

02  
OPEN ORIENTED  
凹凸实验室

# 正则表达式理论篇

暖暖

学习正则表达式的你们，有没有发现，一开始总是记不住语法。  
我这就来加深你们的印象。。。

## 正则表达式可以干嘛

- \* 数据验证。
- \* 复杂的字符串搜寻、替换。
- \* 基于模式匹配从字符串中提取子字符串。



正则表达式—理论篇



正则表达式—实践篇



正则表达式—理论篇

```
var regex = new RegExp('xyz', 'i');  
var regex = new RegExp(/xyz/i);  
var regex = /xyz/i;
```

```
// ES6  
new RegExp(/abc/ig, 'i').flags
```

// ES6的写法。ES5不允许此时使用第二个参数，会报错。

// 返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。下面代码返回“i”。

RegExp 怎么读？

一开始我很难记住这个单词，太难了！！

直到我知道了它的全拼是 regular expression

`String.search()`

参数：一个正则表达式。

返回：第一个与参数匹配的子串的起始位置，如果找不到，返回-1。

`String.replace()` 检索和替换。

第一个参数：正则表达式，

第二个参数：要进行替换的字符串，也可以是函数。

`String.match()`

参数：一个正则表达式。

返回：一个由匹配结果组成的数组。

`String.split()` 参数：正则表达式或字符串。返回：子串组成的数组。

`String.search()`

参数：一个正则表达式。

返回：第一个与参数匹配的子串的起始位置，如果找不到，返回-1。不支持全局搜索，如果参数是字符串，会先通过RegExp构造函数转换成正则表达式。

`String.replace()` 检索和替换。

第一个参数：正则表达式，

第二个参数：要进行替换的字符串，也可以是函数。

通过在替换字符串中使用“\$n”，可以使用子表达式相匹配的文本来替换字符。

`String.match()`

参数：一个正则表达式。

返回：一个由匹配结果组成的数组。设置g则返回所有匹配结果，否则数组的第一个元素是匹配的字符串，剩下的是圆括号中的子表达式，即a[n]中存放的是\$ n的内容。

`String.split()` 参数：正则表达式或字符串。返回：子串组成的数组。

`exec()`

参数：字符串。

在一个字符串中执行匹配检索，与String.macth()

非全局检索类似，返回一个数组或null。

`test()`

参数：字符串。返回true or false

`toString()`

转换成字符串形式。



\ 反斜杠字符（转义字符），用来转义

- 连字符 当且仅当在中括号[]的内部表示一个范围，比如[A-Z]就是表示范围从A到Z；如果需要在字符组里面表示普通字符-，放在字符组的开头即可[-A-Z]。

/ 正则表达式模式的开始或结尾

\ 反斜杠字符，用来转义

- 连字符 当且仅当在中括号[]的内部表示一个范围，比如[A-Z]就是表示范围从A到Z；如果需要在字符组里面表示普通字符-，放在字符组的开头即可[-A-Z]。

. 匹配除换行符 `\n` 之外的任何单个字符。

`\d` 等价 `[0-9]`

`\D` 与 `\d` 相反，等价于 `[^0-9]`

`\w` 与以下任意字符匹配：A-Z、a-z、0-9 和下划线，等价于 `[A-Za-z0-9_]`

`\W` 与 `\w` 相反，即 `[^A-Za-z0-9_]`

将 `^` 用作括号 `[]` 表达式中的第一个字符，则会对字符集求反。

\* 等价  $\{0,\}$   
+ 等价  $\{1,\}$   
? 等价  $\{0,1\}$   
{n} 正好匹配 n 次  
{n,} 至少匹配 n 次。  
{n,m} 匹配至少 n 次，至多 m 次。

\*+?这三个属于\*\*单字符限定符\*\*；

\*\*显示限定符\*\*位于大括号 {} 中，并包含指示出现次数上下限的数值。

- 1、显示限定符中，逗号和数字之间不能有空格！
- 2、贪婪量词\*和+：javascript默认是贪婪匹配，也就是说匹配重复字符是尽可能多地匹配。
- 3、惰性（最少重复匹配）量词?问号：当进行非贪婪匹配，只需要在待匹配的字符后面跟随一个问号?即可。

\*? 重复任意次，但尽可能少重复  
+? 重复1次或更多次，但尽可能少重复  
?? 重复0次或1次，但尽可能少重复  
{n,m}? 重复n到m次，但尽可能少重复  
{n,}? 重复n次以上，但尽可能少重复

`^` 匹配开始的位置。将 `^` 用作括号 `[]` 表达式中的第一个字符，则会对字符集求反。

`$` 匹配结尾的位置。

`\b` 与一个字边界匹配（`er\b` 与“never”中的“er”匹配，但与“verb”中的“er”不匹配。）

`\B` 非边界字匹配。

| 指示在两个或多个项之间进行选择。



中括号：

[] 标记括号表达式的开始和结尾，起到的作用是匹配这个或者匹配那个。

[...] 匹配方括号内\*\*任意字符\*\*。

很多字符在[]都会失去本来的意义：[^...]匹配不在方括号内的任意字符；[?.]匹配普通的问号和点号。

大括号：  
{} 标记限定符表达式的开始和结尾。{n,m}

() 标记子表达式的开始和结尾，主要作用是分组，对内容进行区分。

(模式) 可以记住和这个模式匹配的匹配项（捕获分组）。

(?:模式) 与模式 匹配，但不保存匹配项(非捕获分组)。

(?=模式) 零宽正向先行断言，要求匹配 与模式匹配的搜索字符串。不保存匹配项。

(?!模式) 零宽负向先行断言，要求匹配 与模式不匹配的搜索字符串。不保存匹配项。

有点晕?

先行断言：x只有在y前面才匹配，必须写成`/x(=y)/`。  
先行否定断言：x只有不在y前面才匹配，必须写成`/x(!y)/`。  
`。

ES7 提案:

后行断言: 与"先行断言"相反, x只有在y后面才匹配, 必须写成`/(?<=y)x^`。先匹配`/(?<=y)x/`的x, 然后再回到左边, 匹配y的部分, 即先右后左"的执行顺序。

后行否定断言: 与"先行否定断言"相反, x只有不在y后面才匹配, 必须写成`/(?<!y)x^`。

零宽负向先行断言的例子：

```
var str=`<div class="pin">
    <div class="box">
        
    </div>
</div>`;
var reg = /<(?!img)(?:.\\r\\n)*?>/gi;
console.log(str.replace(reg,""));
```

// 得到的结果为:

```
// < img src="img/P_001.jpg" />
```



反向引用：主要作用是给分组加上标识符\

n 表示引用字符，与第n个子表达式第一次匹配的字符相匹配

例子：

```
var reg = /(Mike)(\1)(s)/;  
var str = "MikeMikes";  
console.log(str.replace(reg,"$1$2'$3"));  
// MikeMike's
```

\s 任何空白字符。即[ \f\n\r\t\v]

\S 任何非空白字符。

\t Tab 字符(\u0009)。

\n 换行符(\u000A)

\v 垂直制表符(\u000B)。

\f 换页符(\u000C)

\r 回车符(\u000D)。

注意：\n和\r一起使用，即 /[\r\n]/g来匹配换行，因为unix扩展的系统以\n标志结尾，window以\n\r标志结尾。

`\cx` 匹配 `x` 指示的控制字符，要求 `x` 的值必须在 A-Z 或 a-z 范围内。

`\xn` 匹配 `n`，`n` 是一个十六进制转义码，两位数长。

`\un` 匹配 `n`，其中 `n` 是以四位十六进制数表示的 Unicode 字符。

`\nm` 或 `\n` 先尝试反向引用，不可则再尝试标识为一个八进制转义码。

`\nml` 当 `n` 是八进制数字 (0-3)，`m` 和 `l` 是八进制数字 (0-7) 时，匹配八进制转义码 `nml`。

i 执行不区分大小写的匹配。

g 执行一个全局匹配，简而言之，即找到所有的匹配，而不是在找到第一个之后就停止。

m 多行匹配模式，^匹配一行的开头和字符串的开头，\$匹配行的结束和字符串的结束。

u修饰符：

含义为“Unicode模式”，用来正确处理大于\uFFFF的Unicode字符。也就是说，会正确处理四个字节的UTF-16编码。

例子：

```
...
```

// 加u修饰符以后，ES6就会识别\uD83D\uDC2A为一个字符，返回false。

```
/^uD83D/u.test("\uD83D\uDC2A")  
...
```

// ES5会将\uD83D\uDC2A识别为两个字符，因此匹配成功。

```
/^uD83D/.test("\uD83D\uDC2A")
```

### y修饰符

与g修饰符都是全局匹配，不同之处在于，g修饰符只要剩余位置中存在匹配就可，而y修饰符确保匹配必须从剩余的第一个位置开始，这也就是“粘连”的涵义。

```
...  
/b/y.exec('aba') // null  
...
```

lastIndex属性指定每次搜索的开始位置，g修饰符从这个位置开始向后搜索，直到发现匹配为止。y修饰符要求必须在lastIndex指定的位置发现匹配。

\ 转义符  
(, (?), (?=), [] 小括号和中括号  
, +, ?, {n}, {n,}, {n,m} 限定符  
^, \$, \ 定位点和序列  
| 替换



JS 是 NFA 引擎。

特点：

- \* 急于邀功请赏，所以最左子正则式优先匹配成功，因此偶尔会错过最佳匹配结果（多选分支的情况）；
- \* backtracking（回溯），导致速度慢；
- \* 以贪婪方式进行，尽可能匹配更多字符。

急于邀功请赏的例子：

```
'''  
'nfa not'.match(/nfa\nfa not/)  
// 返回["nfa"]  
'''
```

回溯的例子：

```
```
```

```
'Lalala. Hi, barret. xxxxxxxx.Hello, John'.match(/H(ilello)/g)
```

```
// 返回["Hi", "Hello"]
```

```
```
```

回溯的例子：

```
...  
    ↓ 做一个标记  
    lastIndex  
'Lalala. Hi, barret. xxxxxxxx.Hello, John'.match(/H(ilello)/g)  
  
// 返回["Hi", "Hello"]  
...
```

回溯的例子：

```
```
```

```
'Lalala. Hi, barret. xxxxxxxx.Hello, John'.match(/H(ilello)/g)
```

```
// 返回["Hi", "Hello"]
```

```
```
```

Hi 分支匹配不了



回溯的例子：

回溯到刚才我们做标记的位置

```

'Lalala. Hi, barret. xxxxxxxx.Hello, John'.match(/H(ilello)/g)

// 返回["Hi", "Hello"]

```

回溯的例子：

Hello匹配，做一个标记lastIndex

```

'Lalala. Hi, barret. xxxxxxxx.Hello, John'.match(/H(ilello)/g)

// 返回["Hi", "Hello"]

```

贪婪的例子：

```
```
```

```
"AB1111BA111BAcccc".match('AB.*BA')
```

```
// 返回["AB1111BA111BA"]
```

```
```
```

```
"AB1111BA111BA".match('AB.*?BA')
```

```
// 返回 ["AB1111BA"]
```




贪婪的例子：

...  
↓ AB  
"AB1111BA111BAcccc".match('AB.\*BA')  
// 返回["AB1111BA111BA"]  
...

贪婪的例子：

```
...  
"AB1111BA111BAcccc".match('AB.*BA')  
// 返回["AB1111BA111BA"]  
...
```



为了匹配星号后面的B，回溯到上一次匹配的地方，  
也就是上次.\*匹配的位置，  
倒数第一个字符不是B，再回溯

贪婪的例子：

```

"AB1111BA111BAcccc".match('AB.\*BA')

// 返回["AB1111BA111BA"]

```

贪婪的例子:

再回溯，向前找B。  
找到字符B，继续匹配A

```

"AB1111BA111BAcccc".match('AB.\*BA')

// 返回["AB1111BA111BA"]

```

贪婪的例子：

找到字符A，匹配完成，停止匹配

```



`"AB1111BA111BAcccc".match('AB.*BA')`

`// 返回["AB1111BA111BA"]`

```



正则表达式—实践篇

1. 与搜索字符串开始处的 3 个数字匹配。
2. 与除 a、b 和 c 以外的任何字符匹配。
3. 在搜索字符串“1234567”中，`\d{1,3}/` 匹配的结果。
4. 不以“th”开头的单词匹配。
5. 对密码应用以下限制：其长度必须介于 4 到 8 个字符之间，并且必须至少包含一个数字。
6. 匹配中文

1. 与搜索字符串开始处的 3 个数字匹配: `/^\d{3}/`
2. 与除 a、b 和 c 以外的任何字符匹配: `/[^abc]/`
3. 在搜索字符串“1234567”中, `\d{1,3}/` 与 “123”、“456”和“7”匹配 (贪婪原则)。
5. 对密码应用以下限制: 其长度必须介于 4 到 8 个字符之间, 并且必须至少包含一个数字。正则为 `/^(?=.*\d).\{4,8}$/`。 `.*` 表示单个字符 (除换行符 `\n` 外) 零次或多次, 后面跟着一个数字; 而 `\{4,8\}` 与包含 4-8 个字符的字符串匹配。
6. 匹配中文 `/[\u4e00-\u9fa5]/`



要想在复杂性和完整性之间取得平衡，一个重要因素是要了解将要搜索的文本。

好的正则表达式：

- \* 只匹配期望的文本，排除不期望的文本；
- \* 易于控制和理解；
- \* 保证效率

有时候处理各种极端情况会降低成本/收益的比例，所以某些情况下，不完全依赖正则表达式完成全部工作。比如某些字段用子表达式()括起来，让内存记忆下来，然后再用其他程序来验证。

- 1、匹配美元
- 2、匹配IP地址
- 3、24小时制的时间，比如 09:59

```
/^\$[0-9]+(\.[0-9][0-9])?\$ /
```

分为四部分：

\* `^\$`以美元符号开头

\* `[0-9]+` 至少包含一个数字

\* `(\.[0-9][0-9])?`由一个点和两位数组成，匹配0次或1次

\* `\$`最后的`$`表示以数字结尾的

缺点：不能匹配\$1,000

规则：点号分开的四个字段，每个字段在0-255之间。

第一步：

如果一个字段是一个数或两个数，肯定是在0-255的范围内的；如果三位数，那么以0或者1开头的三位数也是合法的，即000-199。

从上面的陈述中我们就可以得到：

```
\d\d\d|01\d\d
```

我们稍微合并一下这三个多选分支，得到：

```
[01]?d\d?
```

第二步：

我们再来看以2开头的三位数：第二位数小于5的时候，第三位数范围[0-9]都可以；第二位数为5的时候，第三位数范围[0-5]：

`2[0-4]\d|25[0-5]`

第三步：

最后合并起来，得到一个字段0-255的表示方法：

```
[01]?\\d\\d?!2[0-4]\\d|25[0-5]
```



第四步：

IP地址正则如下：

```
/^([01]?[d\d]?[2[0-4]\d|25[0-5])\.([01]?[d\d]?[2[0-4]\d|25[0-5])\.([01]?[d\d]?[2[0-4]\d|25[0-5])\.([01]?[d\d]?[2[0-4]\d|25[0-5])$/
```

第四步：

IP地址正则如下：

```
/^([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])$/
```

点号要转义一下，^和\$需要加上，否则可能匹配52123.3.22.993，因为其中的123.3.22.99是符合的。

虽然0.0.0.0是合法的，但它是非法的IP地址，如果支持环视功能，可以加上(?!0+\.0+\.0+\.0+\$)

### 测试

```
> /^( [01]? \d\d? | 2[0-4]\d | 25[0-5])\. ( [01]? \d\d? | 2[0-4]\d | 25[0-5])\. ( [01]? \d\d? | 2[0-4]\d | 25[0-5])\. ( [01]? \d\d? | 2[0-4]\d | 25[0-5])$/ .exec( '123.11.22.33.44')
< null
> /^( [01]? \d\d? | 2[0-4]\d | 25[0-5])\. ( [01]? \d\d? | 2[0-4]\d | 25[0-5])\. ( [01]? \d\d? | 2[0-4]\d | 25[0-5])$/ .exec( '123.11.22.33')
< ["123.11.22.33", "123", "11", "22", "33"]
> /^( [01]? \d\d? | 2[0-4]\d | 25[0-5])\. ( [01]? \d\d? | 2[0-4]\d | 25[0-5])\. ( [01]? \d\d? | 2[0-4]\d | 25[0-5])$/ .exec( '52123.3.22.993')
< ["123.3.22.99", "123", "3", "22", "99"]
```

小时部分：

方法一：分类逻辑为第一个数字分别为0、1、2，可以分为三部分，即上午 00到09点（0可选），白天10到19点，晚上20到23点。

因此得到的小时为：

`0?[0-9]|1[0-9]|2[0-3]`

还可以优化一下，合并前面的两个多选分支，得到

`[01]?[0-9]|2[0-3]`

方法二：分类逻辑也可以为**第二个数字**，第二个数字分组为[0-3]以及[4-9]，为什么这么分？看看下面这个图就知道了，[0-3]多了以2为第一个数字的一行：

`[012]?[0-3][01]?[4-9]`

0	1	2	3		4	5	6	7	8	9
00	01	02	03		04	05	06	07	08	09
10	11	12	13		14	15	16	17	18	19
20	21	22	23							

分钟数：

比较简单，第一个数范围在0-5之间，第二个数在0-9之间，因此得到分钟数为：

`[0-5][0-9]`

小时、冒号、分钟数加起来，就得到最终的结果：

`0?[0-9]|1[0-9]|2[0-3]:[0-5][0-9]`

或者

`[012]?[0-3][01]?[4-9]:[0-5][0-9]`

- 1、匹配HTML Tag
- 2、匹配日期
- 3、匹配HTTP URL



MDN

<http://barretlee.com/blog/2014/01/18/cb-how-regular-expressions-work/>

[http://www.cnblogs.com/hustskyking/archive/2013/06/04/  
RegExp.html#greedyandlazy](http://www.cnblogs.com/hustskyking/archive/2013/06/04/RegExp.html#greedyandlazy)

《精通正则表达式》



THANKS FOR LISTENING

王彩暖