正则表达式知识点

目录

[1. 正则表达式 正则表达式 一种字符串操作语言表达式 1](#_Toc21394)

[1.1. 简介 1](#_Toc18929)

[2. 正则表达式通常被用来检索、替换那些符合某个模式(规则)的文本。 2](#_Toc11517)

[3. 起源 2](#_Toc13460)

[3.1. 特点 2](#_Toc16909)

[4. 引擎 3类：一、DFA，二、传统型NFA，三、POSIX NFA。 3](#_Toc18835)

[5. 符号 4](#_Toc3358)

[5.1. 常用运算符与表达式： 7](#_Toc13917)

[5.2. 语法与释义： 7](#_Toc6257)

# 正则表达式 正则表达式 一种字符串操作语言表达式

[编辑](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/javascript:;)

**正则表达式**，又称规则表达式**。**（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。正则表达式通常被用来检索、替换那些符合某个模式(规则)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

正则表达式是对[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6" \t "https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)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。

## 简介

[编辑](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/javascript:;)

正则表达式是对字符串（包括普通字符（例如，a 到 z 之间的字母）和特殊字符（称为“元字符”））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。正则表达式是一种文本模式，模式描述在搜索文本时要匹配的一个或多个字符串

# 正则表达式通常被用来检索、替换那些符合某个模式(规则)的文本。

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。

验证 校验一些东西

# 起源

[编辑](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/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](https://baike.baidu.com/item/%E6%96%B0%E6%B3%BD%E8%A5%BF%E5%B7%9E" \t "https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)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1951 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

## 特点

[编辑](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/javascript:;)

正则表达式的特点是：

1. 灵活性、逻辑性和功能性非常强；

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。

# 引擎 3类：一、DFA，二、传统型NFA，三、POSIX NFA。

[编辑](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/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](https://baike.baidu.com/item/DFA" \t "https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank) 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](https://baike.baidu.com/item/%E5%9B%9E%E6%BA%AF%E7%AE%97%E6%B3%95" \t "https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)，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。

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

## 常用符号

.点 用来代表任何字符

\*星好 匹配前面的子表达式任意次

[编辑](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/javascript:;)

（摘自《[正则表达式之道](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F%E4%B9%8B%E9%81%93" \t "https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)》）

正则表达式由一些普通[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6" \t "https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)和一些[元字符](https://baike.baidu.com/item/%E5%85%83%E5%AD%97%E7%AC%A6" \t "https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F/_blank)（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。 |
| ^ | 匹配输入字行首。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入行尾。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”，也能匹配“zo”以及“zoo”。\*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”。?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为一组，后三个o为一组。“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地匹配所搜索的字符串，而默认的贪婪模式则尽可能多地匹配所搜索的字符串。例如，对于字符串“oooo”，“o+”将尽可能多地匹配“o”，得到结果[“oooo”]，而“o+?”将尽可能少地匹配“o”，得到结果 ['o', 'o', 'o', 'o'] |
| .点 | 匹配除“\n”和"\r"之外的任何单个字符。要匹配包括“\n”和"\r"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时很有用。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”。  \*python的正则表达式没有完全按照正则表达式规范实现，所以一些高级特性建议使用其他语言如java、scala等 |
| (?<!patte\_n) | 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”。  \*python的正则表达式没有完全按照正则表达式规范实现，所以一些高级特性建议使用其他语言如java、scala等 |
| x|y | 匹配x或y。例如，“z|food”能匹配“z”或“food”(此处请谨慎)。“[zf]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”任一字符。 |
| [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 | 匹配一个单词的边界，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”；“\b1\_”可以匹配“1\_23”中的“1\_”，但不能匹配“21\_3”中的“1\_”。 |
| \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]。grep 要加上-P，perl正则支持 |
| \D | 匹配一个非数字字符。等价于[^0-9]。grep要加上-P，perl正则支持 |
| \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\_]”。 |
| \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。 |

|  |  |  |  |
| --- | --- | --- | --- |
| **.** | **[ ]** | **^** | **$** |

四个字符是所有语言都支持的正则表达式，所以这四个是基础的正则表达式。正则难理解因为里面有一个等价的概念，这个概念大大增加了理解难度，让很多初学者看起来会懵，如果把等价都恢复成原始写法，自己书写正则就超级简单了，就像说话一样去写你的正则了

## **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

## **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

# 类似文件搜索通配符

主要有星号(\*)和问号(?)，用来模糊搜索文件。