

4-3 直播 TypeScript

模块问题答疑

上手 TypeScript

相关概念

强类型与弱类型

静态类型与动态类型

小结

TypeScript 简介

安装与编译

常规安装

通过 Vue CLI 安装

依赖项

配置文件

shim 文件

在项目中使用 TypeScript

语法

类型注解

原始类型

函数

函数表达式

类型推断与类型断言

类型推断

类型断言

对象

接口

索引签名

数组

元组

枚举

类

访问修饰符

接口与类

抽象类

泛型

结语

内容介绍：

- 模块问题答疑
- TypeScript

直播代码：演示使用了在线编辑器，语法代码见文档后续内容。

模块问题答疑

1. 实际工作中，框架开发的任务安排是怎样的？会不会有像老师您这样的文档？
 - a. 实际工作中，任务是你的上级安排的，不同公司的安排方式没有统一标准的。
 - b. 但可以确定的是，不可能有详细的文档，就算有，这个文档也是你自己写的。
2. 如何以这个虚拟项目作面试准备，可以的话，说一下细节。
 - a. 首先要找准一个功能点来说
 - i. 例如你想说课程功能，那就只说课程，或课程中的某一块功能，没必要一会儿说课程一会儿说用户，言多必失。
 - b. 取精华而不是全部都介绍。
 - i. 例如我用了 @click 绑定事件，通过 methods 存储事件处理程序，这种语法层面的不要说，除非面试官问你事件绑定方式有哪些，不然自己说这种内容没有亮点。
 - ii. 找一些你做项目中认为比较困难（且完全克服了）的功能来讲解，一是印象深不容易出问题，二是你可以抛出问题询问面试官有没有更好的解法，将面试变成技术交流，值回路费（开玩笑）。
 - iii. 以上为常规情况，还是多准备一些内容以备面试官不按套路出牌。技术硬才是王道。
3. 再解析一下 index.scss 的设置
 - a. index.scss 的设置方式 `prependData: `@import "~@/styles/variables.scss";``

其实就是给 webpack 添加一条命令，在每个 css 文件打包的时候添加上面的一句代码，省的自己手写了。
 - b. 具体配置可以通过 `vue inspect > output.js` 输出 webpack 的配置看到。
4. 最后一步打包部署：npm run build，然后进入到dist目录通过server服务器访问。真实项目中，前端开发打包完成后，代码是提交到git上，后端直接获取，然后他们部署到服务器上？还是什么样的操作流程之后用户可以访问页面？
 - a. 部署的问题本质与前端无关，但基本流程应该清楚，前端将打包结果（dist）通过提交到 github 或任意方式将代码给到相应的负责人，最终你的 dist 被后端、运维部署到服务器上被用户访问。

5. 项目搭建的时候，在main.js将App.vue组件绑定到id为app的div元素上，APP.vue中有个id是app的元素，public目录中index.html中也有一个id为app的元素，绑定的是哪个，这个index.html文件用途是啥？不是很理解界面加载流程，比如main.js是何时被加载的，没看到哪个文件有导入的动作
- 绑定的是 public/index.html 里面的 div#app，是在 Vue CLI 内容中讲解的。
 - 可以通过 vue inspect 命令看到 HtmlWebpackPlugin 的配置情况，里面写了具体路径，以及入口 entry 配置的 main.js 的路径。
 - 脚手架工具 Vue CLI 的意义是为了减轻我们使用打包工具的操作（目的是让常规配置自动化），如果希望清晰的知道打包时每一步都做了什么，可以直接通过 Webpack 打包 Vue 文件，但那是非常复杂的。所以 Vue 才推出了针对 Vue 的脚手架工具。
 - 言归正传，我们还是要更好的去理解学习每一个功能、工具的意义。

上手 TypeScript

官方文档: <https://www.typescriptlang.org/>

中文手册: <https://typescript.bootcss.com/>

首先我们来思考，为什么要使用 TypeScript？

```
> data.push(6)
✖ ▶ Uncaught TypeError: data.push is not a function
   at <anonymous>:1:6

> data.title.slice(1)
✖ ▶ Uncaught TypeError: Cannot read property 'slice' of undefined
   at <anonymous>:1:12
```

上图的错误相信大家不会陌生，没错，这就是每个前端同学最常见的报错类型，TypeError。

TypeScript 作为 JavaScript 的超集，提供了强大的类型系统以解决此问题。

当然除了类型系统外，TS 还提供了许多其他功能，具体可参见文档。

简而言之，TypeScript 可以大大提高代码的可靠程度。

相关概念

在讲解 TypeScript 之前，我们先来回顾一下 JavaScript 关于类型的相关概念。

类型的描述通常从两个维度进行：

- 类型安全
 - 强类型
 - 弱类型
- 类型检测
 - 静态类型
 - 动态类型

需要提及的是，类型安全的语言并不等于类型安全的程序，反之亦然。我们完全可以通过类型不安全的编程语言书写出类型安全的程序，所以，语言的“安全”是相对的，实际安全的核心还是在于开发者自身。

强类型与弱类型

强类型：

- 变量在声明时必须明确类型，且不能隐式转换为不相关类型，具有更强的类型约束。

弱类型：

- 变量在声明时无须设置类型，可以进行任意的隐式转换，几乎没有类型约束。

JavaScript 是弱类型语言，所以变量声明使用的 `var`、`let`、`const` 均不会限制变量类型，变量更改为任意类型而不会出现语法错误。

```
1 let num = 10
2 if (num) {
3   console.log('会执行')
4 }
```

需要注意的是，强类型弱类型指的都是语言层面的类型约束，也就是在语言的编译阶段执行的类型安全检查。类似最开始我们演示的 `data.push()` 这种报错，已经是处于运行阶段出现的功能性异常，与强弱类型的类型安全检查无关。

静态类型与动态类型

静态类型：

编译阶段确定类型，且不可更改。

动态类型：

运行阶段实时确定类型，因为变量没有实际类型，类型取决于变量当前存储的数据的类型。

```
1 let num = 10
2 num = 'Hello'
```

小结

JavaScript 是弱类型语言，是动态类型语言。

这两大特性使得 JavaScript 代码的灵活度很高，但随之而来的是代码可靠性的降低。早期设计 JavaScript 只是用来完成非常简单的功能，这种灵活写法可以降低学习与开发难度。但发展到现阶段，这些特性会增加大型的 JavaScript 项目的开发与维护成本。

为了能够增强代码的可靠性，我们可以为 JavaScript 引入类型系统，现阶段最终解决方案就是 TypeScript。

TypeScript 的类型系统（强类型）具有以下作用：

- 更早暴露错误

```
1 let num = 10
2 num.push(2) // 类型系统在编写代码时即可检测错误，而不是在运行后才发现
```

- 代码提示更加智能（如，参数类型提示）

```
1 function fn (value) {
2   value // 使用时由于编辑器无法明确 value 类型，所以无法智能提示对应类型的属性
3   方法
4 }
4 fn([1, 2, 3]) // 调用时编辑器无法提示类型，可能导致传参类型错误导致功能异常
```

- 减少代码中不必要的类型判断
 - 以前为了代码健壮性，我们会在使用关键数据时提前检测功能，类型系统可以帮我们完成这一步骤。
- 代码重构更方便
 - 修改代码功能后，其它依赖于当前功能的代码会提示错误，而无需自己查找。

TypeScript 简介

TypeScript 是 JavaScript 的超集，是由微软开发的一门编程语言。

TS 的功能是基于 JavaScript 实现的，内部支持 JS 功能以及**静态类型系统**，并对 ES6+ 功能进行了支持。

最终通过编译得到处理后的 JavaScript 文件。



学习 TypeScript 可以给我们带来好处而同时不会造成太大的负担，因为 TypeScript 的学习与 Vue 类似，是“渐进式”的，就算你完全不会 TS 语法，也可以使用 JS 语法来编写 TS 文件；通过不断学习再将代码迁移到 TS。

Vue 3.0 都采用 TypeScript 实现，你不学一套么？

安装与编译

TypeScript 使用以 .ts 为后缀的文件，但此文件无法直接应用（如浏览器中、Node.js 中），而是需要通过编译器进行编译（类似 Less）。

常规安装

通过 npm 命令安装 TypeScript 的编译器。

- 可以自己选择全局安装或仅安装在当前项目下

```
1 npm install -g typescript
```

安装 TypeScript 编译器后，可以通过编译器对 .ts 文件进行编译，编译后会得到同名的 .js 后缀文件。

- 如果为本地项目安装，则该命令只能在项目下使用
- 由 node_modules/.bin/tsc 支持

```
1 tsc xxx.ts
```

通过 Vue CLI 安装

首先，通过 Vue CLI 创建项目，并手动选择功能，在功能列表中勾选 TypeScript

```
1 ? Please pick a preset: Manually select features
2 ? Check the features needed for your project:
3 (*) Choose Vue version
4 (*) Babel
5 >(*) TypeScript
6 ( ) Progressive Web App (PWA) Support
7 ( ) Router
8 ( ) Vuex
9 ( ) CSS Pre-processors
10 (*) Linter / Formatter
11 ( ) Unit Testing
12 ( ) E2E Testing
```

勾选 class 风格组件语法。

```
1 ? Use class-style component syntax? (Y/n) y
```

让 Babel 与 TS 结合编译 JavaScript。

- TS 本身就包含将 ES6 编译为 ES3 的功能，Babel 也支持，不过编译结果为 ES5。
- 此选项表示 TS 编译器只处理 TS 转 JS 的功能，而语法降级还是通过 Babel 完成。

```
1 ? Use Babel alongside TypeScript (required for modern mode, auto-detected polyfills, transpiling JSX)? (Y/n) Y
```

完成，可以发现以前项目中的 .js 文件，都变成了 .ts 文件。

注：如果是已有项目想添加 TypeScript，可添加 Vue 官方提供的 TypeScript 适配插件。

```
1 vue add @vue/typescript
```

依赖项

dependencies 生产依赖：

依赖项	说明
vue-class-component	支持提供 Class 风格的 Vue 组件
vue-property-decorator	用于增强 Class 装饰器功能

devDependencies 开发依赖：

依赖项	说明
@typescript-eslint/eslint-plugin	通过 ESLint 校验 TS 代码
@typescript-eslint/parser	将 ES 转换为 AST 供 ESLint 校验使用
@vue/cli-plugin-eslint	Vue CLI 处理 TS 的核心插件
@vue/eslint-config-typescript	为 ESLint 提供 TS 的校验规则
typescript	TypeScript 编译器

配置文件

位于项目根目录下的 tsconfig.js 是 ts 的配置文件，并且提供了一些初始配置。（具体配置见文档）

- 非 Vue CLI 项目可手动通过 `tsc --init` 命令创建

```
1 {
2   "compilerOptions": {
3     "target": "es5",
4     "module": "esnext",
5     "strict": true,
6     "jsx": "preserve",
7     "importHelpers": true,
8     "moduleResolution": "node",
9     "experimentalDecorators": true,
10    "skipLibCheck": true,
11    "esModuleInterop": true,
12    "allowSyntheticDefaultImports": true,
13    "sourceMap": true,
14    "baseUrl": ".",
15    "types": [
16      "webpack-env"
17    ],
18    "paths": {
19      "@/*": [
20        "src/*"
21      ]
22    },
23    "lib": [
24      "esnext",
25      "dom",
26      "dom.iterable",
27      "scripthost"
28    ]
29  },
30  "include": [
31    "src/**/*.ts",
32    "src/**/*.tsx",
33    "src/**/*.vue",
34    "tests/**/*.ts",
35    "tests/**/*.tsx"
36  ],
37  "exclude": [
```



```
38     "node_modules"
39   ]
40 }
```

shim 文件

在 src 目录新增了两个文件：

- shims-tsx.d.ts
- shims-vue.d.ts

由于 TS 本身不支持 .vue 后缀的文件，所以通过 shims-vue.d.ts 文件进行了中转处理。

```
1 // shims-vue.d.ts
2 declare module '*.vue' {
3   import Vue from 'vue'
4   export default Vue
5 }
```

该段功能用于声明所有以 .vue 结尾的文件的类型都为 Vue 类，这样加载 .vue 文件时就不会报错了。同理，shims-tsx.d.ts 文件用于处理 tsx 文件（JSX 对应的类型），补充了一些类型声明，避免报错。

在项目中使用的 TypeScript

由于我们的项目初始并未添加 TypeScript，所以这里需要自己安装相关的包。

1. 执行 `vue add @vue/typescript` 命令，安装相关环境。

```
1 $>vue add @vue/typescript
2 00 Installing @vue/cli-plugin-typescript...
3
4 + @vue/cli-plugin-typescript@4.5.11
5 added 49 packages from 42 contributors and audited 1526 packages
  in 16.019s
6
7 83 packages are looking for funding
8   run `npm fund` for details
9
10 ✓ Successfully installed plugin: @vue/cli-plugin-typescript
11
12 ? Use class-style component syntax? Yes
```

```

13     使用 class 风格的组件语法
14 ? Use Babel alongside TypeScript (required for modern mode, auto-
    detected polyfills, transpiling JSX)? Yes
15     让 Babel 与 TS 结合编译
16 ? Convert all .js files to .ts? Yes
17     将 .js 文件转换为 .ts
18 ? Allow .js files to be compiled? Yes
19     允许 .js 文件被编译
20 ? Skip type checking of all declaration files (recommended for ap
    ps)? Yes
21     跳过声明文件 (xxx.d.ts) 的类型检查
22
23 00 Invoking generator for @vue/cli-plugin-typescript...
24 00 Installing additional dependencies...
25
26 added 17 packages from 8 contributors and audited 1543 packages i
    n 21.693s
27
28 87 packages are looking for funding
29 run `npm fund` for details
30
31 ⚙ Running completion hooks...
32
33 ✓ Successfully invoked generator for plugin: @vue/cli-plugin-typ
    escript

```

2. 重启静态资源服务器，成功。

a. 虽然运行时控制台出现一些报错，但其实都是语法检查导致的提示，不影响运行，这也体现了 TS 的渐进式用法。

```
1 npm run serve
```

3. 以 home 为例

```

1 <script>
2 export default {
3   name: 'HomeIndex',
4   data () {

```

```

5     return {
6         str: '内容' // 设置数据
7     }
8 },
9 methods: {
10     test () {
11         this.str.          // 使用数据，联想出的功能并不完全准备，经常啊导致
                             大家不小心选错了。
12         this.str = 10    // 没有类型约束，在程序中更改类型可能导致其他代码出错
                             ，但不运行无法发现。
13     }
14 }
15 }
16 </script>

```

这时我们给 `<script>` 标签添加 ts 支持。

```

1 // 添加 ts 支持
2 <script lang="ts">
3 export default {
4     name: 'HomeIndex',
5     data () {
6         return {
7             str: '内容'
8         }
9     },
10    methods: {
11        test () {
12            this.str.          // 功能联想更准确
13            this.str = 10    // 报错，赋值类型错误
14        }
15    }
16 }
17 </script>

```

这里仅为基本演示，下面我们来学习 TypeScript 的更多语法吧。

语法

学习语法推荐使用 TypeScript 官网的在线编辑器，可以自动实时将 TS 编译为 JS。

地址：<https://www.typescriptlang.org/play>

类型注解

通过 TypeScript 的类型系统，我们可以变量设置类型注解以进行类型约束。

```
1 let str: string = 'William'
2 str = 10 // 错误
```

原始类型

```
1 // 字符串
2 let myName: string = 'William'
3 myName = 'jack'
4 myName = 123 // 错误
5
6 // 数值
7 let myAge: number = 18
8 myAge = 123;
9 myAge = 123.45;
10 myAge = '123'; // 错误
11
12 // 布尔
13 let isMale: boolean = true
14 isMale = false;
15 isMale = 'false'; // 错误
16
17 // 给变量设置多种类型（可按需添加多个）
18 let value: number | string = 10
19 value = '123'
20 value = true // 错误
21
22
23 // 不限制类型（非特殊情况，不推荐）
24 let value: any = 10
25 value = 'abc'
26 value = true
27
```

```

28 // null 与 undefined 类型的变量，不常用
29 let un: undefined = undefined
30 let nu: null = null

```

如果给变量设置为错误类型，会报错。

```

// 字符串
let myName: string = 'William'
let myName: string
不能将类型“number”分配给类型“string”。
速览问题 (Alt+F8) 没有可用的快速修复
myName = 200

```

但在 TypeScript 中，null 与 undefined 是所有类型的子类型，所以以下设置不会报错（严格模式除外）。

```

// 字符串
let myName: string = 'William'
myName = null

```

函数

函数可通过类型注解设置参数与返回值类型。

```

1 function fn (a: number, b: string): boolean {
2   a.replace() // 错误，提示数值类型的 a 不应具有 replace()
3   b * 5 // 错误，提示 * 不能应用于 string
4   return 200 // 错误，返回值类型错误
5 }
6
7 fn(10) // 错误，缺少一个参数
8 fn(10, 20) // 错误，参数 b 类型错误
9 fn(10, 'a', 20) // 错误，传入多余参数

```

参数还可设置进行其他设置。

- any 类型代表该参数可为任意类型（不要随意使用）
- 参数名加 ? 表示参数可选。
 - 或者为参数设置默认值，也代表参数可选
- ...rest 用于处理后续参数，类型注解方式见数组部分

```

1 function fn (a: number, b: any, c?: string, ...rest:any) {
2   return 100
3 }
4 fn(10) // 错误, 缺少必选参数 b
5 fn(10, 20)
6 fn(10, 'abc')
7 fn(10, 20, 'abc')
8 fn(10, 20, 30) // 错误, 参数 c 类型错误

```

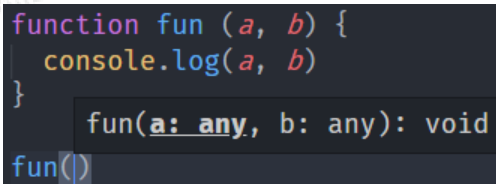
如果函数明确不存在返回值, 可设置为类型 void。

```

1 function fn (): void {
2   return 1 + 2 // 错误, 函数不应存在返回值
3 }
4 fn()

```

下面是一些演示:

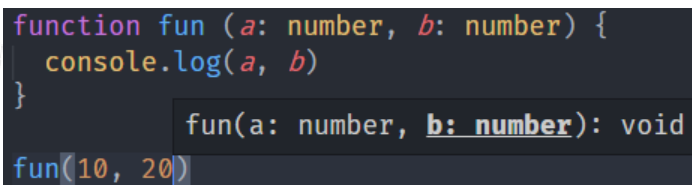


```

function fun (a, b) {
  console.log(a, b)
}
fun()

```

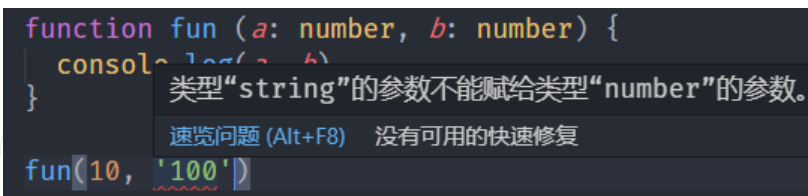
TS 添加类型注解: 当我们在任意位置调用函数时, 都可以明确类型, 避免问题。



```

function fun (a: number, b: number) {
  console.log(a, b)
}
fun(10, 20)

```



```

function fun (a: number, b: number) {
  console.log(a, b)
}
fun(10, '100')

```

函数表达式

```

1 let fn: (a: number, b: number) => boolean

```

```

2
3 fn = function (a: number, b: number):boolean {
4     return a > b
5 }

```

箭头函数写法，只是在赋值时有区别，

```

1 let fn: (a: number) => void
2
3 fn = (a: number):void => {
4     console.log('执行代码..')
5 }

```

类型推断与类型断言

类型推断

除了可以显示的添加类型注解外，TypeScript 也支持**隐式类型推断**。

- 当我们在声明时添加类型注解，TypeScript 会根据变量初始值的类型进行类型推断。

```

1 let str = 'William'
2 str = 10 // 错误，因为初始值为 string
3
4 let un // 没有声明初始值，TS 会认为变量可存储任意类型。
5 un = 10
6 un = 'a'

```

注意：尽管 TS 存在类型推断，但依然**建议为变量添加类型注解**！书写更加规范，可读性更强。

类型断言

有时候 TS 无法预测出程序的类型，这时我们就可以通过类型断言明确告知 TS 当前数据类型，以通过类型检查（谨慎使用）。

```

1 const numArr = [1, 2, 3]
2 const result = numArr.find(item => item > 0)
3 console.log(result * 5) // 错误，TS 认为 find 有可能返回 undefined，所以无法进行数值操作。
4

```

```

5 // 这时可以通过类型断言的方式通过 TS 检测
6 // - 格式为 数据 as 类型
7 const numArr = [1, 2, 3]
8 const result = numArr.find(item => item > 0) as number
9 console.log(result * 5) // 错误, TS 认为 find 有可能返回 undefined, 所以无法进行数值操作。
10
11 // 类型断言也可以写为 <类型>变量, 但会与 JSX 的语法冲突, 不建议使用。
12 const numArr = [1, 2, 3]
13 const result = numArr.find(item => item > 0) as number
14 console.log(result * 5) // 错误, TS 认为 find 有可能返回 undefined, 所以无法进行数值操作。

```

对象

TS 中的 `object` 类型泛指所有对象, 例如数组、函数等。

```

1 let obj: object = {
2     name: 'William'
3 }
4
5 obj = [1, 2, 3]
6 obj = function () {}
7 obj = new Date()
8
9 obj = 1 // 错误
10 obj = 'a', // 错误
11 obj = true // 错误

```

如果希望限制为 `Object` 类型, 可以书写为 `{}`, 内部可进行属性限制。

```

1 // 注意, 这仅仅是类型注解, 并不是赋值
2 let obj: {
3     name: string; // 可以写 , 或 ; 也可以不写
4     age: number;
5     readonly gender: string; // 只读属性
6     email?: string; // 可选属性
7 }

```



```

8
9 // 下面是单独的赋值演示，也可以与前面的代码合并，但太长不便于观察
10 obj = {
11     name: 'William',
12     age: 18,
13     gender: '男'
14 }
15 obj.name = 'jack'
16 obj.gender = '女' // 错误，不能修改只读属性
17
18
19 obj = {
20     name: 'William'
21     // 错误，缺少 age、gender 属性
22 }
23
24 obj = {
25     name: 'William',
26     age: 18,
27     gender: 1, // 错误，类型应为 string
28     email: 'xxx@xx.com'
29 }

```

方法定义：

```

1 // 注意，这仅仅是类型注解，并不是赋值
2 let obj: {
3     name: string;
4     age: number;
5     // 定义方法时，可定义方法名称与参数和返回值信息
6     sayHi(value: string) : boolean
7 }
8
9 // 下面是单独的赋值演示，也可以与前面的代码合并，但太长不便于观察
10 obj = {
11     name: 'William',
12     age: 18,
13     sayHi (value: string) {
14         console.log('你好' + value)

```

```

15         return true
16     }
17 }
18 console.log(obj)
19 obj.sayHi('很高兴认识你')

```

接口

接口是 TS 的一个核心功能，用于约定对象的结构。

```

1 // 例如，希望设置多个对象都具有 name: string; age: number，可通过接口统一
  说明
2 interface Obj {
3     name: string;
4     age: number
5 }
6
7 // 声明数据，将类型设置为指定接口，都可以享受对应的类型设置
8 let obj1: Obj = {
9     name: 'Jack',
10    age: 18
11 }
12
13 let obj2: Obj = {
14     name: 'Rose',
15     age: 21
16 }

```

结合函数参数使用：

```

1 interface Data {
2     title: string;
3     content: string;
4 }
5
6 function fn (data: Data): void {
7     data.title
8     data.content

```

```

9     data.other           // 错误, data 不具有 other 属性
10 }
11
12 fn({
13     title: '标题文本',
14     content: '内容文本',
15     other: '其他内容' // 错误, 参数不应存在 other 属性
16 })

```

索引签名

当我们在使用对象时，并不能总是在初期就规定好所有属性，但临时添加属性会导致报错。

```

1 interface Data {
2     name: string
3     age: number
4 }
5
6 let obj: Data = {
7     name: 'jack',
8     age: 18,
9     gender: '男' // 错误
10 }

```

这时如果需要添加额外的不确定属性，可以设置索引签名。

- 注意：
 - 索引前面中的 key 仅为语义，可自行更改为其他语义如 title、username 等。
 - key 后面的 string 表示属性名为字符串类型，也可使用 number，例如 obj[0]

```

1 interface Data {
2     [key: string]: any
3     name: string
4     age: number
5 }
6
7 let obj: Data = {
8     name: 'jack',
9     age: 18,

```

```

10     gender: '男',
11     phone: 15300000000
12 }
13
14 // number 类型的索引签名
15 interface Data {
16     [name: number]: string
17 }
18
19 let scoreData: Data = {
20     0: '张三',
21     1: '李四',
22     2: '王五',
23     four: 'jack' // 错误，属性名不能为字符串类型
24 }

```

注意，一旦明确了类型签名，则所有属性都需要符合索引签名（的类型）。

```

1 interface Data {
2     [attr: string]: string
3     name: string
4     age: number // 错误，类型应为 string
5 }
6 // 例如，当我们希望限制一个对象用于存储某一特定类型时使用非常方便。
7 interface Data {
8     [score: string]: number
9 }
10
11 let scoreData: Data = {
12     jack: 10,
13     rose: 20,
14     lily: 30,
15     lucy: 40,
16     me: 'a' // 错误，所有属性都应为 number 类型
17 }

```

数组

TypeScript 通过对数组进行类型注解可以限制数组元素的类型。

```
1 // 方式1
2 let arr: Array<number> = [1, 2, 3]
3 arr.push(4)
4 arr.push('a')
5
6 // 方式2: 功能与方式1相同
7 let numArr: number[] = [1, 2, 3]
8 numArr.push(4)
9 numArr.push('a') // 错误
10
11 let strArr: string[] = ['a', 'b', 'c']
12 strArr[0] = 'x'
13 strArr[1] = 123 // 错误
14
15 let boolArr: boolean[] = [true, false]
16 boolArr = [false, false]
17 boolArr = [1, 2] // 错误
18 boolArr = 100 // 错误
```

我们也可以将数组类型与其他类型结合。

```
1 function fn (...arg: number[]) {
2   console.log(arg)
3 }
4 fn(1, 2, 3)
5 fn(1, 2, 'a') // 错误, 所有参数都应为 number 类型
```

元组

简单来说, 元组类似数组, 指的是一个确定了元素个数以及每个元素类型的数据结构。

```
1 let tuple: [number, string, string] = [1, 'a', 'b']
2 tuple[0] = 'a'
```

枚举

枚举类型在 JavaScript 中并不存在，是 TS 中添加的一种类型。枚举类型用于对一组关联数据进行统一存储。

在以往的开发中，这种功能都是通过对象来完成。

```
1 // 用于存储文章状态的数据，避免手写导致错误
2 let PublishStatus = {
3     Drafted: 0,
4     Unpublished: 1,
5     Published: 2
6 }
7
8 // 数据
9 const article = {
10     title: '今天天气晴朗，专家建议开窗通风',
11     content: '...',
12     status: PublishStatus.Drafted
13 }
```

通过 TS 提供的枚举类型可以更好的处理这种关联的数据集。

```
1 // 注意写法
2 enum PublishStatus {
3     Drafted = 0,
4     Unpublished = 1,
5     Published = 2
6 }
7
8 // 数据
9 const article = {
10     title: '今天天气晴朗，专家建议开窗通风',
11     content: '...',
12     status: PublishStatus.Drafted
13 }
14
15 PublishStatus.abc // 错误，无法访问 enum 中不存在的数据
16 PublishStatus.Drafted = 100 // 错误，数据只读
```

也可书写为以下形式：

```
1 // enum 会自动将属性设置为递增的值，默认从 0 开始
2 enum PublishStatus {
3     Drafted,          // 0
4     Unpublished,      // 1
5     Published         // 2
6 }
7
8 enum PublishStatus {
9     Drafted = 10,
10    Unpublished,      // 11
11    Published         // 12
12 }
13
14 // 如果设置为字符串值，则无法递增，所以设置了字符串值后，后续属性必须赋值
15 enum PublishStatus {
16     Drafted = 'a',
17     Unpublished = 'b',
18     Published = 'c'
19 }
```

以上为 enum 的基本使用方式。

最后，我们观察编译后的代码会发现 enum 的编译结果创建了一个可以双向访问的数据结构。

```

"use strict";
var PublishStatus;
(function (PublishStatus) {
    PublishStatus[PublishStatus["Drafted"] = 0] = "Drafted";
    PublishStatus[PublishStatus["Unpublished"] = 1] = "Unpublished";
    PublishStatus[PublishStatus["Published"] = 2] = "Published";
})(PublishStatus || (PublishStatus = {}));
// 数据
const article = {
    title: '今天天气晴朗，专家建议开窗通风',
    content: '...',
    status: PublishStatus.Drafted
};
PublishStatus.abc;

```

通过这种处理方式，enum 可以进行键与值的双向访问。

```

1 enum PublishStatus {
2     Drafted = 0,
3     Unpublished = 1,
4     Published = 2
5 }
6 ...
7
8 console.log(PublishStatus.Drafted) // 0
9 console.log(PublishStatus[0])      // 'Drafted'

```

如果没有这种访问需求，我们可以设置常量枚举。

```

1 // 添加 const 后，数据就不会进行双向处理了，具体按需决定
2 const enum PublishStatus {
3     Drafted = 0,
4     Unpublished = 1,
5     Published = 2
6 }

```


类

```
1 class Person {
2     // ES 2016 中定义实例属性的方式
3     name: string
4     age: number // = 10
5     constructor (name: string, age: number) {
6         // 定义属性后必须设置初始值或声明默认值，否则报错
7         this.name = name
8         this.age = age
9     }
10
11     sayHi (value: string): void {
12         console.log(`姓名${this.name}, 年龄${this.age}`)
13     }
14 }
```

访问修饰符

- private 私有属性，只能在类内部使用
- public 共有属性，默认即为 public，类内外均可使用
- protected 受保护的，只能在类内部或子类中访问。
- readonly 只读，如果同时存在前面三个修饰符，将 readonly 跟在后面即可。

```
1 class Person {
2     public readonly name: string
3     private age: number
4     protected gender: boolean
5     constructor (name: string, age: number, gender: boolean) {
6         this.name = name
7         this.age = age
8         this.gender = gender
9     }
10 }
11
12 // 创建 Person 实例
13 let p1 = new Person('William', 18, true)
```

```

14 console.log(p1.name)
15 console.log(p1.age)           // 错误，无法访问私有属性
16 console.log(p1.gender)       // 错误，无法访问受保护属性
17 this.name = '老王'           // 错误，无法修改只读属性
18
19 // 创建子类 Student
20 class Student extends Person {
21     constructor (name: string, age: number, gender: boolean) {
22         super(name, age, gender)
23         console.log(this.age)    // 错误，无法访问私有属性
24         console.log(this.gender) // 子类可以访问受保护属性
25     }
26 }

```

接口与类

```

interface Eat {
    eat (food: string): void
}

interface Run {
    run (distance: number): void
}

class Person implements Eat, Run {
    eat (food: string): void {
        console.log(`优雅的进餐: ${food}`)
    }

    run (distance: number) {
        console.log(`直立行走: ${distance}`)
    }
}

class Animal implements Eat, Run {
    eat (food: string): void {
        console.log(`呼噜呼噜的吃: ${food}`)
    }

    run (distance: number) {
        console.log(`爬行: ${distance}`)
    }
}

```

抽象类

抽象类是对类的一种特性描述，例如某些类专门用于被继承，无需实例，就应当设置为抽象类。抽象类更多的可以理解为一种语义化的产物，规范操作，避免不必要的实例化操作。

```

abstract class Animal {
  eat (food: string): void {
    console.log(`呼噜呼噜的吃: ${food}`)
  }

  abstract run (distance: number): void
}

class Dog extends Animal {
  run(distance: number): void {
    console.log('四脚爬行', distance)
  }
}

const d = new Dog()
d.eat('嗯西马')
d.run(100)

```

泛型

泛型的概念可以让某段功能的数据在不同类型组合间进行切换，增加灵活性。

```

1 // 例如，存在函数 A 用于对数值进行操作
2 function fnNum (a:number, b: number): void {
3   console.log(a, b)
4 }
5
6 // 后续我们由需要一组类似功能但用于处理字符串
7 function fnStr (a:number, b: number): void {
8   console.log(a, b)
9 }
10
11 // 尽管功能相同，但由于类型限制不同，无法直接合并为一个函数，这时就可以通过泛型
    解决
12 function fnAny<T> (a:T, b: T): void {
13   console.log(a, b)
14 }
15
16 fnAny<number>(1, 2)

```

```
17 fnAny<string>('a', 'b')
18 fnAny<string>('a', 1)    // 错误, 参数2违反泛型约定的类型
```

通过上面的示例可以发现, 泛型让类型从“声明时定义”变为了“调用时定义”, 大大提高了代码灵活性。而泛型结构中的 T 指的是类型的“占位符”, 我们可以根据实际需求进行自定义, 但一般建议以 T 开头, 例如 Tkey、Treturn 等等。

当然, 如果需要, 泛型中可以设置多个类型, 名称可自行定义。

```
1 function fnAny<T, U> (a:T, b: U): void {
2   console.log(a, b)
3 }
4
5 fnAny<number, number>(1, 2)
```

结语

上述语法内容为 TypeScript 中的常用功能, 在 TypeScript 中还有其他功能, 可自行通过文档查询。阅读文档应是前端开发者或任何技术开发人员的基础能力, 大家闲暇之余也应自行尝试编写技术文档, 有利于思路的梳理与功能的掌握。

Good Hunting!