不过，PerformanceEntry 实际上是一个抽象基类。所有记录条目虽然都继承 PerformanceEntry，但最终还是如下某个具体类的实例：

- PerformanceMark
- PerformanceMeasure
- PerformanceFrameTiming
- PerformanceNavigationTiming
- PerformanceResourceTiming
- PerformancePaintTiming

上面每个类都会增加大量属性，用于描述与相应条目有关的元数据。每个实例的 name 和 entryType 属性会因为各自的类不同而不同。

1. User Timing API

User Timing API 用于记录和分析自定义性能条目。如前所述，记录自定义性能条目要使用 performance.mark()方法：

```
performance.mark('foo');

console.log(performance.getEntriesByType('mark')[0]);
// PerformanceMark {
//   name: "foo",
//   entryType: "mark",
//   startTime: 269.8800000362098,
//   duration: 0
// }
```

在计算开始前和结束后各创建一个自定义性能条目可以计算时间差。最新的标记（mark）会被推到 getEntriesByType()返回数组的开始：

```
performance.mark('foo');
for (let i = 0; i < 1E6; ++i) {}
performance.mark('bar');

const [endMark, startMark] = performance.getEntriesByType('mark');
console.log(startMark.startTime - endMark.startTime); // 1.3299999991431832
```

除了自定义性能条目，还可以生成 PerformanceMeasure（性能度量）条目，对应由名字作为标识的两个标记之间的持续时间。PerformanceMeasure 的实例由 performance.measure()方法生成：

```
performance.mark('foo');
for (let i = 0; i < 1E6; ++i) {}
performance.mark('bar');

performance.measure('baz', 'foo', 'bar');

const [differenceMark] = performance.getEntriesByType('measure');

console.log(differenceMark);
// PerformanceMeasure {
//   name: "baz",
//   entryType: "measure",
//   startTime: 298.9800000214018,
//   duration: 1.349999976810068
// }
```

2. Navigation Timing API

Navigation Timing API 提供了高精度时间戳，用于度量当前页面加载速度。浏览器会在导航事件发

生时自动记录 PerformanceNavigationTiming 条目。这个对象会捕获大量时间戳，用于描述页面是何时以及如何加载的。

下面的例子计算了 loadEventStart 和 loadEventEnd 时间戳之间的差：

```
const [performanceNavigationTimingEntry] = performance.getEntriesByType('navigation');

console.log(performanceNavigationTimingEntry);
// PerformanceNavigationTiming {
//   connectEnd: 2.259999979287386
//   connectStart: 2.259999979287386
//   decodedBodySize: 122314
//   domComplete: 631.9899999652989
//   domContentLoadedEventEnd: 300.92499998863786
//   domContentLoadedEventStart: 298.8950000144541
//   domInteractive: 298.88499999651685
//   domainLookupEnd: 2.259999979287386
//   domainLookupStart: 2.259999979287386
//   duration: 632.819999998901
//   encodedBodySize: 21107
//   entryType: "navigation"
//   fetchStart: 2.259999979287386
//   initiatorType: "navigation"
//   loadEventEnd: 632.819999998901
//   loadEventStart: 632.0149999810383
//   name: " https://foo.com "
//   nextHopProtocol: "h2"
//   redirectCount: 0
//   redirectEnd: 0
//   redirectStart: 0
//   requestStart: 7.7099999762140214
//   responseEnd: 130.50999998813495
//   responseStart: 127.16999999247491
//   secureConnectionStart: 0
//   serverTiming: []
//   startTime: 0
//   transferSize: 21806
//   type: "navigate"
//   unloadEventEnd: 132.73999997181818
//   unloadEventStart: 132.41999997990206
//   workerStart: 0
// }

console.log(performanceNavigationTimingEntry.loadEventEnd -
            performanceNavigationTimingEntry.loadEventStart);
// 0.805000017862767
```

3. Resource Timing API

Resource Timing API 提供了高精度时间戳，用于度量当前页面加载时请求资源的速度。浏览器会在加载资源时自动记录 PerformanceResourceTiming。这个对象会捕获大量时间戳，用于描述资源加载的速度。

下面的例子计算了加载一个特定资源所花的时间：

```
const performanceResourceTimingEntry = performance.getEntriesByType('resource')[0];

console.log(performanceResourceTimingEntry);
// PerformanceResourceTiming {
//   connectEnd: 138.11499997973442
```

```
//   connectStart: 138.11499997973442
//   decodedBodySize: 33808
//   domainLookupEnd: 138.11499997973442
//   domainLookupStart: 138.11499997973442
//   duration: 0
//   encodedBodySize: 33808
//   entryType: "resource"
//   fetchStart: 138.11499997973442
//   initiatorType: "link"
//   name: "https://static.foo.com/bar.png",
//   nextHopProtocol: "h2"
//   redirectEnd: 0
//   redirectStart: 0
//   requestStart: 138.11499997973442
//   responseEnd: 138.11499997973442
//   responseStart: 138.11499997973442
//   secureConnectionStart: 0
//   serverTiming: []
//   startTime: 138.11499997973442
//   transferSize: 0
//   workerStart: 0
// }

console.log(performanceResourceTimingEntry.responseEnd -
            performanceResourceTimingEntry.requestStart);
// 493.9600000507198
```

通过计算并分析不同时间的差，可以更全面地审视浏览器加载页面的过程，发现可能存在的性能瓶颈。

20.11　Web 组件

这里所说的 Web 组件指的是一套用于增强 DOM 行为的工具，包括影子 DOM、自定义元素和 HTML 模板。这一套浏览器 API 特别混乱。

- 并没有统一的“Web Components”规范：每个 Web 组件都在一个不同的规范中定义。
- 有些 Web 组件如影子 DOM 和自定义元素，已经出现了向后不兼容的版本问题。
- 浏览器实现极其不一致。

由于存在这些问题，因此使用 Web 组件通常需要引入一个 Web 组件库，比如 Polymer。这种库可以作为腻子脚本，模拟浏览器中缺失的 Web 组件。

> **注意**　本章只介绍 Web 组件的最新版本。

20.11.1　HTML 模板

在 Web 组件之前，一直缺少基于 HTML 解析构建 DOM 子树，然后在需要时再把这个子树渲染出来的机制。一种间接方案是使用 `innerHTML` 把标记字符串转换为 DOM 元素，但这种方式存在严重的安全隐患。另一种间接方案是使用 `document.createElement()` 构建每个元素，然后逐个把它们添加到孤儿根节点（不是添加到 DOM），但这样做特别麻烦，完全与标记无关。

相反，更好的方式是提前在页面中写出特殊标记，让浏览器自动将其解析为 DOM 子树，但跳过渲

染。这正是 HTML 模板的核心思想，而<template>标签正是为这个目的而生的。下面是一个简单的 HTML 模板的例子：

```
<template id="foo">
  <p>I'm inside a template!</p>
</template>
```

1. 使用 DocumentFragment

在浏览器中渲染时，上面例子中的文本不会被渲染到页面上。因为<template>的内容不属于活动文档，所以 document.querySelector()等 DOM 查询方法不会发现其中的<p>标签。这是因为<p>存在于一个包含在 HTML 模板中的 DocumentFragment 节点内。

在浏览器中通过开发者工具检查网页内容时，可以看到<template>中的 DocumentFragment：

```
<template id="foo">
  #document-fragment
    <p>I'm inside a template!</p>
</template>
```

通过<template>元素的 content 属性可以取得这个 DocumentFragment 的引用：

```
console.log(document.querySelector('#foo').content); // #document-fragment
```

此时的 DocumentFragment 就像一个对应子树的最小化 document 对象。换句话说，DocumentFragment 上的 DOM 匹配方法可以查询其子树中的节点：

```
const fragment = document.querySelector('#foo').content;

console.log(document.querySelector('p')); // null
console.log(fragment.querySelector('p')); // <p>...<p>
```

DocumentFragment 也是批量向 HTML 中添加元素的高效工具。比如，我们想以最快的方式给某个 HTML 元素添加多个子元素。如果连续调用 document.appendChild()，则不仅费事，还会导致多次布局重排。而使用 DocumentFragment 可以一次性添加所有子节点，最多只会有一次布局重排：

```
// 开始状态：
// <div id="foo"></div>
//
// 期待的最终状态：
// <div id="foo">
//   <p></p>
//   <p></p>
//   <p></p>
// </div>
// 也可以使用 document.createDocumentFragment()
const fragment = new DocumentFragment();

const foo = document.querySelector('#foo');

// 为 DocumentFragment 添加子元素不会导致布局重排
fragment.appendChild(document.createElement('p'));
fragment.appendChild(document.createElement('p'));
fragment.appendChild(document.createElement('p'));

console.log(fragment.children.length); // 3

foo.appendChild(fragment);
```

```
console.log(fragment.children.length); // 0

console.log(document.body.innerHTML);
// <div id="foo">
//   <p></p>
//   <p></p>
//   <p></p>
// </div>
```

2. 使用<template>标签

注意，在前面的例子中，DocumentFragment 的所有子节点都高效地转移到了 foo 元素上，转移之后 DocumentFragment 变空了。同样的过程也可以使用<template>标签重现：

```
const fooElement = document.querySelector('#foo');
const barTemplate = document.querySelector('#bar');
const barFragment = barTemplate.content;

console.log(document.body.innerHTML);
// <div id="foo">
// </div>
// <template id="bar">
//   <p></p>
//   <p></p>
//   <p></p>
// </template>

fooElement.appendChild(barFragment);

console.log(document.body.innerHTML);
// <div id="foo">
//   <p></p>
//   <p></p>
//   <p></p>
// </div>
// <tempate id="bar"></template>
```

如果想要复制模板，可以使用 importNode()方法克隆 DocumentFragment：

```
const fooElement = document.querySelector('#foo');
const barTemplate = document.querySelector('#bar');
const barFragment = barTemplate.content;

console.log(document.body.innerHTML);
// <div id="foo">
// </div>
// <template id="bar">
//   <p></p>
//   <p></p>
//   <p></p>
// </template>

fooElement.appendChild(document.importNode(barFragment, true));

console.log(document.body.innerHTML);
// <div id="foo">
//   <p></p>
//   <p></p>
//   <p></p>
```

```
// </div>
// <template id="bar">
//   <p></p>
//   <p></p>
//   <p></p>
// </template>
```

3. 模板脚本

脚本执行可以推迟到将 DocumentFragment 的内容实际添加到 DOM 树。下面的例子演示了这个过程：

```
// 页面 HTML:
//
// <div id="foo"></div>
// <template id="bar">
//   <script>console.log('Template script executed');</script>
// </template>

const fooElement = document.querySelector('#foo');
const barTemplate = document.querySelector('#bar');
const barFragment = barTemplate.content;

console.log('About to add template');
fooElement.appendChild(barFragment);
console.log('Added template');

// About to add template
// Template script executed
// Added template
```

如果新添加的元素需要进行某些初始化，这种延迟执行是有用的。

20.11.2 影子 DOM

概念上讲，影子 DOM（shadow DOM）Web 组件相当直观，通过它可以将一个完整的 DOM 树作为节点添加到父 DOM 树。这样可以实现 DOM 封装，意味着 CSS 样式和 CSS 选择符可以限制在影子 DOM 子树而不是整个顶级 DOM 树中。

影子 DOM 与 HTML 模板很相似，因为它们都是类似 document 的结构，并允许与顶级 DOM 有一定程度的分离。不过，影子 DOM 与 HTML 模板还是有区别的，主要表现在影子 DOM 的内容会实际渲染到页面上，而 HTML 模板的内容不会。

1. 理解影子 DOM

假设有以下 HTML 标记，其中包含多个类似的 DOM 子树：

```
<div>
  <p>Make me red!</p>
</div>
<div>
  <p>Make me blue!</p>
</div>
<div>
  <p>Make me green!</p>
</div>
```

从其中的文本节点可以推断出，这 3 个 DOM 子树会分别渲染为不同的颜色。常规情况下，为了给

每个子树应用唯一的样式，又不使用 `style` 属性，就需要给每个子树添加一个唯一的类名，然后通过相应的选择符为它们添加样式：

```
<div class="red-text">
  <p>Make me red!</p>
</div>
<div class="green-text">
  <p>Make me green!</p>
</div>
<div class="blue-text">
  <p>Make me blue!</p>
</div>

<style>
.red-text {
  color: red;
}
.green-text {
  color: green;
}
.blue-text {
  color: blue;
}
</style>
```

当然，这个方案也不是十分理想，因为这跟在全局命名空间中定义变量没有太大区别。尽管知道这些样式与其他地方无关，所有 CSS 样式还会应用到整个 DOM。为此，就要保持 CSS 选择符足够特别，以防这些样式渗透到其他地方。但这也仅是一个折中的办法而已。理想情况下，应该能够把 CSS 限制在使用它们的 DOM 上：这正是影子 DOM 最初的使用场景。

2. 创建影子 DOM

考虑到安全及避免影子 DOM 冲突，并非所有元素都可以包含影子 DOM。尝试给无效元素或者已经有了影子 DOM 的元素添加影子 DOM 会导致抛出错误。

以下是可以容纳影子 DOM 的元素。

- 任何以有效名称创建的自定义元素（参见 HTML 规范中相关的定义）
- `<article>`
- `<aside>`
- `<blockquote>`
- `<body>`
- `<div>`
- `<footer>`
- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`
- `<header>`
- `<main>`
- `<nav>`
- `<p>`

- <section>
-

影子 DOM 是通过 attachShadow()方法创建并添加给有效 HTML 元素的。容纳影子 DOM 的元素被称为**影子宿主**（shadow host）。影子 DOM 的根节点被称为**影子根**（shadow root）。

attachShadow()方法需要一个 shadowRootInit 对象，返回影子 DOM 的实例。shadowRootInit 对象必须包含一个 mode 属性，值为"open"或"closed"。对"open"影子 DOM 的引用可以通过 shadowRoot 属性在 HTML 元素上获得，而对"closed"影子 DOM 的引用无法这样获取。

下面的代码演示了不同 mode 的区别：

```
document.body.innerHTML = `
  <div id="foo"></div>
  <div id="bar"></div>
`;

const foo = document.querySelector('#foo');
const bar = document.querySelector('#bar');

const openShadowDOM = foo.attachShadow({ mode: 'open' });
const closedShadowDOM = bar.attachShadow({ mode: 'closed' });

console.log(openShadowDOM);   // #shadow-root (open)
console.log(closedShadowDOM); // #shadow-root (closed)

console.log(foo.shadowRoot);  // #shadow-root (open)
console.log(bar.shadowRoot);  // null
```

一般来说，需要创建保密（closed）影子 DOM 的场景很少。虽然这可以限制通过影子宿主访问影子 DOM，但恶意代码有很多方法绕过这个限制，恢复对影子 DOM 的访问。简言之，不能为了安全而创建保密影子 DOM。

> **注意** 如果想保护独立的 DOM 树不受未信任代码影响，影子 DOM 并不适合这个需求。对<iframe>施加的跨源限制更可靠。

3. 使用影子 DOM

把影子 DOM 添加到元素之后，可以像使用常规 DOM 一样使用影子 DOM。来看下面的例子，这里重新创建了前面红/绿/蓝子树的示例：

```
for (let color of ['red', 'green', 'blue']) {
  const div = document.createElement('div');
  const shadowDOM = div.attachShadow({ mode: 'open' });

  document.body.appendChild(div);
  shadowDOM.innerHTML = `
    <p>Make me ${color}</p>

    <style>
    p {
      color: ${color};
    }
    </style>
  `;
}
```

虽然这里使用相同的选择符应用了 3 种不同的颜色，但每个选择符只会把样式应用到它们所在的影子 DOM 上。为此，3 个<p>元素会出现 3 种不同的颜色。

可以这样验证这些元素分别位于它们自己的影子 DOM 中：

```
for (let color of ['red', 'green', 'blue']) {
  const div = document.createElement('div');
  const shadowDOM = div.attachShadow({ mode: 'open' });

  document.body.appendChild(div);

  shadowDOM.innerHTML = `
    <p>Make me ${color}</p>

    <style>
    p {
      color: ${color};
    }
    </style>
  `;
}

function countP(node) {
  console.log(node.querySelectorAll('p').length);
}

countP(document); // 0

for (let element of document.querySelectorAll('div')) {
  countP(element.shadowRoot);
}

// 1
// 1
// 1
```

在浏览器开发者工具中可以更清楚地看到影子 DOM。例如，前面的例子在浏览器检查窗口中会显示成这样：

```
<body>
<div>
  #shadow-root (open)
    <p>Make me red!</p>
    <style>
    p {
      color: red;
    }
    </style>
</div>
<div>
  #shadow-root (open)
    <p>Make me green!</p>

    <style>
    p {
      color: green;
    }
    </style>
```

```
</div>
<div>
  #shadow-root (open)
    <p>Make me blue!</p>

    <style>
    p {
      color: blue;
    }
    </style>
</div>
</body>
```

影子 DOM 并非铁板一块。HTML 元素可以在 DOM 树间无限制移动：

```
document.body.innerHTML = `
<div></div>
<p id="foo">Move me</p>
`;

const divElement = document.querySelector('div');
const pElement = document.querySelector('p');

const shadowDOM = divElement.attachShadow({ mode: 'open' });

// 从父 DOM 中移除元素
divElement.parentElement.removeChild(pElement);

// 把元素添加到影子 DOM
shadowDOM.appendChild(pElement);

// 检查元素是否移动到了影子 DOM 中
console.log(shadowDOM.innerHTML); // <p id="foo">Move me</p>
```

4. 合成与影子 DOM 槽位

影子 DOM 是为自定义 Web 组件设计的，为此需要支持嵌套 DOM 片段。从概念上讲，可以这么说：位于影子宿主中的 HTML 需要一种机制以渲染到影子 DOM 中去，但这些 HTML 又不必属于影子 DOM 树。

默认情况下，嵌套内容会隐藏。来看下面的例子，其中的文本在 1000 毫秒后会被隐藏：

```
document.body.innerHTML = `
<div>
  <p>Foo</p>
</div>
`;

setTimeout(() => document.querySelector('div').attachShadow({ mode: 'open' }), 1000);
```

影子 DOM 一添加到元素中，浏览器就会赋予它最高优先级，优先渲染它的内容而不是原来的文本。在这个例子中，由于影子 DOM 是空的，因此 会在 1000 毫秒后变成空的。

为了显示文本内容，需要使用<slot>标签指示浏览器在哪里放置原来的 HTML。下面的代码修改了前面的例子，让影子宿主中的文本出现在了影子 DOM 中：

```
document.body.innerHTML = `
<div id="foo">
  <p>Foo</p>
</div>
`;
```