下面这个函数检测元素是不是位于顶部，如果不是则把它滚动回顶部：

```
function scrollToTop(element) {
  if (element.scrollTop != 0) {
    element.scrollTop = 0;
  }
}
```

这个函数使用 scrollTop 获取并设置值。

4. 确定元素尺寸

浏览器在每个元素上都暴露了 getBoundingClientRect()方法，返回一个 DOMRect 对象，包含 6 个属性：left、top、right、bottom、height 和 width。这些属性给出了元素在页面中相对于视口的位置。图 16-4[①]展示了这些属性的含义。

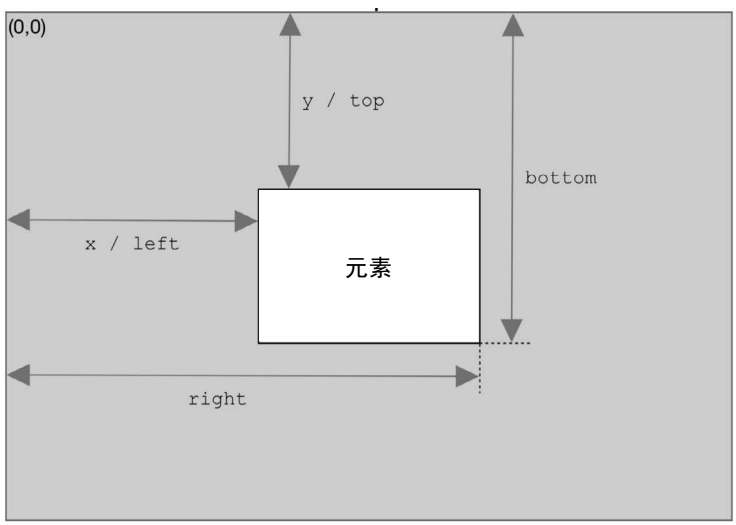

图　16-4

16.3　遍历

DOM2 Traversal and Range 模块定义了两个类型用于辅助顺序遍历 DOM 结构。这两个类型——NodeIterator 和 TreeWalker——从某个起点开始执行对 DOM 结构的深度优先遍历。

如前所述，DOM 遍历是对 DOM 结构的深度优先遍历，至少允许朝两个方向移动（取决于类型）。遍历以给定节点为根，不能在 DOM 中向上超越这个根节点。来看下面的 HTML：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
```

① 这张插图为译者补充，图片来源为 MDN 文档的 Element.getBoundingClientRect()英文版页面。——译者注

```
    <p><b>Hello</b> world!</p>
  </body>
</html>
```

这段代码构成的 DOM 树如图 16-5 所示。

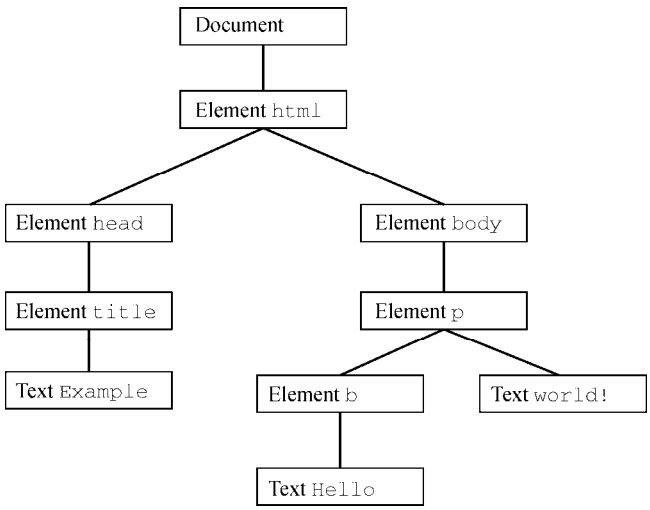

图 16-5

其中的任何节点都可以成为遍历的根节点。比如，假设以<body>元素作为遍历的根节点，那么接下来是<p>元素、<b>元素和两个文本节点（都是<body>元素的后代）。但这个遍历不会到达<html>元素、<head>元素，或者其他不属于<body>元素子树的元素。而以 document 为根节点的遍历，则可以访问到文档中的所有节点。图 16-6 展示了以 document 为根节点的深度优先遍历。

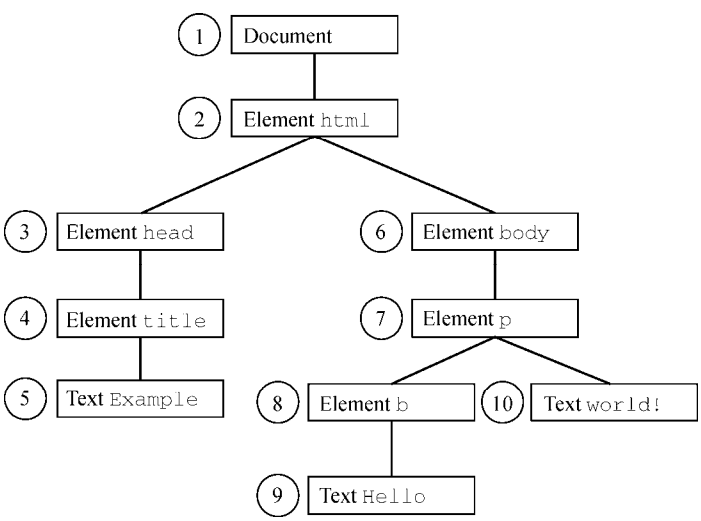

图 16-6

从 document 开始，然后循序移动，第一个节点是 document，最后一个节点是包含" world!"的文本节点。到达文档末尾最后那个文本节点后，遍历会在 DOM 树中反向回溯。此时，第一个访问的节点就是包含" world!"的文本节点,而最后一个是 document 节点本身。NodeIterator 和 TreeWalker 都以这种方式进行遍历。

16.3.1 NodeIterator

NodeIterator 类型是两个类型中比较简单的，可以通过 document.createNodeIterator()方法创建其实例。这个方法接收以下 4 个参数。

- root，作为遍历根节点的节点。
- whatToShow，数值代码，表示应该访问哪些节点。
- filter，NodeFilter 对象或函数，表示是否接收或跳过特定节点。
- entityReferenceExpansion，布尔值，表示是否扩展实体引用。这个参数在 HTML 文档中没有效果，因为实体引用永远不扩展。

whatToShow 参数是一个位掩码，通过应用一个或多个过滤器来指定访问哪些节点。这个参数对应的常量是在 NodeFilter 类型中定义的。

- NodeFilter.SHOW_ALL，所有节点。
- NodeFilter.SHOW_ELEMENT，元素节点。
- NodeFilter.SHOW_ATTRIBUTE，属性节点。由于 DOM 的结构，因此实际上用不上。
- NodeFilter.SHOW_TEXT，文本节点。
- NodeFilter.SHOW_CDATA_SECTION，CData 区块节点。不是在 HTML 页面中使用的。
- NodeFilter.SHOW_ENTITY_REFERENCE，实体引用节点。不是在 HTML 页面中使用的。
- NodeFilter.SHOW_ENTITY，实体节点。不是在 HTML 页面中使用的。
- NodeFilter.SHOW_PROCESSING_INSTRUCTION,处理指令节点。不是在 HTML 页面中使用的。
- NodeFilter.SHOW_COMMENT，注释节点。
- NodeFilter.SHOW_DOCUMENT，文档节点。
- NodeFilter.SHOW_DOCUMENT_TYPE，文档类型节点。
- NodeFilter.SHOW_DOCUMENT_FRAGMENT，文档片段节点。不是在 HTML 页面中使用的。
- NodeFilter.SHOW_NOTATION，记号节点。不是在 HTML 页面中使用的。

这些值除了 NodeFilter.SHOW_ALL 之外，都可以组合使用。比如，可以像下面这样使用按位或操作组合多个选项：

```
let whatToShow = NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT;
```

createNodeIterator()方法的 filter 参数可以用来指定自定义 NodeFilter 对象，或者一个作为节点过滤器的函数。NodeFilter 对象只有一个方法 acceptNode()，如果给定节点应该访问就返回 NodeFilter.FILTER_ACCEPT，否则返回 NodeFilter.FILTER_SKIP。因为 NodeFilter 是一个抽象类型，所以不可能创建它的实例。只要创建一个包含 acceptNode()的对象，然后把它传给 createNodeIterator()就可以了。以下代码定义了只接收<p>元素的节点过滤器对象：

```
let filter = {
  acceptNode(node) {
    return node.tagName.toLowerCase() == "p" ?
```

```
            NodeFilter.FILTER_ACCEPT :
            NodeFilter.FILTER_SKIP;
    }
};

let iterator = document.createNodeIterator(root, NodeFilter.SHOW_ELEMENT,
                                           filter, false);
```

filter 参数还可以是一个函数，与 acceptNode()的形式一样，如下面的例子所示：

```
let filter = function(node) {
    return node.tagName.toLowerCase() == "p" ?
           NodeFilter.FILTER_ACCEPT :
           NodeFilter.FILTER_SKIP;
};

let iterator = document.createNodeIterator(root, NodeFilter.SHOW_ELEMENT,
                                           filter, false);
```

通常，JavaScript 会使用这种形式，因为更简单也更像普通 JavaScript 代码。如果不需要指定过滤器，则可以给这个参数传入 null。

要创建一个简单的遍历所有节点的 NodeIterator，可以使用以下代码：

```
let iterator = document.createNodeIterator(document, NodeFilter.SHOW_ALL,
                                           null, false);
```

NodeIterator 的两个主要方法是 nextNode()和 previousNode()。nextNode()方法在 DOM 子树中以深度优先方式进前一步，而 previousNode()则是在遍历中后退一步。创建 NodeIterator 对象的时候，会有一个内部指针指向根节点，因此第一次调用 nextNode()返回的是根节点。当遍历到达 DOM 树最后一个节点时，nextNode()返回 null。previousNode()方法也是类似的。当遍历到达 DOM 树最后一个节点时，调用 previousNode()返回遍历的根节点后，再次调用也会返回 null。

以下面的 HTML 片段为例：

```
<div id="div1">
  <p><b>Hello</b> world!</p>
  <ul>
    <li>List item 1</li>
    <li>List item 2</li>
    <li>List item 3</li>
  </ul>
</div>
```

假设想要遍历 元素内部的所有元素，那么可以使用如下代码：

```
let div = document.getElementById("div1");
let iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
                                           null, false);

let node = iterator.nextNode();
while (node !== null) {
  console.log(node.tagName);     // 输出标签名
  node = iterator.nextNode();
}
```

这个例子中第一次调用 nextNode()返回 元素。因为 nextNode()在遍历到达 DOM 子树末尾时返回 null，所以这里通过 while 循环检测每次调用 nextNode()的返回值是不是 null。以上代

码执行后会输出以下标签名：

```
DIV
P
B
UL
LI
LI
LI
```

如果只想遍历<li>元素，可以传入一个过滤器，比如：

```
let div = document.getElementById("div1");
let filter = function(node) {
  return node.tagName.toLowerCase() == "li" ?
    NodeFilter.FILTER_ACCEPT :
    NodeFilter.FILTER_SKIP;
};

let iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
       filter, false);

let node = iterator.nextNode();
while (node !== null) {
  console.log(node.tagName);     // 输出标签名
  node = iterator.nextNode();
}
```

在这个例子中，遍历只会输出<li>元素的标签。

nextNode()和 previousNode()方法共同维护 NodeIterator 对 DOM 结构的内部指针，因此修改 DOM 结构也会体现在遍历中。

16.3.2 TreeWalker

TreeWalker 是 NodeIterator 的高级版。除了包含同样的 nextNode()、previousNode()方法，TreeWalker 还添加了如下在 DOM 结构中向不同方向遍历的方法。

- parentNode()，遍历到当前节点的父节点。
- firstChild()，遍历到当前节点的第一个子节点。
- lastChild()，遍历到当前节点的最后一个子节点。
- nextSibling()，遍历到当前节点的下一个同胞节点。
- previousSibling()，遍历到当前节点的上一个同胞节点。

TreeWalker 对象要调用 document.createTreeWalker()方法来创建，这个方法接收与 document.createNodeIterator()同样的参数：作为遍历起点的根节点、要查看的节点类型、节点过滤器和一个表示是否扩展实体引用的布尔值。因为两者很类似，所以 TreeWalker 通常可以取代 NodeIterator，比如：

```
let div = document.getElementById("div1");
let filter = function(node) {
  return node.tagName.toLowerCase() == "li" ?
    NodeFilter.FILTER_ACCEPT :
    NodeFilter.FILTER_SKIP;
};
```

```
let walker = document.createTreeWalker(div, NodeFilter.SHOW_ELEMENT,
                                       filter, false);

let node = iterator.nextNode();
while (node !== null) {
  console.log(node.tagName);      // 输出标签名
  node = iterator.nextNode();
}
```

不同的是，节点过滤器（filter）除了可以返回 NodeFilter.FILTER_ACCEPT 和 NodeFilter.FILTER_SKIP，还可以返回 NodeFilter.FILTER_REJECT。在使用 NodeIterator 时，NodeFilter.FILTER_SKIP 和 NodeFilter.FILTER_REJECT 是一样的。但在使用 TreeWalker 时，NodeFilter.FILTER_SKIP 表示跳过节点，访问子树中的下一个节点，而 NodeFilter.FILTER_REJECT 则表示跳过该节点以及该节点的整个子树。例如，如果把前面示例中的过滤器函数改为返回 NodeFilter.FILTER_REJECT（而不是 NodeFilter.FILTER_SKIP），则会导致遍历立即返回，不会访问任何节点。这是因为第一个返回的元素是 ，其中标签名不是"li"，因此过滤函数返回 NodeFilter.FILTER_REJECT，表示要跳过整个子树。因为 本身就是遍历的根节点，所以遍历会就此结束。

当然，TreeWalker 真正的威力是可以在 DOM 结构中四处游走。如果不使用过滤器，单纯使用 TreeWalker 的漫游能力同样可以在 DOM 树中访问<li>元素，比如：

```
let div = document.getElementById("div1");
let walker = document.createTreeWalker(div, NodeFilter.SHOW_ELEMENT, null, false);

walker.firstChild();   // 前往<p>
walker.nextSibling();  // 前往<ul>

let node = walker.firstChild();  // 前往第一个<li>
while (node !== null) {
  console.log(node.tagName);
  node = walker.nextSibling();
}
```

因为我们知道<li>元素在文档结构中的位置，所以可以直接定位过去。先使用 firstChild()前往<p>元素，再通过 nextSibling()前往<ul>元素，然后使用 firstChild()到达第一个<li>元素。注意，此时的 TreeWalker 只返回元素（这是因为传给 createTreeWalker()的第二个参数）。最后就可以使用 nextSibling()访问每个<li>元素，直到再也没有元素，此时方法返回 null。

TreeWalker 类型也有一个名为 currentNode 的属性，表示遍历过程中上一次返回的节点（无论使用的是哪个遍历方法）。可以通过修改这个属性来影响接下来遍历的起点，如下面的例子所示：

```
let node = walker.nextNode();
console.log(node === walker.currentNode);  // true
walker.currentNode = document.body;        // 修改起点
```

相比于 NodeIterator，TreeWalker 类型为遍历 DOM 提供了更大的灵活性。

16.4 范围

为了支持对页面更细致的控制，DOM2 Traversal and Range 模块定义了范围接口。范围可用于在文档中选择内容，而不用考虑节点之间的界限。（选择在后台发生，用户是看不到的。）范围在常规 DOM 操作的粒度不够时可以发挥作用。

16.4.1　DOM 范围

DOM2 在 Document 类型上定义了一个 createRange()方法，暴露在 document 对象上。使用这个方法可以创建一个 DOM 范围对象，如下所示：

```
let range = document.createRange();
```

与节点类似，这个新创建的范围对象是与创建它的文档关联的，不能在其他文档中使用。然后可以使用这个范围在后台选择文档特定的部分。创建范围并指定它的位置之后，可以对范围的内容执行一些操作，从而实现对底层 DOM 树更精细的控制。

每个范围都是 Range 类型的实例，拥有相应的属性和方法。下面的属性提供了与范围在文档中位置相关的信息。

- startContainer，范围起点所在的节点（选区中第一个子节点的父节点）。
- startOffset，范围起点在 startContainer 中的偏移量。如果 startContainer 是文本节点、注释节点或 CData 区块节点，则 startOffset 指范围起点之前跳过的字符数；否则，表示范围中第一个节点的索引。
- endContainer，范围终点所在的节点（选区中最后一个子节点的父节点）。
- endOffset，范围起点在 startContainer 中的偏移量（与 startOffset 中偏移量的含义相同）。
- commonAncestorContainer，文档中以 startContainer 和 endContainer 为后代的最深的节点。

这些属性会在范围被放到文档中特定位置时获得相应的值。

16.4.2　简单选择

通过范围选择文档中某个部分最简单的方式，就是使用 selectNode()或 selectNodeContents()方法。这两个方法都接收一个节点作为参数，并将该节点的信息添加到调用它的范围。selectNode()方法选择整个节点，包括其后代节点，而 selectNodeContents()只选择节点的后代。假设有如下 HTML：

```
<!DOCTYPE html>
<html>
  <body>
    <p id="p1"><b>Hello</b> world!</p>
  </body>
</html>
```

以下 JavaScript 代码可以访问并创建相应的范围：

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

例子中的这两个范围包含文档的不同部分。range1 包含<p>元素及其所有后代，而 range2 包含<b>元素、文本节点"Hello"和文本节点" world!"，如图 16-7 所示。

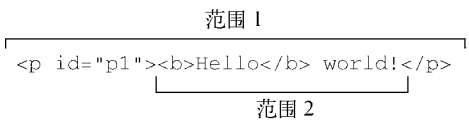

图　16-7

调用 selectNode()时，startContainer、endContainer 和 commonAncestorContainer 都等于传入节点的父节点。在这个例子中，这几个属性都等于 document.body。startOffset 属性等于传入节点在其父节点 childNodes 集合中的索引（在这个例子中，startOffset 等于 1，因为 DOM 的合规实现把空格当成文本节点），而 endOffset 等于 startOffset 加 1（因为只选择了一个节点）。

在调用 selectNodeContents()时，startContainer、endContainer 和 commonAncestorContainer 属性就是传入的节点，在这个例子中是<p>元素。startOffset 属性始终为 0，因为范围从传入节点的第一个子节点开始，而 endOffset 等于传入节点的子节点数量（node.childNodes.length），在这个例子中等于 2。

在像上面这样选定节点或节点后代之后，还可以在范围上调用相应的方法，实现对范围中选区的更精细控制。

- setStartBefore(*refNode*)，把范围的起点设置到 *refNode* 之前，从而让 *refNode* 成为选区的第一个子节点。startContainer 属性被设置为 refNode.parentNode，而 startOffset 属性被设置为 *refNode* 在其父节点 childNodes 集合中的索引。
- setStartAfter(*refNode*)，把范围的起点设置到 *refNode* 之后，从而将 *refNode* 排除在选区之外，让其下一个同胞节点成为选区的第一个子节点。startContainer 属性被设置为 refNode.parentNode，startOffset 属性被设置为 *refNode* 在其父节点 childNodes 集合中的索引加 1。
- setEndBefore(*refNode*)，把范围的终点设置到 *refNode* 之前，从而将 *refNode* 排除在选区之外、让其上一个同胞节点成为选区的最后一个子节点。endContainer 属性被设置为 refNode.parentNode，endOffset 属性被设置为 *refNode* 在其父节点 childNodes 集合中的索引。
- setEndAfter(*refNode*)，把范围的终点设置到 *refNode* 之后，从而让 *refNode* 成为选区的最后一个子节点。endContainer 属性被设置为 refNode.parentNode，endOffset 属性被设置为 *refNode* 在其父节点 childNodes 集合中的索引加 1。

调用这些方法时，所有属性都会自动重新赋值。不过，为了实现复杂的选区，也可以直接修改这些属性的值。

16.4.3 复杂选择

要创建复杂的范围，需要使用 setStart()和 setEnd()方法。这两个方法都接收两个参数：参照节点和偏移量。对 setStart()来说，参照节点会成为 startContainer，而偏移量会赋值给 startOffset。对 setEnd()而言，参照节点会成为 endContainer，而偏移量会赋值给 endOffset。

使用这两个方法，可以模拟 selectNode()和 selectNodeContents()的行为。比如：

```
let range1 = document.createRange(),
    range2 = document.createRange(),
    p1 = document.getElementById("p1"),
    p1Index = -1,
    i,
    len;
for (i = 0, len = p1.parentNode.childNodes.length; i < len; i++) {
  if (p1.parentNode.childNodes[i] === p1) {
    p1Index = i;
    break;
  }
}
```

```
range1.setStart(p1.parentNode, p1Index);
range1.setEnd(p1.parentNode, p1Index + 1);
range2.setStart(p1, 0);
range2.setEnd(p1, p1.childNodes.length);
```

注意，要选择节点（使用 range1），必须先确定给定节点（p1）在其父节点 childNodes 集合中的索引。而要选择节点的内容（使用 range2），则不需要这样计算，因为可以直接给 setStart()和 setEnd()传默认值。虽然可以模拟 selectNode()和 selectNodeContents()，但 setStart()和 setEnd()真正的威力还是选择节点中的某个部分。

假设我们想通过范围从前面示例中选择从"Hello"中的"llo"到" world!"中的"o"的部分。很简单，第一步是取得所有相关节点的引用，如下面的代码所示：

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

文本"Hello"其实是<p>的孙子节点，因为它是<b>的子节点。为此可以使用 p1.firstChild 取得<b>，而使用 p1.firstChild.firstChild 取得"Hello"这个文本节点。文本节点" world!"是<p>的第二个（也是最后一个）子节点，因此可以使用 p1.lastChild 来取得它。然后，再创建范围，指定其边界，如下所示：

```
let range = document.createRange();
range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);
```

因为选区起点在"Hello"中的字母"e"之后，所以要给 setStart()传入 helloNode 和偏移量 2（"e"后面的位置，"H"的位置是 0）。要设置选区终点，则要给 setEnd()传入 worldNode 和偏移量 3，即不属于选区的第一个字符的位置，也就是"r"的位置 3（位置 0 是一个空格）。图 16-8 展示了范围对应的选区。

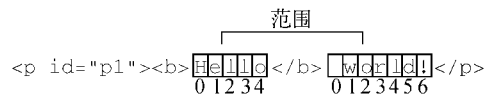

图　16-8

因为 helloNode 和 worldNode 是文本节点，所以它们会成为范围的 startContainer 和 endContainer，这样 startOffset 和 endOffset 实际上表示每个节点中文本字符的位置，而不是子节点的位置（传入元素节点时的情形）。而 commonAncestorContainer 是<p>元素，即包含这两个节点的第一个祖先节点。

当然，只选择文档中的某个部分并不是特别有用，除非可以对选中部分执行操作。

16.4.4　操作范围

创建范围之后，浏览器会在内部创建一个文档片段节点，用于包含范围选区中的节点。为操作范围的内容，选区中的内容必须格式完好。在前面的例子中，因为范围的起点和终点都在文本节点内部，并不是完好的 DOM 结构，所以无法在 DOM 中表示。不过，范围能够确定缺失的开始和结束标签，从而可以重构出有效的 DOM 结构，以便后续操作。

仍以前面例子中的范围来说，范围发现选区中缺少一个开始的标签，于是会在后台动态补上这

个标签，同时还需要补上封闭"He"的结束标签，结果会把 DOM 修改为这样：

```
<p><b>He</b><b>llo</b> world!</p>
```

而且，" world!"文本节点会被拆分成两个文本节点，一个包含" wo"，另一个包含"rld!"。最终的 DOM 树，以及范围对应的文档片段如图 16-9 所示。

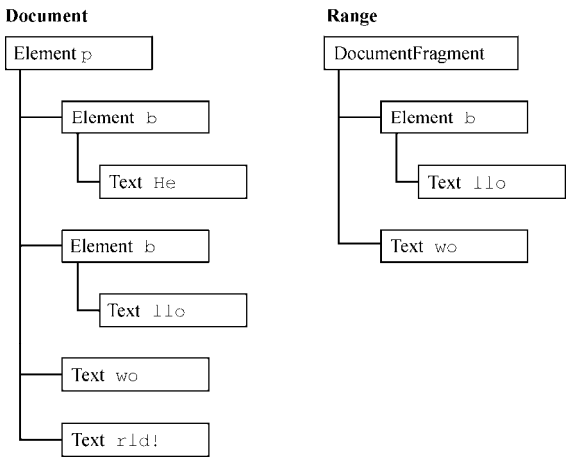

图 16-9

这样创建了范围之后，就可以使用很多方法来操作范围的内容。（注意，范围对应文档片段中的所有节点，都是文档中相应节点的指针。）

第一个方法最容易理解和使用：deleteContents()。顾名思义，这个方法会从文档中删除范围包含的节点。下面是一个例子：

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

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

range.deleteContents();
```

执行上面的代码之后，页面中的 HTML 会变成这样：

```
<p><b>He</b>rld!</p>
```

因为前面介绍的范围选择过程通过修改底层 DOM 结构保证了结构完好，所以即使删除范围之后，剩下的 DOM 结构照样是完好的。

另一个方法 extractContents()跟 deleteContents()类似，也会从文档中移除范围选区。但不同的是，extractContents()方法返回范围对应的文档片段。这样，就可以把范围选中的内容插入文档中其他地方。来看一个例子：

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