TypeScript 学习笔记

TS 的特点

- TypeScript 是 JavaScript 的超集，.js 文件可以直接重命名为 .ts 即可，用 TypeScript 编写 React 时，可以 .tsx 为后缀。

  TypeScript 遵循最新 ES6 和 ES5 规范，并且扩展了 JavaScript 的语法，引入了诸如接口、泛型等功能。

- TypeScript 可以定义从简单到复杂的几乎一切类型。

  不显式定义的类型，也能够自动做出类型推断。

- 类型系统实际上是最好的文档，大部分的函数看看类型的定义就可以知道如何使用了。

  本质上就是良好的注释，可以有更好的代码自动补全功能。

- 可以在编译阶段就发现大部分错误，这总比在运行时候出错好。

- TypeScript 能够直接编译成 JavaScript ，

  编译后的 JavaScript 可以运行到任何符合版本的浏览器上，并且现今的流行框架都可以继承 TypeScript 。

- 编译错误的代码片段，默认为编译通过，

  也就是说 **TypeScript 编译的时候即使报错了，还是会生成编译结果，我们仍然可以使用这个编译之后的 JavaScript 文件。**

- 兼容第三方库，即使第三方库不是用 TypeScript 写的，也可以编写单独的类型文件供 TypeScript 读取。

安装

可以通过下面的方式在全局安装 TypeScript

```
npm install -g typescript
或 yarn add typescript -g
```

安装 ts-node，这个插件可以直接运行 .ts 文件，但是不会编译出来 .js 文件。

```
npm i -g ts-node

// 运行 ts-node index.ts
```

start 脚本添加到 package.json

```
{
  "scripts": {
    "build": "tsc",
    "start": "ts-node index.ts"
  }
}
```

执行 npm start 即可运行代码。

创建一个 tsconfig.json 文件

```
tsc --init
```

创建一个 `ts` 文件夹放置测试代码文件，然后创建一个 `index.ts` 文件在 `ts` 文件夹下。

然后在命令行通过 `cd` 命令进入项目所在的目录路径，安装 TypeScript 开发的两个主要依赖包：

1. typescript 这个包是用 TypeScript 编程的语言依赖包
2. ts-node 是让 Node 可以运行 TypeScript 的执行环境

然后修改 scripts 字段，增加一个 `dev:ts` 的 script：

```
{
  "name": "hello-node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev:cjs": "node src/cjs/index.cjs",
    "dev:esm": "node src/esm/index.mjs",
    "dev:ts": "ts-node src/ts/index.ts",
    "compile": "babel src/babel --out-dir compiled",
    "serve": "node server/index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "md5": "^2.3.0"
  },
  "devDependencies": {
    "ts-node": "^10.7.0",
    "typescript": "^4.6.3"
  }
}
```

准备工作完毕！

> 请注意，`dev:ts` 这个 script 是用了 `ts-node` 来代替原来在用的 `node`，因为使用 `node` 无法识别 TypeScript 语言。

`src/ts/index.ts` 里：

```
// src/ts/index.ts
function getFirstWord(msg) {
  console.log(msg.split(' ')[0])
}

getFirstWord('Hello World')

getFirstWord(123)
```

然后在命令行运行 `npm run dev:ts` 来看看这次的结果：

```
TSError: × Unable to compile TypeScript:
src/ts/index.ts:1:23 - error TS7006: Parameter 'msg' implicitly has an 'any'
type.

1 function getFirstWord(msg) {
                        ~~~
```

这是告知 `getFirstWord` 的入参 `msg` 带有隐式 any 类型，

这个时候可能还不了解 any 代表什么意思，没关系，来看下如何修正这段代码：

```
// src/ts/index.ts
function getFirstWord(msg: string) {
  console.log(msg.split(' ')[0])
}

getFirstWord('Hello World')

getFirstWord(123)
```

留意到没有，现在函数的入参 `msg` 已经变成了 `msg: string` ，这是 TypeScript 指定参数为字符串类型的一个写法。

现在再运行 `npm run dev:ts` ，上一个错误提示已经不再出现，取而代之的是一个新的报错：

```
TSError: × Unable to compile TypeScript:
src/ts/index.ts:7:14 - error TS2345:
Argument of type 'number' is not assignable to parameter of type 'string'.

7 getFirstWord(123)
               ~~~
```

这次的报错代码是在 `getFirstWord(123)` 这里，

告诉 `number` 类型的数据不能分配给 `string` 类型的参数，也就是故意传入一个会报错的数值进去，被 TypeScript 检查出来了！

可以再仔细留意一下控制台的信息，会发现没有报错的 `getFirstWord('Hello World')` 也没有打印出结果，

这是因为 TypeScript 需要先被编译成 JavaScript ，然后再执行。

这个机制让有问题的代码能够被及早发现，一旦代码出现问题，编译阶段就会失败。

移除会报错的那行代码，只保留如下：

```
// src/ts/index.ts
function getFirstWord(msg: string) {
  console.log(msg.split(' ')[0])
}

getFirstWord('Hello World')
```

再次运行 `npm run dev:ts` ，这次完美运行！

```
npm run dev:ts

> demo@1.0.0 dev:ts
> ts-node src/ts/index.ts

Hello
```

编译

安装 TypeScript 后通过内置的命令 tsc 就能将 ts 文件编译为对应的 js 文件

```
tsc index.ts
```

- 上面的方法太麻烦,可以在编译器(这里只说在 vscode 中的配置)中自动监视,

  在项目目录中运行 tsc --init 生成 tsconfig.json

  修改 tsconfig.json 中的 outDir 选项为 "outDir": "./js",

  以后所有的 js 代码都会在这个文件夹中编译

  然后在 vscode 中打开命令面板(Ctrl/Command + Shift + P)选择输入task,

  选择 Run Task ，监听 tsconfig.json，就可以 vscode 中实时监控 TypeScript 编译
- 在工程化项目中可以使用 webpack 等打包工具来进行编译。

如何编译为 JavaScript 代码

前面一直是基于 `dev:ts` 命令，它调用的是 `ts-node` 来运行的 TS 文件：

```
{
  // ...
  "scripts": {
    // ...
    "dev:ts": "ts-node src/ts/index.ts"
  }
  // ...
}
```

但最终可能需要的是一个 JS 文件，比如要通过 `<script src>` 来放到 HTML 页面里，这就涉及到对 TypeScript 的编译。

来看看如何把一个 TS 文件编译成 JS 文件，让其从 TypeScript 变成 JavaScript 代码。

编译单个文件

先在 package.json 里增加一个 build script：

```
{
  "name": "hello-node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev:cjs": "node src/cjs/index.cjs",
```

```
    "dev:esm": "node src/esm/index.mjs",
    "dev:ts": "ts-node src/ts/index.ts",
    "build": "tsc src/ts/index.ts --outDir dist",
    "compile": "babel src/babel --out-dir compiled",
    "serve": "node server/index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "md5": "^2.3.0"
  },
  "devDependencies": {
    "@types/md5": "^2.3.2",
    "ts-node": "^10.7.0",
    "typescript": "^4.6.3"
  }
}
```

这样在命令行运行 `npm run build` 的时候，

就会把 `src/ts/index.ts` 这个 TS 文件编译，并输出到项目下与 src 文件夹同级的 dist 目录下。

其中 `tsc` 是 TypeScript 用来编译文件的命令， `--outDir` 是它的一个选项，用来指定输出目录，

如果不指定，则默认生成到源文件所在的目录下面。

把下面函数的重载用过的这个例子放到 `src/ts/index.ts` 文件里，

因为它是一段比较典型的、包含了多个知识点的 TypeScript 代码：

```
// 对单人或者多人打招呼
function greet(name: string): string
function greet(name: string[]): string[]
function greet(name: string | string[]) {
  if (Array.isArray(name)) {
    return name.map((n) => `Welcome, ${n}!`)
  }
  return `Welcome, ${name}!`
}

// 单个问候语
const greeting = greet('Petter')
console.log(greeting)

// 多个问候语
const greetings = greet(['Petter', 'Tom', 'Jimmy'])
console.log(greetings)
```

可以先执行 `npm run dev:ts` 测试它的可运行性，

当然，如果期间的代码运行有问题，在编译阶段也会给报错。

现在来编译它，现在在命令行输入 `npm run build` 并回车执行。

可以看到多了一个 dist 文件夹，里面多了一个 `index.js` 文件。

```
hello-node
| # 构建产物
├─dist
| | # 编译后的 JS 文件
| └─index.js
| # 源码文件夹
├─src
| # 锁定安装依赖的版本号
├─package-lock.json
| # 项目清单
└─package.json
```

`index.js` 文件里面的代码如下：

```
function greet(name) {
  if (Array.isArray(name)) {
    return name.map(function (n) {
      return 'Welcome, '.concat(n, '!')
    })
  }
  return 'Welcome, '.concat(name, '!')
}

// 单个问候语
var greeting = greet('Petter')
console.log(greeting)

// 多个问候语
var greetings = greet(['Petter', 'Tom', 'Jimmy'])
console.log(greetings)
```

可以看到已经成功把 TypeScript 代码编译成 JavaScript 代码了。

在命令行执行 `node dist/index.js` ，像之前测试 JS 文件一样使用 `node` 命令，运行 dist 目录下的 `index.js` 文件，它可以正确运行：

```
node dist/index.js
Welcome, Petter!
[ 'Welcome, Petter!', 'Welcome, Tom!', 'Welcome, Jimmy!' ]
```

编译多个模块

刚才只是编译一个 `index.ts` 文件，

如果 `index.ts` 里引入了其他模块，此时 `index.ts` 是作为入口文件，入口文件 `import` 进来使用的模块也会被 TypeScript 一并编译。

拆分一下模块，把 `greet` 函数单独抽离成一个模块文件 `src/ts/greet.ts` ：

```
// src/ts/greet.ts
function greet(name: string): string
function greet(name: string[]): string[]
function greet(name: string | string[]) {
  if (Array.isArray(name)) {
    return name.map((n) => `Welcome, ${n}!`)
  }
  return `Welcome, ${name}!`
}

export default greet
```

在 `src/ts/index.ts` 这边，把这个模块导进来：

```
// src/ts/index.ts
import greet from './greet'

// 单个问候语
const greeting = greet('Petter')
console.log(greeting)

// 多个问候语
const greetings = greet(['Petter', 'Tom', 'Jimmy'])
console.log(greetings)
```

package.json scripts 的 build无需修改，依然只编译 `index.ts` ，但因为导入了 `greet.ts` ，所以 TypeScript 也会一并编译，

试一下运行 `npm run build` ， 现在 dist 目录下就有两个文件了：

```
hello-node
| # 构建产物
├─dist
| ├─greet.js  # 多了这个文件
| └─index.js
|
| # 其他文件这里省略...
└─package.json
```

来看看这一次的编译结果：

先看看 `greet.js` ：

```
// dist/greet.js
'use strict'
exports.__esModule = true
function greet(name) {
  if (Array.isArray(name)) {
    return name.map(function (n) {
      return 'Welcome, '.concat(n, '!')
    })
  }
  return 'Welcome, '.concat(name, '!')
}
exports['default'] = greet
```

再看看 `index.js`：

```
// dist/index.js
'use strict'
exports.__esModule = true
var greet_1 = require('./greet')
// 单个问候语
var greeting = (0, greet_1['default'])('Petter')
console.log(greeting)
// 多个问候语
var greetings = (0, greet_1['default'])(['Petter', 'Tom', 'Jimmy'])
console.log(greetings)
```

这个代码风格就是 CommonJS 模块代码。

其实在编译单个文件代码的时候，它也是 CommonJS，只不过因为只有一个文件，没有涉及到模块化，所以第一眼看不出来。

还是在命令行执行 `node dist/index.js`，

虽然也是运行 dist 目录下的 `index.js` 文件，但这次它的作用是充当一个入口文件了，引用到的 `greet.js` 模块文件也会被调用。

这次一样可以得到正确的结果：

```
node dist/index.js
Welcome, Petter!
[ 'Welcome, Petter!', 'Welcome, Tom!', 'Welcome, Jimmy!' ]
```

修改编译后的 JS 版本

还可以修改编译配置，让 TypeScript 编译成不同的 JavaScript 版本。

修改 package.json 里的 build script，在原有的命令后面增加一个 `--target` 选项：

```
{
  // ...
  "scripts": {
    // ...
    "build": "tsc src/ts/index.ts --outDir dist --target es6"
  }
  // ...
}
```

`--target` 选项的作用是控制编译后的 JavaScript 版本，

可选的值目前有：

`es3` ， `es5` ， `es6` ， `es2015` ， `es2016` ， `es2017` ， `es2018` ， `es2019` ， `es2020` ， `es2021` ， `es2022` ， `esnext` ，

分别对应不同的 JS 规范（所以未来的可选值会根据 JS 规范一起增加）。

之前编译出来的 JavaScript 是 CommonJS 规范，本次配置的是 `es6` ，这是支持 ES Module 规范的版本。

> 通常还需要配置一个 `--module` 选项，用于决定编译后是 CJS 规范还是 ESM 规范，
>
> 但如果没有，默认会根据 `--target` 来决定。

再次在命令行运行 `npm run build` ，这次看看变成了什么：

先看看 `greet.js` ：

```
// dist/greet.js
function greet(name) {
  if (Array.isArray(name)) {
    return name.map((n) => `Welcome, ${n}!`)
  }
  return `Welcome, ${name}!`
}
export default greet
```

再看看 `index.js` ：

```
// dist/index.js
import greet from './greet'
// 单个问候语
const greeting = greet('Petter')
console.log(greeting)
// 多个问候语
const greetings = greet(['Petter', 'Tom', 'Jimmy'])
console.log(greetings)
```

这次编译出来的都是基于 ES6 的 JavaScript 代码，因为涉及到 ESM 模块，所以不能直接在 node 运行它了，

但是可以手动改一下扩展名，改成 `.mjs` （包括 index 文件里 `import` 的 greet 文件名也要改），

然后再运行 `node dist/index.mjs` 。

其他事项

在尝试编译单个文和编译多个模块的时候，相信各位开发者应该没有太大的疑问，

但是来到修改编译后的 JS 版本 这里，事情就开始变得复杂了起来，应该能感觉到编译的选项和测试成本都相应的增加了很多。

事实上刚才编译的 JS 文件，

因为涉及到 ESM 模块化，是无法通过普通的 `<script />` 标签在 HTML 页面里使用的（单个文件可以，因为没有涉及模块），

不仅需要加上 ESM 模块所需的 `<script type="module" />` 属性，本地开发还需要启动本地服务器通过 HTTP 协议访问页面，

才允许在浏览器里使用 ESM 模块。

因此在实际的项目开发中，需要借助构建工具来处理很多编译过程中的兼容性问题，降低开发成本。

而刚才用到的诸如 `--target` 这样的选项，可以用一个更简单的方式来管理，

类似于 package.json 项目清单， TypeScript 也有一份适用于项目的配置清单 tsconfig.json 。

初识 TS

在 TypeScript 中，我们使用 **: 指定变量的类型，: 的前后有没有空格都可以。**

TypeScript 只会在编译时对类型进行静态检查，如果发现有错误，编译的时候就会报错。

而在运行时，与普通的 JavaScript 文件一样，TypeScript 不会对类型进行检查。

如果我们需要保证运行时的参数类型，还是得手动对类型进行判断，例如用 typeof：

```
function sayHello(person: string) {
    if (typeof person === 'string') {
        return 'Hello, ' + person;
    } else {
        throw new Error('person is not a string');
    }
}

let user = 'Tom';
console.log(sayHello(user));
```

如果要在报错的时候终止 js 文件的生成，可以在 tsconfig.json 中配置 noEmitOnError 即可。

TypeScript 类型概览

1、原始类型

string、number、 boolean、null、undefined、void、symbol、bigint

2、顶级类型

any、unknown

3、底部类型

never

4、非原始类型

object

普通对象、数组、元组、枚举通通都是 object 类型

原始类型

TypeScript 并不使用“在左边进行类型声明”的形式，比如 `int x = 0`；

布尔值

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

直接调用 Boolean 也可以返回一个 boolean 类型：

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

使用构造函数 Boolean 创造的对象不是布尔值：

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

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

事实上 new Boolean() 返回的是一个 Boolean 对象：

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

数值

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

编译结果：

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

其中 0b1010 和 0o744 是 ES6 中的二进制和八进制表示法，它们会被编译为十进制数字。

字符串

使用 string 定义字符串类型：

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

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

编译结果：

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

其中 ` 用来定义 ES6 中的模板字符串，${expr} 用来在模板字符串中嵌入表达式。（这与 JSX 类似）

void

JavaScript 没有空值（void）的概念，

在 TypeScript 中，可以用 **void 表示没有任何返回值的函数**：

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

声明一个 void 类型的变量没有什么用，因为你只能将它赋值为 undefined 和 null：

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

```
function run(): void {
  console.log(123)
  // return 不能有返回值,否则报错
}

function run1(): number {
  console.log(456)
  return 123 // 必须有返回值,并且返回值为number类型,否则报错
}

function run2(): any {
  console.log(789) // 因为any是任意类型,所以也可以不要返回值
}
```

null 和 undefined

与 void 的区别是，**null 和 undefined 是所有类型的子类型。**

也就是说 undefined 类型的变量，可以赋值给 number 以及其它类型的变量（当 strictNullChecks 为 false 时）：

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

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

而 void 类型的变量不能赋值给 number 类型的变量：

```
let u: void;
let num: number = u;
// Type 'void' is not assignable to type 'number'.
```

当 strictNullChecks为 true 时，`null`和`undefined`只能赋值给`void`和它们自己。

如果不赋值的话默认变量还是 undefined 类型，

如果没有指定 undefined 直接使用该变量的话会报错,只有自己显式指定为 undefined 类型才不会报错

```
let flag1: undefined
/*
    也可以 let flag1:undefined = undefined;
*/
console.log(flag) // undefined

let flag2: null = null // 如果不指定值为null那么打印的时候也会报错
console.log(flag2) // null
```

`strictNullChecks` 关闭

当 strictNullChecks选项关闭的时候，如果一个值可能是 `null` 或者 `undefined`，

它依然可以被正确的访问，或者被赋值给任意类型的属性。这有点类似于没有空值检查的语言 (比如 C# ，Java) 。

这些检查的缺少，是导致 bug 的主要源头，所以我们始终推荐开发者开启 strictNullChecks选项。

`strictNullChecks` 打开

当 strictNullChecks选项打开的时候，如果一个值可能是 `null` 或者 `undefined`，

你需要在用它的方法或者属性之前，需要先检查这些值，就像用可选的属性之前，先检查一下 是否是 `undefined` ，

我们也可以使用类型收窄（narrowing）检查值是否是 `null`：

```
function doSomething(x: string | null) {
  if (x === null) {
    // do nothing
  } else {
    console.log("Hello, " + x.toUpperCase());
  }
}
```

具体用法

与联合类型一起使用为变量指定多种可能的类型

```
let flag: number | undefined // 这种写法就不会在没有赋值的时候报错了，因为设置了可能为undefined
console.log(flag) // undefined
flag = 123
console.log(flag) // 123 也可以改为数值类型

// 也可以设定多个类型
let flag1: number | string | null | undefined
flag1 = 123
console.log(flag1) // 123
flag1 = null
console.log(flag1) // null
flag1 = 'string'
console.log(flag1) // string
```

Symbol 类型

```
const sym = Symbol();

let obj = {
  [sym]: "Tom",
};

console.log(obj[sym]); // Tom
```

顶级类型

any

任意值（any）表示允许赋值为任意类型。

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

如果是一个普通类型，在赋值过程中改变类型是不被允许的：

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

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

但如果是 any 类型，则允许被赋值为任意类型。

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

如果你没有指定一个类型，TypeScript 也不能从上下文推断出它的类型，编译器就会默认设置为 any 类型。

TypeScript 默认对 any 不做类型检查。

开启编译项 noImplicitAny ，当隐式推断为 any 时，TypeScript 就会报错。

任意值的属性和方法

在任意值上访问任何属性都是允许的：

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

也允许调用任何方法：

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

未声明类型的变量

变量如果在声明的时候，未指定其类型，那么它会被识别为任意值（any）类型：

```
let something;
something = 'seven';
something = 7;
something.setName('Tom');
```

等价于

```
let something: any;
something = 'seven';
something = 7;
something.setName('Tom');
```

定义类型为 any 时，

这个变量你可以随便用，调用不存在的方法 ts 也不会报错，但是你运行就会报错，

因为你没有在 anyThing 上挂载 setName 方法，你可以这么写，但是运行的是 js 。

具体用法

```
// 如果在 ts 中想要获取 DOM 节点,就需要使用任意类型
let oDiv: any = document.getElementsByTagName('div')[0]
oDiv.style.backgroundColor = 'red'
// 按道理说 DOM 节点应该是个对象,但是 TypeScript 中没有对象的基本类型，所以必须使用 any 类型才不会报错
```

unknown

TypeScript 3.0 引入了新的`unknown`类型，它是`any`类型对应的安全类型。

就像所有类型都可以被归为`any`，**所有类型也都可以被归为`unknown`。**

这使得`unknown`成为 TypeScript 类型系统的另一种顶级类型（另一种是`any`）。

unknown 与 any 的区别

`unknown`和`any`的主要区别是`unknown`类型会更加严格：

在对`unknown`类型的值执行大多数操作之前，必须进行某种形式的检查。

而在对`any`类型的值执行操作之前，不必进行任何检查。

unknown 类型变量的值可以被赋值为任意其它类型

```
let value: unknown
value = true // OK
value = 42 // OK
value = 'Hello World' // OK
value = [] // OK
value = {} // OK
value = Math.random // OK
value = null // OK
value = undefined // OK
value = new TypeError() // OK
value = Symbol('type') // OK
```

unknown 类型变量只能赋值给 any 类型变量和 unknown 类型变量本身。

这是有道理的：只有能够保存任意类型值的容器才能保存 unknown 类型的值。

毕竟我们不知道变量 value 中存储了什么类型的值。

```
let value: unknown
let value1: unknown = value // OK
let value2: any = value // OK
let value3: boolean = value // Error
let value4: number = value // Error
let value5: string = value // Error
let value6: object = value // Error
let value7: any[] = value // Error
let value8: Function = value // Error
```

unknown 类型不能直接进行操作，这些操作都不再像 any 一样被认为是类型正确的：

```
let value: unknown

value.foo.bar // Error
value.trim() // Error
value() // Error
new value() // Error
value[0][1] // Error
```

只能对 unknown 类型进行等于、不等于的运算符操作，不能进行其他的操作

```
let value: unknown
let value2: number = 1
console.log(value === value2)
console.log(value !== value2)
```

底部类型（never）

never 类型是其它类型（包括 null 和 undefine）的子类型，表示不会出现值（永远不会存在值的类型）。

声明 never 类型的变量只能被 never 类型所赋值。

例如，

never 类型可以是那些总是会抛出异常或根本不会有返回值的函数表达式或箭头函数表达式的返回值类型。

```
let a: undefined
a = undefined

let b: null
b = null

// 空数组，而且要求数组永远是空的
const empty: never[] = []

let c: never // c 不能被任何值赋值，包括 null 和 undefined ，never 指的是不会出现值
c = (() => {
  throw new Error('error!!')
})() // 可以这样赋值，因为箭头函数表达式不会返回值


// 比如在一个函数中 throw 一个 Error，或者这个函数内部有个死循环，那么这种永远不会返回值的函数，返回类型就是 never
// 抛出异常的函数永远不会有返回值，返回 never 的函数必须存在无法达到的终点
function error(message: string): never {
  throw new Error(message)
}

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

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

在 TypeScript 中，可以利用 never 类型的特性来实现全面性检查，具体示例如下：

```
type Foo = string | number;

function controlFlowAnalysisWithNever(foo: Foo) {
  if (typeof foo === "string") {
    // 这里 foo 被收窄为 string 类型
  } else if (typeof foo === "number") {
    // 这里 foo 被收窄为 number 类型
  } else {
    // foo 在这里是 never
    const check: never = foo;
  }
}
```

注意在 else 分支里面，我们把收窄为 never 的 foo 赋值给一个显示声明的 never 变量。

如果一切逻辑正确，那么这里应该能够编译通过。

但是假如后来有一天你的同事修改了 Foo 的类型：

```
type Foo = string | number | boolean;
```

然而他忘记同时修改 controlFlowAnalysisWithNever 方法中的控制流程，

这时候 else 分支的 foo 类型会被收窄为 boolean 类型，导致无法赋值给 never 类型，

这时就会产生一个编译错误。

通过这个方式，

使用 never 可以避免新增联合类型却没有对应实现的错误，目的就是写出类型绝对安全的代码。

非原始类型

对象的类型——接口

接口是命名对象类型的另一种方式

接口用于类型的静态检查，ts 编译为 js 时并不会转换

object 类型与缺点

当我们希望一个变量或者函数的参数的类型是一个对象的时候，使用这个类型，比如：

```
let obj: object
obj = { name: 'Jack' }
obj = 123 // error 不能将类型“123”分配给类型“object”
```

这里有一点要注意了，

你可能会想给 obj 指定类型为 object 对象类型，然后给它赋值一个对象，

后面通过属性访问操作符访问这个对象的某个属性，实际操作一下你就会发现会报错：

```
let obj: object
obj = { name: 'Jack' }
console.log(obj.name) // error 类型“object”上不存在属性“name”
```

这里报错说类型 object 上没有 name 这个属性。

如果想要达到这种需求，应该使用后面要讲到的接口。

那 object 类型适合什么时候使用呢?

我们前面说了，当你希望一个值必须是对象而不是数值等类型时，

比如我们定义一个函数，参数必须是对象，这个时候就用到 object 类型了：

```
function getKeys (obj: object) {
    return Object.keys(obj) // 会以列表的形式返回 obj 中的值
}

getKeys({ a: 'a' }) // ['a']
getKeys(123) // error 类型“123”的参数不能赋给类型“object”的参数
```

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

什么是接口

在面向对象语言中，接口（Interfaces）是一个很重要的概念，

它是对行为的抽象（描述、轮廓、雏形），而具体如何行动需要由类（classes）去实现（implement）。

TypeScript 中的接口是一个非常灵活的概念，除了可用于对类的一部分行为进行抽象以外，

也常用于对「对象的形状（Shape）」进行描述，表示对「对象的外形」的要求，限定传入的对象的结构。

接口方便对象类型的复用

```
interface Types {
  name: string,
  age: number
}

const testObj: Types = { name: "Hello", age: 18 }

const testObj1: Types = { name: "World", age: 18 }
```

> 对象的类型定义通常采用 Upper Camel Case 大驼峰命名法（也叫帕斯卡命名法），也就是每个单词的首字母大写，例如 UserItem 、 GameDetail
>
> 这是为了跟普通变量进行区分（变量通常使用 Lower Camel Case 小驼峰写法，也就是第一个单词的首字母小写，其他首字母大写，例如 userItem ）。

简单的例子

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

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

上面的例子中，

我们定义了一个接口 Person，接着定义了一个变量 tom，它的类型是 Person。

这样，我们就约束了 tom 的形状必须和接口 Person 一致。

接口一般首字母大写，可以用 I 字母开头。

注意在定义接口的时候，不要把它理解为是在定义一个对象

（虽然通常用于声明对象字面量的类型，但是它还可以声明函数），

所以要理解为 {} 括号包裹的是一个代码块，里面是一条条声明语句，

只不过声明的不是变量的值而是变量的类型。

声明也不用等号赋值，而是冒号指定类型。

每条声明之前用换行分隔即可，或者也可以使用分号或者逗号，都是可以的。

```
interface List {
  id: number;
  name: string;
}

interface Result {
  data: List[]
}

let result = {
  data: [
    {id: 1, name: 'A'},
    {id: 2, name: 'B'},
  ]
}

function render(result: Result) {
  result.data.forEach((value) => {
    console.log(value.id, value.name);
  })
}

render(result);
// 1 A
// 2 B
```

额外属性检查（属性数量检查）

定义的变量比接口少了一些属性是不允许的：

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

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

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

多一些属性也是不允许的：

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

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

// index.ts(9,5): error TS2322: Type '{ name: string; age: number; gender:
string; }' is not assignable to type 'Person'.
//   Object literal may only specify known properties, and 'gender' does not
exist in type 'Person'.
```

可见，赋值的时候，变量的形状（属性）必须和接口的形状（属性）保持一致。

函数参数的不同

对于函数类型的类型检查来说，

函数的参数名不需要与接口里定义的名字相匹配（不需要和接口的形状保持一致）。

```
interface LabelledValue {
  label: string;
}

function printLabel(labelledObj: LabelledValue) {
  console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

在外面将该对象用另一个变量 myObj 接收，myObj 不会经过额外属性检查，

ts 会将 myObj 根据类型推断为 `let myObj: { size: number; label: string } = { size: 10, label: "Size 10 Object" };`

然后将这个 myObj 再赋值给 labeledObj，

此时根据类型的兼容性，两种对象类型，参照鸭式辨型法，因为都具有 label 属性，

所以被认定为两个对象类型相同，故而可以用此法来绕开多余的类型检查。

需要注意的是，

我们传入的对象参数实际上会包含很多属性，

但是编译器只会检查那些必需的属性是否存在，并且其类型是否匹配。

类型检查器不会去检查属性的顺序，只要相应的属性存在并且类型是对的就可以。

额外属性检查的解决办法

假设我们有一个 Config 接口如下

```
interface Config {
  width?: number;
}

function CalculateAreas(config: Config): { area: number } {
  let square = 100;
  if (config.width) {
    square = config.width * config.width;
  }
  return {area: square};
}

let mySquare = CalculateAreas({ widdth: 5 });
```

注意我们传入的参数是 widdth，并不是 width。

此时 TypeScript 会认为这段代码可能存在问题。

对象字面量当被赋值给变量或作为参数传递的时候，会被特殊对待，并且会经过"额外属性检查"。

如果一个对象字面量存在任何"目标类型"不包含的属性时，你会得到一个错误。

```
// error: 'widdth' not expected in type 'Config'
let mySquare = CalculateAreas({ widdth: 5 });
```

目前官网推荐了三种主流的解决办法：

第一种使用类型断言：

```
let mySquare = CalculateAreas({ widdth: 5 } as Config);
```

第二种添加字符串索引签名：

```
interface Config {
   width?: number;
   [propName: string]: any;
}
```

这样， Config 可以有任意数量的属性，只要不是 width，无所谓它们的类型是什么。

第三种将字面量赋值给另外一个变量：

```
let options: any = { widdth: 5 };
let mySquare = CalculateAreas(options);
```

本质上是转化为 any 类型，除非有万不得已的情况，不建议采用这种方法。

可选属性

当我们定义一些结构的时候，

一些结构对于某些字段的要求是可选的，有这个字段就做处理，没有就忽略

我们希望不要完全匹配一个形状，那么可以用可选属性：

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

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

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

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

可选属性的含义是该属性可以不存在。

但是仍然不允许添加未定义的属性：

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

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

// examples/playground/index.ts(9,5): error TS2322: Type '{ name: string; age: number; gender: string; }' is not assignable to type 'Person'.
//   Object literal may only specify known properties, and 'gender' does not exist in type 'Person'.
```

任意（额外、多余）属性

有时候我们希望一个接口允许有任意的属性，可以使用如下方式：

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

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

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

只要 propName 属性是 string 类型就行。

[propName: string] 表示 Person 可以有任意数量的属性，

并且只要它们不是 name 和 age ， 无所谓它们的类型是什么

```
interface RandomMap {
  [propName: string]: string;
}
const test: RandomMap = {
  a: 'hello',
  b: 'test',
  c: 'test'
}
interface LikeArray {
  [index: number]: string
}

const likeArray: LikeArray = ['1', '2', '3']
```

需要注意的是，一旦定义了任意属性，那么

确定属性值的类型和可选属性值的类型都必须是任意属性值的类型的子集：

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

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

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

上例中，任意属性的值允许是 string，

但是可选属性 age 的值却是 number，number 不是 string 的子属性，所以报错了。

另外，在报错信息中可以看出，

此时 { name: 'Tom', age: 25, gender: 'male' } 的类型被推断成了

{ [x: string]: string | number; name: string; age: number; gender: string; }，

这是联合类型和接口的结合。

一个接口中只能定义一个任意属性。

如果接口中该任意属性有多个类型的属性，则可以在任意属性中使用联合类型：

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

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

只读属性

有时候我们希望**对象中的一些字段只能在创建的时候被赋值，之后无法修改**

那么可以用 readonly 定义只读属性：

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

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

tom.id = 9527;

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

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

注意，只读的约束最先存在于第一次给对象赋值的时候，而不只是第一次给只读属性赋值的时候：

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

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

tom.id = 89757;
```

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

上例中，报错信息有两处，

第一处是在对 tom 进行赋值的时候，没有给 id 赋值。

第二处是在给 tom.id 赋值的时候，由于它是只读属性，所以报错了。

调用自身接口的属性

如果一些属性的结构跟本身一致，也可以直接引用，

比如下面例子里的 `friendList` 属性，用户的好友列表，它就可以继续使用 `UserItem` 这个接口作为数组的类型：

```
interface UserItem {
  name: string
  age: number
  enjoyFoods: string[]
  // 这个属性引用了本身的类型
  friendList: UserItem[]
}

const petter: UserItem = {
  name: 'Petter',
  age: 18,
  enjoyFoods: ['rice', 'noodle', 'pizza'],
  friendList: [
    {
      name: 'Marry',
      age: 16,
      enjoyFoods: ['pizza', 'ice cream'],
      friendList: [],
    },
    {
      name: 'Tom',
      age: 20,
      enjoyFoods: ['chicken', 'cake'],
      friendList: [],
    }
  ],
}
```

接口扩展（继承）

接口扩展与类的继承类似，可以用子接口扩展父接口，从而拿到多个接口的限制条件

```
interface Types {
  readonly name: string,
  readonly age: number,
  sex?: string
}
```

```ts
interface ChildrenType extends Types { // ChildrenType 接口继承了父级 Types 接口
  hobby: []
}

const testObj: ChildrenType = {
  name: "小明",
  age: 18,
  hobby: ["篮球", "羽毛球"]
}
```

```ts
interface Animal {
  eat(): void
}

interface Person extends Animal {
  // 继承父接口 Animal 的限制条件
  name: string
  work(): void
}

class Student implements Person {
  // 类实现接口，会同时将前面两者的接口限制合并，Student 类必须实现这些限制
  constructor(public name: string) {}
  eat() {
    console.log(this.name + '吃饭')
  }
  work() {
    console.log(this.name + '上学')
  }
}

let stu: Student = new Student('小明')
stu.eat()
stu.work()
```

```ts
// 接口和继承相结合
interface Animal {
  eat(): void
}

interface Plant {
  wait(): void
}

// 一个接口也可以继承多个接口,用逗号隔开
interface Person extends Animal, Plant {
  name: string
  work(): void
}

class YoungPerson {
```

```
  constructor(public name: string) {}
  drink() {
    console.log(this.name + '喝水')
  }
}

// 类继承并实现接口
class Student extends YoungPerson implements Person {
  constructor(name: string) {
    super(name)
  }
  eat() {
    console.log(this.name + '吃饭')
  }
  work() {
    console.log(this.name + '上学')
  }
  wait() {
    console.log(this.name + '停下')
  }
}

let stu: Student = new Student('小明')
stu.eat()
stu.drink()
stu.work()
stu.wait()
```

比如要对用户设置管理员，管理员信息也是一个对象，但要比普通用户多一个权限级别的属性，

那么就可以使用继承，通过 `extends` 来实现：

```
interface UserItem {
  name: string
  age: number
  enjoyFoods: string[]
  friendList: UserItem[]
}

// 这里继承了 UserItem 的所有属性类型，并追加了一个权限等级属性
interface Admin extends UserItem {
  permissionLevel: number
}

const admin: Admin = {
  name: 'Petter',
  age: 18,
  enjoyFoods: ['rice', 'noodle', 'pizza'],
  friendList: [
    {
      name: 'Marry',
      age: 16,
      enjoyFoods: ['pizza', 'ice cream'],
      friendList: [],
    },
```

```
    {
      name: 'Tom',
      age: 20,
      enjoyFoods: ['chicken', 'cake'],
      friendList: [],
    }
  ],
  permissionLevel: 1,
}
```

如果觉得这个 `Admin` 类型不需要记录这么多属性，也可以在继承的过程中舍弃某些属性，

通过 `Omit` 帮助类型来实现，`Omit` 的类型如下：

```
type Omit<T, K extends string | number | symbol>
```

其中 `T` 代表已有的一个对象类型，`K` 代表要删除的属性名，

如果只有一个属性就直接是一个字符串，如果有多个属性，用 `|` 来分隔开，下面的例子就是删除了两个不需要的属性：

```
interface UserItem {
  name: string
  age: number
  enjoyFoods: string[]
  friendList?: UserItem[]
}

// 这里在继承 UserItem 类型的时候，删除了两个多余的属性
interface Admin extends Omit<UserItem, 'enjoyFoods' | 'friendList'> {
  permissionLevel: number
}

// 现在的 admin 就非常精简了
const admin: Admin = {
  name: 'Petter',
  age: 18,
  permissionLevel: 1,
}
```

类型别名

有的时候，一个类型会被使用多次，此时我们希望通过一个单独的名字来引用它。

类型别名用来给一个类型起个新名字。

简单的例子

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;

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

上例中，我们使用 type 创建类型别名。

类型别名常用于联合类型。

接口与类型别名

接口与类型别名可以认为是同一概念的两种语法

类型别名更为通用，其右侧可以包含类型表达式（联合类型、交叉类型、条件类型），

但接口右侧只能是某种结构（{...}）

不同点

扩展语法： interface 使用 extends，type 使用'&'

同名合并：interface 支持，type 不支持。

描述类型：对象、函数两者都适用，但是 type 可以用于基础类型、联合类型、元组。

计算属性：type 支持计算属性，生成映射类型，interface 不支持。

相同点

两者都可以用来描述对象或函数的类型。

两者都可以实现继承。

两者使用场景（重点）

1. 基本类型：用 `type` ，因为 `interface` 做不到

   ```
   type Nullish = null | undefined;

   type Fruit = 'apple' | 'pear' | 'orange';

   type Num = number | bigint;
   ```

2. 元组：用 `type`，因为 `interface` 也做不到

   ```
   type row = [colOne: number, colTwo: string];
   ```

3. 函数：用 `type` ，因为可读性

```
// via type
type Sum = (x: number, y: number) => number;

// via interface
interface Sum {  (x: number, y: number): number;}
```

4. UnionType 联合类型：用 `type`，因为 `interface` 没法 union

```
type Fruit = 'apple' | 'pear' | 'orange';

type Vegetable = 'broccoli' | 'carrot' | 'lettuce';

// 'apple' | 'pear' | 'orange' | 'broccoli' | 'carrot' | 'lettuce';

type HealthyFoods = Fruit | Vegetable;
```

5. MappedType 映射类型：用 `type`，因为 `interface` 做不到：

```
type Fruit = 'apple' | 'orange' | 'banana';

type FruitCount = {
  [key in Fruit]: number;
}

const fruits: FruitCount = {
  apple: 2,
  orange: 3,
  banana: 4
}
```

除此以外，通常都用 interface

https://www.codesky.me/archives/typescript-difference-type-interface.wind

1.对象/函数

接口和类型别名都可以用来描述对象的形状或函数签名：

接口

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

// 函数
interface SetPoint {
  (x: number, y: number): void;
}
```

类型别名

```
// 对象
type Point = {
  x: number;
  y: number;
};

// 函数
type SetPoint = (x: number, y: number) => void;
```

2.其它类型

与接口类型不一样，类型别名可以用于一些其他类型，比如原始类型、联合类型和元组：

```
// primitive
type Name = string;

// object
type PartialPointX = { x: number; };
type PartialPointY = { y: number; };

// union
type PartialPoint = PartialPointX | PartialPointY;

// tuple
type Data = [number, string];
```

3.扩展

接口和类型别名都能够被扩展，但语法有所不同。

此外，接口和类型别名不是互斥的。

接口可以扩展类型别名，而反过来是不行的。

接口间继承 (extends) 时 TS 会检查二者关系，

但类型联合时 TS 会尽最大努力尝试，不会报错。

接口扩展接口

extends 关键字允许我们从其他命名的类型中复制成员，并添加任何新成员，减少重复的类型声明

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

```
interface Colorful {
  color: string;
}

interface Circle {
  radius: number;
```

```
}

interface ColorfulCircle extends Colorful, Circle {}

const cc: ColorfulCircle = {
  color: "red",
  radius: 42,
};
```

类型别名扩展类型别名

```
type PartialPointX = { x: number; };
type Point = PartialPointX & { y: number; };
```

接口扩展类型别名

```
type PartialPointX = { x: number; };
interface Point extends PartialPointX { y: number; }
```

类型别名扩展接口

```
interface PartialPointX { x: number; }
type Point = PartialPointX & { y: number; };
```

4.实现

类可以以相同的方式实现接口或类型别名，但类不能实现使用类型别名定义的联合类型：

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

class SomePoint implements Point {
  x = 1;
  y = 2;
}

type Point2 = {
  x: number;
  y: number;
};

class SomePoint2 implements Point2 {
  x = 1;
  y = 2;
}

type PartialPoint = { x: number; } | { y: number; };

// A class can only implement an object type or
// intersection of object types with statically known members.
class SomePartialPoint implements PartialPoint { // Error
  x = 1;
```

```
  y = 2;
}
```

5.声明合并

与类型别名不同，接口可以定义多次，会被自动合并为单个接口。

同一个作用域中的多个同名接口声明会被合并，而多个同名类型别名会报错。

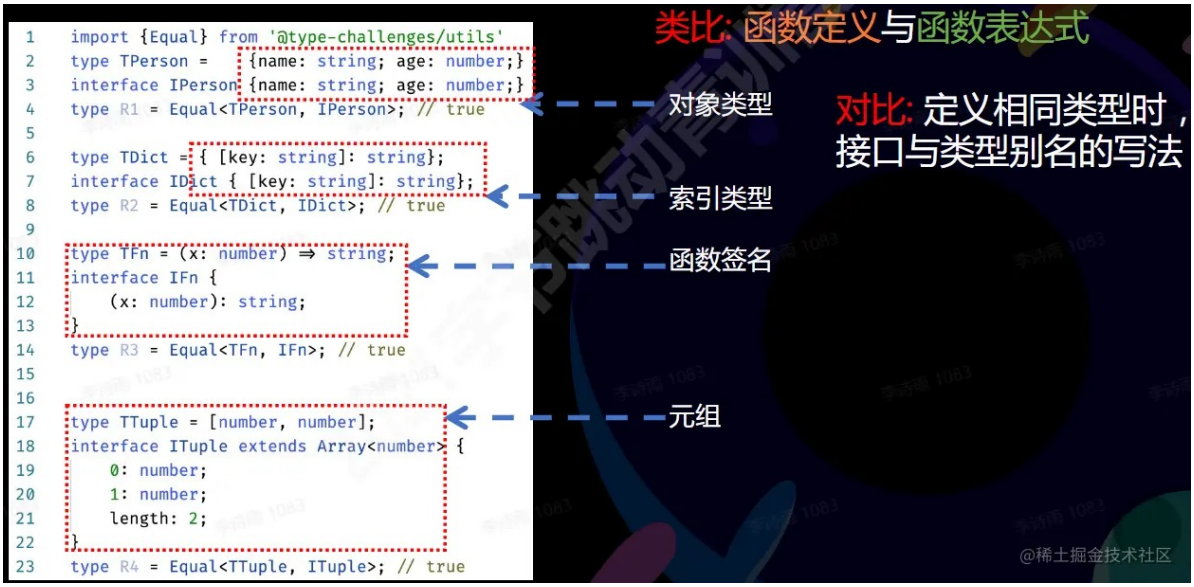
```
interface Point { x: number; }
interface Point { y: number; }

const point: Point = { x: 1, y: 2 };
```

总结

公共的用 interface 实现，不能用 interface 实现的再用 type 实现。

至于在业务中到底用谁，tslint 中的建议是当 interface 和 type 都可使用时优先考虑 interface 。

一个项目最好保持一致。



数组

TypeScript 中，数组类型有多种定义方式，比较灵活。

```
// 第一种定义数组方法
let arr1: number[] = [1, 2, 3]; // 一个全是数字的数组

// 第二种定义数组方法
let arr2: Array<number> = [1, 2, 3];

// 第三种定义数组方法(元组类型)
let arr3: [number, string] = [123,"string"];

// 第四种定义数组方法(任意类型)
let arr4: any[] = [1, "string", true, null];
```

「类型 + 方括号」表示法

最简单的方法是使用「类型 + 方括号」来表示数组：

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

这样声明后，数组的项中**不允许**出现其他的类型：

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

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

数组的一些方法的参数也会根据数组在定义时约定的类型进行限制：

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

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

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

这里 "8" 是一个字符串字面量类型。

> **注意 `[number]` 和 `number[]` 表示不同的意思，即元组和数组的区别**

多种类型的数组

默认表示为联合类型

```
let multipleTypeArray = [1, true, 3];
// 等价于
let multipleTypeArrayExplicit: (number | boolean)[] = [1, true, 3];
```

数组泛型表示数组

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

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

数组里的数据	类型写法 1	类型写法 2
字符串	string[]	Array
数值	number[]	Array
布尔值	boolean[]	Array
大整数	bigint[]	Array
符号	symbol[]	Array
不存在	null[]	Array
未定义	undefined[]	Array

在实际的编程过程中，

如果数组一开始就有初始数据（数组长度不为 0 ），那么 ts 也会根据数组里面的项目类型，自动推断这个数组的类型，

这种情况下也可以省略类型定义：

```
// 这种有初始项目的数组， TS 也会帮推导它们的类型
const strs = ['Hello World', 'Hi World']
const nums = [1, 2, 3]
const bools = [true, true, false]
```

但是，如果一开始是 `[]` ，那么就必须显式的指定数组类型（取决于的 tsconfig.json的配置，可能会引起报错）：

```
// 这个时候会认为是 any[] 或者 never[] 类型
const nums = []

// 这个时候再 push 一个 number 数据进去，也不会使其成为 number[]
nums.push(1)
```

实例化数组

也可以实例化一个强类型的对象数组。这相当于创建一个新的数组而不分配任何值。

```
let myArray = new Array<number>();
printArray(myArray);

// 等价于
let myArray2: Array<number> = [];
printArray(myArray2);

// 等价于
let myArray3: number[] = [];
printArray(myArray3);

function printArray(a: number[]): void {
    console.log(`Before: ${a}`);
    a.push(1);
    console.log(`After: ${a}`);
}
```

any 在数组中的应用

一个比较常见的做法是，用 any 表示数组中允许出现任意类型：

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

用接口表示数组（比较少用）

接口也可以用来描述数组：

```
interface NumberArray {
  [index: number]: number;
}

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

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

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

不过有一种情况例外，那就是**接口用来表示类数组。**

类数组

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

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

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

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

```
function sum() {
  let args: {
      [index: number]: number; // 当索引的类型是数字时，值的类型必须是数字
      length: number;
      callee: Function;
  } = arguments;
}
```

注意，上例也约束了 arguments 有 length 和 callee 两个属性。

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

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

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

只读数组（不可变数组）

前面表述的数组都是可变数组

只读数组中的数组成员和数组本身的 length 等属性都不能够修改,并且也不能赋值给已赋值的数组

```
let a: number[] = [1, 2, 3, 4]
let ro: ReadonlyArray<number> = a // 其实只读数组只是一个内置定义的泛型接口
ro[1] = 5           // 报错
ro.length = 5       // 报错
ro.push(5)          // 报错
a = ro              // 报错,因为 ro 的类型为 Readonly,已经改变了类型
a = ro as number[] // 正确,不能通过上面的方法复制,但是可以通过类型断言的方式,类型断言见下面
```

元组

TS 中，元组类型属于数组的一种，数组合并相同类型，而元组（Tuple）合并不同类型。

元组表示一个已知元素数量和类型的数组。

简单的例子

定义一对值分别为 string 和 number 的元组：

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

当赋值或访问一个已知索引的元素时，会得到正确的类型：

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

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

也可以只赋值其中一项：

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

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

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

let tom: [string, number];
tom = ['Tom'];
// Property '1' is missing in type '[string]' but required in type '[string,
number]'.
```

简而言之，元组赋值时，元素的类型、数量、顺序必须与声明时一致。

```
let x: [string, number];

x = ['hello', 10]; // OK

x = ['hello', 10, false] // Error
x = ['hello'] // Error
x = [10, 'hello']; // Error
```

我们可以把元组看成严格版的数组，比如 [string, number] 我们可以看成是:

```
interface Tuple extends Array<string | number> {
  0: string;
  1: number;
  length: 2;
}
```

越界的元素

当添加越界的元素时，它的类型会被限制为元组中每个类型的联合类型：

```
let tom: [string, number];
tom = ['Tom', 25];
tom.push('male');
tom.push(true);
// Argument of type 'true' is not assignable to parameter of type 'string |
number'.
```

注意:

与数组一样，元组也可以使用`readonly`修饰，`readonly`类型修饰符只能用于数组类型和元组类型

```
let err1: readonly Set<number>    // error!
let err2: readonly Array<boolean> // error!

let okay: readonly boolean[] // works fine
```

解构元组

```
function doSomething(stringHash: [string, number]) {
  const [inputString, hash] = stringHash;
  console.log(inputString);
  // const inputString: string
  console.log(hash);
  // const hash: number
}
```

可选的元组元素

元组可以通过在元素的类型后面写问号，？可选的元组元素只能出现在末尾，而且会影响类型的 length

```
type Either2dOr3d = [number, number, number?];

function setCoordinate(coord: Either2dOr3d) {
  const [x, y, z] = coord;
  // const z: number | undefined

  console.log(`Provided coordinates had ${coord.length} dimensions`);
  // (property) length: 2 | 3
}
```

元组的 rest 元素

元组也可以有 rest 元素，这些元素必须是一个数组/元组类型。

```
type StringNumberBooleans = [string, number, ...boolean[]];
type StringBooleansNumber = [string, ...boolean[], number];
type BooleansStringNumber = [...boolean[], string, number];
```

StringNumberBooleans 描述了一个元组，

前两个元素分别是字符串和数值，但后面可以有任意数量的布尔类型。

StringBooleansNumber 描述一个元组，

第一个元素是字符串，然后是任意数量的布尔类型，最后是一个数值。

BooleansStringNumber 描述了一个元组，

起始元素是任意数量的布尔类型，然后是一个字符串，最后是一个数值。

带有 rest 元素的元组没有设置“length”——它只有一组位于不同位置的已知元素。

```
const a: StringNumberBooleans = ["hello", 1];
const b: StringNumberBooleans = ["beautiful", 2, true];
const c: StringNumberBooleans = ["world", 3, true, false, true, false, true];
```

元组类型可以在 rest 形参和实参中使用，因此：

```
function readButtonInput(...args: [string, number, ...boolean[]]) {
  const [name, version, ...input] = args;
  // ...
}
```

基本等价于

```
function readButtonInput(name: string, version: number, ...input: boolean[]) {
  // ...
}
```

当你希望使用 rest 形参接受可变数量的参数，并且需要最少数量的元素，

但又不想引入中间变量时，这是非常方便的。

枚举

枚举用于描述一个值可能是多个常量中的一个。

当一个变量有几种可能的取值时，可以将它定义为枚举类型。

数值枚举

枚举类型声明一组命名的常数

```
// 定义一个枚举类型
// 枚举变量即枚举类型
enum 枚举变量 {
  标识符 = 值,
  标识符 = 值,
  ......
  标识符 = 值
};
// 获取值
枚举变量.标识符
// 获取标识符
枚举变量.值
```

```
enum Direction {
  Up,
  Down,
  Left,
  Right
}

console.log(Direction.Up === 0); // true
console.log(Direction.Down === 1); // true
console.log(Direction.Left === 2); // true
console.log(Direction.Right === 3); // true

enum Direction {
  Up,
  Down,
  Left,
  Right
}

console.log(Direction[0]); // Up
```

注意：

- **可以把标识符用引号括起来，效果不受影响**
- **如果没有给标识符赋值，那么标识符的值默认为索引值**
- 如果某个标识符进行了赋值，而之后的标识符没有赋值，那么之后的标识符的索引依次为前面的值加 1
- 枚举可以引用内部的标识符的值或者外部变量的值

```
const o = 5

enum Color {
  red = 1,
  blue = red, // 引用内部标识符
  orange = o  // 引用外部变量
}
```

```
enum Direction {
  Up,
  Down,
  Left,
  Right
}

declare let a: Direction

enum Animal {
  Dog,
  Cat
}

a = Direction.Up // ok
a = Animal.Dog // 不能将类型“Animal.Dog”分配给类型“Direction”
```

我们把 a 声明为 Direction 类型，可以看成我们声明了一个联合类型：

Direction.Up | Direction.Down | Direction.Left | Direction.Right，

只有这四个类型其中的成员才符合要求

枚举类型被编译为 JavaScript 后的样子:

```
var Direction;
(function (Direction) {
  Direction[Direction["Up"] = 10] = "Up";
  Direction[Direction["Down"] = 11] = "Down";
  Direction[Direction["Left"] = 12] = "Left";
  Direction[Direction["Right"] = 13] = "Right";
})(Direction || (Direction = {}));
```

默认变量 Direction 为一个空对象，Direction = {}

Direction[Direction["Up"] = 10] = "Up";

先执行里层的赋值 Direction["Up"] = 10 ，

Direction 变为 {"UP": 10}

执行 Direction["Up"] = 10 ， 赋值运算符会返回 10，

所以执行到外层的赋值时是： Direction[10] ="UP" ，

Direction 最终为 {"10": "UP", "Up": 10}

我们可以分开声明枚举,它们会自动合并

```
enum Direction {
  Up = 'Up',
  Down = 'Down',
  Left = 'Left',
  Right = 'Right'
}

enum Direction {
  Center = 1
}
```

编译为 JavaScript 后的代码如下:

```
var Direction;
(function (Direction) {
  Direction["Up"] = "Up";
  Direction["Down"] = "Down";
  Direction["Left"] = "Left";
  Direction["Right"] = "Right";
})(Direction || (Direction = {}));
(function (Direction) {
  Direction[Direction["Center"] = 1] = "Center";
})(Direction || (Direction = {}));
```

字符串枚举

枚举的值除了使用整数外还可以是纯的字符串，主要利用其键值相互获取的特性

```
enum Direction {
  NORTH = "NORTH",
  SOUTH = "SOUTH",
  EAST = "EAST",
  WEST = "WEST",
}
```

```
var Direction;
(function (Direction) {
  Direction[Direction["NORTH"] = 0] = "NORTH";
  Direction[Direction["SOUTH"] = 1] = "SOUTH";
  Direction[Direction["EAST"] = 2] = "EAST";
  Direction[Direction["WEST"] = 3] = "WEST";
})(Direction || (Direction = {}));
var dirName = Direction[0]; // NORTH
var dirVal = Direction["NORTH"]; // 0
console.log("dirName", dirName);
console.log("dirVal", dirVal);
```

```
enum Direction {
  NORTH,
  SOUTH,
  EAST,
  WEST,
}

let dirName = Direction[0]; // NORTH
let dirVal = Direction["NORTH"]; // 0
```

异构枚举

简单来说就是：既包含数字又包含字符串的枚举类型

```
// 异构枚举也可以使用内部的变量，但不能使用外部的变量
enum Message {
  Error = 0,
  Success = 'success',
  Failed = 'failed'
}
```

常量枚举

在正常情况下，枚举不像是 type 定义的类型这样编译后是不会存在的，

枚举类型创建后就是默认开辟了一个枚举变量的空间，并且枚举值我们一般用作提高代码的可读性：

```
enum Status {
  Success: 200,
}

console.log(res.status === Status.Success) // 通常我们会这样为响应对象提高代码可读性
```

如果不想要枚举变量真实存在，只是想自己创建一个别值，那么可以在枚举变量前加上 const 关键字

```
// 加了 const 关键字以后相当于原来的枚举变量只是个占位符
const enum Status {
  Success: 200,
}
```

使用 const 关键字修饰的枚举，常量枚举会使用内联语法，不会为枚举类型编译生成任何 JavaScript。

为了理解这句话，我们来看一个具体的例子：

```
const enum Direction {
  NORTH,
  SOUTH,
  EAST,
  WEST,
}

let dir: Direction = Direction.NORTH;
```

ES5 代码

```
"use strict";
var dir = 0 /* NORTH */;
```

如果你要 TypeScript 保留对象 Direction , 那么可以添加编译选项 --preserveConstEnums

枚举作为类型使用

当满足一定条件时，枚举类型和其中的成员都可以当作是类型来使用

- enum E { A }：无初始值，但是这种类型的成员必须有前一个成员，而且前一个成员是数值类型
- enum E { A = 1 }：基本的有初始化的枚举类型
- enum E { A = 'a' }：字符串枚举

```
enum Animals {
  Dog = 1,
  Cat = 2
}

// type 的值只能是 Animals 中的 Dog 成员，也就是说 type 只能是 1
interface Dog {
  type: Animals.Dog
}

const dog: Dog = {
  type: Animals.Dog
  // type: 1，type 也可以直接赋值为数值
}

// 如果直接使用枚举变量，那么相当于高级类型的字符自变量类型
enum Animals {
  Dog = 1,
  Cat = 2
}

interface Animal {
  type: Animals // type 的值可为 1 或 2
}

const dog: Dog = {
  type: Animals.Dog // type: 1
  // type: Animals.Cat // type: 2
}
```

函数

函数声明

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

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

注意，**输入多余的（或者少于要求的）参数，是不被允许的**：

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

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

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

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

即便你对参数没有做类型注解，TypeScript 依然会检查传入参数的数量是否正确

函数表达式

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

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

这是可以通过编译的，

不过事实上，上面的代码只对等号右侧的匿名函数进行了类型定义，

而等号左边的 mySum，是通过赋值操作进行类型推断来的。

如果需要我们手动给 mySum 添加类型，则应该是这样：

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

注意不要混淆了 TypeScript 中的 => 和 ES6 中的 =>。

在 TypeScript 的类型定义中，=> 用来表示函数的定义，

左边是输入类型，需要用括号括起来，右边是输出类型。

异步函数的返回值

对于异步函数，需要用 `Promise<T>` 类型来定义它的返回值，这里的 `T` 是泛型，

取决于函数最终返回一个什么样的值（ `async / await` 也适用这个类型）。

例如这个例子，这是一个异步函数，会 `resolve` 一个字符串，

所以它的返回类型是 `Promise<string>` （假如没有 `resolve` 数据，那么就是 `Promise<void>` ）。

```
// 注意这里的返回值类型
function queryData(): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Hello World')
    }, 3000)
  })
}

queryData().then((data) => console.log(data))
```

用接口定义函数的类型

函数也是对象类型，所以接口也可以描述函数类型。

我们需要给接口定义一个签名（参数个数、参数类型、返回值类型）。

它就像是一个只有参数列表（参数列表里的每个参数都要有名字和类型）和返回值类型的函数定义。

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

```
// 创建一个函数类型的变量
let mySearch: SearchFunc;

// 将一个同类型的函数赋值给这个变量
mySearch = function(source: string, subString: string) {
  let result = source.search(subString);
  return result > -1;
}

// 上面相当于约定成这样
mySearch = function(source: string, subString: string) : boolean {
  let result = source.search(subString);
  return result > -1;
}
```

函数的参数名不需要与接口里定义的名字相匹配。

```
let mySearch: SearchFunc;

mySearch = function(src: string, sub: string): boolean {
  let result = src.search(sub);
  return result > -1;
}
```

但是，函数的参数会逐个进行检查，要求对应位置上的参数类型是兼容的。

如果不指定参数的类型，

TypeScript 的类型系统会推断出参数类型，因为函数直接赋值给了 SearchFunc 类型变量。

函数的返回值类型是通过其返回值推断出来的（下例是 false 和 true）。

如果让这个函数返回数字或字符串，

类型检查器会警告我们函数的返回值类型与 SearchFunc 接口中的定义不匹配。

```
let mySearch: SearchFunc;

mySearch = function(src, sub) {
  let result = src.search(sub);
  return result > -1;
}
```

注意：

采用函数表达式或接口定义函数的方式时，对等号左侧进行类型限制，

可以保证以后对函数名赋值时保证参数个数、参数类型、返回值类型不变。

可选参数

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

与接口中的可选属性类似，我们用 ? 表示可选的参数：

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

> **注意：可选参数必须接在必需参数后面。**

换句话说，**可选参数后面不允许再出现必需参数了**：

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

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

参数默认值

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

```
function buildName(firstName: string, lastName: string = 'Cat') {
  return firstName + ' ' + lastName;
}

let tomcat = buildName('Tom', 'Cat');
let tom = buildName('Tom');
```

此时就不受「可选参数必须接在必需参数后面」的限制了：

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

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

剩余参数

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

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

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

事实上，items 是一个数组。所以我们可以用数组的类型来定义它：

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

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

注意：rest 参数只能是最后一个参数

重载

在其他一些强类型语言中，

函数重载是指定义几个函数名相同，但参数个数或类型不同的函数，

在调用时传入不同的参数，编译器会自动调用适合的函数。

着重强调的是，这里的函数重载区别于其他语言中的重载，

TypeScript 中的重载是为了针对不同参数个数和类型，推断返回值类型。

在 TypeScript 中有函数重载的概念，

但并不是定义几个同名实体函数，而是根据不同的参数个数或类型来自动调用相应的函数。

TypeScript 重载允许一个函数接受数量不同或类型不同的参数时，作出不同的处理。

也就是说：**可以定义多个参数名称相同但参数类型（或参数数量）和返回值类型不同的函数。**

```
function add(a:string, b:string): string;   // 重载

function add(a:number, b:number): number;  // 重载

function add(a: any, b:any): any {          // 函数实现
  return a + b;
}

add("Hello ", "Steve"); // returns "Hello Steve"
add(10, 20); // returns 30
```

定义了两个同名的重载函数，

重载函数没有实际的函数体逻辑，而是只定义函数名、参数及参数类型以及函数的返回值类型。

最后一个函数是函数实现，

是一个完整的实体函数，包含函数名、参数及参数类型、返回值类型和函数体。

根据前两个函数声明（重载），由于返回类型可以是字符串或数字，

因此我们必须在函数定义时使用兼容的参数类型，所以返回类型为 any 。

再比如，我们需要实现一个函数 reverse，

输入数字 123 的时候，输出反转的数字 321，

输入字符串 'hello' 的时候，输出反转的字符串 'olleh'。

利用联合类型，我们可以这么实现：

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

然而这样有一个缺点，就是不能够精确的表达，

输入为数字的时候，输出也应该为数字，输入为字符串的时候，输出也应该为字符串。

这时，我们可以使用重载定义多个 reverse 的函数类型：

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

上例中，我们重复定义了多次函数 reverse，前两次都是函数定义，最后一次是函数实现。

在编辑器的代码提示中，可以正确的看到前两个提示。

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
  // Check to see if we're working with an object/array
  // if so, they gave us the deck and we'll pick the card
  if (typeof x == "object") {
    let pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  }
  // Otherwise just let them pick the card
  else if (typeof x == "number") {
    let pickedSuit = Math.floor(x / 13);
    return { suit: suits[pickedSuit], card: x % 13 };
  }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, {
suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

pickCard 方法根据传入参数的不同会返回两种不同的类型。

注意，function pickCard(x): any 并不是重载列表的一部分，

因此这里只有两个重载：一个是接收对象，另一个是接收数字。

以其它参数调用 pickCard 会产生错误。

为了让编译器能够选择正确的检查类型，它与 JavaScript 里的处理流程相似。

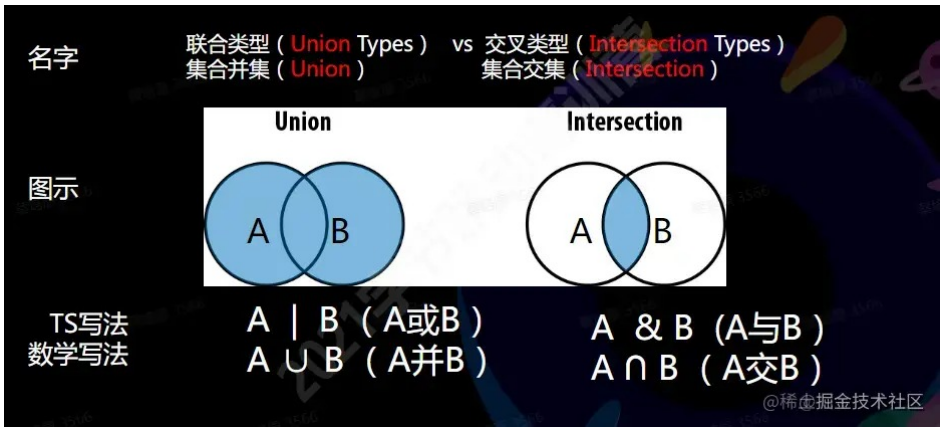
编译器会根据这个重载列表去处理函数的调用。

它查找重载列表，尝试使用第一个重载定义。 如果匹配的话就使用这个。

因此，在定义重载的时候，一定要把最精确的定义放在最前面。

所以多个函数定义如果有包含关系，需要优先把精确的定义写在前面。

最后还有一点要注意的是，这里重载只能用 function 来定义，不能使用接口、类型别名等。



联合类型

一个联合类型是由两个或者更多类型组成的类型，表示值可能是这些类型中的任意一个，这其中每个类型都是联合类型的成员。

联合类型（Union Types）表示取值可以为多种类型中的一种。

简单的例子

联合类型使用 | 分隔每个类型。

这里的 let myFavoriteNumber: string | number 的含义是：

允许 myFavoriteNumber 的类型是 string 或者 number，但是不能是其他类型。

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

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

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

访问联合类型的属性或方法

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，

TypeScript 会要求你做的事情，必须对每个联合的成员都是有效的

所以，我们**只能访问此联合类型的所有类型里共有的属性或方法**：

```
function getLength(something: string | number): number {
  return something.length;
}

// index.ts(2,22): error TS2339: Property 'length' does not exist on type 'string
| number'.
//   Property 'length' does not exist on type 'number'.
```

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

访问 string 和 number 的共有属性是没问题的：

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

举个例子，如果你有一个联合类型 string | number , 你不能使用只存在 string 上的方法：

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

解决方案是用代码收窄联合类型

```
function printId(id: number | string) {
  if (typeof id === "string") {
    // In this branch, id is of type 'string'
    console.log(id.toUpperCase());
  } else {
    // Here, id is of type 'number'
    console.log(id);
  }
}
```

再举一个例子，使用函数，比如 `Array.isArray`：

```
function welcomePeople(x: string[] | string) {
  if (Array.isArray(x)) {
    // Here: 'x' is 'string[]'
    console.log("Hello, " + x.join(" and "));
  } else {
    // Here: 'x' is 'string'
    console.log("Welcome lone traveler " + x);
  }
}
```

注意在 `else` 分支，我们并不需要做任何特殊的事情，如果 `x` 不是 `string[]`，那么它一定是 `string`

联合类型的变量在被赋值的时候，会根据类型推断的规则推断出一个类型：

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

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

上例中，第二行的 myFavoriteNumber 被推断成了 string，访问它的 length 属性不会报错。

而第四行的 myFavoriteNumber 被推断成了 number，访问它的 length 属性时就报错了。

字符串字面量类型

字符串字面量类型用来约束取值只能是某几个字符串中的一个。

简单的例子

```
type EventNames = 'click' | 'scroll' | 'mousemove';

function handleEvent(ele: Element, event: EventNames) {
  // do something
}

handleEvent(document.getElementById('hello'), 'scroll');  // 没问题
handleEvent(document.getElementById('world'), 'dblclick'); // 报错，event 不能为
'dblclick'
// index.ts(7,47): error TS2345: Argument of type '"dblclick"' is not assignable
to parameter of type 'EventNames'.
```

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

注意，类型别名与字符串字面量类型都是使用 type 进行定义。

类型保护（类型收窄）

对于联合类型的变量，在使用时通过类型断言或者类型保护，确切告诉编译器它是哪一种类型

使用类型保护时，TS 可以进一步缩小变量的类型

```
let getRandomValue = ():(string | number) => {
  let num = Math.random();
  return (num >= 0.5) ? 'abc' : 123.123;
}
// let value = getRandomValue();
// console.log(value);

/*
// 虽然通过类型断言可以确切的告诉编译器当前的变量是什么类型，
// 但是每一次使用的时候都需要手动的告诉编译器，这样比较麻烦，冗余代码也比较多
if ((value as string).length) {
    console.log((value as string).length);
} else {
    console.log((value as number).toFixed());
}
 */

/*
// 定义了一个类型保护函数，这个函数的'返回类型'是一个字符串类型
// 这个函数的返回值类型是，传入的参数 + is 具体类型
function isString(value:(string | number)): value is string {
    return typeof value === 'string';
}

if (isString(value)) {
    console.log(value.length);
} else {
    console.log(value.toFixed());
}
 */

/*
// 除了可以通过定义类型保护函数的方式来告诉编译器使用时联合类型的变量具体是什么类型以外
// 我们还可以使用 typeof 来实现类型保护
// 注意点：
// 如果使用 typeof 来实现类型保护，那么只能使用 === / !==
// 如果使用 typeof 来实现类型保护，那么只能保护 number/string/boolean/symbol 类型
if (typeof value === 'string') {
    console.log(value.length);
} else {
    console.log(value.toFixed());
}
 */

// 除了可以通过 typeof 类实现类型保护以外，我们还可以通过 instanceof 来实现类型保护
class Person {
  name:string = 'lnj';
}
```

```
class Animal {
  age: number = 18;
}

let getRandomObject = ():(Person | Animal) => {
  let num = Math.random();
  return (num >= 0.5) ? new Person() : new Animal();
};

let obj = getRandomObject();
console.log(obj);

if (obj instanceof Person) {
  console.log(obj.name);
} else {
  console.log(obj.age);
}
```

in 关键字

```
interface Admin {
  name: string;
  privileges: string[];
}

interface Employee {
  name: string;
  startDate: Date;
}

type UnknownEmployee = Employee | Admin;

function printEmployeeInformation(emp: UnknownEmployee) {
  console.log("Name: " + emp.name);
  if ("privileges" in emp) {
    console.log("Privileges: " + emp.privileges);
  }
  if ("startDate" in emp) {
    console.log("Start Date: " + emp.startDate);
  }
}
```

交叉类型

TypeScript 中交叉类型是将多个类型合并为一个类型。

通过 & 运算符可以将现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。

```ts
type PartialPointX = { x: number; };
type Point = PartialPointX & { y: number; };

let point: Point = {
  x: 1,
  y: 1
}
```

上面代码中我们先定义了 PartialPointX 类型，

接着使用 & 运算符创建一个新的 Point 类型，表示一个含有 x 和 y 坐标的点，

然后定义了一个 Point 类型的变量并初始化。

同名基础类型属性的合并

假设在合并多个类型的过程中，刚好出现某些类型存在相同的成员，但对应的类型又不一致，比如：

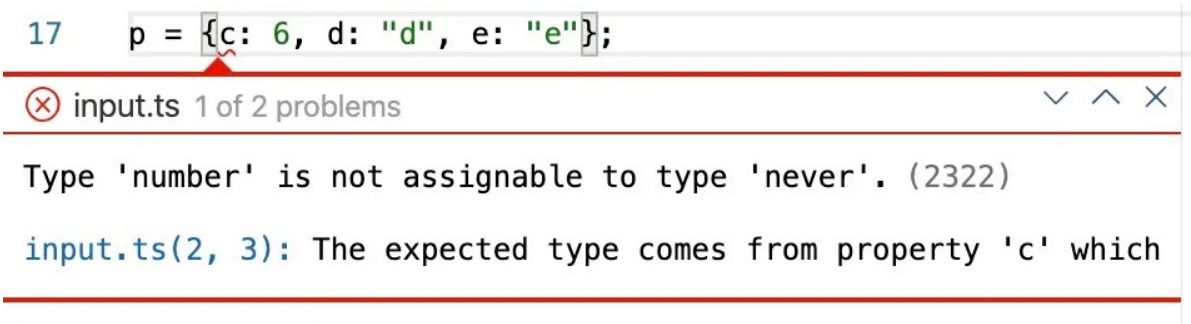
```ts
interface X {
  c: string;
  d: string;
}

interface Y {
  c: number;
  e: string
}

type XY = X & Y;
type YX = Y & X;

let p: XY;
let q: YX;
```

上面代码中，接口 X 和接口 Y 都含有一个相同的成员 c，但它们的类型不一致。

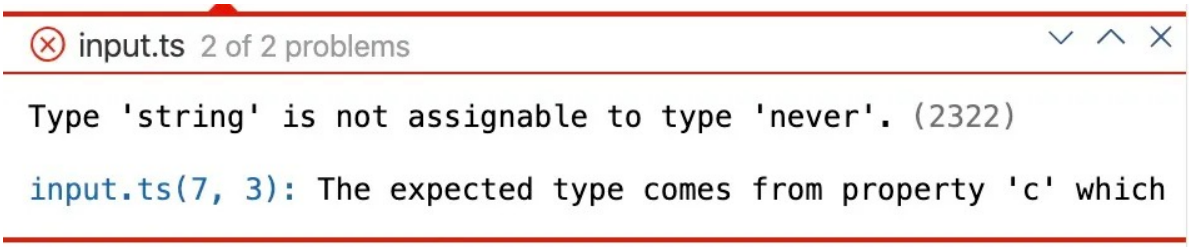
对于这种情况，此时 XY 类型或 YX 类型中成员 c 的类型是不是可以是 `string` 或 `number` 类型呢？

比如下面的例子：

```ts
p = { c: 6, d: "d", e: "e" };
```




```ts
q = { c: "c", d: "d", e: "e" };
```



为什么接口 X 和接口 Y 混入后，成员 c 的类型会变成 never 呢?

这是因为混入后成员 c 的类型为 string & number，

即成员 c 的类型既可以是 string 类型又可以是 number 类型。

很明显这种类型是不存在的，所以混入后成员 c 的类型为 never。

同名非基础类型属性的合并

```
interface D { d: boolean; }
interface E { e: string; }
interface F { f: number; }

interface A { x: D; }
interface B { x: E; }
interface C { x: F; }

type ABC = A & B & C;

let abc: ABC = {
  x: {
    d: true,
    e: 'semlinker',
    f: 666
  }
};

console.log('abc:', abc);
```

在混入多个类型时，若存在相同的成员，且成员类型为非基本数据类型，是可以成功合并的。

字面量类型

字面量类型就是将 `string`、 `number`、 `boolean` 特定的值作为类型

除了常见的类型 `string` 和 `number` ，我们也可以将类型声明为更具体的数字或者字符串。

众所周知，在 JavaScript 中，有多种方式可以声明变量。

比如 `var` 和 `let`，这种方式声明的变量后续可以被修改，

还有 `const`，这种方式声明的变量则不能被修改，这会影响 TypeScript 为字面量创建类型。

```
let changingString = "Hello World";
changingString = "Olá Mundo";
// Because `changingString` can represent any possible string, that
// is how TypeScript describes it in the type system
changingString;
// let changingString: string
```

```
const constantString = "Hello World";
// Because `constantString` can only represent 1 possible string, it
// has a literal type representation
constantString;
// const constantString: "Hello World"
```

字面量类型本身并没有什么太大用：

```
let x: "hello" = "hello";
// OK
x = "hello";
// ...
x = "howdy";
// Type '"howdy"' is not assignable to type '"hello"'.
```

如果结合联合类型，就显得有用多了。举个例子，当函数只能传入一些固定的字符串时：

```
function printText(s: string, alignment: "left" | "right" | "center") {
  // ...
}
printText("Hello, world", "left");
printText("G'day, mate", "centre");
// Argument of type '"centre"' is not assignable to parameter of type '"left" |
"right" | "center"'.
```

数字字面量类型也是一样的：

```
function compare(a: string, b: string): -1 | 0 | 1 {
  return a === b ? 0 : a > b ? 1 : -1;
}
```

当然了，你也可以跟非字面量类型联合：

```
interface Options {
  width: number;
}
function configure(x: Options | "auto") {
  // ...
}
configure({ width: 100 });
configure("auto");
configure("automatic");

// Argument of type '"automatic"' is not assignable to parameter of type 'Options
| "auto"'.
```

还有一种字面量类型，布尔字面量。

因为只有两种布尔字面量类型， `true` 和 `false` ，类型 `boolean` 实际上就是联合类型 `true | false` 的别名。

字面量推断

当你初始化变量为一个对象的时候，TypeScript 会假设这个对象的属性的值未来会被修改，

举个例子，如果你写下这样的代码：

```
const obj = { counter: 0 };
if (someCondition) {
  obj.counter = 1;
}
```

TypeScript 并不会认为 `obj.counter` 之前是 `0` ， 现在被赋值为 `1` 是一个错误。

换句话说， `obj.counter` 必须是 `number` 类型，但不要求一定是 `0` ，因为类型可以决定读写行为。

这也同样应用于字符串:

```
declare function handleRequest(url: string, method: "GET" | "POST"): void;

const req = { url: "https://example.com", method: "GET" };
handleRequest(req.url, req.method);

// Argument of type 'string' is not assignable to parameter of type '"GET" |
"POST"'.
```

在上面这个例子里， `req.method` 被推断为 `string` ，而不是 `"GET"` ，

因为在创建 `req` 和 调用 `handleRequest` 函数之间，可能还有其他的代码，或许会将 `req.method` 赋值一个新字符串比如 `"Guess"` 。

所以 TypeScript 就报错了。

有两种方式可以解决：

1. 添加一个类型断言改变推断结果：

```
// Change 1:
const req = { url: "https://example.com", method: "GET" as "GET" };
// Change 2
handleRequest(req.url, req.method as "GET");
```

Change 1 表示“我有意让 `req.method` 的类型为字面量类型 `"GET"`，这会阻止未来可能赋值为 `"GUESS"` 等字段”。

Change 2 表示“我知道 `req.method` 的值是 `"GET"`”.

1. 你也可以使用 `as const` 把整个对象转为一个类型字面量：

```
const req = { url: "https://example.com", method: "GET" } as const;
handleRequest(req.url, req.method);
```

`as const` 效果跟 `const` 类似，但是对类型系统而言，它可以确保所有的属性都被赋予一个字面量类型，

而不是一个更通用的类型比如 `string` 或者 `number` 。

泛型

泛型是广泛的数据类型，不是特定的数据类型，解决了函数、接口或类的类型复用性问题。

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

刚接触 TypeScript 泛型首次看到 `<T>` 语法会感到陌生。

但这没什么可担心的，就像普通传递参数一样，不过传递的是想要用于某个函数调用的类型而已。

普通函数参数处理值，而泛型处理类型参数，都有声明和调用。

泛型在函数调用时将 <> 的类型参数（或者 ts 自动推导类型）传递给函数定义的 <> ，再传递给函数定义里的参数和返回值。

```
// Example 1
// Declare a regular function
function regularFunc(x: any) {
  // You can use x here
}
// Call it: x will be 1
regularFunc(1)

// Declare a generic function
function genericFunc<T>() {
  // You can use T here
}
// Call it: T will be number
genericFunc<number>()


// Example 2
// Specify x to be number
function regularFunc(x: number)
// Success
```

```
regularFunc(1)
// Error
regularFunc('foo')

// Limits the type of T
function genericFunc<T extends number>()
// Success
genericFunc<number>()
// Error
genericFunc<string>()


// Example 3
// Set the default value of x
function regularFunc(x = 2)
// x will be 2 inside the function
regularFunc()

// Set the default type of T
function genericFunc<T = number>()
// T will be number inside the function
genericFunc()
```

图中 `<T>` 内部的 `T` 被称为类型变量，一种可以捕获类型的方式，它是我们希望传递给 identity 函数的类型占位符，

同时它被分配给 `value` 参数用来代替它的类型：

此时 `T` 充当的是类型，而不是特定的 Number 类型。

其中 `T` 代表 Type，在定义泛型时通常用作第一个类型变量名称。

但实际上 `T` 可以用任何有效名称代替。

除了 `T` 之外，以下是常见泛型变量代表的意思：

- **K（Key）：表示对象中的键类型；**
- **V（Value）：表示对象中的值类型；**
- **E（Element）：表示元素类型。**

注意：动态图最后一句错了，console.log(identity([1,2,3])) 这里注入类型应该是

```
function identity<T>(value: T): T {
  return value
}

console.log(identity<number[]>([1,2,3]))
```

变成：

```
function identity<number[]>(value: number[]): number[] {
  return value
}

console.log(identity<number[]>([1,2,3]))
```

而且，基本类型都是小写的 number ，不是大写的 Number

我们在什么时候需要使用泛型呢?

通常在决定是否使用泛型时，我们有以下两个参考标准：

1、当函数、接口或类将处理多种数据类型时；

2、当函数、接口或类在多个地方使用该数据类型时。

简单的例子

```
// 普通类型定义
type Dog<T> = { name: string, type: T }
// 普通类型使用
const dog: Dog<number> = { name: 'ww', type: 20 }

// 类定义
class Cat<T> {
```

```
  private type: T;
  constructor(type: T) { this.type = type; }
}
// 类使用
const cat: Cat<number> = new Cat<number>(20); // 或简写 const cat = new Cat(20)

// 函数定义
function swipe<T, U>(value: [T, U]): [U, T] {
  return [value[1], value[0]];
}
// 函数使用
swipe<Cat<number>, Dog<number>>([cat, dog])  // 或简写 swipe([cat, dog])
```

首先，我们来实现一个函数 `createArray`，

它可以创建一个指定长度的数组，同时将每一项都填充一个默认值：

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

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

上例中，我们使用了**之前提到过的数组泛型**来定义返回值的类型。

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

`Array<any>` 允许数组的每一项都为任意类型。

但是我们预期的是，数组中每一项都应该是输入的 `value` 的类型。

这时候，泛型就派上用场了：

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

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

上例中，我们在函数名后添加了 `<T>`，其中 `T` 用来指代任意输入的类型，

在后面的输入 `value: T` 和输出 `Array<T>` 中即可使用了。

接着在调用的时候，可以指定它具体的类型为 `string`。

当然，也可以不手动指定，让类型推断自动推算出来：

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

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

多个类型参数

定义泛型的时候，可以一次定义多个类型参数：

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

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

上例中，我们定义了一个 `swap` 函数，用来交换输入的元组。

泛型推导（常用）

TS 会自动根据变量定义时的类型推导出变量类型，这一般是发生在函数调用的场合的。

也叫类型参数推断（类型推断）

```ts
type Dog<T> = { name: string, type: T }

function adopt<T>(dog: Dog<T>) { return dog };

const dog = { name: 'ww', type: 'hsq' };  // 这里按照 Dog 类型的定义一个 type 为 string 的对象
adopt(dog);  // Pass: 函数会根据入参类型推断出 T 为 string（更为常见）
```

不使用泛型推导，定义变量类型时必须指定泛型类型。

```ts
const dog: Dog<string> = { name: 'ww', type: 'hsq' }  // 不可省略<string>这部分（使用较少）
```

如果我们想不指定，可以使用泛型默认值的方案。

```ts
type Dog<T = any> = { name: string, type: T }
const dog: Dog = { name: 'ww', type: 'hsq' }
dog.type = 123;    // 不过这样 type 类型就是 any 了，无法自动推导出来，失去了泛型的意义
```

泛型的默认类型

在 TypeScript 2.3 以后，我们可以为泛型中的类型参数指定默认类型。

当使用泛型时没有在代码中直接指定类型参数，从实际值参数中也无法推测出时，

这个默认类型就会起作用。

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

```
function makeState<S extends number | string = number>() {
  let state: S
  function getState() {
    return state
  }
  function setState(x: S) {
    state = x
  }
  return { getState, setState }
}

// 不需要传入 <number>
const numState = makeState()
numState.setState(1)
console.log(numState.getState()) // 1
```

泛型约束

泛型没有约束，表示可以任意传入某种类型（传入后，类型就固定了），

泛型约束表示只能传入约束的类型，其它类型都不接受

```
function makePair<
  F extends number | string,
  S extends boolean | F
>() {
  let pair: { first: F; second: S }
  function getPair() {
    return pair
  }
  function setPair(x: F, y: S) {
    pair = {
      first: x,
      second: y
    }
  }
  return { getPair, setPair }
}
```

```
// These will work
makePair<number, boolean>()
makePair<number, number>()
makePair<string, boolean>()
makePair<string, string>()

// This will fail because the second
// parameter must extend boolean | number,
// but instead it's string
makePair<number, string>()
```

确保属性存在

在函数内部使用泛型变量的时候，

由于事先不知道它是哪种类型，所以不能随意的操作它的属性或方法：

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

// index.ts(2,19): error TS2339: Property 'length' does not exist on type 'T'.
```

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

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

这就是泛型约束：

```
interface Lengthwise {
  length: number;
}

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

上例中，我们使用了 `extends` 约束了泛型 `T` 必须符合接口 `Lengthwise` 的形状，

也就是必须包含 `length` 属性。

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

```ts
interface Lengthwise {
 length: number;
}

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

loggingIdentity(7);

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

多个类型参数之间也可以互相约束：

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

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

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

上例中，我们使用了两个类型参数，

其中要求 `T` 继承 `U`，这样就保证了 `U` 上不会出现 `T` 中不存在的字段，

即 `U` 可以约束 `T`，但 `T` 不会影响 `U`。

检查对象上的键是否存在

泛型约束的另一个常见的使用场景就是检查对象上的键是否存在。

在看具体示例之前，我们得来了解一下 keyof 操作符，keyof 操作符是在 TypeScript 2.1 版本引入的，

该操作符可以用于获取某种类型的所有键，其返回类型是联合类型。

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

type K1 = keyof Person; // "name" | "age" | "location"
type K2 = keyof Person[];  // number | "length" | "push" | "concat" | ...
type K3 = keyof { [x: string]: Person };  // string | number
```

通过 keyof 操作符，我们就可以获取指定类型的所有键，

之后我们就可以结合前面介绍的 extends 约束，即限制输入的属性名包含在 keyof 返回的联合类型中。

具体的使用方式如下：

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}
```

在以上的 getProperty 函数中，

我们通过 K extends keyof T 确保参数 key 一定是对象中含有的键，

这样就不会发生运行时错误，这是一个类型安全的解决方案，与简单调用 let value = obj[key]; 不同。

我们来看一下如何使用 getProperty 函数：

```
enum Difficulty {
  Easy,
  Intermediate,
  Hard
}

function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

let tsInfo = {
  name: "Typescript",
  supersetOf: "Javascript",
  difficulty: Difficulty.Intermediate
}

let difficulty: Difficulty = getProperty(tsInfo, 'difficulty'); // OK

let supersetOf: string = getProperty(tsInfo, 'superset_of'); // Error
```

以上示例中，对于 getProperty(tsInfo, 'superset_of') 这个表达式，

TypeScript 编译器会提示以下错误信息：

```
Argument of type '"superset_of"' is not assignable to parameter of type
'"difficulty" | "name" | "supersetOf"'.(2345)
```

通过使用泛型约束，在编译阶段我们就可以提前发现错误，大大提高了程序的健壮性和稳定性。

泛型条件类型

TypeScript 2.8 中引入了条件类型，

使得我们可以根据某些条件得到不同的类型，这里所说的条件是类型兼容性约束。

尽管代码中使用了 extends 关键字，也不一定要强制满足继承关系，而是检查是否满足结构兼容性。

下面表达式的意思是：若 T 能够赋值给 U，那么类型是 X，否则为 Y。

```
T extends U ? X : Y
```

泛型推断 infer

infer 的中文是“推断”的意思，一般是搭配上面的泛型条件语句使用的，

所谓推断，就是你不用预先指定在泛型列表中，在运行时会自动判断，不过你得先预定义好整体的结构。

看 extends 后面的内容，{t: infer Test} 可以看成是一个包含 t 属性的类型定义，

这个 t 属性的 value 类型通过 infer 进行推断后会赋值给 Test 类型，

如果泛型实际参数符合{t: infer Test}的定义那么返回的就是 Test 类型，否则返回的就是默认的 string 类型。

举个例子

```
type Foo<T> = T extends {t: infer Test} ? Test: string

type One = Foo<number>
// string，因为 number 不是一个包含 t 的对象类型， T extends string
type Two = Foo<{t: boolean}>
// boolean，因为泛型参数匹配上了，使用 infer 对应的 type， T extends boolean
type Three = Foo<{a: number, t: () => void}>
// () => void，泛型定义是参数的子集，同样适配， T extends () => void
```

另一个例子

```
type ParamType<T> = T extends (param: infer P) => any ? P : T;

interface User {
  name: string;
  age: number;
}

type Func = (user: User) => void

type Param = ParamType<Func>;   // Param = User
type AA = ParamType<string>;    // string
```

泛型接口

接受类型参数的接口叫泛型接口。

提取泛型接口前

```
function makePair<F, S>() {
  let pair: { first: F; second: S }
  function getPair() {
    return pair
  }
  function setPair(x: F, y: S) {
    pair = {
      first: x,
      second: y
    }
  }
  return { getPair, setPair }
}
```

我们可以将 `{ first: F, second: S }` 重构为接口或类型别名，方便复用

提取泛型接口后

首先提取 pair 的类型为泛型接口，

为了与 makePair 的类型参数区分开来，使用 A 和 B 作为类型参数，

使用这个接口来声明 pair 的类型

```
// 提取成一个泛型接口，使其可以复用
interface Pair<A, B> {
  first: A
  second: B
}

function makePair<F, S>() {
  // 使用： F 传给 A ， S 传给 B
  let pair: Pair<F, S>
  function getPair() {
    return pair
  }
  function setPair(x: F, y: S) {
    pair = {
      first: x,
      second: y
    }
  }
  return { getPair, setPair }
}
```

提取泛型类型别名

或者，我们可以将其提取为泛型类型别名。

对于对象类型，类型别名与接口基本相同，因此可以使用你喜欢的任何一种。

```
// 提取成一个泛型类型别名
// 与使用泛型接口基本相同
type Pair<A, B> = {
  first: A
  second: B
}

function makePair<F, S>() {
  // 使用： F 传给 A ， S 传给 B
  let pair: Pair<F, S>
  function getPair() {
    return pair
  }
  function setPair(x: F, y: S) {
    pair = {
      first: x,
      second: y
    }
  }
```

```
  return { getPair, setPair }
}
```

之前学习过可以使用接口的方式来定义一个函数需要符合的形状：

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

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

当然也可以使用含有泛型的接口来定义函数的形状：

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

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

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

进一步，我们可以把泛型参数提前到接口名上：

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

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

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

注意，此时在使用泛型接口的时候，需要定义泛型的类型。

```
interface Identities<V, M> {
  value: V,
  message: M
```

```
}

function identity<T, U> (value: T, message: U): Identities<T, U> {
  console.log(value + ": " + typeof (value));
  console.log(message + ": " + typeof (message));
  let identities: Identities<T, U> = {
    value,
    message
  };
  return identities;
}

console.log(identity(68, "Tom"));
// 68: number
// Tom: string
// {value: 68, message: "Tom"}
```

泛型类

泛型函数

```
function makeState<S>() {
  let state: S
  function getState() {
    return state
  }
  function setState(x: S) {
    state = x
  }
  return { getState, setState }
}

// Creates a number-only state
const numState = makeState<number>()
numState.setState(1)
console.log(numState.getState())
```

泛型类

```
class State<S> {
  state: S
  getState() {
    return this.state
  }
  setState(x: S) {
    this.state = x
  }
}

// Pass a type parameter on initialization
const numState = new State<number>()
numState.setState(1)
// Prints 1
console.log(numState.getState())
```

泛型类看上去与泛型函数差不多， 泛型类使用 <> 括起泛型类型，跟在类名后面

泛型函数在调用时采用类型参数，而泛型类是在实例化时采用类型参数。

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

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

下面代码里，我们以实例化 `myNumberClass` 为例，来分析一下其调用过程：

- 在实例化 `IdentityClass` 对象时，我们传入 `Number` 类型和构造函数参数值 `68`；
- 之后在 `IdentityClass` 类中，类型变量 `T` 的值变成 `Number` 类型；
- `IdentityClass` 类实现了 `GenericInterface<T>`，而此时 `T` 表示 `Number` 类型，因此等价于该类实现了 `GenericInterface<Number>` 接口；
- 而对于 `GenericInterface<U>` 接口来说，类型变量 `U` 也变成了 `Number`。这里我有意使用不同的变量名，以表明类型值沿链向上传播，且与变量名无关。

```
interface GenericInterface<U> {
  value: U
  getIdentity: () => U
}

class IdentityClass<T> implements GenericInterface<T> {
  value: T

  constructor(value: T) {
    this.value = value
  }

  getIdentity(): T {
    return this.value
  }
}

const myNumberClass = new IdentityClass<Number>(68);
console.log(myNumberClass.getIdentity()); // 68

const myStringClass = new IdentityClass<string>("Tom!");
console.log(myStringClass.getIdentity()); // Tom!
```

泛型类可确保在整个类中一致地使用指定的数据类型。

比如，你可能已经注意到在使用 Typescript 的 React 项目中使用了以下约定：

```
type Props = {
  className?: string
   ...
};

type State = {
  submitted?: bool
   ...
};

class MyComponent extends React.Component<Props, State> {
   ...
}
```

在以上代码中，我们将泛型与 React 组件一起使用，确保组件的 props 和 state 是类型安全的。

操作符

typeof

在 TypeScript 中，typeof 操作符可以用来获取一个变量声明或对象的类型。

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

const tom : Person = { name: 'Tom', age: 33 };
type Tom = typeof tom ; // -> Person

function toArray(x: number): Array<number> {
  return [x];
}

type Func = typeof toArray; // -> (x: number) => number[]
```

keyof

keyof 操作符提取一个对象类型中的所有 key 值组成的联合类型

keyof 是索引类型查询操作符。

[]是索引访问操作符，可以单独使用相当于是对象书写的 [] 形式，也可以与 keyof 操作符一起使用

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

type K1 = keyof Person;   // "name" | "age"
type K2 = keyof Person[]; // "length" | "toString" | "pop" | "push" | "concat" |
"join"
type K3 = keyof { [x: string]: Person };  // string | number
```

TypeScript 中支持两种索引签名，数字索引和字符串索引：

```
interface StringArray {
  // 字符串索引 -> keyof StringArray => string | number
  [index: string]: string;
}

interface StringArray1 {
  // 数字索引 -> keyof StringArray1 => number
  [index: number]: string;
}

interface Map<T> {
  [key: string]: T
}
let keys: keyof Map<number>
// 可索引接口的类型如果是 string 的话，keys 的类型为 string 或者 number

interface Map<T> {
  [key: number]: T;
}
let keys: keyof Map<number>;
// 可索引接口为 number 则 keys 的类型只能是 number
```

为了同时支持两种索引类型，TS 要求数字索引的返回值必须是字符串索引返回值的子类。

其中的原因就是当使用数值索引时，JavaScript 在执行索引操作时，

会先把数值索引先转换为字符串索引。

所以 keyof { [x: string]: Person } 的结果会返回 string | number。

keyof 操作可以配合 type 的类型别名来实现类似字面量类型的结果

```
interface Type {
  a: string
  b: number
  c: boolean
  d: undefined
  e: null
  f: never
  h: object
}
type Props = keyof Type // 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'h'
type Test = Type[Props] //string | number | boolean | object | null | undefined
// 注意 never 被排除掉了
```

注意： 使用keyof或类似keyof这样需要在一些类型中选择类型时会将null、undefined、never类型排除掉

keyof 操作符经常与 extends 关键字一起作为泛型的约束来使用

```
// K只能是T的索引属性组成的一个数组，并且返回一个T属性值对应值组成的数组
function getValue<T, K extends keyof T>(
  obj: T,
```

```
  names: K[]
): Array<T[K]> /* 可以写成T[K][] */ {
  return names.map((n) => obj[n])
}

let obj = {
  name: '张三',
  age: 18
}

getValue(obj, ['name', 'age'])
```

in

in 用来遍历枚举类型：

```
type Keys = "a" | "b" | "c"

type Obj =  {
  [p in Keys]: any
} // -> { a: any, b: any, c: any }
```

注意：

k in keyof keys与k in keys是有区别的，前者是对类似对象类型的类型使用，后者是对联合类型使用。

前者的 keyof 只是把类似对象类型的类型的属性名转变为联合类型再进行后者的操作。

前者不能直接对泛型或者类似对象类型使用k in keys这样的语法，会报错，而对联合类型如果使用前者不会报错

工具类型

Partial<Type>

将类型 Type 里所有的属性变为可选项？

```
/**
 * node_modules/typescript/lib/lib.es5.d.ts
 * Make all properties in T optional
 */
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```

在以上代码中，首先通过 keyof T 拿到 T 的所有属性名，

然后使用 in 进行遍历，将值赋给 P，

最后通过 T[P] 取得相应的属性值。

中间的？号，用于将所有属性变为可选。

Required<Type>

将类型 Type 里所有的属性变为必选项。

```
interface Props {
  a?: number;
  b?: string;
}

const obj: Props = { a: 5 };

const obj2: Required<Props> = { a: 5 };
// Property 'b' is missing in type '{ a: number; }' but required in type
'Required<Props>'.
```

Readonly<Type>

Type 类型所有的属性设为 readonly 只读，不可以重新赋值

```
interface Todo {
  title: string;
}

const todo: Readonly<Todo> = {
  title: "Delete inactive users",
};

todo.title = "Hello";
Cannot assign to 'title' because it is a read-only property.
```

Pick<Type, Keys>

在一个声明好的对象中，挑选一部分出来组成一个新的声明对象

```
interface Todo {
  title: string;
  description: string;
  done: boolean;
}

type TodoBase = Pick<Todo, "title" | "done">;

// 相当于
type TodoBase = {
    title: string;
    done: boolean;
}
```

Omit<Type, Keys>

从类型里中删除某些属性

```
type Person = {
  name: string,
  age: number
}

const Tom: Omit<Person, "name"> = {age: 8};
const Tom: Omit<Person, "name" | "age"> = {};
```

Record<Keys, Type>

构建一个对象类型，该对象类型属性的键是Keys，属性的值是Type。

这个工具类型可以将一个类型的属性映射到另一个类型。

```
type CatName = "miffy" | "boris" | "mordred";

interface CatInfo {
  age: number;
  breed: string;
}

const cats: Record<CatName, CatInfo> = {
  miffy: { age: 10, breed: "Persian" },
  boris: { age: 5, breed: "Maine Coon" },
  mordred: { age: 16, breed: "British Shorthair" },
};

cats.boris;
```

Parameters<Type>

`Parameters<T>` 返回类型为 T 的函数的参数类型所组成的数组

```
type T0 = Parameters<() => string>;  // []

type T1 = Parameters<(s: string) => void>;  // [string]
```

ReturnType<Type>

提取函数的返回值类型

```
type T0 = ReturnType<() => string>;  // string

type T1 = ReturnType<(s: string) => void>;  // void
```

Extract<Type, Union>

从 Type 中提取与 Union 的所有 union 成员匹配的类型，构造一个类型。

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">;
// type T0 = "a"

type T1 = Extract<string | number | (() => void), Function>;
// type T1 = () => void
```

Exclude<Type, Union>

从 Type 中排除与 Union 的所有 union 成员匹配的类型，剩余的属性构成新的类型

```
type T0 = Exclude<'a' | 'b' | 'c', 'a'>;

// 相当于
type T0 = "b" | "c"
```

类

在 TypeScript，通过类得到的变量，它的类型就是这个类，可能这句话看起来有点难以理解，来看个例子，可以在 demo 里运行它：

ts

```
// 定义一个类
class User {
  // constructor 上的数据需要先这样定好类型
  name: string

  // 入参也要定义类型
  constructor(userName: string) {
    this.name = userName
  }

  getName() {
    console.log(this.name)
  }
}

// 通过 new 这个类得到的变量，它的类型就是这个类
const petter: User = new User('Petter')
petter.getName() // Petter
```

```
class Greeter {
  // 静态属性
  static cname: string = "Greeter";
  // 成员属性
  greeting: string;
```

```
  // 构造函数 - 执行初始化操作
  constructor(message: string) {
    this.greeting = message;
  }

  // 静态方法
  static getClassName() {
    return "Class name is Greeter";
  }

  // 成员方法
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world");
```

类的简写

```
class Animal {
    constructor(public name: string, private age: number, protected sex: string)
{}
}
```

等价于:

```
class Animal {
    public name: string;
    private age: number;
    protected sex: string;
    constructor(name: string, age: number, sex: string) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }
}
```

类的关键字

public 默认公开访问

private 只有类本身可访问，子类和实例都不可访问

protected 只有类本身和子类可访问，注意这里不是实例，也不是子类实例。

类实现接口

implements 关键字只能在 class 中使用，实现一个新的类，从父级或者从接口实现所有的属性和方法，

下面代码中，如果在 Person 类里面没写接口已有的属性和方法则会报错。

```
interface frontEnd {
    name: string,
    fn: () => void
}

class Person implements frontEnd {
    name: "前端";
    fn() {

    }
}
```

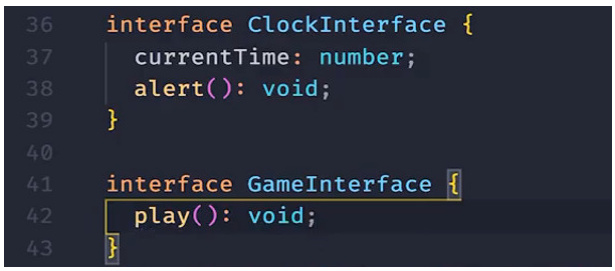
接口定义多个类之间共有的属性与方法，方便多个类复用，但只约束类的实例类型

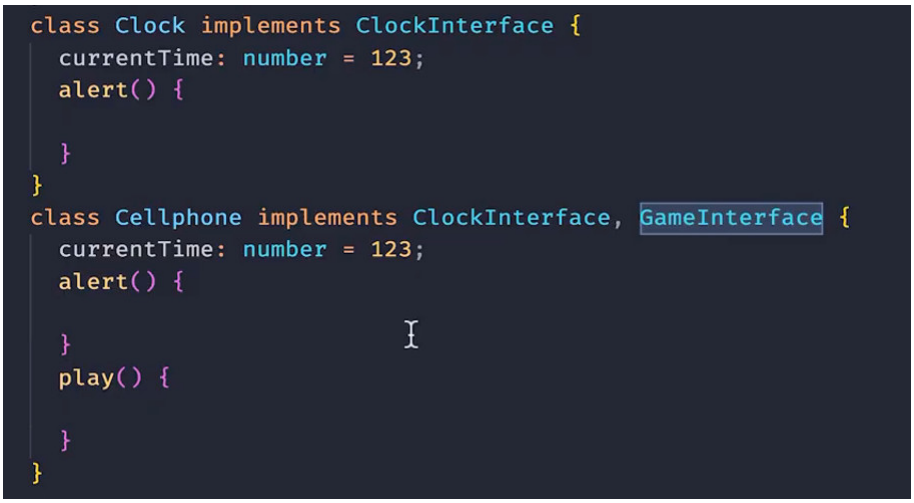
一个类可以实现多个接口：

```
interface Age {
  age: number;
}

interface Title {
  title: string;
}

class title implements Title, Age {
  title: string = '兔兔';
  age: number = 18;
}
```

实例类型接口

实例类型接口是针对类的接口。



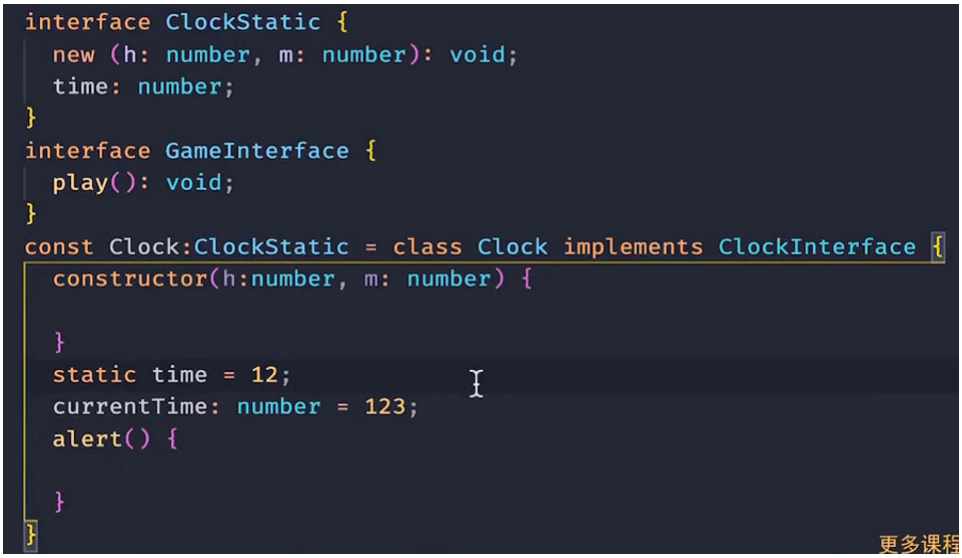


静态类型接口

静态类型接口是针对类构造函数的接口，原因是：静态属性或静态方法都直接挂在构造函数上。

对于静态类型需要新建另一个接口来约束，

静态类型接口与实例类型接口结合使用，如下




```
// 针对类构造函数的接口
interface CPerson {
  new(name: string);
}

// 针对类的接口
interface IPerson {
  name: string;
  age: number;
}

function create(c: CPerson, name: string): IPerson {
  return new c(name);
}

class People implements IPerson {
  name: string;
  age: number;
  // 这里未声明 构造函数，根据 ES6 规定会有默认的顶上来
}

let p = create(People, 'funlee'); // 可以
```

```
// 静态类型接口
interface CPerson {
  new(name: string):any;
}

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

```
let p: CPerson = class People implements IPerson {
    name: string;
    age: number;
    constructor(name: string) {}
}
```

抽象类

abstract 关键字定义抽象类，在抽象类内部定义抽象方法。

A 或者 B 正好需要一个公共属性，然后本身还有一些自己的逻辑，就可以使用抽象类。

```
// 抽象类
abstract class Boss {
  name = "张三";
  call() {} // 抽象方法不能写函数体
}

class A extends Boss {
  call() {
    console.log(this.name);
    console.log("A")
  }
}

class B extends Boss {
  call() {
    console.log("B")
  }
}

new A().call()
```

如果抽象类里面方法是抽象的，那么本身的类也必须是抽象的。

abstract 抽象类作为其它子类的基类使用，不能实例化。

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log("roaming the earth...");
    }
}

let animal = new Animal(); // Error: 抽象类不允许被实例化
```

抽象方法不能写函数体。父类里面有抽象方法，那么子类也必须要重写该方法。

也就是说，抽象类中的抽象方法不包含具体实现，但是必须在子类中实现。

抽象方法的语法与接口方法相似，两者都是定义函数签名但不包含函数体。

抽象方法必须包含 abstract 关键字，可以包含访问修饰符。

```
abstract class Department {
    constructor(public name: string) {
    }
    printName(): void {
        console.log('Department name: ' + this.name);
    }
    abstract printMeeting(): void; // 必须在子类中实现
}

class AccountingDepartment extends Department {
    constructor() {
        super('Accounting and Auditing'); // 在子类的构造函数中必须调用 super()
    }
    printMeeting(): void {
        console.log('The Accounting Department meets each Monday at 10am.');
    }
    generateReports(): void {
        console.log('Generating accounting reports...');
    }
}

let department: Department; // OK：允许创建一个对抽象类型的引用
department = new Department(); // Error: 不能创建一个抽象类的实例
department = new AccountingDepartment(); // OK：允许对一个抽象子类进行实例化和赋值
department.printName(); // OK
department.printMeeting(); // OK
department.generateReports(); // Error: 方法在声明的抽象类中不存在
```

类型推断

TypeScript 会自动推断类型。举个例子，**变量的类型可以基于初始值进行推断**：

```
// No type annotation needed -- 'myName' inferred as type 'string'
let myName = "Alice";
```

跟变量类型注解一样，也不需要总是添加返回值类型注解，TypeScript 会基于它的 `return` 语句推断函数的返回类型。

两种场景：变量初始化和函数返回

TypeScript 在没有明确的指定类型的时候会自动推测出一个类型，这就是类型推断。

明确声明类型与否取决于你的编码风格。

对于可维护性来说，明确声明是一个好的做法。

以下代码虽然没有指定类型，但是会在编译的时候报错：

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

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

事实上，它等价于：

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

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

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

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

类型断言

我们更希望 TypeScript 不要帮我们进行类型检查，而是交给我们自己来时，所以就用到了类型断言。

类型断言表示手动指定一个值的类型，好比其它语言里的类型转换，把某个值强行指定为特定类型。

它没有运行时的影响，只是在编译阶段起作用，TypeScript 会假设程序员已经检查过。

语法

```
值 as 类型
```

或

```
<类型>值
```

两种形式

“尖括号”语法

```
let someValue: any = "this is a string";

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

as 语法

```
let someValue: any = "this is a string";

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

两种形式是等价的。 至于使用哪个大多数情况下是凭个人喜好；

> **注意，在 TypeScript 里使用 JSX 时，只有 as 语法断言是被允许的。**

举个例子，

如果你使用 `document.getElementById`，

TypeScript 仅仅知道它会返回一个 `HTMLElement`，但是你却知道，你要获取的是一个 `HTMLCanvasElement`。

这时，你可以使用类型断言将其指定为一个更具体的类型：

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

就像类型注解一样，类型断言也会被编译器移除，并且不会影响任何运行时的行为。

你也可以使用尖括号语法（注意不能在 `.tsx` 文件内使用），是等价的：

```
const myCanvas = <HTMLCanvasElement>document.getElementById("main_canvas");
```

> **谨记：因为类型断言会在编译的时候被移除，所以运行时并不会有类型断言的检查，即使类型断言是错误的，也不会有异常或者 `null` 产生。**

类型断言的用途

将一个联合类型断言为其中一个类型

之前提过，当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，

我们只能访问此联合类型的所有类型中共有的属性或方法：

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

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

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

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

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

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

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

上面的例子中，获取 animal.swim 的时候会报错。

此时可以使用类型断言，将 animal 断言成 Fish：

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

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

需要注意的是，

类型断言只能够「欺骗」TypeScript 编译器，但是无法避免运行时的错误，

反而滥用类型断言可能会导致运行时错误：

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

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

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

上面的例子编译时不会报错，但在运行时会报错：

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

原因是 (animal as Fish).swim()

这段代码隐藏了 animal 可能为 Cat 的情况，将 animal 直接断言为 Fish 了，

而 TypeScript 编译器信任了我们的断言，故在调用 swim() 时没有编译错误。

可是 swim 函数接受的参数是 Cat | Fish，

一旦传入的参数是 Cat 类型的变量，由于 Cat 上没有 swim 方法，就会导致运行时错误了。

总之，使用类型断言时一定要格外小心，

尽量避免断言后调用方法或引用深层属性，以减少不必要的运行时错误。

用类型断言可以修改类型的限制：

联合类型可以被断言为其中一个类型

父类可以被断言为子类

任何类型都可以被断言为 any

any 可以被断言为任何类型

总结成一条规律就是：要使得 A 能够被断言为 B，只需要 A 兼容 B 或 B 兼容 A 即可。

TypeScript 仅仅允许类型断言转换为一个更加具体或者更不具体的类型。这个规则可以阻止一些不可能的强制类型转换，比如：

```
const x = "hello" as number;
// Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.
```

有的时候，这条规则会显得非常保守，阻止了你原本有效的类型转换。

如果发生了这种事情，你可以使用双重断言，先断言为 `any` （或者是 `unknown`），然后再断言为期望的类型：

```
const a = (expr as any) as T;
```

双重断言

利用上述 3 和 4 两条规则，可以强制把一个值改为任意其他类型

```
let a = 3
(a as any) as string).split // ok
```

const 断言

```
// Type '"hello"'
const x = { text: "hello" } as const;

// Type 'readonly [10, 20]'
let y = [10, 20] as const;

// Type '{ readonly text: "hello" }'
let z = { text: "hello" } as const;
```

表示该表达式中的字面类型不被扩展（例如：不能从“hello”转换为字符串），为表达式推断出最具体的类型

对象属性变为只读属性

数组变为只读元组

当我们使用关键字 const 声明一个字面量时，类型是等号右边的文字；

如果我们用 let 而不是 const， 那么该变量会被重新分配，并且类型会被扩展为字符串类型，如下所示：

```
const x = 'x'; // has the type 'x'
let y = 'x';   // has the type string
```

利用 const 断言，我们可以：

```
let y = 'x' as const; // y has type 'x'
```

例子：

```
const args = [8, 5];
// const args: number[]
const angle = Math.atan2(...args); // error! Expected 2 arguments, but got 0 or
more.
console.log(angle);
```

编译器看到 const args = [8, 5]; 并推断出 number[] 的类型，

是一个由零个或多个数字类型的元素组成的可变数组。

编译器不知道有多少个元素或哪些元素。

这样的推断通常是合理的。

下一行，Math.atan2(...args)，导致一个错误。

因为Math.atan2()函数只是需要两个数字参数。

但编译器对 args 的了解是：它是一个数字数组。

所以编译器抱怨说你在调用Math.atan2() 时有 "0或更多 "的参数，而它需要的正是两个参数。

将上述代码与用 as const 的代码比较：

```
const args = [8, 5] as const;
// const args: readonly [8, 5]
const angle = Math.atan2(...args); // okay
console.log(angle);
```

编译器推断出 args 的类型是只读的 [8, 5]，一个只读的元组，其值正好是按顺序排列的数字 8 和 5。

具体来说，编译器知道 args.length 正好是 2 。

所以下一行的 Math.atan2() 不会报错。

编译器知道 Math.atan2(...args) 与 Math.atan2(8, 5) 相同，这是一个有效的调用。

另外：两个版本在运行时，没有任何区别，在控制台打印都是 1.0121970114513341。

const 断言让编译器知道更多关于代码的意图，更准确地分辨出代码正确与否。

非空断言操作符

TypeScript 提供了一个特殊的语法，可以在不做任何检查的情况下，从类型中移除 `null` 和 `undefined`，

这就是在任意表达式后面写上 `!` ，这是一个有效的类型断言，表示它的值不可能是 `null` 或者 `undefined`：

```
function liveDangerously(x?: number | null) {
  // No error
  console.log(x!.toFixed());
}
```

就像其他的类型断言，这也不会更改任何运行时的行为。

重要的事情说一遍，只有当你明确的知道这个值不可能是 `null` 或者 `undefined` 时才使用 `!`

类型兼容

在 TypeScript 中另一大特性为类型兼容，

TypeScript 类型兼容性是基于结构子类型的，同时结构类型只使用其成员来描述类型

属性兼容性

规则：如果 x 要兼容 y，那么 y 至少具有与 x 相同的属性（x 是 y 的子集）

在为对象赋值时，会检测对象中是否含有应该有的属性，同时也会检验额外的属性，

如果直接使用对象字面量进行赋值会报错，

而如果对象字面量先赋值给其它变量，使用其它变量再给对象赋值就能通过检测。

官方说法：

如果认为 S 相对于 T 具有额外属性，首先 S 是一个 `fresh object literal type`，

并且 S 中含有 T 不期望存在的属性。

我们这里可以简单的理解为 `fresh object literal type` 就是一个直接的对象字面量

`fresh object literal types` 失去 freshness 情况如下:

- fresh 类型数据被 widened，结果数据的类型失去 freshness
- 类型断言后产生的数据类型类型失去 freshness

```
interface Info {
  name: string
}

let info: Info
const info1 = { name: '张三' }
const info2 = { age: 18 }
const info3 = { name: '张三', age: 18 }

info = info1
info = info2 //报错，因为没有name字段
info = info3 // 不会报错
info = { name: '张三', age: 18 } // 直接给会报错
info = { name: '张三', age: 18 } as any // 失去freshness，不会报错
```

这种检测为递归检测，会对对应的变量的值进行检测

```
interface Info {
  name: string
  info: { age: number }
}

let info: Info
const info1 = { name: '张三', info: { age: 18 } }
const info2 = { age: 18 }
const info3 = { name: '张三', age: 18 }

info = info1
// info = info2 报错，因为没有name字段
info = info3 // 报错，递归检测到info属性对象没有age属性
```

函数兼容性

参数兼容

个数兼容

函数的参数个数不同能够向下兼容，也就是参数个数少的函数能赋值给参数个数多的函数。

但是反之就不能成立。

```
let funcX = (num: number) => 0
let funcY = (num: number, str: string) => 0

funcY = funcX
func = funcY // 报错
```

在 TypeScript 中，我们使用数组的forEach方法的时候，

我们传入的回调函数可以传 1~3 个参数，如果传多了就会报错，

这是使用回调函数赋值给forEach方法内部使用的案列

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

arr.forEach((v, i, a) => {
  console.log(v)
})

arr.forEach((v) => {
  console.log(v)
})
```

名称与类型兼容

参数的名称没必要是相同的，只要与对应的参数类型一致就行，所以要确保参数的类型一致才能赋值

```
let funcX = (n: number) => 0
let funcY = (num: number, str: string) => 0

funcY = funcX
```

可选参数兼容

被赋值变量上有额外的可选参数不会出错，赋值变量的可选参数在被复制变量里没有对应的参数也不会出错

```
let funcX = (ant: number, address?: string, target?: string) => 0
let funcY = (num: number, str?: string) => 0

funcY = funcX // funcX虽然有两个额外参数，但它们都是可选的，所以不会报错
```

参数双向协变兼容

默认是兼容的，可以开启严格模式使其不兼容参数双向协变

```
let funcX = (num: string | number) => 0
let funcY = (num: number) => 0

// 在使用时默认是原来创建函数的类型，funcY只能传入number类型，而funcX是可以接受string和number类型的
funcY = funcX
funcX = funcY
```

返回值类型兼容

函数的返回值同新版的参数双向协变一样，需要源函数的返回值需要能够赋值给目标函数返回值类型

```
let funcX = (): string | number => 0
let funcY = (): number => 0

funcX = funcY
funY = funcX // 报错
```

如果目标函数的返回值类型是 void，那么源函数返回值可以是任意类型

```
let funcX = (): void => {}
let funcY = (): number => 0

funcX = funcY // 不会报错
```

函数重载

对于有重载的函数，源函数的每个重载都要在目标函数上找到对应的函数签名，规则同上

```
function merge(arg1: number, arg2: number): number
function merge(arg1: string, arg2: string): string
function merge(arg1: any, arg2: any): any {
  return arg1 + arg2
}

function sum(arg1: number, arg2: number): number
function sum(arg1: any, arg2: any): any {
  return arg1 + arg2
}

let func = merge
func = sum // 报错
```

枚举兼容性

枚举的兼容性体现在除了可以被赋值为枚举成员，

还能够被赋值给纯数字（必须要枚举成员中有数字类型的成员存在），

但是不能被赋值为其他的枚举变量或者字符串。

```
enum Status {
  On,
  Off
}

enum Animal {
  Dog,
  Cat
}

let s = Status.On
s = 1 // 不会报错
s = Animal.Cat // 报错
s = '1' // 报错
```

类兼容性

类和接口的兼容性非常类似，但是类分实例部分和静态部分。

比较两个类类型数据时，只有实例成员会被比较，静态成员和构造函数不会比较。

```
class Animal {
  public static age: number
```

```
  constructor(public name: string) {}
}

class People {
  public static age: string
  constructor(public name: string) {}
}

class Food {
  constructor(public name: number) {}
}

let animal: Animal = new People('张三') // 正确，只会比较实例属性方法
let animal2: Animal = new Food('食物') // 报错，因为实例成员类型不正确
```

实例属性方法比较时不会检验多余的属性和方法

```
class Animal {
  public static age: number

  constructor(public name: string) {}
}

class People {
  public static age: string
  public gender: string = '男' // 多余的属性
  constructor(public name: string) {}
}

let animal: Animal = new People('张三') // 不会报错
```

类私有成员兼容

私有成员会影响兼容性判断，

如果目标类型包含一个私有成员（或受保护类型），那么源类型必须包含来自同一个类的这个私有成员。

允许子类赋值给父类，但是不能赋值给其它有同样类型的类

```
class Parent {
  constructor(private age: number) {}
}

class Child extends Parent {
  constructor(age: number) {
    super(age)
  }
}

class Other {
  constructor(private age: number) {}
}

let c: Parent = new Child(18)
```

```
let other: Parent = new Other(18) // 报错
// 上面的private换成protected也一样
```

泛型兼容性

泛型的类型参数影响数据的成员，即使进行了泛型的限定，

如果没有真正的使用，对数据没有任何影响的

```
interface Empty<T> {}
let obj: Empty<number> = {} // 不会报错
```

所以能够这样：

```
interface Empty<T> {}
let x: Empty<number>
let y: Empty<string> = {}
x = y
```

但是如果在内部使用了，给了成员，就无法赋值了

```
interface Data<T> {
  data: T
}
let x: Data<number>
let y: Data<string> = { data: 'str' }

x = y // 报错
```

命名空间

命名空间是为了解决重名问题

index.ts

```
namespace SomeNameSpaceName {
  const q = {}

  export interface obj {
    name: string
  }
}
```

定义好一个命名空间，可以看到变量 q 没有写 export 关键字，这证明它是内部的变量，

就算别的 .ts 文件引入 index.ts，它也不会暴露出去。而 interface 这个 obj 接口是可以被全局访问的。

在别的文件引入并访问当前命名空间

1. reference 引入

```
/// <reference path="./index.ts" />
namespace SomeNameSpaceName {
    export class person implements obj {
        name: "前端"
    }
}
```

2. import 引入

```
export interface valueData {
  name: string
}
```

```
import { valueData } from "./xxx.ts"
```

声明

声明语句

假如我们想使用第三方库 jQuery，

一种常见的方式是在 html 中通过 `<script>` 标签引入 jQuery，

然后就可以使用全局变量 $ 或 jQuery 了。

我们通常这样获取一个 id 是 foo 的元素：

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

但是在 ts 中，编译器并不知道 $ 或 jQuery 是什么东西：

```
jQuery('#foo');
// ERROR: Cannot find name 'jQuery'.
```

这时，我们需要使用 declare var 来定义它的类型：

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

jQuery('#foo');
```

与其类似的，还有 declare let 和 declare const，使用 let 与使用 var 没有什么区别

上例中，declare var 并没有真的定义一个变量，

只是定义了全局变量 jQuery 的类型，仅仅会用于编译时的检查，在编译结果中会被删除。

它的编译结果是：

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

需要注意的是，声明语句中只能定义类型，切勿在声明语句中定义具体的实现：

```
declare const jQuery = function(selector) {
    return document.querySelector(selector);
};
// ERROR: An implementation cannot be declared in ambient contexts.
```

除了 declare var 之外，还有其他很多种声明语句：

- declare function 声明全局方法
- declare class 声明全局类
- declare enum 声明全局枚举类型
- declare namespace 声明（含有子属性的）全局对象

声明文件

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

```
// src/jQuery.d.ts
declare var jQuery: (selector: string) => any;
```

```
// src/index.ts
jQuery('#foo');
```

声明文件必需以 .d.ts 为后缀。

一般来说，ts 会解析项目中所有的 *.ts 文件，当然也包含以 .d.ts 结尾的文件。

所以当我们将 jQuery.d.ts 放到项目中时，其他所有 *.ts 文件就都可以获得 jQuery 的类型定义了

这里只演示了全局变量这种模式的声明文件，

假如是通过模块导入的方式使用第三方库的话，那么引入声明文件又是另一种方式了

声明合并

如果定义了两个相同名字的函数、接口或类，那么它们会合并成一个类型：

函数的合并

之前学习过，我们可以使用重载定义多个函数类型：

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

接口的合并

接口中的属性在合并时会简单的合并到一个接口中：

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

相当于：

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

注意，**合并的属性的类型必须是唯一的**：

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

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

接口中方法的合并，与函数的合并一样：

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

相当于：

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

类的合并

类的合并与接口的合并规则一致。

tsconfig.json

tsconfig.json 是配置如何将 ts 文件编译成 js 的文件。

tsc --init 这个命令会生成该文件。

tsconfig.json 是 TypeScript 项目的配置文件。

如果一个目录下存在一个 tsconfig.json 文件，那么往往意味着这个目录就是 TypeScript 项目的根目录。

tsconfig.json 包含 TypeScript 编译的相关配置，

通过更改编译配置项，我们可以让 TypeScript 编译出 ES6、ES5、node 的代码。

tsconfig.json 的作用

- 用于标识 TypeScript 项目的根路径；
- 用于配置 TypeScript 编译器；
- 用于指定编译的文件。

tsconfig.json 重要字段

- files - 设置要编译的文件的名称；
- include - 设置需要进行编译的文件，支持路径模式匹配；
- exclude - 设置无需进行编译的文件，支持路径模式匹配；
- compilerOptions - 设置与编译流程相关的选项。

compilerOptions 选项

compilerOptions 支持很多选项，常见的有 baseUrl、 target、baseUrl、 moduleResolution 和 lib 等。

compilerOptions 每个选项的详细说明如下：

```
{
  "compilerOptions": {

    /* 基本选项 */
    "target": "es5",                       // 指定 ECMAScript 目标版本: 'ES3'
(default), 'ES5', 'ES6'/'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'
    "module": "commonjs",                  // 指定使用模块: 'commonjs', 'amd',
'system', 'umd' or 'es2015'
    "lib": [],                             // 指定要包含在编译中的库文件
```

```
    "allowJs": true,                       // 允许编译 javascript 文件
    "checkJs": true,                       // 报告 javascript 文件中的错误
    "jsx": "preserve",                     // 指定 jsx 代码的生成: 'preserve',
'react-native', or 'react'
    "declaration": true,                   // 生成相应的 '.d.ts' 文件
    "sourceMap": true,                     // 生成相应的 '.map' 文件
    "outFile": "./",                       // 将输出文件合并为一个文件
    "outDir": "./",                        // 指定输出目录
    "rootDir": "./",                       // 用来控制输出目录结构 --outDir.
    "removeComments": true,                // 删除编译后的所有的注释
    "noEmit": true,                        // 不生成输出文件
    "importHelpers": true,                 // 从 tslib 导入辅助工具函数
    "isolatedModules": true,               // 将每个文件做为单独的模块 （与
'ts.transpileModule' 类似）.

    /* 严格的类型检查选项 */
    "strict": true,                        // 启用所有严格类型检查选项
    "noImplicitAny": true,                 // 在表达式和声明上有隐含的 any类型时报错
    "strictNullChecks": true,              // 启用严格的 null 检查
    "noImplicitThis": true,                // 当 this 表达式值为 any 类型的时候，生成
一个错误
    "alwaysStrict": true,                  // 以严格模式检查每个模块，并在每个文件里加入
'use strict'

    /* 额外的检查 */
    "noUnusedLocals": true,                // 有未使用的变量时，抛出错误
    "noUnusedParameters": true,            // 有未使用的参数时，抛出错误
    "noImplicitReturns": true,             // 并不是所有函数里的代码都有返回值时，抛出错
误
    "noFallthroughCasesInSwitch": true,    // 报告 switch 语句的 fallthrough 错误。
（即，不允许 switch 的 case 语句贯穿）

    /* 模块解析选项 */
    "moduleResolution": "node",            // 选择模块解析策略： 'node' (Node.js) or
'classic' (TypeScript pre-1.6)
    "baseUrl": "./",                       // 用于解析非相对模块名称的基目录
    "paths": {},                           // 模块名到基于 baseUrl 的路径映射的列表
    "rootDirs": [],                        // 根文件夹列表，其组合内容表示项目运行时的结
构内容
    "typeRoots": [],                       // 包含类型声明的文件列表
    "types": [],                           // 需要包含的类型声明文件名列表
    "allowSyntheticDefaultImports": true,  // 允许从没有设置默认导出的模块中默认导入。

    /* Source Map Options */
    "sourceRoot": "./",                    // 指定调试器应该找到 TypeScript 文件而不
是源文件的位置
    "mapRoot": "./",                       // 指定调试器应该找到映射文件而不是生成文件的
位置
    "inlineSourceMap": true,               // 生成单个 soucemaps 文件，而不是将
sourcemaps 生成不同的文件
    "inlineSources": true,                 // 将代码与 sourcemaps 生成到一个文件中，
要求同时设置了 --inlineSourceMap 或 --sourceMap 属性

    /* 其他选项 */
    "experimentalDecorators": true,        // 启用装饰器
```

```
    "emitDecoratorMetadata": true           // 为装饰器提供元数据的支持
  }
}
```

装饰器

开启对装饰器的支持，命令行编译文件时：

```
tsc --target ES5 --experimentalDecorators test.ts
```

配置文件 tsconfig.json

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

在一个末日生存游戏里，你有一个安全屋和一张床，每次睡觉都可以恢复一定的精力，但是初始的床什么都没有，每天睡在木板上只能恢复20点精力，某天你出外搜寻了材料并制作了一个床垫，将床垫放在床上，每次睡觉可以额外恢复20点精力，之后你又找到了被絮和三件套，甚至还给床进行改装增加了按摩功能，每次睡觉可以额外恢复的精力越来越多了...

在这个例子中，对于使用床的人而言，“睡觉”这个动作没有发生改变，床还是那张床，只不过我们通过床垫、被絮、按摩机增加了额外的功能。

在程序设计中，它们就可以分别以装饰器的形式进行设计，再通过组合使床拥有全部的装饰特征。

装饰者模式（Decorator Pattern）也称为装饰器模式，在不改变对象自身的基础上，动态增加额外的职责。

属于结构型模式的一种。

使用装饰者模式的优点：把对象核心职责和要装饰的功能分开了。非侵入式的行为修改。

举个例子来说，

原本长相一般的女孩，借助美颜功能，也能拍出逆天的颜值。

只要善于运用辅助的装饰功能，开启瘦脸，增大眼睛，来点磨皮后，咔嚓一拍，惊艳无比。

经过这一系列叠加的装饰，你还是你，长相不增不减，却能在镜头前增加了多重美。

如果你愿意，还可以尝试不同的装饰风格，只要装饰功能做的好，你就能成为“百变星君”。

```
let girl = {
  faceValue() {
    console.log('我原本的脸')
  }
}

function thinFace() {
  console.log('开启瘦脸')
}
```

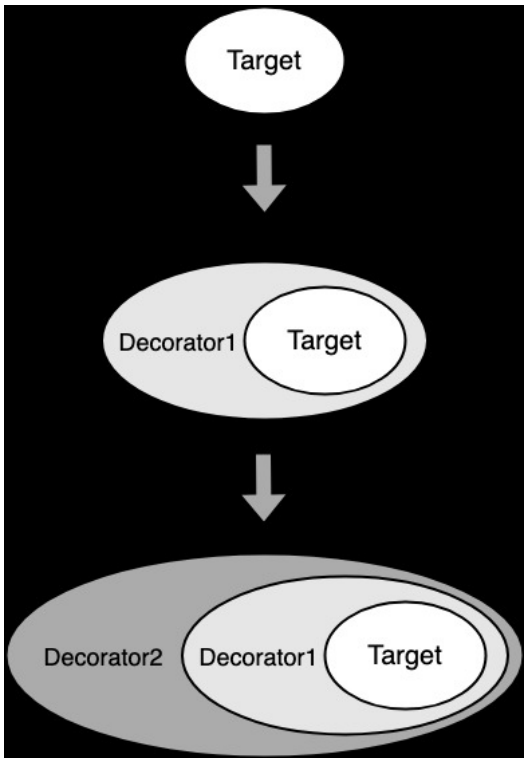
```
function IncreasingEyes() {
  console.log('增大眼睛')
}

girl.faceValue = function(){
  const originalFaveValue = girl.faceValue;  // 原来的功能
  return function() {
    originalFaveValue.call(girl);
    thinFace.call(girl);
  }
}()

girl.faceValue = function(){
  const originalFaveValue = girl.faceValue;  // 原来的功能
  return function() {
    originalFaveValue.call(girl);
    IncreasingEyes.call(girl);
  }
}()

girl.faceValue();
// 我原本的脸
// 开启瘦脸
// 增大眼睛
```

在不改变原来代码的基础上，通过先保留原来函数，重新改写，在重写的代码中调用原来保留的函数。

```
var plane = {
  fire: function(){
    console.log( '发射普通子弹' );
  }
}

var missileDecorator = function(){
  console.log( '发射导弹' );
}

var atomDecorator = function () {
  console.log('发射原子弹');
}

var fire1 = plane.fire;

// 相当于
// var fire2 = function () {
//   console.log( '发射普通子弹' );
//   missileDecorator();
// }
plane.fire = function () {
  fire1();
  missileDecorator();
}

// 相当于
// var fire2 = function () {
```

```
//   console.log( '发射普通子弹' );
//   missileDecorator();
// }
var fire2 = plane.fire;

// 相当于
// plane.fire = function () {
//   console.log( '发射普通子弹' );
//   missileDecorator();
//   atomDecorator();
// }
plane.fire = function () {
  fire2();
  atomDecorator();
}

plane.fire();
// 分别输出： 发射普通子弹、发射导弹、发射原子弹
```

用一张图来表示装饰者模式的原理：



从图中可以看出来，通过一层层的包装，增加了原先对象的功能。

装饰器是什么

装饰器实际上就是一个函数，在使用时前面加上 @ 符号，写在要装饰的声明之前。

TypeScript 中的装饰器使用 @expression 这种形式，expression 求值后为一个函数，

它在运行时被调用，被装饰的声明信息会作为参数传入。

多个装饰器同时作用在一个声明时，可以写一行或换行写：

```
// 换行写
@test1
@test2
declaration

//写一行
@test1 @test2 ...
declaration
```

装饰器的分类

参数装饰器（Parameter decorators）

方法装饰器（Method decorators）

访问器装饰器（Accessor decorators）

属性装饰器（Property decorators）

类装饰器（Class decorators）

装饰器	参数装饰器	方法装饰器	访问器装饰器	属性装饰器	类装饰器
位置	bar(@foo para: string) {}	@foo public bar() {}	@foo get bar()	@foo() bar: number	@foo class Bar {}
传入参数	target, propertyKey, parameterIndex	target, propertyKey, descriptor	target, propertyKey, descriptor	target, propertyKey	constructor
返回值	返回值被用作方法的属性描述符	返回值被用作方法的属性描述符	被忽略	被忽略	用返回值提供的构造函数来替换类的声明

参数释义：

- `constructor`：类构造函数
- `target`：被装饰的对象，对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
- `propertyKey`：被装饰的属性，成员的名字
- `descriptor`：属性的描述，成员的属性描述符
- `parameterIndex`：参数在函数参数列表中的索引

```
// 类装饰器
@classDecorator
class Bird {
  // 属性装饰器
  @propertyDecorator
  name: string;

  // 方法装饰器
  @methodDecorator
```

```
  fly(
    // 参数装饰器
    @parameterDecorator
    meters: number
  ) {}

  // 访问器装饰器
  @accessorDecorator
  get egg() {}
}
```

装饰器的使用

定义 face.ts 文件：

```
function thinFace() {
  console.log('开启瘦脸')
}

@thinFace
class Girl {
}
```

编译成 js 代码，在运行时，会直接调用 thinFace 函数。这个装饰器作用在类上，称之为类装饰器。

如果需要附加多个功能，可以组合多个装饰器一起使用：

```
function thinFace() {
  console.log('开启瘦脸')
}

function IncreasingEyes() {
  console.log('增大眼睛')
}

@thinFace
@IncreasingEyes
class Girl {
}
```

多个装饰器组合在一起，在运行时，要注意，调用顺序是 从下至上 依次调用，正好和书写的顺序相反。

例子中给出的运行结果是：

```
'增大眼睛'
'开启瘦脸'
```

下面代码中，smile 是一个装饰器，`@smile` 语法规定了 greet 方法将使用 smile 装饰器

装饰器 smile 本身其实是一个函数，

它接收 target（被装饰的对象），propertyKey（被装饰的属性）和 descriptor（属性的描述）作为参数，

本例中，

target 表示 Greeter 的原型对象，即 Greeter.prototype，

propertyKey 是 "greet" ，

descriptor 是Greeter.prototype.greet 的属性描述对象，类似下面这样：

```
{
  value: [Function],
  writable: true,
  enumerable: true,
  configurable: true
}
```

smile 方法本身没有返回任何值，只是执行简单的打印 "smile" 的功能

```
function smile(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  console.log('smile');
}

class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  @smile
  greet(name: string): string {
    console.log(`welcome, ${name}!`);
    return "Hello";
  }
}

const g = new Greeter('msg');

g.greet('tom');
// "smile"
// "welcome, tom!"
```

装饰器工厂

通过装饰器工厂方法，可以额外传参，普通装饰器无法传参。

有时需要给装饰器传递一些参数，这要借助于装饰器工厂函数。

装饰器工厂函数实际上就是一个高阶函数，在调用后返回一个函数，返回的函数作为装饰器函数。

```
function thinFace(value: string){
  console.log('1-瘦脸工厂方法')
  return function(){
    console.log(`4-我是瘦脸的装饰器，要瘦脸${value}`)
  }
}
```

```
function IncreasingEyes(value: string) {
  console.log('2-增大眼睛工厂方法')
  return function(){
    console.log(`3-我是增大眼睛的装饰器，要${value}`)
  }
}

@thinFace('50%')
@IncreasingEyes('增大一倍')
class Girl {
}
```

调用工厂函数是依次从上到下执行，装饰器函数的运行顺序依然是从下到上依次执行。

运行结果：

```
1-瘦脸工厂方法
2-增大眼睛工厂方法
3-我是增大眼睛的装饰器，要增大一倍
4-我是瘦脸的装饰器，要瘦脸50%
```

根据不同场景确定"smile"的打印次数，

greet方法执行的时候需要打印3次smile，

打印的次数可以作为参数传递给smile装饰器灵活控制

```
function smile(times: number) {
  return function(
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    for (let i = 0; i < times; i++) {
      console.log('smile');
    }
  }
}

class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  @smile(3)
  greet(name: string): string {
    console.log(`welcome, ${name}!`);
    return "Hello";
  }
}

const g = new Greeter('msg');

g.greet('tom');
```

```
// "smile"
// "smile"
// "smile"
// "welcome, tom!"
```

经过上面的改造，smile 装饰器需接收 1 个参数，

通过 @smile(3) 的形式使用装饰器即可在调用 greet 方法时打印3 次"smile"。

类装饰器

作用在类声明上的装饰器，可以给我们改变类的机会，通过类装饰器扩展类的属性和方法。

下面代码中，

activate 装饰器的 target 参数指代 Greeter 构造器，该装饰器修改了 Greeter 类的 static 属性 active 的值

```
function activate(target: any) {
  target.active = true; // target 指代 Greeter 函数
}

@activate
class Greeter {
  static active = false;
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet(name: string): string {
    console.log(`welcome, ${name}!`);
    return "Hello";
  }
}

console.log(Greeter.active); // true
```

在执行装饰器函数时，会把类构造函数传递给装饰器函数。

```
function extension<T extends { new(...args:any[]): {} }>(constructor: T) {
  // 重载构造函数
  return class extends constructor {
    // 扩展属性
    public coreHour = '10:00-15:00'
    // 函数重载
    meeting() {
      console.log('重载：Daily meeting!')
    }
  }
}

@extension
class Employee {
  public name!: string
  public department!: string
```

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

  meeting() {
    console.log('Every Monday!')
  }

}

let e = new Employee('Tom', 'IT')
console.log(e) // Employee { name: 'Tom', department: 'IT', coreHour: '10:00-15:00' }
e.meeting()    // 重载：Daily meeting!
```

```
const extension = (constructor: Function) => {
  constructor.prototype.coreHour = '10:00-15:00'

  constructor.prototype.meeting = () => {
    console.log('重载：Daily meeting!');
  }
}

@extension
class Employee {
  public name!: string
  public department!: string

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

  meeting() {
    console.log('Every Monday!')
  }
}

let e: any = new Employee('Tom', 'IT')
console.log(e.coreHour) // 10:00-15:00
e.meeting()             // 重载：Daily meeting!
```

方法装饰器

```
 function log(
      target: Object,
      propertyName: string,
      descriptor: TypedPropertyDescriptor<(...args: any[]) => any>
    ) {
      const method = descriptor.value;
```

```
      descriptor.value = function(...args: any[]) {
        // 将参数转为字符串
        const params: string = args.map(a => JSON.stringify(a)).join();

        const result = method!.apply(this, args);

        // 将结果转为字符串
        const resultString: string = JSON.stringify(result);

        console.log(`Call:${propertyName}(${params}) => ${resultString}`);

        return result;
      };
    }

class Author {
  constructor(private firstName: string, private lastName: string) {}

  @log
  say(message: string): string {
    return `${message} by: ${this.lastName}${this.firstName}`;
  }
}

const author:Author = new Author('Lei','Li');
author.say('《Typescript》');// Call:say("《Typescript》") => "《Typescript》
by:LiLei"
```

```
function foo() {
    console.log("foo(): evaluated");
    return function (target: Object, propertyKey: string, descriptor:
PropertyDescriptor) {
        console.log("foo(): called");
    }
}

function bar() {
    console.log("bar(): evaluated");
    return function (target: Object, propertyKey: string, descriptor:
PropertyDescriptor) {
        console.log("bar(): called");
    }
}

class C {
    @foo()
    @bar()
    method() {}
}
// => foo(): evaluated
// => bar(): evaluated
// => bar(): called
// => foo(): called
```

装饰器的执行优先级

实例成员： 参数装饰器 -> 方法 / 访问器 / 属性 装饰器 —— 优先级 1

静态成员：参数装饰器 -> 方法 / 访问器 / 属性 装饰器 —— 优先级 2

构造函数: 参数装饰器 —— 优先级 3

类装饰器 —— 优先级 4

```
function f(key: string): any {
  console.log("evaluate: ", key);
  return function () {
    console.log("call: ", key);
  };
}

@f("Class Decorator")
class C {
  @f("Static Property")
  static prop?: number;

  @f("Static Method")
  static method(@f("Static Method Parameter") foo) {}

  constructor(@f("Constructor Parameter") foo) {}

  @f("Instance Method")
  method(@f("Instance Method Parameter") foo) {}

  @f("Instance Property")
  prop?: number;
}
```

打印如下

```
evaluate:  Instance Method
evaluate:  Instance Method Parameter
call:  Instance Method Parameter
call:  Instance Method
evaluate:  Instance Property
call:  Instance Property
evaluate:  Static Property
call:  Static Property
evaluate:  Static Method
evaluate:  Static Method Parameter
call:  Static Method Parameter
call:  Static Method
evaluate:  Class Decorator
evaluate:  Constructor Parameter
call:  Constructor Parameter
call:  Class Decorator
```

执行实例属性 prop 晚于实例方法 method， 执行静态属性 static prop 早于静态方法 static method 。

对于方法/访问器/属性装饰器而言，执行顺序取决于声明它们的顺序。