# 前端基础进阶（十）：面向对象实战之封装拖拽对象

终于

前面几篇文章，我跟大家分享了JavaScript的一些基础知识，这篇文章，将会进入第一个实战环节：利用前面几章的所涉及到的知识，封装一个拖拽对象。为了能够帮助大家了解更多的方式与进行对比，我会使用三种不同的方式来实现拖拽。

- 不封装对象直接实现；
- 利用原生JavaScript封装拖拽对象；
- 通过扩展jQuery来实现拖拽对象。

> 本文的例子会放置于codepen.io中，供大家在阅读时直接查看。如果对于codepen不了解的同学，可以花点时间稍微了解一下。

拖拽的实现过程会涉及到非常多的实用小知识，因此为了巩固我自己的知识积累，也为了大家能够学到更多的知识，我会尽量详细的

将一些细节分享出来，相信大家认真阅读之后，一定能学到一些东西。

### 1、如何让一个DOM元素动起来

我们常常会通过修改元素的top，left，translate来其的位置发生改变。在下面的例子中，每点击一次按钮，对应的元素就会移动5px。大家可点击查看。

点击查看一个让元素动起来的小例子

> 由于修改一个元素top/left值会引起页面重绘，而translate不会，因此从性能优化上来判断，我们会优先使用translate属性。

### 2、如何获取当前浏览器支持的transform兼容写法

transform是css3的属性，当我们使用它时就不得不面对兼容性的问题。不同版本浏览器的兼容写法大致有如下几种：

```
['transform', 'webkitTransform', 'MozTransform',
'msTransform', 'OTransform']
```

因此我们需要判断当前浏览器环境支持的transform属性是哪一种，方法如下：

```
// 获取当前浏览器支持的transform兼容写法
function getTransform() {
    var transform = '',
        divStyle = document.createElement('div').style,
        // 可能涉及到的几种兼容性写法，通过循环找出浏览器识别的那一个
        transformArr = ['transform', 'webkitTransform', 'MozTransform'

        i = 0,
        len = transformArr.length;

    for(; i < len; i++)  {
        if(transformArr[i] in divStyle) {
```

```
            // 找到之后立即返回，结束函数
            return transform = transformArr[i];
        }
    }

    // 如果没有找到，就直接返回空字符串
    return transform;
}
```

该方法用于获取浏览器支持的transform属性。如果返回的为空字符串，则表示当前浏览器并不支持transform，这个时候我们就需要使用left，top值来改变元素的位置。如果支持，就改变transform的值。

### 3、如何获取元素的初始位置

我们首先需要获取到目标元素的初始位置，因此这里我们需要一个专门用来获取元素样式的功能函数。

但是获取元素样式在IE浏览器与其他浏览器有一些不同，因此我们需要一个兼容性的写法。

```
function getStyle(elem, property) {
    // ie通过currentStyle来获取元素的样式，其他浏览器通过getComputedStyle来
    return document.defaultView.getComputedStyle ? document.defaultVie
}
```

有了这个方法之后，就可以开始动手写获取目标元素初始位置的方法了。

```
function getTargetPos(elem) {
    var pos = {x: 0, y: 0};
    var transform = getTransform();
    if(transform) {
        var transformValue = getStyle(elem, transform);
        if(transformValue == 'none') {
            elem.style[transform] = 'translate(0, 0)';
```

```
            return pos;
        } else {
            var temp = transformValue.match(/-?\d+/g);
            return pos = {
                x: parseInt(temp[4].trim()),
                y: parseInt(temp[5].trim())
            }
        }
    } else {
        if(getStyle(elem, 'position') == 'static') {
            elem.style.position = 'relative';
            return pos;
        } else {
            var x = parseInt(getStyle(elem, 'left') ? getStyle(elem, '
            var y = parseInt(getStyle(elem, 'top') ? getStyle(elem, 't
            return pos = {
                x: x,
                y: y
            }
        }
    }
}
```

在拖拽过程中，我们需要不停的设置目标元素的新位置，这样它才会移动起来，因此我们需要一个设置目标元素位置的方法。

```
// pos = { x: 200, y: 100 }
function setTargetPos(elem, pos) {
    var transform = getTransform();
    if(transform) {
        elem.style[transform] = 'translate('+ pos.x +'px, '+ pos.y +'p
    } else {
        elem.style.left = pos.x + 'px';
        elem.style.top = pos.y + 'px';
    }
    return elem;
}
```

## 5、我们需要用到哪些事件?

在pc上的浏览器中，结合mousedown、mousemove、mouseup这三个事件可以帮助我们实现拖拽。

- mousedown 鼠标按下时触发
- mousemove 鼠标按下后拖动时触发
- mouseup 鼠标松开时触发

> 而在移动端，分别与之对应的则是touchstart、touchmove、touchend。

当我们将元素绑定这些事件时，有一个事件对象将会作为参数传递给回调函数，通过事件对象，我们可以获取到当前鼠标的精确位置，鼠标位置信息是实现拖拽的关键。

> 事件对象十分重要，其中包含了非常多的有用的信息，这里我就不扩展了，大家可以在函数中将事件对象打印出来查看其中的具体属性，这个方法对于记不清事件对象重要属性的童鞋非常有用。

## 6、拖拽的原理

当事件触发时，我们可以通过事件对象获取到鼠标的精切位置。这是实现拖拽的关键。当鼠标按下(mousedown触发)时，我们需要记住鼠标的初始位置与目标元素的初始位置，我们的目标就是实现当鼠标移动时，目标元素也跟着移动，根据常理我们可以得出如下关系：

```
移动后的鼠标位置 - 鼠标初始位置 = 移动后的目标元素位置 - 目标元素的初始位置
```

如果鼠标位置的差值我们用dis来表示，那么目标元素的位置就等于：

移动后目标元素的位置 = dis + 目标元素的初始位置

通过事件对象，我们可以精确的知道鼠标的当前位置，因此当鼠标拖动(mousemove)时，我们可以不停的计算出鼠标移动的差值，以此来求出目标元素的当前位置。这个过程，就实现了拖拽。

而在鼠标松开(mouseup)结束拖拽时，我们需要处理一些收尾工作。详情见代码。

## 7、我又来推荐思维导图辅助写代码了

常常有新人朋友跑来问我，如果逻辑思维能力不强，能不能写代码做前端。我的答案是：能。因为借助思维导图，可以很轻松的弥补逻辑的短板。而且比在自己头脑中脑补逻辑更加清晰明了，不易出错。

上面第六点我介绍了原理，因此如何做就显得不是那么难了，而具体的步骤，则在下面的思维导图中明确给出，我们只需要按照这个步骤来写代码即可，试试看，一定很轻松。

使用思维导图清晰的表达出整个拖拽过程我们需要干的事情

## 8、代码实现

## part1、准备工作

```
// 获取目标元素对象
var oElem = document.getElementById('target');

// 声明2个变量用来保存鼠标初始位置的x，y坐标
var startX = 0;
var startY = 0;

// 声明2个变量用来保存目标元素初始位置的x，y坐标
var sourceX = 0;
var sourceY = 0;
```

## part2、功能函数

因为之前已经贴过代码，就不再重复

```
// 获取当前浏览器支持的transform兼容写法
function getTransform() {}

// 获取元素属性
function getStyle(elem, property) {}

// 获取元素的初始位置
function getTargetPos(elem) {}

// 设置元素的初始位置
function setTargetPos(elem, potions) {}
```

## part3、声明三个事件的回调函数

这三个方法就是实现拖拽的核心所在，我将严格按照上面思维导图中的步骤来完成我们的代码。

```
// 绑定在mousedown上的回调，event为传入的事件对象
function start(event) {
    // 获取鼠标初始位置
    startX = event.pageX;
    startY = event.pageY;

    // 获取元素初始位置
    var pos = getTargetPos(oElem);

    sourceX = pos.x;
    sourceY = pos.y;

    // 绑定
    document.addEventListener('mousemove', move, false);
    document.addEventListener('mouseup', end, false);
}
```

```
function move(event) {
    // 获取鼠标当前位置
    var currentX = event.pageX;
    var currentY = event.pageY;

    // 计算差值
    var distanceX = currentX - startX;
    var distanceY = currentY - startY;

    // 计算并设置元素当前位置
    setTargetPos(oElem, {
        x: (sourceX + distanceX).toFixed(),
        y: (sourceY + distanceY).toFixed()
    })
}

function end(event) {
    document.removeEventListener('mousemove', move);
    document.removeEventListener('mouseup', end);
    // do other things
}
```

OK，一个简单的拖拽，就这样愉快的实现了。点击下面的链接，可以在线查看该例子的demo。

使用原生js实现拖拽

## 9、封装拖拽对象

在前面一章我给大家分享了面向对象如何实现，基于那些基础知识，我们来将上面实现的拖拽封装为一个拖拽对象。我们的目标是，只要我们声明一个拖拽实例，那么传入的目标元素将自动具备可以被拖拽的功能。

在实际开发中，一个对象我们常常会单独放在一个js文件中，这个js文件将单独作为一个模块，利用各种模块的方式组织起来使用。当然这里没有复杂的模块交互，因为这个例子，我们只需要一个模块即可。

为了避免变量污染，我们需要将模块放置于一个函数自执行方式模拟的块级作用域中。

```
;
(function() {
    ...
})();
```

> 在普通的模块组织中，我们只是单纯的将许多js文件压缩成为一个js文件，因此此处的第一个分号则是为了防止上一个模块的结尾不用分号导致报错。必不可少。当然在通过require或者ES6模块等方式就不会出现这样的情况。

我们知道，在封装一个对象的时候，我们可以将属性与方法放置于构造函数或者原型中，而在增加了自执行函数之后，我们又可以将属性和方法防止与模块的内部作用域。这是闭包的知识。

**那么我们面临的挑战就在于，如何合理的处理属性与方法的位置。**

当然，每一个对象的情况都不一样，不能一概而论，我们需要清晰的知道这三种位置的特性才能做出最适合的决定。

- 构造函数中： 属性与方法为当前实例单独拥有，只能被当前实例访问，并且每声明一个实例，其中的方法都会被重新创建一次。
- 原型中： 属性与方法为所有实例共同拥有，可以被所有实例访问，新声明实例不会重复创建方法。
- 模块作用域中：属性和方法不能被任何实例访问，但是能被内部方法访问，新声明的实例，不会重复创建相同的方法。

对于方法的判断比较简单。

因为在构造函数中的方法总会在声明一个新的实例时被重复创建，因此我们声明的方法都尽量避免出现在构造函数中。

而如果你的方法中需要用到构造函数中的变量，或者想要公开，那就需要放在原型中。

如果方法需要私有不被外界访问，那么就放置在模块作用域中。

对于属性放置于什么位置有的时候很难做出正确的判断，因此我很难给出一个准确的定义告诉你什么属性一定要放在什么位置，这需要在实际开发中不断的总结经验。但是总的来说，仍然要结合这三个位置的特性来做出最合适的判断。

如果属性值只能被实例单独拥有，比如person对象的name，只能属于某一个person实例，又比如这里拖拽对象中，某一个元素的初始位置，也仅仅只是这个元素的当前位置，这个属性，则适合放在构造函数中。

而如果一个属性仅仅供内部方法访问，这个属性就适合放在模块作用域中。

> 关于面向对象，上面的几点思考我认为是这篇文章最值得认真思考的精华。如果在封装时没有思考清楚，很可能会遇到很多你意想不到的bug，所以建议大家结合自己的开发经验，多多思考，总结出自己的观点。

根据这些思考，大家可以自己尝试封装一下。然后与我的做一些对比，看看我们的想法有什么不同，在下面例子的注释中，我将自己的想法表达出来。

[点击查看已经封装好的demo]

js 源码

```
;
(function() {
    // 这是一个私有属性，不需要被实例访问
    var transform = getTransform();

    function Drag(selector) {
        // 放在构造函数中的属性，都是属于每一个实例单独拥有
        this.elem = typeof selector == 'Object' ? selector : document.
        this.startX = 0;
        this.startY = 0;
        this.sourceX = 0;
        this.sourceY = 0;

        this.init();
    }


    // 原型
    Drag.prototype = {
        constructor: Drag,

        init: function() {
            // 初始时需要做些什么事情
            this.setDrag();
        },

        // 稍作改造，仅用于获取当前元素的属性，类似于getName
        getStyle: function(property) {
            return document.defaultView.getComputedStyle ? document.de
        },

        // 用来获取当前元素的位置信息，注意与之前的不同之处
        getPosition: function() {
            var pos = {x: 0, y: 0};
            if(transform) {
                var transformValue = this.getStyle(transform);
                if(transformValue == 'none') {
                    this.elem.style[transform] = 'translate(0, 0)';
                } else {
                    var temp = transformValue.match(/-?\d+/g);
                    pos = {
                        x: parseInt(temp[4].trim()),
                        y: parseInt(temp[5].trim())
```

```
            }
        }
    } else {
        if(this.getStyle('position') == 'static') {
            this.elem.style.position = 'relative';
        } else {
            pos = {
                x: parseInt(this.getStyle('left') ? this.getSt
                y: parseInt(this.getStyle('top') ? this.getSty
            }
        }
    }

    return pos;
},

// 用来设置当前元素的位置
setPostion: function(pos) {
    if(transform) {
        this.elem.style[transform] = 'translate('+ pos.x +'px,
    } else {
        this.elem.style.left = pos.x + 'px';
        this.elem.style.top = pos.y + 'px';
    }
},

// 该方法用来绑定事件
setDrag: function() {
    var self = this;
    this.elem.addEventListener('mousedown', start, false);
    function start(event) {
        self.startX = event.pageX;
        self.startY = event.pageY;

        var pos = self.getPosition();

        self.sourceX = pos.x;
        self.sourceY = pos.y;

        document.addEventListener('mousemove', move, false);
        document.addEventListener('mouseup', end, false);
    }

    function move(event) {
```

```
                var currentX = event.pageX;
                var currentY = event.pageY;

                var distanceX = currentX - self.startX;
                var distanceY = currentY - self.startY;

                self.setPostion({
                    x: (self.sourceX + distanceX).toFixed(),
                    y: (self.sourceY + distanceY).toFixed()
                })
            }

            function end(event) {
                document.removeEventListener('mousemove', move);
                document.removeEventListener('mouseup', end);
                // do other things
            }
        }
    }

    // 私有方法，仅仅用来获取transform的兼容写法
    function getTransform() {
        var transform = '',
            divStyle = document.createElement('div').style,
            transformArr = ['transform', 'webkitTransform', 'MozTransf

            i = 0,
            len = transformArr.length;

        for(; i < len; i++)  {
            if(transformArr[i] in divStyle) {
                return transform = transformArr[i];
            }
        }

        return transform;
    }

    // 一种对外暴露的方式
    window.Drag = Drag;
})();

// 使用：声明2个拖拽实例
new Drag('target');
```

```
new Drag('target2');
```

这样一个拖拽对象就封装完毕了。

建议大家根据我提供的思维方式，多多尝试封装一些组件。比如封装一个弹窗，封装一个循环轮播等。练得多了，面向对象就不再是问题了。这种思维方式，在未来任何时候都是能够用到的。

下一章分析jQuery对象的实现，与如何将我们这里封装的拖拽对象扩展为jQuery插件。