TypeScript
专业人员笔记

Chapter 6: Functions

Section 6.1: Optional and Default Parameters

Section 6.2: Function as a parameter

Chapter 7: Classes

Section 7.1: Abstract Classes

Section 7.2: Simple class

Chapter 13: TypeScript basic examples

Section 13.1: 1 basic class inheritance example using extends and super keyword

Section 13.2: 2 static class variable example - count how many time method is being invoked

80+ 页

专业提示和技巧

TypeScript
Notes for Professionals

Chapter 6: Functions

Section 6.1: Optional and Default Parameters

Section 6.2: Function as a parameter

Chapter 7: Classes

Section 7.1: Abstract Classes

Section 7.2: Simple class

Chapter 13: TypeScript basic examples

Section 13.1: 1 basic class inheritance example using extends and super keyword

Section 13.2: 2 static class variable example - count how many time method is being invoked

80+ pages

of professional hints and tricks

目录

Contents

关于

About

第1章：TypeScript 入门

版本发布日期	
2.8.3	2018-04-20
2.8	2018-03-28
2.8 预发布版	2018-03-16
2.7.2	2018-02-16
2.7.1	2018-02-01
2.7 beta	2018-01-18
2.6.1	2017-11-01
2.5.2	2017-09-01
2.4.1	2017-06-28
2.3.2	2017-04-28
2.3.1	2017-04-25
2.3.0 beta	2017-04-04
2.2.2	2017-03-13
2.2	2017-02-17
2.1.6	2017-02-07
2.2 beta	2017-02-02
2.1.5	2017-01-05
2.1.4	2016-12-05
2.0.8	2016-11-08
2.0.7	2016-11-03
2.0.6	2016-10-23
2.0.5	2016-09-22
2.0 Beta	2016-07-08
1.8.10	2016-04-09
1.8.9	2016-03-16
1.8.5	2016-03-02
1.8.2	2016-02-17
1.7.5	2015-12-14
1.7	2015-11-20
1.6	2015-09-11
1.5.4	2015-07-15
1.5	2015-07-15
1.4	2015-01-13
1.3	2014-10-28
1.1.0.1	2014-09-23

第1.1节：安装与设置

背景

TypeScript 是 JavaScript 的一个带类型的超集，直接编译成 JavaScript 代码。TypeScript 文件通常使用.ts 扩展名。许多集成开发环境（IDE）支持 TypeScript，无需其他设置，但 TypeScript 也可以通过命令行使用 TypeScript Node.JS 包进行编译。

Chapter 1: Getting started with TypeScript

Version	Release Date
2.8.3	2018-04-20
2.8	2018-03-28
2.8 RC	2018-03-16
2.7.2	2018-02-16
2.7.1	2018-02-01
2.7 beta	2018-01-18
2.6.1	2017-11-01
2.5.2	2017-09-01
2.4.1	2017-06-28
2.3.2	2017-04-28
2.3.1	2017-04-25
2.3.0 beta	2017-04-04
2.2.2	2017-03-13
2.2	2017-02-17
2.1.6	2017-02-07
2.2 beta	2017-02-02
2.1.5	2017-01-05
2.1.4	2016-12-05
2.0.8	2016-11-08
2.0.7	2016-11-03
2.0.6	2016-10-23
2.0.5	2016-09-22
2.0 Beta	2016-07-08
1.8.10	2016-04-09
1.8.9	2016-03-16
1.8.5	2016-03-02
1.8.2	2016-02-17
1.7.5	2015-12-14
1.7	2015-11-20
1.6	2015-09-11
1.5.4	2015-07-15
1.5	2015-07-15
1.4	2015-01-13
1.3	2014-10-28
1.1.0.1	2014-09-23

Section 1.1: Installation and setup

Background

TypeScript is a typed superset of JavaScript that compiles directly to JavaScript code. TypeScript files commonly use the `.ts` extension. Many IDEs support TypeScript without any other setup required, but TypeScript can also be compiled with the TypeScript Node.JS package from the command line.

集成开发环境（IDE）

Visual Studio

- Visual Studio 2015 包含 TypeScript。
- Visual Studio 2013 更新 2 或更高版本包含 TypeScript，或者您可以 下载早期版本的 TypeScript 。

Visual Studio Code

- Visual Studio Code（vscode）提供上下文自动补全以及重构和调试工具，适用于 TypeScript。vscode 本身是用 TypeScript 实现的。支持 Mac OS X、Windows 和 Linux。

WebStorm

- WebStorm2016.2内置 TypeScript 和编译器。[WebStorm 不是免费软件]

IntelliJ IDEA

- IntelliJ IDEA2016.2通过 JetBrains 团队维护的插件支持 TypeScript 和编译器。[IntelliJ 不是免费软件]

Atom & atom-typescript

- Atom通过atom-typescript包支持 TypeScript。

Sublime Text

- Sublime Text通过TypeScript包支持 TypeScript。

安装命令行界面
安装Node.js
全局安装 npm 包

您可以全局安装 TypeScript，以便从任何目录访问它。

```
npm install -g typescript
```

或者

本地安装 npm 包

您可以本地安装 TypeScript 并保存到 package.json，以限制在某个目录内使用。

```
npm install typescript --save-dev
```
安装渠道

您可以从以下渠道安装：

- 稳定渠道：npm install typescript
- 测试渠道：npm install typescript@beta
- 开发渠道：npm install typescript@next

编译 TypeScript 代码

tsc编译命令随typescript一起提供，可用于编译代码。

```
tsc my-code.ts
```

这将创建一个my-code.js文件。

使用 tsconfig.json 编译

IDEs

Visual Studio

- Visual Studio 2015 includes TypeScript.
- Visual Studio 2013 Update 2 or later includes TypeScript, or you can download TypeScript for earlier versions.

Visual Studio Code

- Visual Studio Code (vscode) provides contextual autocomplete as well as refactoring and debugging tools for TypeScript. vscode is itself implemented in TypeScript. Available for Mac OS X, Windows and Linux.

WebStorm

- WebStorm 2016.2 comes with TypeScript and a built-in compiler. [WebStorm is not free]

IntelliJ IDEA

- IntelliJ IDEA 2016.2 has support for TypeScript and a compiler via a plugin maintained by the JetBrains team. [IntelliJ is not free]

Atom & atom-typescript

- Atom supports TypeScript with the atom-typescript package.

Sublime Text

- Sublime Text supports TypeScript with the TypeScript package.

Installing the command line interface
Install Node.js
Install the npm package globally

You can install TypeScript globally to have access to it from any directory.

```
npm install -g typescript
```

or

Install the npm package locally

You can install TypeScript locally and save to package.json to restrict to a directory.

```
npm install typescript --save-dev
```
Installation channels

You can install from:

- Stable channel: `npm install typescript`
- Beta channel: `npm install typescript@beta`
- Dev channel: `npm install typescript@next`

Compiling TypeScript code

The `tsc` compilation command comes with `typescript`, which can be used to compile code.

```
tsc my-code.ts
```

This creates a `my-code.js` file.

Compile using tsconfig.json

你也可以通过 tsconfig.json文件为代码提供随代码一起携带的编译选项。要开始一个新的 TypeScript项目，cd进入项目根目录的终端窗口并运行 tsc --init。该命令将生成一个带有最小配置选项的 tsconfig.json文件，类似如下。

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false,
        "pretty": true
    },
    "exclude": [
        "node_modules"
    ]
}
```

在你的 TypeScript 项目根目录放置一个 tsconfig.json 文件后，你可以使用 tsc 命令来运行编译。

第 1.2 节：基本语法

TypeScript 是 JavaScript 的一个带类型的超集，这意味着所有的 JavaScript 代码都是有效的 TypeScript 代码。TypeScript 在此基础上添加了许多新特性。

TypeScript 使 JavaScript 更像一种强类型的面向对象语言，类似于 C# 和 Java。这意味着TypeScript 代码更适合大型项目，代码也更容易理解和维护。强类型还意味着语言可以（并且确实）被预编译，变量不能被赋予超出其声明范围的值。例如，当一个 TypeScript 变量被声明为number 类型时，你不能给它赋值为文本。

这种强类型和面向对象的特性使得 TypeScript 更易于调试和维护，而这正是标准 JavaScript 的两个弱点。

类型声明

你可以为变量、函数参数和函数返回类型添加类型声明。类型写在变量名后面的冒号后面，格式如下：var num: number = 5; 编译器会在编译时（在可能的情况下）检查类型并报告类型错误。

```
var num: number = 5;
num = "this is a string";  // 错误：类型 'string' 不能赋值给类型 'number'。
```

基本类型有：

- 数字（包括整数和浮点数）
- 字符串
- boolean
- 数组。你可以指定数组元素的类型。有两种等效的方式来定义数组类型：Array<T> 和 T[]。例如：
  - number[] - 数字数组
  - Array<string> - 字符串数组
- 元组。元组具有固定数量的元素且每个元素有特定类型。
  - [boolean, string] - 元组，第一个元素是布尔值，第二个是字符串。
  - [number, number, number] - 三个数字的元组。

You can also provide compilation options that travel with your code via a tsconfig.json file. To start a new TypeScript project, cd into your project's root directory in a terminal window and run tsc --init. This command will generate a tsconfig.json file with minimal configuration options, similar to below.

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false,
        "pretty": true
    },
    "exclude": [
        "node_modules"
    ]
}
```

With a tsconfig.json file placed at the root of your TypeScript project, you can use the tsc command to run the compilation.

Section 1.2: Basic syntax

TypeScript is a typed superset of JavaScript, which means that all JavaScript code is valid TypeScript code. TypeScript adds a lot of new features on top of that.

TypeScript makes JavaScript more like a strongly-typed, object-oriented language akin to C# and Java. This means that TypeScript code tends to be easier to use for large projects and that code tends to be easier to understand and maintain. The strong typing also means that the language can (and is) precompiled and that variables cannot be assigned values that are out of their declared range. For instance, when a TypeScript variable is declared as a number, you cannot assign a text value to it.

This strong typing and object orientation makes TypeScript easier to debug and maintain, and those were two of the weakest points of standard JavaScript.

Type declarations

You can add type declarations to variables, function parameters and function return types. The type is written after a colon following the variable name, like this: `var num: number = 5;` The compiler will then check the types (where possible) during compilation and report type errors.

```
var num: number = 5;
num = "this is a string";  // error: Type 'string' is not assignable to type 'number'.
```

The basic types are :

- number (both integers and floating point numbers)
- string
- boolean
- Array. You can specify the types of an array's elements. There are two equivalent ways to define array types: Array<T> and T[]. For example:
  - number[] - array of numbers
  - Array<string> - array of strings
- Tuples. Tuples have a fixed number of elements with specific types.
  - [boolean, string] - tuple where the first element is a boolean and the second is a string.
  - [number, number, number] - tuple of three numbers.

- {} - 对象，你可以定义它的属性或索引器
  - {name: string, age: number} - 具有 name 和 age 属性的对象
  - {[key: string]: number} - 以字符串为索引的数字字典
- enum - { Red = 0, Blue, Green } - 映射到数字的枚举
- 函数。你需要为参数和返回值指定类型：
  - (param: 数字) => 字符串 - 接受一个数字参数并返回字符串的函数
  - () => 数字 - 无参数并返回数字的函数。
  - (a:字符串, b?: 布尔值) => 无返回值 - 接受一个字符串和一个可选布尔值且无返回值的函数。

- any - 允许任何类型。涉及any的表达式不进行类型检查。
- void - 表示"无"，可以用作函数的返回值。只有 null 和 undefined 是

  void 类型的一部分。
- 从不
  - let foo: never; -作为类型保护下永远不为真的变量类型。
  - function error(message: string): never { throw new Error(message); } - 作为永不返回的函数的返回类型。

- null - 值的类型 null。null 隐式地属于每种类型，除非启用了严格的 null 检查。

铸造

您可以通过尖括号进行显式类型转换，例如：

```
var derived: MyInterface;
(<ImplementingClass>derived).someSpecificMethod();
```

此示例展示了一个派生类，该类被编译器视为MyInterface。若第二行不进行类型转换，编译器会抛出异常，因为它无法识别someSpecificMethod()，但通过<ImplementingClass>派生的类型转换，编译器就知道该如何处理。

在 TypeScript 中，另一种类型转换方式是使用as关键字：

```
var derived: MyInterface;
(derived as ImplementingClass).someSpecificMethod();
```

自 TypeScript 1.6 起，默认使用as关键字，因为在.jsx文件中使用<>语法存在歧义。这一点在TypeScript 官方文档中有提及。

类

可以在 TypeScript 代码中定义和使用类。想了解更多关于类的信息，请参阅类文档页面。

第1.3节：你好，世界

```
class Greeter {
greeting: string;

    constructor(message: string) {
        this.问候语 = 消息;
    }
greet(): 字符串 {
        return this.问候语;
    }
};
```

- {} - object, you can define its properties or indexer
  - {name: string, age: number} - object with name and age attributes
  - {[key: string]: number} - a dictionary of numbers indexed by string
- enum - { Red = 0, Blue, Green } - enumeration mapped to numbers
- Function. You specify types for the parameters and return value:
  - (param: number) => string - function taking one number parameter returning string
  - () => number - function with no parameters returning an number.
  - (a: string, b?: boolean) => void - function taking a string and optionally a boolean with no return value.
- any - Permits any type. Expressions involving any are not type checked.
- void - represents "nothing", can be used as a function return value. Only null and undefined are part of the void type.
- never
  - let foo: never; -As the type of variables under type guards that are never true.
  - function error(message: string): never { throw new Error(message); } - As the return type of functions that never return.

- null - type for the value null. null is implicitly part of every type, unless strict null checks are enabled.

Casting

You can perform explicit casting through angle brackets, for instance:

```
var derived: MyInterface;
(<ImplementingClass>derived).someSpecificMethod();
```

This example shows a derived class which is treated by the compiler as a MyInterface. Without the casting on the second line the compiler would throw an exception as it does not understand someSpecificMethod(), but casting through <ImplementingClass>derived suggests the compiler what to do.

Another way of casting in TypeScript is using the as keyword:

```
var derived: MyInterface;
(derived as ImplementingClass).someSpecificMethod();
```

Since TypeScript 1.6, the default is using the as keyword, because using <> is ambiguous in **.jsx** files. This is mentioned in TypeScript official documentation.

Classes

Classes can be defined and used in TypeScript code. To learn more about classes, see the Classes documentation page.

Section 1.3: Hello World

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }
    greet(): string {
        return this.greeting;
    }
};
```

```
let 问候者 = new 问候者("Hello, world!");
console.log(问候者.greet());
```

这里有一个类，问候者，包含一个构造函数和一个greet方法。我们可以使用new关键字构造该类的实例，并传入一个字符串，greet方法会将其输出到控制台。我们的问候者类的实例存储在问候者变量中，然后我们用它来调用greet方法。

第1.4节：使用 ts-node 运行 TypeScript

ts-node 是一个 npm 包，允许用户直接运行 TypeScript 文件，无需使用tsc进行预编译。它还提供了R EPL。

全局安装 ts-node 使用

```
npm install -g ts-node
```

ts-node 不包含 TypeScript 编译器，因此您可能需要安装它。

```
npm install -g typescript
```

执行脚本

要执行名为main.ts的脚本，运行

ts-node main.ts

```
// main.ts
console.log("Hello world");
```

示例用法

```
$ ts-node main.ts
Hello world
```

运行REPL

要运行REPL，执行命令 `ts-node`

示例用法

```
$ ts-node
> const sum = (a, b): number => a + b;
undefined
> sum(2, 2)
4
>
   exit
```

要退出 REPL，请使用命令 .exit 或按两次 CTRL+C。

第1.5节：Node.js中的TypeScript REPL

要在Node.js中使用TypeScript REPL，可以使用 tsun 包

通过以下命令全局安装

```
let greeter = new Greeter("Hello, world!");
console.log(greeter.greet());
```

Here we have a class, `Greeter`, that has a `constructor` and a `greet` method. We can construct an instance of the class using the **`new`** keyword and pass in a string we want the `greet` method to output to the console. The instance of our `Greeter` class is stored in the `greeter` variable which we then us to call the `greet` method.

Section 1.4: Running TypeScript using ts-node

ts-node is an npm package which allows the user to run typescript files directly, without the need for precompilation using `tsc`. It also provides REPL.

Install ts-node globally using

```
npm install -g ts-node
```

ts-node does not bundle typescript compiler, so you might need to install it.

```
npm install -g typescript
```

Executing script

To execute a script named *main.ts*, run

```
ts-node main.ts
```

```
// main.ts
console.log("Hello world");
```

Example usage

```
$ ts-node main.ts
Hello world
```

Running REPL

To run REPL run command `ts-node`

Example usage

```
$ ts-node
> const sum = (a, b): number => a + b;
undefined
> sum(2, 2)
4
> .exit
```

To exit REPL use command `.exit` or press `CTRL+C` twice.

Section 1.5: TypeScript REPL in Node.js

For use TypeScript REPL in Node.js you can use tsun package

Install it globally with

```
npm install -g tsun
```

然后在终端或命令提示符中运行 `tsun` 命令

使用示例：

```
$ tsun
TSUN : TypeScript 升级版 Node
类型 在 TypeScript 表达式中求值
类型 :帮助 用于 repl 中的命令
$ 函数 multiply(x, y) {
..返回 x * y;
..}
未定义
$ multiply(3, 4)
12
```

```
npm install -g tsun
```

and run in your terminal or command prompt with `tsun` command

Usage example:

```
$ tsun
TSUN : TypeScript Upgraded Node
type in TypeScript expression to evaluate
type :help for commands in repl
$ function multiply(x, y) {
..return x * y;
..}
undefined
$ multiply(3, 4)
12
```

第2章：为什么以及何时使用 TypeScript

如果你普遍认同类型系统的论点，那么你会喜欢 TypeScript。

它为 JavaScript 生态系统带来了类型系统的许多优势（安全性、可读性、改进的工具支持）。它也存在类型系统的一些缺点（增加的复杂性和不完整性）。

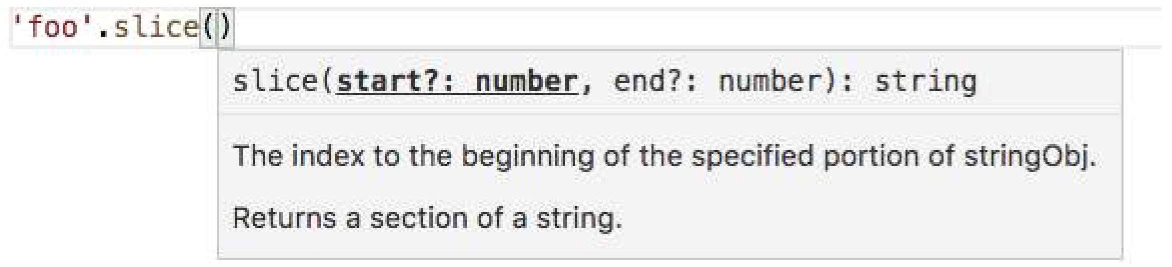
第2.1节：安全性

TypeScript 通过静态分析及早捕获类型错误：

```
function double(x: number): number {
  return 2 * x;
}
double('2');
//     ~~~ 类型 '"2"' 的参数不能赋给类型为 'number' 的参数。
```

第2.2节：可读性

TypeScript 使编辑器能够提供上下文文档：



```
'foo'.slice()
slice(start?: number, end?: number): string
```

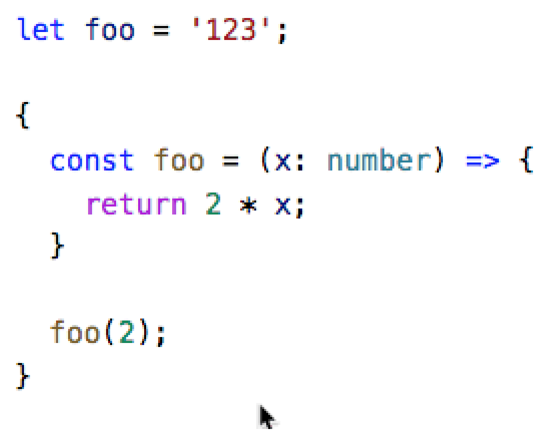
The index to the beginning of the specified portion of stringObj.

Returns a section of a string.

你再也不会忘记 String.prototype.slice 是接受 (start, stop) 还是 (start, length) 了！

第2.3节：工具支持

TypeScript 允许编辑器执行了解语言规则的自动重构。



```
let foo = '123';

{
  const foo = (x: number) => {
    return 2 * x;
  }

  foo(2);
}
```

例如，这里 Visual Studio Code 能够重命名内部的 foo 引用，而不改变外部的 foo。仅用简单的查找/替换很难做到这一点。

Chapter 2: Why and when to use TypeScript

If you find the arguments for type systems persuasive in general, then you'll be happy with TypeScript.

It brings many of the advantages of type system (safety, readability, improved tooling) to the JavaScript ecosystem. It also suffers from some of the drawbacks of type systems (added complexity and incompleteness).

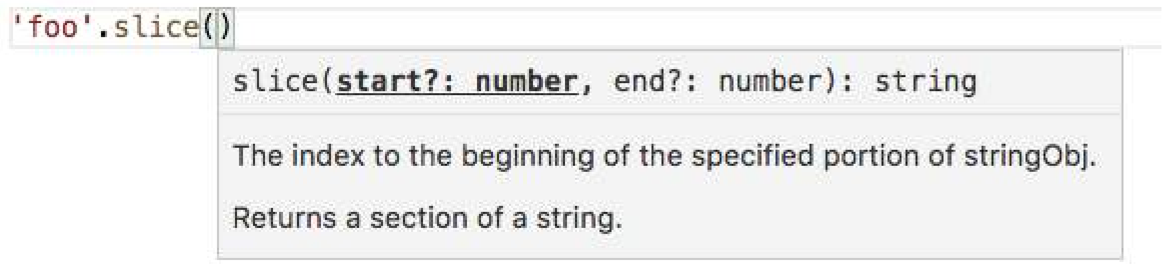
Section 2.1: Safety

TypeScript catches type errors early through static analysis:

```
function double(x: number): number {
  return 2 * x;
}
double('2');
//     ~~~ Argument of type '"2"' is not assignable to parameter of type 'number'.
```

Section 2.2: Readability

TypeScript enables editors to provide contextual documentation:



```
'foo'.slice()
slice(start?: number, end?: number): string
```

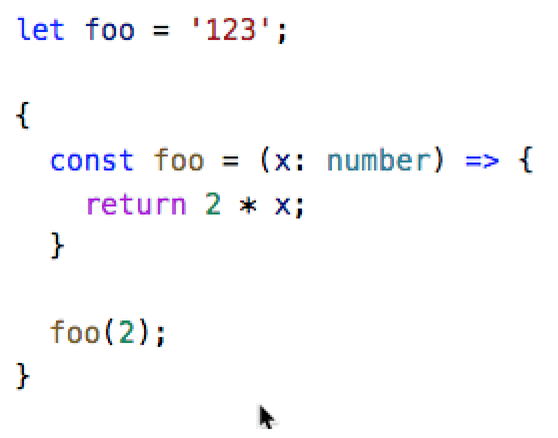
The index to the beginning of the specified portion of stringObj.

Returns a section of a string.

You'll never forget whether `String.prototype.slice` takes `(start, stop)` or `(start, length)` again!

Section 2.3: Tooling

TypeScript allows editors to perform automated refactors which are aware of the rules of the languages.



```
let foo = '123';

{
  const foo = (x: number) => {
    return 2 * x;
  }

  foo(2);
}
```

Here, for instance, Visual Studio Code is able to rename references to the inner `foo` without altering the outer `foo`. This would be difficult to do with a simple find/replace.

第3章：TypeScript核心类型

第3.1节：字符串字面量类型

字符串字面量类型允许你指定字符串可以具有的确切值。

```
let myFavoritePet: "dog";
myFavoritePet = "dog";
```

任何其他字符串都会报错。

```
// 错误：类型 '"rock"' 不能赋值给类型 '"dog"'。
// myFavoritePet = "rock";
```

结合类型别名和联合类型，你可以获得类似枚举的行为。

```
type Species = "cat" | "dog" | "bird";

function buyPet(pet: Species, name: string) : Pet { /*...*/ }

buyPet(myFavoritePet /* 上面定义的 "dog" */, "Rocky");

// 错误：类型 '"rock"' 的参数不能赋值给类型 "'cat' | \"dog\" | \"bird\"" 的参数。
类型 '"rock"' 不能赋值给类型 '"bird"'。
// buyPet("rock", "Rocky");
```

字符串字面量类型可用于区分重载。

```
function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet { /*...*/ }

let dog = buyPet(myFavoritePet /* 上面定义的 "dog" */, "Rocky");
// dog 的类型是 Dog (dog: Dog)
```

它们非常适合用户自定义类型保护。

```
interface Pet {
    species: Species;
    eat();
sleep();
}

interface Cat extends Pet {
    species: "cat";
}

interface Bird extends Pet {
    species: "bird";
sing();
}

function petIsCat(pet: Pet): pet 是 Cat {
    return pet.species === "cat";
}
```

Chapter 3: TypeScript Core Types

Section 3.1: String Literal Types

String literal types allow you to specify the exact value a string can have.

```
let myFavoritePet: "dog";
myFavoritePet = "dog";
```

Any other string will give an error.

```
// Error: Type '"rock"' is not assignable to type '"dog"'.
// myFavoritePet = "rock";
```

Together with Type Aliases and Union Types you get a enum-like behavior.

```
type Species = "cat" | "dog" | "bird";

function buyPet(pet: Species, name: string) : Pet { /*...*/ }

buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");

// Error: Argument of type '"rock"' is not assignable to parameter of type "'cat' | "dog" | "bird".
Type '"rock"' is not assignable to type '"bird"'.
// buyPet("rock", "Rocky");
```

String Literal Types can be used to distinguish overloads.

```
function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet { /*...*/ }

let dog = buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");
// dog is from type Dog (dog: Dog)
```

They works well for User-Defined Type Guards.

```
interface Pet {
    species: Species;
    eat();
    sleep();
}

interface Cat extends Pet {
    species: "cat";
}

interface Bird extends Pet {
    species: "bird";
    sing();
}

function petIsCat(pet: Pet): pet is Cat {
    return pet.species === "cat";
}
```

```
function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet){
    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)
pet.eat();
        pet.sleep();
    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)
pet.eat();
        pet.sing();
        pet.sleep();
    }
}
```

完整示例代码

```
let myFavoritePet: "dog";
myFavoritePet = "dog";

// 错误：类型 '"rock"' 不能赋值给类型 '"dog"'。
// myFavoritePet = "rock";

type Species = "cat" | "dog" | "bird";

interface Pet {
species: Species;
    name: string;
    eat();
walk();
    sleep();
}

interface Cat extends Pet {
    species: "cat";
}

interface 狗 extends 宠物 {
   物种: "dog";
}

interface Bird extends Pet {
    species: "bird";
sing();
}

// 错误：接口 'Rock' 错误地继承了接口 'Pet'。属性 'species' 的类型不兼容。类型 '"rock"' 不能赋值给类型 '"cat" | "dog
" | "bird"'。类型 '"rock"' 不能赋值给类型 '"bird"'。

// interface Rock extends Pet {
//      类型: "rock";
// }

function 购买宠物(宠物: 物种, 名字: string) : 宠物;
function 购买宠物(宠物: "cat", 名字: string): 猫;
function 购买宠物(宠物: "dog", 名字: string): 狗;
function 购买宠物(宠物: "bird", 名字: string): 鸟;
function 购买宠物(宠物: 物种, 名字: string) : 宠物 {
    if(宠物 === "cat") {
```

```
function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet){
    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)
        pet.eat();
        pet.sleep();
    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)
        pet.eat();
        pet.sing();
        pet.sleep();
    }
}
```

Full example code

```
let myFavoritePet: "dog";
myFavoritePet = "dog";

// Error: Type '"rock"' is not assignable to type '"dog"'.
// myFavoritePet = "rock";

type Species = "cat" | "dog" | "bird";

interface Pet {
    species: Species;
    name: string;
    eat();
    walk();
    sleep();
}

interface Cat extends Pet {
    species: "cat";
}

interface Dog extends Pet {
    species: "dog";
}

interface Bird extends Pet {
    species: "bird";
    sing();
}

// Error: Interface 'Rock' incorrectly extends interface 'Pet'. Types of property 'species' are
incompatible. Type '"rock"' is not assignable to type '"cat" | "dog" | "bird"'. Type '"rock"' is not
assignable to type '"bird"'.
// interface Rock extends Pet {
//      type: "rock";
// }

function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet {
    if(pet === "cat") {
        return {
物种: "cat",
          名字: 名字,
吃: function () {
                console.log(`${this.名字} 吃东西.`);
            }, 走: function () {
console.log(`${this.name} 走路。`);
            }, sleep: function () {
console.log(`${this.name} 睡觉.`);
            }
        } 作为 猫;
    } 否则如果(pet === "dog") {
        返回 {
物种: "狗",
          名字: name,
吃: function () {
                console.log(`${this.名字} 吃东西.`);
            }, 走: function () {
console.log(`${this.name} 走路。`);
            }, sleep: function () {
console.log(`${this.name} 睡觉.`);
            }
        } 作为 狗;
    } 否则如果(pet === "bird") {
        返回 {
物种: "鸟",
          名字: name,
吃: function () {
                console.log(`${this.名字} 吃东西.`);
            }, 走: function () {
console.log(`${this.name} 走路。`);
            }, sleep: function () {
console.log(`${this.name} 睡觉.`);
            }, 唱歌: 函数 () {
console.log(`${this.name} 唱歌.`);
            }
        } 作为 鸟;
    } 否则 {
        throw `抱歉，我们没有 ${pet}。 您想买一只狗吗?`;
    }
}

function petIsCat(pet: Pet): pet 是 Cat {
    return pet.species === "cat";
}

function petIsDog(pet: 宠物): pet 是 狗 {
    return pet.种类 === "dog";
}

function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: 宠物) {
    console.log(`嘿 ${pet.名字}, 一起玩吧.`);

    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)

pet.吃();
        pet.睡觉();
```

```
        return {
            species: "cat",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }
        } as Cat;
    } else if(pet === "dog") {
        return {
            species: "dog",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }
        } as Dog;
    } else if(pet === "bird") {
        return {
            species: "bird",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }, sing: function () {
                console.log(`${this.name} sings.`);
            }
        } as Bird;
    } else {
        throw `Sorry we do not have a ${pet}. Would you like to buy a dog?`;
    }
}

function petIsCat(pet: Pet): pet is Cat {
    return pet.species === "cat";
}

function petIsDog(pet: Pet): pet is Dog {
    return pet.species === "dog";
}

function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet) {
    console.log(`Hey ${pet.name}, lets play.`);

    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)

        pet.eat();
        pet.sleep();
        // 错误: 类型 '"bird"' 不能赋值给类型 '"cat"'。
        // pet.type = "bird";

        // 错误: 类型 'Cat' 上不存在属性 'sing'。
        // pet.sing();

    } 否则如果(petIsDog(pet)) {
        // pet 现在是 Dog 类型 (pet: Dog)

pet.吃();
        pet.走();
        pet.睡觉();

    } 否则如果(pet是鸟类(pet)) {
        // pet 现在是 Bird 类型 (pet: Bird)

pet.吃();
        pet.唱歌();
        pet.睡觉();
    } 否则 {
        抛出 "未知的宠物。你买了一块石头吗？";
    }
}

让 狗 = 购买宠物(我最喜欢的宠物 /* 上面定义的 "dog" */, "Rocky");
// dog 是 Dog 类型 (dog: Dog)

              // 错误：类型 '"rock"' 的参数不能赋值给类型 '"cat' | 'dog' | 'bird"' 的参数。
类型 '"rock"' 不能赋值给类型 '"bird"'。
// buyPet("rock", "Rocky");

和宠物玩(狗);
// 输出：嘿 Rocky，来玩吧。
//        Rocky 吃东西。
//        Rocky 走路。
//        Rocky 睡觉。
```

第3.2节：元组

已知且可能不同类型的数组类型：

```
let day: [数字,字符串];
day = [0, '星期一'];      // 有效
day = ['零', '星期一']; // 无效：'零' 不是数字类型
console.log(day[0]); // 0
console.log(day[1]); // 星期一

day[2] = '星期六'; // 有效: [0, '星期六']
day[3] = false;      // 无效: 必须是 '数字 | 字符串' 的联合类型
```

第3.3节：布尔值

布尔值表示 TypeScript 中最基本的数据类型，目的是赋值 true/false。

```
// 设置初始值 (true 或 false)
let isTrue: boolean = true;

// 未显式设置时默认为 'undefined'
let unsetBool: boolean;
```

```
        // Error: Type '"bird"' is not assignable to type '"cat"'.
        // pet.type = "bird";

        // Error: Property 'sing' does not exist on type 'Cat'.
        // pet.sing();

    } else if(petIsDog(pet)) {
        // pet is now from type Dog (pet: Dog)

        pet.eat();
        pet.walk();
        pet.sleep();

    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)

        pet.eat();
        pet.sing();
        pet.sleep();
    } else {
        throw "An unknown pet. Did you buy a rock?";
    }
}

let dog = buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");
// dog is from type Dog (dog: Dog)

// Error: Argument of type '"rock"' is not assignable to parameter of type "'cat' | "dog" | "bird".
Type '"rock"' is not assignable to type '"bird"'.
// buyPet("rock", "Rocky");

playWithPet(dog);
// Output: Hey Rocky, lets play.
//         Rocky eats.
//         Rocky walks.
//         Rocky sleeps.
```

Section 3.2: Tuple

Array type with known and possibly different types:

```
let day: [number, string];
day = [0, 'Monday'];      // valid
day = ['zero', 'Monday']; // invalid: 'zero' is not numeric
console.log(day[0]); // 0
console.log(day[1]); // Monday

day[2] = 'Saturday'; // valid: [0, 'Saturday']
day[3] = false;      // invalid: must be union type of 'number | string'
```

Section 3.3: Boolean

A boolean represents the most basic datatype in TypeScript, with the purpose of assigning true/false values.

```
// set with initial value (either true or false)
let isTrue: boolean = true;

// defaults to 'undefined', when not explicitly set
let unsetBool: boolean;
```

```
// 也可以设置为 'null'
let nullableBool: boolean = null;
```

第3.4节：交叉类型

交叉类型结合了两个或多个类型的成员。

```
interface Knife {
    cut();
}

interface BottleOpener{
    openBottle();
}

interface Screwdriver{
    turnScrew();
}

type SwissArmyKnife = Knife & BottleOpener & Screwdriver;

function use(tool: SwissArmyKnife){
    console.log("我什么都能做！");

    tool.cut();
tool.openBottle();
    tool.turnScrew();
}
```

第3.5节：函数参数和返回值中的类型。数字

当你在TypeScript中创建函数时，可以指定函数参数的数据类型以及返回值的数据类型

示例：

```
function sum(x: number, y: number): number {
    return x + y;
}
```

这里的语法 x: number, y: number 表示函数可以接受两个参数 x 和 y，且它们只能是数字，(...): number { 表示返回值只能是数字

用法：

```
sum(84 + 76) // 将返回160
```

注意：

你不能这样做

```
function sum(x: string, y: string): number {
    return x + y;
}
```

或者

```
// can also be set to 'null' as well
let nullableBool: boolean = null;
```

Section 3.4: Intersection Types

A Intersection Type combines the member of two or more types.

```
interface Knife {
    cut();
}

interface BottleOpener{
    openBottle();
}

interface Screwdriver{
    turnScrew();
}

type SwissArmyKnife = Knife & BottleOpener & Screwdriver;

function use(tool: SwissArmyKnife){
    console.log("I can do anything!");

    tool.cut();
    tool.openBottle();
    tool.turnScrew();
}
```

Section 3.5: Types in function arguments and return value. Number

When you create a function in TypeScript you can specify the data type of the function's arguments and the data type for the return value

Example:

```
function sum(x: number, y: number): number {
    return x + y;
}
```

Here the syntax `x: number, y: number` means that the function can accept two argumentsx and y and they can only be numbers and `(...): number {` means that the return value can only be a number

Usage:

```
sum(84 + 76) // will be return 160
```

Note:

You can not do so

```
function sum(x: string, y: string): number {
    return x + y;
}
```

or

```
function sum(x: number, y: number): string {
    return x + y;
}
```

它将收到以下错误：

错误 TS2322: 类型 'string' 不能赋值给类型 'number'，且错误 TS2322: 类型 'number' 不能赋值给类型 'string'

第3.6节：函数参数和返回值的类型。字符串

示例：

```
function hello(name: string): string {
    return `Hello ${name}!`;
}
```

这里的语法 name: string 表示函数可以接受一个 name 参数，且该参数只能是字符串，(...): string { 表示返回值只能是字符串

用法：

```
hello('StackOverflow Documentation') // 将返回 Hello StackOverflow Documentation!
```

第3.7节：const 枚举

const 枚举与普通枚举相同。不同之处在于编译时不会生成对象。取而代之的是，在使用 const 枚举的地方会直接替换为字面量值。

```
// TypeScript：const 枚举可以像普通枚举一样定义（带起始值、特定值等）

const enum 忍者活动 {
    间谍活动,
破坏,
    暗杀
}

// JavaScript：但不会生成任何代码

// TypeScript：除非你使用它
let 我最喜欢的忍者活动 = 忍者活动.间谍活动;
console.log(我最喜欢的海盗活动); // 0

// JavaScript：然后只有值的数字被编译进代码
// var 我最喜欢的忍者活动 = 0 /* 间谍活动 */;
// console.log(我最喜欢的海盗活动); // 0

// TypeScript：其他常量示例也是一样
console.log(忍者活动["破坏"]); // 1

// JavaScript：只有数字，注释中带值的名称
// console.log(1 /* "破坏" */); // 1

// TypeScript：但没有对象，运行时无法访问
// 错误：const 枚举成员只能通过字符串字面量访问。
// console.log(忍者活动[我最喜欢的忍者活动]);
```

```
function sum(x: number, y: number): string {
    return x + y;
}
```

it will receive the following errors:

`error TS2322: Type 'string' is not assignable to type 'number'` and `error TS2322: Type 'number' is not assignable to type 'string'` respectively

Section 3.6: Types in function arguments and return value. String

Example:

```
function hello(name: string): string {
    return `Hello ${name}!`;
}
```

Here the syntax `name: string` means that the function can accept one `name` argument and this argument can only be string and `(...): string {` means that the return value can only be a string

Usage:

```
hello('StackOverflow Documentation') // will be return Hello StackOverflow Documentation!
```

Section 3.7: const Enum

A const Enum is the same as a normal Enum. Except that no Object is generated at compile time. Instead, the literal values are substituted where the const Enum is used.

```
// TypeScript: A const Enum can be defined like a normal Enum (with start value, specific values,
etc.)
const enum NinjaActivity {
    Espionage,
    Sabotage,
    Assassination
}

// JavaScript: But nothing is generated

// TypeScript: Except if you use it
let myFavoriteNinjaActivity = NinjaActivity.Espionage;
console.log(myFavoritePirateActivity); // 0

// JavaScript: Then only the number of the value is compiled into the code
// var myFavoriteNinjaActivity = 0 /* Espionage */;
// console.log(myFavoritePirateActivity); // 0

// TypeScript: The same for the other constant example
console.log(NinjaActivity["Sabotage"]); // 1

// JavaScript: Just the number and in a comment the name of the value
// console.log(1 /* "Sabotage" */); // 1

// TypeScript: But without the object none runtime access is possible
// Error: A const enum member can only be accessed using a string literal.
// console.log(NinjaActivity[myFavoriteNinjaActivity]);
```

作为比较，普通的枚举

```
// TypeScript：一个普通的枚举
enum PirateActivity {
    登船,
喝酒,
    击剑
}

// JavaScript：编译后的枚举
// var PirateActivity;
// (function (PirateActivity) {
//     PirateActivity[PirateActivity["登船"] = 0] = "登船";
//     PirateActivity[PirateActivity["喝酒"] = 1] = "喝酒";
//     PirateActivity[PirateActivity["击剑"] = 2] = "击剑";
// })(PirateActivity || (PirateActivity = {}));

// TypeScript：该枚举的一个普通用法
let myFavoritePirateActivity = PirateActivity.登船;
console.log(myFavoritePirateActivity); // 0

// JavaScript：在JavaScript中看起来非常相似
// var myFavoritePirateActivity = PirateActivity.Boarding;
// console.log(myFavoritePirateActivity); // 0

// TypeScript：以及其他一些普通用法
console.log(PirateActivity["Drinking"]); // 1

// JavaScript: 在JavaScript中看起来非常相似
// console.log(PirateActivity["Drinking"]); // 1

// TypeScript: 在运行时，你可以访问一个普通的枚举
console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"

// JavaScript: 并且它将在运行时被解析
// console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"
```

第3.8节：数字

和JavaScript一样，数字是浮点数值。

```
let pi: number = 3.14;           // 默认十进制
let hexadecimal: number = 0xFF;  // 十进制的255
```

ECMAScript 2015支持二进制和八进制。

```
let binary: number = 0b10;   // 十进制的2
let octal: number = 0o755;   // 十进制的493
```

第3.9节：字符串

文本数据类型：

```
let singleQuotes: string = 'single';
let doubleQuotes: string = "double";
let templateString: string = `I am ${ singleQuotes }`; // I am single
```

For comparison, a normal Enum

```
// TypeScript: A normal Enum
enum PirateActivity {
    Boarding,
    Drinking,
    Fencing
}

// JavaScript: The Enum after the compiling
// var PirateActivity;
// (function (PirateActivity) {
//     PirateActivity[PirateActivity["Boarding"] = 0] = "Boarding";
//     PirateActivity[PirateActivity["Drinking"] = 1] = "Drinking";
//     PirateActivity[PirateActivity["Fencing"] = 2] = "Fencing";
// })(PirateActivity || (PirateActivity = {}));

// TypeScript: A normal use of this Enum
let myFavoritePirateActivity = PirateActivity.Boarding;
console.log(myFavoritePirateActivity); // 0

// JavaScript: Looks quite similar in JavaScript
// var myFavoritePirateActivity = PirateActivity.Boarding;
// console.log(myFavoritePirateActivity); // 0

// TypeScript: And some other normal use
console.log(PirateActivity["Drinking"]); // 1

// JavaScript: Looks quite similar in JavaScript
// console.log(PirateActivity["Drinking"]); // 1

// TypeScript: At runtime, you can access an normal enum
console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"

// JavaScript: And it will be resolved at runtime
// console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"
```

Section 3.8: Number

Like JavaScript, numbers are floating point values.

```
let pi: number = 3.14;           // base 10 decimal by default
let hexadecimal: number = 0xFF;  // 255 in decimal
```

ECMAScript 2015 allows binary and octal.

```
let binary: number = 0b10;   // 2 in decimal
let octal: number = 0o755;   // 493 in decimal
```

Section 3.9: String

Textual data type:

```
let singleQuotes: string = 'single';
let doubleQuotes: string = "double";
let templateString: string = `I am ${ singleQuotes }`; // I am single
```

第3.10节：数组

一个值的数组：

```
let threePigs: number[] = [1, 2, 3];
let genericStringArray: Array<string> = ['first', '2nd', '3rd'];
```

第3.11节：枚举

用于命名一组数字值的类型：

数字值默认从0开始：

```
enum Day { 星期一, 星期二, 星期三, 星期四, 星期五, 星期六, 星期日 };
let bestDay: Day = Day.星期六;
```

设置默认起始数字：

```
enum TenPlus { 十 = 10, 十一, 十二 }
```

或指定数值：

```
enum MyOddSet { 三 = 3, 五 = 5, 七 = 7, 九 = 9 }
```

第3.12节：Any

当不确定类型时，any 可用：

```
let anything: any = '我是一个字符串';
anything = 5; // 但现在我是数字5
```

第3.13节：Void

如果您根本没有类型，通常用于不返回任何内容的函数：

```
function log(): void {
    console.log('我不返回任何内容');
}
```

void 类型只能被赋值为 null 或 undefined。

Section 3.10: Array

An array of values:

```
let threePigs: number[] = [1, 2, 3];
let genericStringArray: Array<string> = ['first', '2nd', '3rd'];
```

Section 3.11: Enum

A type to name a set of numeric values:

Number values default to 0:

```
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
let bestDay: Day = Day.Saturday;
```

Set a default starting number:

```
enum TenPlus { Ten = 10, Eleven, Twelve }
```

or assign values:

```
enum MyOddSet { Three = 3, Five = 5, Seven = 7, Nine = 9 }
```

Section 3.12: Any

When unsure of a type, any is available:

```
let anything: any = 'I am a string';
anything = 5; // but now I am the number 5
```

Section 3.13: Void

If you have no type at all, commonly used for functions that do not return anything:

```
function log(): void {
    console.log('I return nothing');
}
```

void types Can only be assigned **null** or **undefined**.

第4章：数组

第4.1节：在数组中查找对象

使用 find()

```
const inventory = [
    {name: '苹果', quantity: 2},
    {name: '香蕉', quantity: 0},
    {name: '樱桃', quantity: 5}
];

function findCherries(fruit) {
    return fruit.name === '樱桃';
}

inventory.find(findCherries); // { name: '樱桃', quantity: 5 }

/* 或者 */

inventory.find(e => e.name === '苹果'); // { name: '苹果', quantity: 2 }
```

Chapter 4: Arrays

Section 4.1: Finding Object in Array

Using find()

```
const inventory = [
    {name: 'apples', quantity: 2},
    {name: 'bananas', quantity: 0},
    {name: 'cherries', quantity: 5}
];

function findCherries(fruit) {
    return fruit.name === 'cherries';
}

inventory.find(findCherries); // { name: 'cherries', quantity: 5 }

/* OR */

inventory.find(e => e.name === 'apples'); // { name: 'apples', quantity: 2 }
```

第5章：枚举

第5.1节：带显式值的枚举

默认情况下，所有枚举值都会被解析为数字。假设你有如下内容

```
enum MimeType {
  JPEG,
PNG,
  PDF
}
```

例如，MimeType.PDF背后的实际值将是2。

但有时让枚举解析为不同类型很重要。例如，你从
后端/前端/另一个系统接收到的值肯定是字符串。这可能会很麻烦，但幸运的是有这个
方法：

```
enum MimeType {
JPEG = <any>'image/jpeg',
  PNG = <any>'image/png',
  PDF = <any>'application/pdf'
}
```

这会将MimeType.PDF解析为application/pdf。

自 TypeScript 2.4 起，可以声明字符串枚举：

```
enum MimeType {
  JPEG = 'image/jpeg',
  PNG = 'image/png',
  PDF = 'application/pdf',
}
```

你可以使用相同的方法显式提供数值

```
enum MyType {
   Value = 3,
   ValueEx = 30,
   ValueEx2 = 300
}
```

更复杂的类型也可以，因为非常量枚举在运行时是真实对象，例如

```
enum FancyType {
   OneArr = <any>[1],
   TwoArr = <any>[2, 2],
   ThreeArr = <any>[3, 3, 3]
}
```

变成

```
var FancyType;
(function (FancyType) {
FancyType[FancyType["OneArr"] = [1]] = "OneArr";
    FancyType[FancyType["TwoArr"] = [2, 2]] = "TwoArr";
```

Chapter 5: Enums

Section 5.1: Enums with explicit values

By default all enum values are resolved to numbers. Let's say if you have something like

```
enum MimeType {
  JPEG,
  PNG,
  PDF
}
```

the real value behind e.g. MimeType.PDF will be 2.

But some of the time it is important to have the enum resolve to a different type. E.g. you receive the value from backend / frontend / another system which is definitely a string. This could be a pain, but luckily there is this method:

```
enum MimeType {
  JPEG = <any>'image/jpeg',
  PNG = <any>'image/png',
  PDF = <any>'application/pdf'
}
```

This resolves the MimeType.PDF to application/pdf.

Since TypeScript 2.4 it's possible to declare string enums:

```
enum MimeType {
  JPEG = 'image/jpeg',
  PNG = 'image/png',
  PDF = 'application/pdf',
}
```

You can explicitly provide numeric values using the same method

```
enum MyType {
   Value = 3,
   ValueEx = 30,
   ValueEx2 = 300
}
```

Fancier types also work, since non-const enums are real objects at runtime, for example

```
enum FancyType {
   OneArr = <any>[1],
   TwoArr = <any>[2, 2],
   ThreeArr = <any>[3, 3, 3]
}
```

becomes

```
var FancyType;
(function (FancyType) {
    FancyType[FancyType["OneArr"] = [1]] = "OneArr";
    FancyType[FancyType["TwoArr"] = [2, 2]] = "TwoArr";
FancyType[FancyType["ThreeArr"] = [3, 3, 3]] = "ThreeArr";
})(FancyType || (FancyType = {}));
```

第5.2节：如何获取所有枚举值

```
enum SomeEnum { A, B }

let enumValues:Array<string>= [];

for(let value in SomeEnum) {
    if(typeof SomeEnum[value] === 'number') {
        enumValues.push(value);
    }
}

enumValues.forEach(v=> console.log(v))
//A
//B
```

第5.3节：在不自定义枚举实现的情况下扩展枚举

```
enum SourceEnum {
  value1 = <any>'value1',
  value2 = <any>'value2'
}

enum AdditionToSourceEnum {
  value3 = <any>'value3',
  value4 = <any>'value4'
}

// 我们需要这个类型以便TypeScript正确解析类型
类型 TestEnumType = SourceEnum | AdditionToSourceEnum;
// 我们需要这个值 "instance" 来使用值
let TestEnum = Object.assign({}, SourceEnum, AdditionToSourceEnum);
// TypeScript 2 的特性也同样适用
// let TestEnum = { ...SourceEnum, ...AdditionToSourceEnum };

function check(test: TestEnumType) {
  return test === TestEnum.value2;
}

console.log(TestEnum.value1);
console.log(TestEnum.value2 === <any>'value2');
console.log(check(TestEnum.value2));
console.log(check(TestEnum.value3));
```

第5.4节：自定义枚举实现：枚举的扩展

有时需要自己实现枚举。例如，没有明确的方法来扩展其他枚举。
自定义实现允许这样做：

```
class Enum {
constructor(protected value: string) {}

  public toString() {
    return String(this.value);
```

```
    FancyType[FancyType["ThreeArr"] = [3, 3, 3]] = "ThreeArr";
})(FancyType || (FancyType = {}));
```

Section 5.2: How to get all enum values

```
enum SomeEnum { A, B }

let enumValues:Array<string>= [];

for(let value in SomeEnum) {
    if(typeof SomeEnum[value] === 'number') {
        enumValues.push(value);
    }
}

enumValues.forEach(v=> console.log(v))
//A
//B
```

Section 5.3: Extending enums without custom enum implementation

```
enum SourceEnum {
  value1 = <any>'value1',
  value2 = <any>'value2'
}

enum AdditionToSourceEnum {
  value3 = <any>'value3',
  value4 = <any>'value4'
}

// we need this type for TypeScript to resolve the types correctly
type TestEnumType = SourceEnum | AdditionToSourceEnum;
// and we need this value "instance" to use values
let TestEnum = Object.assign({}, SourceEnum, AdditionToSourceEnum);
// also works fine the TypeScript 2 feature
// let TestEnum = { ...SourceEnum, ...AdditionToSourceEnum };

function check(test: TestEnumType) {
  return test === TestEnum.value2;
}

console.log(TestEnum.value1);
console.log(TestEnum.value2 === <any>'value2');
console.log(check(TestEnum.value2));
console.log(check(TestEnum.value3));
```

Section 5.4: Custom enum implementation: extends for enums

Sometimes it is required to implement Enum on your own. E.g. there is no clear way to extend other enums. Custom implementation allows this:

```
class Enum {
  constructor(protected value: string) {}

  public toString() {
    return String(this.value);
  }

public is(value: Enum | string) {
    return this.value = value.toString();
  }
}

class SourceEnum extends Enum {
  public static value1 = new SourceEnum('value1');
  public static value2 = new SourceEnum('value2');
}

class TestEnum extends SourceEnum {
  public static value3 = new TestEnum('value3');
  public static value4 = new TestEnum('value4');
}

function check(test: TestEnum) {
  return test === TestEnum.value2;
}

let value1 = TestEnum.value1;

console.log(value1 + 'hello');
console.log(value1.toString() === 'value1');
console.log(value1.is('value1'));
console.log(!TestEnum.value3.is(TestEnum.value3));
console.log(check(TestEnum.value2));
// 这段代码可以运行，但你的 TSLint 可能会报错
// 注意！不支持使用 ===
// 使用 .is() 替代
console.log(TestEnum.value1 == <any>'value1');
```

```
  }

  public is(value: Enum | string) {
    return this.value = value.toString();
  }
}

class SourceEnum extends Enum {
  public static value1 = new SourceEnum('value1');
  public static value2 = new SourceEnum('value2');
}

class TestEnum extends SourceEnum {
  public static value3 = new TestEnum('value3');
  public static value4 = new TestEnum('value4');
}

function check(test: TestEnum) {
  return test === TestEnum.value2;
}

let value1 = TestEnum.value1;

console.log(value1 + 'hello');
console.log(value1.toString() === 'value1');
console.log(value1.is('value1'));
console.log(!TestEnum.value3.is(TestEnum.value3));
console.log(check(TestEnum.value2));
// this works but perhaps your TSLint would complain
// attention! does not work with ===
// use .is() instead
console.log(TestEnum.value1 == <any>'value1');
```

第6章：函数

第6.1节：可选参数和默认参数

可选参数

在TypeScript中，函数的每个参数默认都是必需的。你可以在参数名后添加?来将其设置为可选参数。

例如，这个函数的lastName参数是可选的：

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

可选参数必须放在所有非可选参数之后：

```
function buildName(firstName?: string, lastName: string) // 无效
```

默认参数

如果用户传入`undefined`或未指定参数，则会赋予默认值。这些称为
*默认初始化*参数。

例如，"Smith" 是lastName参数的默认值。

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
buildName('foo', 'bar');      // firstName == 'foo', lastName == 'bar'
buildName('foo');             // firstName == 'foo', lastName == 'Smith'
buildName('foo', undefined);  // firstName == 'foo', lastName == 'Smith'
```

第6.2节：函数作为参数

假设我们想接收一个函数作为参数，可以这样做：

```
function foo(otherFunc: Function): void {
    ...
}
```

如果我们想接收一个构造函数作为参数：

```
function foo(constructorFunc: { new() }) {
    new constructorFunc();
}

function foo(constructorWithParamsFunc: { new(num: number) }) {
    new constructorWithParamsFunc(1);
}
```

或者为了更易读，我们可以定义一个描述构造函数的接口：

```
interface IConstructor {
    new();
```

Chapter 6: Functions

Section 6.1: Optional and Default Parameters

Optional Parameters

In TypeScript, every parameter is assumed to be required by the function. You can add a **?** at the end of a parameter name to set it as optional.

For example, the `lastName` parameter of this function is optional:

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

Optional parameters must come after all non-optional parameters:

```
function buildName(firstName?: string, lastName: string) // Invalid
```

Default Parameters

If the user passes `undefined` or doesn't specify an argument, the default value will be assigned. These are called *default-initialized* parameters.

For example, "Smith" is the default value for the `lastName` parameter.

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
buildName('foo', 'bar');      // firstName == 'foo', lastName == 'bar'
buildName('foo');             // firstName == 'foo', lastName == 'Smith'
buildName('foo', undefined);  // firstName == 'foo', lastName == 'Smith'
```

Section 6.2: Function as a parameter

Suppose we want to receive a function as a parameter, we can do it like this:

```
function foo(otherFunc: Function): void {
    ...
}
```

If we want to receive a constructor as a parameter:

```
function foo(constructorFunc: { new() }) {
    new constructorFunc();
}

function foo(constructorWithParamsFunc: { new(num: number) }) {
    new constructorWithParamsFunc(1);
}
```

Or to make it easier to read we can define an interface describing the constructor:

```
interface IConstructor {
    new();
}

function foo(contructorFunc: IConstructor) {
    new constructorFunc();
}
```

或者带参数的：

```
interface INumberConstructor {
    new(num: number);
}

function foo(contructorFunc: INumberConstructor) {
    new contructorFunc(1);
}
```

甚至带泛型的：

```
interface ITConstructor<T, U> {
    new(item: T): U;
}

function foo<T, U>(contructorFunc: ITConstructor<T, U>, item: T): U {
    return new contructorFunc(item);
}
```

如果我们想接收一个简单的函数而不是构造函数，几乎是一样的：

```
function foo(func: { (): void }) {
    func();
}

function foo(constructorWithParamsFunc: { (num: number): void }) {
    new constructorWithParamsFunc(1);
}
```

或者为了更易读，我们可以定义一个描述该函数的接口：

```
interface IFunction {
    (): void;
}

function foo(func: IFunction ) {
    func();
}
```

或者带参数的：

```
interface INumberFunction {
    (num: number): string;
}

function foo(func: INumberFunction ) {
    func(1);
}
```

甚至带泛型的：

```
}

function foo(contructorFunc: IConstructor) {
    new constructorFunc();
}
```

Or with parameters:

```
interface INumberConstructor {
    new(num: number);
}

function foo(contructorFunc: INumberConstructor) {
    new contructorFunc(1);
}
```

Even with generics:

```
interface ITConstructor<T, U> {
    new(item: T): U;
}

function foo<T, U>(contructorFunc: ITConstructor<T, U>, item: T): U {
    return new contructorFunc(item);
}
```

If we want to receive a simple function and not a constructor it's almost the same:

```
function foo(func: { (): void }) {
    func();
}

function foo(constructorWithParamsFunc: { (num: number): void }) {
    new constructorWithParamsFunc(1);
}
```

Or to make it easier to read we can define an interface describing the function:

```
interface IFunction {
    (): void;
}

function foo(func: IFunction ) {
    func();
}
```

Or with parameters:

```
interface INumberFunction {
    (num: number): string;
}

function foo(func: INumberFunction ) {
    func(1);
}
```

Even with generics:

```
interface ITFunc<T, U> {
    (item: T): U;
}

function foo<T, U>(contructorFunc: ITFunc<T, U>, item: T): U {
    return func(item);
}
```

第6.3节：带有联合类型的函数

TypeScript函数可以使用联合类型接收多种预定义类型的参数。

```
function whatTime(hour:number|string, minute:number|string):string{
    return hour+':'+minute;
}

whatTime(1,30)         //'1:30'
whatTime('1',30)       //'1:30'
whatTime(1,'30')       //'1:30'
whatTime('1','30')     //'1:30'
```

TypeScript将这些参数视为其他类型的联合类型，因此你的函数必须能够处理联合中任何类型的参数。

```
function addTen(start:number|string):number{
    if(typeof number === 'string'){
        return parseInt(number)+10;
    }else{
        else return number+10;
    }
}
```

第6.4节：函数类型

具名函数

```
function multiply(a, b) {
    return a * b;
}
```

匿名函数

```
let multiply = function(a, b) { return a * b; };
```

Lambda / 箭头函数

```
let multiply = (a, b) => { return a * b; };
```

```
interface ITFunc<T, U> {
    (item: T): U;
}

function foo<T, U>(contructorFunc: ITFunc<T, U>, item: T): U {
    return func(item);
}
```

Section 6.3: Functions with Union Types

A TypeScript function can take in parameters of multiple, predefined types using union types.

```
function whatTime(hour:number|string, minute:number|string):string{
    return hour+':'+minute;
}

whatTime(1,30)         //'1:30'
whatTime('1',30)       //'1:30'
whatTime(1,'30')       //'1:30'
whatTime('1','30')     //'1:30'
```

TypeScript treats these parameters as a single type that is a union of the other types, so your function must be able to handle parameters of any type that is in the union.

```
function addTen(start:number|string):number{
    if(typeof number === 'string'){
        return parseInt(number)+10;
    }else{
        else return number+10;
    }
}
```

Section 6.4: Types of Functions

Named functions

```
function multiply(a, b) {
    return a * b;
}
```

Anonymous functions

```
let multiply = function(a, b) { return a * b; };
```

Lambda / arrow functions

```
let multiply = (a, b) => { return a * b; };
```

第7章：类

TypeScript 与 ECMAScript 6 一样，支持使用类进行面向对象编程。这与旧版本的 JavaScript 形成对比，旧版本仅支持基于原型的继承链。

TypeScript 中的类支持类似于 Java 和 C# 等语言，类可以继承其他类，而对象则作为类的实例被创建。

与那些语言类似，TypeScript 类可以实现接口或使用泛型。

第7.1节：抽象类

```
abstract class Machine {
    constructor(public manufacturer: string) {
    }

    // 抽象类可以定义自己的方法，或者...
summary(): string {
        return `${this.manufacturer} 制造了 这台机器.`;
    }

    // 要求继承类实现方法
    abstract moreInfo(): string;
}

class Car extends Machine {
constructor(manufacturer: string, public position: number, protected speed: number) {
        super(manufacturer);
    }

move() {
        this.position += this.speed;
    }

moreInfo() {
        return `这是一辆位于 ${this.position} 并以 ${this.speed} 英里每小时行驶的汽车！`;
    }
}

let myCar = new Car("Konda", 10, 70);
myCar.move(); // 位置现在是 80
console.log(myCar.summary()); // 输出 "Konda 制造了这台机器。"
console.log(myCar.moreInfo()); // 输出 "这是一辆位于 80 并以 70 英里每小时行驶的汽车！"
```

抽象类是其他类可以继承的基类。它们本身不能被实例化（即你 cannot 使用 new Machine("Konda")）。

TypeScript 中抽象类的两个关键特征是：

1. 它们可以实现自己的方法。
2. 它们可以定义继承类 must 实现的方法。

因此，抽象类在概念上可以被视为接口和类的结合体。

第7.2节：简单类

```
class Car {
```

Chapter 7: Classes

TypeScript, like ECMAScript 6, support object-oriented programming using classes. This contrasts with older JavaScript versions, which only supported prototype-based inheritance chain.

The class support in TypeScript is similar to that of languages like Java and C#, in that classes may inherit from other classes, while objects are instantiated as class instances.

Also similar to those languages, TypeScript classes may implement interfaces or make use of generics.

Section 7.1: Abstract Classes

```
abstract class Machine {
    constructor(public manufacturer: string) {
    }

    // An abstract class can define methods of its own, or...
    summary(): string {
        return `${this.manufacturer} makes this machine.`;
    }

    // Require inheriting classes to implement methods
    abstract moreInfo(): string;
}

class Car extends Machine {
    constructor(manufacturer: string, public position: number, protected speed: number) {
        super(manufacturer);
    }

    move() {
        this.position += this.speed;
    }

    moreInfo() {
        return `This is a car located at ${this.position} and going ${this.speed}mph!`;
    }
}

let myCar = new Car("Konda", 10, 70);
myCar.move(); // position is now 80
console.log(myCar.summary()); // prints "Konda makes this machine."
console.log(myCar.moreInfo()); // prints "This is a car located at 80 and going 70mph!"
```

Abstract classes are base classes from which other classes can extend. They cannot be instantiated themselves (i.e. you **cannot** do `new Machine("Konda")`).

The two key characteristics of an abstract class in TypeScript are:

1. They can implement methods of their own.
2. They can define methods that inheriting classes **must** implement.

For this reason, abstract classes can conceptually be considered a **combination of an interface and a class**.

Section 7.2: Simple class

```
class Car {
```

```
public position: number = 0;
    private speed: number = 42;

    move() {
        this.position += this.speed;
    }
}
```

在这个例子中，我们声明了一个简单的类Car。该类有三个成员：一个私有属性speed，一个公共属性position和一个公共方法move。注意每个成员默认都是公共的。这就是为什么即使我们没有使用public关键字，move()也是公共的
。

```
var car = new Car();         // 创建一个Car实例
car.move();                  // 调用一个方法
console.log(car.position);  // 访问公共属性
```

第7.3节：基本继承

```
class Car {
public position: number = 0;
    protected speed: number = 42;

    move() {
        this.position += this.speed;
    }
}

class 自动驾驶汽车 extends 汽车 {

   移动() {
        // 开始移动 :-)
        super.移动();
        super.移动();
    }
}
```

这个示例展示了如何使用 extends 关键字创建一个非常简单的 汽车 类的子类。
自动驾驶汽车 类重写了 移动() 方法，并使用 super 调用基类的实现。

第7.4节：构造函数

在这个示例中，我们使用 构造函数 在基类中声明了一个公共属性 位置 和一个受保护属性 速度。这些属性称为 参数属性。它们允许我们在一个地方同时声明构造函数参数和成员。

TypeScript中最棒的功能之一是自动将构造函数参数赋值给相关属性。

```
class Car {
public 位置: number;
   protected 速度: number;

constructor(position: number, speed: number) {
        this.position = position;
        this.speed = speed;
    }

move() {
        this.position += this.speed;
```

```
    public position: number = 0;
    private speed: number = 42;

    move() {
        this.position += this.speed;
    }
}
```

In this example, we declare a simple class `Car`. The class has three members: a private property `speed`, a public property `position` and a public method `move`. Note that each member is public by default. That's why `move()` is public, even if we didn't use the `public` keyword.

```
var car = new Car();         // create an instance of Car
car.move();                  // call a method
console.log(car.position);   // access a public property
```

Section 7.3: Basic Inheritance

```
class Car {
    public position: number = 0;
    protected speed: number = 42;

    move() {
        this.position += this.speed;
    }
}

class SelfDrivingCar extends Car {

    move() {
        // start moving around :-)
        super.move();
        super.move();
    }
}
```

This examples shows how to create a very simple subclass of the `Car` class using the `extends` keyword. The `SelfDrivingCar` class overrides the `move()` method and uses the base class implementation using `super`.

Section 7.4: Constructors

In this example we use the `constructor` to declare a public property `position` and a protected property `speed` in the base class. These properties are called *Parameter properties*. They let us declare a constructor parameter and a member in one place.

One of the best things in TypeScript, is automatic assignment of constructor parameters to the relevant property.

```
class Car {
    public position: number;
    protected speed: number;

    constructor(position: number, speed: number) {
        this.position = position;
        this.speed = speed;
    }

    move() {
        this.position += this.speed;
    }
}
```

所有这些代码都可以用一个构造函数来简化：

```
class Car {
constructor(public position: number, protected speed: number) {}

    move() {
        this.position += this.speed;
    }
}
```

这两种写法都会被从TypeScript（设计时和编译时）转译成JavaScript，结果相同，但代码量显著减少：

```
var Car = (function () {
    function Car(position, speed) {
        this.position = position;
        this.speed = speed;
    }
Car.prototype.move = function () {
        this.position += this.speed;
    };
    return Car;
}());
```

派生类的构造函数必须调用基类构造函数，使用 super()。

```
class 自动驾驶汽车 extends 汽车 {
    constructor(启动自动驾驶: boolean) {
        super(0, 42);
        if (启动自动驾驶) {
            this.移动();
        }
    }
}

let 车 = new 自动驾驶汽车(true);
console.log(车.位置);  // 访问公共属性位置
```

第7.5节：访问器

在此示例中，我们修改了“简单类”示例以允许访问速度属性。TypeScript访问器允许我们在getter或setter中添加额外代码。

```
class Car {
public 位置: number = 0;
   private _速度: number = 42;
   private _最大速度 = 100

   移动() {
        this.位置 += this._速度;
    }

    get speed(): number {
        return this._speed;
    }
```

```
    }
}
```

All this code can be resumed in one single constructor:

```
class Car {
    constructor(public position: number, protected speed: number) {}

    move() {
        this.position += this.speed;
    }
}
```

And both of them will be transpiled from TypeScript (design time and compile time) to JavaScript with same result, but writing significantly less code:

```
var Car = (function () {
    function Car(position, speed) {
        this.position = position;
        this.speed = speed;
    }
    Car.prototype.move = function () {
        this.position += this.speed;
    };
    return Car;
}());
```

Constructors of derived classes have to call the base class constructor with `super()`.

```
class SelfDrivingCar extends Car {
    constructor(startAutoPilot: boolean) {
        super(0, 42);
        if (startAutoPilot) {
            this.move();
        }
    }
}

let car = new SelfDrivingCar(true);
console.log(car.position);  // access the public property position
```

Section 7.5: Accessors

In this example, we modify the "Simple class" example to allow access to the `speed` property. TypeScript accessors allow us to add additional code in getters or setters.

```
class Car {
    public position: number = 0;
    private _speed: number = 42;
    private _MAX_SPEED = 100

    move() {
        this.position += this._speed;
    }

    get speed(): number {
        return this._speed;
    }
    set speed(value: number) {
        this._speed = Math.min(value, this._MAX_SPEED);
    }
}

let car = new Car();
car.speed = 120;
console.log(car.speed);  // 100
```

第7.6节：转译

给定一个类SomeClass，让我们看看TypeScript如何被转译成JavaScript。

TypeScript源代码

```
class SomeClass {

public static SomeStaticValue: string = "hello";
    public someMemberValue: number = 15;
private somePrivateValue: boolean = false;

    constructor () {
SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }

public static getGoodbye(): string {
        return "goodbye!";
    }

public getFortyTwo(): number {
        return 42;
    }

private getTrue(): boolean {
        return true;
    }

}
```

JavaScript 源代码

使用 TypeScript v2.2.2 转译时，输出如下：

```
var SomeClass = (function () {
    function SomeClass() {
        this.someMemberValue = 15;
        this.somePrivateValue = false;
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }
SomeClass.getGoodbye = function () {
        return "goodbye!";
    };
SomeClass.prototype.getFortyTwo = function () {
        return 42;
    };
SomeClass.prototype.getTrue = function () {
        return true;
    };
```

```
    set speed(value: number) {
        this._speed = Math.min(value, this._MAX_SPEED);
    }
}

let car = new Car();
car.speed = 120;
console.log(car.speed);  // 100
```

Section 7.6: Transpilation

Given a class `SomeClass`, let's see how the TypeScript is transpiled into JavaScript.

TypeScript source

```
class SomeClass {

    public static SomeStaticValue: string = "hello";
    public someMemberValue: number = 15;
    private somePrivateValue: boolean = false;

    constructor () {
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }

    public static getGoodbye(): string {
        return "goodbye!";
    }

    public getFortyTwo(): number {
        return 42;
    }

    private getTrue(): boolean {
        return true;
    }

}
```

JavaScript source

When transpiled using TypeScript `v2.2.2`, the output is like so:

```
var SomeClass = (function () {
    function SomeClass() {
        this.someMemberValue = 15;
        this.somePrivateValue = false;
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }
    SomeClass.getGoodbye = function () {
        return "goodbye!";
    };
    SomeClass.prototype.getFortyTwo = function () {
        return 42;
    };
    SomeClass.prototype.getTrue = function () {
        return true;
    };
    return SomeClass;
}());
SomeClass.SomeStaticValue = "hello";
```

观察

- 类的原型修改被封装在一个IIFE中。
- 成员变量定义在主类的function内部。
- 静态属性直接添加到类对象，而实例属性则添加到原型上。

第7.7节：向现有类中猴子补丁一个函数

有时能够为类扩展新功能是很有用的。例如，假设一个字符串需要被转换成驼峰式字符串。那么我们需要告诉 TypeScript，String 包含一个名为 toCamelCase 的函数，该函数返回一个 string。

```
interface String {
    toCamelCase(): string;
}
```

现在我们可以将这个函数补丁到 String 的实现中。

```
String.prototype.toCamelCase = function() : string {
    return this.replace(/[^a-z ]/ig, '')
.replace(/(?:^\w|[A-Z]|\b\w|\s+)/g, (match: any, index: number) => {
            return +match === 0 ? "" : match[index === 0 ? 'toLowerCase' : 'toUpperCase']();
        });
}
```

如果这个对 String 的扩展被加载，就可以这样使用：

```
"This is an example".toCamelCase();    // => "thisIsAnExample"
```

```
    return SomeClass;
}());
SomeClass.SomeStaticValue = "hello";
```

Observations

- The modification of the class' prototype is wrapped inside an IIFE.
- Member variables are defined inside the main class **function**.
- Static properties are added directly to the class object, whereas instance properties are added to the prototype.

Section 7.7: Monkey patch a function into an existing class

Sometimes it's useful to be able to extend a class with new functions. For example let's suppose that a string should be converted to a camel case string. So we need to tell TypeScript, that `String` contains a function called `toCamelCase`, which returns a `string`.

```
interface String {
    toCamelCase(): string;
}
```

Now we can patch this function into the `String` implementation.

```
String.prototype.toCamelCase = function() : string {
    return this.replace(/[^a-z ]/ig, '')
        .replace(/(?:^\w|[A-Z]|\b\w|\s+)/g, (match: any, index: number) => {
            return +match === 0 ? "" : match[index === 0 ? 'toLowerCase' : 'toUpperCase']();
        });
}
```

If this extension of `String` is loaded, it's usable like this:

```
"This is an example".toCamelCase();    // => "thisIsAnExample"
```

第8章：类装饰器

参数	详情
目标	被装饰的类

第8.1节：使用类装饰器生成元数据

这次我们将声明一个类装饰器，当应用到类时，它会向类添加一些元数据：

```
function addMetadata(target: any) {

    // 添加一些元数据
target._customMetadata = {
        someKey: "someValue"
    };

    // 返回目标
    return target;

}
```

然后我们可以应用这个类装饰器：

```
@addMetadata
class Person {
private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
public greet() {
        return this._name;
    }
}

function getMetadataFromClass(target: any) {
    return target.__customMetadata;
}

console.log(getMetadataFromClass(Person));
```

装饰器是在类声明时应用的，而不是在创建类的实例时应用的。这意味着元数据在类的所有实例之间是共享的：

```
function getMetadataFromInstance(target: any) {
    return target.constructor.__customMetadata;
}

let person1 = new Person("John");
let person2 = new Person("Lisa");

console.log(getMetadataFromInstance(person1));
console.log(getMetadataFromInstance(person2));
```

第8.2节：向类装饰器传递参数

我们可以用另一个函数包装类装饰器以允许自定义：

Chapter 8: Class Decorator

Parameter	Details
target	The class being decorated

Section 8.1: Generating metadata using a class decorator

This time we are going to declare a class decorator that will add some metadata to a class when we applied to it:

```
function addMetadata(target: any) {

    // Add some metadata
    target.__customMetadata = {
        someKey: "someValue"
    };

    // Return target
    return target;

}
```

We can then apply the class decorator:

```
@addMetadata
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}

function getMetadataFromClass(target: any) {
    return target.__customMetadata;
}

console.log(getMetadataFromClass(Person));
```

The decorator is applied when the class is declared not when we create instances of the class. This means that the metadata is shared across all the instances of a class:

```
function getMetadataFromInstance(target: any) {
    return target.constructor.__customMetadata;
}

let person1 = new Person("John");
let person2 = new Person("Lisa");

console.log(getMetadataFromInstance(person1));
console.log(getMetadataFromInstance(person2));
```

Section 8.2: Passing arguments to a class decorator

We can wrap a class decorator with another function to allow customization:

```
function addMetadata(metadata: any) {
    return function log(target: any) {

        // 添加元数据
target._customMetadata = metadata;

        // 返回目标
        return target;

    }
}
```

addMetadata 接受一些用作配置的参数，然后返回一个无名函数，这个无名函数就是实际的装饰器。在装饰器中我们可以访问这些参数，因为这里存在闭包。

然后我们可以调用装饰器并传入一些配置值：

```
@addMetadata({ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" })
class Person {
private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
public greet() {
        return this._name;
    }
}
```

我们可以使用以下函数来访问生成的元数据：

```
function getMetadataFromClass(target: any) {
    return target.__customMetadata;
}

console.log(getMetadataFromInstance(Person));
```

如果一切正常，控制台应该显示：

```
{ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" }
```

第8.3节：基本类装饰器

类装饰器只是一个函数，它以类作为唯一参数，并在对其进行某些操作后返回该类：

```
function log<T>(target: T) {

    // 对 target 进行某些操作
console.log(target);

    // 返回目标
    return target;

}
```

然后我们可以将类装饰器应用到一个类上：

```
function addMetadata(metadata: any) {
    return function log(target: any) {

        // Add metadata
        target.__customMetadata = metadata;

        // Return target
        return target;

    }
}
```

The addMetadata takes some arguments used as configuration and then returns an unnamed function which is the actual decorator. In the decorator we can access the arguments because there is a closure in place.

We can then invoke the decorator passing some configuration values:

```
@addMetadata({ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" })
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}
```

We can use the following function to access the generated metadata:

```
function getMetadataFromClass(target: any) {
    return target.__customMetadata;
}

console.log(getMetadataFromInstance(Person));
```

If everything went right the console should display:

```
{ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" }
```

Section 8.3: Basic class decorator

A class decorator is just a function that takes the class as its only argument and returns it after doing something with it:

```
function log<T>(target: T) {

    // Do something with target
    console.log(target);

    // Return target
    return target;

}
```

We can then apply the class decorator to a class:

```
@log
class Person {
private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
public greet() {
        return this._name;
    }
}
```

```
@log
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}
```

第9章：接口

接口指定了任何实现该接口的类可能需要具备的字段和函数列表。
反过来，除非类拥有接口中指定的所有字段和函数，否则不能实现该接口。

使用接口的主要好处是，它允许以多态的方式使用不同类型的对象。
这是因为任何实现该接口的类至少拥有那些字段和函数。

第9.1节：扩展接口

假设我们有一个接口：

```
interface IPerson {
    name: string;
    age: number;

    breath(): void;
}
```

如果我们想创建一个更具体的接口，具有与Person相同的属性，可以使用
extends 关键字：

```
interface IManager extends IPerson {
    managerId: number;

managePeople(people: IPerson[]): void;
}
```

此外，可以扩展多个接口。

第9.2节：类接口

在接口中声明public变量和方法类型，以定义其他TypeScript代码如何与其交互。

```
interface ISampleClassInterface {
  sampleVariable: string;

sampleMethod(): void;

  optionalVariable?: string;
}
```

这里我们创建一个实现该接口的类。

```
class SampleClass implements ISampleClassInterface {
  public sampleVariable: string;
private answerToLifeTheUniverseAndEverything: number;

  constructor() {
    this.sampleVariable = 'string value';
    this.answerToLifeTheUniverseAndEverything = 42;
  }

public sampleMethod(): void {
    // 什么也不做
  }
```

Chapter 9: Interfaces

An interfaces specifies a list of fields and functions that may be expected on any class implementing the interface. Conversely, a class cannot implement an interface unless it has every field and function specified on the interface.

The primary benefit of using interfaces, is that it allows one to use objects of different types in a polymorphic way. This is because any class implementing the interface has at least those fields and functions.

Section 9.1: Extending Interface

Suppose we have an interface:

```
interface IPerson {
    name: string;
    age: number;

    breath(): void;
}
```

And we want to create more specific interface that has the same properties of the person, we can do it using the `extends` keyword:

```
interface IManager extends IPerson {
    managerId: number;

    managePeople(people: IPerson[]): void;
}
```

In addition it is possible to extend multiple interfaces.

Section 9.2: Class Interface

Declare `public` variables and methods type in the interface to define how other typescript code can interact with it.

```
interface ISampleClassInterface {
  sampleVariable: string;

  sampleMethod(): void;

  optionalVariable?: string;
}
```

Here we create a class that implements the interface.

```
class SampleClass implements ISampleClassInterface {
  public sampleVariable: string;
  private answerToLifeTheUniverseAndEverything: number;

  constructor() {
    this.sampleVariable = 'string value';
    this.answerToLifeTheUniverseAndEverything = 42;
  }

  public sampleMethod(): void {
    // do nothing
  }
private answer(q: any): number {
    return this.answerToLifeTheUniverseAndEverything;
  }
}
```

该示例展示了如何创建一个接口ISampleClassInterface和一个实现该接口的类SampleClass。

第9.3节：使用接口实现多态

使用接口实现多态的主要原因是为开发者提供未来以自己的方式实现接口方法的可能性。

假设我们有一个接口和三个类：

```
interface Connector{
    doConnect(): boolean;
}
```

这是连接器接口。现在我们将为Wifi通信实现它。

```
导出类 WifiConnector 实现 Connector{

   public doConnect(): boolean{
console.log("通过wifi连接");
        console.log("获取密码");
console.log("租用IP 24小时");
        console.log("已连接");
        返回 true
    }

}
```

这里我们开发了一个具体类，名为WifiConnector，拥有自己的实现。它现在是Connector类型。

现在我们创建了一个包含组件Connector的System。这称为依赖注入。

```
导出类 System {
构造函数(private connector: Connector){ #注入 Connector 类型
      connector.doConnect()
    }
}
```

构造函数(private connector: Connector) 这一行非常重要。 Connector 是一个接口，必须有 doConnect()。由于 Connector 是接口，System 类具有更大的灵活性。我们可以传入任何实现了 Connector 接口的类型。未来开发者将获得更多灵活性。例如，现在开发者想添加蓝牙连接模块：

```
导出类 BluetoothConnector 实现 Connector{

   public doConnect(): boolean{
console.log("通过蓝牙连接");
        console.log("使用PIN码配对");
console.log("Connected");
        返回 true
    }
```

```
  private answer(q: any): number {
    return this.answerToLifeTheUniverseAndEverything;
  }
}
```

The example shows how to create an interface `ISampleClassInterface` and a class `SampleClass` that `implements` the interface.

Section 9.3: Using Interfaces for Polymorphism

The primary reason to use interfaces to achieve polymorphism and provide developers to implement on their own way in future by implementing interface's methods.

Suppose we have an interface and three classes:

```
interface Connector{
    doConnect(): boolean;
}
```

This is connector interface. Now we will implement that for Wifi communication.

```
export class WifiConnector implements Connector{

    public doConnect(): boolean{
        console.log("Connecting via wifi");
        console.log("Get password");
        console.log("Lease an IP for 24 hours");
        console.log("Connected");
        return true
    }

}
```

Here we have developed our concrete class named `WifiConnector` that has its own implementation. This is now type `Connector`.

Now we are creating our `System` that has a component `Connector`. This is called dependency injection.

```
export class System {
    constructor(private connector: Connector){ #inject Connector type
        connector.doConnect()
    }
}
```

`constructor(private connector: Connector)` this line is very important here. `Connector` is an interface and must have `doConnect()`. As `Connector` is an interface this class `System` has much more flexibility. We can pass any Type which has implemented `Connector` interface. In future developer achieves more flexibility. For example, now developer want to add Bluetooth Connection module:

```
export class BluetoothConnector implements Connector{

    public doConnect(): boolean{
        console.log("Connecting via Bluetooth");
        console.log("Pair with PIN");
        console.log("Connected");
        return true
    }
}
```

请注意，WiFi 和蓝牙各自有自己的实现方式。它们有各自不同的连接方式。然而，由于两者都实现了 Type Connector，因此它们现在都是 Type Connector。这样我们就可以将它们中的任何一个作为构造函数参数传递给 System 类。这就是多态。类 System 现在并不知道它是蓝牙还是 WiFi，甚至我们可以通过实现 Connector 接口来添加其他通信模块，比如红外、蓝牙5 等。

这被称为 鸭子类型。 Connector 类型现在是动态的，因为 doConnect() 只是一个占位符，开发者可以根据自己的需求来实现它。

如果在 constructor(private connector: WifiConnector) 中，WifiConnector 是一个具体类，会发生什么？那么 System 类将只与 WifiConnector 紧密耦合，别无其他。这里接口通过多态解决了我们的问题。

第9.4节：泛型接口

像类一样，接口也可以接收多态参数（即泛型）。

在接口上声明泛型参数

```
interface IStatus<U> {
    code: U;
}

interface IEvents<T> {
    list: T[];
emit(event: T): void;
    getAll(): T[];
}
```

这里，你可以看到我们的两个接口接受一些泛型参数，T 和 U。

实现泛型接口

我们将创建一个简单的类来实现接口 IEvents。

```
class State<T> 实现 IEvents<T> {

    list: T[];

构造函数() {
        this.list = [];
    }

emit(event: T): void {
        this.list.push(event);
    }

getAll(): T[] {
        return this.list;
    }

}
```

让我们创建一些 State 类的实例。

在我们的示例中，State 类将通过使用 IStatus<T> 来处理泛型状态。通过这种方式，接口

```
}
```

See that Wifi and Bluetooth have its own implementation. Their own different way to connect. However, hence both have implemented Type `Connector` the are now Type `Connector`. So that we can pass any of those to `System` class as the constructor parameter. This is called polymorphism. The class `System` is now not aware of whether it is Bluetooth / Wifi even we can add another Communication module like Infrared, Bluetooth5 and whatsoever by just implementing `Connector` interface.

This is called Duck typing. `Connector` type is now dynamic as `doConnect()` is just a placeholder and developer implement this as his/her own.

if at `constructor(private connector: WifiConnector)` where `WifiConnector` is a concrete class what will happen? Then `System` class will tightly couple only with WifiConnector nothing else. Here interface solved our problem by polymorphism.

Section 9.4: Generic Interfaces

Like classes, interfaces can receive polymorphic parameters (aka Generics) too.

Declaring Generic Parameters on Interfaces

```
interface IStatus<U> {
    code: U;
}

interface IEvents<T> {
    list: T[];
    emit(event: T): void;
    getAll(): T[];
}
```

Here, you can see that our two interfaces take some generic parameters, **T** and **U**.

Implementing Generic Interfaces

We will create a simple class in order to implements the interface **IEvents**.

```
class State<T> implements IEvents<T> {

    list: T[];

    constructor() {
        this.list = [];
    }

    emit(event: T): void {
        this.list.push(event);
    }

    getAll(): T[] {
        return this.list;
    }

}
```

Let's create some instances of our **State** class.

In our example, the `State` class will handle a generic status by using `IStatus<T>`. In this way, the interface

IEvent<T> 也将处理一个 IStatus<T>。

```
const s = new State<IStatus<number>>();

// 'code' 属性预期为数字，因此：
s.emit({ code: 200 }); // 正常
s.emit({ code: '500' }); // 类型错误

s.getAll().forEach(event => console.log(event.code));
```

这里我们的 State 类被定义为 IStatus<number> 类型。

```
const s2 = new State<IStatus<Code>>();

// 我们可以以 Code 类型发出 code
s2.emit({ code: { message: 'OK', status: 200 } });

s2.getAll().map(event => event.code).forEach(event => {
     console.log(event.message);
console.log(event.status);
});
```

我们的 State 类被定义为 IStatus<Code> 类型。通过这种方式，我们能够向 emit 方法传递更复杂的类型。

如你所见，泛型接口对于静态类型代码来说是非常有用的工具。

第9.5节：向现有接口添加函数或属性

假设我们有一个对JQuery类型定义的引用，并且我们想扩展它以包含我们引入的插件中的额外函数，而该插件没有官方的类型定义。我们可以通过在一个同名的JQuery接口声明中声明插件添加的函数来轻松扩展它：

```
interface JQuery {
pluginFunctionThatDoesNothing(): void;

   // 创建可链式调用的函数
manipulateDOM(HTMLElement): JQuery;
}
```

编译器会将所有同名声明合并为一个——更多细节请参见声明合并。

第9.6节：隐式实现与对象形状

TypeScript支持接口，但编译器输出的是不支持接口的JavaScript。因此，接口在编译步骤中实际上会丢失。这就是为什么接口的类型检查依赖于对象的形状——即对象是否支持接口上的字段和函数——而不是对象是否真正实现了该接口。

```
interface IKickable {
  kick(distance: number): void;
}
class Ball {
kick(距离:数字): void {
    console.log("踢了", 距离, "米！");
  }
```

IEvent<T> will also handle a IStatus<T>.

```
const s = new State<IStatus<number>>();

// The 'code' property is expected to be a number, so:
s.emit({ code: 200 }); // works
s.emit({ code: '500' }); // type error

s.getAll().forEach(event => console.log(event.code));
```

Here our State class is typed as IStatus<number>.

```
const s2 = new State<IStatus<Code>>();

//We are able to emit code as the type Code
s2.emit({ code: { message: 'OK', status: 200 } });

s2.getAll().map(event => event.code).forEach(event => {
    console.log(event.message);
    console.log(event.status);
});
```

Our State class is typed as IStatus<Code>. In this way, we are able to pass more complex type to our emit method.

As you can see, generic interfaces can be a very useful tool for statically typed code.

Section 9.5: Add functions or properties to an existing interface

Let's suppose we have a reference to the JQuery type definition and we want to extend it to have additional functions from a plugin we included and which doesn't have an official type definition. We can easily extend it by declaring functions added by plugin in a separate interface declaration with the same JQuery name:

```
interface JQuery {
  pluginFunctionThatDoesNothing(): void;

  // create chainable function
  manipulateDOM(HTMLElement): JQuery;
}
```

The compiler will merge all declarations with the same name into one - see declaration merging for more details.

Section 9.6: Implicit Implementation And Object Shape

TypeScript supports interfaces, but the compiler outputs JavaScript, which doesn't. Therefore, interfaces are effectively lost in the compile step. This is why type checking on interfaces relies on the *shape* of the object - meaning whether the object supports the fields and functions on the interface - and not on whether the interface is actually implemented or not.

```
interface IKickable {
  kick(distance: number): void;
}
class Ball {
  kick(distance: number): void {
    console.log("Kicked", distance, "meters!");
  }
}
let kickable: IKickable = new Ball();
kickable.kick(40);
```

因此，即使Ball没有显式实现IKickable，Ball实例仍然可以被赋值为（并作为）IKickable类型来操作，即使类型被指定了。

第9.7节：使用接口来强制类型

TypeScript的核心优势之一是它强制执行你在代码中传递的值的数据类型，以帮助防止错误。

假设你正在制作一个宠物交友应用程序。

你有这样一个简单的函数，用来检查两只宠物是否相互兼容......

```
checkCompatible(petOne, petTwo) {
  if (petOne.species === petTwo.species &&
      Math.abs(petOne.age - petTwo.age) <= 5) {
    return true;
  }
}
```

这段代码完全可用，但对于其他没有编写此函数的应用程序开发者来说，很容易不知道他们应该传入带有“species”和“age”属性的对象。他们可能错误地尝试checkCompatible(petOne.species, petTwo.species)，然后在函数尝试访问petOne.species.species或petOne.species.age时，自己去解决抛出的错误！

我们可以通过指定宠物参数所需的属性来防止这种情况发生：

```
        checkCompatible(petOne: {species: string, age: number}, petTwo: {species: string, age: number}) {
    //...
}
```

在这种情况下，TypeScript 会确保传递给函数的所有参数都具有 'species' 和 'age' 属性（即使它们有额外的属性也没关系），但即使只指定了两个属性，这种方法也有些笨重。
使用接口，有更好的方法！

首先我们定义接口：

```
interface Pet {
  species: string;
  age: number;
  //如果需要，我们可以添加更多属性。
}
```

现在我们只需将参数的类型指定为我们新定义的接口，像这样......

```
checkCompatible(petOne: Pet, petTwo: Pet) {
  //...
}
```

......TypeScript 就会确保传递给函数的参数包含 Pet 接口中指定的属性！

```
}
let kickable: IKickable = new Ball();
kickable.kick(40);
```

So even if Ball doesn't explicitly implement IKickable, a Ball instance may be assigned to (and manipulated as) an IKickable, even when the type is specified.

Section 9.7: Using Interfaces to Enforce Types

One of the core benefits of TypeScript is that it enforces data types of values that you are passing around your code to help prevent mistakes.

Let's say you're making a pet dating application.

You have this simple function that checks if two pets are compatible with each other...

```
checkCompatible(petOne, petTwo) {
  if (petOne.species === petTwo.species &&
      Math.abs(petOne.age - petTwo.age) <= 5) {
    return true;
  }
}
```

This is completely functional code, but it would be far too easy for someone, especially other people working on this application who didn't write this function, to be unaware that they are supposed to pass it objects with 'species' and 'age' properties. They may mistakenly try checkCompatible(petOne.species, petTwo.species) and then be left to figure out the errors thrown when the function tries to access petOne.species.species or petOne.species.age!

One way we can prevent this from happening is to specify the properties we want on the pet parameters:

```
checkCompatible(petOne: {species: string, age: number}, petTwo: {species: string, age: number}) {
    //...
}
```

In this case, TypeScript will make sure everything passed to the function has 'species' and 'age' properties (it is okay if they have additional properties), but this is a bit of an unwieldy solution, even with only two properties specified. With interfaces, there is a better way!

First we define our interface:

```
interface Pet {
  species: string;
  age: number;
  //We can add more properties if we choose.
}
```

Now all we have to do is specify the type of our parameters as our new interface, like so...

```
checkCompatible(petOne: Pet, petTwo: Pet) {
  //...
}
```

... and TypeScript will make sure that the parameters passed to our function contain the properties specified in the Pet interface!

第10章：泛型

第10.1节：通用接口

声明通用接口

```
interface IResult<T> {
    wasSuccessful: boolean;
    error: T;
}

var result: IResult<string> = ....
var error: string = result.error;
```

具有多个类型参数的通用接口

```
interface IRunnable<T, U> {
    run(input: T): U;
}

var runnable: IRunnable<string, number> = ...
var input: string;
var result: number = runnable.run(input);
```

实现一个通用接口

```
接口 IResult<T>{
    是否成功: boolean;
    错误: T;

克隆(): IResult<T>;
}
```

用泛型类实现它：

```
类 Result<T> 实现 IResult<T> {
    构造函数(public 结果: boolean, public 错误: T) {
     }

public 克隆(): IResult<T> {
        返回 new Result<T>(this.结果, this.错误);
     }
}
```

用非泛型类实现它：

```
类 StringResult 实现 IResult<string> {
    构造函数(public 结果: boolean, public 错误: string) {
     }

public 克隆(): IResult<string> {
        return new StringResult(this.result, this.error);
     }
}
```

第10.2节：通用类

```
class Result<T> {
```

Chapter 10: Generics

Section 10.1: Generic Interfaces

Declaring a generic interface

```
interface IResult<T> {
    wasSuccessful: boolean;
    error: T;
}

var result: IResult<string> = ....
var error: string = result.error;
```

Generic interface with multiple type parameters

```
interface IRunnable<T, U> {
    run(input: T): U;
}

var runnable: IRunnable<string, number> = ...
var input: string;
var result: number = runnable.run(input);
```

Implementing a generic interface

```
interface IResult<T>{
    wasSuccessful: boolean;
    error: T;

    clone(): IResult<T>;
}
```

Implement it with generic class:

```
class Result<T> implements IResult<T> {
    constructor(public result: boolean, public error: T) {
    }

    public clone(): IResult<T> {
        return new Result<T>(this.result, this.error);
    }
}
```

Implement it with non generic class:

```
class StringResult implements IResult<string> {
    constructor(public result: boolean, public error: string) {
    }

    public clone(): IResult<string> {
        return new StringResult(this.result, this.error);
    }
}
```

Section 10.2: Generic Class

```
class Result<T> {
```

```
constructor(public wasSuccessful: boolean, public error: T) {
    }

public clone(): Result<T> {
       ...
    }
}

let r1 = new Result(false, 'error: 42');  // 编译器推断 T 为 string
let r2 = new Result(false, 42);           // 编译器推断 T 为 number
let r3 = new Result<string>(true, null);  // 明确指定 T 为 string
let r4 = new Result<string>(true, 4);     // 编译错误，因为 4 不是 string
```

第10.3节：作为约束的类型参数

在 TypeScript 1.8 中，类型参数约束可以引用同一类型参数列表中的其他类型参数。之前这是错误的。

```
 function assign<T extends U, U>(target: T, source: U): T {
    for (let id in source) {
target[id] = source[id];
    }
    return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };
assign(x, { b: 10, d: 20 });
assign(x, { e: 0 });  // 错误
```

第10.4节：泛型约束

简单约束：

```
接口 IRunnable {
    run(): void;
}

接口 IRunner<T extends IRunnable> {
    runSafe(runnable: T): void;
}
```

更复杂的约束：

```
接口 IRunnble<U> {
    run(): U;
}

接口 IRunner<T extends IRunnable<U>, U> {
    runSafe(runnable: T): U;
}
```

更复杂一些：

```
接口 IRunnble<V> {
    run(parameter: U): V;
}

interface IRunner<T extends IRunnable<U, V>, U, V> {
```

```
    constructor(public wasSuccessful: boolean, public error: T) {
    }

    public clone(): Result<T> {
       ...
    }
}

let r1 = new Result(false, 'error: 42');  // Compiler infers T to string
let r2 = new Result(false, 42);           // Compiler infers T to number
let r3 = new Result<string>(true, null);  // Explicitly set T to string
let r4 = new Result<string>(true, 4);     // Compilation error because 4 is not a string
```

Section 10.3: Type parameters as constraints

With TypeScript 1.8 it becomes possible for a type parameter constraint to reference type parameters from the same type parameter list. Previously this was an error.

```
 function assign<T extends U, U>(target: T, source: U): T {
    for (let id in source) {
        target[id] = source[id];
    }
    return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };
assign(x, { b: 10, d: 20 });
assign(x, { e: 0 });  // Error
```

Section 10.4: Generics Constraints

Simple constraint:

```
interface IRunnable {
    run(): void;
}

interface IRunner<T extends IRunnable> {
    runSafe(runnable: T): void;
}
```

More complex constraint:

```
interface IRunnble<U> {
    run(): U;
}

interface IRunner<T extends IRunnable<U>, U> {
    runSafe(runnable: T): U;
}
```

Even more complex:

```
interface IRunnble<V> {
    run(parameter: U): V;
}

interface IRunner<T extends IRunnable<U, V>, U, V> {
```

```
runSafe(runnable: T, parameter: U): V;
}
```

内联类型约束：

```
interface IRunnable<T extends { run(): void }> {
    runSafe(runnable: T): void;
}
```

第10.5节：泛型函数

在接口中：

```
interface IRunner {
runSafe<T extends IRunnable>(runnable: T): void;
}
```

在类中：

```
class Runner implements IRunner {

    public runSafe<T extends IRunnable>(runnable: T): void {
        try {
runnable.run();
        } catch(e) {
        }
    }

}
```

简单函数：

```
function runSafe<T extends IRunnable>(runnable: T): void {
    try {
runnable.run();
    } catch(e) {
    }
}
```

第10.6节：使用泛型类和函数：

创建泛型类实例：

```
var stringRunnable = new Runnable<string>();
```

运行泛型函数：

```
function runSafe<T extends Runnable<U>, U>(runnable: T);

// 指定泛型类型：
runSafe<Runnable<string>, string>(stringRunnable);

// 让 TypeScript 自行推断泛型类型：
runSafe(stringRunnable);
```

```
    runSafe(runnable: T, parameter: U): V;
}
```

Inline type constraints:

```
interface IRunnable<T extends { run(): void }> {
    runSafe(runnable: T): void;
}
```

Section 10.5: Generic Functions

In interfaces:

```
interface IRunner {
    runSafe<T extends IRunnable>(runnable: T): void;
}
```

In classes:

```
class Runner implements IRunner {

    public runSafe<T extends IRunnable>(runnable: T): void {
        try {
            runnable.run();
        } catch(e) {
        }
    }

}
```

Simple functions:

```
function runSafe<T extends IRunnable>(runnable: T): void {
    try {
        runnable.run();
    } catch(e) {
    }
}
```

Section 10.6: Using generic Classes and Functions:

Create generic class instance:

```
var stringRunnable = new Runnable<string>();
```

Run generic function:

```
function runSafe<T extends Runnable<U>, U>(runnable: T);

// Specify the generic types:
runSafe<Runnable<string>, string>(stringRunnable);

// Let typescript figure the generic types by himself:
runSafe(stringRunnable);
```

第11章：严格的空值检查

第11.1节：严格空值检查的实际应用

默认情况下，TypeScript 中所有类型都允许空值（null）：

```
function getId(x: Element) {
  return x.id;
}
getId(null);  // TypeScript 不会报错，但这是运行时错误。
```

TypeScript 2.0 增加了对严格空值检查的支持。如果你在运行 tsc 时设置了 --strictNullChecks（或者在你的 tsconfig.json 中设置了该标志），那么类型将不再允许空值（null）：

```
function getId(x: Element) {
  return x.id;
}
getId(null);  // 错误：类型为 'null' 的参数不能赋值给类型为 'Element' 的参数。
```

你必须显式允许空值（null）：

```
function getId(x: Element|null) {
  return x.id;  // 错误 TS2531：对象可能为 'null'。
}
getId(null);
```

有了适当的保护，代码类型检查通过且运行正确：

```
function getId(x: Element|null) {
  if (x) {
    return x.id;  // 在此分支中，x 的类型是 Element
  } else {
    return null;  // 在此分支中，x 的类型是 null。
  }
}
getId(null);
```

第11.2节：非空断言

非空断言操作符 ！允许你断言一个表达式不是 null 或 undefined，当 TypeScript 编译器无法自动推断时：

```
type ListNode = { data: number; next?: ListNode; };

function addNext(node: ListNode) {
    if (node.next === undefined) {
        node.next = {data: 0};
    }
}

function setNextValue(node: ListNode, value: number) {
    addNext(node);

    // 尽管我们知道 `node.next` 已定义，因为我们刚刚调用了 `addNext`，
    // 但 TypeScript 无法在下面这行代码中推断出这一点：
    // node.next.data = value;
```

Chapter 11: Strict null checks

Section 11.1: Strict null checks in action

By default, all types in TypeScript allow `null`:

```
function getId(x: Element) {
  return x.id;
}
getId(null);  // TypeScript does not complain, but this is a runtime error.
```

TypeScript 2.0 adds support for strict null checks. If you set `--strictNullChecks` when running `tsc` (or set this flag in your `tsconfig.json`), then types no longer permit `null`:

```
function getId(x: Element) {
  return x.id;
}
getId(null);  // error: Argument of type 'null' is not assignable to parameter of type 'Element'.
```

You must permit `null` values explicitly:

```
function getId(x: Element|null) {
  return x.id;  // error TS2531: Object is possibly 'null'.
}
getId(null);
```

With a proper guard, the code type checks and runs correctly:

```
function getId(x: Element|null) {
  if (x) {
    return x.id;  // In this branch, x's type is Element
  } else {
    return null;  // In this branch, x's type is null.
  }
}
getId(null);
```

Section 11.2: Non-null assertions

The non-null assertion operator, `!`, allows you to assert that an expression isn't `null` or `undefined` when the TypeScript compiler can't infer that automatically:

```
type ListNode = { data: number; next?: ListNode; };

function addNext(node: ListNode) {
    if (node.next === undefined) {
        node.next = {data: 0};
    }
}

function setNextValue(node: ListNode, value: number) {
    addNext(node);

    // Even though we know `node.next` is defined because we just called `addNext`,
    // TypeScript isn't able to infer this in the line of code below:
    // node.next.data = value;
    // 因此，我们可以使用非空断言操作符 !,
    // 来断言 node.next 不是 undefined，从而消除编译器警告
node.next!.data = value;
}
```

```
    // So, we can use the non-null assertion operator, !,
    // to assert that node.next isn't undefined and silence the compiler warning
    node.next!.data = value;
}
```

第12章：用户自定义类型保护

第12.1节：类型保护函数

你可以声明任何逻辑的函数作为类型保护。

它们的形式为：

```
function functionName(variableName: any): variableName is DesiredType {
    // 返回布尔值的函数体
}
```

如果函数返回 true，TypeScript 会在任何由该函数调用保护的代码块中将类型缩小为 DesiredType。

例如（试试）：

```
function isString(test: any): test is string {
    return typeof test === "string";
}

function example(foo: any) {
    if (isString(foo)) {
        // 在此代码块中，foo 的类型被视为字符串
        console.log("它是字符串: " + foo);
    } else {
        // 在此代码块中，foo 的类型为 any
console.log("不知道这是什么！[" + foo + "]");
    }
}

example("hello world");          // 输出 "它是字符串: hello world"
example({ something: "else" });  // 输出 "不知道这是什么！[[object Object]]"
```

类型保护函数的类型谓词（函数返回类型位置的 foo is Bar）在编译时用于缩小类型，函数体在运行时使用。类型谓词和函数必须一致，否则代码无法正常工作。

类型保护函数不必使用 typeof 或 instanceof，它们可以使用更复杂的逻辑。

例如，这段代码通过检查 jQuery 对象的版本字符串来判断你是否拥有一个 jQuery 对象。

```
function isJQuery(foo): foo 是 JQuery {
    // 测试 jQuery 的版本字符串
    return foo.jquery !== undefined;
}

function example(foo) {
    if (isJQuery(foo)) {
        // foo 在这里被类型化为 JQuery
       foo.eq(0);
    }
}
```

Chapter 12: User-defined Type Guards

Section 12.1: Type guarding functions

You can declare functions that serve as type guards using any logic you'd like.

They take the form:

```
function functionName(variableName: any): variableName is DesiredType {
    // body that returns boolean
}
```

If the function returns true, TypeScript will narrow the type to `DesiredType` in any block guarded by a call to the function.

For example (try it):

```
function isString(test: any): test is string {
    return typeof test === "string";
}

function example(foo: any) {
    if (isString(foo)) {
        // foo is type as a string in this block
        console.log("it's a string: " + foo);
    } else {
        // foo is type any in this block
        console.log("don't know what this is! [" + foo + "]");
    }
}

example("hello world");          // prints "it's a string: hello world"
example({ something: "else" });  // prints "don't know what this is! [[object Object]]"
```

A guard's function type predicate (the `foo is Bar` in the function return type position) is used at compile time to narrow types, the function body is used at runtime. The type predicate and function must agree, or your code won't work.

Type guard functions don't have to use **`typeof`** or **`instanceof`**, they can use more complicated logic.

For example, this code determines if you've got a jQuery object by checking for its version string.

```
function isJQuery(foo): foo is JQuery {
    // test for jQuery's version string
    return foo.jquery !== undefined;
}

function example(foo) {
    if (isJQuery(foo)) {
        // foo is typed JQuery here
        foo.eq(0);
    }
}
```

第12.2节：使用 instanceof

instanceof 要求变量的类型为 any。

这段代码（试试它）：

```
class 宠物 { }
class 狗 extends 宠物 {
    bark() {
console.log("汪");
    }
}
class 猫 extends 宠物 {
    purr() {
console.log("喵");
    }
}

function example(foo: any) {
    if (foo instanceof 狗) {
        // 这个代码块中 foo 的类型是 狗
        foo.bark();
    }

    if (foo instanceof 猫) {
        // 这个代码块中 foo 的类型是 猫
        foo.purr();
    }
}

example(new Dog());
example(new Cat());
```

打印

```
汪
喵
```

到控制台。

第12.3节：使用typeof

typeof 用于区分类型 数字、字符串、布尔值 和 符号。当使用其他字符串常量时不会报错，但也不会用于缩小类型范围。

与 **instanceof** 不同，**typeof** 可以用于任何类型的变量。下面的例子中，foo 可以被类型定义为 数字 | 字符串，且不会有问题。

这段代码（试试它）：

```
function example(foo: any) {
    if (typeof foo === "number") {
        // 在此代码块中，foo 的类型是数字
        console.log(foo + 100);
    }

    if (typeof foo === "string") {
```

Section 12.2: Using instanceof

instanceof requires that the variable is of type any.

This code (try it):

```
class Pet { }
class Dog extends Pet {
    bark() {
        console.log("woof");
    }
}
class Cat extends Pet {
    purr() {
        console.log("meow");
    }
}

function example(foo: any) {
    if (foo instanceof Dog) {
        // foo is type Dog in this block
        foo.bark();
    }

    if (foo instanceof Cat) {
        // foo is type Cat in this block
        foo.purr();
    }
}

example(new Dog());
example(new Cat());
```

prints

```
woof
meow
```

to the console.

Section 12.3: Using typeof

typeof is used when you need to distinguish between types number, string, boolean, and symbol. Other string constants will not error, but won't be used to narrow types either.

Unlike **instanceof**, **typeof** will work with a variable of any type. In the example below, foo could be typed as number | string without issue.

This code (try it):

```
function example(foo: any) {
    if (typeof foo === "number") {
        // foo is type number in this block
        console.log(foo + 100);
    }

    if (typeof foo === "string") {
        // foo 在此代码块中是字符串类型
console.log("不是数字: " + foo);
    }
}

example(23);
example("foo");
```

打印

```
123
不是数字: foo
```

```
        // foo is type string in this block
        console.log("not a number: " + foo);
    }
}

example(23);
example("foo");
```

prints

```
123
not a number: foo
```

第13章：TypeScript基础示例

第13.1节：使用extends和super关键字的基本类继承示例

一个通用的汽车类具有一些汽车属性和一个描述方法

```
class Car{
name:string;
engineCapacity:string;

    constructor(name:string,engineCapacity:string){
        this.name = name;
        this.engineCapacity = engineCapacity;
    }

describeCar(){
console.log(`${this.name} 车配备了 ${this.engineCapacity} 排量`);
    }
}

new Car("maruti ciaz","1500cc").describeCar();
```

HondaCar 继承了现有的通用汽车类并添加了新属性。

```
class HondaCar extends Car{
seatingCapacity:number;

constructor(name:string,engineCapacity:string,seatingCapacity:number){
        super(name,engineCapacity);
        this.seatingCapacity=seatingCapacity;
    }

describeHondaCar(){
        super.describeCar();
console.log(`这辆车的座位容量是 ${this.seatingCapacity}`);
    }
}
new 本田车("honda jazz","1200cc",4).describeHondaCar();
```

第13.2节：2 静态类变量示例 - 统计方法被调用的次数

这里 countInstance 是一个静态类变量

```
class StaticTest{
    static countInstance : number= 0;
    constructor(){
StaticTest.countInstance++;
    }
}

new StaticTest();
new StaticTest();
console.log(StaticTest.countInstance);
```

Chapter 13: TypeScript basic examples

Section 13.1: 1 basic class inheritance example using extends and super keyword

A generic Car class has some car property and a description method

```
class Car{
    name:string;
    engineCapacity:string;

    constructor(name:string,engineCapacity:string){
        this.name = name;
        this.engineCapacity = engineCapacity;
    }

    describeCar(){
        console.log(`${this.name} car comes with ${this.engineCapacity} displacement`);
    }
}

new Car("maruti ciaz","1500cc").describeCar();
```

HondaCar extends the existing generic car class and adds new property.

```
class HondaCar extends Car{
    seatingCapacity:number;

    constructor(name:string,engineCapacity:string,seatingCapacity:number){
        super(name,engineCapacity);
        this.seatingCapacity=seatingCapacity;
    }

    describeHondaCar(){
        super.describeCar();
        console.log(`this cars comes with seating capacity of ${this.seatingCapacity}`);
    }
}
new HondaCar("honda jazz","1200cc",4).describeHondaCar();
```

Section 13.2: 2 static class variable example - count how many time method is being invoked

here countInstance is a static class variable

```
class StaticTest{
    static countInstance : number= 0;
    constructor(){
        StaticTest.countInstance++;
    }
}

new StaticTest();
new StaticTest();
console.log(StaticTest.countInstance);
```

第14章：导入外部库

第14.1节：查找定义文件

针对 TypeScript 2.x：

来自DefinitelyTyped的定义可通过@types npm包获得

```
npm i --save lodash
npm i --save-dev @types/lodash
```

但如果你想使用其他仓库的类型，可以使用旧方法：

针对 TypeScript 1.x：

Typings是一个 npm 包，可以自动将类型定义文件安装到本地项目中。我建议你阅读快速入门。

```
npm install -global typings
```

现在我们可以使用 typings 命令行工具了。

1.第一步是搜索项目使用的包

```
typings 搜索 lodash
名称            来源  主页                                 描述  版本
更新日期
lodash          dt    http://lodash.com/                               2
 2016-07-20T00:13:09.000Z
lodash             global                                                           1
 2016-07-01T20:51:07.000Z
lodash          npm   https://www.npmjs.com/package/lodash                  1
 2016-07-01T20:51:07.000Z
```

2. 然后决定你应该从哪个来源安装。我使用 dt，代表DefinitelyTyped，一个 GitHub 仓库 社区可以编辑 typings，通常也是最近更新的版本。
3. 安装 typings 文件

```
typings install dt~lodash --global --save
```

让我们分解一下最后的命令。我们正在将 lodash 的 DefinitelyTyped 版本作为全局 typings 文件安装到我们的项目中，并将其保存为 typings.json 中的依赖项。现在无论我们在哪里导入 lodash，TypeScript 都会加载 lodash 的 typings 文件。

4. 如果我们想安装仅用于开发环境的 typings，可以提供 --save-dev 标志：

```
typings install chai --save-dev
```

Chapter 14: Importing external libraries

Section 14.1: Finding definition files

for typescript 2.x:

definitions from DefinitelyTyped are available via @types npm package

```
npm i --save lodash
npm i --save-dev @types/lodash
```

but in case if you want use types from other repos then can be used old way:

for typescript 1.x:

Typings is an npm package that can automatically install type definition files into a local project. I recommend that you read the quickstart.

```
npm install -global typings
```

Now we have access to the typings cli.

1. The first step is to search for the package used by the project

```
typings search lodash
NAME                SOURCE  HOMEPAGE                                        DESCRIPTION VERSIONS
UPDATED
lodash              dt      http://lodash.com/                                          2
 2016-07-20T00:13:09.000Z
lodash              global                                                              1
 2016-07-01T20:51:07.000Z
lodash              npm     https://www.npmjs.com/package/lodash                        1
 2016-07-01T20:51:07.000Z
```

2. Then decide which source you should install from. I use dt which stands for DefinitelyTyped a GitHub repo where the community can edit typings, it's also normally the most recently updated.
3. Install the typings files

```
typings install dt~lodash --global --save
```

Let's break down the last command. We are installing the DefinitelyTyped version of lodash as a global typings file in our project and saving it as a dependency in the `typings.json`. Now wherever we import lodash, typescript will load the lodash typings file.

4. If we want to install typings that will be used for development environment only, we can supply the `--save-dev` flag:

```
typings install chai --save-dev
```

第14.2节：从 npm 导入模块

如果你有该模块的类型定义文件（d.ts），你可以使用 import 语句。

```
import _ = require('lodash');
```

如果你没有该模块的定义文件，TypeScript 在编译时会报错，因为它找不到你试图导入的模块。

在这种情况下，您可以使用常规运行时的require函数导入模块。然而，这会将其作为any类型返回。

```
// _ 变量的类型是 any，因此 TypeScript 不会进行任何类型检查。
const _: any = require('lodash');
```

从 TypeScript 2.0 开始，你也可以使用简写的环境模块声明来告诉 TypeScript 某个模块存在，即使你没有该模块的类型定义文件。不过在这种情况下，TypeScript 无法提供任何有意义的类型检查。

```
declare module "lodash";

// 现在你可以用任何你喜欢的方式从 lodash 导入：
import { flatten } from "lodash";
import * as _ from "lodash";
```

从 TypeScript 2.1 开始，规则进一步放宽。只要模块存在于你的 node_modules 目录中，TypeScript 就允许你导入它，即使没有任何模块声明。（注意，如果使用了--noImplicitAny 编译选项，下面的代码仍会产生警告。）

```
// 如果 `node_modules/someModule/index.js` 存在，或者
`node_modules/someModule/package.json` 有有效的 "main" 入口点，则会生效
import { foo } from "someModule";
```

第14.3节：使用没有类型定义的全局外部库

虽然模块是理想的，但如果你使用的库是通过全局变量（如 $ 或 _）引用的，因为它是通过<script>标签加载的，你可以创建一个环境声明来引用它：

```
declare const _: any;
```

第14.4节：使用 TypeScript 2.x 查找定义文件

在 TypeScript 2.x 版本中，类型定义现在可以从 npm @types 仓库 获取。这些类型定义会被 TypeScript 编译器自动解析，使用起来更加简单。

要安装类型定义，只需将其作为开发依赖安装到项目的 package.json 中，例如：

```
npm i -S lodash
npm i -D @types/lodash
```

安装完成后，你可以像以前一样使用该模块

Section 14.2: Importing a module from npm

If you have a type definition file (d.ts) for the module, you can use an `import` statement.

```
import _ = require('lodash');
```

If you don't have a definition file for the module, TypeScript will throw an error on compilation because it cannot find the module you are trying to import.

In this case, you can import the module with the normal runtime `require` function. This returns it as the `any` type, however.

```
// The _ variable is of type any, so TypeScript will not perform any type checking.
const _: any = require('lodash');
```

As of TypeScript 2.0, you can also use a *shorthand ambient module declaration* in order to tell TypeScript that a module exists when you don't have a type definition file for the module. TypeScript won't be able to provide any meaningful typechecking in this case though.

```
declare module "lodash";

// you can now import from lodash in any way you wish:
import { flatten } from "lodash";
import * as _ from "lodash";
```

As of TypeScript 2.1, the rules have been relaxed even further. Now, as long as a module exists in your `node_modules` directory, TypeScript will allow you to import it, even with no module declaration anywhere. (Note that if using the `--noImplicitAny` compiler option, the below will still generate a warning.)

```
// Will work if `node_modules/someModule/index.js` exists, or if
`node_modules/someModule/package.json` has a valid "main" entry point
import { foo } from "someModule";
```

Section 14.3: Using global external libraries without typings

Although modules are ideal, if the library you are using is referenced by a global variable (like $ or _), because it was loaded by a `script` tag, you can create an ambient declaration in order to refer to it:

```
declare const _: any;
```

Section 14.4: Finding definition files with TypeScript 2.x

With the 2.x versions of TypeScript, typings are now available from the npm @types repository. These are automatically resolved by the TypeScript compiler and are much simpler to use.

To install a type definition you simply install it as a dev dependency in your projects package.json

e.g.

```
npm i -S lodash
npm i -D @types/lodash
```

after install you simply use the module as before

```
import * as _ from 'lodash'
```

```
import * as _ from 'lodash'
```

第15章：模块——导出与导入

第15.1节：Hello world模块

```
//hello.ts
导出函数 hello(name: string){
    console.log(`Hello ${name}!`);
}
函数 helloES(name: string){
    console.log(`Hola ${name}!`);
}
导出 {helloES};
导出 默认 hello;
```

使用目录索引加载

如果目录包含名为 `index.ts` 的文件，则可以仅使用目录名加载（对于 `index.ts` 文件名是可选的）。

```
//welcome/index.ts
导出函数 welcome(name: string){
    console.log(`Welcome ${name}!`);
}
```

定义模块的示例用法

```
import {hello, helloES} from "./hello";  // 加载指定元素
import defaultHello from "./hello";      // 将默认导出加载到名称 defaultHello
import * as Bundle from "./hello";       // 将所有导出作为 Bundle 加载
import {welcome} from "./welcome";       // 注意省略了 index.ts

hello("World");                          // 你好，世界！
helloES("Mundo");                        // 你好，世界！
defaultHello("World");                   // 你好，世界！

Bundle.hello("World");                   // 你好，世界！
Bundle.helloES("Mundo");                 // 你好，世界！

welcome("Human");                        // 欢迎，人类！
```

第15.2节：重新导出

TypeScript 允许重新导出声明。

```
//Operator.ts
interface Operator {
    eval(a: number, b: number): number;
}
export default Operator;
```

```
//Add.ts
import Operator from "./Operator";
export class Add implements Operator {
    eval(a: number, b: number): number {
        return a + b;
    }
```

Chapter 15: Modules - exporting and importing

Section 15.1: Hello world module

```
//hello.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
function helloES(name: string){
    console.log(`Hola ${name}!`);
}
export {helloES};
export default hello;
```

Load using directory index

If directory contains file named `index.ts` it can be loaded using only directory name (for `index.ts` filename is optional).

```
//welcome/index.ts
export function welcome(name: string){
    console.log(`Welcome ${name}!`);
}
```

Example usage of defined modules

```
import {hello, helloES} from "./hello";  // load specified elements
import defaultHello from "./hello";      // load default export into name defaultHello
import * as Bundle from "./hello";       // load all exports as Bundle
import {welcome} from "./welcome";       // note index.ts is omitted

hello("World");                          // Hello World!
helloES("Mundo");                        // Hola Mundo!
defaultHello("World");                   // Hello World!

Bundle.hello("World");                   // Hello World!
Bundle.helloES("Mundo");                 // Hola Mundo!

welcome("Human");                        // Welcome Human!
```

Section 15.2: Re-export

TypeScript allow to re-export declarations.

```
//Operator.ts
interface Operator {
    eval(a: number, b: number): number;
}
export default Operator;
```

```
//Add.ts
import Operator from "./Operator";
export class Add implements Operator {
    eval(a: number, b: number): number {
        return a + b;
    }
}
```

```
//Mul.ts
import Operator from "./Operator";
export class Mul implements Operator {
    eval(a: number, b: number): number {
        return a * b;
    }
}
```

你可以将所有操作打包到单个库中

```
//Operators.ts
import {Add} from "./Add";
import {Mul} from "./Mul";

export {Add, Mul};
```

命名声明可以使用更简短的语法重新导出

```
//NamedOperators.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
```

默认导出也可以被导出，但没有简短的语法可用。请记住，每个
模块只能有一个默认导出。

```
//Calculator.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
import Operator from "./Operator";

export default Operator;
```

可以重新导出打包导入

```
//RepackedCalculator.ts
export * from "./Operators";
```

当重新导出打包时，声明在显式声明时可能会被覆盖。

```
//FixedCalculator.ts
export * from "./Calculator"
import Operator from "./Calculator";
export class Add implements Operator {
    eval(a: number, b: number): number {
        return 42;
    }
}
```

使用示例

```
//run.ts
import {Add, Mul} from "./FixedCalculator";

const add = new Add();
const mul = new Mul();
```

```
}
```

```
//Mul.ts
import Operator from "./Operator";
export class Mul implements Operator {
    eval(a: number, b: number): number {
        return a * b;
    }
}
```

You can bundle all operations in single library

```
//Operators.ts
import {Add} from "./Add";
import {Mul} from "./Mul";

export {Add, Mul};
```

Named declarations can be re-exported using shorter syntax

```
//NamedOperators.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
```

Default exports can also be exported, but no short syntax is available. Remember, only one default export per module is possible.

```
//Calculator.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
import Operator from "./Operator";

export default Operator;
```

Possible is re-export of **bundled import**

```
//RepackedCalculator.ts
export * from "./Operators";
```

When re-exporting bundle, declarations may be overridden when declared explicitly.

```
//FixedCalculator.ts
export * from "./Calculator"
import Operator from "./Calculator";
export class Add implements Operator {
    eval(a: number, b: number): number {
        return 42;
    }
}
```

Usage example

```
//run.ts
import {Add, Mul} from "./FixedCalculator";

const add = new Add();
const mul = new Mul();
```

```
console.log(add.eval(1, 1)); // 42
console.log(mul.eval(3, 4)); // 12
```

第15.3节：导出/导入声明

任何声明（变量、常量、函数、类等）都可以从模块中导出，以便在其他模块中导入。

TypeScript 提供两种导出类型：命名导出和默认导出。

命名导出

```
// adams.ts
导出函数 hello(name: string){
    console.log(`Hello ${name}!`);
}
export const answerToLifeTheUniverseAndEverything = 42;
export const unused = 0;
```

导入具名导出时，可以指定要导入的元素。

```
import {hello, answerToLifeTheUniverseAndEverything} from "./adams";
hello(answerToLifeTheUniverseAndEverything);   // Hello 42!
```

默认导出

每个模块可以有一个默认导出

```
// dent.ts
const defaultValue = 54;
export default defaultValue;
```

可以使用以下方式导入

```
import dentValue from "./dent";
console.log(dentValue);        // 54
```

捆绑进口

TypeScript 提供了将整个模块导入变量的方法

```
// adams.ts
导出函数 hello(name: string){
    console.log(`Hello ${name}!`);
}
export const answerToLifeTheUniverseAndEverything = 42;
```

```
import * as Bundle from "./adams";
Bundle.hello(Bundle.answerToLifeTheUniverseAndEverything);  // Hello 42!
console.log(Bundle.unused);                                 // 0
```

```
console.log(add.eval(1, 1)); // 42
console.log(mul.eval(3, 4)); // 12
```

Section 15.3: Exporting/Importing declarations

Any declaration (variable, const, function, class, etc.) can be exported from module to be imported in other module.

TypeScript offer two export types: named and default.

Named export

```
// adams.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
export const answerToLifeTheUniverseAndEverything = 42;
export const unused = 0;
```

When importing named exports, you can specify which elements you want to import.

```
import {hello, answerToLifeTheUniverseAndEverything} from "./adams";
hello(answerToLifeTheUniverseAndEverything);   // Hello 42!
```

Default export

Each module can have one default export

```
// dent.ts
const defaultValue = 54;
export default defaultValue;
```

which can be imported using

```
import dentValue from "./dent";
console.log(dentValue);        // 54
```

Bundled import

TypeScript offers method to import whole module into variable

```
// adams.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
export const answerToLifeTheUniverseAndEverything = 42;
```

```
import * as Bundle from "./adams";
Bundle.hello(Bundle.answerToLifeTheUniverseAndEverything);  // Hello 42!
console.log(Bundle.unused);                                 // 0
```

第16章：发布 TypeScript 定义文件

第16.1节：在 npm 上包含库的定义文件

向你的 package.json 添加 typings

```
{
...
"typings": "path/file.d.ts"
...
}
```

现在每当导入该库时，TypeScript 都会加载 typings 文件

Chapter 16: Publish TypeScript definition files

Section 16.1: Include definition file with library on npm

Add typings to your package.json

```
{
...
"typings": "path/file.d.ts"
...
}
```

Now whenever that library is imported typescript will load the typings file

第17章：在webpack中使用TypeScript

第17.1节：webpack.config.js

安装加载器 npm install --save-dev ts-loader source-map-loader

tsconfig.json

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react" // 如果你想使用react jsx
  }
}
```

```
module.exports = {
    entry: "./src/index.ts",
    output: {
文件名: "./dist/bundle.js",
    },

    // 启用 sourcemaps 以调试 webpack 的输出。
devtool: "source-map",

    resolve: {
        // 添加 '.ts' 和 '.tsx' 作为可解析的扩展名。
extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

module: {
        loaders: [
                            // 所有扩展名为 '.ts' 或 '.tsx' 的文件将由 'ts-loader' 处理。
            {test: /\.tsx?$/, loader: "ts-loader"}
        ],

preLoaders: [
            // 所有输出的 '.js' 文件将由 'source-map-loader' 重新处理 sourcemaps。
            {test: /\.js$/, loader: "source-map-loader"}
        ]
    },
    /******************************
     *  如果你想使用 react *
     ****************************/


    // 当导入的模块路径匹配以下之一时，
    // 只需假设存在相应的全局变量并使用它。
    // 这很重要，因为它允许我们避免打包所有依赖，
    // 这样浏览器可以在构建之间缓存这些库。
    // externals: {
    //     "react": "React",
    //     "react-dom": "ReactDOM"
    // },
};
```

Chapter 17: Using TypeScript with webpack

Section 17.1: webpack.config.js

install loaders `npm install --save-dev ts-loader source-map-loader`

tsconfig.json

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react" // if you want to use react jsx
  }
}
```

```
module.exports = {
    entry: "./src/index.ts",
    output: {
        filename: "./dist/bundle.js",
    },

    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

    module: {
        loaders: [
            // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
            {test: /\.tsx?$/, loader: "ts-loader"}
        ],

        preLoaders: [
            // All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.
            {test: /\.js$/, loader: "source-map-loader"}
        ]
    },
    /******************************
     *  If you want to use react *
     ****************************/


    // When importing a module whose path matches one of the following, just
    // assume a corresponding global variable exists and use that instead.
    // This is important because it allows us to avoid bundling all of our
    // dependencies, which allows browsers to cache those libraries between builds.
    // externals: {
    //     "react": "React",
    //     "react-dom": "ReactDOM"
    // },
};
```

第18章：混入（Mixins）

参数	描述
derivedCtor	你想用作组合类的类
baseCtors	一个要添加到组合类的类数组

第18.1节：混入示例

要创建混入，只需声明可用作“行为”的轻量级类。

```
class Flies {
fly() {
alert('是鸟吗？是飞机吗？');
    }
}

class 爬行者 {
爬行() {
alert('我的蜘蛛感应在发抖。');
    }
}

class 防弹者 {
偏转() {
alert('我的翅膀是钢铁盾牌。');
    }
}
```

然后你可以将这些行为应用到一个组合类中：

```
class 甲壳虫人 implements 爬行者, 防弹者 {
      爬行: () => void;
偏转: () => void;
}
applyMixins (甲虫人, [攀爬,防弹]);
```

需要applyMixins函数来完成组合的工作。

```
function applyMixins(派生构造函数: any,基类构造函数数组: any[]) {
   基类构造函数数组.forEach(基类构造函数 => {
        Object.getOwnPropertyNames(基类构造函数.prototype).forEach(名称 => {
             如果 (名称 !== 'constructor') {
派生构造函数.prototype[名称] = 基类构造函数.prototype[名称];
            }
        });
    });
}
```

Chapter 18: Mixins

Parameter	Description
derivedCtor	The class that you want to use as the composition class
baseCtors	An array of classes to be added to the composition class

Section 18.1: Example of Mixins

To create mixins, simply declare lightweight classes that can be used as "behaviours".

```
class Flies {
    fly() {
        alert('Is it a bird? Is it a plane?');
    }
}

class Climbs {
    climb() {
        alert('My spider-sense is tingling.');
    }
}

class Bulletproof {
    deflect() {
        alert('My wings are a shield of steel.');
    }
}
```

You can then apply these behaviours to a composition class:

```
class BeetleGuy implements Climbs, Bulletproof {
        climb: () => void;
        deflect: () => void;
}
applyMixins (BeetleGuy, [Climbs, Bulletproof]);
```

The `applyMixins` function is needed to do the work of composition.

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            if (name !== 'constructor') {
                derivedCtor.prototype[name] = baseCtor.prototype[name];
            }
        });
    });
}
```

第19章：如何使用没有类型定义文件的JavaScript库

虽然一些现有的JavaScript库有类型定义文件，但很多没有。

TypeScript提供了几种模式来处理缺失的声明。

第19.1节：制作一个导出默认any的模块

对于更复杂的项目，或者在您打算逐步键入依赖项的情况下，创建一个模块可能会更清晰。

以使用 JQuery（尽管它确实有可用的类型定义）为例：

```
// 放在 jquery.d.ts 中
声明 let $: any;
导出 默认 $;
```

然后在项目中的任何文件中，你可以通过以下方式导入此定义：

```
// 其他某个 .ts 文件
import $ from "jquery";
```

导入后，$ 将被类型定义为 any。

如果库有多个顶级变量，则改为按名称导出和导入：

```
// 放在 jquery.d.ts 中
声明 模块 "jquery" {
    let $: any;
    let jQuery: any;

    导出 { $ };
    导出 { jQuery };
}
```

然后你可以导入并使用这两个名称：

```
// 其他某个 .ts 文件
import {$, jQuery} from "jquery";

$.doThing();
jQuery.doOtherThing();
```

第19.2节：声明一个 any 类型的全局变量

有时在简单项目中，直接声明一个 any 类型的全局变量是最简单的做法。

如果 jQuery 没有类型声明（实际上有），你可以写成

```
declare var $: any;
```

现在对 $ 的任何使用都会被视为 any 类型。

Chapter 19: How to use a JavaScript library without a type definition file

While some existing JavaScript libraries have type definition files, there are many that don't.

TypeScript offers a couple patterns to handle missing declarations.

Section 19.1: Make a module that exports a default any

For more complicated projects, or in cases where you intend to gradually type a dependency, it may be cleaner to create a module.

Using JQuery (although it does have typings available) as an example:

```
// place in jquery.d.ts
declare let $: any;
export default $;
```

And then in any file in your project, you can import this definition with:

```
// some other .ts file
import $ from "jquery";
```

After this import, $ will be typed as any.

If the library has multiple top-level variables, export and import by name instead:

```
// place in jquery.d.ts
declare module "jquery" {
    let $: any;
    let jQuery: any;

    export { $ };
    export { jQuery };
}
```

You can then import and use both names:

```
// some other .ts file
import {$, jQuery} from "jquery";

$.doThing();
jQuery.doOtherThing();
```

Section 19.2: Declare an any global

It is sometimes easiest to just declare a global of type any, especially in simple projects.

If jQuery didn't have type declarations (it does), you could put

```
declare var $: any;
```

Now any use of $ will be typed any.

第19.3节：使用环境模块

如果你只是想表示导入的意图（所以不想声明全局变量），但又不想麻烦于任何显式定义，你可以导入一个环境模块。

```
// 在声明文件中（如 declarations.d.ts）
声明模块"jquery"; // 注意这里没有定义导出内容
```

然后你可以从该环境模块中导入。

```
// 其他某个 .ts 文件
import {$, jQuery} from "jquery";
```

从声明的模块中导入的任何内容（如$和jQuery）类型都将是any

Section 19.3: Use an ambient module

If you just want to indicate the *intent* of an import (so you don't want to declare a global) but don't wish to bother with any explicit definitions, you can import an ambient module.

```
// in a declarations file (like declarations.d.ts)
declare module "jquery";    // note that there are no defined exports
```

You can then import from the ambient module.

```
// some other .ts file
import {$, jQuery} from "jquery";
```

Anything imported from the declared module (like $ and `jQuery`) above will be of type `any`

第20章：TypeScript安装typescript及运行typescript编译器tsc

如何安装TypeScript并从命令行对.ts文件运行TypeScript编译器。

第20.1节：步骤

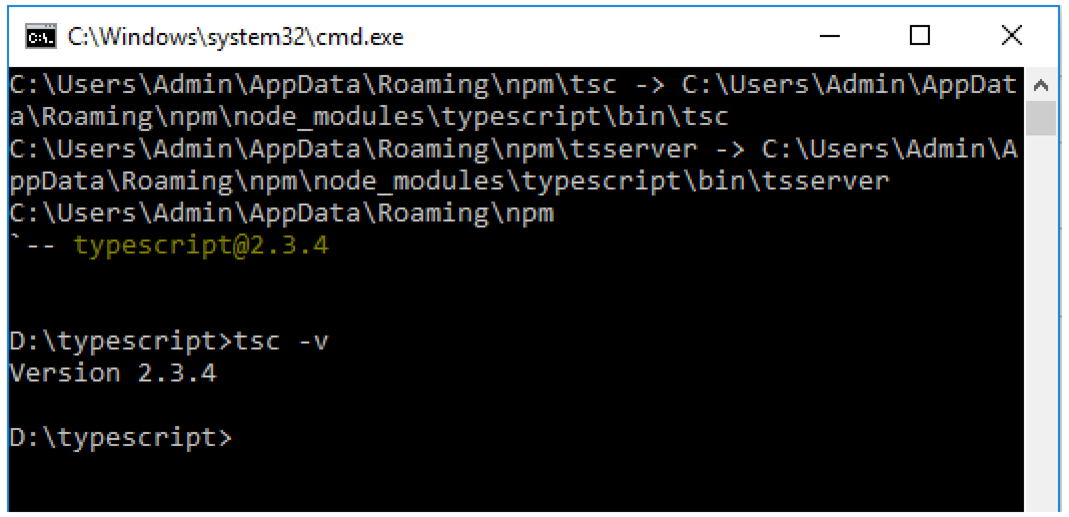
安装TypeScript并运行typescript编译器。

安装TypeScript编译器

```
npm install -g typescript
```

检查typescript版本

```
tsc -v
```



```
C:\Users\Admin\AppData\Roaming\npm\tsc -> C:\Users\Admin\AppDat
a\Roaming\npm\node_modules\typescript\bin\tsc
C:\Users\Admin\AppData\Roaming\npm\tsserver -> C:\Users\Admin\A
ppData\Roaming\npm\node_modules\typescript\bin\tsserver
C:\Users\Admin\AppData\Roaming\npm
`-- typescript@2.3.4


D:\typescript>tsc -v
Version 2.3.4

D:\typescript>
```

下载适用于Linux/Windows的Visual Studio Code

Visual Code下载链接

1. 打开Visual Studio Code
2. 打开安装了TypeScript编译器的同一文件夹
3. 点击左侧窗格的加号图标添加文件
4. 创建一个基本类。
5. 编译你的TypeScript文件并生成输出。

Chapter 20: TypeScript installing typescript and running the typescript compiler tsc

How to install TypeScript and run the TypeScript compiler against a .ts file from the command line.

Section 20.1: Steps

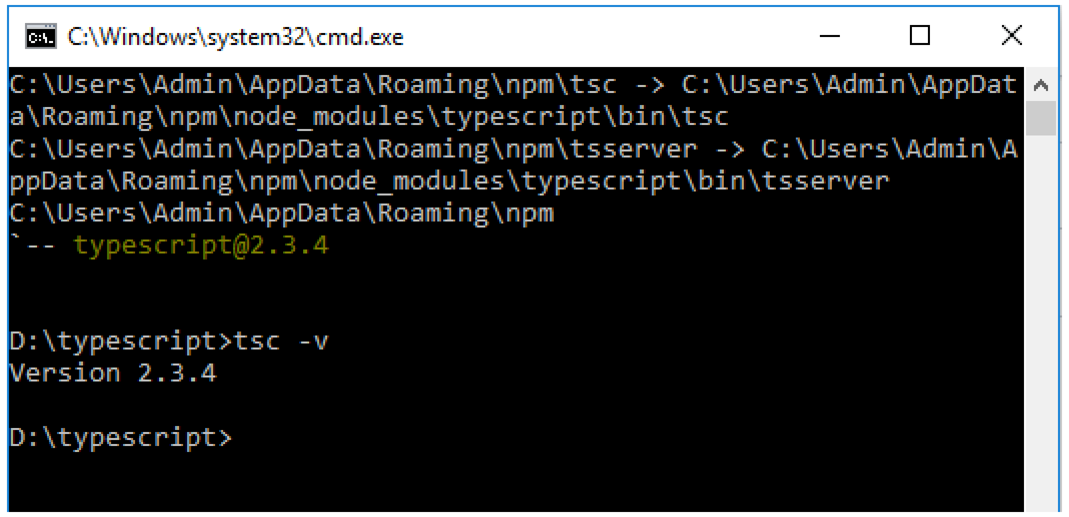
Installing TypeScript and running typescript compiler.

To install TypeScript Compiler

```
npm install -g typescript
```

To check with the typescript version

```
tsc -v
```

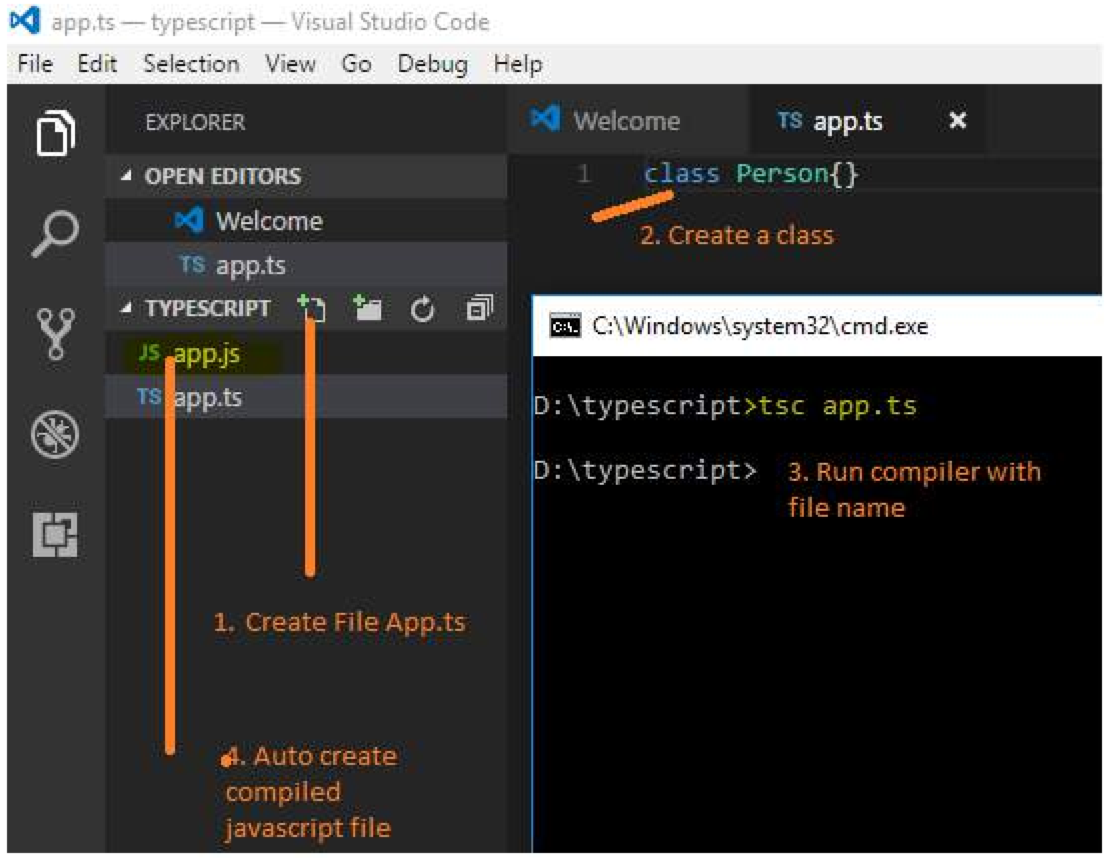


```
C:\Users\Admin\AppData\Roaming\npm\tsc -> C:\Users\Admin\AppDat
a\Roaming\npm\node_modules\typescript\bin\tsc
C:\Users\Admin\AppData\Roaming\npm\tsserver -> C:\Users\Admin\A
ppData\Roaming\npm\node_modules\typescript\bin\tsserver
C:\Users\Admin\AppData\Roaming\npm
`-- typescript@2.3.4


D:\typescript>tsc -v
Version 2.3.4

D:\typescript>
```

Download Visual Studio Code for Linux/Windows

Visual Code Download Link

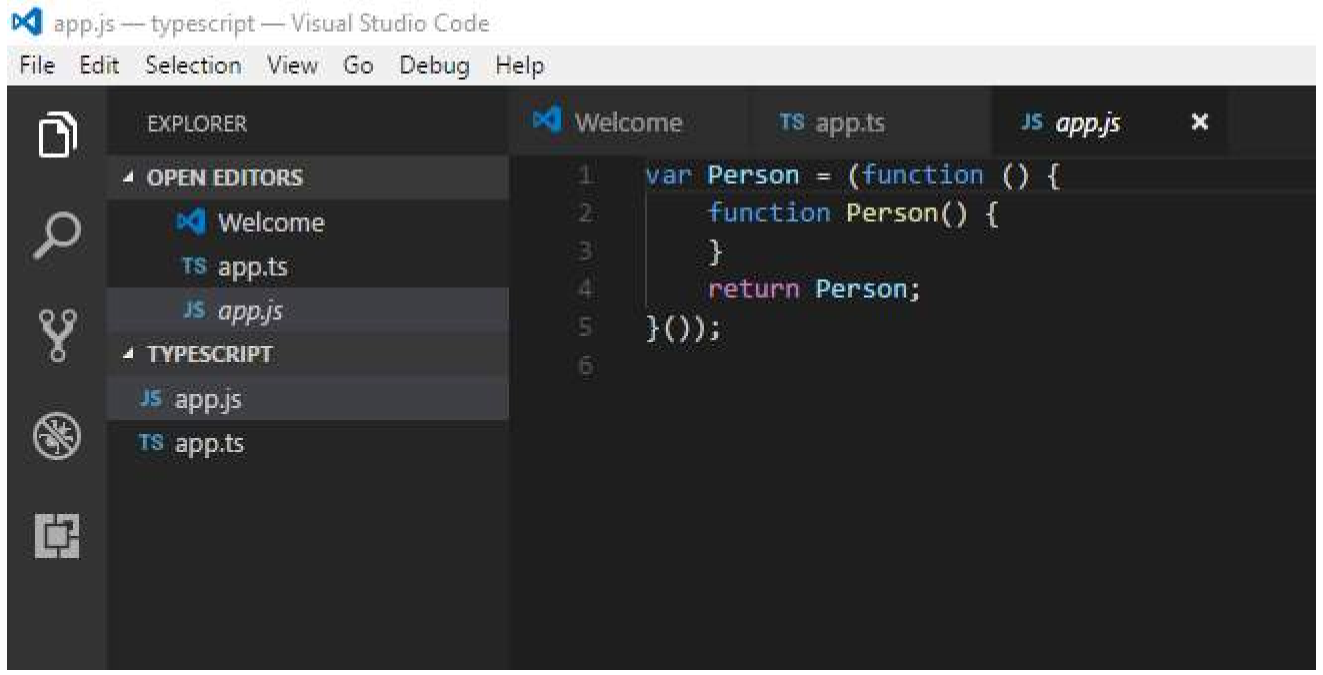
1. Open Visual Studio Code
2. Open Same Folder where you have installed TypeScript compiler
3. Add File by clicking on plus icon on left pane
4. Create a basic class.
5. Compile your type script file and generate output.


app.ts — typescript — Visual Studio Code
File Edit Selection View Go Debug Help
EXPLORER
OPEN EDITORS
Welcome
app.ts
TYPESCRIPT
app.js
app.ts
1. Create File App.ts
2. Create a class
3. Run compiler with file name
4. Auto create compiled javascript file
C:\Windows\system32\cmd.exe

```typescript
class Person{}
```

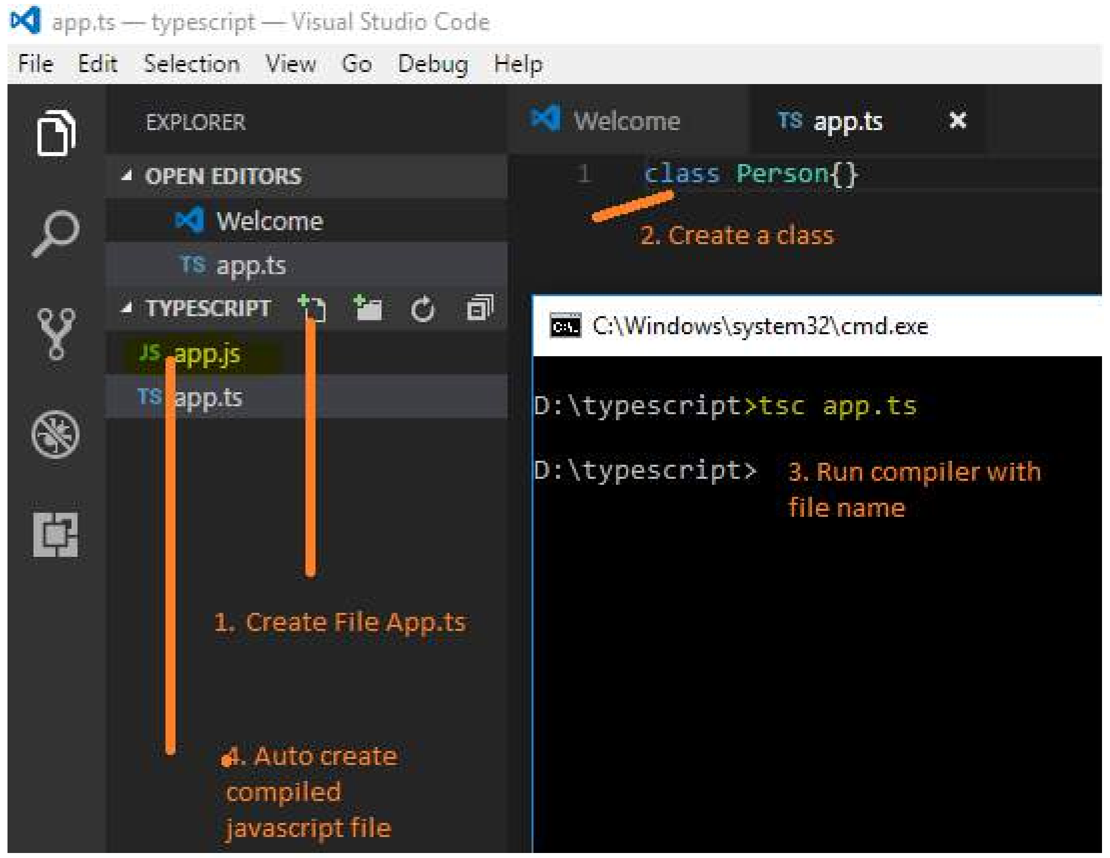
```
D:\typescript>tsc app.ts

D:\typescript>
```

查看编译后的JavaScript代码中TypeScript代码的结果。


app.js — typescript — Visual Studio Code
File Edit Selection View Go Debug Help
EXPLORER
OPEN EDITORS
Welcome
app.ts
app.js
TYPESCRIPT
app.js
app.ts

```javascript
var Person = (function () {
    function Person() {
    }
    return Person;
}());
```

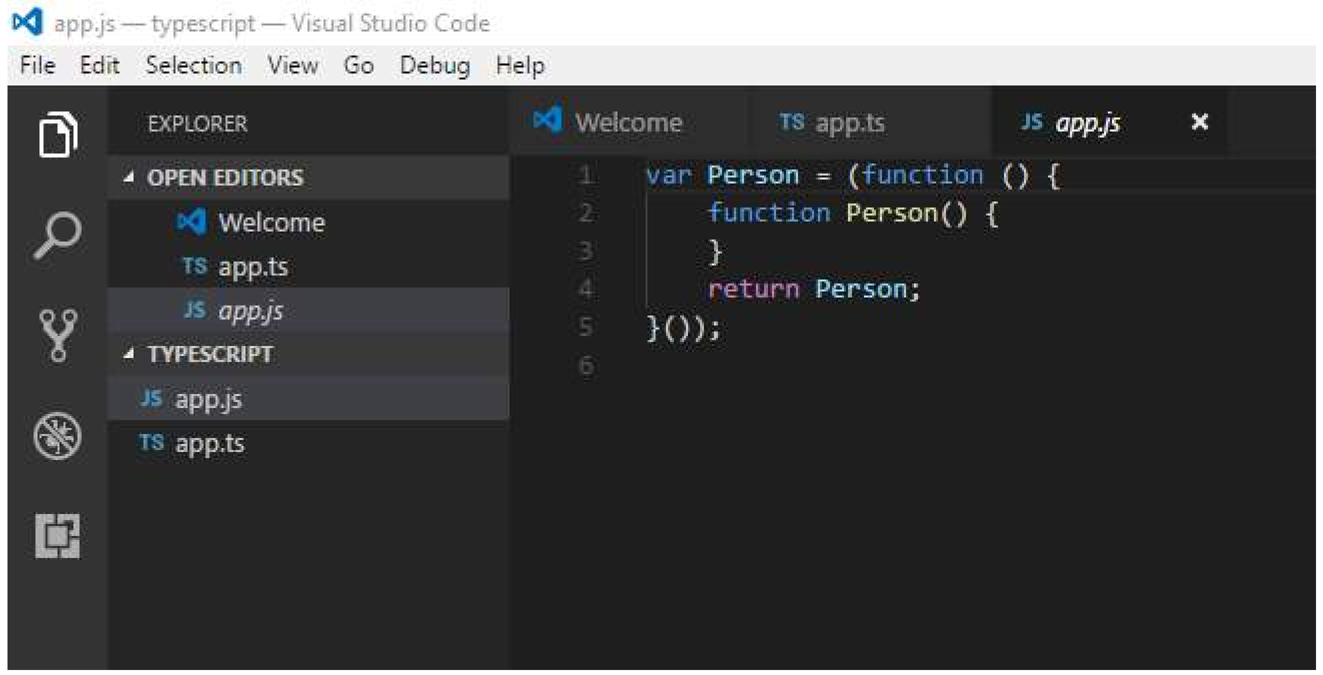
谢谢。


app.ts — typescript — Visual Studio Code
File Edit Selection View Go Debug Help
EXPLORER
OPEN EDITORS
Welcome
app.ts
TYPESCRIPT
app.js
app.ts
1. Create File App.ts
2. Create a class
3. Run compiler with file name
4. Auto create compiled javascript file
C:\Windows\system32\cmd.exe

```typescript
class Person{}
```

```
D:\typescript>tsc app.ts

D:\typescript>
```

See the result in compiled javascript of written typescript code.


app.js — typescript — Visual Studio Code
File Edit Selection View Go Debug Help
EXPLORER
OPEN EDITORS
Welcome
app.ts
app.js
TYPESCRIPT
app.js
app.ts

```javascript
var Person = (function () {
    function Person() {
    }
    return Person;
}());
```

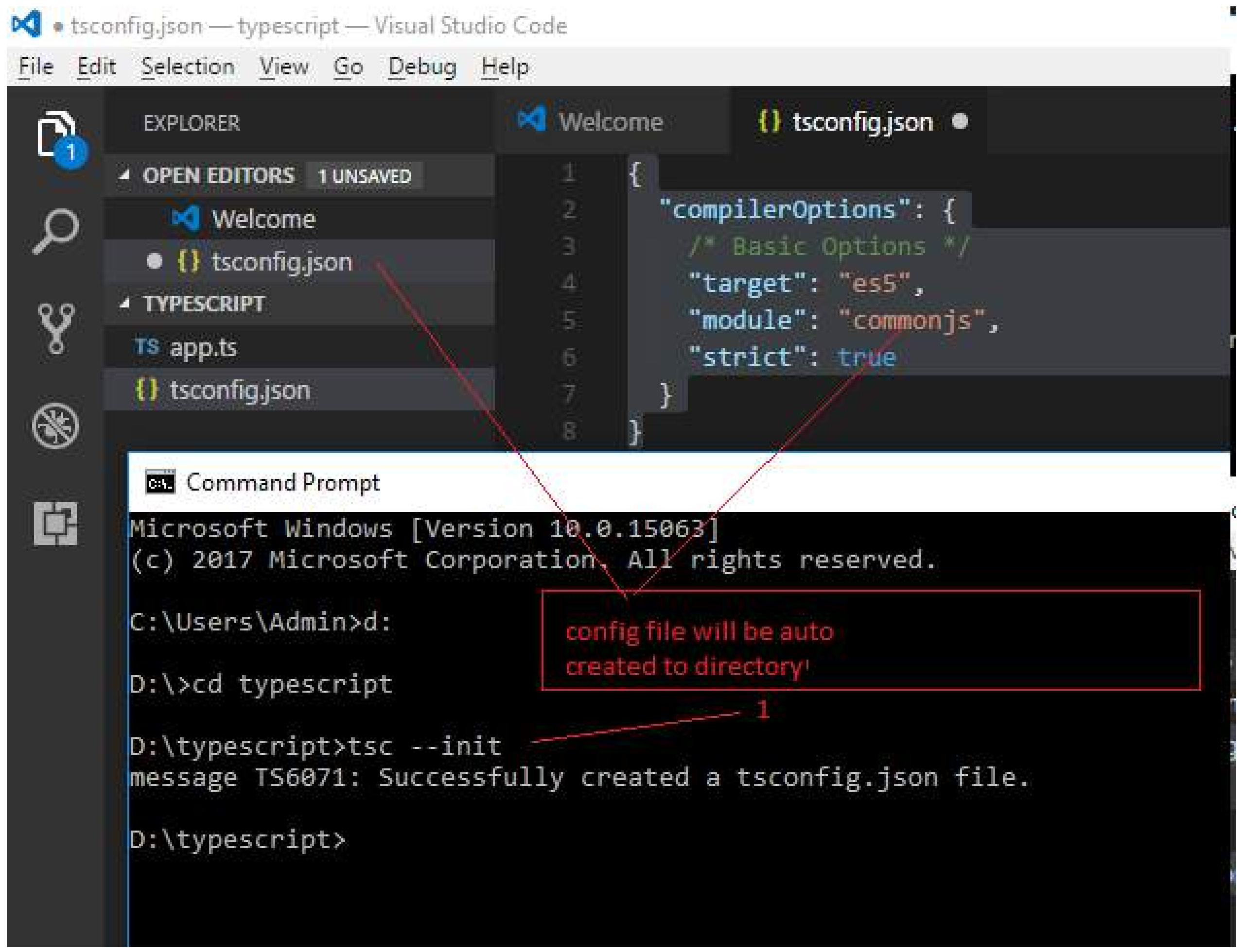
Thank you.

第21章：配置 TypeScript 项目以编译所有 TypeScript 文件。

创建你的第一个 .tsconfig 配置文件，该文件将告诉 TypeScript 编译器如何处理你的 .ts 文件

第21.1节：TypeScript 配置文件设置

- 输入命令 "tsc --init" 并按回车。
- 在此之前，我们需要使用命令 "tsc app.ts" 编译 ts 文件，现在所有内容都已在下面的配置文件中自动定义。



```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",
    "module": "commonjs",
    "strict": true
  }
}
```

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Admin>d:

D:\>cd typescript

D:\typescript>tsc --init
message TS6071: Successfully created a tsconfig.json file.

D:\typescript>
```

config file will be auto created to directory

1

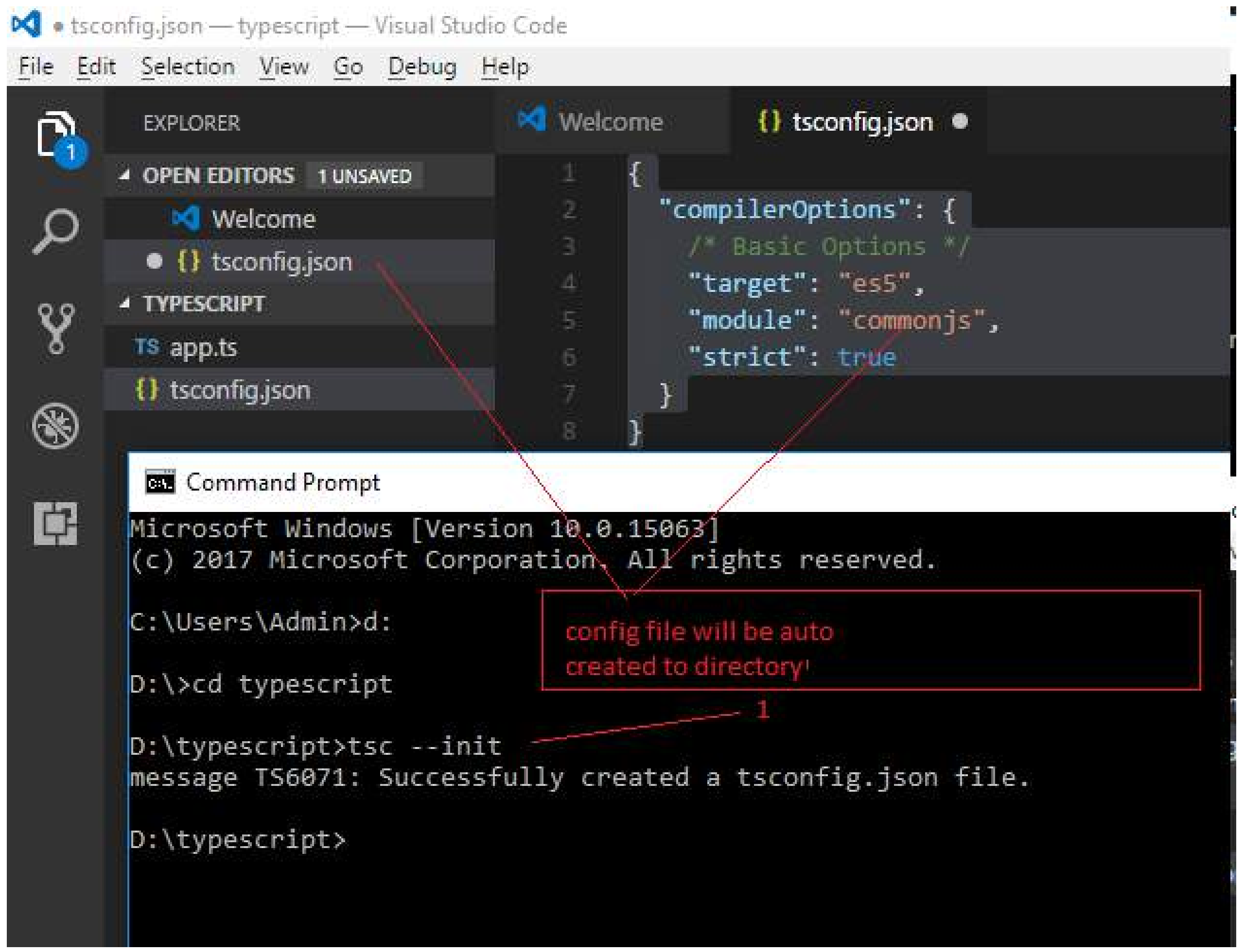
- 现在，你可以通过命令 "tsc" 编译所有 TypeScript 文件。它会自动为你的 TypeScript 文件创建 ".js" 文件。

Chapter 21: Configure typescript project to compile all files in typescript.

creating your first .tsconfig configuration file which will tell the TypeScript compiler how to treat your .ts files

Section 21.1: TypeScript Configuration file setup

- Enter command "**tsc --init**" and hit enter.
- Before that we need to compile ts file with command "**tsc app.ts**" now it is all defined in below config file automatically.



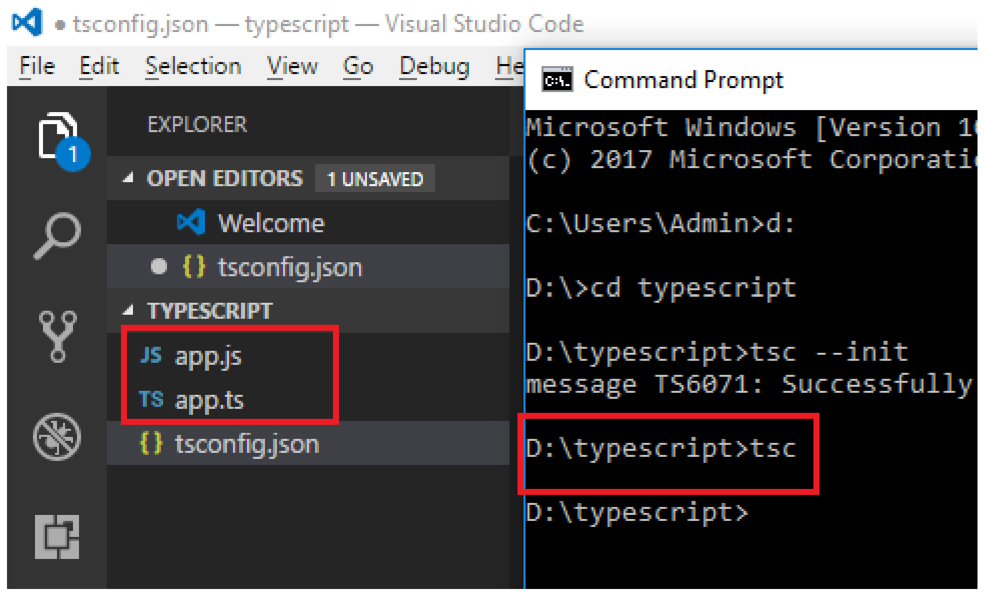
```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",
    "module": "commonjs",
    "strict": true
  }
}
```

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Admin>d:

D:\>cd typescript

D:\typescript>tsc --init
message TS6071: Successfully created a tsconfig.json file.

D:\typescript>
```

config file will be auto created to directory

1

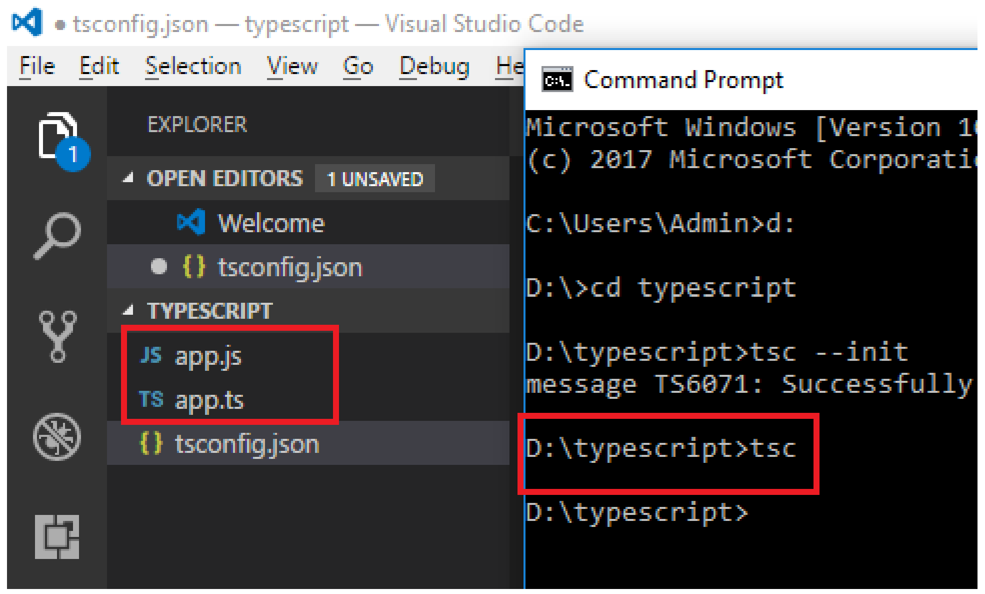
- Now, You can compile all typescripts by command "**tsc**". it will automatically create ".js" file of your typescript file.



```
Microsoft Windows [Version 1
(c) 2017 Microsoft Corporati

C:\Users\Admin>d:

D:\>cd typescript

D:\typescript>tsc --init
message TS6071: Successfully

D:\typescript>tsc

D:\typescript>
```

- 如果你创建了另一个 TypeScript 文件并在命令提示符或终端中执行 "tsc" 命令，JavaScript 文件将自动为该 TypeScript 文件创建。

谢谢，



```
Microsoft Windows [Version 1
(c) 2017 Microsoft Corporati

C:\Users\Admin>d:

D:\>cd typescript

D:\typescript>tsc --init
message TS6071: Successfully

D:\typescript>tsc

D:\typescript>
```

- If you will create another typescript and hit "tsc" command in command prompt or terminal javascript file will be automatically created for typescript file.

Thank you,

第22章：与构建工具集成

第22.1节：Browserify

安装

```
npm install tsify
```

使用命令行界面

```
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

使用API

```
var browserify = require("browserify");
var tsify = require("tsify");

browserify()
.add("main.ts")
.plugin("tsify", { noImplicitAny: true })
  .bundle()
.pipe(process.stdout);
```

更多详情：smrq/tsify

第22.2节：Webpack

安装

```
npm install ts-loader --save-dev
```

基础 webpack.config.js

webpack 2.x, 3.x

```
module.exports = {
resolve: {
extensions: ['.ts', '.tsx', '.js']
    },
module: {
        rules: [
            {
                // 为 .ts/.tsx 文件设置 ts-loader，并排除 node_modules 中的任何导入。
test: /\.tsx?$/,
                loaders: ['ts-loader'],
                exclude: /node_modules/
            }
        ]
    },
entry: [
        // 设置 index.tsx 作为应用程序入口点。
        './index.tsx'
    ],
output: {
filename: "bundle.js"
    }
};
```

webpack 1.x

```
module.exports = {
    entry: "./src/index.tsx",
    output: {
filename: "bundle.js"
```

Chapter 22: Integrating with Build Tools

Section 22.1: Browserify

Install

```
npm install tsify
```

Using Command Line Interface

```
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

Using API

```
var browserify = require("browserify");
var tsify = require("tsify");

browserify()
  .add("main.ts")
  .plugin("tsify", { noImplicitAny: true })
  .bundle()
  .pipe(process.stdout);
```

More details: smrq/tsify

Section 22.2: Webpack

Install

```
npm install ts-loader --save-dev
```

Basic webpack.config.js

webpack 2.x, 3.x

```
module.exports = {
    resolve: {
        extensions: ['.ts', '.tsx', '.js']
    },
    module: {
        rules: [
            {
                // Set up ts-loader for .ts/.tsx files and exclude any imports from node_modules.
                test: /\.tsx?$/,
                loaders: ['ts-loader'],
                exclude: /node_modules/
            }
        ]
    },
    entry: [
        // Set index.tsx as application entry point.
        './index.tsx'
    ],
    output: {
      filename: "bundle.js"
    }
};
```

webpack 1.x

```
module.exports = {
    entry: "./src/index.tsx",
    output: {
        filename: "bundle.js"
    },
resolve: {
        // 添加 '.ts' 和 '.tsx' 作为可解析的扩展名。
extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },
module: {
        loaders: [
                                // 所有扩展名为 '.ts' 或 '.tsx' 的文件将由 'ts-loader' 处理
            { test: /\.ts(x)?$/, loader: "ts-loader", exclude: /node_modules/ }
        ]
    }
}
```

有关 ts-loader 的更多详情请见此处。

替代方案：

- awesome-typescript-loader

第22.3节：Grunt

安装

```
npm install grunt-ts
```

基础 Gruntfile.js

```
module.exports = function(grunt) {
    grunt.initConfig({
ts: {
            default : {
src: ["**/*.ts", "!node_modules/**/*.ts"]
            }
        }
    });
grunt.loadNpmTasks("grunt-ts");
    grunt.registerTask("default", ["ts"]);
};
```

更多详情：TypeStrong/grunt-ts

第22.4节：Gulp

安装

```
npm install gulp-typescript
```

基础 gulpfile.js

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
    var tsResult = gulp.src("src/*.ts")
        .pipe(ts({
noImplicitAny: true,
            out: "output.js"
        }));
    return tsResult.js.pipe(gulp.dest("built/local"));
});
```

gulpfile.js 使用现有的 tsconfig.json

```
var gulp = require("gulp");
```

```
    },
    resolve: {
        // Add '.ts' and '.tsx' as a resolvable extension.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },
    module: {
        loaders: [
            // all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'
            { test: /\.ts(x)?$/, loader: "ts-loader", exclude: /node_modules/ }
        ]
    }
}
```

See more details on ts-loader here.

Alternatives:

- awesome-typescript-loader

Section 22.3: Grunt

Install

```
npm install grunt-ts
```

Basic Gruntfile.js

```
module.exports = function(grunt) {
    grunt.initConfig({
        ts: {
            default : {
                src: ["**/*.ts", "!node_modules/**/*.ts"]
            }
        }
    });
    grunt.loadNpmTasks("grunt-ts");
    grunt.registerTask("default", ["ts"]);
};
```

More details: TypeStrong/grunt-ts

Section 22.4: Gulp

Install

```
npm install gulp-typescript
```

Basic gulpfile.js

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
    var tsResult = gulp.src("src/*.ts")
        .pipe(ts({
            noImplicitAny: true,
            out: "output.js"
        }));
    return tsResult.js.pipe(gulp.dest("built/local"));
});
```

gulpfile.js using an existing tsconfig.json

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

var tsProject = ts.createProject('tsconfig.json', {
    noImplicitAny: true // You can add and overwrite parameters here
});

gulp.task("default", function () {
    var tsResult = tsProject.src()
        .pipe(tsProject());
    return tsResult.js.pipe(gulp.dest('release'));
});
```

More details: ivogabe/gulp-typescript

第22.5节：MSBuild

更新项目文件以包含本地安装的Microsoft.TypeScript.Default.props（在顶部）和 Microsoft.TypeScript.targets（在底部）文件：

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- 在底部包含默认props -->
  <Import

 Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Micr
osoft.TypeScript.Default.props"

 Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeS
cript\Microsoft.TypeScript.Default.props')" />

  <!-- TypeScript 配置写在这里 -->
  <PropertyGroup Condition="'$(Configuration)' == 'Debug'">
    <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
    <TypeScriptSourceMap>true</TypeScriptSourceMap>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)' == 'Release'">
    <TypeScriptRemoveComments >true</TypeScriptRemoveComments >
    <TypeScriptSourceMap>false</TypeScriptSourceMap>
  </PropertyGroup>

  <!-- 在底部包含默认目标 -->
  <Import

 Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Micr
osoft.TypeScript.targets"

 Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeS
cript\Microsoft.TypeScript.targets')" />
</Project>
```

关于定义 MSBuild 编译器选项的更多细节：在 MSBuild 项目中设置编译器选项

第22.6节：NuGet

- 右键点击 -> 管理 NuGet 包
- 搜索 Microsoft.TypeScript.MSBuild
- 点击 安装
- 安装完成后，重新构建！

```
var ts = require("gulp-typescript");

var tsProject = ts.createProject('tsconfig.json', {
    noImplicitAny: true // You can add and overwrite parameters here
});

gulp.task("default", function () {
    var tsResult = tsProject.src()
        .pipe(tsProject());
    return tsResult.js.pipe(gulp.dest('release'));
});
```

More details: ivogabe/gulp-typescript

Section 22.5: MSBuild

Update project file to include locally installed Microsoft.TypeScript.Default.props (at the top) and Microsoft.TypeScript.targets (at the bottom) files:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Include default props at the bottom -->
  <Import

 Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Micr
osoft.TypeScript.Default.props"

 Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeS
cript\Microsoft.TypeScript.Default.props')" />

  <!-- TypeScript configurations go here -->
  <PropertyGroup Condition="'$(Configuration)' == 'Debug'">
    <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
    <TypeScriptSourceMap>true</TypeScriptSourceMap>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)' == 'Release'">
    <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
    <TypeScriptSourceMap>false</TypeScriptSourceMap>
  </PropertyGroup>

  <!-- Include default targets at the bottom -->
  <Import

 Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Micr
osoft.TypeScript.targets"

 Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeS
cript\Microsoft.TypeScript.targets')" />
</Project>
```

More details about defining MSBuild compiler options: Setting Compiler Options in MSBuild projects

Section 22.6: NuGet

- Right-Click -> Manage NuGet Packages
- Search for Microsoft.TypeScript.MSBuild
- Hit Install
- When install is complete, rebuild!

更多详情请参见 包管理器对话框 和 使用 NuGet 的夜间构建版本

第22.7节：安装和配置 webpack + 加载器

安装

```
npm install -D webpack typescript ts-loader
```

webpack.config.js

```
module.exports = {
  entry: {
app: ['./src/'],
  },
output: {
path: __dirname,
    filename: './dist/[name].js',
  },
resolve: {
extensions: ['', '.js', '.ts'],
  },
module: {
loaders: [{
test: /\.ts(x)$/, loaders: ['ts-loader'], exclude: /node_modules/
    }],
  }
};
```

More details can be found at Package Manager Dialog and using nightly builds with NuGet

Section 22.7: Install and configure webpack + loaders

Installation

```
npm install -D webpack typescript ts-loader
```

webpack.config.js

```
module.exports = {
  entry: {
    app: ['./src/'],
  },
  output: {
    path: __dirname,
    filename: './dist/[name].js',
  },
  resolve: {
    extensions: ['', '.js', '.ts'],
  },
  module: {
    loaders: [{
      test: /\.ts(x)$/, loaders: ['ts-loader'], exclude: /node_modules/
    }],
  }
};
```

第23章：在RequireJS中使用TypeScript

RequireJS是一个JavaScript文件和模块加载器。它针对浏览器内使用进行了优化，但也可以用于Rhino和Node等其他JavaScript环境。使用像RequireJS这样的模块化脚本加载器将提升代码的速度和质量。

在RequireJS中使用TypeScript需要配置tsconfig.json，并在任何HTML文件中包含一段代码片段。编译器将把导入从TypeScript的语法转换为RequireJS的格式。

第23.1节：使用RequireJS CDN包含已编译TypeScript文件的HTML示例

```
<body onload="__init();">
    ...
    <script src="http://requirejs.org/docs/release/2.3.2/comments/require.js"></script>
    <script>
function __init() {
         require(["view/index.js"]);
     }
    </script>
</body>
```

第23.2节：使用RequireJS导入风格编译到view文件夹的tsconfig.json示例

```
{
  "module": "amd",     // 使用适用于RequireJS的AMD模块代码生成器
  "rootDir": "./src", // 将此更改为您的源文件夹
  "outDir": "./view",
   ...
}
```

Chapter 23: Using TypeScript with RequireJS

RequireJS is a JavaScript file and module loader. It is optimized for in-browser use, but it can be used in other JavaScript environments, like Rhino and Node. Using a modular script loader like RequireJS will improve the speed and quality of your code.

Using TypeScript with RequireJS requires configuration of tsconfig.json, and including an snippet in any HTML file. Compiler will traduce imports from the syntax of TypeScript to RequireJS' format.

Section 23.1: HTML example using RequireJS CDN to include an already compiled TypeScript file

```
<body onload="__init();">
    ...
    <script src="http://requirejs.org/docs/release/2.3.2/comments/require.js"></script>
    <script>
      function __init() {
        require(["view/index.js"]);
      }
    </script>
</body>
```

Section 23.2: tsconfig.json example to compile to view folder using RequireJS import style

```
{
  "module": "amd",     // Using AMD module code generator which works with RequireJS
  "rootDir": "./src", // Change this to your source folder
  "outDir": "./view",
    ...
}
```

第24章：使用AngularJS的TypeScript

名称	描述
controllerAs	是一个别名，可以将变量或函数赋值给它。@see: https://docs.angularjs.org/guide/directive
$inject	依赖注入列表，由 Angular 解析并作为参数传递给构造函数。

第24.1节：指令

```
interface IMyDirectiveController {
    // 在此指定暴露的控制器方法和属性
getUrl(): string;
}

class MyDirectiveController implements IMyDirectiveController {

    // 每个指令的内部注入
public static $inject = ["$location", "toaster"];

    constructor(private $location: ng.ILocationService, private toaster: any) {
        // $location 和 toaster 现在是控制器的属性
    }

public getUrl(): string {
        return this.$location.url(); // 使用 $location 获取 URL
    }
}

/*
* 外部注入，用于只运行一次的控制。
* 例如，我们将所有模板放在一个值中，并希望使用它。
 */
export function myDirective(templatesUrl: ITemplates): ng.IDirective {
    return {
controller: MyDirectiveController,
        controllerAs: "vm",

link: (scope: ng.IScope,
              element: ng.IAugmentedJQuery,
              attributes: ng.IAttributes,
              controller: IMyDirectiveController): void => {

            let url = controller.getUrl();
          element.text("当前网址: " + url);

        },

replace: true,
require: "ngModel",
        restrict: "A",
templateUrl: templatesUrl.myDirective,
    };
}

myDirective.$inject = [
    Templates.prototype.slug,
];
```

Chapter 24: TypeScript with AngularJS

Name	Description
controllerAs	is an alias name, to which variables or functions can be assigned to. @see: https://docs.angularjs.org/guide/directive
$inject	Dependency Injection list, it is resolved by angular and passing as an argument to constructor functions.

Section 24.1: Directive

```
interface IMyDirectiveController {
    // specify exposed controller methods and properties here
    getUrl(): string;
}

class MyDirectiveController implements IMyDirectiveController {

    // Inner injections, per each directive
    public static $inject = ["$location", "toaster"];

    constructor(private $location: ng.ILocationService, private toaster: any) {
        // $location and toaster are now properties of the controller
    }

    public getUrl(): string {
        return this.$location.url(); // utilize $location to retrieve the URL
    }
}

/*
 * Outer injections, for run once control.
 * For example we have all templates in one value, and we want to use it.
 */
export function myDirective(templatesUrl: ITemplates): ng.IDirective {
    return {
        controller: MyDirectiveController,
        controllerAs: "vm",

        link: (scope: ng.IScope,
               element: ng.IAugmentedJQuery,
               attributes: ng.IAttributes,
               controller: IMyDirectiveController): void => {

            let url = controller.getUrl();
            element.text("Current URL: " + url);

        },

        replace: true,
        require: "ngModel",
        restrict: "A",
        templateUrl: templatesUrl.myDirective,
    };
}

myDirective.$inject = [
    Templates.prototype.slug,
];
```

```
// 在项目中统一使用 slug 命名简化指令名称的更改
myDirective.prototype.slug = "myDirective";

// 你可以将此放在某个启动文件中，或者与它们放在同一文件中
angular.module("myApp").
    directive(myDirective.prototype.slug, myDirective);
```

第24.2节：简单示例

```
export function myDirective($location: ng.ILocationService): ng.IDirective {
    return {

link: (scope: ng.IScope,
            element: ng.IAugmentedJQuery,
            attributes: ng.IAttributes): void => {

            element.text("当前网址: " + $location.url());

        },

replace: true,
require: "ngModel",
        restrict: "A",
templateUrl: templatesUrl.myDirective,
    };
}

// 在项目中统一使用 slug 命名简化指令名称的更改
myDirective.prototype.slug = "myDirective";

// 你可以将此放在某个启动文件中，或者与它们放在同一文件中
angular.module("myApp").
    directive(myDirective.prototype.slug, [
        Templates.prototype.slug,
       myDirective
    ]);
```

第24.3节：组件

为了更容易过渡到Angular 2，建议使用Component，自Angular 1.5.8起可用

myModule.ts

```
import { MyModuleComponent } from "./components/myModuleComponent";
import { MyModuleService } from "./services/MyModuleService";

angular
.module("myModule", [])
.component("myModuleComponent", new MyModuleComponent())
    .service("myModuleService", MyModuleService);
```

components/myModuleComponent.ts

```
import IComponentOptions = angular.IComponentOptions;
import IControllerConstructor = angular.IControllerConstructor;
import Injectable = angular.Injectable;
import { MyModuleController } from "../controller/MyModuleController";

export class MyModuleComponent implements IComponentOptions {
```

```
// Using slug naming across the projects simplifies change of the directive name
myDirective.prototype.slug = "myDirective";

// You can place this in some bootstrap file, or have them at the same file
angular.module("myApp").
    directive(myDirective.prototype.slug, myDirective);
```

Section 24.2: Simple example

```
export function myDirective($location: ng.ILocationService): ng.IDirective {
    return {

        link: (scope: ng.IScope,
            element: ng.IAugmentedJQuery,
            attributes: ng.IAttributes): void => {

            element.text("Current URL: " + $location.url());

        },

        replace: true,
        require: "ngModel",
        restrict: "A",
        templateUrl: templatesUrl.myDirective,
    };
}

// Using slug naming across the projects simplifies change of the directive name
myDirective.prototype.slug = "myDirective";

// You can place this in some bootstrap file, or have them at the same file
angular.module("myApp").
    directive(myDirective.prototype.slug, [
        Templates.prototype.slug,
        myDirective
    ]);
```

Section 24.3: Component

For an easier transition to Angular 2, it's recommended to use `Component`, available since Angular 1.5.8

myModule.ts

```
import { MyModuleComponent } from "./components/myModuleComponent";
import { MyModuleService } from "./services/MyModuleService";

angular
    .module("myModule", [])
    .component("myModuleComponent", new MyModuleComponent())
    .service("myModuleService", MyModuleService);
```

components/myModuleComponent.ts

```
import IComponentOptions = angular.IComponentOptions;
import IControllerConstructor = angular.IControllerConstructor;
import Injectable = angular.Injectable;
import { MyModuleController } from "../controller/MyModuleController";

export class MyModuleComponent implements IComponentOptions {
```

```
public templateUrl: string = "./app/myModule/templates/myComponentTemplate.html";
    public controller: Injectable<IControllerConstructor> = MyModuleController;
public bindings: {[boundProperty: string]: string} = {};
}
```

templates/myModuleComponent.html

```
<div class="my-module-component">
   {{$ctrl.someContent}}
</div>
```

controller/MyModuleController.ts

```
import IController = angular.IController;
import { MyModuleService } from "../services/MyModuleService";

export class MyModuleController implements IController {
public static readonly $inject: string[] = ["$element", "myModuleService"];
    public someContent: string = "Hello World";

constructor($element: JQuery, private myModuleService: MyModuleService) {
        console.log("element", $element);
    }

public doSomething(): void {
        // 实现代码..
    }
}
```

services/MyModuleService.ts

```
export class MyModuleService {
    public static readonly $inject: string[] = [];

    constructor() {
    }

public doSomething(): void {
        // 执行某些操作
    }
}
```

somewhere.html

```
<my-module-component ></my-module-component >
```

```
    public templateUrl: string = "./app/myModule/templates/myComponentTemplate.html";
    public controller: Injectable<IControllerConstructor> = MyModuleController;
    public bindings: {[boundProperty: string]: string} = {};
}
```

templates/myModuleComponent.html

```
<div class="my-module-component">
    {{$ctrl.someContent}}
</div>
```

controller/MyModuleController.ts

```
import IController = angular.IController;
import { MyModuleService } from "../services/MyModuleService";

export class MyModuleController implements IController {
    public static readonly $inject: string[] = ["$element", "myModuleService"];
    public someContent: string = "Hello World";

    constructor($element: JQuery, private myModuleService: MyModuleService) {
        console.log("element", $element);
    }

    public doSomething(): void {
        // implementation..
    }
}
```

services/MyModuleService.ts

```
export class MyModuleService {
    public static readonly $inject: string[] = [];

    constructor() {
    }

    public doSomething(): void {
        // do something
    }
}
```

somewhere.html

```
<my-module-component></my-module-component>
```

第25章：使用SystemJS的TypeScript

第25.1节：使用SystemJS在浏览器中显示Hello World

安装systemjs和plugin-typescript

```
npm install systemjs
npm install plugin-typescript
```

注意：这将安装尚未发布的 TypeScript 2.0.0 编译器。

对于 TypeScript 1.8，您必须使用 plugin-typescript 4.0.16

创建 hello.ts 文件

```
导出 函数 greeter(person: String) {
    返回 'Hello, ' + person;
}
```

创建 hello.html 文件

```
<!doctype html>
<html>
<head>
    <title>TypeScript 中的 Hello World</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="config.js"></script>

    <script>
window.addEventListener('load', function() {
            System.import('./hello.ts').then(function(hello) {
                document.body.innerHTML = hello.greeter('World');
            });
        });
    </script>


</head>
<body>
</body>
</html>
```

创建 config.js - SystemJS 配置文件

```
System.config({
packages: {
        "plugin-typescript": {
            "main": "plugin.js"
        },
        "typescript": {
            "main": "lib/typescript.js",
            "meta": {
                "lib/typescript.js": {
                    "exports": "ts"
                }
            }
        }
```

Chapter 25: TypeScript with SystemJS

Section 25.1: Hello World in the browser with SystemJS

Install systemjs and plugin-typescript

```
npm install systemjs
npm install plugin-typescript
```

NOTE: this will install typescript 2.0.0 compiler which is not released yet.

For TypeScript 1.8 you have to use plugin-typescript 4.0.16

Create `hello.ts` file

```
export function greeter(person: String) {
    return 'Hello, ' + person;
}
```

Create `hello.html` file

```
<!doctype html>
<html>
<head>
    <title>Hello World in TypeScript</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="config.js"></script>

    <script>
        window.addEventListener('load', function() {
            System.import('./hello.ts').then(function(hello) {
                document.body.innerHTML = hello.greeter('World');
            });
        });
    </script>


</head>
<body>
</body>
</html>
```

Create `config.js` - SystemJS configuration file

```
System.config({
    packages: {
        "plugin-typescript": {
            "main": "plugin.js"
        },
        "typescript": {
            "main": "lib/typescript.js",
            "meta": {
                "lib/typescript.js": {
                    "exports": "ts"
                }
            }
        }
    },
map: {
        "plugin-typescript": "node_modules/plugin-typescript/lib/",
        /* 注意：这是 针对 npm 3 (node 6) */
        /* 针对 npm 2, typescript 路径将是 */
        /* node_modules/plugin-typescript/node_modules/typescript */
        "typescript": "node_modules/typescript/"
    },
transpiler: "plugin-typescript",
    meta: {
        "./hello.ts": {
format: "esm",
            loader: "plugin-typescript"
        }
    },
typescriptOptions: {
        typeCheck: 'strict'
    }
});
```

注意：如果您不想进行类型检查，请从 config.js 中移除loader: "plugin-typescript" 和 typescriptOptions。还要注意，它永远不会检查 JavaScript 代码，特别是 HTML 示例中 <script> 标签内的代码。

测试它

```
npm install live-server
./node_modules/.bin/live-server --open=hello.html
```

为生产环境构建

```
npm install systemjs-builder
```

创建 build.js 文件：

```
var Builder = require('systemjs-builder');
var builder = new Builder();
builder.loadConfig('./config.js').then(function() {
    builder.bundle('./hello.ts', './hello.js', {minify: true});
});
```

从 hello.ts 构建 hello.js

```
node build.js
```

在生产环境中使用它

只需在首次使用前通过script标签加载hello.js

hello-production.html 文件：

```
<!doctype html>
<html>
<head>
    <title>TypeScript 中的 Hello World</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="config.js"></script>
    <script src="hello.js"></script>
```

```
    },
    map: {
        "plugin-typescript": "node_modules/plugin-typescript/lib/",
        /* NOTE: this is for npm 3 (node 6) */
        /* for npm 2, typescript path will be */
        /* node_modules/plugin-typescript/node_modules/typescript */
        "typescript": "node_modules/typescript/"
    },
    transpiler: "plugin-typescript",
    meta: {
        "./hello.ts": {
            format: "esm",
            loader: "plugin-typescript"
        }
    },
    typescriptOptions: {
        typeCheck: 'strict'
    }
});
```

NOTE: if you don't want type checking, remove `loader: "plugin-typescript"` and `typescriptOptions` from `config.js`. Also note that it will never check javascript code, in particular code in the `<script>` tag in html example.

Test it

```
npm install live-server
./node_modules/.bin/live-server --open=hello.html
```

Build it for production

```
npm install systemjs-builder
```

Create `build.js` file:

```
var Builder = require('systemjs-builder');
var builder = new Builder();
builder.loadConfig('./config.js').then(function() {
    builder.bundle('./hello.ts', './hello.js', {minify: true});
});
```

build hello.js from hello.ts

```
node build.js
```

Use it in production

Just load hello.js with a script tag before first use

`hello-production.html` file:

```
<!doctype html>
<html>
<head>
    <title>Hello World in TypeScript</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="config.js"></script>
    <script src="hello.js"></script>
    <script>
window.addEventListener('load', function() {
          System.import('./hello.ts').then(function(hello) {
              document.body.innerHTML = hello.greeter('World');
          });
        });
    </script>


</head>
<body>
</body>
</html>
```

```
    <script>
        window.addEventListener('load', function() {
            System.import('./hello.ts').then(function(hello) {
                document.body.innerHTML = hello.greeter('World');
            });
        });
    </script>


</head>
<body>
</body>
</html>
```

第26章：在React中使用TypeScript（JS和原生）

第26.1节：用TypeScript编写的ReactJS组件

你可以轻松地在TypeScript中使用ReactJS的组件。只需将文件扩展名从'jsx'改为'tsx'：

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

但是为了充分利用 TypeScript 的主要特性（静态类型检查），你必须做几件事：

1) 将 React.createClass 转换为 ES6 类：

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

有关转换为 ES6 的更多信息，请查看 here

2) 添加 Props 和 State 接口：

```
interface Props {
    name:string;
    optionalParam?:number;
}

interface State {
  //empty in our case
}

class HelloMessage extends React.Component<Props, State> {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
// TypeScript will allow you to create without the optional parameter
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
// But it does check if you pass in an optional parameter of the wrong type
ReactDOM.render(<HelloMessage name="Sebastian" optionalParam='foo' />, mountNode);
```

现在如果程序员忘记传递props，或者尝试传递接口中未定义的props，TypeScript会显示错误。

Chapter 26: Using TypeScript with React (JS & native)

Section 26.1: ReactJS component written in TypeScript

You can use ReactJS's components easily in TypeScript. Just rename the 'jsx' file extension to 'tsx':

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

But in order to make full use of TypeScript's main feature (static type checking) you must do a couple things:

1) convert React.createClass to an ES6 Class:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

For more info on converting to ES6 look here

2) Add Props and State interfaces:

```
interface Props {
    name:string;
    optionalParam?:number;
}

interface State {
  //empty in our case
}

class HelloMessage extends React.Component<Props, State> {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
// TypeScript will allow you to create without the optional parameter
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
// But it does check if you pass in an optional parameter of the wrong type
ReactDOM.render(<HelloMessage name="Sebastian" optionalParam='foo' />, mountNode);
```

Now TypeScript will display an error if the programmer forgets to pass props. Or if trying to pass in props that are not defined in the interface.

第26.2节：TypeScript、React和webpack

全局安装typescript、typings和webpack

```
npm install -g typescript typings webpack
```

安装加载器并链接 TypeScript

```
npm install --save-dev ts-loader source-map-loader npm link typescript
```

> 链接 TypeScript 允许 ts-loader 使用你全局安装的 TypeScript，而不需要单独的本地副本 typescript 文档

使用 typescript 2.x 安装 .d.ts 文件

```
npm i @types/react --save-dev
npm i @types/react-dom --save-dev
```

使用 typescript 1.x 安装 .d.ts 文件

```
typings install --global --save dt~react
typings install --global --save dt~react-dom
```

tsconfig.json 配置文件

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react"
  }
}
```

webpack.config.js 配置文件

```
module.exports = {
entry: "<入口点路径>",// 例如 ./src/helloMessage.tsx
    output: {
filename: "<path to bundle file>", // 例如 ./dist/bundle.js
    },

    // 启用 sourcemaps 以调试 webpack 的输出。
devtool: "source-map",

    resolve: {
        // 添加 '.ts' 和 '.tsx' 作为可解析的扩展名。
extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

module: {
        loaders: [
                                // 所有扩展名为 '.ts' 或 '.tsx' 的文件将由 'ts-loader' 处理。
            {test: /\.tsx?$/, loader: "ts-loader"}
        ],
```

Section 26.2: TypeScript & react & webpack

Installing typescript, typings and webpack globally

```
npm install -g typescript typings webpack
```

Installing loaders and linking typescript

```
npm install --save-dev ts-loader source-map-loader npm link typescript
```

> Linking TypeScript allows ts-loader to use your global installation of TypeScript instead of needing a separate local copy typescript doc

installing `.d.ts` files with typescript 2.x

```
npm i @types/react --save-dev
npm i @types/react-dom --save-dev
```

installing `.d.ts` files with typescript 1.x

```
typings install --global --save dt~react
typings install --global --save dt~react-dom
```

`tsconfig.json` configuration file

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react"
  }
}
```

`webpack.config.js` configuration file

```
module.exports = {
    entry: "<path to entry point>",// for example ./src/helloMessage.tsx
    output: {
        filename: "<path to bundle file>", // for example ./dist/bundle.js
    },

    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

    module: {
        loaders: [
            // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
            {test: /\.tsx?$/, loader: "ts-loader"}
        ],
preLoaders: [
            // 所有输出的 '.js' 文件将由 'source-map-loader' 重新处理 sourcemaps。
            {test: /\.js$/, loader: "source-map-loader"}
        ]
    },

    // 当导入的模块路径匹配以下之一时，
    // 只需假设存在相应的全局变量并使用它。
    // 这很重要，因为它允许我们避免打包所有依赖，
    // 这样浏览器可以在构建之间缓存这些库。
externals: {
        "react": "React",
        "react-dom": "ReactDOM"
    },
};
```

最后运行 `webpack` 或 `webpack -w`（用于监听模式）

注意：React 和 ReactDOM 被标记为外部依赖

```
        preLoaders: [
            // All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.
            {test: /\.js$/, loader: "source-map-loader"}
        ]
    },

    // When importing a module whose path matches one of the following, just
    // assume a corresponding global variable exists and use that instead.
    // This is important because it allows us to avoid bundling all of our
    // dependencies, which allows browsers to cache those libraries between builds.
    externals: {
        "react": "React",
        "react-dom": "ReactDOM"
    },
};
```

finally run `webpack` or `webpack -w` (for watch mode)

Note: React and ReactDOM are marked as external

第27章：TSLint - 确保代码质量和一致性

TSLint 对代码进行静态分析，检测代码中的错误和潜在问题。

第27.1节：配置以减少编程错误

此 tslint.json 示例包含一组配置，用于强制更多类型检查，捕获常见错误或其他容易导致错误的混淆结构，并更好地遵循 TypeScript 贡献者编码指南。

要强制执行这些规则，请将 tslint 包含在构建流程中，并在使用 tsc 编译代码之前检查代码。

```
{
  "rules": {
      // TypeScript 特定
      "member-access": true, // 需要对类成员进行显式的可见性声明。
      "no-any": true, // 禁止将 any 用作类型声明。
      // 功能性
      "label-position": true, // 只允许标签出现在合理的位置。
      "no-bitwise": true, // 禁止使用按位运算符。
      "no-eval": true, // 禁止调用 eval 函数。
      "no-null-keyword": true, // 禁止使用 null 关键字字面量。
      "no-unsafe-finally": true, // 禁止在 finally 块中使用控制流语句，如 return、continue、
break 和 throw。
      "no-var-keyword": true, // 禁止使用 var 关键字。
      "radix": true, // 调用 parseInt 时必须指定基数参数。
      "triple-equals": true, // 要求使用 === 和 !== 代替 == 和 !=。
      "use-isnan": true, // 强制使用 isNaN() 函数检查 NaN 引用，而不是与 NaN 常量进行比较。

      // 风格
      "class-name": true, // 强制使用 PascalCase 命名类和接口名称。
      "interface-name": [ true, "never-prefix" ], // 要求接口名称以大写字母开头，且不以'I'前缀开头

      "no-angle-bracket-type-assertion": true, // 要求使用 as Type 进行类型断言，而不是 <Type>。

      "one-variable-per-declaration": true, // 不允许在同一声明语句中定义多个变量。

      "quotemark": [ true, "double", "avoid-escape" ], // 要求字符串字面量使用双引号。
      "semicolon": [ true, "always" ], // 强制每条语句末尾使用分号。

      "variable-name": [true, "ban-keywords", "check-format", "allow-leading-underscore"] // 检查变量名的各种错误。不允许使用某些 TypeScript 关键字（any、Number、number、String、string、Boolean、boolean、undefined）作为变量或参数名。只允许使用驼峰式命名或全大写命名。允许变量名开头使用下划线（仅在指定了"check-format"时生效）。


  }
}
```

第27.2节：安装与设置

要安装 tslint，请运行命令

```
npm install -g tslint
```

Tslint 通过文件 tslint.json 进行配置。要初始化默认配置，请运行命令

Chapter 27: TSLint - assuring code quality and consistency

TSLint performs static analysis of code and detect errors and potential problems in code.

Section 27.1: Configuration for fewer programming errors

This tslint.json example contains a set of configuration to enforce more typings, catch common errors or otherwise confusing constructs that are prone to producing bugs and following more the Coding Guidelines for TypeScript Contributors.

To enforce this rules, include tslint in your build process and check your code before compiling it with tsc.

```
{
  "rules": {
      // TypeScript Specific
      "member-access": true, // Requires explicit visibility declarations for class members.
      "no-any": true, // Disallows usages of any as a type declaration.
      // Functionality
      "label-position": true, // Only allows labels in sensible locations.
      "no-bitwise": true, // Disallows bitwise operators.
      "no-eval": true, // Disallows eval function invocations.
      "no-null-keyword": true, // Disallows use of the null keyword literal.
      "no-unsafe-finally": true, // Disallows control flow statements, such as return, continue,
break and throws in finally blocks.
      "no-var-keyword": true, // Disallows usage of the var keyword.
      "radix": true, // Requires the radix parameter to be specified when calling parseInt.
      "triple-equals": true, // Requires === and !== in place of == and !=.
      "use-isnan": true, // Enforces use of the isNaN() function to check for NaN references instead
of a comparison to the NaN constant.
      // Style
      "class-name": true, // Enforces PascalCased class and interface names.
      "interface-name": [ true, "never-prefix" ], // Requires interface names to begin with a capital
'I'
      "no-angle-bracket-type-assertion": true, // Requires the use of as Type for type assertions
instead of <Type>.
      "one-variable-per-declaration": true, // Disallows multiple variable definitions in the same
declaration statement.
      "quotemark": [ true, "double", "avoid-escape" ], // Requires double quotes for string literals.
      "semicolon": [ true, "always" ], // Enforces consistent semicolon usage at the end of every
statement.
      "variable-name": [true, "ban-keywords", "check-format", "allow-leading-underscore"] // Checks
variable names for various errors. Disallows the use of certain TypeScript keywords (any, Number,
number, String, string, Boolean, boolean, undefined) as variable or parameter. Allows only camelCased
or UPPER_CASED variable names. Allows underscores at the beginning (only has an effect if "check-
format" specified).
  }
}
```

Section 27.2: Installation and setup

To install tslint run command

```
npm install -g tslint
```

Tslint is configured via file `tslint.json`. To initialize default configuration run command

```
tslint --init
```

要检查文件中的可能错误，请运行命令

```
tslint filename.ts
```

第27.3节：TSLint规则集

- tslint-microsoft-contrib
- tslint-eslint-rules
- codelyzer

Yeoman 生成器支持所有这些预设，并且也可以扩展：

- generator-tslint

第27.4节：基本的 tslint.json 配置

这是一个基本的 tslint.json 配置，

- 禁止使用 any
- 要求 `if`/`else`/`for`/`do`/`while` 语句必须使用大括号
- 要求字符串使用双引号（"）

```
{
    "rules": {
        "no-any": true,
        "curly": true,
        "quotemark": [true, "double"]
    }
}
```

第27.5节：使用预定义规则集作为默认设置

tslint 可以扩展现有的规则集，并随默认的 tslint:recommended 和 tslint:latest 一起发布。

> tslint:recommended 是一套稳定且带有一定主观性的规则集，我们鼓励在一般的TypeScript 编程中使用。该配置遵循语义化版本控制（semver），因此在次要版本或补丁版本更新中不会有破坏性变更。
>
> tslint:latest 继承自 tslint:recommended，并持续更新以包含每个 TSLint 版本中最新规则的配置。使用此配置可能会在次要版本更新中引入破坏性变更，
> 因为启用了新规则，可能导致代码中出现 lint 失败。当 TSLint 进行主版本升级时，tslint:recommended 将更新为与 tslint:latest 完全一致。

预定义规则集的文档 和 源代码

因此可以简单地使用：

```
{
  "extends": "tslint:recommended"
}
```

```
tslint --init
```

To check file for possible errors in file run command

```
tslint filename.ts
```

Section 27.3: Sets of TSLint Rules

- tslint-microsoft-contrib
- tslint-eslint-rules
- codelyzer

Yeoman generator supports all these presets and can be extends also:

- generator-tslint

Section 27.4: Basic tslint.json setup

This is a basic `tslint.json` setup which

- prevents use of `any`
- requires curly braces for `if`/`else`/`for`/`do`/`while` statements
- requires double quotes (`"`) to be used for strings

```
{
    "rules": {
        "no-any": true,
        "curly": true,
        "quotemark": [true, "double"]
    }
}
```

Section 27.5: Using a predefined ruleset as default

`tslint` can extend an existing rule set and is shipped with the defaults `tslint:recommended` and `tslint:latest`.

> `tslint:recommended` is a stable, somewhat opinionated set of rules which we encourage for general TypeScript programming. This configuration follows semver, so it will not have breaking changes across minor or patch releases.
>
> `tslint:latest` extends tslint:recommended and is continuously updated to include configuration for the latest rules in every TSLint release. Using this config may introduce breaking changes across minor releases as new rules are enabled which cause lint failures in your code. When TSLint reaches a major version bump, tslint:recommended will be updated to be identical to tslint:latest.

Docs and source code of predefined ruleset

So one can simply use:

```
{
  "extends": "tslint:recommended"
}
```

来获得一个合理的初始配置。

然后可以通过 rules 覆盖该预设中的规则，例如对于 Node 开发者来说，将 no-console 设置为 false 是合理的：

```
{
  "extends": "tslint:recommended",
  "rules": {
    "no-console": false
  }
}
```

to have a sensible starting configuration.

One can then overwrite rules from that preset via `rules`, e.g. for node developers it made sense to set `no-console` to **`false`**:

```
{
  "extends": "tslint:recommended",
  "rules": {
    "no-console": false
  }
}
```

第28章：tsconfig.json

第28.1节：使用tsconfig.json创建TypeScript项目

存在 tsconfig.json 文件表示当前目录是一个启用TypeScript的项目根目录。

初始化TypeScript项目，或者更准确地说，初始化tsconfig.json文件，可以通过以下命令完成：

```
tsc --init
```

从TypeScript v2.3.0及更高版本开始，默认会创建以下tsconfig.json：

```
{
  "compilerOptions": {
    /* 基本选项 */
    "target": "es5",                          /* 指定ECMAScript目标版本：'ES3'（默认）、
'ES5'、'ES2015'、'ES2016'、'ES2017'或'ESNEXT'。 */
    "module": "commonjs",                     /* 指定模块代码生成方式：'commonjs'、'amd'、
'system'、'umd'或'es2015'。 */
    // "lib": [],                             /* 指定要包含在编译中的库文件： */

    // "allowJs": true,                       /* 允许编译JavaScript文件。 */
    // "checkJs": true,                       /* 报告.js文件中的错误。 */
    // "jsx": "preserve",                     /* 指定 JSX 代码生成方式：'preserve'、'react-native' 或 'react'。 */

    // "declaration": true,                   /* 生成相应的 '.d.ts' 文件。 */
    // "sourceMap": true,                     /* 生成相应的 '.map' 文件。 */
    // "outFile": "./",                       /* 连接并输出到单个文件。 */
    // "outDir": "./",                        /* 将输出结构重定向到该目录。 */
    // "rootDir": "./",                       /* 指定输入文件的根目录。用于
通过 --outDir 控制输出目录结构。 */
    // "removeComments": true,                /* 不输出注释。 */
    // "noEmit": true,                        /* 不输出文件。 */
    // "importHelpers": true,                 /* 从 'tslib' 导入辅助函数。 */
    // "downlevelIteration": true,            /* 在目标为 'ES5' 或 'ES3' 时，提供对 'for-of'、
扩展运算符和解构赋值中可迭代对象的完整支持。 */
    // "isolatedModules": true,               /* 将每个文件作为单独模块转译（类似于
'ts.transpileModule'）。 */

    /* 严格类型检查选项 */
    "strict": true                            /* 启用所有严格类型检查选项。 *//// "noImplicitAny": true,
          /* 对隐式 'any' 类型的表达式和声明报错。 */

    // "strictNullChecks": true,              /* 启用严格的空值检查。 *//// "noImplicitThis": tr
ue,                /* 对隐式 'any' 类型的 'this' 表达式报错。 */

    // "alwaysStrict": true,                  /* 以严格模式解析并为每个源文件输出 "use strict"。 */


    /* 额外检查 */
    // "noUnusedLocals": true,                /* 对未使用的局部变量报错。 *//// "noUnusedParame
ters": true,            /* 对未使用的参数报错。 *//// "noImplicitReturns": true,            /* 当函
数的所有代码路径未返回值时报错。 */

    // "noFallthroughCasesInSwitch": true,    /* 对 switch 语句中贯穿情况报错。 */


    /* 模块解析选项 */
    // "moduleResolution": "node",            /* 指定模块解析策略：'node'（Node.js）
```

Chapter 28: tsconfig.json

Section 28.1: Create TypeScript project with tsconfig.json

The presence of a **tsconfig.json** file indicates that the current directory is the root of a TypeScript enabled project.

Initializing a TypeScript project, or better put tsconfig.json file, can be done through the following command:

```
tsc --init
```

As of TypeScript v2.3.0 and higher this will create the following tsconfig.json by default:

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",                          /* Specify ECMAScript target version: 'ES3' (default),
'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'. */
    "module": "commonjs",                     /* Specify module code generation: 'commonjs', 'amd',
'system', 'umd' or 'es2015'. */
    // "lib": [],                             /* Specify library files to be included in the
compilation:  */
    // "allowJs": true,                       /* Allow javascript files to be compiled. */
    // "checkJs": true,                       /* Report errors in .js files. */
    // "jsx": "preserve",                     /* Specify JSX code generation: 'preserve', 'react-
native', or 'react'. */
    // "declaration": true,                   /* Generates corresponding '.d.ts' file. */
    // "sourceMap": true,                     /* Generates corresponding '.map' file. */
    // "outFile": "./",                       /* Concatenate and emit output to single file. */
    // "outDir": "./",                        /* Redirect output structure to the directory. */
    // "rootDir": "./",                       /* Specify the root directory of input files. Use to
control the output directory structure with --outDir. */
    // "removeComments": true,                /* Do not emit comments to output. */
    // "noEmit": true,                        /* Do not emit outputs. */
    // "importHelpers": true,                 /* Import emit helpers from 'tslib'. */
    // "downlevelIteration": true,            /* Provide full support for iterables in 'for-of',
spread, and destructuring when targeting 'ES5' or 'ES3'. */
    // "isolatedModules": true,               /* Transpile each file as a separate module (similar to
'ts.transpileModule'). */

    /* Strict Type-Checking Options */
    "strict": true                            /* Enable all strict type-checking options. */
    // "noImplicitAny": true,                 /* Raise error on expressions and declarations with an
implied 'any' type. */
    // "strictNullChecks": true,              /* Enable strict null checks. */
    // "noImplicitThis": true,                /* Raise error on 'this' expressions with an implied
'any' type. */
    // "alwaysStrict": true,                  /* Parse in strict mode and emit "use strict" for each
source file. */

    /* Additional Checks */
    // "noUnusedLocals": true,                /* Report errors on unused locals. */
    // "noUnusedParameters": true,            /* Report errors on unused parameters. */
    // "noImplicitReturns": true,             /* Report error when not all code paths in function
return a value. */
    // "noFallthroughCasesInSwitch": true,    /* Report errors for fallthrough cases in switch
statement. */

    /* Module Resolution Options */
    // "moduleResolution": "node",            /* Specify module resolution strategy: 'node' (Node.js)
或“经典”（TypeScript 1.6 之前的版本）。*/
    // "baseUrl": "./",                        /* 解析非绝对模块名称的基础目录。*/
*/
    // "paths": {},                            /* 一系列条目，将导入重新映射到相对于 'baseUrl' 的查找位置。 */

    // "rootDirs": [],                         /* 根文件夹列表，其组合内容
表示运行时项目的结构。 */
    // "typeRoots": [],                        /* 包含类型定义的文件夹列表。 */
    // "types": [],                            /* 要包含的类型声明文件 */
编译。*/
    // "allowSyntheticDefaultImports": true,  /* 允许从没有默认导出的模块进行默认导入。
这不会影响代码生成，仅影响类型检查。 */

    /* 源映射选项 */
    // "sourceRoot": "./",                     /* 指定调试器应定位TypeScript文件的位置，而不是源代码位置。 */

    // "mapRoot": "./",                        /* 指定调试器应定位映射文件的位置，而不是生成的位置。 */

    // "inlineSourceMap": true,                /* 生成带有源映射的单个文件，而不是生成单独的文件。 */

    // "inlineSources": true,                  /* 在单个文件中与源映射一起生成源代码；需要设置 '--inlineSourceMap' 或 '
--sourceMap'。 */

    /* 实验性选项 */
    // "experimentalDecorators": true,         /* 启用对 ES7 装饰器的实验性支持。 *///  "emitDecoratorMetadata": true,
          /* 启用对装饰器类型元数据发射的实验性支持。 */

  }
}
```

大多数（如果不是全部）选项都是自动生成的，只有必要的选项未被注释。

较旧版本的 TypeScript，例如 v2.0.x 及更低版本，会生成如下的 tsconfig.json：

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false
    }
}
```

第28.2节：减少编程错误的配置

有一些非常好的配置可以强制类型检查并获得更有帮助的错误提示，但默认未启用。

```
{
  "compilerOptions": {

    "alwaysStrict": true, // 以严格模式解析并为每个源文件生成 "use strict"。

    // 如果引用的文件大小写错误，例如文件名是 Global.ts，但你用 ///
<reference path="global.ts" /> 来引用该文件，则可能导致意外错误。
访问：http://stackoverflow.com/questions/36628612/typescript-transpiler-casing-issue
    "forceConsistentCasingInFileNames": true, // 不允许对同一文件使用大小写不一致的引用。


    // "allowUnreachableCode": false, // 不报告不可达代码的错误。（默认：False）
    // "allowUnusedLabels": false, // 不报告未使用标签的错误。（默认：False）
```

```
or 'classic' (TypeScript pre-1.6). */
    // "baseUrl": "./",                       /* Base directory to resolve non-absolute module names.
*/
    // "paths": {},                           /* A series of entries which re-map imports to lookup
locations relative to the 'baseUrl'. */
    // "rootDirs": [],                        /* List of root folders whose combined content
represents the structure of the project at runtime. */
    // "typeRoots": [],                       /* List of folders to include type definitions from. */
    // "types": [],                           /* Type declaration files to be included in
compilation. */
    // "allowSyntheticDefaultImports": true,  /* Allow default imports from modules with no default
export. This does not affect code emit, just typechecking. */

    /* Source Map Options */
    // "sourceRoot": "./",                    /* Specify the location where debugger should locate
TypeScript files instead of source locations. */
    // "mapRoot": "./",                       /* Specify the location where debugger should locate
map files instead of generated locations. */
    // "inlineSourceMap": true,               /* Emit a single file with source maps instead of
having a separate file. */
    // "inlineSources": true,                 /* Emit the source alongside the sourcemaps within a
single file; requires '--inlineSourceMap' or '--sourceMap' to be set. */

    /* Experimental Options */
    // "experimentalDecorators": true,        /* Enables experimental support for ES7 decorators. */
    // "emitDecoratorMetadata": true,         /* Enables experimental support for emitting type
metadata for decorators. */
  }
}
```

Most, if not all, options are generated automatically with only the bare necessities left uncommented.

Older versions of TypeScript, like for example v2.0.x and lower, would generate a tsconfig.json like this:

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false
    }
}
```

Section 28.2: Configuration for fewer programming errors

There are very good configurations to force typings and get more helpful errors which are not activated by default.

```
{
  "compilerOptions": {

    "alwaysStrict": true, // Parse in strict mode and emit "use strict" for each source file.

    // If you have wrong casing in referenced files e.g. the filename is Global.ts and you have a ///
<reference path="global.ts" /> to reference this file, then this can cause to unexpected errors.
Visite: http://stackoverflow.com/questions/36628612/typescript-transpiler-casing-issue
    "forceConsistentCasingInFileNames": true, // Disallow inconsistently-cased references to the
same file.

    // "allowUnreachableCode": false, // Do not report errors on unreachable code. (Default: False)
    // "allowUnusedLabels": false, // Do not report errors on unused labels. (Default: False)
    "noFallthroughCasesInSwitch": true, // 报告 switch 语句中贯穿（fall through）情况的错误。
    "noImplicitReturns": true, // 当函数的所有代码路径未返回值时报告错误。

    "noUnusedParameters": true, // 报告未使用参数的错误。
    "noUnusedLocals": true, // 报告未使用局部变量的错误。

    "noImplicitAny": true, // 对隐式"any"类型的表达式和声明抛出错误。

    "noImplicitThis": true, // 对隐式"any"类型的 this 表达式抛出错误。

    "strictNullChecks": true, // null 和 undefined 值不属于每种类型的域
且仅可赋值给它们自身和 any 类型。

    // 要强制执行此规则，请添加此配置。
    "noEmitOnError": true     // 如果报告了任何错误，则不生成输出。
  }
}
```

还不够？如果你是个硬核程序员并且想要更多，那么你可能会对在用 tsc 编译之前用 tslint 检查你的 TypeScript 文件感兴趣。查看如何配置 tslint 以实现更严格的代码规范。

第28.3节：compileOnSave

设置顶层属性 compileOnSave 表示 IDE 在保存时为给定的 tsconfig.json 生成所有文件。

```
{
    "compileOnSave": true,
    "compilerOptions": {
      ...
    },
    "exclude": [
      ...
    ]
}
```

此功能自 TypeScript 1.8.4 及以后版本可用，但需要 IDE 直接支持。目前，支持的 IDE 示例有：

- Visual Studio 2015 带 Update 3
- JetBrains WebStorm
- Atom with atom-typescript

第28.4节：注释

tsconfig.json 文件可以包含行注释和块注释，使用与 ECMAScript 相同的规则。

```
//前置注释
{
    "compilerOptions": {
        //这是一个行注释
        "module": "commonjs", //行尾注释
        "target" /*内联块注释*/ : "es5",
        /* 这是一个
       块
       注释 */
    }
}
```

```
    "noFallthroughCasesInSwitch": true, // Report errors for fall through cases in switch statement.
    "noImplicitReturns": true, // Report error when not all code paths in function return a value.

    "noUnusedParameters": true, // Report errors on unused parameters.
    "noUnusedLocals": true, // Report errors on unused locals.

    "noImplicitAny": true, // Raise error on expressions and declarations with an implied "any"
type.
    "noImplicitThis": true, // Raise error on this expressions with an implied "any" type.

    "strictNullChecks": true, // The null and undefined values are not in the domain of every type
and are only assignable to themselves and any.

    // To enforce this rules, add this configuration.
    "noEmitOnError": true     // Do not emit outputs if any errors were reported.
  }
}
```

Not enough? If you are a hard coder and want more, then you may be interested to check your TypeScript files with tslint before compiling it with tsc. Check how to configure tslint for even stricter code.

Section 28.3: compileOnSave

Setting a top-level property `compileOnSave` signals to the IDE to generate all files for a given **tsconfig.json** upon saving.

```
{
    "compileOnSave": true,
    "compilerOptions": {
        ...
    },
    "exclude": [
        ...
    ]
}
```

This feature is available since TypeScript 1.8.4 and onward, but needs to be directly supported by IDE's. Currently, examples of supported IDE's are:

- Visual Studio 2015 with Update 3
- JetBrains WebStorm
- Atom with atom-typescript

Section 28.4: Comments

A tsconfig.json file can contain both line and block comments, using the same rules as ECMAScript.

```
//Leading comment
{
    "compilerOptions": {
        //this is a line comment
        "module": "commonjs", //eol line comment
        "target" /*inline block*/ : "es5",
        /* This is a
        block
        comment */
    }
}
```

```
/* 尾随注释 */
```

第28.5节：preserveConstEnums

TypeScript 支持通过 const enum 声明的常量枚举。

这通常只是语法糖，因为常量枚举会被内联到编译后的JavaScript中。

例如，以下代码

```
const enum 三态 {
    真,
假,
    未知
}

var 某物 = 三态.真;
```

编译为

```
var 某物 = 0;
```

虽然内联带来了性能提升，但即使是常量枚举，你可能仍然希望保留枚举（例如：你可能希望在开发代码中保持可读性），为此你需要在 **tsconfig.json** 中将 `preserveConstEnums` 选项设置为 the `compilerOptions` 中的 **true**。

```
{
    "compilerOptions": {
        "preserveConstEnums" = true,
        ...
    },
    "exclude": [
        ...
    ]
}
```

通过这种方式，前面的示例将像其他枚举一样编译，如以下代码片段所示。

```
var 三态;
(function (Tristate) {
Tristate[Tristate["True"] = 0] = "True";
    Tristate[Tristate["False"] = 1] = "False";
    Tristate[Tristate["Unknown"] = 2] = "Unknown";
})(Tristate || (Tristate = {}));

var something = Tristate.True
```

```
/* trailing comment */
```

Section 28.5: preserveConstEnums

TypeScript supports constant enumerables, declared through `const enum`.

This is usually just syntax sugar as the constant enums are inlined in compiled JavaScript.

For instance the following code

```
const enum Tristate {
    True,
    False,
    Unknown
}

var something = Tristate.True;
```

compiles to

```
var something = 0;
```

Although the performance benefit from inlining, you may prefer to keep enums even if constant (ie: you may wish readability on development code), to do this you have to set in **tsconfig.json** the `preserveConstEnums` clause into the `compilerOptions` to **true**.

```
{
    "compilerOptions": {
        "preserveConstEnums" = true,
        ...
    },
    "exclude": [
        ...
    ]
}
```

By this way the previous example would be compiled as any other enums, as shown in following snippet.

```
var Tristate;
(function (Tristate) {
    Tristate[Tristate["True"] = 0] = "True";
    Tristate[Tristate["False"] = 1] = "False";
    Tristate[Tristate["Unknown"] = 2] = "Unknown";
})(Tristate || (Tristate = {}));

var something = Tristate.True
```

第29章：调试

运行和调试TypeScript有两种方式：

转译为JavaScript，在node中运行并使用映射回溯到TypeScript源文件

或者

直接使用 ts-node 运行TypeScript

本文介绍了使用Visual Studio Code和WebStorm的两种方法。所有示例均假设您的主文件是*index.ts*。

第29.1节：在WebStorm中使用ts-node运行TypeScript

将此脚本添加到您的package.json中：

```
"start:idea": "ts-node %NODE_DEBUG_OPTION% --ignore false index.ts",
```

右键点击脚本并选择创建 'test:idea'...，然后确认"确定"以创建调试配置：

Chapter 29: Debugging

There are two ways of running and debugging TypeScript:

Transpile to JavaScript, run in node and use mappings to link back to the TypeScript source files

or

Run TypeScript directly using ts-node

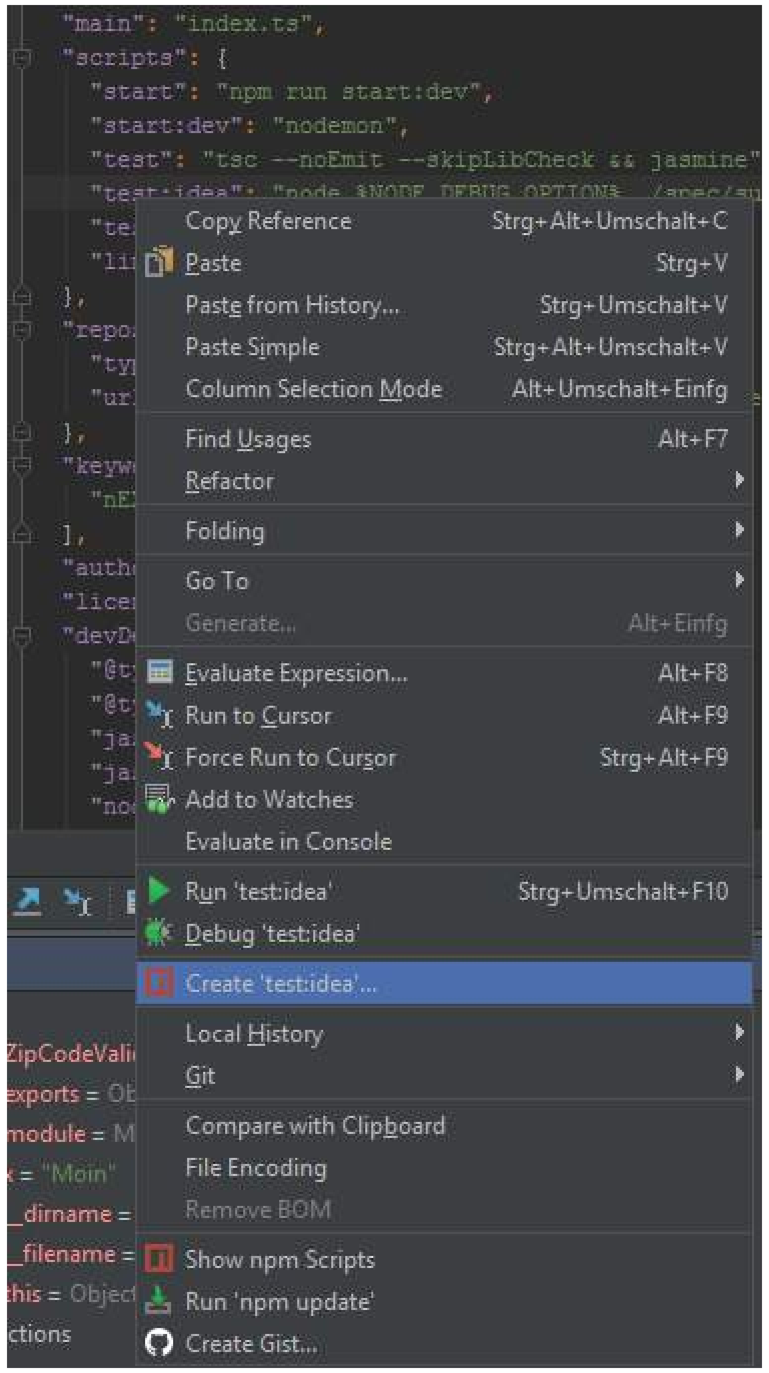
This article describes both ways using Visual Studio Code and WebStorm. All examples presume that your main file is *index.ts*.

Section 29.1: TypeScript with ts-node in WebStorm

Add this script to your `package.json`:

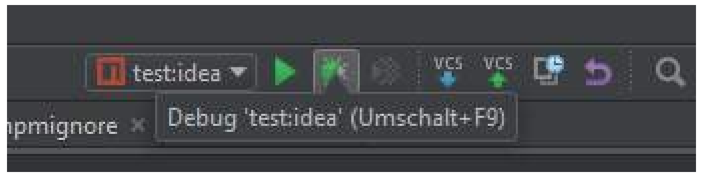
```
"start:idea": "ts-node %NODE_DEBUG_OPTION% --ignore false index.ts",
```

Right click on the script and select *Create 'test:idea'...* and confirm with 'OK' to create the debug configuration:



```
"main": "index.ts",
"scripts": {
  "start": "npm run start:dev",
  "start:dev": "nodemon",
  "test": "tsc --noEmit --skipLibCheck && jasmine"
```

使用此配置启动调试器：



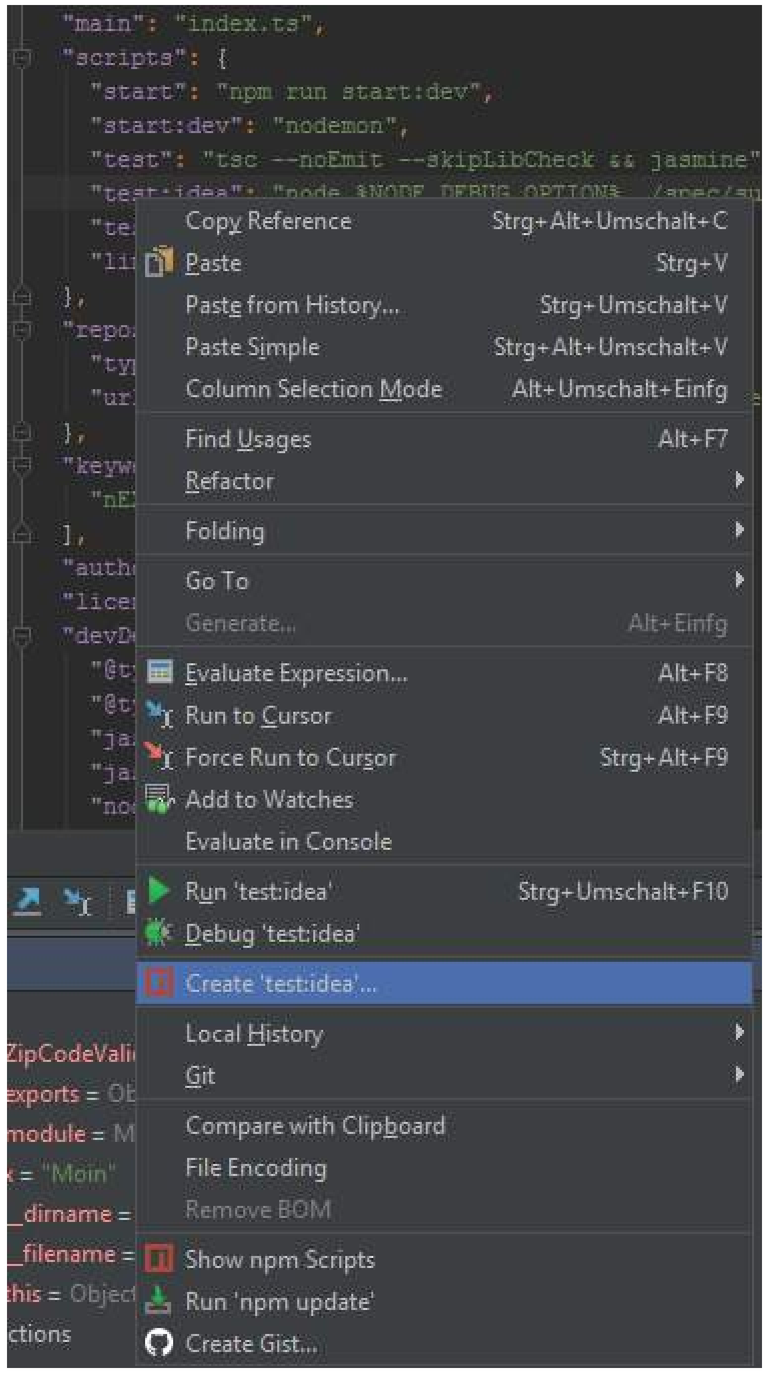
第29.2节：Visual Studio Code中的TypeScript与ts-node

将ts-node添加到你的TypeScript项目中：

```
npm i ts-node
```

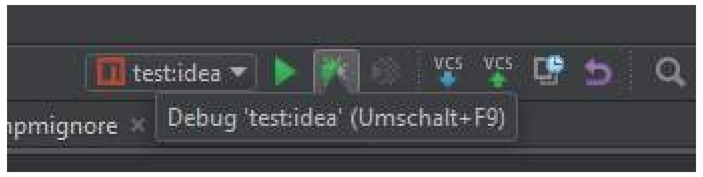
向你的package.json添加一个脚本：

```
"start:debug": "ts-node --inspect=5858 --debug-brk --ignore false index.ts"
```



```
"main": "index.ts",
"scripts": {
  "start": "npm run start:dev",
  "start:dev": "nodemon",
  "test": "tsc --noEmit --skipLibCheck && jasmine"
```

Start the debugger using this configuration:



Section 29.2: TypeScript with ts-node in Visual Studio Code

Add ts-node to your TypeScript project:

```
npm i ts-node
```

Add a script to your `package.json`:

```
"start:debug": "ts-node --inspect=5858 --debug-brk --ignore false index.ts"
```

需要配置launch.json以使用node2类型，并启动npm运行start:debug脚本：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node2",
            "request": "launch",
            "name": "启动程序",
            "runtimeExecutable": "npm",
            "windows": {
                "runtimeExecutable": "npm.cmd"
            },
            "runtimeArgs": [
                "run-script",
                "start:debug"
            ],
            "cwd": "${workspaceRoot}/server",
            "outFiles": [],
            "port": 5858,
            "sourceMaps": true
        }
    ]
}
```

第29.3节：Visual Studio Code中的带SourceMaps的JavaScript

在 tsconfig.json 文件中设置

```
"sourceMap": true,
```

使用 tsc 命令从TypeScript源文件生成js文件的同时生成映射。
launch.json 文件：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "启动程序",
            "program": "${workspaceRoot}\\index.js",
            "cwd": "${workspaceRoot}",
            "outFiles": [],
            "sourceMaps": true
        }
    ]
}
```

这会启动带有生成的 index.js（如果你的主文件是 index.ts）文件的 Node，并在 Visual Studio Code 中启动调试器，调试器会在断点处暂停并解析 TypeScript 代码中的变量值。

第29.4节：WebStorm中的带SourceMaps的JavaScript

创建一个Node.js调试配置并使用index.js作为Node参数。

The `launch.json` needs to be configured to use the *node2* type and start npm running the `start:debug` script:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node2",
            "request": "launch",
            "name": "Launch Program",
            "runtimeExecutable": "npm",
            "windows": {
                "runtimeExecutable": "npm.cmd"
            },
            "runtimeArgs": [
                "run-script",
                "start:debug"
            ],
            "cwd": "${workspaceRoot}/server",
            "outFiles": [],
            "port": 5858,
            "sourceMaps": true
        }
    ]
}
```

Section 29.3: JavaScript with SourceMaps in Visual Studio Code

In the `tsconfig.json` set

```
"sourceMap": true,
```

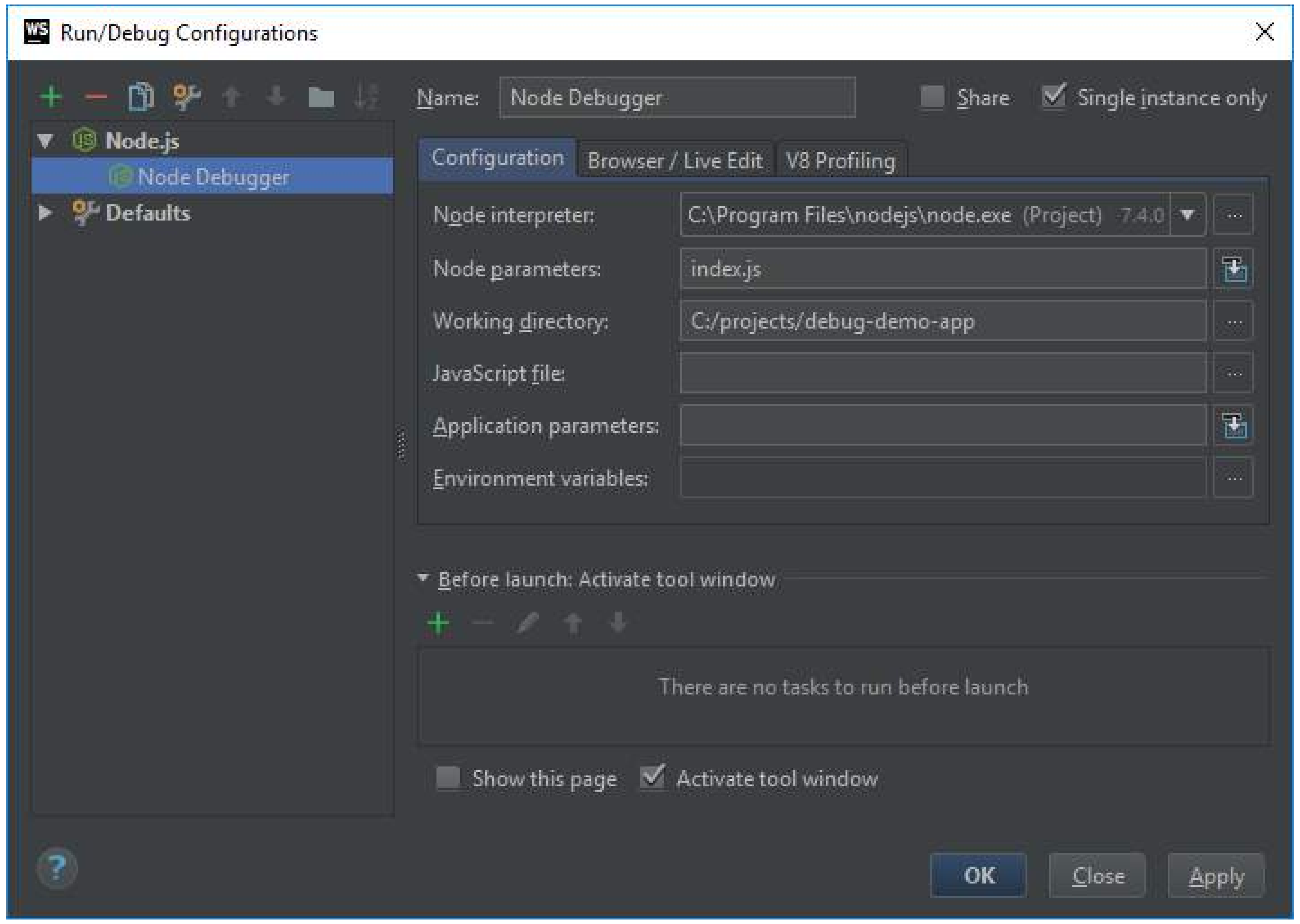
to generate mappings alongside with js-files from the TypeScript sources using the `tsc` command.
The launch.json file:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "program": "${workspaceRoot}\\index.js",
            "cwd": "${workspaceRoot}",
            "outFiles": [],
            "sourceMaps": true
        }
    ]
}
```

This starts node with the generated index.js (if your main file is index.ts) file and the debugger in Visual Studio Code which halts on breakpoints and resolves variable values within your TypeScript code.

Section 29.4: JavaScript with SourceMaps in WebStorm

Create a *Node.js* debug configuration and use `index.js` as *Node parameters*.


Run/Debug Configurations

Name: Node Debugger
Share [✓ Single instance only]

Node.js
Node Debugger
Defaults

Configuration | Browser / Live Edit | V8 Profiling

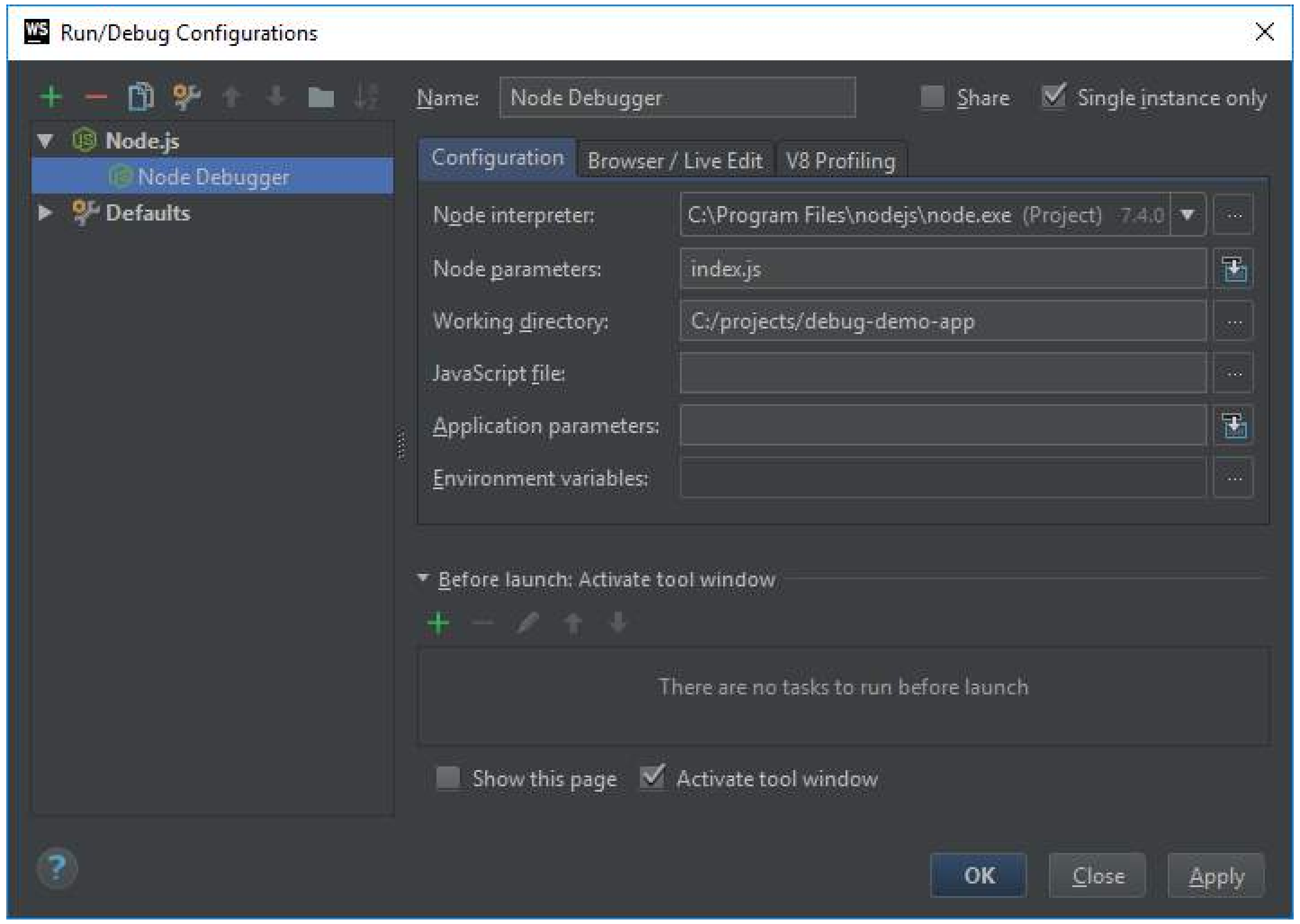
Node interpreter:	C:\Program Files\nodejs\node.exe (Project) 7.4.0
Node parameters:	index.js
Working directory:	C:/projects/debug-demo-app
JavaScript file:	
Application parameters:	
Environment variables:	

Before launch: Activate tool window

There are no tasks to run before launch

Show this page [✓ Activate tool window]

OK Close Apply


Run/Debug Configurations

Name: Node Debugger
Share [✓ Single instance only]

Node.js
Node Debugger
Defaults

Configuration | Browser / Live Edit | V8 Profiling

Node interpreter: C:\Program Files\nodejs\node.exe (Project) 7.4.0
Node parameters: index.js
Working directory: C:/projects/debug-demo-app
JavaScript file:
Application parameters:
Environment variables:

Before launch: Activate tool window

There are no tasks to run before launch

Show this page [✓ Activate tool window]

OK Close Apply

第30章：单元测试

第30.1节：tape

tape是一个极简的JavaScript测试框架，它输出符合TAP标准的标记。

使用npm安装ape，运行命令

```
npm install --save-dev tape @types/tape
```

要在 TypeScript 中使用 tape，需要全局安装 ts-node，运行以下命令即可

```
npm install -g ts-node
```

现在你可以开始编写你的第一个测试了

```
//math.test.ts
import * as test from "tape";

test("数学测试", (t) => {
t.equal(4, 2 + 2);
t.true(5 > 2 + 2);

t.end();
});
```

执行测试请运行命令

```
ts-node node_modules/tape/bin/tape math.test.ts
```

在输出中你应该看到

```
TAP 版本 13
# 数学测试
ok 1 应该相等
ok 2 应该为真

1..2
# 测试 2
# 通过 2

# ok
```

干得好，你刚刚运行了你的 TypeScript 测试。

运行多个测试文件

你可以使用路径通配符一次运行多个测试文件。要执行 tests 目录下的所有 TypeScript 测试，请运行命令

```
ts-node node_modules/tape/bin/tape tests/**/*.ts
```

Chapter 30: Unit Testing

Section 30.1: tape

tape is minimalistic JavaScript testing framework, it outputs TAP-compliant markup.

To install `tape` using `npm` run command

```
npm install --save-dev tape @types/tape
```

To use `tape` with TypeScript you need to install `ts-node` as global package, to do this run command

```
npm install -g ts-node
```

Now you are ready to write your first test

```
//math.test.ts
import * as test from "tape";

test("Math test", (t) => {
    t.equal(4, 2 + 2);
    t.true(5 > 2 + 2);

    t.end();
});
```

To execute test run command

```
ts-node node_modules/tape/bin/tape math.test.ts
```

In output you should see

```
TAP version 13
# Math test
ok 1 should be equal
ok 2 should be truthy

1..2
# tests 2
# pass  2

# ok
```

Good job, you just ran your TypeScript test.

Run multiple test files

You can run multiple test files at once using path wildcards. To execute all TypeScript tests in `tests` directory run command

```
ts-node node_modules/tape/bin/tape tests/**/*.ts
```

第 30.2 节：jest (ts-jest)

jest 是 Facebook 提供的无痛 JavaScript 测试框架，配合 ts-jest 可用于测试 TypeScript 代码。

使用 npm 运行命令安装 jest

```
npm install --save-dev jest @types/jest ts-jest typescript
```

为了方便使用，将jest安装为全局包

```
npm install -g jest
```

为了让jest支持TypeScript，需要在package.json中添加配置

```
//package.json
{
...
"jest": {
    "transform": {
      ".(ts|tsx)": "<rootDir>/node_modules/ts-jest/preprocessor.js"
    },
    "testRegex": "(/__tests__/.*|\\.(test|spec))\\.(ts|tsx|js)$",
    "moduleFileExtensions": ["ts", "tsx", "js"]
  }
}
```

现在jest已准备就绪。假设我们有一个示例fizz buz需要测试

```
//fizzBuzz.ts
export function fizzBuzz(n: number): string {
    let output = "";
    for (let i = 1; i <= n; i++) {
        if (i % 5 && i % 3) {
            output += i + ' ';
        }
        if (i % 3 === 0) {
            output += 'Fizz ';
        }
        if (i % 5 === 0) {
            output += 'Buzz ';
        }
    }
    return output;
}
```

示例测试可能如下所示

```
//FizzBuzz.test.ts
/// <reference types="jest" />

import {fizzBuzz} from "./fizzBuzz";
test("FizzBuzz 测试", () =>{
expect(fizzBuzz(2)).toBe("1 2 ");
    expect(fizzBuzz(3)).toBe("1 2 Fizz ");
});
```

执行测试运行

Section 30.2: jest (ts-jest)

jest is painless JavaScript testing framework by Facebook, with ts-jest can be used to test TypeScript code.

To install jest using npm run command

```
npm install --save-dev jest @types/jest ts-jest typescript
```

For ease of use install `jest` as global package

```
npm install -g jest
```

To make `jest` work with TypeScript you need to add configuration to `package.json`

```
//package.json
{
...
"jest": {
    "transform": {
      ".(ts|tsx)": "<rootDir>/node_modules/ts-jest/preprocessor.js"
    },
    "testRegex": "(/__tests__/.*|\\.(test|spec))\\.(ts|tsx|js)$",
    "moduleFileExtensions": ["ts", "tsx", "js"]
  }
}
```

Now `jest` is ready. Assume we have sample fizz buz to test

```
//fizzBuzz.ts
export function fizzBuzz(n: number): string {
    let output = "";
    for (let i = 1; i <= n; i++) {
        if (i % 5 && i % 3) {
            output += i + ' ';
        }
        if (i % 3 === 0) {
            output += 'Fizz ';
        }
        if (i % 5 === 0) {
            output += 'Buzz ';
        }
    }
    return output;
}
```

Example test could look like

```
//FizzBuzz.test.ts
/// <reference types="jest" />

import {fizzBuzz} from "./fizzBuzz";
test("FizzBuzz test", () =>{
    expect(fizzBuzz(2)).toBe("1 2 ");
    expect(fizzBuzz(3)).toBe("1 2 Fizz ");
});
```

To execute test run

```
jest
```

在输出中你应该看到

```
通过  ./fizzBuzz.test.ts
✓ FizzBuzz 测试 (3毫秒)

测试套件：1 通过，1 总计
测试：      1 通过，1 总计
快照：      0 总计
时间：       1.46秒，预计 2秒
运行了所有测试套件。
```

代码覆盖率

jest 支持生成代码覆盖率报告。

要在 TypeScript 中使用代码覆盖率，需要在package.json中添加另一行配置。

```
{
...
  "jest": {
  …
    "testResultsProcessor": "<rootDir>/node_modules/ts-jest/coverageprocessor.js"
  }
}
```

要运行带有覆盖率报告生成的测试，请运行

```
jest --coverage
```

如果与我们的示例 fizz buzz 一起使用，您应该会看到

```
通过  ./fizzBuzz.test.ts
✓ FizzBuzz 测试 (3毫秒)

-------------|----------|----------|----------|----------|----------------|
文件         |  % 语句  | % 分支   |  % 函数  |  % 行    |未覆盖行数      |
-------------|----------|----------|----------|----------|----------------|
所有文件     |    92.31 |     87.5 |      100 |    91.67 |                |
fizzBuzz.ts |    92.31 |     87.5 |      100 |    91.67 |             13 |
-------------|----------|----------|----------|----------|----------------|
测试套件：1 通过，1 总计
测试：      1 通过，1 总计
快照：      0 总计
时间：      1.857秒
运行了所有测试套件。
```

`jest` 还创建了文件夹 `coverage`，里面包含多种格式的覆盖率报告，包括用户友好的 html 报告，位于 `coverage/lcov-report/index.html`

```
jest
```

In output you should see

```
 PASS  ./fizzBuzz.test.ts
✓ FizzBuzz test (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.46s, estimated 2s
Ran all test suites.
```

Code coverage

`jest` supports generation of code coverage reports.

To use code coverage with TypeScript you need to add another configuration line to `package.json`.

```
{
...
  "jest": {
  ...
    "testResultsProcessor": "<rootDir>/node_modules/ts-jest/coverageprocessor.js"
  }
}
```

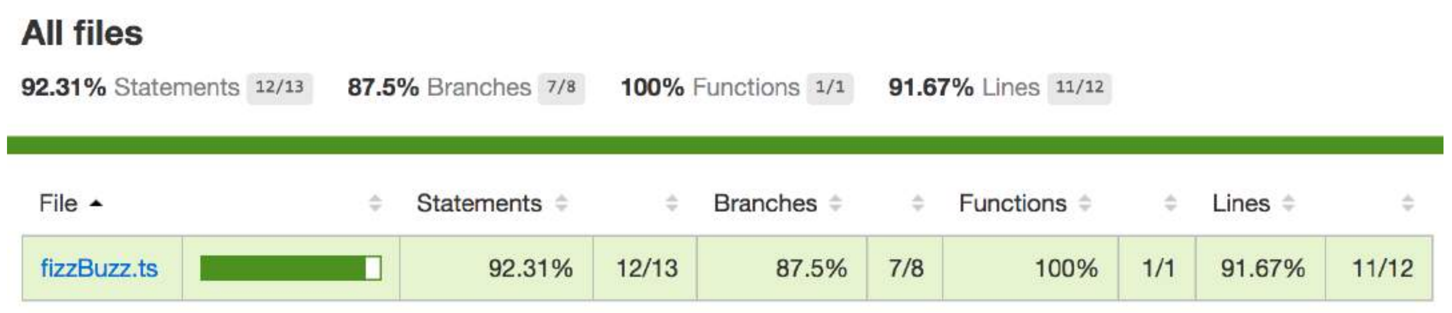
To run tests with generation of coverage report run

```
jest --coverage
```

If used with our sample fizz buzz you should see

```
 PASS  ./fizzBuzz.test.ts
✓ FizzBuzz test (3ms)

-------------|----------|----------|----------|----------|----------------|
File         |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-------------|----------|----------|----------|----------|----------------|
All files    |    92.31 |     87.5 |      100 |    91.67 |                |
fizzBuzz.ts |    92.31 |     87.5 |      100 |    91.67 |             13 |
-------------|----------|----------|----------|----------|----------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.857s
Ran all test suites.
```

`jest` also created folder `coverage` which contains coverage report in various formats, including user friendly html report in `coverage/lcov-report/index.html`



All files

92.31% Statements 12/13 87.5% Branches 7/8 100% Functions 1/1 91.67% Lines 11/12

File ▲		Statements		Branches		Functions		Lines	
fizzBuzz.ts		92.31%	12/13	87.5%	7/8	100%	1/1	91.67%	11/12

第 30.3 节：Alsatian

Alsatian 是一个用 TypeScript 编写的单元测试框架。它支持使用测试用例，并输出符合 TAP-标准的标记。

要使用它，请通过 npm 安装：

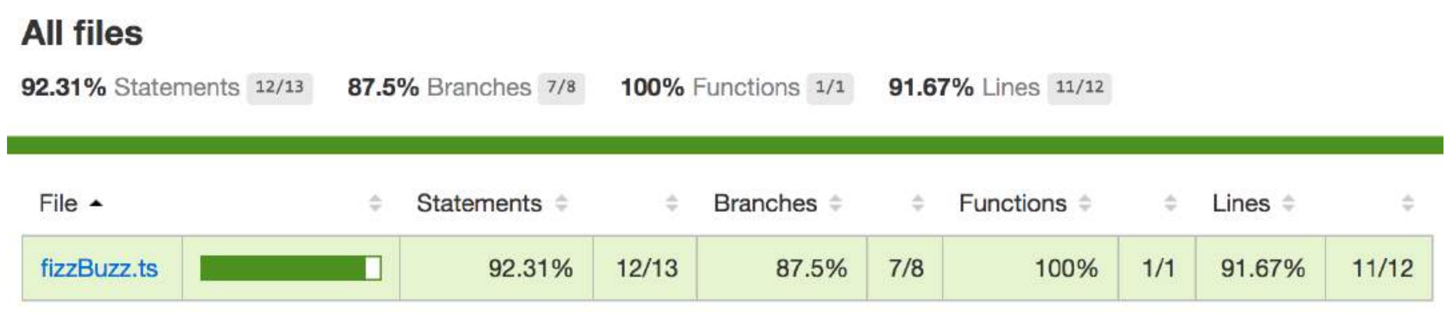
```
npm install alsatian --save-dev
```

然后设置一个测试文件：

```
import { Expect, Test, TestCase } from "alsatian";
import { SomeModule } from "../src/some-module";

export SomeModuleTests {

    @Test()
public statusShouldBeTrueByDefault() {
        let instance = new SomeModule();

Expect(instance.status).toBe(true);
    }

    @Test("Name should be null by default")
    public nameShouldBeNullByDefault() {
        let instance = new SomeModule();

Expect(instance.name).toBe(null);
    }

    @TestCase("first name")
    @TestCase("apples")
public shouldSetNameCorrectly(name: string) {
        let instance = new SomeModule();

instance.setName(name);

        Expect(instance.name).toBe(name);
    }

}
```

有关完整文档，请参见Alsatian的GitHub仓库。

第30.4节：chai-immutable 插件



All files

92.31% Statements 12/13 87.5% Branches 7/8 100% Functions 1/1 91.67% Lines 11/12

File ▲		Statements		Branches		Functions		Lines	
fizzBuzz.ts		92.31%	12/13	87.5%	7/8	100%	1/1	91.67%	11/12

Section 30.3: Alsatian

Alsatian is a unit testing framework written in TypeScript. It allows for usage of Test Cases, and outputs TAP-compliant markup.

To use it, install it from npm:

```
npm install alsatian --save-dev
```

Then set up a test file:

```
import { Expect, Test, TestCase } from "alsatian";
import { SomeModule } from "../src/some-module";

export SomeModuleTests {

    @Test()
    public statusShouldBeTrueByDefault() {
        let instance = new SomeModule();

        Expect(instance.status).toBe(true);
    }

    @Test("Name should be null by default")
    public nameShouldBeNullByDefault() {
        let instance = new SomeModule();

        Expect(instance.name).toBe(null);
    }

    @TestCase("first name")
    @TestCase("apples")
    public shouldSetNameCorrectly(name: string) {
        let instance = new SomeModule();

        instance.setName(name);

        Expect(instance.name).toBe(name);
    }

}
```

For a full documentation, see Alsatian's GitHub repo.

Section 30.4: chai-immutable plugin

1. 从npm安装chai、chai-immutable和ts-node

```
npm install --save-dev chai chai-immutable ts-node
```

2. 安装mocha和chai的类型定义

```
npm install --save-dev @types/mocha @types/chai
```

3. 编写简单的测试文件：

```
  import {List, Set} from 'immutable';
  import * as chai from 'chai';
  import * as chaiImmutable from 'chai-immutable';

chai.use(chaiImmutable);

  describe('chai immutable example', () => {
    it('example', () => {
expect(Set.of(1,2,3)).to.not.be.empty;

      expect(Set.of(1,2,3)).to.include(2);
      expect(Set.of(1,2,3)).to.include(5);
    })
  })
```

4. 在控制台运行：

```
mocha --compilers ts:ts-node/register,tsx:ts-node/register 'test/**/*.spec.@(ts|tsx)'
```

1. Install from npm chai, chai-immutable, and ts-node

```
npm install --save-dev chai chai-immutable ts-node
```

2. Install types for mocha and chai

```
npm install --save-dev @types/mocha @types/chai
```

3. Write simple test file:

```
import {List, Set} from 'immutable';
import * as chai from 'chai';
import * as chaiImmutable from 'chai-immutable';

chai.use(chaiImmutable);

describe('chai immutable example', () => {
  it('example', () => {
    expect(Set.of(1,2,3)).to.not.be.empty;

    expect(Set.of(1,2,3)).to.include(2);
    expect(Set.of(1,2,3)).to.include(5);
  })
})
```

4. Run it in the console:

```
mocha --compilers ts:ts-node/register,tsx:ts-node/register 'test/**/*.spec.@(ts|tsx)'
```

致谢

非常感谢所有来自 Stack Overflow Documentation 的人员，他们帮助提供了这些内容，
更多更改可以发送至 web@petercv.com 以发布或更新新内容

2426021684	第1章、第14章和第16章
ABabin	第9章
亚历克·汉森	第1章
亚历克斯·菲拉托夫	第22章和第27章
阿尔蒙德	第14章
阿米纳达夫	第9章
阿隆	第9章
阿尔捷姆	第9章、第14章和第25章
布拉克斯（Blackus）	第14章
布尼兰（bnieland）	第28章
布拉德（br4d）	第6章
布鲁诺LM（BrunoLM）	第1章、第17章和第22章
布鲁图斯（Brutus）	第14章
钱斯M（ChanceM）	第1章
科布斯·克鲁格	第9章
danvk	第1、2和11章
dimitrisli	第5章
dublicator	第14章
Equiman	第7章
Fenton	第3和18章
弗洛里安·哈默勒	第5章
Fylax	第1章、第3章和第28章
goenning	第28章
hansmaad	第7章和第10章
哈里	第14章
伊拉克利·基塔里什维利	第17章和第26章
islandman93	第1章、第6章、第9章、第14章和第26章
詹姆斯·蒙格	第7章、第27章和第30章
JKillian	第11章和第14章
乔尔·戴	第14章
约翰·鲁德尔	第22章
约书亚·布里登	第1章和第9章
朱利恩	第3章和第28章
贾斯汀·奈尔斯	第7章
k0pernikus	第1章和第27章
凯文·蒙特罗斯	第5章、第12章和第19章
凯文·杜斯	第22章
KnottytOmo	第1章、第10章和第14章
库巴·贝拉内克	第1章
莱克纳斯	第1章
leonidv	第30章
lilezek	第23章
Magu	第3、27和28章
马特·利什曼	第1章
马修·哈伍德	第30章
米哈伊尔	第1和3章
mleko	第1章、第15章、第22章、第27章和第30章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

2426021684	Chapters 1, 14 and 16
ABabin	Chapter 9
Alec Hansen	Chapter 1
Alex Filatov	Chapters 22 and 27
Almond	Chapter 14
Aminadav	Chapter 9
Aron	Chapter 9
artem	Chapters 9, 14 and 25
Blackus	Chapter 14
bnieland	Chapter 28
br4d	Chapter 6
BrunoLM	Chapters 1, 17 and 22
Brutus	Chapter 14
ChanceM	Chapter 1
Cobus Kruger	Chapter 9
danvk	Chapters 1, 2 and 11
dimitrisli	Chapter 5
dublicator	Chapter 14
Equiman	Chapter 7
Fenton	Chapters 3 and 18
Florian Hämmerle	Chapter 5
Fylax	Chapters 1, 3 and 28
goenning	Chapter 28
hansmaad	Chapters 7 and 10
Harry	Chapter 14
irakli khitarishvili	Chapters 17 and 26
islandman93	Chapters 1, 6, 9, 14 and 26
James Monger	Chapters 7, 27 and 30
JKillian	Chapters 11 and 14
Joel Day	Chapter 14
John Ruddell	Chapter 22
Joshua Breeden	Chapters 1 and 9
Juliën	Chapters 3 and 28
Justin Niles	Chapter 7
k0pernikus	Chapters 1 and 27
Kevin Montrose	Chapters 5, 12 and 19
Kewin Dousse	Chapter 22
KnottytOmo	Chapters 1, 10 and 14
Kuba Beránek	Chapter 1
Lekhnath	Chapter 1
leonidv	Chapter 30
lilezek	Chapter 23
Magu	Chapters 3, 27 and 28
Matt Lishman	Chapter 1
Matthew Harwood	Chapter 30
Mikhail	Chapters 1 and 3
mleko	Chapters 1, 15, 22, 27 and 30

穆策里希	第6章
穆罕默德·阿韦斯	第10章
保罗·布特斯	第9章
人员配备	第29章
拉胡尔	第20章和第21章
拉贾布·沙基罗夫	第14章和第26章
RationalDev	第1和3章
雷莫·H·詹森	第8章
罗宾	第7章
罗曼·M·科斯	第24章
罗伊·迪克图斯	第1章
赛福尔·阿扎德	第1章和第9章
萨姆	第1章
萨姆·艾尔文	第1章
SilentLupin	第6章
斯拉瓦·什皮塔尔尼	第6、9、10和14章
smnbbrv	第5章
斯特凡·莱因	第24章
Sunnyok	第9章
泰泰	第10章
乌德雷·纳蒂	第4章
user3893988	第28章
瓦希什特	第13章
芥末风扇	第1章

muetzerich	Chapter 6
Muhammad Awais	Chapter 10
Paul Boutes	Chapter 9
Peopleware	Chapter 29
Rahul	Chapters 20 and 21
Rajab Shakirov	Chapters 14 and 26
RationalDev	Chapters 1 and 3
Remo H. Jansen	Chapter 8
Robin	Chapter 7
Roman M. Koss	Chapter 24
Roy Dictus	Chapter 1
Saiful Azad	Chapters 1 and 9
Sam	Chapter 1
samAlvin	Chapter 1
SilentLupin	Chapter 6
Slava Shpitalny	Chapters 6, 9, 10 and 14
smnbbrv	Chapter 5
Stefan Rein	Chapter 24
Sunnyok	Chapter 9
Taytay	Chapter 10
Udlei Nati	Chapter 4
user3893988	Chapter 28
vashishth	Chapter 13
Wasabi Fan	Chapter 1

C#
Notes for Professionals
700+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

CSS
Notes for Professionals
200+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

AngularJS
Notes for Professionals
100+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

HTML5
Notes for Professionals
100+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Java
Notes for Professionals
900+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

JavaScript
Notes for Professionals
400+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

jQuery
Notes for Professionals
50+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Python
Notes for Professionals
700+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

SQL
Notes for Professionals
100+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books