TypeScript面向对象

王红元 coderwhy

目录
content

1 TypeScript类的使用

2 TypeScript中抽象类

3 TypeScript对象类型

4 TypeScript接口补充

5 特殊:严格字面量检测

6 TypeScript枚举类型

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，所以我们也需要掌握它们。**

- **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。**

```ts
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")
  }
}
```

```ts
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)
```

参数属性（Parameter Properties）

- TypeScript 提供了特殊的语法，可以把一个构造函数参数转成一个同名同值的类属性。
  - 这些就被称为参数属性（parameter properties）；
  - 你可以通过在构造函数参数前添加一个可见性修饰符 public private protected 或者 readonly 来创建参数属性，最后这些类属性字段也会得到这些修饰符；

```
class Person {
  constructor(public name: string, private _age: number) {
  }
  set age(newAge) {
    this._age = newAge
  }
  get age() {
    return this._age
  }
}
```

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的话（也就是不能通过new创建）
  - 抽象类可以包含抽象方法，也可以包含有实现体的方法；
  - 有抽象方法的类，必须是一个抽象类；
  - 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

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

```ts
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
  }
}
```

```ts
const circle = new Circle(10)
const rectangle = new Rectangle(20, 30)

function calcArea(shape: Shape) {
  console.log(shape.getArea())
}

calcArea(circle)
calcArea(rectangle)
```

类的类型

- 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");
  },
};
```

对象类型的属性修饰符（Property Modifiers）

- **对象类型中的每个属性可以说明它的类型、属性是否可选、属性是否只读等信息。**
- **可选属性（Optional Properties）**
  - 我们可以在属性名后面加一个？标记表示这个属性是可选的；
- **只读属性（Readonly Properties）**
  - 在 TypeScript 中，属性可以被标记为 readonly，这不会改变任何运行时的行为；
  - 但在类型检查的时候，一个标记为 readonly的属性是不能被写入的。

```ts
interface IPerson {
  name: string
  age?: number
  readonly height: number
}

const p: IPerson = {
  name: "why",
  height: 1.88
}
```

索引签名（Index Signatures）

- **什么是索引签名呢?**
  - 有的时候，你不能提前知道一个类型里的所有属性的名字，但是你知道这些值的特征；
  - 这种情况，你就可以用一个索引签名 (index signature) 来描述可能的值的类型；

```ts
interface ICollection {
  [index: number]: any
  length: number
}

function logCollection(collection: ICollection) {
  for (let i = 0; i < collection.length; i++) {
    console.log(collection[i])
  }
}
```

```ts
const tuple: [string, number, number] = ["why", 18, 1.88]
const array: string[] = ["aaa", "bbb", "ccc"]
logCollection(tuple)
console.log(array);
```

- **一个索引签名的属性类型必须是 string 或者是 number。**
  - 虽然 TypeScript 可以同时支持 string 和 number 类型，但数字索引的返回类型一定要是字符索引返回类型的子类型；（了解）

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）

```ts
interface Person {
  name: string
  eating: () => void
}

interface Animal {
  running: () => void
}

interface Student extends Person, Animal {
  sno: number
}
```

```ts
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；

```ts
interface ISwim {
  swimming: () => void
}

interface IRun {
  running: () => void
}

class Person implements ISwim, IRun {
  swimming() {
    console.log("swimming")
  }

  running() {
    console.log("running")
  }
}
```

```ts
function swim(swimmer: ISwim) {
  swimmer.swimming()
}

const p = new Person()
swim(p)
```

抽象类和接口的区别（了解）

- **抽象类在很大程度上和接口会有点类似：都可以在其中定义一个方法，让子类或实现类来实现对应的方法。**
- **那么抽象类和接口有什么区别呢?**
  - 抽象类是事物的抽象，抽象类用来捕捉子类的通用特性，接口通常是一些行为的描述；
  - 抽象类通常用于一系列关系紧密的类之间，接口只是用来描述一个类应该具有什么行为；
  - 接口可以被多层实现，而抽象类只能单一继承；
  - 抽象类中可以有实现体，接口中只能有函数的声明；
- **通常我们会这样来描述类和抽象类、接口之间的关系：**
  - 抽象类是对事物的抽象，表达的是 is a 的关系。猫是一种动物（动物就可以定义成一个抽象类）
  - 接口是对行为的抽象，表达的是 has a 的关系。猫拥有跑（可以定义一个单独的接口）、爬树（可以定义一个单独的接口）的行为。

严格的字面量赋值检测

- 对于对象的字面量赋值，在TypeScript中有一个非常有意思的现象：

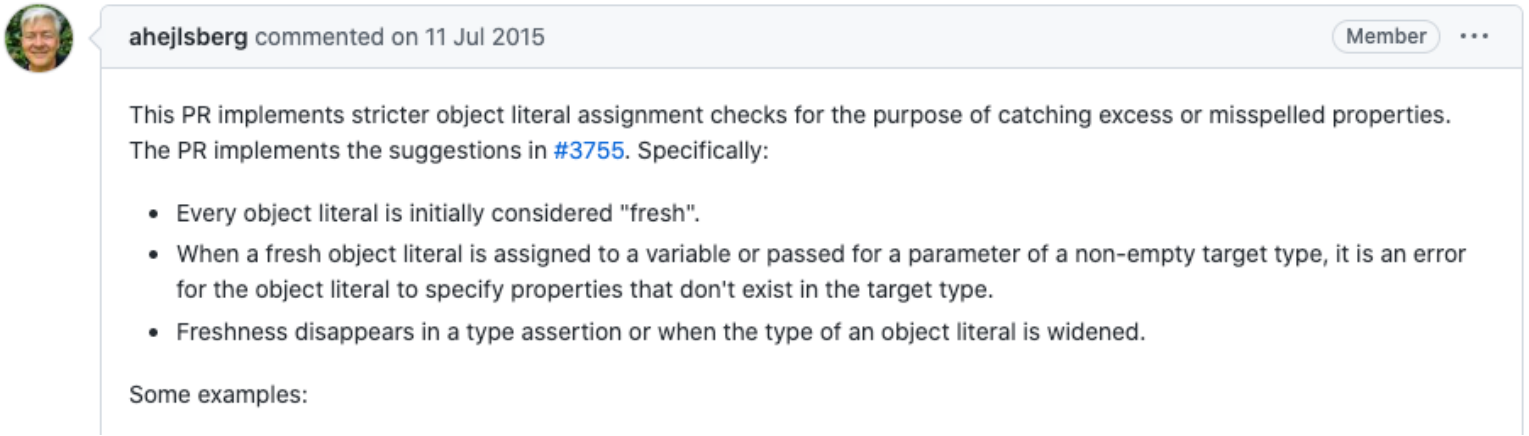
```ts
interface IPerson {
  name: string
  eating: () => void
}

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

```ts
function printInfo(info: IPerson) {
  console.log(info.name, info.age)
}

printInfo({name: "why", age: 18, height: 1.88})
```

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

const obj = {
  name: "why",
  age: 18,
  eating: function() {

  }
}

const p: IPerson = obj
```

为什么会出现这种情况呢?

- 这里我引入TypeScript成员在GitHub的issue中的回答:



ahejlsberg commented on 11 Jul 2015 Member

This PR implements stricter object literal assignment checks for the purpose of catching excess or misspelled properties. The PR implements the suggestions in #3755. Specifically:

- Every object literal is initially considered "fresh".
- When a fresh object literal is assigned to a variable or passed for a parameter of a non-empty target type, it is an error for the object literal to specify properties that don't exist in the target type.
- Freshness disappears in a type assertion or when the type of an object literal is widened.

Some examples:

- 简单对上面的英文进行翻译解释:
  - 每个对象字面量最初都被认为是“新鲜的（fresh）”。
  - 当一个新的对象字面量分配给一个变量或传递给一个非空目标类型的参数时，对象字面量指定目标类型中不存在的属性是错误的。
  - 当类型断言或对象字面量的类型扩大时，新鲜度会消失。

TypeScript枚举类型

- **枚举类型是为数不多的TypeScript特性有的特性之一：**
  - 枚举其实就是将一组可能出现的值，一个个列举出来，定义在一个类型中，这个类型就是枚举类型；
  - 枚举允许开发者定义一组命名常量，常量可以是数字、字符串类型；

```
enum Direction {
  LEFT,
  RIGHT,
  TOP,
  BOTTOM
}
```

```
function turnDirection(direction: Direction) {
  switch (direction) {
    case Direction.LEFT:
      console.log("转向左边~")
      break;
    case Direction.RIGHT:
      console.log("转向右边~")
      break;
    case Direction.TOP:
      console.log("转向上边~")
      break;
    case Direction.BOTTOM:
      console.log("转向下边~")
      break;
    default:
      const myDirection: never = direction;
  }
}
```

枚举类型的值

- 枚举类型默认是有值的，比如上面的枚举，默认值是这样的：
- 当然，我们也可以给枚举其他值：
  - 这个时候会从100进行递增；
- 我们也可以给他们赋值其他的类型：

```
enum Direction {
  LEFT = 0,
  RIGHT = 1,
  TOP = 2,
  BOTTOM = 3
}
```

```
enum Direction {
  LEFT = 100,
  RIGHT,
  TOP,
  BOTTOM
}
```

```
enum Direction {
  LEFT,
  RIGHT,
  TOP = "TOP",
  BOTTOM = "BOTTOM"
}
```