简介

TypeScript 是 JavaScript 的超集，是微软公司开发。主要功能是为 JavaScript 添加类型系统。

类型是人为添加的一种编程约束和用法提示。

在语法上，JavaScript 属于动态类型语言，TypeScript 引入了一个更强大、更严格的类型系统，属于静态类型语言。

1. 静态类型的优点

（1）有利于代码的静态分析。

不必运行代码，就可以确定变量的类型，从而推断代码有没有错误。这就叫做代码的静态分析。在开发阶段运行静态检查，就可以发现很多问题，避免交付有问题的代码，大大降低了线上风险。

（2）有利于发现错误。

每个值、每个变量、每个运算符都有严格的类型约束，TypeScript 就能轻松发现拼写错误、语义错误和方法调用错误，节省时间。

```
let obj = { message: '' };
console.log(obj.messege); // Property 'messege' does not exist on type '{ message:
string; }'. Did you mean 'message'?
```

```
const a = 0;
const b = true;
const result = a + b; // Operator '+' cannot be applied to types 'number' and
'boolean'.
```

```
function hello() {
  return 'hello world';
}

hello().find('hello'); // Property 'find' does not exist on type 'string'.
```

（3）更好的 IDE 支持，做到语法提示和自动补全。

IDE（比如 VSCode）一般都会利用类型信息，提供语法提示功能和自动补全功能（只键入一部分的变量名或函数名，编辑器补全后面的部分）。

（4）提供了代码文档。

类型信息可以部分替代代码文档，解释应该如何使用这些代码。

（5）有助于代码重构。

类型信息大大减轻了重构的成本。越是大型的、多人合作的项目，类型信息能够提供的帮助越大。

2. 静态类型的缺点

（1）丧失了动态类型的代码灵活性。

（2）增加了编程工作量。

不仅需要编写功能，还需要编写类型声明。

（3）更高的学习成本。

（4）引入了独立的编译步骤。

将 TypeScript 代码转成 JavaScript 代码，在 JavaScript 引擎运行。

（5）兼容性问题。

过去大部分 JavaScript 项目都没有做 TypeScript 适配。

TypeScript 不一定适合那些小型的、短期的个人项目。

基本用法

1. 类型声明

变量只有赋值后才能使用，否则就会报错。

```
let x:number;
console.log(x); // 在赋值前使用了变量“x”。
```

2. 类型推断

所有 JavaScript 代码都是合法的 TypeScript 代码。

TypeScript 的设计思想是，类型声明是可选的。即使不加类型声明，依然是有效的 TypeScript 代码，只是这时不能保证 TypeScript 会正确推断出类型。

这样设计还有一个好处，将以前的 JavaScript 项目改为 TypeScript 项目时，你可以逐步地为老代码添加类型，即使有些代码没有添加，也不会无法运行。

3. TypeScript 编译

JavaScript 的运行环境（浏览器和 Node.js）不认识 TypeScript 代码。所以，TypeScript 项目要想运行，必须先转为 JavaScript 代码，这个代码转换的过程就叫做“编译”（compile）。编译时，会将类型声明和类型相关的代码全部删除。**TypeScript 的类型检查只是编译时的类型检查，而不是运行时的类型检查。**

4. 值与类型

TypeScript 代码只涉及类型，不涉及值。所有跟“值”相关的处理，都由 JavaScript 完成。TypeScript 项目里面存在两种代码，一种是底层的“值代码”，另一种是上层的“类型代码”。前者使用 JavaScript 语法，后者使用 TypeScript 的类型语法。TypeScript 的编译过程，实际上就是把“类型代码”全部拿掉，只保留“值代码”。

5. TypeScript Playground

官网的在线编译 TypeScript Playground 。把 TypeScript 代码贴进文本框，它就会在当前页面自动编译出 JavaScript 代码，还可以在浏览器执行编译产物。如果编译报错，它也会给出详细的报错信息。

这个页面还具有支持完整的 IDE 支持，可以自动语法提示。此外，它支持把代码片段和编译器设置保存成 URL，分享给他人。

6. tsc 编译器

TypeScript 官方提供的编译器叫做 tsc，可以将 TypeScript 脚本编译成 JavaScript 脚本。本机想要编译 TypeScript 代码，必须安装 tsc。TypeScript 脚本文件使用 `.ts` 后缀名，JavaScript 脚本文件使用 `.js` 后缀名。tsc 的作用就是把 `.ts` 脚本转变成 `.js` 脚本。

7. ts-node 模块

ts-node 是一个非官方的 npm 模块，可以直接运行 TypeScript 代码。

使用时，可以先全局安装它。

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

安装后，就可以直接运行 TypeScript 脚本。

```
ts-node script.ts
```

上面命令运行了 TypeScript 脚本 script.ts，给出运行结果。

如果不安装 `ts-node`，也可以通过 npx 调用它来运行 TypeScript 脚本。

```
npx ts-node script.ts
```

上面命令中，npx 会在线调用 `ts-node`，从而在不安装的情况下，运行 `script.ts`。

如果执行 ts-node 命令不带有任何参数，它会提供一个 TypeScript 的命令行 REPL 运行环境，你可以在这个环境中输入 TypeScript 代码，逐行执行。

```
$ ts-node
>
```

上例中，单独运行ts-node命令，会给出一个大于号，这就是 TypeScript 的 REPL 运行环境，可以逐行输入代码运行。

```
$ ts-node
> const twice = (x:string) => x + x;
> twice('abcd')
'abcdabcd'
>
```

上例中，在 TypeScript 命令行 REPL 环境中，先输入一个函数 `twice`，然后调用该函数，就会得到结果。

要退出这个 REPL 环境，可以按下 `Ctrl + d`，或者输入 `.exit`。

如果只是想简单运行 TypeScript 代码看看结果，ts-node 不失为一个便捷的方法。

any 类型，unknown 类型，never 类型

1. any 类型

变量类型一旦设为 any，TypeScript 实际上会关闭这个变量的类型检查。即使有明显的类型错误，只要句法正确，都不会报错。

```
let x:any = 'hello';

x(1) // 不报错
x.foo = 100; // 不报错
```

应该尽量避免使用 any 类型，否则就失去了使用 TypeScript 的意义。

实际开发中，any 类型主要适用以下两个场合。

（1）出于特殊原因，需要关闭某些变量的类型检查，就可以把该变量的类型设为 any。

（2）为了适配以前老的 JavaScript 项目，让代码快速迁移到 TypeScript，可以把变量类型设为 any。有些年代很久的大型 JavaScript 项目，尤其是别人的代码，很难为每一行适配正确的类型，这时你为那些类型复杂的变量加上 any，TypeScript 编译时就不会报错。

any 类型可以看成是所有其他类型的全集，包含了一切可能的类型。TypeScript 将这种类型称为“顶层类型”（top type），意为涵盖了所有下层。

1.1. 类型推断问题

没有指定类型、TypeScript 必须自己推断类型的那些变量，如果无法推断出类型，TypeScript 就会认为该变量的类型是 any。**对于那些类型不明显的变量，一定要显式声明类型，防止被推断为 any。**

1.2. 污染问题

any 类型除了关闭类型检查，还有一个很大的问题，就是它会“污染”其他变量。它可以赋值给其他任何类型的变量（因为没有类型检查），导致其他变量出错。

```
let x:any = 'hello';
let y:number;

y = x; // 不报错

y * 123 // 不报错
y.toFixed() // 不报错
```

污染其他具有正确类型的变量，把错误留到运行时，这就是不宜使用 any 类型的另一个主要原因。

2. unknown 类型

为了解决 any 类型"污染"其他变量的问题，TypeScript 3.0 引入了 unknown 类型。它与 any 含义相同，表示类型不确定，可能是任意类型，但是它的使用有一些限制，不像 any 那样自由，可以视为严格版的 any。

2.1. unknown 和 any 相似之处

unknown 跟 any 的相似之处，在于所有类型的值都可以分配给 unknown 类型。

```
let x:unknown;

x = true; // 正确
x = 42; // 正确
x = 'Hello World'; // 正确
```

2.2. unknown 和 any 不同之处

- unknown 类型的变量，不能直接赋值给其他类型的变量。
- 不能直接调用 unknown 类型变量的方法和属性。
- unknown 类型变量能够进行的运算是有限的。

unknown 类型的变量，不能直接赋值给其他类型的变量（除了 any 类型和 unknown 类型）。克服了 any 类型"污染"其他变量问题的一大缺点。

```
let v:unknown = 123;

let v1:boolean = v; // Type 'unknown' is not assignable to type 'boolean'.
let v2:number = v; // Type 'unknown' is not assignable to type 'number'.
let v3:any = v;
let v4:unknown = v;
```

直接调用 unknown 类型变量的属性和方法，或者直接当作函数执行，都会报错。

```
let v1:unknown = { foo: 123 };
let a1:any = { foo: 123 };
v1.foo;  // 'v1' is of type 'unknown'.
a1.foo;

let v2:unknown = 'hello';
let a2:any = 'hello';
v2.trim(); // 'v2' is of type 'unknown'.
a2.trim();

let v3:unknown = (n = 0) => n + 1;
let a3:any = (n = 0) => n + 1;
v3(); // 'v3' is of type 'unknown'.
a3.foo;
```

unknown 类型变量能够进行的运算是有限的，只能进行比较运算（运算符==、===、!=、!==、||、&&、?）、取反运算（运算符!）、typeof 运算符和 instanceof 运算符这几种，其他运算都会报错。

```
let a:unknown = 1;

a + 1; // 'a' is of type 'unknown'.
a === 1;
```

怎么才能使用 unknown 类型变量呢?

只有经过"类型缩小"，unknown 类型变量才可以使用。所谓"类型缩小"，就是缩小 unknown 变量的类型范围，确保不会出错。

```
let a:unknown = 1;

if (typeof a === 'number') {
  let r = a + 10; // 正确
  console.log("r", r);
}
```

```
let s:unknown = 'hello';

if (typeof s === 'string') {
  s.length; // 正确
}
```

unknown 可以看作是更安全的 any。一般来说，凡是需要设为 any 类型的地方，通常都应该优先考虑设为 unknown类型。

在集合论上，unknown 也可以视为所有其他类型（除了any）的全集，所以它和 any 一样，也属于 TypeScript 的顶层类型。

3. never 类型

never 类型是"空类型"，即该类型为空，不包含任何值。

如果一个变量可能有多种类型（即联合类型），通常需要使用分支处理每一种类型。这时，处理所有可能的类型之后，剩余的情况就属于 never 类型。

```
function fn(x:string|number) {
  if (typeof x === 'string') {
    // ...
  } else if (typeof x === 'number') {
    // ...
  } else {
```

```
    x; // never 类型
  }
}
```

参数变量 `x` 可能是字符串，也可能是数值，判断了这两种情况后，剩下的最后那个 `else` 分支里面，`x` 就是 `never` 类型了。

`never` 类型的一个可以赋值给任意其他类型。

```
function f():never {
  throw new Error('Error');
}

let v1:number = f(); // 不报错
let v2:string = f(); // 不报错
let v3:boolean = f(); // 不报错
```

函数 `f()` 会抛错，所以返回值类型可以写成 `never`，即不可能返回任何值。各种其他类型的变量都可以赋值为 `f()` 的运行结果（`never` 类型）。

为什么 `never` 类型可以赋值给任意其他类型呢?

这也跟集合论有关，空集是任何集合的子集，任何类型都包含了 `never` 类型。因此，`never` 类型是任何其他类型所共有的，TypeScript 把这种情况称为"底层类型"（bottom type）。

TypeScript 有两个"顶层类型"（`any` 和 `unknown`），但是"底层类型"只有 `never` 唯一一个。

类型系统

TypeScript 包含 number、string、boolean、bigint、symbol、object、undefined、null 八种类型。

1. 基本类型

1.1. bigint 类型

bigint 类型包含所有的大整数。

```
const x:bigint = 123n;
const y:bigint = 0xffffn;
```

bigint 与 number 类型不兼容。

```
const x:bigint = 123; // Type 'number' is not assignable to type 'bigint'.
const y:bigint = 3.14; // Type 'number' is not assignable to type 'bigint'.
```

1.2. object 类型

`object` 类型包含了所有对象、数组和函数。

```
const x:object = { foo: 123 };
const y:object = [1, 2, 3];
const z:object = (n:number) => n + 1;
```

1.3. undefined 类型，null 类型

`undefined` 和 `null` 是两种独立类型，它们各自都只有一个值。

`undefined` 类型只包含一个值 `undefined`，设置成其他值都会报错。 表示未定义（即还未给出定义，以后可能会有定义）。

```
let x:undefined = undefined; // 正确

let x:undefined = '1'; // Type '"1"' is not assignable to type 'undefined'.
let x:undefined = 1; // Type '1' is not assignable to type 'undefined'.
let x:undefined = true; // Type 'true' is not assignable to type 'undefined'.
let x:null = Symbol(); // Type 'symbol' is not assignable to type 'undefined'.
```

上例中，变量 `x` 就属于 `undefined` 类型。两个 `undefined` 里面，第一个是类型，第二个是值。

`null` 类型也只包含一个值 `null`，设置成其他值都会报错。 表示为空（即此处没有值）。

```
const x:null = null; // 正确

let x:null = 1; // Type '1' is not assignable to type 'null'.
let x:null = '1'; // Type '"1"' is not assignable to type 'null'.
let x:null = true; // Type 'true' is not assignable to type 'null'.
let x:null = Symbol(); // Type 'symbol' is not assignable to type 'null'.
```

上例中，变量 `x` 就属于 `null` 类型。两个 `null` 里面，第一个是类型，第二个是值。

如果没有声明类型的变量，被赋值为 `undefined` 或 `null`，它们的类型会被推断为`any`。

```
let a = undefined;   // any
const b = undefined; // any

a = ''; // 可以被赋值为其他类型的值，证明 a 是 any 类型。b 不能被赋值为其他类型，因为 b 是常量。

let c = null;        // any
const d = null;      // any

c = true; // 可以被赋值为其他类型的值，证明 c 是 any 类型。d 不能被赋值为其他类型，因为 b 是常量。
```

2. 包装对象类型

JavaScript 的 8 种类型之中，`undefined` 和 `null` 其实是两个特殊值，`object` 属于复合类型，剩下的五种属于原始类型（primitive value），代表最基本的、不可再分的值。

- boolean
- string
- number
- bigint
- symbol

上面这五种原始类型的值，都有对应的包装对象（wrapper object）。所谓"包装对象"，指的是这些值在需要时，会自动产生的对象。

```
'hello'.charAt(1) // 'e'
```

上例中，字符串 `hello` 执行了 `charAt()` 方法。但是，在 JavaScript 语言中，只有对象才有方法，原始类型的值本身没有方法。这行代码之所以可以运行，**就是因为在调用方法时，字符串会自动转为包装对象**，`charAt()` 方法其实是定义在包装对象上。这样的设计大大方便了字符串处理，省去了将原始类型的值手动转成对象实例。

五种包装对象之中，symbol 类型和 bigint 类型无法直接获取它们的包装对象（即 `Symbol()` 和 `BigInt()` 不能作为构造函数使用），但是剩下三种可以。

- Boolean()
- String()
- Number()

这三个构造函数，执行后可以直接获取某个原始类型值的包装对象。

```
const s = new String('hello');
typeof s // 'object'
s.charAt(1) // 'e'

const s1 = String(1);
s1; // "1"
typeof s1; // "string"
```

上例中，`s` 就是字符串 `hello` 的包装对象，`typeof` 运算符返回 `object`，不是 `string`，但是本质上它还是字符串，可以使用所有的字符串方法。

`String()` 只有当作构造函数使用时（即带有 `new` 命令调用），才会返回包装对象。如果当作普通函数使用（不带有 `new` 命令），返回就是一个普通字符串。其他两个构造函数 `Number()` 和 `Boolean()` 也是如此。

```
let n0 = new Number('1');
n0; // 1
typeof n0; // "number"

let n1 = Number('1');
n1; // 1
typeof n1; // "number"

let b1 = Number(1);
b1; // true
typeof b1; // "boolean"

let b2 = Number(0);
b2; // false
typeof b2; // "boolean"
```

2.1. 包装对象类型和字面量类型

由于包装对象的存在，导致每一个原始类型的值都有包装对象和字面量两种情况。

```
'hello' // 字面量
new String('hello') // 包装对象
```

为了区分这两种情况，TypeScript 对五种原始类型分别提供了大写和小写两种类型。

- Boolean 和 boolean
- String 和 string
- Number 和 number
- BigInt 和 bigint
- Symbol 和 symbol

其中，大写类型同时包含包装对象和字面量两种情况，小写类型只包含字面量，不包含包装对象。

```
const s1:String = 'hello'; // 正确
const s2:String = new String('hello'); // 正确

const s3:string = 'hello'; // 正确
const s4:string = new String('hello'); // Type 'String' is not assignable to type
'string'. 'string' is a primitive, but 'String' is a wrapper object. Prefer using
'string' when possible.
```

上例中，`String` 类型可以赋值为字符串的字面量，也可以赋值为包装对象。但是，`string` 类型只能赋值为字面量，赋值为包装对象就会报错。

建议只使用小写类型，不使用大写类型。 因为绝大部分使用原始类型的场合，都是使用字面量，不使用包装对象。而且，TypeScript 把很多内置方法的参数，定义成小写类型，使用大写类型会报错。

```
const n1:number = 1;
const n2:Number = 1;

Math.abs(n1) // 1
Math.abs(n2) // Argument of type 'Number' is not assignable to parameter of type
'number'.'number' is a primitive, but 'Number' is a wrapper object. Prefer using
'number' when possible.
```

上例中，`Math.abs()` 方法的参数类型被定义成小写的 `number`，传入大写的 `Number` 类型就会报错。

3. Object 类型与 object 类型

TypeScript 的对象类型也有大写 `Object` 和小写 `object` 两种。

3.1. Object 类型

大写的Object类型代表 JavaScript 语言里面的广义对象。所有可以转成对象的值，都是Object类型，这囊括了几乎所有的值。**原始类型值、对象、数组、函数都是合法的 `Object` 类型。**

```
let obj:Object;

obj = true;
obj = 'hi';
```

```
obj = 1;
obj = { foo: 123 };
obj = [1, 2];
obj = (a:number) => a + 1;
```

除了 `undefined` 和 `null` 这两个值不能转为对象，其他任何值都可以赋值给Object类型。

```
let obj:Object;

obj = undefined; // Type 'undefined' is not assignable to type 'Object'.
obj = null; // Type 'null' is not assignable to type 'Object'.
```

空对象 `{}` 是 `Object` 类型的简写形式，所以使用 `Object` 时常常用空对象代替。

```
let obj:{};

obj = true;
obj = 'hi';
obj = 1;
obj = { foo: 123 };
obj = [1, 2];
obj = (a:number) => a + 1;

obj = undefined; // Type 'undefined' is not assignable to type 'Object'.
obj = null; // Type 'null' is not assignable to type 'Object'.
```

3.2. object 类型

小写的 `object` 类型代表 JavaScript 里面的狭义对象，即可以用字面量表示的对象，只包含对象、数组和函数，不包括原始类型的值。

```
let obj:object;

obj = { foo: 123 };
obj = [1, 2];
obj = (a:number) => a + 1;
obj = true; // Type 'boolean' is not assignable to type 'object'.
obj = 'hi'; // Type 'string' is not assignable to type 'object'.
obj = 1; // Type 'number' is not assignable to type 'object'.
```

大多数时候，我们使用对象类型，只希望包含真正的对象，不希望包含原始类型。所以，**建议总是使用小写类型object，不使用大写类型Object。**

无论是大写的 `Object` 类型，还是小写的 `object` 类型，都只包含 JavaScript 内置对象原生的属性和方法，用户自定义的属性和方法都不存在于这两个类型之中。

```
const o1:Object = { foo: 0 };
const o2:object = { foo: 0 };

o1.toString() // 正确
o1.foo // Property 'foo' does not exist on type 'Object'.

o2.toString() // 正确
o2.foo // Property 'foo' does not exist on type 'Object'.
```

对象类型需要明确定义属性：

```
const o1:{foo: number} = { foo: 0 };
const o2:{foo: number} = { foo: 0 };

o1.toString() // 正确
o1.foo // 正确

o2.toString() // 正确
o2.foo // 正确
```

4. 值类型

TypeScript 规定，单个值也是一种类型，称为“值类型”。

```
var c = 'world'; // var c: string
let a = 'hello'; // let a: string
const b = 'hello'; // const b: "hello"
```

上面例子中，没有显示定义类型，TypeScript 将使用类型推断。使用 `var` 和 `let` 定义的变量，类型推断为 `string`，当使用 `const` 定义的变量，类型推断为值类型 `hello`。

```
let x:'hello';

x = 'hello'; // 正确
x = 'world'; // Type '"hello2"' is not assignable to type '"hello"'.
```

上例中，变量 `x` 的类型是字符串 `hello`，导致它只能赋值为这个字符串，赋值为其他字符串就会报错。

TypeScript 推断类型时，遇到 `const` 命令声明的变量，如果代码里面没有注明类型，就会推断该变量是值类型。

```
// x 的类型是 "https"
const x = 'https';
```

```
// y 的类型是 string
const y:string = 'https';
```

上例中，变量x是const命令声明的，TypeScript 就会推断它的类型是值 `https`，而不是 `string` 类型。这样推断是合理的，因为 const 命令声明的变量，一旦声明就不能改变，相当于常量。值类型就意味着不能赋为其他值。

const命令声明的变量，如果赋值为对象，并不会推断为值类型。

```
// x 的类型是 { foo: number }
const x = { foo: 1 };
```

上例中，变量x没有被推断为值类型，而是推断属性 `foo` 的类型是 `number`。这是因为 JavaScript 里面，`const` 变量赋值为对象时，属性值是可以改变的。

值类型可能会出现一些很奇怪的报错。

```
const x:5 = 4 + 1; // Type 'number' is not assignable to type '5'.
```

上例中，等号左侧的类型是数值 `5`，等号右侧 `4 + 1` 的类型为 `number`。由于 5 是 `number` 的子类型，父类型不能赋值给子类型。反过来是可以的，子类型可以赋值给父类型。

```
let x:5 = 5; // 值类型 '5'
let y:number = 4 + 1; // number 类型

x = y; // Type 'number' is not assignable to type '5'.
y = x; // 正确
```

如果一定要让子类型可以赋值为父类型的值，就要用到类型断言。

```
const x:5 = (4 + 1) as 5; // 正确
```

上例中，在 `4 + 1` 后面加上 `as 5`，实现了类型断言，告诉编译器可以把 `4 + 1` 的类型视为值类型 `5`，这样就不会报错了。

5. 联合类型

联合类型（union types）指的是多个类型组成的一个新类型，使用符号 `|` 表示。联合类型 `A|B` 表示，任何一个类型只要属于 A或B，就属于联合类型 A|B。

```
let x:string|number;
```

```
x = 123; // 正确
x = 'abc'; // 正确
```

联合类型可以与值类型相结合，表示一个变量的值有若干种可能。

```
let setting:true|false;
let gender:'male'|'female';
let rainbowColor:'赤'|'橙'|'黄'|'绿'|'青'|'蓝'|'紫';

rainbowColor = '1'; // Type '"1"' is not assignable to type '"赤" | "橙" | "黄" |
"绿" | "青" | "蓝" | "紫"'.
```

上面的示例都是由值类型组成的联合类型，非常清晰地表达了变量的取值范围。其中，`true|false` 其实就是布尔类型 `boolean`。

前面提到，打开编译选项strictNullChecks后，其他类型的变量不能赋值为undefined或null。这时，如果某个变量确实可能包含空值，就可以采用联合类型的写法。

```
let name1:string|null;

name1 = 'John';
name1 = null;
```

联合类型的第一个成员前面，也可以加上竖杠|，这样便于多行书写。

```
let x:
  | 'one'
  | 'two'
  | 'three'
  | 'four';
```

如果一个变量有多种类型，读取该变量时，往往需要进行“类型缩小”（type narrowing），区分该值到底属于哪一种类型，然后再进一步处理。

```
function printId(
  id:number|string
) {
    console.log(id.toUpperCase()); // Property 'toUpperCase' does not exist on
type 'number'.
}
```

`toUpperCase()` 方法只存在于字符串，不存在于数值。解决方法就是对参数 `id` 做一下类型缩小，确定它的类型以后再进行处理。

```
function printId(
  id:number|string
) {
  if (typeof id === 'string') {
    console.log(id.toUpperCase());
  } else {
    console.log(id);
  }
}
```

“类型缩小”是 TypeScript 处理联合类型的标准方法，凡是遇到可能为多种类型的场合，都需要先缩小类型，再进行处理。 实际上，联合类型本身可以看成是一种“类型放大”（type widening），处理时就需要“类型缩小”（type narrowing）。

下面是“类型缩小”的另一个例子。

```
function getPort(
  scheme: 'http'|'https'
) {
  switch (scheme) {
    case 'http':
      return 80;
    case 'https':
      return 443;
  }
}
```

上例中，函数体内部对参数变量 `scheme` 进行类型缩小，根据不同的值类型，返回不同的结果。

6. 交叉类型

交叉类型（intersection types）指的多个类型组成的一个新类型，使用符号 `&` 表示。交叉类型 `A&B` 表示，任何一个类型必须同时属于A和B，才属于交叉类型 `A&B`，即交叉类型同时满足A和B的特征。交叉类型的主要用途是表示对象的合成。

```
let obj:
  { foo: string } &
  { bar: string };

obj = {
  foo: 'hello',
  bar: 'world'
};
```

变量 `obj` 同时具有属性 foo 和属性 `bar`。

交叉类型常常用来为对象类型添加新属性。下面为类型 B 增加了属性 bar。

```
type A = { foo: number };
type B = A & { bar: number };
const bbb:B = {foo: 1, bar: 2};
```

7. type 命令

type命令用来定义一个类型的别名。

```
type Age = number;
let age:Age = 55;
```

`type` 命令为 `number` 类型定义了一个别名 `Age`。这样就能像使用 `number` 一样，使用 `Age` 作为类型。

别名可以让类型的名字变得更有意义，也能增加代码的可读性，还可以使复杂类型用起来更方便，便于以后修改变量的类型。别名不允许重名。

```
type Color = 'red';
type Color = 'blue'; // Duplicate identifier 'Color'.
```

别名的作用域是块级作用域。这意味着，代码块内部定义的别名，影响不到外部。

```
type Color = 'red';

if (Math.random() < 0.5) {
  type Color = 'blue'; // if代码块内部的类型别名Color，跟外部的Color是不一样的。
}
```

别名支持使用表达式，也可以在定义一个别名时，使用另一个别名，即别名允许嵌套。

```
type World = "world";
type Greeting = `hello ${World}`;
```

上例中，别名 `Greeting` 使用了模板字符串，读取另一个别名 `World`。

`type` 命令属于类型相关的代码，编译成 JavaScript 的时候，会被全部删除。

8. typeof 运算符

JavaScript 语言中，typeof 运算符是一个一元运算符，返回一个字符串，代表操作数的类型。**这时 typeof 的操作数是一个值。**

```
typeof 'foo'; // 'string'
```

JavaScript 里面，typeof运算符只可能返回八种结果，而且都是字符串。

```
typeof undefined; // "undefined"
typeof true; // "boolean"
typeof 1337; // "number"
typeof "foo"; // "string"
typeof {}; // "object"
typeof parseInt; // "function"
typeof Symbol(); // "symbol"
typeof 127n // "bigint"
```

TypeScript 将 `typeof` 运算符移植到了类型运算，它的操作数依然是一个值，但是返回的不是字符串，而是该值的 TypeScript 类型。

```
const a = { x: 0 };
type T0 = typeof a;   // { x: number }
type T1 = typeof a.x; // number
```

`typeof a` 表示返回变量 `a` 的 TypeScript 类型（`{ x: number }`）。同理，`typeof a.x` 返回的是属性 `x` 的类型（`number`）。

这种用法的 `typeof` 返回的是 TypeScript 类型，所以只能用在类型运算之中（即跟类型相关的代码之中），不能用在值运算。

也就是说，同一段代码可能存在两种 `typeof` 运算符，一种用在值相关的 JavaScript 代码部分，另一种用在类型相关的 TypeScript 代码部分。

```
let a = 1;
let b:typeof a;

if (typeof a === 'number') {
  b = a;
}
```

上例中，用到了两个typeof，第一个是类型运算，第二个是值运算。

JavaScript 的 typeof 遵守 JavaScript 规则，TypeScript 的 typeof 遵守 TypeScript 规则。它们的一个重要区别在于，编译后，前者会保留，后者会被全部删除。

typeof 的参数只能是标识符，不能是需要运算的表达式。

```
type T = typeof Date(); // Expression expected.

type T = typeof Date;
```

typeof 的参数不能是一个值的运算式，而 Date() 需要运算才知道结果。

另外，typeof 命令的参数不能是类型。

```
type Age = number;
type MyAge = typeof Age; // 'Age' only refers to a type, but is being used as a
value here.
 //Exported type alias 'MyAge' has or is using private name 'Age'.
```

typeof 是一个很重要的 TypeScript 运算符，有些场合不知道某个变量 foo 的类型，这时使用 typeof foo 就可以获得它的类型。

9. 块级类型声明

TypeScript 支持块级类型声明，即类型可以声明在代码块（用大括号表示）里面，并且只在当前代码块有效。

```
if (true) {
  type T = number;
  let v:T = 5;
} else {
  type T = string;
  let v:T = 'hello';
}
```

上例中，存在两个代码块，其中分别有一个类型T的声明。这两个声明都只在自己的代码块内部有效，在代码块外部无效。

10. 类型的兼容

TypeScript 的类型存在兼容关系，某些类型可以兼容其他类型。

```
type T = number|string;

let a:number = 1;
let b:T = a;
```

变量 a 赋值给变量 b 并不会报错，b 的类型兼容 a 的类型。

如果类型 A 的值可以赋值给类型 B，那么类型 A 就称为类型 B 的子类型（subtype）。在上例中，类型 number 就是类型 number|string 的子类型。

TypeScript 的一个规则是，凡是可以使用父类型的地方，都可以使用子类型，但是反过来不行。

```
let a:'hi' = 'hi';
let b:string = 'hello';

b = a; // 正确
a = b; // Type 'string' is not assignable to type '"hi"'. 类型 'string' 不能赋值给
类型 'hi'
```

`hi` 类型是 `string` 类型的子类型，`string` 是 `hi` 的父类型。所以，变量 `a` 可以赋值给变量 `b`，但是反过来就会报错。

之所以有这样的规则，是因为子类型继承了父类型的所有特征，所以可以用在父类型的场合。但是，子类型还可能有一些父类型没有的特征，所以父类型不能用在子类型的场合。

数组

TypeScript 数组有一个根本特征：所有成员的类型必须相同，但是成员数量是不确定的，可以是无限数量的成员，也可以是零成员。

数组类型有两种写法：

```
let arr1:number[] = [1, 2, 3];
let arr2:(number|string)[] = [1, 2, 3];

let arr3: Array<number> = [1, 2, 3];
let arr4: Array<number|string> = [1, 2, 3];
```

由于成员数量可以动态变化，所以 TypeScript 不会对数组边界进行检查，越界访问数组并不会报错。

```
let arr:number[] = [1, 2, 3];
let foo = arr[3]; // 正确
```

TypeScript 允许使用方括号读取数组成员的类型。

```
type Names = string[];
type Name = Names[0]; // string
```

上例中，类型 `Names` 是字符串数组，那么 `Names[0]` 返回的类型就是 `string`。

由于数组成员的索引类型都是 `number`，所以读取成员类型也可以写成下面这样。

```
type Names = string[];
type Name = Names[number]; // string
```

上例中，`Names[number]` 表示数组 Names 所有数值索引的成员类型，所以返回 `string`。

1. 数组类型推断

如果数组变量没有声明类型，TypeScript 就会推断数组成员的类型。这时，推断行为会因为值的不同，而有所不同。

如果变量的初始值是空数组，那么 TypeScript 会推断数组类型是 `any[]`。

```
// 推断为 any[]
const arr = [];
```

后面，为这个数组赋值时，TypeScript 会自动更新类型推断。

```
const arr = [];
arr // 推断为 any[]

arr.push(123);
arr // 推断类型为 number[]

arr.push('abc');
arr // 推断类型为 (string|number)[]
```

上例中，数组变量 `arr` 的初始值是空数组，然后随着新成员的加入，TypeScript 会自动修改推断的数组类型。

类型推断的自动更新只发生初始值为空数组的情况

如果初始值不是空数组，类型推断就不会更新。

```
// 推断类型为 number[]
const arr = [123];

arr.push('abc'); // Argument of type 'string' is not assignable to parameter of
type 'number'.
```

上例中，数组变量 `arr` 的初始值是 `[123]`，TypeScript 就推断成员类型为 `number`。新成员如果不是这个类型，TypeScript 就会报错，而不会更新类型推断。

2. 只读数组

TypeScript 允许声明只读数组，方法是在数组类型前面加上 `readonly` 关键字。

```
const arr:readonly number[] = [0, 1];

arr[1] = 2; // 报错
arr.push(3); // 报错
delete arr[0]; // 报错
```

上例中，`arr` 是一个只读数组，删除、修改、新增数组成员都会报错。

TypeScript 将 `readonly number[]` 与 `number[]` 视为两种不一样的类型，后者是前者的子类型。这是因为只读数组没有 `pop()`、`push()` 之类会改变原数组的方法，所以 `number[]` 的方法数量要多于 `readonly number[]`，这意味着 `number[]` 其实是 `readonly number[]` 的子类型。

子类型继承了父类型的所有特征，并加上了自己的特征，所以子类型 `number[]` 可以用于所有使用父类型的场合，反过来就不行。

```
let a1:number[] = [0, 1];
let a2:readonly number[] = a1; // 正确

a1 = a2; // The type 'readonly number[]' is 'readonly' and cannot be assigned to
the mutable type 'number[]'.
```

上例中，子类型 `number[]` 可以赋值给父类型 `readonly number[]`，但是反过来就会报错。只读类型不可分配给可变类型。

由于只读数组是数组的父类型，所以它不能代替数组。这一点很容易产生令人困惑的报错。

```
function getSum(s:number[]) {
  // ...
}

const arr:readonly number[] = [1, 2, 3];

getSum(arr) // The type 'readonly number[]' is 'readonly' and cannot be assigned
to the mutable type 'number[]'.
```

`readonly` 关键字不能与数组的泛型写法一起使用。

```
const arr:readonly Array<number> = [0, 1]; // 'readonly' type modifier is only
permitted on array and tuple literal types.
```

TypeScript 提供了两个专门的泛型，用来生成只读数组的类型。

```
const a1:ReadonlyArray<number> = [0, 1];

const a2:Readonly<number[]> = [0, 1];
```

泛型 `ReadonlyArray<T>` 和 `Readonly<T[]>` 都可以用来生成只读数组类型。两者尖括号里面的写法不一样，`Readonly<T[]>` 的尖括号里面是整个数组（`number[]`），而 `ReadonlyArray<T>` 的尖括号里面是数组成员（`number`）。

3. const 断言

只读数组还有一种声明方法，就是使用“const 断言”。

```
const arr = [0, 1] as const;

arr[0] = [2]; // Cannot assign to '0' because it is a read-only property.
```

4. 多维数组

TypeScript 使用 `T[][]` 的形式，表示二维数组，`T` 是最底层数组成员的类型。

```
let multi1:number[][] = [[1,2,3], [23,24,25]];
let multi2:Array<Array<number>> = [[1,2,3], [23,24,25]];
```

上例中，变量multi的类型是number[][]，表示它是一个二维数组，最底层的数组成员类型是number。

元组

元组（tuple）是 TypeScript 特有的数据类型，JavaScript 没有单独区分这种类型。它表示成员类型可以自由设置的数组，即数组的各个成员的类型可以不同。

元组必须明确声明每个成员的类型。

```
const s:[string, string, boolean] = ['a', 'b', true];
```

元组 `s` 的前两个成员的类型是 `string`，最后一个成员的类型是 `boolean`。

数组的成员类型写在方括号外面（`number[]`），元组的成员类型是写在方括号里面（`[number]`）。成员类型写在方括号里面的就是元组，写在外面的就是数组。

```
let a:[number] = [1];
```

变量 `a` 是一个元组，只有一个成员，类型是 `number`。

使用元组时，必须明确给出类型声明（上例的 `[number]`），不能省略，否则 TypeScript 会把一个值自动推断为数组。

```
// a 的类型为 (number | boolean)[]
let a = [1, true];
```

上例中，变量 `a` 的值其实是一个元组，但是 TypeScript 会将其推断为一个联合类型的数组，即 `a` 的类型为 `(number | boolean)[]`。

元组成员的类型可以添加问号后缀（`?`），表示该成员是可选的。

```
let a:[number, number?] = [1];
```

上例中，元组 `a` 的第二个成员是可选的，可以省略。

问号只能用于元组的尾部成员，也就是说，所有可选成员必须在必选成员之后。

```
type myTuple = [
  number,
  number,
  number?,
  string?
];
```

上例中，元组myTuple的最后两个成员是可选的。也就是说，它的成员数量可能有两个、三个和四个。

由于需要声明每个成员的类型，所以大多数情况下，元组的成员数量是有限的，从类型声明就可以明确知道，元组包含多少个成员，越界的成员会报错。

```
let x:[string, string] = ['a', 'b'];

x[2] = 'c'; // Tuple type '[string, string]' of length '2' has no element at index
'2'.
```

上例中，变量 x 是一个只有两个成员的元组，如果对第三个成员赋值就报错了。

使用扩展运算符（...），可以表示不限成员数量的元组。

```
type NamedNums = [
  string,
  ...number[]
];

const a:NamedNums = ['A', 1, 2];
const b:NamedNums = ['B', 1, 2, 3];
```

上例中，元组类型 NamedNums 的第一个成员是字符串，后面的成员使用扩展运算符来展开一个数组，从而实现了不定数量的成员。

扩展运算符用在元组的任意位置都可以，但是它后面只能是数组或元组。

```
// 上例中，扩展运算符分别在元组的尾部、中部和头部。
type t1 = [string, number, ...boolean[]];
type t2 = [string, ...boolean[], number];
type t3 = [...boolean[], string, number];

const t1: t1 = ['1', 1, true, false, true, true];
const t2: t2 = ['1', true, false, false, 1];
const t3: t3 = [false, true, '1', 1];
```

如果不确定元组成员的类型和数量，可以写成下面这样。

```
type Tuple = [...any[]];
```

上例中，元组Tuple可以放置任意数量和类型的成员。但是这样写，也就失去了使用元组和 TypeScript 的意义。

元组可以通过方括号，读取成员类型。

```
type Tuple = [string, number];
type Age = Tuple[1]; // number
type Name = Tuple[0]; // string
let age: Age = 18;
let nickName: Name = 'lisi';
```

上例中，`Tuple[1]` 返回 `1` 号位置的成员类型 `number`，`Tuple[0]` 返回 `0` 号位置的成员类型 `string`。

由于元组的成员都是数值索引，即索引类型都是 `number`，所以可以像下面这样读取。

```
type Tuple = [string, number, Date];
type TupleEl = Tuple[number];  // string|number|Date

let a1: TupleEl = 'a';
let a2: TupleEl = 1;
let a3: TupleEl = new Date();
```

上例中，`Tuple[number]` 表示元组 `Tuple` 的所有数值索引的成员类型，所以返回 `string|number|Date`，即这个类型是三种值的联合类型。

2. 只读元组

元组也可以是只读的，不允许修改，有两种写法。

```
// 写法一
type t = readonly [number, string]

// 写法二
type t = Readonly<[number, string]>
```

跟数组一样，只读元组是元组的父类型。所以，元组可以替代只读元组，而只读元组不能替代元组。

```
type t1 = readonly [number, number];
type t2 = [number, number];

let x:t2 = [1, 2];
let y:t1 = x; // 正确

x = y; // The type 't1' is 'readonly' and cannot be assigned to the mutable type
't2'.
```

由于只读元组不能替代元组，所以会产生一些令人困惑的报错。

```
function distanceFromOrigin([x, y]:[number, number]) {
  return Math.sqrt(x**2 + y**2);
}

let point = [3, 4] as const;

distanceFromOrigin(point); // 报错
```

上例中，函数distanceFromOrigin()的参数是一个元组，传入只读元组就会报错，因为只读元组不能替代元组。

上例中[3, 4] as const的写法，在上一章讲到，生成的是只读数组，其实生成的同时也是只读元组。因为它生成的实际上是一个只读的“值类型”readonly [3, 4]，把它解读成只读数组或只读元组都可以。

上面示例报错的解决方法，就是使用类型断言，在最后一行将传入的参数断言为普通元组。

```
distanceFromOrigin(
  point as [number, number]
)
```

3. 成员数量的推断

如果没有可选成员和扩展运算符，TypeScript 会推断出元组的成员数量（即元组长度）。

```
function f(point: [number, number]) {
  if (point.length === 3) {  // This comparison appears to be unintentional
because the types '2' and '3' have no overlap.
    // ...
  }
}
```

上面示例会报错，原因是 TypeScript 发现元组 `point` 的长度是 2，不可能等于 3，这个判断无意义。

如果包含了可选成员，TypeScript 会推断出可能的成员数量。

```
function f(
  point:[number, number?, number?]
) {
  if (point.length === 4) {  //  This comparison appears to be unintentional
because the types '1 | 2 | 3' and '4' have no overlap.这种比较似乎是无意的，因为类
型'1 | 2 | 3'和'4'没有重叠。
    // ...
  }
}
```

如果使用了扩展运算符，TypeScript 就无法推断出成员数量。

```
const myTuple:[...string[]] = ['a', 'b', 'c'];
if (myTuple.length === 4) { // 正确
  // ...
}
```

上例中，myTuple只有三个成员，但是 TypeScript 推断不出它的成员数量，因为它的类型用到了扩展运算符，TypeScript 把myTuple当成数组看待，而数组的成员数量是不确定的。

一旦扩展运算符使得元组的成员数量无法推断，TypeScript 内部就会把该元组当成数组处理。

4. 扩展运算符与成员数量

扩展运算符（...）将数组（不是元组）转换成一个逗号分隔的序列，这时 TypeScript 会认为这个序列的成员数量是不确定的，因为数组的成员数量是不确定的。

这导致如果函数调用时，使用扩展运算符传入函数参数，可能发生参数数量与数组长度不匹配的报错。

```
const arr = [1, 2];
function add(x:number, y:number){
  // ...
}
add(...arr) // A spread argument must either have a tuple type or be passed to a rest parameter.扩展参数必须具有元组类型，或者传递给rest形参。
```

上面示例会报错，原因是函数add()只能接受两个参数，但是传入的是 `...arr`，TypeScript 认为转换后的参数个数是不确定的。

需要改成下面这样：

```
const t: [number, number] = [1, 2];
function add(x: number, y: number) {
  // ...
}
add(...t); // 正确
```

另一种写法是使用as const断言。

```
const arr = [1, 2] as const;
```

上面这种写法也可以，因为 TypeScript 会认为 `arr` 的类型是 `readonly [1, 2]`，这是一个只读的值类型，可以当作数组，也可以当作元组。

symbol 类型

Symbol 是 ES2015 新引入的一种原始类型的值。它类似于字符串，但是每一个 Symbol 值都是独一无二的，与其他任何值都不相等。

`Symbol` 值通过 `Symbol()` 函数生成。在 TypeScript 里面，Symbol 的类型使用 `symbol` 表示。

```
let x:symbol = Symbol();
let y:symbol = Symbol();

x === y // false
```

上例中，变量 `x` 和 `y` 的类型都是 `symbol`，且都用 `Symbol()` 生成，但是它们是不相等的，因为他们都是独一无二的。

1. unique symbol

symbol类型包含所有的 Symbol 值，但是无法表示某一个具体的 Symbol 值。

比如，5是一个具体的数值，就用5这个字面量来表示，这也是它的值类型。但是，Symbol 值不存在字面量，必须通过变量来引用，所以写不出只包含单个 Symbol 值的那种值类型。

为了解决这个问题，TypeScript 设计了symbol的一个子类型unique symbol，它表示单个的、某个具体的 Symbol 值。

因为unique symbol表示单个值，所以这个类型的变量是不能修改值的，只能用const命令声明，不能用let声明。

```
const x:unique symbol = Symbol();
let y:unique symbol = Symbol(); // A variable whose type is a 'unique symbol' type must be 'const'.类型为“唯一符号”类型的变量必须为“const”。
```

`const` 命令为变量赋值 `Symbol` 值时，变量类型默认就是 `unique symbol`，所以类型可以省略不写。

```
const x:unique symbol = Symbol();
// 等同于
const x = Symbol();
```

每个声明为unique symbol类型的变量，它们的值都是不一样的，其实属于两个值类型。

```
const a:unique symbol = Symbol();
const b:unique symbol = Symbol();
```

```
a === b // This comparison appears to be unintentional because the types 'typeof
a' and 'typeof b' have no overlap.这种比较似乎是无意的，因为类型'typeof a'和'typeof
b'没有重叠。
```

上例中，变量 a 和变量 b 的类型虽然都是 unique symbol，但其实是两个值类型。不同类型的值肯定是不相等的，所以最后一行就报错了。

而且，由于变量a和b是两个类型，就不能把一个赋值给另一个。

```
const a:unique symbol = Symbol();
const b:unique symbol = a; // 报错
```

上例中，变量 a 和变量 b 的类型都是 unique symbol，但是其实类型不同，所以把 a 赋值给 b 会报错。

上例变量b的类型，如果要写成与变量a同一个unique symbol值类型，只能写成类型为typeof a。

```
const a:unique symbol = Symbol();
const b:typeof a = a; // 正确
```

unique symbol 类型是 symbol 类型的子类型，所以可以将前者赋值给后者，但是反过来就不行。

```
const a:unique symbol = Symbol();
const b:symbol = a; // 正确
const c:unique symbol = b; // Type 'symbol' is not assignable to type 'unique
symbol'.
```

unique symbol 类型的一个作用，就是用作属性名，这可以保证不会跟其他属性名冲突。 如果要把某一个特定的 Symbol 值当作属性名，那么它的类型只能是 unique symbol，不能是 symbol。

```
const x:unique symbol = Symbol();
const y:symbol = Symbol();

interface Foo {
  [x]: string; // 正确
  [y]: string; // A computed property name in an interface must refer to an
expression whose type is a literal type or a 'unique symbol' type.接口中的计算属性
名称必须引用其类型为文字类型或“唯一符号”类型的表达式。
}
```

unique symbol类型也可以用作类（class）的属性值，但只能赋值给类的readonly static属性。

```
class C {
  static readonly foo:unique symbol = Symbol();
```

```
}
```

上例中，静态只读属性 `foo` 的类型就是 `unique symbol`。注意，这时 `static` 和 `readonly` 两个限定符缺一不可，这是为了保证这个属性是固定不变的。

2. 类型推断

如果变量声明时没有给出类型，TypeScript 会推断某个 Symbol 值变量的类型。

let命令声明的变量，推断类型为 symbol。

```
// 类型为 symbol
let x = Symbol();
```

const命令声明的变量，推断类型为 unique symbol。

```
// 类型为 unique symbol
const x = Symbol();
```

但是，const命令声明的变量，如果赋值为另一个 symbol 类型的变量，则推断类型为 symbol。

```
let x = Symbol();

// 类型为 symbol
const y = x;
```

let 命令声明的变量，如果赋值为另一个 `unique symbol` 类型的变量，则推断类型还是 `symbol`。

```
const x = Symbol();

// 类型为 symbol
let y = x;
```

函数

函数的实际参数个数，可以少于类型指定的参数个数，但是不能多于，即 TypeScript 允许省略参数。

```
let myFunc: (a:number, b:number) => number;
myFunc = (a:number) => a; // 正确
myFunc = (a:number, b:number, c:number) => a + b + c; // 报错
```

上例中，变量 `myFunc` 的类型只能接受两个参数，如果被赋值为只有一个参数的函数，并不报错。但是，被赋值为有三个参数的函数，就会报错。

这是因为 JavaScript 函数在声明时往往有多余的参数，实际使用时可以只传入一部分参数。比如，数组的 `forEach()` 方法的参数是一个函数，该函数默认有三个参数 `(item, index, array) => void`，实际上往往只使用第一个参数 `(item) => void`。因此，TypeScript 允许函数传入的参数不足。

```
let x = (a:number) => 0;
let y = (b:number, s:string) => 0;

y = x; // 正确
x = y; // Target signature provides too few arguments. Expected 2 or more, but got
1.
```

```
interface myFn {
  (a:number, b:number): number;
}
var add:myFn = (a, b) => a + b;
```

上例中，`interface` 命令定义了接口 `myFn`，这个接口的类型就是一个用对象表示的函数。

1. Function 类型

TypeScript 提供 `Function` 类型表示函数，任何函数都属于这个类型。

```
function doSomething(f:Function) {
  return f(1, 2, 3);
}
```

上例中，参数 `f` 的类型就是 `Function`，代表这是一个函数。`Function` 类型的值都可以直接执行。`Function` 类型的函数可以接受任意数量的参数，每个参数的类型都是 `any`，返回值的类型也是 `any`，代表没有任何约束，**所以不建议使用 `Function` 这个类型，给出函数详细的类型声明会更好。**

```
type Person = { name: string };
const people = ['alice', 'bob', 'jan'].map((name):Person => ({name}));
```

上例中，`Person` 是一个类型别名，代表一个对象，该对象有属性 `name`。变量 `people` 是数组的 `map()` 方法的返回值。`map()` 方法的参数是一个箭头函数 `(name):Person => ({name})`，该箭头函数的参数 `name` 的类型省略了，因为可以从 `map()` 的类型定义推断出来，箭头函数的返回值类型为 `Person`。相应地，变量 `people` 的类型是 `Person[]`。

2. 可选参数

如果函数的某个参数可以省略，则在参数名后面加问号表示。

```
function f(x?:number) {
  // ...
}
f(); // OK
f(10); // OK
```

参数名带有问号，表示该参数的类型实际上是 `原始类型|undefined`，它有可能为 `undefined` 。比如，上例的 `x` 虽然类型声明为 `number`，但是实际上是 `number|undefined`。

```
function f(x?:number) {
  return x;
}
f(undefined); // 正确
```

上例中，参数 `x` 是可选的，等同于说 `x` 可以赋值为 `undefined`。但是，反过来就不成立，类型显式设为 `undefined` 的参数，就不能省略。

```
function f(x:number|undefined) {
  return x;
}
f() // 报错
```

上例中，参数 `x` 的类型是 `number|undefined`，表示要么传入一个数值，要么传入 `undefined`，如果省略这个参数，就会报错。

函数的可选参数只能在参数列表的尾部，跟在必选参数的后面。

```
let myFunc: (a?:number, b:number) => number; // A required parameter cannot follow
an optional parameter. 必选参数不能跟在可选参数后面。
```

如果前部参数有可能为空，这时只能显式注明该参数类型可能为 `undefined`。

```
let myFunc:
  (
    a:number|undefined,
    b:number
  ) => number;
```

上例中，参数 `a` 有可能为空，就只能显式注明类型包括 `undefined`，传参时也要显式传入 `undefined`。函数体内部用到可选参数时，需要判断该参数是否为 `undefined`。

```
let myFunc: (a:number, b?:number) => number;

myFunc = function (x, y) {
  if (y === undefined) {
    return x;
  }
  return x + y;
}
```

上例中，由于函数的第二个参数为可选参数，所以函数体内部需要判断一下，该参数是否为空。

3. 参数默认值

设置了默认值的参数，就是可选的。如果不传入该参数，它就会等于默认值。

```
function createPoint(
  x:number = 0,
  y:number = 0
):[number, number] {
  return [x, y];
}
createPoint() // [0, 0]
```

上例中，参数 `x` 和 `y` 的默认值都是 `0`，调用 `createPoint()` 时，这两个参数都是可以省略的。这里其实可以省略 `x` 和 `y` 的类型声明，因为可以从默认值推断出来。

```
function createPoint(
  x = 0, y = 0
) {
  return [x, y];
}
```

可选参数与默认值不能同时使用。

```
// 报错
function f(x?: number = 0) { // Parameter cannot have question mark and
initializer. 形参不能有问号和初始化值
  // ...
}
```

设有默认值的参数，如果传入 `undefined`，也会触发默认值。

```
function f(x = 456) {
  return x;
}
f(undefined) // 456
```

具有默认值的参数如果不位于参数列表的末尾，调用时不能省略，如果要触发默认值，必须显式传入 undefined。

```
function add(
  x:number = 0,
  y:number
) {
  return x + y;
}
add(1) // 报错
add(undefined, 1) // 正确
```

4. readonly 只读参数

如果函数内部不能修改某个参数，可以在函数定义时，在参数类型前面加上 `readonly` 关键字，表示这是只读参数。

```
function arraySum(
  arr:readonly number[]
) {
  // ...
  arr[0] = 0; // 报错
}
```

上例中，参数 `arr` 的类型是 `readonly number[]`，表示为只读参数。如果函数体内部修改这个数组，就会报错。

5. void 类型

void 类型表示函数没有返回值。

```
function f():void {
  console.log('hello');
}
```

上例中，函数 `f` 没有返回值，类型就要写成 `void`。如果返回其他值，就会报错。

```
function f():void {
  return 123; // Type 'number' is not assignable to type 'void'.
}
```

`void` 类型允许返回 `undefined`。

```
function f():void {
  return undefined; // 正确
}
function f():void {
  return null; // Type 'null' is not assignable to type 'void'.
}
```

如果变量、对象方法、函数参数是一个返回值为 void 类型的函数，那么并不代表不能赋值为有返回值的函数。恰恰相反，该变量、对象方法和函数参数可以接受返回任意值的函数，这时并不会报错。

```
type voidFunc = () => void;
const f:voidFunc = () => {
  return 123;
};
```

上例中，变量 `f` 的类型是 `voidFunc`，是一个没有返回值的函数。但是实际上，`f` 的值可以是一个有返回值的函数（返回 `123`），编译时不会报错。

这是因为，这时 TypeScript 认为，这里的 void 类型只是表示该函数的返回值没有利用价值，或者说不应该使用该函数的返回值。只要不用到这里的返回值，就不会报错。

这样设计是有现实意义的。举例来说，数组方法 `Array.prototype.forEach(fn)` 的参数 `fn` 是一个函数，而且这个函数应该没有返回值，即返回值类型是 `void`。

但是，实际应用中，很多时候传入的函数是有返回值，但是它的返回值不重要，或者不产生作用。

```
const src = [1, 2, 3];
const ret = [];
src.forEach(el => ret.push(el));
```

上例中，`push()` 有返回值，表示新的数组长度。但是，对于 `forEach()` 方法来说，这个返回值是没有作用的，根本用不到，所以 TypeScript 不会报错。

如果后面使用了这个函数的返回值，就违反了约定，则会报错。

```
type voidFunc = () => void;
const f:voidFunc = () => {
  return 123;
};
f() * 2 // The left-hand side of an arithmetic operation must be of type 'any',
'number', 'bigint' or an enum type. 算术运算的左边必须是“any”、“number”、“bigint”或
枚举类型。
```

上例中，最后一行报错了，因为根据类型声明，**f()没有返回值，但是却用到了它的返回值，因此报错了。**

注意，这种情况仅限于变量、对象方法和函数参数，函数字面量如果声明了返回值是 void 类型，还是不能有返回值。

```
function f():void {
  return true; // 报错
}
const f3 = function ():void {
  return true; // 报错
};
```

上例中，函数字面量声明了返回 `void` 类型，这时只要有返回值（除了 `undefined` 和 `null`）就会报错。

函数的运行结果如果是抛错，也允许将返回值写成 `void`。

```
function throwErr():void {
  throw new Error('something wrong');
}
```

上例中，函数 `throwErr()` 会抛错，返回值类型写成 `void` 是允许的。

6. 局部类型

函数内部允许声明其他类型，该类型只在函数内部有效，称为局部类型。

```
function hello(txt:string) {
  type message = string;
  let newTxt:message = 'hello ' + txt;
  return newTxt;
}
const newTxt:message = hello('world'); // Cannot find name 'message'. Exported
```

```
variable 'newTxt' has or is using private name 'message'. 找不到名称“message”。 导出变量'newTxt'已经或正在使用私有名称'message'。
```

上例中，类型 `message` 是在函数 `hello()` 内部定义的，只能在函数内部使用。在函数外部使用，就会报错。

7. 高阶函数

一个函数的返回值还是一个函数，那么前一个函数就称为高阶函数（higher-order function）。

下面箭头函数返回的还是一个箭头函数：

```
(someValue: number) => (multiplier: number) => someValue * multiplier;
```

8. 函数重载

有些函数可以接受不同类型或不同个数的参数，并且根据参数的不同，会有不同的函数行为。这种根据参数类型不同，执行不同逻辑的行为，称为函数重载（function overload）。

```
reverse('abc') // 'cba'
reverse([1, 2, 3]) // [3, 2, 1]
```

上例中，函数 `reverse()` 可以将参数颠倒输出。参数可以是字符串，也可以是数组。这意味着，该函数内部有处理字符串和数组的两套逻辑，**根据参数类型的不同，分别执行对应的逻辑，这就叫“函数重载”。**

```
function reverse(str:string):string;
function reverse(arr:any[]):any[];
```

上例中，分别对函数 `reverse()` 的两种参数情况，给予了类型声明。但是，到这里还没有结束，后面还必须对函数 `reverse()` 给予完整的类型声明。

```
function reverse(str:string):string;
function reverse(arr:any[]):any[];
function reverse(
  stringOrArray:string|any[]
):string|any[] {
  if (typeof stringOrArray === 'string')
    return stringOrArray.split('').reverse().join('');
  else
    return stringOrArray.slice().reverse();
}
```

上例中，前两行类型声明列举了重载的各种情况。第三行是函数本身的类型声明，它必须与前面已有的重载声明兼容。

有一些编程语言允许不同的函数参数，对应不同的函数实现。但是，JavaScript 函数只能有一个实现，必须在这个实现当中，处理不同的参数。因此，函数体内部就需要判断参数的类型及个数，并根据判断结果执行不同的操作。

```
function add(
  x:number,
  y:number
):number;
function add(
  x:any[],
  y:any[]
):any[];
function add(
  x:number|any[],
  y:number|any[]
):number|any[] {
  if (typeof x === 'number' && typeof y === 'number') {
    return x + y;
  } else if (Array.isArray(x) && Array.isArray(y)) {
    return [...x, ...y];
  }

  throw new Error('wrong parameters');
}
```

上例中，函数 `add()` 内部使用 `if` 代码块，分别处理参数的两种情况。 **重载的各个类型描述与函数的具体实现之间，不能有其他代码，否则报错。**

另外，虽然函数的具体实现里面，有完整的类型声明。但是，函数实际调用的类型，以前面的类型声明为准。比如，上例的函数实现，参数类型和返回值类型都是 `number|any[]` ，但不意味着参数类型为 `number` 时返回值类型为 `any[]`。

函数重载的每个类型声明之间，以及类型声明与函数实现的类型之间，不能有冲突。

```
// This overload signature is not compatible with its implementation signature.
此重载签名与其实现签名不兼容。
function fn(x:boolean):void;
function fn(x:string):void;
function fn(x:number|string) {
  console.log(x);
}
```

重载声明的排序很重要，因为 TypeScript 是按照顺序进行检查的，一旦发现符合某个类型声明，就不再往下检查了， **类型最宽的声明应该放在最后面，防止覆盖其他类型声明。**

```
function f(x:any):number;
function f(x:string): 0|1;
```

```
function f(x:any):any {
  // ...
}
const a:0|1 = f('hi'); // Type 'number' is not assignable to type '0 | 1'. 'x' is
declared but its value is never read. 类型'number'不能赋值给类型'0 | 1'。 声明
了'x'，但永远不会读取它的值。
```

上面声明中，第一行类型声明 `x:any` 范围最宽，导致函数 `f()` 的调用都会匹配这行声明，无法匹配第二行类型声明，所以最后一行调用就报错了，因为等号两侧类型不匹配，左侧类型是0|1，右侧类型是number。这个函数重载的正确顺序是，第二行类型声明放到第一行的位置。

对象的方法也可以使用重载。

```
class StringBuilder {
  #data = '';

  add(num:number): this;
  add(bool:boolean): this;
  add(str:string): this;
  add(value:any): this {
    this.#data += String(value);
    return this;
  }

  toString() {
    return this.#data;
  }
}
```

上例中，方法 `add()` 也使用了函数重载。

函数重载也可以用来精确描述函数参数与返回值之间的对应关系。

```
function createElement(
  tag:'a'
):HTMLAnchorElement;
function createElement(
  tag:'canvas'
):HTMLCanvasElement;
function createElement(
  tag:'table'
):HTMLTableElement;
function createElement(
  tag:string
):HTMLElement {
  // ...
}
```

上例中，函数重载精确描述了参数 `tag` 的三个值，所对应的不同的函数返回值。

这个示例的函数重载，也可以用对象表示。

```
type CreateElement = {
  (tag:'a'): HTMLAnchorElement;
  (tag:'canvas'): HTMLCanvasElement;
  (tag:'table'): HTMLTableElement;
  (tag:string): HTMLElement;
}
```

由于重载是一种比较复杂的类型声明方法，为了降低复杂性，一般来说，如果可以的话，应该优先使用联合类型替代函数重载。

```
// 写法一：  函数重载
function len(s:string):number;
function len(arr:any[]):number;
function len(x:any):number {
  return x.length;
}

// 写法二：联合类型
function len(x:any[]|string):number {
  return x.length;
}
```

上例中，写法二使用联合类型，要比写法一的函数重载简单很多。

9., 构造函数

JavaScript 语言使用构造函数，生成对象的实例。**构造函数的最大特点，就是必须使用 `new` 命令调用。**

```
const d = new Date();
```

上例中，`Date()` 就是一个构造函数，使用 `new` 命令调用，返回 `Date` 对象的实例。

构造函数的类型写法，就是在参数列表前面加上 `new` 命令。

```
class Animal {
  numLegs:number = 4;
}
type AnimalConstructor = new () => Animal;
function create(c:AnimalConstructor):Animal {
  return new c();
}
const a = create(Animal);
```

上面示例中，类型AnimalConstructor就是一个构造函数，而函数create()需要传入一个构造函数。 **在JavaScript 中，类（class）本质上是构造函数**，所以Animal这个类可以传入create()。

构造函数还有另一种类型写法，就是采用对象形式。

```
type F = { new (s:string): object; };
```

上例中，类型 `F` 就是一个构造函数。类型写成一个可执行对象的形式，并且在参数列表前面要加上 `new` 命令。

某些函数既是构造函数，又可以当作普通函数使用，比如 `Date()`。这时，类型声明可以写成下面这样。

```
type F = {
  new (s:string): object;
  (n?:number): number;
}
```

上例中，`F` 既可以当作普通函数执行，也可以当作构造函数使用。

对象类型

除了原始类型，对象是 JavaScript 最基本的数据结构。TypeScript 对于对象类型有很多规则。

对象属性的类型可以用分号结尾，也可以用逗号结尾。最后一个属性后面，可以写分号或逗号，也可以不写。

一旦声明了类型，对象赋值时，就不能缺少指定的属性，也不能有多余的属性。

```
type MyObj = {
  x:number;
  y:number;
};
const o1:MyObj = { x: 1 }; // 报错
const o2:MyObj = { x: 1, y: 1, z: 1 }; // 报错
```

上例中，变量 o1 缺少了属性 y，变量 o2 多出了属性 z，都会报错。

读写不存在的属性也会报错。

```
const obj:{
  x:number;
  y:number;
} = { x: 1, y: 1 };
console.log(obj.z); // Property 'z' does not exist on type '{ x: number; y:
number; }'.
obj.z = 1; // Property 'z' does not exist on type '{ x: number; y: number; }'.
```

同样地，也不能删除类型声明中存在的属性，修改属性值是可以的。

```
const myUser = {
  name: "Sabrina",
};
myUser.name = "Cynthia"; // 正确
delete myUser.name // The operand of a 'delete' operator must be optional.
'delete'操作符的操作数必须是可选的。
```

上面声明中，删除类型声明中存在的属性name会报错，但是可以修改它的值。

对象的方法使用函数类型描述。

```
const obj:{
  x: number;
  y: number;
  add(x:number, y:number): number;
```

```
  // 或者写成
  // add: (x:number, y:number) => number;
} = {
  x: 1,
  y: 1,
  add(x, y) {
    return x + y;
  }
};
```

对象类型可以使用方括号读取属性的类型。

```
type User = {
  name: string,
  age: number
};
type Name = User['name']; // type Name = string
type Age = User.age; // User 是类型不是命名空间
```

除了 `type` 命令可以为对象类型声明一个别名，TypeScript 还提供了 `interface` 命令，可以把对象类型提炼为一个接口。

```
// 写法一
type MyObj = {
  x:number;
  y:number;
};
const obj:MyObj = { x: 1, y: 1 };

// 写法二
interface MyObj {
  x: number;
  y: number;
}
const obj:MyObj = { x: 1, y: 1 };
```

TypeScript 不区分对象自身的属性和继承的属性，一律视为对象的属性。

```
interface MyInterface {
  toString(): string; // 继承的属性
  prop: number; // 自身的属性
}
const obj:MyInterface = { // 正确
  prop: 123,
};
// 或者
const obj:MyInterface = { // 正确
```

```
  toString: () => '122',
  prop: 123,
};
```

1. 可选属性

如果某个属性是可选的（即可以忽略），需要在属性名后面加一个问号。

```
const obj: {
  x: number;
  y?: number;
} = { x: 1 };
```

可选属性等同于允许赋值为 `undefined`，下面两种写法是等效的。

```
type User = {
  firstName: string;
  lastName?: string;
}; // type User = { firstName: string; lastName?: string | undefined; }

// 等价于
type User = {
  firstName: string;
  lastName?: string|undefined;
}; // type User = { firstName: string; lastName?: string | undefined; }
```

上例中，类型 `User` 的可选属性 `lastName` 可以是字符串，也可以是 `undefined`，即可选属性可以赋值为 `undefined`。

```
const obj: {
  x: number;
  y?: number;
} = { x: 1, y: undefined };
```

上例中，可选属性 `y` 赋值为 `undefined`，不会报错。

同样地，读取一个没有赋值的可选属性时，返回 `undefined`。

```
type MyObj = {
  x: string,
  y?: string
};
const obj:MyObj = { x: 'hello' };
obj.y.toLowerCase() // 'obj.y' is possibly 'undefined'.
```

所以，读取可选属性之前，必须检查一下是否为undefined。

```
const user:{
  firstName: string;
  lastName?: string;
} = { firstName: 'Foo'};
if (user.lastName !== undefined) {
  console.log(`hello ${user.firstName} ${user.lastName}`)
}
```

上例中，`lastName` 是可选属性，需要判断是否为 `undefined` 以后，才能使用。

```
// 写法一
let firstName = (user.firstName === undefined) ? 'Foo' : user.firstName;
let lastName = (user.lastName === undefined) ? 'Bar' : user.lastName;

// 写法二
let firstName = user.firstName ?? 'Foo';
let lastName = user.lastName ?? 'Bar';
```

可选属性与允许设为 `undefined` 的必选属性是不等价的。 可选属性可以设置该属性，也可以不设置。但允许设置为 `undefined` 的属性必须设置，只是可以设置为 `undefined`。

```
type A = { x:number, y?:number };
type B = { x:number, y:number|undefined };

const ObjA:A = { x: 1 }; // 正确
const ObjB:B = { x: 1 }; // Property 'y' is missing in type '{ x: number; }' but
required in type 'B'.
```

2. 只读属性

属性名前面加上 `readonly` 关键字，表示这个属性是只读属性，不能修改。

```
interface MyInterface {
  readonly prop: number;
}
```

上例中，`prop` 属性是只读属性，不能修改它的值。

```
const person:{
  readonly age: number
} = { age: 20 };
```

```
person.age = 21; // Cannot assign to 'age' because it is a read-only property. 不能赋值给'age'，因为它是一个只读属性。
```

只读属性只能在对象初始化期间赋值，此后就不能修改该属性。

```
type Point = {
  readonly x: number;
  readonly y: number;
};
const p:Point = { x: 0, y: 0 };
p.x = 100; // Cannot assign to 'x' because it is a read-only property.
```

如果属性值是一个对象，`readonly` 修饰符并不禁止修改该对象的属性，只是禁止完全替换掉该对象。

```
interface Home {
  readonly resident: {
    name: string;
    age: number
  };
}
const h:Home = {
  resident: {
    name: 'Vicky',
    age: 42
  }
};
h.resident.age = 32; // 正确
h.resident = {
  name: 'Kate',
  age: 23
} // Cannot assign to 'resident' because it is a read-only property.
```

上例中，`h.resident` 是只读属性，它的值是一个对象。修改这个对象的 `age` 属性是可以的，但是整个替换掉 `h.resident` 属性会报错。

另一个需要注意的地方是，如果一个对象有两个引用，即两个变量对应同一个对象，其中一个变量是可写的，另一个变量是只读的，那么从可写变量修改属性，会影响到只读变量。

```
interface Person {
  name: string;
  age: number;
}
interface ReadonlyPerson {
  readonly name: string;
  readonly age: number;
}
let w:Person = {
```

```
  name: 'Vicky',
  age: 42,
};
let r:ReadonlyPerson = w;
w.age += 1;
r.age // 43
r.age = 45; // Cannot assign to 'age' because it is a read-only property.
```

上例中，变量 `w` 和 `r` 指向同一个对象，其中 `w` 是可写的，`r` 是只读的。那么，对 `w` 的属性修改，会影响到 `r`。

如果希望属性值是只读的，除了声明时加上 `readonly` 关键字，还有一种方法，就是在赋值时，在对象后面加上只读断言 `as const`。

```
const myUser = {
  name: "Sabrina",
} as const;
myUser.name = "Cynthia"; // Cannot assign to 'name' because it is a read-only
property.
```

上例中，对象后面加了只读断言 `as const`，就变成只读对象了，不能修改属性了。

上面的 `as const` 属于 TypeScript 的类型推断，如果变量明确地声明了类型，那么 TypeScript 会以声明的类型为准。

```
const myUser:{ name: string } = {
  name: "Sabrina",
} as const;
myUser.name = "Cynthia"; // 正确

const myUser2:{ readonly name: string } = {
  name: "Sabrina",
};
myUser2.name = "Cynthia"; // Cannot assign to 'name' because it is a read-only
property.
```

上例中，根据变量 `myUser` 的类型声明，`name` 不是只读属性，但是赋值时又使用只读断言 `as const`。这时会以声明的类型为准，因为 `name` 属性可以修改。如果类型声明时明确了属性是 `readonly` 的便不可修改.

3. 属性名的索引类型

如果对象的属性非常多，一个个声明类型就很麻烦，而且有些时候，无法事前知道对象会有多少属性，比如外部 API 返回的对象。这时 TypeScript 允许采用属性名表达式的写法来描述类型，称为"属性名的索引类型"。

索引类型里面，最常见的就是属性名的字符串索引。

```
type MyObj = {
  [property: string]: string
```

```
};
const obj:MyObj = {
  foo: 'a',
  bar: 'b',
  baz: 'c',
};
```

上例中，类型 `MyObj` 的属性名类型就采用了表达式形式，写在方括号里面。`[property: string]` 的 `property` 表示属性名，这个是可以随便起的，它的类型是 `string`，即属性名类型为 `string`。也就是说，**不管这个对象有多少属性，只要属性名为字符串，且属性值也是字符串，就符合这个类型声明。**

JavaScript 对象的属性名（即上例的property）的类型有三种可能，除了上例的string，还有number和symbol。

```
type T1 = {
  [property: number]: string
};
type T2 = {
  [property: symbol]: string
};
```

上例中，对象属性名的类型分别为 `number` 和 `symbol`。

```
type MyArr = {
  [n:number]: number;
};
const arr:MyArr = [1, 2, 3];
// 或者
const arr:MyArr = {
  0: 1,
  1: 2,
  2: 3,
};
```

上例中，对象类型 `MyArr` 的属性名是 `[n:number]`，就表示它的属性名都是数值，比如 0、1、2。

对象可以同时有多种类型的属性名索引，比如同时有数值索引和字符串索引。但是，数值索引不能与字符串索引发生冲突，必须服从后者，这是因为在 JavaScript 语言内部，所有的数值属性名都会自动转为字符串属性名。

```
type MyType = {
  [x: number]: boolean; // 'number' index type 'boolean' is not assignable to
'string' index type 'string'. 'number'索引类型'boolean'不能赋值给'string'索引类
型'string'。
  [x: string]: string;
}
```

上例中，类型 `MyType` 同时有两种属性名索引，但是数值索引与字符串索引冲突了，所以报错了。由于字符属性名的值类型是 `string`，数值属性名的值类型只有同样为 `string`，才不会报错。

同样地，可以既声明属性名索引，也声明具体的单个属性名。如果单个属性名符合属性名索引的范围，两者不能有冲突，否则报错。

```
type MyType = {
  foo: boolean; // Property 'foo' of type 'boolean' is not assignable to 'string'
index type 'string'.
  [x: string]: string;
}
```

上例中，属性名 `foo` 符合属性名的字符串索引，但是两者的属性值类型不一样，所以报错了。

属性的索引类型写法，建议谨慎使用，因为属性名的声明太宽泛，约束太少。 **属性名的数值索引不宜用来声明数组，因为采用这种方式声明数组，就不能使用各种数组方法以及length属性，因为类型里面没有定义这些东西。**

```
type MyArr = {
  [n:number]: number;
};
const arr:MyArr = [1, 2, 3];
arr.length // Property 'length' does not exist on type 'MyArr'.
// 改成下面这样就正确
type MyArr = {
  [n:string]: number;
};
const arr:MyArr = { 0: 1, 1: 2, 2: 3, length: 3};
arr.length;
```

上例中，读取 `arr.length` 属性会报错，因为类型 `MyArr` 没有这个属性。

4. 解构赋值

解构赋值用于直接从对象中提取属性。

`const {id, name, price} = product;` 语句从对象 `product` 提取了三个属性，并声明属性名的同名变量。

解构赋值的类型写法，跟为对象声明类型是一样的。

```
const {id, name, price}:{
  id: string;
  name: string;
  price: number
} = product;
```

目前没法为解构变量指定类型，因为对象解构里面的冒号，JavaScript 指定了其他用途。

```
let { x: foo, y: bar } = obj;
// 等同于
let foo = obj.x;
let bar = obj.y;
```

上例中，冒号不是表示属性 x 和 y 的类型，而是为这两个属性指定新的变量名。如果要为 x 和 y 指定类型，不得不写成下面这样。

```
let { x: foo, y: bar } : { x: string; y: number } = obj;
```

这一点要特别小心，TypeScript 里面很容易搞糊涂。

```
function draw({
  shape: Shape,
  xPos: number = 100,
  yPos: number = 100
}) {
  let myShape = shape; // 报错
  let x = xPos; // 报错
}
```

上例中，函数 draw() 的参数是一个对象解构，里面的冒号很像是为变量指定类型，其实是为对应的属性指定新的变量名。所以，TypeScript 就会解读成，函数体内不存在变量 shape，而是属性 shape 的值被赋值给了变量 Shape。

5. 结构类型原则

只要对象 B 满足对象 A 的结构特征，TypeScript 就认为对象 B 兼容对象 A 的类型，这称为“结构类型”原则（structural typing）。

```
type A = {
  x: number;
};
type B = {
  x: number;
  y: number;
};
```

上面示例中，对象 A 只有一个属性 x，类型为 number。对象 B 满足这个特征，因此兼容对象 A，只要可以使用A的地方，就可以使用 B。

```
type A = {
  x: number;
};
const B = {
  x: 1,
  y: 1
};
const a:A = B; // 正确
```

根据“结构类型”原则，TypeScript 检查某个值是否符合指定类型时，并不是检查这个值的类型名（即“名义类型”），而是检查这个值的结构是否符合要求（即“结构类型”）。

TypeScript 之所以这样设计，是为了符合 JavaScript 的行为。JavaScript 并不关心对象是否严格相似，只要某个对象具有所要求的属性，就可以正确运行。

如果类型 `B` 可以赋值给类型 `A`，TypeScript 就认为 `B` 是 `A` 的子类型（`subtype`），`A` 是 `B` 的父类型。**子类型满足父类型的所有结构特征，同时还具有自己的特征。凡是可以使用父类型的地方，都可以使用子类型，即子类型兼容父类型。**

这种设计有时会导致令人惊讶的结果。

```
type myObj = {
  x: number,
  y: number,
};

function getSum(obj:myObj) {
  let sum = 0;
  for (const n of Object.keys(obj)) {
    const v = obj[n]; // Element implicitly has an 'any' type because expression of type 'string' can't be used to index type 'myObj'. No index signature with a parameter of type 'string' was found on type 'myObj'. 元素隐式具有'any'类型，因为'string'类型的表达式不能用于索引'myObj'类型。在类型'myObj'上找不到带有'string'类型参数的索引签名。
    sum += Math.abs(v);
  }
  return sum;
}
```

上例中，函数 `getSum()` 要求传入参数的类型是 `myObj`，但是实际上所有与 `myObj` 兼容的对象都可以传入。这会导致 `const v = obj[n]` 这一行报错，原因是 `obj[n]` 取出的属性值不一定是数值（number），使得变量 `v` 的类型被推断为 `any`。如果项目设置为不允许变量类型推断为 `any`，代码就会报错。写成下面这样，就不会报错。

```
type MyObj = {
  x: number,
  y: number,
};
```

```
function getSum(obj:MyObj) {
  return Math.abs(obj.x) + Math.abs(obj.y);
}
```

上例中，因为函数体内部只使用了属性 `x` 和 `y`，这两个属性有明确的类型声明，保证 `obj.x` 和 `obj.y` 肯定是数值。虽然与 `MyObj` 兼容的任何对象都可以传入函数 `getSum()`，但是只要不使用其他属性，就不会有类型报错。

6. 严格字面量检查

如果对象使用字面量表示，会触发 TypeScript 的严格字面量检查（strict object literal checking）。如果字面量的结构跟类型定义的不一样（比如多出了未定义的属性），就会报错。

```
const point:{
  x:number;
  y:number;
} = {
  x: 1,
  y: 1,
  z: 1 // Type '{ x: number;  y: number;  z: number;  }' is not assignable to type
'{ x: number;  y: number;  }'.
};
```

如果等号右边不是字面量，而是一个变量，根据结构类型原则，是不会报错的。

```
const myPoint = {
  x: 1,
  y: 1,
  z: 1
};
const point:{
  x:number;
  y:number;
} = myPoint; // 正确
```

上例中，等号右边是一个变量，就不会触发严格字面量检查，从而不报错。

TypeScript 对字面量进行严格检查的目的，主要是防止拼写错误。一般来说，字面量大多数来自手写，容易出现拼写错误，或者误用 API。

```
type Options = {
  title:string;
  darkMode?:boolean;
};
const obj:Options = {
  title: '我的网页',
```

```
  darkmode: true, // 报错
};
```

上例中，属性 darkMode 拼写错了，成了 darkmode。如果没有严格字面量规则，就不会报错，因为 darkMode 是可选属性，根据结构类型原则，任何对象只要有 title 属性，都认为符合 Options 类型。

规避严格字面量检查，可以使用中间变量。

```
let myOptions = {
  title: '我的网页',
  darkmode: true,
};
const obj:Options = myOptions;
```

上例中，创建了一个中间变量 myOptions，就不会触发严格字面量规则，因为这时变量 obj 的赋值，不属于直接字面量赋值。

也可以使用类型断言规避严格字面量检查。

```
const obj:Options = {
  title: '我的网页',
  darkmode: true,
} as Options;
```

上例中，使用类型断言 as Options，告诉编译器，字面量符合 Options 类型，就能规避这条规则。

如果允许字面量有多余属性，可以像下面这样在类型里面定义一个通用属性。

```
let x: {
  foo: number,
  [x: string]: any
};
x = { foo: 1, baz: 2 };
```

上例中，变量 x 的类型声明里面，有一个属性的字符串索引（[x: string]），导致任何字符串属性名都是合法的。

由于严格字面量检查，字面量对象传入函数必须很小心，不能有多余的属性。

```
interface Point {
  x: number;
  y: number;
}
function computeDistance(point: Point) { /*...*/ }
```

```
computeDistance({ x: 1, y: 2, z: 3 }); // 报错
computeDistance({x: 1, y: 2}); // 正确
```

上例中，对象字面量传入函数 `computeDistance()` 时，不能有多余的属性，否则就通不过严格字面量检查。

为了避免这种情况，TypeScript 2.4 引入了一个“最小可选属性规则”，也称为“弱类型检测”（weak type detection）。

```
type Options = {
  a?:number;
  b?:number;
  c?:number;
};
const opts = { d: 123 };
const obj:Options = opts; // 报错
```

如果某个类型的所有属性都是可选的，那么该类型的对象必须至少存在一个可选属性，不能所有可选属性都不存在。这就叫做“最小可选属性规则”。

如果想规避这条规则，要么在类型里面增加一条索引属性（[propName: string]: someType），要么使用类型断言（opts as Options）。

7. 空对象

```
const obj = {};
obj.prop = 123; // 报错
```

上例中，变量 `obj` 的值是一个空对象，然后对 `obj.prop` 赋值就会报错。原因是推断变量 `obj` 的类型为空对象，实际执行的是 `const obj:{} = {};`

空对象没有自定义属性，所以对自定义属性赋值就会报错。空对象只能使用继承的属性，即继承自原型对象Object.prototype的属性。

obj.toString() // 正确

上面示例中，toString()方法是一个继承自原型对象的方法，TypeScript 允许在空对象上使用。

这种写法其实在 JavaScript 很常见：先声明一个空对象，然后向空对象添加属性。但是，**TypeScript 不允许动态添加属性，所以对象不能分步生成，必须生成时一次性声明所有属性。**

```
// 错误
const pt = {};
pt.x = 3;
pt.y = 4;

// 正确
const pt = {
```

```
  x: 3,
  y: 4
};
```

如果确实需要分步声明，一个比较好的方法是，使用扩展运算符（...）合成一个新对象。

```
const pt0 = {};
const pt1 = { x: 3 };
const pt2 = { y: 4 };
const pt = {
  ...pt0, ...pt1, ...pt2
};
```

上例中，对象 pt 是三个部分合成的，这样既可以分步声明，也符合 TypeScript 静态声明的要求。

空对象作为类型，其实是Object类型的简写形式。

```
let d:{};
// 等同于
// let d:Object;

d = {};
d = { x: 1 };
d = 'hello';
d = 2;
```

上例中，各种类型的值（除了 null 和 undefined）都可以赋值给空对象类型，跟 Object 类型的行为是一样的。

因为 Object 可以接受各种类型的值，而空对象是 Object 类型的简写，所以它不会有严格字面量检查，赋值时总是允许多余的属性，只是不能读取这些属性。

```
interface Empty { }
const b:Empty = {myProp: 1, anotherProp: 2}; // 正确
b.myProp // Property 'myProp' does not exist on type 'Empty'.
```

上例中，变量 b 的类型是空对象，视同 Object 类型，不会有严格字面量检查，但是读取多余的属性会报错。

如果想强制使用没有任何属性的对象，可以采用下面的写法。

```
interface WithoutProperties {
  [key: string]: never;
}
const a:WithoutProperties = { prop: 1 }; // Type 'number' is not assignable to
type 'never'.
```

上例中，`[key: string]: never` 表示字符串属性名是不存在的，因此其他对象进行赋值时就会报错。

interface

interface(接口) 是对象的模板，使用了某个模板的对象，就拥有了指定的类型结构。

```
interface Person {
  firstName: string;
  lastName: string;
  age: number;
}
```

上面示例中，定义了一个接口 `Person`，它指定一个对象模板，拥有三个属性 `firstName`、`lastName` 和 `age`。任何实现这个接口的对象，都必须部署这三个属性，并且必须符合规定的类型。

只要指定该接口作为对象的类型即可实现该接口。

```
const p:Person = {
  firstName: 'John',
  lastName: 'Smith',
  age: 25
};
```

上例中，变量 `p` 的类型就是接口 `Person`，所以必须符合 `Person` 指定的结构。

方括号运算符可以取出 `interface` 某个属性的类型。

```
interface Foo {
  a: string;
}
type A = Foo['a']; // type A = string
```

上例中，Foo['a']返回属性 `a` 的类型，所以类型 `A` 就是 `string`。

`interface` 可以表示对象的各种语法，它的成员有 `5` 种形式。

- 对象属性
- 对象的属性索引
- 对象方法
- 函数
- 构造函数

（1）对象属性

```
interface Point {
  x: number;
```

```
  y: number;
}
```

上例中，x 和 y 都是对象的属性，分别使用冒号指定每个属性的类型。属性之间使用分号或逗号分隔，最后一个属性结尾的分号或逗号可以省略。

如果属性是可选的，就在属性名后面加一个问号。

```
interface Foo {
  x?: string;
}
```

如果属性是只读的，需要加上readonly修饰符。

```
interface A {
  readonly a: string;
}

const x: A = { a : '1' };
x.a = '2'; // Cannot assign to 'a' because it is a read-only property.
```

（2）对象的属性索引

```
interface A {
  [prop: string]: number;
}
```

上面示例中，[prop: string] 就是属性的字符串索引，表示属性名只要是字符串，都符合类型要求。

属性索引共有 string、number 和 symbol 三种类型。

```
interface A {
  [prop: string]: number;
  [prop: number]: number;
  [prop: symbol]: number;
  [prop: bigint]: number; // An index signature parameter type must be 'string', 'number', 'symbol', or a template literal type. 索引签名参数类型必须是“字符串”、“数字”、“符号”或模板文字类型。
}
```

```
// 属性索引是 string 类型
interface A {
  [prop: string]: number;
```

```
}
let a:A = { 1: 1 } // 正确
let b:A = { '1': 1 } // 正确
let c:A = { [Symbol()]: 1 } // 正确
```

```
// 属性索引是 number 类型
interface A {
  [prop: number]: number;
}
let a:A = { 1: 1 } // 正确
let b:A = { '1': 1 } // 正确
let c:A = { [Symbol()]: 1 } // 正确
```

```
// 属性索引是 symbol 类型
interface A {
  [prop: symbol]: number;
}
let a:A = { 1: 1 };
// Type '{ 1: number; }' is not assignable to type 'A'. Object literal may only
specify known properties, and '1' does not exist in type 'A'. 类型“{1:数字;}'不能赋
值给类型'A'。对象字面量只能指定已知的属性，并且'1'在类型'A'中不存在。

let b:A = { '1': 1 }; // 和上面报错一致
let c:A = { [Symbol()]: 1 };
```

一个接口中，最多只能定义一个字符串索引。字符串索引会约束该类型中所有名字为字符串的属性。

```
interface MyObj {
  [prop: string]: number;
  a: boolean; // Property 'a' of type 'boolean' is not assignable to 'string'
index type 'number'. 类型为布尔的属性'a'不能赋值给索引类型为'number'的'string'。
}
```

上例中，属性索引指定所有名称为字符串的属性，它们的属性值必须是数值（number）。属性 a 的值为布尔值就报错了。将 `a` 的类型改为 `number` 就对了。

属性的数值索引，其实是指定数组的类型。

```
interface A {
  [prop: number]: string;
}
const obj:A = ['a', 'b', 'c'];
```

上例中，`[prop: number]` 表示属性名的类型是数值，所以可以用数组对变量 `obj` 赋值。

同样的，一个接口中最多只能定义一个数值索引。数值索引会约束所有名称为数值的属性。

如果一个 interface 同时定义了字符串索引和数值索引，那么数值索性必须服从于字符串索引。因为在 JavaScript 中，**数值属性名最终是自动转换成字符串属性名。**

```
interface A {
  [prop: string]: number;
  [prop: number]: string; // 'number' index type 'string' is not assignable to
'string' index type 'number'.
}

interface B {
  [prop: string]: number;
  [prop: number]: number; // 正确
}
```

上面示例中，数值索引的属性值类型与字符串索引不一致，就会报错。数值索引必须兼容字符串索引的类型声明。

（3）对象的方法

对象的方法共有三种写法。

```
// 写法一
interface A {
  f(x: boolean): string;
}
let bool1 = true;
let a: A = { f: (bool1) => '12'};
// 等价于
let a: A = { f: function(bool1) { return '12'}};

// 写法二
interface B {
  f: (x: boolean) => string;
}
let bool2 = true;
let b: B = { f: (bool2) => '12'};

// 写法三
interface C {
  f: { (x: boolean): string };
}
let bool3 = true;
let c: C = { f: (bool3) => '12'};
```

属性名可以采用表达式，所以下面的写法也是可以的。

const f = 'f';

interface A { [f](x: boolean): string; }

（4）函数

interface 也可以用来声明独立的函数。

```
interface Add {
  (x:number, y:number): number;
}
const myAdd:Add = (x,y) => x + y;
```

上面示例中，接口Add声明了一个函数类型。

（5）构造函数

interface 内部可以使用 `new` 关键字，表示构造函数。

```
interface ErrorConstructor {
  new (message?: string): Error;
}
```

上面示例中，接口 `ErrorConstructor` 内部有 `new` 命令，表示它是一个构造函数。

1. interface 的继承

interface 可以继承其他类型，主要有下面几种情况。

1.1. `interface` 继承 `interface`，`interface` 可以使用 `extends` 关键字，继承其他 `interface`。

```
interface Shape {
  name: string;
}
interface Circle extends Shape {
  radius: number;
}
const c: Circle = {name: 'c', radius: 1};
```

上例中，`Circle` 继承了 `Shape`，所以 `Circle` 其实有两个属性 `name` 和 `radius`。这时，`Circle` 是子接口，`Shape` 是父接口。

`extends` 关键字会从继承的接口里面拷贝属性类型，这样就不必书写重复的属性。

interface 允许多重继承。

```
interface Style {
  color: string;
}
interface Shape {
  name: string;
}
interface Circle extends Style, Shape {
  radius: number;
}
```

上例中，`Circle` 同时继承了 `Style` 和 `Shape`，所以拥有三个属性 `color`、`name` 和 `radius`。

多重接口继承，实际上相当于多个父接口的合并。

如果子接口与父接口存在同名属性，那么子接口的属性会覆盖父接口的属性。注意，子接口与父接口的同名属性必须是类型兼容的，不能有冲突，否则会报错。

```
interface Foo {
  id: string;
}
interface Bar extends Foo {
  id: number;
  // Interface 'Bar' incorrectly extends interface 'Foo'. Types of property 'id'
are incompatible. Type 'number' is not assignable to type 'string'. 接口'Bar'错误地
扩展了接口'Foo'。属性'id'的类型不兼容。类型'number'不能赋值给类型'string'。
}
```

多重继承时，如果多个父接口存在同名属性，那么这些同名属性不能有类型冲突，否则会报错。

```
interface Foo {
  id: string;
}
interface Bar {
  id: number;
}
// 报错
interface Baz extends Foo, Bar { // Named property 'id' of types 'Foo' and 'Bar'
are not identical. 类型'Foo'和'Bar'的命名属性'id'不相同。
  type: string;
}
```

1.2. interface 继承 type

`interface` 可以继承 `type` 命令定义的对象类型。

```

```

```
type Country = {
  name: string;
  capital: string;
}
interface CountryWithPop extends Country {
  population: number;
}

type NewProps = {
    [Prop in keyof CountryWithPop]: boolean;
};
// keyof 运算符返回对象的键组成的联合类型，in 运算符取出联合类型中每一个成员。[Prop in keyof CountryWithPop] 取出了 'population', 'name', 'capital'
// type NewProps = { population: boolean; name: boolean; capital: boolean; }
```

如果 `type` 命令定义的类型不是对象，interface 就无法继承。

1.3. interface 继承 class

interface 还可以继承 class，即继承该类的所有成员。

```
class A {
  x:string = '';

  y():boolean {
    return true;
  }
}
interface B extends A {
  z: number
}
type props = {
  [prop in keyof B]: boolean;
}
// type props = { z: boolean; x: boolean; y: boolean; }
```

实现 `B` 接口的对象就需要实现这些属性。

```
const b:B = {
  x: '',
  y: function() { return true },
  z: 123
}
```

2. 接口合并

多个同名接口会合并成一个接口。

```
interface Box {
  height: number;
  width: number;
}
interface Box {
  length: number;
}
```

上中，两个 `Box` 接口会合并成一个接口，同时有 `height`、`width` 和 `length` 三个属性。

Web 网页开发经常会对 `windows` 对象和 `document` 对象添加自定义属性，但是 TypeScript 会报错，因为原始定义没有这些属性。解决方法就是把自定义属性写成 `interface`，合并进原始定义。

```
interface Document {
  foo: string;
}
document.foo = 'hello';
```

上例中，接口 `Document` 增加了一个自定义属性 `foo`，从而就可以在 `document` 对象上使用自定义属性。

同名接口合并时，同一个属性如果有多个类型声明，彼此不能有类型冲突。

```
interface A {
  a: number;
}
interface A {
  a: string; // Subsequent property declarations must have the same type.
Property 'a' must be of type 'number', but here has type 'string'. 随后的属性声明必须具有相同的类型。属性'a'必须是'number'类型，但这里有'string'类型。
}
```

同名接口合并时，如果同名方法有不同的类型声明，那么会发生函数重载。而且，后面的定义比前面的定义具有更高的优先级。

```
interface Cloner {
  clone(animal: Animal): Animal;
}
interface Cloner {
  clone(animal: Sheep): Sheep;
}
interface Cloner {
  clone(animal: Dog): Dog;
  clone(animal: Cat): Cat;
}
// 等同于
interface Cloner {
```

```
  clone(animal: Dog): Dog;
  clone(animal: Cat): Cat;
  clone(animal: Sheep): Sheep;
  clone(animal: Animal): Animal;
}
```

上例中，`clone()` 方法有不同的类型声明，会发生函数重载。这时，越靠后的定义，优先级越高，排在函数重载的越前面。比如，`clone(animal: Animal)` 是最先出现的类型声明，就排在函数重载的最后，属于 `clone()` 函数最后匹配的类型。

这个规则有一个例外。同名方法之中，如果有一个参数是字面量类型，字面量类型有更高的优先级。

```
interface A {
  f(x:'foo'): boolean;
}
interface A {
  f(x:any): void;
}
// 等同于
interface A {
  f(x:'foo'): boolean;
  f(x:any): void;
}
```

上例中，`f()` 方法有一个类型声明的参数 `x` 是字面量类型，这个类型声明的优先级最高，会排在函数重载的最前面。

一个实际的例子是 `Document` 对象的 `createElement()` 方法，它会根据参数的不同，而生成不同的 `HTML` 节点对象。

```
interface Document {
  createElement(tagName: any): Element;
}
interface Document {
  createElement(tagName: "div"): HTMLDivElement;
  createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
  createElement(tagName: string): HTMLElement;
  createElement(tagName: "canvas"): HTMLCanvasElement;
}
// 等同于
interface Document {
  createElement(tagName: "canvas"): HTMLCanvasElement;
  createElement(tagName: "div"): HTMLDivElement;
  createElement(tagName: "span"): HTMLSpanElement;
  createElement(tagName: string): HTMLElement;
  createElement(tagName: any): Element;
}
```

上例中，`createElement()` 方法的函数重载，参数为字面量的类型声明会排到最前面，返回具体的 `HTML` 节点对象。**类型越不具体的参数，排在越后面，返回通用的 `HTML` 节点对象。**

如果两个 interface 组成的联合类型存在同名属性，那么该属性的类型也是联合类型。

```
interface Circle {
  area: bigint;
}
interface Rectangle {
  area: number;
}
declare const s: Circle | Rectangle;
s.area;  // bigint | number
```

上例中，接口 `Circle` 和 `Rectangle` 组成一个联合类型 `Circle | Rectangle` 。因此，这个联合类型的同名属性 `area`，也是一个联合类型。本例中的 `declare` 命令表示变量 `s` 的具体定义，由其他脚本文件给出。

3. interface 与 type 的异同

interface 命令与 type 命令作用类似，都可以表示对象类型。

很多对象类型既可以用 `interface` 表示，也可以用 `type` 表示。而且，两者往往可以换用，**几乎所有的 `interface` 命令都可以改写为 `type` 命令。**

它们的相似之处，首先表现在都能为对象类型起名。

```
type Country = {
  name: string;
  capital: string;
}
interface City {
  name: string;
  capital: string;
}
```

`class` 命令也有类似作用，通过定义一个类，同时定义一个对象类型。但是，它会创造一个值，编译后依然存在。如果只是单纯想要一个类型，应该使用 `type` 或 `interface`。

interface 与 type 的区别有下面几点。

- （1）`type` 能够表示非对象类型，而 `interface` 只能表示对象类型（包括数组、函数等）。

- （2）`interface` 可以继承其他类型，`type` 不支持继承。

继承的主要作用是添加属性，`type` 定义的对象类型如果想要添加属性，只能使用 `&` 运算符，重新定义一个类型。

```
type Animal = {
  name: string
}
type Bear = Animal & {
  honey: boolean
}
```

上例中，类型 `Bear` 在 `Animal` 的基础上添加了一个属性 `honey`。`&` 运算符，表示同时具备两个类型的特征，因此可以起到两个对象类型合并的作用。

作为比较，`interface` 添加属性，采用的是继承的写法。

```
interface Animal {
  name: string
}
interface Bear extends Animal {
  honey: boolean
}
```

继承时，`type` 和 `interface` 是可以换用的。`interface` 可以继承 `type`。`type` 也可以继承 `interface`。

```
type Foo = { x: number; };
interface Bar extends Foo {
  y: number;
}
```

```
interface Foo {
  x: number;
}
type Bar = Foo & { y: number; };
```

- （3）同名 `interface` 会自动合并，同名 `type` 则会报错。也就是说，**TypeScript 不允许使用 `type` 多次定义同一个类型。**

```
interface A { foo:number };
interface A { bar:number };
const obj:A = {
  foo: 1,
  bar: 1
};

type B = { foo:number }; // Duplicate identifier 'B'.
type B = { bar:number }; // Duplicate identifier 'B'.
```

这表明，interface 是开放的，可以添加属性，type 是封闭的，不能添加属性，只能定义新的 type。

- （4）interface不能包含属性映射（mapping），type 可以。

```
interface Point {
  x: number;
  y: number;
}

// 正确
type PointCopy1 = {
  [Key in keyof Point]: Point[Key];
};

// 报错
interface PointCopy2 {
  [Key in keyof Point]: Point[Key];
};
```

- （5）this 关键字只能用于 interface。

```
// 正确
interface Foo {
  add(num:number): this;
};
// 报错
type Foo = {
  add(num:number): this;
};
```

上面示例中，type 命令声明的方法 add()，返回 this 就报错了。interface 命令没有这个问题。

```
class Calculator implements Foo {
  result = 0;
  add(num:number) {
    this.result += num;
    return this;
  }
}
```

- （6）type 可以扩展原始数据类型，interface 不行。

```
// 正确
type MyStr = string & {
  type: 'new'
};
```

```
// 报错
interface MyStr extends string {
  type: 'new'
}
```

上例中，type 可以扩展原始数据类型 string，interface 就不行。

- （7）`interface` 无法表达某些复杂类型（比如交叉类型和联合类型），但是 `type` 可以。

```
type A = { /* ... */ };
type B = { /* ... */ };

type AorB = A | B;
type AorBWithName = AorB & {
  name: string
};
```

上例中，类型 `AorB` 是一个联合类型，`AorBWithName` 则是为 `AorB` 添加一个属性。这两种运算，`interface` 都没法表达。

综上所述，**如果有复杂的类型运算，那么没有其他选择只能使用 `type`；一般情况下，`interface` 灵活性比较高，便于扩充类型或自动合并，建议优先使用。**

class 类型

1. 简介

1.1. 属性的类型

类的属性可以在顶层声明，也可以在构造方法内部声明。 对于顶层声明的属性，可以在声明时同时给出类型。

```
class Point {
  x:number;
  y:number;
}
```

上面声明中，属性 x 和 y 的类型都是 number。如果不给出类型，TypeScript 会认为 x 和 y 的类型都是 any。

```
class Point {
  x;
  y;
}
```

上面示例中，x 和 y 的类型都是 any。

如果声明时给出初值，可以不写类型，TypeScript 会自行推断属性的类型。

```
class Point {
  x = 0;
  y = 0;
}
```

上例中，属性 x 和 y 的类型都会被推断为 number。

1.2. readonly 修饰符

属性名前面加上 readonly 修饰符，就表示该属性是只读的。实例对象不能修改这个属性。

```
class A {
  readonly id = 'foo';
}
const a = new A();
a.id = 'bar'; // Cannot assign to 'id' because it is a read-only property. 不能赋值给'id'，因为它是一个只读属性。
```

readonly 属性的初始值，可以写在顶层属性，也可以写在构造方法里面。

```
class A {
  readonly id:string;
  constructor() {
    this.id = 'bar'; // 正确
  }
  fn1() {
    this.id = 'a'; // Cannot assign to 'id' because it is a read-only property.
  }
}
```

上例中，构造方法内部设置只读属性的初值，这是可以的。但在其他地方修改就会报错。

```
class A {
  readonly id:string = 'foo';
  constructor() {
    this.id = 'bar'; // 正确
  }
  fn1() {
    this.id = 'a'; // Cannot assign to 'id' because it is a read-only property.
  }
}
```

上例中，构造方法修改只读属性的值也是可以的。或者说，如果两个地方都设置了只读属性的值，以构造方法为准。**在其他方法修改只读属性都会报错。**

1.3. 方法的类型

类的方法就是普通函数，类型声明方式与函数一致。

```
class Point {
  x:number;
  y:number;

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

  add(point:Point) {
    return new Point(
      this.x + point.x,
      this.y + point.y
    );
  }
}
```

上例中，构造方法 `constructor()` 和普通方法 `add()` 都注明了参数类型，但是省略了返回值类型，因为 TypeScript 可以自己推断出来。

类的方法跟普通函数一样，可以使用参数默认值，以及函数重载。

参数默认值：

```
class Point {
  x: number;
  y: number;

  constructor(x = 0, y = 0) {
    this.x = x;
    this.y = y;
  }
}
```

上例中，如果新建实例时，不提供属性 `x` 和 `y` 的值，它们都等于默认值 `0`。

函数重载：

```
class Point {
  constructor(x:number, y:string);
  constructor(s:string);
  constructor(xs:number|string, y?:string) {
    // ...
  }
}
```

上例中，构造方法可以接受一个参数，也可以接受两个参数，采用函数重载进行类型声明。

构造方法不能声明返回值类型，否则报错，因为它总是返回实例对象。

```
class B {
  constructor():object { // Type annotation cannot appear on a constructor
declaration. 类型注释不能出现在构造函数声明中。
    // ...
  }
}
```

1.4. 存取器方法

存取器（accessor）是特殊的类方法，包括取值器（`getter`）和存值器（`setter`）两种方法。它们用于读写某个属性，取值器用来读取属性，存值器用来写入属性。

```
class C {
  _name = '';
  get name() {
    return this._name;
  }
  set name(value) {
    this._name = value;
  }
}
```

上例中，`get name()` 是取值器，其中 `get` 是关键词，`name` 是属性名。外部读取 `name` 属性时，实例对象会自动调用这个方法，该方法的返回值就是 `name` 属性的值。`set name()` 是存值器，其中 `set` 是关键词，`name` 是属性名。外部写入 `name` 属性时，实例对象会自动调用这个方法，并将所赋的值作为函数参数传入。

TypeScript 对存取器有以下规则。

（1）**如果某个属性只有 `get` 方法，没有set方法，那么该属性自动成为只读属性。**

```
class C {
  _name = 'foo';

  get name() {
    return this._name;
  }
}

const c = new C();
c.name = 'bar'; // Cannot assign to 'name' because it is a read-only property.
```

（2）TypeScript 5.1 版之前，set方法的参数类型，必须兼容get方法的返回值类型，否则报错。

```
// TypeScript 5.1 版之前
class C {
  _name = '';
  get name():string {  // 报错
    return this._name;
  }
  set name(value:number) {
    this._name = String(value);
  }
}
```

上面示例中，`get` 方法的返回值类型是字符串，与 `set` 方法的参数类型 `number` 不兼容，导致报错。改成下面这样，就不会报错。

```

```

```
class C {
  _name = '';
  get name():string {
    return this._name;
  }
  set name(value:number|string) {
    this._name = String(value);
  }
}
```

上例中，set 方法的参数类型（number|string）兼容 get 方法的返回值类型（string），这是允许的。

TypeScript 5.1 版做出了改变，现在两者可以不兼容。

（3）**get方法与set方法的可访问性必须一致，要么都为公开方法，要么都为私有方法。**

1.5. 属性索引

类允许定义属性索引。

```
class MyClass {
  [s:string]: boolean |
    ((s:string) => boolean);

  get(s:string) {
    return this[s] as boolean;
  }
}
```

上例中，[s:string] 表示所有属性名类型为字符串的属性，它们的属性值要么是布尔值，要么是返回布尔值的函数。

由于类的方法是一种特殊属性（属性值为函数的属性），所以属性索引的类型定义也涵盖了方法。如果一个对象同时定义了属性索引和方法，那么前者必须包含后者的类型。

```
class MyClass {
  [s:string]: boolean;
  f() { // Property 'f' of type '() => boolean' is not assignable to 'string'
index type 'boolean'. 属性'f'的类型'()=&gt;' Boolean '不能赋值给'string'索引类型'
Boolean '。
    return true;
  }
}
```

上例中，属性索引的类型里面不包括方法，导致后面的方法 f() 定义直接报错。正确的写法是：

```
class MyClass {
  [s:string]: boolean | (() => boolean);
  f() {
    return true;
  }
}
```

属性存取器视同属性。

```
class MyClass {
  [s:string]: boolean;

  get isInstance() {
    return true;
  }
}
```

上例中，属性 `inInstance` 的读取器虽然是一个函数方法，但是视同属性，所以属性索引虽然没有涉及方法类型，但是不会报错。

2. 类的 interface 接口

2.1. implements 关键字

```
interface Point {
  x: number;
  y: number;
}
class MyPoint implements Point {
  x = 1;
  y = 1;
  z:number = 1;
}
```

上例中，`MyPoint` 类实现了 `Point` 接口，但是内部还定义了一个额外的属性 `z`。

`implements` 关键字后面，不仅可以是接口，也可以是另一个类。这时，后面的类将被当作接口。

```
class Car {
  id:number = 1;
  move():void {};
}
class MyCar implements Car {
  id = 2; // 不可省略
  move():void {};   // 不可省略
}
```

上例中，`implements` 后面是类 `Car`，这时 TypeScript 就把 `Car` 视为一个接口，要求 `MyCar` 实现 `Car` 里面的每一个属性和方法，否则就会报错。所以，这时不能因为 `Car` 类已经实现过一次，而在`MyCar` 类省略属性或方法。

`interface` 描述的是类的对外接口，也就是实例的公开属性和公开方法，不能定义私有的属性和方法。这是因为 TypeScript 设计者认为，私有属性是类的内部实现，接口作为模板，不应该涉及类的内部代码写法。

```
interface Foo {
  private member:{}; // 'private' modifier cannot appear on a type member.
“private”修饰符不能出现在类型成员上。
}
```

2.2. 实现多个接口

类可以实现多个接口（其实是接受多重限制），每个接口之间使用逗号分隔。

```
class Car implements MotorVehicle, Flyable, Swimmable {
  // ...
}
```

上面示例中，`Car` 类同时实现了 `MotorVehicle`、`Flyable`、`Swimmable` 三个接口。 **这意味着，它必须部署这三个接口声明的所有属性和方法，满足它们的所有条件。**

但是，同时实现多个接口并不是一个好的写法，容易使得代码难以管理，可以使用两种方法替代。

第一种方法是类的继承。

```
class Car implements MotorVehicle {
}
class SecretCar extends Car implements Flyable, Swimmable {
}
```

上例中，`Car` 类实现了 `MotorVehicle`，而 `SecretCar` 类继承了 `Car` 类，然后再实现 `Flyable` 和 `Swimmable` 两个接口，相当于 `SecretCar` 类同时实现了三个接口。

第二种方法是接口的继承。

```
interface A {
  a:number;
}
interface B extends A {
  b:number;
}
```

上面示例中，接口 B 继承了接口 A，类只要实现接口 B，就相当于实现 A 和 B 两个接口。

前一个例子可以用接口继承改写。

```
interface MotorVehicle {
  // ...
}
interface Flyable {
  // ...
}
interface Swimmable {
  // ...
}

interface SuperCar extends MotoVehicle,Flyable, Swimmable {
  // ...
}

class SecretCar implements SuperCar {
  // ...
}
```

上面示例中，类 SecretCar 通过 SuperCar 接口，就间接实现了多个接口。

发生多重实现时（即一个接口同时实现多个接口），不同接口不能有互相冲突的属性。

```
interface Flyable {
  foo:number;
}
interface Swimmable {
  foo:string;
}
```

上例中，属性 foo 在两个接口里面的类型不同，如果同时实现这两个接口，就会报错。

2.3. 类和接口的合并

如果一个类和一个接口同名，那么接口会被合并进类。

```
class A {
  x:number = 1;
}
interface A {
  y:number;
}

let a = new A();
a.y = 10;
```

```
a.x // 1
a.y // 10
```

上例中，类 `A` 与接口 `A` 同名，后者会被合并进前者的类型定义。

合并进类的非空属性（上例的 `y`），如果在赋值之前读取，会返回 `undefined`。

```
class A {
  x:number = 1;
}
interface A {
  y:number;
}
let a = new A();
a.y // undefined
```

3. Class 类型

3.1. 实例类型

TypeScript 的类本身就是一种类型，但是它代表该类的实例类型，而不是 class 的自身类型。

```
class Color {
  name:string;

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

const green:Color = new Color('green');
```

上例中，定义了一个类 `Color`。它的类名就代表一种类型，实例对象 `green` 就属于该类型。

对于引用实例对象的变量来说，既可以声明类型为 `Class`，也可以声明类型为 `Interface`，因为两者都代表实例对象的类型。

```
interface MotorVehicle {}
class Car implements MotorVehicle {}

// 写法一
const c1:Car = new Car();
// 写法二
const c2:MotorVehicle = new Car();
```

上例中，变量的类型可以写成类 `Car`，也可以写成接口 `MotorVehicle`。它们的区别是，如果类 `Car` 有接口 `MotoVehicle` 没有的属性和方法，那么只有变量 `c1` 可以调用这些属性和方法。

作为类型使用时，类名只能表示实例的类型，不能表示类的自身类型。

```
class Point {
  x:number;
  y:number;

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

function createPoint(
  PointClass:Point,
  x: number,
  y: number
) {
  return new PointClass(x, y); // This expression is not constructable. Type
'Point' has no construct signatures. 这个表达式不能构造。类型“Point”没有构造签名。
}
```

上例中，函数 `createPoint()` 的第一个参数 `PointClass`，需要传入 `Point` 这个类，但是如果把参数的类型写成 `Point` 就会报错，因为 `Point` 描述的是实例类型，而不是 `Class` 的自身类型。

由于类名作为类型使用，实际上代表一个对象，因此可以把类看作为对象类型起名。事实上，TypeScript 有三种方法可以为对象类型起名：type、interface 和 class。

3.2. 类的自身类型

要获得一个类的自身类型，一个简便的方法就是使用 typeof 运算符。

```
function createPoint(
  PointClass:typeof Point,
  x:number,
  y:number
):Point {
  return new PointClass(x, y);
}
```

上例中，`createPoint()` 的第一个参数 `PointClass` 是 `Point` 类自身，要声明这个参数的类型，简便的方法就是使用 `typeof Point`。因为 `Point` 类是一个值，`typeof Point` 返回这个值的类型。注意，`createPoint()` 的返回值类型是`Point`，代表实例类型。

3.3. 结构类型原则

`Class` 也遵循“结构类型原则”。一个对象只要满足 `Class` 的实例结构，就跟该 `Class` 属于同一个类型。

```
class Foo {
  id!:number;
}
function fn(arg:Foo) {
  // ...
}
const bar = {
  id: 10,
  amount: 100,
};
fn(bar); // 正确
```

上例中，对象 `bar` 满足类 `Foo` 的实例结构，只是多了一个属性 `amount`。所以，它可以当作参数，传入函数 `fn()`。

如果两个类的实例结构相同，那么这两个类就是兼容的，可以用在对方的使用场合。 两个结构相同的类，被视为相同的类。

```
class Person {
  name: string = '';
}
class Customer {
  name: string = '';
}

const c:Customer = new Person();
```

上例中，`Person` 和 `Customer` 是两个结构相同的类，TypeScript 将它们视为相同类型，因此 `Person` 可以用在类型为 `Customer` 的场合。

为 `Person` 类添加一个属性：

```
class Person {
  name: string = '';
  age: number = '';
}

class Customer {
  name: string = '';
}

const c:Customer = new Person();
```

上例中，`Person` 类添加了一个属性 `age`，跟 `Customer` 类的结构不再相同。但是这种情况下，TypeScript 依然认为，`Person` 属于 `Customer` 类型。

这是因为根据"结构类型原则"，只要 Person 类具有 name 属性，就满足 Customer 类型的实例结构，所以可以代替它。反过来就不行，如果 Customer 类多出一个属性，就会报错。

```
class Person {
  name: string = '';
}
class Customer {
  name: string = '';
  age: number = 0;
}
const c:Customer = new Person(); // Property 'age' is missing in type 'Person' but
required in type 'Customer'.
```

上例中，Person 类比 Customer 类少一个属性 age，它就不满足 Customer 类型的实例结构，就报错了。因为在使用 Customer 类型的情况下，可能会用到它的 age 属性，而 Person 类就没有这个属性。只要 A 类具有 B 类的结构，哪怕还有额外的属性和方法，TypeScript 也认为 A 兼容 B 的类型。

不仅是类，**如果某个对象跟某个 class 的实例结构相同，TypeScript 也认为两者的类型相同。**

```
class Person {
  name: string = '';
}
const obj = { name: 'John' };
const p:Person = obj;
```

上例中，对象 obj 并不是 Person 的实例，但是赋值给变量 p 不会报错，TypeScript 认为 obj 也属于 Person 类型，因为它们的属性相同。

由于这种情况，运算符 instanceof 不适用于判断某个对象是否跟某个 class 属于同一类型。obj instanceof Person 结果是 false。

空类不包含任何成员，任何其他类都可以看作与空类结构相同。因此，凡是类型为空类的地方，所有类（包括对象）都可以使用。

```
class Empty {}

function fn(x:Empty) {
  // ...
}

fn({});
fn(window);
fn(fn);
```

上例中，函数 fn() 的参数是一个空类，这意味着任何对象都可以用作 fn() 的参数。

确定两个类的兼容关系时，只检查实例成员，不考虑静态成员和构造方法。

```
class Point {
  x: number;
  y: number;
  static t: number;
  constructor(x:number) {}
}
class Position {
  x: number;
  y: number;
  z: number;
  constructor(x:string) {}
}
const point:Point = new Position('');
```

上例中，`Point` 与 `Position` 的静态属性和构造方法都不一样，但因为 `Point` 的实例成员与 `Position` 相同，所以 `Position` 兼容 `Point`。

如果类中存在私有成员（`private`）或保护成员（`protected`），那么确定兼容关系时，TypeScript 要求私有成员和保护成员来自同一个类，这意味着两个类需要存在继承关系。

```
// 情况一
class A {
  private name = 'a';
}
class B extends A {
}
const a:A = new B();

// 情况二
class A {
  protected name = 'a';
}
class B extends A {
  protected name = 'b';
}
const a:A = new B();
```

上例中，`A` 和 `B` 都有私有成员（或保护成员）`name`，这时只有在 `B` 继承 `A` 的情况下（class B extends A），`B` 才兼容 `A`。

4. 类的继承

类（这里又称“子类”）可以使用 extends 关键字继承另一个类（这里又称“基类”）的所有属性和方法。

```
class A {
  greet() {
    console.log('Hello, world!');
  }
```

```
}
class B extends A {
}

const b = new B();
b.greet() // "Hello, world!"
```

上例中，子类 `B` 继承了基类 `A`，因此就拥有了 `greet()` 方法，不需要再次在类的内部定义这个方法了。

根据结构类型原则，子类也可以用于类型为基类的场合。

```
const a:A = b;
a.greet();
```

上例中，变量 `a` 的类型是基类，但是可以赋值为子类的实例。

子类可以覆盖基类的同名方法。

```
class B extends A {
  greet(name?: string) {
    if (name === undefined) {
      super.greet();
    } else {
      console.log(`Hello, ${name}`);
    }
  }
}
```

上例中，子类 `B` 定义了一个方法 `greet()`，覆盖了基类 `A` 的同名方法。其中，参数 `name` 省略时，就调用基类 `A` 的 `greet()` 方法，这里可以写成 `super.greet()`，使用 `super` 关键字指代基类是常见做法。

子类的同名方法不能与基类的类型定义相冲突。

```
class A {
  greet() {
    console.log('Hello, world!');
  }
}

class B extends A {
  greet(name:string) { // Property 'greet' in type 'B' is not assignable to the
same property in base type 'A'.
    console.log(`Hello, ${name}`);
  }
}
```

上例中，子类 B 的 greet() 有一个 name 参数，跟基类 A 的 greet() 定义不兼容，因此就报错了。

如果基类包括保护成员（protected 修饰符），子类可以将该成员的可访问性设置为公开（public 修饰符），也可以保持保护成员不变，但是不能改用私有成员（private修饰符）。

```
class A {
  protected x: string = '';
  protected y: string = '';
  protected z: string = '';
}

class B extends A {
  // 正确
  public x:string = '';

  // 正确
  protected y:string = '';

  // 报错
  private z: string = '';
}
```

上面示例中，子类 B 将基类 A 的受保护成员改成私有成员，就会报错。

extends **关键字后面不一定是类名，可以是一个表达式，只要它的类型是构造函数就可以了。**

```
// 例一
class MyArray extends Array<number> {}

// 例二
class MyError extends Error {}

// 例三
class A {
  greeting() {
    return 'Hello from A';
  }
}
class B {
  greeting() {
    return 'Hello from B';
  }
}

interface Greeter {
  greeting(): string;
}

interface GreeterConstructor {
  new (): Greeter;
```

```
}

function getGreeterBase():GreeterConstructor {
  return Math.random() >= 0.5 ? A : B;
}

class Test extends getGreeterBase() {
  sayHello() {
    console.log(this.greeting());
  }
}
```

上面示例中，例一和例二的 `extends` 关键字后面都是构造函数，例三的 `extends` 关键字后面是一个表达式，执行后得到的也是一个构造函数。

5. 可访问性修饰符

类的内部成员的外部可访问性，由三个可访问性修饰符（access modifiers）控制：`public`、`private` 和 `protected`。这三个修饰符的位置，都写在属性或方法的最前面。

5.1. public

`public` 修饰符表示这是公开成员，外部可以自由访问。`public` 修饰符是默认修饰符，如果省略不写，实际上就带有该修饰符。因此，类的属性和方法默认都是外部可访问的。正常情况下，除非为了醒目和代码可读性，public都是省略不写的。

```
class Greeter {
  public greet() {
    console.log("hi!");
  }
}

const g = new Greeter();
g.greet();
```

上例中，`greet()` 方法前面的 `public` 修饰符，表示该方法可以在类的外部调用，即外部实例可以调用。

5.2. private

`private` **修饰符表示私有成员，只能用在当前类的内部，类的实例和子类都不能使用该成员。**

```
class A {
  private x:number = 0;
}

const a = new A();
a.x // Property 'x' is private and only accessible within class 'A'.

class B extends A {
```

```
  showX() {
    console.log(this.x); // Property 'x' is private and only accessible within
class 'A'.
  }
}
```

上例中，属性 `x` 前面有 `private` 修饰符，表示这是私有成员。因此，实例对象和子类使用该成员，都会报错。

子类不能定义父类私有成员的同名成员。

```
class A {
  private x = 0;
}

class B extends A {
  x = 1; // Class 'B' incorrectly extends base class 'A'. Property 'x' is private
in type 'A' but not in type 'B'. 'x' is declared but its value is never read.
类'B'错误地扩展了基类'A'。属性'x'在类型'A'中是私有的，但在类型'B'中不是。声明了'x'，但永
远不会读取它的值。
}
```

上例中，`A` 类有一个私有属性 `x`，子类 `B` 就不能定义自己的属性 `x` 了。

如果在类的内部，当前类的实例可以获取私有成员。

```
class A {
  private x = 10;

  f(obj:A) {
    console.log(obj.x);
  }
}

const a = new A();
a.f(a) // 10
```

上例中，在类 `A` 内部，`A` 的实例对象可以获取私有成员x。

严格地说，private定义的私有成员，并不是真正意义的私有成员。一方面，编译成 JavaScript 后，`private` 关键字就被剥离了，这时外部访问该成员就不会报错。另一方面，由于前一个原因，TypeScript 对于访问 `private` 成员没有严格禁止，使用方括号写法（`[]`）或者 `in` 运算符，实例对象就能访问该成员。

```
class A {
  private x = 1;
}

const a = new A();
```

```
a.x; // Property 'x' is private and only accessible within class 'A'.
a['x']; // 正确

if ('x' in a) { // 正确
  // ...
}
```

上例中，A 类的属性 x 是私有属性，但是实例使用方括号，就可以读取这个属性，或者使用 in 运算符检查这个属性是否存在，都可以正确执行。

```
// es2022 的写法
class A {
  #x = 1;
}

const a = new A();
a['x'] // Element implicitly has an 'any' type because expression of type '"x"'
can't be used to index type 'A'. Property 'x' does not exist on type 'A'. 元素隐式
地具有'any'类型，因为类型' x '的表达式不能用于索引类型'A'。 类型'A'上不存在属性'x'。
```

构造方法也可以是私有的，这就直接防止了使用 new 命令生成实例对象，只能在类的内部创建实例对象。这时一般会有一个静态方法，充当工厂函数，强制所有实例都通过该方法生成。

```
class Singleton {
  private static instance?: Singleton;

  private constructor() {}

  static getInstance() {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
}
const s = Singleton.getInstance();
```

上例使用私有构造方法，实现了单例模式。想要获得 Singleton 的实例，不能使用 new 命令，只能使用 getInstance() 方法。

5.3. protected

protected **修饰符表示该成员是保护成员，只能在类的内部使用该成员，实例无法使用该成员，但是子类内部可以使用。**

```
class A {
  protected x = 1;
```

```
}

class B extends A {
  getX() {
    return this.x;
  }
}

const a = new A();
const b = new B();

a.x; // Property 'x' is protected and only accessible within class 'A' and its
subclasses.
b.getX(); // 1
```

子类不仅可以拿到父类的保护成员，还可以定义同名成员。

```
class A {
  protected x = 1;
}

class B extends A {
  x = 2;
}
```

上面示例中，子类 `B` 定义了父类 `A` 的同名成员 `x`，并且父类的 `x` 是保护成员，子类将其改成了公开成员。`B` 类的 `x` 属性前面没有修饰符，等同于修饰符是 `public`，外界可以读取这个属性。

在类的外部，实例对象不能读取保护成员，但是在类的内部可以。

```
class A {
  protected x = 1;

  f(obj:A) {
    console.log(obj.x);
  }
}
const a = new A();
a.x; // Property 'x' is protected and only accessible within class 'A' and its
subclasses.
a.f(a); // 1
```

上例中，属性 `x` 是类 `A` 的保护成员，在类的外部，实例对象 `a` 拿不到这个属性。但是，实例对象 `a` 传入类 `A` 的内部，就可以从 `a` 拿到 `x`。

5.4. 实例属性的简写形式

实际开发中，很多实例属性的值，是通过构造方法传入的。

```
class Point {
  x:number;
  y:number;

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

上例中，属性 x 和 y 的值是通过构造方法的参数传入的。

这样的写法等于对同一个属性要声明两次类型，一次在类的头部，另一次在构造方法的参数里面。这有些累赘，TypeScript 就提供了一种简写形式。

```
class Point {
  constructor(
    public x:number,
    public y:number
  ) {}
}
const p = new Point(10, 10);
p.x // 10
p.y // 10
```

上例中，构造方法的参数 x 前面有 public 修饰符，这时 TypeScript 就会自动声明一个公开属性 x，不必在构造方法里面写任何代码，同时还会设置 x 的值为构造方法的参数值。这里的 public 不能省略。

除了 public 修饰符，构造方法的参数名只要有 private、protected、readonly 修饰符，都会自动声明对应修饰符的实例属性。

```
class A {
  constructor(
    public a: number,
    protected b: number,
    private c: number,
    readonly d: number
  ) {}
}
```

```
"use strict";
class A {
  constructor(a, b, c, d) {
    this.a = a;
    this.b = b;
    this.c = c;
```

```
    this.d = d;
  }
}
```

`readonly` 还可以与其他三个可访问性修饰符，一起使用。

```
class A {
  constructor(
    public readonly x:number,
    protected readonly y:number,
    private readonly z:number
  ) {}
}
```

6. 静态成员

类的内部可以使用 `static` 关键字，定义静态成员。静态成员是只能通过类本身使用的成员，不能通过实例对象使用。

```
class MyClass {
  static x = 0;
  static printX() {
    console.log(MyClass.x);
  }
}
MyClass.x // 0
MyClass.printX() // 0

mc.x; // Property 'x' does not exist on type 'MyClass'. Did you mean to access the
static member 'MyClass.x' instead?
mc.printX(); // Property 'printX' does not exist on type 'MyClass'. Did you mean
to access the static member 'MyClass.printX' instead?
```

上例中，`x` 是静态属性，`printX()` 是静态方法。它们都必须通过 `MyClass` 获取，而不能通过实例对象调用。

`static` 关键字前面可以使用 `public`、`private`、`protected` 修饰符。

```
class MyClass {
  private static x = 0;
}
MyClass.x // Property 'x' is private and only accessible within class 'MyClass'.
```

上例中，静态属性 `x` 前面有 `private` 修饰符，表示只能在 `MyClass` 内部使用，如果在外部调用这个属性就会报错。

静态私有属性也可以用 ES6 语法的 `#` 前缀表示：

```
class MyClass {
  static #x = 0;
}
```

`public` 和 `protected` 的静态成员可以被继承。

```
class A {
  public static x = 1;
  protected static y = 1;
}
class B extends A {
  static getY() {
    return B.y;
  }
}

B.x // 1
B.getY() // 1
```

上例中，类 `A` 的静态属性 `x` 和 `y` 都被 `B` 继承，公开成员 `x` 可以在 `B` 的外部获取，保护成员 `y` 只能在 `B` 的内部获取。

7. 泛型类

类也可以写成泛型，使用类型参数。

```
class Box<Type> {
  contents: Type;

  constructor(value:Type) {
    this.contents = value;
  }
}

const b:Box<string> = new Box('hello!');
```

上例中，类 `Box` 有类型参数 `Type`，因此属于泛型类。新建实例时，变量的类型声明需要带有类型参数的值，不过本例等号左边的 `Box<string>` 可以省略不写，因为可以从等号右边推断得到。

静态成员不能使用泛型的类型参数。

```
class Box<Type> {
  static defaultContents: Type; // 报错
}
```

上例中，静态属性 `defaultContents` 的类型写成类型参数 `Type` 会报错。因为这意味着调用时必须给出类型参数（即写成 `Box<string>.defaultContents`），并且类型参数发生变化，这个属性也会跟着变，这并不是好的做法。

8.抽象类，抽象成员

TypeScript 允许在类的定义前面，加上关键字 `abstract`，表示该类不能被实例化，只能当作其他类的模板。这种类就叫做"抽象类"（abstract class）。

```
abstract class A {
  id = 1;
}
const a = new A(); // Cannot create an instance of an abstract class.
```

抽象类只能当作基类使用，用来在它的基础上定义子类。

```
abstract class A {
  id = 1;
}
class B extends A {
  amount = 100;
}
const b = new B();
b.id // 1
b.amount // 100
```

上例中，`A` 是一个抽象类，`B` 是 `A` 的子类，继承了 `A` 的所有成员，并且可以定义自己的成员和实例化。

抽象类的子类也可以是抽象类，也就是说，抽象类可以继承其他抽象类。

```
abstract class A {
  foo:number;
}
abstract class B extends A {
  bar:string;
}
```

抽象类的内部可以有已经实现好的属性和方法，也可以有还未实现的属性和方法。后者就叫做"抽象成员"（`abstract member`），即属性名和方法名有 `abstract` 关键字，表示该方法需要子类实现。如果子类没有实现抽象成员，就会报错。

```
abstract class A {
  abstract foo:string;
  bar:string = '';
}
```

```
class B extends A { // Non-abstract class 'B' does not implement all abstract
members of 'A' 非抽象类'B'没有实现'A'的所有抽象成员
  foo2 = 'b';
}
```

上例中，抽象类 `A` 定义了抽象属性 `foo`，子类 `B` 必须实现这个属性，否则会报错。

如果抽象类的方法前面加上 `abstract`，就表明子类必须给出该方法的实现。

```
abstract class A {
  abstract execute():string;
}
class B extends A {
  execute() {
    return `B executed`;
  }
}
```

这里有几个注意点。

- （1）抽象成员只能存在于抽象类，不能存在于普通类。
- （2）抽象成员不能有具体实现的代码。也就是说，已经实现好的成员前面不能加 `abstract` 关键字。
- （3）抽象成员前也不能有 `private` 修饰符，否则无法在子类中实现该成员。
- （4）**一个子类最多只能继承一个抽象类。**

总之，抽象类的作用是，确保各种相关的子类都拥有跟基类相同的接口，可以看作是模板。其中的抽象成员都是必须由子类实现的成员，非抽象成员则表示基类已经实现的、由所有子类共享的成员。

9. this 问题

类的方法经常用到 `this` 关键字，它表示该方法当前所在的对象。

```
class A {
  name = 'A';

  getName() {
    return this.name;
  }
}

const a = new A();
a.getName() // 'A'

const b = {
  name: 'b',
  getName: a.getName
```

```
};
b.getName() // 'b'
```

上例中，变量 a 和 b 的 getName() 是同一个方法，但是执行结果不一样，原因就是它们内部的 this 指向不一样的对象。如果 getName() 在变量 a 上运行，this 指向 a；如果在 b 上运行，this 指向 b。

有些场合需要给出 this 类型，但是 JavaScript 函数通常不带有 this 参数，这时 TypeScript 允许函数增加一个名为 this 的参数，放在参数列表的第一位，用来描述函数内部的 this 关键字的类型。

```
// 编译前
function fn(
  this: SomeType,
  x: number
) {
  /* ... */
}
```

```
// 编译后
function fn(x) {
  /* ... */
}
```

上例中，函数 fn() 的第一个参数是 this，用来声明函数内部的 this 的类型。编译时，TypeScript 一旦发现函数的第一个参数名为 this，则会去除这个参数，即编译结果不会带有该参数。

```
class A {
  name = 'A';

  getName(this: A) {
    return this.name;
  }
}

const a = new A();
const b = a.getName;

b(); // The 'this' context of type 'void' is not assignable to method's 'this' of
type 'A'. 类型'void'的'this'上下文不能分配给类型'A'的方法'this'。
```

上例中，类 A 的 getName() 添加了 this 参数，如果直接调用这个方法，this 的类型就会跟声明的类型不一致，从而报错。

this 参数的类型可以声明为各种对象。

```
function foo(
  this: { name: string }
) {
  this.name = 'Jack';
  this.name = 0; // Type 'number' is not assignable to type 'string'.
}

foo.call({ name: 123 }); // Type 'number' is not assignable to type 'string'.
```

上例中，参数 this 的类型是一个带有 name 属性的对象，不符合这个条件的 this 都会报错。

在类的内部，this 本身也可以当作类型使用，表示当前类的实例对象。

```
class Box {
  contents:string = '';

  set(value:string):this {
    this.contents = value;
    return this;
  }
}
```

上例中，set() 方法的 **返回值类型就是this，表示当前的实例对象。**

注意，this类型不允许应用于静态成员。

```
class A {
  static a:this; // A 'this' type is available only in a non-static member of a
class or interface. “this”类型仅在类或接口的非静态成员中可用。
}
```

上例中，静态属性 a 的返回值类型是 this，就报错了。原因是this类型表示实例对象，但是静态成员拿不到实例对象。

有些方法返回一个布尔值，表示当前的 this 是否属于某种类型。这时，这些方法的返回值类型可以写成 this is Type 的形式，其中用到了 is 运算符。

```
class FileSystemObject {
  isFile(): this is FileRep {
    return this instanceof FileRep;
  }

  isDirectory(): this is Directory {
    return this instanceof Directory;
  }
```

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

上例中，两个方法的返回值类型都是布尔值，写成 `this is Type` 的形式，可以精确表示返回值。

泛型

有些时候，函数返回值的类型与参数类型是相关的。

```
function getFirst(arr) {
  return arr[0];
}
```

上例中，函数 `getFirst()` 总是返回参数数组的第一个成员。参数数组成员是什么类型，返回值就是什么类型。

为了解决这个问题，TypeScript 就引入了“泛型”（generics）。泛型的特点就是带有“类型参数”（type parameter）。

```
function getFirst<T>(arr:T[]):T {
  return arr[0];
}
```

上例中，函数 `getFirst()` 的函数名后面尖括号的部分，就是类型参数，参数要放在一对尖括号（<>）里面。本例只有一个类型参数T，可以将其理解为类型声明需要的变量，需要在调用时传入具体的参数类型。

上例的函数 `getFirst()` 的参数类型是 `T[]`，返回值类型是 `T`，就清楚地表示了两者之间的关系。比如，输入的参数类型是 `number[]`，那么 `T` 的值就是 `number`，因此返回值类型也是 `number`。

函数调用时，需要提供类型参数。

```
getFirst<number>([1, 2, 3])
```

上例中，调用函数 `getFirst()` 时，需要在函数名后面使用尖括号，给出类型参数 `T` 的值，本例是 `<number>`。

不过为了方便，函数调用时，往往省略不写类型参数的值，让 TypeScript 自己推断。

```
getFirst([1, 2, 3])
```

上例中，TypeScript 会从实际参数 `[1, 2, 3]`，推断出类型参数 `T` 的值为 `number`。

有些复杂的使用场景，TypeScript 可能推断不出类型参数的值，这时就必须显式给出了。

```
function comb<T>(arr1:T[], arr2:T[]):T[] {
  return arr1.concat(arr2);
```

```
}
```

上例中，两个参数 `arr1`、`arr2` 和返回值都是同一个类型。如果不给出类型参数的值，下面的调用会报错。

```
comb([1, 2], ['a', 'b']) // Type 'string' is not assignable to type 'number'.
```

上面示例会报错，TypeScript 认为两个参数不是同一个类型。但是，如果类型参数是一个联合类型，就不会报错。

```
comb<number|string>([1, 2], ['a', 'b']) // 正确
```

上例中，类型参数是一个联合类型，使得两个参数都符合类型参数，就不报错了。这种情况下，类型参数是不能省略不写的。

类型参数的名字，可以随便取，但是必须为合法的标识符。习惯上，类型参数的第一个字符往往采用大写字母。一般会使用T（type 的第一个字母）作为类型参数的名字。如果有多个类型参数，则使用 T 后面的 U、V 等字母命名，各个参数之间使用逗号（“,”）分隔。

```
function map<T, U>(arr:T[], f:(arg:T) => U):U[] {
  return arr.map(f);
}

// 用法实例
map<string, number>(['1', '2', '3'], (n) => parseInt(n)); // 返回 [1, 2, 3]
```

上例将数组的实例方法 `map()` 改写成全局函数，它有两个类型参数 `T` 和 `U`。含义是，原始数组的类型为 `T[]`，对该数组的每个成员执行一个处理函数 `f`，将类型 `T` 转成类型 `U`，那么就会得到一个类型为 `U[]` 的数组。

泛型可以理解成一段类型逻辑，需要类型参数来表达。有了类型参数以后，可以在输入类型与输出类型之间，建立一一对应关系。

1. 泛型的写法

泛型主要用在四个场合：函数、接口、类和别名。

1.1. 函数的泛型写法

`function` 关键字定义的泛型函数，类型参数放在尖括号中，写在函数名后面。

```
function id<T>(arg:T):T {
  return arg;
}
```

那么对于变量形式定义的函数，泛型有下面两种写法。

```
// 写法一
let myId:<T>(arg:T) => T = id;

// 写法二
let myId:{ <T>(arg:T): T } = id;
```

1.2. 接口的泛型写法

`interface` 也可以采用泛型的写法。

```
interface Box<Type> {
  contents: Type;
}
let box:Box<string>;
```

上面示例中，使用泛型接口时，需要给出类型参数的值（本例是string）。

```
interface Comparator<T> {
  compareTo(value:T): number;
}
class Rectangle implements Comparator<Rectangle> {
  compareTo(value:Rectangle): number {
    // ...
  }
}
```

上例中，先定义了一个泛型接口，然后将这个接口用于一个类。

泛型接口还有第二种写法。

```
interface Fn {
  <Type>(arg:Type): Type;
}
function id<Type>(arg:Type): Type {
  return arg;
}
let myId:Fn = id;
```

上例中，`Fn` 的类型参数 `Type` 的具体类型，需要函数 `id` 在使用时提供。所以，最后一行的赋值语句不需要给出 `Type` 的具体类型。

此外，第二种写法还有一个差异之处。那就是它的类型参数定义在某个方法之中，其他属性和方法不能使用该类型参数。前面的第一种写法，类型参数定义在整个接口，接口内部的所有属性和方法都可以使用该类型参

数。

1.3. 类的泛型写法

泛型类的类型参数写在类名后面。

```
class Pair<K, V> {
  key: K;
  value: V;
}
```

继承泛型类：

```
class A<T> {
  value: T;
}
class B extends A<any> {
}
```

上例中，类 `A` 有一个类型参数 `T`，使用时必须给出 `T` 的类型，所以类 `B` 继承时要写成 `A<any>`。

泛型也可以用在类表达式。

```
const Container = class<T> {
  constructor(private readonly data:T) {}
};
const a = new Container<boolean>(true);
const b = new Container<number>(0);
```

上例中，新建实例时，需要同时给出类型参数 `T` 和类参数 `data` 的值。

```
class C<NumType> {
  value!: NumType;
  add!: (x: NumType, y: NumType) => NumType;
}
let foo = new C<number>();
foo.value = 0;
foo.add = function (x, y) {
  return x + y;
};
```

上例中，先新建类 `C` 的实例 `foo`，然后再定义实例的 `value` 属性和 `add()` 方法。类的定义中，属性和方法后面的感叹号是非空断言，告诉 TypeScript 它们都是非空的，后面会赋值。

JavaScript 的类本质上是一个构造函数，因此也可以把泛型类写成构造函数。

```
type MyClass<T> = new (...args: any[]) => T;
// 或者
interface MyClass<T> {
  new(...args: any[]): T;
}
// 用法实例
function createInstance<T>(AnyClass: MyClass<T>, ...args: any[]):T {
  return new AnyClass(...args);
}
```

上面示例中，函数createInstance()的第一个参数AnyClass是构造函数（也可以是一个类），它的类型是MyClass，这里的T是createInstance()的类型参数，在该函数调用时再指定具体类型。

注意，泛型类描述的是类的实例，不包括静态属性和静态方法，因为这两者定义在类的本身。因此，它们不能引用类型参数。

```
class C<T> {
  static data: T;  // 报错
  constructor(public value:T) {}
}
```

上例中，静态属性 `data` 引用了类型参数 `T`，这是不可以的，因为类型参数只能用于实例属性和实例方法，所以报错了。

1.4. 类型别名的泛型写法

type 命令定义的类型别名，也可以使用泛型。

```
type Nullable<T> = T | undefined | null;
```

上例中，`Nullable<T>` 是一个泛型，只要传入一个类型，就可以得到这个类型与 `undefined` 和 `null` 的一个联合类型。

```
type Container<T> = { value: T };
const a: Container<number> = { value: 0 };
const b: Container<string> = { value: 'b' };
```

定义树形结构：

```
type Tree<T> = { value: T; left: Tree<T> | null; right: Tree<T> | null; };
```

上例中，类型别名 `Tree` 内部递归引用了 `Tree` 自身。

2. 类型参数的默认值

类型参数可以设置默认值。使用时，如果没有给出类型参数的值，就会使用默认值。

```
function getFirst<T = string>( arr:T[]):T {
  return arr[0];
}
```

上例中，`T = string` 表示类型参数的默认值是 `string`。调用 `getFirst()` 时，如果不给出 `T` 的值，TypeScript 就认为 `T` 等于 `string`。

但是，因为 TypeScript 会从实际参数推断出 `T` 的值，从而覆盖掉默认值，所以下面的代码不会报错。

```
getFirst([1, 2, 3]); // 正确，根据实际参数覆盖掉默认值，即便实际参数和默认值类型不同
```

上例中，实际参数是 `[1, 2, 3]`，TypeScript 推断 `T` 等于 `number`，从而覆盖掉默认值 `string`。

类型参数的默认值，往往用在类中。

```
class Generic<T = string> {
  list:T[] = []
  add(t:T) {
    this.list.push(t)
  }
}
```

上例中，类 `Generic` 有一个类型参数 `T`，默认值为 `string`。这意味着，属性 `list` 默认是一个字符串数组，方法 `add()` 的默认参数是一个字符串。

```
const g = new Generic();
g.add(4) // Argument of type 'number' is not assignable to parameter of type
'string'.
g.add('hello') // 正确
```

上例中，新建 `Generic` 的实例 `g` 时，没有给出类型参数 `T` 的值，所以 `T` 就等于 `string` 。因此，向 `add()` 方法传入一个数值会报错，传入字符串就不会。

```
const g = new Generic<number>();
g.add(4) // 正确
g.add('hello') // Argument of type 'string' is not assignable to parameter of type
'number'.
```

上例中，新建实例 `g` 时，给出了类型参数 `T` 的值是 `number`，因此 `add()` 方法传入数值不会报错，传入字符串会报错。

一旦类型参数有默认值，就表示它是可选参数。如果有多个类型参数，可选参数必须在必选参数之后。

```
<T = boolean, U> // 错误
<T, U = boolean> // 正确
```

上例中，依次有两个类型参数 `T` 和 `U`。如果 `T` 是可选参数，`U` 不是，就会报错。

3. 数组的泛型表示

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

上例中，`Array<number>` 就是一个泛型，类型参数的值是 `number`，表示该数组的全部成员都是数值。

在 TypeScript 内部，Array是一个泛型接口，类型定义基本是下面的样子。

```
interface Array<Type> {
  length: number;
  pop(): Type|undefined;
  push(...items:Type[]): number;
  // ...
}
```

上面代码中，`push()` 方法的参数 `item` 的类型是 `Type[]`，跟 `Array()` 的参数类型 `Type` 保持一致，表示只能添加同类型的成员。调用 `push()` 的时候，TypeScript 就会检查两者是否一致。

其他的 TypeScript 内部数据结构，比如 `Map`、`Set` 和 `Promise`，其实也是泛型接口，完整的写法是 `Map<K, V>`、`Set<T>` 和 `Promise<T>`。

TypeScript 默认还提供一个 `ReadonlyArray<T>` 接口，表示只读数组。

```
function doStuff(values:ReadonlyArray<string>) {
  values.push('hello!');  // Property 'push' does not exist on type 'readonly
string[]'.
}
```

上例中，参数 `values` 的类型是 `ReadonlyArray<string>`，表示不能修改这个数组，所以函数体内部新增数组成员就会报错。因此，如果不希望函数内部改动参数数组，就可以将该参数数组声明为 `ReadonlyArray<T>` 类型。

4. 类型参数的约束条件

很多类型参数并不是无限制的，对于传入的类型存在约束条件。

```
function comp<Type>(a:Type, b:Type) {
  if (a.length >= b.length) {
    return a;
  }
  return b;
}
```

上例中，类型参数 `Type` 有一个隐藏的约束条件：它必须存在 `length` 属性。如果不满足这个条件，就会报错。

TypeScript 提供了一种语法，允许在类型参数上面写明约束条件，如果不满足条件，编译时就会报错。这样也可以有良好的语义，对类型参数进行说明。

```
function comp<T extends { length: number }>(a: T, b: T) {
  if (a.length >= b.length) {
    return a;
  }
  return b;
}
comp([1, 2], [1, 2, 3]) // 正确
comp('ab', 'abc') // 正确
comp(1, 2) // 'number'类型的参数不能赋值给'{length: number;}”。
```

上例中，`T extends { length: number }` 就是约束条件，表示类型参数 `T` 必须满足 `{ length: number }`，否则就会报错。

类型参数的约束条件采用：`<TypeParameter extends ConstraintType>`。`TypeParameter` 表示类型参数，`extends` 是关键字，这是必须的，`ConstraintType` 表示类型参数要满足的条件，即类型参数应该是 `ConstraintType` 的子类型。

类型参数可以同时设置约束条件和默认值，前提是默认值必须满足约束条件。

```
type Fn<A extends string, B extends string = 'world'> =  [A, B];
type Result = Fn<'hello'>; // type Result = ["hello", "world"]
type Result2 = Fn<'hello', 'world2'>;  // type Result = ["hello", "world2"]
```

上例中，类型参数 `A` 和 `B` 都有约束条件，并且 `B` 还有默认值。所以，调用 `Fn` 的时候，可以只给出 `A` 的值，不给出 `B` 的值。

另外，上例也可以看出，泛型本质上是一个类型函数，通过输入参数，获得结果，两者是一一对应关系。

如果有多个类型参数，一个类型参数的约束条件，可以引用其他参数。

```
<T, U extends T>
// 或者
<T extends U, U>
```

上例中，U 的约束条件引用 T，或者 T 的约束条件引用 U，都是正确的。

但是，约束条件不能引用类型参数自身。

```
<T extends T> // Type parameter 'T' has a circular constraint. 类型参数“T”有一个循环约束。
<T extends U, U extends T> // Type parameter 'T' has a circular constraint. Type parameter 'U' has a circular constraint.
```

上例中，T 的约束条件不能是 T 自身。同理，多个类型参数也不能互相约束（即 T 的约束条件是 U、U 的约束条件是 T），因为互相约束就意味着约束条件就是类型参数自身。

5. 使用注意点

泛型有一些使用注意点。

- （1）尽量少用泛型。

泛型虽然灵活，但是会加大代码的复杂性，使其变得难读难写。**一般来说，只要使用了泛型，类型声明通常都不太易读，容易写得很复杂。因此，可以不用泛型就不要用。**

- （2）类型参数越少越好。

多一个类型参数，多一道替换步骤，加大复杂性。因此，类型参数越少越好。

```
function filter<T, Fn extends (arg:T) => boolean>(arr:T[], func:Fn): T[] {
  return arr.filter(func);
}
```

上例有两个类型参数，但是第二个类型参数 Fn 是不必要的，完全可以直接写在函数参数的类型声明里面。

```
function filter<T>(arr:T[], func:(arg:T) => boolean): T[] {
  return arr.filter(func);
}
```

上面示例中，类型参数简化成了一个，效果与前一个示例是一样的。

- （3）类型参数需要出现两次。

如果类型参数在定义后只出现一次，那么很可能是不必要的。

```
function greet<Str extends string>(s:Str) {
  console.log('Hello, ' + s);
}
```

上例中，类型参数 `Str` 只在函数声明中出现一次（除了它的定义部分），这往往表明这个类型参数是不必要。

```
function greet(s:string) {
  console.log('Hello, ' + s);
}
```

上例把前面的类型参数省略了，效果与前一个示例是一样的。**也就是说，只有当类型参数用到两次或两次以上，才是泛型的适用场合。**

- （4）泛型可以嵌套。

类型参数可以是另一个泛型。

```
type OrNull<Type> = Type|null;
type OneOrMany<Type> = Type|Type[];
type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>; // OneOrMany<Type> | null
<=>  Type | Type[] | null
```

上例中，最后一行的泛型 `OrNull` 的类型参数，就是另一个泛型 `OneOrMany`。

Enum 类型

Enum（枚举）是 TypeScript 新增的一种数据结构和类型，用来将相关常量放在一个容器里面。

```
enum Color {
    Red,     // 0
    Green,   // 1
    Blue     // 2
}

console.log(Color.Red);  // 0
console.log(Color['Green']); // 1
console.log(Color['Blue']); // 2
```

三个成员 `Red`、`Green` 和 `Blue`。第一个成员的值默认为整数 `0`，第二个为 `1`，第三个为 `2`，以此类推。调用 Enum 的某一个成员，与调用对象属性的写法一样，可以使用点运算符，也可以使用方扩号运算符。

```
let c:Color = Color.Green; // 正确
let c:number = Color.Green; // 正确
```

上例中，变量 `c` 的类型写成 `Color` 或 `number` 都可以。但是，`Color` 类型的语义更好。

Typescript 代码编译前:

```
enum Color {
  Red,
  Green,
  Blue
}
```

JavaScript 代码编译后:

```
"use strict";
var Color;
(function (Color) {
    Color[Color["Red"] = 0] = "Red";
    Color[Color["Green"] = 1] = "Green";
    Color[Color["Blue"] = 2] = "Blue";
})(Color || (Color = {}));
```

由于 TypeScript 的定位是 JavaScript 语言的类型增强，所以官方建议谨慎使用 Enum 结构，因为它不仅仅是类型，还会为编译后的代码加入一个对象。

Enum 结构比较适合的场景是，成员的值不重要，名字更重要，从而增加代码的可读性和可维护性。

```
enum Operator {
  ADD,
  DIV,
  MUL,
  SUB
}

function compute(
  op:Operator,
  a:number,
  b:number
) {
  switch (op) {
    case Operator.ADD:
      return a + b;
    case Operator.DIV:
      return a / b;
    case Operator.MUL:
      return a * b;
    case Operator.SUB:
      return a - b;
    default:
      throw new Error('wrong operator');
  }
}

compute(Operator.ADD, 1, 3) // 4
compute(Operator.DIV, 6, 3) // 2
compute(Operator.ADD, 1, 3) // 3
compute(Operator.ADD, 3, 1) // 2
```

由于 Enum 结构编译后是一个对象，所以不能有与它同名的变量（包括对象、函数、类等）。

```
enum Color {
  Red,
  Green,
  Blue
}

const Color = 'red'; // Enum declarations can only merge with namespace or other
enum declarations. 枚举声明只能与命名空间或其他枚举声明合并。
```

上例中，Enum 结构与变量同名，导致报错。

Enum 结构可以被对象的 `as const` 断言替代。

```
enum Foo {
  A,
  B,
  C,
}

const Bar = {
  A: 0,
  B: 1,
  C: 2,
} as const;

let x = 2;
if (x === Foo.A) {}
// 等同于
if (x === Bar.A) {}
```

上例中，对象 `Bar` 使用了 `as const` 断言，作用就是使得它的属性无法修改。这样的话，`Foo` 和 `Bar` 的行为就很类似了，前者完全可以用后者替代，而且后者还是 JavaScript 的原生数据结构。

1. Enum 成员的值

Enum 成员默认不必赋值，系统会从零开始逐一递增，按照顺序为每个成员赋值，比如 0、1、2……，但是，也可以为 Enum 成员显式赋值。

```
enum Color {
  Red,
  Green,
  Blue
}

// 等同于
enum Color {
  Red = 0,
  Green = 1,
  Blue = 2
}
```

成员的值可以是任意数值，但不能是大整数（Bigint）。

```
// Enum 成员的值可以是小数，但不能是 Bigint。
enum Color {
  Red = 90,
  Green = 0.5,
  Blue = 7n // Type 'bigint' is not assignable to type 'number' as required for
computed enum member values. 类型'bigint'不能赋值给类型'number'作为计算枚举成员值的要
求。
}
```

成员的值甚至可以相同。

```
enum Color {
  Red = 0,
  Green = 0,
  Blue = 0
}
console.log(Color3.Red);  // 0
console.log(Color3['Green']); // 0
console.log(Color3['Blue']); // 0
```

如果只设定第一个成员的值，后面成员的值就会从这个值开始递增。

```
enum Color4 {
  Red = 7,
  Green,
  Blue
}
console.log(Color4.Red);  // 7
console.log(Color4['Green']); // 8
console.log(Color4['Blue']); // 9

enum Color5 {
  Red,
  Green = 7,
  Blue
}
console.log(Color5.Red);  // 0
console.log(Color5['Green']); // 7
console.log(Color5['Blue']); // 8
```

Enum 成员的值也可以使用计算式，和函数的返回值。

```
enum MyEnum1 {
  A = 123,
  B = 1 + 2
}

enum MyEnum2 {
  A = 123,
  B = Math.random(),
}
```

Enum 成员值都是只读的，不能重新赋值。

```
enum Color {
  Red,
  Green,
  Blue
}

Color.Red = 4; // Cannot assign to 'Red' because it is a read-only property. 不能赋值给'Red', 因为它是一个只读属性。
```

通常会在 enum 关键字前面加上 `const` 修饰，表示这是常量，不能再次赋值。

```
const enum Color {
  Red,
  Green,
  Blue
}
```

加上 `const` 还有一个好处，就是编译为 JavaScript 代码后，代码中 Enum 成员会被替换成对应的值，这样能提高性能表现。

```
const enum Color {
  Red,
  Green,
  Blue
}

const x = Color.Red;
const y = Color.Green;
const z = Color.Blue;

// 编译后
"use strict";
const x = 0 /* Color.Red */;
const y = 1 /* Color.Green */;
const z = 2 /* Color.Blue */;
```

由于 Enum 结构前面加了 `const` 关键字，所以编译产物里面就没有生成对应的对象，而是把所有 Enum 成员出现的场合，都替换成对应的常量。

2. 同名 Enum 的合并

多个同名的 Enum 结构会自动合并。

```
enum Foo {
  A,
}
```

```
enum Foo {
  B = 1,
}

enum Foo {
  C = 2,
}

// 等同于
enum Foo {
  A,
  B = 1,
  C = 2
}
```

Enum 结构合并时，只允许其中一个的首成员省略初始值，否则报错。

```
enum Foo {
  A,
}

enum Foo {
  B, // In an enum with multiple declarations, only one declaration can omit an initializer for its first enum element. 在具有多个声明的枚举中，只有一个声明可以省略其第一个枚举元素的初始化式。
}
```

同名 Enum 合并时，不能有同名成员，否则报错。

```
enum Foo {
  A,
  B
}

enum Foo {
  B = 1, //  Duplicate identifier 'B'. 重复标识符'B'。
  C
}
```

同名 Enum 合并的另一个限制是，所有定义必须同为 const 枚举或者非 const 枚举，不允许混合使用。

```
// 正确
enum E1 {
  A,
}
enum E1 {
```

```
  B = 1,
}

// 正确
const enum E2 {
  A,
}
const enum E2 {
  B = 1,
}

// Enum declarations can only merge with namespace or other enum declarations. 枚举声明只能与命名空间或其他枚举声明合并。
enum E3 {
  A,
}
const enum E3 {
  B = 1,
}
```

同名 Enum 的合并，最大用处就是补充外部定义的 Enum 结构。

3. 字符串 Enum

Enum 成员的值除了设为数值，还可以设为字符串。也就是说，Enum 也可以用作一组相关字符串的集合。

```
enum Direction {
  Up = 'UP',
  Down = 'DOWN',
  Left = 'LEFT',
  Right = 'RIGHT',
}
```

Direction 就是字符串枚举，每个成员的值都是字符串。

字符串枚举的所有成员值，都必须显式设置。如果没有设置，成员值默认为数值，且位置必须在字符串成员之前。

```
enum Foo1 {
  A, // 0,  第一个成员不设初始值默认为 0
  B = 'hello',
  C //  Enum member must have initializer. 枚举成员必须有初始化式。
}

// 成员的类型可以是 number 和 string，两者可以混合在一个 Enum 中
enum Foo2 {
  A = 1,
  B = 'hello',
  C = 0
```

```
}

enum Foo3 {
  A, // 0
  B = 'hello',
  C = 'world'
}
```

A 之前没有其他成员，所以可以不设置初始值，默认等于 0；C 之前有一个字符串成员，所以 C 必须有初始值，不赋值就报错了。

Enum 成员可以是字符串和数值混合赋值。

```
enum Enum {
  One = 'One',
  Two = 'Two',
  Three = 3,
  Four = 4,
}
```

除了数值和字符串，Enum 成员不允许使用其他值（比如 Symbol、Boolean）。

```
enum Foo {
  A = true // Type 'boolean' is not assignable to type 'number' as required for
computed enum member values. 类型“boolean”不能赋值给类型“number”，因为计算枚举成员值
需要赋值。
}

enum Foo2 {
  A = true // Type 'symbol' is not assignable to type 'number' as required for
computed enum member values. 类型“symbol”不能赋值给类型“number”，因为计算枚举成员值需
要赋值。
}
```

变量类型如果是字符串 Enum，就不能再赋值为字符串，这跟数值 Enum 不一样。

```
enum MyEnum {
  One = 'One',
  Two = 'Two',
}

let s = MyEnum.One;
s = 'One'; // Type '"One"' is not assignable to type 'MyEnum'.
```

上例中，变量s的类型是MyEnum，再赋值为字符串就报错。由于这个原因，如果函数的参数类型是字符串 Enum，传参时就不能直接传入字符串，而要传入 Enum 成员。

```
enum MyEnum {
  One = 'One',
  Two = 'Two',
}

function f(arg:MyEnum) {
  return 'arg is ' + arg;
}

f(MyEnum['One']) // 正确
f(MyEnum.One) // 正确

f('One') //  Argument of type '"One"' is not assignable to parameter of type
'MyEnum'.
```

Enum 成员值可以保存一些有用的信息，所以 TypeScript 才设计了字符串 Enum。

```
const enum MediaTypes {
  JSON = 'application/json',
  XML = 'application/xml',
}
const url = 'localhost';
fetch(url, {
  headers: {
    Accept: MediaTypes.JSON,
  },
}).then(response => {
  // ...
});
```

上例中，函数 `fetch()` 的参数对象的属性 `Accept` ，只能接受一些指定的字符串。这时就很适合把字符串放进一个 Enum 结构，通过成员值来引用这些字符串。

字符串 Enum 可以使用联合类型（union）代替。 效果跟指定为字符串 Enum 是一样的

```
function move (where:'Up'|'Down'|'Left'|'Right') {
  // ...
}
```

字符串 Enum 的成员值，不能使用字符串表达式赋值。可以使用数值表达式、函数返回值，和可以转换为数值的计算值。

```
enum MyEnum1 {
  A = 'one',
  B = ['1', '2', '3'].join('') // Type 'string' is not assignable to type 'number'
as required for computed enum member values. 类型“string”不能按计算枚举成员值的要求赋
```

```
值给类型“number”。
}
enum MyEnum2 {
  A = 'one',
  B = String(1) // Type 'string' is not assignable to type 'number' as required
for computed enum member values.
}
enum MyEnum3 {
  A = 123,
  B = ['1', '2'][0] // Type 'string' is not assignable to type 'number' as
required for computed enum member values.
}

enum MyEnum10 {
  A = 'one',
  B = Number(['1', '2', '3'].join('')), // 正确，数值表达式
}
enum MyEnum11 {
  A = 'one',
  B = Number('1') // 正确
}
enum MyEnum12 {
  A = 123,
  B = Number(['1', '2'][0]) // 正确
}
enum MyEnum13 {
  A = 123,
  B = [1, 2][0] // 正确
}
enum MyEnum14 {
  A = 123,
  B = Math.random(), // 正确，函数返回值
}
enum MyEnum15 {
  A = 123,
  B = 1 + 2, // 正确，数值计算值
}
enum MyEnum16 {
  A = 123,
  B = '1' + '2', // 正确
}
type e = {[key in MyEnum16]: number};  // type e = { 123: number; 12: number; }
```

成员 `B` 的值是一个字符串表达式，导致报错。

4. keyof 运算符

keyof 运算符可以取出 Enum 结构的所有成员名，作为联合类型返回。

```
enum MyEnum {
  A = 'a',
```

```
  B = 'b'
}

type Foo = keyof typeof MyEnum;  // type Foo = "A" | "B"
```

`keyof typeof MyEnum` 可以取出 `MyEnum` 的所有成员名，所以类型 `Foo` 等同于联合类型`'A'|'B'`。

这里的 `typeof` 是必需的，否则 `keyof MyEnum` 相当于 `keyof number` 。

```
type Foo = keyof MyEnum;
// "toString" | "toFixed" | "toExponential" | "toPrecision" | "valueOf" |
"toLocaleString"
```

这是因为 Enum 作为类型，本质上属于 `number` 或 `string` 的一种变体，而 `typeof MyEnum` 会将 `MyEnum` 当作一个值处理，从而先其转为对象类型，就可以再用 `keyof` 运算符返回该对象的所有属性名。

如果要返回 Enum 所有的成员值，可以使用 `in` 运算符。

```
enum MyEnum13 {
  A = 'a',
  B = 'b'
}
type Foo13 = { [key in MyEnum]: number }; // type Foo13 = { a: number; b: number;
}
```

5. 反向映射

数值 Enum 存在反向映射，即可以通过成员值获得成员名。

```
enum Weekdays {
  Monday = 1,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
}

console.log(Weekdays[3]) // Wednesday
console.log(Weekdays['Wednesday']) // 3
```

上例中，Enum 成员 `Wednesday` 的值等于 `3`，从而可以从成员值 `3` 取到对应的成员名 `Wednesday`，这是反向映射。

这是因为 TypeScript 会将上面的 Enum 结构，编译成下面的 JavaScript 代码。

```
"use strict";
var Weekdays;
(function (Weekdays) {
    Weekdays[Weekdays["Monday"] = 1] = "Monday";
    Weekdays[Weekdays["Tuesday"] = 2] = "Tuesday";
    Weekdays[Weekdays["Wednesday"] = 3] = "Wednesday";
    Weekdays[Weekdays["Thursday"] = 4] = "Thursday";
    Weekdays[Weekdays["Friday"] = 5] = "Friday";
    Weekdays[Weekdays["Saturday"] = 6] = "Saturday";
    Weekdays[Weekdays["Sunday"] = 7] = "Sunday";
})(Weekdays || (Weekdays = {}));
```

上面代码中，实际进行了两组赋值，以第一个成员为例。

```
Weekdays[Weekdays["Monday"] = 1] = "Monday";
```

上面代码有两个赋值运算符（=），实际上等同于下面的代码。

```
Weekdays["Monday"] = 1;
Weekdays[1] = "Monday";
```

这种情况只发生在数值 Enum，对于字符串 Enum，不存在反向映射。这是因为字符串 Enum 编译后只有一组赋值。

编译前

```
enum MyEnum {
  A = 'a',
  B = 'b'
}
```

编译后：

```
"use strict";
var MyEnum;
(function (MyEnum) {
    MyEnum["A"] = "a";
    MyEnum["B"] = "b";
})(MyEnum || (MyEnum = {}));
```

类型断言

TypeScript 提供了“类型断言”这样一种手段，允许开发者在代码中“断言”某个值的类型，告诉编译器此处的值是什么类型。TypeScript 一旦发现存在类型断言，就不再对该值进行类型推断，而是直接采用断言给出的类型。

类型断言有两种语法。

```
// 语法一：<类型>值
<Type>value

// 语法二：值 as 类型
value as Type
```

上面两种语法是等价的，`value` 表示值，`Type` 表示类型。早期只有语法一，后来因为 TypeScript 开始支持 React 的 JSX 语法（尖括号表示 HTML 元素），为了避免两者冲突，就引入了语法二。目前，推荐使用语法二。

```
type T = 'a'|'b'|'c';
let foo = 'a';

// 语法一
let bar:T = <T>foo; // JSX element 'T' has no corresponding closing tag.

// 语法二
let bar:T = foo as T;
```

上面示例是两种类型断言的语法，其中的语法一因为跟 JSX 语法冲突，使用时必须关闭 TypeScript 的 React 支持，否则会无法识别。由于这个原因，现在一般都使用语法二。

```
const username = document.getElementById('username'); // const username:
HTMLElement | null
(username as HTMLInputElement).value; // 正确
HTMLInputElement.value; // Property 'value' does not exist on type '{ new (): 
HTMLInputElement; prototype: HTMLInputElement; }'.
```

上例中，变量 `username` 的类型是 `HTMLElement | null`，`null` 类型是没有 `value` 属性的。如果 `username` 是一个输入框，那么就可以通过类型断言，将它的类型改成 `HTMLInputElement`，就可以读取 `value` 属性。

类型断言不应滥用，因为它改变了 TypeScript 的类型检查，很可能埋下错误的隐患。

```
const data:object = {
  a: 1,
  b: 2,
```

```
  c: 3
};
data.length; // Property 'length' does not exist on type 'object'.
(data as Array<string>).length; // 正确
```

变量 data 是一个对象，没有 length 属性。但是通过类型断言，可以将它的类型断言为数组，这样使用 length 属性就能通过类型检查。但是，编译后的代码在运行时依然会报错，所以类型断言可以让错误的代码通过编译。

类型断言可以指定 unknown 类型的变量的具体类型。

```
const value:unknown = 'Hello World';
const s1:string = value; // Type 'unknown' is not assignable to type 'string'.
const s2:string = value as string; // 正确
```

类型断言也适合指定联合类型的值的具体类型。

```
const s1:number|string = 123;
const s2:number = s1 as number; // 断言 s1 类型为 number，便可以给类型为 number 的变量 s2 赋值
```

上例中，变量 s1 是联合类型，可以断言其为联合类型里面的一种具体类型，再将其赋值给变量 s2。

1. 类型断言的条件

类型断言并不意味着，可以把某个值断言为任意类型。

```
const n = 1;
const m:string = n as string; // Conversion of type 'number' to type 'string' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.
```

上例中，变量 n 是数值类型 '1'，是 number 类型的子类型，无法把它断言成字符串，TypeScript 会报错。

类型断言的使用前提是，值的实际类型与断言的类型必须满足一个条件。expr as T，expr 是实际的值，T 是类型断言，它们必须满足下面的条件：expr 是 T 的子类型，或者 T 是 expr 的子类型。

也就是说，**类型断言要求实际的类型与断言的类型兼容，实际类型可以断言为一个更加宽泛的类型（父类型），也可以断言为一个更加精确的类型（子类型），但不能断言为一个完全无关的类型。**

```
const n1 = 1; // const n1: 1, 数值类型 1
const m1:number = n1 as number; // 断言为父类型 number

const n2:number = 1; // const n2: number, number 类型
const m2:number = n2 as 1; // 断言为数值类型 1
```

但是，如果真的要断言成一个完全无关的类型，也是可以做到的。那就是连续进行两次类型断言，**先断言成 unknown 类型或 any 类型，然后再断言为目标类型。** 因为 any 类型和 unknown 类型是所有其他类型的父类型，所以可以作为两种完全无关的类型的中介。

```
expr as unknown as T
```

上例中，expr 连续进行了两次类型断言，第一次断言为 unknown 类型，第二次断言为 T 类型。这样的话，expr 就可以断言成任意类型 T，而不报错。

```
const n = 1;
const m:string = n as unknown as string; // 正确
```

通过两次类型断言，变量 n 的类型就从数值，变成了完全无关的字符串，从而赋值时不会报错。

2. as const 断言

如果没有声明变量类型，let 命令声明的变量，会被类型推断为 TypeScript 内置的基本类型之一；const 命令声明的变量，则被推断为值类型常量。

```
let s1 = 'JavaScript'; // let s1: string
const s2 = 'JavaScript'; // const s2: "JavaScript"
```

上例中，变量 s1 的类型被推断为 string，变量 s2 的类型推断为值类型 JavaScript 。**后者是前者的子类型，相当于 const 命令有更强的限定作用，可以缩小变量的类型范围。**

有些时候，let 变量会出现一些意想不到的报错，变更成 const 变量就能消除报错。

```
let s = 'JavaScript';
type Lang =
  |'JavaScript'
  |'TypeScript'
  |'Python';
function setLang(language:Lang) {
  /* ... */
}
setLang(s); // Argument of type 'string' is not assignable to parameter of type
'Lang'.
```

函数 setLang() 的参数 language 类型是 Lang，这是一个联合类型。传入的字符串 s 的类型被推断为 string，属于 Lang 的父类型。父类型不能替代子类型，导致报错。

一种解决方法就是把 let 命令改成 const 命令。

```
const s = 'JavaScript';
```

这样的话，变量 s 的类型就是值类型 "JavaScript"，它是联合类型 Lang 的子类型，传入函数 setLang() 就不会报错。

另一种解决方法是使用类型断言。TypeScript 提供了一种特殊的类型断言 as const，用于告诉编译器，推断类型时，可以将这个值推断为常量，即把 let 变量断言为 const 变量，从而把内置的基本类型变更为值类型。

```
let s = 'JavaScript' as const;
setLang(s);  // 正确
```

变量 s 虽然是用 let 命令声明的，但是使用了 as const 断言以后，就等同于是用 const 命令声明的，变量s的类型会被推断为*值类型* *"JavaScript"*。使用了 as const 断言以后，let 变量就不能再改变值了。

```
let s = 'JavaScript' as const; // let s: "JavaScript"
s = 'Python'; // Type '"Python"' is not assignable to type '"JavaScript"'.
```

let 命令声明的变量 s，使用 as const 断言以后，就不能改变值了，否则报错。

as const 断言只能用于字面量，不能用于变量。

```
let s1 = 'JavaScript';
let s2 = s1 as const; // A 'const' assertions can only be applied to references to enum members, or string, number, boolean, array, or object literals.'const'断言只能应用于对枚举成员或字符串、数字、布尔值、数组或对象字面值的引用。
```

as const也不能用于表达式。

```
let s = ('Java' + 'Script') as const; //  // A 'const' assertions can only be applied to references to enum members, or string, number, boolean, array, or object literals.'const'断言只能应用于对枚举成员或字符串、数字、布尔值、数组或对象字面值的引用。
```

as const断言可以用于整个对象，也可以用于对象的单个属性，这时它的类型缩小效果是不一样的。

```
const v1 = {
  x: 1,
  y: 2,
}; // const v1: { x: number; y: number; }

const v2 = {
```

```
  x: 1 as const, // 对属性x缩小类型
  y: 2,
}; // const v2: { x: 1; y: number; }

// 对整个对象缩小类型
const v3 = {
  x: 1,
  y: 2,
} as const; // const v3: { readonly x: 1; readonly y: 2; }
```

`as const` **会将字面量的类型断言为不可变类型，缩小成 TypeScript 允许的最小类型。**

```
const a1 = [1, 2, 3]; // const a1: number[]
let a2 = [1, 2, 3]; // let a2: number[]

const a3 = [1, 2, 3] as const; // const a3: readonly [1, 2, 3]
let a4 = [1, 2, 3] as const; // let a4: readonly [1, 2, 3]
```

数组字面量使用 `as const` 断言后，类型推断就变成了只读元组。由于 `as const` 会将数组变成只读元组，所以很适合用于函数的 `rest` 参数。

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

const numbers = [1, 2]; // const numbers: number[]
const total = add(...numbers); // A spread argument must either have a tuple type
or be passed to a rest parameter. 扩展参数必须具有元组类型，或者传递给rest形参。
```

变量 `numbers` 的类型推断为 `number[]`，导致使用扩展运算符...传入函数 `add()` 会报错，因为 `add()` 只能接受两个参数，而 `...numbers` 并不能保证参数的个数。

事实上，对于固定参数个数的函数，如果传入的参数包含扩展运算符，那么扩展运算符只能用于元组。只有当函数定义使用了 rest 参数，扩展运算符才能用于数组。

解决方法就是使用 `as const` 断言，将数组变成元组。

```
const numbers = [1, 2] as const; // const numbers: readonly [1, 2]
const total = add(...numbers);
```

上例中，使用 `as const` 断言后，变量 `numbers` 的类型会被推断为 `readonly [1, 2]`，使用扩展运算符展开后，正好符合函数 `add()` 的参数类型。

Enum 成员也可以使用 `as const` 断言。

```
enum Foo {
  X,
  Y,
}
let e1 = Foo.X; // let e1: Foo
let e2 = Foo.X as const; // let e2: Foo.X
```

上例中，如果不使用 `as const` 断言，变量 `e1` 的类型被推断为整个 `Enum` 类型；使用了 `as const` 断言以后，变量 `e2` 的类型被推断为 Enum 的某个成员，这意味着它不能变更为其他成员。

3. 非空断言

对于那些可能为空的变量（即可能等于 `undefined` 或 `null`），TypeScript 提供了非空断言，保证这些变量不会为空，写法是在变量名后面加上感叹号 `!`。

```
const root = document.getElementById('root');

// 'root' is possibly 'null'.
root.addEventListener('click', e => {
  /* ... */
});
```

如果可以确认root元素肯定会在网页中存在，这时就可以使用非空断言。

```
const root = document.getElementById('root')!;
root!.addEventListener('click', e => {
  /* ... */
});
```

`getElementById()` 方法加上后缀 `!`，或者 `root` 加上非空断言 `!` 表示这个方法肯定返回非空结果。

比较保险的做法还是手动检查一下是否为空。

```
const root = document.getElementById('root');
if (root === null) {
  throw new Error('Unable to find DOM element #root');
}
root.addEventListener('click', e => {
  /* ... */
});
```

4. 断言函数

断言函数是一种特殊函数，用于保证函数参数符合某种类型。如果函数参数达不到要求，就会抛出错误，中断程序执行；如果达到要求，就不进行任何操作，让代码按照正常流程运行。

```
function isString(value:unknown):asserts value is string {
  if (typeof value !== 'string')
    throw new Error('Not a string');
}
```

上例中，函数 `isString()` 的返回值类型写成 `asserts value is string`，其中 `asserts` 和 `is` 都是关键词，`value` 是函数的参数名，`string` 是函数参数的预期类型。它的意思是，该函数用来断言参数 `value` 的类型是 `string` ，如果达不到要求，程序就会在这里中断。

断言函数的asserts语句等同于void类型，所以如果返回除了undefined和null以外的值，都会报错。

```
function isString(value:unknown):asserts value is string {
  if (typeof value !== 'string')
    throw new Error('Not a string');
  return true; // 报错
}
```

上例中，断言函数返回了true，导致报错。

```
type AccessLevel = 'r' | 'w' | 'rw';

function allowsReadAccess(level:AccessLevel):asserts level is 'r' | 'rw' {
  if (!level.includes('r'))
    throw new Error('Read not allowed');
}
```

函数 `allowsReadAccess()` 用来断言参数 `level` 一定等于 `r` 或 `rw`。

使用断言函数的同时又想定义函数返回类型，可以使用箭头函数：

```
type AssertIsNumber = (value:unknown) => asserts value is number;

const assertIsNumber:AssertIsNumber = (value): string => {
    if (typeof value !== 'number')
        throw Error('Not a number');
    return "value is " + value;
};
```