课程大纲

- 搭建 TypeScript 开发环境。
- 掌握 TypeScript 的基础类型、联合类型和交叉类型。
- 详解类型断言的作用和用法。
- 详解 TypeScript 中函数、类中的类型声明方式。
- 掌握类型别名、接口的作用和定义。
- 掌握泛型的应用场景，熟练应用泛型。
- 灵活运用条件类型、映射类型与内置类型。
- 创建和使用自定义类型。
- 理解命名空间、模块的概念已经使用场景。
- 详解 TS 中的类型保护，装包拆包
- 巧妙运用类型推导简化代码。
- 深入理解 TypeScript 类型层级系统。
- 详解函数的协变与逆变。
- 深入研究 infer 的用法与技巧。
- 详解模板字符串类型。
- 灵活编写与运用类型声明文件，扩展 TypeScript 的类型系统。
- 详解 TS 中类型文件查找规则。
- 熟练使用装饰器，运用反射元数据扩展装饰器的功能，实现控制反转、依赖注入。
- 深度解析 TSConfig 配置文件。
- TS 类型体操
- 实现一个完整的 Axios 库

1.Typescript基础

> 目前大部分企业的中大型前端项目都采用了 Typescript，那么为什么我们需要它?

JavaScript 的核心特点就是灵活，但随着项目规模的增大，灵活反而增加开发者的心智负担。例如在代码中一个变量可以被赋予字符串、布尔、数字、甚至是函数，这样就充满了不确定性。而且这些不确定性可能需要在代码运行的时候才能被发现，**所以我们需要类型的约束**。

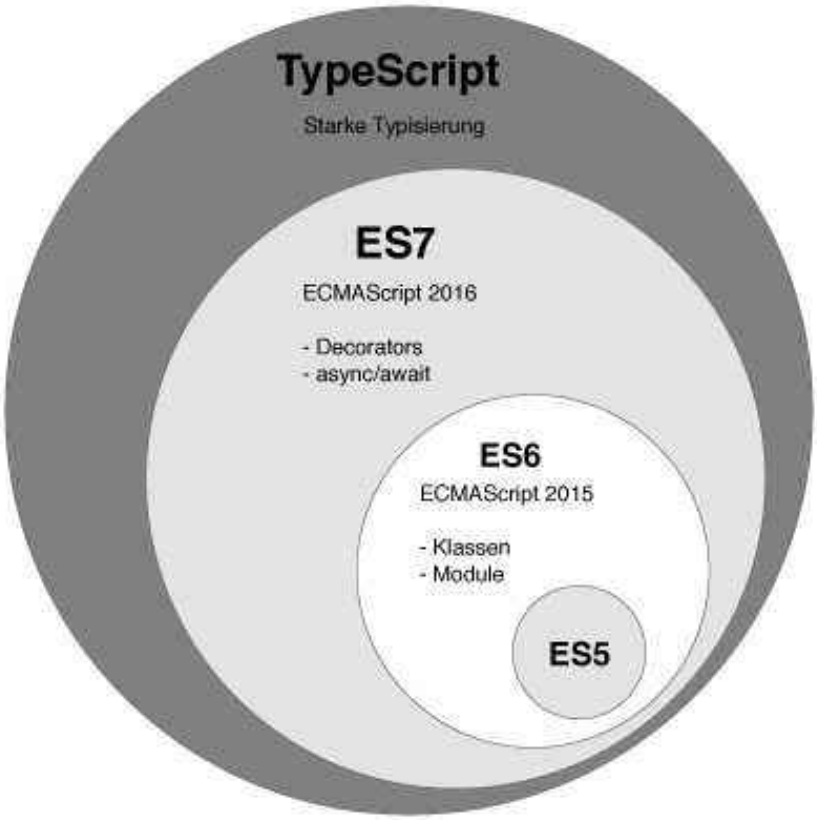
当然不可否认的是有了类型的加持多少会影响开发效率，但是可以让大型项目更加健壮

- Typescript 更像后端 JAVA，让`JS`可以开发大型企业应用；
- TS 提供的类型系统可以帮助我们在写代码时提供丰富的语法提示；
- 在编写代码时会对代码进行类型检查从而避免很多线上错误；

> 越来越多的项目开始拥抱 TS 了，典型的 Vue3、Pinia、第三方工具库、后端 NodeJS 等。我们也经常为了让编辑器拥有更好的支持去编写**.d.ts 文件**。

1. 什么是 Typescript

TypeScript 是一门编程语言，`TypeScript`是`Javascript`的超集（任何的`JS`代码都可以看成`TS`代码），同时`Typescript`扩展了`Javascript`语法添加了静态类型支持以及其他一些新特性。



> TypeScript 代码最终会被编译成 JavaScript 代码，以在各种不同的运行环境中执行。

2.环境配置

2-1.全局编译 TS 文件

全局安装`typescript`对`TS`进行编译

```
npm install typescript -g
tsc --init # 生成tsconfig.json
```

```
tsc # 可以将ts文件编译成js文件
tsc --watch # 监控ts文件变化生成js文件
```

2-2 ts-node 执行 TS 文件

> 采用`vscode code runner插件`运行文件

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

直接右键运行当前文件快速拿到执行结果。

2-3.配置`rollup`开发环境

- 安装依赖

  ```
  pnpm install rollup typescript rollup-plugin-typescript2 @rollup/plugin-node-resolve rollup-plugin-serve -D
  ```

- 初始化`TS`配置文件

  ```
  npx tsc --init
  ```

- `rollup`配置操作`rollup.config.mjs`

  ```
  import ts from "rollup-plugin-typescript2";
  import { nodeResolve } from "@rollup/plugin-node-resolve";
  import serve from "rollup-plugin-serve";
  import path from "path";
  import { fileURLToPath } from "url";
  const __filename = fileURLToPath(import.meta.url);
  ```

```
const __dirname = path.dirname(__filename);
export default {
  input: "src/index.ts",
  output: {
    format: "iife",
    file: path.resolve(__dirname,
"dist/bundle.js"),
    sourcemap: true,
  },
  plugins: [
    nodeResolve({
      extensions: [".js", ".ts"],
    }),
    ts({
      tsconfig: path.resolve(__dirname,
"tsconfig.json"),
    }),
    serve({
      open: true,
      openPage: "/public/index.html",
      port: "3000",
    }),
  ],
};
```

- `package.json`配置

```
"scripts": {
    "start": "rollup -c -w"
}
```

> 我们可以通过`npm run start`启动服务来使用 typescript 啦~

3.常用插件

- Error Lens 提示错误插件
- TypeScript 内置配置 （code->首选项->settings）根据需要打开设置即可。

4.基础类型

TS 中有很多类型：内置的类型 （DOM、Promise 等都在 typescript 模块中） 基础类型、高级类型、自定义类型。

TS 中冒号后面的都为类型标识，等号后面的都是值。

- ts 类型要考虑安全性，一切从安全角度上触发。
- ts 在使用的时候程序还没有运行
- ts 中有类型推导， 会自动根据赋予的值来返回类型，只有无法推到或者把某个值赋予给某个变量的时候我们需要添加类型。

4-1.布尔、数字、字符串类型

```
let name: string = "Jiang"; // 全局也有name属性，需要采用模块化解决冲突问题
let age: number = 30;
let handsome: boolean = true;
```

> 我们标识类型的时候 原始数据类型全部用小写的类型，如果描述实例类型则用大写类型（大写类型就是**装箱类型**，其中也包含**拆箱类型**）

```
let s1: string = "abc";
let s2: string = new String("abc"); // 不支持
let s3: String = new String("abc");
let s4: String = "abc";
```

什么是包装对象?

我们在使用原始数据类型时，调用原始数据类型上的方法，默认会将原始数据类型包装成对象类型。

4-2.数组

数组用于储存多个相同类型数据的集合。 TypeScript 中有两种方式来声明一个数组类型

```
let arr1: number[] = [1, 2, 3];
let arr2: string[] = ["1", "2", "3"];
let arr3: (number | string)[] = [1, "2", 3]; // 联合类型
let arr4: Array<number | string> = [1, "2", 3]; // 后面讲泛型的时候 详细说为什么可以这样写
```

4-3.元组类型

元组的特点就 固定长度 固定类型的一个数组

```
let tuple1: [string, number, boolean] = ["jw", 30, true];
tuple[3]; // 长度为 "3" 的元组类型 "[string, number, boolean]" 在索引 "3" 处没有元素。
let tuple2: [name: string, age: number, handsome?: boolean] = ["jw", 30, true]; // 具名元祖
```

```
let tuple3: [string, number, boolean] = ["jw", 30, true];
tuple3.push("回龙观"); // ✅ 像元组中增加数据，只能增加元组中存放的类型，但是为了安全依然无法取到新增的属性
// tuple3.push({ address: "回龙观" }); // ❎

let tuple4: readonly [string, number, boolean] = ["jw", 30, true];
// 仅读元祖，不能修改，同时会禁用掉修改数组的相关方法
```

> 我要求媳妇有车有房，满足即可（底线），有可能我媳妇还有钱，but 这个钱不能花，因为不知道有没有。

4-4.枚举类型

枚举可以看做是自带类型的对象，枚举的值为数字时会自动根据第一个的值来递增，枚举中里面是数字的时候可以反举。

```
enum USER_ROLE {
  USER, // 默认从0开始
  ADMIN,
  MANAGER,
}
// {0: "USER", 1: "ADMIN", 2: "MANAGER", USER: 0,
ADMIN: 1, MANAGER: 2}
```

可以枚举，也可以反举

```
// 编译后的结果
(function (USER_ROLE) {
  USER_ROLE[(USER_ROLE["USER"] = 0)] = "USER";
  USER_ROLE[(USER_ROLE["ADMIN"] = 1)] = "ADMIN";
  USER_ROLE[(USER_ROLE["MANAGER"] = 2)] =
"MANAGER";
})(USER_ROLE || (USER_ROLE = {}));
```

异构枚举

```
enum USER_ROLE {
  USER = "user",
  ADMIN = 1,
  MANAGER, // 2
}
```

常量枚举

```
const enum USER_ROLE {
  USER,
  ADMIN,
  MANAGER,
}
console.log(USER_ROLE.USER); // console.log(0 /*
USER */);
```

4-5.null 和 undefined

任何类型的子类型，如果`TSconfig`配置中`strictNullChecks`的值为 true，则不能把 null 和 undefined 赋给其他类型。

```
let u1: undefined = undefined;
let n1: null = null; // 默认情况下 只能null给null ，
undefiend给undefiend
```

```
let name1: number | boolean;
name1 = null;
name1 = undefined; // 非严格模式
```

4-6.void 类型

只能接受 null，undefined。void 表示的是空 (通常在函数的返回值中里来用)；undefiend 也是空，所以 undefiend 可以赋值给 void。严格模式下不能将 null 赋予给 void。

```
function fn1() {}
function fn2() {
  return;
}
function fn3(): void {
  return undefined;
}
```

4-7.never 类型

任何类型的子类型，never 代表不会出现的值（这个类型不存在）。不能把其他类型赋值给 never。

```
function fn(): never {
  //   throw new Error();
  while (true) {}
}
let a: never = fn(); // never只能赋予给never
let b: number = a; // never是任何类型的子类型，可以赋值给任何类型
```

never 实现完整性保护

```
function validate(type: never) {} // 类型“boolean”的参数不能赋给类型“never”的参数。
function getResult(strOrNumOrBool: string | number | boolean) {
  if (typeof strOrNumOrBool === "string") {
    return strOrNumOrBool.split("");
  } else if (typeof strOrNumOrBool === "number") {
    return strOrNumOrBool.toFixed(2);
  }
  // 能将类型“boolean”分配给类型“never”。
  validate(strOrNumOrBool);
}
```

联合类型自动去除 never

```
let noNever: string | number | boolean | never = 1; // never自动过滤
```

4-8.object 对象类型

`object`表示非原始类型

```
let create = (obj: object) => {};
create({});
create([]);
create(function () {});
```

这里要注意不能使用大写的 Object 或 {} 作为类型，因为万物皆对象（涵盖了原始数据类型）。

object、Object、{} 的区别

- `object`非原始类型；
- `Object`所有值都可以赋予给这个包装类型；
- `{}`字面量对象类型；

4-9.Symbol 类型

Symbol 表示独一无二

```
const s1 = Symbol("key");
const s2 = Symbol("key");
console.log(s1 == s2); // 此条件将始终返回 "false"，因为类型 "typeof s11" 和 "typeof s12" 没有重叠
```

4-10.BigInt 类型

```
const num1 = Number.MAX_SAFE_INTEGER + 1;
const num2 = Number.MAX_SAFE_INTEGER + 2;
console.log(num1 == num2); // true

let max: bigint = BigInt(Number.MAX_SAFE_INTEGER);
console.log(max + BigInt(1) === max + BigInt(2));
```

> `number`类型和`bigInt`类型是不兼容的

4-11.any 类型

不进行类型检测，一旦写了 any 之后任何的校验都会失效。声明变量没有赋值时默认为 any 类型，写多了 any 就变成 AnyScript 了，当然有些场景下 any 是必要的。

```
let arr: any = ["jw", true];
arr = "回龙观";
```

> 可以在 any 类型的变量上任意地进行操作，包括赋值、访问、方法调用等等，当然出了问题就要自己负责了。

5.变量类型推断

TypeScript 的类型推断是根据变量的初始化值来进行推断的。如果声明变量没有赋予值时默认变量是`any`类型。

```
let name; // 类型为any
name = "jiangwen";
name = 30;
```

声明变量赋值时则以赋值类型为准

```
let name = "jiangwen"; // name被推导为字符串类型
name = 30;
```

6.联合类型

在使用联合类型时，没有赋值只能访问联合类型中共有的方法和属性。

```
let name: string | number; // 联合类型
console.log(name.toString()); // 公共方法
name = 30;
console.log(name.toFixed(2)); // number方法
name = "jiangwen";
console.log(name.toLowerCase()); // 字符串方法
```

6-1.字面量联合类型

```
// 通常字面量类型与联合类型一同使用
type Direction = "Up" | "Down" | "Left" | "Right";
let direction: Direction = "Down";
```

> 可以用字面量当做类型，同时也表明只能采用这几个值（限定值）。类似枚举。

6-2.对象的联合类型

```
type women =
  | {
      wealthy: true;
      waste: string;
    }
  | {
      wealthy: false;
```

```
    morality: string;
  };

let richWoman: women = {
  wealthy: true,
  waste: "不停的购物",
  morality: "勤俭持家", // 对象类型的互斥
};
```

> 可以实现对象中的属性互斥。

7.类型断言

将变量的已有类型更改为新指定的类型，默认只能断言成包含的某个类型。

- 非空断言

```
let ele: HTMLElement | null =
document.getElementById("#app");
console.log(ele?.style.color); // JS中链判断运算符
ele!.style.color = "red"; // TS中非空断言ele元素一定有值
```

  - 可选链操作符 `?.` 在访问对象的属性或方法时，先检查目标对象及其属性是否存在。
  - 空值合并操作符 `??` ，当左侧的表达式结果为 `null` 或 `undefined` 时，会返回右侧的值。

- 类型断言

```
let name: string | number;
(name! as number).toFixed(2); // 强制
(<number>name!).toFixed(2);

name as boolean; // 错误 类型 "string | number" 到类型 "boolean" 的转换可能是错误的
```

> 尽量使用第一种类型断言因为在 React 中第二种方式会被认为是`jsx`语法

- 双重断言

```
let name: string | boolean;
name! as any as string;
```

> 尽量不要使用双重断言，会破坏原有类型关系，断言为 any 是因为 any 类型可以被赋值给其他类型。

8.函数类型

函数的类型就是描述了**函数入参类型与函数返回值类型**

- 通过 function 关键字来进行声明

```
function sum(a: string, b: string): string {
  return a + b;
}
sum("a", "b");
```

> 可以用来限制函数的参数和返回值类型

- 通过表达式方式声明

```
type Sum = (a1: string, b1: string) => string;
let sum: Sum = (a: string, b: string) => {
  return a + b;
};
```