了 name 属性，因此就不用再设置了。随后，value 属性被赋值为"left"。为把这个新属性添加到元素上，可以使用元素的 setAttributeNode()方法。添加这个属性后，可以通过不同方式访问它，包括 attributes 属性、getAttributeNode()和 getAttribute()方法。其中，attributes 属性和 getAttributeNode()方法都返回属性对应的 Attr 节点，而 getAttribute()方法只返回属性的值。

> **注意**　将属性作为节点来访问多数情况下并无必要。推荐使用 getAttribute()、removeAttribute()和 setAttribute()方法操作属性，而不是直接操作属性节点。

14.2　DOM 编程

很多时候，操作 DOM 是很直观的。通过 HTML 代码能实现的，也一样能通过 JavaScript 实现。但有时候，DOM 也没有看起来那么简单。浏览器能力的参差不齐和各种问题，也会导致 DOM 的某些方面会复杂一些。

14.2.1　动态脚本

<script>元素用于向网页中插入 JavaScript 代码，可以是 src 属性包含的外部文件，也可以是作为该元素内容的源代码。动态脚本就是在页面初始加载时不存在，之后又通过 DOM 包含的脚本。与对应的 HTML 元素一样，有两种方式通过<script>动态为网页添加脚本：引入外部文件和直接插入源代码。

动态加载外部文件很容易实现，比如下面的<script>元素：

```
<script src="foo.js"></script>
```

可以像这样通过 DOM 编程创建这个节点：

```
let script = document.createElement("script");
script.src = "foo.js";
document.body.appendChild(script);
```

这里的 DOM 代码实际上完全照搬了它要表示的 HTML 代码。注意，在上面最后一行把<script>元素添加到页面之前，是不会开始下载外部文件的。当然也可以把它添加到<head>元素，同样可以实现动态脚本加载。这个过程可以抽象为一个函数，比如：

```
function loadScript(url) {
  let script = document.createElement("script");
  script.src = url;
  document.body.appendChild(script);
}
```

然后，就可以像下面这样加载外部 JavaScript 文件了：

```
loadScript("client.js");
```

加载之后，这个脚本就可以对页面执行操作了。这里有个问题：怎么能知道脚本什么时候加载完？这个问题并没有标准答案。第 17 章会讨论一些与加载相关的事件，具体情况取决于使用的浏览器。

另一个动态插入 JavaScript 的方式是嵌入源代码，如下面的例子所示：

```
<script>
  function sayHi() {
    alert("hi");
```

```
  }
</script>
```

使用 DOM，可以实现以下逻辑：

```
let script = document.createElement("script");
script.appendChild(document.createTextNode("function sayHi(){alert('hi');}"));
document.body.appendChild(script);
```

以上代码可以在 Firefox、Safari、Chrome 和 Opera 中运行。不过在旧版本的 IE 中可能会导致问题。这是因为 IE 对<script>元素做了特殊处理，不允许常规 DOM 访问其子节点。但<script>元素上有一个 text 属性，可以用来添加 JavaScript 代码，如下所示：

```
var script = document.createElement("script");
script.text = "function sayHi(){alert('hi');}";
document.body.appendChild(script);
```

这样修改后，上面的代码可以在 IE、Firefox、Opera 和 Safari 3 及更高版本中运行。Safari 3 之前的版本不能正确支持这个 text 属性，但这些版本却支持文本节点赋值。对于早期的 Safari 版本，需要使用以下代码：

```
var script = document.createElement("script");
var code = "function sayHi(){alert('hi');}";
try {
  script.appendChild(document.createTextNode("code"));
} catch (ex){
  script.text = "code";
}
document.body.appendChild(script);
```

这里先尝试使用标准的 DOM 文本节点插入方式，因为除 IE 之外的浏览器都支持这种方式。IE 此时会抛出错误，那么可以在捕获错误之后再使用 text 属性来插入 JavaScript 代码。于是，我们就可以抽象出一个跨浏览器的函数：

```
function loadScriptString(code){
  var script = document.createElement("script");
  script.type = "text/javascript";
  try {
    script.appendChild(document.createTextNode(code));
  } catch (ex){
    script.text = code;
  }
  document.body.appendChild(script);
}
```

这个函数可以这样调用：

```
loadScriptString("function sayHi(){alert('hi');}");
```

以这种方式加载的代码会在全局作用域中执行，并在调用返回后立即生效。基本上，这就相当于在全局作用域中把源代码传给 eval()方法。

注意，通过 innerHTML 属性创建的<script>元素永远不会执行。浏览器会尽责地创建<script>元素，以及其中的脚本文本，但解析器会给这个<script>元素打上永不执行的标签。只要是使用 innerHTML 创建的<script>元素，以后也没有办法强制其执行。

14.2.2　动态样式

CSS 样式在 HTML 页面中可以通过两个元素加载。<link>元素用于包含 CSS 外部文件，而<style>元素用于添加嵌入样式。与动态脚本类似，动态样式也是页面初始加载时并不存在，而是在之后才添加到页面中的。

来看下面这个典型的<link>元素：

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

这个元素很容易使用 DOM 编程创建出来：

```
let link = document.createElement("link");
link.rel = "stylesheet";
link.type = "text/css";
link.href = "styles.css";
let head = document.getElementsByTagName("head")[0];
head.appendChild(link);
```

以上代码在所有主流浏览器中都能正常运行。注意应该把<link>元素添加到<head>元素而不是<body>元素，这样才能保证所有浏览器都能正常运行。这个过程可以抽象为以下通用函数：

```
function loadStyles(url){
  let link = document.createElement("link");
  link.rel = "stylesheet";
  link.type = "text/css";
  link.href = url;
  let head = document.getElementsByTagName("head")[0];
  head.appendChild(link);
}
```

然后就可以这样调用这个 loadStyles()函数了：

```
loadStyles("styles.css");
```

通过外部文件加载样式是一个异步过程。因此，样式的加载和正执行的 JavaScript 代码并没有先后顺序。一般来说，也没有必要知道样式什么时候加载完成。

另一种定义样式的方式是使用<script>元素包含嵌入的 CSS 规则，例如：

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

逻辑上，下列 DOM 代码会有同样的效果：

```
let style = document.createElement("style");
style.type = "text/css";
style.appendChild(document.createTextNode("body{background-color:red}"));
let head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

以上代码在 Firefox、Safari、Chrome 和 Opera 中都可以运行，但 IE 除外。IE 对<style>节点会施加限制，不允许访问其子节点，这一点与它对<script>元素施加的限制一样。事实上，IE 在执行到给<style>添加子节点的代码时，会抛出与给<script>添加子节点时同样的错误。对于 IE，解决方案是访问元素的 styleSheet 属性，这个属性又有一个 cssText 属性，然后给这个属性添加 CSS 代码：

```
let style = document.createElement("style");
style.type = "text/css";
try{
  style.appendChild(document.createTextNode("body{background-color:red}"));
} catch (ex){
  style.styleSheet.cssText = "body{background-color:red}";
}
let head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

与动态添加脚本源代码类似，这里也使用了 try...catch 语句捕获 IE 抛出的错误，然后再以 IE 特有的方式来设置样式。这是最终的通用函数：

```
function loadStyleString(css){
  let style = document.createElement("style");
  style.type = "text/css";
  try{
    style.appendChild(document.createTextNode(css));
  } catch (ex){
    style.styleSheet.cssText = css;
  }
  let head = document.getElementsByTagName("head")[0];
    head.appendChild(style);
}
```

可以这样调用这个函数：

```
loadStyleString("body{background-color:red}");
```

这样添加的样式会立即生效，因此所有变化会立即反映出来。

> **注意** 对于 IE，要小心使用 styleSheet.cssText。如果重用同一个<style>元素并设置该属性超过一次，则可能导致浏览器崩溃。同样，将 cssText 设置为空字符串也可能导致浏览器崩溃。

14.2.3 操作表格

表格是 HTML 中最复杂的结构之一。通过 DOM 编程创建<table>元素，通常要涉及大量标签，包括表行、表元、表题，等等。因此，通过 DOM 编程创建和修改表格时可能要写很多代码。假设要通过 DOM 来创建以下 HTML 表格：

```
<table border="1" width="100%">
  <tbody>
    <tr>
      <td>Cell 1,1</td>
      <td>Cell 2,1</td>
    </tr>
    <tr>
      <td>Cell 1,2</td>
      <td>Cell 2,2</td>
    </tr>
  </tbody>
</table>
```

下面就是以 DOM 编程方式重建这个表格的代码：

```
// 创建表格
let table = document.createElement("table");
table.border = 1;
table.width = "100%";

// 创建表体
let tbody = document.createElement("tbody");
table.appendChild(tbody);

// 创建第一行
let row1 = document.createElement("tr");
tbody.appendChild(row1);
let cell1_1 = document.createElement("td");
cell1_1.appendChild(document.createTextNode("Cell 1,1"));
row1.appendChild(cell1_1);
let cell2_1 = document.createElement("td");
cell2_1.appendChild(document.createTextNode("Cell 2,1"));
row1.appendChild(cell2_1);

// 创建第二行
let row2 = document.createElement("tr");
tbody.appendChild(row2);
let cell1_2 = document.createElement("td");
cell1_2.appendChild(document.createTextNode("Cell 1,2"));
row2.appendChild(cell1_2);
let cell2_2= document.createElement("td");
cell2_2.appendChild(document.createTextNode("Cell 2,2"));
row2.appendChild(cell2_2);

// 把表格添加到文档主体
document.body.appendChild(table);
```

以上代码相当烦琐，也不好理解。为了方便创建表格，HTML DOM 给<table>、<tbody>和<tr>元素添加了一些属性和方法。

<table>元素添加了以下属性和方法：

- caption，指向<caption>元素的指针（如果存在）；
- tBodies，包含<tbody>元素的 HTMLCollection；
- tFoot，指向<tfoot>元素（如果存在）；
- tHead，指向<thead>元素（如果存在）；
- rows，包含表示所有行的 HTMLCollection；
- createTHead()，创建<thead>元素，放到表格中，返回引用；
- createTFoot()，创建<tfoot>元素，放到表格中，返回引用；
- createCaption()，创建<caption>元素，放到表格中，返回引用；
- deleteTHead()，删除<thead>元素；
- deleteTFoot()，删除<tfoot>元素；
- deleteCaption()，删除<caption>元素；
- deleteRow(*pos*)，删除给定位置的行；
- insertRow(*pos*)，在行集合中给定位置插入一行。

<tbody>元素添加了以下属性和方法：

- rows，包含<tbody>元素中所有行的 HTMLCollection；

- deleteRow(*pos*)，删除给定位置的行；
- insertRow(*pos*)，在行集合中给定位置插入一行，返回该行的引用。

<tr>元素添加了以下属性和方法：

- cells，包含<tr>元素所有表元的 HTMLCollection；
- deleteCell(*pos*)，删除给定位置的表元；
- insertCell(*pos*)，在表元集合给定位置插入一个表元，返回该表元的引用。

这些属性和方法极大地减少了创建表格所需的代码量。例如，使用这些方法重写前面的代码之后是这样的（加粗代码表示更新的部分）：

```
// 创建表格
let table = document.createElement("table");
table.border = 1;
table.width = "100%";

// 创建表体
let tbody = document.createElement("tbody");
table.appendChild(tbody);

// 创建第一行
tbody.insertRow(0);
tbody.rows[0].insertCell(0);
tbody.rows[0].cells[0].appendChild(document.createTextNode("Cell 1,1"));
tbody.rows[0].insertCell(1);
tbody.rows[0].cells[1].appendChild(document.createTextNode("Cell 2,1"));

// 创建第二行
tbody.insertRow(1);
tbody.rows[1].insertCell(0);
tbody.rows[1].cells[0].appendChild(document.createTextNode("Cell 1,2"));
tbody.rows[1].insertCell(1);
tbody.rows[1].cells[1].appendChild(document.createTextNode("Cell 2,2"));

// 把表格添加到文档主体
document.body.appendChild(table);
```

这里创建<table>和<tbody>元素的代码没有变。变化的是创建两行的部分，这次使用了 HTML DOM 表格的属性和方法。创建第一行时，在<tbody>元素上调用了 insertRow()方法。传入参数 0，表示把这一行放在什么位置。然后，使用 tbody.rows[0]来引用这一行，因为这一行刚刚创建并被添加到了<tbody>的位置 0。

创建表元的方式也与之类似。在<tr>元素上调用 insertCell()方法，传入参数 0，表示把这个表元放在什么位置上。然后，使用 tbody.rows[0].cells[0]来引用这个表元，因为这个表元刚刚创建并被添加到了<tr>的位置 0。

虽然以上两种代码在技术上都是正确的，但使用这些属性和方法创建表格让代码变得更有逻辑性，也更容易理解。

14.2.4 使用 NodeList

理解 NodeList 对象和相关的 NamedNodeMap、HTMLCollection，是理解 DOM 编程的关键。这 3 个集合类型都是“实时的”，意味着文档结构的变化会实时地在它们身上反映出来，因此它们的值始终

代表最新的状态。实际上，NodeList 就是基于 DOM 文档的实时查询。例如，下面的代码会导致无穷循环：

```
let divs = document.getElementsByTagName("div");

for (let i = 0; i < divs.length; ++i){
  let div = document.createElement("div");
  document.body.appendChild(div);
}
```

第一行取得了包含文档中所有 元素的 HTMLCollection。因为这个集合是“实时的”，所以任何时候只要向页面中添加一个新 元素，再查询这个集合就会多一项。因为浏览器不希望保存每次创建的集合，所以就会在每次访问时更新集合。这样就会出现前面使用循环的例子中所演示的问题。每次循环开始，都会求值 i < divs.length。这意味着要执行获取所有 元素的查询。因为循环体中会创建并向文档添加一个新 元素，所以每次循环 divs.length 的值也会递增。因为两个值都会递增，所以 i 将永远不会等于 divs.length。

使用 ES6 迭代器并不会解决这个问题，因为迭代的是一个永远增长的实时集合。以下代码仍然会导致无穷循环：

```
for (let div of document.getElementsByTagName("div")){
  let newDiv = document.createElement("div");
  document.body.appendChild(newDiv);
}
```

任何时候要迭代 NodeList，最好再初始化一个变量保存当时查询时的长度，然后用循环变量与这个变量进行比较，如下所示：

```
let divs = document.getElementsByTagName("div");

for (let i = 0, len = divs.length; i < len; ++i) {
  let div = document.createElement("div");
  document.body.appendChild(div);
}
```

在这个例子中，又初始化了一个保存集合长度的变量 len。因为 len 保存着循环开始时集合的长度，而这个值不会随集合增大动态增长，所以就可以避免前面例子中出现的无穷循环。本章还会使用这种技术来演示迭代 NodeList 对象的首选方式。

另外，如果不想再初始化一个变量，也可以像下面这样反向迭代集合：

```
let divs = document.getElementsByTagName("div");

for (let i = divs.length - 1; i >= 0; --i) {
  let div = document.createElement("div");
  document.body.appendChild(div);
}
```

一般来说，最好限制操作 NodeList 的次数。因为每次查询都会搜索整个文档，所以最好把查询到的 NodeList 缓存起来。

14.3　MutationObserver 接口

不久前添加到 DOM 规范中的 MutationObserver 接口，可以在 DOM 被修改时异步执行回调。使用 MutationObserver 可以观察整个文档、DOM 树的一部分，或某个元素。此外还可以观察元素属性、

子节点、文本，或者前三者任意组合的变化。

> **注意** 新引进 MutationObserver 接口是为了取代废弃的 MutationEvent。

14.3.1 基本用法

MutationObserver 的实例要通过调用 MutationObserver 构造函数并传入一个回调函数来创建：

```
let observer = new MutationObserver(() => console.log('DOM was mutated!'));
```

1. observe()方法

新创建的 MutationObserver 实例不会关联 DOM 的任何部分。要把这个 observer 与 DOM 关联起来，需要使用 observe()方法。这个方法接收两个必需的参数：要观察其变化的 DOM 节点，以及一个 MutationObserverInit 对象。

MutationObserverInit 对象用于控制观察哪些方面的变化，是一个键/值对形式配置选项的字典。例如，下面的代码会创建一个观察者（observer）并配置它观察<body>元素上的属性变化：

```
let observer = new MutationObserver(() => console.log('<body> attributes changed'));

observer.observe(document.body, { attributes: true });
```

执行以上代码后，<body>元素上任何属性发生变化都会被这个 MutationObserver 实例发现，然后就会异步执行注册的回调函数。<body>元素后代的修改或其他非属性修改都不会触发回调进入任务队列。可以通过以下代码来验证：

```
let observer = new MutationObserver(() => console.log('<body> attributes changed'));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';
console.log('Changed body class');

// Changed body class
// <body> attributes changed
```

注意，回调中的 console.log()是后执行的。这表明回调并非与实际的 DOM 变化同步执行。

2. 回调与 MutationRecord

每个回调都会收到一个 MutationRecord 实例的数组。MutationRecord 实例包含的信息包括发生了什么变化，以及 DOM 的哪一部分受到了影响。因为回调执行之前可能同时发生多个满足观察条件的事件，所以每次执行回调都会传入一个包含按顺序入队的 MutationRecord 实例的数组。

下面展示了反映一个属性变化的 MutationRecord 实例的数组：

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

document.body.setAttribute('foo', 'bar');
// [
//   {
//     addedNodes: NodeList [],
```

```
//     attributeName: "foo",
//     attributeNamespace: null,
//     nextSibling: null,
//     oldValue: null,
//     previousSibling: null
//     removedNodes: NodeList [],
//     target: body
//     type: "attributes"
//   }
// ]
```

下面是一次涉及命名空间的类似变化：

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

document.body.setAttributeNS('baz', 'foo', 'bar');

// [
//   {
//     addedNodes: NodeList [],
//     attributeName: "foo",
//     attributeNamespace: "baz",
//     nextSibling: null,
//     oldValue: null,
//     previousSibling: null
//     removedNodes: NodeList [],
//     target: body
//     type: "attributes"
//   }
// ]
```

连续修改会生成多个 MutationRecord 实例，下次回调执行时就会收到包含所有这些实例的数组，顺序为变化事件发生的顺序：

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';
document.body.className = 'bar';
document.body.className = 'baz';

// [MutationRecord, MutationRecord, MutationRecord]
```

下表列出了 MutationRecord 实例的属性。

属　性	说　明
target	被修改影响的目标节点
type	字符串，表示变化的类型："attributes"、"characterData"或"childList"
oldValue	如果在 MutationObserverInit 对象中启用（attributeOldValue 或 characterData OldValue 为 true），"attributes"或"characterData"的变化事件会设置这个属性为被替代的值 "childList"类型的变化始终将这个属性设置为 null

（续）

属 性	说 明
attributeName	对于"attributes"类型的变化，这里保存被修改属性的名字 其他变化事件会将这个属性设置为 null
attributeNamespace	对于使用了命名空间的"attributes"类型的变化，这里保存被修改属性的名字 其他变化事件会将这个属性设置为 null
addedNodes	对于"childList"类型的变化，返回包含变化中添加节点的 NodeList 默认为空 NodeList
removedNodes	对于"childList"类型的变化，返回包含变化中删除节点的 NodeList 默认为空 NodeList
previousSibling	对于"childList"类型的变化，返回变化节点的前一个同胞 Node 默认为 null
nextSibling	对于"childList"类型的变化，返回变化节点的后一个同胞 Node 默认为 null

传给回调函数的第二个参数是观察变化的 MutationObserver 的实例，演示如下：

```
let observer = new MutationObserver(
    (mutationRecords, mutationObserver) => console.log(mutationRecords,
mutationObserver));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';

// [MutationRecord], MutationObserver
```

3. disconnect()方法

默认情况下，只要被观察的元素不被垃圾回收，MutationObserver 的回调就会响应 DOM 变化事件，从而被执行。要提前终止执行回调，可以调用 disconnect()方法。下面的例子演示了同步调用 disconnect()之后，不仅会停止此后变化事件的回调，也会抛弃已经加入任务队列要异步执行的回调：

```
let observer = new MutationObserver(() => console.log('<body> attributes changed'));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';

observer.disconnect();

document.body.className = 'bar';

//（没有日志输出）
```

要想让已经加入任务队列的回调执行，可以使用 setTimeout()让已经入列的回调执行完毕再调用 disconnect()：

```
let observer = new MutationObserver(() => console.log('<body> attributes changed'));

observer.observe(document.body, { attributes: true });
```