**正则表达式**

**1.定义**  var reg=new RegExp("hello")或 var reg=/hello/

2. String.prototype.search()方法  
 "abchello".search(/hello/); // 3

用来找出原字符串中某个子字符串首次出现的index，没有则返回-1

3. String.prototype.replace()方法  
 "abchello".replace(/hello/,"hi"); // "abchi"

用来替换字符串中的子串

4. String.prototype.split（）方法  
 "abchelloasdasdhelloasd".split(/hello/); //["abc", "asdasd", "asd"]

用来分割字符串，类似将‘hello’视为分割符号，所以它也不见了

5. String.prototype.match（）方法.

用来捕获字符串中的子字符串到一个数组中。默认情况下只捕获一个结果到数组中，正则表达式有”全局捕获“的属性时(定义正则表达式的时候添加参数g)，会捕获所有结果到数组中。

"abchelloasdasdhelloasd".match(/hello/); //["hello"]

"abchelloasdasdhelloasd".match(/hello/g); //["hello","hello"]

**g代表全局搜索**

1-5为对象为字符串的方法

6. RegExp.prototype.test方法

用来测试字符串中是否含有子字符串

/hello/.test("abchello"); // true

7. RegExp.prototype.exec方法

和字符串的match方法类似，这个方法也是从字符串中捕获满足条件的字符串到数组中，但是也有两个区别。

（1）. exec方法一次只能捕获一份子字符串到数组中，无论正则表达式是否有全局属性

var reg=/hello/g;

reg.exec("abchelloasdasdhelloasd"); // ["hello"]

（2）. 正则表达式对象(也就是JavaScript中的RegExp对象)有一个lastIndex属性，用来表示下一次从哪个位置开始捕获，每一次执行exec方法后，lastIndex就会往后推，直到找不到匹配的字符返回null，然后又从头开始捕获。 这个属性可以用来遍历捕获字符串中的子串。

var reg=/hello/g;

reg.lastIndex; //0

reg.exec("abchelloasdasdhelloasd"); // ["hello"]

reg.lastIndex; //8

reg.exec("abchelloasdasdhelloasd"); // ["hello"]

reg.lastIndex; //19

reg.exec("abchelloasdasdhelloasd"); // null

reg.lastIndex; //0

**8. 元字符**

//匹配数字: \d

"ad3ad2ad".match(/\d/g); // ["3", "2"]

//匹配除换行符以外的任意字符: .

"a\nb\rc".match(/./g); // ["a", "b", "c"]

//匹配字母或数字或下划线 ： \w

"a5\_ 汉字@!-=".match(/\w/g); // ["a", "5", "\_"]

//匹配空白符:\s

"\n \r".match(/\s/g); //[" ", " ", ""] 第一个结果是\n，最后一个结果是\r

\n换行(LF) ，将当前位置移到下一行开头

\r回车(CR) ，将当前位置移到本行开头

//匹配【单词开始或结束】的位置 ： \b

"how are you".match(/\b\w/g); //["h", "a", "y"]

// 匹配【字符串开始和结束】的位置: 开始 ^ 结束 $

"how are you".match(/^\w/g); // ["h"]

"how are you".match(/\w$/g); // ["u"]

反义元字符----写法就是把上面的小写字母变成大写的，比如 ， 匹配所有不是数字的字符： \D

**9.在 [] 中使用符号 -  ，可以用来表示字符范围。**

// 匹配字母 a-z 之间所有字母

/[a-z]/

// 匹配Unicode中 数字 0 到 字母 z 之间的所有字符

/[0-z]/

// unicode编码查询地址：

//https://en.wikibooks.org/wiki/Unicode/Character\_reference/0000-0FFF

//根据上面的内容，我们可以找出汉字的Unicode编码范围是 \u4E00 到 \u9FA5，所以我们可以写一个正则表达式来判断一个字符串中是否有汉字

/[\u4E00-\u9FA5]/.test("测试"); // true

//重复n次 {n}

"test12".match(/test\d{3}/); // null

"test123".match(/test\d{3}/); // ["test123"]

//重复n次或更多次 {n,}

"test123".match(/test\d{3,}/); // ["test123"]

//重复n到m次

"test12".match(/test\d{3,5}/); // null

"test12345".match(/test\d{3,5}/); // ["test12345"]

"test12345678".match(/test\d{3,5}/); // ["test12345"]

// 匹配字符test后边跟着数字，数字重复0次或多次——\*

"test".match(/test\d\*/); // ["test"]

"test123".match(/test\d\*/); // ["test123"]

//重复一次或多次—— +

"test".match(/test\d+/) ; // null

"test1".match(/test\d\*/); //["test1"]

//重复一次或0次—— 0

"test".match(/test\d?/) ; // null

"test1".match(/test\d?/); //["test1"]

从上面的结果可以看到，字符test后边跟着的数字可以重复0次或多次时，正则表达式捕获的子字符串会返回尽量多的数字，比如/test\d\*/匹配 test123 ，返回的是test123，而不是test或者test12。

正则表达式捕获字符串时，在满足条件的情况下捕获尽可能多的字符串，这就是所谓的“贪婪模式”。

对应的”懒惰模式“，就是在满足条件的情况下捕获尽可能少的字符串，使用懒惰模式的方法，就是在字符重复标识后面加上一个 "?"，写法如下

// 数字重复3~5次，满足条件的情况下返回尽可能少的数字

"test12345".match(/test\d{3,5}?/); //["test123"]

// 数字重复1次或更多，满足条件的情况下只返回一个数字

"test12345".match(/test\d+?/); // ["test1"]

10.字符转义

在正则表达式中元字符是有特殊的含义的，当我们要匹配元字符本身时，就需要用到字符转义，比如：

/\./.test("."); // true

\n \t \w \v \r \d \ $ \* ^ + ? \b \f \s

11. 正则表达式可以用 " () " 来进行分组，具有分组的正则表达式除了正则表达式整体会匹配子字符串外，分组中的正则表达式片段也会匹配字符串。

" the best language in the world is java ".replace(/(java)/,"$1script");

// " the best language in the world is javascript "

"/static/app1/js/index.js".replace(/(\/\w+)\.js/,"$1-v0.0.1.js");

//"/static/app1/js/index-v0.0.1.js"

(\/\w+)分组匹配的就是 /index ，在第二个参数中为其添加上版本号

match函数中，当正则表达式有全局属性时，会捕获所有满足正则表达式的子字符串

"abchellodefhellog".match(/h(ell)o/g); //["hello", "hello"]

但是当正则表达式没有全局属性，且正则表达式中有分组的时候，match函数只会返回整个正则表达式匹配的第一个结果，同时会将分组匹配到的字符串也放入结果数组中：

"abchellodefhellog".match(/h(ell)o/); //["hello", "ell"]

exec函数在正则表达式中有分组的情况下，表现和match函数很像，只是无论正则表达式是否有全局属性，exec函数都只返回一个结果，并捕获分组的结果

/h(ell)o/g.exec("abchellodefhellog"); //["hello", "ell"]

a|b ——a或b

12. **(?:exp) ,**用此方式定义的分组，正则表达式会匹配分组中的内容，但是不再给此分组分配组号，此分组在replace、match等函数中的作用也会消失，效果如下：

/(hello)\sworld/.exec("asdadasd hello world asdasd")

// ["hello world", "hello"],正常捕获结果字符串和分组字符串

/(?:hello)\sworld/.exec("asdadasd hello world asdasd")

// ["hello world"]

"/static/app1/js/index.js".replace(/(\/\w+)\.js/,"$1-v0.0.1.js"); //"/static/app1/js/index-v0.0.1.js"

"/static/app1/js/index.js".replace(/(?:\/\w+)\.js/,"$1-v0.0.1.js"); //"/static/app1/js$1-v0.0.1.js"

//忽略大小写 i

"abchelloasdasdHelloasd".match(/hello/g); //["hello"]

"abchelloasdasdHelloasd".match(/hello/gi); //["hello","Hello"]