正则表达式

正则表达式（英文：Regular Expression）在计算机科学中，是指一个用来描述或者匹配一系列符合某个句法规则的字符串的单个字符串。

常用的正则表达式

常用正则表达式

正则表达式用于字符串处理、表单验证等场合，实用高效。现将一些常用的表达式收集于此，以备不时之需。

用户名：/^[a-z0-9_-]{3,16}$/

密码：/^[a-z0-9_-]{6,18}$/

十六进制值：/^#?([a-f0-9]{6}|[a-f0-9]{3})$/

电子邮箱：/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/

URL：/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$/

IP 地址：
/^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/

HTML 标签：/^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$/

Unicode 编码中的汉字范围：/^[u4e00-u9fa5],{0,}$/

匹配中文字符的正则表达式： [\u4e00-\u9fa5]
评注：匹配中文还真是个头疼的事，有了这个表达式就好办了

匹配双字节字符(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）

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

匹配 HTML 标记的正则表达式：<(\S*?)[^>]*>.*?</\1>|<.*? />
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

匹配首尾空白字符的正则表达式：^\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、换页符等等)，非常有用的表达式

匹配 Email 地址的正则表达式：
\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*
评注：表单验证时很实用

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

匹配帐号是否合法(字母开头，允许 5-16 字节，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

匹配国内电话号码：\d{3}-\d{8}|\d{4}-\d{7}
评注：匹配形式如 0511-4405222 或 021-87888822

匹配腾讯 QQ 号：[1-9][0-9]{4,}
评注：腾讯 QQ 号从 10000 开始

匹配中国大陆邮政编码：[1-9]\d{5}(?!\d)
评注：中国大陆邮政编码为 6 位数字

匹配身份证：\d{15}|\d{18}
评注：中国大陆的身份证为 15 位或 18 位

匹配 ip 地址：\d+\.\d+\.\d+\.\d+
评注：提取 ip 地址时有用

匹配特定数字：
^[1-9]\d*$　　//匹配正整数
^-[1-9]\d*$　 //匹配负整数
^-?[1-9]\d*$　　//匹配整数
^[1-9]\d*|0$　//匹配非负整数（正整数 + 0）
^-[1-9]\d*|0$　　//匹配非正整数（负整数 + 0）
^[1-9]\d*\.\d*|0\.\d*[1-9]\d*$　　//匹配正浮点数
^-([1-9]\d*\.\d*|0\.\d*[1-9]\d*)$　//匹配负浮点数
^-?([1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.0+|0)$　//匹配浮点数
^[1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.0+|0$　　//匹配非负浮点数(正浮点数 + 0)
^(-([1-9]\d*\.\d*|0\.\d*[1-9]\d*))|0?\.0+|0$　 //匹配非正浮点数(负浮点数 + 0)
评注：处理大量数据时有用，具体应用时注意修正

匹配特定字符串：
^[A-Za-z]+$　　//匹配由 26 个英文字母组成的字符串
^[A-Z]+$　　//匹配由 26 个英文字母的大写组成的字符串
^[a-z]+$　　//匹配由 26 个英文字母的小写组成的字符串
^[A-Za-z0-9]+$　　//匹配由数字和 26 个英文字母组成的字符串
^\w+$　　//匹配由数字、26 个英文字母或者下划线组成的字符串

表达式全集

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“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”。
.	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)	负向预查，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
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*nn*	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。
\unnnn	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(?)。

正则表达式有多种不同的风格。下表是在 PCRE 中元字符及其在正则表达式上下文中的行为的一个完整列表：

以下是以 PHP 的语法所写的示例

验证字符串是否只含数字与英文，字符串长度并在 4~16 个字符之间

```
<?php
$str = 'a1234';
if (preg_match("^[a-zA-Z0-9]{4,16}$", $str)) {
    echo "驗證成功";
} else {
    echo "驗證失敗";
}
?>
```

简易的台湾身份证字号验证

```
<?php
$str = 'a1234';
if (preg_match("/^\w[12]\d{8}$/", $str)) {
    echo "驗證成功";
} else {
    echo "驗證失敗";
}
?>
```

以下示例是用 Perl 语言写的，与上面的示例功能相同

```
print $str = "a1234" =~ m:^[a-zA-Z0-9]{4,16}$: ? "COMFIRM" : "FAILED";
print $str = "a1234" =~ m"^\w[12]\d{8}$" ? "COMFIRM" : "INVAILD";
```

如何写出高效率的正则表达式

如果纯粹是为了挑战自己的正则水平，用来实现一些特效（例如使用正则表达式计算质数、解线性方程），效率不是问题；如果所写的正则表达式只是为了满 足一两次、几十次的运行，优化与否区别也不太大。但是，如果所写的正则表达式会百万次、千万次地运行，效率就是很大的问题了。我这里总结了几条提升正则表 达式运行效率的经验（工作中学到的，看书学来的，自己的体会），贴在这里。如果您有其它的经验而这里没有提及，欢迎赐教。

为行文方便，先定义两个概念。

误匹配：指正则表达式所匹配的内容范围超出了所需要范围，有些文本明明不符合要求，但是被所写的正则式“击中了”。例如，如果使用\d{11}来匹配 11 位的手机号，\d{11}不单能匹配正确的手机号，它还会匹配 98765432100 这样的明显不是手机号的字符串。我们把这样的匹配称之为误匹配。

漏匹配：指正则表达式所匹配的内容所规定的范围太狭窄，有些文本确实是所需要的，但是所写的正则没有将这种情况囊括在内。例如，使用\d{18}来匹配 18 位的身份证号码，就会漏掉结尾是字母 X 的情况。

写出一条正则表达式，既可能**只出现**误匹配（条件写得极宽松，其范围大于目标文本），也可能**只出现**漏匹配（只描述了目标文本中多种情况种的一种），还可能**既有误匹配又有漏匹配**。例如，使用\w+\.com 来匹配.com 结尾的域名，既会误匹配 abc_.com 这样的字串（合法的域名中不含下划线，\w 包含了下划线这种情况），又会漏掉 ab-c.com 这样的域名（合法域名中可以含中划线，但是\w 不匹配中划线）。

精准的正则表达式意味着既无误匹配且无漏匹配。当然，现实中存在这样的情况：只能看到有限数量的文本，根据这些文本写规则，但是这些规则将会用到海 量的文本中。这种情况下，尽可能地（如果不是完全地）消除误匹配以及漏匹配，并提升运行效率，就是我们的目标。本文所提出的经验，主要是针对这种情况。

掌握语法细节。正则表达式在各种语言中，其语法大致相同，细节各有千秋。明确所使用语言的正则的语法的细节，是写出正确、高效正则表达式的基础。例如，perl 中与\w 等效的匹配范围是[a-zA-Z0-9_]；perl 正则式不支持肯定逆序环视中使用可变的重复（variable repetition inside lookbehind，例如(?<=.*)abc），但是.Net 语法是支持这一特性的；又如，JavaScript 连逆序环视（Lookbehind,如(?<=ab)c） 都不支持，而 perl 和 python 是支持的。《精通正则表达式》第 3 章《正则表达式的特性和流派概览》明确地列出了各大派系正则的异同，这篇文章也简要 地列出了几种常用语言、工具中正则的比较。对于具体使用者而言，至少应该详细了解正在使用的那种工作语言里正则的语法细节。

先粗后精，先加后减。使用正则表达式语法对于目标文本进行描述和界定，可以像画素描一样，先大致勾勒出框架，再逐步在局步实现细节。仍举刚才的手机号的例子，先界定\d{11}，总不会错；再细化为 1[358]\d{9}， 就向前迈了一大步（至于第二位是不是 3、5、8，这里无意深究，只举这样一个例子，说明逐步

细化的过程）。这样做的目的是先消除漏匹配（刚开始先尽可能多 地匹配，做加法），然后再一点一点地消除误匹配（做减法）。这样有先有后，在考虑时才不易出错，从而向“不误不漏”这个目标迈进。

留有余地。所能看到的文本 sample 是有限的，而待匹配检验的文本是海量的，暂时不可见的。对于这样的情况，在写正则表达 式时要跳出所能见到的文本的圈子，开拓思路，作出“战略性前瞻”。例如，经常收到这样的垃圾短信：“发*票”、“发#漂”。如果要写规则屏蔽这样烦人的垃 圾短信，不但要能写出可以匹配当前文本的正则表达式 发[*#](?:票|漂)，还要能够想到 发.(?:票|漂|飘)之类可能出现的“变种”。这在具体的领域或许会有针对性的规则，不多言。这样做的目的是消除漏匹配，延长正则表达式的生命周期。

明确。具体说来，就是**谨慎**用点号这样的元字符，**尽可能**不用星号和加号这样的任意量词。只要能确 定范围的，例如\w，就不要用点号；只要能够预测重复次数的，就不要用任意量词。例如，写析取 twitter 消息的脚本，假设一条消息的 xml 正文部分结 构是…且正文中无尖括号，那么[^<]{1,480}这种写法**的思路**要好于.*，原因有二：一是使用[^<]，它保证了文本的范围不会超出下一个小于号所在的位置；二是明确长度范围，{1,480}，其依据是一条 twitter 消息大致能的字符长度范围。当然，480 这个长度是否正确还可推敲，但是这种思路是值得借鉴的。说得狠一点，“滥用点号、星号和加号是不环保、不负责任的做法”。

不要让稻草压死骆驼。每使用一个普通括号()而不是非捕获型括号(?:…)，就会保留一部分内存等着你再次访问。这样的正则表达式、无限次地运行次数，无异于一根根稻草的堆加，终于能将骆驼压死。养成合理使用(?:…)括号的习惯。

宁简勿繁。将一条复杂的正则表达式拆分为两条或多条简单的正则表达式，编程难度会降低，运行效率会提升。例如用来消除行首和行尾空白字符的正则表达式 s/^\s+|\s+$//g;，其运行效率理论上要低于 s/^\s+//g; s/\s+$//g; 。这个例子出自《精通正则表达式》第五章，书中对它的评论是“它几乎总是最快的，而且显然最容易理解”。既快又容易理解，何乐而不为？工作中我们还有其它的理由要将 C==(A|B)这 样的正则表达式拆为 A 和 B 两条表达式分别执行。例如，虽然 A 和 B 这两种情况只要有一种能够击中所需要的文本模式就会成功匹配，但是如果只要有一条子表达式 （例如 A）会产生误匹配，那么不论其它的子表达式（例如 B）效率如何之高，范围如何精准，C 的总体精准度也会因 A 而受到影响。

巧妙定位。有时候，我们需要匹配的 the，是作为单词的 the（两边有空格），而不是作为单词一部分的 t-h-e 的有序排列（例如 together 中的 the）。在适当的时候用上^，$，\b 等等定位锚点，能有效提升找到成功匹配、淘汰不成功匹配的效率。

正则表达式： http://114.xixik.com/regex/