JavaScript RegExp 对象

RegExp 对象

正则表达式是描述字符模式的对象。

正则表达式用于对字符串模式匹配及检索替换，是对字符串执行模式匹配的强大工具。

语法

```
var patt=new RegExp(pattern,modifiers);

或者更简单的方式:

var patt=/pattern/modifiers;
```

- pattern（模式） 描述了表达式的模式
- modifiers(修饰符) 用于指定全局匹配、区分大小写的匹配和多行匹配

***注意：当使用构造函数**创造正则对象时，需要常规的字符转义规则（在前面加反斜杠 \）。比如，以下是等价的：*

```
var re = new RegExp("\\w+");
var re = /\w+/;
```

更多关于 RegExp 对象请阅读我们的 JavaScript RegExp 对象教程。

修饰符

修饰符用于执行区分大小写和全局匹配:

修饰符	描述
i	执行对大小写不敏感的匹配。
g	执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
m	执行多行匹配。

方括号

方括号用于查找某个范围内的字符：

表达式	描述
[abc]	查找方括号之间的任何字符。
[^abc]	查找任何不在方括号之间的字符。
[0-9]	查找任何从 0 至 9 的数字。
[a-z]	查找任何从小写 a 到小写 z 的字符。
[A-Z]	查找任何从大写 A 到大写 Z 的字符。
[A-z]	查找任何从大写 A 到小写 z 的字符。
[adgk]	查找给定集合内的任何字符。
[^adgk]	查找给定集合外的任何字符。
(red|blue|green)	查找任何指定的选项。

元字符

元字符（Metacharacter）是拥有特殊含义的字符：

元字符	描述
.	查找单个字符，除了换行和行结束符。
\w	查找单词字符。
\W	查找非单词字符。
\d	查找数字。
\D	查找非数字字符。
\s	查找空白字符。
\S	查找非空白字符。
\b	匹配单词边界。

\B	匹配非单词边界。
\0	查找 NULL 字符。
\n	查找换行符。
\f	查找换页符。
\r	查找回车符。
\t	查找制表符。
\v	查找垂直制表符。
\xxx	查找以八进制数 xxx 规定的字符。
\xdd	查找以十六进制数 dd 规定的字符。
\uxxxx	查找以十六进制数 xxxx 规定的 Unicode 字符。

量词

量词	描述
n+	匹配任何包含至少一个 n 的字符串。 例如，/a+/ 匹配 "candy" 中的 "a"，"caaaaaaandy" 中所有的 "a"。
n*	匹配任何包含零个或多个 n 的字符串。 例如，/bo*/ 匹配 "A ghost booooed" 中的 "boooo"，"A bird warbled" 中的 "b"，但是不匹配 "A g grunted"。
n?	匹配任何包含零个或一个 n 的字符串。 例如，/e?le?/ 匹配 "angel" 中的 "el"，"angle" 中的 "le"。
n{X}	匹配包含 X 个 n 的序列的字符串。

	例如，/a{2}/ 不匹配 "candy," 中的 "a"，但是匹配 "caandy," 中的两个 "a"，且匹配 "caaandy." 中前两个 "a"。
n{X,}	X 是一个正整数。前面的模式 n 连续出现至少 X 次时匹配。 例如，/a{2,}/ 不匹配 "candy" 中的 "a"，但是匹配 "caandy" 和 "caaaaaaandy." 中所有的 "a"。
n{X,Y}	X 和 Y 为正整数。前面的模式 n 连续出现至少 X 次，至多 Y 次时匹配。 例如，/a{1,3}/ 不匹配 "cndy"，匹配 "candy," 中的 "a"，"caandy," 中的两个 "a"，匹配 "caaaaaaandy" 中的前面三个 "a"。注意，当匹配 "caaaaaaandy" 时，即使原始字符串拥有更多 "a"，匹配项也是 "aaa"。
n$	匹配任何结尾为 n 的字符串。
^n	匹配任何开头为 n 的字符串。
?=n	匹配任何其后紧接指定字符串 n 的字符串。
?!n	匹配任何其后没有紧接指定字符串 n 的字符串。

RegExp 对象方法

方法	描述	FF	IE
compile	编译正则表达式。	1	4
exec	检索字符串中指定的值。返回找到的值，并确定其位置。	1	4
test	检索字符串中指定的值。返回 true 或 false。	1	4

支持正则表达式的 String 对象的方法

方法	描述	FF	IE
search	检索与正则表达式相匹配的值。	1	4
match	找到一个或多个正则表达式的匹配。	1	4
replace	替换与正则表达式匹配的子串。	1	4

split	**把字符串分割**为字符串数组。	1	4

JavaScript 正则表达式

正则表达式（英语：Regular Expression，**在代**码中常简写为regex、regexp**或**RE）**使用**单个字符串来描述、匹配一系列符合某个句法规则的字符串搜索模式。

搜索模式可用于文本搜索和文本替换。

什么是正则表达式？

正则表达式是由一个字符序列形成的搜索模式。

当你在文本中搜索数据时，你可以用搜索模式来描述你要查询的内容。

正则表达式可以是一个简单的字符，或一个更复杂的模式。

正则表达式可用于所有文本搜索和文本替换的操作。

语法

```
/正则表达式主体/修饰符(可选)
```

其中修饰符是可选的。

实例：

```
var patt = /runoob/i
```

实例解析：

/runoob/i **是一个正**则表达式。

runoob **是一个正则表达式主体** (**用于**检索)。

i **是一个修饰符** (**搜索不区分大小写**)。

使用字符串方法

在 JavaScript **中**，**正**则表达式通常用于两个字符串方法 : search() **和** replace()。

search() 方法 **用于**检索字符串中指定的子字符串，或检索与正则表达式相匹配的子字符串，并返回子串的起始位置。

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。

search() 方法使用正则表达式

实例

使用正则表达式搜索 "Runoob" 字符串，且不区分大小写：

```
var str = "Visit Runoob!"; var n = str.search(/Runoob/i);
```

输出结果为：

```
6
```

尝试一下 »

search() 方法使用字符串

search 方法可使用字符串作为参数。字符串参数会转换为正则表达式：

实例

检索字符串中 "Runoob" 的子串：

```
var str = "Visit Runoob!"; var n = str.search("Runoob");
```

尝试一下 »

replace() 方法使用正则表达式

实例

使用正则表达式且不区分大小写将字符串中的 Microsoft 替换为 Runoob：

```
var str = document.getElementById("demo").innerHTML; var txt =
str.replace(/microsoft/i,"Runoob");
```

结果输出为:

```
Visit Runoob!
```

尝试一下 »

replace() 方法使用字符串

replace() 方法将接收字符串作为参数：

```
var str = document.getElementById("demo").innerHTML; var txt =
str.replace("Microsoft","Runoob");
```

尝试一下 »

你注意到了吗？

正则表达式参数可用在以上方法中 (替代字符串参数)。
正则表达式使得搜索功能更加强大(如实例中不区分大小写)。

正则表达式修饰符

修饰符 可以在全局搜索中不区分大小写:

修饰符	描述
i	执行对大小写不敏感的匹配。
g	执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
m	执行多行匹配。

正则表达式模式

方括号用于查找某个范围内的字符：

表达式	描述
[abc]	查找方括号之间的任何字符。
[0-9]	查找任何从 0 至 9 的数字。
(x|y)	查找任何以 | 分隔的选项。

元字符是拥有特殊含义的字符：

元字符	描述

\d	查找数字。
\s	查找空白字符。
\b	匹配单词边界。
\uxxxx	查找以十六进制数 xxxx 规定的 Unicode 字符。

量词:

量词	描述
n+	匹配任何包含至少一个 *n* 的字符串。
n*	匹配任何包含零个或多个 *n* 的字符串。
n?	匹配任何包含零个或一个 *n* 的字符串。

使用 RegExp 对象

在 JavaScript 中，RegExp 对象是一个预定义了属性和方法的正则表达式对象。

使用 test()

test() 方法是一个正则表达式方法。

test() 方法用于检测一个字符串是否匹配某个模式，如果字符串中含有匹配的文本，则返回 true，否则返回 false。

以下实例用于搜索字符串中的字符 "e"：

实例

```
var patt = /e/;
patt.test("The best things in life are free!");
```

字符串中含有 "e"，所以该实例输出为：

```
true
```

尝试一下 »

你可以不用设置正则表达式的变量，以上两行代码可以合并为一行：

```
/e/.test("The best things in life are free!")
```

使用 exec()

exec() 方法是一个正则表达式方法。

exec() 方法用于检索字符串中的正则表达式的匹配。

该函数返回一个数组，其中存放匹配的结果。如果未找到匹配，则返回值为 null。

以下实例用于搜索字符串中的字母 "e":

Example 1

```
/e/.exec("The best things in life are free!");
```

字符串中含有 "e"，所以该实例输出为:

```
e
```

尝试一下 »

更多实例

- JS 判断输入字符串是否为数字、字母、下划线组成
- JS 判断输入字符串是否全部为字母
- JS 判断输入字符串是否全部为数字

完整的 RegExp 参考手册

完整的 RegExp 对象参考手册，请参考我们的 JavaScript RegExp 参考手册。

该参考手册包含了所有 RegExp 对象的方法和属性。

JavaScript 类型转换

JavaScript 错误 – Throw、Try 和 Catch

笔记列表

1. qq1056125478
   105***5478@qq.com

正则表达式表单验证实例：

```
/*是否带有小数*/
function    isDecimal(strValue )  {
   var  objRegExp= /^\d+\.\d+$/;
   return  objRegExp.test(strValue);
```

```
}

/*校验是否中文名称组成 */
function ischina(str) {
    var reg=/^[\u4E00-\u9FA5]{2,4}$/;   /*定义验证表达式*/
    return reg.test(str);     /*进行验证*/
}

/*校验是否全由8位数字组成 */
function isStudentNo(str) {
    var reg=/^[0-9]{8}$/;   /*定义验证表达式*/
    return reg.test(str);     /*进行验证*/
}

/*校验电话码格式 */
function isTelCode(str) {
    var reg= /^((0\d{2,3}-\d{7,8})|(1[3584]\d{9}))$/;
    return reg.test(str);
}

/*校验邮件地址是否合法 */
function IsEmail(str) {
    var reg=/^([a-zA-Z0-9_-])+@([a-zA-Z0-9_-])+(\.[a-zA-Z0-9_-])+/;
    return reg.test(str);
}
```

尝试一下 »

正则表达式中的特殊字符

字符 含意

\ 做为转意，即通常在"\"后面的字符不按原来意义解释，如/b/匹配字符"b"，当b前面加了反斜杆后/\b/，转意为匹配一个单词的边界。

-或-

对正则表达式功能字符的还原，如"*"匹配它前面元字符0次或多次，/a*/将匹配a,aa,aaa，加了"\"后，/a*/将只匹配"a*"。

^ 匹配一个输入或一行的开头，/^a/匹配"an A"，而不匹配"An a"

$ 匹配一个输入或一行的结尾，/a$/匹配"An a"，而不匹配"an A"

* 匹配前面元字符0次或多次，/ba*/将匹配b,ba,baa,baaa

+ 匹配前面元字符1次或多次，/ba*/将匹配ba,baa,baaa

? 匹配前面元字符0次或1次，/ba*/将匹配b,ba

(x) 匹配x保存x在名为$1...$9的变量中

x|y 匹配x或y

{n} 精确匹配n次

{n,} 匹配n次以上

{n,m} 匹配n-m次

[xyz] 字符集(character set)，匹配这个集合中的任一一个字符(或元字符)

[^xyz] 不匹配这个集合中的任何一个字符

[\b] 匹配一个退格符

\b 匹配一个单词的边界

\B 匹配一个单词的非边界

\cX 这儿·X是一个控制符，/\cM/匹配Ctrl-M

\d 匹配一个字数字符，/\d/ = /[0-9]/

\D 匹配一个非字数字符，/\D/ = /[^0-9]/

\n 匹配一个换行符

\r 匹配一个回车符

\s 匹配一个空白字符，包括\n,\r,\f,\t,\v等

\S 匹配一个非空白字符，等于/[^\n\f\r\t\v]/

\t 匹配一个制表符

\v 匹配一个重直制表符

\w 匹配一个可以组成单词的字符(alphanumeric，这是我的意译·含数字)，包括下划线·如[\w]匹配"$5.98"中的5，等于[a-zA-Z0-9]

\W 匹配一个不可以组成单词的字符·如[\W]匹配"$5.98"中的$，等于[^a-zA-Z0-9]。

用re = new RegExp("pattern",["flags"]) 的方式比较好 pattern：正则表达式 flags: g （全文查找出现的所有 pattern） i （忽略大小写） m （多行查找）	vaScript动态正则表达式问题 请问正则表达式可以动态生成吗? 例如JavaScript中: var str = "strTemp"; 要生成: var re = /strTemp/; 如果是字符连接: var re = "/" + str + "/"即可 但是要生成表达式,可以实现吗?怎样实现?	

正则表达式是一个描述字符模式的对象。

JavaScript的RegExp对象和String对象定义了使用正则表达式来执行强大的模式匹配和文本检索与替换函数的方法.

在JavaScript中,正则表达式是由一个RegExp对象表示的.当然,可以使用一个RegExp()构造函数来创建RegExp对象,
也可以用JavaScript 1.2中的新添加的一个特殊语法来创建RegExp对象.就像字符串直接量被定义为包含在引号内的字符一样,
正则表达式直接量也被定义为包含在一对斜杠(/)之间的字符.所以,JavaScript可能会包含如下的代码:

var pattern = /s$/;

这行代码创建一个新的RegExp对象,并将它赋给变量parttern.这个特殊的RegExp对象和所有以字母"s"结尾的字符串都匹配.用RegExp()也可以定义
一个等价的正则表达式,代码如下:

var pattern = new RegExp("s$");

无论是用正则表达式直接量还是用构造函数RegExp(),创建一个RegExp对象都是比较容易的.较为困难的任务是用正则表达式语法来描述字符的模式.
JavaScript采用的是Perl语言正则表达式语法的一个相当完整的子集.
正则表达式的模式规范是由一系列字符构成的.大多数字符(包括所有字母数字字符)描述的都是按照字面意思进行匹配的字符.这样说来,正则表达式/java/就和所有包含子串 "java" 的字符串相匹配.
虽然正则表达式中的其它字符不是按照字面意思进行匹配的,但它们都具有特殊的意义.正则表达式 /s$/ 包含两个字符. 第一个特殊字符 "s" 是按照字面意思与自身相匹配.第二个字符 "$" 是一个特殊字符,它所匹配的是字符串的结尾.所以正则表达式 /s$/ 匹配的就是以字母 "s" 结尾的字符串.
1.直接量字符
我们已经发现了,在正则表达式中所有的字母字符和数字都是按照字面意思与自身相匹配的
.JavaScript的正则表达式还通过以反斜杠(\)开头的转义序列支持某些非字母字符.例如,序列 "\n" 在字符串中匹配的是一个直接量换行符.在正则表达式中,许多标点符号都有特殊的含义.下面是这些字符和它们的含义:
正则表达式的直接量字符
字符 匹配

字母数字字符 自身
\ f 换页符
\ n 换行符
\ r 回车
\ t 制表符
\ v 垂直制表符
\ / 一个 / 直接量
\ \ 一个 \ 直接量
\ . 一个 . 直接量
\ * 一个 * 直接量

\ + 一个 + 直接量
\ ? 一个 ? 直接量
\ | 一个 | 直接量
\ (一个 (直接量
\ ) 一个) 直接量
\ [一个 [直接量
\ ] 一个] 直接量
\ { 一个 { 直接量
\ } 一个 } 直接量
\ XXX 由十进制数 XXX 指 定的ASCII码字符
\ Xnn 由十六进制数 nn 指定的ASCII码字符
\ cX 控制字符^X. 例如, \cI等价于 \t, \cJ等价于 \n

如果想在正则表达式中使用特殊的标点符号,必须在它们之前加上一个 "\" .

2.字符类

将单独的直接符放进中括号内就可以组合成字符类.一个字符类和它所包含的任何一个字符都匹配,所以正则表达式 / [abc] / 和字母 "a" , "b" , "c" 中的任何一个都匹配.另外还可以定义否定字符类,这些类匹配的是除那些包含在中括号之内的字符外的所有字符.定义否定字符尖时,要将一个 ^ 符号作为从左中括号算起的第一个字符.正则表达式的集合是 / [a-zA-z0-9] / .

由于某些字符类非常常用,所以JavaScript的正则表达式语法包含一些特殊字符和转义序列来表示这些常用的类.例如, \s 匹配的是空格符,制表符和其它空白符, \s 匹配的则是空白符之外的任何字符.

正则表灰式的字符类

字符 匹配

[...] 位于括号之内的任意字符
[^...] 不在括号之中的任意字符
. 除了换行符之外的任意字符,等价于[^\n]
\w 任何单字字符, 等价于[a-zA-Z0-9]
\W 任何非单字字符,等价于[^a-zA-Z0-9]
\s 任何空白符,等价于[\ t \ n \ r \ f \ v]
\S 任何非空白符,等价于[^\ t \ n \ r \ f \ v]
\d 任何数字,等价于[0-9]
\D 除了数字之外的任何字符,等价于[^0-9]
[\b] 一个退格直接量(特例)

3.复制

用以上的正则表式的语法,可以把两位数描述成 / \ d \ d /,把四位数描述成 / \d \ d \ d \ d /.但我们还没有一种方法可以用来描述具有任意多数位的数字或者是一个字符串.这个串由三个字符以及跟随在字母之后的一位数字构成.这些复杂的模式使用的正则表达式语法指定了该表达式中每个元素要重复出现的次数.

指定复制的字符总是出现在它们所作用的模式后面.由于某种复制类型相当常用.所以有一些特殊的字符专门用于表示它们.例如: +号匹配的就是复制前一模式一次或多次的模式.下面的表列出了复制语法.先看一个例子:

/\d{2, 4}/ //匹配2到4间的数字.

/\w{3} \d?/ //匹配三个单字字符和一个任意的数字.

/\s+java\s+/ //匹配字符串"java" ,并且该串前后可以有一个或多个空格.

/[^"] * / //匹配零个或多个非引号字符.

正则表达式的复制字符

字符 含义

{n, m} 匹配前一项至少n次,但是不能超过m次

{n, } 匹配前一项n次,或者多次

{n} 匹配前一项恰好n次

? 匹配前一项0次或1次,也就是说前一项是可选的. 等价于 {0, 1}

+ 匹配前一项1次或多次,等价于{1,}

* 匹配前一项0次或多次.等价于{0,}

4.选择,分组和引用

正则表达式的语法还包括指定选择项,对子表达式分组和引用前一子表达式的特殊字符.字符| 用于分隔供选择的字符.例如: /ab|cd|ef/ 匹配的是字符串 "ab",或者是字符串 "cd",又或者 "ef".

/\d{3}|[a-z]{4}/ 匹配的是要么是一个三位数,要么是四个小写字母.在正则表达式中括号具有几种作用.它的主要作用是把单独的项目分组成子表达式,以便可以像处理一个独立的单元那种用 *、+或 ? 来处理那些项目.例如: /java(script) ?/ 匹配的是字符串 "java",其后既可以有 "script",也可以没有.

/

(ab|cd) + |ef) / 匹配的既可以是字符串 "ef",也可以是字符串"ab" 或者 "cd" 的一次或多次重复.

在正则表达式中,括号的第二个用途是在完整的模式中定义子模式。当一个正则表达式成功地和目标字符串相匹配时,可以从目标串中抽出和括号中的子模式相匹配的部分.例如,假定我们正在检索的模式是一个或多个字母后面跟随一位或多位数字,那么我们可以使用模式 / [a-z] + \ d+/.但是由于假定我们真正关心的是每个匹配尾部的数字,那么如果我们将模式的数字部分放在括号中 (/ [a-z] + (\d+)/) ,我们就可以从所检索到的任何匹配中抽取数字了,之后我们会对此进行解析的.

代括号的子表达式的另一个用途是,允许我们在同一正则表达式的后面引用前面的子表达式.这是通过在字符串 \ 后加一位或多位数字来实现的.数字指的是代括号的子表达式在正则表达式中的位置.例如: \1 引用的是第一个代括号的子表达式. \3 引用的是第三个代括号的子表达式.注意,由于子表达式可以嵌套在其它子表达式中,
所以它的位置是被计数的左括号的位置.
例如:在下面的正则表达式被指定为 \2:

/([Jj]ava([Ss]cript)) \sis \s (fun\w*) /

对正则表达式中前一子表达式的引用所指定的并不是那个子表达式的模式,而是与那个模式相匹配的文本.这样,引用就不只是帮助你输入正则表达式的重复部分的快
捷方式了,它还实施了一条规约,那就是一个字符串各个分离的部分包含的是完全相同的字符.例如:下面的正则表达式匹配的就是位于单引号或双引号之内的所有字符.但是,它要求开始和结束的引号匹配(例如两个都是双引号或者都是单引号):

/[' "] [^ ' "]*[' "]/

如果要求开始和结束的引号匹配,我们可以使用如下的引用:

/([' "]) [^ ' "] * \1/

\1匹配的是第一个代括号的子表达式所匹配的模式.在这个例子中,它实施了一种规约,那就是开始的引号必须和结束的引号相匹配.注意,如果反斜杠后跟随的数字比代括号的子表达式数多,那么它就会被解析为一个十进制的转义序列,而不是一个引用.你可以坚持使用完整的三个字符来表示转义序列,这们就可以避免混淆了.例如, 使用 \044,而不是\44.下面是正则表达式的选择、分组和引用字符:
字符 含义

| 选择.匹配的要么是该符号左边的子表达式,要么它右边的子表达式
(...) 分组.将几个项目分为一个单元.这个单元可由 *、+、? 和|等符号使用,而且还可以记住和这个组匹配的字符以供此后引
用使用
\n 和第n个分组所匹配的字符相匹配.分组是括号中的子表达式(可能是嵌套的).分组号是从左到右计数的左括号数

5.指定匹配的位置
我们已经看到了,一个正则表达式中的许多元素才能够匹配字符串的一个字符.例如: \s 匹配的只是一个空白符.还有一些正则表达式的元素匹配的是字符之间宽度为0的空间,而不是实际的字符例如: \b 匹配的是一个词语的边界,也就是处于一个/w字字符和一个\w非字字符之间的边界.像\b 这样的字符并不指定任何一个匹配了的字符串中的字符,它们指定的是匹配所发生的合法位置.有时我们称

这些元素为正则表达式的锚.因为它们将模式定位在检索字符串中的一个特定位置.最常用的锚元素是 ^, 它使模式依赖于字符串的开头,而锚元素$则使模式定位在字符串的末尾.
例如:要匹配词 "javascript" ,我们可以使用正则表达式 /^ javascript $/. 如果我们想检索 "java" 这个词自身 (不像在 "javascript" 中那样作为前缀),那么我们可以使用模式 /\s java \s /, 它要求在词语java之前和之后都有空格.但是这样作有两个问题.第一: 如果 "java" 出现在一个字符的开头或者是结尾.该模式就不会与之匹配,除非在开头和结尾处有一个空格. 第二: 当这个模式找到一个与之匹配的字符时,它返回的匹配的字符串前端和后端都有空格,这并不是我们想要的.因此,我们使用词语的边界 \b 来代替真正的空格符 \s 进行匹配. 结果表达式是 /\b java \b/.
下面是正则表达式的锚字符:
字符 含义

---

^ 匹配的是字符的开头,在多行检索中,匹配的是一行的开头
$ 匹配的是字符的结尾,在多行检索中,匹配的是一行的结尾
\b 匹配的是一个词语的边界.简而言之就是位于字符\w 和 \w之间的位置(注意:[\b]匹配的是退格符)
\B 匹配的是非词语的边界的字符

6.属性
有关正则表达式的语法还有最后一个元素,那就是正则表达式的属性,它说明的是高级模式匹配的规则.和其它正则表达式语法不同,属性是在 / 符号之外说明的.即它们不出现在两个斜杠之间,而是位于第二个斜杠之后.javascript 1.2支持两个属性.属性 i 说明模式匹配应该是大小写不敏感的.属性 g 说明模式匹配应该是全局的.也
就是说,应该找出被检索的字符串中所有的匹配.这两种属性联合起来就可以执行一个全局的,大小写不敏感的匹配.
例如: 要执行一个大小不敏感的检索以找到词语 "java" (或者是 "java" 、"JAVA"等) 的第一个具体值,我们可以使用大小不敏感的正则表达式 /\b java\b/i .如果要在一个字符串中找到 "java" 所有的具体值,我们还可以添加属性 g, 即 /\b java \b/gi .
以下是正则表达式的属性:

字符 含义

---

i 执行大小写不敏感的匹配
g 执行一个全局的匹配,简而言之,就是找到所有的匹配,而不是在找到第一个之后就停止了

除属性 g 和 i 之外,正则表达式就没有其它像属性一样的特性了.如果将构造函数 RegExp 的静态属性 multiline 设置为 true ,那么模式匹配将以多行的模式进行.在这种模式下,锚字符 ^ 和 $ 匹配的

不只是检索字符串的开头和结尾,还匹配检索字符串内部的一行的开头和结尾.例如: 模式 /Java$/ 匹配的是 "Java",但是并不匹配
"Java\nis fun" .如果我们设置了 multiline 属性,那么后者也将被匹配:

RegExp.multiline = true;
在JAVASCRIPT里面判断一个字符串是否是电子邮件的格式：

复制代码代码如下:

```
if(formname.email.value!=formname.email.value.match(/^\w +[@]\w +[.][\w.] +$/))
{
alert("您的电子邮件格式错误！");
formname.email.focus();
return false;
}

[RED]function dateVerify(date){
var reg = /^(\d{4})(-)(\d{2})\2(\d{2})$/;
var r = date.match(reg);
if(r==null) return false;
var d= new Date(r[1], r[3]-1,r[4]);
var newStr=d.getFullYear()+r[2]+(d.getMonth()+1)+r[2]+d.getDate();
date=r[1]+r[2]+((r[3]-1)+1)+r[2]+((r[4]-1)+1);
return newStr==date;
}[/RED]
```

javascript的17种正则表达式

```
"^\\d+$"    //非负整数（正整数 + 0）
"^[0-9]*[1-9][0-9]*$"    //正整数
"^((-\\d+)|(0+))$"    //非正整数（负整数 + 0）
"^-[0-9]*[1-9][0-9]*$"    //负整数
"^-?\\d+$"        //整数
"^\\d+(\\.\\d+)?$"    //非负浮点数（正浮点数 + 0）
"^(([0-9]+\\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\\.[0-9]+)|([0-9]*[1-9][0-9]*))$"    //正浮点数
"^((-\\d+(\\.\\d+)?)|(0+(\\.0+)?))$"    //非正浮点数（负浮点数 + 0）
"^(-(([0-9]+\\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\\.[0-9]+)|([0-9]*[1-9][0-9]*)))$"    //负浮点数
"^(-?\\d+)(\\.\\d+)?$"    //浮点数
```

"^[A-Za-z]+$" //由26个英文字母组成的字符串
"^[A-Z]+$" //由26个英文字母的大写组成的字符串
"^[a-z]+$" //由26个英文字母的小写组成的字符串
"^[A-Za-z0-9]+$" //由数字和26个英文字母组成的字符串
"^\\w+$" //由数字、26个英文字母或者下划线组成的字符串
"^[\\w-]+(\\.[\\w-]+)*@[\\w-]+(\\.[\\w-]+)+$" //email地址
"^[a-zA-z]+://(\\w+(-\\w+)*)(\\.(\\w+(-\\w+)*))*(\\?\\S*)?$" //url

正则表达式对象的属性及方法

预定义的正则表达式拥有有以下静态属性：input, multiline, lastMatch, lastParen, leftContext, rightContext和$1到$9。其中input和multiline可以预设置。其他属性的值在执行过exec或test方法后被根据不同条件赋以不同的值。许多属性同时拥有长和短(perl风格)的两个名字，并且，这两个名字指向同一个值。(JavaScript模拟perl的正则表达式)

正则表达式对象的属性

属性 含义

$1...$9 如果它(们)存在，是匹配到的子串

$_ 参见input

$* 参见multiline

$& 参见lastMatch

$+ 参见lastParen

$` 参见leftContext

$" 参见rightContext

constructor 创建一个对象的一个特殊的函数原型

global 是否在整个串中匹配(bool型)

ignoreCase 匹配时是否忽略大小写(bool型)

input 被匹配的串

lastIndex 最后一次匹配的索引

lastParen 最后一个括号括起来的子串

leftContext 最近一次匹配以左的子串

multiline 是否进行多行匹配(bool型)

prototype 允许附加属性给对象

rightContext 最近一次匹配以右的子串

source 正则表达式模式

lastIndex 最后一次匹配的索引

正则表达式对象的方法

方法 含义

compile 正则表达式比较

exec 执行查找

test 进行匹配

toSource 返回特定对象的定义(literal representing)，其值可用来创建一个新的对象。重载Object.toSource方法得到的。

toString 返回特定对象的串。重载Object.toString方法得到的。

valueOf 返回特定对象的原始值。重载Object.valueOf方法得到

例子

复制代码代码如下:

```
<script language = "JavaScript">
var myReg = /(w+)s(w+)/;
var str = "John Smith";
var newstr = str.replace(myReg, "$2, $1");
document.write(newstr);
</script>
```

将输出"Smith, John"

javascript正则表达式检验

复制代码代码如下:

```
//校验是否全由数字组成
function isDigit(s)
{
var patrn=/^[0-9]{1,20}$/;
if (!patrn.exec(s)) return false
return true
}
//校验登录名：只能输入5-20个以字母开头、可带数字、“_”、“.”的字串
function isRegisterUserName(s)
{
var patrn=/^[a-zA-Z]{1}([a-zA-Z0-9]|[._]){4,19}$/;
if (!patrn.exec(s)) return false
return true
}
```

```
//校验用户姓名：只能输入1-30个以字母开头的字串
function isTrueName(s)
{
var patrn=/^[a-zA-Z]{1,30}$/;
if (!patrn.exec(s)) return false
return true
}
//校验密码：只能输入6-20个字母、数字、下划线
function isPasswd(s)
{
var patrn=/^(\w){6,20}$/;
if (!patrn.exec(s)) return false
return true
}
//校验普通电话、传真号码：可以"+"开头，除数字外，可含有"-"
function isTel(s)
{
//var patrn=/^[+]{0,1}(\d){1,3}[ ]?([-]?(\d){1,12})+$/;
var patrn=/^[+]{0,1}(\d){1,3}[ ]?([-]?((\d)|[ ]){1,12})+$/;
if (!patrn.exec(s)) return false
return true
}
//校验手机号码：必须以数字开头，除数字外，可含有"-"
function isMobil(s)
{
var patrn=/^[+]{0,1}(\d){1,3}[ ]?([-]?((\d)|[ ]){1,12})+$/;
if (!patrn.exec(s)) return false
return true
}
//校验邮政编码
function isPostalCode(s)
{
//var patrn=/^[a-zA-Z0-9]{3,12}$/;
var patrn=/^[a-zA-Z0-9 ]{3,12}$/;
if (!patrn.exec(s)) return false
return true
}
```

```
//校验搜索关键字
function isSearch(s)
{
var
patrn=/^[^`~!@#$%^&*()+=|\\\][\]\{\}:;'\,.<>/?]{1}[^`~!@$%^&()+=|\\\][\]\{\}:;'\,.<>?]
{0,19}$/;
if (!patrn.exec(s)) return false
return true
}
function isIP(s) //by zergling
{
var patrn=/^[0-9.]{1,20}$/;
if (!patrn.exec(s)) return false
return true
}
```

正则表达式regular expression详述(一)

正则表达式是regular expression，看来英文比中文要好理解多了，就是检查表达式符不符合规定！！正则表达式有一个功能十分强大而又十分复杂的对象RegExp，在JavaScript1.2 版本以上提供。

下面我们看看有关正则表达式的介绍：

正则表达式对象用来规范一个规范的表达式(也就是表达式符不符合特定的要求，比如是不是Email地址格式等)，它具有用来检查给出的字符串是否符合规则的属性和方法。

除此之外，你用RegExp构造器建立的个别正则表达式对象的属性，就已经预先定义好了正则表达式对象的静态属性，你可以随时使用它们。

核心对象：

在JavaScript 1.2, NES 3.0以上版本提供。

在JavaScript 1.3以后版本增加了toSource方法。

建立方法：

文字格式或RegExp构造器函数。

文字建立格式使用以下格式：

/pattern/flags即/模式/标记

构造器函数方法使用方法如下：

new RegExp("pattern"[, "flags"])即new RegExp("模式"[,"标记"])

参数：

pattern(模式)

表示正则表达式的文本

flags(标记)

如果指定此项，flags可以是下面值之一：

g: global match(全定匹配)

i: ignore case(忽略大小写)

gi: both global match and ignore case(匹配所有可能的值，也忽略大小写)

注意：文本格式中的参数不要使用引号标记，而构造器函数的参数则要使用引号标记。所以下面的表达式建立同样的正则表达式：

/ab+c/i

new RegExp("ab+c", "i")

描述：

当使用构造函数的时候，必须使用正常的字符串避开规则(在字符串中加入前导字符\ )是必须的。

例如，下面的两条语句是等价的：

re = new RegExp("\\w+")

re = /\w+/

下面的提供了在正则表达式中能够使用的完整对特殊字符的一个完整的列表和描述。

表1.3：正则表达式中的特殊字符：

字符\

意义：对于字符，通常表示按字面意义，指出接着的字符为特殊字符，\不作解释。

例如：/b/匹配字符'b',通过在b 前面加一个反斜杠\，也就是/\b/，则该字符变成特殊字符，表示匹配一个单词的分界线。

或者：

对于几个字符，通常说明是特殊的，指出紧接着的字符不是特殊的，而应该按字面解释。

例如：*是一个特殊字符，匹配任意个字符(包括0个字符)；例如：/a*/意味匹配0个或多个a。

为了匹配字面上的*，在a前面加一个反斜杠；例如：/a*/匹配'a*'。

字符^

意义：表示匹配的字符必须在最前边。

例如：/^A/不匹配"an A,"中的'A'，但匹配"An A."中最前面的'A'。

字符$

意义：与^类似，匹配最末的字符。

例如：/t$/不匹配"eater"中的't'，但匹配"eat"中的't'。

字符*

意义：匹配*前面的字符0次或n次。

例如:/bo*/匹配"A ghost booooed"中的'boooo'或"A bird warbled"中的'b'，但不匹配"A goat grunted"中的任何字符。

字符+

意义：匹配+号前面的字符1次或n次。等价于{1,}。

例如：/a+/匹配"candy"中的'a'和"caaaaaaandy."中的所有'a'。

字符?

意义：匹配?前面的字符0次或1次。

例如：/e?le?/匹配"angel"中的'el'和"angle."中的'le'。

字符.

意义：(小数点)匹配除换行符外的所有单个的字符。

例如：/.n/匹配"nay, an apple is on the tree"中的'an'和'on'，但不匹配'nay'。

字符(x)

意义：匹配'x'并记录匹配的值。

例如：/(foo)/匹配和记录"foo bar."中的'foo'。匹配子串能被结果数组中的素[1], ..., [n] 返回，或被RegExp对象的属性$1, ..., $9返回。

字符x|y

意义：匹配'x'或者'y'。

例如：/green|red/匹配"green apple"中的'green'和"red apple."中的'red'。

字符{n}

意义：这里的n是一个正整数。匹配前面的n个字符。

例如：/a{2}/不匹配"candy,"中的'a'，但匹配"caandy," 中的所有'a'和"caaandy."中前面的两个'a'。

字符{n,}

意义：这里的n是一个正整数。匹配至少n个前面的字符。

例如：/a{2,}不匹配"candy"中的'a'，但匹配"caandy"中的所有'a'和"caaaaaaandy."中的所有'a'

字符{n,m}

意义：这里的n和m都是正整数。匹配至少n个最多m个前面的字符。

例如：/a{1,3}/不匹配"cndy"中的任何字符，但匹配 "candy,"中的'a'，"caandy," 中的前面两个'a'和"caaaaaaandy"中前面的三个'a'，注意：即使"caaaaaaandy" 中有很多个'a'，但只匹配前面的三个'a'即"aaa"。

字符[xyz]

意义：一字符列表，匹配列出中的任一字符。你可以通过连字符-指出一个字符范围。

例如：[abcd]跟[a-c]一样。它们匹配"brisket"中的'b'和"ache"中的'c'。

字符[^xyz]

意义：一字符补集，也就是说，它匹配除了列出的字符外的所有东西。 你可以使用连字符-指出一字符范围。

例如：[^abc]和[^a-c]等价，它们最早匹配"brisket"中的'r'和"chop."中的'h'。

字符[\b]

意义：匹配一个空格(不要与\b混淆)

字符\b

意义：匹配一个单词的分界线，比如一个空格(不要与[\b]混淆)

例如：/\bn\w/匹配"noonday"中的'no'，/\wy\b/匹配"possibly yesterday."中的'ly'。

字符\B

意义：匹配一个单词的非分界线

例如：/\w\Bn/匹配"noonday"中的'on'，/y\B\w/匹配"possibly yesterday."中的'ye'。

字符\cX

意义：这里的X是一个控制字符。匹配一个字符串的控制字符。

例如：/\cM/匹配一个字符串中的control-M。

字符\d

意义：匹配一个数字，等价于[0-9]。

例如：/\d/或/[0-9]/匹配"B2 is the suite number."中的'2'。

字符\D

意义：匹配任何的非数字，等价于[^0-9]。

例如：/\D/或/[^0-9]/匹配"B2 is the suite number."中的'B'。

字符\f

意义：匹配一个表单符

字符\n

意义：匹配一个换行符

字符\r

意义：匹配一个回车符

字符\s

意义：匹配一个单个white空格符，包括空格，tab，form feed，换行符，等价于[\f\n\r\t\v]。

例如：/\s\w*/匹配"foo bar."中的' bar'。

字符\S

意义：匹配除white空格符以外的一个单个的字符，等价于[^ \f\n\r\t\v]。

例如：/\S/\w*匹配"foo bar."中的'foo'。

字符\t

意义：匹配一个制表符

字符\v

意义：匹配一个顶头制表符

字符\w

意义：匹配所有的数字和字母以及下划线，等价于[A-Za-z0-9_]。

例如：/\w/匹配"apple,"中的'a'，"$5.28,"中的'5'和"3D."中的'3'。

字符\W

意义：匹配除数字、字母外及下划线外的其它字符，等价于[^A-Za-z0-9_]。

例如：/\W/或者/[^$A-Za-z0-9_]/匹配"50%."中的'%'。

字符\n

意义：这里的n是一个正整数。匹配一个正则表达式的最后一个子串的n的值(计数左圆括号)。

例如：/apple(,)\sorange\1/匹配"apple, orange, cherry, peach."中的'apple, orange'，下面有一个更加完整的例子。
注意：如果左圆括号中的数字比\n指定的数字还小，则\n取下一行的八进制escape作为描述。
字符\ooctal和\xhex
意义：这里的\ooctal是一个八进制的escape值，而\xhex是一个十六进制的escape值，允许在一个正则表达式中嵌入ASCII码。

当表达式被检查的时候，文字符号提供了编辑正则表达式的方法。利用文字符号可以使到正则表达式保持为常数。例如，如果你在一个循环中使用文字符号来构造一个正则表达式，正则表达式不需进行反复编译。
正则表达式对象构造器，例如，new RegExp("ab+c")，提供正则表达式的运行时编译。当你知道正则表达式的模式会变化的时候，应该使用构造函数，或者你不知道正则表达式的模式，而它们是从另外的源获得的时候，比如由用户输入时。一旦你定义好了正则表达式，该正则表达式可在任何地方使用，并且可以改变，你可以使用编译方法来编译一个新的正则表达式以便重新使用。
一个分离预先定义的RegExp对象可以在每个窗口中使用；也就是说，每个分离的JavaScript线程运行以获得自己的RegExp对象。因为每个脚本在一个线程中是不可中断的，这就确保了不同的脚本不会覆盖RegExp对象的值。
预定义的RegExp对象包含的静态属性：input, multiline, lastMatch,lastParen, leftContext, rightContext, 以及从$1到$9。input和multiline属性能被预设。其它静态属性的值是在执行个别正则表达式对象的exec和test方法后，且在执行字符串的match和replace方法后设置的。
属性
注意RegExp对象的几个属性既有长名字又有短名字(象Perl)。这些名字都是指向相同的值。Perl是一种编程语言，而JavaScript模仿了它的正则表达式。
属性$1, ..., $9
取得匹配的子串，如果有的话
属性$_
参考input
属性$*
参考multiline
属性$&
参考lastMatch
属性$+
参考lastParen
属性$`
参考leftContext
属性$'
参考rightContext

属性constructor
指定用来建立对象原型函
属性global
决定是否测试正则表达式是否不能匹配所有的字符串，或者只是与最先的冲突。
属性ignoreCase
决定试图匹配字符串的时候是否忽略大小写
属性input
当正则表达式被匹配的时候，为相反的字符串。
属性lastIndex
决定下一次匹配从那里开始
属性lastMatch
最后一个匹配的字符
属性lastParen
子串匹配的时候，最后一个parenthesized，如果有的话。
属性leftContext
最近一次匹配前的子串。
属性multiline
是否在串的多行中搜索。
属性prototype
允许附加属性到所有的对象
属性rightContext
最近一次匹配后的的子串。
属性source
模式文本
方法
compile方法
编译一个正则表达式对象
exec方法
运行正则表达式匹配
test方法
测试正则达式匹配
toSource方法
返回一个对象的文字描述指定的对象；你可以使用这个值来建立一个新的对象。不考虑Object.toSource方法。
toString方法
返回一个字符串描述指定的对象，不考虑Object.toString对象。

valueOf方法
返回指定对角的原始值。不考虑Object.valueOf方法。
另外，这个对象继承了对象的watch和unwatch方法

例子：
例 1、下述示例脚本使用replace方法来转换串中的单词。在替换的文本中，脚本使用全局 RegExp 对象的$1和$2属性的值。注意，在作为第二个参数传递给replace方法的时候，RegExp对象的$属性的名称。

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr=str.replace(re,"$2, $1");
document.write(newstr)
</SCRIPT>
```

显示结果："Smith, John".
例 2、下述示例脚本中，RegExp.input由Change事件处理句柄设置。在getInfo函数中，exec 方法使用RegExp.input的值作为它的参数，注意RegExp预置了$属性。

```
<SCRIPT LANGUAGE="JavaScript1.2">
function getInfo(abc)
{
re = /(\w+)\s(\d+)/;
re.exec(abc.value);
window.alert(RegExp.$1 + ", your age is " + RegExp.$2);
}
</SCRIPT>
```

请输入你的姓和年龄，输入完后按回车键。

```
<FORM><INPUT TYPE="TEXT" NAME="NameAge" onChange="getInfo(this);"></FORM>
</HTML>
```

$1, ..., $9属性
用圆括号括着的匹配子串，如果有的话。
是RegExp的属性
静态，只读
在JavaScript 1.2, NES 3.0以上版本提供
描述：因为input是静态属性，不是个别正则表达式对象的属性。你可以使用RegExp.input 访问该属性。

能加上圆括号的子串的数量不受限制，但正则表达式对象只能保留最后9 条。如果你要访问所有的圆括号内的匹配字串，你可以使用返回的数组。
这些属性能用在RegExp.replace方法替换后的字符串(输出结果)。当使用这种方式的时候，不用预先考虑RegExp对象。下面给出例子。当正则表达式中没有包含圆括号的时候，该脚本解释成$n的字面意义。(这里的n是一个正整数)。
例如：
下例脚本使用replace 方法来交换串中单词的位置。在替换后的文本字串中，脚本使用正则表达式RegExp对象的$1和$2属性的值。注意：当它们向replace方法传递参数的时候，这里没有考虑 $ 属性的
RegExp对象的名称。

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr=str.replace(re,"$2, $1");
document.write(newstr)
</SCRIPT>
```

显示的输出结果为：Smith, John。

正则表达式regular expression详述(二)
正则表达式详述（二）
以下这些不是正则表达式的新增对象请参阅对应的JavaScript对象的属性 $_属性 参考input $*属性 参考multiline $&属性 参考lastMatch $+属性 参考lastParen $`属性
参考leftContext $'属性 参考rightContext compile方法 在脚本运行期间编译正则表达式对象
属于RegExp的方法 在JavaScript 1.2, NES 3.0以上版本提供 语法：
regexp.compile(pattern[, flags]) 以数： regexp 正则表达式的名称，可以是变量名或文字串。
pattern 正则表达式的定义文本。 flags 如果指定的话，可以是下面其中的一个： "g": 匹配所有可能的字串
"i": 忽略大小写 "gi": 匹配所有可能的字串及忽略大小写 描述：
使用compile方法来编译一个正则表达式 created with the RegExp constructor function。这样就强制正则表达式只编译一次，而不是每次遇到正则表达式的时候都编译一次。当你确认正则表达式能
保持不变的时候可使用compile 方法来编译它(在获得它的匹配模式后)， 这样就可以在脚本中重复多次使用它。
你亦可以使用compile 方法来改变在运行期间改变正则表达式。例如，假如正则表达式发生变化
，
你可以使用compile方法来重新编译该对象来提高使用效率。
使用该方法将改变正则表达式的source, global和ignoreCasesource属性的值。 constructor

指出建立对象原型的function。注意这个属性的值由函数本身提供，而不是一个字串包含RegExp的name.Property提供。
在JavaScript 1.1, NES 2.0以上版本提供 ECMA版本ECMA-262 描述：参考Object.constructor.
exec方法 在指定的字符串运行匹配搜索。返回一个结果数组。 是RegExp的方法
在JavaScript 1.2, NES 3.0以上版本提供 语法： regexp.exec([str])regexp([str])
参数： regexp，正则表达式的名称，可以是一个变量名或文字定义串。
str，要匹配正则表达式的字符串，如果省略，将使用RegExp.input的值。
描述：就如在语法描述中的一样，正则表达工的exec方法能够被直接调用(使用regexp.exec(str))或者间接调用(使用regexp(str))。
假如你只是运行以找出是否匹配，可以使用String搜索方法。
假如匹配成功，exec方法返回一个数组并且更新正则表达式对象属性的值和预先定义的正则表达式对象、RegExp。如果匹配失败，exec方法返回null。
请看下例： <SCRIPT LANGUAGE="JavaScript1.2"> //匹配一个b接着一个或多个d，再接着一个b //忽略大小写 myRe=/d(b+)(d)/ig; myArray = myRe.exec("cdbBdbsbz");
</SCRIPT> 下面是该脚本的返回值：对象 属性/Index 描述 例子

myArray
myArray的内容 ["dbBd", "bB", "d"]

index
基于0的匹配index 1

input
原始字符串 cdbBdbsbz

[0]
最后匹配的字符 dbBd

[1], ...[n]
用圆括号括住的匹配字符串，如果有的话。不限制括号的个数。 [1] = bB

[2] = d

myRe

lastIndex
开始下次匹配操作的index值 5

ignoreCase
指出"i"是否使用以忽略大小写 true

global
指出是否使用"g"标记来进行匹配所有可能的字串 true

source
定义模式的文本字符串 d(b+)(d)

RegExp

lastMatch$&

最后匹配的字符 dbBd

leftContext$\Q
最新匹配前面的子串 c

rightContext$'
最新匹配后面的子串 bsbz

$1, ...$9
圆括号内的匹配子串，如果有的话。圆括号的个数不受限制，但RegExp只能保留最后9个 $1 =

bB

$2 = d

lastParen $+
最后一个加上圆括号的匹配子串，如果有的话 d
假如你的正则表达式使用了"g"标记，你可以多次使用exec 方法来连续匹配相同的串。当你这样做的时候，新的匹配将从由正则表达式的lastIndex 属性值确定的子串中开始。例如，假定你使用下面的脚本：

<SCRIPT LANGUAGE="JavaScript1.2"> myRe=/ab*/g;str = "abbcdefabh"

myArray = myRe.exec(str);

document.writeln("Found "+myArray[0]+". Next match starts at "+myRe.lastIndex)

mySecondArray = myRe.exec(str);

document.writeln("Found "+mySecondArray[0]+". Next match starts at "+myRe.lastIndex)

</SCRIPT>
这个脚本显示如下结果： Found abb. Next match starts at 3

Found ab. Next match starts at 9 例子：
在下面的例子中，用户输入一个名字，脚本根据输入执行匹配操作。接着检查数组看是否和其它用户的名字匹配。
本脚本假定已注册的用户的姓已经存进了数组A中，或许从一个数据库中取得。

<HTML>

<SCRIPT LANGUAGE="JavaScript1.2"> A = ["zhao","qian","sun","li","liang"]

function lookup() { firstName = /\w+/i(); if (!firstName)
window.alert (RegExp.input + "非法输入"); else { count=0;

for (i=0;i 输入你的姓然后按回车键。

<FORM><INPUT TYPE:"TEXT" NAME="FirstName" onChange="lookup(this);"></FORM>

</HTML>
global属性 正则表达式中是否使用了"g"标记。 RegExp属性，只读
在JavaScript 1.2, NES 3.0以上版本提供 描述： global是一个个别正则表达式对象的属性
如果使用了"g"标记，global的值为true；否则为 false。"g"标记指定正则表达式测试所有可能的匹配。
你不能直接改变该属性的值，但可以调用compile方法来改变它。 ignoreCase 检查正则表达式是否

使用了"i"标记
RegExp属性，只读 在JavaScript 1.2, NES 3.0以上版本提供 描述：
ignoreCase是个别正则表达式对象的一个属性。
如果使用了"i"标记，则返回true，否则返回false。"i"标记指示在进行匹配的时候忽略大小写。
你不能直接改变该属性的值，但可以通过调用compile方法来改变它 input 指出正则表达式要测试那个字串。$_是这个属性的另一个名字。
RegExp的属性，静态 在JavaScript 1.2, NES 3.0以上版本提供
描述：因为input是静态的，不是某个个别的正则表达式对象的属性。你也可以使用 RegExp.input 来表示。
如果没有给正则表达式的exec或test方法提供字符串，并且RegExp.input中有值，则使用它的值来调用该方法。
脚本或浏览器能够预置input属性。如果被预置了值且调用exec或 test方法的时候没有提供字符串则调用exec或test的时候使用input的值。input可以被浏览器以下面的方式设置：
当text表单域处理句柄被调用的时候，input被设置为该text输入的字串。
当textarea表单域处理句柄被调用的时候，input被设置为textarea域内输入的字串。注意multiline亦被设置成true从而能匹配多行文本。当select表单域处理句柄被调用的时候，input被设置成selected text的值。
当链接对象的处理句柄被调用的时候，input被设置成<A HREF=...>和</A>之间的字符串。
事件理现句柄处理完毕后，input属性的值被清除。 lastIndex 可读/可写的一个整数属性，指出下一次匹配从哪里开始。
RegExp的属性 在JavaScript 1.2, NES 3.0以上版本提供
描述：lastIndex 是个别的正则表达式对象的属性。 这个属性只有当正则表达式的"g"标记被使用以进行全串匹配的时候才被设置。实行以下规则：
如果lastIndex大小字符串的长度，regexp.test和regexp.exec失败，且lastIndex被设为0。
如果lastIndex等于字串的长度且正则表达式匹配空字符串，则正则表达式从lastIndex的位置开始匹配。
如果lastIndex等于字符串的长度且正则表达式不匹配空字符串，则正则表达式不匹配input，且lastIndex被置为0。
否则，lastIndex被设置成最近一次匹配的下一点。 例如，按下面的顺序执行脚本： re = /(hi)?/g 匹配空字符串
re("hi") 返回["hi", "hi"]，lastIndex置为2
re("hi") 返回[""]，一个空数组，它的下标为0的元素就是匹配字符串。在这种情况下，返回空串是因为lastIndex等于2(且仍然是2)，并且"hi"的长度也是2。lastMatch 最后一次匹配字符串，$& 是同样的意思。
RegExp的属性，静态，只读 在JavaScript 1.2, NES 3.0以上版本提供
描述：因为lastMatch是静态的，所以它不是个别指定正则表达式的属性。你也可以使用 RegExp.lastMatch。lastParen

最后一次加上括号的匹配字符串，如果有的话。$+是同样的意思。 RegExp属性，静态，只读
在JavaScript 1.2, NES 3.0以上版本提供
描述：因为lastParen是静态的，它不是某个个别正则式的属性，你可以使用RegExp.lastParen 表达同样的意思。
leftContext 最近一次匹配前面的子串，$`具有相同的意思。RegExp的属性，静态，只读
在JavaScript 1.2, NES 3.0以上版本提供
描述：因为leftContext是静态的，不是某一个正则表达式的属性，所以可以使用RegExp.leftContext来表达想同的意思。
multiline 反映是否匹配多行文本，$*是相同的意思。RegExp的属性，静态
在JavaScript 1.2, NES 3.0以上版本提供
描述：因为multiline是静态的，而不是某个个别正则表达式的属性，所以能够用RegExp.multiline表达相同的意思。
如果允许匹配多行文本，则multiline为true，如果搜索必须在换行时停止，则为false。
脚本或浏览器能够设置multiline属性。当一个textarea的事件处理句柄被调用的时候，multiline被置为true。在事件处理句柄处理完毕后，multiline属性值被清除。也就是说，如果你设置了multiline为true，则执行任何的事件处理句柄后，multiline被置为false。prototype
描绘类的原型。你可以根据要求使用prototype来增加类的属性或方法。为了获得prototypes 的资料，请参阅RegExp的Function.prototype.Property属性。从JavaScript 1.1, NES 2.0版本开始提供
ECMA版本ECMA-262 rightContext 最后一次匹配的右边的字符串，$'是同样的效果。
RegExp的属性，静态，只读 从 JavaScript 1.2, NES 3.0以上版本开始提供
描述：因为rightContext是静态的，不是某个个别正则表达工的属性，可以使用RegExp.rightContext来达到相同的效果。
source 一个只读属性，包含正则表达式定义的模式，不包侨forward slashes和"g"或"i"标记。
RegExp的属性，只读
从JavaScript 1.2, NES 3.0以上版本开始提供
描述：source是个别正则表达式对象的属性，你不能直接改变它的值，但可以通过调用compile 方法来改变它。 test
执行指定字符串的正则表达式匹配搜索，返回true或false。RegExp的方法
从JavaScript 1.2, NES 3.0以上版本开始提供 语法：regexp.test([str])
参数：regexp，正则表达式的名称，可以是变量名或正则表达式定义文字串
str，要匹配的字符串，如果省略，将使用RegExp.input的值为作参数
描述：当你需要知道一个字符串能否匹配某个正则表达工，可以使用test方法(与String.search方法类似)；为了获得更多的信息(但速度将变慢)，可以使用exec方法(与String.match方法类似)。例子：下面的例子显示test是否成功的提示：

```
function testinput(re, str){
if (re.test(str)) midstring = " contains ";
```

else midstring = " does not contain ";

document.write (str + midstring + re.source); } toSource
返回一个字符串象征对象的源码 RegExp的方法 从JavaScript 1.3以上版本开始提供 语法：

toSource()
参数：没有 描述：toSource方法返回下述的值： 对于内置的RegExp对象，toSource返回下面的字符象征源码不可用：

function Boolean(){ [native code] }
在RegExp场合中, toSource返回象征源码的字符串，通常这个方法是由JavaScript内部自动调用而不是不代码中显式调用。
更多请看Object.toSource toString 返回描绘指定对象的字符串。 RegExp的方法
从JavaScript 1.1, NES 2.0开始提供 ECMA版本ECMA-262 语法：toString() 参数：无
描述：RegExp对象不考虑Object对象的toString方法；它不继承Object.toString，对于RegExp 对象，toString方法返回一个代表该对象的字符串。 例如：下面的例子显示象征RegExp对象的字符串

myExp = new RegExp("a+b+c"); alert(myExp.toString())
displays "/a+b+c/" 更多请看：Object.toString valueOf 返回一个RegExp对象的原始值
RegExp的方法 从JavaScript 1.1版本开始提供 ECMA版本：ECMA-262 语法：valueOf()
参数：无 描述：RegExp的valueOf方法以字符串形式返回RegExp对象的原始值，这个值与RegExp.toString相等。
该方法通常由JavaScript内部自动调用而不是显式调用 例子： myExp = new RegExp("a+b+c");

alert(myExp.valueOf()) displays "/a+b+c/"

正则表达式在javascript中的几个实例1(转)
! 去除字符串两端空格的处理
如果采用传统的方式,就要可能就要采用下面的方式了
复制代码代码如下:

```
//清除左边空格
function js_ltrim(deststr)
{
if(deststr==null)return "";
var pos=0;
var retStr=new String(deststr);
if (retStr.lenght==0) return retStr;
while (retStr.substring(pos,pos+1)==" ") pos++;
retStr=retStr.substring(pos);
return(retStr);
```

```
}
//清除右边空格
function js_rtrim(deststr)
{
if(deststr==null)return "";
var retStr=new String(deststr);
var pos=retStr.length;
if (pos==0) return retStr;
while (pos && retStr.substring(pos-1,pos)==" " ) pos--;
retStr=retStr.substring(0,pos);
return(retStr);
}
//清除左边和右边空格
function js_trim(deststr)
{
if(deststr==null)return "";
var retStr=new String(deststr);
var pos=retStr.length;
if (pos==0) return retStr;
retStr=js_ltrim(retStr);
retStr=js_rtrim(retStr);
return retStr;
}
```

采用正则表达式,来去除两边的空格,只需以下代码

```
String.prototype.trim = function()
{
return this.replace(/(^\s*)|(\s*$)/g, "");
}
```

一句就搞定了,

可见正则表达式为我们节省了相当的编写代码量

! 移动手机号的校验

如果采用传统的校验方式至少就要完成下面三步的校验,

(1). 是否是数字

(2).是否是11位

(3).数字的第三位是否是5,6,7,8,9

如果采用正则表达式校验,只需以下代码

复制代码代码如下:

```
function checkMobile1(form)
{
if (form.mobile.value > "")
{
var reg=/13[5,6,7,8,9]\d{8}/;
if ( form.mobile.value.match(reg)== null)
{
alert("请输入正确的移动手机号码！");
form.mobile.focus(); return false;
}
}
return true;
}
```

从上面的代码可以看出校验移动手机号只需定义一个var reg=/13[5,6,7,8,9]\d{8}/;模式匹配串就可以完成合法性校验了

! URL的校验,

条件:必须以http:// 或 https:// 开头, 端口号必须为在1－65535 之间, 以下代码完成了合法性校验

复制代码代码如下:

```
//obj:数据对象
//dispStr :失败提示内容显示字符串
function checkUrlValid( obj, dispStr)
{
if(obj == null)
{
alert("传入对象为空");
return false;
}
var str = obj.value;
var urlpatern0 = /^https?:\/\/.+$/i;
if(!urlpatern0.test(str))
{
alert(dispStr+"不合法：必须以'http:\/\/'或'https:\/\/'开头!");
obj.focus();
```

```
return false;
}
var urlpatern2= /^https?:\/\/([a-zA-Z0-9_-])+(\.)?)*(:\d+)?.+$/i;
if(!urlpatern2.test(str))
{
alert(dispStr+"端口号必须为数字且应在1－65535之间!");
obj.focus();
return false;
}

var urlpatern1 =/^https?:\/\/([a-zA-Z0-9_-])+(\.)?)*(:\d+)?(\/((\.)?(\?)?=?&?[a-zA-Z0-9_-
](\?)?)*)*$/i;
if(!urlpatern1.test(str))
{
alert(dispStr+"不合法,请检查!");
obj.focus();
return false;
}
var s = "0";
var t =0;
var re = new RegExp(":\\d+","ig");
while((arr = re.exec(str))!=null)
{
s = str.substring(RegExp.index+1,RegExp.lastIndex);
if(s.substring(0,1)=="0")
{
alert(dispStr+"端口号不能以0开头!");
obj.focus();
return false;
}
t = parseInt(s);
if(t<1 || t >65535)
{
alert(dispStr+"端口号必须为数字且应在1－65535之间!");
obj.focus();
return false;
}
```

```
}
return true;
}
```

对url的校验,看上去有很多的代码,这是因为要给予出错提示, 否则只需var urlpatern1 =/^https?:\/\/(([a-zA-Z0-9_-])+(\.)?)*(:\d+)?(\/((\.)?(\?)?=?&?[a-zA-Z0-9_-](\?)?)*)*$/i; 一句就可以校验出url合法性了

正则表达式在JavaScript应用

去掉字符串头尾多余的空格

/g是全文查找所有匹配

```
function String.prototype.Trim(){return this.replace(/(^\s*)|(\s*$)/g, "");}
function String.prototype.LTrim(){return this.replace(/(^\s*)/g, "");}
function String.prototype.RTrim(){return this.replace(/(\s*$)/g, "");}
```

应用：计算字符串的长度（一个双字节字符长度计2，ASCII字符计1）

```
String.prototype.len=function(){return this.replace([^\x00-\xff]/g,"aa").length;}
```

应用：javascript中没有像vbscript那样的trim函数，我们就可以利用这个表达式来实现，如下：

```
String.prototype.trim = function()
{
return this.replace(/(^\s*)|(\s*$)/g, "");
}
```

得用正则表达式从URL地址中提取文件名的javascript程序，如下结果为page1

```
s="http://www.jb51.net/page1.htm"
s=s.replace(/(.*\/){0,}([^\.]+).*/ig,"$2")
alert(s)
```

##利用正则表达式限制网页表单里的文本框输入内容：

用正则表达式限制只能输入中文：onkeyup="value=value.replace(/[^\u4E00-\u9FA5]/g,')"

onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\u4E00-\u9FA5]/g,'))"

用正则表达式限制只能输入全角字符： onkeyup="value=value.replace(/[^\uFF00-\uFFFF]/g,')"

onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\uFF00-\uFFFF]/g,''))"

用正则表达式限制只能输入数字：onkeyup="value=value.replace(/[^\d]/g,')

"onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\d]/g,''))"

用正则表达式限制只能输入数字和英文：onkeyup="value=value.replace(/[\W]/g,')

"onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\d]/g,''))"

用正则表达式和javascript对表单进行全面验证

使用时请将下面的javascript代码存到一个单一的js文件中。

1、表单要求

<form name="formname" onSubmit="return validateForm(this)"></form>

将对表单中的所有以下类型的域依次验证，所有验证是去除了前导和后缀空格的，要注意是区分大小写的。

2、空值验证

表单中任意域加上emptyInfo属性将对此域是否为空进行验证（可以和最大长度验证\一般验证方式同时使用）。

无此属性视为此域允许空值。

如：<input type="text" name="fieldNamename" emptyInfo="字段不能为空！">

3、最大长度验证（可以和空值验证、一般验证方式同时使用）：

<input type="text" name="fieldNamename" maxlength="20" lengthInfo="最大长度不能超过20！">

或,<textarea maxlength="2000" lengthInfo="最大长度不能超过2000！">

3、一般验证方式(不对空值做验证）:

如：<input type="text" validator="^(19|20)[0-9]{2}$" errorInfo="不正确的年份!" >

4、标准验证(不与其它验证方式同时使用）：

全部通过<input type="hidden">来实现，并且不需要name属性以免提交到服务器。

4.1、合法日期验证：

<input type="text" name="yearfieldName" value="2004">注：这里也可以是<select name="yearfieldName"></select>，以下同

<input type="text" name="monthfieldName" value="02">

<input type="text" name="dayfieldName" value="03">

<input type="hidden" validatorType="DateGroup" year="yearfieldName" month="monthfieldName" day="dayfieldName" errorInfo="不正确的日期!">

yearfieldName、monthfieldName、dayfieldName分别为年月日字段，月和日可以是两位(MM)或一位格式(M),

此处不对每个字段分别检验(如果要检验，请在年月日三个域分别使用前面的一般验证方式），只对日期的最大值是否合法检查;

4.2、日期格式验证(请注意，此验证不对日期是否有效进行验证，还未找到从格式中得到年月日数据的方法^_^)：

<input type="text" name="datefieldName" value="2003-01-03 21:31:00">

<input type="hidden" validatorType="Date" fieldName="datefieldName"; format="yyyy-MM-dd HH:mm:ss" errorInfo="不正确的日期!">

其中格式仅对y、M、d、H、m、s进行支持(其它字符视为非时间的字符)

4.3、列表验证：

检验列表(checkbox、redio、select)是否至少选中了一条记录(对select主要用于多项选择)

<input type="checkbox" name="checkbox1">

<input type="hidden" validatorType="Checkbox" fieldName="checkbox1" errorInfo="请至少选中一条记录!">

其中validatorType可以是Checkbox、R、Select;

对于一个select表单，如果要求选择一条不能是第一条的记录，请用下列方式:

<select name="select1" emptyInfo="请选择一个选项!">

<option value="">==请选择==</option>

<option value="1">1</option>

<select>

4.4、Email验证：

<input type="text" name="email">

<input type="hidden" fieldName="email" validatorType="Email" separator="," errorInfo="不正确的Email!">

其中separator为可选项，表示输入多个email时的分隔符(无此选项只能是一个地址)

4.5、加入其它javascript操作：

<script type="text/javascript">

function functionname(){

自定义方法

}

</script>

表单中加入<input type="hidden" validatorType="javascript" functionName="functionname">（此时emptyInfo等属性无效）

时将调用function属性中指定的javascript方法（要求方法返回true或false，返回false将不再验证表单，也不提交表单)。

5、在表单通过验证提交前disable一个按钮（也可将其它域disable,不能与其它验证同在一个域),不要求按钮是表单中的最后一个

<input type="button" name="提交" validatorType="disable">

6、不验证表单

<input type="hidden" name="validate" value="0" functionName="functionname">
当validator域值为0时不对表单进行验证，直接提交表单或执行指定function并返回true后提交表单

functionName为可选

复制代码代码如下:

```
<script type="text/javascript">
function getStringLength(str){
var endvalue=0;
var sourcestr=new String(str);
var tempstr;
for (var strposition = 0; strposition < sourcestr.length; strposition ++) {
tempstr=sourcestr.charAt(strposition);
if (tempstr.charCodeAt(0)>255 || tempstr.charCodeAt(0)<0) {
endvalue=endvalue+2;
} else {
endvalue=endvalue+1;
}
}
return(endvalue);
}
function trim(str){
if(str==null) return "";
if(str.length==0) return "";
var i=0,j=str.length-1,c;
for(;i<str.length;i++){
c=str.charAt(i);
if(c!=' ') break;
}
for(;j>-1;j--){
c=str.charAt(j);
if(c!=' ') break;
}
if(i>j) return "";
return str.substring(i,j+1);
}
```

```
function validateDate(date,format,alt){
var time=trim(date.value);
if(time=="") return;
var reg=format;
var reg=reg.replace(/yyyy/,"[0-9]{4}");
var reg=reg.replace(/yy/,"[0-9]{2}");
var reg=reg.replace(/MM/,"((0[1-9])|1[0-2])");
var reg=reg.replace(/M/,"(([1-9])|1[0-2])");
var reg=reg.replace(/dd/,"((0[1-9])|([1-2][0-9])|30|31)");
var reg=reg.replace(/d/,"([1-9]|[1-2][0-9]|30|31))");
var reg=reg.replace(/HH/,"(([0-1][0-9])|20|21|22|23)");
var reg=reg.replace(/H/,"([0-9]|1[0-9]|20|21|22|23)");
var reg=reg.replace(/mm/,"([0-5][0-9])");
var reg=reg.replace(/m/,"([0-9]|([1-5][0-9]))");
var reg=reg.replace(/ss/,"([0-5][0-9])");
var reg=reg.replace(/s/,"([0-9]|([1-5][0-9]))");
reg=new RegExp("^"+reg+"$");
if(reg.test(time)==false){//验证格式是否合法
alert(alt);
date.focus();
return false;
}
return true;
}
function validateDateGroup(year,month,day,alt){
var array=new Array(31,28,31,30,31,30,31,31,30,31,30,31);
var y=parseInt(year.value);
var m=parseInt(month.value);
var d=parseInt(day.value);
var maxday=array[m-1];
if(m==2){
if((y%4==0&&y%100!=0)||y%400==0){
maxday=29;
}
}
if(d>maxday){
alert(alt);
```

```
return false;
}
return true;
}
function validateCheckbox(obj,alt){
var rs=false;
if(obj!=null){
if(obj.length==null){
return obj.checked;
}
for(i=0;i<obj.length;i++){
if(obj[i].checked==true){
return true;
}
}
}
alert(alt);
return rs;
}
function validateRadio(obj,alt){
var rs=false;
if(obj!=null){
if(obj.length==null){
return obj.checked;
}
for(i=0;i<obj.length;i++){
if(obj[i].checked==true){
return true;
}
}
}
alert(alt);
return rs;
}
function validateSelect(obj,alt){
var rs=false;
if(obj!=null){
```

```
for(i=0;i<obj.options.length;i++){
if(obj.options[i].selected==true){
return true;
}
}
}
alert(alt);
return rs;
}
function validateEmail(email,alt,separator){
var mail=trim(email.value);
if(mail=="") return;
var em;
var myReg = /^[_a-z0-9]+@([_a-z0-9]+\.)+[a-z0-9]{2,3}$/;
if(separator==null){
if(myReg.test(email.value)==false){
alert(alt);
email.focus();
return false;
}
}
else{
em=email.value.split(separator);
for(i=0;i<em.length;i++){
em[i]=em[i].trim();
if(em[i].length>0&&myReg.test(em[i])==false){
alert(alt);
email.focus();
return false;
}
}
}
return true;
}
function validateForm(theForm){// 若验证通过则返回true
var disableList=new Array();
var field = theForm.elements; // 将表单中的所有元素放入数组
```

```
for(var i = 0; i < field.length; i++){
var vali=theForm.validate;
if(vali!=null){
if(vali.value=="0"){
var fun=vali.functionName;
if(fun!=null){
return eval(fun+"()");
}
else{
return true;
}
}
}
var empty=false;
var value=trim(field[i].value);
if(value.length==0){//是否空值
empty=true;
}
var emptyInfo=field[i].emptyInfo;//空值验证
if(emptyInfo!=null&&empty==true){
alert(emptyInfo);
field[i].focus();
return false;
}
var lengthInfo=field[i].lengthInfo;//最大长度验证
if(lengthInfo!=null&&getStringLength(value)>field[i].maxLength){
alert(lengthInfo);
field[i].focus();
return false;
}
var validatorType=field[i].validatorType;
if(validatorType!=null){//其它javascript
var rs=true;
if(validatorType=="javascript"){
eval("rs="+field[i].functionName+"()");
if(rs==false){
return false;
```

```
}
else{
continue;
}
}
else if(validatorType=="disable"){//提交表单前disable的按钮
disableList.length++;
disableList[disableList.length-1]=field[i];
continue;
}
else if(validatorType=="Date"){
rs=validateDate(theForm.elements(field[i].fieldName),field[i].format,field[i].errorInfo);
}
else if(validatorType=="DateGroup"){
rs=validateDateGroup(theForm.elements(field[i].year),theForm.elements(field[i].month),theForm
.elements(field[i].day),field[i].errorInfo);
}
else if(validatorType=="Checkbox"){
rs=validateCheckbox(theForm.elements(field[i].fieldName),field[i].errorInfo);
}
else if(validatorType=="Radio"){
rs=validateRadio(theForm.elements(field[i].fieldName),field[i].errorInfo);
}
else if(validatorType=="Select"){
rs=validateSelect(theForm.elements(field[i].fieldName),field[i].errorInfo);
}
else if(validatorType=="Email"){
rs=validateEmail(theForm.elements(field[i].fieldName),field[i].errorInfo);
}
else{
alert("验证类型不被支持, fieldName: "+field[i].name);
return false;
}
if(rs==false){
return false;
}
}
```

```
else{//一般验证
if(empty==false){
var v = field[i].validator; // 获取其validator属性
if(!v) continue; // 如果该属性不存在,忽略当前元素
var reg=new RegExp(v);
if(reg.test(field[i].value)==false){
alert(field[i].errorInfo);
field[i].focus();
return false;
}
}
}
}
for(i=0;i<disableList.length;i++){
disableList[i].disabled=true;
}
return true;
}
</script>
```