> 此页面由社区从英文翻译而来。了解更多并加入 MDN Web Docs 社区。

DOM 概述

文档对象模型（*DOM*）是 web 上构成文档结构和内容的对象的数据表示。本指南将介绍 DOM，研究 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）是一个网络文档的编程接口。它代表页面，以便程序可以改变文档的结构、风格和内容。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。

网页是一个既可以在浏览器窗口中显示，也可以作为 HTML 源代码的文档。在这两种情况下，它都是同一个文档，但文档对象模型（DOM）的表示方式使它可以被操作。作为一个面向对象的网页表示，它可以用脚本语言（如 JavaScript）进行修改。

例如，DOM 中指定下面代码中的 `querySelectorAll` 方法必须要返回文档中所有 `<p>` 元素组成的列表：

JS

```
const paragraphs = document.querySelectorAll("p");
// paragraphs[0] 是第一个 <p> 元素
// paragraphs[1] 是第二个 <p> 元素，依此类推
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，代表文档本身的 `document` 对象，任何实现 `HTMLTableElement` 访问 HTML 表格的 `table` 对象等，都是对象。

DOM 是使用多个 API 构建的，它们一起工作。核心 DOM 定义了描述任何文档和其中对象的实体。其他 API 会根据需要对其进行扩展，为 DOM 增加新的特性和功能。例如，HTML DOM API

为核心 DOM 增加了对表示 HTML 文档的支持，而 SVG API 增加了对表示 SVG 文档的支持。

DOM 和 JavaScript

上一个简短的例子，和几乎所有的例子一样，是 JavaScript。也就是说，它是用 JavaScript *写*的，但却*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 语言就不会有任何关于网页、HTML 文档、SVG 文档及其组成部分的模型或概念。文档作为一个整体，标题、文档中的表格、表头、表格单元中的文本以及文档中的所有其他元素都是该文档对象模型的一部分。它们都可以使用 DOM 和像 JavaScript 这样的脚本语言进行访问和操作。

DOM 不是 JavaScript 语言的一部分，而是用于建立网站的 Web API。JavaScript 也可以在其他情况下使用。例如，Node.js 在计算机上运行 JavaScript 程序，但提供了一套不同的 API，而且 DOM API 不是 Node.js 运行时的核心部分。

DOM 被设计成与特定编程语言相独立，使文档的结构化表述可以通过单一，一致的 API 获得。尽管我们在本参考文档中会专注于使用 JavaScript，但 DOM 也可以使用其他的语言来实现，以 Python 为例，代码如下：

PYTHON

```python
# Python DOM 示例
import xml.dom.minidom as m
doc = m.parse(r"C:\Projects\Py\chap1.xml")
doc.nodeName # 文档对象的 DOM 属性
p_list = doc.getElementsByTagName("para")
```

要获取更多在网页上使用 JavaScript 的信息，请参阅 JavaScript 技术概述。

访问 DOM

在使用 DOM 时，不需要做任何其他特殊的操作。你可以在*脚本*中直接使用 JavaScript 的 API，这是一个由浏览器运行的程序。

当你创建一个脚本时，无论是内联到 `<script>` 元素中还是包含在网页中，你都可以立即开始使用 `document` 或 `window` 对象的 API 来操作文档本身，或网页中的任何元素（文档的子元素）。你的 DOM 编程可能就像下面这个示例一样简单，通过使用 `console.log()` 函数在控制台显示一条信息：

HTML

```
<body onload="console.log('欢迎来到我的主页！');">
  …
</body>
```

由于一般不建议将页面的结构（用 HTML 编写）和对 DOM 的操作（用 JavaScript 编写）混在一起，所以 JavaScript 部分将在这里分组，并与 HTML 分开。

例如，以下函数创建了一个新的 h1 元素，向该元素添加文本，然后将其添加到文档的树中：

HTML

```
<html lang="zh-CN">
  <head>
    <script>
      // 当文档加载时，运行这段函数
      window.onload = () => {
        // 在空 HTML 页面中创建一系列元素
        const heading = document.createElement("h1");
        const headingText = document.createTextNode("Big Head!");
        heading.appendChild(headingText);
        document.body.appendChild(heading);
      };
    </script>
  </head>
  <body></body>
</html>
```

基本的数据类型

本页试图用简单的语言描述各种对象和类型。但是，有一些不同的数据类型在 API 中传递，你应该注意到。

> **备注：** 因为绝大多数使用 DOM 的代码都是围绕着操作 HTML 文档进行的，所以我们通常把 DOM 中的节点称为**元素**（element），尽管严格来说并不是每个节点都是一个元素。

下面的表格大致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 document 对象（例如，元素的 ownerDocument 属性返回它所属的 document），这个对象就是 root document 对象本身。DOM document 参考一章对 document 对象进行了描述。
Node	位于文档中的每个对象都是某种类型的节点。在一个 HTML 文档中，一个对象可以是一个元素节点，也可以是一个文本节点或属性节点。
Element	element 类型是基于 node 的。它指的是一个元素或一个由 DOM API 的成员返回的 element 类型的节点。例如，我们不说 document.createElement() 方法返回一个 node 的对象引用，而只是说这个方法返回刚刚在 DOM 中创建的 element。 element 对象实现了 DOM 的 Element 接口和更基本的 Node 接口，这两个接口都包含在本参考中。在 HTML 文档中，元素通过 HTML DOM API 的 HTMLElement 接口以及其他描述特定种类元素能力的接口（例如用于 <table> 元素的 HTMLTableElement 接口）进一步强化。
NodeList	nodeList 是由元素组成的数组，如同 document.querySelectorAll() 等方法返回的类型。 nodeList 中的条目通过索引有两种方式进行访问： • list.item(1) • list[1] 两种方式是等价的，第一种方式中 item() 是 nodeList 对象中的单独方法。后面的方式则使用了经典的数组语法来获取列表中的第二个条目。
Attr	当 attribute 通过成员函数（例如通过 createAttribute() 方法）返回时，它是一个为属性暴露出专门接口的对象引用。DOM 中的属性也是节点，就像元素一样，只不过你可能会很少使用它。
NamedNodeMap	namedNodeMap 和数组类似，但是条目是由名称或索引访问的，虽然后一种方式仅仅是为了枚举方便，因为在 list 中本来就没有特定的顺序。出于这个目的， namedNodeMap 有一个 item() 方法，你也可以从 namedNodeMap 添加或移除条目。

还有一些常见的术语需要记住。例如，通常把任何 Attr 节点称为 attribute，把 DOM 节点组成的数组称为 nodeList。你会发现这些术语和其他术语将在整个文档中被介绍和使用。

DOM 接口

本指南是关于对象和你可以用来操作 DOM 层次结构的实际*事物*。在很多地方，理解这些东西的工作方式会让人感到困惑。例如，代表 HTML `form` 元素的对象从 `HTMLFormElement` 接口获得其

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，因此本节试图讲述一些关于 DOM 规范中的实际接口以及它们如何被提供的内容。

接口及对象

许多对象实现了几个不同的接口。例如，table 对象实现了一个专门的 `HTMLTableElement` 接口，其中包括诸如 `createCaption` 和 `insertRow` 等方法。但由于它也是一个 HTML 元素，所以 `table` 实现了 DOM `Element` 参考章节中描述的 `Element` 接口。最后，因为就 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，所以表格对象也实现了更基本的 `Node` 接口， `Element` 就是从这个接口衍生出来的。

正如下面的示例，当你得到一个 `table` 对象的引用时，你经常会轮流使用对象实现的三个不同接口的方法，但并不知其所以然。

JS

```
const table = document.getElementById("table");
const tableAttrs = table.attributes; // Node/Element 接口
for (let i = 0; i < tableAttrs.length; i++) {
  // HTMLTableElement 接口：border 属性
  if (tableAttrs[i].nodeName.toLowerCase() === "border") {
    table.border = "1";
  }
}
// HTMLTableElement 接口：summary 属性
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。我们的想法不是在这里描述这些 API 的作用，而是让你了解在你使用 DOM 时经常会看到的各种方法和属性。这些常用的 API 在本节最后的 DOM 实例一章中的较长的示例中使用。

`document` 和 `window` 对象是你在 DOM 编程中最常使用的接口对象。简单来说， `window` 对象代表类似浏览器的东西，而 `document` 对象是文档本身的根。 `Element` 继承自通用的 `Node` 接口，这两个接口一起提供了许多你在单个元素上使用的方法和属性。这些元素也可以有特定的接口来处理这些元素持有的数据种类，如上一节中的 `table` 对象的示例。

下面是在 web 和 XML 页面脚本中使用 DOM 时，一些常用的 API 简要列表。

- `document.querySelector()`
- `document.querySelectorAll()`
- `document.createElement()`
- `Element.innerHTML`
- `Element.setAttribute()`
- `Element.getAttribute()`
- `EventTarget.addEventListener()`
- `HTMLElement.style`
- `Node.appendChild()`
- `window.onload`
- `window.scrollTo()`

示例

设置文本内容

本实例使用了包含了一个 `<textarea>` 和两个 `<button>` 元素的 `<div>` 元素。当用户点击第一个按钮时，我们在 `<textarea>` 中放置一些文本。当用户点击第二个按钮时，我们清除文本内容。我们使用：

- `Document.querySelector()` 来访问 `<textarea>` 和按钮
- `EventTarget.addEventListener()` 来监听按钮的点击事件
- `Node.textContent` 来设置和清除文本

HTML

```
HTML                                                    Play
```

```
<div class="container">
  <textarea class="story"></textarea>
  <button id="set-text" type="button">设置文本内容</button>
  <button id="clear-text" type="button">清除文本内容</button>
</div>
```

CSS

CSS Play

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

button {
  width: 200px;
}
```

JavaScript

JS Play

```
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

Play

添加子元素

本实例使用了包含了一个 `<div>` 和两个 `<button>` 元素的 `<div>` 元素。当用户点击第一个按钮时，我们创建一个新的元素，并添加到 `<div>` 的子元素。当用户点击第二个按钮时，我们移除那个子元素。我们使用：

- `Document.querySelector()` 来访问 `<div>` 和按钮
- `EventTarget.addEventListener()` 来监听按钮点击事件
- `Document.createElement` 来创建元素
- `Node.appendChild()` 来添加子元素
- `Node.removeChild()` 来移除子元素

HTML

HTML Play

```
<div class="container">
  <div class="parent">父元素</div>
  <button id="add-child" type="button">添加子元素</button>
  <button id="remove-child" type="button">移除子元素</button>
</div>
```

CSS

CSS Play

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

button {
```

```
  width: 100px;
}

div.parent {
  border: 1px solid black;
  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

JS Play

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

const addChild = document.body.querySelector("#add-child");
addChild.addEventListener("click", () => {
  // 只在文本节点“父节点”没有子节点时添加一个子节点
  if (parent.childNodes.length > 1) {
    return;
  }
  const child = document.createElement("div");
  child.classList.add("child");
  child.textContent = "子节点";
  parent.appendChild(child);
});

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

结果

Play

父元素

添加子元素

移除子元素

规范

Specification
DOM Standard

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on 2023年8月7日 by MDN contributors.