第9章 文档对象模型

在上一章中，我们介绍了浏览器对象模型。而在这一章中，我们将会为大家介绍使用频率更高的文档对象模型。通过掌握文档对象模型的相关知识，可以让我们很轻松的实现利用JavaScript来动态地修改网页的内容，以及制作出绚丽的网页特效。

本章将学习如下内容：

- DOM基本介绍
- 对节点进行增删改查操作
- 获取和设置节点的属性以及类
- 修改元素的CSS样式
- 利用DOM操作表格

9-1 DOM基本介绍

9-1-1 什么是DOM

首先，我们需要介绍什么是DOM。DOM的英语全称为Document Object Model，翻译成中文就是文档对象模型。也就是说，将整个文档看作是一个对象。而一个文档又是由很多节点组成的，那么这些节点也可以被看作是一个个的对象。DOM里面的对象属于宿主对象，需要浏览器来作为宿主。一旦离开了浏览器这个环境，那么该对象将不复存在。同样，上一章我们所介绍的BOM也是如此，需要浏览器来作为宿主，所以它也是一个宿主对象。

DOM的作用如下：

- 浏览器提供的操纵HTML文档内容的应用程序接口
- 用于对文档进行动态操作，如增加文档内容，删除文档内容，修改文档内容等等

9-1-2 DOM历史

在介绍了什么是DOM之后，接下来我们来看一下DOM的一个发展史。而一说到DOM的发展史，那就不得不介绍DOM的级别。这里我们对DOM的级别来进行一个简单的介绍，如下：

DOM Level 0：首先，我们需要确定的是在DOM标准中并没有`DOM0`级这个级别。所谓的`DOM0`级是DOM历史坐标中的一个参照点而已，怎么说呢，`DOM0`级指的是IE4和Netscape 4.0这些浏览器最初支持的DOM相关方法。主要关注于常见的页面元素，比如图像，链接和表单。有些现在图像和表单的那些方法，目前依然可以被用在当前版本的DOM中。

DOM Level 1：于1998年10月成为W3C的推荐标准。`DOM1`级由两个模块组成：DOM核心(DOM Core)和DOM HTML。这个版本引入了网页的完整模型，允许在网页的每个部分进行导航。

DOM Level 2：对DOM level 1做了扩展，于2000年出版，引入了流行的`getElementById()`方法，让访问网页上的特定元素变得更加容易。

DOM Level 3：对DOM level 2做了进一步的扩展，于2004年出版。

9-1-3 节点类型与节点名称

一个文档是由大量的节点所构成的。而每一个节点都有一个叫做`nodeType`的属性，用于表明节点的类型。不同的节点类型对应了不同的数值，具体对应的数值如下表：

节点名称	对应数值
元素节点	Node.ELEMENT_NODE(1)
属性节点	Node.ATTRIBUTE_NODE(2)
文本节点	Node.TEXT_NODE(3)
CDATA节点	Node.CDATA_SECTION_NODE(4)
实体引用名称节点	Node.ENTRY_REFERENCE_NODE(5)
实体名称节点	Node.ENTITY_NODE(6)
处理指令节点	Node.PROCESSING_INSTRUCTION_NODE(7)
注释节点	Node.COMMENT_NODE(8)
文档节点	Node.DOCUMENT_NODE(9)
文档类型节点	Node.DOCUMENT_TYPE_NODE(10)
文档片段节点	Node.DOCUMENT_FRAGMENT_NODE(11)
DTD声明节点	Node.NOTATION_NODE(12)

从上面我们可以看出，不同的节点对应了不同的节点类型，我们可以通过`nodeType`属性来获取到该节点的节点类型，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
```

```
    <script>
        let test1 = document.getElementById("test1");
        let content = test1.firstChild;
        let attr = test1.getAttributeNode("class");
        console.log(test1.nodeType);//1 获取元素节点的节点类型
        console.log(content.nodeType);//3 获取文本节点的节点类型
        console.log(attr.nodeType);//2 获取属性节点的节点类型
        console.log(document.nodeType);//9 获取整个文档的节点类型
    </script>
</body>
```

`nodeType`属性可以和if配合使用，确保不会在错误的节点类型上执行错误的操作，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        if(test1.nodeType == 1)//如果是元素节点
        {
            //设置该节点的颜色为红色
            test1.style.color = "red";
        }
    </script>
</body>
```

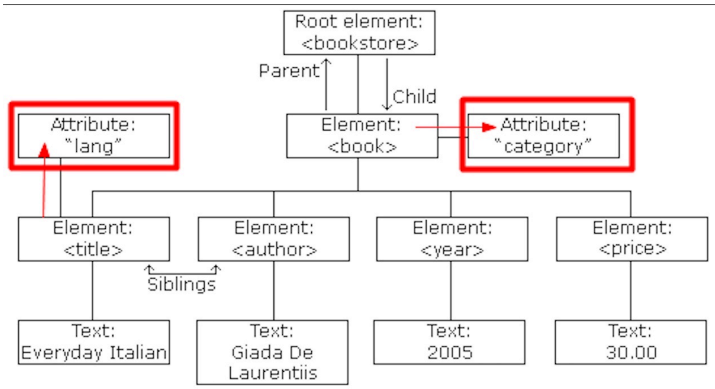
效果：

Lorem ipsum dolor sit amet.

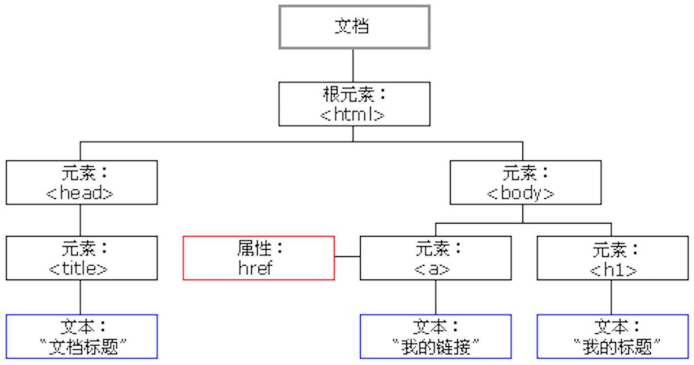
除了获取节点类型以外，我们还可以通过`nodeName`属性来获取节点的名称，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        let content = test1.firstChild;
        let attr = test1.getAttributeNode("class");
        console.log(test1.nodeName);//P
        console.log(content.nodeName);//#text
        console.log(attr.nodeName);//class
        console.log(document.nodeName);//#document
    </script>
</body>
```

还有一点需要说明的是，属性节点并不是元素节点的子节点，只能算作是元素节点的一个附属节点，如下图：



http://www.w3schools.com/



http://www.w3school.com.cn/

这里我们通过一段代码来证明上面说的这一点，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        console.log(test1.childNodes);//打印出该节点的所有子节点
    </script>
</body>
```

效果：

```
▼ NodeList(1) i
  ▶ 0: text
    length: 1
  ▶ __proto__: NodeList
```

可以看到，当我们查看 `<p>` 元素下面的子节点时，打印出来的子节点只有一个文本节点，通过这个例子就可以证明属性节点并不是元素节点的子节点，只能算作是元素节点的一个附属节点。

9-2 旧的DOM用法

接下来，我们来看一下来自于`DOM0`级的旧的DOM用法。前面我们在介绍DOM历史的时候有提到过，`DOM0`级主要关注于页面上常见的元素，比如图像，链接和表单等。这些方法目前依然可以被用在当前版本的DOM中。

9-2-1 document.body

返回网页的`<body>`元素
示例如下:我在我的HTML文件里面书写如下的代码

```
<body>
    <script>
        let i = document.body;
        console.log('this is ',i);
    </script>
</body>
```

效果：

```
this is  ▶<body>…</body>
```

可以看到，这里我们确实就引用到了文档的body元素

9-2-2 document.images

返回文档中所有图像的一个节点列表

```
<body>
    <img src="" alt="图片1">
    <img src="" alt="图片2">
    <img src="" alt="图片3">
    <img src="" alt="图片4">
    <img src="" alt="图片5">
    <script>
        let i = document.images;
        console.log(i);
    </script>
</body>
```

效果：

```
▶ (5) [img, img, img, img, img]
```

可以看到这里关于页面里面所有的图片就都出来了。这是一个节点列表，所以我们可以像使用数组一样来取得每一个图片元素

```
<body>
    <img src="" alt="图片1">
    <img src="" alt="图片2">
    <img src="" alt="图片3">
    <img src="" alt="图片4">
    <img src="" alt="图片5">
    <script>
        let i = document.images;
        console.log(i);//获取页面所有的图片元素
        console.log(i[0]);//引用第一个图片元素
    </script>
</body>
```

效果：

```
▶ (5) [img, img, img, img, img]
  <img src(unknown) alt="图片1">
```

9-2-3 document.links

返回所有具有href属性的 `<a>` 元素和 `<area>` 元素的一个节点列表

```
<body>
    <a href="">链接1</a>
    <a href="">链接2</a>
    <a href="">链接3</a>
    <a href="">链接4</a>
    <a href="">链接5</a>
    <script>
        let i = document.links;
        console.log(i);//获取页面所有的链接元素
        console.log(i[0]);//获取第一个链接元素
    </script>
</body>
```

效果：

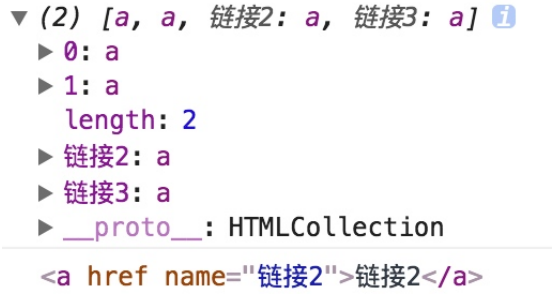
```
▶ (5) [a, a, a, a, a]
  <a href>链接1</a>
```

9-2-4 document.anchors

返回所有具有name属性的`<a>`元素的一个节点列表
示例如下：

```
<body>
    <a href="">链接1</a>
    <a href="" name="链接2">链接2</a>
    <a href="" name="链接3">链接3</a>
    <a href="">链接4</a>
    <a href="">链接5</a>
    <script>
        let i = document.anchors;
        console.log(i);//只会获取带有name属性的a元素
        console.log(i[0]);
    </script>
</body>
```

效果：



9-2-5 document.forms

返回文档中所有表单的一个节点列表
示例如下：

```
<body>
    <form action="" name="form1"></form>
    <form action="" name="form2"></form>
    <script>
        let i = document.forms;
        console.log(i);
        console.log(i[0]);
    </script>
</body>
```

效果：

```
▼ (2) [form, form, form1: form, form2: form]
  ▶ 0: form
  ▶ 1: form
  ▶ form1: form
  ▶ form2: form
    length: 2
  ▶ __proto__: HTMLCollection
```

```
<form action name="form1"></form>
```

9-3 快速查找节点

从DOM2级开始引入了`getElementById()`等可以快速查找节点的方法，让我们查找特定的元素变得非常的容易，在这一节我们就来介绍一下这些快速查找某一个节点的方式。

9-3-1 getElementById()

这个方法到现在都还是非常流行的，通过这个方法可以快速锁定id为某个值的节点，示例如下：

```
<body>
    <p id="test">Lorem ipsum dolor sit amet.</p>
    <a href="">链接1</a>
    <script>
        //快速找到id为test的节点元素
        let i = document.getElementById("test");
        console.log(i);
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

9-3-2 getElementsByTagName()

除了上面的通过节点的id来查找节点元素以外，还可以通过标签的名称来快速查找节点，不过通过标签名来查找节点的方式得到的是一个节点列表，我们需要通过类似于数组的方式才能定位到具体的某一个节点。

```
<body>
    <p id="test1">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="">链接1</a>
    <script>
        let i = document.getElementsByTagName("p");
        console.log(i);//获取所有的p元素
        console.log(i[0]);//获取元素列表的第一个p元素
    </script>
</body>
```

效果：

```
▼ (2) [p#test1, p#test2, test1: p#test1, test2: p#test2]
  ▶ 0: p#test1
  ▶ 1: p#test2
    length: 2
  ▶ test1: p#test1
  ▶ test2: p#test2
  ▶ __proto__: HTMLCollection
<p id="test1">Lorem ipsum dolor sit amet.</p>
```

9-3-3 getElementsByClassName()

className，顾名思义，就是通过类名来查找到元素。而我们知道，类名也是可以有相同的，所以通过这种方式返回的也会是一个元素列表，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <script>
        let i = document.getElementsByClassName("abc");
        console.log(i);//获取类名为abc的元素
        console.log(i[0]);//获取类名为abc的第一个元素
    </script>
</body>
```

效果：

```
▼ (2) [p#test1.abc, a.abc, test1: p#test1.abc]
  ▶ 0: p#test1.abc
  ▶ 1: a.abc
    length: 2
  ▶ test1: p#test1.abc
  ▶ __proto__: HTMLCollection
<p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
```

9-3-4 document.getElementsByName()

使用这个方法可以访问到对应name值的元素节点。因为节点允许有相同的name值，所以这个方法返回的也会是一个节点列表。示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2" name="qwe">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
```

```
        <li name="qwe">item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementsByName("qwe");
        console.log(i);
    </script>
</body>
```

效果：

```
▼ (2) [p#test2, li]
  ▶ 0: p#test2
  ▶ 1: li
    length: 2
  ▶ __proto__: NodeList
```

9-3-5 document.querySelector()

这是HTML5新增的查找节点方法，该方法最大的特点在于可以通过CSS的语法来查找文档中所匹配的第一个元素，注意，只是第一个元素！示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <script>
        let i = document.querySelector(".abc");
        console.log(i);
    </script>
</body>
```

效果：

```
<p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
```

可以看到这里就找到了符合要求的第一个元素

9-3-6 document.querySelectorAll()

这个方法相比上面的方法多了一个All的标志符，我们可以猜想，这就是会返回所有符合要求的元素，同样还是使用CSS的语法来进行查找，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
```

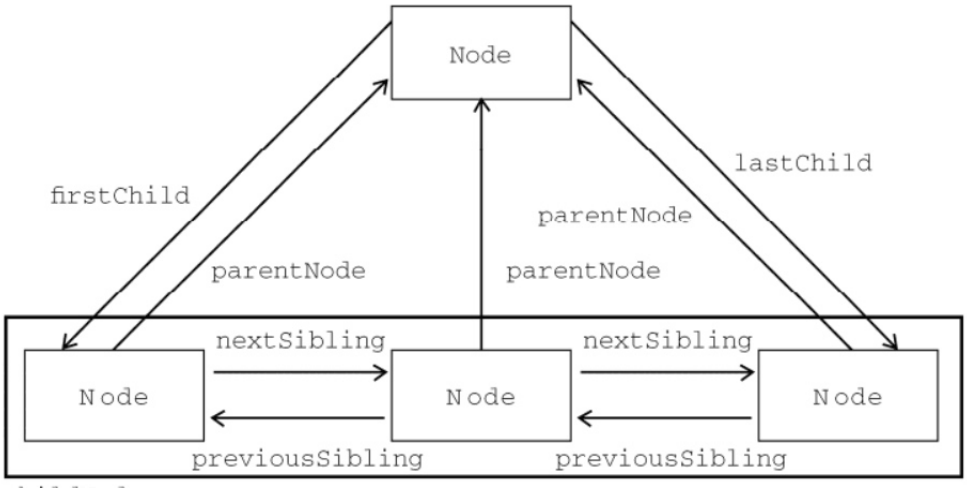
```
    <a href="" class="abc">链接1</a>
    <script>
        let i = document.querySelectorAll(".abc");
        console.log(i);
        console.log(i[0]);
    </script>
</body>
```

效果：

可以看到，这里返回的就是一个节点列表，我们仍然可以通过类似数组的形式来精确访问到某一个具体的节点。

9-4 关系查找节点

在DOM2级之前，还没有快速查找节点的方式，这个时候只有通过节点之间的关系来对节点进行查找。文档中所有节点之间都存在这样那样的关系。节点间的各种关系可以用传统的家族关系来描述，类似于一个家谱图。节点与节点之间的关系如下图所示：



9-4-1 childNodes属性

每个节点都有一个`childNodes`属性，其中保存着一个`NodeList`的类数组对象。该对象包含了该节点下面所有的子节点。`NodeList`对象是自动变化的。这里让我们先来看一下这个`childNodes`属性，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        //获取ul节点
        let i = document.getElementById("test3")
        //输出ul节点的childNodes属性
        console.log(i.childNodes);
    </script>
</body>
```

效果：

```
▼ (7) [text, li, text, li, text, li, text]
  ▶ 0: text
  ▶ 1: li
  ▶ 2: text
  ▶ 3: li
  ▶ 4: text
  ▶ 5: li
  ▶ 6: text
    length: 7
  ▶ __proto__: NodeList
```

可以看到，`<ul>` 的 `childNodes` 属性是该节点的所有子节点的一个列表。有人估计会觉得奇怪，为什么明明 `<ul>` 下面只有3个 `<li>`，但是长度显示的却是7。实际上这里所指的子节点不仅仅只是指的下面的元素节点，还包括了文本节点，空白节点等，都一概被含入了进去。

9-4-2 children属性

`children` 属性只返回一个节点下面的所有子元素节点，所以会忽略所有的文本节点和空白节点，我们还是以上面的代码来举例，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test3")
        console.log(i.children);//获取下面的所有元素节点
    </script>
</body>
```

效果：

```
▼ (3) [li, li, li]
  ▶ 0: li
  ▶ 1: li
  ▶ 2: li
    length: 3
  ▶ __proto__: HTMLCollection
```

可以看到，这次获取到的就只是元素节点，长度为3

9-4-3 firstChild 和 lastChild

接下来我们首先要介绍的是这两个属性，分别是访问一个节点的第一个子节点以及最后一个节

点，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test3");
        console.log(i.firstChild);
    </script>
</body>
```

效果：

▶ #text

可以看到这里就访问到了 `<ul>` 下面的第一个子节点，但是其实这个节点是一个空白节点，什么意思呢？就是说在DOM里面会将空格和换行也视为是一个节点。这样的节点叫做空白节点。如果我现在将 `<ul>` 元素和 `<li>` 元素之间的空白给删除掉，那么第一个子元素就应该为 `<ul>` 下面的第一个 `<li>`，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3"><li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test3");
        console.log(i.firstChild);
    </script>
</body>
```

效果：

item1

这时 `<ul>` 下面的第一个子元素就变为了第一个 `<li>` 元素。

`lastChild` 基本上就和刚才的 `firstChild` 相反，获取的是子节点里面的最后一个节点，示例

如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test3");
        console.log(i.lastChild);
    </script>
</body>
```

效果：

▶ #text

这里也是获取到的是一个空白节点。

9-4-4 parentNode

从英文的意思我们就可以知道，这就是获取父级节点的属性，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test3");
        console.log(i.parentNode);
    </script>
</body>
```

效果：

▶ <body>…</body>

这里就访问到了 `<ul>` 节点的父节点 `<body>` 元素。

9-4-5 previousSibling 和 nextSibling

`previousSiblings` 属性返回同一父节点下的前一个相邻节点。如果该节点已经是父节点的第一个节点，则返回 `null`，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test3");
        console.log(i);
        console.log(i.previousSibling.previousSibling);
    </script>
</body>
```

效果：

```
▶ <ul id="test3">…</ul>
  <a href class="abc">链接1</a>
```

连续写两次 `previousSiblings`，可以跳过第一个空白节点，定位到 `<a>` 节点。

接下来我们来看一下 `nextSibling`， `nextSibling` 属性返回同一父节点的下一个相邻节点。如果节点是父节点的最后一个节点，则返回 `null`，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test1");
        console.log(i.nextSibling.nextSibling);
    </script>
```

```
</body>
```

效果：这里也是同样，如果只写一个`nextSibling`，那么访问到的是一个空白节点，连续写两个，就跳过了这个空白节点，访问到了下一个元素节点。

```
<p id="test2">Lorem ipsum dolor sit amet.</p>
```

9-4-6 previousElementSibling 和 nextElementSibling

把前后的换行也算作是一个空白节点，这样的处理确实也有太麻烦了。所以，现在添加上了`previousElementSibling`和`nextElementSibling`这两个属性，直接用于查询某一个节点的上一个或者下一个元素节点，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test2");
        console.log(i.previousElementSibling);
    </script>
</body>
```

效果：

```
<p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
```

再来看一个`nextElementSibling`属性的例子

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test2");
```

```
        console.log(i.nextElementSibling);
    </script>
</body>
```

效果：

```
<a href class="abc">链接1</a>
```

9-5 获取节点的值

往往我们在获取到元素节点以后，是想要获取到元素节点里面的值，当然，我们可以通过节点与节点之间的关系来获取到文本节点，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test3");
        console.log(i.firstChild.nextSibling.firstChild);
    </script>
</body>
```

效果：

```
"item1"
```

但是，这样确实也太麻烦了。我们只是想要获取元素节点的值，却需要一层一层的向下找。所以，在DOM里面还为我们提供了以下几个获取节点值的属性。

9-5-1 nodeValue

用于获取一个元素节点的文本值，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test1");
        //p元素节点的子节点(文本节点)的节点值
```

```
        console.log(i.firstChild.nodeValue);
    </script>
</body>
```

效果：

```
Lorem ipsum dolor sit amet.
```

需要注意的是nodeValue是获取节点的值，所以还是需要先定位到元素节点下面的文本节点。

9-5-2 textContent

好吧，我不得不承认，上面的方法还是很复杂。既然都需要我们定位到文本节点了，那么剩下的就只是 `firstChild` 和 `nodeValue` 书写不同而已罢了。
接下来我们要介绍的这个 `textContent` ，可以直接取出元素节点的文本值，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test1");
        console.log(i.textContent);
    </script>
</body>
```

效果：

```
Lorem ipsum dolor sit amet.
```

这里我们就直接获取到了元素节点的文本值

9-5-3 innerText 和 innerHTML

实际上在textContent出现之前，获取元素节点最常用的是这两个属性。一个是 `innerText` ，另一个是 `innerHTML` 。那么这两个有什么区别呢？我们来看一个示例就能够明白，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
```

```
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        let i = document.getElementById("test3");
        console.log(i.innerText);
        console.log(i.innerHTML);
    </script>
</body>
```

效果：

```
item1
item2
item3
```

```
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
```

可以看到，区别还是非常明显的，`innerText`用于获取元素节点的文本值，而`innerHTML`则用于用户元素节点下面的所有东西，包括HTML标签。

9-5-4 获取表单节点的值

可以通过`value`属性来获取表单节点的值，示例如下：

```
<body>
    <form action="">
        <input type="text" name="" id="test" value="this is a test">
        <textarea name="" id="test2" cols="30" rows="10">this is a test,too<
/textarea>
        <script>
            console.log(test.value); // this is a test
            console.log(test2.value); // this is a test,too
            // 如果是文本域，还可以通过innerHTML来获取值
            console.log(test2.innerHTML); // this is a test,too
        </script>
    </form>
</body>
```

9-6 节点操作

操作节点其实就有点像操作数据库数据一样，无非就是对节点进行增删改查，当然查就是前面我们所讲的查找节点，接下来我们来看一下对节点的增删改的操作。

9-6-1 创建和添加节点

首先我们来看一下如何创建一个节点。在`document`对象里面存在一个`createElement()`的方法，允许我们来创建一个元素节点，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        //创建一个新的p元素
        let newP = document.createElement("p");
    </script>
</body>
```

通过前面的学习，我们都知道了，元素节点是元素节点，文本节点是文本节点，所以我们这里光有一个`<p>`的元素节点是没什么意义，我们还需要添加文本节点。

在DOM里面，使用`createTextNode()`方法来创建一个文本节点，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        //创建一个新的p元素
        let newP = document.createElement("p");
```

```
        //创建一个文本节点
        let newContent = document.createTextNode("这是一个测试");
    </script>
</body>
```

好，至此我们的文本节点也就已经创建好了。

我们已经知道，一般来讲，一个元素里面有内容的话，那么这个文本节点将会是元素节点的子节点，所以我们现在需要做的事儿，就是将这个文本节点添加到元素节点的后面。

在DOM里面有一个叫做`appendChild()`的方法，可以用于将另一个节点添加为自己的子节点，语法如下：
`父节点.appendChild(子节点)`，我们来试一试：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        //创建一个新的p元素
        let newP = document.createElement("p");
        //创建一个文本节点
        let newContent = document.createTextNode("这是一个测试");
        //将文本节点添加给p元素节点作为子节点
        newP.appendChild(newContent);
    </script>
</body>
```

但是到目前为止，我们页面上还看不到任何的效果。什么原因呢？是因为虽然我们这个有内容的元素节点已经创建好了，但是我们还没有将其添加到我们的文档里面。

怎么将这个新创建好的节点添加到文档里面呢？方法非常简单，还是使用刚才的`appendChild()`方法，将其添加为某一个节点的子节点即可，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
```

```
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        //创建一个新的p元素
        let newP = document.createElement("p");
        //创建一个文本节点
        let newContent = document.createTextNode("这是一个测试");
        //将文本节点添加给p元素节点作为子节点
        newP.appendChild(newContent);
        //获取body节点
        let doc = document.body;
        //为body节点添加子节点
        doc.appendChild(newP);
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet.

链接1

- item1
- item2
- item3

这是一个测试

可以看到，这里我们就已经确实将新创建的`<p>`元素添加到文档里面了。

实际上，如果是要往文档中添加一些内容。上面的方法还是显得非常的笨重。下面介绍一个简便的添加内容方法，直接使用我们上一小节介绍过的innerHTML就可以做到，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
```

```
        let body = document.getElementsByTagName("body")[0];
        body.innerHTML += "<p>这是一个测试</p>";
    </script>
</body>
```

这种方式的效果和上面是一样的，但是代码明显要简单很多。如果是往页面添加少许的内容，推荐使用这种方式。

上面我们已经向大家介绍了`appendChild()`方法了，除了此方法外，关于节点的操作还有一些其他常见的方法。例如`insertBefore()`，`appendChild()`是添加在节点的前面，而这个`insertBefore()`则是添加到节点的后面，语法如下：

`父节点.insertBefore(新节点，旧节点)`

这样就会在旧节点的前面插入新的节点，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        //创建一个新的p元素
        let newP = document.createElement("p");
        //创建一个文本节点
        let newContent = document.createTextNode("测试测试");
        //将文本节点添加给p元素节点作为子节点
        newP.appendChild(newContent);
        //获取id为test2的p元素节点
        let p = document.getElementById("test2");
        //将newP添加到p的前面
        document.body.insertBefore(newP,p);
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

测试测试

Lorem ipsum dolor sit amet.

链接1

- item1
- item2
- item3

可以看到，我们这里就已经将新创建的节点添加到第二个段落的前面去了。

9-6-2 删除节点

既然有添加节点，那么自然也就有删除节点，删除节点的语法如下：
`父节点.remove（子节点）`
语法是非常的简单，接下来我们来试验一下，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2" name="qwe">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li name="qwe">item2</li>
        <li>item3</li>
    </ul>
    <script>
        let test2 = document.getElementById("test2");
        //删除第2个段落
        document.body.removeChild(test2);
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

链接1

- item1
- item2
- item3

可以看到，第二个段落就已经被删除掉了

9-6-3 替换节点

有些时候，我们需要对节点进行替换，这也是可以实现的，语法如下：
`父节点.replaceChild(新节点，旧节点)`
接下来我们来试一试，代码如下：

```
<body>
        <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
        <p id="test2">Lorem ipsum dolor sit amet.</p>
        <a href="" class="abc">链接1</a>
        <ul id="test3">
            <li>item1</li>
            <li>item2</li>
            <li>item3</li>
        </ul>
        <script>
            //创建一个新的p元素
            let newP = document.createElement("p");
            //创建一个文本节点
            let newContent = document.createTextNode("测试测试");
            //将文本节点添加给p元素节点作为子节点
            newP.appendChild(newContent);
            //获取id为test2的p元素节点
            let p = document.getElementById("test2");
            //用newP添来替换第二个p
            document.body.replaceChild(newP,p);
        </script>
    </body>
```

效果：

Lorem ipsum dolor sit amet.

测试测试

链接1

- item1
- item2
- item3

可以看到，这里我们的第二个段落就确实被替换成了新的元素节点

9-6-4 克隆节点

有些时候，我们不想创建和插入一个完全新的元素，而是想要复制某个节点，这个时候就轮到克隆节点登场了。克隆节点分为浅克隆和深克隆，只需要向`cloneNode()`方法传入一个布尔值即可实现浅克隆和深克隆。
克隆节点的语法如下：
`节点.clone(布尔值)`

浅克隆

所谓浅克隆，就是指指克隆某一个节点，仅仅是克隆该节点而已，方法参数传入false
示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        //获取ul元素节点
        let ul = document.getElementById("test3");
        //进行浅克隆
        let newUl = ul.cloneNode(false);
        //添加到文档上面
        document.body.appendChild(newUl);
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

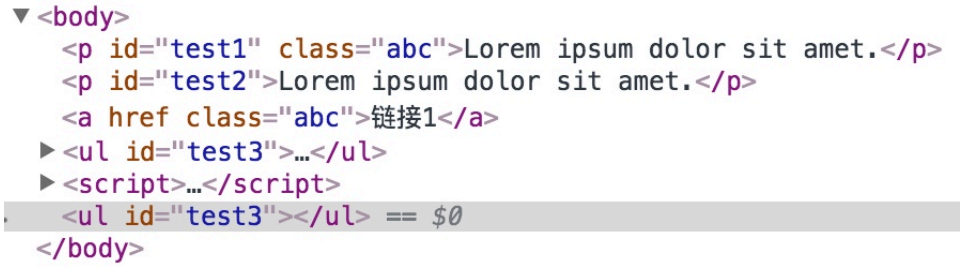
Lorem ipsum dolor sit amet.

链接1

- item1
- item2
- item3

光看上面的效果，我们感觉好像没有克隆成功，但实际上是克隆成功了的，我们打开开发者工具

进行查看



可以看到，打开开发者工具以后我们看到确实对ul进行了克隆

深克隆

所谓深克隆，就是指复制某个节点和其子节点，需要传入的布尔值为true
示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
    <script>
        //获取ul元素节点
        let ul = document.getElementById("test3");
        //进行深克隆,只需要将false改为true
        let newUl = ul.cloneNode(true);
        //添加到文档上面
        document.body.appendChild(newUl);
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet.

链接1

- item1
- item2
- item3

- item1
- item2
- item3

可以看到，我们不用打开开发者工具，直接从页面上就能直观的感受到区别，这里已经将ul以及下面的子元素进行了完完整整的复制。

9-6-5 文档碎片

在前面，我们已经学会了向文档里面添加节点，但是存在一个问题，那就是如果要添加大量节点的话，这种逐个添加的方法会显得效率很低，因为添加一个节点就会刷新一次页面。这个时候，我们就可以使用DOM里面提供的`createElementFragment()`方法，将节点先添加在一个文档碎片里面，然后最后再一次性添加到文档里面

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2" name="qwe">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li name="qwe">item2</li>
        <li>item3</li>
    </ul>
    <script>
        let  arrText = ["first","second","third"];
        let oF = document.createDocumentFragment();
        for(let i=0;i<arrText.length;i++)
        {
            //创建元素节点p
            let oP = document.createElement("p");
            //给元素节点添加文本内容
            oP.innerText = arrText[i];
            //将元素节点添加到文档碎片里面
            oF.appendChild(oP);
        }
```

```
        //将oF里面所有的节点一次性添加到文档上面
        document.body.appendChild(oF);
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet.

链接1

- item1
- item2
- item3

first

second

third

9-6-6 创建注释节点

关于节点的操作，基本上就是上面给大家介绍的那么多。这里再顺带介绍一个创建注释节点作为扩展。在DOM中提供了创建注释节点的方法`createComment()`，使用该方法可以创建一个注释，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2" name="qwe">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li name="qwe">item2</li>
        <li>item3</li>
    </ul>
    <script>
        //获取第一个段落节点
        let test1 = document.getElementById("test1");
        //创建一个注释节点
        let zhushi = document.createComment("这是一个注释");
        //将注释节点添加到第一个段落节点上面
        test1.appendChild(zhushi);
    </script>
```

```
</body>
```

效果：直接看页面效果是看不到区别的，需要打开开发者工具查看源代码

```
<!DOCTYPE html>
<html lang="en">
▶ <head>…</head>
..▼ <body> == $0
  ▼ <p id="test1" class="abc">
      "Lorem ipsum dolor sit amet."
      <!--这是一个注释-->
    </p>
    <p id="test2" name="qwe">Lorem ipsum dolor sit amet.</p>
    <a href class="abc">链接1</a>
  ▶ <ul id="test3">…</ul>
  ▶ <script>…</script>
  </body>
</html>
```

9-6-7 实时集合

最后，需要说一下的是，节点的相关信息是实时进行改变的。举个例子，如果我们添加或者删除了一个节点，那么当我们访问父节点的长度时马上就会有改变，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <p id="test2" name="qwe">Lorem ipsum dolor sit amet.</p>
    <a href="" class="abc">链接1</a>
    <ul id="test3">
        <li>item1</li>
        <li name="qwe">item2</li>
        <li>item3</li>
    </ul>
    <script>
        let ul = document.getElementById("test3");
        console.log(ul.childNodes.length);//7
        let newLi = document.createElement("li");
        newLi.innerText = "新的li元素";
        ul.appendChild(newLi);
        console.log(ul.childNodes.length);//8 长度已经实时的发生了改变
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet.

链接1

- item1
- item2
- item3
- 新的li元素

这里有一个坑，使用以前的DOM方法，例如 `document.getElementsByTagName` 来获取到的元素是实时集合的，返回的是 `HTMLCollection` ，而使用新的 `querySelectorAll` 来获取到的元素集合不是实时集合，返回来的是 `NodeList` ，这个的长度是不会实时更新的。

9-7 属性和类的操作

9-7-1 获取和设置元素属性

在DOM中提供了两个方法来获取和设置元素节点的属性，分别是`getAttribute()`以及`setAttribute()`
`getAttribute()`顾名思义就是获取到元素节点的属性值，而`setAttribute()`则是设置元素节点的属性值，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        console.log(test1.getAttribute("id"));//test1
        console.log(test1.getAttribute("class"));//abc
    </script>
</body>
```

还有一种方法，也可以获取到元素节点的属性值，是通过`getNamedItem()`方法来返回指定的属性名。需要注意的是，使用该方法时，首先需要使用`attributes`属性来获取到一个元素节点的所有的属性集合，然后再在该集合上调用`getNamedItem()`方法来返回指定的属性名。

示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        //获取到属性节点 id="test1"
        console.log(test1.attributes.getNamedItem("id"));
        //获取到属性节点的节点值 test1
        console.log(test1.attributes.getNamedItem("id").value);
    </script>
</body>
```

接下来我们来看一下设置属性，语法如下：
`setAttribute("属性名","具体值")`
具体示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        test1.setAttribute("class","qwe");
    </script>
</body>
```

效果：

```
<html lang="en">
▶ <head>…</head>
..▼ <body> == $0
    <p id="test1" class="qwe">Lorem ipsum dolor sit amet.</p>
  ▶ <script>…</script>
  </body>
</html>
```

可以看到，这里我们就确实已经改变了元素的属性了

同样，设置属性也提供了叫做 `setNamedItem()` 的方法来对元素节点的属性进行设置。和上面获取属性值使用到的 `getNamedItem()` 方法一样，也是需要用在属性集合上，也就是说要先使用 `attributes` 来获取到一个元素节点的属性集合，然后在该属性集合上来使用该方法，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        //创建一个新的属性节点
        let newAttr = document.createAttribute("class");
        //为属性节点赋值 如果该属性已经存在就覆盖 否则就是新增
        newAttr.nodeValue = "qwe";
        test1.attributes.setNamedItem(newAttr);
    </script>
</body>
```

效果：

```
<html lang="en">
▶ <head>…</head>
..▼ <body> == $0
    <p id="test1" class="qwe">Lorem ipsum dolor sit amet.</p>
  ▶ <script>…</script>
  </body>
</html>
```

这里我们用到了一个新的 `createAttribute()` 方法，该方法是用于创建属性节点，也就是说要设置属性值时，需要先创建一个属性节点，然后给这个属性节点赋值，最后将这个属性节点挂在元素节点上面。这里由于 `<p>` 元素节点存在 `class` 属性，所以是做的覆盖操作。

下面我们来演示一个为元素新增属性，如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        //创建一个新的属性节点
        let newAttr = document.createAttribute("name");
        //为属性节点赋值 如果该属性已经存在就覆盖 否则就是新增
        newAttr.nodeValue = "intro";
        test1.attributes.setNamedItem(newAttr);
    </script>
</body>
```

效果：

```
<html lang="en">
▶ <head>…</head>
..▼ <body> == $0
    <p id="test1" class="abc" name="intro">Lorem ipsum dolor sit amet.</p>
  ▶ <script>…</script>
  </body>
</html>
```

9-7-2 删除元素属性

如果要删除属性，可以使用DOM里面提供的`removeAttribute()`方法来进行删除，直接传入要删除的属性名作为参数即可，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        //删除class属性
       test1.removeAttribute("class");
    </script>
</body>
```

效果：

```
<html lang="en">
▶ <head>…</head>
.▼ <body> == $0
    <p id="test1">Lorem ipsum dolor sit amet.</p>
  ▶ <script>…</script>
  </body>
</html>
```

同样也可以使用 `removeNamedItem()` 方法来删除属性，也是同样要先用 `attributes` 获取到元素节点的属性集合，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        test1.attributes.removeNamedItem("class");
    </script>
</body>
```

效果：

```
<html lang="en">
▶ <head>…</head>
▼ <body> == $0
    <p id="test1">Lorem ipsum dolor sit amet.</p>
  ▶ <script>…</script>
  </body>
</html>
```

9-7-3 获取属性索引

`item()` 方法，该方法也是在属性集合的基础上使用，传入的参数为数字，可以定位到某一个属性上面，示例如下：

```
<body>
    <p id="test1" class="abc">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        console.log(test1.attributes.item(0));//id="test1"
        console.log(test1.attributes.item(0).nodeValue);//test1
        console.log(test1.attributes.item(1));//class="abc"
        console.log(test1.attributes.item(1).nodeValue);//abc
    </script>
</body>
```

9-7-4 获取元素的类

虽然说一个元素节点的 `class` 属性也可以通过获取属性的方法来获取到，但是新的DOM里面可以使用 `classList` 来获取一个元素节点所有的类，示例如下：

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
```

```
    <script>
        let test1 = document.getElementById("test1");
        console.log(test1.classList);
    </script>
</body>
```

效果：

```
▼ (2) ["abc", "qwe", value: "abc qwe"]
    0: "abc"
    1: "qwe"
    length: 2
    value: "abc qwe"
  ▶ __proto__: DOMTokenList
```

可以看到，通过`classList`属性确实获取到了一个元素节点的所有的类。

如果只是想要获取一个元素节点的类名，那么可以通过`className`这个属性来进行获取，示例如下：

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        console.log(test1.className);//abc qwe
    </script>
</body>
```

但是相比这个`className`，上面介绍的`classList`属性用得更多一些。因为这个属性是获取一个元素节点所有类的集合，下面要介绍的方法基本上都是建立在这个类集合上面的。

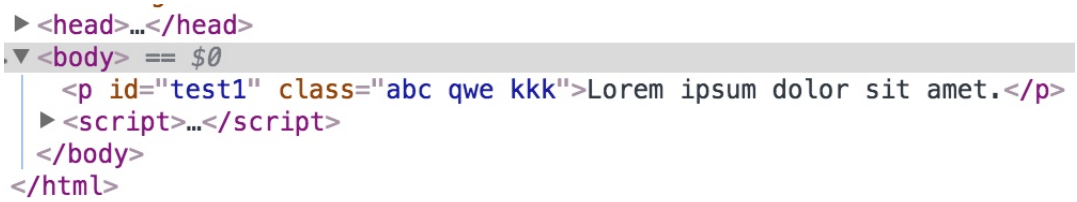
9-7-5 添加类

在新的DOM里面也新添加了一个`add()`方法来给一个元素节点快速添加类，示例如下：

> 需要注意的是该方法需要用在类名集合上面，也就是说要先使用`classList`来获取一个元素节点的所有类

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        test1.classList.add("kkk");
    </script>
</body>
```

效果：

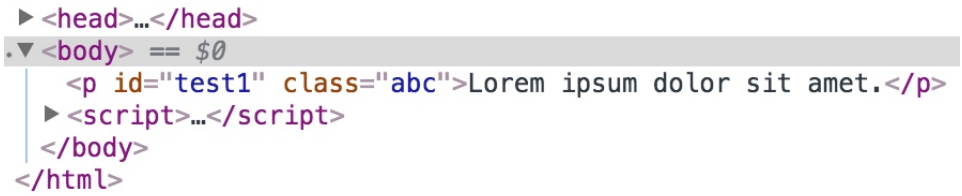


9-7-6 删除类

可以通过 `remove()` 方法来移除一个元素节点的类，也是要使用到类的集合 `classList` 上面，示例如下：

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        test1.classList.remove("qwe");
    </script>
</body>
```

效果：



9-7-7 是否含有某个类

可以使用 `contains()` 方法来检测一个元素节点是否含有指定的某一个类，示例如下：

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        let i = test1.classList.contains("qwe");
        console.log(i);//true
        let j = test1.classList.contains("kkk");
        console.log(j);//false
    </script>
</body>
```

9-7-8 切换类

`toggle()`方法是一个非常有用的方法，可以用来对类进行切换。如果元素节点没有给出的类，就添加该类，如果有该类，就删除该类。如果类被添加了，就返回true，如果是被删除了，就返回false。示例如下：

> 需要注意也是用在类的集合上面，也就是`classList`上面

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        test1.classList.toggle("kkk");
        console.log(test1.classList);
        test1.classList.toggle("kkk");
        console.log(test1.classList);
    </script>
</body>
```

效果：

```
▶ (3) ["abc", "qwe", "kkk", value: "abc qwe kkk"]
▶ (2) ["abc", "qwe", value: "abc qwe"]
```

9-8 操作CSS

每个元素节点都有一个`style`属性，通过这个属性就可以修改任何元素节点的样式规则。示例如下：

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
    <script>
        let test1 = document.getElementById("test1");
        test1.style.color = "red";
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

需要注意的是，在CSS里面有一些属性是采用的横杠隔开的，例如：`background-color`，而如果要用DOM来修改样式的话，需要修改为驼峰命名法，也就是`backgroundColor`，如下：

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
    <script>
        let i = document.body;
        i.style.backgroundColor = "red";
    </script>
</body>
```

效果：

Lorem ipsum dolor sit amet.

除了使用点的方法以外，也可以使用中括号来表示CSS属性，也就是说CSS属性可以提供为一个字符串，而不需要必须使用驼峰命名法，示例如下：

```
<body>
    <p id="test1" class="abc qwe">Lorem ipsum dolor sit amet.</p>
    <script>
        let i = document.body;
        i.style['background-color'] = "pink";
    </script>
```

```
</body>
```

效果：

Lorem ipsum dolor sit amet.

接下来我们来做一个练习，使用DOM来操作一个元素显示和隐藏，如下：

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        #test{
            width: 200px;
            height: 200px;
            background-color: pink;
            margin-top: 5px;
        }
    </style>
</head>
<body>
    <button onclick="show()">显示/隐藏</button>
    <div id="test"></div>
    <script>
        let test = document.getElementById("test");
        let show = function(){
            if(test.style.display == 'none')
            {
                test.style.display = 'block';
            }
            else{
                test.style.display = 'none';
            }
        }
    </script>
</body>
```

效果：点击后div盒子进行显示和隐藏

显示/隐藏

获取非行内样式和样式的检查

上面介绍的`style`属性只适用于行内样式，也就是说，如果我们使用`style`来获取元素节点的属性的时候，它排除了最常用的来自于外部样式表的样式，不仅外部样式表的样式获取不到，连内嵌的样式也同样无法获取。证明如下：

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <link rel="stylesheet" href="1.css">
    <style>
        p{
            color: pink;
        }
    </style>
</head>
<body>
    <p id="test">Lorem ipsum dolor sit amet.</p>
    <script>
        let test = document.getElementById("test");
        console.log(test.style);
    </script>
</body>
```

效果：

```
captionSide: ""
caretColor: ""
clear: ""
clip: ""
clipPath: ""
clipRule: ""
color: ""
colorInterpolation: ""
colorInterpolationFilters: ""
colorRendering: ""
columnCount: ""
columnFill: ""
```

可以看到，这里我们通过内嵌样式表给 `<p>` 元素节点设置了一个颜色的样式，但是当我们通过 `style` 属性访问 `<p>` 元素节点的样式时， `color` 属性显示的却为空。

我们可以通过 `document.styleSheets` 来获取到一个网页中所有样式表的引用。例如这里我们尝试来获取一个网页中内嵌的样式规则，示例如下：

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        p{
            color: red;
        }
        div{
            color: blue;
        }
    </style>
</head>
<body>
    <p>Lorem ipsum dolor sit amet.</p>
    <div>Lorem ipsum, dolor sit amet consectetur adipisicing elit.</div>
    <script>
        let cssRules = document.styleSheets[0].cssRules;
        console.log(cssRules);
    </script>
</body>
```

DOM为样式表指定了一个称为 `cssRules` 的集合，该集合里面包含了内嵌样式表表中定义的所有CSS规则，效果如下：

```
▼ CSSRuleList {0: CSSStyleRule, 1: CSSStyleRule, length: 2}
  ▼ 0: CSSStyleRule
      cssText: "p { color: red; }"
      parentRule: null
    ▶ parentStyleSheet: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
      selectorText: "p"
    ▶ style: CSSStyleDeclaration {0: "color", alignContent: "", alignItems: "", alignSelf: "", alignmentBaseline: "", all: "", …}
      type: 1
    ▶ __proto__: CSSStyleRule
  ▼ 1: CSSStyleRule
      cssText: "div { color: blue; }"
      parentRule: null
    ▶ parentStyleSheet: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
      selectorText: "div"
    ▶ style: CSSStyleDeclaration {0: "color", alignContent: "", alignItems: "", alignSelf: "", alignmentBaseline: "", all: "", …}
      type: 1
    ▶ __proto__: CSSStyleRule
    length: 2
  ▶ __proto__: CSSRuleList
```

可以看到这里成功获取到了内嵌样式表。

如果是外部样式表，一样的可以通过`document.styleSheets`来获取到样式表的引用，示例如下：

> 注意：无法像内嵌样式表一样通过`cssRules`属性来进一步获取到样式的具体规则。只会返回一个`null`。

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
   <link rel="stylesheet" href="1.css">
   <link rel="stylesheet" href="2.css">
</head>
<body>
    <p>Lorem ipsum dolor sit amet.</p>
    <div>Lorem ipsum, dolor sit amet consectetur adipisicing elit.</div>
    <script>
        let cssRules1 = document.styleSheets[0];
        console.log(cssRules1);
        let cssRules2 = document.styleSheets[1];
        console.log(cssRules2);
        let cssRules3 = document.styleSheets[0].cssRules;
        console.log(cssRules3);
    </script>
</body>
```

效果：

```
▶ CSSStyleSheet {ownerRule: null, cssRules: null, rules: null, type: "text/css", href: "file:///Users/Jie/Desktop/1.css", …}
▶ CSSStyleSheet {ownerRule: null, cssRules: null, rules: null, type: "text/css", href: "file:///Users/Jie/Desktop/2.css", …}
null
```

我们可以通过`getComputedStyle()`方法可以返回一个元素节点的所有样式信息。这是一个只读的属性，只能用于查找一个元素的样式信息，示例如下：

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <link rel="stylesheet" href="1.css">
    <style>
        p{
            color: pink;
        }
    </style>
</head>
<body>
    <p id="test">Lorem ipsum dolor sit amet.</p>
    <script>
        let test = document.getElementById("test");
        let i = getComputedStyle(test);
        console.log(i);
    </script>
</body>
```

效果：

可以看到，在调用 `getComputedStyle()` 方法以后，返回了一个 `CSSStyleDeclaration` 对象。该对象就包含了已经应用到该元素节点上的所有CSS样式的属性键值对。

我们可以点击左边的小三角将其展开，就可以看到每个CSS属性所对应的值

```
clipPath: "none"
clipRule: "nonzero"
color: "rgb(255, 0, 0)"
colorInterpolation: "sRGB"
colorInterpolationFilters: "linearRGB"
colorRendering: "auto"
```

最后，需要特别说明一下的是，虽然我们可以通过DOM提供的接口轻松的改变元素节点的样式，但是，动态的改变元素的类，并且将每个类的相关样式保存在单独的样式表里面才是更好的做法。

也就是说，相比：

```
<body>
    <p id="test">Lorem ipsum dolor sit amet.</p>
    <script>
        let test = document.getElementById("test");
        test.style.color = 'red';
    </script>
</body>
```

更好的做法是：

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <link rel="stylesheet" href="1.css">
</head>
<body>
    <p id="test">Lorem ipsum dolor sit amet.</p>
    <script>
        let test = document.getElementById("test");
        test.classList.add('newStyle');//为元素添加一个新的类
    </script>
</body>
```

然后在一个样式表里面(这里的1.css)添加如下的CSS

```
.newStyle{
    color: red;
}
```

9-9 操作表格

在实际开发中，经常会涉及到对表格的操作。在DOM里面，table对象就代表着HTML中的一个表格。在HTML文档中，`<table>`标签每出现一次，就代表着一个table对象被创建。

9-9-1 获取表格的行和列

我们知道，一个表格是由行和列组成的，我们可以通过`rows`和`cells`来获取到一个表格对象的行与列
示例如下：

```
<body>
    <table border="1" id="tab">
        <tr>
            <td>姓名</td>
            <td>年龄</td>
        </tr>
        <tr>
            <td>谢杰</td>
            <td>18</td>
        </tr>
        <tr>
            <td>雅静</td>
            <td>20</td>
        </tr>
    </table>
    <script>
        let tab = document.getElementById("tab");
        console.log(tab.rows);//获取表格的行数
        //获取表格的列数,需要具体到某一行
        console.log(tab.rows[0].cells);
        //获取表格里面一个单元格
        console.log(tab.rows[1].cells[0]);//<td>谢杰</td>
        //获取表格里面一个单元格的具体的值
        console.log(tab.rows[1].cells[0].innerText);//谢杰
    </script>
</body>
```

效果：

```
▶ (3) [tr, tr, tr]
▶ (2) [td, td]
  <td>谢杰</td>
谢杰
```

如果需要修改表格里面的内容，只需要定位到具体的单元格然后重新赋值即可，如下：

```
<body>
    <table border="1" id="tab">
        <tr>
            <td>姓名</td>
            <td>年龄</td>
        </tr>
        <tr>
            <td>谢杰</td>
            <td>18</td>
        </tr>
        <tr>
            <td>雅静</td>
            <td>20</td>
        </tr>
    </table>
    <script>
        let tab = document.getElementById("tab");
        tab.rows[2].cells[0].innerText = "小宋";
    </script>
</body>
```

效果：

姓名	年龄
谢杰	18
小宋	20

9-9-2 遍历表格的内容

前面有讲过，数组里面的二维数组，就类似于一个表格。既然遍历二维数组需要双层for循环。那么自然遍历表格的内容也是使用双层for循环来搞定的，具体示例如下：

```
<body>
    <table border="1" id="tab">
        <tr>
            <td>姓名</td>
```

```
            <td>年龄</td>
        </tr>
        <tr>
            <td>谢杰</td>
            <td>18</td>
        </tr>
        <tr>
            <td>雅静</td>
            <td>20</td>
        </tr>
    </table>
    <script>
        let tab = document.getElementById("tab");
        for(let i=0;i<tab.rows.length;i++)
        {
            for(let j=0;j<tab.rows[i].cells.length;j++)
            {
                console.log(tab.rows[i].cells[j].innerText);
            }
        }
    </script>
</body>
```

效果：

姓名
年龄
谢杰
18
雅静
20

9-9-3 表格插入新的行与列

在DOM里面提供了`insertRow()`以及`insertCell()`这两个方法来为表格插入行与列，语法如下：

`inserRow(position)`：在表格对象的指定位置上插入一个新行。

`insertCell(postion)`：在rows集合中的指定位置上插入一个新的单元格。

示例如下：

```
<body>
    <table border="1" id="tab">
        <tr>
            <td>姓名</td>
            <td>年龄</td>
```

```
        </tr>
        <tr>
            <td>谢杰</td>
            <td>18</td>
        </tr>
        <tr>
            <td>雅静</td>
            <td>20</td>
        </tr>
    </table>
    <script>
        let tab = document.getElementById("tab");//获取到表格对象
        tab.insertRow(3);//插入第4行
        //在第4行的第1列添加文本内容
        tab.rows[3].insertCell(0).appendChild(document.createTextNode("希之")
);
        //在第4行的第2列添加文本内容
        tab.rows[3].insertCell(1).appendChild(document.createTextNode("1"));
    </script>
</body>
```

效果：

姓名	年龄
谢杰	18
雅静	20
希之	1

9-9-4 删除行与列

既然有增加表格的行与列，那么必然会有删除一个表格的行与列。在DOM中也是提供了对应的两个方法，如下：

`deleteRow(postion)`：删除表格对象上指定位置的一行。

`deleteCell(postion)`：删除在rows集合中指定位置上的一个单元格。

示例如下：

```
<body>
    <table border="1" id="tab">
        <tr>
            <td>姓名</td>
            <td>年龄</td>
        </tr>
        <tr>
            <td>谢杰</td>
```

```
            <td>18</td>
        </tr>
        <tr>
            <td>雅静</td>
            <td>20</td>
        </tr>
    </table>
    <script>
        let tab = document.getElementById("tab");//获取到表格对象
        tab.deleteRow(2);//删除表格对象中的第3行
        tab.rows[1].deleteCell(1);//删除表格对象中第2行的第2个单元格
    </script>
</body>
```

效果：

姓名	年龄
谢杰	

9-9-5 表格其他相关属性和方法

通过上面我们给大家所介绍的表格属性以及方法，大家已经能够对页面里面的表格实现最基本的增删改查操作了。但是除了上面所介绍的属性和方法以外，实际上DOM里面还提供了其他很多表格相关的属性和方法。这里以表格的形式列举出来，如下：

表格的属性或方法	作用
caption	指向 `<caption>` 元素(如果存在)
tHead	指向 `<thead>` 元素(如果存在)
tBodies	指向 `<tbody>` 元素(如果存在)
tFoot	指向 `<tFoot>` 元素(如果存在)
rows	表格中所有行的集合
cells	一行里面所有单元格的集合
createCaption()	创建 `<caption>` 元素并将其放入表格
createTHead()	创建 `<thead>` 元素并将其放入表格
createTFoot()	创建 `<tFoot>` 元素并将其放入表格
deleteCaption()	删除 `<caption>` 元素

deleteTHead()	删除 `<thead>` 元素
deleteTFoot()	删除 `<tFoot>` 元素
insertRow(postion)	在表格对象的指定位置上插入一个新行
deleteRow(postion)	删除表格对象上指定位置的一行
insertCell(postion)	在rows集合中的指定位置上插入一个新的单元格
deleteCell(postion)	删除在rows集合中指定位置上的一个单元格

上面表格中所展示的属性和方法，有一部分是我们在介绍对表格做增删改查操作时所介绍过的，而有些则没有进行具体介绍，但是基本用法都是大同小异的，所以这里不再做具体演示。

总结

1. DOM是文档对象模型。将HTML整个文档看作是一个对象，而一个文档又是又一个个元素节点对象组成。
2. DOM的历史可以分为DOM 0级到DOM 4级。
3. 节点有不同的节点名称和节点类型。
4. 在DOM 0级中提供了一组方法来访问到页面中常见的元素，例如图像，链接和表单等。
5. 从DOM 2级开始提供了一组可以快速定位节点位置的方法。
6. 在DOM 2级之前，通常使用节点之间的关系来对节点进行查找。
7. 在定位到具体的节点之后，我们可以通过nodeValue、textContent、innerText以及innerHTML来获取节点内部的值。
8. 对节点的操作无非就是增删改查，DOM中提供了大量的方法可以让我们很方便的实现这些功能。
9. 通过DOM我们也能够获取和设置元素节点的属性以及类。
10. 利用DOM我们还可以很轻松的实现修改元素节点的样式，改变它们的CSS规则。
11. DOM中还专门提供了一组表格相关的属性和方法让我们很轻松的对表格数据实现增删改查。