【TypeScript 4.5】006-第 6 章 对象类型

【TypeScript 4.5】006-第 6 章 对象类型
一、认识对象类型
1、概述
说明
对象类型
2、代码演示
二、可选属性
1、属性修改器
2、可选属性
三、只读属性
1、概述
2、代码演示
不同层次的只读属性
另外一种情况
四、索引签名
1、概述
2、代码演示
索引类型为 number
索引类型为 string
索引类型为 boolean （报错）
关于其他键值对
只读索引签名
索引签名和数组区别
五、扩展类型
1、概述
说明
代码示例
2、代码演示
单继承
多继承
六、交叉类型
1、概述
说明
代码示例
2、代码演示
七、接口与交叉类型
1、概述
说明
二者主要区别
2、代码演示
同名接口
同名类型别名
八、泛型对象类型
1、概述
说明
代码示例
2、代码演示

一、认识对象类型

1、概述

说明

在 JavaScript 中我们分组和传递数据的基本方式是通过对象完成的

在 TypeScript 中我们通过对象类型来表示对象

对象类型

匿名对象、接口命名、类型别名。

2、代码演示

```
// 1、匿名方式
function greet(person: { name: string, age: number }) {
    return "hello " + person.name + "I am " + person.age + " years old!"
}
// 接口对象类型
interface Person {
    name: string,
    age: number
}
// 2、接口方式
function greet1(person: Person) {
    return "hello " + person.name + "I am " + person.age + " years old!"
}
// 类型别名对象类型
type Person1 = {
    name: string,
    age: number
}
// 3、类型别名方式
function greet2(person: Person1) {
    return "hello " + person.name + "I am " + person.age + " years old!"
}
```

二、可选属性

1、属性修改器

对象类型中的每个属性都可以指定：

1）定义对象类型

2）设置属性是否是可选的

3）属性是否可以被写入

2、可选属性

```
type Shape = {}
interface PaintOptions {
    shape: Shape,
    xPos?: number,
    yPos?: number
}
function printShape(opts: PaintOptions) {
    // 此处，当 xPos 或 yPos 未传入的时候打印结果未 undefined
    // 如果加判断给默认值的话，会很麻烦！
    // 我们通过解构的方式赋予默认值，见 printShape1()
}
function printShape1({ shape, xPos = 0, yPos = 0 }: PaintOptions) {
    // 如果在解构里这么写：{ shape: Shape, xPos: number = 0, yPos = 0 } Shape 和
number 并不是类型，而是别名（ ES6 语法）
}
const shape: Shape = {}
printShape({ shape })
printShape({ shape, xPos: 100 })
printShape({ shape, yPos: 100 })
printShape({ shape, xPos: 100, yPos:100 })
```

三、只读属性

1、概述

它不会在运行的时候改变任何行为

但是在**类型检查期间**

一个标记为**只读**的属性

是不能够被写入其他值的

2、代码演示

不同层次的只读属性

```
// 不同层次的只读属性
interface SomeType {
    // 只读属性
    readonly prop: string
}
function doSth(obj: SomeType) {
    console.log(obj.prop)
    // obj.prop = "hello" // 报错：无法分配到 "prop" ，因为它是只读属性。
}
interface Home {
    readonly resident: {
        // 如果在 name 和 age 前面也加上 readonly 那么对其修改也会报错
        name: string,
```

```
        age: number
    }
}
function visitForBirthday(home: Home) {
    console.log(home.resident.name)
    console.log(home.resident.age)
    home.resident.name = "hello" // 正常，不报错
    home.resident.age ++ // 正常，不报错
}
function evict(home: Home) {
    home.resident = { // 报错：无法分配到 "resident" ，因为它是只读属性。
        name: "Hello",
        age: 22
    }
}
```

另外一种情况

```
// 另一种情况
interface Person {
    name: string,
    age: number
}
interface ReadonlyPerson {
    readonly name: string,
    readonly age: number
}
let writablePerson: Person = {
    name: "訾博",
    age: 25
}
let readonlyPerosn: ReadonlyPerson = writablePerson // 这里不报错
console.log(readonlyPerosn.name) // 正常，不报错
console.log(readonlyPerosn.age) // 正常，不报错，结果为 25
writablePerson.age ++
console.log(readonlyPerosn.age) // 结果为 26
```

四、索引签名

1、概述

有时候我们**不**能提前**知道**一个类型的所有**属性的名称**

但是我们**知道这个值**的形状

这种情况，我们可以使用**索引签名**来描述可能的值的类型

> 所谓索引签名，就是知道属性值的类型，不知道属性名，定义一个类似 `[props: string]: number` 的东西，表示属性名是字符串，属性值是 number 类型！
>
> 参考文章：https://blog.csdn.net/weixin_43294560/article/details/104994109

2、代码演示

> 课程只是冰山一角，TypeScript 还有星辰大海！

索引类型为 number

```
// 索引类型为 number
interface StringArray {
    [index: number]: string
}
// 下面 myArr 与 myArr01 两种写法应该是等效的
const myArr: StringArray = ['a', 'b', 'c']
console.log(myArr[0]) // a
console.log(myArr[1]) // b
console.log(myArr[2]) // c
console.log(myArr[3]) // undefined
const myArr01: StringArray = {
    0: 'a',
    1: 'b',
    2: 'c'
}
console.log(myArr01[0]) // a
console.log(myArr01[1]) // b
console.log(myArr01[2]) // c
console.log(myArr01[3]) // undefined
// 下面这种写法得出：前面的 1 2 3 不会影响实际索引！
const myArr02: StringArray = {
    1: 'a',
    2: 'b',
    3: 'c'
}
console.log(myArr01[0]) // a
console.log(myArr01[1]) // b
console.log(myArr01[2]) // c
console.log(myArr01[3]) // undefined
```

索引类型为 string

```
// 索引类型为 string
interface TestString {
    [props: string]: number
}
// 下面 testString 与 testString01 两种写法应该是等效的
let testStr: TestString = {
    x: 100,
    y: 200,
    z: 300
}
console.log(testStr.x) // 100
console.log(testStr.y) // 200
console.log(testStr.z) // 300
console.log(testStr.a) // undefined
let testStr01: TestString = {
    'x': 100,
```

```
    'y': 200,
    'z': 300
}
console.log(testStr01.x) // 100
console.log(testStr01.y) // 200
console.log(testStr01.z) // 300
console.log(testStr01.a) // undefined
console.log(testStr01[0]) // undefined
console.log(testStr01[1]) // undefined
console.log(testStr01[2]) // undefined
console.log(testStr01[3]) // undefined
// 下面这种写法值得重视，跟想象中不一样！
let testStr02: TestString = {
    1: 100,
    2: 200,
    3: 300
}
console.log(testStr02[0]) // undefined
console.log(testStr02[1]) // 100
console.log(testStr02[2]) // 200
console.log(testStr02[3]) // 300
let testStr03: TestString = {
    0: 50,
    1: 100,
    2: 200
}
console.log(testStr03[0]) // 50
console.log(testStr03[1]) // 100
console.log(testStr03[2]) // 200
console.log(testStr03[3]) // undefined
// 这么写报错：不能将类型“number[]”分配给类型“TestString”。
// 类型“number[]”中缺少类型“string”的索引签名。
// let testStr04: TestString = [100, 200, 300]
```

索引类型为 boolean （报错）

```
// 索引类型为 boolean
interface TestBoolean {
    [props: boolean]: number // 报错：索引签名参数类型必须是
“string”、“number”、“symbol”或模板文本类型。
}
// 也就说明属性名的取值范围是：“string”、“number”、“symbol”或模板文本类型。
```

关于其他键值对

> 其他键值对都要满足索引签名！

```ts
interface TestOther {
    [index: string]: number,
    // 接口中的其他键值对都要满足索引签名
    length: number,
    name: string // 报错：类型“string”的属性“name”不能赋给“string”索引类型“number”。
}
interface TestOther01 {
    [index: string]: number | string,
    // 接口中的其他键值对都要满足索引签名
    length: number,
    name: string // 正常
}
```

只读索引签名

```ts
// 只读索引签名
interface TestReadonly {
    readonly [index: number]: string
}
let testReadonly: TestReadonly = ["a", "b", "c"]
testReadonly[0] = "hello" // 报错：类型“TestReadonly”中的索引签名仅允许读取。
interface TestReadonly01 {
    readonly [props: string]: string
}
let testReadonly01: TestReadonly01 = {
    name: "訾博",
    gender: "男"
}
testReadonly01.name = "刘备" // 报错：类型“TestReadonly01”中的索引签名仅允许读取。
```

索引签名和数组区别

> 没有数组的方法或属性

```ts
interface StringArray {
    [index: number]: string
}
const myArr: StringArray = ['a', 'b', 'c']
// myArr.push("hello") // 报错：类型“StringArray”上不存在属性“push”。
// 使用数组的方法或属性
// 方式一：类型断言
const myArr01: StringArray = ['a', 'b', 'c']
const strs: string[] = myArr01 as string[]
strs.push("hello")
console.log(myArr01) // [ 'a', 'b', 'c', 'hello' ]
// 方式二：在定义索取签名时添加需要的属性或方法
// 暂不演示方法***
interface StringArray01 {
    [index: number]: number,
    length: number // 返回的直接是属性的数量（除了 length 之外的）
}
const myArr02: StringArray01 = [10, 20, 30, 40, 50]
```

```
console.log(myArr02)
console.log("myArr02.length ", myArr02.length)
```

五、扩展类型

1、概述

说明

有一些类型可能是其他类型更具体的版本

代码示例

> 看到 extends 就懂了！

```
interface BasicAddress {
    // ...
}
interface AddressWithUnit extends BasicAddress {
    unit: string
}
```

2、代码演示

> 继承和扩展是一个意思！

单继承

```
interface BasicAddress {
    name?: string,
    street: string,
    city: string,
    country: string,
    postalCode: string
}
interface AddressWithUnit extends BasicAddress {
    unit: string
}
let zibo: AddressWithUnit = {
    name: "砥博",
    street: "街道",
    city: "城市",
    country: "国家",
    postalCode: "000000",
    unit: "单元"
}
console.log(zibo.name) // 砥博
```

多继承

```
interface Father {
    tall: number
}
interface Monther {
    face: string
}
interface Son extends Father, Monther {
    knowledge: string
}
let son: Son = {
    tall: 188,
    face: "beautiful",
    knowledge: "abundant"
}
```

六、交叉类型

1、概述

说明

> 就是多个对象类型的**并集**!

接口允许我们通过扩展其他类型建立起新类型

TypeScript 还提供另外一种其他结构

称为**交叉类型**

主要用于组合现有的对象类型

代码示例

```
type ColorfulCircle = Colorful & Circle
```

2、代码演示

```
interface Father {
    tall: number
}
interface Monther {
    face: string
}
type Son = Father & Monther

let son: Son = {
    tall: 188,
    face: 'beautiful'
```

```
}
function getInfo(son: Father & Monther) {
    console.log(son.tall) // 188
    console.log(son.face) // beautiful
}
// 可以这么使用
getInfo(son)
// 试试能否多个对象类型进行交叉
interface Grandpa {
    habit: string
}
interface Grandma {
    hobby: string
}
type NewSon = Father & Monther & Grandpa & Grandma
let newSon: NewSon = {
    tall: 188,
    face: "beautiful",
    habit: "sport",
    hobby: "study"
}
console.log(newSon.face) // beautiful
```

七、接口与交叉类型

1、概述

说明

接口可以使用 extends 来扩展其他类型

交叉类型，我们可以通过 type 类型别名定义

把两个类型之间用 & 符号交叉联合起来

二者主要区别

在于如何处理冲突?

2、代码演示

同名接口

> 同名接口的属性会合并

```
interface Student {
    name: string
}
interface Student {
    age: number
}
// 结论：同名接口的属性会合并（但同名接口有同名属性的话会报错，除非一摸一样的属性）
let stu: Student = {
    name: 'zibo',
    age: 25
}
console.log(stu.name) // zibo
console.log(stu.age) // 25
```

同名类型别名

无法定义同名类型别名。

八、泛型对象类型

1、概述

说明

之前我们定义对象类型

可以定义任意的属性以及属性的类型

这些类型都是一些具体的类型

我们能否泛化这些类型呢?

代码示例

> 使用 any 带来了编写程序的遍历，但失去了使用类型的意义！

```
interface Box {
    contents: any
}
```

> unknown 要求非常严苛，不能随意给其赋值！

```
interface Box {
    contents: unknown
}
```

> 泛型大法好！

```
interface Box<T> {
    contents: T
}
let box: Box<string> = {
    contents: "大家好！"
}
```

2、代码演示

```
interface Box<T> {
    contents: T
}
let box: Box<string> = {
    contents: "大家好！"
}
// 使用类型别名定义
type NewBox<T> = {
    contents: T
}
// 扩展演示（套娃）
type OrNull<T> = T | null
type OneOrMany<T> = T | T[]
type OneOrManyOrNull<T> = OrNull<OneOrMany<T>>
type OneOrManyOrNullString = OneOrManyOrNull<string>
```