关的库对它们进行转义。

15.3.7 scrollIntoView()

DOM 规范中没有涉及的一个问题是如何滚动页面中的某个区域。为填充这方面的缺失，不同浏览器实现了不同的控制滚动的方式。在所有这些专有方法中，HTML5 选择了标准化 scrollIntoView()。

scrollIntoView()方法存在于所有 HTML 元素上，可以滚动浏览器窗口或容器元素以便包含元素进入视口。这个方法的参数如下：

- alignToTop 是一个布尔值。
  - true：窗口滚动后元素的顶部与视口顶部对齐。
  - false：窗口滚动后元素的底部与视口底部对齐。
- scrollIntoViewOptions 是一个选项对象。
  - behavior：定义过渡动画，可取的值为"smooth"和"auto"，默认为"auto"。
  - block：定义垂直方向的对齐，可取的值为"start"、"center"、"end"和"nearest"，默认为 "start"。
  - inline：定义水平方向的对齐，可取的值为"start"、"center"、"end"和"nearest"，默认为 "nearest"。
- 不传参数等同于 alignToTop 为 true。

来看几个例子：

```
// 确保元素可见
document.forms[0].scrollIntoView();

// 同上
document.forms[0].scrollIntoView(true);
document.forms[0].scrollIntoView({block: 'start'});

// 尝试将元素平滑地滚入视口
document.forms[0].scrollIntoView({behavior: 'smooth', block: 'start'});
```

这个方法可以用来在页面上发生某个事件时引起用户关注。把焦点设置到一个元素上也会导致浏览器将元素滚动到可见位置。

15.4 专有扩展

尽管所有浏览器厂商都理解遵循标准的重要性，但它们也都有为弥补功能缺失而为 DOM 添加专有扩展的历史。虽然这表面上看是一件坏事，但专有扩展也为开发者提供了很多重要功能，而这些功能后来则有可能被标准化，比如进入 HTML5。

除了已经标准化的，各家浏览器还有很多未被标准化的专有扩展。这并不意味着它们将来不会被纳入标准，只不过在本书编写时，它们还只是由部分浏览器专有和采用。

15.4.1 children 属性

IE9 之前的版本与其他浏览器在处理空白文本节点上的差异导致了 children 属性的出现。children 属性是一个 HTMLCollection，只包含元素的 Element 类型的子节点。如果元素的子节点

类型全部是元素类型，那 children 和 childNodes 中包含的节点应该是一样的。可以像下面这样使用 children 属性：

```
let childCount = element.children.length;
let firstChild = element.children[0];
```

15.4.2 contains()方法

DOM 编程中经常需要确定一个元素是不是另一个元素的后代。IE 首先引入了 contains()方法，让开发者可以在不遍历 DOM 的情况下获取这个信息。contains()方法应该在要搜索的祖先元素上调用，参数是待确定的目标节点。

如果目标节点是被搜索节点的后代，contains()返回 true，否则返回 false。下面看一个例子：

```
console.log(document.documentElement.contains(document.body)); // true
```

这个例子测试<html>元素中是否包含<body>元素，在格式正确的 HTML 中会返回 true。

另外，使用 DOM Level 3 的 compareDocumentPosition()方法也可以确定节点间的关系。这个方法会返回表示两个节点关系的位掩码。下表给出了这些位掩码的说明。

掩　码	节点关系
0x1	断开（传入的节点不在文档中）
0x2	领先（传入的节点在 DOM 树中位于参考节点之前）
0x4	随后（传入的节点在 DOM 树中位于参考节点之后）
0x8	包含（传入的节点是参考节点的祖先）
0x10	被包含（传入的节点是参考节点的后代）

要模仿 contains()方法，就需要用到掩码 16（0x10）。compareDocumentPosition()方法的结果可以通过按位与来确定参考节点是否包含传入的节点，比如：

```
let result = document.documentElement.compareDocumentPosition(document.body);
console.log(!!(result & 0x10));
```

以上代码执行后 result 的值为 20（或 0x14，其中 0x4 表示“随后”，加上 0x10“被包含”）。对 result 和 0x10 应用按位与会返回非零值，而两个叹号将这个值转换成对应的布尔值。

IE9 及之后的版本，以及所有现代浏览器都支持 contains()和 compareDocumentPosition()方法。

15.4.3 插入标记

HTML5 将 IE 发明的 innerHTML 和 outerHTML 纳入了标准，但还有两个属性没有入选。这两个剩下的属性是 innerText 和 outerText。

1. innerText 属性

innerText 属性对应元素中包含的所有文本内容，无论文本在子树中哪个层级。在用于读取值时，innerText 会按照深度优先的顺序将子树中所有文本节点的值拼接起来。在用于写入值时，innerText 会移除元素的所有后代并插入一个包含该值的文本节点。来看下面的 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>
```

对这个例子中的 而言，innerText 属性会返回以下字符串：

```
This is a paragraph with a list following it.
Item 1
Item 2
Item 3
```

注意不同浏览器对待空格的方式不同，因此格式化之后的字符串可能包含也可能不包含原始 HTML 代码中的缩进。

下面再看一个使用 innerText 设置 元素内容的例子：

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

执行这行代码后，HTML 页面中的这个 元素实际上会变成这个样子：

```
<div id="content">Hello world!</div>
```

设置 innerText 会移除元素之前所有的后代节点，完全改变 DOM 子树。此外，设置 innerText 也会编码出现在字符串中的 HTML 语法字符（小于号、大于号、引号及和号）。下面是一个例子：

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

执行之后的结果如下：

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

因为设置 innerText 只能在容器元素中生成一个文本节点，所以为了保证一定是文本节点，就必须进行 HTML 编码。innerText 属性可以用于去除 HTML 标签。通过将 innerText 设置为等于 innerText，可以去除所有 HTML 标签而只剩文本，如下所示：

```
div.innerText = div.innerText;
```

执行以上代码后，容器元素的内容只会包含原先的文本内容。

> **注意**　Firefox 45（2016 年 3 月发布）以前的版本中只支持 textContent 属性，与 innerText 的区别是返回的文本中也会返回行内样式或脚本代码。innerText 目前已经得到所有浏览器支持，应该作为取得和设置文本内容的首选方法使用。

2. outerText 属性

outerText 与 innerText 是类似的，只不过作用范围包含调用它的节点。要读取文本值时，outerText 与 innerText 实际上会返回同样的内容。但在写入文本值时，outerText 就大不相同了。写入文本值时，outerText 不止会移除所有后代节点，而是会替换整个元素。比如：

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

这行代码的执行效果就相当于以下两行代码：

```
let text = document.createTextNode("Hello world!");
div.parentNode.replaceChild(text, div);
```

本质上，这相当于用新的文本节点替代 outerText 所在的元素。此时，原来的元素会与文档脱离关系，因此也无法访问。

outerText 是一个非标准的属性，而且也没有被标准化的前景。因此，不推荐依赖这个属性实现重要的操作。除 Firefox 之外所有主流浏览器都支持 outerText。

15.4.4 滚动

如前所述，滚动是 HTML5 之前 DOM 标准没有涉及的领域。虽然 HTML5 把 scrollIntoView() 标准化了，但不同浏览器中仍然有其他专有方法。比如，scrollIntoViewIfNeeded() 作为 HTMLElement 类型的扩展可以在所有元素上调用。scrollIntoViewIfNeeded(alingCenter)会在元素不可见的情况下，将其滚动到窗口或包含窗口中，使其可见；如果已经在视口中可见，则这个方法什么也不做。如果将可选的参数 alingCenter 设置为 true，则浏览器会尝试将其放在视口中央。Safari、Chrome 和 Opera 实现了这个方法。

下面使用 scrollIntoViewIfNeeded()方法的一个例子：

```
// 如果不可见，则将元素可见
document.images[0].scrollIntoViewIfNeeded();
```

考虑到 scrollIntoView()是唯一一个所有浏览器都支持的方法，所以只用它就可以了。

15.5 小结

虽然 DOM 规定了与 XML 和 HTML 文档交互的核心 API，但其他几个规范也定义了对 DOM 的扩展。很多扩展都基于之前的已成为事实标准的专有特性标准化而来。本章主要介绍了以下 3 个规范。

- Selectors API 为基于 CSS 选择符获取 DOM 元素定义了几个方法：querySelector()、querySelectorAll()和 matches()。
- Element Traversal 在 DOM 元素上定义了额外的属性，以方便对 DOM 元素进行遍历。这个需求是因浏览器处理元素间空格的差异而产生的。
- HTML5 为标准 DOM 提供了大量扩展。其中包括对 innerHTML 属性等事实标准进行了标准化，还有焦点管理、字符集、滚动等特性。

DOM 扩展的数量总体还不大，但随着 Web 技术的发展一定会越来越多。浏览器仍然没有停止对专有扩展的探索，如果出现成功的扩展，那么就可能成为事实标准，或者最终被整合到未来的标准中。

第16章

DOM2 和 DOM3

本章内容

- DOM2 到 DOM3 的变化
- 操作样式的 DOM API
- DOM 遍历与范围

DOM1（DOM Level 1）主要定义了 HTML 和 XML 文档的底层结构。DOM2（DOM Level 2）和 DOM3（DOM Level 3）在这些结构之上加入更多交互能力，提供了更高级的 XML 特性。实际上，DOM2 和 DOM3 是按照模块化的思路来制定标准的，每个模块之间有一定关联，但分别针对某个 DOM 子集。这些模式如下所示。

- DOM Core：在 DOM1 核心部分的基础上，为节点增加方法和属性。
- DOM Views：定义基于样式信息的不同视图。
- DOM Events：定义通过事件实现 DOM 文档交互。
- DOM Style：定义以编程方式访问和修改 CSS 样式的接口。
- DOM Traversal and Range：新增遍历 DOM 文档及选择文档内容的接口。
- DOM HTML：在 DOM1 HTML 部分的基础上，增加属性、方法和新接口。
- DOM Mutation Observers：定义基于 DOM 变化触发回调的接口。这个模块是 DOM4 级模块，用于取代 Mutation Events。

本章介绍除 DOM Events 和 DOM Mutation Observers 之外的其他所有模块，第 17 章会专门介绍事件，而 DOM Mutation Observers 第 14 章已经介绍过了。DOM3 还有 XPath 模块和 Load and Save 模块，将在第 22 章介绍。

> **注意** 比较老旧的浏览器（如 IE8）对本章内容支持有限。如果你的项目要兼容这些低版本浏览器，在使用本章介绍的 API 之前先确认浏览器的支持情况。推荐参考 Can I Use 网站。

16.1 DOM 的演进

DOM2 和 DOM3 Core 模块的目标是扩展 DOM API，满足 XML 的所有需求并提供更好的错误处理和特性检测。很大程度上，这意味着支持 XML 命名空间的概念。DOM2 Core 没有新增任何类型，仅仅在 DOM1 Core 基础上增加了一些方法和属性。DOM3 Core 则除了增强原有类型，也新增了一些新类型。

类似地，DOM View 和 HTML 模块也丰富了 DOM 接口，定义了新的属性和方法。这两个模块很小，

因此本章将在讨论 JavaScript 对象的基本变化时将它们与 Core 模块放在一起讨论。

> **注意** 本章只讨论浏览器实现的 DOM API，不会提及未被浏览器实现的。

16.1.1 XML 命名空间

XML 命名空间可以实现在一个格式规范的文档中混用不同的 XML 语言，而不必担心元素命名冲突。严格来讲，XML 命名空间在 XHTML 中才支持，HTML 并不支持。因此，本节的示例使用 XHTML。

命名空间是使用 xmlns 指定的。XHTML 的命名空间是"http://www.w3.org/1999/xhtml"，应该包含在任何格式规范的 XHTML 页面的<html>元素中，如下所示：

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example XHTML page</title>
  </head>
  <body>
    Hello world!
  </body>
</html>
```

对这个例子来说，所有元素都默认属于 XHTML 命名空间。可以使用 xmlns 给命名空间创建一个前缀，格式为“xmlns：前缀”，如下面的例子所示：

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <xhtml:head>
    <xhtml:title>Example XHTML page</xhtml:title>
  </xhtml:head>
  <xhtml:body>
    Hello world!
  </xhtml:body>
</xhtml:html>
```

这里为 XHTML 命名空间定义了一个前缀 xhtml，同时所有 XHTML 元素都必须加上这个前缀。为避免混淆，属性也可以加上命名空间前缀，比如：

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <xhtml:head>
    <xhtml:title>Example XHTML page</xhtml:title>
  </xhtml:head>
  <xhtml:body xhtml:class="home">
    Hello world!
  </xhtml:body>
</xhtml:html>
```

这里的 class 属性被加上了 xhtml 前缀。如果文档中只使用一种 XML 语言，那么命名空间前缀其实是多余的，只有一个文档混合使用多种 XML 语言时才有必要。比如下面这个文档就使用了 XHTML 和 SVG 两种语言：

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example XHTML page</title>
  </head>
  <body>
    <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
```

```
        viewBox="0 0 100 100" style="width:100%; height:100%">
      <rect x="0" y="0" width="100" height="100" style="fill:red" />
    </svg>
  </body>
</html>
```

在这个例子中，通过给<svg>元素设置自己的命名空间，将其标识为当前文档的外来元素。这样一来，<svg>元素及其属性，包括它的所有后代都会被认为属于"https://www.w3.org/2000/svg"命名空间。虽然这个文档从技术角度讲是 XHTML 文档，但由于使用了命名空间，其中包含的 SVG 代码也是有效的。

对于这样的文档，如果调用某个方法与节点交互，就会出现一个问题。比如，创建了一个新元素，那这个元素属于哪个命名空间？查询特定标签名时，结果中应该包含哪个命名空间下的元素？DOM2 Core 为解决这些问题，给大部分 DOM1 方法提供了特定于命名空间的版本。

1. Node 的变化

在 DOM2 中，Node 类型包含以下特定于命名空间的属性：

- localName，不包含命名空间前缀的节点名；
- namespaceURI，节点的命名空间 URL，如果未指定则为 null；
- prefix，命名空间前缀，如果未指定则为 null。

在节点使用命名空间前缀的情况下，nodeName 等于 prefix + ":" + localName。比如下面这个例子：

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example XHTML page</title>
  </head>
  <body>
    <s:svg xmlns:s="http://www.w3.org/2000/svg" version="1.1"
       viewBox="0 0 100 100" style="width:100%; height:100%">
      <s:rect x="0" y="0" width="100" height="100" style="fill:red" />
    </s:svg>
  </body>
</html>
```

其中的<html>元素的 localName 和 tagName 都是"html",namespaceURL 是"http://www.w3.org/1999/xhtml"，而 prefix 是 null。对于<s:svg>元素，localName 是"svg"，tagName 是"s:svg"，namespaceURI 是"https://www.w3.org/2000/svg"，而 prefix 是"s"。

DOM3 进一步增加了如下与命名空间相关的方法：

- isDefaultNamespace(*namespaceURI*)，返回布尔值，表示 *namespaceURI* 是否为节点的默认命名空间；
- lookupNamespaceURI(*prefix*)，返回给定 *prefix* 的命名空间 URI；
- lookupPrefix(*namespaceURI*)，返回给定 *namespaceURI* 的前缀。

对前面的例子，可以执行以下代码：

```
console.log(document.body.isDefaultNamespace("http://www.w3.org/1999/
xhtml")); // true

// 假设 svg 包含对<s:svg>元素的引用
console.log(svg.lookupPrefix("http://www.w3.org/2000/svg"));  // "s"
console.log(svg.lookupNamespaceURI("s"));  // "http://www.w3.org/2000/svg"
```

这些方法主要用于通过元素查询前面和命名空间 URI，以确定元素与文档的关系。

2. Document 的变化

DOM2 在 Document 类型上新增了如下命名空间特定的方法：

- createElementNS(*namespaceURI*, *tagName*)，以给定的标签名 *tagName* 创建指定命名空间 *namespaceURI* 的一个新元素；
- createAttributeNS(*namespaceURI*, *attributeName*)，以给定的属性名 *attributeName* 创建指定命名空间 *namespaceURI* 的一个新属性；
- getElementsByTagNameNS(*namespaceURI*, *tagName*)，返回指定命名空间 *namespaceURI* 中所有标签名为 *tagName* 的元素的 NodeList。

使用这些方法都需要传入相应的命名空间 URI（不是命名空间前缀），如下面的例子所示：

```
// 创建一个新 SVG 元素
let svg = document.createElementNS("http://www.w3.org/2000/svg", "svg");

// 创建一个任意命名空间的新属性
let att = document.createAttributeNS("http://www.somewhere.com", "random");

// 获取所有 XHTML 元素
let elems = document.getElementsByTagNameNS("http://www.w3.org/1999/xhtml", "*");
```

这些命名空间特定的方法只在文档中包含两个或两个以上命名空间时才有用。

3. Element 的变化

DOM2 Core 对 Element 类型的更新主要集中在对属性的操作上。下面是新增的方法：

- getAttributeNS(*namespaceURI*, *localName*)，取得指定命名空间 *namespaceURI* 中名为 *localName* 的属性；
- getAttributeNodeNS(*namespaceURI*, *localName*)，取得指定命名空间 *namespaceURI* 中名为 *localName* 的属性节点；
- getElementsByTagNameNS(*namespaceURI*, *tagName*)，取得指定命名空间 *namespaceURI* 中标签名为 *tagName* 的元素的 NodeList；
- hasAttributeNS(*namespaceURI*, *localName*)，返回布尔值，表示元素中是否有命名空间 *namespaceURI* 下名为 *localName* 的属性（注意，DOM2 Core 也添加不带命名空间的 hasAttribute()方法）；
- removeAttributeNS(*namespaceURI*, *localName*)，删除指定命名空间 *namespaceURI* 中名为 *localName* 的属性；
- setAttributeNS(*namespaceURI*, *qualifiedName*, *value*)，设置指定命名空间 *namespaceURI* 中名为 *qualifiedName* 的属性为 *value*；
- setAttributeNodeNS(*attNode*)，为元素设置（添加）包含命名空间信息的属性节点 *attNode*。

这些方法与 DOM1 中对应的方法行为相同，除 setAttributeNodeNS()之外都只是多了一个命名空间参数。

4. NamedNodeMap 的变化

NamedNodeMap 也增加了以下处理命名空间的方法。因为 NamedNodeMap 主要表示属性，所以这些方法大都适用于属性：

- getNamedItemNS(*namespaceURI*, *localName*)，取得指定命名空间 *namespaceURI* 中名为 *localName* 的项；
- removeNamedItemNS(*namespaceURI*, *localName*)，删除指定命名空间 *namespaceURI* 中名为 *localName* 的项；
- setNamedItemNS(*node*)，为元素设置（添加）包含命名空间信息的节点。

这些方法很少使用，因为通常都是使用元素来访问属性。

16.1.2　其他变化

除命名空间相关的变化，DOM2 Core 还对 DOM 的其他部分做了一些更新。这些变化与 XML 命名空间无关，主要关注 DOM API 的完整性与可靠性。

1. DocumentType 的变化

DocumentType 新增了 3 个属性：publicId、systemId 和 internalSubset。publicId、systemId 属性表示文档类型声明中有效但无法使用 DOM1 API 访问的数据。比如下面这个 HTML 文档类型声明：

```
<!DOCTYPE HTML PUBLIC "-// W3C// DTD HTML 4.01// EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

其 publicId 是"-// W3C// DTD HTML 4.01// EN",而 systemId 是"http://www.w3.org/TR/html4/strict.dtd"。支持 DOM2 的浏览器应该可以运行以下 JavaScript 代码：

```
console.log(document.doctype.publicId);
console.log(document.doctype.systemId);
```

通常在网页中很少需要访问这些信息。

internalSubset 用于访问文档类型声明中可能包含的额外定义，如下面的例子所示：

```
<!DOCTYPE html PUBLIC "-// W3C// DTD XHTML 1.0 Strict// EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
[<!ELEMENT name (#PCDATA)>] >
```

对于以上声明，document.doctype.internalSubset 会返回"<!ELEMENT name (#PCDATA)>"。HTML 文档中几乎不会涉及文档类型的内部子集，XML 文档中稍微常用一些。

2. Document 的变化

Document 类型的更新中唯一跟命名空间无关的方法是 importNode()。这个方法的目的是从其他文档获取一个节点并导入到新文档，以便将其插入新文档。每个节点都有一个 ownerDocument 属性，表示所属文档。如果调用 appendChild()方法时传入节点的 ownerDocument 不是指向当前文档，则会发生错误。而调用 importNode()导入其他文档的节点会返回一个新节点，这个新节点的 ownerDocument 属性是正确的。

importNode()方法跟 cloneNode()方法类似，同样接收两个参数：要复制的节点和表示是否同时复制子树的布尔值，返回结果是适合在当前文档中使用的新节点。下面看一个例子：

```
let newNode = document.importNode(oldNode, true);   // 导入节点及所有后代
document.body.appendChild(newNode);
```

这个方法在 HTML 中使用得并不多，在 XML 文档中的使用会更多一些（第 22 章会深入讨论）。

DOM2 View 给 Document 类型增加了新属性 defaultView，是一个指向拥有当前文档的窗口（或

窗格<frame>)的指针。这个规范中并没有明确视图何时可用，因此这是添加的唯一一个属性。defaultView 属性得到了除 IE8 及更早版本之外所有浏览器的支持。IE8 及更早版本支持等价的 parentWindow 属性，Opera 也支持这个属性。因此要确定拥有文档的窗口，可以使用以下代码：

```
let parentWindow = document.defaultView || document.parentWindow;
```

除了上面这一个方法和一个属性，DOM2 Core 还针对 document.implementation 对象增加了两个新方法：createDocumentType()和 createDocument()。前者用于创建 DocumentType 类型的新节点，接收 3 个参数：文档类型名称、publicId 和 systemId。比如，以下代码可以创建一个新的 HTML 4.01 严格型文档：

```
let doctype = document.implementation.createDocumentType("html",
          "-// W3C// DTD HTML 4.01// EN",
          "http://www.w3.org/TR/html4/strict.dtd");
```

已有文档的文档类型不可更改，因此 createDocumentType()只在创建新文档时才会用到，而创建新文档要使用 createDocument()方法。createDocument()接收 3 个参数：文档元素的 namespaceURI、文档元素的标签名和文档类型。比如，下列代码可以创建一个空的 XML 文档：

```
let doc = document.implementation.createDocument("", "root", null);
```

这个空文档没有命名空间和文档类型，只指定了<root>作为文档元素。要创建一个 XHTML 文档，可以使用以下代码：

```
let doctype = document.implementation.createDocumentType("html",
          "-// W3C// DTD XHTML 1.0 Strict// EN",
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd");

let doc = document.implementation.createDocument("http://www.w3.org/1999/xhtml",
        "html", doctype);
```

这里使用了适当的命名空间和文档类型创建一个新 XHTML 文档。这个文档只有一个文档元素<html>，其他一切都需要另行添加。

DOM2 HTML 模块也为 document.implamentation 对象添加了 createHTMLDocument()方法。使用这个方法可以创建一个完整的 HTML 文档，包含<html>、<head>、<title>和<body>元素。这个方法只接收一个参数，即新创建文档的标题（放到<title>元素中），返回一个新的 HTML 文档。比如：

```
let htmldoc = document.implementation.createHTMLDocument("New Doc");
console.log(htmldoc.title);         // "New Doc"
console.log(typeof htmldoc.body);   // "object"
```

createHTMLDocument()方法创建的对象是 HTMLDocument 类型的实例，因此包括该类型所有相关的方法和属性，包括 title 和 body 属性。

3. Node 的变化

DOM3 新增了两个用于比较节点的方法：isSameNode()和 isEqualNode()。这两个方法都接收一个节点参数，如果这个节点与参考节点相同或相等，则返回 true。节点相同，意味着引用同一个对象；节点相等，意味着节点类型相同，拥有相等的属性（nodeName、nodeValue 等），而且 attributes 和 childNodes 也相等（即同样的位置包含相等的值）。来看一个例子：

```
let div1 = document.createElement("div");
div1.setAttribute("class", "box");

let div2 = document.createElement("div");
```