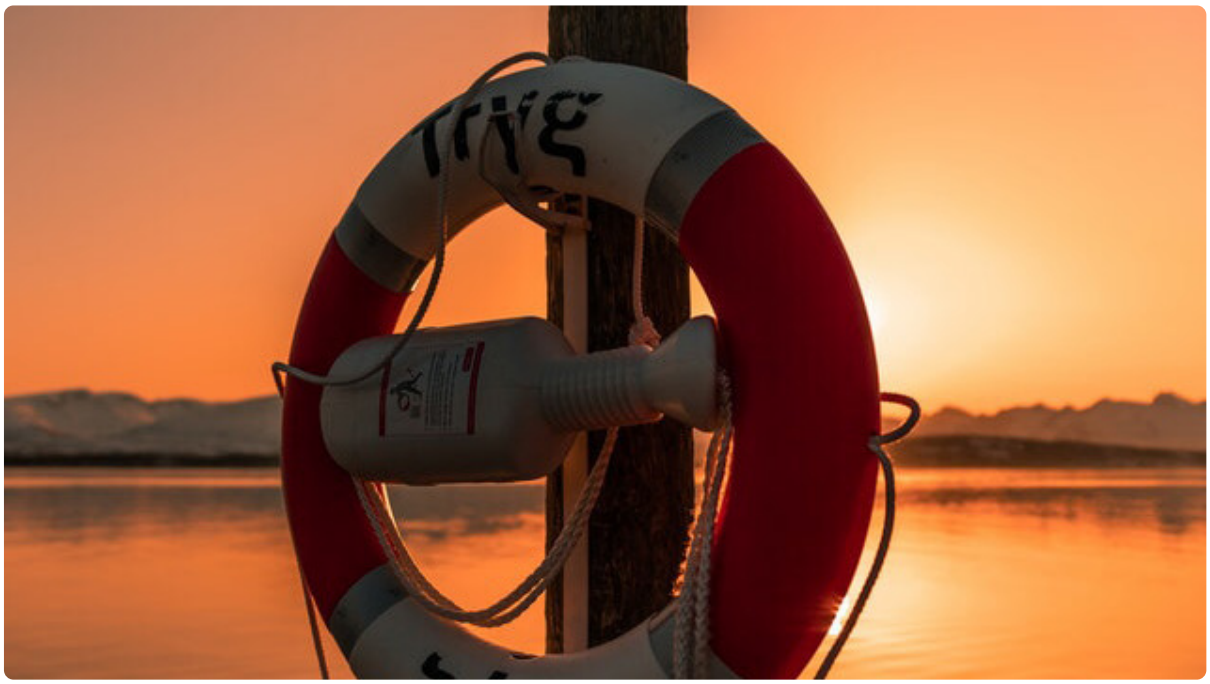
12 使用泛型拯救你的any

更新时间：2019-06-12 16:37:49



> 人的一生可能燃烧也可能腐朽，我不能腐朽，我愿意燃烧起来！
>
> ——奥斯特洛夫斯基

在前面的小节中我们学习了any类型，当我们要表示一个值可以为任意类型的时候，则指定它的类型为any，比如下面这个例子：

```
const getArray = (value: any, times: number = 5): any[] => {
  return new Array(times).fill(value);
};
```

这个函数接受两个参数。第一个参数为任意类型的值，第二个参数为数值类型的值，默认为 5。函数的功能是返回一个以 times 为元素个数，每个元素都是 value 的数组。这个函数我们从逻辑上可以知道，传入的 value 是什么类型，那么返回的数组的每个元素也应该是什么类型。

接下来我们实际用一下这个函数：

```
getArray([1], 2).forEach(item => {
  console.log(item.length);
});
getArray(2, 3).forEach(item => {
  console.log(item.length);
});
```

我们调用了两次这个方法，使用 forEach 方法遍历得到的数组，在传入 forEach 的函数中获取当前遍历到的数组元素的 length 属性。第一次调用这个方法是没问题的，因为我们第一次传入的值为数组，得到的会是一个二维数组 `[ [1], [1] ]`。每次遍历的元素为 `[1]`，它也是数组，所以打印它的 length 属性是可以的。而我们第二次传入的是一个数字 2，生成的数组是 `[2, 2, 2]`，访问 2 的 length 属性是没有的，所以应该报错，但是这里却不会报错，因为我们在定义 `getArray` 函数的时候，指定了返回值是 `any` 类型的元素组成的数组，所以这里遍历其返回值中每一个元素的时候，类型都是any，所以不管做任何操作都是可以的，因此，上面例子中第二次调用 `getArray` 的返回值每个元素应该是数值类型，遍历这个数组时我们获取数值类型的length属性也没报错，因为这里item的类型是any。

所以要解决这种情况，泛型就可以搞定，接下来我们来学习泛型。

2.9.1. 简单使用

要解决上面这个场景的问题，就需要使用泛型了。泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

还拿上面这个例子中的逻辑来举例，我们既要允许传入任意类型的值，又要正确指定返回值类型，就要使用泛型。我们先来看怎么改写：

```
const getArray = <T>(value: T, times: number = 5): T[] => {
  return new Array(times).fill(value);
};
```

我们在定义函数之前，使用 `<>` 符号定义了一个泛型变量 T，这个 T 在这次函数定义中就代表某一种类型，它可以是基础类型，也可以是联合类型等高级类型。定义了泛型变量之后，你在函数中任何需要指定类型的地方使用 T 都代表这一种类型。比如当我们传入 value 的类型为数值类型，那么返回的数组类型 `T[]` 就表示 `number[]`。现在我们再来调用一下这个 getArray 函数：

```
getArray<number[]>([1, 2], 3).forEach(item => {
  console.log(item.length);
});
getArray<number>(2, 3).forEach(item => {
  console.log(item.length); // 类型“number”上不存在属性“length”
});
```

我们在调用 getArray 的时候，在方法名后面使用 `<>` 传入了我们的泛型变量 T 的类型 `number[]`，那么在定义 getArray 函数时使用 T 指定类型的地方，都会使用 `number[]` 指定。但是你也可以省略这个 `<number[]>`，TypeScript 会根据你传入函数的 value 值的类型进行推断：

```
getArray(2, 3).forEach(item => {
  console.log(item.length); // 类型“number”上不存在属性“length”
});
```

2.9.2. 泛型变量

当我们使用泛型的时候，你必须在处理类型涉及到泛型的数据的时候，把这个数据当做任意类型来处理。这就意味着不是所有类型都能做的操作不能做，不是所有类型都能调用的方法不能调用。可能会有点绕口，我们来看个例子：

```
const getLength = <T>(param: T): number => {
  return param.length; // error 类型“T”上不存在属性“length”
};
```

当我们获取一个类型为泛型的变量 param 的 length 属性值时，如果 param 的类型为数组 Array 或字符串 string 类型是没问题的，它们有 length 属性。但是如果此时传入的 param 是数值 number 类型，那这里就会有问题了。

这里的`T`并不是固定的，你可以写为`A`、`B`或者其他名字，而且还可以在一个函数中定义多个泛型变量。我们来看个复杂点的例子：

```
const getArray = <T, U>(param1: T, param2: U, times: number): [T, U][] => {
  return new Array(times).fill([param1, param2]);
};
getArray(1, "a", 3).forEach(item => {
  console.log(item[0].length); // error 类型“number”上不存在属性“length”
  console.log(item[1].toFixed(2)); // error 属性“toFixed”在类型“string”上不存在
});
```

这个例子中，我们定义了两个泛型变量`T`和`U`。第一个参数的类型为 T，第二个参数的类型为 U，最后函数返回一个二维数组，函数返回类型我们指定是一个元素类型为`[T, U]`的数组。所以当我们调用函数，最后遍历结果时，遍历到的每个元素都是一个第一个元素是数值类型、第二个元素是字符串类型的数组。

2.9.3. 泛型函数类型

我们可以定义一个泛型函数类型，还记得我们之前学习函数一节时，给一个函数定义函数类型，现在我们可以使用泛型定义函数类型：

```
// ex1: 简单定义
const getArray: <T>(arg: T, times: number) => T[] = (arg, times) => {
  return new Array(times).fill(arg);
};
// ex2: 使用类型别名
type GetArray = <T>(arg: T, times: number) => T[];
const getArray: GetArray = <T>(arg: T, times: number): T[] => {
  return new Array(times).fill(arg);
};
```

当然了，我们也可以使用接口的形式来定义泛型函数类型：

```
interface GetArray {
  <T>(arg: T, times: number): T[];
}
const getArray: GetArray = <T>(arg: T, times: number): T[] => {
  return new Array(times).fill(arg);
};
```

你还可以把接口中泛型变量提升到接口最外层，这样接口中所有属性和方法都能使用这个泛型变量了。我们先来看怎么用：

```
interface GetArray<T> {
  (arg: T, times: number): T[];
  tag: T;
}
const getArray: GetArray<number> = <T>(arg: T, times: number): T[] => {
  // error 不能将类型“{ <T>(arg: T, times: number): T[]; tag: string; }”分配给类型“GetArray<number>”。
  // 属性“tag”的类型不兼容。
  return new Array(times).fill(arg);
};
getArray.tag = "a"; // 不能将类型“"a"”分配给类型“number”
getArray("a", 1); // 不能将类型“"a"”分配给类型“number”
```

上面例子中将泛型变量定义在接口最外层，所以不仅函数的类型中可以使用 T，在属性 tag 的定义中也可以使用。但在使用接口的时候，要在接口名后面明确传入一个类型，也就是这里的`GetArray<number>`，那么后面的 arg 和 tag 的类型都得是 number 类型。当然了，如果你还是希望 T 可以是任何类型，你可以把`GetArray<number>`换成`GetArray<any>`。

2.9.4 泛型约束

当我们使用了泛型时，就意味着这个这个类型是任意类型。但在大多数情况下，我们的逻辑是对特定类型处理的。还记得我们前面讲泛型变量时举的那个例子——当访问一个泛型类型的参数的 length 属性时，会报错"类型“T”上不存在属性“length”"，是因为并不是所有类型都有 length 属性。

所以我们在这里应该对 T 有要求，那就是类型为 T 的值应该包含 length 属性。说到这个需求，你应该能想到接口的使用，我们可以使用接口定义一个对象必须有哪些属性：

```
interface ValueWithLength {
  length: number;
}
const v: ValueWithLength = {}; // error Property 'length' is missing in type '{}' but required in type 'ValueWithLength'
```

泛型约束就是使用一个类型和 extends 对泛型进行约束，之前的例子就可以改为下面这样：

```
interface ValueWithLength {
  length: number;
}
const getLength = <T extends ValueWithLength>(param: T): number => {
  return param.length;
};
getLength("abc"); // 3
getLength([1, 2, 3]); // 3
getLength({ length: 3 }); // 3
getLength(123); // error 类型“123”的参数不能赋给类型“ValueWithLength”的参数
```

这个例子中，泛型变量T受到约束。它必须满足接口*ValueWithLength*，也就是不管它是什么类型，但必须有一个length属性，且类型为数值类型。例子中后面四次调用getLength方法，传入了不同的值，传入字符串 "abc"、数组 [1, 2, 3] 和一个包含length属性的对象 { length: 3 } 都是可以的，但是传入数值 123 不行，因为它没有length属性。

2.9.4 在泛型约束中使用类型参数

当我们定义一个对象，想要对只能访问对象上存在的属性做要求时，该怎么办？先来看下这个需求是什么样子：

```
const getProps = (object, propName) => {
  return object[propName];
};
const obj = { a: "aa", b: "bb" };
getProps(obj, "c"); // undefined
```

当我们访问这个对象的'c'属性时，这个属性是没有的。这里我们需要用到索引类型 keyof 结合泛型来实现对这个问题的检查。索引类型在高级类型一节会详细讲解，这里你只要知道这个例子就可以了：

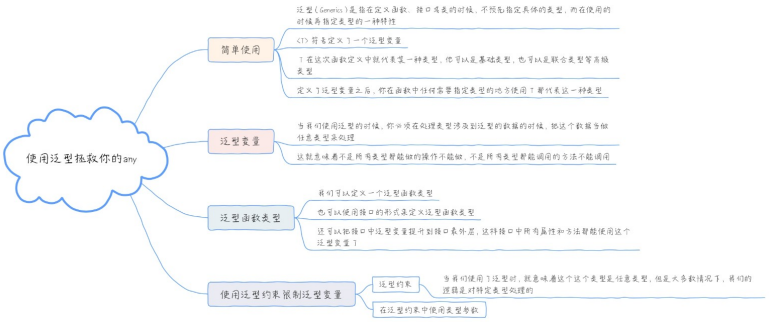
```
const getProp = <T, K extends keyof T>(object: T, propName: K) => {
  return object[propName];
};
const obj = { a: "aa", b: "bb" };
getProp(obj, "c"); // 类型“"c"”的参数不能赋给类型“"a" | "b"”的参数
```

这里我们使用让 K 来继承索引类型 keyof T，你可以理解为 keyof T 相当于一个由泛型变量T的属性名构成的联合类型，在这里 K 就被约束为了只能是"a"或"b"，所以当我们传入字符串"c"想要获取对象*obj*的属性"c"时就会报错。

小结

本小节我们学习了泛型的相关知识；学习了使用泛型来弥补使用any造成的类型信息缺失；当我们的类型是灵活任意的，又要准确使用类型信息时，就需要使用泛型来关联类型信息，其中离不开的是泛型变量；泛型变量可以是多个，且命名随意；如果需要对泛型变量的类型做进一步的限制，则需要用到我们最后讲的泛型约束；使用泛型约束通过 extends 关键字指定要符合的类型，从而满足更多场景的需求。

下个小节我们将学习类的知识，学习TypeScript中的类的知识之前，你需要先详细学习ES6标准中新增的类的知识，建议你先学习下阮一峰的《ECMAScript 6 入门》中类的部分。之所以要学习ES6中的类，是因为TypeScript中类的语法基本上是遵循ES6标准的，但是有一些区别，我们会在下个小节学习。



11 为函数和函数参数定义类型

13 TS中的类，小心它与ES标准的差异