Hello 大家好 我是鲨鱼哥 这次给大家带来的是我曾经非常嫌弃 如今却**爱不释手**的 **TS 技术** 哈哈 大家看往期文章可能已经发现鲨鱼哥之前主要是 Vue 技术栈的 然后因为 Vue2 和 TS 的结合总感觉不是很丝滑 所以我果断就在技术选型的时候去掉了 TS（其实我是觉得用起来很烦 和我之前最讨厌的 eslint 一样 各种**报错**让人不爽）但是 鲨鱼哥今年换了新公司 开启了全新的 **react hook+ts** 这一套组合拳 然后在重新认真学习并在项目里用上了 ts 之后 确实**真香** 哈哈 最直观的感受就是可以帮我们规避很多类型错误 更友好的提示 甚至有些方法我们根据定义的类型大概就知道作用是什么了（去掉了写注释的麻烦）况且如今大火的 **Vue3** 也是 TS 重构的 然后 react 和 ts 的结合就更不必说了 所以还没有开始 ts 的同学就从现在开始跟着鲨鱼哥一起来学习吧 最后欢迎大家点击 链接 加入到鲨鱼哥的前端群 内推 讨论技术 摸鱼 求助 皆可

> 整理不易 如果觉得本文有帮助 记得**点赞三连**哦 十分感谢!

---

1 ts 安装和编译

- 第一步 新建一个空文件夹用来学习 ts
- 第二步 全局安装 ts 和 ts-node

```
cnpm i typescript -g //全局安装ts
cnpm i -g ts-node //全局安装ts-node
```

- 第三步 生成 tsconfig.js 配置文件

```
tsc --init
```

我们就先按照自动生成的 tsconfig 配置项去使用 里面的配置咱们可以先不去管他 后续熟练了再去配置

- 第四步 在项目下新建一个`index.ts` 直接写入

```
const a: string = "hello";
console.log(a);
```

- 第五步 编译 ts 为 js 在控制台（终端）输入命令

```

tsc index.ts
```

神奇的事情发生了 项目下出现了一个同名的 index.js 文件 至此我们已经可以把 ts 文件编译成 js 文件了

不过到这里聪明的小伙伴就会发现了 我们全局安装的 **ts-node** 有什么作用呢 其实这个包是帮助我们在不需要编译成 js 的前提下就可以直接执行 ts 代码 比如 我们在控制台输入

```

ts-node index.ts
```

可以看到我们打印的`hello`已经输出了

那可能 还有的小伙伴会发现 我们每次改动都要手动去执行编译 这样很麻烦 其实我们可以加一个参数来实现每次文件变动 ts 帮我们**自动编译成 js** 的效果

```
tsc --watch index.ts
```

好了 环境安装完毕了 接下来出发去学习 ts 核心吧

2 TS 类型

2.1 布尔类型(boolean)

```
ts
const flag: boolean = true;
```

2.2 Number 类型

```
ts
const flag: number = 1;
```

2.3 String 类型

```
ts
const flag: string = "hello";
```

2.4 Enum 类型

使用枚举我们可以很好的描述一些特定的业务场景，比如一年中的春、夏、秋、冬，还有每周的周一到周天，还有各种颜色，以及可以用它来描述一些状态信息，比如错误码等

```
ts
// 普通枚举 初始值默认为 0 其余的成员会会按顺序自动增长 可以理解为数组下标
enum Color {
  RED,
  PINK,
  BLUE,
}

const pink: Color = Color.PINK;
console.log(pink); // 1

// 设置初始值
enum Color {
  RED = 10,
  PINK,
  BLUE,
}
const pink: Color = Color.PINK;
console.log(pink); // 11

// 字符串枚举 每个都需要声明
enum Color {
```

```
  RED = "红色",
  PINK = "粉色",
  BLUE = "蓝色",
}

const pink: Color = Color.PINK;
console.log(pink); // 粉色

// 常量枚举 它是使用 const 关键字修饰的枚举，常量枚举与普通枚举的区别是，整个枚举会在编译阶段被删除 我们可以看下编译之后的效果

const enum Color {
  RED,
  PINK,
  BLUE,
}

const color: Color[] = [Color.RED, Color.PINK, Color.BLUE];

//编译之后的js如下：
var color = [0 /* RED */, 1 /* PINK */, 2 /* BLUE */];
// 可以看到我们的枚举并没有被编译成js代码 只是把color这个数组变量编译出来了
```

2.5 数组类型(array)

```
ts
const flag1: number[] = [1, 2, 3];
const flag2: Array<number> = [1, 2, 3];
```

2.6 元组类型(tuple)

在 TypeScript 的基础类型中，元组（ Tuple ）表示一个已知**数量**和**类型**的数组 其实可以理解为他是一种特殊的数组

```
ts
const flag: [string, number] = ["hello", 1];
```

2.7 Symbol

我们在使用 Symbol 的时候，必须添加 es6 的编译辅助库 需要在 tsconfig.json 的 `libs` 字段加上 `ES2015`
Symbol 的值是唯一不变的

```
ts
const sym1 = Symbol("hello");
const sym2 = Symbol("hello");
console.log(Symbol("hello") === Symbol("hello"));
```

2.8 任意类型(any)

任何类型都可以被归为 `any` 类型 这让 `any` 类型成为了类型系统的 顶级类型 (也被称作 全局超级类型) TypeScript 允许我们对 `any` 类型的值执行任何操作 而无需事先执行任何形式的检查

一般使用场景： 第三方库没有提供类型文件时可以使用 `any` 类型转换遇到困难或者数据结构太复杂难以定义 不过不要太依赖 `any` 否则就失去了 ts 的意义了

```
ts
const flag: any = document.getElementById("root");
```

2.9 null 和 undefined

```
undefined` 和 `null` 两者有各自的类型分别为 `undefined` 和 `null
ts
let u: undefined = undefined;
let n: null = null;
```

2.10 Unknown 类型

`unknown` 和 `any` 的主要区别是 `unknown` 类型会更加严格 在对 `unknown` 类型的值执行大多数操作之前 我们必须进行某种形式的检查 而在对 `any` 类型的值执行操作之前 我们不必进行任何检查 所有类型都可以被归为 `unknown` 但`unknown`类型只能被赋值给 `any` 类型和 `unknown` 类型本身 而 `any` 啥都能分配和被分配

```
ts
let value: unknown;

value = true; // OK
value = 42; // OK
value = "Hello World"; // OK
value = []; // OK
value = {}; // OK

let value1: unknown = value; // OK
let value2: any = value; // OK
let value3: boolean = value; // Error
let value4: number = value; // Error
let value5: string = value; // Error
let value6: object = value; // Error
```

2.11 void 类型

`void` 表示没有任何类型 当一个函数没有返回值时 TS 会认为它的返回值是 `void` 类型。

```
ts
function hello(name: string): void {}
```

2.12 never 类型

`never` 一般表示用户无法达到的类型 例如`never` 类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型

```ts
ts
function neverReach(): never {
  throw new Error("an error");
}
```

> 思考:never 和 void 的区别 void 可以被赋值为 null 和 undefined 的类型。 never 则是一个不包含值的类型。 拥有 void 返回值类型的函数能正常运行。拥有 never 返回值类型的函数无法正常返回，无法终止，或会抛出异常。

2.13 BigInt 大数类型

使用 `BigInt` 可以安全地存储和操作大整数 我们在使用 `BigInt` 的时候 必须添加 `ESNext` 的编译辅助库 需要在 tsconfig.json 的 `libs` 字段加上`ESNext` 要使用`1n`需要 `"target": "ESNext"` `number` 和 `BigInt` 类型不一样 不兼容

```ts
ts
const max1 = Number.MAX_SAFE_INTEGER; // 2**53-1
console.log(max1 + 1 === max1 + 2); //true

const max2 = BigInt(Number.MAX_SAFE_INTEGER);
console.log(max2 + 1n === max2 + 2n); //false

let foo: number;
let bar: bigint;
foo = bar; //error
bar = foo; //error
```

2.14 object, Object 和 {} 类型

object 类型用于表示非原始类型

```ts
ts
let objectCase: object;
objectCase = 1; // error
objectCase = "a"; // error
objectCase = true; // error
objectCase = null; // error
objectCase = undefined; // error
objectCase = {}; // ok
```

大 Object 代表所有拥有 toString、hasOwnProperty 方法的类型 所以所有原始类型、非原始类型都可以赋给 Object(严格模式下 `null` 和 `undefined` 不可以)

```ts
let ObjectCase: Object;
ObjectCase = 1; // ok
ObjectCase = "a"; // ok
ObjectCase = true; // ok
ObjectCase = null; // error
ObjectCase = undefined; // error
ObjectCase = {}; // ok
```

{} 空对象类型和大 Object 一样 也是表示原始类型和非原始类型的集合

```ts
let simpleCase: {};
simpleCase = 1; // ok
simpleCase = "a"; // ok
simpleCase = true; // ok
simpleCase = null; // error
simpleCase = undefined; // error
simpleCase = {}; // ok
```

2.15 类型推论

指编程语言中能够自动推导出值的类型的能力 它是一些强静态类型语言中出现的特性 定义时未赋值就会推论成 `any` 类型 如果定义的时候就赋值就能利用到类型推论

```ts
let flag; //推断为any
let count = 123; //为number类型
let hello = "hello"; //为string类型
```

2.16 联合类型

联合类型（Union Types）表示取值可以为多种类型中的一种 未赋值时联合类型上只能访问两个类型共有的属性和方法

```ts
let name: string | number;
console.log(name.toString());
name = 1;
console.log(name.toFixed(2));
name = "hello";
console.log(name.length);
```

2.17 类型断言

有时候你会遇到这样的情况，你会比 TypeScript 更了解某个值的详细信息。通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。其实就是你需要手动告诉 ts 就按照你断言的那个类型通过编译（这一招很关键 有时候可以帮助你解决很多编译报错）

类型断言有两种形式：

```ts
// 尖括号 语法
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;

// as 语法
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

> 以上两种方式虽然没有任何区别，但是尖括号格式会与 react 中 JSX 产生语法冲突，因此我们更推荐使用 as 语法。

非空断言 在上下文中当类型检查器无法断定类型时 一个新的后缀表达式操作符 ! 可以用于断言操作对象是非 `null` 和非 `undefined` 类型

```ts
let flag: null | undefined | string;
flag!.toString(); // ok
flag.toString(); // error
```

2.18 字面量类型

在 TypeScript 中，字面量不仅可以表示值，还可以表示类型，即所谓的字面量类型。 目前，TypeScript 支持 3 种字面量类型：字符串字面量类型、数字字面量类型、布尔字面量类型，对应的字符串字面量、数字字面量、布尔字面量分别拥有与其值一样的字面量类型，具体示例如下：

```ts
let flag1: "hello" = "hello";
let flag2: 1 = 1;
let flag3: true = true;
```

2.19 类型别名

类型别名用来给一个类型起个新名字

```ts
type flag = string | number;

function hello(value: flag) {}
```

2.20 交叉类型

交叉类型是将多个类型合并为一个类型。通过 & 运算符可以将现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性

```ts
type Flag1 = { x: number };
type Flag2 = Flag1 & { y: string };

let flag3: Flag2 = {
  x: 1,
  y: "hello",
  henb,
};
```

2.21 类型保护

类型保护就是一些表达式，他们在编译的时候就能通过类型信息确保某个作用域内变量的类型 其主要思想是尝试检测属性、方法或原型，以确定如何处理值

typeof 类型保护

```ts
function double(input: string | number | boolean) {
  if (typeof input === "string") {
    return input + input;
  } else {
    if (typeof input === "number") {
      return input * 2;
    } else {
      return !input;
    }
  }
}
```

in 关键字

```ts
interface Bird {
  fly: number;
}

interface Dog {
  leg: number;
}

function getNumber(value: Bird | Dog) {
  if ("fly" in value) {
    return value.fly;
  }
  return value.leg;
}
```

instanceof 类型保护

```ts
class Animal {
  name!: string;
}
class Bird extends Animal {
  fly!: number;
}
function getName(animal: Animal) {
  if (animal instanceof Bird) {
    console.log(animal.fly);
  } else {
    console.log(animal.name);
  }
}
```

自定义类型保护

通过 `type is xxx` 这样的类型谓词来进行类型保护

例如下面的例子 `value is object` 就会认为如果函数返回 true 那么定义的 value 就是 object 类型

```ts
function isObject(value: unknown): value is object {
  return typeof value === "object" && value !== null;
}

function fn(x: string | object) {
  if (isObject(x)) {
    // ....
  } else {
    // .....
  }
}
```

3 函数

3.1 函数的定义

可以指定参数的类型和返回值的类型

```ts
function hello(name: string): void {
  console.log("hello", name);
}
hello("hahaha");
```

3.2 函数表达式

定义函数类型

```ts
type SumFunc = (x: number, y: number) => number;

let countNumber: SumFunc = function (a, b) {
  return a + b;
};
```

3.3 可选参数

在 TS 中函数的形参和实参必须一样，不一样就要配置可选参数,而且必须是**最后一个参数**

```ts
function print(name: string, age?: number): void {
  console.log(name, age);
}
print("hahaha");
```

3.4 默认参数

```ts
function ajax(url: string, method: string = "GET") {
  console.log(url, method);
}
ajax("/users");
```

3.5 剩余参数

```ts
function sum(...numbers: number[]) {
  return numbers.reduce((val, item) => (val += item), 0);
}
console.log(sum(1, 2, 3));
```

3.6 函数重载

函数重载或方法重载是使用相同名称和不同参数数量或类型创建多个方法的一种能力。 在 TypeScript 中，表现为给同一个函数提供多个函数类型定义

```ts
let obj: any = {};
function attr(val: string): void;
function attr(val: number): void;
function attr(val: any): void {
  if (typeof val === "string") {
    obj.name = val;
  } else {
```

```
    obj.age = val;
  }
}
attr("hahaha");
attr(9);
attr(true);
console.log(obj);
```

> 注意：函数重载真正执行的是同名函数最后定义的函数体 在最后一个函数体定义之前全都属于函数类型定义 不能写具体的函数实现方法 只能定义类型

4 类

4.1 类的定义

在 TypeScript 中，我们可以通过 `Class` 关键字来定义一个类

```
ts
class Person {
  name!: string; //如果初始属性没赋值就需要加上!
  constructor(_name: string) {
    this.name = _name;
  }
  getName(): void {
    console.log(this.name);
  }
}
let p1 = new Person("hello");
p1.getName();
```

当然 如果我们图省事 我们也可以把属性定义直接写到构造函数的参数里面去(不过一般不建议这样写 因为会让代码增加阅读难度)

```
ts
class Person {
  constructor(public name: string) {}
  getName(): void {
    console.log(this.name);
  }
}
let p1 = new Person("hello");
p1.getName();
```

> 注意：当我们定义一个类的时候,会得到 **2 个类型** 一个是构造函数类型的函数类型(当做普通构造函数的类型) 另一个是类的实例类型（代表实例）

具体看例子

```ts
class Component {
  static myName: string = "静态名称属性";
  myName: string = "实例名称属性";
}
//ts 一个类型 一个叫值
//放在=后面的是值
let com = Component; //这里是代表构造函数
//冒号后面的是类型
let c: Component = new Component(); //这里是代表实例类型
let f: typeof Component = com;
```

4.2 存取器

在 TypeScript 中，我们可以通过存取器来改变一个类中属性的读取和赋值行为

```ts
class User {
  myname: string;
  constructor(myname: string) {
    this.myname = myname;
  }
  get name() {
    return this.myname;
  }
  set name(value) {
    this.myname = value;
  }
}

let user = new User("hello");
user.name = "world";
console.log(user.name);
```

其实我们可以看看翻译成 es5 的代码 原理很简单 就是使用了 Object.defineProperty 在类的原型上面拦截了属性对应的 get 和 set 方法

```js
var User = /** @class */ (function () {
  function User(myname) {
    this.myname = myname;
  }
  Object.defineProperty(User.prototype, "name", {
    get: function () {
      return this.myname;
    },
    set: function (value) {
      this.myname = value;
    },
    enumerable: false,
```

```
    configurable: true,
  });
  return User;
})();
var user = new User("hello");
user.name = "world";
console.log(user.name);
```

4.3 readonly 只读属性

readonly 修饰的变量只能在**构造函数**中初始化 TypeScript 的类型系统同样也允许将 interface、type、 class 上的属性标识为 readonly readonly 实际上只是在编译阶段进行代码检查。

```
ts
class Animal {
  public readonly name: string;
  constructor(name: string) {
    this.name = name;
  }
  changeName(name: string) {
    this.name = name; //这个ts是报错的
  }
}

let a = new Animal("hello");
```

4.4 继承

子类继承父类后子类的实例就拥有了父类中的属性和方法，可以增强代码的可复用性

将子类公用的方法抽象出来放在父类中，自己的特殊逻辑放在子类中重写父类的逻辑

super 可以调用父类上的方法和属性

在 TypeScript 中，我们可以通过 extends 关键字来实现继承

```
ts
class Person {
  name: string; //定义实例的属性，默认省略public修饰符
  age: number;
  constructor(name: string, age: number) {
    //构造函数
    this.name = name;
    this.age = age;
  }
  getName(): string {
    return this.name;
  }
  setName(name: string): void {
    this.name = name;
  }
}
```

```
class Student extends Person {
  no: number;
  constructor(name: string, age: number, no: number) {
    super(name, age);
    this.no = no;
  }
  getNo(): number {
    return this.no;
  }
}
let s1 = new Student("hello", 10, 1);
console.log(s1);
```

4.5 类里面的修饰符

public 类里面 子类 其它任何地方外边都可以访问 **protected** 类里面 子类 都可以访问,其它任何地方不能访问 **private** 类里面可以访问，子类和其它任何地方都不可以访问

```
ts
class Parent {
  public name: string;
  protected age: number;
  private car: number;
  constructor(name: string, age: number, car: number) {
    //构造函数
    this.name = name;
    this.age = age;
    this.car = car;
  }
  getName(): string {
    return this.name;
  }
  setName(name: string): void {
    this.name = name;
  }
}
class Child extends Parent {
  constructor(name: string, age: number, car: number) {
    super(name, age, car);
  }
  desc() {
    console.log(`${this.name} ${this.age} ${this.car}`); //car访问不到 会报错
  }
}

let child = new Child("hello", 10, 1000);
console.log(child.name);
console.log(child.age); //age访问不到 会报错
console.log(child.car); //car访问不到 会报错
```

4.6 静态属性 静态方法

类的静态属性和方法是直接定义在类本身上面的 所以也只能通过直接调用类的方法和属性来访问

```
ts
class Parent {
  static mainName = "Parent";
  static getmainName() {
    console.log(this); //注意静态方法里面的this指向的是类本身 而不是类的实例对象 所以静态方法里面只能访问类的静态属性和方法
    return this.mainName;
  }
  public name: string;
  constructor(name: string) {
    //构造函数
    this.name = name;
  }
}
console.log(Parent.mainName);
console.log(Parent.getmainName());
```

4.7 抽象类和抽象方法

抽象类，无法被实例化，只能被继承并且无法创建抽象类的实例 子类可以对抽象类进行不同的实现

抽象方法只能出现在抽象类中并且抽象方法不能在抽象类中被具体实现，只能在抽象类的子类中实现（必须要实现）

使用场景： 我们一般用抽象类和抽象方法抽离出事物的共性 以后所有继承的子类必须按照规范去实现自己的具体逻辑 这样可以增加代码的可维护性和复用性

使用 `abstract` 关键字来定义抽象类和抽象方法

```
ts
abstract class Animal {
  name!: string;
  abstract speak(): void;
}
class Cat extends Animal {
  speak() {
    console.log("喵喵喵");
  }
}
let animal = new Animal(); //直接报错 无法创建抽象类的实例
let cat = new Cat();
cat.speak();
```

> 思考 1:重写(override)和重载(overload)的区别

重写是指子类重写继承自父类中的方法 **重载**是指为同一个函数提供多个类型定义

```
ts
```

```
class Animal {
  speak(word: string): string {
    return "动物:" + word;
  }
}
class Cat extends Animal {
  speak(word: string): string {
    return "猫:" + word;
  }
}
let cat = new Cat();
console.log(cat.speak("hello"));
// 上面是重写
//-----------------------------------------
// 下面是重载
function double(val: number): number;
function double(val: string): string;
function double(val: any): any {
  if (typeof val == "number") {
    return val * 2;
  }
  return val + val;
}

let r = double(1);
console.log(r);
```

> 思考 2:什么是**多态**

在父类中定义一个方法，在子类中有多个实现，在程序运行的时候，根据不同的对象执行不同的操作，实现运行时的绑定。

```
ts
abstract class Animal {
  // 声明抽象的方法，让子类去实现
  abstract sleep(): void;
}
class Dog extends Animal {
  sleep() {
    console.log("dog sleep");
  }
}
let dog = new Dog();
class Cat extends Animal {
  sleep() {
    console.log("cat sleep");
  }
}
let cat = new Cat();
let animals: Animal[] = [dog, cat];
animals.forEach((i) => {
```

```
  i.sleep();
});
```

5 接口

接口既可以在面向对象编程中表示为行为的抽象，也可以用来描述对象的形状

我们用 `interface` 关键字来定义接口 在接口中可以用分号或者逗号分割每一项，也可以什么都不加

5.1 对象的形状

```
ts
//接口可以用来描述`对象的形状`
//接口可以用来描述`对象的形状`
interface Speakable {
  speak(): void;
  readonly lng: string; //readonly表示只读属性 后续不可以更改
  name?: string; //? 表示可选属性
}

let speakman: Speakable = {
  //   speak() {}, //少属性会报错
  name: "hello",
  lng: "en",
  age: 111, //多属性也会报错
};
```

5.2 行为的抽象

接口可以把一些类中共有的属性和方法抽象出来,可以用来约束实现此接口的类

一个类可以实现多个接口，一个接口也可以被多个类实现

我们用 `implements` 关键字来代表 实现

```
ts
//接口可以在面向对象编程中表示为行为的抽象
interface Speakable {
  speak(): void;
}
interface Eatable {
  eat(): void;
}
//一个类可以实现多个接口
class Person implements Speakable, Eatable {
  speak() {
    console.log("Person说话");
  }
  //   eat() {} //需要实现的接口包含eat方法 不实现会报错
}
```

5.3 定义任意属性

如果我们在定义接口的时候无法预先知道有哪些属性的时候,可以使用 `[propName:string]:any`,propName 名字是任意的

```
ts
interface Person {
  id: number;
  name: string;
  [propName: string]: any;
}

let p1 = {
  id: 1,
  name: "hello",
  age: 10,
};
```

这个接口表示 必须要有 id 和 name 这两个字段 然后还可以新加其余的未知字段

5.4 接口的继承

我们除了类可以继承 接口也可以继承 同样的使用 `extends` 关键字

```
ts
interface Speakable {
  speak(): void;
}
interface SpeakChinese extends Speakable {
  speakChinese(): void;
}
class Person implements SpeakChinese {
  speak() {
    console.log("Person");
  }
  speakChinese() {
    console.log("speakChinese");
  }
}
```

5.5 函数类型接口

可以用接口来定义函数类型

```
ts
interface discount {
  (price: number): number;
}
let cost: discount = function (price: number): number {
  return price * 0.8;
};
```

5.6 构造函数的类型接口

使用特殊的 new()关键字来描述类的构造函数类型

```
ts
class Animal {
  constructor(public name: string) {}
}
//不加new是修饰函数的,加new是修饰类的
interface WithNameClass {
  new (name: string): Animal;
}
function createAnimal(clazz: WithNameClass, name: string) {
  return new clazz(name);
}
let a = createAnimal(Animal, "hello");
console.log(a.name);
```

其实这样的用法一般出现在 当我们需要把一个类作为参数的时候 我们需要对传入的类的构造函数类型进行约束 所以需要使用 new 关键字代表是类的构造函数类型 用以和普通函数进行区分

> 思考：接口和类型别名的区别 这个题目是经典的 **ts 面试题**

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。

1.基础数据类型 与接口不同，类型别名还可以用于其他类型，如基本类型（原始值）、联合类型、元组

```
ts
// primitive
type Name = string;

// union
type PartialPoint = PartialPointX | PartialPointY;

// tuple
type Data = [number, string];

// dom
let div = document.createElement("div");
type B = typeof div;
```

2.重复定义

接口可以定义多次 会被自动合并为单个接口 类型别名不可以重复定义

```ts
ts
interface Point {
  x: number;
}
interface Point {
  y: number;
}
const point: Point = { x: 1, y: 2 };
```

3.扩展 接口可以扩展类型别名，同理，类型别名也可以扩展接口。但是两者实现扩展的方式不同

接口的扩展就是继承，通过 extends 来实现。类型别名的扩展就是交叉类型，通过 & 来实现。

```ts
ts
// 接口扩展接口
interface PointX {
  x: number;
}

interface Point extends PointX {
  y: number;
}
// ----
// 类型别名扩展类型别名
type PointX = {
  x: number;
};

type Point = PointX & {
  y: number;
};
// ----
// 接口扩展类型别名
type PointX = {
  x: number;
};
interface Point extends PointX {
  y: number;
}
// ----
// 类型别名扩展接口
interface PointX {
  x: number;
}
type Point = PointX & {
  y: number;
};
```

4.实现 这里有一个特殊情况 类无法实现定义了联合类型的类型别名

```ts
type PartialPoint = { x: number } | { y: number };

// A class can only implement an object type or
// intersection of object types with statically known members.
class SomePartialPoint implements PartialPoint {
  // Error
  x = 1;
  y = 2;
}
```

6 泛型

泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性

为了更好的了解泛型的作用 我们可以看下面的一个例子

```
function createArray(length: number, value: any): any[] {
  let result = [];
  for (let i = 0; i < length; i++) {
    result[i] = value;
  }
  return result;
}

createArray(3, "x"); // ['x', 'x', 'x']
```

上述这段代码用来生成一个长度为 length 值为 value 的数组 但是我们其实可以发现一个问题 不管我们传入什么类型的 value 返回值的数组永远是 any 类型 如果我们想要的效果是 我们预先不知道会传入什么类型 但是我们希望不管我们传入什么类型 我们的返回的数组的指里面的类型应该和参数保持一致 那么这时候 泛型就登场了

使用**泛型**改造

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

createArray<string>(3, "x"); // ['x', 'x', 'x']
```

我们可以使用<>的写法 然后再面传入一个变量 T 用来表示后续函数需要用到的类型 当我们真正去调用函数的时候再传入 T 的类型就可以解决很多预先无法确定类型相关的问题

6.1 多个类型参数

如果我们需要有多个未知的类型占位 那么我们可以定义任何的字母来表示不同的类型参数

```
function swap<T, U>(tuple: [T, U]): [U, T] {
  return [tuple[1], tuple[0]];
}

swap([7, "seven"]); // ['seven', 7]
```

6.2 泛型约束

在函数内部使用泛型变量的时候，由于事先不知道它是哪种类型，所以不能随意的操作它的属性或方法：

```
function loggingIdentity<T>(arg: T): T {
  console.log(arg.length);
  return arg;
}

// index.ts(2,19): error TS2339: Property 'length' does not exist on type 'T'.
```

上例中，泛型 T 不一定包含属性 length，所以编译的时候报错了。

这时，我们可以对泛型进行约束，只允许这个函数传入那些包含 length 属性的变量。这就是**泛型约束**

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
```

> 注意：我们在泛型里面使用`extends`关键字代表的是泛型约束 需要和类的继承区分开

6.3 泛型接口

定义接口的时候也可以指定泛型

```
interface Cart<T> {
  list: T[];
}
let cart: Cart<{ name: string; price: number }> = {
  list: [{ name: "hello", price: 10 }],
};
console.log(cart.list[0].name, cart.list[0].price);
```

我们定义了接口传入的类型 T 之后返回的对象数组里面 T 就是当时传入的参数类型

6.4 泛型类

```
class MyArray<T> {
  private list: T[] = [];
  add(value: T) {
    this.list.push(value);
  }
  getMax(): T {
    let result = this.list[0];
    for (let i = 0; i < this.list.length; i++) {
      if (this.list[i] > result) {
        result = this.list[i];
      }
    }
    return result;
  }
}
let arr = new MyArray();
arr.add(1);
arr.add(2);
arr.add(3);
let ret = arr.getMax();
console.log(ret);
```

上诉例子我们实现了一个在数组里面添加数字并且获取最大值的泛型类

6.5 泛型类型别名

```
ts
type Cart<T> = { list: T[] } | T[];
let c1: Cart<string> = { list: ["1"] };
let c2: Cart<number> = [1];
```

6.6 泛型参数的默认类型

我们可以为泛型中的类型参数指定默认类型。当使用泛型时没有在代码中直接指定类型参数，从实际值参数中也无法推测出时，这个默认类型就会起作用

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

7 实用技巧

7.1 typeof 关键词

`typeof` 关键词除了做类型保护 还可以从实现推出类型，

```
ts
//先定义变量，再定义类型
let p1 = {
  name: "hello",
  age: 10,
  gender: "male",
};
type People = typeof p1;
function getName(p: People): string {
  return p.name;
}
getName(p1);
```

上面的例子就是使用 typeof 获取一个变量的类型

7.2 keyof 关键词

`keyof` 可以用来取得一个对象接口的所有 key 值

```
ts
interface Person {
  name: string;
  age: number;
  gender: "male" | "female";
}
//type PersonKey = 'name'|'age'|'gender';
type PersonKey = keyof Person;

function getValueByKey(p: Person, key: PersonKey) {
  return p[key];
}
let val = getValueByKey({ name: "hello", age: 10, gender: "male" }, "name");
console.log(val);
```

7.3 索引访问操作符

使用 [] 操作符可以进行索引访问

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

type x = Person["name"]; // x is string
```

7.4 映射类型 in

在定义的时候用 in 操作符去批量定义类型中的属性

```ts
interface Person {
  name: string;
  age: number;
  gender: "male" | "female";
}
//批量把一个接口中的属性都变成可选的
type PartPerson = {
  [Key in keyof Person]?: Person[Key];
};

let p1: PartPerson = {};
```

7.5 infer 关键字

在条件类型语句中，可以用 infer 声明一个类型变量并且对它进行使用。

```ts
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
```

以上代码中 infer R 就是声明一个变量来承载传入函数签名的返回值类型，简单说就是用它取到函数返回值的类型方便之后使用。

7.6 内置工具类型

1. Exclude<T,U> 从 T 可分配给的类型中排除 U

```ts
type Exclude<T, U> = T extends U ? never : T;

type E = Exclude<string | number, string>;
let e: E = 10;
```

1. Extract<T,U> 从 T 可分配给的类型中提取 U

```ts
type Extract<T, U> = T extends U ? T : never;

type E = Extract<string | number, string>;
let e: E = "1";
```

1. NonNullable 从 T 中排除 `null` 和 `undefined`

```ts
type NonNullable<T> = T extends null | undefined ? never : T;

type E = NonNullable<string | number | null | undefined>;
let e: E = null;
```

1. ReturnType `infer` 最早出现在此 PR 中，表示在 `extends` 条件语句中待推断的类型变量

```ts
type ReturnType<T extends (...args: any[]) => any> = T extends (
  ...args: any[]
) => infer R
  ? R
  : any;
function getUserInfo() {
  return { name: "hello", age: 10 };
}

// 通过 ReturnType 将 getUserInfo 的返回值类型赋给了 UserInfo
type UserInfo = ReturnType<typeof getUserInfo>;

const userA: UserInfo = {
  name: "hello",
  age: 10,
};
```

可见 该工具类型主要是获取函数类型的返回类型

1. Parameters 该工具类型主要是获取函数类型的参数类型

```ts
type Parameters<T> = T extends (...args: infer R) => any ? R : any;

type T0 = Parameters<() => string>; // []
type T1 = Parameters<(s: string) => void>; // [string]
type T2 = Parameters<<T>(arg: T) => T>; // [unknown]
```

1. Partial Partial 可以将传入的属性由非可选变为可选

```ts
type Partial<T> = { [P in keyof T]?: T[P] };
interface A {
  a1: string;
  a2: number;
  a3: boolean;
}
type aPartial = Partial<A>;
const a: aPartial = {}; // 不会报错
```

1. Required Required 可以将传入的属性中的可选项变为必选项，这里用了 -? 修饰符来实现。

```ts
interface Person {
  name: string;
  age: number;
  gender?: "male" | "female";
}
/**
 * type Required<T> = { [P in keyof T]-?: T[P] };
 */
let p: Required<Person> = {
  name: "hello",
  age: 10,
  gender: "male",
};
```

1. Readonly Readonly 通过为传入的属性每一项都加上 readonly 修饰符来实现。

```ts
interface Person {
  name: string;
  age: number;
  gender?: "male" | "female";
}
//type Readonly<T> = { readonly [P in keyof T]: T[P] };
let p: Readonly<Person> = {
  name: "hello",
  age: 10,
  gender: "male",
};
p.age = 11; //error
```

1. Pick<T,K> Pick 能够帮助我们从传入的属性中摘取某些返回

```ts
interface Todo {
  title: string;
  description: string;
```

```
  done: boolean;
}
/**
 * From T pick a set of properties K
 * type Pick<T, K extends keyof T> = { [P in K]: T[P] };
 */
type TodoBase = Pick<Todo, "title" | "done">;

// =
type TodoBase = {
  title: string;
  done: boolean;
};
```

1. Record<K,T> 构造一个类型，该类型具有一组属性 K，每个属性的类型为 T。可用于将一个类型的属性映射为另一个类型。Record 后面的泛型就是对象键和值的类型。

简单理解：K 对应对应的 key，T 对应对象的 value，返回的就是一个声明好的对象 但是 K 对应的泛型约束是 `keyof any` 也就意味着只能传入 `string|number|symbol`

```
ts
// type Record<K extends keyof any, T> = {
// [P in K]: T;
// };
type Point = "x" | "y";
type PointList = Record<Point, { value: number }>;
const cars: PointList = {
  x: { value: 10 },
  y: { value: 20 },
};
```

1. Omit<K,T> 基于已经声明的类型进行属性剔除获得新类型

```
ts
// type Omit=Pick<T,Exclude<keyof T,K>>
type User = {
id: string;
name: string;
email: string;
};
type UserWithoutEmail = Omit<User, "email">;// UserWithoutEmail ={id: string;name:
string;}
};
```

8 TypeScript 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、方法装饰器和参数装饰器

装饰器的写法分为普通装饰器和装饰器工厂

使用@装饰器的写法需要把 tsconfig.json 的 `experimentalDecorators` 字段设置为 true

8.1 类装饰器

类装饰器在类声明之前声明，用来监视、修改或替换类定义

```
ts
namespace a {
  //当装饰器作为修饰类的时候，会把构造器传递进去
  function addNameEat(constructor: Function) {
    constructor.prototype.name = "hello";
    constructor.prototype.eat = function () {
      console.log("eat");
    };
  }
  @addNameEat
  class Person {
    name!: string;
    eat!: Function;
    constructor() {}
  }
  let p: Person = new Person();
  console.log(p.name);
  p.eat();
}

namespace b {
  //还可以使用装饰器工厂 这样可以传递额外参数
  function addNameEatFactory(name: string) {
    return function (constructor: Function) {
      constructor.prototype.name = name;
      constructor.prototype.eat = function () {
        console.log("eat");
      };
    };
  }
  @addNameEatFactory("hello")
  class Person {
    name!: string;
    eat!: Function;
    constructor() {}
  }
  let p: Person = new Person();
  console.log(p.name);
```

```
  p.eat();
}

namespace c {
  //还可以替换类,不过替换的类要与原类结构相同
  function enhancer(constructor: Function) {
    return class {
      name: string = "jiagou";
      eat() {
        console.log("吃饭饭");
      }
    };
  }
  @enhancer
  class Person {
    name!: string;
    eat!: Function;
    constructor() {}
  }
  let p: Person = new Person();
  console.log(p.name);
  p.eat();
}
```

8.2 属性装饰器

属性装饰器表达式会在运行时当作函数被调用，传入 2 个参数 第一个参数对于静态成员来说是类的构造函数，对于实例成员是类的原型对象 第二个参数是属性的名称

```
ts
//修饰实例属性
function upperCase(target: any, propertyKey: string) {
  let value = target[propertyKey];
  const getter = function () {
    return value;
  };
  // 用来替换的setter
  const setter = function (newVal: string) {
    value = newVal.toUpperCase();
  };
  // 替换属性，先删除原先的属性，再重新定义属性
  if (delete target[propertyKey]) {
    Object.defineProperty(target, propertyKey, {
      get: getter,
      set: setter,
      enumerable: true,
      configurable: true,
    });
  }
}
```

```
class Person {
  @upperCase
  name!: string;
}
let p: Person = new Person();
p.name = "world";
console.log(p.name);
```

8.3 方法装饰器

方法装饰器顾名思义，用来装饰类的方法。它接收三个参数： target: Object - 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象 propertyKey: string | symbol - 方法名 descriptor: TypePropertyDescript - 属性描述符

```
ts
//修饰实例方法
function noEnumerable(
  target: any,
  property: string,
  descriptor: PropertyDescriptor
) {
  console.log("target.getName", target.getName);
  console.log("target.getAge", target.getAge);
  descriptor.enumerable = false;
}
//重写方法
function toNumber(
  target: any,
  methodName: string,
  descriptor: PropertyDescriptor
) {
  let oldMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    args = args.map((item) => parseFloat(item));
    return oldMethod.apply(this, args);
  };
}
class Person {
  name: string = "hello";
  public static age: number = 10;
  constructor() {}
  @noEnumerable
  getName() {
    console.log(this.name);
  }
  @toNumber
  sum(...args: any[]) {
    return args.reduce((accu: number, item: number) => accu + item, 0);
  }
}
let p: Person = new Person();
```

```
for (let attr in p) {
  console.log("attr=", attr);
}
p.getName();
console.log(p.sum("1", "2", "3"));
```

8.4 参数装饰器

参数装饰器顾名思义，是用来装饰函数参数，它接收三个参数：

target: Object - 被装饰的类 propertyKey: string | symbol - 方法名 parameterIndex: number - 方法中参数的索引值

```
ts
function Log(target: Function, key: string, parameterIndex: number) {
  let functionLogged = key || target.prototype.constructor.name;
  console.log(`The parameter in position ${parameterIndex} at ${functionLogged} has
  been decorated`);
}

class Greeter {
  greeting: string;
  constructor(@Log phrase: string) {
    this.greeting = phrase;
  }
}
```

以上代码成功运行后，控制台会输出以下结果：`"The parameter in position 0 at Greeter has been decorated"`

8.5 装饰器执行顺序

有多个参数装饰器时：从最后一个参数依次向前执行

方法和方法参数中参数装饰器先执行。 方法和属性装饰器，谁在前面谁先执行。因为参数属于方法一部分，所以参数会一直紧紧挨着方法执行

类装饰器总是最后执行

```
ts
function Class1Decorator() {
  return function (target: any) {
    console.log("类1装饰器");
  };
}
function Class2Decorator() {
  return function (target: any) {
    console.log("类2装饰器");
  };
}
function MethodDecorator() {
  return function (
```

```
    target: any,
    methodName: string,
    descriptor: PropertyDescriptor
  ) {
    console.log("方法装饰器");
  };
}
function Param1Decorator() {
  return function (target: any, methodName: string, paramIndex: number) {
    console.log("参数1装饰器");
  };
}
function Param2Decorator() {
  return function (target: any, methodName: string, paramIndex: number) {
    console.log("参数2装饰器");
  };
}
function PropertyDecorator(name: string) {
  return function (target: any, propertyName: string) {
    console.log(name + "属性装饰器");
  };
}

@Class1Decorator()
@Class2Decorator()
class Person {
  @PropertyDecorator("name")
  name: string = "hello";
  @PropertyDecorator("age")
  age: number = 10;
  @MethodDecorator()
  greet(@Param1Decorator() p1: string, @Param2Decorator() p2: string) {}
}

/**
name属性装饰器
age属性装饰器
参数2装饰器
参数1装饰器
方法装饰器
类2装饰器
类1装饰器
 */
```

9 编译

9.1 tsconfig.json 的作用

- 用于标识 TypeScript 项目的根路径；
- 用于配置 TypeScript 编译器；
- 用于指定编译的文件。

9.2 tsconfig.json 重要字段

- files - 设置要编译的文件的名称；
- include - 设置需要进行编译的文件，支持路径模式匹配；
- exclude - 设置无需进行编译的文件，支持路径模式匹配；
- compilerOptions - 设置与编译流程相关的选项。

9.3 compilerOptions 选项

```
ts
{
  "compilerOptions": {

    /* 基本选项 */
    "target": "es5",                       // 指定 ECMAScript 目标版本: 'ES3' (default),
'ES5', 'ES6'/'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'
    "module": "commonjs",                  // 指定使用模块: 'commonjs', 'amd', 'system',
'umd' or 'es2015'
    "lib": [],                             // 指定要包含在编译中的库文件
    "allowJs": true,                       // 允许编译 javascript 文件
    "checkJs": true,                       // 报告 javascript 文件中的错误
    "jsx": "preserve",                     // 指定 jsx 代码的生成: 'preserve', 'react-
native', or 'react'
    "declaration": true,                   // 生成相应的 '.d.ts' 文件
    "sourceMap": true,                     // 生成相应的 '.map' 文件
    "outFile": "./",                       // 将输出文件合并为一个文件
    "outDir": "./",                        // 指定输出目录
    "rootDir": "./",                       // 用来控制输出目录结构 --outDir.
    "removeComments": true,                // 删除编译后的所有的注释
    "noEmit": true,                        // 不生成输出文件
    "importHelpers": true,                 // 从 tslib 导入辅助工具函数
    "isolatedModules": true,               // 将每个文件做为单独的模块 （与
'ts.transpileModule' 类似）.

    /* 严格的类型检查选项 */
    "strict": true,                        // 启用所有严格类型检查选项
    "noImplicitAny": true,                 // 在表达式和声明上有隐含的 any类型时报错
    "strictNullChecks": true,              // 启用严格的 null 检查
    "noImplicitThis": true,                // 当 this 表达式值为 any 类型的时候，生成一个错误
    "alwaysStrict": true,                  // 以严格模式检查每个模块，并在每个文件里加入 'use
strict'

    /* 额外的检查 */
    "noUnusedLocals": true,                // 有未使用的变量时，抛出错误
    "noUnusedParameters": true,            // 有未使用的参数时，抛出错误
    "noImplicitReturns": true,             // 并不是所有函数里的代码都有返回值时，抛出错误
```

```
    "noFallthroughCasesInSwitch": true,     // 报告 switch 语句的 fallthrough 错误。（即，不允
许 switch 的 case 语句贯穿）

    /* 模块解析选项 */
    "moduleResolution": "node",             // 选择模块解析策略： 'node' (Node.js) or
'classic' (TypeScript pre-1.6)
    "baseUrl": "./",                        // 用于解析非相对模块名称的基目录
    "paths": {},                            // 模块名到基于 baseUrl 的路径映射的列表
    "rootDirs": [],                         // 根文件夹列表，其组合内容表示项目运行时的结构内容
    "typeRoots": [],                        // 包含类型声明的文件列表
    "types": [],                            // 需要包含的类型声明文件名列表
    "allowSyntheticDefaultImports": true,   // 允许从没有设置默认导出的模块中默认导入。

    /* Source Map Options */
    "sourceRoot": "./",                     // 指定调试器应该找到 TypeScript 文件而不是源文件的
位置
    "mapRoot": "./",                        // 指定调试器应该找到映射文件而不是生成文件的位置
    "inlineSourceMap": true,                // 生成单个 soucemaps 文件，而不是将 sourcemaps 生
成不同的文件
    "inlineSources": true,                  // 将代码与 sourcemaps 生成到一个文件中，要求同时设
置了 --inlineSourceMap 或 --sourceMap 属性

    /* 其他选项 */
    "experimentalDecorators": true,         // 启用装饰器
    "emitDecoratorMetadata": true           // 为装饰器提供元数据的支持
  }
}
```

10 模块和声明文件

10.1 全局模块

在默认情况下，当你开始在一个新的 TypeScript 文件中写下代码时，它处于全局命名空间中

使用全局变量空间是危险的，因为它会与文件内的代码命名冲突。我们推荐使用下文中将要提到的文件模块

foo.ts

```ts
const foo = 123;
```

bar.ts

```ts
const bar = foo; // allowed
```

10.2 文件模块

- 文件模块也被称为外部模块。如果在你的 TypeScript 文件的根级别位置含有 import 或者 export，那么它会在这个文件中创建一个本地的作用域
- 模块是 TS 中外部模块的简称，侧重于代码和复用
- 模块在其自身的作用域里执行，而不是在全局作用域里
- 一个模块里的变量、函数、类等在外部是不可见的，除非你把它导出
- 如果想要使用一个模块里导出的变量，则需要导入

foo.ts

```
ts
const foo = 123;
export {};
```

bar.ts

```
ts
const bar = foo; // error
```

10.3 声明文件

- 我们可以把类型声明放在一个单独的类型声明文件中
- 文件命名规范为*.d.ts
- 查看类型声明文件有助于了解库的使用方式

typings\jquery.d.ts

```
ts
declare const $: (selector: string) => {
  click(): void;
  width(length: number): void;
};
```

10.4 第三方声明文件

- 可以安装使用第三方的声明文件
- @types 是一个约定的前缀，所有的第三方声明的类型库都会带有这样的前缀
- JavaScript 中有很多内置对象，它们可以在 TypeScript 中被当做声明好了的类型
- 内置对象是指根据标准在全局作用域（Global）上存在的对象。这里的标准是指 ECMAScript 和其他环境（比如 DOM）的标准
- 这些内置对象的类型声明文件，就包含在 TypeScript 核心库的类型声明文件中,具体可以查看ts 核心声明文件

10.5 查找声明文件

- 如果是手动写的声明文件，那么需要满足以下条件之一，才能被正确的识别
- 给 package.json 中的 types 或 typings 字段指定一个类型声明文件地址
- 在项目根目录下，编写一个 index.d.ts 文件
- 针对入口文件（package.json 中的 main 字段指定的入口文件），编写一个同名不同后缀的 .d.ts 文件

```ts
{
    "name": "myLib",
    "version": "1.0.0",
    "main": "lib/index.js",
    "types": "myLib.d.ts",
}
```

查找过程如下：

1.先找 myLib.d.ts

2.没有就再找 index.d.ts

3.还没有再找 lib/index.d.js

4.还找不到就认为没有类型声明了

小结

能看到此处的小伙伴估计是**真爱粉**了 哈哈 其实 ts 没有大家想象的那么难 可能刚开始接触的时候会比较抵触 或者觉得难用 但是只要坚持下去 慢慢就会发现**真香定律** 咱们一开始也没有必要去追求多么花哨或者高级的用法 如果在**没有办法**的情况下就用**any 大法**也不是不可以 总之首先要用起来 只有不断地基于实战练习最终才能掌握一门技术的**精髓** 鲨鱼哥这篇文档只是理论知识 大家一定要下去多练习 另外 这篇文档是基于目前网上多方资源和鲨鱼哥自己的思考整理出来的个人觉得**比较全面**的 ts 学习指南 也感谢很多的优秀博主之前出品的 ts 文章 比如 阿宝哥 俊劫 Jimmy_kiwi 等等