```
// 取得类名为"selected"的第一个元素
let selected = document.querySelector(".selected");

// 取得类名为"button"的图片
let img = document.body.querySelector("img.button");
```

在 Document 上使用 querySelector()方法时，会从文档元素开始搜索；在 Element 上使用 querySelector()方法时，则只会从当前元素的后代中查询。

用于查询模式的 CSS 选择符可繁可简，依需求而定。如果选择符有语法错误或碰到不支持的选择符，则 querySelector()方法会抛出错误。

15.1.2 querySelectorAll()

querySelectorAll()方法跟 querySelector()一样，也接收一个用于查询的参数，但它会返回所有匹配的节点，而不止一个。这个方法返回的是一个 NodeList 的静态实例。

再强调一次，querySelectorAll()返回的 NodeList 实例一个属性和方法都不缺，但它是一个静态的“快照”，而非“实时”的查询。这样的底层实现避免了使用 NodeList 对象可能造成的性能问题。

以有效 CSS 选择符调用 querySelectorAll()都会返回 NodeList，无论匹配多少个元素都可以。如果没有匹配项，则返回空的 NodeList 实例。

与 querySelector()一样，querySelectorAll()也可以在 Document、DocumentFragment 和 Element 类型上使用。下面是几个例子：

```
// 取得 ID 为"myDiv"的<div>元素中的所有<em>元素
let ems = document.getElementById("myDiv").querySelectorAll("em");

// 取得所有类名中包含"selected"的元素
let selecteds = document.querySelectorAll(".selected");

// 取得所有是<p>元素子元素的<strong>元素
let strongs = document.querySelectorAll("p strong");
```

返回的 NodeList 对象可以通过 for-of 循环、item()方法或中括号语法取得个别元素。比如：

```
let strongElements = document.querySelectorAll("p strong");

// 以下 3 个循环的效果一样

for (let strong of strongElements) {
  strong.className = "important";
}

for (let i = 0; i < strongElements.length; ++i) {
  strongElements.item(i).className = "important";
}

for (let i = 0; i < strongElements.length; ++i) {
  strongElements[i].className = "important";
}
```

与 querySelector()方法一样，如果选择符有语法错误或碰到不支持的选择符，则 querySelectorAll()方法会抛出错误。

15.1.3 matches()

matches()方法（在规范草案中称为matchesSelector()）接收一个CSS选择符参数，如果元素匹配则该选择符返回true，否则返回false。例如：

```
if (document.body.matches("body.page1")){
  // true
}
```

使用这个方法可以方便地检测某个元素会不会被querySelector()或querySelectorAll()方法返回。

所有主流浏览器都支持matches()。Edge、Chrome、Firefox、Safari和Opera完全支持，IE9~11及一些移动浏览器支持带前缀的方法。

15.2 元素遍历

IE9之前的版本不会把元素间的空格当成空白节点，而其他浏览器则会。这样就导致了childNodes和firstChild等属性上的差异。为了弥补这个差异，同时不影响DOM规范，W3C通过新的Element Traversal规范定义了一组新属性。

Element Traversal API为DOM元素添加了5个属性：

- childElementCount，返回子元素数量（不包含文本节点和注释）；
- firstElementChild，指向第一个Element类型的子元素（Element版firstChild）；
- lastElementChild，指向最后一个Element类型的子元素（Element版lastChild）；
- previousElementSibling，指向前一个Element类型的同胞元素（Element版previousSibling）；
- nextElementSibling，指向后一个Element类型的同胞元素（Element版nextSibling）。

在支持的浏览器中，所有DOM元素都会有这些属性，为遍历DOM元素提供便利。这样开发者就不用担心空白文本节点的问题了。

举个例子，过去要以跨浏览器方式遍历特定元素的所有子元素，代码大致是这样写的：

```
let parentElement = document.getElementById('parent');
let currentChildNode = parentElement.firstChild;

// 没有子元素，firstChild返回null，跳过循环
while (currentChildNode) {
  if (currentChildNode.nodeType === 1) {
    // 如果有元素节点，则做相应处理
    processChild(currentChildNode);
  }
  if (currentChildNode === parentElement.lastChild) {
    break;
  }
  currentChildNode = currentChildNode.nextSibling;
}
```

使用Element Traversal属性之后，以上代码可以简化如下：

```
let parentElement = document.getElementById('parent');
let currentChildElement = parentElement.firstElementChild;
```

```
// 没有子元素，firstElementChild 返回 null，跳过循环
while (currentChildElement) {
  // 这就是元素节点，做相应处理
  processChild(currentChildElement);
  if (currentChildElement === parentElement.lastElementChild) {
    break;
  }
  currentChildElement = currentChildElement.nextElementSibling;
}
```

IE9 及以上版本，以及所有现代浏览器都支持 Element Traversal 属性。

15.3　HTML5

HTML5 代表着与以前的 HTML 截然不同的方向。在所有以前的 HTML 规范中，从未出现过描述 JavaScript 接口的情形，HTML 就是一个纯标记语言。JavaScript 绑定的事，一概交给 DOM 规范去定义。

然而，HTML5 规范却包含了与标记相关的大量 JavaScript API 定义。其中有的 API 与 DOM 重合，定义了浏览器应该提供的 DOM 扩展。

> **注意**　因为 HTML5 覆盖的范围极其广泛，所以本节主要讨论其影响所有 DOM 节点的部分。HTML5 的其他部分将在本书后面的相关章节中再讨论。

15.3.1　CSS 类扩展

自 HTML4 被广泛采用以来，Web 开发中一个主要的变化是 `class` 属性用得越来越多，其用处是为元素添加样式以及语义信息。自然地，JavaScript 与 CSS 类的交互就增多了，包括动态修改类名，以及根据给定的一个或一组类名查询元素，等等。为了适应开发者和他们对 `class` 属性的认可，HTML5 增加了一些特性以方便使用 CSS 类。

1. getElementsByClassName()

`getElementsByClassName()`是 HTML5 新增的最受欢迎的一个方法，暴露在 `document` 对象和所有 HTML 元素上。这个方法脱胎于基于原有 DOM 特性实现该功能的 JavaScript 库，提供了性能高好的原生实现。

`getElementsByClassName()`方法接收一个参数，即包含一个或多个类名的字符串，返回类名中包含相应类的元素的 `NodeList`。如果提供了多个类名，则顺序无关紧要。下面是几个示例：

```
// 取得所有类名中包含"username"和"current"元素
// 这两个类名的顺序无关紧要
let allCurrentUsernames = document.getElementsByClassName("username current");
// 取得 ID 为"myDiv"的元素子树中所有包含"selected"类的元素
let selected = document.getElementById("myDiv").getElementsByClassName("selected");
```

这个方法只会返回以调用它的对象为根元素的子树中所有匹配的元素。在 `document` 上调用 `getElementsByClassName()`返回文档中所有匹配的元素，而在特定元素上调用 `getElementsByClassName()`则返回该元素后代中匹配的元素。

如果要给包含特定类（而不是特定 ID 或标签）的元素添加事件处理程序，使用这个方法会很方便。不过要记住，因为返回值是 `NodeList`，所以使用这个方法会遇到跟使用 `getElementsByTagName()`

和其他返回 NodeList 对象的 DOM 方法同样的问题。

IE9 及以上版本，以及所有现代浏览器都支持 getElementsByClassName()方法。

2. classList 属性

要操作类名，可以通过 className 属性实现添加、删除和替换。但 className 是一个字符串，所以每次操作之后都需要重新设置这个值才能生效，即使只改动了部分字符串也一样。以下面的 HTML 代码为例：

```
<div class="bd user disabled">...</div>
```

这个 元素有 3 个类名。要想删除其中一个，就得先把 className 拆开，删除不想要的那个，再把包含剩余类的字符串设置回去。比如：

```
// 要删除"user"类
let targetClass = "user";

// 把类名拆成数组
let classNames = div.className.split(/\s+/);

// 找到要删除类名的索引
let idx = classNames.indexOf(targetClass);

// 如果有则删除
if (idx > -1) {
  classNames.splice(i,1);
}

// 重新设置类名
div.className = classNames.join(" ");
```

这就是从 元素的类名中删除"user"类要写的代码。替换类名和检测类名也要涉及同样的算法。添加类名只涉及字符串拼接，但必须先检查一下以确保不会重复添加相同的类名。很多 JavaScript 库为这些操作实现了便利方法。

HTML5 通过给所有元素增加 classList 属性为这些操作提供了更简单也更安全的实现方式。classList 是一个新的集合类型 DOMTokenList 的实例。与其他 DOM 集合类型一样，DOMTokenList 也有 length 属性表示自己包含多少项，也可以通过 item()或中括号取得个别的元素。此外，DOMTokenList 还增加了以下方法。

- add(*value*)，向类名列表中添加指定的字符串值 value。如果这个值已经存在，则什么也不做。
- contains(*value*)，返回布尔值，表示给定的 value 是否存在。
- remove(*value*)，从类名列表中删除指定的字符串值 value。
- toggle(*value*)，如果类名列表中已经存在指定的 value，则删除；如果不存在，则添加。

这样以来，前面的例子中那么多行代码就可以简化成下面的一行：

```
div.classList.remove("user");
```

这行代码可以在不影响其他类名的情况下完成删除。其他方法同样极大地简化了操作类名的复杂性，如下面的例子所示：

```
// 删除"disabled"类
div.classList.remove("disabled");

// 添加"current"类
div.classList.add("current");
```

```
// 切换"user"类
div.classList.toggle("user");

// 检测类名
if (div.classList.contains("bd") && !div.classList.contains("disabled")){
  // 执行操作
)

// 迭代类名
for (let class of div.classList){
  doStuff(class);
}
```

添加了 classList 属性之后，除非是完全删除或完全重写元素的 class 属性，否则 className 属性就用不到了。IE10 及以上版本（部分）和其他主流浏览器（完全）实现了 classList 属性。

15.3.2　焦点管理

HTML5 增加了辅助 DOM 焦点管理的功能。首先是 document.activeElement，始终包含当前拥有焦点的 DOM 元素。页面加载时，可以通过用户输入（按 Tab 键或代码中使用 focus()方法）让某个元素自动获得焦点。例如：

```
let button = document.getElementById("myButton");
button.focus();
console.log(document.activeElement === button); // true
```

默认情况下，document.activeElement 在页面刚加载完之后会设置为 document.body。而在页面完全加载之前，document.activeElement 的值为 null。

其次是 document.hasFocus()方法，该方法返回布尔值，表示文档是否拥有焦点：

```
let button = document.getElementById("myButton");
button.focus();
console.log(document.hasFocus()); // true
```

确定文档是否获得了焦点，就可以帮助确定用户是否在操作页面。

第一个方法可以用来查询文档，确定哪个元素拥有焦点，第二个方法可以查询文档是否获得了焦点，而这对于保证 Web 应用程序的无障碍使用是非常重要的。无障碍 Web 应用程序的一个重要方面就是焦点管理，而能够确定哪个元素当前拥有焦点（相比于之前的猜测）是一个很大的进步。

15.3.3　HTMLDocument 扩展

HTML5 扩展了 HTMLDocument 类型，增加了更多功能。与其他 HTML5 定义的 DOM 扩展一样，这些变化同样基于所有浏览器事实上都已经支持的专有扩展。为此，即使这些扩展的标准化相对较晚，很多浏览器也早就实现了相应的功能。

1. readyState 属性

readyState 是 IE4 最早添加到 document 对象上的属性，后来其他浏览器也都依葫芦画瓢地支持这个属性。最终，HTML5 将这个属性写进了标准。document.readyState 属性有两个可能的值：

- loading，表示文档正在加载；
- complete，表示文档加载完成。

实际开发中，最好是把 document.readState 当成一个指示器，以判断文档是否加载完毕。在这个属性得到广泛支持以前，通常要依赖 onload 事件处理程序设置一个标记，表示文档加载完了。这个属性的基本用法如下：

```
if (document.readyState == "complete"){
  // 执行操作
}
```

2. compatMode 属性

自从 IE6 提供了以标准或混杂模式渲染页面的能力之后，检测页面渲染模式成为一个必要的需求。IE 为 document 添加了 compatMode 属性，这个属性唯一的任务是指示浏览器当前处于什么渲染模式。如下面的例子所示，标准模式下 document.compatMode 的值是"CSS1Compat"，而在混杂模式下，document.compatMode 的值是"BackCompat"：

```
if (document.compatMode == "CSS1Compat"){
  console.log("Standards mode");
} else {
  console.log("Quirks mode");
}
```

HTML5 最终也把 compatMode 属性的实现标准化了。

3. head 属性

作为对 document.body（指向文档的<body>元素）的补充，HTML5 增加了 document.head 属性，指向文档的<head>元素。可以像下面这样直接取得<head>元素：

```
let head = document.head;
```

15.3.4 字符集属性

HTML5 增加了几个与文档字符集有关的新属性。其中，characterSet 属性表示文档实际使用的字符集，也可以用来指定新字符集。这个属性的默认值是"UTF-16"，但可以通过<meta>元素或响应头，以及新增的 characterSeet 属性来修改。下面是一个例子：

```
console.log(document.characterSet); // "UTF-16"
document.characterSet = "UTF-8";
```

15.3.5 自定义数据属性

HTML5 允许给元素指定非标准的属性，但要使用前缀 data-以便告诉浏览器，这些属性既不包含与渲染有关的信息，也不包含元素的语义信息。除了前缀，自定义属性对命名是没有限制的，data-后面跟什么都可以。下面是一个例子：

```
<div id="myDiv" data-appId="12345" data-myname="Nicholas"></div>
```

定义了自定义数据属性后，可以通过元素的 dataset 属性来访问。dataset 属性是一个 DOMStringMap 的实例，包含一组键/值对映射。元素的每个 data-name 属性在 dataset 中都可以通过 data-后面的字符串作为键来访问（例如，属性 data-myname、data-myName 可以通过 myname 访问，但要注意 data-my-name、data-My-Name 要通过 myName 来访问）。下面是一个使用自定义数据属性的例子：

```
// 本例中使用的方法仅用于示范

let div = document.getElementById("myDiv");

// 取得自定义数据属性的值
let appId = div.dataset.appId;
let myName = div.dataset.myname;

// 设置自定义数据属性的值
div.dataset.appId = 23456;
div.dataset.myname = "Michael";

// 有"myname"吗?
if (div.dataset.myname){
  console.log(`Hello, ${div.dataset.myname}`);
}
```

自定义数据属性非常适合需要给元素附加某些数据的场景，比如链接追踪和在聚合应用程序中标识页面的不同部分。另外，单页应用程序框架也非常多地使用了自定义数据属性。

15.3.6　插入标记

DOM 虽然已经为操纵节点提供了很多 API，但向文档中一次性插入大量 HTML 时还是比较麻烦。相比先创建一堆节点，再把它们以正确的顺序连接起来，直接插入一个 HTML 字符串要简单（快速）得多。HTML5 已经通过以下 DOM 扩展将这种能力标准化了。

1. innerHTML 属性

在读取 innerHTML 属性时，会返回元素所有后代的 HTML 字符串，包括元素、注释和文本节点。而在写入 innerHTML 时，则会根据提供的字符串值以新的 DOM 子树替代元素中原来包含的所有节点。比如下面的 HTML 代码：

```
<div id="content">
  <p>This is a <strong>paragraph</strong> with a list following it.</p>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</div>
```

对于这里的 元素而言，其 innerHTML 属性会返回以下字符串：

```
<p>This is a <strong>paragraph</strong> with a list following it.</p>
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

实际返回的文本内容会因浏览器而不同。IE 和 Opera 会把所有元素标签转换为大写，而 Safari、Chrome 和 Firefox 则会按照文档源代码的格式返回，包含空格和缩进。因此不要指望不同浏览器的 innerHTML 会返回完全一样的值。

在写入模式下，赋给 innerHTML 属性的值会被解析为 DOM 子树，并替代元素之前的所有节点。因为所赋的值默认为 HTML，所以其中的所有标签都会以浏览器处理 HTML 的方式转换为元素（同样，

转换结果也会因浏览器不同而不同）。如果赋值中不包含任何 HTML 标签，则直接生成一个文本节点，如下所示：

```
div.innerHTML = "Hello world!";
```

因为浏览器会解析设置的值，所以给 innerHTML 设置包含 HTML 的字符串时，结果会大不一样。来看下面的例子：

```
div.innerHTML = "Hello & welcome, <b>\"reader\"!</b>";
```

这个操作的结果相当于：

```
<div id="content">Hello &amp; welcome, <b>&quot;reader&quot;!</b></div>
```

设置完 innerHTML，马上就可以像访问其他节点一样访问这些新节点。

> **注意** 设置 innerHTML 会导致浏览器将 HTML 字符串解析为相应的 DOM 树。这意味着设置 innerHTML 属性后马上再读出来会得到不同的字符串。这是因为返回的字符串是将原始字符串对应的 DOM 子树序列化之后的结果。

2. 旧 IE 中的 innerHTML

在所有现代浏览器中，通过 innerHTML 插入的<script>标签是不会执行的。而在 IE8 及之前的版本中，只要这样插入的<script>元素指定了 defer 属性，且<script>之前是“受控元素”（scoped element），那就是可以执行的。<script>元素与<style>或注释一样，都是“非受控元素”（NoScope element），也就是在页面上看不到它们。IE 会把 innerHTML 中从非受控元素开始的内容都删掉，也就是说下面的例子是行不通的：

```
// 行不通
div.innerHTML = "<script defer>console.log('hi');<\/script>";
```

在这个例子中，innerHTML 字符串以一个非受控元素开始，因此整个字符串都会被清空。为了达到目的，必须在<script>前面加上一个受控元素，例如文本节点或没有结束标签的元素（如<input>）。因此，下面的代码就是可行的：

```
// 以下都可行
div.innerHTML = "_<script defer>console.log('hi');<\/script>";
div.innerHTML = "<div> </div><script defer>console.log('hi');<\/script>";
div.innerHTML = "<input type=\"hidden\"><script defer>console.
log('hi');<\/script>";
```

第一行会在<script>元素前面插入一个文本节点。为了不影响页面排版，可能稍后需要删掉这个文本节点。第二行与之类似，使用了包含空格的 元素。空 是不行的，必须包含一点内容，以强制创建一个文本节点。同样，这个 元素可能也需要事后删除，以免影响页面外观。第三行使用了一个隐藏的<input>字段来达成同样的目的。因为这个字段不影响页面布局，所以应该是最理想的方案。

在 IE 中，通过 innerHTML 插入<style>也会有类似的问题。多数浏览器支持使用 innerHTML 插入<style>元素：

```
div.innerHTML = "<style type=\"text/css\">body {background-color: red; }</style>";
```

但在 IE8 及之前的版本中，<style>也被认为是非受控元素，所以必须前置一个受控元素：

```
div.innerHTML = "_<style type=\"text/css\">body {background-color: red; }</style>";
div.removeChild(div.firstChild);
```

> **注意**　Firefox 在内容类型为 application/xhtml+xml 的 XHTML 文档中对 innerHTML 更加严格。在 XHTML 文档中使用 innerHTML，必须使用格式良好的 XHTML 代码。否则，在 Firefox 中会静默失败。

3. outerHTML 属性

读取 outerHTML 属性时，会返回调用它的元素（及所有后代元素）的 HTML 字符串。在写入 outerHTML 属性时，调用它的元素会被传入的 HTML 字符串经解释之后生成的 DOM 子树取代。比如下面的 HTML 代码：

```
<div id="content">
  <p>This is a <strong>paragraph</strong> with a list following it.</p>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</div>
```

在这个 元素上调用 outerHTML 会返回相同的字符串，包括 本身。注意，浏览器因解析和解释 HTML 代码的机制不同，返回的字符串也可能不同。（跟 innerHTML 的情况是一样的。）

如果使用 outerHTML 设置 HTML，比如：

```
div.outerHTML = "<p>This is a paragraph.</p>";
```

则会得到与执行以下脚本相同的结果：

```
let p = document.createElement("p");
p.appendChild(document.createTextNode("This is a paragraph."));
div.parentNode.replaceChild(p, div);
```

新的 元素会取代 DOM 树中原来的 元素。

4. insertAdjacentHTML()与 insertAdjacentText()

关于插入标签的最后两个新增方法是 insertAdjacentHTML()和 insertAdjacentText()。这两个方法最早源自 IE，它们都接收两个参数：要插入标记的位置和要插入的 HTML 或文本。第一个参数必须是下列值中的一个：

- "beforebegin"，插入当前元素前面，作为前一个同胞节点；
- "afterbegin"，插入当前元素内部，作为新的子节点或放在第一个子节点前面；
- "beforeend"，插入当前元素内部，作为新的子节点或放在最后一个子节点后面；
- "afterend"，插入当前元素后面，作为下一个同胞节点。

注意这几个值是不区分大小写的。第二个参数会作为 HTML 字符串解析（与 innerHTML 和 outerHTML 相同）或者作为纯文本解析（与 innerText 和 outerText 相同）。如果是 HTML，则会在解析出错时抛出错误。下面展示了基本用法[①]：

---

① 假设当前元素是 Hello world! ，则"beforebegin"和"afterbegin"中的"begin"指开始标签 ；而"afterend"和"beforeend"中的"end"指结束标签 。——译者注

```
// 作为前一个同胞节点插入
element.insertAdjacentHTML("beforebegin", "<p>Hello world!</p>");
element.insertAdjacentText("beforebegin", "Hello world!");

// 作为第一个子节点插入
element.insertAdjacentHTML("afterbegin", "<p>Hello world!</p>");
element.insertAdjacentText("afterbegin", "Hello world!");

// 作为最后一个子节点插入
element.insertAdjacentHTML("beforeend", "<p>Hello world!</p>");
element.insertAdjacentText("beforeend", "Hello world!");

// 作为下一个同胞节点插入
element.insertAdjacentHTML("afterend", "<p>Hello world!</p>"); element.
insertAdjacentText("afterend", "Hello world!");
```

5. 内存与性能问题

使用本节介绍的方法替换子节点可能在浏览器（特别是 IE）中导致内存问题。比如，如果被移除的子树元素中之前有关联的事件处理程序或其他 JavaScript 对象（作为元素的属性），那它们之间的绑定关系会滞留在内存中。如果这种替换操作频繁发生，页面的内存占用就会持续攀升。在使用 innerHTML、outerHTML 和 insertAdjacentHTML()之前，最好手动删除要被替换的元素上关联的事件处理程序和 JavaScript 对象。

使用这些属性当然有其方便之处，特别是 innerHTML。一般来讲，插入大量的新 HTML 使用 innerHTML 比使用多次 DOM 操作创建节点再插入来得更便捷。这是因为 HTML 解析器会解析设置给 innerHTML（或 outerHTML）的值。解析器在浏览器中是底层代码（通常是 C++代码），比 JavaScript 快得多。不过，HTML 解析器的构建与解构也不是没有代价，因此最好限制使用 innerHTML 和 outerHTML 的次数。比如，下面的代码使用 innerHTML 创建了一些列表项：

```
for (let value of values){
  ul.innerHTML += '<li>${value}</li>';  // 别这样做!
}
```

这段代码效率低，因为每次迭代都要设置一次 innerHTML。不仅如此，每次循环还要先读取 innerHTML，也就是说循环一次要访问两次 innerHTML。为此，最好通过循环先构建一个独立的字符串，最后再一次性把生成的字符串赋值给 innerHTML，比如：

```
let itemsHtml = "";
for (let value of values){
  itemsHtml += '<li>${value}</li>';
}
ul.innerHTML = itemsHtml;
```

这样修改之后效率就高多了，因为只有对 innerHTML 的一次赋值。当然，像下面这样一行代码也可以搞定：

```
ul.innerHTML = values.map(value => '<li>${value}</li>').join('');
```

6. 跨站点脚本

尽管 innerHTML 不会执行自己创建的<script>标签，但仍然向恶意用户暴露了很大的攻击面，因为通过它可以毫不费力地创建元素并执行 onclick 之类的属性。

如果页面中要使用用户提供的信息，则不建议使用 innerHTML。与使用 innerHTML 获得的方便相比，防止 XSS 攻击更让人头疼。此时一定要隔离要插入的数据，在插入页面前必须毫不犹豫地使用相