JS正则表达式元字符

时间 2015-01-08 01:51:41 segmentfault-博客
原文 http://segmentfault.com/blog/jslite/1190000002471140
主题 JavaScript 正则表达式

正则	描述	正则	描述
\f	匹配换页符	\t	匹配制表符
\n	匹配换行符	\v	匹配垂直制表符
\r	匹配回车	\s	匹配单个空格，等同于 [\f\n\r\t\v] ；
\S	表示非空格字符	\d	在正则中表示数字
\w	表示单词字符，等同于字符集合 [a-zA-Z0-9_]	\	创建正则对象时，要转义
+	表示出现一到多次	^	表示后边出现的数字必须在开头
$	表示前面出现的数字必须出现在结尾		

视频教程

g 是模式修正符，表示在整个字符串里多次查找

match方法来找到和reg正则相匹配的内容

```
js
"ab23839cd".match(/\d+/)
```

空白\s（包括空格回车制表符等）

\f 匹配换页符， \n 匹配换行符， \r 匹配回车， \t 匹配制表符， \v 匹配垂直制表符。

\s 匹配单个空格，等同于 [\f\n\r\t\v] 。例如：

例1：

```
js
var reg = /\s.+/;
var str='This is a test String.';
alert(reg.exec(str));
```

返回 “is a test String.” ，正则的意思是匹配第一个空格以及其后的所有非换行字符。

同样， \S 表示非空格字符。

例2：

```
js
var reg = /\S+/;
var str='This is a test String.';
alert(reg.exec(str));
```

匹配结果为 This ，当遇到第一个空格之后，正则就停止匹配了。

在正则中最常用到的一个是：

例3： var reg=/^\s*$/; //匹配任意空或空白字符，如果你什么也没输入，或输入的只有空格、回车、换行等字符，则匹配成功。这样就可以验证用户是否正确输入内容了。

这个用来验证输入框里是否写了有效效字符,用法如下：

```
js
var  reg=/^\s*$/;
if(reg.test(value)){
  alert('请输入有效值');
  return false;
}
```

单字符\w

\w 表示单词字符，等同于字符集合 [a-zA-Z0-9_] 。例如：

```
js
var reg = /\w+/;
var str='zhufengpeixun';
alert(reg.exec(str));
```

返回完整的zhufengpeixun字符串，因为所有字符都是单词字符。

```
js
var reg = /\w+/;
var str='.className';
alert(reg.exec(str));
```

结果显示匹配了字符串中的 className ，只有第一个 “.” 唯一的非单词字符没有匹配。

```
js
var reg = /\w+/;
```

```
var str='正则教程';
alert(reg.exec(str));
```

试图用单词字符去匹配中文自然行不通了，返回 null 。

\W 表示非单词字符，等效于 [^a-zA-Z0-9_]

```
js
var reg = /\W+/;
var str='正则教程';
alert(reg.exec(str));
```

返回完整的字符串，因为，中文算作是非单词字符。

分组和分组的引用

请见在线视频的正则专题部分的

正则表达式基础第三讲：分组、分组的引用、选择等
http://online.zhufengpeixun.cn/viewCourseDetail.do?courseId=141415

形式如下：/(子正则表达式)\1/ 依旧用例子来说明：

1.例子

```
js
var reg = /\w/;
var str='zhufengpeixun';
alert(reg.exec(str));
//返回z。
```

2.例子

```
js
var reg = /(\w)(\w)/;
var str='zhufengpeixun';
alert(reg.exec(str));
```

返回 zh,z,h ， zh 是整个正则匹配的内容， z 是第一个括号里的子正则表达式匹配的内容，h 是第二个括号匹配的内容。

3.例子

```
js
var reg = /(\w)\1/;
var str='zhufengpeixun';
alert(reg.exec(str));
```

则会返回 null 。这里的 “\1” 就叫做反向引用，它表示的是第一个括号内的子正则表达式匹配的内容。在上面的例子中，第一个括号里的 (\w) 匹配了z，因此 “\1” 就同样表示z了，在余下的字符串里自然找不到z了。与第二个例子对比就可以发现， “\1” 是等同于 “第1个括号匹配的内容” ，而不是 “第一个括号的内容” 。

```
js
```

```
var reg = /(\w)\1/;
var str='bbs.zhufengpeixun.cn';
alert(reg.exec(str));
```

这个正则则会匹配到 bb,b 。同样，前面有几个子正则表达式我们就可以使用几个反向引用。例如：

```
js
var reg = /(\w)(\w)\2\1/;
var str='woow';
alert(reg.exec(str));
```

会匹配成功，因为第一个括号匹配到w，第二个括号匹配到 o ，而 \2\1 则表示 ow ，恰好匹配了字符串的最后两个字符。

括号 () ，表示子表达式，也叫分组

前面我们曾经讨论过一次括号的问题，见下面这个例子：

```
js
var reg = /^(b|c).+/;
var str='bbs.blueidea.com';
alert(reg.exec(str));
```

这个正则是为了实现只匹配以b或者c开头的字符串，一直匹配到换行字符，但是。上面我们已经看到了，可以使用“\1”来反向引用这个括号里的子正则表达式所匹配的内容。而且exec方法也会将这个字正则表达式的匹配结果保存到返回的结果中。

不记录子正则表达式的匹配结果[匹配不捕获]

使用形如 (?:pattern) 的正则就可以避免保存括号内的匹配结果。例如：

```
js
var reg = /^(?:b|c).+/;
var str='bbs.blueidea.com';
alert(reg.exec(str));
```

可以看到返回的结果不再包括那个括号内的字正则表达式多匹配的内容。同理，反向引用也不好使了：

```
js
var reg = /^(b|c)\1/;
var str='bbs.zhufengpeixun.cn';
alert(reg.exec(str));
```

返回 bb,b 。 bb 是整个正则表达式匹配的内容，而b是第一个子正则表达式匹配的内容。

```
js
var reg = /^(?:b|c)\1/;
var str='bbs.zhufengpeixun.cn';
alert(reg.exec(str));
```

返回 null 。由于根本就没有记录括号内匹配的内容，自然没有办法反向引用了。

正向预查

形式： (?=pattern) 所谓正向预查，意思就是：要匹配的字符串，后面必须紧跟着 pattern ！我们知道正则表达式 /cainiao/ 会匹配 cainiao 。同样，也会匹配 cainiao9 中的 cainiao 。但是我们可能希望， cainiao 只能匹配 cainiao8 中的 cainiao 。这时候就可以像下面这样写：/cainiao(?=8)/ ，看两个实例：

```
js
var reg = /cainiao(?=8)/;
var  str='cainiao9';
alert(reg.exec(str));
//返回null。
```

```
js
var reg = /cainiao(?=8)/;
var  str='cainiao8';
alert(reg.exec(str));
```

匹配 cainiao 。需要注意的是，括号里的内容并不参与真正的匹配，只是检查一下后面的字符是否符合要求而已，例如上面的正则，返回的是 cainiao ，而不是 cainiao8 。

再来看几个例子：

```
js
var reg = /zhufeng(?=peixun)/;
var  str='zhufengpeixun';
alert(reg.exec(str));
```

匹配到 zhufeng ，而不是 peixun 。

```
js
var reg = /zhufeng(?=peixun)/;
var str=' zhufengonline' ;
alert(reg.exec(str));
```

返回 null ，因为 zhufeng 后面不是 peixun 。

```
js
var reg = /zhufeng(?=peixun)/;
var str='onlinepeixun';
alert(reg.exec(str));
```

同样返回 null 。

?!

形式 (?!pattern) 和 ?= 恰好相反，要求字符串的后面不能紧跟着某个pattern，还拿上面的例子：

```
js
var reg = /zhufeng(?!js)/;
var str=`zhufengjs`;
alert(reg.exec(str));
```

返回 null ，因为正则要求， zhufeng 的后面不能是 js 。

```
js
var reg = /zhufeng(?!js)/;
var  str='zhufengpeixun';
alert(reg.exec(str));
```

则成功返回 zhufeng 。

匹配元字符

首先要搞清楚什么是元字符呢？我们之前用过 * ， + ， ？之类的符号，它们在正则表达式中都有一定的特殊含义，类似这些有特殊功能的字符都叫做元字符。例如

```
js
var reg = /c*/;
```

表示有任意个c，但是如果我们真的想匹配 c* 这个字符串的时候怎么办呢？只要将*转义了就可以了，如下：

```
js
var reg = /c\*/;
var str='c*';
alert(reg.exec(str));
```

返回匹配的字符串： c* 。

同理，要匹配其他元字符，只要在前面加上一个“ \ ”就可以了。

正则表达式的修饰符

全局匹配，修饰符 g

形式： /pattern/g 例子： reg = /b/g; 后面再说这个 g 的作用。先看后面的两个修饰符。不区分大小写，修饰符 i

形式： /pattern/i 例子：

```
js
var reg = /b/;
var str = 'BBS';
alert(reg.exec(str));
```

返回 null ，因为大小写不符合。

```
js
var reg = /b/i;
var str = 'BBS';
alert(reg.exec(str));
```

匹配到 B ，这个就是 i 修饰符的作用了。

行首行尾，修饰符 m

形式： /pattern/m m 修饰符的作用是修改 ^ 和 $ 在正则表达式中的作用，让它们分别表示行首和行尾。例如：

```
js
var reg = /^b/;
var str = 'test\nbbs';
alert(reg.exec(str));
```

匹配失败，因为字符串的开头没有 b 字符。但是加上m修饰符之后：

```
js
var reg = /^b/m;
var str = 'test\nbbs';
alert(reg.exec(str));;
```

匹配到 b ，因为加了 m 修饰符之后， ^ 已经表示行首，由于 bbs 在字符串第二行的行首，所以可以成功地匹配。

匹配固定的n个 c{n}

{1} 表示一个的意思。 /c{1}/ 只能匹配一个 c ，和 /c/ 是一个意思，一般匹配只出现一次的字符，后边的 {1} 就不写了。 /c{2}/ 则会匹配两个连续的c。以此类推， /c{n}/ 则会匹配 n 个连续的 c 。看下面的例子：

```
js
var reg = /c{1}/;
var str='china_zhufengpeixun';
alert(reg.exec(str));
//输出结果是：c

var reg = /o{2}/;
var str='money';
alert(reg.exec(str));
//返回结果`null`，表示没有匹配成功。

reg = /o{2}/;
str='good food';
alert(reg.exec(str))
//输出结果oo。（其实是第一组oo，不会匹配到第二组oo，因为正则的匹配是懒惰的，不加模式匹配符g，则表示只去匹配-
```

如果写成

```
js
reg=/o{2}/g;
alert(str.match(reg))
//则输出oo,oo了
```

c{m,n} 匹配最少m个，最多n个

c{3,4} 的意思是，连续的3个 c 或者4个 c 。例如：

```
js
reg = /o{3,4}/;// (匹配三到四个o)
str='good正则教程';
alert(reg.exec(str));
```

返回结果null，表示没有匹配成功。例：

```
js
reg = /o{3,4}/;
str='goood正则教程';
alert(reg.exec(str));
```

弹出结果是：ooo。例：

```
js
reg = /o{3,4}/;
str='very gooood正则教程';
alert(reg.exec(str));
```

输出的结果是： oooo ，这表明正则会尽量多匹配，可3可4的时候它会选择多匹配一个。（这就是贪婪匹配）例：

```
js
reg = /c{3,4}/;
str='cccccTest';
alert(reg.exec(str));
```

仍然会匹配4个c。

由以上例子可以推断出： c{m,n} 表示m到n个c，且m小于等于n。

c{n,}表示最少匹配n个c，最多不限制

c{1,} 表示1个以上的c,相当于 + 。如下：

例：

```
js
reg = /c{1,}/;str='cainiao';
alert(reg.exec(str));
//结果弹出c。
```

例：

```
js
reg = /c{1,}/;
str='ccccTest';
alert(reg.exec(str));
//返回ccccc，再次说明了正则表达式会尽量多地匹配。
```

例：

```
js
reg = /c{2,}/;
str='cainiao';
alert(reg.exec(str));
```

结果返回 null ， c{2,} 表示2个以上的c，而cainiao中只有1个c。

由以上例子可知，c{n,}表示最少n个c，最多则不限个数。

综合： *,+,?

* 表示 次或者多次，等同于 {0,} ，即 c* 和 c{0,} 是一个意思。

+ 表示一次或者多次，等同于 {1,} ，即 c+ 和 c{1,} 是一个意思。

最后， ? 表示 次或者 次，等同于 {0,1} ，即 c? 和 c{0,1} 是一个意思。

贪心与非贪心【贪婪匹配和非贪婪匹配】

人都是贪婪的，正则也是如此。我们在例子 reg = /c{3,4}/;str='ccccTest'; 的例子中已经看到了，能匹配四个的时候，正则绝对不会去匹配三个。上面所介绍的所有的正则都是这样，只要在合法的情况下，它们会尽量多去匹配字符，这就叫做贪心模式。如果我们希望正则尽量少地匹配字符，那么就可以在表示数字的符号后面加上一个 ?（即：问号加在量词的后边，则表示非贪婪匹配） 。组成如下的形式：

```
js
{n,}?, *?, +?, ??, {m,n}?
```

例：

```
js
reg = /c{1,}?/;
str='ccccc';
 alert(reg.exec(str));
```

返回的结果只有1个c，尽管有5个c可以匹配，但是由于正则表达式是非贪心模式，所以只会匹配一个。

/^开头,结尾$/ 【表示位置】

^表示只匹配字符串的开头。看下面的例子：

例1：

```
js
reg = /^c/;
str='维生素c';
alert(reg.exec(str));
```

结果为 null ，因为字符串 ‘ 维生素c ’ 的开头并不是c，所以匹配失败。

例2：

```
js
reg = /^z/;
str='zhufengpeixun';
alert(reg.exec(str));;
```

这次则返回 c ，匹配成功，因为cainiao恰恰是以z开头的。

与 ^ 相反， $ 则只匹配字符串结尾的字符，同样，看例子：

例3：

```
js
reg = /z$/;
str='zhufengpeixun';
alert(reg.exec(str));
```

输出 null ，表示正则表达式没能在字符串的结尾找到 z 这个字符。

例4：

```
js
reg = /d$/;
str='正则教程good';
alert(reg.exec(str));
```

这次返回的结果是 d ，表明匹配成功。

元字符点’.’的用法

. 会匹配字符串中除了换行符 \n 之外的所有字符，例如

```
js
reg = /./; //一个点表示匹配字符串中出现的第一个非换行符字符。
str='zhufengpeixun';
alert(reg.exec(str));;
```

结果显示，正则匹配到了字符 z 。

```
js
reg = /./;
str='online.zhufengpeixun';
alert(reg.exec(str));
```

这次是 o ，只要有一个是非换行字符，就表示匹配成功，就不会往下再继续了。

```
js
reg = /.+/;
str='zhufengpeixun_  前端开发权威培训';
alert(reg.exec(str));
```

结果是“zhufengpeixun_前端开发“也就是说所有的字符都被匹配掉了，包括一个空格，一个下滑线

【贪婪匹配】。

例1：

```
js
reg = /.+/;
```

```
str='online.zhufengpeixun.cn';
alert(reg.exec(str));;
```

同样，直接返回整个字符串——online.zhufengpeixun.cn，可见”.”也匹配”.”本身。

例2：

```
js
reg = /^./; //这样表示必须以非换行符开始。
str='\nzhufengpeixun';
alert(reg.exec(str));
```

结果是 null ，终于失败了，正则要求字符串的第一个字符不是换行，但是恰恰字符是以\n开始的。

“|“，正则表达式中的或，把“|”左右两边的一到多个字符当成一个整体对待

b|c 表示，匹配 b 或者 c （这里相当于 [bc] ）。 ab|ac 表示匹配 ab 或 ac （但这里不相当于 [abc] ， [] 表示在一组字符中任选一个）。

例1：

```
js
/z|o/.exec('zhufengpeixun'))
//结果是z。
```

例2：

```
js
/z|o/.exec('online');
//结果是o。
```

例3：

```
js
/^z|o.+/.exec('online');
//匹配掉整个online。
```

例4：

```
js
/^z|o.+/.exec('zhufengpeixun.cn');
//结果只有一个z，而不是整个字符串。因为上面正则表达式的意思是，匹配开头的z或者是o.+。
```

和括号结合使用

例：

```
js
/^(z|o).+/.exec('zhufengpeixun');
```

这次的结果是整个串zhufengpeixun，加上上面的括号这后，这个正则的意思是，如果字符串的开头是z或者o，那么匹配开头的z或者o以及其后的所有的非换行字符。如果你也实验了的话，会发现返回的结果后面多出来一个“z“，这是 () 内的 z|o 所匹配的内容(这个叫分组或子正则)。我们在正则表达式内括号里写的内容会被认为是子正则表达式，所匹配的结果也会被记录下来供后面使用。我们暂且不去理会这个特性。

方括号的作用:[]

[abc] 表示a或者b或者c中的任意一个字符。

例：

```
js
var reg = /^[abc]/;
//这个有点像/^(a|b|c)/
str='bbs.zhufengpeixun.cn';
alert(reg.exec(str));;
//返回结果是b。
```

例：

```
js
reg = /^[abc]/;
str='test';
alert(reg.exec(str));
//这次的结果就是null了。
```

我们在字字符集合中使用如下的表示方式: [a-z],[A-Z],[0-9] ，分别表示小写字母，大写字母，数字。例如：

```
js
reg = /^[a-zA-Z][a-zA-Z0-9_]+/;
//后面这个其实这个就是元字符\w表示的含意
str='test';
alert(reg.exec(str));
//结果是整个test，正则的意思是开头必须是英文字母，后面可以是英文字母或者数字以及下划线。
```

关于正则中的连续字符

在正则中/[0-9]/表示匹配字符从0到9中的任意一个，/[a-z]/表示匹配从a到z中的任意一个字母只要是在ASCII码表里连续出现的字符，都可以用这样的表示法。

请百度一下“ASCII码表”，参照里面字符出现的顺序和对应的16进制或10进制的编码。

比如 var reg=/^[!-z]$/; 就会匹配从字符“ ！ ”开始，到字符“ z ”结束的任意一个字符。

```
js
var reg=/^[!-z]$/;
alert(reg.test("8"));//true,
alert(reg.test("*"))//true,
alert(reg.test("}"))//false,因为"}"不在从！到z的这个范围内
```

正则 /^[!-z]$/ 也可以用16进制来表示。如果用16进制表示，则需要用 \u 开头，表示这是在以16进制的方式定义 unicode 字符，并且后边的16进制的编码要写成四位，不足4位的前边补0。则上边的那个正则，也可以写成

```
js
var reg=/^[\u0021-\u007a]$/;
// 字符“！”对应的16进制编码是21，字符“z”的16进制编码是7a。
```

这种方式。中文是扩展的ASCII字符编码，匹配UTF8中文的正则是： /^[\u4e00-\u9fa5]+$/

php中utf-8编码下用正则表达式匹配汉字的正则是： /^[\x{4e00}-\x{9fa5}]+$/u

注意：写成 [1-13] 不是表示从数字1到数字13，而是表示从1到1和3，也就是1和3。因为正则里是在表示连续出现的字符，而不是数字。

反字符集合[^abc]

^ 在正则表达式开始部分的时候表示开头的意思，例如 /^c/ 表示开头是c；但是在字符集和中，它表示的是类似“ 非 “的意思，例如 [^abc] 就表示不能是a，b或者c中的任何一个。例如：

```
js
var reg = /[^abc]/;
var str='blueidea';
alert(reg.exec(str));
```

返回的结果是1，因为它是第一个非abc的字符（即第一个b没有匹配）。同样：

例：

```
js
var reg = /[^abc]/;
var str='cbazhufengpeixun';
alert(reg.exec(str));
```

输出 z ，前三个字符都是 [abc] 集合中的。由此我们可知： [^0-9] 表示非数字， [^a-z] 表示非小写字母，依次类推。

边界与非边界

\b 表示的边界的意思，也就是说，只有字符串的开头和结尾才算数。例如 /\bc/ 就表示字符串开始的c。看下面的例子：

```
js
/\bc/.exec('cainiao');
//返回结果c。匹配到了左边界的c字符。
```

```
js
/\bc/.exec('???c');
//仍然返回c，不过这次返回的是右侧边界的c。
```

```
js
/\bc/.exec('bcb');
//这次匹配失败，因为bcb字符串中的c被夹在中间，既不在左边界也不再右边界。
```

与 \b 对应 \B 表示非边界。例如：

```
js
/\Bc/.exec('bcb');
//这次会成功地匹配到bcb中的c，。然而
```

```
js
/\Bc/.exec('cainiao');
//则会返回null。因为\B告诉正则，只匹配非边界的c。
```

数字与非数字

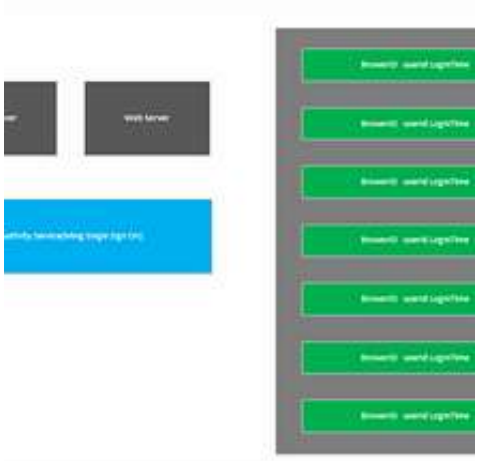
\d 表示数字的意思，相反， \D 表示非数字。

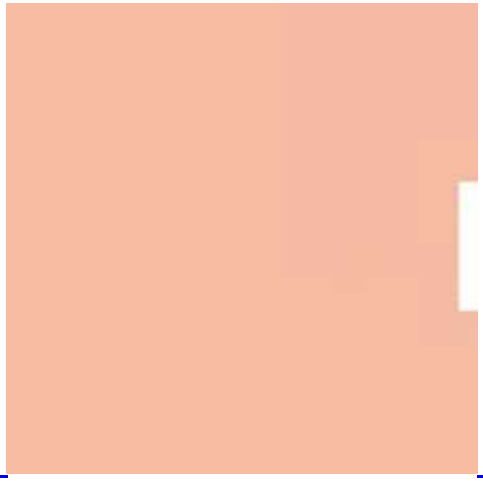
例：

```
js
/\d/.exec('cainiao8')
//返回的匹配结果为8，因为它是第一个数字字符。
```

例：

```
js
/\D/.exec('cainiao8');
//返回c，第一个非数字字符。
```



- 刊主：undefined 《默认推刊》 54

我来评几句

请输入评论内容...

发表评论

已发表评论数(0)

相关站点

segmentfault-博客

已订阅

热门文章

1. Node.js 运行机制探知
2. 如何避免javascript中的冲突
3. 2015年最棒的10个 JavaScript 框架
4. 強迫 Blogger (Blogspot) 使用 blogspot.com 的網域 (而非 .tw)
5. 【前端也要学点算法】 归并排序的JavaScript实现
6. 下面我简单说两句Node.js@v4.0.0

×

用户登录

597455873@qq.com

登 录

收藏到推刊 创建推刊

收 藏 取消

已收藏到推刊！

推刊名(必填) 请填写推刊名

推刊描述

描述不能大于100个字符!

权限设置： 公开 仅自己可见

创建 取消

网站相关

- 关于我们
- 移动应用
- 建议反馈

关注我们

- 推酷网
- tuicool2012
- QQ群:164644910

友情链接

人人都是产品经理 TMTForum 魔部网 PM256 品途网 移动信息化 行晓网 Code4App 智城外包网 LAMP人 安卓航班网 虎嗅 缘创派 IT耳朵 艾瑞网 创媒工场 雷锋网 经理人分享 市场部网 砍柴网 CocoaChina 北风网 云智慧 我赢职场 大数据时代 奇笛网 咕噜网 红联linux Win10之家 鸟哥笔记 爱游戏 投资潮 31会议网 极光推送 Teambition Cocos引擎中文官网 硅谷网 更多链接>>