## **一、ts的概念**

ts是js的超集

它强调了 TypeScript 的两个最重要的特性——类型系统、适用于任何规模。

「类型」是其最核心的特性。

ts是静态类型（在编译时才会类型检查），而js是动态类型（在运行时才会类型检查）

ts和js一样都是弱语言，即允许隐式类型转换

TypeScript 可以和 JavaScript 共存，这意味着 JavaScript 项目能够渐进式的迁移到 TypeScript。

TypeScript 可以编译为 JavaScript，然后运行在浏览器、Node.js 等任何能运行 JavaScript 的环境中。

TypeScript 编译的时候即使报错了，还是会生成编译结果，我们仍然可以使用这个编译之后的文件。

## **二、简单使用**

在 TypeScript 中，我们使用 : 指定变量的类型，: 的前后有没有空格都可以。

function sayHello(person: string) {  
    return 'Hello, ' + person;  
}  
let user = 'Tom';  
console.log(sayHello(user));

执行以下命令，生成一个编译好的js文件

tsc xxx.ts

也可以全局安装

npm install -g typescript

执行以下命令，不会生成js文件，而是直接在控制台输出结果

## **三、类型**

### **1.空值**

JavaScript 没有空值（Void）的概念，在 TypeScript 中，可以用 void 表示没有任何返回值的函数：

function alertName(): void {  
    alert('My name is Tom');  
}

用void表示的变量，只能被赋值为 undefined 和 null（只在tsconfig.json中的 --strictNullChecks 未指定时）

### **2.Null 和 Undefined**

在 TypeScript 中，可以使用 null 和 undefined 来定义这两个原始数据类型

let u: undefined = undefined;  
let n: null = null;

与 void 的区别是，undefined 和 null 是所有类型的子类型。也就是说 undefined 类型的变量，可以赋值给 任意类型的变量：

let num: number = undefined;  
// 这样也不会报错  
let u: undefined;  
let num: number = u;

注意：此处如果要将undefined赋值为任意类型的变量，需要在tsconfig.json文件中设置strict为false

### **3.数组**

有两种方式可以定义数组。 第一种，可以在元素类型后面接上 []，表示由此类型元素组成的一个数组：

let list: number[] = [1, 2, 3];

第二种方式是使用数组泛型，Array<元素类型>：

let list: Array<number> = [1, 2, 3];

### **4.元组Tuple**

元组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。 比如，你可以定义一对值分别为 string和number类型的元组。

let x: [string, number];  
// Initialize it  
x = ['hello', 10]; // OK  
// Initialize it incorrectly  
x = [10, 'hello']; // Error

x.push(1)

x.push(‘2’)

console.log(x);

当访问超出元组长度的元素时，它的类型会被限制为元组中每个类型的联合类型

### **5.枚举类型**

使用枚举类型可以为一组数值赋予友好的名字。

默认情况下，从0开始为元素编号。 你也可以手动的指定成员的数值。 例如：

enum Color {Red = 1, Green, Blue}  
let c: Color = Color.Green;  
console.log(c);//2

或者，全部都采用手动赋值：

enum Color {Red = 1, Green = 2, Blue = 4}  
let c: Color = Color.Green;//2

枚举类型提供的一个便利是你可以由枚举的值得到它的名字。 例如，我们知道数值为2，但是不确定它映射到Color里的哪个名字，我们可以查找相应的名字：

enum Color {Red = 1, Green, Blue}  
let colorName: string = Color[2];  
​  
console.log(colorName);  // 显示'Green'因为上面代码里它的值是2

### **6.任意类型**

在赋值过程中改变类型是不被允许的

但是如果是any类型，则允许被赋值为任意类型

let myFavoriteNumber: any = 'seven';  
myFavoriteNumber = 7;  
console.log(myFavoriteNumber);//7  
​  
myFavoriteNumber='hhh'  
console.log(myFavoriteNumber);//'hhh'  
​  
myFavoriteNumber=false  
console.log(myFavoriteNumber);//false  
​  
myFavoriteNumber=[1,2,5]  
console.log(myFavoriteNumber);//[1,2,5]

在任意值上访问任何属性和方法都是允许的：

let anyThing: any = 'hello';  
console.log(anyThing.myName);  
console.log(anyThing.myName.firstName);

注意：ts是强类型的，在编译时需要类型检查，any的作用是在有些对象还不明确推测出类型时，编译时跳过类型检查

### **7.未声明类型的变量**

变量如果在声明的时候，未指定其类型，那么它会被识别为任意值类型

与js定义变量基本一致

### **8.联合类型**

联合类型（Union Types）表示取值可以为多种类型中的一种。

联合类型使用 | 分隔每个类型。

例子：

let myFavoriteNumber: string | number;  
myFavoriteNumber = 'seven';  
myFavoriteNumber = 7;

let myFavoriteNumber: string | number;  
myFavoriteNumber = true;  
​  
// index.ts(2,1): error TS2322: Type 'boolean' is not assignable to type 'string | number'.  
//   Type 'boolean' is not assignable to type 'number'.

#### **（1）访问联合类型的属性或方法**

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们只能访问此联合类型的所有类型里共有的属性或方法：

function getLength(something: string | number): number {  
    return something.length;  
}  
​  
// index.ts(2,22): error TS2339: Property 'length' does not exist on type 'string | number'.  
//   Property 'length' does not exist on type 'number'.

上例中，length 不是 string 和 number 的共有属性，所以会报错。

但是访问 string 和 number 的共有属性是没问题的：

function getString(something: string | number): string {  
    return something.toString();  
}

### **9.接口类型**

在 TypeScript 中，我们使用接口（Interfaces）来定义对象的类型。

在面向对象语言中，接口（Interfaces）是一个很重要的概念，它是对行为的抽象，而具体如何行动需要由类（classes）去实现（implement）。

TypeScript 中的接口是一个非常灵活的概念，除了可用于[对类的一部分行为进行抽象](https://ts.xcatliu.com/advanced/class-and-interfaces.html" \l "%E7%B1%BB%E5%AE%9E%E7%8E%B0%E6%8E%A5%E5%8F%A3)以外，也常用于对「对象的形状（Shape）」进行描述。

#### **（1）简单例子**

定义了一个接口 Person，接着定义了一个变量 tom，它的类型是 Person。这样，我们就约束了 tom 的形状必须和接口 Person 一致。

interface Person {  
    name: string;  
    age: number;  
}  
​  
let tom: Person = {  
    name: 'Tom',  
    age: 25  
};

接口一般首字母大写。有的编程语言中会建议接口的名称加上 I 前缀

定义的变量比接口少了一些属性，或者多一些属性都是不允许的

#### **（2）可选属性**

有时我们希望不要完全匹配一个形状，那么可以用可选属性：

interface Person {  
    name: string;  
    age?: number;  
}  
​  
let tom: Person = {  
    name: 'Tom'  
};

这时仍然不允许在变量中添加未定义的属性

#### **（3）任意属性**

有时候我们希望一个接口允许有任意的属性，可以使用如下方式：

使用 [propName: string] 定义了任意属性取 string 类型的值。

需要注意的是，一旦定义了任意属性，那么确定属性和可选属性的类型都必须是它的类型的子集：

interface Person {  
    name: string;  
    age?: number;  
   [propName: string]: any;  
}  
​  
let tom: Person = {  
    name: 'Tom',  
    gender: 'male'  
};

如果接口中有多个类型的属性，则可以在任意属性中使用联合类型：

interface Person {  
    name: string;  
    age?: number;  
   [propName: string]: string | number;  
}  
​  
let tom: Person = {  
    name: 'Tom',  
    age: 25,  
    gender: 'male'  
};

#### **（4）只读属性**

有时候我们希望对象中的一些字段只能在创建的时候被赋值，那么可以用 readonly 定义只读属性：

interface Person {  
    readonly id: number;  
    name: string;  
    age?: number;  
   [propName: string]: any;  
}  
​  
let tom: Person = {  
    id: 89757,  
    name: 'Tom',  
    gender: 'male'  
};  
​  
tom.id = 9527;  
​  
// index.ts(14,5): error TS2540: Cannot assign to 'id' because it is a constant or a read-only property.

注意，只读的约束存在于第一次给对象赋值的时候，而不是第一次给只读属性赋值的时候,给只读属性赋值会报错

interface Person {  
    readonly id: number;  
    name: string;  
    age?: number;  
   [propName: string]: any;  
}  
​  
let tom: Person = {  
    name: 'Tom',  
    gender: 'male'  
};  
​  
tom.id = 89757;  
​  
// index.ts(8,5): error TS2322: Type '{ name: string; gender: string; }' is not assignable to type 'Person'.  
//   Property 'id' is missing in type '{ name: string; gender: string; }'.  
// index.ts(13,5): error TS2540: Cannot assign to 'id' because it is a constant or a read-only property.

上例中，报错信息有两处，第一处是在对 tom 进行赋值的时候，没有给 id 赋值。

第二处是在给 tom.id 赋值的时候，由于它是只读属性，所以报错了。

#### **（5）接口的合并**

接口中的属性在合并时会简单的合并到一个接口中：

interface Alarm {  
    price: number;  
}  
interface Alarm {  
    weight: number;  
}

相当于

interface Alarm {  
    price: number;  
    weight: number;  
}

合并的属性的类型必须是一致的：

interface Alarm {  
    price: number;  
}  
interface Alarm {  
    price: number;  // 虽然重复了，但是类型都是 `number`，所以不会报错  
    weight: number;  
}

interface Alarm {  
    price: number;  
}  
interface Alarm {  
    price: string;  // 类型不一致，会报错  
    weight: number;  
}  
​  
// index.ts(5,3): error TS2403: Subsequent variable declarations must have the same type. Variable 'price' must be of type 'number', but here has type 'string'.

其中方法的合并，与函数的合并一样

interface Alarm {  
    price: number;  
    alert(s: string): string;  
}  
interface Alarm {  
    weight: number;  
    alert(s: string, n: number): string;  
}

相当于：

interface Alarm {  
    price: number;  
    weight: number;  
    alert(s: string): string;  
    alert(s: string, n: number): string;  
}

### **10.数组类型**

#### **（1）常见定义**

在 TypeScript 中，数组类型有多种定义方式，比较灵活。

最简单的方法是使用「类型 + 方括号」来表示数组：

let arr: number[] = [1, 1, 2, 3, 5];

#### **（2）数组泛型**

我们也可以使用数组泛型（Array Generic） Array<elemType> 来表示数组：

let arr: Array<number> = [1, 1, 2, 3, 5];

#### **（3）接口定义**

interface NumberArray {  
   [index: number]: number;  
}  
let fibonacci: NumberArray = [1, 1, 2, 3, 5];

NumberArray 表示：只要索引的类型是数字时，那么值的类型必须是数字。

虽然接口也可以用来描述数组，但是我们一般不会这么做，因为这种方式比前两种方式复杂多了。

#### **（4）类数组**

类数组（Array-like Object）不是数组类型，比如 arguments：

不能用普通的数组的方式来描述，而应该用接口：

function sum() {  
    let args: {  
       [index: number]: number;  
        length: number;  
        callee: Function;  
   } = arguments;  
}

在这个例子中，我们除了约束当索引的类型是数字时，值的类型必须是数字之外，也约束了它还有 length 和 callee 两个属性。

事实上常用的类数组都有自己的接口定义，如 IArguments, NodeList, HTMLCollection 等：

function sum() {  
    let args: IArguments = arguments;  
}

其中 IArguments 是 TypeScript 中定义好了的类型，它实际上就是：

interface IArguments {  
   [index: number]: any;  
    length: number;  
    callee: Function;  
}

### **11.函数类型**

#### **（1）函数声明**

一个函数有输入和输出，要在 TypeScript 中对其进行约束，需要把输入和输出都考虑到，其中函数声明的类型定义较简单：

function sum(x: number, y: number): number {  
    return x + y;  
}

注意，输入多余的（或者少于要求的）参数，是不被允许的

#### **（2）函数表达式**

let mySum = function (x: number, y: number): number {  
    return x + y;  
};

这是可以通过编译的，不过事实上，上面的代码只对等号右侧的匿名函数进行了类型定义，而等号左边的 mySum，是通过赋值操作进行类型推论而推断出来的。如果需要我们手动给 mySum 添加类型，则应该是这样：

let mySum: (x: number, y: number) => number = function (x: number, y: number): number {  
    return x + y;  
};

注意不要混淆了 TypeScript 中的 => 和 ES6 中的 =>。

在 TypeScript 的类型定义中，=> 用来表示函数的定义，左边是输入类型，需要用括号括起来，右边是输出类型。

在 ES6 中，=> 叫做箭头函数，应用十分广泛

#### **（3）用接口定义函数的形状**

interface SearchFunc {  
   (source: string, subString: string): boolean;  
}  
let mySearch: SearchFunc;  
mySearch = function(source: string, subString: string) {  
    return source.search(subString) !== -1;  
}

#### **（4）可选参数**

function buildName(firstName: string, lastName?: string) {  
    if (lastName) {  
        return firstName + ' ' + lastName;  
   } else {  
        return firstName;  
   }  
}  
let tomcat = buildName('Tom', 'Cat');  
let tom = buildName('Tom');

可选参数必须接在必需参数后面。换句话说，可选参数后面不允许再出现必需参数了

#### **（5）参数默认值**

TypeScript 会将添加了默认值的参数识别为可选参数

此时就不受「可选参数必须接在必需参数后面」的限制了：

function buildName(firstName: string = 'Tom', lastName: string) {  
    return firstName + ' ' + lastName;  
}  
let tomcat = buildName('Tom', 'Cat');  
let cat = buildName(undefined, 'Cat');

但是此时就必须实参写全

#### **（6）剩余参数**

ES6 中，可以使用 ...rest 的方式获取函数中的剩余参数（rest 参数）：

我们可以用数组的类型来定义剩余参数：

function push(array: any[], ...items: any[]) {  
    items.forEach(function(item) {  
        array.push(item);  
   });  
}  
​  
let a = [];  
push(a, 1, 2, 3);

注意，rest 参数只能是最后一个参数

#### **（7）重载**

重载允许一个函数接受不同数量或类型的参数时，作出不同的处理。

通常和联合类型一起使用

function reverse(x: number): number;  
function reverse(x: string): string;  
function reverse(x: number | string): number | string | void {  
    if (typeof x === 'number') {  
        return Number(x.toString().split('').reverse().join(''));  
   } else if (typeof x === 'string') {  
        return x.split('').reverse().join('');  
   }  
}

注意，TypeScript 会优先从最前面的函数定义开始匹配，所以多个函数定义如果有包含关系，需要优先把精确的定义写在前面。

该特性不常使用

## **四、类型推断和类型断言**

### **1.类型推断**

如果没有明确的指定类型，那么 TypeScript 会依照类型推论（Type Inference）的规则推断出一个类型。

如果定义的时候没有赋值，不管之后有没有赋值，都会被推断成 any 类型而完全不被类型检查：

let myFavoriteNumber;  
myFavoriteNumber = 'seven';  
myFavoriteNumber = 7;

### **2.类型断言**

用来手动指定一个值的类型。

#### **（1）语法**

值 as 类型

或

<类型>值

在 tsx 语法（React 的 jsx 语法的 ts 版）中必须使用前者，即 值 as 类型。

第二种语法在ts中不仅表示类型断言外，还可能表示一个泛型

故建议大家在使用类型断言时，统一使用 值 as 类型 这样的语法

#### **（2）将一个联合类型断言为其中一个类型**

interface Cat {  
    name: string;  
    run(): void;  
}  
interface Fish {  
    name: string;  
    swim(): void;  
}  
​  
function isFish(animal: Cat | Fish) {  
    if (typeof (animal as Fish).swim === 'function') {  
        return true;  
   }  
    return false;  
}

使用该方法会避免编译报错，但是运行时如果没有对应的属性依然会报错

使用类型断言时一定要格外小心，尽量避免断言后调用方法或引用深层属性，以减少不必要的运行时错误。

#### **（3）父类可以被断言为子类**

class ApiError extends Error {  
    code: number = 0;  
}  
class HttpError extends Error {  
    statusCode: number = 200;  
}  
​  
function isApiError(error: Error) {  
    if (typeof (error as ApiError).code === 'number') {  
        return true;  
   }  
    return false;  
}

#### **（4）将任何一个类型断言为 any**

在 any 类型的变量上，访问任何属性都是允许的。

(window as any).foo = 1;

否则会报错

window.foo = 1;  
​  
// index.ts:1:8 - error TS2339: Property 'foo' does not exist on type 'Window & typeof globalThis'.

需要注意的是，将一个变量断言为 any 可以说是解决 TypeScript 中类型问题的最后一个手段。

它极有可能掩盖了真正的类型错误，所以如果不是非常确定，就不要使用 as any。

#### **（5）将 any 断言为一个具体的类型**

通过类型断言及时的把 any 断言为精确的类型

function getCacheData(key: string): any {  
    return (window as any).cache[key];  
}  
​  
interface Cat {  
    name: string;  
    run(): void;  
}  
​  
const tom = getCacheData('tom') as Cat;  
tom.run();

#### **（6）类型断言的限制**

TypeScript 是结构类型系统，类型之间的对比只会比较它们最终的结构，而会忽略它们定义时的关系。

不管定义时的关系，只看它们的结构关系

interface Animal {  
    name: string;  
}  
interface Cat {  
    name: string;  
    run(): void;  
}

相当于

interface Animal {  
    name: string;  
}  
interface Cat extends Animal {  
    run(): void;  
}

所以，以下可以实现，不会报错

interface Animal {  
    name: string;  
}  
interface Cat {  
    name: string;  
    run(): void;  
}  
​  
let tom: Cat = {  
    name: 'Tom',  
    run: () => { console.log('run') }  
};  
let animal: Animal = tom;

就像面向对象编程中,我们可以将子类的实例赋值给类型为父类的变量

但是，不能将父类的实例赋值给类型为子类的变量。

综上所述：

* 联合类型可以被断言为其中一个类型
* 父类可以被断言为子类（子类可以被断言为父类）
* 任何类型都可以被断言为 any
* any 可以被断言为任何类型
* 要使得 A 能够被断言为 B，只需要 A 兼容 B 或 B 兼容 A 即可

#### **（7）双重断言**

根据

* 任何类型都可以被断言为 any
* any 可以被断言为任何类型
* 可以使用 as any as xxx来断言

interface Cat {  
    run(): void;  
}  
interface Fish {  
    swim(): void;  
}  
​  
function testCat(cat: Cat) {  
    return (cat as any as Fish);  
}

如果使用了这种双重断言，它很可能会导致运行时错误。

除非迫不得已，千万别用双重断言。

#### **（8）类型断言和类型转换**

类型断言只会影响 TypeScript 编译时的类型，类型断言语句在编译结果中会被删除

function toBoolean(something: any): boolean {  
    return something as boolean;  
}  
toBoolean(1);  
// 返回值为 1

相当于

function toBoolean(something) {  
    return something;  
}  
toBoolean(1);  
// 返回值为 1

若要进行类型转换，需要直接调用类型转换的方法

#### **（9）类型断言和类型声明**

function getCacheData(key: string): any {  
    return (window as any).cache[key];  
}  
​  
interface Cat {  
    name: string;  
    run(): void;  
}  
​  
const tom: Cat = getCacheData('tom');  
tom.run();

其中的

const tom = getCacheData('tom') as Cat;

等价于

const tom: Cat = getCacheData('tom');

深入的讲，它们的核心区别就在于：

* animal 断言为 Cat，只需要满足 Animal 兼容 Cat 或 Cat 兼容 Animal 即可
* animal 赋值给 tom，需要满足 Cat 兼容 Animal 才行

优先使用类型声明，这也比类型断言的 as 语法更加优雅

#### **（10）类型断言 vs 泛型**

function getCacheData<T>(key: string): T {  
    return (window as any).cache[key];  
}  
​  
interface Cat {  
    name: string;  
    run(): void;  
}  
​  
const tom = getCacheData<Cat>('tom');  
tom.run();

使用泛型来实现代替类型断言，可以更加规范的实现对 getCacheData 返回值的约束，这也同时去除掉了代码中的 any，是最优的一个解决方案

## **五、泛型**

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。

### **1.简单例子**

function createArray<T>(length: number, value: T): Array<T> {  
    let result: T[] = [];  
    for (let i = 0; i < length; i++) {  
        result[i] = value;  
   }  
    return result;  
}  
​  
createArray<string>(3, 'x'); // ['x', 'x', 'x']

通过泛型准确的定义返回值的类型

Array<any> 允许数组的每一项都为任意类型。但是我们预期的是，数组中每一项都应该是输入的 value 的类型。

在函数名后添加了 <T>，其中 T 用来指代任意输入的类型，在后面的输入 value: T 和输出 Array<T> 中即可使用了。

接着在调用的时候，可以指定它具体的类型为 string。当然，也可以不手动指定，而让类型推论自动推算出来：

createArray(3, 'x'); // ['x', 'x', 'x']

### **2.多个类型参数**

定义泛型的时候，可以一次定义多个类型参数：

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

### **3.泛型约束**

在函数内部使用泛型变量的时候，由于事先不知道它是哪种类型，所以不能随意的操作它的属性或方法：

这时，我们可以对泛型进行约束，只允许这个函数传入那些包含 length 属性的变量。这就是泛型约束

interface Lengthwise {  
    length: number;  
}  
​  
function loggingIdentity<T extends Lengthwise>(arg: T): T {  
    console.log(arg.length);  
    return arg;  
}

多个类型参数之间也可以互相约束

function copyFields<T extends U, U>(target: T, source: U): T {  
    for (let id in source) {  
        target[id] = (<T>source)[id];  
   }  
    return target;  
}  
​  
let x = { a: 1, b: 2, c: 3, d: 4 };  
​  
copyFields(x, { b: 10, d: 20 });

使用了两个类型参数，其中要求 T 继承 U，这样就保证了 U 上不会出现 T 中不存在的字段。

### **4.泛型接口**

以把泛型参数提前到接口名上

interface CreateArrayFunc<T> {  
 (length: number, value: T): Array<T>  
}  
​  
let createArray: CreateArrayFunc<any>  
createArray = function <T>(length: number, value: T): Array<T> {  
  let result: T[] = []  
  for (let i = 0; i < length; i++) {  
    result[i] = value  
 }  
  return result  
}  
​  
console.log(createArray(3, 'x'))// ['x', 'x', 'x']

使用泛型接口的时候，需要定义泛型的类型

### **5.泛型类**

与泛型接口类似，泛型也可以用于类的类型定义中

class GenericNumber<T> {  
    zeroValue: T;  
    add: (x: T, y: T) => T;  
}  
​  
let myGenericNumber = new GenericNumber<number>();  
myGenericNumber.zeroValue = 0;  
myGenericNumber.add = function(x, y) { return x + y; };

### **6.泛型参数的默认类型**

在 TypeScript 2.3 以后，我们可以为泛型中的类型参数指定默认类型。当使用泛型时没有在代码中直接指定类型参数，从实际值参数中也无法推测出时，这个默认类型就会起作用。

function createArray<T = string>(length: number, value: T): Array<T> {  
    let result: T[] = [];  
    for (let i = 0; i < length; i++) {  
        result[i] = value;  
   }  
    return result;  
}

## **六、声明文件**

### **1.声明语句**

假如我们想使用第三方库 jQuery，一种常见的方式是在 html 中通过 <script> 标签引入 jQuery，然后就可以使用全局变量 $ 或 jQuery 了。

但是在 ts 中，编译器无法解析 $ 或 jQuery

这时，我们可以使用 declare var 来定义它的类型

declare var jQuery: (selector: string) => any;  
​  
jQuery('#foo');

### **2.声明文件**

#### **（1）概念**

通常我们会把声明语句放到一个单独的文件（jQuery.d.ts）中，这就是声明文件

声明文件必需以 .d.ts 为后缀

假如无法解析，那么可以检查下 tsconfig.json 中的 files、include 和 exclude 配置，确保其包含了 jQuery.d.ts 文件。

#### **（2）第三方声明文件**

大部分的js库的声明文件不需要我们来定义，可以直接下载下来使用，但是更推荐的是使用 @types 统一管理第三方库的声明文件。

@types 的使用方式很简单，直接用 npm 安装对应的声明模块即可，以 jQuery 举例：

npm install @types/jquery --save-dev

## **七、类型别名**

类型别名用来给一个类型起个新名字。

type Name = string;  
type NameResolver = () => string;  
type NameOrResolver = Name | NameResolver;  
function getName(n: NameOrResolver): Name {  
    if (typeof n === 'string') {  
        return n;  
   } else {  
        return n();  
   }  
}

我们使用 type 创建类型别名。

类型别名常用于联合类型。

## **八、字符串字面量类型**

字符串字面量类型用来约束取值只能是某几个字符串中的一个。

type EventNames = 'click' | 'scroll' | 'mousemove';  
function handleEvent(ele: Element, event: EventNames) {  
    // do something  
}  
​  
handleEvent(document.getElementById('hello'), 'scroll');  // 没问题  
handleEvent(document.getElementById('world'), 'dblclick'); // 报错，event 不能为 'dblclick'  
​  
// index.ts(7,47): error TS2345: Argument of type '"dblclick"' is not assignable to parameter of type 'EventNames'.

使用 type 定了一个字符串字面量类型 EventNames，它只能取三种字符串中的一种。

注意，类型别名与字符串字面量类型都是使用 type 进行定义。

## **九、元组**

数组合并了相同类型的对象，而元组（Tuple）合并了不同类型的对象。

### **1.简单使用**

定义一对值分别为 string 和 number 的元组：

let tom: [string, number] = ['Tom', 25];

当赋值或访问一个已知索引的元素时，会得到正确的类型：

let tom: [string, number];  
tom=['',0]//初始化  
tom[0] = 'Tom';  
tom[1] = 25;  
​  
tom[0].slice(1);  
tom[1].toFixed(2);

也可以只赋值其中一项：

let tom: [string, number];  
tom[0] = 'Tom';

但是当直接对元组类型的变量进行初始化或者赋值的时候，需要提供所有元组类型中指定的项。

let tom: [string, number];  
tom = ['Tom', 25];  
​  
let tom: [string, number];  
tom = ['Tom'];  
​  
// Property '1' is missing in type '[string]' but required in type '[string, number]'.

### **2.越界的元素**

当添加越界的元素时，它的类型会被限制为元组中每个类型的联合类型：

let tom: [string, number];  
tom = ['Tom', 25];  
tom.push('male');  
tom.push(true);  
​  
// Argument of type 'true' is not assignable to parameter of type 'string | number'.

此处tom元组内的类型为string和number，则越界的元素类型必须是其中之一

## **十、枚举**

枚举（Enum）类型用于取值被限定在一定范围内的场景，比如一周只能有七天，颜色限定为红绿蓝等。

### **1.简单使用**

枚举使用 enum 关键字来定义：

enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

枚举成员会被赋值为从 0 开始递增的数字，同时也会对枚举值到枚举名进行反向映射：

console.log(Days["Sun"] === 0); // true  
console.log(Days["Mon"] === 1); // true  
console.log(Days["Tue"] === 2); // true  
console.log(Days["Sat"] === 6); // true  
​  
console.log(Days[0] === "Sun"); // true  
console.log(Days[1] === "Mon"); // true  
console.log(Days[2] === "Tue"); // true  
console.log(Days[6] === "Sat"); // true

相当于

var Days;  
(function (Days) {  
    Days[Days["Sun"] = 3] = "Sun";  
    Days[Days["Mon"] = 1] = "Mon";  
    Days[Days["Tue"] = 2] = "Tue";  
    Days[Days["Wed"] = 3] = "Wed";  
    Days[Days["Thu"] = 4] = "Thu";  
    Days[Days["Fri"] = 5] = "Fri";  
    Days[Days["Sat"] = 6] = "Sat";  
})(Days || (Days = {}));

### **2.手动赋值**

我们也可以给枚举项手动赋值：

enum Days {Sun = 7, Mon = 1, Tue, Wed, Thu, Fri, Sat};  
​  
console.log(Days["Sun"] === 7); // true  
console.log(Days["Mon"] === 1); // true  
console.log(Days["Tue"] === 2); // true  
console.log(Days["Sat"] === 6); // true

未手动赋值的枚举项会接着上一个枚举项递增

如果未手动赋值的枚举项与手动赋值的重复了，TypeScript 是不会察觉到这一点的：

enum Days {Sun = 3, Mon = 1, Tue, Wed, Thu, Fri, Sat};  
​  
console.log(Days["Sun"] === 3); // true  
console.log(Days["Wed"] === 3); // true  
console.log(Days[3] === "Sun"); // false  
console.log(Days[3] === "Wed"); // true

上面的例子中，递增到 3 的时候与前面的 Sun 的取值重复了，但是 TypeScript 并没有报错，导致 Days[3] 的值先是 "Sun"，而后又被 "Wed" 覆盖了。

手动赋值的枚举项可以不是数字，此时需要使用类型断言来让 tsc 无视类型检查 (编译出的 js 仍然是可用的)：

enum Days {Sun = 7, Mon, Tue, Wed, Thu, Fri, Sat = <any>"S"};

当然，手动赋值的枚举项也可以为小数或负数，此时后续未手动赋值的项的递增步长仍为 1：

enum Days {Sun = 7, Mon = 1.5, Tue, Wed, Thu, Fri, Sat};  
​  
console.log(Days["Sun"] === 7); // true  
console.log(Days["Mon"] === 1.5); // true  
console.log(Days["Tue"] === 2.5); // true  
console.log(Days["Sat"] === 6.5); // true

### **3.常数项和计算所得项**

枚举项有两种类型：常数项（constant member）和计算所得项（computed member）。

前面我们所举的例子都是常数项，一个典型的计算所得项的例子：

enum Color {Red, Green, Blue = "blue".length};

上面的例子中，"blue".length 就是一个计算所得项。

如果紧接在计算所得项后面的是未手动赋值的项，那么它就会因为无法获得初始值而报错

enum Color {Red = "red".length, Green, Blue};  
​  
// index.ts(1,33): error TS1061: Enum member must have initializer.  
// index.ts(1,40): error TS1061: Enum member must have initializer.

下面是常数项和计算所得项的完整定义:

当满足以下条件时，枚举成员被当作是常数：

* 不具有初始化函数并且之前的枚举成员是常数。在这种情况下，当前枚举成员的值为上一个枚举成员的值加 1。但第一个枚举元素是个例外。如果它没有初始化方法，那么它的初始值为 0。
* 枚举成员使用常数枚举表达式初始化。常数枚举表达式是 TypeScript 表达式的子集，它可以在编译阶段求值。当一个表达式满足下面条件之一时，它就是一个常数枚举表达式：
  + - 数字字面量
    - 引用之前定义的常数枚举成员（可以是在不同的枚举类型中定义的）如果这个成员是在同一个枚举类型中定义的，可以使用非限定名来引用
    - 带括号的常数枚举表达式
    - +, -, ~ 一元运算符应用于常数枚举表达式
    - +, -, \*, /, %, <<, >>, >>>, &, |, ^ 二元运算符，常数枚举表达式做为其一个操作对象。若常数枚举表达式求值后为 NaN 或 Infinity，则会在编译阶段报错
    - 所有其它情况的枚举成员被当作是需要计算得出的值。

### **4.常数枚举**

常数枚举是使用 const enum 定义的枚举类型：

const enum Directions {  
    Up,  
    Down,  
    Left,  
    Right  
}  
​  
let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

常数枚举与普通枚举的区别是，它会在编译阶段被删除，并且不能包含计算成员。

上例的编译结果是：

var directions = [0 /\* Up \*/, 1 /\* Down \*/, 2 /\* Left \*/, 3 /\* Right \*/];

假如包含了计算成员，则会在编译阶段报错：

const enum Color {Red, Green, Blue = "blue".length};  
​  
// index.ts(1,38): error TS2474: In 'const' enum declarations member initializer must be constant expression.

### **5.外部枚举**

外部枚举（Ambient Enums）是使用 declare enum 定义的枚举类型：

declare enum Directions {  
    Up,  
    Down,  
    Left,  
    Right  
}  
​  
let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

之前提到过，declare 定义的类型只会用于编译时的检查，编译结果中会被删除。所以此处无法打印，否则报错

上例的编译结果是：

var directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

外部枚举与声明语句一样，常出现在声明文件中。

同时使用 declare 和 const 也是可以的：

declare const enum Directions {  
    Up,  
    Down,  
    Left,  
    Right  
}  
​  
let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

编译结果：

var directions = [0 /\* Up \*/, 1 /\* Down \*/, 2 /\* Left \*/, 3 /\* Right \*/];

## **十一、类**

TypeScript 可以使用三种访问修饰符（Access Modifiers），分别是 public、private 和 protected。

* public 修饰的属性或方法是公有的，可以在任何地方被访问到，默认所有的属性和方法都是 public 的
* private 修饰的属性或方法是私有的，不能在声明它的类的外部访问
* protected 修饰的属性或方法是受保护的，它和 private 类似，区别是它在子类中也是允许被访问的

### **1.参数属性**

修饰符和readonly还可以使用在构造函数参数中，等同于类中定义该属性同时给该属性赋值，使代码更简洁。

class Animal {  
  public constructor(public name) {  
    // this.name = name;  
 }  
}

相当于

class Animal {  
  public name;  
  public constructor(name) {  
    // this.name = name;  
 }  
}

### **2.readonly**

只读属性关键字，只允许出现在属性声明或索引签名或构造函数中。

在类的外部，只能读取，无法赋值

class Animal {  
  readonly name;  
  public constructor(name) {  
    this.name = name;  
 }  
}  
​  
let a = new Animal('Jack');  
console.log(a.name); // Jack  
a.name = 'Tom';  
// index.ts(10,3): TS2540: Cannot assign to 'name' because it is a read-only property.

如果 readonly 和其他访问修饰符同时存在的话，需要写在其后面

class Animal {  
  // public readonly name;  
  public constructor(public readonly name) {  
    // this.name = name;  
 }  
}

### **3.抽象类**

abstract 用于定义抽象类和其中的抽象方法。

#### **（1）不允许被实例化**

abstract class Animal {  
  public name;  
  public constructor(name) {  
    this.name = name;  
 }  
  public abstract sayHi();  
}  
​  
let a = new Animal('Jack');  
​  
// index.ts(9,11): error TS2511: Cannot create an instance of the abstract class 'Animal'.

#### **（2）抽象方法必须被子类实现**

如果定义了一个类 Cat 继承了抽象类 Animal，但是没有实现抽象方法 sayHi，那么编译时会报错。

注意：抽象方法不能具体实现

abstract class Animal {  
  public name;  
  public constructor(name) {  
    this.name = name;  
 }  
  public abstract sayHi();  
}  
​  
class Cat extends Animal {  
  public sayHi() {  
    console.log(`Meow, My name is ${this.name}`);  
 }  
}  
​  
let cat = new Cat('Tom');

#### **（3）类的类型**

给类加上 TypeScript 的类型，与接口类似：

class Animal {  
  name: string;  
  constructor(name: string) {  
    this.name = name;  
 }  
  sayHi(): string {  
    return `My name is ${this.name}`;  
 }  
}  
​  
let a: Animal = new Animal('Jack');  
console.log(a.sayHi()); // My name is Jack

## **十二、类与接口**

实现（implements）是面向对象中的一个重要概念。一般来讲，一个类只能继承自另一个类，有时候不同类之间可以有一些共有的特性，这时候就可以把特性提取成接口（interfaces），用 implements 关键字来实现。这个特性大大提高了面向对象的灵活性。

### **1.类实现多个接口**

interface Alarm {  
    alert(): void;  
}  
​  
interface Light {  
    lightOn(): void;  
    lightOff(): void;  
}  
​  
class Car implements Alarm, Light {  
    alert() {  
        console.log('Car alert');  
   }  
    lightOn() {  
        console.log('Car light on');  
   }  
    lightOff() {  
        console.log('Car light off');  
   }  
}

### **2.接口继承接口**

interface Alarm {  
    alert(): void;  
}  
​  
interface LightableAlarm extends Alarm {  
    lightOn(): void;  
    lightOff(): void;  
}

这很好理解，LightableAlarm 继承了 Alarm，除了拥有 alert 方法之外，还拥有两个新方法 lightOn 和 lightOff。

### **3.接口继承类**

常见的面向对象语言中，接口是不能继承类的，但是在 TypeScript 中却是可以的：

class Point {  
    x: number;  
    y: number;  
    constructor(x: number, y: number) {  
        this.x = x;  
        this.y = y;  
   }  
}  
​  
interface Point3d extends Point {  
    z: number;  
}  
​  
let point3d: Point3d = {x: 1, y: 2, z: 3};

当我们在声明 class Point 时，除了会创建一个名为 Point 的类之外，同时也创建了一个名为 Point 的类型（实例的类型）。

所以我们既可以将 Point 当做一个类来用（使用 new Point 创建它的实例），也可以将 Point 当做一个类型来用（使用 : Point 表示参数的类型）。

class Point {  
    x: number;  
    y: number;  
    constructor(x: number, y: number) {  
        this.x = x;  
        this.y = y;  
   }  
}  
​  
interface PointInstanceType {  
    x: number;  
    y: number;  
}

function printPoint(p: Point) {  
    console.log(p.x, p.y);  
}

可以等价于

function printPoint(p: PointInstanceType) {  
    console.log(p.x, p.y);  
}

所以，可以说TypeScript 支持接口继承类

"接口继承类"和"接口继承接口"没有什么本质的区别。