深入浅出之正则表达式（一）

前言：

半年前我对正则表达式产生了兴趣，在网上查找过不少资料，看过不少的教程，最后在使用一个正则表达式工具 RegexBuddy 时发现他的教程写的非常好，可以说是我目前见过最好的正则表达式教程。于是一直想把他翻译过来。这个愿望直到这个五一长假才得以实现，结果就有了这篇文章。关于本文的名字，使用"深入浅出"似乎已经太俗。但是通读原文以后，觉得只有用"深入浅出"才能准确的表达出该教程给我的感受，所以也就不能免俗了。

本文是 Jan Goyvaerts 为 RegexBuddy 写的教程的译文，但是为了尊重原作者和译者的劳动，请注明出处！谢谢！

1. 什么是正则表达式

基本说来，正则表达式是一种用来描述一定数量文本的模式。Regex 代表 Regular Express。本文将用<<regex>>来表示一段具体的正则表达式。

一段文本就是最基本的模式，简单的匹配相同的文本。

2. 不同的正则表达式引擎

正则表达式引擎是一种可以处理正则表达式的软件。通常，引擎是更大的应用程序的一部分。在软件世界，不同的正则表达式并不互相兼容。本教程会集中讨论 Perl 5 类型的引擎，因为这种引擎是应用最广泛的引擎。同时我们也会提到一些和其他引擎的区别。许多近代的引擎都很类似，但不完全一样。例如.NET 正则库，JDK 正则包。

3. 文字符号

最基本的正则表达式由单个文字符号组成。如<<a>>，它将匹配字符串中第一次出现的字符“a”。如对字符串“Jack is a boy”。“J”后的“a”将被匹配。而第二个“a”将不会被匹配。

正则表达式也可以匹配第二个“a”，这必须是你告诉正则表达式引擎从第一次匹配的地方开始搜索。在文本编辑器中，你可以使用“查找下一个”。在编程语言中，会有一个函数可以使你从前一次匹配的位置开始继续向后搜索。

类似的，<<cat>>会匹配“About cats and dogs”中的“cat”。这等于是告诉正则表达式引擎，找到一个<<c>>，紧跟一个<<a>>，再跟一个<<t>>。

要注意，正则表达式引擎缺省是大小写敏感的。除非你告诉引擎忽略大小写，否则<<cat>>不会匹配“Cat”。

- 特殊字符

对于文字字符，有 12 个字符被保留作特殊用途。他们是：

[ ] \ ^ $. | ? * + ( )

这些特殊字符也被称作元字符。

如果你想在正则表达式中将这些字符用作文本字符，你需要用反斜杠“\”对其进行换码 (escape)。例如你想匹配“1+1=2”，正确的表达式为<<1\+1=2>>.

需要注意的是，<<1+1=2>>也是有效的正则表达式。但它不会匹配“1+1=2”，而会匹配“123+111=234”中的“111=2”。因为“+”在这里表示特殊含义（重复 1 次到多次）。

在编程语言中，要注意，一些特殊的字符会先被编译器处理，然后再传递给正则引擎。因此正则表达式<<1\+2=2>>在 C++中要写成“1\\+1=2”。为了匹配“C:\temp”，你要用正则表达式<<C:\\temp>>。而在 C++中，正则表达式则变成了“C:\\\\temp”。

- 不可显示字符

  可以使用特殊字符序列来代表某些不可显示字符：

  <<\t>>代表 Tab(0x09)

  <<\r>>代表回车符(0x0D)

  <<\n>>代表换行符(0x0A)

  要注意的是 Windows 中文本文件使用“\r\n”来结束一行而 Unix 使用“\n”。

4. 正则表达式引擎的内部工作机制

知道正则表达式引擎是如何工作的有助于你很快理解为何某个正则表达式不像你期望的那样工作。

有两种类型的引擎：文本导向(text-directed)的引擎和正则导向(regex-directed)的引擎。Jeffrey Friedl 把他们称作 DFA 和 NFA 引擎。本文谈到的是正则导向的引擎。这是因为一些非常有用的特性，如“惰性”量词(lazy quantifiers)和反向引用(backreferences)，只能在正则导向的引擎中实现。所以毫不意外这种引擎是目前最流行的引擎。

你可以轻易分辨出所使用的引擎是文本导向还是正则导向。如果反向引用或“惰性”量词被实现，则可以肯定你使用的引擎是正则导向的。你可以作如下测试：将正则表达式<<regex|regex not>>应用到字符串“regex not”。如果匹配的结果是 regex，则引擎是正则导向的。如果结果是 regex not，则是文本导向的。因为正则导向的引擎是“猴急”的，它会很急切的进行表功，报告它找到的第一个匹配。

- 正则导向的引擎总是返回最左边的匹配

  这是需要你理解的很重要的一点：即使以后有可能发现一个“更好”的匹配，正则导向的引擎也总是返回最左边的匹配。

  当把<<cat>>应用到“He captured a catfish for his cat”，引擎先比较<<c>>和“H”，结果失败了。于是引擎再比较<<c>>和“e”，也失败了。直到第四个字符，<<c>>匹配了“c”。<<a>>匹配了第五个字符。到第六个字符<<t>>没能匹配“p”，也失败了。引擎再继续从第五个字符重新检查匹配性。直到第十五个字符开始，<<cat>>匹配上了“catfish”中的“cat”，正则表达式引擎急切的返回第一个匹配的结果，而不会再继续查找是否有其他更好的匹配。

5. 字符集

字符集是由一对方括号“[]”括起来的字符集合。使用字符集，你可以告诉正则表达式引擎仅仅匹配多个字符中的一个。如果你想匹配一个“a”或一个“e”，使用<<[ae]>>。你可以使用<<gr[ae]y>>匹配 gray 或 grey。这在你不确定你要搜索的字符是采用美国英语还是英国英语时特别有用。相反，<<gr[ae]y>>将不会匹配 graay 或 graey。字符集中的字符顺序并没有什么关系，结果都是相同的。

你可以使用连字符“-”定义一个字符范围作为字符集。<<[0-9]>>匹配 0 到 9 之间的单个数字。你可以使用不止一个范围。<<[0-9a-fA-F] >>匹配单个的十六进制数字，并且大小

写不敏感。你也可以结合范围定义与单个字符定义。<<[0-9a-fxA-FX]>>匹配一个十六进制数字或字母 X。再次强调一下，字符和范围定义的先后顺序对结果没有影响。

- 字符集的一些应用

  查找一个可能有拼写错误的单词，比如<<sep[ae]r[ae]te>> 或 <<li[cs]en[cs]e>>。

  查找程序语言的标识符，<<A-Za-z_][A-Za-z_0-9]*>>。(*表示重复 0 或多次)

  查找 C 风格的十六进制数<<0[xX][A-Fa-f0-9]+>>。(+表示重复一次或多次)

- 取反字符集

  在左方括号“[”后面紧跟一个尖括号“^”，将会对字符集取反。结果是字符集将匹配任何不在方括号中的字符。不像“.”，取反字符集是可以匹配回车换行符的。

  需要记住的很重要的一点是，取反字符集必须要匹配一个字符。<<q[^u]>>并不意味着：匹配一个 q，后面没有 u 跟着。它意味着：匹配一个 q，后面跟着一个不是 u 的字符。所以它不会匹配“Iraq”中的 q，而会匹配“Iraq is a country”中的 q 和一个空格符。事实上，空格符是匹配中的一部分，因为它是一个“不是 u 的字符”。

  如果你只想匹配一个 q，条件是 q 后面有一个不是 u 的字符，我们可以用后面将讲到的向前查看来解决。

- 字符集中的元字符

  需要注意的是，在字符集中只有 4 个 字符具有特殊含义。它们是：“] \ ^ -”。“]”代表字符集定义的结束；“\”代表转义；“^”代表取反；“-”代表范围定义。其他常见的元字符在字符集定义内部都是正常字符，不需要转义。例如，要搜索星号*或加号+，你可以用<<[+*]>>。当然，如果你对那些通常的元字符进行转义，你的正则表达式一样会工作得很好，但是这会降低可读性。

  在字符集定义中为了将反斜杠“\”作为一个文字字符而非特殊含义的字符，你需要用另一个反斜杠对它进行转义。<<[\\x]>>将会匹配一个反斜杠和一个 X。“]^-”都可以用反斜杠进行转义，或者将他们放在一个不可能使用到他们特殊含义的位置。我们推荐后者，因为这样可以增加可读性。比如对于字符“^”，将它放在除了左括号“[”后面的位置，使用的都是文字字符含义而非取反含义。如<<[x^]>>会匹配一个 x 或^。<<[]x]>>会匹配一个“]”或“x”。<<[-x]>>或<<[x-]>>都会匹配一个“-”或“x”。

- 字符集的简写

  因为一些字符集非常常用，所以有一些简写方式。

  <<\d>>代表<<[0-9]>>;

  <<\w>>代表单词字符。这个是随正则表达式实现的不同而有些差异。绝大多数的正则表达式实现的单词字符集都包含了<<A-Za-z0-9_]>>。

  <<\s>>代表“白字符”。这个也是和不同的实现有关的。在绝大多数的实现中，都包含了空格符和 Tab 符，以及回车换行符<<\r\n>>。

  字符集的缩写形式可以用在方括号之内或之外。<<\s\d>>匹配一个白字符后面紧跟一个数字。<<[\s\d]>>匹配单个白字符或数字。<<[\da-fA-F]>>将匹配一个十六进制数字。

  取反字符集的简写

  <<[\S]>> = <<[^\s]>>

  <<[\W]>> = <<[^\w]>>

  <<[\D]>> = <<[^\d]>>

- 字符集的重复

如果你用“?*+”操作符来重复一个字符集，你将会重复整个字符集。而不仅是它匹配的那个字符。正则表达式<<[0-9]+>>会匹配 837 以及 222。

如果你仅仅想重复被匹配的那个字符，可以用向后引用达到目的。我们以后将讲到向后引用。

6. 使用?*或+ 进行重复

?：告诉引擎匹配前导字符 0 次或一次。事实上是表示前导字符是可选的。

+：告诉引擎匹配前导字符 1 次或多次

*：告诉引擎匹配前导字符 0 次或多次

<[A-Za-z][A-Za-z0-9]*>匹配没有属性的 HTML 标签，“<”以及“>”是文字符号。第一个字符集匹配一个字母，第二个字符集匹配一个字母或数字。

我们似乎也可以用<[A-Za-z0-9]+>。但是它会匹配<1>。但是这个正则表达式在你知道你要搜索的字符串不包含类似的无效标签时还是足够有效的。

- 限制性重复

许多现代的正则表达式实现，都允许你定义对一个字符重复多少次。词法是：{min,max}。min 和 max 都是非负整数。如果逗号有而 max 被忽略了，则 max 没有限制。如果逗号和 max 都被忽略了，则重复 min 次。

因此{0,}和*一样，{1，}和+ 的作用一样。

你可以用<<\b[1-9][0-9]{3}\b>>匹配 1000~9999 之间的数字(“\b”表示单词边界)。<<\b[1-9][0-9]{2,4}\b>>匹配一个在 100~99999 之间的数字。

- 注意贪婪性

假设你想用一个正则表达式匹配一个 HTML 标签。你知道输入将会是一个有效的 HTML 文件，因此正则表达式不需要排除那些无效的标签。所以如果是在两个尖括号之间的内容，就应该是一个 HTML 标签。

许多正则表达式的新手会首先想到用正则表达式<< <.+> >>，他们会很惊讶的发现，对于测试字符串，“This is a <EM>first</EM> test”，你可能期望会返回<EM>，然后继续进行匹配的时候，返回</EM>。

但事实是不会。正则表达式将会匹配“<EM>first</EM>”。很显然这不是我们想要的结果。原因在于“+”是贪婪的。也就是说，“+”会导致正则表达式引擎试图尽可能的重复前导字符。只有当这种重复会引起整个正则表达式匹配失败的情况下，引擎会进行回溯。也就是说，它会放弃最后一次的“重复”，然后处理正则表达式余下的部分。

和“+”类似，“?*”的重复也是贪婪的。

- 深入正则表达式引擎内部

让我们来看看正则引擎如何匹配前面的例子。第一个记号是“<”，这是一个文字符号。第二个符号是“.”，匹配了字符“E”，然后“+”一直可以匹配其余的字符，直到一行的结束。然后到了换行符，匹配失败(“.”不匹配换行符)。于是引擎开始对下一个正则表达式符号进行匹配。也即试图匹配“>”。到目前为止，“<.+”已经匹配了“<EM>first</EM> test”。引擎会试图将“>”与换行符进行匹配，结果失败了。于是引擎进行回溯。结果是现在“<.+”匹配“<EM>first</EM> tes”。于是引擎将“>”与“t”进行匹配。显然还是会失败。这个

过程继续，直到“<.+”匹配“<EM>first</EM”，“>”与“>”匹配。于是引擎找到了一个匹配“<EM>first</EM>”。记住，正则导向的引擎是“急切的”，所以它会急着报告它找到的第一个匹配。而不是继续回溯，即使可能会有更好的匹配，例如“<EM>”。所以我们可以看到，由于“+”的贪婪性，使得正则表达式引擎返回了一个最左边的最长的匹配。

- 用懒惰性取代贪婪性

一个用于修正以上问题的可能方案是用“+”的惰性代替贪婪性。你可以在“+”后面紧跟一个问号“?”来达到这一点。“*”，“{}”和“?”表示的重复也可以用这个方案。因此在上面的例子中我们可以使用“<.+?>”。让我们再来看看正则表达式引擎的处理过程。

再一次，正则表达式记号“<”会匹配字符串的第一个“<”。下一个正则记号是“.”。这次是一个懒惰的“+”来重复上一个字符。这告诉正则引擎，尽可能少的重复上一个字符。因此引擎匹配“.”和字符“E”，然后用“>”匹配“M”，结果失败了。引擎会进行回溯，和上一个例子不同，因为是惰性重复，所以引擎是扩展惰性重复而不是减少，于是“<.+”现在被扩展为“<EM”。引擎继续匹配下一个记号“>”。这次得到了一个成功匹配。引擎于是报告“<EM>”是一个成功的匹配。整个过程大致如此。

- 惰性扩展的一个替代方案

我们还有一个更好的替代方案。可以用一个贪婪重复与一个取反字符集：“<[^>]+>”。之所以说这是一个更好的方案在于使用惰性重复时，引擎会在找到一个成功匹配前对每一个字符进行回溯。而使用取反字符集则不需要进行回溯。

最后要记住的是，本教程仅仅谈到的是正则导向的引擎。文本导向的引擎是不回溯的。但是同时他们也不支持惰性重复操作。

7. 使用“.”匹配几乎任意字符

在正则表达式中，“.”是最常用的符号之一。不幸的是，它也是最容易被误用的符号之一。

“.”匹配一个单个的字符而不用关心被匹配的字符是什么。唯一的例外是新行符。在本教程中谈到的引擎，缺省情况下都是不匹配新行符的。因此在缺省情况下，“.”等于是字符集[^\n\r](Window)或[^\n](Unix)的简写。

这个例外是因为历史的原因。因为早期使用正则表达式的工具是基于行的。它们都是一行一行的读入一个文件，将正则表达式分别应用到每一行上去。在这些工具中，字符串是不包含新行符的。因此“.”也就从不匹配新行符。

现代的工具和语言能够将正则表达式应用到很大的字符串甚至整个文件上去。本教程讨论的所有正则表达式实现都提供一个选项，可以使“.”匹配所有的字符，包括新行符。在 RegexBuddy, EditPad Pro 或 PowerGREP 等工具中，你可以简单的选中“点号匹配新行符”。在 Perl 中，“.”可以匹配新行符的模式被称作“单行模式”。很不幸，这是一个很容易混淆的名词。因为还有所谓“多行模式”。多行模式只影响行首行尾的锚定(anchor)，而单行模式只影响“.”。

其他语言和正则表达式库也采用了 Perl 的术语定义。当在.NET Framework 中使用正则表达式类时，你可以用类似下面的语句来激活单行模式：

Regex.Match(“string”,”regex”,RegexOptions.SingleLine)

- 保守的使用点号“.”

点号可以说是最强大的元字符。它允许你偷懒：用一个点号，就能匹配几乎所有的字符。但是问题在于，它也常常会匹配不该匹配的字符。

我会以一个简单的例子来说明。让我们看看如何匹配一个具有“mm/dd/yy”格式的日期，但是我们想允许用户来选择分隔符。很快能想到的一个方案是<<\d\d.\d\d.\d\d>>。看上去它能匹配日期“02/12/03”。问题在于 02512703 也会被认为是一个有效的日期。

<<\d\d[-/.]\d\d[-/.]\d\d>>看上去是一个好一点的解决方案。记住点号在一个字符集里不是元字符。这个方案远不够完善，它会匹配“99/99/99”。而<<[0-1]\d[-/.][0-3]\d[-/.]\d\d>>又更进一步。尽管他也会匹配“19/39/99”。你想要你的正则表达式达到如何完美的程度取决于你想达到什么样的目的。如果你想校验用户输入，则需要尽可能的完美。如果你只是想分析一个已知的源，并且我们知道没有错误的数据，用一个比较好的正则表达式来匹配你想要搜寻的字符就已经足够。

8. 字符串开始和结束的锚定

锚定和一般的正则表达式符号不同，它不匹配任何字符。相反，他们匹配的是字符之前或之后的位置。“^”匹配一行字符串第一个字符前的位置。<<^a>>将会匹配字符串“abc”中的 a。<<^b>>将不会匹配“abc”中的任何字符。

类似的，$匹配字符串中最后一个字符的后面的位置。所以<<c$>>匹配“abc”中的 c。

- 锚定的应用

在编程语言中校验用户输入时，使用锚定是非常重要的。如果你想校验用户的输入为整数，用<<^\d+$>>。

用户输入中，常常会有多余的前导空格或结束空格。你可以用<<^\s*>>和<<\s*$>>来匹配前导空格或结束空格。

- 使用“^”和“$”作为行的开始和结束锚定

如果你有一个包含了多行的字符串。例如：“first line\n\rsecond line”(其中\n\r 表示一个新行符)。常常需要对每行分别处理而不是整个字符串。因此，几乎所有的正则表达式引擎都提供一个选项，可以扩展这两种锚定的含义。“^”可以匹配字串的开始位置(在 f 之前)，以及每一个新行符的后面位置(在\n\r 和 s 之间)。类似的，$会匹配字串的结束位置(最后一个 e 之后)，以及每个新行符的前面(在 e 与\n\r 之间)。

在.NET 中，当你使用如下代码时，将会定义锚定匹配每一个新行符的前面和后面位置：Regex.Match("string", "regex", RegexOptions.Multiline)

应用：string str = Regex.Replace(Original, "^", "> ", RegexOptions.Multiline)--将会在每行的行首插入“> ”。

- 绝对锚定

<<\A>>只匹配整个字符串的开始位置，<<\Z>>只匹配整个字符串的结束位置。即使你使用了“多行模式”，<<\A>>和<<\Z>>也从不匹配新行符。

即使\Z 和$只匹配字符串的结束位置，仍然有一个例外的情况。如果字符串以新行符结束，则\Z 和$将会匹配新行符前面的位置，而不是整个字符串的最后面。这个“改进”是由 Perl 引进的，然后被许多的正则表达式实现所遵循，包括 Java，.NET 等。如果应用<<^[a-z]+$>>到“joe\n”，则匹配结果是“joe”而不是“joe\n”。

深入浅出之正则表达式（二）

前言：

本文是前一片文章《深入浅出之正则表达式（一）》的续篇，在本文中讲述了正则表达式中的组与向后引用，先前向后查看，条件测试，单词边界，选择符等表达式及例子，并分析了正则引擎在执行匹配时的内部机理。

9. 单词边界

元字符<<\b>>也是一种对位置进行匹配的“锚”。这种匹配是 0 长度匹配。

有 4 种位置被认为是“单词边界”：

1) 在字符串的第一个字符前的位置(如果字符串的第一个字符是一个“单词字符”)
2) 在字符串的最后一个字符后的位置(如果字符串的最后一个字符是一个“单词字符”)
3) 在一个“单词字符”和“非单词字符”之间，其中“非单词字符”紧跟在“单词字符”之后
4) 在一个“非单词字符”和“单词字符”之间，其中“单词字符”紧跟在“非单词字符”后面

“单词字符”是可以用“\w”匹配的字符，“非单词字符”是可以用“\W”匹配的字符。在大多数的正则表达式实现中，“单词字符”通常包括<<[a-zA-Z0-9_]>>。

例如：<<\b4\b>>能够匹配单个的 4 而不是一个更大数的一部分。这个正则表达式不会匹配“44”中的 4。

换种说法，几乎可以说<<\b>>匹配一个“字母数字序列”的开始和结束的位置。

“单词边界”的取反集为<<\B>>，他要匹配的位置是两个“单词字符”之间或者两个“非单词字符”之间的位置。

- 深入正则表达式引擎内部

让我们看看把正则表达式<<\bis\b>>应用到字符串“This island is beautiful”。引擎先处理符号<<\b>>。因为\b 是 0 长度，所以第一个字符 T 前面的位置会被考察。因为 T 是一个“单词字符”，而它前面的字符是一个空字符(void)，所以\b 匹配了单词边界。接着<<i>>和第一个字符“T”匹配失败。匹配过程继续进行，直到第五个空格符，和第四个字符“s”之间又匹配了<<\b>>。然而空格符和<<i>>不匹配。继续向后，到了第六个字符“i”，和第五个空格字符之间匹配了<<\b>>，然后<<is>>和第六、第七个字符都匹配了。然而第八个字符和第二个“单词边界”不匹配，所以匹配又失败了。到了第 13 个字符 i，因为和前面一个空格符形成“单词边界”，同时<<is>>和“is”匹配。引擎接着尝试匹配第二个<<\b>>。因为第 15 个空格符和“s”形成单词边界，所以匹配成功。引擎“急着”返回成功匹配的结果。

10. 选择符

正则表达式中“|”表示选择。你可以用选择符匹配多个可能的正则表达式中的一个。

如果你想搜索文字“cat”或“dog”，你可以用<<cat|dog>>。如果你想有更多的选择，你只要扩展列表<<cat|dog|mouse|fish>>。

选择符在正则表达式中具有最低的优先级，也就是说，它告诉引擎要么匹配选择符左边的所有表达式，要么匹配右边的所有表达式。你也可以用圆括号来限制选择符的作用范围。如<<\b(cat|dog)\b>>，这样告诉正则引擎把(cat|dog)当成一个正则表达式单位来处理。

- 注意正则引擎的“急于表功”性

正则引擎是急切的，当它找到一个有效的匹配时，它会停止搜索。因此在一定条件下，选择符两边的表达式的顺序对结果会有影响。假设你想用正则表达式搜索一个编程语言的函数列表：Get，GetValue，Set 或 SetValue。一个明显的解决方案是<<Get|GetValue|Set|SetValue>>。让我们看看当搜索 SetValue 时的结果。

因为<<Get>>和<<GetValue>>都失败了，而<<Set>>匹配成功。因为正则导向的引擎都是“急切”的，所以它会返回第一个成功的匹配，就是“Set”，而不去继续搜索是否有其他更好的匹配。

和我们期望的相反，正则表达式并没有匹配整个字符串。有几种可能的解决办法。一是考虑到正则引擎的“急切”性，改变选项的顺序，例如我们使用<<GetValue|Get|SetValue|Set>>，这样我们就可以优先搜索最长的匹配。我们也可以把四个选项结合起来成两个选项：<<Get(Value)?|Set(Value)?>>。因为问号重复符是贪婪的，所以 SetValue 总会在 Set 之前被匹配。

一个更好的方案是使用单词边界：<<\b(Get|GetValue|Set|SetValue)\b>>或<<\b(Get(Value)?|Set(Value)?\b>>。更进一步，既然所有的选择都有相同的结尾，我们可以把正则表达式优化为<<\b(Get|Set)(Value)?\b>>。

11. 组与向后引用

把正则表达式的一部分放在圆括号内，你可以将它们形成组。然后你可以对整个组使用一些正则操作，例如重复操作符。

要注意的是，只有圆括号“()”才能用于形成组。“[]”用于定义字符集。“{}”用于定义重复操作。

当用“()”定义了一个正则表达式组后，正则引擎则会把被匹配的组按照顺序编号，存入缓存。当对被匹配的组进行向后引用的时候，可以用“\数字”的方式进行引用。<<\1>>引用第一个匹配的后向引用组，<<\2>>引用第二个组，以此类推，<<\n>>引用第 n 个组。而<<\0>>则引用整个被匹配的正则表达式本身。我们看一个例子。

假设你想匹配一个 HTML 标签的开始标签和结束标签，以及标签中间的文本。比如<B>This is a test</B>，我们要匹配<B>和</B>以及中间的文字。我们可以用如下正则表达式：“<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>”

首先，“<”将会匹配“<B>”的第一个字符“<”。然后[A-Z]匹配 B，[A-Z0-9]*将会匹配 0 到多次字母数字，后面紧接着 0 到多个非“>”的字符。最后正则表达式的“>”将会匹配“<B>”的“>”。接下来正则引擎将对结束标签之前的字符进行惰性匹配，直到遇到一个“</”符号。然后正则表达式中的“\1”表示对前面匹配的组“([A-Z][A-Z0-9]*)”进行引用，在本例中，被引用的是标签名“B”。所以需要被匹配的结尾标签为“</B>”

你可以对相同的后向引用组进行多次引用，<<([a-c])x\1x\1>>将匹配“axaxa”、“bxbxb”以及“cxcxc”。如果用数字形式引用的组没有有效的匹配，则引用到的内容简单的为空。

一个后向引用不能用于它自身。<<([abc]\1)>>是错误的。因此你不能将<<\0>>用于一个正则表达式匹配本身，它只能用于替换操作中。

后向引用不能用于字符集内部。<<(a)[\1b]>>中的<<\1>>并不表示后向引用。在字符集内部，<<\1>>可以被解释为八进制形式的转码。

向后引用会降低引擎的速度，因为它需要存储匹配的组。如果你不需要向后引用，你可以告诉引擎对某个组不存储。例如：<<Get(?:Value)>>。其中“(”后面紧跟的“?:”会告诉引擎对于组(Value)，不存储匹配的值以供后向引用。

- 重复操作与后向引用

当对组使用重复操作符时，缓存里后向引用内容会被不断刷新，只保留最后匹配的内容。例如：<<([abc]+)=\1>>将匹配“cab=cab”，但是<<([abc])+=\1>>却不会。因为([abc])第一次匹配“c”时，“\1”代表“c”；然后([abc])会继续匹配“a”和“b”。最后“\1”代表“b”，所以它会匹配“cab=b”。

应用：检查重复单词--当编辑文字时，很容易就会输入重复单词，例如“the the”。使用<<\b(\w+)\s+\1\b>>可以检测到这些重复单词。要删除第二个单词，只要简单的利用替换功能替换掉“\1”就可以了。

- 组的命名和引用

在 PHP，Python 中，可以用<<(?P<name>group)>>来对组进行命名。在本例中，词法?P<name>就是对组(group)进行了命名。其中 name 是你对组的起的名字。你可以用(?P=name)进行引用。

.NET 的命名组

.NET framework 也支持命名组。不幸的是，微软的程序员们决定发明他们自己的语法，而不是沿用 Perl、Python 的规则。目前为止，还没有任何其他的正则表达式实现支持微软发明的语法。

下面是.NET 中的例子：

(?<first>group)(?’second’group)

正如你所看到的，.NET 提供两种词法来创建命名组：一是用尖括号“<>”，或者用单引号“’’”。尖括号在字符串中使用更方便，单引号在 ASP 代码中更有用，因为 ASP 代码中“<>”被用作 HTML 标签。

要引用一个命名组，使用\k<name>或\k’name’.

当进行搜索替换时，你可以用“${name}”来引用一个命名组。

12. 正则表达式的匹配模式

本教程所讨论的正则表达式引擎都支持三种匹配模式：

<</i>>使正则表达式对大小写不敏感，

<</s>>开启“单行模式”，即点号“.”匹配新行符

<</m>>开启“多行模式”，即“^”和“$”匹配新行符的前面和后面的位置。

- 在正则表达式内部打开或关闭模式

如果你在正则表达式内部插入修饰符(?ism)，则该修饰符只对其右边的正则表达式起作用。(?-i)是关闭大小写不敏感。你可以很快的进行测试。<<(?i)te(?-i)st>>应该匹配 TEst，但是不能匹配 teST 或 TEST.

13. 原子组与防止回溯

在一些特殊情况下，因为回溯会使得引擎的效率极其低下。

让我们看一个例子：要匹配这样的字串，字串中的每个字段间用逗号做分隔符，第 12 个字段由 P 开头。

我们容易想到这样的正则表达式<<^(.*?,){11}P>>。这个正则表达式在正常情况下工作的很好。但是在极端情况下，如果第 12 个字段不是由 P 开头，则会发生灾难性的回溯。如要搜索的字串为“1,2,3,4,5,6,7,8,9,10,11,12,13”。首先，正则表达式一直成功匹配直到第 12 个字符。这时，前面的正则表达式消耗的字串为“1,2,3,4,5,6,7,8,9,10,11,”，到了下一个字符，<<P>>并不匹配“12”。所以引擎进行回溯，这时正则表达式消耗的字串为“1,2,3,4,5,6,7,8,9,10,11”。继续下一次匹配过程，下一个正则符号为点号<<.>>，可以匹配下一个逗号“,”。然而<<，>>并不匹配字符“12”中的“1”。匹配失败，继续回溯。大家可以想象，这样的回溯组合是个非常大的数量。因此可能会造成引擎崩溃。

用于阻止这样巨大的回溯有几种方案：

一种简单的方案是尽可能的使匹配精确。用取反字符集代替点号。例如我们用如下正则表达式<<^([^,\r\n]*,){11}P>>，这样可以使失败回溯的次数下降到 11 次。

另一种方案是使用原子组。

原子组的目的是使正则引擎失败的更快一点。因此可以有效的阻止海量回溯。原子组的语法是<<(?>正则表达式)>>。位于(?>)之间的所有正则表达式都会被认为是一个单一的正则符号。一旦匹配失败，引擎将会回溯到原子组前面的正则表达式部分。前面的例子用原子组可以表达成<<^(?>(.*?,){11})P>>。一旦第十二个字段匹配失败，引擎回溯到原子组前面的<<^>>。

14. 向前查看与向后查看

Perl 5 引入了两个强大的正则语法：“向前查看”和“向后查看”。他们也被称作“零长度断言”。他们和锚定一样都是零长度的（所谓零长度即指该正则表达式不消耗被匹配的字符串）。不同之处在于“前后查看”会实际匹配字符，只是他们会抛弃匹配只返回匹配结果：匹配或不匹配。这就是为什么他们被称作“断言”。他们并不实际消耗字符串中的字符，而只是断言一个匹配是否可能。

几乎本文讨论的所有正则表达式的实现都支持“向前向后查看”。唯一的一个例外是 Javascript 只支持向前查看。

- 肯定和否定式的向前查看

如我们前面提过的一个例子：要查找一个 q，后面没有紧跟一个 u。也就是说，要么 q 后面没有字符，要么后面的字符不是 u。采用否定式向前查看后的一个解决方案为<<q(?!u)>>。否定式向前查看的语法是<<(?!查看的内容)>>。

肯定式向前查看和否定式向前查看很类似：<<(?=查看的内容)>>。

如果在“查看的内容”部分有组，也会产生一个向后引用。但是向前查看本身并不会产生向后引用，也不会被计入向后引用的编号中。这是因为向前查看本身是会被抛弃掉的，只保留匹配与否的判断结果。如果你想保留匹配的结果作为向后引用，你可以用<<(?=(regex))>>来产生一个向后引用。

- 肯定和否定式的先后查看

向后查看和向前查看有相同的效果，只是方向相反

否定式向后查看的语法是：<<(?<!查看内容)>>

肯定式向后查看的语法是：<<(?<=查看内容)>>

我们可以看到，和向前查看相比，多了一个表示方向的左尖括号。

例：<<(?<!a)b>>将会匹配一个没有“a”作前导字符的“b”。

值得注意的是：向前查看从当前字符串位置开始对“查看”正则表达式进行匹配；向后查看则从当前字符串位置开始先后回溯一个字符，然后再开始对“查看”正则表达式进行匹配。

- 深入正则表达式引擎内部

让我们看一个简单例子。

把正则表达式<<q(?!u)>>应用到字符串“Iraq”。正则表达式的第一个符号是<<q>>。正如我们知道的，引擎在匹配<<q>>以前会扫过整个字符串。当第四个字符“q”被匹配后，“q”后面是空字符(void)。而下一个正则符号是向前查看。引擎注意到已经进入了一个向前查看正则表达式部分。下一个正则符号是<<u>>，和空字符不匹配，从而导致向前查看里的正则表达式匹配失败。因为是一个否定式的向前查看，意味着整个向前查看结果是成功的。于是匹配结果“q”被返回了。

我们在把相同的正则表达式应用到“quit”。<<q>>匹配了“q”。下一个正则符号是向前查看部分的<<u>>，它匹配了字符串中的第二个字符“i”。引擎继续走到下个字符“i”。然而引擎这时注意到向前查看部分已经处理完了，并且向前查看已经成功。于是引擎抛弃被匹配的字符串部分，这将导致引擎回退到字符“u”。

因为向前查看是否定式的，意味着查看部分的成功匹配导致了整个向前查看的失败，因此引擎不得不进行回溯。最后因为再没有其他的“q”和<<q>>匹配，所以整个匹配失败了。

为了确保你能清楚地理解向前查看的实现，让我们把<<q(?=u)i>>应用到“quit”。<<q>>首先匹配“q”。然后向前查看成功匹配“u”，匹配的部分被抛弃，只返回可以匹配的判断结果。引擎从字符“i”回退到“u”。由于向前查看成功了，引擎继续处理下一个正则符号<<i>>。结果发现<<i>>和“u”不匹配。因此匹配失败了。由于后面没有其他的“q”，整个正则表达式的匹配失败了。

- 更进一步理解正则表达式引擎内部机制

让我们把<<(?<=a)b>>应用到“thingamabob”。引擎开始处理向后查看部分的正则符号和字符串中的第一个字符。在这个例子中，向后查看告诉正则表达式引擎回退一个字符，然后查看是否有一个“a”被匹配。因为在“t”前面没有字符，所以引擎不能回退。因此向后查看失败了。引擎继续走到下一个字符“h”。再一次，引擎暂时回退一个字符并检查是否有个“a”被匹配。结果发现了一个“t”。向后查看又失败了。

向后查看继续失败，直到正则表达式到达了字符串中的“m”，于是肯定式的向后查看被匹配了。因为它是零长度的，字符串的当前位置仍然是“m”。下一个正则符号是<<b>>，和“m”匹配失败。下一个字符是字符串中的第二个“a”。引擎向后暂时回退一个字符，并且发现<<a>>不匹配“m”。

在下一个字符是字符串中的第一个“b”。引擎暂时性的向后退一个字符发现向后查看被满足了，同时<<b>>匹配了“b”。因此整个正则表达式被匹配了。作为结果，正则表达式返回字符串中的第一个“b”。

- 向前向后查看的应用

我们来看这样一个例子：查找一个具有 6 位字符的，含有“cat”的单词。

首先，我们可以不用向前向后查看来解决问题，例如：

<< cat\w{3}|\wcat\w{2}|\w{2}cat\w|\w{3}cat>>

足够简单吧！但是当需求变成查找一个具有 6-12 位字符，含有“cat”，“dog”或“mouse”的单词时，这种方法就变得有些笨拙了。

我们来看看使用向前查看的方案。在这个例子中，我们有两个基本需求要满足：一是我们需要一个 6 位的字符，二是单词含有“cat”。

满足第一个需求的正则表达式为<<\b\w{6}\b>>。满足第二个需求的正则表达式为<<\b\w*cat\w*\b>>。

把两者结合起来，我们可以得到如下的正则表达式：

<<(?=\b\w{6}\b)\b\w*cat\w*\b>>

具体的匹配过程留给读者。但是要注意的一点是，向前查看是不消耗字符的，因此当判断单词满足具有 6 个字符的条件后，引擎会从开始判断前的位置继续对后面的正则表达式进行匹配。

最后作些优化，可以得到下面的正则表达式：

<<\b(?=\w{6}\b)\w{0,3}cat\w*>>

15. 正则表达式中的条件测试

条件测试的语法为<<(?ifthen|else)>>。“if”部分可以是向前向后查看表达式。如果用向前查看，则语法变为：<<(?(?=regex)then|else)>>，其中 else 部分是可选的。

如果 if 部分为 true，则正则引擎会试图匹配 then 部分，否则引擎会试图匹配 else 部分。

需要记住的是，向前先后查看并不实际消耗任何字符，因此后面的 then 与 else 部分的匹配时从 if 测试前的部分开始进行尝试。

16. 为正则表达式添加注释

在正则表达式中添加注释的语法是：<<(?#comment)>>

例：为用于匹配有效日期的正则表达式添加注释：

(?#year)(19|20)\d\d[- /.](?#month)(0[1-9]|1[012])[-/.](?#day)(0[1-9]|[12][0-9]|3[01])

字符	描述
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个向后引用、或一个八进制转义符。例如，“n”匹配字符“n”。“\n”匹配一个换行符。序列“\\”匹配“\”而“\(”则匹配“(”。
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^也匹配“\n”或“\r”之后的位置。

$	匹配输入字符串的结束位置。如果设置了 RegExp 对象的 Multiline 属性，$也匹配“\n”或“\r”之前的位置。
*****	匹配前面的子表达式零次或多次。例如，zo*能匹配“z”以及“zoo”。*等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。
?	匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“does”或“does”中的“do”。?等价于{0,1}。
{*n*}	*n* 是一个非负整数。匹配确定的 *n* 次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个 o。
{*n*,}	*n* 是一个非负整数。至少匹配 *n* 次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有 o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o*”。
{*n*,*m*}	*m* 和 *n* 均为非负整数，其中 *n*<=*m*。最少匹配 *n* 次且最多匹配 *m* 次。例如，“o{1,3}”将匹配“fooooood”中的前三个 o。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。
?	当该字符紧跟在任何一个其他限制符（*,+,?，{*n*}，{*n*,}，{*n*,*m*}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+?”将匹配单个“o”，而“o+”将匹配所有“o”。
.	匹配除“*n*”之外的任何单个字符。要匹配包括“*n*”在内的任何字符，请使用像“(.|\n)”的模式。
(pattern)	匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在 VBScript 中使用 SubMatches 集合，在 JScript 中则使

	用$0…$9 属性。要匹配圆括号字符，请使用“\(”或“\)”。
(?:pattern)	匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用或字符“(|)”来组合一个模式的各个部分是很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。
(?=pattern)	正向肯定预查，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?!pattern)	正向否定预查，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始
(?<=pattern)	反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。
(?<!pattern)	反向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。
x|y	匹配 x 或 y。例如，“z|food”能匹配“z”或“food”。“(z|f)ood”则匹配“zood”或“food”。
[xyz]	字符集合。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。

[^xyz]	负值字符集合。匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“p”。
[a-z]	字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。
\B	匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。
\cx	匹配由 x 指明的控制字符。例如，\cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的“c”字符。
\d	匹配一个数字字符。等价于[0-9]。
\D	匹配一个非数字字符。等价于[^0-9]。
\f	匹配一个换页符。等价于\x0c 和\cL。
\n	匹配一个换行符。等价于\x0a 和\cJ。
\r	匹配一个回车符。等价于\x0d 和\cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于[\f\n\r\t\v]。

\S	匹配任何非空白字符。等价于[^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于\x09 和\cI。
\v	匹配一个垂直制表符。等价于\x0b 和\cK。
\w	匹配包括下划线的任何单词字符。等价于“`[A-Za-z0-9_]`”。
\W	匹配任何非单词字符。等价于“`[^A-Za-z0-9_]`”。
\x*n*	匹配 *n*，其中 *n* 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，“`\x41`”匹配“`A`”。“`\x041`”则等价于“`\x04&1`”。正则表达式中可以使用 ASCII 编码。.
num	匹配 *num*，其中 *num* 是一个正整数。对所获取的匹配的引用。例如，“`(.)\1`”匹配两个连续的相同字符。
n	标识一个八进制转义值或一个向后引用。如果*n* 之前至少 *n* 个获取的子表达式，则 *n* 为向后引用。否则，如果 *n* 为八进制数字（0-7），则 *n* 为一个八进制转义值。
nm	标识一个八进制转义值或一个向后引用。如果*nm* 之前至少有 *nm* 个获得子表达式，则 *nm* 为向后引用。如果*nm* 之前至少有 *n* 个获取，则 *n* 为一个后跟文字 *m* 的向后引用。如果前面的条件都不满足，若 *n* 和 *m* 均为八进制数字（0-7），则*nm* 将匹配八进制转义值 *nm*。
nml	如果 *n* 为八进制数字（0-3），且 *m 和 l* 均为八进制数字（0-7），则匹配八进制转义值 *nm*l。
\u*n*	匹配 *n*，其中 *n* 是一个用四个十六进制数字表示的 Unicode 字符。例如，\u00A9 匹配版权符号（©）。