enabled
frequencychange

Fullscreen API events
fullscreenchange
fullscreenerror

Gamepad API events
gamepadconnected
gamepaddisconnected

HTML DOM events
DOMContentLoaded
abort
afterprint
afterscriptexecute
beforeprint
beforescriptexecute
beforeunload
blur
cancel
canplay
canplaythrough
change
click
close
connect
contextmenu
durationchange
emptied
error
focus
hashchange
input
invalid
languagechange
load
loadeddata
loadedmetadata
loadend
loadstart
message
offline
online
open
pagehide
pageshow
play
playing
popstate
progress
readystatechange
rejectionhandled
reset
seeked
seeking
select
show
sort
stalled
storage
submit
suspend
timeupdate
toggle
unhandledrejection
unload
volumechange
waiting

HTML Drag and Drop API events
drag
dragend
dragenter
dragexit
dragleave
dragover
dragstart
drop

IndexedDB events
abort
blocked
close
complete
error
success
upgradeneeded
versionchange

Inter-App Connection API events
message

Media Capture and Streams events
active
addtrack
devicechange
ended
inactive
mute
overconstrained
ratechange
removetrack
started
unmute

Media Source Extensions events
abort
addsourcebuffer
error
removesourcebuffer
sourceclose
sourceended
sourceopen
update
updateend
updatestart

MediaStream Recording events
dataavailable
error
pause
resume
start
stop

Mobile Connection API events
cardstatechange
icccardlockerror

Mobile Messaging API events
close
deliveryerror
deliverysuccess
error
failed
message
open
received
retrieving
sending
sent

Network Information API events
change

Page Visibility API events
visibilitychange

Payment Request API events
shippingaddresschange
shippingoptionchange

Performance API events
resourcetimingbufferfull

Pointer events
gotpointercapture
lostpointercapture
pointercancel
pointerdown
pointerenter
pointerleave
pointermove
pointerout
pointerover
pointerup

Pointer Lock API events
pointerlockchange
pointerlockerror

Presentation API events
change
sessionavailable
sessionconnect

Proximity events
deviceproximity
userproximity

Push API events
push
pushsubscriptionchange

Screen Orientation API events
change

Selection API events
selectionchange
selectstart

Server Sent events
error
message
open

Service Workers API events
activate
controllerchange
error
fetch
install
message
statechange
updatefound

Settings API events
settingchange

Simple Push API events
error
success

Speaker Manager API events
speakerforcedchange

SVG events
DOMAttrModified
DOMCharacterDataModified
DOMNodeInserted
DOMNodeInsertedIntoDocument
DOMNodeRemoved
DOMNodeRemovedFromDocument
DOMSubtreeModified
SVGAbort
SVGError
SVGLoad
SVGResize
SVGScroll
SVGUnload
SVGZoom
activate
beginEvent
click
endEvent
focusin
focusout
mousedown
mousemove
mouseout
mouseover
mouseup
repeatEvent

TCP Socket API events
connect
data
drain
error

Time and Clock API events
moztimechange

Touch events
touchcancel
touchend
touchmove
touchstart

TV API events
currentchannelchanged
currentsourcechanged
eitbroadcasted
scanningstatechanged

UDP Socket API events
message

Web Audio API events
audioprocess
complete

ended
loaded
message
nodecreate
statechange

Web Components events
slotchange

WebGL events
webglcontextcreationerror
webglcontextlost
webglcontextrestored

Web Manifest events
install

Web MIDI API events
midimessage
statechange

Web Notifications events
click
close
error
show

WebRTC events
addstream
close
datachannel
error
icecandidate
iceconnectionstatechange
icegatheringstatechange
identityresult
idpassertionerror
idpvalidationerror
isolationchange
message
negotiationneeded
open
peeridentity
removestream
signalingstatechange
tonechange

Websockets API events
close
error
message
open

Web Speech API events
audioend
audiostart
boundary
end_(SpeechRecognition)
end_(SpeechSynthesis)
error_(SpeechRecognitionError)
error_(SpeechSynthesis)
mark
nomatch
pause_(SpeechSynthesis)
result
resume
soundend
soundstart
speechend
speechstart
start_(SpeechRecognition)
start_(SpeechSynthesis)

Web Storage API events
storage

Web Telephony API events
incoming

WebVR API events
vrdisplayactivate
vrdisplayblur
vrdisplayconnected
vrdisplaydeactivate
vrdisplaydisconnected
vrdisplayfocus
vrdisplaypresentchange

WebVTT events
addtrack
change
cuechange
enter

```
exit
removetrack

WiFi Information API events
connectioninfoupdate
statuschange

WiFi P2P API events
disabled
enabled
peerinfoupdate
statuschange

XMLHttpRequest events
abort
error
load
loadend
loadstart
progress
readystatechange
timeout
```

17.5　内存与性能

因为事件处理程序在现代 Web 应用中可以实现交互，所以很多开发者会错误地在页面中大量使用它们。在创建 GUI 的语言如 C#中，通常会给 GUI 上的每个按钮设置一个 onclick 事件处理程序。这样做不会有什么性能损耗。在 JavaScript 中，页面中事件处理程序的数量与页面整体性能直接相关。原因有很多。首先，每个函数都是对象，都占用内存空间，对象越多，性能越差。其次，为指定事件处理程序所需访问 DOM 的次数会先期造成整个页面交互的延迟。只要在使用事件处理程序时多注意一些方法，就可以改善页面性能。

17.5.1　事件委托

“过多事件处理程序”的解决方案是使用**事件委托**。事件委托利用事件冒泡，可以只使用一个事件处理程序来管理一种类型的事件。例如，click 事件冒泡到 document。这意味着可以为整个页面指定一个 onclick 事件处理程序，而不用为每个可点击元素分别指定事件处理程序。比如有以下 HTML：

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

这里的 HTML 包含 3 个列表项，在被点击时应该执行某个操作。对此，通常的做法是像这样指定 3 个事件处理程序：

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

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

item2.addEventListener("click", (event) => {
  document.title = "I changed the document's title";
});

item3.addEventListener("click", (event) => {
  console.log("hi");
});
```

如果对页面中所有需要使用 onclick 事件处理程序的元素都如法炮制，结果就会出现大片雷同的只为指定事件处理程序的代码。使用事件委托，只要给所有元素共同的祖先节点添加一个事件处理程序，就可以解决问题。比如：

```
let list = document.getElementById("myLinks");

list.addEventListener("click", (event) => {
  let target = event.target;

  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":
      console.log("hi");
      break;
  }
});
```

这里只给<ul id="myLinks">元素添加了一个 onclick 事件处理程序。因为所有列表项都是这个元素的后代，所以它们的事件会向上冒泡，最终都会由这个函数来处理。但事件目标是每个被点击的列表项，只要检查 event 对象的 id 属性就可以确定，然后再执行相应的操作即可。相对于前面不使用事件委托的代码，这里的代码不会导致先期延迟，因为只访问了一个 DOM 元素和添加了一个事件处理程序。结果对用户来说没有区别，但这种方式占用内存更少。所有使用按钮的事件（大多数鼠标事件和键盘事件）都适用于这个解决方案。

只要可行，就应该考虑只给 document 添加一个事件处理程序，通过它处理页面中所有某种类型的事件。相对于之前的技术，事件委托具有如下优点。

- document 对象随时可用，任何时候都可以给它添加事件处理程序（不用等待 DOMContentLoaded 或 load 事件）。这意味着只要页面渲染出可点击的元素，就可以无延迟地起作用。
- 节省花在设置页面事件处理程序上的时间。只指定一个事件处理程序既可以节省 DOM 引用，也可以节省时间。
- 减少整个页面所需的内存，提升整体性能。

最适合使用事件委托的事件包括：click、mousedown、mouseup、keydown 和 keypress。mouseover 和 mouseout 事件冒泡，但很难适当处理，且经常需要计算元素位置（因为 mouseout 会在光标从一个元素移动到它的一个后代节点以及移出元素之外时触发）。

17.5.2 删除事件处理程序

把事件处理程序指定给元素后，在浏览器代码和负责页面交互的 JavaScript 代码之间就建立了联系。这种联系建立得越多，页面性能就越差。除了通过事件委托来限制这种连接之外，还应该及时删除不用的事件处理程序。很多 Web 应用性能不佳都是由于无用的事件处理程序长驻内存导致的。

导致这个问题的原因主要有两个。第一个是删除带有事件处理程序的元素。比如通过真正的 DOM

方法 removeChild()或 replaceChild()删除节点。最常见的还是使用 innerHTML 整体替换页面的某一部分。这时候，被 innerHTML 删除的元素上如果有事件处理程序，就不会被垃圾收集程序正常清理。比如下面的例子：

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

    // 执行操作

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

这里的按钮在 元素中。单击按钮，会将自己删除并替换为一条消息，以阻止双击发生。这是很多网站上常见的做法。问题在于，按钮被删除之后仍然关联着一个事件处理程序。在 元素上设置 innerHTML 会完全删除按钮，但事件处理程序仍然挂在按钮上面。某些浏览器，特别是 IE8 及更早版本，在这时候就会有问题了。很有可能元素的引用和事件处理程序的引用都会残留在内存中。如果知道某个元素会被删除，那么最好在删除它之前手工删除它的事件处理程序，比如：

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

    // 执行操作

    btn.onclick = null;   // 删除事件处理程序

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

在这个重写后的例子中，设置 元素的 innerHTML 属性之前，按钮的事件处理程序先被删除了。这样就可以确保内存被回收，按钮也可以安全地从 DOM 中删掉。

但也要注意，在事件处理程序中删除按钮会阻止事件冒泡。只有事件目标仍然存在于文档中时，事件才会冒泡。

> **注意**　事件委托也有助于解决这种问题。如果提前知道页面某一部分会被使用 innerHTML 删除，就不要直接给该部分中的元素添加事件处理程序了。把事件处理程序添加到更高层级的节点上同样可以处理该区域的事件。

另一个可能导致内存中残留引用的问题是页面卸载。同样，IE8 及更早版本在这种情况下有很多问题，不过好像所有浏览器都会受这个问题影响。如果在页面卸载后事件处理程序没有被清理，则它们仍

然会残留在内存中。之后，浏览器每次加载和卸载页面（比如通过前进、后退或刷新），内存中残留对象的数量都会增加，这是因为事件处理程序不会被回收。

一般来说，最好在 onunload 事件处理程序中趁页面尚未卸载先删除所有事件处理程序。这时候也能体现使用事件委托的优势，因为事件处理程序很少，所以很容易记住要删除哪些。关于卸载页面时的清理，可以记住一点：onload 事件处理程序中做了什么，最好在 onunload 事件处理程序中恢复。

> **注意** 在页面中使用 onunload 事件处理程序意味着页面不会被保存在往返缓存（bfcache）中。如果这对应用很重要，可以考虑只在 IE 中使用 onunload 来删除事件处理程序。

17.6 模拟事件

事件就是为了表示网页中某个有意义的时刻。通常，事件都是由用户交互或浏览器功能触发。事实上，可能很少有人知道可以通过 JavaScript 在任何时候触发任意事件，而这些事件会被当成浏览器创建的事件。这意味着同样会有事件冒泡，因而也会触发相应的事件处理程序。这种能力在测试 Web 应用时特别有用。DOM3 规范指明了模拟特定类型事件的方式。IE8 及更早版本也有自己模拟事件的方式。

17.6.1 DOM 事件模拟

任何时候，都可以使用 document.createEvent()方法创建一个 event 对象。这个方法接收一个参数，此参数是一个表示要创建事件类型的字符串。在 DOM2 中，所有这些字符串都是英文复数形式，但在 DOM3 中，又把它们改成了英文单数形式。可用的字符串值是以下值之一。

- "UIEvents"（DOM3 中是"UIEvent"）：通用用户界面事件（鼠标事件和键盘事件都继承自这个事件）。
- "MouseEvents"（DOM3 中是"MouseEvent"）：通用鼠标事件。
- "HTMLEvents"（DOM3 中没有）：通用 HTML 事件（HTML 事件已经分散到了其他事件大类中）。

注意，键盘事件不是在 DOM2 Events 中规定的，而是后来在 DOM3 Events 中增加的。

创建 event 对象之后，需要使用事件相关的信息来初始化。每种类型的 event 对象都有特定的方法，可以使用相应数据来完成初始化。方法的名字并不相同，这取决于调用 createEvent()时传入的参数。

事件模拟的最后一步是触发事件。为此要使用 dispatchEvent()方法，这个方法存在于所有支持事件的 DOM 节点之上。dispatchEvent()方法接收一个参数，即表示要触发事件的 event 对象。调用 dispatchEvent()方法之后，事件就“转正”了，接着便冒泡并触发事件处理程序执行。

1. 模拟鼠标事件

模拟鼠标事件需要先创建一个新的鼠标 event 对象，然后再使用必要的信息对其进行初始化。要创建鼠标 event 对象，可以调用 createEvent()方法并传入"MouseEvents"参数。这样就会返回一个 event 对象，这个对象有一个 initMouseEvent()方法，用于为新对象指定鼠标的特定信息。initMouseEvent()方法接收 15 个参数，分别对应鼠标事件会暴露的属性。这些参数列举如下。

- type（字符串）：要触发的事件类型，如"click"。
- bubbles（布尔值）：表示事件是否冒泡。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。基本上始终是 document.defaultView。
- detail（整数）：关于事件的额外信息。只被事件处理程序使用，通常为 0。
- screenX（整数）：事件相对于屏幕的 *x* 坐标。
- screenY（整数）：事件相对于屏幕的 *y* 坐标。
- clientX（整数）：事件相对于视口的 *x* 坐标。
- clientY（整数）：事件相对于视口的 *y* 坐标。
- ctrlkey（布尔值）：表示是否按下了 Ctrl 键。默认为 false。
- altkey（布尔值）：表示是否按下了 Alt 键。默认为 false。
- shiftkey（布尔值）：表示是否按下了 Shift 键。默认为 false。
- metakey（布尔值）：表示是否按下了 Meta 键。默认为 false。
- button（整数）：表示按下了哪个按钮。默认为 0。
- relatedTarget（对象）：与事件相关的对象。只在模拟 mouseover 和 mouseout 时使用。

显然，initMouseEvent()方法的这些参数与鼠标事件的 event 对象属性是一一对应的。前 4 个参数是正确模拟事件唯一重要的几个参数，这是因为它们是浏览器要用的，其他参数则是事件处理程序要用的。event 对象的 target 属性会自动设置为调用 dispatchEvent()方法时传入的节点。下面来看一个使用默认值模拟单击事件的例子：

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

// 创建 event 对象
let event = document.createEvent("MouseEvents");

// 初始化 event 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);
```

所有鼠标事件，包括 dblclick 都可以像这样在 DOM 合规的浏览器中模拟出来。

2. 模拟键盘事件

如前所述，DOM2 Events 中没有定义键盘事件，因此模拟键盘事件并不直观。键盘事件曾在 DOM2 Events 的草案中提到过，但最终成为推荐标准前又被删掉了。要注意的是，DOM3 Events 中定义的键盘事件与 DOM2 Events 草案最初定义的键盘事件差别很大。

在 DOM3 中创建键盘事件的方式是给 createEvent()方法传入参数"KeyboardEvent"。这样会返回一个 event 对象，这个对象有一个 initKeyboardEvent()方法。这个方法接收以下参数。

- type（字符串）：要触发的事件类型，如"keydown"。
- bubbles（布尔值）：表示事件是否冒泡。为精确模拟键盘事件，应该设置为 true。
- cancelable（布尔值）：表示事件是否可以取消。为精确模拟键盘事件，应该设置为 true。
- view（AbstractView）：与事件关联的视图。基本上始终是 document.defaultView。
- key（字符串）：按下按键的字符串代码。

- location（整数）：按下按键的位置。0 表示默认键，1 表示左边，2 表示右边，3 表示数字键盘，4 表示移动设备（虚拟键盘），5 表示游戏手柄。
- modifiers（字符串）：空格分隔的修饰键列表，如"Shift"。
- repeat（整数）：连续按了这个键多少次。

注意，DOM3 Events 废弃了 keypress 事件，因此只能通过上述方式模拟 keydown 和 keyup 事件：

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

// 按照 DOM3 的方式创建 event 对象
if (document.implementation.hasFeature("KeyboardEvents", "3.0")) {
   event = document.createEvent("KeyboardEvent");

   // 初始化 event 对象
   event.initKeyboardEvent("keydown", true, true, document.defaultView, "a",
                           0, "Shift", 0);

}
// 触发事件
textbox.dispatchEvent(event);
```

这个例子模拟了同时按住 Shift 键和键盘上 A 键的 keydown 事件。在使用 document.createEvent("KeyboardEvent")之前，最好检测一下浏览器对 DOM3 键盘事件的支持情况，其他浏览器会返回非标准的 KeyboardEvent 对象。

Firefox 允许给 createEvent()传入"KeyEvents"来创建键盘事件。这时候返回的 event 对象包含的方法叫 initKeyEvent()，此方法接收以下 10 个参数。

- type（字符串）：要触发的事件类型，如"keydown"。
- bubbles（布尔值）：表示事件是否冒泡。为精确模拟键盘事件，应该设置为 true。
- cancelable（布尔值）：表示事件是否可以取消。为精确模拟键盘事件，应该设置为 true。
- view（AbstractView）：与事件关联的视图，基本上始终是 document.defaultView。
- ctrlkey（布尔值）：表示是否按下了 Ctrl 键。默认为 false。
- altkey（布尔值）：表示是否按下了 Alt 键。默认为 false。
- shiftkey（布尔值）：表示是否按下了 Shift 键。默认为 false。
- metakey（布尔值）：表示是否按下了 Meta 键。默认为 false。
- keyCode（整数）：表示按下或释放键的键码。在 keydown 和 keyup 中使用。默认为 0。
- charCode（整数）：表示按下键对应字符的 ASCII 编码。在 keypress 中使用。默认为 0。

键盘事件也可以通过调用 dispatchEvent()并传入 event 对象来触发，比如：

```
// 仅适用于 Firefox
let textbox = document.getElementById("myTextbox");

// 创建 event 对象
let event = document.createEvent("KeyEvents");

// 初始化 event 对象
event.initKeyEvent("keydown", true, true, document.defaultView, false,
                   false, true, false, 65, 65);

// 触发事件
textbox.dispatchEvent(event);
```