TS笔记

简介

什么是TypeScript

> 添加了类型系统的JavaScript，适用于任何规模的项目

它强调了TS的两个最重要的特性：

1. 类型系统
2. 适用于任何规模

TS的特性

类型系统

从TS的名称可以看出，[类型]是其最核心的特性。

我们知道，JS是一门非常灵活的编程语言：

- 它没有类型约束，一个变量可能初始化时是字符串，过一会儿又被赋值为数字
- 由于隐式类型转换的存在，有的变量的类型很难在运行前就确定
- 基于原型的面向对象编程，使得原型上的属性或方法可以在运行时被修改
- 函数是 JavaScript 中的一等公民，可以赋值给变量，也可以当作参数或返回值

这种灵活性就像一把双刃剑，一方面使得 JavaScript 蓬勃发展，无所不能，从 2013 年开始就一直蝉联最普遍使用的编程语言排行榜冠军；另一方面也使得它的代码质量参次不起，维护成本高，运行时错误多。

而 TypeScript 的类型系统，在很大程度上弥补了 JavaScript 的缺点。

TS是静态类型

类型系统按照[类型检查的时机]来进行分类，可分为**动态类型和静态类型**

动态类型是指在运行时才会进行类型检查，这种语言的类型错误往往会导致运行时错误。**JavaScript 是一门解释型语言，没有编译阶段，所以它是动态类型**

静态类型是指编译阶段就能确定每个变量的类型，这种语言的类型错误往往会导致语法错误。TypeScript 在运行前需要先编译为 JavaScript，而在编译阶段就会进行类型检查，所以 TypeScript 是静态类型。

TS是弱类型

类型系统按照「是否允许隐式类型转换」来分类，可以分为强类型和弱类型。 TypeScript 是完全兼容 JavaScript 的，它不会修改 JavaScript 运行时的特性，所以它们都是弱类型。

适用于任何规模

TypeScript 非常适用于大型项目——这是显而易见的，类型系统可以为大型项目带来更高的可维护性，以及更少的 bug。

在中小型项目中推行 TypeScript 的最大障碍就是认为使用 TypeScript 需要写额外的代码，降低开发效率。但事实上，由于有[类型推论][]，大部分类型都不需要手动声明了。相反，TypeScript 增强了编辑器（IDE）的功能，包括代码补全、接口提示、跳转到定义、代码重构等，这在很大程度上提高了开发效率。而且 TypeScript 有近百个[编译选项][]，如果你认为类型检查过于严格，那么可以通过修改编译选项来降低类型检查的标准。

TypeScript 还可以和 JavaScript 共存。这意味着如果你有一个使用 JavaScript 开发的旧项目，又想使用 TypeScript 的特性，那么你不需要急着把整个项目都迁移到 TypeScript，你可以使用 TypeScript 编写新文件，然后在后续更迭中逐步迁移旧文件。如果一些 JavaScript 文件的迁移成本太高，TypeScript 也提供了一个方案，可以让你在不修改 JavaScript 文件的前提下，编写一个[类型声明文件][]，实现旧项目的渐进式迁移。

基础

原始数据类型

JavaScript 的类型分为两种：原始数据类型（Primitive data types）和对象类型（Object types）。

原始数据类型包括：布尔值、数值、字符串、`null`、`undefined` 以及 ES6 中的新类型 `Symbol` 和 ES10 中的新类型 `BigInt`。

看下**前五种**原始数据类型在 TypeScript 中的应用。

布尔值

布尔值是最基础的数据类型，在 TypeScript 中，使用 `boolean` 定义布尔值类型：

```
let isDone: boolean = false;

// 编译通过
// 后面约定，未强调编译错误的代码片段，默认为编译通过
```

注意，使用构造函数 `Boolean` 创造的对象**不是**布尔值：

```
let createdByNewBoolean: boolean = new Boolean(1);

// Type 'Boolean' is not assignable to type 'boolean'.
//   'boolean' is a primitive, but 'Boolean' is a wrapper object. Prefer using 'boolean'
when possible.
```

事实上 `new Boolean()` 返回的是一个 `Boolean` 对象：

```
let createdByNewBoolean: Boolean = new Boolean(1);
```

直接调用 `Boolean` 也可以返回一个 `boolean` 类型：

```
let createdByBoolean: boolean = Boolean(1);
```

在 TypeScript 中，`boolean` 是 JavaScript 中的基本类型，而 `Boolean` 是 JavaScript 中的构造函数。其他基本类型（除了 `null` 和 `undefined`）一样，不再赘述。

数值

使用 `number` 定义数值类型：

```
let decLiteral: number = 6;
let hexLiteral: number = 0xf00d;
// ES6 中的二进制表示法
let binaryLiteral: number = 0b1010;
// ES6 中的八进制表示法
let octalLiteral: number = 0o744;
let notANumber: number = NaN;
let infinityNumber: number = Infinity;
```

编译结果：

```
var decLiteral = 6;
var hexLiteral = 0xf00d;
// ES6 中的二进制表示法
var binaryLiteral = 10;
// ES6 中的八进制表示法
var octalLiteral = 484;
var notANumber = NaN;
var infinityNumber = Infinity;
```

字符串

使用 `string` 定义字符串类型：

```
let myName: string = 'Tom';
let myAge: number = 25;

// 模板字符串
let sentence: string = `Hello, my name is ${myName}.
I'll be ${myAge + 1} years old next month.`;
```

编译结果：

```
var myName = 'Tom';
var myAge = 25;
// 模板字符串
var sentence = "Hello, my name is " + myName + ".
I'll be " + (myAge + 1) + " years old next month.";
```

空值

JavaScript 没有空值（Void）的概念，在 TypeScript 中，可以用 `void` 表示没有任何返回值的函数：

```
function alertName(): void {
    alert('My name is Tom');
}
```

声明一个 `void` 类型的变量没有什么用，因为你只能将它赋值为 `undefined` 和 `null`（只在 --strictNullChecks 未指定时）：

```
let unusable: void = undefined;
```

Null 和 Undefined

在 TypeScript 中，可以使用 `null` 和 `undefined` 来定义这两个原始数据类型：

```
let u: undefined = undefined;
let n: null = null;
```

与 `void` 的区别是，`undefined` 和 `null` 是所有类型的子类型。也就是说 `undefined` 类型的变量，可以赋值给 `number` 类型的变量：

```
// 这样不会报错
let num: number = undefined;
```

```
// 这样也不会报错
let u: undefined;
let num: number = u;
```

而 `void` 类型的变量不能赋值给 `number` 类型的变量：

```
let u: void;
let num: number = u;

// Type 'void' is not assignable to type 'number'.
```

任意值

任意值（Any）用来表示允许赋值为任意类型。

什么是任意值类型

如果是一个普通类型，在赋值过程中改变类型是不被允许的：

```
let myFavoriteNumber: string = 'seven';
myFavoriteNumber = 7;

// index.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.
```

但如果是 `any` 类型，则允许被赋值为任意类型。

```
let myFavoriteNumber: any = 'seven';
myFavoriteNumber = 7;
```

任意值的属性和方法

在任意值上访问任何属性都是允许的：

```
let anyThing: any = 'hello';
console.log(anyThing.myName);
console.log(anyThing.myName.firstName);
```

也允许调用任何方法：

```
let anyThing: any = 'Tom';
anyThing.setName('Jerry');
anyThing.setName('Jerry').sayHello();
anyThing.myName.setFirstName('Cat');
```

可以认为，**声明一个变量为任意值之后，对它的任何操作，返回的内容的类型都是任意值。**

未声明类型的变量

变量如果在声明的时候，未指定其类型，那么它会被识别为任意值类型：

```
let something;
something = 'seven';
something = 7;

something.setName('Tom');
```

等价于

```
let something: any;
something = 'seven';
something = 7;

something.setName('Tom');
```

类型推论(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。

什么是类型推论

以下代码虽然没有指定类型，但是会在编译的时候报错：

```
let myFavoriteNumber = 'seven';
myFavoriteNumber = 7;

// index.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.
```

事实上，它等价于：

```
let myFavoriteNumber: string = 'seven';
myFavoriteNumber = 7;

// index.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.
```

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;
```

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。

如：

```
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven';
myFavoriteNumber = 7;
```

```
let myFavoriteNumber: string | number;
myFavoriteNumber = true;

// index.ts(2,1): error TS2322: Type 'boolean' is not assignable to type 'string | number'.
//   Type 'boolean' is not assignable to type 'number'.
```

联合类型使用 `|` 分隔每个类型。

这里的 `let myFavoriteNumber: string | number` 的含义是，允许 `myFavoriteNumber` 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和方法

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;
}

// index.ts(2,22): error TS2339: 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：

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

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 myFavoriteNumber: string | number;
myFavoriteNumber = 'seven';
console.log(myFavoriteNumber.length); // 5
myFavoriteNumber = 7;
console.log(myFavoriteNumber.length); // 编译时报错

// index.ts(5,30): error TS2339: Property 'length' does not exist on type 'number'.
```

对象类型--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。

如：

```
interface Person {
    name: string;
    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**：

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

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

// index.ts(6,5): error TS2322: Type '{ name: string; }' is not assignable to type
'Person'.
//   Property 'age' is missing in type '{ name: string; }'.
```

多一些属性也是**不允许的**：

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

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

// index.ts(9,5): error TS2322: 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：

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

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

```ts
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**：

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

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

// examples/playground/index.ts(9,5): error TS2322: 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：

```ts
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**：

```ts
interface Person {
    name: string;
    age?: number;
```

```
    [propName: string]: string;
}

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

// index.ts(3,5): error TS2411: Property 'age' of type 'number' is not assignable to string
index type 'string'.
// index.ts(7,5): error TS2322: Type '{ [x: string]: string | number; name: string; age:
number; gender: string; }' is not assignable to type 'Person'.
//   Index signatures are incompatible.
//     Type 'string | number' is not assignable to type 'string'.
//       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。

一个接口中只能定义一个任意属性。如果接口中有多个类型的属性，则可以在任意属性中使用联合类型：

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

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

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;
```

```
// index.ts(14,5): error TS2540: Cannot assign to 'id' because it is a constant or 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;

// index.ts(8,5): error TS2322: Type '{ name: string; gender: string; }' is not assignable to type 'Person'.
//   Property 'id' is missing in type '{ name: string; gender: string; }'.
// index.ts(13,5): error TS2540: Cannot assign to 'id' because it is a constant or 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。

数组类型

在 TypeScript 中，数组类型有多种定义方式，比较灵活。

[类型+方括号] 表示法

最简单的方法是使用「类型 + 方括号」来表示数组：

```
let fibonacci: number[] = [1, 1, 2, 3, 5];
```

数组的项中**不允许**出现其他的类型：

```
let fibonacci: number[] = [1, '1', 2, 3, 5];

// Type 'string' is not assignable to type 'number'.
```

数组的一些方法的参数也会根据数组在定义时约定的类型进行限制：

```ts
let fibonacci: number[] = [1, 1, 2, 3, 5];
fibonacci.push('8');

// Argument of type '"8"' is not assignable to parameter of type 'number'.
```

上例中，`push` 方法只允许传入 `number` 类型的参数，但是却传了一个 `"8"` 类型的参数，所以报错了。

数组泛型表示法

我们也可以使用数组泛型（Array Generic） `Array<elemType>` 来表示数组：

```ts
let fibonacci: Array<number> = [1, 1, 2, 3, 5];
```

用接口表示法

接口也可以用来描述数组：

```ts
interface NumberArray {
    [index: number]: number;
}
let fibonacci: NumberArray = [1, 1, 2, 3, 5];
```

`NumberArray` 表示：只要索引的类型是数字时，那么值的类型必须是数字。

虽然接口也可以用来描述数组，但是我们一般不会这么做，因为这种方式比前两种方式复杂多了。

不过有一种情况例外，那就是它常用来表示类数组。

类数组表示法

类数组（Array-like Object）不是数组类型，比如 `arguments`：

```ts
function sum() {
    let args: number[] = arguments;
}

// Type 'IArguments' is missing the following properties from type 'number[]': pop, push,
concat, join, and 24 more.
```

上例中，`arguments` 实际上是一个类数组，不能用普通的数组的方式来描述，而应该用接口：

```ts
function sum() {
    let args: {
        [index: number]: number;
        length: number;
        callee: Function;
    } = arguments;
}
```

在这个例子中，我们除了约束当索引的类型是数字时，值的类型必须是数字之外，也约束了它还有 `length` 和 `callee` 两个属性。

事实上常用的类数组都有自己的接口定义，如 `IArguments`，`NodeList`，`HTMLCollection` 等：

```
function sum() {
    let args: IArguments = arguments;
}
```

其中 `IArguments` 是 TypeScript 中定义好了的类型，它实际上就是：

```
interface IArguments {
    [index: number]: any;
    length: number;
    callee: Function;
}
```

any表示法

一个比较常见的做法是，用 `any` 表示数组中允许出现任意类型：

```
let list: any[] = ['xcatliu', 25, { website: 'http://xcatliu.com' }];
```

函数的类型

> 函数是 JavaScript 中的一等公民

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);

// index.ts(4,1): error TS2346: Supplied parameters do not match any signature of call
target.
```

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

// index.ts(4,1): error TS2346: Supplied parameters do not match any signature of call
target.
```

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。

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;
}
```

采用函数表达式|接口定义函数的方式时，对等号左侧进行类型限制，可以保证以后对函数名赋值时保证参数个数、参数类型、返回值类型不变。

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');

// index.ts(1,40): error TS1016: A required parameter cannot follow an optional parameter.
```

默认参数值

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');
```

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: any[] = [];
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

函数的重载

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

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。

类型断言

类型断言（Type Assertion）可以用来手动指定一个值的类型。

语法

```
值 as 类型
```

或是

```
<类型>值
```

在 tsx 语法（React 的 jsx 语法的 ts 版）中必须使用前者，即 `值 as 类型`。

故建议大家在使用类型断言时，统一使用 `值 as 类型` 这样的语法。

类型断言的用途

类型断言的常见用途有以下几种：

将一个联合类型断言为其中一个类型

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中共有的属性或方法**：

```
interface Cat {
    name: string;
    run(): void;
}
interface Fish {
    name: string;
    swim(): void;
}

function getName(animal: Cat | Fish) {
    return animal.name;
}
```

而有时候，我们确实需要在还不确定类型的时候就访问其中一个类型特有的属性或方法，比如：

```
interface Cat {
    name: string;
    run(): void;
}
interface Fish {
    name: string;
    swim(): void;
}

function isFish(animal: Cat | Fish) {
    if (typeof animal.swim === 'function') {
        return true;
    }
    return false;
}

// index.ts:11:23 - error TS2339: Property 'swim' does not exist on type 'Cat | Fish'.
//   Property 'swim' does not exist on type 'Cat'.
```

上面的例子中，获取 `animal.swim` 的时候会报错。

此时可以使用类型断言，将 `animal` 断言成 `Fish`：

```
nterface Cat {
    name: string;
    run(): void;
}
interface Fish {
    name: string;
    swim(): void;
}

function isFish(animal: Cat | Fish) {
    if (typeof (animal as Fish).swim === 'function') {
        return true;
    }
    return false;
}
```

这样就可以解决访问 `animal.swim` 时报错的问题了。

需要注意的是，类型断言只能够「欺骗」TypeScript 编译器，无法避免运行时的错误，反而滥用类型断言可能会导致运行时错误：

```
interface Cat {
    name: string;
    run(): void;
}
interface Fish {
    name: string;
```

```
    swim(): void;
}

function swim(animal: Cat | Fish) {
    (animal as Fish).swim();
}

const tom: Cat = {
    name: 'Tom',
    run() { console.log('run') }
};
swim(tom);
// Uncaught TypeError: animal.swim is not a function`
```

上面的例子编译时不会报错，但在运行时会报错：

```
Uncaught TypeError: animal.swim is not a function`
```

原因是 `(animal as Fish).swim()` 这段代码隐藏了 `animal` 可能为 `Cat` 的情况，将 `animal` 直接断言为 `Fish` 了，而 TypeScript 编译器信任了我们的断言，故在调用 `swim()` 时没有编译错误。

可是 `swim` 函数接受的参数是 `Cat | Fish`，一旦传入的参数是 `Cat` 类型的变量，由于 `Cat` 上没有 `swim` 方法，就会导致运行时错误了。

总之，使用类型断言时一定要格外小心，尽量避免断言后调用方法或引用深层属性，以减少不必要的运行时错误。

将一个父类断言为更加具体的子类

当类之间有继承关系时，类型断言也是很常见的：

```
class ApiError extends Error {
    code: number = 0;
}
class HttpError extends Error {
    statusCode: number = 200;
}

function isApiError(error: Error) {
    if (typeof (error as ApiError).code === 'number') {
        return true;
    }
    return false;
}
```

上面的例子中，我们声明了函数 `isApiError`，它用来判断传入的参数是不是 `ApiError` 类型，为了实现这样一个函数，它的参数的类型肯定得是比较抽象的父类 `Error`，这样的话这个函数就能接受 `Error` 或它的子类作为参数了。

但是由于父类 `Error` 中没有 `code` 属性，故直接获取 `error.code` 会报错，需要使用类型断言获取 `(error as ApiError).code`。

大家可能会注意到，在这个例子中有一个更合适的方式来判断是不是 `ApiError`，那就是使用 `instanceof`：

```
class ApiError extends Error {
    code: number = 0;
}
class HttpError extends Error {
    statusCode: number = 200;
}

function isApiError(error: Error) {
    if (error instanceof ApiError) {
        return true;
    }
    return false;
}
```

上面的例子中，确实使用 `instanceof` 更加合适，因为 `ApiError` 是一个 JavaScript 的类，能够通过 `instanceof` 来判断 `error` 是否是它的实例。

但是有的情况下 `ApiError` 和 `HttpError` 不是一个真正的类，而只是一个 TypeScript 的接口（`interface`），接口是一个类型，不是一个真正的值，它在编译结果中会被删除，当然就无法使用 `instanceof` 来做运行时判断了：

```
interface ApiError extends Error {
    code: number;
}
interface HttpError extends Error {
    statusCode: number;
}

function isApiError(error: Error) {
    if (error instanceof ApiError) {
        return true;
    }
    return false;
}

// index.ts:9:26 - error TS2693: 'ApiError' only refers to a type, but is being used as a
value here.
```

此时就只能用类型断言，通过判断是否存在 `code` 属性，来判断传入的参数是不是 `ApiError` 了：

```
interface ApiError extends Error {
    code: number;
}
interface HttpError extends Error {
    statusCode: number;
}

function isApiError(error: Error) {
    if (typeof (error as ApiError).code === 'number') {
        return true;
    }
    return false;
}
```

将任何一个类型断言为any

理想情况下，TypeScript 的类型系统运转良好，每个值的类型都具体而精确。

当我们引用一个在此类型上不存在的属性或方法时，就会报错：

```
const foo: number = 1;
foo.length = 1;

// index.ts:2:5 - error TS2339: Property 'length' does not exist on type 'number'.
```

上面的例子中，数字类型的变量 `foo` 上是没有 `length` 属性的，故 TypeScript 给出了相应的错误提示。

这种错误提示显然是非常有用的。

但有的时候，我们非常确定这段代码不会出错，比如下面这个例子：

```
window.foo = 1;

// index.ts:1:8 - error TS2339: Property 'foo' does not exist on type 'Window & typeof
globalThis'.
```

上面的例子中，我们需要将 `window` 上添加一个属性 `foo`，但 TypeScript 编译时会报错，提示我们 `window` 上不存在 `foo` 属性。

此时我们可以使用 `as any` 临时将 `window` 断言为 `any` 类型：

```
(window as any).foo = 1;
```

在 `any` 类型的变量上，访问任何属性都是允许的。

需要注意的是，将一个变量断言为 `any` 可以说是解决 TypeScript 中类型问题的最后一个手段。

它极有可能掩盖了真正的类型错误，所以如果不是非常确定，就不要使用 `as any`。

上面的例子中，我们也可以通过[扩展 window 的类型（TODO）][]解决这个错误，不过如果只是临时的增加 `foo` 属性，`as any` 会更加方便。

总之，**一方面不能滥用 `as any`，另一方面也不要完全否定它的作用，我们需要在类型的严格性和开发的便利性之间掌握平衡**（这也是 TypeScript 的设计理念之一），才能发挥出 TypeScript 最大的价值。

将any断言为一个具体的类型

在日常的开发中，我们不可避免的需要处理 `any` 类型的变量，它们可能是由于第三方库未能定义好自己的类型，也有可能是历史遗留的或其他人编写的烂代码，还可能是受到 TypeScript 类型系统的限制而无法精确定义类型的场景。

遇到 `any` 类型的变量时，我们可以选择无视它，任由它滋生更多的 `any`。

我们也可以选择改进它，通过类型断言及时的把 `any` 断言为精确的类型，亡羊补牢，使我们的代码向着高可维护性的目标发展。

举例来说，历史遗留的代码中有个 `getCacheData`，它的返回值是 `any`：

```
function getCacheData(key: string): any {
    return (window as any).cache[key];
}
```

那么我们在使用它时，最好能够将调用了它之后的返回值断言成一个精确的类型，这样就方便了后续的操作：

```
function getCacheData(key: string): any {
    return (window as any).cache[key];
}

interface Cat {
    name: string;
    run(): void;
}

const tom = getCacheData('tom') as Cat;
tom.run();
```

上面的例子中，我们调用完 `getCacheData` 之后，立即将它断言为 `Cat` 类型。这样的话明确了 `tom` 的类型，后续对 `tom` 的访问时就有了代码补全，提高了代码的可维护性。

类型断言的限制

从上面的例子中，我们可以总结出：

- 联合类型可以被断言为其中一个类型
- 父类可以被断言为子类
- 任何类型都可以被断言为 any
- any 可以被断言为任何类型

那么类型断言有没有什么限制呢？是不是任何一个类型都可以被断言为任何另一个类型呢？

答案是否定的——并不是任何一个类型都可以被断言为任何另一个类型。

具体来说，若 `A` 兼容 `B`，那么 `A` 能够被断言为 `B`，`B` 也能被断言为 `A`。

下面我们通过一个简化的例子，来理解类型断言的限制：

```ts
interface Animal {
    name: string;
}
interface Cat {
    name: string;
    run(): void;
}

let tom: Cat = {
    name: 'Tom',
    run: () => { console.log('run') }
};
let animal: Animal = tom;
```

我们知道，TypeScript 是结构类型系统，类型之间的对比只会比较它们最终的结构，而会忽略它们定义时的关系。

在上面的例子中，`Cat` 包含了 `Animal` 中的所有属性，除此之外，它还有一个额外的方法 `run`。TypeScript 并不关心 `Cat` 和 `Animal` 之间定义时是什么关系，而只会看它们最终的结构有什么关系——所以它与 `Cat extends Animal` 是等价的：

```ts
interface Animal {
    name: string;
}
interface Cat extends Animal {
    run(): void;
}
```

那么也不难理解为什么 `Cat` 类型的 `tom` 可以赋值给 `Animal` 类型的 `animal` 了——就像面向对象编程中我们可以将子类的实例赋值给类型为父类的变量。

我们把它换成 TypeScript 中更专业的说法，即：`Animal` 兼容 `Cat`。

当 `Animal` 兼容 `Cat` 时，它们就可以互相进行类型断言了：

```ts
interface Animal {
    name: string;
}
interface Cat {
    name: string;
    run(): void;
}

function testAnimal(animal: Animal) {
    return (animal as Cat);
}
function testCat(cat: Cat) {
    return (cat as Animal);
}
```

这样的设计其实也很容易就能理解：

- 允许 `animal as Cat` 是因为「父类可以被断言为子类」，这个前面已经学习过了

- 允许 `cat as Animal` 是因为既然子类拥有父类的属性和方法，那么被断言为父类，获取父类的属性、调用父类的方法，就不会有任何问题，故「子类可以被断言为父类」

需要注意的是，这里我们使用了简化的父类子类的关系来表达类型的兼容性，而实际上 TypeScript 在判断类型的兼容性时，比这种情况复杂很多。

总之，若 `A` 兼容 `B`，那么 `A` 能够被断言为 `B`，`B` 也能被断言为 `A`。

同理，若 `B` 兼容 `A`，那么 `A` 能够被断言为 `B`，`B` 也能被断言为 `A`。

所以这也可以换一种说法：

要使得 `A` 能够被断言为 `B`，只需要 `A` 兼容 `B` 或 `B` 兼容 `A` 即可，这也是为了在类型断言时的安全考虑，毕竟毫无根据的断言是非常危险的。

双重断言

既然：

- 任何类型都可以被断言为 any
- any 可以被断言为任何类型

那么我们是不是可以使用双重断言 `as any as Foo` 来将任何一个类型断言为任何另一个类型呢?

```
interface Cat {
    run(): void;
}
interface Fish {
    swim(): void;
}

function testCat(cat: Cat) {
    return (cat as any as Fish);
}
```

在上面的例子中，若直接使用 `cat as Fish` 肯定会报错，因为 `Cat` 和 `Fish` 互相都不兼容。

但是若使用双重断言，则可以打破「要使得 `A` 能够被断言为 `B`，只需要 `A` 兼容 `B` 或 `B` 兼容 `A` 即可」的限制，将任何一个类型断言为任何另一个类型。

若你使用了这种双重断言，那么十有八九是非常错误的，它很可能会导致运行时错误。

除非迫不得已，千万别用双重断言。

类型断言VS类型转换

类型断言只会影响 TypeScript 编译时的类型，类型断言语句在编译结果中会被删除：

```
unction toBoolean(something: any): boolean {
    return something as boolean;
}

toBoolean(1);
// 返回值为 1
```

在上面的例子中，将 `something` 断言为 `boolean` 虽然可以通过编译，但是并没有什么用，代码在编译后会变成：

```
function toBoolean(something) {
    return something;
}

toBoolean(1);
// 返回值为 1
```

所以类型断言不是类型转换，它不会真的影响到变量的类型。

若要进行类型转换，需要直接调用类型转换的方法：

```
function toBoolean(something: any): boolean {
    return Boolean(something);
}

toBoolean(1);
// 返回值为 true
```

类型断言VS类型声明

如：

```
function getCacheData(key: string): any {
    return (window as any).cache[key];
}

interface Cat {
    name: string;
    run(): void;
}

const tom = getCacheData('tom') as Cat;
tom.run();
```

我们使用 `as Cat` 将 `any` 类型断言为了 `Cat` 类型。

但实际上还有其他方式可以解决这个问题：

```
function getCacheData(key: string): any {
    return (window as any).cache[key];
}

interface Cat {
    name: string;
    run(): void;
}

const tom: Cat = getCacheData('tom');
tom.run();
```

上面的例子中，我们通过类型声明的方式，将 `tom` 声明为 `Cat`，然后再将 `any` 类型的 `getCacheData('tom')` 赋值给 `Cat` 类型的 `tom`。

这和类型断言是非常相似的，而且产生的结果也几乎是一样的——`tom` 在接下来的代码中都变成了 `Cat` 类型。

它们的区别，可以通过这个例子来理解：

```
interface Animal {
    name: string;
}
interface Cat {
    name: string;
    run(): void;
}

const animal: Animal = {
    name: 'tom'
};
let tom = animal as Cat;
```

在上面的例子中，由于 `Animal` 兼容 `Cat`，故可以将 `animal` 断言为 `Cat` 赋值给 `tom`。

但是若直接声明 `tom` 为 `Cat` 类型：

```
interface Animal {
    name: string;
}
interface Cat {
    name: string;
    run(): void;
}

const animal: Animal = {
    name: 'tom'
};
let tom: Cat = animal;

// index.ts:12:5 - error TS2741: Property 'run' is missing in type 'Animal' but required in
type 'Cat'.
```

则会报错，不允许将 `animal` 赋值为 `Cat` 类型的 `tom`。

这很容易理解，`Animal` 可以看作是 `Cat` 的父类，当然不能将父类的实例赋值给类型为子类的变量。

深入的讲，它们的核心区别就在于：

- `animal` 断言为 `Cat`，只需要满足 `Animal` 兼容 `Cat` 或 `Cat` 兼容 `Animal` 即可
- `animal` 赋值给 `tom`，需要满足 `Cat` 兼容 `Animal` 才行

但是 `Cat` 并不兼容 `Animal`。

而在前一个例子中，由于 `getCacheData('tom')` 是 `any` 类型，`any` 兼容 `Cat`，`Cat` 也兼容 `any`，故

```
const tom = getCacheData('tom') as Cat;
```

等价于

```
const tom: Cat = getCacheData('tom');
```

知道了它们的核心区别，就知道了类型声明是比类型断言更加严格的。

所以为了增加代码的质量，我们最好优先使用类型声明，这也比类型断言的 `as` 语法更加优雅。

类型断言VS泛型

如：

```
function getCacheData(key: string): any {
    return (window as any).cache[key];
}

interface Cat {
    name: string;
    run(): void;
}

const tom = getCacheData('tom') as Cat;
tom.run();
```

我们还有第三种方式可以解决这个问题，那就是泛型：

```
function getCacheData<T>(key: string): T {
    return (window as any).cache[key];
}

interface Cat {
    name: string;
    run(): void;
}

const tom = getCacheData<Cat>('tom');
tom.run();
```

通过给 `getCacheData` 函数添加了一个泛型 `<T>`，我们可以更加规范的实现对 `getCacheData` 返回值的约束，这也同时去除掉了代码中的 `any`，是最优的一个解决方案。

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

- `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
- `///` 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：

```
$('#foo');
// or
jQuery('#foo');
```

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

```
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：

```
// 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。

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 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：

```
// 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：

```
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]：

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

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

```ts
// 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：

```ts
// 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：

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

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

```ts
// 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：

```ts
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：

```
// 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：

```
// 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`：

```
// 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：

```
// 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：

```
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：

```
// 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',
    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：

```
// 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');
```

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，后面的配置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，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：

```
// 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：

```
// 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：

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

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

```ts
// src/index.ts

import { foo } from 'foo';

console.log(foo.name);
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：

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

export default function foo(): string;
```

```ts
// 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]：

```ts
// 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：

```ts
// 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：

```ts
// 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：

```ts
// 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`：

```ts
// 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：

```ts
// 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：

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

export = foo;

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

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：

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

export as namespace foo;
export = foo;

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

当然它也可以与 `export default` 一起使用：

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

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：

```
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：

```
// 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:

```
// 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:

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

import * as moment from 'moment';

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

```
// src/index.ts

import * as moment from 'moment';
import 'moment-plugin';

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:

```ts
// types/foo-bar.d.ts

declare module 'foo' {
    export interface Foo {
        foo: string;
    }
}

declare module 'bar' {
    export function bar(): string;
}
```

```ts
// src/index.ts

import { Foo } from 'foo';
import * as bar from 'bar';

let f: Foo;
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：

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

import * as moment from 'moment';

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

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

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:

```
// 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:

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

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

export function foo(p: NodeJS.Process): string;
// src/index.ts

import { foo } from 'node-plugin';

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:

```
// node_modules/@types/jquery/index.d.ts

/// <reference types="sizzle" />
/// <reference path="JQueryStatic.d.ts" />
/// <reference path="JQuery.d.ts" />
/// <reference path="misc.d.ts" />
/// <reference path="legacy.d.ts" />

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：

```
/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：

```ts
// src/index.ts

export * from './bar';

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

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

export function bar() {
    return 'bar';
}
```

```ts
// lib/index.d.ts

export * from './bar';
export default function foo(): string;
```

```ts
// 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。

内置对象

JavaScript 中有很多内置对象，它们可以直接在 TypeScript 中当做定义好了的类型。

内置对象是指根据标准在全局作用域（Global）上存在的对象。这里的标准是指 ECMAScript 和其他环境（比如 DOM）的标准。

ECMAScript 的内置对象

ECMAScript 标准提供的内置对象有：

`Boolean`、`Error`、`Date`、`RegExp` 等。

我们可以在 TypeScript 中将变量定义为这些类型：

```
let b: Boolean = new Boolean(1);
let e: Error = new Error('Error occurred');
let d: Date = new Date();
let r: RegExp = /[a-z]/;
```

更多的内置对象，可以查看 MDN 的文档。

而他们的定义文件，则在 TypeScript 核心库的定义文件中。

DOM 和 BOM 的内置对象

DOM 和 BOM 提供的内置对象有：

`Document`、`HTMLElement`、`Event`、`NodeList` 等。

TypeScript 中会经常用到这些类型：

```ts
let body: HTMLElement = document.body;
let allDiv: NodeList = document.querySelectorAll('div');
document.addEventListener('click', function(e: MouseEvent) {
  // Do something
});
```

它们的定义文件同样在 TypeScript 核心库的定义文件中。

TypeScript 核心库的定义文件

TypeScript 核心库的定义文件中定义了所有浏览器环境需要用到的类型，并且是预置在 TypeScript 中的。

当你在使用一些常用的方法的时候，TypeScript 实际上已经帮你做了很多类型判断的工作了，比如：

```ts
Math.pow(10, '2');

// index.ts(1,14): error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
```

上面的例子中，`Math.pow` 必须接受两个 `number` 类型的参数。事实上 `Math.pow` 的类型定义如下：

```ts
interface Math {
    /**
     * Returns the value of a base expression taken to a specified power.
     * @param x The base value of the expression.
     * @param y The exponent value of the expression.
     */
    pow(x: number, y: number): number;
}
```

再举一个 DOM 中的例子：

```ts
document.addEventListener('click', function(e) {
    console.log(e.targetCurrent);
});

// index.ts(2,17): error TS2339: Property 'targetCurrent' does not exist on type 'MouseEvent'.
```

上面的例子中，`addEventListener` 方法是在 TypeScript 核心库中定义的：

```ts
interface Document extends Node, GlobalEventHandlers, NodeSelector, DocumentEvent {
    addEventListener(type: string, listener: (ev: MouseEvent) => any, useCapture?: boolean): void;
}
```

所以 `e` 被推断成了 `MouseEvent`，而 `MouseEvent` 是没有 `targetCurrent` 属性的，所以报错了。

注意，TypeScript 核心库的定义中不包含 Node.js 部分。

进阶

类型别名

类型别名用来给一个类型起个新名字。

如：

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === 'string') {
        return n;
    } else {
        return n();
    }
}
```

上例中，我们使用 `type` 创建类型别名。

类型别名常用于联合类型。

字符串字面量类型

字符串字面量类型用来约束取值只能是某几个字符串中的一个。

```
type EventNames = 'click' | 'scroll' | 'mousemove';
function handleEvent(ele: Element, event: EventNames) {
    // do something
}

handleEvent(document.getElementById('hello'), 'scroll');  // 没问题
handleEvent(document.getElementById('world'), 'dblclick'); // 报错，event 不能为 'dblclick'

// index.ts(7,47): error TS2345: Argument of type '"dblclick"' is not assignable to
parameter of type 'EventNames'.
```

上例中，我们使用 `type` 定了一个字符串字面量类型 `EventNames`，它只能取三种字符串中的一种。

注意，**类型别名与字符串字面量类型都是使用 `type` 进行定义。**

元组

数组合并了相同类型的对象，而元组（Tuple）合并了不同类型的对象。

元组起源于函数编程语言（如 F#），这些语言中会频繁使用元组。

如：

定义一对值分别为 `string` 和 `number` 的元组：

```
let tom: [string, number] = ['Tom', 25];
```

当赋值或访问一个已知索引的元素时，会得到正确的类型：

```
let tom: [string, number];
tom[0] = 'Tom';
tom[1] = 25;

tom[0].slice(1);
tom[1].toFixed(2);
```

也可以只赋值其中一项：

```
let tom: [string, number];
tom[0] = 'Tom';
```

但是当直接对元组类型的变量进行初始化或者赋值的时候，需要提供所有元组类型中指定的项。

```
let tom: [string, number];
tom = ['Tom', 25];
```

```
let tom: [string, number];
tom = ['Tom'];

// Property '1' is missing in type '[string]' but required in type '[string, number]'.
```

越界的元素

当添加越界的元素时，它的类型会被限制为元组中每个类型的联合类型：

```
let tom: [string, number];
tom = ['Tom', 25];
tom.push('male');
tom.push(true);

// Argument of type 'true' is not assignable to parameter of type 'string | number'.
```

枚举

枚举（Enum）类型用于取值被限定在一定范围内的场景，比如一周只能有七天，颜色限定为红绿蓝等。

如：

```
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

枚举成员会被赋值为从 `0` 开始递增的数字，同时也会对枚举值到枚举名进行反向映射：

```
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 0); // true
console.log(Days["Mon"] === 1); // true
console.log(Days["Tue"] === 2); // true
console.log(Days["Sat"] === 6); // true

console.log(Days[0] === "Sun"); // true
console.log(Days[1] === "Mon"); // true
console.log(Days[2] === "Tue"); // true
console.log(Days[6] === "Sat"); // true
```

事实上，上面的例子会被编译为：

```
var Days;
(function (Days) {
    Days[Days["Sun"] = 0] = "Sun";
    Days[Days["Mon"] = 1] = "Mon";
    Days[Days["Tue"] = 2] = "Tue";
    Days[Days["Wed"] = 3] = "Wed";
    Days[Days["Thu"] = 4] = "Thu";
    Days[Days["Fri"] = 5] = "Fri";
    Days[Days["Sat"] = 6] = "Sat";
})(Days || (Days = {}));
```

手动赋值

也可以给枚举项手动赋值：

```
enum Days {Sun = 7, Mon = 1, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 7); // true
console.log(Days["Mon"] === 1); // true
console.log(Days["Tue"] === 2); // true
console.log(Days["Sat"] === 6); // true
```

上面的例子中，未手动赋值的枚举项会接着上一个枚举项递增。

如果未手动赋值的枚举项与手动赋值的重复了，TypeScript 是不会察觉到这一点的：

```
enum Days {Sun = 3, Mon = 1, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 3); // true
console.log(Days["Wed"] === 3); // true
console.log(Days[3] === "Sun"); // false
console.log(Days[3] === "Wed"); // true
```

上面的例子中，递增到 `3` 的时候与前面的 `Sun` 的取值重复了，但是 TypeScript 并没有报错，导致 `Days[3]` 的值先是 `"Sun"`，而后又被 `"Wed"` 覆盖了。编译的结果是：

```
var Days;
(function (Days) {
    Days[Days["Sun"] = 3] = "Sun";
    Days[Days["Mon"] = 1] = "Mon";
    Days[Days["Tue"] = 2] = "Tue";
    Days[Days["Wed"] = 3] = "Wed";
    Days[Days["Thu"] = 4] = "Thu";
    Days[Days["Fri"] = 5] = "Fri";
    Days[Days["Sat"] = 6] = "Sat";
})(Days || (Days = {}));
```

所以使用的时候需要注意，最好不要出现这种覆盖的情况。

手动赋值的枚举项可以不是数字，此时需要使用类型断言来让 tsc 无视类型检查 (编译出的 js 仍然是可用的)：

```
enum Days {Sun = 7, Mon, Tue, Wed, Thu, Fri, Sat = <any>"S"};
```

```
var Days;
(function (Days) {
    Days[Days["Sun"] = 7] = "Sun";
    Days[Days["Mon"] = 8] = "Mon";
    Days[Days["Tue"] = 9] = "Tue";
    Days[Days["Wed"] = 10] = "Wed";
    Days[Days["Thu"] = 11] = "Thu";
    Days[Days["Fri"] = 12] = "Fri";
    Days[Days["Sat"] = "S"] = "Sat";
})(Days || (Days = {}));
```

当然，手动赋值的枚举项也可以为小数或负数，此时后续未手动赋值的项的递增步长仍为 `1`：

```
enum Days {Sun = 7, Mon = 1.5, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 7); // true
console.log(Days["Mon"] === 1.5); // true
console.log(Days["Tue"] === 2.5); // true
console.log(Days["Sat"] === 6.5); // true
```

常数项和计算所得项

枚举项有两种类型：常数项（constant member）和计算所得项（computed member）。

前面我们所举的例子都是常数项，一个典型的计算所得项的例子：

```
enum Color {Red, Green, Blue = "blue".length};
```

上面的例子中，`"blue".length` 就是一个计算所得项。

上面的例子不会报错，但是**如果紧接在计算所得项后面的是未手动赋值的项，那么它就会因为无法获得初始值而报错**：

```
enum Color {Red = "red".length, Green, Blue};

// index.ts(1,33): error TS1061: Enum member must have initializer.
// index.ts(1,40): error TS1061: Enum member must have initializer.
```

下面是常数项和计算所得项的完整定义，部分引用自[中文手册 - 枚举]:

当满足以下条件时，枚举成员被当作是常数：

- 不具有初始化函数并且之前的枚举成员是常数。在这种情况下，当前枚举成员的值为上一个枚举成员的值加 `1`。但第一个枚举元素是个例外。如果它没有初始化方法，那么它的初始值为 `0`。
- 枚举成员使用常数枚举表达式初始化。常数枚举表达式是 TypeScript 表达式的子集，它可以在编译阶段求值。当一个表达式满足下面条件之一时，它就是一个常数枚举表达式：
  - 数字字面量
  - 引用之前定义的常数枚举成员（可以是在不同的枚举类型中定义的）如果这个成员是在同一个枚举类型中定义的，可以使用非限定名来引用
  - 带括号的常数枚举表达式
  - `+`, `-`, `~` 一元运算符应用于常数枚举表达式
  - `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `>>>`, `&`, `|`, `^` 二元运算符，常数枚举表达式做为其一个操作对象。若常数枚举表达式求值后为 NaN 或 Infinity，则会在编译阶段报错

所有其它情况的枚举成员被当作是需要计算得出的值。

常数枚举

常数枚举是使用 `const enum` 定义的枚举类型：

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

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

常数枚举与普通枚举的区别是，它会在编译阶段被删除，并且不能包含计算成员。

上例的编译结果是：

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
```

假如包含了计算成员，则会在编译阶段报错：

```
const enum Color {Red, Green, Blue = "blue".length};

// index.ts(1,38): error TS2474: In 'const' enum declarations member initializer must be
constant expression.
```

外部枚举

外部枚举（Ambient Enums）是使用 `declare enum` 定义的枚举类型：

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

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

之前提到过，`declare` 定义的类型只会用于编译时的检查，编译结果中会被删除。

上例的编译结果是：

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

外部枚举与声明语句一样，常出现在声明文件中。

同时使用 `declare` 和 `const` 也是可以的：

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

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

编译结果：

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
```

> TypeScript 的枚举类型的概念来源于 C#。

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

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 {
    public name;
    constructor(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：

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

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

// index.ts(9,13): error TS2341: Property 'name' is private and only accessible within
class 'Animal'.
// index.ts(10,1): error TS2341: 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：

```ts
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：

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

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

// index.ts(11,17): error TS2341: 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');

// index.ts(7,19): TS2675: Cannot extend a class 'Animal'. Class constructor is marked as private.
// index.ts(13,9): TS2673: 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');
```

```
// index.ts(13,9): TS2674: Constructor of class 'Animal' is protected and only accessible
within the class declaration.
```

参数属性

修饰符和 `readonly` 还可以使用在构造函数参数中，等同于类中定义该属性同时给该属性赋值，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或构造函数中。

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

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

// index.ts(10,3): TS2540: 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:

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

let a = new Animal('Jack');

// index.ts(9,11): error TS2511: Cannot create an instance of the abstract class 'Animal'.
```

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：

```ts
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');

// index.ts(9,7): error TS2515: 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：

```ts
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

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(): void;
}

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(): void;
}

interface Light {
    lightOn(): void;
    lightOff(): void;
}

class Car implements Alarm, Light {
    alert() {
        console.log('Car alert');
    }
    lightOn() {
        console.log('Car light on');
```

```
    }
    lightOff() {
        console.log('Car light off');
    }
}
```

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：

```
interface Alarm {
    alert(): void;
}

interface LightableAlarm extends Alarm {
    lightOn(): void;
    lightOff(): void;
}
```

这很好理解，`LightableAlarm` 继承了 `Alarm`，除了拥有 `alert` 方法之外，还拥有两个新方法 `lightOn` 和 `lightOff`。

接口继承类

常见的面向对象语言中，接口是不能继承类的，但是在 TypeScript 中却是可以的：

```
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

为什么 TypeScript 会支持接口继承类呢?

实际上，当我们在声明 `class Point` 时，除了会创建一个名为 `Point` 的类之外，同时也创建了一个名为 `Point` 的类型（实例的类型）。

所以我们既可以将 `Point` 当做一个类来用（使用 `new Point` 创建它的实例）：

```
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

const p = new Point(1, 2);
```

也可以将 `Point` 当做一个类型来用（使用 `: Point` 表示参数的类型）：

```
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

function printPoint(p: Point) {
    console.log(p.x, p.y);
}

printPoint(new Point(1, 2));
```

这个例子实际上可以等价于：

```
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

interface PointInstanceType {
    x: number;
    y: number;
}

function printPoint(p: PointInstanceType) {
    console.log(p.x, p.y);
}

printPoint(new Point(1, 2));
```

上例中我们新声明的 `PointInstanceType` 类型，与声明 `class Point` 时创建的 `Point` 类型是等价的。

所以回到 `Point3d` 的例子中，我们就能很容易的理解为什么 TypeScript 会支持接口继承类了：

```
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

interface PointInstanceType {
    x: number;
    y: number;
}

// 等价于 interface Point3d extends PointInstanceType
interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

当我们声明 `interface Point3d extends Point` 时，`Point3d` 继承的实际上是类 `Point` 的实例的类型。

换句话说，可以理解为定义了一个接口 `Point3d` 继承另一个接口 `PointInstanceType`。

所以「接口继承类」和「接口继承接口」没有什么本质的区别。

值得注意的是，`PointInstanceType` 相比于 `Point`，缺少了 `constructor` 方法，这是因为声明 `Point` 类时创建的 `Point` 类型是不包含构造函数的。另外，除了构造函数是不包含的，静态属性或静态方法也是不包含的（实例的类型当然不应该包括构造函数、静态属性或静态方法）。

换句话说，声明 `Point` 类时创建的 `Point` 类型只包含其中的实例属性和实例方法：

```
class Point {
    /** 静态属性，坐标系原点 */
    static origin = new Point(0, 0);
    /** 静态方法，计算与原点距离 */
    static distanceToOrigin(p: Point) {
        return Math.sqrt(p.x * p.x + p.y * p.y);
    }
    /** 实例属性，x 轴的值 */
    x: number;
    /** 实例属性，y 轴的值 */
    y: number;
    /** 构造函数 */
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
    /** 实例方法，打印此点 */
    printPoint() {
```

```
        console.log(this.x, this.y);
    }
}

interface PointInstanceType {
    x: number;
    y: number;
    printPoint(): void;
}

let p1: Point;
let p2: PointInstanceType;
```

上例中最后的类型 `Point` 和类型 `PointInstanceType` 是等价的。

同样的，在接口继承类的时候，也只会继承它的实例属性和实例方法。

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。

如：

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']
```

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;
}

// index.ts(2,19): error TS2339: 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);

// index.ts(10,17): error TS2345: Argument of type '7' is not assignable to parameter of
type 'Lengthwise'.
```

多个类型参数之间也可以互相约束：

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

let x = { a: 1, b: 2, c: 3, d: 4 };

copyFields(x, { b: 10, d: 20 });
```

上例中，我们使用了两个类型参数，其中要求 `T` 继承 `U`，这样就保证了 `U` 上不会出现 `T` 中不存在的字段。

泛型接口

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; };
```

泛型参数的默认类型

在 TypeScript 2.3 以后，我们可以为泛型中的类型参数指定默认类型。当使用泛型时没有在代码中直接指定类型参数，从实际值参数中也无法推测出时，这个默认类型就会起作用。

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

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

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 {
    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**：

```ts
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;
}
```

```ts
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：

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

相当于：

```ts
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。