```
document.body.className = 'foo';

setTimeout(() => {
  observer.disconnect();
  document.body.className = 'bar';
}, 0);

// <body> attributes changed
```

4. 复用 MutationObserver

多次调用 observe()方法，可以复用一个 MutationObserver 对象观察多个不同的目标节点。此时，MutationRecord 的 target 属性可以标识发生变化事件的目标节点。下面的示例演示了这个过程：

```
let observer = new MutationObserver(
               (mutationRecords) => console.log(mutationRecords.map((x) =>
x.target)));

// 向页面主体添加两个子节点
let childA = document.createElement('div'),
    childB = document.createElement('span');
document.body.appendChild(childA);
document.body.appendChild(childB);

// 观察两个子节点
observer.observe(childA, { attributes: true });
observer.observe(childB, { attributes: true });

// 修改两个子节点的属性
childA.setAttribute('foo', 'bar');
childB.setAttribute('foo', 'bar');

// [<div>, <span>]
```

disconnect()方法是一个“一刀切”的方案，调用它会停止观察所有目标：

```
let observer = new MutationObserver(
               (mutationRecords) => console.log(mutationRecords.map((x) =>
x.target)));

// 向页面主体添加两个子节点
let childA = document.createElement('div'),
    childB = document.createElement('span');
document.body.appendChild(childA);
document.body.appendChild(childB);

// 观察两个子节点
observer.observe(childA, { attributes: true });
observer.observe(childB, { attributes: true });

observer.disconnect();

// 修改两个子节点的属性
childA.setAttribute('foo', 'bar');
childB.setAttribute('foo', 'bar');

//  (没有日志输出)
```

5. 重用 MutationObserver

调用 disconnect()并不会结束 MutationObserver 的生命。还可以重新使用这个观察者，再将它关联到新的目标节点。下面的示例在两个连续的异步块中先断开然后又恢复了观察者与<body>元素的关联：

```
let observer = new MutationObserver(() => console.log('<body> attributes
changed'));

observer.observe(document.body, { attributes: true });

// 这行代码会触发变化事件
document.body.setAttribute('foo', 'bar');

setTimeout(() => {
  observer.disconnect();

  // 这行代码不会触发变化事件
  document.body.setAttribute('bar', 'baz');
}, 0);


setTimeout(() => {
  // Reattach
  observer.observe(document.body, { attributes: true });

  // 这行代码会触发变化事件
  document.body.setAttribute('baz', 'qux');
}, 0);

// <body> attributes changed
// <body> attributes changed
```

14.3.2 MutationObserverInit 与观察范围

MutationObserverInit 对象用于控制对目标节点的观察范围。粗略地讲，观察者可以观察的事件包括属性变化、文本变化和子节点变化。

下表列出了 MutationObserverInit 对象的属性。

属　性	说　明
subtree	布尔值，表示除了目标节点，是否观察目标节点的子树（后代） 如果是 false，则只观察目标节点的变化；如果是 true，则观察目标节点及其整个子树 默认为 false
attributes	布尔值，表示是否观察目标节点的属性变化 默认为 false
attributeFilter	字符串数组，表示要观察哪些属性的变化 把这个值设置为 true 也会将 attributes 的值转换为 true 默认为观察所有属性
attributeOldValue	布尔值，表示 MutationRecord 是否记录变化之前的属性值 把这个值设置为 true 也会将 attributes 的值转换为 true 默认为 false

（续）

属　性	说　明
characterData	布尔值，表示修改字符数据是否触发变化事件 默认为 false
characterDataOldValue	布尔值，表示 MutationRecord 是否记录变化之前的字符数据 把这个值设置为 true 也会将 characterData 的值转换为 true 默认为 false
childList	布尔值，表示修改目标节点的子节点是否触发变化事件 默认为 false

注意　在调用 observe()时，MutationObserverInit 对象中的 attribute、characterData 和 childList 属性必须至少有一项为 true（无论是直接设置这几个属性，还是通过设置 attributeOldValue 等属性间接导致它们的值转换为 true）。否则会抛出错误，因为没有任何变化事件可能触发回调。

1. 观察属性

MutationObserver 可以观察节点属性的添加、移除和修改。要为属性变化注册回调，需要在 MutationObserverInit 对象中将 attributes 属性设置为 true，如下所示：

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

// 添加属性
document.body.setAttribute('foo', 'bar');

// 修改属性
document.body.setAttribute('foo', 'baz');

// 移除属性
document.body.removeAttribute('foo');

// 以上变化都被记录下来了
// [MutationRecord, MutationRecord, MutationRecord]
```

把 attributes 设置为 true 的默认行为是观察所有属性，但不会在 MutationRecord 对象中记录原来的属性值。如果想观察某个或某几个属性，可以使用 attributeFilter 属性来设置白名单，即一个属性名字符串数组：

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributeFilter: ['foo'] });

// 添加白名单属性
document.body.setAttribute('foo', 'bar');

// 添加被排除的属性
document.body.setAttribute('baz', 'qux');
```

```
// 只有 foo 属性的变化被记录了
// [MutationRecord]
```

如果想在变化记录中保存属性原来的值，可以将 attributeOldValue 属性设置为 true：

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords.map((x) => x.oldValue)));

observer.observe(document.body, { attributeOldValue: true });

document.body.setAttribute('foo', 'bar');
document.body.setAttribute('foo', 'baz');
document.body.setAttribute('foo', 'qux');

// 每次变化都保留了上一次的值
// [null, 'bar', 'baz']
```

2. 观察字符数据

MutationObserver 可以观察文本节点（如 Text、Comment 或 ProcessingInstruction 节点）中字符的添加、删除和修改。要为字符数据注册回调，需要在 MutationObserverInit 对象中将 characterData 属性设置为 true，如下所示：[①]

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

// 创建要观察的文本节点
document.body.firstChild.textContent = 'foo';

observer.observe(document.body.firstChild, { characterData: true });

// 赋值为相同的字符串
document.body.firstChild.textContent = 'foo';

// 赋值为新字符串
document.body.firstChild.textContent = 'bar';

// 通过节点设置函数赋值
document.body.firstChild.textContent = 'baz';

// 以上变化都被记录下来了
// [MutationRecord, MutationRecord, MutationRecord]
```

将 characterData 属性设置为 true 的默认行为不会在 MutationRecord 对象中记录原来的字符数据。如果想在变化记录中保存原来的字符数据，可以将 characterDataOldValue 属性设置为 true：

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords.map((x) => x.oldValue)));
document.body.innerText = 'foo';

observer.observe(document.body.firstChild, { characterDataOldValue: true });

document.body.innerText = 'foo';
document.body.innerText = 'bar';
```

① 设置元素文本内容的标准方式是 textContent 属性。Element 类也定义了 innerText 属性，与 textContent 类似。但 innerText 的定义不严谨，浏览器间的实现也存在兼容性问题，因此不建议再使用了。——译者注

```
document.body.firstChild.textContent = 'baz';

// 每次变化都保留了上一次的值
// ["foo", "foo", "bar"]
```

3. 观察子节点

MutationObserver 可以观察目标节点子节点的添加和移除。要观察子节点，需要在 MutationObserverInit 对象中将 childList 属性设置为 true。

下面的例子演示了添加子节点：

```
// 清空主体
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { childList: true });

document.body.appendChild(document.createElement('div'));

// [
//   {
//     addedNodes: NodeList[div],
//     attributeName: null,
//     attributeNamespace: null,
//     oldValue: null,
//     nextSibling: null,
//     previousSibling: null,
//     removedNodes: NodeList[],
//     target: body,
//     type: "childList",
//   }
// ]
```

下面的例子演示了移除子节点：

```
// 清空主体
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { childList: true });

document.body.appendChild(document.createElement('div'));

// [
//   {
//     addedNodes: NodeList[],
//     attributeName: null,
//     attributeNamespace: null,
//     oldValue: null,
//     nextSibling: null,
//     previousSibling: null,
//     removedNodes: NodeList[div],
//     target: body,
//     type: "childList",
//   }
// ]
```

对子节点**重新排序**（尽管调用一个方法即可实现）会报告两次变化事件，因为从技术上会涉及先移除和再添加：

```
// 清空主体
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

// 创建两个初始子节点
document.body.appendChild(document.createElement('div'));
document.body.appendChild(document.createElement('span'));

observer.observe(document.body, { childList: true });

// 交换子节点顺序
document.body.insertBefore(document.body.lastChild, document.body.firstChild);

// 发生了两次变化：第一次是节点被移除，第二次是节点被添加
// [
//   {
//     addedNodes: NodeList[],
//     attributeName: null,
//     attributeNamespace: null,
//     oldValue: null,
//     nextSibling: null,
//     previousSibling: div,
//     removedNodes: NodeList[span],
//     target: body,
//       type: childList,
//   },
//   {
//     addedNodes: NodeList[span],
//     attributeName: null,
//     attributeNamespace: null,
//     oldValue: null,
//     nextSibling: div,
//     previousSibling: null,
//     removedNodes: NodeList[],
//     target: body,
//     type: "childList",
//   }
// ]
```

4. 观察子树

默认情况下，`MutationObserver` 将观察的范围限定为一个元素及其子节点的变化。可以把观察的范围扩展到这个元素的子树（所有后代节点），这需要在 `MutationObserverInit` 对象中将 `subtree` 属性设置为 `true`。

下面的代码展示了观察元素及其后代节点属性的变化：

```
// 清空主体
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

// 创建一个后代
document.body.appendChild(document.createElement('div'));
```

```
// 观察<body>元素及其子树
observer.observe(document.body, { attributes: true, subtree: true });

// 修改<body>元素的子树
document.body.firstChild.setAttribute('foo', 'bar');

// 记录了子树变化的事件
// [
//   {
//     addedNodes: NodeList[],
//     attributeName: "foo",
//     attributeNamespace: null,
//     oldValue: null,
//     nextSibling: null,
//     previousSibling: null,
//     removedNodes: NodeList[],
//     target: div,
//     type: "attributes",
//   }
// ]
```

有意思的是，被观察子树中的节点被移出子树之后仍然能够触发变化事件。这意味着在子树中的节点离开该子树后，即使严格来讲该节点已经脱离了原来的子树，但它仍然会触发变化事件。

下面的代码演示了这种情况：

```
// 清空主体
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

let subtreeRoot = document.createElement('div'),
    subtreeLeaf = document.createElement('span');

// 创建包含两层的子树
document.body.appendChild(subtreeRoot);
subtreeRoot.appendChild(subtreeLeaf);

// 观察子树
observer.observe(subtreeRoot, { attributes: true, subtree: true });

// 把节点转移到其他子树
document.body.insertBefore(subtreeLeaf, subtreeRoot);

subtreeLeaf.setAttribute('foo', 'bar');

// 移出的节点仍然触发变化事件
// [MutationRecord]
```

14.3.3　异步回调与记录队列

MutationObserver 接口是出于性能考虑而设计的，其核心是异步回调与记录队列模型。为了在大量变化事件发生时不影响性能，每次变化的信息（由观察者实例决定）会保存在 MutationRecord 实例中，然后添加到**记录队列**。这个队列对每个 MutationObserver 实例都是唯一的，是所有 DOM 变化事件的有序列表。

1. 记录队列

每次 MutationRecord 被添加到 MutationObserver 的记录队列时，仅当之前没有已排期的微任务回调时（队列中微任务长度为 0），才会将观察者注册的回调（在初始化 MutationObserver 时传入）作为微任务调度到任务队列上。这样可以保证记录队列的内容不会被回调处理两次。

不过在回调的微任务异步执行期间，有可能又会发生更多变化事件。因此被调用的回调会接收到一个 MutationRecord 实例的数组，顺序为它们进入记录队列的顺序。回调要负责处理这个数组的每一个实例，因为函数退出之后这些实现就不存在了。回调执行后，这些 MutationRecord 就用不着了，因此记录队列会被清空，其内容会被丢弃。

2. takeRecords()方法

调用 MutationObserver 实例的 takeRecords()方法可以清空记录队列，取出并返回其中的所有 MutationRecord 实例。看这个例子：

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';
document.body.className = 'bar';
document.body.className = 'baz';

console.log(observer.takeRecords());
console.log(observer.takeRecords());

// [MutationRecord, MutationRecord, MutationRecord]
// []
```

这在希望断开与观察目标的联系，但又希望处理由于调用 disconnect()而被抛弃的记录队列中的 MutationRecord 实例时比较有用。

14.3.4 性能、内存与垃圾回收

DOM Level 2 规范中描述的 MutationEvent 定义了一组会在各种 DOM 变化时触发的事件。由于浏览器事件的实现机制，这个接口出现了严重的性能问题。因此，DOM Level 3 规定废弃了这些事件。MutationObserver 接口就是为替代这些事件而设计的更实用、性能更好的方案。

将变化回调委托给微任务来执行可以保证事件同步触发，同时避免随之而来的混乱。为 MutationObserver 而实现的记录队列，可以保证即使变化事件被爆发式地触发，也不会显著地拖慢浏览器。

无论如何，使用 MutationObserver 仍然**不是没有代价**的。因此理解什么时候避免出现这种情况就很重要了。

1. MutationObserver 的引用

MutationObserver 实例与目标节点之间的引用关系是非对称的。MutationObserver 拥有对要观察的目标节点的弱引用。因为是弱引用，所以不会妨碍垃圾回收程序回收目标节点。

然而，目标节点却拥有对 MutationObserver 的强引用。如果目标节点从 DOM 中被移除，随后被垃圾回收，则关联的 MutationObserver 也会被垃圾回收。

2. MutationRecord 的引用

记录队列中的每个 MutationRecord 实例至少包含对已有 DOM 节点的一个引用。如果变化是 childList 类型，则会包含多个节点的引用。记录队列和回调处理的默认行为是耗尽这个队列，处理每个 MutationRecord，然后让它们超出作用域并被垃圾回收。

有时候可能需要保存某个观察者的完整变化记录。保存这些 MutationRecord 实例，也就会保存它们引用的节点，因而会妨碍这些节点被回收。如果需要尽快地释放内存，建议从每个 MutationRecord 中抽取出最有用的信息，然后保存到一个新对象中，最后抛弃 MutationRecord。

14.4　小结

文档对象模型（DOM，Document Object Model）是语言中立的 HTML 和 XML 文档的 API。DOM Level 1 将 HTML 和 XML 文档定义为一个节点的多层级结构，并暴露出 JavaScript 接口以操作文档的底层结构和外观。

DOM 由一系列节点类型构成，主要包括以下几种。

- Node 是基准节点类型，是文档一个部分的抽象表示，所有其他类型都继承 Node。
- Document 类型表示整个文档，对应树形结构的根节点。在 JavaScript 中，document 对象是 Document 的实例，拥有查询和获取节点的很多方法。
- Element 节点表示文档中所有 HTML 或 XML 元素，可以用来操作它们的内容和属性。
- 其他节点类型分别表示文本内容、注释、文档类型、CDATA 区块和文档片段。

DOM 编程在多数情况下没什么问题，在涉及<script>和<style>元素时会有一点兼容性问题。因为这些元素分别包含脚本和样式信息，所以浏览器会将它们与其他元素区别对待。

要理解 DOM，最关键的一点是知道影响其性能的问题所在。DOM 操作在 JavaScript 代码中是代价比较高的，NodeList 对象尤其需要注意。NodeList 对象是“实时更新”的，这意味着每次访问它都会执行一次新的查询。考虑到这些问题，实践中要尽量减少 DOM 操作的数量。

MutationObserver 是为代替性能不好的 MutationEvent 而问世的。使用它可以有效精准地监控 DOM 变化，而且 API 也相对简单。

第 15 章
DOM 扩展

本章内容
- ❑ 理解 Selectors API
- ❑ 使用 HTML5 DOM 扩展

尽管 DOM API 已经相当不错，但仍然不断有标准或专有的扩展出现，以支持更多功能。2008 年以前，大部分浏览器对 DOM 的扩展是专有的。此后，W3C 开始着手将这些已成为事实标准的专有扩展编制成正式规范。

基于以上背景，诞生了描述 DOM 扩展的两个标准：Selectors API 与 HTML5。这两个标准体现了社区需求和标准化某些手段及 API 的愿景。另外还有较小的 Element Traversal 规范，增加了一些 DOM 属性。专有扩展虽然还有，但这两个规范（特别是 HTML5）已经涵盖其中大部分。本章也会讨论专有扩展。

本章所有内容已经得到市场占有率名列前茅的所有主流浏览器支持，除非特别说明。

15.1 Selectors API

JavaScript 库中最流行的一种能力就是根据 CSS 选择符的模式匹配 DOM 元素。比如，jQuery 就完全以 CSS 选择符查询 DOM 获取元素引用，而不是使用 `getElementById()` 和 `getElementsByTagName()`。

Selectors API（参见 W3C 网站上的 Selectors API Level 1）是 W3C 推荐标准，规定了浏览器原生支持的 CSS 查询 API。支持这一特性的所有 JavaScript 库都会实现一个基本的 CSS 解析器，然后使用已有的 DOM 方法搜索文档并匹配目标节点。虽然库开发者在不断改进其性能，但 JavaScript 代码能做到的毕竟有限。通过浏览器原生支持这个 API，解析和遍历 DOM 树可以通过底层编译语言实现，性能也有了数量级的提升。

Selectors API Level 1 的核心是两个方法：`querySelector()` 和 `querySelectorAll()`。在兼容浏览器中，`Document` 类型和 `Element` 类型的实例上都会暴露这两个方法。

Selectors API Level 2 规范在 `Element` 类型上新增了更多方法，比如 `matches()`、`find()` 和 `findAll()`。不过，目前还没有浏览器实现或宣称实现 `find()` 和 `findAll()`。

15.1.1 querySelector()

`querySelector()` 方法接收 CSS 选择符参数，返回匹配该模式的第一个后代元素，如果没有匹配项则返回 `null`。下面是一些例子：

```
// 取得<body>元素
let body = document.querySelector("body");

// 取得 ID 为"myDiv"的元素
let myDiv = document.querySelector("#myDiv");
```