DOM 简介

文档**对象模型**(*DOM*) 是构成 Web 文档结构和内容的对象的数据表示。本指南将介绍 DOM，了解 DOM 如何在内存中表示HTML文档以及如何使用 API 来创建 Web 内容和应用程序。

什么是 DOM?

文档对象模型 (DOM) 是 Web 文档的编程接口。它代表页面，以便程序可以更改文档结构、样式和内容。DOM 将文档表示为节点和对象；这样，编程语言就可以与页面交互。

网页是可以显示在浏览器窗口中或作为 HTML 源的文档。在这两种情况下，它都是同一个文档，但文档对象模型 (DOM) 表示允许对其进行操作。作为网页的面向对象表示，可以使用 JavaScript 等脚本语言对其进行修改。

例如，DOM 指定此代码片段中的方法必须返回文档中 `querySelectorAll` 所有元素的列表： `<p>`

```
const paragraphs = document.querySelectorAll("p");
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

所有可用于操作和创建网页的属性、方法和事件都被组织到对象中。例如，`document` 表示文档本身的对象、 `table` 实现 `HTMLTableElement` 用于访问 HTML 表格的 DOM 接口的任何对象等等，都是对象。

DOM 是使用多个协同工作的 API 构建的。核心DOM定义了描述任何文档及其中的对象的实体。其他 API 会根据需要对此进行扩展，这些 API 会向

DOM 添加新的特性和功能。例如，HTML DOM API向核心 DOM 添加了对表示 HTML 文档的支持，而 SVG API 添加了对表示 SVG 文档的支持。

DOM 和 JavaScript

与几乎所有示例一样，前面的简短示例是JavaScript。也就是说，它是用JavaScript*编写的*，但*使用*DOM 来访问文档及其元素。DOM 不是一种编程语言，但如果没有它，JavaScript 语言就不会有网页、HTML 文档、SVG 文档及其组成部分的任何模型或概念。作为一个整体的文档、头部、文档中的表格、表格标题、表格单元格中的文本以及文档中的所有其他元素都是该文档的文档对象模型的一部分。它们都可以使用 DOM 和脚本语言（如 JavaScript）进行访问和操作。

DOM 不是 JavaScript 语言的一部分，而是一种用于构建网站的 Web API。JavaScript 也可以在其他上下文中使用。例如，Node.js 在计算机上运行 JavaScript 程序，但提供了一组不同的 API，并且 DOM API 不是 Node.js 运行时的核心部分。

DOM 被设计为独立于任何特定的编程语言，使文档的结构表示可从单个一致的 API 获得。即使大多数 Web 开发人员只会通过 JavaScript 使用 DOM，也可以为任何语言构建 DOM 的实现，如这个 Python 示例所示：

```
# Python DOM example
import xml.dom.minidom as m
doc = m.parse(r"C:\Projects\Py\chap1.xml")
doc.nodeName # DOM property of document object
p_list = doc.getElementsByTagName("para")
```

有关在 Web 上编写 JavaScript 涉及哪些技术的更多信息，请参阅 JavaScript 技术概述。

访问 DOM

您无需执行任何特殊操作即可开始使用 DOM。您可以直接在 JavaScript 中从所谓的*脚本*（由浏览器运行的程序）中使用 API。

当您创建脚本时，无论是内嵌在元素中 `<script>` 还是包含在网页中，您都可以立即开始使用 `document` 或 `window` 对象的 API 来操作文档本身，或网页中的任何各种元素（的后代元素文档）。您的 DOM 编程可能像以下示例一样简单，它使用以下函数在控制台上显示一条消息 `console.log()`：

```
<body onload="console.log('Welcome to my home page!');">
  …
</body>
```

由于通常不建议混合页面结构（用 HTML 编写）和 DOM 操作（用 JavaScript 编写），因此 JavaScript 部分将在这里组合在一起，并与 HTML 分开。

例如，以下函数创建一个新的h1元素，向该元素添加文本，然后将其添加到文档树中：

```
<html lang="en">
  <head>
    <script>
      // run this function when the document is loaded
      window.onload = () => {
        // create a couple of elements in an otherwise empty
HTML page
        const heading = document.createElement("h1");
        const headingText = document.createTextNode("Big
Head!");
        heading.appendChild(headingText);
        document.body.appendChild(heading);
      };
    </script>
  </head>
  <body></body>
</html>
```

基本数据类型

本页试图用简单的术语描述各种对象和类型。但是您应该了解 API 周围传递的许多不同数据类型。

> **注意：**因为绝大多数使用 DOM 的代码都围绕着操作 HTML 文档，所以通常将 DOM 中的节点称为**元素**，尽管严格来说并不是每个节点都是元素。

下表简要描述了这些数据类型。

数据类型（接口）	描述
`Document`	当一个成员返回一个类型的对象时 `document` （例如， `ownerDocument` 一个元素的属性返回 `document` 它所属的），这个对象就是根 `document` 对象本身。DOM `document` 参考章节 描述了该 `document` 对象。
`Node`	文档中的每个对象都是某种节点。在 HTML 文档中，对象可以是元素节点，也可以是文本节点或属性节点。
`Element`	该 `element` 类型基于 `node`。`element` 它指的是DOM API 成员返回的元素或类型的节点。例如，与其说该 `document.createElement()` 方法返回对 a 的对象引用 `node`，不如说该方法返回 `element` 刚刚在 DOM 中创建的。 `element` 对象实现 DOM `Element` 接口和更基本的 `Node` 接口，这两者都包含在本参考中。在 HTML 文档中，HTML DOM API 的 `HTMLElement` 接口以及描述特定类型元素功能的其他接口进一步增强了元素（例如， `HTMLTableElement` 对于 `<table>` 元素）。
`NodeList`	A `nodeList` 是一个元素数组，类似于方法返回的类型 `document.querySelectorAll()`。a 中的项目 `nodeList` 通过索引以两种方式之一访问： • 列表.item(1) • 名单[1]

数据类型（接口）	描述
	这两个是等价的。首先， `item()` 是对象上的单一方法 `nodeList` 。后者使用典型的数组语法来获取列表中的第二项。
`Attr`	当 an `attribute` 由成员（例如，通过 `createAttribute()` 方法）返回时，它是一个对象引用，为属性公开一个特殊的（尽管很小）接口。属性就像元素一样是 DOM 中的节点，尽管您可能很少这样使用它们。
`NamedNodeMap`	A `namedNodeMap` 就像一个数组，但项目是通过名称或索引访问的，尽管后一种情况只是为了方便枚举，因为它们在列表中没有特定的顺序。A `namedNodeMap` 有一个 `item()` 用于此目的的方法，您还可以从 `namedNodeMap` .

还需要记住一些常见的术语注意事项。例如，通常将任何 `Attr` 节点称为 `attribute` ，并将 DOM 节点数组称为 `nodeList` 。您会发现这些术语和其他术语在整个文档中都有介绍和使用。

DOM 接口

本指南是关于对象和可用于操作 DOM 层次结构的实际*事物*。在很多地方，理解这些工作原理可能会令人困惑。例如，表示 HTML 元素的对象从接口 `form` 获取其属性，但从接口获取其属性。在这两种情况下，您想要的属性都在该表单对象中。 `name` `HTMLFormElement` `className` `HTMLElement`

但是对象和它们在 DOM 中实现的接口之间的关系可能会令人困惑，因此本节试图对 DOM 规范中的实际接口以及如何提供它们进行一些说明。

接口和对象

许多对象实现几个不同的接口。例如，表对象实现了一个专用 `HTMLTableElement` 接口，其中包括 `createCaption` 和等方法 `insertRow` 。

但由于它也是一个 HTML 元素， `table` 因此实现了 `Element` DOM `Element` 参考章节中描述的接口。最后，由于就 DOM 而言，HTML 元素也是构成 HTML 或 XML 页面对象模型的节点树中的一个节点，因此表对象还实现了更基本的接口， `Node` 从中 `Element` 派生。

当您获得对某个对象的引用时 `table` ，如以下示例所示，您通常会在对象上交替使用所有这三个接口，而您可能并不知道。

```
const table = document.getElementById("table");
const tableAttrs = table.attributes; // Node/Element interface
for (let i = 0; i < tableAttrs.length; i++) {
  // HTMLTableElement interface: border attribute
  if (tableAttrs[i].nodeName.toLowerCase() === "border") {
    table.border = "1";
  }
}
// HTMLTableElement interface: summary attribute
table.summary = "note: increased border";
```

DOM 中的核心接口

本节列出了 DOM 中一些最常用的接口。这里的想法不是描述这些 API 的作用，而是让您了解在使用 DOM 时会经常看到的方法和属性的种类。这些通用 API 用于本书末尾DOM 示例一章中较长的示例。

The `document` 和 `window` 对象是您通常在 DOM 编程中最常使用其接口的对象。简单来说， `window` 对象代表浏览器之类的东西， `document` 对象就是文档本身的根。 `Element` 继承自通用 `Node` 接口，这两个接口一起提供了您在单个元素上使用的许多方法和属性。这些元素也可能有特定的接口来处理这些元素所持有的数据类型，如上 `table` 一节中的对象示例。

以下是使用 DOM 编写 Web 和 XML 页面脚本的常见 API 的简要列表。

- `document.querySelector(selector)`
- `document.querySelectorAll(name)`
- `document.createElement(name)`
- `parentNode.appendChild(node)`

- element.innerHTML
- element.style.left
- element.setAttribute()
- element.getAttribute()
- element.addEventListener()
- Window.onload
- window.scrollTo()

例子

设置文字内容

此示例使用 包含 a <textarea> 和两个 <button> 元素的元素。当用户点击第一个按钮时，我们在 <textarea> . 当用户单击第二个按钮时，我们清除文本。我们用：

- Document.querySelector() 访问 <textarea> 和按钮
- EventTarget.addEventListener() 监听按钮点击
- Node.textContent 设置和清除文本。

HTML

```
<div class="container">
  <textarea class="story"></textarea>
  <button id="set-text" type="button">Set text content</button>
  <button id="clear-text" type="button">Clear text
content</button>
</div>
```

CSS

```
.container {
  display: flex;
  gap: 0.5rem;
  flex-direction: column;
}
```

```
button {
  width: 200px;
}
```

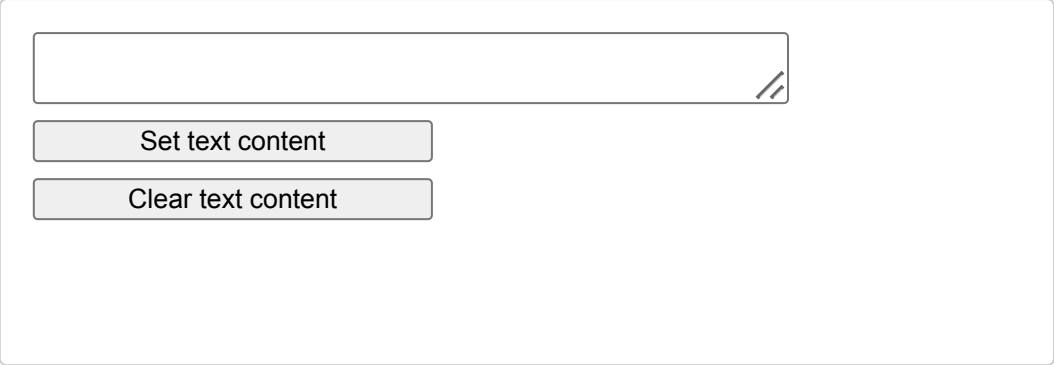
JavaScript

```
const story = document.body.querySelector(".story");

const setText = document.body.querySelector("#set-text");
setText.addEventListener("click", () => {
  story.textContent = "It was a dark and stormy night...";
});

const clearText = document.body.querySelector("#clear-text");
clearText.addEventListener("click", () => {
  story.textContent = "";
});
```

结果



添加子元素

此示例使用 `<div>` 包含 a `<div>` 和两个 `<button>` 元素的元素。当用户单击第一个按钮时，我们创建一个新元素并将其添加为 `<div>`。当用户单击第二个按钮时，我们删除子元素。我们用：

- `Document.querySelector()` 访问 `<div>` 和按钮
- `EventTarget.addEventListener()` 监听按钮点击
- `Document.createElement` 创建元素
- `Node.appendChild()` 添加孩子
- `Node.removeChild()` 删除孩子。

HTML

```
<div class="container">
  <div class="parent">parent</div>
  <button id="add-child" type="button">Add a child</button>
  <button id="remove-child" type="button">Remove child</button>
</div>
```

CSS

```
.container {
  display: flex;
  gap: 0.5rem;
  flex-direction: column;
}

button {
  width: 100px;
}
```



```
  padding: 5px;
  width: 100px;
  height: 100px;
}

div.child {
  border: 1px solid red;
  margin: 10px;
  padding: 5px;
  width: 80px;
  height: 60px;
  box-sizing: border-box;
}
```

JavaScript

```
const parent = document.body.querySelector(".parent");

const addChild = document.body.querySelector("#add-child");
addChild.addEventListener("click", () => {
  // Only add a child if we don't already have one
  // in addition to the text node "parent"
```

```
  if (parent.childNodes.length > 1) {
    return;
  }
  const child = document.createElement("div");
  child.classList.add("child");
  child.textContent = "child";
  parent.appendChild(child);
});

const removeChild = document.body.querySelector("#remove-
child");
removeChild.addEventListener("click", () => {
  const child = document.body.querySelector(".child");
  parent.removeChild(child);
});
```

结果

规格

规格
DOM 标准

此页面最后修改于 2023 年 4 月 15 日由MDN 贡献者提供。