订阅—发布者一体模式模拟 Angular.js 中的双向绑定

2016.9.20 于京东一面后

9.19 日下午京东一面结束前，一面面试官对“拼 K”后台管理系统的高频数据交互特点，建议我了解一下 Angular.js 中数据绑定的特性。9.19 日晚及 9.20 日晚，我简单研究了一下 Angular.js 中双向绑定特性，依据自己的理解，采用原生 js 对此特性进行实现。

1 底层模块——传统的观察者模式

1.1 发布者类

发布者在观察者模式中占主导地位，拥有主动权，如：添加订阅者、删除订阅者、发布消息、维护订阅者列表等功能。js 实现如下：

```
//发布者类
function Publisher(){
    this.observers = [];    //观察者列表
    this.state = "";        //待发布的消息
}

//  发布者方法1：添加订阅消息的“订阅者”
Publisher.prototype.addOb=function(observer){
    var flag = false;
    for (var i = this.observers.length - 1; i >= 0; i--) {
        if(this.observers[i]===observer){
            flag=true;
        }
    };
    if(!flag){
        this.observers.push(observer);
    }
}
```

```
// 发布者方法2：移除某个订阅消息的“订阅者”，即：退订
Publisher.prototype.removeOb=function(observer){
    var observers = this.observers;
    for (var i = 0; i < observers.length; i++) {
        if(observers[i]===observer){
            observers.splice(i,1);
        }
    };
}
```

```
// 发布者方法3：发布消息
    Publisher.prototype.notice=function(){
        var observers = this.observers;
        for (var i = 0; i < observers.length; i++) {     //遍历所有订阅消息的“订阅者”
            observers[i].update(this.state);     //对其发布消息
        };
    }
```

1.2 订阅者类

订阅者在观察者模式中处于被动地位，只需要被动地对发布者推送的消息进行处理就可以了，以下是其 js 实现：

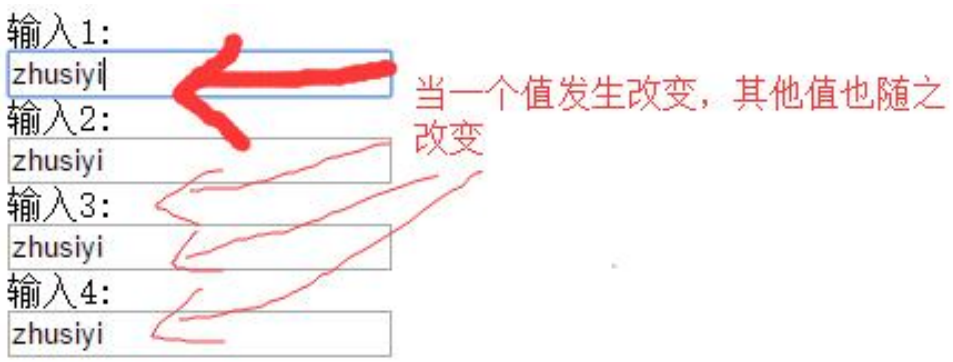
```
//  订阅者类
    function Subscribe(){
        // 仅有一个方法，接收到推送的消息后，更新数据。
        this.update = function(data){
            eleIdGroup[this.id].value = data;    //在这里体现为改变自己的值
        };
    }
```

这里将订阅者对消息处理的方式规定为，将自身的值（input 的 value）设置为推送过来的消息的值(data)。

2 中层模块——“观察者/推送者”一体模式

前面也提到，在 Angular.js 中，双向绑定时任意一个值发生变化，另一个元素也会随之发生变化。实现如下：

```
<body ng-app="myApp">
输入1:<input type="text" ng-model="name"/>
输入2:<input type="text" ng-model="name"/>
输入3:<input type="text" ng-model="name"/>
输入4:<input type="text" ng-model="name"/>
</body>
<script src="http://cdn.static.runoob.com/libs/
<script>
    angular.module('myApp', []);
</script>
```



根据这种特性，我们可以将该特性以观察者模式的角度进行思考：1、每个元素的改变都

会对其他元素造成影响——每个元素有“发布属性”2、每个元素都能对其他元素的改变做出响应——每个元素都是“订阅者”。由以上两种特性，我决定实现一种“订阅/发布”合二为一的功能，即每个元素拥有推送的功能，也拥有对其它元素的推送做出反应的订阅功能。js 实现如下：

```
初始化“订阅—发布”者
function init(eleArray){
    var i,j;
    for(i=0;i<eleArray.length;i++) {
        eleArray[i].pub = new Publisher();  //为每一个元素添加一个pub发布者属性
        eleArray[i].sub = new Subscribe();  //为每一个元素添加一个sub订阅者属性
        eleArray[i].sub.id = i;             //重要！辨别该元素的id
    }
    for(i=0;i<eleArray.length;i++){
        for(j=0;j<eleArray.length;j++){
            eleArray[i].pub.addOb(eleArray[j].sub);  //二重循环，为每一个元素的
        }                                            //发布属性添加订阅者
    }
}
init(eleIdGroup);
```

3 上层——配置、设置触发事件及初始化

底层模块编写完毕，剩下的就是在上层对其进行正确地配置、并设置相应的触发事件。

3.1 触发事件的编写

上层需对元素设置一些触发事件，这里因为举例的元素是 input，所以自然使用 oninput 进行模拟。考虑到在实际使用中有可能会出现新加到 DOM 树的元素也需要绑定，所以这里使用了事件委托机制。

```
document.oninput = function(e){          //事件委托
    var e = e || window.event;
    var target = e.target || e.srcElement;
    if(eleIdGroup.indexOf(target)!=-1){      //如果“订阅/发布者”列表里面有这个input元素
        target.pub.state = target.value;     //设置消息
        target.pub.notice();                 //推送消息给所有订阅者
    }
};
```

3.2 初始化配置

配置就比较简单了，只需要得到需要绑定的元素，将其放入“订阅/发布者”列表就可以了。js 实现如下：

```
var input1 = document.getElementById('input1');
var input2 = document.getElementById('input2');
var input3 = document.getElementById('input3');
var input4 = document.getElementById('input4');

var eleIdGroup = [input1,input2,input3,input4];
```

4　状态更改模块（工具类）

1、2、3 点已经解释了 js 实现数据绑定的全部过程，接下来简要描述演示程序中的按钮状态转移实现。

4.1 核心：状态缓存

这里我使用闭包来实现按键状态的缓存，在演示程序的例子中，按键有两个状态：已绑定、未绑定。所以我将状态初始值设为 1，接着每次返回其相反数，这样就可以实现两个状态相互转换。js 实现如下：

```
var Cache = function(){     //缓存类，闭包实现按钮的状态
    var status = 1;
    return {
        getCache:function(){
            return status*=-1;
        }
    }
};
```

4.2 状态转移使相应元素绑定/解绑

```
(function(){
    var btnGroup = document.getElementsByTagName('button'),
            i;
    for(i=0;i<btnGroup.length;i++){
        (function(num){                    //闭包实现为每个button设置点击事件
            btnGroup[num].onclick = function(e){
                var e = e || window.event,
                    target = e.target || e.srcElement;
                if(btnIdGroup.indexOf(target)!=-1){
                    var i = target.status.getCache(),
                            ele = document.getElementById('input'+(num+1));
                    if (1 != i) {                                   //如果之前是绑定状态
                        if (eleIdGroup.indexOf(ele) != -1) {        //列表里面又有该元素
                            eleIdGroup.splice(eleIdGroup.indexOf(ele), 1);  //对其解绑，从列表中删去
                            target.style.backgroundColor = 'red';
                            target.innerHTML = '已解绑';
                        }
                    } else {                                        //如果之前是未绑定状态
                        if(eleIdGroup.indexOf(ele) == -1) {         //列表里面又有该元素
                            eleIdGroup.push(ele);                   //将其添加进列表，绑定
                            target.style.backgroundColor = 'green';
                            target.innerHTML = '已绑定';
                        }
                    }
                    init(eleIdGroup);                               //重新初始化各元素发布属性的订阅者
                }
            };
        })(i);
    }
})();
```

5 反思及改进

模块化、顶层以插件形式进行配置、调用