# 概述

* TypeScript：静态类型检查器
* JavaScript 的类型化超集
* 擦除类型：TypeScript 的编译器检查完你的代码，它就会删除类型以生成生成的“编译”代码。

# 静态类型检查

TypeScript在代码运行之前找到这些错误，这就是静态类型检查器所做的。静态类型系统描述了当我们运行程序时我们的值的形状和行为。像 TypeScript 这样的类型检查器使用这些信息并告诉我们什么时候事情可能会出错。

[TypeScript 静态类型检查](TypeScript%20静态类型检查.png)

其目的是捕捉我们程序中的合法错误，但是TypeScript 捕获了很多合法的错误。

[TypeScript 静态类型检查 合法错误捕捉](TypeScript%20静态类型检查%20合法错误捕捉.png)

# TypeScript 编译器

tsc编译器TypeScript的类型检查器，通过 npm 获取它：

npm install -g typescript

# 显式类型

可以编辑代码以告诉TypeScript这person是一个string，并且date应该是一个Date对象。

function greet(person: string, date: Date) {

return `Hello ${person}, today is ${date.toDateString()}!`;

}

greet("Maddison", Date());

Argument of type 'string' is not assignable to parameter of type 'Date'.

Date()在 JavaScript中调用会返回一个string！

但是并不总是必须编写显式类型注释。在许多情况下，TypeScript 甚至可以为推断（或“找出”）类型，即使我们省略它们。

# 擦除类型

编译上述函数greet以tsc输出 JavaScript 时，

function greet(person, date) {

return "Hello ".concat(person, ", today is ").concat(date.toDateString(), "!");

}

greet("Maddison", new Date());

* person和date参数不再有类型注释。
* “模板字符串”——那个使用反引号（`字符）的字符串——被转换为带有连接（+）的纯字符串

## 降级

上面的另一个区别是我们的模板字符串是从`Hello ${person}, today is ${date.toDateString()}!`;到"Hello " + person + ", today is " + date.toDateString() + "!";。

模板字符串是 ECMAScript 版本中的一项功能，称为 ECMAScript 2015。TypeScript 能够将代码从较新版本的 ECMAScript 重写为较旧的版本，例如 ECMAScript 3 或 ECMAScript 5（又名 ES3 和 ES5）。从 ECMAScript 的新版本或“更高”版本向下移动到旧版本或“更低”版本的过程有时称为downleveling。

默认情况下，TypeScript 以 ES3 为目标，这是一个非常旧的 ECMAScript 版本。target通过使用该选项，我们可以选择更新一点的东西。运行--target es2015更改 TypeScript 以针对 ECMAScript 2015，这意味着代码应该能够在任何支持 ECMAScript 2015 的地方运行。所以运行tsc --target es2015 hello.ts会给我们以下输出：

function greet(person, date) {

console.log(`Hello ${person}, today is ${date.toDateString()}!`);

}

greet("Maddison", new Date());

## 严格

TypeScript 有几个可以打开或关闭的类型检查严格标志，除非另有说明，否则我们所有的示例都将在启用所有这些标志的情况下编写。strictCLI 或"strict": truea中的标志tsconfig.json同时打开它们，但我们可以单独选择退出它们。你应该知道的最大的两个是noImplicitAny和strictNullChecks。

* noImplicitAny：打开该noImplicitAny标志将对任何类型隐式推断为 的变量发出错误any。
* strictNullChecks：该strictNullChecks标志使处理null和undefined更明确。

# 基础类型

JavaScript 具有三个非常常用的原始类型：string、number和boolean。每个在 TypeScript 中都有对应的类型。typeof如您所料，如果您对这些类型的值使用 JavaScript 运算符，您会看到这些名称。

## 布尔值

最基本的数据类型就是简单的true/false值，在JavaScript和TypeScript里叫做boolean。

let isDone: boolean = false;

## 数字

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

let decLiteral: number = 6;

let hexLiteral: number = 0xf00d;

let binaryLiteral: number = 0b1010;

let octalLiteral: number = 0o744;

## 字符串

JavaScript程序的另一项基本操作是处理网页或服务器端的文本数据。 像其它语言里一样，我们使用 string表示文本数据类型。 和JavaScript一样，可以使用双引号（ "）或单引号（'）表示字符串。

let name: string = "bob";

name = "smith";

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

let name: string = `Gene`;

let age: number = 37;

let sentence: string = `Hello, my name is ${ name }.I'll be ${ age + 1 } years old next month.`;

这与下面定义sentence的方式效果相同：

let sentence: string = "Hello, my name is " + name + ".\n\n" + "I'll be " + (age + 1) + " years old next month.";

## 空值

JavaScript 没有空值（Void）的概念，在 TypeScript 中，可以用 void 表示没有任何返回值的函数。声明一个 void 类型的变量没有什么用，因为你只能将它赋值为 undefined 和 null（只在 --strictNullChecks 未指定时）：

let unusable: void = undefined;

## Null 和 Undefined

在 TypeScript 中，可以使用 null 和 undefined 来定义这两个原始数据类型：

let u: undefined = undefined;

let n:null = null;

与 void 的区别是，undefined 和 null 是所有类型的子类型。也就是说 undefined 类型的变量，可以赋值给 number 类型的变量：

// 这样不会报错

let num: number = undefined;

// 这样也不会报错

let u: undefined;

let num: number = u;

而 void 类型的变量不能赋值给 number 类型的变量：

let u: void;

let num: number = u;

// Type 'void' is not assignable to type 'number'.

## Never类型

never 是完全没有返回值的类型，只有一种情况会如此：代码阻断。那么当执行 throw new Error 、 return process.exit(1)、while(true){} 时都满足此条件：

function error(message: string): never {

throw new Error(message);

}

如果函数的返回值类型是 never 意味的此函数必须不能被顺利完整执行，而发生中断行为。

never 是所有类型的子类型，因此可以理解为：所有的函数的返回值都包含 never 类型：

function fun(s: string): number {

if (s == 'a') return 1;

if (s == 'b') return 2;

throw new Error;

}

// 等同于 fun(s: string): number | never

只要不是单独申明为 never 类型，都会被显性的展示起来，因此下面的声明中也会忽略 never：

type Temp = 1 | 'a' | never

// 得到

// type Temp = 1 | 'a'

Type ne = never

// 得到

// type ne = never

# 变量声明

let和const是 JavaScript 中变量声明的两个相对较新的概念。某些方面let类似于var，但允许用户避免用户在 JavaScript 中遇到的一些常见“陷阱”。const是 的增强，let因为它可以防止重新分配给变量。由于 TypeScript 是 JavaScript 的扩展，该语言自然支持let和const。

## var声明

在JavaScript 中声明变量总是使用var关键字来完成的。

var a = 10;

还可以在函数内部声明一个变量并访问这些相同的变量：

function f() {

var a = 10;

return function g() {

var b = a + 1;

return b;

};

}

var g = f();

g(); // returns '11'

## 范围规则

function f(shouldInitialize: boolean) {

if (shouldInitialize) {

var x = 10;

}

return x;

}

f(true); // returns '10'

f(false); // returns 'undefined'

这是因为声明可以在其包含的函数、模块、命名空间或全局范围内的任何地方访问，参数也是函数范围的。

for (var i = 0; i < 10; i++) {

setTimeout(function () {

console.log(i);

}, 100 \* i);

}

setTimeout将在几毫秒后运行一个函数，但只有在for循环停止执行之后；到for循环停止执行时， 的i值为10。所以每次调用给定的函数时，它都会打印出来10。一个常见的解决方法是使用 IIFE - 一个立即调用的函数表达式 -i在每次迭代时捕获：

for (var i = 0; i < 10; i++) {

// capture the current state of 'i'

// by invoking a function with its current value

(function (i) {

setTimeout(function () {

console.log(i);

}, 100 \* i);

})(i);

}

## let声明

var存在一些问题，这正是let引入语句的原因。除了使用的关键字之外，let语句的编写方式与var语句相同。

let hello = "Hello!";

当使用声明变量时let，它使用一些所谓的词法作用域或块作用域。与声明的变量var的作用域泄漏到其包含函数不同，块作用域变量在其最近的包含块或循环之外不可见for。

function f(input: boolean) {

let a = 100;

if (input) {

// Still okay to reference 'a'

let b = a + 1;

return b;

}

// Error: 'b' doesn't exist here

return b;

}

块作用域变量的另一个特性是它们在实际声明之前不能被读取或写入。虽然这些变量在其整个范围内都是“存在的”，但直到它们声明之前的所有点都是它们临时死区的一部分。这只是在声明之前说您无法访问它们的一种复杂方式let， TypeScript 会提示这一点。

a++; // illegal to use 'a' before it's declared;

let a;

## 重新声明

对于var声明，我们提到声明变量的次数无关紧要。

function f(x) {

var x;

var x;

if (true) {

var x;

}

}

在上面的例子中，所有的声明x实际上都指向同一个 x，这是完全有效的。这通常最终成为错误的来源。值得庆幸的是，let声明并不那么宽容。

let x = 10;

let x = 20; // error: can't re-declare 'x' in the same scope

在嵌套的范围内引入新名称的行为称为遮蔽。它有点像一把双刃剑，因为它可以在意外阴影的情况下自行引入某些错误，同时还可以防止某些错误。例如，假设我们使用变量编写了早期的sumMatrix函数。let

function sumMatrix(matrix: number[][]) {

let sum = 0;

for (let i = 0; i < matrix.length; i++) {

var currentRow = matrix[i];

for (let i = 0; i < currentRow.length; i++) {

sum += currentRow[i];

}

}

return sum;

}

这个版本的循环实际上将正确地执行求和，因为内循环的i阴影i来自外循环。为了编写更清晰的代码，通常应该避免阴影。虽然在某些情况下可能适合利用它，但您应该使用最佳判断。

let当声明为循环的一部分时，声明具有截然不同的行为。这些声明不仅仅是为循环本身引入一个新环境，而是在每次迭代时创建一个新的范围。由于这就是我们对 IIFE 所做的事情，因此我们可以将旧setTimeout示例更改为仅使用let声明。

for (let i = 0; i < 10; i++) {

setTimeout(function () {

console.log(i);

}, 100 \* i);

}

## const声明

const声明是声明变量的另一种方式。

const numLivesForCat = 9;

它们就像let声明，但正如它们的名字所暗示的，它们的值一旦被绑定就不能改变。换句话说，它们具有与 相同的范围规则let，但您不能重新分配给它们。这不应与它们引用的值是不可变的想法相混淆。

const numLivesForCat = 9;

const kitty = {

name: "Aurora",

numLives: numLivesForCat,

};

// Error

kitty = {

name: "Danielle",

numLives: numLivesForCat,

};

// all "okay"

kitty.name = "Rory";

kitty.name = "Kitty";

kitty.name = "Cat";

kitty.numLives--;

## let对比const

鉴于我们有两种类型的声明具有相似的作用域语义，很自然地会发现自己询问使用哪一种。像大多数广泛的问题一样，答案是：视情况而定。应用最小特权原则，除了您打算修改的声明之外的所有声明都应该使用const。基本原理是，如果不需要写入变量，则在同一代码库上工作的其他人不应该自动写入该对象，并且需要考虑他们是否真的需要重新分配给该变量。const在推理数据流时，使用还可以使代码更可预测。

# 任意值

任意值（Any）用来表示允许赋值为任意类型。

## 任意值类型

如果是一个普通类型，在赋值过程中改变类型是不被允许的：

// index.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.

let myFavoriteNumber: string = 'seven';

myFavoriteNumber = 7;

但如果是 any 类型，则允许被赋值为任意类型。

let myFavoriteNumber: any = 'seven';

myFavoriteNumber = 7;

## 任意值的属性和方法

在任意值上访问任何属性都是允许的：

let anyThing: any = 'hello';

console.log(anyThing.myName);

console.log(anyThing.myName.firstName);

也允许调用任何方法：

let anyThing: any = 'Tom';

anyThing.setName('Jerry');

anyThing.setName('Jerry').sayHello();

anyThing.myName.setFirstName('Cat');

可以认为，声明一个变量为任意值之后，对它的任何操作，返回的内容的类型都是任意值。

## 未声明类型的变量

变量如果在声明的时候，未指定其类型，那么它会被识别为任意值类型：

let something;

something = 'seven';

something = 7;

something.setName('Tom');

等价于

let something: any;

something = 'seven';

something = 7;

something.setName('Tom');

## unknown

TypeScript 3.0 引入了一个顶级的 unknown 类型。 对照于 any，unknown 是类型安全的。 任何值都可以赋给 unknown，但是当没有类型断言或基于控制流的类型细化时 unknown 不可以赋值给其它类型，除了它自己和 any 外。 同样地，在 unknown 没有被断言或细化到一个确切类型之前，是不允许在其上进行任何操作的。

* 任何类型都可以赋值给 unknown 类型，但是 unknown 类型除了赋值给 any 和它本身之外，不可以赋值给其他类型
* unknown 在没有明确它类型的情况下，不可以使用上面的任何属性和方法
* unknown 也可以被使用与双重断言
* 所以当你不知道一个变量的类型的时候，尽量使用 unknown 来代替 any
* {}这个类型包含来所有的值，除了 null 和 undefined
* Object 类型包含了所有的非原始值类型，包含 null，array，objects 但是不包含 undefined。

# Symbols

自ECMAScript 2015起，symbol成为了一种新的原生类型，就像number和string一样。symbol类型的值是通过Symbol构造函数创建的。

let sym1 = Symbol();

let sym2 = Symbol("key"); // 可选的字符串key

Symbols是不可改变且唯一的。

let sym2 = Symbol("key");

let sym3 = Symbol("key");

sym2 === sym3; // false, symbols是唯一的

像字符串一样，symbols也可以被用做对象属性的键。

let sym = Symbol();

let obj = {

[sym]: "value"

};

console.log(obj[sym]); // "value"

Symbols也可以与计算出的属性名声明相结合来声明对象的属性和类成员。

const getClassNameSymbol = Symbol();

class C {

[getClassNameSymbol](){

return "C";

}

}

let c = new C();

let className = c[getClassNameSymbol](); // "C"

除了用户定义的symbols，还有一些已经众所周知的内置symbols。 内置symbols用来表示语言内部的行为。以下为这些symbols的列表：

* Symbol.hasInstance：方法，会被instanceof运算符调用。构造器对象用来识别一个对象是否是其实例。
* Symbol.isConcatSpreadable：布尔值，表示当在一个对象上调用Array.prototype.concat时，这个对象的数组元素是否可展开。
* Symbol.iterator：方法，被for-of语句调用。返回对象的默认迭代器。
* Symbol.match：方法，被String.prototype.match调用。正则表达式用来匹配字符串。
* Symbol.replace：方法，被String.prototype.replace调用。正则表达式用来替换字符串中匹配的子串。
* Symbol.search：方法，被String.prototype.search调用。正则表达式返回被匹配部分在字符串中的索引。
* Symbol.species：函数值，为一个构造函数。用来创建派生对象。
* Symbol.split：方法，被String.prototype.split调用。正则表达式来用分割字符串。
* Symbol.toPrimitive：方法，被ToPrimitive抽象操作调用。把对象转换为相应的原始值。
* Symbol.toStringTag：方法，被内置方法Object.prototype.toString调用。返回创建对象时默认的字符串描述。
* Symbol.unscopables：对象，它自己拥有的属性会被with作用域排除在外。

# 类型兼容性

TypeScript里的类型兼容性是基于结构子类型的。 结构类型是一种只使用其成员来描述类型的方式。 它正好与名义（nominal）类型形成对比。（译者注：在基于名义类型的类型系统中，数据类型的兼容性或等价性是通过明确的声明和/或类型的名称来决定的。这与结构性类型系统不同，它是基于类型的组成结构，且不要求明确地声明。） 看下面的例子：

interface Named {

name: string;

}

class Person {

name: string;

}

let p: Named;

// OK, because of structural typing

p = new Person();

在使用基于名义类型的语言，比如C#或Java中，这段代码会报错，因为Person类没有明确说明其实现了Named接口。TypeScript的结构性子类型是根据JavaScript代码的典型写法来设计的。 因为JavaScript里广泛地使用匿名对象，例如函数表达式和对象字面量，所以使用结构类型系统来描述这些类型比使用名义类型系统更好。

## 可靠性的注意事项

TypeScript的类型系统允许某些在编译阶段无法确认其安全性的操作。当一个类型系统具此属性时，被当做是“不可靠”的。TypeScript允许这种不可靠行为的发生是经过仔细考虑的。通过这篇文章，我们会解释什么时候会发生这种情况和其有利的一面。TypeScript结构化类型系统的基本规则是，如果x要兼容y，那么y至少具有与x相同的属性。比如：

interface Named {

name: string;

}

let x: Named;

// y's inferred type is { name: string; location: string; }

let y = { name: 'Alice', location: 'Seattle' };

x = y;

这里要检查y是否能赋值给x，编译器检查x中的每个属性，看是否能在y中也找到对应属性。 在这个例子中，y必须包含名字是name的string类型成员。y满足条件，因此赋值正确。检查函数参数时使用相同的规则：

function greet(n: Named) {

console.log('Hello, ' + n.name);

}

greet(y); // OK

## 比较函数

相对来讲，在比较原始类型和对象类型的时候是比较容易理解的，问题是如何判断两个函数是兼容的。 下面我们从两个简单的函数入手，它们仅是参数列表略有不同：

let x = (a: number) => 0;

let y = (b: number, s: string) => 0;

y = x; // OK

x = y; // Error

要查看x是否能赋值给y，首先看它们的参数列表。 x的每个参数必须能在y里找到对应类型的参数。 注意的是参数的名字相同与否无所谓，只看它们的类型。 这里，x的每个参数在y中都能找到对应的参数，所以允许赋值。第二个赋值错误，因为y有个必需的第二个参数，但是x并没有，所以不允许赋值。

忽略额外的参数在JavaScript里是很常见的。 例如，Array#forEach给回调函数传3个参数：数组元素，索引和整个数组。 尽管如此，传入一个只使用第一个参数的回调函数也是很有用的：

let items = [1, 2, 3];

// Don't force these extra arguments

items.forEach((item, index, array) => console.log(item));

// Should be OK!

items.forEach((item) => console.log(item));

let x = () => ({name: 'Alice'});

let y = () => ({name: 'Alice', location: 'Seattle'});

x = y; // OK

y = x; // Error, because x() lacks a location property

类型系统强制源函数的返回值类型必须是目标函数返回值类型的子类型。

### 函数参数双向协变

当比较函数参数类型时，只有当源函数参数能够赋值给目标函数或者反过来时才能赋值成功。 这是不稳定的，因为调用者可能传入了一个具有更精确类型信息的函数，但是调用这个传入的函数的时候却使用了不是那么精确的类型信息。 实际上，这极少会发生错误，并且能够实现很多JavaScript里的常见模式。例如：

enum EventType { Mouse, Keyboard }

interface Event { timestamp: number; }

interface MouseEvent extends Event { x: number; y: number }

interface KeyEvent extends Event { keyCode: number }

function listenEvent(eventType: EventType, handler: (n: Event) => void) {

/\* ... \*/

}

// Unsound, but useful and common

listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x + ',' + e.y));

// Undesirable alternatives in presence of soundness

listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x + ',' + (<MouseEvent>e).y));

listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) => console.log(e.x + ',' + e.y)));

// Still disallowed (clear error). Type safety enforced for wholly incompatible types

listenEvent(EventType.Mouse, (e: number) => console.log(e));

### 可选参数及剩余参数

比较函数兼容性的时候，可选参数与必须参数是可互换的。 源类型上有额外的可选参数不是错误，目标类型的可选参数在源类型里没有对应的参数也不是错误。当一个函数有剩余参数时，它被当做无限个可选参数。这对于类型系统来说是不稳定的，但从运行时的角度来看，可选参数一般来说是不强制的，因为对于大多数函数来说相当于传递了一些undefinded。有一个好的例子，常见的函数接收一个回调函数并用对于程序员来说是可预知的参数但对类型系统来说是不确定的参数来调用：

function invokeLater(args: any[], callback: (...args: any[]) => void) {

/\* ... Invoke callback with 'args' ... \*/

}

// Unsound - invokeLater "might" provide any number of arguments

invokeLater([1, 2], (x, y) => console.log(x + ', ' + y));

// Confusing (x and y are actually required) and undiscoverable

invokeLater([1, 2], (x?, y?) => console.log(x + ', ' + y));

### 函数重载

对于有重载的函数，源函数的每个重载都要在目标函数上找到对应的函数签名。 这确保了目标函数可以在所有源函数可调用的地方调用。

## 高级主题

目前为止，我们使用了“兼容性”，它在语言规范里没有定义。 在TypeScript里，有两种兼容性：子类型和赋值。 它们的不同点在于，赋值扩展了子类型兼容性，增加了一些规则，允许和any来回赋值，以及enum和对应数字值之间的来回赋值。

Any、unknown、object、void、undefined、null、和never可分配性。下表总结了一些抽象类型之间的可分配性。行表示每个可分配的内容，列表示可分配给他们的内容。“ ✓ ”表示仅在strictNullChecks关闭时兼容的组合。

[引用表](引用表.png)

* 一切都可以分配给它自己。
* any并且unknown在可分配给他们的方面是相同的，不同之处在于unknown除了 . 之外不可分配给任何东西any。
* unknown并且never是彼此的倒数。一切都可分配给unknown，never可分配给一切。没有任何东西可以分配给never，unknown也不能分配给任何东西（除了any）。
* void不能分配给任何东西或从任何东西分配，但以下例外：any、unknown、never、undefined和null
* 何时strictNullChecks关闭，null类似于undefined:never可分配给大多数类型，大多数类型不可分配给它们。它们可以相互分配。
* WhenstrictNullChecks处于打开状态，null并且undefined行为更像void: 不可分配给或从任何东西，除了any, unknown,never和void(undefined总是可分配给void)。

# 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。

## 什么是类型推论

以下代码虽然没有指定类型，但是会在编译的时候报错：

// index.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.

let myFavoriteNumber = 'seven';

myFavoriteNumber = 7;

事实上，它等价于：

// index.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.

let myFavoriteNumber: string = 'seven';

myFavoriteNumber = 7;

TypeScript 会在没有明确的指定类型的时候推测出一个类型，这就是类型推论。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 myFavoriteNumber;

myFavoriteNumber = 'seven';

myFavoriteNumber = 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。

## 例子

let myFavoriteNumber: string | number;

myFavoriteNumber = 'seven';

myFavoriteNumber = 7;

let myFavoriteNumber: string | number;

myFavoriteNumber = true;

// index.ts(2,1): error TS2322: Type 'boolean' is not assignable to type 'string | number'.

// Type 'boolean' is not assignable to type 'number'.

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

## 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：

// index.ts(2,22): error TS2339: Property 'length' does not exist on type 'string | number'.

// Property 'length' does not exist on type 'number'.

function getLength(something: string | number): number {

return something.length;

}

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

// index.ts(5,30): error TS2339: Property 'length' does not exist on type 'number'.

上例中，第二行的 myFavoriteNumber 被推断成了 string，访问它的 length 属性不会报错。而第四行的 myFavoriteNumber 被推断成了 number，访问它的 length 属性时就报错了。

# 对象的类型——接口

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

## 什么是接口

在面向对象语言中，接口（Interfaces）是一个很重要的概念，它是对行为的抽象，而具体如何行动需要由类（classes）去实现（implement）。TypeScript 中的接口是一个非常灵活的概念，除了可用于对类的一部分行为进行抽象以外，也常用于对「对象的形状（Shape）」进行描述。以下的例子：

interface Person {

name: string;

age: number;

}

let tom: Person = {

name: 'Tom',

age: 25

};

上面的例子中，我们定义了一个接口 Person，接着定义了一个变量 tom，它的类型是 Person。这样，我们就约束了 tom 的形状必须和接口 Person 一致。定义的变量比接口少了一些属性是不允许的，多一些属性也是不允许的。可见，赋值的时候，变量的形状必须和接口的形状保持一致。

## 可选属性

有时我们希望不要完全匹配一个形状，那么可以用可选属性：

interface Person {

name: string;

age?: number;

}

let tom: Person = {

name: 'Tom'

};

可选属性的含义是该属性可以不存在。这时仍然不允许添加未定义的属性。

## 任意属性

有时候我们希望一个接口允许有任意的属性，可以使用如下方式：

interface Person {

name: string;

age?: number;

[propName: string]: any;

}

let tom: Person = {

name: 'Tom',

gender: 'male'

};

使用 [propName: string] 定义了任意属性取 string 类型的值。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。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'

};

## 只读属性

有时候我们希望对象中的一些字段只能在创建的时候被赋值，那么可以用 readonly 定义只读属性：

// index.ts(14,5): error TS2540: Cannot assign to 'id' because it is a constant or a read-only property.

interface Person {

readonly id: number;

name: string;

age?: number;

[propName: string]: any;

}

let tom: Person = {

id: 89757,

name: 'Tom',

gender: 'male'

};

tom.id = 9527;

上例中，使用 readonly 定义的属性 id 初始化后，又被赋值了，所以报错了。注意，只读的约束存在于第一次给对象赋值的时候，而不是第一次给只读属性赋值的时候。只读属性必须初始化赋值。

# 数组

在TypeScrip中，数组类型有多种定义方式，比较灵活。

## 数组定义

最简单的方法是使用「类型 + 方括号」来表示数组：

let fibonacci: number[] = [1, 1, 2, 3, 5];

数组的项中不允许出现其他的类型。数组的一些方法的参数也会根据数组在定义时约定的类型进行限制。

## 数组泛型

我们也可以使用数组泛型（Array Generic）Array<elemType> 来表示数组：

let fibonacci: Array<number> = [1, 1, 2, 3, 5];

## 接口表示数组

接口也可以用来描述数组：

interface NumberArray {

[index: number]: number;

}

let fibonacci: NumberArray = [1, 1, 2, 3, 5];

NumberArray 表示：只要索引的类型是数字时，那么值的类型必须是数字。虽然接口也可以用来描述数组，但是我们一般不会这么做，因为这种方式比前两种方式复杂多了。不过有一种情况例外，那就是它常用来表示类数组。

## 类数组

类数组（Array-like Object）不是数组类型，不能用普通的数组的方式来描述，而应该用接口：

function sum() {

let args: {

[index: number]: number;

length: number;

callee: Function;

} = arguments;

}

在这个例子中，我们除了约束当索引的类型是数字时，值的类型必须是数字之外，也约束了它还有 length 和 callee 两个属性。

事实上常用的类数组都有自己的接口定义，如 IArguments, NodeList, HTMLCollection 等：

function sum() {

let args: IArguments = arguments;

}

其中 IArguments 是 TypeScript 中定义好了的类型，它实际上就是：

interface IArguments {

[index: number]: any;

length: number;

callee: Function;

}

## any 在数组中的应用

一个比较常见的做法是，用 any 表示数组中允许出现任意类型：

let list: any[] = ['xcatliu', 25, { website: 'http://xcatliu.com' }];

# 函数的类型

## 函数声明

在 JavaScript 中，有两种常见的定义函数的方式——函数声明（Function Declaration）和函数表达式（Function Expression）：

// 函数声明（Function Declaration）

function sum(x, y) {

return x + y;

}

// 函数表达式（Function Expression）

let mySum = function (x, y) {

return x + y;

};

一个函数有输入和输出，要在 TypeScript 中对其进行约束，需要把输入和输出都考虑到，其中函数声明的类型定义较简单：

function sum(x: number, y: number): number {

return x + y;

}

注意，输入多余的（或者少于要求的）参数，是不被允许的。

## 函数表达式

如果要我们现在写一个对函数表达式（Function Expression）的定义，可能会写成这样：

let mySum = function (x: number, y: number): number {

return x + y;

};

这是可以通过编译的，不过事实上，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 mySum，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 mySum 添加类型，则应该是这样：

let mySum: (x: number, y: number) => number = function (x: number, y: number): number {

return x + y;

};

注意不要混淆了 TypeScript 中的 => 和 ES6 中的 =>。在 TypeScript 的类型定义中，=> 用来表示函数的定义，左边是输入类型，需要用括号括起来，右边是输出类型。在 ES6 中，=> 叫做箭头函数，应用十分广泛

## 接口定义函数

我们也可以使用接口的方式来定义一个函数需要符合的形状：

interface SearchFunc {

(source: string, subString: string): boolean;

}

let mySearch: SearchFunc;

mySearch = function(source: string, subString: string) {

return source.search(subString) !== -1;

}

采用函数表达式|接口定义函数的方式时，对等号左侧进行类型限制，可以保证以后对函数名赋值时保证参数个数、参数类型、返回值类型不变。

## 可选参数

前面提到，输入多余的（或者少于要求的）参数，是不允许的。那么如何定义可选的参数呢？与接口中的可选属性类似，我们用 ? 表示可选的参数：

function buildName(firstName: string, lastName?: string) {

if (lastName) {

return firstName + ' ' + lastName;

} else {

return firstName;

}

}

let tomcat = buildName('Tom', 'Cat');

let tom = buildName('Tom');

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。

## 参数默认值

在ES6中，我们允许给函数的参数添加默认值，TypeScript会将添加了默认值的参数识别为可选参数：

function buildName(firstName: string, lastName: string = 'Cat') {

return firstName + ' ' + lastName;

}

let tomcat = buildName('Tom', 'Cat');

let tom = buildName('Tom');

此时就不受「可选参数必须接在必需参数后面」的限制了。

## 剩余参数

ES6 中，可以使用 ...rest 的方式获取函数中的剩余参数（rest 参数）：

function push(array, ...items) {

items.forEach(function(item) {

array.push(item);

});

}

let a: any[] = [];

push(a, 1, 2, 3);

事实上，items 是一个数组。所以我们可以用数组的类型来定义它。注意，rest 参数只能是最后一个参数。

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

items.forEach(function(item) {

array.push(item);

});

}

## 重载

重载允许一个函数接受不同数量或类型的参数时，作出不同的处理。比如，我们需要实现一个函数 reverse，输入数字 123 的时候，输出反转的数字 321，输入字符串 'hello' 的时候，输出反转的字符串 'olleh'。

利用联合类型，我们可以这么实现：

function reverse(x: number | string): number | string | void {

if (typeof x === 'number') {

return Number(x.toString().split('').reverse().join(''));

} else if (typeof x === 'string') {

return x.split('').reverse().join('');

}

}

然而这样有一个缺点，就是不能够精确的表达，输入为数字的时候，输出也应该为数字，输入为字符串的时候，输出也应该为字符串。这时，我们可以使用重载定义多个 reverse 的函数类型：

function reverse(x: number): number;

function reverse(x: string): string;

function reverse(x: number | string): number | string | void {

if (typeof x === 'number') {

return Number(x.toString().split('').reverse().join(''));

} else if (typeof x === 'string') {

return x.split('').reverse().join('');

}

}

注意，TypeScript 会优先从最前面的函数定义开始匹配，所以多个函数定义如果有包含关系，需要优先把精确的定义写在前面。

# 类型断言

类型断言（Type Assertion）可以用来手动指定一个值的类型。

## 语法

值 as 类型

或

<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。

## 用途

类型断言的常见用途有以下几种：

* **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。此时可以使用类型断言，将 animal 断言成 Fish：

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。

* **将一个父类断言为更加具体的子类**

当类之间有继承关系时，类型断言也是很常见的：

class ApiError extends Error {

code: number = 0;

}

class HttpError extends Error {

statusCode: number = 200;

}

function isApiError(error: Error) {

if (typeof (error as ApiError).code === 'number') {

return true;

}

return false;

}

上面的例子中，我们声明了函数 isApiError，它用来判断传入的参数是不是 ApiError 类型，为了实现这样一个函数，它的参数的类型肯定得是比较抽象的父类 Error，这样的话这个函数就能接受 Error 或它的子类作为参数了。

* **将任何一个类型断言为 any**

理想情况下，TypeScript 的类型系统运转良好，每个值的类型都具体而精确。当我们引用一个在此类型上不存在的属性或方法时，就会报错：

// index.ts:2:5 - error TS2339: Property 'length' does not exist on type 'number'.

const foo: number = 1;

foo.length = 1;

上面的例子中，数字类型的变量 foo 上是没有 length 属性的，故 TypeScript 给出了相应的错误提示。这种错误提示显然是非常有用的。但有的时候，我们非常确定这段代码不会出错，比如下面这个例子，此时我们可以使用 as any 临时将 window 断言为 any 类型：

(window as any).foo = 1;

在 any 类型的变量上，访问任何属性都是允许的。需要注意的是，将一个变量断言为 any 可以说是解决 TypeScript 中类型问题的最后一个手段。它极有可能掩盖了真正的类型错误，所以如果不是非常确定，就不要使用 as any。

* **将 any 断言为一个具体的类型**

遇到 any 类型的变量时，我们可以选择无视它，任由它滋生更多的 any。我们也可以选择改进它，通过类型断言及时的把 any 断言为精确的类型，亡羊补牢，使我们的代码向着高可维护性的目标发展。举例来说，历史遗留的代码中有个 getCacheData，它的返回值是 any：

function getCacheData(key: string): any {

return (window as any).cache[key];

}

那么我们在使用它时，最好能够将调用了它之后的返回值断言成一个精确的类型，这样就方便了后续的操作：

function getCacheData(key: string): any {

return (window as any).cache[key];

}

interface Cat {

name: string;

run(): void;

}

const tom = getCacheData('tom') as Cat;

tom.run();

上面的例子中，我们调用完 getCacheData 之后，立即将它断言为 Cat 类型。这样的话明确了 tom 的类型，后续对 tom 的访问时就有了代码补全，提高了代码的可维护性。

## 类型断言的限制

从上面的例子中，我们可以总结出：

* 联合类型可以被断言为其中一个类型
* 父类可以被断言为子类
* 任何类型都可以被断言为 any
* any 可以被断言为任何类型

并不是任何一个类型都可以被断言为任何另一个类型。具体来说，若 A 兼容 B，那么 A 能够被断言为 B，B 也能被断言为 A。以下的例子：

nterface Animal {

name: string;

}

interface Cat {

name: string;

run(): void;

}

let tom: Cat = {

name: 'Tom',

run: () => { console.log('run') }

};

let animal: Animal = tom;

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。在上面的例子中，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

这样的设计其实也很容易就能理解：

* 允许 animal as Cat 是因为「父类可以被断言为子类」。
* 允许 cat as Animal 是因为既然子类拥有父类的属性和方法，那么被断言为父类，获取父类的属性、调用父类的方法，就不会有任何问题，故「子类可以被断言为父类」。

总之，若 A 兼容 B，那么 A 能够被断言为 B，B 也能被断言为 A。同理，若 B 兼容 A，那么 A 能够被断言为 B，B 也能被断言为 A。所以这也可以换一种说法：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，这也是为了在类型断言时的安全考虑，毕竟毫无根据的断言是非常危险的。

综上所述：

* 联合类型可以被断言为其中一个类型
* 父类可以被断言为子类
* 任何类型都可以被断言为 any
* any 可以被断言为任何类型
* 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
* 其实前四种情况都是最后一个的特例。

## 双重断言

既然：

* 任何类型都可以被断言为 any
* any 可以被断言为任何类型

那么我们是不是可以使用双重断言 as any as Foo 来将任何一个类型断言为任何另一个类型呢？

interface Cat {

run(): void;

}

interface Fish {

swim(): void;

}

function testCat(cat: Cat) {

return (cat as any as Fish);

}

在上面的例子中，若直接使用 cat as Fish 肯定会报错，因为 Cat 和 Fish 互相都不兼容。但是若使用双重断言，则可以打破「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」的限制，将任何一个类型断言为任何另一个类型。若你使用了这种双重断言，那么十有八九是非常错误的，它很可能会导致运行时错误。

## 类型断言 vs 类型转换

类型断言只会影响 TypeScript 编译时的类型，类型断言语句在编译结果中会被删除：

function toBoolean(something: any): boolean {

return something as boolean;

}

toBoolean(1);

// 返回值为 1

在上面的例子中，将 something 断言为 boolean 虽然可以通过编译，但是并没有什么用，代码在编译后会变成：

function toBoolean(something) {

return something;

}

toBoolean(1);

// 返回值为 1

所以类型断言不是类型转换，它不会真的影响到变量的类型。若要进行类型转换，需要直接调用类型转换的方法：

function toBoolean(something: any): boolean {

return Boolean(something);

}

toBoolean(1);

// 返回值为 true

## 类型断言 vs 类型声明

在这个例子中：

function getCacheData(key: string): any {

return (window as any).cache[key];

}

interface Cat {

name: string;

run(): void;

}

const tom = getCacheData('tom') as Cat;

tom.run();

我们使用 as Cat 将 any 类型断言为了 Cat 类型。但实际上还有其他方式可以解决这个问题：

function getCacheData(key: string): any {

return (window as any).cache[key];

}

interface Cat {

name: string;

run(): void;

}

const tom: Cat = getCacheData('tom');

tom.run();

上面的例子中，我们通过类型声明的方式，将 tom 声明为 Cat，然后再将 any 类型的 getCacheData('tom') 赋值给 Cat 类型的 tom。这和类型断言是非常相似的，而且产生的结果也几乎是一样的——tom 在接下来的代码中都变成了 Cat 类型。它们的区别，可以通过这个例子来理解：

interface Animal {

name: string;

}

interface Cat {

name: string;

run(): void;

}

const animal: Animal = {

name: 'tom'

};

let tom = animal as Cat;

在上面的例子中，由于 Animal 兼容 Cat，故可以将 animal 断言为 Cat 赋值给 tom。

但是若直接声明 tom 为 Cat 类型：

interface Animal {

name: string;

}

interface Cat {

name: string;

run(): void;

}

const animal: Animal = {

name: 'tom'

};

let tom: Cat = animal;

// index.ts:12:5 - error TS2741: Property 'run' is missing in type 'Animal' but required in type 'Cat'.

则会报错，不允许将 animal 赋值为 Cat 类型的 tom。这很容易理解，Animal 可以看作是 Cat 的父类，当然不能将父类的实例赋值给类型为子类的变量。深入的讲，它们的核心区别就在于：

* animal 断言为 Cat，只需要满足 Animal 兼容 Cat 或 Cat 兼容 Animal 即可
* animal 赋值给 tom，需要满足 Cat 兼容 Animal 才行，但是 Cat 并不兼容 Animal。

而在前一个例子中，由于 getCacheData('tom') 是 any 类型，any 兼容 Cat，Cat 也兼容 any，故

const tom = getCacheData('tom') as Cat;

等价于

const tom: Cat = getCacheData('tom');

知道了它们的核心区别，就知道了类型声明是比类型断言更加严格的。所以为了增加代码的质量，我们最好优先使用类型声明，这也比类型断言的 as 语法更加优雅。

## 类型断言 vs 泛型

以上哪个例子，我们还有第三种方式可以解决这个问题，那就是泛型：

function getCacheData<T>(key: string): T {

return (window as any).cache[key];

}

interface Cat {

name: string;

run(): void;

}

const tom = getCacheData<Cat>('tom');

tom.run();

通过给 getCacheData 函数添加了一个泛型 <T>，我们可以更加规范的实现对 getCacheData 返回值的约束，这也同时去除掉了代码中的 any，是最优的一个解决方案。

# 实用程序类型

TypeScript 提供了几种实用程序类型来促进常见的类型转换。这些实用程序在全局范围内可用。

## Partial<Type>

构造一个所有属性都Type设置为可选的类型。此实用程序将返回一个表示给定类型的所有子集的类型。

interface Todo {

title: string;

description: string;

}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {

return { ...todo, ...fieldsToUpdate };

}

const todo1 = {

title: "organize desk",

description: "clear clutter",

};

const todo2 = updateTodo(todo1, {

description: "throw out trash",

});

## Required<Type>

Type构造一个由set to required的所有属性组成的类型。的反面Partial。

interface Props {

a?: number;

b?: string;

}

const obj: Props = { a: 5 };

const obj2: Required<Props> = { a: 5 };

Property 'b' is missing in type '{ a: number; }' but required in type 'Required<Props>'.

## Readonly<Type>

构造一个所有属性都Type设置为的类型readonly，这意味着构造类型的属性不能重新分配。

interface Todo {

title: string;

}

const todo: Readonly<Todo> = {

title: "Delete inactive users",

};

todo.title = "Hello";

Cannot assign to 'title' because it is a read-only property.

## Record<Keys, Type>

构造一个对象类型，其属性键为Keys，其属性值为Type。此实用程序可用于将一种类型的属性映射到另一种类型。

interface CatInfo {

age: number;

breed: string;

}

type CatName = "miffy" | "boris" | "mordred";

const cats: Record<CatName, CatInfo> = {

miffy: { age: 10, breed: "Persian" },

boris: { age: 5, breed: "Maine Coon" },

mordred: { age: 16, breed: "British Shorthair" },

};

cats.boris;

## Pick<Type, Keys>

通过从 中选择一组属性Keys（字符串文字或字符串文字的并集）来构造类型Type

interface Todo {

title: string;

description: string;

completed: boolean;

}

type TodoPreview = Pick<Todo, "title" | "completed">;

const todo: TodoPreview = {

title: "Clean room",

completed: false,

};

todo;

## Omit<Type, Keys>

Type通过从中选择所有属性然后删除Keys（字符串文字或字符串文字的联合）来构造类型。

interface Todo {

title: string;

description: string;

completed: boolean;

createdAt: number;

}

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {

title: "Clean room",

completed: false,

createdAt: 1615544252770,

};

todo;

const todo: TodoPreview

type TodoInfo = Omit<Todo, "completed" | "createdAt">;

const todoInfo: TodoInfo = {

title: "Pick up kids",

description: "Kindergarten closes at 5pm",

};

todoInfo;

## Exclude<UnionType, ExcludedMembers>

UnionType通过从所有可分配给 的联合成员中排除来构造一个类型ExcludedMembers。

type T0 = Exclude<"a" | "b" | "c", "a">;

type T0 = "b" | "c"

type T1 = Exclude<"a" | "b" | "c", "a" | "b">;

type T1 = "c"

type T2 = Exclude<string | number | (() => void), Function>;

type T2 = string | number

## Extract<Type, Union>

Type通过从可分配给 的所有联合成员中提取来构造一个类型Union。

type T0 = Extract<"a" | "b" | "c", "a" | "f">;

type T0 = "a"

type T1 = Extract<string | number | (() => void), Function>;

type T1 = () => void

## NonNullable<Type>

null通过 exclude和undefinedfrom构造一个类型Type。

type T0 = NonNullable<string | number | undefined>;

type T0 = string | number

type T1 = NonNullable<string[] | null | undefined>;

type T1 = string[]

## Parameters<Type>

从函数类型的参数中使用的类型构造元组类型Type。

declare function f1(arg: { a: number; b: string }): void;

type T0 = Parameters<() => string>;

type T0 = []

type T1 = Parameters<(s: string) => void>;

type T1 = [s: string]

type T2 = Parameters<<T>(arg: T) => T>;

type T2 = [arg: unknown]

type T3 = Parameters<typeof f1>

type T3 = [arg: {

a: number;

b: string;

}]

type T4 = Parameters<any>;

type T4 = unknown[]

type T5 = Parameters<never>;

type T5 = never

type T6 = Parameters<string>;

Type 'string' does not satisfy the constraint '(...args: any) => any'.

type T6 = never

type T7 = Parameters<Function>;

Type 'Function' does not satisfy the constraint '(...args: any) => any'.

Type 'Function' provides no match for the signature '(...args: any): any'.

type T7 = never

## ConstructorParameters<Type>

从构造函数类型的类型构造元组或数组类型。它产生一个包含所有参数类型的元组类型（或者never如果Type不是函数的类型）。

type T0 = ConstructorParameters<ErrorConstructor>;

type T0 = [message?: string]

type T1 = ConstructorParameters<FunctionConstructor>;

type T1 = string[]

type T2 = ConstructorParameters<RegExpConstructor>;

type T2 = [pattern: string | RegExp, flags?: string]

type T3 = ConstructorParameters<any>

type T3 = unknown[]

type T4 = ConstructorParameters<Function>;

Type 'Function' does not satisfy the constraint 'abstract new (...args: any) => any'.

Type 'Function' provides no match for the signature 'new (...args: any): any'.

type T4 = never

## ReturnType<Type>

构造一个由 function 的返回类型组成的类型Type。

declare function f1(): { a: number; b: string };

type T0 = ReturnType<() => string>;

type T0 = string

type T1 = ReturnType<(s: string) => void>;

type T1 = void

type T2 = ReturnType<<T>() => T>;

type T2 = unknown

type T3 = ReturnType<<T extends U, U extends number[]>() => T>;

type T3 = number[]

type T4 = ReturnType<typeof f1>;

type T4 = {

a: number;

b: string;

}

type T5 = ReturnType<any>;

type T5 = any

type T6 = ReturnType<never>;

type T6 = never

type T7 = ReturnType<string>;

Type 'string' does not satisfy the constraint '(...args: any) => any'.

type T7 = any

type T8 = ReturnType<Function>;

Type 'Function' does not satisfy the constraint '(...args: any) => any'.

Type 'Function' provides no match for the signature '(...args: any): any'.

type T8 = any

## InstanceType<Type>

构造一个类型，该类型由 中的构造函数的实例类型组成Type。

class C {

x = 0;

y = 0;

}

type T0 = InstanceType<typeof C>;

type T0 = C

type T1 = InstanceType<any>;

type T1 = any

type T2 = InstanceType<never>;

type T2 = never

type T3 = InstanceType<string>;

Type 'string' does not satisfy the constraint 'abstract new (...args: any) => any'.

type T3 = any

type T4 = InstanceType<Function>;

Type 'Function' does not satisfy the constraint 'abstract new (...args: any) => any'.

Type 'Function' provides no match for the signature 'new (...args: any): any'.

type T4 = any

## ThisParameterType<Type>

提取函数类型的this参数的类型，如果函数类型没有参数，则为未知this。

function toHex(this: Number) {

return this.toString(16);

}

function numberToString(n: ThisParameterType<typeof toHex>) {

return toHex.apply(n);

}

## OmitThisParameter<Type>

从中删除this参数Type。如果Type没有显式声明this的参数，则结果是简单的Type。否则，将this创建一个没有参数的新函数类型Type。泛型被删除，只有最后一个重载签名被传播到新的函数类型中。

function toHex(this: Number) {

return this.toString(16);

}

const fiveToHex: OmitThisParameter<typeof toHex> = toHex.bind(5);

console.log(fiveToHex());

## ThisType<Type>

此实用程序不返回转换后的类型。相反，它用作上下文this类型的标记。请注意，noImplicitThis必须启用该标志才能使用此实用程序。

type ObjectDescriptor<D, M> = {

data?: D;

methods?: M & ThisType<D & M>; // Type of 'this' in methods is D & M

};

function makeObject<D, M>(desc: ObjectDescriptor<D, M>): D & M {

let data: object = desc.data || {};

let methods: object = desc.methods || {};

return { ...data, ...methods } as D & M;

}

let obj = makeObject({

data: { x: 0, y: 0 },

methods: {

moveBy(dx: number, dy: number) {

this.x += dx; // Strongly typed this

this.y += dy; // Strongly typed this

},

},

});

obj.x = 10;

obj.y = 20;

obj.moveBy(5, 5);

## 内在字符串操作类型

* Uppercase<StringType>
* Lowercase<StringType>
* Capitalize<StringType>
* Uncapitalize<StringType>

# 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。

## 新语法索引

* declare var 声明全局变量
* declare function 声明全局方法
* declare class 声明全局类
* declare enum 声明全局枚举类型
* declare namespace 声明（含有子属性的）全局对象
* interface 和 type 声明全局类型
* export 导出变量
* export namespace 导出（含有子属性的）对象
* export default ES6 默认导出
* export = commonjs 导出模块
* export as namespace UMD 库声明全局变量
* declare global 扩展全局变量
* declare module 扩展模块
* /// <reference /> 三斜线指令

## 声明语句

假如我们想使用第三方库 jQuery，一种常见的方式是在 html 中通过 <script> 标签引入 jQuery，然后就可以使用全局变量$或 jQuery 了。我们通常这样获取一个 id 是 foo 的元素：

$('#foo');

// or

jQuery('#foo');

但是在 ts 中，编译器并不知道 $ 或 jQuery 是什么东西：

jQuery('#foo');

// ERROR: Cannot find name 'jQuery'.

这时，我们需要使用 declare var 来定义它的类型2：

declare var jQuery: (selector: string) => any;

jQuery('#foo');

上例中，declare var 并没有真的定义一个变量，只是定义了全局变量 jQuery 的类型，仅仅会用于编译时的检查，在编译结果中会被删除。它编译结果是：

jQuery('#foo');

## 声明文件

通常我们会把声明语句放到一个单独的文件（jQuery.d.ts）中，这就是声明文件3：

// src/jQuery.d.ts

declare var jQuery: (selector: string) => any;

// src/index.ts

jQuery('#foo');

声明文件必需以 .d.ts 为后缀。一般来说，ts 会解析项目中所有的 \*.ts 文件，当然也包含以 .d.ts 结尾的文件。所以当我们将 jQuery.d.ts 放到项目中时，其他所有 \*.ts 文件就都可以获得 jQuery 的类型定义了。

/path/to/project

├── src

| ├── index.ts

| └── jQuery.d.ts

└── tsconfig.json

假如仍然无法解析，那么可以检查下 tsconfig.json 中的 files、include 和 exclude 配置，确保其包含了 jQuery.d.ts 文件。这里只演示了全局变量这种模式的声明文件，假如是通过模块导入的方式使用第三方库的话，可以直接下载下来使用，但是更推荐的是使用 @types 统一管理第三方库的声明文件。@types 的使用方式很简单，直接用 npm 安装对应的声明模块即可，以 jQuery 举例：

npm install @types/jquery --save-dev

当一个第三方库没有提供声明文件时，就需要自己书写声明文件了。在不同的场景下，声明文件的内容和使用方式会有所区别。库的使用场景主要有以下几种：

* 全局变量：通过 <script> 标签引入第三方库，注入全局变量
* npm 包：通过 import foo from 'foo' 导入，符合 ES6 模块规范
* UMD 库：既可以通过 <script> 标签引入，又可以通过 import 导入
* 直接扩展全局变量：通过 <script> 标签引入后，改变一个全局变量的结构
* 在 npm 包或 UMD 库中扩展全局变量：引用 npm 包或 UMD 库后，改变一个全局变量的结构
* 模块插件：通过 <script> 或 import 导入后，改变另一个模块的结构

### 全局变量

全局变量是最简单的一种场景，之前举的例子就是通过 <script> 标签引入 jQuery，注入全局变量 $ 和 jQuery。使用全局变量的声明文件时，如果是以 npm install @types/xxx --save-dev 安装的，则不需要任何配置。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（或者对应的源码目录下）：

/path/to/project

├── src

| ├── index.ts

| └── jQuery.d.ts

└── tsconfig.json

如果没有生效，可以检查下 tsconfig.json 中的 files、include 和 exclude 配置，确保其包含了 jQuery.d.ts 文件。全局变量的声明文件主要有以下几种语法：

* declare var 声明全局变量：在所有的声明语句中，declare var 是最简单的，如之前所学，它能够用来定义一个全局变量的类型。与其类似的，还有 declare let 和 declare const，使用 let 与使用 var 没有什么区别：

// src/jQuery.d.ts

declare let jQuery: (selector: string) => any;

// src/index.ts

jQuery('#foo');

// 使用 declare let 定义的 jQuery 类型，允许修改这个全局变量

jQuery = function(selector) {

return document.querySelector(selector);

};

而当我们使用 const 定义时，表示此时的全局变量是一个常量，不允许再去修改它的值了：

// src/jQuery.d.ts

declare const jQuery: (selector: string) => any;

jQuery('#foo');

// 使用 declare const 定义的 jQuery 类型，禁止修改这个全局变量

jQuery = function(selector) {

return document.querySelector(selector);

};

// ERROR: Cannot assign to 'jQuery' because it is a constant or a read-only property.

一般来说，全局变量都是禁止修改的常量，所以大部分情况都应该使用 const 而不是 var 或 let。需要注意的是，声明语句中只能定义类型，切勿在声明语句中定义具体的实现：

declare const jQuery = function(selector) {

return document.querySelector(selector);

};

// ERROR: An implementation cannot be declared in ambient contexts.

* declare function 声明全局方法：用来定义全局函数的类型。jQuery 其实就是一个函数，所以也可以用 function 来定义：

declare function 用来定义全局函数的类型。jQuery 其实就是一个函数，所以也可以用 function 来定义：

// src/jQuery.d.ts

declare function jQuery(selector: string): any;

// src/index.ts

jQuery('#foo');

在函数类型的声明语句中，函数重载也是支持的：

// src/jQuery.d.ts

declare function jQuery(selector: string): any;

declare function jQuery(domReadyCallback: () => any): any;

// src/index.ts

jQuery('#foo');

jQuery(function() {

alert('Dom Ready!');

});

* declare class 声明全局类：当全局变量是一个类的时候，我们用 declare class 来定义它的类型：

// src/Animal.d.ts

declare class Animal {

name: string;

constructor(name: string);

sayHi(): string;

}

// src/index.ts

let cat = new Animal('Tom');

同样的，declare class 语句也只能用来定义类型，不能用来定义具体的实现，比如定义 sayHi 方法的具体实现则会报错：

// src/Animal.d.ts

declare class Animal {

name: string;

constructor(name: string);

sayHi() {

return `My name is ${this.name}`;

};

// ERROR: An implementation cannot be declared in ambient contexts.

}

* declare enum 声明全局枚举类型：使用 declare enum 定义的枚举类型也称作外部枚举（Ambient Enums），举例如下。

// src/Directions.d.ts

declare enum Directions {

Up,

Down,

Left,

Right

}

// src/index.ts

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

与其他全局变量的类型声明一致，declare enum 仅用来定义类型，而不是具体的值。Directions.d.ts 仅仅会用于编译时的检查，声明文件里的内容在编译结果中会被删除。它编译结果是：

var directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

其中 Directions 是由第三方库定义好的全局变量。

* declare namespace 声明（含有子属性的）全局对象：namespace 是 ts 早期时为了解决模块化而创造的关键字，中文称为命名空间。由于历史遗留原因，在早期还没有 ES6 的时候，ts 提供了一种模块化方案，使用 module 关键字表示内部模块。但由于后来 ES6 也使用了 module 关键字，ts 为了兼容 ES6，使用 namespace 替代了自己的 module，更名为命名空间。

随着 ES6 的广泛应用，现在已经不建议再使用 ts 中的 namespace，而推荐使用 ES6 的模块化方案了，故我们不再需要学习 namespace 的使用了。namespace 被淘汰了，但是在声明文件中，declare namespace 还是比较常用的，它用来表示全局变量是一个对象，包含很多子属性。比如 jQuery 是一个全局变量，它是一个对象，提供了一个 jQuery.ajax 方法可以调用，那么我们就应该使用 declare namespace jQuery 来声明这个拥有多个子属性的全局变量。

// src/jQuery.d.ts

declare namespace jQuery {

function ajax(url: string, settings?: any): void;

}

// src/index.ts

jQuery.ajax('/api/get\_something');

注意，在 declare namespace 内部，我们直接使用 function ajax 来声明函数，而不是使用 declare function ajax。类似的，也可以使用 const, class, enum 等语句：

// src/jQuery.d.ts

declare namespace jQuery {

function ajax(url: string, settings?: any): void;

const version: number;

class Event {

blur(eventType: EventType): void

}

enum EventType {

CustomClick

}

}

// src/index.ts

jQuery.ajax('/api/get\_something');

console.log(jQuery.version);

const e = new jQuery.Event();

e.blur(jQuery.EventType.CustomClick);

如果对象拥有深层的层级，则需要用嵌套的 namespace 来声明深层的属性的类型：

// src/jQuery.d.ts

declare namespace jQuery {

function ajax(url: string, settings?: any): void;

namespace fn {

function extend(object: any): void;

}

}

// src/index.ts

jQuery.ajax('/api/get\_something');

jQuery.fn.extend({

check: function() {

return this.each(function() {

this.checked = true;

});

}

});

假如 jQuery 下仅有 fn 这一个属性（没有 ajax 等其他属性或方法），则可以不需要嵌套 namespace：

// src/jQuery.d.ts

declare namespace jQuery.fn {

function extend(object: any): void;

}

// src/index.ts

jQuery.fn.extend({

check: function() {

return this.each(function() {

this.checked = true;

});

}

});

* interface 和 type：除了全局变量之外，可能有一些类型我们也希望能暴露出来。在类型声明文件中，我们可以直接使用 interface 或 type 来声明一个全局的接口或类型。

// src/jQuery.d.ts

interface AjaxSettings {

method?: 'GET' | 'POST'

data?: any;

}

declare namespace jQuery {

function ajax(url: string, settings?: AjaxSettings): void;

}

这样的话，在其他文件中也可以使用这个接口或类型了：

// src/index.ts

let settings: AjaxSettings = {

method: 'POST',

data: {

name: 'foo'

}

};

jQuery.ajax('/api/post\_something', settings);

* 防止命名冲突：暴露在最外层的 interface 或 type 会作为全局类型作用于整个项目中，我们应该尽可能的减少全局变量或全局类型的数量。故最好将他们放到 namespace 下。

// src/jQuery.d.ts

declare namespace jQuery {

interface AjaxSettings {

method?: 'GET' | 'POST'

data?: any;

}

function ajax(url: string, settings?: AjaxSettings): void;

}

注意，在使用这个 interface 的时候，也应该加上 jQuery 前缀：

// src/index.ts

let settings: jQuery.AjaxSettings = {

method: 'POST',

data: {

name: 'foo'

}

};

jQuery.ajax('/api/post\_something', settings);

* 声明合并：假如 jQuery 既是一个函数，可以直接被调用 jQuery('#foo')，又是一个对象，拥有子属性 jQuery.ajax()（事实确实如此），那么我们可以组合多个声明语句，它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 包

一般我们通过 import foo from 'foo' 导入一个 npm 包，这是符合 ES6 模块规范的。在我们尝试给一个 npm 包创建声明文件之前，需要先看看它的声明文件是否已经存在。一般来说，npm 包的声明文件可能存在于两个地方：

1. 与该 npm 包绑定在一起。判断依据是 package.json 中有 types 字段，或者有一个 index.d.ts 声明文件。这种模式不需要额外安装其他包，是最为推荐的，所以以后我们自己创建 npm 包的时候，最好也将声明文件与 npm 包绑定在一起。
2. 发布到 @types 里。我们只需要尝试安装一下对应的 @types 包就知道是否存在该声明文件，安装命令是 npm install @types/foo --save-dev。这种模式一般是由于 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。

假如以上两种方式都没有找到对应的声明文件，那么我们就需要自己为它写声明文件了。由于是通过 import 语句导入的模块，所以声明文件存放的位置也有所约束，一般有两种方案：

1. 创建一个 node\_modules/@types/foo/index.d.ts 文件，存放 foo 模块的声明文件。这种方式不需要额外的配置，但是 node\_modules 目录不稳定，代码也没有被保存到仓库中，无法回溯版本，有不小心被删除的风险，故不太建议用这种方案，一般只用作临时测试。
2. 创建一个 types 目录，专门用来管理自己写的声明文件，将 foo 的声明文件放到 types/foo/index.d.ts 中。这种方式需要配置下 tsconfig.json 中的 paths 和 baseUrl 字段。

tsconfig.json 内容：

{

"compilerOptions": {

"module": "commonjs",

"baseUrl": "./",

"paths": {

"\*": ["types/\*"]

}

}

}

如此配置之后，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。注意 module 配置可以有很多种选项，不同的选项会影响模块的导入导出模式。这里我们使用了 commonjs 这个最常用的选项，后面的教程也都默认使用的这个选项。

npm 包的声明文件主要有以下几种语法：

* export 导出变量

npm 包的声明文件与全局变量的声明文件有很大区别。在 npm 包的声明文件中，使用 declare 不再会声明一个全局变量，而只会在当前文件中声明一个局部变量。只有在声明文件中使用 export 导出，然后在使用方 import 导入后，才会应用到这些类型声明。export 的语法与普通的 ts 中的语法类似，区别仅在于声明文件中禁止定义具体的实现：

// types/foo/index.d.ts

export const name: string;

export function getName(): string;

export class Animal {

constructor(name: string);

sayHi(): string;

}

export enum Directions {

Up,

Down,

Left,

Right

}

export interface Options {

data: any;

}

对应的导入和使用模块应该是这样：

// src/index.ts

import { name, getName, Animal, Directions, Options } from 'foo';

console.log(name);

let myName = getName();

let cat = new Animal('Tom');

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

let options: Options = {

data: {

name: 'foo'

}

};

我们也可以使用 declare 先声明多个变量，最后再用 export 一次性导出。上例的声明文件可以等价的改写为：

// types/foo/index.d.ts

declare const name: string;

declare function getName(): string;

declare class Animal {

constructor(name: string);

sayHi(): string;

}

declare enum Directions {

Up,

Down,

Left,

Right

}

interface Options {

data: any;

}

export { name, getName, Animal, Directions, Options };

注意，与全局变量的声明文件类似，interface 前是不需要 declare 的。

* export namespace 导出（含有子属性的）对象

与 declare namespace 类似，export namespace 用来导出一个拥有子属性的对象：

// types/foo/index.d.ts

export namespace foo {

const name: string;

namespace bar {

function baz(): string;

}

}

// src/index.ts

import { foo } from 'foo';

console.log(foo.name);

foo.bar.baz();

* export default ES6 默认导出

在 ES6 模块系统中，使用 export default 可以导出一个默认值，使用方可以用 import foo from 'foo' 而不是 import { foo } from 'foo' 来导入这个默认值。在类型声明文件中，export default 用来导出默认值的类型：

// types/foo/index.d.ts

export default function foo(): string;

// src/index.ts

import foo from 'foo';

foo();

注意，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 = commonjs 导出模块

在 commonjs 规范中，我们用以下方式来导出一个模块：

// 整体导出

module.exports = foo;

// 单个导出

exports.bar = bar;

在 ts 中，针对这种模块导出，有多种方式可以导入，第一种方式是 const ... = require：

// 整体导入

const foo = require('foo');

// 单个导入

const bar = require('foo').bar;

第二种方式是 import ... from，注意针对整体导出，需要使用 import \* as 来导入：

// 整体导入

import \* as foo from 'foo';

// 单个导入

import { bar } from 'foo';

第三种方式是 import ... require，这也是 ts 官方推荐的方式：

// 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，假如要为它写类型声明文件的话，就需要使用到 export = 这种语法了：

// types/foo/index.d.ts

export = foo;

declare function foo(): string;

declare namespace foo {

const bar: number;

}

上例中使用了 export = 之后，就不能再单个导出 export { bar } 了。所以我们通过声明合并，使用 declare namespace foo 来将 bar 合并到 foo 里。准确地讲，export = 不仅可以用在声明文件中，也可以用在普通的 ts 文件中。实际上，import ... require 和 export = 都是 ts 为了兼容 AMD 规范和 commonjs 规范而创立的新语法，由于并不常用也不推荐使用，所以这里就不详细介绍了，感兴趣的可以看官方文档。

由于很多第三方库是 commonjs 规范的，所以声明文件也就不得不用到 export = 这种语法了。但是还是需要再强调下，相比与 export =，我们更推荐使用 ES6 标准的 export default 和 export。

### 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。相比于 npm 包的类型声明文件，我们需要额外声明一个全局变量，为了实现这种方式，ts 提供了一个新语法 export as namespace。一般使用 export as namespace 时，都是先有了 npm 包的声明文件，再基于它添加一条 export as namespace 语句，即可将声明好的一个变量声明为全局变量，举例如下：

// types/foo/index.d.ts

export as namespace foo;

export = foo;

declare function foo(): string;

declare namespace foo {

const bar: number;

}

当然它也可以与 export default 一起使用：

// types/foo/index.d.ts

export as namespace foo;

export default foo;

declare function foo(): string;

declare namespace foo {

const bar: number;

}

### 直接扩展全局变量

有的第三方库扩展了一个全局变量，可是此全局变量的类型却没有相应的更新过来，就会导致 ts 编译错误，此时就需要扩展全局变量的类型。比如扩展 String 类型：

interface String {

prependHello(): string;

}

'foo'.prependHello();

通过声明合并，使用 interface String 即可给 String 添加属性或方法。也可以使用 declare namespace 给已有的命名空间添加类型声明：

// types/jquery-plugin/index.d.ts

declare namespace JQuery {

interface CustomOptions {

bar: string;

}

}

interface JQueryStatic {

foo(options: JQuery.CustomOptions): string;

}

// src/index.ts

jQuery.foo({

bar: ''

});

### 在npm包或UMD库中扩展全局变量

如之前所说，对于一个 npm 包或者 UMD 库的声明文件，只有 export 导出的类型声明才能被导入。所以对于 npm 包或 UMD 库，如果导入此库之后会扩展全局变量，则需要使用另一种语法在声明文件中扩展全局变量的类型，那就是 declare global

使用 declare global 可以在 npm 包或者 UMD 库的声明文件中扩展全局变量的类型：

// types/foo/index.d.ts

declare global {

interface String {

prependHello(): string;

}

}

export {};

// src/index.ts

'bar'.prependHello();

注意即使此声明文件不需要导出任何东西，仍然需要导出一个空对象，用来告诉编译器这是一个模块的声明文件，而不是一个全局变量的声明文件。

### 模块插件

有时通过 import 导入一个模块插件，可以改变另一个原有模块的结构。此时如果原有模块已经有了类型声明文件，而插件模块没有类型声明文件，就会导致类型不完整，缺少插件部分的类型。ts 提供了一个语法 declare module，它可以用来扩展原有模块的类型。

如果是需要扩展原有模块的话，需要在类型声明文件中先引用原有模块，再使用 declare module 扩展原有模块：

// types/moment-plugin/index.d.ts

import \* as moment from 'moment';

declare module 'moment' {

export function foo(): moment.CalendarKey;

}

// src/index.ts

import \* as moment from 'moment';

import 'moment-plugin';

moment.foo();

declare module 也可用于在一个文件中一次性声明多个模块的类型：

// types/foo-bar.d.ts

declare module 'foo' {

export interface Foo {

foo: string;

}

}

declare module 'bar' {

export function bar(): string;

}

// src/index.ts

import { Foo } from 'foo';

import \* as bar from 'bar';

let f: Foo;

bar.bar();

### 声明文件中的依赖

一个声明文件有时会依赖另一个声明文件中的类型，比如在前面的 declare module 的例子中，我们就在声明文件中导入了 moment，并且使用了 moment.CalendarKey 这个类型。除了可以在声明文件中通过 import 导入另一个声明文件中的类型之外，还有一个语法也可以用来导入另一个声明文件，那就是三斜线指令。

**三斜线指令：**与 namespace 类似，三斜线指令也是 ts 在早期版本中为了描述模块之间的依赖关系而创造的语法。随着 ES6 的广泛应用，现在已经不建议再使用 ts 中的三斜线指令来声明模块之间的依赖关系了。但是在声明文件中，它还是有一定的用武之地。类似于声明文件中的 import，它可以用来导入另一个声明文件。与 import 的区别是，当且仅当在以下几个场景下，我们才需要使用三斜线指令替代 import：

* 当我们在书写一个全局变量的声明文件时：在全局变量的声明文件中，是不允许出现 import, export 关键字的。一旦出现了，那么他就会被视为一个 npm 包或 UMD 库，就不再是全局变量的声明文件了。故当我们在书写一个全局变量的声明文件时，如果需要引用另一个库的类型，那么就必须用三斜线指令了

// types/jquery-plugin/index.d.ts

/// <reference types="jquery" />

declare function foo(options: JQuery.AjaxSettings): string;

// src/index.ts

foo({});

三斜线指令的语法如上，/// 后面使用 xml 的格式添加了对 jquery 类型的依赖，这样就可以在声明文件中使用 JQuery.AjaxSettings 类型了。注意，三斜线指令必须放在文件的最顶端，三斜线指令的前面只允许出现单行或多行注释。

* 当我们需要依赖一个全局变量的声明文件时：当我们需要依赖一个全局变量的声明文件时，由于全局变量不支持通过 import 导入，当然也就必须使用三斜线指令来引入了

// types/node-plugin/index.d.ts

/// <reference types="node" />

export function foo(p: NodeJS.Process): string;

// src/index.ts

import { foo } from 'node-plugin';

foo(global.process);

在上面的例子中，我们通过三斜线指引入了 node 的类型，然后在声明文件中使用了 NodeJS.Process 这个类型。最后在使用到 foo 的时候，传入了 node 中的全局变量 process。由于引入的 node 中的类型都是全局变量的类型，它们是没有办法通过 import 来导入的，所以这种场景下也只能通过三斜线指令来引入了。以上两种使用场景下，都是由于需要书写或需要依赖全局变量的声明文件，所以必须使用三斜线指令。在其他的一些不是必要使用三斜线指令的情况下，就都需要使用 import 来导入。

**拆分声明文件：**当我们的全局变量的声明文件太大时，可以通过拆分为多个文件，然后在一个入口文件中将它们一一引入，来提高代码的可维护性。比如 jQuery 的声明文件就是这样的：

// node\_modules/@types/jquery/index.d.ts

/// <reference types="sizzle" />

/// <reference path="JQueryStatic.d.ts" />

/// <reference path="JQuery.d.ts" />

/// <reference path="misc.d.ts" />

/// <reference path="legacy.d.ts" />

export = jQuery;

其中用到了 types 和 path 两种不同的指令。它们的区别是：types 用于声明对另一个库的依赖，而 path 用于声明对另一个文件的依赖。上例中，sizzle 是与 jquery 平行的另一个库，所以需要使用 types="sizzle" 来声明对它的依赖。而其他的三斜线指令就是将 jquery 的声明拆分到不同的文件中了，然后在这个入口文件中使用 path="foo" 将它们一一引入。

### 自动生成声明文件

如果库的源码本身就是由 ts 写的，那么在使用 tsc 脚本将 ts 编译为 js 的时候，添加 declaration 选项，就可以同时也生成 .d.ts 声明文件了。我们可以在命令行中添加 --declaration（简写 -d），或者在 tsconfig.json 中添加 declaration 选项。这里以 tsconfig.json 为例：

{

"compilerOptions": {

"module": "commonjs",

"outDir": "lib",

"declaration": true,

}

}

上例中添加了 outDir 选项，将 ts 文件的编译结果输出到 lib 目录下，然后添加了 declaration 选项，设置为 true，表示将会由 ts 文件自动生成 .d.ts 声明文件，也会输出到 lib 目录下。运行 tsc 之后，目录结构如下：

// src/index.ts

export \* from './bar';

export default function foo() {

return 'foo';

}

// src/bar/index.ts

export function bar() {

return 'bar';

}

// lib/index.d.ts

export \* from './bar';

export default function foo(): string;

// lib/bar/index.d.ts

export declare function bar(): string;

自动生成的声明文件基本保持了源码的结构，而将具体实现去掉了，生成了对应的类型声明。使用 tsc 自动生成声明文件时，每个 ts 文件都会对应一个 .d.ts 声明文件。这样的好处是，使用方不仅可以在使用 import foo from 'foo' 导入默认的模块时获得类型提示，还可以在使用 import bar from 'foo/lib/bar' 导入一个子模块时，也获得对应的类型提示。除了 declaration 选项之外，还有几个选项也与自动生成声明文件有关：

* declarationDir 设置生成 .d.ts 文件的目录
* declarationMap 对每个 .d.ts 文件，都生成对应的 .d.ts.map（sourcemap）文件
* emitDeclarationOnly 仅生成 .d.ts 文件，不生成 .js 文件

### 发布声明文件

当我们为一个库写好了声明文件之后，下一步就是将它发布出去了。此时有两种方案：

* 将声明文件和源码放在一起

如果声明文件是通过 tsc 自动生成的，那么无需做任何其他配置，只需要把编译好的文件也发布到 npm 上，使用方就可以获取到类型提示了。如果是手动写的声明文件，那么需要满足以下条件之一，才能被正确的识别：

* + 给 package.json 中的 types 或 typings 字段指定一个类型声明文件地址
  + 在项目根目录下，编写一个 index.d.ts 文件
  + 针对入口文件（package.json 中的 main 字段指定的入口文件），编写一个同名不同后缀的 .d.ts 文件

第一种方式是给 package.json 中的 types 或 typings 字段指定一个类型声明文件地址。比如：

{

"name": "foo",

"version": "1.0.0",

"main": "lib/index.js",

"types": "foo.d.ts",

}

指定了 types 为 foo.d.ts 之后，导入此库的时候，就会去找 foo.d.ts 作为此库的类型声明文件了。typings 与 types 一样，只是另一种写法。如果没有指定 types 或 typings，那么就会在根目录下寻找 index.d.ts 文件，将它视为此库的类型声明文件。如果没有找到 index.d.ts 文件，那么就会寻找入口文件（package.json 中的 main 字段指定的入口文件）是否存在对应同名不同后缀的 .d.ts 文件。比如 package.json 是这样时：

{

"name": "foo",

"version": "1.0.0",

"main": "lib/index.js"

}

就会先识别 package.json 中是否存在 types 或 typings 字段。发现不存在，那么就会寻找是否存在 index.d.ts 文件。如果还是不存在，那么就会寻找是否存在 lib/index.d.ts 文件。假如说连 lib/index.d.ts 都不存在的话，就会被认为是一个没有提供类型声明文件的库了。

有的库为了支持导入子模块，比如 import bar from 'foo/lib/bar'，就需要额外再编写一个类型声明文件 lib/bar.d.ts 或者 lib/bar/index.d.ts，这与自动生成声明文件类似，一个库中同时包含了多个类型声明文件。

* 将声明文件发布到 @types 下

如果我们是在给别人的仓库添加类型声明文件，但原作者不愿意合并 pull request，那么就需要将声明文件发布到 @types 下。与普通的 npm 模块不同，@types 是统一由 DefinitelyTyped 管理的。要将声明文件发布到 @types 下，就需要给 DefinitelyTyped 创建一个 pull-request，其中包含了类型声明文件，测试代码，以及 tsconfig.json 等。pull-request 需要符合它们的规范，并且通过测试，才能被合并，稍后就会被自动发布到 @types 下。

这两种方案中优先选择第一种方案。保持声明文件与源码在一起，使用时就不需要额外增加单独的声明文件库的依赖了，而且也能保证声明文件的版本与源码的版本保持一致。仅当我们在给别人的仓库添加类型声明文件，但原作者不愿意合并 pull request 时，才需要使用第二种方案，将声明文件发布到 @types 下。

# 内置对象

JavaScript 中有很多内置对象，它们可以直接在 TypeScript 中当做定义好了的类型。内置对象是指根据标准在全局作用域（Global）上存在的对象。这里的标准是指 ECMAScript 和其他环境（比如 DOM）的标准。

## ECMAScript 的内置对象

ECMAScript 标准提供的内置对象有：

Boolean、Error、Date、RegExp 等。

我们可以在 TypeScript 中将变量定义为这些类型：

let b: Boolean = new Boolean(1);

let e: Error = new Error('Error occurred');

let d: Date = new Date();

let r: RegExp = /[a-z]/;

更多的内置对象，可以查看 MDN 的文档。

## DOM 和 BOM 的内置对象

DOM 和 BOM 提供的内置对象有：

Document、HTMLElement、Event、NodeList 等。

TypeScript 中会经常用到这些类型：

let body: HTMLElement = document.body;

let allDiv: NodeList = document.querySelectorAll('div');

document.addEventListener('click', function(e: MouseEvent) {

// Do something

});

它们的定义文件同样在 TypeScript 核心库的定义文件中。

## TypeScript 核心库的定义文件

TypeScript 核心库的定义文件中定义了所有浏览器环境需要用到的类型，并且是预置在 TypeScript 中的。当你在使用一些常用的方法的时候，TypeScript 实际上已经帮你做了很多类型判断的工作了，比如：

Math.pow(10, '2');

// index.ts(1,14): error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.

上面的例子中，Math.pow 必须接受两个 number 类型的参数。事实上 Math.pow 的类型定义如下：

interface Math {

/\*\*

\* Returns the value of a base expression taken to a specified power.

\* @param x The base value of the expression.

\* @param y The exponent value of the expression.

\*/

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

}

再举一个 DOM 中的例子：

document.addEventListener('click', function(e) {

console.log(e.targetCurrent);

});

// index.ts(2,17): error TS2339: Property 'targetCurrent' does not exist on type 'MouseEvent'.

上面的例子中，addEventListener 方法是在 TypeScript 核心库中定义的：

interface Document extends Node, GlobalEventHandlers, NodeSelector, DocumentEvent {

addEventListener(type: string, listener: (ev: MouseEvent) => any, useCapture?: boolean): void;

}

所以 e 被推断成了 MouseEvent，而 MouseEvent 是没有 targetCurrent 属性的，所以报错了。注意，TypeScript 核心库的定义中不包含 Node.js 部分。Node.js 不是内置对象的一部分，如果想用 TypeScript 写 Node.js，则需要引入第三方声明文件：

npm install @types/node --save-dev

# 类型别名

类型别名用来给一个类型起个新名字。

type Name = string;

type NameResolver = () => string;

type NameOrResolver = Name | NameResolver;

function getName(n: NameOrResolver): Name {

if (typeof n === 'string') {

return n;

} else {

return n();

}

}

上例中，我们使用 type 创建类型别名。类型别名常用于联合类型。

# 字符串字面量类型

字符串字面量类型用来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

}

handleEvent(document.getElementById('hello'), 'scroll'); // 没问题

handleEvent(document.getElementById('world'), 'dblclick'); // 报错，event 不能为 'dblclick'

// index.ts(7,47): error TS2345: Argument of type '"dblclick"' is not assignable to parameter of type 'EventNames'.

上例中，我们使用 type 定了一个字符串字面量类型 EventNames，它只能取三种字符串中的一种。注意，类型别名与字符串字面量类型都是使用 type 进行定义。

# 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。 比如，你可以定义一对值分别为string和number类型的元组。

// Declare a tuple type

let x: [string, number];

// Initialize it

x = ['hello', 10]; // OK

// Initialize it incorrectly

x = [10, 'hello']; // Error

当访问一个已知索引的元素，会得到正确的类型：

console.log(x[0].substr(1)); // OK

console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'

当访问一个越界的元素，会使用联合类型替代：

x[3] = 'world'; // OK, 字符串可以赋值给(string | number)类型

console.log(x[5].toString()); // OK, 'string' 和 'number' 都有 toString

x[6] = true; // Error, 布尔不是(string | number)类型

# 枚举

枚举（Enum）类型用于取值被限定在一定范围内的场景。枚举使用 enum 关键字来定义：

enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

枚举成员会被赋值为从 0 开始递增的数字，同时也会对枚举值到枚举名进行反向映射：

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

事实上，上面的例子会被编译为：

var Days;

(function (Days) {

Days[Days["Sun"] = 0] = "Sun";

Days[Days["Mon"] = 1] = "Mon";

Days[Days["Tue"] = 2] = "Tue";

Days[Days["Wed"] = 3] = "Wed";

Days[Days["Thu"] = 4] = "Thu";

Days[Days["Fri"] = 5] = "Fri";

Days[Days["Sat"] = 6] = "Sat";

})(Days || (Days = {}));

## 手动赋值

我们也可以给枚举项手动赋值：

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

上面的例子中，递增到 3 的时候与前面的 Sun 的取值重复了，但是 TypeScript 并没有报错，导致 Days[3] 的值先是 "Sun"，而后又被 "Wed" 覆盖了。编译的结果是：

var Days;

(function (Days) {

Days[Days["Sun"] = 3] = "Sun";

Days[Days["Mon"] = 1] = "Mon";

Days[Days["Tue"] = 2] = "Tue";

Days[Days["Wed"] = 3] = "Wed";

Days[Days["Thu"] = 4] = "Thu";

Days[Days["Fri"] = 5] = "Fri";

Days[Days["Sat"] = 6] = "Sat";

})(Days || (Days = {}));

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"};

var Days;

(function (Days) {

Days[Days["Sun"] = 7] = "Sun";

Days[Days["Mon"] = 8] = "Mon";

Days[Days["Tue"] = 9] = "Tue";

Days[Days["Wed"] = 10] = "Wed";

Days[Days["Thu"] = 11] = "Thu";

Days[Days["Fri"] = 12] = "Fri";

Days[Days["Sat"] = "S"] = "Sat";

})(Days || (Days = {}));

当然，手动赋值的枚举项也可以为小数或负数，此时后续未手动赋值的项的递增步长仍为 1：

enum Days {Sun = 7, Mon = 1.5, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 7); // true

console.log(Days["Mon"] === 1.5); // true

console.log(Days["Tue"] === 2.5); // true

console.log(Days["Sat"] === 6.5); // true

## 计算所得项

枚举项有两种类型：常数项（constant member）和计算所得项（computed member）

前面的例子都是常数项，一个典型的计算所得项的例子：

enum Color {Red, Green, Blue = "blue".length};

上面的例子中，"blue".length 就是一个计算所得项。上面的例子不会报错，但是如果紧接在计算所得项后面的是未手动赋值的项，那么它就会因为无法获得初始值而报错：

enum Color {Red = "red".length, Green, Blue};

// index.ts(1,33): error TS1061: Enum member must have initializer.

// index.ts(1,40): error TS1061: Enum member must have initializer.

当满足以下条件时，枚举成员被当作是常数：

* 不具有初始化函数并且之前的枚举成员是常数。在这种情况下，当前枚举成员的值为上一个枚举成员的值加 1。但第一个枚举元素是个例外。如果它没有初始化方法，那么它的初始值为 0。
* 枚举成员使用常数枚举表达式初始化。常数枚举表达式是 TypeScript 表达式的子集，它可以在编译阶段求值。当一个表达式满足下面条件之一时，它就是一个常数枚举表达式：
  + 数字字面量
  + 引用之前定义的常数枚举成员（可以是在不同的枚举类型中定义的）如果这个成员是在同一个枚举类型中定义的，可以使用非限定名来引用
  + 带括号的常数枚举表达式
  + +, -, ~ 一元运算符应用于常数枚举表达式
  + +, -, \*, /, %, <<, >>, >>>, &, |, ^ 二元运算符，常数枚举表达式做为其一个操作对象。若常数枚举表达式求值后为 NaN 或 Infinity，则会在编译阶段报错

所有其它情况的枚举成员被当作是需要计算得出的值。

## 常数枚举

常数枚举是使用 const enum 定义的枚举类型：

const enum Directions {

Up,

Down,

Left,

Right

}

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

常数枚举与普通枚举的区别是，它会在编译阶段被删除，并且不能包含计算成员。上例的编译结果是：

var directions = [0 /\* Up \*/, 1 /\* Down \*/, 2 /\* Left \*/, 3 /\* Right \*/];

假如包含了计算成员，则会在编译阶段报错：

const enum Color {Red, Green, Blue = "blue".length};

// index.ts(1,38): error TS2474: In 'const' enum declarations member initializer must be constant expression.

## 外部枚举

外部枚举（Ambient Enums）是使用 declare enum 定义的枚举类型：

declare enum Directions {

Up,

Down,

Left,

Right

}

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

之前提到过，declare 定义的类型只会用于编译时的检查，编译结果中会被删除。上例的编译结果是：

var directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

外部枚举与声明语句一样，常出现在声明文件中。同时使用 declare 和 const 也是可以的：

declare const enum Directions {

Up,

Down,

Left,

Right

}

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];

编译结果：

var directions = [0 /\* Up \*/, 1 /\* Down \*/, 2 /\* Left \*/, 3 /\* Right \*/];

TypeScript 的枚举类型的概念来源于 C#。

# 类

传统方法中，JavaScript 通过构造函数实现类的概念，通过原型链实现继承。而在 ES6 中，我们终于迎来了 class。TypeScript 除了实现了所有 ES6 中的类的功能以外，还添加了一些新的用法。

## 类的概念

这里对类相关的概念做一个简单的介绍。

* 类（Class）：定义了一件事物的抽象特点，包含它的属性和方法
* 对象（Object）：类的实例，通过 new 生成
* 面向对象（OOP）的三大特性：封装、继承、多态
* 封装（Encapsulation）：将对数据的操作细节隐藏起来，只暴露对外的接口。外界调用端不需要（也不可能）知道细节，就能通过对外提供的接口来访问该对象，同时也保证了外界无法任意更改对象内部的数据
* 继承（Inheritance）：子类继承父类，子类除了拥有父类的所有特性外，还有一些更具体的特性
* 多态（Polymorphism）：由继承而产生了相关的不同的类，对同一个方法可以有不同的响应。比如 Cat 和 Dog 都继承自 Animal，但是分别实现了自己的 eat 方法。此时针对某一个实例，我们无需了解它是 Cat 还是 Dog，就可以直接调用 eat 方法，程序会自动判断出来应该如何执行 eat
* 存取器（getter & setter）：用以改变属性的读取和赋值行为
* 修饰符（Modifiers）：修饰符是一些关键字，用于限定成员或类型的性质。比如 public 表示公有属性或方法
* 抽象类（Abstract Class）：抽象类是供其他类继承的基类，抽象类不允许被实例化。抽象类中的抽象方法必须在子类中被实现
* 接口（Interfaces）：不同类之间公有的属性或方法，可以抽象成一个接口。接口可以被类实现（implements）。一个类只能继承自另一个类，但是可以实现多个接口

## ES6 中类的用法

### 属性和方法

使用 class 定义类，使用 constructor 定义构造函数。通过 new 生成新实例的时候，会自动调用构造函数。

class Animal {

public name;

constructor(name) {

this.name = name;

}

sayHi() {

return `My name is ${this.name}`;

}

}

let a = new Animal('Jack');

console.log(a.sayHi()); // My name is Jack

### 类的继承

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

class Cat extends Animal {

constructor(name) {

super(name); // 调用父类的 constructor(name)

console.log(this.name);

}

sayHi() {

return 'Meow, ' + super.sayHi(); // 调用父类的 sayHi()

}

}

let c = new Cat('Tom'); // Tom

console.log(c.sayHi()); // Meow, My name is Tom

存取器

使用 getter 和 setter 可以改变属性的赋值和读取行为：

class Animal {

constructor(name) {

this.name = name;

}

get name() {

return 'Jack';

}

set name(value) {

console.log('setter: ' + value);

}

}

let a = new Animal('Kitty'); // setter: Kitty

a.name = 'Tom'; // setter: Tom

console.log(a.name); // Jack

### 存取器

使用 getter 和 setter 可以改变属性的赋值和读取行为：

class Animal {

constructor(name) {

this.name = name;

}

get name() {

return 'Jack';

}

set name(value) {

console.log('setter: ' + value);

}

}

let a = new Animal('Kitty'); // setter: Kitty

a.name = 'Tom'; // setter: Tom

console.log(a.name); // Jack

### 静态方法

使用 static 修饰符修饰的方法称为静态方法，它们不需要实例化，而是直接通过类来调用：

class Animal {

static isAnimal(a) {

return a instanceof Animal;

}

}

let a = new Animal('Jack');

Animal.isAnimal(a); // true

a.isAnimal(a); // TypeError: a.isAnimal is not a function

## ES7 中类的用法

### 实例属性

ES6 中实例的属性只能通过构造函数中的 this.xxx 来定义，ES7 提案中可以直接在类里面定义：

class Animal {

name = 'Jack';

constructor() {

// ...

}

}

let a = new Animal();

console.log(a.name); // Jack

### 静态属性

ES7 提案中，可以使用 static 定义一个静态属性：

class Animal {

static num = 42;

constructor() {

// ...

}

}

console.log(Animal.num); // 42

## TypeScript 中类的用法

### public private 和 protected

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

下面举一些例子：

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

name 被设置为了 public，所以直接访问实例的 name 属性是允许的。很多时候，我们希望有的属性是无法直接存取的，这时候就可以用 private 了：

class Animal {

private name;

public constructor(name) {

this.name = name;

}

}

let a = new Animal('Jack');

console.log(a.name);

a.name = 'Tom';

// index.ts(9,13): error TS2341: Property 'name' is private and only accessible within class 'Animal'.

// index.ts(10,1): error TS2341: Property 'name' is private and only accessible within class 'Animal'.

TypeScript 编译之后的代码中，并没有限制 private 属性在外部的可访问性。上面的例子编译后的代码是：

var Animal = (function () {

function Animal(name) {

this.name = name;

}

return Animal;

})();

var a = new Animal('Jack');

console.log(a.name);

a.name = 'Tom';

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);

}

}

// index.ts(11,17): error TS2341: Property 'name' is private and only accessible within class 'Animal'.

而如果是用 protected 修饰，则允许在子类中访问：

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');

// index.ts(7,19): TS2675: Cannot extend a class 'Animal'. Class constructor is marked as private.

// index.ts(13,9): TS2673: Constructor of class 'Animal' is private and only accessible within the class declaration.

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');

// index.ts(13,9): TS2674: Constructor of class 'Animal' is protected and only accessible within the class declaration.

### 参数属性

修饰符和readonly还可以使用在构造函数参数中，等同于类中定义该属性同时给该属性赋值，使代码更简洁。

class Animal {

// public name: string;

public constructor(public name) {

// this.name = name;

}

}

### readonly

只读属性关键字，只允许出现在属性声明或索引签名或构造函数中。

class Animal {

readonly name;

public constructor(name) {

this.name = name;

}

}

let a = new Animal('Jack');

console.log(a.name); // Jack

a.name = 'Tom';

// index.ts(10,3): TS2540: Cannot assign to 'name' because it is a read-only property.

注意如果 readonly 和其他访问修饰符同时存在的话，需要写在其后面。

class Animal {

// public readonly name;

public constructor(public readonly name) {

// this.name = name;

}

}

### 抽象类

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');

// index.ts(9,11): error TS2511: Cannot create an instance of the abstract class 'Animal'.

其次，抽象类中的抽象方法必须被子类实现：

abstract class Animal {

public name;

public constructor(name) {

this.name = name;

}

public abstract sayHi();

}

class Cat extends Animal {

public eat() {

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

}

}

let cat = new Cat('Tom');

// index.ts(9,7): error TS2515: Non-abstract class 'Cat' does not implement inherited abstract member 'sayHi' from class 'Animal'.

下面是一个正确使用抽象类的例子：

abstract class Animal {

public name;

public constructor(name) {

this.name = name;

}

public abstract sayHi();

}

class Cat extends Animal {

public sayHi() {

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

}

}

let cat = new Cat('Tom');

需要注意的是，即使是抽象方法，TypeScript 的编译结果中，仍然会存在这个类，上面的代码的编译结果是：

var \_\_extends =

(this && this.\_\_extends) ||

function (d, b) {

for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];

function \_\_() {

this.constructor = d;

}

d.prototype = b === null ? Object.create(b) : ((\_\_.prototype = b.prototype), new \_\_());

};

var Animal = (function () {

function Animal(name) {

this.name = name;

}

return Animal;

})();

var Cat = (function (\_super) {

\_\_extends(Cat, \_super);

function Cat() {

\_super.apply(this, arguments);

}

Cat.prototype.sayHi = function () {

console.log('Meow, My name is ' + this.name);

};

return Cat;

})(Animal);

var cat = new Cat('Tom');

## 类的类型

给类加上 TypeScript 的类型很简单，与接口类似：

class Animal {

name: string;

constructor(name: string) {

this.name = name;

}

sayHi(): string {

return `My name is ${this.name}`;

}

}

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

console.log(a.sayHi()); // My name is Jack

# 类与接口

接口（Interfaces）可以用于对「对象的形状（Shape）」进行描述。介绍接口的另一个用途，对类的一部分行为进行抽象。

## 类实现接口

实现（implements）是面向对象中的一个重要概念。一般来讲，一个类只能继承自另一个类，有时候不同类之间可以有一些共有的特性，这时候就可以把特性提取成接口（interfaces），用 implements 关键字来实现。这个特性大大提高了面向对象的灵活性。

interface Alarm {

alert(): void;

}

class Door {

}

class SecurityDoor extends Door implements Alarm {

alert() {

console.log('SecurityDoor alert');

}

}

class Car implements Alarm {

alert() {

console.log('Car alert');

}

}

一个类可以实现多个接口：

interface Alarm {

alert(): void;

}

interface Light {

lightOn(): void;

lightOff(): void;

}

class Car implements Alarm, Light {

alert() {

console.log('Car alert');

}

lightOn() {

console.log('Car light on');

}

lightOff() {

console.log('Car light off');

}

}

## 接口继承接口

接口与接口之间可以是继承关系：

interface Alarm {

alert(): void;

}

interface LightableAlarm extends Alarm {

lightOn(): void;

lightOff(): void;

}

LightableAlarm 继承了 Alarm，除了拥有 alert 方法之外，还拥有两个新方法 lightOn 和 lightOff。

## 接口继承类

常见的面向对象语言中，接口是不能继承类的，但是在 TypeScript 中却是可以的：

class Point {

x: number;

y: number;

constructor(x: number, y: number) {

this.x = x;

this.y = y;

}

}

interface Point3d extends Point {

z: number;

}

let point3d: Point3d = {x: 1, y: 2, z: 3};

实际上，当我们在声明 class Point 时，除了会创建一个名为 Point 的类之外，同时也创建了一个名为 Point 的类型（实例的类型）。所以我们既可以将 Point 当做一个类来用（使用 new Point 创建它的实例）：

class Point {

x: number;

y: number;

constructor(x: number, y: number) {

this.x = x;

this.y = y;

}

}

const p = new Point(1, 2);

也可以将 Point 当做一个类型来用（使用 : Point 表示参数的类型）：

class Point {

x: number;

y: number;

constructor(x: number, y: number) {

this.x = x;

this.y = y;

}

}

function printPoint(p: Point) {

console.log(p.x, p.y);

}

printPoint(new Point(1, 2));

# 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。

function createArray<T>(length: number, value: T): Array<T> {

let result: T[] = [];

for (let i = 0; i < length; i++) {

result[i] = value;

}

return result;

}

createArray<string>(3, 'x'); // ['x', 'x', 'x']

上例中，我们在函数名后添加了 <T>，其中 T 用来指代任意输入的类型，在后面的输入 value: T 和输出 Array<T> 中即可使用了。接着在调用的时候，可以指定它具体的类型为 string。当然，也可以不手动指定，而让类型推论自动推算出来：

function createArray<T>(length: number, value: T): Array<T> {

let result: T[] = [];

for (let i = 0; i < length; i++) {

result[i] = value;

}

return result;

}

createArray(3, 'x'); // ['x', 'x', 'x']

## 多个类型参数

定义泛型的时候，可以一次定义多个类型参数：

function swap<T, U>(tuple: [T, U]): [U, T] {

return [tuple[1], tuple[0]];

}

swap([7, 'seven']); // ['seven', 7]

上例中，我们定义了一个 swap 函数，用来交换输入的元组。

## 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：

interface Lengthwise {

length: number;

}

function loggingIdentity<T extends Lengthwise>(arg: T): T {

console.log(arg.length);

return arg;

}

上例中，我们使用了 extends 约束了泛型 T 必须符合接口 Lengthwise 的形状，也就是必须包含 length 属性。此时如果调用 loggingIdentity 的时候，传入的 arg 不包含 length，那么在编译阶段就会报错了：

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'.

多个类型参数之间也可以互相约束：

function copyFields<T extends U, U>(target: T, source: U): T {

for (let id in source) {

target[id] = (<T>source)[id];

}

return target;

}

let x = { a: 1, b: 2, c: 3, d: 4 };

copyFields(x, { b: 10, d: 20 });

上例中，我们使用了两个类型参数，其中要求 T 继承 U，这样就保证了 U 上不会出现 T 中不存在的字段。

## 泛型接口

可以使用接口的方式来定义一个函数需要符合的形状：

interface SearchFunc {

(source: string, subString: string): boolean;

}

let mySearch: SearchFunc;

mySearch = function(source: string, subString: string) {

return source.search(subString) !== -1;

}

当然也可以使用含有泛型的接口来定义函数的形状：

interface CreateArrayFunc {

<T>(length: number, value: T): Array<T>;

}

let createArray: CreateArrayFunc;

createArray = function<T>(length: number, value: T): Array<T> {

let result: T[] = [];

for (let i = 0; i < length; i++) {

result[i] = value;

}

return result;

}

createArray(3, 'x'); // ['x', 'x', 'x']

进一步，可以把泛型参数提前到接口名上：

interface CreateArrayFunc<T> {

(length: number, value: T): Array<T>;

}

let createArray: CreateArrayFunc<any>;

createArray = function<T>(length: number, value: T): Array<T> {

let result: T[] = [];

for (let i = 0; i < length; i++) {

result[i] = value;

}

return result;

}

createArray(3, 'x'); // ['x', 'x', 'x']

注意，此时在使用泛型接口的时候，需要定义泛型的类型。

## 泛型类

与泛型接口类似，泛型也可以用于类的类型定义中：

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; };

## 泛型参数的默认类型

在TypeScript 2.3以后，我们可以为泛型中的类型参数指定默认类型。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;

}

# 声明合并

如果定义了两个相同名字的函数、接口或类，那么它们会合并成一个类型。

## 函数的合并

之前学习过，我们可以使用重载定义多个函数类型：

function reverse(x: number): number;

function reverse(x: string): string;

function reverse(x: number | string): number | string {

if (typeof x === 'number') {

return Number(x.toString().split('').reverse().join(''));

} else if (typeof x === 'string') {

return x.split('').reverse().join('');

}

}

## 接口的合并

接口中的属性在合并时会简单的合并到一个接口中：

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;

}

// index.ts(5,3): error TS2403: Subsequent variable declarations must have the same type. Variable 'price' must be of type 'number', but here has type 'string'.

接口中方法的合并，与函数的合并一样：

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

类的合并与接口的合并规则一致。

# JSX

JSX是一种嵌入式的类似XML的语法。 它可以被转换成合法的JavaScript，尽管转换的语义是依据不同的实现而定的。 JSX因React框架而流行，但也存在其它的实现。 TypeScript支持内嵌，类型检查以及将JSX直接编译为JavaScript。

## 基本用法

想要使用JSX必须做两件事：

* 给文件一个.tsx扩展名
* 启用jsx选项

TypeScript具有三种JSX模式：preserve，react和react-native。 这些模式只在代码生成阶段起作用 - 类型检查并不受影响。 在preserve模式下生成代码中会保留JSX以供后续的转换操作使用（比如：Babel）。 另外，输出文件会带有.jsx扩展名。 react模式会生成React.createElement，在使用前不需要再进行转换操作了，输出文件的扩展名为.js。 react-native相当于preserve，它也保留了所有的JSX，但是输出文件的扩展名是.js。

| **模式** | **输入** | **输出** | **输出文件扩展名** |
| --- | --- | --- | --- |
| preserve | <div /> | <div /> | .jsx |
| react | <div /> | React.createElement("div") | .js |
| react-native | <div /> | <div /> | .js |

你可以通过在命令行里使用--jsx标记或tsconfig.json里的选项来指定模式。

## as操作符

类型断言：

var foo = <foo>bar;

这里断言bar变量是foo类型的。 因为TypeScript也使用尖括号来表示类型断言，在结合JSX的语法后将带来解析上的困难。因此，TypeScript在.tsx文件里禁用了使用尖括号的类型断言。

由于不能够在.tsx文件里使用上述语法，因此我们应该使用另一个类型断言操作符：as。 上面的例子可以很容易地使用as操作符改写：

var foo = bar as foo;

as操作符在.ts和.tsx里都可用，并且与尖括号类型断言行为是等价的。

## 类型检查

为了理解JSX的类型检查，你必须首先理解固有元素与基于值的元素之间的区别。 假设有这样一个JSX表达式<expr />，expr可能引用环境自带的某些东西（比如，在DOM环境里的div或span）或者是你自定义的组件。 这是非常重要的，原因有如下两点：

* 对于React，固有元素会生成字符串（React.createElement("div")），然而由你自定义的组件却不会生成（React.createElement(MyComponent)）。
* 传入JSX元素里的属性类型的查找方式不同。 固有元素属性本身就支持，然而自定义的组件会自己去指定它们具有哪个属性。

TypeScript使用与React相同的规范 来区别它们。 固有元素总是以一个小写字母开头，基于值的元素总是以一个大写字母开头。

### 固有元素

固有元素使用特殊的接口JSX.IntrinsicElements来查找。 默认地，如果这个接口没有指定，会全部通过，不对固有元素进行类型检查。 然而，如果这个接口存在，那么固有元素的名字需要在JSX.IntrinsicElements接口的属性里查找。 例如：

declare namespace JSX {

interface IntrinsicElements {

foo: any

}

}

<foo />; // 正确

<bar />; // 错误

在上例中，<foo />没有问题，但是<bar />会报错，因为它没在JSX.IntrinsicElements里指定。也可以在JSX.IntrinsicElements上指定一个用来捕获所有字符串索引：

declare namespace JSX {

interface IntrinsicElements {

[elemName: string]: any;

}

}

### 基于值的元素

基于值的元素会简单的在它所在的作用域里按标识符查找。

import MyComponent from "./myComponent";

<MyComponent />; // 正确

<SomeOtherComponent />; // 错误

有两种方式可以定义基于值的元素：

* 无状态函数组件 (SFC)
* 类组件

由于这两种基于值的元素在JSX表达式里无法区分，因此TypeScript首先会尝试将表达式做为无状态函数组件进行解析。如果解析成功，那么TypeScript就完成了表达式到其声明的解析操作。如果按照无状态函数组件解析失败，那么TypeScript会继续尝试以类组件的形式进行解析。如果依旧失败，那么将输出一个错误。

#### 无状态函数组件

正如其名，组件被定义成JavaScript函数，它的第一个参数是props对象。 TypeScript会强制它的返回值可以赋值给JSX.Element。

interface FooProp {

name: string;

X: number;

Y: number;

}

declare function AnotherComponent(prop: {name: string});

function ComponentFoo(prop: FooProp) {

return <AnotherComponent name={prop.name} />;

}

const Button = (prop: {value: string}, context: { color: string }) => <button>

由于无状态函数组件是简单的JavaScript函数，所以我们还可以利用函数重载。

interface ClickableProps {

children: JSX.Element[] | JSX.Element

}

interface HomeProps extends ClickableProps {

home: JSX.Element;

}

interface SideProps extends ClickableProps {

side: JSX.Element | string;

}

function MainButton(prop: HomeProps): JSX.Element;

function MainButton(prop: SideProps): JSX.Element {

...

}

#### 类组件

我们可以定义类组件的类型。 然而，我们首先最好弄懂两个新的术语：元素类的类型和元素实例的类型。现在有<Expr />，元素类的类型为Expr的类型。 所以在上面的例子里，如果MyComponent是ES6的类，那么类类型就是类的构造函数和静态部分。 如果MyComponent是个工厂函数，类类型为这个函数。

一旦建立起了类类型，实例类型由类构造器或调用签名（如果存在的话）的返回值的联合构成。 再次说明，在ES6类的情况下，实例类型为这个类的实例的类型，并且如果是工厂函数，实例类型为这个函数返回值类型。

class MyComponent {

render() {}

}

// 使用构造签名

var myComponent = new MyComponent();

// 元素类的类型 => MyComponent

// 元素实例的类型 => { render: () => void }

function MyFactoryFunction() {

return {

render: () => {

}

}

}

// 使用调用签名

var myComponent = MyFactoryFunction();

// 元素类的类型 => FactoryFunction

// 元素实例的类型 => { render: () => void }

元素的实例类型必须赋值给JSX.ElementClass或抛出一个错误。 默认的JSX.ElementClass为{}，但是它可以被扩展用来限制JSX的类型以符合相应的接口。

#### 属性类型检查

属性类型检查的第一步是确定元素属性类型。 这在固有元素和基于值的元素之间稍有不同。对于固有元素，这是JSX.IntrinsicElements属性的类型。

declare namespace JSX {

interface IntrinsicElements {

foo: { bar?: boolean }

}

}

// `foo`的元素属性类型为`{bar?: boolean}`

<foo bar />;

对于基于值的元素，就稍微复杂些。 它取决于先前确定的在元素实例类型上的某个属性的类型。 至于该使用哪个属性来确定类型取决于JSX.ElementAttributesProperty。 它应该使用单一的属性来定义。 这个属性名之后会被使用。 TypeScript 2.8，如果未指定JSX.ElementAttributesProperty，那么将使用类元素构造函数或SFC调用的第一个参数的类型。

declare namespace JSX {

interface ElementAttributesProperty {

props; // 指定用来使用的属性名

}

}

class MyComponent {

// 在元素实例类型上指定属性

props: {

foo?: string;

}

}

// `MyComponent`的元素属性类型为`{foo?: string}`

<MyComponent foo="bar" />

元素属性类型用于的JSX里进行属性的类型检查。 支持可选属性和必须属性。

declare namespace JSX {

interface IntrinsicElements {

foo: { requiredProp: string; optionalProp?: number }

}

}

<foo requiredProp="bar" />; // 正确

<foo requiredProp="bar" optionalProp={0} />; // 正确

<foo />; // 错误, 缺少 requiredProp

<foo requiredProp={0} />; // 错误, requiredProp 应该是字符串

<foo requiredProp="bar" unknownProp />; // 错误, unknownProp 不存在

<foo requiredProp="bar" some-unknown-prop />; // 正确, `some-unknown-prop`不是个合法的标识符

另外，JSX还会使用JSX.IntrinsicAttributes接口来指定额外的属性，这些额外的属性通常不会被组件的props或arguments使用 - 比如React里的key。还有，JSX.IntrinsicClassAttributes<T>泛型类型也可以用来做同样的事情。这里的泛型参数表示类实例类型。在React里，它用来允许Ref<T>类型上的ref属性。通常来讲，这些接口上的所有属性都是可选的，除非你想要用户在每个JSX标签上都提供一些属性。延展操作符也可以使用：

var props = { requiredProp: 'bar' };

<foo {...props} />; // 正确

var badProps = {};

<foo {...badProps} />; // 错误

#### 子孙类型检查

从TypeScript 2.3开始，我们引入了children类型检查。children是元素属性(attribute)类型的一个特殊属性(property)，子JSXExpression将会被插入到属性里。 与使用JSX.ElementAttributesProperty来决定props名类似，我们可以利用JSX.ElementChildrenAttribute来决定children名。 JSX.ElementChildrenAttribute应该被声明在单一的属性(property)里。

declare namespace JSX {

interface ElementChildrenAttribute {

children: {}; // specify children name to use

}

}

如不特殊指定子孙的类型，我们将使用React typings里的默认类型。

<div>

<h1>Hello</h1>

</div>;

<div>

<h1>Hello</h1>

World

</div>;

const CustomComp = (props) => <div>props.children</div>

<CustomComp>

<div>Hello World</div>

{"This is just a JS expression..." + 1000}

</CustomComp>

## JSX结果类型

默认地JSX表达式结果的类型为any。 你可以自定义这个类型，通过指定JSX.Element接口。 然而，不能够从接口里检索元素，属性或JSX的子元素的类型信息。 它是一个黑盒。

## 嵌入的表达式

JSX允许你使用{ }标签来内嵌表达式。

var a = <div>

{['foo', 'bar'].map(i => <span>{i / 2}</span>)}

</div>

上面的代码产生一个错误，因为你不能用数字来除以一个字符串。 输出如下，若你使用了preserve选项：

var a = <div>

{['foo', 'bar'].map(function (i) { return <span>{i / 2}</span>; })}

</div>

## 工厂函数

jsx: react编译选项使用的工厂函数是可以配置的。可以使用jsxFactory命令行选项，或内联的@jsx注释指令在每个文件上设置。比如，给createElement设置jsxFactory，<div />会使用createElement("div")来生成，而不是React.createElement("div")。注释指令可以像下面这样使用（在TypeScript 2.8里）：

import preact = require("preact");

/\* @jsx preact.h \*/

const x = <div />;

生成：

const preact = require("preact");

const x = preact.h("div", null);

工厂函数的选择同样会影响JSX命名空间的查找（类型检查）。如果工厂函数使用React.createElement定义（默认），编译器会先检查React.JSX，之后才检查全局的JSX。如果工厂函数定义为h，那么在检查全局的JSX之前先检查h.JSX。

# 装饰器

随着TypeScript和ES6里引入了类，在一些场景下我们需要额外的特性来支持标注或修改类及其成员。 装饰器（Decorators）为我们在类的声明及成员上通过元编程语法添加标注提供了一种方式。装饰器是一项实验性特性，在未来的版本中可能会发生改变。若要启用实验性的装饰器特性，你必须在命令行或tsconfig.json里启用experimentalDecorators编译器选项：

* 命令行: tsc --target ES5 –experimentalDecorators
* tsconfig.json:

{

"compilerOptions": {

"target": "ES5",

"experimentalDecorators": true

}

}

装饰器是一种特殊类型的声明，它能够被附加到类声明，方法， 访问符，属性或参数上。 装饰器使用 @expression这种形式，expression求值后必须为一个函数，它会在运行时被调用，被装饰的声明信息做为参数传入。例如，有一个@sealed装饰器，我们会这样定义sealed函数：

function sealed(target) {

// do something with "target" ...

}

## 装饰器工厂

要定制一个修饰器如何应用到一个声明上，我们得写一个装饰器工厂函数。 装饰器工厂就是一个简单的函数，它返回一个表达式，以供装饰器在运行时调用。我们可以通过下面的方式来写一个装饰器工厂函数：

function color(value: string) { // 这是一个装饰器工厂

return function (target) { // 这是装饰器

// do something with "target" and "value"...

}

}

## 装饰器组合

多个装饰器可以同时应用到一个声明上，就像下面的示例：

* 书写在同一行上：

@f @g x

* 书写在多行上：

@f

@g

x

当多个装饰器应用于一个声明上，它们求值方式与复合函数相似。在这个模型下，当复合f和g时，复合的结果(f ∘ g)(x)等同于f(g(x))。同样的，在TypeScript里，当多个装饰器应用在一个声明上时会进行如下步骤的操作：

1. 由上至下依次对装饰器表达式求值。
2. 求值的结果会被当作函数，由下至上依次调用。

## 装饰器求值

类中不同声明上的装饰器将按以下规定的顺序应用：

* 参数装饰器，然后依次是方法装饰器，访问符装饰器，或属性装饰器应用到每个实例成员。
* 参数装饰器，然后依次是方法装饰器，访问符装饰器，或属性装饰器应用到每个静态成员。
* 参数装饰器应用到构造函数。
* 类装饰器应用到类。

## 类装饰

类装饰器在类声明之前被声明（紧靠着类声明）。 类装饰器应用于类构造函数，可以用来监视，修改或替换类定义。 类装饰器不能用在声明文件中( .d.ts)，也不能用在任何外部上下文中（比如declare的类）。类装饰器表达式会在运行时当作函数被调用，类的构造函数作为其唯一的参数。如果类装饰器返回一个值，它会使用提供的构造函数来替换类的声明。下面是使用类装饰器(@sealed)的例子，应用在Greeter类：

@sealed

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {

return "Hello, " + this.greeting;

}

}

我们可以这样定义@sealed装饰器：

function sealed(constructor: Function) {

Object.seal(constructor);

Object.seal(constructor.prototype);

}

当@sealed被执行的时候，它将密封此类的构造函数和原型。下面是一个重载构造函数的例子。

function classDecorator<T extends {new(...args:any[]):{}}>(constructor:T) {

return class extends constructor {

newProperty = "new property";

hello = "override";

}

}

@classDecorator

class Greeter {

property = "property";

hello: string;

constructor(m: string) {

this.hello = m;

}

}

console.log(new Greeter("world"));

## 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）。 它会被应用到方法的 属性描述符上，可以用来监视，修改或者替换方法定义。 方法装饰器不能用在声明文件( .d.ts)，重载或者任何外部上下文（比如declare的类）中。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：

* 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。
* 成员的名字。
* 成员的属性描述符。
* 注意：
  + 如果代码输出目标版本小于ES5，属性描述符将会是undefined。
  + 如果代码输出目标版本小于ES5返回值会被忽略。

下面是一个方法装饰器（@enumerable）的例子，应用于Greeter类的方法上：

function enumerable(value: boolean) {

return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {

descriptor.enumerable = value;

};

}

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

@enumerable(false)

greet() {

return "Hello, " + this.greeting;

}

}

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。

## 访问器装饰器

访问器装饰器声明在一个访问器的声明之前（紧靠着访问器声明）。 访问器装饰器应用于访问器的 属性描述符并且可以用来监视，修改或替换一个访问器的定义。 访问器装饰器不能用在声明文件中（.d.ts），或者任何外部上下文（比如 declare的类）里。

TypeScript不允许同时装饰一个成员的get和set访问器。取而代之的是，一个成员的所有装饰的必须应用在文档顺序的第一个访问器上。这是因为，在装饰器应用于一个属性描述符时，它联合了get和set访问器，而不是分开声明的。

访问器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：

* 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。
* 成员的名字。
* 成员的属性描述符。
* 如果代码输出目标版本小于ES5，Property Descriptor将会是undefined。
* 如果代码输出目标版本小于ES5返回值会被忽略。

下面是使用了访问器装饰器（@configurable）的例子，应用于Point类的成员上：

function configurable(value: boolean) {

return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {

descriptor.configurable = value;

};

}

class Point {

private \_x: number;

private \_y: number;

constructor(x: number, y: number) {

this.\_x = x;

this.\_y = y;

}

@configurable(false)

get x() { return this.\_x; }

@configurable(false)

get y() { return this.\_y; }

}

## 属性装饰器

属性装饰器声明在一个属性声明之前（紧靠着属性声明）。 属性装饰器不能用在声明文件中（.d.ts），或者任何外部上下文（比如 declare的类）里。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：

* 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。
* 成员的名字。

属性描述符不会做为参数传入属性装饰器，这与TypeScript是如何初始化属性装饰器的有关。 因为目前没有办法在定义一个原型对象的成员时描述一个实例属性，并且没办法监视或修改一个属性的初始化方法。返回值也会被忽略。因此，属性描述符只能用来监视类中是否声明了某个名字的属性。

我们可以用它来记录这个属性的元数据，如下例所示：

import "reflect-metadata";

const formatMetadataKey = Symbol("format");

function format(formatString: string) {

return Reflect.metadata(formatMetadataKey, formatString);

}

function getFormat(target: any, propertyKey: string) {

return Reflect.getMetadata(formatMetadataKey, target, propertyKey);

}

class Greeter {

@format("Hello, %s")

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {

let formatString = getFormat(this, "greeting");

return formatString.replace("%s", this.greeting);

}

}

这个@format("Hello, %s")装饰器是个 装饰器工厂。 当 @format("Hello, %s")被调用时，它添加一条这个属性的元数据，通过reflect-metadata库里的Reflect.metadata函数。 当 getFormat被调用时，它读取格式的元数据。

## 参数装饰器

参数装饰器声明在一个参数声明之前（紧靠着参数声明）。 参数装饰器应用于类构造函数或方法声明。 参数装饰器不能用在声明文件（.d.ts），重载或其它外部上下文（比如 declare的类）里。参数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：

* 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。
* 成员的名字。
* 参数在函数参数列表中的索引。
* 参数装饰器只能用来监视一个方法的参数是否被传入。

下例定义了参数装饰器（@required）并应用于Greeter类方法的一个参数：

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

@validate

greet(@required name: string) {

return "Hello " + name + ", " + this.greeting;

}

}

然后我们使用下面的函数定义 @required 和 @validate 装饰器：

import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol, parameterIndex: number) {

let existingRequiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];

existingRequiredParameters.push(parameterIndex);

Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, target, propertyKey);

}

function validate(target: any, propertyName: string, descriptor: TypedPropertyDescriptor<Function>) {

let method = descriptor.value;

descriptor.value = function () {

let requiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target, propertyName);

if (requiredParameters) {

for (let parameterIndex of requiredParameters) {

if (parameterIndex >= arguments.length || arguments[parameterIndex] === undefined) {

throw new Error("Missing required argument.");

}

}

}

return method.apply(this, arguments);

}

}