关于术语的一点说明：
请务必注意一点，TypeScript 1.5里术语名已经发生了变化。
“内部模块”现在称做“命名空间”。
“外部模块”现在则简称为“模块”，这是为了与ECMAScript 2015里的术语保持一致，(也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

介绍

这篇文章将概括介绍在TypeScript里使用模块与命名空间来组织代码的方法。
我们也会谈及命名空间和模块的高级使用场景，和在使用它们的过程中常见的陷阱。

查看模块章节了解关于模块的更多信息。
查看命名空间章节了解关于命名空间的更多信息。

使用命名空间

命名空间是位于全局命名空间下的一个普通的带有名字的JavaScript对象。
这令命名空间十分容易使用。
它们可以在多文件中同时使用，并通过`--outFile`结合在一起。
命名空间是帮你组织Web应用不错的方式，你可以把所有依赖都放在HTML页面的`<script>`标签里。

但就像其它的全局命名空间污染一样，它很难去识别组件之间的依赖关系，尤其是在大型的应用中。

使用模块

像命名空间一样，模块可以包含代码和声明。
不同的是模块可以*声明*它的依赖。

模块会把依赖添加到模块加载器上（例如CommonJs / Require.js）。
对于小型的JS应用来说可能没必要，但是对于大型应用，这一点点的花费会带来长久的模块化和可维护性上的便利。
模块也提供了更好的代码重用，更强的封闭性以及更好的使用工具进行优化。

对于Node.js应用来说，模块是默认并推荐的组织代码的方式。

从ECMAScript 2015开始，模块成为了语言内置的部分，应该会被所有正常的解释引擎所支持。
因此，对于新项目来说推荐使用模块做为组织代码的方式。

命名空间和模块的陷阱

这部分我们会描述常见的命名空间和模块的使用陷阱和如何去避免它们。

对模块使用/// <reference>

一个常见的错误是使用/// <reference>引用模块文件，应该使用import。
要理解这之间的区别，我们首先应该弄清编译器是如何根据import路径（例如，import x from "...";或import x = require("...")里面的...，等等）来定位模块的类型信息的。

编译器首先尝试去查找相应路径下的.ts，.tsx再或者.d.ts。
如果这些文件都找不到，编译器会查找*外部模块声明*。
回想一下，它们是在.d.ts文件里声明的。

- myModules.d.ts

```
// In a .d.ts file or .ts file that is not a module:
declare module "SomeModule" {
    export function fn(): string;
}
```

- myOtherModule.ts

```
/// <reference path="myModules.d.ts" />
import * as m from "SomeModule";
```

这里的引用标签指定了外来模块的位置。
这就是一些TypeScript例子中引用node.d.ts的方法。

不必要的命名空间

如果你想把命名空间转换为模块，它可能会像下面这个文件一件：

- shapes.ts

```
export namespace Shapes {
    export class Triangle { /* ... */ }
    export class Square { /* ... */ }
}
```

顶层的模块Shapes包裹了Triangle和Square。
对于使用它的人来说这是令人迷惑和讨厌的：

- shapeConsumer.ts

```
import * as shapes from "./shapes";
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

TypeScript里模块的一个特点是不同的模块永远也不会在相同的作用域内使用相同的名字。
因为使用模块的人会为它们命名，所以完全没有必要把导出的符号包裹在一个命名空间里。

再次重申，不应该对模块使用命名空间，使用命名空间是为了提供逻辑分组和避免命名冲突。

模块文件本身已经是一个逻辑分组，并且它的名字是由导入这个模块的代码指定，所以没有必要为导出的对象增加额外的模块层。

下面是改进的例子：

- shapes.ts

```
export class Triangle { /* ... */ }
export class Square { /* ... */ }
```

- shapeConsumer.ts

```
import * as shapes from "./shapes";
let t = new shapes.Triangle();
```

模块的取舍

就像每个JS文件对应一个模块一样，TypeScript里模块文件与生成的JS文件也是一一对应的。
这会产生一种影响，根据你指定的目标模块系统的不同，你可能无法连接多个模块源文件。
例如当目标模块系统为`commonjs`或`umd`时，无法使用`outFile`选项，但是在TypeScript 1.8以上的版本[能够](./release notes/TypeScript 1.8.md#concatenate-amd-and-system-modules-with---outfile)使用`outFile`当目标为`amd`或`system`。