

TypeScript语法细节

王红元 coderwhy

目录

content



1 联合类型和交叉类型

2 type和interface使用

3 类型断言和非空断言

4 字面量类型和类型缩小

5 函数的类型和函数签名

6 函数的重载和this类型

联合类型

- TypeScript的类型系统允许我们使用多种运算符，从**现有类型中构建新类型**。
- 我们来使用第一种组合类型的方法：**联合类型 (Union Type)**
 - 联合类型是由**两个或者多个其他类型组成的类型**；
 - 表示**可以是这些类型中的任何一个值**；
 - 联合类型中的每一个类型被称之为**联合成员 (union's members)** ；

```
function printId(id: number | string) {  
  console.log("你的id是:", id)  
}  
  
printId(10)  
printId("abc")
```

使用联合类型

■ 传入给一个联合类型的值是非常简单的：只要保证是联合类型中的某一个类型的值即可

□ 但是我们拿到这个值之后，我们应该如何使用它呢？因为它可能是任何一种类型。

□ 比如我们拿到的值可能是string或者number，我们就不能对其调用string上的一些方法；

■ 那么我们怎么处理这样的问题呢？

□ 我们需要使用缩小（narrow）联合（后续我们还会专门讲解缩小相关的功能）；

□ TypeScript可以根据我们缩小的代码结构，推断出更加具体的类型；

```
function printId(id: number | string) {  
  if (typeof id === 'string') {  
    // 确定id是string类型  
    console.log("你的id是:", id.toUpperCase())  
  } else {  
    // 确定id是number类型  
    console.log("你的id是", id)  
  }  
}
```

类型别名

- 在前面，我们通过在类型注解中编写 对象类型 和 联合类型，但是当我们想要多次在其他地方使用时，就要编写多次。
- 比如我们可以给对象类型起一个别名：

```
type Point = {  
  x: number  
  y: number  
}  
  
function printPoint(point: Point) {  
  console.log(point.x, point.y)  
}  
  
function sumPoint(point: Point) {  
  console.log(point.x + point.y)  
}  
  
printPoint({x: 20, y: 30})  
sumPoint({x: 20, y: 30})
```

```
type ID = number | string  
  
function printId(id: ID) {  
  console.log("您的id:", id)  
}
```

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  
}
```

- 那么它们有什么区别呢？

- 类型别名和接口非常相似，在定义对象类型时，大部分时候，你可以任意选择使用。
- 接口的几乎所有特性都可以在 type 中使用（后续我们还会学习interface的很多特性）；

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  
}
```

■ 所以，interface可以为现有的接口提供更多的扩展。

□ 接口还有很多其他的用法，我们会在后续详细学习

交叉类型

■ 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() {  
    ...  
  }  
}
```

- 有时候TypeScript无法获取具体的类型信息，这个我们需要使用类型断言（Type Assertions）。

- 比如我们通过 document.getElementById，TypeScript只知道该函数会返回 HTMLElement，但并不知道它具体的类型：

```
const myEl = document.getElementById("my-img") as HTMLImageElement

myEl.src = "图片地址"
```

- TypeScript只允许类型断言转换为 更具体 或者 不太具体 的类型版本，此规则可防止不可能的强制转换：

```
myEl.src = "图
// Conversion overlaps with
const name = "coderwhy" as number;
```

Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first. ts(2352)

[View Problem](#) (^F8) [Quick Fix...](#) (⌘.)

```
const name = ("coderwhy" as unknown) as number;
```

非空类型断言!

- 当我们编写下面的代码时，在执行ts的编译阶段会报错：

- 这是因为传入的message有可能是为undefined的，这个时候是不能执行方法的；

```
function printMessage(message?: string) {  
  // error TS2532: Object is possibly 'undefined'  
  console.log(message.toUpperCase())  
}  
  
printMessage("hello")
```

- 但是，我们确定传入的参数是有值的，这个时候我们可以使用非空类型断言：

- 非空断言使用的是！，表示可以确定某个标识符是有值的，跳过ts在编译阶段对它的检测；

```
function printMessage(message?: string) {  
  console.log(message!.toUpperCase())  
}
```

字面量类型

- 除了前面我们所讲过的类型之外，也可以使用字面量类型（literal types）：

```
let message: "Hello World" = "Hello World"
// Type '"你好啊, 李银河"' is not assignable to type '"Hello World"'.
message = "你好啊, 李银河"
```

- 那么这样做有什么意义呢？

- 默认情况下这么做是没有太大的意义的，但是我们可以将多个类型联合在一起；

```
type Alignment = 'left' | 'right' | 'center'
function changeAlign(align: Alignment) {
  console.log("修改方向:", align)
}

changeAlign("left")
```

■ 我们来看下面的代码：

```
const info = {  
  url: "https://coderwhy.org/abc",  
  method: "GET"  
}  
  
function request(url: string, method: "GET" | "POST") {  
  console.log(url, method)  
}  
  
request(info.url, info.method)
```

- 这是因为我们的对象在进行字面量推理的时候，info其实是一个 {url: string, method: string}，所以我们没办法将一个 string 赋值给一个 字面量 类型。

// 方式一：

```
request(info.url, info.method as "GET")
```

```
const info = {  
  url: "https://coderwhy.org/abc",  
  method: "GET"  
} as const
```

■ 什么是类型缩小呢？

- 类型缩小的英文是 **Type Narrowing**（也有人翻译成类型收窄）；
- 我们可以通过类似于 `typeof padding === "number"` 的判断语句，来**改变TypeScript的执行路径**；
- 在给定的执行路径中，我们可以**缩小比声明时更小的类型**，这个过程称之为 **缩小（Narrowing）**；
- 而我们编写的 `typeof padding === "number"` 可以称之为 **类型保护（type guards）**；

■ 常见的类型保护有如下几种：

- `typeof`
- 平等缩小（比如`===`、`!==`）
- `instanceof`
- `in`
- 等等...

■ 在 TypeScript 中，检查返回的值typeof是一种类型保护：

□ 因为 TypeScript 对如何typeof操作不同的值进行编码。

```
type ID = number | string

function printId(id: ID) {
  if (typeof id === 'string') {
    console.log(id.toUpperCase())
  } else {
    console.log(id)
  }
}
```

- 我们可以使用Switch或者相等的一些运算符来表达相等性（比如===, !==, ==, and != ）：

```
type Direction = 'left' | 'right' | 'center'
function turnDirection(direction: Direction) {
  switch (direction) {
    case 'left':
      console.log("调用left方法")
      break
    case 'right':
      console.log("调用right方法")
      break
    case 'center':
      console.log("调用center方法")
      break
    default:
      console.log("调用默认方法")
  }
}
```


instanceof

- JavaScript 有一个运算符来检查一个值是否是另一个值的“实例”：

```
function printValue(date: Date|string) {  
  if (date instanceof Date) {  
    console.log(date.toLocaleString())  
  } else {  
    console.log(date)  
  }  
}
```

in操作符

■ Javascript 有一个运算符，用于确定对象是否具有带名称的属性：in运算符

□ 如果指定的属性在指定的对象或其原型链中，则in 运算符返回true;

```
type Fish = {swim: () => void}
type Dog = {run: () => void}

function move(animal: Fish | Dog) {
  if ('swim' in animal) {
    animal.swim()
  } else {
    animal.run()
  }
}
```

TypeScript函数类型

- 在JavaScript开发中，函数是重要的组成部分，并且函数可以**作为一等公民**（可以作为参数，也可以作为返回值进行传递）。
- 那么在使用函数的过程中，函数是否也可以有自己的类型呢？
- 我们可以编写**函数类型的表达式（Function Type Expressions）**，来表示函数类型；

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

function calc(fn: CalcFunc) {
  console.log(fn(20, 30))
}

function sum(num1: number, num2: number) {
  return num1 + num2
}

function mul(num1: number, num2: number) {
  return num1 * num2
}

calc(sum)
calc(mul)
```

TypeScript函数类型解析

■ 在上面的语法中 `(num1: number, num2: number) => void`，代表的就是一个函数类型：

- 接收两个参数的函数：num1和num2，并且都是number类型；
- 并且这个函数是没有返回值的，所以是void；

■ 注意：在某些语言中，可能参数名称num1和num2是可以省略，但是TypeScript是不可以的：

Note that the parameter name is **required**. The function type `(string) => void` means "a function with a parameter named `string` of type `any` "!

调用签名 (Call Signatures)

- 在 JavaScript 中，函数除了可以被调用，自己也是可以有属性值的。
 - 然而前面讲到的函数类型表达式并不能支持声明属性；
 - 如果我们想描述一个带有属性的函数，我们可以在一个对象类型中写一个调用签名 (call signature) ；

```
interface ICalcFn {  
  name: string  
  (num1: number, num2: number): void  
}  
  
function calc(calcFn: ICalcFn) {  
  console.log(calcFn.name)  
  calcFn(10, 20)  
}
```

- 注意这个语法跟函数类型表达式稍有不同，在参数列表和返回的类型之间用的是：而不是 =>。

构造签名 (Construct Signatures)

■ JavaScript 函数也可以使用 new 操作符调用，当被调用的时候，TypeScript 会认为这是一个构造函数(constructors)，因为他们会产生一个新对象。

□ 你可以写一个构造签名 (Construct Signatures)，方法是在调用签名前面加一个 new 关键词；

```
interface IPerson {  
  new (name: string): Person  
}  
  
function factory(ctor: IPerson) {  
  return new ctor("why")  
}  
  
class Person {  
  name: string  
  constructor(name: string) {  
    this.name = name  
  }  
}  
  
factory(Person)
```

参数的可选类型

- 我们可以指定某个参数是可选的：

```
function foo(x: number, y?: number) {  
  console.log(x, y)  
}
```

- 这个时候这个参数y依然是有类型的，它是什么类型呢？ `number | undefined`

Although the parameter is specified as type `number`, the `x` parameter will actually have the type `number | undefined` because unspecified parameters in JavaScript get the value `undefined`.

- 另外可选类型需要在必传参数的后面：

```
function foo(x?: number, y: number) {}
```

'y' is declared but its value is never read. ts(6133)

A required parameter cannot follow an optional parameter. ts(1016)

(parameter) y: number

[View Problem](#) (`\F8`) [Quick Fix...](#) (`\F.`)

- 从ES6开始，JavaScript是支持默认参数的，TypeScript也是支持默认参数的：

```
function foo(x: number, y: number = 6) {  
  console.log(x, y)  
}  
  
foo(10)
```

- 这个时候y的类型其实是 `undefined` 和 `number` 类型的联合。

剩余参数

- 从ES6开始，JavaScript也支持剩余参数，剩余参数语法允许我们将一个不定数量的参数放到一个数组中。

```
function sum(...nums: number[]) {  
  let total = 0  
  for (const num of nums) {  
    total += num  
  }  
  return total  
}  
  
const result1 = sum(10, 20, 30)  
console.log(result1)  
  
const result2 = sum(10, 20, 30, 40)  
console.log(result2)
```

函数的重载（了解）

- 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, 有实现体的函数, 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；

可推导的this类型

■ this是JavaScript中一个比较难以理解和把握的知识点：

□ 我在公众号也有一篇文章专门讲解this: https://mp.weixin.qq.com/s/hYm0JgBl25grNG_2sCRITA;

■ 当然在目前的Vue3和React开发中你不一定会使用到this：

□ Vue3的Composition API中很少见到this，React的Hooks开发中也很少见到this了；

■ 但是我们还是简单掌握一些TypeScript中的this，TypeScript是如何处理this呢？我们先来看两个例子：

```
const obj = {  
  name: "obj",  
  foo: function() {  
    console.log(this.name)  
  }  
}  
  
obj.foo()
```

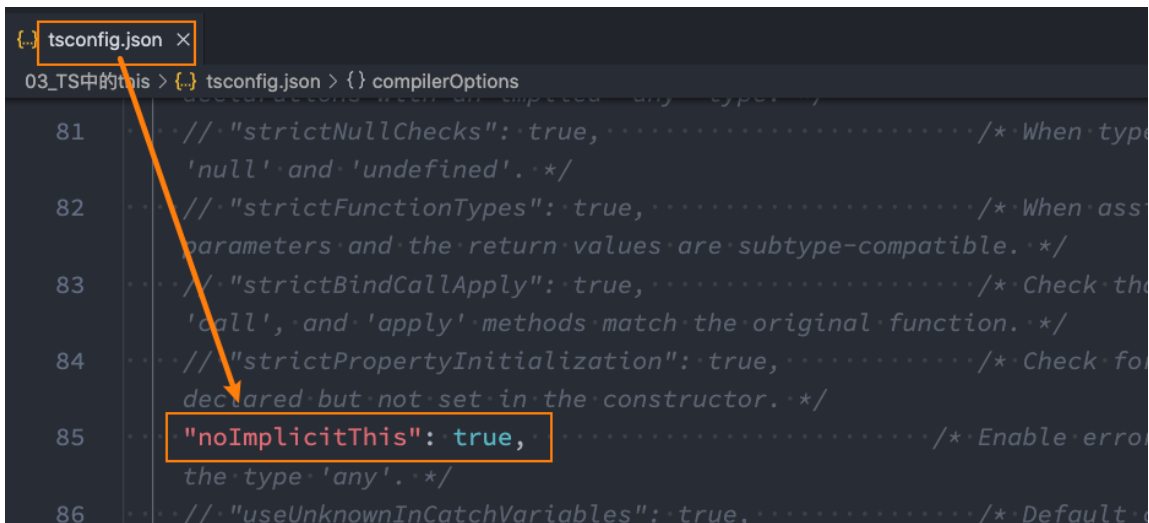
```
function foo1() {  
  console.log(this)  
}  
  
foo1()
```

■ 上面的代码默认情况下是可以正常运行的，也就是TypeScript在编译时，认为我们的this是可以正确去使用的：

□ 这是因为在没有指定this的情况，this默认情况下是any类型的；

this的编译选项

- VSCode在检测我们的TypeScript代码时，默认情况下运行不确定的this按照any类型去使用。
 - 但是我们可以创建一个tsconfig.json文件，并且在其中告知VSCode this必须明确执行（不能是隐式的）；



```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "strictBindCallApply": true,
    "strictPropertyInitialization": true,
    "noImplicitThis": true,
    "useUnknownInCatchVariables": true
  }
}
```

- 在设置了`noImplicitThis`为true时，TypeScript会根据上下文推导this，但是在不能正确推导时，就会报错，需要我们明确的指定this。

```
function foo1() {
  console.log(this)
}

foo1()
```

any

'this' implicitly has type 'any' because it does not have a type annotation. ts(2683)

指定this的类型

- 在开启noImplicitThis的情况下，我们必须指定this的类型。
- 如何指定呢？函数的第一个参数类型：
 - 函数的第一个参数我们可以根据该函数之后被调用的情况，用于声明this的类型（名词必须叫this）；
 - 在后续调用函数传入参数时，从第二个参数开始传递的，this参数会在编译后被抹除；

```
function foo1(this: { name: string }) {  
  console.log(this)  
}  
  
foo1.call({ name: "why" })
```

this相关的内置工具

■ **Typescript 提供了一些工具类型来辅助进行常见的类型转换，这些类型全局可用。**

■ **ThisParameterType:**

- 用于提取一个函数类型Type的this (opens new window)参数类型;
- 如果这个函数类型没有this参数返回unknown;

```
function foo(this: { name: string }) {  
  console.log(this.name)  
}  
  
// 获取一个函数中this的类型  
type ThisType = ThisParameterType<typeof foo>
```

■ **OmitThisParameter:**

- 用于移除一个函数类型Type的this参数类型, 并且返回当前的函数类型

```
// 用于移除一个函数类型Type的this参数类型, 并且返回当前的函数类型  
type FnType = OmitThisParameter<typeof foo>
```


this相关的内置工具 - ThisType

- 这个类型不返回一个转换过的类型，它被用作标记一个上下文的this类型。（官方文档）

- 事实上官方文档的不管是解释，还是案例都没有说明出来ThisType类型的作用；

- 我这里用另外一个例子来给大家进行说明：

```
interface IState {  
  name: string  
  age: number  
}  
  
interface IData {  
  state: IState  
  running: () => void  
  eating: () => void  
}
```

```
const info: IData & ThisType<IState> = {  
  state: { name: "why", age: 18 },  
  running: function() {  
    console.log(this.name)  
  },  
  eating: function() {  
  }  
}  
  
info.running.call(info.state)
```