

TypeScript语法精讲（四）

王红元 coderwhy

函数的重载

- 在TypeScript中，如果我们编写了一个add函数，希望对字符串和数字类型进行相加，应该如何编写呢？
- 我们可能会这样来编写，但是其实是错误的：

```
function sum(a1: number | string, a2: number | string): number | string {  
  return a1 + a2  
}
```

Operator '+' cannot be applied to types 'string | number' and 'string | number'. ts(2365)

(parameter) a2: string | number

[View Problem \(\F8\)](#) No quick fixes available

- 那么这个代码应该如何去编写呢？
 - 在TypeScript中，我们可以去编写不同的重载签名（*overload signatures*）来表示函数可以以不同的方式进行调用；
 - 一般是编写两个或者以上的重载签名，再去编写一个通用的函数以及实现；

sum函数的重载

■ 比如我们对sum函数进行重构：

□ 在我们调用sum的时候，它会根据我们传入的参数类型来决定执行函数体时，到底执行哪一个函数的重载签名；

```
function sum(a1: number, a2: number): number;
function sum(a1: string, a2: string): string;
function sum(a1: any, a2: any): any {
  return a1 + a2
}
```

```
console.log(sum(20, 30))
console.log(sum("aaa", "bbb"))
```

■ 但是注意，有实现提的函数，是不能直接被调用的：

```
sum({name: "why"}, {age: 18})
```

联合类型和重载

■ 我们现在有一个需求：定义一个函数，可以传入字符串或者数组，获取它们的长度。

■ 这里有两种实现方案：

□ 方案一：使用联合类型来实现；

□ 方案二：实现函数重载来实现；

```
function getLength(a: string|any[]) {  
    return a.length  
}
```

```
function getLength(a: string): number;  
function getLength(a: any[]): number;  
function getLength(a: any) {  
    return a.length  
}
```

■ 在开发中我们选择使用哪一种呢？

□ 在可能的情况下，尽量选择使用联合类型来实现；

认识类的使用

- 在早期的JavaScript开发中（ES5）我们需要通过函数和原型链来实现类和继承，从ES6开始，引入了class关键字，可以更加方便的定义和使用类。
- TypeScript作为JavaScript的超集，也是支持使用class关键字的，并且还可以对类的属性和方法等进行静态类型检测。
- 实际上在JavaScript的开发过程中，我们更加习惯于函数式编程：
 - 比如React开发中，目前更多使用的函数组件以及结合Hook的开发模式；
 - 比如在Vue3开发中，目前也更加推崇使用 Composition API；
- 但是在封装某些业务的时候，类具有更强大封装性，所以我們也需要掌握它們。
- 类的定义我们通常会使用class关键字：
 - 在面向对象的世界里，任何事物都可以使用类的结构来描述；
 - 类中包含特有的属性和方法；

类的定义

- 我们来定义一个Person类：
- 使用class关键字来定义一个类；
- 我们可以声明一些类的属性：在类的内部声明类的属性以及对应的类型
 - 如果类型没有声明，那么它们默认是any的；
 - 我们也可以给属性设置初始化值；
 - 在默认的strictPropertyInitialization模式下面我们的属性是必须初始化的，如果没有初始化，那么编译时就会报错；
 - ✓ 如果我们在strictPropertyInitialization模式下确实不希望给属性初始化，可以使用 name!: string语法；
- 类可以有自己的构造函数constructor，当我们通过new关键字创建一个实例时，构造函数会被调用；
 - 构造函数不需要返回任何值，默认返回当前创建出来的实例；
- 类中可以有自己的函数，定义的函数称之为方法；

```
class Person {  
    name!: string  
    age: number  
  
    constructor(name: string, age: number) {  
        // this.name = name  
        this.age = age  
    }  
  
    running() {  
        console.log(this.name + " running")  
    }  
  
    eating() {  
        console.log(this.name + " eating")  
    }  
}
```

类的继承

- 面向对象的其中一大特性就是继承，继承不仅仅可以减少我们的代码量，也是多态的使用前提。
- 我们使用extends关键字来实现继承，子类中使用super来访问父类。
- 我们来看一下Student类继承自Person：
 - Student类可以有自己的属性和方法，并且会继承Person的属性和方法；
 - 在构造函数中，我们可以通过super来调用父类的构造方法，对父类中的属性进行初始化；

```
class Student extends Person {  
  · sno: number  
  
  · constructor(name: string, age: number, sno: number) {  
    · super(name, age)  
    · this.sno = sno  
  }  
  
  · studying() {  
    · console.log(this.name + " studying")  
  }  
}
```

```
eating() {  
  · console.log("student eating")  
}  
  
running() {  
  · super.running();  
  · console.log("student running")  
}
```


类的成员修饰符

- 在TypeScript中，类的属性和方法支持三种修饰符：public、private、protected
 - public 修饰的是在任何地方可见、公有的属性或方法，默认编写的属性就是public的；
 - private 修饰的是仅在同一类中可见、私有的属性或方法；
 - protected 修饰的是仅在类自身及子类中可见、受保护的属性或方法；
- public是默认的修饰符，也是可以直接访问的，我们这里来演示一下protected和private。

```
class Person {  
  protected name: string  
  
  constructor(name: string) {  
    this.name = name;  
  }  
}  
  
class Student extends Person {  
  constructor(name: string) {  
    super(name)  
  }  
  
  running() {  
    console.log(this.name + " running")  
  }  
}
```

```
class Person {  
  private name: string  
  
  constructor(name: string) {  
    this.name = name  
  }  
}  
  
const p = new Person("why")  
// Property 'name' is private and only accessible within  
// console.log(p.name)
```


只读属性readonly

- 如果有一个属性我们不希望外界可以任意的修改，只希望确定值后直接使用，那么可以使用readonly：

```
class Person {  
  readonly name: string  
  
  constructor(name: string) {  
    this.name = name  
  }  
}  
  
const p = new Person("why")  
console.log(p.name)  
// Cannot assign to 'name' because it is a read-only property.  
// p.name = "coderwhy"  
  
export {}
```

getters/setters

- 在前面一些私有属性我们是不能直接访问的，或者某些属性我们想要监听它的获取(getter)和设置(setter)的过程，这个时候我们可以使用存取器。

```
class Person {  
  private _name: string  
  
  set name(newName) {  
    this._name = newName  
  }  
  
  get name() {  
    return this._name  
  }  
  
  constructor(name: string) {  
    this.name = name  
  }  
}
```

```
const p = new Person("why")  
p.name = "coderwhy"  
console.log(p.name)
```

- 前面我们在类中定义的成员和方法都属于对象级别的, 在开发中, 我们有时候也需要定义类级别的成员和方法。
- 在TypeScript中通过关键字static来定义 :

```
class Student {  
  static time: string = "20:00"  
  
  static attendClass() {  
    console.log("去上课")  
  }  
}  
  
console.log(Student.time)  
Student.attendClass()
```



抽象类abstract

- 我们知道，继承是多态使用的前提。
 - 所以在定义很多通用的调用接口时，我们通常会让调用者传入父类，通过多态来实现更加灵活的调用方式。
 - 但是，父类本身可能并不需要对某些方法进行具体的实现，所以父类中定义的方法，我们可以定义为抽象方法。
- 什么是 抽象方法？在TypeScript中没有具体实现的方法(没有方法体)，就是抽象方法。
 - 抽象方法，必须存在于抽象类中；
 - 抽象类是使用abstract声明的类；
- 抽象类有如下的特点：
 - 抽象类是不能被实例化的话（也就是不能通过new创建）
 - 抽象方法必须被子类实现，否则该类必须是一个抽象类；

抽象类演练

```
abstract class Shape {  
  abstract getArea(): number  
}
```

```
class Circle extends Shape {  
  private r: number  
  constructor(r: number) {  
    super()  
    this.r = r  
  }  
  
  getArea() {  
    return this.r * this.r * 3.14  
  }  
}  
  
class Rectangle extends Shape {  
  private width: number  
  private height: number  
  
  constructor(width: number, height: number) {  
    super()  
    this.width = width  
    this.height = height  
  }  
  
  getArea() {  
    return this.width * this.height  
  }  
}
```

```
const circle = new Circle(10)  
const rectangle = new Rectangle(20, 30)  
  
function calcArea(shape: Shape) {  
  console.log(shape.getArea())  
}  
  
calcArea(circle)  
calcArea(rectangle)
```

- 类本身也是可以作为一种数据类型的：

```
class Person {  
  name: string;  
  constructor(name: string) {  
    this.name = name;  
  }  
  running() {  
    console.log(this.name + " running");  
  }  
}  
  
const p1: Person = new Person("why");  
const p2: Person = {  
  name: "kobe",  
  running: function() {  
    console.log(this.name + " running");  
  },  
};
```

接口的声明

- 在前面我们通过type可以用来声明一个对象类型：

```
type Point = {  
  x: number  
  y: number  
}
```

- 对象的另外一种声明方式就是通过接口来声明：

```
interface Point {  
  x: number  
  y: number  
}
```

- 他们在使用上的区别，我们后续再来说明。
- 接下来我们继续学习一下接口的其他特性。

- 接口中我们也可以定义可选属性：

```
interface Person {  
  name: string  
  age?: number  
  friend?: {  
    name: string  
  }  
}  
  
const person: Person = {  
  name: "why",  
  age: 18,  
  friend: {  
    name: "kobe"  
  }  
}  
  
console.log(person.name)  
console.log(person.friend?.name)
```

■ 接口中也可以定义只读属性：

□ 这样就意味着我们再初始化之后，这个值是不可以被修改的；

```
interface Person {  
  · readonly name: string  
  · age?: number  
  · readonly friend?: {  
    · name: string  
  }  
}  
  
const person: Person = {  
  · name: "why",  
  · age: 18,  
  · friend: {  
    · name: "kobe"  
  }  
}
```

```
// person.name = "coderwhy" // 不可以设置  
// person.friend = {} // 不可以设置  
// 下面的代码是可以执行的  
if (person.friend) {  
  · person.friend.name = "123"  
}
```

索引类型

- 前面我们使用interface来定义对象类型，这个时候其中的属性名、类型、方法都是确定的，但是有时候我们会遇到类似下面的对象：

```
interface FrontLanguage {  
  [index: number]: string  
}  
  
const frontend: FrontLanguage = {  
  1: "HTML",  
  2: "CSS",  
  3: "JavaScript"  
}
```

```
interface LanguageBirth {  
  [name: string]: number  
  Java: number  
}  
  
const language: LanguageBirth = {  
  "Java": 1995,  
  "JavaScript": 1996,  
  "C": 1972  
}
```

- 前面我们都是通过interface来定义对象中普通的属性和方法的，实际上它也可以用来定义函数类型：

```
interface CalcFunc {  
  (num1: number, num2: number): number  
}  
  
const add: CalcFunc = (num1, num2) => {  
  return num1 + num2  
}  
  
const sub: CalcFunc = (num1, num2) => {  
  return num1 - num2  
}
```

- 当然，除非特别的情况，还是推荐大家使用类型别名来定义函数：

```
type CalcFunc = (num1: number, num2: number) => number
```

接口继承

- 接口和类一样是可以进行继承的，也是使用extends关键字：
 - 并且我们会发现，接口是支持多继承的（类不支持多继承）

```
interface Person {  
  · name: string  
  · eating: () => void  
}  
  
interface Animal {  
  · running: () => void  
}  
  
interface Student extends Person, Animal {  
  · sno: number  
}
```

```
const stu: Student = {  
  · sno: 110,  
  · name: "why",  
  · eating: function() {  
  
  },  
  · running: function() {  
    · · ·  
  }  
}
```

接口的实现

■ 接口定义后，也是可以被类实现的：

□ 如果被一个类实现，那么在之后需要传入接口的地方，都可以将这个类传入；

□ 这就是面向接口开发；

```
interface ISwim {  
  swimming: () => void  
}  
  
interface IRun {  
  running: () => void  
}  
  
class Person implements ISwim, IRun {  
  swimming() {  
    console.log("swimming")  
  }  
  
  running() {  
    console.log("running")  
  }  
}
```

```
function swim(swimmer: ISwim) {  
  swimmer.swimming()  
}  
  
const p = new Person()  
swim(p)
```

■ 前面我们学习了联合类型：

- 联合类型表示多个类型中一个即可

```
type Alignment = 'left' | 'right' | 'center'
```

■ 还有另外一种类型合并，就是交叉类型（Intersection Types）：

- 交叉类型表示需要满足多个类型的条件；
- 交叉类型使用 & 符号；

■ 我们来看下面的交叉类型：

- 表达的含义是number和string要同时满足；
- 但是有同时满足是一个number又是一个string的值吗？其实是没有的，所以MyType其实是一个never类型；

```
type MyType = number & string
```


交叉类型的应用

- 所以，在开发中，我们进行交叉时，通常是对对象类型进行交叉的：

```
interface Colorful {  
  color: string  
}  
  
interface IRun {  
  running: () => void  
}  
  
type NewType = Colorful & IRun  
  
const obj: NewType = {  
  color: "red",  
  running: function() {  
    ...  
  }  
}
```

interface和type区别

- 我们会发现interface和type都可以用来定义对象类型，那么在开发中定义对象类型时，到底选择哪一个呢？
 - 如果是定义非对象类型，通常推荐使用type，比如Direction、Alignment、一些Function；
- 如果是定义对象类型，那么他们是有区别的：
 - interface 可以重复的对某个接口来定义属性和方法；
 - 而type定义的是别名，别名是不能重复的；

```
interface IPerson {  
  name: string  
  running: () => void  
}
```

```
interface IPerson {  
  age: number  
}
```

```
type Person = {  
  name: string  
  running: () => void  
}  
  
// error: Duplicate identifier 'Person'.ts(2300)  
type Person = {  
  age: number  
}
```

字面量赋值

■ 我们来看下面的代码：

```
interface IPerson {  
  name: string  
  eating: () => void  
}  
  
// Object literal may only specify known properties, and 'age' does not exist  
const p: IPerson = {  
  name: "why",  
  age: 18,  
  eating: function() {  
  }  
}
```

```
interface IPerson {  
  name: string  
  eating: () => void  
}  
  
const obj = {  
  name: "why",  
  age: 18,  
  eating: function() {  
  }  
}  
  
const p: IPerson = obj
```

■ 这是因为TypeScript在字面量直接赋值的过程中，为了进行类型推导会进行严格的类型限制。

□ 但是之后如果我们是将一个 变量标识符 赋值给其他的变量时，会进行freshness擦除操作。