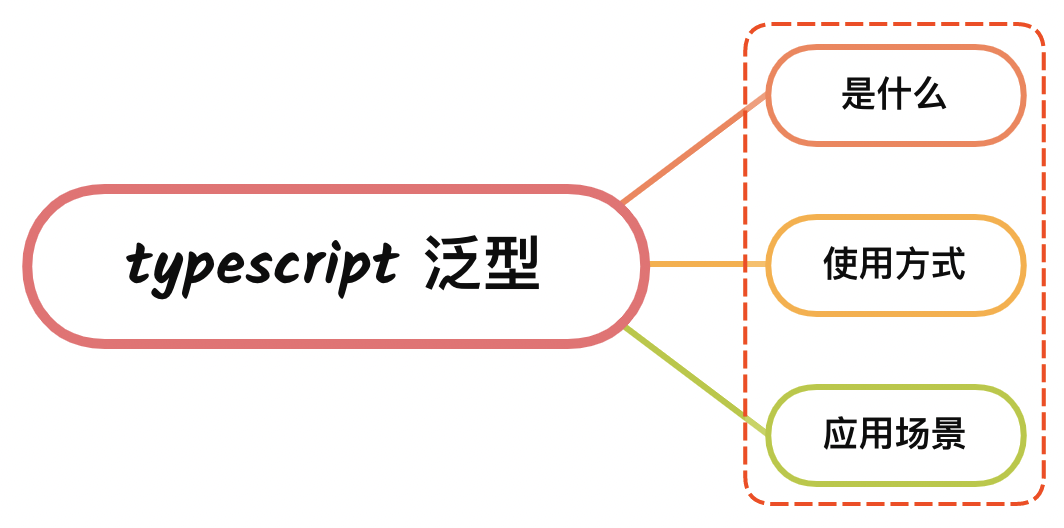
# 面试官：说说你对 TypeScript 中泛型的理解？应用场景？



## 一、是什么

泛型程序设计（generic programming）是程序设计语言的一种风格或范式

泛型允许我们在强类型程序设计语言中编写代码时使用一些以后才指定的类型，在实例化时作为参数指明这些类型 在typescript中，定义函数，接口或者类的时候，不预先定义好具体的类型，而在使用的时候在指定类型的一种特性

假设我们用一个函数，它可接受一个 number 参数并返回一个number 参数，如下写法：

function returnItem (para: number): number {  
 return para  
}

如果我们打算接受一个 string 类型，然后再返回 string类型，则如下写法：

function returnItem (para: string): string {  
 return para  
}

上述两种编写方式，存在一个最明显的问题在于，代码重复度比较高

虽然可以使用 any类型去替代，但这也并不是很好的方案，因为我们的目的是接收什么类型的参数返回什么类型的参数，即在运行时传入参数我们才能确定类型

这种情况就可以使用泛型，如下所示：

function returnItem<T>(para: T): T {  
 return para  
}

可以看到，泛型给予开发者创造灵活、可重用代码的能力

## 二、使用方式

泛型通过<>的形式进行表述，可以声明：

* 函数
* 接口
* 类

### 函数声明

声明函数的形式如下：

function returnItem<T>(para: T): T {  
 return para  
}

定义泛型的时候，可以一次定义**多个类型参数**，比如我们可以同时定义泛型 T 和 泛型 U：

function swap<T, U>(tuple: [T, U]): [U, T] {  
 return [tuple[1], tuple[0]];  
}  
  
swap([7, 'seven']); // ['seven', 7]

### 接口声明

声明接口的形式如下：

interface ReturnItemFn<T> {  
 (para: T): T  
}

那么当我们想传入一个number作为参数的时候，就可以这样声明函数:

const returnItem: ReturnItemFn<number> = para => para

### 

### 类声明

使用泛型声明类的时候，既可以作用于类本身，也可以作用与类的成员函数

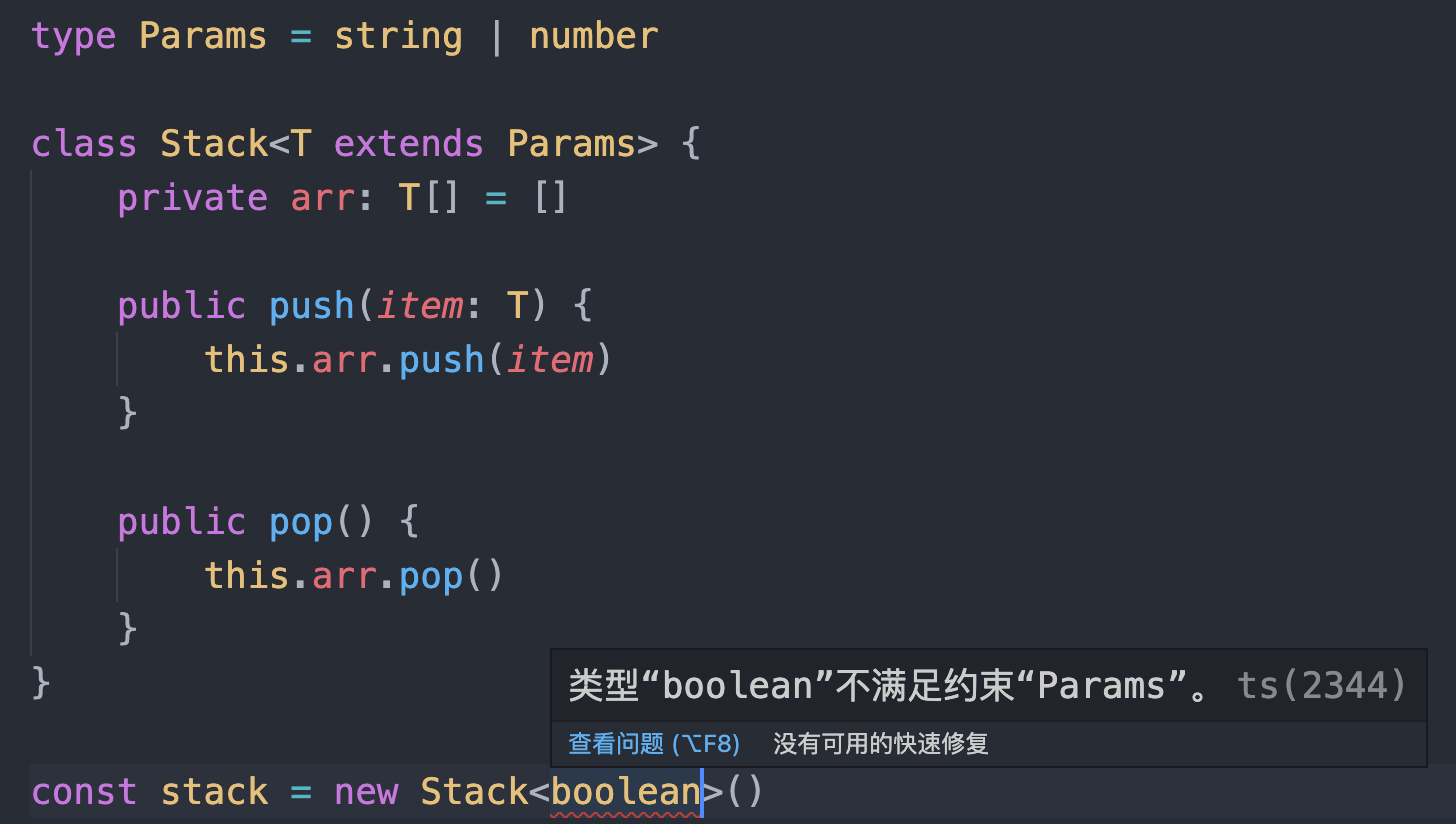
下面简单实现一个元素同类型的栈结构，如下所示：

class Stack<T> {  
 private arr: T[] = []  
  
 public push(item: T) {  
 this.arr.push(item)  
 }  
  
 public pop() {  
 this.arr.pop()  
 }  
}

使用方式如下：

const stack = new Stacn<number>()

如果上述只能传递 string 和 number 类型，这时候就可以使用 <T extends xx> 的方式猜实现**约束泛型**，如下所示：



除了上述的形式，泛型更高级的使用如下：

例如要设计一个函数，这个函数接受两个参数，一个参数为对象，另一个参数为对象上的属性，我们通过这两个参数返回这个属性的值

这时候就设计到泛型的索引类型和约束类型共同实现

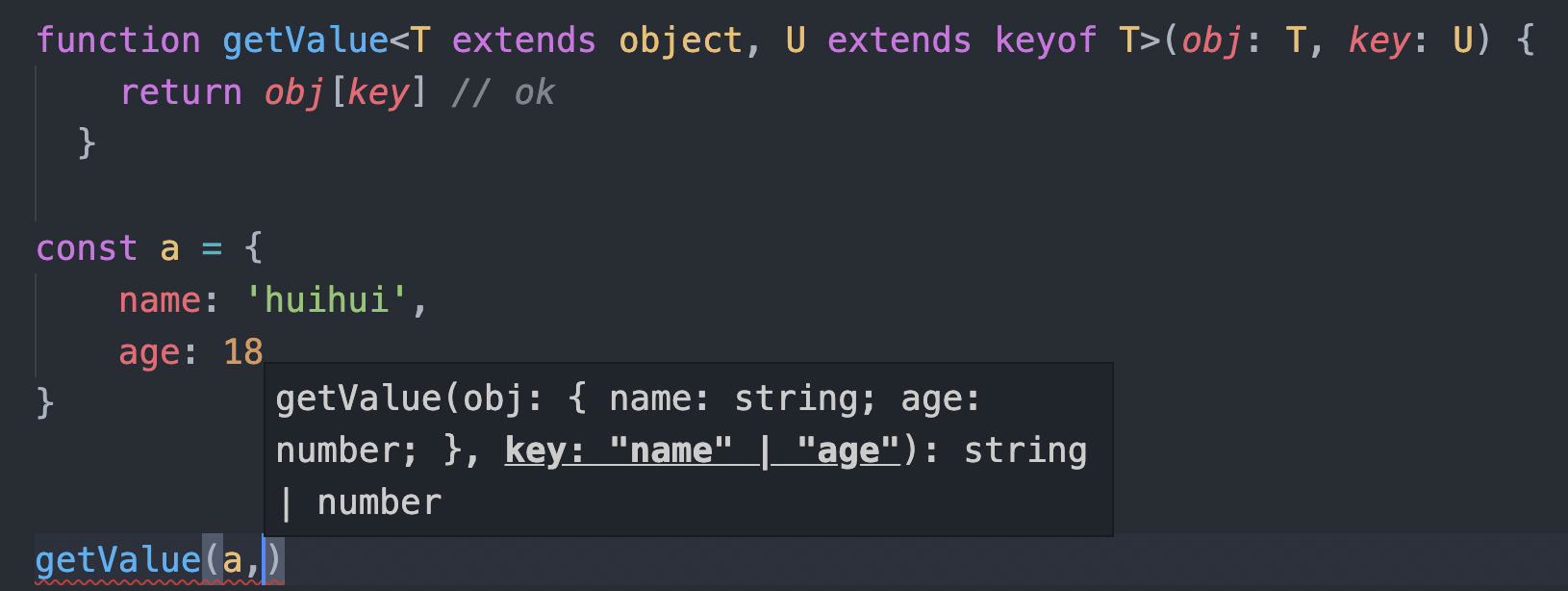
### 索引类型、约束类型

索引类型 keyof T 把传入的对象的属性类型取出生成一个联合类型，这里的泛型 U 被约束在这个联合类型中，如下所示：

function getValue<T extends object, U extends keyof T>(obj: T, key: U) {  
 return obj[key] // ok  
}

上述为什么需要使用泛型约束，而不是直接定义第一个参数为 object类型，是因为默认情况 object 指的是{}，而我们接收的对象是各种各样的，一个泛型来表示传入的对象类型，比如 T extends object

使用如下图所示：



### 多类型约束

例如如下需要实现两个接口的类型约束：

interface FirstInterface {  
 doSomething(): number  
}  
  
interface SecondInterface {  
 doSomethingElse(): string  
}

可以创建一个接口继承上述两个接口，如下：

interface ChildInterface extends FirstInterface, SecondInterface {  
  
}

正确使用如下：

class Demo<T extends ChildInterface> {  
 private genericProperty: T  
  
 constructor(genericProperty: T) {  
 this.genericProperty = genericProperty  
 }  
 useT() {  
 this.genericProperty.doSomething()  
 this.genericProperty.doSomethingElse()  
 }  
}

通过泛型约束就可以达到多类型约束的目的

## 三、应用场景

通过上面初步的了解，后述在编写 typescript 的时候，定义函数，接口或者类的时候，不预先定义好具体的类型，而在使用的时候在指定类型的一种特性的时候，这种情况下就可以使用泛型

灵活的使用泛型定义类型，是掌握typescript 必经之路

## 参考文献

* https://www.tslang.cn/docs/handbook/generics.html