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08章：TypeScript进阶

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

2025/3/28

本章概要

- 泛型
- 模块与命名空间
- 装饰器
- 错误处理
- 异步编程
- 编译
- 使用TypeScript开发项目

1. 泛型 1.1 简介

- 在编写代码的过程中，往往需要考虑代码的通用性。如果一些代码模块不仅可以支持当前已有的数据类型，还支持以后可能定义的数据类型，那么这些代码模块就具备良好的通用性。
- 使用泛型创建具备通用性的代码模块，这些代码模块可以支持多种类型的数据，开发者可以根据自己的需求，指定具体的数据类型来复用该代码模块。
- 使用泛型的优势如下。
  - 代码重用：可以编写与特定类型无关的通用代码，提高代码的复用性。
  - 类型安全：在编译时进行类型检查，避免在运行时出现类型错误。
  - 抽象性：允许编写更抽象和通用的代码，适应不同的数据类型和数据结构。

1. 泛型 1.2 泛型函数

- 泛型函数能够处理多种数据类型。通过在函数名后面使用尖括号 <T> 来声明泛型类型参数，T 代表任意类型。

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

// 使用泛型函数
let output1 = identity<string>("myString");
let output2 = identity<number>(100);

console.log(output1);
console.log(output2);
```

1. 泛型 1.4 泛型接口

- 可以使用泛型来定义接口。

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}

function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: GenericIdentityFn<number> = identity;
console.log(myIdentity(20));
```

1. 泛型 1.3 泛型类

- 泛型类与泛型函数类似，在类名后面使用尖括号 <T> 来声明泛型类型参数。

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

    constructor(zeroValue: T, add: (x: T, y: T) => T) {
        this.zeroValue = zeroValue;
        this.add = add;
    }
}

let myGenericNumber = new GenericNumber<number>(0, function (x, y) { return x + y; });
console.log(myGenericNumber.add(5, 3));
```

1. 泛型 1.5 泛型约束

- 有时候可能需要对泛型类型进行一些限制，这时就可以使用泛型约束。通过 extends 关键字来实现泛型约束。

```
interface Lengthwise {
    length: number;
}

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

// 正确使用
loggingIdentity("hello");
// 错误使用，数字没有 length 属性
// loggingIdentity(10);
```

1. 泛型 1.6 注意事项

- 尽量少用泛型
  - 泛型虽然灵活，但是会加大代码的复杂性，使其变得难读难写。一般来说，只要使用了泛型，类型声明通常都不太易读，容易写得很复杂。因此，可以不用泛型就不要用。
- 类型参数越少越好
  - 多一个类型参数，多一道替换步骤，加大复杂性。

2. 模块与命名空间

- 在编写程序代码时，不仅要考虑如何实现功能，还需要考虑如何组织代码。大型项目通常涉及多个团队、多个文件，单个文件的代码行通常也是数以千计的，如果无法有效地组织代码，程序将难以维护。
- 在TypeScript中，通过模块和命名空间组织代码。

2. 模块与命名空间 2.1 模块

- 模块是 TypeScript 中组织代码的推荐方式。
- 模块化代码可以将功能拆分为多个文件，每个文件都是一个独立的模块。
- 模块通过 import 和 export 关键字来导入和导出功能。
- 模块有以下特点。
  - 文件即模块：每个文件都是一个独立的模块。
  - 作用域隔离：模块中的变量、函数、类等默认是私有的，除非显式导出。
  - 依赖管理：模块之间通过 import 和 export 明确依赖关系。
  - 支持动态加载：模块可以动态加载（如使用 import()）。

2. 模块与命名空间 2.1 模块

- 导出（Export）
  - 模块导出使用关键字 export 关键字导出模块中的变量、函数、类等。

```ts
// math.ts
export function add(a: number, b: number): number {
    return a + b;
}

export const PI = 3.14;
```

2. 模块与命名空间 2.1 模块

- 导出（Export）
  - 命令行内导出：变量、函数、类等进行声明时就导出，export关键字与声明语句位于同一行
  - 命名子句导出：在花括号中包含多个声明，进行批量导出。使用命名子句导出还可以为导出的声明指定别名，这样其他文件在导入这些声明时必须使用指定的别名，而非原始声明名称。

```
export let x: number = 11;
export const y: string = "abc";
export function sayHello() {
    console.log("hello!");
}
export class A { name: string }
export interface B { sayHello: () => {} }
export type NumberOrString = number | string;
```

```
let x: number = 11;
const y: string = "abc";
function sayHello() {
    console.log("hello!");
}
export { x, y, sayHello }
```

```
let x: number = 11;
const y: string = "abc";
function sayHello() {
    console.log("hello!");
}

export { x as myX, y as myY, sayHello as mySayHello }
```

2. 模块与命名空间 2.1 模块

- 导出（Export）
  - 命名子句导出：使用命名子句导出还可以为导出的声明指定别名，这样其他文件在导入这些声明时必须使用指定的别名，而非原始声明名称。
  - 默认导出：可以为模块指定默认导出。如果其他文件在导入该模块时未指定具体声明，则会使用该模块的默认导出。一个模块只有一个默认导出。
  - 空导出如果需要将一个文件标记为模块，以便其他文件引用，但并不对外暴露任何声明，或者只想让当前模块拥有隔离的作用域，可以对模块进行空导出。

```
let x: number = 11;
const y: string = "abc";
function sayHello() {
    console.log("hello!");
}

export { x as myX, y as myY, sayHello as mySayHello }
```

```
let x: number = 11;
export default x;
```

2. 模块与命名空间 2.1 模块

- 导入（Import）
  - 使用 import 关键字导入其他模块的内容。

```
let a = "a";
let b = "b";
let c = "c";
export let d = "d";
export { a, b as default, c as myC }
```

```
import { a, d, myC } from "./a.js";
console.log(a);    //输出"a"
console.log(myC);  //输出"c"
console.log(d);    //输出"d"
```

```
import { a as otherA, d as otherD, myC as
otherC, default as otherB } from "./a.js";
console.log(otherA); //输出"a"
console.log(otherB); //输出"b"
console.log(otherC); //输出"c"
console.log(otherD); //输出"d"
```

```
import * as moduleA from "./a.js";
console.log(moduleA.a);       //输出"a"
console.log(moduleA.default); //输出"b"
console.log(moduleA.myC);     //输出"c"
console.log(moduleA.d);       //输出"d"
```

2. 模块与命名空间 2.1 模块

- 注意事项
  - 每个模块拥有各自独立的作用域
    - 在未使用export/import关键字将文件标记为模块前，多个文件的声明在同一个作用域中。如果两个文件具有同名的声明，将出现编译错误。
    - 一旦使用export关键字将文件标记为模块，则每个模块都将被视为互相隔离的作用域，即使各个模块拥有同名变量，也不会互相影响。
  - import/export语句必须位于模块的顶级
    - export/import语句必须位于模块的顶级，不能嵌套在代码块中，否则将引起编译错误。

2. 模块与命名空间 2.1 模块

- 编译与运行模块
  - 对于模块，TypeScript有多种编译方式，每种编译方式产生的JavaScript代码各不相同，运行方式也有所区别，这都与JavaScript的模块规范有关。
    - CommonJS规范。
    - 异步模块定义(Asynchronous Module Definition，AMD)规范。
    - 通用模块定义(Universal Module Definition，UMD)规范。
    - ECMAScript规范（这是最新的规范，将替代以往的所有规范）。

2. 模块与命名空间 2.1 模块

- 编译与运行模块
  - CommonJS 规范
    - 概述
      - CommonJS 是服务器端模块规范，由 Mozilla 的工程师于 2009 年提出，主要用于服务器端编程，Node.js 采用了该规范。它的核心思想是通过 require 函数来同步加载模块，使用 exports 或 module.exports 来导出模块内容。
    - 特点
      - 同步加载：在使用 require 加载模块时，程序会暂停执行，直到模块加载完成，这在服务器端环境下不会有太大问题，因为服务器端的文件系统读取速度相对较快。
      - 模块缓存：同一个模块只会被加载一次，后续的 require 调用会直接返回之前加载的模块实例，提高了性能。
    - 编译与运行
      - 在 TypeScript 中使用 CommonJS 规范，需要在 tsconfig.json 中设置 module 为 commonjs，然后使用 tsc 编译，最后用 Node.js 运行生成的 JavaScript 文件。

```
// mathUtils.js
// 定义一个函数并导出
function add(a, b) {
    return a + b;
}
// 将 add 函数导出
module.exports = {
    add: add
};

// main.js
// 引入 mathUtils 模块
const mathUtils = require('./mathUtils');
// 使用模块中的 add 函数
const result = mathUtils.add(3, 5);
console.log(result);
```

2. 模块与命名空间 2.1 模块

- 编译与运行模块
  - AMD 规范
    - 概述
      - 异步模块定义（Asynchronous Module Definition，AMD）是为浏览器环境设计的模块规范，由 RequireJS 推广开来。由于浏览器环境中网络请求是异步的，AMD 采用异步加载模块的方式，避免了同步加载可能导致的页面卡顿问题。
    - 特点
      - 异步加载：模块可以异步加载，不会阻塞页面的渲染和其他脚本的执行。
      - 依赖前置：在定义模块时，需要明确列出该模块所依赖的其他模块，这样可以提前加载依赖，提高加载效率。
    - 编译与运行
      - 在 TypeScript 中使用 AMD 规范，需要在 tsconfig.json 中设置 module 为 amd，编译后需要引入 RequireJS 并在 HTML 文件中配置。

```
// mathUtils.js
// 定义一个 AMD 模块
define(function() {
    function add(a, b) {
        return a + b;
    }
    return {
        add: add
    };
});

// main.js
// 配置 RequireJS
require.config({
    baseUrl: './',
    paths: {
        mathUtils: 'mathUtils'
    }
});
// 加载模块并使用
require(['mathUtils'], function(mathUtils) {
    const result = mathUtils.add(3, 5);
    console.log(result);
});
```

2. 模块与命名空间 2.1 模块

- 编译与运行模块
  - UMD 规范
    - 概述
      - 通用模块定义（Universal Module Definition，UMD）是一种兼容 CommonJS 和 AMD 的模块规范，它可以让模块在服务器端和浏览器端都能正常使用。UMD 会先判断当前环境支持哪种模块规范，然后采用相应的方式来定义和加载模块。
    - 特点
      - 通用性：可以在不同的环境中使用，无论是服务器端的 Node.js 还是浏览器端的 JavaScript 都能兼容。
      - 灵活性：结合了 CommonJS 和 AMD 的优点，既支持同步加载，也支持异步加载。
    - 编译与运行
      - 在 TypeScript 中使用 UMD 规范，需要在 tsconfig.json 中设置 module 为 umd，编译后可以在服务器端使用 Node.js 运行，也可以在浏览器端直接引入生成的 JavaScript 文件。

```
// mathUtils.js
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD 环境
        define([], factory);
    } else if (typeof module === 'object' && module.exports) {
        // CommonJS 环境
        module.exports = factory();
    } else {
        // 全局环境
        root.mathUtils = factory();
    }
}(this, function () {
    function add(a, b) {
        return a + b;
    }
    return {
        add: add
    };
}));

// main.js
// 在 CommonJS 环境中使用
const mathUtils = require('./mathUtils');
const result = mathUtils.add(3, 5);
console.log(result);
```

2. 模块与命名空间 2.1 模块

- 编译与运行模块
  - ECMAScript 规范
    - 概述
      - ECMAScript 模块规范是 JavaScript 官方的模块规范，从 ECMAScript 2015（ES6）开始引入。它采用静态导入和导出的方式，使得模块的依赖关系在编译时就能确定，有利于代码的静态分析和优化。
    - 特点
      - 静态导入导出：模块的导入和导出语句是静态的，在编译时就能确定模块之间的依赖关系，便于进行代码分割和打包。
      - 原生支持：现代浏览器和 Node.js 都原生支持 ECMAScript 模块规范，无需额外的工具或库。
    - 编译与运行
      - 在 TypeScript 中使用 ECMAScript 模块规范，需要在 tsconfig.json 中设置 module 为 esnext，编译后在 Node.js 中运行时，需要在 package.json 中添加 "type": "module"，在浏览器中运行时，需要在 HTML 文件中使用 <script type="module"> 引入。

```
// mathUtils.js
// 导出函数
export function add(a, b) {
    return a + b;
}

// main.js
// 导入函数
import { add } from './mathUtils.js';
const result = add(3, 5);
console.log(result);
```

2. 模块与命名空间 2.2 命名空间

- 命名空间是 TypeScript 早期用于组织代码的方式，主要用于将相关的代码分组到一个命名空间内，避免全局作用域污染。
- 命名空间通过 namespace 关键字定义。
- 命名空间有以下特点。
  - 逻辑分组：将相关的代码组织到一个命名空间内。
  - 避免全局污染：命名空间内的内容默认不会暴露到全局作用域。
  - 需要显式导出：命名空间内的内容需要通过 export 导出才能被外部访问。

2. 模块与命名空间 2.2 命名空间

- 定义命名空间：TypeScript使用 namespace 关键字定义命名空间

```typescript
namespace MathUtils {
    export function add(a: number, b: number): number {
        return a + b;
    }

    export const PI = 3.14;
}
```

2. 模块与命名空间 2.2 命名空间

- 命名空间的嵌套：命名空间可以嵌套使用。

```
namespace Outer {
    export namespace Inner {
        export function greet(): void {
            console.log("Hello from Inner!");
        }
    }
}

Outer.Inner.greet(); // 输出: Hello from Inner!
```

2. 模块与命名空间 2.2 命名空间

- 使用命名空间：通过命名空间名称访问其内容。

```ts
// main3.ts
/// <reference path="shapes.ts" />
let circle = new Shapes.Circle(5);
console.log(circle.area());

let square = new Shapes.Square(4);
console.log(square.area());
```

- 这里的 /// <reference path="shapes.ts" /> 是三斜线指令，它告知编译器在编译时要包含 shapes.ts 文件。

2. 模块与命名空间 2.2 命名空间

- 模块与命名空间的对比
  - 模块
    - 适用于大型项目，能够更好地管理依赖和代码分割。
    - 遵循 ES6 模块规范，与现代 JavaScript 生态系统兼容。
    - 每个模块都有自己独立的作用域，不会污染全局命名空间。
  - 命名空间
    - 适用于小型项目或者在没有模块系统的环境中使用。
    - 主要用于避免全局命名冲突，将相关代码组织在一起。
    - 命名空间会在全局作用域中创建一个对象，可能会造成全局命名空间的污染。

3. 装饰器

- 装饰器（Decorator）是一种语法结构，用来在定义时修改类（class）的行为。
- 在语法上，装饰器有如下几个特征。
  - 第一个字符（或者说前缀）是@，后面是一个表达式。
  - @后面的表达式，必须是一个函数（或者执行后可以得到一个函数）。
  - 这个函数接受所修饰对象的一些相关值作为参数。
  - 这个函数要么不返回值，要么返回一个新对象取代所修饰的目标对象。

3. 装饰器

- 类装饰器：类装饰器用于类的构造函数，可对类的定义进行观察、修改或者替换。

```
function logClass(constructor: Function) {
    console.log(`Class ${constructor.name} has been created.`);
}

@logClass
class MyClass {
    constructor() {
        console.log('MyClass instance created.');
    }
}

const myInstance = new MyClass();
```

3. 装饰器

- 方法装饰器：方法装饰器应用于类的方法上，接收三个参数：目标对象、方法名以及属性描述符。

```
function logMethod(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const originalMethod = descriptor.value;
    descriptor.value = function (...args: any[]) {
        console.log(`Calling method ${propertyKey} with arguments: ${JSON.stringify(args)}`);
        const result = originalMethod.apply(this, args);
        console.log(`Method ${propertyKey} returned: ${result}`);
        return result;
    };
    return descriptor;
}

class Calculator {
    @logMethod
    add(a: number, b: number) {
        return a + b;
    }
}

const calculator = new Calculator();
const result = calculator.add(2, 3);
```

3. 装饰器

- 属性装饰器：属性装饰器应用于类的属性上，接收两个参数：目标对象和属性名。

```
function logProperty(target: any, propertyKey: string) {
    let value = target[propertyKey];
    const getter = () => {
        console.log(`Getting value of ${propertyKey}: ${value}`);
        return value;
    };
    const setter = (newValue: any) => {
        console.log(`Setting value of ${propertyKey} to: ${newValue}`);
        value = newValue;
    };
    Object.defineProperty(target, propertyKey, {
        get: getter,
        set: setter,
        enumerable: true,
        configurable: true
    });
}

class Person {
    @logProperty
    name: string;

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

const person = new Person('John');
person.name = 'Jane';
console.log(person.name);
```

3. 装饰器

- 参数装饰器：参数装饰器应用于类的方法参数上，接收三个参数：目标对象、方法名和参数索引。

```
function logParameter(target: any, propertyKey: string, parameterIndex: number) {
    console.log(`Parameter at index ${parameterIndex} of method ${propertyKey} has been decorated.`);
}

class Greeter {
    greet(@logParameter message: string) {
        return `Hello, ${message}!`;
    }
}

const greeter = new Greeter();
greeter.greet('World');
```

3. 装饰器

- 装饰器工厂：装饰器工厂是一个返回装饰器的函数，它允许你在应用装饰器时传递参数。

```
function logWithMessage(message: string) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        const originalMethod = descriptor.value;
        descriptor.value = function (...args: any[]) {
            console.log(`${message} Calling method ${propertyKey} with arguments: ${JSON.stringify(args)}`);
            const result = originalMethod.apply(this, args);
            console.log(`${message} Method ${propertyKey} returned: ${result}`);
            return result;
        };
        return descriptor;
    };
}

class Calculator {
    @logWithMessage('Custom message:')
    add(a: number, b: number) {
        return a + b;
    }
}

const calculator = new Calculator();
const result = calculator.add(2, 3);
```

3. 装饰器

- 装饰器执行顺序
  - 当有多个装饰器应用于同一个声明时，它们的执行顺序如下：
  - 参数装饰器，从左到右依次执行。
  - 方法装饰器、属性装饰器，按照声明顺序执行。
  - 类装饰器，从下到上依次执行。
- 使用场景
  - 日志记录：像前面示例那样，在方法调用前后记录日志。
  - 权限验证：在方法执行前检查用户是否有执行该方法的权限。
  - 性能监控：测量方法的执行时间。
  - 依赖注入：自动注入依赖项。

- TypeScript 装饰器能在不修改类的核心逻辑的情况下，添加额外的功能和行为。

4. 错误处理

- 在 TypeScript 里，错误处理是确保程序健壮性与稳定性的重要部分。
- 抛出和捕获异常：TypeScript 采用 try...catch 语句块来捕获和处理异常，使用 throw 语句来抛出异常。

```typescript
function divide(a: number, b: number): number {
    if (b === 0) {
        throw new Error("除数不能为零");
    }
    return a / b;
}

try {
    const result = divide(10, 0);
    console.log(result);
} catch (error) {
    if (error instanceof Error) {
        console.error(error.message);
    }
}
```

4. 错误处理

- 自定义错误类型：抛出和捕获异常：可以创建自定义的错误类型，从而让错误处理更具针对性。

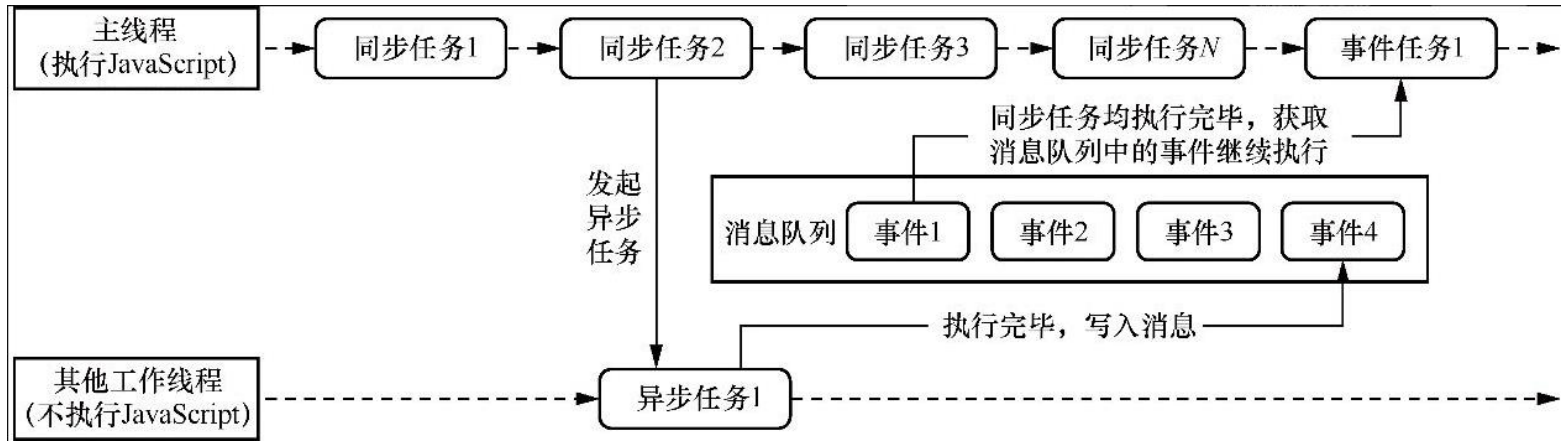
```
class CustomError extends Error {
    constructor(message: string) {
        super(message);
        this.name = 'CustomError';
    }
}

function validateAge(age: number): void {
    if (age < 0) {
        throw new CustomError("年龄不能为负数");
    }
}

try {
    validateAge(-5);
} catch (error) {
    if (error instanceof CustomError) {
        console.error(`${error.name}: ${error.message}`);
    }
}
```

5. 异步编程

- TypeScript代码最终会编译成JavaScript代码来执行，而JavaScript代码是单线程执行的，即同一时间只能执行一个任务。
- JavaScript之所以不支持多线程执行，主要由于它起源于浏览器。
  - 作为浏览器脚本，它会与用户进行交互，并操作UI的DOM结构，如果支持多线程并发操作，将会引起比较复杂的问题，例如，线程1在DOM上添加一个节点，而线程2删除该节点，此时操作将出现冲突。为了避免产生这类问题，JavaScript被设计成了单线程执行模式。
- JavaScript之所以不支持多线程执行，主要由于它起源于浏览器。作为浏览器脚本，它会与用户进行交互，并操作UI的DOM结构，如果支持多线程并发操作，将会引起比较复杂的问题，例如，线程1在DOM上添加一个节点，而线程2删除该节点，此时操作将出现冲突。为了避免产生这类问题，JavaScript被设计成了单线程执行模式。

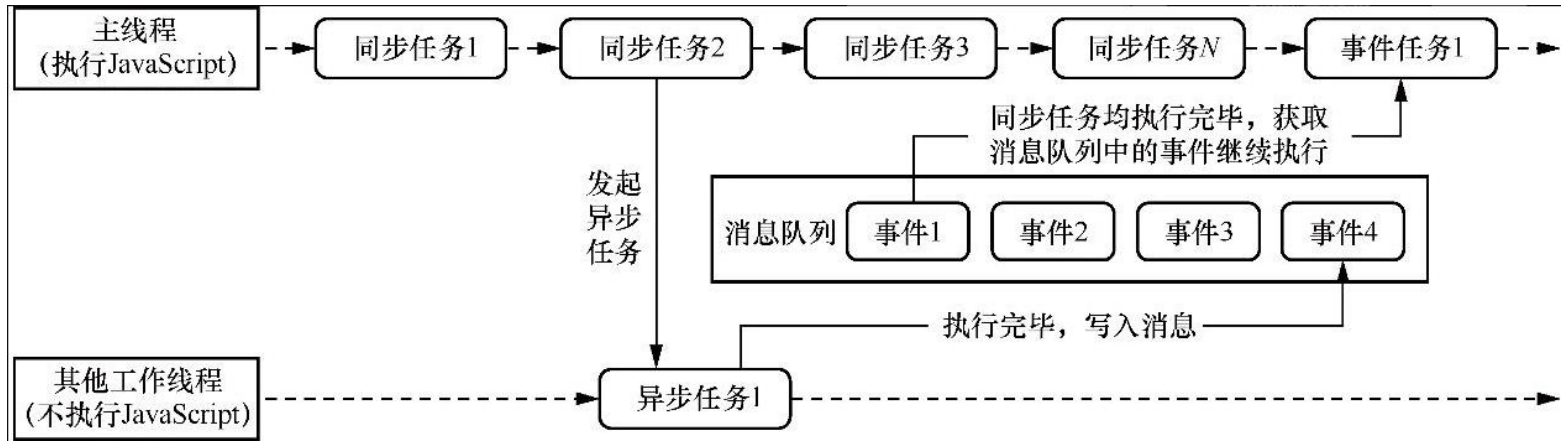
5. 异步编程

- JavaScript主线程依次执行各个同步任务。当执行到特定代码时，JavaScript会通知其他非JavaScript线程执行任务，这些任务不运行JavaScript，不进入JavaScript主线程。JavaScript主线程不会等待，而继续执行下一个同步任务。当非JavaScript线程执行完成后，会将结果以事件通知的形式存到JavaScript任务队列中。当JavaScript主线程的所有同步任务执行完成后，会检查JavaScript任务队列中有哪些事件，并定位到具体回调函数代码上，然后以同步任务的形式依次执行它们。
- JavaScript是单线程的，是指只有一个线程来执行JavaScript脚本。但运行环境（Node.js或浏览器）不是单线程的，一些I/O操作、网络请求、定时器和事件监听等都是由运行环境提供的，是由其他非JavaScript线程来完成的，这些非JavaScript线程的任务执行完成后，会将结果以事件通知的形式存到任务队列中。只要JavaScript主线程中的任务清空了，就会从任务队列中获取任务，然后继续执行，这个过程不断重复，这就是JavaScript的异步任务运行机制。



5. 异步编程

- JavaScript主线程依次执行各个同步任务。当执行到特定代码时，JavaScript会通知其他非JavaScript线程执行任务，这些任务不运行JavaScript，不进入JavaScript主线程。JavaScript主线程不会等待，而继续执行下一个同步任务。当非JavaScript线程执行完成后，会将结果以事件通知的形式存到JavaScript任务队列中。当JavaScript主线程的所有同步任务执行完成后，会检查JavaScript任务队列中有哪些事件，并定位到具体回调函数代码上，然后以同步任务的形式依次执行它们。
- JavaScript是单线程的，是指只有一个线程来执行JavaScript脚本。但运行环境（Node.js或浏览器）不是单线程的，一些I/O操作、网络请求、定时器和事件监听等都是由运行环境提供的，是由其他非JavaScript线程来完成的，这些非JavaScript线程的任务执行完成后，会将结果以事件通知的形式存到任务队列中。只要JavaScript主线程中的任务清空了，就会从任务队列中获取任务，然后继续执行，这个过程不断重复，这就是JavaScript的异步任务运行机制。



5. 异步编程

- 基于JavaScript的异步运行机制，先后诞生了不同的异步编程模式，主要的异步编程模式如下。
  - 回调函数
  - Promise对象(ECMAScript 6)。
  - async/await语法(ECMAScript 7)。

5. 异步编程

- 回调函数：回调函数是实现异步编程的基础方式，它把一个函数作为参数传递给另一个函数，在异步操作完成后调用该函数。

```
function fetchData(callback: (data: string) => void) {
    setTimeout(() => {
        const data = '异步获取的数据';
        callback(data);
    }, 1000);
}

fetchData((data) => {
    console.log(data);
});
```

5. 异步编程

- 回调函数：回调函数是实现异步编程的基础方式，它把一个函数作为参数传递给另一个函数，在异步操作完成后调用该函数。

```
function fetchData(callback: (data: string) => void) {
    setTimeout(() => {
        const data = '异步获取的数据';
        callback(data);
    }, 1000);
}

fetchData((data) => {
    console.log(data);
});
```

- 在这个例子中，fetchData 函数模拟一个异步操作，操作完成后调用传入的回调函数并传递数据。

5. 异步编程

- Promise：Promise 是 ES6 引入的一种异步编程解决方案，用于处理异步操作的结果，它有三种状态：pending（进行中）、fulfilled（已成功）和 rejected（已失败）。

```
function fetchDataPromise(): Promise<string> {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const data = '异步获取的数据';
            resolve(data);
        }, 1000);
    });
}

fetchDataPromise()
  .then((data) => {
        console.log(data);
    })
  .catch((error) => {
        console.error(error);
    });
```

- fetchDataPromise 函数返回一个 Promise 对象，在异步操作完成后，通过 resolve 方法传递成功结果，使用 .then() 方法处理成功结果，使用 .catch() 方法处理失败结果。

5. 异步编程

- async/await：async/await 是 ES8 引入的语法糖，基于 Promise 实现，能让异步代码看起来更像同步代码。

```
function fetchDataAsync(): Promise<string> {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const data = '异步获取的数据';
            resolve(data);
        }, 1000);
    });
}

async function main() {
    try {
        const data = await fetchDataAsync();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

main();
```

- main 函数被定义为 async 函数，使用 await 关键字等待 fetchDataAsync 函数的 Promise 结果，若出现错误，使用 try...catch 语句块捕获并处理。

5. 异步编程

- 异步迭代器和生成器：TypeScript 支持异步迭代器和生成器，可用于处理异步数据流。

```typescript
async function* asyncGenerator() {
    let i = 0;
    while (i < 3) {
        await new Promise(resolve => setTimeout(resolve, 1000));
        yield i++;
    }
}

(async () => {
    for await (const num of asyncGenerator()) {
        console.log(num);
    }
})();
```

- asyncGenerator 是一个异步生成器函数，使用 yield 关键字返回异步数据，使用 for await...of 循环遍历异步数据流。

5. 异步编程

- 在异步编程中，错误处理至关重要。可以使用 .catch() 方法处理 Promise 中的错误，使用 try...catch 语句块处理 async/await 中的错误。

```
function fetchDataWithError(): Promise<string> {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject(new Error('获取数据时出错'));
        }, 1000);
    });
}

fetchDataWithError()
  .then((data) => {
        console.log(data);
    })
  .catch((error) => {
        console.error(error);
    });

async function mainWithError() {
    try {
        await fetchDataWithError();
    } catch (error) {
        console.error(error);
    }
}

mainWithError();
```

6. 编译

- 单个文件编译：如果你只想编译一个 TypeScript 文件，比如 example.ts，可以直接在命令行输入。

```bash
tsc example.ts
```

- 多个文件编译：当需要编译多个 TypeScript 文件时，只需在 tsc 命令后面依次列出这些文件的名称，用空格分隔。

```bash
tsc file1.ts file2.ts file3.ts
```

6. 编译

- 使用 tsconfig.json 配置文件
  - tsconfig.json 是 TypeScript 项目的核心配置文件，它能定义编译选项以及项目文件的范围。若项目中还没有这个文件，可以使用以下命令进行初始化。

```bash
tsc --init
```

  - 在有 tsconfig.json 文件的项目中，只需在命令行输入 tsc 命令，编译器就会自动读取 tsconfig.json 文件中的配置选项，并编译其中指定的文件。
    - 这种方式非常适合大型项目，因为可以通过配置文件集中管理编译选项，而不需要在每次编译时都在命令行中指定各种参数。

6. 编译

- 使用 tsconfig.json 配置文件
  - 指定输出目录（--outDir）：使用 --outDir 选项可以将编译后的 JavaScript 文件输出到指定的目录。例如，将 example.ts 编译后的文件输出到 dist 目录。

```bash
tsc --outDir dist example.ts
```

  - 监听文件变化（--watch）：--watch 选项可以让 TypeScript 编译器监听文件的变化，当文件发生改动时自动重新编译。例如，监听 example.ts 文件的变化。

```bash
tsc --watch example.ts
```

6. 编译

- 使用 tsconfig.json 配置文件
  - 生成 Source Map 文件（--sourceMap）：Source Map 文件可以将编译后的 JavaScript 代码映射回原始的 TypeScript 代码，方便调试。使用 --sourceMap 选项可以在编译时生成对应的 Source Map 文件。执行该命令后，除了生成 example.js 文件外，还会生成一个 example.js.map 文件

```bash
tsc --sourceMap example.ts
```

  - 指定目标 ECMAScript 版本（--target）：--target 选项用于指定编译后的 JavaScript 代码所遵循的 ECMAScript 版本。常见的值有 ES3、ES5、ES6 等。例如，将代码编译为 ES6 版本。

```bash
tsc --target ES6 example.ts
```

6. 编译

- 使用 tsconfig.json 配置文件
  - 增量编译（--incremental）：--incremental 选项可以开启增量编译模式，它会记录上一次编译的状态，只重新编译那些有变化的文件，从而提高编译速度。

```bash
tsc --incremental
```

  - 生成声明文件（--declaration）：使用 --declaration 选项可以在编译时生成对应的 .d.ts 声明文件，这些文件可以用于类型检查和代码提示。

```bash
tsc --declaration example.ts
```

6. 编译

- 使用 tsconfig.json 配置文件
  - tsconfig.json 文件可以对编译选项进行更细致的配置。以下是一个常见的 tsconfig.json 文件示例。

```json
{
    "compilerOptions": {
        "target": "ES6", // 指定目标 ECMAScript 版本
        "module": "commonjs", // 指定模块系统
        "outDir": "./dist", // 指定输出目录
        "rootDir": "./src", // 指定源代码的根目录
        "strict": true, // 启用所有严格类型检查选项
        "esModuleInterop": true, // 支持 ES 模块和 CommonJS 模块之间的互操作性
        "skipLibCheck": true, // 跳过对类型声明文件的检查
        "forceConsistentCasingInFileNames": true // 强制文件名称大小写一致
    },
    "include": ["src/**/*.ts"], // 指定需要编译的文件范围
    "exclude": ["node_modules", "dist"] // 指定需要排除的文件或目录
}
```

6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤1：项目初始化

```bash
mkdir book-management-system
cd book-management-system
npm init -y
```

6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 2：安装 TypeScript

```bash
npm install commander
npm install typescript @types/node @types/commander --save-dev
```

6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 3：配置 TypeScript
    - 初始化 TypeScript 配置文件 tsconfig.json

```bash
npx tsc --init
```

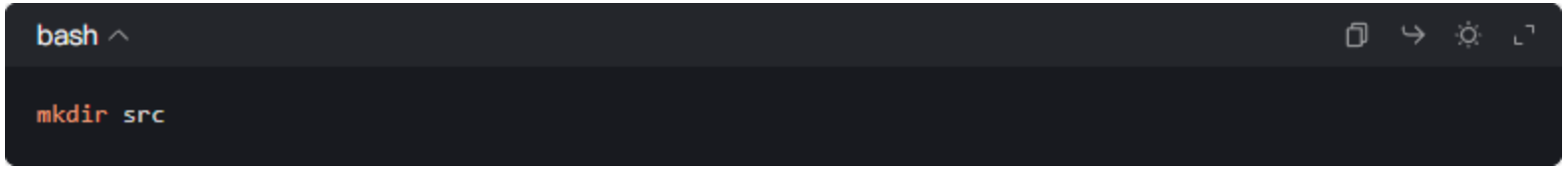
6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 3：配置 TypeScript
    - 编辑 tsconfig.json 文件，设置如下配置

```json
{
    "compilerOptions": {
        "target": "ES6",
        "module": "commonjs",
        "outDir": "./dist",
        "rootDir": "./src",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true
    },
    "include": ["src/**/*.ts"]
}
```

6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 4：创建项目结构
    - 在项目根目录下创建 src 目录，用于存放 TypeScript 源代码



6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 5：编写代码
    - 定义图书接口和图书管理类
    - 在 src 目录下创建 bookManagement.ts 文件，编写以下代码

```typescript
import * as fs from 'fs';
import * as path from 'path';

// 定义图书接口
interface Book {
    id: number;
    title: string;
    author: string;
}

// 定义图书管理类
class BookManagement {
    private books: Book[] = [];
    private dataFilePath = path.join(__dirname, 'books.json');

    constructor() {
        this.loadBooksFromFile();
    }

    // 从文件中加载图书数据
    private loadBooksFromFile() {
        try {
            if (fs.existsSync(this.dataFilePath)) {
                const data = fs.readFileSync(this.dataFilePath, 'utf8');
                this.books = JSON.parse(data);
            }
        } catch (error) {
            console.error('读取图书数据文件时出错:', error);
        }
    }

    // 将图书数据保存到文件
    private saveBooksToFile() {
        try {
            const data = JSON.stringify(this.books, null, 2);
            fs.writeFileSync(this.dataFilePath, data, 'utf8');
        } catch (error) {
            console.error('保存图书数据文件时出错:', error);
        }
    }
```

6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 5：编写代码
    - 图书接口：Book 接口定义了图书的基本属性，包括 id、title 和 author。
    - 图书管理类：BookManagement 类封装了图书的添加、删除、查询和显示所有图书的功能，同时负责从文件中加载数据和将数据保存到文件。
    - 文件操作：loadBooksFromFile 方法用于从 books.json 文件中读取图书数据，saveBooksToFile 方法用于将图书数据保存到 books.json 文件。

```
    // 添加图书
    addBook(title: string, author: string): Book {
        const newBook: Book = {
            id: this.books.length > 0 ? this.books[this.books.length - 1].id + 1 : 1,
            title,
            author
        };
        this.books.push(newBook);
        this.saveBooksToFile();
        return newBook;
    }

    // 删除图书
    removeBook(id: number): boolean {
        const index = this.books.findIndex(book => book.id === id);
        if (index !== -1) {
            this.books.splice(index, 1);
            this.saveBooksToFile();
            return true;
        }
        return false;
    }

    // 查询图书
    findBookById(id: number): Book | undefined {
        return this.books.find(book => book.id === id);
    }

    // 显示所有图书
    getAllBooks(): Book[] {
        return this.books;
    }
}

export { Book, BookManagement };
```

6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 5：编写代码
    - 编写 main.ts 文件以支持命令行操作
    - commander 库的使用：commander 库用于解析命令行参数。我们定义了四个命令：add、remove、find 和 list，分别对应添加图书、删除图书、查询图书和显示所有图书的操作。
    - 命令的执行：每个命令都有一个对应的 action 函数，当用户在命令行中输入相应的命令时，会执行该函数并调用 BookManagement 类的相应方法。

```typescript
import { program } from 'commander';
import { Book, BookManagement } from './bookManagement';

// 创建图书管理实例
const bookManager = new BookManagement();

// 配置命令行选项
program
    .version('1.0.0')
    .description('简单的图书管理系统命令行工具');

// 添加图书命令
program
    .command('add <title> <author>')
    .description('添加一本新图书')
    .action((title, author) => {
        const newBook = bookManager.addBook(title, author);
        console.log('成功添加图书：', newBook);
    });

// 删除图书命令
program
    .command('remove <id>')
    .description('根据 ID 删除图书')
    .action((id) => {
        const isRemoved = bookManager.removeBook(Number(id));
        if (isRemoved) {
            console.log('成功删除图书，ID 为：', id);
        } else {
            console.log('未找到 ID 为', id, '的图书');
        }
    });

// 查询图书命令
program
    .command('find <id>')
    .description('根据 ID 查询图书')
    .action((id) => {
        const foundBook = bookManager.findBookById(Number(id));
```

6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 5：编写代码
    - 编写 main.ts 文件以支持命令行操作
    - commander 库的使用：commander 库用于解析命令行参数。定义了四个命令：add、remove、find 和 list，分别对应添加图书、删除图书、查询图书和显示所有图书的操作。
    - 命令的执行：每个命令都有一个对应的 action 函数，当用户在命令行中输入相应的命令时，会执行该函数并调用 BookManagement 类的相应方法。

```
        console.log('成功添加图书：', newBook);
    });

// 删除图书命令
program
    .command('remove <id>')
    .description('根据 ID 删除图书')
    .action((id) => {
        const isRemoved = bookManager.removeBook(Number(id));
        if (isRemoved) {
            console.log('成功删除图书，ID 为：', id);
        } else {
            console.log('未找到 ID 为', id, '的图书');
        }
    });

// 查询图书命令
program
    .command('find <id>')
    .description('根据 ID 查询图书')
    .action((id) => {
        const foundBook = bookManager.findBookById(Number(id));
        if (foundBook) {
            console.log('查询到的图书：', foundBook);
        } else {
            console.log('未找到 ID 为', id, '的图书');
        }
    });

// 显示所有图书命令
program
    .command('list')
    .description('显示所有图书')
    .action(() => {
        const allBooks = bookManager.getAllBooks();
        console.log('所有图书：', allBooks);
    });

// 解析命令行参数
program.parse(process.argv);
```

6. 使用TypeScript开发项目

- 实现一个简单的图书管理系统。支持以命令行的方式实现图书的添加、删除、查询以及显示所有图书的功能。
  - 步骤 6：编译并运行项目

```bash
npx tsc
```

  - 步骤 7：运行编译后的 JavaScript 代码，使用不同的命令进行操作

```bash
# 添加图书
node dist/main.js add "深入理解计算机系统" "Randal E. Bryant"
# 删除图书
node dist/main.js remove 1
# 查询图书
node dist/main.js find 2
# 显示所有图书
node dist/main.js list
```

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