【TypeScript 4.5】007-第 7 章 类型操纵

一、从类型中创建类型

1、概述

我们可以通过各种**类型操作符**

用一种简洁的、可维护的方式表达复杂的**操作和值**

2、方法

泛型类型、keyof 类型操作符、typeof 类型操作符、索引访问类型、条件类型、映射类型、模板字面量类型

二、泛型-HelloWorld

1、概述

软件工程的一个主要部分是建立**组件**

它们不仅有定义明确和一致的 api

还可以**重复使用**

这些组件为大型行项目提供**灵活**的能力

泛型是创建组件的重要工具

2、HelloWorld 演示

代码示例一

> 问题：只能返回 number 类型！

```
function identity(arg: number): number {
    return arg
}
```

代码示例二

> 问题：失去了使用类型的意义！

```
function identity(arg: any): any {
    return arg
}
```

代码示例三

> 完美，与代码示例一类型精确度一样！

```
function identity<T>(arg: T): T {
    return arg
}
```

两种调用泛型的方式

```
// 方式一：传入类型
let a = identity<string>("hello world")
// 方式二：自动推断
let b = identity("hello world")
```

三、使用通用类型变量

1、发现问题

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length) // 报错：类型“T”上不存在属性“length”。
    return arg;
}
```

2、解决问题

> 定义成数组

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

四、泛型-泛型类型

1、概述

说明

如何通过给一个变量设置这个函数的泛型类型

就需要使用泛型类型或泛型接口

代码示例

```
interface GenericIdentityFn {
    <T>(arg: T): T
}
```

2、代码演示

泛型类型

```
function identity<T>(arg: T): T {
    return arg
}

// 写法一
let a: <T>(arg: T) => T = identity
// 写法二
let b: { <T>(arg: T): T } = identity
```

泛型接口

```
function identity<T>(arg: T): T {
    return arg
}

interface GenericIdentityFn {
    <T>(arg: T): T
}
let c: GenericIdentityFn = identity
```

五、泛型-泛型类

1、概述

说明

一个泛型类的形状和泛型接口是类似的

泛型类就是在类的名称后面加 `<>`

写入泛型参数列表

代码示例

> 此处报错，我们可以将 `tscfig.json` 里面的 `strictPropertyInitialization` 属性值设置为 `fasle` 以关闭该提示！

```
class GenericNumber<NumberType> {
    // 属性
    zeroValue: NumberType
    // 函数
    add: (x: NumberType, y: NumberType) => NumberType
}
```

2、代码演示

```
class GenericNumber<NumberType> {
    zeroValue: NumberType
    add: (x: NumberType, y: NumberType) => NumberType
}
// number
let myGeneric = new GenericNumber<number>()
myGeneric.zeroValue = 0
myGeneric.add = function(x, y) {
    return x + y
}
// string
let myGeneric1 = new GenericNumber<string>()
myGeneric1.zeroValue = "0"
myGeneric1.add = function(x, y) {
    return x + y
}
```

六、泛型-泛型约束

1、概述

说明

用户调用函数传入参数的时候就告诉其所传入参数必须具有的属性

代码示例

```
function loggingIdentity<Type extends Lengthwise>{}
```

2、代码演示

```
interface Lengthwise {
    length: number
}
function loggingIdentity<Type extends Lengthwise>(arg: Type): Type{
    console.log(arg.length)
    return arg
}
// 测试
loggingIdentity("zibo") // 4
loggingIdentity([1, 2, 3]) // 3
```

七、泛型-在泛型约束中使用类参数

1、概述

说明

我们可以声明一个受另一个类型参数约束的类型参数

代码示例

```
<Key extends keyof Type>
```

2、代码演示

```
function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) {
    return obj[key]
}
let x = {
    a: 1,
    b: 2,
    c: 3,
    d: 4
}

getProperty(x, 'a') // 此处传入的第二个参数 'a' 必须是 x 里面存在的 key
// getProperty(x, 'm') // 报错：类型“"m"”的参数不能赋给类型“"a" | "b" | "c" | "d"”的参数。
```

八、泛型-在泛型中使用类类型

1、概述

说明

在 TypeScript 中使用泛型来创建工厂函数的时候

有必要通过其构造函数引用类的类型

代码示例

> 注意体会这个写法！是要传入一个类的，而不是类的实例！

```
function create<Type>(c: { new(): Type }): Type {
    return new c()
}
```

2、代码演示

```
class Beekeeper {
    hasMask: boolean = true
}

class Zookeeper {
    nametag: string = "liubei"
}

class Animal {
    numLegs: number = 4
}

class Bee extends Animal {
    keeper: Beekeeper = new Beekeeper()
}

class Lion extends Animal {
    keeper: Zookeeper = new Zookeeper()
}

function createInstance<A extends Animal>(c: new() => A): A {
    return new c()
}

createInstance(Lion).keeper.nametag
createInstance(Bee).keeper.hasMask
// 报错:
// 类型“typeof Beekeeper”的参数不能赋给类型“new () => Animal”的参数。
// 类型 "Beekeeper" 中缺少属性 "numLegs"，但类型 "Animal" 中需要该属性。
// createInstance(Beekeeper)
```

九、keyof 类型操作符

1、概述

说明

keyof 运算符可以接收一个对象类型

它会产生它的 key 的字符串

或者与数字字面量的结合

或者可以说是一个联合类型

代码示例

```
type Point = { x: number, y: number}
type P = keyof Point
// P 是 "x" | "y"
const p1: P = "x"
const p2: P = "y"
```

2、代码演示

普通示例

```
type Point = { x: number, y: number}
type P = keyof Point
// P 是 "x" | "y"
const p1: P = "x"
const p2: P = "y"
const p3:P = "z" // 报错：不能将类型“"z"”分配给类型“keyof Point”。
```

索引签名

```
// number 类型
type Arrayish = {
    [n: number]: unknown
}
type A = keyof Arrayish
// 此时为任意数字
const a01: A = 0
const a02: A = 2
const a03: A = 19

// string | numer 类型
type Mapish = {
    [k: string]: boolean
}
type M = keyof Mapish
const m01: M = 0
const m02: M = "hello"
const m03: M = false // 报错：不能将类型“boolean”分配给类型“string | number”。
```

十、typeof 类型操作符

1、概述

说明

typeof 可以让我们在类型的上下文中使用它来引用一个变量或属性的类型

代码示例

```
let s = "hello"
// 使得 n 的类型为 s 的类型
let n: typeof s
// 打印的时候结果为：string
console.log(typeof s)
n = "world"
```

2、代码演示

普通例子

```
console.log(typeof "hello") // string
console.log(typeof 100) // number
console.log(typeof true) // boolean
console.log(typeof { name: 'zibo', age: 25 }) // object
console.log(typeof function go(){}) // function
```

ReturnType 例子

> ReturnType：Ts 预定义的类型，返回当前函数返回值的类型！

```
type Predicate = (x: unknown) => boolean
// ReturnType：Ts 预定义的类型，返回当前函数返回值的类型
type K = ReturnType<Predicate>
let k: K = true
// 报错：不能将类型“number”分配给类型“boolean”。
let k01: K = 100

function f() {
    return {
        x: 10,
        y: 3
    }
}
console.log(typeof f)
type P = ReturnType<typeof f>
// 报错：不能将类型“number”分配给类型“{ x: number; y: number; }”。ts(2322)
const p: P = 100
```

十一、索引访问类型

1、概述

说明

我们可以使用索引访问类型访问另一个类型上的**特定属性**

代码示例

```
type Person = {
    age: number,
    name: string,
    alive: boolean
}
type Age = Person["age"]
```

2、代码演示

基本示例

```
type Person = {
    age: number,
    name: string,
    alive: boolean
}
type Age = Person["age"]
let age: Age = 100
// 报错：不能将类型“string”分配给类型“number”。
// let age01: Age = "100"
```

联合类型

```
// Person 省略，见上面！
type Age = Person["age" | 'name']
let age: Age = 100
let age01: Age = "100"
// 报错：不能将类型“boolean”分配给类型“Age”。
let age02: Age = true
```

使用 keyof

```
// Person 省略，见上面！
// 此处 Age 的类型是 number | string | boolean
type Age = Person[keyof Person]
let age: Age = 100
let age01: Age = "100"
let age02: Age = true
```

使用联合文字类型

```
// Person 省略，见上面!
type myStr = "age" | "name"
// 此处 Age 的类型是 number | string
type Age = Person[myStr]
let age: Age = 100
let age01: Age = "100"
// 报错：不能将类型“boolean”分配给类型“Age”。
let age02: Age = true
```

从数组里面获得类型

```
const MyArray = [
    { name: '大哥刘备', age: 35 },
    { name: '二哥关羽', age: 33 },
    { name: '三哥张飞', age: 31 }
]
type Person = typeof MyArray[number]
let p: Person = {
    name: '訾博',
    age: 25
}
type P1 = Person['name']
let myName: P1 = "zibo"
```

另一种写法

```
const key = 'name'
// 此写法报错：
// 类型“key”不能作为索引类型使用。ts(2538)
// “key”表示值，但在此处用作类型。是否指“类型 key”?
type P2 = Person[key]
// 正确写法如下
type P3 = Person[typeof key]
let myName01: P3 = "name"
// 另一种写法
type key2 = 'name'
type P4 = Person[key2]
let myName02: P4 = "name"
```

十二、条件类型

1、概述

说明

所谓条件类型类似条件表达式（三目运算符）

代码示例

```
SomeType extends OtherType ? TrueType : FalseType
```

2、代码演示

基本使用

```
interface Animal {
    live(): void
}
interface Dog extends Animal {
    woof(): void
}
// type Example01 = number
type Example01 = Dog extends Animal ? number : string
```

优化函数重载

```
interface IdLabel {
    id: number
}
interface NameLabel {
    name: string
}
function createLabel(id: number): IdLabel
function createLabel(name: string): NameLabel
function createLabel(idOrName: number | string): IdLabel | NameLabel
function createLabel(idOrName: number | string): IdLabel | NameLabel {
    throw ''
}
// 写起来很麻烦，我们可以使用条件类型解决
// 下面通过【条件类型】实现
type IdOrName<T extends number | string> = T extends number ? IdLabel :
NameLabel
function createLabel01<T extends number | string>(idOrName: T): IdOrName<T> {
    throw ''
}
// NameLabel
let a01 = createLabel01('hello')
// IdLabel
let a02 = createLabel01(111)
// IdLabel | NameLabel
let a03 = createLabel01(Math.random() > 0.5 ? 100 : "111")
```

十三、条件类型约束

1、概述

说明

通常条件类型中的检查会给我们提供一些新的信息

就像我们使用类型守卫缩小范围一样

可以给我们一个**更具体的类型**

条件类型的真正分支将通过我们的检查类型进一步约束泛型

代码示例

```
type MessageOf<T> = T extends { message: unknowm } ? T['message'] : never
```

2、代码演示

```
// 报错：类型“"message"”无法用于索引类型“T”
// type MessageOf<T> = T['message']

// 改造
type MessageOf<T extends { message: unknown }> = T['message']

interface Email {
    message: string
}
type EmailMessageContents = MessageOf<Email>
let xO1: EmailMessageContents = "hello"

// 再升级
type MessageOf01<T> = T extends { message: unknown } ? T['message'] : never
interface Dog {
    name: string
}
type DogMessageContents = MessageOf01<Dog> // 不报错
let xO2: DogMessageContents = "1111" // 报错：不能将类型“string”分配给类型“never”。

// 再来一个示例代码
type Flatten<T> = T extends any[] ? T[number] : T
type myStr = Flatten<string[]>
type myNum = Flatten<number>
```

十四、在条件类型内推理

1、概述

说明

条件类型为我们提供了一种方法

来推断我们在真实分支中

使用 infer 关键字来进行对比的类型

代码示例

> 我们用 infer 关键字定义一个 Item 类型!

```
type Flatten<T> = T extends Array<infer Item> ? Item : T
```

2、代码演示

基本示例

```
type GetReturnType<T> = T extends (...args: never[]) => infer R ? R : never
// number
 type Num = GetReturnType<() => number>
 let num: Num = 100
 // string
 type Str = GetReturnType<(x: string) => string>
 let str: Str = 'hello'
 // boolean[]
 type Bools = GetReturnType<(a: number, b: string, c: boolean) => Boolean[]>
 let bools: Bools = [true, false]
 // Never
 type Never = GetReturnType<string>
 let nev: Never = "never" // 报错：不能将类型“string”分配给类型“never”。
```

重载函数拓展

```
function stringOrNum(x: string): number
function stringOrNum(x: number): string
function stringOrNum(x: string | number): string | number
function stringOrNum(x: string | number): string | number {
    return Math.random() > 0.5 ? "hello" : 100
}

type T1 = ReturnType<typeof stringOrNum>
const t1: T1 = "ss"
const t2: T1 = 222
const t3: T1 = false // 报错：不能将类型“boolean”分配给类型“string | number”
```

十五、分布式条件类型

1、概述

什么

当条件类型作用于一个通用类型的时候

我们给定它一个联合类型

它就变成了一个分布式的了

代码示例

> 此时，返回的类型是 string[] | number[]

```
type ToArray<T> = T extends any ? T[] : never
type StrOrNumArr = ToArray<string | number>
```

2、代码演示

分布式

```
type ToArray<T> = T extends any ? T[] : never
type StrOrNumArr = ToArray<string | number>
// 此时，返回的类型是 string[] | number[]
let son: StrOrNumArr = ["111"]
let son01: StrOrNumArr = [111]
let son02: StrOrNumArr = [true] // 报错：不能将类型“boolean”分配给类型“string |
number”
```

非分布式

```
type ToArrayNonDist<T> = [T] extends [any] ? T[] : never
type StrArrOrNumArr = ToArrayNonDist<string | number>
// 此时，返回的类型是 string | number
let son03: StrArrOrNumArr = ["11"]
let son04: StrArrOrNumArr = [11]
let son05: StrArrOrNumArr = [true] // 报错：不能将类型“boolean”分配给类型“string |
number
```