JavaScript正则表达式

菜鸟(http://www.cainiao8.com/)
邮箱:runningtortoise@hotmail.com

创建一个正则表达式

第一种方法：
var reg = /pattern/;
第二种方法：
var reg = new RegExp('pattern');

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 = 'testString';
var result = reg.exec(str);
alert(result);
</script>
将会输出test，因为正则表达式reg会匹配str('testString')中的’test’子字符串，并且将其返回。

我们使用下面的函数来做匹配正则的练习：
function execReg(reg,str){
var result = reg.exec(str);
alert(result);
}
函数接受一个正则表达式参数reg和一个目标字符串参数str，执行之后会alert出正则表达式与字符串的匹配结果。
用这个函数测试上面的例子就是：
<script type="text/javascript">
function execReg(reg,str){
var result = reg.exec(str);
alert(result);
}
var reg = /test/;
var str = 'testString';
execReg(reg,str);
</script>
上面的例子用正则里的test去匹配字符串里的test，实在是很无聊，同样的任务用indexOf方法就可以完成了。用正则，自然是要完成更强大的功能：

一片两片三四片，落尽正则全不见

上面的小标题翻译成正则就是{1},{2},{3,4},{1,}。

c{n}

{1}表示一个的意思。
/c{1}/只能匹配一个c。
/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：
reg = /c{1}/;
str='cainiao';
execReg(reg,str);
返回结果c

reg = /c{2}/;
str='cainiao';
execReg(reg,str);
返回结果null，表示没有匹配成功。

reg = /c{2}/;
str='ccVC???';
execReg(reg,str);
返回结果cc。

c{m,n}

c{3,4}的意思是，连续的3个c或者4个c。
例如
reg = /c{3,4}/;
str='ccVC???';
execReg(reg,str);
返回结果null，表示没有匹配成功。

reg = /c{3,4}/;
str='cccTest';
execReg(reg,str);
结果返回ccc。

reg = /c{3,4}/;
str='ccccTest';
execReg(reg,str);
结果返回cccc，这表明正则会尽量多品牌，可3可4的时候它会选择多匹配一个。

reg = /c{3,4}/;

str='cccccTest';
execReg(reg,str);
仍然只匹配4个c。

由以上例子可以推断出，c{m,n}表示m个到n个c，且m小于等于n。

c{n,}

c{1,}表示1个以上的c。例如：

reg = /c{1,}/;
str='cainiao';
execReg(reg,str);
结果返回c。

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

reg = /c{2,}/;
str='cainiao';
execReg(reg,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。

***,+,?**

*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。

贪心与非贪心

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?。组成如下的形式：
{n,}?, *?, +?, ??, {m,n}?

同样来看一个例子：
reg = /c{1,}?/;
str='ccccc';
execReg(reg,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$/

^表示只匹配字符串的开头。看下面的例子：

reg = /^c/;
str='???c';
execReg(reg,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。

reg = /^c/;
str='cainiao';
execReg(reg,str);
这次则返回c，匹配成功，因为cainiao恰恰是以c开头的。

与^相反，$则只匹配字符串结尾的字符，同样，看例子：

reg = /c$/;
str='cainiao';
execReg(reg,str);
返回null，表示正则表达式没能在字符串的结尾找到c这个字符。

reg = /c$/;
str='???c';
execReg(reg,str);
这次返回的结果是c，表明匹配成功。

点’.’

‘.’会匹配字符串中除了换行符\n之外的所有字符，例如
reg = /./;
str='cainiao';
execReg(reg,str);
结果显示，正则匹配到了字符c。

reg = /./;
str='blueidea';
execReg(reg,str);
这次是b。

reg = /.+/;
str='blueidea——???? ?_?';
execReg(reg,str);
结果是“blueidea——经典论坛 好_。“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，和一个破折号。

reg = /.+/;
reg = /.+/;
str='bbs.blueidea.com';
execReg(reg,str);
同样，直接返回整个字符串——bbs.blueidea.com，可见”.”也匹配”.”本身。

reg = /^./;
str='\ncainiao';
execReg(reg,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。

二选一，正则表达式中的或，“|“

b|c表示，匹配b或者c。
例如：
reg = /b|c/;
str='blueidea';
execReg(reg,str);
结果是b。

reg = /b|c/;
str='cainiao';
execReg(reg,str);
结果是c。

reg = /^b|c.+/;
str='cainiao';
execReg(reg,str);
匹配掉整个cainiao。

reg = /^b|c.+/;
str='bbs.blueidea.com';
execReg(reg,str);
结果只有一个b，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b或者是c.+。

括号

reg = /^(b|c).+/;
str='bbs.blueidea.com';
execReg(reg,str);
这次的结果是整个串bbs.blueidea.com，机上上面的括号这后，这个正则的意思是，如果字符串的开头是b或者c，那么匹配开头的b或者c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“,b“，这是()内的b|c所匹配的内容。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。

字符集合**[abc]**

[abc]表示a或者b或者c中的任意一个字符。例如：
reg = /^[abc]/;
str='bbs.blueidea.com';
execReg(reg,str);
返回结果是b。

reg = /^[abc]/;
str='test';
execReg(reg,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_]+/;
str='test';
execReg(reg,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。

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：
reg = /[^abc]/;
str='blueidea';
execReg(reg,str);
返回的结果是l，因为它是第一个非abc的字符（即第一个b没有匹配）。同样：
reg = /[^abc]/;
str='cainiao';
execReg(reg,str);

则返回i，前两个字符都是[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，一次类推。

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或者是结尾的c。看下面的例子：

reg = /\bc/;
str='cainiao';
execReg(reg,str);
返回结果c。匹配到了左边界的c字符。

reg = /\bc/;
str='???c';
execReg(reg,str);
仍然返回c，不过这次返回的是右侧边界的c。

reg = /\bc/;
str='bcb';
execReg(reg,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：

reg = /\Bc/;
str='bcb';
execReg(reg,str);
这次会成功地匹配到bcb中的c，。然而

reg = /\Bc/;
str='cainiao';
execReg(reg,str);
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。例如：
reg = /\d/;
str='cainiao8';
execReg(reg,str);
返回的匹配结果为8，因为它是第一个数字字符。

reg = /\D/;
str='cainiao8';

execReg(reg,str);
返回c，第一个非数字字符。

空白

\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：

reg = /\s.+/;
str='This is a test String.';
execReg(reg,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。
reg = /\S+/;
str='This is a test String.';
execReg(reg,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。

单词字符

\w表示单词字符，等同于字符集合[a-zA-Z0-9_]。例如：
reg = /\w+/;
str='blueidea';
execReg(reg,str);
返回完整的blueidea字符串，因为所有字符都是单词字符。

reg = /\w+/;
str='.className';
execReg(reg,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。

reg = /\w+/;
str='?????';
execReg(reg,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_]
reg = /\W+/;
str='?????';
execReg(reg,str);
返回完整的字符串，因为，无论是中文和“？”都算作是非单词字符。

反向引用

形式如下：/(子正则表达式)\1/
依旧用例子来说明：
1.
reg = /\w/;
str='blueidea';
execReg(reg,str);
返回b。
2.
reg = /(\w)(\w)/;
str='blueidea';
execReg(reg,str);
返回bl,b,l
bl是整个正则匹配的内容，b是第一个括号里的子正则表达式匹配的内容，l是第二个括号匹配的内容。
3.
reg = /(\w)\1/;
str='blueidea';
execReg(reg,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字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b，因此“\1”就同样表示b了，在余下的字符串里自然找不到b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”。

reg = /(\w)\1/;
str='bbs.blueidea.com';
execReg(reg,str);
这个正则则会匹配到bb。

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：
reg = /(\w)(\w)\2\1/;
str='woow';
execReg(reg,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(**2**)

前面我们曾经讨论过一次括号的问题，见下面这个例子：
reg = /^(b|c).+/;
str='bbs.blueidea.com';
execReg(reg,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

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：
reg = /^(?:b|c).+/;
str='bbs.blueidea.com';
execReg(reg,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：
reg = /^(b|c)\1/;
str='bbs.blueidea.com';
execReg(reg,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。
reg = /^(?:b|c)\1/;
str='bbs.blueidea.com';
execReg(reg,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中的菜鸟。这时候就可以像下面这样写：/cainiao(?=8)/，看两个实例：
reg = /cainiao(?=8)/;
str='cainiao9';
execReg(reg,str);
返回null。

reg = /cainiao(?=8)/;
str='cainiao8';
execReg(reg,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。

再来看两个例子：
reg = /blue(?=idea)/;
str='blueidea';
execReg(reg,str);
匹配到blue，而不是blueidea。

reg = /blue(?=idea)/;

str='bluetooth';
execReg(reg,str);
返回null，因为blue后面不是idea。

reg = /blue(?=idea)/;
str='bluetoothidea';
execReg(reg,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：
reg = /blue(?!idea)/;
str='blueidea';
execReg(reg,str);
返回null，因为正则要求，blue的后面不能是idea。

reg = /blue(?!idea)/;
str='bluetooth';
execReg(reg,str);
则成功返回blue。

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
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：
reg = /c/;*
str='c';*
execReg(reg,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：
var reg = /b/;
var str = 'BBS';
execReg(reg,str);
返回null，因为大小写不符合。

var reg = /b/i;
var str = 'BBS';
execReg(reg,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：
var reg = /^b/;
var str = 'test\nbbs';
execReg(reg,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';
execReg(reg,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。

exec方法详解

exec方法的返回值

exec方法返回的其实并不是匹配结果字符串，而是一个对象，简单地修改一下execReg函数，来做一个实验就可以印证这一点：
function execReg(reg,str){
var result = reg.exec(str);
alert(typeof result);
}
var reg = /b/;
var str='bbs.bblueidea.com';
execReg(reg,str);
结果显示result的类型是object。而且是一个类似数组的对象。使用for in可以知道它的属性: index input 0。其中index是表示匹配在原字符串中的索引；而input则是表示输入的字符串；
至于0则是表示只有一个匹配结果，可以用下标0来引用这个匹配结果，这个数量可能改变。我们可以通过返回值的length属性来得知匹配结果的总数量。

根据以上对返回值的分析，修改execReg函数如下：

```
function execReg(reg,str){
var result = reg.exec(str);
document.write('index:'+result.index+'<br />'
+'input:'+result.input+'<br />'
);
for(i=0;i<result.length;i++){
document.write('result['+i+']:'+result[i]+'<br />')
}
}
```

马上来实验一下：

```
var reg = /\w/;
var str='bbs.bblueidea.com';
execReg(reg,str);
```

结果如下：

index:0
input:bbs.bblueidea.com
result[0]:b
输入字符串为bbs.bblueidea.com；
匹配的b在原字符串的索引是0。
正则的匹配结果为一个，b；

```
var reg = /(\w)(\w)(.+)/;
var str='bbs.bblueidea.com';
execReg(reg,str);
```

结果为：

index:0
input:bbs.bblueidea.com
result[0]:bbs.bblueidea.com
result[1]:b
result[2]:b
result[3]:s.bblueidea.com

由上面两个例子可见，返回对象[0]就是整个正则表达式所匹配的内容。后续的元素则是各个子正则表达式的匹配内容。

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表达式，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 = /b/;
var str = 'bbs.blueidea.com';
execReg(reg,str);
execReg(reg,str);
?????
index:0
input:bbs.blueidea.com
```

result[0]:b
index:0
input:bbs.blueidea.com
result[0]:b
也就是说，两次匹配的结果完全一样，从索引可以看出来，匹配的都是字符串首的b字符。
下面看看设置了g的正则表达式表现如何：
```
var reg = /b/g;
var str = 'bbs.blueidea.com';
execReg(reg,str);
execReg(reg,str);
```
结果如下：
index:0
input:bbs.blueidea.com
result[0]:b
index:1
input:bbs.blueidea.com
result[0]:b
可以看得出来，第二次匹配的是字符串的字符串的第二个b。这也就是g修饰符的作用了，下面来看exec是如何区别对待g和非g正则表达式的。

如果正则表达式没有设置g，那么exec方法不会对正则表达式有任何的影响，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。

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。同样建立一个简单的测试函数：
```
function testReg(reg,str){
alert(reg.test(str));
}
```
实例1
```
var reg = /b/;
var str = 'bbs.blueidea.com';
testReg(reg,str);
```
成功，输出true。
实例2
```
var reg = /9/;
var str = 'bbs.blueidea.com';
testReg(reg,str);
```
失败，返回false。

使用字符串的方法执行正则表达式

match方法

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类似数组的对象，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 ){
document.write('index:'+result.index+'<br />'
+'input:'+result.input+'<br />'
);
for(i=0;i<result.length;i++){
document.write('result['+i+']:'+result[i]+'<br />')
}
}else{
alert('null??????')
}
}
```

例如：

```
var reg = /b/;
var str = 'bbs.blueidea.com';
matchReg(reg,str);
```

结果如下：

```
index:0
input:bbs.blueidea.com
result[0]:b
```

可见，和exec的结果一样。
但是如果正则表达式设置了g修饰符，exec和match的行为可就不一样了，见下例：

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

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。

replace方法

形式：str. replace (reg,’new str’);
它的作用是将str字符串中匹配reg的部分用’ ’ new str”部分代码，值得注意的是原字符串并不会被修改，而是作为返回值被返回。例子：

```
var reg = /b/;
var str = 'bbs.blueidea.com';
```

var newStr = str.replace(reg,'c');
document.write(newStr);
结果为cbs.blueidea.com，只有第一个b被替换为c。

var reg = /b/g;
var str = 'bbs.blueidea.com';
var newStr = str.replace(reg,'c');
document.write(newStr);
输出ccs.clueidea.com
由于，设置了g修饰符，所以会替换掉所有的b。

var reg = /\w+/g;
var str = 'bbs.blueidea.com';
var newStr = str.replace(reg,'word');
document.write(newStr);
输出：
word.word.word。

在replace函数中使用$引用子正则表达式匹配内容

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：
var reg = /(\w+).(\w+).(\w+)/;
var str = 'bbs.blueidea.com';
var newStr = str.replace(reg,'$1.$1.$1');
document.write(newStr);
输出的结果为：
bbs.bbs.bbs
首先，我们知道第一个子正则表达式匹配到了bbs，那么$1也就代表bbs了。其后我们把替换字符串设置为'$1.$1.$1',其实也就是“bbs.bbs.bbs”。同理，$2就是blueidea，$3就是com。

在来看一个例子，颠倒空格前后两个单词的顺序。
var reg = /(\w+)\s(\w+)/;
var str = 'cainiao gaoshou';
var newStr = str.replace(reg,'$2 $1');
document.write(newStr);
结果为：gaoshou cainiao，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$$，例如：
var reg = /(\w+)\s(\w+)/;
var str = 'cainiao gaoshou';
var newStr = str.replace(reg,'$$ $$');
document.write(newStr);
结果为：$ $。

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 = /idea/;
var str = 'blueidea';
var pos = str.search(reg);
document.write(pos);
```

结果为4。
下面的例子找出第一个非单词字符：

```
var reg = /\W/;
var str = 'bbs.blueidea.com';
var pos = str.search(reg);
document.write(pos);
```

结果为3，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 = 'bbs.blueidea.com';
var arr = str.split(reg);
document.write(arr);
```

结果为：bbs,blueidea,com，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);
document.write(arr.length+'<br />');
document.write(arr);
```

结果为：
7
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。