# Typescript

阮一峰讲解文档

<https://ts.xcatliu.com/basics/declaration-files.html#zai-npm-bao-huo-umd-ku-zhong-kuo-zhan-quan-ju-bian-liang>

## 基础

### 原始数据类型

JavaScript 的类型分为两种：原始数据类型（[Primitive data types](https://developer.mozilla.org/en-US/docs/Glossary/Primitive)）和对象类型（Object types）。

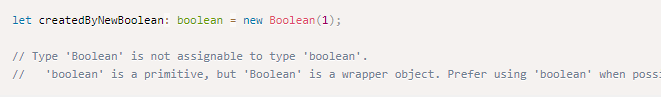
原始数据类型包括：布尔值、数值、字符串、null、undefined 以及 [ES6 中的新类型 Symbol](http://es6.ruanyifeng.com/#docs/symbol)。

#### 布尔值

布尔值是最基础的数据类型，在 TypeScript 中，使用 boolean 定义布尔值类型：



**注意**：使用构造函数 Boolean 创造的对象**不是**布尔值、而是对象





拓展

在 TypeScript 中，boolean 是 JavaScript 中的基本类型，而 Boolean 是 JavaScript 中的构造函数。其他基本类型（除了 null 和 undefined）一样，不再赘述。

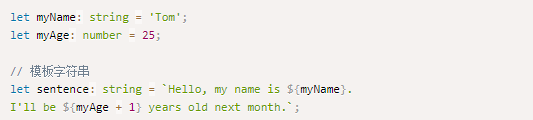
#### 数值

使用 number 定义数值类型



#### 字符串

使用 string 定义字符串类型



**注意**：其中 ` 用来定义 [ES6 中的模板字符串](http://es6.ruanyifeng.com/#docs/string#%E6%A8%A1%E6%9D%BF%E5%AD%97%E7%AC%A6%E4%B8%B2)，${expr} 用来在模板字符串中嵌入表达式。

#### 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：

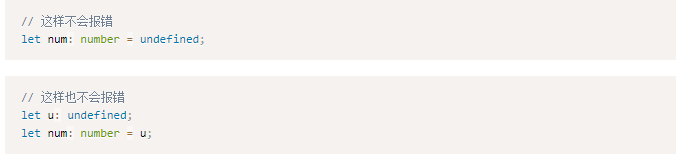


#### null和undefined

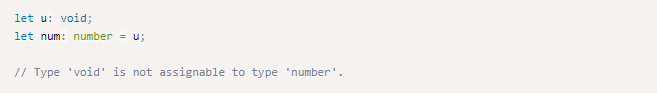
在 TypeScript 中，可以使用 null 和 undefined 来定义这两个原始数据类型



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：



而 void 类型的变量不能赋值给 number 类型的变量：

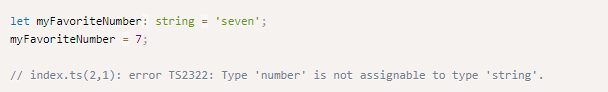


### 任意值

任意值（Any）用来表示允许赋值为任意类型。

#### 什么是任意值类型

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

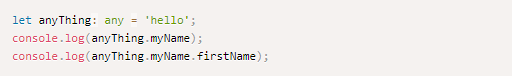


但如果是 any 类型，则允许被赋值为任意类型



#### 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



也允许调用任何方法



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**：

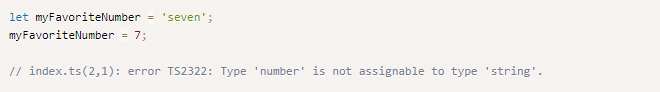


### 类型推论

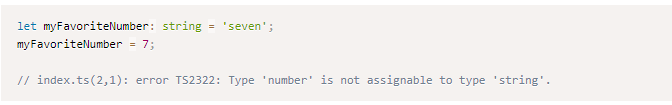
如果没有明确的指定类型，那么 TypeScript 会依照类型推论（Type Inference）的规则推断出一个类型。

#### 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：

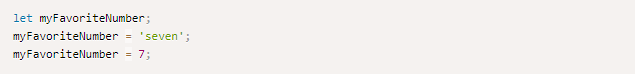


事实上，它等价于：



**结论**：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。

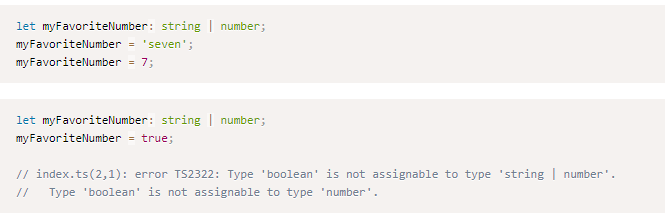
**如果定义的时候没有赋值，不管之后有没有赋值，都会被推断成 any 类型而完全不被类型检查**



### 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：

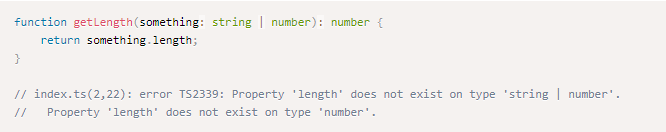


**联合类型使用 | 分隔每个类型**

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。

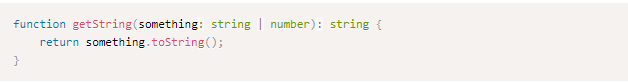
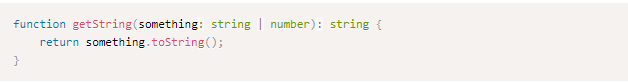
#### 访问联合类型的属性或方法

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**：

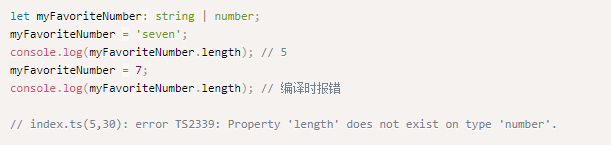


上例中，length 不是 string 和 number 的共有属性，所以会报错。

访问 string 和 number 的共有属性是没问题的：



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



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

### 对象数据类型--接口

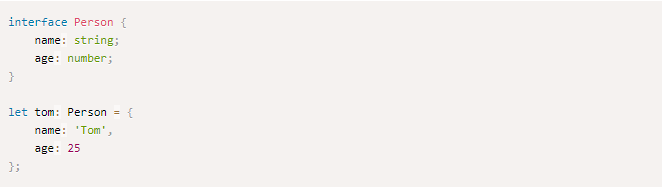
在 TypeScript 中，我们使用接口（Interfaces）来定义对象的类型。

#### 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）。

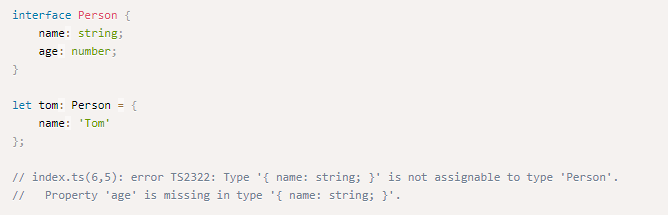
TypeScript 中的接口是一个非常灵活的概念，除了可用于[对类的一部分行为进行抽象](https://ts.xcatliu.com/advanced/class-and-interfaces.html" \l "%E7%B1%BB%E5%AE%9E%E7%8E%B0%E6%8E%A5%E5%8F%A3)以外，也常用于对「对象的形状（Shape）」进行描述。

**例：**

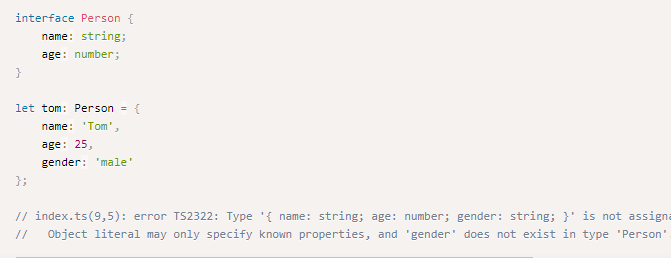


上面的例子中，我们定义了一个接口 Person，接着定义了一个变量 tom，它的类型是 Person。这样，我们就约束了 tom 的形状必须和接口 Person 一致。接口一般首字母大写

定义的变量比接口少了一些属性是不允许的

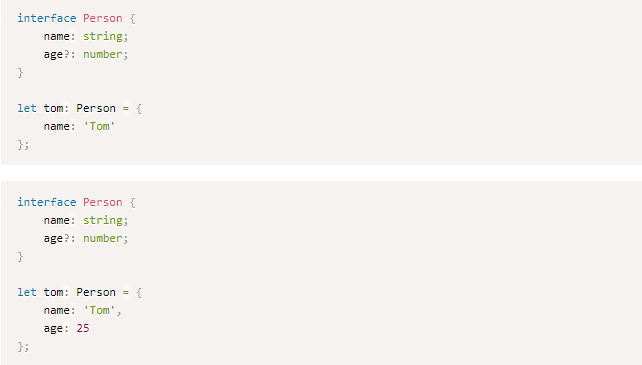


多一些属性也是不允许的：



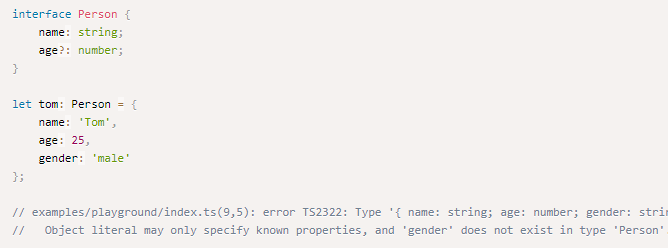
可见，**赋值的时候，变量的形状必须和接口的形状保持一致**。

#### 可选属性



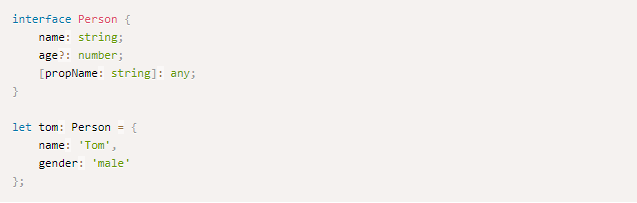
可选属性的含义是该属性可以不存在。

这时**仍然不允许添加未定义的属性**



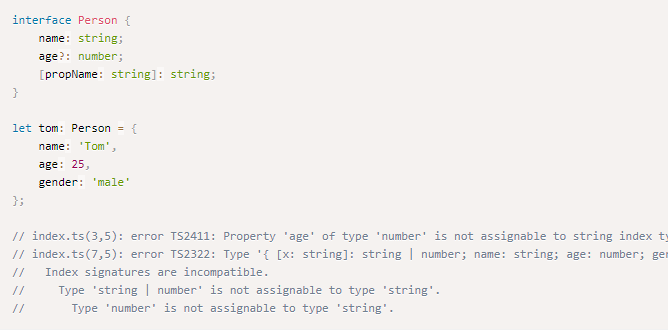
#### 任意属性

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



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

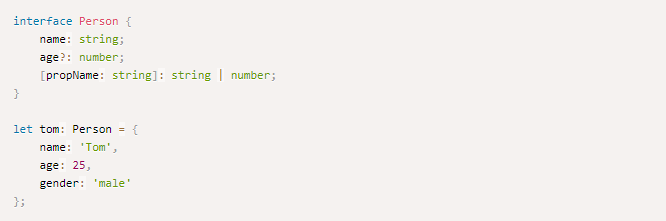
需要注意的是，**一旦定义了任意属性，那么确定属性和可选属性的类型都必须是它的类型的子集**



上例中，任意属性的值允许是 string，但是可选属性 age 的值却是 number，number 不是 string 的子属性，所以报错了。

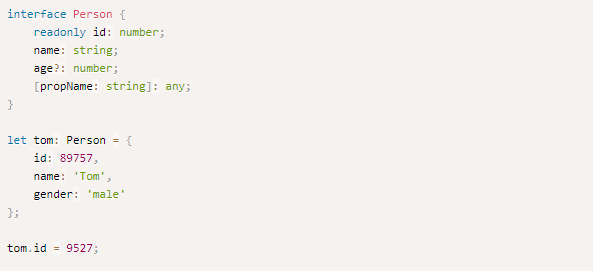
另外，在报错信息中可以看出，此时 { name: 'Tom', age: 25, gender: 'male' } 的类型被推断成了 { [x: string]: string | number; name: string; age: number; gender: string; }，这是联合类型和接口的结合。

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：



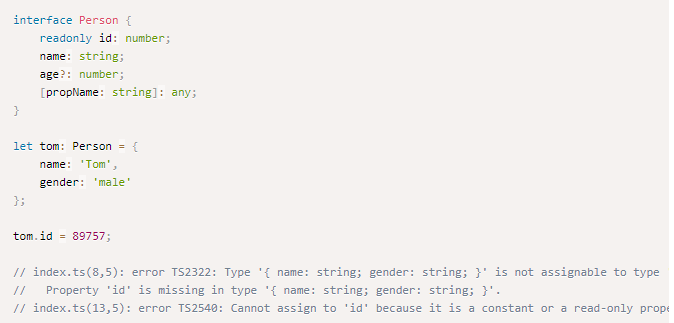
#### 只读属性

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：



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**



上例中，报错信息有两处，第一处是在对 tom 进行赋值的时候，没有给 id 赋值。

第二处是在给 tom.id 赋值的时候，由于它是只读属性，所以报错了。

### ****数组的类型****

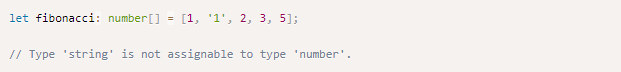
在 TypeScript 中，数组类型有多种定义方式，比较灵活。

#### 「类型 + 方括号」表示法

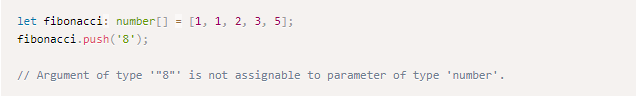
最简单的方法是使用「类型 + 方括号」来表示数组：



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



上例中，push 方法只允许传入 number 类型的参数，但是却传了一个 "8" 类型的参数，所以报错了。这里 "8" 是一个字符串字面量类型，会在后续章节中详细介绍。

#### 数组泛型

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

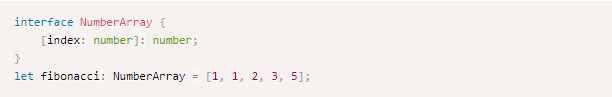


泛型

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。

#### 用接口表示数组

接口也可以用来描述数组



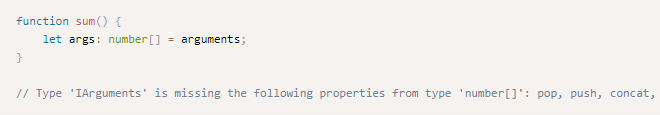
NumberArray 表示：只要索引的类型是数字时，那么值的类型必须是数字。

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。

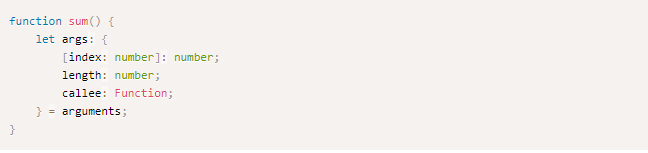
不过有一种情况例外，那就是它常用来表示类数组。

#### 类数组

类数组（Array-like Object）不是数组类型，比如 arguments：

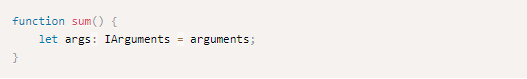


上例中，arguments 实际上是一个类数组，不能用普通的数组的方式来描述，而应该用接口：



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：



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



内置对象

JavaScript 中有很多[内置对象](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)，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 标准提供的内置对象有：**

Boolean、Error、Date、RegExp 等

#### 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

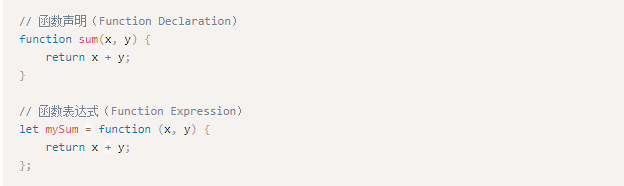


### 函数的类型

[函数是 JavaScript 中的一等公民](https://llh911001.gitbooks.io/mostly-adequate-guide-chinese/content/ch2.html)

#### 函数声明

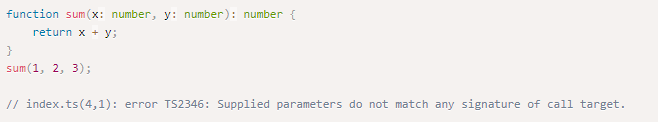
在 JavaScript 中，有两种常见的定义函数的方式——函数声明（Function Declaration）和函数表达式（Function Expression）：

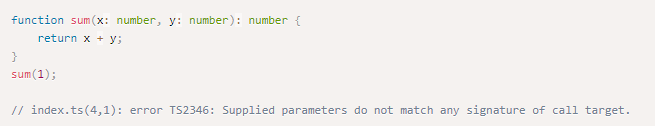


一个函数有输入和输出，要在 TypeScript 中对其进行约束，需要把输入和输出都考虑到，其中函数声明的类型定义较简单



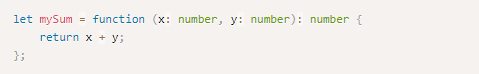
注意，**输入多余的（或者少于要求的）参数，是不被允许的**：



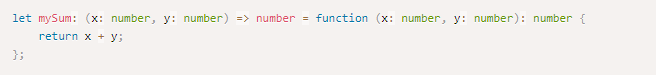


#### 函数表达式

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：



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：

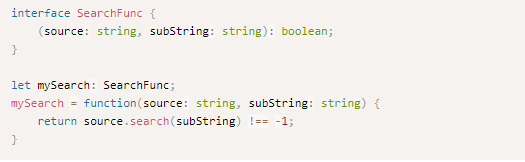


**注意：**不要混淆了 TypeScript 中的 => 和 ES6 中的 =>。

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，可以参考 [ES6 中的箭头函数](http://es6.ruanyifeng.com/#docs/function#%E7%AE%AD%E5%A4%B4%E5%87%BD%E6%95%B0)。

#### 用接口定义函数的形状

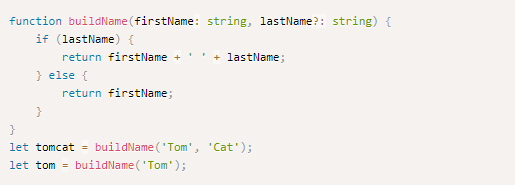


采用函数表达式|接口定义函数的方式时，对等号左侧进行类型限制，可以保证以后对函数名赋值时保证参数个数、参数类型、返回值类型不变。

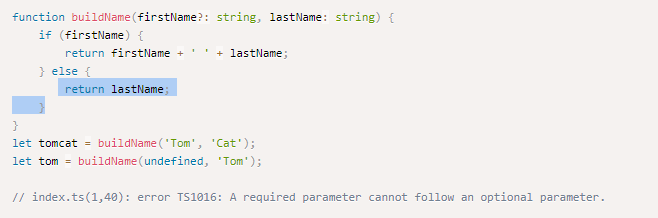
#### 可选参数

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：

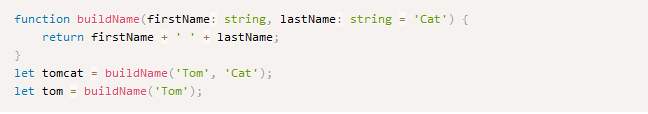


需要注意的是，可选参数必须接在必需参数后面。换句话说，**可选参数后面不允许再出现必需参数了**：

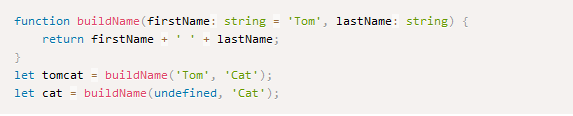


#### 参数默认值

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**：



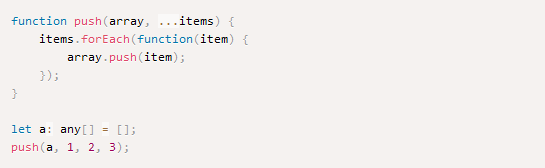
此时就不受「可选参数必须接在必需参数后面」的限制了：



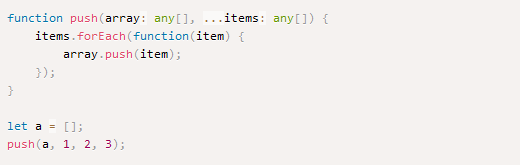
关于默认参数，可以参考 [ES6 中函数参数的默认值](http://es6.ruanyifeng.com/#docs/function#%E5%87%BD%E6%95%B0%E5%8F%82%E6%95%B0%E7%9A%84%E9%BB%98%E8%AE%A4%E5%80%BC)。

#### 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）：



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：



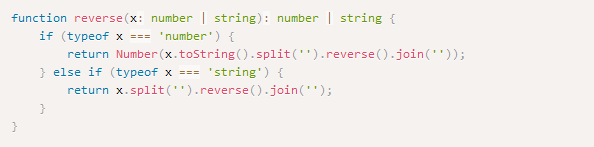
注意，rest 参数只能是最后一个参数，关于 rest 参数，可以参考 [ES6 中的 rest 参数](http://es6.ruanyifeng.com/#docs/function#rest%E5%8F%82%E6%95%B0)。

#### 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。

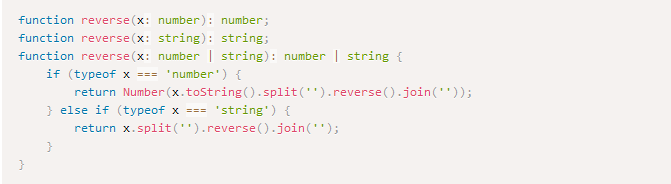
比如，我们需要实现一个函数 reverse，输入数字 123 的时候，输出反转的数字 321，输入字符串 'hello' 的时候，输出反转的字符串 'olleh'。

利用联合类型，我们可以这么实现：



**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：



上例中，我们重复定义了多次函数 reverse，前几次都是函数定义，最后一次是函数实现。在编辑器的代码提示中，可以正确的看到前两个提示。

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



或者



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](https://ts.xcatliu.com/advanced/generics.html)。

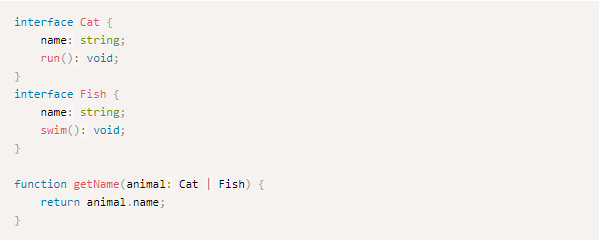
故建议大家在使用类型断言时，统一使用 值 as 类型 这样的语法，本书中也会贯彻这一思想。

#### 类型断言的用途

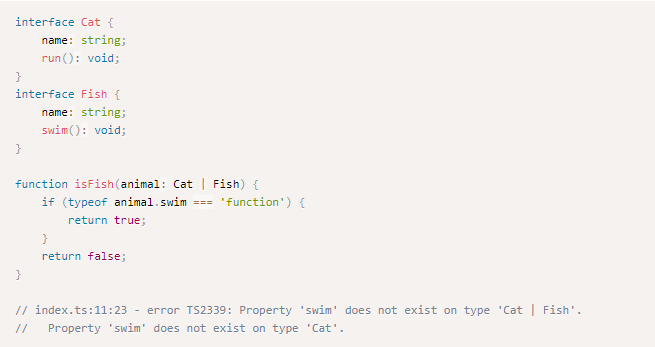
类型断言的常见用途有以下几种：

##### 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**



而有时候，我们确实需要在还不确定类型的时候就访问其中一个类型特有的属性或方法，比如：



上面的例子中，获取 animal.swim 的时候会报错。

此时可以使用类型断言，将 animal 断言成 Fish：



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：



上面的例子编译时不会报错，但在运行时会报错：

Uncaught TypeError: animal.swim is not a function`

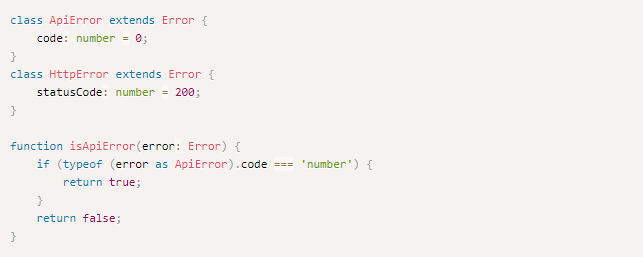
原因是 (animal as Fish).swim() 这段代码隐藏了 animal 可能为 Cat 的情况，将 animal 直接断言为 Fish 了，而 TypeScript 编译器信任了我们的断言，故在调用 swim() 时没有编译错误。

可是 swim 函数接受的参数是 Cat | Fish，一旦传入的参数是 Cat 类型的变量，由于 Cat 上没有 swim 方法，就会导致运行时错误了。

总之，使用类型断言时一定要格外小心，尽量避免断言后调用方法或引用深层属性，以减少不必要的运行时错误。

##### 将一个父类断言为更加具体的子类

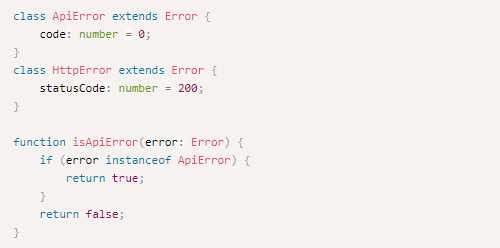
当类之间有继承关系时，类型断言也是很常见的：



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。

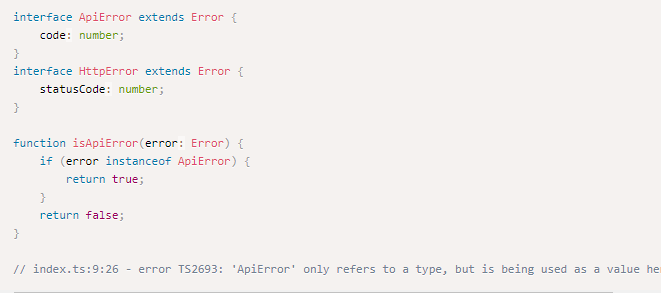
但是由于父类 Error 中没有 code 属性，故直接获取 error.code 会报错，需要使用类型断言获取 (error as ApiError).code。

大家可能会注意到，在这个例子中有一个更合适的方式来判断是不是 ApiError，那就是使用 instanceof：



上面的例子中，确实使用 instanceof 更加合适，因为 ApiError 是一个 JavaScript 的类，能够通过 instanceof 来判断 error 是否是它的实例。

但是有的情况下 ApiError 和 HttpError 不是一个真正的类，而只是一个 TypeScript 的接口（interface），接口是一个类型，不是一个真正的值，它在编译结果中会被删除，当然就无法使用 instanceof 来做运行时判断了：



此时就只能用类型断言，通过判断是否存在 code 属性，来判断传入的参数是不是 ApiError了：

interface ApiError extends Error {

code: number;

}

interface HttpError extends Error {

statusCode: number;

}

function isApiError(error: Error) {

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

return true;

}

return false;

}

##### 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：

const foo: number = 1;

foo.length = 1;

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

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：

window.foo = 1;

// index.ts:1:8 - error TS2339: Property 'foo' does not exist on type 'Window & typeof globalThis'.

上面的例子中，我们需要将 window 上添加一个属性 foo，但 TypeScript 编译时会报错，提示我们 window 上不存在 foo 属性。

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。**

上面的例子中，我们也可以通过[扩展 window 的类型（TODO）][]解决这个错误，不过如果只是临时的增加 foo 属性，as any 会更加方便。

总之，**一方面不能滥用 as any，另一方面也不要完全否定它的作用，我们需要在类型的严格性和开发的便利性之间掌握平衡**（这也是 [TypeScript 的设计理念](https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals)之一），才能发挥出 TypeScript 最大的价值。

##### 将any断言为一个具体的类型

在日常的开发中，我们不可避免的需要处理 any 类型的变量，它们可能是由于第三方库未能定义好自己的类型，也有可能是历史遗留的或其他人编写的烂代码，还可能是受到 TypeScript 类型系统的限制而无法精确定义类型的场景。

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

那么类型断言有没有什么限制呢？是不是任何一个类型都可以被断言为任何另一个类型呢？

答案是否定的——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。

下面我们通过一个简化的例子，来理解类型断言的限制：

interface Animal {

name: string;

}

interface Cat {

name: string;

run(): void;

}

let tom: Cat = {

name: 'Tom',

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

};

let animal: Animal = tom;

我们知道，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：



那么也不难理解为什么 Cat 类型的 tom 可以赋值给 Animal 类型的 animal 了——就像面向对象编程中我们可以将子类的实例赋值给类型为父类的变量。

我们把它换成 TypeScript 中更专业的说法，即：Animal 兼容 Cat。

当 Animal 兼容 Cat 时，它们就可以互相进行类型断言了：

interface Animal {

name: string;

}

interface Cat {

name: string;

run(): void;

}

function testAnimal(animal: Animal) {

return (animal as Cat);

}

function testCat(cat: Cat) {

return (cat as Animal);

}

* 允许 animal as Cat 是因为「父类可以被断言为子类」，这个前面已经学习过了
* 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」

需要注意的是，这里我们使用了简化的父类子类的关系来表达类型的兼容性，而实际上 TypeScript 在判断类型的兼容性时，比这种情况复杂很多，详细请参考[类型的兼容性（TODO)][]章节。

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。

**除非迫不得已，千万别用双重断言。**

### 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

由于本章涉及大量新语法，故在本章开头列出新语法的索引，方便大家在使用这些新语法时能快速查找到对应的讲解：

* [declare var](https://ts.xcatliu.com/basics/declaration-files.html#declare-var) 声明全局变量
* [declare function](https://ts.xcatliu.com/basics/declaration-files.html#declare-function) 声明全局方法
* [declare class](https://ts.xcatliu.com/basics/declaration-files.html#declare-class) 声明全局类
* [declare enum](https://ts.xcatliu.com/basics/declaration-files.html#declare-enum) 声明全局枚举类型
* [declare namespace](https://ts.xcatliu.com/basics/declaration-files.html#declare-namespace) 声明（含有子属性的）全局对象
* [interface 和 type](https://ts.xcatliu.com/basics/declaration-files.html#interface-he-type) 声明全局类型
* [export](https://ts.xcatliu.com/basics/declaration-files.html#export) 导出变量
* [export namespace](https://ts.xcatliu.com/basics/declaration-files.html#export-namespace) 导出（含有子属性的）对象
* [export default](https://ts.xcatliu.com/basics/declaration-files.html#export-default) ES6 默认导出
* [export =](https://ts.xcatliu.com/basics/declaration-files.html#export-1) commonjs 导出模块
* [export as namespace](https://ts.xcatliu.com/basics/declaration-files.html#export-as-namespace) UMD 库声明全局变量
* [declare global](https://ts.xcatliu.com/basics/declaration-files.html#declare-global) 扩展全局变量
* [declare module](https://ts.xcatliu.com/basics/declaration-files.html#declare-module) 扩展模块
* [/// <reference />](https://ts.xcatliu.com/basics/declaration-files.html#san-xie-xian-zhi-ling) 三斜线指令

#### 什么是声明语句

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[1](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/01-jquery)：

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](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/02-declare-var)：

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');

#### 什么是声明文件

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](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/03-jquery-d-ts)：

// 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，那么引入声明文件又是另一种方式了，将会在后面详细介绍。

#### 第三方声明文件

当然，jQuery 的声明文件不需要我们定义了，社区已经帮我们定义好了：[jQuery in DefinitelyTyped](https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types/jquery/index.d.ts)。

我们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

可以在[这个页面](https://microsoft.github.io/TypeSearch/)搜索你需要的声明文件。

#### 书写声明文件

当一个第三方库没有提供声明文件时，我们就需要自己书写声明文件了。前面只介绍了最简单的声明文件内容，而真正书写一个声明文件并不是一件简单的事，以下会详细介绍如何书写声明文件。

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](https://ts.xcatliu.com/basics/declaration-files.html#quan-ju-bian-liang)：通过 <script> 标签引入第三方库，注入全局变量
* [npm 包](https://ts.xcatliu.com/basics/declaration-files.html#npm-bao)：通过 import foo from 'foo' 导入，符合 ES6 模块规范
* [UMD 库](https://ts.xcatliu.com/basics/declaration-files.html#umd-ku)：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](https://ts.xcatliu.com/basics/declaration-files.html#zhi-jie-kuo-zhan-quan-ju-bian-liang)：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](https://ts.xcatliu.com/basics/declaration-files.html#zai-npm-bao-huo-umd-ku-zhong-kuo-zhan-quan-ju-bian-liang)：引用 npm 包或 UMD 库后，改变一个全局变量的结构
* [模块插件](https://ts.xcatliu.com/basics/declaration-files.html#mo-kuai-cha-jian)：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](https://ts.xcatliu.com/basics/declaration-files.html#declare-var) 声明全局变量
* [declare function](https://ts.xcatliu.com/basics/declaration-files.html#declare-function) 声明全局方法
* [declare class](https://ts.xcatliu.com/basics/declaration-files.html#declare-class) 声明全局类
* [declare enum](https://ts.xcatliu.com/basics/declaration-files.html#declare-enum) 声明全局枚举类型
* [declare namespace](https://ts.xcatliu.com/basics/declaration-files.html#declare-namespace) 声明（含有子属性的）全局对象
* [interface 和 type](https://ts.xcatliu.com/basics/declaration-files.html#interface-he-type) 声明全局类型

##### declare var

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[4](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/04-declare-const-jquery)：

// 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[5](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/05-declare-jquery-value)：

declare const jQuery = function(selector) {

return document.querySelector(selector);

};

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

##### Declare function

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[6](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/06-declare-function)：

// 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

当全局变量是一个类的时候，我们用 declare class 来定义它的类型[7](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/07-declare-class)：

// 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

使用 declare enum 定义的枚举类型也称作外部枚举（Ambient Enums），举例如下[8](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/08-declare-enum)：

// 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

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[9](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/09-declare-namespace)：

// 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);

###### 嵌套的命名空间

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[10](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/10-declare-namespace-nesting)：

// 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[11](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/11-declare-namespace-dot)：

// 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[12](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/12-interface)：

// 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);

type 与 interface 类似，不再赘述。

##### 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[13](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/13-avoid-name-conflict)：

// 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[14](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/14-declaration-merging)：.

// 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');

关于声明合并的更多用法，可以查看[声明合并](https://ts.xcatliu.com/advanced/declaration-merging.html)章节。

#### Npm包

（npm包管理知识：

<https://blog.csdn.net/qq_41499782/article/details/106973662>

）

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。

目录结构：

/path/to/project

├── src

| └── index.ts

├── types

| └── foo

| └── index.d.ts

└── tsconfig.json

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。

不管采用了以上两种方式中的哪一种，我都**强烈建议**大家将书写好的声明文件（通过给第三方库发 pull request，或者直接提交到 @types 里）发布到开源社区中，享受了这么多社区的优秀的资源，就应该在力所能及的时候给出一些回馈。只有所有人都参与进来，才能让 ts 社区更加繁荣。

npm 包的声明文件主要有以下几种语法：

* [export](https://ts.xcatliu.com/basics/declaration-files.html#export) 导出变量
* [export namespace](https://ts.xcatliu.com/basics/declaration-files.html#export-namespace) 导出（含有子属性的）对象
* [export default](https://ts.xcatliu.com/basics/declaration-files.html#export-default) ES6 默认导出
* [export =](https://ts.xcatliu.com/basics/declaration-files.html#export-1) commonjs 导出模块

##### export

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[15](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/15-export)：

// 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的

// 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'

}

};

##### 混用declare和export

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[16](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/16-declare-and-export)：

// 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 default

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[18](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/18-export-default)：

// 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[19](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/19-export-default-enum-error)：

// types/foo/index.d.ts

export default enum Directions {

// ERROR: Expression expected.

Up,

Down,

Left,

Right

}

上例中 export default enum 是错误的语法，需要使用 declare enum 定义出来，然后使用 export default 导出：

// types/foo/index.d.ts

declare enum Directions {

Up,

Down,

Left,

Right

}

export default Directions;

针对这种默认导出，我们一般会将导出语句放在整个声明文件的最前面[20](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/20-export-default-enum)：

// types/foo/index.d.ts

export default Directions;

declare enum Directions {

Up,

Down,

Left,

Right

}

##### Export =

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; //bar 为单个声明文件

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[21](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/21-export-equal)：

// types/foo/index.d.ts

export = foo;

declare function foo(): string;

declare namespace foo {

const bar: number;

}

需要注意的是，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](https://www.typescriptlang.org/docs/handbook/modules.html" \l "export--and-import--require)。

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。

##### 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[22](https://github.com/xcatliu/typescript-tutorial/tree/master/examples/declaration-files/22-export-as-namespace)：

// 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;

}

### 内置对象

JavaScript 中有很多[内置对象](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)，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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)。

而他们的定义文件，则在 [TypeScript 核心库的定义文件](https://github.com/Microsoft/TypeScript/tree/master/src/lib)中。

#### 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](https://github.com/Microsoft/TypeScript/tree/master/src/lib)中。

#### TypeScript核心库的定义文件

[TypeScript 核心库的定义文件](https://github.com/Microsoft/TypeScript/tree/master/src/lib)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。

#### 使用TypeScript 写Node.js

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

#### 参考

* [内置对象](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)
* [TypeScript 核心库的定义文件](https://github.com/Microsoft/TypeScript/tree/master/src/lib)

## 进阶

### 类型别名

类型别名用来给一个类型起个新名字。

#### 简单例子

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。

#### 参考

* [Advanced Types # Type Aliases](http://www.typescriptlang.org/docs/handbook/advanced-types.html#type-aliases)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Advanced%20Types.html#%E7%B1%BB%E5%9E%8B%E5%88%AB%E5%90%8D)）

### 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。

#### 简单例子

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。**

#### 参考

* [Advanced Types # Type Aliases](http://www.typescriptlang.org/docs/handbook/advanced-types.html#string-literal-types)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Advanced%20Types.html#%E5%AD%97%E7%AC%A6%E4%B8%B2%E5%AD%97%E9%9D%A2%E9%87%8F%E7%B1%BB%E5%9E%8B)）

### 元组

数组合并了相同类型的对象，而元组（Tuple）合并了不同类型的对象。

元组起源于函数编程语言（如 F#），这些语言中会频繁使用元组。

#### 简单例子

定义一对值分别为 string 和 number 的元组：

let tom: [string, number] = ['Tom', 25];

当赋值或访问一个已知索引的元素时，会得到正确的类型：

let tom: [string, number];

tom[0] = 'Tom';

tom[1] = 25;

tom[0].slice(1);

tom[1].toFixed(2);

也可以只赋值其中一项：

let tom: [string, number];

tom[0] = 'Tom';

但是当直接对元组类型的变量进行初始化或者赋值的时候，需要提供所有元组类型中指定的项。

let tom: [string, number];

tom = ['Tom', 25];

let tom: [string, number];

tom = ['Tom'];

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

#### 越界的元素[§](https://ts.xcatliu.com/advanced/tuple.html#%E8%B6%8A%E7%95%8C%E7%9A%84%E5%85%83%E7%B4%A0)

当添加越界的元素时，它的类型会被限制为元组中每个类型的联合类型：

let tom: [string, number];

tom = ['Tom', 25];

tom.push('male');

tom.push(true);

// Argument of type 'true' is not assignable to parameter of type 'string | number'.

#### 参考

* [Basic Types # Tuple](http://www.typescriptlang.org/docs/handbook/basic-types.html#tuple)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Basic%20Types.html#%E5%85%83%E7%BB%84-tuple)）

### 枚举

枚举（Enum）类型用于取值被限定在一定范围内的场景，比如一周只能有七天，颜色限定为红绿蓝等。

#### 简单的例子

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 = {}));

所以使用的时候需要注意，最好不要出现这种覆盖的情况。

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：

#### 常数项和计算所得项

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

前面我们所举的例子都是常数项，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.

下面是常数项和计算所得项的完整定义，部分引用自[中文手册 - 枚举](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Enums.html)：

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#](https://msdn.microsoft.com/zh-cn/library/sbbt4032.aspx)。

#### 参考

* [Enums](http://www.typescriptlang.org/docs/handbook/enums.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Enums.html)）
* [C# Enum](https://msdn.microsoft.com/zh-cn/library/sbbt4032.aspx)

### 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。

这一节主要介绍类的用法，下一节再介绍如何定义类的类型。

#### 类的概念

虽然 JavaScript 中有类的概念，但是可能大多数 JavaScript 程序员并不是非常熟悉类，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

下面我们先回顾一下 ES6 中类的用法，更详细的介绍可以参考 [ECMAScript 6 入门 - Class](http://es6.ruanyifeng.com/#docs/class)。

##### 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

##### 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

ES7 中有一些关于类的提案，TypeScript 也实现了它们，这里做一个简单的介绍。

##### 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

上面的例子中，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); // Jack

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'.

需要注意的是，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。

什么是抽象类？

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'.

上面的例子中，我们定义了一个抽象类 Animal，并且定义了一个抽象方法 sayHi。在实例化抽象类的时候报错了。

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'.

上面的例子中，我们定义了一个类 Cat 继承了抽象类 Animal，但是没有实现抽象方法 sayHi，所以编译报错了。

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');

上面的例子中，我们实现了抽象方法 sayHi，编译通过了。

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

#### 参考

* [Classes](http://www.typescriptlang.org/docs/handbook/classes.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Classes.html)）
* [ECMAScript 6 入门 - Class](http://es6.ruanyifeng.com/#docs/class)

### 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。

#### 简单的例子

首先，我们来实现一个函数 createArray，它可以创建一个指定长度的数组，同时将每一项都填充一个默认值：

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

let result = [];

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

result[i] = value;

}

return result;

}

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

上例中，我们使用了[之前提到过的数组泛型](https://ts.xcatliu.com/basics/type-of-array.html#%E6%95%B0%E7%BB%84%E6%B3%9B%E5%9E%8B)来定义返回值的类型。

这段代码编译不会报错，但是一个显而易见的缺陷是，它并没有准确的定义返回值的类型：

Array<any> 允许数组的每一项都为任意类型。但是我们预期的是，数组中每一项都应该是输入的 value 的类型。

这时候，泛型就派上用场了：

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']

#### 多个类型的参数

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

[之前学习过](https://ts.xcatliu.com/basics/type-of-function.html#%E6%8E%A5%E5%8F%A3%E4%B8%AD%E5%87%BD%E6%95%B0%E7%9A%84%E5%AE%9A%E4%B9%89)，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，我们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;

}

#### 参考

* [Generics](http://www.typescriptlang.org/docs/handbook/generics.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/generics.html)）
* [Generic parameter defaults](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-3.html#generic-parameter-defaults)

### 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](https://ts.xcatliu.com/basics/type-of-function.html#%E9%87%8D%E8%BD%BD)，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

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。

#### 参考

* [Declaration Merging](http://www.typescriptlang.org/docs/handbook/declaration-merging.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Declaration%20Merging.html)）

### 扩展阅读

此处记录了[官方手册](http://www.typescriptlang.org/docs/handbook/basic-types.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/)）中包含，但是本书未涉及的概念。

我认为它们是一些不重要或者不属于 TypeScript 的概念，所以这里只给出一个简单的释义，详细内容可以点击链接深入理解。

* [Never](http://www.typescriptlang.org/docs/handbook/basic-types.html#never)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Basic%20Types.html#never)）：永远不存在值的类型，一般用于错误处理函数
* [Variable Declarations](http://www.typescriptlang.org/docs/handbook/variable-declarations.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Variable%20Declarations.html)）：使用 let 和 const 替代 var，这是 [ES6 的知识](http://es6.ruanyifeng.com/#docs/let)
* [this](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Functions.html#this)：箭头函数的运用，这是 [ES6 的知识](http://es6.ruanyifeng.com/#docs/function)
* [Using Class Types in Generics](http://www.typescriptlang.org/docs/handbook/generics.html#using-class-types-in-generics)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Generics.html#%E5%9C%A8%E6%B3%9B%E5%9E%8B%E9%87%8C%E4%BD%BF%E7%94%A8%E7%B1%BB%E7%B1%BB%E5%9E%8B)）：创建工厂函数时，需要引用构造函数的类类型
* [Best common type](http://www.typescriptlang.org/docs/handbook/type-inference.html#best-common-type)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Type%20Inference.html#%E6%9C%80%E4%BD%B3%E9%80%9A%E7%94%A8%E7%B1%BB%E5%9E%8B)）：数组的类型推论
* [Contextual Type](http://www.typescriptlang.org/docs/handbook/type-inference.html#contextual-type)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Type%20Inference.html#%E4%B8%8A%E4%B8%8B%E6%96%87%E7%B1%BB%E5%9E%8B)）：函数输入的类型推论
* [Type Compatibility](http://www.typescriptlang.org/docs/handbook/type-compatibility.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Type%20Compatibility.html)）：允许不严格符合类型，只需要在一定规则下兼容即可
* [Advanced Types](http://www.typescriptlang.org/docs/handbook/advanced-types.html#intersection-types)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Advanced%20Types.html#%E4%BA%A4%E5%8F%89%E7%B1%BB%E5%9E%8B%EF%BC%88intersection-types%EF%BC%89)）：使用 & 将多种类型的共有部分叠加成一种类型
* [Type Guards and Differentiating Types](http://www.typescriptlang.org/docs/handbook/advanced-types.html#type-guards-and-differentiating-types)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Advanced%20Types.html#%E7%B1%BB%E5%9E%8B%E4%BF%9D%E6%8A%A4%E4%B8%8E%E5%8C%BA%E5%88%86%E7%B1%BB%E5%9E%8B%EF%BC%88type-guards-and-differentiating-types%EF%BC%89)）：联合类型在一些情况下被识别为特定的类型
* [Discriminated Unions](http://www.typescriptlang.org/docs/handbook/advanced-types.html#discriminated-unions)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Advanced%20Types.html#%E5%8F%AF%E8%BE%A8%E8%AF%86%E8%81%94%E5%90%88%EF%BC%88discriminated-unions%EF%BC%89)）：使用 | 联合多个接口的时候，通过一个共有的属性形成可辨识联合
* [Polymorphic this types](http://www.typescriptlang.org/docs/handbook/advanced-types.html#polymorphic-this-types)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Advanced%20Types.html#%E5%A4%9A%E6%80%81%E7%9A%84this%E7%B1%BB%E5%9E%8B)）：父类的某个方法返回 this，当子类继承父类后，子类的实例调用此方法，返回的 this 能够被 TypeScript 正确的识别为子类的实例。
* [Symbols](http://www.typescriptlang.org/docs/handbook/symbols.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Symbols.html)）：新原生类型，这是 [ES6 的知识](http://es6.ruanyifeng.com/#docs/symbol)
* [Iterators and Generators](http://www.typescriptlang.org/docs/handbook/iterators-and-generators.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Iterators%20and%20Generators.html)）：迭代器，这是 [ES6 的知识](http://es6.ruanyifeng.com/#docs/iterator)
* [Namespaces](http://www.typescriptlang.org/docs/handbook/namespaces.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Namespaces.html)）：避免全局污染，现在已被 [ES6 Module](http://es6.ruanyifeng.com/#docs/module) 替代
* [Decorators](http://www.typescriptlang.org/docs/handbook/decorators.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Decorators.html)）：修饰器，这是 [ES7 的一个提案](http://es6.ruanyifeng.com/#docs/decorator)
* [Mixins](http://www.typescriptlang.org/docs/handbook/mixins.html)（[中文版](https://zhongsp.gitbooks.io/typescript-handbook/content/doc/handbook/Mixins.html)）：一种编程模式，与 TypeScript 没有直接关系，可以参考 [ES6 中 Mixin 模式的实现](http://es6.ruanyifeng.com/#docs/class#Mixin%E6%A8%A1%E5%BC%8F%E7%9A%84%E5%AE%9E%E7%8E%B0)

## 工程

掌握了 TypeScript 的语法就像学会了砌墙的工艺。

我们学习 TypeScript 的目的不是为了造一间小茅屋，而是为了造高楼大厦，这也正是 TypeScript 的类型系统带来的优势。

那么一项大工程应该如何开展呢？本部分的内容就会介绍 TypeScript 工程化的最佳实践，具体内容包括：

* [代码检查](https://ts.xcatliu.com/engineering/lint.html)
* [编译选项](https://ts.xcatliu.com/engineering/compiler-options.html)