正则表达式

学习备注：
主要看脑图来选择表达式方式，正则可视化网站来确认。

完整教程
https://juejin.im/post/5965943ff265da6c30653879#heading-43

正则可视化（必用）
https://jex.im/regulex/#!flags=&re=%5E(a%7Cb)*%3F%24

正则测试
http://tool.chinaz.com/regex/

在线正则表达式编辑器和可视化工具:

- debuggex
- regex101
- regexper
- RegExr

正则表达式基本案例

正则表达式需要掌握两个部分，/模式/的语法，匹配字符串的方法

```
//创建检验的字符串
var str = '123ab';

//创建正则规则
var patt = /[0-9]+/;

//使用match方法，返回字符串的正则结果
alert(str.match(patt));
```

常用元字符

代码	说明
.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线
\s	匹配任意的空白符
\d	匹配数字
\b	匹配单词的开始或结束
^	匹配字符串的开始
$	匹配字符串的结束

常用限定符

代码/语法	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

常用反义词

代码/语法	说明
\W	匹配任意不是字母，数字，下划线，汉字的字符
\S	匹配任意不是空白符的字符
\D	匹配任意非数字的字符
\B	匹配不是单词开头或结束的位置
[^x]	匹配除了x以外的任意字符
[^aeiou]	匹配除了aeiou这几个字母以外的任意字符

JavaScript RegExp 对象

JavaScript RegExp 对象参考手册

正则表达式模式的基本语法（模式+修饰符）：

```
var patt=/pattern/modifiers;

//不推荐的方法：var patt=new RegExp(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

修饰符modifiers

i -- case insensitive大小写不敏感

```
模式/abc/i
匹配的字符串：abc，ABC,Abc....
```

g -- globe全局匹配，找到第一个后不停止，直接找到最后

```
模式/a/g
匹配的字符串a,aa,aaa.....
返回结果[a,a,a.....]
```

m -- multiline多行匹配

```
//查找多行匹配（\n后作为新行处理
str = 'abcggab\nabcoab';
patt = /^a/g; //a
patt = /^a/mg; //a,a
```

u -- unicode

y -- sticky

正则表达式模式pattern

精确匹配（没啥用）

直接写在//内的字符将会被精确匹配

```
/abc/
"abc" //true
"ac" //false
```

模糊匹配

纵向模糊

多个条件匹配一个字符位

. , \,[ ] -- 多个条件匹配一位

一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade

```
/a[12]b/
"ab" //false
"a1b" //true
"a2b" //true
"a3b" //false
"a12b" //false
```

【 】的子组成部分

方括号[ ]，表示一个字符的多个匹配值，规则累加

```
[abc] = a,b,c


//-范围表示法
[1-3] = 1 or 2 or 3
[a-c] = a or b or c

//^非符号
[^a] = not a
 [^abc] = not a not b not c
 ^abc = 以abc开头的行

 //特殊符号表达法，想要直接表达-和^,可以调换顺序或转义
 [-az] = [\-az] = -,a,z //[a-z]会被当成a到z，[-az]会被当作- or a or z;
 [\^abc] = ^,a,b,c
```

元字符（ [ ]的简写形式 ）

相当于简写，特殊含义的预定义字符或转义字符

预定义字符集（可以写在字符集[...]中）			
\d	数字：[0-9]	a\dc	a1c
\D	非数字：[^\d]	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符：[^\s]	a\Sc	abc
\w	单词字符：[A-Za-z0-9_]	a\wc	abc
\W	非单词字符：[^\w]	a\Wc	a c

```
\d = [0-9] 数字
\D = [^0-9] 非数字
\w = [0-9a-zA-Z_] 单词 = 数字+大小写字母+下划线
\W = [^0-9a-zA-Z_] 非单词
\s = [ \t\v\n\r\f] 空格和通配符
\S = [^ \t\v\n\r\f] 非空格和非通配符
. = [^\n\r\u2028\u2029] 非换行外全部
```

数量词（横向模糊）

数量词修饰前一个字符，匹配n次

横向{ }，*，+，？ -- 一个条件匹配n位

数量词（用在字符或(...)之后）			
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	

```
//var x = 匹配字符串出现次数
* //x>=0
+ //x>=1
? //0<=x<=1
{n} //x=n
{n,} //x>=n
{n,m} //n<=x<=m

/ab{2,3}c/
```

```
"abc" //false
"abbc" //true
"abbbc" //ture
"abbbbc" //false
```

贪婪匹配和惰性匹配

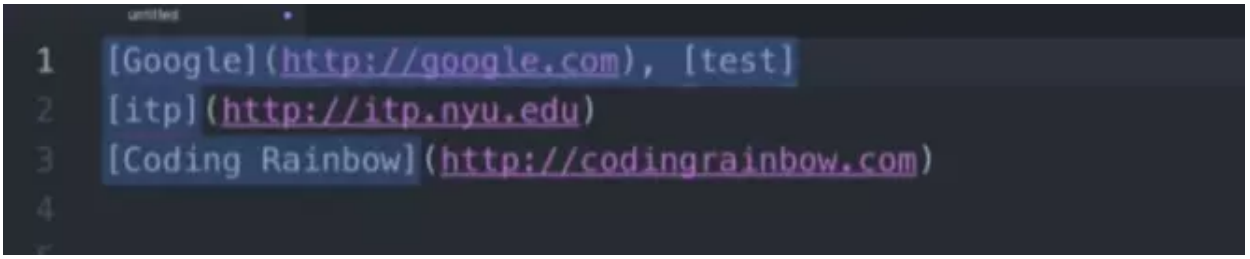
案例：

匹配markdown中的link标签，并替换为html标签

```
[google](http://google.com)
[itp](http://itp.nyu.edu)
[Coding Rainbow](http://codingrainbow.com)
```

解析： 这道题有些坑，需要慢慢来。

看到这个，第一个想考虑匹配[google]这个东西，立马想到正则表达式\[.*\]。 这个是巨大的坑，在当前来看，它的确能正确匹配到上面的三条。 但是如果文本是这样的：



看到了，第一行的内容会全部匹配下来，而不能区分[google]和[test]。 之所以这样，**是因为.是贪婪的，他表示所有，所有能匹配到的，所以当然也包括了]，一直到这一行的最后一个]，它才停止。**

所以为了让它能正确匹配，需要去掉这种贪婪的属性。这里用到？ 。 当?放在了quantifiers符号后，表示去掉贪婪属性，匹配到终止条件，即可停下。

\[.*?\]这样子，就可以将[google]和[test]分开

```
//贪婪匹配（默认）尽可能匹配最多
/\d{2,5}/
"12345" //12345
```

```
//惰性匹配 加问号,尽可能匹配少
/\d{2,5}?/
"12345" //12

 //惰性匹配就是量词后加个?
```

位置指定

边界匹配（不消耗待匹配字符串中的字符）			
^	匹配字符串开头。 在多行模式中匹配每一行的开头。	^abc	abc
$	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	abc$	abc
\A	仅匹配字符串开头。	\Aabc	abc
\Z	仅匹配字符串末尾。	abc\Z	abc
\b	匹配\w和\W之间。	a\b!bc	a!bc
\B	[^\b]	a\Bbc	abc

锚字符Anchors

^ $ \b \B (?=p) (?!p)

```
^ 匹配行的开始
/^abc/ 本行开始后面是abc的匹配


$ 匹配行的结束
/abc$/行结束是abc的匹配

\b 单词边界
具体就是 \w 与 \W 之间的位置，也包括 \w 与 ^ 之间的位置，和 \w 与 $ 之间的位置
 备注：可用于在单词间增加逗号
  "[JS] Lesson_01.mp4".replace(/\b/g, '#');
// => "[#JS#] #Lesson_01#.#mp4#


\B 非单词边界

```

断言

```
x(?=p) p前面的位置，p为任意正则positive lookahead
//先查找正则p，在p的前面找正则x
//或者，在p的位置前面，增加字符，其余相仿


x(?!p) 非p前面的位置negative lookahead
//先查找非p正则，在非p的前面找正则x



 //es6
 (?<=p)x p后面的位置positive lookbehind
 (?<!p)x 非p后面的位置negative lookbehind
```

^的两种用法

```
//1，用在[ ]以外的地方，匹配行的开始
/^A/
"And" //true
"DNA" //false
//m模式下，支持\n
/^A/m
//
"B and \nA" //true



 //2，用在[ ]内，等同于not
 /[a]/
 "a1a2a3" //a,a,a
 /[^a]/
 "a1a2a3" //1,2,3
```

逻辑、分组			
|	|代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果|没有被包括在()中，则它的范围是整个正则表达式。	abc|def	abc def
(...)	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式中的|仅在该组中有效。	(abc){2} a(123|456)c	abcabc a456c
(?P<name>...)	分组，除了原有的编号外再指定一个额外的别名。	(?P<id>abc){2}	abcabc
\<number>	引用编号为<number>的分组匹配到的字符串。	(\d)abc\1	1abc1 5abc5
(?P=name)	引用别名为<name>的分组匹配到的字符串。	(?P<id>\d)abc(?P=id)	1abc1 5abc5

分支 |

分支表示任意一个均可

```
/aaa|bbb|ccc/
"aaabbbccc" //aaa,bbb,ccc

//分支是惰性的
/good|goodbye/
"goodbye" //good
//重新排序
/goodbye|good/
"goodbye" //goodbye

 //只要满足第一个条件，就不会去匹配第二个条件
```

分组括号（ ）

```
//主要任务是分组
/(a|b)c/
"ac" //true
"bc" //true
"abc" //false

//配合量词
/(ab){2}/
"abab" //true

 //分组捕获,将有子匹配任务
 /(a|b)(a|b)c/
```

```
"abc"; //"abc","a","b"
//分组捕获后的值存储在全局对象RegExp.$0-Reg.Exp$9,用于用正则替换字串场合,str.replce(patt,'$2 $1')
$0 = "abc"
$1 = "a"
$2 = "b"

//\1-\9,是()的序号，\1为第一个()，将分组的值在正则中引用，叫做反向引用
```

反向引用 \1-\n,

```
//先通过括号分组
/(a|b)(a|b)(a|b)/
"aaa","bbb","aba"... //true
//通过\1....\9,调用分组,\1等于第一个()匹配上的值
/(a|b)a\1/
"aaa","bab"//true
"baa","aab" //false

//引用不存在的分组会变成转义
 /(abc)\1\2/
 \1 = (abc)
 \2 = 2
```

括号嵌套怎么办？

以左括号（开括号）为准。比如：

```
var regex = /^((\d)(\d(\d)))\1\2\3\4$/;
var string = "1231231233";
console.log( regex.test(string) ); // true
console.log( RegExp.$1 ); // 123
console.log( RegExp.$2 ); // 1
console.log( RegExp.$3 ); // 23
console.log( RegExp.$4 ); // 3
```

我们可以看看这个正则匹配模式：

- 第一个字符是数字，比如说1，
- 第二个字符是数字，比如说2，
- 第三个字符是数字，比如说3，
- 接下来的是\1，是第一个分组内容，那么看第一个开括号对应的分组是什么，是123，
- 接下来的是\2，找到第2个开括号，对应的分组，匹配的内容是1，
- 接下来的是\3，找到第3个开括号，对应的分组，匹配的内容是23，
- 最后的是\4，找到第3个开括号，对应的分组，匹配的内容是3。