**TypeScript 基础**

**[该类型的内容暂不支持下载]**

|  |
| --- |
| [**⼀、TypeScript 简介**](#heading_0)  [**⼆、为何需要 TypeScript**](#heading_1)  [**1. JavaScript 当年诞⽣时的定位是浏览器脚本语⾔，⽤于在⽹⻚中嵌⼊简单的逻辑，且代码量很少。**](#heading_2)  [**2. 现如今的 JavaScript 应⽤场景⽐当年丰富的多，代码量也⽐当年⼤很多，随便⼀个 JavaScript 项⽬的代码量，可以轻松的达到⼏万⾏，甚⾄⼗⼏万⾏！**](#heading_3)  [**3. 在代码运⾏前进⾏检查，发现代码的错误或不合理之处，减⼩运⾏时出现异常的⼏率，此种检 查叫『静态类型检查』，TypeScript 和核⼼就是『静态类型检查』，简⾔之就是把运⾏时的错误前置。**](#heading_4)  [**三、编译 TypeScript**](#heading_5)  [**浏览器不能直接运⾏ TypeScript 代码，需要编译为 JavaScript 再交由浏览器解析器执⾏。**](#heading_6)  [**四 、类型声明**](#heading_7)  [**在 : 后也可以写字⾯量类型，不过实际开发中⽤的不多。**](#heading_8)  [**五、类型推断**](#heading_9)  [**但要注意，类型推断不是万能的，⾯对复杂类型时推断容易出问题，所以尽量还是明确的编写类 型声明！**](#heading_10)  [**六、类型总览**](#heading_11)  [**JavaScript 中的数据类型**](#heading_12)  [**TypeScript 中的数据类型**](#heading_13)  [**4. 原始类型 VS 包装对象**](#heading_14)  [**5. ⾃动装箱：JavaScript 在必要时会⾃动将原始类型包装成对象，以便调⽤⽅法或访问属性**](#heading_15)  [**七、常⽤类型与语法**](#heading_16)  [**1. any**](#heading_17)  [**2. unknown**](#heading_18)  [**3. never**](#heading_19)  [**4. void**](#heading_20)  [**4.1 以下写法均符合规范**](#heading_21)  [**理解 void 与 undefined**](#heading_22)  [**总结：**](#heading_23)  [**5. object**](#heading_24)  [**object（⼩写）**](#heading_25)  [**Object（⼤写）**](#heading_26)  [**5.1 由于限制的范围实在太⼤了！所以实际开发中使⽤频率极低。**](#heading_27)  [**声明对象类型**](#heading_28)  [**声明函数类型**](#heading_29)  [**5.2 TypeScript 中的 => 在函数类型声明时表示函数类型，描述其参数类型和返回类型。**](#heading_30)  [**5.3 函数类型声明还可以使⽤：接⼝、⾃定义类型等⽅式，下⽂中会详细讲解。**](#heading_31)  [**6. tuple**](#heading_32)  [**7. enum**](#heading_33)  [**7.1 数字枚举**](#heading_34)  [**也可以指定枚举成员的初始值，其后的成员值会⾃动递增。**](#heading_35)  [**使⽤数字枚举完成刚才 walk 函数中的逻辑，此时我们发现： 代码更加直观易读，⽽且类型安全，同时也更易于维护。**](#heading_36)  [**7.2 字符串枚举**](#heading_37)  [**7.3 常量枚举**](#heading_38)  [**何为编译时内联？**](#heading_39)  [**⽽不是⽣成额外的枚举对象。这可以减少⽣成的 JavaScript 代码量，并提⾼运⾏时性能。**](#heading_40)  [**编译后⽣成的 JavaScript 代码量较⼩：**](#heading_41)  [**8. type**](#heading_42)  [**type 可以为任意类型创建别名，让代码更简洁、可读性更强，同时能更⽅便地进⾏类型复⽤和扩展。**](#heading_43)  [**联合类型是⼀种⾼级类型，它表示⼀个值可以是⼏种不同类型之⼀。**](#heading_44)  [**交叉类型（Intersection Types）允许将多个类型合并为⼀个类型。合并后的类型将拥有所有被合并类型的成员。交叉类型通常⽤于对象类型。**](#heading_45)  [**8.1 ⼀个特殊情况**](#heading_46)  [**为什么会这样？**](#heading_47)  [**官⽅⽂档的说明：AssignabilityofFunctions**](#heading_48)  [**8.2 复习类相关知识**](#heading_49)  [**本⼩节是复习类相关知识，如果有相关基础可以跳过。**](#heading_50)  [**9. 属性修饰符**](#heading_51)  [**public 修饰符**](#heading_52)  [**属性的简写形式**](#heading_53)  [**protected 修饰符**](#heading_54)  [**private 修饰符**](#heading_55)  [**readonly 修饰符**](#heading_56)  [**10. 抽象类**](#heading_57)  [**10.1 概述：抽象类是⼀种⽆法被实例化的类，专⻔⽤来定义类的结构和⾏为，类中可以写抽象**](#heading_58)  [**10.2 简记：抽象类不能实例化，其意义是可以被继承，抽象类⾥可以有普通⽅法、也可以有抽 象⽅法。**](#heading_59)  [**总结：何时使⽤抽象类？**](#heading_60)  [**10.3 提供：在抽象类中提供某些⽅法或为其提供基础实现，这样派⽣类就可以继承这 些实现。**](#heading_61)  [**11. interface（接⼝）**](#heading_62)  [**定义类结构**](#heading_63)  [**定义对象结构**](#heading_64)  [**定义函数结构**](#heading_65)  [**接⼝之间的继承**](#heading_66)  [**总结：何时使⽤接⼝？**](#heading_67)  [**11.1 定义对象的格式：**](#heading_68)  [**11.2 类的契约：规定⼀个类需要实现哪些属性和⽅法。**](#heading_69)  [**11.3 ⼀些相似概念的区别**](#heading_70)  [**⼋、泛型**](#heading_71)  [**泛型允许我们在定义函数、类或接⼝时，使⽤类型参数来表示未指定的类型，这些参数在具体 使⽤时，才被指定具体的类型，泛型能让同⼀段代码适⽤于多种类型，同时仍然保持类型的安 全性。**](#heading_72)  [**九、类型声明⽂件**](#heading_73)  [**类型声明⽂件是 TypeScript 中的⼀种特殊⽂件，通常以 。d.ts 作为扩展名。它的主要作⽤是为现有的 JavaScript 代码提供类型信息，使得 TypeScript 能够在使⽤这些 JavaScript 库或模块时进⾏类型检查和提示。**](#heading_74)  [**十、装饰器**](#heading_75) |

**⼀、TypeScript 简介**

1. TypeScript 由微软开发，是基于 JavaScript 的⼀个扩展语⾔。
2. TypeScript 包含了 JavaScript 的所有内容，即： TypeScript 是 JavaScrip t 的超集。
3. TypeScript 增加了：静态类型检查、接⼝、 泛型等很多现代开发特性，更适合⼤型项⽬的开发。
4. TypeScript 需要编译为 JavaScript ，然后交给浏览器或其他 JavaScript 运⾏环境执⾏。

|  |
| --- |
| **出身寒微不是耻辱,能屈能伸方为高级语言 !** |

**⼆、为何需要 TypeScript**

今⾮昔⽐的 JavaScript（了解）

1. **JavaScript 当年诞⽣时的定位是浏览器脚本语⾔，⽤于在⽹⻚中嵌⼊简单的逻辑，且代码量很少。**

* 随着时间的推移，JavaScript 变得越来越流⾏，如今的 JavaScript 已经可以全栈编程了。

2. **现如今的 JavaScript 应⽤场景⽐当年丰富的多，代码量也⽐当年⼤很多，随便⼀个 JavaScript 项⽬的代码量，可以轻松的达到⼏万⾏，甚⾄⼗⼏万⾏！**

* 然⽽ JavaScript 当年“出⽣简陋”，没考虑到如今的应⽤场景和代码量，逐渐就出现了很多困扰。

JavaScript 中的困扰

😞 1. 不清楚的数据类型

|  |
| --- |
| JavaScript let welcome = 'hello' welcome（） // 此⾏报错：TypeError: welcome is not a function |

😞 2. 有漏洞的逻辑

|  |
| --- |
| JavaScript const str = Date.now() % 2 ? '奇数' : '偶数'  if (str !== '奇数'）  { alert（'hello'）}  else if （str === '偶数'） { alert('world') } |

😞 3. 访问不存在的属性

|  |
| --- |
| JavaScript const obj = { width: 10, height: 15 }; const area = obj.width \* obj.heigth; |

😞 4. 低级的拼写错误

|  |
| --- |
| JavaScript const message = 'hello,world' message.toUperCase() |

『静态类型检查』

3. **在代码运⾏前进⾏检查，发现代码的错误或不合理之处，减⼩运⾏时出现异常的⼏率，此种检 查叫『静态类型检查』，TypeScript 和核⼼就是『静态类型检查』，简⾔之就是把运⾏时的错误前置。**

* 同样的功能，TypeScript 的代码量要⼤于 JavaScript，但由于 TypeScript 的代码结构更加清晰，在后期代码的维护中 TypeScript 却胜于 JavaScript。

**三、编译 TypeScript**

**浏览器不能直接运⾏ TypeScript 代码，需要编译为 JavaScript 再交由浏览器解析器执⾏。**

前置条件

⚠️ ：电脑需要安装nodejs（ 这里就不介绍了 ）

1. **命令⾏编译**

要把 .ts ⽂件编译为 .js ⽂件，需要配置 TypeScript 的编译环境，步骤如下：

* 第⼀步：创建⼀个 demo.ts ⽂件，例如：

|  |
| --- |
| JavaScript const person = {  name:'李四'，  age:18 } console.log（`我叫${person.name}，我今年${person.age}岁了`） |

* 第⼆步：全局安装 TypeScript

npm i typescript -g

* 第三步：使⽤命令编译 。ts ⽂件

tsc demo.ts

1. **⾃动化编译**

* 第⼀步：创建 TypeScript 编译控制⽂件

tsc --init

1. ⼯程中会⽣成⼀个 tsconfig.json 配置⽂件，其中包含着很多编译时的配置。
2. 观察发现，默认编译的 JS 版本是 ES7 ，我们可以⼿动调整为其他版本。

* 第⼆步：监视⽬录中的 。ts ⽂件变化

tsc --watch 或 tsc -w

* 第三步：⼩优化，当编译出错时不⽣成 。js ⽂件

tsc --noEmitOnError --watch

备注：当然也可以修改 tsconfig.json 中的 noEmitOnError 配置

**四 、类型声明**

使⽤ : 来对变量或函数形参，进⾏类型声明：

|  |
| --- |
| JavaScript let a: string//变量 a 只能存储字符串 let b: number //变量 b 只能存储数值 let c: boolean //变量 c 只能存储布尔值  a = 'hello' a = 100 //警告：不能将类型“number”分配给类型“string”  b = 666 b = '你好'//警告：不能将类型“string”分配给类型“number”  c = true c = 666 //警告：不能将类型“number”分配给类型“boolean”  // 参数 x 必须是数字，参数 y 也必须是数字，函数返回值也必须是数字 function demo(x:number,y:number):number{ return x + y }  demo(100,200) demo（100，'200'）//警告：类型“string”的参数不能赋给类型“number”的参数 demo（100,200,300） //警告：应有 2 个参数，但获得 3 个 demo（100）//警告：应有 2 个参数，但获得 1 个 |

**在 : 后也可以写字⾯量类型，不过实际开发中⽤的不多。**

|  |
| --- |
| JavaScript let a: '你好' //a 的值只能为字符串“你好” let b: 100//b 的值只能为数字 100  a = '欢迎'//警告：不能将类型“"欢迎"”分配给类型“"你好"” b = 200//警告：不能将类型“200”分配给类型“100” |

**五、类型推断**

TS 会根据我们的代码，进⾏类型推导，例如下⾯代码中的变量 d ，只能存储数字

|  |
| --- |
| JavaScript let d = -99 //TypeScript 会推断出变量 d 的类型是数字 d = false//警告：不能将类型“boolean”分配给类型“number” |

**但要注意，类型推断不是万能的，⾯对复杂类型时推断容易出问题，所以尽量还是明确的编写类 型声明！**

**六、类型总览**

**JavaScript 中的数据类型**

**① string**

**② number**

**③ boolean**

**④ null**

**⑤ undefined**

**⑥ bigint**

**⑦ symbol**

**⑧ object**

备注：其中 **object** 包含： **Array** 、 **Function** 、 **Date** 、 **Error** 等......

**TypeScript 中的数据类型**

1. 上述所有 JavaScript 类型
2. 六个新类型：

① **any**

② **unknown**

③ **never**

④ **void**

⑤ **tuple**

⑥ **enum**

1. 两个⽤于⾃定义类型的⽅式：

**① type**

**② interface**

注意点！

在 JavaScript 中的这些内置构造函数： Number 、 String 、 Boolean ，⽤于创建对应的包装对象， 在⽇常开发时很少使⽤，在 TypeScript 中也是同理，所以在 TypeScript 中进⾏类型声明时，通常都是⽤⼩写的 number 、 string 、 boolean

例如下⾯代码：

|  |
| --- |
| JavaScript let str1: string str1 = 'hello' str1 = new String（'hello'） //报错  let str2: String str2 = 'hello' str2 = new String('hello')  console.log(typeof str1) console.log(typeof str2) |

4. **原始类型 VS 包装对象**

* 原始类型：如 number 、 string 、 boolean ，在 JavaScript 中是简单数据类型，它们在内存中占⽤空间少，处理速度快。
* 包装对象：如 Number 对象、 String 对象、 Boolean 对象，是复杂类型，在内存中占⽤更多空间，在⽇常开发时很少由开发⼈员⾃⼰创建包装对象。

5. **⾃动装箱：JavaScript 在必要时会⾃动将原始类型包装成对象，以便调⽤⽅法或访问属性**

|  |
| --- |
| JavaScript // 原始类型字符串 let str = 'hello';  // 当访问 str.length 时，JavaScript 引擎做了以下⼯作： let size = (function() { // 1. ⾃动装箱：创建⼀个临时的 String 对象包装原始字符串 let tempStringObject = new String(str);  // 2. 访问 String 对象的 length 属性 let lengthValue = tempStringObject.length;  // 3. 销毁临时对象，返回⻓度值 // （JavaScript 引擎⾃动处理对象销毁，开发者⽆感知） return lengthValue; })();  console.log(size); // 输出： 5 |

**七、常⽤类型与语法**

1. **any**

any 的含义是：任意类型，⼀旦将变量类型限制为 any ，那就意味着放弃了对该变量的类型检查。

|  |
| --- |
| JavaScript // 明确的表示 a 的类型是 any —— 【显式的 any】 let a: any // 以下对 a 的赋值，均⽆警告 a = 100 a = '你好' a = false  // 没有明确的表示 b 的类型是 any，但 TS 主动推断出来 b 是 any —— 隐式的 any let b //以下对 b 的赋值，均⽆警告 b = 100 b = '你好' b = false  注意点： any 类型的变量，可以赋值给任意类型的变量  /\* 注意点：any 类型的变量，可以赋值给任意类型的变量 \*/ let c:any c = 9  let x: string x = c // ⽆警告 |

2. **unknown**

unknown 的含义是：未知类型，适⽤于：起初不确定数据的具体类型，要后期才能确定

1. unknown 可以理解为⼀个类型安全的 any 。

|  |
| --- |
| JavaScript // 设置 a 的类型为 unknown let a: unknown  //以下对 a 的赋值，均符合规范 a = 100 a = false a = '你好'  // 设置 x 的数据类型为 string let x: string x = a //警告：不能将类型“unknown”分配给类型“string” |

1. unknown 会强制开发者在使⽤之前进⾏类型检查，从⽽提供更强的类型安全性。

|  |
| --- |
| JavaScript // 设置 a 的类型为 unknown let a: unknown a = 'hello'  //第⼀种⽅式：加类型判断 if（typeof a === 'string'）{ x = a console.log(x) }  //第⼆种⽅式：加断⾔x = a as string  //第三种⽅式：加断⾔x = <string>a |

1. 读取 any 类型数据的任何属性都不会报错，⽽ unknown 正好与之相反。

|  |
| --- |
| JavaScript let str1: string str1 = 'hello' str1.toUpperCase（） //⽆警告  let str2: any str2 = 'hello' str2.toUpperCase（） //⽆警告  let str3: unknown str3 = 'hello'; str3.toUpperCase（） //警告：“str3”的类型为“未知”  // 使⽤断⾔强制指定 str3 的类型为 string （str3 as string）。toUpperCase（） //⽆警告 |

3. **never**

never 的含义是：任何值都不是，即：不能有值，例如 undefined 、 nul l 、 '' 、 0 都不⾏！

1. ⼏乎不⽤ never 去直接限制变量，因为没有意义，例如：

|  |
| --- |
| JavaScript  /\* 指定 a 的类型为 never，那就意味着 a 以后不能存任何的数据了 \*/ let a: never  // 以下对 a 的所有赋值都会有警告 a = 1 a = true a = undefined a = null |

1. never ⼀般是 TypeScript 主动推断出来的，例如：

|  |
| --- |
| JavaScript // 指定 a 的类型为 string let a: string // 给 a 设置⼀个值 a = 'hello'  if (typeof a === 'string') { console.log(a.toUpperCase()) } else { console.log（a） // TypeScript 会推断出此处的 a 是 never，因为没有任何⼀个值符合此处的 逻辑 |

1. never 也可⽤于限制函数的返回值

|  |
| --- |
| JavaScript // 限制 throwError 函数不需要有任何返回值，任何值都不⾏，像 undeifned、null 都不⾏ function throwError（str: string）: never { throw new Error（'程序异常退出：' + str） } |

4. **void**

void 的含义是空，即：函数不返回任何值，调⽤者也不应依赖其返回值进⾏任何操作！

1. void 通常⽤于函数返回值声明

|  |
| --- |
| JavaScript function logMessage(msg:string):void{ console.log(msg) } logMessage（'你好'） |

注意：编码者没有编写 return 指定函数返回值，所以 logMessage 函数是没有显式返回值的，但会有⼀个隐式返回值 ，是 undefined ，虽然函数返回类型为 void ，但也是可以接受 undefined 的，简单记： **undefined** 是 **void** 可以接受的⼀种“空”。

4.1 **以下写法均符合规范**

|  |
| --- |
| JavaScript  // ⽆警告 function logMessage(msg:string):void{ console.log(msg) }  // ⽆警告 function logMessage(msg:string):void{ console.log(msg) return; }  // ⽆警告 function logMessage(msg:string):void{ console.log(msg) return undefined } |

1. 那限制函数返回值时，是不是 undefined 和 void 就没区别呢？—— 有区别。因为还有这句话 ：【返回值类型为 **void** 的函数，调⽤者不应依赖其返回值进⾏任何操作！】对⽐下⾯两段代码：

|  |
| --- |
| JavaScript function logMessage(msg:string):void{ console.log(msg) }  let result = logMessage（'你好'）  if（result）{ // 此⾏报错：⽆法测试 "void" 类型的表达式的真实性 console.log（'logMessage 有返回值'） }   function logMessage(msg:string):undefined{ console.log(msg) } let result = logMessage（'你好'） if（result）{ // 此⾏⽆警告 console.log（'logMessage 有返回值'） } |

**理解 void 与 undefined**

* void 是⼀个⼴泛的概念，⽤来表达“空”，⽽ undefined 则是这种“空”的具体实现。
* 因此可以说 undefined 是 void 能接受的⼀种“空”的状态。
* 也可以理解为： void 包含 undefined ，但 void 所表达的语义超越了 undefi ned ， void 是⼀种意图上的约定，⽽不仅仅是特定值的限制。

**总结：**

如果⼀个函数返回类型为 void ，那么：

1. 从语法上讲：函数是可以返回 undefined 的，⾄于显式返回，还是隐式返回，这⽆所谓！
2. 从语义上讲：函数调⽤者不应关⼼函数返回的值，也不应依赖返回值进⾏任何操作！ 即使我们知道它返回了 undefined 。

5. **object**

关于 **object** 与 **Object** ，直接说结论：实际开发中⽤的相对较少，因为范围太⼤了。

**object（⼩写）**

**object** （⼩写）的含义是：所有⾮原始类型，可存储：对象、函数、数组等，由于限制的范围⽐较宽泛，在实际开发中使⽤的相对较少。

|  |
| --- |
| JavaScript let a:object //a 的值可以是任何【⾮原始类型】，包括：对象、函数、数组等  // 以下代码，是将【⾮原始类型】赋给 a，所以均符合要求 a = {} a = {name:'张三'} a = [1,3,5,7,9] a = function(){} a = new String('123') class Person {} a = new Person()  // 以下代码，是将【原始类型】赋给 a，有警告 a = 1// 警告：不能将类型“number”分配给类型“object” a = true// 警告：不能将类型“boolean”分配给类型“object” a = '你好'// 警告：不能将类型“string”分配给类型“object” a = null// 警告：不能将类型“null”分配给类型“object” a = undefined // 警告：不能将类型“undefined”分配给类型“object” |

**Object（⼤写）**

* 官⽅描述：所有可以调⽤ **Object** ⽅法的类型。
* 简单记忆：除了 undefined 和 null 的任何值。

5.1 **由于限制的范围实在太⼤了！所以实际开发中使⽤频率极低。**

|  |
| --- |
| JavaScript  let b:Object //b 的值必须是 Object 的实例对象（除去 undefined 和 null 的任何值）  // 以下代码，均⽆警告，因为给 a 赋的值，都是 Object 的实例对象 b = {} b = {name:'张三'} b = [1,3,5,7,9] b = function(){} b = new String('123') class Person {} b = new Person() b = 1// 1 不是 Object 的实例对象，但其包装对象是 Object 的实例 b = true// truue 不是 Object 的实例对象，但其包装对象是 Object 的实例 b = '你好' // “你好”不是 Object 的实例对象，但其包装对象是 Object 的实例  // 以下代码均有警告 b = null// 警告：不能将类型“null”分配给类型“Object” b = undefined // 警告：不能将类型“undefined”分配给类型“Object” |

**声明对象类型**

1. 实际开发中，限制⼀般对象，通常使⽤以下形式

|  |
| --- |
| JavaScript // 限制 person1 对象必须有 name 属性，age 为可选属性 let person1: { name: string, age?: number }  // 含义同上，也能⽤分号做分隔 let person2: { name: string; age?: number }  // 含义同上，也能⽤换⾏做分隔 let person3: { name: string age?: number }  // 如下赋值均可以 person1 = {name:'李四'，age:18} person2 = {name:'张三'} person3 = {name:'王五'}  // 如下赋值不合法，因为 person3 的类型限制中，没有对 gender 属性的说明 person3 = {name:'王五'，gender:'男'}  索引签名： 允许定义对象可以具有任意数量的属性，这些属性的键和类型是可变的， 常⽤于：描述类型不确定的属性，（具有动态属性的对象）。  // 限制 person 对象必须有 name 属性，可选 age 属性但值必须是数字，同时可以有任意数量、任意类型的其他属性 let person: { name: string age?: number [key: string]: any // 索引签名，完全可以不⽤key 这个单词，换成其他的也可以 }  // 赋值合法 person = { name:'张三'， age:18， gender:'男' } |

**声明函数类型**

|  |
| --- |
| JavaScript let count: (a: number, b: number) => number  count = function (x, y) { return x + y } |

备注：

5.2 **TypeScript 中的 => 在函数类型声明时表示函数类型，描述其参数类型和返回类型。**

* JavaScript 中的 => 是⼀种定义函数的语法，是具体的函数实现。

5.3 **函数类型声明还可以使⽤：接⼝、⾃定义类型等⽅式，下⽂中会详细讲解。**

声明数组类型

|  |
| --- |
| JavaScript let arr1: string[] let arr2: Array<string>  arr1 = ['a','b','c'] arr2 = ['hello','world'] |

备注：上述代码中的 Array<string> 属于泛型，下⽂会详细讲解。

6. **tuple**

元组 （Tuple） 是⼀种特殊的数组类型，可以存储固定数量的元素，并且每个元素的类型是已知的且可以不同。元组⽤于精确描述⼀组值的类型， ？ 表示可选元素。

|  |
| --- |
| JavaScript // 第⼀个元素必须是 string 类型，第⼆个元素必须是 number 类型。 let arr1: [string,number] // 第⼀个元素必须是 number 类型，第⼆个元素是可选的，如果存在，必须是 boolean 类型。 let arr2: [number,boolean?] // 第⼀个元素必须是 number 类型，后⾯的元素可以是任意数量的 string 类型 let arr3: [number,...string[]]  // 可以赋值 arr1 = ['hello',123] arr2 = [100,false] arr2 = [200] arr3 = [100,'hello','world'] arr3 = [100] // 不可以赋值，arr1 声明时是两个元素，赋值的是三个 arr1 = ['hello',123,false] |

7. **enum**

枚举（ enum ）可以定义⼀组命名常量，它能增强代码的可读性，也让代码更好维护。

如下代码的功能是：

根据调⽤ walk 时传⼊的不同参数，执⾏不同的逻辑，存在的问题是调⽤ w alk 时传参时没有任何提示，编码者很容易写错字符串内容；并且⽤于判断逻辑的 up 、 dow n 、 left 、 right 是连续且相关的⼀组值，那此时就特别适合使⽤ 枚举（ **enum** ）。

|  |
| --- |
| JavaScript function walk(str:string) { if (str === 'up') { console.log（"向【上】⾛"）； } else if （str === 'down'） { console.log（"向【下】⾛"）； } else if （str === 'left'） { console.log（"向【左】⾛"）； } else if （str === 'right'） { console.log（"向【右】⾛"）； } else { console.log（"未知⽅向"）； } }  walk('up') walk('down') walk('left') walk('right') |

7.1 **数字枚举**

数字枚举⼀种最常⻅的枚举类型，其成员的值会⾃动递增，且数字枚举还具备反向映射的 特点，在下⾯代码的打印中，不难发现：可以通过值来获取对应的枚举成员名称 。

|  |
| --- |
| JavaScript // 定义⼀个描述【上下左右】⽅向的枚举 Direction enum Direction { Up, Down, Left, Right }  console.log（Direction） // 打印 Direction 会看到如下内容 /\* { 0:'Up', 1:'Down', 2:'Left', 3:'Right', Up:0, Down:1, Left:2, Right:3 } \*/  // 反向映射 console.log（Direction.Up） console.log（Direction[0]）  // 此⾏代码报错，枚举中的属性是只读的 Direction.Up = 'shang' |

**也可以指定枚举成员的初始值，其后的成员值会⾃动递增。**

|  |
| --- |
| JavaScript enum Direction { Up = 6, Down, Left, Right }  console.log(Direction.Up); // 输出： 6 console.log（Direction.Down）； // 输出： 7 |

**使⽤数字枚举完成刚才 walk 函数中的逻辑，此时我们发现： 代码更加直观易读，⽽且类型安全，同时也更易于维护。**

|  |
| --- |
| JavaScript enum Direction { Up, Down, Left, Right, }  function walk(n: Direction) { if (n === Direction.Up) { console.log（"向【上】⾛"）； } else if （n === Direction.Down） { console.log（"向【下】⾛"）； } else if （n === Direction.Left） { console.log（"向【左】⾛"）； } else if （n === Direction.Right） { console.log（"向【右】⾛"）； } else { console.log（"未知⽅向"）； } }  walk(Direction.Up) walk(Direction.Down) |

7.2 **字符串枚举**

枚举成员的值是字符串

|  |
| --- |
| JavaScript enum Direction  {   Up = "up",  Down = "down",   Left = "left",  Right = "right" }  let dir: Direction = Direction.Up; console.log(dir); // 输出： "up" |

7.3 **常量枚举**

官⽅描述：常量枚举是⼀种特殊枚举类型，它使⽤ const 关键字定义，在编译时会被内联，避免⽣成⼀些额外的代码。

**何为编译时内联？**

所谓“内联”其实就是 TypeScript 在编译时，会将枚举成员引⽤替换为它们的实际值，

**⽽不是⽣成额外的枚举对象。这可以减少⽣成的 JavaScript 代码量，并提⾼运⾏时性能。**

使⽤普通枚举的 TypeScript 代码如下：

|  |
| --- |
| JavaScript enum Directions {  Up, Down, Left, Right }  let x = Directions.Up; |

编译后⽣成的 JavaScript 代码量较⼤ ：

|  |
| --- |
| JavaScript "use strict"; var Directions; (function (Directions) { Directions[Directions["Up"] = 0] = "Up"; Directions[Directions["Down"] = 1] = "Down"; Directions[Directions["Left"] = 2] = "Left"; Directions[Directions["Right"] = 3] = "Right"; })(Directions || (Directions = {})); let x = Directions.Up; |

使⽤常量枚举的 TypeScript 代码如下：

|  |
| --- |
| JavaScript const enum Directions { Up, Down, Left, Right }  let x = Directions.Up; |

**编译后⽣成的 JavaScript 代码量较⼩：**

|  |
| --- |
| JavaScript "use strict"; let x = 0 /\* Directions.Up \*/; |

8. **type**

**type 可以为任意类型创建别名，让代码更简洁、可读性更强，同时能更⽅便地进⾏类型复⽤和扩展。**

1. 基本⽤法

类型别名使⽤ type 关键字定义， type 后跟类型名称，例如下⾯代码中 num 是类型别名。

|  |
| --- |
| JavaScript type num = number;  let price: num price = 100 |

1. 联合类型

**联合类型是⼀种⾼级类型，它表示⼀个值可以是⼏种不同类型之⼀。**

|  |
| --- |
| JavaScript type Status = number | string type Gender = '男' | '⼥'  function printStatus(status: Status) { console.log(status); }  function logGender(str:Gender){ console.log(str) }  printStatus(404); printStatus('200'); printStatus('501');  logGender（'男'） logGender（'⼥'） |

1. 交叉类型

**交叉类型（Intersection Types）允许将多个类型合并为⼀个类型。合并后的类型将拥有所有被合并类型的成员。交叉类型通常⽤于对象类型。**

|  |
| --- |
| JavaScript //⾯积 type Area = { height: number; //⾼width: number; //宽 };  //地址 type Address = { num: number; //楼号 cell: number; //单元号 room: string; //房间号 };  // 定义类型 House，且 House 是 Area 和 Address 组成的交叉类型 type House = Area & Address;  const house: House = { height: 180, width: 75, num: 6, cell: 3, room: '702' }; |

8.1 **⼀个特殊情况**

先来观察如下两段代码：

代码段 1（正常）

在函数定义时，限制函数返回值为 void ，那么函数的返回值就必须是空。

|  |
| --- |
| JavaScript function demo():void{ // 返回 undefined 合法 return undefined  // 以下返回均不合法 return 100 return false return null return [] } demo() |

代码段 2（特殊）

使⽤限制函数返回值为 void 时， **TypeScript** 并不会严格要求函数返回空。

|  |
| --- |
| JavaScript type LogFunc = () => void  const f1: LogFunc = () => { return 100; // 允许返回⾮空值 };  const f2: LogFunc = () => 200; // 允许返回⾮空值  const f3: LogFunc = function () { return 300; // 允许返回⾮空值 }; |

**为什么会这样？**

是为了确保如下代码成⽴，我们知道 Array.prototype.push 的返回值是⼀个数字，

⽽ Array.prototype.forEach ⽅法期望其回调的返回类型是 void 。

|  |
| --- |
| JavaScript  const src = [1, 2, 3]; const dst = [0];  src.forEach((el) => dst.push(el)); |

**官⽅⽂档的说明：**[**AssignabilityofFunctions**](https://www.typescriptlang.org/docs/handbook/2/functions.html#assignability-of-functions)

8.2 **复习类相关知识**

**本⼩节是复习类相关知识，如果有相关基础可以跳过。**

类 classTypeScript

|  |
| --- |
| JavaScript   class Person { // 属性声明 name: string age: number // 构造器 constructor(name: string, age: number) { this.name = name this.age = age } // ⽅法 speak() { console.log（`我叫：${this.name}，今年${this.age}岁`） } }  // Person 实例 const p1 = new Person（'周杰伦'， 38）   Student 继承 PersonTypeScript  class Student extends Person { grade: string // 构造器 constructor(name: string, age: number, grade: string) { super(name, age) this.grade = grade } // 备注本例中若 Student 类不需要额外的属性，Student 的构造器可以省略 // 重写从⽗类继承的⽅法 override speak() { console.log（`我是学⽣，我叫：${this.name}，今年${this.age}岁，在读${this.grade} 年级`，） } // ⼦类⾃⼰的⽅法 study() { console.log（`${this.name}正在努⼒学习中`） } } |

9. **属性修饰符**

|  |  |  |
| --- | --- | --- |
| 修饰符 | 含义 | 具体规则 |
| public | 公开的 | 可以被：类内部、⼦类、类外部访问 。 |
| protected | 受保护的 | 可以被：类内部、⼦类访问。 |
| private | 私有的 | 可以被：类内部访问。 |
| readonly | 只读属性 | 属性⽆法修改。 |

**public 修饰符**

Person 类 TypeScript

|  |
| --- |
| JavaScript   class Person { // name 写了 public 修饰符，age 没写修饰符，最终都是 public 修饰符 public name: string age: number constructor(name: string, age: number) { this.name = name this.age = age } speak() { // 类的【内部】可以访问 public 修饰的 name 和 age console.log（`我叫：${this.name}，今年${this.age}岁`） } }  const p1 = new Person（'张三'， 18） // 类的【外部】可以访问 public 修饰的属性 console.log(p1.name)   Student 继承 PersonTypeScript  class Student extends Person { constructor(name: string, age: number) { super(name, age) } study() { // 【⼦类中】可以访问⽗类中 public 修饰的：name 属性、age 属性 console.log（`${this.age}岁的${this.name}正在努⼒学习`） } } |

**属性的简写形式**

完整写法 TypeScript

|  |
| --- |
| JavaScript class Person { public name: string; public age: number;  constructor(name: string, age: number) { this.name = name; this.age = age; } } |

简写形式 TypeScript

|  |
| --- |
| JavaScript class Person { constructor( public name: string, public age: number ) { } } |

**protected 修饰符**

Person 类 TypeScript

|  |
| --- |
| JavaScript class Person { // name 和 age 是受保护属性，不能在类外部访问，但可以在【类】与【⼦类】中访问 constructor( protected name: string, protected age: number ) {} // getDetails 是受保护⽅法，不能在类外部访问，但可以在【类】与【⼦类】中访问 protected getDetails（）: string { // 类中能访问受保护的 name 和 age 属性 return `我叫：${this.name}，年龄是：${this.age}` } // introduce 是公开⽅法，类、⼦类、类外部都能使⽤ introduce() { // 类中能访问受保护的 getDetails⽅法 console.log(this.getDetails()); } }  const p1 = new Person（'杨超越'，18） // 可以在类外部访问 introduce p1.introduce()  // 以下代码均报错 // p1.getDetails() // p1.name // p1.age |

Student 继承 PersonTypeScript

|  |
| --- |
| JavaScript class Student extends Person { constructor(name:string,age:number){ super(name,age) } study(){ // ⼦类中可以访问 introduce this.introduce（） // ⼦类中可以访问 name console.log（`${this.name}正在努⼒学习`） } }  const s1 = new Student('tom',17) s1.introduce() |

**private 修饰符**

|  |
| --- |
| JavaScript class Person { constructor( public name: string, public age: number, // IDCard 属性为私有的（private）属性，只能在【类内部】使⽤ private IDCard: string ) { } private getPrivateInfo(){ // 类内部可以访问私有的（private）属性 —— IDCard return `身份证号码为：${this.IDCard}` } getInfo() { // 类内部可以访问受保护的（protected）属性 —— name 和 age return `我叫： ${this.name}， 今年刚满${this.age}岁`； } getFullInfo(){ // 类内部可以访问公开的 getInfo⽅法，也可以访问私有的 getPrivateInfo⽅法 return this.getInfo() + ', ' + this.getPrivateInfo() } }  const p1 = new Person（'张三'，18，'110114198702034432'） console.log（p1.getFullInfo（）） console.log（p1.getInfo（））  // 以下代码均报错 // p1.name // p1.age // p1.IDCard // p1.getPrivateInfo() |

**readonly 修饰符**

|  |
| --- |
| JavaScript class Car { constructor( public readonly vin: string， //⻋辆识别码，为只读属性 public readonly year: number，//出⼚年份，为只读属性 public color: string, public sound: string ) { }  // 打印⻋辆信息 displayInfo() { console.log(` 识别码：${this.vin}， 出⼚年份：${this.year}， 颜⾊：${this.color}， ⾳响：${this.sound} `); } }  const car = new Car（'1HGCM82633A123456'， 2018， '⿊⾊'， 'Bose⾳响'）； car.displayInfo()  // 以下代码均错误：不能修改 readonly 属性 // car.vin = '897WYE87HA8SGDD8SDGHF'; // car.year = 2020; |

10. **抽象类**

10.1 **概述：抽象类是⼀种⽆法被实例化的类，专⻔⽤来定义类的结构和⾏为，类中可以写抽象**

⽅法，也可以写具体实现。抽象类主要⽤来为其派⽣类提供⼀个基础结构，要求其派⽣类 必须实现其中的抽象⽅法。

10.2 **简记：抽象类不能实例化，其意义是可以被继承，抽象类⾥可以有普通⽅法、也可以有抽 象⽅法。**

通过以下场景，理解抽象类：

我们定义⼀个抽象类 Package ，表示所有包裹的基本结构，任何包裹都有重量属性 weigh

t ，包裹都需要计算运费。但不同类型的包裹（如：标准速度、特快专递）都有不同的运费计算

⽅式，因此⽤于计算运费的 calculate ⽅法是⼀个抽象⽅法，必须由具体的⼦类来实现。

Package 类 TypeScript

|  |
| --- |
| JavaScript abstract class Package { constructor(public weight: number) { } // 抽象⽅法：⽤来计算运费，不同类型包裹有不同的计算⽅式 abstract calculate(): number // 通⽤⽅法：打印包裹详情 printPackage() { console.log（`包裹重量为： ${this.weight}kg，运费为： ${this.calculate（）}元`）； } } |

StandardPackage 类继承了 Package ，实现了 calculate ⽅法：

StandardPackage 类 （标快包裹）TypeScript

// 标准包裹

class StandardPackage extends Package { constructor(

weight: number,

public unitPrice: number // 每公⽄的固定费率

) { super(weight) }

// 实现抽象⽅法：计算运费

calculate(): number {

return this.weight \* this.unitPrice;

}

}

// 创建标准包裹实例

const s1 = new StandardPackage(10,5) s1.printPackage()

ExpressPackage 类继承了 Package ，实现了 calculate ⽅法：

|  |
| --- |
| JavaScript ExpressPackage 类（特快包裹）TypeScript  class ExpressPackage extends Package { constructor( weight: number, private unitPrice: number， // 每公⽄的固定费率（快速包裹更⾼） private additional: number // 超出 10kg 以后的附加费 ) { super(weight) }  // 实现抽象⽅法：计算运费 calculate(): number { if(this.weight > 10){ // 超出 10kg 的部分，每公⽄多收 additional 对应的价格 return 10 \* this.unitPrice + (this.weight - 10) \* this.additional }else { return this.weight \* this.unitPrice; } } }  // 创建特快包裹实例 const e1 = new ExpressPackage(13,8,2) e1.printPackage() |

**总结：何时使⽤抽象类？**

1. 定义：为⼀组相关的类定义通⽤的⾏为（⽅法或属性）时。

10.3 **提供：在抽象类中提供某些⽅法或为其提供基础实现，这样派⽣类就可以继承这 些实现。**

1. 确保：强制派⽣类实现⼀些关键⾏为。
2. 代码和逻辑：当多个类需要共享部分代码时，抽象类可以避免代码重复。

11. **interface（接⼝）**

interface 是⼀种定义结构的⽅式，主要作⽤是为：类、对象、函数等规定⼀种契约，这样可以确保代码的⼀致性和类型安全，但要注意 interface 只能定义格式，不能包含任何实现 ！

**定义类结构**

|  |
| --- |
| JavaScript // PersonInterface 接⼝，⽤与限制 Person 类的格式 interface PersonInterface { name: string age: number speak(n: number): void }  // 定义⼀个类 Person，实现 PersonInterface 接⼝ class Person implements PersonInterface { constructor( public name: string, public age: number ) { } // 实现接⼝中的 speak ⽅法 speak(n: number): void { for (let i = 0; i < n; i++) { // 打印出包含名字和年龄的问候语句 console.log（`你好，我叫${this.name}，我的年龄是${this.age}`）； } } }  // 创建⼀个 Person 类的实例 p1，传⼊名字 'tom' 和年龄 18 const p1 = new Person（'tom'， 18）； p1.speak(3) |

**定义对象结构**

|  |
| --- |
| JavaScript interface UserInterface { name: string readonly gender: string // 只读属性 age?： number // 可选属性 run: (n: number) => void }  const user: UserInterface = { name: "张三"， gender: '男'， age: 18, run(n) { console.log（`奔跑了${n}⽶`） } }; |

**定义函数结构**

|  |
| --- |
| JavaScript interface CountInterface { (a: number, b: number): number; }  const count: CountInterface = (x, y) => { return x + y } |

**接⼝之间的继承**

⼀个 interface 继承另⼀个 interface ，从⽽实现代码的复⽤

|  |
| --- |
| JavaScript interface PersonInterface { name: string // 姓名 age: number // 年龄 }  interface StudentInterface extends PersonInterface { grade: string // 年级 }  const stu: StudentInterface = { name: "张三"， age: 25， grade: '⾼三'， }  接⼝⾃动合并（可重复定义）  // PersonInterface 接⼝interface PersonInterface { // 属性声明 name: string age: number }  // 给 PersonInterface 接⼝添加新属性 interface PersonInterface { // ⽅法声明 speak(): void }  // Person 类实现 PersonInterface class Person implements PersonInterface { name: string age: number // 构造器 constructor(name: string, age: number) { this.name = name this.age = age } // ⽅法 speak() { console.log（'你好！我是⽼师：'， this.name） } } |

**总结：何时使⽤接⼝？**

11.1 **定义对象的格式：**

描述数据模型、API 响应格式、配置对象等等，是开发中⽤的最多的场景。

11.2 **类的契约：规定⼀个类需要实现哪些属性和⽅法。**

扩展已有接⼝：⼀般⽤于扩展第三⽅库的类型， 这种特性在⼤型项⽬中可能会⽤到。

11.3 **⼀些相似概念的区别**

1. interface 与 type 的区别

* 相同点： interface 和 type 都可以⽤于定义对象结构，在定义对象结构时两者可以互换。
* 不同点：

interface ：更专注于定义对象和类的结构，⽀持继承、合并。

type ：可以定义类型别名、联合类型、交叉类型，但不⽀持继承和⾃动合并。

interface 和 type 都可以定义对象结构 TypeScript

|  |
| --- |
| JavaScript  // 使⽤ interface 定义 Person 对象 interface PersonInterface { name: string; age: number; speak(): void; }  // 使⽤ type 定义 Person 对象 type PersonType = { name: string; age: number; speak(): void; };  // 使⽤PersonInterface /\* let person: PersonInterface = { name:'张三'， age:18, speak(){ console.log（`我叫：${this.name}，年龄：${this.age}`） } } \*/  // 使⽤PersonType let person: PersonType = { name:'张三'， age:18, speak(){ console.log（`我叫：${this.name}，年龄：${this.age}`） } }  interface 可以继承、合并 TypeScript  interface PersonInterface { name: string // 姓名 age: number // 年龄 }  interface PersonInterface { speak: () => void }  interface StudentInterface extends PersonInterface { grade: string // 年级 }  const student: StudentInterface = { name: '李四'， age: 18, grade: '⾼⼆', speak() { console.log(this.name,this.age,this.grade) } } |

type 的交叉类型 TypeScript

|  |
| --- |
| JavaScript // 使⽤ type 定义 Person 类型，并通过交叉类型实现属性的合并 type PersonType = { name: string; // 姓名 age: number； // 年龄 } & { speak: () => void; };  // 使⽤ type 定义 Student 类型，并通过交叉类型继承 PersonType type StudentType = PersonType & { grade: string; // 年级 };  const student: StudentType = { name: '李四'， age: 18, grade: '⾼⼆', speak() { console.log(this.name, this.age, this.grade); } }; |

1. interface 与 抽象类的区别

* 相同点：都能定义⼀个类的格式（定义类应遵循的契约）
* 不相同：

接⼝：只能描述结构，不能有任何实现代码，⼀个类可以实现多个接⼝。

抽象类：既可以包含抽象⽅法，也可以包含具体⽅法， ⼀个类只能继承⼀个抽象类。

⼀个类可以实现多个接⼝TypeScript

|  |
| --- |
| JavaScript // FlyInterface 接⼝ interface FlyInterface { fly(): void; }  // 定义 SwimInterface 接⼝ interface SwimInterface { swim(): void; }  // Duck 类实现了 FlyInterface 和 SwimInterface 两个接⼝ class Duck implements FlyInterface, SwimInterface { fly(): void { console.log（'鸭⼦可以⻜'）； }  swim（）: void { console.log（'鸭⼦可以游泳'）； } }  // 创建⼀个 Duck 实例 const duck = new Duck(); duck.fly(); // 输出： 鸭⼦可以⻜duck.swim（）； // 输出： 鸭⼦可以游泳 |

**⼋、泛型**

**泛型允许我们在定义函数、类或接⼝时，使⽤类型参数来表示未指定的类型，这些参数在具体 使⽤时，才被指定具体的类型，泛型能让同⼀段代码适⽤于多种类型，同时仍然保持类型的安 全性。**

举例：如下代码中 <T> 就是泛型，（不⼀定⾮叫 T ），设置泛型后即可在函数中使⽤ T 来表示该类型：

泛型函数

|  |
| --- |
| TypeScript function logData<T>(data: T): T { console.log(data) return data }  logData<number>(100) logData<string>('hello') |

泛型可以有多个

|  |
| --- |
| TypeScript  function logData<T, U>(data1: T, data2: U): T | U { console.log(data1,data2) return Date.now() % 2 ? data1 : data2 }  logData<number, string>(100, 'hello') logData<string, boolean>('ok', false) |

泛型接⼝

|  |
| --- |
| TypeScript interface PersonInterface<T> { name: string, age: number, extraInfo: T }  let p1: PersonInterface<string> let p2: PersonInterface<number>  p1 = { name: '张三'， age: 18， extraInfo: '⼀个好⼈' } p2 = { name: '李四'， age: 18， extraInfo: 250 } |

泛型约束

|  |
| --- |
| TypeScript interface LengthInterface { length: number }  // 约束规则是：传⼊的类型 T 必须具有 length 属性 function logPerson<T extends LengthInterface>(data: T): void { console.log(data.length) }  logPerson<string>('hello') // 报错：因为 number 不具备 length 属性 // logPerson<number>(100) |

泛型类

|  |
| --- |
| TypeScript class Person<T> { constructor( public name: string, public age: number, public extraInfo: T ) { } speak() { console.log（`我叫${this.name}今年${this.age}岁了`） console.log（this.extraInfo） } }  // 测试代码 1 const p1 = new Person<number>("tom", 30, 250);  // 测试代码 2 type JobInfo = { title: string; company: string; } const p2 = new Person<JobInfo>（"tom"， 30， { title: '研发总监'， company: '发发发科技公司' }）； |

**九、类型声明⽂件**

**类型声明⽂件是 TypeScript 中的⼀种特殊⽂件，通常以 。d.ts 作为扩展名。它的主要作⽤是为现有的 JavaScript 代码提供类型信息，使得 TypeScript 能够在使⽤这些 JavaScript 库或模块时进⾏类型检查和提示。**

demo.js

|  |
| --- |
| TypeScript export function add(a, b) { return a + b; }  export function mul(a, b) { return a \* b; }    demo.d.tsTypeScript  declare function add(a: number, b: number): number; declare function mul(a: number, b: number): number;  export { add, mul }; |

index.ts

|  |
| --- |
| TypeScript // example.ts import { add, mul } from "./demo.js";  const x = add(2, 3); // x 类型为 number const y = mul（4， 5）； // y 类型为 number  console.log(x,y) |

**十、装饰器**

[TypeScript 装饰器](https://ling-yun.feishu.cn/docx/V7YSdIGaOoFFzsxNmTjcovuKnTg)