## 初见 TypeScript

### TypeScript 简介

#### 什么是 TypeScript

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这个语言添加了可选的静态类型和基于类的面向对象编程。它扩展了 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。TypeScript 是为大型应用之开发而设计，而编译时它产生 JavaScript 以确保兼容性。

#### TypeScript 与 JavaScript 的区别

* 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。

用一张表格来更清晰的观察两者的区别：

|  | **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/服务端 | 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 |
| 参考选择 | 小型项目 | TS 是一种面向对象语言，代码更简洁，可读性和复用性强。因此 TS 更适合大型项目。 |

### 为何选择 TypeScript

* TypeScript 增加了代码的可读性和可维护性。
* 新增了其他语言的语法，比如 Class（类）、Interface（接口）、Generics(泛型)、Enums(枚举）等。
* TypeScript 拥抱了 ES6 规范。
* 兼容很多第三方库，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

更值得一提的是，TypeScript 在开发时就能给出编译错误，而 JavaScript 错误则需要在运行时才能暴露。作为强类型语言，你可以明确知道数据的类型，代码可读性极强，几乎每个人都能理解。TypeScript 被很多业界大佬使用，像 Asana、Circle CL 和 Slack 这些公司都在用 TypeScript。

### 安装使用 TypeScript

打开终端 terminal 输入全局安装命令：

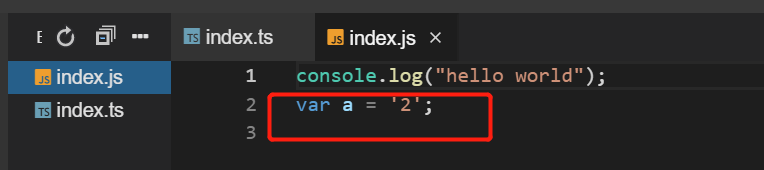
cnpm install -g typescript

新建一个文件index.ts，输入以下内容：

console.log("hello world");

var a: string = "2"; //这是ts写法，暂时不需要掌握，后续会讲到

在终端输入tsc index.ts编译文件，编译成功则会生成一个同名的 js 文件。



生成的 js 文件里则将 ts 语法转换成了 js 语法。

## 基本数据类型

### 基本数据类型有哪些

数据类型有原始数据类型和非原始数据类型两种。

#### 原始数据类型

原始数据类型有以下六种：

* boolean 布尔值
* number 数值
* string 字符串
* null 空值
* undefined 未定义
* Symbol （ES6 中的新类型）

本节主要介绍前五种原始数据类型在 TypeScript 中的应用。

#### 非原始数据类型

非原始数据类型有以下九种：

* 数组
* Tuple 元祖
* enum 枚举
* never 永不存在的值的类型
* void
* any 任意类型
* 联合类型
* 函数类型
* 对象类型

其中元组、枚举、任意值、void 类型和 never 类型是 TypeScript 有别于 JavaScript 的特有类型。在 TypeScript 中声明变量，需要加上类型声明，例如 boolean 和 string 等。通过静态类型约束，在编译时执行类型检查，可以避免一些类型混用的低级错误

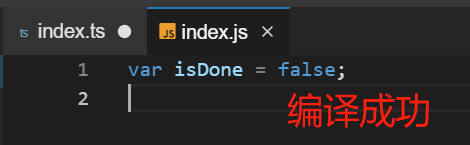
### 原始数据类型

#### 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;

运行tsc index.ts 进行编译，之后不再提醒。



#### 数字

和 JavaScript 一样，TypeScript 里的所有数字都是浮点数。 这些浮点数的类型是 number。 除了支持十进制和十六进制字面量，TypeScript 还支持 ECMAScript 2015 中引入的二进制和八进制字面量。

let a: number = 6;

// ES6 中的二进制表示法

let b: number = 0b1010;

// ES6 中的八进制表示法

let c: number = 0o744;



#### 字符串

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

let myName: string = "xiaoming";

let myAge: number = 25;

还可以使用模版字符串，它可以定义多行文本和内嵌表达式。 这种字符串是被反引号包围，并且使用$这种形式嵌入表达式。

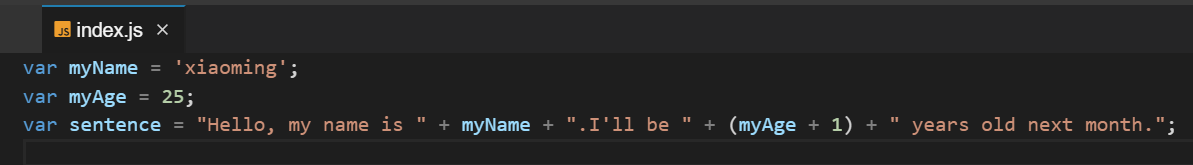
let myName: string = "xiaoming";

let myAge: number = 25;

let sentence: string = `Hello, my name is ${myName}.I'll be ${

myAge + 1

} years old next month.`;



#### 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 {

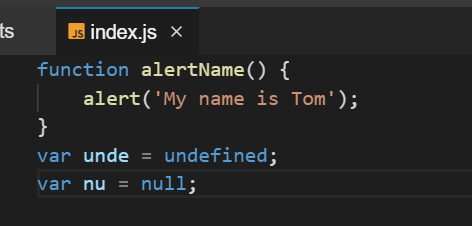
alert("My name is Tom");

}

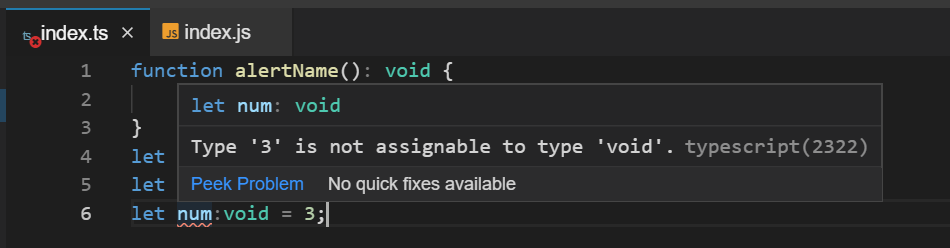
声明一个 void 类型的变量没有什么用，因为你只能将它赋值为 undefined 和 null：

let unde: void = undefined;

let nu: void = null;



当我们尝试把一个数字赋值给声明了void类型的变量。



这个时候就会给出错误提示。

#### 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 un: undefined = undefined;

let nu: 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 un: undefined;

let num2: number = un;

### 非原始数据类型

#### 数组

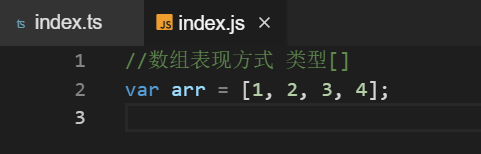
定义数组有两种方式。

**第一种：普通方式。**

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，数组的项中不允许出现其他的类型：

//数组表现方式 类型[]

let arr: number[] = [1, 2, 3, 4];



**第二种:泛型方式 Array<元素类型>。**

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

#### 元祖

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];

x = ["Hello", 10];

#### 枚举

使用枚举我们可以定义一些带名字的常量。 使用枚举可以清晰地表达意图或创建一组有区别的用例。 TypeScript 支持数字的和基于字符串的枚举。

// 利用 const 关键词也可以声明一组常量，例如，声明十二生肖的排位

const rat: number = 1;

const cattle: number = 2;

const tiger: number = 3;

const rabbit: number = 4;

const dragon: number = 5;

上述只声明了 5 个，如果声明全排位，需要声明 12 个变量，并且注明类型，但是却多了很多重复性工作，利用数字枚举，我们可以轻松声明同样的一组常量。

**数字枚举**

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。

enum ChineseZodiac {

rat,

cattle,

tiger,

rabbit,

dragon,

}

**字符串枚举**

在一个字符串枚举里，每个成员都必须用字符串字面量，或另外一个字符串枚举成员进行初始化。

enum Person {

name = "NAME",

age = "AGE",

love = "LOVE",

hobby = "HOBBY",

}

console.log(Person.name); // NAME

console.log(Person.hobby); // HOBBY

#### never

never 类型是任何类型的子类型，也可以赋值给任何类型，一般作为函数返回值。

// 返回never的函数必须存在无法达到的终点

function error(message: string): never {

throw new Error(message);

}

#### void

void 类型像是与 any 类型相反，它表示没有任何类型。 当一个函数没有返回值时，你通常会见到其返回值类型是 void。

function warn(): void {

console.log("This is my warning message");

}

#### any 任意类型

任意值（Any）用来表示允许赋值为任意类型。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。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 notSure: any = 4;

notSure = "这是字符串";

notSure = false;

当你不希望类型检查器对某些值进行检查而是直接让它们通过编译阶段的检查可以使用 any 类型来标记这些变量。

#### 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。

let num: string | number;

num = "seven";

num = 7;

联合类型使用 | 分隔每个类型。这里的 let num: string | number 的含义是，允许 num 的类型是 string 或者 number，但是不能是其他类型。

#### 对象类型

object 表示非原始类型，也就是除 number，string，boolean，symbol，null 或 undefined 之外的类型。

declare function create(o: object | null): void;

create({ name: 0 }); // OK

create(null); // OK

create(2); // Error

create("字符串"); // Error

create(false); // Error

create(undefined); // Error

注意：函数类型我们会放到后面进行学习。

### 类型断言

类型断言可以用来手动指定一个值的类型，即允许变量从一种类型更改为另一种类型。

语法格式：

<类型>值

或

值 as 类型

#### 方式一

let someAny: any = "my name is tony";

let strLength: number = (<string>someAny).length;

someAny 作为 number 类型是不可以使用字符串方法的，所以这个时候我们可以使用类型断言对此进行转换。

#### 方式二

let someAny: any = "my name is tony";

let strLength: number = (someAny as string).length;

最终的效果是一样的。

## 接口

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

### 接口的基本概念

在 TypeScript 中，我们使用接口（Interfaces）来定义对象的类型，在面向对象编程中接口都是比较常见且重要的概念。

**接口的作用：**

1. 对类的一部分行为进行抽象（类的部分后续会讲到）
2. 描述对象的形状

interface Person {

name: string;

age: number;

}

let tom: Person = {

name: "tom",

age: 18,

};

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。

**注意：**

接口一般首字母大写，也可以在接口之前加大写 I ，赋值的时候，变量的形状必须和接口的形状保持一致。不允许添加未定义的属性。

### 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;

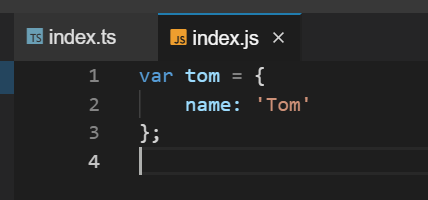
age?: number;

}

let tom: Person = {

name: "Tom",

};



在属性后使用 ? 表示这个属性是可选属性，可选属性的含义是该属性可以不存在，但是仍然不允许添加未定义的属性。

### 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;

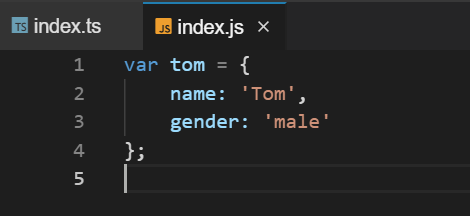
}

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;

}

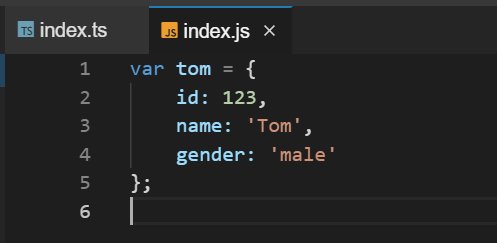
let tom: Person = {

id: 123,

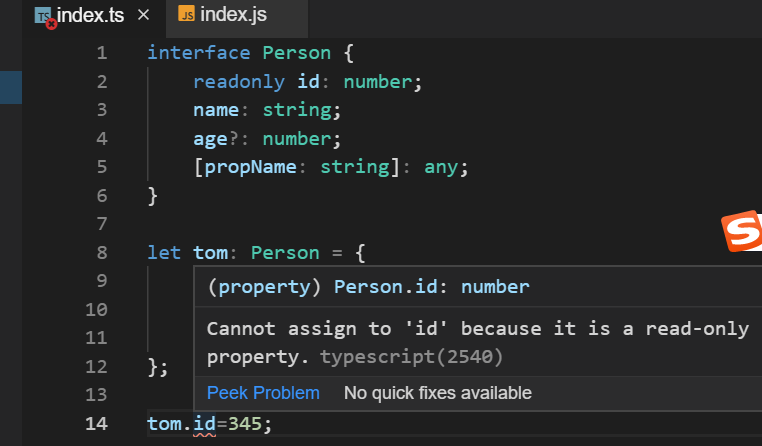
name: "Tom",

gender: "male",

};



如果使用 readonly 定义的属性 id 初始化后，又被赋值了，那么就会报错了：



## 类（Class）

传统方法中，JavaScript 通过构造函数实现类的概念，通过原型链实现继承，而在 ES6 中，我们能够使用class。TypeScript 除了实现了所有 ES6 中的类的功能以外，还添加了一些新的用法。这一节主要介绍类的用法，下一节再介绍如何定义类的类型。

### 类的概念

如果你有其他面向对象语言的经验，那么（Class） 类就是比较简单的常用的类型，如果你之前没有使用过，那么先介绍一下概念方便理解。

#### 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 run 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 run 方法，程序会自动判断出来应该如何执行 run。

#### 存取器（getter & setter）

用以改变属性的读取和赋值行为。

#### 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。

#### 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。

#### 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。

### 类的定义

使用 class 定义类，使用 constructor 定义构造函数。类一般包括属性 + 方法。

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {

alert("Hello, " + this.greeting);

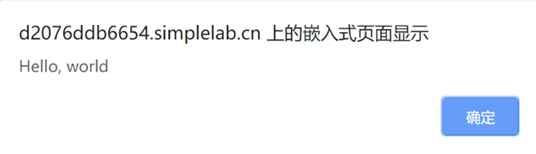
}

}

let greeter = new Greeter("world");

greeter.greet();

新建一个 index.html 文件，引入 index.js，打开显示：



### 类的继承

使用 extends 关键字实现继承，子类中使用 super 关键字来调用父类的构造函数和方法。

// 父类

class Person {

name: string;

constructor(name: string) {

// 构造函数，实例化类的时候触发的方法

this.name = name;

}

run(): void {

console.log(`${this.name}在运动`);

}

}

let p = new Person("父亲");

p.run(); // 父亲在运动

// 子类继承父类

class Child extends Person {

constructor(name: string) {

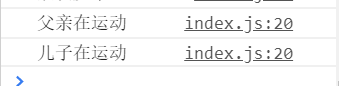
super(name); // 初始化父类的构造函数

}

}

let c = new Child("儿子");

c.run(); // 儿子在运动



注意：如果父类的方法和子类的方法一致，子类会执行子类自己的方法。

### 类里面的修饰符

Typescript 里面定义属性的时候给我们提供了三种修饰符：

* public：公有，在当前类里面、子类、类外面都可以访问。
* protected：保护类型，在当前类里面、子类里面可以访问，在类外部没法访问。
* private：私有，在当前类里面可以访问，子类、类外部都没法访问。

注意：属性如果不加修饰符，默认就是公有（public）。

#### public

class Person {

public name: string; // 公有

constructor(name: string) {

this.name = name;

}

run(): void {

console.log(`${this.name}在运动`); // 在类里能访问

}

}

let p = new Person("张三");

p.run();

console.log(p.name); // 在类外面能访问

class Child extends Person {

constructor(name: string) {

super(name);

}

run(): void {

console.log(`${this.name}在运动--子类`); // 子类能访问

}

}

let c = new Child("李四");

c.run(); // 李四在运动--子类

console.log(c.name); // 在类外面能访问

#### protected

class Person {

protected name: string; // 保护

constructor(name: string) {

this.name = name;

}

run(): void {

console.log(`${this.name}在运动`); // 在类里能访问

}

}

let p = new Person("张三");

p.run();

// console.log(p.name); // 报错，在类外面不能访问

class Child extends Person {

constructor(name: string) {

super(name);

}

run(): void {

console.log(`${this.name}在运动--子类`); // 子类能访问

}

}

let c = new Child("李四");

c.run(); // 李四在运动--子类

// console.log(c.name); // 报错，在类外面不能访问

#### private

class Person {

private name: string; // 私有

constructor(name: string) {

this.name = name;

}

run(): void {

console.log(`${this.name}在运动`); // 在类里能访问

}

}

let p = new Person("张三");

p.run();

// console.log(p.name); // 报错，在类外面不能访问

class Child extends Person {

constructor(name: string) {

super(name);

}

run(): void {

// console.log(`${this.name}在运动--子类`); // 报错，子类不能访问

}

}

let c = new Child("李四");

c.run(); // 李四在运动--子类

// console.log(c.name); // 报错，在类外面不能访问

### 静态属性和方法

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，若加上修饰符,和其他属性和方法效果一样。

ES5 中的静态方法、静态属性：

function Person() {

this.run1 = function () {}; // 实例方法，实例化后调用

}

Person.run2 = function () {}; // 静态方法，类名直接调用

Person.name = "lucy"; // 静态属性

Person.run2(); // 静态方法的调用

TypeScript 中的静态方法、静态属性：

class Person {

public name: string;

public age: number = 20;

static sex = "男"; // 静态属性

constructor(name: string) {

this.name = name;

}

run() {

console.log(`${this.name}在运动`);

}

work() {

console.log(`${this.name}在工作`);

}

static print() {

console.log("print静态方法" + Person.sex); // 静态方法，没法直接调用类的属性

}

}

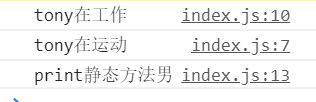
var p = new Person("tony");

p.work();

p.run();

Person.sex; // 男

Person.print();



### 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) {

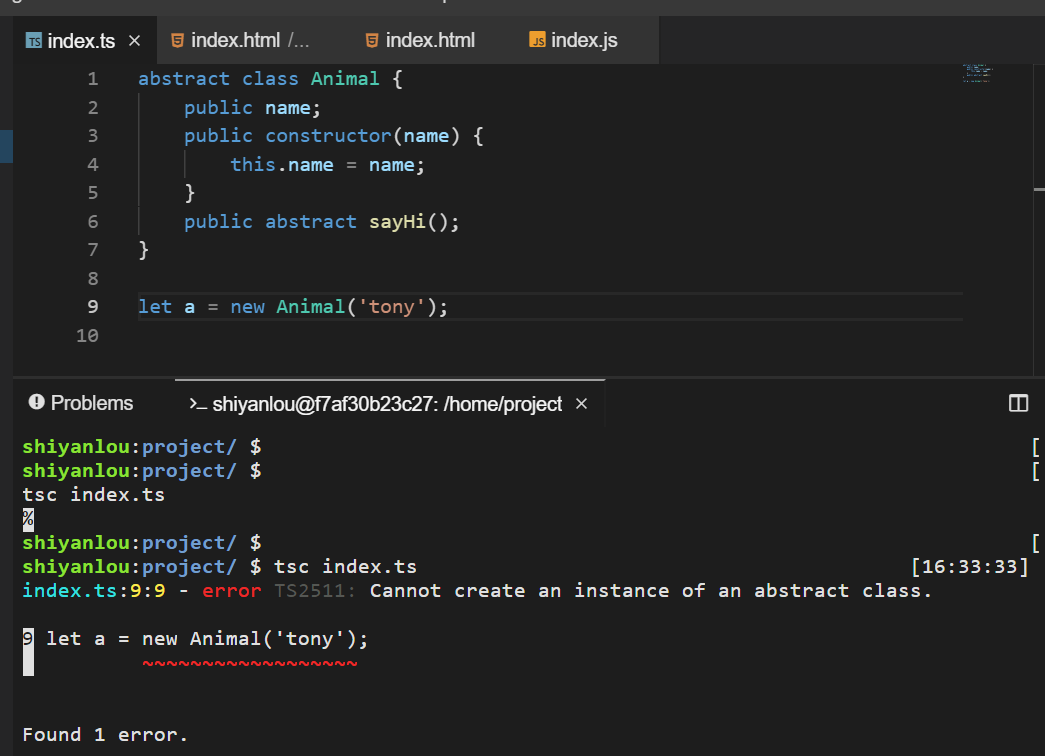
this.name = name;

}

public abstract sayHi();

}

let a = new Animal("tony");



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) {

this.name = name;

}

public abstract sayHi();

}

class Cat extends Animal {

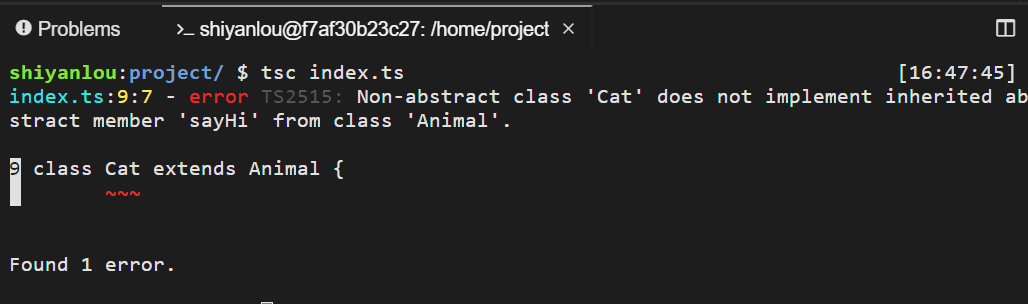
public eat() {

console.log(`${this.name} is eating.`);

}

}

let cat = new Cat("Tom");



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。

正确使用：

abstract class Animal {

public name;

public constructor(name) {

this.name = name;

}

public abstract sayHi();

}

class Cat extends Animal {

public sayHi() {

alert(`Meow, My name is ${this.name}`);

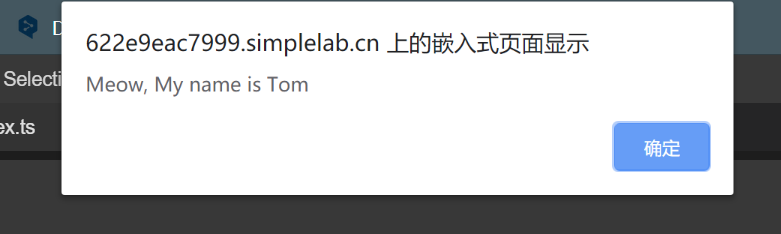
}

}

let cat = new Cat("Tom");

cat.sayHi();

执行效果：



## 函数

函数是 JavaScript 应用程序的基础。 它帮助你实现抽象层，模拟类，信息隐藏和模块。 在 TypeScript 里，函数仍然是主要的定义行为的地方。 TypeScript 为 JavaScript 函数添加了额外的功能，让我们可以更容易地使用。

### 函数的定义

和 JavaScript 一样，TypeScript 函数可以创建有名字的函数和匿名函数。 你可以随意选择适合应用程序的方式，不论是定义一系列 API 函数还是只使用一次的函数。

ES5 函数定义：函数声明、匿名函数、传参。

// 函数声明

function fn1() {

console.log(123);

}

fn1(); // 调用

// 匿名函数

let fn2 = function () {

console.log(456);

};

fn2();

// 传参

function fn3(name, age) {

console.log(`姓名：${name}，年龄：${age}`);

}

fn3("张三", 18);

TypeScript 函数定义：函数声明、匿名函数、传参。

// 函数声明

function fn1(): number {

// number 是函数返回值类型，没有返回值为 void

console.log("有返回值");

return 123;

}

fn1();

// 匿名函数

let fn2 = function (): void {

console.log(456);

};

fn2();

// 传参

function fn3(name: string, age: number): void {

console.log(`姓名：${name}，年龄：${age}`);

}

fn3("张三", 18);

### 可选参数

ES5 里面方法的实参和行参可以不一样，但是 TypeScript 中必须一样，如果不一样就需要配置可选参数。

function getInfo(name: string, age?: number): string {

// age 为可选参数

if (age) {

return `姓名：${name}，年龄：${age}`;

} else {

return `姓名：${name}，年龄：保密`;

}

}

console.log(getInfo("张三", 18));

console.log(getInfo("李四"));

**注意**：可选参数必须配置到参数的最后面。

### 默认参数

ES5 里面不能设置默认参数，ES6 和 TypeScript 中都可以设置默认参数。

function getInfo(name: string, age: number = 20): string {

// age 默认为20

if (age) {

return `姓名：${name}，年龄：${age}`;

} else {

return `姓名：${name}，年龄：保密`;

}

}

console.log(getInfo("张三", 18)); // 姓名：张三，年龄：18

console.log(getInfo("李四")); // 姓名：李四，年龄：20

### 剩余参数

ES6 中，可以使用 ...rest 的方式获取函数中的剩余参数：

function push(array, ...items) {

items.forEach(function (item) {

array.push(item);

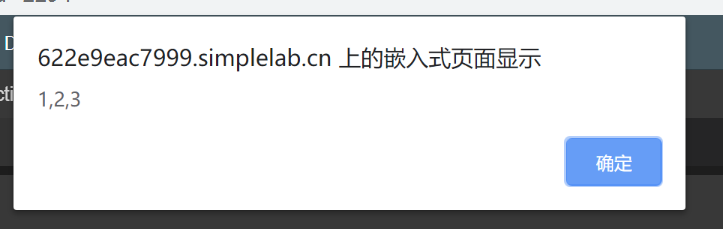
});

}

let a = [];

push(a, 1, 2, 3);

alert(a);



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，在 TypeScript 里，你可以把所有参数收集到一个变量里：

function push(array: any[], ...items: any[]) {

items.forEach(function (item) {

array.push(item);

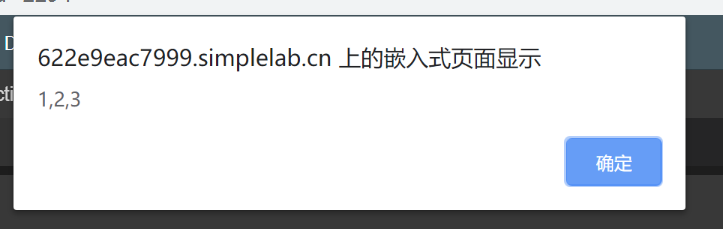
});

}

let a = [];

push(a, 1, 2, 3);

alert(a);



### 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 Fn {

(x: number, y: number): boolean;

//左边表示函数的输入类型，右边表示输出类型

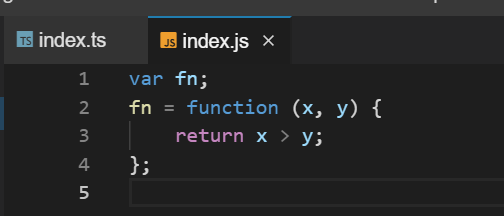
}

let fn: Fn;

fn = function (x: number, y: number) {

return x > y;

};



### 函数重载

JavaScript 中方法的重载：重载指的是两个或者两个以上同名函数，但它们的参数不一样，这时会出现函数重载的情况。

Typescript 中的重载：通过为同一个函数提供多个函数类型定义来实现多种功能的目的。

ES5 中出现同名方法，下面的会替换上面的方法：

function getInfo(name) {}

function getInfo(name, age) {}

Typescript 中的重载：

1. 单个参数，不同类型。

function getInfo(name: string): string;

function getInfo(age: number): string;

function getInfo(str: any): any {

if (typeof str === "string") {

alert(`姓名：${str}`);

} else {

alert(`年龄：${str}`);

}

}

getInfo("张三");

getInfo(18);

getInfo(true); // 错误的写法

1. 多个参数，可选参数。

function getInfo(name: string): string;

function getInfo(name: string, age: number): string;

function getInfo(name: string, age?: number): void {

if (age) {

alert(`姓名：${name}，年龄：${age}`);

} else {

alert(`姓名：${name}`);

}

}

getInfo("张三");

getInfo(18); // 错误

getInfo("李四", 20);

## 模块

从 ECMAScript 2015(ES6) 开始，JavaScript 引入了模块的概念。 TypeScript 也沿用这个概念。模块在其自身的作用域里执行，而不是在全局作用域里。这意味着定义在一个模块里的变量，函数，类等等在模块外部是不可见的，除非你使用 export 导出它们。 相反，如果想使用其它模块导出的变量，函数，类，接口等的时候，你可以通过 import 导入它们。本节将会学习模块的基础知识。

### 模块的导入与导出

模块是自声明的。在 TypeScript 里，两个模块之间的关系是通过在文件级别上使用 import 和 export 建立的。下面是一个基本例子：

//a.ts

export class Animal {

name: string;

show(): string {

return this.name;

}

}

//b.ts

import { Animal } from "./a";

let dog = new Animal();

dog.name = "狗狗";

dog.show();

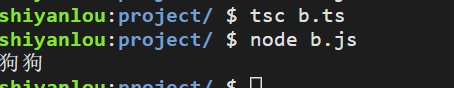
console.log(dog.name);

上面的例子里，在 a.ts 里声明了一个类 Animal，通过 export 导出。在 b.ts 里，指定相对文件路径，通过 import 导入，就可以使用 Animal 类。

执行以下命令进行编译：

$ tsc b.ts

$ node b.js



### 导入和导出的重命名

模块导入和导出时默认使用的内部对象的名称。TypeScript 也支持在导出前和导入后进行重命名。将上面的例子修改一下:

//a.ts

class Animal {

name: string;

show(): string {

return this.name;

}

}

export { Animal as ANI };

//b.ts

import { ANI as Animal } from "./a";

let dog = new Animal();

dog.name = "狗狗";

dog.show();

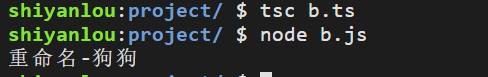
console.log("重命名-狗狗");

导入和导出时，通过 as 关键字对模块进行重命名。这个地方有一点要注意，当导出的模块重命名后，导入时重命名前的模块名要与导出重命名后的模块名保持一致，否则编译器将提示错误信息。以上面的这个例子为例，导出的模块被重命名为ANI，将此模块在另外一个文件 b.ts里导入时，as 关键字前面的模块名必须是 ANI。

执行以下命令进行编译：

$ tsc b.ts

$ node b.js



如果不知道导入模块的名称，则可以用 \* 号代替。

//b.ts

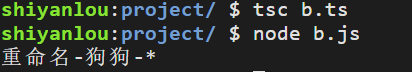
import \* as animal\_module from "./a";

let dog = new animal\_module.ANI();

dog.name = "狗狗";

dog.show();

console.log("重命名-狗狗-\*");



### 导出和导出多个对象

通常情况，模块里会定义多个类型对象，然后一并导出。而导入的模块里也可能会有多个。

//a.ts

enum Color {

Red = "Red",

Blue = "Blue",

White = "White",

}

class Animal {

name: string;

color: Color;

show(): string {

return this.name + this.color;

}

}

export { Animal, Color };

//b.ts

import \* as animal\_module from "./a";

let dog = new animal\_module.Animal();

dog.name = "狗狗";

dog.color = animal\_module.Color.Red;

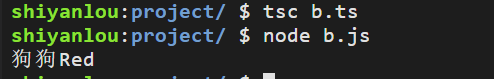
dog.show();

console.log(dog.name + dog.color);

执行以下命令进行编译：

$ tsc b.ts

$ node b.js



### 默认导出

对于导出内容的命名无关紧要，只要给定名称即可，但默认导出只有一个。

//a.ts

class Animal {

name: string;

show(): string {

return this.name;

}

}

export default Animal;

//b.ts

import Animal from "./a";

let dog = new Animal();

dog.name = "狗狗";

dog.show();

console.log(dog.name);

执行以下命令进行编译：

$ tsc b.ts

$ node b.js

