# TypeScript入门

--Rise老师

# 一、什么是TypeScript

> JavaScript的超集，可以编译成JavaScript。添加了类型系统的JavaScript，可以适用于任何规模的项目。

## TypeScript特性

### 类型系统

从 TypeScript 的名字就可以看出来，「类型」是其最核心的特性。

我们知道，JavaScript 是一门非常灵活的编程语言：

- 它没有类型约束，一个变量可能初始化时是字符串，过一会儿又被赋值为数字。
- 由于隐式类型转换的存在，有的变量的类型很难在运行前就确定。
- 基于原型的面向对象编程，使得原型上的属性或方法可以在运行时被修改。
- 函数是 JavaScript 中的一等公民，可以赋值给变量，也可以当作参数或返回值。

这种灵活性就像一把双刃剑，一方面使得 JavaScript 蓬勃发展，无所不能，从 2013 年开始就一直蝉联最普遍使用的编程语言排行榜冠军[3]；另一方面也使得它的代码质量参差不齐，维护成本高，运行时错误多。

而 TypeScript 的类型系统，在很大程度上弥补了 JavaScript 的缺点。

### TypeScript 是静态类型

类型系统按照「类型检查的时机」来分类，可以分为动态类型和静态类型。

动态类型是指在运行时才会进行类型检查，这种语言的类型错误往往会导致运行时错误。JavaScript 是一门解释型语言，没有编译阶段，所以它是动态类型，以下这段代码在运行时才会报错：

```
let foo = 1;
foo.split(' ');
// Uncaught TypeError: foo.split is not a function
// 运行时会报错（foo.split 不是一个函数），造成线上 bug
```

静态类型是指编译阶段就能确定每个变量的类型，这种语言的类型错误往往会导致语法错误。TypeScript 在运行前需要先编译为 JavaScript，而在编译阶段就会进行类型检查，所以 **TypeScript 是静态类型**，这段 TypeScript 代码在编译阶段就会报错了：

```
let foo = 1;
foo.split(' ');
// Property 'split' does not exist on type 'number'.
// 编译时会报错（数字没有 split 方法），无法通过编译
```

你可能会奇怪，这段 TypeScript 代码看上去和 JavaScript 没有什么区别呀。

没错！大部分 JavaScript 代码都只需要经过少量的修改（或者完全不用修改）就变成 TypeScript 代码，这得益于 TypeScript 强大的[类型推论][]，即使不去手动声明变量 `foo` 的类型，也能在变量初始化时自动推论出它是一个 `number` 类型。

完整的 TypeScript 代码是这样的：

```
let foo: number = 1;
foo.split(' ');
// Property 'split' does not exist on type 'number'.
// 编译时会报错（数字没有 split 方法），无法通过编译
```

## TypeScript 是弱类型

类型系统按照「是否允许隐式类型转换」来分类，可以分为强类型和弱类型。

以下这段代码不管是在 JavaScript 中还是在 TypeScript 中都是可以正常运行的，运行时数字 `1` 会被隐式类型转换为字符串 `'1'`，加号 `+` 被识别为字符串拼接，所以打印出结果是字符串 `'11'`。

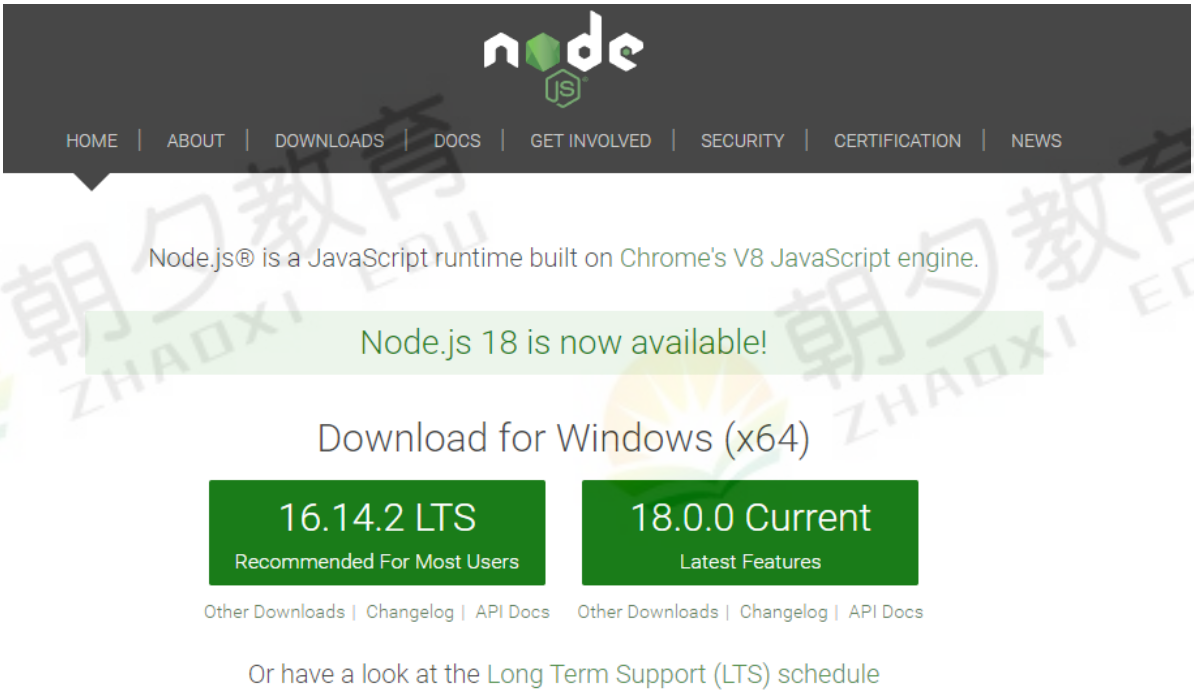
```
console.log(1 + '1');
// 打印出字符串 '11'
```

TypeScript 是完全兼容 JavaScript 的，它不会修改 JavaScript 运行时的特性，所以**它们都是弱类型。**

# 二、安装并编译TypeScript

安装TypeSscript需要nodejs环境，如果电脑没有npm命令的，可以去官网下载并安装nodejs

https://nodejs.org/en/




TypeScript安装命令

```
npm install -g typescript
//通过tsc --version 可以检查版本号确保安装成功
```

安装以后编译ts文件很简单，我们在电脑上新建一个目录，code，新建一个文件index.ts

然后在当前目录下输入命令：

```
tsc index.ts
```

编译完成后会在当面目录下输出一个index.js文件，编译成功。

我们有时候想指定目录进行输出：

```
tsc --outFile ./js/index.js index.ts
```

# 三、基本数据类型

## 布尔值

布尔值是最基础的数据类型，在 TypeScript 中，使用 `boolean` 定义布尔值类型：

```
let isDone: boolean = false;
```

## 数值

使用 `number` 定义数值类型：

```
let num : number = 1;
```

## 字符串

使用 `string` 定义字符串类型：

```
let myName: string = 'Tom';
// 模板字符串
let sentence: string = `Hello, my name is ${myName}.`;
```

## 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');
}
```

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;
```

# 四、任意值（Any）

任意值（Any）用来表示允许赋值为任意类型。

如果是一个普通类型，在赋值过程中改变类型是不被允许的：

```
let myFavoriteNumber: string = 'seven';
myFavoriteNumber = 7;

// index.ts(2,1): error TS2322: Type 'number' is not assignable to type
'string'.
```

但如果是 `any` 类型，则允许被赋值为任意类型。

```
let myFavoriteNumber: any = 'seven';
myFavoriteNumber = 7;
```

在任意值上访问任何属性都是允许的：

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

也允许调用任何方法：

```
let anyThing: any = 'Tom';
anyThing.setName('Jerry');
anyThing.setName('Jerry').sayHello();
anyThing.myName.setFirstName('Cat');
```

所以，**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。**

# 五、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。

以下代码虽然没有指定类型，但是会在编译的时候报错：

```
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;

// index.ts(2,1): error TS2322: Type 'number' is not assignable to type
'string'.
```

TypeScript 会在没有明确的指定类型的时候推测出一个类型，这就是类型推论。

**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;
```

# 六、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。

举个栗子

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

联合类型使用 `|` 分隔每个类型。

这里的 `let myFavoriteNumber: string | number` 的含义是，允许 `myFavoriteNumber` 的类型是 `string` 或者 `number`，但是不能是其他类型。

比如下面这个栗子就会报错：

```
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'.
```

## 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。

访问 `string` 和 `number` 的共有属性是没问题的：

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

联合类型的变量在被赋值的时候，会根据类型推论的规则推断出一个类型：

```
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven';
console.log(myFavoriteNumber.length); // 5
myFavoriteNumber = 7;
console.log(myFavoriteNumber.length); // 编译时报错

// index.ts(5,30): error TS2339: Property 'length' does not exist on type
'number'.
```

上例中，第二行的 `myFavoriteNumber` 被推断成了 `string`，访问它的 `length` 属性不会报错。

而第四行的 `myFavoriteNumber` 被推断成了 `number`，访问它的 `length` 属性时就报错了。

# 七、接口

在 TypeScript 中，我们使用Interfaces来定义一个接口类型的对象。

## 什么是接口

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去实现。

TypeScript的核心原则之一是对值所具有的结构进行类型检查。 它有时被称做“鸭式辨型法”或“结构性子类型化”。 在TypeScript里，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契约。

举个例子

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25
};
```

上面的例子中，我们定义了一个接口 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。

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是不允许的：

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom'
};

// index.ts(6,5): error TS2322: Type '{ name: string; }' is not assignable to
type 'Person'.
//   Property 'age' is missing in type '{ name: string; }'.
```

多一些属性也是不允许的：

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};

// index.ts(9,5): error TS2322: Type '{ name: string; age: number; gender:
string; }' is not assignable to type 'Person'.
//   Object literal may only specify known properties, and 'gender' does not
exist in type 'Person'.
```

可见，**赋值的时候，变量的形状必须和接口的形状保持一致。**

## 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'
};
```

```
interface Person {
    name: string;
    age?: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25
};
```

## 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：

```
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。

## 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.
```

上例中，使用 `readonly` 定义的属性 `id` 初始化后，又被赋值了，所以报错了。

# 八、数组

存放多个元素的集合

最简单的方法是使用「类型 + 方括号」来表示数组：

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

数组的项中**不允许**出现其他的类型：

```
let fibonacci: number[] = [1, '1', 2, 3, 5];

// Type 'string' is not assignable to type 'number'.
```

数组的一些方法的参数也会根据数组在定义时约定的类型进行限制：

```
let fibonacci: number[] = [1, 1, 2, 3, 5];
fibonacci.push('8');

// Argument of type '"8"' is not assignable to parameter of type 'number'.
```

上例中，`push` 方法只允许传入 `number` 类型的参数，但是却传了一个 `"8"` 类型的参数，所以报错了。这里 `"8"` 是一个字符串字面量类型，会在后续章节中详细介绍。

也可以指定一个any类型的数组：

```
let list: any[] = ['xcatliu', 25, { website: 'http://xcatliu.com' }];
```

# 九、函数01，函数声明、函数表达式

## 函数声明

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**：

```
function sum(x: number, y: number): number {
    return x + y;
}
sum(1, 2, 3);

// index.ts(4,1): error TS2346: Supplied parameters do not match any signature of
call target.
```

```
function sum(x: number, y: number): number {
    return x + y;
}
sum(1);

// index.ts(4,1): error TS2346: Supplied parameters do not match any signature of
call target.
```

## 函数表达式

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;
};
```

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，可以参考ES6 中的箭头函数

## 用接口定义函数的形状

我们也可以使用接口的方式来定义一个函数需要符合的形状：

```
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。

# 十、函数02，可选参数，参数默认值

## 可选参数

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。换句话说，**可选参数后面不允许再出现必需参数了**：

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

// index.ts(1,40): error TS1016: A required parameter cannot follow an optional
parameter.
```

## 参数默认值

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');
```

# 十一、函数03，剩余参数、重载

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, ...items) {
    items.forEach(function(item) {
        array.push(item);
    });
}

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

事实上，`items` 是一个数组。所以我们可以用数组的类型来定义它：

```
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，关于 rest 参数，可以参考 ES6 中的 rest 参数。

## 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。方法名相同，参数列表和类型不同。

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'`。

利用联合类型，我们可以这么实现：

```
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('');
    }
}
```

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

上例中，我们重复定义了多次函数 `reverse`，前几次都是函数定义，最后一次是函数实现。在编辑器的代码提示中，可以正确的看到前两个提示。

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。

# 十二、类型断言01，语法，将一个联合类型断言为其中一个类型

类型断言（Type Assertion）可以用来手动指定一个值的类型。

## 语法

```
值 as 类型
```

或者

```
<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`。

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。

类型断言有多重用途

## 将一个联合类型断言为其中一个类型

之前提到过，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中共有的属性或方法**：

```
interface Cat {
    name: string;
    run(): void;
}
interface Fish {
    name: string;
    swim(): void;
}

function getName(animal: Cat | Fish) {
    return animal.name;
}
```

而有时候，我们确实需要在还不确定类型的时候就访问其中一个类型特有的属性或方法，比如：

```
interface Cat {
    name: string;
    run(): void;
}
interface Fish {
    name: string;
    swim(): void;
}
```

```
function isFish(animal: Cat | Fish) {
    if (typeof animal.swim === 'function') {
        return true;
    }
    return false;
}

// index.ts:11:23 - error TS2339: Property 'swim' does not exist on type 'Cat |
Fish'.
//   Property 'swim' does not exist on type 'Cat'.
```

上面的例子中，获取 `animal.swim` 的时候会报错。

此时可以使用类型断言，将 `animal` 断言成 `Fish`：

```
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;
}
```

这样就可以解决访问 `animal.swim` 时报错的问题了。

需要注意的是，类型断言只能够「欺骗」TypeScript 编译器，无法避免运行时的错误，反而滥用类型断言可能会导致运行时错误：

```
interface Cat {
    name: string;
    run(): void;
}
interface Fish {
    name: string;
    swim(): void;
}

function swim(animal: Cat | Fish) {
    (animal as Fish).swim();
}

const tom: Cat = {
    name: 'Tom',
    run() { console.log('run') }
};
swim(tom);
// Uncaught TypeError: animal.swim is not a function`
```

上面的例子编译时不会报错，但在运行时会报错：

```
Uncaught TypeError: animal.swim is not a function`
```

原因是 `(animal as Fish).swim()` 这段代码隐藏了 `animal` 可能为 `Cat` 的情况，将 `animal` 直接断言为 `Fish` 了，而 TypeScript 编译器信任了我们的断言，故在调用 `swim()` 时没有编译错误。

可是 `swim` 函数接受的参数是 `Cat | Fish`，一旦传入的参数是 `Cat` 类型的变量，由于 `Cat` 上没有 `swim` 方法，就会导致运行时错误了。

总之，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。

# 十三、类型断言02，将一个父类断言为更加具体的子类

当类之间有继承关系时，类型断言也是很常见的：

```
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;
}
```

上面的例子中，我们声明了函数 isApiError，它用来判断传入的参数是不是 ApiError 类型，为了实现这样一个函数，它的参数的类型肯定得是比较抽象的父类 Error，这样的话这个函数就能接受 Error 或它的子类作为参数了。

但是由于父类 Error 中没有 code 属性，故直接获取 error.code 会报错，需要使用类型断言获取 (error as ApiError).code。

大家可能会注意到，在这个例子中有一个更合适的方式来判断是不是 ApiError，那就是使用 instanceof：

```
class ApiError extends Error {
    code: number = 0;
}
class HttpError extends Error {
    statusCode: number = 200;
}

function isApiError(error: Error) {
    if (error instanceof ApiError) {
        return true;
    }
    return false;
}
```

上面的例子中，确实使用 instanceof 更加合适，因为 ApiError 是一个 JavaScript 的类，能够通过 instanceof 来判断 error 是否是它的实例。

但是有的情况下 ApiError 和 HttpError 不是一个真正的类，而只是一个 TypeScript 的接口（interface），接口是一个类型，不是一个真正的值，它在编译结果中会被删除，当然就无法使用 instanceof 来做运行时判断了：

```
interface ApiError extends Error {
    code: number;
}
interface HttpError extends Error {
    statusCode: number;
}

function isApiError(error: Error) {
    if (error instanceof ApiError) {
        return true;
    }
    return false;
}

// index.ts:9:26 - error TS2693: 'ApiError' only refers to a type, but is being
used as a value here.
```

此时就只能用类型断言，通过判断是否存在 `code` 属性，来判断传入的参数是不是 `ApiError` 了：

```
interface ApiError extends Error {
    code: number;
}
interface HttpError extends Error {
    statusCode: number;
}

function isApiError(error: Error) {
    if (typeof (error as ApiError).code === 'number') {
        return true;
    }
    return false;
}
```

# 十四、类型断言03，将任何一个类型断言为 any

理想情况下，TypeScript 的类型系统运转良好，每个值的类型都具体而精确。

当我们引用一个在此类型上不存在的属性或方法时，就会报错：

```
const foo: number = 1;
foo.length = 1;
// index.ts:2:5 - error TS2339: Property 'length' does not exist on type
'number'.
```

上面的例子中，数字类型的变量 `foo` 上是没有 `length` 属性的，故 TypeScript 给出了相应的错误提示。

这种错误提示显然是非常有用的。

但有的时候，我们非常确定这段代码不会出错，比如下面这个例子：

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

上面的例子中，我们需要将 window 上添加一个属性 foo，但 TypeScript 编译时会报错，提示我们 window 上不存在 foo 属性。

此时我们可以使用 as any 临时将 window 断言为 any 类型：

```
(window as any).foo = 1;
```

在 any 类型的变量上，访问任何属性都是允许的。

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。

总之，一方面不能滥用 as any，另一方面也不要完全否定它的作用，我们需要在类型的严格性和开发的便利性之间掌握平衡（这也是 TypeScript 的设计理念之一），才能发挥出 TypeScript 最大的价值。

# 十五、类型断言04，将 any 断言为一个具体的类型

在日常的开发中，我们不可避免的需要处理 any 类型的变量，它们可能是由于第三方库未能定义好自己的类型，也有可能是历史遗留的或其他人编写的烂代码，还可能是受到 TypeScript 类型系统的限制而无法精确定义类型的场景。

遇到 any 类型的变量时，我们可以选择无视它，任由它滋生更多的 any。

我们也可以选择改进它，通过类型断言及时的把 any 断言为精确的类型，亡羊补牢，使我们的代码向着高可维护性的目标发展。

举例来说，历史遗留的代码中有个 getCacheData，它的返回值是 any：

```
function getCacheData(key: string): any {
    return (window as any).cache[key];
}
```

那么我们在使用它时，最好能够将调用了它之后的返回值断言成一个精确的类型，这样就方便了后续的操作：

```
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();
```

上面的例子中，我们调用完 getCacheData 之后，立即将它断言为 Cat 类型。这样的话明确了 tom 的类型，后续对 tom 的访问时就有了代码补全，提高了代码的可维护性。

# 十六、类型断言05，类型断言的限制

从上面的例子中，我们可以总结出：

- 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

那么类型断言有没有什么限制呢？是不是任何一个类型都可以被断言为任何另一个类型呢？

答案是否定的——并不是任何一个类型都可以被断言为任何另一个类型。

具体来说，若 `A` 兼容 `B`，那么 `A` 能够被断言为 `B`，`B` 也能被断言为 `A`。

下面我们通过一个简化的例子，来理解类型断言的限制：

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

function testAnimal(animal: Animal) {
    return (animal as Cat);
}
function testCat(cat: Cat) {
    return (cat as Animal);
}
```

上面的栗子中是可以断言的，我们再看看下面的栗子：

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

function testAnimal(animal: Animal) {
  return (animal as Cat);
}
function testCat(cat: Cat) {
  return (cat as Animal);
}
```

这个时候会提示错误，两者不能充分重叠，这意味要想断言成功， 还必须具备一个条件：

- 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

# 十七、类型断言06，双重断言

既然：

- 任何类型都可以被断言为 any
- any 可以被断言为任何类型

那么我们是不是可以使用双重断言 `as any as Foo` 来将任何一个类型断言为任何另一个类型呢？

```
interface Cat {
    run(): void;
}
interface Fish {
    swim(): void;
}

function testCat(cat: Cat) {
    return (cat as any as Fish);
}
```

在上面的例子中，若直接使用 `cat as Fish` 肯定会报错，因为 `Cat` 和 `Fish` 互相都不兼容。

但是若使用双重断言，则可以打破「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」的限制，将任何一个类型断言为任何另一个类型。

若你使用了这种双重断言，那么十有八九是非常错误的，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。**

# 十八、类型断言07，类型断言 vs 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
```

在上面的例子中，将 `something` 断言为 `boolean` 虽然可以通过编译，但是并没有什么用，代码在编译后会变成：

```
function toBoolean(something) {
    return something;
}

toBoolean(1);
// 返回值为 1
```

所以类型断言不是类型转换，它不会真的影响到变量的类型。

若要进行类型转换，需要直接调用类型转换的方法：

```
function toBoolean(something: any): boolean {
    return Boolean(something);
}

toBoolean(1);
// 返回值为 true
```

# 十九、类型断言08，类型断言 vs 类型声明

在这个例子中：

```
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();
```

我们使用 `as Cat` 将 `any` 类型断言为了 `Cat` 类型。

但实际上还有其他方式可以解决这个问题：

```
function getCacheData(key: string): any {
    return (window as any).cache[key];
}

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

const tom: Cat = getCacheData('tom');
tom.run();
```

上面的例子中，我们通过类型声明的方式，将 `tom` 声明为 `Cat`，然后再将 `any` 类型的 `getCacheData('tom')` 赋值给 `Cat` 类型的 `tom`。

这和类型断言是非常相似的，而且产生的结果也几乎是一样的——`tom` 在接下来的代码中都变成了 `Cat` 类型。

它们的区别，可以通过这个例子来理解：

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

const animal: Animal = {
    name: 'tom'
};
let tom = animal as Cat;
```

在上面的例子中，由于 `Animal` 兼容 `Cat`，故可以将 `animal` 断言为 `Cat` 赋值给 `tom`。

但是若直接声明 `tom` 为 `Cat` 类型：

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

const animal: Animal = {
    name: 'tom'
};
let tom: Cat = animal;

// index.ts:12:5 - error TS2741: Property 'run' is missing in type 'Animal' but
required in type 'Cat'.
```

则会报错，不允许将 `animal` 赋值为 `Cat` 类型的 `tom`。

我们可以得出结论：

```
//a断言为b时，a和b有重叠的部分即可
//a声明为b时，a必须具备b的所有属性和方法
```

知道了它们的核心区别，就知道了类型声明是比类型断言更加严格的。

所以为了增加代码的质量，我们最好优先使用类型声明，这也比类型断言的 `as` 语法更加优雅。

# 二十、类型断言09，类型断言 vs 泛型

还是这个例子：

```
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();
```

我们还有第三种方式可以解决这个问题，那就是泛型：

```
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();
```

通过给 `getCacheData` 函数添加了一个泛型 `<T>`，我们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。

# 二一、使用type关键字定义类型别名和字符串字面量类型

我们来看一个方法：

```
function getName(n: string| (() => string)): string {
    if (typeof n === 'string') {
        return n;
    } else {
        return n();
    }
}
```

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();
    }
}
```

上例中，我们使用 `type` 创建类型别名。

类型别名常用于联合类型。

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'
```

上例中，我们使用 `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。**

# 二二、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。

元组起源于函数编程语言（如 F#），这些语言中会频繁使用元组。

## 举个例子

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] = 'Tom';
tom[1] = 25;
```

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];
```

下面这样就不行：

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

// Property '1' is missing in type '[string]' but required in type '[string,
number]'.
```

## 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');//可以添加string
tom.push(true);//但不能添加boolean

// Argument of type 'true' is not assignable to parameter of type 'string |
number'.
```

# 二三、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。

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：

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

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
```

事实上，上面的例子会被编译为：

```
var Days;
(function (Days) {
    Days[Days["Sun"] = 0] = "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 = {}));
```

# 二四、类01，概念、构造函数、属性和方法

## 类的概念

虽然 JavaScript 中有类的概念，但是可能大多数 JavaScript 程序员并不是非常熟悉类，这里对类相关的概念做一个简单的介绍。

- 类（Class）：定义了一件事物的抽象特点，包含它的属性和方法
- 对象（Object）：类的实例，通过 `new` 生成
- 面向对象（OOP）的三大特性：封装、继承、多态
- 封装（Encapsulation）：将对数据的操作细节隐藏起来，只暴露对外的接口。外界调用端不需要（也不可能）知道细节，就能通过对外提供的接口来访问该对象，同时也保证了外界无法任意更改对象内部的数据
- 继承（Inheritance）：子类继承父类，子类除了拥有父类的所有特性外，还有一些更具体的特性
- 多态（Polymorphism）：由继承而产生了相关的不同的类，对同一个方法可以有不同的响应。比如 `Cat` 和 `Dog` 都继承自 `Animal`，但是分别实现了自己的 `eat` 方法。此时针对某一个实例，我们无需了解它是 `Cat` 还是 `Dog`，就可以直接调用 `eat` 方法，程序会自动判断出来应该如何执行 `eat`
- 存取器（getter & setter）：用以改变属性的读取和赋值行为
- 修饰符（Modifiers）：修饰符是一些关键字，用于限定成员或类型的性质。比如 `public` 表示公有属性或方法
- 抽象类（Abstract Class）：抽象类是供其他类继承的基类，抽象类不允许被实例化。抽象类中的抽象方法必须在子类中被实现
- 接口（Interfaces）：不同类之间公有的属性或方法，可以抽象成一个接口。接口可以被类实现（implements）。一个类只能继承自另一个类，但是可以实现多个接口

使用 `class` 定义类，使用 `constructor` 定义构造函数。

通过 `new` 生成新实例的时候，会自动调用构造函数。

```
class Animal {
  public _name;
  constructor(name:string) {
      this._name = name;
  }
  sayHi() {
      return `My name is ${this._name}`;
  }
}

let a = new Animal('Jack');
console.log(a.sayHi()); // My name is Jack
```

# 二五、类02，存取器：get,set

使用 getter 和 setter 可以改变属性的赋值和读取行为：

```
class Animal {
    constructor(name) {
      this.name = name;
    }
    get name() {
      return 'Jack';
    }
    set name(value) {
      console.log('setter: ' + value);
    }
  }

  let a = new Animal('Kitty'); // setter: Kitty
  a.name = 'Tom'; // setter: Tom
  console.log(a.name); // Jack
```

# 二六、类03，静态方法

使用 `static` 修饰符修饰的方法称为静态方法，它们不需要实例化，而是直接通过类来调用：

```
class Animal {
    public _name;
    constructor(name:string) {
        this._name = name;
    }
    sayHi() {
        return `My name is ${this._name}`;
    }
    static sayHello(){
      return "hello,你好呀"
    }
  }

  let a = new Animal('Jack');
```

```
console.log(a.sayHi()); // My name is Jack
console.log(Animal.sayHello()); // hello,你好呀
```

# 二七、类04，三种访问修饰符：public、private 和 protected

- public：全局的，公共的，当前类所涉及到的地方都可以使用

```
class Animal {
  public _name;
  public constructor(name:string) {
    this._name = name;
  }
}

let a = new Animal('Jack');
console.log(a._name); // Jack
a._name = 'Tom';
console.log(a._name); // Tom
```

- private：私有的，只能在类的内部使用，无法在实例化后通过类的实例.属性来访问

```
class Animal {
  private _name;
  public constructor(name:string) {
    this._name = name;
  }
}

let a = new Animal('Jack');
console.log(a._name); // 报错
a._name = 'Tom';      // 报错
console.log(a._name); // 报错
```

- protected：受保护的，private不允许子类访问，使用protected就可以了

```
class Animal {
  protected name;
  public constructor(name:string) {
    this.name = name;
  }
}

class Cat extends Animal {
  constructor(name:string) {
    super(name);
    console.log(this.name);
  }
}
```

# 二八、类05，参数属性和只读属性关键字

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: string) {
    // this.name = name;
  }
}
```

只读属性

```
class Animal {
  readonly name;
  public constructor(name:string) {
    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.
```

# 二九、类06，抽象类

`abstract` 用于定义抽象类和其中的抽象方法。

什么是抽象类?

首先，抽象类是不允许被实例化的：

```
abstract class Animal {
  public name;
  public constructor(name:string) {
    this.name = name;
  }
  public abstract sayHi():void;
}

let a = new Animal('Jack');

// index.ts(9,11): error TS2511: Cannot create an instance of the abstract class
'Animal'.
```

上面的例子中，我们定义了一个抽象类 `Animal`，并且定义了一个抽象方法 `sayHi`。在实例化抽象类的时候报错了。

其次，抽象类中的抽象方法必须被子类实现：

```
abstract class Animal {
  public name;
  public constructor(name: string) {
    this.name = name;
  }
  public abstract sayHi(): void;
}

class Cat extends Animal {
  public eat() {
    console.log(`${this.name} is eating.`);
  }
}

let cat = new Cat('Tom');

// index.ts(9,7): error TS2515: Non-abstract class 'Cat' does not implement
inherited abstract member 'sayHi' from class 'Animal'.
```

上面的例子中，我们定义了一个类 `Cat` 继承了抽象类 `Animal`，但是没有实现抽象方法 `sayHi`，所以编译报错了。

下面是一个正确使用抽象类的例子：

```
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');
```

# 三十、类与接口，类继承接口，接口继承接口，接口继承类

## 类继承接口

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。

举例来说，门是一个类，防盗门是门的子类。如果防盗门有一个报警器的功能，我们可以简单的给防盗门添加一个报警方法。这时候如果有另一个类，车，也有报警器的功能，就可以考虑把报警器提取出来，作为一个接口，防盗门和车都去实现它：

```
interface Alarm {
  alert(): void;
}

class Door {
}

class SecurityDoor extends Door implements Alarm {
  alert() {
      console.log('SecurityDoor alert');
  }
}

class Car implements Alarm {
  alert() {
      console.log('Car alert');
  }
}
```

一个类可以实现多个接口：

```
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');
  }
}
```

上例中，`Car` 实现了 `Alarm` 和 `Light` 接口，既能报警，也能开关车灯。

## 接口继承接口

接口与接口之间可以是继承关系：

```
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`。

## 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};
```

但是这里不推荐这样去使用，我们在定义接口的时候只做定义，具体实现通过继承接口的类去实现。养成好习惯。

# 三一、泛型01，概念，简单示例

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。

首先，我们来实现一个函数 `createArray`，它可以创建一个指定长度的数组，同时将每一项都填充一个默认值：

```
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']
```

上例中，我们使用了之前提到过的数组泛型来定义返回值的类型。

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 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']
```

上例中，我们在函数名后添加了 `<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：

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

# 三二、泛型02，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]
```

上例中，我们定义了一个 `swap` 函数，用来交换输入的元组。

# 三三、泛型03，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：

```
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;
}
```

上例中，我们使用了 `extends` 约束了泛型 `T` 必须符合接口 `Lengthwise` 的形状，也就是必须包含 `length` 属性。

此时如果调用 `loggingIdentity` 的时候，传入的 `arg` 不包含 `length`，那么在编译阶段就会报错了：

```
interface Lengthwise {
    length: number;
}

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

loggingIdentity(7);

// index.ts(10,17): error TS2345: Argument of type '7' is not assignable to
parameter of type 'Lengthwise'.
```

# 三四、泛型04，泛型接口

之前学习过，可以使用接口的方式来定义一个函数需要符合的形状：

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}

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

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']
```

注意，此时在使用泛型接口的时候，需要定义泛型的类型。

# 三五、泛型05，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;
}
```

# 三六、声明合并，同名函数、接口、类的合并

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

之前学习过，我们可以使用重载定义多个函数类型：

```
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('');
    }
}
```

## 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;
}

// 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'.
```

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

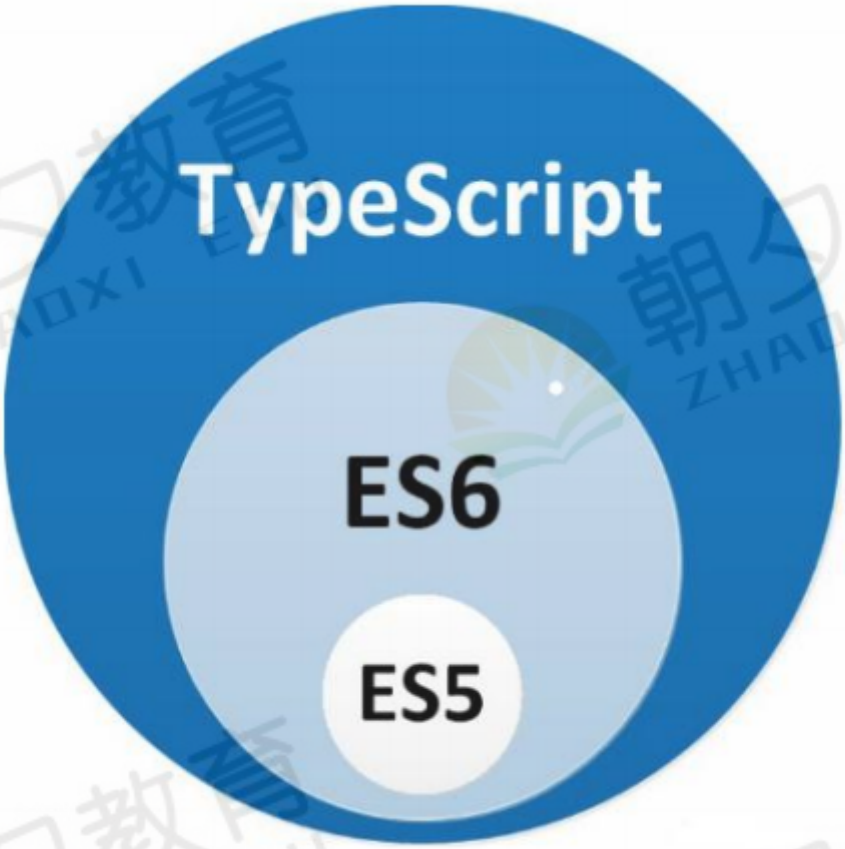
类的合并与接口的合并规则一致。

PS：但是一般情况下，不建议创建多个同名接口或者类，虽然可以自动合并，但是可能会发现意想不到的问题。代码不要写在两个地方，不然不好维护。

# 三七、写在结尾

TypeScript的应用非常广泛，最新的Vue和React均集成了TypeScript，这里推荐大家使用Vue3，Vue3天然支持TS。

另外一方面，TypeScript中有很多的ES语法，这里用一张图来示意两者之间的关系：




所以对ES5或者ES6不了解的同学，可以学习ES，了解一下相关的语法，这对于学习TypeScript也有一定的帮助。

在学习TS的过程中，也推荐大家多看官方文档，官网的文档比较详细和丰富：

https://www.tslang.cn/docs/home.html