

第五章 模式匹配

- ↓ 第五章 模式匹配
 - ↓ 5.1 正则表达式箴言
 - ↓ 5.2 模式匹配操作符
 - ↓ 5.2.1 模式修饰词
 - ↓ 5.2.2 m// 操作符（匹配）
 - ↓ 5.2.3 s/// 操作符（替换）
 - ↓ 5.2.3.1 顺便修改一下字符串
 - ↓ 5.2.3.2 当全局替换不够“全局”地时候
 - ↓ 5.2.4 tr/// 操作符（转换）
 - ↓ 5.3 元字符和元符号
 - ↓ 5.3.1 元字符表
 - ↓ 5.3.2 特定的字符
 - ↓ 5.3.3 通配元符号
 - ↓ 5.4 字符表
 - ↓ 5.4.1 客户化字符表
 - ↓ 5.4.2 典型 Perl 字符表缩写
 - ↓ 5.4.3 Unicode 属性
 - ↓ 5.4.3.1 Perl 的 Unicode 属性
 - ↓ 5.4.3.2 标准的 Unicode 属性

Perl 内置的模式匹配让你能够简便高效地搜索大量的数据。不管你是在一个巨型的商业门户网站上用于扫描每日感兴趣的新闻报道，还是在一个政府组织里用于精确地描述人口统计（或者人类基因组图），或是在一个教育组织里用于在你的 web 站点上生成一些动态信息，Perl 都是你可选的工具。这里的一部分原因是 Perl 的数据库联接能力，但是更重要的原因是 Perl 的模式匹配能力。如果你把“文本”的含义尽可能地扩展，那么可能你做的工作中 90% 是在处理文本。这个领域实在就是 Perl 的最初专业，而且一直是 Perl 的目的——实际上，它甚至是 Perl 的名字：Practical Extraction and Report Language（实用抽取和报表语言）。Perl 的模式提供了在堆积成山的数据中扫描数据和抽取有用信息的强大工具。

你可以通过创建一个正则表达式（或者叫 **regex**）来声明一个模式，然后 Perl 的正则表达式引擎（我们在本章余下的部分称之为“引擎”）把这个正则表达式拿过并判断模式是否（以及如何）和你的数据相匹配。因为你的数据可能大部分由文本字符串组成，所以你没有理由不用正则表达式搜索和替换任意字节序列，甚至有些你认为是“二进制”的数据也可以用正则处理。对于 Perl 而言，字节只不过碰巧是那些数值小于 256 的自然数而已。（更多相关内容见第十五章，Unicode。）

如果你通过别的途径已经知道正则表达式了，那么我们必须先警告你在 Perl 里的正则表达式是有些不同的。首先，理论上讲，它们并不是完全“正则”的，这意味着 Perl 里的正则可以处理比计算机科学课程里教的正则表达式更多的事情。第二，因为它们在 Perl 里用得实在是太广泛了，所以在这门语言里，它们有属于自己的特殊的变量，操作符，和引用习惯；这些东西都和语言本身紧密地结合在一起。而不象其他语言那样通过库松散地组合在一起。Perl 的程序员新手常常徒劳地寻找地这样的函数：

```
match( $string, $pattern );
subst( $string, $pattern, $replacement );
```

要知道在 Perl 里，匹配和子串都是非常基本的任务，所以它们只是单字符操作符：m/PATTERN/和 s/PATTERN/REPLACEMENT/（缩写为 m// 和 s///）。它们不仅语法简单，而且还象双引号字符串那样分析，而不只是象普通操作符那样处理；当然它们的操作还是象操作符的，所以我们才叫它们操作符。你在本章通篇都会看到这些操作符用于匹配字符串。如果字符串的一部分匹配了模式，我们就说是一次成功的模式匹配。特别是在你用 s/// 的时候，成功匹配的部分将被你在 REPLACEMENT 里声明的内容代替。

本章所有的内容都是有关如何制作和使用模式的。Perl 的正则表达式非常有效，把许多含义包含到了一个很小的表达式里。所以如果你想直接理解一个很长的模式，那很有可能被吓着。不过如果你能把长句分解成短句，并且还知道引擎如何解释这些短句，那你就理解任何正则表达式。一行正则表达式相当于好几百行 C 或者 JAVA 程序并不罕见。正则表达式可能比一个长程序的单一一行要难理解，但是如果从整体来看，正则表达式通常要比很长的程序要好理解。你只需要提前知道这些事情就可以了。

5.1 正则表达式箴言

在我们开始讲述正则表达式之前，让我们先看看一些模式的样子是什么。正则表达式里的大多数字符只是代表自身。如果你把几个字符排在一行，它们必须按顺序匹配，就象你希望的那样。因此如果你写出一个模式匹配：

```
/Frodo/    (译注：记得电影“魔戒”吗? ;) )
```

你可以确信除非该字串在什么地方包含子字串“Frodo”，否则该模式不会匹配上。（一个子字串只是字串的一部分。）这样的匹配可以发生在字串里的任何位置，只要这五个字符以上面的顺序在什么地方出现就行。

其他字符并不匹配自身，而是从某种角度来说表现得有些“怪异”。我们把这些字符称做元字符。（大多数元字符都是自己淘气，但是有一些坏得把旁边的字符也带“坏”了。）

下面的就是这些字符：

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

实际上元字符非常有用，而且在模式里有特殊的含义；我们会一边讲述，一边告诉你所有的那些含义。不过我们还是要告诉你，你仍然可以在任意时刻使用前置反斜杠的方法来匹配这十二个字符本身。比如，反斜杠本身是一个元字符，因此如果你要匹配一个文本的反斜杠，你要在反斜杠前面放一个反斜杠：\\。

要知道，反斜杠就是那种让其他字符“怪异”的字符。事实是如果你让一个怪异元字符再次怪异，它就会正常——双重否定等于肯定。因此反斜杠一个字符能够让它正确反映文本值，但是这条只对标点符号字符有用；反斜杠（平时正常）的字母数字字符作用相反：它把该文本字符变成某些特殊的东西。不论什么时候你看到下面的双字符序列：

```
\b \D \t \3 \s
```

你就应该知道这些序列是一个元符号，它们匹配某些特殊的东西。比如，\b 匹配一个字边界，而 \t 匹配一个普通水平制表字符。请注意一个水平制表符是一个字符宽，而一个字边界是零字符宽，因为它是两个字符之间的位置。所以我们管 \b 叫一个零宽度断言。当然，\t 和 \b 还是相似的，因为他们都断言某些和字串里的某个特殊位置相关的东西。当你在正则表达式里断言某些东西的时候，你的意思只是说，如果要匹配模式，那些东西必须是真的。

一个正则表达式里的大多数部分都是某种断言，包括那些普通字符，只不过它们是断言它们必须匹配自身。准确来说，它们还断言下一个东西将匹配字串里下一个字符，这也是为什么我们说水平制表符是“单字符宽”。有些断言（比如 \t）当匹配的时候就吞掉字串的一部分，而其他的（比如 \b）不会这样。但是通常我们把“断言”这个词保留给零宽度断言用。为避免混淆，我们把这些东西称做宽度为一个原子的东西。（如果你是一个物理学家，你可以把非零宽的原子当作物质，相比之下零宽断言类似无质量的光子。）

你还会看到有些元字符不是断言；而是结构元素（就好象花括弧和分号定义普通 Perl 代码的结构，但是实际上什么也不干）。这些结构元字符在某种程度上来说是最重要的元字符，因为学习阅读正则

表达式关键的第一步就是让你的眼睛学会挑出结构元字符。一旦你学会了挑出结构元字符，阅读正则表达式就是如坐春风（注：当然，有时候风力强劲，但绝对不会把你刮跑。）

有一个结构元字符是竖直线，表示侯选项：

```
/Frodo|Pippin|Merry|Sam/ （译注：电影“魔戒”里Shire的四个小矮人）
```

这意味着这些字串的任何都会触发匹配；这个内容在本章稍后的“侯选项”节描述。并且我们还会在那节后面的“捕获和集群”节里告诉你如何使用圆括弧，把你的模式各个部分括起来分组：

```
/(Frodo|Drogo|Bilbo) Baggins/ （译注：Bilbo Baggins 是 Frodo 的叔叔，老一辈魔戒承
```

或者甚至：

```
/(Frod|Drog|Bilb)o Baggins/
```

你看到的其他的東西是我們称之为量詞的東西，它表示在一行里面前面匹配的東西應該匹配幾個。量詞是這些東西：

```
* + ? *? {3} {2,5}
```

不過你永遠不會看到它們這樣獨立地存在。量詞只有附着在原子後面才有意義——也就是說，斷言那些有寬度的（注：量詞有點象第四章，語句和聲明，里的語句修飾詞，也是只能附着在單個語句後面。給一個零寬度的斷言附着量詞就象試圖給一個聲明語句附着一個 **while** 修飾詞一樣——兩種做法都和你跟藥劑師要一斤光子一樣無聊。藥劑師只賣原子什么的。）量詞只附着在前一個原子上，從我們人類的眼光來看，這通常量化為只有一個字符。如果你想匹配一行里的三個“**bar**”的副本，你得用圓括弧把“**bar**”的三個獨立的字符組合成一個“分子”，象這樣：

```
/(bar){3}/
```

這樣將和“**barbarbar**”匹配。如果你用的是 `/bar{3}/`，那麼匹配的是“**barr**”——這東西表明你是蘇格蘭人（译注：愛爾蘭人說英文的時候尾音比較長），而不是 **barbarbar** 人。（話又說回來，也可能不是。我們有些很喜歡的元字符就是愛爾蘭人。）有關量詞的更多東西，參閱後面的“量詞”。

你已經看到了一些繼承了正則表達式的野獸，現在一定迫不及待地想馴服它們。不過，在我們開始認真地討論正則表達式之前，我們需要先向回追溯一些然後再談談使用正則表達式的模式匹配操作符。（並且如果你在学习过程中碰巧多遇到几只“野獸”，那麼不妨给我们的学习向导留一条不错的技巧。）

5.2 模式匹配操作符

從動物學角度來說，Perl 的模式匹配操作符函數是某種用來關正則表達式的籠子。我們是有意這麼設計的；如果我們任由正則怪獸在語言里四處亂逛，Perl 就完全是一個原始叢林了。當然，世界需要叢林——它們是生物種類多樣性的引擎，但是，叢林畢竟應該放在它們應該在的位置。一樣，儘管也是組合多樣化的引擎，正則表達式也應該放在它們應該在的模式匹配操作符里面。那里是另外一個叢林。

因為正則表達式還不夠強大，`m//` 和 `s///` 操作符還提供了（同樣也是限制）雙引號代換的能力。因為模式是按照類似雙引號字串那樣分析的，所以所有的雙引號代換都有效，包括變量代換（除非你用單引號做分隔符）和用反斜杠逃逸標識的特殊字符。（參閱本章後面的“特殊字符”。）在字串被解釋成正則表達式之前首先應用這些代換。（也是 Perl 語言里極少數的幾個地方之一，在這些地方一個字串要經過多於一次處理。）第一次處理是不那麼正常的雙引號代換，不正常是因為它知道它應該轉換什麼和它應該給正則表達式分析器傳遞什麼。因此，任何後面緊跟豎直條，閉圓括弧或者字符串結尾的 `$` 都不會被當作變量代換，而是當作典型的正則表達式的行尾斷言。所以，如果你說：

```
$foo = "bar";
/$foo$/;
```

双引号代换过程是知道那两个 `$` 符作用是不同的。它先做 `$foo` 的变量代换，然后把剩下的交给正则表达式分析器：

```
/bar$/;
```

这种两回合分析的另一个结果是普通的 Perl 记号分析器首先查找正则表达式的结尾，就好像它在查找一个普通字串的结尾分隔符一样。只有在它找到字串的结尾后（并且完成任意变量代换），该模式才被当作正则表达式对待。这意味着你无法在一个正则构造里面“隐藏”模式的结尾分隔符（比如一个字符表或者一个正则注释，我们还没有提到这些东西）。Perl 总是会在任何地方识别该分隔符并且在该处结束该模式。

你还应该知道在模式里面代换变量会降低模式匹配的速度，因为它会觉得需要检查变量是否曾经变化过，如果变化过，那么它必须重新编译模式（这样更会降低速度）。参阅本章后面的“变量代换”。

`tr///` 转换操作符不做变量代换；它甚至连正则表达式都不用！（实际上，它可能并不属于本章，但我们实在想不出更好的地方放它。）不过，它在一个方面还是和 `m//` 和 `s///` 一样的：它用 `=~` 和 `!~` 操作符与变量绑定。

第三章，单目和双目操作符，里描述的 `=~` 和 `!~` 操作符把它们左边的标量表达式和在右边的三个引起类操作符之一绑定在一起：`m//` 用于匹配一个模式，`s///` 用于将某个符合模式的子字符串代换为某个字串，而 `tr///`（或者其同义词，`y///`）用于将一套字符转换成另一套。（如果把斜杠用做分隔符，你可以把 `m//` 写成 `//`，不用写 `m`。）如果 `=~` 或 `!~` 的右手边不是上面三个，它仍然当作是 `m//` 匹配操作，不过此时你已经没有地方放跟在后面的修饰词了（参阅后面的“模式修饰词”），并且你必须操作自己的引号包围：

```
print "matches" if $somestring =~ $somepattern;
```

不过，我们实在没道理不明确地说出来：

```
print "matches" if $somestring =~ m/$somepattern/;
```

当用于匹配操作时，有时候 `=~` 和 `!~` 分别读做“匹配”和“不匹配”（因为“包含”和“不包含”会让人觉得有点模糊）。

除了在 `m//` 和 `s///` 操作符里使用外，在 Perl 的另外两个地方也使用正则表达式。`split` 函数的第一个参数是一个特殊的匹配操作符，它声明的是当把字符串分解成多个子字符串后不返回什么东西。参阅第二十九章，函数，里的关于 `split` 的描述和例子。`qr//`（“引起正则表达”）操作符同样也通过正则表达式声明一个模式，但是它不是为了匹配任何东西（和 `m//` 做的不一样）。相反，编译好的正则表达的形式返回后用于将来的处理。参阅“变量代换”获取更多信息。

你用 `m//`，`s///` 或者 `tr///` 操作符和 `=~`（从语言学上来说，它不是真正的操作符，只是某种形式的标题符）绑定操作符把某一字串和这些操作符之一绑定起来。下面是一些例子：

```
$haystack =~ m/meedle/           # 匹配一个简单模式
$haystack =~ /needle/           # 一样的东西

$italiano =~ s/butter/olive oil/ # 一个健康的替换

$rotate13 =~ tr/a-zA-Z/n-za-mN-ZA-M/ # 简单的加密
```

如果没有绑定操作符，隐含地用 `$_` 做“标题”：

```
/new life/ and                  # 搜索 $_ 和（如果找到）
```

```

/new civilizations/      # 再次更宽范围地搜索 $_

s/sugar/aspartame/      # 把一个替换物替换到 $_ 里

tr/ATCG/TAGC            # 修改在 $_ 里表示的DNA

```

因为 **s///** 和 **tr///** 修改它们所处理的标量，因此你只能把它们用于有效的左值：

```
"onshore" =~ s/on/off/;      # 错；编译时错误
```

不过，**m//** 可以应用于任何标量表达式的结果：

```

if (( lc $magic_hat->fetch_contents->as_string) != /rabbit/) {
    print "Nyaa, what's up doc?\n";
}
else {
    print "That trick never works!\n";
}

```

但是，在这里你得更小心一些，因为 **=~** 和 **!~** 的优先级相当高——在前一个例子里，左边的项的圆括弧是必须的（注：如果没有圆括弧，低优先级的 **lc** 将会应用于整个模式匹配而不只是对 **magic hat** 对象的方法调用。）。**!~** 绑定操作符作用和 **=~** 类似，只是把逻辑结果取反：

```

if ($song !~ /words/) {
    print qq/"$song" appears to be a song without words. \n/;
}

```

因为 **m//**，**s///**，和 **tr///** 都是引号包围操作符，所以你可以选择自己的分隔符。这时其运行方式和引起操作符 **q//**，**qq//**，**qr//**，和 **qw//** 一样（参阅第二章，集腋成裘，中的“选择自己的引号”）。

```

$path =~ s#/tmp#/var/tmp/scratch#;

if ($dir =~ m[/bin]) {
    print "No binary directories please.\n";
}

```

当你把成对的分隔符和 **s///** 或者 **tr///** 用在一起的时候，如果第一部分是四种客户化的括弧对之一（尖括弧，圆括弧，方括弧或者花括弧），那么你可以为第二部分选用不同于第一部分的分隔符：

```

s(egg)<larva>;
s{larva}{pupa};
s[pupa]/imago/;

```

也可以在实际使用的分隔符前面加空白字符：

```

s (egg) <larva>;
s {larva} {pupa};
s [pupa] /imago/;

```

每次成功匹配了一个模式（包括替换中的模式），操作符都会把变量 **\$`**，**\$&**，和 **\$'** 分别设置为匹配内容左边内容，匹配的内容和匹配内容的右边的文本。这个功能对于把字符串分解为组件很有用：

```

"hot cross buns" =~ /cross/;
print "Matched: <$`> $& <$'>\n";      # Matched: <hot > cross < buns>
print "Left:    <$`>\n";                # Left:    <hot >
print "Match:   <$&>\n";                # Match:   <cross>
print "Right:   <$'>\n";                # Right:   < buns>

```


为了有更好的颗粒度和提高效率，你可以用圆括弧捕捉你特别想分离出来的部分。每对圆括弧捕捉与圆括弧内的模式相匹配的子模式。圆括弧由左圆括弧的位置从左到右依次排序；对应那些子模式的子字符串在匹配之后可以通过顺序的变量 `$1`，`$2`，`$3` 等等获得：

```
$_ = "Bilbo Baggins's birthday is September 22";
/(.*)'s birthday is (.*)/;
print "Person: $1\n";
print "Date: $2\n";
```

`$``，`$&`，`$'` 和排序的变量都是全局变量，它们隐含地局部化为属于此闭合的动态范围。它们的存在直到下一次成功的匹配或者当前范围的结尾，以先到者为准。我们稍后在其它课题里有关于这方面内容里更多介绍。

一旦 Perl 认为你的程序的任意部分需要 `$``，`$&`，或 `$'`，它就会为每次模式匹配提供这些东西。这样做会微微减慢你的程序的速度。Perl 同样还利用类似的机制生成 `$1`，`$2` 等等，因此你也会为每个包含捕捉圆括弧的模式付出一些代价。（参阅“集群”获取在保留分组的特征的同时避免捕获的开销的方法。）但如果你从不使用 `$``，`$&` 或者 `$'`，那么不带捕捉圆括弧的模式不会有性能损失。因此，如果可能地话，通常你应该避免使用 `$``，`$&` 和 `$'`，尤其是在库模块里。但是如果你必须至少使用它们一次（而且有些算法的确因此获益非浅），那么你就随便使用它们吧，因为你已经为之付出代价了。在最近的 Perl 版本里，`$&` 比另外两个开销少。

5.2.1 模式修饰词

我们稍后将逐个讨论模式匹配操作符，但首先我们先谈谈另一个这些模式操作符都有的共性：修饰词。

你可以在一个 `m//`，`s///`，`qr//`，或者 `tr///` 操作符的最后一个分隔符后面，以任意顺序放一个或多个单字母修饰词。为了保持清晰，修饰词通常写成“/o 修饰词”并且读做“斜杠 o 修饰词”，即使最后的分隔符可能不是一个斜杠也这么叫。（有时候人们把“修饰词”叫做“标志”或者“选项”也可以。）

有些修饰词改变单个操作符的特性，因此我们将在后面仔细讨论它们。其他的修改正则表达式的解释方式，所以我们在这里讨论它们。`m//`，`s///` 和 `qr//` 操作符（`tr///` 操作符并不接受正则表达式，所以这些修饰词并不适用。）的最后一个分隔符后面都接受下列修饰词：

修饰词	含义
/i	忽略字母的大小写（大小写无关）
/s	令 <code>.</code> 匹配换行符并且忽略不建议使用的 <code>\$*</code> 变量
/m	令 <code>^</code> 和 <code>\$</code> 匹配下一个嵌入的 <code>\n</code> 。
/x	忽略（大多数）空白并且允许模式中的注释
/o	只编译模式一次

/i 修饰词是说同时匹配大写或者小写（以及在 Unicode 里的标题）。也是为什么 `/perl/i` 将匹配字符串 "PROPERLY" 或 "perlaceous"（几乎是完全不同的东西）。`use locale` 用法可能也会对被当作相同的东西有影响。（这可能对包含 Unicode 的字符串有负面影响。）

/s 和 /m 修饰词并不涉及任何古怪的东西。它们只是影响 Perl 对待那些包含换行符的匹配的态度。不过它们和你的字符串是否包含换行符无关；它们关心的是 Perl 是否应该假设你的字符串包含单个行（/s）还是多个行（/m），因为有些元字符根据你是否需要让它们工作于面向行的模式而有不同的行为。

通常，元字符 `.` 匹配除了换行符以外的任何单个字符，因为它的传统含义是匹配一行内的某个字

符。不过，带有 `/s` 时，`"."` 元字符也可以匹配一个换行符，因为你已经告诉 Perl 忽略该字符串可能包含多个换行符的情况。（`/s` 修饰词同样还令 Perl 忽略我们已经不鼓励使用的 `$*` 变量，我们也希望你也忽略。）另一方面，`/m` 修饰词还修改元字符 `^` 和 `$` 的解释——通过令它们匹配字符串里的换行符后面的东西，而不仅仅是字符串的结尾。参阅本章的“位置”节的例子。

`/o` 操作符控制模式的重新编译。除非你选用的分隔符是单引号（`m'PATTERN'`，`s'PATTERN'REPLACEMENT'`，或者 `qr'PATTERN'`），否则每次计算模式操作符的时候，任何模式里的变量都会被代换（并且可能会导致模式的重新编译）。如果你希望这样的模式被且只被编译一次；那么就该使用 `/o` 修饰词。这么做可以避免开销巨大的运行时重新编译；这么做非常有用，尤其是你在转换的值在执行中不会改变的情况下。不过，`/o` 实际上是让你做出了不会改变模式中的变量的承诺。如果你改变了这些变量，Perl 设置都不会注意到。为了更好地控制重编译，你可以使用 `qr//` 正则表达式引起操作符。详情请参阅本章后面的“变量代换”节。

`/x` 是表达修饰词：它允许你利用空白和解释性注释扩展你的模式的易读性，你甚至还可以把模式扩展得超过一行的范围。

也就是说，`/x` 修改空白字符（还有 `#` 字符）的含义：它们不再是普通字符那样的自匹配字符，而是转换成元字符，这些元字符的特征类似空白（和注释字符）。因此，`/x` 允许（在模式里面）将空白，水平制表符和换行符用于格式化，就象普通 Perl 代码一样。它还允许用通常在模式里没有特殊含义的 `#` 字符引入延伸到当前模式行行尾的注释。（注：请注意不要在注释里包含模式分隔符——因为“先找结尾”的规则，Perl 没办法知道你在该点上并不想结束。）如果你想匹配一个真正的空白字符（或者 `#` 字符），那你就要把它们放到字符表里，或者用反斜杠逃逸，或者用八进制或者十六进制逃逸的编码。（但是空白通常用一个 `\s*` 或 `\s+` 序列匹配，因此实际中这种情况出现得并不多。）

总结而言，这些特性朝着把传统的正则表达式变成更可读的语言迈进了一大步。从“回字有四种写法”精神出发，现在写一个正则表达式的方法是不止一种了。实际上，我们不止两种的方法：（译注：TMTOWTDI: "There's More Than One Way To Do It", "做事的方法不止一种". Perl 文化口号，见本书尾部的词汇表。）

```
m/\w+: (\s+\w+) \s* \d+ /;      # 一个词，冒号，空白，词，空白，数字。
m/\w+: (\s+ \w+) \s* \d+ /x;    # 一个词，冒号，空白，词，空白，数字。

m{
    \w+:          # 匹配一个词和一个冒号。
    (            # 分组开始。
        \s+      # 匹配一个或多个空白。
        \w+      # 匹配另外一个词。
    )            # 分组结束。
    \s*          # 匹配零或更多空白。
    \d+          # 匹配一些数字
}x;
```

我们会在本章稍后描述这些元符号。（本节本来是讲模式修饰词的，但是我们却因为对 `/x` 过于兴奋而超出了我们的控制。）下面是一个正则表达式，它找出一个段落里面的重复的词，我们从 Perl Cookbook 里直接把这个例子偷了出来。它使用 `/x` 和 `/i` 修饰词，以及后面描述的 `/g` 修饰词。

```
# 找出段落里面的重复的单词，可能会跨越行界限。
# 将 /x 用于空白和注释，/i 以匹配在 "Is is this ok?" 里的两个 'is'
# 用 /g 找出所有重复。

$/ = "";          # "paragrep" 模式
while( <> ) {
    while ( m{
        \b          # 从字边界开始
```

```

        (\w\S+)          # 找出一个字块
    (
        \s+              # 由一些空白分隔
        \1               # 然后再次分块
    )+                  # 重复动作
    \b                   # 直到另外一个字边界
}xig
)
{
    print "dup word '$1' at paragraph $. \n";
}
}

```

当对本章运行这个程序时，它的输出象下面这样：

dup word 'that' at paragraph 100（译注：只对英文原版有效 :)）

看到这些，我们就知道这个重复是我们有意做的。

5.2.2 m// 操作符（匹配）

```

EXPR =~ m/PATTERN/cgimosx
EXPR =~ /PATTERN/cgimosx
EXPR =~ ?PATTERN?cgimosx
m/PATTERN/cgimosx
/PATTERN/cgimosx
?PATTERN?cgimosx

```

m// 操作符搜索标量 **EXPR** 里面的字串，查找 **PATTERN**。如果使用 **/** 或 **?** 做分隔符，那么开头的 **m** 是可选的。**?** 和 **'** 做分隔符时都有特殊含义：前者表示只匹配一次；后者禁止进行变量代换和六种转换逃逸（**\U** 等，后面描述）。

如果 **PATTERN** 计算出的结果是空字串，则要么是你用 **//** 把它声明成空字串或者是因为一个代换过来的变量就是空字串，这时就用没有隐藏在内层块（或者一个 **split**，**grep**，或者 **map**）里的最后执行成功的正则表达式替代。

在标量环境里，该操作符在成功时返回真（**1**），失败时返回假（**""**）。这种形式常见于布尔环境：

```

if($shire =~ m/Baggins/) { ...}    # 在 $shire 里找Baggins, 译注: shire 知道哪里
if($shire =~ /Baggins/) { ...}    # 在 $shire 里找Baggins

if(m#Baggins#) { ...}             # 在 $_ 里找
if( /Baggins/ ) { ...}            # 在 $_ 里找

```

在列表环境里使用，**m//** 返回一个子字串的列表，这些子字串匹配模式里的捕获圆括弧（也就是 **\$1**，**\$2**，**\$3** 等等），这些捕获圆括弧将在稍后的“捕获和集群”里描述。当列表返回的时候，这些序列数变量仍然是平滑的。如果在列表环境里匹配失败，则返回一个空列表。如果在列表环境中匹配成功，但是没有使用捕获圆括弧（也没有 **/g**），则返回则返回一列（**1**）。因此它在失败时返回一列空列表，所以这种形式的 **m//** 仍然能用于布尔环境，但是仅限于通过列表赋值间接参与的情况：

```

if( ($key, $values) = /(\w+): (.*)/) { ... }

```

用于 **m//**（不管是什么形式）的合法修饰词见表 5-1。

表 5-1。 **m//** 修饰词

--	--

修饰词	含义
/i	或略字母大小写
/m	令 ^ 和 \$ 匹配随后嵌入的 \n。
/s	令 . 匹配换行符并且忽略废弃了的 \$*。
/x	或略（大多数）空白并且允许在模式里的注释
/o	只编译模式一次
/g	全局地查找所有匹配
/cg	在 /g 匹配失败后允许继续查找

头五个用于正则表达式的修饰词我们前面描述过了。后面两个修改匹配操作本身的特性。**/g** 修饰词声明一个全局匹配——也就是说，在该字符串里匹配尽可能多的次数。它的具体特性取决于环境。在列表环境里，**m//g** 返回所有找到的东西的列表。下面的语句找出所有我们提到的 "perl"，"Perl"，"PERL" 的地方：

```
if( @perls = $paragraph =~ /perl/gi) {
    printf "Perl mentioned %d times.\n", scalar @perls;
}
```

如果在 **/g** 模式里没有捕获圆括弧，那么返回完整的匹配。如果有捕获圆括弧，那么返回捕获到的字符串。想象一下这样的字符串：

```
$string = "password=xyzzzy verbose=9 score=0";
```

并且假设你想用这个字符串初始化下面这样的散列：

```
%hash = (password => "xyzzzy", verbose => 9, socre => 0);
```

当然，你有字符串但还没有列表。要获取对应的列表，你可以在列表环境里用 **m//g** 操作符，从字符串里捕获所有的键/值对：

```
%hash = $string =~ /(\w+)= (\w+)/g;
```

(\w+) 序列捕获一个字母数字单词。参阅“捕获和集群”节。

在标量环境里使用时，**/g** 修饰词表明一次渐进地匹配，它令 Perl 从上一次匹配停下来的位置开始一次对同一个变量的新的匹配。**\G** 断言表示字符串中的那个位置；**\G** 的描述请参阅本章后面的“位置”一节。如果除了用 **/g**，你还用了 **/c**（表示“连续”）修饰词，那么当 **/g** 运行结束后，失败的匹配不会重置位置指针。

如果分隔符是 **?**，就象 **?PATTERN?**，那么运行起来和 **/PATTERN/** 搜索一样，区别是它在两次 **reset** 操作符调用之间只匹配一次。如果你只想匹配程序运行中模式出现的第一次，而不是所有的出现，那么这是一个很方便的优化方法。你每次调用此操作符时都会运行搜索，直到它最终匹配了什么东西，然后它就关闭自身，在你明确地用 **reset** 把它重置之前它一直返回假。Perl 替你跟踪这个匹配状态。

当一个普通模式匹配想找出最后一个匹配而不是第一个，那么 **??** 操作符很好用：

```
open DICT, "/usr/dict/words" or die "Can't open words: $!\n";
while (<DICT>) {
    $first = $1 if ?(^neur.*)?;
    $last = $1 if /(^neur.*)/;
}
print $first, "\n";      # 打印"neurad"
print $last, "\n";      # 打印 "neurypnology"
```

在调用 `reset` 操作符时，`reset` 只重置那些编译进同一个包的 ?? 记录。你说 `m??` 的时候等效于说 ??。

5.2.3 s/// 操作符（替换）

```
LVALUE =~ s/PATTERN/REPLACEMENT/egimosx
s/PATTERN/REPLACEMENT/egimosx
```

这个操作符在字符串里搜索 `PATTERN`，如果找到，则用 `REPLACEMENT` 文本替换匹配的子字符串。（修饰词在本节稍后描述。）

```
$lotr = $hobbit;      # 只是拷贝Hobbit 译注：影片魔戒里，Hobbit 人住在 Shire :)
$lotr =~ s/Bilbo/Frodo/g;  # 然后用最简单的方法写结局，译注：Frodo 代替 Bilbo 成
```

一个 `s///` 操作符的返回值（在标量和列表环境里都差不多）是它成功的次数（如果与 `/g` 修饰词一起使用，返回值可能大于一）。如果失败，因为它替换了零次，所以它返回假（""），它等效于数字 0。

```
if( $lotr =~ s/Bilbo/Frodo/) { print "Successfully wrote sequel." }
$change_count = $lotr =~ s/Bilbo/Frodo/g;
```

替换部分被当作双引号包围的字符串看待。你可以在替换字符串里使用我们前面描述过的任何动态范围的模式变量（`$``，`$&`，`$'`，`$1`，`$2`，等等），以及任何其他你准备使用的双引号包围的小发明。比如下面是一个小例子，用于找出所有字符串 `"revision"`，`"version"`，或者 `"release"`，并且用对应的大写字串替换，我们可以用 `\u` 逃逸处理替换的目标部分：

```
s/revision|version|release/\u$&/g;  # | 用于表示模式中的“或”
```

所有的标量变量都在双引号包围的环境中扩展开，而不仅仅是这些特殊的变量。假设你有一个散列 `%Names`，把版本号映射为内部的项目名；比如，`$Name{"3.0"}` 可能是名为 `"Isengard"` 的代码名。你可以用 `s///` 找出版本号并且用它们对应的项目名替换掉：

```
s/version ([0-9.]+)/the $Names{$1} release/g;
```

在在替换字符串里，`$1` 返回第一对（也是唯一的一对）捕获圆括弧。（愿意的话你还可以在模式里用 `\1`，但是这个用法在替换中已经废弃了，在一个普通的双引号包围的字符串里，`\1` 的意思是 `Control-A`。）

如果 `PATTERN` 是一个空字符串，则使用上一次成功执行的正则表达式取代。`PATTERN` 和 `REPLACEMENT` 都需要经受变量代换，不过每次计算 `s///` 操作符的时候都进行代换，而 `REPLACEMENT` 只是在有匹配的时候才做变量代换。（如果你使用了 `/g` 修饰词，那么 `PATTERN` 在一次计算中可能匹配多次。）

和前面一样，表 5-2 中的头五个修饰词修改正则表达式的性质；他们与 `m//` 和 `qr//` 中的一样。后面两个修改替换操作符本身。

表 5-2 s/// 修饰词

修饰词	含义
/i	或略字母大小写
/m	令 <code>^</code> 和 <code>\$</code> 匹配随后嵌入的 <code>\n</code> 。
/s	令 <code>.</code> 匹配换行符并且忽略废弃了的 <code>\$*</code> 。
/x	或略（大多数）空白并且允许在模式里的注释
/o	只编译模式一次

/g	全局地查找所有匹配
/e	把右边当作一个表达式计算

/g 修饰词用于 **s///** 的时候就会把每个匹配 **PATTERN** 的东西用 **REPLACEMENT** 值替换，而不仅仅是所找到的第一个。一个 **s///g** 操作符的作用象一次全局的搜索和替换，令所有修改同时发生，很象 **m//g**，只不过 **m//g** 不改变任何东西。（而且 **s///g** 也和标量 **m//g** 不一样，它不是递增匹配。）

/e 修饰词把 **REPLACEMENT** 当作一个 Perl 代码块，而不仅仅是一个替换的字串。执行这段代码后得出的结果当作替换字串使用。比如，**s/(0-9+)/sprintf("%#x", \$1)/ge** 将把所有数字转换成十六进制，比如，把 **2581** 变成 **0xb23**。或者，假设在我们前一个例子里，你不知道是否所有版本都有名称，因此，你希望把这些没有名称的保留不动。可以利用稍微有点创造力的 **/x** 格式，你可以说：

```
s{
    version
    \s+
    (
        [0-9.]+
    )
}{
    $Names{$1}
    ? "the $Names{$1} release"
    : $&
}xge;
```

你的 **s///e** 的右边（或者本例中的下半部分）在编译时与你的程序的其他部分一起做语法检查和编译。在编译过程中，任何语法错误都会被检测到，而运行时例外则被忽略。在第一个 **/e** 后面每多一个 **e**（象 **/ee**，**/eee** 等等）都等效于对生成的代码调用 **eval STRING**，每个 **/e** 相当于一次调用。这么做等于计算了代码表达式的结果并且把例外俘获在特殊变量 **\$@** 里。参阅本章后面的“编程化模式”获取更多信息。

5.2.3.1 顺便修改一下字串

有时候你想要一个新的，修改过的字串，而不是在旧字串上一阵乱改，新字串以旧字串为基础。你不用写：

```
$lotr = $hobbit;
$lotr =~ s/Bilbo/Frodo/g;
```

你可以把这些组合成一个语句。因为优先级关系，必须在赋值周围使用圆括弧，因为它们大多和使用了 **=~** 的表达式结合在一起。

```
( $lotr = $hobbit ) =~ s/Bilbo/Frodo/g;
```

如果没有赋值语句周围的圆括弧，你只修改了 **\$hobbit** 并且把替换的个数存储在 **\$lotr** 里，那样会得到很傻的结局。

你不能对数组直接使用 **s///** 操作符。这时，你需要一个循环。幸运的是，**for/foreach** 的别名特性加上它把 **\$_** 当作缺省循环变量，这样就产生了 Perl 标准的用于搜索和替换一个数组里每个元素的俗语：

```
for (@chapters) { s/Bilbo/Frodo/g }      # 一章一章的替换
s/bilbo/Frodo/g for @chapters;           # 一样的东西
```

就象一个简单的标量变量一样，如果你想把初始的值保留在其他地方，你也可以把替换和赋值结合在一起：

```
@oldhues = ('bluebird', 'bluegrass', 'bluefish', 'the blues');
for (@newhues = @oldhues) { s/blue/red/}
print "@newhues\n";      # 打印: redbird redgrass redfish the reds
```

对同一个变量执行重复替换的最经典的方法是用一个单程循环。比如，下面是规范变量里的空白的方法：

```
for ($string) {
    s/^\s+//;    # 丢弃开头的空白
    s/\s+$//;    # 丢弃结尾的空白
    s/\s+/ /g;   # 压缩内部的空白
}
```

这个方法正好和下面的是一样的：

```
$string = join(" ", split " ", $string);
```

你还可以把这样的循环和赋值用在一起，就象我们在数组的例子所做的那样：

```
for( $newshow = $oldshow ) {
    s/Fred/Homer/g;
    s/Wilma/Marge/g;
    s/Pebbles/Lisa/g;
    s/Dino/Bart/g;
}
```

5.2.3.2 当全局替换不够“全局”地时候

有时候，你用 `/g` 不能实现全部修改的发生，这时要么是因为替换是从右向左发生的，要么是因为你要求 `$`` 的长度在不同的匹配之间改变。通常你可以通过反复调用 `s///` 做你想做的事情。不过，通常你希望当 `s///` 失败的时候循环停下来，因此你必须把它放进条件里，这样又让循环的主体无所事事。因此我们只写一个 `1`，这也是一件无聊的事情，不过有时候无聊比没希望好。下面是一些例子，它们又用了一些正则表达式怪兽：

```
# 把逗号放在一个整数的合理的位置
1 while s/(\d) (\d\d\d) (?! \d) /$1,$2/;

# 把水平制表符扩展为八列空间
1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;

# 删除（嵌套（甚至深层嵌套（象这样）））的括弧
1 while s/\([^(]*)\)/g;

# 删除重复的单词（以及三重的（和四重的。。。））
1 while s/\b(\w+) \1\b/$1/gi;
```

最后一个需要一个循环是因为如果没有循环，它会吧：

```
Paris in THE THE THE THE spring.
```

转换成：

```
Paris in THE THE spring.
```

这看起来会让那些懂点法文的人觉得巴黎位于一个喷冰茶的喷泉中间，因为“the”（法文）是法文“tea”的单词。当然，巴黎人从来不会上当。

5.2.4 tr/// 操作符（转换）

```
LVALUE =~ tr/SEARCHLIST/REPLACEMENTLIST/cds
tr/SEARCHLIST/REPLACEMENTLIST/cds
```

对于 `sed` 的爱好者而言，`y///` 就是 `tr///` 的同义词。这就是为什么你不能调用名为 `y` 的函数，同样也不能调用名为 `q` 或 `m` 的函数。在所有的其他方面，`y///` 都等效于 `tr///`，并且我们不会再提及它。

把这个操作符放进关于模式匹配的章节看起来其实有点不合适，因为它不使用模式。这个操作符逐字符地扫描一个字串，然后把每个在 `SEARCHLIST`（不是正则表达式）里出现的字符替换成对应的来自 `REPLACEMENTLIST`（也不是替换字串）的字符。但是它看上去象 `m//` 和 `s///`，你甚至还可以和它一起使用 `=~` 和 `!~` 绑定操作符，因此我们在这里描述它。（`qr//` 和 `split` 都是模式匹配操作符，但是它们不能和绑定操作符一起使用，因此因此在我们本书的别处描述它们。自己找找看。）

转换返回替换或者删除了的字符个数。如果没有通过 `=~` 或者 `!~` 操作符声明的字串，那么使用 `$_` 字串。`SEARCHLIST` 和 `REPLACEMENTLIST` 可以用一个划线定义一个顺序字符的范围：

```
$message =~ tr/A-Za-z/N-ZA-Mn-za-m?;    # 旋转13加密
```

请注意想 `A-Z` 这样的范围假设你用的是线性字符集，比如 `ASCII`。但是不同的字符集的字符排列顺序是不一样的。一个合理的原则是，只使用起始点都是相同大小写的字母序列，如 `(a-e,A-E)`，或者数字 `(0-4)`。任何其他范围都有问题。如果觉得有问题，他们写成你用的整个字符集：`ABCDE`。

`SEARCHLIST` 和 `REPLACEMENTLIST` 都不会象双引号字串那样进行变量代换；不过，你可以使用那些映射为特殊字符的反斜杠序列，比如 `\n` 或 `\015`。

表 5-3 是可用于 `tr///` 操作符的修饰词。它们和用于 `m//`，`s///`，或 `qr//` 上的完全不同，即使有些看起来有点象。

表 5-3. `tr///` 修饰词

修饰词	含义
<code>/c</code>	与 <code>SEARCHLIST</code> 为补
<code>/d</code>	删除找到的但是没有替换的字符
<code>/s</code>	消除重复的字符。

如果声明了 `/c` 修饰词，那么 `SEARCHLIST` 里的字符会被求补；也就是说，实际的搜索列表包含所有不在 `SEARCHLIST` 里的字符。如果是 `Unicode`，这样可能会代表许多字符，不过因为它们是逻辑存储的，而不是物理存储，所以你不用害怕会用光内存。

`/d` 修饰词把 `tr///` 转换成所谓的“过滤吸收”操作符：任何由 `SEARCHLIST` 声明的但是没有在 `REPLACEMENTLIST` 里给出替换的字符将被删除。（这样比一些 `tr(1)` 程序的性质显得更加灵活，那些程序删除它们在 `SEARCHLIST` 里找到的任何东西。）

如果声明了 `/s` 修饰词，被转换成相同字符的顺序字符将被压缩成单个字符。

如果使用了 `/d` 修饰词，那么 `REPLACEMENTLIST` 总是严格地解释成声明的样子。否则，如果 `REPLACEMENTLIST` 比 `SEARCHLIST` 短，则复制 `REPLACEMENTLIST` 的最后一个字符，直到足够长为止。如果 `REPLACEMENTLIST` 为空，则复制 `SEARCHLIST`，这一点虽然奇怪，但很有用，尤其是当你只是想计算字符数，而不是改变它们的时候。也有利于用 `/s` 压缩字符。


```

tr/aeiou/!/;          # 把所有元音字母转换成!
tr{/\|\\r\n\b\f. }{ _};      # 把怪字符转成下划线

tr/A-Z/a-z/ for @ARGV;      # 把字符规则化为小写ASCII

$count = ($para =~ tr/\n//);  # 计算$para里的换行符
$count = tr/0-9//;          # 计算$_里的位

$word =~ tr/a-zA-Z//s;      # bookkeeper -> bokeper

tr/@$%*//d;              # 删除这里几个字符
tr#A-Za-z0-9+/##cd;      # 删除非base64字符

# 顺便修改
($HOST = $host) =~ tr/a-z/A-Z/;

$pathname =~ tr/a-zA-Z/_/cs;  # 把非ASCII字母换成下划线

tr [\200-\377]
    {\000-\177};          # 剥除第八位，字节操作

```

如果在 **SEARCHLIST** 里同一个字符出现的次数多于一次，那么只有第一个有效。因此：

```
tr/AAA/XYZ/
```

将只会把（\$_ 里的）任何单个字符 **A** 转换成 **X**。

尽管变量不会代换进入 **tr///**，但是你还是可以用 **eval** **EXPR** 实现同样效果：

```

$count = eval "tr/$oldlist/$newlist/";
die if $@;  # 传播非法eval内容的例外

```

最后一条信息：如果你想把你的文本转换为大写或者小写，不要用 **tr///**。用双引号里的 **\U** 或者 **\L** 序列（或者等效的 **uc** 和 **lc** 函数），因为它们会关心区域设置或 **Unicode** 信息，而 **tr/a-z/A-Z/** 不关心这些。另外在 **Unicode** 字符串里，**\u** 序列和它的对应 **ucfirst** 函数能够识别标题格式，对某些语言来说，比简单地转换成大写更突出。

5.3 元字符和元符号

既然我们尊重这些神奇的笼子，那么我们就可以回过头来看看笼子里的动物了，也就是那些你放在模式里好看的符号。到现在你应该已经看到这样的事实，就是这些符号并不是普通的函数调用或者算术操作符那样的 **Perl** 代码。正则表达式本身就是嵌入 **Perl** 的小型语言。（在现实社会里总是有小丛林。）

Perl 里的模式识别所有的 12 个传统的元字符（所谓十二烂人），以及它们的所有潜能和表现力。许多其他正则表达式包里也能看到它们：

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

它们中有些曲解规则，令跟在它们后面本来正常的字符变成特殊的。我们不喜欢把长序列叫做“字符”，因此，如果它们组成长序列后，我们叫它们元符号（有时候干脆就叫“符号”）。但是在顶级，这十二个元字符就是你（和 **Perl**）需要考虑的所有内容。任何东西都是从这里开始的。

有些简单的元字符就代表它们自己，象 **.** 和 **^** 和 **\$**。它们并不直接影响它们周围的任何东西。有些元字符运行起来象前缀操作符，控制任何跟在后面的东西，象反斜杠 “****”。其他的则像后缀操作

符，控制紧排在它们前面的东西，像 `*`，`+`，和 `?`。有一个元字符：`|`，其作用象中缀操作符，站在它控制的操作数中间。甚至还有括弧操作符，作用类似包围操作符，控制一些被它包围的东西，像 `(...)` 和 `[...]`。圆括弧尤其重要，因为它们在内部声明 `|` 的范围，而在外部声明 `*`，`+` 和 `?` 的范围。

如果你只学习十二个元字符中的一个，那么选反斜杠。（恩。。。还有圆括弧）这是因为反斜杠令其他元字符失效。如果在一个 Perl 模式里，一个反斜杠放在一个非字母数字字符前，这样就让下一个字符成为一个字面的文本。如果你象在一个模式文本里匹配十二个元字符中的任何一个，你可以在它们前面写一个反斜杠。因此，`\.` 匹配一个真正的点，`\$` 匹配真正的美圆符，`\\` 是一个真正的反斜杠等等。这样做被称做“逃逸”元字符，或曰“引号包围之”，或者有时候就叫做“反斜杠某”。（当然，你已经知道反斜杠可以用于禁止在双引号字符串里进行变量代换。）

虽然一个反斜杠把一个元字符转换成一个文本字符，它对后继的字母数字字符的作用却是完全另一码事。它把本来普通的东西变特别。也就是说，它们在一起形成元字符。我们在表 5-7 里给出了一个按字母排序的元字符表。

5.3.1 元字符表

符号	原子性	含义
<code>\...</code>	变化	反逃逸下一个非字母数字字符，转意下一个字母数字（可能）
<code>... ...</code>	否	可选（匹配前者或者后者）。
<code>(...)</code>	是	分组（当作单元对待）。
<code>[...]</code>	是	字符表（匹配一个集合中的一个字符）。
	否	如果在字符串开头则为真（或者可能是在任何换行符后面。）
<code>.</code>	是	匹配一个字符（通常除了换行符以外）。
<code>\$</code>	否	在字符串尾部时为真（或者可能是在任何换行符前面）。

至于量词，我们会在它们自己的节里详细描述。量词表示前导的原子（也就是说，单字符或者分组）应该匹配的次数。它们列在表 5-5 中。

表 5-5。 正则量词

量词	原子性	含义
<code>*</code>	否	匹配 0 或者更多次数（最大）。
<code>+</code>	否	匹配 或者更多次数（最大）。
<code>?</code>	否	匹配 1 或者0次（最大）。
<code>{COUNT}</code>	否	匹配COUNT 次
<code>{MIN,}</code>	否	匹配至少MIN次（最大）。
<code>{MIN,MAX}</code>	否	匹配至少MIN次但不超过MAX次（最大）
<code>*?</code>	否	匹配0或者更多次（最小）
<code>+?</code>	否	匹配1或者更多次（最小）
<code>??</code>	否	匹配0或者1次（最小）
<code>{MIN,}? </code>	否	匹配最多MIN次（最小）
<code>{MIN,MAX}? </code>	否	匹配至少MIN次但不超过MAX次（最小）

最小量词会试图匹配在它的许可范围内的尽可能少的次数。最大量词会试图匹配在它的许可范围内的尽可能多的次数。比如，`.+` 保证至少匹配字符串的一个字符，但是如果有机会，它会匹配所有机会。这里的机会将在稍后的“小引擎的/能与不能/”节里讲。

你还会注意量词决不能量化。

我们想给新类型的元符号一个可以扩展的语法。因为我们只需要使用十二个元字符，所以我们选用原先被认为是非法正则的序列做任意语法扩展。这些元符号的形式都是 **(?KEY...)**；也就是，一个开圆括弧后面跟着一个问号，然后是 **KEY** 和模式其余部分。**KEY** 字符表明它是哪种正则扩展。参阅表 5-6 看看正则扩展的一个列表。它们中大多数性质象列表，因为它们基于圆括弧，不过它们还是有附加含义。同样，只有原子可以量化，因为它们代表真正存在（潜在地）的东西。

表 5-6 扩展的正则序列

扩展	原子性	含义
(?# ...)	否	注释，抛弃
(?:...)	是	只集群，不捕获的圆括弧
(?imsx-imsx)	否	打开/关闭模式修饰词
(?imsx-imsx:...)	是	集群圆括弧加修饰词
(?= ...)	否	如果前向查找断言成功，返回真
(?!...)	否	如果前向查找断言失败，返回真
(?<=...)	否	如果前向查找断言成功，返回真
(?<...)	否	如果前向查找断言失败，返回真
(?>...)	是	匹配未反向跟踪的子模式
(?{...})	否	执行嵌入的Perl代码
(??{...})	是	匹配来自嵌入Perl代码。
(?(...)...)	是	匹配if-then-else模式
(?(...)...)	是	匹配if-then模式

最后，表 5-7 显示了所有你常用的字母数字元符号。（那些在变量代换回合处理过的符号在原子性列里用一个划线标记，因为引擎永远不会看到它们。）

表 5-7. 字母数字正则元符号

符号	原子性	含义
\0	是	匹配空字符（ASCII NUL）。
\NNN	是	匹配给出八进制的字符，最大值为\377。
\n	是	匹配前面第n个捕获字串（十进制）。
\a	是	匹配警钟字符（BEL）。
\A	否	如果在字串的开头为真
\b	是	匹配退各字符（BS）。
\B	否	不在字边界时为真
\cX	是	匹配控制字符 Control-x（\cZ，\c[，等）。
\C	是	匹配一个字节（C字符），甚至在utf8中也如此（危险）
\d	是	匹配任何数字字符
\D	是	匹配任何非数字字符
\e	是	匹配逃逸字符（ASCII ESC，不是反斜杠）。
\E	——	结束大小写（\L，\U）或者掩码（\Q）转换
\f	是	匹配进页字符（FF）。
\G	否	如果在前一个m//g的匹配结尾位置时为真
\l	——	只把下一个字符变成小写

<code>\L</code>	——	把\E以前的字母都变成小写
<code>\n</code>	是	匹配换行符字符（通常是NL，但是在Mac上是CR）。
<code>\N{NAME}</code>	是	匹配命名字符（ <code>\N{greek:Sigma}</code> ）。
<code>\p{PROP}</code>	是	匹配任何有命名属性的字符
<code>\P{PROP}</code>	是	匹配任何没有命名属性的字符
<code>\Q</code>	——	引起（消元）直到\E前面的字符
<code>\r</code>	是	匹配返回字符（通常是CR，但是在Mac上是NL）。
<code>\s</code>	是	匹配任何空白字符。
<code>\S</code>	是	匹配任何非空白字符。
<code>\t</code>	是	匹配水平制表符（HT）。
<code>\u</code>	——	只把下一个字符变成标题首字符
<code>\U</code>	——	大写（不是标题首字符）\E 以前的字符。
<code>\w</code>	是	匹配任何“字”字符（字母数字加“_”）。
<code>\W</code>	是	匹配任何“非字”字符。
<code>\x{abcd}</code>	是	匹配在十六进制中给出的字符。
<code>\X</code>	是	匹配Unicode里的”组合字符序列““字串。
<code>\z</code>	否	只有在字串结尾时为真
<code>\Z</code>	否	在字串结尾或者在可选的换行符之前为真。

如果在 `\p` 和 `\P` 里的属性名字是一个字符，那么花括弧是可选的。如果 `\x` 里的十六进制数为两位或者更少，那么花括弧也是可选的。在 `\N` 里的花括弧决不能省略。

只有在含义中带“匹配。。。 ”或者“匹配任何。。。 ”字样的元符号才能够在字符表（方括弧）里面使用。也就是说，字符表仅限于包含特定的字符集，因此在字符表里面，你只能使用那些描述其他特定字符集的元符号，或者那些描述特定独立字符的元符号。当然，这些元符号也可以和其他非分类元符号一起在字符表外面用，不过，这里请注意 **=b** 是两只完全不同的怪兽：它在字符表内是退格字符，而在外边是一个字边界断言。

一个模式可以匹配的字符的数量和一个双引号字串可以替换的字符的数量有一些重叠。因为正则要经历两个回合，所以有时候应该由哪个回合处理一个给定的字符会显得有些混乱。如果出现混乱，这种字符的变量代换回合就推迟给正则表达式分析器。

但是只有当变量代换回合知道它正在分析一个正则的时候，它才能把变量代换推迟给正则分析器。你可以把正则表达式声明为普通的双引号包围字串，但这样你就必须遵循普通的双引号包围规则。任何前面碰巧映射为时间字符的元符号仍然生效，即使它们没有被推迟给正则分析器也如此。但是在普通的双引号里你不能使用任何其它的元符号（或者任何类似的构造，比如 ``...``，`qq(...)`，`qx(...)`，或者等效的“此处”文档）。如果你想你的字串分析成一个正则表达式而不做任何匹配，你应该使用 `qr//`（引号构造正则）操作符。

请注意大小写和元引号包围转换逃逸（`\U` 和它的伙伴）必须在变量代换回合处理，因为这些元符号的用途就是影响变量代换。如果你用单引号禁止变量代换，那么你也无法获得转换逃逸。在任何单引号字串里，都不会进行变量或者转换逃逸（`\U`等）的扩展，在单号包围的 `m'...'` 或者 `qr'...'` 操作符里也不会。甚至在你做代换的时候，如果这些转换逃逸是一个变量代换的结果，那么它们也会被忽略，因为这个时候他们想要影响变量代换已经太晚了。

尽管字符替换操作符不处理正则表达式，但是我们讨论过的任何匹配单个特定字符的元符号在 `tr///` 操作中仍然可用。而其他的用不了（除了反斜杠以外，它继续按照它原来的样子运转。）

5.3.2 特定的字符

如前所述，非特殊的东西在模式里匹配自身。这意味着一个 `/a/` 匹配一个 `"a"`，一个 `/=/` 匹配一个 `"="` 等等。不过，有些字符可不是那么容易在键盘上敲进去，或者即使你敲进去了，也不会在打印输出中显示出来；最臭名昭著的例子就是控制字符。在正则表达式里，Perl 识别下列双引号包围的字符别名：

逃逸	含义
<code>\0</code>	空字符（ASCII NUL）
<code>\a</code>	警铃（BEL）
<code>\e</code>	逃逸（ESC）
<code>\f</code>	进纸（FF）
<code>\n</code>	换行符（NL，Mac里的CR）
<code>\r</code>	回车（CR，Mac里的NL）
<code>\t</code>	水平制表符（HT）

就象在双引号包围字符串里一样，Perl 还识别模式里的下面四种元符号：

`\cX`

一个命名的控制字符，象 `\cC` 指 Control-C，`\cZ` 指 Control-Z，`\c[` 指 ESC，而 `\c?` 表示 DEL。

`\NNN`

用两位或者三位八进制码声明的字符。除了小于 `010` 的数值（十进制 8）外，前导的 0 是可选的，因为（和在双引起字符串里不同）一位数字的东西总认为是用于在模式里捕获字符串的引用。如果你在模式里先捕获了至少 `n` 个子字符串，那么多位数字解释成第 `n` 个引用（这里 `n` 被认为是一个十进制数）。否则，他们被解释成一个用八进制声明的字符。

`\x{LONGHEX}`

一个用一到两个十六进制数位（`[0-9a-fA-F]`）声明的字符，比如 `\x1B`。一位数字的形式只有在后面跟随的字符不是一个十六进制数字才可用。如果使用了花括弧，你想用多少位数字都可以，这时候结果可能是一个 Unicode 字符。比如，`\x{262}` 匹配一个 Unicode YIN YANG。

`\N{NAME}`

一个命名字符，如 `\N{GREEK SMALL LETTER EPSILON}`，`\N{Greek:epsilon}`，或者 `\N{epsilon}`。它要求使用第三十一章，用法模块，里描述的 `use charnames` 用法，它同时还判断你可能使用那些名字中的哪一个（分别是 `":long"`，`":full"`，`":short"`，对应上面三个风格。）

你可以在离你最近的 Unicode 标准文档里找到所有 Unicode 字符名字列表，或者在 `PATH_TO_PERLLIB/unicode/Names.txt` 里也有。

5.3.3 通配元符号

三个特殊的元符号可以用做通用通配符，它们的每一个都可以匹配"任何"字符（是"任何"中的某些字符）。它们是句点（`"."`），`\c` 和 `\x`。它们都不能在字符表里使用。你不能在字符表里用句点是因为它会匹配（几乎）任何存在的字符，因此它本身就是某种万能字符。如果你想包括或者排除所有东西，也没有什么必要使用一个字符表。特殊通配符 `\C` 和 `\X` 有着特殊的结构化含义，而这些特殊含

义和选择单个 **Unicode** 字符的表示法关联得并不好，而该表示法才是字符表运行的层次。

句点元字符匹配除了换行符以外的任何单字符。（如果带着 `/s` 修饰词，也能匹配换行符。）和十二个特殊字符里的其它字符一样，如果你想匹配一个文本句点，你就必须用一个反斜杠来逃逸它。比如，下面的代码检查一个文件名是否以一个句点后面跟着一个单字符扩展名结尾的：

```
if ($pathname =~ /\.(.)\z/s) {
    print "Ends in $1\n";
}
```

第一个句点是逃逸了的，是文本句点，而第二个句点说“匹配任何字符”。`\z` 说只匹配字符串末尾的东西，而 `\s` 修饰词句点也可以匹配换行符。（的确，用换行符做文件扩展名不怎么漂亮，但并不是说就不能做。）

点元字符经常和量词一起使用。`.*` 匹配尽可能多的字符，而 `.*?` 匹配尽可能少的字符。不过有时候它不用量词而是自己解决长度问题：`/(..):(..):(..)/` 匹配三个用冒号分隔的域，每个域两个字符长。

如果你在一个 `use utf8` 用法的词法范围里编译的模式里使用一个点，那么它就匹配任何 **Unicode** 字符。（你可能不需要用 `use utf8`，不过偶然还是会发生的，在你阅读到这里的时候你可能不需要这个用法。）

```
use utf8;
use charnames qw/:full/;
%BWV[887] = "G\N{MUSIC SHARP SIGN} minor";
($note, $black, $mode) = $BWV[886] =~ /^([A-G])(.)\s+(\S+)/;
print "That's lookin' sharp!\n" if $black eq chr(9839);
```

元符号 `\X` 在更广的概念上匹配字符。它实际上是匹配一个由一个或多个 **Unicode** 字符组成的字符串，这个字符串就是所谓的“组合字符序列”。这样的序列包括一个基本字符和后面跟着任意个“标志”字符（象变音符和分音符那样的区分标志）一起组成一个逻辑单元。`\X` 实际上等效于 `(?:\PM\pM*)`。这样做允许匹配一个逻辑字符，即使这几个字符实际上是由几个独立的字符组成的也行。如果匹配任意组合字符，那么在 `/\X/` 里匹配的长度将超过一个字符长。（而且这里的字符长度和字节长度没有什么关系）。

如果你在使用 **Unicode** 并且真的想获取单字节而不是单字符，那么你可以使用 `\C` 元字符。它将总是匹配一个字节（具体说，就是一个 C 语言的 `char` 类型），而不管是否会与你的 **Unicode** 字符流步调失调。参阅第十五章里关于做这些事情时合适的警告。

5.4 字符表

在模式匹配里，你可以匹配任意字符，不管它们有没有特殊性质。有四种声明字符表的方法（译注：孔乙己？：）。你可以按照传统的方法声明字符集——用方括弧和枚举可能的字符，或者或者你可以使用三种记忆法中的任意一种：经典 **Perl** 表，新 [PerlUnicode](#) 属性，或者标准 **POSIX** 表。这些缩写均只匹配其字符集中的一个字符。你可以量化它们，使它们可以匹配更多的字符，比如 `\d+` 匹配一个或者多个数字。（一个比较容易犯的错误是认为 `\w` 匹配一个字。用 `\w+` 匹配一个字。）

5.4.1 客户化字符表

一个方括弧中的一个枚举字符列表被称为字符表，它匹配列表中的任何一个字符。比如，`[aeiouy]` 匹配一个英文中的元音字母。（对于威尔士要加 `"w"`，对于苏格兰加个 `"r"`。）要匹配一个右方括弧，你可以用反斜杠来逃逸之或者把它放在列表开头。

字符范围可以用一个连字符和 `a-z` 表示法表示。你可以合并多个范围；比如 `[0-9a-fA-F]` 匹配一个十六进制“位”。你可以用反斜杠来避免连字符被解释为一个范围分隔符，或者把它放在表的开头或者

结尾（后面的方法虽然不易读，但是比较常用）。

在字符表开头的脱字符（或者说是抑扬符号，或者帽子，或者向上箭头 "^"）反转该字符表，结果是匹配任何不在此列表中的字符。（要匹配脱字符，要么不要放在开头，或者用反斜杠逃逸）。比如，`[^aeiouy]` 匹配任何不是元音的字母。不过，对待字符表反意要小心些，因为字符的领域在不断扩展。比如，那个字符表匹配辅音——而在西利尔语，希腊语和几乎任何语言里还匹配空白，换行符和其他任何东西（包括元音），更不用说中日韩文里的标记了。而且以后还可能有 **Cirth**，**Tengwar**，和 **Klingon**。（当然，还可能有 **Linear B** 和 **Etruscan**）所以你最好还是明确声明你的辅音，比如`[cbdfghjklmnpqrstvwxyz]`，或者简写为 `[b-df-hj-p-tv-z]`。（这样还解决了“y”需要在两个地方同时出现的问题，排除了一个补集。）

字符表里支持普通字符元符号，（参阅“声明字符”），比如 `\n`，`\t`，`\cX`，`\NNN`，和 `\N{NAME}`。另外，你可以在一个字符表里使用 `\b` 表示一个退格，就象它在双引号串里显示的那样。通常，在一个模式匹配里，它意味着一个字边界。但是零宽度的断言在字符表里没有任何意义，因此这里的 `\b` 返回到它在字串里的普通含义。你还可以使用我们本章稍后预定义的任何字符表（经典，**Unicode** 或 **POSIX**），但是不要把它们用在一个范围的结尾——那样没有意义，所以 `"-` 会被解释成文本。

所有其他的元符号在方括弧中都失去特殊意义。实际上，你不能在这里面使用三个普通通配符中的任何一个：`"."`，`\X` 或 `\C`。不允许第一个字符通常令人奇怪，不过把普遍意义的字符表用做有限制的形式的确没有什么意义，而且你常会在一个字符表中要用到文本句点——比如，当你要匹配文件名的时候。而且在字符表里声明量词，断言或者候选都是没有意义的，因为这些字符都是独立解释的。比如，`[fee|fie|foe|foo]` 和 `[feio]` 是一样的。

5.4.2 典型 Perl 字符表缩写

从一开始，Perl 就已经提供了一些字符表缩写。它们在表 5-8 中列出。它们都是反斜杠字母元字符，而且把它们的字母换成大写后，它们的含义就是小写版本的反义。这些元字符的含义并不像你想象的那么固定，其含义可能在新的 **Unicode** 标准出台后改变，因为新标准会增加新的数字和字母。

（为了保持旧的字节含义，你总是可以使用 `use bytes`。要解释 `utf8` 的含义，参阅本章后面的“**Unicode** 属性”。不管怎样，`utf8` 含义都是字节含义的超集。）

表 5-8 典型字符表

符号	含义	做字节	做 utf8
<code>\d</code>	数字	<code>[0-9]</code>	<code>\p{IsDigit}</code>
<code>\D</code>	非数字	<code>[^0-9]</code>	<code>\P{IsDigit}</code>
<code>\s</code>	空白	<code>[\t\n\r\f]</code>	<code>\p{IsSpace}</code>
<code>\S</code>	非空白	<code>[^\t\n\r\f]</code>	<code>\P{IsSpace}</code>
<code>\w</code>	字	<code>[a-zA-Z0-9_]</code>	<code>\p{IsWord}</code>
<code>\W</code>	非字	<code>[^a-zA-Z0-9_]</code>	<code>\P{IsWord}</code>

（好好好，我们知道大多数数字里面没有数字和下划线，`\w` 的用意是用于匹配典型的编程语言里的记号。就目前而言，是匹配 Perl 里的记号。）

这些元符号在方括弧外面或者里面都可以使用，也就是说不管作为独立存在的符号还是作为一个构造成的字符表的一部分存在都行：

```
if ($var =~ /\D/) { warn "contains non-digit" }
if ($var =~ /[^\w\s.]/) { warn "contains non-(word, space, dot)"}

```

5.4.3 Unicode 属性

Unicode 属性可以用 `\p{PROP}` 及其补集 `\P{PROP}` 获取。对于那些比较少见的名字里只有一个字符的属性，花括弧是可选的，就象 `\pN` 表示一个数字字符（不一定是十进制数 — 罗马数字也是数字字符）。这些属性表可以独自使用或者与一个字符表构造一起使用：

```
if ($var =~ /^ \p{IsAlpha}+$/) {print "all alphabetic" }
if ($var =~ s/[ \p{Zl} \p{Zp}]/\n/g) {print "fixed newline wannabes" }
```

有些属性是直接由 Unicode 标准定义的，而有些属性是 Perl 基于标准属性组合定义的，`Zl` 和 `Zp` 都是标准 Unicode 属性，分别代表行分隔符和段分隔符，而 `IsAlpha`² 是 Perl 而 `IsAlpha`² 是 Perl 定义的，是一个组合了标准属性 `Li`, `Lu`, `Lt`, 和 `Lo`（也就是小写，大写，标题或者其它字母）的属性表。在 Perl 5.6.0 里，要想用这些属性，你得用 `use utf8`。将来会放松这个限制。

还有很多其它属性。我们会列出我们知道的那些，但是这个列表肯定不完善。很可能在新版本的 Unicode 里会定义新的属性，而且你甚至也可以定义自己的属性。稍后我们将更详细地介绍这些。

Unicode 委员会制作了在线资源，这些资源成为 Perl 用于其 Unicode 实现里的各种文件。关于更多这些文件的信息，请参阅第 15 章。你可以在文档 `PATH_TO_PERLLIB/unicode/Unicode3.html` 里获得很不错的关于 Unicode 的概述性介绍。这里 `PATH_TO_PERLLIB` 是下面命令的打印输出：

```
perl -MConfig -le 'print $Config{privlib}'
```

大多数 Unicode 的属性形如 `\p{IsPROP}`。`Is` 是可选的，因为它太常见了，不过你还是会愿意把它们写出来的，因为可读性好。

5.4.3.1 Perl 的 Unicode 属性

首先，表 5-9 列出了 Perl 的组合属性。它们定义得合理地接近于标准 POSIX 定义的字符表。

表5-9. 组合 Unicode 属性

属性	等效
<code>IsASCII</code> ²	<code>[\x00-\x7f]</code>
<code>IsAlnum</code> ²	<code>[\p{IsLi}\p{IsLu}\p{IsLt}\p{IsLo}\p{IsNd}]</code>
<code>IsAlpha</code> ²	<code>[\p{IsLi}\p{IsLu}\p{IsLt}\p{IsLo}]</code>
<code>IsCntrl</code> ²	<code>\p{IsC}</code>
<code>IsDigit</code> ²	<code>\p{Nd}</code>
<code>IsGraph</code> ²	<code>[^\pC\p{IsSpace}]</code>
<code>IsLower</code> ²	<code>\p{IsLi}</code>
<code>IsPrint</code> ²	<code>\P{IsC}</code>
<code>IsPunct</code> ²	<code>\p{IsP}</code>
<code>IsSpace</code> ²	<code>[\t\n\f\r\p{IsZ}]</code>
<code>IsUpper</code> ²	<code>[\p{IsLu}\p{IsLt}]</code>
<code>IsWord</code> ²	<code>[_\p{IsLi}\p{IsLu}\p{IsLt}\p{IsLo}\p{IsNd}]</code>
<code>IsXDigit</code> ²	<code>[0-9a-fA-F]</code>

Perl 还为标准 Unicode 属性（见下节）的每个主要范畴提供了下列组合：

属性	含义	是否规范的
<code>IsC</code> ²	错误控制代码等	是
<code>IsL</code> ²	字母	部分

IsM²	标志	是
IsN²	数字	是
IsP²	标点	否
IsS²	符号	否
IsZ²	分隔符	是

5.4.3.2 标准的 Unicode 属性

表 5-10 列出了大多数基本的标准 Unicode 属性，源自每个字符的类别。没有哪个字符是多于一个类别的成员。有些属性是规范的，而有些只是提示性的。请参阅 [Unicode](#) 标准获取那些标准的说辞，看看什么是规范信息，而什么又是提示信息。

表 5-10 标准 Unicode 属性

属性	含义	规范化
IsCc²	其他，控制	是
IsCf²	其他，格式	是
IsCn²	其他，未赋值	是
IsCo²	其他，私有使用	是
IsCs²	其他，代理	是
IsLl²	字母，小写	是
IsLm²	字母，修饰词	否
IsLo²	字母，其他	否
IsLt²	字母，抬头	是
IsLu²	字母，大写	是
IsMc²	标记，组合	是
IsMe²	标记，闭合	是
IsMn²	标记，非空白	是
IsNd²	数字，十进制数	是
IsNI²	数字，字母	是
IsNo²	数字，其他	是
IsPc²	标点，联接符	否
IsPd²	标点，破折号	否
IsPe²	标点，关闭	否
IsPf²	标点，结束引用	否
IsPi²	标点，初始引用	否
IsPo²	标点，其他	否
IsPs²	标点，打开	否
IsSc²	符号，货币	否
IsSk²	符号，修饰词	否
IsSm²	符号，数学	否
IsSo²	符号，其他	否
IsZl²	分隔符，行	是
IsZp²	分隔符，段落	是
IsZs²	分隔符，空白	是

另外一个有用的属性集是关于一个字符是否可以分解为更简单的字符。（规范分解或者兼容分解）。规范分解不会丢失任何格式化信息。兼容分解可能会丢失格式信息，比如一个字符是否上标等。

属性	信息丢失
IsDecoCanon[?]	无
IsDecoCompat[?]	有一些（下列之一）
IsDCcircle[?]	字符周围的圆
IsDCfinal[?]	最终的位置（阿拉伯文）
IsDCfont[?]	字体变体的选择
IsDCfraction[?]	俚语字符片段
IsDCinitial[?]	起始位置选择（阿拉伯文）
IsDCisolated[?]	隔离位置选择（阿拉伯文）
IsDCmedial[?]	中间位置选择（阿拉伯文）
IsDCnarrow[?]	窄字符
IsDCnoBreadk[?]	空白或连字符的非中断选
IsDCsmall[?]	小字符
IsDCsquare[?]	CJK字符周围的方块
IsDCsub[?]	脚标
IsDCsuper[?]	上标
IsDCvertical[?]	旋转（水平或垂直）
IsDCwide[?]	宽字符
IsDCcompat[?]	标识（杂项）

下面是那些对双向书写的人感兴趣的属性：

属性	含义
IsBidiL[?]	从左向右（阿拉伯语，希伯来语）
IsBidiLRE[?]	从左向右插入
IsBidiLRO[?]	从左向右覆盖
IsBidiR[?]	从右向左
IsBidiAL[?]	阿拉伯语从右向左
IsBidiRLE[?]	从右向左插入
IsBidiRLO[?]	从右向左覆盖
IsBidiPDF[?]	流行方向的格式
IsBidiEN[?]	欧洲数字
IsBidiES[?]	欧洲数字分隔符
IsBidiET[?]	欧洲数字结束符
IsBidiAN[?]	阿拉伯数字
IsBidiCS[?]	普通数字分隔符
IsBidiNSM[?]	非空白标记
IsBidiBN[?]	与边界无色
IsBidiB[?]	段落分隔符
IsBidiS[?]	段分隔符
IsBidiWS[?]	空白

IsBidiON?	其他无色字符
IsMirrored?	当使用从右向左模式时反向

下面的属性根据元音的发音把它们分类为各种音节:

IsSylA	IsSylE	IsSylO	IsSylWAA	IsSylWII
IsSylAA	IsSylEE	IsSylOO	IsSylWC	IsSylWO
IsSylAAI	IsSylI	IsSylU	IsSylWE	IsSylWOO
IsSylAI	IsSylII	IsSylV	IsSylWEE	IsSylWU
IsSylC	IsSylN	IsSylWA	IsSylWI	IsSylWV

比如, `\p{IsSylA}` 将匹配 `\N{KATAKANA LETTER KA}`, 但不匹配 `\N{KATAKANA LETTER KU}`。

既然你现在基本上已经知道了所有的这些 **Unicode 3.0** 属性, 那我们还要说的是在版本 **5.6** 的 **Perl** 里有几个比较秘传的属性还没有实现, 因为它们的实现有一部分是基于 **Unicode 2.0** 的, 而且, 象那些双向的算法还在我们的制作之中。不过, 等到你读到这些的时候, 那些缺失的属性可能早就实现了, 所以我们还是把它们列出来了。

Revision: r1.13 - 14 Oct 2005 - 04:53 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [PatternMatching](#)

版权 © 1999-2006 归这里所有作者. [PostgreSQL](#) 的中文文档版权归何伟平所有.
 向为这里贡献想法,文章的人致敬 [PostgreSQL](#) 中文网
[反馈意见](#)