**TS汇总文档**

**前言**

**1. 仓库地址**

<https://git.ywwl.com/ywfe/yw-share>

**2. 分享步骤**

a、ts 配置升级；

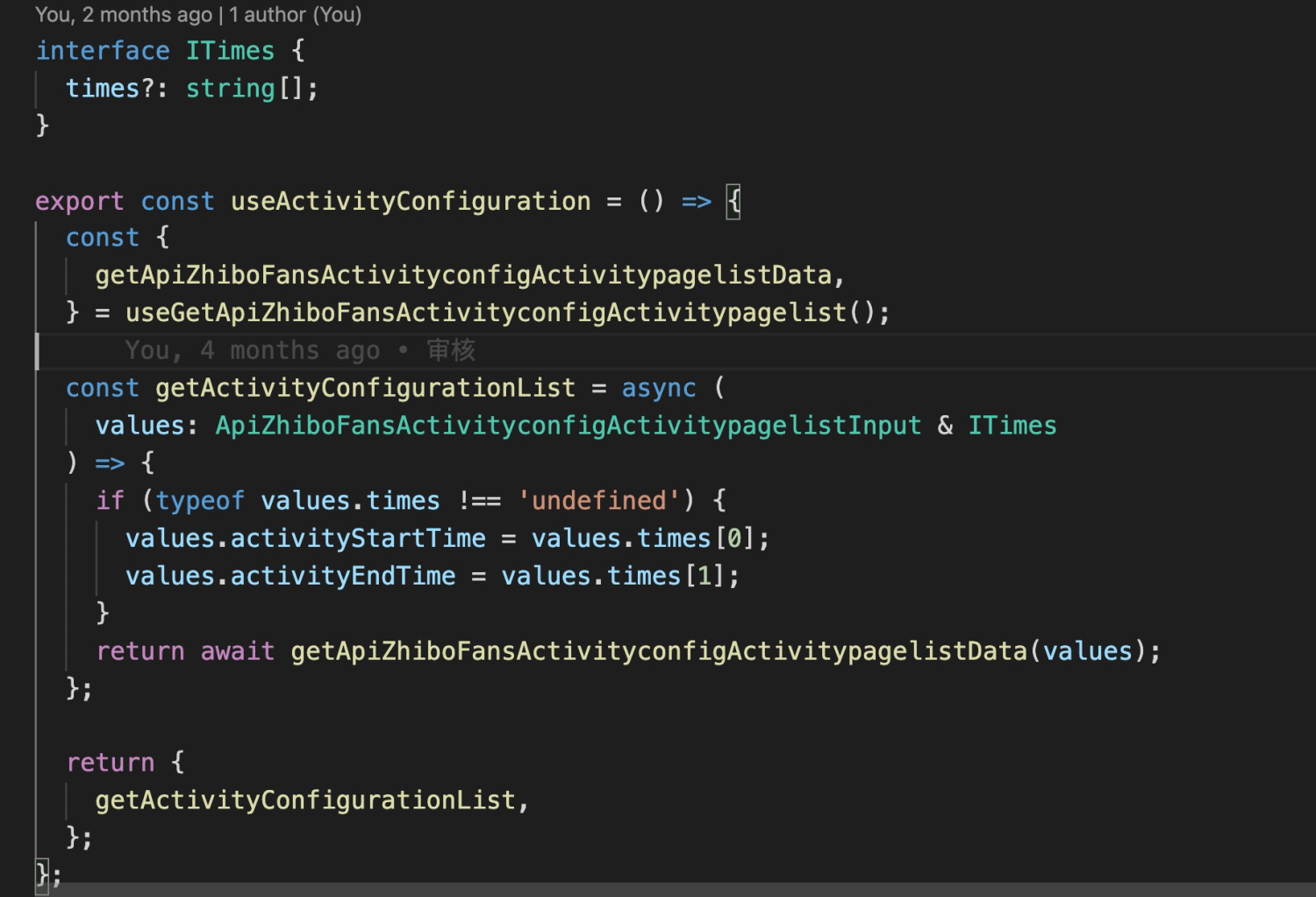
b、一. 1、5、10、11、12

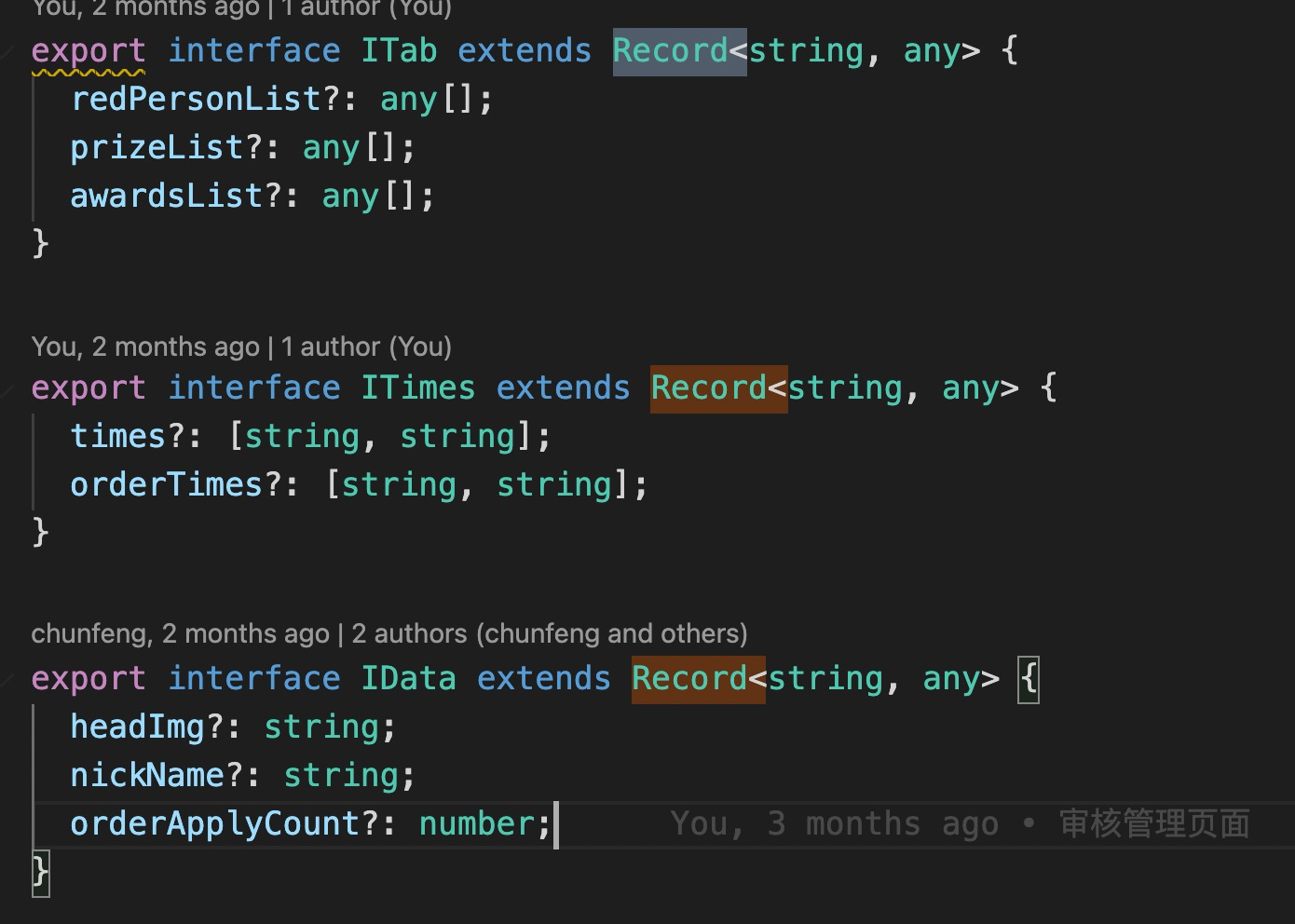
二. 8

c、六，

d、九大内置类型

e、代码中





**一、基础**

**1. 声明变量**

备注: **必须声明类型**

let str: string;let strArray: string[] | undefined;let num: number = 12;let arr1: number[] = [1, 2, 3];let arr2: Array<number> = [1,2,3];let strAndNum: string | number;let strAndNumArray: (string | number)[] = [1, 2, 3, "hello world"];

interface ITest { type?: number; name: string;}

let test:ITest = { type: 1, name: "Tom"}

a、这个是 **pack** 错误频率最高之一；  
b、错误举例

42 let test; ~~~~src/pages/PublishGoodsConfig/index.tsx:50:15 - error TS6133: 'testNumber' is declared but its value is never read.50 const testNumber: any = test; ~~~~~~~~~~src/pages/PublishGoodsConfig/index.tsx:50:33 - error TS7005: Variable 'test' implicitly has an 'any' type.50 const testNumber: any = test;

**2. 元组类型**

元组(Tuple)表示一个已知 **数量** 和 **类型** 数组，一般用于表示一个固定的结构

interface ITest { type?: number; name: string;}

let positiion: [number, number] = [100, 100];let tuple1: [ITest, number] = [ { type: 1, name: "Tom" }, 10086]

**3. enum 普通枚举**

事先考虑某一个变量的所有可能的值，尽量用自然语言中的单词表示它的每一个值

enum Gender { BOY, GIRL}

enum Week { MONDAY = 1, TUESDAY = 2}

// 用法const boy: number = Gender.BOY;

编译之后的文件 ，编译之后就是一个对象

var Gender;(function (Gender) { Gender[Gender["BOY"] = 0] = "BOY"; Gender[Gender["GIRL"] = 1] = "GIRL";})(Gender || (Gender = {}));

var Week;(function (Week) { Week[Week["MONDAY"] = 1] = "MONDAY"; Week[Week["TUESDAY"] = 2] = "TUESDAY";})(Week || (Week = {}));

**4. enum 常数枚举**

常数枚举与普通枚举的区别是，它会在编辑阶段被删除，并且不能包含计算成员；

const enum Color { RED, YELLOW, BLUE}

const colors = [Color.RED, Color.YELLOW, Color.YELLOW];

// 错误调用方法const colors = [Color.RED, Color.YELLOW, Color[0]];

编译之后的文件

var colors = [0 /\* RED \*/, 1 /\* YELLOW \*/, 2 /\* BLUE \*/];

**5. ！强制断言**

**！** 非空断言表示告诉 TS，这个属性肯定存在，让后续代码放心执行

const root: HTMLElement | null = document.getElementById("root");

// 1. 非空断言root!.style.color = "red";

// 2. 强制转换(root as any).style.color = "red";

// 3. demointerface IColors { red?: string;}const colors01: IColors = { red: "Red"}

// 表示一定存在，如果不添加 ！ 编译出错；let red:string = colors01.red!;

**6. as 类型断言**

可以把一个联合类型指定一个更具具体多类型

**7. never**

never 三种情况

a、函数的返回值永远拿不到的;

b、Unreachable code detected, 永远无法到达的代码;

c、严格模式，函数执行不了到地方；

function getName(): never { const a = 10010; throw new Error("can not get name"); // 在 vsCode 下面注意此行代码颜色 const b = 10086;}

function count(): never { const a = 10010; while(true){ console.log("hello world"); } // 在 vsCode 下面注意此行代码颜色 const b = 10086;}

function fn(x: string | number) { if (typeof x === "string") { console.log(x); } else if (typeof x === "number") { console.log(x); } else { // parameter x: never console.log(x); }}

**8. 字面量类型 & 联合类型**

a、字面量类型：只能是某 **几个字符串** 中的一个；

b、联合类型： 联合类型表示取值可以为多种类型中的一种；

// 字面量类型let v1: 1 | 2 | 4;

// 联合类型let v2: string | number;

**9. 不同类型 ！使用**

在未初始化时，直接使用场景

interface Empty<T> { }

let x: Empty<string>;let y!: Empty<number>;

// 在赋值前使用了变量“y”。x = y;

class Animal { // 属性“name”没有初始化表达式，且未在构造函数中明确赋值 name!: string;}

const animal = new Animal();animal.name = "Dog";

let num1!: number;let num2!: number;num2 = num1;

**10. any & unknown**

a、 any 类型类似于纯 JavaScript 的工作方式。我们有时可能需要描述一个我们根本不知道类型的变量。

在 TypeScript 中，任何东西可以赋值给 any 。它通常被称为 **top type** 。

b、 通过使用 any，我们破坏了 TypeScript 的能力引起了一些麻烦。没有强制的类型检查，可能会给您带来一些麻烦；

const uncertain: any = 'Hello world!';uncertain.hello();

c、unknown 说明

<https://zhuanlan.zhihu.com/p/104296850>

**11. typeof**

let p1 = { name: "Tom", age: 1, talk: () => {}}

type PersonType = typeof p1;

let p2: PersonType = { name: "小明", age: 12, talk: () => {}}

**12. keyof**

获取 key 值

interface Person { name: string; age: number;}

function getValueByKey(p: Person, key: keyof Person){ return p[key];}

const keys = getValueByKey({ name: "Tom", age: 10086}, "age");

console.log(keys);

/\*\* 运算顺序\* 1. keyof Person\* 2. key in \*/type OptionalPerson1 = { [key in keyof Person]?: Person[key]}

type OptionalPerson2 = Partial<Person>;

let p2: OptionalPerson1 = { name: "小名", age: 10010}

**二、函数 & 类**

**1. type 函数类型**

实现了声明类型可以不用 **指定函数参数类型和返回值类型**

type GetUsernameType = (x: string, y: string) => string;

let getUsername01: GetUsernameType = (x: string, y: string): string => { return x + y;}

let getUsername02: GetUsernameType = (x, y) => { return x + y;}

**2. 可选参数**

可选参数必须在必填参数后面

let getPerson = (name: string, age: number, sex?: number) => {}

**2. 剩余参数**

剩余参数必须在所有参数最后

let sum = (initNumber: number, n1: number, n2?: number, ...rest: Array<number>): number => { if (typeof n2 === "number") { return rest.reduce((a, b) => a + b, initNumber + n1 + n2); } else { return rest.reduce((a, b) => a + b, initNumber + n1); }}

**3. 类的 get & set**

可以尝试看看编译之后的代码

class Person { myname: string;

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

get name(): string{ return this.myname; }

set name(newValue: string){ this.name = newValue.toUpperCase(); }}

**4. 类 readonly**

只在编译阶段起作用

class Animal { public readonly name: string = "Tom";}

**5. 修饰符**

a、 公有 在类里面、子类、类外部都可以访问；

b、 保护类型 在类里面 子类里面可以访问，在类外部没法访问；

c、 私有 在类里面可以访问，在子类、类外部都没法访问；

class Father { constructor(public name: string, protected age: number, private type: number) { this.name = name; this.age = age; this.type = type; }}

**6. 静态属性、方法**

可以继承

class Father { static className = "Father"; static getClassName() { return Father.className; }}

class Child extends Father {

}console.log(Child.getClassName());

**7. 类的类型 type**

写一个类的时候，其实得到两个类型

a、第一个类型，就是实例的类型；

b、第二个类型，就是构造函数类型；

class Person { name: string = "zhufeng"; constructor(name: string, public age: number) { this.name = name; this.age = age; }}

const p1: Person = { name: "Tom", age: 100}

type PersonType = typeof Person;

const p2: PersonType = Person;

**8. 装饰器**

装饰器是一种特殊类型的声明，它能够被附加到类声明、方法、属性或参数上，可以修改类的行为。

常见的装饰器有 **类装饰器** 、 **属性装饰器** 、 **方法装饰器** 、 **参数装饰器** 。

**8.1. 类装饰器01**

a、不修改类源代码时，扩充 类 的方法和属性

b、装饰器参数是 类的构造函数

**执行顺序：**

1.如多个参数装饰器时，从最后一个依次向前执行；

2.方法和方法参数中的装饰器先执行；

3.类装饰器总最后执行；

4.方法和属性装饰器，谁在前谁先执行。

function enhancer(target: any) { target.prototype.name = "zhufeng"; target.prototype.getName = function(){ console.log(this.name); };}

@enhancerclass Person {

}

const p1 = new Person();(p1 as any).getName();

**8.2. 类装饰器02**

直接覆盖类，返回一个新的类

function connect(target: any) { console.log(target); return class { name: string = "Tom"; age: number = 18;

getName() { console.log(this.name); } }}

@connectclass App { getName() { console.log("this.name"); }}

const a1 = new App();

a1.getName(); // 调用的是第7行第方法

**8.3. 属性装饰器**

a、修改类的属性，行为

b、参数 **target: any, propertyKey: string**

/\*\* \* 属性类装饰器 \* @param target 原型 \* @param propertyKey 装饰的属性名称 \*/function uppercase(target: any, propertyKey: string) { let value = target[propertyKey]; const getter = function () { return value; } const setter = function (newValue: string) { value = newValue.toUpperCase(); } if (delete target[propertyKey]) { Object.defineProperty(target, propertyKey, { get: getter, set: setter, enumerable: true, configurable: true }) }}

class Person { @uppercase name: string = "yao\_wang";}

const p1 = new Person();console.log(p1.name); // YAO\_WANG

**8.4 方法装饰器**

修改方法的行为

/\*\* \* 方法装饰器 -- 修改属性描述器 \* @param target 原型 \* @param methodName 方法名称 \* @param descriptor 属性描述器 \*/function noEnumerable(target: any, methodName: string, descriptor: PropertyDescriptor) { descriptor.enumerable = false;}

/\*\* \* 方法装饰器 -- 修改参数 \* @param target 原型 \* @param methodName 方法名称 \* @param descriptor 属性描述器 \*/function convertToNumber(target: any, methodName: string, descriptor: PropertyDescriptor) { let oldMethod = target[methodName]; descriptor.value = function (...args: any[]) { args = args.map((item: any) => parseInt(item)); return oldMethod.apply(this, args); }}

class Person { name: string = "yao\_wang";

@noEnumerable getName() { console.log(this.name); }

@convertToNumber sum(...args: any[]) { return args.reduce((total: number, current: number) => total + current, 0); }}

const p1 = new Person();console.log("sum", p1.sum(1, "2", 3, "4"));

**8.5 参数装饰器**

a、静态成员 => 构造函数

b、普通成员 => 类的原型

// 扩展 Person 的实例属性, 特殊用法interface Person { age: number;}

/\*\* \* 参数装饰器 \* @param target 构造函数 \* @param methodName 方法名称 \* @param paramIndex \*/function addHome(target: any, methodName: string, paramIndex: number) { console.log(target, methodName, paramIndex); target.home = "北京";}

/\*\* \* 参数装饰器 \* @param target 原型 \* @param methodName 方法名称 \* @param paramIndex \*/function addAge(target: any, methodName: string, paramIndex: number) { console.log(target, methodName, paramIndex); target.age = 10;}

class Person {

static getClassName(@addHome name: string) { return `${(Person as any).home} ` + name; }

login(@addAge username: string, password: string) { console.log(this.age, username, password); }}

const p1 = new Person();console.log(p1.login("tom", "10011"), p1);

const s = Person.getClassName("欢迎你");console.log(s);

**9. 抽象类**

抽象描述一种抽象的概念，只能被继承

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

getName() {}

abstract speak(): void;}

class Dog extends Animal { speak(): void { throw new Error("Method not implemented."); }}

**11. test**

**三 、接口**

**1. 函数 & 对象**

// 函数接口interface IFunction { (a: number, b: number): number;}

const fn1: IFunction = (a, b) => { return a + b;}

// 对象接口interface IObject { add: (a: number, b: number) => number;}

const obj: IObject = { add: (a, b) => { return a + b; }}

**2. implements 实现多个**

interface IFun1 { run(): void;}interface IFun2 { speak(): string;}

class Fun implements IFun1, IFun2 { speak(): string { throw new Error("Method not implemented."); }

run(): void { throw new Error("Method not implemented."); }}

**3. 可索引接口**

可以对数组和对象进行约束

interface IUser { [index: number]: string;}

const array: IUser = ["a", "b"];const object: IUser = { 0: "a", 1: "b"}

**4. 接口类型合并**

用接口去约束类, 相同类，自动合并

interface Speakable { name: string; speak(words: string): void;}

interface Speakable { age: number}

// 接口约束类的实例class Animal implements Speakable { name: string; age: number; constructor(name: string, age: number) { this.name = name; this.age = age; } speak(words: string): void { throw new Error("Method not implemented."); }}

**5. 接口约束构造函数**

一个类，代表一个实例，

// Animal 代表一个类型，代表类的实例class Animal { constructor(public name: string) { this.name = name; }}

interface WithNameClass { new(name: string): Animal;}

function createAnimal(clazz: WithNameClass, name: string) { return new clazz(name);}

const animal = createAnimal(Animal, "dog");console.log(animal.name);

**5. test**

**四、泛型**

**1. 概念**

a、 指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候指定类型的一种特性；

b、泛型 T 作用域只限于函数内部使用。

**2. 简单 泛型 类**

class MyArray<T> { private list: T[] = [];

add(value: T) { this.list.push(value); }

getMax(): T { let result: T = this.list[0]; return result; }}

**3. 泛型接口**

可以泛型接口来约束函数

interface Calculate { <T>(a: T, b: T): T;}

const add: Calculate = function <T>(a: T, b: T): T { return a;}

**4. 多个类型参数**

function swap<A, B>(tuple: [A, B]): [B, A] { return [tuple[1], tuple[0]];}

const a1 = swap<string, number>(["hello", 100]);console.log(a1);

**5. 默认 泛型**

function createArray<T = number>(length: number, value: T): Array<T> { const result: T[] = []; for (let i = 0; i < length; i++) { result[i] = value; } return result;}

createArray<string>(10, "hello world");

**5. 泛型约束**

类，函数只要有类似属性、方法即可

interface LengthWise { length: number;}

class Length implements LengthWise { length: number; constructor(length: number) { this.length = length; }}

function logger<T extends LengthWise>(value: T) { console.log(value.length);}

**5. 泛型接口**

定义接口的时候也可以使用泛型

interface Cart<T> { list: T[];}

interface Product { id: number; name: string;}

const cart: Cart<Product> = { list: [{ id: 1, name: "LiNing" }]}

**5. type 指的是泛型类型的别名**

type CartType<T> = { list: T[] } | T[];const type1: CartType<number> = { list: [1, 2, 3] };const type2: CartType<number> = [1, 2, 3];

**5. test**

**五、兼容性**

**1.1. 接口兼容性**

如果传入的变量和声明的类型不匹配，TS就会进行兼容性检查。

原理： Duck-Check，只要目标类型中声明的属性变量在源类型中都存在就是兼容的。

interface Animal { name: string; age: number;}

interface Person { name: string; age: number; gender: number;}

function getName(animal: Animal): string { return animal.name;}

const person: Person = { name: "人", age: 100, gender: 1}

const name = getName(person);console.log(name);

**1.2. 自动装箱的兼容性**

**String <===> new String()**

let compToString: { toString(): string;}let str2: string = "yaowang";

compToString = str2;compToString = 2;compToString = true;

**3. 类的兼容性**

子类可以给父类，父类不可以给子类，

**类型并不重要，属性才重要**

class Animal { name: string | undefined;}

class Bird extends Animal { swing: number | undefined;}

let animal: Animal;let bird: Bird;animal = bird;

// const bird: Animal = new Bird();

**4. 函数类型兼容**

函数参数只能少或者相等，不能多；

type sumFunc = (a: number, b: number) => number;let sum: sumFunc;

const fn1 = (a: number, b: number) => a + b;sum = fn1;

const fn2 = (a: number) => a;sum = fn2;

const fn3 = () => 1;sum = fn3;

const fn4 = (a: number, b: number, c: number) => a + b + c;// sum = fn4;

**5. 函数参数的双向协变**

当比较函数参数当类型当时候，只有当源函数参数能够赋值给目标函数或者返回来，才能赋值成功。

let sourceFunc = (args: number | string): void => {};let target1Func = (args: number): void => {};let target2Func = (args: number | string | boolean): void => {};// sourceFunc = target1Func; // 错误写法sourceFunc = target2Func;

**6. 泛型兼容性**

泛型在判断兼容性当时候，会先判断具体的类型，然后再进行兼容性判断

// 得到一个对象interface Empty<T> { data: T}

let x!: Empty<string>;let y!: Empty<number | string>;

// 在赋值前使用了变量“y”。// x = y;y = x;

**7. 枚举兼容性**

默认情况下，是 number

enum Colors { Red, Yellow,}

let c1: Colors = Colors.Red;let c2 = 10086;c1 = c2;c2 = c1;

**8. test**

**9. test**

**六、类型保护**

**1. typeof**

function double(x: string | number | boolean) { if (typeof x === "string") { console.log(x); } else if (typeof x === "number") { console.log(x); } else if (typeof x === "boolean") { console.log(x); } else { console.log(x); }}

**2. null 类型保护**

function getFirstLetter1(s: string | null) { function log() { // s?.trim(); // a ? b : c s!.trim(); // 一定存在 }

s = s || ""; return s.charAt(0);}

function getFirstLetter2(s: string | null) { if (s === null) { return s; } return s.charAt(0);}

**3. in 操作符**

interface Bird { swing: number;}interface Fish { skin: number;}function getName(x: Bird | Fish) { if ("swing" in x) { console.log(x); } else if ("skin" in x) { console.log(x); } else { console.log(x); }}

**4. 自定义类型保护**

会在运行时检查类型信息，以确保某个作用域的类型是符号预期的，

function isType1(type: Type1Class| Type2Class): type is Type1Class { return (<Type1Class>type).fun !== "undefined"}

demo 自定义类型保护

interface Bird { legs: number;}

interface Dog { legs: number;}

// 自定义判断类型function isBird(animal: Bird | Dog): animal is Bird { return (<Bird>animal).legs === 2;}

function isDog(animal: Bird | Dog): animal is Bird { return (<Dog>animal).legs === 4;}

function getNumber(animal: Bird | Dog) { if (isBird(animal)) { console.log("该动物是鸟"); } else if (isDog(animal)) { console.log("该动物是狗"); } else { console.log("该动物什么都不是"); } }

let bird: Bird = { legs: 2 }; let dog: Dog = { legs: 4 }; getNumber(bird);

**5. 交叉类型**

两个 interface 合并

interface Bird { name: string; fly(): void;}interface Person { name: string; talk(): void;}

type BirdPerson = Bird & Person;let bp: BirdPerson = { name: "Bird", fly: () => {}, talk: () => {},}

**七、其他**

**1. type & interface 区别**

a、 用interface描述\*\*数据结构\*\*，用type描述\*\*类型关系\*\*

b、type 就是 别名，小明

type NUMBER = number;let a1: NUMBER = 12;

<https://juejin.cn/post/6844903749501059085>

**2. typeof & instanceof & as 区别**

**3. 九大内置类型**

<https://juejin.cn/post/6893071406481801224>

**4. test**

**八、项目 ts 配置升级**

**1. 升级说明**

**九、常见pack错误**

**1. 错误01**



**2. 错误02**