考虑到导出多个值是常见的操作，ES6 模块也支持对导出声明分组，可以同时为部分或全部导出值指定别名：

```
const foo = 'foo';
const bar = 'bar';
const baz = 'baz';
export { foo, bar as myBar, baz };
```

默认导出（default export）就好像模块与被导出的值是一回事。默认导出使用 `default` 关键字将一个值声明为默认导出，每个模块只能有一个默认导出。重复的默认导出会导致 `SyntaxError`。

下面的例子定义了一个默认导出，外部模块可以导入这个模块，而这个模块本身就是 `foo` 的值：

```
const foo = 'foo';
export default foo;
```

另外，ES6 模块系统会识别作为别名提供的 `default` 关键字。此时，虽然对应的值是使用命名语法导出的，实际上则会成为默认导出：

```
const foo = 'foo';

// 等同于 export default foo;
export { foo as default };
```

因为命名导出和默认导出不会冲突，所以 ES6 支持在一个模块中同时定义这两种导出：

```
const foo = 'foo';
const bar = 'bar';

export { bar };
export default foo;
```

这两个 `export` 语句可以组合为一行：

```
const foo = 'foo';
const bar = 'bar';

export { foo as default, bar };
```

ES6 规范对不同形式的 `export` 语句中可以使用什么不可以使用什么规定了限制。某些形式允许声明和赋值，某些形式只允许表达式，而某些形式则只允许简单标识符。注意，有的形式使用了分号，有的则没有：

```
// 命名行内导出
export const baz = 'baz';
export const foo = 'foo', bar = 'bar';
export function foo() {}
export function* foo() {}
export class Foo {}

// 命名子句导出
export { foo };
export { foo, bar };
export { foo as myFoo, bar };

// 默认导出
export default 'foo';
export default 123;
export default /[a-z]*/;
export default { foo: 'foo' };
```

```
export { foo, bar as default };
export default foo
export default function() {}
export default function foo() {}
export default function*() {}
export default class {}

// 会导致错误的不同形式:

// 行内默认导出中不能出现变量声明
export default const foo = 'bar';

// 只有标识符可以出现在 export 子句中
export { 123 as foo }

// 别名只能在 export 子句中出现
export const foo = 'foo' as myFoo;
```

> **注意** 什么可以或不可以与 exprot 关键字出现在同一行可能很难记住。一般来说，声明、赋值和导出标识符最好分开。这样就不容易搞错了，同时也可以让 export 语句集中在一块。

26.4.5 模块导入

模块可以通过使用 import 关键字使用其他模块导出的值。与 export 类似，import 必须出现在模块的顶级：

```
// 允许
import ...

// 不允许
if (condition) {
   import ...
}
```

import 语句被提升到模块顶部。因此，与 export 关键字类似，import 语句与使用导入值的语句的相对位置并不重要。不过，还是推荐把导入语句放在模块顶部。

```
// 允许
import { foo } from './fooModule.js';
console.log(foo); // 'foo'

// 允许，但应该避免
console.log(foo); // 'foo'
import { foo } from './fooModule.js';
```

模块标识符可以是相对于当前模块的相对路径，也可以是指向模块文件的绝对路径。它必须是纯字符串，不能是动态计算的结果。例如，不能是拼接的字符串。

如果在浏览器中通过标识符原生加载模块，则文件必须带有.js 扩展名，不然可能无法正确解析。不过，如果是通过构建工具或第三方模块加载器打包或解析的 ES6 模块，则可能不需要包含文件扩展名。

```
// 解析为/components/bar.js
import ... from './bar.js';
```

```
// 解析为/bar.js
import ... from '../bar.js';

// 解析为/bar.js
import ... from '/bar.js';
```

不是必须通过导出的成员才能导入模块。如果不需要模块的特定导出，但仍想加载和执行模块以利用其副作用，可以只通过路径加载它：

```
import './foo.js';
```

导入对模块而言是只读的，实际上相当于 const 声明的变量。在使用*执行批量导入时，赋值给别名的命名导出就好像使用 Object.freeze()冻结过一样。直接修改导出的值是不可能的，但可以修改导出对象的属性。同样，也不能给导出的集合添加或删除导出的属性。要修改导出的值，必须使用有内部变量和属性访问权限的导出方法。

```
import foo, * as Foo './foo.js';

foo = 'foo';     // 错误

Foo.foo = 'foo'; // 错误

foo.bar = 'bar'; // 允许
```

命名导出和默认导出的区别也反映在它们的导入上。命名导出可以使用*批量获取并赋值给保存导出集合的别名，而无须列出每个标识符：

```
const foo = 'foo', bar = 'bar', baz = 'baz';
export { foo, bar, baz }
import * as Foo from './foo.js';

console.log(Foo.foo); // foo
console.log(Foo.bar); // bar
console.log(Foo.baz); // baz
```

要指名导入，需要把标识符放在 import 子句中。使用 import 子句可以为导入的值指定别名：

```
import { foo, bar, baz as myBaz } from './foo.js';

console.log(foo);   // foo
console.log(bar);   // bar
console.log(myBaz); // baz
```

默认导出就好像整个模块就是导出的值一样。可以使用 default 关键字并提供别名来导入。也可以不使用大括号，此时指定的标识符就是默认导出的别名：

```
// 等效
import { default as foo } from './foo.js';
import foo from './foo.js';
```

如果模块同时导出了命名导出和默认导出，则可以在 import 语句中同时取得它们。可以依次列出特定导出的标识符来取得，也可以使用*来取得：

```
import foo, { bar, baz } from './foo.js';

import { default as foo, bar, baz } from './foo.js';

import foo, * as Foo from './foo.js';
```

> **注意** 本书写作时，有一个动态导入模块的提案处在第三阶段（stage 3），参见 GitHub 上的 tc39/proposals 页面。

26.4.6 模块转移导出

模块导入的值可以直接通过管道转移到导出。此时，也可以将默认导出转换为命名导出，或者相反。如果想把一个模块的所有命名导出集中在一块，可以像下面这样在 bar.js 中使用*导出：

```
export * from './foo.js';
```

这样，foo.js 中的所有命名导出都会出现在导入 bar.js 的模块中。如果 foo.js 有默认导出，则该语法会忽略它。使用此语法也要注意导出名称是否冲突。如果 foo.js 导出 baz，bar.js 也导出 baz，则最终导出的是 bar.js 中的值。这个“重写”是静默发生的：

foo.js

```
export const baz = 'origin:foo';
```

bar.js

```
export * from './foo.js';
export const baz = 'origin:bar';
```

main.js

```
import { baz } from './bar.js';
console.log(baz); // origin:bar
```

此外也可以明确列出要从外部模块转移本地导出的值。该语法支持使用别名：

```
export { foo, bar as myBar } from './foo.js';
```

类似地，外部模块的默认导出可以重用为当前模块的默认导出：

```
export { default } from './foo.js';
```

这样不会复制导出的值，只是把导入的引用传给了原始模块。在原始模块中，导入的值仍然是可用的，与修改导入相关的限制也适用于再次导出的导入。

在重新导出时，还可以在导入模块修改命名或默认导出的角色。比如，可以像下面这样将命名导出指定为默认导出：

```
export { foo as default } from './foo.js';
```

26.4.7 工作者模块

ECMAScript 6 模块与 Worker 实例完全兼容。在实例化时，可以给工作者传入一个指向模块文件的路径，与传入常规脚本文件一样。Worker 构造函数接收第二个参数，用于说明传入的是模块文件。

下面是两种类型的 Worker 的实例化行为：

```
// 第二个参数默认为{ type: 'classic' }
const scriptWorker = new Worker('scriptWorker.js');

const moduleWorker = new Worker('moduleWorker.js', { type: 'module' });
```

在基于模块的工作者内部，self.importScripts()方法通常用于在基于脚本的工作者中加载外部脚本，调用它会抛出错误。这是因为模块的 import 行为包含了 importScripts()。

26.4.8　向后兼容

ECMAScript 模块的兼容是个渐进的过程，能够同时兼容支持和不支持的浏览器对早期采用者是有价值的。对于想要尽可能在浏览器中原生使用 ECMAScript 6 模块的用户，可以提供两个版本的代码：基于模块的版本与基于脚本的版本。如果嫌麻烦，可以使用第三方模块系统（如 SystemJS）或在构建时将 ES6 模块进行转译，这都是不错的方案。

第一种方案涉及在服务器上检查浏览器的用户代理，与支持模块的浏览器名单进行匹配，然后基于匹配结果决定提供哪个版本的 JavaScript 文件。这个方法不太可靠，而且比较麻烦，不推荐。更好、更优雅的方案是利用脚本的 `type` 属性和 `nomodule` 属性。

浏览器在遇到`<script>`标签上无法识别的 `type` 属性时会拒绝执行其内容。对于不支持模块的浏览器，这意味着`<script type="module">`不会被执行。因此，可以在`<script type="module">`标签旁边添加一个回退`<script>`标签：

```
// 不支持模块的浏览器不会执行这里的代码
<script type="module" src="module.js"></script>

// 不支持模块的浏览器会执行这里的代码
<script src="script.js"></script>
```

当然，这样一来支持模块的浏览器就有麻烦了。此时，前面的代码会执行两次，显然这不是我们想要的结果。为了避免这种情况，原生支持 ECMAScript 6 模块的浏览器也会识别 `nomodule` 属性。此属性通知支持 ES6 模块的浏览器不执行脚本。不支持模块的浏览器无法识别该属性，从而忽略这个属性的存在。

因此，下面代码会生成一个设置，在这个设置中，支持模块和不支持模块的浏览器都只会执行一段脚本：

```
// 支持模块的浏览器会执行这段脚本
// 不支持模块的浏览器不会执行这段脚本
<script type="module" src="module.js"></script>

// 支持模块的浏览器不会执行这段脚本
// 不支持模块的浏览器会执行这段脚本
<script nomodule src="script.js"></script>
```

26.5　小结

模块模式是管理复杂性的永恒工具。开发者可以通过它创建逻辑彼此独立的代码段，在这些代码段之间声明依赖，并将它们连接在一起。此外，这种模式也是经证明能够优雅扩展到任意复杂度且跨平台的方案。

多年以来，CommonJS 和 AMD 这两个分别针对服务器端环境和受延迟限制的客户端环境的模块系统长期分裂。两个系统都获得了爆炸性增强，但为它们编写的代码则在很多方面不一致，经常也会带有冗余的样板代码。而且，这两个系统都没有在浏览器中实现。缺乏兼容导致出现了相关工具，从而让在浏览器中实现模块模式成为可能。

ECMAScript 6 规范重新定义了浏览器模块，集之前两个系统之长于一身，并通过更简单的声明性语法暴露出来。浏览器对原生模块的支持越来越好，但也提供了稳健的工具以实现从不支持到支持 ES6 模块的过渡。

第27章 工作者线程

本章内容

- 工作者线程简介
- 使用专门的工作者线程执行后台任务
- 使用共享的工作者线程
- 通过服务工作者线程管理请求

前端开发者常说："JavaScript 是单线程的。"这种说法虽然有些简单，但描述了 JavaScript 在浏览器中的一般行为。因此，作为帮助 Web 开发人员理解 JavaScript 的教学工具，它非常有用。

单线程就意味着不能像多线程语言那样把工作委托给独立的线程或进程去做。JavaScript 的单线程可以保证它与不同浏览器 API 兼容。假如 JavaScript 可以多线程执行并发更改，那么像 DOM 这样的 API 就会出现问题。因此，POSIX 线程或 Java 的 `Thread` 类等传统并发结构都不适合 JavaScript。

而这也正是工作者线程的价值所在：允许把主线程的工作转嫁给独立的实体，而不会改变现有的单线程模型。虽然本章要介绍的各种工作者线程有不同的形式和功能，但它们的共同的特点是都独立于 JavaScript 的主执行环境。

27.1 工作者线程简介

JavaScript 环境实际上是运行在托管操作系统中的虚拟环境。在浏览器中每打开一个页面，就会分配一个它自己的环境。这样，每个页面都有自己的内存、事件循环、DOM，等等。每个页面就相当于一个沙盒，不会干扰其他页面。对于浏览器来说，同时管理多个环境是非常简单的，因为所有这些环境都是**并行**执行的。

使用**工作者线程**，浏览器可以在原始页面环境之外再分配一个完全独立的二级子环境。这个子环境不能与依赖单线程交互的 API（如 DOM）互操作，但可以与父环境并行执行代码。

27.1.1 工作者线程与线程

作为介绍，通常需要将工作者线程与执行线程进行比较。在许多方面，这是一个恰当的比较，因为工作者线程和线程确实有很多共同之处。

- **工作者线程是以实际线程实现的**。例如，Blink 浏览器引擎实现工作者线程的 `WorkerThread` 就对应着底层的线程。
- **工作者线程并行执行**。虽然页面和工作者线程都是单线程 JavaScript 环境，每个环境中的指令则可以并行执行。

- **工作者线程可以共享某些内存**。工作者线程能够使用 `SharedArrayBuffer` 在多个环境间共享内容。虽然线程会使用锁实现并发控制，但 JavaScript 使用 `Atomics` 接口实现并发控制。

工作者线程与线程有很多类似之处，但也有重要的区别。

- **工作者线程不共享全部内存**。在传统线程模型中，多线程有能力读写共享内存空间。除了 `SharedArrayBuffer` 外，从工作者线程进出的数据需要复制或转移。
- **工作者线程不一定在同一个进程里**。通常，一个进程可以在内部产生多个线程。根据浏览器引擎的实现，工作者线程可能与页面属于同一进程，也可能不属于。例如，Chrome 的 Blink 引擎对共享工作者线程和服务工作者线程使用独立的进程。
- **创建工作者线程的开销更大**。工作者线程有自己独立的事件循环、全局对象、事件处理程序和其他 JavaScript 环境必需的特性。创建这些结构的代价不容忽视

无论形式还是功能，工作者线程都不是用于替代线程的。HTML Web 工作者线程规范是这样说的：

> 工作者线程相对比较重，不建议大量使用。例如，对一张 400 万像素的图片，为每个像素都启动一个工作者线程是不合适的。通常，工作者线程应该是长期运行的，启动成本比较高，每个实例占用的内存也比较大。

27.1.2　工作者线程的类型

Web 工作者线程规范中定义了三种主要的工作者线程：**专用工作者线程**、**共享工作者线程**和**服务工作者线程**。现代浏览器都支持这些工作者线程。

> **注意**　Web 工作者线程规范参见 HTML Standard 网站。

1. 专用工作者线程

专用工作者线程，通常简称为工作者线程、Web Worker 或 Worker，是一种实用的工具，可以让脚本单独创建一个 JavaScript 线程，以执行委托的任务。专用工作者线程，顾名思义，只能被创建它的页面使用。

2. 共享工作者线程

共享工作者线程与专用工作者线程非常相似。主要区别是共享工作者线程可以被多个不同的上下文使用，包括不同的页面。任何与创建共享工作者线程的脚本同源的脚本，都可以向共享工作者线程发送消息或从中接收消息。

3. 服务工作者线程

服务工作者线程与专用工作者线程和共享工作者线程截然不同。它的主要用途是拦截、重定向和修改页面发出的请求，充当网络请求的仲裁者的角色。

> **注意**　还有其他一些工作者线程规范，比如 ChromeWorker 或 Web Audio API，但它们并未得到广泛支持，或者定位于小众应用程序，因此本书没有包含与之相关的内容。

27.1.3 WorkerGlobalScope

在网页上，window 对象可以向运行在其中的脚本暴露各种全局变量。在工作者线程内部，没有 window 的概念。这里的全局对象是 WorkerGlobalScope 的实例，通过 self 关键字暴露出来。

1. WorkerGlobalScope 属性和方法

self 上可用的属性是 window 对象上属性的严格子集。其中有些属性会返回特定于工作者线程的版本。

- navigator：返回与工作者线程关联的 WorkerNavigator。
- self：返回 WorkerGlobalScope 对象。
- location：返回与工作者线程关联的 WorkerLocation。
- performance：返回（只包含特定属性和方法的）Performance 对象。
- console：返回与工作者线程关联的 Console 对象；对 API 没有限制。
- caches：返回与工作者线程关联的 CacheStorage 对象；对 API 没有限制。
- indexedDB：返回 IDBFactory 对象。
- isSecureContext：返回布尔值，表示工作者线程上下文是否安全。
- origin：返回 WorkerGlobalScope 的源。

类似地，self 对象上暴露的一些方法也是 window 上方法的子集。这些 self 上的方法也与 window 上对应的方法操作一样。

- atob()
- btoa()
- clearInterval()
- clearTimeout()
- createImageBitmap()
- fetch()
- setInterval()
- setTimeout()

WorkerGlobalScope 还增加了新的全局方法 importScripts()，只在工作者线程内可用。本章稍后会介绍该方法。

2. WorkerGlobalScope 的子类

实际上并不是所有地方都实现了 WorkerGlobalScope。每种类型的工作者线程都使用了自己特定的全局对象，这继承自 WorkerGlobalScope。

- 专用工作者线程使用 DedicatedWorkerGlobalScope。
- 共享工作者线程使用 SharedWorkerGlobalScope。
- 服务工作者线程使用 ServiceWorkerGlobalScope。

本章稍后会在这些全局对象对应的小节中讨论其差异。

27.2 专用工作者线程

专用工作者线程是最简单的 Web 工作者线程，网页中的脚本可以创建专用工作者线程来执行在页面线程之外的其他任务。这样的线程可以与父页面交换信息、发送网络请求、执行文件输入/输出、进行密集计算、处理大量数据，以及实现其他不适合在页面执行线程里做的任务（否则会导致页面响应迟钝）。

> **注意**　在使用工作者线程时，脚本在哪里执行、在哪里加载是非常重要的概念。除非另有说明，否则本章假定 main.js 是从 https://example.com 域的根路径加载并执行的顶级脚本。

27.2.1　专用工作者线程的基本概念

可以把专用工作者线程称为**后台脚本**（background script）。JavaScript 线程的各个方面，包括生命周期管理、代码路径和输入/输出，都由初始化线程时提供的脚本来控制。该脚本也可以再请求其他脚本，但一个线程总是从一个脚本源开始。

1. 创建专用工作者线程

创建专用工作者线程最常见的方式是加载 JavaScript 文件。把文件路径提供给 `Worker` 构造函数，然后构造函数再在后台异步加载脚本并实例化工作者线程。传给构造函数的文件路径可以是多种形式。

下面的代码演示了如何创建空的专用工作者线程：

emptyWorker.js

```
// 空的 JS 工作者线程文件
```

main.js

```
console.log(location.href); // "https://example.com/"
const worker = new Worker(location.href + 'emptyWorker.js');
console.log(worker);        // Worker {}
```

这个例子非常简单，但涉及几个基本概念。

- emptyWorker.js 文件是从绝对路径加载的。根据应用程序的结构，使用绝对 URL 经常是多余的。
- 这个文件是在后台加载的，工作者线程的初始化完全独立于 main.js。
- 工作者线程本身存在于一个独立的 JavaScript 环境中，因此 main.js 必须以 `Worker` 对象为代理实现与工作者线程通信。在上面的例子中，该对象被赋值给了 `worker` 变量。
- 虽然相应的工作者线程可能还不存在，但该 `Worker` 对象已在原始环境中可用了。

前面的例子可修改为使用相对路径。不过，这要求 main.js 必须与 emptyWorker.js 在同一个路径下：

```
const worker = new Worker('./emptyWorker.js');
console.log(worker);   // Worker {}
```

2. 工作者线程安全限制

工作者线程的脚本文件只能从与父页面相同的源加载。从其他源加载工作者线程的脚本文件会导致错误，如下所示：

```
// 尝试基于 https://example.com/worker.js 创建工作者线程
const sameOriginWorker = new Worker('./worker.js');

// 尝试基于 https://untrusted.com/worker.js 创建工作者线程
const remoteOriginWorker = new Worker('https://untrusted.com/worker.js');

// Error: Uncaught DOMException: Failed to construct 'Worker':
// Script at https://untrusted.com/main.js cannot be accessed
// from origin https://example.com
```

> **注意**　不能使用非同源脚本创建工作者线程，并不影响执行其他源的脚本。在工作者线程内部，使用 `importScripts()` 可以加载其他源的脚本。本章稍后会介绍。

基于加载脚本创建的工作者线程不受文档的内容安全策略限制，因为工作者线程在与父文档不同的上下文中运行。不过，如果工作者线程加载的脚本带有全局唯一标识符（与加载自一个二进制大文件一样），就会受父文档内容安全策略的限制。

> **注意** 27.2.5 节会介绍基于二进制大文件创建工作者线程。

3. 使用 Worker 对象

Worker()构造函数返回的 Worker 对象是与刚创建的专用工作者线程通信的连接点。它可用于在工作者线程和父上下文间传输信息，以及捕获专用工作者线程发出的事件。

> **注意** 要管理好使用 Worker()创建的每个 Worker 对象。在终止工作者线程之前，它不会被垃圾回收，也不能通过编程方式恢复对之前 Worker 对象的引用。

Worker 对象支持下列事件处理程序属性。

- onerror：在工作者线程中发生 ErrorEvent 类型的错误事件时会调用指定给该属性的处理程序。
  - 该事件会在工作者线程中抛出错误时发生。
  - 该事件也可以通过 worker.addEventListener('error', handler)的形式处理。
- onmessage：在工作者线程中发生 MessageEvent 类型的消息事件时会调用指定给该属性的处理程序。
  - 该事件会在工作者线程向父上下文发送消息时发生。
  - 该事件也可以通过使用 worker.addEventListener('message', handler)处理。
- onmessageerror：在工作者线程中发生 MessageEvent 类型的错误事件时会调用指定给该属性的处理程序。
  - 该事件会在工作者线程收到无法反序列化的消息时发生。
  - 该事件也可以通过使用 worker.addEventListener('messageerror', handler)处理。

Worker 对象还支持下列方法。

- postMessage()：用于通过异步消息事件向工作者线程发送信息。
- terminate()：用于立即终止工作者线程。没有为工作者线程提供清理的机会，脚本会突然停止。

4. DedicatedWorkerGlobalScope

在专用工作者线程内部，全局作用域是 DedicatedWorkerGlobalScope 的实例。因为这继承自 WorkerGlobalScope，所以包含它的所有属性和方法。工作者线程可以通过 self 关键字访问该全局作用域。

globalScopeWorker.js

```
console.log('inside worker:', self);
```

main.js

```
const worker = new Worker('./globalScopeWorker.js');

console.log('created worker:', worker);

// created worker: Worker {}

// inside worker: DedicatedWorkerGlobalScope {}
```