
正则表达式理论篇
/ ^ [A-Z] \d{5} $ /gi

正则表达式理论篇

by 暖暖 on 2016-11-17

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。嗯，加深大家的印象的同时，我也是来找同道中人的。

首先你要记住它的名字

正则表达式
regular expression
缩写 regexp 、regex 、egrep。

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。

概述

正则表达式包括普通字符（例如，a 到 z 之间的字母）和特殊字符（称为“元字符”）。

若要匹配这些特殊字符，必须首先转义字符，即，在字符前面加反斜杠字符 `\` **。
例如，若要搜索 “+” 文本字符，可使用表达式 `\+` 。

但是大多数 特殊字符 在中括号表达式内出现时失去本来的意义，并恢复为普通字符。

构造函数（四种写法）

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

// ES6的写法。ES5在第一个参数是正则时，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”。
new RegExp(/abc/ig, 'i').flags
```

用于模式匹配的String方法

- String.search()
  参数：要搜索的子字符串，或者一个正则表达式。
  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。
  说明：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()
  作用：查找并替换字符串。
  第一个参数：字符串或正则表达式，
  第二个参数：要进行替换的字符串，也可以是函数。
  用法：

替换文本中的$字符有特殊含义：

```
$1、$2、...、$99 与 regexp 中的第 1 到第 99 个子表达式相匹配的文本。
$&       与 regexp 相匹配的子串。
$`       位于匹配子串左侧的文本。
$'       位于匹配子串右侧的文本。
$$       普通字符$。
```

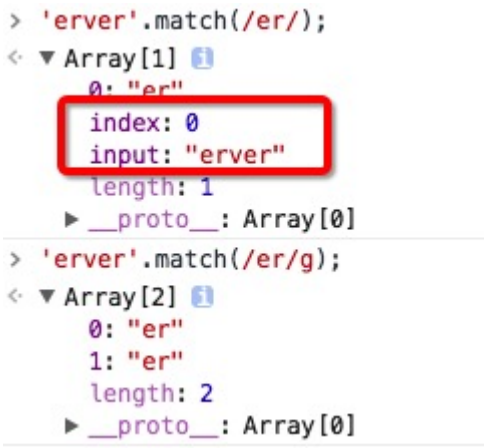
如：

```
'abc'.replace(/b/g, "{$$$`$&$'}")
// 结果为 "a{$abc}c"，即把b换成了{$abc}
```

- String.match()
  参数：要搜索的子字符串，或者一个正则表达式。
  返回：一个由匹配结果组成的数组。

非全局检索：如果没有找到任何匹配的文本返回null；否则数组的第一个元素是匹配的字符串，剩下的是小括号中的子表达式，即a[n]中存放的是$n的内容。非全局检索返回三个属性：length 属性；index 属性声明的是匹配文本的第一个字符的位置；input 属性则存放的是被检索的字符串 string。

全局检索：设置标志g则返回所有匹配子字符串，即不提供与子表达式相关的信息。没有 index 属性或 input 属性。



- String.split()
  作用：把一个字符串分割成字符串数组。
  参数：正则表达式或字符串。
  返回：子串组成的数组。

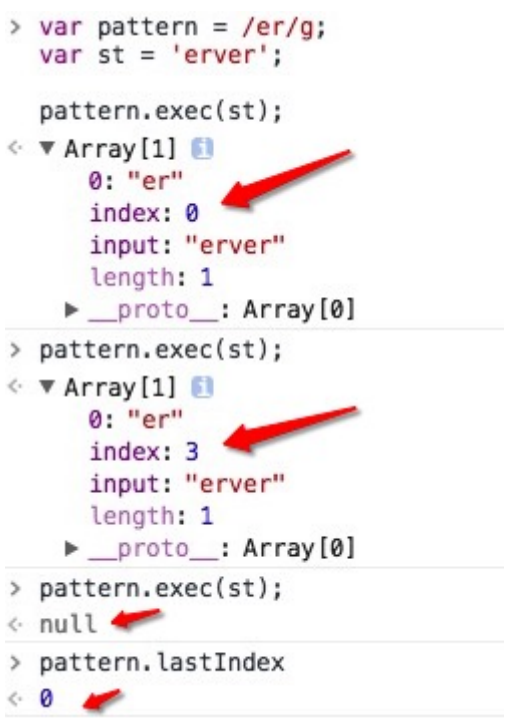
RegExp的方法

- RegExpObject.exec()
  参数：字符串。
  返回：

非全局检索：与String.macth()非全局检索相同，返回一个数组或null。

全局检索：尽管是全局匹配的正则表达式，但是exec方法只对指定的字符串进行一次匹配。但是可以反复调用来实现全局检索。在RegExpObject 的lastIndex 属性指定的字符处开始检索字符串；匹配后，将更新lastIndex为匹配文本的最后一个字符的下一个位置；再也找不到匹配的文本时，将返回null，并把 lastIndex 属性重置为 0。

如：



- RegExpObject.test()
  参数：字符串。
  返回：true或false。
- RegExpObject.toString()
  返回：字符串

字符

`|` 指示在两个或多个项之间进行选择。类似js中的或，又称分支条件。
`/` 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 当且仅当在字符组[]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-，放在字符组的开头或者尾部即可。

`.` 匹配除换行符 \n 之外的任何单个字符。
`\d` 等价[0-9]，匹配0到9字符。
`\D` 等价[^0-9]，与 `\d` 相反。
`\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]

限定符（量词字符）

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；`*+?` 这三个字符属于单字符限定符：

`{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。
`*` 等价{0,}
`+` 等价{1,}
`?` 等价{0,1}

注意：

- 显示限定符中，逗号和数字之间不能有空格，否则返回null！

- 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。
- 惰性（最少重复匹配）量词 ? ：当进行非贪婪匹配，只需要在待匹配的字符后面跟随一个 ? 即可。

```
var reg = /a+/;
var reg2 = /a+?/;
var str = 'aaab';

str.match(reg); // ["aaa"]
str.match(reg2); // ["a"]
```

定位点（锚字符、边界）

^ 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，如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。

标记

- 中括号 [] 字符组；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。
  注意：反斜杠字符 \ 在 [] 中仍为转义字符。若要匹配反斜杠字符，请使用两个反斜杠 \\ 。
  另外不要滥用字符组这个失去意义的特性，比如不要使用 [.] 来代替 \. 转义点号，因为需要付出处理字符组的代价。
- 大括号 {} 标记限定符表达式的开始和结尾。

- 小括号`()`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）。不要滥用括号，如果不需要保存子表达式，可使用非捕获型括号`(?:)`来进行性能优化。
`(?: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。 找到一个匹配项后，将在匹配文本之前开始搜索下一个匹配项；但不会保存匹配项。
`(?!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。 找到一个匹配项后，将在匹配文本之前开始搜索下一个匹配项；但不会保存匹配项。

有点晕？
好，换个说法。。。

先行断言`(?=模式)`：x只有在y前面才匹配，必须写成`/x(?=y)/`。 解释：找一个x，那个x的后面有y。
先行否定断言`(?!模式)`： x只有不在y前面才匹配，必须写成`/x(?!y)/`。 解释：找一个x，那个x的后面没有y。

稳住，又来了两个断言，来自ES7提案：

后行断言`(?<=模式)`：与”先行断言”相反， x只有在y后面才匹配，必须写成`/(?<=y)x/`。解释：找一个x，那个x的前面要有y。
后行否定断言`(?<!模式)`： 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/`。 解释：找一个x，那个x的前面没有y。

可以看出，后行断言先匹配/(?<=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。

零宽负向先行断言的例子：

```
1  var str=`<div class="o2">
```

```
            <div class="o2_team">
                <img src="img/logo.jpg" />
            </div>
        </div>`;
// <(?!img) 表示找一个左尖括号<，而且左尖括号<的后面没有img字符；
// (?:.|\r|\n)*? 表示匹配左右尖括号<>里面的.或\r或\n，而且匹配次数为*?；(?:)不保存匹配项，提高性能；
// *后面加个? 表示非贪婪匹配。
var reg = /<(?!img)(?:.|\r|\n)*?>/gi;
str.match(reg);
// 返回结果 ["<div class="o2">", "<div class="o2_team">", "</div>", "</div>"]
```

- 反向引用：主要作用是给分组加上标识符\n。
  `\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。

反向引用的例子，给MikeMike字符后加个单引号：

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

非打印字符

`\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 `\r\n` 标志结尾。

其他

`\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。

修饰符

- `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。

ES6新增u和y修饰符：

- `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。

```
// 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')  // false
/^\uD83D/.test('\uD83D\uDC2A') // true
```

- `y` 修饰符

与g修饰符都是全局匹配，不同之处在于：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；但是 `y` 修饰符要求必须在lastIndex指定的位置发现匹配，即 `y` 修饰符确保匹配必须从剩余的第一个位置开始，这也是“粘连”的涵义。

```
/b/y.exec('aba')  // null
/b/.exec('aba')  // ["b"]
```

优先级顺序：

1. `\` 转义符
2. `(), (?:), (?=), []` 括号和中括号
3. `*、+、?、{n}、{n,}、{n,m}` 限定符
4. 任何元字符 `^、$、\` 定位点和序列
5. `|` 替换

关于引擎

JS 是 NFA 引擎。

NFA 引擎的特点：

- 以贪婪方式进行，尽可能匹配更多字符。
- 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（多选条件分支的情况）。

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

- 回溯（backtracking），导致速度慢。

举个贪婪与回溯结合的例子：

```
"AB01CD23CD45CEff".match('AB.*CD')
// 返回 ["AB01CD23CD"]
```

匹配顺序如图所示：

参考

MDN

w3school

http://es6.ruanyifeng.com/#docs/regex

http://imweb.io/topic/56e804ef1a5f05dc50643106

http://www.cnblogs.com/deerchao/archive/2006/08/24/zhengzhe30fengzhongjiaocheng.html
http://www.cnblogs.com/hustskyking/p/how-regular-expressions-work.html

上次更新：2016-12-30 10:19:37

❮ XCel 项目总结 - Electron 与 Vue 的性能优化

「塔罗牌」 - 轻氧 V1.4 尝鲜体验邀请 ❯

评论框出错啦(990015): 服务异常,请联系客服人员