TypeScript泛型编程

王红元 coderwhy

目录
content

认识泛型

- **软件工程的主要目的是构建不仅仅明确和一致的API，还要让你的代码具有很强的可重用性：**
  - 比如我们可以通过函数来封装一些API，通过传入不同的函数参数，让函数帮助我们完成不同的操作；
  - 但是对于参数的类型是否也可以参数化呢？
- **什么是类型的参数化？**
  - 我们来提一个需求：封装一个函数，传入一个参数，并且返回这个参数；
- **如果我们是TypeScript的思维方式，要考虑这个参数和返回值的类型需要一致：**

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

- **上面的代码虽然实现了，但是不适用于其他类型，比如string、boolean、Person等类型：**

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

泛型实现类型参数化

- **虽然any是可以的，但是定义为any的时候，我们其实已经丢失了类型信息：**
  - 比如我们传入的是一个number，那么我们希望返回的可不是any类型，而是number类型；
  - 所以，我们需要在函数中可以捕获到参数的类型是number，并且同时使用它来作为返回值的类型；
- **我们需要在这里使用一种特性的变量 - 类型变量（type variable），它作用于类型，而不是值：**

```
function foo<Type>(arg: Type): Type {
  return arg
}
```

- **这里我们可以使用两种方式来调用它：**
  - 方式一：通过 <类型> 的方式将类型传递给函数；
  - 方式二：通过类型推导（type argument inference），自动推到出我们传入变量的类型：
    - ✓ 在这里会推导出它们是 字面量类型的，因为字面量类型对于我们的函数也是适用的

```
foo<string>("abc")
foo<number>(123)
```

```
foo("abc")
foo(123)
```

泛型的基本补充

- 当然我们也可以传入多个类型：

```
function foo<T, E>(a1: T, a2: E) {

}
```

- 平时在开发中我们可能会看到一些常用的名称：
  - T：Type的缩写，类型
  - K、V：key和value的缩写，键值对
  - E：Element的缩写，元素
  - O：Object的缩写，对象

泛型接口

- 在定义接口的时候我们也可以使用泛型：

```ts
interface IFoo<T> {
  initialValue: T,
  valueList: T[],
  handleValue: (value: T) => void
}

const foo: IFoo<number> = {
  initialValue: 0,
  valueList: [0, 1, 3],
  handleValue: function(value: number) {
    console.log(value)
  }
}
```

```ts
interface IFoo<T = number> {
  initialValue: T,
  valueList: T[],
  handleValue: (value: T) => void
}
```

泛型类

- 我们也可以编写一个泛型类：

```
class Point<T> {
  x: T
  y: T

  constructor(x: T, y: T) {
    this.x = x
    this.y = y
  }
}

const p1 = new Point(10, 20)
const p2 = new Point<number>(10, 20)
const p3: Point<number> = new Point(10, 20)
```

泛型约束（Generic Constraints）

- **有时候我们希望传入的类型有某些共性，但是这些共性可能不是在同一种类型中：**
  - 比如string和array都是有length的，或者某些对象也是会有length属性的；
  - 那么只要是拥有length的属性都可以作为我们的参数类型，那么应该如何操作呢？

```ts
interface ILength {
  length: number
}

function getLength<T extends ILength>(args: T) {
  return args.length
}

console.log(getLength("abc"))
console.log(getLength(["abc", "cba"]))
console.log(getLength({length: 100, name: "why"}))
```

- **这里表示是传入的类型必须有这个属性，也可以有其他属性，但是必须至少有这个成员。**

泛型约束（Generic Constraints）

- **在泛型约束中使用类型参数（Using Type Parameters in Generic Constraints）**
  - 你可以声明一个类型参数，这个类型参数被其他类型参数约束；
- **举个栗子：我们希望获取一个对象给定属性名的值**
  - 我们需要确保我们不会获取 obj 上不存在的属性；
  - 所以我们在两个类型之间建立一个约束；

```
function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) {
  return obj[key]
}

const info = {
  name: "why",
  age: 18
}

console.log(getProperty(info, "name"))
```

映射类型（Mapped Types）

- **有的时候，一个类型需要基于另外一个类型，但是你又不想拷贝一份，这个时候可以考虑使用映射类型。**
  - 大部分内置的工具都是通过映射类型来实现的；
  - 大多数类型体操的题目也是通过映射类型完成的；
- **映射类型建立在索引签名的语法上：**
  - 映射类型，就是使用了 PropertyKeys 联合类型的泛型；
  - 其中 PropertyKeys 多是通过 keyof 创建，然后循环遍历键名创建一个类型；

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

type MapType<Type> = {
  [property in keyof Type]: boolean
}

type NewPerson = MapType<IPerson>
```

映射修饰符（Mapping Modifiers）

- **在使用映射类型时，有两个额外的修饰符可能会用到：**
  - 一个是 readonly，用于设置属性只读；
  - 一个是？，用于设置属性可选；
- **你可以通过前缀 - 或者 + 删除或者添加这些修饰符，如果没有写前缀，相当于使用了 + 前缀。**

```ts
type MapType<Type> = {
  [property in keyof Type]-?: Type[property]
}

interface IPerson {
  name: string
  age: number
  height: number
}

type NewPerson = MapType<IPerson>
```

内置工具和类型体操

- **类型系统其实在很多语言里面都是有的，比如Java、Swift、C++等等，但是相对来说TypeScript的类型非常灵活：**
  - 这是因为TypeScript的目的是为JavaScript添加一套类型校验系统，因为JavaScript本身的灵活性，也让TypeScript类型系统不得不增加更附加的功能以适配JavaScript的灵活性；
  - 所以TypeScript是一种可以支持类型编程的类型系统；
- **这种类型编程系统为TypeScript增加了很大的灵活度，同时也增加了它的难度：**
  - 如果你不仅仅在开发业务的时候为自己的JavaScript代码增加上类型约束，那么基本不需要太多的类型编程能力；
  - 但是如果你在开发一些框架、库，或者通用性的工具，为了考虑各种适配的情况，就需要使用类型编程；
- **TypeScript本身为我们提供了类型工具，帮助我们辅助进行类型转换（前面有用过关于this的类型工具）。**
- **很多开发者为了进一步增强自己的TypeScript编程能力，还会专门去做一些类型体操的题目：**
  - https://github.com/type-challenges/type-challenges
  - https://ghaiklor.github.io/type-challenges-solutions/en/
- **我们课堂上会学习TypeScript的编程能力的语法，并且通过学习内置工具来练习一些类型体操的题目。**

条件类型（Conditional Types）

- **很多时候，日常开发中我们需要基于输入的值来决定输出的值，同样我们也需要基于输入的值的类型来决定输出的值的类型。**
- **条件类型（Conditional types）**就是用来帮助我们描述输入类型和输出类型之间的关系。
  - 条件类型的写法有点类似于 JavaScript 中的条件表达式（condition ? trueExpression : falseExpression ）：

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

```ts
function sum<T extends number | string>(arg1: T, arg2: T): T extends string ? string: number
function sum(arg1: any, arg2: any) {
  return arg1 + arg2
}

const res1 = sum(10, 20)
const res2 = sum("aaa", "bbb")
```

在条件类型中推断（inter)

- **在条件类型中推断（Inferring Within Conditional Types）**
  - 条件类型提供了 infer 关键词，可以从正在比较的类型中推断类型，然后在 true 分支里引用该推断结果；
- **比如我们现在有一个数组类型，想要获取到一个函数的参数类型和返回值类型：**

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

type HYReturnType<T extends (...args: any[]) => any> = T extends (...args: any[]) => infer R? R: never
type CalcReturnType = HYReturnType<CalcFnType>

type HYParameters<T extends (...args: any[]) => any> = T extends (...args: infer P) => any? P: never
type CalcParameterType = HYParameters<CalcFnType>
```

分发条件类型（Distributive Conditional Types）

- 当在泛型中使用条件类型的时候，如果传入一个联合类型，就会变成 **分发的（distributive）**

```ts
type toArray<Type> = Type extends any ? Type[]: never

// string[] | number[]
type newType = toArray<number | string>
```

- 如果我们在 ToArray 传入一个联合类型，这个条件类型会被应用到联合类型的每个成员：
  - 当传入string | number时，会遍历联合类型中的每一个成员；
  - 相当于ToArray<string> | ToArray<number>；
  - 所以最后的结果是：string[] | number[]；

Partial<Type>

- 用于构造一个Type下面的所有属性都设置为可选的类型

```ts
// 自定义自己的Partial类型
type MyPartial<T> = {
  [K in keyof T]?: T[K]
}

interface IPerson {
  name: string
  age: number
  height: number
}

const info: IPerson = {
  name: 'why',
  age: 18,
  height: 1.88
}

function updatePerson(person: IPerson, partPerson: MyPartial<IPerson>) {
  return { ...person, ...partPerson }
}
```

Required<Type>

- 用于构造一个Type下面的所有属性全都设置为必填的类型，这个工具类型跟 Partial 相反。

```ts
type HYRequired<T> = {
  [K in keyof T]-?: T[K]
}

interface IPerson {
  name: string
  age?: number
  height?: number
}

type IPersonRequired = HYRequired<IPerson>

const info: IPersonRequired = {
  name: "why",
  age: 18,
  height: 1.88
}
```

Readonly<Type>

- 用于构造一个Type下面的所有属性全都设置为只读的类型，意味着这个类型的所有的属性全都不可以重新赋值。

```ts
type HYReadonly<T> = {
  readonly [K in keyof T]: T[K]
}

interface IPerson {
  name: string
  age: number
}

const info: HYReadonly<IPerson> = {
  name: "why",
  age: 18
}

const obj: IPerson = {
  name: "kobe",
  age: 30
}
```

Record<Keys, Type>

- 用于构造一个对象类型，它所有的key(键)都是Keys类型，它所有的value(值)都是Type类型。

```ts
type HYRecord<K extends keyof any, T> = {
  [P in K]: T
}

interface IPerson {
  name: string
  age: number
}

const p1: IPerson = { name: "why", age: 18 }
const p2: IPerson = { name: "kobe", age: 30 }

type CityType = "上海" | "洛杉矶"

const data: HYRecord<CityType, IPerson> = {
  "上海": p1,
  "洛杉矶": p2
}
```

Pick<Type, Keys>

- 用于构造一个类型，它是从Type类型里面挑了一些属性Keys

```ts
type HYPick<T, K extends keyof T> = {
  [P in K]: T[P]
}

interface IPerson {
  name: string
  age: number
  height: number
}

type IKun = Pick<IPerson, "name" | "age">
```

Omit<Type, Keys>

- 用于构造一个类型，它是从Type类型里面过滤了一些属性Keys

```ts
// 如果每次都pick可能类型太多了
type HYOmit<T, K> = {
  [P in keyof T as P extends K? never: P]: T[P]
}

interface IPerson {
  name: string
  age: number
  height: number
}

type IKun = Omit<IPerson, "height">
```

Exclude<UnionType, ExcludedMembers>

- 用于构造一个类型，它是从UnionType联合类型里面排除了所有可以赋给ExcludedMembers的类型。

```ts
// Exclude<UnionType, ExcludedMembers>
// Constructs a type by excluding from UnionType all union members that are assignable to
ExcludedMembers

type HYExclude<T, U> = T extends U? never: T
type HYOmit<T, K> = Pick<T, HYExclude<keyof T, K>>

type PropertyTypes = "name" | "age" | "height"
type PropertyTypes2 = HYExclude<PropertyTypes, "height">
```

- 有了HYExclude，我们可以使用它来实现HYOmit。

Extract<Type, Union>

- 用于构造一个类型，它是从Type类型里面提取了所有可以赋给Union的类型。

```ts
// Extract<Type, Union>
// Constructs a type by extracting from Type all union members that are assignable to Union.

type HYExtract<T, U> = T extends U? T: never

type PropertyTypes = "name" | "age" | "height"
type PropertyTypes2 = HYExtract<PropertyTypes, "name"|"age">
```

NonNullable<Type>

- 用于构造一个类型，这个类型从Type中排除了所有的null、undefined的类型。

```
// Constructs a type by excluding null and undefined from Type.
type HYNonNullable<T> = T extends undefined | null? never: T

type unionType= string | number | undefined | null
type unionType2 = HYNonNullable<unionType>
```

ReturnType<Type>

- 用于构造一个含有Type函数的返回值的类型。

```ts
// Constructs a type consisting of the return type of function Type.
// 第一个extends是对传入的条件进行限制
// 第二个extends是为了进行条件获取类型
type HYReturnType<T extends (...args: any) => any> = T extends (...args: any) => infer R? R: never

type T1 = HYReturnType<() => string>
type T2 = HYReturnType<() => void>
type T3 = HYReturnType<(num1: number, num2: number) => string>

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

function getInfo(info: {name: string, age: number}) {
  return info.name + info.age
}

type T4 = HYReturnType<typeof sum>
type T5 = HYReturnType<typeof getInfo>
type T6 = HYReturnType<() => void>
```

InstanceType<Type>

- 用于构造一个由所有Type的构造函数的实例类型组成的类型。

```ts
type HYInstanceType<T extends new (...args: any[]) => any> = T extends new (...args:any[]) => infer R? R: never

class Person {
  name: string
  age: number
  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }
}

type T = typeof Person
type PersonType = InstanceType<typeof Person>

// 对于普通的定义来说似乎是没有区别的
const info: Person = { name: "why", age: 18 }
const into2: PersonType = { name: "kobe", age: 30 }

// 但是如果我们想要做一个工厂函数，用于帮助我们创建某种类型的对象
// 这里的返回值不可以写T，因为T的类型会是typeof Person
// 这里就可以使用InstanceType<T>，它可以帮助我们返回构造函数的返回值类型(构造函数创建出来的对象类型)
function factory<T extends new (...args: any[]) => any>(ctor: T): HYInstanceType<T> {
  return new ctor()
}

const p1 = factory(Person)
```