

1. 正则表达式的概念

正则表达式(regular expression)描述了一种字符串匹配的模式。这种模式，我们可以理解成是一种“规则”。根据这种规则再去匹配符合条件的结果，而匹配的过程就是检索，查找、提取的过程。

正则表达式只能对字符串进行操作。这一点需要明确知道。

正则表达式的“祖先”可以一直上溯至对人类神经系统如何工作的早期研究。Warren McCulloch 和 Walter Pitts 这两位神经生理学家研究出一种数学方式来描述这些神经网络。

1956 年, 一位叫 Stephen Kleene 的数学家在 McCulloch 和 Pitts 早期工作的基础上, 发表了一篇标题为“神经网络事件的表示法”的论文, 引入了正则表达式的概念。正则表达式就是用来描述他称为“正则集的代数”的表达式, 因此采用“正则表达式”这个术语。随后, 发现可以将这一工作应用于使用 Ken Thompson 的计算搜索算法的一些早期研究, Ken Thompson 是 Unix 的主要发明人。正则表达式的第一个实用应用程序就是 Unix 中的 qed 编辑器。

剩下的就是众所周知的历史了。从那时起直至现在正则表达式都是基于文本的编辑器和搜索工具中的一个重要部分。

基于不同平台或者是语言的 regular expression 并不相同, 以下说的都是基于 JavaScript 语言的正则表达式。

2. RegExp 对象

2.1 RegExp 对象

RegExp 是 JavaScript 中内置的正则对象，通过以下方法均可以创建一个正则对象的实例。

- **构造函数**

```
var pattern = new RegExp('regexp','modifier');
```

regexp : 匹配的模式，也就是上文指的正则规则，实际上我们编写正则表达式的过程，都是对“规则”的定义过程。

modifier : 正则实例的修饰符。

示例：

```
var pattern = new RegExp('world','i');  
var reg = new RegExp('name','ig');
```

- **字面量**

```
var pattern = /regexp/modifier;
```

示例：

```
var pattern = /world/i;  
var reg = /name/ig;
```

说明：

字面量与构造函数的区别就是，字面量更类似于定义一个变量，采用赋值的方式。这种方式创建正则实例是我们更加推荐使用的方式。

注意的是，字面量方式下正则实例创建的格式，即用一对斜杠 (//) 来包裹定义的规则，规则不能用定界符括起来，随后在附加上修饰符。

2.2 修饰符说明

“修饰符” 其含义类似于正则实例的附加属性。用于说明正则规则适用匹配的范围。

i : 表示区分大小写字母匹配。

m : 表示多行匹配。

g : 表示全局匹配。

在非全局的情况下，正则会根据指定的“规则”从左至右对字符串进行匹

配，一旦规则匹配完，便会停止匹配操作，返回结果。

在全局的的情况下，正则会根据指定的“规则”从左至右对字符串进行匹

配，一旦规则匹配完，便会在当前字符串匹配位置

重新使用“规则”继续向下匹配，一直到字符串匹配完成。这也是下文中，

我们会提到 `lastIndex` 属性存在的必要性。

2.3 RegExp 对象方法

以下方法都是 `RegExp` 对象的内置方法，这些方法可以根据指定的规则来对字符串进行匹配，查找等。

可以支持正则表达式的方法，不仅仅只有 `RegExp` 对象才有，实际上 `String` 对象也具有同样功能的方法。这些我们下文会讲解到。

- **test()**

`test` 方法用于测试正则规则在指定的字符串中是否具有符合的匹配结果，如果匹配到则返回 `true`，否则返回 `false`。

示例:

```
var pattern = /a/;  
console.log(pattern.test('edcba')) // => true
```

当 `test` 方法在全局模式下被多次执行的时候，每次执行的返回值会因为匹配结果的不同而不同，但在实际的应用中，

我只需要进行一次匹配即可。

```
var pattern = /a/g;
```

```
console.log(pattern.test( "fdafd" )) //true
```

```
console.log(pattern.test( "fdafd" )) //false
```

解释：因为正则/a/g 有全局 g 标志，所以同样的正则实例 pattern 第一次从 0 开始查找，

结果为 true ，第二次从 3 开始查找，结果为 false。所以第二次调用若想返回相同结果，

需要手动设置 test 方法起始查找位置：pattern.lastIndex = 0;

在匹配到字符串的末尾后，会从头到尾再重新循环匹配。

示例：

```
1 var pattern = /a/g;
2 console.log(pattern.test('edcba')) // => true 第一次执行。
3 console.log(pattern.test('edcba')) // => false 第二次执行。
4 console.log(pattern.test('edcba')) // => true 第三次执行。从头到尾重新循环
  执行。
```

- **exec()**

在非全局的匹配模式下，一旦匹配到符合规则的结果，便会停止执行。

在全局模式下，当匹配到符合规则的结果也会停止执行，但是若多次重复执行

exec 方法，则会根据 lastIndex 属性的值为锚点依次向后匹配，

exec 方法可以返回匹配的结果，以及结果在字符串中的索引和下一次匹配的

起始位置。如果正则表达式没有匹配到结果，

那么返回的值就是 null。

exec 方法在匹配到的情况下，返回的是一个数组，数组的格式如下：

```
1 var result = /a/.exec('abcdaefga');
2 result[0] // -> 当前匹配到的结果。
3 result.input // -> 进行匹配操作的字符串。
4 result.index // -> 当前匹配结果首字母在字符串中的索引
```

如果存在分组匹配的情况下：



result[1] -> 表示第一个分组匹配到的结果。

result[2] -> 表示第二个分组匹配到的结果。

...

result[n] -> 表示第 n 个分组匹配到的结果。

示例：

```

1 var str = '2012 google';
2 var pattern = /(\d{4})\s(\w+)/;
3 var result = pattern.exec(str);
4 console.log(result[0]);
5 console.log(result[1]);
6 console.log(result[2]);

```

如果想使用 exec 方法对字符串中的某个结果进行全面匹配，那么正则表达式必须要开启全局模式。在非全局的模式下，

exec 方法一旦匹配到结果，便会停止执行。

示例：

```
1 var pattern = /a/g;
2 while(result = pattern.exec('abababab')) {
3     console.log(result+'index:'+ result.index + ' input:'+ result.input);
4 }
```

- **compile()**

compile 可以重新指定正则实例的规则与修饰符。

示例：

```
var pattern = /res/i;
pattern.compile('rp','g') -> /rp/g
```

2.4 RegExp 对象的属性

- **静态属性**

静态属性是 RegExp 这个内置对象的固有属性。访问这些静态属性，不需要进行声明实例化，而是直接调用。

调用格式：RegExp.attribute

下面所有属性的说明，就以：

```
var desc = 'Hello, everyone. My name is gtshen';  
reg = /na(.*?) /g;  
reg.test(desc);
```

这段代码测试为例，进行说明：

- input

功能：返回当前要匹配的字符串

示例：console.log('input:'+RegExp.input) // -> 'Hello,everyone.My name
is gtshen'

短名：RegExp.\$_;

注意：opera 低版本并不支持。

- lastMatch

功能：最后一次匹配到的子串结果，需要开启修饰符-g。

示例：console.log('lastMatch:'+RegExp.lastMatch) // -> nam

短名：RegExp['\$&'];

注意：opera 低版本并不支持。

- lastParen

功能：最后一次分组中捕获到的内容。需要开启修饰符-g。

示例：console.log('lastParen:'+RegExp.lastParen) // -> 'm';

短名：RegExp['\$+'];

注意：opera 低版本并不支持。

- leftContext

功能：以当前匹配到的子串为上下文，返回之前的子串。

示例：`console.log('leftContext:'+RegExp.leftContext) // ->`

`'Hello,everyone.My ';`

短名：`RegExp['$&`'];`

- rightContext

功能：以当前匹配到的子串为上下文，返回之后的子串。

示例：`console.log('rightContext:'+RegExp.rightContext) // -> 'e is`

`gtshen';`

短名：`RegExp['$\\'];`

- multiline

功能：是否支持多行。返回值为 boolean 值，true 表示支持，false 表示不支持。

示例：`console.log('multiline:'+RegExp.multiline);`

短名：`RegExp['$*'];`

注意：IE 并不支持。

- \$1 - \$9

功能：返回 1 - 9 个分组的值。

示例：`console.log('$1:'+ RegExp.$1) // -> 'm'`

* 注意的是“RegExp”指的是最近一次在程序运行中进行匹配操作的正则实例对象。

• 实例属性

实例属性，就是指必须经过声明或实例化后的正则表达式对象方能调用的属性。

下面所有属性的说明，就以下代码为例进行说明：

```
var pattern = /\w?/g;
```

- ignoreCase

功能：修饰符属性，只读。判断指定正则表达式是否开启大小写，返回值布尔值。

示例：pattern.ignoreCase

- global

功能：修饰符属性，只读。判断指定正则表达式是否开启大小写，返回值布尔值。

示例：pattern.global

- multiline

功能：修饰符属性，只读。判断指定正则表达式是否开启多行模式，返回值布尔值。

示例：pattern.multiline

- lastIndex

功能：返回当前匹配结果下一次匹配的起始位置。也可以手动设置lastIndex 的值，用于设置指定的起始位置。

示例：

```
1 var desc = 'hellow,hi,oh';
2 pattern.test(desc)
3 console.log(pattern.lastIndex) //-> 0;
4 pattern.lastIndex = 2;
5 console.log(pattern.lastIndex) // -> 2;
```

- source

功能：返回正则表达式对象的正则规则。

示例：console.log(pattern.source) // -> \w?

* ignoreCase、global、multiline 等均为只读属性，不能直接修改正则表达式的修饰符。

* lastIndex 的值，只能在 RegExp 内置方法中才能够使用到。

3. JS 的正则基础语法

正则表达式是有两种字符模式组成：“普通字符”、“元字符”。通过这两种字符模式的结合使用，可以编写出符合我们要求的正则规则。

普通字符：

就是由显性的没有特殊含义的打印字符和没有指定为元字符的非打印字符组成。

显性的打印字符，它是实际可见的，例如 0-9，a-z，除了可打印的字符，还存在一些非打印的字符，例如 ASCII 码值在 0-31 的为控制字符，

无法显示和打印，但实际存在。

元字符：元字符更接近计算机语言中的变量含义，它可以代表某种特殊的含义，并且会根据使用场合不同，其具体的含义也不尽相同。

普通字符很好理解，按照计算机语言中的字面量去理解就可以，我们着重研究“元字符”。

元字符，根据其特性与含义，又可以分为以下几个小类去说明：

3.1 特殊字符一览表：

特殊符号	功能说明
\	转义运算符， <u>正则中优先级最高的运算符。</u>
\cX	匹配与x相对应的控制字符(control+x)
\xHH	匹配指定的两位十六进制数，例如:/[\x00 - \xff]/。
\uNNNN	匹配指定的四位unicode码，例如:/[\u0000-\uFFFF]/。
\Numbe	匹配重复的分组。
\f	匹配换页符
\n	匹配换行符
\r	匹配回车符
\v	匹配垂直制表符
\t	匹配水平制表符
\s	匹配空字符，等价于[\f\r\t\n\x0B]， <u>在JavaScript也可以直接使用空格进行匹配。</u>
\S	匹配非空字符。等价于[^\f\r\t\n\x0B]。
\w	匹配数字字母及_，等价于[a-zA-Z0-9_]
\W	匹配非数字字母及_，等价于[^a-zA-Z0-9_]
\d	匹配数字字符，等价于[0-9]
\D	匹配非数字字符，等价于[^0-9]
\b	匹配单词边界
\B	匹配非单词边界
^	匹配指定起始的字符
\$	匹配指定 结尾 的字符
.	匹配除换行符回车符外的任意 字符 ，等价于[^\n\r]
?	匹配0个或1个
*	匹配0个或多个
+	匹配1个或多个
	或运算符， <u>正则中优先级最低的运算符。</u>
[]	字符类
()	分组
{}	量词

3.2 转义运算符

功能：对元字符进行转义，使其转换为普通字符。

示例：

```
1 var pattern = /\[/;
2 var str = '[';
3 console.log(pattern.test(str)) // -> true;
```

3.3 量词

? : 表示匹配 0 次或 1 次

+ : 表示匹配 1 次或多次

{n} : 表示匹配 n 次

{n,m} : 表示匹配 n 到 m 次

{n,} : 表示至少匹配 n 次

3.4 边界

\b : 匹配单词边界，用于匹配一个整体的单词时使用。

\B : 匹配非单词边界。

^ : 强制首匹配，以指定规则开始的字符，避免前导的其它字符。

\$: 强制尾匹配，以指定规则结束的字符，避免后导的其它字符。

3.5 类

“类”是具有相同特征的集合，是一个泛指。

- 字符类

[abc] : 只根据区间内的内容进行匹配。

- 范围类

[a-zA-Z0-9] : 匹配大小写字符 a-z 以及数组 0-9 的范围

- 反向类

[^a-z] : 取反匹配。匹配不在这个区间范围内的内容，

示例：

```
1 var str = 'google';
2 var pattern = /[gle]/;
3 console.log(pattern.test(str))
```

3.6 贪婪模式与非贪婪模式

贪婪模式会按照匹配规则尽可能的去匹配，一直到匹配失败。

非贪婪模式则会根据最小匹配规则去匹配。

以匹配规则为例来说，贪婪是指匹配所有符合规则的结果，按照最大可能的结果去匹配，而非贪婪则只要有匹配结果便会停止匹配。

从量词上看： $*+?$ 都是贪婪匹配，因为 $*+?$ 都会按照匹配规则尽可能的去匹配。

例如 $*$ 匹配零个或多个，但实际匹配上，会以多个的情况为优先匹配。

贪婪匹配	非贪婪匹配
$+$ 一次或多次	$+?$ 只匹配一次
$?$ 0 次或一次	$??$ 匹配 0 次
$*$ 0 次或多次	$*?$ 匹配 0 次
$\{n\}$ n 次	$\{n\}?$ 按照 n 次匹配
$\{n,\}$ 最少 n 次	$\{n,\}?$ 按照 n 次匹配
$\{n,m\}$ n 到 m 次	$\{n,m\}?$ 按照 n 次匹配

取消贪婪模式也很简单，在原有的基础上附加上一个 $?$ 号，这时就改为非贪婪模式，一旦条件满足，就不再往下匹配。

示例：

```
1 var str = 'aaa',
2 pattern1 = /a*/,
3 pattern2 = /a*?/;
4 pattern1.exec(str); // -> aaa
5 pattern2.exec(str); // -> ''
```

3.7 分组

分组就是把要匹配的结果作为一个组一个整体来看待。因此只要是属于该分组的信息，都要放在 $()$ 符号内。

• $()$ 捕获性分组

示例：

```
1 var str = '123abc';
2 var pattern = /(\d{3})(\w+)/;
3 pattern.test(str);
4 console.log(RegExp.$1) //-> 123
5 console.log(RegExp.$2) //-> abc
```

- **(?:) 非捕获性分组**


示例：

```
1 var str = '123abc';
2 var pattern = /(\d{3})(?:\w+)/;
3 pattern.test(str);
4 console.log(RegExp.$1) //-> 123
5 console.log(RegExp.$2) //-> '';
```


- **((...)) 嵌套分组**

说明：嵌套分组从外向内获取

示例：



```
1 var str = 'abc';
2 var pattern = /(a?(b?(c?)))/;
3 pattern.test(str);
4 console.log(RegExp.$1) //-> abc
5 console.log(RegExp.$2) //-> bc
6 console.log(RegExp.$3) //-> c
```



3.8 前瞻

对于人的习惯而言，我们认为我们在看一段话的顺序是从前到后的阅读顺序，但是对于计算机而言，

已经识别到的则是“后”，而未能或者即将识别的区域则是“前”。所以正则前瞻的本质含义就是判断前方的内容是否符合匹配条件。

如果符合匹配条件，那么“正向前瞻”就为 true，否则为 false。如果不符合匹配条

件，那么“负向前瞻”为 true，否则为 false。

下面解释用到的“后续内容”字眼，是从我们人的习惯出发加以说明的。

• (?:) 正向前瞻

说明：“正向前瞻”是根据后续内容是否符合匹配条件来返回匹配的结果。

示例：

```
1 var str = 'google';
2 var pattern = /goo(?:gle)/;
3 console.log(pattern.exec(str)[0]); // -> goo
```

从示例我们可以看出，goo 是我们匹配的结果，而(?:gle)则是后续内容的判断条件，一旦字符串中有符合 goo 后续为 gle 的字符串，那么就返回匹配结果。goo。

• (?!) 负向前瞻

说明：“负向前瞻”是根据后续内容是否不符合条件来返回匹配的结果。

“负向前瞻”与“正向前瞻”含义与功能相反。

示例：

```
1 var str = 'google';
2 var pattern1 = /goo(?:!gle)/;
3 var pattern2 = /goo(?:!ogl)/;
4 pattern1.test(str) // -> false 因为 gle 是符合 goo 的后续内容的，所以返回 false，条件不成立。
5 pattern2.test(str) // -> true 因为 ogl 不是 goo 的后续内容，所以返回 true，条件成立。
```

* 前瞻是一种零宽度断言匹配。

世界本来没有路，只是人走多了，自然就有了路，同样的道理，零宽度断言，虽然听上去怪怪的，还有点反人类的感觉，

但是多说说也就习惯了。毕竟只是一个取名的问题而已，无需较真。所谓的“零宽度”，指的就是其匹配的位置不会发生改变。

而“断言”，则用来声明一个应该为真的事实。因此我们前面学习到的“正

向前瞻” 也叫 “零宽度正预测先行断言” 。

用一句话概括其功能：

前瞻（零宽度断言）匹配不会捕获或保存匹配结果，更加不会改变匹配时的匹配位置(lastIndex)。

匹配的起始位置也是它匹配的结束位置。所以匹配位置固定不变。

示例：

```
1 var str = 'find123';  
2 var pattern = /(?=find)\d+/  
3 alert(pattern.test(str)) // -> false
```

就如我们前面说到的那样,前瞻是一种零宽度匹配,而零宽度的特点概要来说,就是不捕获匹配内容,不改变匹配位置。

首先,整个匹配顺序是从左到右,左为后,右为前。在位置 0 处正则的控制权交由(?=find)去匹配,此时在位置 0 处向右尝试检索 find 字符串,

并且成功检索到所以匹配成功,但并不捕获内容,也不改变匹配位置,所以从位置 0 处开始匹配,也从位置 0 处结束匹配,

接着匹配控制权交由\d+,然后从位置 0 处开始匹配(因为之前的零宽度并未改变匹配位置),

再向右检索数字的时候,发现位置 0 为 f,位置 1 位 i... 所以匹配失败,最终返回 false。

```
1 var str = 'abcfind123';  
2 var pattern = /abc(?=find)find\d+/  
3 alert(pattern.test(str)) // -> true
```

这个示例的返回结果为 true,让我们仔细分析以下,首先,从左向右匹配,位置 0,位置 1,

位置 2 等处的字符串 abc 正好与正则表达式吻合，所以匹配成功，接着从位置 3 处将匹配控制权交由(?=find)，

此时正则表达式从位置 3 处开始向右检索字符串 find 检索到所以匹配成功，然后在位置 3 处结束检索，

将控制权分别交由位置 3 的 find 到 位置 6 的 d 这一过程都是匹配成功的，再接着将匹配控制权交由\d+，

从位置 7 开始到结束都是数字，所以最终的匹配结果为 true。

3.9 运算符

相同优先级的从左到右进行运算，不同优先级的运算先高后低。各种操作符的优先级从高到低如下：

操作符	描述
\	转义符
(), (?:), (?!), []	分组和类
*, +, ?, {n}, {n, }, {n, m}	量词
^, \$, \anymetacharacter	边界以及字符顺序
	“或”操作

3.10 其它

• 控制字符匹配

\cX

X 是 a-Z 之间的字符，\cX 则表示匹配 control+X 形式的控制字符。

但是在实际的测试中，发现通过\cX 匹配任意字符都是为空，因此猜测该正则浏览器中并不支持，或者是所应用的场合环境比较特殊。

下面是具体的测试代码：


```

1 var pattern = '',
2   str = '',
3   c = '',
4   r = '';
5 for(var i=0;i<255;i++){
6   str+=String.fromCharCode(i);
7 }
8 for(var i=65;i<90;i++){
9   c = String.fromCharCode(i);
10  pattern = new RegExp('\\c'+c);
11  r = str.match(pattern);
12  r?console.log(r,c,pattern):'';
13 }

```

• 修饰符 m 对位置元字符的影响。

我们知道修饰符 m 表示多行匹配，而 ^\$ 表示匹配的起始于结束位置，那么如果这两种情况如果一起使用，或者分别使用，有什么区别呢？

^\$m：匹配每一行的起始于结束位置。

^\$：只匹配整个字符串的起始于结束位置。

示例：

```

1 var str = 'World\nWorld\nWorld',
2   pattern1 = /^World$/g;
3   pattern2 = /^World$/gm;
4 str.match(pattern1); //-> null 匹配整个字符串开始到结束
5 str.match(pattern2); // -> [ "World", "World", "World" ] 匹配每行开始和结束

```

由此我们可以看出对于正则来说，字符串如果存在多行，那么每一行都会存在 ^\$ 匹配操作。

• \Number 指定重复分组

在分组匹配捕获时，也可以在正则规则中指定 \n (注:n:1-9) 的形式来代指某个分组的值。

示例：

```

1 var str = 'abcabc',
2 pattern = /(.)b(.)\1b\2/;
3 pattern.test(str); // -> true

```

示例：找出重复最多的字符。

```

1 /*
2  * 方法说明
3  * split('') 将字符串通过“空字符串”来拆分为数组。
4  * sort() 将数组元素的值按照 ASCII 码进行排序。
5  * join('') 将数组转换为字符串。
6
7 */
8
9 var str = 'abcababaaacddcd',
10     pattern = /(\w)\1+/g,
11     len = 0,
12     result = '',
13     data = '';
14
15 str = str.split('').sort().join(''); //通过这些步骤，我们可以得到一个具有
    相同字符 Y 一起排列出现的新字符串。aaaaaabbccccddd
16 while(result = pattern.exec(str)) {
17
18     if(result[0].length > len) {
19         len = result[0].length;
20         data = RegExp.$1;
21     }
22 }

```

• 换行匹配

在正则中字符串的换行是用\n进行表示的。

示例：

```

1 var str = '1.baidu\n2.google\n3.bing';
2 var pattern1 = /\d/g; //此种方式也可以做到，但是建议使用标准的换行模式。
3 var pattern2 = /\d/gm;
4 document.write(str.replace(pattern1, '#'));
5 document.write(str.replace(pattern2, '#'));

```

4. 支持正则的 String 方法

除了 RegExp 对象具有支持正则表达式的方法外，字符串 String 对象也具有可以支持正则表达式作为参数进行匹配筛选的方法。

4.1 replace()

格式: str.replace(pattern, given);

功能：根据匹配规则 pattern 来用指定的内容 given 去替换 str 或其部分字符。

其中 pattern 可以是 str 中的部分字符也可以是一个正则表达式。

示例：


```
1 var str = 'i see you See you';
2 var pattern = /you/;
3 str.replace(pattern, '*'); // -> i see * See you
```

注意：given 所代表替换的指定内容，既可以是字符串，也可以是一个回调函数。

```
1 var str = 'i see you See you';
2 var pattern = /\byou\b/g;
3 str.replace(pattern, function(result, [$1...$9], index, input) {});
```

- result：表示当前匹配的结果
- [\$1 - \$9]：存在分组的情况下，表示当前 1 - 9 个分组的内容。
- index：当前匹配内容首字母在字符串中索引。
- input：进行匹配的原字符串。

示例：敏感词过滤

```

1 var str = '世界上最遥远的距离 不是生与死的距离 而是我就站在你的面前 你却不知道我爱你。',
2     st = ['死', '爱', '距离'],
3     pattern = '',
4     alt = '';
5
6 for(var i=0; i<st.length; i++) {
7
8
9     pattern = new RegExp(st[i], 'g');
10
```

```

11     for(var j=0;j<st[i].length;j++){
12         alt+='*';
13     }
14
15     str = str.replace(pattern, alt);
16     alt = '';
17 }

```



示例：回调函数使用

```

1 var str = '2016/10/29';
2 var pattern = /(\d+) (\w+)/g;
3 var data = str.replace(pattern, function(result, $1, $2) {
4     return $1+'.'+$2;
5 });

```

* replace 方法只会返回原字符串被操作后的副本，并不会对原字符串进行改动。

4.2 match()

格式：str.match(pattern)

功能：

match 在功能上与正则对象自带的方法 exec 很类似。

match 根据匹配规则 pattern 匹配指定的字符串 str ,如果匹配成功则返回一个数组格式的结果用于存放匹配文本有关的信息，

如果没有匹配到则返回 null。

```
var result = str.match(pattern);
```

格式如下：

result --> 当前的匹配结果。

result[0] --> 从数组中读取当前的匹配结果。

result[1]

.... ---> 指定分组中捕获的内容。

result[n+1]

result.index --> 当前匹配结果的首字母在给定字符串中的索引。

result.input --> 进行匹配操作的给定字符串。

如果 match 的匹配规则是一个正则，并且具有全局 g 属性，那么 match 返回的匹配结果，便是一个包含所有匹配结果的纯数组。

```
1 var str = 'abcdabceda';
2 var result = str.match(/a/g);
3 result --> [a, a, a]
4 result.input --> undefined
5 result.index --> undefined
```

4.3 split()

格式：str.split(pattern,length)

功能：

根据规则 pattern 将字符串拆分为数组，拆分后的数组并不包含作为拆分依据的那个参数。

默认情况下是空字符进行拆分，也就是每个任意的字符作为一个数组元素。

pattern 参数，可以是正则表达式，也可以是单纯的字符或字符串。

length 参数，用于设置拆分数组后数组最大的长度（即数组元素的个数）。

缺省该项，表示将字符全部拆分为数组。

示例：

```
1 var str = 'hellow word!';
2 str.spilit(''); // --> '' 空字符（并非空格字符）["h", "e", "l", "l", "o", "w", " ", "w", "o", "r", "l", "d", "!"]
3 str.split('',5) // --> ["h", "e", "l", "l", "o"]
4 str.split(/\o/g) // --> ["hell", "w w", "rld!"]
```

4.4 search();

格式：str.search(pattern)

功能：根据匹配规则 pattern 在字符串中检索指定的结果，如果检索到则返回该结

果首字母在原字符中的索引，否则返回-1。其功能类似于 indexOf,

只是 indexOf 并不支持正则匹配。

示例：

```
1 var str = 'hellow world!';
2 str.search('o') // --> 4
3 str.search('x') // --> -1
```

注意：该方法忽略全局修饰符 g，也不支持 lastIndex 也就意味着它不能被多次调用，一旦检索到结果，便会停止检索。

5. 常用正则表达式收集

1. 匹配任意字符

正则：/[^]/

2. 匹配 ASCII 码范围

规则：\x00 表示十进制 0，\xff 表示十进制 255,0-255 正好是 ASCII 码表可表示的范围。

正则：/[[]\x00-\xff/[]]g

说明：同理，/[[]^\x00-\xff/[]]g 则表示匹配汉字或者是全角字符。

3. 匹配汉字

规则：\u4e00：在 Unicode 码中汉字的开始编码，\u9fa5：在 Unicode 码中汉字的结束编码。

正则：/[[]^\u4e00-\u9fa5/[]]+\$/[/]

4. 手机号码检测

规则：[3|4|5|7|8]：手机号 11 位，但是第二位数字只有这几种固定的网段。

正则：`/^1[3|4|5|7|8]\d{9}$/`

5. 邮政编码

规则：邮政编码必须为数字，长度为 6 位且第一位不能为 0，示例：224000

正则：`/[1-9]\d{5}/`

6. 电子邮件

规则：以@为上下文表示，左侧只能是数字，字母及下划线组成。

而右侧又以.符号为上下文，左侧与上述一样，而右侧只能为字母结尾。

正则：`/^([\w\-\.\+])@([\w\-\+])\.[a-zA-Z]{2,4})$/`

7. 匹配前后空格

正则：`/^\s+|\s+$/g` //以空格开头或以空格结尾，或|的运算符最低

8. QQ 号匹配

规则：首位不能为 0，位数 5-12 位。

正则：`/^[1-9]\d{4,11}$/;`

9. 匹配网址 url 的正则表达式

正则：`[a-zA-z]+://[^\s]*`

10. 匹配国内电话号码

正则：`\d{3}-\d{8}|\d{4}-\d{7}`

11. 匹配国内身份证号码

规则：简单的身份证匹配，根据位数以及结尾是否有字母。

正则：`/^\d{15}(\d{2}[A-Za-z0-9])?$/`

12. 匹配 IP 地址

正则：`/^(\d+)\.(\d+)\.(\d+)\.(\d+)$/g`

13. 验证是否含有`^%&',;=?$\\"等字符`

正则：`/[\\^%&',;=?$\\x22]+/`

14. 匹配空行的正则表达式

正则：`\\n[\\s|]*\\r`

15. 数学正则

整数：`/^[\\-\\+]?\\d+$/`

浮点数：`/^[\\-\\+]?\\d+(\\.\\d+)?$/`

Chinese：`/^[\\u0391-\\uFFE5]+$/`

实数：`^[\\-\\+]?\\d+(\\.\\d+)?$`

附录：参考页面

PS：感谢这些作者们！

<http://javascript.ruanyifeng.com/stdlib/regexp.html#toc13> (讲解很到位的正则教程)

<http://www.cnblogs.com/52cik/p/js-regular-control-character.html> (关于控制字符的代码测试实例)

<http://baike.baidu.com/view/1112575.htm?fr=aladdin> (关于控制字符的说明)

<http://www.cnblogs.com/hustskyking/archive/2013/06/04/RegExp.html> (正则表达式 30 分钟入门教程 PS:内容很丰富，适合全面了解正则的教程，但是里面讲到的很多功能并不能被浏览器实现)

<http://www.cnblogs.com/dwlsxj/p/Reg.html> (对零宽度讲解很好的文章)

<http://www.cnblogs.com/moqing/p/5665126.html>