# 面试官：说说你对 TypeScript 中高级类型的理解？有哪些？



## 一、是什么

除了string、number、boolean 这种基础类型外，在 typescript 类型声明中还存在一些高级的类型应用

这些高级类型，是typescript为了保证语言的灵活性，所使用的一些语言特性。这些特性有助于我们应对复杂多变的开发场景

## 二、有哪些

常见的高级类型有如下：

* 交叉类型
* 联合类型
* 类型别名
* 类型索引
* 类型约束
* 映射类型
* 条件类型

### 交叉类型

通过 & 将多个类型合并为一个类型，包含了所需的所有类型的特性，本质上是一种并的操作

语法如下：

T & U

适用于对象合并场景，如下将声明一个函数，将两个对象合并成一个对象并返回：

function extend<T , U>(first: T, second: U) : T & U {  
 let result: <T & U> = {}  
 for (let key in first) {  
 result[key] = first[key]  
 }  
 for (let key in second) {  
 if(!result.hasOwnProperty(key)) {  
 result[key] = second[key]  
 }  
 }  
 return result  
}

### 联合类型

联合类型的语法规则和逻辑 “或” 的符号一致，表示其类型为连接的多个类型中的任意一个，本质上是一个交的关系

语法如下：

T | U

例如 number | string | boolean 的类型只能是这三个的一种，不能共存

如下所示：

function formatCommandline(command: string[] | string) {  
 let line = '';  
 if (typeof command === 'string') {  
 line = command.trim();  
 } else {  
 line = command.join(' ').trim();  
 }  
}

### 类型别名

类型别名会给一个类型起个新名字，类型别名有时和接口很像，但是可以作用于原始值、联合类型、元组以及其它任何你需要手写的类型

可以使用 type SomeName = someValidTypeAnnotation的语法来创建类型别名：

type some = boolean | string  
  
const b: some = true // ok  
const c: some = 'hello' // ok  
const d: some = 123 // 不能将类型“123”分配给类型“some”

此外类型别名可以是泛型:

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

也可以使用类型别名来在属性里引用自己：

type Tree<T> = {  
 value: T;  
 left: Tree<T>;  
 right: Tree<T>;  
}

可以看到，类型别名和接口使用十分相似，都可以描述一个对象或者函数

两者最大的区别在于，interface只能用于定义对象类型，而 type 的声明方式除了对象之外还可以定义交叉、联合、原始类型等，类型声明的方式适用范围显然更加广泛

### 类型索引

keyof 类似于 Object.keys ，用于获取一个接口中 Key 的联合类型。

interface Button {  
 type: string  
 text: string  
}  
  
type ButtonKeys = keyof Button  
// 等效于  
type ButtonKeys = "type" | "text"

### 类型约束

通过关键字 extend 进行约束，不同于在 class 后使用 extends 的继承作用，泛型内使用的主要作用是对泛型加以约束

type BaseType = string | number | boolean  
  
// 这里表示 copy 的参数  
// 只能是字符串、数字、布尔这几种基础类型  
function copy<T extends BaseType>(arg: T): T {  
 return arg  
}

类型约束通常和类型索引一起使用，例如我们有一个方法专门用来获取对象的值，但是这个对象并不确定，我们就可以使用 extends 和 keyof 进行约束。

function getValue<T, K extends keyof T>(obj: T, key: K) {  
 return obj[key]  
}  
  
const obj = { a: 1 }  
const a = getValue(obj, 'a')

### 映射类型

通过 in 关键字做类型的映射，遍历已有接口的 key 或者是遍历联合类型，如下例子：

type Readonly<T> = {  
 readonly [P in keyof T]: T[P];  
};  
  
interface Obj {  
 a: string  
 b: string  
}  
  
type ReadOnlyObj = Readonly<Obj>

上述的结构，可以分成这些步骤：

* keyof T：通过类型索引 keyof 的得到联合类型 'a' | 'b'
* P in keyof T 等同于 p in 'a' | 'b'，相当于执行了一次 forEach 的逻辑，遍历 'a' | 'b'

所以最终ReadOnlyObj的接口为下述：

interface ReadOnlyObj {  
 readonly a: string;  
 readonly b: string;  
}

### 条件类型

条件类型的语法规则和三元表达式一致，经常用于一些类型不确定的情况。

T extends U ? X : Y

上面的意思就是，如果 T 是 U 的子集，就是类型 X，否则为类型 Y

## 三、总结

可以看到，如果只是掌握了 typeScript 的一些基础类型，可能很难游刃有余的去使用 typeScript，需要了解一些typescript的高阶用法

并且typescript在版本的迭代中新增了很多功能，需要不断学习与掌握

## 参考文献

* https://www.tslang.cn/docs/handbook/advanced-types.html
* https://juejin.cn/post/6844904003604578312
* https://zhuanlan.zhihu.com/p/103846208