

# TypeScript面试真题(12题)

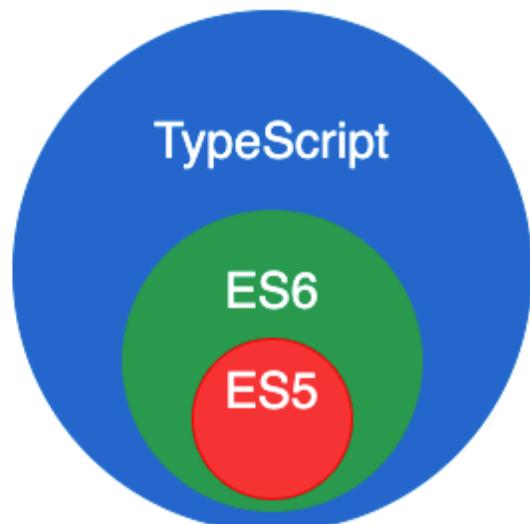
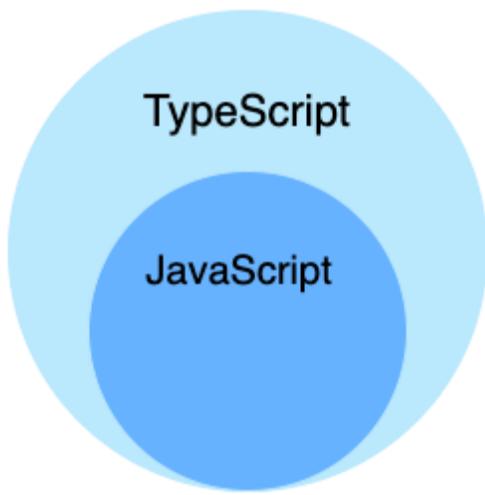
## 1. 说说你对 TypeScript 的理解？与 JavaScript 的区别？



### 1.1. 是什么

TypeScript 是 JavaScript 的类型的超集，支持 ES6 语法，支持面向对象编程的概念，如类、接口、继承、泛型等

超集，不得不说另外一个概念，子集，怎么理解这两个呢，举个例子，如果一个集合 A 里面的的所有元素集合 B 里面都存在，那么我们可以理解集合 B 是集合 A 的超集，集合 A 为集合 B 的子集



其是一种静态类型检查的语言，提供了类型注解，在代码编译阶段就可以检查出数据类型的错误同时扩展了 `JavaScript` 的语法，所以任何现有的 `JavaScript` 程序可以不加改变的在 `TypeScript` 下工作

为了保证兼容性，`TypeScript` 在编译阶段需要编译器编译成纯 `JavaScript` 来运行，是为大型应用之开发而设计的语言，如下：

`ts` 文件如下：

```
1 const hello: string = "Hello World!";
2 console.log(hello);
```

编译文件后：

```
1 const hello = "Hello World!";
2 console.log(hello);
```

## 1.2. 特性

`TypeScript` 的特性主要有如下：

- **类型批注和编译时类型检查**：在编译时批注变量类型
- **类型推断**：ts 中没有批注变量类型会自动推断变量的类型
- **类型擦除**：在编译过程中批注的内容和接口会在运行时利用工具擦除
- **接口**：ts 中用接口来定义对象类型
- **枚举**：用于取值被限定在一定范围内的场景
- **Mixin**：可以接受任意类型的值
- **泛型编程**：写代码时使用一些以后才指定的类型
- **名字空间**：名字只在该区域内有效，其他区域可重复使用该名字而不冲突
- **元组**：元组合并了不同类型的对象，相当于一个可以装不同类型数据的数组
- ...

### 1.2.1. 类型批注

通过类型批注提供在编译时启动类型检查的静态类型，这是可选的，而且可以忽略而使用 `JavaScript` 常规的动态类型

```
1 function Add(left: number, right: number): number {
```

```
2     return left + right;
3 }
```

对于基本类型的批注是 `number`、`bool` 和 `string`，而弱或动态类型的结构则是 `any` 类型

## 1.2.2. 类型推断

当类型没有给出时，TypeScript 编译器利用类型推断来推断类型，如下：

```
1 let str = "string";
```

变量 `str` 被推断为字符串类型，这种推断发生在初始化变量和成员，设置默认参数值和决定函数返回值时

如果缺乏声明而不能推断出类型，那么它的类型被视作默认的动态 `any` 类型

## 1.2.3. 接口

接口简单来说就是用来描述对象的类型 数据的类型有 `number`、`null`、`string` 等数据格式，对象的类型就是用接口来描述的

```
1 interface Person {
2     name: string;
3     age: number;
4 }
5 let tom: Person = {
6     name: "Tom",
7     age: 25,
8 };
```

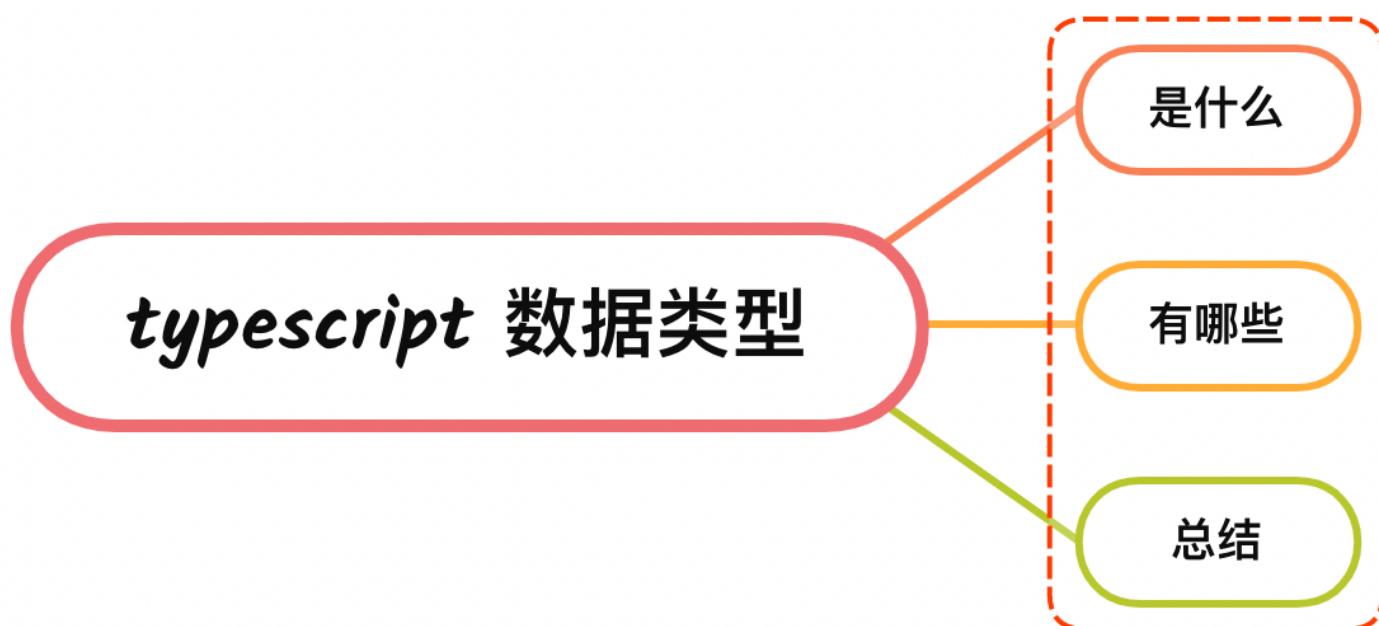
## 1.3. 区别

- TypeScript 是 JavaScript 的超集，扩展了 JavaScript 的语法
- TypeScript 可处理已有的 JavaScript 代码，并只对其中的 TypeScript 代码进行编译
- TypeScript 文件的后缀名 `.ts` (`.ts`, `.tsx`, `.dts`)，JavaScript 文件是 `.js`
- 在编写 TypeScript 的文件的时候就会自动编译成 `js` 文件

更多的区别如下图所示：

JavaScript		TypeScript
语言	脚本语言	面向对象编程语言
学习难度	灵活易学	需要有脚本编程经验
类型	轻量级解释编程语言	强类型的面向对象编程语言
客户端/服务	客户端服务端都有	侧重客户端
拓展名	.js	.ts 或 .tsx
耗时	更快	编译代码需要些时间
数据绑定	没有类型和接口的概念	使用类型和接口表示数据
语法	所有的语句都写在脚本标签内。浏览器将脚本标签内的文本识别为脚本	一个 TypeScript 程序由模块、方法、变量、语句、表达式和注释构成
静态类型	JS 中没有静态类型的概念	支持静态类型
模块支持	不支持模块	支持模块
接口	没有接口	支持接口
可选参数方法	不支持	支持
原型	没有这种特性	支持原型特性

## 2. 说说 typescript 的数据类型有哪些?



### 2.1. 是什么

`typescript` 和 `javascript` 几乎一样，拥有相同的数据类型，另外在 `javascript` 基础上提供了更加实用的类型供开发使用

在开发阶段，可以为明确的变量定义为某种类型，这样 `typescript` 就能在编译阶段进行类型检查，当类型不合符预期结果的时候则会出现错误提示

## 2.2. 有哪些

`typescript` 的数据类型主要有如下：

- `boolean` (布尔类型)
- `number` (数字类型)
- `string` (字符串类型)
- `array` (数组类型)
- `tuple` (元组类型)
- `enum` (枚举类型)
- `any` (任意类型)
- `null` 和 `undefined` 类型
- `void` 类型
- `never` 类型
- `object` 对象类型

### 2.2.1. `boolean`

布尔类型

```
1 let flag:boolean = true;
2 // flag = 123; // 错误
3 flag = false; // 正确
```

### 2.2.2. `number`

数字类型，和 `javascript` 一样，`typescript` 的数值类型都是浮点数，可支持二进制、八进制、十进制和十六进制

```
1 let num:number = 123;
2 // num = '456'; // 错误
3 num = 456; // 正确
```

进制表示：

```
1 let decLiteral: number = 6; // 十进制
2 let hexLiteral: number = 0xf00d; // 十六进制
3 let binaryLiteral: number = 0b1010; // 二进制
4 let octalLiteral: number = 0o744; // 八进制
```

## 2.2.3. string

字符串类型，和 JavaScript 一样，可以使用双引号 (") 或单引号 ('') 表示字符串

```
1 let str:string = 'this is ts';
2 str = 'test';
```

作为超集，当然也可以使用模版字符串``进行包裹，通过 \${} 嵌入变量

```
1 let name: string =
2 Gene
3 ;
4 let age: number = 37;
5 let sentence: string = `Hello, my name is ${ name }`
```

## 2.2.4. array

数组类型，跟 javascript 一致，通过 [] 进行包裹，有两种写法：

方式一：元素类型后面接上 []

```
1 let arr:string[] = ['12', '23'];
2 arr = ['45', '56'];
```

方式二：使用数组泛型， Array<元素类型>：

```
1 let arr:Array<number> = [1, 2];
2 arr = ['45', '56'];
```

## 2.2.5. tuple

元祖类型，允许表示一个已知元素数量和类型的数组，各元素的类型不必相同

```
1 let tupleArr:[number, string, boolean];
2 tupleArr = [12, '34', true]; //ok
3 tupleArr = [12, '34'] // no ok
```

赋值的类型、位置、个数需要和定义（声明）的类型、位置、个数一致

## 2.2.6. enum

`enum` 类型是对JavaScript标准数据类型的一个补充，使用枚举类型可以为一组数值赋予友好的名字

```
1 enum Color {Red, Green, Blue}
2 let c: Color = Color.Green;
```

## 2.2.7. any

可以指定任何类型的值，在编程阶段还不清楚类型的变量指定一个类型，不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查，这时候可以使用 `any` 类型

使用 `any` 类型允许被赋值为任意类型，甚至可以调用其属性、方法

```
1 let num:any = 123;
2 num = 'str';
3 num = true;
```

定义存储各种类型数据的数组时，示例代码如下：

```
1 let arrayList: any[] = [1, false, 'fine'];
2 arrayList[1] = 100;
```

## 2.2.8. null 和 undefined

在 `JavaScript` 中 `null` 表示“什么都没有”，是一个只有一个值的特殊类型，表示一个空对象引用，而 `undefined` 表示一个没有设置值的变量

默认情况下 `null` 和 `undefined` 是所有类型的子类型，就是说你可以把 `null` 和 `undefined` 赋值给 `number` 类型的变量

```
1 let num:number | undefined; // 数值类型 或者 undefined
2 console.log(num); // 正确
3 num = 123;
4 console.log(num); // 正确
```

但是 ts 配置了 `--strictNullChecks` 标记，`null` 和 `undefined` 只能赋值给 `void` 和它们各自

## 2.2.9. void

用于标识方法返回值的类型，表示该方法没有返回值。

```
1 function hello(): void {
2     alert("Hello Runoob");
3 }
```

## 2.2.10. never

`never` 是其他类型（包括 `null` 和 `undefined`）的子类型，可以赋值给任何类型，代表从不会出现的值

但是没有类型是 `never` 的子类型，这意味着声明 `never` 的变量只能被 `never` 类型所赋值。

`never` 类型一般用来指定那些总是会抛出异常、无限循环

```
1 let a:never;
2 a = 123; // 错误的写法
3 a = (() => { // 正确的写法
4     throw new Error('错误');
5 })()
6 // 返回never的函数必须存在无法达到的终点
7 function error(message: string): never {
8     throw new Error(message);
9 }
```

## 2.2.11. object

对象类型，非原始类型，常见的形式通过 `{}` 进行包裹

```
1 let obj:object;
2 obj = {name: 'Wang', age: 25};
```

## 2.3. 总结

和 `javascript` 基本一致，也分成：

- 基本类型
- 引用类型

在基础类型上，`typescript` 增添了 `void`、`any`、`enum` 等原始类型

## 3. 说说你对 TypeScript 中高级类型的理解？有哪些？



### 3.1. 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

### 3.2. 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

### 3.2.1. 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：

```
1 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：

```
1 function extend<T , U>(first: T, second: U) : T & U {  
2     let result: <T & U> = {}  
3     for (let key in first) {  
4         result[key] = first[key]  
5     }  
6     for (let key in second) {  
7         if(!result.hasOwnProperty(key)) {  
8             result[key] = second[key]  
9         }  
10    }  
11    return result  
12 }
```

### 3.2.2. 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：

```
1 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：

```
1 function formatCommandLine(command: string[] | string) {  
2     let line = '';
```

```
3  if (typeof command === 'string') {  
4      line = command.trim();  
5  } else {  
6      line = command.join(' ').trim();  
7  }  
8 }
```

### 3.2.3. 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、元组以及其他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：

```
1 type some = boolean | string  
2 const b: some = true // ok  
3 const c: some = 'hello' // ok  
4 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：

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

也可以使用类型别名来在属性里引用自己：

```
1 type Tree<T> = {  
2     value: T;  
3     left: Tree<T>;  
4     right: Tree<T>;  
5 }
```

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

### 3.2.4. 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。

```
1 interface Button {  
2     type: string  
3     text: string  
4 }  
5 type ButtonKeys = keyof Button  
6 // 等效于  
7 type ButtonKeys = "type" | "text"
```

### 3.2.5. 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

```
1 type BaseType = string | number | boolean  
2 // 这里表示 copy 的参数  
3 // 只能是字符串、数字、布尔这几种基础类型  
4 function copy<T extends BaseType>(arg: T): T {  
5     return arg  
6 }
```

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。

```
1 function getValue<T, K extends keyof T>(obj: T, key: K) {  
2     return obj[key]  
3 }  
4 const obj = { a: 1 }  
5 const a = getValue(obj, 'a')
```

### 3.2.6. 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：

```
1 type Readonly<T> = {  
2     readonly [P in keyof T]: T[P];  
3 };  
4 interface Obj {  
5     a: string  
6     b: string  
7 }  
8 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：

```
1 interface ReadOnlyObj {  
2     readonly a: string;  
3     readonly b: string;  
4 }
```

### 3.2.7. 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。

```
1 T extends U ? X : Y
```

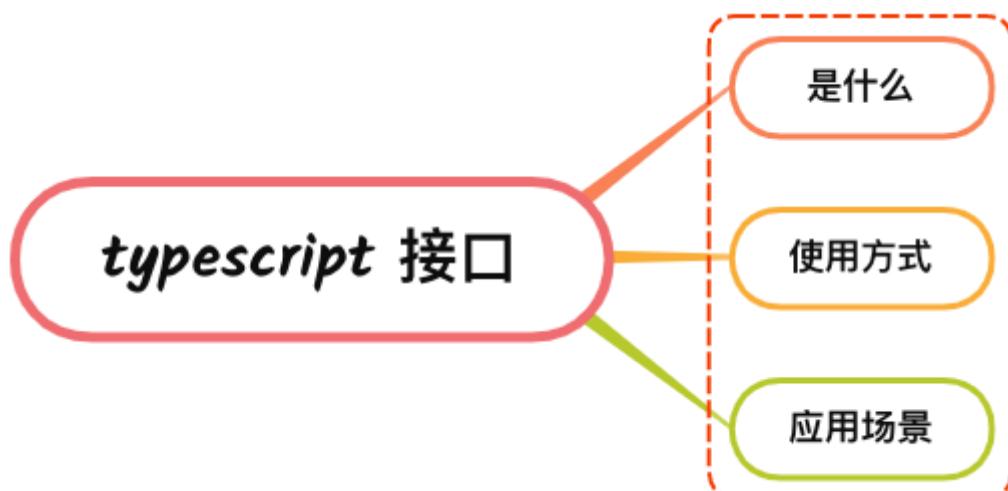
上面的意思就是，如果 `T` 是 `U` 的子集，就是类型 `X`，否则为类型 `Y`

## 3.3. 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

## 4. 说说你对 TypeScript 中接口的理解？应用场景？



## 4.1. 是什么

接口是一系列抽象方法的声明，是一些方法特征的集合，这些方法都应该是抽象的，需要由具体的类去实现，然后第三方就可以通过这组抽象方法调用，让具体的类执行具体的方法

简单来讲，一个接口所描述的是一个对象相关的属性和方法，但并不提供具体创建此对象实例的方法

`typescript` 的核心功能之一就是对类型做检测，虽然这种检测方式是“鸭式辨型法”，而接口的作用就是为这些类型命名和为你的代码或第三方代码定义一个约定

## 4.2. 使用方式

接口定义如下

```
1 interface interface_name {  
2 }
```

例如有一个函数，这个函数接受一个 `User` 对象，然后返回这个 `User` 对象的 `name` 属性：

```
1 const.getUserName = (user) => user.name
```

可以看到，参数需要有一个 `user` 的 `name` 属性，可以通过接口描述 `user` 参数的结构

```
1 interface User {  
2     name: string  
3     age: number  
4 }  
5 const.getUserName = (user: User) => user.name
```

这些属性并不一定全部实现，上述传入的对象必须拥有 `name` 和 `age` 属性，否则 `typescript` 在编译阶段会报错，如下图：

```
interface User {  
  name: string,  
  age: Number  
}
```

```
const fn = (user: User) => {user.name}
```

类型“{ name: string; }”的参数不能赋给类型“User”的参数。

类型 “{ name: string; }” 中缺少属性 “age”，但类型 “User” 中需要该属性。 ts(2345)

index.ts(4, 5): 在此处声明了 “age”。

查看问题 (F8) 没有可用的快速修复

```
fn({name:"huihui"})
```

如果不想要 age 属性的话，这时候可以采用可选属性，如下表示：

```
1 interface User {  
2   name: string  
3   age?: number  
4 }
```

这时候 age 属性则可以是 number 类型或者 undefined 类型

有些时候，我们想要一个属性变成只读属性，在 typescript 只需要使用 readonly 声明，如下：

```
1 interface User {  
2   name: string  
3   age?: number  
4   readonly isMale: boolean  
5 }
```

当我们修改属性的时候，就会出现警告，如下所示：

```
interface User {  
  name: string,  
  age: Number,  
  readonly isOnly: boolean  
}
```

无法分配到 "isOnly"，因为它是只读属性。ts(2540)

(property) User.isOnly: any

```
const fn 查看问题 (F8) 没有可用的快速修复  
  user.isOnly = false  
}
```

这是属性中有一个函数，可以如下表示：

```
1 interface User {  
2   name: string  
3   age?: number  
4   readonly isMale: boolean  
5   say: (words: string) => string  
6 }
```

如果传递的对象不仅仅是上述的属性，这时候可以使用：

- **类型推断**

```
1 interface User {  
2   name: string  
3   age: number  
4 }  
5 const.getUserName = (user: User) => user.name  
6 getUserName({color: 'yellow'} as User)
```

- **给接口添加字符串索引签名**

```
1 interface User {  
2   name: string  
3   age: number
```

```
4     [propName: string]: any;
5 }
```

接口还能实现继承，如下图：

```
interface Father {
  color: String
}

interface Son extends Father{
  name: string
  age: Number
}

const fn = (user: Son) => {
  user.|
  }          ↗ age          (property) Son.age: Num...
  ↗ color
  ↗ name
```

也可以继承多个，父类通过逗号隔开，如下：

```
1 interface Father {
2   color: String
3 }
4 interface Mother {
5   height: Number
6 }
7 interface Son extends Father,Mother{
8   name: string
9   age: Number
10 }
```

## 4.3. 应用场景

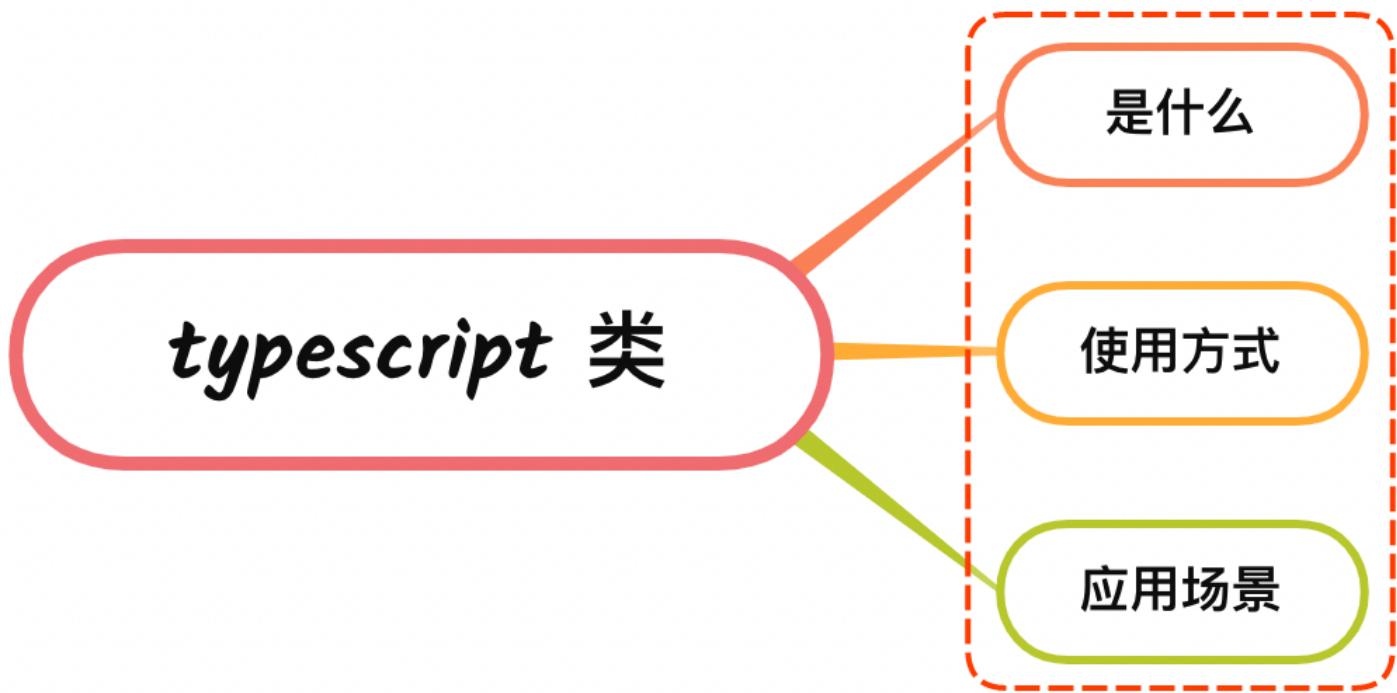
例如在 `javascript` 中定义一个函数，用来获取用户的姓名和年龄：

```
1 const getUserInfo = function(user) {  
2     // ...  
3     return name:  
4 ${user.name}, age: $  
5 {user.age}  
6 }
```

如果多人开发的都需要用到这个函数的时候，如果没有注释，则可能出现各种运行时的错误，这时候就可以使用接口定义参数变量：

```
1 // 先定义一个接口  
2 interface IUser {  
3     name: string;  
4     age: number;  
5 }  
6 const getUserInfo = (user: IUser): string => {  
7     return  
8     name: ${user.name}, age: ${user.age}  
9 ;  
10 };  
11 // 正确的调用  
12 getUserInfo({name: "koala", age: 18});
```

## 5. 说说你对 TypeScript 中类的理解？应用场景？



## 5.1. 是什么

类 (Class) 是面向对象程序设计 (OOP, Object-Oriented Programming) 实现信息封装的基础

类是一种用户定义的引用数据类型，也称类类型

传统的面向对象语言基本都是基于类的，`JavaScript` 基于原型的方式让开发者多了很多理解成本

在 `ES6` 之后，`JavaScript` 拥有了 `class` 关键字，虽然本质依然是构造函数，但是使用起来已经方便了许多

但是 `JavaScript` 的 `class` 依然有一些特性还没有加入，比如修饰符和抽象类

`TypeScript` 的 `class` 支持面向对象的所有特性，比如类、接口等

## 5.2. 使用方式

定义类的关键字为 `class`，后面紧跟类名，类可以包含以下几个模块（类的数据成员）：

- **字段**：字段是类里面声明的变量。字段表示对象的有关数据。
- **构造函数**：类实例化时调用，可以为类的对象分配内存。
- **方法**：方法为对象要执行的操作

如下例子：

```

1 class Car {
2     // 字段
3     engine:string;
4     // 构造函数
5     constructor(engine:string) {
6         this.engine = engine
  
```

```
7     }
8     // 方法
9     disp():void {
10         console.log("发动机为 : " + this.engine)
11     }
12 }
```

## 5.2.1. 继承

类的继承使用过 `extends` 的关键字

```
1 class Animal {
2     move(distanceInMeters: number = 0) {
3         console.log(
4             Animal moved ${distanceInMeters}m.
5         );
6     }
7 }
8 class Dog extends Animal {
9     bark() {
10         console.log('Woof! Woof!');
11     }
12 }
13 const dog = new Dog();
14 dog.bark();
15 dog.move(10);
16 dog.bark();
```

`Dog` 是一个派生类，它派生自 `Animal` 基类，派生类通常被称作子类，基类通常被称作超类

`Dog` 类继承了 `Animal` 类，因此实例 `dog` 也能够使用 `Animal` 类 `move` 方法

同样，类继承后，子类可以对父类的方法重新定义，这个过程称之为方法的重写，通过 `super` 关键字是对父类的直接引用，该关键字可以引用父类的属性和方法，如下：

```
1 class PrinterClass {
2     doPrint():void {
3         console.log("父类的 doPrint() 方法。")
4     }
5 }
6 class StringPrinter extends PrinterClass {
7     doPrint():void {
8         super.doPrint() // 调用父类的函数
9         console.log("子类的 doPrint() 方法。")
```

```
10      }
11 }
```

## 5.2.2. 修饰符

可以看到，上述的形式跟 `ES6` 十分的相似， `typescript` 在此基础上添加了三种修饰符：

- 公共 `public`: 可以自由的访问类程序里定义的成员
- 私有 `private`: 只能够在该类的内部进行访问
- 受保护 `protect`: 除了在该类的内部可以访问，还可以在子类中仍然可以访问

## 5.2.3. 私有修饰符

只能够在该类的内部进行访问，实例对象并不能够访问

```
class Father {
  private name: String
  constructor(name: String) {
    this.name = name
  }
}

const father = new Father('huihui')
```

属性“name”为私有属性，只能在类“Father”中访问。 ts(2341)

(property) Father.name: String

查看问题 (F8) 没有可用的快速修复

father.name

并且继承该类的子类并不能访问，如下图所示：

```
class Father {
  private name: String
  constructor(name: String) {
    this.name = name
  }
}

class Son extends Father{
  say() {
    console.log(`my name is ${this.name}`)
  }
}
```

属性“name”为私有属性，只能在类“Father”中访问。 ts(2341)

(property) Father.name: String

查看问题 (F8) 没有可用的快速修复

console.log(`my name is \${this.name}`)

## 5.2.4. 受保护修饰符

跟私有修饰符很相似，实例对象同样不能访问受保护的属性，如下：

```
class Father {
  protected name: String
  constructor(name: String) {
    this.name = name
  }
}

const father = new Father('huihui')

属性“name”受保护，只能在类“Father”及其子类中访问。 ts(2445)
(property) Father.name: String
查看问题 (F8) 没有可用的快速修复
father.name
```

有一点不同的是 `protected` 成员在子类中仍然可以访问

```
class Father {
  protected name: String
  constructor(name: String) {
    this.name = name
  }
}

class Son extends Father{
  say() {
    console.log(`my name is ${this.name}`)
  }
}
```

除了上述修饰符之外，还有只读修饰符

### 5.2.4.1. 只读修饰符

通过 `readonly` 关键字进行声明，只读属性必须在声明时或构造函数里被初始化，如下：

```

class Father {
  readonly name: String
  constructor(name: String) {
    this.name = name
  }
}

const father = new Father('huihui')

    无法分配到 "name"，因为它是只读属性。ts(2540)
    (property) Father.name: any
    查看问题(F8) 没有可用的快速修复
father.name = 'change'

```

除了实例属性之外，同样存在静态属性

### 5.2.5. 静态属性

这些属性存在于类本身上面而不是类的实例上，通过 `static` 进行定义，访问这些属性需要通过 `类型.静态属性` 的这种形式访问，如下所示：

```

1 class Square {
2   static width = '100px'
3 }
4 console.log(Square.width) // 100px

```

上述的类都能发现一个特点就是，都能够被实例化，在 `typescript` 中，还存在一种抽象类

### 5.2.6. 抽象类

抽象类做为其它派生类的基类使用，它们一般不会直接被实例化，不同于接口，抽象类可以包含成员的实现细节

`abstract` 关键字是用于定义抽象类和在抽象类内部定义抽象方法，如下所示：

```

1 abstract class Animal {
2   abstract makeSound(): void;
3   move(): void {
4     console.log('roaming the earch...');
5   }
6 }

```

这种类并不能被实例化，通常需要我们创建子类去继承，如下：

```
1 class Cat extends Animal {  
2     makeSound() {  
3         console.log('miao miao')  
4     }  
5 }  
6 const cat = new Cat()  
7 cat.makeSound() // miao miao  
8 cat.move() // roaming the earch...
```

### 5.3. 应用场景

除了日常借助类的特性完成日常业务代码，还可以将类（class）也可以作为接口，尤其在 `React` 工程中是很常用的，如下：

```
1 export default class Carousel extends React.Component<Props, State> {}
```

由于组件需要传入 `props` 的类型 `Props`，同时有需要设置默认 `props` 即 `defaultProps`，这时候更加适合使用 `class` 作为接口

先声明一个类，这个类包含组件 `props` 所需的类型和初始值：

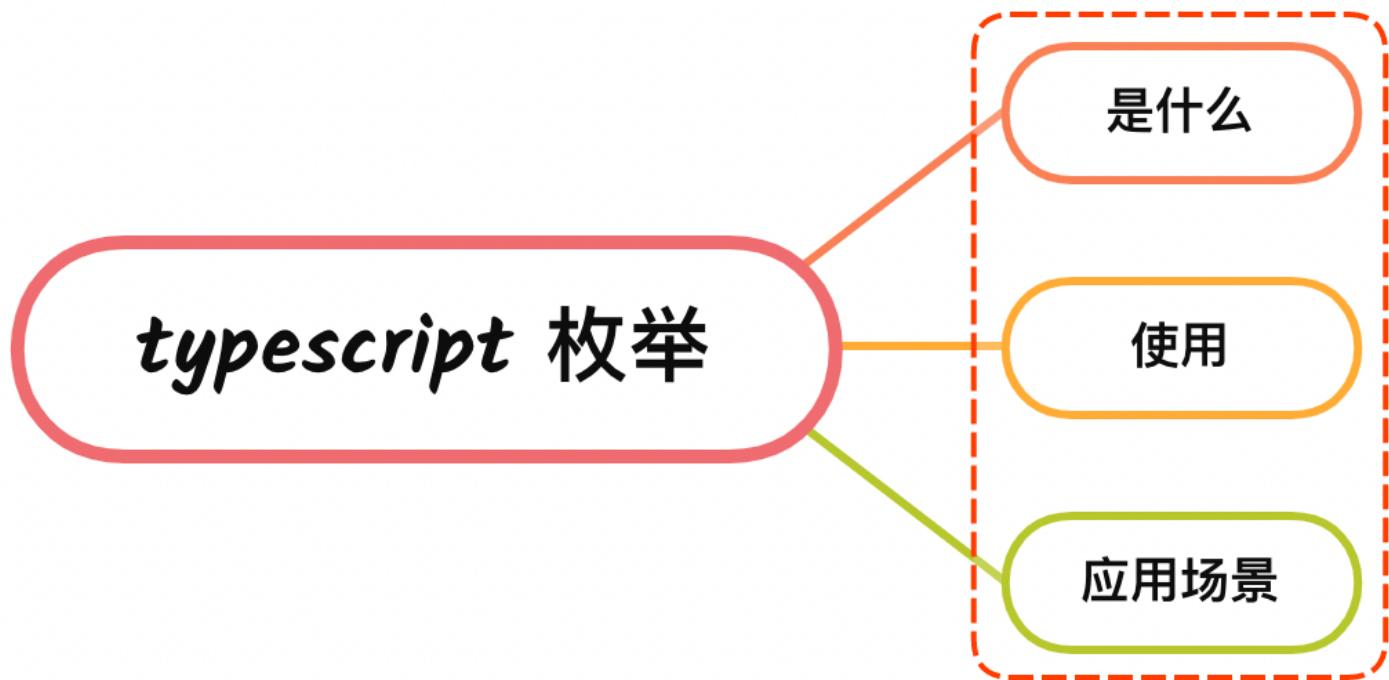
```
1 // props的类型  
2 export default class Props {  
3     public children: Array<React.ReactElement<any>> | React.ReactElement<any> |  
4         never[] = []  
5     public speed: number = 500  
6     public height: number = 160  
7     public animation: string = 'easeInOutQuad'  
8     public isAuto: boolean = true  
9     public autoPlayInterval: number = 4500  
10    public afterChange: () => {}  
11    public beforeChange: () => {}  
12    public selectedColor: string  
13    public showDots: boolean = true  
14 }
```

当我们需要传入 `props` 类型的时候直接将 `Props` 作为接口传入，此时 `Props` 的作用就是接口，而当需要我们设置 `defaultProps` 初始值的时候，我们只需要：

```
1 public static defaultProps = new Props()
```

Props 的实例就是 defaultProps 的初始值，这就是 class 作为接口的实际应用，我们用一个 class 起到了接口和设置初始值两个作用，方便统一管理，减少了代码量

## 6. 说说你对 TypeScript 中枚举类型的理解？应用场景？



### 6.1. 是什么

枚举是一个被命名的整型常数的集合，用于声明一组命名的常数，当一个变量有几种可能的取值时，可以将它定义为枚举类型

通俗来说，枚举就是一个对象的所有可能取值的集合

在日常生活中也很常见，例如表示星期的SUNDAY、MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY就可以看成是一个枚举

枚举的说明与结构和联合相似，其形式为：

```
1 enum 枚举名{  
2     标识符①[=整型常数]，  
3     标识符②[=整型常数]，  
4     ...  
5     标识符N[=整型常数]，  
6 }枚举变量；
```

## 6.2. 使用

枚举的使用是通过 `enum` 关键字进行定义，形式如下：

```
1 enum xxx { ... }
```

声明关键字为枚举类型的方式如下：

```
1 // 声明d为枚举类型Direction
2 let d: Direction;
```

类型可以分成：

- 数字枚举
- 字符串枚举
- 异构枚举

### 6.2.1. 数字枚举

当我们声明一个枚举类型时，虽然没有给它们赋值，但是它们的值其实是默认的数字类型，而且默认从0开始依次累加：

```
1 enum Direction {
2     Up,    // 值默认为 0
3     Down,  // 值默认为 1
4     Left,  // 值默认为 2
5     Right // 值默认为 3
6 }
7 console.log(Direction.Up === 0); // true
8 console.log(Direction.Down === 1); // true
9 console.log(Direction.Left === 2); // true
10 console.log(Direction.Right === 3); // true
```

如果我们将第一个值进行赋值后，后面的值也会根据前一个值进行累加1：

```
1 enum Direction {
2     Up = 10,
3     Down,
```

```
4     Left,
5     Right
6 }
7 console.log(Direction.Up, Direction.Down, Direction.Left, Direction.Right); // 10 11 12 13
```

## 6.2.2. 字符串枚举

1 枚举类型的值其实也可以是字符串类型：

```
2 enum Direction {
3     Up = 'Up',
4     Down = 'Down',
5     Left = 'Left',
6     Right = 'Right'
7 }
8 console.log(Direction['Right'], Direction.Up); // Right Up
```

如果设定了一个变量为字符串之后，后续的字段也需要赋值字符串，否则报错：

```
1 enum Direction {
2     Up = 'UP',
3     Down, // error TS1061: Enum member must have initializer
4     Left, // error TS1061: Enum member must have initializer
5     Right // error TS1061: Enum member must have initializer
6 }
```

## 6.2.3. 异构枚举

即将数字枚举和字符串枚举结合起来混合起来使用，如下：

```
1 enum BooleanLikeHeterogeneousEnum {
2     No = 0,
3     Yes = "YES",
4 }
```

通常情况下我们很少会使用异构枚举

## 6.2.4. 本质

现在一个枚举的案例如下：

```
1 enum Direction {  
2     Up,  
3     Down,  
4     Left,  
5     Right  
6 }
```

通过编译后，`javascript` 如下：

```
1 var Direction;  
2 (function (Direction) {  
3     Direction[Direction["Up"] = 0] = "Up";  
4     Direction[Direction["Down"] = 1] = "Down";  
5     Direction[Direction["Left"] = 2] = "Left";  
6     Direction[Direction["Right"] = 3] = "Right";  
7 })(Direction || (Direction = {}));
```

上述代码可以看到，`Direction[Direction["Up"] = 0] = "Up"` 可以分成

- `Direction["Up"] = 0`
- `Direction[0] = "Up"`

所以定义枚举类型后，可以通过正反射拿到对应的值，如下：

```
1 enum Direction {  
2     Up,  
3     Down,  
4     Left,  
5     Right  
6 }  
7 console.log(Direction.Up === 0); // true  
8 console.log(Direction[0]); // Up
```

并且多处定义的枚举是可以进行合并操作，如下：

```
1 enum Direction {  
2     Up = 'Up',  
3     Down = 'Down',  
4     Left = 'Left',  
5     Right = 'Right'
```

```
6 }
7 enum Direction {
8     Center = 1
9 }
```

编译后，`js` 代码如下：

```
1 var Direction;
2 (function (Direction) {
3     Direction["Up"] = "Up";
4     Direction["Down"] = "Down";
5     Direction["Left"] = "Left";
6     Direction["Right"] = "Right";
7 })(Direction || (Direction = {}));
8 (function (Direction) {
9     Direction[Direction["Center"] = 1] = "Center";
10 })(Direction || (Direction = {}));
```

可以看到，`Direction` 对象属性回叠加

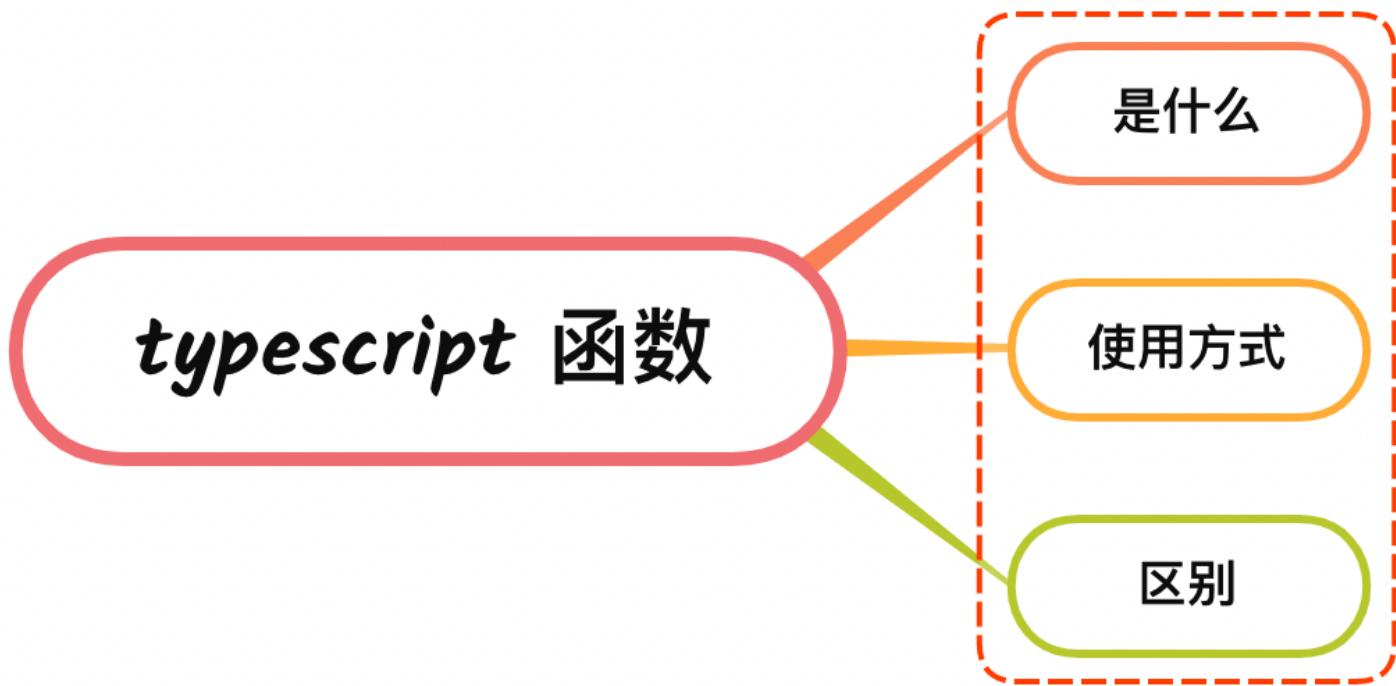
### 6.3. 应用场景

就拿回生活的例子，后端返回的字段使用 0 - 6 标记对应的日期，这时候就可以使用枚举可提高代码可读性，如下：

```
1 enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
2 console.log(Days["Sun"] === 0); // true
3 console.log(Days["Mon"] === 1); // true
4 console.log(Days["Tue"] === 2); // true
5 console.log(Days["Sat"] === 6); // true
```

包括后端日常返回0、1 等等状态的时候，我们都可以通过枚举去定义，这样可以提高代码的可读性，便于后续的维护

## 7. 说说你对 TypeScript 中函数的理解？与 JavaScript 函数的区别？



## 7.1. 是什么

函数是 `JavaScript` 应用程序的基础，帮助我们实现抽象层、模拟类、信息隐藏和模块

在 `TypeScript` 里，虽然已经支持类、命名空间和模块，但函数仍然是主要定义行为的方式，`TypeScript` 为 `JavaScript` 函数添加了额外的功能，丰富了更多的应用场景

函数类型在 `TypeScript` 类型系统中扮演着非常重要的角色，它们是可组合系统的核心构建块

## 7.2. 使用方式

跟 `javascript` 定义函数十分相似，可以通过 `funciton` 关键字、箭头函数等形式去定义，例如下面一个简单的加法函数：

```
1 const add = (a: number, b: number) => a + b
```

上述只定义了函数的两个参数类型，这个时候整个函数虽然没有被显式定义，但是实际上 `TypeScript` 编译器是能够通过类型推断到这个函数的类型，如下图所示：

```
1
2      const add: (a: number, b: number) => number
3 const add = (a: number, b: number) => a + b
```

当鼠标放置在第三行 `add` 函数名的时候，会出现完整的函数定义类型，通过 `:` 的形式来定于参数类型，通过 `=>` 连接参数和返回值类型

当我们没有提供函数实现的情况下，有两种声明函数类型的方式，如下所示：

```
1 // 方式一
2 type LongHand = {
3   (a: number): number;
4 };
5 // 方式二
6 type ShortHand = (a: number) => number;
```

当存在函数重载时，只能使用方式一的形式

### 7.2.1. 可选参数

当函数的参数可能是不存在的，只需要在参数后面加上 `?` 代表参数可能不存在，如下：

```
1 const add = (a: number, b?: number) => a + (b ? b : 0)
```

这时候参数 `b` 可以是 `number` 类型或者 `undefined` 类型，即可以传一个 `number` 类型或者不传都可以

### 7.2.2. 剩余类型

剩余参数与 `JavaScript` 的语法类似，需要用 `...` 来表示剩余参数

如果剩余参数 `rest` 是一个由 `number` 类型组成的数组，则如下表示：

```
1 const add = (a: number, ...rest: number[]) => rest.reduce((a, b) => a + b), a)
```

### 7.2.3. 函数重载

允许创建数项名称相同但输入输出类型或个数不同的子程序，它可以简单地称为一个单独功能可以执行多项任务的能力

关于 `typescript` 函数重载，必须要把精确的定义放在前面，最后函数实现时，需要使用 `|` 操作符或者 `?` 操作符，把所有可能的输入类型全部包含进去，用于具体实现

这里的函数重载也只是多个函数的声明，具体的逻辑还需要自己去写，`typescript` 并不会真的将你的多个重名 `function` 的函数体进行合并

例如我们有一个 `add` 函数，它可以接收 `string` 类型的参数进行拼接，也可以接收 `number` 类型的参数进行相加，如下：

```
1 // 上边是声明
```

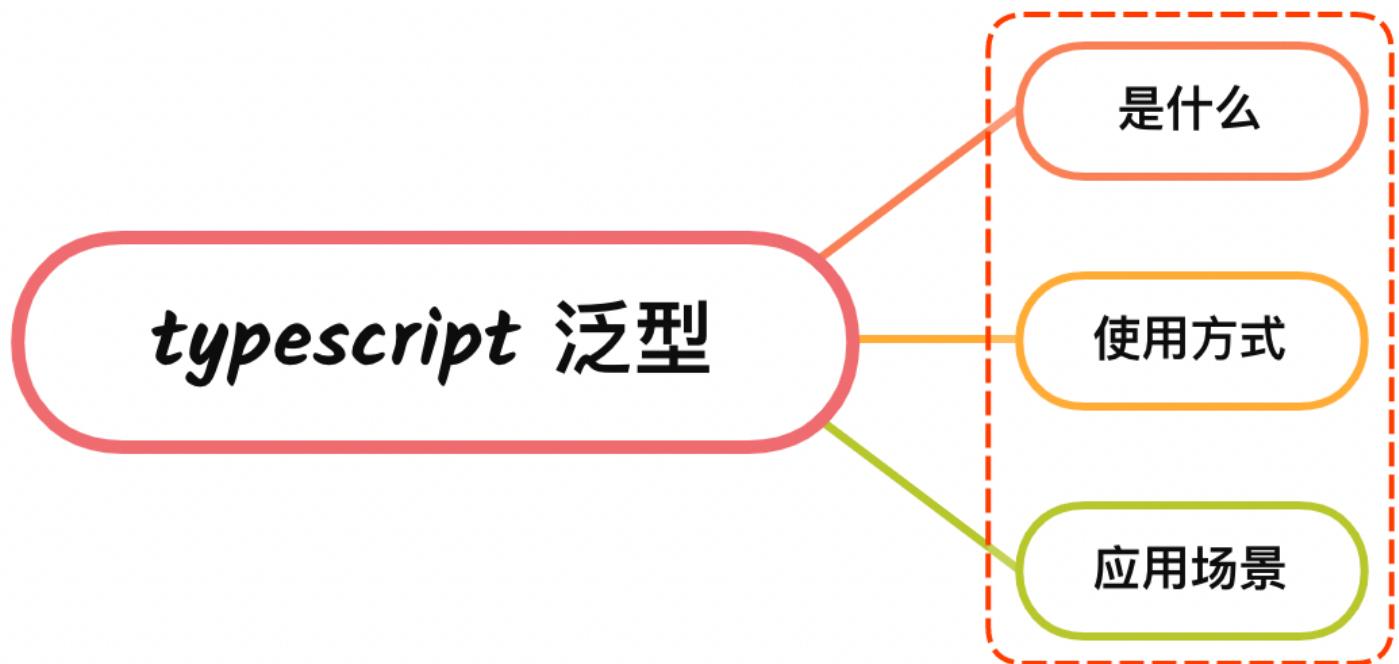
```
2 function add (arg1: string, arg2: string): string
3 function add (arg1: number, arg2: number): number
4 // 因为我们在下边有具体函数的实现, 所以这里并不需要添加 declare 关键字
5 // 下边是实现
6 function add (arg1: string | number, arg2: string | number) {
7     // 在实现上我们要注意严格判断两个参数的类型是否相等, 而不能简单的写一个 arg1 + arg2
8     if (typeof arg1 === 'string' && typeof arg2 === 'string') {
9         return arg1 + arg2
10    } else if (typeof arg1 === 'number' && typeof arg2 === 'number') {
11        return arg1 + arg2
12    }
13 }
```

### 7.3. 区别

从上面可以看到:

- 从定义的方式而言, typescript 声明函数需要定义参数类型或者声明返回值类型
- typescript 在参数中, 添加可选参数供使用者选择
- typescript 增添函数重载功能, 使用者只需要通过查看函数声明的方式, 即可知道函数传递的参数个数以及类型

## 8. 说说你对 TypeScript 中泛型的理解? 应用场景?



### 8.1. 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：

```
1 function returnItem (para: number): number {  
2     return para  
3 }
```

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：

```
1 function returnItem (para: string): string {  
2     return para  
3 }
```

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：

```
1 function returnItem<T>(para: T): T {  
2     return para  
3 }
```

可以看到，泛型给予开发者创造灵活、可重用代码的能力

## 8.2. 使用方式

泛型通过 `<>` 的形式进行表述，可以声明：

- 函数
- 接口
- 类

### 8.2.1. 函数声明

声明函数的形式如下：

```
1 function returnItem<T>(para: T): T {  
2     return para  
3 }
```

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：

```
1 function swap<T, U>(tuple: [T, U]): [U, T] {  
2     return [tuple[1], tuple[0]];  
3 }  
4 swap([7, 'seven']); // ['seven', 7]
```

## 8.2.2. 接口声明

声明接口的形式如下：

```
1 interface ReturnItemFn<T> {  
2     (para: T): T  
3 }
```

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：

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

## 8.2.3. 类声明

使用泛型声明类的时候，既可以作用于类本身，也可以作用于类的成员函数

下面简单实现一个元素同类型的栈结构，如下所示：

```
1 class Stack<T> {  
2     private arr: T[] = []  
3     public push(item: T) {  
4         this.arr.push(item)  
5     }  
6     public pop() {  
7         this.arr.pop()  
8     }  
9 }
```

使用方式如下：

```
1 const stack = new Stack<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：

```
type Params = string | number

class Stack<T extends Params> {
  private arr: T[] = []

  public push(item: T) {
    this.arr.push(item)
  }

  public pop() {
    this.arr.pop()
  }
}

const stack = new Stack<boolean>()
```

类型“boolean”不满足约束“Params”。 ts(2344)

[查看问题 \(F8\)](#) 没有可用的快速修复

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

#### 8.2.4. 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：

```
1 function getValue<T extends object, U extends keyof T>(obj: T, key: U) {
2   return obj[key] // ok
3 }
```

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：

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

const a = {
  name: 'huihui',
  age: 18
} getValue(obj: { name: string; age: number }, key: "name" | "age"): string | number
getValue(a,)
```

## 8.2.5. 多类型约束

例如如下需要实现两个接口的类型约束：

```
1 interface FirstInterface {
2   doSomething(): number
3 }
4 interface SecondInterface {
5   doSomethingElse(): string
6 }
```

可以创建一个接口继承上述两个接口，如下

```
1 interface ChildInterface extends FirstInterface, SecondInterface {
2 }
```

正确使用如下：

```
1 class Demo<T extends ChildInterface> {
2   private genericProperty: T
3   constructor(genericProperty: T) {
4     this.genericProperty = genericProperty
5 }
```

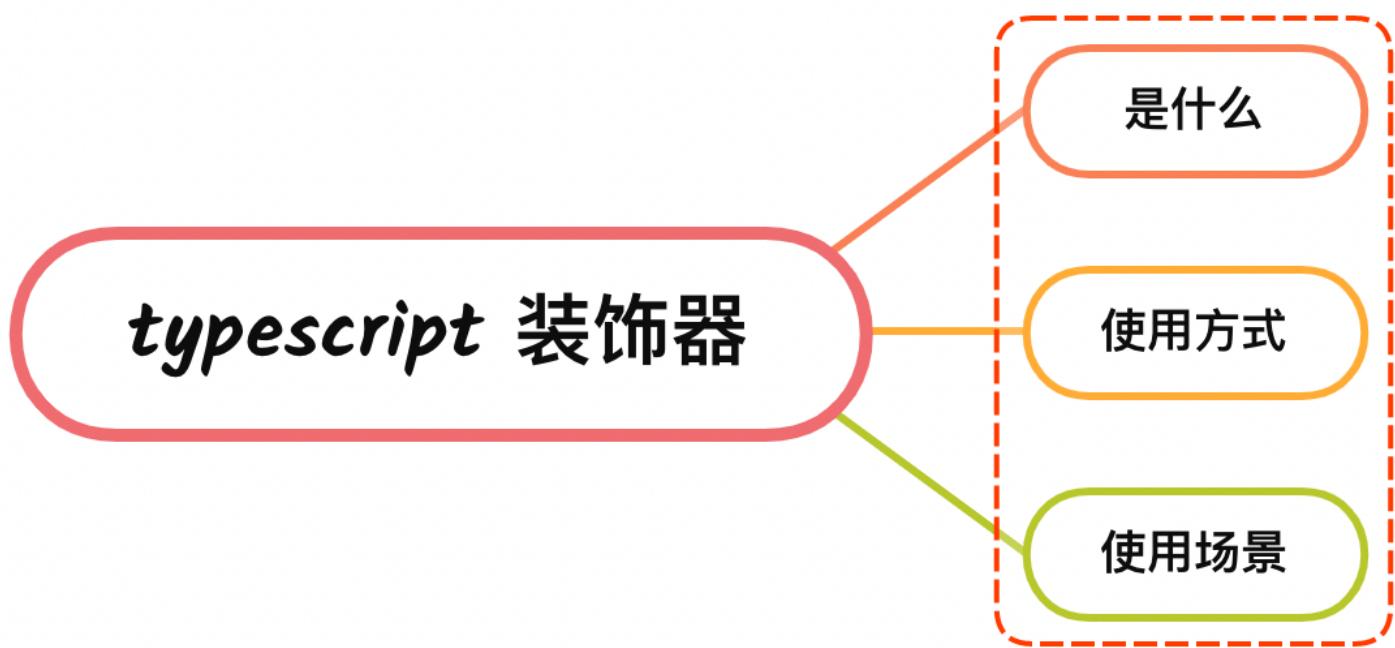
```
6  useT() {  
7      this.genericProperty.doSomething()  
8      this.genericProperty.doSomethingElse()  
9  }  
10 }
```

通过泛型约束就可以达到多类型约束的目的

### 8.3. 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

## 9. 说说你对 TypeScript 装饰器的理解？应用场景？



### 9.1. 是什么

装饰器是一种特殊类型的声明，它能够被附加到类声明，方法，访问符，属性或参数上  
是一种在不改变原类和使用继承的情况下，动态地扩展对象功能

同样的，本质也不是什么高大上的结构，就是一个普通的函数，`@expression` 的形式其实是  
`Object.defineProperty` 的语法糖

`expression` 求值后必须也是一个函数，它会在运行时被调用，被装饰的声明信息做为参数传入

## 9.2. 使用方式

由于 `typescript` 是一个实验性特性，若要使用，需要在 `tsconfig.json` 文件启动，如下：

```
1 {
2     "compilerOptions": {
3         "target": "ES5",
4         "experimentalDecorators": true
5     }
6 }
```

`typescript` 装饰器的使用和 `javascript` 基本一致

类的装饰器可以装饰：

- 类
- 方法/属性
- 参数
- 访问器

### 9.2.1. 类装饰

例如声明一个函数 `addAge` 去给 Class 的属性 `age` 添加年龄。

```
1 function addAge(constructor: Function) {
2     constructor.prototype.age = 18;
3 }
4 @addAge
5 class Person{
6     name: string;
7     age!: number;
8     constructor() {
9         this.name = 'huihui';
10    }
11 }
12 let person = new Person();
13 console.log(person.age); // 18
```

上述代码，实际等同于以下形式：

```
1 Person = addAge(function Person() { ... });
```

上述可以看到，当装饰器作为修饰类的时候，会把构造器传递进去。

`constructor.prototype.age` 就是在每一个实例化对象上面添加一个 `age` 属性

## 9.2.2. 方法/属性装饰

同样，装饰器可以用于修饰类的方法，这时候装饰器函数接收的参数变成了：

- `target`: 对象的原型
- `propertyKey`: 方法的名称
- `descriptor`: 方法的属性描述符

可以看到，这三个属性实际就是 `Object.defineProperty` 的三个参数，如果是类的属性，则没有传递第三个参数

如下例子：

```
1 // 声明装饰器修饰方法/属性
2 function method(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
3     console.log(target);
4     console.log("prop " + propertyKey);
5     console.log("desc " + JSON.stringify(descriptor) + "\n\n");
6     descriptor.writable = false;
7 }
8 function property(target: any, propertyKey: string) {
9     console.log("target", target)
10    console.log("propertyKey", propertyKey)
11 }
12 class Person{
13     @property
14     name: string;
15     constructor() {
16         this.name = 'huihui';
17     }
18     @method
19     say(){
20         return 'instance method';
21     }
22     @method
23     static run(){
24         return 'static method';
25     }
26 }
27 const xmz = new Person();
```

```
28 // 修改实例方法say
29 xmz.say = function() {
30   return 'edit'
31 }
```

输出如下图所示：

```
target ►{constructor: f, say: f}                                index.ts:12
propertyKey name                                              index.ts:13
►{constructor: f, say: f}                                index.ts:5
prop say                                              index.ts:6
desc {"writable":true,"enumerable":false,"configurable":true} index.ts:7

class Person {
  constructor() {
    this.name = 'xiaomuzhu';
  }
  say() {
    return 'instance method';
  }
  static run() {
    return 'static method';
  }
}
prop run                                              index.ts:6
desc {"writable":true,"enumerable":false,"configurable":true} index.ts:7

►Uncaught TypeError: Cannot assign to read only property 'say' of object '#<Person>'      index.ts:37
  at index.ts:37
```

### 9.2.3. 参数装饰

接收3个参数，分别是：

- target：当前对象的原型
- propertyKey：参数的名称
- index：参数数组中的位置

```
1 function logParameter(target: Object, propertyName: string, index: number) {
2   console.log(target);
3   console.log(propertyName);
4   console.log(index);
5 }
6 class Employee {
7   greet(@logParameter message: string): string {
8     return
9     hello ${message}
10   ;
11 }
12 }
13 const emp = new Employee();
```

```
14 emp.greet('hello');
```

输入如下图：

```
▼ Object 1
  ► constructor: class Employee
  ► greet: f greet(message)
  ► [[Prototype]]: Object
greet
0
```

## 9.2.4. 访问器装饰

使用起来方式与方法装饰一致，如下：

```
1
2 function modification(target: Object, propertyKey: string, descriptor: 
  PropertyDescriptor) {
3   console.log(target);
4   console.log("prop " + propertyKey);
5   console.log("desc " + JSON.stringify(descriptor) + "\n\n");
6 }
7 class Person{
8   _name: string;
9   constructor() {
10     this._name = 'huihui';
11   }
12   @modification
13   get name() {
14     return this._name
15   }
16 }
```

## 9.2.5. 装饰器工厂

如果想要传递参数，使装饰器变成类似工厂函数，只需要在装饰器函数内部再函数一个函数即可，如下：

```
1 function addAge(age: number) {
2   return function(ctor: Function) {
3     ctor.prototype.age = age
4   }
5 }
6 @addAge(10)
7 class Person{
```

```
8  name: string;
9  age!: number;
10 constructor() {
11   this.name = 'huihui';
12 }
13 }
14 let person = new Person();
```

## 9.2.6. 执行顺序

当多个装饰器应用于一个声明上，将由上至下依次对装饰器表达式求值，求值的结果会被当作函数，由下至上依次调用，例如如下：

```
1 function f() {
2   console.log("f(): evaluated");
3   return function (target, propertyKey: string, descriptor:
4     PropertyDescriptor) {
5     console.log("f(): called");
6   }
7 function g() {
8   console.log("g(): evaluated");
9   return function (target, propertyKey: string, descriptor:
10    PropertyDescriptor) {
11     console.log("g(): called");
12   }
13 class C {
14   @f()
15   @g()
16   method() {}
17 }
18 // 输出
19 f(): evaluated
20 g(): evaluated
21 g(): called
22 f(): called
```

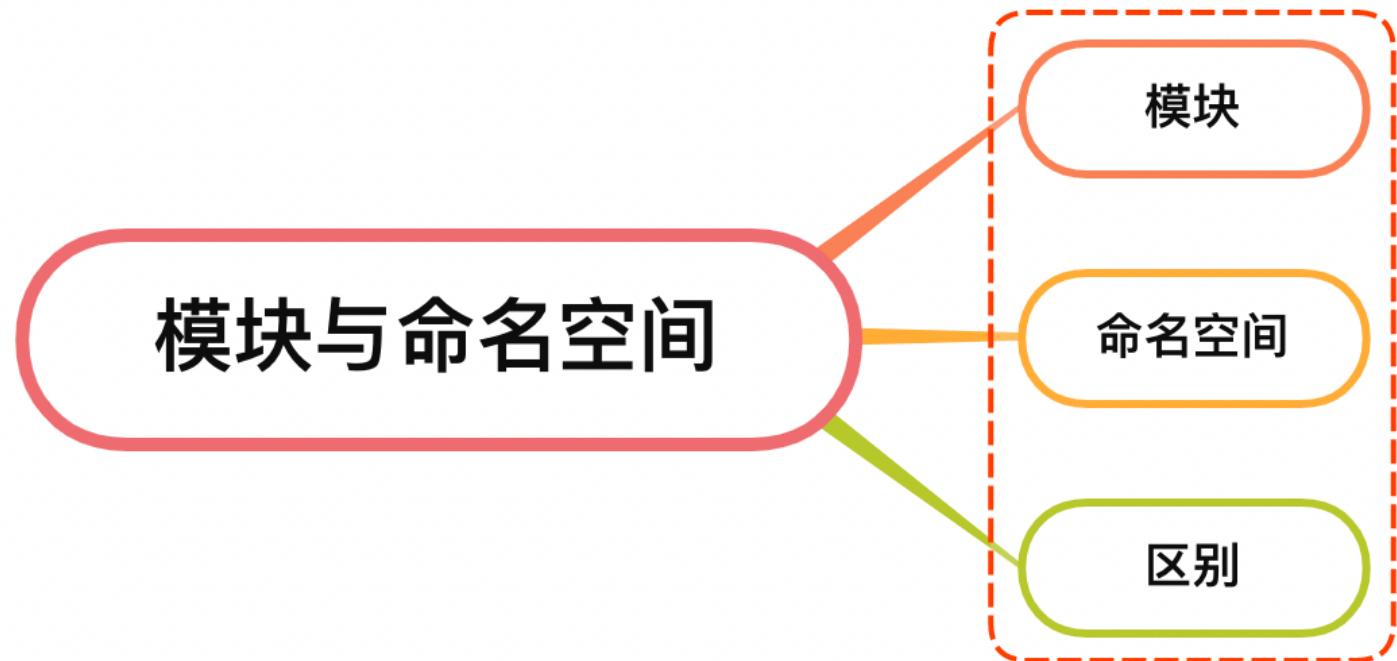
## 9.3. 应用场景

可以看到，使用装饰器存在两个显著的优点：

- 代码可读性变强了，装饰器命名相当于一个注释
- 在不改变原有代码情况下，对原来功能进行扩展

后面的使用场景中，借助装饰器的特性，除了提高可读性之后，针对已经存在的类，可以通过装饰器的特性，在不改变原有代码情况下，对原来功能进行扩展

## 10. 说说对 TypeScript 中命名空间与模块的理解？区别？



### 10.1. 模块

TypeScript 与 ECMAScript 2015 一样，任何包含顶级 `import` 或者 `export` 的文件都被当成一个模块

相反地，如果一个文件不带有顶级的 `import` 或者 `export` 声明，那么它的内容被视为全局可见的。例如我们在在一个 TypeScript 工程下建立一个文件 `1.ts`，声明一个变量 `a`，如下：

```
1 const a = 1
```

然后在另一个文件同样声明一个变量 `a`，这时候会出现错误信息

```
const a = 20;
```

无法重新声明块范围变量“a”。 ts(2451)

1.ts(3, 7): 此处也声明了 "a"。

```
const a: 20
```

[查看问题 \(F8\)](#) 没有可用的快速修复

提示重复声明 `a` 变量，但是所处的空间是全局的

如果需要解决这个问题，则通过 `import` 或者 `export` 引入模块系统即可，如下：

```
1 const a = 10;
2 export default a
```

在 `typescript` 中，`export` 关键字可以导出变量或者类型，用法与 `es6` 模块一致，如下：

```
1 export const a = 1
2 export type Person = {
3     name: String
4 }
```

通过 `import` 引入模块，如下：

```
1 import { a, Person } from './export';
```

## 10.2. 命名空间

命名空间一个最明确的目的就是解决重名问题

命名空间定义了标识符的可见范围，一个标识符可在多个名字空间中定义，它在不同名字空间中的含义是互不相干的

这样，在一个新的名字空间中可定义任何标识符，它们不会与任何已有的标识符发生冲突，因为已有的定义都处于其他名字空间中

TypeScript 中命名空间使用 `namespace` 来定义，语法格式如下：

```
1 namespace SomeNameSpaceName {  
2     export interface ISomeInterfaceName { }  
3     export class SomeClassName { }  
4 }
```

以上定义了一个命名空间 `SomeNameSpaceName`，如果我们需要在外部可以调用

`SomeNameSpaceName` 中的类和接口，则需要在类和接口添加 `export` 关键字

使用方式如下：

```
1 SomeNameSpaceName.SomeClassName
```

命名空间本质上是一个对象，作用是将一系列相关的全局变量组织到一个对象的属性，如下：

```
1 namespace Letter {  
2     export let a = 1;  
3     export let b = 2;  
4     export let c = 3;  
5     // ...  
6     export let z = 26;  
7 }
```

编译成 `js` 如下：

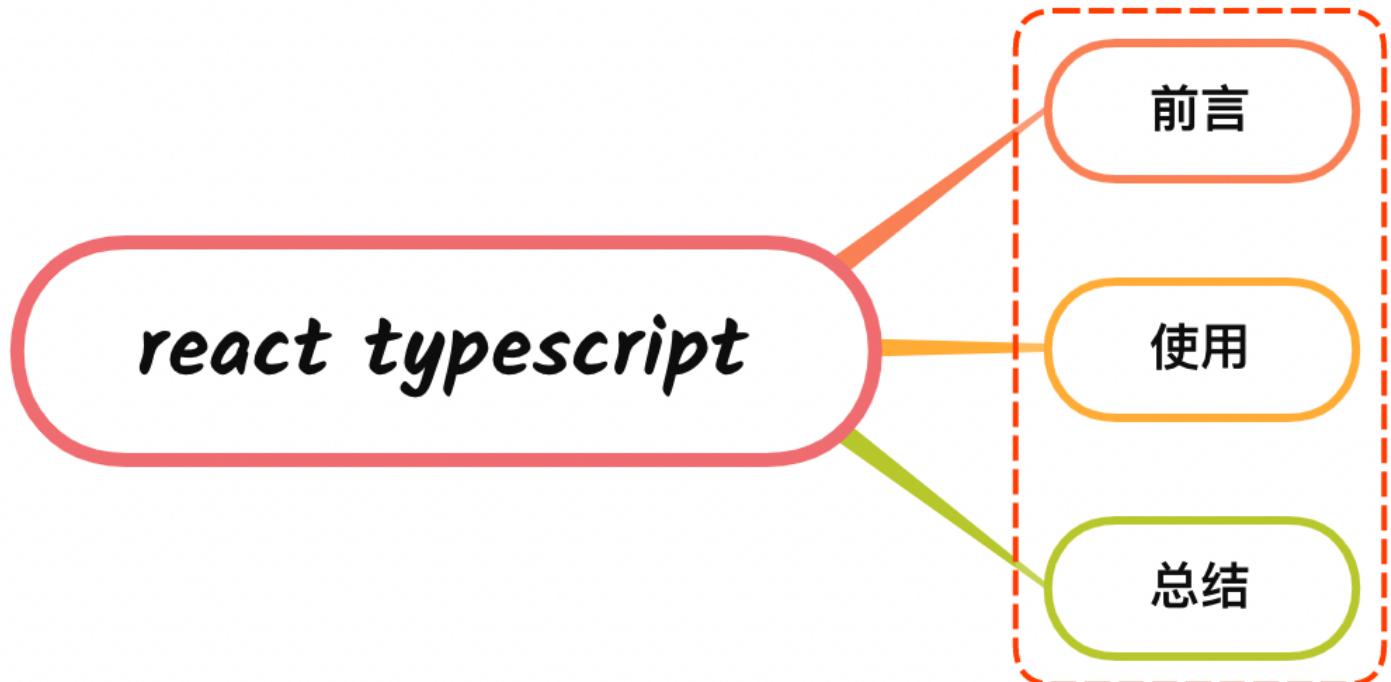
```
1 var Letter;  
2 (function (Letter) {  
3     Letter.a = 1;  
4     Letter.b = 2;  
5     Letter.c = 3;  
6     // ...  
7     Letter.z = 26;  
8 })(Letter || (Letter = {}));
```

### 10.3. 区别

- 命名空间是位于全局命名空间下的一个普通的带有名字的 JavaScript 对象，使用起来十分容易。但就像其它的全局命名空间污染一样，它很难去识别组件之间的依赖关系，尤其是在大型的应用中

- 像命名空间一样，模块可以包含代码和声明。不同的是模块可以声明它的依赖
- 在正常的TS项目开发过程中并不建议用命名空间，但通常在通过 d.ts 文件标记 js 库类型的时候使用命名空间，主要作用是给编译器编写代码的时候参考使用

## 11. 说说如何在 React 项目中应用 TypeScript?



### 11.1. 前言

单独的使用 `TypeScript` 并不会导致学习成本很高，但是绝大部分前端开发者的项目都是依赖于框架的

例如与 `Vue`、`React` 这些框架结合使用的时候，会有一定的门槛

使用 `TypeScript` 编写 `React` 代码，除了需要 `TypeScript` 这个库之外，还需要安装 `@types/react`、`@types/react-dom`

```
1 npm i @types/react -s
2 npm i @types/react-dom -s
```

至于上述使用 `@types` 的库的原因在于，目前非常多的 `JavaScript` 库并没有提供自己关于 `TypeScript` 的声明文件

所以，`ts` 并不知道这些库的类型以及对应导出的内容，这里 `@types` 实际就是社区中的 `DefinitelyTyped` 库，定义了目前市面上绝大多数的 `JavaScript` 库的声明

所以下载相关的 `JavaScript` 对应的 `@types` 声明时，就能够使用使用该库对应的类型定义

## 11.2. 使用方式

在编写 `React` 项目的时候，最常见的使用的组件就是：

- 无状态组件
- 有状态组件
- 受控组件

### 11.2.1. 无状态组件

主要作用是用于展示 `UI`，如果使用 `js` 声明，则如下所示：

```
1 import * as React from "React";
2 export const Logo = (props) => {
3   const { logo, className, alt } = props;
4   return <img src={logo} className={className} alt={alt} />;
5 };
```

但这时候 `ts` 会出现报错提示，原因在于没有定义 `props` 类型，这时候就可以使用 `interface` 接口去定义 `props` 即可，如下：

```
1 import * as React from "React";
2 interface IProps {
3   logo?: string;
4   className?: string;
5   alt?: string;
6 }
7 export const Logo = (props: IProps) => {
8   const { logo, className, alt } = props;
9   return <img src={logo} className={className} alt={alt} />;
10 };
```

但是我们都知道 `props` 里面存在 `children` 属性，我们不可能每个 `props` 接口里面定义多一个 `children`，如下：

```
1 interface IProps {
2   logo?: string;
3   className?: string;
4   alt?: string;
5   children?: ReactNode;
6 }
```

更加规范的写法是使用 `React` 里面定义好的 `FC` 属性，里面已经定义好 `children` 类型，如下：

```
1 export const Logo: React.FC<IProps> = (props) => {
2   const { logo, className, alt } = props;
3   return <img src={logo} className={className} alt={alt} />;
4 };
```

- `React.FC` 显式地定义了返回类型，其他方式是隐式推导的
- `React.FC` 对静态属性：`displayName`、`propTypes`、`defaultProps` 提供了类型检查和自动补全
- `React.FC` 为 `children` 提供了隐式的类型 (`ReactElement | null`)

### 11.2.2. 有状态组件

可以是一个类组件且存在 `props` 和 `state` 属性

如果使用 `TypeScript` 声明则如下所示：

```
1 import * as React from "React";
2 interface IProps {
3   color: string;
4   size?: string;
5 }
6 interface IState {
7   count: number;
8 }
9 class App extends React.Component<IProps, IState> {
10   public state = {
11     count: 1,
12   };
13   public render() {
14     return <div>Hello world</div>;
15   }
16 }
```

上述通过泛型对 `props`、`state` 进行类型定义，然后在使用的时候就可以在编译器中获取更好的智能提示

关于 `Component` 泛型类的定义，可以参考下 `React` 的类型定义文件 `node_modules/@types/React/index.d.ts`，如下所示：

```
1 class Component<P, S> {
2   readonly props: Readonly<{ children?: ReactNode }> & Readonly<P>;
3   state: Readonly<S>;
4 }
```

从上述可以看到，`state` 属性也定义了可读类型，目的是为了防止直接调用 `this.state` 更新状态

### 11.2.3. 受控组件

受控组件的特性在于元素的内容通过组件的状态 `state` 进行控制

由于组件内部的事件是合成事件，不等同于原生事件，

例如一个 `input` 组件修改内部的状态，常见的定义的时候如下所示：

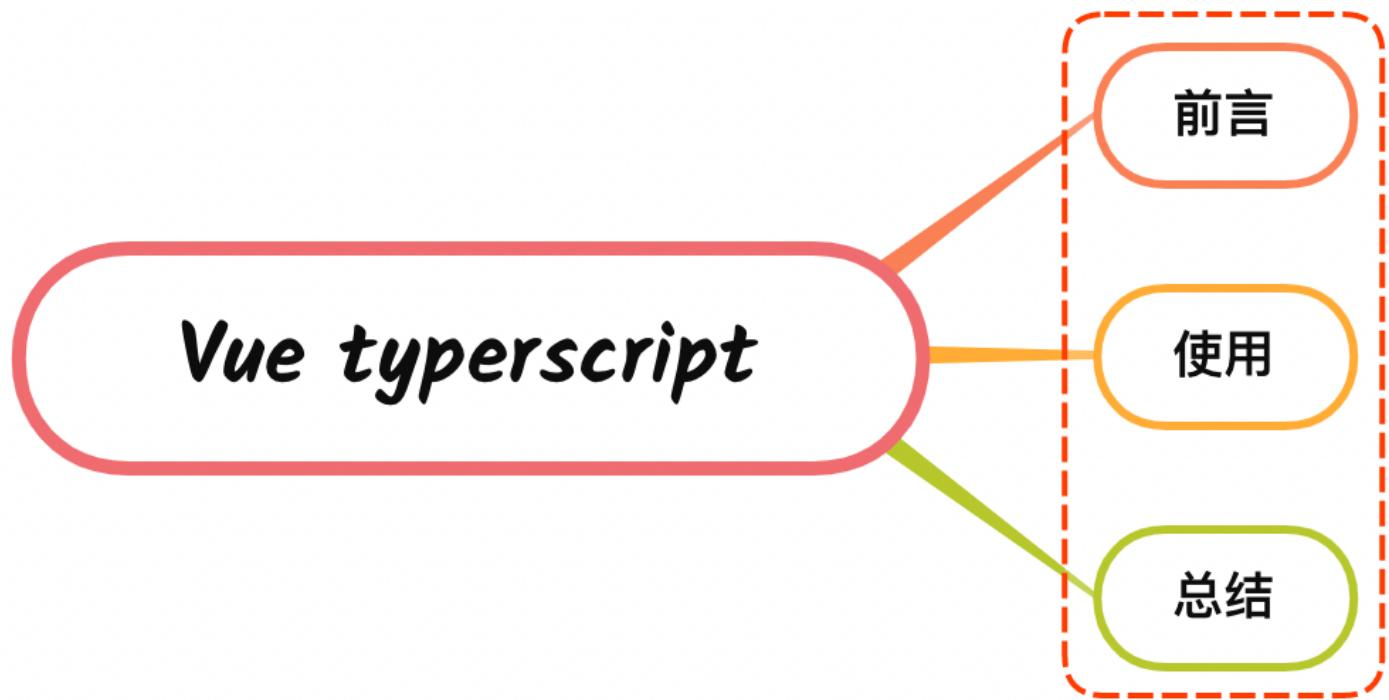
```
1 private updateValue(e: React.ChangeEvent<HTMLInputElement>) {
2   this.setState({ itemText: e.target.value })
3 }
```

常用 `Event` 事件对象类型：

- `ClipboardEvent<T = Element>` 剪贴板事件对象
- `DragEvent<T = Element>` 拖拽事件对象
- `ChangeEvent<T = Element>` Change 事件对象
- `KeyboardEvent<T = Element>` 键盘事件对象
- `MouseEvent<T = Element>` 鼠标事件对象
- `TouchEvent<T = Element>` 触摸事件对象
- `WheelEvent<T = Element>` 滚轮事件对象
- `AnimationEvent<T = Element>` 动画事件对象
- `TransitionEvent<T = Element>` 过渡事件对象

`T` 接收一个 `DOM` 元素类型

## 12. 说说如何在Vue项目中应用TypeScript?



## 12.1. 前言

与link类似

在 VUE 项目中应用 typescript，我们需要引入一个库 vue-property-decorator，其是基于 vue-class-component 库而来，这个库 vue 官方推出的一个支持使用 class 方式来开发 vue 单文件组件的库

主要的功能如下：

- methods 可以直接声明为类的成员方法
- 计算属性可以被声明为类的属性访问器
- 初始化的 data 可以被声明为类属性
- data、render 以及所有的 Vue 生命周期钩子可以直接作为类的成员方法
- 所有其他属性，需要放在装饰器中

## 12.2. 使用

vue-property-decorator 主要提供了多个装饰器和一个函数：

### 12.2.1. @Component

Component 装饰器它注明了此类为一个 Vue 组件，因此即使没有设置选项也不能省略

如果需要定义比如 name、components、filters、directives 以及自定义属性，就可以在 Component 装饰器中定义，如下：

```
1 import {Component,Vue} from 'vue-property-decorator';
```

```
2 import {componentA,componentB} from '@/components';
3 @Component({
4     components:{
5         componentA,
6         componentB,
7     },
8     directives: {
9         focus: {
10             // 指令的定义
11             inserted: function (el) {
12                 el.focus()
13             }
14         }
15     }
16 })
17 export default class YourCompoent extends Vue{
18 }
```

## 12.2.2. computed、data、methods

这里取消了组件的data和methods属性，以往data返回对象中的属性、methods中的方法需要直接定义在Class中，当做类的属性和方法

```
1 @Component
2 export default class HelloDecorator extends Vue {
3     count: number = 123 // 类属性相当于以前的 data
4     add(): number { // 类方法就是以前的方法
5         this.count + 1
6     }
7     // 获取计算属性
8     get total(): number {
9         return this.count + 1
10    }
11    // 设置计算属性
12    set total(param:number): void {
13        this.count = param
14    }
15 }
```

## 12.2.3. @props

组件接收属性的装饰器，如下使用：

```
1 import {Component, Vue, Prop} from vue-property-decorator;
2 @Component
3 export default class YourComponent extends Vue {
4     @Prop(String)
5     propA:string;
6     @Prop([String,Number])
7     propB:string|number;
8     @Prop({
9         type: String, // type: [String , Number]
10        default: 'default value', // 一般为String或Number
11        //如果是对象或数组的话。默认值从一个工厂函数中返回
12        // defatult: () => {
13        //     return ['a', 'b']
14        // }
15        required: true,
16        validator: (value) => {
17            return [
18                'InProcess',
19                'Settled'
20            ].indexOf(value) !== -1
21        }
22    })
23    propC:string;
24 }
```

## 12.2.4. @watch

实际就是 `Vue` 中的监听器，如下：

```
1 import { Vue, Component, Watch } from 'vue-property-decorator'
2 @Component
3 export default class YourComponent extends Vue {
4     @Watch('child')
5     onChildChanged(val: string, oldVal: string) {}
6     @Watch('person', { immediate: true, deep: true })
7     onPersonChanged1(val: Person, oldVal: Person) {}
8     @Watch('person')
9     onPersonChanged2(val: Person, oldVal: Person) {}
10 }
```

## 12.2.5. @emit

`vue-property-decorator` 提供的 `@Emit` 装饰器就是代替 `Vue` 中的事件的触发 `$emit`，如下：

```
1 import {Vue, Component, Emit} from 'vue-property-decorator';
2     @Component({})
3     export default class Some extends Vue{
4         mounted(){
5             this.$on('emit-todo', function(n) {
6                 console.log(n)
7             })
8             this.emitTodo('world');
9         }
10        @Emit()
11        emitTodo(n: string){
12            console.log('hello');
13        }
14    }
```

## 12.3. 总结

可以看到上述 `typescript` 版本的 `vue class` 的语法与平时 `javascript` 版本使用起来还是有很大的不同，多处用到 `class` 与装饰器，但实际上本质是一致的，只有不断编写才会得心应手