- 类型
- 静态类型
- 类型声明
- 编译
    - typescript 编译成 javascript - tsc 编译器
    - tsconfig.json 存储配置文件
    - ts-node 模块直接运行 typescript 代码
- any 类型 / Unknown 类型 / never 类型 / 联合类型
    - 类型推断
- 类型系统
    - 包装对象类型
    - 字面量类型
    - Object 类型和 object 类型
    - null / undefined类型
    - 值类型
    - const 关键字
    - 联合类型
    - 交叉类型
    - type命令-定义一个类型的别名
    - 块级类型声明
- 数组
    - 数组类型推断
    - 只读数组，const断言
    - 多维数组
- 元组
    - 只读元组
- symbol
    - unique symbol
- typescript函数类型
    - Function类型
    - 箭头函数
    - 可设置参数默认值
    - 参数解构
    - rest参数
    - readonly只读参数
    - void类型表示函数无返回值
    - never类型

- 模块
  - import type
- namespace
- 装饰器Decorator - @后接一个函数（或执行后可以得到一个函数）
- declare关键字
- 运算符
  - keyof - 单目运算符
  - in运算符
  - [ ]
  - extends...?: 条件运算符
  - 类型映射
  - 映射修饰符 - readonly
  - 键名重映射 - 允许修改键名
- 类型工具
- tsconfig.json
  - extends
  - files
  - include
  - reference
- tsc

类型

```
function addOne(n: number) {
  return n + 1;
}
//表明n只能用number数值，传入其他类型 的值会报错
addOne("hello"); //error
```

JavaScript 语言就没有这个功能，不会检查类型对不对。开发阶段很可能发现不了这个问题，代码也许就会原样发布，导致用户在使用时遇到错误。

作为比较，TypeScript 是在开发阶段报错，这样有利于提早发现错误，避免使用时报错。另一方面，函数定义里面加入类型，具有提示作用，可以告诉开发者这个函数怎么用。

静态类型

```
let x = 1;
x = "fpp"; //error，typesciript已经推断了类型，后面不允许修改

let x = { foo: 1 };
delete x.foo; //error
x.bar = 2; //error。对象的属性是静态的，不允许随意增删
```

希望一旦报错就停止编译，不生成编译产物 `--noEmitOnError`

```
tsc --noEmitOnError app.ts
```

类型声明

在 javascript 变量上添加了类型声明：

```
let foo: string;

//1.函数参数和返回值，也是这样来声明类型
function toString(num: number): string {
  return String(num);
}

//2.变量只有赋值后才能使用
let x: number;
console.log(x); //error
```

编译

typescript 编译成 javascript - tsc 编译器

```
//安装(全局)，也可以在项目中 将tsc安装为一个依赖模块
npm install -g typescript

//检查版本
tsc -v

//编译脚本
tsc app.ts

//一次编译多个,在当前目录生成3个脚本文件
```

```
tsc app.ts bpp.ts cpp.ts

//将多个typescript脚本编译成javascript文件
tsc file1.ts file2.ts --outFile

//将结果保存到其他目录
tsc app.ts --outDir dist

//指定编译后的javascript版本
tsc --target es2015 app.ts
```

tsconfig.json 存储配置文件

TypeScript 允许将 `tsc` 的编译参数，写在配置文件 `tsconfig.json` 。只要当前目录有这个文件，`tsc` 就会自动读取，所以运行时可以不写参数。

```bash
$ tsc file1.ts file2.ts --outFile dist/app.js
```

上面这个命令写成 `tsconfig.json` ，就是下面这样。

```json
{
  "files": ["file1.ts", "file2.ts"],
  "compilerOptions": {
    "outFile": "dist/app.js"
  }
}
```

有了这个配置文件，编译时直接调用 `tsc` 命令就可以了。

```bash
$ tsc
```

ts-node 模块直接运行 typescript 代码

```
//全局安装
npm install -g ts-node

//安装完可以直接运行ts脚本
ts-node script.ts
```

```
//
```

any 类型 / Unknown 类型 / never 类型 / 联合类型

关闭类型检查，尽量不用

类型推断

没有指定类型，则认为该变量类型为 any

```
function add(x, y) {
  return x + y;
}

add(1, [2, 3]);
//terrible! 尽量不用，一定要显式声明
```

unknown与 any 类似，区别：

- 不能直接调用 unknown 类型的变量的方法和属性

```
let v: unknown = 11;

let v1: boolean = v; //error
```

- 不能直接调用 unknown 类型变量的方法和属性

```
let v1: unknown = { foo: 123 };
v1.foo; // 报错

let v2: unknown = "hello";
v2.trim(); // 报错

let v3: unknown = (n = 0) => n + 1;
v3(); // 报错
```

never 类型 空类型

为什么 `never` 类型可以赋值给任意其他类型呢？这也跟集合论有关，空集是任何集合的子集。TypeScript 就相应规定，任何类型都包含了 `never` 类型。因此，`never` 类型是任何其他类型所共有的，TypeScript 把这种情况称为“底层类型”（bottom type）。

总之，TypeScript 有两个“顶层类型”（`any` 和 `unknown`），但是“底层类型”只有 `never` 唯一一个。

类型系统

js 值 8 种类型

```
//下面5个是原始类型Primitive value
boolean;
string;
number;
bigint;
symbol;

object;
//特殊值
undefined;
//复合类型
null;
//特殊值
```

包装对象类型

上面 5 种原始类型值，会产生包装对象(wrapper object)。

```
"hello".charAt(1);
//e
```

五种包装对象之中，symbol 类型和 bigint 类型无法直接获取它们的包装对象（即 `Symbol()` 和 `BigInt()` 不能作为构造函数使用），但是剩下三种可以。

- `Boolean()`
- `String()`
- `Number()`

字面量类型

由于包装对象的存在，导致每一个原始类型的值都有包装对象和字面量两种情况。

```javascript
"hello"; // 字面量
new String("hello"); // 包装对象
```

上面示例中，第一行是字面量，第二行是包装对象，它们都是字符串。

为了区分这两种情况，TypeScript 对五种原始类型分别提供了大写和小写两种类型。

- Boolean 和 boolean
- String 和 string
- Number 和 number
- BigInt 和 bigint
- Symbol 和 symbol

其中，大写类型同时包含包装对象和字面量两种情况，小写类型只包含字面量，不包含包装对象。

```typescript
const s1: String = "hello"; // 正确
const s2: String = new String("hello"); // 正确

const s3: string = "hello"; // 正确
const s4: string = new String("hello"); // 报错
```

上面示例中，`String` 类型可以赋值为字符串的字面量，也可以赋值为包装对象。但是，`string` 类型只能赋值为字面量，赋值为包装对象就会报错。

Object 类型和 object 类型

Object 类型

- 除了 undefined和 null类型，其他类型都可以赋值给 Object对象。
- 空对象 {}是 Object 类型的简写形式，所以使用 Object 时常常用空对象代替.

```
let obj: Object;
//等价于 let obj: {};

obj = {1};

obj = true;

obj = "hi";
obj = [1, 2];
obj = (a: number) => a + 1;
//correct

obj = undefined;//error
obj = null;//error
```

object 类型 代表 js 中的狭义对象，即可以用字面量表示的对象。 ：对象字面量：在代码中直接定义和初始化一个对象的方式。 只包含**对象、数组、函数**

注意，无论是大写的 `Object` 类型，还是小写的 `object` 类型，都只包含 JavaScript 内置对象原生的属性和方法，用户自定义的属性和方法都不存在于这两个类型之中。

```typescript
const o1: Object = { foo: 0 };
const o2: object = { foo: 0 };

o1.toString(); // 正确
o1.foo; // 报错

o2.toString(); // 正确
o2.foo; // 报错
```

上面示例中，`toString()` 是对象的原生方法，可以正确访问。`foo` 是自定义属性，访问就会报错。如何描述对象的自定义属性，详见《对象类型》一章。

null / undefined类型

值类型

ts规定：单个值也是一种类型

```
let x:"hello";
x = "hello";//co'r


x = "bye";//error
```

值类型可能会出现一些很奇怪的报错。

```typescript
const x: 5 = 4 + 1; // 报错
```

上面示例中，等号左侧的类型是数值 `5`，等号右侧 `4 + 1` 的类型，TypeScript 推测为 `number`。由于 `5` 是 `number` 的子类型，`number` 是 `5` 的父类型，父类型不能赋值给子类型，所以报错了（详见本章后文）。

但是，反过来是可以的，子类型可以赋值给父类型。

```typescript
let x: 5 = 5;
let y: number = 4 + 1;

x = y; // 报错
y = x; // 正确
```

上面示例中，变量 `x` 属于子类型，变量 `y` 属于父类型。`y` 不能赋值为子类型 `x`，但是反过来是可以的。

如果一定要让子类型可以赋值为父类型的值，就要用到类型断言（详见《类型断言》一章）。

```typescript
const x: 5 = (4 + 1) as 5; // 正确
```

const 关键字

TypeScript 推断类型时，遇到 `const` 命令声明的变量，如果代码里面没有注明类型，就会推断该变量是值类型。

```typescript
// x 的类型是 "https"
const x = "https";

// y 的类型是 string
const y: string = "https";
```

上面示例中，变量 `x` 是 `const` 命令声明的，TypeScript 就会推断它的类型是值 `https`，而不是 `string` 类型。

联合类型

多个类型组成一个新类型，符号"|"

```
let x: string | number;

x = 123; // 正确
x = "abc"; // 正确
```

如果一个变量有多种类型，需要使用分支处理每种类型 此时，处理所有可能的类型后，剩余的情况就属于 never 类型。

```
function fn(x:string | number){
    if(type x === "string"){

    }
    else if(type x === "number"){

    }
    else {
        x
    }
}
```

交叉类型

交叉类型A&B表示，任何一个类型必须同时属于A和B，才属于交叉类型A&B，即交叉类型同时满足A和B的特征。

```
let x: number & strin
```

上面示例中，变量 `x` 同时是数值和字符串，这当然是不可能的，所以 TypeScript 会认为 `x` 的类型实际是 `never` 。

交叉类型的主要用途是表示对象的合成。

typescript

```typescript
let obj: { foo: string } & { bar: string };

obj = {
  foo: "hello",
  bar: "world",
};
```

上面示例中，变量 `obj` 同时具有属性 `foo` 和属性 `bar` 。

交叉类型常常用来为对象类型添加新属性。

typescript

```typescript
type A = { foo: number };

type B = A & { bar: number };
```

上面示例中，类型 `B` 是一个交叉类型，用来在 `A` 的基础上增加了属性 `bar` 。

type命令-定义一个类型的别名

```
type Age = number;

let age: Age = 55;
```

块级类型声明

数组

- 所有成员类型必须相同。

```
//两种写法
let arr: number[] = [1, 2, 3];

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

typescript数组成员数量可以动态变化，所以typescript不会对数组边界进行检查，越界访问数组并不会报错

```
let arr: number[] = [1, 2, 3];
let foo = arr[3]; // 正确
```

数组类型推断

```
// 推断为 any[]
const arr = [];

arr.push(123);
arr; // 推断类型为 number[]

arr.push("abc");
arr; // 推断类型为 (string|number)[]
```

只读数组，const断言

在数组类型前面加上`readonly`关键字

```
const array: readonly number[] = [0, 1];

arr[1] = 2;//error
arr.push(3);//error
```

多维数组

TypeScript 使用 `T[][]` 的形式，表示二维数组，`T` 是最底层数组成员的类型。

```typescript
var multi: number[][] = [
  [1, 2, 3],
  [23, 24, 25],
];
```

上面示例中，变量 `multi` 的类型是 `number[][]`，表示它是一个二维数组，最底层的数组成员类型是 `number`。

元组

```
const s: [string, string, boolean] = ["a", "b", true];

//与数组区别：
成员类型写在方括号中
let a: [numbe] = [1];

//元组成员类型可添加后缀"?"，代表该成员可选，且问好只能用于元组的尾部成员
let a: [ number,number?] = [1];
```

```
type myTuple = [number, number, number?, string?];
```

上面示例中，元组 `myTuple` 的最后两个成员是可选的。也就是说，它的成员数量可能有两个、三个和四个。

扩展运算符...表示不限于成员数量的元组。

```
type NamedNums = [string, ...number[]];

const a: NamedNums = ["A", 1, 2];
const b: NamedNums = ["B", 1, 2, 3];
```

只读元组

```
type t = readonly[number, string];
//写法 1

type t = Readonly<[number, string]>;
//写法二· 是一个泛型，用到了工具类型Readonly<T>P
```

symbol

Symbol值通过Symbol()函数生成。

```
let x: symbol = Symbol();
let y: symbol = Symbol();

x === y; // false
```

unique symbol

`symbol` 类型包含所有的 Symbol 值，但是无法表示某一个具体的 Symbol 值。

比如，`5` 是一个具体的数值，就用 `5` 这个字面量来表示，这也是它的值类型。但是，Symbol 值不存在字面量，必须通过变量来引用，所以写不出只包含单个 Symbol 值的那种值类型。

为了解决这个问题，TypeScript 设计了 `symbol` 的一个子类型 `unique symbol`，它表示单个的、某个具体的 Symbol 值。

因为 `unique symbol` 表示单个值，所以这个类型的变量是不能修改值的，只能用 `const` 命令声明，不能用 `let` 声明。

```typescript
// 正确
const x: unique symbol = Symbol();

// 报错
let y: unique symbol = Symbol();
```

上面示例中，`let` 命令声明的变量，不能是 `unique symbol` 类型，会报错。

typescript函数类型

typescript的函数的类型声明，需要在声明函数时，给出参数的类型和返回值的类型（返回值类型通常可以不写，靠ts推断

```
function hello(a: string): void{
  console.log(a + "txt");
}
```

- 函数类型中的参数名可以和实际参数名不一致：

```
let f: (x:number) => number

f = function (y: number){
  return y;
}
```

- js函数在声明时可以有多余的餐宿，实际使用时之传入一部分参数。

```typescript
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // 正确
x = y; // 报错
```

上面示例中，函数 `x` 只有一个参数，函数 `y` 有两个参数， `x` 可以赋值给 `y` ，反过来就不行。

/
|

- 解决方法：

如果一个变量要套用另一个函数类型，有一个小技巧，就是使用 `typeof` 运算符。

```typescript
function add(x: number, y: number) {
  return x + y;
}

const myAdd: typeof add = function (x, y) {
  return x + y;
};
```

Function类型

不建议使用

箭头函数

可设置参数默认值

```
function createPoint(x: number = 0, y: number = 0): [number, number] {
  return [x, y];
}

createPoint(); // [0, 0]
```

参数解构

- **对象解构，允许直接从对象中提取属性值。

```
interface User {
  name: string;
  age: number;
  email?: string;
}

const user: User = {
  name: "Alice",
  age: 30,
  email: "alice@example.com"
};

//可以定义一个函数，使用对象解构访问name / age属性
function greet({name, age}: User){
  console.log("hello,${name},you are${age}years old")
}

greet(user);

//可以提供默认值
function greet({ name, age = 18, email = "n/a" }: User) {
  console.log(`Hello, ${name}! You are ${age} years old. Your email is ${email}.`);
}

greet({ name: "Bob" }); // 输出：Hello, Bob! You are 18 years old. Your email is
n/a.

//可选属性
function logUserDetails({ name, age, ...otherDetails }: User) {
  console.log(`Name: ${name}, Age: ${age}`);
  console.log("Other details:", otherDetails);
}

logUserDetails({
  name: "Diana",
  age: 35,
  email: "diana@example.com",
  phone: "123-456-7890"
});
// 输出:
// Name: Diana, Age: 35
// Other details: { email: "diana@example.com", phone: "123-456-7890" }
```

- **数组解构**

```
const numbers = [10, 20];

function sum([a, b]: [number, number]) {
  return a + b;
}
```

```
console.log(sum(numbers)); // 输出：30
```

rest参数

表示函数剩余的所有参数。。 可以是数组：剩余参数类型相同 可以是元组：剩余参数类型不同。 **元组需要声明每一个剩余参数的类型。如果元组里面的参数是可选的，则要使用可选参数"?"。**

```
function joinNumbers(...nums: number[]){

}
//数组

function joinTurples(...args:[boolean, numebr?]){}
```

rest参数可以与变量解构结合使用

```
function repeat(...[str, times]: [string, number]): string {
  return str.repeat(times);
}

// 等同于
function repeat(str: string, times: number): string {
  return str.repeat(times);
}
```

readonly只读参数

```
function arraySum(arr: readonly number[]) {
  // ...表示函数内部不能修改某个参数
  arr[0] = 0; // 报错
}
```

void类型表示函数无返回值

void类型允许返回undefined / null. 如果打开了strictNullChecks编译选项，那么 void 类型只允许返回undefined。如果返回null，就会报错。这是因为 JavaScript 规定，如

果函数没有返回值，就等同于返回undefined。

```
function f(): void {
  return 123; // 报错
}

function f(): void {
  return undefined; // 正确
}

function f(): void {
  return null; // 正确
}
```

需要特别注意的是，如果变量、对象方法、函数参数的类型是 void 类型的函数，那么并不代表不能赋值为有返回值的函数。恰恰相反，该变量、对象方法和函数参数可以接受返回任意值的函数，这时并不会报错。


```
type voidFunc = () => void;

const f: voidFunc = () => {
  return 123;
};
```

上面示例中，变量 `f` 的类型是 `voidFunc` ，是一个没有返回值的函数类型。但是实际上， `f` 的值是一个有返回值的函数（返回 `123` ），编译时不会报错。

这是因为，这时 TypeScript 认为，这里的 void 类型只是表示该函数的返回值没有利用价值，或者说不应该使用该函数的返回值。只要不用到这里的返回值，就不会报错。

这样设计是有现实意义的。举例来说，数组方法 `Array.prototype.forEach(fn)` 的参数 `fn` 是一个函数，而且这个函数应该没有返回值，即返回值类型是 `void` 。

never类型

用于表示肯定不会出现的值。 表示某个函数肯定不会有返回值，即函数不会正常执行结束。

两种情况：

- 抛出错误的函数：

```
function fail(msg: string): never {
  throw new Error(msg);
}
//函数fail()会抛错，不会正常退出，所以返回值类型是never

//注意，只有抛出错误，才是 never 类型。如果显式用return语句返回一个 Error 对象，返回值就不是 never 类型。
function fail(): Error {
  return new Error("Something failed");
}
```

- 无限执行函数

```
const sing = function (): never {
  while (true) {
    console.log("sing");
  }
};
```

注意，never类型不同于void类型。前者表示函数没有执行结束，不可能有返回值；后者表示函数正常执行结束，但是不返回值，或者说返回undefined。

局部类型

只在函数内部有效

高阶函数

一个函数的返回值还是一个函数，那个前一个函数称为高阶函数（higher-order function）。

如箭头函数返回的仍然是箭头函数；

```
(someValue: number) => (multiplier: number) => sommeValue * multiplier;
```

函数重载

根据参数类型不同，执行不同逻辑

```ts
function reverse(str: string): string;
function reverse(arr: any[]): any[];
function reverse(stringOrArray: string | any[]): string | any[] {
  if (typeof stringOrArray === "string")
    return stringOrArray.split("").reverse().join("");
  else return stringOrArray.slice().reverse();
}

reverse("abc");
//cba
reverse([1, 2, 3]);
//[3, 2, 1]
```

构造函数

```ts
class Animal {
  numLegs: number = 4;
}

type AnimalConstructor = new () => Animal;

function create(c: AnimalConstructor): Animal {
  return new c();
}

const a = create(Animal);
```

对象

可以使用方括号读取属性的类型

```ts
type User = {
  name: string;
  age: number;
};
type Name = User["name"]; // string
```

interface命令可以把对象类型作为一个接口：

```ts
// 写法一
type MyObj = {
  x: number;
```

```
  y: number;
};

const obj: MyObj = { x: 1, y: 1 };

// 写法二
interface MyObj {
  x: number;
  y: number;
}

const obj: MyObj = { x: 1, y: 1 };
```

可选属性

```
const obj: {
  x: number;
  y?: number;
} = { x: 1 };
```

所以，读取可选属性之前，必须检查一下是否为undefined。

```
const user: {
  firstName: string;
  lastName?: string;
} = { firstName: "Foo" };

if (user.lastName !== undefined) {
  console.log(`hello ${user.firstName} ${user.lastName}`);
}
```

只读属性 readonly / 只读断言 as const

```
type Point = {
  readonly x: number;
  readonly y: number;
};

const p: Point = { x: 0, y: 0 };

p.x = 100; // 报错
```

- **如果属性值是一个对象，readonly修饰符并不禁止修改该对象的属性，只是禁止完全替换掉该对象。**

```
interface Home {
  readonly resident: {
    name: string;
    age: number;
  };
}

const h: Home = {
  resident: {
    name: "Vicky",
    age: 42,
  },
};

h.resident.age = 32; // 正确
h.resident = {
  name: "Kate",
  age: 23,
}; // 报错
```

- **如果一个对象有两个引用，即两个变量对应同一个对象，其中一个变量是可写的，另一个变量是只读的，那么从可写变量修改属性，会影响到只读变量。**

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

interface ReadonlyPerson {
  readonly name: string;
  readonly age: number;
}

let w: Person = {
  name: "Vicky",
  age: 42,
};

let r: ReadonlyPerson = w;

w.age += 1;
r.age; // 43
```

如果希望属性值只是只读的方法： 1.声明时加入readonly关键词； 2.赋值时，在对象后面加上只读断言`as const`;

```
const myUser = {
  name: "Sabrina",
} as const;

myUser.name = "Cynthia"; // 报错
```

解构赋值

解构类型原则

只要对象 B 满足 对象 A 的结构特征，TypeScript 就认为对象 B 兼容对象 A 的类型，这称为“结构类型”原则（structual typing）。

空对象

空对象只能使用继承的属性，即继承自原型对象`Object.prototype`的属性：

```
const obj = {};

obj.toString();//cor
```

使用扩展运算符（...）合成一个新对象

```
const p1=  {};
const p2 = {x:3};
const p3 = {y:4, z: 5};

const p = {
  ...p1,
  ...p2,
  ...p3
}
```

interface接口

看作**对象的模板。**

```
interface Person {
  firstName: string;
  lastName?: string;//? 表示属性可选
  age: number;
  readonly sex:boolean;//readonly表示属性只读
}

//[]可以取出接口某个属性的类型
type A = Person[firstname];//string
```

实现接口 指定他作为对象的类型。

```
const p: Person = {
  firstName: "John",
  lastName: "Smith",
  age: 25,
};//变量p的类型就是接口Person
```

对象的属性索引 属性索引有string / number / symbol三种类型

数值索引必须兼容字符串索引的类型声明：

```
interface A {
  [prop: string]: number;
  [prop: number]: string; // 报错
}

interface B {
  [prop: string]: number;
  [prop: number]: number; // 正确
}
```

对象方法的3种表达方式

```
/
interface A{
  f(x: boolean): string;//1

  g(y:boolean) => string;//2

  h:{(z:boolean): string};//3
```

```
}
```

interface的函数重载

interface 里面的函数重载，不需要给出实现。但是，由于对象内部定义方法时，无法使用函数重载的语法，所以需要额外在对象外部给出函数方法的实现。

```typescript
interface A {
  f(): number;
  f(x: boolean): boolean;
  f(x: string, y: string): string;
}

function MyFunc(): number;
function MyFunc(x: boolean): boolean;
function MyFunc(x: string, y: string): string;
function MyFunc(x?: boolean | string, y?: string): number | boolean | string {
  if (x === undefined && y === undefined) return 1;
  if (typeof x === "boolean" && y === undefined) return true;
  if (typeof x === "string" && typeof y === "string") return "hello";
  throw new Error("wrong parameters");
}

const a: A = {
  f: MyFunc,
};
```

上面示例中，接口 `A` 的方法 `f()` 有函数重载，需要额外定义一个函数 `MyFunc()` 实现这个重载，然后部署接口 `A` 的对象 `a` 的属性 `f` 等于函数 `MyFunc()` 就可以了。

interface继承

- **interface继承interface** 如果子接口与父接口存在同名属性，那么子接口的属性会覆盖父接口的属性。注意，子接口与父接口的同名属性必须是类型兼容的，不能有冲突，否则会报错。

```
interface Shape {
  name: string;
```

```
}

interface Circle extends Shape {
  radius: number;
}

//子接口和父接口同名属性必须互相兼容
interface Foo {
  id: string;
}

interface Bar extends Foo {
  id: number; // 报错
}
```

- **interface继承type**

```
interface Foo {
  id: string;
}

interface Bar extends Foo {
  id: number; // 报错
}
```

- **interface继承class**

```
class A {
  x: string = "";

  y(): boolean {
    return true;
  }
}

interface B extends A {
  z: number;
}

//实现B接口的对象
const b: B = {
  x: "",
  y: function () {
    return true;
  },
  z: 123,
};
```

接口合并

多个同名接口会合并成一个接口。 作用：

```
interface A{
  height: number;
  width: number;
}
interface A{
  length: number;
}
```

同名接口合并时，如果同名方法有不同的类型声明，那么会发生函数重载。而且，后面的定义比前面的定义具有更高的优先级。

typescript

```typescript
interface Cloner {
  clone(animal: Animal): Animal;
}

interface Cloner {
  clone(animal: Sheep): Sheep;
}

interface Cloner {
  clone(animal: Dog): Dog;
  clone(animal: Cat): Cat;
}

// 等同于
interface Cloner {
  clone(animal: Dog): Dog;
  clone(animal: Cat): Cat;
  clone(animal: Sheep): Sheep;
  clone(animal: Animal): Animal;
}
```

上面示例中，`clone()` 方法有不同的类型声明，会发生函数重载。这时，越靠后的定义，优先级越高，排在函数重载的越前面。比如，`clone(animal: Animal)` 是最先出现的类型声明，就排在函数重载的最后，属于 `clone()` 函数最后匹配的类型。

这个规则有一个例外。同名方法之中，如果有一个参数是字面量类型，字面量类型有更高的优先级。

interface 和 type

class 类

方法的类型

```
class Point {
  x: number;
  y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }

  add(point: Point) {
    return new Point(this.x + point.x, this.y + point.y);
  }
}
```

存取器方法

accessor: getter / setter

- 如果某个属性只有get()方法，没有set()方法，则该属性自动成为只读属性：

```
class C {
  _name = "foo";

  get name() {
    return this._name;
  }
}

const c = new C();
c.name = "bar"; // 报错
```

- set()需兼容get()的参数类型
- 可访问性必须一致

类的interface接口

implements关键字 用于指定一个类实现了某个接口。

```
interface Animal {
  name: string;
  age: number;
  speak(): string;
}

interface Swimmer {
  swim(): void;
}

class Dog implements Animal, Swimmer {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  speak(): string {
    return `${this.name} says Woof!`;
  }

  swim(): void {
    console.log(`${this.name} is swimming.`);
  }
}

// 创建类的实例
const dog = new Dog('Buddy', 3);
console.log(dog.speak()); // 输出：Buddy says Woof!
dog.swim(); // 输出：Buddy is swimming.
```

实现多个接口（并不是好写法） 同时实现多个接口容易使代码难管理，替代方法：

- 类的继承

```
class Car implements MotorVehicals{}

class SecretCar extends Car implements Flyable,Swimmable{}
```

- 接口的继承

```
interface A {
  a: number;
}

interface B extends A {
  b: number;
}
```

类合并接口

如果一个类和一个接口同名，则接口会被合并进类。

Class类型

实例类型 Ts的类本身就是一种类型，代表该类的实例类型，而不是class的自身类型。

```
class Color{
  name: string;

  constructor(name: string){
    this.name = name;
  }
}

const green: Color = new Color("green");
```

对于引用实例对象的变量来说，既可以声明类型为 Class，也可以声明类型为 Interface，因为两者都代表实例对象的类型。

```typescript
interface MotorVehicle {}

class Car implements MotorVehicle {}

// 写法一
const c1: Car = new Car();
// 写法二
const c2: MotorVehicle = new Car();
```

上面示例中，变量的类型可以写成类 `Car` ，也可以写成接口 `MotorVehicle` 。它们的区别是，如果类 `Car` 有接口 `MotoVehicle` 没有的属性和方法，那么只有变量 `c1` 可以调用这些属性和方法。

类的自身类型 - 使用typeof运算符

类的自身类型就是一个构造函数，可以单独定义一个接口来表示。

```
class Point {
  x: number;
  y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
}

// 错误
function createPoint(PointClass: Point, x: number, y: number) {
  return new PointClass(x, y);
}

function createPoint(PointClass: typeof Point, x: number, y: number): Point {
  return new PointClass(x, y);
}
//正确
```

class 的结构类型原则

一个对象只要满足 Class 的实例结构，就跟该 Class 属于同一个类型。

类的继承

- 子类可以覆盖基类的同名方法

```
class B extends A {
  greet(name?: string) {
    if (name === undefined) {
      super.greet();
    } else {
      console.log(`Hello, ${name}`);
    }
  }
}
```

- 子类的同名方法不能和基类的类型定义冲突。

typescript中的静态成员 - static

泛型类

```
class Box<Type>{
  contents:Type;

  constructor(value:Type){
    this.contents = value;
  }
}

const b:Box<string> = new Box("hello");
```

静态成员不能使用泛型的类型参数

```typescript
class Box<Type> {
  static defaultContents: Type; // 报错
}
```

上面示例中，静态属性`defaultContents`的类型写成类型参数`Type`会报错。因为这意味着调用时必须给出类型参数（即写成`Box<string>.defaultContents`），并且类型参数发生变化，这个属性也会跟着变，这并不是好的做法。

抽象类

表示该类不能被实例化，只能作为其他类的模板。

```
abstract class A{
  id = 1;
}

class B extends A{};

const a = new A();
//error
const b = new B();//correct
```

this - 表示该方法当前所在的对象

泛型

泛型主要用在四个场合： 函数 / 接口 / 类 / 别名

- **多个类型参数** 如果有多个类型参数，则使用 T 后面的 U、V 等字母命名，各个参数之间使用逗号（ “,” ）分隔。

```
function map<T, U>(arr: T[], f: (arg: T) => U): U[] {
  return arr.map(f);
}

// 用法实例
map<string, number>(["1", "2", "3"], (n) => parseInt(n)); // 返回 [1, 2, 3]
```

函数的泛型

```
function id<T>(arg: T): T{
  return arg;
}

//对于变量形式定义的函数，有两种泛型写法
// 写法一
let myId: <T>(arg: T) => T = id;

// 写法二
let myId: { <T>(arg: T): T } = id;
```

接口的泛型

```
interface Box<Type>{
  contents:Type;
}

let box: Box<string>;

//第二种写法
interface Fn {
  <Type>(arg: Type): Type;
}

function id<Type>(arg: Type): Type {
  return arg;
```

```
}

let myId: Fn = id;
```

类的泛型写法

```
class Pair<K , V>{
  key : K;
  V: value;
}
```

泛型类描述的是类的实例，不包含静态属性和静态方法，因为这两者是定义在类的本身。所以，他们不能引用类型参数

```
class C<T> {
  static data: T; // 报错
  constructor(public value: T) {}
}
```

类型别名的泛型

```
type Nullable<T> = T | undefined | null;
```

数组的泛型

```
interface Array<Type> {
  length: number;

  pop(): Type | undefined;

  push(...items: Type[]): number;

  // ...
}
```

其他的 TypeScript 内部数据结构，比如 `Map` 、 `Set` 和 `Promise` ，其实也是泛型接口，完整的写法是 `Map<K, V>` 、 `Set<T>` 和 `Promise<T>` 。

ts默认提供`ReadonlyArray<T>`接口，表示只读数组。

ENUM

```typescript
enum Color {
  Red, // 0
  Green, // 1
  Blue, // 2
}
```

上面示例声明了一个 Enum 结构 `Color` ，里面包含三个成员 `Red` 、 `Green` 和 `Blue` 。第一个成员的值默认为整数 `0` ，第二个为 `1` ，第二个为 `2` ，以此类推。

使用时，调用 Enum 的某个成员，与调用对象属性的写法一样，可以使用点运算符，也可以使用方括号运算符。

```typescript
let c = Color.Green; // 1
// 等同于
let c = Color["Green"]; // 1
```

Enum 结构本身也是一种类型。比如，上例的变量 `c` 等于 `1` ，它的类型可以是 Color，也可以是 `number` 。

```typescript
let c: Color = Color.Green; // 正确
let c: number = Color.Green; // 正确
```

keyof可以取出ENUM结构的所有成员名，作为联合类型返回。

```
enum MyEnum {
  A = "a",
  B = "b",
}
```

```typescript
// 'A'|'B'
type Foo = keyof typeof MyEnum;
```

类型断言

模块

任何包含 import 或 export 语句的文件，就是一个模块（module）。相应地，如果文件不包含 export 语句，就是一个全局的脚本文件。

模块本身就是一个作用域，不属于全局作用域。模块内部的变量、函数、类只在内部可见，对于模块外部是不可见的。暴露给外部的接口，必须用 export 命令声明；如果其他文件要使用模块的接口，必须用 import 命令来输入。

如果一个文件不包含 export 语句，但是希望把它当作一个模块（即内部变量对外不可见），可以在脚本头部添加一行语句。

```typescript
export {};
```

上面这行语句不产生任何实际作用，但会让当前文件被当作模块处理，所有它的代码都变成了内部代码。

相较ES6特别之处：**允许输出和输入类型**

```typescript
export type Bool = true | false;
```

上面示例中，当前脚本输出一个类型别名 `Bool`。这行语句把类型定义和接口输出写在一行，也可以写成两行。

```typescript
type Bool = true | false;

export { Bool };
```

假定上面的模块文件为 `a.ts`，另一个文件 `b.ts` 就可以使用 import 语句，输入这个类型。

```typescript
import { Bool } from "./a";

let foo: Bool = true;
```

上面示例中，import 语句加载的是一个类型。注意，加载文件写成 `./a`，没有写脚本文件的后缀名。TypeScript 允许加载模块时，省略模块文件的后缀名，它会自动定位，将 `./a` 定位到 `./a.ts`。

编译时，可以两个脚本同时编译。

```bash
$ tsc a.ts b.ts
```

上面命令会将 `a.ts` 和 `b.ts` 分别编译成 `a.js` 和 `b.js`。

也可以只编译 `b.ts`，因为它是入口脚本，tsc 会自动编译它依赖的所有脚本。

```bash
$ tsc b.ts
```

上面命令发现 `b.ts` 依赖 `a.js`，就会自动寻找 `a.ts`，也将其同时编译，因此编译产物还是 `a.js` 和 `b.js` 两个文件。

如果想将 `a.ts` 和 `b.ts` 编译成一个文件，可以使用 `--outFile` 参数。

```typescript
$ tsc --outFile result.js b.ts
```

上面示例将 `a.ts` 和 `b.ts` 合并编译为 `result.js`。

import type

namespace

将相关代码组织在一起。

它出现在 ES 模块诞生之前，作为 TypeScript 自己的模块格式而发明的。但是，自从有了 ES 模块，官方已经不推荐使用 namespace 了。

装饰器Decorator - @后接一个函数（或执行后可以得到一个函数）

用来在定义时修改类的行为

```
function simpleDecorator(target: any, context: any) {
  console.log("hi, this is " + target);
  return target;
}

@simpleDecorator
class A {} // "hi, this is class A {}"
```

declare关键字

告诉编译器，某个类型存在，可以在当前文件中使用。

```
x = 123;//error

declare let x: number;
x = 123;
```

可以描述下面类型： 变量（const、let、var 命令声明） type 或者 interface 命令声明的类型 class enum 函数（function） 模块（module） 命名空间（namespace）

运算符

keyof - 单目运算符

接受一个对象类型作为参数，返回该对象的所有键名组成的联合类型。 用于精确表达对象的属性类型。

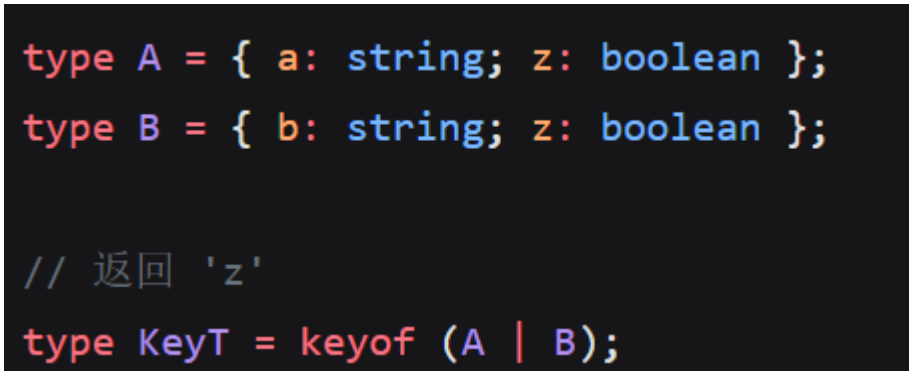
```
interface T {
  0: boolean;
  a: string;
  b(): void;
}

type KeyT = keyof T; // 0 | 'a' | 'b'
//由于 JavaScript 对象的键名只有三种类型，所以对于任意对象的键名的联合类型就是
string|number|symbol。


//应用于数组或元组类型
type arr = keyof["a" , "b" , "c"];
// 返回 number | "0" | "1" | "2"
// | "length" | "pop" | "push" | ···
```

由于 JavaScript 对象的键名只有三种类型，所以对于任意对象的键名的联合类型就是 string|number|symbol。

- 对联合类型返回成员共有键名



- 对交叉类型返回所有键名

```typescript
type A = { a: string; x: boolean };
type B = { b: string; y: number };

// 返回 'a' | 'x' | 'b' | 'y'
type KeyT = keyof (A & B);

// 相当于
keyof (A & B) ≡ keyof A | keyof B
```

in运算符

确定对象是否包含某个属性名

```
const obj = {a: 23};
if("a" in obj ) console.log("found a");
```

TypeScript 语言的类型运算中，`in` 运算符有不同的用法，用来取出（遍历）联合类型的每一个成员类型。

```typescript
type U = "a" | "b" | "c";

type Foo = {
  [Prop in U]: number;
};
// 等同于
type Foo = {
  a: number;
  b: number;
  c: number;
};
```

上面示例中，`[Prop in U]` 表示依次取出联合类型 `U` 的每一个成员。

上一小节的例子也提到，`[Prop in keyof Obj]` 表示取出对象 `Obj` 的每一个键名。

[ ]

取出对象的键值类型

```
type Person = {
  age: number;
  name: string;
  alive: boolean;
};

// Age 的类型是 number
type Age = Person["age"];
```

extends...?: 条件运算符

类型映射

mapping 将一种类型按照映射规则，转换成另一种类型。

```typescript
type A = {
  foo: number;
  bar: number;
};

type B = {
  foo: string;
  bar: string;
};
```

上面示例中，这两个类型的属性结构是一样的，但是属性的类型不一样。如果属性数量多的话，逐个写起来就很麻烦。

使用类型映射，就可以从类型 `A` 得到类型 `B`。

```typescript
type A = {
  foo: number;
  bar: number;
};

type B = {
  [prop in keyof A]: string;
};
```

上面示例中，类型 `B` 采用了属性名索引的写法，`[prop in keyof A]` 表示依次得到类型 `A` 的所有属性名，然后将每个属性的类型改成 `string`。

为了增加代码复用性，可以把常用的映射写成泛型：

```
type ToBoolean<Type> = {
  [Prop in keyof Type]: boolean;
}
```

映射修饰符 - readonly

键名重映射 - 允许修改键名

```
type A = {
  foo: number;
  bar: number;
};

type B = {
  [p in keyof A as `${p}ID`]: number;
};

// 等同于
type B = {
  fooID: number;
  barID: number;
};
```

类型工具

tsconfig.json

放在项目的根目录 **可以不必手写，使用`tsc --init`参数自动生成tsconfig.json

反过来说，如果一个目录中有tsconfig.json，ts就认为这事项目的根目录。

```
tsc --init
//自动生成tsconfig.json

tsc -p ./dir
//指定tsconfig.json的位置


//tsconfig.json的文件格式示例
{
  "compilerOptions": {
    "outDir": "./built",
    "allowJs": true,
    "target": "es5"
  },
  "include": ["./src/**/*"]
}
```

- include：指定哪些文件需要编译。
- allowJs：指定源目录的 JavaScript 文件是否原样拷贝到编译后的目录。
- outDir：指定编译产物存放的目录。

- target：指定编译产物的 JS 版本。

extends

tsconfig.json可以继承另一个tsconfig.json文件的配置。如果一个项目有多个配置，可以把共同的配置写成tsconfig.base.json，其他的配置文件继承该文件，这样便于维护和修改。

extends属性用来指定所要继承的配置文件。它可以是本地文件。

```
{
  "extends": "../tsconfig.base.json"
}
```

files

指定编译的文件列表

```
{
  "files": ["a.ts" , "b.ts"];
}
```

如果文件较多，建议使用include 和 exclude 。

include

指定所要编译的文件列表，即支持逐一列出文件和支持通配符。

```
{
  "include" : ["src/**/*", "tests/**/*"]
}
```

支持三种通配符：

- ?:指代单个字符
- *：指代任意字符，不含路径分隔符

- **：指定任意目录层级

reference

tsc

tsc 默认使用当前目录下的配置文件tsconfig.json，但也可以接受独立的命令行参数。命令行参数会覆盖tsconfig.json，比如命令行指定了所要编译的文件，那么 tsc 就会忽略tsconfig.json的files属性。

```
# 使用 tsconfig.json 的配置
tsc

//只编译 index.ts
tsc index.ts

//编译 src 目录的所有 .ts 文件
//tsc src/*.ts

# 指定编译配置文件
tsc --project tsconfig.production.json

# 只生成类型声明文件，不编译出 JS 文件
tsc index.js --declaration --emitDeclarationOnly

# 多个 TS 文件编译成单个 JS 文件
tsc app.ts util.ts --target esnext --outfile index.js
```