23-第二十三章 RegExp

一、 什么是正则表达式

> 正则表达式是描述字符模式的对象。
>
> - 正则表达式用于对字符串模式匹配及检索替换，是对 `字符串` 执行模式匹配的强大工具。

- 而 `String` 和 `RegExp` 都定义了使用正则表达式进行强大的模式匹配和文本检索与替换的函数。
- 正则表达式主要用来验证客户端的输入数据。可以节约大量的服务器端的系统资源，并且提供更好的用户体验。

二、创建正则表达式

1、直接量

语法：Reg = /pattern/modifiers;

```
1. var Reg = /box/gi;
```

2、new RegExp

语法 Reg = `new RegExp(` pattern , modifiers `)`;
pattern ，modifiers此时是`字符串`

```
1. var Reg = new RegExp("box","gi");
```

- 何种方法创建都是一样的
- pattern `模式 模板` ，要匹配的内容
- modifiers `修饰符`

三、正则表达式用法及区别

1、String中正则表达式方法

方法	描述
`match( Reg )`	返回RegExp匹配的包含全部`字符串`的数组或`null`
`search( Reg )`	返回RegExp匹配字符串首次出现的`位置`
`replace( Reg , newStr )`	用`newStr`替换RegExp匹配结果，并返回新字符串
`split( Reg )`	返回字符串按指定RegExp拆分的数组

使用

```
var str = 'hello';
var Reg = /e/i;
str.match(Reg);
```

2、RegExp对象的方法

方 法

方法	描述
`exec（）`	在字符串中执行匹配搜索，返回首次匹配结果`数组`，
`test（）`	在字符串中测试模式匹配，返回`true`或`false`

使用

```
    var pattern = new RegExp(“box”,”gi”);
    pattern.test(str);
    pattern.exec(str);

```

> 注意区别正则方法和字符串方法使用避免混淆

正则方法：pattern.`test(`str`)`; 方法的主体是`正则表达式`
字符串方法：str.`match(`pattern`)`;方法的主体是`字符串`

四、修饰符

> 修饰符用于`执行区分大小写`和`全局`匹配:

- `i` `忽略大小`写匹配
- `g` `全局匹配`，默认只匹配第一个元素，就不在进行匹配
- `m` 执行`多行匹配`

```
1.      var patt =  /pattern/i;          //忽略大小写匹配
2.      var patt =  /pattern/g;          //全局匹配
3.      var patt =  /pattern/m;          //执行多行匹配
```

四、pattern 模式

1、基本匹配

xxx ——————————————匹配`xxx`字符

```
1.  var Reg = /abc/;
```

`x|y|z` ——————————————匹配`x`或`y`或`z`字符

```
1.  var Reg = /abc|bac|cba/;
```

2、[]

`[abc]` ——————————匹配abc之中的`任何一个`字符
非
`[^abc]` ——————————匹配`非`a`非`b`非`c字符的
到

`[0-9]` ——————————匹配`0至9`之间的数字
`[a-z]` ——————————匹配`小写a至小写z`的字符
`[A-Z]` ——————————匹配`大写A至大写Z`的字符
匹配中文 `[\u4e00-\u9fa5]`

还可以组合

```
1.  var Reg  = /hello [0-9a-zA-z]/;
```

3、元字符(`转义字符`)

`.` ——————————匹配`单个字符`，除了换行和行结束符
`\w` ——————————匹配`单词字符`,`数字`,`_`(下划线)
`\W` ——————————匹配`非`（`单词字符`和`_`(下划线)）
`\d` ——————————匹配`数字`
`\D` ——————————匹配`非数字`
`\s` ——————————匹配空白字符（`空格`）
`\S` ——————————匹配`非空格`字符
`\b` ——————————匹配`单词边界`（除了（字)字母 数字_ 都算单词边界)
`\B` ——————————匹配`非单词边界`
`\n` ——————————匹配`换行符`

特殊的转译字符`.` `\` `/`

```
1. var reg = /\./;//匹配.
2. var reg = /\\/;//匹配\
3. var reg = /\//;//匹配/
```

4、量词

`n?` ——————————匹配 `0个或一个` n的字符串
`n*` ——————————匹配 `0个或多个` 字符串(`任意个`)
`n+` ——————————匹配 `至少一个` n字符串
`n{X}` ——————————匹配包含 `X个n` 的序列的字符串
`n{X,Y}` ——————————匹配包含 `至少X或至多Y个` n的序列的字符串
`n{x,}` ——————————匹配 `至少X个` n的序列字符串

`^n` ——————————匹配 `以n开头` 的字符串
`n$` ——————————匹配 `以n结尾` 的字符串

5、贪婪惰性

> **贪婪**: `尽可能多` 的匹配
> **惰性**: `尽可能少` 的匹配
> 前提条件都是要匹配到内容

—— 贪婪 ——	—— 惰性 ——
`+`	+ `?`
`?`	? `?`
`*`	* `?`
`{n}`	{n} `?`
`{n,m}`	{n,m} `?`
`{n,}`	{n,} `?`

6、子组(子表达式)

子组:使用 `()` 小括号,指定一个子表达式后,称之为分组

- `捕获型`
- `非捕获型`

1）、捕获型

```
var str = 'abcdefg';
var reg = /(abc)d/;//匹配abcd
var val = reg.exec( str);
console.log( val );   //["abcd", "abc", index: 0, input: "abcdefg"]
```

索引0 为匹配的结果
索引1 为第一个子表达式 匹配结果
index :首次匹配成功的索引值，
input: **匹配目标**

—— 字符 ——		引用
`(pattern)`	匹配pattern并`捕获结果`，`自动设置组号`,是从1开始的正整数	`\num`

引用是`值的引用，匹配结果的引用`不是匹配形式引用

1）、非捕获型

- `(? :pattern)`
- `(?=pattern)` 零宽度正向预言

```
Windows (?=2000) //匹配windows且后面跟2000
```

匹配 “Windows2000” 中的 “Windows”
不匹配 “Windows3.1” 中的 “Windows”。

- `(?!pattern)` 零宽度负向预言

```
 Windows (?!2000)//匹配windows且后面非2000
```

匹配 “Windows3.1” 中的 “Windows”
不匹配 “Windows2000” 中的 “Windows”。