### 基础类型

#### 布尔值

boolean类型

|  |
| --- |
| let isDone: boolean = false; |

#### 数字

number类型，Typescript里所有数字都是浮点数

|  |
| --- |
| let decLiteral: number = 6; |

#### 字符串

string类型

|  |
| --- |
| let name: string = "bob"; |

#### 数组

两种方式定义数组：

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

在元素类型后面跟上[]，表示由此类型元素组成的一个数组。

使用数组泛型，Array<元素类型>：

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

#### 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

|  |
| --- |
| let x: [string, number];  x = ['hello', 10]; // ok  x = [10, 'hello']; // Error |

#### 枚举

enum类型

|  |
| --- |
| enum Color {Red, Green, Blue}  let c: Color = Color.Green; |

这里c: Color 表名c的值只能是0，1，2

编译后：

|  |
| --- |
| var Color;  (function (Color) {  Color[Color["Red"] = 0] = "Red";  Color[Color["Green"] = 1] = "Green";  Color[Color["Blue"] = 2] = "Blue";  })(Color || (Color = {})); |

**Color["Red"] = 0 且 Color[0] = "Red";**

默认情况下，从0开始为元素编号，但也可以手动编号：

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

全部手动编号：

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

#### Any

**可能是任意类型**，可以用any

区别于Object：

any类型允许给变量赋任何类型值，且可以使用具体类型值的方法；

但是Object只允许赋任何值，但不能使用对应值的方法；

|  |
| --- |
| let notSure: any = 4;  notSure.toFixed(); // okay  let prettySure: Object = 4;  prettySure.toFixed(); // Error |

#### 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，通常其返回值类型就是void

|  |
| --- |
| function warnUser(): void {  console.log("This is my warning message");  } |

声明一个void类型的变量没有什么大用，因为你只能为它赋予undefined和null

|  |
| --- |
| let unusable: void = undefined; |

#### Null和Undefined

TypeScript里，undefined和null两者各自有自己的类型分别叫做undefined和null。

默认情况下null和undefined是所有类型的**子类型**。 就是说你可以把 null和undefined赋值给number类型的变量。

但若当你制定了**—strictNullChecks** 标记，null和undefined只能赋值给void和他们各自；

|  |
| --- |
| // Not much else we can assign to these variables!  let u: undefined = undefined;  let n: null = null; |

#### Never

never类型表示的是那些**永不存在的值**的类型。

never类型是任何类型的子类型，可以赋值给任何类型；

但是，没有类型是never的子类型或可以赋值给never类型（除了never本身之外）；

#### Object

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

#### 类型断言

通过类型断言这种方式可以告诉编译器，“相信我，我知道自己在干什么”。

类似于其它语言里的强制类型转换（type casting），区别在于类型断言只是编译时的，不像类型转换一样具有运行时影响：

两种方式：

”尖括号“语法：

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

"as" 语法：

|  |
| --- |
| let someValue: any = "this is a string";  let strLength: number = (someValue as string).length; |

### 变量声明

#### 解构

数组：

用于函数参数：（可以指定类型）

|  |
| --- |
| function f([first, second]: [number, number]) {  console.log(first);  console.log(second);  }  f(input); |

对象：

|  |
| --- |
| let { a, ...passthrough } = o; |

指定类型：

|  |
| --- |
| let {a, b}: {a: string, b: number} = o; |

### 接口

#### 简介

|  |
| --- |
| interface LabelledValue {  label: string;  }  function printLabel(labelledObj: LabelledValue) {  console.log(labelledObj.label);  }  let myObj = {size: 10, label: "Size 10 Object"};  printLabel(myObj); |

myObj被传入printLabel中，需要经过LabelledValue的检查，所以myObj中必须含有string类型的属性label, 但也可以含有其他属性如size。

#### 可选属性

接口里的属性不全是必需的。

可选属性名字定义的后面加一个 **?** 符号。

|  |
| --- |
| interface SquareConfig {  color?: string;  width?: number;  } |

#### 只读属性

一些对象属性只能在对象创建的时候指定其值，之后不能修改。

在属性名前加**readonly**。

|  |
| --- |
| interface Point {  readonly x: number;  readonly y: number;  }  let p1: Point = { x: 10, y: 20 };  p1.x = 5; // error! |

**ReadonlyArray<T>**类型:

与Array<T>相似，只是把所有可变方法去掉了，保证数组创建之后不能被修改：

|  |
| --- |
| let a: number[] = [1, 2, 3, 4];  let ro: ReadonlyArray<number> = a;  ro[0] = 12; // error!  ro.push(5); // error!  ro.length = 100; // error!  a = ro; // error! |

最后一行，即便是把整个ReadonlyArray数组赋值给普通数组也不行，但可以用类型断言重写：

|  |
| --- |
| a = ro as number[]; |

#### 额外的属性检查

当传入的对象存在任何“目标类型”不包含的属性时，你会得到一个错误。

|  |
| --- |
| interface SquareConfig {  color?: string;  width?: number;  }  function createSquare(config: SquareConfig): { color: string; area: number } {  // ...  }  let mySquare = createSquare({ colour: "red", width: 100 }); |

此时typescript会检查colour属性不属于SquareConfig，从而报错。

绕开额外属性检查3种方法：

1、类型断言：

|  |
| --- |
| let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig); |

2、添加一个字符串**索引签名**：

|  |
| --- |
| interface SquareConfig {  color?: string;  width?: number;  [propName: string]: any;  } |

这里表示SquareConfig可以有任意数量的属性；

3、将对象赋值给另一个对象

|  |
| --- |
| let squareOptions = { colour: "red", width: 100 };  let mySquare = createSquare(squareOptions); |

这里squareOptions不会经过额外属性检查。

#### 函数类型

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

它就像是一个只有**参数列表**和**返回值类型**的函数定义。参数列表里的每个参数都需要名字和类型。

创建一个函数类型的变量，并将一个同类型的函数赋值给这个变量：

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

**函数参数名不需要和接口里定义的名字相匹配，按位置匹配：**

|  |
| --- |
| let mySearch: SearchFunc;  //函数参数按顺序对应接口里的参数，参数名可以与接口不对应  mySearch = function(src: string, sub: string): boolean {  let result = src.search(sub);  return result > -1;  } |

亦可不指定函数参数类型和返回值类型，因为mySearch是SearchFunc类型变量，

Typescirpt会推断出各参数类型和返回值类型，所以也可这样写：

|  |
| --- |
| let mySearch: SearchFunc;  // 省略了参数类型和函数返回值类型  mySearch = function(src, sub) {  let result = src.search(sub);  return result > -1;  } |

#### 可索引类型

使用接口描述那些“通过索引得到的类型”，比如a[10] 或 ageMap['john']。

可索引类型具有一个 **索引签名**，它描述了对象索引的类型，还有相应的索引返回值类型：

|  |
| --- |
| interface StringArray {  [index: number]: string;  }  let myArray: StringArray;  myArray = ["Bob", "Fred"];  let myStr: string = myArray[0];  或：  let myObj: StringArray = {  0: 'test0',  1: 'test1'  } |

当用 number去索引StringArray时会得到string类型的返回值。

TypeScript支持两种索引签名：**字符串和数字**

可以同时使用两种类型，但是数字索引的返回值必须是字符串索引返回值类型的子类型。因为：当使用 number来索引时，JavaScript会将它转换成string然后再去索引对象。 也就是说用 100（一个number）去索引等同于使用"100"（一个string）去索引，因此两者需要保持一致。

|  |
| --- |
| class Animal {  name: string;  }  class Dog extends Animal {  breed: string;  }  // 错误：使用数值型的字符串索引，有时会得到完全不同的Animal!  interface NotOkay {  [x: number]: Animal;  [x: string]: Dog;  } |

字符串索引声明了obj.property和obj["property"]两种形式都可以。

下面的例子里， name的类型与字符串索引类型不匹配，所以类型检查器给出一个错误提示：

|  |
| --- |
| interface NumberDictionary {  [index: string]: number;  length: number; // 可以，length是number类型  name: string // 错误，`name`的类型与索引类型返回值的类型不匹配  } |

上面定义了字符串索引的返回值必须是number类型，所以下面length和name也必须是string类型。

可以将索引签名设置为只读，这样就防止了给索引赋值：

|  |
| --- |
| interface ReadonlyStringArray {  readonly [index: number]: string;  }  let myArray: ReadonlyStringArray = ["Alice", "Bob"];  myArray[2] = "Mallory"; // error! |

#### 继承接口

接口可以互相继承：

|  |
| --- |
| interface Shape {  color: string;  }  interface Square extends Shape {  sideLength: number;  }  let square = <Square>{};  square.color = "blue";  square.sideLength = 10; |

一个接口可以继承多个接口，用逗号隔开即可：

|  |
| --- |
| interface Shape {  color: string;  }  interface PenStroke {  penWidth: number;  }  interface Square extends Shape, PenStroke {  sideLength: number;  }  let square = <Square>{};  square.color = "blue";  square.sideLength = 10;  square.penWidth = 5.0; |

#### 混合类型

一个接口同时包含上面提到的多种类型：

|  |
| --- |
| interface Counter {  (start: number): string; // 函数类型  interval: number; //  reset(): void;  }  function getCounter(): Counter {  let counter = <Counter>function (start: number) { };  counter.interval = 123;  counter.reset = function () { };  return counter;  }  let c = getCounter();  c(10);  c.reset();  c.interval = 5.0; |

### 类

#### 定义

|  |
| --- |
| class Greeter {  greeting: string;  constructor(message: string) { // 构造函数  this.greeting = message;  }  greet() {  return "Hello, " + this.greeting;  }  }  let greeter = new Greeter("world"); |

#### 继承

类从基类中继承了**属性**和**方法**。

|  |
| --- |
| class Animal {  name: string;  constructor(theName: string) { this.name = theName; }  move(distanceInMeters: number = 0) {  console.log(`${this.name} moved ${distanceInMeters}m.`);  }  }  class Snake extends Animal {  constructor(name: string) { super(name); }  move(distanceInMeters = 5) {  console.log("Slithering...");  super.move(distanceInMeters);  }  } |

子类继承父类的构造函数，必须先调用super(), 之后才能使用this

注意： super() 调用父类的constructor()方法

super.prop() 调用父类上的方法

#### 修饰符

**public**：默认，可省略

**private**：不能在声明它的类的外部访问（不能在**子类**和**实例**中访问）

|  |
| --- |
| class Animal {  private name: string;  constructor(theName: string) { this.name = theName; }  }  new Animal("Cat").name; // 错误: 'name' 是私有的. |

protected：可以在子类中访问，但不能再实例中访问。

readonly: 将属性设置为只读，只读属性必须在声明时或构造函数里被初始化。

**参数属性**：方便地让我们在一个地方定义并初始化一个成员。

在构造函数里使用readonly name: string 参数来创建和初始化name成员，把声明和赋值合并一处。

|  |
| --- |
| //原始  class Octopus {  readonly name: string;  readonly numberOfLegs: number = 8;  constructor (theName: string) {  this.name = theName;  }  }  // 使用参数属性  class Octopus {  readonly numberOfLegs: number = 8;  constructor(readonly name: string) {  }  } |

参数修饰符也可以用public和protected。

#### 存取器

get方法：访问实例属性时调用；

set方法：设置实例属性值时调用

|  |
| --- |
| let passcode = "secret passcode";  class Employee {  private \_fullName: string;  get fullName(): string {  return this.\_fullName;  }  set fullName(newName: string) {  if (passcode && passcode == "secret passcode") {  this.\_fullName = newName;  }  else {  console.log("Error: Unauthorized update of employee!");  }  }  }  let employee = new Employee();  employee.fullName = "Bob Smith";  if (employee.fullName) {  alert(employee.fullName);  } |

注意：只带有 get不带有 set的存取器自动被推断为 **readonly**。

#### 静态属性

静态属性存在于类本身上面而不是类的实例上，所以使用 **类名.** 访问静态属性；

|  |
| --- |
| class Grid {  static origin = {x: 0, y: 0};  calculateDistanceFromOrigin(point: {x: number; y: number;}) {  let xDist = (point.x - **Grid**.origin.x);  let yDist = (point.y - **Grid**.origin.y);  return Math.sqrt(xDist \* xDist + yDist \* yDist) / this.scale;  }  constructor (public scale: number) { }  } |

origin是静态属性，所以用Grid.origin来访问

scale是公共属性，所以用this.scale来访问（this代表实例）

#### 抽象类

**abstract**关键字是用于定义抽象类和在抽象类内部定义抽象方法。

抽象类一般不直接被实例化。

抽象类中的抽象方法不包含具体实现（不含方法体），必须在子类中实现。

|  |
| --- |
| abstract class Department {  constructor(public name: string) {  }  printName(): void {  console.log('Department name: ' + this.name);  }  **public** **abstract** printMeeting(): void; // 必须在派生类中实现  }  class AccountingDepartment extends Department {  constructor() {  super('Accounting and Auditing'); // 在派生类的构造函数中必须调用 super()  }  printMeeting(): void { // 实现了抽象方法  console.log('The Accounting Department meets each Monday at 10am.');  }  generateReports(): void {  console.log('Generating accounting reports...');  }  }  let department: Department; // 允许创建一个对抽象类型的引用  department = new Department(); // 错误: 不能创建一个抽象类的实例  department = new AccountingDepartment(); // 允许对一个抽象子类进行实例化和赋值  department.printName();  department.printMeeting();  department.generateReports(); // 错误: 方法在声明的抽象类中不存在 |

abstract前还可以跟访问修饰符，public/protected/private

类定义是会创建两个东西：类的实例类型和一个构造函数

let department: Department 意思是：实例**department的类型是Department**

**实例的类型是类**

**类的类型是function**

### 函数