第10章 事件

事件最早是在IE3和Netscape Navigator2中出现的，当时是作为分担服务器运算负担的一种手段。要实现和网页的互动，就需要通过JavaScript里面的事件来实现。每次用户与一个网页进行交互，例如点击链接，按下一个按键或者移动鼠标时，就会触发一个事件。我们的程序可以检测这些事件，然后对此作出响应。从而形成一种交互。这样可以使我们的页面变得更加的有意思，而不仅仅像以前一样只能进行浏览。

本章将学习如下内容：

- 事件的介绍
- 什么是事件流
- 事件冒泡和事件捕获
- 添加事件监听器
- 事件对象介绍
- 事件类型介绍
- 事件模拟

10-1 事件与事件流

10-1-1 事件介绍

JavaScript和HTML之间的交互是通过当用户或者浏览器操作网页时发生的事件来处理的。页面装载时，是一个事件，用户点击页面上的某个按钮时，也是一个事件。在早期拨号上网的年代，如果所有的功能都放在服务器端进行处理的话，效率是相当之低的。所以JavaScript最初被设计出来就是用来解决这些问题的。通过允许一些功能在客户端上发生，以节省到服务器的往返时间。

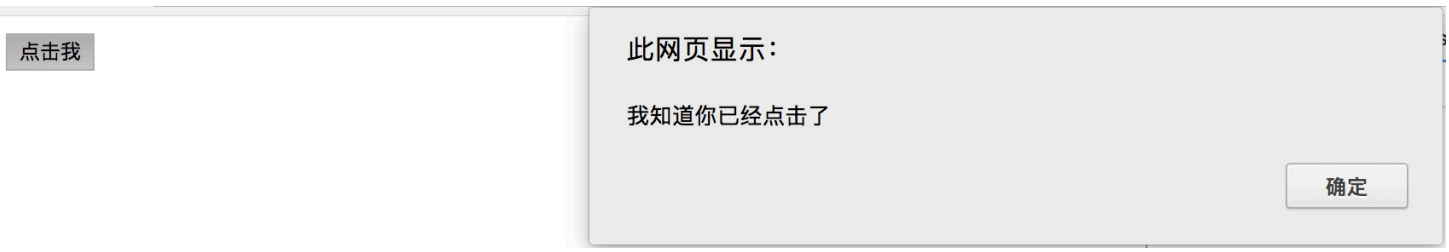
JavaScript中采用一个叫做事件监听器的东西来监听事件是否发生。这个事件监听器类似于一个通知，当事件发生时，事件监听器会让我们知道，然后程序就可以做出相应的响应。通过这种方式，就可以避免让程序不断地去检查事件是否发生，让程序在等待事件发生的同时，可以继续做其他的任务。这其实就是后面我们会介绍到的异步，后面的章节会具体的介绍异步相关的知识。

接下来我们来看一个事件的快速入门案例，如下：

```
<body>
    <button onclick="test()">点击我</button>
    <script>
```

```
        let test = function(){
            alert("我知道你已经点击了");
        }
    </script>
</body>
```

效果：点击按钮以后弹出一个对话框



这里我们就给`<button>`元素节点绑定了一个点击事件，当用户点击该按钮时，会执行`test()`程序。

10-1-2 事件流介绍

当浏览器发展到第四代时(IE4及Netscape4)，浏览器开发团队遇到了一个很有意思的问题：页面的哪一部分会拥有某个特定的事件？想象画在一张纸上的一组同心圆。如果把手指放在圆心上，那么手指指向的不是一个圆，而是纸上的所有圆。

好在两家公司的浏览器开发团队在看待浏览器事件方面还是一致的。如果单击了某个按钮，他们都认为单击事件不仅仅发生在按钮上，甚至也单击了整个页面。

但有意思的是，IE和Netscape开发团队居然提出了差不多是完全相反的事件流的概念。IE的事件流是事件冒泡流，而Netscape的事件流是事件捕获流。

10-1-3 事件冒泡

IE的事件流叫做事件冒泡(event bubbling)，即事件开始时由最具体的元素(文档中嵌套层次最深的那个节点)接收，然后逐级向上传播到较为不具体的节点(文档)，以下列HTML结构为例，来说明事件冒泡。如下：

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Document</title>
    </head>
    <body>
        <div></div>
    </body>
```

```
</html>
```

如果单击了页面中的`<div>`元素，那么这个`click`事件沿DOM树向上传播，在每一级节点上都会发生，按照如下顺序传播：

```
1.<div>
2.<body>
3.<html>
4.document
```

> 注意:所有现代浏览器都支持事件冒泡，但在具体实现在还是有一些差别。IE9、Firefox、Chrome、Safari将事件一直冒泡到window对象

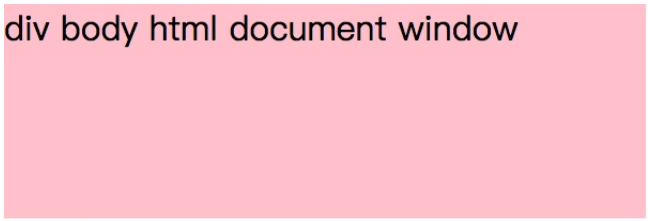
```
1.<div>
2.<body>
3.<html>
4.document
5.window
```

我们可以通过下面的代码，来查看文档具体的冒泡顺序，示例如下：

```
<body>
    <div id="box" style="height:100px;width:300px;background-color:pink;"></
div>
    <button id="reset">还原</button>
    <script>
        //IE8-浏览器返回div body html document
        //其他浏览器返回div body html document window
        reset.onclick = function () {
            history.go();
        }
        box.onclick = function () {
            box.innerHTML += 'div\n';
        }
        document.body.onclick = function () {
            box.innerHTML += 'body\n';
        }
        document.documentElement.onclick = function () {
            box.innerHTML += 'html\n';
        }
        document.onclick = function () {
            box.innerHTML += 'document\n';
        }
```

```
        window.onclick = function () {
            box.innerHTML += 'window\n';
        }
    </script>
</body>
```

效果：当我们点击了页面上的div元素以后，出现的文字顺序如下



事件代理

事件代理，又被称之为事件委托。在JavaScript中，添加到页面上的事件处理程序数量将直接关系到页面整体的运行性能。导致这一问题的原因是多方面的。首先，每个函数都是对象，都会占用内存。内存中的对象越多，性能就越差。其次，必须事先指定所有事件处理程序而导致的DOM访问次数，会延迟整个页面的交互就绪时间。

对事件处理程序过多问题的解决方案就是事件代理。事件代理利用了事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。例如，click事件会一直冒泡到document层次。也就是说，我们可以为整个页面指定一个onclick事件处理程序，而不必给每个可单击的元素分别添加事件处理程序。

举一个具体的例子：例如现在我的列表项有如下内容

```
<body>
    <ul id="color-list">
        <li>red</li>
        <li>yellow</li>
        <li>blue</li>
        <li>green</li>
        <li>black</li>
        <li>white</li>
    </ul>
</body>
```

如果我们想把事件监听器绑定到所有的 `<li>` 元素上面，这样它们被单击的时候就弹出一些文字，为此我们需要给每一个元素来绑定一个事件监听器。虽然上面的例子中好像问题也不大，但是想象一下如果这个列表有100个元素，那我们就需要添加100个事件监听器，这个工作量还是

很恐怖的。

这个时候我们就可以利用事件代理来帮助我们解决这个问题。将事件监听器绑定到父元素 `<ul>` 上，这样即可对所有的 `<li>` 元素添加事件，如下：

```
<body>
    <ul id="color-list">
        <li>red</li>
        <li>yellow</li>
        <li>blue</li>
        <li>green</li>
        <li>black</li>
        <li>white</li>
    </ul>
    <script>
        let colorList = document.getElementById("color-list");
        colorList.addEventListener("click",function(){
            alert("Hello");
        })
    </script>
</body>
```

现在我们单击列表中的任何一个 `<li>` 都会弹出东西，就好像这些 `<li>` 元素就是 `click` 事件的目标一样。并且如果我们之后再为这个 `<ul>` 添加新的 `<li>` 元素的话，新的 `<li>` 元素也会自动添加上相同的事件。

但是，这个时候也存在一个问题，虽然我们使用事件代理解决了为每一个 `<li>` 元素添加事件的问题，但是如果我们没有点击 `<li>`，而是点击的 `<ul>`，同样会触发事件。这也非常好理解，因为你事件就是绑定在人家 `<ul>` 上面的。我们可以对点击的节点进行一个小小的判断，从而保证用户只在点击 `<li>` 的时候才触发事件，如下：

```
<body>
    <ul id="color-list">
        <li>red</li>
        <li>yellow</li>
        <li>blue</li>
        <li>green</li>
        <li>black</li>
        <li>white</li>
    </ul>
    <script>
        let colorList = document.getElementById("color-list");
        colorList.addEventListener("click", function (e) {
            if (e.target.nodeName === 'LI') {
```

```
                alert('点击 li')
            }
        })
    </script>
</body>
```

阻止冒泡

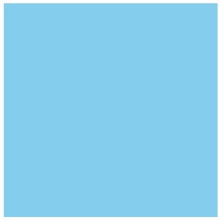
上面我们介绍了事件代理，其本质就是通过事件冒泡来实现的。事件冒泡允许多个操作被集中处理，把事件处理器添加到一个父级元素上，避免把事件处理器添加到多个子级元素上，这可以看作是事件冒泡的一个优点。

但是事件冒泡也有其它的缺点。那就是如果父级元素和后代元素都绑定了相同的事件，那么后代元素触发事件时，父级元素的事件也会被触发。
这里我们来看一个示例如下：

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        .father{
            width: 200px;
            height: 200px;
            background-color: pink;
        }
        .son{
            width: 100px;
            height: 100px;
            background-color: skyblue;
            position: absolute;
            left: 500px;
        }
    </style>
</head>
<body>
    <div class="father" onclick="test()">
        <div class="son" onclick="test2()"></div>
    </div>
    <script>
        let test = function(){
            console.log("你点击了父元素");
        }
        let test2 = function(){
            console.log("你点击了子元素");
        }
```

```
    </script>
</body>
```

效果：当我们点击了子元素之后，父元素上面绑定的事件也同时被触发



```
你点击了子元素
你点击了父元素
```

通常情况下我们都是一步到位，明确自己的事件触发源，并不希望浏览器自作聪明、漫无目的地去帮我们找合适的事件处理程序，所以这种情况下我们不需要事件冒泡。

另外，通过对事件冒泡的理解，我们知道事件冒泡会让程序将做很多额外的事情，这必然增大程序开销。事件冒泡处理可能会激活我们本来不想激活的事件，导致程序错乱，甚至无从下手调试，这常成为对事件冒泡不熟悉程序员的棘手问题。

所以必要时，我们要阻止事件冒泡。

我们可以使用`stopPropagation()`方法来阻止冒泡。下面演示现代浏览器中通过`stopPropagation()`来阻止冒泡的方式。

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        .father{
            width: 200px;
            height: 200px;
            background-color: pink;
        }
        .son{
            width: 100px;
            height: 100px;
            background-color: skyblue;
            position: absolute;
            left: 500px;
        }
```

```
    </style>
</head>
<body>
    <div class="father" onclick="test()">
        <div class="son" onclick="test2()"></div>
    </div>
    <script>
        let test = function(){
            console.log("你点击了父元素");
        }
        let test2 = function(){
            console.log("你点击了子元素");
            event.stopPropagation();//阻止事件向上冒泡
        }
    </script>
</body>
```

效果：当我们再次点击子元素来触发事件时，不会再同时触发绑定在父元素上面的事件

```
你点击了子元素
```

10-1-4 事件捕获

Netscape Communicator团队提出的另一种事件流叫做事件捕获(event captruing)。事件捕获的思想是不太具体的节点应该更早接收到事件，而最具体的节点应该最后接收到事件。事件捕获的用意在于在事件到达预定目标之前就捕获它。以同样的HTML结构为例，说明事件捕获，如下：

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Document</title>
    </head>
    <body>
        <div></div>
    </body>
</html>
```

在事件捕获过程中，document对象首先接收到click事件，然后事件沿DOM树依次向下，一直传播到事件的实际目标，即 `<div>` 元素

```
1.document
2.<html>
3.<body>
```

```
4.<div>
```

> 注意:IE9、Firefox、Chrome、Safari等现代浏览器都支持事件捕获，但是也是从window对象开始捕获。

```
1.window
2.document
3.<html>
4.<body>
5.<div>
```

后面我们会学习`addEventListener()`这个方法。在这个方法中当第三个参数设置为true时，即为事件捕获。我们由此可以来观察一下事件捕获的一个顺序，如下：

```
<body>
    <div id="box" style="height:100px;width:300px;background-color:pink;"></
div>
    <button id="reset">还原</button>
    <script>
        //IE8-浏览器不支持
        //其他浏览器返回window document html body div
        reset.onclick = function () {
            history.go();
        }
        box.addEventListener('click', function () {
            box.innerHTML += 'div\n'
        }, true)
        document.body.addEventListener('click', function () {
            box.innerHTML += 'body\n';
        }, true);
        document.documentElement.addEventListener('click',function(){
            box.innerHTML+='html\n';
        },true);
        document.addEventListener('click', function () {
            box.innerHTML += 'document\n';
        }, true);
        window.addEventListener('click', function () {
            box.innerHTML += 'window\n';
        }, true);
    </script>
</body>
```

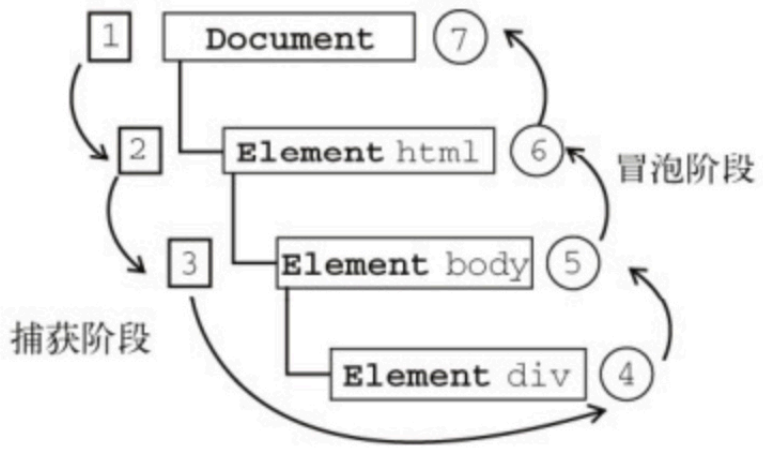
效果：当我们点击了页面的div元素以后，出现的文字顺序如下：

window document html body div

还原

10-1-5 DOM事件流

DOM标准采用捕获+冒泡。两种事件流都会触发DOM的所有对象，从document对象开始，也在document对象结束。换句话说，起点和终点都是document对象(很多浏览器可以一直捕获/冒泡到window对象)
DOM事件流示意图：



DOM标准规定事件流包括三个阶段：事件捕获阶段、处于目标阶段和事件冒泡阶段。
事件捕获阶段：实际目标 `<div>` 在捕获阶段不会接收事件。也就是说在捕获阶段，事件从 `document` 到 `<html>` 再到 `<body>` 就停止了。上图中为1~3。
处于目标阶段：事件在 `<div>` 上发生并处理。但是事件处理会被看成是冒泡阶段的一部分。
冒泡阶段：事件又传播回文档。

10-2 事件监听器

事件监听器，又被称之为事件处理程序。简单来说，就是和事件绑定在一起的函数。当事件发生时就会执行函数中相应的代码。事件监听器根据DOM级别的不同写法上也是有所区别的，不仅写法有区别，功能上也是逐渐的在完善。

10-2-1 HTML事件监听器

HTML事件监听器，又被称之为行内事件监听器。这是在浏览器中处理事件最原始的方法。我们在最开始的时候写的事件快速入门案例就是一个HTML事件处理程序。

```
<body>
    <button onclick="test()">点击我</button>
    <script>
        let test = function(){
            alert("我知道你已经点击了");
        }
    </script>
</body>
```

但是有一点需要注意，就是这种方法已经过时了，原因如下：

- JavaScript代码与HTML标记混杂在一起，破坏了结构和行为分离的理念
- 每个元素只能为每种事件类型绑定一个事件处理器
- 事件处理器的代码隐藏于标记中，很难找到事件是在哪里声明的

但是如果是做简单的事件测试，那么这种写法还是非常方便快捷的。

10-2-2 DOM 0级事件监听器

这种方式是首先取到要为其绑定事件的元素节点对象，然后给这些节点对象的事件处理属性赋值一个函数。这样就可以达到JavaScript代码和HTML代码相分离的目的，如下：

```
<body>
    <button id="test">点击我</button>
    <script>
        let test = document.getElementById("test");
        test.onclick = function(){
            console.log("this is a test");
        }
```

```
    </script>
</body>
```

这种方式虽然相比HTML事件监听器有所改进，但是它也有一个缺点，那就是它依然存在每个元素只能绑定一个函数的局限性。下面我们尝试使用这种方式为同一个元素节点绑定2个事件，如下：

```
<body>
    <button id="test">点击我</button>
    <script>
        let test = document.getElementById("test");
        test.onclick = function(){
            console.log("this is a test");
        }
        test.onclick = function(){
            console.log("this is a test,too");
        }
    </script>
</body>
```

效果：点击按钮会只会弹出后面的"this is a test,too"，因为后面的事件处理程序把前面的事件处理程序给覆盖掉了。

10-2-3 DOM 2级事件监听器

DOM2级事件处理程序通过`addEventListener()`可以为一个元素添加多个事件处理程序。
这个方法接收3个参数：事件名，事件处理函数，布尔值。如果这个布尔值为true，则在捕获阶段处理事件，如果为false，则在冒泡阶段处理事件。若最后的布尔值不填写，则和false效果一样，也就是说默认为false，在冒泡阶段进行事件的处理。

接下来我们来看下面的示例：这里我们为button元素绑定了两个事件处理程序，并且两个事件处理程序都是通过点击来触发。

```
<body>
    <button id="test">点击我</button>
    <script>
        let test = document.getElementById("test");
        test.addEventListener("click",function(){
            console.log("this is a test");
        },false);
        test.addEventListener("click",function(){
            console.log("this is a test,too");
        },false);
```

```
    </script>
</body>
```

效果：可以看到绑定在上面的两个事件处理程序都工作正常

```
this is a test
this is a test,too
```

10-2-4 删除事件监听器

如果是通过DOM 0级来添加的事件，那么删除的方法很简单，只需要将节点对象的事件处理属性赋值为null即可，如下：

```
<body>
    <button id="test">点击我</button>
    <script>
        let test = document.getElementById("test");
        test.onclick = function(){
            console.log("this is a test");
        }
        test.onclick = null;//删除事件绑定
    </script>
</body>
```

如果是通过DOM 2级来添加的事件，我们可以使用 `removeEventLister()` 来进行事件的删除。但是需要注意的是，如果要通过该方法移除某一类事件类型的一个事件的话，在通过 `addEventListener()` 来绑定事件时的写法就要稍作改变。先单独将绑定函数写好，然后 `addEventListener()` 进行绑定时第二个参数传入要绑定的函数名称即可。

示例如下：

```
<body>
    <button id="test">点击我</button>
    <script>
        let test = document.getElementById("test");
        //DOM 2级添加事件
        let fn1 = function(){
            console.log("this is a test");
        }
        let fn2 = function(){
            console.log("this is a test,too");
        }
        test.addEventListener("click",fn1,false);
        test.addEventListener("click",fn2,false);
```

```
        test.removeEventListener("click",fn1);//只删除第一个点击事件
    </script>
</body>
```

效果：第一个点击事件已经被移除掉了

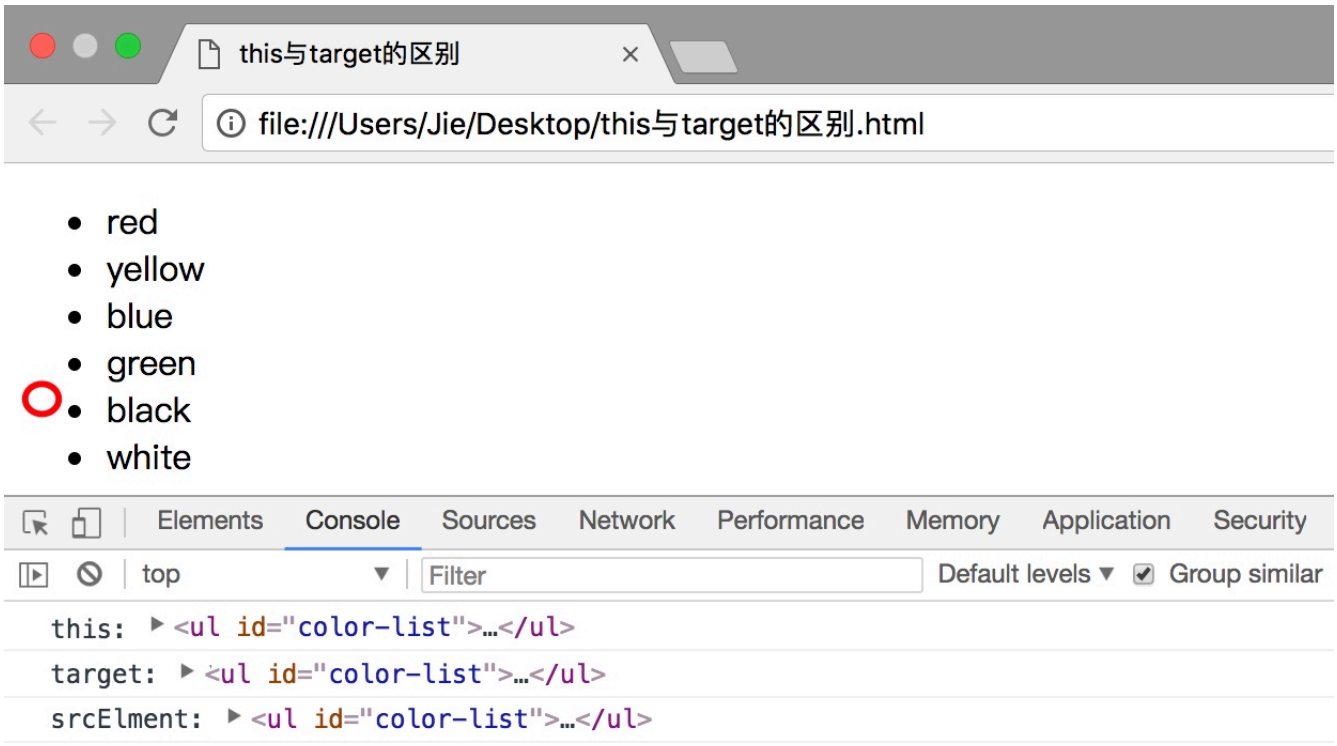
```
this is a test,too
```

10-2-5 this值

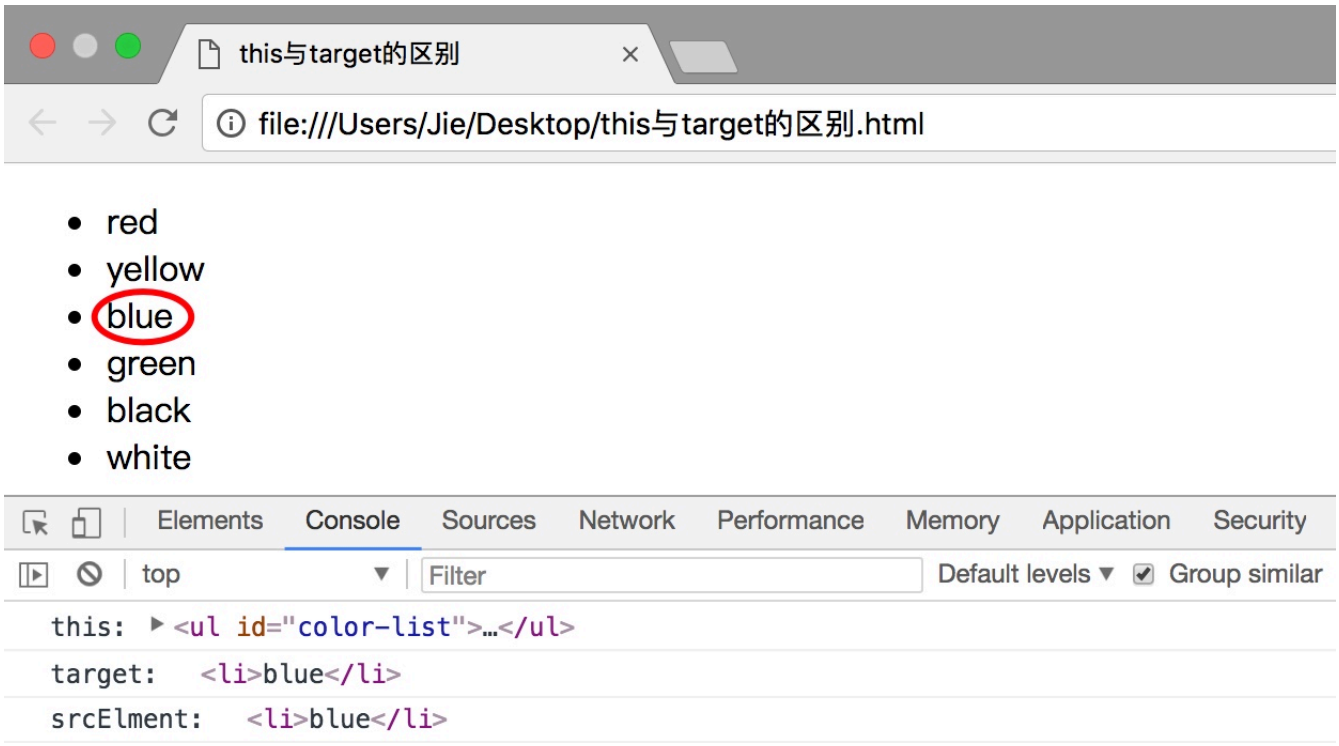
当我们触发一个事件的时候，事件处理程序里面的this指代的是绑定事件的元素，而事件对象的`target`属性指代的是触发事件的元素，`target`和`srcElememnt`是等价的，我们来看下面的例子：

```
<body>
    <ul id="color-list">
        <li>red</li>
        <li>yellow</li>
        <li>blue</li>
        <li>green</li>
        <li>black</li>
        <li>white</li>
    </ul>
    <script>
        // this 是绑定事件的元素
        // target 是触发事件的元素 和srcElememnt等价
        let colorList = document.getElementById("color-list");
        colorList.addEventListener("click", function (event) {
            console.log('this:', this);
            console.log('target:', event.target);
            console.log('srcElement:', event.srcElement);
        })
    </script>
</body>
```

效果：当我点击`<ul>`的地方时，由于绑定事件的元素和触发事件的元素相同，所以都显示出`<ul>`。红色画圈部分是我点击的位置，如下：



当我点击`<li>`的时候，区别就出来了，红色画圈部分是我点击的位置，如下：

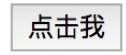


当我们绑定事件的元素和触发事件的元素为**同一个元素**时，我们可以使用`this`来指代触发事件的元素本身，从而对触发事件的元素进行修改，示例如下：

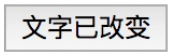
```
<body>
    <button id="test">点击我</button>
    <script>
        let test = document.getElementById("test");
        test.addEventListener("click",function(){
```

```
            this.innerText = "文字已改变";
        })
    </script>
</body>
```

效果：点击前



点击后：



> 注：必须绑定事件的元素和触发事件的元素为同一个元素，如果不为同一个元素，那还是只有通过`事件对象.target`来进行修改。

有些时候我们会遇到一些困扰，比如在div节点的事件函数内部，有一个局部的callback方法，该方法被作为普通函数调用时，callback内部的this是指向全局对象window的，示例如下：

```
<body>
    <div id="div1">我是一个div</div>
    <script>
        window.id = 'window';
        document.getElementById('div1').onclick = function(){
            console.log(this.id); // div1
            let callback = function(){
                console.log(this.id); // 因为是普通函数调用，所以this指向window
            }
            callback();
        }
    </script>
</body>
```

此时有一种简单的解决方案，可以用一个变量保存div节点的引用，如下：

```
<body>
    <div id="div1">我是一个div</div>
    <script>
        window.id = 'window';
        document.getElementById('div1').onclick = function(){
            console.log(this.id); // div1
            let that = this; // 保存当前this的指向
            let callback = function(){
                console.log(that.id); // div1
            }
            callback();
```

```
        }
    </script>
</body>
```

在ECMAScript5的严格模式下，这种情况下的this已经被规定为不会指向全局对象，而是undefined

```
<body>
    <div id="div1">我是一个div</div>
    <script>
        window.id = 'window';
        document.getElementById('div1').onclick = function(){
            console.log(this); // <div id="div1">我是一个div</div>
            let callback = function(){
                "use strict"
                console.log(this); // undefined
            }
            callback();
        }
    </script>
</body>
```

10-3 事件对象

在上一节中我们已经介绍了事件监听器。每当事件被触发时，事件监听器就会执行相应的代码。而每当事件监听器被触发时，监听器上所绑定的回调函数里面都会自动传入一个事件对象`event`。该对象包含了许多和当前所触发的事件相关的信息。例如导致事件的元素是什么，事件的类型是什么以及其他的与特定事件相关的信息。这一小节我们就来看一下有关事件对象的相关知识。

首先，事件对象是以事件的处理函数中的参数形式出现，并不需要我们自己创建，直接使用即可。

语法结构如下：

```
事件源.addEventListener(eventName, function(event){
    // event 就是事件对象
}, boolean)
```

事件对象说明：

- 当事件发生时，只能在事件函数内部访问的对象
- 处理函数结束后会自动销毁

兼容的事件对象

使用用 DOM 标准的事件绑定时，Event 事件对象在 IE8 及之前的版本浏览器器情况有所不不同。

- IE9 及之后的版本和其他浏览器：通过事件的处理函数的形参直接得到Event对象。

```
btn.onclick = function(event){
  console.log(event);
}
```

- IE8 及之前的版本浏览器：Event事件对象被提供在window对象中。

```
btn.onclick = function(event){
    console.log(window.event);
}
```

想要实现Event事件对象的兼容，我们可以在事件的处理理函数中添加以下代码：

```
btn.onclick = function(event){
    event = event || window.event;
}
```

10-3-1 事件类型

事件对象的`type`属性会返回当前所触发的事件类型是什么，示例如下：

```
<body>
    <button id="test">点击我</button>
    <script>
        let test = document.getElementById("test");
        test.addEventListener("click",function(event){
            console.log(event.type);//click
        },false);
    </script>
</body>
```

> 注意：`event`对象会自动被传入到所绑定的回调函数里面，所以即便回调函数没有书写以`event`作为的形参，函数内部依然可以正常使用。

10-3-2 事件目标

`target`属性返回对触发事件的元素节点对象的一个引用，示例如下：

```
<body>
    <button id="test">点击我</button>
    <script>
        let test = document.getElementById("test");
        test.addEventListener("click",function(){
            console.log(event.target);//<button id="test">点击我</button>
        },false);
    </script>
</body>
```

一般来讲，这个事件目标的值和当前的`this`值是相等的，都包含了触发事件的那个元素节点。

```
<body>
    <button id="test">点击我</button>
```

```
    <script>
        let test = document.getElementById("test");
        test.addEventListener("click",function(){
            console.log(event.target === this);//true
        },false);
    </script>
</body>
```

10-3-3 事件坐标

事件对象有几个属性，可以找到鼠标事件发生的位置。如下：

相对于浏览器器位置:

- event.clientX：返回当事件被触发时鼠标指针相对于浏览器页面(或客户区)的水平坐标。(只包含文档的可见部分，不包含窗口本身的控件以及滚动条)
- event.clientY：返回当事件被触发时鼠标指针相对于浏览器页面(客户区)的垂直坐标。(只包含文档的可见部分，不包含窗口本身的控件以及滚动条)
- event.pageX：可以获取事件发生时光标相对于当前窗口的水平坐标信息(包含窗口自身的控件和滚动条)
- event.pageY：可以获取事件发生时光标相对于当前窗口的垂直坐标信息(包含窗口自身的控件和滚动条)

相对于屏幕位置:

- event.screenX：返回事件发生时鼠标指针相对于电脑屏幕左上角的水平坐标。
- event.screenY：返回事件发生时鼠标指针相对于电脑屏幕左上角的垂直坐标。

相对于事件源位置:

- event.offsetX：返回事件发生时鼠标指针相对于事件源的水平坐标
- event.offsetY：返回事件发生时鼠标指针相对于事件源的垂直坐标
- event.layerX：返回事件发生时鼠标指针相对于事件源的水平坐标(Firefox)
- event.layerY：返回事件发生时鼠标指针相对于事件源的垂直坐标(Firefox)

这几个事件属性看似类似，但是有那么一些细微的不同。它们有助于查明点击的位置，或者鼠标光标的位置。示例如下：

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        body {
```

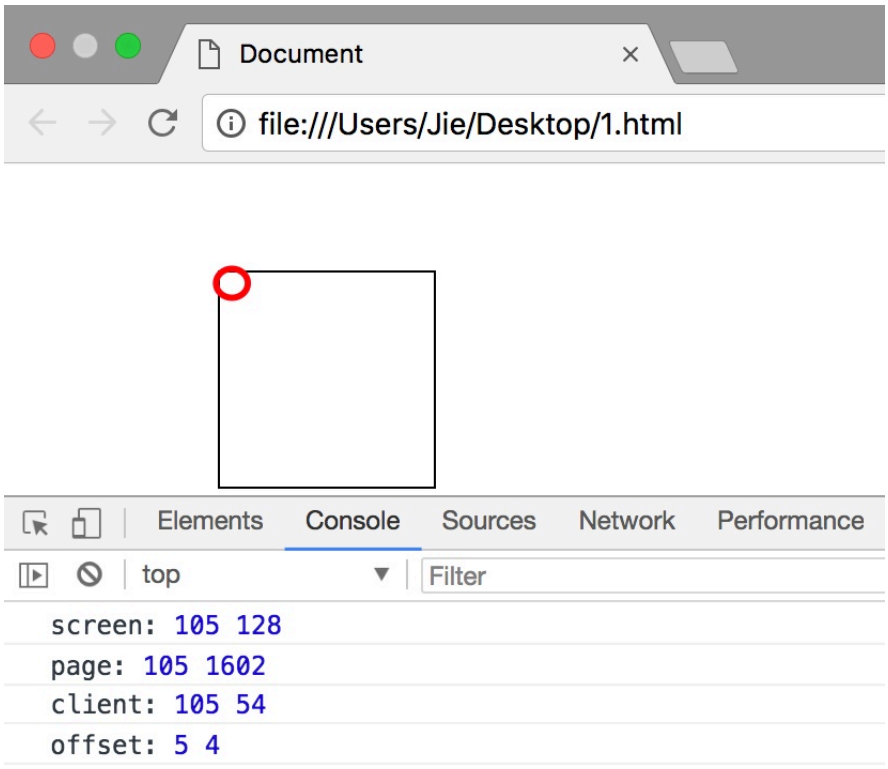
```
            height: 5000px;
        }

        div {
            position: fixed;
            top: 50px;
            left: 100px;
            width: 100px;
            height: 100px;
            border: 1px solid;
        }
    </style>
</head>

<body>
    <div id="box"></div>
    <script>
        box.addEventListener("click", function () {
            console.log("screen:", event.screenX, event.screenY);
            console.log("page:", event.pageX, event.pageY);
            console.log("client:", event.clientX, event.clientY);
            console.log("offset:", event.offsetX, event.offsetY);
        }, false);
    </script>
</body>
```

效果：将窗口向下滚动一段距离后，点击鼠标，出现了以下数值，这里分别做一些解释

- screen：参考点为电脑屏幕左上角
- page：参考点为文档左上角，这里由于我滚动了页面，所以滚动的部分也会被记入其中
- client：参照点始终为当前视口的左上角，无论是否滚动了页面
- offset：当前事件源的左上角作为参考点



10-3-4 事件流

事件对象的`eventPhase`属性可以返回一个整数值，表示事件目前所处的事件流阶段。0表示事件没有发生，1表示当前事件流处于捕获阶段，2表示处于目标阶段，3表示冒泡阶段。

> 注意：IE8及以下浏览器不支持

```
<body>
    <button id="test">点击我</button>
    <script>
        test.onclick = function(){
            test.innerText = event.eventPhase;//2
        }
    </script>
</body>
```

效果：点击按钮后里面的文本变为2，说明处于目标阶段。

```
<body>
    <button id="test">点击我</button>
    <script>
        document.addEventListener("click",function(){
            test.innerText = event.eventPhase;//1
        },true);//最后的布尔参数值为true，说明在捕获阶段处理事件
    </script>
</body>
```

效果：点击按钮后里面的文本变为1，说明处于捕获阶段。

```
<body>
    <button id="test">点击我</button>
    <script>
        document.addEventListener("click",function(){
            test.innerText = event.eventPhase;//3
        },false);//最后的布尔参数值为false，说明在冒泡阶段处理事件
    </script>
</body>
```

效果：点击按钮后里面的文本变为3，说明处于冒泡阶段。

10-3-5 阻止默认行为

常见的默认行为有点击链接后，浏览器跳转到指定页面，或者按一下空格键，页面向下滚动一段距离

关于取消默认行为的方式
有`cancelable`、`defaultPrevented`、`preventDefault()`和`returnValue`

- 在DOM0级事件处理程序中取消默认行为，使用`returnValue`、`preventDefault()`和`return false`都有效
- 在DOM2级事件处理程序中取消默认行为，使用`return false`无效
- 在IE事件处理程序中取消默认行为，使用`preventDefault()`无效

举个例子：点击下列锚点时，会自动打开百度

```
<a href="http://www.baidu.com">百度</a>
```

cancelable

`cancelable`属性返回一个布尔值，表示事件是否可以取消。该属性为只读属性。返回true时，表示可以取消。否则，表示不可取消。

> 注意:IE8及以下浏览器不支持

```
<body>
    <a id="test" href="http://www.baidu.com">百度</a>
    <script>
```

```
        let test = document.getElementById("test");
        test.onclick = function(){
           test.innerHTML = event.cancelable;//true
        }
    </script>
</body>
```

效果：首先`<a>`标签的文本内容会变为`true`，然后跳转到百度页面。

preventDefault()

`preventDefault()`方法DOM中最标准的取消浏览器默认行为的方式，无返回值。

> 注意:IE8及以下浏览器不支持

```
<body>
    <a id="test" href="http://www.baidu.com">百度</a>
    <script>
        let test = document.getElementById("test");
        test.onclick = function(){
            event.preventDefault();
        }
    </script>
</body>
```

returnValue

`returnValue`属性可读写，默认值是true，但将其设置为`false`就可以取消事件的默认行为，与`preventDefault()`方法的作用相同。最早在IE中事件对象中实现取消默认行为的方式，但是现在大多数浏览器都实现了该方式。

> 注意:firefox和IE9+浏览器不支持

```
<body>
    <a id="test" href="http://www.baidu.com">百度</a>
    <script>
        let test = document.getElementById("test");
        test.onclick = function(){
           event.returnValue = false;
        }
    </script>
</body>
```

return false

除了以上方法外，取消默认事件还可以使用`return false`

```
<body>
    <a id="test" href="http://www.baidu.com">百度</a>
    <script>
         let test = document.getElementById("test");
         test.onclick = function(){
            return false;
         }
    </script>
</body>
```

defaultPrevented

`defaultPrevented`属性表示默认行为是否被阻止，返回true时表示被阻止，返回false时，表示未被阻止

> 注意:IE8及以下浏览器不支持

```
<body>
    <a id="test" href="http://www.baidu.com">百度</a>
    <script>
        let test = document.getElementById("test");
        test.onclick = function(){
            //采用两种不同的方式来阻止浏览器默认行为，这是为了照顾其兼容性
            if(event.preventDefault)
            {
                event.preventDefault();
            }
            else{
                event.returnValue = false;
            }
            //将是否阻止默认行为的结果赋值给<a>标签的文本内容
            test.innerHTML = event.defaultPrevented;
        }
    </script>
</body>
```

效果：点击`<a>`标签之后里面的文本内容会变为`true`，因为默认行为已经被阻止。

10-4 事件类型

前面有提到过，点击鼠标可以触发一个事件，按下键盘上的一个键也能触发一个事件。而这两种事件的事件类型是不相同的。在JavaScript里面的事件类型有多种，不同的事件类型具有不同的信息。下面将重点介绍鼠标，键盘和触摸设备时发生的较为常见的事件。

10-4-1 鼠标事件

鼠标事件是Web开发中最常见的一类事件。DOM3级事件中定义了9个鼠标事件。

- click：用户单机主鼠标按钮（一般是左键）或者按下回车键时触发。
- dblclick：用户双击鼠标按钮时触发
- mousedown：用户按下了任意鼠标按钮时触发
- mouseenter：进入元素时触发，但是再进入到子元素时不会再触发
- mouseleave：从某个元素出来时触发，但不包括子元素
- mousemove：当鼠标在元素范围内移动内，就会不停的重复地触发
- mouseout：从当前元素出来时触发，进出子元素也会触发(因为进出子元素可以看作是从当前元素出来了)
- mouseover：进入元素时触发，进出子元素时也会触发(因为进出子元素可以看作是进入了新的元素)
- mouseup：在用户释放鼠标按钮时触发

上面是对这9个鼠标事件的一个简单介绍，下面将针对这9种鼠标事件做具体的演示：

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        div{
            width: 100px;
            height: 100px;
            border: 1px solid;
            float: left;
            text-align: center;
        }
        .son{
            width: 30px;
            height: 30px;
            border: 1px solid red;
        }
    </style>
```

```
</head>
<body>
    <!-- 设置9个div，其中4,5,7,8需要设置子元素 -->
     <div id="one">1.click</div>
     <div id="two">2.dblclick</div>
     <div id="three">3.mousedown</div>
     <div id="four">4.mouseenter<div class="son"></div></div>
     <div id="five">5.mouseleave<div class="son"></div></div>
     <div id="six">6.mousemove</div>
     <div id="seven">7.mouseout<div class="son"></div></div>
     <div id="eight">8.mouseover<div class="son"></div></div>
     <div id="nine">9.mouseup</div>
     <script>
         //事件处理程序
         let test = function (){
             console.log("你已经触发了事件");
         }
         //9种不同的事件
         one.addEventListener("click",test);
         two.addEventListener("dblclick",test);
         three.addEventListener("mousedown",test);
         four.addEventListener("mouseenter",test);
         five.addEventListener("mouseleave",test);
         six.addEventListener("mousemove",test);
         seven.addEventListener("mouseout",test);
         eight.addEventListener("mouseover",test);
         nine.addEventListener("mouseup",test);
     </script>
</body>
```

效果：

需要注意的就是`mouseleave`与`mouseout`，还有就是`mouseenter`和`mouseover`的区别。
当我们触发鼠标事件时，会有一些对应的事件对象属性被填入event对象里面，分别是：

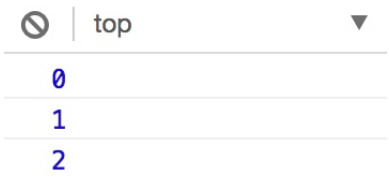
- 坐标属性(clientX和clientY)：当前鼠标的坐标位置
- type属性：触发的是哪一种鼠标事件
- target(DOM)或srcElement(IE)属性：触发鼠标事件的元素节点是哪一个
- button属性

这里要单独说一下这个`button`属性。该属性是在`mousedown`和`mouseup`事件触发时，会存在的属性。表示按下或者释放的按钮。DOM中的`button`属性可以有如下的3个值：0表示按下的

是鼠标左键，1表示按下的是鼠标中键，2表示按下的是鼠标右键。示例如下：

```
<body>
    <p id="test">Lorem ipsum dolor sit amet.</p>
    <script>
        test.onmousedown = function(event){
            console.log(event.button);
        }
    </script>
</body>
```

效果：按下鼠标不同的键位显示出不同的数字



10-4-2 键盘事件

键盘事件有3个，分别为 `keydown` ， `keypress` 和 `keyup`

- keydown：按下键盘上任意键时触发，而且如果按住不放的话，会重复触发此事件。
- keypress：按下键盘上一个字符键(不包括shift和alt键)时触发。mac上按住不放不会重复触发。
- keyup：当用户释放键盘上的键时触发。

同样的，键盘事件也会有属性被传入event对象，会传入event对象的属性如下：

- keyCode属性
- key属性
- target(DOM)或srcElement(IE)属性

`keyCode` 属性会发生在 `keydown` 和 `keyup` 事件被触发时，每一个 `keyCode` 的值与键盘上一个特定的键对应。该属性的值与ASCII码中对应的编码相同。

```
<body>
    <input type="text" id="test">
    <script>
        let test = document.getElementById("test");
        test.addEventListener("keypress",function(){
            console.log("你已经触发了事件");
            console.log(event.keyCode);
        },false);
```

```
    </script>
</body>
```

效果：按下一个键，就会有对应的keyCode被打印到控制台

```
aA
```

```
你已经触发了事件
97
你已经触发了事件
65
```

而我们在做游戏时，常用的四个方向按键，左上右下(顺时针)的键码分别是37、38、39、40

`key`属性是为了取代keyCode而新增的，它的值是一个字符串。在按下某个字符键时，`key`的值就是相应的文本字符。在按下非字符键时，key的值是相应键的名，默认为空字符串。

> 注意：IE8以下浏览器不支持，而safari浏览器不支持`keypress`事件中的`key`属性

```
<body>
    <input type="text" id="test">
    <script>
        let test = document.getElementById("test");
        test.addEventListener("keydown",function(){
            console.log("你已经触发了事件");
            console.log(event.key);
        },false);
    </script>
</body>
```

效果：会打印出用户按下的哪一个键

```
aA
```

```
你已经触发了事件
a
你已经触发了事件
Shift
你已经触发了事件
A
你已经触发了事件
Enter
你已经触发了事件
Shift
你已经触发了事件
Meta
```

10-4-5 页面事件

页面事件主要是指当我们对整个HTML页面作相应的操作时会触发的事件。常见的有页面加载，页面写在卸载以及页面滚动等。

1.页面加载

页面事件中最常用的一个事件就是`load`事件。当页面完全加载后(包括所有图像，JavaScript文件，CSS外部资源)，就会触发window上面的`load`事件，最常见的写法如下：

```
<body>
    <p>在看到我之前，应该有一个弹出框</p>
    <script>
        window.onload = function(){
            alert("你正在加载一个页面！");
        }
    </script>
</body>
```

效果：首先页面完全加载后，触发`load`事件，出现弹出框。然后页面才被加载出来。

当然该事件也可以用于某个单独的元素上面，示例如下：

```
<body>
    <p>在看到我之前，应该有一个弹出框</p>
    <img src="./1.jpg" alt="" onload="alert('Image loaded')">
</body>
```

效果：上面的代码表示，当图像加载完毕以后就会显示一个警告框。

2.页面卸载

与`load`事件对应的就是这个`unload`事件，这个事件是在文档完全被卸载后触发。只要用户从一个页面切换到另一个页面，或者关闭浏览器，就会触发这个事件。示例如下：

```
<body onunload="alert('unloaded')">
    <p>Lorem ipsum dolor sit amet.</p>
</body>
```

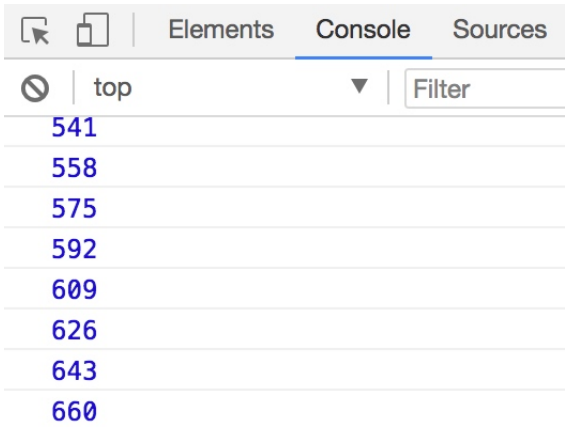
> 注意：现代浏览器规定当触发`unload`事件之后，`alert()`，`confim()`，`prompt()`等模态对话框不再允许弹出，所以这里无法显示出其效果。

3.页面滚动

对页面进行滚动时，对应的`scroll`滚动事件会被触发。通过`<body>`元素的`scrollLeft`和`scrollTop`可以监控到这一变化。`scrollLeft`会监控横向滚动条的变化，`scrollTop`会监控垂直方向的滚动条变化。示例如下：

```
<body style="height:5000px;" id="body">
    <p>Lorem ipsum dolor sit amet.</p>
    <script>
        body.onscroll = function(){
            console.log(document.documentElement.scrollTop);
        }
    </script>
</body>
```

效果：滚动页面时在控制台会打印出当前滚动了的高度。



4.窗口设置大小

当浏览器窗口被调整到一个新的高度或者宽度时，就会触发`resize`事件。示例如下：

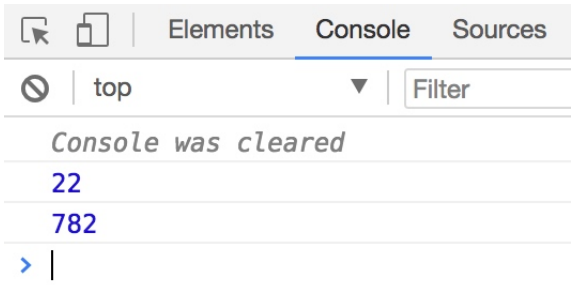
```
<body>
    <p>Lorem ipsum dolor sit amet.</p>
    <script>
        window.onresize = function(){
            console.log("窗口大小被重置了");
        }
    </script>
</body>
```

效果：

我们可以通过 `document.body.clientHeight` 以及 `document.body.clientWidth` 来获取 `body` 元素的宽高，配合 `resize` 事件，可以看到这两个属性实时变化的一个效果。

```
<body>
    <p>Lorem ipsum dolor sit amet.</p>
    <script>
        window.onresize = function(){
            console.clear();//清空控制台信息
            console.log(document.body.clientHeight);
            console.log(document.body.clientWidth);
        }
    </script>
</body>
```

效果：



10-4-6 辅助按键

按下键盘的 `shift` ， `Ctrl` ， `Alt` 和 `Meta` (Mac上的 `command`)等辅助按键，会触发 `keydown` 和 `keyup` 事件，但是不会触发 `keypress` 事件，因为它们不会在屏幕上产生任何字符。这里大家可以通过下面两段代码自行进行对比，如下：

```
<body>
    <script>
        window.onkeyup = function(){
            console.log("OK");
        }
    </script>
</body>
```

效果：此时按下 `shift` ， `Ctrl` ， `Alt` 和 `Meta` (Mac上的 `command`)等辅助按键时将会触发事件。

```
<body>
    <script>
        window.onkeypress = function(){
```

```
            console.log("OK");
        }
    </script>
</body>
```

效果：此时按下 `shift` ， `Ctrl` ， `Alt` 和 `Meta` (Mac上的 `command`)等辅助按键时不会触发事件。

事件对象还有 `shiftKey` ， `ctrlKey` ， `altKey` 和 `metaKey` 属性，可以返回在键盘事件发生时，是否按下了对应的辅助按键。如果这几个属性返回 `true` ，就表示对应的键被按下了。例如：如下的代码会检测在用户按下 `c` 键的同时，是否按下了 `Ctrl` 键

```
<body>
    <script>
        window.onkeydown = function(){
            if((event.key === 'c') && event.ctrlKey)
            {
                console.log("Yes,you pressed ctrl and c");
            }
        }
    </script>
</body>
```

使用下面的代码可以检测在鼠标单击时，是否按下了 `Shift` 键，如下：

```
<body>
    <script>
        window.onclick = function(){
            if(event.shiftKey)
            {
                console.log("Yes,you hold shiftKey and click the mouse");
            }
        }
    </script>
</body>
```

10-4-7 剪贴板事件

剪贴板操作看起来不起眼，但却十分有用，可以增强用户体验，方便用户操作。剪贴板操作包括剪切(cut)、复制(copy)和粘贴(paste)这三个操作，快捷键分别是 `ctrl+x` 、 `ctrl+c` 、 `ctrl+v` 。当然也可以使用鼠标右键菜单进行操作。

关于这3个操作对应下列剪贴板事件：

- copy：在发生复制操作时触发
- cut：在发生剪切操作时触发
- paste：在发生粘贴操作时触发

> 注意：IE浏览器只有在文本中选定字符时，copy和cut事件才会发生。且在非文本框中(如div元素)只能发生copy事件，firfox浏览器只有焦点在文本框中才会发生paste事件。

```
<body>
    <input value="text" id="test">
    <script>
        test.onpaste = test.oncopy = test.oncut = function (e) {
            e = e || event;
            test.value = e.type;
            return false;
        }
    </script>
</body>
```

效果：针对文本框里面的文本内容做剪切复制或者粘贴时，文本框的内容会产生变化，显示出用户是做的什么操作。

text

复制时：

copy

剪切时：

cut

粘贴时：

paste

- beforecopy：在发生复制操作前触发
- beforecut：在发生剪切操作前触发
- beforepaste：在发生粘贴操作前触发

示例代码如下：

```
<body>
    <input value="text" id="test">
    <script>
        test.onbeforepaste = test.onbeforecopy = test.onbeforecut = function
 (e) {
```

```
            e = e || event;
            test.value = e.type;
            return false;
        }
    </script>
</body>
```

对象方法

剪贴板中的数据存储在`clipboardData`对象中。对于IE浏览器来说，这个对象是`window`对象的属性。对于其他浏览器来说，这个对象是事件对象的属性。可以向下面这样书写，来兼容两种不同的浏览器：

```
e = e || event;
let clipboardData = e.clipboardData || window.clipboardData;
```

这个对象有三个方法：`getData()`、`setData()`和`clearData()`。下面依次对这3个方法进行介绍。

getData()

`getData()`方法用于从剪贴板中取得数据，它接受一个参数，即要取得的数据的格式。在IE中，有两种数据格式："text"和"URL"。在其他浏览器中，这个参数是一种MIME类型；不过，可以用"text"代表

> 注意在IE浏览器中，`cut`和`copy`事件中的`getData()`方法始终返回`null`，而其他浏览器始终返回空字符串''。但如果和`setDada()`方法配合，就可以正常使用

示例如下：

```
<body>
    <input id="test" value="123">
    <script>
        test.onpaste = function (e) {
            e = e || event;
            let clipboardData = e.clipboardData || window.clipboardData;
            test.value = '测试' + clipboardData.getData('text');
            return false;
        }
    </script>
</body>
```

效果：复制文本框中的内容，然后粘贴。粘贴时前面会添加有"测试"的内容

setData()

`setData()`方法的第一个参数也是数据类型，第二个参数是要放在剪贴板中的文本。对于第一个参数的规则与`getData()`相同。

> 注意：在IE浏览器中，该方法在成功将文本放到剪贴板中后，返回true，否则返回false；而其他浏览器中，该方法无返回值。在paste事件中，只有IE浏览器可以正常使用setData()方法，chrome浏览器会静默失败，而firefox浏览器会报错。

示例如下：

```
<body>
    <input id="test" value="123">
    <script>
        test.oncopy = function (e) {
            e = e || event;
            var clipboardData = e.clipboardData || window.clipboardData;
            clipboardData.setData('text', '测试');
            test.value = clipboardData.getData('text');
            return false;
        }
    </script>
</body>
```

clearData()

`clearData()`方法用于从剪贴板中删除数据，它接受一个参数，即要取得的数据的格式。在IE中，有两种数据格式："text"和"URL"。在其他浏览器中，这个参数是一种MIME类型；不过，可以用"text"表示

> 注意：在IE浏览器中，该方法在成功将文本放到剪贴板中后，返回true，否则返回false；而其他浏览器该方法的返回值为undefined。在paste事件中，chrome浏览器和IE浏览器可以正常使用setData()方法，而firefox浏览器会报错

示例如下：

```
<body>
    <input id="test" value="123">
    <script>
        test.oncopy = function (e) {
            e = e || event;
            var clipboardData = e.clipboardData || window.clipboardData;
```

```
            test.value = clipboardData.clearData('text');
            return false;
        }
    </script>
</body>
```

实际应用

1.屏蔽剪贴板

通过阻止默认行为来屏蔽剪贴板。对于一些受保护的文档来说是一种选择，示例代码如下：

```
<body>
    <input value="text">
    <button id="test">屏蔽剪贴板</button>
    <script>
        test.onclick = function () {
            document.oncopy = document.oncut = document.onpaste = function (
e) {
                e = e || event;
                alert('该文档不允许复制剪贴操作，谢谢配合')
                return false;
            }
        }
    </script>
</body>
```

2.过滤字符

如果确保粘贴到文本框中的文本中包含某些字符，或者符合某种形式时，可以使用剪贴板事件。比如只允许粘贴数字。

```
<body>
    <input id="test">
    <script>
        test.onpaste = function (e) {
            e = e || event;
            var clipboardData = e.clipboardData || window.clipboardData;
            if (!/^\d+$/.test(clipboardData.getData('text'))) {
                return false;
            }
        }
    </script>
</body>
```

10-5 事件模拟

事件是网页中某个特别的瞬间，经常由用户操作来触发。但实际上，也可以使用JavaScript在任意时刻来触发特定的事件，而此时的事件就如同用户操作来触发的事件一样。这种通过代码来模拟事件触发的方式我们称之为事件的模拟。

10-5-1 click()方法

使用`click()`方法，我们可以模拟用户的点击行为，示例如下：

```
<body>
    <button id="btn">点击我</button>
    <script>
        btn.onclick = function(){
            alert("You have clicked!");
        }
        btn.click();
    </script>
</body>
```

效果：页面加载出来后就会触发点击事件，弹出对话框。

10-5-2 事件模拟机制

虽然上面介绍的`click()`方法完美的模拟了`click`事件，但是对于其他事件并没有相应的模拟方法，所以这个时候就需要使用到事件模拟了。

事件模拟包括3个部分：**创建事件**、**初始化**以及**触发事件**。某些情况下，初始化与创建事件一起进行。最终，通过`dispatchEvent()`方法来触发事件。
对于不同的事件类型，有不同的创建方法。下面以mouseover事件为例。

MouseEvent()

使用`MouseEvent()`方法可以创建鼠标事件。实际上，`MouseEvent()`方法在创建事件的同时，也包括了初始化的操作。

> 注意：IE浏览器和safari浏览器不支持

最后使用`dispatchEvent()`方法在当前节点上触发指定事件。该方法返回一个布尔值，只要有一个监听函数调用了`Event.preventDefault()`，则返回值为false，否则为true。

> 注意：IE8以下浏览器不支持。

核心代码如下：

```
function simulateMouseOver(obj) {
  let event = new MouseEvent('mouseover', {
    'bubbles': true,
    'cancelable': true
  });
  obj.dispatchEvent(event);
}
```

接下来我们来看一个具体的示例：

```
<body>
    <button id="btn1">按钮一</button>
    <button id="btn2">按钮二</button>
    <script>
        btn1.addEventListener('mouseover', function () { alert(1); });
        function simulateMouseOver(obj) {
            let event = new MouseEvent('mouseover', {
                'bubbles': true,
                'cancelable': true
            });
            obj.dispatchEvent(event);
        }
        btn2.onmouseover = function () {
            simulateMouseOver(btn1);
        }
    </script>
</body>
```

createEvent()

除了上面介绍的`MouseEvent()`用于创建鼠标事件以外，还有诸如`UIEvents`，`KeyboardEvent`等构造函数可以创建对应的事件类型。但是，在document对象上使用`createEvent()`方法可以用来创建event对象。这个方法接收一个参数，表示要创建的事件类型的字符串。换句话说，使用`createEvent()`这一个方法就可以创建任意类型的事件。

> 注意：IE8-浏览器不支持`createEvent()`方法。

在使用`document.createElement`创建事件之后，还需要使用与事件有关的信息对其进行初始化，每种不同类型的事件都有不同的初始化方法。

```
事件类型                    事件初始化方法
UIEvents                    event.initUIEvent
MouseEvents                 event.initMouseEvent
MutationEvents              event.initMutationEvent
HTMLEvents                  event.initEvent
Event                       event.initEvent
CustomEvent                 event.initCustomEvent
KeyboardEvent               event.initKeyEvent
```

这里我们以创建`MouseEvent()`为例，使用`createEvent()`方法来创建上面介绍过的`MouseEvent()`事件。首先介绍`initMouseEvent()`方法的参数，它们与鼠标事件的event对象所包含的属性一一对应。其中，前4个参数对正确地激发事件至关重要，因为浏览器要用到这些参数。而剩下的所有参数只有在事件处理程序中才会用到。

> 注意：IE8-浏览器不支持initMouseEvent()方法

- type(字符串):表示要触发的事件类型，例如"click"
- bubbles(布尔值):表示事件是否应该冒泡。为精确地模拟鼠标事件，应该把这个参数设置为true
- cancelable(布尔值):表示事件是否可以取消。为精确地模拟鼠标事件，应该把这个参数设置为crue
- view(AbstractView):与事件关联的视图。这个参数几乎总是要设置为document.defaultView
- detail(整数):与事件有关的详细信息。这个值一般只有事件处理程序使用，但通常都设置为0
- screenx(整数):事件相对于屏幕的X坐标
- screenY(整数):事件相对于屏幕的Y坐标
- clientX(整数):事件相对于视口的X坐标
- clientY(整数):事件相对于视口的Y坐标
- ctrlKey(布尔值):表示是否按下Ctrl键。默认值为false
- altkey(布尔值):表示是否按下了Alt键。默认值为false
- shiftKey(布尔值):表示是否按下了Shift键。默认值为false
- metaKey(布尔值):表示是否按下了Meta键。默认值为false
- button(整数):表示按下了哪一个鼠标键。默认值为0
- relatedTarget(对象):表示与事件相关的对象。这个参数只在模拟mouseover或mouseout时使用

接下来我们来演示使用`createEvent()`的方法来创建`MouseEvent()`事件，核心代码如下：

```
//创建自定义事件
function simulateMouseOver(obj) {
    let event = document.createEvent('MouseEvents');
    event.initMouseEvent(
```

```
            'mouseover',
            true,
            true,
            document.defaultView,
            0,
            0,
            0,
            0,
            0,
            false,
            false,
            false,
            false,
            0,
            null
        );
        //触发事件
        obj.dispatchEvent(event);
    }
```

完整的示例代码如下：

```
<body>
    <button id="btn1">按钮一</button>
    <button id="btn2">按钮二</button>
    <script>
        //给按钮一绑定事件
        btn1.addEventListener('mouseover', function () { alert(1); })
        //创建自定义事件
        function simulateMouseOver(obj) {
            var event = document.createEvent('MouseEvents');
            event.initMouseEvent(
                'mouseover',
                true,
                true,
                document.defaultView,
                0,
                0,
                0,
                0,
                0,
                false,
                false,
                false,
                false,
                0,
                null
```

```
            );
            //触发事件
            obj.dispatchEvent(event);
        }
        //给按钮二绑定事件
        btn2.onmouseover = function () {
            simulateMouseOver(btn1);
        }
    </script>
</body>
```

10-5-3 自定义事件

自定义事件不是由DOM原生触发的，它的目的是让开发人员创建自己的事件。

Event()

最简单的自定义事件的方式就是使用`Event()`构造函数，示例如下：

> 注意：IE和safari浏览器不支持

```
<body>
    <button id="btn">按钮</button>
    <script>
        function customEvent(obj) {
            let event = new Event('changeColor');
            obj.addEventListener('changeColor', function () {
                this.style.backgroundColor = 'pink';
            })
            return event;
        }
        btn.onclick = function () {
            this.dispatchEvent(customEvent(this));
        }
    </script>
</body>
```

效果：点击按钮会会触发`changeColor`事件，按钮的背景颜色变为粉色
点击前：

点击后：

CustomEvent()

如果需要在触发事件的同时，传入指定的数据，需要使用CustomEvent构造函数生成自定义的事件对象。示例如下：

```
<body>
    <button id="btn">按钮</button>
    <script>
        function customEvent(obj) {
            let event = new CustomEvent('changeColor', { 'detail': 'hello' }
);
            obj.addEventListener('changeColor', function (e) {
                e = e || event;
                this.style.backgroundColor = 'lightblue';
                this.innerHTML = e.detail;
            })
            return event;
        }
        btn.onclick = function () {
            this.dispatchEvent(customEvent(this));
        }
    </script>
</body>
```

效果：这里新声明了一个"changeColor"的事件，并为事件对象添加了属性"detail"
点击前：

点击后：

createEvent()

除了前面的`Event()`构造函数来创建自定义事件以外，我们还可以调用`createEvent("CustomEvent")`来创建新的自定义事件。返回的对象有一个名为`initCustomEvent()`的方法，接收如下4个参数：

- type(字符串)：触发的事件类型，例如"keydown"
- bubbles(布尔值)：表示事件是否应该冒泡
- cancelable(布尔值)：表示事件是否可以取消
- detail(对象)：任意值，保存在event对象的detail属性中

> 注意：IE8-浏览器不支持

示例代码如下：

```
<body>
    <button id="btn">按钮</button>
    <script>
        function customEvent(obj) {
            let event = document.createEvent('CustomEvent');
            event.initCustomEvent('changeColor', true, true, 'hello');
            obj.addEventListener('changeColor', function (e) {
                e = e || event;
                this.style.backgroundColor = 'lightblue';
                this.innerHTML = e.detail;
            })
            return event;
        }
        btn.onclick = function () {
            this.dispatchEvent(customEvent(this));
        }
    </script>
</body>
```

效果：和前面的示例一样，成功自定义了一个事件，并且拥有事件对象属性
点击前：

点击后：

最后需要说明一下，事件模拟主要是用来触发自定义的事件函数，而不是来触发浏览器默认行为的。所以，试图通过事件模拟的形式来触发浏览器默认行为是不可行的。比如点击鼠标右键实现键盘 `backspace` 键的删除效果是不可行的。

总结

1. JavaScript和HTML之间的交互是通过事件来实现的。
2. 事件流分为事件冒泡流以及事件捕获流。IE实现的是冒泡流，网景公司实现的是捕获流，DOM标准则是采用的是冒泡加捕获的方式。
3. 事件代理是利用了事件冒泡的特性，只指定一个事件处理程序，就可以管理某一类型的所有事件。
4. 根据业务的不同，有些时候我们需要组织事件冒泡。
5. 事件监听器又被称之为事件处理程序。给一个事件添加事件监听器有不止一种方式来实现。
6. 事件对象是事件监听器在调用回调函数时自动传入的对象。该对象包含了当前触发的事件的

相关信息。

7. 事件可以分为各种各样的类型。不同的事件类型，填入事件对象的属性也有所不同。
8. 除了用户操作来实现事件的触发以外，我们还可以通过一些方式来模拟事件的触发。
9. 事件模拟主要是用来触发自定义的事件函数，而不是来触发浏览器默认的行为。