1. 事件绑定、事件监听、事件委托

在 JavaScript 的学习中，我们经常会遇到 JavaScript 的事件机制，例如，事件绑定、事件监听、事件委托（事件代理）等。这些名词是什么意思呢，有什么作用呢?

1.2 事件绑定

要想让 JavaScript 对用户的操作作出响应，首先要对 DOM 元素绑定事件处理函数。所谓事件处理函数，就是处理用户操作的函数，不同的操作对应不同的名称。

在 JavaScript 中，有三种常用的绑定事件的方法：

- 在 DOM 元素中直接绑定；
- 在 JavaScript 代码中绑定；
- 绑定事件监听函数。

我们可以在 DOM 元素上绑定 onclick、onmouseover、onmouseout、onmousedown、onmouseup、ondblclick、onkeydown、onkeypress、onkeyup 等。好多不一一列出了。如果想知道更多事件类型请查看， DOM 事件 。

方式一:

```
<input type="button" value="click me" onclick="hello()">
   function hello(){ alert("hello world!");}
```

在 JavaScript 代码中绑定事件

方式二:

在 JavaScript 代码中绑定事件可以使 JavaScript 代码与 HTML 标签分离，文档结构清晰，便于管理和开发。

```
<input type="button" value="click me" id="btn">

document.getElementById("btn").onclick = function(){
 alert("hello world!");
}
```

特点：

事件名称之间一定要加上 on，比如：onclick、onload、onmousemove。

兼容主流的浏览器，包括低版本的 IE。

当同一个元素绑定多个事件时，只有最后一个事件会被添加，并且传播模式只能是冒泡模式。

方式三事件监听：
绑定事件的另一种方法是用 addEventListener() 或 attachEvent() 来绑定事件监听函数。下面详细介绍，事件监听。

```
document.getElementById("btn").addEventListener( ‘click’ ,function(){
  alert("hello world!");
})
```

1.3 事件监听

关于事件监听，W3C 规范中定义了 3 个事件阶段，依次是捕获阶段、目标阶段、冒泡阶段。

起初 Netscape 制定了 JavaScript 的一套事件驱动机制（即事件捕获）。随即 IE 也推出了自己的一套事件驱动机制（即事件冒泡）。最后 W3C 规范了两种事件机制，分为捕获阶段、目标阶段、冒泡阶段。IE8 以前 IE 一直坚持自己的事件机制（前端人员一直头痛的兼容性问题），IE9 以后 IE 也支持了 W3C 规范。

W3C 规范

语法：

element.addEventListener(event, function, useCapture)

event：（必需）事件名，支持所有 DOM 事件 。
function：（必需）指定要事件触发时执行的函数。
useCapture：（可选）指定事件是否在捕获或冒泡阶段执行。true，捕获。false，冒泡。默认 false。

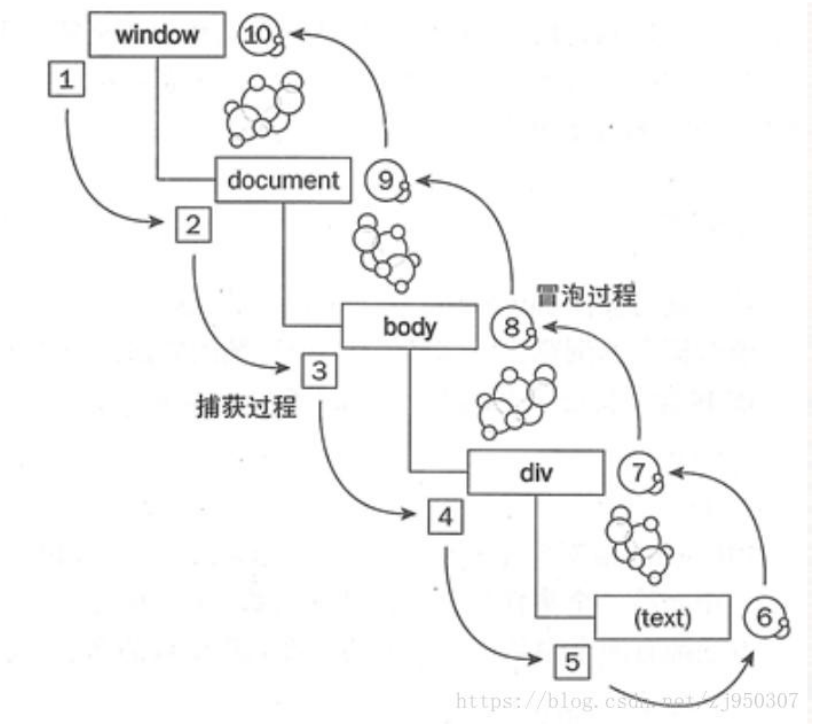
addEventListener()方法特点：

- element.addEventListener(event, function, useCapture)中的第三个参数可以控制指定事件是否在捕获或冒泡阶段执行。true - 事件句柄在捕获阶段执行。false- 默认- 事件句柄在冒泡阶段执行。

- addEventListener() 可以给同一个元素绑定多个事件，不会发生覆盖的情况。如果给同一个元素绑定多个事件，那么采用先绑定先执行的规则。
- 可以使用 removeEventListener() 来移除之前绑定过的事件。
- addEventListener() 在绑定事件的时候，事件名称之前不需带 on 。
- 注意该方法的兼容性，如果要兼容 IE6-8 ，不能使用该方法，可以采用以下方法。

特点一：

首先我们要知道：一个完整的 js 事件流是从 window 开始，最后回到 window 的过程，如下图所示：



1-5 是捕获过程，5-6 是目标阶段，6-10 是冒泡阶段；

事件捕获：

首先 window 会获捕获到事件，之后 document、documentElement、body 会捕获到，再之后就是在 body 中 DOM 元素一层一层的捕获到事件，有 wrapDiv、innerP；
目标阶段：真正点击的元素 textSpan 的事件发生了两次，因为在上面的 JavaScript 代码中，textSapn 既在捕获阶段绑定了事件，又在冒泡阶段绑定了事件，所以发生了两次；

事件冒泡：

和捕获阶段相反的将事件一步一步地冒泡到 window；

可以用例子说明

注：IE8 以下不支持。

特点二：

1、可以绑定多个事件。

<input type="button" value="click me" id="btn3">

```
var btn3 = document.getElementById("btn3");
btn3.onclick = function(){
  alert("hello 1"); //不执行
}
btn3.onclick = function(){
  alert("hello 2"); //执行
}
```

常规的事件绑定只执行最后绑定的事件。

```
<input type="button" value="click me" id="btn4">
```

但是 addEventListener 可以绑定多个，并且都会执行

```
var btn4 = document.getElementById("btn4");
btn4.addEventListener("click",hello1);
btn4.addEventListener("click",hello2);

function hello1(){
  alert("hello 1");
}
function hello2(){
  alert("hello 2");
}
```

两个事件都执行了。

2、可以解除相应的绑定

```
<input type="button" value="click me" id="btn5">

var btn5 = document.getElementById("btn5");
btn5.addEventListener("click",hello1);//执行了
btn5.addEventListener("click",hello2);//不执行
btn5.removeEventListener("click",hello2);
function hello1(){
  alert("hello 1");
}
function hello2(){
  alert("hello 2");
}
```

3、封装事件监听

```
<input type="button" value="click me" id="btn5">

//绑定监听事件
function addEventHandler(target,type,fn){
  if(target.addEventListener){
```

```
 target.addEventListener(type,fn);
 }else{
 target.attachEvent("on"+type,fn);
 }
}

//移除监听事件
function removeEventHandler(target,type,fn){
 if(target.removeEventListener){
 target.removeEventListener(type,fn);
 }else{
 target.detachEvent("on"+type,fn);
 }
}

//测试
var btn5 = document.getElementById("btn5");
addEventHandler(btn5,"click",hello1);//添加事件 hello1
addEventHandler(btn5,"click",hello2);//添加事件 hello2
removeEventHandler(btn5,"click",hello1);//移除事件 hello1
```

1.4 IE 标准 attachEvent

语法:

element.attachEvent(event, function)

event: （必需）事件类型。需加“on “，例如：onclick。

function: （必需）指定要事件触发时执行的函数。

```
<input type="button" value="click me" id="btn2">

document.getElementById("btn2").attachEvent("onclick",hello);
function hello(){
   alert("hello world!");
}
```

attachEvent()

方法事例：

```
window.attachEvent('onload',function(){
      alert("页面加载成功");
});
```

attachEvent()方法特点：

attachEvent 是 IE 有的方法，它不遵循 W3C 标准，而其他的主流浏览器如 FF 等遵循 W3C 标准的浏览器都使用 addEventListener，所以实际开发中需分开处理。
attachEvent()是 后绑定先执行。
绑定事件时，attachEvent 必须带 on，如 onclick，onmouseover 等

1.5 举例总结说明：

1.5.1 传统事件绑定方法

```
window.onload=function(){
    alert("页面加载完毕");
}
document.getElementById("btn").onclick=function(){
    alert("按钮被点击");
}
document.onmousemove=function(){
    console.log("鼠标在移动");
}
```

1.5.2 addEventListener()事件监听

方法事例：

```
window.addEventListener('load',function(){
    alert("页面加载成功");
},false);
```

浏览器兼容性：Internet Explorer 8 及更早 IE 版本不支持 addEventListener() 方法，Opera 7.0 及 Opera 更早版本也不支持。

事件委托

事件委托就是利用冒泡的原理，把事件加到父元素或祖先元素上，触发执行效果。

```
<input type="button" value="click me" id="btn6">

var btn6 = document.getElementById("btn6");
document.onclick = function(event){
 event = event || window.event;
 var target = event.target || event.srcElement;
```

```
 if(target == btn6){
     alert(btn5.value);
 }
}
```

或者

```
window.addEventListener( ‘click’ ,function(event){
 event = event || window.event;
 var target = event.target || event.srcElement;
 if(target == btn6){
     alert(btn5.value);
 }
})
```

标签有 id 这个属性，不使用 getElementById 方法，也可以直接用 id 获取 dom 元素。

如果 dom 元素的 id 名称不和 js 内置属性或全局变量重名的话，该名称自动成为 window 对象的属性，指向表示文档元素的 HTMLElement 元素，所以可以直接用来操作 dom。
看网上的说法是，这个是 IE 首先支持，火狐谷歌后面才支持的。不过现在还未形成标准，为了保险，还是不推荐使用。
不过看各大浏览器都支持.

事件委托优点

1、提高 JavaScript 性能。

事件委托可以显著的提高事件的处理速度，减少内存的占用。 实例分析 JavaScript 中的事件委托和事件绑定。

传统写法

```
<ul id="list">
 <li id="item1" >item1</li>
 <li id="item2" >item2</li>
 <li id="item3" >item3</li>
</ul>

<script>
var item1 = document.getElementById("item1");
var item2 = document.getElementById("item2");
var item3 = document.getElementById("item3");

item1.onclick = function(){
```

```
 alert("hello item1");
}
item2.onclick = function(){
 alert("hello item2");
}
item3.onclick = function(){
 alert("hello item3");
}
</script>
```

事件委托

```
<ul id="list">
  <li id="item1" >item1</li>
  <li id="item2" >item2</li>
  <li id="item3" >item3</li>
</ul>

<script>
var item1 = document.getElementById("item1");
var item2 = document.getElementById("item2");
var item3 = document.getElementById("item3");

document.addEventListener("click",function(event){
 var target = event.target;
 if(target == item1){
     alert("hello item1");
 }else if(target == item2){
     alert("hello item2");
 }else if(target == item3){
     alert("hello item3");
 }
})
</script>
```

2、动态的添加 DOM 元素

不需要因为元素的改动而修改事件绑定。

传统写法

```
<ul id="list">
  <li id="item1" >item1</li>
  <li id="item2" >item2</li>
  <li id="item3" >item3</li>
</ul>
```

```
<script>
var list = document.getElementById("list");

var item = list.getElementsByTagName("li");
for(var i=0;i<item.length;i++){
 (function(i){
        item[i].onclick = function(){
            alert(item[i].innerHTML);
        }
        })(i)
}

var node=document.createElement("li");
var textnode=document.createTextNode("item4");
node.appendChild(textnode);
list.appendChild(node);

</script>
```

点击 item1 到 item3 都有事件响应，但是点击 item4 时，没有事件响应。说明传统的事件绑定无法对动态添加的元素而动态的添加事件。

事件委托

```
<ul id="list">
  <li id="item1" >item1</li>
  <li id="item2" >item2</li>
  <li id="item3" >item3</li>
</ul>

<script>
var list = document.getElementById("list");

document.addEventListener("click",function(event){
 var target = event.target;
 if(target.nodeName == "LI"){
       alert(target.innerHTML);
 }
})

var node=document.createElement("li");
var textnode=document.createTextNode("item4");
node.appendChild(textnode);
list.appendChild(node);
```

</script>
当点击 item4 时，item4 有事件响应。说明事件委托可以为新添加的 DOM 元素动态的添加事件。

补充：事件代理（事件委托）

要求：鼠标放到 li 上 li 的背景颜色变为灰色；

```
<ul>
                <li>item1</li>
                <li>item2</li>
                <li>item3</li>
                <li>item4</li>
                <li>item5</li>
                <li>item6</li>
            </ul>
```

不使用事件代理：

```
$("li").on("mouseover",function(){

$(this).css("background-color","#ddd").siblings().css("background-color","white");

 })
```

使用事件代理：

```
$("ul").on("mouseover",function(e){

$(e.target).css("background-color","#ddd").siblings().css("background-color","white");

})
```

看上去并没有什么区别，但是使用事件代理的方法少了遍历所有 li 节点的操作，性能上肯定更加优化，而且如果后期又动态的增加 li 节点，不使用事件代理的话还需要再重新给新增的 li 节点绑定事件，但是使用事件代理的话则不需要了。

事件捕获和事件冒泡例子

以一段具体的代码来看

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style type="text/css">
    #wrapDiv, #innerP, #textSpan{
        margin: 5px;padding: 5px;box-sizing: border-box;cursor: default;
    }
    #wrapDiv{
        width: 300px;height: 300px;border: indianred 3px solid;
    }
    #innerP{
        width: 200px;height: 200px;border: hotpink 3px solid;
    }
    #textSpan{
        display: block;width: 100px;height: 100px;border: orange 3px solid;
    }
    </style>
</head>
<body>
     <div id="wrapDiv">wrapDiv
        <p id="innerP">innerP
            <span id="textSpan">textSpan</span>
        </p>
    </div>
    <script>
    var wrapDiv = document.getElementById("wrapDiv");
    var innerP = document.getElementById("innerP");
    var textSpan = document.getElementById("textSpan");

    // 捕获阶段绑定事件
    window.addEventListener("click", function(e){
        console.log("window 捕获", e.target.nodeName, e.currentTarget.nodeName);
    }, true);

    document.addEventListener("click", function(e){
        console.log("document 捕获", e.target.nodeName, e.currentTarget.nodeName);
    }, true);

    document.documentElement.addEventListener("click", function(e){
        console.log("documentElement 捕获", e.target.nodeName,
e.currentTarget.nodeName);
```

```
    }, true);

    document.body.addEventListener("click", function(e){
        console.log("body 捕获", e.target.nodeName, e.currentTarget.nodeName);
    }, true);

    wrapDiv.addEventListener("click", function(e){
        console.log("wrapDiv 捕获", e.target.nodeName, e.currentTarget.nodeName);
    }, true);

    innerP.addEventListener("click", function(e){
        console.log("innerP 捕获", e.target.nodeName, e.currentTarget.nodeName);
    }, true);

    textSpan.addEventListener("click", function(e){
        console.log("textSpan 捕获", e.target.nodeName, e.currentTarget.nodeName);
    }, true);

    // 冒泡阶段绑定的事件
    window.addEventListener("click", function(e){
        console.log("window 冒泡", e.target.nodeName, e.currentTarget.nodeName);
    }, false);

    document.addEventListener("click", function(e){
        console.log("document 冒泡", e.target.nodeName, e.currentTarget.nodeName);
    }, false);

    document.documentElement.addEventListener("click", function(e){
        console.log("documentElement 冒泡", e.target.nodeName,
e.currentTarget.nodeName);
    }, false);

    document.body.addEventListener("click", function(e){
        console.log("body 冒泡", e.target.nodeName, e.currentTarget.nodeName);
    }, false);

    wrapDiv.addEventListener("click", function(e){
        console.log("wrapDiv 冒泡", e.target.nodeName, e.currentTarget.nodeName);
    }, false);

    innerP.addEventListener("click", function(e){
        console.log("innerP 冒泡", e.target.nodeName, e.currentTarget.nodeName);
    }, false);
```

```
        textSpan.addEventListener("click", function(e){
            console.log("textSpan 冒泡", e.target.nodeName, e.currentTarget.nodeName);
        }, false);
</script>
</body>
</html>
```