**第26章 事件绑定及深入**

学习要点：

1. 传统事件绑定的问题
2. W3C事件处理函数
3. IE事件处理函数
4. 事件对象的其他补充

事件绑定分为两种：一种是传统事件绑定(内联模型，脚本模型)，一种是现代事件绑定(DOM2级模型)。现代事件绑定在传统绑定上提供了更强大更方便的功能。

1. **传统事件绑定的问题**

传统事件绑定有内联模型和脚本模型，内联模型我们不做讨论，基本很少去用。先来看一下脚本模型，脚本模型将一个函数赋值给一个事件处理函数。

var box = document.getElementById('box'); //获取元素

box.onclick = function () { //元素点击触发事件

alert('Lee');

};

问题一：一个事件处理函数触发两次事件

window.onload = function () { //第一组程序项目或第一个JS文件

alert('Lee');

};

window.onload = function () { //第二组程序项目或第二个JS文件

alert('Mr.Lee');

};

当两组程序或两个JS文件同时执行的时候，后面一个会把前面一个完全覆盖掉。导致前面的window.onload完全失效了。

解决覆盖问题，我们可以这样去解决：

window.onload = function () { //第一个要执行的事件，会被覆盖

alert('Lee');

};

if (typeof window.onload == 'function') { //判断之前是否有window.onload

var saved = null; //创建一个保存器

saved = window.onload; //把之前的window.onload保存起来

}

window.onload = function () { //最终一个要执行事件

if (saved) saved(); //执行之前一个事件

alert('Mr.Lee'); //执行本事件的代码

};

问题二：事件切换器

box.onclick = toBlue; //第一次执行boBlue()

function toRed() {

this.className = 'red';

this.onclick = toBlue; //第三次执行toBlue()，然后来回切换

}

function toBlue() {

this.className = 'blue';

this.onclick = toRed; //第二次执行toRed()

}

这个切换器在扩展的时候，会出现一些问题：

1.如果增加一个执行函数，那么会被覆盖

box.onclick = toAlert; //被增加的函数

box.onclick = toBlue; //toAlert被覆盖了

1. 如果解决覆盖问题，就必须包含同时执行，但又出新问题

box.onclick = function () { //包含进去，但可读性降低

toAlert(); //第一次不会被覆盖，但第二次又被覆盖

toBlue.call(this); //还必须把this传递到切换器里

};

综上的三个问题：覆盖问题、可读性问题、this传递问题。我们来创建一个自定义的事件处理函数，来解决以上三个问题。

function addEvent(obj, type, fn) { //取代传统事件处理函数

var saved = null; //保存每次触发的事件处理函数

if (typeof obj['on' + type] == 'function') { //判断是不是事件

saved = obj['on' + type]; //如果有，保存起来

}

obj['on' + type] = function () { //然后执行

if (saved) saved(); //执行上一个

fn.call(this); //执行函数，把this传递过去

};

}

addEvent(window, 'load', function () { //执行到了

alert('Lee');

});

addEvent(window, 'load', function () { //执行到了

alert('Mr.Lee');

});

PS：以上编写的自定义事件处理函数，还有一个问题没有处理，就是两个相同函数名的函数误注册了两次或多次，那么应该把多余的屏蔽掉。那，我们就需要把事件处理函数进行遍历，如果有同样名称的函数名就不添加即可。(这里就不做了)

addEvent(window, 'load', init); //注册第一次

addEvent(window, 'load', init); //注册第二次，应该忽略

function init() {

alert('Lee');

}

用自定义事件函数注册到切换器上查看效果：

addEvent(window, 'load', function () {

var box = document.getElementById('box');

addEvent(box, 'click', toBlue);

});

function toRed() {

this.className = 'red';

addEvent(this, 'click', toBlue);

}

function toBlue() {

this.className = 'blue';

addEvent(this, 'click', toRed);

}

PS：当你单击很多很多次切换后，浏览器直接卡死，或者弹出一个错误：too much recursion(太多的递归)。主要的原因是，每次切换事件的时候，都保存下来，没有把无用的移除，导致越积越多，最后卡死。

function removeEvent(obj, type) {

if (obj['on'] + type) obj['on' + type] = null; //删除事件处理函数

}

以上的删除事件处理函数只不过是一刀切的删除了，这样虽然解决了卡死和太多递归的问题。但其他的事件处理函数也一并被删除了，导致最后得不到自己想要的结果。如果想要只删除指定的函数中的事件处理函数，那就需要遍历，查找。(这里就不做了)

1. **W3C事件处理函数**

“DOM2级事件”定义了两个方法，用于添加事件和删除事件处理程序的操作：addEventListener()和removeEventListener()。所有DOM节点中都包含这两个方法，并且它们都接受3个参数；事件名、函数、冒泡或捕获的布尔值(true表示捕获，false表示冒泡)。

window.addEventListener('load', function () {

alert('Lee');

}, false);

window.addEventListener('load', function () {

alert('Mr.Lee');

}, false);

PS：W3C的现代事件绑定比我们自定义的好处就是：1.不需要自定义了；2.可以屏蔽相同的函数；3.可以设置冒泡和捕获。

window.addEventListener('load', init, false); //第一次执行了

window.addEventListener('load', init, false); //第二次被屏蔽了

function init() {

alert('Lee');

}

事件切换器

window.addEventListener('load', function () {

var box = document.getElementById('box');

box.addEventListener('click', function () { //不会被误删

alert('Lee');

}, false);

box.addEventListener('click', toBlue, false); //引入切换也不会太多递归卡死

}, false);

function toRed() {

this.className = 'red';

this.removeEventListener('click', toRed, false);

this.addEventListener('click', toBlue, false);

}

function toBlue() {

this.className = 'blue';

this.removeEventListener('click', toBlue, false);

this.addEventListener('click', toRed, false);

}

设置冒泡和捕获阶段

之前我们上一章了解了事件冒泡，即从里到外触发。我们也可以通过event对象来阻止某一阶段的冒泡。那么W3C现代事件绑定可以设置冒泡和捕获。

document.addEventListener('click', function () {

alert('document');

}, true); //把布尔值设置成true，则为捕获

box.addEventListener('click', function () {

alert('Lee');

}, true); //把布尔值设置成false，则为冒泡

1. **IE事件处理函数**

IE实现了与DOM中类似的两个方法：attachEvent()和detachEvent()。这两个方法接受相同的参数：事件名称和函数。

在使用这两组函数的时候，先把区别说一下：1.IE不支持捕获，只支持冒泡；2.IE添加事件不能屏蔽重复的函数；3.IE中的this指向的是window而不是DOM对象。4.在传统事件上，IE是无法接受到event对象的，但使用了attchEvent()却可以，但有些区别。

window.attachEvent('onload', function () {

var box = document.getElementById('box');

box.attachEvent('onclick', toBlue);

});

function toRed() {

var that = window.event.srcElement;

that.className = 'red';

that.detachEvent('onclick', toRed);

that.attachEvent('onclick', toBlue);

}

function toBlue() {

var that = window.event.srcElement;

that.className = 'blue';

that.detachEvent('onclick', toBlue);

that.attachEvent('onclick', toRed);

}

PS：IE不支持捕获，无解。IE不能屏蔽，需要单独扩展或者自定义事件处理。IE不能传递this，可以call过去。

window.attachEvent('onload', function () {

var box = document.getElementById('box');

box.attachEvent('onclick', function () {

alert(this === window); //this指向的window

});

});

window.attachEvent('onload', function () {

var box = document.getElementById('box');

box.attachEvent('onclick', function () {

toBlue.call(box); //把this直接call过去

});

});

function toThis() {

alert(this.tagName);

}

在传统绑定上，IE是无法像W3C那样通过传参接受event对象，但如果使用了attachEvent()却可以。

box.onclick = function (evt) {

alert(evt); //undefined

}

box.attachEvent('onclick', function (evt) {

alert(evt); //object

alert(evt.type); //click

});

box.attachEvent('onclick', function (evt) {

alert(evt.srcElement === box); //true

alert(window.event.srcElement === box); //true

});

最后，为了让IE和W3C可以兼容这个事件切换器，我们可以写成如下方式：

function addEvent(obj, type, fn) { //添加事件兼容

if (obj.addEventListener) {

obj.addEventListener(type, fn);

} else if (obj.attachEvent) {

obj.attachEvent('on' + type, fn);

}

}

function removeEvent(obj, type, fn) { //移除事件兼容

if (obj.removeEventListener) {

obj.removeEventListener(type, fn);

} else if (obj.detachEvent) {

obj.detachEvent('on' + type, fn);

}

}

function getTarget(evt) { //得到事件目标

if (evt.target) {

return evt.target;

} else if (window.event.srcElement) {

return window.event.srcElement;

}

}

PS：调用忽略，IE兼容的事件，如果要传递this，改成call即可。

PS：IE中的事件绑定函数attachEvent()和detachEvent()可能在实践中不去使用，有几个原因：1.IE9就将全面支持W3C中的事件绑定函数；2.IE的事件绑定函数无法传递this；3.IE的事件绑定函数不支持捕获；4.同一个函数注册绑定后，没有屏蔽掉；5.有内存泄漏的问题。至于怎么替代，我们将在以后的项目课程中探讨。

1. **事件对象的其他补充**

在W3C提供了一个属性：relatedTarget；这个属性可以在mouseover和mouseout事件中获取从哪里移入和从哪里移出的DOM对象。

box.onmouseover = function (evt) { //鼠标移入box

alert(evt.relatedTarget); //获取移入box最近的那个元素对象

} //span

box.onmouseout = function (evt) { //鼠标移出box

alert(evt.relatedTarget); //获取移出box最近的那个元素对象

} //span

IE提供了两组分别用于移入移出的属性：fromElement和toElement，分别对应mouseover和mouseout。

box.onmouseover = function (evt) { //鼠标移入box

alert(window.event.fromElement.tagName); //获取移入box最近的那个元素对象span

}

box.onmouseout = function (evt) { //鼠标移入box

alert(window.event.toElement.tagName); //获取移入box最近的那个元素对象span

}

PS：fromElement和toElement如果分别对应相反的鼠标事件，没有任何意义。

剩下要做的就是跨浏览器兼容操作：

function getTarget(evt) {

var e = evt || window.event; //得到事件对象

if (e.srcElement) { //如果支持srcElement，表示IE

if (e.type == 'mouseover') { //如果是over

return e.fromElement; //就使用from

} else if (e.type == 'mouseout') { //如果是out

return e.toElement; //就使用to

}

} else if (e.relatedTarget) { //如果支持relatedTarget，表示W3C

return e.relatedTarget;

}

}

有时我们需要阻止事件的默认行为，比如：一个超链接的默认行为就点击然后跳转到指定的页面。那么阻止默认行为就可以屏蔽跳转的这种操作，而实现自定义操作。

取消事件默认行为还有一种不规范的做法，就是返回false。

link.onclick = function () {

alert('Lee');

return false; //直接给个假，就不会跳转了。

};

PS：虽然return false；可以实现这个功能，但有漏洞；第一：必须写到最后，这样导致中间的代码执行后，有可能执行不到return false；第二：return false写到最前那么之后的自定义操作就失效了。所以，最好的方法应该是在最前面就阻止默认行为，并且后面还能执行代码。

link.onclick = function (evt) {

evt.preventDefault(); //W3C，阻止默认行为，放哪里都可以

alert('Lee');

};

link.onclick = function (evt) { //IE，阻止默认行为

window.event.returnValue = false;

alert('Lee');

};

跨浏览器兼容

function preDef(evt) {

var e = evt || window.event;

if (e.preventDefault) {

e.preventDefault();

} else {

e.returnValue = false;

}

}

上下文菜单事件：contextmenu，当我们右击网页的时候，会自动出现windows自带的菜单。那么我们可以使用contextmenu事件来修改我们指定的菜单，但前提是把右击的默认行为取消掉。

addEvent(window, 'load', function () {

var text = document.getElementById('text');

addEvent(text, 'contextmenu', function (evt) {

var e = evt || window.event;

preDef(e);

var menu = document.getElementById('menu');

menu.style.left = e.clientX + 'px';

menu.style.top = e.clientY + 'px';

menu.style.visibility = 'visible';

addEvent(document, 'click', function () {

document.getElementById('myMenu').style.visibility = 'hidden';

});

});

});

PS：contextmenu事件很常用，这直接导致浏览器兼容性较为稳定。

卸载前事件：beforeunload，这个事件可以帮助在离开本页的时候给出相应的提示，“离开”或者“返回”操作。

addEvent(window, 'beforeunload', function (evt) {

preDef(evt);

});

鼠标滚轮(mousewheel)和DOMMouseScroll，用于获取鼠标上下滚轮的距离。

addEvent(document, 'mousewheel', function (evt) { //非火狐

alert(getWD(evt));

});

addEvent(document, 'DOMMouseScroll', function (evt) { //火狐

alert(getWD(evt));

});

function getWD(evt) {

var e = evt || window.event;

if (e.wheelDelta) {

return e.wheelDelta;

} else if (e.detail) {

return -evt.detail \* 30; //保持计算的统一

}

}

PS：通过浏览器检测可以确定火狐只执行DOMMouseScroll。

DOMContentLoaded事件和readystatechange事件，有关DOM加载方面的事件，关于这两个事件的内容非常多且繁杂，我们先点明在这里，在项目课程中使用的时候详细讨论。