前言

> *本小册是《千锋大前端小册》系列之 TypeScript 部分。通过本小册，可以系统学习TypeScript基础知识，为将来TypeScript在大前端项目的应用打下坚实的基础。*
>
> *—— 作者：千锋教育·古艺散人*

什么是TypeScript

TypeScript是Microsoft公司注册商标。

TypeScript具有类型系统，且是JavaScript的超集。 它可以编译成普通的JavaScript代码。 TypeScript支持任意浏览器，任意环境，任意系统并且是开源的。

安装TypeScript

1、在全局环境里安装TS

```
npm install -g typescript
```

2、用 tsc 命令编译 .ts 文件

app.ts 文件：

```
let title: string = '千锋教育'
```

在命令行里输入以下命令都可以将.ts文件编译为.js文件：

```
tsc ./src/app.ts --outFile ./dist/app.js
tsc ./src/* --outDir ./dist --watch
```

3、tsconfig.json 配置文件

在命令行里输入 `tsc --init` 命令，创建一个 tsconfig.json 文件，在此配置文件里修改：

```
"outDir": "./dist",
"rootDir": "./src"
```

5分钟了解TypeScript

构建第一个TypeScript文件

在编辑器，将下面的代码输入到 *src/greeter.ts* 文件里。我们注意到 *person: string*，表示 string 是 person 函数的参数类型注解：

```
function greeter(person: string) {
 return "Hello, " + person
}

let user = "Jane User"
console.log(greeter(user))
```

类型注解

TypeScript里的类型注解是一种轻量级的为函数或变量添加约束的方式。 在这个例子里，我们希望greeter函数接收一个字符串参数。 然后尝试把 greeter的调用改成传入一个数组：

```
function greeter(person: string) {
 return "Hello, " + person
}

let user = [0, 1, 2]
console.log(greeter(user))
```

重新编译，你会看到产生了一个错误：

```
Argument of type 'number[]' is not assignable to parameter of type 'string'.
```

类似地，尝试删除greeter调用的所有参数。 TypeScript会告诉你使用了非期望个数的参数调用了这个函数。 在这两种情况中，TypeScript提供了静态的代码分析，它可以分析代码结构和提供的类型注解。

要注意的是尽管有错误，greeter.js文件还是被创建了。 就算你的代码里有错误，你仍然可以使用TypeScript。但在这种情况下，TypeScript会警告你代码可能不会按预期执行。

接口

让我们开发这个示例应用。这里我们使用接口来描述一个拥有firstName和lastName字段的对象。 在

TypeScript里，只在两个类型内部的结构兼容那么这两个类型就是兼容的。 这就允许我们在实现接口时候只要保证包含了接口要求的结构就可以，而不必明确地使用 implements语句。

```
interface Person {
 firstName: string
 lastName: string
}

function greeter(person: Person) {
 return "Hello, " + person.firstName + " " + person.lastName
}

let user = { firstName: "Jane", lastName: "User" }
console.log(greeter(user))
```

类

最后，让我们使用类来改写这个例子。 TypeScript支持JavaScript的新特性，比如支持基于类的面向对象编程。让我们创建一个Student类，它带有一个构造函数和一些公共字段。 注意类和接口可以一起工作。

还要注意的是，在构造函数的参数上使用public等同于创建了同名的成员变量。

> ***注：***`public`*修饰符会引发* `Parameter 'firstName' implicitly has an 'any' type.` *，解决方法是在* `tsconfig.json` *文件中，添加* `"noImplicitAny": false` *，或者将* `"strict": true` *，改为* `"strict": false`

```
class Student {
 fullName: string
 constructor(public firstName, public middleInitial, public lastName) {
 this.fullName = firstName + " " + middleInitial + " " + lastName
 }
}

interface Person {
 firstName: string
 lastName: string
}

function greeter(person: Person) {
 return "Hello, " + person.firstName + " " + person.lastName
}

```

```
let user = new Student("Jane", "M.", "User")
console.log(greeter(user))
```

基础

基础类型-入门

介绍

TypeScript包含的最简单的数据单元有：数字，字符串，布尔值，Null 和 Undefined等。TypeScript支持与JavaScript几乎相同的数据类型，此外还提供了实用的枚举类型方便我们使用。本节介绍基础类型的布尔值、数字、字符串、数组、元组、枚举、any 和 void 等，其他几种基础类型详见 基础类型-高级。

布尔值

最基本的数据类型就是简单的 `true/false` 值，在JavaScript和TypeScript里叫做 `boolean` 。

```
let isDone: boolean = false
```

数字

和JavaScript一样，TypeScript里的所有数字都是浮点数。 这些浮点数的类型是 `number` 。

```
let decLiteral: number = 6
```

字符串

TypeScript像其它语言里一样，使用 `string` 表示文本数据类型。 和JavaScript一样，可以使用双引号（”）或单引号（’）表示字符串。

```
let from: string = "千锋教育"
from = "好程序员"
```

也使用模版字符串，定义多行文本和内嵌表达式。 这种字符串是被反引号包围（`），并且以${ expr }这种形式嵌入表达式。

```
let surname: string = `Felix`
```

```
let age: number = 37
let sentence: string = `Hello, my name is ${ surname }.

I'll be ${ age + 1 } years old next month.`
```

数组

TypeScript像JavaScript一样可以操作数组元素。 有两种方式可以定义数组。 第一种，可以在元素类型后面接上 `[]` ，表示由此类型元素组成的一个数组：

```
let list: number[] = [1, 2, 3]
```

第二种方式是使用数组泛型，Array<元素类型>：

```
let list: Array<number> = [1, 2, 3]
```

元组 Tuple

元组类型允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。 比如，你可以定义一对值分别为 string和number类型的元组。

```
// 声明一个元组类型 x
let x: [string, number]
// 初始化 x
x = ['hello', 10] // OK
// 无效的初始值
x = [10, 'hello'] // Error
```

当访问一个已知索引的元素，会得到正确的类型：

```
console.log(x[0].substr(1)) // OK
console.log(x[1].substr(1)) // Error, 'number' 不存在 'substr' 方法
```

当访问一个越界的元素，会出现错误：

```
x[3] = "world" // Error, '[string, number]' 未定义第 3 个元素的类型.
console.log(x[5].toString()) // Error, '[string, number]' 未定义第 5 个元素的类型.
```

枚举

enum类型是对JavaScript标准数据类型的一个补充。 使用枚举类型可以为一组数值赋予友好的名字。

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green
```

默认情况下，从 0 开始为元素编号。 你也可以手动的指定成员的数值。 例如，我们将上面的例子改成从 1 开始编号：

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green
```

或者，全部都采用手动赋值：

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green
```

枚举类型提供的一个便利是你可以由枚举的值得到它的名字。 例如，我们知道数值为2，但是不确定它映射到Color里的哪个名字，我们可以查找相应的名字：

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2]
console.log(colorName) // 'Green'
```

any

有时候，我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。 这些值可能来自于动态的内容，比如来自用户输入或第三方代码库。 这种情况下，我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。 那么我们可以使用 `any` 类型来标记这些变量：

```
let notSure: any = 4
notSure = "maybe a string instead" // OK 赋值了一个字符串
notSure = false // OK 赋值了一个布尔值
```

在对现有代码进行改写的时候， `any` 类型是十分有用的，它允许你在编译时可选择地包含或移除类型检查。

```
let notSure: any = 4
notSure.ifItExists() // okay, ifItExists函数在运行时可能存在
notSure.toFixed() // okay, toFixed 函数存在 (在编译时不做检查)
```

当你只知道一部分数据的类型时， `any` 类型也是有用的。 比如，你有一个数组，它包含了不同的类型的数据：

```
let list: any[] = [1, true, "free"]
list[1] = 100
```

void

某种程度上来说， `void` 类型像是与 `any` 类型相反，它表示没有任何类型。 当一个函数没有返回值时，你通常会见到其返回值类型是 `void` ：

```
function echo(): void {
 console.log('做真实的自己，用良心做教育')
}
```

声明一个void类型的变量没有什么大用，因为你只能为它赋予undefined和null：

```
let unusable: void = undefined
let greeting: void = 'hello world' // void 类型不能赋值为字符串
```

函数的类型

函数声明

在 JavaScript 中，有两种常见的定义函数的方式——函数声明（Function Declaration）和函数表达式（Function Expression）：

```
// 函数声明（Function Declaration）
function sum(x, y) {
 return x + y
}

// 函数表达式（Function Expression）
let mySum = function (x, y) {
 return x + y
}
```

一个函数有输入和输出，要在 TypeScript 中对其进行约束，需要把输入和输出都考虑到，其中函数声明的类型定义较简单：

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

注意，**输入多余的（或者少于要求的）参数，是不被允许的**：

```
function sum(x: number, y: number): number {
 return x + y
}
sum(1, 2, 3)

// Expected 2 arguments, but got 3.
function sum(x: number, y: number): number {
```

```
  return x + y
}
sum(1)

// An argument for 'y' was not provided.
```

函数表达式

如果要我们现在写一个对函数表达式（Function Expression）的定义，可能会写成这样：

```
let mySum = function (x: number, y: number): number {
  return x + y
}
```

这是可以通过编译的，不过事实上，上面的代码只对等号右侧的匿名函数进行了类型定义，而等号左边的 `mySum` ，是通过赋值操作进行类型推论而推断出来的。如果需要我们手动给 `mySum` 添加类型，则应该是这样：

```
let mySum: (x: number, y: number) => number = function (x: number, y: number): number {
  return x + y
}
```

注意不要混淆了 TypeScript 中的 `=>` 和 ES6 中的 `=>` 。

在 TypeScript 的类型定义中， `=>` 用来表示函数的定义，左边是输入类型，需要用括号括起来，右边是输出类型。

在 ES6 中， `=>` 叫做箭头函数，应用十分广泛，可以参考 [ES6 中的箭头函数][]。

用接口定义函数的形状

我们也可以使用接口的方式来定义一个函数需要符合的形状：

```
interface SearchFunc {
  (source: string, subString: string): boolean
}

let mySearch: SearchFunc
mySearch = function(source: string, subString: string) {
  return source.search(subString) !== -1
}
```

可选参数

前面提到，输入多余的（或者少于要求的）参数，是不允许的。那么如何定义可选的参数呢？

与接口中的可选属性类似，我们用 `?` 表示可选的参数：

```
function buildName(firstName: string, lastName?: string) {
 if (lastName) {
 return firstName + ' ' + lastName
 } else {
 return firstName
 }
}
let tomcat = buildName('Tom', 'Cat')
let tom = buildName('Tom')
```

需要注意的是，可选参数必须接在必需参数后面。换句话说，**可选参数后面不允许再出现必需参数了**：

```
function buildName(firstName?: string, lastName: string) {
 if (firstName) {
 return firstName + ' ' + lastName
 } else {
 return lastName
 }
}
let tomcat = buildName('Tom', 'Cat')
let tom = buildName(undefined, 'Tom')

// A required parameter cannot follow an optional parameter.
```

参数默认值

在 ES6 中，我们允许给函数的参数添加默认值，**TypeScript 会将添加了默认值的参数识别为可选参数**：

```
function buildName(firstName: string, lastName: string = 'Cat') {
 return firstName + ' ' + lastName
}
let tomcat = buildName('Tom', 'Cat')
let tom = buildName('Tom')
```

此时就不受「可选参数必须接在必需参数后面」的限制了：

```
function buildName(firstName: string = 'Tom', lastName: string) {
 return firstName + ' ' + lastName
```

```
}
let tomcat = buildName('Tom', 'Cat')
let cat = buildName(undefined, 'Cat')
```

关于默认参数，可以参考 [ES6 中函数参数的默认值][]。

剩余参数

ES6 中，可以使用 `...rest` 的方式获取函数中的剩余参数（rest 参数）：

```
function push(array, ...items) {
 items.forEach(function(item) {
 array.push(item)
 })
}

let a = []
push(a, 1, 2, 3)
```

事实上，`items` 是一个数组。所以我们可以用数组的类型来定义它：

```
function push(array: any[], ...items: any[]) {
 items.forEach(function(item) {
 array.push(item)
 })
}

let a = []
push(a, 1, 2, 3)
```

注意，rest 参数只能是最后一个参数，关于 rest 参数，可以参考 [ES6 中的 rest 参数][]。

重载

重载允许一个函数接受不同数量或类型的参数时，作出不同的处理。

比如，我们需要实现一个函数 `reverse`，输入数字 `123` 的时候，输出反转的数字 `321`，输入字符串 `'hello'` 的时候，输出反转的字符串 `'olleh'`。

利用联合类型，我们可以这么实现：

```
function reverse(x: number | string): number | string {
 if (typeof x === 'number') {
 return Number(x.toString().split('').reverse().join(''))
 } else if (typeof x === 'string') {
```

```
  } else if (typeof x === 'string') {
   return x.split('').reverse().join('')
  }
 }
```

然而这样有一个缺点，就是不能够精确的表达，输入为数字的时候，输出也应该为数字，输入为字符串的时候，输出也应该为字符串。

这时，我们可以使用重载定义多个 `reverse` 的函数类型：

```
function reverse(x: number): number
function reverse(x: string): string
function reverse(x: number | string): number | string {
 if (typeof x === 'number') {
 return Number(x.toString().split('').reverse().join(''))
 } else if (typeof x === 'string') {
 return x.split('').reverse().join('')
 }
}
```

上例中，我们重复定义了多次函数 `reverse` ，前几次都是函数定义，最后一次是函数实现。在编辑器的代码提示中，可以正确的看到前两个提示。

注意，TypeScript 会优先从最前面的函数定义开始匹配，所以多个函数定义如果有包含关系，需要优先把精确的定义写在前面。

接口

在 TypeScript 中，我们使用接口（Interfaces）来定义对象的类型。

什么是接口

在面向对象语言中，接口（Interfaces）是一个很重要的概念，它是对行为的抽象，而具体如何行动需要由类（classes）去实现（implement）。
TypeScript 中的接口是一个非常灵活的概念，除了可用于对类的一部分行为进行抽象以外，也常用于对「对象的形状（Shape）」进行描述。

简单的例子

```
interface Person {
 name: string
 age: number
```

```
  age: number
}

let tom: Person = {
  name: 'Tom',
  age: 25
}
```

上面的例子中，我们定义了一个接口 `Person`，接着定义了一个变量 `tom`，它的类型是 `Person`。这样，我们就约束了 `tom` 的形状必须和接口 `Person` 一致。

接口一般首字母大写。有的编程语言中会建议接口的名称加上 I 前缀。

定义的变量比接口少了一些属性是不允许的：

```
interface Person {
  name: string
  age: number
}

let tom: Person = {
  name: 'Tom'
}
// Property 'age' is missing in type '{ name: string }' but required in type 'Person'.
```

多一些属性也是不允许的：

```
interface Person {
  name: string
  age: number
}

let tom: Person = {
  name: 'Tom',
  age: 25,
  gender: 'male'
}

// Type '{ name: string age: number gender: string }' is not assignable to type 'Person'.
// Object literal may only specify known properties, and 'gender' does not exist in type 'Person'.
```

可见，*赋值的时候，变量的形状必须和接口的形状保持一致。*

可选属性

有时我们希望不要完全匹配一个形状，那么可以用可选属性：

```
interface Person {
 name: string
 age?: number
}

let tom: Person = {
 name: 'Tom'
}
```

```
interface Person {
 name: string
 age?: number
}

let tom: Person = {
 name: 'Tom',
 age: 25
}
```

可选属性的含义是该属性可以不存在。

这时仍然不允许添加未定义的属性：

```
interface Person {
 name: string
 age?: number
}

let tom: Person = {
 name: 'Tom',
 age: 25,
 gender: 'male'
}

// Type '{ name: string age: number gender: string }' is not assignable to type 'Person'.
// Object literal may only specify known properties, and 'gender' does not exist in type 'Person'.
```

任意属性

有时候我们希望一个接口允许有任意的属性，可以使用如下方式：

```
interface Person {
 name: string
 age?: number
 [propName: string]: any
}

let tom: Person = {
 name: 'Tom',
 gender: 'male'
}
```

使用 [propName: string] 定义了任意属性取 string 类型的值。

需要注意的是，一旦定义了任意属性，**那么确定属性和可选属性的类型都必须是它的类型的子集**：

```
interface Person {
 name: string
 age?: number
 [propName: string]: string
}

let tom: Person = {
 name: 'Tom',
 age: 25,
 gender: 'male'
}

// Property 'age' of type 'number | undefined' is not assignable to string index type 'string'.
// Type '{ name: string age: number gender: string }' is not assignable to type 'Person'.
// Property 'age' is incompatible with index signature.
// Type 'number' is not assignable to type 'string'.
```

上例中，任意属性的值允许是 string，但是可选属性 age 的值却是 number，number 不是 string 的子属性，所以报错了。

另外，在报错信息中可以看出，此时 `{ name: 'Tom', age: 25, gender: 'male' }` 的类型被推断成了 `{ [x: string]: string | number name: string age: number gender: string }`，这是联合类型和接口的结合。

只读属性

只读属性

有时候我们希望对象中的一些字段只能在创建的时候被赋值，那么可以用 `readonly` 定义只读属性：

```
interface Person {
 readonly id: number
 name: string
 age?: number
 [propName: string]: any
}

let tom: Person = {
 id: 89757,
 name: 'Tom',
 gender: 'male'
}

tom.id = 9527
// Cannot assign to 'id' because it is a read-only property.
```

上例中，使用 `readonly` 定义的属性 `id` 初始化后，又被赋值了，所以报错了。

注意，只读的约束存在于第一次给对象赋值的时候，而不是第一次给只读属性赋值的时候：

```
interface Person {
 readonly id: number
 name: string
 age?: number
 [propName: string]: any
}

let tom: Person = {
 name: 'Tom',
 gender: 'male'
}

tom.id = 89757
// Property 'id' is missing in type '{ name: string gender: string }' but required in type 'Person'.
// Cannot assign to 'id' because it is a read-only property.
```

上例中，报错信息有两处，第一处是在对 tom 进行赋值的时候，没有给 id 赋值。

第二处是在给 tom.id 赋值的时候，由于它是只读属性，所以报错了。

类

传统方法中，JavaScript 通过构造函数实现类的概念，通过原型链实现继承。而在 ES6 中，我们终于迎来了 `class` 。

TypeScript 除了实现了所有 ES6 中的类的功能以外，还添加了一些新的用法。

这一节主要介绍类的用法，下一节再介绍如何定义类的类型。

类的概念

虽然 JavaScript 中有类的概念，但是可能大多数 JavaScript 程序员并不是非常熟悉类，这里对类相关的概念做一个简单的介绍。

- 类(Class)：定义了一件事物的抽象特点，包含它的属性和方法
- 对象（Object）：类的实例，通过 `new` 生成
- 面向对象（OOP）的三大特性：封装、继承、多态
- 封装（Encapsulation）：将对数据的操作细节隐藏起来，只暴露对外的接口。外界调用端不需要（也不可能）知道细节，就能通过对外提供的接口来访问该对象，同时也保证了外界无法任意更改对象内部的数据
- 继承（Inheritance）：子类继承父类，子类除了拥有父类的所有特性外，还有一些更具体的特性
- 多态（Polymorphism）：由继承而产生了相关的不同的类，对同一个方法可以有不同的响应。比如 `Cat` 和 `Dog` 都继承自 `Animal` ，但是分别实现了自己的 `eat` 方法。此时针对某一个实例，我们无需了解它是 `Cat` 还是 `Dog` ，就可以直接调用 `eat` 方法，程序会自动判断出来应该如何执行 `eat`
- 存取器（getter & setter）：用以改变属性的读取和赋值行为
- 修饰符（Modifiers）：修饰符是一些关键字，用于限定成员或类型的性质。比如 `public` 表示公有属性或方法
- 抽象类（Abstract Class）：抽象类是供其他类继承的基类，抽象类不允许被实例化。抽象类中的抽象方法必须在子类中被实现
- 接口（Interfaces）：不同类之间公有的属性或方法，可以抽象成一个接口。接口可以被类实现（implements）。一个类只能继承自另一个类，但是可以实现多个接口

ES6 中类的用法

下面我们先回顾一下 ES6 中类的用法。

属性和方法

使用 `class` 定义类，使用 `constructor` 定义构造函数。

通过 `new` 生成新实例的时候，会自动调用构造函数。

```
class Animal {
 constructor(public name) {
 this.name = name
 }
 sayHi() {
 return `My name is ${this.name}`
 }
}

let a = new Animal('Jack')
console.log(a.sayHi()) // My name is Jack
```

类的继承

使用 `extends` 关键字实现继承，子类中使用 `super` 关键字来调用父类的构造函数和方法。

```
class Cat extends Animal {
 constructor(name) {
 super(name) // 调用父类的 constructor(name)
 console.log(this.name)
 }
 sayHi() {
 return 'Meow, ' + super.sayHi() // 调用父类的 sayHi()
 }
}

let c = new Cat('Tom') // Tom
console.log(c.sayHi()) // Meow, My name is Tom
```

存取器

使用 getter 和 setter 可以改变属性的赋值和读取行为：

```
class Animal {
 constructor(name) {
 this.name = name
 }
 get name() {
```

```
  return 'Jack'
  }
  set name(value) {
  console.log('setter: ' + value)
  }
 }

 let a = new Animal('Kitty') // setter: Kitty
a.name = 'Tom' // setter: Tom
 console.log(a.name) // Jack
```

静态方法

使用 `static` 修饰符修饰的方法称为静态方法，它们不需要实例化，而是直接通过类来调用：

```
 class Animal {
  static isAnimal(a) {
  return a instanceof Animal
  }
 }

 let a = new Animal('Jack')
Animal.isAnimal(a) // true
 a.isAnimal(a) // TypeError: a.isAnimal is not a function
```

ES7 中类的用法

ES7 中有一些关于类的提案，TypeScript 也实现了它们，这里做一个简单的介绍。

实例属性

ES6 中实例的属性只能通过构造函数中的 `this.xxx` 来定义，ES7 提案中可以直接在类里面定义：

```
 class Animal {
  name = 'Jack'

  constructor() {
  // ...
  }
 }

```

```
let a = new Animal()
console.log(a.name) // Jack
```

静态属性

ES7 提案中，可以使用 `static` 定义一个静态属性：

```
class Animal {
 static num = 42

 constructor() {
  // ...
 }
}

console.log(Animal.num) // 42
```

TypeScript 中类的用法

public private 和 protected

TypeScript 可以使用三种访问修饰符（Access Modifiers），分别是 `public` 、 `private` 和 `protected` 。

- `public` 修饰的属性或方法是公有的，可以在任何地方被访问到，默认所有的属性和方法都是 `public` 的
- `private` 修饰的属性或方法是私有的，不能在声明它的类的外部访问
- `protected` 修饰的属性或方法是受保护的，它和 `private` 类似，区别是它在子类中也是允许被访问的

下面举一些例子：

```
class Animal {
 public name
 public constructor(name) {
 this.name = name
 }
}

let a = new Animal('Jack')
console.log(a.name) // Jack
```

```
a.name = 'Tom'
console.log(a.name) // Tom
```

上面的例子中，`name` 被设置为了 `public`，所以直接访问实例的 `name` 属性是允许的。

很多时候，我们希望有的属性是无法直接存取的，这时候就可以用 `private` 了：

```
class Animal {
 private name
 public constructor(name) {
 this.name = name
 }
}

let a = new Animal('Jack')
console.log(a.name) // Jack
a.name = 'Tom'

// Property 'name' is private and only accessible within class 'Animal'.
// Property 'name' is private and only accessible within class 'Animal'.
```

需要注意的是，TypeScript 编译之后的代码中，并没有限制 `private` 属性在外部的可访问性。

上面的例子编译后的代码是：

```
var Animal = (function () {
 function Animal(name) {
 this.name = name
 }
 return Animal
}())
var a = new Animal('Jack')
console.log(a.name)
a.name = 'Tom'
```

使用 `private` 修饰的属性或方法，在子类中也是不允许访问的：

```
class Animal {
 private name
 public constructor(name) {
 this.name = name
 }
}

class Cat extends Animal {
```

```
  constructor(name) {
  super(name)
  console.log(this.name)
  }
 }

 // Property 'name' is private and only accessible within class 'Animal'.
```

而如果是用 `protected` 修饰，则允许在子类中访问：

```
 class Animal {
  protected name
  public constructor(name) {
  this.name = name
  }
 }

 class Cat extends Animal {
  constructor(name) {
  super(name)
  console.log(this.name)
  }
 }
```

当构造函数修饰为 `private` 时，该类不允许被继承或者实例化：

```
 class Animal {
  public name
  private constructor (name) {
  this.name = name
  }
 }
 class Cat extends Animal {
  constructor (name) {
  super(name)
  }
 }

 let a = new Animal('Jack')

 // Cannot extend a class 'Animal'. Class constructor is marked as private.
 // Constructor of class 'Animal' is private and only accessible within the class declaration.
```

当构造函数修饰为 `protected` 时，该类只允许被继承：

```
class Animal {
 public name
 protected constructor (name) {
 this.name = name
 }
}
class Cat extends Animal {
 constructor (name) {
 super(name)
 }
}

let a = new Animal('Jack')

// Constructor of class 'Animal' is protected and only accessible within the class declaration.
```

修饰符还可以使用在构造函数参数中，等同于类中定义该属性，使代码更简洁。

```
class Animal {
 // public name: string
 public constructor (public name) {
 this.name = name
 }
}
```

readonly

只读属性关键字，只允许出现在属性声明或索引签名中。

```
class Animal {
 readonly name
 public constructor(name) {
 this.name = name
 }
}

let a = new Animal('Jack')
console.log(a.name) // Jack
a.name = 'Tom'

```

```
// Cannot assign to 'name' because it is a read-only property.
```

注意如果 `readonly` 和其他访问修饰符同时存在的话，需要写在其后面。

```
class Animal {
 // public readonly name
 public constructor(public readonly name) {
 this.name = name
 }
}
```

抽象类

`abstract` 用于定义抽象类和其中的抽象方法。

什么是抽象类？

首先，抽象类是不允许被实例化的：

```
abstract class Animal {
 public name
 public constructor(name) {
 this.name = name
 }
 public abstract sayHi()
}

let a = new Animal('Jack')

// Cannot create an instance of an abstract class.
```

上面的例子中，我们定义了一个抽象类 `Animal` ，并且定义了一个抽象方法 `sayHi` 。在实例化抽象类的时候报错了。

其次，抽象类中的抽象方法必须被子类实现：

```
abstract class Animal {
 public name
 public constructor(name) {
 this.name = name
 }
 public abstract sayHi()
}

```

```
class Cat extends Animal {
 public eat() {
  console.log(`${this.name} is eating.`)
 }
}

let cat = new Cat('Tom')

// Non-abstract class 'Cat' does not implement inherited abstract member 'sayHi' from class 'Animal'.
```

上面的例子中，我们定义了一个类 `Cat` 继承了抽象类 `Animal` ，但是没有实现抽象方法 `sayHi` ，所以编译报错了。

下面是一个正确使用抽象类的例子：

```
abstract class Animal {
 public name
 public constructor(name) {
  this.name = name
 }
 public abstract sayHi()
}

class Cat extends Animal {
 public sayHi() {
  console.log(`Meow, My name is ${this.name}`)
 }
}

let cat = new Cat('Tom')
```

上面的例子中，我们实现了抽象方法 `sayHi` ，编译通过了。

需要注意的是，即使是抽象方法，TypeScript 的编译结果中，仍然会存在这个类，上面的代码的编译结果是：

```
var __extends = (this && this.__extends) || function (d, b) {
 for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]
 function __() { this.constructor = d }
 d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __())
}
var Animal = (function () {
 function Animal(name) {
```

```
  this.name = name
  }
  return Animal
 }())
var Cat = (function (_super) {
  __extends(Cat, _super)
  function Cat() {
  _super.apply(this, arguments)
  }
  Cat.prototype.sayHi = function () {
  console.log('Meow, My name is ' + this.name)
  }
  return Cat
 }(Animal))
var cat = new Cat('Tom')
```

类的类型

给类加上 TypeScript 的类型很简单，与接口类似：

```
class Animal {
  name: string
  constructor(name: string) {
  this.name = name
  }
  sayHi(): string {
  return `My name is ${this.name}`
  }
 }

 let a: Animal = new Animal('Jack')
 console.log(a.sayHi()) // My name is Jack
```

类与接口

之前学习过，接口（Interfaces）可以用于对「对象的形状（Shape）」进行描述。

这一章主要介绍接口的另一个用途，对类的一部分行为进行抽象。

类实现接口

实现（implements）是面向对象中的一个重要概念。一般来讲，一个类只能继承自另一个类，有时候不同类之间可以有一些共有的特性，这时候就可以把特性提取成接口（interfaces），用 `implements` 关键字来实现。这个特性大大提高了面向对象的灵活性。

举例来说，门是一个类，防盗门是门的子类。如果防盗门有一个报警器的功能，我们可以简单的给防盗门添加一个报警方法。这时候如果有另一个类，车，也有报警器的功能，就可以考虑把报警器提取出来，作为一个接口，防盗门和车都去实现它：

```
interface Alarm {
 alert()
}

class Door {
}

class SecurityDoor extends Door implements Alarm {
 alert() {
 console.log('SecurityDoor alert')
 }
}

class Car implements Alarm {
 alert() {
 console.log('Car alert')
 }
}
```

一个类可以实现多个接口：

```
interface Alarm {
 alert()
}

interface Light {
 lightOn()
 lightOff()
}

class Car implements Alarm, Light {
 alert() {
```

```
12.   console.log('Car alert')
13.   }
14.   lightOn() {
15.   console.log('Car light on')
16.   }
17.   lightOff() {
18.   console.log('Car light off')
19.   }
20.  }
```

上例中，`Car` 实现了 `Alarm` 和 `Light` 接口，既能报警，也能开关车灯。

接口继承接口

接口与接口之间可以是继承关系：

```
1.  interface Alarm {
2.   alert()
3.  }
4.
5.  interface LightableAlarm extends Alarm {
6.   lightOn()
7.   lightOff()
8.  }
```

上例中，我们使用 `extends` 使 `LightableAlarm` 继承 `Alarm` 。

接口继承类

接口也可以继承类：

```
1.  class Point {
2.   x: number
3.   y: number
4.  }
5.
6.  interface Point3d extends Point {
7.   z: number
8.  }
9.
10. let point3d: Point3d = {x: 1, y: 2, z: 3}
```

混合类型

混合类型

之前学习过，可以使用接口的方式来定义一个函数需要符合的形状：

```
interface SearchFunc {
 (source: string, subString: string): boolean
}

let mySearch: SearchFunc
mySearch = function(source: string, subString: string) {
 return source.search(subString) !== -1
}
```

泛型

泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

简单的例子

首先，我们来实现一个函数 `createArray` ，它可以创建一个指定长度的数组，同时将每一项都填充一个默认值：

```
function createArray(length: number, value: any): Array<any> {
 let result = []
 for (let i = 0; i < length; i++) {
 result[i] = value
 }
 return result
}

createArray(3, 'x'); // ['x', 'x', 'x']
```

上例中，我们使用了之前提到过的数组泛型来定义返回值的类型。

这段代码编译不会报错，但是一个显而易见的缺陷是，它并没有准确的定义返回值的类型：

`Array<any>` 允许数组的每一项都为任意类型。但是我们预期的是，数组中每一项都应该是输入的 `value` 的类型。

这时候，泛型就派上用场了：

```
function createArray<T>(length: number, value: T): Array<T> {
 let result: T[] = []
 for (let i = 0; i < length; i++) {
 result[i] = value
 }
 return result
}

createArray<string>(3, 'x'); // ['x', 'x', 'x']
```

上例中，我们在函数名后添加了 `<T>` ，其中 `T` 用来指代任意输入的类型，在后面的输入 `value: T` 和输出 `Array<T>` 中即可使用了。

接着在调用的时候，可以指定它具体的类型为 `string` 。当然，也可以不手动指定，而让类型推论自动推算出来：

```
function createArray<T>(length: number, value: T): Array<T> {
 let result: T[] = []
 for (let i = 0; i < length; i++) {
 result[i] = value
 }
 return result
}

createArray(3, 'x') // ['x', 'x', 'x']
```

多个类型参数

定义泛型的时候，可以一次定义多个类型参数：

```
function swap<T, U>(tuple: [T, U]): [U, T] {
 return [tuple[1], tuple[0]]
}

swap([7, 'seven']) // ['seven', 7]
```

上例中，我们定义了一个 `swap` 函数，用来交换输入的元组。

泛型约束

在函数内部使用泛型变量的时候，由于事先不知道它是哪种类型，所以不能随意的操作它的属性或方法：

```
function loggingIdentity<T>(arg: T): T {
 console.log(arg.length)
 return arg
}

// Property 'length' does not exist on type 'T'.
```

上例中，泛型 `T` 不一定包含属性 `length` ，所以编译的时候报错了。

这时，我们可以对泛型进行约束，只允许这个函数传入那些包含 `length` 属性的变量。这就是泛型约束：

```
interface Lengthwise {
 length: number
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
 console.log(arg.length)
 return arg
}
```

上例中，我们使用了 `extends` 约束了泛型 `T` 必须符合接口 `Lengthwise` 的形状，也就是必须包含 `length` 属性。

此时如果调用 `loggingIdentity` 的时候，传入的 `arg` 不包含 `length` ，那么在编译阶段就会报错了：

```
interface Lengthwise {
 length: number
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
 console.log(arg.length)
 return arg
}

loggingIdentity(7)

// Argument of type '7' is not assignable to parameter of type 'Lengthwise'.
```

泛型接口

之前学习过接口中函数的定义，可以使用接口的方式来定义一个函数需要符合的形状：

```
interface SearchFunc {
 (source: string, subString: string): boolean
}

let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
 return source.search(subString) !== -1
}
```

当然也可以使用含有泛型的接口来定义函数的形状：

```
interface CreateArrayFunc {
 <T>(length: number, value: T): Array<T>
}

let createArray: CreateArrayFunc;
createArray = function<T>(length: number, value: T): Array<T> {
 let result: T[] = []
 for (let i = 0; i < length; i++) {
 result[i] = value
 }
 return result
}

createArray(3, 'x') // ['x', 'x', 'x']
```

进一步，我们可以把泛型参数提前到接口名上：

```
interface CreateArrayFunc<T> {
 (length: number, value: T): Array<T>
}

let createArray: CreateArrayFunc<any>
createArray = function<T>(length: number, value: T): Array<T> {
 let result: T[] = []
 for (let i = 0; i < length; i++) {
 result[i] = value
 }
 return result
}

createArray(3, 'x'); // ['x', 'x', 'x']
```

注意，此时在使用泛型接口的时候，需要定义泛型的类型。

泛型类

与泛型接口类似，泛型也可以用于类的类型定义中：

```
class GenericNumber<T> {
 zeroValue: T
 add: (x: T, y: T) => T
}

let myGenericNumber = new GenericNumber<number>()
myGenericNumber.zeroValue = 0
myGenericNumber.add = function(x, y) { return x + y }
```

此处 zeroValue，add 未赋值会出错，设置 “strictPropertyInitialization”: false, 关闭提示

高级

基础类型

基础类型-高级

类型推断

如果没有明确的指定类型，那么 TypeScript 会依照类型推论（Type Inference）的规则推断出一个类型。

什么是类型推断

以下代码虽然没有指定类型，但是会在编译的时候报错：

```
let lunarDay = '初一'
lunarDay = 1
// Type '1' is not assignable to type 'string'.
```

事实上，它等价于：

```
let lunarDay: string = '初一'
lunarDay = 1
```

TypeScript 会在没有明确的指定类型的时候推测出一个类型，这就是类型推论。

如果定义的时候没有赋值，不管之后有没有赋值，都会被推断成 any 类型而完全不被类型检查：

```
let myFavoriteNumber
myFavoriteNumber =  ‘seven’
myFavoriteNumber = 7

let someValue: any = “this is a string”

let strLength: number = (<string>someValue).length
```

另一个为as语法：

```
let someValue: any = “this is a string”

let strLength: number = (someValue as string).length
```

两种形式是等价的。 至于使用哪个大多数情况下是凭个人喜好；然而，当你在TypeScript里使用JSX时，只有 as语法断言是被允许的。

联合类型

联合类型（Union Types）表示取值可以为多种类型中的一种。

简单的例子

```
let lunarDay: string | number
lunarDay = '初一'
lunarDay = 1
```

联合类型使用 | 分隔每个类型。

这里的 `let lunarDay: string | number` 的含义是，允许 lunarDay 的类型是 string 或者 number，但是不能是其他类型。

访问联合类型的属性或方法

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们只能访问此联合类型的所有类型里共有的属性或方法：

```
function getLength(something: string | number): number {
 return something.length
}
// Property 'length' does not exist on type 'string | number'.
// Property 'length' does not exist on type 'number'.
```

上例中，length 不是 string 和 number 的共有属性，所以会报错。

访问 string 和 number 的共有属性是没问题的

访问 string 和 number 的共有属性是没问题的：

```
function getString(something: string | number): string {
 return something.toString()
}
```

联合类型赋值的类型推断

联合类型的变量在被赋值的时候，会根据类型推论的规则推断出一个类型：

```
let lunarDay: string | number
lunarDay = '初一'
console.log(lunarDay.length) // 2
lunarDay = 1
console.log(lunarDay.length) // 编译时报错
```

上例中，第二行的 lunarDay 被推断成了 string，访问它的 length 属性不会报错。
而第四行的 lunarDay 被推断成了 number，访问它的 length 属性时就报错了。

Null 和 Undefined

`null` 是一个只有一个值的特殊类型。表示一个空对象引用。用 typeof 检测 null 返回是 `object` 。
typeof 一个没有值的变量会返回 `undefined` 。

null 和 Undefined 是其他任何类型（包括 void）的子类型，可以赋值给其它类型，如数字类型，此时，赋值后的类型会变成 null 或 undefined。

在TypeScript中启用严格的空校验（—strictNullChecks）特性，使得 `null` 和 `undefined` 只能被赋值给 `void` 或本身对应的类型

在 tsconfig.json 中启用 —strictNullChecks

```
let x: number
x = 1 // 运行正确
x = undefined // 运行错误
x = null // 运行错误
```

在 tsconfig.json 中启用 —strictNullChecks，需要将x赋值为联合类型

```
let x: number | null | undefined //本身对应的类型
x = 1 // 运行正确
x = undefined // 运行正确
x = null // 运行正确
```

Never

never类型表示的是那些永不存在的值的类型。 例如， never类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型； 变量也可能是 never类型，当它们被永不为真的类型保护所约束时。

never类型是任何类型的子类型，也可以赋值给任何类型；然而，没有类型是never的子类型或可以赋值给never类型（除了never本身之外）。 即使 any也不可以赋值给never。

下面是一些返回never类型的函数：

```
// 返回never的函数必须存在无法达到的终点
function error(message: string): never {
 throw new Error(message)
}

// 推断的返回值类型为never
function fail() {
 return error("Something failed")
}

// 返回never的函数必须存在无法达到的终点
function infiniteLoop(): never {
 while (true) {
 }
}
```

Symbols

自ECMAScript 2015起，symbol成为了一种新的原生类型，就像number和string一样。
symbol类型的值是通过Symbol构造函数创建的。

```
let sym1 = Symbol();
let sym2 = Symbol("key"); // 可选的字符串key
```

Symbols是不可改变且唯一的。

```
let sym2 = Symbol("key")
let sym3 = Symbol("key")

sym2 === sym3 // false, symbols是唯一的
```

像字符串一样，symbols也可以被用做对象属性的键。

```
let sym = Symbol()

let obj = {
 [sym]: "value"
}

console.log(obj[sym]) // "value"
```

Symbols也可以与计算出的属性名声明相结合来声明对象的属性和类成员。

```
const getClassNameSymbol = Symbol()

class C {
 [getClassNameSymbol](){
  return "C"
 }
}

let c = new C()
let className = c[getClassNameSymbol]() // "C"
```

迭代器和生成器

可迭代性

for..of 语句

for..of会遍历可迭代的对象，调用对象上的Symbol.iterator方法。 下面是在数组上使用 for..of的简单例子：

```
let someArray = [1, "string", false]

for (let entry of someArray) {
 console.log(entry) // 1, "string", false
}
```

for..of vs. for..in 语句

for..of和for..in均可迭代一个列表；但是用于迭代的值却不同，for..in迭代的是对象的键的列表，而for..of则迭代对象的键对应的值。

下面的例子展示了两者之间的区别：

```
let list = [4, 5, 6]

for (let i in list) {
 console.log(i) // "0", "1", "2",
}

for (let i of list) {
 console.log(i) // "4", "5", "6"
}
```

另一个区别是for..in可以操作任何对象，它提供了查看对象属性的一种方法。 但是 for..of关注于迭代对象的值。内置对象Map和Set已经实现了Symbol.iterator方法，让我们可以访问它们保存的值。

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals"

for (let pet in pets) {
 console.log(pet) // "species"
}

for (let pet of pets) {
 console.log(pet) // "Cat", "Dog", "Hamster"
}
```

装饰器

介绍

> *装饰器是一种特殊类型的声明，它能够附加到类、类的函数、类属性、类函数的参数上，以达到修改类的行为。*

装饰器的种类

根据装饰器的位置

- 类装饰器

- 类装饰器
- 类函数装饰器
- 类属性装饰器
- 类函数参数装饰器

根据装饰器是否有参数

- 无参装饰器(一般装饰器)
- 有参装饰器(装饰器工厂)

类的装饰器

- 类装饰器的写法

```
function desc(target) {
 console.log(target) // 输出 [Function: Person]表示当前装饰的类
}

@desc
class Person {
 public name: string | undefined
 public age: number | 0

 constructor(name: string, age: number) {
 this.name = name
 this.age = age
 }
}
```

此处 target 类型会隐式定义为any, 这样会引发一个TS问题，解决方案：设置tsconfig.json

1、”noImplicitAny”: false

或者

2、”strict”: false

- 使用类的装饰器扩展类的属性和方法

```
interface Class {
new(…args: any[]): {}
}

function desc<T extends Class>(Target: T) {
console.log(Target)
```

```
return class extends Target {
gender = '男'
say() {
console.log(this.gender)
}
}
}

@desc
class Person {
public name: string | undefined
public age: number | 0

constructor(name, age) {
this.name = name
this.age = age
}

say() {
console.log(this.name, this.age)
}
}

let p = new Person('Felix', 20)
console.log(p)
p.say()
```

1. * 使用装饰器修改类的构造函数(构造函数的重载、方法重载)

```
function desc(target) {
return class extends target{
name = 'Felixlu'
age = 18
sayHell() {
console.log('我是重载后的', this.name)
}
}
}

@desc
class Person {
public name: string | undefined
public age: number | 0
```

```
constructor() {
this.name = ‘Felix’
this.age = 20
}

sayHell() {
console.log(‘hello word’, this.name)
}
}

let p = new Person()
console.log(p)
p.sayHell()
```

1. * 装饰器工厂的写法

```
function desc(params: string) {
return function (target: any) {
console.log(‘params’, params)
console.log(‘target’, target)
// 直接在原型上扩展一个属性
target.prototype.apiUrl = params
}
}

@desc(‘好程序员’)
class P {
say() {
console.log(‘说话’)
}
}

let p: any = new P()
console.log(p.apiUrl)
```

1. ### 类函数装饰器
2.
3. > 它应用到方法上，可以用来监视、修改、替换该方法
4.
5. * 基本使用

```
function desc(target, key, descriptor) {
console.log(‘target’, target) // Person { say: [Function] } 表示类的原型
console.log(‘key’, key) // 被装饰的函数名
```

```
console.log(‘descriptor’, descriptor) // 被装饰的函数的对象属性
}

class Person {
public name: string | undefined
public age: number | 0

constructor(name, age) {
this.name = name
this.age = age
}

@desc
say() {
console.log(‘说的方法’)
}
}
```

1. * 在装饰器中添加类的原型属性和原型方法

```
function desc(target, key, descriptor) {
target.gender = ‘男’
target.foo = function () {
console.log(‘我是原型上的方法’)
}
}

class Person {
public name: string | undefined
public age: number | 0

constructor(name, age) {
this.name = name
this.age = age
}

@desc
say() {
console.log(‘说的方法’)
}
}

// 测试代码
let p: any = new Person(‘Felixlu’, 20)
```

```
console.log(p)
console.log(Person.prototype)
p.say()
console.log(p.gender); // 使用p原型链上的属性
p.foo() // 调用了p原型链上的方法

1.  * 使用装饰器拦截函数的调用（替换）
function desc(params: string) {
return function (target: any, key: string, descriptor: {[propsName: string]: any}) {
// 修改被装饰的函数
descriptor.value = function (…args: Array<any>) {
args = args.map(it => String(it))
console.log(args)
}
}
}

class Person {
public name: string | undefined
public age: number | 0

constructor(name, age) {
this.name = name
this.age = age
}

@desc(‘装饰器上的参数’)
say() {
console.log(‘说的方法’)
}
}

let p: any = new Person(‘Felixlu’, 20)
console.log(p)
p.say(123, 23, ‘你好’)

1.  * 使用装饰器拦截函数的调用(附加新的功能)
function desc(params: string) {
return function (target: any, key: string, descriptor: {[propsName: string]: any}) {
// 修改被装饰的函数的
let method = descriptor.value
descriptor.value = function (…args: Array<any>) {
args = args.map(it => String(it))
```

```
args = args.map(it => String(it))
console.log(args)
method.apply(this, args)
}
}
}
class Person {
public name: string | undefined
public age: number | 0

constructor(name, age) {
this.name = name
this.age = age
}

@desc('装饰器上的参数')
say(...args) {
console.log('说的方法', args)
}
}

let p = new Person('Felixlu', 20)
console.log(p)
p.say(123, 23, '你好')
```

```
### 类属性装饰器

* 基本用法
function desc(target, name) {
console.log('target', target, target.constructor) // 表示类的原型
console.log('name', name) // 表示被装饰属性名
}

class Person {
public name: string | undefined
public age: number | 0

@desc
private gender: string | undefined

constructor(name, age) {
this.name = name
this.age = age
```

```
}
}

let p = new Person(‘Felixlu’, 20)
console.log(p)

 1.  * 在装饰器中修改属性值
function desc(target, name) {
target[name] = ‘女’
}

class Person {
public name: string | undefined
public age: number | 0

@desc
public gender: string | undefined

constructor(name, age) {
this.name = name
this.age = age
}

say() {
console.log(this.name, this.age, this.gender)
}
}

let p = new Person(‘Felixlu’, 20)
console.log(p)
p.say()

 1.  * 类函数参数的装饰器
 2.
 3.  > 参数装饰器表达式会在运行时候当做函数被调用，以使用参数装饰器为类的原型上附加一些元数据
 4.
 5.  * 基本用法
function desc(params: string) {
return function (target: any, key, index) {
console.log(target); // 类的原型
console.log(key); // 被装饰的名字
console.log(index); // 序列化
`
```

```
}
}
class Person {
public name: string | undefined
public age: number | 0

constructor(name, age) {
this.name = name
this.age = age
}

say(@desc(‘参数装饰器’) age: number) {
console.log(‘说的方法’)
}
}

let p = new Person(‘Felixlu’, 20)
console.log(p)
p.say(20)
```

1. * 为类的原型上添加一些东西

```
function desc(params: string) {
return function (target: any, key, index) {
console.log(target); // 类的原型
console.log(key); // 被装饰的名字
console.log(index); // 序列化
target.message = params;
}
}
class Person {
public name: string | undefined;
public age: number | 0;

constructor(name, age) {
this.name = name;
this.age = age;
}

say(@desc(‘参数装饰器’) age: number) {
console.log(‘说的方法’)
}
}
```

```
let p: any = new Person(‘哈哈’, 20);
console.log(p);
p.say(20);
console.log(p.message)
```

1. ### 几种装饰器的执行顺序

```
function logCls(params: string) {
return function (target: any) {
console.log(‘4.类的装饰器’);
}
}

function logMethod(params: string) {
return function (target: any, key: string, descriptor: {[propsName: string]: any}) {
console.log(‘3.类的函数装饰器’);
}
}

function logParams(params: string) {
return function (target: any, name: string) {
console.log(‘1.类属性装饰器’);
}
}

function logQuery(params: string) {
return function (target: any, key: string, index: number) {
console.log(‘2.函数参数装饰器’);
}
}

@logCls(‘类的装饰器’)
class Person{
@logParams(‘属性装饰器’)
public name: string | undefined;

@logMehod(‘函数装饰器’)
getData(@logQuery(‘函数参数装饰器’) age: number, @logQuery(‘函数参数装饰器’) gender:
string) {
console.log(‘—— ‘);
}
}
```

1

```
2.
```

声明文件

当使用第三方库时，我们需要引用它的声明文件，才能获得对应的代码补全、接口提示等功能。

新语法索引

由于本章涉及大量新语法，故在本章开头列出新语法的索引，方便大家在使用这些新语法时能快速查找到对应的讲解：

- `declare var` 声明全局变量
- `declare function` 声明全局方法
- `declare class` 声明全局类
- `declare enum` 声明全局枚举类型
- `declare namespace` 声明（含有子属性的）全局对象
- `interface` 和 `type` 声明全局类型
- `export` 导出变量
- `export namespace` 导出（含有子属性的）对象
- `export default` ES6 默认导出
- `export =` commonjs 导出模块
- `export as namespace` UMD 库声明全局变量
- `declare global` 扩展全局变量
- `declare module` 扩展模块
- `/// <reference />` 三斜线指令

什么是声明语句

假如我们想使用第三方库 jQuery，一种常见的方式是在 html 中通过 `<script>` 标签引入 jQuery，然后就可以使用全局变量 `$` 或 `jQuery` 了。

我们通常这样获取一个 `id` 是 `foo` 的元素：

```
1.  $('#foo');
2.  // or
3.  jQuery('#foo');
```

但是在 ts 中，编译器并不知道 `$` 或 `jQuery` 是什么东西[1]：

```
1.  jQuery('#foo');
```

```
// ERROR: Cannot find name 'jQuery'.
```

这时，我们需要使用 `declare var` 来定义它的类型[2]：

```
declare var jQuery: (selector: string) => any;

jQuery('#foo');
```

上例中，`declare var` 并没有真的定义一个变量，只是定义了全局变量 `jQuery` 的类型，仅仅会用于编译时的检查，在编译结果中会被删除。它编译结果是：

```
jQuery('#foo');
```

除了 `declare var` 之外，还有其他很多种声明语句，将会在后面详细介绍。

什么是声明文件

通常我们会把声明语句放到一个单独的文件（`jQuery.d.ts`）中，这就是声明文件[3]：

```
// src/jQuery.d.ts

declare var jQuery: (selector: string) => any;
```

```
// src/index.ts

jQuery('#foo');
```

声明文件必需以 `.d.ts` 为后缀。

一般来说，ts 会解析项目中所有的 `*.ts` 文件，当然也包含以 `.d.ts` 结尾的文件。所以当我们将 `jQuery.d.ts` 放到项目中时，其他所有 `*.ts` 文件就都可以获得 `jQuery` 的类型定义了。

```
/path/to/project
├── src
|   ├── index.ts
|   └── jQuery.d.ts
└── tsconfig.json
```

假如仍然无法解析，那么可以检查下 `tsconfig.json` 中的 `files`、`include` 和 `exclude` 配置，确保其包含了 `jQuery.d.ts` 文件。

这里只演示了全局变量这种模式的声明文件，假如是通过模块导入的方式使用第三方库的话，那么引入声明文件又是另一种方式了，将会在后面详细介绍。

第三方声明文件

当然，jQuery 的声明文件不需要我们定义了，社区已经帮我们定义好了：jQuery in DefinitelyTyped。

我们可以直接下载下来使用，但是更推荐的是使用 `@types` 统一管理第三方库的声明文件。

`@types` 的使用方式很简单，直接用 npm 安装对应的声明模块即可，以 jQuery 举例：

```
npm install @types/jquery --save-dev
```

可以在这个页面搜索你需要的声明文件。

书写声明文件

当一个第三方库没有提供声明文件时，我们就需要自己书写声明文件了。前面只介绍了最简单的声明文件内容，而真正书写一个声明文件并不是一件简单的事，以下会详细介绍如何书写声明文件。

在不同的场景下，声明文件的内容和使用方式会有所区别。

库的使用场景主要有以下几种：

- 全局变量：通过 `<script>` 标签引入第三方库，注入全局变量
- npm 包：通过 `import foo from 'foo'` 导入，符合 ES6 模块规范
- UMD 库：既可以通过 `<script>` 标签引入，又可以通过 `import` 导入
- 直接扩展全局变量：通过 `<script>` 标签引入后，改变一个全局变量的结构
- 在 npm 包或 UMD 库中扩展全局变量：引用 npm 包或 UMD 库后，改变一个全局变量的结构
- 模块插件：通过 `<script>` 或 `import` 导入后，改变另一个模块的结构

全局变量

全局变量是最简单的一种场景，之前举的例子就是通过 `<script>` 标签引入 jQuery，注入全局变量 `$` 和 `jQuery`。

使用全局变量的声明文件时，如果是以 `npm install @types/xxx --save-dev` 安装的，则不需要任何配置。如果是将声明文件直接存放于当前项目中，则建议和其他源码一起放到 `src` 目录下（或者对应的源码目录下）：

```
/path/to/project
├── src
|  ├── index.ts
|  └── jQuery.d.ts
└── tsconfig.json
```

如果没有生效，可以检查下 `tsconfig.json` 中的 `files`、`include` 和 `exclude` 配置，确保其包含了 `jQuery.d.ts` 文件。

全局变量的声明文件主要有以下几种语法：

- `declare var` 声明全局变量

- `declare var` 声明全局变量
- `declare function` 声明全局方法
- `declare class` 声明全局类
- `declare enum` 声明全局枚举类型
- `declare namespace` 声明（含有子属性的）全局对象
- `interface` 和 `type` 声明全局类型

`declare var`

在所有的声明语句中，`declare var` 是最简单的，如之前所学，它能够用来定义一个全局变量的类型。与其类似的，还有 `declare let` 和 `declare const`，使用 `let` 与使用 `var` 没有什么区别：

```
// src/jQuery.d.ts

declare let jQuery: (selector: string) => any;
```

```
// src/index.ts

jQuery('#foo');
// 使用 declare let 定义的 jQuery 类型，允许修改这个全局变量
jQuery = function(selector) {
 return document.querySelector(selector);
};
```

而当我们使用 `const` 定义时，表示此时的全局变量是一个常量，不允许再去修改它的值了[4]：

```
// src/jQuery.d.ts

declare const jQuery: (selector: string) => any;

jQuery('#foo');
// 使用 declare const 定义的 jQuery 类型，禁止修改这个全局变量
jQuery = function(selector) {
 return document.querySelector(selector);
};
// ERROR: Cannot assign to 'jQuery' because it is a constant or a read-only property.
```

一般来说，全局变量都是禁止修改的常量，所以大部分情况都应该使用 `const` 而不是 `var` 或 `let`。

需要注意的是，声明语句中只能定义类型，切勿在声明语句中定义具体的实现[5]：

```
declare const jQuery = function(selector) {
 return document.querySelector(selector);
}
```

```
};
// ERROR: An implementation cannot be declared in ambient contexts.
```

declare function

`declare function` 用来定义全局函数的类型。jQuery 其实就是一个函数，所以也可以用 `function` 来定义：

```
// src/jQuery.d.ts

declare function jQuery(selector: string): any;
```

```
// src/index.ts

jQuery('#foo');
```

在函数类型的声明语句中，函数重载也是支持的[6]：

```
// src/jQuery.d.ts

declare function jQuery(selector: string): any;
declare function jQuery(domReadyCallback: () => any): any;
```

```
// src/index.ts

jQuery('#foo');
jQuery(function() {
  alert('Dom Ready!');
});
```

declare class

当全局变量是一个类的时候，我们用 `declare class` 来定义它的类型[7]：

```
// src/Animal.d.ts

declare class Animal {
  name: string;
  constructor(name: string);
  sayHi(): string;
}
```

```
// src/index.ts

let cat = new Animal('Tom');
```

同样的，`declare class` 语句也只能用来定义类型，不能用来定义具体的实现，比如定义 `sayHi` 方法的具体实现则会报错：

方法的具体实现则会报错：

```
// src/Animal.d.ts

declare class Animal {
 name: string;
 constructor(name: string);
 sayHi() {
 return `My name is ${this.name}`;
 };
 // ERROR: An implementation cannot be declared in ambient contexts.
}
```

declare enum

使用 `declare enum` 定义的枚举类型也称作外部枚举（Ambient Enums），举例如下[8]：

```
// src/Directions.d.ts

declare enum Directions {
 Up,
 Down,
 Left,
 Right
}
```

```
// src/index.ts

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
```

与其他全局变量的类型声明一致，`declare enum` 仅用来定义类型，而不是具体的值。

`Directions.d.ts` 仅仅会用于编译时的检查，声明文件里的内容在编译结果中会被删除。它编译结果是：

```
var directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
```

其中 `Directions` 是由第三方库定义好的全局变量。

declare namespace

`namespace` 是 ts 早期时为了解决模块化而创造的关键字，中文称为命名空间。

由于历史遗留原因，在早期还没有 ES6 的时候，ts 提供了一种模块化方案，使用 `module` 关键字表示内部模块。但由于后来 ES6 也使用了 `module` 关键字，ts 为了兼容 ES6，使用 `namespace` 替代了自己的 `module`，更名为命名空间。

随着 ES6 的广泛应用，现在已经不建议再使用 ts 中的 `namespace`，而推荐使用 ES6 的模块化方案了，故我们不再需要学习 `namespace` 的使用了。

`namespace` 被淘汰了，但是在声明文件中，`declare namespace` 还是比较常用的，它用来表示全局变量是一个对象，包含很多子属性。

比如 `jQuery` 是一个全局变量，它是一个对象，提供了一个 `jQuery.ajax` 方法可以调用，那么我们就应该使用 `declare namespace jQuery` 来声明这个拥有多个子属性的全局变量。

```
// src/jQuery.d.ts

declare namespace jQuery {
 function ajax(url: string, settings?: any): void;
}
```

```
// src/index.ts

jQuery.ajax('/api/get_something');
```

注意，在 `declare namespace` 内部，我们直接使用 `function ajax` 来声明函数，而不是使用 `declare function ajax`。类似的，也可以使用 `const`，`class`，`enum` 等语句[9]：

```
// src/jQuery.d.ts

declare namespace jQuery {
 function ajax(url: string, settings?: any): void;
 const version: number;
 class Event {
 blur(eventType: EventType): void
 }
 enum EventType {
 CustomClick
 }
}
```

```
// src/index.ts

jQuery.ajax('/api/get_something');
console.log(jQuery.version);
const e = new jQuery.Event();
e.blur(jQuery.EventType.CustomClick);
```

嵌套的命名空间

如果对象拥有深层的层级，则需要用嵌套的 `namespace` 来声明深层的属性的类型[10]：

```
// src/jQuery.d.ts

declare namespace jQuery {
 function ajax(url: string, settings?: any): void;
 namespace fn {
 function extend(object: any): void;
 }
}
// src/index.ts

jQuery.ajax('/api/get_something');
jQuery.fn.extend({
 check: function() {
 return this.each(function() {
 this.checked = true;
 });
 }
});
```

假如 `jQuery` 下仅有 `fn` 这一个属性（没有 `ajax` 等其他属性或方法），则可以不需要嵌套 `namespace` [11]：

```
// src/jQuery.d.ts

declare namespace jQuery.fn {
 function extend(object: any): void;
}
// src/index.ts

jQuery.fn.extend({
 check: function() {
 return this.each(function() {
 this.checked = true;
 });
 }
});
```

`interface` 和 `type`

除了全局变量之外，可能有一些类型我们也希望能暴露出来。在类型声明文件中，我们可以直接使用 `interface` 或 `type` 来声明一个全局的接口或类型[12]：

```
// src/jQuery.d.ts

interface AjaxSettings {
 method?: 'GET' | 'POST'
 data?: any;
}
declare namespace jQuery {
 function ajax(url: string, settings?: AjaxSettings): void;
}
```

这样的话，在其他文件中也可以使用这个接口或类型了：

```
// src/index.ts

let settings: AjaxSettings = {
 method: 'POST',
 data: {
 name: 'foo'
 }
};
jQuery.ajax('/api/post_something', settings);
```

`type` 与 `interface` 类似，不再赘述。

防止命名冲突

暴露在最外层的 `interface` 或 `type` 会作为全局类型作用于整个项目中，我们应该尽可能的减少全局变量或全局类型的数量。故最好将他们放到 `namespace` 下[13]：

```
// src/jQuery.d.ts

declare namespace jQuery {
 interface AjaxSettings {
 method?: 'GET' | 'POST'
 data?: any;
 }
 function ajax(url: string, settings?: AjaxSettings): void;
}
```

注意，在使用这个 `interface` 的时候，也应该加上 `jQuery` 前缀：

```
// src/index.ts

let settings: jQuery.AjaxSettings = {
 method: 'POST'
```

```
  method: 'POST',
  data: {
  name: 'foo'
  }
};
jQuery.ajax('/api/post_something', settings);
```

声明合并

假如 jQuery 既是一个函数，可以直接被调用 `jQuery('#foo')` ，又是一个对象，拥有子属性 `jQuery.ajax()` （事实确实如此），那么我们可以组合多个声明语句，它们会不冲突的合并起来[14]：

```
// src/jQuery.d.ts

declare function jQuery(selector: string): any;
declare namespace jQuery {
  function ajax(url: string, settings?: any): void;
}
```

```
// src/index.ts

jQuery('#foo');
jQuery.ajax('/api/get_something');
```

关于声明合并的更多用法，可以查看声明合并章节。

npm 包

一般我们通过 `import foo from 'foo'` 导入一个 npm 包，这是符合 ES6 模块规范的。

在我们尝试给一个 npm 包创建声明文件之前，需要先看看它的声明文件是否已经存在。一般来说，npm 包的声明文件可能存在于两个地方：

1. 与该 npm 包绑定在一起。判断依据是 `package.json` 中有 `types` 字段，或者有一个 `index.d.ts` 声明文件。这种模式不需要额外安装其他包，是最为推荐的，所以以后我们自己创建 npm 包的时候，最好也将声明文件与 npm 包绑定在一起。
2. 发布到 `@types` 里。我们只需要尝试安装一下对应的 `@types` 包就知道是否存在该声明文件，安装命令是 `npm install @types/foo --save-dev` 。这种模式一般是由于 npm 包的维护者没有提供声明文件，所以只能由其他人将声明文件发布到 `@types` 里了。

假如以上两种方式都没有找到对应的声明文件，那么我们就需要自己为它写声明文件了。由于是通过 `import` 语句导入的模块，所以声明文件存放的位置也有所约束，一般有两种方案：

1. 创建一个 `node_modules/@types/foo/index.d.ts` 文件，存放 `foo` 模块的声明文件。这种方式不需要额外的配置，但是 `node_modules` 目录不稳定，代码也没有被保存到仓库中，无法回溯版

本，有不小心被删除的风险，故不太建议用这种方案，一般只用作临时测试。

2. 创建一个 `types` 目录，专门用来管理自己写的声明文件，将 `foo` 的声明文件放到 `types/foo/index.d.ts` 中。这种方式需要配置下 `tsconfig.json` 中的 `paths` 和 `baseUrl` 字段。

目录结构：

```
/path/to/project
├─── src
|  └─── index.ts
├─── types
|  └─── foo
|  └─── index.d.ts
└─── tsconfig.json
```

`tsconfig.json` 内容：

```
{
 "compilerOptions": {
 "module": "commonjs",
 "baseUrl": "./",
 "paths": {
 "*": ["types/*"]
  }
 }
}
```

如此配置之后，通过 `import` 导入 `foo` 的时候，也会去 `types` 目录下寻找对应的模块的声明文件了。

注意 `module` 配置可以有很多种选项，不同的选项会影响模块的导入导出模式。这里我们使用了 `commonjs` 这个最常用的选项，后面的教程也都默认使用的这个选项。

不管采用了以上两种方式中的哪一种，我都**强烈建议**大家将书写好的声明文件（通过给第三方库发 pull request，或者直接提交到 `@types` 里）发布到开源社区中，享受了这么多社区的优秀的资源，就应该在力所能及的时候给出一些回馈。只有所有人都参与进来，才能让 ts 社区更加繁荣。

npm 包的声明文件主要有以下几种语法：

- `export` 导出变量
- `export namespace` 导出（含有子属性的）对象
- `export default` ES6 默认导出
- `export =` commonjs 导出模块

`export`

npm 包的声明文件与全局变量的声明文件有很大区别。在 npm 包的声明文件中，使用 `declare` 不再会声明一个全局变量，而只会在当前文件中声明一个局部变量。只有在声明文件中使用 `export` 导出，然后在使用方 `import` 导入后，才会应用到这些类型声明。

`export` 的语法与普通的 ts 中的语法类似，区别仅在于声明文件中禁止定义具体的实现[15]：

```
// types/foo/index.d.ts

export const name: string;
export function getName(): string;
export class Animal {
 constructor(name: string);
 sayHi(): string;
}
export enum Directions {
 Up,
 Down,
 Left,
 Right
}
export interface Options {
 data: any;
}
```

对应的导入和使用模块应该是这样：

```
// src/index.ts

import { name, getName, Animal, Directions, Options } from 'foo';

console.log(name);
let myName = getName();
let cat = new Animal('Tom');
let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
let options: Options = {
 data: {
 name: 'foo'
 }
};
```

混用 `declare` 和 `export`

我们也可以使用 `declare` 先声明多个变量，最后再用 `export` 一次性导出。上例的声明文件可以等价的改写为[16]：

```
// types/foo/index.d.ts

declare const name: string;
declare function getName(): string;
declare class Animal {
 constructor(name: string);
 sayHi(): string;
}
declare enum Directions {
 Up,
 Down,
 Left,
 Right
}
interface Options {
 data: any;
}

export { name, getName, Animal, Directions, Options };
```

注意，与全局变量的声明文件类似，`interface` 前是不需要 `declare` 的。

`export namespace`

与 `declare namespace` 类似，`export namespace` 用来导出一个拥有子属性的对象[17]：

```
// types/foo/index.d.ts

export namespace foo {
 const name: string;
 namespace bar {
 function baz(): string;
 }
}
```

```
// src/index.ts

import { foo } from 'foo';

console.log(foo.name);
foo.bar.baz();
```

```
foo.bar.baz();
```

`export default`

在 ES6 模块系统中，使用 `export default` 可以导出一个默认值，使用方可以用 `import foo from 'foo'` 而不是 `import { foo } from 'foo'` 来导入这个默认值。

在类型声明文件中，`export default` 用来导出默认值的类型[18]：

```
// types/foo/index.d.ts

export default function foo(): string;
// src/index.ts

import foo from 'foo';

foo();
```

注意，只有 `function` 、 `class` 和 `interface` 可以直接默认导出，其他的变量需要先定义出来，再默认导出[19]：

```
// types/foo/index.d.ts

export default enum Directions {
// ERROR: Expression expected.
  Up,
  Down,
  Left,
  Right
}
```

上例中 `export default enum` 是错误的语法，需要使用 `declare enum` 定义出来，然后使用 `export default` 导出：

```
// types/foo/index.d.ts

declare enum Directions {
  Up,
  Down,
  Left,
  Right
}

export default Directions;
```

针对这种默认导出，我们一般会将导出语句放在整个声明文件的最前面[20]

针对这种默认导出，我们一般会将导出语句放在整个声明文件的最前面[20]：

```
// types/foo/index.d.ts

export default Directions;

declare enum Directions {
  Up,
  Down,
  Left,
  Right
}
```

`export =`

在 commonjs 规范中，我们用以下方式来导出一个模块：

```
// 整体导出
module.exports = foo;
// 单个导出
exports.bar = bar;
```

在 ts 中，针对这种模块导出，有多种方式可以导入，第一种方式是 `const ... = require`：

```
// 整体导入
const foo = require('foo');
// 单个导入
const bar = require('foo').bar;
```

第二种方式是 `import ... from`，注意针对整体导出，需要使用 `import * as` 来导入：

```
// 整体导入
import * as foo from 'foo';
// 单个导入
import { bar } from 'foo';
```

第三种方式是 `import ... require`，这也是 ts 官方推荐的方式：

```
// 整体导入
import foo = require('foo');
// 单个导入
import bar = foo.bar;
```

对于这种使用 commonjs 规范的库，假如要为它写类型声明文件的话，就需要使用到 `export =` 这种语法了[21]：

```
// types/foo/index.d.ts
```

```
2.
3.  export = foo;
4.
5.  declare function foo(): string;
6.  declare namespace foo {
7.   const bar: number;
8.  }
```

需要注意的是，上例中使用了 `export =` 之后，就不能再单个导出 `export { bar }` 了。所以我们通过声明合并，使用 `declare namespace foo` 来将 `bar` 合并到 `foo` 里。

准确地讲，`export =` 不仅可以用在声明文件中，也可以用在普通的 ts 文件中。实际上，`import ... require` 和 `export =` 都是 ts 为了兼容 AMD 规范和 commonjs 规范而创立的新语法，由于并不常用也不推荐使用，所以这里就不详细介绍了，感兴趣的可以看官方文档。

由于很多第三方库是 commonjs 规范的，所以声明文件也就不得不用到 `export =` 这种语法了。但是还是需要再强调下，相比与 `export =`，我们更推荐使用 ES6 标准的 `export default` 和 `export`。

UMD 库

既可以通过 `<script>` 标签引入，又可以通过 `import` 导入的库，称为 UMD 库。相比于 npm 包的类型声明文件，我们需要额外声明一个全局变量，为了实现这种方式，ts 提供了一个新语法 `export as namespace`。

`export as namespace`

一般使用 `export as namespace` 时，都是先有了 npm 包的声明文件，再基于它添加一条 `export as namespace` 语句，即可将声明好的一个变量声明为全局变量，举例如下[22]：

```
1.  // types/foo/index.d.ts
2.
3.  export as namespace foo;
4.  export = foo;
5.
6.  declare function foo(): string;
7.  declare namespace foo {
8.   const bar: number;
9.  }
```

当然它也可以与 `export default` 一起使用：

```
1.  // types/foo/index.d.ts
2.
```

```
export as namespace foo;
export default foo;

declare function foo(): string;
declare namespace foo {
 const bar: number;
}
```

直接扩展全局变量

有的第三方库扩展了一个全局变量，可是此全局变量的类型却没有相应的更新过来，就会导致 ts 编译错误，此时就需要扩展全局变量的类型。比如扩展 `String` 类型[23]：

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

'foo'.prependHello();
```

通过声明合并，使用 `interface String` 即可给 `String` 添加属性或方法。

也可以使用 `declare namespace` 给已有的命名空间添加类型声明[24]：

```
// types/jquery-plugin/index.d.ts

declare namespace JQuery {
 interface CustomOptions {
  bar: string;
 }
}

interface JQueryStatic {
 foo(options: JQuery.CustomOptions): string;
}
```

```
// src/index.ts

jQuery.foo({
 bar: ''
});
```

在 npm 包或 UMD 库中扩展全局变量

如之前所说，对于一个 npm 包或者 UMD 库的声明文件，只有 `export` 导出的类型声明才能被导入。所以对于 npm 包或 UMD 库，如果导入此库之后会扩展全局变量，则需要使用另一种语法在声明文件中扩展全局变量的类型，那就是 `declare global` 。

`declare global`

使用 `declare global` 可以在 npm 包或者 UMD 库的声明文件中扩展全局变量的类型[25]：

```
// types/foo/index.d.ts

declare global {
 interface String {
 prependHello(): string;
 }
}

export {};
// src/index.ts

'bar'.prependHello();
```

注意即使此声明文件不需要导出任何东西，仍然需要导出一个空对象，用来告诉编译器这是一个模块的声明文件，而不是一个全局变量的声明文件。

模块插件

有时通过 `import` 导入一个模块插件，可以改变另一个原有模块的结构。此时如果原有模块已经有了类型声明文件，而插件模块没有类型声明文件，就会导致类型不完整，缺少插件部分的类型。ts 提供了一个语法 `declare module` ，它可以用来扩展原有模块的类型。

`declare module`

如果是需要扩展原有模块的话，需要在类型声明文件中先引用原有模块，再使用 `declare module` 扩展原有模块[26]：

```
// types/moment-plugin/index.d.ts

import * as moment from 'moment';

declare module 'moment' {
 export function foo(): moment.CalendarKey;
}
// src/index.ts
```

```
2.
3.  import * as moment from 'moment';
4.  import 'moment-plugin';
5.
6.  moment.foo();
```

`declare module` 也可用于在一个文件中一次性声明多个模块的类型[27]：

```
1.  // types/foo-bar.d.ts
2.
3.  declare module 'foo' {
4.   export interface Foo {
5.   foo: string;
6.   }
7.  }
8.
9.  declare module 'bar' {
10.  export function bar(): string;
11.  }
1.  // src/index.ts
2.
3.  import { Foo } from 'foo';
4.  import * as bar from 'bar';
5.
6.  let f: Foo;
7.  bar.bar();
```

声明文件中的依赖

一个声明文件有时会依赖另一个声明文件中的类型，比如在前面的 `declare module` 的例子中，我们就在声明文件中导入了 `moment` ，并且使用了 `moment.CalendarKey` 这个类型：

```
1.  // types/moment-plugin/index.d.ts
2.
3.  import * as moment from 'moment';
4.
5.  declare module 'moment' {
6.   export function foo(): moment.CalendarKey;
7.  }
```

除了可以在声明文件中通过 `import` 导入另一个声明文件中的类型之外，还有一个语法也可以用来导入另一个声明文件，那就是三斜线指令。

三斜线指令

二斜线指令

与 `namespace` 类似，三斜线指令也是 ts 在早期版本中为了描述模块之间的依赖关系而创造的语法。随着 ES6 的广泛应用，现在已经不建议再使用 ts 中的三斜线指令来声明模块之间的依赖关系了。

但是在声明文件中，它还是有一定的用武之地。

类似于声明文件中的 `import`，它可以用来导入另一个声明文件。与 `import` 的区别是，当且仅当在以下几个场景下，我们才需要使用三斜线指令替代 `import`：

- 当我们在**书写**一个全局变量的声明文件时
- 当我们需要**依赖**一个全局变量的声明文件时

书写一个全局变量的声明文件

这些场景听上去很拗口，但实际上很好理解——在全局变量的声明文件中，是不允许出现 `import`，`export` 关键字的。一旦出现了，那么他就会被视为一个 npm 包或 UMD 库，就不再是全局变量的声明文件了。故当我们在书写一个全局变量的声明文件时，如果需要引用另一个库的类型，那么就必须用三斜线指令了[28]：

```
// types/jquery-plugin/index.d.ts

/// <reference types="jquery" />

declare function foo(options: JQuery.AjaxSettings): string;
```

```
// src/index.ts

foo({});
```

三斜线指令的语法如上，`///` 后面使用 xml 的格式添加了对 `jquery` 类型的依赖，这样就可以在声明文件中使用 `JQuery.AjaxSettings` 类型了。

注意，三斜线指令必须放在文件的最顶端，三斜线指令的前面只允许出现单行或多行注释。

依赖一个全局变量的声明文件

在另一个场景下，当我们需要依赖一个全局变量的声明文件时，由于全局变量不支持通过 `import` 导入，当然也就必须使用三斜线指令来引入了[29]：

```
// types/node-plugin/index.d.ts

/// <reference types="node" />

export function foo(p: NodeJS.Process): string;
```

```
// src/index.ts
```

```
2.
3.  import { foo } from 'node-plugin';
4.
5.  foo(global.process);
```

在上面的例子中，我们通过三斜线指引入了 `node` 的类型，然后在声明文件中使用了 `NodeJS.Process` 这个类型。最后在使用到 `foo` 的时候，传入了 `node` 中的全局变量 `process` 。

由于引入的 `node` 中的类型都是全局变量的类型，它们是没有办法通过 `import` 来导入的，所以这种场景下也只能通过三斜线指令来引入了。

以上两种使用场景下，都是由于需要书写或需要依赖全局变量的声明文件，所以必须使用三斜线指令。在其他的一些不是必要使用三斜线指令的情况下，就都需要使用 `import` 来导入。

拆分声明文件

当我们的全局变量的声明文件太大时，可以通过拆分为多个文件，然后在一个入口文件中将它们一一引入，来提高代码的可维护性。比如 `jQuery` 的声明文件就是这样的：

```
1.  // node_modules/@types/jquery/index.d.ts
2.
3.  /// <reference types="sizzle" />
4.  /// <reference path="JQueryStatic.d.ts" />
5.  /// <reference path="JQuery.d.ts" />
6.  /// <reference path="misc.d.ts" />
7.  /// <reference path="legacy.d.ts" />
8.
9.  export = jQuery;
```

其中用到了 `types` 和 `path` 两种不同的指令。它们的区别是： `types` 用于声明对另一个库的依赖，而 `path` 用于声明对另一个文件的依赖。

上例中， `sizzle` 是与 `jquery` 平行的另一个库，所以需要使用 `types="sizzle"` 来声明对它的依赖。而其他的三斜线指令就是将 `jquery` 的声明拆分到不同的文件中了，然后在这个入口文件中使用 `path="foo"` 将它们一一引入。

其他三斜线指令

除了这两种三斜线指令之外，还有其他的三斜线指令，比如 `/// <reference no-default-lib="true"/>` ， `/// <amd-module />` 等，但它们都是废弃的语法，故这里就不介绍了，详情可见官网。

自动生成声明文件

如果库的源码本身就是由 ts 写的，那么在使用 `tsc` 脚本将 ts 编译为 js 的时候，添加 `declaration` 选项，就可以同时也生成 `.d.ts` 声明文件了。

我们可以在命令行中添加 `--declaration` （简写 `-d` ），或者在 `tsconfig.json` 中添加 `declaration` 选项。这里以 `tsconfig.json` 为例：

```
{
 "compilerOptions": {
 "module": "commonjs",
 "outDir": "lib",
 "declaration": true,
 }
}
```

上例中我们添加了 `outDir` 选项，将 ts 文件的编译结果输出到 `lib` 目录下，然后添加了 `declaration` 选项，设置为 `true` ，表示将会由 ts 文件自动生成 `.d.ts` 声明文件，也会输出到 `lib` 目录下。

运行 `tsc` 之后，目录结构如下[30]：

```
/path/to/project
├── lib
| ├── bar
| | ├── index.d.ts
| | └── index.js
| ├── index.d.ts
| └── index.js
├── src
| ├── bar
| | └── index.ts
| └── index.ts
├── package.json
└── tsconfig.json
```

在这个例子中， `src` 目录下有两个 ts 文件，分别是 `src/index.ts` 和 `src/bar/index.ts` ，它们被编译到 `lib` 目录下的同时，也会生成对应的两个声明文件 `lib/index.d.ts` 和 `lib/bar/index.d.ts` 。它们的内容分别是：

```
// src/index.ts

export * from './bar';

export default function foo() {
 return 'foo';
```

```
}
// src/bar/index.ts

export function bar() {
 return 'bar';
}
// lib/index.d.ts

export * from './bar';
export default function foo(): string;
// lib/bar/index.d.ts

export declare function bar(): string;
```

可见，自动生成的声明文件基本保持了源码的结构，而将具体实现去掉了，生成了对应的类型声明。

使用 `tsc` 自动生成声明文件时，每个 ts 文件都会对应一个 `.d.ts` 声明文件。这样的好处是，使用方不仅可以在使用 `import foo from 'foo'` 导入默认的模块时获得类型提示，还可以在使用 `import bar from 'foo/lib/bar'` 导入一个子模块时，也获得对应的类型提示。

除了 `declaration` 选项之外，还有几个选项也与自动生成声明文件有关，这里只简单列举出来，不做详细演示了：

- `declarationDir` 设置生成 `.d.ts` 文件的目录
- `declarationMap` 对每个 `.d.ts` 文件，都生成对应的 `.d.ts.map`（sourcemap）文件
- `emitDeclarationOnly` 仅生成 `.d.ts` 文件，不生成 `.js` 文件

发布声明文件

当我们为一个库写好了声明文件之后，下一步就是将它发布出去了。

此时有两种方案：

1. 将声明文件和源码放在一起
2. 将声明文件发布到 `@types` 下

这两种方案中优先选择第一种方案。保持声明文件与源码在一起，使用时就不需要额外增加单独的声明文件库的依赖了，而且也能保证声明文件的版本与源码的版本保持一致。

仅当我们在给别人的仓库添加类型声明文件，但原作者不愿意合并 pull request 时，才需要使用第二种方案，将声明文件发布到 `@types` 下。

将声明文件和源码放在一起

如果声明文件是通过 `tsc` 自动生成的，那么无需做任何其他配置，只需要把编译好的文件也发布到 npm 上，使用方就可以获取到类型提示了。

如果是手动写的声明文件，那么需要满足以下条件之一，才能被正确的识别：

- 给 `package.json` 中的 `types` 或 `typings` 字段指定一个类型声明文件地址
- 在项目根目录下，编写一个 `index.d.ts` 文件
- 针对入口文件（`package.json` 中的 `main` 字段指定的入口文件），编写一个同名不同后缀的 `.d.ts` 文件

第一种方式是给 `package.json` 中的 `types` 或 `typings` 字段指定一个类型声明文件地址。比如：

```
{
  "name": "foo",
  "version": "1.0.0",
  "main": "lib/index.js",
  "types": "foo.d.ts",
}
```

指定了 `types` 为 `foo.d.ts` 之后，导入此库的时候，就会去找 `foo.d.ts` 作为此库的类型声明文件了。

`typings` 与 `types` 一样，只是另一种写法。

如果没有指定 `types` 或 `typings`，那么就会在根目录下寻找 `index.d.ts` 文件，将它视为此库的类型声明文件。

如果没有找到 `index.d.ts` 文件，那么就会寻找入口文件（`package.json` 中的 `main` 字段指定的入口文件）是否存在对应同名不同后缀的 `.d.ts` 文件。

比如 `package.json` 是这样时：

```
{
  "name": "foo",
  "version": "1.0.0",
  "main": "lib/index.js"
}
```

就会先识别 `package.json` 中是否存在 `types` 或 `typings` 字段。发现不存在，那么就会寻找是否存在 `index.d.ts` 文件。如果还是不存在，那么就会寻找是否存在 `lib/index.d.ts` 文件。假如说连 `lib/index.d.ts` 都不存在的话，就会被认为是一个没有提供类型声明文件的库了。

有的库为了支持导入子模块，比如 `import bar from 'foo/lib/bar'`，就需要额外再编写一个类型声明文件 `lib/bar.d.ts` 或者 `lib/bar/index.d.ts`，这与自动生成声明文件类似，一个库中同时包含

了多个类型声明文件。

将声明文件发布到 `@types` 下

如果我们是在给别人的仓库添加类型声明文件，但原作者不愿意合并 pull request，那么就需要将声明文件发布到 `@types` 下。

与普通的 npm 模块不同，`@types` 是统一由 [DefinitelyTyped][] 管理的。要将声明文件发布到 `@types` 下，就需要给 [DefinitelyTyped][] 创建一个 pull-request，其中包含了类型声明文件，测试代码，以及 `tsconfig.json` 等。

pull-request 需要符合它们的规范，并且通过测试，才能被合并，稍后就会被自动发布到 `@types` 下。

在 [DefinitelyTyped][] 中创建一个新的类型声明，需要用到一些工具，[DefinitelyTyped][] 的文档中已经有了详细的介绍，这里就不赘述了，以官方文档为准。

声明合并

如果定义了两个相同名字的函数、接口或类，那么它们会合并成一个类型：

函数的合并

之前学习过重载，我们可以使用重载定义多个函数类型：

```
function reverse(x: number): number
function reverse(x: string): string
function reverse(x: number | string): number | string {
 if (typeof x === 'number') {
 return Number(x.toString().split('').reverse().join(''))
 } else if (typeof x === 'string') {
 return x.split('').reverse().join('')
 }
}
```

接口的合并

接口中的属性在合并时会简单的合并到一个接口中：

```
interface Alarm {
```

```
interface Alarm {
 price: number
}
interface Alarm {
 weight: number
}
```

相当于：

```
interface Alarm {
 price: number
 weight: number
}
```

注意，**合并的属性的类型必须是唯一的**：

```
interface Alarm {
 price: number
}
interface Alarm {
 price: number // 虽然重复了，但是类型都是 `number`，所以不会报错
 weight: number
}
interface Alarm {
 price: number
}
interface Alarm {
 price: string // 类型不一致，会报错
 weight: number
}

// index.ts(5,3): error TS2403: Subsequent variable declarations must have the same type. Variable 'price' must be of type 'number', but here has type 'string'.
```

接口中方法的合并，与函数的合并一样：

```
interface Alarm {
 price: number
 alert(s: string): string
}
interface Alarm {
 weight: number
 alert(s: string, n: number): string
}
```

相当于：

```
interface Alarm {
 price: number
 weight: number
 alert(s: string): string
 alert(s: string, n: number): string
}
```

类的合并

类的合并与接口的合并规则一致。

变量声明

变量声明

let和const是JavaScript里相对较新的变量声明方式。 像我们之前提到过的， let在很多方面与var是相似的，但是可以帮助大家避免在JavaScript里常见一些问题。 const是对let的一个增强，它能阻止对一个变量再次赋值。

因为TypeScript是JavaScript的超集，所以它本身就支持let和const。下面我们会详细说明这些新的声明方式以及为什么推荐使用它们来代替 var。

var 声明

一直以来我们都是通过var关键字定义JavaScript变量。

```
var a = 10
```

大家都能理解，这里定义了一个名为a值为10的变量。
我们也可以在函数内部定义变量：

```
function f() {
 var message = "千锋教育·好程序员"
 return message
}
```

并且我们也可以在其它函数内部访问相同的变量。

```
function f() {
 var a = 10
```

```
  return function g() {
  var b = a + 1
  return b
  }
 }
var g = f()
 g() // returns 11
```

上面的例子里，g可以获取到f函数里定义的a变量。 每当g被调用时，它都可以访问到f里的a变量。

即使当g在f已经执行完后才被调用，它仍然可以访问及修改a。

```
function f() {
 var a = 1

 a = 2
 var b = g()
 a = 3

 return b

 function g() {
 return a
 }
}

f() // returns 2
```

作用域规则

对于熟悉其它语言的人来说，var声明有些奇怪的作用域规则。 看下面的例子：

```
function f(shouldInitialize: boolean) {
 if (shouldInitialize) {
 var x = 10
 }
 return x
}

f(true) // returns '10'
 f(false) // returns 'undefined'
```

有些读者可能要多看几遍这个例子。变量x是定义在*if语句里面*，但是我们却可以在语句的外面访问它。这是因为，var声明可以在包含它的函数，模块，命名空间或全局作用域内部任何位置被访问（包含它的代码块

是因为var声明可以在包含它的函数，模块，命名空间或全局作用域的部任何位置被访问，包含它的代码块对此没有什么影响。有些人称此为*var作用域或函数作用域*。函数参数也使用函数作用域。

这些作用域规则可能会引发一些错误。 其中之一就是，多次声明同一个变量并不会报错：

```
function sumMatrix(matrix: number[][]) {
 var sum = 0
 for (var i = 0; i < matrix.length; i++) {
 var currentRow = matrix[i]
 for (var i = 0; i < currentRow.length; i++) {
 sum += currentRow[i]
 }
 }
 return sum
}
```

这里很容易看出一些问题，里层的for循环会覆盖变量i，因为所有i都引用相同的函数作用域内的变量。 有经验的开发者们很清楚，这些问题可能在代码审查时漏掉，引发无穷的麻烦。

捕获变量怪异之处

快速的猜一下下面的代码会返回什么：

```
for (var i = 0; i < 10; i++) {
 setTimeout(function() { console.log(i) }, 100 * i)
}
```

介绍一下，setTimeout会在若干毫秒的延时后执行一个函数（等待其它代码执行完毕）。

好吧，看一下结果：

```
10
10
10
10
10
10
10
10
10
10
```

很多JavaScript程序员对这种行为已经很熟悉了。但也有大多数人可能期望输出结果是这样：

```
0
1
```

```
2
3
4
5
6
7
8
9
```

还记得我们上面提到的捕获变量吗？

我们传给setTimeout的每一个函数表达式实际上都引用了相同作用域里的同一个i。

让我们花点时间思考一下这是为什么。 setTimeout在若干毫秒后执行一个函数，并且是在for循环结束后。for循环结束后，i的值为10。 所以当函数被调用的时候，它会打印出 10！

一个通常的解决方法是使用立即执行的函数表达式（IIFE）来捕获每次迭代时i的值：

```
for (var i = 0 i < 10 i++) {
 // 捕获当前i的状态
 // 应用当前的值作为参数来执行函数
 (function(i) {
 setTimeout(function() { console.log(i) }, 100 * i)
 })(i)
}
```

let 声明

现在你已经知道了var存在一些问题，这恰好说明了为什么用let语句来声明变量。 除了名字不同外， let与var的写法一致。

```
let hello = "Hello!"
```

主要的区别不在语法上，而是语义，我们接下来会深入研究。

块作用域

当用let声明一个变量，它使用的是词法作用域或块作用域。不同于使用var声明的变量那样可以在包含它们的函数外访问，块作用域变量在包含它们的块或for循环之外是不能访问的。

```
function f(input: boolean) {
 let a = 100

 if (input) {
```

```
    // 这里仍然可以引用a
    let b = a + 1
    return b
    }

    // 报错: 'b' 在这里不存在
    return b
}
```

这里我们定义了2个变量a和b。a的作用域是在f函数体内，而b的作用域是在if语句块里。

在catch语句里声明的变量也具有同样的作用域规则。

```
try {
 throw "oh no!"
}
catch (e) {
 console.log("Oh well.")
}

// 错误: 'e' 在这里不存在
console.log(e)
```

拥有块级作用域的变量的另一个特点是，它们不能在被声明之前读或写。虽然这些变量始终“存在”于它们的作用域里，但在直到声明它的代码之前的区域都属于暂时性死区。它只是用来说明我们不能在let语句之前访问它们，幸运的是TypeScript可以告诉我们这些信息。

```
a++ // 在声明a之前引用a是非法的
let a
```

重定义及屏蔽

我们提过使用var声明时，它不在乎你声明多少次；你只会得到1个。

```
function f(x) {
 var x
 var x

 if (true) {
 var x
 }
}
```

在上面的例子里，所有x的声明实际上都引用一个相同的x，并且这是完全有效的代码。这经常会成为bug的来源。好的是，let声明就不会这么宽松了。

来源。好的是，let声明就不会这么宽松了。

```
let x = 10
let x = 20 // 错误，不能在1个作用域里多次声明`x`
```

并不是要求两个均是块级作用域的声明TypeScript才会给出一个错误的警告。

```
function f(x) {
 let x = 100 // error: interferes with parameter declaration
}

function g() {
 let x = 100
 var x = 100 // error: can't have both declarations of 'x'
}
```

并不是说块级作用域变量不能用函数作用域变量来声明。 而是块级作用域变量需要在明显不同的块里声明。

```
function f(condition, x) {
 if (condition) {
 let x = 100
 return x
 }

 return x
}

f(false, 0) // returns 0
f(true, 0) // returns 100
```

在一个嵌套作用域里引入一个新名字的行为称做屏蔽。它是一把双刃剑，它可能会不小心地引入新问题，同时也可能会解决一些错误。 例如，假设我们现在用let重写之前的sumMatrix函数。

```
function sumMatrix(matrix: number[][]) {
 let sum = 0
 for (let i = 0 i < matrix.length i++) {
 var currentRow = matrix[i]
 for (let i = 0 i < currentRow.length i++) {
 sum += currentRow[i]
 }
 }
 return sum
}
```

这个版本的循环能得到正确的结果，因为内层循环的i可以屏蔽掉外层循环的i。

通常来讲应该避免使用屏蔽，因为我们需要写出清晰的代码。

通常来讲应该避免使用屏蔽，因为我们需要写出清晰的代码。

块级作用域变量的获取

在我们最初谈及获取用var声明的变量时，我们简略地探究了一下在获取到了变量之后它的行为是怎样的。直观地讲，每次进入一个作用域时，它创建了一个变量的环境。就算作用域内代码已经执行完毕，这个环境与其捕获的变量依然存在。

```
function theCityThatAlwaysSleeps() {
 let getCity
 if (true) {
 let city = "Seattle"
 getCity = function() {
 return city
 }
 }

 return getCity()
}
```

因为我们已经在city的环境里获取到了city，所以就算if语句执行结束后我们仍然可以访问它。

回想一下前面setTimeout的例子，我们最后需要使用立即执行的函数表达式来获取每次for循环迭代里的状态。 实际上，我们做的是为获取到的变量创建了一个新的变量环境。 这样做挺痛苦的，但是幸运的是，你不必在TypeScript里这样做了。

当let声明出现在循环体里时拥有完全不同的行为。 不仅是在循环里引入了一个新的变量环境，而是针对每次迭代都会创建这样一个新作用域。这就是我们在使用立即执行的函数表达式时做的事，所以在setTimeout例子里我们仅使用let声明就可以了。

```
for (let i = 0 i < 10 i++) {
 setTimeout(function() {console.log(i) }, 100 * i)
}
```

会输出与预料一致的结果：

```
0
1
2
3
4
5
6
7
8
```

```
8
9
```

const 声明

const 声明是声明变量的另一种方式。

```
const numLivesForCat = 9
```

它们与let声明相似，但是就像它的名字所表达的，它们被赋值后不能再改变。换句话说，它们拥有与let相同的作用域规则，但是不能对它们重新赋值。

这很好理解，它们引用的值是不可变的。

```
const numLivesForCat = 9
const kitty = {
 name: "Aurora",
 numLives: numLivesForCat,
}

// 错误
kitty = {
 name: "Danielle",
 numLives: numLivesForCat
}

// 以下全部正确
kitty.name = "Rory"
kitty.name = "Kitty"
kitty.name = "Cat"
kitty.numLives--
```

除非你使用特殊的方法去避免，实际上const变量的内部状态是可修改的。

let vs. const

现在我们有两种作用域相似的声明方式，我们自然会问到底应该使用哪个。与大多数泛泛的问题一样，答案是：依情况而定。

使用最小特权原则，所有变量除了你计划去修改的都应该使用const。基本原则就是如果一个变量不需要对它写入，那么其它使用这些代码的人也不能够写入它们，并且要思考为什么会需要对这些变量重新赋值。使用 const也可以让我们更容易的推测数据的流动。

解构

解构数组

最简单的解构莫过于数组的解构赋值了：

```
let input = [1, 2]
let [first, second] = input
console.log(first) // outputs 1
console.log(second) // outputs 2
```

这创建了2个命名变量 first 和 second。 相当于使用了索引，但更为方便：

```
first = input[0]
second = input[1]
```

解构作用于已声明的变量会更好：

```
// 变量交换
[first, second] = [second, first]
```

作用于函数参数：

```
function f([first, second]: [number, number]) {
 console.log(first)
 console.log(second)
}
f(input)
```

你可以在数组里使用…语法创建剩余变量：

```
let [first, ...rest] = [1, 2, 3, 4]
console.log(first) // outputs 1
console.log(rest) // outputs [ 2, 3, 4 ]
```

当然，由于是JavaScript, 你可以忽略你不关心的尾随元素：

```
let [first] = [1, 2, 3, 4]
console.log(first) // outputs 1
```

或其它元素：

```
let [, second, , fourth] = [1, 2, 3, 4]
```

对象解构

你也可以解构对象：

```
let o = {
 a: "foo",
 b: 12,
 c: "bar"
}
let { a, b } = o
```

这通过 o.a and o.b 创建了 a 和 b 。 注意，如果你不需要 c 你可以忽略它。

就像数组解构，你可以用没有声明的赋值：

```
({ a, b } = { a: "baz", b: 101 })
```

注意，我们需要用括号将它括起来，因为Javascript通常会将以 { 起始的语句解析为一个块。

你可以在对象里使用…语法创建剩余变量：

```
let { a, ...passthrough } = o
let total = passthrough.b + passthrough.c.length
```

属性重命名

你也可以给属性以不同的名字：

```
let { a: newName1, b: newName2 } = o
```

这里的语法开始变得混乱。 你可以将 a: newName1 读做 “a 作为 newName1”。 方向是从左到右，好像你写成了以下样子：

```
let newName1 = o.a
let newName2 = o.b
```

令人困惑的是，这里的冒号不是指示类型的。 如果你想指定它的类型，仍然需要在其后写上完整的模式。

```
let {a, b}: {a: string, b: number} = o
```

默认值

默认值可以让你在属性为 undefined 时使用缺省值：

```
function keepWholeObject(wholeObject: { a: string, b?: number }) {
 let { a, b = 1001 } = wholeObject
}
```

现在，即使 b 为 undefined，keepWholeObject 函数的变量 wholeObject 的属性 a 和 b 都会有值。

函数声明

解构也能用于函数声明。看以下简单的情况：

```
type C = { a: string, b?: number }
function f({ a, b }: C): void {
  //
```

```
    // ...
}
```

但是，通常情况下更多的是指定默认值，解构默认值有些棘手。 首先，你需要在默认值之前设置其格式。

```
function f({ a="", b=0 } = {}): void {
 // ...
}
f()
```

上面的代码是一个类型推断的例子

其次，你需要知道在解构属性上给予一个默认或可选的属性用来替换主初始化列表。 要知道 C 的定义有一个 b 可选属性：

```
function f({ a, b = 0 } = { a: "" }): void {
 // ...
}
f({ a: "yes" }) // 正确，提供了参数，a值为字符串"yes"，默认 b = 0
f() // 正确，如果不提供参数，默认a值为空字符串 {a: ""}，b默认值为0：b = 0
f({}) // 错误，如果提供了参数，a必须要给值
```

要小心使用解构。从前面的例子可以看出，就算是最简单的解构表达式也是难以理解的。尤其当存在深层嵌套解构的时候，就算这时没有堆叠在一起的重命名，默认值和类型注解，也是令人难以理解的。解构表达式要尽量保持小而简单。你自己也可以直接使用解构将会生成的赋值表达式。

展开

展开操作符正与解构相反。它允许你将一个数组展开为另一个数组，或将一个对象展开为另一个对象。 例如：

```
let first = [1, 2]
let second = [3, 4]
let bothPlus = [0, ...first, ...second, 5]
```

这会令bothPlus的值为[0, 1, 2, 3, 4, 5]。 展开操作创建了first和second的一份浅拷贝。它们不会被展开操作所改变。

你还可以展开对象：

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" }
let search = { ...defaults, food: "rich" }
```

search的值为{ food: “rich”, price: “$$”, ambiance: “noisy” }。 对象的展开比数组的展开要复杂的多。像数组展开一样，它是从左至右进行处理，但结果仍为对象。这就意味着出现在展开对象后面的属性会覆盖前面的属性。 因此，如果我们修改上面的例子，在结尾处进行展开的话：

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" }
```

```
2.  let search = { food: "rich", ...defaults }
```

那么，defaults里的food属性会重写food: “rich”，在这里这并不是我们想要的结果。

对象展开还有其它一些意想不到的限制。 首先，它仅包含对象自身的可枚举属性。 大体上是说当你展开一个对象实例时，你会丢失其方法：

```
1.  class C {
2.   p = 12
3.   m() {
4.   }
5.  }
6.  let c = new C()
7.  let clone = { ...c }
8.  clone.p // 正确
9.  clone.m() // 错误!
```

其次，TypeScript编译器不允许展开泛型函数上的类型参数。这个特性会在TypeScript的未来版本中考虑实现。