```
range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

let fragment = range.extractContents();
p1.parentNode.appendChild(fragment);
```

这个例子提取了范围的文档片段，然后把它添加到文档<body>元素的最后。（别忘了，在把文档片段传给 appendChild()时，只会添加片段的子树，不包含片段自身。）结果就会得到如下 HTML：

```
<p><b>He</b>rld!</p>
<b>llo</b> wo
[P595 代码三]
```

如果不想把范围从文档中移除，也可以使用 cloneContents()创建一个副本，然后把这个副本插入到文档其他地方。比如：

```
let p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

let fragment = range.cloneContents();
p1.parentNode.appendChild(fragment);
```

这个方法跟 extractContents()很相似，因为它们都返回文档片段。主要区别是 cloneContents()返回的文档片段包含范围中节点的副本，而非实际的节点。执行上面操作之后，HTML 页面会变成这样：

```
<p><b>Hello</b> world!</p>
<b>llo</b> wo
```

此时关键是要知道，为保持结构完好而拆分节点的操作，只有在调用前述方法时才会发生。在 DOM 被修改之前，原始 HTML 会一直保持不变。

16.4.5　范围插入

上一节介绍了移除和复制范围的内容，本节来看一看怎么向范围中插入内容。使用 insertNode()方法可以在范围选区的开始位置插入一个节点。例如，假设我们想在前面例子中的 HTML 中插入如下 HTML：

```
<span style="color: red">Inserted text</span>
```

可以使用下列代码：

```
let p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

let span = document.createElement("span");
span.style.color = "red";
span.appendChild(document.createTextNode("Inserted text"));
range.insertNode(span);
```

运行上面的代码会得到如下 HTML 代码：

```
<p id="p1"><b>He<span style="color: red">Inserted text</span>llo</b> world</p>
```

注意，正好插入到"Hello"中的"llo"之前，也就是范围选区的前面。同时，也要注意原始的 HTML 并没有添加或删除<b>元素，因为这里并没有使用之前提到的方法。使用这个技术可以插入有用的信息，比如在外部链接旁边插入一个小图标。

除了向范围中插入内容，还可以使用 surroundContents()方法插入包含范围的内容。这个方法接收一个参数，即包含范围内容的节点。调用这个方法时，后台会执行如下操作：

(1) 提取出范围的内容；

(2) 在原始文档中范围之前所在的位置插入给定的节点；

(3) 将范围对应文档片段的内容添加到给定节点。

这种功能适合在网页中高亮显示某些关键词，比如：

```
let p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.selectNode(helloNode);
let span = document.createElement("span");
span.style.backgroundColor = "yellow";
range.surroundContents(span);
```

执行以上代码会以黄色背景高亮显示范围选择的文本。得到的 HTML 如下所示：

```
<p><b><span style="background-color:yellow">Hello</span></b> world!</p>
```

为了插入元素，范围中必须包含完整的 DOM 结构。如果范围中包含部分选择的非文节点，这个操作会失败并报错。另外，如果给定的节点是 Document、DocumentType 或 DocumentFragment 类型，也会导致抛出错误。

16.4.6 范围折叠

如果范围并没有选择文档的任何部分，则称为**折叠**（collapsed）。折叠范围有点类似文本框：如果文本框中有文本，那么可以用鼠标选中以高亮显示全部文本。这时候，如果再单击鼠标，则选区会被移除，光标会落在某两个字符中间。而在折叠范围时，位置会被设置为范围与文档交界的地方，可能是范围选区的开始处，也可能是结尾处。图 16-10 展示了范围折叠时会发生什么。

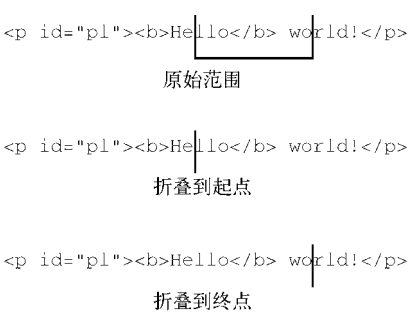

图 16-10

折叠范围可以使用 collapse()方法，这个方法接收一个参数：布尔值，表示折叠到范围哪一端。true 表示折叠到起点，false 表示折叠到终点。要确定范围是否已经被折叠，可以检测范围的 collapsed 属性：

```
range.collapse(true);           // 折叠到起点
console.log(range.collapsed);   // 输出 true
```

测试范围是否被折叠，能够帮助确定范围中的两个节点是否相邻。例如有以下 HTML 代码：

```
<p id="p1">Paragraph 1</p><p
id="p2">Paragraph 2</p>
```

如果事先并不知道标记的结构（比如自动生成的标记），则可以像下面这样创建一个范围：

```
let p1 = document.getElementById("p1"),
    p2 = document.getElementById("p2"),
    range = document.createRange();
range.setStartAfter(p1);
range.setStartBefore(p2);
console.log(range.collapsed);  // true
```

在这种情况下，创建的范围是折叠的，因为 p1 后面和 p2 前面没有任何内容。

16.4.7　范围比较

如果有多个范围，则可以使用 compareBoundaryPoints()方法确定范围之间是否存在公共的边界（起点或终点）。这个方法接收两个参数：要比较的范围和一个常量值，表示比较的方式。这个常量参数包括：

- Range.START_TO_START（0），比较两个范围的起点；
- Range.START_TO_END（1），比较第一个范围的起点和第二个范围的终点；
- Range.END_TO_END（2），比较两个范围的终点；
- Range.END_TO_START（3），比较第一个范围的终点和第二个范围的起点。

compareBoundaryPoints()方法在第一个范围的边界点位于第二个范围的边界点之前时返回-1，在两个范围的边界点相等时返回 0，在第一个范围的边界点位于第二个范围的边界点之后时返回 1。来看下面的例子：

```
let range1 = document.createRange();
let range2 = document.createRange();
let p1 = document.getElementById("p1");

range1.selectNodeContents(p1);
range2.selectNodeContents(p1);
range2.setEndBefore(p1.lastChild);

console.log(range1.compareBoundaryPoints(Range.START_TO_START, range2));  // 0
console.log(range1.compareBoundaryPoints(Range.END_TO_END, range2));      // 1
```

在这段代码中，两个范围的起点是相等的，因为它们都是 selectNodeContents()默认返回的值。因此，比较二者起点的方法返回 0。不过，因为 range2 的终点被使用 setEndBefore()修改了，所以导致 range1 的终点位于 range2 的终点之后（见图 16-11），结果这个方法返回了 1。

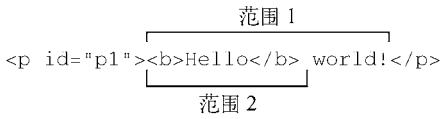

图 16-11

16.4.8 复制范围

调用范围的 cloneRange()方法可以复制范围。这个方法会创建调用它的范围的副本：

```
let newRange = range.cloneRange();
```

新范围包含与原始范围一样的属性，修改其边界点不会影响原始范围。

16.4.9 清理

在使用完范围之后，最好调用 detach()方法把范围从创建它的文档中剥离。调用 detach()之后，就可以放心解除对范围的引用，以便垃圾回收程序释放它所占用的内存。下面是一个例子：

```
range.detach();  // 从文档中剥离范围
range = null;    // 解除引用
```

这两步是最合理的结束使用范围的方式。剥离之后的范围就不能再使用了。

16.5 小结

DOM2 规范定义了一些模块，用来丰富 DOM1 的功能。DOM2 Core 在一些类型上增加了与 XML 命名空间有关的新方法。这些变化只有在使用 XML 或 XHTML 文档时才会用到，在 HTML 文档中则没有用处。DOM2 增加的与 XML 命名空间无关的方法涉及以编程方式创建 Document 和 DocumentType 类型的新实例。

DOM2 Style 模块定义了如何操作元素的样式信息。

- 每个元素都有一个关联的 style 对象，可用于确定和修改元素特定的样式。
- 要确定元素的计算样式，包括应用到元素身上的所有 CSS 规则，可以使用 getComputedStyle()方法。
- 通过 document.styleSheets 集合可以访问文档上所有的样式表。

DOM2 Traversal and Range 模块定义了与 DOM 结构交互的不同方式。

- NodeIterator 和 TreeWalker 可以对 DOM 树执行深度优先的遍历。
- NodeIterator 接口很简单，每次只能向前和向后移动一步。TreeWalker 除了支持同样的行为，还支持在 DOM 结构的所有方向移动，包括父节点、同胞节点和子节点。
- 范围是选择 DOM 结构中特定部分并进行操作的一种方式。
- 通过范围的选区可以在保持文档结构完好的同时从文档中移除内容，也可复制文档中相应的部分。

第17章 事　件

本章内容

- ❑ 理解事件流
- ❑ 使用事件处理程序
- ❑ 了解不同类型的事件

JavaScript 与 HTML 的交互是通过**事件**实现的，事件代表文档或浏览器窗口中某个有意义的时刻。可以使用仅在事件发生时执行的**监听器**（也叫处理程序）订阅事件。在传统软件工程领域，这个模型叫“观察者模式”，其能够做到页面行为（在 JavaScript 中定义）与页面展示（在 HTML 和 CSS 中定义）的分离。

事件最早是在 IE3 和 Netscape Navigator 2 中出现的，当时的用意是把某些表单处理工作从服务器转移到浏览器上来。到了 IE4 和 Netscape Navigator 3 发布的时候，这两家浏览器都提供了类似但又不同的 API，而且持续了好几代。DOM2 开始尝试以符合逻辑的方式来标准化 DOM 事件 API。目前所有现代浏览器都实现了 DOM2 Events 的核心部分。IE8 是最后一个使用专有事件系统的主流浏览器。

浏览器的事件系统非常复杂。即使所有主流浏览器都实现了 DOM2 Events，规范也没有涵盖所有的事件类型。BOM 也支持事件，这些事件与 DOM 事件之间的关系由于长期以来缺乏文档，经常容易被混淆（HTML5 已经致力于明确这些关系）。而 DOM3 新增的事件 API 又让这些问题进一步复杂化了。根据具体的需求不同，使用事件可能会相对简单，也可能会非常复杂。但无论如何，理解其中的核心概念还是最重要的。

17.1　事件流

在第四代 Web 浏览器（IE4 和 Netscape Communicator 4）开始开发时，开发团队碰到了一个有意思的问题：页面哪个部分拥有特定的事件呢？要理解这个问题，可以在一张纸上画几个同心圆。把手指放到圆心上，则手指不仅是在一个圆圈里，而且是在所有的圆圈里。两家浏览器的开发团队都是以同样的方式看待浏览器事件的。当你点击一个按钮时，实际上不光点击了这个按钮，还点击了它的容器以及整个页面。

事件流描述了页面接收事件的顺序。结果非常有意思，IE 和 Netscape 开发团队提出了几乎完全相反的事件流方案。IE 将支持事件冒泡流，而 Netscape Communicator 将支持事件捕获流。

17.1.1　事件冒泡

IE 事件流被称为**事件冒泡**，这是因为事件被定义为从最具体的元素（文档树中最深的节点）开始触发，然后向上传播至没有那么具体的元素（文档）。比如有如下 HTML 页面：

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


在点击页面中的 元素后，click 事件会以如下顺序发生：

(1)

(2) <body>

(3) <html>

(4) document

也就是说， 元素，即被点击的元素，最先触发 click 事件。然后，click 事件沿 DOM 树一路向上，在经过的每个节点上依次触发，直至到达 document 对象。图 17-1 形象地展示了这个过程。

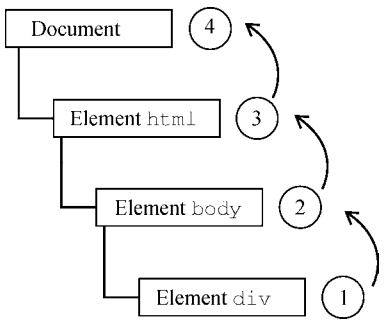

图 17-1

所有现代浏览器都支持事件冒泡，只是在实现方式上会有一些变化。IE5.5 及早期版本会跳过<html>元素（从<body>直接到 document）。现代浏览器中的事件会一直冒泡到 window 对象。

17.1.2 事件捕获

Netscape Communicator 团队提出了另一种名为**事件捕获**的事件流。事件捕获的意思是最不具体的节点应该最先收到事件，而最具体的节点应该最后收到事件。事件捕获实际上是为了在事件到达最终目标前拦截事件。如果前面的例子使用事件捕获，则点击 元素会以下列顺序触发 click 事件：

(1) document

(2) <html>

(3) <body>

(4)

在事件捕获中，click 事件首先由 document 元素捕获，然后沿 DOM 树依次向下传播，直至到达实际的目标元素 。这个过程如图 17-2 所示。

虽然这是 Netscape Communicator 唯一的事件流模型，但事件捕获得到了所有现代浏览器的支持。实际上，所有浏览器都是从 window 对象开始捕获事件，而 DOM2 Events 规范规定的是从 document 开始。

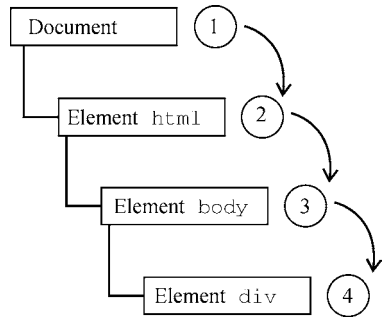

图　17-2

由于旧版本浏览器不支持，因此实际当中几乎不会使用事件捕获。通常建议使用事件冒泡，特殊情况下可以使用事件捕获。

17.1.3　DOM 事件流

DOM2 Events 规范规定事件流分为 3 个阶段：事件捕获、到达目标和事件冒泡。事件捕获最先发生，为提前拦截事件提供了可能。然后，实际的目标元素接收到事件。最后一个阶段是冒泡，最迟要在这个阶段响应事件。仍以前面那个简单的 HTML 为例，点击 元素会以如图 17-3 所示的顺序触发事件。

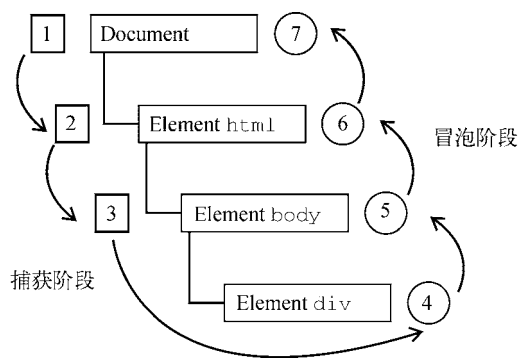

图　17-3

在 DOM 事件流中，实际的目标（ 元素）在捕获阶段不会接收到事件。这是因为捕获阶段从 document 到<html>再到<body>就结束了。下一阶段，即会在 元素上触发事件的“到达目标”阶段，通常在事件处理时被认为是冒泡阶段的一部分（稍后讨论）。然后，冒泡阶段开始，事件反向传播至文档。

大多数支持 DOM 事件流的浏览器实现了一个小小的拓展。虽然 DOM2 Events 规范明确捕获阶段不命中事件目标，但现代浏览器都会在捕获阶段在事件目标上触发事件。最终结果是在事件目标上有两个机会来处理事件。

> **注意**　所有现代浏览器都支持 DOM 事件流，只有 IE8 及更早版本不支持。

17.2 事件处理程序

事件意味着用户或浏览器执行的某种动作。比如，单击（click）、加载（load）、鼠标悬停（mouseover）。为响应事件而调用的函数被称为**事件处理程序**（或**事件监听器**）。事件处理程序的名字以"on"开头，因此 click 事件的处理程序叫作 onclick，而 load 事件的处理程序叫作 onload。有很多方式可以指定事件处理程序。

17.2.1 HTML 事件处理程序

特定元素支持的每个事件都可以使用事件处理程序的名字以 HTML 属性的形式来指定。此时属性的值必须是能够执行的 JavaScript 代码。例如，要在按钮被点击时执行某些 JavaScript 代码，可以使用以下 HTML 属性：

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

点击这个按钮后，控制台会输出一条消息。这种交互能力是通过为 onclick 属性指定 JavaScript 代码值来实现的。注意，因为属性的值是 JavaScript 代码，所以不能在未经转义的情况下使用 HTML 语法字符，比如和号（&）、双引号（"）、小于号（<）和大于号（>）。此时，为了避免使用 HTML 实体，可以使用单引号代替双引号。如果确实需要使用双引号，则要把代码改成下面这样：

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

在 HTML 中定义的事件处理程序可以包含精确的动作指令，也可以调用在页面其他地方定义的脚本，比如：

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

在这个例子中，单击按钮会调用 showMessage()函数。showMessage()函数是在单独的<script>元素中定义的，而且也可以在外部文件中定义。作为事件处理程序执行的代码可以访问全局作用域中的一切。

以这种方式指定的事件处理程序有一些特殊的地方。首先，会创建一个函数来封装属性的值。这个函数有一个特殊的局部变量 event，其中保存的就是 event 对象（本章后面会讨论）：

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

有了这个对象，就不用开发者另外定义其他变量，也不用从包装函数的参数列表中去取了。

在这个函数中，this 值相当于事件的目标元素，如下面的例子所示：

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

这个动态创建的包装函数还有一个特别有意思的地方，就是其作用域链被扩展了。在这个函数中，document 和元素自身的成员都可以被当成局部变量来访问。这是通过使用 with 实现的：

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

这意味着事件处理程序可以更方便地访问自己的属性。下面的代码与前面的示例功能一样：

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

如果这个元素是一个表单输入框，则作用域链中还会包含表单元素，事件处理程序对应的函数等价于如下这样：

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

本质上，经过这样的扩展，事件处理程序的代码就可以不必引用表单元素，而直接访问同一表单中的其他成员了。下面的例子就展示了这种成员访问模式：

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

点击这个例子中的按钮会显示出文本框中包含的文本。注意，事件处理程序中的代码直接引用了 username。

在 HTML 中指定事件处理程序有一些问题。第一个问题是时机问题。有可能 HTML 元素已经显示在页面上，用户都与其交互了，而事件处理程序的代码还无法执行。比如在前面的例子中，如果 showMessage()函数是在页面后面，在按钮中代码的后面定义的，那么当用户在 showMessage()函数被定义之前点击按钮时，就会发生错误。为此，大多数 HTML 事件处理程序会封装在 try/catch 块中，以便在这种情况下静默失败，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()函数被定义之前点击了按钮，就不会发生 JavaScript 错误了，这是因为错误在浏览器收到之前已经被拦截了。

另一个问题是对事件处理程序作用域链的扩展在不同浏览器中可能导致不同的结果。不同 JavaScript 引擎中标识符解析的规则存在差异，因此访问无限定的对象成员可能导致错误。

使用 HTML 指定事件处理程序的最后一个问题是 HTML 与 JavaScript 强耦合。如果需要修改事件处理程序，则必须在两个地方，即 HTML 和 JavaScript 中，修改代码。这也是很多开发者不使用 HTML 事件处理程序，而使用 JavaScript 指定事件处理程序的主要原因。

17.2.2 DOM0 事件处理程序

在 JavaScript 中指定事件处理程序的传统方式是把一个函数赋值给（DOM 元素的）一个事件处理程序属性。这也是在第四代 Web 浏览器中开始支持的事件处理程序赋值方法，直到现在所有现代浏览器仍然都支持此方法，主要原因是简单。要使用 JavaScript 指定事件处理程序，必须先取得要操作对象的引用。

每个元素（包括 window 和 document）都有通常小写的事件处理程序属性，比如 onclick。只要把这个属性赋值为一个函数即可：

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

这里先从文档中取得按钮，然后给它的 onclick 事件处理程序赋值一个函数。注意，前面的代码在运行之后才会给事件处理程序赋值。因此如果在页面中上面的代码出现在按钮之后，则有可能出现用户点击按钮没有反应的情况。

像这样使用 DOM0 方式为事件处理程序赋值时，所赋函数被视为元素的方法。因此，事件处理程序会在元素的作用域中运行，即 this 等于元素。下面的例子演示了使用 this 引用元素本身：

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

点击按钮，这段代码会显示元素的 ID。这个 ID 是通过 this.id 获取的。不仅仅是 id，在事件处理程序里通过 this 可以访问元素的任何属性和方法。以这种方式添加事件处理程序是注册在事件流的冒泡阶段的。

通过将事件处理程序属性的值设置为 null，可以移除通过 DOM0 方式添加的事件处理程序，如下面的例子所示：

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

把事件处理程序设置为 null，再点击按钮就不会执行任何操作了。

> **注意** 如果事件处理程序是在 HTML 中指定的，则 onclick 属性的值是一个包装相应 HTML 事件处理程序属性值的函数。这些事件处理程序也可以通过在 JavaScript 中将相应属性设置为 null 来移除。

17.2.3 DOM2 事件处理程序

DOM2 Events 为事件处理程序的赋值和移除定义了两个方法：addEventListener()和 removeEventListener()。这两个方法暴露在所有 DOM 节点上，它们接收 3 个参数：事件名、事件处理函数和一个布尔值，true 表示在捕获阶段调用事件处理程序，false（默认值）表示在冒泡阶段调用事件处理程序。

仍以给按钮添加 click 事件处理程序为例，可以这样写：