观察者模式

什么是观察者模式？

- **高程三：**

> 观察者模式由两类对象组成：主体和观察者。主体主要负责发布事件，同时观察者通过订阅这些时间来观察主体。该模式的一个关键概念是主体并不知道观察者的任何事情，也就是说它可以对存在并正常运作即使观察者不存在。从另一个方面来说，观察者知道主体并能注册时间的回调函数（事件处理程序）。（说了一堆，我特么一点没读懂，满脸懵逼）

- **耿大爷说：**

> 说的内容忘了，不过看课件的时候有些内容还是记得，写的时候都是模拟 `addEventListener` 来写的, `addEventListener` 这个方法它就是一个接口，它把 `click` 所表示的这个行为，和 `fn` 这个行为联系在一起了，有了 `addEventListener` 这个方法，就可以实现当 `click` 发现的时候， `fn` 也发生!
>
> 如果能像addEventListener一样，来解决两个不同的事（行为、function）之间的协作（A发生则B随着A的发生而发生，简言之就是A执行，则B就会执行）—-然后我们封装了 `on,run,off`
>
> 1. down，move,up看成是主体行为.//这三个应该知道啥吧？不知道？那还不去看看！！！
> 2. 把约定down，move,up的方法看成仆从行为
> 3. on,run是down,move,up的秘书
> 4. on就负责登记记录约定某一个行为的其它行为
> 5. run负责当某一个行为发生的时候，去通知由on已经登记好的那些行为,run什么时候通知（执行）

- **某个骚猪说：**

> 订阅和发布就好比我们看杂志或者订阅RSS，再比如我们浏览网页，只有很少人会只浏览一个网页，同样也只有很少的网站只为一个人而开，我们每天浏览很多网站，同样这些网站也在被很多人浏览，大家相互没有必然的联系，只有我们在想访问这些网站的时候，才会浏览网页，此时对于网页而言，我就是订阅者，反之对于我来说，网页就是发布者。

- **我说：**

> 我总感觉观察者是一种管理模式，为了方便管理。可以保持主线的内容不受支线内容的破坏，而且我可以随便写支线内容，想怎么玩就怎么玩，而且我想让你滚犊子的时候就必须滚蛋（清除掉）！拿耿大爷那个烧水例子：

```
function noodles(e){
    console.log("煮面条","此时此刻的煮面条是由"+e.message+"触发的");
}
function drink(){
    console.log("喝水");
}
function bath(){
    console.log("洗澡");
}
//普通的形式
Kettle.prototype.boiling=function(){
    var that=this;
    window.setTimeout(function(){
        console.log("水开了！");//等了一万年水终于开了，我要开始泡面了，喝汤了，洗澡了
        noodles();
        drink();
        bath();
        //....还有很多！
        //可是在我要泡面的时候、朋友说请吃大餐，那特么谁还吃面条子，肯定吃大餐呀，然后就得清除掉我刚刚写的东西，一件件的删除！你可以试想一下，在项目中，如果代码量特别多，要修改的地方特别多，你这样写的话得改多久，改的时候肯定会骂街，说一些（！@#！￥！@#%#@￥！%！@）的话；
    },6000);
};

订阅发布模式呢？
Kettle.prototype.boiling=function(){
    var e={type:"boiling",message:"kettle-boiling"};
    var that=this;
    window.setTimeout(function(){
        console.log("水开了！");
        that.run();//run 是啥我也不知道哈！自己想！还有为什么要在这写？反正我是不知道！
    },6000);
};
var kettle1=new Kettle;
kettle1.on("boiling",noodles);
kettle1.on("boiling",drink);
kettle1.on("boiling",bath);
//把我要做的事提前订阅好，静等水开，然后我干我要做的事，当然在这个时候你也可以盼望有人请你吃大餐，因为修改行程很简单，我有秘书，有事秘书干！按下：110...
秘书么？我不吃面条了，我要吃大餐，我要取消行程：kettle1.off(.......)！
这是110！不是你秘书，也不是你妈！！
```

这样就是我理解的哈，那么 `你理解的` 呢？

拖拽

第一个版本（面向对象版）：

> 要用面向对象封装东西肯定要明白几点规则
>
> 1. this的指向。则是类的方法，this都要指向这个类当前的实例
> 2. 一个类就是一个具体完整功能的模块（这个在学完node应该理解的更深了吧），这个模块（类）上定义了此功用所需要所有的属性和方法
> 3. 我们在定义这个类的时候，最其码要实现它的最基本的功能。还要留好扩展的余地，以便升级
> 4. 把全局变量都作为私有属性
> 5. 全局函数都作为公有方法
> 6. 注意constructor指向（这个写不写分情况哈，啥时候写自己想吧）
> 7. init（ ）初始化函数中，就是我们代码执行的思路（这个我个人感觉只适合那种只实现一个固定效果的，不好修改，感觉有点不好用—-当然最主要的原因是我特么不会啊，装逼装不成^^*O(∩∩)O*哈哈~）

```
    function MyEvent() {//方便添加自定义事件用！！本人有点蠢哈，得写两个，理论上可以合并的！！
    }
    MyEvent.prototype.on = function (type, fn) {//订阅事件
        if (!this["myEvent" + type]) {
            this["myEvent" + type] = [];
        }
        var ary = this["myEvent" + type];
        for (var i = 0, len = ary.length; i < len; i++) {
            if (ary[i] == fn)return;
        }
        ary.push(fn)
    };
    MyEvent.prototype.run = function (myE, sysE) {//myE是个对象类型的且myE.type和on里的type是一样的(慢慢认真的想想哈)
        var ary = this["myEvent" + myE.type];
        if (ary) {
            for (var i = 0, len = ary.length; i < len; i++) {
                if (typeof ary[i] === "function") {
                    ary[i].call(this, myE, sysE)
                } else {
                    ary.splice(i, 1);//用splice的时候一定要记住，`数组会坍塌``数组会坍塌``数组会坍塌`
                    i--;
                }
            }
        }
    };
    MyEvent.prototype.off = function (type, fn) {
        var ary = this["myEvent" + type];
        if (ary) {
            for (var i = 0, len = ary.length; i < len; i++) {
                if (ary[i] == fn) {
                    ary[i] = null;
                    return;
                }
            }
        }
    };
    function on(ele, type, fn) {//事件的on，run，off！！不会写了！！♪(^∇^*)，就是不会写了！
        return ele.addEventListener(type, fn, false);
    }
    function off(ele, type, fn) {
        return ele.removeEventListener(type, fn, false);
    }
```

开始拖拽了哈！！！

```
function Drag(ele) {
        this.ele = ele;
        this.x = null;
        this.y = null;
        this.l = null;
        this.t = null;
       this.DOWN = this.down.bind(this);//只是简单的注释掉了这一行，为啥会死掉？从这可以考虑看看在封装构造函数的时候，注意哪些东西？
        this.MOVE = this.move.bind(this);
        this.UP = this.up.bind(this);
        on(ele, "mousedown", this.DOWN);
    }
    //继承
    Drag.prototype = new MyEvent();//这种继承有啥缺点？还有那种继承？能不能想起来？O(∩_∩)O哈哈~
    Drag.prototype.down = function (e) {
        this.x = e.pageX;
        this.y = e.pageY;
        this.l = this.ele.offsetLeft;
        this.t = this.ele.offsetTop;
        on(document, "mousemove", this.MOVE);
        on(document, "mouseup", this.UP);//为啥不是this.on？？？
        this.on("dragstart",this.addBorder);//为啥是this.on？？？
        this.run.call(this, {type: "dragstart"});//如果你明白上两个问题，可是问题又来了！！为啥只需要绑定一次run，我可是明明on了两个方法的？？？！！（你可能会说，你瞎啊，明明绑定了一个！好吧，我不说话了，我瞎哈）
        e.preventDefault();//为啥加这个！！？
    };
    Drag.prototype.move = function (e) {
        this.ele.style.left = e.pageX - this.x + this.l + "px";
        this.ele.style.top = e.pageY - this.y + this.t + "px";
    };
    Drag.prototype.up = function () {
        off(document, "mousemove", this.MOVE);
        off(document, "mouseup", this.UP);
        this.on("dragend",removeBorder);//为啥没有移除掉border呢？？什么bug？
        this.run.call(this, {type: "dragend"});//为什么在这里添加自定义事件的run，不加不行么？为啥是this.run？
    };
    Drag.prototype.backG = function () {
        this.on("dragstart", this.addBac);//绑定自定义事件
        this.on("dragend", this.removeBac);//解除绑定
    };
    Drag.prototype.addBac = function () {
        console.log(this);
        this.ele.style.background = "pink";//这个里为什么要写this.ele！！！这里的this打印出来明明是div，为什么我不写ele不能出效果？---会更深的体会到构造函数的注意事项（反正我是感觉到了）
```

```
    };
    Drag.prototype.removeBac = function () {
        this.ele.style.background = "green"
    };
    Drag.prototype.addBorder=function(){
        this.ele.style.border="2px dashed yellow";
    };
    Drag.prototype.removeBorder=function(){
        this.ele.style.border="none";
    };
    var oDiv = document.getElementById("div");
    var a = new Drag(oDiv);
    a.backG();//这里为什么还要运行一下，不运行不行么？明明都绑定了！！！
    //不知道上面继承的问题有木有考虑哈，可是慈悲的我还是会告诉你答案的，上面那种继承的缺点：会覆盖原来的Kettle类默认的原型对象，并且这种继承方法只能写在定义原型方法之前
    //Drag.prototype.__proto__=myEvent.prototype;//有兼容问题，但是更安全，并且此行表达式的位置不限
```

第一版本结束了，有木有什么感受哈~~~华丽的分割线马上来了哈

第二个版本（高程三）：

> 这个版本我在高程三上看的，可以自己去看看哈（但是我建议直接看我的，因为 高程三 上的不对，凭空少内容，特么的，为了找到它缺少的东西我看了好多遍，关键是没有！ O(≧口≦)O,宝宝心里苦 ）这个版本就自己琢磨琢磨吧，不过我会说一下我对这个版本的理解！！当然 每个人的理解都不一样 ，我真的很希望 听到你说 ，你特么傻逼啊， 这明显不对 ，然后 指着我骂出来 ，然后我会 默默的求着你 告诉我 对 的东西哈~~~~

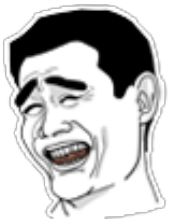
```
function EventTarget() {
        this.handlers = {};//为什么我在第一版没有这个？
    }
    EventTarget.prototype = {//自定义事件
        constructor: EventTarget,//为啥写这个？我也不知道哈！
        on: function (type, fn) {
            if (!this.handlers[type]) {
                this.handlers[type] = [];
            }
            var ary = this.handlers[type];
            for (var i = 0, len = ary.length; i < len; i++) {
                if (ary[i] == fn)return;
            }
            this.handlers[type].push(fn);
        },
        fire: function (event) {//这里的event是个对象！！
            if (!event.target) {
                event.target = this;//这个是啥，我是真的不知道，这个真心的希望告诉我为啥，第一版的那些还是懂的这个，真的不懂！！
            }
            var ary = this.handlers[event.type];
            if (this.handlers[event.type]  instanceof  Array) {//这是啥意思？应该可以简写吧？
                for (var i = 0, len = ary.length; i < len; i++) {
                    ary[i].call(this, event)
                }
            }
        },
        off: function (type, fn) {
            var ary = this.handlers[type];
            if (ary) {
                for (var i = 0, len = ary.length; i < len; i++) {
                    if (ary[i] == fn) {
                        //ary.splice(i,1);//为啥在这删除不可以？会发生什么？在63行删除却可以呢？
                        break;
                    }
                }
                ary.splice(i, 1);//结合上个问题感受下
            }
        }
    };
    //EventUtil是事件的on,run,off 如果你在第一版的时候没写，可以看看这个哈！O(∩_∩)O哈哈~，而且城里人超会玩，这个是单例模式的哈！！！不要问我为什么这么写，我任性好吧？？(∩_∩)
    EventUtil = {
        on: function (ele, type, fn) {
            if (ele.addEventListener) {
```

```
                return ele.addEventListener(type, fn, false)
            }
            if (!ele["myEvent" + type]) {
                ele["myEvent" + type] = [];
                ele.attachEvent("on" + type, function () {
                    run.call(ele);
                })
            }
            var ary = ele["myEvent" + type];
            for (var i = 0; i < ary.length; i++) {
                if (ary[i] == fn) {
                    return;
                }
            }
            ary.push(fn);
        },
        run: function () {
            var e = window.event;
            var type = e.type;
            if (!e.target) {
                e.target = e.srcElement;
                e.stopPropagation = function () {
                    e.cancelBubble = true
                };
                e.preventDefault = function () {
                    e.returnValue = false;
                }
            }
            var ary = ele["myEvent" + type];
            if (ary) {
                for (var i = 0; i < ary.length; i++) {
                    if (typeof  ary[i] === "function") {
                        ary[i].call(this, e)
                    } else {
                        ary.splice(i, 1);
                        i--;//在用splice的时候 一定要记得数组会塌陷、、、
                    }
                }
            }
        },
        off: function (ele, type, fn) {
            if (ele.removeEventListener) {
                ele.removeEventListener(type, fn, false);
                return
            }
            var ary = ele["myEvent" + type];
            if (ary) {
                for (var i = 0; i < ary.length; i++) {
                    if (ary[i] === fn) {
```

```
                        ary[i] = null;
                        return
                    }
                }
            }
        }
    };
    //开始拖拽了哈
    var DragDrop = function () {
        var dragging = null;
        function handleEvent(event) {
            switch (event.type) {//确定事件类型
                case "mousedown":
                    if (event.target.className.indexOf("draggable")
> -1) {//这个不错，通过判断这个事件源上是否有这个属性，有的话，就可以拖拽，----这个你想到了什么？如果你能想到angular里的拖拽，那么恭喜你，你angular学的不错哈！！！
                        dragging = event.target;//获取事件源
                        dragging.x = event.clientX;//这个可以考虑下，因为我在写的时候这遇到问题了，我说下我的思路哈（我是直接var的 这几个变量，可是不行！原因是啥，为什么咱们普通的版本的可以var，还有第一版把这些放在哪了？？？我希望你自己考虑下，自己想的收获真的很多的！然后才是这个自定义属性，好吧主要是我蠢，我承认哈）
                        dragging.y = event.clientY;
                        dragging.l = dragging.offsetLeft;
                        dragging.t = dragging.offsetTop;
                    }
                    break;
                case "mousemove":
                    if (dragging !== null) {
                        dragging.style.left = event.clientX - dragg
ing.x + dragging.l + "px";
                        dragging.style.top = event.clientY - draggi
ng.y + dragging.t + "px";
                    }
                    break;
                case "mouseup":
                    dragging = null;
                    break
            }
            event.preventDefault();
        }
        return {//公共接口
            enable: function () {
                EventUtil.on(document, "mousedown", handleEvent);
                EventUtil.on(document, "mousemove", handleEvent);
                EventUtil.on(document, "mouseup", handleEvent)
            },
            disable: function () {
```

```
                EventUtil.off(document, "mousedown", handleEvent);
                EventUtil.off(document, "mousemove", handleEvent);
                EventUtil.off(document, "mouseup", handleEvent)
            }
        }
    }();
    DragDrop.enable();//只要写了 这行代码，就可以实现拖拽了
    //DragDrop.disable();
```

> 说下我对这个版本的理解哈：
>
> 优点：
>
> 1. 这个是用 `单例模式` 写的，每个方法都是 `模块` 和 `模块` 之间调用的！
> 2. 自定义事件的 `off` 的方法比较新（反正我是这么觉得的）
> 3. 绑定的是 `document` ，并且只要有 `draggable` 的属性就可以实现拖拽！—— `angular` 的拖拽是不是就是通过 `A` –（ `Attribute` ）来找的
> 4. 公共接口这个接触的少，可以试着去看看（其实我是不懂的！）
>
> 缺点：
>
> 1. 添加 `自定义事件` 的时候没有 `构造函数` 方便， `不` 方便管理！（你可以试着自己去用这个写一下，因为等我不会，在这等着你教我呢）
> 2. 如果 `移除` 的话，会把 `所有` 的 `拖拽` 事件 `都移除掉` ~~~有点霸道，一荣俱荣，一损俱损~！！！

华丽的分割线又来了哈！！哈 

第三版本（angular）：

> 说实话这个版本我不想在说啥了哈，我感觉挺简单的

```
var app = angular.module('appModule', []);
    app.directive("mydrag", function () {
        return {
            restrict: 'A',
            link: function (scope, element, attrs) {
                element.on("mousedown", function (e) {
                    var l = e.pageX - element[0].offsetLeft,
                            t = e.pageY - element[0].offsetTop;
                    angular.element(document).on("mousemove", funct
ion (e) {
                        element.css("left", (e.pageX - l) + "px");
                        element.css("top", (e.pageY - t) + "px");
                    });
                    angular.element(document).on("mouseup", functio
n () {
                        angular.element(document).off()
                    });
                    e.preventDefault();
                })
            }
        }
    })
不用找注释，因为一点都没有！！！
```

华丽的分割线又来了哈！！哈啦啦啦

总结：

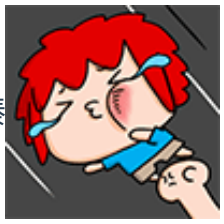
> 如果你能看到这里，我相信一定是真爱哈！！
> 如果我上面写的你能好好看看，是不是对你有那么一丢丢帮助？
> 如果没有的话，你也别揍我就好，虽然我浪费了你不少时间！

观察者模式是啥呢？

1. 在观察者模式由两个词出现的最多，`订阅`，`发布`！
    - 什么是`订阅`？

        在主线任务上，把我要执行的别的事先`添加`到主线任务上！
    - 什么是`发布`？

        在主线任务执行的时候，会把我之前添加到主线任务上的事也给执行了

        （`这些都是我自己理解的哈`，错了别揍我，揍 

        ）
        1. 那么高程三里的观察者由两类`对象`组成：`主体` `观察者`！

    - 什么是`主体`？

        主体是观察者观察的对象（看到这句话是不是想揍我？对，没错，就是废话）—-（我认为就是事件主干，代码核心内容）
    - `主体`应该具备的三个特性：
        1. 持有监听的观察者的引用
        2. 支持增加和删除观察者
        3. 主题状态改变，通知观察者

    - 什么是`观察者`？

        当主体发生变化，收到通知进行具体的处理是观察者必须具备的特征。（百度的自己去理解）—-（就是一些我想增加的额外的一些方法，当然最主要得实现我想加就加，想删除就删除这么任性的想法）
    - 观察者模式的好处：
        1. 观察者增加或删除无需修改主体的代码，只需调用主体对应的增加或者删除的方法即可。
        2. 主体只负责通知观察者，但无需了解观察者如何处理通知。举个例子，送奶站只负责送递牛奶，不关心客户是喝掉还是洗脸（当然也可以洗脚）。
        3. 观察者只需等待主体通知，无需观察主体相关的细节。还是那个例子，客户只需关心送奶站送到牛奶，不关心牛奶由哪个快递人员，使用何种交通工具送达。

`我上面说的这么多，是不是对高程三说的内容有了新的了解？彼此间相互独立！互不干扰，你干你的，我干我的，谁也不打扰谁，有事通知下就行---有事真上！！！`

终于完事啦！如果想要代码可以和我说哈，
关注下我的 **git**
https://github.com/sunha1yang

希望能对你有所帮助哈！！哈哈哈