```
let btn = document.getElementById("myBtn");
btn.addEventListener("click", () => {
  console.log(this.id);
}, false);
```

以上代码为按钮添加了会在事件冒泡阶段触发的 onclick 事件处理程序（因为最后一个参数值为 false）。与 DOM0 方式类似，这个事件处理程序同样在被附加到的元素的作用域中运行。使用 DOM2 方式的主要优势是可以为同一个事件添加多个事件处理程序。来看下面的例子：

```
let btn = document.getElementById("myBtn");
btn.addEventListener("click", () => {
  console.log(this.id);
}, false);
btn.addEventListener("click", () => {
  console.log("Hello world!");
}, false);
```

这里给按钮添加了两个事件处理程序。多个事件处理程序以添加顺序来触发，因此前面的代码会先打印元素 ID，然后显示消息“Hello world!”。

通过 addEventListener()添加的事件处理程序只能使用 removeEventListener()并传入与添加时同样的参数来移除。这意味着使用 addEventListener()添加的匿名函数无法移除，如下面的例子所示：

```
let btn = document.getElementById("myBtn");
btn.addEventListener("click", () => {
  console.log(this.id);
 }, false);

// 其他代码

btn.removeEventListener("click", function() {    // 没有效果!
  console.log(this.id);
}, false);
```

这个例子通过 addEventListener()添加了一个匿名函数作为事件处理程序。然后，又以看起来相同的参数调用了 removeEventListener()。但实际上，第二个参数与传给 addEventListener()的完全不是一回事。传给 removeEventListener()的事件处理函数必须与传给 addEventListener()的是同一个，如下面的例子所示：

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

// 其他代码

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

这个例子有效，因为调用 addEventListener()和 removeEventListener()时传入的是同一个函数。

大多数情况下，事件处理程序会被添加到事件流的冒泡阶段，主要原因是跨浏览器兼容性好。把事件处理程序注册到捕获阶段通常用于在事件到达其指定目标之前拦截事件。如果不需要拦截，则不要使用事件捕获。

17.2.4 IE 事件处理程序

IE 实现了与 DOM 类似的方法，即 attachEvent()和 detachEvent()。这两个方法接收两个同样的参数：事件处理程序的名字和事件处理函数。因为 IE8 及更早版本只支持事件冒泡，所以使用 attachEvent()添加的事件处理程序会添加到冒泡阶段。

要使用 attachEvent()给按钮添加 click 事件处理程序，可以使用以下代码：

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

注意，attachEvent()的第一个参数是"onclick"，而不是 DOM 的 addEventListener()方法的"click"。

在 IE 中使用 attachEvent()与使用 DOM0 方式的主要区别是事件处理程序的作用域。使用 DOM0 方式时，事件处理程序中的 this 值等于目标元素。而使用 attachEvent()时，事件处理程序是在全局作用域中运行的，因此 this 等于 window。来看下面使用 attachEvent()的例子：

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

理解这些差异对编写跨浏览器代码是非常重要的。

与使用 addEventListener()一样，使用 attachEvent()方法也可以给一个元素添加多个事件处理程序。比如下面的例子：

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

这里调用了两次 attachEvent()，分别给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，这里的事件处理程序会以添加它们的顺序反向触发。换句话说，在点击例子中的按钮后，控制台中会先打印出"Hello world!"，然后再打印出"Clicked"。

使用 attachEvent()添加的事件处理程序将使用 detachEvent()来移除，只要提供相同的参数。与使用 DOM 方法类似，作为事件处理程序添加的匿名函数也无法移除。但只要传给 detachEvent()方法相同的函数引用，就可以移除。下面的例子演示了附加和剥离事件：

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

// 其他代码

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

这里先把事件处理程序保存到变量 handler，之后又将其传给 detachEvent()来移除事件处理程序。

17.2.5　跨浏览器事件处理程序

为了以跨浏览器兼容的方式处理事件，很多开发者会选择使用一个 JavaScript 库，其中抽象了不同浏览器的差异。有些开发者也可能会自己编写代码，以便使用最合适的事件处理手段。自己编写跨浏览器事件处理代码也很简单，主要依赖能力检测。要确保事件处理代码具有最大兼容性，只需要让代码在冒泡阶段运行即可。

为此，需要先创建一个 `addHandler()`方法。这个方法的任务是根据需要分别使用 DOM0 方式、DOM2 方式或 IE 方式来添加事件处理程序。这个方法会在 `EventUtil` 对象（本章示例使用的对象）上添加一个方法，以实现跨浏览器事件处理。添加的这个 `addHandler()`方法接收 3 个参数：目标元素、事件名和事件处理函数。

有了 `addHandler()`，还要写一个也接收同样的 3 个参数的 `removeHandler()`。这个方法的任务是移除之前添加的事件处理程序，不管是通过何种方式添加的，默认为 DOM0 方式。

以下就是包含这两个方法的 `EventUtil` 对象：

```
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;
    }
  }
};
```

两个方法都是首先检测传入元素上是否存在 DOM2 方式。如果有 DOM2 方式，就使用该方式，传入事件类型和事件处理函数，以及表示冒泡阶段的第三个参数 `false`。否则，如果存在 IE 方式，则使用该方式。注意这时候必须在事件类型前加上`"on"`，才能保证在 IE8 及更早版本中有效。最后是使用 DOM0 方式（在现代浏览器中不会到这一步）。注意使用 DOM0 方式时使用了中括号计算属性名，并将事件处理程序或 `null` 赋给了这个属性。

可以像下面这样使用 `EventUtil` 对象：

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

// 其他代码

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

这里的 addHandler()和 removeHandler()方法并没有解决所有跨浏览器一致性问题，比如 IE 的作用域问题、多个事件处理程序执行顺序问题等。不过，这两个方法已经实现了跨浏览器添加和移除事件处理程序。另外也要注意，DOM0 只支持给一个事件添加一个处理程序。好在 DOM0 浏览器已经很少有人使用了，所以影响应该不大。

17.3 事件对象

在 DOM 中发生事件时，所有相关信息都会被收集并存储在一个名为 event 的对象中。这个对象包含了一些基本信息，比如导致事件的元素、发生的事件类型，以及可能与特定事件相关的任何其他数据。例如，鼠标操作导致的事件会生成鼠标位置信息，而键盘操作导致的事件会生成与被按下的键有关的信息。所有浏览器都支持这个 event 对象，尽管支持方式不同。

17.3.1 DOM 事件对象

在 DOM 合规的浏览器中，event 对象是传给事件处理程序的唯一参数。不管以哪种方式（DOM0 或 DOM2）指定事件处理程序，都会传入这个 event 对象。下面的例子展示了在两种方式下都可以使用事件对象：

```
let btn = document.getElementById("myBtn");
btn.onclick = function(event) {
  console.log(event.type);  // "click"
};

btn.addEventListener("click", (event) => {
  console.log(event.type);  // "click"
}, false);
```

这个例子中的两个事件处理程序都会在控制台打出 event.type 属性包含的事件类型。这个属性中始终包含被触发事件的类型，如"click"（与传给 addEventListener()和 removeEventListener()方法的事件名一致）。

在通过 HTML 属性指定的事件处理程序中，同样可以使用变量 event 引用事件对象。下面的例子中演示了如何使用这个变量：

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

以这种方式提供 event 对象，可以让 HTML 属性中的代码实现与 JavaScript 函数同样的功能。

如前所述，事件对象包含与特定事件相关的属性和方法。不同的事件生成的事件对象也会包含不同的属性和方法。不过，所有事件对象都会包含下表列出的这些公共属性和方法。

属性/方法	类　型	读/写	说　明
bubbles	布尔值	只读	表示事件是否冒泡
cancelable	布尔值	只读	表示是否可以取消事件的默认行为
currentTarget	元素	只读	当前事件处理程序所在的元素
defaultPrevented	布尔值	只读	true 表示已经调用 preventDefault()方法（DOM3 Events 中新增）
detail	整数	只读	事件相关的其他信息

（续）

属性/方法	类　型	读/写	说　明
eventPhase	整数	只读	表示调用事件处理程序的阶段：1 代表捕获阶段，2 代表到达目标，3 代表冒泡阶段
preventDefault()	函数	只读	用于取消事件的默认行为。只有 cancelable 为 true 才可以调用这个方法
stopImmediatePropagation()	函数	只读	用于取消所有后续事件捕获或事件冒泡，并阻止调用任何后续事件处理程序（DOM3 Events 中新增）
stopPropagation()	函数	只读	用于取消所有后续事件捕获或事件冒泡。只有 bubbles 为 true 才可以调用这个方法
target	元素	只读	事件目标
trusted	布尔值	只读	true 表示事件是由浏览器生成的。false 表示事件是开发者通过 JavaScript 创建的（DOM3 Events 中新增）
type	字符串	只读	被触发的事件类型
View	AbstractView	只读	与事件相关的抽象视图。等于事件所发生的 window 对象

在事件处理程序内部，this 对象始终等于 currentTarget 的值，而 target 只包含事件的实际目标。如果事件处理程序直接添加在了意图的目标，则 this、currentTarget 和 target 的值是一样的。下面的例子展示了这两个属性都等于 this 的情形：

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

上面的代码检测了 currentTarget 和 target 的值是否等于 this。因为 click 事件的目标是按钮，所以这 3 个值是相等的。如果这个事件处理程序是添加到按钮的父节点（如 document.body）上，那么它们的值就不一样了。比如下面的例子在 document.body 上添加了单击处理程序：

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

这种情况下点击按钮，this 和 currentTarget 都等于 document.body，这是因为它是注册事件处理程序的元素。而 target 属性等于按钮本身，这是因为那才是 click 事件真正的目标。由于按钮本身并没有注册事件处理程序，因此 click 事件冒泡到 document.body，从而触发了在它上面注册的处理程序。

type 属性在一个处理程序处理多个事件时很有用。比如下面的处理程序中就使用了 event.type：

```
let btn = document.getElementById("myBtn");
let handler = function(event) {
  switch(event.type) {
    case "click":
      console.log("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;
```


在这个例子中，函数 handler 被用于处理 3 种不同的事件：click、mouseover 和 mouseout。当按钮被点击时，应该在控制台打印一条消息，如前面的例子所示。而把鼠标放到按钮上，会导致按钮背景变成红色，接着把鼠标从按钮上移开，背景颜色应该又恢复成默认值。这个函数使用 event.type 属性确定了事件类型，从而可以做出不同的响应。

preventDefault()方法用于阻止特定事件的默认动作。比如，链接的默认行为就是在被单击时导航到 href 属性指定的 URL。如果想阻止这个导航行为，可以在 onclick 事件处理程序中取消，如下面的例子所示：

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

任何可以通过 preventDefault()取消默认行为的事件，其事件对象的 cancelable 属性都会设置为 true。

stopPropagation()方法用于立即阻止事件流在 DOM 结构中传播，取消后续的事件捕获或冒泡。例如，直接添加到按钮的事件处理程序中调用 stopPropagation()，可以阻止 document.body 上注册的事件处理程序执行。比如：

```
let btn = document.getElementById("myBtn");
btn.onclick = function(event) {
  console.log("Clicked");
  event.stopPropagation();
};

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

如果这个例子中不调用stopPropagation(),那么点击按钮就会打印两条消息。但这里由于click事件不会传播到 document.body，因此 onclick 事件处理程序永远不会执行。

eventPhase 属性可用于确定事件流当前所处的阶段。如果事件处理程序在捕获阶段被调用，则 eventPhase 等于 1；如果事件处理程序在目标上被调用，则 eventPhase 等于 2；如果事件处理程序在冒泡阶段被调用，则 eventPhase 等于 3。不过要注意的是，虽然“到达目标”是在冒泡阶段发生的，但其 eventPhase 仍然等于 2。下面的例子展示了 eventPhase 在不同阶段的值：

```
let btn = document.getElementById("myBtn");
btn.onclick = function(event) {
  console.log(event.eventPhase);   // 2
};
```

```
document.body.addEventListener("click", (event) => {
  console.log(event.eventPhase);   // 1
}, true);

document.body.onclick = (event) => {
  console.log(event.eventPhase);   // 3
};
```

在这个例子中，点击按钮首先会触发注册在捕获阶段的 document.body 上的事件处理程序，显示 eventPhase 为 1。接着，会触发按钮本身的事件处理程序（尽管是注册在冒泡阶段），此时显示 eventPhase 等于 2。最后触发的是注册在冒泡阶段的 document.body 上的事件处理程序，显示 eventPhase 为 3。而当 eventPhase 等于 2 时，this、target 和 currentTarget 三者相等。

> **注意**　event 对象只在事件处理程序执行期间存在，一旦执行完毕，就会被销毁。

17.3.2　IE 事件对象

与 DOM 事件对象不同，IE 事件对象可以基于事件处理程序被指定的方式以不同方式来访问。如果事件处理程序是使用 DOM0 方式指定的，则 event 对象只是 window 对象的一个属性，如下所示：

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

这里，window.event 中保存着 event 对象，其 event.type 属性保存着事件类型（IE 的这个属性的值与 DOM 事件对象中一样）。不过，如果事件处理程序是使用 attachEvent()指定的，则 event 对象会作为唯一的参数传给处理函数，如下所示：

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

使用 attachEvent()时，event 对象仍然是 window 对象的属性（像 DOM0 方式那样），只是出于方便也将其作为参数传入。

如果是使用 HTML 属性方式指定的事件处理程序，则 event 对象同样可以通过变量 event 访问（与 DOM 模型一样）。下面是在 HTML 事件属性中使用 event.type 的例子：

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

IE 事件对象也包含与导致其创建的特定事件相关的属性和方法，其中很多都与相关的 DOM 属性和方法对应。与 DOM 事件对象一样，基于触发的事件类型不同，event 对象中包含的属性和方法也不一样。不过，所有 IE 事件对象都会包含下表所列的公共属性和方法。

属性/方法	类　型	读/写	说　明
cancelBubble	布尔值	读/写	默认为 false，设置为 true 可以取消冒泡（与 DOM 的 stopPropagation()方法相同）

（续）

属性/方法	类 型	读/写	说 明
returnValue	布尔值	读/写	默认为 true，设置为 false 可以取消事件默认行为（与 DOM 的 preventDefault()方法相同）
srcElement	元素	只读	事件目标（与 DOM 的 target 属性相同）
type	字符串	只读	触发的事件类型

由于事件处理程序的作用域取决于指定它的方式，因此 this 值并不总是等于事件目标。为此，更好的方式是使用事件对象的 srcElement 属性代替 this。下面的例子表明，不同事件对象上的 srcElement 属性中保存的都是事件目标：

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

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

在第一个以 DOM0 方式指定的事件处理程序中，srcElement 属性等于 this，而在第二个事件处理程序中（运行在全局作用域下），两个值就不相等了。

returnValue 属性等价于 DOM 的 preventDefault()方法，都是用于取消给定事件默认的行为。只不过在这里要把 returnValue 设置为 false 才是阻止默认动作。下面是一个设置该属性的例子：

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

在这个例子中，returnValue 在 onclick 事件处理程序中被设置为 false，阻止了链接的默认行为。与 DOM 不同，没有办法通过 JavaScript 确定事件是否可以被取消。

cancelBubble 属性与 DOMstopPropagation()方法用途一样，都可以阻止事件冒泡。因为 IE8 及更早版本不支持捕获阶段，所以只会取消冒泡。stopPropagation()则既取消捕获也取消冒泡。下面是一个取消冒泡的例子：

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

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

通过在按钮的 onclick 事件处理程序中将 cancelBubble 设置为 true，可以阻止事件冒泡到 document.body，也就阻止了调用注册在它上面的事件处理程序。于是，点击按钮只会输出一条消息。

17.3.3 跨浏览器事件对象

虽然 DOM 和 IE 的事件对象并不相同，但它们有足够的相似性可以实现跨浏览器方案。DOM 事件

对象中包含 IE 事件对象的所有信息和能力，只是形式不同。这些共性可让两种事件模型之间的映射成为可能。本章前面的 EventUtil 对象可以像下面这样再添加一些方法：

```
var EventUtil = {
  addHandler: function(element, type, handler) {
     // 为节省版面，删除了之前的代码
  },

  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) {
    // 为节省版面，删除了之前的代码
  },

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

};
```

这里一共给 EventUtil 增加了 4 个新方法。首先是 getEvent()，其返回对 event 对象的引用。IE 中事件对象的位置不同，而使用这个方法可以不用管事件处理程序是如何指定的，都可以获取到 event 对象。使用这个方法的前提是，事件处理程序必须接收 event 对象，并把它传给这个方法。下面是使用 EventUtil 中这个方法统一获取 event 对象的一个例子：

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

在 DOM 合规的浏览器中，event 对象会直接传入并返回。而在 IE 中，event 对象可能并没有被定义（因为使用了 attachEvent()），因此返回 window.event。这样就可以确保无论使用什么浏览器，都可以获取到事件对象。

第二个方法是 getTarget()，其返回事件目标。在这个方法中，首先检测 event 对象是否存在 target 属性。如果存在就返回这个值；否则，就返回 event.srcElement 属性。下面是使用这个方法的示例：

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

第三个方法是 preventDefault()，其用于阻止事件的默认行为。在传入的 event 对象上，如果有 preventDefault()方法，就调用这个方法；否则，就将 event.returnValue 设置为 false。下面是使用这个方法的例子：

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

以上代码能在所有主流浏览器中阻止单击链接后跳转到其他页面。这里首先通过 EventUtil.getEvent()获取事件对象，然后又把它传给了 EventUtil.preventDefault()以阻止默认行为。

第四个方法 stopPropagation()以类似的方式运行。同样先检测用于停止事件流的 DOM 方法，如果没有再使用 cancelBubble 属性。下面是使用这个通用 stopPropagation()方法的示例：

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

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

同样，先通过 EventUtil.getEvent()获取事件对象，然后又把它传给了 EventUtil.stopPropagation()。不过，这个方法在浏览器上可能会停止事件冒泡，也可能会既停止事件冒泡也停止事件捕获。

17.4 事件类型

Web 浏览器中可以发生很多种事件。如前所述，所发生事件的类型决定了事件对象中会保存什么信息。DOM3 Events 定义了如下事件类型。

- **用户界面事件**（UIEvent）：涉及与 BOM 交互的通用浏览器事件。
- **焦点事件**（FocusEvent）：在元素获得和失去焦点时触发。
- **鼠标事件**（MouseEvent）：使用鼠标在页面上执行某些操作时触发。
- **滚轮事件**（WheelEvent）：使用鼠标滚轮（或类似设备）时触发。
- **输入事件**（InputEvent）：向文档中输入文本时触发。
- **键盘事件**（KeyboardEvent）：使用键盘在页面上执行某些操作时触发。
- **合成事件**（CompositionEvent）：在使用某种 IME（Input Method Editor，输入法编辑器）输入字符时触发。

除了这些事件类型之外，HTML5 还定义了另一组事件，而浏览器通常在 DOM 和 BOM 上实现专有事件。这些专有事件基本上都是根据开发者需求而不是按照规范增加的，因此不同浏览器的实现可能不同。