事件（event）

什么是事件

我们可以简单的把事件理解为是浏览器感知系统（神经系统）。比如：它可以感觉到用户是否用鼠标点击了页面(click)、鼠标是否进入了网页上的某个元素上面(mouseover或mouseenter)、鼠标是否离开了网页(mouseout或mouseleave)、浏览器是否加载完了网页上的资源(window.onload)、文档树是否生成（DOMContentLoaded）、键盘上的某个键是否按下（keydown）、鼠标的滚轮是否滚动了等等！

其实事件的原理并非是浏览器的感觉系统，它的本质是一个行为发生时，对另外一个行为的回调。在讲事件原理和设计模式时我们会把这事讲清楚。

事件的实现（事件绑定）

事件绑定就是：当这个事件发生的时候，运行一个或多个方法（function），比如说当鼠标点击网页的时候，就弹出一个“事件”，则写成:

document.onclick=function(){alert(“事件”)}

当然我们在第一天的教材里也说过，事件绑定相当于“作计划”，绑定在事件上的方法执行了，相当于是“计划的事发生了”。所以一般情况下，事件属性前面都有“on”，如：ele.onclick,ele.onmousedown,ele.onmouseup，这里的on，其实相当于：当什么什么的时候。“作计划”早于“计划的事发生”。

当然我们也可以不给事件绑定处理方法，也就是说当此事件发生的时候，什么也不需要做。事件常有，而事件上绑定的方法不一定要有。

我们给网页元素的某个事件绑定处理方法时，经常还会有一个形式参数e，但我们运行的时候，却没办法传实参给这个e，比如：

function fn(e){

//标准浏览器中：定义了一个形参e，但当事件触发的时候，并没有给e赋实际的值，则浏览器会把“事件”的对象赋给这个e.这会这个e是个系统级的对象：事件

//IE中的事件对象是个全局的属性window.event;

//事件是浏览器感知系统（神经系统）

e=e||window.event;//事件对象，解决一下浏览器兼容性问题

//以下是常用的一些事件对象的属性

//var x=e.clientX;//所有浏览器都支持，相对于浏览器的鼠标坐标

//var y=e.clientY;

var x=e.pageX;//IE8或以下浏览器不支持

var y=e.pageY;//相对于文档的鼠标坐标

var target=e.target||e.srcElement;//事件源的概念：事件最终发生在那个网页元素上

//事件源和事件传播是息息相关的

//事件传播包括：冒泡和捕获。事件传播是浏览器在处理事件行为的机制。冒泡阶段或捕获阶段。

//IE6，7，8里，事件传播只有冒泡阶段

div1.innerHTML="当前的事件类型："+e.type+"鼠标的X坐标："+x+",鼠标的Y坐标："+y+",事件发生在那儿："+e.target||e.srcElement;

}

document.onmousemove=fn;//把fn直接赋给document的onmousemove这个属性

**示例2：和键盘相关的事件和属性**

var ele=document.getElementById('div1');

document.onkeydown=function(e){

e=e||window.event;//处理事件兼容性

//alert(e.keyCode);//keyCode是当前的这个键值对应的ASCII码

switch(e.keyCode){

case 37://键盘上的左键

ele.style.left=ele.offsetLeft-5+'px';

break;

case 38://上

ele.style.top=ele.offsetTop-5+'px';

break;

case 39://右

ele.style.left=ele.offsetLeft+5+'px';

break;

case 40://下

ele.style.top=ele.offsetTop+5+'px';

break;

}

}

以上这样的事件绑定方法（就是直接把fn赋给document.onmousemove的方式）叫DOM 0级事件绑定，它是相对于DOM2级事件绑定来说（下面会讲）

DOM元素的默认行为

很多的网页元素都会有默认的行为，比如说当你点击一下超链接a标签的时候，它会有一个跳转的行为；当你在网页上点鼠标右键时会出现一个右键菜；当你在一个form表单里点击提交按钮时网页会产生提交行为并刷新网页，当你网页上滚动鼠标滚轮时，网页的滚动条会动等等。这些都叫事件的默认行为，如果想把这默认行为取消了，相应的JS代码如下：

1. onclick=function(){return false;}//在方法里加个return false，就阻止超链接点击时的跳转行为了

document.oncontextmenu=function(){

/\* 在这里你还可以加一些代码，实现自定义的右键菜单 \*/

return false;//系统自带的右键菜单就失效了

}

Form.onsubmit=function(){return false;}//这样表单就不会产生提交行为了

document.onmousewheel=function(){return false}//IE和chrome的方式，取消鼠标的滚轮的默认行为，网页的滚动条就不会动了

document.addEventListener('DOMMouseScroll',function(e){e.preventDefault=true});//功能同上，火狐的方式。火狐只能用DOM二级的绑定方式，并且用e.preventDefault=true

我们要知道常见的事件默认行为有那些，并且要知道阻止默认行为，只要在绑定到这个行为事件的方法里最后加一句：return false;就可以了。

**但要强调注意的是：**如果你的事件绑定是用addEventListener来实现的，那阻止默认行为必须用e.preventDefault=true。

事件传播和阻止事件传播

当事件发生在子元素中的时候，往往会起连锁反应，就是在它的祖先元素上也会发生这个事件，比如说你点击了div一个，也相当于点击了一个body,同样相当于点击了html，同样相当于点击了document

在理解事件传播的时候要注意两点：

一是事件本身在传播，而不是绑定在事件上的方法会传播

二是并非所有的事件都会传播，像onfocus，onblur事件就不传播，onmouseenter和onmuseleave事件也不会传播。

事件委托

事件委托是利用事件传播的机制，通过判断事件源来实现的，是一种高性能的事件处理方式。对事件委托的好处和概念，请详见《高程3》第402页。

我们这里通过一个简单的示例来看事件的好处。

需求：有如下的HTML代码，当点击这个网页中的每一个元素时，弹出这个元素对应的标签名。

<body>

<div id="outer" >

outer

<div id="inner" >

inner,inner

<p >ppppppp

<span >

span

<a href="###" >珠峰培训</a>

</span>

</p>

</div>

</div>

</body>

一般的思路是把所有的元素都取到，然后循环绑定。这样做的缺点不仅是性能不好，并且还要处理事件传播的问题，不优化的代码如下：

var eles=document.getElementsByTagName("\*");

for(var i=0;i<eles.length;i++){

eles.item(i).onclick=function(e){

alert(this.tagName);

e.stopPropagation();//加上阻止事件传播是可以的，但性能不是最优的

return false;//防止超链接的默认行为

}

}

以下用事件委托实现

事件委托:事件委托就是用的事件传播的机制，无论那一个网页元素，它的click事件都会最终传到document上。这样，则只需在document上处理click事件既可

document.onclick=function(e){

e=e||window.event;

**var target=e.target||e.srcElement;//获得事件源是关键**

alert(target.nodeName);

return false;

}

**事件委托的关键是理解好事件源的概念。**

DOM二级事件

我们原来学过DOM，DOM是解决文档里元素关系的一套模式，其实那只是DOM的第一个版本解决的问题。在DOM的第二个版本里，解决的问题就不仅仅是文档里元素之间的关系里，还把DOM元素的事件问题重新给了一套解决方案，这套方案就叫“DOM二级事件”。

DOM二级事件解决了原来的同一事件绑定多个处理方法时，后面的绑定会覆盖前面的绑定的问题，如：

ele.onclick=fn1

ele.onclick=fn2;

这样处理的结果就是，ele的onclick事件上，fn2方法把fn1方法给覆盖了，这样不容易实现在同一事件上绑定多个方法。

W3C给出的方法是这样的：

ele.addEventListener(“click”,fn,false);

IE6/7/8给出的方案是：

ele.attachEvent(“onclick”,fn);

**解决标准浏览器和低版本的IE方案如下：事件绑定**

function bind(ele,type,handler){

if(ele.addEventListener){//标准浏览器专用

ele.addEventListener(type,handler,false)

}else if(ele.attachEvent){//IE专用

ele.attachEvent("on"+type,handler);

}

}

**移除事件绑定：**

function unbind(ele,type,handler){

if(ele.removeEventListener){//标准流程器专用

ele.removeEventListener(type,handler,false)

}else if(ele.detachEvent){//IE专用

ele.detachEvent("on"+type,handler);

}

}

**注意：**虽然IE、火狐、chrome等浏览器都给出了DOM2级事件的处理方法，但IE的方法却存在着很多问题，并且非常严重。一是被绑定的方法在事件触发执行时，this关键字竟然是window，二是IE中被绑定到事件上的方法的执行顺序是混乱的。

在W3C的标准是在同一事件上，先绑定的方法先执行，并且不能重复绑定同一个方法在同一个事件上。但IE6/7/8中，如果绑定的方法少于9个，执行的顺序是相反的，超过9个，执行是混乱的。

这些IE中的这些问题都比较严重，我们必须解决好。

事件的兼容性问题总结（常见的）

• 事件对象本身：标准浏览器是事件发生时自动给方法传一个实参，这个实参就是事件对象。IE是全局的window.event

• 阻止事件传播：e.stopPropagation这个方法，IE是e.cancelBubble=true这个属性

• 阻止默认行为：e.preventDefault()方法，IE是e.returnValue=false;

• 事件源：e.target.IE是e.srcElement

• e.pageX,e.pageY，IE不支持这两个属性

• DOM二级的事件绑定：ele.addEventListener,IE是ele.attachEvent

• IE的attachEvent绑定的方法上：1、this不是当前的元素，2执行顺序是混乱的

DOM二级事件事件兼容性问题解决之一：解决this关键字

**注意：**以下代码中的handler是个形参，它可以表示不同的方法，所以不要把handler认为是某一个具体的方法。如果你bind(ele,”click”,fn1)，则fn1是handler，如果bind(ele,”click”,fn2)，则fn2是handler。新手一定要理解好把握好。

关键思路：关键是两个问题，一是理解好call的用途作用，二是理解好在一个程序里写的代码，是为了解决另外一个程序中的问题的这种思路。

我们修改一下bind这个方法

如果只是解决被绑定的方法的this指向，这倒好办，只需在IE的专用的代码里，帮如下修改即可

else if(ele.detachEvent){//IE专用

function fnTemp=function(){handler.call(ele)};//使用call方法，强制使handler方法在运行时this指向被绑定的ele这个DOM元素

ele.detachEvent("on"+type,fnTemp);//再绑定时，就不是直接绑定handler这个方法了，而是绑定经过“化装”的fnTemp这个方法

}

或直接这样写：

else if(ele.detachEvent){//IE专用

ele.detachEvent("on"+type,function(){handler.call(ele)});

}

这样确实在事件触发时，handler运行，并且让handler的this指向了被绑定的元素ele，但由于我们已经不是直接绑定的handler方法了，而是经过call“变形”后的fnTemp方法，那在移除绑定的时候，我们就没办法移除handler方法。

那我们怎么能找到这个化装之后fnTemp，并将其移除呢？这件事必须要在绑定事件的时候就要考虑好：在bind方法里，把fnTemp方法**保存下来**，并且**还要能用某种方式识别出**这个fnTemp方法是由那一个handler“变形”而来的。

这儿是个不好理解的难点，做这件事，我们需要两步：

【这儿两次都用到了对象的自定义属性，不好理解，望认真思考】

1. 把fnTemp保存下来，不能用全局变量保存，因为容易被污染；保存在bind的某个变量里，局部变量在unbind这个作用域里也访问不到。在不同的作用域里，还能访问到一个非全局变量的值，那用什么呢？最方便的方式就是把fnTemp**保存在ele这个DOM元素的属性上**，因为这个ele是两个函数都要操作的引用类型的变量，那么我们在ele上定义一个自定义属性，那这个属性在bind和unbind两个作用域里都能访问的到。具体实现方式如下：

}else if(ele.attachEvent){//IE专用代码

if(!ele["aBind"+type]){//如果不存在这个属性，则创建一个

ele["aBind"+type]=[];//使用数组来保存被绑定到不同事件上的那些方法（相当的事件上，可能会被绑定很多个handler）

//这个属性是以aBind为前缀，以type为区分符的。Type是事件类型，这是一个非常重要的技巧，

}

var tempFn=function (){handler.call(ele);}//变形这个方法，让这个方法运行的时候this指向被绑定的元素

ele["aBind"+type].push(tempFn);//把变形后的tempFn保存到这个数组里

ele.attachEvent("on"+type,tempFn);

｝

**补充：**理解好ele["aBind"+type]=[]这个属性定义时，”aBind”这个字符串是个区别符的意思。先来看如果没有这个”aBind”会怎么样？那就定义成了ele[type]=[]，如果type是”click”，则会出现ele.click=[]，而ele本身就有click这个方法属性，我们是没办法修改这个原生属性的，这样定义就失效了。**所以才给”click”前面加个”aBind”做为前缀**，以避免或减少和原生的属性冲突。这种加前缀的技巧还是很常见的。

1. 给tempFn再加一个自定义属性，用来标识当前这个tempFn是由handler“变形”而来的

var tempFn=function (){handler.call(ele);}//变形这个方法，让这个方法运行的时候this指向被绑定的元素

tempFn.photo=handler;//我们通过tempFn的photo这个属性，就可以分辨出tempFn这个方法是由那个handler变形而来的。Photo这个属性只是在这儿定义，而使用它是在unbind函数里。

1. 还要强调一个DOM2级事件绑定编程的原则，即：**一个函数不能被重复绑定在同一个事件上**。比如：不能把fn1这个函数重复绑定给ele的click事件：

ele.addEventListener(“click”,fn1,false);

ele.addEventListener(“click”,fn1,false);//绑定两次或多次，但是事件触发时，后面的绑定是无效的。

**当然，低版的IE浏览器没有遵循这个原则，**所以这儿还要解决一下这个问题，加一个判断即可（在ele）

for(var i=0;i<ele["aBind"+type].length;i++){

if(ele["aBind"+type][i].photo==handler){//如果数组里已经存在了经过化装的handler方法，则退出执行

return;//保证一个方法只能被绑定到某事件上一次

}

}

**Bind方法完整的代码如下：**

**function bind(ele,type,handler)**{

if(ele.addEventListener){

ele.addEventListener(type,handler,false)

}else if(ele.attachEvent){

if(!ele["aBind"+type]){//如果不存在这个属性，则创建一个

ele["aBind"+type]=[];//这个属性是以aBind为前缀，以type为区分符的

}

var tempFn=function (){handler.call(ele);}//变形这个方法，让这个方法运行的时候this指向被绑定的元素

tempFn.photo=handler//photo标识型的属性,用做和handler关联

for(var i=0;i<ele["aBind"+type].length;i++){

if(ele["aBind"+type][i].photo==handler){

return;//保证一个方法只能被绑定到某事件上一次

}

}

ele["aBind"+type].push(tempFn);

ele.attachEvent("on"+type,tempFn);

};

}

我们再来看unbind。该做的准备工作，都已经在bind里完成了。Unbind负责把绑定在事件上的方法移除，但现在已经不是移除handler这个方法了，而是移除经过化装后的这个方法，这个方法被保存在ele的"aBind"+type这个属性上。

我们知道ele["aBind"+type]这是个数组，先把它取到，赋给一个短变量a：var a=ele["aBind"+type]。操作a这个短变量比操作ele["aBind"+type]这个属性更方便。

然后遍历这个数组，逐个比较那个是经过化装的handler方法，当然比较的依据是photo这个属性，所以是：

**1.**for(var i=0;i<a.length;i++){//遍历

**2.** if(a[i].photo==handler){//通过photo属性做比较。在bind里定义的photo属性

**3.** ele.detachEvent("on"+type,a[i]);//把这个方法从事件移除

**4.** a.splice(i,1);//并且一定要把这个方法从数组里移除了，要不然下一次就不能再绑定了。

**5.** return ;//因为每一次绑定都是唯一的（这是原则，上边讲了），所以移除后直接结束这个函数的运行就可以了

}

}

解释第四行：用a.splice会造成“数组塌陷”，所以在正式的代码里，用a[i]=null来解决的。看下面代码的第10行。大家要注意上课时看老师的BUG测试演示。

**完整的unbind代码如下：**

1. **function unbind(ele,type,handler)**{
2. if(ele.removeEventListener){//标准浏览器
3. ele.removeEventListener(type,handler,false);
4. }else if(ele.detachEvent){//IE浏览器
5. var a=ele["aBind"+type];//在bind里约定好的那个自定义
6. if(a){//如果存在这个数组
7. for(var i=0;i<a.length;i++){
8. if(a[i].photo==handler){//这里用了在bind里定义的photo属性
9. ele.detachEvent("on"+type,a[i]);
10. **a[i]=null;//改进之后的，避免动态移除的时候会出问题**
11. return ;
12. }
13. }
14. }
15. }
16. }

DOM二级事件事件兼容性问题解决之二：执行顺序问题

首先要明确的是在IE6/7/8中，如果一个事件上绑定多个方法，它的执行顺序是相反的或混乱的，这也是IE中另一个很严重的问题。

解决这个问题的原则，其实相当于把浏览器的事件机制给重写了一遍。

**就是是把IE浏览器自己的那套事件机制抛弃了，然后自己构造一个可以按顺序执行的程序池（或叫程序清单也行），然后再编写一个run方法，让某个事件发生的时候，让run方法依次去遍历执行程序池里的那些方法。**

比如，我们要给ele的click事件绑定fn1,fn2,fn3,fn4,fn5等若干个方法，我们不直接写成bind(ele,”click”,fn1)这样的方式，而是先创建一个数组，这个数组保存在ele的一个自定义属性上，**以”aEvent”这个字符串为前缀，以事件类型为区分符**（这个技巧在解决this指向的时候用过）。每绑定一个方法，我们就push进去一个。

伪代码如下：

var type=”click”

ele[”aEvent”+type]=[];

ele[”aEvent”+type].push(fn1);

ele[”aEvent”+type].push(fn2);

ele[”aEvent”+type].push(fn3);

ele[”aEvent”+type].push(fn4);

ele[”aEvent”+type].push(fn5);

那当事件发生的时候怎么去执行保存在这个ele[”aEvent”+type]的方法呢？

先定义一run方法，由这个run方法来负责去数组里遍历执行保存好的那些方法。

1. function run(){
2. var a=this["aEvent"+event.type];//这里的this是ele，因为run函数被绑定给了ele。
3. if(a){//为了确定代码不出意外，先判断一下数组是否存在
4. for(var i=0;i<a.length;i++){//遍历
5. a[i].call(this, event);**//用call方法去调用执行，确保了数组里的方法的this是指向被绑定元素的。这是个不好理解的点。event是事件对象**
6. }
7. }
8. }

然后真正绑定事件的时候，用上面写好的bind函数绑定这个run函数：

**bind(ele,”click”,run);**

这样，当click事件发生的时候，真正由事件触发而执行的函数其实是run（这里用的是浏览器的事件机制），而run再去数组里遍历执行保存下来的那些函数（这里就避开了浏览器的执行机制）。因为数组的循环执行是有序的，所以这个保存在数组里的方法肯定也是按顺序执行的。

只要理解了这儿，总体的思路就没有问题了，下面就是把代码写的更有通用性。

相对完整的代码如下：

先要写一个on方法，负责创建程序池，并且把需要绑定在事件上的方法保存在这个池子里：

**function on(ele,type,handler){//订阅:产生一张清单**

if(!ele["aevent"+type]){

ele["aevent"+type]=[];

}

var a=ele["aevent"+type];

for(var i=0;i<a.length;i++){

if(a[i]==handler)return;//避免重复绑定

}

a.push(handler);

**bind(ele,type,run);//真正的绑定，虽然on方法会多次运行，但不会多次被绑定这个run方法,这是在bind函数里约定好的原则**

}

**function run(e){**//这个负责通知:遍历那张清单并且执行

e=e||window.event;

var a=this["aevent"+e.type];//在on里产生的那个清单

//下面是解决兼容性问题的代码，这是个整体的事件解决方案

if(!e.stopPropagation){

e.stopPropagation=function(){e.cancelBubble=true;}

}

if(!e.preventDefault){

e.preventDefault=function(){e.returnValue=false}

}

if(typeof e.pageX=="undefined"){

e.pageX=(document.documentElement.scrollLeft||document.body.scrollLeft)+e.clientX;

e.pageY=(document.documentElement.scrollTop||document.body.scrollTop)+e.clientY;

}

if(typeof e.target =="undefined"){

e.target=e.srcElement;

}

if(a){

for(var i=0;i<a.length;i++){//遍历

a[i].call(this, e);//e是事件对象

}

}

}

拖拽的最基本原始代码

拖拽的基本原理：

拖拽是三种事件的组合

当鼠标按下时准备拖拽，当鼠标按着并且鼠标移动时进行拖拽，当鼠标松开时，结束拖拽。盒子是随着鼠标移动而移动的，鼠标移动多远它就移动多远。

鼠标移动的距离是=鼠标现在的位置-鼠标按下去的时的位置。

盒子移动的公式是：盒子现在的位置=盒子原来的位置（mousedown时的）+鼠标移动的距离

在学习这个案例的时候，一个是要把握好对this的理解和应用，另一个就是把握好对自定义属性的应用。

以下的代码，每个函数都是低耦合的，每一个函数里都不会出现具体的某一个被拖拽元素的变量名，而是灵活使用this关键字来获得被拖拽的元素。this的好处在这儿能很明显的体显出来。

把获得到的值保存在当前被拖拽的元素的自定义属性上，避免了全局变量容易被污染的问题。

var ele=document.getElementById('div1');

ele.onmousedown=down;

function **down**(e){//准备拖拽

this.x=this.offsetLeft;//先要把盒子的原始位置保存下来，保存到自定义属性x上

this.y=this.offsetTop;//把盒子的垂直坐标保存在this.y上

this.mx=e.clientX;

this.my=e.clientY;

this.onmousemove=move;

this.onmouseup=up;

}

function move(e){

this.style.left=e.clientX-this.mx+this.x+"px";//当鼠标移动时，重新计算盒子的坐标

this.style.top=e.clientY-this.my+this.y+"px";

}

function up(){//当鼠标按键抬起的时候，结束拖拽

this.onmousemove=null;//把事件属性清空

this.onmouseup=null;

}

经过优化和改进之后的拖拽代码

ele.onmousedown=down;

/\*

因为在下面的拖拽里经常需要让事件绑定的方法在运行的时候，this关键字指向被拖拽的元素，所以把这段经常被重复利用的逻辑封装成一个小方法。

目的就是让fn方法在运行的时候，功能不变，但this指向指定的对象obj。

\*/

**function processThis(obj,fn){//这个函数一定要理解好**

**return function(e){fn.call(obj,e);} //返回这个方法。事件在绑定的时候，其实是绑定的这个方法，而运行的时候，其实还是运行fn这个方法**

**}**

function down(e){

e=e||window.event;

this.x=this.offsetLeft;

this.y=this.offsetTop;

this.mx=e.clientX;

this.my=e.clientY;

if(this.setCapture){//

**this.setCapture();//专门处理mousemove事件的方法，使DOM元素不能丢失鼠标。目前IE和FireFox支持此方法**

this.onmousemove=move;

this.onmouseup=up;

}else{//chrome浏览器,把给document绑定mousemove事件.只有文档这个大容器不会丢掉鼠标。

//var that=this;//这是开始用的技巧，注意体会对this的处理

//this.MOVE=function(e){move.call(that,e)}

//this.UP=function(e){up.call(that,e)}

//document.onmousemove=this.MOVE;

//document.onmouseup=this.UP;

**document.onmousemove=processThis(this,move);//用了上边定义的这个方法**

**document.onmouseup=processThis(this,up)**

}

}

function move(e){

e=e||window.event;

this.style.left=e.clientX-this.mx+this.x+"px";

this.style.top=e.clientY-this.my+this.y+"px";

}

function up(){

if(this.releaseCapture){

this.onmousemove=null;

this.onmouseup=null;

this.releaseCapture();

}else{

document.onmousemove=null;

document.onmouseup=null;

}

}

附录：写在HTML标签里的事件是如何被浏览器解析并执行的

比如HTML代码如下：

<div id="div1" onclick=”fn()” onClick="fn.call(this,event)" onClick="fn(event)" >

这里面写的三种格式，是为了对比

</div>

写在HTML代码里的事件绑定，相当于把事件后边的字符串当参数，传给 new Function("fn()"),相当于生成了这样一个方法：function(){fn();}

div1.onclick=function(){fn();}

FireFox里生成的匿名方法是这样的

div1.onclick=function(event){fn(event);}

这个function(event){}匿名方法才是直接被绑定到click事件上的方法，它形式参数名叫event。而fn是在匿名方法里被调用(invoke)，如果fn需要事件对象，则必须传实参：就是event.

只有一个方法，直接被绑定给事件才不需要传实参

实在写不动了，下周继续吧。

2014年10月11日晚23点 耿银鹏

给拖拽加效果并且发现代码在设计思想上的问题

回顾上一章的代码

上一章里解决事件问题的兼容性，写了两个方法，用bind解决this关键字的问题，on来解决执行顺序的问题，并且随带在run方法里把IE中的更多的兼容性问题也给解决了。

尤其要理解好on方法和run方法的作用，这是个难点。这两个方法其实是把浏览器的事件机制又给重新写了一次。

我们把每次事件绑定看成是一次“约会”，既然是“约会”，那肯定要提前“约”，到了某个条件成熟的时候（比如说鼠标点了一下的时候），再发生这个“会”on方法负责“约”，就是把响应某个事件的那些方法保存下来，记到一下数组里。当这个事件发生的时候，再由run方法去统一、按顺序调用那保存在数组里的那些方法。

上一章里bind，on，run这三个方法的关系是这样的：

1. on负责把要绑定在事件上的方法保存在一个数组里.
2. run负责方法当事件发生时按顺序调用保存在数组里的方法，这样就解决了IE中绑定在事件上的方法执行顺序混乱的问题。并且还随便把其它IE的兼容性问题给解决了。
3. bind方法其实是解决this指向问题的（当然也解决了重复绑定的问题）。如果不考虑this的问题，可以把run方法直接用attachEvent来绑定：ele.attachEvent(“on”+type,run);

其实这个bind方法是可有可无的，因为解决this关键和执行顺序的问题，在on方法里也可以搞定。并且解决兼容性问题主要是解决IE的问题，其它标准浏览器也不需要这样做，那么我们继续优化上一章里的代码。

继续优化上一章节里讲的事件库

着重要理解好：**on方法里做的这些处理，都是为run方法准备的。**

1. function on(ele,type,handler){
2. if(ele.addEventListener){//如果是标准浏览器，就直接用addEventListener全搞定了，没有那么多的麻烦事
3. ele.addEventListener(type,handler,false);
4. }else if(ele.attachEvent){//如果是IE6/7/8麻烦就大了
5. if(!ele["IEevent"+type]){//判断这个数组是否存在
6. ele["IEevent"+type]=[];//这个数组是用来保存绑定在事件上的那些方法的。这里的属性前缀叫"IEevent"。这样的技巧上一章里讲过。
7. }
8. var a=ele["IEevent"+type];
9. for(var i=0;i<a.length;i++){
10. if(a[i]==handler){//避免数组里有重复的方法，即避免被一个方法被重复绑定
11. return;
12. }
13. }
14. a.push(handler);//如果没有重复绑定过，就把这个函数保存在数组里
16. if(!ele["attachEvent"+type]){//如果ele["attachEvent"+type]这个属性不存在，说明是在这个事件类型上第一次绑定方法（下面的run也只能绑定这么一次）
17. ele["attachEvent"+type]=processThis(ele,run);
18. //真正绑定的是run(改造之后的)，只能在同一事件上绑定一次
19. //关于processThis方法，上一章已经介绍过了。
20. ele.attachEvent("on"+type,ele["attachEvent"+type]);
21. **//这里是把run方法绑定给ele的type事件，并且让run的this关键字指向ele，并且run方法不能被重复绑定。从16行到20行代码，是上一章里bind方法要做的事。**
22. }
23. }
24. }

**一定要把on函数和run函数综合在一起考虑。Run方法只会在IE浏览器中才运行。**

下面是run方法，当事件发生事，其实真正运行的是run方法。Run再去相对应的数组里按顺序执行保存下来的那些方法。

1. function **run()**{//这次的run只解决IE的问题，所以就不需要给形参e了
2. var e=window.event;//只操作IE的事件对象window.event即可
3. var type=e.type;//取出当前事件的类型
4. var a=this["IEevent"+type];//根据事件类型去找到对应的数组
5. If(!a)return;//如果不存在这个数组，说明这个事件上没有绑定方法，退出
6. **//下面是开始解决IE的一系列兼容性问题**
7. e.target=e.srcElement;
8. e.preventDefault=function(){
9. e.returnValue=false;
10. }
11. e.stopPropagation=function(){e.cancelBubble=true}
12. e.pageX=(document.documentElement.scrollLeft||document.body.scrollLeft)+e.clientX;
13. e.pageY=(document.documentElement.scrollTop||document.body.scrollTop)+e.clientY;
14. for(var i=0;i<a.length;){
15. if(a[i]===null){//这里对应off方法的移除绑定时的赋值为null的动作。
16. a.splice(i,1);//如果已经移除里，则将这一项删除
17. }else{
18. a[i].call(this,e);//如果没有移除绑定，则执行。**call的作用在于：**1、**执行时让此函数的this指向run里的this，即指向当前被绑定的元素（因为run被绑定在了这个DOM元素上，并且在on方法里已经做过相关处理。见上一小节on方法的第17行代码）。2、并且按着标准浏览器的原则，把事件对象当成第一个实参传给这个方法。这些原则要认真理解。**
19. i++;
20. }
21. }
22. }

Off方法负责移除绑定，在IE中的原则是把原来保存到数组里的方法清空

代码如下：

1. function off(ele,type,handler){
2. if(ele.removeEventListener){
3. ele.removeEventListener(type,handler,false)
4. }else if(ele.detachEvent){
5. var a=ele["IEevent"+type];
6. if(a){
7. for(var i=0;i<a.length;i++){
8. if(a[i]==handler){
9. a[i]=null;//不是用splice方法删除，而是赋值为null，是为了避免“数组塌陷”造成的BUG. 如果有过多的函数被off，会造成数组里出现很多的空项，造成性能浪费，解决这个问题，在run方法的第15行给出了方案。
10. }
11. }
12. }
13. }
14. }

用完整的事件库重新实现通用的拖拽代码

1. var oDiv1=document.getElementById('div1');
2. on(oDiv1,"mousedown",down);
3. function down(e){//准备拖拽
4. this.x=this.offsetLeft;
5. this.y=this.offsetTop;
6. this.mx=e.pageX;
7. this.my=e.pageY;
9. if(this.setCapture){
10. this.setCapture();
11. on(this,"mousemove",move);
12. on(this,"mouseup",up)
13. }else{
14. this.MOVE=processThis(this,move);
15. this.UP=processThis(this,up);
16. on(document,"mousemove",this.MOVE);
17. on(document,"mouseup",this.UP);
19. }
20. e.preventDefault();
21. }
22. function move(e){//进行拖拽
23. this.style.left=e.pageX-this.mx+this.x+"px";
24. this.style.top=e.pageY-this.my+this.y+"px";
25. }
26. function up(e){//结束拖拽
27. if(this.releaseCapture){
28. this.releaseCapture();
29. off(this,"mousemove",move) ;
30. off(this,"mouseup",up);
31. }else{
32. off(document,"mousemove",this.MOVE);
33. off(document,"mouseup",this.UP);
34. }
35. }

给拖拽增加更多动画效果

我们以上边的事件库和拖拽的基本代码为基础，不断的丰富不同的效果。

对不同的动画效果对应的函数和难点的说明：

**主要并且通用的技巧还是this和自定义属性的使用，这个一定要体会好**

1. fly函数是实现的抛出去的动画效果。难点是如何计算出抛出去的速度。是在拖拽的move方法里加了以下代码计算出来的：
   1. if(this.previous===undefined){//判断是否存在上一次速度
   2. this.previous=e.pageX;
   3. this.prevD=new Date;//测试用代码，用来保存第一次
   4. }else{
   5. this.speed=e.pageX-this.previous;//速度=这次的鼠标坐标-上次的鼠标坐标
   6. this.previous=e.pageX;//让当次的鼠标坐标成为“下一次”的鼠标坐标。“当次”和“下一次”只是个相对概念。因为move方法是不断的执行的，“这次”的鼠标坐标，就是“下次”move方法里的“上一次”鼠标坐标
   7. this.currentD=new Date;
   8. console.log(this.currentD-this.prevD);//测试用代码，用来计算两次mousemove事件之间的时间间隔
   9. this.prevD=this.currentD;//测试用代码，用来得到本次
   10. }

虽然以上代码并没有写在fly里，但这个this.speed是fly方法里必须用到的。

**这两个动画共同的难点这里的难点有：一、定时器里的this处理方法；二、过界判断；三、如何判断动画应该停止的条件。尤其是drop方法里的全局变量flag的使用技巧。**

fly方法的代码如下：

1. function fly(){
2. clearTimeout(this.flyTimer);//清除动画积累。写动画的规定动作
3. var w=(document.documentElement.clientWidth||document.body.clientWidth)-this.offsetWidth;//浏览器的宽-盒子本身的宽：是盒子最右边的边界
4. if(this.offsetLeft+this.speed>=w){//右过界判断
5. this.style.left=w+"px";//最大到右边界
6. this.speed\*=-1;//一但撞到边界上，则让其往相反的方向运动
7. }else if(this.offsetLeft+this.speed<=0){//左过界判断
8. this.style.left=0;//最左不能越过浏览器的左边
9. this.speed\*=-1;
10. }else{//正常运动
11. this.style.left=this.offsetLeft+this.speed+"px";
12. }
13. this.speed\*=.93;//让速度不断衰减，模拟空气摩擦
14. if(Math.abs(this.speed)>=0.5){//当速度太小时则让动画停止
15. //由于用定时器来执行的函数this会是window，所以这里要用processThis来处理一下。关键技巧之一。
16. this.flyTimer=window.setTimeout(processThis(this,fly),30);
17. }
18. }
19. drop方法实现了自由落体反弹的效果。就像一个皮球落在了地面上还会弹起来，因为空气的阻力（空气的摩擦力），最后弹不起来的效果。

代码如下：

1. **var flag=0;**//全局变量，drop函数里用来记录连续撞击地面的次数的。这是个重要的技巧。
2. function drop(){
3. window.clearTimeout(this.dropTimer);
4. var h=(document.documentElement.clientHeight||document.body.clientHeight)-this.offsetHeight;
5. if(!this.dropSpeed){
6. this.dropSpeed=9;//初始化下落的速度
7. }else{
8. this.dropSpeed+=9;//让速度每次累加。9是任意给的。
9. }
10. if(this.offsetTop+this.dropSpeed>=h){
11. this.style.top=h+"px"
12. this.dropSpeed\*=-1;//一但到达了边界，则往反方向运动
13. flag++;//每到达一次边界，则让flag累加一次。
14. }else{
15. this.style.top=this.offsetTop+this.dropSpeed+"px";
16. flag=0;//但是：正常情况下又让其归0
17. }
18. this.dropSpeed\*=.93;
19. if(flag<3)//什么情况flag会是2或3，甚至大于3呢？
20. this.dropTimer=window.setTimeout(processThis(this,arguments.callee),40);
22. }

这个效果是在模拟把皮球扔出去，皮球沿着扔的方向飞走并下落，落到地面再反弹的效果。因此，这两个函数应该在拖拽结束的时候运行，所以应该把这两个函数运行在up方法里。在up方法的最后加两句代码：

fly.call(this);//让fly里的this也是当前被拖拽的元素

drop.call(this);

代码模块化重构思想

**写出效果来并非是程序员的终极目标，即便是能很熟练的写出这些效果来，也没什么了不起的，最多也就是个三流的程序员。**

**那问题出在那儿呢？就是每加一个效果，我就都要去修改原代码，把本应该独立的功能，都搅合在了一起，这就违背了编程“低耦合高内聚”的原则。**

**那我们应该怎么办呢？就是：拖拽功能应该是完全独立的，它应该是一个基本的，成熟完善的独立模块。而像fly方法和drop方法，也应该是完全独立的功能模块，而不能把这三个模块写在一起。当然也不能为了给拖拽加另外的效果而去破坏原来的源码。**

**既要让这三个效果在一起发生，又不能把代码写在一起，这不是不讲理难为人吗？**

**这怎么办呢？上设计模式。一说设计模式就立马高大上了，那先忘掉这个词，先说几个容易理解的吧：**

想让三个函数既是独立的，还要能在必要的时候联系在一起，那就要为这三个函数之间搭一个桥梁，为这个函数和其它函数之间留一个通道，如果用编程术语来解释，就是为每一个功能模块留好接口。那这个接口叫什么呢？在JS中有一种叫法叫“事件”。好，现在我们就来颠覆原先你对事件的认识吧！把原来学过的知识先忘掉，再来看几个故事。

**当你结婚的时候我去喝喜酒**

----你和我是完全独立的个体，那你结婚这个行为和我喝酒这个行为更是独立、没有关联的了。

上帝在创建"人"这个类的时候，每个"人"之间都是独立的个体，人的行为之间更应该（也应该是）没有任何关联的，每一个function之间应该是零耦合的。

那怎么让不同的行为之间，产生关联呢？还拿你结婚来说事。

其实就是我喝酒这个行为，**“约定”**你结婚这个行为。怎么约呢？我告诉你：你结婚的时候我去喝酒，你要记下来（记在一个本子上吧），可能约定你结婚这事的人不止我一个，**那你就要一个个的记下来（编程语言就叫保存在数组里）**，当你结婚这个行为执行的时候，拿出这个本子，一个个的把约定你结婚这件事的人都叫来执行它们的喝酒行为，如果用编程代码来描述一下就是这样(假定你叫刘德华)：

On//

刘德华.on("结婚",张三.喝酒)//约好了当刘德华结婚的时候张三去喝酒

刘德华.on("结婚",李四.喝酒)//李四也约了这事

刘德华.on("结婚",王五.喝酒)//还有王五

……//后面省略n多人等

那么刘德华需要在自己的记事本上开一个明细目录，把这些人记下来

Var a=刘天王的**结婚事件宾朋清单//其实就是定义了一个数组**

1. **push(**王五.喝酒**)//把喝酒方法保存下来**

**这样的记录，也可以叫订阅，监听**

比如这个明细目录名称叫：结婚事件宾朋清单。

（当然刘德华还有其它好多事件的清单呢，比如还有“生娃事件宾朋清单”，“上海演唱会宾朋清单”等等。）

**如果刘德华的结婚行为发生了，那他就拿出记事本，按着“结婚事件宾朋清单”来逐一通知这些人来喝酒**，用编程代码来描述：

发布--发通知

刘天王的行为：看成a

把所有的订阅刘天王的行为的行为称之为b

B订阅了a

B是a的观察者

addEventListener//增加一个观察者（）

订阅发布模式--观察者模式基本相同。

for(var i=0;i<结婚事件宾朋清单.length;i++){

结婚事件宾朋清单[i]();//加括号是指让这个行为发生

}

那如果是生娃这件事，则是这样的代码

for(var i=0;i<生娃事件宾朋清单.length;i++){

生娃事件宾朋清单[i]();//加括号是指让这个行为发生

}

“生娃事件”，“结婚事件”这些是事件的类型

通用的代码是：

for(var i=0;i<event.type宾朋清单.length;i++){

event.type宾朋清单[i]();//加括号是指让这个行为发生

}

我们可以把这些起通知作用的代码叫事件通知器，其实我们是需要把事件通知器封装成一个函数的。

你是不是看着这些代码很眼熟。JS中，这些约定叫事件绑定，就是约定某个行为（function)。所谓的事件绑定是：两个不同行为之间，联系在一起的通道（接口）