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

这篇文章描述了如何在TypeScript里使用命名空间（之前叫做“内部模块”）来组织你的代码。
就像我们在术语说明里提到的那样，“内部模块”现在叫做“命名空间”。
另外，任何使用`module`关键字来声明一个内部模块的地方都应该使用`namespace`关键字来替换。
这就避免了让新的使用者被相似的名称所迷惑。

第一步

我们先来写一段程序并将在整篇文章中都使用这个例子。
我们定义几个简单的字符串验证器，假设你会使用它们来验证表单里的用户输入或验证外部数据。

所有的验证器都放在一个文件里

```
interface StringValidator {
    isAcceptable(s: string): boolean;
}

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();
```

```
// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        let isMatch = validators[name].isAcceptable(s);
        console.log(`'${ s }' ${ isMatch ? "matches" : "does not match" } '${ n
    }
}
```

命名空间

随着更多验证器的加入，我们需要一种手段来组织代码，以便于在记录它们类型的同时还不用担心与其它对象产生命名冲突。
因此，我们把验证器包裹到一个命名空间内，而不是把它们放在全局命名空间下。

下面的例子里，把所有与验证器相关的类型都放到一个叫做`Validation`的命名空间里。
因为我们想让这些接口和类在命名空间之外也是可访问的，所以需要使用`export`。
相反的，变量`lettersRegexp`和`numberRegexp`是实现的细节，不需要导出，因此它们在命名空间外是不能访问的。
在文件末尾的测试代码里，由于是在命名空间之外访问，因此需要限定类型的名称，比如`Validation.LettersOnlyValidator`。

使用命名空间的验证器

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }

    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
```

```
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches"
    }
}
```

分离到多文件

当应用变得越来越大时，我们需要将代码分离到不同的文件中以便于维护。

多文件中的命名空间

现在，我们把`Validation`命名空间分割成多个文件。
尽管是不同的文件，它们仍是同一个命名空间，并且在使用的时候就如同它们在一个文件中定义的一样。
因为不同文件之间存在依赖关系，所以我们加入了引用标签来告诉编译器文件之间的关联。
我们的测试代码保持不变。

Validation.ts

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}
```

LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }
}
```

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
```

```
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log("""" + s + "" "" + (validators[name].isAcceptable(s) ? " matc
    }
}
```

当涉及到多文件时，我们必须确保所有编译后的代码都被加载了。
我们有两种方式。

第一种方式，把所有的输入文件编译为一个输出文件，需要使用`--outFile`标记：

```
tsc --outFile sample.js Test.ts
```

编译器会根据源码里的引用标签自动地对输出进行排序。你也可以单独地指定每个文件。

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.
```

第二种方式，我们可以编译每一个文件（默认方式），那么每个源文件都会对应生成一个JavaScript文件。
然后，在页面上通过`<script>`标签把所有生成的JavaScript文件按正确的顺序引进来，比如：

MyTestPage.html (excerpt)

```
    <script src="Validation.js" type="text/javascript" />
    <script src="LettersOnlyValidator.js" type="text/javascript" />
    <script src="ZipCodeValidator.js" type="text/javascript" />
    <script src="Test.js" type="text/javascript" />
```

别名

另一种简化命名空间操作的方法是使用`import q = x.y.z`给常用的对象起一个短的名字。不要与用来加载模块的`import x = require('name')`语法弄混了，这里的语法是为指定的符号创建一个别名。
你可以用这种方法为任意标识符创建别名，也包括导入的模块中的对象。

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle { }
        export class Square { }
    }
}
```

```
import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as "new Shapes.Polygons.Square()"
```

注意，我们并没有使用`require`关键字，而是直接使用导入符号的限定名赋值。
这与使用`var`相似，但它还适用于类型和导入的具有命名空间含义的符号。
重要的是，对于值来讲，`import`会生成与原始符号不同的引用，所以改变别名的`var`值并不会影响原始变量的值。

使用其它的JavaScript库

为了描述不是用TypeScript编写的类库的类型，我们需要声明类库导出的API。
由于大部分程序库只提供少数的顶级对象，命名空间是用来表示它们的一个好办法。

我们称其为声明是因为它不是外部程序的具体实现。
我们通常在`.d.ts`里写这些声明。
如果你熟悉C/C++，你可以把它们当做`.h`文件。
让我们看一些例子。

外部命名空间

流行的程序库D3在全局对象`d3`里定义它的功能。
因为这个库通过一个`<script>`标签加载（不是通过模块加载器），它的声明文件使用内部模块来定义它的类型。
为了让TypeScript编译器识别它的类型，我们使用外部命名空间声明。
比如，我们可以像下面这样写：

D3.d.ts (部分摘录)

```
declare namespace D3 {
    export interface Selectors {
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        };
    }

    export interface Event {
        x: number;
        y: number;
    }

    export interface Base extends Selectors {
        event: Event;
    }
}

declare var d3: D3.Base;
```