**正则表达式**，又称**正规表示法**、**常规表示法**（英语：Regular Expression，在代码中常简写为regex、regexp或RE），计算机科学的一个概念。正则表达式使用单个[字符](http://baike.baidu.com/view/263416.htm)串来描述、匹配一系列符合某个句法规则的[字符串](http://baike.baidu.com/view/56072.htm)。在很多[文本编辑器](http://baike.baidu.com/view/487023.htm)里，正则表达式通常被用来检索、替换那些符合某个模式的文本。

许多程序设计语言都支持利用正则表达式进行字符串操作。例如，在[Perl](http://baike.baidu.com/view/46614.htm)中就内建了一个功能强大的正则表达式引擎。正则表达式这个概念最初是由[Unix](http://baike.baidu.com/view/8095.htm)中的工具软件（例如sed和[grep](http://baike.baidu.com/view/1057278.htm)）普及开的。正则表达式通常缩写成“regex”，[单数](http://baike.baidu.com/view/1525497.htm)有regexp、regex，[复数](http://baike.baidu.com/view/10078.htm)有regexps、regexes、regexen。

起源

[编辑](javascript:;)

正则表达式的“鼻祖”或许可一直追溯到科学家对人类神经系统工作原理的早期研究。美国新泽西州的Warren McCulloch和出生在美国底特律的Walter Pitts这两位神经生理方面的科学家，研究出了一种用数学方式来描述神经网络的新方法，他们创造性地将神经系统中的神经元描述成了小而简单的自动控制元，从而作出了一项伟大的工作革新。

在1956 年,一位名叫Stephen Kleene的数学科学家，他在Warren McCulloch和Walter Pitts早期工作的基础之上，发表了一篇题目是《神经网事件的表示法》的论文，利用称之为正则集合的数学符号来描述此模型，引入了正则表达式的概念。正则表达式被作为用来描述其称之为“正则集的代数”的一种表达式，因而采用了“正则表达式”这个术语。

之后一段时间，人们发现可以将这一工作成果应用于其他方面。Ken Thompson就把这一成果应用于计算[搜索算法](http://baike.baidu.com/view/3688332.htm)的一些早期研究，Ken Thompson是 Unix的主要发明人，也就是大名鼎鼎的Unix之父。Unix之父将此[符号系统](http://baike.baidu.com/view/6552866.htm)引入编辑器QED，然后是Unix上的编辑器ed，并最终引入grep。Jeffrey Friedl 在其著作《Mastering Regular Expressions (2nd edition)》（中文版译作：精通正则表达式，已出到第三版）中对此作了进一步阐述讲解，如果你希望更多了解正则表达式理论和历史，推荐你看看这本书。

自此以后，正则表达式被广泛地应用到各种UNIX或类似于UNIX的工具中，如大家熟知的[Perl](http://baike.baidu.com/view/46614.htm)。Perl的正则表达式源自于Henry Spencer编写的regex，之后已演化成了pcre（Perl兼容正则表达式Perl Compatible Regular Expressions），pcre是一个由Philip Hazel开发的、为很多现代工具所使用的库。正则表达式的第一个实用应用程序即为Unix中的 qed 编辑器。

然后，正则表达式在各种计算机语言或各种应用领域得到了广大的应用和发展，演变成为计算机技术森林中的一只形神美丽且声音动听的百灵鸟。

以上是关于正则表达式的起源和发展的历史描述，如今正则表达式在基于文本的[编辑器](http://baike.baidu.com/view/499644.htm)和搜索工具中依然占据着一个非常重要的地位。

在最近的六十年中，正则表达式逐渐从模糊而深奥的数学概念，发展成为在计算机各类工具和软件包应用中的主要功能。不仅仅众多UNIX工具支持正则表达式，近二十年来，在WINDOWS的阵营下，正则表达式的思想和应用在大部分 Windows 开发者工具包中得到支持和嵌入应用！从正则式在Microsoft Visual Basic 6 或 Microsoft VBScript到.NET Framework中的探索和发展，WINDOWS系列产品对正则表达式的支持发展到无与伦比的高度，几乎所有 Microsoft 开发者和所有.NET语言都可以使用正则表达式。如果你是一位接触计算机语言的工作者，那么你会在主流操作系统（\*nix[Linux, Unix等]、Windows、HP、BeOS等）、主流的开发语言（delphi、[Scala](http://baike.baidu.com/view/1588150.htm)、[PHP](http://baike.baidu.com/view/99.htm)、[C#](http://baike.baidu.com/view/6590.htm)、Java、[C++](http://baike.baidu.com/view/824.htm)、Objective-c、Swift、VB、[Javascript](http://baike.baidu.com/view/16168.htm)、[Ruby](http://baike.baidu.com/view/45135.htm)以及[Python](http://baike.baidu.com/view/21087.htm)等）、数以亿万计的各种应用软件中，都可以看到正则表达式优美的舞姿。

概念

[编辑](javascript:;)

**正则表达式是对**[**字符**](http://baike.baidu.com/view/263416.htm)**串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。**

给定一个正则表达式和另一个字符串，我们可以达到如下的目的：

1. 给定的字符串是否符合正则表达式的过滤逻辑（称作“匹配”）；

2. 可以通过正则表达式，从字符串中获取我们想要的特定部分。

正则表达式的特点是：

1. 灵活性、逻辑性和功能性非常的强；

2. 可以迅速地用极简单的方式达到字符串的复杂控制。

3. 对于刚接触的人来说，比较晦涩难懂。

由于正则表达式主要[应用对象](http://baike.baidu.com/view/5314991.htm)是文本，因此它在各种[文本编辑器](http://baike.baidu.com/view/487023.htm)场合都有应用，小到著名编辑器EditPlus，大到Microsoft Word、Visual Studio等大型编辑器，都可以使用正则表达式来处理文本内容。

引擎

[编辑](javascript:;)

正则引擎主要可以分为两大类：一种是**DFA**，一种是**NFA**。这两种引擎都有了很久的历史(至今二十多年)，当中也由这两种引擎产生了很多变体！于是POSIX的出台规避了不必要变体的继续产生。这样一来，主流的正则引擎又分为3类：一、DFA，二、传统型NFA，三、**POSIX NFA**。

[DFA](http://baike.baidu.com/view/289418.htm) 引擎在线性时状态下执行，因为它们不要求回溯（并因此它们永远不测试相同的字符两次）。DFA 引擎还可以确保匹配最长的可能的字符串。但是，因为 DFA 引擎只包含有限的状态，所以它不能匹配具有反向引用的模式；并且因为它不构造显示扩展，所以它不可以捕获子表达式。

传统的 NFA 引擎运行所谓的“贪婪的”匹配[回溯算法](http://baike.baidu.com/view/6056523.htm)，以指定顺序测试正则表达式的所有可能的扩展并接受第一个匹配项。因为传统的 NFA 构造正则表达式的特定扩展以获得成功的匹配，所以它可以捕获子表达式匹配和匹配的反向引用。但是，因为传统的 NFA 回溯，所以它可以访问完全相同的状态多次（如果通过不同的路径到达该状态）。因此，在最坏情况下，它的执行速度可能非常慢。因为传统的 NFA 接受它找到的第一个匹配，所以它还可能会导致其他（可能更长）匹配未被发现。

POSIX NFA 引擎与传统的 NFA 引擎类似，不同的一点在于：在它们可以确保已找到了可能的最长的匹配之前，它们将继续回溯。因此，POSIX NFA 引擎的速度慢于传统的 NFA 引擎；并且在使用 POSIX NFA 时，您恐怕不会愿意在更改回溯搜索的顺序的情况下来支持较短的匹配搜索，而非较长的匹配搜索。

使用DFA引擎的程序主要有：awk,egrep,flex,lex,MySQL,Procmail等；

使用传统型NFA引擎的程序主要有：GNU Emacs,Java,ergp,less,more,.NET语言,PCRE library,Perl,PHP,Python,Ruby,sed,vi；

使用POSIX NFA引擎的程序主要有：mawk,Mortice Kern Systems’ utilities,GNU Emacs(使用时可以明确指定)；

也有使用DFA/NFA混合的引擎：GNU awk,GNU grep/egrep,Tcl。

举例简单说明NFA与DFA工作的区别：

比如有字符串this is yansen’s blog，正则表达式为 /ya(msen|nsen|nsem)/ (不要在乎表达式怎么样，这里只是为了说明引擎间的工作区别)。 NFA工作方式如下，先在字符串中查找 y 然后匹配其后是否为 a ，如果是 a 则继续，查找其后是否为 m 如果不是则匹配其后是否为 n (此时淘汰msen选择支)。然后继续看其后是否依次为 s,e，接着测试是否为 n ，是 n 则匹配成功，不是则测试是否为 m 。为什么是 m ？因为 NFA 工作方式是以正则表达式为标准，反复测试字符串，这样同样一个字符串有可能被反复测试了很多次！

而DFA则不是如此，DFA会从 this 中 t 开始依次查找 y，定位到 y ，已知其后为 a ，则查看表达式是否有 a ，此处正好有 a 。然后字符串 a 后为 n ，DFA依次测试表达式，此时 msen 不符合要求淘汰。nsen 和 nsem 符合要求，然后DFA依次检查字符串，检测到sen 中的 n 时只有nsen 分支符合，则匹配成功！

由此可以看出来，两种引擎的工作方式完全不同，一个(NFA)以表达式为主导，一个(DFA)以文本为主导！一般而论，DFA引擎则搜索更快一些！但是NFA以表达式为主导，反而更容易操纵，因此一般程序员更偏爱NFA引擎！ 两种引擎各有所长，而真正的引用则取决与你的需要以及所使用的语言！

符号

[编辑](javascript:;)

（摘自《[正则表达式之道](http://baike.baidu.com/view/1427403.htm)》）

正则表达式[1]  由一些普通[字符](http://baike.baidu.com/view/263416.htm)和一些[元字符](http://baike.baidu.com/view/1061241.htm)（metacharacters）组成。**普通字符包括大小写的字母和数字**，**而元字符则具有特殊的含义**，我们下面会给予解释。

在最简单的情况下，一个正则表达式看上去就是一个普通的查找串。例如，正则表达式"testing"中没有包含任何元字符，它可以匹配"testing"和"testing123"等字符串，但是不能匹配"Testing"。

要想真正的用好正则表达式，正确的理解元字符是最重要的事情。下表列出了所有的元字符和对它们的一个简短的描述。

|  |  |
| --- | --- |
| 元字符 | 描述 |
| \ | 将下一个字符标记符、或一个向后引用、或一个八进制转义符。例如，“\\n”匹配\n。“\n”匹配换行符。序列“\\”匹配“\”而“\(”则匹配“(”。即相当于多种编程语言中都有的“转义字符”的概念。 |
| ^ | 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任意次。例如，zo\*能匹配“z”，“zo”以及“zoo”，但是不匹配“bo”。\*等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次(大于等于1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“do”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”。 |
| .点 | 匹配除“\r\n”之外的任何单个字符。要匹配包括“\r\n”在内的任何字符，请使用像“[\s\S]”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) | 非获取匹配，匹配pattern但不获取匹配结果，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) | 非获取匹配，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，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) | 非获取匹配，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，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”。 |
| (?<=pattern) | 非获取匹配，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) | 非获取匹配，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”。这个地方不正确，有问题 |
| x|y | 匹配x或y。例如，“z|food”能匹配“z”或“food”(此处请谨慎)。“(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“plin”。 |
| [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。  注意:只有连字符在字符组内部时,并且出现在两个字符之间时,才能表示字符的范围; 如果出字符组的开头,则只能表示连字符本身. |
| [^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（即正则表达式的“匹配”有两种概念，一种是匹配字符，一种是匹配位置，这里的\b就是匹配位置的）。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 | 匹配任何不可见字符，包括空格、制表符、换页符等等。等价于[ \f\n\r\t\v]。 |
| \S | 匹配任何可见字符。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。类似但不等价于“[A-Za-z0-9\_]”，这里的"单词"字符使用Unicode字符集。 |
| \W | 匹配任何非单词字符。等价于“[^A-Za-z0-9\_]”。 |
| \xn | 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（0-7），且m和l均为八进制数字（0-7），则匹配八进制转义值nml。 |
| \un | 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（&copy;）。 |
| \p{P} | 小写 p 是 property 的意思，表示 Unicode 属性，用于 Unicode 正表达式的前缀。中括号内的“P”表示Unicode 字符集七个字符属性之一：标点字符。  其他六个属性：  L：字母；  M：标记符号（一般不会单独出现）；  Z：分隔符（比如空格、换行等）；  S：符号（比如数学符号、货币符号等）；  N：数字（比如阿拉伯数字、罗马数字等）；  C：其他字符。  *\*注：此语法部分语言不支持，例：javascript。* |
| < > | 匹配词（word）的开始（<）和结束（>）。例如正则表达式<the>能够匹配字符串"for the wise"中的"the"，但是不能匹配字符串"otherwise"中的"the"。注意：这个元字符不是所有的软件都支持的。 |
| ( ) | 将( 和 ) 之间的表达式定义为“组”（group），并且将匹配这个表达式的字符保存到一个临时区域（一个正则表达式中最多可以保存9个），它们可以用 \1 到\9 的符号来引用。 |
| | | 将两个匹配条件进行逻辑“或”（Or）运算。例如正则表达式(him|her) 匹配"it belongs to him"和"it belongs to her"，但是不能匹配"it belongs to them."。注意：这个元字符不是所有的软件都支持的。 |
| + | 匹配1或多个正好在它之前的那个字符。例如正则表达式9+匹配9、99、999等。注意：这个元字符不是所有的软件都支持的。 |
| ? | 匹配0或1个正好在它之前的那个字符。注意：这个元字符不是所有的软件都支持的。 |
| {i} {i,j} | 匹配指定数目的字符，这些字符是在它之前的表达式定义的。例如正则表达式A[0-9]{3} 能够匹配字符"A"后面跟着正好3个数字字符的串，例如A123、A348等，但是不匹配A1234。而正则表达式[0-9]{4,6} 匹配连续的任意4个、5个或者6个数字 |

最简单的[元字符](http://baike.baidu.com/view/1061241.htm)是点，它能够匹配任何单个字符（注意不包括换行符）

**等价：**

等价是等同于的意思，表示同样的功能，用不同符号来书写。

?,\*,+,\d,\w 都是等价字符  
　　?等价于匹配长度{0,1}  
　　\*等价于匹配长度{0,}   
　　+等价于匹配长度{1,}  
　　\d等价于[0-9]  
　　\W等价于[A-Za-z\_0-9]。

**常用运算符与表达式：**^ 开始  
　　（） 域段  
　　[] 包含,默认是一个字符长度  
　　[^] 不包含,默认是一个字符长度  
　　{n,m} 匹配长度   
　　. 任何单个字符(\. 字符点)  
　　| 或  
　　\ 转义  
　　$ 结尾  
　　[A-Z] 26个大写字母  
　　[a-z] 26个小写字母  
　　[0-9] 0至9数字

[A-Za-z0-9] 26个大写字母、26个小写字母和0至9数字  
　　， 分割  
　　.  
　　  
　　**分割语法：**  
　　[A,H,T,W] 包含A或H或T或W字母  
　　[a,h,t,w] 包含a或h或t或w字母  
　　[0,3,6,8] 包含0或3或6或8数字

　　**语法与释义：**基础语法 "^([]{})([]{})([]{})$"  
　　正则字符串 = "开始（[包含内容]{长度}）（[包含内容]{长度}）（[包含内容]{长度}）结束"   
　　  
　　?,\*,+,\d,\w 这些都是简写的,完全可以用[]和{}代替，在(?:)(?=)(?!)(?<=)(?<!)(?i)(\*?)(+?)这种特殊组合情况下除外。  
　　初学者可以忽略?,\*,+,\d,\w一些简写标示符，学会了基础使用再按表自己去等价替换

1.验证用户名和密码：（"^[a-zA-Z]\w{5,15}$"）正确格式："[A-Z][a-z]\_[0-9]"组成,并且第一个字必须为字母6~16位；

2.验证电话号码：（"^(\\d{3,4}-)\\d{7,8}$"）正确格式：xxx/xxxx-xxxxxxx/xxxxxxxx；

3.验证手机号码："^1[3|4|5|7|8][0-9]\\d{8}$"；

4.验证身份证号（15位或18位数字）："\\d{14}[[0-9],0-9xX]"；

5.验证Email地址：("^\w+([-+.]\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*$")；

6.只能输入由数字和26个英文字母组成的字符串：("^[A-Za-z0-9]+$") ;

7.整数或者小数：^[0-9]+([.][0-9]+){0,1}$

8.只能输入数字："^[0-9]\*$"。

9.只能输入n位的数字："^\d{n}$"。

10.只能输入至少n位的数字："^\d{n,}$"。

11.只能输入m~n位的数字："^\d{m,n}$"。

12.只能输入零和非零开头的数字："^(0|[1-9][0-9]\*)$"。

13.只能输入有两位小数的正实数："^[0-9]+(.[0-9]{2})?$"。

14.只能输入有1~3位小数的正实数："^[0-9]+(\.[0-9]{1,3})?$"。

15.只能输入非零的正整数："^\+?[1-9][0-9]\*$"。

16.只能输入非零的负整数："^\-[1-9][0-9]\*$"。

17.只能输入长度为3的字符："^.{3}$"。

18.只能输入由26个英文字母组成的字符串："^[A-Za-z]+$"。

19.只能输入由26个大写英文字母组成的字符串："^[A-Z]+$"。

20.只能输入由26个小写英文字母组成的字符串："^[a-z]+$"。

21.验证是否含有^%&',;=?$\"等字符："[%&',;=?$\\^]+"。

22.只能输入汉字："^[\u4e00-\u9fa5]{0,}$"。

23.验证URL："^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]\*)?$"。

24.验证一年的12个月："^(0?[1-9]|1[0-2])$"正确格式为："01"～"09"和"10"～"12"。

25.验证一个月的31天："^((0?[1-9])|((1|2)[0-9])|30|31)$"正确格式为；"01"～"09"、"10"～"29"和“30”~“31”。

26.获取日期正则表达式：\\d{4}[年|\-|\.]\d{\1-\12}[月|\-|\.]\d{\1-\31}日?

评注：可用来匹配大多数年月日信息。

27.匹配双[字节](http://baike.baidu.com/view/60408.htm)[字符](http://baike.baidu.com/view/263416.htm)(包括汉字在内)：[^\x00-\xff]

评注：可以用来计算字符串的长度（一个双字节字符长度计2，ASCII字符计1）

28.匹配空白行的正则表达式：\n\s\*\r

评注：可以用来删除空白行

29.匹配HTML标记的正则表达式：<(\S\*?)[^>]\*>.\*?</>|<.\*? />

评注：网上流传的版本太糟糕，上面这个也仅仅能匹配部分，对于复杂的嵌套标记依旧无能为力

30.匹配首尾空白[字符](http://baike.baidu.com/view/263416.htm)的正则表达式：^\s\*|\s\*$

评注：可以用来删除行首行尾的空白字符(包括空格、[制表符](http://baike.baidu.com/view/1138182.htm)、换页符等等)，非常有用的表达式

31.匹配网址URL的正则表达式：[a-zA-z]+://[^\s]\*

评注：网上流传的版本功能很有限，上面这个基本可以满足需求

32.匹配帐号是否合法(字母开头，允许5-16[字节](http://baike.baidu.com/view/60408.htm)，允许字母数字下划线)：^[a-zA-Z][a-zA-Z0-9\_]{4,15}$

评注：表单验证时很实用

33.匹配腾讯QQ号：[1-9][0-9]{4,}

评注：腾讯QQ号从10 000 开始

34.匹配中国邮政编码：[1-9]\\d{5}(?!\d)

评注：中国邮政编码为6位数字

35.匹配ip地址：([1-9]{1,3}\.){3}[1-9]。

评注：提取ip地址时有用

36.匹配MAC地址：([A-Fa-f0-9]{2}\:){5}[A-Fa-f0-9]

零宽断言

[编辑](javascript:;)

用于查找在某些内容(但并不包括这些内容)之前或之后的东西，也就是说它们像\b,^,$那样用于指定一个位置，这个位置应该满足一定的条件(即断言)，因此它们也被称为零宽断言。最好还是拿例子来说明吧：

(?=exp)也叫零宽度正预测先行断言[3]  ，它断言自身出现的位置的后面能匹配表达式exp。比如\b\w+(?=ing\b)，匹配以ing结尾的单词的前面部分(除了ing以外的部分)，如查找I'm singing while you're dancing.时，它会匹配sing和danc。

(?<=exp)也叫零宽度正回顾后发断言[3]  ，它断言自身出现的位置的前面能匹配表达式exp。比如(?<=\bre)\w+\b会匹配以re开头的单词的后半部分(除了re以外的部分)，例如在查找reading a book时，它匹配ading。

假如你想要给一个很长的数字中每三位间加一个逗号(当然是从右边加起了)，你可以这样查找需要在前面和里面添加逗号的部分：((?<=\D)\D{3})+\b，用它对xxxxxxxxxx进行查找时结果是xxxxxxxxx

下面这个例子同时使用了这两种断言：(?<=\s)\d+(?=\s)匹配以空白符间隔的数字(再次强调，不包括这些空白符)

断言用来声明一个应该为真的事实。正则表达式中只有当断言为真时才会继续进行匹配。

负向零宽

[编辑](javascript:;)

如果我们只是想要**确保某个字符没有出现，但并不想去匹配它**时怎么办？例如，如果我们想查找这样的单词--它里面出现了字母q，但是q后面跟的不是字母u,我们可以尝试这样：

/\b\w\*q[^u]\w\*\b匹配包含**后面不是字母u的字母q**的单词。但是如果多做测试(或者你思维足够敏锐，直接就观察出来了)，你会发现，如果q出现在单词的结尾的话，像**Iraq**,**Benq**，这个表达式就会出错。这是因为[^u]总要匹配一个字符，所以如果q是单词的最后一个字符的话，后面的[^u]将会匹配q后面的单词分隔符(可能是空格，或者是句号或其它的什么)，后面的\w\*\b将会匹配下一个单词，于是\b\w\*q[^u]\w\*\b就能匹配整个Iraq fighting。负向零宽断言能解决这样的问题，因为它只匹配一个位置，并不**消费**任何字符。，我们可以这样来解决这个问题：\b\w\*q(?!u)\w\*\b。

零宽度负预测先行断言(?!exp)，断言此位置的后面不能匹配表达式exp。例如：\d{3}(?!\d)匹配三位数字，而且这三位数字的后面不能是数字；\b((?!abc)\w)+\b匹配不包含连续字符串abc的单词。

同理，我们可以用(?<!exp),零宽度负回顾后发断言来断言此位置的前面不能匹配表达式exp：(?<![a-z])\d{7}匹配前面不是小写字母的七位数字。

请详细分析表达式(?<=<(\w+)>).\*(?=<\/\1>)，这个表达式最能表现零宽断言的真正用途。

一个更复杂的例子：(?<=<(\w+)>).\*(?=<\/\1>)匹配不包含属性的简单HTML标签内里的内容。(?<=<(\w+)>)指定了这样的前缀：被尖括号括起来的单词(比如可能是<b>)，然后是.\*(任意的字符串),最后是一个后缀(?=<\/\1>)。注意后缀里的\/，它用到了前面提过的字符转义，将”/“转义；\1则是一个反向引用，引用的正是捕获的第一组，前面的(\w+)匹配的内容，这样如果前缀实际上是<b>的话，后缀就是</b>了。整个表达式匹配的是<b>和</b>之间的内容(再次提醒，不包括前缀和后缀本身)。