**Type Script**

**TypeScript 是什么?**

[TypeScript](http://www.typescriptlang.org/)  是 JavaScript 的一个超集，主要提供了 **类型系统** 和 **对 ES6 的支持** ，它由 Microsoft 开发，代码 [开源于 GitHub](https://github.com/Microsoft/TypeScript)  上。

**为什么选择 TypeScript?**

**TypeScript 增加了代码的可读性和可维护性.**

* 类型系统实际上是最好的文档，大部分的函数看类型的定义就可以知道如何使用
* 可以在编译阶段就发现大部分错误
* 增强了编辑器和 IDE 的功能，包括代码补全、接口提示、跳转到定义、代码重构等

**TypeScript 非常包容**

* TypeScript 是 JavaScript 的超集， .js 文件可以直接重命名为 .ts 即可
* 即使不显式的定义类型，也能够自动做出类型推论
* TypeScript 的类型系统是图灵完备的，可以定义从简单到复杂的几乎一切类型
* 即使 TypeScript 编译报错，也可以生成 JavaScript 文件
* 兼容第三方库，即使第三方库不是用 TypeScript 写的，也可以编写单独的类型文件供 TypeScript 读取

**TypeScript 拥有活跃的社区**

* 大部分第三方库都有提供给 TypeScript 的类型定义文件
* Angular、Vue、VS Code、Ant Design 等等耳熟能详的项目都是使用 TypeScript 编写的
* TypeScript 拥抱了 ES6 规范，支持 ESNext 草案中处于第三阶状态（Stage 3）的特性

**TypeScript 的缺点**

* 有一定的学习成本，需要理解接口（Interfaces）、泛型（Generics）、类（Classes）、枚举类型（Enums）等前端工程师可能不是很熟悉的概念
* 短期会增加一些开发成本，不过对于一个需要长期维护的项目，TypeScript 能够减少其维护成本

**原始数据类型**

为了让程序有价值，我们需要能够处理最简单的数据单元：数字，字符串，结构体，布尔值等。 TypeScript支持与JavaScript几乎相同的数据类型，此外还提供了实用的枚举类型方便我们使用

**Boolean 类型**

最基本的数据类型就是简单的true/false值，在JavaScript和TypeScript里叫做 boolean （其它语言中也一样）。

let isDone: boolean = false;

注意，使用构造函数 Boolean  创造的对象 **不是** 布尔值：

let createdByNewBoolean: boolean = new Boolean(1);// Type 'Boolean' is not assignable to type 'boolean'.// 'boolean' is a primitive, but 'Boolean' is a wrapper object. Prefer using 'boolean' when possible.//事实上 new Boolean() 返回的是一个 Boolean 对象：let createdByNewBoolean: Boolean = new Boolean(1);//直接调用 Boolean 也可以返回一个 boolean 类型：let createdByBoolean: boolean = Boolean(1);

**Number 类型**

使用 number  定义数值类型：

let decLiteral: number = 6;let hexLiteral: number = 0xf00d;// ES6 中的二进制表示法let binaryLiteral: number = 0b1010;// ES6 中的八进制表示法let octalLiteral: number = 0o744;let notANumber: number = NaN;let infinityNumber: number = Infinity;

**String 类型**

用 string  定义字符串类型：

let myName: string = 'Tom';let myAge: number = 25;// 模板字符串let sentence: string = `Hello, my name is ${myName}.I'll be ${myAge + 1} years old next month.`;

**Array类型**

TypeScript像JavaScript一样可以操作数组元素。 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];//数组的项中不允许出现其他的类型：//第二种方式是使用数组泛型，Array<元素类型>：let list: Array<number> = [1, 2, 3];//用接口表示数组interface NumberArray { [index: number]: number;}let fibonacci: NumberArray = [1, 1, 2, 3, 5];//类数组 （Array-like Object）不是数组类型，比如 arguments：function sum() { let args: number[] = arguments;}// Type 'IArguments' is missing the following properties from type 'number[]': pop, push, concat, join, and 24 more.//上例中，arguments 实际上是一个类数组，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;}//any 在数组中的应用 一个比较常见的做法是，用 any 表示数组中允许出现任意类型:let list: any[] = ['xcatliu', 25, { website: 'http://xcatliu.com' }];

**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。

// Declare a tuple typelet x: [string, number];// Initialize itx = ['hello', 10]; // OK// Initialize it incorrectlyx = [10, 'hello']; // Error//当访问一个已知索引的元素，会得到正确的类型：console.log(x[0].substr(1)); // OKconsole.log(x[1].substr(1)); // Error, 'number' does not have 'substr'//当访问一个越界的元素，会使用联合类型替代：x[3] = 'world'; // OK, 字符串可以赋值给(string | number)类型console.log(x[5].toString()); // OK, 'string' 和 'number' 都有 toStringx[6] = true; // Error, 布尔不是(string | number)类型

**枚举**

enum 类型是对JavaScript标准数据类型的一个补充。 像C#等其它语言一样，使用枚举类型可以为一组数值赋予友好的名字。

enum Color {Red, Green, Blue}let c: Color = Color.Green;//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，我们将上面的例子改成从 1开始编号：enum Color {Red = 1, Green, Blue}let c: Color = Color.Green//或者，全部都采用手动赋值：enum Color {Red = 1, Green = 2, Blue = 4}let c: Color = Color.Green;//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

**Void类型**

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');}声明一个 void 类型的变量没有什么用，因为你只能将它赋值为 undefined 和 null：let unusable: void = undefined;

**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 number 类型的变量：// 这样不会报错let num: number = undefined;// 这样也不会报错let u: undefined;let num: number = u;而 void 类型的变量不能赋值给 number 类型的变量：let u: void;let num: number = u;// Type 'void' is not assignable to type 'number'.

**任意值**

在 TypeScript 中，任何类型都可以被归为 any 类型。这让 any 类型成为了类型系统的顶级类型（也被称作全局超级类型）。

let notSure: any = 666;notSure = "Semlinker";notSure = false;any 类型本质上是类型系统的一个逃逸舱。作为开发者，这给了我们很大的自由：TypeScript 允许我们对 any 类型的值执行任何操作，而无需事先执行任何形式的检查。比如：let anyThing: any = 'hello';console.log(anyThing.myName);console.log(anyThing.myName.firstName);可以认为，声明一个变量为任意值之后，对它的任何操作，返回的内容的类型都是任意值。

**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** ：

let something;something = 'seven';something = 7;something.setName('Tom');//等价于let something: any;something = 'seven';something = 7;

something.setName('Tom');

**类型推论**

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。

let myFavoriteNumber = 'seven';myFavoriteNumber = 7;// index.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.//事实上，它等价于：let myFavoriteNumber: string = 'seven';myFavoriteNumber = 7;

\* **注意与上一个例子的区别在于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。**

**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。

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

// index.ts(4,1): error TS2322: Type 'boolean' is not assignable to type 'string | number'.// Type 'boolean' is not assignable to type 'number'./\*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(9,22): error TS2339: Property 'length' does not exist on type 'string | number'.// Property 'length' does not exist on type 'number'.

**对象的类型——接口**

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以外，也常用于对「对象的形状（Shape）」进行描述。

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](https://msdn.microsoft.com/en-us/library/8bc1fexb%28v=vs.71%29.aspx) 。

interface Person { name: string; age: number;}let tom: Person = { name: 'Tom', age: 25};//定义的变量比接口少了一些属性是不允许的//有时我们希望不要完全匹配，那么可以用可选属性：interface Person { name: string; age?: number;}

let tom: Person = { name: 'Tom'};//带有可选属性的接口与普通的接口定义差不多，只是在可选属性名字定义的后面加一个?符号。//这时仍然不允许添加未定义的属性：//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：interface Person { name: string; age?: number; [propName: string]: any;}

let tom: Person = { name: 'Tom', gender: 'male'};

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

**\*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]: string;}let tom: Person = { name: 'Tom', age: 25, gender: 'male'};// index.ts(3,5): error TS2411: Property 'age' of type 'number' is not assignable to string index type 'string'.// index.ts(7,5): error TS2322: Type '{ [x: string]: string | number; name: string; age: number; gender: string; }' is not assignable to type 'Person'.// Index signatures are incompatible.// Type 'string | number' is not assignable to type 'string'.// Type 'number' is not assignable to type 'string'.

上例中，任意属性的值允许是 string ，但是可选属性 age 的值却是 number ， number 不是 string 的子属性，所以报错了。

另外，在报错信息中可以看出，此时 { name: 'Tom', age: 25, gender: 'male' } 的类型被推断成了 { [x: string]: string | number; name: string; age: number; gender: string; } ，这是联合类型和接口的结合。

一个接口中只能定义一个任意属性。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'};

**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：interface Person { readonly id: number; name: string; age?: number; [propName: string]: any;}

let tom: Person = { name: 'Tom', gender: 'male'};

tom.id = 89757;

// index.ts(22,5): 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(27,5): error TS2540: Cannot assign to 'id' because it is a constant or a read-only property.

**类**

传统的JavaScript程序使用函数和基于原型的继承来创建可重用的组件，但对于熟悉使用面向对象方式的程序员来讲就有些棘手，因为他们用的是基于类的继承并且对象是由类构建出来的。 从ECMAScript 2015，也就是ECMAScript 6开始，JavaScript程序员将能够使用基于类的面向对象的方式。 使用TypeScript，允许开发者现在就使用这些特性，并且编译后的JavaScript可以在所有主流浏览器和平台上运行，而不需要等到下个JavaScript版本。

**TypeScript 中类的用法**

**public private 和 protected**

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

class Animal { public name; public constructor(name) { this.name = name; }}let a = new Animal('Jack');console.log(a.name); // Jacka.name = 'Tom';console.log(a.name); // Tom//上面的例子中，name 被设置为了 public，所以直接访问实例的 name 属性是允许的。//很多时候，我们希望有的属性是无法直接存取的，这时候就可以用 private 了：class Animal { private name; public constructor(name) { this.name = name; }}

let a = new Animal('Jack');console.log(a.name); // Jacka.name = 'Tom';

// index.ts(21,13): error TS2341: Property 'name' is private and only accessible within class 'Animal'.// index.ts(22,1): error TS2341: Property 'name' is private and only accessible within class 'Animal'.

需要注意的是，TypeScript 编译之后的代码中，并没有限制 private  属性在外部的可访问性。

使用 private  修饰的属性或方法，在子类中也是不允许访问的.

而如果是用 protected  修饰，则允许在子类中访问：

class Animal { protected name; public constructor(name) { this.name = name; }}class Cat extends Animal { constructor(name) { super(name); console.log(this.name); }}

当构造函数修饰为 private  时，该类不允许被继承或者实例化

当构造函数修饰为 protected  时，该类只允许被继承

**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 name: string; public constructor(public name) { // this.name = name; }}//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。class Animal { readonly name; public constructor(name) { this.name = name; }}

let a = new Animal('Jack');console.log(a.name); // Jacka.name = 'Tom';

// index.ts(17,3): TS2540: Cannot assign to 'name' because it is a read-only property.//注意如果 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; }}

**抽象类**

abstract  用于定义抽象类和其中的抽象方法。 抽象类是不允许被实例化, 抽象类中的抽象方法必须被子类实现

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');

**类的类型**

给类加上 TypeScript 的类型很简单，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

**函数的类型**

在 JavaScript 中，有两种常见的定义函数的方式——函数声明（Function Declaration）和函数表达式（Function Expression）：

// 函数声明（Function Declaration）function sum(x, y) { return x + y;}// 函数表达式（Function Expression）let mySum = function (x, y) { return x + y;};//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：//如果要我们现在写一个对函数表达式（Function Expression）的定义，可能会写成这样：let mySum = function (x: number, y: number): number { return x + y;};//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。

**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;}

//采用函数表达式|接口定义函数的方式时，对等号左侧进行类型限制，//可以保证以后对函数名赋值时保证参数个数、参数类型、返回值类型不变。//前面提到，输入多余的（或者少于要求的）参数，是不允许的。那么如何定义可选的参数呢？//与接口中的可选属性类似，我们用 ? 表示可选的参数：function buildName(firstName: string, lastName?: string) { if (lastName) { return firstName + ' ' + lastName; } else { return firstName; }}let tomcat = buildName('Tom', 'Cat');let tom = buildName('Tom');

//\*需要注意的是，可选参数必须接在必需参数后面。

//在 ES6 中，我们允许给函数的参数添加默认值，TypeScript 会将添加了默认值的参数识别为可选参数：

function buildName(firstName: string, lastName: string = 'Cat') { return firstName + ' ' + lastName;}let tomcat = buildName('Tom', 'Cat');let tom = buildName('Tom');//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');ES6 中，可以使用 ...rest 的方式获取函数中的剩余参数（rest 参数）：function push(array: any[], ...items: any[]) { items.forEach(function(item) { array.push(item); });}

let a = [];push(a, 1, 2, 3);

**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。

比如，我们需要实现一个函数 reverse ，输入数字 123 的时候，输出反转的数字 321 ，输入字符串 'hello' 的时候，输出反转的字符串 'olleh' 。

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

然而这样有一个缺点，就是不能够精确的表达，输入为数字的时候，输出也应该为数字，输入为字符串的时候，输出也应该为字符串。

这时，我们可以使用重载定义多个 reverse 的函数类型：

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

**声明合并**

如果定义了两个相同名字的函数、接口或类，那么它们会合并成一个类型：

**函数的合并**

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

**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;}//------------------------------------------------------------------------------//注意，合并的属性的类型必须是唯一的：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;}//------------------------------------------------------------------------------//接口中方法的合并，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;}

**类的合并**

类的合并与接口的合并规则一致。

**类型断言**

类型断言（Type Assertion）可以用来手动指定一个值的类型。

//值 as 类型 或 <类型>值let someValue: any = "this is a string";let strLength: number = (someValue as string).length;

let someValue: any = "this is a string";let strLength: number = (<string>someValue).length;

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

形如 <Foo> 的语法在 tsx 中表示的是一个 ReactNode ，在 ts 中除了表示类型断言之外，也可能是表示一个泛型。

* 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
* 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

**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(); }}

**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。

function createArray(length: number, value: any): Array<any> { let result = []; for (let i = 0; i < length; i++) { result[i] = value; } return result;}createArray(3, 'x'); // ['x', 'x', 'x']

这段代码编译不会报错，但是一个显而易见的缺陷是，它并没有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。

这时候，泛型就派上用场了：

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']

**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]

上例中，我们定义了一个 swap  函数，用来交换输入的元组。

**泛型约束**

function loggingIdentity<T>(arg: T): T { console.log(arg.length); return arg;}// index.ts(2,19): error TS2339: Property 'length' does not exist on type 'T'.

上例中，泛型 T 不一定包含属性 length ，所以编译的时候报错了。

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;}

**泛型接口**

可以使用接口的方式来定义一个函数需要符合的形状, 当然也可以使用含有泛型的接口来定义函数的形状：

interface CreateArrayFunc { <T>(length: number, value: T): Array<T>;}

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

createArray(3, 'x'); // ['x', 'x', 'x']//进一步，我们可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;}

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

**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; }

**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;}

**声明文件**

当使用第三方库时，我们需要引用它的声明文件，才能获得对应的代码补全、接口提示等功能。

**新语法索引**

* declare var 声明全局变量
* declare function 声明全局方法
* declare class 声明全局类
* declare enum 声明全局枚举类型
* declare namespace 声明（含有子属性的）全局对象
* interface 和 type 声明全局类型
* export 导出变量
* export namespace 导出（含有子属性的）对象
* export default ES6 默认导出
* export = commonjs 导出模块
* export as namespace UMD 库声明全局变量
* declare global 扩展全局变量
* declare module 扩展模块
* /// <reference /> 三斜线指令

此部分内容比较多且繁杂，如有兴趣可自行去官网查询了解此处不再一一介绍。