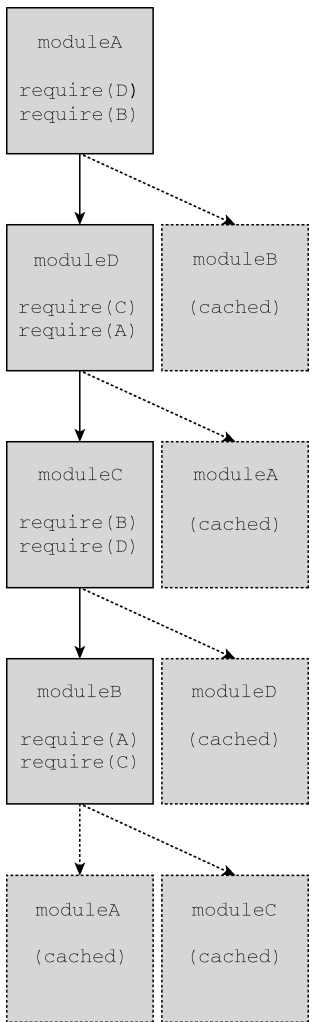

图　26-2

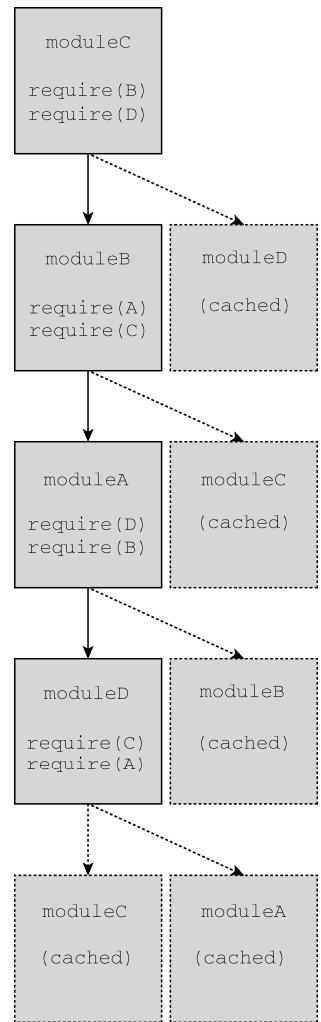

图　26-3

26.2　凑合的模块系统

为按照模块模式提供必要的封装，ES6 之前的模块有时候会使用函数作用域和立即调用函数表达式（IIFE，Immediately Invoked Function Expression）将模块定义封装在匿名闭包中。模块定义是立即执行的，如下：

```
(function() {
  // 私有 Foo 模块的代码
  console.log('bar');
})();

// bar
```

如果把这个模块的返回值赋给一个变量，那么实际上就为模块创建了命名空间：

```
var Foo = (function() {
  console.log('bar');
})();

'bar'
```

为了暴露公共 API，模块 IIFE 会返回一个对象，其属性就是模块命名空间中的公共成员：

```
var Foo = (function() {
  return {
    bar: 'baz',
    baz: function() {
      console.log(this.bar);
    }
  };
})();

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

类似地，还有一种模式叫作“泄露模块模式”（revealing module pattern）。这种模式只返回一个对象，其属性是私有数据和成员的引用：

```
var Foo = (function() {
  var bar = 'baz';
  var baz = function() {
    console.log(bar);
  };

  return {
    bar: bar,
    baz: baz
  };
})();

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

在模块内部也可以定义模块，这样可以实现命名空间嵌套：

```
var Foo = (function() {
  return {
    bar: 'baz'
  };
})();

Foo.baz = (function() {
  return {
    qux: function() {
      console.log('baz');
    }
  };
})();

console.log(Foo.bar); // 'baz'
Foo.baz.qux();        // 'baz'
```

为了让模块正确使用外部的值，可以将它们作为参数传给 IIFE：

```
var globalBar = 'baz';

var Foo = (function(bar) {
  return {
    bar: bar,
    baz: function() {
      console.log(bar);
    }
  };
})(globalBar);

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

因为这里的模块实现其实就是在创建 JavaScript 对象的实例，所以完全可以在定义之后再扩展模块：

```
// 原始的 Foo
var Foo = (function(bar) {
  var bar = 'baz';

  return {
    bar: bar
  };
})();

// 扩展 Foo
var Foo = (function(FooModule) {
  FooModule.baz = function() {
    console.log(FooModule.bar);
  }

  return FooModule;
})(Foo);

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

无论模块是否存在，配置模块扩展以执行扩展也很有用：

```
// 扩展 Foo 以增加新方法
var Foo = (function(FooModule) {
  FooModule.baz = function() {
    console.log(FooModule.bar);
  }

  return FooModule;
})(Foo || {});

// 扩展 Foo 以增加新数据
var Foo = (function(FooModule) {
  FooModule.bar = 'baz';

  return FooModule;
})(Foo || {});

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

当然，自己动手写模块系统确实非常有意思，但实际开发中并不建议这么做，因为不够可靠。前面的例子除了使用恶意的 eval 之外并没有其他更好的动态加载依赖的方法。因此必须手动管理依赖和排序。要添加异步加载和循环依赖非常困难。最后，对这样的系统进行静态分析也是个问题。

26.3 使用 ES6 之前的模块加载器

在 ES6 原生支持模块之前，使用模块的 JavaScript 代码本质上是希望使用默认没有的语言特性。因此，必须按照符合某种规范的模块语法来编写代码，另外还需要单独的模块工具把这些模块语法与 JavaScript 运行时连接起来。这里的模块语法和连接方式有不同的表现形式，通常需要在浏览器中额外加载库或者在构建时完成预处理。

26.3.1 CommonJS

CommonJS 规范概述了同步声明依赖的模块定义。这个规范主要用于在服务器端实现模块化代码组织，但也可用于定义在浏览器中使用的模块依赖。CommonJS 模块语法不能在浏览器中直接运行。

> **注意** 一般认为，Node.js 的模块系统使用了 CommonJS 规范，实际上并不完全正确。Node.js 使用了轻微修改版本的 CommonJS，因为 Node.js 主要在服务器环境下使用，所以不需要考虑网络延迟问题。考虑到一致性，本节使用 Node.js 风格的模块定义语法。

CommonJS 模块定义需要使用 require() 指定依赖，而使用 exports 对象定义自己的公共 API。下面的代码展示了简单的模块定义：

```
var moduleB = require('./moduleB');

module.exports = {
  stuff: moduleB.doStuff();
};
```

moduleA 通过使用模块定义的相对路径来指定自己对 moduleB 的依赖。什么是“模块定义”，以及如何将字符串解析为模块，完全取决于模块系统的实现。比如在 Node.js 中，模块标识符可能指向文件，也可能指向包含 index.js 文件的目录。

请求模块会加载相应模块，而把模块赋值给变量也非常常见，但赋值给变量不是必需的。调用 require() 意味着模块会原封不动地加载进来：

```
console.log('moduleA');
require('./moduleA'); // "moduleA"
```

无论一个模块在 require() 中被引用多少次，模块永远是单例。在下面的例子中，moduleA 只会被打印一次。这是因为无论请求多少次，moduleA 只会被加载一次。

```
console.log('moduleA');
var a1 = require('./moduleA');
var a2 = require('./moduleA');

console.log(a1 === a2); // true
```

模块第一次加载后会被缓存，后续加载会取得缓存的模块（如下代码所示）。模块加载顺序由依赖图决定。

```
console.log('moduleA');
require('./moduleA');
require('./moduleB'); // "moduleA"
require('./moduleA');
```

在 CommonJS 中，模块加载是模块系统执行的同步操作。因此 `require()`可以像下面这样以编程方式嵌入在模块中：

```
console.log('moduleA');
if (loadCondition) {
  require('./moduleA');
}
```

这里，`moduleA` 只会在 `loadCondition` 求值为 `true` 时才会加载。这个加载是同步的，因此 `if()` 块之前的任何代码都会在加载 `moduleA` 之前执行，而 `if()`块之后的任何代码都会在加载 `moduleA` 之后执行。同样，加载顺序规则也会适用。因此，如果 `moduleA` 已经在前面某个地方加载过了，这个条件 `require()`就意味着只暴露 `moduleA` 这个命名空间而已。

在上面的例子中，模块系统是 Node.js 实现的，因此 `./moduleB` 是相对路径，指向与当前模块位于同一目录中的模块目标。Node.js 会使用 `require()`调用中的模块标识符字符串去解析模块引用。在 `Node.js` 中可以使用绝对或相对路径，也可以使用安装在 node_modules 目录中依赖的模块标识符。我们并不关心这些细节，重要的是知道在不同的 CommonJS 实现中模块字符串引用的含义可能不同。不过，所有 CommonJS 风格的实现共同之处是模块不会指定自己的标识符，它们的标识符由其在模块文件层级中的位置决定。

指向模块定义的路径可能引用一个目录，也可能是一个 JavaScript 文件。无论是什么，这与本地模块实现无关，而 `moduleB` 被加载到本地变量中。`moduleA` 在 `module.exports` 对象上定义自己的公共接口，即 `foo` 属性。

如果有模块想使用这个接口，可以像下面这样导入它：

```
var moduleA = require('./moduleA');

console.log(moduleA.stuff);
```

注意，此模块不导出任何内容。即使它没有公共接口，如果应用程序请求了这个模块，那也会在加载时执行这个模块体。

`module.exports` 对象非常灵活，有多种使用方式。如果只想导出一个实体，可以直接给 `module.exports` 赋值：

```
module.exports = 'foo';
```

这样，整个模块就导出一个字符串，可以像下面这样使用：

```
var moduleA = require('./moduleB');

console.log(moduleB); // 'foo'
```

导出多个值也很常见，可以使用对象字面量赋值或每个属性赋一次值来实现：

```
// 等价操作：

module.exports = {
  a: 'A',
  b: 'B'
};
```

```
module.exports.a = 'A';
module.exports.b = 'B';
```

模块的一个主要用途是托管类定义（这里使用 ES6 风格的类定义，不过 ES5 风格也兼容）：

```
class A {}

module.exports = A;
var A = require('./moduleA');

var a = new A();
```

也可以将类实例作为导出值：

```
class A {}

module.exports = new A();
```

此外，CommonJS 也支持动态依赖：

```
if (condition) {
  var A = require('./moduleA');
}
```

CommonJS 依赖几个全局属性如 `require` 和 `module.exports`。如果想在浏览器中使用 CommonJS 模块，就需要与其非原生的模块语法之间构筑“桥梁”。模块级代码与浏览器运行时之间也需要某种“屏障”，因为没有封装的 CommonJS 代码在浏览器中执行会创建全局变量。这显然与模块模式的初衷相悖。

常见的解决方案是提前把模块文件打包好，把全局属性转换为原生 JavaScript 结构，将模块代码封装在函数闭包中，最终只提供一个文件。为了以正确的顺序打包模块，需要事先生成全面的依赖图。

26.3.2 异步模块定义

CommonJS 以服务器端为目标环境，能够一次性把所有模块都加载到内存，而异步模块定义（AMD，Asynchronous Module Definition）的模块定义系统则以浏览器为目标执行环境，这需要考虑网络延迟的问题。AMD 的一般策略是让模块声明自己的依赖，而运行在浏览器中的模块系统会按需获取依赖，并在依赖加载完成后立即执行依赖它们的模块。

AMD 模块实现的核心是用函数包装模块定义。这样可以防止声明全局变量，并允许加载器库控制何时加载模块。包装函数也便于模块代码的移植，因为包装函数内部的所有模块代码使用的都是原生 JavaScript 结构。包装模块的函数是全局 `define` 的参数，它是由 AMD 加载器库的实现定义的。

AMD 模块可以使用字符串标识符指定自己的依赖，而 AMD 加载器会在所有依赖模块加载完毕后立即调用模块工厂函数。与 CommonJS 不同，AMD 支持可选地为模块指定字符串标识符。

```
// ID 为'moduleA'的模块定义。moduleA 依赖 moduleB,
// moduleB 会异步加载
define('moduleA', ['moduleB'], function(moduleB) {
  return {
    stuff: moduleB.doStuff();
  };
});
```

AMD 也支持 `require` 和 `exports` 对象，通过它们可以在 AMD 模块工厂函数内部定义 CommonJS 风格的模块。这样可以像请求模块一样请求它们，但 AMD 加载器会将它们识别为原生 AMD 结构，而

不是模块定义：

```
define('moduleA', ['require', 'exports'], function(require, exports) {
  var moduleB = require('moduleB');

  exports.stuff = moduleB.doStuff();
});
```

动态依赖也是通过这种方式支持的：

```
define('moduleA', ['require'], function(require) {
  if (condition) {
    var moduleB = require('moduleB');
  }
});
```

26.3.3　通用模块定义

为了统一 CommonJS 和 AMD 生态系统，通用模块定义（UMD，Universal Module Definition）规范应运而生。UMD 可用于创建这两个系统都可以使用的模块代码。本质上，UMD 定义的模块会在启动时检测要使用哪个模块系统，然后进行适当配置，并把所有逻辑包装在一个立即调用的函数表达式（IIFE）中。虽然这种组合并不完美，但在很多场景下足以实现两个生态的共存。

下面是只包含一个依赖的 UMD 模块定义的示例（来源为 GitHub 上的 UMD 仓库）：

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    // AMD。注册为匿名模块
    define(['moduleB'], factory);
  } else if (typeof module === 'object' && module.exports) {
  // Node。不支持严格 CommonJS
  // 但可以在 Node 这样支持 module.exports 的
  // 类 CommonJS 环境下使用
  module.exports = factory(require(' moduleB '));
} else {
  // 浏览器全局上下文（root 是 window）
  root.returnExports = factory(root. moduleB);
}
}(this, function (moduleB) {
  // 以某种方式使用 moduleB

  // 将返回值作为模块的导出
  // 这个例子返回了一个对象
  // 但是模块也可以返回函数作为导出值
  return {};
}));
```

此模式有支持严格 CommonJS 和浏览器全局上下文的变体。不应该期望手写这个包装函数，它应该由构建工具自动生成。开发者只需专注于模块的内由容，而不必关心这些样板代码。

26.3.4　模块加载器终将没落

随着 ECMAScript 6 模块规范得到越来越广泛的支持，本节展示的模式最终会走向没落。尽管如此，为了了解为什么选择设计决策，了解 ES6 模块规范的由来仍是非常有用的。CommonJS 与 AMD 之间的冲突正是我们现在享用的 ECMAScript 6 模块规范诞生的温床。

26.4 使用 ES6 模块

ES6 最大的一个改进就是引入了模块规范。这个规范全方位简化了之前出现的模块加载器，原生浏览器支持意味着加载器及其他预处理都不再必要。从很多方面看，ES6 模块系统是集 AMD 和 CommonJS 之大成者。

26.4.1 模块标签及定义

ECMAScript 6 模块是作为一整块 JavaScript 代码而存在的。带有 type="module"属性的<script>标签会告诉浏览器相关代码应该作为模块执行，而不是作为传统的脚本执行。模块可以嵌入在网页中，也可以作为外部文件引入：


```
<script type="module">
  // 模块代码
</script>

<script type="module" src="path/to/myModule.js"></script>
```

即使与常规 JavaScript 文件处理方式不同，JavaScript 模块文件也没有专门的内容类型。

与传统脚本不同，所有模块都会像<script defer>加载的脚本一样按顺序执行。解析到<script type="module">标签后会立即下载模块文件，但执行会延迟到文档解析完成。无论对嵌入的模块代码，还是引入的外部模块文件，都是这样。<script type="module">在页面中出现的顺序就是它们执行的顺序。与<script defer>一样，修改模块标签的位置，无论是在<head>还是在<body>中，只会影响文件什么时候加载，而不会影响模块什么时候加载。

下面演示了嵌入模块代码的执行顺序：

```
<!-- 第二个执行 -->
<script type="module"></script>

<!-- 第三个执行 -->
<script type="module"></script>

<!-- 第一个执行 -->
<script></script>
```

另外，可以改为加载外部 JS 模块定义：

```
<!-- 第二个执行 -->
<script type="module" src="module.js"></script>

<!-- 第三个执行 -->
<script type="module" src="module.js"></script>

<!-- 第一个执行 -->
<script><script>
```

也可以给模块标签添加 async 属性。这样影响就是双重的：不仅模块执行顺序不再与<script>标签在页面中的顺序绑定，模块也不会等待文档完成解析才执行。不过，入口模块仍必须等待其依赖加载完成。

与<script type="module">标签关联的 ES6 模块被认为是模块图中的入口模块。一个页面上有多少个入口模块没有限制，重复加载同一个模块也没有限制。同一个模块无论在一个页面中被加载多少

次，也不管它是如何加载的，实际上都只会加载一次，如下面的代码所示：

```
<!-- moduleA 在这个页面上只会被加载一次 -->

<script type="module">
  import './moduleA.js'
<script>
<script type="module">
  import './moduleA.js'
<script>
<script type="module" src="./moduleA.js"></script>
<script type="module" src="./moduleA.js"></script>
```

嵌入的模块定义代码不能使用 `import` 加载到其他模块。只有通过外部文件加载的模块才可以使用 `import` 加载。因此，嵌入模块只适合作为入口模块。

26.4.2　模块加载

ECMAScript 6 模块的独特之处在于，既可以通过浏览器原生加载，也可以与第三方加载器和构建工具一起加载。有些浏览器还没有原生支持 ES6 模块，因此可能还需要第三方工具。事实上，很多时候使用第三方工具可能会更方便。

完全支持 ECMAScript 6 模块的浏览器可以从顶级模块加载整个依赖图，且是异步完成的。浏览器会解析入口模块，确定依赖，并发送对依赖模块的请求。这些文件通过网络返回后，浏览器就会解析它们的内容，确定它们的依赖，如果这些二级依赖还没有加载，则会发送更多请求。这个异步递归加载过程会持续到整个应用程序的依赖图都解析完成。解析完依赖图，应用程序就可以正式加载模块了。

这个过程与 AMD 风格的模块加载非常相似。模块文件按需加载，且后续模块的请求会因为每个依赖模块的网络延迟而同步延迟。即，如果 `moduleA` 依赖 `moduleB`，`moduleB` 依赖 `moduleC`。浏览器在对 `moduleB` 的请求完成之前并不知道要请求 `moduleC`。这种加载方式效率很高，也不需要外部工具，但加载大型应用程序的深度依赖图可能要花费很长时间。

26.4.3　模块行为

ECMAScript 6 模块借用了 CommonJS 和 AMD 的很多优秀特性。下面简单列举一些。

- 模块代码只在加载后执行。
- 模块只能加载一次。
- 模块是单例。
- 模块可以定义公共接口，其他模块可以基于这个公共接口观察和交互。
- 模块可以请求加载其他模块。
- 支持循环依赖。

ES6 模块系统也增加了一些新行为。

- ES6 模块默认在严格模式下执行。
- ES6 模块不共享全局命名空间。
- 模块顶级 `this` 的值是 `undefined`（常规脚本中是 `window`）。
- 模块中的 `var` 声明不会添加到 `window` 对象。
- ES6 模块是异步加载和执行的。

浏览器运行时在知道应该把某个文件当成模块时，会有条件地按照上述 ECMAScript 6 模块行为来施加限制。与<script type="module">关联或者通过 import 语句加载的 JavaScript 文件会被认定为模块。

26.4.4 模块导出

ES6 模块的公共导出系统与 CommonJS 非常相似。控制模块的哪些部分对外部可见的是 export 关键字。ES6 模块支持两种导出：命名导出和默认导出。不同的导出方式对应不同的导入方式，下一节会介绍导入。

export 关键字用于声明一个值为命名导出。导出语句必须在模块顶级，不能嵌套在某个块中：

```
// 允许
export ...

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

导出值对模块内部 JavaScript 的执行没有直接影响，因此 export 语句与导出值的相对位置或者 export 关键字在模块中出现的顺序没有限制。export 语句甚至可以出现在它要导出的值之前：

```
// 允许
const foo = 'foo';
export { foo };

// 允许
export const foo = 'foo';

// 允许，但应该避免
export { foo };
const foo = 'foo';
```

命名导出（named export）就好像模块是被导出值的容器。行内命名导出，顾名思义，可以在同一行执行变量声明。下面展示了一个声明变量同时又导出变量的例子。外部模块可以导入这个模块，而 foo 将成为这个导入模块的一个属性：

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

变量声明跟导出可以不在一行。可以在 export 子句中执行声明并将标识符导出到模块的其他地方：

```
const foo = 'foo';
export { foo };
```

导出时也可以提供别名，别名必须在 export 子句的大括号语法中指定。因此，声明值、导出值和为导出值提供别名不能在一行完成。在下面的例子中，导入这个模块的外部模块可以使用 myFoo 访问导出的值：

```
const foo = 'foo';
export { foo as myFoo };
```

因为 ES6 命名导出可以将模块作为容器，所以可以在一个模块中声明多个命名导出。导出的值可以在导出语句中声明，也可以在导出之前声明：

```
export const foo = 'foo';
export const bar = 'bar';
export const baz = 'baz';
```