# 目标

# 学习目标

- 了解正则表达式概述,什么是正则表达式,正则表达式目的是什么
- 了解正则表达式在Javascript中的使用
- 掌握正则表达式中的特殊字符
- 掌握正则表达式中的替换

# 正则表达式概述

## 什么是正则表达式

正则表达式（Regular Expression 有规律的表达式,因为正则是写//里面的,所以也人喜欢叫写在注释里面的表达式）是用于匹配字符串中字符组合的模式。在JavaScript中，正则表达式也是对象。

正则表通常被用来检索、替换那些符合某个模式（规则）的文本，例如验证表单：用户名表单只能输入英文字母、数字或者下划线， 昵称输入框中可以输入中文(匹配)。此外，正则表达式还常用于过滤掉页面内容中的一些敏感词(替换)，或从字符串中获取我们想要的特定部分(提取)等 。


其他语言也会使用正则表达式，本阶段我们主要是利用JavaScript 正则表达式完成表单验证。

## 正则表达式的特点

1. 灵活性、逻辑性和功能性非常的强。
2. 可以迅速地用极简单的方式达到字符串的复杂控制。
3. 对于刚接触的人来说，比较晦涩难懂。比如：^\w+([-+.]\w+)@\w+([-.]\w+).\w+([-.]\w+)*$ 验证邮箱的表达式
4. 实际开发,一般都是直接复制写好的正则表达式. 但是要求会使用正则表达式并且根据实际情况修改正则表达式. 比如用户名: /^[a-z0-9_-]

{3,16}$/

# 正则表达式在js中的使用

## 正则表达式的创建

在 JavaScript 中，可以通过**两种**方式创建一个正则表达式。

### 方式一：通过调用RegExp对象的构造函数创建

```
var 变量名 = new RegExp(/表达式/);
```

### 方式二：利用字面量创建 正则表达式

```
var 变量名 = /表达式/;
```

## 测试正则表达式

test() 正则对象方法，用于检测字符串是否符合该规则，该对象会返回 true 或 false，其参数是测试字符串。

```
regexObj.test(str)

说明:
1.regexObj 是写的正则表达式
2.str 我们要测试的文本
3. test方法 就是检测str文本是否符合我们写的正则表达式规范
```

### 举例:

```
var reg = /123/;
console.log(reg.test(123));//匹配字符中是否出现123 出现结果为true
console.log(reg.test('abc'));//匹配字符中是否出现abc 未出现结果为false
```

```
// 在 JavaScript 中，可以通过两种方式创建一个正则表达式。
// 方式一：通过调用RegExp对象的构造函数创建    RegExp是内置对象
// var 变量名 = new RegExp(/表达式/);
var reg1 = new RegExp(/abc/);
console.dir( reg1 );

// 方式二：利用字面量创建 正则表达式   推荐使用,更加简单
// var 变量名 = /表达式/;
var reg2 = /abc/;
console.dir( reg2 );
console.log("");


// 测试正则表达式
// test() 正则对象方法，用于检测字符串是否符合该规则，该对象会返回 true 或 false，其参数是测试字符串。
// 语法:
// regexObj.test(str)


// 说明:
// 1.regexObj 是写的正则表达式
// 2.str 我们要测试的文本
// 3.test方法 就是检测str文本是否符合我们写的正则表达式规范


//    /abc/  只要含有abc的字符串就返回true
console.log( reg2.test("123abc456") );// true
console.log( reg2.test("ab123c") );// false
console.log( reg2.test("abc789abc") );// true
console.log( reg2.test("qwezxc") );// false
```

# 正则表达式中的特殊字符

## 正则表达式的组成

一个正则表达式可以由简单的字符构成，比如 /abc/，也可以是简单和特殊字符的组合，比如 /ab*c/ 。其中特殊字符也被称为元字符，在正则表达式中是具有特殊意义的专用符号，如 ^ 、$ 、+、 * 、 ?等。

特殊字符非常多，可以参考：

MDN https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions

jQuery 手册：正则表达式部分（ 手册老师放在其他资料里面了 ）


| 字符 | 描述 |
| --- | --- |
| \ | 将下一个字符标记为一个特殊字符、或一个原义字符、或一个向后引用、或一个八进制转义符。例如，“n”匹配字符“n”。“\n”匹配一个换行符。串行“\\”匹配“\”而“\(”则匹配“(”。 |
| ^ | 匹配输入字符串的开始位置。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入字符串的结束位置。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。 |
| * | 匹配前面的子表达式零次或多次。例如，zo*能匹配“z”以及“zoo”。*等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。 |
| ? | 匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“does”或“does”中的“do”。?等价于{0,1}。 |
| {n} | n是一个非负整数。匹配确定的n次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。 |
| {n,} | n是一个非负整数。至少匹配n次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“fooooood”中的所有o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o*”。 |
| {n,m} | m和n均为非负整数，其中n<=m。最少匹配n次且最多匹配m次。例如，“o{1,3}”将匹配“fooooood”中的前三个o。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。 |
| ? | 当该字符紧跟在任何一个其他限制符（*,+,?，{n}，{n,}，{n,m}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+?”将匹配单个“o”，而“o+”将匹配所有“o”。 |
| . | 匹配除“\n”之外的任何单个字符。要匹配包括“\n”在内的任何字符，请使用像“(.|\n)”的模式。 |
| (pattern) | 匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到，在VBScript中使用SubMatches集合，在JScript中则使用$0…$9属性。要匹配圆括号字符，请使用“\(”或“\)”。 |
| (?:pattern) | 匹配pattern但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用或字符“(|)”来组合一个模式的各个部分是很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。 |
| (?=pattern) | 正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。 |
| (?!pattern) | 正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始 |
| (?<=pattern) | 反向肯定预查，与正向肯定预查类拟，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。 |
| (?<!pattern) | 反向否定预查，与正向否定预查类拟，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。 |

# 边界符

正则表达式中的边界符（位置符）用来提示字符所处的位置，主要有两个字符

| 边界符 | 说明 |
| --- | --- |
| ^ | 表示匹配行首的文本（以谁开始） |
| $ | 表示匹配行尾的文本（以谁结束） |

**如果 ^和 $ 在一起，表示必须是精确匹配。**

**举例:**

```
<script>
    // 边界符
    // 正则表达式中的边界符（位置符）用来提示字符所处的位置，主要有两个字符
    // ^ 表示匹配行首的文本（以谁开始）
    // $ 表示匹配行尾的文本（以谁结束）
    // 如果 ^和 $ 在一起，表示必须是精确匹配。


    var reg1 = /abc/;// 表示只要含有abc就返回true
    console.log( reg1.test("123abc") );// true
    console.log( reg1.test("abc456") );// true
    console.log( reg1.test("123abc456") );// true
    console.log( reg1.test("abc") );// true
    console.log( reg1.test("a123b456c") );// false
    console.log( reg1.test("abc456abc") );// true
    console.log( reg1.test("abcabc") );// true
    console.log("");


    var reg2 = /^abc/;// 表示需要以abc开头才返回true
    console.log( reg2.test("123abc") );// false
    console.log( reg2.test("abc456") );// true
    console.log( reg2.test("123abc456") );// false
    console.log( reg2.test("abc") );// true
    console.log( reg2.test("a123b456c") );// false
    console.log( reg2.test("abc456abc") );// true
    console.log( reg2.test("abcabc") );// true
    console.log("");


    var reg3 = /abc$/;// 表示需要以abc结尾才返回true
    console.log( reg3.test("123abc") );// true
    console.log( reg3.test("abc456") );// false
    console.log( reg3.test("123abc456") );// false
    console.log( reg3.test("abc") );// true
    console.log( reg3.test("a123b456c") );// false
    console.log( reg3.test("abc456abc") );// true
    console.log( reg3.test("abcabc") );// true
    console.log("");


    var reg4 = /^abc$/; // 精确匹配 字符串必须是abc才正确,其他都是错
    console.log( reg4.test("123abc") );// false
    console.log( reg4.test("abc456") );// false
    console.log( reg4.test("123abc456") );// false
    console.log( reg4.test("abc") );// true
    console.log( reg4.test("a123b456c") ); // false
    console.log( reg4.test("abc456abc") );// false
    console.log( reg4.test("abcabc") );// false
</script>
```

## 字符类

字符类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。

### [] 方括号

[] 方括号,表示有一系列字符可供选择，只要匹配其中一个就可以了

### 范围符-

[-] 方括号内部 范围符-

方括号内部加上 - 表示范围

可以在中括号里面加一个短横线表示一个范围 ,比如[a-z]表示a到z的26个英文字母任何一个字母

### 字符组合

```
/[a-z1-9]/.test('andy') // true
```

方括号内部可以使用字符组合，这里表示包含 a 到 z 的26个英文字母和 1 到 9 的数字都可以

### 取反符^

[^] 方括号内部 取反符^

```
/[^abc]/.test('andy') // false
```

方括号内部加上 ^ 表示取反，只要包含方括号内的字符，都返回 false 也就是 查找到任何不在方括号内的内容返回true

**比如:**

**[^abc] 匹配 表达式用于查找任何不在方括号之间的字符**

**举例:**

```
//注意：正则表达式里面写字符,不管字母还是数字还是其他符号,都不需要加引号
```

```
var reg1 = /a/;// 这个正则代表含有a就返回true
console.log( reg1.test("abc") ); // true
console.log( reg1.test("abcaaa") ); // true
console.log( reg1.test("bcd") ); // false
console.log( reg1.test("123") ); // false
console.log("");


var reg2 = /abc/;// 这个正则代表含有abc才返回true
console.log( reg2.test("a") ); // false
console.log( reg2.test("a456") ); // false
console.log( reg2.test("b123") );// false
console.log( reg2.test("c789") );// false
console.log( reg2.test("abc") );// true
console.log( reg2.test("123abc") );// true
console.log("");


/* 字符类
字符类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。 */


/* [] 方括号
[] 方括号,表示有一系列字符可供选择，只要匹配其中一个就可以了 */


var reg3 = /[abc]/;// 表示只要含有 a 或者  含有b  或者含有c 其他一个字符都返回true
console.log( reg3.test("a") ); // true
console.log( reg3.test("a456") ); // true
console.log( reg3.test("b123") );// true
console.log( reg3.test("c789") );// true
console.log( reg3.test("abc") );// true
console.log( reg3.test("123abc") );// true
console.log("");


/* 范围符-
[-] 方括号内部 范围符-
方括号内部加上 - 表示范围
可以在中括号里面加一个短横线表示一个范围 ,比如[a-z]表示a到z的26个英文字母任何一个字母 */
var reg4 = /[a-z]/;  // 只要含有a到z其中一个小写英文字母,包括a和z都返回true
console.log( reg4.test("a") );// true
console.log( reg4.test("A") );// false
console.log( reg4.test("a123") );// true
console.log( reg4.test("z123") );// true
console.log( reg4.test("12b3") );// true
console.log( reg4.test("1#2c3") );// true
console.log( reg4.test("-") );// false
console.log("");


/* 字符组合
/[a-z1-9]/.test('andy') // true


方括号内部可以使用字符组合，这里表示包含 a 到 z 的26个英文字母和 1 到 9 的数字都可以 */


var reg5 = /[a-zA-Z0-9_-]/; // 代表含有26个小写大写字母以及0~9以及下划线和短横杠其中一个都返回true
console.log( reg5.test("a") );// true
console.log( reg5.test("A") );// true
console.log( reg5.test("a123") );// true
console.log( reg5.test("z123") );// true
console.log( reg5.test("12b3") );// true
console.log( reg5.test("1#2c3") );// true
console.log( reg5.test("456-abc") );// true
console.log( reg5.test("abc_zhagnsan") );// true
console.log( reg5.test("5") );// true
console.log( reg5.test("-") );// true
console.log( reg5.test("_") );// true
console.log("");
```

```
/* 取反符^
[^] 方括号内部 取反符^
/[^abc]/.test('andy') // false
"方括号内部"加上 ^ 表示取反，只要包含方括号内的字符，都返回 false  */


// 如果^没在方括号内部,代表边界符,以什么开头
var reg6 = /^abc/; // 代表以abc开头的字符串才会返回true
console.log( reg6.test("abc123456") );// true
console.log( reg6.test("a123456bc") );// false
console.log( reg6.test("123456abc") );// false
console.log( reg6.test("123abc456") );// false
console.log( reg6.test("^123abc456") );// false
console.log( reg6.test("^abc123456") );// false
console.log("");


var reg7 = /[abc]/;
console.log( reg7.test("a") );// true
console.log( reg7.test("b") );// true
console.log( reg7.test("c") );// true
console.log( reg7.test("789a456") );// true
console.log( reg7.test("d789") );// false
console.log( reg7.test("123") );// false
console.log("");

// 如果^在方括号内部 表示取反，只要包含方括号内的其中一个字符，都返回 false
var reg8 = /[^abc]/;
console.log( reg8.test("a") );// false
console.log( reg8.test("b") );// false
console.log( reg8.test("c") );// false
console.log( reg8.test("123") );// true
console.log( reg8.test("12a3") );// true
console.log( reg8.test("@c") );// true
console.log("");


var reg9 = /[^a-z]/; // 匹配字符串中,是否有"某个"字符符合这个正则表达式
console.log( reg9.test("a") );// false
console.log( reg9.test("A") );// true
console.log( reg9.test("123") );// true
console.log( reg9.test("123a456") );// true
console.log( reg9.test("1") );// true
console.log( reg9.test("b") );// false
console.log( reg9.test("1b") );// true
```

## 练习

1. 匹配字符串"abc"中是否含有a　　/a/
2. 匹配字符串"a1234abc56"中是否含有abc　/abc/
3. 匹配字符串"Zhagnsan456789"中是否含有小写字母a到z,大写字母A到z,0到9之间的数字的其中一个字符　/a-zA-Z0-9/

## 量词符

量词符用来设定某个模式出现的次数。

| 量词 | 说明 |
| --- | --- |
| * | 重复0次或更多次 |

| | |
|---|---|
| + | 重复1次或更多次 |
| | |
| ? | 重复0次或1次 |
| {n} | 重复n次 |
| {n,} | 重复n次或更多次 |
| {n,m} | 重复n到m次 |

**举例:**

```
<script>
        // 量词符: 用来设定某个模式出现的次数


        // 简单理解: 就是让下面的a这个字符串重复多少次
        var reg = /^a$/; // 精确匹配,表示字符串中只能是a
        console.log( reg.test("a") );// true
        console.log( reg.test("aa") );// false
        console.log( reg.test("aaaa") );// false
        console.log( reg.test("b") );// false
        console.log( reg.test("") );// false
        console.log("");


        // * 相当于 >= 0 可以出现0次或者很多次
        var reg1 = /^a*$/;  // 精确匹配,加上了次数限制
        console.log( reg1.test("") );// true
        console.log( reg1.test("a") );// true
        console.log( reg1.test("aa") );// true
        console.log( reg1.test("aaaa") );// true
        console.log("");


        // + 相当于 >= 1 可以出现1次或者很多次
        var reg2 = /^a+$/;
        console.log( reg2.test("") );// false
        console.log( reg2.test("a") );// true
        console.log( reg2.test("aa") );// true
        console.log( reg2.test("aaaa") );// true
        console.log("");


        // ? 相当于 1 || 0
        var reg3 = /^a?$/;
        console.log( reg3.test("") );// true
        console.log( reg3.test("a") );// true
        console.log( reg3.test("aa") );// false
        console.log( reg3.test("aaaa") );// false
        console.log("");


        var reg4 = /a+/; // 表示可以出现在字符串任何地方 +代表 大于等于1的次数
```

```
console.log( reg4.test("") );// false
console.log( reg4.test("abc") );// true
console.log( reg4.test("abac") );// true
console.log( reg4.test("ab123") );// true
console.log( reg4.test("456ab789abc") );// true
console.log("");


var reg5 = /^a+$/;// 精确匹配
console.log( reg5.test("") );// false
console.log( reg5.test("abc") );// false
console.log( reg5.test("abac") );// false
console.log( reg5.test("ab123") );// false
console.log( reg5.test("456ab789abc") );// false
console.log( reg5.test("a") );// true
console.log( reg5.test("aa") );// true
console.log( reg5.test("aaa") );// true
console.log("");


// {n} 重复n次
// {3} 就是重复3次
var reg6 = /a{3}/;
console.log( reg6.test("456aaa123"));// true
console.log( reg6.test("456a12aa3"));// false
console.log( reg6.test("aaa789"));// true
console.log( reg6.test("789aaa"));// true
console.log( reg6.test("aa123"));// false
console.log( reg6.test("123"));// false
console.log( reg6.test("789aaaa"));// true
console.log("");


// {n,} 重复n次或更多次
// {3, } 大于等于3
var reg7 = /^a{3,}$/;
console.log( reg7.test("1234b56") );// false
console.log( reg7.test("a123456") );// false
console.log( reg7.test("aa123456") );// false
console.log( reg7.test("aaa123456") );// false
console.log( reg7.test("aaa") );// true
console.log( reg7.test("aaaa") );// true
console.log( reg7.test("a") );// false
console.log( reg7.test("1aa23aaaa789") );// false
console.log("");


// {n,m} 重复n到m次
// {3,6} 大于等于3 并且 小于等于6
var reg8 = /[0-9]{3,6}/;
console.log( reg8.test("123abc") );// true
console.log( reg8.test("12abc") );// false
console.log( reg8.test("12456789abc") );// true
console.log( reg8.test("abc") );// false
```

```
        console.log("");



        var reg9 = /^[0-9]{3,6}$/;
        console.log( reg9.test("12456789abc") );// false
        console.log( reg9.test("12456789") );// false
        console.log( reg9.test("124567abc") );// false
        console.log( reg9.test("124567") );// true
        console.log( reg9.test("12") );// false
        console.log( reg9.test("123") );// true
        console.log( reg9.test("12345") );// true
    </script>
```

## 案例-用户名表单验证

### 功能需求:

1. 如果用户名输入合法，则后面提示信息为：用户名合法,并且颜色为绿色
2. 如果用户名输入不合法，则后面提示信息为：用户名不符合规范，并且颜色为红色

zhangsan 用户名格式输入正确

lisi 用户名格式输入不正确

### 分析:

1. 用户名只能为英文字母,数字,下划线或者短横线组成，并且用户名长度为6~16位
2. 首先准备好这种正则表达式模式/^[a-zA-Z0-9-_]{6,16}$/
3. 当表单失去焦点就开始验证
4. 如果符合正则规范，则让后面的span标签添加 right类
5. 如果不符合正则规范，则让后面的span标签添加 wrong类

### 代码:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        span {
            color: #aaa;
            font-size: 14px;
        }
```

```
        .right {
            color: green;
        }

        .wrong {
            color: red;
        }
    </style>
</head>
<body>
    <input type="text" class="uname">
    <span class="message">请输入用户名</span>
    <script>
        /*
        要求
        1. 如果用户名输入合法，则后面提示信息为：用户名合法,并且颜色为绿色
        2. 如果用户名输入不合法，则后面提示信息为：用户名不符合规范，并且颜色为红色


        分析：
        1. 用户名只能为英文字母,数字,下划线或者短横线组成，并且用户名长度为6~16位
        2. 首先准备好这种正则表达式模式/^[a-zA-Z0-9-_]{6,16}$/
        3. 当表单失去焦点就开始验证
        4. 如果符合正则规范，则让后面的span标签添加 right类
        5. 如果不符合正则规范，则让后面的span标签添加 wrong类 */


        // 获取相关对象
        var uname = document.querySelector(".uname");
        var message = document.querySelector(".message");
        var reg = /^[a-zA-z0-9_-]{6,16}$/;
        // 给文本框绑定失去焦点事件
        uname.onblur = function(){
            if( reg.test( this.value) ){
                // 正确
                message.innerHTML = "用户名合法";
                message.className = "right";
            }else{
                // 错误
                message.innerHTML = "用户名不符合规范";
                message.className = "wrong";
            }
        }
    </script>
</body>
</html>
```

## 括号总结

1. 中括号 字符集合 匹配方括号中的任意字符
2. 大括号 量词符 里面表示重复次数
3. 小括号表示优先级

我们还可以在线测试正则表达式

菜鸟工具 https://c.runoob.com/

regexper https://regexper.com/

正则测试工具 http://tool.oschina.net/regex

**举例:**

```
// 1.中括号[] 字符集合  匹配方括号中的任意字符
var reg1 = /^[abc]$/;
console.log( reg1.test("a") );// true
console.log( reg1.test("A") );// false
console.log( reg1.test("d") );// false
console.log( reg1.test("b") );// true
console.log( reg1.test("1a") );// false
console.log("");


// 2.大括号{}  量词符  里面表示重复次数
var reg2 = /^abc{2}$/; // 代表只重复c两次,也就是要精确匹配abcc
console.log( reg2.test("a") );// false
console.log( reg2.test("aa") );// false
console.log( reg2.test("aaa") );// false
console.log( reg2.test("b") );// false
console.log( reg2.test("c") );// false
console.log( reg2.test("cc") );// false
console.log( reg2.test("ccc") );// false
console.log( reg2.test("abc") );// false
console.log( reg2.test("abcc") );
console.log( reg2.test("abcabc") );
console.log("");



// 3.小括号() 表示优先级,也可以表示分组
var reg3 = /^(abc){2}$/;// 精确匹配 重复abc两次
console.log( reg3.test("abcc") );// false
console.log( reg3.test("abcabc") );// true
console.log( reg3.test("abcccc") );// false
console.log( reg3.test("abcabcabc") );// false
```

# 预定义类

预定义类指的是某些常见模式的简写方式

| 预定类 | 说明 |
| --- | --- |
| \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] |


| | |
| --- | --- |
| (?<=pattern) | 反向肯定预查，与正向肯定预查类似，只是方向相反。例如，"(?<=95\|98\|NT\|2000)Windows"能匹配"2000Windows"中的"Windows"，但不能匹配"3.1Windows"中的"Windows"。 |
| (?<!pattern) | 反向否定预查，与正向否定预查类似，只是方向相反。例如"(?<!95\|98\|NT\|2000)Windows"能匹配"3.1Windows"中的"Windows"，但不能匹配"2000Windows"中的"Windows"。 |
| x\|y | 匹配x或y。例如，"z\|food"能匹配"z"或"food"。"(z\|f)ood"则匹配"zood"或"food"。 |
| [xyz] | 字符集合。匹配所包含的任意一个字符。例如，"[abc]"可以匹配"plain"中的"a"。 |
| [^xyz] | 负值字符集合。匹配未包含的任意字符。例如，"[^abc]"可以匹配"plain"中的"p"。 |
| [a-z] | 字符范围。匹配指定范围内的任意字符。例如，"[a-z]"可以匹配"a"到"z"范围内的任意小写字母字符。 |
| [^a-z] | 负值字符范围。匹配任何不在指定范围内的任意字符。例如，"[^a-z]"可以匹配任何不在"a"到"z"范围内的任意字符。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置。例如，"er\b"可以匹配"never"中的"er"，但不能匹配"verb"中的"er"。 |
| \B | 匹配非单词边界。"er\B"能匹配"verb"中的"er"，但不能匹配"never"中的"er"。 |
| \cx | 匹配由x指明的控制字符。例如，\cM匹配一个Control-M或回车符。x的值必须为A-Z或a-z之一。否则，将c视为一个原义的"c"字符。 |
| \d | 匹配一个数字字符。等价于[0-9]。 |
| \D | 匹配一个非数字字符。等价于[^0-9]。 |
| \f | 匹配一个换页符。等价于\x0c和\cL。 |
| \n | 匹配一个换行符。等价于\x0a和\cJ。 |
| \r | 匹配一个回车符。等价于\x0d和\cM。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等等。等价于[ \f\n\r\t\v]。 |
| \S | 匹配任何非空白字符。等价于[^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于\x09和\cI。 |
| \v | 匹配一个垂直制表符。等价于\x0b和\cK。 |
| \w | 匹配包括下划线的任何单词字符。等价于"[A-Za-z0-9_]"。 |
| \W | 匹配任何非单词字符。等价于"[^A-Za-z0-9_]"。 |
| \x*n* | 匹配*n*，其中*n*为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，"\x41"匹配"A"。"\x041"则等价于"\x04&1"。正则表达式中可以使用ASCII编码。. |
| *num* | 匹配*num*，其中*num*是一个正整数。对所获取的匹配的引用。例如，"(.)\1"匹配两个连续的相同字符。 |
| *n* | 标识一个八进制转义值或一个向后引用。如果*n*之前至少*n*个获取的子表达式，则*n*为向后引用。否则，如果*n*为八进制数字（0-7），则*n*为一个八进制转义值。 |
| *nm* | 标识一个八进制转义值或一个向后引用。如果*nm*之前至少有*nm*个获得子表达式，则*nm*为向后引用。如果*nm*之前至少有*n*个获取，则*n*为一个后跟文字*m*的向后引用。如果前面的条件都不满足，若*n*和*m*均为八进制数字（0-7），则*nm*将匹配八进制转义值*nm*。 |
| *nml* | 如果*n*为八进制数字（0-3），且*m*和*l*均为八进制数字（0-7），则匹配八进制转义值*nml*。 |
| \u*n* | 匹配*n*，其中*n*是一个用四个十六进制数字表示的Unicode字符。例如，\u00A9匹配版权符号（©）。 |

```
<script>
    // 预定义类
    // \d表示0~9之间是所有数字,等同于[0-9]
    var reg1 = /[0-9]{3}/;
    console.log( reg1.test("abc123") );// true
    console.log( reg1.test("a12bc") );// false
    console.log( reg1.test("a1234bc") );// true
    console.log("");


    var reg2 = /\d{3}/;
    console.log( reg1.test("abc123") );// true
    console.log( reg1.test("a12bc") );// false
    console.log( reg1.test("a1234bc") );// true
    console.log("");


    var reg3 = /^\w$/; // \w等同于 [a-zA-z0-9_]
    console.log( reg3.test("a") );// true
    console.log( reg3.test("B") );// true
    console.log( reg3.test("_") );// true
    console.log( reg3.test("-") );// false
    console.log( reg3.test("@") );// false
    console.log( reg3.test("9") );// true
    console.log( reg3.test("10") );// false
    console.log("");


    var reg4 = /^\w+\d{3}$/;  // [a-zA-z0-9_]出现大于等于1次   [0-9]出现3次
    console.log( reg4.test("abc123") );// true
    console.log( reg4.test("1234") );// true
    console.log( reg4.test("-234") );// false
    console.log( reg4.test("_234") );// true
    console.log( reg4.test("_234@") );// false
</script>
```

# 案例-验证座机号码

```
1  正则里面的或者符号是  |
2  座机号码验证：全国座机号码 两个格式：010-12345678 或者0753-1234567
```

**举例:**

```
<script>
    // 正则里面的或者符号是 |
    // 座机号码验证：全国座机号码 两个格式：010-12345678 或者 0753-1234567
    var reg1 = /(^\d{3}-\d{8}$)|(^\d{4}-\d{7}$)/;
    console.log( reg1.test( "010-12345678" ) );
    console.log( reg1.test( "0753-1234567" ) );
    console.log( reg1.test( "020-123456789123" ) );
    console.log( reg1.test( "123456789123" ) );
    console.log( reg1.test( "0-123456789123" ) );
    console.log( reg1.test( "123-1234567" ));
    console.log( reg1.test( "1234-12345678" ));
    console.log("");


    // 如果用以下方式,会出现细微的漏洞
    var reg2 = /^\d{3,4}-\d{7,8}$/;
    console.log( reg2.test( "010-12345678" ) );
    console.log( reg2.test( "0753-1234567" ) );
    console.log( reg2.test( "020-123456789123" ) );
    console.log( reg2.test( "123456789123" ) );
    console.log( reg2.test( "0-123456789123" ) );
    console.log( reg2.test( "123-1234567" ));
    console.log( reg2.test( "1234-12345678" ));
</script>
```

# 匹配模式

匹配模式也叫修饰符：表示正则匹配的附加规则，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。

在正则表达式中，匹配模式常用的有两种形式：

g ： global缩写，代表全局匹配，匹配出所有满足条件的结果，不加g第一次匹配成功后，正则对象就停止向下匹配;

i ： ignore缩写，代表忽略大小写，匹配时，会自动忽略字符串的大小写

gi：全局匹配 + 忽略大小写

**语法：**

```
var reg = /正则表达式/匹配模式;
或者
var reg = new RegExp(/正则表达式/, 匹配模式);
```

# 正则替换replace

replace() 方法可以实现替换字符串操作，用来替换的参数可以是一个字符串或是一个正则表达式。

**语法:**

```
stringObject.replace(regexp/substr,replacement)


1. 第一个参数：被替换的字符串 或者 正则表达式
2. 第二个参数：替换为的字符串
3. 返回值是一个替换完毕的新字符串
```

**举例:**

```
<script>
    // 在字符串对象中有一个  replace的方法可以替换指定字符串
    // 正则替换replace
    // replace() 方法可以实现替换字符串操作，用来替换的参数可以是一个字符串或是一个正则表达式。
    // 语法:
    // stringObject.replace(regexp/substr,replacement)
    // 1. 第一个参数：被替换的字符串 或者 正则表达式
    // 2. 第二个参数：替换为的字符串
    // 3. 返回值是一个替换完毕的新字符串


    var str1 = "Abc123abcAbc";


    // 替换的参数可以是一个字符串
    var str2 = str1.replace("A","啊");
    console.log( str2 );
    console.log("");


    // 替换的参数也可以是一个正则表达式
    var reg1 = /A/;
    var str3 = str1.replace( reg1 , "啊" );
    console.log( str3 );
    console.log("");



    // 匹配模式g  g代表global全局的意思 会全文匹配满足条件所有内容
    var reg2 = /A/g;
    var str4 = str1.replace( reg2 , "啊" );
    console.log( str4 );
    console.log("");


    // 匹配模式i i代表ignore忽略的意思,指的是忽略大小字母
    var reg3 = /a/i;
    var str5 = str1.replace( reg3 , "啊" );
    console.log( str5 );
    console.log("");



    // 组合的匹配模式 gi 或 ig 一样的效果  忽略大小写并且全局匹配
    var reg4 = /a/gi;
    // var reg4 = new RegExp(/a/gi,"gi");
    // var reg4 = /a/ig;
    var str6 = str1.replace( reg4 , "啊" );
    console.log( str6 );
    console.log("");
</script>
```

啊bc123abcAbc

啊bc123abcAbc

啊bc123abc啊bc

啊bc123abcAbc

啊bc123啊bc啊bc

## 案例-过滤敏感词汇

去你妹的徐大大

提交

去*的徐*

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>

    <textarea id="message" rows="10" cols="50"></textarea>
    <button>提交</button>
    <div></div>


    <script>
        // 获取相关对象
        var message = document.querySelector("#message");
        var button  = document.querySelector("button");
        var div     = document.querySelector("div");


        // 定义敏感词汇的正则表达式
        var reg = /你妹|你大爷|大大/g;


        // 给按钮绑定事件
        button.onclick = function(){
            // 获取文本域中的内容
            // console.log( message.value );
            var newStr = message.value.replace(reg,"*");
            div.innerHTML = newStr;
        }
    </script>
</body>
</html>
```

# 正则对象相关方法

**test(str)**：判断字符串中是否具有指定模式的子串，返回结果是一个布尔类型的值;

**exec(str)**：返回字符串中指定模式的子串，一次只能获取一个与之匹配的结果;

**格式:**

```
返回值 = 正则.exec(目标字符串)
```

**功能:** 匹配。在目标字符串中，找出符合正则表达式要求的子串。

```
返回值:
```

```
  如果能够匹配某个子串，则返回值是一个数组。其中：
  第一个元素是匹配成功的子串；
  第二个元素index，表示在哪里匹配到的。
  第三个元素input，表示目标字符串

如果不能匹配，则返回值是null
```

```
子串的概念
如果有一个目标字符串是"abcd"，它的子串有：
（1）长度为1的子串：a,b ,c d,
（2）长度为2的子串：ab ,bc,cd （ “bd”不是子串，因为它们不相邻）
（3）长度为3的子串：abc,bcd
（4）长度为4的子串：abcd
```

一次exec只能得到一个匹配成功的结果， 如果要全部匹配出来，则需要调用多次exec.

注意一下:g不可少!

```
<script type="text/javascript">
    var str = "生日是:19901-10-10 生日是:1998-11-12 生日是2003-05-03 生日是3003-06-07
        生日是2008-08-08";
    var reg = /[12]\d{3}-\d{2}-\d{2}/;
    // var reg = /[12]\d{3}-\d{2}-\d{2}/g;
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );

    /*while(  result = reg.exec( str )  ){
       //输出匹配结果
       console.log( result )
    }*/
</script>
```

```
<script type="text/javascript">
    var str = "生日是:19901-10-10 生日是:1998-11-12 生日是2003-05-03 生日是3003-06-07
        生日是2008-08-08";
    // var reg = /[12]\d{3}-\d{2}-\d{2}/;
    var reg = /[12]\d{3}-\d{2}-\d{2}/g;
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );

    /*while(  result = reg.exec( str )  ){
       //输出匹配结果
       console.log( result )
    }*/
</script>
```

有一个问题，我们事先并不知道，这个匹配会成功多少次？所以我不能用for循环去执行匹配。

### 可以使用while循环匹配

格式:

```
While( 匹配结果= 正则.exec(“目标字符串”) ){
  //输出匹配结果
}
```

```
<script type="text/javascript">
    var str = "生日是:19901-10-10 生日是:1998-11-12 生日是2003-05-03 生日是3003-06-07
        生日是2008-08-08";
    // var reg = /[12]\d{3}-\d{2}-\d{2}/;
    var reg = /[12]\d{3}-\d{2}-\d{2}/g;
    /*console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );
    console.log( reg.exec( str ) );*/

    while(  result = reg.exec( str )  ){
       //输出匹配结果
       console.log( result )
    }
</script>
```

```
result = reg.exec(str) 做一次匹配，把结果保存在result中。
如果匹配成功，则result是一个数组;
如果匹配失败，则result是一个null;
上面是一个赋值表达式，表达式本身的值也就是rs的值，会在while进行判断
while( 表达式 ){} 的流程是:先检查表达式的逻辑true或者是false。
```

如果是true就执行循环体，如果是假，则结束循环。

当reg.exec(str)的结果是null，则while条件不成立，循环结束掉。

**举例:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 从字符串str里面找到所有生日  生日 xxxx-xx-xx
        var str = "生日是:19901-10-10 生日是:1998-11-12 生日是2003-05-03 生日是3003-06-07 生日是2008-08-08";

        // var reg = /[12]\d{3}-\d{2}-\d{2}/;
        // regObj.test(str) : 判断字符串中是否具有指定模式的子串，返回结果是一个布尔类型的值;
        // console.log( reg.test(str) );// true

        // regObj.exec(str) : 返回字符串中指定模式的子串，一次只能获取一个与之匹配的结果;
        // 格式:
        // 返回值 = 正则.exec(目标字符串)
        /* var result =  reg.exec( str ) ;
        console.log( result );


        var result =  reg.exec( str ) ;
        console.log( result ); */
        // 功能:  匹配。在目标字符串中，找出符合正则表达式要求的子串。
        // 返回值:
        // 如果能够匹配某个子串，则返回值是一个"数组"。其中:
        // 第一个元素是匹配成功的子串;
        // 第二个元素index，表示在哪里匹配到的。
        // 第三个元素input，表示目标字符串


        // 如果不能匹配，则返回值是 null


        // 子串的概念
        // 如果有一个目标字符串是"abcd"，它的子串有:
        // （1）长度为1的子串:a,b ,c d,
        // （2）长度为2的子串:ab ,bc,cd ( “bd”不是子串，因为它们不相邻)
        // （3）长度为3的子串:abc,bcd
        // （4）长度为4的子串:abcd


        // 一次exec只能得到一个匹配成功的结果 ，如果要全部匹配出来，则需要调用多次exec
        // 注意一下:g不可少!
        /* var reg = /[12]\d{3}-\d{2}-\d{2}/g;
        var result =  reg.exec( str ) ;
        console.log( result );


        var result =  reg.exec( str ) ;
        console.log( result );


        var result =  reg.exec( str ) ;
        console.log( result );


        var result =  reg.exec( str ) ;
        console.log( result );
```

```
        var result =  reg.exec( str ) ;
        console.log( result );


        var result =  reg.exec( str ) ;
        console.log( result );


        var result =  reg.exec( str ) ;
        console.log( result );


        var result =  reg.exec( str ) ;
        console.log( result );


        var result =  reg.exec( str ) ;
        console.log( result ); */


        // 有一个问题，我们事先并不知道，这个匹配会成功多少次？所以我不能用for循环去执行匹配。
        // 要使用while循环配合exec方法把所有满足条件的内容输出
        var reg = /[12]\d{3}-\d{2}-\d{2}/g;
        while( result =  reg.exec( str ) ){
            console.log( result );
        }
    </script>
</body>
</html>
```

# String对象跟正则相关方法

search(reg)： 与indexOf非常类似，返回指定模式的子串在字符串首次出现的位置

match(reg)： 以数组的形式返回指定模式的字符串，可以返回所有匹配的结果

replace(reg,'替换后的字符')： 把指定模式的子串进行替换操作

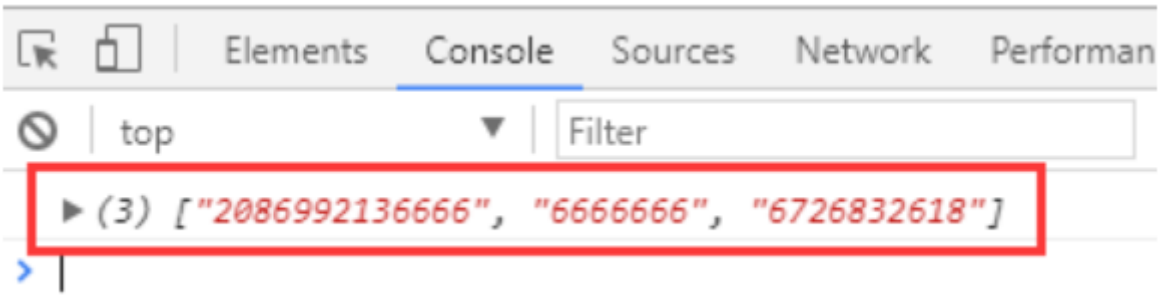
split(reg)： 以指定模式分割字符串，返回结果为数组

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 字符串对象跟正则相关方法
        // replace(reg,'替换后的字符') ：把指定模式的子串进行替换操作
        // search(reg) ：与indexOf非常类似，返回指定模式的子串在字符串首次出现的位置
        var str = "zxcv12abc456qwe789";
        var reg = /\d{3}/;
        console.log( str.search( reg ) );// 9
        console.log("");


        // match(reg) ：以"数组"的形式返回指定模式的字符串，可以返回所有匹配的结果 推荐 如果想要所有的结果,记得使用全局模式噢
        var birthday = "生日是:19901-10-10 生日是:1998-11-12 生日是2003-05-03 生日是3003-06-07 生日是2008-08-08";
        // 全局模式匹配
        var reg2= /[12]\d{3}-\d{2}-\d{2}/g;
        var arr = birthday.match( reg2 );
        console.log( arr );
        console.log("");


        // 不全局模式匹配
        var reg3= /[12]\d{3}-\d{2}-\d{2}/;
        // match方法如果不是全局匹配,匹配出来的效果跟正则对象的exec方法一样
        var arr2 = birthday.match( reg3 );
        console.log( arr2 );


        var arr3 = birthday.match( reg3 );
        console.log( arr3 );


        console.log( reg3.exec( birthday ) );
        console.log( reg3.exec( birthday ) );
        console.log("");

        // split(reg) ：以指定模式分割字符串，返回结果为数组
        var str = "zxcv12abc456qwe789";
        var result = str.split(/\d{3}/);
        console.log( result );// ["zxcv12abc", "qwe", ""]
    </script>
</body>
</html>
```

# 贪婪模式

对QQ号码进行校验要求5~13位,不能以0开头,只能是数字

```
1 var str = '我的QQ20869921366666666666，nsd你的是6726832618吗？';
2 var reg = /[1-9]\d{4,12}/g;
3 console.log(str.match(reg));
```



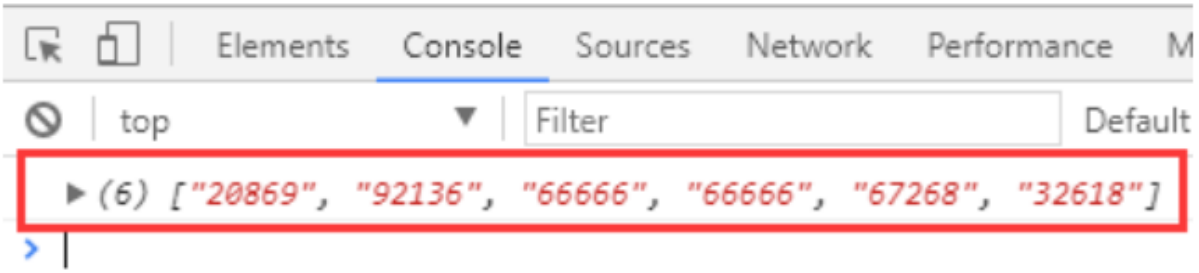
我们会发现以上代码运行结果中，默认优先配到 13 位，在对后面的进行匹配；为什么不是优先匹配 5 位后，在对后面的进行匹配呢?

因为在正则表达式中，默认情况下，能匹配多的就不匹配少的，我们把这种匹配模式就称之为 **贪婪匹配**,也叫做 **贪婪模式**所有的正则表达式，默认情况下采用的都是贪婪匹配原则。

如果在限定符的后面添加一个问号？，那我们的贪婪匹配原则就会转化为**非贪婪**匹配原则，优先匹配少的，也叫**惰性匹配**；

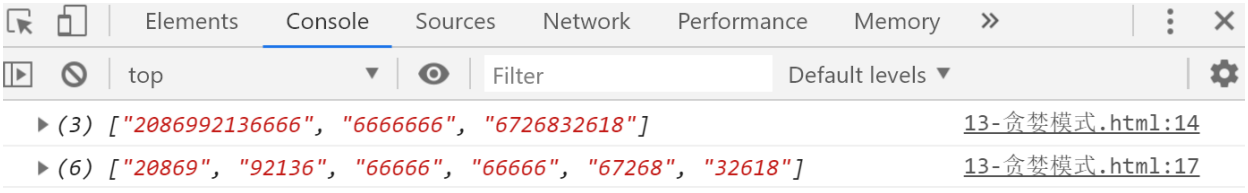
```
var str = '我的QQ2086992136666666666，nsd你的是6726832618吗？';
//非贪婪模式匹配，
var reg = /[1-9]\d{4,12}?/g;
console.log(str.match(reg));
```

```
<script>
    var str = '我的QQ2086992136666666666，nsd你的是6726832618吗？';
    var reg = /[1-9]\d{4,12}?/g;
    console.log(str.match(reg));
</script>
```




**举例:**

```
<script>
    // 因为在正则表达式中，默认情况下，能匹配多的就不匹配少的，我们把这种匹配模式就称之为 贪婪匹配,也叫做 贪婪模式所有的正则表达式，默认情况下采用的都是贪婪匹配原则。
    var str = '我的QQ2086992136666666666，nsd你的是6726832618吗？';
    var reg = /[1-9]\d{4,12}/g; // 匹配QQ号的正则表达式
    // 全局模式下使用match可以得到所有符合正则表达式的内容
    console.log( str.match( reg ) );
    // 如果在限定符(量词符)的后面添加一个问号？，那我们的贪婪匹配原则就会转化为非贪婪匹配原则，优先匹配少的，也叫惰性匹配；
    var reg2 =  /[1-9]\d{4,12}?/g;
    console.log( str.match( reg2 ) );
</script>
```




# 今日总结

xmind要做

# 今日作业

请看作业文件夹