DOM3 Events 在 DOM2 Events 基础上重新定义了事件，并增加了新的事件类型。所有主流浏览器都支持 DOM2 Events 和 DOM3 Events。

17.4.1　用户界面事件

用户界面事件或 UI 事件不一定跟用户操作有关。这类事件在 DOM 规范出现之前就已经以某种形式存在了，保留它们是为了向后兼容。UI 事件主要有以下几种。

- DOMActivate：元素被用户通过鼠标或键盘操作激活时触发（比 click 或 keydown 更通用）。这个事件在 DOM3 Events 中已经废弃。因为浏览器实现之间存在差异，所以不要使用它。
- load：在 window 上当页面加载完成后触发，在窗套（<frameset>）上当所有窗格（<frame>）都加载完成后触发，在<img>元素上当图片加载完成后触发，在<object>元素上当相应对象加载完成后触发。
- unload：在 window 上当页面完全卸载后触发，在窗套上当所有窗格都卸载完成后触发，在<object>元素上当相应对象卸载完成后触发。
- abort：在<object>元素上当相应对象加载完成前被用户提前终止下载时触发。
- error：在 window 上当 JavaScript 报错时触发，在<img>元素上当无法加载指定图片时触发，在<object>元素上当无法加载相应对象时触发，在窗套上当一个或多个窗格无法完成加载时触发。
- select：在文本框（<input>或 textarea）上当用户选择了一个或多个字符时触发。
- resize：在 window 或窗格上当窗口或窗格被缩放时触发。
- scroll：当用户滚动包含滚动条的元素时在元素上触发。<body>元素包含已加载页面的滚动条。

大多数 HTML 事件与 window 对象和表单控件有关。

除了 DOMActivate，这些事件在 DOM2 Events 中都被归为 HTML Events（DOMActivate 在 DOM2 中仍旧是 UI 事件）。

1. load 事件

load 事件可能是 JavaScript 中最常用的事件。在 window 对象上，load 事件会在整个页面（包括所有外部资源如图片、JavaScript 文件和 CSS 文件）加载完成后触发。可以通过两种方式指定 load 事件处理程序。第一种是 JavaScript 方式，如下所示：

```
window.addEventListener("load", (event) => {
  console.log("Loaded!");
});
```

这是使用 addEventListener()方法来指定事件处理程序。与其他事件一样，事件处理程序会接收到一个 event 对象。这个 event 对象并没有提供关于这种类型事件的额外信息，虽然在 DOM 合规的浏览器中，event.target 会被设置为 document，但在 IE8 之前的版本中，不会设置这个对象的 srcElement 属性。

第二种指定 load 事件处理程序的方式是向<body>元素添加 onload 属性，如下所示：

```
<!DOCTYPE html>
<html>
<head>
  <title>Load Event Example</title>
</head>
```

```
<body onload="console.log('Loaded!')">

</body>
</html>
```

一般来说，任何在 window 上发生的事件，都可以通过给<body>元素上对应的属性赋值来指定，这是因为 HTML 中没有 window 元素。这实际上是为了保证向后兼容的一个策略，但在所有浏览器中都能得到很好的支持。实际开发中要尽量使用 JavaScript 方式。

> **注意** 根据 DOM2 Events，load 事件应该在 document 而非 window 上触发。可是为了向后兼容，所有浏览器都在 window 上实现了 load 事件。

图片上也会触发 load 事件，包括 DOM 中的图片和非 DOM 中的图片。可以在 HTML 中直接给<img>元素的 onload 属性指定事件处理程序，比如：

```
<img src="smile.gif" onload="console.log('Image loaded.')">
```

这个例子会在图片加载完成后输出一条消息。同样，使用 JavaScript 也可以为图片指定事件处理程序：

```
let image = document.getElementById("myImage");
image.addEventListener("load", (event) => {
  console.log(event.target.src);
});
```

这里使用 JavaScript 为图片指定了 load 事件处理程序。处理程序会接收到 event 对象，虽然这个对象上没有多少有用的信息。这个事件的目标是<img>元素，因此可以直接从 event.target.src 属性中取得图片地址并打印出来。

在通过 JavaScript 创建新<img>元素时，也可以给这个元素指定一个在加载完成后执行的事件处理程序。在这里，关键是要在赋值 src 属性前指定事件处理程序，如下所示：

```
window.addEventListener("load", () => {
  let image = document.createElement("img");
  image.addEventListener("load", (event) => {
    console.log(event.target.src);
  });
  document.body.appendChild(image);
  image.src = "smile.gif";
});
```

这个例子首先为 window 指定了一个 load 事件处理程序。因为示例涉及向 DOM 中添加新元素，所以必须确保页面已经加载完成。如果在页面加载完成之前操作 document.body，则会导致错误。然后，代码创建了一个新的<img>元素，并为这个元素设置了 load 事件处理程序。最后，才把这个元素添加到文档中并指定了其 src 属性。注意，下载图片并不一定要把<img>元素添加到文档，只要给它设置了 src 属性就会立即开始下载。

同样的技术也适用于 DOM0 的 Image 对象。在 DOM 出现之前，客户端都使用 Image 对象预先加载图片。可以像使用前面（通过 createElement()方法创建）的<img>元素一样使用 Image 对象，只是不能把后者添加到 DOM 树。下面的例子使用新 Image 对象实现了图片预加载：

```
window.addEventListener("load", () => {
  let image = new Image();
  image.addEventListener("load", (event) => {
    console.log("Image loaded!");
```

```
  });
  image.src = "smile.gif";
});
```

这里调用 Image 构造函数创建了一个新图片，并给它设置了事件处理程序。有些浏览器会把 Image 对象实现为<img>元素，但并非所有浏览器都如此。所以最好把它们看成是两个东西。

> **注意**　在 IE8 及早期版本中，如果图片没有添加到 DOM 文档中，则 load 事件发生时不会生成 event 对象。对未被添加到文档中的<img>元素以及 Image 对象来说都是这样。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。<script>元素会在 JavaScript 文件加载完成后触发 load 事件，从而可以动态检测。与图片不同，要下载 JavaScript 文件必须同时指定 src 属性并把<script>元素添加到文档中。因此指定事件处理程序和指定 src 属性的顺序在这里并不重要。下面的代码展示了如何给动态创建的<script>元素指定事件处理程序：

```
window.addEventListener("load", () => {
  let script = document.createElement("script");
  script.addEventListener("load", (event) => {
    console.log("Loaded");
  });
  script.src = "example.js";
  document.body.appendChild(script);
});
```

这里 event 对象的 target 属性在大多数浏览器中是<script>节点。IE8 及更早版本不支持<script>元素触发 load 事件。

IE 和 Opera 支持<link>元素触发 load 事件，因而支持动态检测样式表是否加载完成。下面的代码展示了如何设置这样的事件处理程序：

```
window.addEventListener("load", () => {
  let link = document.createElement("link");
  link.type = "text/css";
  link.rel= "stylesheet";
  link.addEventListener("load", (event) => {
    console.log("css loaded");
  });
  link.href = "example.css";
  document.getElementsByTagName("head")[0].appendChild(link);
});
```

与<script>节点一样，在指定 href 属性并把<link>节点添加到文档之前不会下载样式表。

2. unload 事件

与 load 事件相对的是 unload 事件，unload 事件会在文档卸载完成后触发。unload 事件一般是在从一个页面导航到另一个页面时触发，最常用于清理引用，以避免内存泄漏。与 load 事件类似，unload 事件处理程序也有两种指定方式。第一种是 JavaScript 方式，如下所示：

```
window.addEventListener("unload", (event) => {
  console.log("Unloaded!");
});
```

这个事件生成的 event 对象在 DOM 合规的浏览器中只有 target 属性（值为 document）。IE8 及

更早版本在这个事件上不提供 srcElement 属性。

第二种方式与 load 事件类似，就是给<body>元素添加 onunload 属性：

```
<!DOCTYPE html>
<html>
<head>
  <title>Unload Event Example</title>
</head>
<body onunload="console.log('Unloaded!')">

</body>
</html>
```


无论使用何种方式，都要注意事件处理程序中的代码。因为 unload 事件是在页面卸载完成后触发的，所以不能使用页面加载后才有的对象。此时要访问 DOM 或修改页面外观都会导致错误。

> **注意** 根据 DOM2 Events，unload 事件应该在<body>而非 window 上触发。可是为了向后兼容，所有浏览器都在 window 上实现了 unload 事件。

3. resize 事件

当浏览器窗口被缩放到新高度或宽度时，会触发 resize 事件。这个事件在 window 上触发，因此可以通过 JavaScript 在 window 上或者为<body>元素添加 onresize 属性来指定事件处理程序。优先使用 JavaScript 方式：

```
window.addEventListener("resize", (event) => {
  console.log("Resized");
});
```

类似于其他在 window 上发生的事件，此时会生成 event 对象，且这个对象的 target 属性在 DOM 合规的浏览器中是 document。而 IE8 及更早版本中并没有提供可用的属性。

不同浏览器在决定何时触发 resize 事件上存在重要差异。IE、Safari、Chrome 和 Opera 会在窗口缩放超过 1 像素时触发 resize 事件，然后随着用户缩放浏览器窗口不断触发。Firefox 早期版本则只在用户停止缩放浏览器窗口时触发 resize 事件。无论如何，都应该避免在这个事件处理程序中执行过多计算。否则可能由于执行过于频繁而导致浏览器响应明确变慢。

> **注意** 浏览器窗口在最大化和最小化时也会触发 resize 事件。

4. scroll 事件

虽然 scroll 事件发生在 window 上，但实际上反映的是页面中相应元素的变化。在混杂模式下，可以通过<body>元素检测 scrollLeft 和 scrollTop 属性的变化。而在标准模式下，这些变化在除早期版的 Safari 之外的所有浏览器中都发生在<html>元素上（早期版的 Safari 在<body>上跟踪滚动位置）。下面的代码演示了如何处理这些差异：

```
window.addEventListener("scroll", (event) => {
  if (document.compatMode == "CSS1Compat") {
    console.log(document.documentElement.scrollTop);
  } else {
    console.log(document.body.scrollTop);
  }
});
```

以上事件处理程序会在页面滚动时输出垂直方向上滚动的距离，而且适用于不同渲染模式。因为 Safari 3.1 之前不支持 document.compatMode，所以早期版本会走第二个分支。

类似于 resize，scroll 事件也会随着文档滚动而重复触发，因此最好保持事件处理程序的代码尽可能简单。

17.4.2　焦点事件

焦点事件在页面元素获得或失去焦点时触发。这些事件可以与 document.hasFocus() 和 document.activeElement 一起为开发者提供用户在页面中导航的信息。焦点事件有以下 6 种。

- blur：当元素失去焦点时触发。这个事件不冒泡，所有浏览器都支持。
- DOMFocusIn：当元素获得焦点时触发。这个事件是 focus 的冒泡版。Opera 是唯一支持这个事件的主流浏览器。DOM3 Events 废弃了 DOMFocusIn，推荐 focusin。
- DOMFocusOut：当元素失去焦点时触发。这个事件是 blur 的通用版。Opera 是唯一支持这个事件的主流浏览器。DOM3 Events 废弃了 DOMFocusOut，推荐 focusout。
- focus：当元素获得焦点时触发。这个事件不冒泡，所有浏览器都支持。
- focusin：当元素获得焦点时触发。这个事件是 focus 的冒泡版。
- focusout：当元素失去焦点时触发。这个事件是 blur 的通用版。

焦点事件中的两个主要事件是 focus 和 blur，这两个事件在 JavaScript 早期就得到了浏览器支持。它们最大的问题是不冒泡。这导致 IE 后来又增加了 focusin 和 focusout，Opera 又增加了 DOMFocusIn 和 DOMFocusOut。IE 新增的这两个事件已经被 DOM3 Events 标准化。

当焦点从页面中的一个元素移到另一个元素上时，会依次发生如下事件。

(1) focuscout 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。

17.4.3　鼠标和滚轮事件

鼠标事件是 Web 开发中最常用的一组事件，这是因为鼠标是用户的主要定位设备。DOM3 Events 定义了 9 种鼠标事件。

- click：在用户单击鼠标主键（通常是左键）或按键盘回车键时触发。这主要是基于无障碍的考虑，让键盘和鼠标都可以触发 onclick 事件处理程序。
- dblclick：在用户双击鼠标主键（通常是左键）时触发。这个事件不是在 DOM2 Events 中定义的，但得到了很好的支持，DOM3 Events 将其进行了标准化。
- mousedown：在用户按下任意鼠标键时触发。这个事件不能通过键盘触发。

- mouseenter：在用户把鼠标光标从元素外部移到元素内部时触发。这个事件不冒泡，也不会在光标经过后代元素时触发。mouseenter 事件不是在 DOM2 Events 中定义的，而是 DOM3 Events 中新增的事件。
- mouseleave：在用户把鼠标光标从元素内部移到元素外部时触发。这个事件不冒泡，也不会在光标经过后代元素时触发。mouseleave 事件不是在 DOM2 Events 中定义的，而是 DOM3 Events 中新增的事件。
- mousemove：在鼠标光标在元素上移动时反复触发。这个事件不能通过键盘触发。
- mouseout：在用户把鼠标光标从一个元素移到另一个元素上时触发。移到的元素可以是原始元素的外部元素，也可以是原始元素的子元素。这个事件不能通过键盘触发。
- mouseover：在用户把鼠标光标从元素外部移到元素内部时触发。这个事件不能通过键盘触发。
- mouseup：在用户释放鼠标键时触发。这个事件不能通过键盘触发。

页面中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，都可以被取消，而这会影响浏览器的默认行为。

由于事件之间存在关系，因此取消鼠标事件的默认行为也会影响其他事件。

比如，click 事件触发的前提是 mousedown 事件触发后，紧接着又在同一个元素上触发了 mouseup 事件。如果 mousedown 和 mouseup 中的任意一个事件被取消，那么 click 事件就不会触发。类似地，两次连续的 click 事件会导致 dblclick 事件触发。只要有任何逻辑阻止了这两个 click 事件发生（比如取消其中一个 click 事件或者取消 mousedown 或 mouseup 事件中的任一个），dblclick 事件就不会发生。这 4 个事件永远会按照如下顺序触发：

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

click 和 dblclick 在触发前都依赖其他事件触发，mousedown 和 mouseup 则不会受其他事件影响。

IE8 及更早版本的实现中有个问题，这会导致双击事件跳过第二次 mousedown 和 click 事件。相应的顺序变成了：

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

鼠标事件在 DOM3 Events 中对应的类型是"MouseEvent"，而不是"MouseEvents"。

鼠标事件还有一个名为**滚轮事件**的子类别。滚轮事件只有一个事件 mousewheel，反映的是鼠标滚轮或带滚轮的类似设备上滚轮的交互。

1. 客户端坐标

鼠标事件都是在浏览器视口中的某个位置上发生的。这些信息被保存在 event 对象的 clientX 和

clientY 属性中。这两个属性表示事件发生时鼠标光标在视口中的坐标，所有浏览器都支持。图 17-4 展示了视口中的**客户端坐标**。

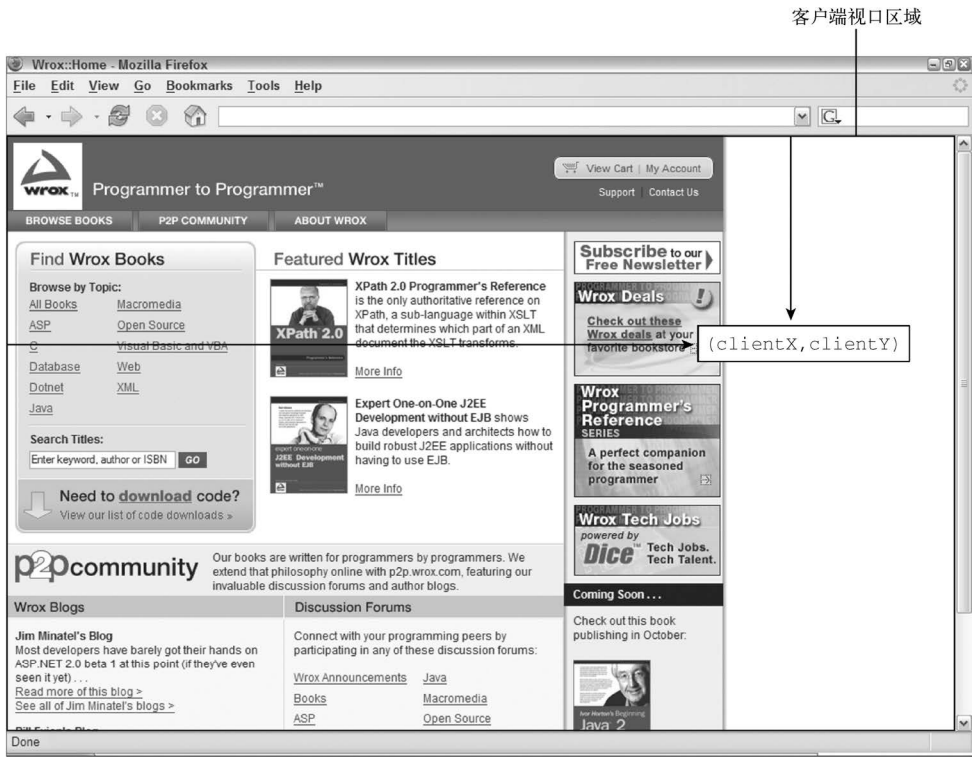

图　17-4

可以通过下面的方式获取鼠标事件的客户端坐标：

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
  console.log(`Client coordinates: ${event.clientX}, ${event.clientY}`);
});
```

这个例子为 元素指定了一个 onclick 事件处理程序。当元素被点击时，会显示事件发生时鼠标光标在客户端视口中的坐标。注意客户端坐标不考虑页面滚动，因此这两个值并不代表鼠标在页面上的位置。

2. 页面坐标

客户端坐标是事件发生时鼠标光标在客户端视口中的坐标，而**页面坐标**是事件发生时鼠标光标在页面上的坐标，通过 event 对象的 pageX 和 pageY 可以获取。这两个属性表示鼠标光标在页面上的位置，因此反映的是光标到页面而非视口左边与上边的距离。

可以像下面这样取得鼠标事件的页面坐标：

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
  console.log(`Page coordinates: ${event.pageX}, ${event.pageY}`);
});
```

在页面没有滚动时，pageX 和 pageY 与 clientX 和 clientY 的值相同。

IE8 及更早版本没有在 event 对象上暴露页面坐标。不过，可以通过客户端坐标和滚动信息计算出来。滚动信息可以从 document.body（混杂模式）或 document.documentElement（标准模式）的 scrollLeft 和 scrollTop 属性获取。计算过程如下所示：

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
  let 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);
  }
  console.log(`Page coordinates: ${pageX}, ${pageY}`);
});
```

3. 屏幕坐标

鼠标事件不仅是在浏览器窗口中发生的，也是在整个屏幕上发生的。可以通过 event 对象的 screenX 和 screenY 属性获取鼠标光标在屏幕上的坐标。图 17-5 展示了浏览器中触发鼠标事件的光标的**屏幕坐标**。

图 17-5

可以像下面这样获取鼠标事件的屏幕坐标：

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
  console.log(`Screen coordinates: ${event.screenX}, ${event.screenY}`);
});
```

与前面的例子类似，这段代码也为 元素指定了 onclick 事件处理程序。当元素被点击时，会通过控制台打印出事件的屏幕坐标。

4. 修饰键

虽然鼠标事件主要是通过鼠标触发的，但有时候要确定用户想实现的操作，还要考虑键盘按键的状态。键盘上的**修饰键** Shift、Ctrl、Alt 和 Meta 经常用于修改鼠标事件的行为。DOM 规定了 4 个属性来表示这几个修饰键的状态：shiftKey、ctrlKey、altKey 和 metaKey。这几属性会在各自对应的修饰键被按下时包含布尔值 true，没有被按下时包含 false。在鼠标事件发生的，可以通过这几个属性来检测修饰键是否被按下。来看下面的例子，其中在 click 事件发生时检测了每个修饰键的状态：

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
  let 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");
  }

  console.log("Keys: " + keys.join(","));

});
```

在这个例子中，onclick 事件处理程序检查了不同修饰键的状态。keys 数组中包含了在事件发生时被按下的修饰键的名称。每个对应属性为 true 的修饰键的名称都会添加到 keys 中。最后，事件处理程序会输出所有键的名称。

> **注意**　现代浏览器支持所有这 4 个修饰键。IE8 及更早版本不支持 metaKey 属性。

5. 相关元素

对 mouseover 和 mouseout 事件而言，还存在与事件相关的其他元素。这两个事件都涉及从一个元素的边界之内把光标移到另一个元素的边界之内。对 mouseover 事件来说，事件的主要目标是获得光标的元素，相关元素是失去光标的元素。类似地，对 mouseout 事件来说，事件的主要目标是失去光标的元素，而相关元素是获得光标的元素。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>
```


这个页面中只包含一个 元素。如果光标开始在 元素上，然后从它上面移出，则 元素上会触发 mouseout 事件，相关元素为<body>元素。与此同时，<body>元素上会触发 mouseover 事件，相关元素是 元素。

DOM 通过 event 对象的 relatedTarget 属性提供了相关元素的信息。这个属性只有在 mouseover 和 mouseout 事件发生时才包含值，其他所有事件的这个属性的值都是 null。IE8 及更早版本不支持 relatedTarget 属性，但提供了其他的可以访问到相关元素的属性。在 mouseover 事件触发时，IE 会提供 fromElement 属性，其中包含相关元素。而在 mouseout 事件触发时，IE 会提供 toElement 属性，其中包含相关元素。（IE9 支持所有这些属性。）因此，可以在 EventUtil 中增加一个通用的获取相关属性的方法：

```
var EventUtil = {

  // 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;
    }
  },

  // 其他代码

};
```

与前面介绍的其他跨浏览器方法一样，这个方法同样使用特性检测来确定要返回哪个值。可以像下面这样使用 EventUtil.getRelatedTarget()方法：

```
let div = document.getElementById("myDiv");
div.addEventListener("mouseout", (event) => {
  let target = event.target;
  let relatedTarget = EventUtil.getRelatedTarget(event);
  console.log(
    `Moused out of ${target.tagName} to ${relatedTarget.tagName}`);
});
```

这个例子在 元素上注册了 mouseout 事件处理程序。当事件触发时，就会打印出一条消息说明鼠标从哪个元素移出，移到了哪个元素上。