### 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  let y: [string, number, boolean**?**]// 可以用？号代表可选 |

越界时候，只能push string | number的联合类型

#### 枚举

enum类型

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

这里c: Color 表明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; |

常量枚举：

|  |
| --- |
| **const** enum Color {Red = 1, Green = 2, Blue = 4}  Color[‘Red’] // 只能通过string类型的属性访问，不能通过索引1、2、4访问了 |

#### 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 |

any类型的变量可以赋值给其他类型，但这样很**危险**，因为它改变了b的类型：

|  |
| --- |
| let a: any;  a=20;  let b: string = 'hello';  b = a; // 不报错 |

未声明类型的变量：

变量如果在声明的时候，未指定其类型，**且未赋值，**那么它会被识别为任意值类型，不走类型推论：

|  |
| --- |
| let something; // 未赋值  something = 'seven';  something = 7; |

等价于：

|  |
| --- |
| let something: any;  something = 'seven';  something = 7; |

但区别于类型推论：

|  |
| --- |
| let myFavoriteNumber = 'seven';  myFavoriteNumber = 7; // error  // 等价于  let myFavoriteNumber: **string** = 'seven';  myFavoriteNumber = 7; |

若声明时没有指明类型，但直接赋了值，会推论出他的类型

#### unknown

相当于类型安全的any

也能将任意值赋值给unknown类型的变量

unknown类型的值不能直接赋值给其他类型：

|  |
| --- |
| let a: unknown;  a=20;  let b: string = 'hello';  b = a; // 不能将类型“unknown”分配给类型“string”。 |

#### 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; |

使用了--strictNullChecks，可选参数会被自动地加上 | undefined:

|  |
| --- |
| function f(x: number, y?: number) {  return x + (y || 0);  }  f(1, 2);  f(1);  f(1, undefined);  f(1, null); // error, 'null' is not assignable to 'number | undefined' |

可选属性同上

|  |
| --- |
| class C {  a: number;  b?: number;  }  let c = new C();  c.a = 12;  c.a = undefined; // error, 'undefined' is not assignable to 'number'  c.b = 13;  c.b = undefined; // ok  c.b = null; // error, 'null' is not assignable to 'number | undefined' |

#### 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。

#### 联合类型

let myFavoriteNumber: string | number

只能访问此联合类型的所有类型里共有的属性或方法。

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 |

& 表示要同时满足：

|  |
| --- |
| let a: {name: string} & {age: number};  a = {  name: 'yang',  age: 18  } |

#### 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：

使用断言，简单来说就是先做好一个假设，使得编译通过。

类型断言更像是类型的选择，而不是类型转换

类型断言只是用来判断变量的数据类型，是不能用来转换变量的数据类型。

link: https://www.cnblogs.com/wjaaron/p/11697275.html

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; |

用途：

* 将一个联合类型断言为其中一个类型
* 将一个父类断言为更加具体的子类

|  |
| --- |
| 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;  }  const obj = {name: 'haha', message: 'xixi', test: 123}  const res = isApiError(obj);  // 这里可以通过将对象字面量赋值给一个变量的方法，传入Error类型的子类型，所以是可以传入子类型的。这才有了将参数断言为子类的可能 |

虽然可以这么断言，但是若实际的传值，并不含子类专属属性，有可能会在执行时报错！！

* 将任何一个类型断言为 any
* 将 any 断言为一个具体的类型

#### const 断言

没有类型扩展的字面类型：

|  |
| --- |
| const x = 'x'; // has the type 'x'  let y = 'x'; // has the type string |

|  |
| --- |
| let y = 'x' as const; // y has type 'x' |

对象字面量获取只读属性：

仍然可以为type赋值，type被扩展为了string

|  |
| --- |
| let action = <const>{  type: 'SET\_COUNT',  payload: 1  }  // 其实发生的是这样的, 两个属性都不能被修改赋值了  {  readonly type: "SET\_COUNT";  readonly payload: number  }; |

数组字面量成为只读元组:

|  |
| --- |
| const action = <const>{  type: 'SET\_HOURS',  payload: [8, 12, 5, 8]  }  // 相当于转为元祖  {  readonly type: "SET\_HOURS";  readonly payload: readonly [8, 12, 5, 8];  } |

#### 类型断言的限制

TypeScript 是结构类型系统，类型之间的对比只会比较它们**最终**的结构，而会忽略它们定义时的关系。

|  |
| --- |
| interface Animal {  name: string;  }  interface Cat {  name: string;  run(): void;  }  let tom: Cat = {  name: 'Tom',  run: () => { console.log('run') }  };  let animal: Animal = tom; |

Cat 包含了 Animal 中的所有属性，除此之外，它还有一个额外的方法 run。TypeScript 并不关心 Cat 和 Animal 之间定义时是什么关系，而只会看它们最终的结构有什么关系——所以它与 Cat extends Animal 是等价的：

|  |
| --- |
| interface Animal {  name: string;  }  interface Cat extends Animal {  run(): void;  } |

即：**Animal 兼容 Cat (父兼容子)**

这样Animal和Cat可以**互相**断言了：

|  |
| --- |
| interface Animal {  name: string;  }  interface Cat {  name: string;  run(): void;  }  // 父类可以被断言为子类  // 因为可以通过将对象字面量赋值给另一个对象，来绕过额外的类型检查，所以可以传Animal的子类实例进去。  function testAnimal(animal: Animal) {  return (animal as Cat);  }  //但此时断言成子类后使用了Animal中不存在的属性，将有可能报错。  function testAnimal(animal: Animal) {  return (animal as Cat)**.run();**  }  testAnimal({name:”test”});// error  // 子类肯定属于父类  function testCat(cat: Cat) {  return (cat as Animal);  } |

**结论：**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

#### 非空断言

name**!**.trim() // 用感叹号忽略变量的 undefined | null

#### 类型别名

给一个类型起个新名字。

使用 type创建：

|  |
| --- |
| **type** Name = string;  type NameResolver = () => string;  type NameOrResolver = Name | NameResolver;  function getName(n: NameOrResolver): Name {  if (typeof n === 'string') {  return n;  } else {  return n();  }  } |

#### 字符串字面量类型

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  } |

|  |
| --- |
| let a: 10;  a=20; // 不能将类型“20”分配给类型“10”。ts(2322)  let a: 10 | 20; // 表示或，联合类型，可以  a=20; |

#### 用字面量进行类型声明

#### 类型守卫

我们一旦检查过类型，就能在之后的每个分支里清楚地知道变量的类型，从而不再每个地方都用类型断言了

#### 类型判定

|  |
| --- |
| function isFish(pet: Fish | Bird): **pet is Fish** {  return (<Fish>pet).swim !== undefined;  } |

参数 is 类型的语法，若返回值为true,则断言pet是Fish类型的，后面就不用重复断言了

每当使用一些变量调用isFish时，TypeScript会将变量缩减为那个具体的类型，

|  |
| --- |
| // 'swim' 和 'fly' 调用都没有问题了  if (isFish(pet)) {  pet.swim();  }  else {  pet.fly();  } |

注意TypeScript不仅知道在if分支里pet是Fish类型； 它还清楚在else分支里，一定\_不是\_Fish类型，一定是Bird类型。

#### in操作符

x in n

x为字符串，n为**联合类型**，表示n里是否含有x属性

若为true,表示含有，若false，表示没有

|  |
| --- |
| function move(pet: Fish | Bird) {  if ("swim" in pet) {  return pet.swim();  }  return pet.fly();  } |

#### typeof 类型守卫

typeof v === "typename" 可以被识别为类型守卫，若为true,后面就判定x的类型了，就不用类型断言了。

只有两种可以被识别为类型守卫：typeof v === "typename"和typeof v !== "typename"

"typename"必须是"number"，"string"，"boolean"或"symbol"

|  |
| --- |
| function padLeft(value: string, padding: string | number) {  if (typeof padding === "number") {  // 下面就不用类型断言了  return Array(padding + 1).join(" ") + value;  }  if (typeof padding === "string") {  return padding + value;  }  throw new Error(`Expected string or number, got '${padding}'.`);  } |

#### instanceof类型守卫

|  |
| --- |
| // 类型为SpaceRepeatingPadder | StringPadder  let padder: Padder = getRandomPadder();  if (padder instanceof SpaceRepeatingPadder) {  padder; // 类型细化为'SpaceRepeatingPadder'  }  if (padder instanceof StringPadder) {  padder; // 类型细化为'StringPadder'  } |

#### 高级类型

##### 映射类型(从旧类型中创建新类型的一种方式)

Partial<Person> 将Person属性都变为可选

|  |
| --- |
| type Partial<T> = {  [P in keyof T]?: T[P];  } |

Readonly<T>

|  |
| --- |
| type Readonly<T> = {  readonly [P in keyof T]: T[P];  } |

Pick<Person, ‘name’> 从Person里取出name属性， **K可以是联合类型**

|  |
| --- |
| type Pick<T, K extends keyof T> = {  [P in K]: T[P];  } |

Omit<Person, ‘name’> 从Person里去除name属性

|  |
| --- |
| type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>; |

Record<'prop1' | 'prop2' | 'prop3', string> 将是三个属性都设置成string类型

|  |
| --- |
| type Record<K extends string, T> = {  [P in K]: T;  } |

##### 索引类型

keyof T的结果为 T上已知的公共属性名的联合,即一个**联合类型**

|  |
| --- |
| interface Person {  name: string;  age: number;  }  let personProps: keyof Person; // 'name' | 'age'  K extends keyof T 表示K属于 T属性里的一种 |

索引访问：

T[K]

|  |
| --- |
| function pluck<T, K extends keyof T>(o: T, names: K[]): **T[K]**[] {  return names.map(n => o[n]);  }  interface Person {  name: string;  age: number;  }  let person: Person = {  name: 'Jarid',  age: 35  };  let strings: string[] = pluck(person, ['name']); // ok, string[] |

T[K]对于上面例子，就是string或number

##### 字符串索引

字符串索引签名的类型，那么keyof T会是string | number，因为在JavaScript里，你可以使用字符串object['42']或 数字object[42]索引来访问对象属性

|  |
| --- |
| interface Dictionary {  [key: string]: string;  }  let keys: keyof Dictionary; // string | number |

##### 交叉类型

T & U

将多个类型合并为一个类型，需要同时满足T 和 U的所有属性

##### 有条件类型

见印象笔记

### 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); |

解构作用于已声明的变量会更好：

|  |
| --- |
| let first = 1,second =2;  [first, second] = [second, first]; // 可以对已经定义过的变量重新赋值 |

对象：

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

指定类型：

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

解构作用于已声明的变量：

|  |
| --- |
| let a=1,b=2;  **(** {a,b} = {a: 3,b:4} **)**;  //此处要加（）,因为js通常会将以 { 起始的语句解析为一个块 |

### 接口

#### 简介

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**

1. **接口用来定义一个类的结构，定义一个类应该含哪些属性和方法；**

区别抽象类，接口里的方法不能有方法体，但抽象类里的方法是可以有方法体的。

|  |
| --- |
| interface Person {  name: string;  say(): void;  }  class Man implements Person {  name: string = ‘yang’;  constructor(name: string) {  this.name = name;  }  say(){  console.log(2222)  }  } |

**2.接口可以被当做类型声明去使用；**

**接口和类型声明不同：**

**接口可以重复声明，类型声明只能声明一次**

|  |
| --- |
| interface SquareConfig {  color: string;  width: number;  }  interface SquareConfig {  name: string  }  // 最终两者会合并 |

#### 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

对象字面量{color: 1,width:100}会被特殊对待而且会经过 额外属性检查。

如果一个对象字面量存在任何“目标类型”不包含的属性时，你会得到一个错误。

但前提是，目标类型里必须的属性，必须都有;

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

**大前提：传入的参数，必须包含接口里的必传属性！！**

1、类型断言：

|  |
| --- |
| let mySquare = createSquare({color:”blue”, 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, color: “blue” };  let mySquare = createSquare(squareOptions); |

这里squareOptions不会经过额外属性检查。

**但这里因为color和width都是必须参数，所以即使是赋值给另一个对象，也要都有，不然还是会报错。**

**结论：将对象字面量直接传入参数，如果存在接口没有定义的属性，会报错，但是，若将对象赋给另一个变量传入，则不会经过额外的参数检查；**

#### 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;  } |

注意:

**函数类型的接口和 typescript的函数类型注意区分：**

**函数类型的接口：**

|  |
| --- |
| interface SearchFunc {  (source: string, subString: string): boolean; // 用冒号分割，前参数，后返回值  } |

**ts的函数类型：**

|  |
| --- |
| let myAdd: (x: number, y: number) => number =  function(x: number, y: number): number { return x + y; }; |

**前面是参数,中间 => ，后面是返回值**

#### 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']。

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

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：

当用 number

|  |
| --- |
| interface StringObj {  [prop: string]: string; // 表示属性名是字符串类型，值是字符串类型  }  let obj : StringObj = {  name: 'yang',  hobby: 'sing'  } |

定义对象：

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。

**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：**

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! |

#### 类类型

类实现接口：

|  |
| --- |
| interface ClockInterface {  currentTime: Date;  setTime(d: Date);  }  class Clock implements ClockInterface {  currentTime: Date;  setTime(d: Date) {  this.currentTime = d;  }  constructor(h: number, m: number) { }  } |

类类型只表述了类的公共部分，他不会帮你检查类是否具有某些私有成员。

类静态部分和实例部分：

当一个类实现（implements）了一个接口时，只对其实例部分进行类型检查。 constructor存在于类的静态部分，所以不在检查的范围内。

|  |
| --- |
| interface ClockConstructor {  new (hour: number, minute: number): ClockInterface;  }  interface ClockInterface {  tick();  }  function createClock(ctor: ClockConstructor, hour: number, minute: number): ClockInterface {  return new ctor(hour, minute);  }  //implements 继承会检查DigitalClock里的tick方法。但constructor属于静态方法，不会检查。  class DigitalClock implements ClockInterface {  constructor(h: number, m: number) { }  tick() {  console.log("beep beep");  }  }  class AnalogClock implements ClockInterface {  constructor(h: number, m: number) { }  tick() {  console.log("tick tock");  }  }  // DigitalClock作为createClock第一个参数，会进行ClockConstructor类型检查  let digital = createClock(DigitalClock, 12, 17);  let analog = createClock(AnalogClock, 7, 32); |

#### 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; // 方法，区别于函数类型  }  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; |

#### 接口继承类

当接口继承类类型时，只会继承类的成员（包括private和protected成员）；

所以，当创建了一个接口继承了一个拥有私有或受保护的成员的类时，这个接口类型只能被这个类或其子类所实现（implement）,因为私有或受保护成员不能再接口里重新定义。

因为类可以创建出类型，所以你能够在允许使用接口的地方使用类。

|  |
| --- |
| class Point {  x: number;  y: number;  }  interface Point3d extends Point {  z: number;  }  let point3d: Point3d = {x: 1, y: 2, z: 3}; |

#### 函数类型接口和类类型接口重点区分

函数类型：

|  |
| --- |
| interface SearchFunc {  **(source: string, subString: string)**: boolean; // 函数，无函数名  }  // 使用  let mySearch: SearchFunc;  mySearch = function(source: string, subString: string) {  let result = source.search(subString);  return result > -1;  } |

函数类型是作为类型注解使用的，:SearchFunc，且没有函数名！

类类型：

|  |
| --- |
| interface ClockInterface {  currentTime: Date;  **setTime**(d: Date); // 方法，有方法名  }  class Clock implements ClockInterface {  currentTime: Date;  setTime(d: Date) {  this.currentTime = d;  }  constructor(h: number, m: number) { }  } |

强制一个类去符合某种契约。类类型接口描述了类的公共部分

需要其他类去实现 implements

接口里定义的是方法，有方法名的

### 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"); |

声明 class Greeter时，除了会创建一个名为 Greeter的类之外，同时也创建了一个名为 Greeter的**类型**（实例的类型），所以我们既可以将Greeter当做一个类来用，用new Greeter创建他的实例，也可以将Greeter当做一个类型来用（使用: Greeter表示参数的类型）；

声明Greeter类是创建的Greeter类型，只包含其中的**实例属性**和**实例方法，不含构造函数、静态属性或静态方法。**

|  |
| --- |
| class Point {  /\*\* 静态属性，坐标系原点 \*/  static origin = new Point(0, 0);  /\*\* 静态方法，计算与原点距离 \*/  static distanceToOrigin(p: Point) {  return Math.sqrt(p.x \* p.x + p.y \* p.y);  }  /\*\* 实例属性，x 轴的值 \*/  x: number;  /\*\* 实例属性，y 轴的值 \*/  y: number;  /\*\* 构造函数 \*/  constructor(x: number, y: number) {  this.x = x;  this.y = y;  }  /\*\* 实例方法，打印此点 \*/  printPoint() {  console.log(this.x, this.y);  }  }  interface PointInstanceType {  x: number;  y: number;  printPoint(): void;  }  let p1: Point;  let p2: PointInstanceType; |

最后的类型 Point 和类型 PointInstanceType 是等价的

#### 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。

|  |
| --- |
| class Animal {  // public name: string; // constructor 参数加了public, 此处可不用声明  public constructor (public name) {  // this.name = name; // 此处赋值也可以省略  }  } |

#### 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。

|  |
| --- |
| 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 |

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

#### 函数类型

两种方式：

|  |
| --- |
| function add(x: number, y: number): number {  return x + y;  }  let myAdd = function(x: number, y: number): number { return x + y; }; |

**书写完整的函数类型：**

|  |
| --- |
| let myAdd: **(x: number, y: number) => number** = function(x: number, y: number): number { return xxxxx + yyyyy; }; |

红色部分是完整的函数类型，分两部分：

=>前面：参数类型；

=>后面：返回值类型，返回值类型不能为空，无返回值时写void

函数类型中的参数名只是为了增强可读性，只要对应位置的类型匹配，名字可以不同。

**推断类型：**

如果你在赋值语句的一边指定了类型但是另一边没有类型的话，TypeScript编译器会自动识别出类型

|  |
| --- |
| // myAdd has the full function type  let myAdd = function(x: number, y: number): number { return x + y; };  // The parameters `x` and `y` have the type number  let myAdd: (baseValue: number, increment: number) => number =  function(x, y) { return x + y; }; |

#### 可选参数和默认参数

**可选参数**：?:

|  |
| --- |
| function buildName(firstName: string, lastName **?:** string) {} |

可选参数必须在必传参数后面

**默认参数**：

|  |
| --- |
| function buildName(firstName: string, lastName **=** "Smith") {} |

当不传参数，或传**undefined（不能是null）**时，会使用默认值。

默认参数可以不放在必选参数后面，若放在必选参数前，必须明确传入undefined值来获得默认值。

以上两种，可选参数与默认参数共享参数类型：

(firstName: string, lastName?: string) => string

#### 剩余参数

|  |
| --- |
| function buildName(firstName: string, **...restOfName**: string[]) {  return firstName + " " + **restOfName**.join(" ");  } |

#### this

Javascript里this的值在函数被调用的时候才会指定。

|  |
| --- |
| let deck = {  suits: ["hearts", "spades", "clubs", "diamonds"],  cards: Array(52),  createCardPicker: function() {  return function() {  let pickedCard = Math.floor(Math.random() \* 52);  let pickedSuit = Math.floor(pickedCard / 13);  return {suit: this.suits[pickedSuit], card: pickedCard % 13};  }  }  }  let cardPicker = deck.createCardPicker();  let pickedCard = cardPicker();  alert("card: " + pickedCard.card + " of " + pickedCard.suit); |

报错：createCardPicker返回的函数里的this被设置成了window而不是deck对象。

顶级的非方法式调用会将 this视为window。 （注意：在严格模式下， this为undefined而不是window）。

**箭头函数语法：**

**箭头函数能保存函数创建时的 this值，而不是调用时的值**：

|  |
| --- |
| let deck = {  suits: ["hearts", "spades", "clubs", "diamonds"],  cards: Array(52),  createCardPicker: function() {  return () => {  let pickedCard = Math.floor(Math.random() \* 52);  let pickedSuit = Math.floor(pickedCard / 13);  return {suit: this.suits[pickedSuit], card: pickedCard % 13};  }  }  }  let cardPicker = deck.createCardPicker();  let pickedCard = cardPicker();  alert("card: " + pickedCard.card + " of " + pickedCard.suit); |

**this参数:**

|  |
| --- |
| interface Deck {  name: string,  createCardPicker(this: Deck): () => string;  }  let deck: Deck = {  name: 'yang',  createCardPicker: function(**this: Deck**) {  return () => {  return this.name;  }  }  }  const fun = deck.createCardPicker();  **// error:**  const fun = deck.createCardPicker;  // fun调用的主体（callee）已经指定是Deck，顶级的非方法式调用会将 this视//为window，故报错  const res = fun(); |

提供一个显式的this参数，this参数是个假的参数，它出现在参数列表的**最前面**

现在TypeScript知道createCardPicker期望在某个Deck对象上调用。 也就是说 this是Deck类型的，而非any。

#### 重载

前两行为重载列表，函数调用时，编辑器会按顺序匹配，直到匹配上，使用匹配的类型进行检查，否则报错。

|  |
| --- |
| **function pickCard(x: {suit: string; card: number; }[]): number;**  **function pickCard(x: number): {suit: string; card: number; };**  function pickCard(x): any {  if (typeof x == "object") {  let pickedCard = Math.floor(Math.random() \* x.length);  return pickedCard;  }  else if (typeof x == "number") {  let pickedSuit = Math.floor(x / 13);  return { suit: suits[pickedSuit], card: x % 13 };  }  }  let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "hearts", card: 4 }];  let pickedCard1 = pickCard(myDeck);  let pickedCard2 = pickCard(15); |

### 泛型

#### 类型变量

定义：

|  |
| --- |
| function identity<T>(arg: T): T {  return arg;  } |

类型变量T：T帮助我们捕获用户传入的类型（比如：number）， 之后我们再次使用了 T当做返回值类型，从而保证参数和返回值的类型一致。

该identity函数即泛型，因为它可以适用于多个类型。

使用：

|  |
| --- |
| //传入所有的参数，包含类型参数  let output = identity<string>("myString");  //利用了类型推论--即编译器会根据传入的参数自动地帮助我们确定T的类型  let output = identity("myString"); |

使用类型推论，不用使用尖括号<>来明确传入参数类型，编辑器可以获取到参数类型，将其赋值给T

举例：

|  |
| --- |
| Array<number>  // 具体实现：  interface Array<T> {  [n: number]: T  } |

#### 泛型类型

泛型函数的类型与非泛型函数的类型没什么不同，只是有一个类型参数在最前面，像函数声明一样：

|  |
| --- |
| function identity**<T>(**arg: T): T {  return arg;  }  let myIdentity: **<T>(**arg: T) => T = identity;  let myIdentity: **<U>(**arg: U) => U = identity;  //对象字面量来定义泛型函数(最后一个T对应的是返回值的类型，类似于接口)  let myIdentity: **{<T>(arg: T): T}** = identity; |

**泛型接口**：

|  |
| --- |
| interface GenericIdentityFn {  **<T>(arg: T): T;**  }  function identity<T>(arg: T): T {  return arg;  }  let myIdentity: GenericIdentityFn = identity; |

确定泛型类型：

|  |
| --- |
| interface GenericIdentityFn**<T>** {  **(arg: T): T;**  }  function identity<T>(arg: T): T {  return arg;  }  let myIdentity: GenericIdentityFn<number> = identity; |

泛型参数的默认类型：

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;  } |

#### 泛型类

|  |
| --- |
| 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; };  // 亦可以使用string类型  let stringNumeric = new GenericNumber<**string**>();  stringNumeric.zeroValue = "";  stringNumeric.add = function(x, y) { return x + y; }; |

泛型类指的是**实例部分**的类型，所以类的**静态属性**不能使用这个泛型类型。

#### 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'. |

泛型 T 不一定包含属性 length，所以编译的时候报错了。

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。

|  |
| --- |
| 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'. |

**注意:泛型约束并不一定继承接口，也可以是其他类型；**

**extends 接口**

**extends string | number // 联合类型**

**extends 其他类型**

### 枚举

详见 技术学习笔记之 《typescript 枚举 - 简书》

#### 数字枚举

|  |
| --- |
| enum Direction {  Up = 1,  Down,  Left,  Right  } |

顺序自增

1. 通过枚举的属性来访问枚举成员

|  |
| --- |
| enum Response {  No = 0,  Yes = 1,  }  Console.log(Response.Yes) |

1. 枚举的名字来访问枚举类型

|  |
| --- |
| function respond(recipient: string, message: **Response**): void {  // ...  } |

#### 字符串枚举

|  |
| --- |
| enum Direction {  Up = "UP",  Down = "DOWN",  Left = "LEFT",  Right = "RIGHT",  } |

字符串枚举，必须用字符串字面量来初始化。

#### 联合枚举

枚举类型本身变成了每个枚举成员的联合，作为**联合类型**

|  |
| --- |
| enum E {  Foo,  Bar,  }  function f(x: **E**) {  if (x !== E.Foo || x !== E.Bar) {  // ~~~~~~~~~~~  // Error! Operator '!==' cannot be applied to types 'E.Foo' and 'E.Bar'.  }  } |

#### 枚举成员作为类型

枚举成员可以作为类型，例如，某些成员 只能是枚举成员的值

|  |
| --- |
| enum ShapeKind {  Circle,  Square,  }  interface Circle {  kind: **ShapeKind.Circle**;  radius: number;  } |

#### 运行时的枚举

枚举是在运行时真正存在的**对象；**

|  |
| --- |
| enum E {  X, Y, Z  }  function f(obj: { X: number }) {  return obj.X;  }  // Works, since 'E' has a property named 'X' which is a number.  f(E); |

#### 反向映射

**数字**枚举成员可以反向映射；

**字符串**枚举成员**不能**！

|  |
| --- |
| enum Enum {  A  }  let a = Enum.A;  let nameOfA = Enum[a]; // "A" |

#### const枚举

不同于常规的枚举，const枚举在编译阶段会被删除，即运行时不存在

|  |
| --- |
| const enum Enum {  A = 1,  B = 1 >> 2  }  console.log(Enum.A);  //编译后：  console.log(1 /\* A \*/); |

### 类型推论

### 声明合并

#### 函数合并

使用重载定义多个函数类型

#### 接口合并

|  |
| --- |
| interface Alarm {  price: number;  }  interface Alarm {  weight: number;  }  // 相当于  interface Alarm {  price: number;  weight: number;  } |

注意: 合并的属性的类型必须是唯一的：

|  |
| --- |
| interface Alarm {  price: number;  }  interface Alarm {  price: number; // 虽然重复了，但是类型都是 `number`，所以不会报错  weight: number;  }  interface Alarm {  price: number;  }  interface Alarm {  price: string; // 类型不一致，会报错  weight: number;  } |

接口中方法合并，和函数合并一样：

|  |
| --- |
| 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

与接口合并规则一致

### 模块

“内部模块”： 命名空间

“外部模块”： 模块

任何包含顶级import或者export的文件都被当成一个模块。相反地，如果一个文件不带有顶级的import或者export声明，那么它的内容被视为**全局可见的**

#### 导出

**导出声明**：**export** + 任何声明

|  |
| --- |
| export interface StringValidator {  isAcceptable(s: string): boolean;  } |

**导出语句**：可以对导出的部分重命名

|  |
| --- |
| interface StringValidator {  isAcceptable(s: string): boolean;  }  export { StringValidator as SV} |

**重新导出**：

重新导出功能并不会在当前模块导入那个模块或定义一个新的局部变量。

|  |
| --- |
| //ParseIntBasedZipCodeValidator.ts  // 导出原先的验证器但做了重命名  export {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidator"; |

包裹多个模块，把他们导出的内容联合在一起

|  |
| --- |
| **export \* from** "./StringValidator"; // exports interface StringValidator  export \* from "./LettersOnlyValidator"; // exports class LettersOnlyValidator  export \* from "./ZipCodeValidator"; // exports class ZipCodeValidator |

#### 导入

**导入部分内容：**放在{ } 里，导出局部内容

|  |
| --- |
| import { ZipCodeValidator } from "./ZipCodeValidator";  import { ZipCodeValidator as ZCV } from "./ZipCodeValidator"; |

**整体导入：**

|  |
| --- |
| import \* as validator from "./ZipCodeValidator";  let myValidator = new validator.ZipCodeValidator(); |

#### 默认导出

每个模块都可以有一个，且只能有一个default导出

|  |
| --- |
| // test2.ts  export default interface Test {  name: string  }  // test.ts  // T 即 test2.ts里导出的Test的别名；  import T from './test2';  const res: T = {name: 'yang'}; |

**类和函数**声明可以直接被标记为默认导出。 标记为默认导出的类和函数的名字是可以**省略**的：

|  |
| --- |
| export default interface {  name: string  } |

#### export = 和 import = require()

用export = 导出，必须使用import module = require()导入

|  |
| --- |
| export = ZipCodeValidator;  --------------------------------------  import zip = require("./ZipCodeValidator"); |

### 命名空间

命名空间是位于全局命名空间下的一个普通的带有名字的JavaScript对象。

|  |
| --- |
| namespace Validation{  **export** interface StringValidator {  isAcceptable(s: string): boolean;  }  }  let validators: { [s: string]: Validation.StringValidator; } = {}; |

想让接口或类在命名空间外也能被访问，需要用export才行；

#### 分离到多文件：

尽管是不同的文件，它们仍是同一个命名空间，并且在使用的时候就如同它们在一个文件中定义的一样。

|  |
| --- |
| // Validation.ts  namespace Validation {  export interface StringValidator {  isAcceptable(s: string): boolean;  }  }  ------------------------------------------------------------------------  // LettersOnlyValidator.ts  **/// <reference path="Validation.ts" />**  namespace Validation {  const lettersRegexp = /^[A-Za-z]+$/;  export class LettersOnlyValidator implements StringValidator {  isAcceptable(s: string) {  return lettersRegexp.test(s);  }  }  }  ------------------------------------------------------------------------  // ZipCodeValidator.ts  /// <reference path="Validation.ts" />  namespace Validation {  const numberRegexp = /^[0-9]+$/;  export class ZipCodeValidator implements StringValidator {  isAcceptable(s: string) {  return s.length === 5 && numberRegexp.test(s);  }  }  }  ------------------------------------------------------------------------  // Test.ts  /// <reference path="Validation.ts" />  /// <reference path="LettersOnlyValidator.ts" />  /// <reference path="ZipCodeValidator.ts" />  // Some samples to try  let strings = ["Hello", "98052", "101"];  // Validators to use  let validators: { [s: string]: Validation.StringValidator; } = {};  validators["ZIP code"] = new Validation.ZipCodeValidator();  validators["Letters only"] = new Validation.LettersOnlyValidator();  // Show whether each string passed each validator  for (let s of strings) {  for (let name in validators) {  console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" : "does not match" } ${ name }`);  }  } |

确保所有编译后的代码都被加载了:

把所有的输入文件编译为一个输出文件:

tsc **--outFile** sample.js Test.ts

#### 别名

**import q =** x.y.z  
注意：区别于import x = require('name')

|  |
| --- |
| namespace Shapes {  export namespace Polygons {  export class Triangle { }  export class Square { }  }  }  import polygons = Shapes.Polygons;  let sq = new polygons.Square(); // Same as "new Shapes.Polygons.Square()" |

### 声明文件

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

声明文件只能定义类型，切勿在声明文件里定义具体实现。

库的使用场景：

#### 全局变量

* 用<script>标签引入第三方库；
* 若通过npm install @types/xxx --save-dev ，不需要其他配置；
* 如果是将声明文件直接存放于当前项目中，则建议和其他源码一起放到 src 目录下

declare var : 声明全局变量

declare let : 同declare var ，拥有局部作用域

declare const : 全局常量，不可更改

declare function : 定义全局函数类型；声明文件里，支持函数重载

declare class： 定义全局类，不能有具体实现

declare enum：外部枚举类型，仅是定义类型，不是具体的值

declare namespace: 全局变量是个对象，包含很多子属性

|  |
| --- |
| declare namespace jQuery {  function ajax(url: string, settings?: any): void;  const version: number;  } |

interface: 声明一个全局的接口，前面不用加declare

type: 声明一个全局类型，前面不用加declare

声明合并：

假如jQuery既是一个对象，又是一个函数，他们会不冲突的合并起来：

|  |
| --- |
| // src/jQuery.d.ts  ​  declare function jQuery(selector: string): any;  declare namespace jQuery {  function ajax(url: string, settings?: any): void;  } |

|  |
| --- |
| // src/index.ts  ​  jQuery('#foo');  jQuery.ajax('/api/get\_something'); |

#### npm

一般来说，npm包的声明文件在这两个地方：

1. 与该 npm 包绑定在一起。

判断依据是包的 package.json 中有 types 字段，或者有一个 index.d.ts 声明文件。

types字段指定声明文件，若没有声明文件，则默认使用文件夹里的index.d.ts作为声明文件。

1. 发布到@types里

尝试npm install @types/foo --save-dev 就知道是否存在声明文件，这种模式一般是因为npm 包的维护者没有提供声明文件，所以只能由其他人将声明文件发布到 @types 里了。

1. 若以上两种都没有找到声明文件，需要手动创建声明文件：
   1. 放于node\_modules/@types/foo/index.d.ts，不需要配置，但node\_modules不稳定，只适合临时测试；
   2. 创建一个 types 目录，专门管理自定义的声明文件，放于types/foo/index.d.ts，需要配置下 tsconfig.json 中的 paths 和 baseUrl 字段：

|  |
| --- |
| {  "compilerOptions": {  "module": "commonjs",  "baseUrl": "./",  "paths": {  "\*": ["types/\*"]  }  }  } |

如此，通过 import 导入 foo 的时候，也会去 types 目录下寻找对应的模块的声明文件了。

npm包的声明文件：

不同于全局变量的声明文件，npm包的声明文件，使用declare只会在当前文件中声明一个局部变量。只有使用export导出，才能在使用方通过import导入后使用

export 导出变量

import：引入文件

export namespace 导出（含有子属性的）对象

export default 导出默认值

|  |
| --- |
| // types/foo/index.d.ts  export default function foo(): string;  // src/index.ts  import foo from 'foo';  foo(); |

注意：import foo from 'foo' 而不是 import { foo } from 'foo' 来导入这个默认值。

只有 **function、class 和 interface**可以直接默认导出，其他的变量需要先定义出来，再默认导出。

export =

导出模块有两种方式：

|  |
| --- |
| // 整体导出  module.exports = foo;  // 单个导出  exports.bar = bar; |

3种导入方式：

|  |
| --- |
| // 整体导入  const foo = require('foo');  // 单个导入  const bar = require('foo').bar; |

|  |
| --- |
| // 整体导入  import \* as foo from 'foo';  // 单个导入  import { bar } from 'foo'; |

**官方推荐：**

|  |
| --- |
| // 整体导入  import foo = require('foo');  // 单个导入  import bar = foo.bar; |

对于这种使用 commonjs 规范的库，如果要为它写类型声明文件的话，就必须使用到 **export =** 这种语法了

|  |
| --- |
| // types/foo/index.d.ts  ​  export = foo;  ​  declare function foo(): string;  declare namespace foo {  const bar: number;  } |

使用export = 之后就不能再单独导出export { bar }了，可以使用声明合并来将bar合并到foo里。

#### UMD库

既可以通过 <script> 标签引入，又可以通过 import 导入的库，称为 UMD 库。

**export as namespace**

将声明好的一个变量声明为全局变量，前提是现有npm包的声明文件，再基于它添加一条export as namespace 语句即可：

|  |
| --- |
| // types/foo/index.d.ts  ​  **export as namespace foo;**  export = foo;  ​  declare function foo(): string;  declare namespace foo {  const bar: number;  } |

### 执行ts文件

node -r ts-node/register

### 编译选项

tsc -w 监视整个目录下的ts文件，有改动即自动编译

### tsconfig.json

### 应用

#### 单例模式

|  |
| --- |
| class Axios {  static instance: Axios | null = null;  private constructor() {    }  static make (): Axios {  return Axios.instance = Axios.instance || new Axios();  }  } |

### 装饰器

#### 结构

能够被附加到类声明，方法， 访问符，属性或参数上；

|  |
| --- |
| // decorator定义  function decorator(target:Function) {    }  @decorator  class A{}  相当于调用了decorator(A) |

#### 组合装饰器

装饰器调用是从下往上的

|  |
| --- |
| function a(target: Function) {  console.log('a')  }  function b(target: Function) {  console.log('b')  }  @a  @b  class A{}  //输出：  b  a |

#### 类装饰器

类装饰器在类声明之前被声明（紧靠着类声明）

构造函数是唯一参数

|  |
| --- |
| function decorator(target:Function) {  target.prototype.say = () => {  console.log('saying');  }  }  @decorator  class A {  } |

#### 方法装饰器

方法装饰器声明在一个方法的声明之前（紧靠着方法声明）

方法装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1.对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。

2.成员的名字。

3.成员的属性描述符。

注意：

1.方法装饰器在初始化时就会调用一次

2.调用方法时，**先调用装饰器，再调用方法**

|  |
| --- |
| class Greeter {  greeting: string;  constructor(message: string) {  this.greeting = message;  }  @enumerable(false)  greet() {  return "Hello, " + this.greeting;  }  } |

定义@enumerable装饰器：

|  |
| --- |
| function enumerable(value: boolean) {  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {  descriptor.enumerable = value;  };  } |

这里的@enumerable(false)是一个**装饰器工厂**。 当装饰器 @enumerable(false)被调用时，它会修改属性描述符的enumerable属性。

#### 属性装饰器

属性装饰器声明在一个属性声明之前

属性装饰器表达式会在运行时当作函数被调用，传入下列2个参数：

1.对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。

2.成员的名字。

|  |
| --- |
| class Greeter {  greet(@required name: string) {  return "Hello " + name + ", " + this.greeting;  }  }  function required(target: Object, propertyKey: string, parameterIndex: number) {} |

#### 装饰器工厂

|  |
| --- |
| function decoratorFactory(name: string) {  return (target: Function) => {  target.prototype.say = () => {  console.log(name + ' saying');  }  }  }  @decoratorFactory('yang')  class A {} |

先调用decoratorFactory('yang')返回装饰器函数，再调用装饰器函数，传入构造函数A

