") {print " MATCHES!\n"; }
```

这个例子有点复杂。第一个模式(amp..)匹配字符串 ample。这意味转整个模式成为字符串 ample example，其中带下划线的文本对应于\1。因此，模式匹配的是 sample examples。

下面是同样风格更复杂的例子；

```
$string = ' bballball';
$string =~ s" (b)\1(a...)\1\2" $1$2";
```

让我们详细地看看这个例子。该例完成匹配，但是原因不是太明显。对这个字符串的匹配有五个步骤：

1)在圆括号中的第一个 b 匹配字符串的开头，接着将其存放在\1 和$1 中。

2)\1 于是匹配字符串中的第二个 b，因为与 b 相等，而第二个字符碰巧是 b。

3)(a..)匹配字符串 all 且被存在\2 和$2 中。

4)\1 匹配下一个 b。

5)因为\2 等于 all 所以匹配下一个且是最后三个字符(all)。

将他们放到一起就得到正则表达式匹配 bballball，或者说是整个字符串。既然$1 等于'b'，$2 等于 all，则整个表达式：

```
$string = ' bballball' ;
$string =~ s" (b)\1(a..)\1\2" $1$2";
```

(在这个例子中)转换为如下代码：

```
$string =~ s" (b)b(all)ball" ball";
```

或者用行话讲，用 bballball 替换 ball。 '

正则表达式看起来很像图 9-5 所示。

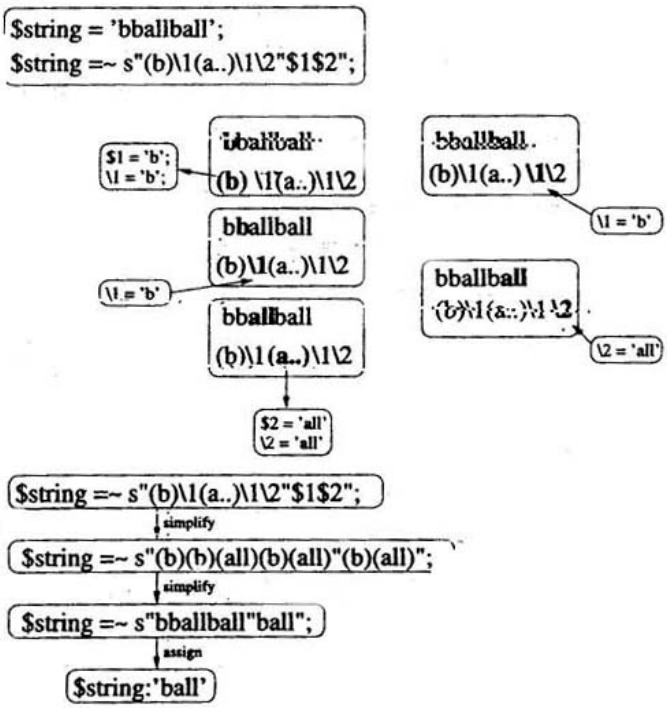

图9-5 正则表达式例子

s″″″中有一些复杂的反向引用。如果理解了最后一个例子。那么读者在理解 Perl 的正则表达式如何工作方面远远走在了前面。反向引用可能而且确实会变得更糟。

5．嵌套反向引用

嵌套反向引用对于复杂的难以用单一顺序(一个字符串跟在另一个字符串后面)进行匹配的字符串作用明显。例如下面的表达式：

```
m″ ((aaa)*)″ ;
```

使用 *来匹配多次出现的 aaa：即匹配″,aaa,aaaaaa,aaaaaaaaa。换句话说，Perl 匹配一行中有多个 3a 的模式。但是这个模式不会匹配 aa。假定想匹配如下的字符串：

```
$string = ' softly slowly surely subtly';
```

那么使用嵌套园括号后下面的正则表达式会匹配：

```
$string = m″ ((s...ly\s*)*)″ ;  # note nested parens.
```

在该例中，最外层的圆括号捕获全部字符串：softly slowly surely subtly 。最内层的圆括号捕获字符串的组合，这种组合是以 s 开头，以 ly 结尾且 ly 后跟空格形成的。因此，正则表达式先捕获 surely，将其抛开，然后捕获 slowly，将其抛开，然后捕获 surely，最后捕获 subtly。这里有一个问题，反向引用按什么顺序出现？读者可能在这个问题上很容易迷惑。是外层圆括号先出现呢，还是内层的内括圆号先出现？最简单的解决办法是记住以下三条原则：

1)在表达式中，一个反向引用越在前，它对应的反向引用编号就越小。例如：

```
$var =~ m″ (a)(b)″;
```

该例中，反向引用(a)变为$1，(b)变成了$2。

2)一个反向引用如果它包含范围越广，则它的反向引用编号就越小。例如：

```
$var =~ m″ (c(a(b)*)*)″;
```

该例中，包含全部内容(m ″(c(a(b)*)*)″)的反向引用成为$1。有 a 嵌套在里面的表达式 m″(c(a(b)*)*)″成为$2。在(m″(c(a(b)*)*)″)中带有 b 的嵌套表达式成为$3。

3)在两个规则冲突的情况下，规则 1 优先。在语句$var =~ m″ (a)(b(c))″ 中，(a)成为$1，b(c)成为$2，(c)成为$3。

因而，在这个例子中，(s...ly\s*)*成为$1，(s...ly\s*)*成为$2。

注意这里有另一个问题。让我们一起返回到刚开始的复杂的正则表达式：

```
$string = 'softly slowly surely subtly'
$string = m″(((s...ly\s*)*)″;    # note nested parens.
```

这里(s...ly\s*)*匹配什么呢？它匹配多个字符串；首先是 softly，接着是 slowly，再接着是 surely，最后是 subtly。既然(s...ly\s*)*匹配多个字符串，那么 Perl 会抛弃掉第一个匹配而使$2 成为 subtly。

即便有这些规则，嵌套圆括号仍然可能引起混乱。要做的最好事情就是实践。再一次用这些逻辑的不同组合去实现正则表达式，然后把它们交给 Perl 解释器。这样做可让读者明白反向引用是以什么次序被 Perl 解释器进行解释的。

9．3．6 原则 6

正则表达式的能力的核心在于通配符和多重匹配运算符。通配符运算符允许匹配字符串中的多个字符。如果正在处理二进制数据，那么通配符就匹配一系列字符。多重匹配运算符可匹配零个、一个或多个字符。就讲解 Perl 的基础而言，到目前为止我们所使用的例子都

是带启发性的，但功能并不是很强大。实际上，读者可能会用 C 子程序去完成它们中的任意一个。Perl 正则表达式集合的强大功能来自于其匹配文本的多模式能力，(即：通过前面提到的逻辑“速记法”来描述许多不向的数据模式)。Perl 正好可提供最好的速记法。

1．通配符

通配符代表字符类。他没有如下字符串，但不知道他们是否大写：

```
. Kumquat
. Kristina
. Kentucky
. Key
. Keeping
```

这种情况下，如下的 Perl 表达式会匹配每个单词的第一个字符：

```
[Kk]
```

这是字符类的一个例子。Perl 中的所有通配符可以用括号[并把想匹配的字符类放在括号中最后加上结尾括号]这种方法表示。前面的通配符告诉正则表达式引擎"好，我正在这里查找"K"或者"R"。如果发现两者之一那么就匹配它”。下面是另一些使用通配符的例于：

```
$scalarName = ' this has a digit (1) in it';
$scalarName =~ m" [0-9]";     # This matches any character between 0 and 9, that is matches any digit.
$scalarName =~ 'this has a capital letter (A) in it';
$scalarName =~ m" [A-Z]";     # This matches any capital letter (A-Z).
$scalarName =~ " this does not match, since the letter after the string 'AN' is an A"
$scalarName =~ m" an[^A]";
```

前两个例子相当直观，[0-9]匹配 this has a digit(1) in it 中的数字 1。[A—Z]匹配 this has acapital letter(A) in it 中的大写字符 A。最后一例稍有点技巧，因为在这个模式中仅有一个 an，所以可能被匹配的字符唯有最后的四个字符，即 an A。

然而，通过询问模式 an[^A]我们已经明确地告诉正则表达式去匹配 a，接着是 n，空格，最后一个为非 A 的字符。因而，该例中没有完成匹配。如果给定模式为 match an A not an e，那么匹配就会完成，因为第一 an 被跳过后，第二个正好匹配！就像如下例子：

```
$scalarName = " This has a tab(   ) or a newline in it so it matches";
$scalarName =~ m" [\t\n]"    # Matches either a tab or a newline.
                             # matches since the tab is present.
```

这个例子说明了用匹配和通配符可以做的一些有趣的事情。首先，读者已经在""字符串插入的相同字符也可以在正则表达式和用括号表示的字符类中([t\n])插入。其中"\t"匹配制表符，"\n"匹配换行符。

其次．如果读者在[]里的开头部分放置一个^，则通配符会匹配非字符组中的字符。同样地，如果在[ ]中放置-，则通配符匹配给定的范围(在这个例子中是所有的数字[0-9]。所有的大写字母([A-Z])。这些运算符还可被合并，从而得到相当特别的通配符：

```
$a =~ m" [a-fh-z]";         # matches any lowercaes letter * except* g.
$a =~ m" [^0-9a-zA-Z]";     # matches any nonword character. (i.e.,NOT
                            # a character in 0-9, a-z or A-Z)
$a =~ m" [0-9^A-Za-z]";     # a mistake, Does not
                            # equal the above. Instead matches 0-9,
$a =~ m" [\t\n]";           # matches a space character: tab, newline or blank).
```

需许意的重要地方是第三个例子,在[0—9^A—Za—z]中的插入记号是一个字面上的插入记号，而不是代表否定，原因是它在字符类的中间出现。因此，如果读者想得到一个否定的字符类。那么就总是要把插入记号放在[]开头。也不要忘记用[]。如果读者忘记了[]，那么得到的将是一个字面上的文本字符串，而不是一个字符类。

(1)公用通配符

碰巧某些通配符是公用的；当读者每次想匹配一个数字时，可能不愿意每次都必须要输入类似于[0-9]这样的代码。对于那些情况，Perl 有几个方便的快捷通配符，使用它们后可使编程工作容易。下面是这些边配符以及它们代表的含义和它们对应的字符组合：

. \d — 匹配数字(字符组合[0-9])。

. \D — 匹配非数(字符组合[^0-9])。

. \w — 匹配单词字符(字符组合[a-zA-Z0-9_])(这里下划线算作一个单词字符)。

. \W — 匹配非单词字符(字符组合[^a-zA-Z0-9_])。

. \s — 匹配空格字符(字符组合[\t\n])(制表符、换行符、空格)。

. \S — 匹配非空格字符(字符组合[\t\n])。

. - 匹配任意字符(在某些情况下)换行符除外(字符组合[^\n])、当输入 m″(.*)″S 时匹配任意字符。参见本章后面的修饰符。

. $- 尽管它实际上并不是一个通配符(它不匹配任何具体字符). 但它是广泛使用的特殊字符；如果将其放在正则表达式的尾部则它匹配“行尾”。零宽度断言。

. ^- 尽管实际上不是一个通配符，但如果它位于正则表达式的开头则它是匹配“行首”的特殊字符。零宽度断言。

. \b,\B- 与$和^相同；不匹配字符，但匹配单词边界(\b)或匹配无单周边(\B)。零宽度断言。

我们从表中注意到的第一点是“点”通配符(.)。它常与多重匹配运算符一道用于在条目间充当填充者。请看以下匹配：

```
$a = ' NOW is the time for all good men to come to the aid of their party';
$a =~ m″ (Now).*(party)″;       # matches, since ' .' matches any
                                character except newline
                                and ' *' means match zero or more characters.
```

*捕获位于 Now 和 party 中间的所有字符，匹配是成功的。(在这种环境下的“所有”意味着“零或者更多，尽可能多”。这就是所谓的贪梦(greediness)；在我们后面谈到多重匹配运算时再谈它。)

下面是通配符的一些其他例子。注意我们在=~的左边使用了单引用字符串(这是一个测试表达式的简单方法)：

```
1 ' 1956.23' =~ m″ (\d+)\.(\d+)″;                    # $1 = 1956, $2 = 23
2 ' 333e+12' =~ m″ (\D+)″;                           # $1 = ' e+'
3 ' $hash($value)' =~ m″ \$(\w+){\$(\w+)}″;          # $1 = ' hash', $2 = 'value'
4 ' $hash($value)' =~ m″ \$(\w+){(\w)*(\w+)(\w*)}″;  # $1 = ' $', $2 = ' hash',
                                                     # $3 = ' $', $4 = ' value'
5 ' VARIABLE = VALUE' =~ m″ (\w+)(\s*) = (\s*)(\w+)″; # $1 = ' VARIABLE',  #$2 = ' ',
                                                     # $3 = ' ', $4 = ' VALUE'
6 ' catch as catch can' =~ m″ ^(.*)can$;             # $1 = ' catch as catch'
7 ' can as catch catch' =~ m″^can(.*)$               # $1 = ' as catch catch'
8 ' word_with_underlines word2' =~ m″ \b(\w+)\b;     # $1 = word_with_underlines
```

每个例子中，我们使用了一个不同的通配符，在该程序中，用*表示在“一行中匹配零

个或者多个通配符”。用+表示“在一行中匹配一个或者多个通配符”。这些例子中有些本身就是有用的：例 5 展示了使用\s*来加强表达式对付散乱空格的方法；例 8 示例了匹配一个单词的一般化方法；例 4 示范了使用关键字匹配哈希结构的一般化方法。

然而，特殊地．例 1 不是匹配 Perl 数字的通用方法。但是如果给出 Perl 支持的所有格式，那么这会是一个十分困难的问题。我们将在后面把它作为一个问题考虑。在该表中还有一个地方需要注意：一些适配符被标记为“零宽度断言”，我们在下面讲述仑们。

(2)零宽度断言和正宽度断言

在表 9-2 中的字符就是读者可能称作的正宽度断言：

表 9-2 正声明

\D	非数字
\d	数字
\w	单词
\W	非单词
\s	空格
\S	非空格
’.’	换行符以外的任意字符。

这些断言在字符串中实际上匹配一个字符。正宽度意味着匹配一个字符，同时正则表达式引擎在匹配过程中“吃掉”它们。在表 9-3 中列出的负宽度断言。

这些断言不是匹配一个字符，它们匹配的是一种条件。换句话讲，^cat 匹配以 cat 开头的字符串，但并不匹配任一给定字符。请看下面的表达式：

```
$ziggurautString = ’ this matches the word zigguraut’;
$ziggurautString =~ m″ \bzigguraut\b″;
$ziggurautString =~ m″ \Wzigguraut\W″;
```

表9-3 负声明

^	字符串开头
$	字符串结尾
\b	单词边界
\B	非单词边界

第一个例子匹配成功，因为它是在两个非单词字符(单词边界)之间查找 ziggurat。而字符串满足这种条件。

第二个例子没有完成匹配，为什么呢？因为在末尾的\W 是正宽度断言，因此．必须匹配一个字符。但是在行未不是字符，而是一种条件。这是重要的区别。

更进一步讲，在第二个例子中即使实现了匹配，那么正则表达式引擎会消去涉及到的字符。因此，如果输入如下代码：

```
$ziggurautString = ″ This matches the word zigguraut now″;
$ziggurautString =~ s″ \Wzigguraut\W″ ″ g;
```

最后得到的结果是 This matches the wordnow。原因是已经把单词和插入的空格替换掉了。因而：

•零宽度断言，例如\b\B 能匹配没有字符的地方。它们在匹配的过程中不会去掉任一字符。下面是关于通配符匹配的其他例子：

```
$example = ’ 111119’;
$example =~ m″ \d\d\d″;    # match the first three digits it can find in the string Matches
’ 111’.
$example = ’ This is a set of words and not of numbers’;
```

```
$example =~ m" of (\w\w\w\w\w)";  # Matches ' of words'..Creates a backreference
```

请注意最后一个例子，该列中，因为在字符串的开头有一个 of(在 words 前面)，所以模式匹配器会匹配这个特定的 of。而不会去匹配后面的 of(在 numbers 前面的那个)。最后一例也展示了我们将讨论论的问题。这就是如果想匹配五个单词字符的话，那么必须打印五次\w，这样就很烦恼。因此、为了便于匹配长模式，Perl 提供了多重匹配运算符。我们接下来讨论这个问题。

2．多重匹配运算符

PerI 中有六个多重匹配运算符。主要用于避免编写重复代码，例如上一节提到的在一行中声明\w 五次。读者可把它们看作是速记的捷径。

PerI 的六个多重匹配运算符是：

- *——匹配零次，一次或多次。
- ?——匹配一次或者多次。
- ?——匹配零次或者一次。
- {X}——匹配‘X’次。
- {X，}——匹配‘X’或者更多次。
- {X，Y}——匹配‘X’到‘Y’次。

这里有两个等价的例子，但是哪个易读呢？

```
$example = ' This is a set of words and not of numbers';
$example =~ m" of (\w\w\w\w\w)";   # Matches ' of word'.
$example =~ m" of (\w{5})";         # Usage of {X} form. Matches 5 characters,
                                    # and backreference $1 becomes the string ' words'.
```

读者可能发现第二个例子的代码易读。这个例子使用了多重匹配运算符来避免写重复、讨厌的代码。第二个例子也用符号来匹配不定数量的字符。正则表达式 a*匹配”，a，aa 或者 aaa，或任意数量的 a。也就是匹配零个或多个 a，例如：

```
$example = ' this matches a set of words and not of numbers';
$example =~ m" of(\w+)";
```

匹配的是字符串 words(of(\w+) eq of words)，又如：

```
$example =~ m" of (\w(2,3))";    # Usage of {X,Y}. Matches the string ' wor'
                                  # (the first three letters of the first match it finds.
matches the string ' wor' (' of \w{2,3}' equals ' of wor' here)
```

与直觉相反．下面代码中的 m" "子句：

```
$example = ' this matches a set of words and not of numbers';
$example =~ m" of(\d*)";
```

匹配该字符串，尽管我们正在用\d*来查找数字。什么原因呢？因为\d*表现为零个到多个，所以表达式匹配零个数字！然而：

```
$example =~ m" of(\d+)";
```

将不会匹配相同的字符串，因为表达式使用的是\d+而不是\d*。这样就意味着查找字符串中位于单词 of 后面的一个或多个数字，而这种条件是这个字符串不具有的。

3．贪婪

到目前为止，上面所有的例子陈述主要的一点是正则表达式引擎如何按给定的表达式匹配给定的字符串。即缺省情况下，多重匹配运算符是贪婪地。

“贪婪”在这里是什么含义呢?“贪婪”意味着在缺省状态下，Perl 的多重匹配运算符能捕获一个字符串中的最大数量的字符，同的仍然有能力来完成模式匹配。读者应掌握好这

一点。理解了贪婪的 Perl 表达式的本质会节省大量时间，以免跟踪古怪的正则表达式行为。

这里有几个简单的关于贪婪行为的例子，例子中的贪婪行为能使程序员发疯。让我们从下面的语句开始：

```
$example = ' This is the best example of the greedy pattern match in Perl5';
```

假定在该例中想匹配 is。相应地，要编写如下代码：

```
$example =~ m            # This (.*) the#;
print $1;                # This does NOT print out the string ' is'!
```

读者希望打印$1 时看到的是 is。但是所得到的是下面的字符串：

```
' is the best example of'
```

程序的工作过程如图 9-6 所示。

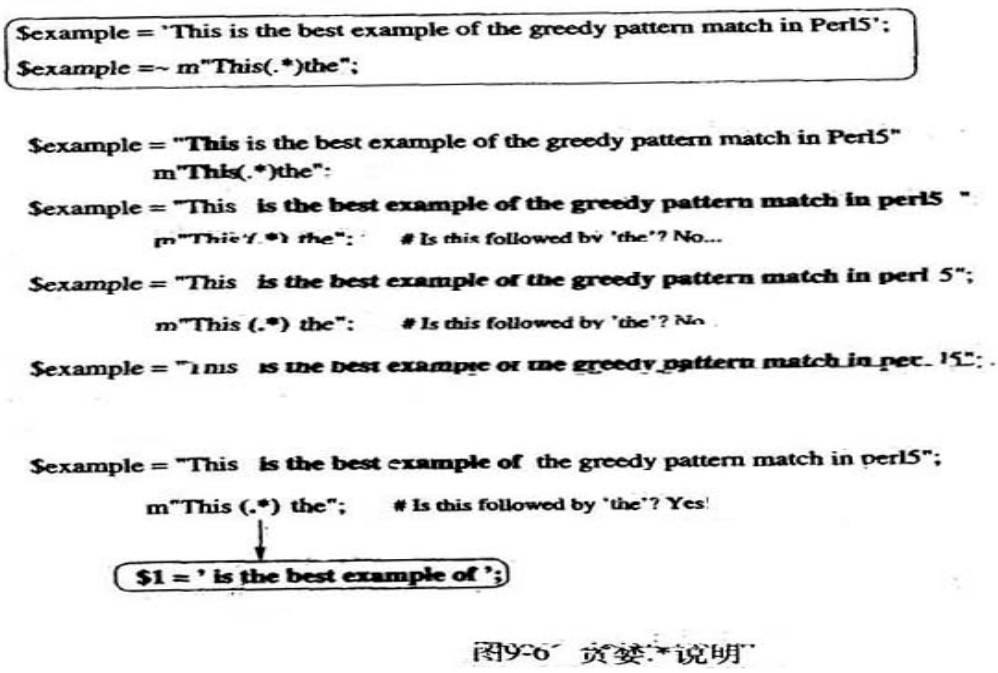

图9-6 贪婪*说明

造成这种结果的原因是多重匹配运算符*的贪婪。运算符*取得所有的字符直到最后出现的字符串 the(在 greedy 前的一个)。那么如果读者不小心，则在使用正则表达式后得到不可预料结果。

这里有更多例子：

```
$example = ' sam I am';
$example =~ m" (.*)am';                              # matches the string ' sam I'
$example = ' RECORD: 1 VALUE: A VALUE2: B';
$example =~ m".RECORD';(.*)VALUE";             # matches ' 1 VALUE: A';
$example = ' RECORD';
$example =~ m" \w{2,3}";                             # matches REC
```

最后一例显示出甚至数字多重匹配运算符也是贪婪的。虽然在 RECORD 中有两个单词字符，但是 PerI 更愿匹配三个，原因是它有这个能力。如果输入' RE' =~ m" \w{2,3}"，则只能匹配两个字符，因为这是可能匹配的最大数量。

4. 回溯和多重通配符

好吧，已经准备很久了. 现在到处理棘手题目的时候了。正如前面说过的，通配符和回溯的结合使正则表达式有极其缓慢的性能。如果读者理解了其中的原因，那么这是读者正“得到” 正则表达式的好标志。

看以下例子：

```
$string =~ m ″ has(.*)multiple(.*)wildcards″;
```

这意味着正则表达式将查找(以数字顺序)：

1)模式 has(m″ has.*multiple.*wildcards″)。

2)正则表达式能发现的最大文本直到它到达最后的 multiple(m″ has(.*)mMltiple(.*) wildcards)。

3)字符串 multiple(m″ has(.*)multiple(.*)wildcards″)。

4)能发现的最大文本直到遇到最后的 wildcards(m″has(.*)multiple(.*)wildcards″)。

5)字符串 wildcards(m″ has(.*)multiple(.*)wildcards″)。

接着考虑一下，使用以下的模式将会发生什么：

```
has many multiple wildcards multiple WILDCARDS
```

所发生的一切是：

1)、Perl 匹配 has(i.e, m″ has(.*)multiple(.*)wildcards);

```
  has many multiple wildcards multiple WILDCARDS
```

2)Perl 执行 m″has(.*)multiple(.*)wildcards 部分且吃掉它能够发现的所有字符，直到遇到最后的 multiple，接着匹配：

```
has many multipIe wildcards multiple WILDCARDS
```

3)Perl 匹配字符串 multiple(即 m″ has(.*)multiple(.*)wildcards):

```
has many multiple wildcards multiple WILDCARDS
```

4)Perl 试图发现字符串 wildcards 但是失败了，接着读出字符串的其余部分：

```
WILDCARDS does not match ' wildcards' !
```

5)现在 Perl 应做什么呢？因为有多个通配符(*)，所以 Perl 回溯。正则表达式可能犯错的地方是在第 2 步，即当它吃掉：

```
has many multiple wildcards multiple WILDCARDS
```

时，因而，它正好返回到 has 后面：

```
has many multiple wildcards multiple WILDCARDS
   ^goes back here
```

6)现在试图更正错误，仅捕获那些位于最后一个 multiple 之前的字符。因而，模式 m″ has(.*)multiple(.*)wildcards″ 匹配：

```
has many multiple wildcards multiple WILDCARDS
```

7)在 m″ has(.*)multiple(.*)wildcards″ 中的 multiple 随后匹配：

```
has many multiple wildcards multiple WILDCARDS
```

8)接着通配符匹配空格——m″ has(.*)multiple(.*)wildcards″)匹配：

```
has many multiple wildcards multiple WILDCARDS
```

9)最后，wildcards(m″ has(.*)multiple(.*)wildcards″)匹配：

```
has many multiple wildcards multiple WILDCARDS
```

因而整个正则表达式匹配 has many multiple wildcards。它给出了预料结果，但是得到该结果肯定走了弯路。肯定地说，Perl 实施正则表达式的算法有些捷径来提高性能，但是刚才给出的逻辑基本上是正确的。我毫不犹豫地认为这个例子是本章中最主要的一个——即使 Perl 高于有时也不能从语义上正确分析正则表达式(很使他们恼火)。一遍又一遍地复习．直到充分理解结果。这之后，尽量按 Perl 的匹配方法跟踪以下代码：

```
$pattern = " afbgchdjafbgche";
$pattern =~ m" a(.*)b(.*)c(.*)d";
```

我们在此随意结出一个：

```
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";)
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- greedy, goes to last ' b'
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";)
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- matches g
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";)
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- backtrack because no ' d'
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- now we take up everything to the next to last b
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";)
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- now the second .* becomes greedy.
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";)
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- still no d.darn.backtracks
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- wildcard becomes less greedy, gobbles to next to
last c
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";)
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- there is only one d in the expression and this
matches up to it
afbgchdjafbgche (m" a(.*)b(.*)c(.*)d";) -- a match!
                                          matches ' afbgchd'.
```

不是很标致，正如读者所看到的，如果读者有多个贪婪的多重匹配运算符，事情可能变得更糟，效率更低。也许从这个例子中得到的最简单的就是左边的多重匹配运算优先于右边的执行。下面这样的模式：

```
m" (.*)(.*)";
```

在一个没有换行符的字符串中，将总是使第一个反向引用包含整个字符串，而第二个却什么也没有。像这种错误最好使用-Dr 命令行选项进行处理，就像在 Perl－Dr.script.p 中一样。我们将在第 21 章讨论这一点。另外，如果读者不想有贪婪会发生什么？好吧，正如我们会在下节见到的，perl(与其他包不同)有能力处理字符的非贪婪形式。

5．非贪婪多重匹配运算符

贪婪可能是一件幸事，但是它也常会引起争论！举一个常用的 C 注释语句的例子(它是很平常的 FAQ 且是在文档中)。假定要匹配下面的粗体文本：

```
/* this is a comment*/ /* another comment */
```

这里试图找出一个贪婪方式的解决方法。我们想匹配后跟所有文本的/* 直到包括 */。如果我们尝试以下代码：

```
m" /\*.*\*/";
```

于是，将匹配：

```
/* this is a comment*/ /* another comment */
```

又一次因为是 greedy 形式所以匹配全部字符串。

以下是笔者所能提供的最好的贪婪方式的解决办法：

```
$commentmatcher =~ m" /\*([^*]*|\**[^/*])*\*/";
```

它不是所有解决方法中最好读的。(我们可以使用 m" "使之更易读，后面将讲到。)我们

将在后面复习这点，因为理解了这个特殊表达式对在总体上掌握正则表达式有很大帮助！现在我们一起看看贪婪形式。对于非贪婪形式这里有一简单规则来记住它们：只要在一个贪婪多重运算符的后面添加?就可使之成为非贪婪。

因而，前面的 commentmatcher 成为：

```
$commentmatcher =~ m" /\*(.*?)\*/";
```

这仍然不是最易读的形式，但是肯定比前面的要好！我们可以简单地描述这条语句如下：“取到/*，接着取最小数量的字符，再取结束的*/”，工作过程如图 9-7 所示。

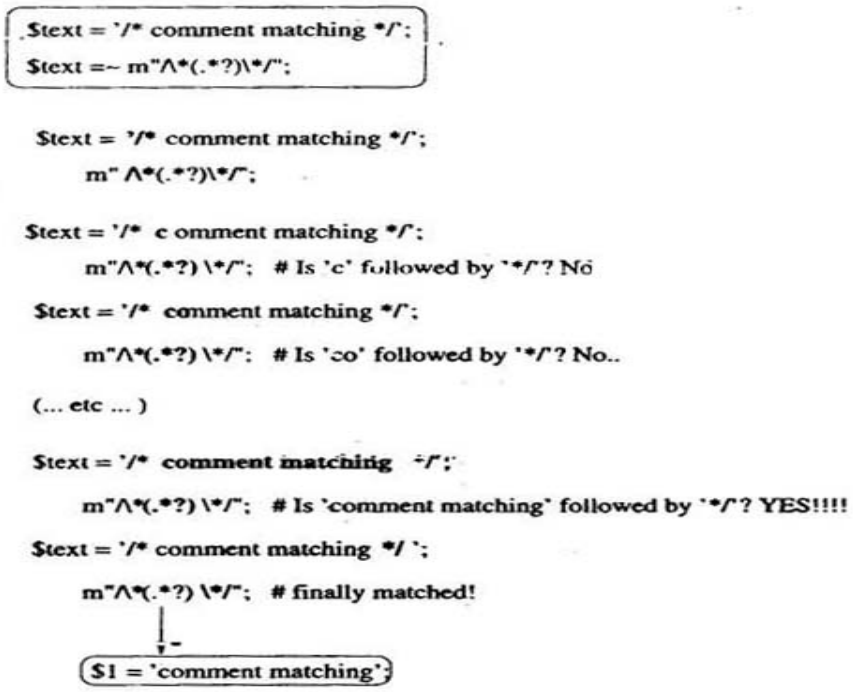

图9-7 最小匹配

“懒惰”是用于描述?的另一术语。正则表达式引擎可以北认为是缓慢地前移，直到遇到第一个可匹配的表达式。在该例中，它正是遇到一个*/以后移到下一步骤。如果读者输入以下代码：

```
$line =~ m" (.*?)(.*?)(.*?)";
```

每个(.*?)符什么也不会匹配。什么原因呢?在这里匹配的最小数量的字符是零(因为读者已经输入*(零到多或全))。因而根据匹配零个字符的要求. 每个(.*?)都完成自己的工作，且缓慢地把控制传到下一个(.*?)，返过来它又取零个字符。下面是懒惰匹配程序的通用规则。只要向他们添加字符类(如.、\d、或[123])，就可得到懒惰行为：

- *?——匹配零、一或多次，但匹配可能的最少次数。
- +?——匹配一或多次，但匹配可能的最少次数。
- ??——匹配零次或一次，仍匹配可能的最少次数。
- {X}?——正好匹配“X”次
- {X, }?——匹配“X”或者更多次，但匹配可能的最少次数。
- {X,Y}?——匹配“X”到“Y”次，但匹配可能的最少次数。

这里有关于最小匹配程序的一些例子，以及那些与非贪婪模式匹配的例子：

```
$example = ‘This is the time for allgood man to come to the aid of their party’ ;
$example = ~ m“ This (.*?) the”;
```

该例处配 ‘is’，如果它是贪婪地则会匹配 ‘ is the time for all good man to come to’。

```
$example = ‘19992113333333333333331’;
if ($example = ~ m “ l (\d{3,}?)” )
{
}
```

该表达式表示匹配后跟三个数（或更多数字）的 l。然而，由于 ‘?’之后什么也没有，它基本上表示匹配后跟三个数字的 1。因此，将匹配到‘1999’。

```
$example = ‘1f999133333333333333331’;
if ($example = ~ m “ l (\d{3,}?)” )
{
}
```

此处，我们有同样的表达式，但是模式匹配器发现的第一个 1 不满足条件. 因为它后面跟‘f ’。于是匹配器找到下一个 1，并且匹配 1333。

```
$example = ‘ 1f999133333333333333331’;
if ($example = ~ m “ l (\d{3,}?)1” )
{
}
```

该例中，匹配与前面不同的内容。我们已经要求无论我们用\d(3,)到什么数据字都必须跟有一个 1。因而，虽然模式匹配器是懒惰的，但是它也必须走到表达式的最后以找到一个匹配。正如读者所看到的，对于正则表达式的逻辑，大家必须非常仔细。对于未入门的人，吃惊的东西太多。对于那些不知道自己正在干什么的人这里有很多方法来使他们不知所措。

掌握了正则表达式的原则就是向前迈了一大步。如果读者需了解更多信息，转到“Perl 调试”部分，在那里我们将给出更多关于调试 Perl 正则表达式的信息。

9．3．7 原则 7

如果想匹配多个字符集则 Perl 使用称为替换的技术。

替换

替换是告诉 Perl 希望匹配两个或多个模式中的一个的方法。换言之，表达式：

```
(able | baker | charlie)
```

告诉 Perl 在正则表达式中“查找字符串 able 或者 baker 或者 charlie”。作为例子，以下面的语句开始：

```
$declaration = ‘ char string [80] ;’ or $declaration = ‘ unsigned char string [80];’
```

假定想匹配字符串 char 或 unsigned char，一次匹配多个字符串将会是很方便的。下面的正则表达式二者都匹配：

```
foreach $declaration ( ‘ char string [80]’, ‘unsigned char string [80]’ )
{
    if ($declaration = ~ m “ (unsigned char | char )” )
    {
        print “ :$1:” ; # prints ‘ :char :’ first time around.
        # prints ‘’ :unsigned char :’ second time around .
    }
}
```

语法 ｜意味着匹配 unsigned char 或者 char 且把已 匹配的字符串保存在反向引用中。替换可以是很微妙的，因为对于替换行为有一件重要的事情要记住：

• 替换总是试图匹配阂括号中的第一个项目；如果匹配不成功，才尝试第二个样式的匹配，

依次类推。

这就是所谓的极左匹配，它能解释人们接触正则表达式时所遇到的许多错误。拿前面的例子来说，假定改变了圆括号中条目的次序，那么该例子变为：

```
$declaration = ‘ unsigned char string [80]’ ;
$declaration = ~ m “ (char | unsigned char )” ;
```

这能匹配字符串 unsigned char （就像在 unsigned char string [80] 中那样）吗？不能，它匹配 char（即， unsigned char string [80]）。因为在列表中 char 处在第一位，所以比字符串 unsigned char 要有限。正则表达式匹配 char，因而在反向引用中保存了错误的替换。这种错误信息很普通，所以这里指出一点：

• 总是拿优先级最高的字符串先去匹配，最符殊的字符串先去匹配。如果读者不这么做，则会陷入痛苦的等待中，例如：

```
$line = ~ m “ (. * | word )” ;
```

绝不会匹配 word。这是因为 word 是.*（也就是四个任一字符）的一个实例。同时正则表达式极左匹配，所以首先拾取.*，因而：

```
$line = “ wordstar” ;
$line = ~ m “ (.* | word)” ;
```

将匹配 wordstar（即，整个字符串而不是 word。.*匹配任何字符集，且因.*在替换中处于第一位，所以始终比“word”优先。因而没有匹配 wordstar 中的 word：

```
$line = ~ m “(word | .*)” ; #since ‘ word’ is first.
```

在不知道某单词是否为定界符或者该单词是否为复数的情况下，替换是很有帮助的。例如，如下代码：

```
$line = ‘ words’ ;
$line = ‘ word’ ;
$line = ~m “word (s|$)” sg; # word may be followed by the character ‘!’ or ‘$’.
```

两者都能匹配。该语法将匹配字符串 word，无论 word 后跟字符串结束标记或者 s。用下式替换$：

```
$line = ~ m “ word (s | \b)” ;
```

这就得到处理复数的好方法。

9. 3. 8 原则 8

Perl 用（?..）语法给正则表达式提供扩展名。

在 Perl 历史上的某一天（大约 Perl 4 向 Perl5 转变时），为了使正则表达式集合增长，人们决定 Perl 必须“达到元字符标准”。一些人争论道有太多的元字符存在，而另一些人不同意，直到键盘上没有太多的元字符为止。

到那是人们在认为形成一种独特的可用于多次扩展的构造是一个好主意。查看键盘后，发现一个相当普通的字符（？）在任何地方都不用。因此，它被定了下来。语法看起来像这样：

```
(? <special character(s)> <text>)
```

这里，<special character(s)>代表扩展名，<text>是表达式使用的文本。四个最常用的扩展名在表 9-5 中给出。

表9-5 正则表达式扩展名

扩 展 名	含 义
?=<regexp>	匹配下一组文本但不“吃掉”它来进一步匹配
?!<regexp>	只在后面没有<regexp>的情形下匹配
?：<regexp>	组合但非反向引用创建括号
?xims	正则表达式的内置修饰符

此外，有一个运算符(?#comment)允许用户把注嵌入正则表达式。不过因为 x 修饰符的出现(后面讨论)这个运算符已过时了。在其它方面，扩展名像其它正则表达式结构一样工作。把它们放到正则表达式中。如果输入：

```
$line = ~ m “ I love(?! oranges)” ;
```

则由于(?!禁止 oranges 跟着字符串 I love，所以此行代码匹配 love figs 而不是 I love oranges。然而，这个表达式匹配的是 I love orange 或者 I love ripe oranges，原因是它仅禁止以字符串 oranges 开头的字符串。可以输入：

```
$line = ~ m “ I love (?!. * orange )”;
```

来禁止这些字符串。

事实上，修饰符(?!是 Perl 语有中最容易理解的结构。人们希望它处理如下事情：

```
$line = ~ m “(?! Oranges ) I love “;
```

使之不匹配 oranges I love。这明显地不会成功。结构(?1)仅当下一个子串不是 oranges 时才匹配。 因而，在这个例子中，使匹配无效的唯一位置是在字符串的开头 在其他任何位置都不会做任何工作。正则表达式向前执行，并已发现紧接着的六个字符不是“oranges”。因而要求满足，执行下一个要求。

另两个用得最多的表达式是(?:…)和(?=…) 。例如：(?:…)通过除去不必要的反向引用而使正则表达式更高效。如果输入：

```
$line = ~ m “ (?:int | unsigned int | char )\s*(\w+)”;
```

以得到变量名，而可能不希望保存变量的类型，因而用(?:)。表达式匹配给定类型，后跟空格，后跟变量名(\w＋)。但是把变量名保存在＄1 中而不是$2 中，(?：)被忽略。这样可节省时间和内存，尤其是在大型模式匹配中。

另一个表达成(?:)仅在使用 g 修饰符时有用，下面我们将仔细介绍。g 修饰符可让读者在正则表达式中从读者停止的位置后面开始，不必从头再来。例如，如果有看起来像下面这样的数据：

```
BLOCK1 <data> BLOCK2 <data2> BLOCK3 <data3>
```

而读者想首先匹配<data>，其次匹配<data2>，第三(最后)匹配<data3>，于是可输入：

```
$line = ~ m “ BLOCK \d(,*?) (BLOCK\d | $)” g;
```

第一次运行会匹配<data>，但是在第二个 BLOCK 之后，就把“匹配指针”放到了错误位置。如果输入为：

```
$line = ~ m “BLOCK \d (.*?) (?=BLOCK\d)” g;
```

则由于说明“匹配 BLOCK1 和 BLOCK2 之间的最少数量的文本”所以匹配相同数量的文本。为了下一次匹配的目的而忽略了 BLOCK2，以致于使下一次调用正则表达式能匹配<data2>。图 9-8 概括了二者之间的区别。用(?＝)形式可以现在输入如下语句：

用(?=)形式可以现在输入如下语句：

```
BLOCK 1 <data> BLOCK2 <data2> BLOCK3 <data3>
while ($line = ~ m “BLOCK\d(.*?) (?=BLOCK\d | $ )” g)
{
          print “ $1 \n”;
}
```

并使其先打印出<data>，接着打印出<data2>，继而打印出<data3>。我们在下面介绍 g 修饰符时将了解更多的道理。

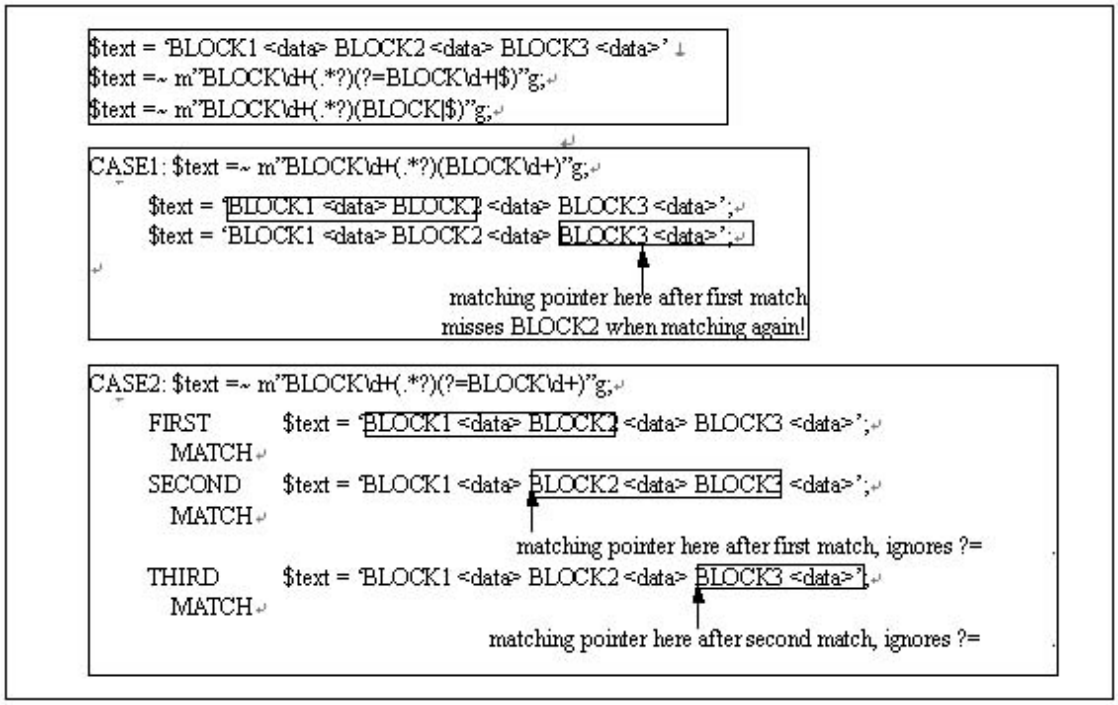

图 9-8 m″ BLOCK(.*?)(?=BLOCK)″ g 和 m″ BLOCK(.*?)BLOCK″ g 之间的区别

9．3．9 正则表达式原则概括

前一节知识应该足够让读者使用正则表达式。虽然称它们为“基本”的正则表达式，但是 我们将看到通过不同的组合，正则表达式在对付数据的战斗中可形成庞大的联盟——这常常是一场战争。八条原则——再声明一次——是：

- 原则 1：正则表达式有三种不同形式(匹配(m//)，替换(s///)和转换(tr///))。
- 原则 2：正则表达式仅对标量匹配($scalar=~ m″ a″; 可以工作; @array=~ m″ a″把@array 作为标量对待，但因此可能不会成功)。
- 原则 3：正则表达式匹配一给定模式的最早的可能匹配。缺省时，仅匹配或替换正则表达式一次($a = ’ string string2’; $a=~s″ string″ ″; 导致$a==’ 1 string 2’)。
- 原则 4：正则表达式能够处理双引号所能处理的任一和全部字符($a =~ m″ $varb″ 在匹配前把 varb 扩展为变量；因而，$varb=’ a’ $a=’ as’ $a=~S″ $varb″ ″ 使 $a 等于 s)。
- 原则 5：正则表达式在求值过程中产生两种情况：结果状态和反向引用。是否$a=~ m″varb″ 告诉如果$a 中有子串 varb 出现，那么$a=~s″(word2)(word1)″ $1 $2″ “调换”这两个单词。
- 原则 6：正则表达式的能力核心在于通配符和多重匹配运算符以及它们如何操作。$a =~ m″ \w+″ 匹配一个或多个单词字符；$a=~m″ \d*″ 匹配零个或多个数字。
- 原则 7：如果欲匹配不只一个字符集合，Perl 使用叫做替换(alternation)的技术。如果输入 m″ (cat|dog)″ 则相当于说明“匹配字符串 cat 或者″ dog″。
- 原则 8：Perl 用(?..)语法给正则表达式提供扩展名。

哇！读者怎样学所有这些原则？我建议先从简单的开始。如果学会了$a =~ m″ ERROR″ 是

在$a 中查找子串 ERROR，那么就已经比在，C 这样的低级语言中得到了的能力。在讨论完两个主要的概念即在正则表达式修饰符和环境之后我们将在下面给出许多实用的例子.

9．3．10 正则表达式修访符

上一节中所有正则表达式都具有如下形式：

```
$a=~m//; # m″  ″is synony2n，as is m { }
```

或者：

```
$a=~S///;  # s″ ″ ″ is synonym，as is s { } { }
```

二者都表示正则表达式的缺省形式，它们从表达式的头部开始，匹配或者替换一次。

假定我们不想“匹配或替换一次”。假定我们想把表达式中的所有 a 替换为 b，或者我们想：不分大小写地匹配。换句话讲，假定我们不想使用缺省行为。幸运地，有一些有帮助的修饰符，我们可以把它们增加到正则表达式中重载它们的行为来做缺省动作之外的事情。正则表达式修改过的形式看起来像下面这样：

```
$a =~ m//gismxo; $a =~ s///geimxo;
```

用一个或者多个“附加”在后面的修饰符来转换 Perl 表达式的行为。我们先讲述在二者之间有共同特征的修饰符(s,m,i,x,o)进而讲述在二个运算符之间有不同含义的修饰符(e,g)。下面就来讲述它们。

1．替换和匹配中的标识符

结构 m″ ″和 s″ ″ ″ 有许多共同的运算符。它们列在下面的表 9-6 中。

下面详细说明这五个修饰符。

(1)x：扩展的可读性规范表达式

正则表达式有时变得很乱，大家前面已看到了。但是那些还不及实际生活中有些表达式的一半。考虑下面的代码---粗略地——匹配 Perl 中的一个子程序：

```
$line=~m″ sub\s+(w+)\s+{(.*?)}\s*(?=sub)″ s;
```

表 9-6

修 饰 符	含 义
x	可读的正则表达式形式
i	大小写不敏感的正则表达式
s	表达式被看作单个字符串
m	将表达式看作多个字符串
o	编译正则表达式一次

这些代码意味着什么?即使读者是一位 Perl 老手，即使给出了注释，这个表达式仍会迫使读者想上一小阵子。尤其缺少空格是件讨厌的事情，特殊字符的数目让读者感到头晕。在 Perl4 中没有提供 x 运算符。因为 x 运算符使在正则表达式中放置空格以使它们易读及允订留出空间来放注释成为可能，所以说 x 运算符是特别的幸事。表达式:

```
$line =~ m″ sub\s+(\w+)\s+{(.*?)}\s*(?=sub)″ s;
```

变为：

```
$line =~ m{
        sub\s+ ( \w+ ) \s+          # matches the ’ sub’ keyword, subroutine name
                                    # and matches the white space afterwords.
        {                          # opening brace
```

```
        ( .*?)              # matches the text of the sub. and saves it
                            # further use.
    }                           # closing brace
    \s*(?=sub)                  # the next sub keyword
    }sx;
```

可是仍然很难看.尽管仍有众多令人头痛的特殊字符,但读者可以非常清楚地看出它们之间的逻辑关系。它更象是实际的逻辑思维过程.花括号放在正确的位置。既然能够写入注释,这就非常有助于给出正在执行内容的详细说明。

然而，注意几个警告。既然在有正则表达式中允许有空格，那么滤掉后，下面的代码将不会匹配：

```
$line =" multi line string\nhere" ;
$line =~ m" multi line string" X;    # this does not match the above because
                                     # the space above GETS MUNGED OUT.
```

注意 x 可读性函数仅在替换运算符的第一个括号处起作用(也就是在 s{}{}中)。这是因为只有第一个括号把它的值内插为一个双引字符串。第二个括号中的一切都是字面上的。例如，下面的例子可能不去做读者想让它做的事情：

```
$line = 'aaaaaa' ;    # we want 'bbbbbb' after the substitute below.
$line = ~ s{
              a
          }
          {
              b
          }     gx;   # we want to do a 'general' match, i.e., match
                      # ALL a's for b's. DOESN'T WORK!
print $line;          # prints '              .  b                b...etc'
          # six times over.
```

这里发生的一切是由于在第二个括号中的项目没有留出空格。相反，a 的每个实例(在这里留出了空格)用 3 个制表符，一个 b 和一个换行符替换，导致混乱。

易读的正则表达式在处理更复杂事情时对保持头脑清醒有巨大作用。

(2)i：大小写不敏感的匹配 *

正则表达式缺省情况下是区分大小写的。使用 i 则表明将进行不区分大小写的匹配。

```
  $pattern = ' Exercise' ;
  $pattern =~ s"exer" EXER" i'    # matches first four characters of Exercise.(Exer)
  $pattern = ' Edward Peschko' ;
  $pattern == ???
```

在两种情况下都发生，他们匹配：第一种将 Exercise 变邮 XERcise，第二种把 Edward Peschko 变成 Edmund Peschko。

i 修饰符在书写几个单调的正则表达式时确实是好的速记法，例如*pattern＝~ m" [Ee][Xx][Ee][Rr]";

(3)s：把模式作为一行处理

没有修饰符，点(.)匹配除换行符之外的任何字符。有时这是有帮助的；有时却是非常令人沮丧的，特别是如果读者有长达多行的数据时。考虑下面的例子：

```
$line =
BLOCK1:
 <text here1>
```

```
END BLOCK
BLOCK2:
 <text here2>
END BLOCK'
```

现在假定读者想匹配在块＜text here[0-9＞之间的文本：

```
$line =~m{
            BLOCK(\d+)
                 (.*?)
            END\ BLOCK   # Note backslash, Space will be ignored otherwise
 }
```

没有成功，因为通配符(.)匹配除换行符之外的每一字符.所以正则表达式遇到第一个换行符时就陷入困境，进而就在那儿停止进行匹配。有时，像在下面这个例子中,用通配符(.)匹配除换行符之外的任何字符是很有帮助的。通过扩展，使通配符(\s)匹配[\n\t]而不是制表符和空格是很有帮助的。这就是 s 运算符的功能。它告诉 Perl 不要假定正在使用的字符串是一行。前面的例子在用一个 s 加到正则表达式末尾之后匹配成功.

```
$line =~ m{
            BLOCK(\d+)
                (.*?)
            END\ BLOCK   # Note backslash, Space will be ignored otherwise
 }s;
```

在末尾加 s，现在匹配完成。

(4)m：把模式看作多行

m 运算符与 s 运算符的功能相反。换句话讲,它是指:"把正则表达式看作多行而不是一行。" 这样基本上使^和$现在不仅匹配字符串的开头和结尾(分别地)，而且使^匹配换行符之后的任意字符且使$匹配一个换行符。在下面的例子中： ’ ’

```
$line = ' a
b
c';
$line = ~m"      ????
```

m 修饰符使反向引用$成为 a 而不是 a\nb\nc。

(5)o：仅编译正则表达式一次

o 修饰诗在处理长表达式时是有帮助的。输入如下语句：

```
 $line =~ m" <very long expression>" ;
```

在这里<very long expression>是一个段落，甚至有几页长。正如它所表示的，每次 Perl 遇到正则表达式时就编译它。这就要花时间，而且如果需要匹配的模式很复杂的话. 那么所用的正则表达式将特别长。在 Jeffreu Friedl 的书中，有一个匹配 e-mail 地址的表达式。e-mail 地址长达 6598 字节!如果不用 o 修饰符就全完了,但是如果仅编辑它一次则是可使用的。然而，有一个应注意的警告。如果读者输入：

```
$line = ~m" $regex" o;
```

则读者向 Perl 承诺$regex 将不会改变。如果改变了，Perl 将不会注意读者的改变。因而，

```
$regex = 'b';
while ('bbbbb' =~ m" $regex o) { $regex = 'c';}
```

实际上是 Perl 中的一个无限循环。$regex 改变了。但是在正则表达式中没有反映(然而,在每个程序中这没有限制用户用一个而且仅仅一个 regexp。带有 o 的每个表达式实例在使用前编译).

2．替换专用修饰符

修饰符(s、m、x、i 和 o)即适用于替换又适用于匹配(s///,m//)，但是有几个修饰符是匹配专用。它们就是下面列出的 g 和 e。

(1)g：替换所有模式为它们的等价部分。

缺省时，运算符 s///仅在第一次发现字符时进行替换。如果想替换每个单一实例的内容为其他内容，就要使用 g 运算符。下面的三个例子是等价的：

```
$pattern='NUM1 NUM2 NUM3;
$pattern=~s" NUM" LETTER" g;   #substitutes NUM for LETTER.
$pattern=~s" num" LETTER" gi;  #Note -- you can stack these modifiers.
                               #does exactly the same thing as the above.
while($pattern=~s" NUM" LETTER"){}
```

所有的例子导致$pattern LETTER1 LETTER2 LETTER3。第 1 个例子依据条件这样做，第二个依据条件(gi 标识符)，第三个慢慢地完成(每次 s"NUM"LETTER"替换一次．首先是 NUM1 LETTER2 LETTER3，其次是 NUM1 NUM2 LETTER3，最后是 NUM1 NUM2 NUM3)。

(2)e：把 s///的第二部分看作完整的“微型 Perl 程序”而不是一个字符串进行求值。用于 s///的 e 修饰符是非常棒的．但是也是很让人投入的。读者可以用它来完成法术般的替换。我们在这里只简单地提及它，并给出一例。我们假定想要把如下字符串中的所有字母替换为相应的 ASCII 数字：

```
$string='hello';
$string=~ s{ ( \w)}   #we save the $1.
          {ord($1):=-" ";}egx;
```

该例会打印出 104 101 108 108 111。在这里每个字符被依次取出然后通过 ord 函数转换为数字。不用说，这在极短时间内可以处理非常强大的材料。这也会冒极难理解的危险。

我们建议读者把使用这一逻辑作为最后的方法，在所有的“技巧”都失败后才用。它的使用有时会隐藏做事的清晰方法。是上面的清晰呢，还是下面的更清晰?

```
$string=turnToAscii($string);
sub turnToAscii
{
   my($string)=@_;
   my($return, @letters);
   @ letters = split(//,$string);
   foreach $letter (@letters)
   {
       $letter = ord($letter)." "if ($letter = ~ m" \w" );
   }
   $return = join(' ',@letter);
   $return;
}
```

后一个例子很明显且易于维护。然而，它在长度上是前者的十余倍而速度要慢几倍；当使用 e 时必须调用一个判断。

9．3．11 匹配和 g 运算符

修饰符(x、i、s 和 e)的工作与匹配运算符 m//相同;，然而有一个明显的改变是 g 运算符的工作方式方面，而且读者将频繁地使用到它。

正如我们前面所看到的，在替换中的 g 运算符意味着要替换正则表达式的每一个实例。

然而，这在匹配环境下是毫无意义的,反向引用表明一个且仅有一个匹配。因而，Perl 在 m//中以不同于 s///的方法使用 g 运算符。

Perl 给 g 运算符连结一个迭代程序。当用$string=~m" " g 匹配一次时，Perl 会记住匹配发生的地方。这意味着读者可以使用迭代程序来匹配离开的地方。当 Perl 遇到字符串结尾时,重新设置迭代程序：

```
$line = " hello stranger hello friend hello sam";
while($line=~ m" hello (\w+)" sg)
{
    print " $1\n";
}
```

这会输出：

```
stranger
friend
sam
```

接着退出，原因是内部的迭代程序到达表达式的尾部。这里有一个提示，如果读者正在使用 g 修饰符，那么对通过赋值匹配的变量的任何修改都会导致重新设置迭代程序。

```
$line = " hello";
while($line=~ m" hello" sg)
{
    $line = $line;
}
```

这是一个无限循环!所以在匹配时要约束自己和避免修改字符串。(做一份拷贝来替代它)

9．3.12 修饰符和环境

如果在开始学习本章之前读者不熟悉 Perl 正则表达式，那么读者的头脑中可能会漂浮着同的修饰符、方法、特殊字符等等。让我们花点时间来看看使用正则表达式的不同形式，然后介绍一些常用的正则表达式例子结束本章。

这一切都与环境有关。记住，在 Perl 中，环境是王者，如果读者注意它，则通过辨识不同表达式所处的环境就可以做许多强有力的事情。我们简单地“结晶”一些我们到现在为止所见到的形式，然后增加几个新的。

1．标量环境下的替换（无修饰符）

这种情况的代码看起来如下所示：

```
if($string =~ s" a" b")  {print: " Substituted Correctly" ;}
```

程序会打印 Substituted Correctly，当字符串实际上匹配后给 if 返回 1。同时它也会把 a 的所有实例替换为 b。

2．在标量环境下的替换(g 修饰符)

这种情况下返回成功匹配的次数。使用这种方法，如果读者输入以下代码：

```
($string =~ s" a" b" g) == ($string =~ s" a" b" g)
```

Perl 将告诉读者是否$string 中有与$string2 中相同数量的 a，同时进行替换。如果读者愿意可以在整个文件上使用它：

```
undef $/;
my $fh = new FileHandle(" File1");
my $fh2 = new FileHandle(" File2");
(($line = <$fh>) =~ s" a" b") == (($line2 = <$fh2>) =~ s" a" b");
```

这会计算两个文件中的 a 数，当再做替换时比较它们。

3．在数组环境下的替换(无修饰符)和数组环境下的替换(g 标识符)

这两个较烦人。它们实际上做与标量环境下替换相同的工作。

4．在标量环境下的匹配(无修饰符)

这与在标量环境下不带修饰符的替换相同。如果读者输入：

```
if($line =~ m"a") {print "Matched an a!\n" ;}
```

它只检查在$line 中是否有一个 a。如果输入：

```
if($line=~m" \b(\w+)\b"){print " $1\n";}
```

这会检查在$line 中是否有单词，接着把它存放在$1 中，打印它。同时：

```
($line=~m" \b(\w+)\b")&&(print " $1\n";);
```

做同样的工作，仅使用短路来打印它。

5.在数组环境下的匹配(无修饰符)

这会匹配正则表达式能匹配的第一个位置，接着简单地把反向引用放入一个可快速访问的表中。例如：

```
($variable,$equals,$value) = ($line = ~m" (\w+)\s*(=)\s*(\w+)");
```

该代码取到第一个引用(\w+)，使之成为$variable；取到第二个引用(＝)使之成为$equals;然后取到第三个引用(\w+)，使之成为$value。

6．在数组环境下匹配(g 修姊符)

取到正则表达式，尽可能多地应用它.然后把结果放人到由所有可能匹配组成的数组中，例如：

```
$line = '1.2 3.4 beta 5.66';
@matches = ($line =~m" (\d*\.\d+)" g);
```

将使@matches 等于 1.2，3.4，5.66。g 修饰符完成迭代，首先匹配 1.2，其次是 3.4，第三是 5.6。同样:

```
undef $/;
my-$FD= new FileHandle(" file");
@comments = (<$FD> =~m" /\*(.*?)\*/");
```

将生成一个包含文件$fd 中所有注释的数组。

7．标量环境下的匹配(g 修饰符)

最后，如果在标量环境下使用匹配运算符，那么读者得到与在正则表达式世界中甚至是在 Perl 世界中完全不同的行为。这就是我们谈到的“迭代”行为。如果输入以下代码：

```
$line = " BEGIN <data> BEGIN <data2> BEGIN <data3>";
while($line =~ m" BEGIN(.*?)(?=BEGIN|$)" sg)
{
   push(@blocks, $1);
}
```

这将匹配以下的文本(粗体)，随后在接下来的 while 迭代中把它加载@blocks 中。

```
BEGIN <data> (%)BEGIN <data2> BEGIN <data3>
BEGIN <data> BEGIN <data2>(%) BEGIN <data3>
BEGIN <data> BEGIN <data2>(%) BEGIN <data3>
```

通过(%)我们已经表明了每次迭代开始匹配的位置。注意本例中(?=)的用法。它是匹配正确方式所必须的；如果不用它，匹配程序将会被放在错误的地方。