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

从ECMAScript 2015开始，JavaScript引入了模块的概念。TypeScript也沿用这个概念。

模块在其自身的作用域里执行，而不是在全局作用域里；这意味着定义在一个模块里的变量，函数，类等等在模块外部是不可见的，除非你明确地使用`export`形式之一导出它们。相反，如果想使用其它模块导出的变量，函数，类，接口等的时候，你必须要导入它们，可以使用`import`形式之一。

模块是自声明的；两个模块之间的关系是通过在文件级别上使用imports和exports建立的。

模块使用模块加载器去导入其它的模块。
在运行时，模块加载器的作用是在执行此模块代码前去查找并执行这个模块的所有依赖。
大家最熟知的JavaScript模块加载器是服务于Node.js的CommonJS和服务于Web应用的Require.js。

TypeScript与ECMAScript 2015一样，任何包含顶级`import`或者`export`的文件都被当成一个模块。

导出

导出声明

任何声明（比如变量，函数，类，类型别名或接口）都能够通过添加`export`关键字来导出。

Validation.ts

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

ZipCodeValidator.ts

```
export const numberRegexp = /^[0-9]+$/;

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

导出语句

导出语句很便利，因为我们可能需要对导出的部分重命名，所以上面的例子可以这样改写：

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

重新导出

我们经常会去扩展其它模块，并且只导出那个模块的部分内容。
重新导出功能并不会在当前模块导入那个模块或定义一个新的局部变量。

ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && parseInt(s).toString() === s;
    }
}

// 导出原先的验证器但做了重命名
export {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidat
```

或者一个模块可以包裹多个模块，并把他们导出的内容联合在一起通过语法：`export * from "module"`。

AllValidators.ts

```
export * from "./StringValidator"; // exports interface StringValidator
export * from "./LettersOnlyValidator"; // exports class LettersOnlyValidator
export * from "./ZipCodeValidator";  // exports class ZipCodeValidator
```

导入

模块的导入操作与导出一样简单。
可以使用以下`import`形式之一来导入其它模块中的导出内容。

导入一个模块中的某个导出内容

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

可以对导入内容重命名

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

将整个模块导入到一个变量，并通过它来访问模块的导出部分

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

具有副作用的导入模块

尽管不推荐这么做，一些模块会设置一些全局状态供其它模块使用。
这些模块可能没有任何的导出或用户根本就不关注它的导出。
使用下面的方法来导入这类模块：

```
import "./my-module.js";
```

默认导出

每个模块都可以有一个`default`导出。
默认导出使用`default`关键字标记；并且一个模块只能够有一个`default`导出。
需要使用一种特殊的导入形式来导入`default`导出。

`default`导出十分便利。
比如，像JQuery这样的类库可能有一个默认导出`jQuery`或`$`，并且我们基本上也会使用同样的名字`jQuery`或`$`导出JQuery。

JQuery.d.ts

```
declare let $: JQuery;
export default $;
```

App.ts

```
import $ from "JQuery";

$("button.continue").html( "Next Step..." );
```

类和函数声明可以直接被标记为默认导出。
标记为默认导出的类和函数的名字是可以省略的。

ZipCodeValidator.ts

```
export default class ZipCodeValidator {
    static numberRegexp = /^[0-9]+$/;
    isAcceptable(s: string) {
        return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
    }
}
```

Test.ts

```
import validator from "./ZipCodeValidator";

let myValidator = new validator();
```

或者

StaticZipCodeValidator.ts

```
const numberRegexp = /^[0-9]+$/;

export default function (s: string) {
    return s.length === 5 && numberRegexp.test(s);
}
```

Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach(s => {
  console.log(`"${s}" ${validate(s) ? " matches" : " does not match"}`);
});
```

`default`导出也可以是一个值

OneTwoThree.ts

```
export default "123";
```

Log.ts

```
import num from "./OneTwoThree";

console.log(num); // "123"
```

export = 和 import = require()

CommonJS和AMD都有一个`exports`对象的概念，它包含了一个模块的所有导出内容。

它们也支持把`exports`替换为一个自定义对象。
默认导出就好比这样一个功能；然而，它们却并不相互兼容。
TypeScript模块支持`export =`语法以支持传统的CommonJS和AMD的工作流模型。

`export =`语法定义一个模块的导出对象。
它可以是类，接口，命名空间，函数或枚举。

若要导入一个使用了`export =`的模块时，必须使用TypeScript提供的特定语法`import let = require("module")`。

ZipCodeValidator.ts

```
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export = ZipCodeValidator;
```

Test.ts

```
import zip = require("./ZipCodeValidator");

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

// Validators to use
let validator = new zip();

// Show whether each string passed each validator
strings.forEach(s => {
  console.log(`"${ s }" - ${ validator.isAcceptable(s) ? "matches" : "does not
});
```

生成模块代码

根据编译时指定的模块目标参数，编译器会生成相应的供Node.js (CommonJS)，Require.js (AMD)，isomorphic (UMD), SystemJS或ECMAScript 2015 native modules (ES6)模块加载系统使用的代码。
想要了解生成代码中`define`，`require` 和 `register`的意义，请参考相应模块加载器的文档。

下面的例子说明了导入导出语句里使用的名字是怎么转换为相应的模块加载器代码的。

SimpleModule.ts

```
import m = require("mod");
export let t = m.something + 1;
```

AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, exports, mod_1) {
    exports.t = mod_1.something + 1;
});
```

CommonJS / Node SimpleModule.js

```
let mod_1 = require("./mod");
exports.t = mod_1.something + 1;
```

UMD SimpleModule.js

```
(function (factory) {
    if (typeof module === "object" && typeof module.exports === "object") {
        let v = factory(require, exports); if (v !== undefined) module.exports
    }
    else if (typeof define === "function" && define.amd) {
        define(["require", "exports", "./mod"], factory);
    }
})(function (require, exports) {
    let mod_1 = require("./mod");
    exports.t = mod_1.something + 1;
});
```

System SimpleModule.js

```
System.register(["./mod"], function(exports_1) {
    let mod_1;
    let t;
    return {
        setters:[
            function (mod_1_1) {
                mod_1 = mod_1_1;
            }],
        execute: function() {
            exports_1("t", t = mod_1.something + 1);
        }
    }
});
```

Native ECMAScript 2015 modules SimpleModule.js

```
import { something } from "./mod";
export let t = something + 1;
```

简单示例

下面我们来整理一下前面的验证器实现，每个模块只有一个命名的导出。

为了编译，我们必需要在命令行上指定一个模块目标。对于Node.js来说，使用`--module commonjs`;
对于Require.js来说，使用``--module amd`。比如：

```
tsc --module commonjs Test.ts
```

编译完成后，每个模块会生成一个单独的`.js`文件。
好比使用了reference标签，编译器会根据`import`语句编译相应的文件。

Validation.ts

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

LettersOnlyValidator.ts

```
import { StringValidator } from "./Validation";

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

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

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";

const numberRegexp = /^[0-9]+$/;

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

Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// 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
strings.forEach(s => {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches"
    }
});
```

可选的模块加载和其它高级加载场景

有时候，你只想在某种条件下才加载某个模块。
在TypeScript里，使用下面的方式来实现它和其它的高级加载场景，我们可以直接调用模块加载器并且可以保证类型完全。

编译器会检测是否每个模块都会在生成的JavaScript中用到。
如果一个模块标识符只在类型注解部分使用，并且完全没有在表达式中使用时，就不会生成`require`这个模块的代码。
省略掉没有用到的引用对性能提升是很有益的，并同时提供了选择性加载模块的能力。

这种模式的核心是`import id = require("...")`语句可以让我们访问模块导出的类型。

模块加载器会被动态调用（通过`require`），就像下面`if`代码块里那样。
它利用了省略引用的优化，所以模块只在被需要时加载。
为了让这个模块工作，一定要注意`import`定义的标识符只能在表示类型处使用（不能在会转换成JavaScript的地方）。

为了确保类型安全性，我们可以使用`typeof`关键字。
`typeof`关键字，当在表示类型的地方使用时，会得出一个类型值，这里就表示模块的类型。

示例：Node.js里的动态模块加载

```
declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    let ZipCodeValidator: typeof Zip = require("./ZipCodeValidator");
    let validator = new ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
}
```

示例：require.js里的动态模块加载

```
declare function require(moduleNames: string[], onLoad: (...args: any[]) => voi

import  * as Zip from "./ZipCodeValidator";

if (needZipValidation) {
    require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zip) => {
        let validator = new ZipCodeValidator.ZipCodeValidator();
        if (validator.isAcceptable("...")) { /* ... */ }
    });
}
```

示例：System.js里的动态模块加载

```
declare const System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    System.import("./ZipCodeValidator").then((ZipCodeValidator: typeof Zip) =>
        var x = new ZipCodeValidator();
        if (x.isAcceptable("...")) { /* ... */ }
    });
}
```

使用其它的JavaScript库

要想描述非TypeScript编写的类库的类型，我们需要声明类库所暴露出的API。

我们叫它声明因为它不是“外部程序”的具体实现。
它们通常是在`.d.ts`文件里定义的。

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。

外部模块

在Node.js里大部分工作是通过加载一个或多个模块实现的。
我们可以使用顶级的export声明来为每个模块都定义一个.d.ts文件，但最好还是写在一个大的.d.ts文件里。
我们使用与构造一个外部命名空间相似的方法，但是这里使用module关键字并且把名字用引号括起来，方便之后import。
例如：

node.d.ts (simplified excerpt)

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?, slashesDenoteHost?
}

declare module "path" {
    export function normalize(p: string): string;
    export function join(...paths: any[]): string;
    export let sep: string;
}
```

现在我们可以/// <reference> node.d.ts并且使用import url = require("url");
或import * as URL from "url"加载模块。

```
/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("http://www.typescriptlang.org");
```

外部模块简写

假如你不想在使用一个新模块之前花时间去编写声明，你可以采用声明的简写形式以便能够快速使用它。

declarations.d.ts

```
declare module "hot-new-module";
```

简写模块里所有导出的类型将是any。

```
import x, {y} from "hot-new-module";
x(y);
```

模块声明通配符

某些模块加载器如SystemJS
和AMD支持导入非JavaScript内容。
它们通常会使用一个前缀或后缀来表示特殊的加载语法。
模块声明通配符可以用来表示这些情况。

```
declare module "*!text" {
    const content: string;
    export default content;
}
// Some do it the other way around.
declare module "json!*" {
    const value: any;
    export default value;
}
```

现在你可以就导入匹配"*!text"或"json!*"的内容了。

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

UMD模块

有些模块被设计成兼容多个模块加载器，或者不使用模块加载器（全局变量）。
它们以UMD或Isomorphic模块为代表。
这些库可以通过导入的形式或全局变量的形式访问。
例如：

math-lib.d.ts

```
export const isPrime(x: number): boolean;
export as namespace mathLib;
```

之后，这个库可以在某个模块里通过导入来使用：

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a mod
```

它同样可以通过全局变量的形式使用，但只能在某个脚本里。
（脚本是指一个不带有导入或导出的文件。）

```
mathLib.isPrime(2);
```

创建模块结构指导

尽可能地在顶层导出

用户应该更容易地使用你模块导出的内容。
嵌套层次过多会变得难以处理，因此仔细考虑一下如何组织你的代码。

从你的模块中导出一个命名空间就是一个增加嵌套的例子。

虽然命名空间有时候有它们的用处，在使用模块的时候它们额外地增加了一层。
这对用户来说是很不便的并且通常是多余的。

导出类的静态方法也有同样的问题 - 这个类本身就增加了一层嵌套。
除非它能方便表述或便于清晰使用，否则请考虑直接导出一个辅助方法。

如果仅导出单个 `class` 或 `function`，使用 `export default`

就像“在顶层上导出”帮助减少用户使用的难度，一个默认的导出也能起到这个效果。
如果一个模块就是为了导出特定的内容，那么你应该考虑使用一个默认导出。
这会令模块的导入和使用变得些许简单。
比如：

MyClass.ts

```
export default class SomeType {
  constructor() { ... }
}
```

MyFunc.ts

```
export default function getThing() { return 'thing'; }
```

Consumer.ts

```
import t from "./MyClass";
import f from "./MyFunc";
let x = new t();
console.log(f());
```

对用户来说这是最理想的。他们可以随意命名导入模块的类型（本例为`t`）并且不需要多余的（.）来找到相关对象。

如果要导出多个对象，把它们放在顶层里导出

MyThings.ts

```
export class SomeType { /* ... */ }
export function someFunc() { /* ... */ }
```

相反地，当导入的时候：

明确地列出导入的名字

Consumer.ts

```
import { SomeType, SomeFunc } from "./MyThings";
let x = new SomeType();
let y = someFunc();
```

使用命名空间导入模式当你要导出大量内容的时候

MyLargeModule.ts

```
export class Dog { ... }
export class Cat { ... }
export class Tree { ... }
export class Flower { ... }
```

Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";
let x = new myLargeModule.Dog();
```

使用重新导出进行扩展

你可能经常需要去扩展一个模块的功能。
JS里常用的一个模式是JQuery那样去扩展原对象。
如我们之前提到的，模块不会像全局命名空间对象那样去*合并*。
推荐的方案是*不要*去改变原来的对象，而是导出一个新的实体来提供新的功能。

假设`Calculator.ts`模块里定义了一个简单的计算器实现。
这个模块同样提供了一个辅助函数来测试计算器的功能，通过传入一系列输入的字符串并在最后给出结果。

Calculator.ts

```
export class Calculator {
    private current = 0;
    private memory = 0;
    private operator: string;

    protected processDigit(digit: string, currentValue: number) {
        if (digit >= "0" && digit <= "9") {
            return currentValue * 10 + (digit.charCodeAt(0) - "0".charCodeAt(0)
        }
    }

    protected processOperator(operator: string) {
        if (["+", "-", "*", "/"].indexOf(operator) >= 0) {
            return operator;
        }
    }

    protected evaluateOperator(operator: string, left: number, right: number): 
        switch (this.operator) {
            case "+": return left + right;
            case "-": return left - right;
            case "*": return left * right;
            case "/": return left / right;
        }
    }

    private evaluate() {
```

```
        if (this.operator) {
            this.memory = this.evaluateOperator(this.operator, this.memory, thi
        }
        else {
            this.memory = this.current;
        }
        this.current = 0;
    }

    public handelChar(char: string) {
        if (char === "=") {
            this.evaluate();
            return;
        }
        else {
            let value = this.processDigit(char, this.current);
            if (value !== undefined) {
                this.current = value;
                return;
            }
            else {
                let value = this.processOperator(char);
                if (value !== undefined) {
                    this.evaluate();
                    this.operator = value;
                    return;
                }
            }
        }
        throw new Error(`Unsupported input: '${char}'`);
    }

    public getResult() {
        return this.memory;
    }
}

export function test(c: Calculator, input: string) {
    for (let i = 0; i < input.length; i++) {
        c.handelChar(input[i]);
    }

    console.log(`result of '${input}' is '${c.getResult()}'`);
}
```

这是使用导出的`test`函数来测试计算器。

TestCalculator.ts

```
import { Calculator, test } from "./Calculator";


let c = new Calculator();
test(c, "1+2*33/11="); // prints 9
```

现在扩展它，添加支持输入其它进制（十进制以外），让我们来创建`ProgrammerCalculator.ts`。

ProgrammerCalculator.ts

```
import { Calculator } from "./Calculator";

class ProgrammerCalculator extends Calculator {
    static digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B"

    constructor(public base: number) {
        super();
        if (base <= 0 || base > ProgrammerCalculator.digits.length) {
            throw new Error("base has to be within 0 to 16 inclusive.");
        }
    }

    protected processDigit(digit: string, currentValue: number) {
        if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
            return currentValue * this.base + ProgrammerCalculator.digits.index
        }
    }
}

// Export the new extended calculator as Calculator
export { ProgrammerCalculator as Calculator };

// Also, export the helper function
export { test } from "./Calculator";
```

新的`ProgrammerCalculator`模块导出的API与原先的`Calculator`模块很相似，但却没有改变原模块里的对象。
下面是测试ProgrammerCalculator类的代码：

TestProgrammerCalculator.ts

```
import { Calculator, test } from "./ProgrammerCalculator";

let c = new Calculator(2);
test(c, "001+010="); // prints 3
```

模块里不要使用命名空间

当初次进入基于模块的开发模式时，可能总会控制不住要将导出包裹在一个命名空间里。
模块具有其自己的作用域，并且只有导出的声明才会在模块外部可见。
记住这点，命名空间在使用模块时几乎没什么价值。

在组织方面，命名空间对于在全局作用域内对逻辑上相关的对象和类型进行分组是很便利的。
例如，在C#里，你会从`System.Collections`里找到所有集合的类型。
通过将类型有层次地组织在命名空间里，可以方便用户找到与使用那些类型。
然而，模块本身已经存在于文件系统之中，这是必须的。
我们必须通过路径和文件名找到它们，这已经提供了一种逻辑上的组织形式。
我们可以创建`/collections/generic/`文件夹，把相应模块放在这里面。

命名空间对解决全局作用域里命名冲突来说是很重要的。
比如，你可以有一个`My.Application.Customer.AddForm`和

`My.Application.Order.AddForm` -- 两个类型的名字相同，但命名空间不同。
然而，这对于模块来说却不是一个问题。
在一个模块里，没有理由两个对象拥有同一个名字。
从模块的使用角度来说，使用者会挑出他们用来引用模块的名字，所以也没有理由发生重名的情况。

> 更多关于模块和命名空间的资料查看[命名空间和模块](./Namespaces and Modules.md)

危险信号

以下均为模块结构上的危险信号。重新检查以确保你没有在对模块使用命名空间：

- 文件的顶层声明是`export namespace Foo { ... }`（删除`Foo`并把所有内容向上层移动一层）
- 文件只有一个`export class`或`export function`（考虑使用`export default`）
- 多个文件的顶层具有同样的`export namespace Foo {`（不要以为这些会合并到一个`Foo`中！）