回结果的精度可能不一样。手机 GPS 的坐标系统可能具有极高的精度，而 IP 地址的精度就要差很多。根据 Geolocation API 规范：

> 地理位置信息的主要来源是 GPS 和 IP 地址、射频识别（RFID）、Wi-Fi 及蓝牙 Mac 地址、GSM/CDMA 蜂窝 ID 以及用户输入等信息。

> **注意**　浏览器也可能会利用 Google Location Service（Chrome 和 Firefox）等服务确定位置。有时候，你可能会发现自己并没有 GPS，但浏览器给出的坐标却非常精确。浏览器会收集所有可用的无线网络，包括 Wi-Fi 和蜂窝信号。拿到这些信息后，再去查询网络数据库。这样就可以精确地报告出你的设备位置。

要获取浏览器当前的位置，可以使用 `getCurrentPosition()` 方法。这个方法返回一个 `Coordinates` 对象，其中包含的信息不一定完全依赖宿主系统的能力：

```
// getCurrentPosition()会以 position 对象为参数调用传入的回调函数
navigator.geolocation.getCurrentPosition((position) => p = position);
```

这个 `position` 对象中有一个表示查询时间的时间戳，以及包含坐标信息的 `Coordinates` 对象：

```
console.log(p.timestamp);  // 1525364883361
console.log(p.coords);     // Coordinates {...}
```

`Coordinates` 对象中包含标准格式的经度和纬度，以及以米为单位的精度。精度同样以确定设备位置的机制来判定。

```
console.log(p.coords.latitude, p.coords.longitude);   // 37.4854409, -122.2325506
console.log(p.coords.accuracy);                       // 58
```

`Coordinates` 对象包含一个 `altitude`（海拔高度）属性，是相对于 1984 世界大地坐标系（World Geodetic System，1984）地球表面的以米为单位的距离。此外也有一个 `altitudeAccuracy` 属性，这个精度值单位也是米。为了取得 `Coordinates` 中包含的这些信息，当前设备必须具备相应的能力（比如 GPS 或高度计）。很多设备因为没有能力测量高度，所以这两个值经常有一个或两个是空的。

```
console.log(p.coords.altitude);          // -8.800000190734863
console.log(p.coords.altitudeAccuracy);  // 200
```

`Coordinates` 对象包含一个 `speed` 属性，表示设备每秒移动的速度。还有一个 `heading`（朝向）属性，表示相对于正北方向移动的角度（$0 \leqslant$ `heading` < 360）。为获取这些信息，当前设备必须具备相应的能力（比如加速计或指南针）。很多设备因为没有能力测量高度，所以这两个值经常有一个是空的，或者两个都是空的。

> **注意**　设备不会根据两点的向量来测量速度和朝向。不过，如果可能的话，可以尝试基于两次连续的测量数据得到的向量来手动计算。当然，如果向量的精度不够，那么计算结果的精度肯定也不够。

获取浏览器地理位置并不能保证成功。因此 `getCurrentPosition()` 方法也接收失败回调函数作为第二个参数，这个函数会收到一个 `PositionError` 对象。在失败的情况下，`PositionError` 对象中会包含一个 `code` 属性和一个 `message` 属性，后者包含对错误的简短描述。`code` 属性是一个整数，表示以下 3 种错误。

- PERMISSION_DENIED：浏览器未被允许访问设备位置。页面第一次尝试访问 Geolocation API 时，浏览器会弹出确认对话框取得用户授权（每个域分别获取）。如果返回了这个错误码，则要么是用户不同意授权，要么是在不安全的环境下访问了 Geolocation API。message 属性还会提供额外信息。
- POSITION_UNAVAILABLE：系统无法返回任何位置信息。这个错误码可能代表各种失败原因，但相对来说并不常见，因为只要设备能上网，就至少可以根据 IP 地址返回一个低精度的坐标。
- TIMEOUT：系统不能在超时时间内返回位置信息。关于如何配置超时，会在后面介绍。

```
// 浏览器会弹出确认对话框请用户允许访问 Geolocation API
// 这个例子显示了用户拒绝之后的结果
navigator.geolocation.getCurrentPosition(
  () => {},
  (e) => {
    console.log(e.code);     // 1
    console.log(e.message);  // User denied Geolocation
  }
);

// 这个例子展示了在不安全的上下文中执行代码的结果
navigator.geolocation.getCurrentPosition(
  () => {},
  (e) => {
    console.log(e.code);     // 1
    console.log(e.message);  // Only secure origins are allowed
  }
);
```

Geolocation API 位置请求可以使用 PositionOptions 对象来配置，作为第三个参数提供。这个对象支持以下 3 个属性。

- enableHighAccuracy：布尔值，true 表示返回的值应该尽量精确，默认值为 false。默认情况下，设备通常会选择最快、最省电的方式返回坐标。这通常意味着返回的是不够精确的坐标。比如，在移动设备上，默认位置查询通常只会采用 Wi-Fi 和蜂窝网络的定位信息。而在 enableHighAccuracy 为 true 的情况下，则会使用设备的 GPS 确定设备位置，并返回这些值的混合结果。使用 GPS 会更耗时、耗电，因此在使用 enableHighAccuracy 配置时要仔细权衡一下。
- timeout：毫秒，表示在以 TIMEOUT 状态调用错误回调函数之前等待的最长时间。默认值是 0xFFFFFFFF（$2^{32}-1$）。0 表示完全跳过系统调用而立即以 TIMEOUT 调用错误回调函数。
- maximumAge：毫秒，表示返回坐标的最长有效期，默认值为 0。因为查询设备位置会消耗资源，所以系统通常会缓存坐标并在下次返回缓存的值（遵从位置缓存失效策略）。系统会计算缓存期，如果 Geolocation API 请求的配置要求比缓存的结果更新，则系统会重新查询并返回值。0 表示强制系统忽略缓存的值，每次都重新查询。而 Infinity 会阻止系统重新查询，只会返回缓存的值。JavaScript 可以通过检查 Position 对象的 timestamp 属性值是否重复来判断返回的是不是缓存值。

2. Connection State 和 NetworkInformation API

浏览器会跟踪网络连接状态并以两种方式暴露这些信息：连接事件和 navigator.onLine 属性。在设备连接到网络时，浏览器会记录这个事实并在 window 对象上触发 online 事件。相应地，当设备

断开网络连接后，浏览器会在 window 对象上触发 offline 事件。任何时候，都可以通过 navigator.onLine 属性来确定浏览器的联网状态。这个属性返回一个布尔值，表示浏览器是否联网。

```
const connectionStateChange = () => console.log(navigator.onLine);

window.addEventListener('online', connectionStateChange);
window.addEventListener('offline', connectionStateChange);

// 设备联网时：
// true

// 设备断网时：
// false
```

当然，到底怎么才算联网取决于浏览器与系统实现。有些浏览器可能会认为只要连接到局域网就算“在线”，而不管是否真正接入了互联网。

navigator 对象还暴露了 NetworkInformation API，可以通过 navigator.connection 属性使用。这个 API 提供了一些只读属性，并为连接属性变化事件处理程序定义了一个事件对象。

以下是 NetworkInformation API 暴露的属性。

- downlink：整数，表示当前设备的带宽（以 Mbit/s 为单位），舍入到最接近的 25kbit/s。这个值可能会根据历史网络吞吐量计算，也可能根据连接技术的能力来计算。
- downlinkMax：整数，表示当前设备最大的下行带宽（以 Mbit/s 为单位），根据网络的第一跳来确定。因为第一跳不一定反映端到端的网络速度，所以这个值只能用作粗略的上限值。
- effectiveType：字符串枚举值，表示连接速度和质量。这些值对应不同的蜂窝数据网络连接技术，但也用于分类无线网络。这个值有以下 4 种可能。
  - slow-2g
    - 往返时间 > 2000ms
    - 下行带宽 < 50kbit/s
  - 2g
    - 2000ms > 往返时间 ⩾ 1400ms
    - 70kbit/s > 下行带宽 ⩾ 50kbit/s
  - 3g
    - 1400ms > 往返时间 ⩾ 270ms
    - 700kbit/s > 下行带宽 ⩾ 70kbit/s
  - 4g
    - 270ms > 往返时间 ⩾ 0ms
    - 下行带宽 ⩾ 700kbit/s
- rtt：毫秒，表示当前网络实际的往返时间，舍入为最接近的 25 毫秒。这个值可能根据历史网络吞吐量计算，也可能根据连接技术的能力来计算。
- type：字符串枚举值，表示网络连接技术。这个值可能为下列值之一。
  - bluetooth：蓝牙。
  - cellular：蜂窝。
  - ethernet：以太网。
  - none：无网络连接。相当于 navigator.onLine === false。

- mixed：多种网络混合。
- other：其他。
- unknown：不确定。
- wifi：Wi-Fi。
- wimax：WiMAX。

❑ saveData：布尔值，表示用户设备是否启用了“节流”（reduced data）模式。

❑ onchange：事件处理程序，会在任何连接状态变化时激发一个 change 事件。可以通过 navigator.connection.addEventListener('change',changeHandler)或 navigator.connection.onchange = changeHandler 等方式使用。

3. Battery Status API

浏览器可以访问设备电池及充电状态的信息。navigator.getBattery()方法会返回一个期约实例，解决为一个 BatteryManager 对象。

```
navigator.getBattery().then((b) => console.log(b));
// BatteryManager { ... }
```

BatteryManager 包含 4 个只读属性，提供了设备电池的相关信息。

❑ charging：布尔值，表示设备当前是否正接入电源充电。如果设备没有电池，则返回 true。

❑ chargingTime：整数，表示预计离电池充满还有多少秒。如果电池已充满或设备没有电池，则返回 0。

❑ dischargingTime：整数，表示预计离电量耗尽还有多少秒。如果设备没有电池，则返回 Infinity。

❑ level：浮点数，表示电量百分比。电量完全耗尽返回 0.0，电池充满返回 1.0。如果设备没有电池，则返回 1.0。

这个 API 还提供了 4 个事件属性，可用于设置在相应的电池事件发生时调用的回调函数。可以通过给 BatteryManager 添加事件监听器，也可以通过给事件属性赋值来使用这些属性。

❑ onchargingchange

❑ onchargingtimechange

❑ ondischargingtimechange

❑ onlevelchange

```
navigator.getBattery().then((battery) => {
  // 添加充电状态变化时的处理程序
  const chargingChangeHandler = () => console.log('chargingchange');
  battery.onchargingchange = chargingChangeHandler;
  // 或
  battery.addEventListener('chargingchange', chargingChangeHandler);

  // 添加充电时间变化时的处理程序
  const chargingTimeChangeHandler = () => console.log('chargingtimechange');
  battery.onchargingtimechange = chargingTimeChangeHandler;
  // 或
  battery.addEventListener('chargingtimechange', chargingTimeChangeHandler);

  // 添加放电时间变化时的处理程序
  const dischargingTimeChangeHandler = () => console.log('dischargingtimechange');
  battery.ondischargingtimechange = dischargingTimeChangeHandler;
  // 或
  battery.addEventListener('dischargingtimechange', dischargingTimeChangeHandler);
```

```
    // 添加电量百分比变化时的处理程序
    const levelChangeHandler = () => console.log('levelchange');
    battery.onlevelchange = levelChangeHandler;
    // 或
    battery.addEventListener('levelchange', levelChangeHandler);
});
```

13.3.3　硬件

浏览器检测硬件的能力相当有限。不过，navigator 对象还是通过一些属性提供了基本信息。

1. 处理器核心数

navigator.hardwareConcurrency 属性返回浏览器支持的逻辑处理器核心数量，包含表示核心数的一个整数值（如果核心数无法确定，这个值就是 1）。关键在于，这个值表示浏览器可以并行执行的最大工作线程数量，不一定是实际的 CPU 核心数。

2. 设备内存大小

navigator.deviceMemory 属性返回设备大致的系统内存大小，包含单位为 GB 的浮点数（舍入为最接近的 2 的幂：512MB 返回 0.5，4GB 返回 4）。

3. 最大触点数

navigator.maxTouchPoints 属性返回触摸屏支持的最大关联触点数量，包含一个整数值。

13.4　小结

客户端检测是 JavaScript 中争议最多的话题之一。因为不同浏览器之间存在差异，所以经常需要根据浏览器的能力来编写不同的代码。客户端检测有不少方式，但下面两种用得最多。

- **能力检测**，在使用之前先测试浏览器的特定能力。例如，脚本可以在调用某个函数之前先检查它是否存在。这种客户端检测方式可以让开发者不必考虑特定的浏览器或版本，而只需关注某些能力是否存在。能力检测不能精确地反映特定的浏览器或版本。
- **用户代理检测**，通过用户代理字符串确定浏览器。用户代理字符串包含关于浏览器的很多信息，通常包括浏览器、平台、操作系统和浏览器版本。用户代理字符串有一个相当长的发展史，很多浏览器都试图欺骗网站相信自己是别的浏览器。用户代理检测也比较麻烦，特别是涉及 Opera 会在代理字符串中隐藏自己信息的时候。即使如此，用户代理字符串也可以用来确定浏览器使用的渲染引擎以及平台，包括移动设备和游戏机。

在选择客户端检测方法时，首选是使用能力检测。特殊能力检测要放在次要位置，作为决定代码逻辑的参考。用户代理检测是最后一个选择，因为它过于依赖用户代理字符串。

浏览器也提供了一些软件和硬件相关的信息。这些信息通过 screen 和 navigator 对象暴露出来。利用这些 API，可以获取关于操作系统、浏览器、硬件、设备位置、电池状态等方面的准确信息。

第 14 章

DOM

本章内容

- 理解文档对象模型（DOM）的构成
- 节点类型
- 浏览器兼容性
- MutationObserver 接口

文档对象模型（DOM，Document Object Model）是 HTML 和 XML 文档的编程接口。DOM 表示由多层节点构成的文档，通过它开发者可以添加、删除和修改页面的各个部分。脱胎于网景和微软早期的动态 HTML（DHTML，Dynamic HTML），DOM 现在是真正跨平台、语言无关的表示和操作网页的方式。

DOM Level 1 在 1998 年成为 W3C 推荐标准，提供了基本文档结构和查询的接口。本章之所以介绍 DOM，主要因为它与浏览器中的 HTML 网页相关，并且在 JavaScript 中提供了 DOM API。

> **注意** IE8 及更低版本中的 DOM 是通过 COM 对象实现的。这意味着这些版本的 IE 中，DOM 对象跟原生 JavaScript 对象具有不同的行为和功能。

14.1 节点层级

任何 HTML 或 XML 文档都可以用 DOM 表示为一个由节点构成的层级结构。节点分很多类型，每种类型对应着文档中不同的信息和（或）标记，也都有自己不同的特性、数据和方法，而且与其他类型有某种关系。这些关系构成了层级，让标记可以表示为一个以特定节点为根的树形结构。以下面的 HTML 为例：

```
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

如果表示为层级结构，则如图 14-1 所示。

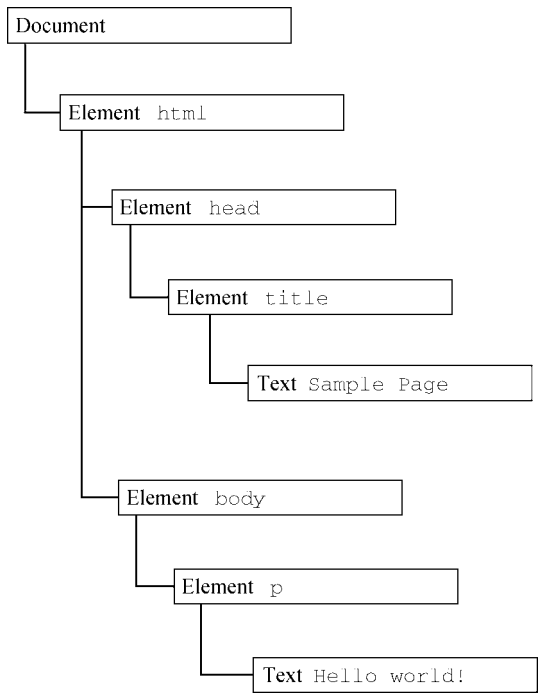

图　14-1

其中，document 节点表示每个文档的根节点。在这里，根节点的唯一子节点是<html>元素，我们称之为**文档元素**（documentElement）。文档元素是文档最外层的元素，所有其他元素都存在于这个元素之内。每个文档只能有一个文档元素。在 HTML 页面中，文档元素始终是<html>元素。在 XML 文档中，则没有这样预定义的元素，任何元素都可能成为文档元素。

HTML 中的每段标记都可以表示为这个树形结构中的一个节点。元素节点表示 HTML 元素，属性节点表示属性，文档类型节点表示文档类型，注释节点表示注释。DOM 中总共有 12 种节点类型，这些类型都继承一种基本类型。

14.1.1　Node 类型

DOM Level 1 描述了名为 Node 的接口，这个接口是所有 DOM 节点类型都必须实现的。Node 接口在 JavaScript 中被实现为 Node 类型，在除 IE 之外的所有浏览器中都可以直接访问这个类型。在 JavaScript 中，所有节点类型都继承 Node 类型，因此所有类型都共享相同的基本属性和方法。

每个节点都有 nodeType 属性，表示该节点的类型。节点类型由定义在 Node 类型上的 12 个数值常量表示：

- Node.ELEMENT_NODE（1）
- Node.ATTRIBUTE_NODE（2）
- Node.TEXT_NODE（3）
- Node.CDATA_SECTION_NODE（4）

- Node.ENTITY_REFERENCE_NODE（5）
- Node.ENTITY_NODE（6）
- Node.PROCESSING_INSTRUCTION_NODE（7）
- Node.COMMENT_NODE（8）
- Node.DOCUMENT_NODE（9）
- Node.DOCUMENT_TYPE_NODE（10）
- Node.DOCUMENT_FRAGMENT_NODE（11）
- Node.NOTATION_NODE（12）

节点类型可通过与这些常量比较来确定，比如：

```
if (someNode.nodeType == Node.ELEMENT_NODE){
  alert("Node is an element.");
}
```

这个例子比较了 someNode.nodeType 与 Node.ELEMENT_NODE 常量。如果两者相等，则意味着 someNode 是一个元素节点。

浏览器并不支持所有节点类型。开发者最常用到的是元素节点和文本节点。本章后面会讨论每种节点受支持的程度及其用法。

1. nodeName 与 nodeValue

nodeName 与 nodeValue 保存着有关节点的信息。这两个属性的值完全取决于节点类型。在使用这两个属性前，最好先检测节点类型，如下所示：

```
if (someNode.nodeType == 1){
  value = someNode.nodeName; // 会显示元素的标签名
}
```

在这个例子中，先检查了节点是不是元素。如果是，则将其 nodeName 的值赋给一个变量。对元素而言，nodeName 始终等于元素的标签名，而 nodeValue 则始终为 null。

2. 节点关系

文档中的所有节点都与其他节点有关系。这些关系可以形容为家族关系，相当于把文档树比作家谱。在 HTML 中，<body>元素是<html>元素的子元素，而<html>元素则是<body>元素的父元素。<head>元素是<body>元素的同胞元素，因为它们有共同的父元素<html>。

每个节点都有一个 childNodes 属性，其中包含一个 NodeList 的实例。NodeList 是一个类数组对象，用于存储可以按位置存取的有序节点。注意，NodeList 并不是 Array 的实例，但可以使用中括号访问它的值，而且它也有 length 属性。NodeList 对象独特的地方在于，它其实是一个对 DOM 结构的查询，因此 DOM 结构的变化会自动地在 NodeList 中反映出来。我们通常说 NodeList 是实时的活动对象，而不是第一次访问时所获得内容的快照。

下面的例子展示了如何使用中括号或使用 item()方法访问 NodeList 中的元素：

```
let firstChild = someNode.childNodes[0];
let secondChild = someNode.childNodes.item(1);
let count = someNode.childNodes.length;
```

无论是使用中括号还是 item()方法都是可以的，但多数开发者倾向于使用中括号，因为它是一个类数组对象。注意，length 属性表示那一时刻 NodeList 中节点的数量。使用 Array.prototype.slice()可以像前面介绍 arguments 时一样把 NodeList 对象转换为数组。比如：

```
let arrayOfNodes = Array.prototype.slice.call(someNode.childNodes,0);
```

当然，使用 ES6 的 Array.from()静态方法，可以替换这种笨拙的方式：

```
let arrayOfNodes = Array.from(someNode.childNodes);
```

每个节点都有一个 parentNode 属性，指向其 DOM 树中的父元素。childNodes 中的所有节点都有同一个父元素，因此它们的 parentNode 属性都指向同一个节点。此外，childNodes 列表中的每个节点都是同一列表中其他节点的同胞节点。而使用 previousSibling 和 nextSibling 可以在这个列表的节点间导航。这个列表中第一个节点的 previousSibling 属性是 null，最后一个节点的 nextSibling 属性也是 null，如下所示：

```
if (someNode.nextSibling === null){
  alert("Last node in the parent's childNodes list.");
} else if (someNode.previousSibling === null){
  alert("First node in the parent's childNodes list.");
}
```

注意，如果 childNodes 中只有一个节点，则它的 previousSibling 和 nextSibling 属性都是 null。

父节点和它的第一个及最后一个子节点也有专门属性：firstChild 和 lastChild 分别指向 childNodes 中的第一个和最后一个子节点。someNode.firstChild 的值始终等于 someNode.childNodes[0]，而 someNode.lastChild 的值始终等于 someNode.childNodes[someNode.childNodes.length-1]。如果只有一个子节点，则 firstChild 和 lastChild 指向同一个节点。如果没有子节点，则 firstChild 和 lastChild 都是 null。上述这些节点之间的关系为在文档树的节点之间导航提供了方便。图 14-2 形象地展示了这些关系。

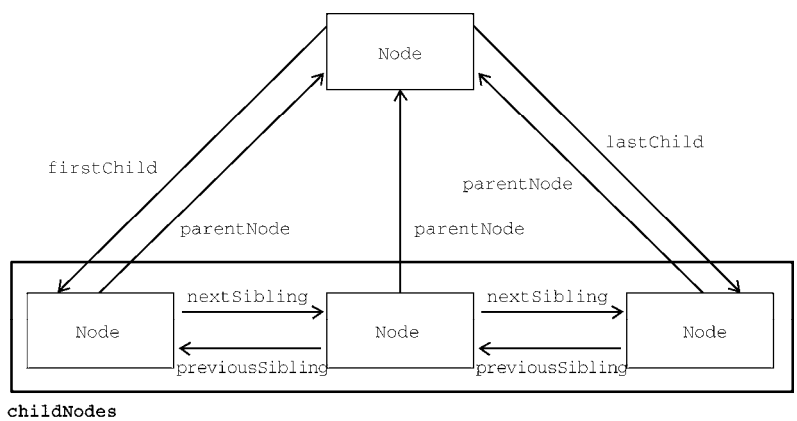

图　14-2

有了这些关系，childNodes 属性的作用远远不止是必备属性那么简单了。这是因为利用这些关系指针，几乎可以访问到文档树中的任何节点，而这种便利性是 childNodes 的最大亮点。还有一个便利的方法是 hasChildNodes()，这个方法如果返回 true 则说明节点有一个或多个子节点。相比查询 childNodes 的 length 属性，这个方法无疑更方便。

最后还有一个所有节点都共享的关系。ownerDocument 属性是一个指向代表整个文档的文档节点的指针。所有节点都被创建它们（或自己所在）的文档所拥有，因为一个节点不可能同时存在于两个或

者多个文档中。这个属性为迅速访问文档节点提供了便利，因为无需在文档结构中逐层上溯了。

> **注意** 虽然所有节点类型都继承了 Node，但并非所有节点都有子节点。本章后面会讨论不同节点类型的差异。

3. 操纵节点

因为所有关系指针都是只读的，所以 DOM 又提供了一些操纵节点的方法。最常用的方法是 appendChild()，用于在 childNodes 列表末尾添加节点。添加新节点会更新相关的关系指针，包括父节点和之前的最后一个子节点。appendChild()方法返回新添加的节点，如下所示：

```
let returnedNode = someNode.appendChild(newNode);
alert(returnedNode == newNode);         // true
alert(someNode.lastChild == newNode);  // true
```

如果把文档中已经存在的节点传给 appendChild()，则这个节点会从之前的位置被转移到新位置。即使 DOM 树通过各种关系指针维系，一个节点也不会在文档中同时出现在两个或更多个地方。因此，如果调用 appendChild()传入父元素的第一个子节点，则这个节点会成为父元素的最后一个子节点，如下所示：

```
// 假设 someNode 有多个子节点
let returnedNode = someNode.appendChild(someNode.firstChild);
alert(returnedNode == someNode.firstChild);  // false
alert(returnedNode == someNode.lastChild);   // true
```

如果想把节点放到 childNodes 中的特定位置而不是末尾，则可以使用 insertBefore()方法。这个方法接收两个参数：要插入的节点和参照节点。调用这个方法后，要插入的节点会变成参照节点的前一个同胞节点，并被返回。如果参照节点是 null，则 insertBefore()与 appendChild()效果相同，如下面的例子所示：

```
// 作为最后一个子节点插入
returnedNode = someNode.insertBefore(newNode, null);
alert(newNode == someNode.lastChild);  // true

// 作为新的第一个子节点插入
returnedNode = someNode.insertBefore(newNode, someNode.firstChild);
alert(returnedNode == newNode);          // true
alert(newNode == someNode.firstChild);  // true

// 插入最后一个子节点前面
returnedNode = someNode.insertBefore(newNode, someNode.lastChild);
alert(newNode == someNode.childNodes[someNode.childNodes.length - 2]); // true
```

appendChild()和 insertBefore()在插入节点时不会删除任何已有节点。相对地，replaceChild()方法接收两个参数：要插入的节点和要替换的节点。要替换的节点会被返回并从文档树中完全移除，要插入的节点会取而代之。下面看一个例子：

```
// 替换第一个子节点
let returnedNode = someNode.replaceChild(newNode, someNode.firstChild);

// 替换最后一个子节点
returnedNode = someNode.replaceChild(newNode, someNode.lastChild);
```

使用 replaceChild()插入一个节点后，所有关系指针都会从被替换的节点复制过来。虽然被替换