TypeScript 泛型

简介

有些时候，函数返回值的类型与参数类型是相关的。

```javascript
function getFirst(arr) {
  return arr[0];
}
```

上面示例中，函数 `getFirst()` 总是返回参数数组的第一个成员。参数数组是什么类型，返回值就是什么类型。

这个函数的类型声明只能写成下面这样。

```typescript
function f(arr: any[]): any {
  return arr[0];
}
```

上面的类型声明，就反映不出参数与返回值之间的类型关系。

为了解决这个问题，TypeScript 就引入了“泛型”（generics）。泛型的特点就是带有“类型参数”（type parameter）。

```typescript
function getFirst<T>(arr: T[]): T {
  return arr[0];
}
```

上面示例中，函数 `getFirst()` 的函数名后面尖括号的部分 `<T>`，就是类型参数，参数要放在一对尖括号（`<>`）里面。本例只有一个类型参数 `T`，可以将其理解为类型声明需要的变量，需

要在调用时传入具体的参数类型。

上例的函数 `getFirst()` 的参数类型是 `T[]` ，返回值类型是 `T` ，就清楚地表示了两者之间的关系。比如，输入的参数类型是 `number[]` ，那么 T 的值就是 `number` ，因此返回值类型也是 `number` 。

函数调用时，需要提供类型参数。

```typescript
getFirst<number>([1, 2, 3]);
```

上面示例中，调用函数 `getFirst()` 时，需要在函数名后面使用尖括号，给出类型参数 `T` 的值，本例是 `<number>` 。

不过为了方便，函数调用时，往往省略不写类型参数的值，让 TypeScript 自己推断。

```typescript
getFirst([1, 2, 3]);
```

上面示例中，TypeScript 会从实际参数 `[1, 2, 3]` ，推断出类型参数 T 的值为 `number` 。

有些复杂的使用场景，TypeScript 可能推断不出类型参数的值，这时就必须显式给出了。

```typescript
function comb<T>(arr1: T[], arr2: T[]): T[] {
  return arr1.concat(arr2);
}
```

上面示例中，两个参数 `arr1` 、 `arr2` 和返回值都是同一个类型。如果不给出类型参数的值，下面的调用会报错。

```typescript
comb([1, 2], ["a", "b"]); // 报错
```

上面示例会报错，TypeScript 认为两个参数不是同一个类型。但是，如果类型参数是一个联合类型，就不会报错。

```typescript
comb<number | string>([1, 2], ["a", "b"]); // 正确
```

上面示例中，类型参数是一个联合类型，使得两个参数都符合类型参数，就不报错了。这种情况下，类型参数是不能省略不写的。

类型参数的名字，可以随便取，但是必须为合法的标识符。习惯上，类型参数的第一个字符往往采用大写字母。一般会使用 `T` （type 的第一个字母）作为类型参数的名字。如果有多个类型参数，则使用 T 后面的 U、V 等字母命名，各个参数之间使用逗号（“,”）分隔。

下面是多个类型参数的例子。

typescript

```
function map<T, U>(arr: T[], f: (arg: T) => U): U[] {
  return arr.map(f);
}

// 用法实例
map<string, number>(["1", "2", "3"], (n) => parseInt(n)); // 返回 [1, 2, 3]
```

上面示例将数组的实例方法 `map()` 改写成全局函数，它有两个类型参数 `T` 和 `U` 。含义是，原始数组的类型为 `T[]` ，对该数组的每个成员执行一个处理函数 `f` ，将类型 `T` 转成类型 `U` ，那么就会得到一个类型为 `U[]` 的数组。

总之，泛型可以理解成一段类型逻辑，需要类型参数来表达。有了类型参数以后，可以在输入类型与输出类型之间，建立一一对应关系。

泛型的写法

泛型主要用在四个场合：函数、接口、类和别名。

函数的泛型写法

上一节提到， `function` 关键字定义的泛型函数，类型参数放在尖括号中，写在函数名后面。

typescript

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

那么对于变量形式定义的函数，泛型有下面两种写法。

typescript

```
// 写法一
let myId: <T>(arg: T) => T = id;

// 写法二
let myId: { <T>(arg: T): T } = id;
```

接口的泛型写法

interface 也可以采用泛型的写法。

typescript

```
interface Box<Type> {
  contents: Type;
}

let box: Box<string>;
```

上面示例中，使用泛型接口时，需要给出类型参数的值（本例是 `string` ）。

下面是另一个例子。

typescript

```
interface Comparator<T> {
  compareTo(value: T): number;
}

class Rectangle implements Comparator<Rectangle> {
  compareTo(value: Rectangle): number {
    // ...
  }
}
```

上面示例中，先定义了一个泛型接口，然后将这个接口用于一个类。

泛型接口还有第二种写法。

typescript

```
interface Fn {
  <Type>(arg: Type): Type;
}

function id<Type>(arg: Type): Type {
```

```
  return arg;
}

let myId: Fn = id;
```

上面示例中， `Fn` 的类型参数 `Type` 的具体类型，需要函数 `id` 在使用时提供。所以，最后一行的赋值语句不需要给出 `Type` 的具体类型。

此外，第二种写法还有一个差异之处。那就是它的类型参数定义在某个方法之中，其他属性和方法不能使用该类型参数。前面的第一种写法，类型参数定义在整个接口，接口内部的所有属性和方法都可以使用该类型参数。

类的泛型写法

泛型类的类型参数写在类名后面。

typescript

```
class Pair<K, V> {
  key: K;
  value: V;
}
```

下面是继承泛型类的例子。

typescript

```
class A<T> {
  value: T;
}

class B extends A<any> {}
```

上面示例中，类 `A` 有一个类型参数 `T` ，使用时必须给出 `T` 的类型，所以类 `B` 继承时要写成 `A<any>` 。

泛型也可以用在类表达式。

typescript

```
const Container = class<T> {
  constructor(private readonly data: T) {}
};
```

```
const a = new Container<boolean>(true);
const b = new Container<number>(0);
```

上面示例中，新建实例时，需要同时给出类型参数 `T` 和类参数 `data` 的值。

下面是另一个例子。

typescript

```
class C<NumType> {
  value!: NumType;
  add!: (x: NumType, y: NumType) => NumType;
}

let foo = new C<number>();

foo.value = 0;
foo.add = function (x, y) {
  return x + y;
};
```

上面示例中，先新建类 `C` 的实例 `foo` ，然后再定义示例的 `value` 属性和 `add()` 方法。类的定义中，属性和方法后面的感叹号是非空断言，告诉 TypeScript 它们都是非空的，后面会赋值。

JavaScript 的类本质上是一个构造函数，因此也可以把泛型类写成构造函数。

typescript

```
type MyClass<T> = new (...args: any[]) => T;

// 或者
interface MyClass<T> {
  new (...args: any[]): T;
}

// 用法实例
function createInstance<T>(AnyClass: MyClass<T>, ...args: any[]): T {
  return new AnyClass(...args);
}
```

上面示例中，函数 `createInstance()` 的第一个参数 `AnyClass` 是构造函数（也可以是一个类），它的类型是 `MyClass<T>` ，这里的 `T` 是 `createInstance()` 的类型参数，在该函数调用时再指定具体类型。

注意，泛型类描述的是类的实例，不包括静态属性和静态方法，因为这两者定义在类的本身。因此，它们不能引用类型参数。

```typescript
class C<T> {
  static data: T; // 报错
  constructor(public value: T) {}
}
```

上面示例中，静态属性 `data` 引用了类型参数 `T` ，这是不可以的，因为类型参数只能用于实例属性和实例方法，所以报错了。

类型别名的泛型写法

type 命令定义的类型别名，也可以使用泛型。

```typescript
type Nullable<T> = T | undefined | null;
```

上面示例中， `Nullable<T>` 是一个泛型，只要传入一个类型，就可以得到这个类型与 `undefined` 和 `null` 的一个联合类型。

下面是另一个例子。

```typescript
type Container<T> = { value: T };

const a: Container<number> = { value: 0 };
const b: Container<string> = { value: "b" };
```

下面是定义树形结构的例子。

```typescript
type Tree<T> = {
  value: T;
  left: Tree<T> | null;
  right: Tree<T> | null;
};
```

上面示例中，类型别名 `Tree` 内部递归引用了 `Tree` 自身。

类型参数的默认值

类型参数可以设置默认值。使用时，如果没有给出类型参数的值，就会使用默认值。

typescript

```
function getFirst<T = string>(arr: T[]): T {
  return arr[0];
}
```

上面示例中， `T = string` 表示类型参数的默认值是 `string` 。调用 `getFirst()` 时，如果不给出 `T` 的值，TypeScript 就认为 `T` 等于 `string` 。

但是，因为 TypeScript 会从实际参数推断出 `T` 的值，从而覆盖掉默认值，所以下面的代码不会报错。

typescript

```
getFirst([1, 2, 3]); // 正确
```

上面示例中，实际参数是 `[1, 2, 3]` ，TypeScript 推断 T 等于 `number` ，从而覆盖掉默认值 `string` 。

类型参数的默认值，往往用在类中。

typescript

```
class Generic<T = string> {
  list: T[] = [];

  add(t: T) {
    this.list.push(t);
  }
}
```

上面示例中，类 `Generic` 有一个类型参数 `T` ，默认值为 `string` 。这意味着，属性 `list` 默认是一个字符串数组，方法 `add()` 的默认参数是一个字符串。

typescript

```
const g = new Generic();

g.add(4); // 报错
g.add("hello"); // 正确
```

上面示例中，新建 `Generic` 的实例 `g` 时，没有给出类型参数 `T` 的值，所以 `T` 就等于 `string` 。因此，向 `add()` 方法传入一个数值会报错，传入字符串就不会。

typescript

```typescript
const g = new Generic<number>();

g.add(4); // 正确
g.add("hello"); // 报错
```

上面示例中，新建实例 `g` 时，给出了类型参数 `T` 的值是 `number` ，因此 `add()` 方法传入数值不会报错，传入字符串会报错。

一旦类型参数有默认值，就表示它是可选参数。如果有多个类型参数，可选参数必须在必选参数之后。

typescript

```typescript
<T = boolean, U> // 错误

<T, U = boolean> // 正确
```

上面示例中，依次有两个类型参数 `T` 和 `U` 。如果 `T` 是可选参数， `U` 不是，就会报错。

数组的泛型表示

《数组》一章提到过，数组类型有一种表示方法是 `Array<T>` 。这就是泛型的写法， `Array` 是 TypeScript 原生的一个类型接口， `T` 是它的类型参数。声明数组时，需要提供 `T` 的值。

typescript

```typescript
let arr: Array<number> = [1, 2, 3];
```

上面的示例中， `Array<number>` 就是一个泛型，类型参数的值是 `number` ，表示该数组的全部成员都是数值。

同样的，如果数组成员都是字符串，那么类型就写成 `Array<string>` 。事实上，在 TypeScript 内部，数组类型的另一种写法 `number[]` 、 `string[]` ，只是 `Array<number>` 、 `Array<string>` 的简写形式。

在 TypeScript 内部， `Array` 是一个泛型接口，类型定义基本是下面的样子。

typescript

```
interface Array<Type> {
  length: number;

  pop(): Type | undefined;

  push(...items: Type[]): number;

  // ...
}
```

上面代码中， `push()` 方法的参数 `item` 的类型是 `Type[]` ，跟 `Array()` 的参数类型 `Type` 保持一致，表示只能添加同类型的成员。调用 `push()` 的时候，TypeScript 就会检查两者是否一致。

其他的 TypeScript 内部数据结构，比如 `Map` 、 `Set` 和 `Promise` ，其实也是泛型接口，完整的写法是 `Map<K, V>` 、 `Set<T>` 和 `Promise<T>` 。

TypeScript 默认还提供一个 `ReadonlyArray<T>` 接口，表示只读数组。

typescript

```
function doStuff(values: ReadonlyArray<string>) {
  values.push("hello!"); // 报错
}
```

上面示例中，参数 `values` 的类型是 `ReadonlyArray<string>` ，表示不能修改这个数组，所以函数体内部新增数组成员就会报错。因此，如果不希望函数内部改动参数数组，就可以将该参数数组声明为 `ReadonlyArray<T>` 类型。

类型参数的约束条件

很多类型参数并不是无限制的，对于传入的类型存在约束条件。

typescript

```
function comp<Type>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a;
  }
  return b;
}
```

上面示例中，类型参数 Type 有一个隐藏的约束条件：它必须存在 `length` 属性。如果不满足这个条件，就会报错。

TypeScript 提供了一种语法，允许在类型参数上面写明约束条件，如果不满足条件，编译时就会报错。这样也可以有良好的语义，对类型参数进行说明。

typescript

```typescript
function comp<T extends { length: number }>(a: T, b: T) {
  if (a.length >= b.length) {
    return a;
  }
  return b;
}
```

上面示例中， `T extends { length: number }` 就是约束条件，表示类型参数 T 必须满足 `{ length: number }` ，否则就会报错。

typescript

```typescript
comp([1, 2], [1, 2, 3]); // 正确
comp("ab", "abc"); // 正确
comp(1, 2); // 报错
```

上面示例中，只要传入的参数类型不满足约束条件，就会报错。

类型参数的约束条件采用下面的形式。

typescript

```typescript
<TypeParameter extends ConstraintType>
```

上面语法中， `TypeParameter` 表示类型参数， `extends` 是关键字，这是必须的， `ConstraintType` 表示类型参数要满足的条件，即类型参数应该是 `ConstraintType` 的子类型。

类型参数可以同时设置约束条件和默认值，前提是默认值必须满足约束条件。

typescript

```typescript
type Fn<A extends string, B extends string = "world"> = [A, B];

type Result = Fn<"hello">; // ["hello", "world"]
```

上面示例中，类型参数 `A` 和 `B` 都有约束条件，并且 `B` 还有默认值。所以，调用 `Fn` 的时候，可以只给出 `A` 的值，不给出 `B` 的值。

另外，上例也可以看出，泛型本质上是一个类型函数，通过输入参数，获得结果，两者是一一对应关系。

如果有多个类型参数，一个类型参数的约束条件，可以引用其他参数。

```typescript
<T, U extends T>
// 或者
<T extends U, U>
```

上面示例中， `U` 的约束条件引用 `T` ，或者 `T` 的约束条件引用 `U` ，都是正确的。

但是，约束条件不能引用类型参数自身。

```typescript
<T extends T>              // 报错
<T extends U, U extends T>  // 报错
```

上面示例中， `T` 的约束条件不能是 `T` 自身。同理，多个类型参数也不能互相约束（即 `T` 的约束条件是 `U` 、 `U` 的约束条件是 `T` ），因为互相约束就意味着约束条件就是类型参数自身。

使用注意点

泛型有一些使用注意点。

（1）尽量少用泛型。

泛型虽然灵活，但是会加大代码的复杂性，使其变得难读难写。一般来说，只要使用了泛型，类型声明通常都不太易读，容易写得很复杂。因此，可以不用泛型就不要用。

（2）类型参数越少越好。

多一个类型参数，多一道替换步骤，加大复杂性。因此，类型参数越少越好。

```typescript
function filter<T, Fn extends (arg: T) => boolean>(arr: T[], func: Fn): T[] {
  return arr.filter(func);
}
```

上面示例有两个类型参数，但是第二个类型参数 `Fn` 是不必要的，完全可以直接写在函数参数的类型声明里面。

typescript

```
function filter<T>(arr: T[], func: (arg: T) => boolean): T[] {
  return arr.filter(func);
}
```

上面示例中，类型参数简化成了一个，效果与前一个示例是一样的。

（3）类型参数需要出现两次。

如果类型参数在定义后只出现一次，那么很可能是不必要的。

typescript

```
function greet<Str extends string>(s: Str) {
  console.log("Hello, " + s);
}
```

上面示例中，类型参数 `Str` 只在函数声明中出现一次（除了它的定义部分），这往往表明这个类型参数是不必要。

typescript

```
function greet(s: string) {
  console.log("Hello, " + s);
}
```

上面示例把前面的类型参数省略了，效果与前一个示例是一样的。

也就是说，只有当类型参数用到两次或两次以上，才是泛型的适用场合。

（4）泛型可以嵌套。

类型参数可以是另一个泛型。

typescript

```
type OrNull<Type> = Type | null;

type OneOrMany<Type> = Type | Type[];

type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>;
```

上面示例中，最后一行的泛型 `OrNull` 的类型参数，就是另一个泛型 `OneOrMany` 。

最后更新: 2023/8/13 15:25

Previous page
类

Next page
Enum 类型