珠峰培训JavaScript正则表达式无废话教材

在学习本教材之前，应掌握 JS 中基本的字符串处理方法的基本使用。

本教材附视频 9 讲，下载地址：http://pan.baidu.com/s/1hq9kADe

教材最后一页是目录

开宗明义：正则（regular expression）是干什么的？

正则表达式是用来处理字符串的，它的特长在于处理复杂的字符串。

正则表达式定义的是字符串的模型（或叫模式，英文 pattern），我们可以使用这个模型来去验证某个字符串（或字符串里的一部分）是否和这个模型（或叫模式）相匹配，或使用这个模型把某个字符串里和这个模型匹配的那一部分找出来。

再重述一遍：**1**、正则定义了一个字符串的模型。**2**、正则的第一个作用是“验证某字符串是否和这个模型相匹配”。**3**、正则的第二个作用是“把匹配到的内容找出来”。

其实正则只是定义了一个字符串的模型，至于如何去验证字符串和查找字符串，是正则类上的方法完成的。

比如：var reg=/\d+/;写在两个/斜杠之间，是语法规范，表示定义了一个正则对象。 \d 在正则中表示数字，+表示出现一到多次。那这样就定义了一个出现一到多次的数字的模型。它可以和以下字符串匹配：”ab23839cd”,”a4d”、”33928”、”3490cf”、”0938FA”、”z9”，因为这些字符串里，都出现了一到多次的数字，如果用正则类的 test 方法去验证它们，都会返回 true。

例：

```
var reg=/\d+/;        //相当于定义了“一个数字出现一到多次”的模型
var str=”ab23839cd”;
alert(reg.test(str)); //弹出 true
```

这里的 test 方法，是正则类的方法，以字符串为参数，就是负责验证 str 是否符合 reg 定义的模式。下面再定义一个字符串：

```
var str2=”abcdef”;
alert(reg.test(str2));//这次弹出的是 false，因为 str2 里，没有出现数字。
```

以上只是验证字符串，如果想把符合验证的字符串找出来，则就要用其它方法了，例：

```
alert(reg.exec(str));//弹出 23839
alert(reg.exec(str2));//弹出 null，因为 reg 和 str2 不匹配。
```

当然，我也还可以用 String 类的 match 方法来找到和 reg 正则相匹配的内容，例：

```
alert(str.match(reg));//弹出 23839
```

exec 和 match，都是处理字符串功能很强大的方法，后边的章节里会有非常详细的阐述。

如果严格的匹配一到多个数字，不能出现其它字符，应该这样写：

```
var reg=/^\d+$/;
```

这个模型才表示从开头到结尾都是数字。^表示后边出现的数字必须在开头，$表示前面出现的数字必须出现在结尾。像^、$、\d、+这些在正则里表示特殊含义的符号，叫“元字符”。

在正则里，不是只允许出现元字符，普通的字符也是可以出现的。在正则里出现的普通字符，就表示此字符本来的含意。比如：

```
var reg=/^\d8\d$/;    //这个表示匹配一个只包含三位数字，中间是 8 的字符串。详解：^\d 表示以任意数字开头，一个\d 表示出现一次；中间的 8 就表示字符 8 本身，后边的\d$表示一个任意数字结尾。所以这个正则可以匹配以下这些字符串：”282”，”389”，”081”等，但不匹配”1899”，”819”，”08”这样的字符串。
```

在/^\d8\d$/这个正则里，首尾的两个\d 和中间的 8，都表示一个字符，它们是组成正则的最基本单位，我们叫它们“原子”，\d 是原子，8 也是原子。

更多的基础知识，请看下面

一、定义正则

1、创建一个正则的两种方式：

```
var reg=/abcd/;                    //这个叫对象直接量方式
var reg=new RegExp(‘abcd’)//这个叫构造函数方式
```

这两种定义是一样的

2、如果有模式修正符，比如说全文查找 abcd 这个字符串，两种写法分别是(g 是模式修正符，表示在整个字符串里多次查找)

```
var  reg=/abcd/g;
var reg=new RegExp('abcd','g');
```

3、有一种情况要注意，就是如果正则中出现了斜杠“\”（回车上边的斜杠），在用构造函数创建正则对象时，要转义，比如：

```
reg = new RegExp("\\w+")//这里的\要转义
reg = /\w+/                    //这样就不需要
```

这两种定义方式之间有什么区别，请参考在线视频

二、元字符及其含意完整例表

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、或一个后向引用、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。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。
(*pattern*)	匹配 *pattern* 并获取这一匹配。在 JScript 中则使用 **$1…$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 "Windows 2000" 中的"Windows" ，但不能匹配 "Windows 3.1" 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 "Windows 3.1" 中的 "Windows"，但不能匹配 "Windows 2000" 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。例如:[a-z]　　[A-Z]　　[0-9]

[^*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'。
\c*x*	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*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后向引用。如果 *n* 之前至少 *n* 个获取的子表达式，则 *n* 为后向引用。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后向引用。如果 *nm* 之前至少有 is preceded by at least *nm* 个获取得子表达式，则 *nm* 为后向引用。如果 *nm* 之前至少有 *n* 个获取，则 *n* 为一个后跟文字 *m* 的后向引用。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 *nml*。
\u*n*	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 (?)。

正则表达式的exec方法简介

语法：

reg.exec(str);

其中str为要执行正则表达式的目标字符串。例如：

```
<script type="text/javascript">
    var reg = /test/;
    var str = 'abcdtestString';
    var result = reg.exec(str);//result 是个数组,result[0]保存的是正则匹配到的字符串"test"
    alert(result);
</script>
```

将会弹出 test，因为正则表达式 reg 会匹配 str（即字符串：‘abcdtestString’）中的’ test’

的部分，并将'test'保存在一个数组的第一项中。然后把数组返回赋给 result。

这里 exec 的返回值是一个 length 为 1 的数组：["test"]，当 alert(result)的时候，系统会自动执行["test"].toString()方法，所以我们见到的弹出结果是 test。

补充：把上面的正则改一下：

```
<script type="text/javascript">
    var reg = /(t)(e)(s)(t)/;//把 test 的每个字符都用小括号包起来
    var str = 'abcdtestString';
    var result = reg.exec(str);//现在 result 就变成["test","t","e","s","t"]
    alert(result);//弹出  test,,t,e,s,t
</script>
```

在正则中给每个原子加了括号，那就表示在一个大正则里，又出现了若干个子正则，那么 exec 不但要把总正则匹配到的字符找出来，还要把子正则里的字符也要找出来，依次放到数组里。关于子正则，这里先做简单了解即可，这会不必纠缠这个知识点。

exec 是个很强大的方法，下面会有更详细的介绍

三、元字符详解及应用实例部分

如果用正则描述“一片两片三四片，落尽正则全不见”则是{1}，{2}，{3,4}，{1,}。

3.1 c{n} 匹配固定的n个 【量词，从3.1节到3.4节】

{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：

```
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，则表示只去匹配一次，匹配到了则返回并且停止。

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

3.2 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:

reg = /o{3,4}/;　　(匹配三到四个 o)

str='good 珠峰培训'; alert(reg.exec(str));

返回结果 null，表示没有匹配成功。

例： reg = /o{3,4}/; str='goood珠峰培训'; alert(reg.exec(str));

弹出结果是：ooo。

例： reg = /o{3,4}/; str='very gooood珠峰培训'; 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： 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。

3.3 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,相当于元字符+。如下：
例： reg = /c{1,}/;str='cainiao'; alert(reg.exec(str));;结果弹出 c。
例： reg = /c{1,}/; str='cccccTest'; 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： 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0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0次或者1次，等同于{0,1}，即c? 和c{0,1} 是一个意思。

3.4 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：

{n,}?, *?, +?, ??, {m,n}?
例： 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。

3.5 /^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： 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： 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，$则只匹配字符串结尾的字符，同样，看例子：
例3： reg = /z$/; 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： reg = /d$/; str='珠峰培训good'; alert(reg.exec(str));;
这次返回的结果是d，表明匹配成功。

3.6 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
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。

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。

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： 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：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。

3.7 “|”，正则表达式中的或（注意它和[]的区别）

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：reg = /z|o/; str='zhufengpeixun'; alert(reg.exec(str));;结果是 z。
例 2：reg = /z|o/; str='online'; alert(reg.exec(str));; 结果是 o。
例 3：reg = /^z|o.+/; str='online'; alert(reg.exec(str));; 匹配掉整个 online。
例 4：reg = /^z|o.+/; str='zhufengpeixun.cn'; alert(reg.exec(str));
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 或者匹配一个 o 后边连续出现一到多个任意字符。而并非表示以 z 或 o 开头。如果是表示以 z 或 o 开头应该写成：
reg = /^(z|o).+/;

3.8 和括号结合使用

例：reg = /^(z|o).+/;
str='zhufengpeixun';
alert(reg.exec(str));;
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。

3.8 .1 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： 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： 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：

```
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。

```
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

```
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

注意一：写成[1-13]不是表示从数字 1 到数字 13，而是表示从 1 到 1 或 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。
注意二:/[^]/表示所有非空字符串（不是非空格）。
注意三：单字元字符出现在[]中，象.(点)，*（星），+（加号），$，^等，不再表示元字符的含意，而是表示这个字符本身。比如：/[.]/，这个.（点）出现在方括号里已经就不表示任意换行字符了，而是表示.（点）本身。

3.8.2 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：

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

返回的结果是s，因为它是第一个非abc的字符（即第一个b没有匹配）。同样：
例：

```
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。

3.9 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：

```
var reg = /\bc/; var str='cainiao'; alert(reg.exec(str));;
```

返回结果 c。匹配到了左边界的 c 字符。

```
var reg = /\bc/; var str='???c'; alert(reg.exec(str));;
```

仍然返回 c，不过这次返回的是右侧边界的 c。

```
var reg = /\bc/; var str='bcb'; alert(reg.exec(str));
```

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：

```
var reg = /\Bc/; var str='bcb'; alert(reg.exec(str));
```

这次会成功地匹配到 bcb 中的 c，。然而

```
var reg = /\Bc/; var str='cainiao'; alert(reg.exec(str));
```

则会返回 null。因为\B 告诉正则，只匹配非边界的 c。

3.10 数字与非数字

\d表示数字的意思，相反，\D表示非数字。

例：

```
var reg = /\d/;
var str='cainiao8';
alert(reg.exec(str));
```

返回的匹配结果为 8，因为它是第一个数字字符。

例：`reg = /\D/; str='cainiao8';alert( reg.exec(str));`

返回c，第一个非数字字符。

3.11 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：

```
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：

```
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：

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

3.12 单字符\w

\w表示单词字符，等同于字符集合[a-zA-Z0-9_]。例如：

```
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。

```
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。

```
var reg = /\w+/;
var str='珠峰培训';
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_]

```
var reg = /\W+/;
var str='珠峰培训';
alert(reg.exec(str));
```

返回完整的字符串，因为，中文算作是非单词字符。

3.13 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.

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

返回z。

2.

```
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.

```
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”。

```
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：

```
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：

```
var reg = /^(b|c).+/;
var str='bbs.zhufengpeixun.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。

3.14 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：

```
var reg = /^(?:b|c).+/;
var str='bbs.zhufengpeixun.cn';
 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：

```
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。

```
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。

3.15 正向预查和负向预查

(?=pattern)正向预查

形式：(?=pattern) 所谓正向预查，意思就是：要匹配的字符串，必须满足pattern这个条件！ 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：

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

返回null。

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

匹配cainiao。需要注意的是，括号里的内容只是条件，并不参与真正的捕获，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：

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

匹配到zhufeng，而不是peixun。

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

返回null，因为zhufeng后面不是peixun。

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

同样返回null。

（?!条件） 负向预查

形式(?!pattern)和（?=pattern）恰好相反，要求做匹配的时候，必须不满足pattern这个条件，还拿上面的例子：

```
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。

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

则成功返回zhufeng。

看下面这个正则，你能理解reg1和reg2这两个正则表达了相同的意思吗？

```
var reg1=/(?=^)\d{2}(?=$)/;//用正向预查，左边要满足是开头，右边要满足是结尾
var reg2=/^\d{2}$/;
```

3.16 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

```
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：

```
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。

3.17 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：

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

返回null，因为大小写不符合。

```
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修饰符的作用是写了^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：

```
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：

```
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。

四、和正则相关的方法详解

exec方法详解 （重点和难点）

1、exec方法的返回值

exec方法的返回值是一个数组

```
var reg = /b/;
var str='bbs.zhufengpeixun.cn';
var result = reg.exec(str);
```

我们使用for in循环来遍历一下这个数组有那些额外属性:

```
for(var attr in result){
```

```
    console.log("reslut的属性: "+attr+", 此属性的值: "+result[attr])
}
```

控制台中输出的结果是（写在【】里的是解释）

```
reslut的属性: 0, 此属性的值: b  【表示匹配到的结果存在索引0这个位置，即result[0]是“b“】
reslut的属性: index, 此属性的值: 0 【这个属性表示”b”在原字符串中的起始位置】
reslut的属性: input, 此属性的值: bbs.zhufengpeixun.cn  【这个属性表示原字符串】
```

还有一个不可枚举的属性length没有被输出到控制台。如果正则中还有分组，则length的值就不是1了。比如：

```
var reg = /(\w)(\w)(\w)/    //不加模式修正符g。
var str='bbs.zhufengpeixun.cn';
var result=reg.exec(str);//exec执行两次，是为了配合本章第2小节的讲解
var result=reg.exec(str);//故意执行两次，如果没加修正符g，执行多少次，返回的结果都
                         一样
for(var attr in result){
        console.log("reslut的属性: "+attr+", 此属性的值: "+result[attr])
}
```

结果为：

```
reslut的属性: 0, 此属性的值: bbs   【整个正则匹配到的内容】
reslut的属性: 1, 此属性的值: b     【第一个括号里的子正则匹配到的内容】
reslut的属性: 2, 此属性的值: b     【第二个括号里的子正则匹配到的内容】
reslut的属性: 3, 此属性的值: s     【第三个括号里的子正则匹配到的内容】
reslut的属性: index, 此属性的值: 0 【被匹配到的内容，出现在原字符串str中的位置】
reslut的属性: input, 此属性的值: bbs.zhufengpeixun.cn   【原字符串str】
```

现在result.length的值是4，表示有四个匹配项被保存在result中，result[0]就是整个正则表达式所匹配的内容。后续的result[1]、result[2],result[3]则是各个子正则表达式(分组)的匹配内容。

2、exec方法对正则表达式的更新

exec方法在返回结果对象的同时，还可能会更新原来的正则（注意：是把正则对象给更新了），这就要看正则表达式是否设置了g修饰符。先来看两个例子吧：

```
var reg = /(\w)(\w)(\w)/g;  //这里加了模式修正符g，表示要在全文内多次匹配查找
var str='bbs.zhufengpeixun.cn';
reg.exec(str); //exec第一次运行
var result=reg.exec(str); //注意这里：result是exec是第二次运行的返回值
for(var attr in result){
        console.log("reslut的属性: "+attr+", 此属性的值: "+result[attr])
}
```

输出的结果为：

```
reslut的属性: 0, 此属性的值: zhu      【字符.（点）与\w这个原字符的描述不能匹配，所以是zhu】
reslut的属性: 1, 此属性的值: z
reslut的属性: 2, 此属性的值: h
reslut的属性: 3, 此属性的值: u
```

reslut的属性：index，此属性的值：4　　**【表示’zhu’出现在str中的索引位置是4】**
reslut的属性：input，此属性的值：bbs.zhufengpeixun.cn

可以看得出来，第二次执行exec方法还能继续匹配并查找，同样还可以进行第三次第四次的匹配和查找。这也就是g修饰符的作用了。如果是多次查找，那如何知道下一次从那个位置开始的呢？这就是正则（这个属性是正则对象的）的一个很重要的属性在发挥作用了，它叫:**lastIndex。**

每个正则实例上都会有一个叫 lastIndex 的属性，它的作用是规定当前这次的匹配的开始位置是从那儿开始的。如果正则表达式没有设置模式修正符 g，那么 lastIndex 的值永远是 0，则表示无论这个正则被使用过多少次，每次都是从字符串 0 的位置去匹配。所以第 1 小节中的 exec 方法虽然也是执行了两次，但对返回的值没有任何的影响，因为每次执行 exec，都是从头开始的。如果设置了 g，那么 exec 执行之后会更新正则表达式的 lastIndex 属性，表示本次匹配后，所匹配字符串的下一个字符的索引，下一次再用这个正则表达式匹配字符串的时候就会从上次的 lastIndex 属性开始匹配，也就是上面两个例子结果不同的原因了。

一定要注意的是，lastIndex 属性是正则对象的属性；而 index 属性和 input 属性是 exec 方法返回的那个数组的属性。

特别强调：两点，一、即使正则设置了g修饰符，exec方法不会自动的进行全文查找,但会修改正则对象的lastIndex的值。二、但它的特长在于不但可以捕捉到整个正则匹配的内容，还可以捕捉到子正则（**分组**）匹配到的内容，如果想把字符串所有的匹配项和子匹配项都取到（就是把总正则和子正则的匹配项都取到），那需要自定义下面一个**这样的方法：**

```
RegExp.prototype.autoExec=function (str){//定义在正则类上
    //this是指当前执行autoExec方法的正则实例
    if(this.global){//必须设置修正符g
            this.lastIndex=0;//把lastIndex的值修正为0，以免reg被使用过
            var a=[];//准备一个数组用来保存每一次捕获到的结果。
            var result=null;
            /*
            //第一方法：用while循环。
            while(result=this.exec(str)){//先执行exec(str)，然后把返回值赋给result，最后判断result是否为null
                    a.push(result);
            }
            */
            do{//第二种方法用了do-while循环，并且判断的是lastIndex。你能理解吗？
                    a.push(this.exec(str))
            }while(this.lastIndex);
            return a;//如果有捕获结果，则会返回一个二维数组
    }else{
            throw new Error("未设置修正符g");
    }
}//要求：在str字符串中，把所有的完整的时间字符串和年月日时分秒都提取出来。
var str="times is 2013-11-2 12:03:36 ; times is 1998-10-12 3:03:36;times is 2012-11-03 12:22:34";
var reg=/(\d{4})-(\d{1,2})-(\d{1,2})
+(\d|[01]\d|[2][0-3]):(\d|[0-5]\d):([0-5]\d|\d)/g;
var r=reg.autoExec(str);
alert(r)
```

```
//结果是这样的一个二维数组：
[
        ["2013-11-2 12:03:36","2013","11","2","12","03","36"],
        ["1998-10-12 3:03:36","1998","10","12","3","03","36"],
        ["2012-11-03 12:22:34","2012","11","03","12","22","34"]
]
```

test方法

test方法仅仅检查是否能够匹配str，并且返回布尔值以表示是否成功。

实例1

```
var reg = /b/;
var str = 'bbs.zhufengpeixun.cn’ ;
alert(reg.test(str));
```

成功，输出true。

实例2

```
var reg = /9/;
var str = 'www.zhufengpeixun.cn';
alert(reg.test(str));
```

失败，输出false。

使用字符串的方法执行正则表达式

match方法

注意：这是一个字符串方法，参数是正则。

形式：str.match(reg);

与正则表达式的exec方法类似，该方法同样返回一个数组，返回值上也有input和index属性。我们定义如下一个函数用来测试：

```
function matchReg(reg,str){
 var result = str.match(reg);
if(result ){
    console.log ('index:'+result.index+'\n' +'input:'+result.input);
    for(i=0;i<result.length;i++){
      console.log('result['+i+']:'+result[i])
    }
}else{ console.log ('没有匹配到结果')}
}
```

例如：

```
var reg = /[bz]/;//表示匹配b或z
var str = 'bbs.zhufengpeixun.cn';
matchReg(reg,str);
```

结果如下：

```
index:0
input:bbs. zhufengpeixun.cn
result[0]:b ;
```

这和 exec 的结果一样。但是如果正则表达式设置了 g 修饰符，即：var reg=/[bz]/g，exec 和 match 的行为可就不一样了，输出结果为：

```
index:undefined
input:undefined
result[0]:b
result[1]:b
result[2]:z
```

match 方法设置了 g 修饰符的正则表达式在完成一次成功匹配后不会停止，而是继续找到所有可以匹配到的字符。返回的结果包括了三个 b。不过没有提供 input 和 index 这些信息。

match方法和exec方法都是处理字符串功能比较强大的方法，这里有必要详细的把这两个方法的相同与不同说一说。这两个方法都是用来提取正则匹配到的内容，**exec处理分组功能更强**。match在没有分组的情况下，能够**更快捷**的把多次匹配到的内容保存到数组里。exec是属于正则类的方法，match是字符串类的方法。

replace 方法详解此方法比较强大，有视频两讲专门讲这一方法

定义和用法：
replace() 方法用于在字符串中用一些字符替换另一些字符，或替换一个与正则表达式匹配的子串。
语法：
str.replace(regexp/substr,replacement)

参数	描述
regexp/substr	参数一：必需。被替换的子字符串或要替换子字符串的正则的模式（RegExp 对象）就是说第一参数是被替换的内容。
replacement	参数二：必需。它可以是字符串，也可以是一个函数。用来替换第一个参数匹配到的内容。

返回值

一个新的字符串，是用 *replacement* 替换了 regexp 的第一次匹配或所有匹配之后得到的。

说明

字符串 str 的 replace() 方法执行的是查找并替换的操作。它将在 str 中查找与 regexp 相匹配的子字符串，然后用 *replacement* 来替换这些子串。如果 regexp 具有全局标志 g，那么 replace() 方法将替换所有匹配的子串。否则，它只替换第一个匹配子串。

replacement 可以是字符串，也可以是函数。如果它是字符串，那么将匹配的内容由此字符串替换。但是 replacement 中的 $ 字符具有特定的含义。如下表所示，它说明从模式匹配得到的字符串将用于替换。

字符	替换文本
$1、$2、...、$99	与 regexp 中的第 1 到第 99 个子表达式（分组）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。
$$	直接量符号。

注意：当 replace() 方法的第二个参数 replacement 是函数而不是字符串时，每次匹配都调用该函数，将这个函数的返回的字符串将作为替换文本使用。这个函数是自定义的替换规则。

当第二个参数是函数时，不仅自动此函数，还要给这个函数传最少三个参数：

1、当正则没有分组的时候，传进去的第一个实参是正则捕获到的内容，第二个参数是捕获到的内容在原字符串中的索引位置，第三个参数是原字符串（输入字符串）

2、当正则有分组的时候，第一个参数是总正则查找到的内容，后面依次是各个子正则查找到的内容。

2、传完查找到的内容之后，再把总正则查找到的内容在原字符串中的索引传进（就是 arguments[0]在 str 中的索引位置）。最后把输入字符串（就是原字符串）传进去

看下面两个示例的说明：(学到 20 页之后再看此示例比较好)

示例 1：

要求：找到字符串中的小写字母，给它在后边加上小括号来注明它在 str 字符串中的位置。比如 str 中的第一个 a，它出现在 str 字符串的第一个索引位置中，则 a 变成 a(1)。下面的 str 最终得到的结果是 Xa(1)ZZc(4)Ud(6)Fe(8)

```
var str="XaZZcUdFe";

var reg=/[a-z]/g;//注意：全文替换必须加 g

str=str.replace(reg,function(){

        return arguments[0]+"("+arguments[1]+")";

        //arguments.length 的值是 3，在 reg 没有分组的情况下 length 属性肯定是 3.

        //其中 arguments[0]是正则捕获查找到的内容；arguments[1]是正则查找到的内容在 str 这个字符串中的索引位置；arguments[2]是 str 字符串本身（叫输入字符串）

        //这个匿名函数被自动执行四次，每一次 arguments 里的值分别是：

        //第一次：arguments[0]是 a，arguments[1]是 1，arguments[2]是原字符 str 本身

        //第二次：arguments[0]是 c，arguments[1]是 4，arguments[2]是原字符 str 本身

        //第三次：arguments[0]是，arguments[1]是 6，arguments[2]是原字符 str 本身

        //第四次：arguments[0]是 e，arguments[1]是 8，arguments[2]是原字符 str 本身


})

alert(str);//弹出 Xa(1)ZZc(4)Ud(6)Fe(8)
```

示例 2：

要求：找出下面字符串中两个连着出现的数字，用它们的和将它们替换。

比如第一次找到 4 和 5，用 9 替换，第二次找到的是 8 和 9，用 17 替换，第三次找到的是 7 和 2，用 9 替换，最终生成的字符串是 96a17b9cs

```
var str="456a89b72cs";

var reg=/(\d)(\d)/g;

str=str.replace(reg,function(){

    return Number(arguments[1])+Number(arguments[2]);

    });

alert(str);//弹出 96a17b9cs

    //上面 replace 里的匿名函数会被自动执行三次（因为匹配到了三次）；

    //每次执行，arguments.length 都是 5;arments[0]是总正则查找到的字符串，arguments[1]是第一个分组查找到的内容，arguments[2]是第二个分组查找到的内容，arguments[3]是总正则查找到的内容在 str 这个字符串中的索引位置，arguments[4]是 str 这个字符串本般

    //第一次这五个参数的值分别是：arguments[0]是"45",arguments[1]是"4",arguments[2]是"5",arguments[3]是 0，arguments[4]是"456a89b72cs"；

    //第二次这五个参数的值分别是：arguments[0]是"89",arguments[1]是"8",arguments[2]是"9",arguments[3]是 4，arguments[4]是"456a89b72cs"；

    //第三次这五个参数的值分别是：arguments[0]是"45",arguments[1]是"4",arguments[2]是"5",arguments[3]是 0，arguments[4]是"456a89b72cs"
```

一、基本示例：

例 1：

```
var str = 'bbs.zhufengpeixun.cn';

var str2=str.replace('b','c');//用字符串做第一个参数，把 str 中的第一个 b 用 c 替换掉

alert(str2);//弹出 cbs.zhufengpeixun.cn ，第二个 b 并没有被替换
```

注意：replace方法不会修改字符串本身，而是把修改之后的结果返回了。如上例中：str本身没有变化，str2是被修改替换之后的结果

例 2：

```
var reg=/b/;

    var str3=str.replace(reg,'c');////用正则做第一个参数，把 str 中的第一个 b 用 c 替换掉

    alert(str3)//结果同上，输出 cbs.zhufengpeixun.cn
```

例 3：

```
var reg=/b/g;//修改一下正则，加上模式修正符 g，表示全文匹配
```

```
var str4=str.replace(reg,'c');
alert(str4);//输出 ccs.zhufengpeixun.cn
```

例 4：

```
var reg = /\w+/g;
var str = 'www.zhufengpeixun.cn';
var newStr = str.replace(reg,'word');
console.log(newStr);//输出：word.word.word
```

二、使用$的示例

就像在正则里我们可以使用\1来引用第一个子正则表达式所匹配的内容一样，我们在replace函数的替换字符里也可以使用$1来引用相同的内容。还是来看一个例子吧：

例1：

```
var reg = /(\w+).(\w+).(\w+)/;
var str = online.zhufengpeixun.cn';
var newStr = str.replace(reg,'$2.$2.$2');  //注意：这几个$是在引号里的
document.write(newStr);
```

输出的结果为：zhufengpeixun. zhufengpeixun. zhufengpeixun首先，我们知道第二个子正则表达式匹配到了zhufengpeixun，那么$2也就代表zhufengpeixun了。其后我们把替换字符串设置为'$2.$2.$2', 其实也就是“zhufengpeixun. zhufengpeixun. zhufengpeixun”。同理，$2就是zhufengpeixun，$3就是cn。

注：还有一种保存正则里分组的方式，就是RegExp.$1,RegExp.$2.....RegExp.$9,详见视频中的讲解

在来看一个例子，颠倒空格前后两个单词的顺序。

例2：

```
var reg = /(\w+)\s+(\w+)/;
var str = ‘zhufeng    peixun;
var newStr = str.replace(reg,'$2 $1');
document.write(newStr);
```

结果为：peixun zhufeng，也就是空格前后的单词被调换顺序了。

由于在替换文本里$有了特殊的含义，所以我们如果想要是用$这个字符的话，需要写成$$。

例3：

```
var reg = /(\w+)\s(\w+)/;
var str = 'cainiao gaoshou';
var newStr = str.replace(reg,'$$ $$');
document.write(newStr);
```

结果为：$ $。

三、在replace中，用函数做为第二个参数

例1：

```
var reg=/[a-z]/g;//定义一个只匹配小写字母的正则
var str='a1b2c3d4';
var str=str.replace(reg,function(result,position,string){
var str='在'+position+'的位置匹配到了'+result+',原字符串是'+string;document.write(str+'<br\>');
    return position;
});
document.write(str);
```

在0的位置匹配到了a,原字符串是a1b2c3d4//第一次匹配时运行方法的输出结果
在2的位置匹配到了b,原字符串是a1b2c3d4//第二次匹配时运行方法的输出结果
在4的位置匹配到了c,原字符串是a1b2c3d4//第三次匹配时运行方法的输出结果
在6的位置匹配到了d,原字符串是a1b2c3d4//第四次匹配时运行方法的输出结果

01224364//最后生成的字符串的输出结果

说明一：reg这个正则，在'a1b2c3d4'这个字符串中会被匹配四次，每匹配一次，replace中的第二个参数（函数）就会运行一次，把这个函数的返回值做为替换字符串，替换掉匹配到的字符串。
说明二：replace这个方法在运行的时候，会自动的给第二个参数（就是这个方法）传三个参数，第一个参数是匹配到的字符串。第二个参数是一个整数，是正则匹配到的内容在原字符串中的位置。第三个参数是原字符串本身。**【前提是正则中没有分组】**

例2：请使用正则表达式，来将如下的字符串中的占位符都换成数组中下标对应的的内容。
字符串为：My name is{0}.I am{1} years old.I am in class{2} grade{3}.
给定的数组为：var a=["珠峰培训",38,4,6];
替换完成之后正确结果为：My name is 珠峰培训.I am 38 years old.I am in class 4 grade 6 .
这题的作法很多最简捷的是使用正则中的分组，如下：

```
var reg=/{(\d)}/g;
var newStr2=str.replace(reg,function(s,i){//参数s没有实际意义，只是占个位置
        return  a[i];//第一个参数没有用。
        //可以直接用 return a[arguments[1]];也可以
        /*
        如果reg这个正则中没有分组，i是当前匹配到的字符串出现在整个字符串里的位置。
        如果正则有分组，此函数的参数：从第二个参数开始，表示的是各个分组，分组表示完了之后，再是匹配到的字符串出现的位置，最后的参数是表示原字字符串本身
        */
        /*      return a[RegExp.$1];//这是第二种思路，不需要参数，而是使用正则类的上分组功能实现。这个是最简捷的方法。          */
        });
alert(newStr2);
```

重点综合示例：给数字加上千分符

比如：把12345修改成12，345，把1234567修改成1，234，567

```
var str="39736484599";//修改成39,736,484,599
```

第一种方案：

```
reg=/^(\d{1,3})((\d{3})+)$/;//这个正则的作用是把字符串分隔成两部分，前一部分是一到三位的数字，后边的数字个数是3的倍数。分解完成之后再进行处理。
var newStr=str.replace(reg,function(){
        var part1=arguments[1];
        var part2=arguments[2];
        //part1是39
        //part2是736484599，对part2进行二次处理，每三位数字加一个逗号，但最后一组不加（就是正则(?!$)的作用，它是负向预查，表示不能是右边界，即不给出现在右边界的字符串加逗号）
        return part1+","+part2.replace(/\d{3}(?!$)/g,function(){
                return arguments[0]+",";//每位加一个逗号
        })
});
alert(newStr);
```

第二种方案：

```
这种方案正则简单，自定义替换规则函数里的逻辑稍稍复杂一点点
var reg=/\d(?!$)/g;//这个正则其实也就是起个数数的作用（记数）
newStr=str.replace(reg,function(){
```

```
            if((arguments[2].length-arguments[1]-1)%3==0){
                //if里的条件相当于在倒着数当前这个数是倒数第几个，如果数到的这个数是3的倍数，则加一个逗号
                return arguments[0]+",";//加上一个逗号返回
            }else{
                return arguments[0];//把原来的字符再返回，一定要返回
            }
            //arguments[0]是正则捕获到的结果，arguments[1]是此结果在输入字符串中的位置，arguments[2]是指输入字符串
        })

        alert(newStr);
```

第三种方案

```
var str="39736484599";
//这是最直接的一种思路，就是先反转，再处理，最后再反转回来
var strTemp1=str.split("").reverse().join("");//反转
var strTemp2=strTemp1.replace(/(\d{3}(?!$))/g,"$1,");//按从左向右的顺序加逗号
//(?!$)负向预测：表示连续出现的三个字符的右边不能是最右边儿（不能是结尾）

var strTemp3=strTemp2.split("").reverse().join("");//再反转
alert(strTemp3);
```

search方法和split方法

同样，字符串的search方法和split方法中也可以使用正则表达式，形式如下：

str.search(reg);

search返回正则表达式第一次匹配的位置。例子：

```
        var reg = /peixun/;
        var str = 'zhufengpeixun';
        var pos = str.search(reg);
        alert(pos);
```

结果为7。

下面的例子找出第一个非单词字符：

```
        var reg = /\W/;
        var str = online.zhufengpeixun. cn';
        var pos = str.search(reg);
        alert(pos);
```

结果为6，也就是那个点“.”的位置。

str.split(reg,’seprator’); split返回分割后的数组，例如：

```
        var reg = /\W/;
        var str = 'online.zhufengpeixun.cn';
         var arr = str.split(reg);
         alert(arr);
```

结果为：online,zhufengpeixun,cn，可见数组被非单词字符分为了有三个元素的数组。

```
        var reg = /\W/;
        var str = 'http://www.baidu.com/';
        var arr = str.split(reg);
        alert(arr.length+'<br />');
        alert(arr);
```

分别弹出：7和http,,,www,baidu,com,
可见字符串被分为了有7个元素的数组，其中包括了三个为空字符串的元素。

附：一些正则的练习题

1. 定义一个字符串
   var str1="2012-04-06 12:30:30"
   要求：将字符串中的年月日时分秒取出来，存到数组里，请用尽可能多的方法实现
   解析：
   思路：
   方法一：写个正则表达式，对 str1 中的年月日时分秒进行分组，用正则表达式的 test 方法对 str1 进行测试，如果返回 true，各捕获的各分组会保存在全局的 RegExp 对象的 $1-$6 中，将 RegExp.$1-RegExp.$6 保存在数组中即可，代码如下：

```
var str1="2012-04-06 12:30:30";
var reg=/^(\d{4})-(\d{2})-(\d{2}) (\d{2}):(\d{2}):(\d{2})$/;
var arr=[];
if(reg.test(str1)){
        for(var i=1;i<=6;i++){
                arr.push(eval('(RegExp.$'+i+')'));
        }
}
```

   循环部分也可以这样写

```
arr.push(RegExp.$1);
arr.push(RegExp.$2);
arr.push(RegExp.$3);
arr.push(RegExp.$4);
arr.push(RegExp.$5);
arr.push(RegExp.$6);
```

   方法二：观察这个字符串，年月日时分秒的数字由短线”-”、冒号”:”或空格” ”分隔，可以使用 String 的 split 方法

```
var str1="2012-04-06 12:30:30";
var reg=/[-:\s]/;
var arr=str1.split(reg);
alert(arr);
```

   方法三：还可以使用正则表达式的 exec 方法来实现

```
var str1="2012-04-06 12:30:30";
var reg=/\d+/g;
var arr=[];
while((result= reg.exec(str1))!=null){
arr.push(result);
}
alert(arr);
```

   方法四：用 String 的 replace 方法实现

```
var str1="2012-04-06 12:30:30";
var reg=/\d+/g;
var arr=[];
str1.replace(reg,function(match){
        arr.push(match);
});
alert(arr);
```

   方法五：

```
var str1="2012-04-06 12:30:30";
```

```
var arr=str1.match(/\d+/g);
alert(arr);
```

2. 定义一个字符串

var str1="2012-04-06 12:30:30"

要求：将其转换为这种格式：

2012 年 04 月 06 日 12 时 30 分 30 秒

解析：

方法一：

```
var str1="2012-04-06 12:30:30";
var reg=/^(\d{4})-(\d{2})-(\d{2}) (\d{2}):(\d{2}):(\d{2})$/;
var str2=str1.replace(reg,'$1 年$2 月$3 日$4 时$5 分$6 秒');
alert(str2);
```

3. 写出同时包含 abc 三个字母(三个字母各出现一次)，但不考虑顺序的正则表达式

http://topic.csdn.net/u/20090525/10/2be5992f-70e5-4685-a943-e6ff3d6f26ac.html

方法一：

```
var str1="abcd";
var reg=/^[^abc]*([abc])[^abc]*((?!\1)[abc])[^abc]*((?!\1)(?!\2)[abc])[^abc]*$/;
alert(reg.test(str1));
```

4. var s = "[content=\"Tech\"]内容 1[/content]其他内容[content=\"Sales\"]内容 2[/content]";

请使用正则取出：Tech,Sales,内容 1,内容 2

要求最好返回值是一个二维数组：如：a[0][0] = 'Tech';a[0][1] = 'Sales';

http://www.mianwww.com/html/2009/08/3675.html

方法一：

```
var s = "[content=\"Tech\"]内容 1[/content]其他内容[content=\"Sales\"]内容 2[/content]";
var t = s.match(/\[content[^\]]+].+?\[\/content]/ig);
var r = [];
for (var i = 0; i < t.length; i ++) {
r.push([/\[content="([^"]+)"]/.exec(t[i])[1],/\[content[^\]]+](.+?)\[\/content]/.exec(t[i])[1]]);
}
alert(r);
```

方法二：

```
var s = "[content=\"Tech\"]内容 1[/content]其他内容[content=\"Sales\"]内容 2[/content]";
var arr=s.match(/(([^"]+)(?="]))|([^\[\]]+(?=\[\/))/g)
alert(arr);
```

5. 请使用正则表达式，来将如下的字符串中的占位符都换成数组中的内容，字符串 str 为：My name is{0}.I am{1} years old.I am in class{2} grade{3}. 给定的数组为：

```
var arrayx=new Array();
arrayx[0]='lilei';
arrayx[1]=13;
arrayx[2]=4;
arrayx[3]=6;
```

方法一：

```
var reg=/{\d}/g;//正则，用来匹配字符串里的花括号和花括号里的数字
var i=0;
var str2=str.replace(reg,function(){
                return ' '+ arrayx[i++]
});
alert(str2);
```

方法二：

```
var reg=/{(\d)}/g;//正则，用来匹配字符串里的花括号和花括号里的数字
var str2=str.replace(reg,function(m,n){
        return ' '+ arrayx[n]
});
alert(str2);
```

方法三：

```
for(var i=0;i<arrayx.length;i++){
    str = str.replace('{'+i+'}'," "+arrayx[i]);
}
alert(str);
```

6. 请使用正则表达式，来将如下的字符串中的占位符都换成 info 对象中的内容，字符串 str1 为：My name is {name}.I am {age} years old.I am in class {class} grade {grade}，info 对象为：info={name:'lilei',age:13,class:4,grade:6};

方法一：

```
var reg=/\{([^{}]*)\}/g;
str1=str1.replace(reg,function(a){
            var a = a.replace(/[{}]/g,'');
            return info[a];
});
alert(str1);
```

方法二：

```
var reg=/\{(\w+)\}/g;
str1=str1.replace(reg,function(){
            return info[RegExp.$1];
});
alert(str1);
```

7. 怎样把一个字符串中的连续空格替换为一个

例如：var str = “zhufeng peixun class name”;

```
reg = /|s+/g;
str1 = str.replace(reg,' ');
alert(str1);
```

8. 如何删除空行

例如： var str = ‘zhufeng peixun 培训 周末班和 脱产班’;

```
var reg = /\s+/g;
str1 = str.replace(reg,'');
alert(str1);
```

9. 请编写一个 JavaScript 函数 parseQueryStr，它的用途是把 URL 参数解析为一个对象：用法如：var obj = parseQueryString(url);例如： www.zhufengpeixun.cn?course1=js&course=css; 则 obj 的值为 {course1:js,course:css }

```
function urlToObj(str){
    var tempa=null//存放临时匹配到的字符串的那个临时数组

    //定义一个取每一对值的正则，把满足要求的内容分别定义成两个分组。匹配到的内容不到包括=?&这三个字符既可
    var reg=/([^=?&]+)=([^=?&]+)/g;
    var obj={}
    while( tempa=reg.exec(str)){//把 exec 的返回值赋给这个 tempa,如果 tempa 不是 null，则 exec 会执行多次。
    //tempa 是一个数组，这个数组的长度是 reg 中匹配到的子表达式（分组）的个数加 1
    //tempa 的第 0 项是整个正则匹配到的内容，所以从索引 1 开始；
            obj[tempa[1]]=tempa[2];
            }
    return obj;
}
```

珠峰培训 JavaScript 正则表达式无废话教材

目　　录