JS通过RegExp类型业支持正则表达式

正则表达式是一个描述字符模式的对象。String/RegExp都定义了很多相关函数。

正则的创建方式：

1、通过RegExp构造函数，new运算符

var pattern = new RegExp(‘abc’,’gi’); //abc参数字符串 gi可选模式修饰符

2、通过字面量方式

var pattern = /abc/gi; //斜杠“/”字面量方式

g：表示全局模式，将被应用于所有字符串，而非在发现第一个匹配项时立即停止

i：表示不区分大小写模式，忽略模式和字符串的大小写

m：表示多行模式，即在到达一行文本末尾时还会继续查找下一行中是否存在匹配的项。

RegExp对象包含两个方法：test() exec()都用于测试字符匹配

test()方法返回布尔值，存在返回true，不存在返回false

var pattern = new RegExp(‘abc’,’gi’);

var str = ‘this is a abc’;

console.log(pattern.test(str));

exec()方法返回数组，存在返回结果数组，不存在返回null

var pattern = new RegExp('abc','gi');

var str = 'this is a abc';

console.log(pattern.exec(str)); //返回数组的值

String对象中的正则表达式的相关方法

match()获取匹配数组

var pattern = new RegExp('abc','i'); //没有参数g，不在全局只查找一个abc后就返回

var str = 'this is a abc abc abc abc';

console.log(str.match(pattern)); //注意这里是字符串str的方法，参数是正则表达式

//在字符串中查找正则表达式，找到就返回数组，找不到返回null

replace(patter,参数1)：用参数1把字符串中pattern替换掉，返回替换后的字符串

var pattern = new RegExp('abc','gi'); //不开全局只限第一个

var str = 'this is a abc abc abc abc';

console.log(str.replace(pattern, '123'));

search(pattern)：返回字符串中参数pattern的开始位置，查找不到返回-1

var pattern = new RegExp('123','gi');

var str = 'this is a abc abc abc abc';

console.log(str. search (str,pattern));

split(pattern)：在字符串中按pattern拆分，并返回字符串按指定参数pattern拆分的数组

var pattern = new RegExp(' ','gi');

var str = 'this is a abc abc abc abc';

console.log(str.split(pattern));

元字符：也叫特殊字符

正则表达式中的元字符包括：() [] {} ^ $ | ? \* .-

()：标记一个子表达式的开始和结束位置

var pattern = /^([a-z]+)\s([0-9]{4})$/; //分组,[a-z]+表示1个或多个字母，[0-9]{4}表示4个数字字符

var str = 'google 2017';

console.log(pattern.exec(str)); //[google 2017,google,2017]

[]：标记一个中括号表达式的开始和结束位置

[xyz]：匹配所包含的任意一个字符

var pattern = /[a-z]oogle/;

var str = 'google'; //var str = 'ggggggoogle'; //true不限定从行首开始，只要最后有符合要求都

console.log(pattern.test(str));

{}：标记限定符表达式的开始

var pattern = /go{2,4}gle/; //o{2,4} 表示匹配o2-4个，包含2、4

var str = 'goooogle';

console.log(pattern.test(str));

var pattern = /go{2,}gle/; //o{2,} 表示匹配o2个、2个以上

var str = 'goooogle';

console.log(pattern.test(str));

锚字符：

^：匹配输入字符串的开始位置，（在括号表达式内使用，表示不接受该字符集合。 [^0-9]） 行首匹配：匹配以a-z（一个字符）开头的字符串

var pattern = /^[a-z]oogle/;

var str = '4oogle'; //var str = '44444oogle';//false 匹配限定从行首开始

console.log(pattern.test(str)); //false

^在括号内使用：

var pattern = /[^a-z]oogle/; //除a-z字母以外的其他字符

var str = '4oogle'; //var str = '44444oogle';

console.log(pattern.test(str)); //true

$：匹配输入字符串的结尾位置，行尾匹配。

var pattern = /google/; //没有限定行尾位置开始匹配

var str = 'aasadsa google 2133441431';

console.log(pattern.test(str)); //true

var pattern = /google{2,4}$/; //e匹配2－4次

var str = 'googlee';

console.log(pattern.test(str)); //true

var pattern = /(google){2,4}$/; //google匹配2－4次

var str = 'googlegooglegoogle';

console.log(pattern.test(str)); //true

var pattern = /^google$/; //强制首尾位置都要匹配，精确匹配

var str = 'aasadsa google 2133441431';

console.log(pattern.test(str)); //false

var pattern = /^[a-z]+\s[0-9]{4}$/; //没有分组

var str = 'google 2017';

console.log(pattern.exec(str)); //[google 2017]

var pattern = /^([a-z]+)\s([0-9]{4})$/; //分组,[a-z]+表示1个或多个字母，[0-9]{4}表示4个数字字符

var str = 'google 2017';

console.log(pattern.exec(str)); //[google 2017,google,2017]

\A：只有匹配字符串开始处

\*：匹配子表达式 （0、1、多）次 > 0

var pattern = /go\*gle/;

var str = 'goooogle';

console.log(pattern.test(str));

+：匹配子表达式 （1、多）次，至少一个及以上 > 1

var pattern = /go+gle/;

var str = 'gogle'; // str = 'google'; str = 'gogle'; str = 'goooogle';

console.log(pattern.test(str)); //true

?：匹配子表达式 （0、1）次，<1

var pattern = /go+gle/;

var str = 'gogle'; // str = 'ggle';

console.log(pattern.test(str)); //true

.：匹配除换行符\n之外的任何单字符（一个字符）。

var pattern = /g.gle/;

var str = 'gogle';

console.log(pattern.test(str)); //true

-：范围表示法，[x-y] 表示x到y整个范围内的字符中的任意一个匹配

[0-9] == \d [^0-9] == \D [0-9a-zA-Z] == \w [^0-9a-zA-Z] == \W

｜：或匹配 x|y 匹配x或y

var pattern = /google|bing|baidu/;

var str =' this is a bing ';

console.log(pattern.test(str));

空白字符

\f：换页符

\n：换行符

\r：回车符

\s：任何空白字符，包括空格、制表符、换页符，＝＝＝[\f\n\r\t\v]

\S：非空白字符。＝＝＝[^\f\n\r\t\v]

\t：制表符

\v：垂直制表符

普通字符组

使用范围表示法：[x-y] [0-9] == [0123456789] [a-z] == [abc…xyz]

根据字符对应的码值（ASCII编码表），码值小的字符在前，码值大的字符在后

运算符优先级

正则表达式从左到右进行计算

\

() (?:) (?=) []

\* + ? {n} {n,} {n,m}

^ $ \任何元字符 任何字符

|