20170806

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
}();
```

正则

> 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：代表本身含义的
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

```
/*
* 1、手机号码（简单）
*    ->以1开头
*    ->11位数字
*/
var reg=/^1\d{10}$/;//->同时加了^和$，此正则可以理解为只能是某某某规则；如果两个都不加，只要字符串中有一部分符号这个规则即可 （var reg=/1\d{10}/;）

/*
* 2、验证年龄：18~65
* ->18或者19
* ->20~59
* ->60~65
* 分三个阶段完成操作
*/
var reg = /^((18|19)|([2-5]\d)|(6[0-5]))$/;
/*
* 3、验证中文姓名的
* ->只能是中文汉字
*      第一个汉字的ASCII码：\u4E00
*      最后一个汉字的ASCII码：\u9FA5
* ->正常一般都是2~4位
*
*/
var reg = /^[\u4E00-\u9FA5]{2,4}$/;
    reg = /^[\u4E00-\u9FA5]{2,5}(·[\u4E00-\u9FA5]{2,5})?$/;//->支持：大卫·贝克汉姆 这种英文译名

/*
* 4、验证是否为有效数字的（正数/负数/小数/整数/0）
* -> 可能出现-，也可能不出现 `-?`
* -> 整数：一位数可以是0，多位数0不能作为开头 `(\d|([1-9]\d+))`
* -> 小数：小数部分可能有可能没有，如果一旦有，小数点后面必须有数字 `(\.\d+)?`
*/
var reg = /^-?(\d|([1-9]\d+))(\.\d+)?$/;


/*
* 5、验证身份证号码的
* ->简单：前17位是数字，最后一位数字或者X
* ->完善：我们需要在身份证中获取到 省市县、出生年月日、性别 等信息
*/
var reg = /^\d{17}(\d|X)$/;//->简单
var reg = /^(\d{6})(\d{4})(\d{2})(\d{2})\d{2}(\d)(?:\d|X)$/;
//->把需要单独获取的部分用分组包起来，正则捕获的时候可以捕获到分组中的内容，如果加了分组，但是不想捕获，在该分组前面加`?:`即可（只匹配不捕获）
```

验证年龄：18~65

```
> var reg=/^((18|19)|([2-5]\d)|(6[0-5]))$/;
< 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"]
```

正则的捕获

> exec：可以实现正则的捕获，每一次执行exec只能捕获到一个匹配的结果，而且结果是一个数组
>
> 第一项：当前正则捕获的内容（字符串）
> index：当前正则捕获的起始索引
> input：当前操作的原始字符串

```
var reg = /\d+/;//->包含1到多个数字
var str = 'zhufeng2017peixun2018';
reg.esec(str);//->["2017", index: 7, input: "zhufeng2017peixun201
8"]
reg.esec(str);//->["2017", index: 7, input: "zhufeng2017peixun201
8"]

/*
* 问题：当前正则执行一次exec只能捕获到一个匹配的内容，我们执行两次exec，第二次捕获到的依然还是第一次的结果，不管执行多少次exec，捕获到的依然都是第一个  =>“正则捕获的懒惰性”
*
* lastIndex：下一次正则捕获的时候，在字符串中查找的开始位置索引
* 原因：第一次查找之前，reg.lastIndex=0，也就是第一次是从字符串的开始位置查找的，所以找到的是2017
* 第一次执行exec结束后，reg.lastIndex值还是0，所以第二次依然是从字符串的开始位置找的，找到的当然还是2017
* ...
* 解决正则的懒惰性：
* 执行exec后，让lastIndex值变为当前这一次捕获的结束位置，这样下一次捕获的时候，就可以接着继续查找了（而不是从头开始了）
* =>我们只需要给正则加一个全局修饰符g，就可以在每一次执行exec后，自动修改它的lastIndex了
*/

var reg = /\d+/g;//->包含1到多个数字
var str = 'zhufeng2017peixun2018';

reg.lastIndex ->0
reg.esec(str);//->["2017"...]

reg.lastIndex ->11
reg.esec(str);//->["2018"...]

reg.lastIndex ->21
reg.esec(str);//->null 捕获不到

reg.lastIndex ->0
reg.esec(str);//->["2017"...]

/*
* 上面的案例中，我们知道执行两次就可以捕获全了，但是如果你不知道具体要捕获多少次，我们该如何是好？
* =>接下来我们自己在RegExp的原型上扩展一个方法：myExecAll，执行这个方法，可以把所有匹配的一次性都捕获到
*/

//->形参(str)：需要捕获的原始字符串
```

```
RegExp.prototype.myExecAll = function myExecAll(str){
    //->this:reg
    //->为了防止不加g的时候，每一次捕获的都是第一个，导致死循环，我们在正则没有加g的时候执行一次即可
    if(!this.global){
    //->没有加g：执行一次exec即可
    return this.exec(str);
    }
    //->已经加g了
    var ary = [];//->存储所有捕获的结果
    var res = this.exec(str);
    while(res){
        ary[ary.length] = res;//->把每一次捕获的结果存放在数组的末尾
        res = this.exec(str);
    }
    return ary;
}
//reg.myExecAll(str);

/*
* 生活如此美好，何必介么麻烦和纠结
* 字符串中有一个方法叫match，执行这个方法，也可以把所有匹配的内容，一次性捕获到，但是前提正则也需要加g，不加g，也只能捕获第一个
*/

var reg = /\d+/;
var str = 'zhufeng2017peixun2018hahah2019hehehhe2020';
str.match(reg); //=>["2017", "2018", "2019", "2020"]

/*
* 难道生活如此简单吗？
* match虽然很简单暴力，但是在需要捕获小分组内容的时候，就不太符合我们的需求了，因为match只能把大正则匹配捕获到，对于小分组匹配的无法捕获
*/
var ary = ["Junior",18];
var str = "my name is {0},I am {1} years old~~";
//->我们想把 {0} 替换成 ary[0]
//->我们想把 {1} 替换成 ary[1]
//->我们既要捕获到 {数字}，也需要把里面的数字单独的获取到（因为这个数字可以充当我们在数组中获取内容的索引）
var reg = /\{(\d+)\}/g;//->大正则匹配的是“一个大括号中包含数字”，(\d+)第一个小分组匹配的是“大括号中的那个数字”

reg.exec(str);//-> ["{0}", "0",...] 第一项大正则匹配的结果，第二项第一个分组匹配的结果，也就是使用exec可以捕获到小分组匹配的内容
reg.exec(str);//-> ["{1}", "1",...]

str.match(reg);//->["{0}", "{1}"] 使用match只能捕获到大正则匹配的，小分组匹配的获取不到
```

```
/*
* match并不是所有情况下，都捕获不到分组的内容，当只需要捕获一次就可以完成的时候(或者不加g的时候)，match获取的结果和exec一样
*/
var reg=/^(\d{6})(\d{4})(\d{2})(\d{2})\d{2}(\d)(?:\d|X)$/g;//->身份证号码的正则
var str="130828198802240761";
str.match(reg);//->["130828198802240761", "130828", "1988", "02", "24", "6", index: 0, input: "130828198802240761"]

/*
* test在某些时候，也是实现捕获的：test匹配的时候，也是把符合的找到了，我们就可以使用一些特殊的手段，把查找的内容取出来
*/
var str = "my name is {0},I am {1} years old~~";
var reg = /\{(\d+)\}/g;
reg.test(str);
RegExp.$1//->获取第一次捕获的时候，第一次分组中的内容（$1） =>"0"

reg.test(str);
RegExp.$1//->获取第一次捕获的时候，第一次分组中的内容（$1） =>"1"并且执行test，如果设置了g，也是可以修改lastIndex的

reg.exec(str);//->null
```

解决正则的懒惰性

```
> var reg = /\d+/g;//->包含1到多个数字
  var str = 'zhufeng2017peixun2018';
undefined
> reg.exec(str);
▶ ["2017", index: 7, input: "zhufeng2017peixun2018"]
> reg.lastIndex
11
> reg.exec(str);
▶ ["2018", index: 17, input: "zhufeng2017peixun2018"]
> reg.lastIndex
21
> var reg = /\d+/g;//->包含1到多个数字
  var str = 'zhufeng2017peixun2018';
undefined
> reg.exec(str);
▶ ["2017", index: 7, input: "zhufeng2017peixun2018"]
> reg.lastIndex
11
> reg.exec(str);
▶ ["2018", index: 17, input: "zhufeng2017peixun2018"]
> reg.exec(str);
null
```

```
> var reg = /\d+/g;//->包含1到多个数字
  var str = 'zhufeng2017peixun2018';
undefined
> dir(reg)
  ▶ /\d+/g
undefined
> reg.global
true
> var reg = /\d+/;
undefined
> reg.global
false
```

在RegExp的原型上扩展一个方法：myExecAll

```
function myExecAll(str){
    if(!this.global){
    return this.exec(str);
    }
    var ary = [];
    var res = this.exec(str);
    while(res){
        ary[ary.length] = res;
        res = this.exec(str);
    }
    return ary;
}
```

```
> var reg = /\d+/g;//->包含1到多个数字
  var str = 'zhufeng2017peixun2018';
undefined
> reg.myExecAll(str)
▶ (2) [Array(1), Array(1)]
> var reg = /\d+/;
undefined
> reg.myExecAll(str)
▶ ["2017", index: 7, input: "zhufeng2017peixun2018"]
> var reg = /\d+/g;//->包含1到多个数字
  var str = 'zhufeng2017peixun2018hahah2019hehehhe2020';
undefined
> reg.myExecAll(str)
▶ (4) [Array(1), Array(1), Array(1), Array(1)]
```

match

```
> var reg = /\d+/g;//->包含1到多个数字
  var str = 'zhufeng2017peixun2018hahah2019hehehhe2020';
undefined
> str.match(reg)
▶ (4) ["2017", "2018", "2019", "2020"]

> var reg = /\d+/;
undefined
> var str = 'zhufeng2017peixun2018hahah2019hehehhe2020';
undefined
> str.match(reg)
▶ ["2017", index: 7, input: "zhufeng2017peixun2018hahah2019hehehhe2020"]
>
```

不加g

exec/match

```
> var reg = /\{(\d+)\}/g;
undefined
> var str = "my name is {0},I am {1} years old~~";
undefined
> reg.exec(str)
▶ (2) ["{0}", "0", index: 11, input: "my name is {0},I am {1} years old~~"]
> reg.exec(str)
▶ (2) ["{1}", "1", index: 20, input: "my name is {0},I am {1} years old~~"]
> str.match(reg);
▶ (2) ["{0}", "{1}"]
```

test

```
> var str = "my name is {0},I am {1} years old~~";
  var reg = /\{(\d+)\}/g;
< undefined
> reg.test(str);
< true
> RegExp.$1
< "0"
> reg.test(str);
< true
> RegExp.$1
< "1"
> reg.exec(str)
< null
```

> 上面是正则捕获：懒惰性，正则捕获还有一个特点：贪婪性

```
var str = "zhufeng2017peixun2018";
var reg = /\d+/g;

reg.exec(str);//->["2017"...] 每次捕获的时候都是把当前正则匹配的最长结果捕获到 ->`贪婪性`

//->取消贪婪性：在量词元字符后面加一个问号就可以了
//=>问号作用很多
//1、如果放在一个非量词元字符的后面，它本身就是代表出现0次或者1次的量词元字符
//2、如果出现在量词元字符的后面，它本身是取消捕获时候的贪婪性
var str = "zhufeng2017peixun2018";
var reg = /\d+?/g;
reg.exec(str);

//3、?:只匹配不捕获
//4、?=:正向预查
//5、?!:负向预查
```

作业：

1、复习（第一周和第二周掌握不扎实的都要好好的复习）

2、表格排序（重点）

3、正则基础的复习，回去后看视频 第二周第三节：课件7/9/10 11补课

补课时间

1-下周二：继承

2-下周三：数据类型检测

7：30~9：30

预习：

下周重点：js盒子模型、图片延迟加载、DOM库…