```
document.querySelector('div')
    .attachShadow({ mode: 'open' })
    .innerHTML = `<div id="bar">
                    <slot></slot>
                  <div>`
```

现在，投射进去的内容就像自己存在于影子 DOM 中一样。检查页面会发现原来的内容实际上替代了<slot>：

```
<body>
<div id="foo">
  #shadow-root (open)
    <div id="bar">
      <p>Foo</p>
    </div>
</div>
</body>
```

注意，虽然在页面检查窗口中看到内容在影子 DOM 中，但这实际上只是 DOM 内容的**投射**（projection）。实际的元素仍然处于外部 DOM 中：

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

document.querySelector('div')
    .attachShadow({ mode: 'open' })
    .innerHTML = `
      <div id="bar">
        <slot></slot>
      </div>`

console.log(document.querySelector('p').parentElement);
// <div id="foo"></div>
```

下面是使用槽位（slot）改写的前面红/绿/蓝子树的例子：

```
for (let color of ['red', 'green', 'blue']) {
  const divElement = document.createElement('div');
  divElement.innerText = `Make me ${color}`;
  document.body.appendChild(divElement)

  divElement
      .attachShadow({ mode: 'open' })
      .innerHTML = `
      <p><slot></slot></p>

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

除了默认槽位，还可以使用**命名槽位**（named slot）实现多个投射。这是通过匹配的 slot/name 属

性对实现的。带有 slot="foo"属性的元素会被投射到带有 name="foo"的<slot>上。下面的例子演示了如何改变影子宿主子元素的渲染顺序：

```
document.body.innerHTML = `
<div>
  <p slot="foo">Foo</p>
  <p slot="bar">Bar</p>
</div>
`;

document.querySelector('div')
    .attachShadow({ mode: 'open' })
    .innerHTML = `
    <slot name="bar"></slot>
    <slot name="foo"></slot>
    `;

// Renders:
// Bar
// Foo
```

5. 事件重定向

如果影子 DOM 中发生了浏览器事件(如 click),那么浏览器需要一种方式以让父 DOM 处理事件。不过,实现也必须考虑影子 DOM 的边界。为此,事件会逃出影子 DOM 并经过**事件重定向**(event retarget) 在外部被处理。逃出后，事件就好像是由影子宿主本身而非真正的包装元素触发的一样。下面的代码演示了这个过程：

```
// 创建一个元素作为影子宿主
document.body.innerHTML = `
<div onclick="console.log('Handled outside:', event.target)"></div>
`;

// 添加影子 DOM 并向其中插入 HTML
document.querySelector('div')
  .attachShadow({ mode: 'open' })
  .innerHTML = `
<button onclick="console.log('Handled inside:', event.target)">Foo</button>
`;

// 点击按钮时:
// Handled inside:  <button onclick="..."></button>
// Handled outside: <div onclick="..."></div>
```

注意，事件重定向只会发生在影子 DOM 中实际存在的元素上。使用<slot>标签从外部投射进来的元素不会发生事件重定向，因为从技术上讲，这些元素仍然存在于影子 DOM 外部。

20.11.3 自定义元素

如果你使用 JavaScript 框架，那么很可能熟悉自定义元素的概念。这是因为所有主流框架都以某种形式提供了这个特性。自定义元素为 HTML 元素引入了面向对象编程的风格。基于这种风格，可以创建自定义的、复杂的和可重用的元素，而且只要使用简单的 HTML 标签或属性就可以创建相应的实例。

1. 创建自定义元素

浏览器会尝试将无法识别的元素作为通用元素整合进 DOM。当然，这些元素默认也不会做任何通

用 HTML 元素不能做的事。来看下面的例子，其中胡乱编的 HTML 标签会变成一个 HTMLElement 实例：

```
document.body.innerHTML = `
<x-foo >I'm inside a nonsense element.</x-foo >
`;

console.log(document.querySelector('x-foo') instanceof HTMLElement); // true
```

自定义元素在此基础上更进一步。利用自定义元素，可以在<x-foo>标签出现时为它定义复杂的行为，同样也可以在 DOM 中将其纳入元素生命周期管理。自定义元素要使用全局属性 customElements，这个属性会返回 CustomElementRegistry 对象。

```
console.log(customElements); // CustomElementRegistry {}
```

调用 customElements.define()方法可以创建自定义元素。下面的代码创建了一个简单的自定义元素，这个元素继承 HTMLElement：

```
class FooElement extends HTMLElement {}
customElements.define('x-foo', FooElement);

document.body.innerHTML = `
<x-foo >I'm inside a nonsense element.</x-foo >
`;

console.log(document.querySelector('x-foo') instanceof FooElement); // true
```

> **注意**　自定义元素名必须至少包含一个不在名称开头和末尾的连字符，而且元素标签不能自关闭。

自定义元素的威力源自类定义。例如，可以通过调用自定义元素的构造函数来控制这个类在 DOM 中每个实例的行为：

```
class FooElement extends HTMLElement {
  constructor() {
    super();
    console.log('x-foo')
  }
}
customElements.define('x-foo', FooElement);

document.body.innerHTML = `
<x-foo></x-foo>
<x-foo></x-foo>
<x-foo></x-foo>
`;

// x-foo
// x-foo
// x-foo
```

> **注意**　在自定义元素的构造函数中必须始终先调用 super()。如果元素继承了 HTMLElement 或相似类型而不会覆盖构造函数，则没有必要调用 super()，因为原型构造函数默认会做这件事。很少有创建自定义元素而不继承 HTMLElement 的。

如果自定义元素继承了一个元素类,那么可以使用 is 属性和 extends 选项将标签指定为该自定义元素的实例:

```
class FooElement extends HTMLDivElement {
  constructor() {
    super();
    console.log('x-foo')
  }
}
customElements.define('x-foo', FooElement, { extends: 'div' });

document.body.innerHTML = `
<div is="x-foo"></div>
<div is="x-foo"></div>
<div is="x-foo"></div>
`;

// x-foo
// x-foo
// x-foo
```

2. 添加 Web 组件内容

因为每次将自定义元素添加到 DOM 中都会调用其类构造函数，所以很容易自动给自定义元素添加子 DOM 内容。虽然不能在构造函数中添加子 DOM（会抛出 DOMException），但可以为自定义元素添加影子 DOM 并将内容添加到这个影子 DOM 中：

```
class FooElement extends HTMLElement {
  constructor() {
    super();

    // this 引用 Web 组件节点
    this.attachShadow({ mode: 'open' });

    this.shadowRoot.innerHTML = `
      <p>I'm inside a custom element!</p>
    `;
  }
}
customElements.define('x-foo', FooElement);

document.body.innerHTML += `<x-foo></x-foo`;

// 结果 DOM:
// <body>
// <x-foo>
//   #shadow-root (open)
//     <p>I'm inside a custom element!</p>
// <x-foo>
// </body>
```

为避免字符串模板和 innerHTML 不干净，可以使用 HTML 模板和 document.createElement() 重构这个例子：

```
// (初始的 HTML)
// <template id="x-foo-tpl">
//   <p>I'm inside a custom element template!</p>
// </template>
```

```
const template = document.querySelector('#x-foo-tpl');

class FooElement extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });

    this.shadowRoot.appendChild(template.content.cloneNode(true));
  }
}
customElements.define('x-foo', FooElement);

document.body.innerHTML += `<x-foo></x-foo`;

// 结果 DOM:
// <body>
// <template id="x-foo-tpl">
//   <p>I'm inside a custom element template!</p>
// </template>
// <x-foo>
//   #shadow-root (open)
//     <p>I'm inside a custom element template!</p>
// <x-foo>
// </body>
```

这样可以在自定义元素中实现高度的 HTML 和代码重用，以及 DOM 封装。使用这种模式能够自由创建可重用的组件而不必担心外部 CSS 污染组件的样式。

3. 使用自定义元素生命周期方法

可以在自定义元素的不同生命周期执行代码。带有相应名称的自定义元素类的实例方法会在不同生命周期阶段被调用。自定义元素有以下 5 个生命周期方法。

- `constructor()`：在创建元素实例或将已有 DOM 元素升级为自定义元素时调用。
- `connectedCallback()`：在每次将这个自定义元素实例添加到 DOM 中时调用。
- `disconnectedCallback()`：在每次将这个自定义元素实例从 DOM 中移除时调用。
- `attributeChangedCallback()`：在每次**可观察属性**的值发生变化时调用。在元素实例初始化时，初始值的定义也算一次变化。
- `adoptedCallback()`：在通过 `document.adoptNode()`将这个自定义元素实例移动到新文档对象时调用。

下面的例子演示了这些构建、连接和断开连接的回调：

```
class FooElement extends HTMLElement {
  constructor() {
    super();
    console.log('ctor');
  }

  connectedCallback() {
    console.log('connected');
  }

  disconnectedCallback() {
    console.log('disconnected');
  }
```

```
}
customElements.define('x-foo', FooElement);

const fooElement = document.createElement('x-foo');
// ctor

document.body.appendChild(fooElement);
// connected

document.body.removeChild(fooElement);
// disconnected
```

4. 反射自定义元素属性

自定义元素既是 DOM 实体又是 JavaScript 对象，因此两者之间应该同步变化。换句话说，对 DOM 的修改应该反映到 JavaScript 对象，反之亦然。要从 JavaScript 对象反射到 DOM，常见的方式是使用获取函数和设置函数。下面的例子演示了在 JavaScript 对象和 DOM 之间反射 bar 属性的过程：

```
document.body.innerHTML = `<x-foo></x-foo>`;

class FooElement extends HTMLElement {
  constructor() {
    super();

    this.bar = true;
  }

  get bar() {
    return this.getAttribute('bar');
  }

  set bar(value) {
    this.setAttribute('bar', value)
  }
}
customElements.define('x-foo', FooElement);

console.log(document.body.innerHTML);
// <x-foo bar="true"></x-foo>
```

另一个方向的反射（从 DOM 到 JavaScript 对象）需要给相应的属性添加监听器。为此，可以使用 observedAttributes()获取函数让自定义元素的属性值每次改变时都调用 attributeChanged-Callback()：

```
class FooElement extends HTMLElement {
  static get observedAttributes() {
    // 返回应该触发 attributeChangedCallback()执行的属性
    return ['bar'];
  }

  get bar() {
    return this.getAttribute('bar');
  }

  set bar(value) {
    this.setAttribute('bar', value)
  }
```

```
  attributeChangedCallback(name, oldValue, newValue) {
        if (oldValue !== newValue) {
        console.log(`${oldValue} -> ${newValue}`);

        this[name] = newValue;
    }
  }
}
customElements.define('x-foo', FooElement);

document.body.innerHTML = `<x-foo bar="false"></x-foo>`;
// null -> false

document.querySelector('x-foo').setAttribute('bar', true);
// false -> true
```

5. 升级自定义元素

并非始终可以先定义自定义元素，然后再在 DOM 中使用相应的元素标签。为解决这个先后次序问题，Web 组件在 CustomElementRegistry 上额外暴露了一些方法。这些方法可以用来检测自定义元素是否定义完成，然后可以用它来升级已有元素。

如果自定义元素已经有定义，那么 CustomElementRegistry.get()方法会返回相应自定义元素的类。类似地，CustomElementRegistry.whenDefined()方法会返回一个期约，当相应自定义元素有定义之后解决：

```
customElements.whenDefined('x-foo').then(() => console.log('defined!'));

console.log(customElements.get('x-foo'));
// undefined

customElements.define('x-foo', class {});
// defined!

console.log(customElements.get('x-foo'));
// class FooElement {}
```

连接到 DOM 的元素在自定义元素有定义时会**自动升级**。如果想在元素连接到 DOM 之前强制升级，可以使用 CustomElementRegistry.upgrade()方法：

```
// 在自定义元素有定义之前会创建 HTMLUnknownElement 对象
const fooElement = document.createElement('x-foo');

// 创建自定义元素
class FooElement extends HTMLElement {}
customElements.define('x-foo', FooElement);

console.log(fooElement instanceof FooElement); // false

// 强制升级
customElements.upgrade(fooElement);

console.log(fooElement instanceof FooElement); // true
```

> **注意**　还有一个 HTML Imports Web 组件，但这个规范目前还是草案，没有主要浏览器支持。浏览器最终是否会支持这个规范目前还是未知数。

20.12 Web Cryptography API

Web Cryptography API 描述了一套密码学工具，规范了 JavaScript 如何以安全和符合惯例的方式实现加密。这些工具包括生成、使用和应用加密密钥对，加密和解密消息，以及可靠地生成随机数。

> **注意** 加密接口的组织方式有点奇怪，其外部是一个 `Crypto` 对象，内部是一个 `SubtleCrypto` 对象。在 Web Cryptography API 标准化之前，`window.crypto` 属性在不同浏览器中的实现差异非常大。为实现跨浏览器兼容，标准 API 都暴露在 `SubtleCrypto` 对象上。

20.12.1 生成随机数

在需要生成随机值时，很多人会使用 `Math.random()`。这个方法在浏览器中是以**伪随机数生成器**（PRNG，PseudoRandom Number Generator）方式实现的。所谓“伪”指的是生成值的过程不是真的随机。PRNG 生成的值只是**模拟了**随机的特性。浏览器的 PRNG 并未使用真正的随机源，只是对一个内部状态应用了固定的算法。每次调用 `Math.random()`，这个内部状态都会被一个算法修改，而结果会被转换为一个新的随机值。例如，V8 引擎使用了一个名为 xorshift128+的算法来执行这种修改。

由于算法本身是固定的，其输入**只是**之前的状态，因此随机数顺序也是确定的。xorshift128+使用 128 位内部状态，而算法的设计让任何初始状态在重复自身之前都会产生 $2^{128}-1$ 个伪随机值。这种循环被称为**置换循环**（permutation cycle），而这个循环的长度被称为一个**周期**（period）。很明显，如果攻击者知道 PRNG 的内部状态，就可以预测后续生成的伪随机值。如果开发者无意中使用 PRNG 生成了私有密钥用于加密，则攻击者就可以利用 PRNG 的这个特性算出私有密钥。

伪随机数生成器主要用于快速计算出看起来随机的值。不过并不适合用于加密计算。为解决这个问题，**密码学安全伪随机数生成器**（CSPRNG，Cryptographically Secure PseudoRandom Number Generator）额外增加了一个熵作为输入，例如测试硬件时间或其他无法预计行为的系统特性。这样一来，计算速度明显比常规 PRNG 慢很多，但 CSPRNG 生成的值就很难预测，可以用于加密了。

Web Cryptography API 引入了 CSPRNG，这个 CSPRNG 可以通过 `crypto.getRandomValues()` 在全局 `Crypto` 对象上访问。与 `Math.random()` 返回一个介于 0 和 1 之间的浮点数不同，`getRandomValues()` 会把随机值写入作为参数传给它的定型数组。定型数组的类不重要，因为底层缓冲区会被随机的二进制位填充。

下面的例子展示了生成 5 个 8 位随机值：

```
const array = new Uint8Array(1);

for (let i=0; i<5; ++i) {
  console.log(crypto.getRandomValues(array));
}

// Uint8Array [41]
// Uint8Array [250]
// Uint8Array [51]
// Uint8Array [129]
// Uint8Array [35]
```

`getRandomValues()` 最多可以生成 2^{16}（65 536）字节，超出则会抛出错误：

```
const fooArray = new Uint8Array(2 ** 16);
console.log(window.crypto.getRandomValues(fooArray)); // Uint32Array(16384) [...]

const barArray = new Uint8Array((2 ** 16) + 1);
console.log(window.crypto.getRandomValues(barArray)); // Error
```

要使用 CSPRNG 重新实现 `Math.random()`，可以通过生成一个随机的 32 位数值，然后用它去除最大的可能值 `0xFFFFFFFF`。这样就会得到一个介于 0 和 1 之间的值：

```
function randomFloat() {
  // 生成 32 位随机值
  const fooArray = new Uint32Array(1);

  // 最大值是 2^32 -1
  const maxUint32 = 0xFFFFFFFF;

  // 用最大可能的值来除
  return crypto.getRandomValues(fooArray)[0] / maxUint32;
}

console.log(randomFloat()); // 0.5033651619458955
```

20.12.2　使用 SubtleCrypto 对象

Web Cryptography API 重头特性都暴露在了 `SubtleCrypto` 对象上，可以通过 `window.crypto.subtle` 访问：

```
console.log(crypto.subtle); // SubtleCrypto {}
```

这个对象包含一组方法，用于执行常见的密码学功能，如加密、散列、签名和生成密钥。因为所有密码学操作都在原始二进制数据上执行，所以 `SubtleCrypto` 的每个方法都要用到 `ArrayBuffer` 和 `ArrayBufferView` 类型。由于字符串是密码学操作的重要应用场景，因此 `TextEncoder` 和 `TextDecoder` 是经常与 `SubtleCrypto` 一起使用的类，用于实现二进制数据与字符串之间的相互转换。

> **注意**　`SubtleCrypto` 对象只能在安全上下文（https）中使用。在不安全的上下文中，`subtle` 属性是 `undefined`。

1. 生成密码学摘要

计算数据的密码学摘要是非常常用的密码学操作。这个规范支持 4 种摘要算法：SHA-1 和 3 种 SHA-2。

- SHA-1（Secure Hash Algorithm 1）：架构类似 MD5 的散列函数。接收任意大小的输入，生成 160 位消息散列。由于容易受到碰撞攻击，这个算法已经不再安全。
- SHA-2（Secure Hash Algorithm 2）：构建于相同耐碰撞单向压缩函数之上的一套散列函数。规范支持其中 3 种：SHA-256、SHA-384 和 SHA-512。生成的消息摘要可以是 256 位（SHA-256）、384 位（SHA-384）或 512 位（SHA-512）。这个算法被认为是安全的，广泛应用于很多领域和协议，包括 TLS、PGP 和加密货币（如比特币）。

`SubtleCrypto.digest()` 方法用于生成消息摘要。要使用的散列算法通过字符串 `"SHA-1"`、`"SHA-256"`、`"SHA-384"` 或 `"SHA-512"` 指定。下面的代码展示了一个使用 SHA-256 为字符串 `"foo"` 生

成消息摘要的例子：

```
(async function() {
  const textEncoder = new TextEncoder();
  const message = textEncoder.encode('foo');
  const messageDigest = await crypto.subtle.digest('SHA-256', message);

  console.log(new Uint32Array(messageDigest));
})();

// Uint32Array(8) [1806968364, 2412183400, 1011194873, 876687389,
//                 1882014227, 2696905572, 2287897337, 2934400610]
```

通常，在使用时，二进制的消息摘要会转换为十六进制字符串格式。通过将二进制数据按 8 位进行分割，然后再调用 `toString(16)` 就可以把任何数组缓冲区转换为十六进制字符串：

```
(async function() {
  const textEncoder = new TextEncoder();
  const message = textEncoder.encode('foo');
  const messageDigest = await crypto.subtle.digest('SHA-256', message);

  const hexDigest = Array.from(new Uint8Array(messageDigest))
    .map((x) => x.toString(16).padStart(2, '0'))
    .join('');

  console.log(hexDigest);
})();

// 2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae
```

软件公司通常会公开自己软件二进制安装包的摘要，以便用户验证自己下载到的确实是该公司发布的版本（而不是被恶意软件篡改过的版本）。下面的例子演示了下载 Firefox v67.0，通过 SHA-512 计算其散列，再下载其 SHA-512 二进制验证摘要，最后检查两个十六进制字符串匹配：

```
(async function() {
  const mozillaCdnUrl = '// download-
origin.cdn.mozilla.net/pub/firefox/releases/67.0 /';
  const firefoxBinaryFilename = 'linux-x86_64/en-US/firefox-67.0.tar.bz2';
  const firefoxShaFilename = 'SHA512SUMS';

  console.log('Fetching Firefox binary...');
  const fileArrayBuffer = await (await fetch(mozillaCdnUrl + firefoxBinaryFilename))
      .arrayBuffer();

  console.log('Calculating Firefox digest...');
  const firefoxBinaryDigest = await crypto.subtle.digest('SHA-512', fileArrayBuffer);
  const firefoxHexDigest = Array.from(new Uint8Array(firefoxBinaryDigest))
    .map((x) => x.toString(16).padStart(2, '0'))
    .join('');

  console.log('Fetching published binary digests...');
  // SHA 文件包含此次发布的所有 Firefox 二进制文件的摘要，
  // 因此要根据其格式进制拆分
  const shaPairs = (await (await fetch(mozillaCdnUrl + firefoxShaFilename)).text())
      .split(/\n/).map((x) => x.split(/\s+/));

  let verified = false;
```