**TypeScript快速入门**

1.TypeScript介绍

1.1.TypeScript为什么要为JS添加类型支持

1.2.TypeScript相比JS优势

2.TypeScript初体验

2.1.安装编译TS的工具包

2.2.编译并运行TS代码

2.3.简化运行TS代码

3.TypeScript常用类型

3.1.类型注解

3.2.常用基础类型

3.3.原始类型 number/string/boolean/null/undefined/symbol

3.4.对象类型（数组）

3.5.类型别名（自定义类型）

3.6.函数类型

void类型

可选参数

3.7.对象类型

可选属性

3.8.接口

接口和类型别名对比

接口继承

3.9.元组

3.10.类型推论

3.11.类型断言

3.12.字面量类型

3.13.枚举类型

3.14.any类型

3.15.typeof

4.TypeScript高级类型

4.1. class类

class基本使用：

构造函数

继承extends，implement

可见性修饰符public，protected，private

只读修饰符readonly

4.2.类型兼容性

类之间兼容性

接口之间兼容性

函数直接兼容性

4.3.交叉类型（&）

4.4.泛型 和 keyof

泛型

泛型约束

泛型接口

泛型类

泛型工具类型Partial，Readonly，Pick，Record

4.5.索引签名类型 和 索引查询类型

索引签名类型

映射类型

索引查询类型

5.TypeScript类型声明文件

TS 的两种文件类型

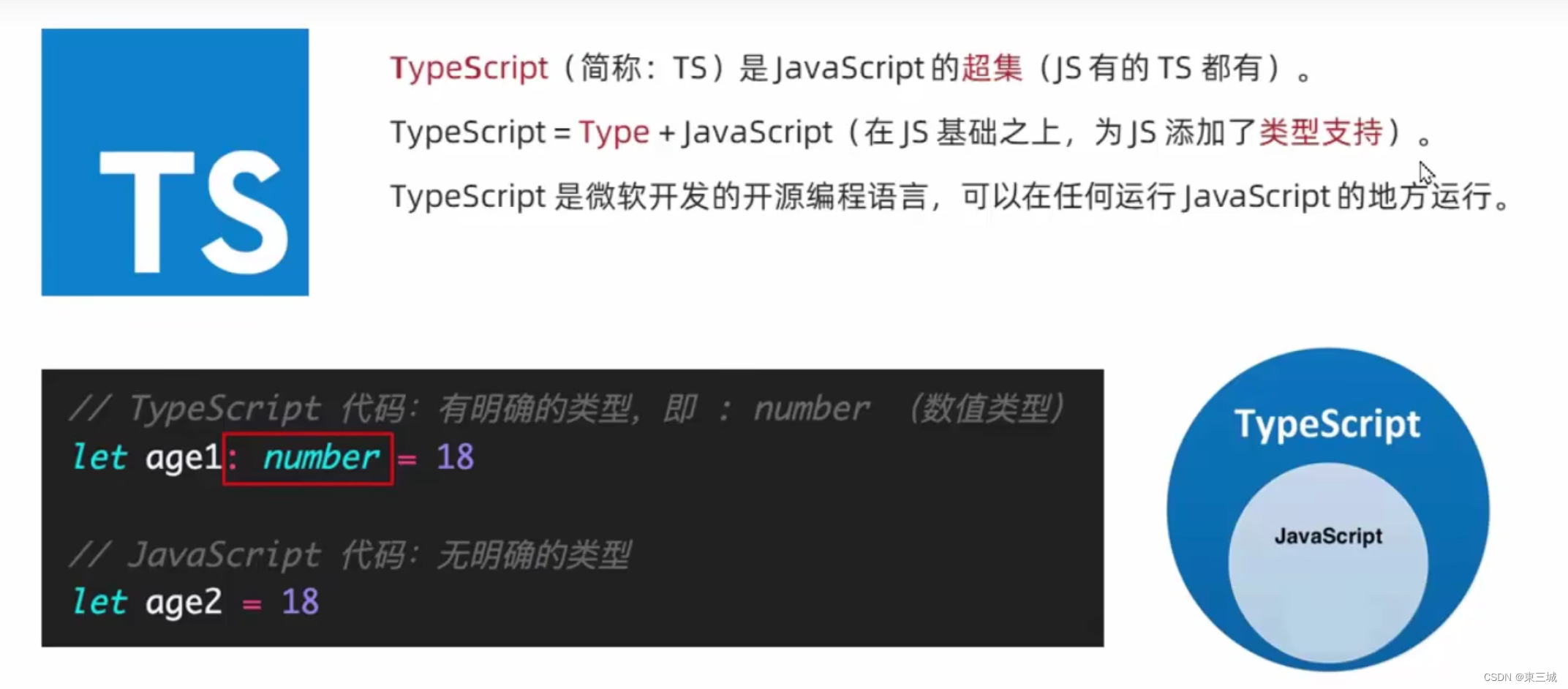
类型声明文件的使用说明

使用已有的类型声明文件

创建自己的类型声明文件

1.TypeScript介绍

typescript是js的超集，主要学习ts里面的原始类型、字面量类型、数组类型、函数类型、类类型、接口类型、类型别名、联合与交叉类型、枚举类型、泛型等类型元素，以及类型推断、类型断言、类型缩小、类型放大等特性。相较于js更加严谨,编写代码的时候静态类型的校验。



1.1.TypeScript为什么要为JS添加类型支持

背景:JS的类型系统有“先天缺陷”，JS代码中的大多数错误都是类型错误Uncaught TypeError。

问题:增加了发现和纠正bug的时间，严重影响了开发效率。

在编程语言的动态性方面，TypeScript是静态类型的编程语言，JS是动态类型的编程语言。

静态类型:在编译时进行类型检查;

动态类型:在执行过程中进行类型检查。

代码编译和代码执行顺序:1编译2执行。

对于JS:你需要等到代码实际执行时才发现错误(后期)。

对于TS:在编译代码时(在执行之前)发现错误。

并且，通过VSCode等开发工具，TS可以在编写代码时提前发现代码中的错误，减少了找Bug和改Bug的时间。

1.2.TypeScript相比JS优势

尽早发现错误(在编写代码时)，减少找Bug和改Bug的时间。，提高开发效率。

程序任何位置的代码都有代码提示，可以随时随地提供一种安全感，增强开发体验。

强大类型系统提高了代码的可维护性，使重构代码变得更容易。

支持最新的ECMAScript语法，优先体验最新的语法，让您走在前端技术的最前沿。

TS类型推断机制不需要在代码中处处显示注释类型，它允许您在最小化成本的同时享受其优势。

除此之外，随着vue3源码用TS重写，Angular默认支持TS, React和TS完美配合，TypeScript已经成为大中型前端项目的首选编程语言。

2.TypeScript初体验

2.1.安装编译TS的工具包

问题:为什么要安装编译TS的工具包?

回答:Node.js/ 浏览器，只认识JS代码，不认识TS代码。在运行TS代码之前，需要将其转换为JS代码。



全局安装命令:

npm i -g typescript@4.5.2

1

typescript包:编译TS代码的包，提供tsc命令转换TS->JS。

验证是否安装成功:tsc-v(查看typescript版本)

tsc -v

1

2.2.编译并运行TS代码

创建hello.ts文件(注意TS文件的后缀名为.ts)

将TS 编译 JS，在终端输入tsc hello.ts命令。(此时，一个同名的JS文件将出现在同级目录中)

tsc hello.ts

1

执行JS代码:在终端中输入命令node hello.js

node hello.js

1

说明：所有合法的JS代码都是TS代码，有JS基础只需要学习TS类型即可

注意：由TS编译生成的JS文件，代码中就没有类型信息了

2.3.简化运行TS代码

使用ts-node包，直接在node.js中执行代码

安装命令：

npm i -g ts-node

1

使用方式：

ts-node hello.ts

1

3.TypeScript常用类型

3.1.类型注解

let age: number = 24

1

说明：代码中 : number就是类型注解

作用：为变量添加约束条件，约定age的类型为number，如果赋值其他类型，就会报错

3.2.常用基础类型

可以将TS中常用基础类型细分为两类：

JS已有类型

原始类型：number/string/boolean/null/undefined/symbol

对象类型：object（包括数组，对象，函数等）

TS新增类型

联合类型，自定义类型（类型别名），接口，元组，字面量类型，枚举，void，any等

3.3.原始类型 number/string/boolean/null/undefined/symbol

let num: number = 1; // number

let str: string = "2"; // string

let bool: boolean = true; // boolean

let nul:null = null; // null

let undef: undefined = undefined; // undefined

let sy: symbol = Symbol(); // symbol

1

2

3

4

5

6

3.4.对象类型（数组）

对象类型在TS中更加细化，每个具体对象都有自己的类型语法

数组类型的两种写法

let strArr: string[] = ['a', 'b'] // 推荐使用这种

let numArr: Array<number> = [1, 2, 3]

1

2

数组中既有number类型，又有string类型

let numOrStrArr: (number | string)[] = [1, 'a']

1

解释：｜（竖线）在TS中叫做联合类型（由两个以上其他类型组成的类型，可以表示这些类型中任意一种）

let age: number | string = 20; // 表示既可以是number，又可以是string

let arr: (number | string)[] = [1, 3, 5, 'a', 'b'] // 添加小括号表示：首先是数组，数组中可以出现number或string

let arr1: number | string[] = ['a','b']

或

let arr1: number | string[] = 123 // 不添加小括号表示：可以是number，或者string数组

1

2

3

4

5

6

7

3.5.类型别名（自定义类型）

类型别名：为任意类型起别名

使用场景：当同一类型被多次使用时，可以通过类型别名，简化该类型的使用

格式：type 别名名称 = 类型定义

type CustomArray = (number | string)[]

let arr: CustomArray = [1, 'a']

1

2

3

3.6.函数类型

函数类型实际上之的是：函数参数和返回值的类型

方式一：单独指定参数，返回值的类型

function add(num1: number, num2: number): number {

return num1 + num2

}

const add1 = (num1: number, num2: number): number => {

return num1 + num2

}

1

2

3

4

5

6

7

方式二：同时指定参数，返回值类型

const add2: (num1: number, num2: number) => number = (num1, num2 ) => {

return num1 + num2

}

1

2

3

这种形式只适用于函数表达式

void类型

如果函数没有返回值，那么函数返回值类型为：void

function greet(name: string): void {

console.log('Hello', name)

}

1

2

3

可选参数

使用函数实现某些功能时，参数可传可不传，这时候就用到可选参数了

语法：在可传可不传的参数名称后面添加?（问号）

function mySlice(start?: number, end?: number): void {

console.log('开始：', start, '结束', end)

}

1

2

3

注意：可选参数只能出现在参数列表的最后，也就是说可选参数后面不能出现必选参数

3.7.对象类型

JS中的对象是由属性和方法构成，而TS中对象的类型就是在描述对象的结构（有什么类型的属性和方法）

let gf : { name: string; age: number; sayLove(): void} = {

name: 'yj',

age: 26,

sayLove() {

console.log("honey")

},

}

let gf1 : {

name: string

age: number

sayLove(): void

} = {

name: 'yj',

age: 26,

sayLove() {

console.log("honey")

},

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

直接使用{}来描述对象结构。采用=属性名：类型=的形式；方法采用方法名():返回值类型的形式

如果方法有参数，就在方法名后面的小括号中中指定参数类型（比如：greet(name:string):void）

在一行代码中指定对象的多个属性类型时，使用；（分号）来分隔。

如果一行代码只指定一个属性类型（通过换行来分隔多个属性类型），可以去掉；（分号）

方法的类型也可以使用箭头函数的形式（比如：{sayHi:()=>void}）

可选属性

对象的属性或方法，也可以是可选的， 此时就用到可选属性了。

比如，我们在使用 axios（{…}）时， 如果发送GET 请求， method 属性就可以省略。

function myAxios(config: { url: string; method?: string }){

console.log(config)

}

1

2

3

可选属性的语法与函数可选参数的语法一致，都使用？ 问号）来表示。

3.8.接口

当一个对象类型被多次使用时，一般会使用==接口（interface）==来描述对象的类型，达到复用的目的。

使用 interface关键字来声明接口

接口名称（比如，此处的IPerson），可以是任意合法的变量名称

声明接口后，直接使用接口名称作为变量的类型

因为每一行只有一个属性类型，因此，属性类型后没有；（分号）

interface IPerson {

name: string

age: number

sayHi(): void

}

let person: IPerson = {

name: 'dsc',

age: 24,

sayHi() {}

}

1

2

3

4

5

6

7

8

9

10

11

接口和类型别名对比

实际上，在大多数的情况下使用接口类型和类型别名的效果等价，但是在某些特定的场景下这两者还是存在很大区别。

重复定义的接口类型，它的属性会叠加，这个特性使得我们可以极其方便地对全局变量、第三方库的类型做扩展

如果我们重复定义类型别名，那么就会报错

接口继承

如果两个接口之间有相同的属性或方法， 可以将公共的属性或方法抽离出来， 通过继承来实现复用。

比如，这两个接口都有x、y两个属性， 重复写两次，可以，但很繁琐。

interface Point2D { x: number; y: number }

interface Point3D { x: number; y: number; z: number }

1

2

更好的方式：

interface Point2D { x: number;y:number }

interface Point3D extends Point2D { z: number }

1

2

解释：

使用extends（继承） 关键字实现了接口Point3D继承Point2D。

继承后，Point3D就有了Point2D的所有属性和方法（此时， Point3D同时有x、y、z三个属性）。

3.9.元组

场景：在地图中，使用经纬度坐标来标记位置信息。

可以使用数组来记录坐标，那么，该数组中只有两个元素，并且这两个元素都是数值类型。

let position: number[] = [39.5427 116.23171]

1

使用number[]的缺点：不严谨，因为该类型的数组中可以出现任意多个数字。

更好的方式：元组（Tuple） 。

元组类型是另一种类型的数组，它确切地知道包含多少个元素，以及特定索引对应的类型。

let position: [ number， number] = [39 .5427， 116.2317]

1

解释：

元组类型可以确切地标记出有多少个元素，以及每个元素的类型。

该示例中，元素有两个元素，每个元素的类型都是number。

3.10.类型推论

在TS中，某些没有明确指出类型的地方，TS的类型推论机制会帮助提供类型

换句话说：由于类型推论的存在，这些地方，类型注解可以省略不写！

发生类型推论的2种常见场景：

声明变量并初始化时

决定函数返回值时。

注意：这两种情况下，类型注解可以省略不写！

推荐：能省略类型注解的地方就省略（偷懒，充分利用TS类型推论的能力，提升开发效率）。

技巧：如果不知道类型，可以通过鼠标放在变量名称上，利用VSCode的提示来查看类型。

3.11.类型断言

有时候开发人员会比TS更加明确一个值的类型， 此时， 可以使用类型断言来指定更具体的类型。比如，

<a href="http://www.baidu.com/" id="link">百度</a>

1

注意： getElementById方法返回值的类型是 HTMLElement，该类型只包含所有标签公共的属性或方法，不包含 a标签 特有的 href 等属性。

因此，这个类型太宽泛（不具体）， 无法操作href等 a标签特有的属性或方法。

解决方式：这种情况下就需要使用类型断言指定更加具体的类型。

使用类型断言：

解释：

使用as关键字实现类型断言

关键字as 后面的类型是一个更加具体的类型 （HTMLAnchorElement是 HTMLElement 的子类型）

通过类型断言，aLink的类型变得更加具体， 这样就可以访问a标签特有的属性或方法了

另一种语法，使用<>语法， 这种语法形式不常用知道即可：

技巧：在浏览器控制台，通过console.dir（） 打印 DOM 元素，在属性列表的最后面， 即可看到该元素的类型。

3.12.字面量类型

思考以下代码， 两个变量的类型分别是什么？

let str1 = 'Hello TS'

const str2 = Hello TS'

1

2

通过TS类型推论机制，可以得到答案：

变量 str1 的类型为：string

变量str2的类型为：'Hello TS'

解释：

str1 是一个变量（let）， 它的值可以是任意字符串，所以类型为： string

str2 是一个常量（const）， 它的值不能变化只能是’Hello TS’，所以， 它的类型为：'Hello TS'

注意：此处的'Hello TS'， 就是一个字面量类型。 也就是说某个特定的字符串也可以作为TS 中的类型

除字符串外，任意的JS字面量（比如， 对象、数字等）都可以作为类型使用。

使用模式：字面量类型配合联合类型一起使用

使用场景： `用来表示一组明确的可选值列表``

比如，游戏的方向可选值上下左右中任意一个

function changDirection(direction: 'up' | 'down' | 'left' | 'right') {

console.log(direction)

}

1

2

3

相比于string类型，使用字面量类型更加精确，严谨

3.13.枚举类型

枚举的功能类似字面量类型+联合类型组合的功能，也可以表示一组明确的可选值。

枚举：定义一组命名常量。它描述一个值，该值可以是这些常量中的一个

enum Direction { Up, Down, Left, Right }

enum Direction {

Up = 'UP',

Down = 'DOWN',

Left = 'LEFT',

Right = 'RIGHT'

}

1

2

3

4

5

6

7

8

枚举的作用在于定义被命名的常量集合，一个默认从 0 开始递增的数字集合，称之为数字枚举。也可以指定值，这里可以指定的值可以是数字或者字符串。

enum Days {

Sunday = 1,

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday

}

let day = Days.Sunday;

1

2

3

4

5

6

7

8

9

10

字符串枚举没有自增长行为，因此，字符串枚举的每个成员必须有初始值

3.14.any类型

原则：不推荐使用any! 这会让TypeScript变为"AnyScript"(失去TS类型保护优势)

因为当值的类型为any时，可以对该值进行任何操作，并且不会有代码提示

以上代码都不会有任何类型的错误提示，即使存在错误

尽可能避免使用any类型，除非临时使用any来“避免”书写很长，很复杂的类型

其他隐式具有any类型的情况：

声明变量不提供类型也不提供默认值

函参数参数不加类型

3.15.typeof

typeof 的主要用途是在类型上下文中获取变量或者属性的类型(类型查询)

// 推断变量的类型

let strA = "2";

type KeyOfType = typeof strA; // string

// 反推出对象的类型作为新的类型

let person = {

name: '张三',

getName(name: string):void {

console.log(name);

}

}

type Person = typeof person;

// 根据已有变量的值，获取该值类型，简化类型书写

let p = { x: 1, y: 2 }

function fornatPoint(point: typeof p) {}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

4.TypeScript高级类型

4.1. class类

TypeScript全面支持ES205中引入的class关键字，并为其添加了类型注解和其他语法（比如可见性修饰符）

class基本使用：

根据TS中类型推论，可以知道Person类的实例对象p的类型是Person

TS中的class，不仅提供了class的语法功能，也作为一种类型存在

实例属性初始化：

声明成员age，类型为number（没有初始值）

声明成员gender，并设置初始值，此时，可省略类型注解（TS类型推论为string类型）

实例方法：

方法的类型注解（参数和返回值）与函数用法相同

构造函数

class Person {

age: number

gender: string

constructor(age: number, gender: string) {

this.age = age

this.gender = gender

}

}

1

2

3

4

5

6

7

8

9

成员初始化后，才可以通过this来访问实例成员

需要为构造函数指定类型注解，否则会被隐式推断为any；构造函数不需要返回值类型

继承extends，implement

class Animal {

move() {

console.log('move')

}

}

class Dog extends Animal {

bark() {

console.log("汪汪汪")

}

}

const dog = new Dog()

1

2

3

4

5

6

7

8

9

10

11

12

13

通过extends关键字实现继承

子类Dog继承父类Animal，则Dog的实例对象dog就同时具有了父类Animal和子类Dog的所有属性和方法

interface Singable {

sing(): void

}

class Person implements Singable {

sing() {

console.log("你我山前没相见山后别相逢")

}

}

1

2

3

4

5

6

7

8

9

通过implements关键字让class实现接口

Person类中必须提供Singable接口中指定的所有方法和属性

可见性修饰符public，protected，private

类成员可见性：可以用TS来控制class的方法或属性对于class外的代码是否可见

可见性修饰符包括：public（公有的），protected（受保护的），private（私有的）

public:公有成员可以被任何地方访问，默认可见性，可以直接省略

protected:表示受保护的，仅对其声明所在类和子类中（非实例对象）可见，实例不可见

private:表示私有的，只在当前类可见，对实例对象以及子类也是不可见

只读修饰符readonly

readonly:表示只读，用来防止在构造函数之外对属性进行赋值

readonly只能修饰属性，不能修饰方法

属性后面类型注解如果不加，则类型为字面量类型

接口或者{}表示的对象类型，也可以使用readonly

4.2.类型兼容性

两种类型系统：1.StructuralType System（结构化类型系统）2 .NominalType System（标明类型系统）

TS 采用的是结构化类型系统，也叫做 ducktyping（鸭子类型），类型检查关注的是值所具有的形状。

也就是说，在结构类型系统中，如果两个对象具有相同的形状，则认为它们属于同一类型。

类之间兼容性

Point 和 Point2D 是两个名称不同的类。

变量 p 的类型被显示标注为 Point 类型，但是它的值却是 Point2D 的实例，并且没有类型错误。

因为 TS 是结构化类型系统，只检查 Point 和 Point2D 的结构是否相同（相同，都具有 x 和 y 两个属性，属性类型也相同）。

但是，如果在 Nominal Type System 中（比如，C#、Java 等），它们是不同的类，类型无法兼容。

在结构化类型系统中，如果两个对象具有相同的形状，则认为它们属于同一类型，这种说法不准确。更准确的说法：对于对象类型来说，y的成员至少与x相同，则x兼容y（成员多的可以赋值给少的）

接口之间兼容性

接口之间兼容性类似于class，并且class和interface之间也可以兼容

函数直接兼容性

函数之间兼容性比较复杂，需要考虑：1参数个数2参数类型3返回值类型

参数个数，参数多的兼容参数少的

参数类型，相同位置的参数类型要相同（原始类型）或兼容（对象类型）

返回值类型，只关注返回值类型本身即可

如果返回值类型是原始类型，此时两个类型要相同

如果返回值类型是对象类型，此时成员多的可以赋值给成员少的

4.3.交叉类型（&）

功能类似于接口继承，用于组合多个类型为一个类型（常用于对象类型）

interface Person { name: string }

interface Contact { phone: string }

type PersonDetail = Person & Contact

let obj: PersonDetail = {

name: 'dsc',

phone: '176...'

}

1

2

3

4

5

6

7

8

交叉类型和接口继承的对比

相同点：都可以实现对象类型的组合

不同点：两种方式实现组合时，对于同名属性间，处理类型冲突的方式不同

以上代码，接口继承会报错（类型不兼容）；交叉类型没有错误，可以简单理解为：

方法重载

fn: (value: string | number) => string

1

4.4.泛型 和 keyof

泛型

泛型是可以在保证类型安全前提下，让函数等与多种类型一起工作，从而实现复用，常用于：函数、接口、class 中。需求：创建一个 id 函数，传入什么数据就返回该数据本身（也就是说，参数和返回值类型相同）。

function id (value: number): number { return value }

1

比如，id (10) 调用以上函数就会直接返回 10 本身。但是，该函数只接收数值类型，无法用于其他类型。

为了能让函数能够接受任意类型，可以将参数类型修改为 any。但是，这样就失去了 TS 的类型保护，类型不安全。

function id (value: any): any { return value }

1

泛型在保证类型安全（不丢失类型信息）的同时，可以让函数等与多种不同的类型一起工作，灵活可复用。实际上，在 C#和 Java 等编程语言中，泛型都是用来实现可复用组件功能的主要工具之一。

创建泛型函数

function id<Type> (value: Type): Type {return value}

1

语法：在函数名称的后面添加<>（尖括号），尖括号中添加类型变量，比如此处的 Type。

类型变量 Type，是一种特殊类型的变量，它处理类型而不是值。

该类型变量相当于一个类型容器，能够捕获用户提供的类型（具体是什么类型由用户调用该函数时指定）。

因为 Type 是类型，因此可以将其作为函数参数和返回值的类型，表示参数和返回值具有相同的类型。

类型变量 Type，可以是任意合法的变量名称。

调用泛型函数

const num = id<number>(10)

const str = id<string>('a')

1

2

语法：在函数名称的后面添加<>（尖括号），尖括号中指定具体的类型，比如，此处的 number。

当传入类型 number 后，这个类型就会被函数声明时指定的类型变量 Type 捕获到。

此时，Type 的类型就是 number，所以，函数 id 参数和返回值的类型也都是 number。

同样，如果传入类型 string，函数 id 参数和返回值的类型就都是 string。

这样，通过泛型就做到了让 id 函数与多种不同的类型一起工作，实现了复用的同时保证了类型安全。

简化泛型函数调用

let num = id(10)

1

在调用泛型函数时，可以省略<类型>来简化泛型函数的调用。

此时，TS 内部会采用一种叫做类型参数推断的机制，来根据传入的实参自动推断出类型变量 Type 的类型。

比如，传入实参 10，TS 会自动推断出变量 num 的类型 number，并作为 Type 的类型。

推荐：使用这种简化的方式调用泛型函数，使代码更短，更易于阅读。

说明：当编译器无法推断类型或者推断的类型不准确时，就需要显式地传入类型参数。

泛型约束

泛型约束：默认情况下，泛型函数的类型变量 Type 可以代表多个类型，这导致无法访问任何属性。

比如，id (‘a’）调用函数时获取参数的长度：

function id<Type>(value: Type): Type{

console.log (value. length)

return value

}

1

2

3

4

解释：Type 可以代表任意类型，无法保证一定存在 length 属性，比如 number 类型就没有 length。

此时，就需要为泛型添加约束来收缩类型（缩窄类型取值范围）。

方式一：指定更加具体的类型

function id<Type>(value: Type[]): Type[] {

console.log(value.length)

return value

}

1

2

3

4

方式二：添加约束

interface ILength { length: number }

function id<Type extends ILength>(value: Type): Type {

console.log(value.length)

return value

}

1

2

3

4

5

该约束表示：传入的类型必须具有length属性

泛型的类型变量可以有多个，并且类型变量之间还可以约束（比如，第二个类型变量受第一个类型变量约束）。比如，创建一个函数来获取对象中属性的值：

function getProp<Type, Key extends keyof Type> (obj: Type, key: Key){

return obj [key]

}

let person = { name: 'jack', age: 18 }

getProp (person, 'name')

1

2

3

4

5

6

解释：

添加了第二个类型变量 Key，两个类型变量之间使用（，）逗号分隔。

Keyof 关键字接收一个对象类型，生成其键名称（可能是字符串或数字）的联合类型。

本示例中 keyof Type 实际上获取的是 person 对象所有键的联合类型，也就是：‘name’|‘age’。

类型变量 Key 受 Type 约束，可以理解为：Key 只能是 Type 所有键中的任意一个，或者说只能访问对象中存在的属性。

泛型接口

接口也可以配合泛型来使用，以增加其灵活性，增加其复用性

在接口名称的后面添加<类型变量>，那么，这个接口就变成了泛型接口。

接口的类型变量，对接口中所有其他成员可见，也就是接口中所有成员都可以使用类型变量。

使用泛型接口时，需要显式指定具体的类型（比如，此处的 IdFunc < number>）。

此时，id 方法的参数和返回值类型都是 number; ids 方法的返回值类型是 number[ ]。

泛型类

class也可以配合泛型来使用

比如，React的class组件的基类Component就是泛型类，不同的组件有不同的props和state

创建泛型类

class GenericNumber<NumType> {

defaultValue: NumType

add: (x:NumType, y:NumType) => NumType

}

1

2

3

4

使用泛型类

const myNum = new GenericNumber<number>()

myNum.defaultValue = 10

1

2

泛型工具类型Partial，Readonly，Pick，Record

Partial< Type> 用来构造（创建）一个类型，讲Type的所有属性设置为可选

interface Props {

id: string

children: number[]

}

type PartialProps = Partial<Props>

1

2

3

4

5

构造出来的新类型PartialProps结构和Props相同，但所有属性都变为可选的

Readonly< Type> 用来构造（创建）一个类型，讲Type的所有属性设置为只读

interface Props {

id: string

children: number[]

}

type PartialProps = Partial<Props>

let props: ReadonlyProps = { id:'1', children: []}

props.id = '2' // 报错，无法分配，因为它是只读

1

2

3

4

5

6

7

构造出来的新类型PartialProps结构和Props相同，但所有属性都变为只读的

Pick <Type, Keys> 从 Type 中选择一组属性来构造新类型

interface Props {

id: string

title: string

children: number[]

}

type PickProps = Pick<Props, 'id' | 'title'>

1

2

3

4

5

6

Pick 工具类型有两个类型变量：1 表示选择谁的属性 2 表示选择哪几个属性。

其中第二个类型变量，如果只选择一个则只传入该属性名即可。

第二个类型变量传入的属性只能是第一个类型变量中存在的属性。

构造出来的新类型 PickProps，只有 id 和 title 两个属性类型。

Record <Keys, Type> 构造一个对象类型，属性键为Keys，属性类型为Type

type RecordObj = Record<'a' | 'b' | 'c', string[]>

let obj:RecordObj = {

a: ['1'],

b: ['2'],

c: ['3']

}

1

2

3

4

5

6

Record工具类型有两个类型变量：1表示对象有哪些属性2表示对象属性的类型

构建的新对象类型RecordObj表示：这个对象有三个属性分别为a/b/c，属性值的类型都是string[]

4.5.索引签名类型 和 索引查询类型

索引签名类型

绝大多数情况下，我们都可以在使用对象前就确定对象的结构，并为对象添加准确的类型。

使用场景：当无法确定对象中有哪些属性（或者说对象中可以出现任意多个属性），此时，就用到索引签名类型了。

使用[key: string]来约束该接口中允许出现的属性名称。表示只要是 string 类型的属性名称，都可以出现在对象中。

这样，对象 obj 中就可以出现任意多个属性（比如，a、b等）。

key 只是一个占位符，可以换成任意合法的变量名称。

隐藏的前置知识：JS 中对象（{}）的键是 string 类型的。

在JS中数组是一类特殊的对象，特殊在数组的键（索引）是数值类型

映射类型

映射类型：基于旧类型创建新类型（对象类型），减少重复、提升开发效率。

比如，类型PropKeys有x/y/z,另一个类型Type1中也有x/y/z,并且Type1中x/y/z的类型相同：

type PropKeys = 'x' | 'y' | 'z'

type Type1 = { x: number; y: number; z: number }

1

2

3

这样书写没错，但x/y/z重复书写了两次。像这种情况，就可以使用映射类型来进行简化。

type PropKeys = 'x' | 'y' | 'z'

type Type2 ={ [Key in PropKeys]: number }

1

2

3

映射类型是基于索引签名类型的，所以，该语法类似于索引签名类型，也使用了[ ]。

Key in PropKeys 表示 Key 可以是 PropKeys 联合类型中的任意一个，类似于 forin (let k in obj）。

使用映射类型创建的新对象类型 Type2 和类型 Type1 结构完全相同。

注意：映射类型只能在类型别名中使用，不能在接口中使用。

映射类型除了根据联合类型创建新类型外，还可以根据对象类型来创建

type Props = { a: number; b: string; c: boolean }

type Type3 = { [key in keyof Props]: number }

1

2

首先执行`keyof Props获取对象类型Props中所有键的联合类型，即’a’|‘b’|‘c’

然后key in ...就表示key可以是Props中所有键的任意一个

泛型工具类Partial实现

索引查询类型

刚刚用到的T[P]语法，在TS中叫做索引查询（访问）类型

作用：用来查询属性的类型

type Props = { a: number; b: string; c: boolean }

type TypeA = Props['a']

1

2

Props['a']表示查询类型Props中属性a对应的类型number，所以TypeA的类型为number

[ ]中的属性必须存在于被查询类型中,否则报错

同时查询多个索引的类型

type Props = { a: number; b: string; c: boolean }

type TypeA = Props['a'｜'b'] // 结果为： string ｜ number

type TypeB = Props[keyof Props] // 结果为： string ｜ number | boolean

1

2

3

4

5.TypeScript类型声明文件

今天几乎所有的 JavaScript 应用都会引入许多第三方库来完成任务需求。

这些第三方库不管是否是用 TS 编写的，最终都要编译成 JS 代码，才能发布给开发者使用。

我们知道是 TS 提供了类型，才有了代码提示和类型保护等机制。

但在项目开发中使用第三方库时，你会发现它们几乎都有相应的 TS 类型，这些类型是怎么来的呢？类型声明文件类型声明文件：用来为已存在的 JS 库提供类型信息。

这样在 TS 项目中使用这些库时，就像用 TS 一样，都会有代码提示、类型保护等机制了。

TS 的两种文件类型

TS 中有两种文件类型：1.ts文件 2.d.ts 文件。

.ts 文件：

既包含类型信息又可执行代码。

可以被编译为.js 文件，然后，执行代码。

用途：编写程序代码的地方。

.d.ts 文件：

只包含类型信息的类型声明文件。

不会生成 .js 文件，仅用于提供类型信息。

用途：为js 提供类型信息。

总结：.ts 是 implementation（代码实现文件）.d.ts 是 declaration（类型声明文件) 。如果要为 JS 库提供类型信息，要使用.d.ts 文件。

类型声明文件的使用说明

使用已有的类型声明文件

使用已有的类型声明文件：1 内置类型声明文件 2 第三方库的类型声明文件。

内置类型声明文件：TS 为 JS 运行时可用的所有标准化内置 API 都提供了声明文件。比如，在使用数组时，数组所有方法都会有相应的代码提示以及类型信息：

实际上这都是 TS 提供的内置类型声明文件。

可以通过 Ctrl+鼠标左键（Mac: option+鼠标左键）来查看内置类型声明文件内容。

比如，查看 forEach 方法的类型声明，在 VSCode中会自动跳转到lib.es5…d.ts 类型声明文件中。当然，像 window、document 等 BOM、DOMAPI也都有相应的类型声明（lib.dom.d.ts）。

第三方库的类型声明文件：目前，几乎所有常用的第三方库都有相应的类型声明文件。第三方库的类型声明文件有两种存在形式：1 库自带类型声明文件 2 由 DefinitelyTyped 提供。

库自带类型声明文件：比如，axios。

这种情况下，正常导入该库，TS 就会自动加载库自己的类型声明文件，以提供该库的类型声明。

由DefinitelyTyped提供。

DefinitelyTyped 是一个 github 仓库，用来提供高质量 TypeScript 类型声明。

可以通过npm/yarn来下载该仓库提供的TS类型声明包，这些包的名称格式为：@types/\*。 比如，@types/react、@types/lodash 等。

说明：在实际项目开发时，如果你使用的第三方库没有自带的声明文件，VSCode 会给出明确的提示。

解释：当安装@types/\*类型声明包后，TS 也会自动加载该类声明包，以提供该库的类型声明。

创建自己的类型声明文件

创建自己的类型声明文件：1 项目内共享类型 2 为已有 JS 文件提供类型声明。

项目内共享类型

如果多个.ts 文件中都用到同一个类型，此时可以创建.d.ts 文件提供该类型，实现类型共享。

操作步骤：

创建index.d.ts 类型声明文件。

创建需要共享的类型，并使用export导出(TS中的类型也可以使用import/export实现模块化功能）。

在需要使用共享类型的.ts 文件中，通过 import导入即可（.d.ts 后缀导入时，直接省略）。

为已有 JS 文件提供类型声明

在将 JS 项目迁移到 TS 项目时，为了让已有的 js 文件有类型声明。

成为库作者，创建库给其他人使用。

注意：类型声明文件的编写与模块化方式相关，不同的模块化方式有不同的写法。但由于历史原因，JS 模块化的发展经历过多种变化（AMD、CommonJS、UMD、ESModule 等），而 TS 支持各种模块化形式的类型声明。这就导致，类型声明文件相关内容又多又杂。

说明：在导入.js 文件时，TS 会自动加载与.js 同名的.d.ts 文件，以提供类型声明。

declare 关键字：用于类型声明，为其他地方（比如，。Js 文件）已存在的变量声明类型，而不是创建一个新的变量。

对于 type、interface 等这些明确就是 TS 类型的（只能在 TS 中使用的），可以省略 declare 关键字。

对于 let、function 等具有双重含义（在 JS、TS 中都能用），应该使用 declare 关键字，明确指定此处用于类型声明。

————————————————

版权声明：本文为CSDN博主「東三城」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/qq\_40440961/article/details/128931213