RegExp 对象

`RegExp`对象提供正则表达式的功能。

概述

正则表达式（regular expression）是一种表达文本模式（即字符串结构）的方法，有点像字符串的模板，常常用来按照“给定模式”匹配文本。比如，正则表达式给出一个 Email 地址的模式，然后用它来确定一个字符串是否为 Email 地址。JavaScript 的正则表达式体系是参照 Perl 5 建立的。

新建正则表达式有两种方法。一种是使用字面量，以斜杠表示开始和结束。

```javascript
var regex = /xyz/;
```

另一种是使用`RegExp`构造函数。

```javascript
var regex = new RegExp('xyz');
```

上面两种写法是等价的，都新建了一个内容为`xyz`的正则表达式对象。它们的主要区别是，第一种方法在引擎编译代码时，就会新建正则表达式，第二种方法在运行时新建正则表达式，所以前者的效率较高。而且，前者比较便利和直观，所以实际应用中，基本上都采用字面量定义正则表达式。

`RegExp`构造函数还可以接受第二个参数，表示修饰符（详细解释见下文）。

```javascript
var regex = new RegExp('xyz', 'i');
// 等价于
var regex = /xyz/i;
```

上面代码中，正则表达式`/xyz/`有一个修饰符`i`。

实例属性

正则对象的实例属性分成两类。

一类是修饰符相关，用于了解设置了什么修饰符。

- `RegExp.prototype.ignoreCase`：返回一个布尔值，表示是否设置了`i`修饰符。
- `RegExp.prototype.global`：返回一个布尔值，表示是否设置了`g`修饰符。

- `RegExp.prototype.multiline`：返回一个布尔值，表示是否设置了`m`修饰符。
- `RegExp.prototype.flags`：返回一个字符串，包含了已经设置的所有修饰符，按字母排序。

上面四个属性都是只读的。

```javascript
var r = /abc/igm;

r.ignoreCase // true
r.global // true
r.multiline // true
r.flags // 'gim'
```

另一类是与修饰符无关的属性，主要是下面两个。

- `RegExp.prototype.lastIndex`：返回一个整数，表示下一次开始搜索的位置。该属性可读写，但是只在进行连续搜索时有意义，详细介绍请看后文。
- `RegExp.prototype.source`：返回正则表达式的字符串形式（不包括反斜杠），该属性只读。

```javascript
var r = /abc/igm;

r.lastIndex // 0
r.source // "abc"
```

实例方法

RegExp.prototype.test()

正则实例对象的`test`方法返回一个布尔值，表示当前模式是否能匹配参数字符串。

```javascript
/cat/.test('cats and dogs') // true
```

上面代码验证参数字符串之中是否包含`cat`，结果返回`true`。

如果正则表达式带有`g`修饰符，则每一次`test`方法都从上一次结束的位置开始向后匹配。

```javascript
var r = /x/g;
var s = '_x_x';

r.lastIndex // 0
r.test(s) // true

r.lastIndex // 2
r.test(s) // true
```

```
r.lastIndex // 4
r.test(s) // false
```

上面代码的正则表达式使用了`g`修饰符，表示是全局搜索，会有多个结果。接着，三次使用`test`方法，每一次开始搜索的位置都是上一次匹配的后一个位置。

带有`g`修饰符时，可以通过正则对象的`lastIndex`属性指定开始搜索的位置。

```javascript
var r = /x/g;
var s = '_x_x';

r.lastIndex = 4;
r.test(s) // false

r.lastIndex // 0
r.test(s)
```

上面代码指定从字符串的第五个位置开始搜索，这个位置为空，所以返回`false`。同时，`lastIndex`属性重置为`0`，所以第二次执行`r.test(s)`会返回`true`。

注意，带有`g`修饰符时，正则表达式内部会记住上一次的`lastIndex`属性，这时不应该更换所要匹配的字符串，否则会有一些难以察觉的错误。

```javascript
var r = /bb/g;
r.test('bb') // true
r.test('-bb-') // false
```

上面代码中，由于正则表达式`r`是从上一次的`lastIndex`位置开始匹配，导致第二次执行`test`方法时出现预期以外的结果。

`lastIndex`属性只对同一个正则表达式有效，所以下面这样写是错误的。

```javascript
var count = 0;
while (/a/g.test('babaa')) count++;
```

上面代码会导致无限循环，因为`while`循环的每次匹配条件都是一个新的正则表达式，导致`lastIndex`属性总是等于0。

如果正则模式是一个空字符串，则匹配所有字符串。

```javascript
```

```
new RegExp('').test('abc')
// true
```

RegExp.prototype.exec()

正则实例对象的`exec`方法，用来返回匹配结果。如果发现匹配，就返回一个数组，成员是匹配成功的子字符串，否则返回`null`。

```javascript
var s = '_x_x';
var r1 = /x/;
var r2 = /y/;

r1.exec(s) // ["x"]
r2.exec(s) // null
```

上面代码中，正则对象`r1`匹配成功，返回一个数组，成员是匹配结果；正则对象`r2`匹配失败，返回`null`。

如果正则表示式包含圆括号（即含有“组匹配”），则返回的数组会包括多个成员。第一个成员是整个匹配成功的结果，后面的成员就是圆括号对应的匹配成功的组。也就是说，第二个成员对应第一个括号，第三个成员对应第二个括号，以此类推。整个数组的`length`属性等于组匹配的数量再加1。

```javascript
var s = '_x_x';
var r = /_(x)/;

r.exec(s) // ["_x", "x"]
```

上面代码的`exec`方法，返回一个数组。第一个成员是整个匹配的结果，第二个成员是圆括号匹配的结果。

`exec`方法的返回数组还包含以下两个属性：

- `input`：整个原字符串。
- `index`：整个模式匹配成功的开始位置（从0开始计数）。

```javascript
var r = /a(b+)a/;
var arr = r.exec('_abbba_aba_');

arr // ["abbba", "bbb"]

arr.index // 1
arr.input // "_abbba_aba_"
```

```

上面代码中的`index`属性等于1，是因为从原字符串的第二个位置开始匹配成功。

如果正则表达式加上`g`修饰符，则可以使用多次`exec`方法，下一次搜索的位置从上一次匹配成功结束的位置开始。

```javascript
var reg = /a/g;
var str = 'abc_abc_abc'

var r1 = reg.exec(str);
r1 // ["a"]
r1.index // 0
reg.lastIndex // 1

var r2 = reg.exec(str);
r2 // ["a"]
r2.index // 4
reg.lastIndex // 5

var r3 = reg.exec(str);
r3 // ["a"]
r3.index // 8
reg.lastIndex // 9

var r4 = reg.exec(str);
r4 // null
reg.lastIndex // 0
```

上面代码连续用了四次`exec`方法，前三次都是从上一次匹配结束的位置向后匹配。当第三次匹配结束以后，整个字符串已经到达尾部，匹配结果返回`null`，正则实例对象的`lastIndex`属性也重置为`0`，意味着第四次匹配将从头开始。

利用`g`修饰符允许多次匹配的特点，可以用一个循环完成全部匹配。

```javascript
var reg = /a/g;
var str = 'abc_abc_abc'

while(true) {
  var match = reg.exec(str);
  if (!match) break;
  console.log('#' + match.index + ':' + match[0]);
}
// #0:a
// #4:a
// #8:a
```
```

上面代码中，只要`exec`方法不返回`null`，就会一直循环下去，每次输出匹配的位置和匹配的文本。

正则实例对象的`lastIndex`属性不仅可读，还可写。设置了`g`修饰符的时候，只要手动设置了`lastIndex`的值，就会从指定位置开始匹配。

字符串的实例方法

字符串的实例方法之中，有4种与正则表达式有关。

- `String.prototype.match()`：返回一个数组，成员是所有匹配的子字符串。
- `String.prototype.search()`：按照给定的正则表达式进行搜索，返回一个整数，表示匹配开始的位置。
- `String.prototype.replace()`：按照给定的正则表达式进行替换，返回替换后的字符串。
- `String.prototype.split()`：按照给定规则进行字符串分割，返回一个数组，包含分割后的各个成员。

String.prototype.match()

字符串实例对象的`match`方法对字符串进行正则匹配，返回匹配结果。

```javascript
var s = '_x_x';
var r1 = /x/;
var r2 = /y/;

s.match(r1) // ["x"]
s.match(r2) // null
```

从上面代码可以看到，字符串的`match`方法与正则对象的`exec`方法非常类似：匹配成功返回一个数组，匹配失败返回`null`。

如果正则表达式带有`g`修饰符，则该方法与正则对象的`exec`方法行为不同，会一次性返回所有匹配成功的结果。

```javascript
var s = 'abba';
var r = /a/g;

s.match(r) // ["a", "a"]
r.exec(s) // ["a"]
```

设置正则表达式的`lastIndex`属性，对`match`方法无效，匹配总是从字符串的第一个字符开始。

```javascript
```

```
var r = /a|b/g;
r.lastIndex = 7;
'xaxb'.match(r) // ['a', 'b']
r.lastIndex // 0
```

上面代码表示，设置正则对象的`lastIndex`属性是无效的。

String.prototype.search()

字符串对象的`search`方法，返回第一个满足条件的匹配结果在整个字符串中的位置。如果没有任何匹配，则返回`-1`。

```javascript
'_x_x'.search(/x/)
// 1
```

上面代码中，第一个匹配结果出现在字符串的`1`号位置。

String.prototype.replace()

字符串对象的`replace`方法可以替换匹配的值。它接受两个参数，第一个是正则表达式，表示搜索模式，第二个是替换的内容。

```javascript
str.replace(search, replacement)
```

正则表达式如果不加`g`修饰符，就替换第一个匹配成功的值，否则替换所有匹配成功的值。

```javascript
'aaa'.replace('a', 'b') // "baa"
'aaa'.replace(/a/, 'b') // "baa"
'aaa'.replace(/a/g, 'b') // "bbb"
```

上面代码中，最后一个正则表达式使用了`g`修饰符，导致所有的`b`都被替换掉了。

`replace`方法的一个应用，就是消除字符串首尾两端的空格。

```javascript
var str = '  #id div.class  ';

str.replace(/^\s+|\s+$/g, '')
// "#id div.class"
```

`replace`方法的第二个参数可以使用美元符号`$`，用来指代所替换的内容。

- `$&`：匹配的子字符串。
- `` $` ``：匹配结果前面的文本。
- `$'`：匹配结果后面的文本。
- `$n`：匹配成功的第`n`组内容，`n`是从1开始的自然数。
- `$$`：指代美元符号`$`。

```javascript
'hello world'.replace(/(\w+)\s(\w+)/, '$2 $1')
// "world hello"

'abc'.replace('b', '[$`-$&-$\']')
// "a[a-b-c]c"
```

上面代码中，第一个例子是将匹配的组互换位置，第二个例子是改写匹配的值。

`replace`方法的第二个参数还可以是一个函数，将每一个匹配内容替换为函数返回值。

```javascript
'3 and 5'.replace(/[0-9]+/g, function (match) {
  return 2 * match;
})
// "6 and 10"

var a = 'The quick brown fox jumped over the lazy dog.';
var pattern = /quick|brown|lazy/ig;

a.replace(pattern, function replacer(match) {
  return match.toUpperCase();
});
// The QUICK BROWN fox jumped over the LAZY dog.
```

作为`replace`方法第二个参数的替换函数，可以接受多个参数。其中，第一个参数是捕捉到的内容，第二个参数是捕捉到的组匹配（有多少个组匹配，就有多少个对应的参数）。此外，最后还可以添加两个参数，倒数第二个参数是捕捉到的内容在整个字符串中的位置（比如从第五个位置开始），最后一个参数是原字符串。下面是一个网页模板替换的例子。

```javascript
var prices = {
  'p1': '$1.99',
  'p2': '$9.99',
  'p3': '$5.00'
};

var template = '<span id="p1"></span>'
  + '<span id="p2"></span>'
  + '<span id="p3"></span>';

template.replace(
```

```
  /(<span id=")(.*?)(">)(<\/span>)/g,
  function(match, $1, $2, $3, $4){
    return $1 + $2 + $3 + prices[$2] + $4;
  }
);
// "<span id="p1">$1.99</span><span id="p2">$9.99</span><span id="p3">$5.00</span>"
```

上面代码的捕捉模式中，有四个括号，所以会产生四个组匹配，在匹配函数中用`$1`到`$4`表示。匹配函数的作用是将价格插入模板中。

String.prototype.split()

字符串对象的`split`方法按照正则规则分割字符串，返回一个由分割后的各个部分组成的数组。

```javascript
str.split(separator, [limit])
```

该方法接受两个参数，第一个参数是正则表达式，表示分隔规则，第二个参数是返回数组的最大成员数。

```javascript
// 非正则分隔
'a,  b,c, d'.split(',')
// [ 'a', '  b', 'c', ' d' ]

// 正则分隔，去除多余的空格
'a,  b,c, d'.split(/, */)
// [ 'a', 'b', 'c', 'd' ]

// 指定返回数组的最大成员
'a,  b,c, d'.split(/, */, 2)
[ 'a', 'b' ]
```

上面代码使用正则表达式，去除了子字符串的逗号后面的空格。

```javascript
// 例一
'aaa*a*'.split(/a*/)
// [ '', '*', '*' ]

// 例二
'aaa**a*'.split(/a*/)
// ["", "*", "*", "*"]
```

上面代码的分割规则是0次或多次的`a`，由于正则默认是贪婪匹配，所以例一的第一个分隔符是`aaa`，第二个分割符是`a`，将字符串分成三个部分，包含开始处的空字符串。例二的第一个分隔符是`aaa`，第二个分隔符是0个`a`（即空字符），第三个分隔符是`a`，所以将字符串分成四个部分。

如果正则表达式带有括号，则括号匹配的部分也会作为数组成员返回。

```javascript
'aaa*a*'.split(/(a*)/)
// [ '', 'aaa', '*', 'a', '*' ]
```

上面代码的正则表达式使用了括号，第一个组匹配是`aaa`，第二个组匹配是`a`，它们都作为数组成员返回。

匹配规则

正则表达式的规则很复杂，下面一一介绍这些规则。

字面量字符和元字符

大部分字符在正则表达式中，就是字面的含义，比如`/a/`匹配`a`，`/b/`匹配`b`。如果在正则表达式之中，某个字符只表示它字面的含义（就像前面的`a`和`b`），那么它们就叫做“字面量字符”（literal characters）。

```javascript
/dog/.test('old dog') // true
```

上面代码中正则表达式的`dog`，就是字面量字符，所以`/dog/`匹配`old dog`，因为它就表示`d`、`o`、`g`三个字母连在一起。

除了字面量字符以外，还有一部分字符有特殊含义，不代表字面的意思。它们叫做“元字符”（metacharacters），主要有以下几个。

（1）点字符（.）

点字符（`.`）匹配除回车（`\r`）、换行(`\n`)、行分隔符（`\u2028`）和段分隔符（`\u2029`）以外的所有字符。注意，对于码点大于`0xFFFF`字符，点字符不能正确匹配，会认为这是两个字符。

```javascript
/c.t/
```

上面代码中，`c.t`匹配`c`和`t`之间包含任意一个字符的情况，只要这三个字符在同一行，比如`cat`、`c2t`、`c-t`等等，但是不匹配`coot`。

（2）位置字符

位置字符用来提示字符所处的位置，主要有两个字符。

- `^` 表示字符串的开始位置
- `$` 表示字符串的结束位置

```javascript
// test必须出现在开始位置
/^test/.test('test123') // true

// test必须出现在结束位置
/test$/.test('new test') // true

// 从开始位置到结束位置只有test
/^test$/.test('test') // true
/^test$/.test('test test') // false
```

（3）选择符（`|`）

竖线符号（`|`）在正则表达式中表示“或关系”（OR），即`cat|dog`表示匹配`cat`或`dog`。

```javascript
/11|22/.test('911') // true
```

上面代码中，正则表达式指定必须匹配`11`或`22`。

多个选择符可以联合使用。

```javascript
// 匹配fred、barney、betty之中的一个
/fred|barney|betty/
```

选择符会包括它前后的多个字符，比如`/ab|cd/`指的是匹配`ab`或者`cd`，而不是指匹配`b`或者`c`。如果想修改这个行为，可以使用圆括号。

```javascript
/a( |\t)b/.test('a\tb') // true
```

上面代码指的是，`a`和`b`之间有一个空格或者一个制表符。

其他的元字符还包括`\`、`\*`、`+`、`?`、`()`、`[]`、`{}`等，将在下文解释。

转义符

正则表达式中那些有特殊含义的元字符，如果要匹配它们本身，就需要在它们前面要加上反斜杠。比如要匹配`+`，就要写成`\+`。

```javascript
/1+1/.test('1+1')
// false

/1\+1/.test('1+1')
// true
```

上面代码中，第一个正则表达式之所以不匹配，因为加号是元字符，不代表自身。第二个正则表达式使用反斜杠对加号转义，就能匹配成功。

正则表达式中，需要反斜杠转义的，一共有12个字符：`^`、`.`、`[`、`$`、`(`、`)`、`|`、`*`、`+`、`?`、`{`和`\`。需要特别注意的是，如果使用`RegExp`方法生成正则对象，转义需要使用两个斜杠，因为字符串内部会先转义一次。

```javascript
(new RegExp('1\+1')).test('1+1')
// false

(new RegExp('1\\+1')).test('1+1')
// true
```

上面代码中，`RegExp`作为构造函数，参数是一个字符串。但是，在字符串内部，反斜杠也是转义字符，所以它会先被反斜杠转义一次，然后再被正则表达式转义一次，因此需要两个反斜杠转义。

特殊字符

正则表达式对一些不能打印的特殊字符，提供了表达方法。

- `\cX` 表示`Ctrl-[X]`，其中的`X`是A-Z之中任一个英文字母，用来匹配控制字符。
- `[\b]` 匹配退格键(U+0008)，不要与`\b`混淆。
- `\n` 匹配换行键。
- `\r` 匹配回车键。
- `\t` 匹配制表符 tab（U+0009）。
- `\v` 匹配垂直制表符（U+000B）。
- `\f` 匹配换页符（U+000C）。
- `\0` 匹配`null`字符（U+0000）。
- `\xhh` 匹配一个以两位十六进制数（`\x00`-`\xFF`）表示的字符。

- `\uhhhh` 匹配一个以四位十六进制数（`\u0000`-`\uFFFF`）表示的 Unicode 字符。

字符类

字符类（class）表示有一系列字符可供选择，只要匹配其中一个就可以了。所有可供选择的字符都放在方括号内，比如`[xyz]` 表示`x`、`y`、`z`之中任选一个匹配。

```javascript
/[abc]/.test('hello world') // false
/[abc]/.test('apple') // true
```

上面代码中，字符串`hello world`不包含`a`、`b`、`c`这三个字母中的任一个，所以返回`false`；字符串`apple`包含字母`a`，所以返回`true`。

有两个字符在字符类中有特殊含义。

（1）脱字符（^）

如果方括号内的第一个字符是`[^]`，则表示除了字符类之中的字符，其他字符都可以匹配。比如，`[^xyz]`表示除了`x`、`y`、`z`之外都可以匹配。

```javascript
/[^abc]/.test('bbc news') // true
/[^abc]/.test('bbc') // false
```

上面代码中，字符串`bbc news`包含`a`、`b`、`c`以外的其他字符，所以返回`true`；字符串`bbc`不包含`a`、`b`、`c`以外的其他字符，所以返回`false`。

如果方括号内没有其他字符，即只有`[^]`，就表示匹配一切字符，其中包括换行符。相比之下，点号作为元字符（`.`）是不包括换行符的。

```javascript
var s = 'Please yes\nmake my day!';

s.match(/yes.*day/) // null
s.match(/yes[^]*day/) // [ 'yes\nmake my day']
```

上面代码中，字符串`s`含有一个换行符，点号不包括换行符，所以第一个正则表达式匹配失败；第二个正则表达式`[^]`包含一切字符，所以匹配成功。

> 注意，脱字符只有在字符类的第一个位置才有特殊含义，否则就是字面含义。

（2）连字符（-）

某些情况下，对于连续序列的字符，连字符（`-`）用来提供简写形式，表示字符的连续范围。比如，`[abc]`可以写成`[a-c]`，`[0123456789]`可以写成`[0-9]`，同理`[A-Z]`表示26个大写字母。

```javascript
/a-z/.test('b') // false
/[a-z]/.test('b') // true
```

上面代码中，当连字号（dash）不出现在方括号之中，就不具备简写的作用，只代表字面的含义，所以不匹配字符`b`。只有当连字号用在方括号之中，才表示连续的字符序列。

以下都是合法的字符类简写形式。

```javascript
[0-9.,]
[0-9a-fA-F]
[a-zA-Z0-9-]
[1-31]
```

上面代码中最后一个字符类`[1-31]`，不代表`1`到`31`，只代表`1`到`3`。

连字符还可以用来指定 Unicode 字符的范围。

```javascript
var str = "\u0130\u0131\u0132";
/[\u0128-\uFFFF]/.test(str)
// true
```

上面代码中，`\u0128-\uFFFF`表示匹配码点在`0128`到`FFFF`之间的所有字符。

另外，不要过分使用连字符，设定一个很大的范围，否则很可能选中意料之外的字符。最典型的例子就是`[A-z]`，表面上它是选中从大写的`A`到小写的`z`之间52个字母，但是由于在 ASCII 编码之中，大写字母与小写字母之间还有其他字符，结果就会出现意料之外的结果。

```javascript
/[A-z]/.test('\\') // true
```

上面代码中，由于反斜杠（'\\'）的ASCII码在大写字母与小写字母之间，结果会被选中。

预定义模式

预定义模式指的是某些常见模式的简写方式。

- `\d` 匹配0-9之间的任一数字，相当于`[0-9]`。
- `\D` 匹配所有0-9以外的字符，相当于`[^0-9]`。

- `\w` 匹配任意的字母、数字和下划线，相当于`[A-Za-z0-9_]`。
- `\W` 除所有字母、数字和下划线以外的字符，相当于`[^A-Za-z0-9_]`。
- `\s` 匹配空格（包括换行符、制表符、空格符等），相等于`[ \t\r\n\v\f]`。
- `\S` 匹配非空格的字符，相当于`[^ \t\r\n\v\f]`。
- `\b` 匹配词的边界。
- `\B` 匹配非词边界，即在词的内部。

下面是一些例子。

```javascript
// \s 的例子
/\s\w*/.exec('hello world') // [" world"]

// \b 的例子
/\bworld/.test('hello world') // true
/\bworld/.test('hello-world') // true
/\bworld/.test('helloworld') // false

// \B 的例子
/\Bworld/.test('hello-world') // false
/\Bworld/.test('helloworld') // true
```

上面代码中，`\s`表示空格，所以匹配结果会包括空格。`\b`表示词的边界，所以`world`的词首必须独立（词尾是否独立未指定），才会匹配。同理，`\B`表示非词的边界，只有`world`的词首不独立，才会匹配。

通常，正则表达式遇到换行符（`\n`）就会停止匹配。

```javascript
var html = "<b>Hello</b>\n<i>world!</i>";

/.*/.exec(html)[0]
// "<b>Hello</b>"
```

上面代码中，字符串`html`包含一个换行符，结果点字符（`.`）不匹配换行符，导致匹配结果可能不符合原意。这时使用`\s`字符类，就能包括换行符。

```javascript
var html = "<b>Hello</b>\n<i>world!</i>";

/[\S\s]*/.exec(html)[0]
// "<b>Hello</b>\n<i>world!</i>"
```

上面代码中，`[\S\s]`指代一切字符。

重复类

模式的精确匹配次数，使用大括号（`{}`）表示。`{n}`表示恰好重复`n`次，`{n,}`表示至少重复`n`次，`{n,m}`表示重复不少于`n`次，不多于`m`次。

```javascript
/lo{2}k/.test('look') // true
/lo{2,5}k/.test('looook') // true
```

上面代码中，第一个模式指定`o`连续出现2次，第二个模式指定`o`连续出现2次到5次之间。

量词符

量词符用来设定某个模式出现的次数。

- `?` 问号表示某个模式出现0次或1次，等同于`{0, 1}`。
- `*` 星号表示某个模式出现0次或多次，等同于`{0,}`。
- `+` 加号表示某个模式出现1次或多次，等同于`{1,}`。

```javascript
// t 出现0次或1次
/t?est/.test('test') // true
/t?est/.test('est') // true

// t 出现1次或多次
/t+est/.test('test') // true
/t+est/.test('ttest') // true
/t+est/.test('est') // false

// t 出现0次或多次
/t*est/.test('test') // true
/t*est/.test('ttest') // true
/t*est/.test('tttest') // true
/t*est/.test('est') // true
```

贪婪模式

上一小节的三个量词符，默认情况下都是最大可能匹配，即匹配直到下一个字符不满足匹配规则为止。这被称为贪婪模式。

```javascript
var s = 'aaa';
s.match(/a+/) // ["aaa"]
```

上面代码中，模式是`/a+/`，表示匹配1个`a`或多个`a`，那么到底会匹配几个`a`呢？因为默认是贪婪模式，会一直匹配到字符`a`不出现为止，所以匹配结果是3个`a`。

如果想将贪婪模式改为非贪婪模式，可以在量词符后面加一个问号。

```javascript
var s = 'aaa';
s.match(/a+?/) // ["a"]
```

上面代码中，模式结尾添加了一个问号`/a+?/`，这时就改为非贪婪模式，一旦条件满足，就不再往下匹配。

除了非贪婪模式的加号，还有非贪婪模式的星号（`*`）和非贪婪模式的问号（`?`）。

- `+?`：表示某个模式出现1次或多次，匹配时采用非贪婪模式。
- `*?`：表示某个模式出现0次或多次，匹配时采用非贪婪模式。
- `??`：表格某个模式出现0次或1次，匹配时采用非贪婪模式。

```javascript
'abb'.match(/ab*b/) // ["abb"]
'abb'.match(/ab*?b/) // ["ab"]

'abb'.match(/ab?b/) // ["abb"]
'abb'.match(/ab??b/) // ["ab"]
```

修饰符

修饰符（modifier）表示模式的附加规则，放在正则模式的最尾部。

修饰符可以单个使用，也可以多个一起使用。

```javascript
// 单个修饰符
var regex = /test/i;

// 多个修饰符
var regex = /test/ig;
```

（1）g 修饰符

默认情况下，第一次匹配成功后，正则对象就停止向下匹配了。`g`修饰符表示全局匹配（global），加上它以后，正则对象将匹配全部符合条件的结果，主要用于搜索和替换。

```javascript
```

```
var regex = /b/;
var str = 'abba';

regex.test(str); // true
regex.test(str); // true
regex.test(str); // true
```

上面代码中，正则模式不含`g`修饰符，每次都是从字符串头部开始匹配。所以，连续做了三次匹配，都返回`true`。

```javascript
var regex = /b/g;
var str = 'abba';

regex.test(str); // true
regex.test(str); // true
regex.test(str); // false
```

上面代码中，正则模式含有`g`修饰符，每次都是从上一次匹配成功处，开始向后匹配。因为字符串`abba`只有两个`b`，所以前两次匹配结果为`true`，第三次匹配结果为`false`。

（2）i 修饰符

默认情况下，正则对象区分字母的大小写，加上`i`修饰符以后表示忽略大小写（ignoreCase）。

```javascript
/abc/.test('ABC') // false
/abc/i.test('ABC') // true
```

上面代码表示，加了`i`修饰符以后，不考虑大小写，所以模式`abc`匹配字符串`ABC`。

（3）m 修饰符

`m`修饰符表示多行模式（multiline），会修改`^`和`$`的行为。默认情况下（即不加`m`修饰符时），`^`和`$`匹配字符串的开始处和结尾处，加上`m`修饰符以后，`^`和`$`还会匹配行首和行尾，即`^`和`$`会识别换行符（`\n`）。

```javascript
/world$/.test('hello world\n') // false
/world$/m.test('hello world\n') // true
```

上面的代码中，字符串结尾处有一个换行符。如果不加`m`修饰符，匹配不成功，因为字符串的结尾不是`world`；加上以后，`$`可以匹配行尾。

```javascript
```

```
/^b/m.test('a\nb') // true
```

上面代码要求匹配行首的`b`，如果不加`m`修饰符，就相当于`b`只能处在字符串的开始处。加上`m`修饰符以后，换行符`\n`也会被认为是一行的开始。

组匹配

（1）概述

正则表达式的括号表示分组匹配，括号中的模式可以用来匹配分组的内容。

```javascript
/fred+/.test('fredd') // true
/(fred)+/.test('fredfred') // true
```

上面代码中，第一个模式没有括号，结果`+`只表示重复字母`d`，第二个模式有括号，结果`+`就表示匹配`fred`这个词。

下面是另外一个分组捕获的例子。

```javascript
var m = 'abcabc'.match(/(.)b(.)/);
m
// ['abc', 'a', 'c']
```

上面代码中，正则表达式`/(.)b(.)/`一共使用两个括号，第一个括号捕获`a`，第二个括号捕获`c`。

注意，使用组匹配时，不宜同时使用`g`修饰符，否则`match`方法不会捕获分组的内容。

```javascript
var m = 'abcabc'.match(/(.)b(.)/g);
m // ['abc', 'abc']
```

上面代码使用带`g`修饰符的正则表达式，结果`match`方法只捕获了匹配整个表达式的部分。这时必须使用正则表达式的`exec`方法，配合循环，才能读到每一轮匹配的组捕获。

```javascript
var str = 'abcabc';
var reg = /(.)b(.)/g;
while (true) {
  var result = reg.exec(str);
  if (!result) break;
  console.log(result);
}
// ["abc", "a", "c"]
```

```
// ["abc", "a", "c"]
```

正则表达式内部，还可以用`\n`引用括号匹配的内容，`n`是从1开始的自然数，表示对应顺序的括号。

```javascript
/(.)b(.)\1b\2/.test("abcabc")
// true
```

上面的代码中，`\1`表示第一个括号匹配的内容（即`a`），`\2`表示第二个括号匹配的内容（即`c`）。

下面是另外一个例子。

```javascript
/y(..)(.)\2\1/.test('yabccab') // true
```

括号还可以嵌套。

```javascript
/y((..)\2)\1/.test('yabababab') // true
```

上面代码中，`\1`指向外层括号，`\2`指向内层括号。

组匹配非常有用，下面是一个匹配网页标签的例子。

```javascript
var tagName = /<([^>]+)>[^<]*<\/\1>/;

tagName.exec("<b>bold</b>")[1]
// 'b'
```

上面代码中，圆括号匹配尖括号之中的标签，而`\1`就表示对应的闭合标签。

上面代码略加修改，就能捕获带有属性的标签。

```javascript
var html = '<b class="hello">Hello</b><i>world</i>';
var tag = /<(\w+)([^>]*)>(.*?)<\/\1>/g;

var match = tag.exec(html);

match[1] // "b"
match[2] // " class="hello""
match[3] // "Hello"
```

```
match = tag.exec(html);

match[1] // "i"
match[2] // ""
match[3] // "world"
```

（2）非捕获组

`(?:x)`称为非捕获组（Non-capturing group），表示不返回该组匹配的内容，即匹配的结果中不计入这个括号。

非捕获组的作用请考虑这样一个场景，假定需要匹配`foo`或者`foofoo`，正则表达式就应该写成`/(foo){1, 2}/`，但是这样会占用一个组匹配。这时，就可以使用非捕获组，将正则表达式改为`/(?:foo){1, 2}/`，它的作用与前一个正则是一样的，但是不会单独输出括号内部的内容。

请看下面的例子。

```javascript
var m = 'abc'.match(/(?:.)b(.)/);
m // ["abc", "c"]
```

上面代码中的模式，一共使用了两个括号。其中第一个括号是非捕获组，所以最后返回的结果中没有第一个括号，只有第二个括号匹配的内容。

下面是用来分解网址的正则表达式。

```javascript
// 正常匹配
var url = /(http|ftp):\/\/([^/\r\n]+)(\/[^\r\n]*)?/;

url.exec('http://google.com/');
// ["http://google.com/", "http", "google.com", "/"]

// 非捕获组匹配
var url = /(?:http|ftp):\/\/([^/\r\n]+)(\/[^\r\n]*)?/;

url.exec('http://google.com/');
// ["http://google.com/", "google.com", "/"]
```

上面的代码中，前一个正则表达式是正常匹配，第一个括号返回网络协议；后一个正则表达式是非捕获匹配，返回结果中不包括网络协议。

（3）先行断言

`x(?=y)`称为先行断言（Positive look-ahead），`x`只有在`y`前面才匹配，`y`不会被计入返回结果。比如，要匹配后面跟着百分号的数字，可以写成`/\d+(?=%)/`。

“先行断言”中，括号里的部分是不会返回的。

```javascript
var m = 'abc'.match(/b(?=c)/);
m // ["b"]
```

上面的代码使用了先行断言，`b`在`c`前面所以被匹配，但是括号对应的`c`不会被返回。

（4）先行否定断言

`x(?!y)`称为先行否定断言（Negative look-ahead），`x`只有不在`y`前面才匹配，`y`不会被计入返回结果。比如，要匹配后面跟的不是百分号的数字，就要写成`/\d+(?!%)/`。

```javascript
/\d+(?!\.)/.exec('3.14')
// ["14"]
```

上面代码中，正则表达式指定，只有不在小数点前面的数字才会被匹配，因此返回的结果就是`14`。

“先行否定断言”中，括号里的部分是不会返回的。

```javascript
var m = 'abd'.match(/b(?!c)/);
m // ['b']
```

上面的代码使用了先行否定断言，`b`不在`c`前面所以被匹配，而且括号对应的`d`不会被返回。

参考链接

- Axel Rauschmayer, [JavaScript: an overview of the regular expression API](http://www.2ality.com/2011/04/javascript-overview-of-regular.html)
- Mozilla Developer Network, [Regular Expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions)
- Axel Rauschmayer, [The flag /g of JavaScript’s regular expressions](http://www.2ality.com/2013/08/regexp-g.html)
- Sam Hughes, [Learn regular expressions in about 55 minutes](http://qntm.org/files/re/re.html)