《JavaScript 闯关记》之 DOM（上）

javascript 劼哥stone 2 天前发布

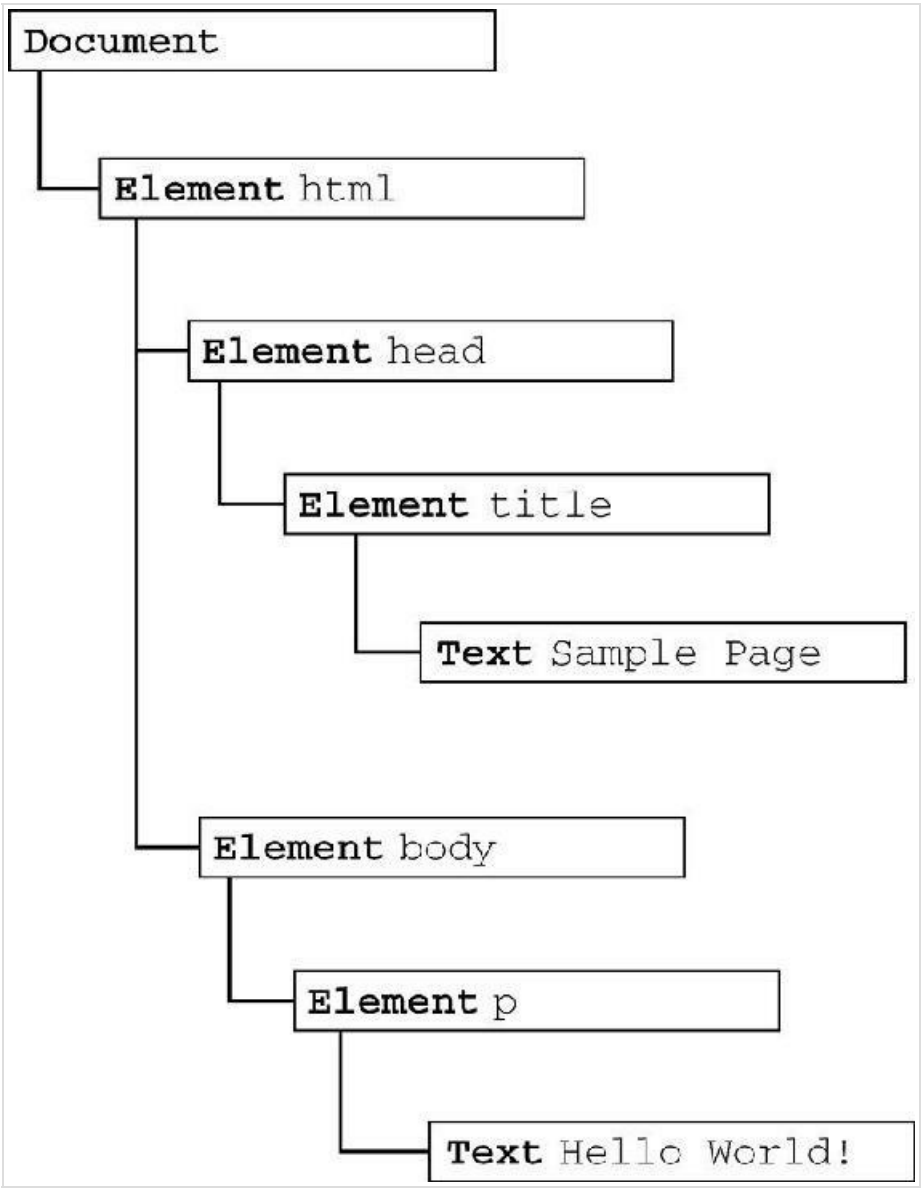
DOM（文档对象模型）是针对 HTML 和 XML 文档的一个 API。DOM 描绘了一个层次化的节点树，允许开发人员添加、移除和修改页面的某一部分。

节点层次

DOM 可以将任何 HTML 或 XML 文档描绘成一个由多层节点构成的结构。节点分为几种不同的类型，每种类型分别表示文档中不同的信息及（或）标记。每个节点都拥有各自的特点、数据和方法，另外也与其他节点存在某种关系。节点之间的关系构成了层次，而所有页面标记则表现为一个以特定节点为根节点的树形结构。以下面的 HTML 为例：

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

可以将这个简单的 HTML 文档表示为一个层次结构，如图下图所示。



在这个例子中，文档元素是文档的最外层元素，文档中的其他所有元素都包含在文档元素中。每个文档只能有一个文档元素。

每一段标记都可以通过树中的一个节点来表示：HTML 元素通过元素节点表示，特性（attribute）通过特性节点表示，文档类型通过文档类型节点表示，而注释则通过注释节点表示。总共有12种节点类型，这些类型都继承自一个基类型。

Node 类型

DOM1 级定义了一个 `Node` 接口，该接口将由 DOM 中的所有节点类型实现。这个 `Node` 接口在 JavaScript 中是作为 `Node` 类型实现的；除了 IE 之外，在其他所有浏览器中都可以访问到这个类型。JavaScript 中的所有节点类型都继承自 `Node` 类型，因此所有节点类型都共享着相同的基本属性和方法。

每个节点都有一个 `nodeType` 属性，用于表明节点的类型。节点类型由在 `Node` 类型中定义的下列12个数值常量来表示，任何节点类型必居其一：

- `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)。

通过比较上面这些常量，可以很容易地确定节点的类型，例如：

```
if (someNode.nodeType == Node.ELEMENT_NODE){    // 在IE中无效
    console.log("Node is an element.");
}
```

这个例子比较了 `someNode.nodeType` 与 `Node.ELEMENT_NODE` 常量。如果二者相等，则意味着 `someNode` 确实是一个元素。然而，由于 IE 没有公开 `Node` 类型的构造函数，因此上面的代码在 IE 中会导致错误。为了确保跨浏览器兼容，最好还是将 `nodeType` 属性与数字值进行比较，如下所示：

```
if (someNode.nodeType == 1){    // 适用于所有浏览器
    console.log("Node is an element.");
}
```

并不是所有节点类型都受到 Web 浏览器的支持。开发人员最常用的就是元素和文本节点。

Node 属性概述

Node 常用属性主要有以下10个，接下来我们会着重讲解部分属性。

- `nodeType`：显示节点的类型
- `nodeName`：显示节点的名称
- `nodeValue`：显示节点的值
- `attributes`：获取一个属性节点
- `firstChild`：表示某一节点的第一个节点
- `lastChild`：表示某一节点的最后一个子节点
- `childNodes`：表示所在节点的所有子节点
- `parentNode`：表示所在节点的父节点
- `nextSibling`：紧挨着当前节点的下一个节点
- `previousSibling`：紧挨着当前节点的上一个节点

`nodeName` 和 `nodeValue` 属性

要了解节点的具体信息，可以使用 `nodeName` 和 `nodeValue` 这两个属性。这两个属性的值完全取决于节点的类型。在使用这两个值以前，最好是像下面这样先检测一下节点的类型。

```
if (someNode.nodeType == 1){
    value = someNode.nodeName;    // nodeName的值是元素的标签名
}
```

在这个例子中，首先检查节点类型，看它是不是一个元素。如果是，则取得并保存 `nodeName` 的值。对于元素节点，`nodeName` 中保存的始终都是元素的标签名，而 `nodeValue` 的值则始终为 `null`。

节点关系

文档中所有的节点之间都存在这样或那样的关系。节点间的各种关系可以用传统的家族关系来描述，相当于把文档树比喻成家谱。

每个节点都有一个 `childNodes` 属性，其中保存着一个 `NodeList` 对象。`NodeList` 是一种类数组对象，用于保存一组有序的节点，可以通过位置来访问这些节点。请注意，虽然可以通过方括号语法来访问 `NodeList` 的值，而且这个对象也有 `length` 属性，但它并不是 `Array` 的实例。`NodeList` 对象的独特之处在于，它实际上是基于 DOM 结构动态执行查询的结果，因此 DOM 结构的变化能够自动反映在 `NodeList` 对象中。

下面的例子展示了如何访问保存在 `NodeList` 中的节点——可以通过方括号，也可以使用 `item()` 方法。

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

无论使用方括号还是使用 `item()` 方法都没有问题，但使用方括号语法看起来与访问数组相似，因此颇受一些开发人员的青睐。另外，要注意

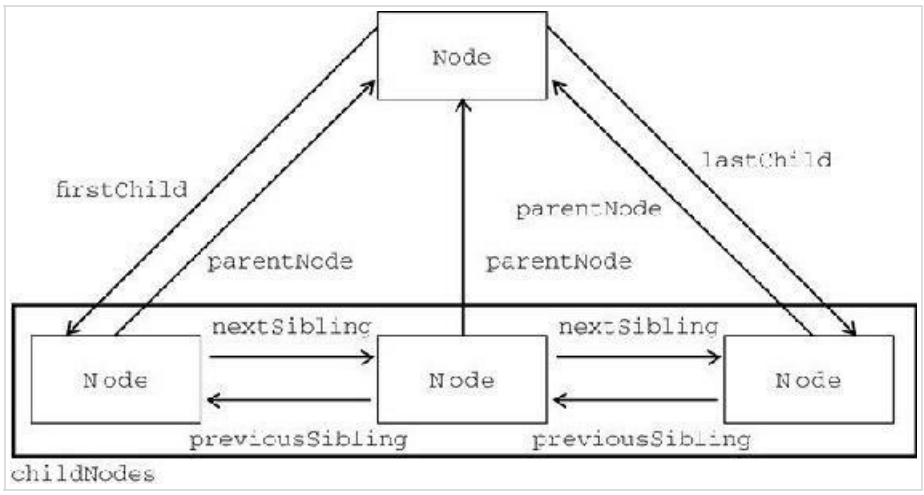
`length` 属性表示的是访问 `NodeList` 的那一刻，其中包含的节点数量。

每个节点都有一个 `parentNode` 属性，该属性指向文档树中的父节点。包含在 `childNodes` 列表中的所有节点都具有相同的父节点，因此它们的 `parentNode` 属性都指向同一个节点。此外，包含在 `childNodes` 列表中的每个节点相互之间都是同胞节点。通过使用列表中每个节点的 `previousSibling` 和 `nextSibling` 属性，可以访问同一列表中的其他节点。列表中第一个节点的 `previousSibling` 属性值为 `null`，而列表中最后一个节点的 `nextSibling` 属性的值同样也为 `null`，如下面的例子所示：

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

当然，如果列表中只有一个节点，那么该节点的 `nextSibling` 和 `previousSibling` 都为 `null`。

父节点与其第一个和最后一个子节点之间也存在特殊关系。父节点的 `firstChild` 和 `lastChild` 属性分别指向其 `childNodes` 列表中的第一个和最后一个节点。其中，`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]`。在只有一个子节点的情况下， `firstChild` 和 `lastChild` 指向同一个节点。如果没有子节点，那么 `firstChild` 和 `lastChild` 的值均为 `null`。明确这些关系能够对我们查找和访问文档结构中的节点提供极大的便利。下图形象地展示了上述关系。



在反映这些关系的所有属性当中，`childNodes` 属性与其他属性相比更方便一些，因为只须使用简单的关系指针，就可以通过它访问文档树中的任何节点。另外，`hasChildNodes()` 也是一个非常有用的方法，这个方法在节点包含一或多个子节点的情况下返回 `true`；应该说，这是比查询 `childNodes` 列表的 `length` 属性更简单的方法。

所有节点都有的最后一个属性是 `ownerDocument`，该属性指向表示整个文档的文档节点。这种关系表示的是任何节点都属于它所在的文档，任何节点都不能同时存在于两个或更多个文档中。通过这个属性，我们可以不必在节点层次中通过层层回溯到达顶端，而是可以直接访问文档节点。

操作节点

因为关系指针都是只读的，所以 DOM 提供了一些操作节点的方法。其中，最常用的方法是 `appendChild()`，用于向 `childNodes` 列表的末尾添加一个节点。添加节点后，`childNodes` 的新增节点、父节点及以前的最后一个子节点的关系指针都会相应地得到更新。更新完成后，`appendChild()` 返回新增的节点。来看下面的例子：

```
var returnedNode = someNode.appendChild(newNode);
console.log(returnedNode == newNode);         // true
console.log(someNode.lastChild == newNode);   // true
```

如果传入到 `appendChild()` 中的节点已经是文档的一部分了，那结果就是将该节点从原来的位置转移到新位置。即使可以将 DOM 树看成是由一系列指针连接起来的，但任何 DOM 节点也不能同时出现在文档中的多个位置上。因此，如果在调用 `appendChild()` 时传入了父节点的第一个子节点，那么该节点就会成为父节点的最后一个子节点，如下面的例子所示。

```
// someNode 有多个子节点
var returnedNode = someNode.appendChild(someNode.firstChild);
console.log(returnedNode == someNode.firstChild);   // false
console.log(returnedNode == someNode.lastChild);    // true
```

如果需要把节点放在 `childNodes` 列表中某个特定的位置上，而不是放在末尾，那么可以使用 `insertBefore()` 方法。这个方法接受两个参数：要插入的节点和作为参照的节点。插入节点后，被插入的节点会变成参照节点的前一个同胞节点 `previousSibling`，同时被方法返回。如果参照节点是 `null`，则 `insertBefore()` 与 `appendChild()` 执行相同的操作，如下面的例子所示。

```
// 插入后成为最后一个子节点
returnedNode = someNode.insertBefore(newNode, null);
console.log(newNode == someNode.lastChild);   // true

// 插入后成为第一个子节点
var returnedNode = someNode.insertBefore(newNode, someNode.firstChild);
console.log(returnedNode == newNode);         // true
console.log(newNode == someNode.firstChild);  // true

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

前面介绍的 `appendChild()` 和 `insertBefore()` 方法都只插入节点，不会移除节点。而下面要介绍的 `replaceChild()` 方法接受的两个参数是：要插入的节点和要替换的节点。要替换的节点将由这个方法返回并从文档树中被移除，同时由要插入的节点占据其位置。来看下面的例子。

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

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

在使用 `replaceChild()` 插入一个节点时，该节点的所有关系指针都会从被它替换的节点复制过来。尽管从技术上讲，被替换的节点仍然还在文档中，但它在文档中已经没有了自己的位置。

如果只想移除而非替换节点，可以使用 `removeChild()` 方法。这个方法接受一个参数，即要移除的节点。被移除的节点将成为方法的返回值，如下面的例子所示。

```
// 移除第一个子节点
var formerFirstChild = someNode.removeChild(someNode.firstChild);

// 移除最后一个子节点
var formerLastChild = someNode.removeChild(someNode.lastChild);
```

与使用 `replaceChild()` 方法一样，通过 `removeChild()` 移除的节点仍然为文档所有，只不过在文档中已经没有了自己的位置。

前面介绍的四个方法操作的都是某个节点的子节点，也就是说，要使用这几个方法必须先取得父节点（使用 `parentNode` 属性）。另外，并不是所有类型的节点都有子节点，如果在不支持子节点的节点上调用了这些方法，将会导致错误发生。

Document 类型

JavaScript 通过 Document 类型表示文档。在浏览器中，`document` 对象是 `HTMLDocument`（继承自 Document 类型）的一个实例，表示整个 HTML 页面。而且，`document` 对象是 `window` 对象的一个属性，因此可以将其作为全局对象来访问。`Document` 节点具有下列特征：

- `nodeType` 的值为9；
- `nodeName` 的值为 `"#document"`；
- `nodeValue` 的值为 `null`；
- `parentNode` 的值为 `null`；
- `ownerDocument` 的值为 `null`；
- 其子节点可能是一个 `DocumentType`（最多一个）、`Element`（最多一个）、`ProcessingInstruction` 或 `Comment`。

`Document` 类型可以表示 HTML 页面或者其他基于 XML 的文档。不过，最常见的应用还是作为 `HTMLDocument` 实例的 `document` 对象。通过这个文档对象，不仅可以取得与页面有关的信息，而且还能操作页面的外观及其底层结构。

文档的子节点

虽然 DOM 标准规定 `Document` 节点的子节点可以是 `DocumentType`、`Element`、`ProcessingInstruction` 或 `Comment`，但还有两个内置的访问其子节点的快捷方式。第一个就是 `documentElement` 属性，该属性始终指向 HTML 页面中的 `html` 元素。另一个就是通过 `childNodes` 列表访问文档元素，但通过 `documentElement` 属性则能更快捷、更直接地访问该元素。以下面这个简单的页面为例。

```
<html>
    <body>
    </body>
</html>
```

这个页面在经过浏览器解析后，其文档中只包含一个子节点，即 `html` 元素。可以通过 `documentElement` 或 `childNodes` 列表来访问这个元素，如下所示。

```
var html = document.documentElement;      // 取得对<html>的引用
console.log(html === document.childNodes[0]);   // true
console.log(html === document.firstChild);      // true
```

这个例子说明，`documentElement`、`firstChild` 和 `childNodes[0]` 的值相同，都指向 `<html>` 元素。

作为 `HTMLDocument` 的实例，`document` 对象还有一个 `body` 属性，直接指向 `<body>` 元素。因为开发人员经常要使用这个元素，所以 `document.body` 在 JavaScript 代码中出现的频率非常高，其用法如下。

```
var body = document.body; // 取得对<body>的引用
```

所有浏览器都支持 `document.documentElement` 和 `document.body` 属性。

`Document` 另一个可能的子节点是 `DocumentType`。通常将 `<!DOCTYPE>` 标签看成一个与文档其他部分不同的实体，可以通过 `doctype` 属性（在浏览器中是 `document.doctype` ）来访问它的信息。

```
var doctype = document.doctype; // 取得对<!DOCTYPE>的引用
```

浏览器对 `document.doctype` 的支持差别很大，可以给出如下总结。

- IE8 及之前版本：如果存在文档类型声明，会将其错误地解释为一个注释并把它当作 `Comment` 节点；而 `document.doctype` 的值始终为 `null`。
- IE9+ 及 Firefox：如果存在文档类型声明，则将其作为文档的第一个子节点；`document.doctype` 是一个 `DocumentType` 节点，也可以通过 `document.firstChild` 或 `document.childNodes[0]` 访问同一个节点。
- Safari、Chrome 和 Opera：如果存在文档类型声明，则将其解析，但不作为文档的子节点。`document.doctype` 是一个 `DocumentType` 节点，但该节点不会出现在 `document.childNodes` 中。

由于浏览器对 `document.doctype` 的支持不一致，因此这个属性的用处很有限。

文档信息

作为 `HTMLDocument` 的一个实例，`document` 对象还有一些标准的 `Document` 对象所没有的属性。这些属性提供了 `document` 对象所表现的网页的一些信息。其中第一个属性就是 `title`，包含着 `<title>` 元素中的文本——显示在浏览器窗口的标题栏或标签页上。通过这个属性可以取得当前页面的标题，也可以修改当前页面的标题并反映在浏览器的标题栏中。

```
// 取得文档标题
var originalTitle = document.title;

// 设置文档标题
document.title = "New page title";
```

接下来要介绍的3个属性都与对网页的请求有关，它们是 `URL`、`domain` 和 `referrer`。`URL` 属性中包含页面完整的 URL（即地址栏中显示的 URL），`domain` 属性中只包含页面的域名，而 `referrer` 属性中则保存着链接到当前页面的那个页面的 URL。在没有来源页面的情况下，`referrer` 属性中可能会包含空字符串。所有这些信息都存在于请求的 HTTP 头部，只不过是通过这些属性让我们能够在 JavaScrip 中访问它们而已，如下面的例子所示。

```
// 取得完整的URL
var url = document.URL;

// 取得域名
var domain = document.domain;

// 取得来源页面的URL
var referrer = document.referrer;
```

查找元素

说到最常见的 DOM 应用，恐怕就要数取得特定的某个或某组元素的引用，然后再执行一些操作了。取得元素的操作可以使用 `document` 对象的几个方法来完成。其中， `Document` 类型为此提供了两个方法： `getElementById()` 和 `getElementsByTagName()` 。

第一个方法， `getElementById()` ，接收一个参数：要取得的元素的 ID。如果找到相应的元素则返回该元素，如果不存在带有相应 ID 的元素，则返回 `null` 。注意，这里的 ID 必须与页面中元素的 `id` 特性（attribute）严格匹配，包括大小写。以下面的元素为例。

```
<div id="myDiv">Some text</div>
```

可以使用下面的代码取得这个元素：

```
var div = document.getElementById("myDiv"); // 取得<div>元素的引用
```

但是，下面的代码在除 IE7 及更早版本之外的所有浏览器中都将返回 `null` 。

```
var div = document.getElementById("mydiv"); // 无效的ID（在IE7及更早版本中可以）
```

IE8 及较低版本不区分 ID 的大小写，因此 `"myDiv"` 和 `"mydiv"` 会被当作相同的元素 ID。如果页面中多个元素的ID值相同， `getElementById()` 只返回文档中第一次出现的元素。

另一个常用于取得元素引用的方法是 `getElementsByTagName()` 。这个方法接受一个参数，即要取得元素的标签名，而返回的是包含零或多个元素的 `NodeList` 。在HTML文档中，这个方法会返回一个 `HTMLCollection` 对象，作为一个“动态”集合，该对象与 `NodeList` 非常类似。例如，下列代码会取得页面中所有的 `<img>` 元素，并返回一个 `HTMLCollection` 。

```
var images = document.getElementsByTagName("img");
```

这行代码会将一个 `HTMLCollection` 对象保存在 `images` 变量中。与 `NodeList` 对象类似，可以使用方括号语法或 `item()` 方法来访问 `HTMLCollection` 对象中的项。而这个对象中元素的数量则可以通过其 `length` 属性取得，如下面的例子所示。

```
console.log(images.length);         // 输出图像的数量
console.log(images[0].src);         // 输出第一个图像元素的src特性
console.log(images.item(0).src);    // 输出第一个图像元素的src特性
```

`HTMLCollection` 对象还有一个方法，叫做 `namedItem()` ，使用这个方法可以通过元素的 `name` 特性取得集合中的项。例如，假设上面提到的页面中包含如下 `<img>` 元素：

```
<img src="myimage.gif" name="myImage">
```

那么就可以通过如下方式从 `images` 变量中取得这个 `<img>` 元素：

```
var myImage = images.namedItem("myImage");
```

在提供按索引访问项的基础上， `HTMLCollection` 还支持按名称访问项，这就为我们取得实际想要的元素提供了便利。而且，对命名的项也可以使用方括号语法来访问，如下所示：

```
var myImage = images["myImage"];
```

对 `HTMLCollection` 而言，我们可以向方括号中传入数值或字符串形式的索引值。在后台，对数值索引就会调用 `item()`，而对字符串索引就会调用 `namedItem()`。

要想取得文档中的所有元素，可以向 `getElementsByTagName()` 中传入 `"*"`。在 JavaScript 及 CSS 中，星号（`*`）通常表示“全部”。下面看一个例子。

```
var allElements = document.getElementsByTagName("*");
```

仅此一行代码返回的 `HTMLCollection` 中，就包含了整个页面中的所有元素——按照它们出现的先后顺序。换句话说，第一项是 `<html>` 元素，第二项是 `<head>` 元素，以此类推。由于 IE 将注释（`Comment`）实现为元素（`Element`），因此在IE中调用 `getElementsByTagName("*")` 将会返回所有注释节点。

第三个方法，也是只有 `HTMLDocument` 类型才有的方法，是 `getElementsByName()`。顾名思义，这个方法会返回带有给定 `name` 特性的所有元素。最常使用 `getElementsByName()` 方法的情况是取得单选按钮；为了确保发送给浏览器的值正确无误，所有单选按钮必须具有相同的 `name` 特性，如下面的例子所示。

```
<fieldset>
    <legend>Which color do you prefer?</legend>
    <ul>
        <li><input type="radio" value="red" name="color" id="colorRed">
            <label for="colorRed">Red</label></li>
        <li><input type="radio" value="green" name="color" id="colorGreen">
            <label for="colorGreen">Green</label></li>
        <li><input type="radio" value="blue" name="color" id="colorBlue">
            <label for="colorBlue">Blue</label></li>
    </ul>
</fieldset>
```

如这个例子所示，其中所有单选按钮的 `name` 特性值都是 `"color"`，但它们的 ID 可以不同。ID 的作用在于将 `<label>` 元素应用到每个单选按钮，而 `name` 特性则用以确保三个值中只有一个被发送给浏览器。这样，我们就可以使用如下代码取得所有单选按钮：

```
var radios = document.getElementsByName("color");
```

与 `getElementsByTagName()` 类似，`getElementsByName()` 方法也会返回一个 `HTMLCollectioin`。但是，对于这里的单选按钮来说，`namedItem()` 方法则只会取得第一项（因为每一项的 `name` 特性都相同）。

特殊集合

除了属性和方法，`document` 对象还有一些特殊的集合。这些集合都是 `HTMLCollection` 对象，为访问文档常用的部分提供了快捷方式，包括：

- `document.anchors`，包含文档中所有带 `name` 特性的 `<a>` 元素；
- `document.applets`，包含文档中所有的 `<applet>` 元素，因为不再推荐使用 `<applet>` 元素，所以这个集合已经不建议使用了；
- `document.forms`，包含文档中所有的 `<form>` 元素，与 `document.getElementsByTagName("form")` 得到的结果相同；
- `document.images`，包含文档中所有的 `<img>` 元素，与 `document.getElementsByTagName("img")` 得到的结果相同；
- `document.links`，包含文档中所有带 `href` 特性的 `<a>` 元素。

这个特殊集合始终都可以通过 `HTMLDocument` 对象访问到，而且，与 `HTMLCollection` 对象类似，集合中的项也会随着当前文档内容的更新而更新。

文档写入

有一个 `document` 对象的功能已经存在很多年了，那就是将输出流写入到网页中的能力。这个能力体现在下列4个方法中：`write()`、`writeln()`、`open()` 和 `close()`。其中，`write()` 和 `writeln()` 方法都接受一个字符串参数，即要写入到输出流中的文本。`write()` 会原样写入，而 `writeln()` 则会在字符串的末尾添加一个换行符 `n`。在页面被加载的过程中，可以使用这两个方法向页面中动态地加入内容，如下面的例子所示。

```
<html>
<head>
    <title>document.write() Example</title>
</head>
<body>
    <p>The current date and time is:
    <script type="text/javascript">
        document.write("<strong>" + (new Date()).toString() + "</strong>");
    </script>
    </p>
</body>
</html>
```

这个例子展示了在页面加载过程中输出当前日期和时间的代码。其中，日期被包含在一个 `<strong>` 元素中，就像在 HTML 页面中包含普通的文本一样。这样做会创建一个 DOM 元素，而且可以在将来访问该元素。通过 `write()` 和 `writeln()` 输出的任何 HTML 代码都将如此处理。

此外，还可以使用 `write()` 和 `writeln()` 方法动态地包含外部资源，例如 JavaScript 文件等。在包含 JavaScript 文件时，必须注意不能像下面的例子那样直接包含字符串 `"</script>"`，因为这会导致该字符串被解释为脚本块的结束，它后面的代码将无法执行。

```
<html>
<head>
    <title>document.write() Example 2</title>
</head>
<body>
    <script type="text/javascript">
        document.write("<script type=\"text/javascript\" src=\"file.js\">" +
            "</script>");
    </script>
</body>
</html>
```

即使这个文件看起来没错，但字符串 `"</script>"` 将被解释为与外部的 `<script>` 标签匹配，结果文本 `");` 将会出现在页面中。为避免这个问题，只需把这个字符串分开写即可；第2章也曾经提及这个问题，解决方案如下。

```
<html>
<head>
    <title>document.write() Example 3</title>
</head>
<body>
    <script type="text/javascript">
        document.write("<script type=\"text/javascript\" src=\"file.js\">" +
            "<\/script>");
    </script>
</body>
</html>
```

字符串 `"</script>"` 不会被当作外部 `<script>` 标签的关闭标签，因而页面中也就不会出现多余的内容了。

前面的例子使用 `document.write()` 在页面被呈现的过程中直接向其中输出了内容。如果在文档加载结束后再调用 `document.write()`，那么输出的内容将会重写整个页面，如下面的例子所示：

```
<html>
<head>
    <title>document.write() Example 4</title>
</head>
<body>
    <p>This is some content that you won't get to see because it will be overwritten.</p>
    <script type="text/javascript">
        window.onload = function(){
            document.write("Hello world!");
        };
    </script>
</body>
</html>
```

在这个例子中，我们使用了 `window.onload` 事件处理程序，等到页面完全加载之后延迟执行函数。函数执行之后，字符串 `"Hello world!"` 会重写整个页面内容。

方法 `open()` 和 `close()` 分别用于打开和关闭网页的输出流。如果是在页面加载期间使用 `write()` 或 `writeln()` 方法，则不需要用到这两个方法。

关卡

仔细想想，下面代码块会输出什么结果呢？

```
<!-- 挑战一 -->
<body>
<div id = "t"><span>aaa</span><span>bbb</span><span>ccc</span></div>
</body>
<script>
    var d = document.getElementById("t");
    document.writeln(d.firstChild.innerHTML);  // ???
    document.writeln(d.lastChild.innerHTML);   // ???
</script>
```

```
<!-- 挑战二 -->
<body name="ddd">
<div id = "t"><span>aaa</span><span>bbb</span><span>ccc</span></div>
</body>
<script>
    var d = document.getElementById("t");
    document.writeln(d.childNodes[1].innerHTML); // ???
    document.writeln(d.parentNode.getAttribute("name")); // ???
</script>
```

```
<!-- 挑战三 -->
<body name="ddd">
<div id = "t"><span>aaa</span><span>bbb</span><span>ccc</span></div>
</body>
<script>
    var d = document.getElementById("t").childNodes[1];
    document.writeln(d.nextSibling.innerHTML);      // ???
    document.writeln(d.previousSibling.innerHTML);  // ???
</script>
```

更多

> 关注微信公众号「劼哥舍」回复「答案」，获取关卡详解。
> 关注 https://github.com/stone0090/javascript-lessons，获取最新动态。

2 天前发布　更多

1 推荐　收藏　￥ 赞赏

你可能感兴趣的文章

红皮书：JavaScript简介（一）　9 收藏，1.5k 浏览

Javascript鸡汤　5 收藏，1.8k 浏览

Javascript简介　3 收藏，511 浏览

讨论区

使用评论询问更多信息或提出修改意见，请不要在评论里回答问题

提交评论

评论支持部分 Markdown 语法：`**bold**` `_italic_` `[link](http://example.com)` `> 引用` `` `code` `` `- 列表` 。

同时，被你 @ 的用户也会收到通知

本文隶属于专栏

劼哥舍

欢迎来到「劼哥舍」，您非要念成「劫个色」也行。这里坚持原创，分享随笔和学习心得，主要涉及 前端 / .NET / Java 等方面的内容。欢迎交流，欢迎提问，欢迎转载，但需注明出处。

劼哥stone
作者 关注作者

关注专栏

系列文章

《JavaScript 闯关记》之对象 6 收藏， 164 浏览

《JavaScript 闯关记》之数组 6 收藏， 232 浏览

《JavaScript 闯关记》之函数 3 收藏， 161 浏览

《JavaScript 闯关记》之正则表达式 30 收藏， 808 浏览

《JavaScript 闯关记》之基本包装类型 18 收藏， 862 浏览

《JavaScript 闯关记》之单体内置对象 11 收藏， 419 浏览

《JavaScript 闯关记》之 BOM 14 收藏， 525 浏览

相关收藏夹

换一组

react
4 个条目 | 0 人关注

啦啦啦啦啦啦
8 个条目 | 0 人关注

数据结构和算法
5 个条目 | 2 人关注

分享扩散：