以及

```
function foo(..) {
    // ..
}

export { foo as default };
```

在第一段代码中，导出的是此时到函数表达式值的绑定，而不是标识符 foo。换句话说，`export default ..` 接受的是一个表达式。如果之后在你的模块中给 foo 赋一个不同的值，模块导入得到的仍然是原来导出的函数，而不是新的值。

顺便说一下，第一段代码也可以这么写：

```
export default function foo(..) {
    // ..
}
```

尽管这里的 `function foo..` 部分严格说是一个函数表达式，但由于模块内部作用域的原因，它被当作函数声明对待，因为 foo 名称和模块的最高级作用域绑定（通常称为“hoisting”）。`export default class Foo..` 也是这样。然而，虽然你可以 `export var foo = ..`，但是现在还不能 `export default var foo = ..`（或者 let 或者 const），这种不一致很令人烦恼。在编写本部分时，已经有讨论建议为了一致性尽快在后 ES6 时期增加这个功能。

再次回忆一下第二段代码：

```
function foo(..) {
    // ..
}

export { foo as default };
```

在这个版本的模块导出中，默认导出绑定实际上绑定到 foo 标识符而不是它的值，所以得到了前面描述的绑定行为（也就是说，如果之后修改了 foo 的值，在导入一侧看到的值也会更新）。

要十分小心默认导出语法的这个微妙陷阱，特别是在代码逻辑需要更新导出值的时候。如果你并不打算更新默认导出的值，那么使用 `export default ..` 就好。如果确实需要更新这个值，就需要使用 `export { .. as default }`。不管怎样，记得通过代码注释来解释你的意图！

因为每个模块只能有一个 default，所以你可能会忍不住把模块设计成默认导出一个对象，其中包含所有的 API 方法，比如：

```
export default {
    foo() { .. },
    bar() { .. },
    ..
};
```

这个模式看起来与大量开发者已经构造的前 ES6 模块紧密呼应，所以似乎是一个自然而然的方法。但不幸的是，它有一些缺陷，同时也是官方不建议采用的。

具体来说，JavaScript 引擎无法静态分析平凡对象的内容，这意味着它无法对静态 `import` 进行性能优化。让每个成员独立且显式地导出的优点是引擎可以对其进行静态分析和优化。

如果你的 API 已有多个成员，这些原则——每个模块一个默认导出，所有的 API 成员作为命名导出——看起来似乎相互冲突，不是吗？但你可以有一个单独的默认导出，同时又有其他命名导出；它们并不互斥。

所以，要取代这个（不推荐的）模式：

```
export default function foo() { .. }

foo.bar = function() { .. };
foo.baz = function() { .. };
```

可以用

```
export default function foo() { .. }

export function bar() { .. }
export function baz() { .. }
```

在前面的代码中，我使用名称 `foo` 作为 `default` 标识的函数。然而，这个名称 `foo` 对于导出来说是忽略的——实际上导出的名字是 `default`。下一小节将会介绍，在导入这个默认绑定的时候，可以任意地给它起一个名称。

也有人更喜欢这种形式：

```
function foo() { .. }
function bar() { .. }
function baz() { .. }

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

后面很快会介绍 `import`，到那时混用默认和命名导出的效果将会更加清晰。但从本质上说，这意味着最简单的默认导入形式只会提取函数 `foo()`。如果需要的话，用户可以继续手动列出 `bar` 和 `baz` 作为命名导入。

可以设想一下，如果提供大量命名导出绑定，那么对模块的用户来说将会是多么麻烦。有一个通配符导入可以用于把模块的所有导出导入到单个命名空间对象中，但无法通配符导入到顶层绑定。

再一次重申，ES6 模块机制的设计意图是不鼓励模块大量导出；相对而言，它是有意想让这样的方法麻烦一些，作为某种社会工程来提倡简单模块设计，而不是大型 / 复杂模块设计。

我可能会建议你避免混用默认导出和命名导出，特别是在有大量 API 并且通过重构拆分模块不实际或者不想这么做的时候。这种情况下，就都用命名导出好了，提供文档说明模块用户会用 `import * as ..`（名字空间导入，下一小节将会介绍）方法来一次把所有 API 引入到某个名字空间中。

前面已经介绍过，但这里让我们再详细讨论一下。除了 `export default ...` 形式导出一个表达式值绑定，所有其他的导出形式都是导出局部标识符的绑定。对于这些绑定来说，如果导出之后在模块内部修改某个值，外部导入的绑定会访问到修改后的值：

```
var foo = 42;
export { foo as default };

export var bar = "hello world";

foo = 10;
bar = "cool";
```

当你导入这个模块的时候，`default` 和 `bar` 导出会绑定到局部变量 `foo` 和 `bar`，也就是说它们会暴露更新后的值 `10` 和 `"cool"`。导出时刻的值无关紧要。导入时刻的值也无关紧要。绑定是活连接，所以重要的是访问这个绑定时刻的当前值。

双向绑定是不允许的。如果从一个模块导入了 `foo`，然后修改导入的 `foo` 变量的值，就会抛出错误！下一小节我们会再次介绍这一点。

你也可以再次导出某个模块的导出，就像这样：

```
export { foo, bar } from "baz";
export { foo as FOO, bar as BAR } from "baz";
export * from "baz";
```

这些形式类似于首先从 `"baz"` 模块导入，然后显式列出它的成员再从你的模块导出。但这些形式中，`"baz"` 模块的成员不会导入到你的模块的局部作用域，它们就像是不留痕迹地穿过。

2. 导入 API 成员

不出意料，使用 `import` 语句导入模块。就像 `export` 有几种变体，`import` 也是一样，所以

花点时间思考接下来的问题，并验证一下你的选择。

如果想导入一个模块 API 的某个特定命名成员到你的顶层作用域，可以使用下面语法：

```
import { foo, bar, baz } from "foo";
```

这里的 { .. } 语法可能看起来像是一个对象字面量，或者甚至是一个对象解构语法。但这种形式是专用于模块的，所以注意不要把它和其他 { .. } 模式混淆。

字符串 "foo" 称为**模块指定符**（module specifier）。因为整体目标是可静态分析的语法，模块指定符必须是字符串字面值，而不能是持有字符串值的变量。

从 ES6 代码和 JavaScript 引擎本身的角度来说，这个字符串字面量的内容是完全透明的，也毫无意义。模块加载器会把这个字符串解释为一个决定去哪儿寻找所需模块的指令，或者作为 URL 路径或者是本地文件系统路径。

列出的标识符 foo、bar 和 baz 必须匹配模块 API 的命名导出（会应用静态分析和错误判定）。它们会在当前作用域绑定为顶层标识符：

```
import { foo } from "foo";

foo();
```

你可以对导入绑定标识符重命名，就像这样：

```
import { foo as theFooFunc } from "foo";

theFooFunc();
```

如果这个模块只有一个你想要导入并绑定到一个标识符的默认导出，绑定时可以省略包围的 { .. } 语法。这种情况下的 import 得到了最简洁优美的 import 语法形式：

```
import foo from "foo";

// 或者：
import { default as foo } from "foo";
```

前面一小节介绍过，模块的 export 中的关键字 default 指定了一个命名导出，名称实际上就是 default，就像第二种更详细的语法形式表明的一样。在后一种语法中，从 default 到这个例子中的 foo 的重命名都是显式的，和前一种隐式语法形式是一样的。

你还可以把默认导出与其他命名导出一起导入，如果这个模块有这样的定义的话。回忆前

面这个模块定义：

```
export default function foo() { .. }

export function bar() { .. }
export function baz() { .. }
```

导入这个模块的默认导出和它的两个命名导出：

```
import FOOFN, { bar, baz as BAZ } from "foo";

FOOFN();
bar();
BAZ();
```

ES6 模块哲学强烈建议的方法是，只从模块导入需要的具体绑定。如果一个模块提供了 10 个 API 方法，但是你只需要其中的 2 个，有些人坚信把所有的 API 绑定都导入进来是一种浪费。

除了代码更清晰，窄导入的另一个好处是使得静态分析和错误检测（比如意外使用了错误的绑定名称）更加健壮。

当然，ES6 设计哲学只影响了标准立场，没有任何强制要求必须采用这种方法。

很多开发者很快会发现这种方法可能更繁复，因为每次意识到需要模块中的新东西的时候，都要重新访问和更新 `import` 语句。因此这种方法要权衡考虑的是便捷性。

考虑到这一点，理想的选择是从模块把所有一切导入到一个单独命名空间，而不是向作用域直接导入独立的成员。幸运的是，`import` 语句有一种语法变体可以支持这种模块导入，称为**命名空间导入**（namespace import）。

考虑一个模块 `"foo"` 导出，如下：

```
export function bar() { .. }
export var x = 42;
export function baz() { .. }
```

你可以把整个 API 导入到单个模块命名空间绑定：

```
import * as foo from "foo";

foo.bar();
foo.x;          // 42
foo.baz();
```

这个 `* as ..` 语句需要一个 `*` 通配符。换句话说，不能用 `import { bar, x } as foo from "foo"` 这样的语句只导入 API 的一部分但仍然绑定到 `foo` 命名空间。我希望有这样的支持，但是 ES6 命名空间导入是要么全有要么全无的。

如果通过 * as .. 导入的模块有默认导出，它在指定的命名空间中的名字就是 default。你还可以在这个命名空间绑定之外把默认导入作为顶层标识符命名。考虑一个模块 "world" 导出，如下：

```
export default function foo() { .. }
export function bar() { .. }
export function baz() { .. }
```

以及下面的导入：

```
import foofn, * as hello from "world";

foofn();
hello.default();
hello.bar();
hello.baz();
```

虽然这个语法是合法的，但可能会令人迷惑，因为这个模块的一个方法（默认导出）绑定到了作用域的顶层，而其余的命名导出（其中一个名为 default）绑定到了另一个（hello）标识符命名空间。

前面已经提到过，我建议避免这样设计模块导出，为的是尽量避免模块用户被这种奇怪的设计所迷惑。

所有导入的绑定都是不可变和 / 或只读的。考虑一下前面的导入，导入之后所有这些试图赋值的动作都会抛出 TypeErrors：

```
import foofn, * as hello from "world";

foofn = 42;          // （运行时）TypeError!
hello.default = 42;  // （运行时）TypeError!
hello.bar = 42;      // （运行时）TypeError!
hello.baz = 42;      // （运行时）TypeError!
```

回忆一下 3.3.3 节，其中讨论了 bar 和 baz 绑定是如何绑定到模块 "world" 内部的实际标识符上的。这意味着如果模块修改了这些值，hello.bar 和 hello.baz 现在指向修改后的新值。

但是你的局部导入绑定的不变性 / 只读性限制了无法从导入的绑定修改它们，否则就会 TypeErrors。这是非常重要的，因为如果没有这样的保护，你的修改就会最终影响这个模块的所有其他用户（别忘了：单例），这会导致出乎意料的副作用！

另外，尽管模块可以从内部修改 API 成员，但如果需要故意这样设计还是要格外小心。ES6 模块应该是静态的，所以要尽可能少地偏离这个原则，并且应该用文档加以认真详尽地说明。

有一些设计哲学实际上想让用户修改 API 的某个属性值，或者模块 API 被设计成是通过增加到 API 命名空间的“插件”来扩展的。前面我们断言，ES6 模块 API 应该被认为或者被设计成静态不可变的，这严格限制和抑制了这些另类的模块设计模式。你可以通过导出平凡对象来绕过这些限制，这个对象当然可以修改。但是要这么做之前一定要三思而后行。

作为 `import` 结果的声明是“提升的”（参见本系列《你不知道的 JavaScript（上卷）》第一部分）。考虑：

```
foo();

import { foo } from "foo";
```

`foo()` 可以运行，不只是因为 `import ..` 语句的静态决议在编译过程中确定了 `foo` 值是什么，也因为它“提升”了在模块作用域顶层的声明，使它在模块所有位置可用。

最后，`import` 最基本的形式是这样的：

```
import "foo";
```

这种形式并没有实际导入任何一个这个模块的绑定到你的作用域。它加载（如果还没有加载的话）、编译（如果还没有编译的话），并求值（如果还没有运行的话）`"foo"` 模块。

一般来说，这种导入没什么太大用处。可能有一些模块定义有副作用（比如把东西赋给 `window` / 全局对象）的情况。你也可以把 `import "foo"` 想象成是对以后可能需要的模块的预加载。

3.3.4 模块依赖环

A 导入 B，B 导入 A。这种情况到底是怎么工作的？

首先必需声明，我尽量避免故意设计带有环形依赖的系统。前面已经说过，我意识到有一些原因导致人们需要这么做，因为它可以解决某些棘手的设计情况。

让我们看一下 ES6 是怎么处理这个问题的。首先，模块 `"A"`：

```
import bar from "B";

export default function foo(x) {
    if (x > 10) return bar( x - 1 );
    return x * 2;
}
```

然后，模块 `"B"`：

```
import foo from "A";
```

```
export default function bar(y) {
    if (y > 5) return foo( y / 2 );
    return y * 3;
}
```

foo(..) 和 bar(..) 这两个函数如果在同一个作用域的话，将会作为标准函数声明，因为这两个声明是“提升”到整个作用域的，所以不管代码顺序如何都可以访问彼此。

有了模块，那么声明就是在完全不同的作用域，所以 ES6 需要额外的工作来支持这样的循环引用。

下面是从粗略概念的意义上循环的 import 依赖如何生效和解析的过程。

- 如果先加载模块 "A"，第一步是扫描这个文件分析所有的导出，这样就可以注册所有可以导入的绑定。然后处理 import .. from "B"，这表示它需要取得 "B"。
- 引擎加载 "B" 之后，会对它的导出绑定进行同样的分析。当看到 import .. from "A"，它已经了解 "A" 的 API，所以可以验证 import 是否有效。现在它了解 "B" 的 API，就可以验证等待的 "A" 模块中 import .. from "B" 的有效性。

本质上说，相互导入，加上检验两个 import 语句的有效性的静态验证，虚拟组合了两个独立的模块空间（通过绑定），这样 foo(..) 可以调用 bar(..)，反过来也是一样。这和如果它们本来是声明在同一个作用域中是对称的。

现在让我们试着来使用这两个模块。首先，试一下 foo(..)：

```
import foo from "foo";
foo( 25 );              // 11
```

也可以使用 bar(..)：

```
import bar from "bar";
bar( 25 );              // 11.5
```

在 foo(25) 或 bar(25) 调用执行的时候，所有模块的所有分析 / 编译都已经完成。这意味着 foo(..) 内部已经直接了解 bar(..)，而 bar(..) 内部也已经直接了解 foo(..)。

如果只是需要与 foo(..) 交互，那么只需要导入 "foo" 模块。对于 bar(..) 和 "bar" 模块也是一样。

当然，需要的话可以导入并使用二者：

```
import foo from "foo";
import bar from "bar";

foo( 25 );              // 11
bar( 25 );              // 11.5
```

import 语句的静态加载语义意味着可以确保通过 import 相互依赖的 "foo" 和 "bar" 在其中任何一个运行之前，二者都会被加载、解析和编译。所以它们的环依赖是静态决议的，就像期望的一样。

3.3.5 模块加载

我们在 3.3 节开始部分介绍过，import 语句使用外部环境（浏览器、Node.js 等）提供的独立机制，来实际把模块标识符字符串解析成可用的指令，用于寻找和加载所需的模块。这个机制就是系统**模块加载器**。

如果在浏览器中，环境提供的默认模块加载器会把模块标识符解析为 URL，（一般来说）如果在像 Node.js 这样的服务器上就解析为本地文件系统路径。默认行为方式假定加载的文件是以 ES6 标准模块格式编写的。

另外，还可以通过 HTML 标签加载模块到浏览器中，这与目前脚本程序加载的方式类似。编写本部分的时候，还不清楚这个标签是 <script type="module"> 还是 <module>。这并非由 ES6 决定，ES6 讨论的同时在合适的规范中对此已经有了很多讨论。

不管这个标签看起来是什么样，可以确定的是在底层将会使用默认加载器（或者是我们将在下一小节介绍的预先指定的自定义加载器）。

就像在 markup 中使用的标签一样，模块加载器本身不是由 ES6 指定的。它是独立的、平行的标准（http:// whatwg.github.io/loader/），目前由 WHATWG 浏览器标准工作组管理。

接下来的讨论反映了编写本部分时 API 设计上的一个初期版本，未来很可能改变。

1. 在模块之外加载模块

直接与模块加载器交互的一个用法是非模块需要加载一个模块的情况。考虑：

```
// 一般脚本在浏览器中通过<script>加载，这里import不合法

Reflect.Loader.import( "foo" ) // 为"foo"返回一个promise
.then( function(foo){
    foo.bar();
} );
```

工具 Reflect.Loader.import(..) 把整个模块导入到命名参数（作为一个命名空间），就像前面讨论的命名空间导入 import * as foo .. 一样。

Reflect.Loader.import(..) 工具返回一个 promise，这个 promise 模块就绪就会完成。要导入多个模块，可以使用 Promise.all([..]) 组合多个 Reflect.Loader.import(..) 调用返回的 promise。关于 Promise 的更多信息，参见 4.1 节。

你也可以在真正的模块中使用 `Reflect.Loader.import(..)` 来动态 / 有条件地加载一个模块，而 `import` 本身无法实现。比如，你可能想要在测试表明当前引擎没有定义某个 ES7+ 特性的情况下才加载一个包含这个特性 polyfill 的模块。

出于性能的考虑，需要尽可能避免动态加载，因为这会妨碍 JavaScript 引擎根据静态分析提前获取代码的能力。

2. 自定义加载

另外一种与模块加载器直接交互的用法，就是需要通过配置甚至重定义来自定义其行为的情况。

在编写本部分的时候，已经有一个模块加载器 API 的 polyfill 在开发之中了（https://github.com/ModuleLoader/es6-module-loader）。因为详细信息还比较少，也很可能会变化，所以我们来探索一下几种最终可能出现的特性。

`Reflect.Loader.import(..)` 调用可能会支持第二个参数，用来指定自定义导入 / 加载任务的各种选项。比如：

```
Reflect.Loader.import( "foo", { address: "/path/to/foo.js" } )
.then( function(foo){
    // ..
})
```

还可能会（通过某些手段）提供自定义支持来嵌入加载模块的过程，在加载完成引擎编译模块之前可以执行一个转换 /transpilation。

举例来说，可以加载某些不符合 ES6 规范的模块格式（比如 CoffeeScript、TypeScript、CommonJS 和 AMD）。转换步骤把它转换为遵循 ES6 规范的模块，然后引擎处理。

3.4 类

几乎从 JavaScript 发展初期开始，语法和开发模式都极力营造支持面向对象开发的假象。通过诸如 `new` 和 `instanceof` 以及 `.constructor` 属性，让人们忍不住以为 JavaScript 在其原型系统内部某处隐藏着类。

当然，JavaScript“类”和传统的类并不相同。二者的区别已经有文档详尽说明，关于这点这里我不再赘述。

要进一步学习 JavaScript 中用来模拟“类”的模式，以及名为“委托”的这种对原型的另外一种看法，参见本系列《你不知道的 JavaScript（上卷）》第二部分的后半部分。