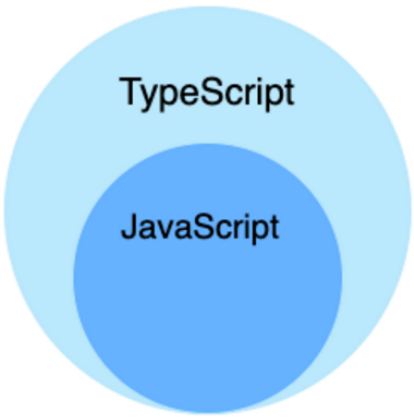
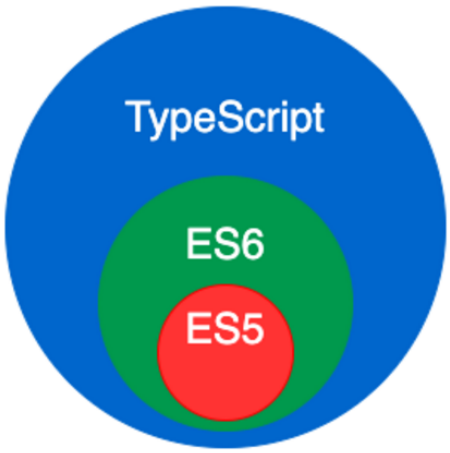
1. 什么是TypeScript

> TypeScript 是 JavaScript 的一个超集，支持 ECMAScript 6 标准（ES6 教程）。
>
> TypeScript 由微软开发的自由和开源的编程语言。
>
> TypeScript 设计目标是开发大型应用，它可以编译成纯 JavaScript，编译出来的 JavaScript 可以运行在任何浏览器上

2. TypeScript和JavaScript区别





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[2]，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。

1.JavaScript 是动态类型

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
```

2.TypeScript 是静态类型

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
```

3.安装

```
npm install -g typescript
```

安装完成后，命令行输入tsc，显示如下内容证明安装成功

```
C:\Users\Administrator>tsc
Version 4.7.4
tsc: The TypeScript Compiler - Version 4.7.4

COMMON COMMANDS

  tsc
  Compiles the current project (tsconfig.json in the working directory.)

  tsc app.ts util.ts
  Ignoring tsconfig.json, compiles the specified files with default compiler options.
```

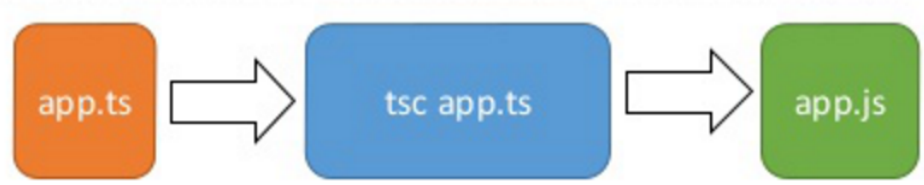
4.第一个ts案例

然后我们新建一个 app.ts 的文件，代码如下：

```
//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。
```

通常我们使用 .ts 作为 TypeScript 代码文件的扩展名。 然后执行以下命令将 TypeScript 转换为 JavaScript 代码：

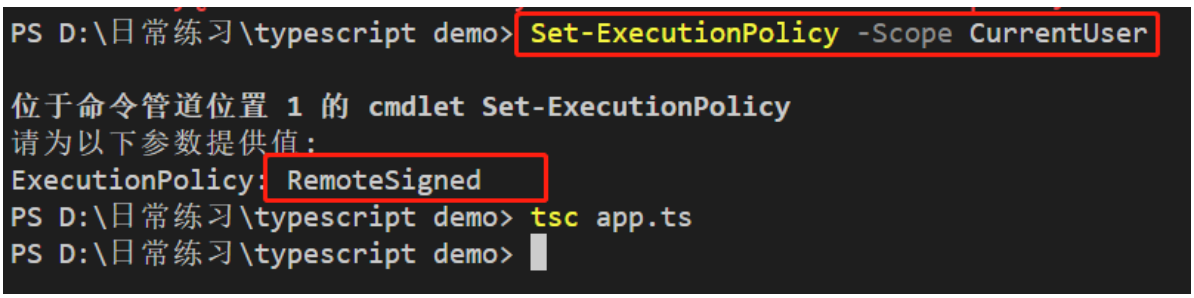
```
tsc app.ts
//
tsc app.ts -w //不需要每次重复执行编译命令
```



这时候再当前目录下（与 app.ts 同一目录）就会生成一个 app.js 文件，代码如下：

```
var message = "Hello World";
console.log(message);
```

如果报错：做如下更改即可



5.数据类型

1.原始数据类型

typescript要求在变量声明的时候要声明值的类型，如下

```
//布尔
let isDone:boolean=true
//数字
let age:number=18
//字符串
let firstName:string="张三"
//undefined
let u:undefined=undefined
//null
let n:null=null
//undefined和null是任意类型的子类，所以下面写法不会报错
let num:number=undefined

// 字面量 |: 或 &: 与
let d:'man'|'woman';
d="man";
d="woman";
let e:number|boolean;
e=1
e=false;
```

2.any类型

> any表示的是任意类型，可以任意赋值，一个变量设置类型为any，相当于对该变量关闭了类型检测，
>
> 如果声明一个变量不设置类型，其实也是any类型。所以在开发中不建议使用该方法。
>
> 那么如果确实有一个变量的类型不知道，可以选择unknown,表示未知类型

如果在声明的时候值的类型不确定

```
let notSure:any=4
notSure="hello"
notSure=true
```

3.unknown类型

> unknown类型是TypeScript在3.0版本新增的类型，它表示未知的类型，这样看来它貌似和any很像，但是还是有区别的，也就是所谓的“unknown相对于any是安全的”。怎么理解呢？我们知道当一个值我们不能确定它的类型的时候，可以指定它是any类型；但是当指定了any类型之后，这个值基本上是“废”了，你可以随意对它进行属性方法的访问，不管有的还是没有的，可以把它当做任意类型的值来使用，这往往会产生问题，如下：

> 任何类型也都可以是unknown类型，但与any完全相反，unknown类型就像是typescript给打上了一个重点检查的标签。在没有对它进行类型检查之前，ubnknow类型的变量是不能进行任何操作的。

```
let score: unknown =5
console.log(score*2)//error

let score: any =5
console.log(score*2)//success
```

那如何使unknown类型能正常使用呢?
1.很简单，就是让typscript编译器"看到"并且"相信"你的操作是合法安全的

```
let a: unknown =5
if(typeof a=="number"){
    console.log(a*2)
}
```

3.数组

> 最简单的方法是使用「类型 + 方括号」来表示数组:

```
let arr:number[]=[1,2,3,4,5]
//或
let arr:Array<number>=[1,2,3,4,5]
//数组的项中不允许出现其他的类型：
let b: number[] = [1, '1', 2, 3, 5];//error
//二维数组
let arry: number[][] = [[1,2,3]];
//对象数组
const objArr: { name: string, age: Number }
[] = [
  { name: "小明", age: 18 },
  { name: "小红", age: 28 },
];
```

往数组里添加的元素如果不符合也会提示

```
let arr:number[]=[1,2,3,4,5]
arr.push("hello") //error
```

数组也可以是any类型

```
let arr:any[]=[1,"hello",true]
```

4.对象

```
let a:object;
a={}
a=function(){}
```

我们这样限定类型其实没什么意义，限定a的值是对象，但是我们知道js中一切皆对象，这么限制其实限制不了什么。我们想限制的是对象中的属性，而不是限制他是不是对象.

我们可以做如下限制。

规定a是个对象并且必须包含一个name属性，并且值得是字符串

```
let a:{name:String}
a={};// Property 'name' is missing in type '{}' but required in type '{ name: String; }'
```

属性后面加问号，代表该属性可有可无

```
let a:{name:string,age?:number}
a={name:"hello"}  //正常执行，因为age可以不给
```

如果我只希望一个对象必须具有a属性，其他的属性有没有随意

```
let a:{name:string,[propName:string]:any}
//propName代表属性名，肯定是字符串，propName只是形参，可以换成别的名字
//可选属性的类型一定是其他类型的父类
a={name:"张三",age:18,salary:3500}
```

特殊的关于函数:

```
let fn:(a:number,b:number)=>number
//定义fn这个函数两个参数必须都是number类型，返回值必须要是number类型
```

1.关于内置对象

```
let b: Boolean = new Boolean(1);
let e: Error = new Error('Error occurred');
let d: Date = new Date();
let r: RegExp = /[a-z]/;
let body: HTMLElement = document.body;
let allDiv: NodeList = document.querySelectorAll('div');
```

5.函数：

1.基本使用

在定义函数的的时候函数的参数和函数的返回值都需要主动规定类型

如果某个参数是可选的，那么要加问号

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
add(x:number,y:number,z?:number):number{
    return x+y
}
//另一种写法也是允许的 let
add=function(x:number,y:number,z?:number):n
umber{return x+y}
add(1,2)//ok
add(1,2,3)//ok
add("hello",5)//error
```

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 add，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 add添加类型，则应该是这样：

```
let add: (x:number,y:number,z?:number) =>
number =
function(x:number,y:number,z?:number):
number {
    return x + y;
};
```

换成箭头函数的写法也可以

```
let add: (x:number,y:number,z?:number) =>
number = (x:number,y:number,z?:number):
number=> {
    return x + y;
};
```

推荐:

```
function fn(a:number,b:string):string{
    return "hello"
}
```

案例:

```
let add: (x: number, y: number) => number;
add = (arg1: number, arg2: number): number
=> arg1 + arg2;
add = (arg1: string, arg2: string): string
=> arg1 + arg2; // error
```

如果函数没有返回值，就是void类型，表示没有返回值，如果设置为void,又加了返回值会报错

```
function fn():void{
    return 123  //这里会报错，因为规定没有返回值
}
```

如果函数的返回值是数组

```
function fn(a:number):number[]{
    return [a,123];
}
```

如果函数的返回值是对象：

```
function fn(a:number):{age:number}{
    return {age:a}
}
fn(100);
```

如果返回值是函数：

```
function fn(a:number):(b:number)=>number{
    return function(b){return a+b}
}
fn(10)(20);
```

2.使用接口定义函数类型

```
interface Add {
  (x: number, y: number): number;
}
let add: Add = (arg1: string, arg2:
string): string => arg1 + arg2; // error 不
能将类型“(arg1: string, arg2: string) =>
string”分配给类型“Add”
```

3.使用类型别名

```
type a=number;
let box:a=5;
```

```
//type关键字可以为原始值、联合类型、元组以及任何我
们定义的类型起一个别名
type Add = (x: number, y: number) =>
number;
let add: Add = (arg1: string, arg2:
string): string => arg1 + arg2; // error 不
能将类型“(arg1: string, arg2: string) =>
string”分配给类型“Add”
```

```
type Directions = 'up' | 'Down' | 'Left'
|'Right'
let towhere : Directions ='Down'
```

对象数组使用类型别名

```
const objArr: { name: string, age: Number }
[] = [
  { name: "小明", age: 18 },
  { name: "小红", age: 28 },
];
//上面的案例使用类型别名简写
type Person = { name: string, age: Number
};
const arr:Person[]= [
  { name: "小明", age: 18 },
  { name: "小红", age: 28 },
];
```

6.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 a=1;
a="hello" //error: Type 'string' is not
assignable to type 'number'.
```

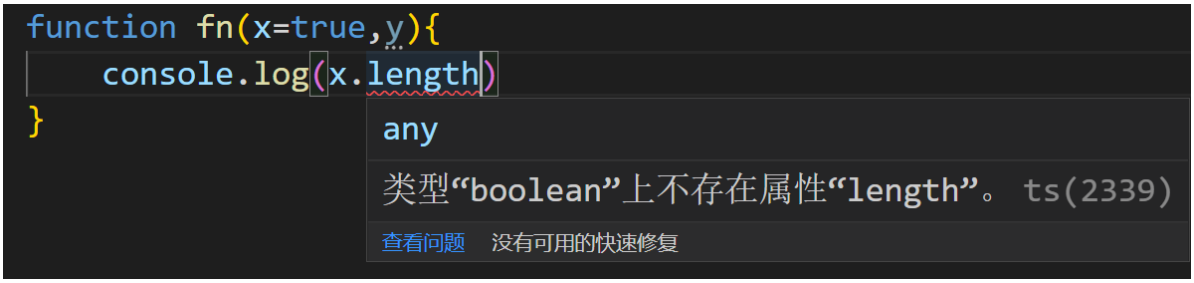
如果定义的时候没有赋值，不管之后有没有赋值，都会被推断成 any 类型而完全不被类型检查：

```
let a;
a="hello"; //正确
a=7; //正确
```

函数的参数的类型推论

```
function fn(x,y){ //x y被推断为any
    console.log(x+y)
}
```

函数被推断为boolean，所以下面的写法会报错



7.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。
var val:string|number |string[]|()=>number
val = 12 //ok
val = "Runoob" //ok
```

所以我们之前定义存储不同数据数组的时候，也可以使用联合类型

```
const arr: (number | string)[] = [1, "string", 2];
```

> 特殊的，如果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();
}
```

> 联合类型的写法

```
let a:string|number|string[]|
{name:string,age:number}|(()=>string);
a="hello";
a=123;
a=["a","b","c"]
a={name:"张三",age:18}
a=function(){
    return "hello"
}
```

8.接口

1.基本使用

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

> 接口其实就是相当于定义一个模板，以后声明的对象都得根据这个模板要求来

```
```

```
interface girl {
    height: number,
    like: boolean,
    money?:number, //可选属性
    readonly age:number //只读属性
}
interface person {
    name: string,
    money: number,
    like: boolean,
    hobby: string[],
    make: () => void,
    girlFriend: girl,
}
//实现接口
let zs: person = {
    name: "张三",
    money: 100,
    like: true,
    hobby: ["打篮球", "踢足球"],
    make: () => { },
    girlFriend: {
        height: 160,
        like: true,
        money:2000,
        age:18
    }
}
```

如果更改只读属性会报错

看起来readonly有点类似于const，不同之处在于，readonly是用在属性上面的，而const是用在值上面的

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'
};
```

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]: string;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};//error 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，所以错误。
```

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'
};
```

2.接口的继承

使用extends来继承

语法:

```
interface  接口名 extends 另一个接口名
```

只要存在接口的继承，那么我们实现接口的对象必须同时实现该接口以及他所继承的接口的所有属性

```
interface Animals{
   name:String
}
interface Cat {
    name:string,
    age:number
}

interface Dog{
    name:string,
    color:string
}
```

三个接口中都有对name的定义，但是这样写很繁琐，所以我们可以用继承来改写

```
interface person{
    name:string,
```

```
    age:number
}

interface boyFriend extends person{
    // name:string,
    // age:number,
    money:number,
    hobby:string[]
}

interface girlFriend{
    name:string,
    age:number,
    beauty:string
}

let xm:boyFriend={
    name:"小明",
    age:20,
    money:2000,
    hobby:["洗衣服","做饭","买菜"]
}
```

一个接口可以被多个接口继承，同样，一个接口也可以继承多个接口，多个接口用逗号隔开

```
interface Animals{
   name:String
```

```
}

interface Friends{
    like:String
}

interface Cat extends Animals {
    name:string,
    age:number
}

interface Dog extends Animals,Friends{
    name:string,
    color:string
}

const dog:Dog={
    name:"旺财",
    color:"white",
    like:"sleep"
}
```

3.接口中使用联合类型

```
interface Person {
    name: string;

    like:string[]|string|(()=>string);
}

let xm:Person={
    name:"小明",
    // like:"玩游戏"
    // like:["篮球","足球","网球"]
    like:():string=>"就是玩"
}
```

4.接口也可以用于定义数组

```
interface namelist {
    [index:number]:string
 }
let arr:namelist=["a","b","c"]
```

9.类型断言

1. 基本用法

类型断言就是我明确的知道我这个数据肯定是字符串，告诉编译器你不用检测他了。

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。
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。
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
*/
```

类型断言的用途：

> **1.将一个联合类型断言为其中一个类型**

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'.
```

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`
```

案例:

使用类型断言（*as关键字*）覆盖其类型推断：

```
let student = {}
// let student: {}

student.name = 'jack'
// 类型“{}”上不存在属性“name”。
student.age = 18
// 类型“{}”上不存在属性“age”。
```

TypeScript类型推断变量`student`是一个没有属性的对象, 所以添加对象时会报错

此时就可以定义一个具有`name`,`age`属性的接口,然后通过`as`关键字来进行类型断言

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

let student = {} as Person
// let student: Person

student.name = 'jack'
student.age = 18
----------------------------如下写法也是正确的-
-------------------------------------
interface Person{
  name:string;
  age: number
```

```
}

let student = <Person>{}
// let student: Person

student.name = 'jack'
student.age = 18
//as 关键字使用在需要断言数据类型之后，而<>使用在断言数据之前
```

> 2. 将任何一个类型断言为 any

```
const foo: number = 1;
foo.length=10//error 类型“number”上不存在属性“length”。
(foo as any).length = 1;//success
```

> 3.将any 断言为任意类型

```
const foo: any = 1;
console.log(foo.length)//success
console.log((foo as number).length)//error
```

> 4.将父类断言为子类

```
class Students{
    make(){
        console.log("i can make money")
    }
}
class Xm extends Students{
    run(){
        console.log("i can run")
    }
}
let a=new Students();
a.run() //类型“Students”上不存在属性“run”。
(a as Xm).run() // 正确
```

2.非空断言

```
let num:number | null | undefined

let num2 = num.toFixed(2) //报错
```

变量num在获取toFixed属性并调用时报错, 告知变量num有可能是null或undefined

而null和undefined类型的值是没有toFixed属性.

此时我们就可以使用非空断言

```
let num2 = num!.toFixed(2)
```

在变量num后面添加非空断言符合!, 来断言num是非空值

> 和其他类型断言一样, 这种断言只会让TypeScript不报错, 但不会改变代码运行时的行为
>
> 简单说就是在编译后运行时如果num变量的值是null 或 undefined ,JavaScript依然会报错

```
type numFn = () => number;
function myFn(numMsg: numFn | undefined) {
    // Object is possibly 'undefined'.
    // Cannot invoke an object which is possibly 'undefined'.
    const num1 = numMsg(); // error
    const num2 = numMsg!(); // ok
}
myFn(function (){
    return 1
})
```

3.双重断言(不推荐使用)

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

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

10.元祖：

> 我们知道数组中元素的数据类型都一般是相同的（any[] 类型的数组可以不同）
>
> 如果存储的元素数据类型不同，则需要使用元组。
>
> 元组中允许存储不同类型的元素，元组可以作为参数传递给函数。元祖的数量是固定的

```
const list: [string, number] = ['Sherlock', 1887]   // ok
const list1: [string, number] = [1887, 'Sherlock']  // error
//或者
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];
//下面写法是错误的
let tom: [string, number];
tom = ['Tom'];

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

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'.
```

11.枚举

1.数字枚举

字符串存在数据库会比较大

我们常常会有这样的场景，比如与后端开发约定订单的状态开始是0，未结账是1，运输中是2，运输完成是3，已收货是4。这样的纯数字会使得代码缺乏可读性。枚举就用于这样的场景。枚举可以让我们定义一些名字有意义的常量。使用枚举可以清晰地表达我们的意图。TypeScript支持基于数字枚举和字符串的枚举。

```
enum OrderStatus{
    Start = 1,
    Unpaid,
    Shipping,
    Shipped,
    Complete,
}
```

就像上面这样，我们通过数字来表达了订单状态。在实际的代码编写时，我们就直接使用 OrderStatus.Start来代替原本的数字1，这就使得代码具备了相当的可读性，甚至可以免去冗余的注释。

当只写Start = 1时，后面的枚举变量就是递增的

如果不写=1,那么默认初始值为0，依次递增

案例：

```
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
```

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
```

上面的例子中，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
```

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"};
```

案例:

```
// 修改起始编号
enum Color {
  Red = 2,
  Blue,
  Yellow
}
console.log(Color.Red, Color.Blue,
Color.Yellow); // 2 3 4
// 指定任意字段的索引值
enum Status {
  Success = 200,
  NotFound = 404,
  Error = 500
}
```

```
console.log(Status.Success,
Status.NotFound, Status.Error); // 200 404
500
// 指定部分字段，其他使用默认递增索引
enum Status {
  Ok = 200,
  Created,
  Accepted,
  BadRequest = 400,
  Unauthorized
}
console.log(Status.Created,
Status.Accepted, Status.Unauthorized); //
201 202 401
```

> **数字枚举**在定义值的时候，可以使用计算值和常量。但是要注意，如果某个字段使用了计算值或常量，那么该字段后面紧接着的字段必须设置初始值，这里不能使用默认的递增值了，来看例子：

```
const getValue = () => {
  return 0;
};
enum ErrorIndex {
  a = getValue(),
  b, // error 枚举成员必须具有初始化的值
  c
}
enum RightIndex {
```

```
  a = getValue(),
  b = 1,
  c
}
const Start = 1;
enum Index {
  a = Start,
  b, // error 枚举成员必须具有初始化的值
  c
}
```

2.反向映射：

> 我们定义一个枚举值的时候，可以通过 Enum['key']或者 Enum.key 的形式获取到对应的值 value。TypeScript 还支持反向映射，但是反向映射只支持数字枚举，我们后面要讲的字符串枚举是不支持的

```
enum Status {
  Success = 200,
  NotFound = 404,
  Error = 500
}
console.log(Status["Success"]); // 200
console.log(Status[200]); // 'Success'
console.log(Status[Status["Success"]]); //
'Success'
```

3.字符串枚举

> 字符串枚举值要求每个字段的值都必须是字符串字面量，或者是该枚举值中另一个字符串枚举成员

```
enum Message {
  Error = "Sorry, error",
  Success = "Hoho, success"
}
console.log(Message.Error); // 'Sorry, error'
```

再来看我们使用枚举值中其他枚举成员的例子：

```
enum Message {
  Error = "error message",
  ServerError = Error,
  ClientError = Error
}
console.log(Message.Error); // 'error message'
console.log(Message.ServerError); // 'error message'
```

4.异构枚举 （不推荐）

简单来说异构枚举就是枚举值中成员值既有数字类型又有字符串类型，如下：

```
enum Result {
  Faild = 0,
  Success = "Success"
}
```

12.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。

假设我们定义一个函数，它可以接收一个number类型做为参数，并且返回一个number类型。

```
function genericDemo(data: number): number
{
  return data;
}
```

按照以上的写法是没有问题的，但是如果我们要接受一个string并返回一个string呢？如果逻辑一样还要在写一遍吗？就像下面这样。

```
function genericDemo(data: string): string
{
  return data;
}
```

这显然代码是很冗余的，我们还有不使用any的写法吗

```
function genericDemo(data: any): any{
  return data;
}
```

使用any类型会导致这个函数可以接收任何类型的arg参数，这样就丢失了一些信息：传入的类型与返回的类型应该是相同的。如果我们传入一个数字，我们只知道任何类型的值都有可能被返回。

因此，我们需要一种方法使返回值的类型与传入参数的类型是相同的。 这里，我们使用了 类型变量，它是一种特殊的变量，只用于表示类型而不是值。

函数中使用泛型

```
function identity<T>(arg: T): T {
    return arg;
}
identity<Number>(100)
//或者identity(100)
```

我们给identity添加了类型变量T。 T帮助我们捕获用户传入的类型（比如：number），之后我们就可以使用这个类型。 之后我们再次使用了 T当做返回值类型。现在我们可以知道参数类型与返回值类型是相同的了。 这允许我们跟踪函数里使用的类型的信息。

我们把这个版本的identity函数叫做泛型，因为它可以适用于多个类型。 不同于使用 any，它不会丢失信息，像第一个例子那像保持准确性，传入数值类型并返回数值类型。

对象接口中使用泛型

```
interface Person<N> {
  name: string;
  age: N;
}
const xiong: Person<string> = {
  name: "xiong",
  age: "18"
// age: 18 //报错
};
```

类中使用泛型

```
class Person<T>{
  name:T,
  age:number
  constructor(name:T,age:number){
    this.name = name
    this.age = age
  }
}
const xiong = new Person<string>
('xiong',18)
console.log(xiong.name,xiong.age) // xiong
18
```

多个类型参数

```
function identity<T,U>(arg: T,arg2 : U): T
{
    return arg;
}
identity<Number,String>(4,"hello")
-----------------------------------------
-----------------------------------------
-
class Person<T, N> {
  name: T;
  age: N;
  job: string;
  constructor(name: T, age: N, job: string)
{
    this.name = name;
    this.age = age;
    this.job = job;
  }
}
const xiong = new Person<string, number>
("xiong", 18, "前端开发");
console.log(xiong); // {name: "xiong", age:
18, job: "前端开发"}
xiong.name = "xxxx"; //pass
```

> 泛型除了能使用基本类型 string number boolean等，同时也可以是接口，数组等

```
function identity<T>(arg: T): T {
    return arg;
}
//接口
interface person{
    name:string
}
identity<person>({name:"hello"});
//对象
identity<{name:string}>({name:"hello"});
//函数
function fn():string{
    return "haha"
}
identity<()=>string>(fn);
//数组
identity<number[]>([1,2,3]);
//字面量
identity<1>(1);
```

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：

> 接口的形式

```
interface Person {
  name: string;
  age: number;
}
//extends 约束了泛型 T 必须符合接口 Person 的形状，
function createPerson<T extends Person>
(person: T) {
  return person;
}
interface Xiaohong {
  name: string;
  age: number;
  count: number;
```

```
}
interface Xiaoming {
    age: number;
    count: number;
  }
createPerson<Xiaohong>({ name:
"xiongxiong", age: 18, count: 2021 });//正确
createPerson<Xiaoming>({ age: 18, count:
2021 });//错误
```

> 对象的形式

还可以写成如下形式

```
function getCnames<T extends { name: string
}>(entities: T[]):string[] {
    return entities.map(entity =>
entity.name);
  }
getCnames([{age:18,name:"xm"}])
```

> 数组的形式

```
function identity<T extends number[]>(arg:
T): void {

    arg.push(9);
}
```

类型参数约束

```
function getValue<T, U extend keyof T>(obj:
T, key: T){
  return obj[key]
}
let a = {a: 1, b: 2, c: 3}
getValue(a,"a");
getValue(a, "d");//error 提示
//Argument of type '"d"' is not assignable
to parameter of type '"a" | "b" | "c" .
```

13.class类

js中的class

```
    class Parent {
        constructor(x) {
            this.x = x;
            this.sayHello = function () {
                console.log("sayHello")
            }
        }
        getX() {
            console.log("getX==>", this.x)
        }
    }
```

ts中的class

```
//class 类 传值必须定义类型和值
class Person {

    name: string = 'zhangsan'
     // 定义实例属性 （只有实例化后才能调用）
    //  const per = new Person()
    // console.log(per.name)

    static age: number = 18
    // 定义类属性（静态属性），前加一个static关键字（直接通过类调用）
    // console.log(Person.name)

    readonly sex: string = '男'
    // readonly修饰属性表示不可以修改，只能读
```

```
    eat() {
        console.log('我在吃东西')
    }
    // 定义实例方法，（只有实例化后才能调用）
    //  const per = new Person()
    // console.log(per.name)

   static sleep() {
        console.log('我在吃东西')
    }
   // 定义类静态方法，前加一个static关键字（直接通过类调用）
   // console.log(Person.sleep)

}
```

1.关于构造函数

我们刚才name属性是直接赋值的，但是我们更多的时候是希望调用构造函数的时候才赋值

```
class Person {
    //注意构造函数要用到的属性需要提前定义
    name:string;
    age:number
    constructor(name:string,age:number){
        this.name=name;
        this.age=age;
    }
}
```

2.继承：

> 通过继承可以将多个类中共有的代码写在一个父类中
>
> 这样就只需要写一次即可让所有的子类都同时拥有父类中的属性和方法
>
> 如果子类和父类名字相同，则会覆盖掉父类方法（方法重写）

```
class Animal {
    name: string
    age: number

    constructor(name: string, age: number)
{
        console.log("父类的构造器调用了")
        this.name = name
        this.age = age
    }
```

```
    eat() {
        console.log('我在吃东西')
        console.log(this.name)
    }

    sleep() {
        console.log('我会睡觉觉')
    }
}

// 定义一个狗类
// 使Dog类继承Animal类
// 父类： Animal， 子类：Dog
class Dog extends Animal {
    sex: string
    constructor(name: string, age: number,
sex: string) {
        //注意子类的构造器必须主动调用父类的构造器
        super(name,age)
        this.sex = sex
    }
    run() {
        console.log('我在跑动')
    }
    //重写
    sleep() {
        console.log("好困")
    }
}
```

```
class Cat extends Animal {
    catch() {
        console.log('我在抓老鼠')
    }
}

const dog = new Dog('小狗', 5,'女')
dog.sleep()
console.log(dog)
const cat = new Cat('猫咪', 10)
 cat.catch()
 cat.sleep()
```

3.访问修饰符

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

关于public

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

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

> 关于private

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

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

使用 private 修饰的属性或方法，在子类中也是不允许访问的：

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

class Cat extends Animal {
  constructor(name) {
    super(name);
    console.log(this.name);//error 子类中不能访问私有属性
  }
}
```

> 关于protected

用 `protected` 修饰，则允许在子类(孙子级等也算)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：

```
class Animal {
  public name;
  private constructor(name) {
    this.name = name;
  }
}
class Cat extends Animal {
  constructor(name) {
    super(name);
  }
}

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

当构造函数修饰为 protected 时，该类只允许被继承：

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

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

4.抽象类

抽象方法只能出现在抽象类中

抽象类中不一定包含抽象方法

`abstract` 用于定义抽象类和其中的抽象方法。

> 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('Jack');//error,抽象类不能被实例化
```

> 其次，抽象类中的抽象方法必须被子类(可以是间接子类)实现(重写):

```
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');//error 子类没有实现父类的抽象方法
```