河南中医药大学信息技术学院（智能医疗行业学院）智能医学工程专业《互联网医疗服务开发》课程

第07章：TypeScript基础

冯顺磊

河南中医药大学信息技术学院（智能医疗行业学院）
河南中医药大学信息技术学院智能医疗教研室
https://aitcm.hactcm.edu.cn
2025/3/21

1

本章概要

- 概述
- 数据类型
- 函数
- 接口
- 类和对象

2

1. 概述 1.1 TypeScript简介

- TypeScript 是由微软开发和维护的开源编程语言，发布于 2012 年 10 月。它是 JavaScript 的一个超集，这意味着所有合法的 JavaScript 代码都可以直接在 TypeScript 中使用，同时 TypeScript 还添加了一些新的特性来增强 JavaScript 的功能。

- 主要特性
  - 静态类型系统：TypeScript 最核心的特性之一。开发者可以为变量、函数参数、返回值等显式指定类型，如 number、string、boolean 等基本类型，也可以自定义复杂类型。

```
let age: number = 25;
function greet(name: string): string {
    return `Hello, ${name}!`;
}
```

  - 静态类型检查有助于在编译阶段发现类型相关的错误，避免在运行时出现因类型不匹配导致的问题，提高代码的可靠性和可维护性。

1. 概述 1.1 TypeScript简介

- 面向对象编程支持
  - 类和继承：TypeScript 支持类的定义和继承，允许开发者使用面向对象的编程范式来组织代码。

```
class Animal {
    constructor(public name: string) {}
    speak() {
        console.log(`${this.name} makes a sound.`);
    }
}

class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}

const myDog = new Dog('Buddy');
myDog.speak();
```

1. 概述 1.1 TypeScript简介

- 面向对象编程支持
  - 接口：接口用于定义对象的结构和类型约束，使代码更加清晰和可维护。

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

function introduce(person: Person) {
    console.log(`My name is ${person.name} and I'm ${person.age} years old.`);
}
```


1. 概述 1.1 TypeScript简介

- 面向对象编程支持
  - 枚举类型：枚举是 TypeScript 新增的一种数据类型，用于定义一组命名的常量。

```typescript
enum Color {
    Red,
    Green,
    Blue
}

let myColor: Color = Color.Green;
```

  - 编译时错误检查：TypeScript 代码在编译阶段会进行严格的类型检查，如果代码中存在类型错误或不符合语法规则的地方，编译器会抛出错误，帮助开发者及时发现并修复问题。

1. 概述 1.1 TypeScript简介

- 应用场景
  - 大型 Web 应用开发：对于复杂的单页应用（SPA），如使用 Angular、Vue.js 或 React 框架开发的项目，TypeScript 可以帮助开发者更好地管理代码，提高代码的可维护性和可扩展性。
  - 后端开发：基于 Node.js 的后端应用也可以使用 TypeScript 编写，如 Express 或 Nest.js 框架，利用其类型系统和面向对象特性构建更健壮的服务器端应用。
  - 跨平台开发：在使用 React Native 开发移动应用、Electron 开发桌面应用等场景中，TypeScript 同样可以发挥重要作用，提升开发效率和代码质量。

1. 概述 1.2 与JavaScript的异同

- 相同点
  - 语法基础：TypeScript 继承了 JavaScript 的语法，包括变量声明、函数定义、控制流语句（如 if、for、while 等）、对象和数组的使用等。因此，熟悉 JavaScript 的开发者可以很容易上手 TypeScript。
  - 运行环境：TypeScript 代码最终会被编译成纯 JavaScript 代码，所以它可以在任何支持 JavaScript 的环境中运行，如浏览器、Node.js 服务器等。
  - 生态系统：由于 TypeScript 是 JavaScript 的超集，它可以直接使用现有的 JavaScript 库和框架，同时也有许多为 TypeScript 专门设计的库和工具。

1. 概述 1.2 与JavaScript的异同

- 不同点
  - 类型系统
    - JavaScript 是一种动态类型语言，变量的类型在运行时确定，开发过程中可以随时改变变量的类型。
    - TypeScript 是静态类型语言，变量在声明时就需要指定类型，并且在编译阶段会进行类型检查，不允许类型不匹配的赋值操作。
  - 代码复杂度和开发效率
    - JavaScript 代码相对简洁，开发速度快，适合快速迭代和小型项目。但在大型项目中，由于缺乏类型检查，代码的可维护性和可扩展性可能会受到影响。
    - TypeScript 虽然增加了一些类型声明的代码，但在大型项目中可以提高代码的可读性和可维护性，减少调试时间，提高开发效率。同时，编辑器的智能提示和自动补全功能在 TypeScript 中更加强大。
  - 编译过程
    - JavaScript 是解释型语言，代码可以直接在浏览器或 Node.js 环境中运行，无需编译。
    - TypeScript 需要通过编译器（如 tsc）将 .ts 文件编译成 .js 文件，然后才能在目标环境中运行。编译过程可以发现潜在的错误，但也增加了一定的开发流程。

9

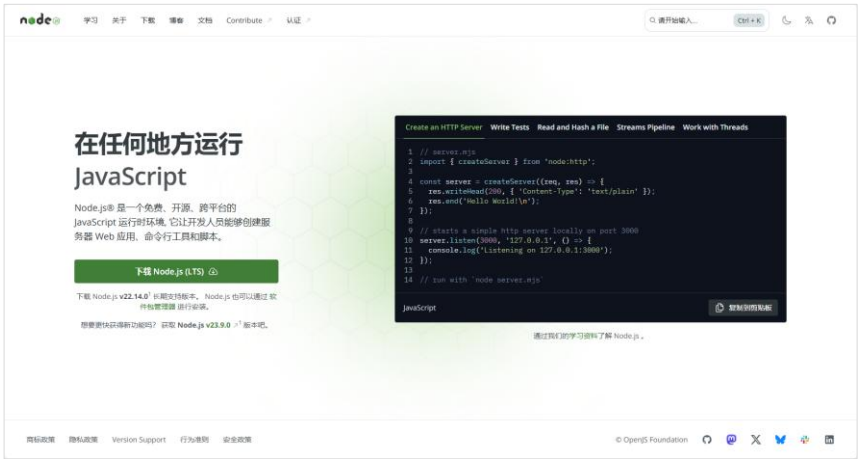
1. 概述 1.2 与JavaScript的异同

- 选用TypeScript的原因主要如下：
  - 能够更早的发现代码中的错误
  - 能够帮助提高生产力
  - 支持JavaScript语言最新特性并且使用与JavaScript语言相同的语法和语义。

10

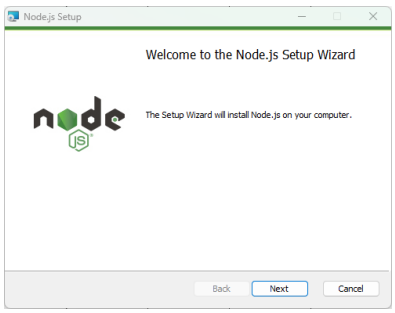
1. 概述 1.3 TypeScript安装

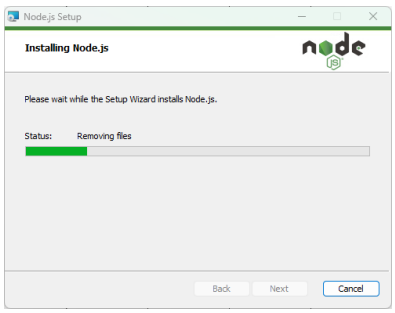
- 安装Node.js
  - 下载地址：https://nodejs.org/zh-cn

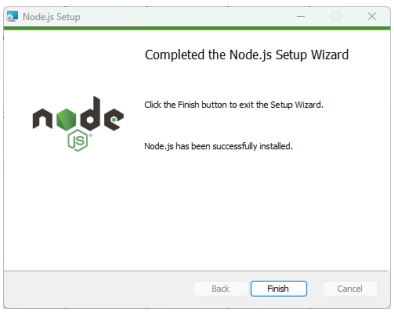


1. 概述 1.3 TypeScript安装

- 安装Node.js
  - 根据提示逐步完成安装

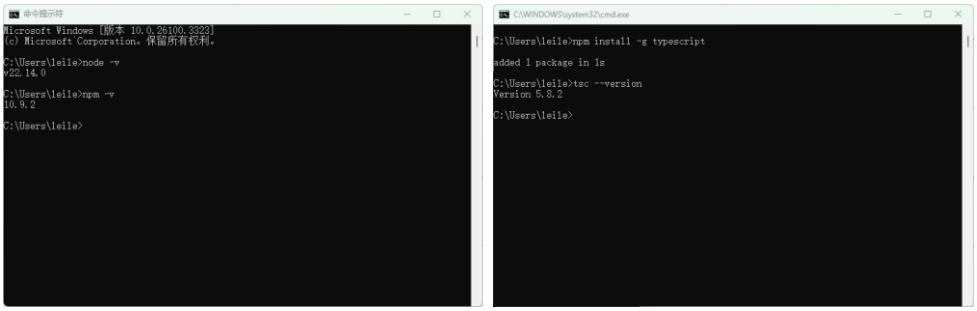






1. 概述 1.3 TypeScript安装

- 确认 Node.js 和 npm 已安装
- 使用 npm 全局安装 TypeScript，并验证安装结果



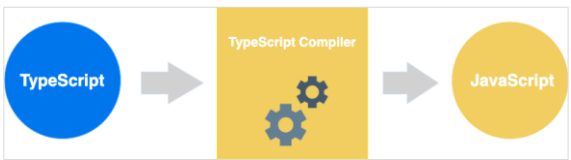
1. 概述 1.3 TypeScript安装

- 运行TypeScript代码

```
var message:string = "Hello World"
console.log(message);
```

tsc .\7-1-1.ts

```
var message = "Hello World";
console.log(message);
```



示例7-1-1：运行TypeScript代码

2. 数据类型 2.1 基础数据类型

- JavaScript 语言（注意，不是 TypeScript）将值分成8种类型。
  - boolean
  - string
  - number
  - bigint
  - symbol
  - object
  - undefined
  - null
- TypeScript 继承了 JavaScript 的类型设计，以上8种类型可以看作 TypeScript 的基本类型。
- 注意，上面所有类型的名称都是小写字母，首字母大写的Number、String、Boolean等在 JavaScript 语言中都是内置对象，而不是类型名称。
- undefined 和 null 既可以作为值，也可以作为类型，取决于在哪里使用它们。
- 这8种基本类型是 TypeScript 类型系统的基础，复杂类型由它们组合而成。

2. 数据类型 2.2 数组（array）

- 在 TypeScript 中，数组是一种常用的数据结构，用于存储多个相同类型或不同类型的值。TypeScript 在 JavaScript 数组的基础上，增加了类型系统，使得数组的使用更加安全和可靠。
- 数组类型的定义
  - 元素类型后面跟方括号：这是最常见的定义数组类型的方式，语法为 元素类型[]。

```
// 定义一个存储数字的数组
let numbers: number[] = [1, 2, 3, 4, 5];

// 定义一个存储字符串的数组
let names: string[] = ['Alice', 'Bob', 'Charlie'];
```

  - 使用泛型数组类型：使用 Array<元素类型> 的语法来定义数组类型。

```
// 定义一个存储布尔值的数组
let flags: Array<boolean> = [true, false, true];
```

2. 数据类型 2.2 数组（array）

- 数组的操作方法：TypeScript 中的数组可以使用 JavaScript 数组的所有方法，例如 push、pop、splice 等。

```
let fruits: string[] = ['apple', 'banana', 'cherry'];

// 添加元素到数组末尾
fruits.push('date');
console.log(fruits);

// 移除数组末尾的元素
let lastFruit = fruits.pop();
console.log(lastFruit);
console.log(fruits);

// 插入元素到指定位置
fruits.splice(1, 0, 'elderberry');
console.log(fruits);
```


2. 数据类型 2.2 数组（array）

- 只读数组：使用 readonly 关键字可以定义只读数组，一旦数组被定义为只读，就不能再对其进行修改操作。

```
// 定义一个只读数字数组
let readonlyNumbers: readonly number[] = [1, 2, 3];

// 下面这行代码会报错，因为只读数组不能进行修改操作
// readonlyNumbers.push(4);
```

示例7-2-1：数组

2. 数据类型 2.3 元组（tuple）

- 定义元组：使用方括号语法来定义元组，在方括号内依次列出每个元素的类型，元素类型之间用逗号分隔。

```
// 定义一个包含字符串和数字的元组
let person: [string, number];
// 正确赋值
person = ['John', 30];
// 错误赋值
person = [30,'John'];
```

- 访问元组元素：可以使用索引来访问元组中的元素，索引从 0 开始。

```
let person: [string, number] = ['John', 30];
// 访问第一个元素
console.log(person[0]);
// 访问第二个元素
console.log(person[1]);
```


2. 数据类型 2.3 元组（tuple）

- 元组的越界问题：TypeScript 会对元组的越界访问进行检查，当访问的索引超出元组定义的范围时，会在编译阶段报错。

```
let person: [string, number] = ['John', 30];
// 编译错误，因为元组只有两个元素，索引 2 越界了
// console.log(person[2]);
```

- 可选元素：元组中可以使用可选元素，通过在类型后面加 ？来表示。可选元素必须放在元组定义的末尾。

```
// 定义一个包含可选元素的元组
let employee: [string, number, boolean?];
// 可以只提供前两个元素
employee = ['Alice', 25];
// 也可以提供所有元素
employee = ['Bob', 35, true];
```

2. 数据类型 2.3 元组（tuple）

- 元组中可以使用剩余元素，使用 ... 语法来表示，它表示可以有任意数量的指定类型的元素。

```
// 定义一个包含剩余元素的元组
let numbers: [number, ...number[]];
// 正确赋值
numbers = [1, 2, 3, 4, 5];
```

- 元组的解构赋值：可以使用解构赋值语法将元组中的元素赋值给变量。

```
let person: [string, number] = ['John', 30];
let [name, age] = person;
console.log(name);
console.log(age);
```

示例7-2-2：元组

2. 数据类型 2.4 枚举（enum ）

- 枚举类型为一组数值赋予友好的名字，使代码更具语义化。比如，在表示一周的天数、颜色等场景下，使用枚举能让代码更易理解。

```
// 定义一个数字枚举
enum Direction {
    Up,
    Down,
    Left,
    Right
}

// 使用枚举成员
let moveUp = Direction.Up;
console.log(moveUp);
```

```
enum Direction {
    Up = 1,
    Down,
    Left,
    Right
}
```

- 在左侧代码中，Direction.Up 的值为 0，Direction.Down 的值为 1，依此类推。
- 在右侧代码中 ，Direction.Up 为 1，Direction.Down 为 2，Direction.Left 为 3，Direction.Right 为 4。

2. 数据类型 2.4 枚举（enum ）

- 字符串枚举

```
// 定义一个字符串枚举
enum Color {
    Red = "RED",
    Green = "GREEN",
    Blue = "BLUE"
}

// 使用字符串枚举成员
let favoriteColor = Color.Red;
console.log(favoriteColor);
```

  - 字符串枚举没有自动递增的行为，每个成员都需要明确赋值。

示例7-2-3：枚举

2. 数据类型 2.5 任意值（any）

- 实际开发中，any类型主要适用以下两个场合。
  - 出于特殊原因，需要关闭某些变量的类型检查，就可以把该变量的类型设为any。
  - 为了适配以前老的 JavaScript 项目，让代码快速迁移到 TypeScript，可以把变量类型设为any。有些年代很久的大型 JavaScript 项目，尤其是别人的代码，很难为每一行适配正确的类型，这时你为那些类型复杂的变量加上any，TypeScript 编译时就不会报错。
- 对于开发者没有指定类型、TypeScript 必须自己推断类型的那些变量，如果无法推断出类型，TypeScript 就会认为该变量的类型是any。
- any类型除了关闭类型检查，还有一个很大的问题，就是它会“污染”其他变量。它可以赋值给其他任何类型的变量（因为没有类型检查），导致其他变量出错。

2. 数据类型 2.5 任意值（any）

- any 类型表示任意类型，当你在编写代码时，不确定某个变量的具体类型，或者想绕过 TypeScript 的类型检查机制时，可以使用 any 类型。这在处理一些动态内容、从第三方库获取的数据或者在项目迁移过程中逐步引入类型检查时非常有用。

```
let x:any;

x = 1; // 正确
x = 'foo'; // 正确
x = true; // 正确
```

  - 变量类型一旦设为any，TypeScript 实际上会关闭这个变量的类型检查。即使有明显的类型错误，只要句法正确，都不会报错。
  - 由于这个原因，应该尽量避免使用any类型，否则就失去了使用 TypeScript 的意义。

2. 数据类型 2.6 unknown

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

```
let x:unknown;

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

2. 数据类型 2.6 unknown

- unknown类型跟any类型的不同之处在于，它不能直接使用，主要有以下几个限制。
  - unknown类型的变量，不能直接赋值给其他类型的变量（除了any类型和unknown类型）。

```
let v:unknown = 123;

let v1:boolean = v; // 报错
let v2:number = v; // 报错
```

  - 不能直接调用unknown类型变量的方法和属性。

```
let v1:unknown = { foo: 123 };
v1.foo  // 报错

let v2:unknown = 'hello';
v2.trim() // 报错

let v3:unknown = (n = 0) => n + 1;
v3() // 报错
```


2. 数据类型 2.6 unknown

- unknown类型跟any类型的不同之处在于，它不能直接使用，主要有以下几个限制。
  - unknown类型变量能够进行的运算是有限的，只能进行比较运算（运算符==、===、!=、!==、||、&&、?）、取反运算（运算符!）、typeof运算符和instanceof运算符这几种，其他运算都会报错。

```
let a:unknown = 1;

a + 1 // 报错
a === 1 // 正确
```

2. 数据类型 2.6 unknown

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

```
let a:unknown = 1;

if (typeof a === 'number') {
  let r = a + 10; // 正确
}
```

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

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

示例7-2-4：unknown类型

2. 数据类型 2.7 void

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

```
function warnUser(): void {
    alert("This is my warning message");
}
```

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

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

2. 数据类型 2.8 never

- never 类型表示那些永远不会出现的值的类型。换句话说，一个变量如果被定义为 never 类型，那么它永远不会有具体的值。
  - 是所有类型的子类型：这意味着 never 类型的值可以赋值给任何其他类型的变量。
  - 没有类型是 never 的子类型：除了 never 本身，没有其他类型的值可以赋值给 never 类型的变量。

```
function fn(x:string|number) {
  if (typeof x === 'string') {
    // ...
  } else if (typeof x === 'number') {
    // ...
  } else {
    x; // never 类型
  }
}
```


2. 数据类型 2.9 字面量类型

- 在 TypeScript 中，字面量类型是一种特殊的类型，它允许你将变量的类型限定为某个具体的字面量值，而不仅仅是宽泛的基本类型。这可以让你对变量的取值进行更精确的控制，提高代码的类型安全性和可读性。
  - 字符串字面量类型：字符串字面量类型允许你将变量的类型限定为特定的字符串值。

```
// 定义一个字符串字面量类型
type Direction = 'up' | 'down' | 'left' | 'right';

// 使用字符串字面量类型
let myDirection: Direction = 'up';
// 以下代码会报错，因为 'invalid' 不是 Direction 类型允许的值
// let wrongDirection: Direction = 'invalid';
```

  - 数字字面量类型：数字字面量类型可以将变量的类型限定为特定的数字值。

```
// 定义一个数字字面量类型
type Month = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12;

// 使用数字字面量类型
let currentMonth: Month = 5;
// 以下代码会报错，因为 13 不是 Month 类型允许的值
// let wrongMonth: Month = 13;
```

2. 数据类型 2.9 字面量类型

- 布尔字面量类型：布尔字面量类型将变量的类型限定为 true 或 false。

```
// 定义一个布尔字面量类型
type Answer = true | false;

// 使用布尔字面量类型
let isCorrect: Answer = true;
// 以下代码会报错，因为除了 true 和 false 外，其他值不被允许
// let wrongAnswer: Answer = 'yes';
```

- 字面量类型的应用场景
  - 函数参数约束：可以使用字面量类型来约束函数的参数，确保传入的值是预期的。

```
// 定义一个函数，参数使用字符串字面量类型
function setAlignment(alignment: 'left' | 'center' | 'right') {
    console.log(`Alignment set to: ${alignment}`);
}

// 正确调用
setAlignment('center');
// 以下代码会报错，因为 'top' 不是允许的参数值
// setAlignment('top');
```


2. 数据类型 2.9 字面量类型

- 字面量类型的应用场景
  - 状态管理：在管理对象的状态时，使用字面量类型可以明确规定状态的可能取值。

```
// 定义一个状态的字面量类型
type LoadingState = 'loading' | 'success' | 'error';

// 定义一个对象，其状态属性使用字面量类型
const request: { state: LoadingState } = { state: 'loading' };

// 根据状态执行不同操作
if (request.state === 'success') {
    console.log('Request succeeded!');
} else if (request.state === 'error') {
    console.log('Request failed!');
} else {
    console.log('Request is still loading...');
}
```

2. 数据类型 2.10 类型断言

- 在 TypeScript 里，类型断言是一种机制，它允许开发者手动告诉编译器某个变量的具体类型，绕过编译器的类型检查。
  - 尖括号语法

```
let someValue: any = "hello world";
let strLength: number = (<string>someValue).length;
```

  - as 语法

```
let someValue: any = "hello world";
let strLength: number = (someValue as string).length;
```

示例7-2-5：类型断言

2. 数据类型 2.11 联合类型

- 联合类型表示取值可以为多种类型中的一种，使用 | 分隔每个类型。

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

  - 函数参数：当函数的参数可以接受多种不同类型的值时，使用联合类型可以增强函数的灵活性。
  - 处理不同类型的数据结构：在处理从不同数据源获取的数据时，数据可能具有不同的结构，使用联合类型可以更好地应对这种情况。

2. 数据类型 2.12 交叉类型

- 交叉类型是将多个类型合并为一个类型。 这让我们可以把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，使用&定义交叉类型。

```
{
  type Useless = string & number;
}
```

  - 合并接口：当你需要创建一个同时具备多个接口特性的类型时，可以使用交叉类型。
  - 混入（Mixin）模式：交叉类型常用于实现混入模式，将多个类的特性合并到一个类中。

2. 数据类型 2.13 类型别名

- 类型别名用来给一个类型起个新名字。类型别名常用于联合类型。

```
type Message = string | string[];
let greet = (message: Message) => {
  // ...
};
```

3. 函数 3.1 变量声明

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

3. 函数 3.1 变量声明

- var声明
  - 作用域：var 声明的变量具有函数作用域，这意味着变量在整个函数内部都是可见的，而不是在块级作用域（如 if 语句、for 循环等代码块）内可见。

```
function testVar() {
    if (true) {
        var x = 10;
    }
    console.log(x); // 输出 10，因为 x 在函数内部都可见
}
testVar();
```

3. 函数 3.1 变量声明

- var声明
  - 变量提升：使用 var 声明的变量会发生变量提升，即变量可以在声明之前使用，但其值为 undefined。

```
console.log(y); // 输出 undefined
var y = 20;
```

  - 可重新赋值和重新声明：var 声明的变量可以被重新赋值，也可以在同一作用域内被重新声明。

```
var z = 30;
z = 40; // 可以重新赋值
var z = 50; // 可以重新声明
console.log(z); // 输出 50
```


3. 函数 3.1 变量声明

- let声明
  - 作用域：let 声明的变量具有块级作用域，这意味着变量只在声明它的代码块（如 if 语句、for 循环等）内可见。

```
function testLet() {
    if (true) {
        let a = 100;
        console.log(a); // 输出 100
    }
    // console.log(a); // 报错，a 在此处不可见
}
testLet();
```

3. 函数 3.1 变量声明

- let声明
  - 变量提升：虽然 let 声明的变量也会发生变量提升，但在变量声明之前访问它会导致 ReferenceError，这种现象被称为 “暂时性死区”（TDZ）。

```
// console.log(b); // 报错，b 处于暂时性死区
let b = 200;
```

  - 可重新赋值和重新声明：let 声明的变量可以被重新赋值，但不能在同一作用域内被重新声明。

```
let c = 300;
c = 400; // 可以重新赋值
// let c = 500; // 报错，不能在同一作用域内重新声明
```


3. 函数 3.1 变量声明

- const声明
  - 作用域：const 同样具有块级作用域，变量只在声明它的代码块内可见，这与 let 类似。

```
function testConst() {
    if (true) {
        const d = 1000;
        console.log(d); // 输出 1000
    }
    // console.log(d); // 报错，d 在此处不可见
}
testConst();
```

3. 函数 3.1 变量声明

- const声明
  - 变量提升：和 let 一样，const 声明的变量也存在暂时性死区，在声明之前访问会引发 ReferenceError。

```
// console.log(e); // 报错，e 处于暂时性死区
const e = 2000;
```

  - 可重新赋值和重新声明：const 声明的变量一旦赋值，就不能再重新赋值，并且也不能在同一作用域内重新声明。不过，如果 const 声明的是一个对象或数组，对象的属性或数组的元素是可以修改的。

```
const f = 3000;
// f = 4000; // 报错，不能重新赋值

const obj = { key: 'value' };
obj.key = 'new value'; // 可以修改对象的属性
```


3. 函数 3.1 变量声明

- 使用建议：在现代 TypeScript 开发中，推荐优先使用 let 和 const，因为它们的块级作用域和更严格的变量声明规则可以减少变量作用域混乱和意外赋值的问题，提高代码的可读性和可维护性。当变量的值不需要改变时，使用 const；当变量的值需要改变时，使用 let。
- 兼容性：let 和 const 是 ES6 引入的特性，在较旧的 JavaScript 环境中可能需要使用 Babel 等工具进行转换。但在 TypeScript 中，它们可以直接使用，因为 TypeScript 会将代码编译成兼容目标环境的 JavaScript 代码。

示例7-3-1：声明

3. 函数 3.2 函数定义

- 在 TypeScript 里，函数是构建程序的重要组成部分，它不仅继承了 JavaScript 函数的灵活性，还通过类型系统增强了代码的可靠性与可维护性。
  - 函数声明：这是最常见的定义函数的方式，使用 function 关键字，语法结构清晰，方便阅读和维护。

```typescript
function greet(name: string): string {
    return `Hello, ${name}!`;
}

const greeting = greet('John');
console.log(greeting);
```


3. 函数 3.2 函数定义

- 在 TypeScript 里，函数是构建程序的重要组成部分，它不仅继承了 JavaScript 函数的灵活性，还通过类型系统增强了代码的可靠性与可维护性。
  - 函数表达式：使用变量来存储函数，这种方式可以让函数作为值进行传递，增加了代码的灵活性。

```typescript
const add = function (a: number, b: number): number
{
    return a + b;
};

const result = add(3, 5);
console.log(result);
```

3. 函数 3.2 函数定义

- 在 TypeScript 里，函数是构建程序的重要组成部分，它不仅继承了 JavaScript 函数的灵活性，还通过类型系统增强了代码的可靠性与可维护性。
  - 函数表达式：还可以使用箭头函数来定义函数表达式，它的语法更加简洁，常用于简单的函数逻辑。

```typescript
const multiply = (a: number, b: number): number => a * b;

const product = multiply(4, 6);
console.log(product);
```


3. 函数 3.3 参数类型和返回值类型

- 参数类型：在 TypeScript 中，为函数的参数指定类型是其重要特性之一，这样可以在编译阶段就发现参数类型不匹配的问题。

```typescript
function printFullName(firstName: string, lastName:
string): void {
    console.log(`${firstName} ${lastName}`);
}

// 正确调用
printFullName('Alice', 'Smith');
// 错误调用，会在编译阶段报错
// printFullName(123, 'Smith');
```

3. 函数 3.3 参数类型和返回值类型

- 返回值类型：同样，为函数指定返回值类型可以明确函数的输出结果，提高代码的可读性和可维护性。如果函数没有返回值，返回值类型应指定为 void。

```
function getRandomNumber(): number {
    return Math.random();
}

const randomNum = getRandomNumber();
console.log(randomNum);

function showMessage(): void {
    console.log('This is a message.');
}
```


3. 函数 3.4 可选参数和默认参数

- 可选参数：当函数的某些参数不是必需的时候，可以使用可选参数。在参数名后面加上？来表示该参数是可选的，并且可选参数必须放在必选参数之后。

```
function createUser(name: string, age?: number): { name:
string; age?: number } {
    return { name, age };
}

const user1 = createUser('Bob');
const user2 = createUser('Eve', 25);
console.log(user1);
console.log(user2);
```

3. 函数 3.4 可选参数和默认参数

- 默认参数：为参数提供默认值，当调用函数时没有传入该参数，就会使用默认值。默认参数可以放在任意位置。

```
function setConfig(host: string = 'localhost', port: number
= 8080): void {
    console.log(`Host: ${host}, Port: ${port}`);
}

setConfig();
setConfig('example.com', 3000);
```


3. 函数 3.5 剩余参数

- 剩余参数：剩余参数允许将多个参数收集到一个数组中，使用 ... 语法来表示。它可以处理参数数量不确定的情况。

```
function sum(...numbers: number[]): number {
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3));
console.log(sum(4, 5, 6, 7));
```

3. 函数 3.6 函数重载

- 函数重载：函数重载允许为同一个函数提供多个不同的调用签名，根据不同的参数类型和数量来实现不同的逻辑。这在处理多种输入情况时非常有用。

```typescript
// 函数重载签名
function processInput(input: string): string;
function processInput(input: number): number;

// 函数实现
function processInput(input: string | number): string | number {
    if (typeof input === 'string') {
        return input.toUpperCase();
    } else {
        return input * 2;
    }
}

console.log(processInput('hello'));
console.log(processInput(5));
```

示例7-3-2：函数示例

4. 接口 4.1 基础接口

- 定义对象结构：接口最常见的用途是定义对象的结构，规定对象必须包含哪些属性以及这些属性的类型。

```typescript
// 定义一个接口来描述用户对象
interface User {
    name: string;
    age: number;
    isAdmin: boolean;
}

// 创建一个符合 User 接口的对象
const user: User = {
    name: 'John Doe',
    age: 30,
    isAdmin: false
};
```

4. 接口 4.1 基础接口

- 可选属性：接口中的属性可以是可选的，在属性名后面加上 ? 即可。

```typescript
interface Book {
    title: string;
    author?: string;
    year?: number;
}

// 创建一个符合 Book 接口的对象
const book1: Book = {
    title: 'The Great Gatsby'
};

const book2: Book = {
    title: 'To Kill a Mockingbird',
    author: 'Harper Lee',
    year: 1960
};
```


4. 接口 4.1 基础接口

- 只读属性：使用 readonly 关键字可以将接口中的属性设置为只读，一旦对象被创建，该属性的值就不能再被修改。

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

const point: Point = { x: 10, y: 20 };
// 下面这行代码会报错，因为 x 是只读属性
// point.x = 30;
```

4. 接口 4.2 函数类型接口

- 接口不仅可以描述对象的属性，还能描述函数的类型。

```typescript
// 定义一个函数类型接口
interface MathOperation {
    (a: number, b: number): number;
}

// 创建一个符合 MathOperation 接口的函数
const add: MathOperation = (a, b) => a + b;
const subtract: MathOperation = (a, b) => a - b;

console.log(add(3, 5));
console.log(subtract(8, 2));
```


4. 接口 4.3 可索引类型接口

- 可索引类型接口允许你定义对象的索引签名，从而可以通过索引访问对象的属性。

```typescript
// 定义一个字符串数组的可索引类型接口
interface StringArray {
    [index: number]: string;
}

const myArray: StringArray = ['apple', 'banana',
'cherry'];
console.log(myArray[1]);

// 定义一个字典的可索引类型接口
interface Dictionary {
    [key: string]: number;
}

const dict: Dictionary = { 'one': 1, 'two': 2, 'three': 3
};
console.log(dict['two']);
```

4. 接口 4.4 接口继承

- 接口可以继承其他接口，使用 extends 关键字来实现继承，继承后会拥有父接口的所有属性和方法。

```typescript
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

// 创建一个符合 Square 接口的对象
const square: Square = {
    color: 'red',
    sideLength: 10
};
```

示例7-4-1：函数

5. 类和对象 5.1 类的定义

- 在 TypeScript 中，类和对象是实现面向对象编程的重要概念，TypeScript 提供了丰富的特性来支持类的定义、继承、封装和多态等面向对象的特性。
  - 类的定义：使用 class 关键字来定义一个类，类中可以包含属性和方法。

```typescript
class Person {
    // 属性
    name: string;
    age: number;

    // 构造函数
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    // 方法
    greet() {
        console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
    }
}
```

4. 类和对象 4.1 类的定义

- 在 TypeScript 中，类和对象是实现面向对象编程的重要概念，TypeScript 提供了丰富的特性来支持类的定义、继承、封装和多态等面向对象的特性。
  - 对象的创建：通过 new 关键字来创建类的实例（对象）。

```
const person = new Person('John', 30);
person.greet();
```


5. 类和对象 5.2 类的访问修饰符

- TypeScript 提供了三种访问修饰符：public、private 和 protected，用于控制类的属性和方法的访问权限。
  - public：默认情况下，类的属性和方法都是 public 的，可以在类的内部和外部自由访问。

```
class Car {
    public brand: string;

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

const car = new Car('Toyota');
console.log(car.brand);
```

5. 类和对象 5.2 类的访问修饰符

- TypeScript 提供了三种访问修饰符：public、private 和 protected，用于控制类的属性和方法的访问权限。
  - private：private 修饰的属性和方法只能在类的内部访问，外部无法直接访问。

```typescript
class BankAccount {
    private balance: number;

    constructor(initialBalance: number) {
        this.balance = initialBalance;
    }

    public deposit(amount: number) {
        this.balance += amount;
    }

    public getBalance() {
        return this.balance;
    }
}

const account = new BankAccount(1000);
// 下面这行代码会报错，因为 balance 是 private 属性
// console.log(account.balance);
console.log(account.getBalance());
```


5. 类和对象 5.2 类的访问修饰符

- TypeScript 提供了三种访问修饰符：public、private 和 protected，用于控制类的属性和方法的访问权限。
  - protected：protected 修饰的属性和方法可以在类的内部以及继承该类的子类中访问，但在类的外部无法访问。

```typescript
class Animal {
    protected name: string;

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

    protected getName() {
        return this.name;
    }
}

class Dog extends Animal {
    constructor(name: string) {
        super(name);
    }

    bark() {
        console.log(`${this.getName()} says Woof!`);
    }
}

const dog = new Dog('Buddy');
// 下面这行代码会报错，因为 getName 是 protected 方法
// console.log(dog.getName());
dog.bark();
```

5. 类和对象 5.2 类的访问修饰符

- TypeScript 提供了三种访问修饰符：public、private 和 protected，用于控制类的属性和方法的访问权限。
  - readonly修饰符：readonly关键字将属性设置为只读的。 只读属性必须在声明时或构造函数里被初始化。

```
lass Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // 错误! name 是只读的.
```


5. 类和对象 5.3 类的继承

- 使用 extends 关键字实现类的继承，子类可以继承父类的属性和方法，并且可以添加自己的属性和方法。

```
class Shape {
    color: string;

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

    getColor() {
        return this.color;
    }
}

class Circle extends Shape {
    radius: number;

    constructor(color: string, radius: number) {
        super(color);
        this.radius = radius;
    }

    getArea() {
        return Math.PI * this.radius * this.radius;
    }
}

const circle = new Circle('red', 5);
console.log(circle.getColor());
console.log(circle.getArea());
```

5. 类和对象 5.4 抽象类

- 抽象类：抽象类是一种不能被实例化的类，它主要用于定义一些通用的属性和方法，供子类继承和实现。使用 abstract 关键字来定义抽象类和抽象方法。

```
class MathUtils {
    static PI = 3.14159;

    static calculateCircleArea(radius: number) {
        return this.PI * radius * radius;
    }
}

console.log(MathUtils.PI);
console.log(MathUtils.calculateCircleArea(5));
```


5. 类和对象 5.5 类的访问器（Getter 和 Setter）

- 使用 get 和 set 关键字来定义访问器，用于控制属性的读取和赋值操作。

```
class Person {
    private _age: number;

    get age() {
        return this._age;
    }

    set age(value: number) {
        if (value >= 0 && value <= 120) {
            this._age = value;
        } else {
            console.log('Invalid age value.');
        }
    }
}

const person = new Person();
person.age = 25;
console.log(person.age);
```

示例7-5-1：类和对象

信创智能医疗系统研发课程体系

河南中医药大学信息技术学院（智能医疗行业学院）

河南中医药大学信息技术学院（智能医疗行业学院）智能医疗教研室

河南中医药大学医疗健康信息工程技术研究所

71