

第 13 章

事件

本章内容

- 理解事件流
- 使用事件处理程序
- 不同的事件类型

JavaScript 与 HTML 之间的交互是通过事件实现的。事件，就是文档或浏览器窗口中发生的一些特定的交互瞬间。可以使用侦听器（或处理程序）来预订事件，以便事件发生时执行相应的代码。这种在传统软件工程中被称为观察员模式的模型，支持页面的行为（JavaScript 代码）与页面的外观（HTML 和 CSS 代码）之间的松散耦合。

事件最早是在 IE3 和 Netscape Navigator 2 中出现的，当时是作为分担服务器运算负载的一种手段。在 IE4 和 Navigator 4 发布时，这两种浏览器都提供了相似但不相同的 API，这些 API 并存经过了好几个主要版本。DOM2 级规范开始尝试以一种符合逻辑的方式来标准化 DOM 事件。IE9、Firefox、Opera、Safari 和 Chrome 全都已经实现了“DOM2 级事件”模块的核心部分。IE8 是最后一个仍然使用其专有事件系统的主要浏览器。

浏览器的事件系统相对比较复杂。尽管所有主要浏览器已经实现了“DOM2 级事件”，但这个规范本身并没有涵盖所有事件类型。浏览器对象模型（BOM）也支持一些事件，这些事件与文档对象模型（DOM）事件之间的关系并不十分清晰，因为 BOM 事件长期没有规范可以遵循（HTML5 后来给出了详细的说明）。随着 DOM3 级的出现，增强后的 DOM 事件 API 变得更加繁琐。使用事件有时相对简单，有时则非常复杂，难易程度会因你的需求而不同。不过，有关事件的一些核心概念是一定要理解的。

13.1 事件流

当浏览器发展到第四代时（IE4 及 Netscape Communicator 4），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？要明白这个问题问的是什麼，可以想象画在一张纸上的一组同心圆。如果你把手指放在圆心上，那么你的手指指向的不是一个圆，而是纸上的所有圆。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。如果你单击了某个按钮，他们都认为单击事件不仅仅发生在按钮上。换句话说，在单击按钮的同时，你也单击了按钮的容器元素，甚至也单击了整个页面。

事件流描述的是从页面中接收事件的顺序。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 Communicator 的事件流是事件捕获流。

13.1.1 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)。以下面的 HTML 页面为例：

```
<!DOCTYPE html>
<html>
<head>
  <title>Event Bubbling Example</title>
</head>
<body>
  <div id="myDiv">Click Me</div>
</body>
</html>
```

如果你单击了页面中的 <div> 元素，那么这个 click 事件会按照如下顺序传播：

- (1) <div>
- (2) <body>
- (3) <html>
- (4) document

也就是说，click 事件首先在 <div> 元素上发生，而这个元素就是我们单击的元素。然后，click 事件沿 DOM 树向上传播，在每一级节点上都会发生，直至传播到 document 对象。图 13-1 展示了事件冒泡的过程。

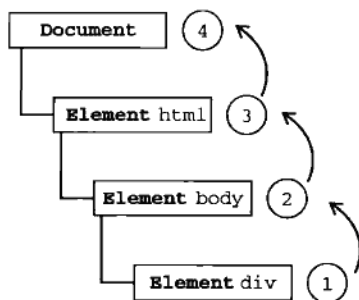


图 13-1

所有现代浏览器都支持事件冒泡，但在具体实现上还是有一些差别。IE5.5 及更早版本中的事件冒泡会跳过 <html> 元素 (从 <body> 直接跳到 document)。IE9、Firefox、Chrome 和 Safari 则将事件一直冒泡到 window 对象。

13.1.2 事件捕获

Netscape Communicator 团队提出的另一种事件流叫做事件捕获 (event capturing)。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捕获它。如果仍以前面的 HTML 页面作为演示事件捕获的例子，那么单击 <div> 元素就会以下列顺序触发 click 事件。

- (1) document

- (2) <html>
- (3) <body>
- (4) <div>

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。图 13-2 展示了事件捕获的过程。

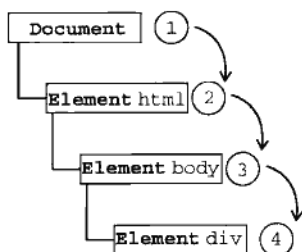


图 13-2

虽然事件捕获是 Netscape Communicator 唯一支持的事件流模型，但 IE9、Safari、Chrome、Opera 和 Firefox 目前也都支持这种事件流模型。尽管“DOM2 级事件”规范要求事件应该从 document 对象开始传播，但这些浏览器都是从 window 对象开始捕获事件的。

由于老版本的浏览器不支持，因此很少有人使用事件捕获。我们也建议读者放心地使用事件冒泡，在有特殊需要时再使用事件捕获。

13.1.3 DOM 事件流

“DOM2 级事件”规定的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。首先发生的是事件捕获，为截获事件提供了机会。然后是实际的目标接收到事件。最后一个阶段是冒泡阶段，可以在这个阶段对事件做出响应。以前面简单的 HTML 页面为例，单击<div>元素会按照图13-3所示顺序触发事件。

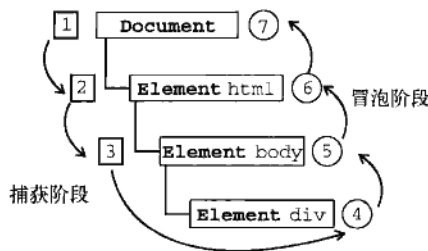


图 13-3

在 DOM 事件流中，实际的目标（<div>元素）在捕获阶段不会接收到事件。这意味着在捕获阶段，事件从 document 到<html>再到<body>后就停止了。下一个阶段是“处于目标”阶段，于是事件在<div>上发生，并在事件处理（后面将会讨论这个概念）中被看成冒泡阶段的一部分。然后，冒泡阶段发生，事件又传播回文档。

多数支持 DOM 事件流的浏览器都实现了一种特定的行为；即使“DOM2 级事件”规范明确要求捕获阶段不会涉及事件目标，但 IE9、Safari、Chrome、Firefox 和 Opera 9.5 及更高版本都会在捕获阶段触发事件对象上的事件。结果，就是有两个机会在目标对象上面操作事件。



IE9、Opera、Firefox、Chrome 和 Safari 都支持 DOM 事件流；IE8 及更早版本不支持 DOM 事件流。

13.2 事件处理程序

事件就是用户或浏览器自身执行的某种动作。诸如 click、load 和 mouseover，都是事件的名字。而响应某个事件的函数就叫做事件处理程序（或事件侦听器）。事件处理程序的名字以“on”开头，因此 click 事件的事件处理程序就是 onclick，load 事件的事件处理程序就是 onload。为事件指定处理程序的方式有好几种。

13.2.1 HTML 事件处理程序

某个元素支持的每种事件，都可以使用一个与相应事件处理程序同名的 HTML 特性来指定。这个特性的值应该是能够执行的 JavaScript 代码。例如，要在按钮被单击时执行一些 JavaScript，可以像下面这样编写代码：

```
<input type="button" value="Click Me" onclick="alert('Clicked')" />
```

当单击这个按钮时，就会显示一个警告框。这个操作是通过指定 onclick 特性并将一些 JavaScript 代码作为它的值来定义的。由于这个值是 JavaScript，因此不能在其中使用未经转义的 HTML 语法字符，例如和号（&）、双引号（"）、小于号（<）或大于号（>）。为了避免使用 HTML 实体，这里使用了单引号。如果想要使用双引号，那么就要将代码改写成如下所示：

```
<input type="button" value="Click Me" onclick="alert(&quot;Clicked&quot;);" />
```

在 HTML 中定义的事件处理程序可以包含要执行的具体动作，也可以调用在页面其他地方定义的本脚本，如下面的例子所示：

```
<script type="text/javascript">
    function showMessage(){
        alert("Hello world!");
    }
</script>
<input type="button" value="Click Me" onclick="showMessage()" />
```

HTMLEventHandlerExample01.htm

在这个例子中，单击按钮就会调用 showMessage() 函数。这个函数是在一个独立的<script>元素中定义的，当然也可以被包含在一个外部文件中。事件处理程序中的代码在执行时，有权访问全局作用域中的任何代码。

这样指定事件处理程序具有一些独到之处。首先，这样会创建一个封装着元素属性值的函数。这个函数中有一个局部变量 event，也就是事件对象（本章稍后讨论）：

```
<!-- 输出 "click" -->
<input type="button" value="Click Me" onclick="alert(event.type)">
```

通过 event 变量，可以直接访问事件对象，你不用自己定义它，也不用从函数的参数列表中读取。在这个函数内部，this 值等于事件的目标元素，例如：

```
<!-- 输出 "Click Me" -->
<input type="button" value="Click Me" onclick="alert(this.value)">
```

关于这个动态创建的函数，另一个有意思的地方是它扩展作用域的方式。在这个函数内部，可以像访问局部变量一样访问 document 及该元素本身的成员。这个函数使用 with 像下面这样扩展作用域：

```
function(){
    with(document){
        with(this){
            //元素属性值
        }
    }
}
```

如此一来，事件处理程序要访问自己的属性就简单多了。下面这行代码与前面的例子效果相同：

```
<!-- 输出 "Click Me" -->
<input type="button" value="Click Me" onclick="alert(value)">
```

如果当前元素是一个表单输入元素，则作用域中还会包含访问表单元素（父元素）的入口，这个函数就变成了如下所示：

```
function(){
    with(document){
        with(this.form){
            with(this){
                //元素属性值
            }
        }
    }
}
```

实际上，这样扩展作用域的方式，无非就是想让事件处理程序无需引用表单元素就能访问其他表单字段。例如：

```
<form method="post">
    <input type="text" name="username" value="">
    <input type="button" value="Echo Username" onclick="alert(username.value)">
</form>
```

[HTMLEventHandlerExample04.htm](#)

在这个例子中，单击按钮会显示文本框中的文本。值得注意的是，这里直接引用了 username 元素。不过，在 HTML 中指定事件处理程序有两个缺点。首先，存在一个时差问题。因为用户可能会在 HTML 元素一出现在页面上就触发相应的事件，但当时的事件处理程序有可能尚不具备执行条件。以前

面的例子来说明，假设 `showMessage()` 函数是在按钮下方、页面的最底部定义的。如果用户在页面解析 `showMessage()` 函数之前就单击了按钮，就会引发错误。为此，很多 HTML 事件处理程序都会被封装在一个 `try-catch` 块中，以便错误不会浮出水面，如下面的例子所示：

```
<input type="button" value="Click Me" onclick="try{showMessage();}catch(ex){}">
```

这样，如果在 `showMessage()` 函数有定义之前单击了按钮，用户将不会看到 JavaScript 错误，因为在浏览器有机会处理错误之前，错误就被捕获了。

另一个缺点是，这样扩展事件处理程序的作用域链在不同浏览器中会导致不同结果。不同 JavaScript 引擎遵循的标识符解析规则略有差异，很可能在访问非限定对象成员时出错。

通过 HTML 指定事件处理程序的最后一个缺点是 HTML 与 JavaScript 代码紧密耦合。如果要更换事件处理程序，就要改动两个地方：HTML 代码和 JavaScript 代码。而这正是许多开发人员摒弃 HTML 事件处理程序，转而使用 JavaScript 指定事件处理程序的原因所在。



要了解关于 HTML 事件处理程序缺点的更多信息，请参考 Garrett Smith 的文章“Event Handler Scope”（www.jibbering.com/faq/names/event_handler.html）。

13.2.2 DOM0 级事件处理程序

通过 JavaScript 指定事件处理程序的传统方式，就是将一个函数赋值给一个事件处理程序属性。这种为事件处理程序赋值的方法是在第四代 Web 浏览器中出现的，而且至今仍然为所有现代浏览器所支持。原因一是简单，二是具有跨浏览器的优势。要使用 JavaScript 指定事件处理程序，首先必须取得一个要操作的对象的引用。

每个元素（包括 `window` 和 `document`）都有自己的事件处理程序属性，这些属性通常全部小写，例如 `onclick`。将这种属性的值设置为一个函数，就可以指定事件处理程序，如下所示：

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert("Clicked");
};
```

在此，我们通过文档对象取得了一个按钮的引用，然后为它指定了 `onclick` 事件处理程序。但要注意，在这些代码运行以前不会指定事件处理程序，因此如果这些代码在页面中位于按钮后面，就有可能在一段时间内怎么单击都没有反应。

使用 DOM0 级方法指定的事件处理程序被认为是元素的方法。因此，这时候的事件处理程序是在元素的作用域中运行；换句话说，程序中的 `this` 引用当前元素。来看一个例子。

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert(this.id);    //"myBtn"
};
```

DOMLevel0EventHandlerExample01.htm

单击按钮显示的是元素的 ID，这个 ID 是通过 `this.id` 取得的。不仅仅是 ID，实际上可以在事件处理程序中通过 `this` 访问元素的任何属性和方法。以这种方式添加的事件处理程序会在事件流的冒泡

阶段被处理。

也可以删除通过 DOM0 级方法指定的事件处理程序，只要像下面这样将事件处理程序属性的值设置为 null 即可：

```
btn.onclick = null; //删除事件处理程序
```

将事件处理程序设置为 null 之后，再单击按钮将不会有任何动作发生。



如果你使用 HTML 指定事件处理程序，那么 onclick 属性的值就是一个包含着在同名 HTML 特性中指定的代码的函数。而将相应的属性设置为 null，也可以删除以这种方式指定的事件处理程序。

13.2.3 DOM2 级事件处理程序

“DOM2 级事件”定义了两个方法，用于处理指定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：要处理的事件名、作为事件处理程序的函数和一个布尔值。最后这个布尔值参数如果是 true，表示在捕获阶段调用事件处理程序；如果是 false，表示在冒泡阶段调用事件处理程序。

要在按钮上为 click 事件添加事件处理程序，可以使用下列代码：

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);
```

上面的代码为一个按钮添加了 onclick 事件处理程序，而且该事件会在冒泡阶段被触发（因为最后一个参数是 false）。与 DOM0 级方法一样，这里添加的事件处理程序也是在其依附的元素的作用域中运行。使用 DOM2 级方法添加事件处理程序的主要好处是可以添加多个事件处理程序。来看下面的例子。



```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);
btn.addEventListener("click", function(){
    alert("Hello world!");
}, false);
```

DOMLevel2EventHandlerExample01.htm

这里为按钮添加了两个事件处理程序。这两个事件处理程序会按照添加它们的顺序触发，因此首先会显示元素的 ID，其次会显示“Hello world!”消息。

通过 addEventListener() 添加的事件处理程序只能使用 removeEventListener() 来移除；移除时传入的参数与添加处理程序时使用的参数相同。这也意味着通过 addEventListener() 添加的匿名函数将无法移除，如下面的例子所示。

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
});
```

```
}, false);

//这里省略了其他代码

btn.removeEventListener("click", function(){ //没有用!
    alert(this.id);
}, false);
```

在这个例子中, 我们使用 `addEventListener()` 添加了一个事件处理程序。虽然调用 `removeEventListener()` 时看似使用了相同的参数, 但实际上, 第二个参数与传入 `addEventListener()` 中的那一个是完全不同的函数。而传入 `removeEventListener()` 中的事件处理程序函数必须与传入 `addEventListener()` 中的相同, 如下面的例子所示。

```
var btn = document.getElementById("myBtn");
var handler = function(){
    alert(this.id);
};
btn.addEventListener("click", handler, false);

//这里省略了其他代码

btn.removeEventListener("click", handler, false); //有效!
```

DOMLevel2EventHandlerExample01.htm

重写后的这个例子没有问题, 是因为在 `addEventListener()` 和 `removeEventListener()` 中使用了相同的函数。

大多数情况下, 都是将事件处理程序添加到事件流的冒泡阶段, 这样可以最大限度地兼容各种浏览器。最好只在需要在事件到达目标之前截获它的时候将事件处理程序添加到捕获阶段。如果不是特别需要, 我们不建议在事件捕获阶段注册事件处理程序。



IE9、Firefox、Safari、Chrome 和 Opera 支持 DOM2 级事件处理程序。

13.2.4 IE 事件处理程序

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两个参数: 事件处理程序名称与事件处理程序函数。由于 IE8 及更早版本只支持事件冒泡, 所以通过 `attachEvent()` 添加的事件处理程序都会被添加到冒泡阶段。

要使用 `attachEvent()` 为按钮添加一个事件处理程序, 可以使用以下代码。

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert("Clicked");
});
```

IEEventHandlerExample01.htm

注意, `attachEvent()` 的第一个参数是 "onclick", 而非 DOM 的 `addEventListener()` 方法中的 "click"。

在 IE 中使用 `attachEvent()` 与使用 DOM0 级方法的主要区别在于事件处理程序的作用域。在使用 DOM0 级方法的情况下，事件处理程序会在其所属元素的作用域内运行；在使用 `attachEvent()` 方法的情况下，事件处理程序会在全局作用域中运行，因此 `this` 等于 `window`。来看下面的例子。

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert(this === window);    //true
});
```

在编写跨浏览器的代码时，牢记这一区别非常重要。

与 `addEventListener()` 类似，`attachEvent()` 方法也可以用来为一个元素添加多个事件处理程序。来看下面的例子。

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert("Clicked");
});
btn.attachEvent("onclick", function(){
    alert("Hello world!");
});
```

IEEventHandlerExample01.htm

这里调用了两次 `attachEvent()`，为同一个按钮添加了两个不同的事件处理程序。不过，与 DOM 方法不同的是，这些事件处理程序不是以添加它们的顺序执行，而是以相反的顺序被触发。单击这个例子中的按钮，首先看到的是“Hello world!”，然后才是“Clicked”。

使用 `attachEvent()` 添加的事件可以通过 `detachEvent()` 来移除，条件是必须提供相同的参数。与 DOM 方法一样，这也意味着添加的匿名函数将不能被移除。不过，只要能够将对相同函数的引用传给 `detachEvent()`，就可以移除相应的事件处理程序。例如：

```
var btn = document.getElementById("myBtn");
var handler = function(){
    alert("Clicked");
};
btn.attachEvent("onclick", handler);

//这里省略了其他代码

btn.detachEvent("onclick", handler);
```

IEEventHandlerExample02.htm

这个例子将保存在变量 `handler` 中的函数作为事件处理程序。因此，后面的 `detachEvent()` 可以使用相同的函数来移除事件处理程序。



支持 IE 事件处理程序的浏览器有 IE 和 Opera。

13.2.5 跨浏览器的事件处理程序

为了以跨浏览器的方式处理事件，不少开发人员会使用能够隔离浏览器差异的 JavaScript 库，还有一些开发人员会自己开发最合适的事件处理的方法。自己编写代码其实也不难，只要恰当地使用能力检

测即可（能力检测在第9章介绍过）。要保证处理事件的代码能在大多数浏览器下一致地运行，只需关注冒泡阶段。

第一个要创建的方法是 `addHandler()`，它的职责是视情况分别使用 DOM0 级方法、DOM2 级方法或 IE 方法来添加事件。这个方法属于一个名叫 `EventUtil` 的对象，本书将使用这个对象来处理浏览器间的差异。`addHandler()` 方法接受 3 个参数：要操作的元素、事件名称和事件处理程序函数。

与 `addHandler()` 对应的方法是 `removeHandler()`，它也接受相同的参数。这个方法的职责是移除之前添加的事件处理程序——无论该事件处理程序是采取什么方式添加到元素中的，如果其他方法无效，默认采用 DOM0 级方法。

`EventUtil` 的用法如下所示。

```
var EventUtil = {
  addHandler: function(element, type, handler){
    if (element.addEventListener){
      element.addEventListener(type, handler, false);
    } else if (element.attachEvent){
      element.attachEvent("on" + type, handler);
    } else {
      element["on" + type] = handler;
    }
  },
  removeHandler: function(element, type, handler){
    if (element.removeEventListener){
      element.removeEventListener(type, handler, false);
    } else if (element.detachEvent){
      element.detachEvent("on" + type, handler);
    } else {
      element["on" + type] = null;
    }
  }
};
```

EventUtil.js

这两个方法首先都会检测传入的元素中是否存在 DOM2 级方法。如果存在 DOM2 级方法，则使用该方法；传入事件类型、事件处理程序函数和第三个参数 `false`（表示冒泡阶段）。如果存在的是 IE 的方法，则采取第二种方案。注意，为了在 IE8 及更早版本中运行，此时的事件类型必须加上“on”前缀。最后一种可能就是使用 DOM0 级方法（在现代浏览器中，应该不会执行这里的代码）。此时，我们使用的是方括号语法来将属性名指定为事件处理程序，或者将属性设置为 `null`。

可以像下面这样使用 `EventUtil` 对象：

```
var btn = document.getElementById("myBtn");
var handler = function(){
  alert("Clicked");
};
EventUtil.addHandler(btn, "click", handler);

//这里省略了其他代码

EventUtil.removeHandler(btn, "click", handler);
```

CrossBrowserEventHandlerExample01.htm

`addHandler()` 和 `removeHandler()` 没有考虑到所有的浏览器问题, 例如在 IE 中的作用域问题。不过, 使用它们添加和移除事件处理程序还是足够了。此外还要注意, DOM0 级对每个事件只支持一个事件处理程序。好在, 只支持 DOM0 级的浏览器已经没有那么多了, 因此这对你而言应该不是什么问题。

13.3 事件对象

在触发 DOM 上的某个事件时, 会产生一个事件对象 `event`, 这个对象中包含着所有与事件有关的信息。包括导致事件的元素、事件的类型以及其他与特定事件相关的信息。例如, 鼠标操作导致的事件对象中, 会包含鼠标位置的信息, 而键盘操作导致的事件对象中, 会包含与按下的键有关的信息。所有浏览器都支持 `event` 对象, 但支持方式不同。

13.3.1 DOM 中的事件对象

兼容 DOM 的浏览器会将一个 `event` 对象传入到事件处理程序中。无论指定事件处理程序时使用什么方法 (DOM0 级或 DOM2 级), 都会传入 `event` 对象。来看下面的例子。

```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.type);    //"click"
};
btn.addEventListener("click", function(event){
    alert(event.type);    //"click"
}, false);
```

这个例子中的两个事件处理程序都会弹出一个警告框, 显示由 `event.type` 属性表示的事件类型。这个属性始终都会包含被触发的事件类型, 例如 `"click"` (与传入 `addEventListener()` 和 `removeEventListener()` 中的事件类型一致)。

在通过 HTML 特性指定事件处理程序时, 变量 `event` 中保存着 `event` 对象。请看下面的例子。

```
<input type="button" value="Click Me" onclick="alert(event.type)"/>
```


以这种方式提供 `event` 对象, 可以让 HTML 特性事件处理程序与 JavaScript 函数执行相同的操作。

`event` 对象包含与创建它的特定事件有关的属性和方法。触发的事件类型不一样, 可用的属性和方法也不一样。不过, 所有事件都会有下表列出的成员。

属性/方法	类 型	读/写	说 明
<code>bubbles</code>	Boolean	只读	表明事件是否冒泡
<code>cancelable</code>	Boolean	只读	表明是否可以取消事件的默认行为
<code>currentTarget</code>	Element	只读	其事件处理程序当前正在处理事件的那个元素
<code>defaultPrevented</code>	Boolean	只读	为 <code>true</code> 表示已经调用了 <code>preventDefault()</code> (DOM3 级事件中新增)
<code>detail</code>	Integer	只读	与事件相关的细节信息
<code>eventPhase</code>	Integer	只读	调用事件处理程序的阶段: 1 表示捕获阶段, 2 表示“处于目标”, 3 表示冒泡阶段

属性/方法	类 型	读/写	说 明
preventDefault()	Function	只读	取消事件的默认行为。如果cancelable是true, 则可以使用这个方法
stopImmediatePropagation()	Function	只读	取消事件的进一步捕获或冒泡, 同时阻止任何事件处理程序被调用 (DOM3级事件中新增)
stopPropagation()	Function	只读	取消事件的进一步捕获或冒泡。如果bubbles为true, 则可以使用这个方法
target	Element	只读	事件的目标
trusted	Boolean	只读	为true表示事件是浏览器生成的。为false表示事件是由开发人员通过JavaScript创建的 (DOM3级事件中新增)
type	String	只读	被触发的事件的类型
view	AbstractView	只读	与事件关联的抽象视图。等同于发生事件的window对象

在事件处理程序内部, 对象 `this` 始终等于 `currentTarget` 的值, 而 `target` 则只包含事件的实际目标。如果直接将事件处理程序指定给了目标元素, 则 `this`、`currentTarget` 和 `target` 包含相同的值。来看下面的例子。



```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.currentTarget === this);    //true
    alert(event.target === this);          //true
};
```

DOMEventObjectExample01.htm


这个例子检测了 `currentTarget` 和 `target` 与 `this` 的值。由于 `click` 事件的目标是按钮, 因此这三个值是相等的。如果事件处理程序存在于按钮的父节点中 (例如 `document.body`), 那么这些值是不相同的。再看下面的例子。

```
document.body.onclick = function(event){
    alert(event.currentTarget === document.body);    //true
    alert(this === document.body);                  //true
    alert(event.target === document.getElementById("myBtn"));    //true
};
```

DOMEventObjectExample02.htm

当单击这个例子中的按钮时, `this` 和 `currentTarget` 都等于 `document.body`, 因为事件处理程序是注册到这个元素上的。然而, `target` 元素却等于按钮元素, 因为它是 `click` 事件真正的目标。由于按钮上并没有注册事件处理程序, 结果 `click` 事件就冒泡到了 `document.body`, 在那里事件才得到了处理。

在需要通过一个函数处理多个事件时, 可以使用 `type` 属性。例如:



```
var btn = document.getElementById("myBtn");
var handler = function(event){
```

```

switch(event.type){
    case "click":
        alert("Clicked");
        break;

    case "mouseover":
        event.target.style.backgroundColor = "red";
        break;

    case "mouseout":
        event.target.style.backgroundColor = "";
        break;
}

};

btn.onclick = handler;
btn.onmouseover = handler;
btn.onmouseout = handler;

```

DOMEventObjectExample03.htm

这个例子定义了一个名为 handler 的函数,用于处理 3 种事件:click、mouseover 和 mouseout。当单击按钮时,会出现一个与前面例子中一样的警告框。当按钮移动到按钮上面时,背景颜色应该会变成红色,而当鼠标移动出按钮的范围时,背景颜色应该会恢复为默认值。这里通过检测 event.type 属性,让函数能够确定发生了什么事件,并执行相应的操作。

要阻止特定事件的默认行为,可以使用 preventDefault() 方法。例如,链接的默认行为就是在被单击时会导航到其 href 特性指定的 URL。如果你想阻止链接导航这一默认行为,那么通过链接的 onclick 事件处理程序可以取消它,如下面的例子所示。

```

var link = document.getElementById("myLink");
link.onclick = function(event){
    event.preventDefault();
};

```

DOMEventObjectExample04.htm

只有 cancelable 属性设置为 true 的事件,才可以使用 preventDefault() 来取消其默认行为。

另外,stopPropagation() 方法用于立即停止事件在 DOM 层次中的传播,即取消进一步的事件捕获或冒泡。例如,直接添加到一个按钮的事件处理程序可以调用 stopPropagation(),从而避免触发注册在 document.body 上面的事件处理程序,如下面的例子所示。

```

var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert("Clicked");
    event.stopPropagation();
};

document.body.onclick = function(event){
    alert("Body clicked");
};

```

DOMEventObjectExample05.htm

对于这个例子而言,如果不调用 `stopPropagation()`, 就会在单击按钮时出现两个警告框。可是, 由于 `click` 事件根本不会传播到 `document.body`, 因此就不会触发注册在这个元素上的 `onclick` 事件处理程序。

事件对象的 `eventPhase` 属性, 可以用来确定事件当前正位于事件流的哪个阶段。如果是在捕获阶段调用的事件处理程序, 那么 `eventPhase` 等于 1; 如果事件处理程序处于目标对象上, 则 `eventPhase` 等于 2; 如果是在冒泡阶段调用的事件处理程序, `eventPhase` 等于 3。这里要注意的是, 尽管“处于目标”发生在冒泡阶段, 但 `eventPhase` 仍然一直等于 2。来看下面的例子。

```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.eventPhase); //2
};

document.body.addEventListener("click", function(event){
    alert(event.eventPhase); //1
}, true);

document.body.onclick = function(event){
    alert(event.eventPhase); //3
};
```

DOMEventObjectExample06.htm

当单击这个例子中的按钮时, 首先执行的事件处理程序是在捕获阶段触发的添加到 `document.body` 中的那一个, 结果会弹出一个警告框显示表示 `eventPhase` 的 1。接着, 会触发在按钮上注册的事件处理程序, 此时的 `eventPhase` 值为 2。最后一个被触发的事件处理程序, 是在冒泡阶段执行的添加到 `document.body` 上的那一个, 显示 `eventPhase` 的值为 3。而当 `eventPhase` 等于 2 时, `this`、`target` 和 `currentTarget` 始终都是相等的。



只有在事件处理程序执行期间, `event` 对象才会存在; 一旦事件处理程序执行完成, `event` 对象就会被销毁。

13.3.2 IE 中的事件对象

与访问 DOM 中的 `event` 对象不同, 要访问 IE 中的 `event` 对象有几种不同的方式, 取决于指定事件处理程序的方法。在使用 DOM0 级方法添加事件处理程序时, `event` 对象作为 `window` 对象的一个属性存在。来看下面的例子。

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    var event = window.event;
    alert(event.type);      //"click"
};
```

在此, 我们通过 `window.event` 取得了 `event` 对象, 并检测了被触发事件的类型 (IE 中的 `type` 属性与 DOM 中的 `type` 属性是相同的)。可是, 如果事件处理程序是使用 `attachEvent()` 添加的, 那么就会有一个 `event` 对象作为参数被传入事件处理程序函数中, 如下所示。

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(event){
    alert(event.type);    //"click"
});
```

在像这样使用 `attachEvent()` 的情况下, 也可以通过 `window` 对象来访问 `event` 对象, 就像使用 DOM0 级方法时一样。不过为方便起见, 同一个对象也会作为参数传递。

如果是通过 HTML 特性指定的事件处理程序, 那么还可以通过一个名叫 `event` 的变量来访问 `event` 对象 (与 DOM 中的事件模型相同)。再看一个例子。

```
<input type="button" value="Click Me" onclick="alert(event.type)">
```

IE 的 `event` 对象同样也包含与创建它的事件相关的属性和方法。其中很多属性和方法都有对应的或者相关的 DOM 属性和方法。与 DOM 的 `event` 对象一样, 这些属性和方法也会因为事件类型的不同而不同, 但所有事件对象都会包含下表所列的属性和方法。

属性/方法	类 型	读/写	说 明
<code>cancelBubble</code>	Boolean	读/写	默认值为 <code>false</code> , 但将其设置为 <code>true</code> 就可以取消事件冒泡 (与 DOM 中的 <code>stopPropagation()</code> 方法的作用相同)
<code>returnValue</code>	Boolean	读/写	默认值为 <code>true</code> , 但将其设置为 <code>false</code> 就可以取消事件的默认行为 (与 DOM 中的 <code>preventDefault()</code> 方法的作用相同)
<code>srcElement</code>	Element	只读	事件的目标 (与 DOM 中的 <code>target</code> 属性相同)
<code>type</code>	String	只读	被触发的事件的类型

因为事件处理程序的作用域是根据指定它的方式来确定的, 所以不能认为 `this` 会始终等于事件目标。故而, 最好还是使用 `event.srcElement` 比较保险。例如:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert(window.event.srcElement === this);    //true
};

btn.attachEvent("onclick", function(event){
    alert(event.srcElement === this);           //false
});
```

IEEventObjectExample01.htm

在第一个事件处理程序中 (使用 DOM0 级方法指定的), `srcElement` 属性等于 `this`, 但在第二个事件处理程序中, 这两者的值不相同。

如前所述, `returnValue` 属性相当于 DOM 中的 `preventDefault()` 方法, 它们的作用都是取消给定事件的默认行为。只要将 `returnValue` 设置为 `false`, 就可以阻止默认行为。来看下面的例子。

```
var link = document.getElementById("myLink");
link.onclick = function(){
    window.event.returnValue = false;
};
```

IEEventObjectExample02.htm

这个例子在 onclick 事件处理程序中使用 returnValue 达到了阻止链接默认行为的目的。与 DOM 不同的是,在此没有办法确定事件是否能被取消。

相应地, cancelBubble 属性与 DOM 中的 stopPropagation() 方法作用相同,都是用来停止事件冒泡的。由于 IE 不支持事件捕获,因而只能取消事件冒泡;但 stopPropagation() 可以同时取消事件捕获和冒泡。例如:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert("Clicked");
    window.event.cancelBubble = true;
};

document.body.onclick = function(){
    alert("Body clicked");
};
```

[IEEventObjectExample03.htm](#)

通过在 onclick 事件处理程序中将 cancelBubble 设置为 true,就可阻止事件通过冒泡而触发 document.body 中注册的事件处理程序。结果,在单击按钮之后,只会显示一个警告框。

13.3.3 跨浏览器的事件对象

虽然 DOM 和 IE 中的 event 对象不同,但基于它们之间的相似性依旧可以拿出跨浏览器的方案来。IE 中 event 对象的全部信息和方法 DOM 对象中都有,只不过实现方式不一样。不过,这种对应关系让实现两种事件模型之间的映射非常容易。可以对前面介绍的 EventUtil 对象加以增强,添加如下方法以求同存异。



```
var EventUtil = {

    addHandler: function(element, type, handler){
        //省略的代码
    },

    getEvent: function(event){
        return event ? event : window.event;
    },

    getTarget: function(event){
        return event.target || event.srcElement;
    },

    preventDefault: function(event){
        if (event.preventDefault){
            event.preventDefault();
        } else {
            event.returnValue = false;
        }
    },

    removeHandler: function(element, type, handler){
        //省略的代码
    },

    stopPropagation: function(event){
```




```

        if (event.stopPropagation){
            event.stopPropagation();
        } else {
            event.cancelBubble = true;
        }
    }
};

```

EventUtil.js

以上代码显示，我们为 EventUtil 添加了 4 个新方法。第一个是 `getEvent()`，它返回对 event 对象的引用。考虑到 IE 中事件对象的位置不同，可以使用这个方法取得 event 对象，而不必担心指定事件处理程序的方式。在使用这个方法时，必须假设有一个事件对象传入到事件处理程序中，而且要把该变量传给这个方法，如下所示。



```

btn.onclick = function(event){
    event = EventUtil.getEvent(event);
};

```

CrossBrowserEventObjectExample01.htm

在兼容 DOM 的浏览器中，event 变量只是简单地传入和返回。而在 IE 中，event 参数是未定义的 (undefined)，因此就会返回 window.event。将这一行代码添加到事件处理程序的开头，就可以确保随时都能使用 event 对象，而不必担心用户使用的是哪种浏览器。

第二个方法是 `getTarget()`，它返回事件的目标。在这个方法内部，会检测 event 对象的 target 属性，如果存在则返回该属性的值；否则，返回 srcElement 属性的值。可以像下面这样使用这个方法。

```

btn.onclick = function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
};

```

CrossBrowserEventObjectExample01.htm

第三个方法是 `preventDefault()`，用于取消事件的默认行为。在传入 event 对象后，这个方法会检查是否存在 `preventDefault()` 方法，如果存在则调用该方法。如果 `preventDefault()` 方法不存在，则将 returnValue 设置为 false。下面是使用这个方法的例子。

```

var link = document.getElementById("myLink");
link.onclick = function(event){
    event = EventUtil.getEvent(event);
    EventUtil.preventDefault(event);
};

```

CrossBrowserEventObjectExample02.htm

以上代码可以确保在所有浏览器中单击该链接都不会打开另一个页面。首先，使用 `EventUtil.getEvent()` 取得 event 对象，然后将其传入到 `EventUtil.preventDefault()` 以取消默认行为。

第四个方法是 `stopPropagation()`，其实现方式类似。首先尝试使用 DOM 方法阻止事件流，否则就使用 `cancelBubble` 属性。下面看一个例子。



```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert("Clicked");
    event = EventUtil.getEvent(event);
    EventUtil.stopPropagation(event);
};

document.body.onclick = function(event){
    alert("Body clicked");
};
```

CrossBrowserEventObjectExample03.htm

在此，首先使用 `EventUtil.getEvent()` 取得了 `event` 对象，然后又将其传入到 `EventUtil.stopPropagation()`。别忘了由于 IE 不支持事件捕获，因此这个方法在跨浏览器的情况下，也只能用来阻止事件冒泡。

13.4 事件类型

Web 浏览器中可能发生的事件有很多类型。如前所述，不同的事件类型具有不同的信息，而“DOM3 级事件”规定了以下几类事件。

- ☐ UI (User Interface, 用户界面) 事件，当用户与页面上的元素交互时触发；
- ☐ 焦点事件，当元素获得或失去焦点时触发；
- ☐ 鼠标事件，当用户通过鼠标在页面上执行操作时触发；
- ☐ 滚轮事件，当使用鼠标滚轮（或类似设备）时触发；
- ☐ 文本事件，当在文档中输入文本时触发；
- ☐ 键盘事件，当用户通过键盘在页面上执行操作时触发；
- ☐ 合成事件，当为 IME (Input Method Editor, 输入法编辑器) 输入字符时触发；
- ☐ 变动 (mutation) 事件，当底层 DOM 结构发生变化时触发。
- ☐ 变动名称事件，当元素或属性名变动时触发。此类事件已经被废弃，没有任何浏览器实现它们，因此本章不做介绍。

除了这几类事件之外，HTML5 也定义了一组事件，而有些浏览器还会在 DOM 和 BOM 中实现其他专有事件。这些专有的事件一般都是根据开发人员需求定制的，没有什么规范，因此不同浏览器的实现有可能不一致。

DOM3 级事件模块在 DOM2 级事件模块基础上重新定义了这些事件，也添加了一些新事件。包括 IE9 在内的所有主流浏览器都支持 DOM2 级事件。IE9 也支持 DOM3 级事件。

13.4.1 UI 事件

UI 事件指的是那些不一定与用户操作有关的事件。这些事件在 DOM 规范出现之前，都是以这种或那种形式存在的，而在 DOM 规范中保留是为了向后兼容。现有的 UI 事件如下。

- ☐ DOMActivate: 表示元素已经被用户操作（通过鼠标或键盘）激活。这个事件在 DOM3 级事件中被废弃，但 Firefox 2+ 和 Chrome 支持它。考虑到不同浏览器实现的差异，不建议使用这个事件。

- ❑ load: 当页面完全加载后在 window 上面触发, 当所有框架都加载完毕时在框架集上面触发, 当图像加载完毕时在元素上面触发, 或者当嵌入的内容加载完毕时在<object>元素上面触发。
- ❑ unload: 当页面完全卸载后在 window 上面触发, 当所有框架都卸载后在框架集上面触发, 或者当嵌入的内容卸载完毕后在<object>元素上面触发。
- ❑ abort: 在用户停止下载过程时, 如果嵌入的内容没有加载完, 则在<object>元素上面触发。
- ❑ error: 当发生 JavaScript 错误时在 window 上面触发, 当无法加载图像时在元素上面触发, 当无法加载嵌入内容时在<object>元素上面触发, 或者当有一或多个框架无法加载时在框架集上面触发。第 17 章将继续讨论这个事件。
- ❑ select: 当用户选择文本框 (<input>或<textarea>) 中的一或多个字符时触发。第 14 章将继续讨论这个事件。
- ❑ resize: 当窗口或框架的大小变化时在 window 或框架上面触发。
- ❑ scroll: 当用户滚动带滚动条的元素中的内容时, 在该元素上面触发。<body>元素中包含所加载页面的滚动条。

多数这些事件都与 window 对象或表单控件相关。

除了 DOMActivate 之外, 其他事件在 DOM2 级事件中都归为 HTML 事件(DOMActivate 在 DOM2 级中仍然属于 UI 事件)。要确定浏览器是否支持 DOM2 级事件规定的 HTML 事件, 可以使用如下代码:

```
var isSupported = document.implementation.hasFeature("HTMLEvents", "2.0");
```

注意, 只有根据“DOM2 级事件”实现这些事件的浏览器才会返回 true。而以非标准方式支持这些事件的浏览器则会返回 false。要确定浏览器是否支持“DOM3 级事件”定义的事件, 可以使用如下代码:

```
var isSupported = document.implementation.hasFeature("UIEvent", "3.0");
```

1. load 事件

JavaScript 中最常用的一个事件就是 load。当页面完全加载后(包括所有图像、JavaScript 文件、CSS 文件等外部资源), 就会触发 window 上面的 load 事件。有两种定义 onload 事件处理程序的方式。第一种方式是使用如下所示的 JavaScript 代码:

```
EventUtil.addHandler(window, "load", function(event){
    alert("Loaded!");
});
```

LoadEventExample01.htm

这是通过 JavaScript 来指定事件处理程序的方式, 使用了本章前面定义的跨浏览器的 EventUtil 对象。与添加其他事件一样, 这里也给事件处理程序传入了一个 event 对象。这个 event 对象中不包含有关这个事件的任何附加信息, 但在兼容 DOM 的浏览器中, event.target 属性的值会被设置为 document, 而 IE 并不会为这个事件设置 srcElement 属性。

第二种指定 onload 事件处理程序的方式是为<body>元素添加一个 onload 特性, 如下面的例子所示:

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Load Event Example</title>
</head>
<body onload="alert('Loaded!')">

</body>
</html>
```

LoadEventExample02.htm

一般来说,在 window 上面发生的任何事件都可以在<body/>元素中通过相应的特性来指定,因为在 HTML 中无法访问 window 元素。实际上,这只是为了保证向后兼容的一种权宜之计,但所有浏览器都能很好地支持这种方式。我们建议读者尽可能使用 JavaScript 方式。



根据“DOM2 级事件”规范,应该在 document 而非 window 上面触发 load 事件。但是,所有浏览器都在 window 上面实现了该事件,以确保向后兼容。

图像上面也可以触发 load 事件,无论是在 DOM 中的图像元素还是 HTML 中的图像元素。因此,可以在 HTML 中为任何图像指定 onload 事件处理程序,例如:

```

```

LoadEventExample03.htm

这样,当例子中的图像加载完毕后就会显示一个警告框。同样的功能也可以使用 JavaScript 来实现,例如:



```
var image = document.getElementById("myImage");
EventUtil.addHandler(image, "load", function(event){
    event = EventUtil.getEvent(event);
    alert(EventUtil.getTarget(event).src);
});
```

LoadEventExample04.htm

这里,使用 JavaScript 指定了 onload 事件处理程序。同时也传入了 event 对象,尽管它也不包含什么有用的信息。不过,事件的目标是元素,因此可以通过 src 属性访问并显示该信息。

在创建新的元素时,可以为其指定一个事件处理程序,以便图像加载完毕后给出提示。此时,最重要的是要在指定 src 属性之前先指定事件,如下面的例子所示。

```
EventUtil.addHandler(window, "load", function(){
    var image = document.createElement("img");
    EventUtil.addHandler(image, "load", function(event){
        event = EventUtil.getEvent(event);
        alert(EventUtil.getTarget(event).src);
    });
    document.body.appendChild(image);
    image.src = "smile.gif";
});
```

LoadEventExample05.htm

在这个例子中，首先为 window 指定了 onload 事件处理程序。原因在于，我们是想向 DOM 中添加一个新元素，所以必须确定页面已经加载完毕——如果在页面加载前操作 document.body 会导致错误。然后，创建了一个新的图像元素，并设置了其 onload 事件处理程序。最后又将这个图像添加到页面中，还设置了它的 src 属性。这里有一点需要格外注意：新图像元素不一定要从添加到文档后才开始下载，只要设置了 src 属性就会开始下载。

同样的功能也可以通过使用 DOM0 级的 Image 对象实现。在 DOM 出现之前，开发人员经常使用 Image 对象在客户端预先加载图像。可以像使用元素一样使用 Image 对象，只不过无法将其添加到 DOM 树中。下面来看一个例子。

```
EventUtil.addHandler(window, "load", function(){
    var image = new Image();
    EventUtil.addHandler(image, "load", function(event){
        alert("Image loaded!");
    });
    image.src = "smile.gif";
});
```

LoadEventExample06.htm

在此，我们使用 Image 构造函数创建了一个新图像的实例，然后又为它指定了事件处理程序。有的浏览器将 Image 对象实现为元素，但并非所有浏览器都如此，所以最好将它们区别对待。



在不属于 DOM 文档的图像（包括未添加到文档的元素和 Image 对象）上触发 load 事件时，IE8 及之前版本不会生成 event 对象。IE9 修复了这个问题。

还有一些元素也以非标准的方式支持 load 事件。在 IE9+、Firefox、Opera、Chrome 和 Safari 3+ 及更高版本中，<script>元素也会触发 load 事件，以便开发人员确定动态加载的 JavaScript 文件是否加载完毕。与图像不同，只有在设置了<script>元素的 src 属性并将该元素添加到文档后，才会开始下载 JavaScript 文件。换句话说，对于<script>元素而言，指定 src 属性和指定事件处理程序的先后顺序就不重要了。以下代码展示了怎样为<script>元素指定事件处理程序。

```
EventUtil.addHandler(window, "load", function(){
    var script = document.createElement("script");
    EventUtil.addHandler(script, "load", function(event){
        alert("Loaded");
    });
    script.src = "example.js";
    document.body.appendChild(script);
});
```

LoadEventExample07.htm

这个例子使用了跨浏览器的 EventUtil 对象为新创建的<script>元素指定了 onload 事件处理程序。此时，大多数浏览器中 event 对象的 target 属性引用的都是<script>节点，而在 Firefox 3 之前的版本中，引用的则是 document。IE8 及更早版本不支持<script>元素上的 load 事件。

IE 和 Opera 还支持<link>元素上的 load 事件，以便开发人员确定样式表是否加载完毕。例如：

```

EventUtil.addHandler(window, "load", function(){
    var link = document.createElement("link");
    link.type = "text/css";
    link.rel = "stylesheet";
    EventUtil.addHandler(link, "load", function(event){
        alert("css loaded");
    });
    link.href = "example.css";
    document.getElementsByTagName("head")[0].appendChild(link);
});

```

LoadEventExample07.htm

与<script>节点类似，在未指定 href 属性并将<link>元素添加到文档之前也不会开始下载样式表。

2. unload 事件

与 load 事件对应的是 unload 事件，这个事件在文档被完全卸载后触发。只要用户从一个页面切换到另一个页面，就会发生 unload 事件。而利用这个事件最多的情况是清除引用，以避免内存泄漏。与 load 事件类似，也有两种指定 onunload 事件处理程序的方式。第一种方式是使用 JavaScript，如下所示：


```

EventUtil.addHandler(window, "unload", function(event){
    alert("Unloaded");
});

```

此时生成的 event 对象在兼容 DOM 的浏览器中只包含 target 属性（值为 document）。IE8 及之前版本则为这个事件对象提供了 srcElement 属性。

指定事件处理程序的第二种方式，也是为<body>元素添加一个特性（与 load 事件相似），如下面的例子所示：



```

<!DOCTYPE html>
<html>
<head>
    <title>Unload Event Example</title>
</head>
<body onunload="alert('Unloaded!')">

</body>
</html>

```

UnloadEventExample01.htm

无论使用哪种方式，都要小心编写 onunload 事件处理程序中的代码。既然 unload 事件是在一切都被卸载之后才触发，那么在页面加载后存在的那些对象，此时就不一定存在了。此时，操作 DOM 节点或者元素的样式就会导致错误。



根据“DOM2 级事件”，应该在<body>元素而非 window 对象上面触发 unload 事件。不过，所有浏览器都在 window 上实现了 unload 事件，以确保向后兼容。

3. resize 事件

当浏览器窗口被调整到一个新的高度或宽度时，就会触发 resize 事件。这个事件在 window（窗口）上面触发，因此可以通过 JavaScript 或者<body>元素中的 onresize 特性来指定事件处理程序。如

前所述, 我们还是推荐使用如下所示的 JavaScript 方式:

```
EventUtil.addHandler(window, "resize", function(event){
    alert("Resized");
});
```

与其他发生在 window 上的事件类似, 在兼容 DOM 的浏览器中, 传入事件处理程序中的 event 对象有一个 target 属性, 值为 document; 而 IE8 及之前版本则未提供任何属性。

关于何时会触发 resize 事件, 不同浏览器有不同的机制。IE、Safari、Chrome 和 Opera 会在浏览器窗口变化了 1 像素时就触发 resize 事件, 然后随着变化不断重复触发。Firefox 则只会在用户停止调整窗口大小时才会触发 resize 事件。由于存在这个差别, 应该注意不要在这个事件的处理程序中加入大计算量的代码, 因为这些代码有可能被频繁执行, 从而导致浏览器反应明显变慢。



浏览器窗口最小化或最大化时也会触发 **resize** 事件。

4. scroll 事件

虽然 scroll 事件是在 window 对象上发生的, 但它实际表示的则是页面中相应元素的变化。在混杂模式下, 可以通过<body>元素的 scrollLeft 和 scrollTop 来监控到这一变化; 而在标准模式下, 除 Safari 之外的所有浏览器都会通过<html>元素来反映这一变化 (Safari 仍然基于<body>跟踪滚动位置), 如下面的例子所示:

```
EventUtil.addHandler(window, "scroll", function(event){
    if (document.compatMode == "CSS1Compat"){
        alert(document.documentElement.scrollTop);
    } else {
        alert(document.body.scrollTop);
    }
});
```

ScrollEventExample01.htm

以上代码指定的事件处理程序会输出页面的垂直滚动位置——根据呈现模式不同使用了不同的元素。由于 Safari 3.1 之前的版本不支持 document.compatMode, 因此旧版本的浏览器就会满足第二个条件。

与 resize 事件类似, scroll 事件也会在文档被滚动期间重复被触发, 所以有必要尽量保持事件处理程序的代码简单。

13.4.2 焦点事件

焦点事件会在页面获得或失去焦点时触发。利用这些事件并与 document.hasFocus() 方法及 document.activeElement 属性配合, 可以知晓用户在页面上的行踪。有以下 6 个焦点事件。

- blur: 在元素失去焦点时触发。这个事件不会冒泡; 所有浏览器都支持它。
- DOMFocusIn: 在元素获得焦点时触发。这个事件与 HTML 事件 focus 等价, 但它冒泡。只有 Opera 支持这个事件。DOM3 级事件废弃了 DOMFocusIn, 选择了 focusin。
- DOMFocusOut: 在元素失去焦点时触发。这个事件是 HTML 事件 blur 的通用版本。只有 Opera 支持这个事件。DOM3 级事件废弃了 DOMFocusOut, 选择了 focusout。

- **focus**: 在元素获得焦点时触发。这个事件不会冒泡; 所有浏览器都支持它。
- **focusin**: 在元素获得焦点时触发。这个事件与 HTML 事件 **focus** 等价, 但它冒泡。支持这个事件的浏览器有 IE5.5+、Safari 5.1+、Opera 11.5+ 和 Chrome。
- **focusout**: 在元素失去焦点时触发。这个事件是 HTML 事件 **blur** 的通用版本。支持这个事件的浏览器有 IE5.5+、Safari 5.1+、Opera 11.5+ 和 Chrome。

这一类事件中最主要的两个是 **focus** 和 **blur**, 它们都是 JavaScript 早期就得到所有浏览器支持的事件。这些事件的最大问题是它们不冒泡。因此, IE 的 **focusin** 和 **focusout** 与 Opera 的 **DOMFocusIn** 和 **DOMFocusOut** 才会发生重叠。IE 的方式最后被 DOM3 级事件采纳为标准方式。

当焦点从页面中的一个元素移动到另一个元素, 会依次触发下列事件:

- (1) **focusout** 在失去焦点的元素上触发;
- (2) **focusin** 在获得焦点的元素上触发;
- (3) **blur** 在失去焦点的元素上触发;
- (4) **DOMFocusOut** 在失去焦点的元素上触发;
- (5) **focus** 在获得焦点的元素上触发;
- (6) **DOMFocusIn** 在获得焦点的元素上触发。

其中, **blur**、**DOMFocusOut** 和 **focusout** 的事件目标是失去焦点的元素; 而 **focus**、**DOMFocusIn** 和 **focusin** 的事件目标是获得焦点的元素。

要确定浏览器是否支持这些事件, 可以使用如下代码:

```
var isSupported = document.implementation.hasFeature("FocusEvent", "3.0");
```



即使 **focus** 和 **blur** 不冒泡, 也可以在捕获阶段侦听到它们。Peter-Paul Koch 就此写过一篇非常棒的文章: www.quirksmode.org/blog/archives/2008/04/delegating_the.html。

13.4.3 鼠标与滚轮事件

鼠标事件是 Web 开发中最常用的一类事件, 毕竟鼠标还是最主要的定位设备。DOM3 级事件中定义了 9 个鼠标事件, 简介如下。

- **click**: 在用户单击主鼠标按钮 (一般是左边的按钮) 或者按下回车键时触发。这一点对确保易访问性很重要, 意味着 **onclick** 事件处理程序既可以通过键盘也可以通过鼠标执行。
- **dblclick**: 在用户双击主鼠标按钮 (一般是左边的按钮) 时触发。从技术上说, 这个事件并不是 DOM2 级事件规范中规定的, 但鉴于它得到了广泛支持, 所以 DOM3 级事件将其纳入了标准。
- **mousedown**: 在用户按下了任意鼠标按钮时触发。不能通过键盘触发这个事件。
- **mouseenter**: 在鼠标光标从元素外部首次移动到元素范围之内时触发。这个事件不冒泡, 而且在光标移动到后代元素上不会触发。DOM2 级事件并没有定义这个事件, 但 DOM3 级事件将它纳入了规范。IE、Firefox 9+ 和 Opera 支持这个事件。
- **mouseleave**: 在位于元素上方的鼠标光标移动到元素范围之外时触发。这个事件不冒泡, 而且在光标移动到后代元素上不会触发。DOM2 级事件并没有定义这个事件, 但 DOM3 级事件将它纳入了规范。IE、Firefox 9+ 和 Opera 支持这个事件。
- **mousemove**: 当鼠标指针在元素内部移动时重复地触发。不能通过键盘触发这个事件。

- ❑ `mouseout`: 在鼠标指针位于一个元素上方, 然后用户将其移入另一个元素时触发。又移入的另一个元素可能位于前一个元素的外部, 也可能是这个元素的子元素。不能通过键盘触发这个事件。
- ❑ `mouseover`: 在鼠标指针位于一个元素外部, 然后用户将其首次移入另一个元素边界之内时触发。不能通过键盘触发这个事件。
- ❑ `mouseup`: 在用户释放鼠标按钮时触发。不能通过键盘触发这个事件。

页面上的所有元素都支持鼠标事件。除了 `mouseenter` 和 `mouseleave`, 所有鼠标事件都会冒泡, 也可以被取消, 而取消鼠标事件将会影响浏览器的默认行为。取消鼠标事件的默认行为还会影响其他事件, 因为鼠标事件与其他事件是密不可分的关系。

只有在同一个元素上相继触发 `mousedown` 和 `mouseup` 事件, 才会触发 `click` 事件; 如果 `mousedown` 或 `mouseup` 中的一个被取消, 就不会触发 `click` 事件。类似地, 只有触发两次 `click` 事件, 才会触发一次 `dblclick` 事件。如果有代码阻止了连续两次触发 `click` 事件(可能是直接取消 `click` 事件, 也可能通过取消 `mousedown` 或 `mouseup` 间接实现), 那么就不会触发 `dblclick` 事件了。这 4 个事件触发的顺序始终如下:

- (1) `mousedown`
- (2) `mouseup`
- (3) `click`
- (4) `mousedown`
- (5) `mouseup`
- (6) `click`
- (7) `dblclick`

显然, `click` 和 `dblclick` 事件都会依赖于其他先行事件的触发; 而 `mousedown` 和 `mouseup` 则不受其他事件的影响。

IE8 及之前版本中的实现有一个小 bug, 因此在双击事件中, 会跳过第二个 `mousedown` 和 `click` 事件, 其顺序如下:

- (1) `mousedown`
- (2) `mouseup`
- (3) `click`
- (4) `mouseup`
- (5) `dblclick`

IE9 修复了这个 bug, 之后顺序就正确了。

使用以下代码可以检测浏览器是否支持以上 DOM2 级事件 (除 `dblclick`、`mouseenter` 和 `mouseleave` 之外):

```
var isSupported = document.implementation.hasFeature("MouseEvents", "2.0");
```

要检测浏览器是否支持上面的所有事件, 可以使用以下代码:

```
var isSupported = document.implementation.hasFeature("MouseEvent", "3.0")
```

注意, DOM3 级事件的 feature 名是 "MouseEvent", 而非 "MouseEvents"。

鼠标事件中还有一类滚轮事件。而说是一类事件, 其实就是一个 `mousewheel` 事件。这个事件跟踪鼠标滚轮, 类似于 Mac 的触控板。

1. 客户区坐标位置

鼠标事件都是在浏览器视口中的特定位置上发生的。这个位置信息保存在事件对象的 `clientX` 和 `clientY` 属性中。所有浏览器都支持这两个属性，它们的值表示事件发生时鼠标指针在视口中的水平和垂直坐标。图 13-4 展示了视口中客户区坐标位置的含义。

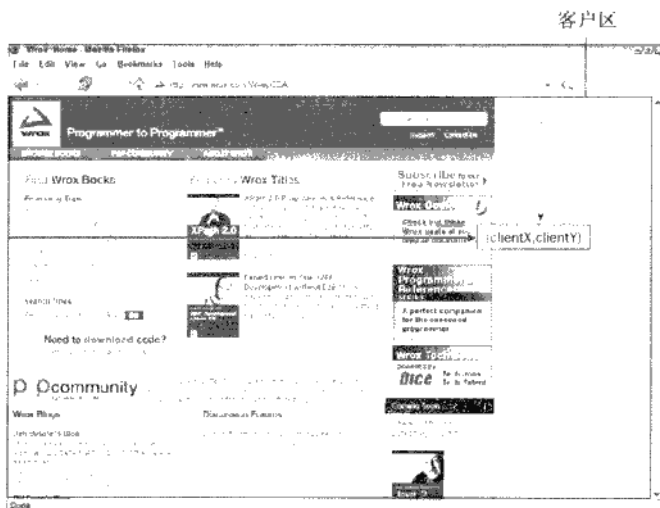


图 13-4

可以使用类似下列代码取得鼠标事件的客户端坐标信息：

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    alert("Client coordinates: " + event.clientX + "," + event.clientY);
});
```

[ClientCoordinatesExample01.htm](#)

这里为一个 `<div>` 元素指定了 `onclick` 事件处理程序。当用户单击这个元素时，就会看到事件的客户端坐标信息。注意，这些值中不包括页面滚动的距离，因此这个位置并不表示鼠标在页面上的位置。

2. 页面坐标位置

通过客户区坐标能够知道鼠标是在视口中什么位置发生的，而页面坐标通过事件对象的 `pageX` 和 `pageY` 属性，能告诉你事件是在页面中的什么位置发生的。换句话说，这两个属性表示鼠标光标在页面中的位置，因此坐标是从页面本身而非视口的左边和顶边计算的。

以下代码可以取得鼠标事件在页面中的坐标：

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    alert("Page coordinates: " + event.pageX + "," + event.pageY);
});
```

[PageCoordinatesExample01.htm](#)

在页面没有滚动的情况下, `pageX` 和 `pageY` 的值与 `clientX` 和 `clientY` 的值相等。

IE8 及更早版本不支持事件对象上的页面坐标, 不过使用客户区坐标和滚动信息可以计算出来。这时候需要用到 `document.body` (混杂模式) 或 `document.documentElement` (标准模式) 中的 `scrollLeft` 和 `scrollTop` 属性。计算过程如下所示:

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    var pageX = event.pageX,
        pageY = event.pageY;

    if (pageX === undefined){
        pageX = event.clientX + (document.body.scrollLeft ||
            document.documentElement.scrollLeft);
    }

    if (pageY === undefined){
        pageY = event.clientY + (document.body.scrollTop ||
            document.documentElement.scrollTop);
    }

    alert("Page coordinates: " + pageX + "," + pageY);
});
```

PageCoordinatesExample01.htm

3. 屏幕坐标位置

鼠标事件发生时, 不仅会有相对于浏览器窗口的位置, 还有一个相对于整个电脑屏幕的位置。而通过 `screenX` 和 `screenY` 属性就可以确定鼠标事件发生时鼠标指针相对于整个屏幕的坐标信息。图 13-5 展示了浏览器中屏幕坐标的含义。

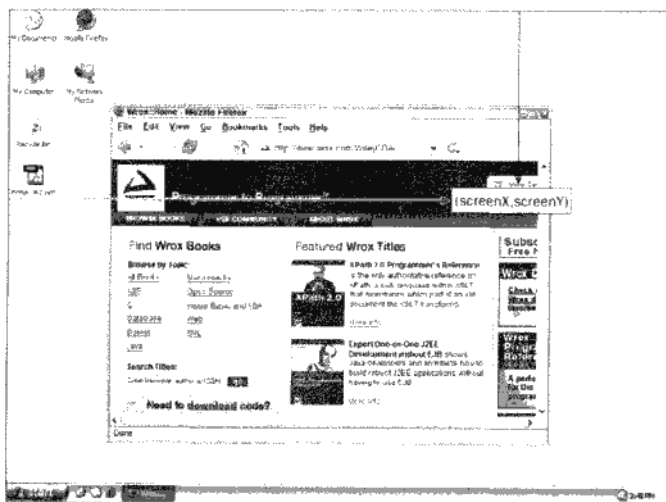


图 13-5

可以使用类似下面的代码取得鼠标事件的屏幕坐标：

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    alert("Screen coordinates: " + event.screenX + "," + event.screenY);
});
```

ScreenCoordinatesExample01.htm

与前一个例子类似，这里也是为<div>元素指定了一个 onclick 事件处理程序。当这个元素被单击时，就会显示出事件的屏幕坐标信息了。

4. 修改键

虽然鼠标事件主要是使用鼠标来触发的，但在按下鼠标时键盘上的某些键的状态也可以影响到所要采取的操作。这些修改键就是 Shift、Ctrl、Alt 和 Meta（在 Windows 键盘中是 Windows 键，在苹果机中是 Cmd 键），它们经常被用来修改鼠标事件的行为。DOM 为此规定了 4 个属性，表示这些修改键的状态：shiftKey、ctrlKey、altKey 和 metaKey。这些属性中包含的都是布尔值，如果相应的键被按下了，则值为 true，否则值为 false。当某个鼠标事件发生时，通过检测这几个属性就可以确定用户是否同时按下了其中的键。来看下面的例子。

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    var keys = new Array();

    if (event.shiftKey){
        keys.push("shift");
    }

    if (event.ctrlKey){
        keys.push("ctrl");
    }

    if (event.altKey){
        keys.push("alt");
    }

    if (event.metaKey){
        keys.push("meta");
    }

    alert("Keys: " + keys.join(", "));
});
```

ModifierKeysExample01.htm

在这个例子中，我们通过一个 onclick 事件处理程序检测了不同修改键的状态。数组 keys 中包含了被按下的修改键的名称。换句话说，如果有属性值为 true，就会将对应修改键的名称添加到 keys 数组中。在事件处理程序的最后，有一个警告框将检测到的键的信息显示给用户。



IE9、Firefox、Safari、Chrome 和 Opera 都支持这 4 个键。IE8 及之前版本不支持 `metaKey` 属性。

5. 相关元素

在发生 `mouseover` 和 `mouseout` 事件时，还会涉及更多的元素。这两个事件都会涉及把鼠标指针从一个元素的边界之内移动到另一个元素的边界之内。对 `mouseover` 事件而言，事件的主目标是获得光标的元素，而相关元素就是那个失去光标的元素。类似地，对 `mouseout` 事件而言，事件的主目标是失去光标的元素，而相关元素则是获得光标的元素。来看下面的例子。



```
<!DOCTYPE html>
<html>
<head>
  <title>Related Elements Example</title>
</head>
<body>
  <div id="myDiv" style="background-color:red;height:100px;width:100px;"></div>
</body>
</html>
```

RelatedElementsExample01.htm

这个例子会在页面上显示一个 `<div>` 元素。如果鼠标指针一开始位于这个 `<div>` 元素上，然后移出了这个元素，那么就会在 `<div>` 元素上触发 `mouseout` 事件，相关元素就是 `<body>` 元素。与此同时，`<body>` 元素上面会触发 `mouseover` 事件，而相关元素变成了 `<div>`。

DOM 通过 `event` 对象的 `relatedTarget` 属性提供了相关元素的信息。这个属性只对于 `mouseover` 和 `mouseout` 事件才包含值；对于其他事件，这个属性的值是 `null`。IE8 及之前版本不支持 `relatedTarget` 属性，但提供了保存着同样信息的不同属性。在 `mouseover` 事件触发时，IE 的 `fromElement` 属性中保存了相关元素；在 `mouseout` 事件触发时，IE 的 `toElement` 属性中保存着相关元素。（IE9 支持所有这些属性。）可以把下面这个跨浏览器取得相关元素的方法添加到 `EventUtil` 对象中。



```
var EventUtil = {
  //省略了其他代码

  getRelatedTarget: function(event){
    if (event.relatedTarget){
      return event.relatedTarget;
    } else if (event.toElement){
      return event.toElement;
    } else if (event.fromElement){
      return event.fromElement;
    } else {
      return null;
    }
  },

  //省略了其他代码
};
```

与以前添加的跨浏览器方法一样，这个方法也使用了特性检测来确定返回哪个值。可以像下面这样使用 `EventUtil.getRelatedTarget()` 方法：

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "mouseout", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
    var relatedTarget = EventUtil.getRelatedTarget(event);
    alert("Moused out of " + target.tagName + " to " + relatedTarget.tagName);
});
```

RelatedElementsExample01.htm

这个例子为 `<div>` 元素的 `mouseout` 事件注册了一个事件处理程序。当事件触发时，会有一个警告框显示鼠标移出和移入的元素信息。

6. 鼠标按钮

只有在主鼠标按钮被单击（或键盘回车键被按下）时才会触发 `click` 事件，因此检测按钮的信息并不是必要的。但对于 `mousedown` 和 `mouseup` 事件来说，则在其 `event` 对象存在一个 `button` 属性，表示按下或释放的按钮。DOM 的 `button` 属性可能有如下 3 个值：0 表示主鼠标按钮，1 表示中间的鼠标按钮（鼠标滚轮按钮），2 表示次鼠标按钮。在常规的设置中，主鼠标按钮就是鼠标左键，而次鼠标按钮就是鼠标右键。

IE8 及之前版本也提供了 `button` 属性，但这个属性的值与 DOM 的 `button` 属性有很大差异。

- 0：表示没有按下按钮。
- 1：表示按下了主鼠标按钮。
- 2：表示按下了次鼠标按钮。
- 3：表示同时按下了主、次鼠标按钮。
- 4：表示按下了中间的鼠标按钮。
- 5：表示同时按下了主鼠标按钮和中间的鼠标按钮。
- 6：表示同时按下了次鼠标按钮和中间的鼠标按钮。
- 7：表示同时按下了三个鼠标按钮。

不难想见，DOM 模型下的 `button` 属性比 IE 模型下的 `button` 属性更简单也更为实用，因为同时按下多个鼠标按钮的情形十分罕见。最常见的做法就是将 IE 模型规范化为 DOM 方式，毕竟除 IE8 及更早版本之外的其他浏览器都原生支持 DOM 模型。而对主、中、次按钮的映射并不困难，只要将 IE 的其他选项分别转换成如同按下这三个按键中的一个即可（同时将主按钮作为优先选取的对象）。换句话说，IE 中返回的 5 和 7 会被转换成 DOM 模型中的 0。

由于单独使用能力检测无法确定差异（两种模型有同名的 `button` 属性），因此必须另辟蹊径。我们知道，支持 DOM 版鼠标事件的浏览器可以通过 `hasFeature()` 方法来检测，所以可以再为 `EventUtil` 对象添加如下 `getButton()` 方法。

```
var EventUtil = {

    //省略了其他代码

    getButton: function(event){
        if (document.implementation.hasFeature("MouseEvent", "2.0")){
            return event.button;
        }
    }
};
```


```

    } else {
        switch(event.button){
            case 0:
            case 1:
            case 3:
            case 5:
            case 7:
                return 0;
            case 2:
            case 6:
                return 2;
            case 4:
                return 1;
        }
    }
    1,
    //省略了其他代码
};

```

EventUtil.js

通过检测"MouseEvents"这个特性，就可以确定 event 对象中存在的 button 属性中是否包含正确的值。如果测试失败，说明是 IE，就必须对相应的值进行规范化。以下是使用该方法的示例。



```

var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "mousedown", function(event){
    event = EventUtil.getEvent(event);
    alert(EventUtil.getButton(event));
});

```

ButtonExample01.htm

在这个例子中，我们为一个<div>元素添加了一个 onmousedown 事件处理程序。当在这个元素上按下鼠标按钮时，会有警告框显示按钮的代码。



在使用 onmouseup 事件处理程序时，button 的值表示释放的是哪个按钮。此外，如果不是按下或释放了主鼠标按钮，Opera 不会触发 mouseup 或 mousedown 事件。

7. 更多的事件信息

“DOM2 级事件”规范在 event 对象中还提供了 detail 属性，用于给出有关事件的更多信息。对于鼠标事件来说，detail 中包含了一个数值，表示在给定位置上发生了多少次单击。在同一个像素上相继地发生一次 mousedown 和一次 mouseup 事件算作一次单击。detail 属性从 1 开始计数，每次单击发生后都会递增。如果鼠标在 mousedown 和 mouseup 之间移动了位置，则 detail 会被重置为 0。

IE 也通过下列属性为鼠标事件提供了更多信息。

- altLeft: 布尔值，表示是否按下了 Alt 键。如果 altLeft 的值为 true，则 altKey 的值也为 true。
- ctrlLeft: 布尔值，表示是否按下了 Ctrl 键。如果 ctrlLeft 的值为 true，则 ctrlKey 的值也为 true。

- `offsetX`: 光标相对于目标元素边界的 x 坐标。
- `offsetY`: 光标相对于目标元素边界的 y 坐标。
- `shiftLeft`: 布尔值, 表示是否按下了 `Shift` 键。如果 `shiftLeft` 的值为 `true`, 则 `shiftKey` 的值也为 `true`。

这些属性的用处并不大, 原因一方面是只有 IE 支持它们, 另一方是它们提供的信息要么没有什么价值, 要么可以通过其他方式计算得来。

8. 鼠标滚轮事件

IE 6.0 首先实现了 `mousewheel` 事件。此后, Opera、Chrome 和 Safari 也都实现了这个事件。当用户通过鼠标滚轮与页面交互、在垂直方向上滚动页面时 (无论向上还是向下), 就会触发 `mousewheel` 事件。这个事件可以在任何元素上面触发, 最终会冒泡到 `document` (IE8) 或 `window` (IE9、Opera、Chrome 及 Safari) 对象。与 `mousewheel` 事件对应的 `event` 对象除包含鼠标事件的所有标准信息外, 还包含一个特殊的 `wheelDelta` 属性。当用户向前滚动鼠标滚轮时, `wheelDelta` 是 120 的倍数; 当用户向后滚动鼠标滚轮时, `wheelDelta` 是 -120 的倍数。图 13-6 展示了这个属性。

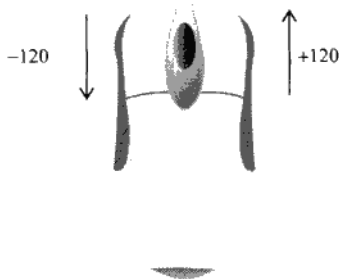


图 13-6

将 `mousewheel` 事件处理程序指定给页面中的任何元素或 `document` 对象, 即可处理鼠标滚轮的交互操作。来看下面的例子。

```
EventUtil.addHandler(document, "mousewheel", function(event){
    event = EventUtil.getEvent(event);
    alert(event.wheelDelta);
});
```

这个例子会在发生 `mousewheel` 事件时显示 `wheelDelta` 的值。多数情况下, 只要知道鼠标滚轮滚动的方向就够了, 而这通过检测 `wheelDelta` 的正负号就可以确定。

有一点要注意: 在 Opera 9.5 之前的版本中, `wheelDelta` 值的正负号是颠倒的。如果你打算支持早期的 Opera 版本, 就需要使用浏览器检测技术来确定实际的值, 如下面的例子所示。

```
EventUtil.addHandler(document, "mousewheel", function(event){
    event = EventUtil.getEvent(event);
    var delta = (client.engine.opera && client.engine.opera < 9.5 ?
        -event.wheelDelta : event.wheelDelta);
    alert(delta);
});
```


以上代码使用第 9 章创建的 `client` 对象检测了浏览器是不是早期版本的 Opera。



由于 `mousewheel` 事件非常流行，而且所有浏览器都支持它，所以 HTML 5 也加入了该事件。

Firefox 支持一个名为 `DOMMouseScroll` 的类似事件，也是在鼠标滚轮滚动时触发。与 `mousewheel` 事件一样，`DOMMouseScroll` 也被视为鼠标事件，因而包含与鼠标事件有关的所有属性。而有关鼠标滚轮的信息则保存在 `detail` 属性中，当向前滚动鼠标滚轮时，这个属性的值是 -3 的倍数，当向后滚动鼠标滚轮时，这个属性的值是 3 的倍数。图 13-7 展示了这个属性。

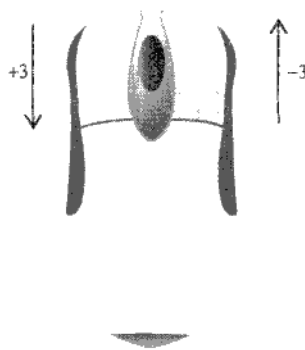


图 13-7

可以将 `DOMMouseScroll` 事件添加到页面中的任何元素，而且该事件会冒泡到 `window` 对象。因此，可以像下面这样针对这个事件来添加事件处理程序。

```
EventUtil.addHandler(window, "DOMMouseScroll", function(event){
    event = EventUtil.getEvent(event);
    alert(event.detail);
});
```

[DOMMouseScrollEventExample01.htm](#)

这个简单的事件处理程序会在鼠标滚轮滚动时显示 `detail` 属性的值。

若要给出跨浏览器环境下的解决方案，第一步就是创建一个能够取得鼠标滚轮增量值（`delta`）的方法。下面是我们添加到 `EventUtil` 对象中的这个方法。

```
var EventUtil = {
    //省略了其他代码

    getWheelDelta: function(event){
        if (event.wheelDelta){
            return (client.engine.opera && client.engine.opera < 9.5 ?
                -event.wheelDelta : event.wheelDelta);
        } else {
            return -event.detail * 40;
        }
    }
};
```

```
},  
  
//省略了其他代码  
};
```

EventUtil.js

这里，`getWheelDelta()`方法首先检测了事件对象是否包含 `wheelDelta` 属性，如果是则通过浏览器检测代码确定正确的值。如果 `wheelDelta` 不存在，则假设相应的值保存在 `detail` 属性中。由于 Firefox 的值有所不同，因此首先要将这个值的符号反向，然后再乘以 40，就可以保证与其他浏览器的值相同了。有了这个方法之后，就可以将相同的事件处理程序指定给 `mousewheel` 和 `DOMMouseScroll` 事件了，例如：

```
(function(){  
  
    function handleMouseWheel(event){  
        event = EventUtil.getEvent(event);  
        var delta = EventUtil.getWheelDelta(event);  
        alert(delta);  
    }  
  
    EventUtil.addHandler(document, "mousewheel", handleMouseWheel);  
    EventUtil.addHandler(document, "DOMMouseScroll", handleMouseWheel);  
  
})();
```

CrossBrowserMouseWheelExample01.htm

我们将相关代码放在了一个私有作用域中，从而不会让新定义的函数干扰全局作用域。这里定义的 `handleMouseWheel()` 函数可以用作两个事件的处理程序（如果指定的事件不存在，则为该事件指定处理程序的代码就会静默地失败）。由于使用了 `EventUtil.getWheelDelta()` 方法，我们定义的这个事件处理程序函数可以适用于任何一种情况。

9. 触摸设备

iOS 和 Android 设备的实现非常特别，因为这些设备没有鼠标。在面向 iPhone 和 iPod 中的 Safari 开发时，要记住以下几点。

- ❑ 不支持 `dblclick` 事件。双击浏览器窗口会放大画面，而且没有办法改变该行为。
- ❑ 轻击可单击元素会触发 `mousemove` 事件。如果此操作会导致内容变化，将不再有其他事件发生；如果屏幕没有因此变化，那么会依次发生 `mousedown`、`mouseup` 和 `click` 事件。轻击不可单击的元素不会触发任何事件。可单击的元素是指那些单击可产生默认操作的元素（如链接），或者那些已经被指定了 `onclick` 事件处理程序的元素。
- ❑ `mousemove` 事件也会触发 `mouseover` 和 `mouseout` 事件。
- ❑ 两个手指放在屏幕上且页面随手指移动而滚动时会触发 `mousewheel` 和 `scroll` 事件。

10. 无障碍性问题

如果你的 Web 应用程序或网站要确保残疾人特别是那些使用屏幕阅读器的人都能访问，那么在使用鼠标事件时就要格外小心。前面提到过，可以通过键盘上的回车键来触发 `click` 事件，但其他鼠标事件却无法通过键盘来触发。为此，我们不建议使用 `click` 之外的其他鼠标事件来展示功能或引发代

码执行。因为这样会给盲人或视障用户造成极大不便。以下是在使用鼠标事件时应当注意的几个易访问性问题。

- ❑ 使用 click 事件执行代码。有人指出通过 onmousedown 执行代码会让人觉得速度更快，对视力正常的人来说这是没错的。但是，在屏幕阅读器中，由于无法触发 mousedown 事件，结果就会造成代码无法执行。
- ❑ 不要使用 onmouseover 向用户显示新的选项。原因同上，屏幕阅读器无法触发这个事件。如果确实非要通过这种方式来显示新选项，可以考虑添加显示相同信息的键盘快捷方式。
- ❑ 不要使用 dblclick 执行重要的操作。键盘无法触发这个事件。

遵照以上提示可以极大地提升残疾人在访问你的 Web 应用程序或网站时的易访问性。



要了解如何在网页中实现无障碍访问的内容，请访问 www.webaim.org 和 <http://yaccessibilityblog.com/>。

13.4.4 键盘与文本事件

用户在使用键盘时会触发键盘事件。“DOM2 级事件”最初规定了键盘事件，但在最终定稿之前又删除了相应内容。结果，对键盘事件的支持主要遵循的是 DOM0 级。

“DOM3 级事件”为键盘事件制定了规范，IE9 率先完全实现了该规范。其他浏览器也在着手实现这一标准，但仍然有很多遗留的问题。

有 3 个键盘事件，简述如下。

- ❑ keydown：当用户按下键盘上的任意键时触发，而且如果按住不放的话，会重复触发此事件。
- ❑ keypress：当用户按下键盘上的字符键时触发，而且如果按住不放的话，会重复触发此事件。

按下 Esc 键也会触发这个事件。Safari 3.1 之前的版本也会在用户按下非字符键时触发 keypress 事件。

- ❑ keyup：当用户释放键盘上的键时触发。

虽然所有元素都支持以上 3 个事件，但只有在用户通过文本框输入文本时才最常用到。

只有一个文本事件：textInput。这个事件是对 keypress 的补充，用意是在将文本显示给用户之前更容易拦截文本。在文本插入文本框之前会触发 textInput 事件。

在用户按了一下键盘上的字符键时，首先会触发 keydown 事件，然后紧接着是 keypress 事件，最后会触发 keyup 事件。其中，keydown 和 keypress 都是在文本框发生变化之前被触发的；而 keyup 事件则是在文本框已经发生变化之后被触发的。如果用户按下了一个字符键不放，就会重复触发 keydown 和 keypress 事件，直到用户松开该键为止。

如果用户按下的是一个非字符键，那么首先会触发 keydown 事件，然后就是 keyup 事件。如果按住这个非字符键不放，那么就会一直重复触发 keydown 事件，直到用户松开这个键，此时会触发 keyup 事件。



键盘事件与鼠标事件一样，都支持相同的修改键。而且，键盘事件的事件对象中也有 shiftKey、ctrlKey、altKey 和 metaKey 属性。IE 不支持 metaKey。

1. 键码

在发生 keydown 和 keyup 事件时, event 对象的 keyCode 属性中会包含一个代码, 与键盘上一个特定的键对应。对数字字母字符键, keyCode 属性的值与 ASCII 码中对应小写字母或数字的编码相同。因此, 数字键 7 的 keyCode 值为 55, 而字母 A 键的 keyCode 值为 65——与 Shift 键的状态无关。DOM 和 IE 的 event 对象都支持 keyCode 属性。请看下面这个例子:

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keyup", function(event){
    event = EventUtil.getEvent(event);
    alert(event.keyCode);
});
```

[KeyUpEventExample01.htm](#)

在这个例子中, 用户每次在文本框中按键触发 keyup 事件时, 都会显示 keyCode 的值。下表列出了所有非字符键的键码。

键	键 码	键	键 码
退格 (Backspace)	8	数字小键盘1	97
制表 (Tab)	9	数字小键盘2	98
回车 (Enter)	13	数字小键盘3	99
上档 (Shift)	16	数字小键盘4	100
控制 (Ctrl)	17	数字小键盘5	101
Alt	18	数字小键盘6	102
暂停/中断 (Pause/Break)	19	数字小键盘7	103
大写锁定 (Caps Lock)	20	数字小键盘8	104
退出 (Esc)	27	数字小键盘9	105
上翻页 (Page Up)	33	数字小键盘+	107
下翻页 (Page Down)	34	数字小键盘及大键盘上的-	109
结尾 (End)	35	数字小键盘.	110
开头 (Home)	36	数字小键盘 /	111
左箭头 (Left Arrow)	37	F1	112
上箭头 (Up Arrow)	38	F2	113
右箭头 (Right Arrow)	39	F3	114
下箭头 (Down Arrow)	40	F4	115
插入 (Ins)	45	F5	116
删除 (Del)	46	F6	117
左Windows键	91	F7	118
右Windows键	92	F8	119
上下文菜单键	93	F9	120
数字小键盘0	96	F10	121

键	键 码	键	键 码
F11	122	正斜杠	191
F12	123	沉音符(`)	192
数字锁 (Num Lock)	144	等于	61
滚动锁 (Scroll Lock)	145	左方括号	219
分号(IE/Safari/Chrome中)	186	反斜杠(\)	220
分号(Opera/FF中)	59	右方括号	221
小于	188	单引号	222
大于	190		

无论 keydown 或 keyup 事件都会存在的一些特殊情况。在 Firefox 和 Opera 中,按分号键时 keyCode 值为 59,也就是 ASCII 中分号的编码;但 IE 和 Safari 返回 186,即键盘中按键的键码。

2. 字符编码

发生 keypress 事件意味着按下的键会影响到屏幕中文本的显示。在所有浏览器中,按下能够插入或删除字符的键都会触发 keypress 事件;按下其他键能否触发此事件因浏览器而异。由于截止到 2008 年,尚无浏览器实现“DOM3 级事件”规范,所以浏览器之间的键盘事件并没有多大的差异。

IE9、Firefox、Chrome 和 Safari 的 event 对象都支持一个 charCode 属性,这个属性只有在发生 keypress 事件时才包含值,而且这个值是按下的那个键所代表字符的 ASCII 编码。此时的 keyCode 通常等于 0 或者也可能等于所按键的键码。IE8 及之前版本和 Opera 则是在 keyCode 中保存字符的 ASCII 编码。要想以跨浏览器的方式取得字符编码,必须首先检测 charCode 属性是否可用,如果不可用则使用 keyCode,如下面的例子所示。

```
var EventUtil = {
    //省略的代码

    getCharCode: function(event){
        if (typeof event.charCode == "number"){
            return event.charCode;
        } else {
            return event.keyCode;
        }
    },
    //省略的代码
};
```

EventUtil.js

这个方法首先检测 charCode 属性是否包含数值(在不支持这个属性的浏览器中,值为 undefined),如果是,则返回该值。否则,就返回 keyCode 属性值。下面是使用这个方法的示例。

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
```

```
    alert(EventUtil.getCharCode(event));  
});
```

KeyPressEventExample01.htm


在取得了字符编码之后，就可以使用 `String.fromCharCode()` 将其转换成实际的字符。

3. DOM3 级变化

尽管所有浏览器都实现了某种形式的键盘事件，DOM3 级事件还是做出了一些改变。比如，DOM3 级事件中的键盘事件，不再包含 `keyCode` 属性，而是包含两个新属性：`key` 和 `char`。

其中，`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（如“k”或“M”）；在按下非字符键时，`key` 的值是相应键的名（如“Shift”或“Down”）。而 `char` 属性在按下字符键时的行为与 `key` 相同，但在按下非字符键时值为 `null`。

IE9 支持 `key` 属性，但不支持 `char` 属性。Safari 5 和 Chrome 支持名为 `keyIdentifier` 的属性，在按下非字符键（例如 Shift）的情况下与 `key` 的值相同。对于字符键，`keyIdentifier` 返回一个格式类似“U+0000”的字符串，表示 Unicode 值。



```
var textbox = document.getElementById("myText");  
EventUtil.addHandler(textbox, "keypress", function(event){  
    event = EventUtil.getEvent(event);  
    var identifier = event.key || event.keyIdentifier;  
    if (identifier){  
        alert(identifier);  
    }  
});
```

DOMLevel3KeyPropertyExample01.htm

由于存在跨浏览器问题，因此本书不推荐使用 `key`、`keyIdentifier` 或 `char`。

DOM3 级事件还添加了一个名为 `location` 的属性，这是一个数值，表示按下了什么位置上的键：0 表示默认键盘，1 表示左侧位置（例如左位的 Alt 键），2 表示右侧位置（例如右侧的 Shift 键），3 表示数字小键盘，4 表示移动设备键盘（也就是虚拟键盘），5 表示手柄（如任天堂 Wii 控制器）。IE9 支持这个属性。Safari 和 Chrome 支持名为 `keyLocation` 的等价属性，但即有 bug——值始终是 0，除非按下了数字键盘（此时，值为 3）；否则，不会是 1、2、4、5。

```
var textbox = document.getElementById("myText");  
EventUtil.addHandler(textbox, "keypress", function(event){  
    event = EventUtil.getEvent(event);  
    var loc = event.location || event.keyLocation;  
    if (loc){  
        alert(loc);  
    }  
});
```

DOMLevel3LocationPropertyExample01.htm

与 `key` 属性一样，支持 `location` 的浏览器也不多，所以在跨浏览器开发中不推荐使用。

最后是给 `event` 对象添加了 `getModifierState()` 方法。这个方法接收一个参数，即等于 `Shift`、`Control`、`AltGraph` 或 `Meta` 的字符串，表示要检测的修改键。如果指定的修改键是活动的（也就是处于被按下的状态），这个方法返回 `true`，否则返回 `false`。



```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    if (event.getModifierState()){
        alert(event.getModifierState("Shift"));
    }
});
```

DOMLevel3LocationGetModifierStateExample01.htm

实际上，通过 event 对象的 shiftKey、altKey、ctrlKey 和 metaKey 属性已经可以取得类似的属性了。IE9 是唯一支持 getModifierState() 方法的浏览器。

4. textInput 事件

“DOM3 级事件”规范中引入了一个新事件，名叫 textInput。根据规范，当用户在可编辑区域中输入字符时，就会触发这个事件。这个用于替代 keypress 的 textInput 事件的行为稍有不同。区别之一就是任何可以获得焦点的元素都可以触发 keypress 事件，但只有可编辑区域才能触发 textInput 事件。区别之二是 textInput 事件只会在用户按下能够输入实际字符的键时才会被触发，而 keypress 事件则在按下那些能够影响文本显示的键时也会触发（例如退格键）。

由于 textInput 事件主要考虑的是字符，因此它的 event 对象中还包含一个 data 属性，这个属性的值就是用户输入的字符（而非字符编码）。换句话说，用户在没有按下档键的情况下按下了 S 键，data 的值就是“s”，而如果在按住上档键时按下该键，data 的值就是“S”。

以下是一个使用 textInput 事件的例子：

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "textInput", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});
```

TextInputEventExample01.htm

在这个例子中，插入到文本框中的字符会通过一个警告框显示出来。

另外，event 对象上还有一个属性，叫 inputMethod，表示把文本输入到文本框中的方式。

- ☐ 0，表示浏览器不确定是怎么输入的。
- ☐ 1，表示是使用键盘输入的。
- ☐ 2，表示文本是粘贴进来的。
- ☐ 3，表示文本是拖放进来的。
- ☐ 4，表示文本是使用 IME 输入的。
- ☐ 5，表示文本是通过在表单中选择某一项输入的。
- ☐ 6，表示文本是通过手写输入的（比如使用手写笔）。
- ☐ 7，表示文本是通过语音输入的。
- ☐ 8，表示文本是通过几种方法组合输入的。
- ☐ 9，表示文本是通过脚本输入的。

使用这个属性可以确定文本是如何输入到控件中的，从而可以验证其有效性。支持 textInput 属性的浏览器有 IE9+、Safari 和 Chrome。只有 IE 支持 inputMethod 属性。

5. 设备中的键盘事件

任天堂 Wii 会在用户按下 Wii 遥控器上的按键时触发键盘事件。尽管没有办法访问 Wii 遥控器中的所有按键，但还是有一些键可以触发键盘事件。图 13-6 展示了一些键的键码，通过这些键码可以知道用户按下了哪个键。

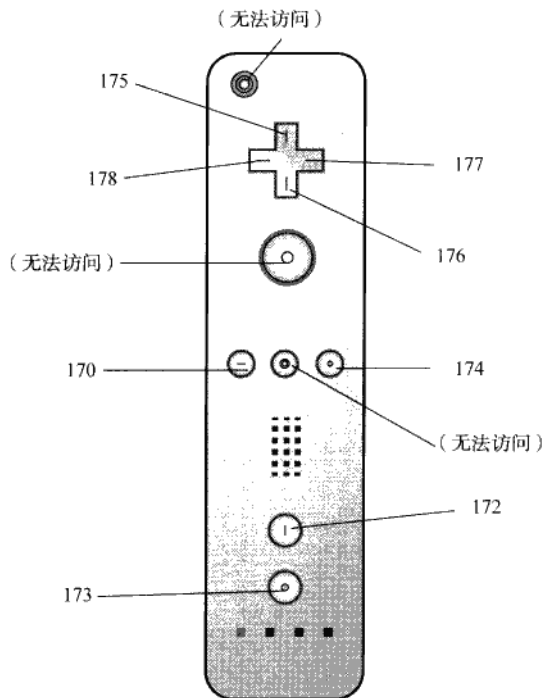


图 13-8

当用户按下十字键盘（键码为 175 ~ 178）、减号（170）、加号（174）、1（172）或 2（173）键时就会触发键盘事件。但没有办法得知用户是否按下了电源开关、A、B 或主页键。

iOS 版 Safari 和 Android 版 WebKit 在使用屏幕键盘时会触发键盘事件。

13.4.5 复合事件

复合事件（composition event）是 DOM3 级事件中新添加的一类事件，用于处理 IME 的输入序列。IME（Input Method Editor，输入法编辑器）可以让用户输入在物理键盘上找不到的字符。例如，使用拉丁文键盘的用户通过 IME 照样能输入日文字符。IME 通常需要同时按住多个键，但最终只输入一个字符。复合事件就是针对检测和处理这种输入而设计的。有以下三种复合事件。

❑ compositionstart：在 IME 的文本复合系统打开时触发，表示要开始输入了。

❑ `compositionupdate`: 在向输入字段中插入新字符时触发。

❑ `compositionend`: 在 IME 的文本复合系统关闭时触发, 表示返回正常键盘输入状态。


复合事件与文本事件在很多方面都很相似。在触发复合事件时, 目标是接收文本的输入字段。但它比文本事件的事件对象多一个属性 `data`, 其中包含以下几个值中的一个:

❑ 如果在 `compositionstart` 事件发生时访问, 包含正在编辑的文本 (例如, 已经选中的需要马上替换的文本);

❑ 如果在 `compositionupdate` 事件发生时访问, 包含正插入的新字符;

❑ 如果在 `compositionend` 事件发生时访问, 包含此次输入会话中插入的所有字符。

与文本事件一样, 必要时可以利用复合事件来筛选输入。可以像下面这样使用它们:



```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "compositionstart", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});

EventUtil.addHandler(textbox, "compositionupdate", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});

EventUtil.addHandler(textbox, "compositionend", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});
```

[CompositionEventsExample01.htm](#)

IE9+是到 2011 年唯一支持复合事件的浏览器。由于缺少支持, 对于需要开发跨浏览器应用的人员, 它的用处不大。要确定浏览器是否支持复合事件, 可以使用以下代码:

```
var isSupported = document.implementation.hasFeature("CompositionEvent", "3.0");
```

13.4.6 变动事件

DOM2 级的变动 (mutation) 事件能在 DOM 中的某一部分发生变化时给出提示。变动事件是为 XML 或 HTML DOM 设计的, 并不特定于某种语言。DOM2 级定义了如下变动事件。

❑ `DOMSubtreeModified`: 在 DOM 结构中发生任何变化时触发。这个事件在其他任何事件触发后都会触发。

❑ `DOMNodeInserted`: 在一个节点作为子节点被插入到另一个节点中时触发。

❑ `DOMNodeRemoved`: 在节点从其父节点中被移除时触发。

❑ `DOMNodeInsertedIntoDocument`: 在一个节点被直接插入文档或通过子树间接插入文档之后触发。这个事件在 `DOMNodeInserted` 之后触发。

❑ `DOMNodeRemovedFromDocument`: 在一个节点被直接从文档中移除或通过子树间接从文档中移除之前触发。这个事件在 `DOMNodeRemoved` 之后触发。

❑ `DOMAttrModified`: 在特性被修改之后触发。

❑ `DOMCharacterDataModified`: 在文本节点的值发生变化时触发。

使用下列代码可以检测出浏览器是否支持变动事件：

```
var isSupported = document.implementation.hasFeature("MutationEvents", "2.0");
```

IE8 及更早版本不支持任何变动事件。下表列出了不同浏览器对不同变动事件的支持情况。

事 件	Opera 9+	Firefox 3+	Safari 3+及Chrome	IE9+
DOMSubtreeModified	~	支持	支持	支持
DOMNodeInserted	支持	支持	支持	支持
DOMNodeRemoved	支持	支持	支持	支持

由于 DOM3 级事件模块作废了很多变动事件，所以本节只介绍那些将来仍然会得到支持的事件。

1. 删除节点

在使用 `removeChild()` 或 `replaceChild()` 从 DOM 中删除节点时，首先会触发 `DOMNodeRemoved` 事件。这个事件的目标 (`event.target`) 是被删除的节点，而 `event.relatedNode` 属性中包含着对目标节点父节点的引用。在这个事件触发时，节点尚未从其父节点删除，因此其 `parentNode` 属性仍然指向父节点（与 `event.relatedNode` 相同）。这个事件会冒泡，因而可以在 DOM 的任何层次上面处理它。

如果被移除的节点包含子节点，那么在其所有子节点以及这个被移除的节点上会相继触发 `DOMNodeRemovedFromDocument` 事件。但这个事件不会冒泡，所以只有直接指定给其中一个子节点的事件处理程序才会被调用。这个事件的目标是相应的子节点或者那个被移除的节点，除此之外 `event` 对象中不包含其他信息。

紧随其后触发的是 `DOMSubtreeModified` 事件。这个事件的目标是被移除节点的父节点；此时的 `event` 对象也不会提供与事件相关的其他信息。

为了理解上述事件的触发过程，下面我们就以一个简单的 HTML 页面为例。

```
<!DOCTYPE html>
<html>
<head>
  <title>Node Removal Events Example</title>
</head>
<body>
  <ul id="myList">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</body>
</html>
```

在这个例子中，我们假设要移除 `` 元素。此时，就会依次触发以下事件。

(1) 在 `` 元素上触发 `DOMNodeRemoved` 事件。`relatedNode` 属性等于 `document.body`。

(2) 在 `` 元素上触发 `DOMNodeRemovedFromDocument` 事件。

(3) 在身为 `` 元素子节点的每个 `` 元素及文本节点上触发 `DOMNodeRemovedFromDocument` 事件。

(4) 在 `document.body` 上触发 `DOMSubtreeModified` 事件，因为 `` 元素是 `document.body` 的直接子元素。

运行下列代码可以验证以上事件发生的顺序。

```
EventUtil.addHandler(window, "load", function(event){
    var list = document.getElementById("myList");

    EventUtil.addHandler(document, "DOMSubtreeModified", function(event){
        alert(event.type);
        alert(event.target);
    });

    EventUtil.addHandler(document, "DOMNodeRemoved", function(event){
        alert(event.type);
        alert(event.target);
        alert(event.relatedNode);
    });

    EventUtil.addHandler(list.firstChild, "DOMNodeRemovedFromDocument", function(event){
        alert(event.type);
        alert(event.target);
    });

    list.parentNode.removeChild(list);
});
```

以上代码为 document 添加了对 DOMSubtreeModified 和 DOMNodeRemoved 事件的处理程序，以便在页面上处理这些事件。由于 DOMNodeRemovedFromDocument 不会冒泡，所以我们将针对它的事件处理程序直接添加给了元素的第一个子节点（在兼容 DOM 的浏览器中是一个文本节点）。在设置了以上事件处理程序后，代码从文档中移除了元素。

2. 插入节点

在使用 appendChild()、replaceChild()或 insertBefore()向 DOM 中插入节点时，首先会触发 DOMNodeInserted 事件。这个事件的目标是被插入的节点，而 event.relatedNode 属性中包含一个对父节点的引用。在这个事件触发时，节点已经被插入到了新的父节点中。这个事件是冒泡的，因此可以在 DOM 的各个层次上处理它。

紧接着，会在新插入的节点上面触发 DOMNodeInsertedIntoDocument 事件。这个事件不冒泡，因此必须在插入节点之前为它添加这个事件处理程序。这个事件的目标是被插入的节点，除此之外 event 对象中不包含其他信息。

最后一个触发的事件是 DOMSubtreeModified，触发于新插入节点的父节点。

我们仍以前面的 HTML 文档为例，可以通过下列 JavaScript 代码来验证上述事件的触发顺序。

```
EventUtil.addHandler(window, "load", function(event){
    var list = document.getElementById("myList");
    var item = document.createElement("li");
    item.appendChild(document.createTextNode("Item 4"));

    EventUtil.addHandler(document, "DOMSubtreeModified", function(event){
        alert(event.type);
        alert(event.target);
    });

    EventUtil.addHandler(document, "DOMNodeInserted", function(event){
        alert(event.type);
        alert(event.target);
        alert(event.relatedNode);
    });

    EventUtil.addHandler(item, "DOMNodeInsertedIntoDocument", function(event){
        alert(event.type);
    });
});
```

```

        alert(event.target);
    });

    list.appendChild(item);
});

```

以上代码首先创建了一个包含文本“Item 4”的新元素。由于 DOMSubtreeModified 和 DOMNodeInserted 事件是冒泡的，所以把它们的事件处理程序添加到了文档中。在将列表项插入到其父节点之前，先将 DOMNodeInsertedIntoDocument 事件的事件处理程序添加给它。最后一步就是使用 appendChild() 来添加这个列表项；此时，事件开始依次被触发。首先是在新元素上触发 DOMNodeInserted 事件，其 relatedNode 是元素。然后是触发新元素上的 DOMNodeInsertedIntoDocument 事件，最后触发的是元素上的 DOMSubtreeModified 事件。

13.4.7 HTML5 事件

DOM 规范没有涵盖所有浏览器支持的所有事件。很多浏览器出于不同的目的——满足用户需求或解决特殊问题，还实现了一些自定义的事件。HTML5 详尽列出了浏览器应该支持的所有事件。本节只讨论其中得到浏览器完善支持的事件，但并非全部事件。（其他事件会在本书其他章节讨论。）

1. contextmenu 事件

Windows 95 在 PC 中引入了上下文菜单的概念，即通过单击鼠标右键可以调出上下文菜单。不久，这个概念也被引入了 Web 领域。为了实现上下文菜单，开发人员面临的主要问题是确定应该显示上下文菜单（在 Windows 中，是右键单击；在 Mac 中，是 Ctrl+单击），以及如何屏蔽与该操作关联的默认上下文菜单。为解决这个问题，就出现了 contextmenu 这个事件，用以表示何时应该显示上下文菜单，以便开发人员取消默认的上下文菜单而提供自定义的菜单。

由于 contextmenu 事件是冒泡的，因此可以为 document 指定一个事件处理程序，用以处理页面中发生的所有此类事件。这个事件的目标是发生用户操作的元素。在所有浏览器中都可以取消这个事件：在兼容 DOM 的浏览器中，使用 event.preventDefault()；在 IE 中，将 event.returnValue 的值设置为 false。因为 contextmenu 事件属于鼠标事件，所以其事件对象中包含与光标位置有关的所有属性。通常使用 contextmenu 事件来显示自定义的上下文菜单，而使用 onclick 事件处理程序来隐藏该菜单。以下面的 HTML 页面为例。

```

<!DOCTYPE html>
<html>
<head>
  <title>ContextMenu Event Example</title>
</head>
<body>
  <div id="myDiv">Right click or Ctrl+click me to get a custom context menu.
    Click anywhere else to get the default context menu.</div>
  <ul id="myMenu" style="position:absolute;visibility:hidden;background-color:
    silver">
    <li><a href="http://www.nczonline.net">Nicholas' site</a></li>
    <li><a href="http://www.wrox.com">Wrox site</a></li>
    <li><a href="http://www.yahoo.com">Yahoo!</a></li>
  </ul>
</body>
</html>

```

这里的<div>元素包含一个自定义的上下文菜单。其中，元素作为自定义上下文菜单，并且在初始时是隐藏的。实现这个例子的 JavaScript 代码如下所示。

```
EventUtil.addHandler(window, "load", function(event){
    var div = document.getElementById("myDiv");

    EventUtil.addHandler(div, "contextmenu", function(event){
        event = EventUtil.getEvent(event);
        EventUtil.preventDefault(event);

        var menu = document.getElementById("myMenu");
        menu.style.left = event.clientX + "px";
        menu.style.top = event.clientY + "px";
        menu.style.visibility = "visible";
    });

    EventUtil.addHandler(document, "click", function(event){
        document.getElementById("myMenu").style.visibility = "hidden";
    });
});
```

ContextMenuEventExample01.htm

在这个例子中，我们为<div>元素添加了 oncontextmenu 事件的处理程序。这个事件处理程序首先会取消默认行为，以保证不显示浏览器默认的上下文菜单。然后，再根据 event 对象 clientX 和 clientY 属性的值，来确定放置元素的位置。最后一步就是通过将 visibility 属性设置为 "visible" 来显示自定义上下文菜单。另外，还为 document 添加了一个 onclick 事件处理程序，以使用户能够通过鼠标单击来隐藏菜单（单击也是隐藏系统上下文菜单的默认操作）。

虽然这个例子很简单，但它却展示了 Web 上所有自定义上下文菜单的基本结构。只需为这个例子中的上下文菜单添加一些 CSS 样式，就可以得到非常棒的效果。

支持 contextmenu 事件的浏览器有 IE、Firefox、Safari、Chrome 和 Opera 11+。

2. beforeunload 事件

之所以有发生在 window 对象上的 beforeunload 事件，是为了让开发人员有可能在页面卸载前阻止这一操作。这个事件会在浏览器卸载页面之前触发，可以通过它来取消卸载并继续使用原有页面。但是，不能彻底取消这个事件，因为那就相当于让用户无法离开当前页面了。为此，这个事件的意图是将控制权交给用户。显示的消息会告知用户页面行将被卸载（正因为如此才会显示这个消息），询问用户是否真的要关闭页面，还是希望继续留下来（见图 13-9）。

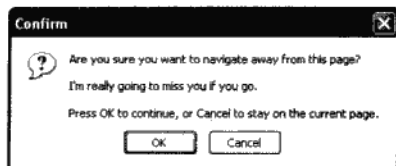


图 13-9

为了显示这个弹出对话框，必须将 event.returnValue 的值设置为要显示给用户的字符串（对 IE 及 Firefox 而言），同时作为函数的值返回（对 Safari 和 Chrome 而言），如下面的例子所示。



```
EventUtil.addHandler(window, "beforeunload", function(event){
    event = EventUtil.getEvent(event);
    var message = "I'm really going to miss you if you go.";
    event.returnValue = message;
    return message;
});
```

BeforeUnloadEventExample01.htm

IE 和 Firefox、Safari 和 Chrome 都支持 beforeunload 事件，也都会弹出这个对话框询问用户是否真想离开。Opera 11 及之前的版本不支持 beforeunload 事件。

3. DOMContentLoaded 事件

如前所述，window 的 load 事件会在页面中的一切都加载完毕时触发，但这个过程可能会因为要加载的外部资源过多而颇费周折。而 DOMContentLoaded 事件则在形成完整的 DOM 树之后就会触发，不理睬图像、JavaScript 文件、CSS 文件或其他资源是否已经下载完毕。与 load 事件不同，DOMContentLoaded 支持在页面下载的早期添加事件处理程序，这也就意味着用户能够尽早地与页面进行交互。

要处理 DOMContentLoaded 事件，可以为 document 或 window 添加相应的事件处理程序（尽管这个事件会冒泡到 window，但它的目标实际上是 document）。来看下面的例子。

```
EventUtil.addHandler(document, "DOMContentLoaded", function(event){
    alert("Content loaded");
});
```

DOMContentLoadedEventExample01.htm

DOMContentLoaded 事件对象不会提供任何额外的信息（其 target 属性是 document）。

IE9+、Firefox、Chrome、Safari 3.1+ 和 Opera 9+ 都支持 DOMContentLoaded 事件，通常这个事件既可以添加事件处理程序，也可以执行其他 DOM 操作。这个事件始终都会在 load 事件之前触发。

对于不支持 DOMContentLoaded 的浏览器，我们建议在页面加载期间设置一个时间为 0 毫秒的超时调用，如下面的例子所示。

```
setTimeout(function(){
    //在此添加事件处理程序
}, 0);
```

这段代码的实际意思就是：“在当前 JavaScript 处理完成后立即运行这个函数。”在页面下载和构建期间，只有一个 JavaScript 处理过程，因此超时调用会在该过程结束时立即触发。至于这个时间与 DOMContentLoaded 被触发的时间能否同步，主要还是取决于用户使用的浏览器和页面中的其他代码。为了确保这个方法有效，必须将其作为页面中的第一个超时调用；即便如此，也还是无法保证在所有环境中该超时调用一定会早于 load 事件被触发。

4. readystatechange 事件

IE 为 DOM 文档中的某些部分提供了 readystatechange 事件。这个事件的目的是提供与文档或元素的加载状态有关的信息，但这个事件的行为有时候也很难预料。支持 readystatechange 事件的每个对象都有一个 readyState 属性，可能包含下列 5 个值中的一个。

□ uninitialized（未初始化）：对象存在但尚未初始化。

- ❑ loading (正在加载): 对象正在加载数据。
- ❑ loaded (加载完毕): 对象加载数据完成。
- ❑ interactive (交互): 可以操作对象了, 但还没有完全加载。
- ❑ complete (完成): 对象已经加载完毕。

这些状态看起来很直观, 但并非所有对象都会经历 readyState 的这几个阶段。换句话说, 如果某个阶段不适用某个对象, 则该对象完全可能跳过该阶段; 并没有规定哪个阶段适用于哪个对象。显然, 这意味着 readystatechange 事件经常会少于 4 次, 而 readyState 属性的值也不总是连续的。

对于 document 而言, 值为 "interactive" 的 readyState 会在与 DOMContentLoaded 大致相同的时刻触发 readystatechange 事件。此时, DOM 树已经加载完毕, 可以安全地操作它了, 因此就会进入交互 (interactive) 阶段。但与此同时, 图像及其他外部文件不一定可用。下面来看一段处理 readystatechange 事件的代码。

```
EventUtil.addHandler(document, "readystatechange", function(event){
    if (document.readyState == "interactive"){
        alert("Content loaded");
    }
});
```

这个事件的 event 对象不会提供任何信息, 也没有目标对象。

在与 load 事件一起使用时, 无法预测两个事件触发的先后顺序。在包含较多或较大的外部资源的页面中, 会在 load 事件触发之前先进入交互阶段; 而在包含较少或较小的外部资源的页面中, 则很难说 readystatechange 事件会发生在 load 事件前面。

让问题变得更复杂的是, 交互阶段可能会早于也可能晚于完成阶段出现, 无法确保顺序。在包含较多外部资源的页面中, 交互阶段更有可能早于完成阶段出现; 而在页面中包含较少外部资源的情况下, 完成阶段先于交互阶段出现的可能性更大。因此, 为了尽可能抢到先机, 有必要同时检测交互和完成阶段, 如下面的例子所示。

```
EventUtil.addHandler(document, "readystatechange", function(event){
    if (document.readyState == "interactive" || document.readyState == "complete"){
        EventUtil.removeHandler(document, "readystatechange", arguments.callee);
        alert("Content loaded");
    }
});
```


对于上面的代码来说, 当 readystatechange 事件触发时, 会检测 document.readyState 的值, 看当前是否已经进入交互阶段或完成阶段。如果是, 则移除相应的事件处理程序以免在其他阶段再执行。注意, 由于事件处理程序使用的是匿名函数, 因此这里使用了 arguments.callee 来引用该函数。然后, 会显示一个警告框, 说明内容已经加载完毕。这样编写代码可以达到与使用 DOMContentLoaded 十分相近的效果。

支持 readystatechange 事件的浏览器有 IE、Firefox 4+ 和 Opera。



虽然使用 readystatechange 可以十分近似地模拟 DOMContentLoaded 事件, 但它们本质上还是不同的。在不同页面中, load 事件与 readystatechange 事件并不能保证以相同的顺序触发。

另外, <script> (在 IE 和 Opera 中) 和 <link> (仅 IE 中) 元素也会触发 readystatechange 事件, 可以用来确定外部的 JavaScript 和 CSS 文件是否已经加载完成。与在其他浏览器中一样, 除非把动态创建的元素添加到页面中, 否则浏览器不会开始下载外部资源。基于元素触发的 readystatechange 事件也存在同样的问题, 即 readyState 属性无论等于 "loaded" 还是 "complete" 都可以表示资源已经可用。有时候, readyState 会停在 "loaded" 阶段而永远不会 "完成"; 有时候, 又会跳过 "loaded" 阶段而直接 "完成"。于是, 还需要像对待 document 一样采取相同的编码方式。例如, 下面展示了一段加载外部 JavaScript 文件的代码。



```
EventUtil.addHandler(window, "load", function(){
    var script = document.createElement("script");


    EventUtil.addHandler(script, "readystatechange", function(event){
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        if (target.readyState == "loaded" || target.readyState == "complete"){
            EventUtil.removeHandler(target, "readystatechange", arguments.callee);
            alert("Script Loaded");
        }
    });
    script.src = "example.js";
    document.body.appendChild(script);
});
```

ReadyStateChangeEventExample01.htm

这个例子为新创建的 <script> 节点指定了一个事件处理程序。事件的目标是该节点本身, 因此当触发 readystatechange 事件时, 要检测目标的 readyState 属性是不是等于 "loaded" 或 "complete"。如果进入了其中任何一个阶段, 则移除事件处理程序 (以防止被执行两次), 并显示一个警告框。与此同时, 就可以执行已经加载完毕的外部文件中的函数了。

同样的编码方式也适用于通过 <link> 元素加载 CSS 文件的情况, 如下面的例子所示。



```
EventUtil.addHandler(window, "load", function(){
    var link = document.createElement("link");
    link.type = "text/css";
    link.rel = "stylesheet";

    EventUtil.addHandler(link, "readystatechange", function(event){
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        if (target.readyState == "loaded" || target.readyState == "complete"){
            EventUtil.removeHandler(target, "readystatechange", arguments.callee);
            alert("CSS Loaded");
        }
    });

    link.href = "example.css";
    document.getElementsByTagName("head")[0].appendChild(link);
});
```

ReadyStateChangeEventExample02.htm

同样,最重要的是要一并检测 `readyState` 的两个状态,并在调用了一次事件处理程序后就将其移除。

5. `pageshow` 和 `pagehide` 事件

Firefox 和 Opera 有一个特性,名叫“往返缓存”(back-forward cache, 或 `bfcache`),可以在用户使用浏览器的“后退”和“前进”按钮时加快页面的转换速度。这个缓存中不仅保存着页面数据,还保存了 DOM 和 JavaScript 的状态;实际上是将整个页面都保存在了内存里。如果页面位于 `bfcache` 中,那么再次打开该页面时就不会触发 `load` 事件。尽管由于内存中保存了整个页面的状态,不触发 `load` 事件也不应该会导致什么问题,但为了更形象地说明 `bfcache` 的行为,Firefox 还是提供了一些新事件。

第一个事件就是 `pageshow`, 这个事件在页面显示时触发,无论该页面是否来自 `bfcache`。在重新加载的页面中, `pageshow` 会在 `load` 事件触发后触发;而对于 `bfcache` 中的页面, `pageshow` 会在页面状态完全恢复的那一刻触发。另外要注意的是,虽然这个事件的目标是 `document`,但必须将其事件处理程序添加到 `window`。来看下面的例子。

```
(function(){
    var showCount = 0;

    EventUtil.addHandler(window, "load", function(){
        alert("Load fired");
    });

    EventUtil.addHandler(window, "pageshow", function(){
        showCount++;
        alert("Show has been fired " + showCount + " times.");
    });
})();
```

这个例子使用了私有作用域,以防止变量 `showCount` 进入全局作用域。当页面首次加载完成时, `showCount` 的值为 0。此后,每当触发 `pageshow` 事件, `showCount` 的值就会递增并通过警告框显示出来。如果你在离开包含以上代码的页面之后,又单击“后退”按钮返回该页面,就会看到 `showCount` 每次递增的值。这是因为该变量的状态,乃至整个页面的状态,都被保存在了内存中,当你返回这个页面时,它们的状态得到了恢复。如果你单击了浏览器的“刷新”按钮,那么 `showCount` 的值就会被重置为 0,因为页面已经完全重新加载了。

除了通常的属性之外, `pageshow` 事件的 `event` 对象还包含一个名为 `persisted` 的布尔值属性。如果页面被保存在了 `bfcache` 中,则这个属性的值为 `true`;否则,这个属性的值为 `false`。可以像下面这样在事件处理程序中检测这个属性。

```
(function(){
    var showCount = 0;

    EventUtil.addHandler(window, "load", function(){
        alert("Load fired");
    });

    EventUtil.addHandler(window, "pageshow", function(){
        showCount++;
        alert("Show has been fired " + showCount +
            " times. Persisted? " + event.persisted);
    });
})();
```

通过检测 `persisted` 属性，就可以根据页面在 `bfcache` 中的状态来确定是否需要采取其他操作。

与 `pageshow` 事件对应的是 `pagehide` 事件，该事件会在浏览器卸载页面的时候触发，而且是在 `unload` 事件之前触发。与 `pageshow` 事件一样，`pagehide` 在 `document` 上面触发，但其事件处理程序必须要添加到 `window` 对象。这个事件的 `event` 对象也包含 `persisted` 属性，不过其用途稍有不同。来看下面的例子。

```
EventUtil.addHandler(window, "pagehide", function(event){
    alert("Hiding. Persisted? " + event.persisted);
});
```

PageShowEventExample01.htm

有时候，可能需要在 `pagehide` 事件触发时根据 `persisted` 的值采取不同的操作。对于 `pageshow` 事件，如果页面是从 `bfcache` 中加载的，那么 `persisted` 的值就是 `true`；对于 `pagehide` 事件，如果页面在卸载之后会被保存在 `bfcache` 中，那么 `persisted` 的值也会被设置为 `true`。因此，当第一次触发 `pageshow` 时，`persisted` 的值一定是 `false`，而在第一次触发 `pagehide` 时，`persisted` 就会变成 `true`（除非页面不会被保存在 `bfcache` 中）。

支持 `pageshow` 和 `pagehide` 事件的浏览器有 Firefox、Safari 5+、Chrome 和 Opera。IE9 及之前版本不支持这两个事件。

指定了 `onunload` 事件处理程序的页面会被自动排除在 `bfcache` 之外，即使事件处理程序是空的。原因在于，`onunload` 最常用于撤销在 `onload` 中所执行的操作，而跳过 `onload` 后再次显示页面很可能就会导致页面不正常。

6. hashchange 事件

HTML5 新增了 `hashchange` 事件，以便在 URL 的参数列表（及 URL 中“#”号后面的所有字符串）发生变化时通知开发人员。之所以新增这个事件，是因为在 Ajax 应用中，开发人员经常要利用 URL 参数列表来保存状态或导航信息。

必须要把 `hashchange` 事件处理程序添加给 `window` 对象，然后 URL 参数列表只要变化就会调用它。此时的 `event` 对象应该额外包含两个属性：`oldURL` 和 `newURL`。这两个属性分别保存着参数列表变化前后的完整 URL。例如：

```
EventUtil.addHandler(window, "hashchange", function(event){
    alert("Old URL: " + event.oldURL + "\nNew URL: " + event.newURL);
});
```

HashChangeEventExample01.htm

支持 `hashchange` 事件的浏览器有 IE8+、Firefox 3.6+、Safari 5+、Chrome 和 Opera 10.6+。在这些浏览器中，只有 Firefox 6+、Chrome 和 Opera 支持 `oldURL` 和 `newURL` 属性。为此，最好是使用 `location` 对象来确定当前的参数列表。

```
EventUtil.addHandler(window, "hashchange", function(event){
    alert("Current hash: " + location.hash);
});
```

使用以下代码可以检测浏览器是否支持 haschange 事件：

```
var isSupported = ("onhashchange" in window); //这里有 bug
```

如果 IE8 是在 IE7 文档模式下运行，即使功能无效它也会返回 true。为解决这个问题，可以使用以下这个更稳妥的检测方式：

```
var isSupported = ("onhashchange" in window) && (document.documentMode ===  
    undefined || document.documentMode > 7);
```

13.4.8 设备事件

智能手机和平板电脑的普及，为用户与浏览器交互引入了一种新的方式，而一类新事件也应运而生。设备事件（device event）可以让开发人员确定用户在怎样使用设备。W3C 从 2011 年开始着手制定一份关于设备事件的新草案（<http://dev.w3.org/geo/api/spec-source-orientation.html>），以涵盖不断增长的设备类型并为它们定义相关的事件。本节会同时讨论这份草案中涉及的 API 和特定于浏览器开发商的事件。

1. orientationchange 事件

苹果公司为移动 Safari 中添加了 orientationchange 事件，以便开发人员能够确定用户何时将设备由横向查看模式切换为纵向查看模式。移动 Safari 的 window.orientation 属性中可能包含 3 个值：0 表示肖像模式，90 表示向左旋转的横向模式（“主屏幕”按钮在右侧），-90 表示向右旋转的横向模式（“主屏幕”按钮在左侧）。相关文档中还提到一个值，即 180 表示 iPhone 头朝下；但这种模式至今尚未得到支持。图 13-10 展示了 window.orientation 的每个值的含义。

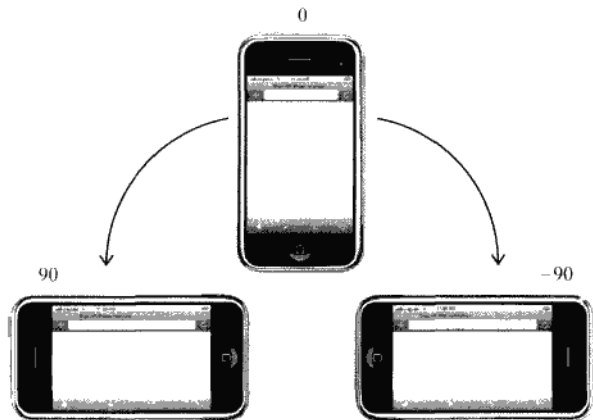


图 13-10

只要用户改变了设备的查看模式，就会触发 orientationchange 事件。此时的 event 对象不包含任何有价值的信息，因为唯一相关的信息可以通过 window.orientation 访问到。下面是使用这个事件的典型示例。

```
EventUtil.addHandler(window, "load", function(event){  
    var div = document.getElementById("myDiv");  
    div.innerHTML = "Current orientation is " + window.orientation;
```



```
EventUtil.addHandler(window, "orientationchange", function(event){
    div.innerHTML = "Current orientation is " + window.orientation;
});
});
```

OrientationChangeEventExample01.htm

在这个例子中,当触发 load 事件时会显示最初的方向信息。然后,添加了处理 orientationchange 事件的处理程序。只要发生这个事件,就会有表示新方向的信息更新页面中的消息。

所有 iOS 设备都支持 orientationchange 事件和 window.orientation 属性。



由于可以将 orientationchange 看成 window 事件,所以也可以通过指定 <body>元素的 onorientationchange 特性来指定事件处理程序。

2. MozOrientation 事件

Firefox 3.6 为检测设备的方向引入了一个名为 MozOrientation 的新事件。(前缀 Moz 表示这是特定于浏览器开发商的事件,不是标准事件。)当设备的加速计检测到设备方向改变时,就会触发这个事件。但这个事件与 iOS 中的 orientationchange 事件不同,该事件只能提供一个平面的方向变化。由于 MozOrientation 事件是在 window 对象上触发的,所以可以使用以下代码来处理。

```
EventUtil.addHandler(window, "MozOrientation", function(event){
    //响应事件
});
```

此时的 event 对象包含三个属性: x、y 和 z。这几个属性的值都介于 1 到 -1 之间,表示不同坐标轴上的方向。在静止状态下, x 值为 0, y 值为 0, z 值为 1 (表示设备处于竖直状态)。如果设备向右倾斜, x 值会减小;反之,向左倾斜, x 值会增大。类似地,如果设备向远离用户的方向倾斜, y 值会减小,向接近用户的方向倾斜, y 值会增大。z 轴检测垂直加速度, 1 表示静止不动,在设备移动时值会减小。(失重状态下值为 0。)以下是输出这三个值的一个简单的例子。

```
EventUtil.addHandler(window, "MozOrientation", function(event){
    var output = document.getElementById("output");
    output.innerHTML = "X=" + event.x + ", Y=" + event.y + ", Z=" + event.z + "<br>";
});
```

MozOrientationEventExample01.htm

只有带加速计的设备才支持 MozOrientation 事件,包括 Macbook、Lenovo Thinkpad、Windows Mobile 和 Android 设备。请大家注意,这是一个实验性 API,将来可能会变(可能会被其他事件取代)。

3. deviceorientation 事件

本质上, DeviceOrientation Event 规范定义的 deviceorientation 事件与 MozOrientation 事件类似。它也是在加速计检测到设备方向变化时在 window 对象上触发,而且具有与 MozOrientation 事件相同的支持限制。不过, deviceorientation 事件的意图是告诉开发人员设备在空间中朝向哪儿,而不是如何移动。

设备在三维空间中是靠 x 、 y 和 z 轴来定位的。当设备静止放在水平表面上时，这三个值都是 0。 x 轴方向是从左往右， y 轴方向是从下往上， z 轴方向是从后往前（参见图 13-11）。

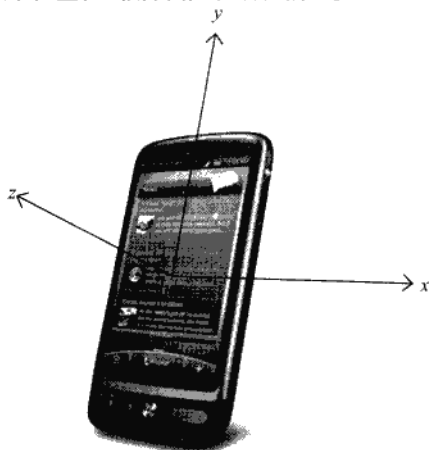


图 13-11

触发 `deviceorientation` 事件时，事件对象中包含着每个轴相对于设备静止状态下发生变化的信息。事件对象包含以下 5 个属性。

- `alpha`: 在围绕 z 轴旋转时（即左右旋转时）， y 轴的度数差；是一个介于 0 到 360 之间的浮点数。
- `beta`: 在围绕 x 轴旋转时（即前后旋转时）， z 轴的度数差；是一个介于 -180 到 180 之间的浮点数。
- `gamma`: 在围绕 y 轴旋转时（即扭转设备时）， z 轴的度数差；是一个介于 -90 到 90 之间的浮点数。
- `absolute`: 布尔值，表示设备是否返回一个绝对值。
- `compassCalibrated`: 布尔值，表示设备的指南针是否校准过。

图 13-12 是 `alpha`、`beta` 和 `gamma` 值含义的示意图。

下面是一个输出 `alpha`、`beta` 和 `gamma` 值的例子。

```
EventUtil.addHandler(window, "deviceorientation", function(event){
    var output = document.getElementById("output");
    output.innerHTML = "Alpha=" + event.alpha + ", Beta=" + event.beta +
        ", Gamma=" + event.gamma + "<br>";
});
```

[DeviceOrientationEventExample01.htm](#)

通过这些信息，可以响应设备的方向，重新排列或修改屏幕上的元素。要响应设备方向的变化而旋转元素，可以参考如下代码。

```
EventUtil.addHandler(window, "deviceorientation", function(event){
    var arrow = document.getElementById("arrow");
    arrow.style.webkitTransform = "rotate(" + Math.round(event.alpha) + "deg)";
});
```

[DeviceOrientationEventExample01.htm](#)

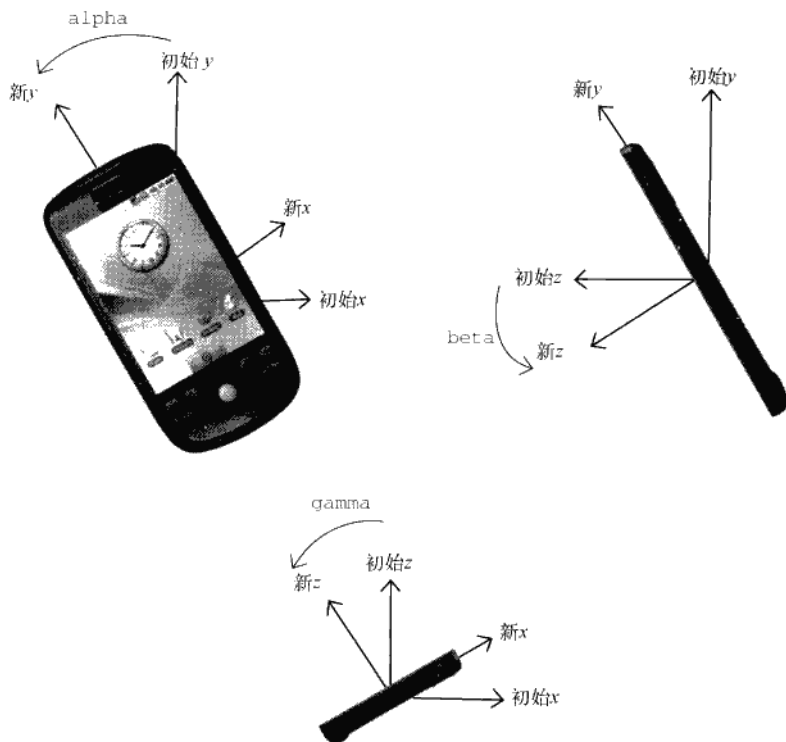


图 13-12

这个例子只能在移动 WebKit 浏览器中运行，因为它使用了专有的 `webkitTransform` 属性（即 CSS 标准属性 `transform` 的临时版）。元素 “arrow” 会随着 `event.alpha` 值的变化而旋转，给人一种指南针的感觉。为了保证旋转平滑，这里的 CSS3 变换使用了舍入之后的值。

到 2011 年，支持 `deviceorientation` 事件的浏览器有 iOS 4.2+ 中的 Safari、Chrome 和 Android 版 WebKit。

4. `devicemotion` 事件

`DeviceOrientation Event` 规范还定义了一个 `devicemotion` 事件。这个事件是要告诉开发人员设备什么时候移动，而不仅仅是设备方向如何改变。例如，通过 `devicemotion` 能够检测到设备是不是正在往下掉，或者是不是被走着的人拿在手里。

触发 `devicemotion` 事件时，事件对象包含以下属性。

- `acceleration`: 一个包含 `x`、`y` 和 `z` 属性的对象，在不考虑重力的情况下，告诉你在每个方向上的加速度。
- `accelerationIncludingGravity`: 一个包含 `x`、`y` 和 `z` 属性的对象，在考虑 `z` 轴自然重力加速度的情况下，告诉你在每个方向上的加速度。
- `interval`: 以毫秒表示的时间值，必须在另一个 `devicemotion` 事件触发前传入。这个值在每个事件中应该是一个常量。

□ `rotationRate`: 一个包含表示方向的 `alpha`、`beta` 和 `gamma` 属性的对象。

如果读取不到 `acceleration`、`accelerationIncludingGravity` 和 `rotationRate` 值, 则它们的值为 `null`。因此, 在使用这三个属性之前, 应该先检测确定它们的值不是 `null`。例如:



```
EventUtil.addHandler(window, "devicemotion", function(event){
    var output = document.getElementById("output");
    if (event.rotationRate !== null){
        output.innerHTML += "Alpha=" + event.rotationRate.alpha + ", Beta=" +
            event.rotationRate.beta + ", Gamma=" +
            event.rotationRate.gamma;
    }
});
```

DeviceMotionEventExample01.htm

与 `deviceorientation` 事件类似, 只有 iOS 4.2+ 中的 Safari、Chrome 和 Android 版 WebKit 实现了 `devicemotion` 事件。

13.4.9 触摸与手势事件

iOS 版 Safari 为了向开发人员传达一些特殊信息, 新增了一些专有事件。因为 iOS 设备既没有鼠标也没有键盘, 所以在为移动 Safari 开发交互性网页时, 常规的鼠标和键盘事件根本不够用。随着 Android 中的 WebKit 的加入, 很多这样的专有事件变成了事实标准, 导致 W3C 开始制定 Touch Events 规范 (参见 <https://dvcs.w3.org/hg/webevents/raw-file/tip/touchevents.html>)。以下介绍的事件只针对触摸设备。

1. 触摸事件

包含 iOS 2.0 软件的 iPhone 3G 发布时, 也包含了一个新版本的 Safari 浏览器。这款新的移动 Safari 提供了一些与触摸 (touch) 操作相关的新事件。后来, Android 上的浏览器也实现了相同的事件。触摸事件会在用户手指放在屏幕上面时、在屏幕上滑动时或从屏幕上移开时触发。具体来说, 有以下几个触摸事件。

- `touchstart`: 当手指触摸屏幕时触发; 即使已经有一个手指放在了屏幕上也会触发。
- `touchmove`: 当手指在屏幕上滑动时连续地触发。在这个事件发生期间, 调用 `preventDefault()` 可以阻止滚动。
- `touchend`: 当手指从屏幕上移开时触发。
- `touchcancel`: 当系统停止跟踪触摸时触发。关于此事件的确切触发时间, 文档中没有明确说明。

上面这几个事件都会冒泡, 也都可以取消。虽然这些触摸事件没有在 DOM 规范中定义, 但它们却是以兼容 DOM 的方式实现的。因此, 每个触摸事件的 `event` 对象都提供了在鼠标事件中常见的属性: `bubbles`、`cancelable`、`view`、`clientX`、`clientY`、`screenX`、`screenY`、`detail`、`altKey`、`shiftKey`、`ctrlKey` 和 `metaKey`。

除了常见的 DOM 属性外, 触摸事件还包含下列三个用于跟踪触摸的属性。

- `touches`: 表示当前跟踪的触摸操作的 Touch 对象的数组。
 - `targetTouches`: 特定于事件目标的 Touch 对象的数组。
 - `changeTouches`: 表示自上次触摸以来发生了什么改变的 Touch 对象的数组。
- 每个 Touch 对象包含下列属性。

- clientX: 触摸目标在视口中的 x 坐标。
- clientY: 触摸目标在视口中的 y 坐标。
- identifier: 标识触摸的唯一 ID。
- pageX: 触摸目标在页面中的 x 坐标。
- pageY: 触摸目标在页面中的 y 坐标。
- screenX: 触摸目标在屏幕中的 x 坐标。
- screenY: 触摸目标在屏幕中的 y 坐标。
- target: 触摸的 DOM 节点目标。

使用这些属性可以跟踪用户对屏幕的触摸操作。来看下面的例子。

```
function handleTouchEvent(event){
    //只跟踪一次触摸
    if (event.touches.length == 1){

        var output = document.getElementById("output");
        switch(event.type){
            case "touchstart":
                output.innerHTML = "Touch started (" + event.touches[0].clientX +
                    "," + event.touches[0].clientY + ")";

                break;
            case "touchend":
                output.innerHTML += "<br>Touch ended (" +
                    event.changedTouches[0].clientX + "," +
                    event.changedTouches[0].clientY + ")";

                break;
            case "touchmove":
                event.preventDefault(); //阻止滚动
                output.innerHTML += "<br>Touch moved (" +
                    event.changedTouches[0].clientX + "," +
                    event.changedTouches[0].clientY + ")";

                break;
        }
    }
}

EventUtil.addHandler(document, "touchstart", handleTouchEvent);
EventUtil.addHandler(document, "touchend", handleTouchEvent);
EventUtil.addHandler(document, "touchmove", handleTouchEvent);
```

TouchEventExample01.htm

以上代码会跟踪屏幕上发生的一次触摸操作。为简单起见，只会在有一次活动触摸操作的情况下输出信息。当 touchstart 事件发生时，会将触摸的位置信息输出到<div>元素中。当 touchmove 事件发生时，会取消其默认行为，阻止滚动（触摸移动的默认行为是滚动页面），然后输出触摸操作的变化信息。而 touchend 事件则会输出有关触摸操作的最终信息。注意，在 touchend 事件发生时，touches 集合中就没有任何 Touch 对象了，因为不存在活动的触摸操作；此时，就必须转而使用 changeTouches 集合。

这些事件会在文档的所有元素上面触发，因而可以分别操作页面的不同部分。在触摸屏幕上的元素时，这些事件（包括鼠标事件）发生的顺序如下：

- (1) touchstart
- (2) mouseover
- (3) mousemove (一次)
- (4) mousedown
- (5) mouseup
- (6) click
- (7) touchend

支持触摸事件的浏览器包括 iOS 版 Safari、Android 版 WebKit、bada 版 Dolfin、OS6+ 中的 BlackBerry WebKit、Opera Mobile 10.1+ 和 LG 专有 OS 中的 Phantom 浏览器。目前只有 iOS 版 Safari 支持多点触摸。桌面版 Firefox 6+ 和 Chrome 也支持触摸事件。

2. 手势事件

iOS 2.0 中的 Safari 还引入了一组手势事件。当两个手指触摸屏幕时就会产生手势，手势通常会改变显示项的大小，或者旋转显示项。有三个手势事件，分别介绍如下。


- gesturestart: 当一个手指已经按在屏幕上而另一个手指又触摸屏幕时触发。
- gesturechange: 当触摸屏幕的任何一个手指的位置发生变化时触发。
- gestureend: 当任何一个手指从屏幕上面移开时触发。

只有两个手指都触摸到事件的接收容器时才会触发这些事件。在一个元素上设置事件处理程序，意味着两个手指必须同时位于该元素的范围之内，才能触发手势事件（这个元素就是目标）。由于这些事件冒泡，所以将事件处理程序放在文档上也可以处理所有手势事件。此时，事件的目标就是两个手指都位于其范围内的那个元素。

触摸事件和手势事件之间存在某种关系。当一个手指放在屏幕上时，会触发 touchstart 事件。如果另一个手指又放在了屏幕上，则会先触发 gesturestart 事件，随后触发基于该手指的 touchstart 事件。如果一个或两个手指在屏幕上滑动，将会触发 gesturechange 事件。但只要有一个手指移开，就会触发 gestureend 事件，紧接着又会触发基于该手指的 touchend 事件。

与触摸事件一样，每个手势事件的 event 对象都包含着标准的鼠标事件属性：bubbles、cancelable、view、clientX、clientY、screenX、screenY、detail、altKey、shiftKey、ctrlKey 和 metaKey。此外，还包含两个额外的属性：rotation 和 scale。其中，rotation 属性表示手指变化引起的旋转角度，负值表示逆时针旋转，正值表示顺时针旋转（该值从 0 开始）。而 scale 属性表示两个手指间距离的变化情况（例如向内收缩会缩短距离）；这个值从 1 开始，并随距离拉大而增长，随距离缩短而减小。

下面是使用手势事件的一个示例。



```
function handleGestureEvent(event){
    var output = document.getElementById("output");
    switch(event.type){
        case "gesturestart":
            output.innerHTML = "Gesture started (rotation=" + event.rotation +
                                ",scale=" + event.scale + ")";
            break;
        case "gestureend":
            output.innerHTML += "<br>Gesture ended (rotation=" + event.rotation +
                                ",scale=" + event.scale + ")";
            break;
        case "gesturechange":
```

```

        output.innerHTML += "<br>Gesture changed (rotation=" + event.rotation +
            ",scale=" + event.scale + ")";
    }
    break;
}

document.addEventListener("gesturestart", handleGestureEvent, false);
document.addEventListener("gestureend", handleGestureEvent, false);
document.addEventListener("gesturechange", handleGestureEvent, false);

```

GestureEventsExample01.htm

与前面演示触摸事件的例子一样，这里的代码只是将每个事件都关联到同一个函数中，然后通过该函数输出每个事件的相关信息。



触摸事件也会返回 `rotation` 和 `scale` 属性，但这两个属性只会在两个手指与屏幕保持接触时才会发生变化。一般来说，使用基于两个手指的手势事件，要比管理触摸事件中的所有交互要容易得多。

13.5 内存和性能

由于事件处理程序可以为现代 Web 应用程序提供交互能力，因此许多开发人员会不分青红皂白地向页面中添加大量的处理程序。在创建 GUI 的语言（如 C#）中，为 GUI 中的每个按钮添加一个 `onclick` 事件处理程序是司空见惯的事，而且这样做也不会导致什么问题。可是在 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的整体运行性能。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。事实上，从如何利用好事件处理程序的角度出发，还是有一些方法能够提升性能的。

13.5.1 事件委托

对“事件处理程序过多”问题的解决方案就是事件委托。事件委托利用了事件冒泡，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。以下面的 HTML 代码为例。

```

<ul id="myLinks">
  <li id="goSomewhere">Go somewhere</li>
  <li id="doSomething">Do something</li>
  <li id="sayHi">Say hi</li>
</ul>

```

EventDelegationExample01.htm

其中包含 3 个被单击后会执行操作的列表项。按照传统的做法，需要像下面这样为它们添加 3 个事件处理程序。

```

var item1 = document.getElementById("goSomewhere");
var item2 = document.getElementById("doSomething");
var item3 = document.getElementById("sayHi");


EventUtil.addHandler(item1, "click", function(event){
    location.href = "http://www.wrox.com";
});

EventUtil.addHandler(item2, "click", function(event){
    document.title = "I changed the document's title";
});

EventUtil.addHandler(item3, "click", function(event){
    alert("hi");
});

```

如果在一个复杂的 Web 应用程序中,对所有可单击的元素都采用这种方式,那么结果就会有数不清的代码用于添加事件处理程序。此时,可以利用事件委托技术解决这个问题。使用事件委托,只需在 DOM 树中尽量最高的层次上添加一个事件处理程序,如下面的例子所示。



```

var list = document.getElementById("myLinks");

EventUtil.addHandler(list, "click", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    switch(target.id){
        case "doSomething":
            document.title = "I changed the document's title";
            break;

        case "goSomewhere":
            location.href = "http://www.wrox.com";
            break;

        case "sayHi":
            alert("hi");
            break;
    }
});

```

EventDelegationExample01.htm

在这段代码里,我们使用事件委托只为元素添加了一个 onclick 事件处理程序。由于所有列表项都是这个元素的子节点,而且它们的事件会冒泡,所以单击事件最终会被这个函数处理。我们知道,事件目标是被单击的列表项,故而可以通过检测 id 属性来决定采取适当的操作。与前面未使用事件委托的代码比一比,会发现这段代码的事前消耗更低,因为只取得了一个 DOM 元素,只添加了一个事件处理程序。虽然对用户来说最终的结果相同,但这种技术需要占用的内存更少。所有用到按钮的事件(多数鼠标事件和键盘事件)都适合采用事件委托技术。

如果可行的话,也可以考虑为 document 对象添加一个事件处理程序,用以处理页面上发生的某种特定类型的事件。这样做与采取传统的做法相比具有如下优点。

- document 对象很快就可以访问,而且可以在页面生命周期的任何时点上为它添加事件处理程序(无需等待 DOMContentLoaded 或 load 事件)。换句话说,只要可单击的元素呈现在页面上,就可以立即具备适当的功能。
- 在页面中设置事件处理程序所需的时间更少。只添加一个事件处理程序所需的 DOM 引用更少,所花的时间也更少。
- 整个页面占用的内存空间更少,能够提升整体性能。

最适合采用事件委托技术的事件包括 click、mousedown、mouseup、keydown、keyup 和 keypress。虽然 mouseover 和 mouseout 事件也冒泡,但要适当处理它们并不容易,而且经常需要计算元素的位置。(因为当鼠标从一个元素移到其子节点时,或者当鼠标移出该元素时,都会触发 mouseout 事件。)

13.5.2 移除事件处理程序

每当将事件处理程序指定给元素时,运行中的浏览器代码与支持页面交互的 JavaScript 代码之间就会建立一个连接。这种连接越多,页面执行起来就越慢。如前所述,可以采用事件委托技术,限制建立的连接数量。另外,在不需要的时候移除事件处理程序,也是解决这个问题的一种方案。内存中留有那些过时不用的“空事件处理程序”(dangling event handler),也是造成 Web 应用程序内存与性能问题的主要原因。

在两种情况下,可能会造成上述问题。第一种情况就是从文档中移除带有事件处理程序的元素时。这可能是通过纯粹的 DOM 操作,例如使用 removeChild() 和 replaceChild() 方法,但更多地是发生在使用 innerHTML 替换页面中某一部分的时候。如果带有事件处理程序的元素被 innerHTML 删除了,那么原来添加到元素中的事件处理程序极有可能无法被当作垃圾回收。来看下面的例子。

```
<div id="myDiv">
  <input type="button" value="Click Me" id="myBtn">
</div>
<script type="text/javascript">
  var btn = document.getElementById("myBtn");
  btn.onclick = function(){

    //先执行某些操作

    document.getElementById("myDiv").innerHTML = "Processing..."; //麻烦了!
  };
</script>
```

这里,有一个按钮被包含在<div>元素中。为避免双击,单击这个按钮时就将按钮移除并替换成一条消息;这是网站设计中非常流行的一种做法。但问题在于,当按钮被从页面中移除时,它还带着一个事件处理程序呢。在<div>元素上设置 innerHTML 可以把按钮移走,但事件处理程序仍然与按钮保持着引用关系。有的浏览器(尤其是 IE)在这种情况下不会作出恰当地处理,它们很有可能会将对元素和对事件处理程序的引用都保存在内存中。如果你知道某个元素即将被移除,那么最好手工移除事件处理程序,如下面的例子所示。

```
<div id="myDiv">
  <input type="button" value="Click Me" id="myBtn">
</div>
<script type="text/javascript">
  var btn = document.getElementById("myBtn");
```

```
btn.onclick = function(){  
    //先执行某些操作  
  
    btn.onclick = null;    //移除事件处理程序  
  
    document.getElementById("myDiv").innerHTML = "Processing...";  
};  
</script>
```

在此，我们在设置<div>的 innerHTML 属性之前，先移除了按钮的事件处理程序。这样就确保了内存可以被再次利用，而从 DOM 中移除按钮也做到了干净利索。

注意，在事件处理程序中删除按钮也能阻止事件冒泡。目标元素在文档中是事件冒泡的前提。



采用事件委托也有助于解决这个问题。如果事先知道将来有可能使用 innerHTML 替换掉页面中的某一部分，那么就可以不直接把事件处理程序添加到该部分的元素中。而通过把事件处理程序指定给较高层次的元素，同样能够处理该区域中的事件。

导致“空事件处理程序”的另一种情况，就是卸载页面的时候。毫不奇怪，IE8 及更早版本在这种情况下依然是问题最多的浏览器，尽管其他浏览器或多或少也有类似的问题。如果在页面被卸载之前没有清理干净事件处理程序，那它们就会滞留在内存中。每次加载完页面再卸载页面时（可能是在两个页面间来回切换，也可以是单击了“刷新”按钮），内存中滞留的对象数目就会增加，因为事件处理程序占用的内存并没有被释放。

一般来说，最好的做法是在页面卸载之前，先通过 onunload 事件处理程序移除所有事件处理程序。在此，事件委托技术再次表现出它的优势——需要跟踪的事件处理程序越少，移除它们就越容易。对这种类似撤销的操作，我们可以把它想象成：只要是通过 onload 事件处理程序添加的东西，最后都要通过 onunload 事件处理程序将它们移除。



不要忘了，使用 onunload 事件处理程序意味着页面不会被缓存在 bfcache 中。如果你在意这个问题，那么就只能在 IE 中通过 onunload 来移除事件处理程序了。

13.6 模拟事件

事件，就是网页中某个特别值得关注的瞬间。事件经常由用户操作或通过其他浏览器功能来触发。但很少有人知道，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浏览器创建的事件一样。也就是说，这些事件该冒泡还会冒泡，而且照样能够导致浏览器执行已经指定的处理它们的事件处理程序。在测试 Web 应用程序，模拟触发事件是一种极其有用的技术。DOM2 级规范为此规定了模拟特定事件的方式，IE9、Opera、Firefox、Chrome 和 Safari 都支持这种方式。IE 有它自己模拟事件的方式。

13.6.1 DOM 中的事件模拟

可以在 document 对象上使用 createEvent() 方法创建 event 对象。这个方法接收一个参数，即表示要创建的事件类型的字符串。在 DOM2 级中，所有这些字符串都使用英文复数形式，而在 DOM3

级中都变成了单数。这个字符串可以是下列几个字符串之一。

- **UIEvents**: 一般化的 UI 事件。鼠标事件和键盘事件都继承自 UI 事件。DOM3 级中是 **UIEvent**。
- **MouseEvents**: 一般化的鼠标事件。DOM3 级中是 **MouseEvent**。
- **MutationEvents**: 一般化的 DOM 变动事件。DOM3 级中是 **MutationEvent**。
- **HTMLEvents**: 一般化的 HTML 事件。没有对应的 DOM3 级事件 (HTML 事件被分散到其他类别中)。

要注意的是,“DOM2 级事件”并没有专门规定键盘事件,后来的“DOM3 级事件”中才正式将其作为一种事件给出规定。IE9 是目前唯一支持 DOM3 级键盘事件的浏览器。不过,在其他浏览器中,在现有方法的基础上,可以通过几种方式来模拟键盘事件。

在创建了 event 对象之后,还需要使用与事件有关的信息对其进行初始化。每种类型的 event 对象都有一个特殊的方法,为它传入适当的数据就可以初始化该 event 对象。不同类型的这个方法的名字也不相同,具体要取决于 `createEvent()` 中使用的参数。


模拟事件的最后一步就是触发事件。这一步需要使用 `dispatchEvent()` 方法,所有支持事件的 DOM 节点都支持这个方法。调用 `dispatchEvent()` 方法时,需要传入一个参数,即表示要触发事件的 event 对象。触发事件之后,该事件就跻身“官方事件”之列了,因而能够照样冒泡并引发相应事件处理程序的执行。

1. 模拟鼠标事件

创建新的鼠标事件对象并为其指定必要的信息,就可以模拟鼠标事件。创建鼠标事件对象的方法是 `createEvent()` 传入字符串“**MouseEvents**”。返回的对象有一个名为 `initMouseEvent()` 方法,用于指定与该鼠标事件有关的信息。这个方法接收 15 个参数,分别与鼠标事件中每个典型的属性——对应;这些参数的含义如下。

- **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 **true**。
- **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): 事件想对于视口的 Y 坐标。
- **ctrlKey** (布尔值): 表示是否按下了 **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。

显而易见, `initMouseEvent()` 方法的这些参数是与鼠标事件的 `event` 对象所包含的属性一一对应的。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。当把 `event` 对象传给 `dispatchEvent()` 方法时, 这个对象的 `target` 属性会自动设置。下面, 我们就通过一个例子来了解如何模拟对按钮的单击事件。



```
var btn = document.getElementById("myBtn");

// 创建事件对象
var event = document.createEvent("MouseEvents");

// 初始化事件对象
event.initMouseEvent("click", true, true, document.defaultView, 0, 0, 0, 0, 0,
                    false, false, false, false, 0, null);

// 触发事件
btn.dispatchEvent(event);
```

SimulateDOMClickExample01.htm

在兼容 DOM 的浏览器中, 也可以通过相同的方式来模拟其他鼠标事件 (例如 `dblclick`)。


2. 模拟键盘事件

前面曾经提到过, “DOM2 级事件” 中没有就键盘事件作出规定, 因此模拟键盘事件并没有现成的思路可循。“DOM2 级事件” 的草案中本来包含了键盘事件, 但在定稿之前又被删除了; Firefox 根据其草案实现了键盘事件。需要提请大家注意的是, “DOM3 级事件” 中的键盘事件与曾包含在 “DOM2 级事件” 草案中的键盘事件有很大区别。

DOM3 级规定, 调用 `createEvent()` 并传入 “KeyboardEvent” 就可以创建一个键盘事件。返回的事件对象会包含一个 `initKeyEvent()` 方法, 这个方法接收下列参数。

- `type` (字符串): 表示要触发的事件类型, 如 “`keydown`”。
- `bubbles` (布尔值): 表示事件是否应该冒泡。为精确模拟鼠标事件, 应该设置为 `true`。
- `cancelable` (布尔值): 表示事件是否可以取消。为精确模拟鼠标事件, 应该设置为 `true`。
- `view` (`AbstractView`): 与事件关联的视图。这个参数几乎总是要设置为 `document.defaultView`。
- `key` (布尔值): 表示按下的键的键码。
- `location` (整数): 表示按下了哪里的键。0 表示默认的主键盘, 1 表示左, 2 表示右, 3 表示数字键盘, 4 表示移动设备 (即虚拟键盘), 5 表示手柄。
- `modifiers` (字符串): 空格分隔的修改键列表, 如 “`Shift`”。
- `repeat` (整数): 在一行中按了这个键多少次。

由于 DOM3 级不提倡使用 `keypress` 事件, 因此只能利用这种技术来模拟 `keydown` 和 `keyup` 事件。



```
var textbox = document.getElementById("myTextbox"),
    event;

// 以 DOM3 级方式创建事件对象
if (document.implementation.hasFeature("KeyboardEvents", "3.0")) {
    event = document.createEvent("KeyboardEvent");

    // 初始化事件对象
    event.initKeyboardEvent("keydown", true, true, document.defaultView, "a",
```

```
0, "Shift", 0);  
}  
  
//触发事件  
textbox.dispatchEvent(event);
```


SimulateDOMKeyEventExample01.htm

这个例子模拟的是按住 Shift 的同时又按下 A 键。在使用 document.createEvent ("KeyboardEvent") 之前, 应该先检测浏览器是否支持 DOM3 级事件; 其他浏览器返回一个非标准的 KeyboardEvent 对象。

在 Firefox 中, 调用 createEvent() 并传入 "KeyEvents" 就可以创建一个键盘事件。返回的事件对象会包含一个 initKeyEvent() 方法, 这个方法接受下列 10 个参数。

- ☐ type (字符串): 表示要触发的事件类型, 如 "keydown"。
- ☐ bubbles (布尔值): 表示事件是否应该冒泡。为精确模拟鼠标事件, 应该设置为 true。
- ☐ cancelable (布尔值): 表示事件是否可以取消。为精确模拟鼠标事件, 应该设置为 true。
- ☐ view (AbstractView): 与事件关联的视图。这个参数几乎总是要设置为 document.defaultView。
- ☐ ctrlKey (布尔值): 表示是否按下了 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。
- ☐ keyCode (整数): 被按下或释放的键的键码。这个参数对 keydown 和 keyup 事件有用, 默认值为 0。
- ☐ charCode (整数): 通过按键生成的字符的 ASCII 编码。这个参数对 keypress 事件有用, 默认值为 0。

将创建的 event 对象传入到 dispatchEvent() 方法就可以触发键盘事件, 如下面的例子所示。



```
//只适用于 Firefox  
var textbox = document.getElementById("myTextbox")  
  
//创建事件对象  
var event = document.createEvent("KeyEvents");  
  
//初始化事件对象  
event.initKeyEvent("keypress", true, true, document.defaultView, false, false,  
                  false, false, 65, 65);  
  
//触发事件  
textbox.dispatchEvent(event);
```

SimulateFFKeyEventExample01.htm

在 Firefox 中运行上面的代码, 会在指定的文本框中输入字母 A。同样, 也可以依此模拟 keyup 和 keydown 事件。

在其他浏览器中, 则需要创建一个通用的事件, 然后再向事件对象中添加键盘事件特有的信息。例如:


```

var textbox = document.getElementById("myTextbox");

//创建事件对象
var event = document.createEvent("Events");

//初始化事件对象
event.initEvent(type, bubbles, cancelable);
event.view = document.defaultView;
event.altKey = false;
event.ctrlKey = false;
event.shiftKey = false;
event.metaKey = false;
event.keyCode = 65;
event.charCode = 65;

//触发事件
textbox.dispatchEvent(event);

```

以上代码首先创建了一个通用事件，然后调用 `initEvent()` 对其进行初始化，最后又为其添加了键盘事件的具体信息。在此必须要使用通用事件，而不能使用 UI 事件，因为 UI 事件不允许向 event 对象中再添加新属性（Safari 除外）。像这样模拟事件虽然会触发键盘事件，但却不会向文本框中写入文本，这是由于无法精确模拟键盘事件所造成的。

3. 模拟其他事件

虽然鼠标事件和键盘事件是在浏览器中最经常模拟的事件，但有时候同样需要模拟变动事件和 HTML 事件。要模拟变动事件，可以使用 `createEvent("MutationEvents")` 创建一个包含 `initMutationEvent()` 方法的变动事件对象。这个方法接受的参数包括：`type`、`bubbles`、`cancelable`、`relatedNode`、`preValue`、`newValue`、`attrName` 和 `attrChange`。下面来看一个模拟变动事件的例子。

```

var event = document.createEvent("MutationEvents");
event.initMutationEvent("DOMNodeInserted", true, false, someNode, "", "", "", 0);
target.dispatchEvent(event);

```

以上代码模拟了 `DOMNodeInserted` 事件。其他变动事件也都可以照这个样子来模拟，只要改一改参数就可以了。

要模拟 HTML 事件，同样需要先创建一个 event 对象——通过 `createEvent("HTMLEvents")`，然后再使用这个对象的 `initEvent()` 方法来初始化它即可，如下面的例子所示。

```

var event = document.createEvent("HTMLEvents");
event.initEvent("focus", true, false);
target.dispatchEvent(event);

```

这个例子展示了如何在给定目标上模拟 `focus` 事件。模拟其他 HTML 事件的方法也是这样。



浏览器中很少使用变动事件和 HTML 事件，因为使用它们会受到一些限制。

4. 自定义 DOM 事件

DOM3 级还定义了“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。要创建新的自定义事件，可以调用 `createEvent("CustomEvent")`。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。
- 可以像分派其他事件一样在 DOM 中分派创建的自定义事件对象。例如:

```
var div = document.getElementById("myDiv"),
    event;

EventUtil.addHandler(div, "myevent", function(event){
    alert("DIV: " + event.detail);
});
EventUtil.addHandler(document, "myevent", function(event){
    alert("DOCUMENT: " + event.detail);
});

if (document.implementation.hasFeature("CustomEvents", "3.0")){
    event = document.createEvent("CustomEvent");
    event.initCustomEvent("myevent", true, false, "Hello world!");
    div.dispatchEvent(event);
}
```

SimulateDOMCustomEventExample01.htm

这个例子创建了一个冒泡事件"myevent"。而 event.detail 的值被设置成了一个简单的字符串, 然后在<div>元素和 document 上侦听这个事件。因为 initCustomEvent() 方法已经指定这个事件应该冒泡, 所以浏览器会负责将事件向上冒泡到 document。

支持自定义 DOM 事件的浏览器有 IE9+ 和 Firefox 6+。

13.6.2 IE 中的事件模拟

在 IE8 及之前版本中模拟事件与在 DOM 中模拟事件的思路相似: 先创建 event 对象, 然后为其指定相应的信息, 然后再使用该对象来触发事件。当然, IE 在实现每个步骤时都采用了不一样的方式。

调用 document.createEventObject() 方法可以在 IE 中创建 event 对象。但与 DOM 方式不同的是, 这个方法不接受参数, 结果会返回一个通用的 event 对象。然后, 你必须手工为这个对象添加所有必要的信息(没有方法来辅助完成这一步骤)。最后一步就是在目标上调用 fireEvent() 方法, 这个方法接受两个参数: 事件处理程序的名称和 event 对象。在调用 fireEvent() 方法时, 会自动为 event 对象添加 srcElement 和 type 属性; 其他属性则都是必须通过手工添加的。换句话说, 模拟任何 IE 支持的事件都采用相同的模式。例如, 下面的代码模拟了在一个按钮上触发 click 事件过程。

```
var btn = document.getElementById("myBtn");

//创建事件对象
var event = document.createEventObject();

//初始化事件对象
event.screenX = 100;
event.screenY = 0;
event.clientX = 0;
event.clientY = 0;
event.ctrlKey = false;
event.altKey = false;
```


```
event.shiftKey = false;
event.button = 0;

//触发事件
btn.fireEvent("onclick", event);
```

SimulateIEClickExample01.htm

这个例子先创建了一个 event 对象，然后又用一些信息对其进行了初始化。注意，这里可以为对象随意添加属性，不会有任何限制——即使添加的属性 IE8 及更早版本并不支持也无所谓。在此添加的属性对事件没有什么影响，因为只有事件处理程序才会用到它们。

采用相同的模式也可以模拟触发 keypress 事件，如下面的例子所示。



```
var textbox = document.getElementById("myTextbox");

//创建事件对象
var event = document.createEventObject();

//初始化事件对象
event.altKey = false;
event.ctrlKey = false;
event.shiftKey = false;
event.keyCode = 65;

//触发事件
textbox.fireEvent("onkeypress", event);
```

SimulateIEKeyEventExample01.htm

由于鼠标事件、键盘事件以及其他事件的 event 对象并没有什么不同，所以可以使用通用对象来触发任何类型的事件。不过，正如在 DOM 中模拟键盘事件一样，运行这个例子也不会因模拟了 keypress 而在文本框中看到任何字符，即使触发了事件处理程序也没有用。

13.7 小结

事件是将 JavaScript 与网页联系在一起的主要方式。“DOM3 级事件”规范和 HTML5 定义了常见的大多数事件。即使有规范定义了基本事件，但很多浏览器仍然在规范之外实现了自己的专有事件，从而为开发人员提供更多掌握用户交互的手段。有些专有事件与特定设备关联，例如移动 Safari 中的 orientationchange 事件就是特定关联 iOS 设备的。

在使用事件时，需要考虑如下一些内存与性能方面的问题。

- 有必要限制一个页面中事件处理程序的数量，数量太多会导致占用大量内存，而且也会让用户感觉页面反应不够灵敏。
- 建立在事件冒泡机制之上的事件委托技术，可以有效地减少事件处理程序的数量。
- 建议在浏览器卸载页面之前移除页面中的所有事件处理程序。

可以使用 JavaScript 在浏览器中模拟事件。“DOM2 级事件”和“DOM3 级事件”规范规定了模拟事件的方法，为模拟各种有定义的事件提供了方便。此外，通过组合使用一些技术，还可以在某种程度上模拟键盘事件。IE8 及之前版本同样支持事件模拟，只不过模拟的过程有些差异。

事件是 JavaScript 中最重要的主题之一，深入理解事件的工作机制以及它们对性能的影响至关重要。