**一.正则表达式**

**()分组 []代表字符类，参数之间为或者的关系 【^】代表字符类取反 {}参数 为重复几遍**

**一个 - 代表范围**

**一个 . 代表一个除了回车和换行符之外的所有字符 等效于[^\r\n]，（注意与\*的区别和含义）**

1.匹配自身的字符：大部分的普通字符匹配自身，普通字符指的是字母、数字或者空格。

以下字符可以匹配自身：

所有的ASCII字母：A~Z a~z

ASCII数字：0~9

空格字符

所有的非ASCII字符

下面这些ASCII标点字符：

！ “ # % ‘ ， / : ; < = > @ - ` ~

根据方言的不同，括号会有所差别。在一些正则表达式方言中，括号匹配自

身，但有些则不然。

2.元字符：

剩下的ASCII字符是保留字符，需要匹配的话必须对其进行转义。

对于非控制字符，可以使用反斜杠转义，也有用于控制字符的表示法：

\r 回车

\n 换行

\t 制表符

\f 换页

\a 警报

\ b 回退

\e 逃逸

3.通配符：

.点号 匹配出了换行之外的任何单个字符，点号不匹配的字符只有回车和换页。

[.\s]字符类:可以匹配包含换行的任意字符。

4.量词：

注意\*在这里是量词，不是充当通配符，充当通配符的是 .

? 出现0次或1次（最多出现1次）

+ 出现1次或多次（至少出现1次）

\* 出现0次或多次（任意次）

{n} 出现n次

{n,m} 出现n到m次

{n,} 出现至少n次；

5.类速记符;

\b: 匹配一个词的边界（一个匹配的词的边界不包含在匹配的内容中） 单词与符号的边界

\B：符号与符号的边界、单词与单词的边界(//一个字符串的开始和结尾都被认为是非单词。)

\d匹配一个数字 等价于[0-9] 例如， /\d/ 或者 /[0-9]/ 匹配"B2 is the suite number."中的'2'

\D匹配一个非数字 等价于[^0-9] 例如， /\D/ 或者 /[^0-9]/ 匹配"B2 is the suite number."中的'B'

\s匹配一个空白字符(空格、制表符、回车或者换行) 例如, /\s\w\*/ 匹配"foo bar."中的' bar'

\S匹配一个非空白字符 例如, /\S\w\*/ 匹配"foo bar."中的' foo'

\w匹配一个单字字符（字母、数字或者下划线）。 等价于[A-Za-z0-9\_]。

\W匹配一个非单字字符。 等价于[^A-Za-z0-9\_]。

6.字符类：【】

7，分组和逆向引用;(在需要分段数据和再次结合成不同次序时，分组和逆向引用很重要)

在表达式内部使用括号，分组后就能够使用重复操作符。

如：匹配所有连续的<br>标签：(<br>)+

逆向引用：可以在表达式的后面逆向引用分组。第一对括号的匹配是\1，第二对是\2，以此类推 （如果分组是嵌套的，从左边的括号算起）

如： 查找没有属性和子元素的所有元素，就是找到以起始标签开头、结束标签结尾的所有元

素。 <([a-zA-Z]+)>[^<]\*</\1>

8.空白：

使用[\r\n(\r\n)]匹配所有的断行，不管是什么类型。

使用\s匹配所有的空白和断行。

使用^和$定位到一行的开头或者结尾。

9.选择符：|

10.贪婪和非贪婪匹配;

**默认为贪婪模式（即尽可能多的匹配），在量词后加？可设置为非贪婪模式**

console.log("AaBbAb\_AaaBbbAba".replace(/Aa+/g,0));//0BbAb\_**0**BbbAba 匹配的是Aaa，贪婪模式

console.log("AaBbAb\_AaaBbbAba".replace(/Aa+?/g,0));// 0BbAb\_0aBbbAba匹配的是aa，非贪婪模式。

11.位置：

^行首 $行尾 \b词的边界 \B非词边界的任何位置 \A文档的开头 \Z文档的结尾，除非文档以一个断行结束，在这种情况下，匹配最后一个断行前的前一个位置。

\A\s\*(<html|<HTML) 查找以<html 或者以<HTML开头因而没有DOCTYPE声明或没有字节顺序标记（BOM）的所有文档

\A\s\*(<body|<BODY) 查找以<body 或者以<BODY开头因而没有正确的html根元素的所有文档

</[hH][tT][mM][lL]>\s\*\Z 查找以</html>各个组合结尾，并有可选空白的所有文档

**12.方法：**

1.search()返回值：search 方法指明是否存在相应的匹配。

如果找到一个匹配，search方法将返回一个整数值，指明这个匹配距离字符串开

始的偏移位置。

如果没有找到匹配，则返回 -1

**注意search忽略 全局g**

2.test（）方法：检测是否存在符合要求的，存在返回true，否则返回false

3.match()方法：返回符合要求的。如果没有符合要求的，返回null

4.exec()方法：获得详细信息，返回一个有属性的数组，属性index表示匹配到的位置

5.split() 方法用于把一个字符串分割成字符串数组。String.split() 执行的操作与 [Array.join](http://www.w3school.com.cn/jsref/jsref_join.asp) 执行的操作是相反的。

//对于非全局模式下返回第一个匹配的和所有的分组项，正则对象的lastIndex不起作用

var execExp = /\d{1,2}(\d)(\d)/;

var retExp = execExp.exec("12s342dsfsf233s");

console.log(retExp instanceof Array,retExp,execExp.lastIndex);

console.log(retExp instanceof Array,retExp,execExp.lastIndex);

//true ["342", "4", "2", index: 3, input: "12s342dsfsf233s"]0:

//true ["342", "4", "2", index: 3, input: "12s342dsfsf233s"] 0

//对于全局模式下 每检测一次lastIndex增加一次，再次用此正则对象匹配时，匹配的起始点为上一次的lastIndex

var execExp2 = /\d{1,2}(\d)(\d)/g;

var ts = "12s342dsfsf233s";

console.log(execExp2.exec(ts),execExp2.lastIndex);//lastIndex为 6

console.log(execExp2.exec(ts),execExp2.lastIndex);//lastIndex为 14

7.**注意：lastIndex**

var regExp = /a/gi;//思考如果加了g，循环了若干次后是true还是false，这是为什么？test中的lastIndex

console.log(regExp.test("ab"));//true **lastIndex:1**

console.log(regExp.test("ab"));//false 为什么？**lastIndex:0**

console.log(regExp.test("ab"));//true**lastIndex:1**

console.log(regExp.test("ab"));//false 为什么？**lastIndex:0**

**8.如果前后同时存在修饰符，后面的修饰符会覆盖前面的修饰符，后面的修饰符起作用。**

例如：//（粘连sticky）修饰符

var s = 'aaa\_aa\_a';

var r1 = /a+/g;

var r2 = /a+/y;

r1.exec(s); // ["aaa"]

r2.exec(s); // ["aaa"]

r1.exec(s); // ["aa"]

r2.exec(s); // null

//上面代码有两个正则表达式，一个使用g修饰符，另一个使用y修饰符。

// 这两个正则表达式各执行了两次，第一次执行的时候，两者行为相同，剩余字符串都是\_aa\_a。

//第二次执行的时候，lastIndex=3,由于g修饰没有位置要求，所以第二次执行会返回结果，而y修饰符要求匹配必须从头部开始，第一个字符是下划线，不符合要求，所以返回null。