

DOM映射机制

在js中获取的DOM元素或者元素集合，和HTML页面上的元素标签是存在“映射关系”的

- js中把DOM元素进行修改，页面中的元素样式也会跟着改变
- 页面中元素的结构发生改变，js中DOM元素也会跟着改变

```
var stuList = document.getElementById("stuList"),
    stuBody = stuList.tBodies[0],
    stuRows = stuBody.rows;
//->开始的时候也没中没有tr，所以stuRows是一个空的类数组
//->ajax获取数据然后做数据绑定
~function () {
    //get data
    var stuData = null;
    var xhr = new XMLHttpRequest;
    xhr.open("GET", "json/data.json", false);
    xhr.onreadystatechange = function () {
        if (xhr.readyState === 4 && xhr.status === 200) {
            stuData = utils.toJSON(xhr.responseText);
        }
    };
    xhr.send(null);

    //bind data
    var str = ``;
    for (var i = 0; i < stuData.length; i++) {
        var curItem = stuData[i];
        str += `<tr>
            <td>${curItem.id}</td>
            <td>${curItem.name}</td>
            <td>${curItem.age}</td>
        </tr>`;
    }
    stuBody.innerHTML = str; //->向页面中增加了20个tr
    console.log(stuRows); //->由于页面中tbody结构中的内容改变了，根据DOM映射机制此处不需要重新的获取，stuRows中存储的就是最新的20条数据
}();
```

所谓的正则：

用来处理 字符串 的规则

- 把验证当前的字符串是否符合规则 — 匹配
- 把字符串中符合规则的字符捕获到 — 捕获

正则匹配：`[正则].test([字符串])`

正则捕获：`[正则].exec([字符串])` 或者 `[字符串].match([正则])` 或者 `[字符串].replace([正则],function...)` 或者 `[字符串].split([正则]) ...`

元字符和修饰符

一个正则就是由元字符和修饰符组成的，想要学会编写自己所需的规则，需要牢牢掌握元字符和修饰符

修饰符 img

- `i(ignoreCase)`：忽略单词大小写匹配
- `m(multiline)`：多行匹配
- `g(global)`：全局匹配

```
// -> 修饰符放在最后一个斜杠的后面（字面量创建方式）
```

```
var reg = /\d+$/img;
```

```
// -> 修饰符放在第二个实参字符串中（实例创建方式）
```

```
var reg = new RegExp("", "img");
```

特殊元字符

- `\`：转义字符，把普通元字符转换为特殊的意义，或者把特殊元字符转换为普通的意义，例如：`/\d/` `d`本身是一个字母，前面加一个转义字符，代表0~9之间的一个数字 或者 `/\./` 点在正则中代表任意字符（特殊含义），此处加上转义字符，代表的就是本身意思点了
- `^`：以某一个元字符开始，例如：`/^1/` 代表当前的字符串应该是以1开始的
- `$`：以某一个元字符结束，例如：`/2$/` 代表当前字符串最后一个字符应该是以2结尾
- `\d`：代表一个0~9之间的数字
- `\D`：和`\d`正好相反，代表一个非0~9之间的任意字符（所有大写字母都和小写字母的是相反的）
- `\w`：数字、字母、下划线三者中的任意一个
- `\n`：匹配一个换行符
- `\b`：匹配一个边界
- `\s`：匹配一个空白字符
- `.`：除了`\n`以外的任意一个字符

- `x|y` : `x`或者`y`中的一个字符
- `()` : 分组
- `[a-z]` : 匹配一个`a-z`中的任意字符 / `[0-9]` / `0~9`之间的任何一个数字, 等价于`\d`
- `[^a-z]` : 除了`a-z`以外的任意一个字符, 这里`^`是取反的意思
- `[xyz]` : `x`或者`y`或者`z`, 三者中的一个
- `[^xyz]` : 除了三者以外的任意一个字符
- `?` : 正向预查
- `?!:` 负向预查
- `?:` : 只匹配不捕获

量词元字符

- `*` : 前面的元字符出现0次到多次
- `+` : 前面的元字符出现1次到多次
- `?` : 前面的元字符出现0次或者1次
- `{n}` : 出现`n`次
- `{n,}` : 出现`n`到多次
- `{n,m}` : 出现`n`到`m`次

```
// -> 两个斜杠中间包起来的都是正则的元字符
// 1、特殊元字符: 有特殊含义的
// 2、量词元字符: 代表出现多少次的元字符
// 3、普通元字符: 代表本身含义

var reg = /^ \d+ $ /

var reg = new RegExp("[元字符]", "[修饰符]");
```

常用的正则表达式

中括号的一些细节问题

```
// -> 1、中括号里面出现的多位数字, 不是多位数, 而是数字中出现的任意一个
var res = /^[18] $ /; // -> 1或者8中的一个数字
var reg = /^[18-65] $ /; // -> 1或者6-8或者5, 三者中的一个数字

// -> 2、\w使用中括号的方式表达: 数字、字母、下划线
var reg = /^[0-9a-zA-Z_] $ /;

// -> 3、中括号中出现的元字符一般都是自己本身的意思 (即使具备特殊的意思, 很多元字符也都自动变为本身的意思了)
var reg = /^[+-. \d?] $ /; // -> 除了 \d 依然代表的是 0~9 中的一个数字, 其余的都是代表本身的意思
```

```
var reg=/^[18-65]$/; //->1或者6-8或者5，三者中的一个数字
```

```
> var reg=/^[18-65]$/;
```

```
✖ ▶ Uncaught SyntaxError: Invalid regular expression: /^[18-65]$/: Range out of order in character class  
at <anonymous>:1:9
```

```
> var reg=/^[16-85]$/;
```

```
< undefined
```

```
> reg.test("1")
```

```
✖ Uncaught SyntaxError: Invalid or unexpected token
```

```
> reg.test("1")
```

```
< true
```

```
> reg.test("5")
```

```
< true
```

```
> reg.test("6")
```

```
< true
```

```
> reg.test("8")
```

```
< true
```

```
> reg.test("16")
```

```
< false
```

```
> reg.test("85")
```

```
< false
```

```
var reg=/^[0-9a-zA-Z_]$/;
```

```
> var reg=/^[0-9a-zA-Z_]$/;
```

```
< undefined
```

```
> reg.test("d");
```

```
< true
```

```
> reg.test("哈");
```

```
< false
```

```
> reg.test("@");
```

```
< false
```

```
> reg.test("1");
```

```
< true
```

```
var reg=/^[+-.\d?]$/; //->除了\d依然代表的是0~9中的一个数字，其余的都是代表本身的意思
```

```

> var reg=/^[+-.\\d?]*$/;
< undefined
> reg.test("呵呵")
< false
> reg.test("@")
< false
> reg.test("+")
< true
> reg.test("-")
< true
> reg.test("? ")
< false
> reg.test("?")
< true
> reg.test("")
✖ Uncaught SyntaxError: Invalid or unexpected token
> reg.test("\\")
< false
> reg.test("d")
< false
> reg.test(".")
< true
> reg.test("\\d")
< false

```

小括号的一些作用和细节

```

/*var reg=/^18|19$/; //->按照我们本身的理解，应该是18或者19两个中的任意一个，符合x|y这个元字符的规则
//->但是现实不是这样的，上面的规则，18/19/181/189/119/819...都符合，它识别和处理的规则特别乱

var reg=/^(18|19)$/; //->当我们使用分组把它包起来的时候就好了，现在只能匹配18或者19了，其余的都不可以*/

//=>正则中分组`() `的第一个作用：改变默认的优先级

//=>                第二个作用：分组引用
//->\1 或者 \2 或者 \数字 代表和对应分组出现一模一样的内容，也就是\1代表和第一个分组出现的内容一模一样
var reg = /^[a-z]([a-z])\2\1$/;
//->oppo moom noon toot ...

=>                第三个作用：分组捕获

```

```

var reg=/^18|19$/; //->上面的规则，18/19/181/189/119/819...都符合，它识别和处理的规则特别乱

```

```
> var reg=/^18|19$/;
< undefined
> reg.test("18");
< true
> reg.test("19");
< true
> reg.test("189");
< true
> reg.test("119");
< true
```

以1开头以9结尾的任何拼接运算

var reg=/^(18|19)\$/; //->当我们使用分组把它包起来的时候就好了，现在只能匹配18或者19了，其余的都不可以

```
> var reg=/^(18|19)$/;
< undefined
> reg.test("19");
< true
> reg.test("189");
< false
> reg.test("18");
< true
```

用分组包起来就可以实现

验证年龄：18~65

```
> var reg=/^((18|19)|([2-5]\d)|([60-5]))$/;
< undefined
> reg.test("18-65");
< false
> reg.test("18");
< true
> reg.test("65");
< true
> reg.test("66");
< false
> reg.test("66");
< false
> reg.test("17");
< false
```

验证中文姓名的

```
> reg=/^[^\u4E00-\u9FA5]{2,5}(\.[^\u4E00-\u9FA5]{2,5})?$/;
< /^[^\u4E00-\u9FA5]{2,5}(\.[^\u4E00-\u9FA5]{2,5})?$/
> reg.test("Fancy·樱桃小丸子");
< false
> reg.test("小丸子·樱桃小丸子");
< true
```

验证身份证号码的[完善]

```
> var reg = /^(\d{6})(\d{4})(\d{2})(\d{2})\d{2}(\d)(\d|X)$/;
< undefined
> reg.exec("152801198902153067");
< ▶ (7) ["152801198902153067", "152801", "1989", "02", "15", "6", "7", index: 0, input: "152801198902153067"]
> var reg = /^(\d{6})(\d{4})(\d{2})(\d{2})\d{2}(\d)(?:\d|X)$/;
< undefined
> reg.exec("152801198902153067");
< ▶ (6) ["152801198902153067", "152801", "1989", "02", "15", "6", index: 0, input: "152801198902153067"]
```