6. 鼠标按键

只有在元素上单击鼠标主键（或按下键盘上的回车键）时 `click` 事件才会触发，因此按键信息并不是必需的。对 `mousedown` 和 `mouseup` 事件来说，`event` 对象上会有一个 `button` 属性，表示按下或释放的是哪个按键。DOM 为这个 `button` 属性定义了 3 个值：0 表示鼠标主键、1 表示鼠标中键（通常也是滚轮键）、2 表示鼠标副键。按照惯例，鼠标主键通常是左边的按键，副键通常是右边的按键。

IE8 及更早版本也提供了 `button` 属性，但这个属性的值与前面说的完全不同：

- 0，表示没有按下任何键；
- 1，表示按下鼠标主键；
- 2，表示按下鼠标副键；
- 3，表示同时按下鼠标主键、副键；
- 4，表示按下鼠标中键；
- 5，表示同时按下鼠标主键和中键；
- 6，表示同时按下鼠标副键和中键；
- 7，表示同时按下 3 个键。

很显然，DOM 定义的 `button` 属性比 IE 这一套更简单也更有用，毕竟同时按多个鼠标键的情况很少见。为此，实践中基本上都以 DOM 的 `button` 属性为准，这是因为除 IE8 及更早版本外的所有主流浏览器都原生支持。主、中、副键的定义非常明确，而 IE 定义的其他情形都可以翻译为按下其中某个键，而且优先翻译为主键。比如，IE 返回 5 或 7 时，就会对应到 DOM 的 0。

7. 额外事件信息

DOM2 Events 规范在 `event` 对象上提供了 `detail` 属性，以给出关于事件的更多信息。对鼠标事件来说，`detail` 包含一个数值，表示在给定位置上发生了多少次单击。单击相当于在同一个像素上发生一次 `mousedown` 紧跟一次 `mouseup`。`detail` 的值从 1 开始，每次单击会加 1。如果鼠标在 `mousedown` 和 `mouseup` 之间移动了，则 `detail` 会重置为 0。

IE 还为每个鼠标事件提供了以下额外信息：

- `altLeft`，布尔值，表示是否按下了左 Alt 键（如果 `altLeft` 是 `true`，那么 `altKey` 也是 `true`）；
- `ctrlLeft`，布尔值，表示是否按下了左 Ctrl 键（如果 `ctrlLeft` 是 `true`，那么 `ctrlKey` 也是 `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，表示是否按下了左 Shift 键（如果 `shiftLeft` 是 `true`，那么 `shiftKey` 也是 `true`）。

这些属性的作用有限，这是因为只有 IE 支持。而且，它们提供的信息要么没必要，要么可以通过其他方式计算。

8. `mousewheel` 事件

IE6 首先实现了 `mousewheel` 事件。之后，Opera、Chrome 和 Safari 也跟着实现了。`mousewheel` 事件会在用户使用鼠标滚轮时触发，包括在垂直方向上任意滚动。这个事件会在任何元素上触发，并（在 IE8 中）冒泡到 `document` 和（在所有现代浏览器中）`window`。`mousewheel` 事件的 `event` 对象包含鼠标事件的所有标准信息，此外还有一个名为 `wheelDelta` 的新属性。当鼠标滚轮向前滚动时，`wheelDelta` 每次都是+120；而当鼠标滚轮向后滚动时，`wheelDelta` 每次都是–120（见图 17-6）。

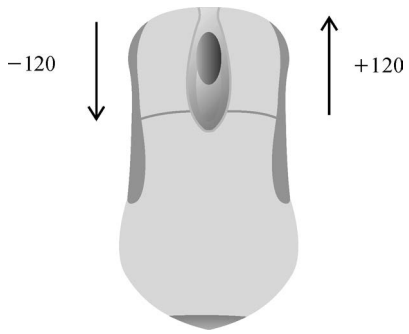

图 17-6

可以为页面上的任何元素或文档添加 onmousewheel 事件处理程序，以处理所有鼠标滚轮交互，比如：

```
document.addEventListener("mousewheel", (event) => {
  console.log(event.wheelDelta);
});
```

这个例子简单地显示了鼠标滚轮事件触发时 wheelDelta 的值。多数情况下只需知道滚轮滚动的方向，而这通过 wheelDelta 值的符号就可以知道。

> **注意** HTML5 也增加了 mousewheel 事件，以反映大多数浏览器对它的支持。

9. 触摸屏设备

iOS 和 Android 等触摸屏设备的实现大相径庭，因为触摸屏通常不支持鼠标操作。在为触摸屏设备开发时，要记住以下事项。

- 不支持 dblclick 事件。双击浏览器窗口可以放大，但没有办法覆盖这个行为。
- 单指点触屏幕上的可点击元素会触发 mousemove 事件。如果操作会导致内容变化，则不会再触发其他事件。如果屏幕上没有变化，则会相继触发 mousedown、mouseup 和 click 事件。点触不可点击的元素不会触发事件。可点击元素是指点击时有默认动作的元素（如链接）或指定了 onclick 事件处理程序的元素。
- mousemove 事件也会触发 mouseover 和 mouseout 事件。
- 双指点触屏幕并滑动导致页面滚动时会触发 mousewheel 和 scroll 事件。

10. 无障碍问题

如果 Web 应用或网站必须考虑残障人士，特别是使用屏幕阅读器的用户，那么必须小心使用鼠标事件。如前所述，按回车键可以触发 click 事件，但其他鼠标事件不能通过键盘触发。因此，建议不要使用 click 事件之外的其他鼠标事件向用户提示功能或触发代码执行，这是因为其他鼠标事件会严格妨碍盲人或视障用户使用。以下是几条使用鼠标事件时应该遵循的无障碍建议。

- 使用 click 事件执行代码。有人认为，当使用 onmousedown 执行代码时，应用程序会运行得更快。对视力正常用户来说确实如此。但在屏幕阅读器上，这样会导致代码无法执行，这是因为屏幕阅读器无法触发 mousedown 事件。

- 不要使用 mouseover 向用户显示新选项。同样，原因是屏幕阅读器无法触发 mousedown 事件。如果必须要通过这种方式显示新选项，那么可以考虑显示相同信息的键盘快捷键。
- 不要使用 dblclick 执行重要的操作，这是因为键盘不能触发这个事件。

遵循这些简单的建议可以极大提升 Web 应用或网站对残障人士的无障碍性。

> **注意**　要了解更多关于网站无障碍的信息，可以参考 WebAIM 网站。

17.4.4　键盘与输入事件

键盘事件是用户操作键盘时触发的。DOM2 Events 最初定义了键盘事件，但该规范在最终发布前删除了相应内容。因此，键盘事件很大程度上是基于原始的 DOM0 实现的。

DOM3 Events 为键盘事件提供了一个首先在 IE9 中完全实现的规范。其他浏览器也开始实现该规范，但仍然存在很多遗留的实现。

键盘事件包含 3 个事件：

- keydown，用户按下键盘上某个键时触发，而且持续按住会重复触发。
- keypress，用户按下键盘上某个键并产生字符时触发，而且持续按住会重复触发。Esc 键也会触发这个事件。DOM3 Events 废弃了 keypress 事件，而推荐 textInput 事件。
- keyup，用户释放键盘上某个键时触发。

虽然所有元素都支持这些事件，但当用户在文本框中输入内容时最容易看到。

输入事件只有一个，即 textInput。这个事件是对 keypress 事件的扩展，用于在文本显示给用户之前更方便地截获文本输入。textInput 会在文本被插入到文本框之前触发。

当用户按下键盘上的某个字符键时，首先会触发 keydown 事件，然后触发 keypress 事件，最后触发 keyup 事件。注意，这里 keydown 和 keypress 事件会在文本框出现变化之前触发，而 keyup 事件会在文本框出现变化之后触发。如果一个字符键被按住不放，keydown 和 keypress 就会重复触发，直到这个键被释放。

对于非字符键，在键盘上按一下这个键，会先触发 keydown 事件，然后触发 keyup 事件。如果按住某个非字符键不放，则会重复触发 keydown 事件，直到这个键被释放，此时会触发 keyup 事件。

> **注意**　键盘事件支持与鼠标事件相同的修饰键。shiftKey、ctrlKey、altKey 和 metaKey 属性在键盘事件中都是可用的。IE8 及更早版本不支持 metaKey 属性。

1. 键码

对于 keydown 和 keyup 事件，event 对象的 keyCode 属性中会保存一个键码，对应键盘上特定的一个键。对于字母和数字键，keyCode 的值与小写字母和数字的 ASCII 编码一致。比如数字 7 键的 keyCode 为 55，而字母 A 键的 keyCode 为 65，而且跟是否按了 Shift 键无关。DOM 和 IE 的 event 对象都支持 keyCode 属性。下面这个例子展示了如何使用 keyCode 属性：

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keyup", (event) => {
  console.log(event.keyCode);
});
```

这个例子在 keyup 事件触发时直接显示出 event 对象的 keyCode 属性值。下表给出了键盘上所有非字符键的键码。

键	键 码	键	键 码
Backspace	8	数字键盘 8	104
Tab	9	数字键盘 9	105
Enter	13	数字键盘+	107
Shift	16	减号（包含数字和非数字键盘）	109
Ctrl	17	数字键盘.	110
Alt	18	数字键盘/	111
Pause/Break	19	F1	112
Caps Lock	20	F2	113
Esc	27	F3	114
Page Up	33	F4	115
Page Down	34	F5	116
End	35	F6	117
Home	36	F7	118
左箭头	37	F8	119
上箭头	38	F9	120
右箭头	39	F10	121
下箭头	40	F11	122
Ins	45	F12	123
Del	46	Num Lock	144
左 Windows	91	Scroll Lock	145
右 Windows	92	分号（IE/Safari/Chrome）	186
Context Menu	93	分号（Opera/FF）	59
数字键盘 0	96	小于号	188
数字键盘 1	97	大于号	190
数字键盘 2	98	反斜杠	191
数字键盘 3	99	重音符（`）	192
数字键盘 4	100	等于号	61
数字键盘 5	101	左中括号	219
数字键盘 6	102	反斜杠（\）	220
数字键盘 7	103	右中括号	221
		单引号	222

2. 字符编码

在 keypress 事件发生时，意味着按键会影响屏幕上显示的文本。对插入或移除字符的键，所有浏览器都会触发 keypress 事件，其他键则取决于浏览器。因为 DOM3 Events 规范才刚刚开始实现，所以不同浏览器之间的实现存在显著差异。

浏览器在 event 对象上支持 charCode 属性，只有发生 keypress 事件时这个属性才会被设置值，

包含的是按键字符对应的 ASCII 编码。通常，charCode 属性的值是 0，在 keypress 事件发生时则是对应按键的键码。IE8 及更早版本和 Opera 使用 keyCode 传达字符的 ASCII 编码。要以跨浏览器方式获取字符编码，首先要检查 charCode 属性是否有值，如果没有再使用 keyCode，如下所示：

```
var EventUtil = {

  // 其他代码

  getCharCode: function(event) {
    if (typeof event.charCode == "number") {
      return event.charCode;
    } else {
      return event.keyCode;
    }
  },

  // 其他代码
};
```

这个方法检测 charCode 属性是否为数值（在不支持的浏览器中是 undefined）。如果是数值，则返回。否则，返回 keyCode 值。可以像下面这样使用：

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keypress", (event) => {
  console.log(EventUtil.getCharCode(event));
});
```

一旦有了字母编码，就可以使用 String.fromCharCode()方法将其转换为实际的字符了。

3. DOM3 的变化

尽管所有浏览器都实现了某种形式的键盘事件，DOM3 Events 还是做了一些修改。比如，DOM3 Events 规范并未规定 charCode 属性，而是定义了 key 和 char 两个新属性。

其中，key 属性用于替代 keyCode，且包含字符串。在按下字符键时，key 的值等于文本字符（如“k”或“M”）；在按下非字符键时，key 的值是键名（如“Shift”或“ArrowDown”）。char 属性在按下字符键时与 key 类似，在按下非字符键时为 null。

IE 支持 key 属性但不支持 char 属性。Safari 和 Chrome 支持 keyIdentifier 属性，在按下非字符键时返回与 key 一样的值（如“Shift”）。对于字符键，keyIdentifier 返回以“U+0000”形式表示 Unicode 值的字符串形式的字符编码。

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keypress", (event) => {
  let identifier = event.key || event.keyIdentifier;
  if (identifier) {
    console.log(identifier);
  }
});
```

由于缺乏跨浏览器支持，因此不建议使用 key、keyIdentifier、和 char。

DOM3 Events 也支持一个名为 location 的属性，该属性是一个数值，表示是在哪里按的键。可能的值为：0 是默认键，1 是左边（如左边的 Alt 键），2 是右边（如右边的 Shift 键），3 是数字键盘，4 是移动设备（即虚拟键盘），5 是游戏手柄（如任天堂 Wii 控制器）。IE9 支持这些属性。Safari 和 Chrome 支持一个等价的 keyLocation 属性，但由于实现有问题，这个属性值始终为 0，除非是数字键盘（此

时值为 3），值永远不会是 1、2、4、5。

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keypress", (event) => {
  let loc = event.location || event.keyLocation;
  if (loc) {
    console.log(loc);
  }
});
```

与 key 属性类似，location 属性也没有得到广泛支持，因此不建议在跨浏览器开发时使用。

最后一个变化是给 event 对象增加了 getModifierState()方法。这个方法接收一个参数，一个等于 Shift、Control、Alt、AltGraph 或 Meta 的字符串，表示要检测的修饰键。如果给定的修饰键处于激活状态（键被按住），则方法返回 true，否则返回 false：

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keypress", (event) => {
  if (event.getModifierState) {
    console.log(event.getModifierState("Shift"));
  }
});
```

当然，event 对象已经通过 shiftKey、altKey、ctrlKey 和 metaKey 属性暴露了这些信息。

4. textInput 事件

DOM3 Events 规范增加了一个名为 textInput 的事件，其在字符被输入到可编辑区域时触发。作为对 keypress 的替代，textInput 事件的行为有些不一样。一个区别是 keypress 会在任何可以获得焦点的元素上触发，而 textInput 只在可编辑区域上触发。另一个区别是 textInput 只在有新字符被插入时才会触发，而 keypress 对任何可能影响文本的键都会触发（包括退格键）。

因为 textInput 事件主要关注字符，所以在 event 对象上提供了一个 data 属性，包含要插入的字符（不是字符编码）。data 的值始终是要被插入的字符，因此如果在按 S 键时没有按 Shift 键，data 的值就是"s"，但在按 S 键时同时按 Shift 键，data 的值则是"S"。

textInput 事件可以这样来用：

```
let textbox = document.getElementById("myText");
textbox.addEventListener("textInput", (event) => {
  console.log(event.data);
});
```

这个例子会实时把输入文本框的文本通过日志打印出来。

event 对象上还有一个名为 inputMethod 的属性，该属性表示向控件中输入文本的手段。可能的值如下：

- 0，表示浏览器不能确定是什么输入手段；
- 1，表示键盘；
- 2，表示粘贴；
- 3，表示拖放操作；
- 4，表示 IME；
- 5，表示表单选项；
- 6，表示手写（如使用手写笔）；
- 7，表示语音；

- 8，表示组合方式；
- 9，表示脚本。

使用这些属性，可以确定用户是如何将文本输入到控件中的，从而可以辅助验证。

5. 设备上的键盘事件

任天堂 Wii 会在用户按下 Wii 遥控器上的键时触发键盘事件。虽然不能访问 Wii 遥控器上所有的键，但其中一些键可以触发键盘事件。图 17-7 中标识出了某些键的键码。

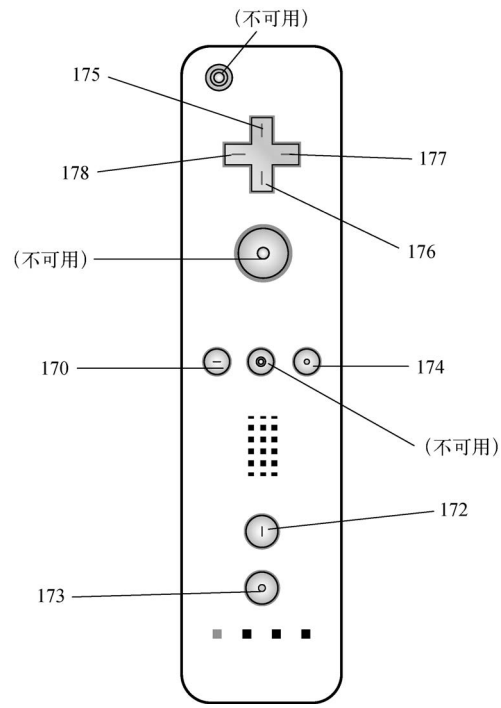

图　17-7

如图所示，按下十字键（175~178）、减号键（170）、加号键（174）、1（172）或 2（173）按钮会触发键盘事件。无法判断电源键、A、B 或 Home 键是否已按下。

17.4.5　合成事件

合成事件是 DOM3 Events 中新增的，用于处理通常使用 IME 输入时的复杂输入序列。IME 可以让用户输入物理键盘上没有的字符。例如，使用拉丁字母键盘的用户还可以使用 IME 输入日文。IME 通常需要同时按下多个键才能输入一个字符。合成事件用于检测和控制这种输入。合成事件有以下 3 种：

- `compositionstart`，在 IME 的文本合成系统打开时触发，表示输入即将开始；
- `compositionupdate`，在新字符插入输入字段时触发；
- `compositionend`，在 IME 的文本合成系统关闭时触发，表示恢复正常键盘输入。

合成事件在很多方面与输入事件很类似。在合成事件触发时，事件目标是接收文本的输入字段。唯一增加的事件属性是 `data`，其中包含的值视情况而异：

- 在 compositionstart 事件中，包含正在编辑的文本（例如，已经选择了文本但还没替换）；
- 在 compositionupdate 事件中，包含要插入的新字符；
- 在 compositionend 事件中，包含本次合成过程中输入的全部内容。

与文本事件类似，合成事件可以用来在必要时过滤输入内容。可以像下面这样使用合成事件：

```
let textbox = document.getElementById("myText");
textbox.addEventListener("compositionstart", (event) => {
  console.log(event.data);
});
textbox.addEventListener("compositionupdate", (event) => {
  console.log(event.data);
});
textbox.addEventListener("compositionend", (event) => {
  console.log(event.data);
});
```


17.4.6 变化事件

DOM2 的**变化事件**（Mutation Events）是为了在 DOM 发生变化时提供通知。

> **注意** 这些事件已经被废弃，浏览器已经在有计划地停止对它们的支持。变化事件已经被 Mutation Observers 所取代，可以参考第 14 章中的介绍。

17.4.7 HTML5 事件

DOM 规范并未涵盖浏览器都支持的所有事件。很多浏览器根据特定的用户需求或使用场景实现了自定义事件。HTML5 详尽地列出了浏览器支持的所有事件。本节讨论 HTML5 中得到浏览器较好支持的一些事件。注意这些并不是浏览器支持的所有事件。（本书后面也会涉及一些其他事件。）

1. contextmenu 事件

Windows 95 通过单击鼠标右键为 PC 用户增加了上下文菜单的概念。不久，这个概念也在 Web 上得以实现。开发者面临的问题是如何确定何时该显示上下文菜单（在 Windows 上是右击鼠标，在 Mac 上是 Ctrl+单击），以及如何避免默认的上下文菜单起作用。结果就出现了 contextmenu 事件，以专门用于表示何时该显示上下文菜单，从而允许开发者取消默认的上下文菜单并提供自定义菜单。

contextmenu 事件冒泡，因此只要给 document 指定一个事件处理程序就可以处理页面上的所有同类事件。事件目标是触发操作的元素。这个事件在所有浏览器中都可以取消，在 DOM 合规的浏览器中使用 event.preventDefault()，在 IE8 及更早版本中将 event.returnValue 设置为 false。contextmenu 事件应该算一种鼠标事件，因此 event 对象上的很多属性都与光标位置有关。通常，自定义的上下文菜单都是通过 oncontextmenu 事件处理程序触发显示，并通过 onclick 事件处理程序触发隐藏的。来看下面的例子：

```
<!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.somewhere.com"> somewhere</a></li>
      <li><a href="http://www.wrox.com">Wrox site</a></li>
      <li><a href="http://www.somewhere-else.com">somewhere-else</a></li>
    </ul>
  </body>
  </html>
```

这个例子中的 元素有一个上下文菜单<ul>。作为上下文菜单，<ul>元素初始时是隐藏的。以下是实现上下文菜单功能的 JavaScript 代码：

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

  div.addEventListener("contextmenu", (event) => {
    event.preventDefault();

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

  document.addEventListener("click", (event) => {
    document.getElementById("myMenu").style.visibility = "hidden";
  });
});
```

这里在 元素上指定了一个 oncontextmenu 事件处理程序。这个事件处理程序首先取消默认行为，确保不会显示浏览器默认的上下文菜单。接着基于 event 对象的 clientX 和 clientY 属性把<ul>元素放到适当位置。最后一步通过将 visibility 属性设置为"visible"让自定义上下文菜单显示出来。另外，又给 document 添加了一个 onclick 事件处理程序，以便在单击事件发生时隐藏上下文菜单（系统上下文菜单就是这样隐藏的）。

虽然这个例子很简单，但它是网页中所有自定义上下文菜单的基础。在这个简单例子的基础上，再添加一些 CSS，上下文菜单就会更漂亮。

2. beforeunload 事件

beforeunload 事件会在 window 上触发，用意是给开发者提供阻止页面被卸载的机会。这个事件会在页面即将从浏览器中卸载时触发，如果页面需要继续使用，则可以不被卸载。这个事件不能取消，否则就意味着可以把用户永久阻拦在一个页面上。相反，这个事件会向用户显示一个确认框，其中的消息表明浏览器即将卸载页面，并请用户确认是希望关闭页面，还是继续留在页面上（见图 17-8）。

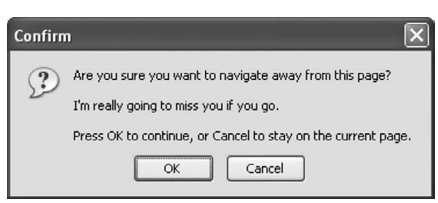

图　17-8

为了显示类似图 17-8 的确认框，需要将 event.returnValue 设置为要在确认框中显示的字符串（对于 IE 和 Firefox 来说），并将其作为函数值返回（对于 Safari 和 Chrome 来说），如下所示：

```
window.addEventListener("beforeunload", (event) => {
  let message = "I'm really going to miss you if you go.";
  event.returnValue = message;
  return message;
});
```


3. DOMContentLoaded 事件

window 的 load 事件会在页面完全加载后触发，因为要等待很多外部资源加载完成，所以会花费较长时间。而 DOMContentLoaded 事件会在 DOM 树构建完成后立即触发，而不用等待图片、JavaScript 文件、CSS 文件或其他资源加载完成。相对于 load 事件，DOMContentLoaded 可以让开发者在外部资源下载的同时就能指定事件处理程序，从而让用户能够更快地与页面交互。

要处理 DOMContentLoaded 事件，需要给 document 或 window 添加事件处理程序（实际的事件目标是 document，但会冒泡到 window）。下面是一个在 document 上监听 DOMContentLoaded 事件的例子：

```
document.addEventListener("DOMContentLoaded", (event) => {
  console.log("Content loaded");
});
```

DOMContentLoaded 事件的 event 对象中不包含任何额外信息（除了 target 等于 document）。

DOMContentLoaded 事件通常用于添加事件处理程序或执行其他 DOM 操作。这个事件始终在 load 事件之前触发。

对于不支持 DOMContentLoaded 事件的浏览器，可以使用超时为 0 的 setTimeout()函数，通过其回调来设置事件处理程序，比如：

```
setTimeout(() => {
  // 在这里添加事件处理程序
}, 0);
```

以上代码本质上意味着在当前 JavaScript 进程执行完毕后立即执行这个回调。页面加载和构建期间，只有一个 JavaScript 进程运行。所以可以在这个进程空闲后立即执行回调，至于是否与同一个浏览器或同一页面上不同脚本的 DOMContentLoaded 触发时机一致并无绝对把握。为了尽可能早一些执行，以上代码最好是页面上的第一个超时代码。即使如此，考虑到各种影响因素，也不一定保证能在 load 事件之前执行超时回调。

4. readystatechange 事件

IE 首先在 DOM 文档的一些地方定义了一个名为 readystatechange 事件。这个有点神秘的事件旨在提供文档或元素加载状态的信息，但行为有时候并不稳定。支持 readystatechange 事件的每个对象都有一个 readyState 属性，该属性具有一个以下列出的可能的字符串值。

- uninitialized：对象存在并尚未初始化。
- loading：对象正在加载数据。
- loaded：对象已经加载完数据。
- interactive：对象可以交互，但尚未加载完成。
- complete：对象加载完成。